Design and implementation of a configurable UART peripheral device for RISC-V Microprocessors

University of Ioannina

Agamemnonas Kyriazis

Efthymiou Aristides, Assistant Professor

September 2023

Dedication

To Mileopo and its people.

Acknowledgements

I want to thank my advisor Efthymiou Aristides, as well as my family Ioannis Kyriazis, Sophia Papadopoulou and Athena Kyriazis, for supporting me throughout my studies. Additionally I wish to extend my gratitude to my friends who endured my frequent complaining during the waveform debugging stage.

Abstract

Agamemnonas Kyriazis , Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, September 2023.
Design and implementation of a configurable UART peripheral device for RISC-V Microprocessors.
Advisor: Efthymiou Aristides, Assistant Professor.
Summary:

This thesis focuses on the design and implementation of an advanced UART module with fully configurable parameters. Both hardware and software aspects were studied for the extremely popular RISC-V architecture.

We begin our project by exploring the fundamentals of UART communication and analyzing the architecture of the suggested design. During this stage we examine all components in hierarchical order and evaluate their purpose and role in the whole circuit.

After successfully analyzing the architecture we shift our focus onto the microcode of the module, the assembly instructions and finally the C routines developed to provide full control of the previously discussed circuit. In order to demonstrate our module's capabilities we integrated it to a complete design formally known as IBEX demo system.

Finally, the thesis investigates the integration of asynchronous FIFO queues to synchronize data flow in multi-clock domain systems.

Through functional verification and code coverage analysis, the reliability and effectiveness of the module are evaluated.

Overall, this thesis offers a comprehensive exploration of our UART module, contributing to the advancement of reliable serial communication in modern embedded systems.

Περίληψη

Αυτή η εργασία εστιάζει στο σχεδιασμό, την υλοποίηση, την προσομοίωση και την ενσωμάτωση ενός σύγχρονου κυκλώματος για το πρωτόκολλο UART με πλήρως διαμορφώσιμες παραμέτρους, μελετώντας το τόσο από την πλευρά του υλικού όσο και του λογισμικού, για την εξαιρετικά δημοφιλή αρχιτεκτονική RISC-V.

Αρχικά ξεκινάμε την μελέτη μας εξερευνώντας τις βασικές αρχές του πρωτοκόλλου επικοινωνίας UART και αναλύοντας την αρχιτεκτονική του σχεδίου μας. Κατά τη διάρκεια αυτού το κεφαλαίου εξετάζουμε όλες τις μονάδες σε ιεραρχική σειρά και αξιολογούμε το ρόλο που εκπληρώνουν σε ολόκληρο το κύκλωμα.

Μετά την ανάλυση του σχεδιασμού, μετατοπίζουμε την προσοχή μας στον μικροκώδικα της μονάδας και στις ρουτίνες C που αναπτύχθηκαν για τον έλεγχο του κυκλώματος. Προκειμένου να παρουσιάσουμε τις δυνατότητες της υλοποίησης μας, ενσωματώσαμε το κύκλωμα σε ένα ήδη έτοιμο σύστημα που περιλαμβάνει μια κεντρική μονάδα επεξεργασίας, μνήμη και άλλα περιφεριακά καθώς και μια μονάδα UART την οποία αντικαταστήσαμε με την δική μας, το IBEX demo system.

Στη συνέχεια, η εργασία αυτή διερευνά την υλοποίηση ασύγχρονων ουρών FIFO (πρώτο μέσα, πρώτο έξω) με σκοπό την δυνατότητα ενσωμάτωσης του σχεδίου μας σε συστήματα με πολλαπλά clock domains, λειτουργία η οποία επίσης έλειπε από το κύκλωμα του IBEX demo system.

Κλείνοντας με τη δια της προσομοίωσης ανάλυση και της code coverage αξιολόγησης των ρουτινών, επαληθεύεται η λειτουργικότητα και η αποτελεσματικότητα του κυκλώματος μας. Χρησιμοποιώντας εργαλεία όπως το Vivado της Xilinx μπορέσαμε να μετρήσουμε την επιφάνεια που καταλαμβάνει στο FPGA arty7a100 το σχέδιο μας και να υπολογίσουμε την μέγιστη ασφαλή συχνότητα λειτουργίας του. Και στις δύο μετρικές το σχέδιο μας φάνηκε να υπερέχει αυτού του IBEX demo system δεδομένων των λειτουργιών που προσθέσαμε.

Contents

Listings

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General

In today's digital-driven era, it is impossible to imagine a microprocessor that does not include a Universal Asynchronous Receiver/Transmitter (UART) component. Whether implemented in software or, in our case, at the hardware level. The UART is an essential element for enabling communication between microprocessors and external devices.

This thesis focuses on the design and implementation of a configurable UART peripheral device. Even though the module can also be operated as a standalone the data exchange protocol between our module and the host was specifically tailored for the IBEX RISC-V microprocessor, aiming to contribute to this popular architecture.

The rise of the RISC-V architecture has garnered significant attention due to its open-source nature, modular design, and scalability. It provides a versatile platform for developing embedded systems across various applications such as Internet of Things (IoT) devices. The UART peripheral serves as a basic communication interface, enabling data exchange between microprocessors and external devices such as sensors and other electronic systems, which modern embedded devices fundamentally rely on.

1.2 Design Implementation

The RTL (Register Transfer Level) implementation of the UART design was carried out entirely using Verilog Hardware Description Language (HDL).

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction.

1.3 Functional Verification

To verify the correctness and functionality of the RTL design, both Icarus Verilog, ModelSim and GTK-Wave software tools were utilized.

Icarus Verilog is a free compiler implementation for the IEEE-1364 Verilog hardware description language. Icarus is maintained by Stephen Williams and it is released under the GNU GPL license.

Both Modelsim and GTK-Wave software are widely used simulation and debugging environments. By simulating the RTL design, it was possible to observe and analyze the behavior of the UART at various levels of abstraction. This allowed for effective debugging of the module whilst moving on to the physical implementation.

1.4 Design Integration

The UART design was integrated on the Arty A7100 Xilinx FPGA development board, both as a standalone peripheral and as an IBEX peripheral.

Integrating the UART design as a standalone peripheral on the FPGA board enabled testing and validating the module's functionality in isolation, ensuring it operates as expected.

Furthermore, the UART design was integrated as an IBEX peripheral. The IBEX 32bit RISC-V microprocessor, previously introduced in the thesis, served as the host for the UART peripheral. Integrating the UART as an IBEX peripheral expands the capabilities of the microprocessor and provides a robust solution for data exchange with external devices.

1.5 FPGA-Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. CLBs themselves consist of flip-flops ,Look up Tables and multiplexers.

FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves.

The main advantage of FPGAs is that they offer a drastic increase of speed and efficiency in a wide range of applications. On the other hand FPGAs do not retain their configuration if the power supply is interrupted.

1.6 Vivado

Vivado is a suite of software tools developed by Xilinx, a leading provider of programmable logic devices (PLDs) and field-programmable gate arrays (FPGAs). It is a comprehensive design environment specifically designed for FPGA and SoC (System-on-Chip) development.

Vivado provides a wide range of capabilities and features to support the entire FPGA design flow, from design entry and synthesis to implementation, verification, and programming of the target FPGA device. It offers an intuitive graphical user interface (GUI) as well as a command-line interface (CLI) for design management and configuration.

Chapter 2

Fundamentals of the Universal Asynchronous Receiver Transmitter Protocol

2.1 Description

UART stands for universal asynchronous receiver / transmitter and defines a protocol, or set of rules, for exchanging serial data between two devices. UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions. Both ends also share a ground connection. [6]

Communication in UART can be simplex (data is sent in one direction only), half-duplex (each side transmits but only one at a time), or full-duplex (both sides can transmit simultaneously). Data in UART is transmitted in the form of frames. The format and content of these frames is briefly described and explained. [8]

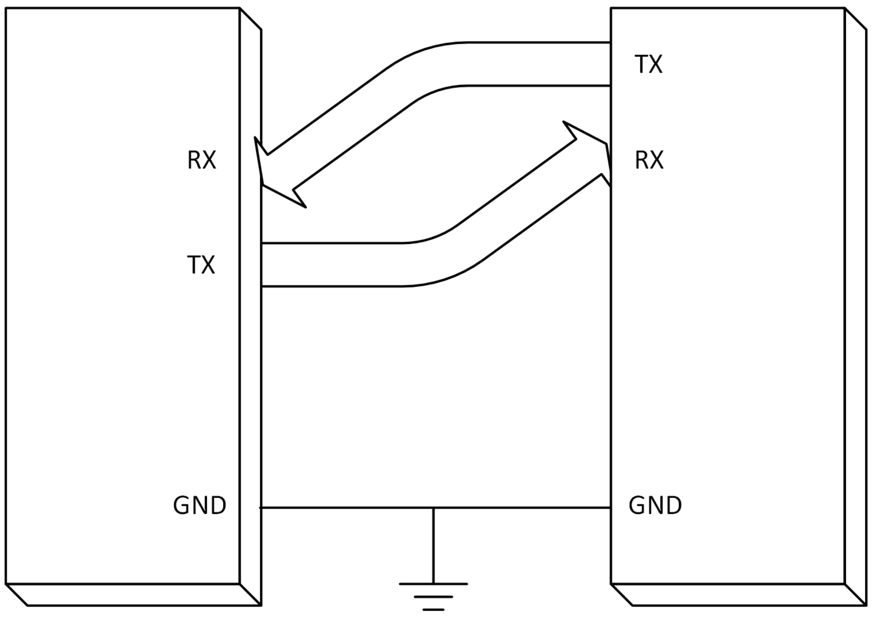


Figure 2.1: UART Block Diagram

2.2 Purpose

UART serves the purpose of transmitting data bits individually, one at a time. Initially, UART was commonly used to connect teletypewriters to operator consoles, representing one of the earliest forms of computer communication devices.

Moreover, UART played a significant role as early hardware systems for facilitating internet connectivity.

A circuit that implements the UART protocol acts as a bridge between digital devices, enabling the simple yet effective exchange of data through serial communication. By implementing a UART module, we unlock the ability to establish reliable and efficient communication between various electronic components, such as microcontrollers, sensors, and peripheral devices.

2.3 Operation

The operation of a UART module involves taking packets of data and transmitting the individual bits in a sequential manner. Upon reaching the destination, a second UART circuit reconstructs the bits into complete frames, reassembling the original data.

While the frame data size is typically 8 bits, it can also vary and be extended to 9 bits or even reduced to 6, or 7, allowing for greater flexibility in data representation.

2.4 UART in Modern Communication: Simple, Reliable, and Essential

In recent years, the popularity of UART has decreased: protocols like SPI and I2C have been replacing UART between chips and components.

Instead of communicating over a serial port, most modern computers and peripherals now use technologies like Ethernet and USB. However, UART is still used for lower-speed and lower-throughput applications, because it is very simple, low-cost and easy to implement.

By providing a reliable and efficient means of data transfer, UART has become a cornerstone in the field of digital communication, facilitating seamless interaction between microprocessors and external devices in a wide range of applications. [2]

2.5 Protocol Description

A UART frame has the following structure:

UART frame, field length in bits			
1	6-9	0-1	1-2
Start bit	Payload	Parity bits	Stop bits

Table 2.1: UART Frame Format

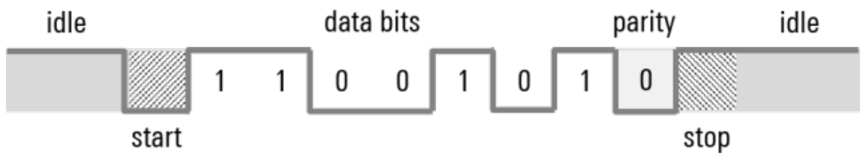


Figure 2.2: UART Protocol Wave Example

The UART protocol requires consistent settings on both the transmitting and receiving sides for successful communication.

- Start Bit: The transmission starts with a single start bit, which is always low (0). It serves as a synchronization signal to alert the receiver that a new frame has just started being transmitted.
- Payload: The payload contains the actual data being transmitted, ranging from 6 to 9 bits in length. The data bits are transmitted sequentially, with the least significant bit (LSB) sent first.
- Parity Bits: Optionally, 0 to 1 parity bits can be included for error detection purposes. Parity can be set to no parity, even parity, or odd parity. The parity bits are calculated based on the data bits and are used by the receiver to verify the integrity of the received data.
- Stop Bits: The frame ends with 1 to 2 stop bits, usually high (1). Stop bits provide a period of time for the receiver to prepare for the next frame and ensure accurate reception. The most common configuration is 1 stop bit.

! It is crucial to configure both the transmitter and receiver with the same settings, including the number of data bits, parity type, and number of stop bits, to ensure proper communication.

The 8N1 configuration, with 8 data bits, no parity, and 1 stop bit, is widely used and offers a good balance between simplicity and reliability.

Chapter 3

Design and Requirements

3.1 Top Module Description

Beginning with an abstract description of the whole circuit. The module consists of five core components. These components include a receiver circuit, a transmitter circuit, FIFO queues for data storage, baud rate signal generator and finally some addressable registers for storing the protocol's parameters.

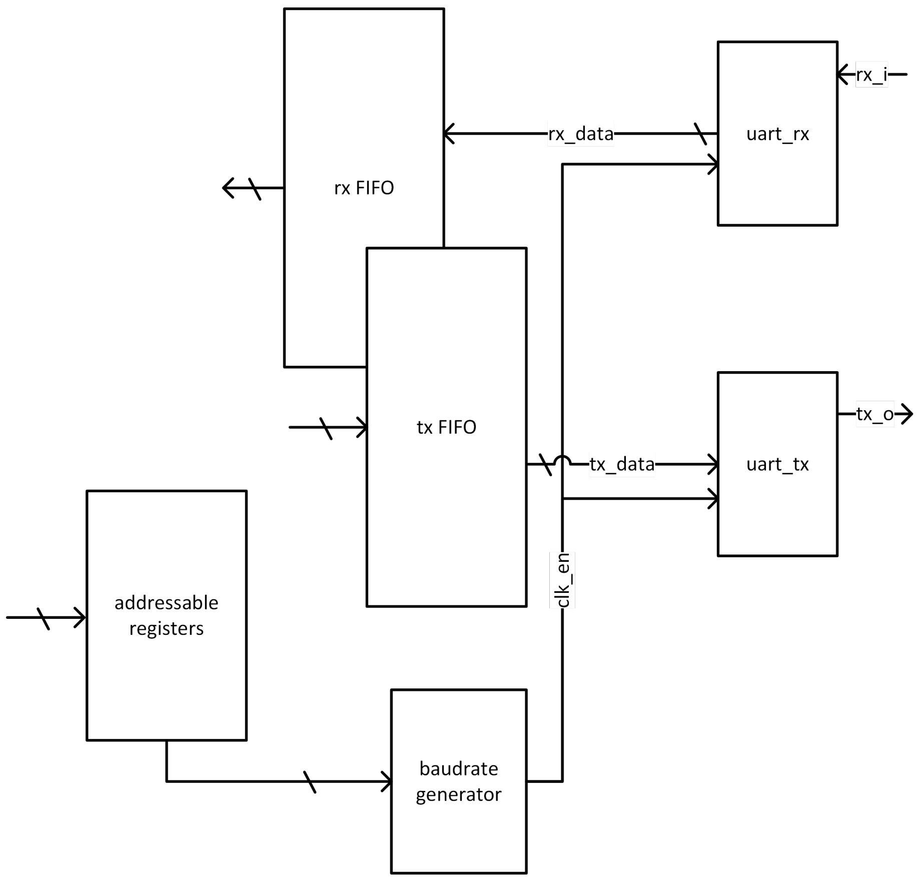


Figure 3.1: UART Block Diagram

1. Receiver Circuit: The receiver circuit is responsible for receiving incoming data frames transmitted over the UART protocol. In our occasion it consists of a shift register and a state machine controller. The shift register captures and stores the received bits, while the state machine controls the transitions between different states to ensure proper data reception.

2. Transmitter Circuit: The transmitter circuit in charge of transmitting data frames to the receiving end of another device. Similar to the receiver circuit, it consist of a shift register and a state machine for ensuring protocol accurate and therefore reliable transmission of the data.

3. FIFO Queues: The UART module incorporates two FIFO (First-In-First-Out) queues for temporary data storage. One FIFO queue is dedicated to the receiver, storing incoming data frames until they can be processed by the microprocessor. The other FIFO queue is assigned to transmitter, holding the data to be transmitted until it can be sent over the UART communication line. The FIFO queues ensure a smooth flow of data, allowing the receiver and transmitter to operate independently at their own respective speeds.

4. Baud rate Generator: This circuit is responsible for generating a clock enable signal with a frequency corresponding to the baud rate specified in the protocol's parameters. The Baud Rate Generator ensures that the transmitter and receiver operate at the same communication speed. The Baud Rate Generator circuit consists of two counters: one for the receiver clock enable and one for the transmitter clock enable.

5. Addressable Memory for Protocol Parameters: The module includes addressable registers for storing various protocol parameters. These parameters include settings such as the number of data bits, parity type, number of stop bits, and baud rate. The addressable memory enables easy access and modification of these parameters, allowing for flexibility in adapting the UART module to different communication requirements.

3.2 Receiver

The receiver circuit for UART operates as a shift register and is accompanied by a state machine that controls its operation. A parity checking circuit is also included for ensuring the received data are error free.

3.2.1 Receiver State Machine

The UART receiver state machine consists of four primary states:

1. IDLE State:

    - In the IDLE state, the receiver's RX line is held high (1) by the transmitter circuit of the other device. This state indicates that no data is currently being transmitted.

- Also in the IDLE state the module initializes the counters used by the state machine according to the protocol's parameters. Such counters include the data_counter which counts the number of data bits in a frame and the stop_counter which is used to add the appropriate delay before receiving the next frame.

2. DATA State:

    - When the receiver transitions to the DATA state, it implies that the transmitter of the other device is actively sending bits on the RX line of our module. These received bits are stored in the data shift register of the UART receiver module.
    - The receiver continuously samples the RX line at specific intervals defined by the baud rate of the protocol alongside a clock enable signal to capture the individual bits being transmitted by the transmitter.
    - With each clock tick in this state the data_counter decreases by one and a new bit is being shifted inside the shift register.

3. PARITY State:

    - In the PARITY state our receiver expects, if enabled, to receive a parity bit corresponding to the transmitted data. The UART module supports different parity options, including even, odd, or no parity (0 parity bits).
    - If parity checking is enabled, the receiver verifies the received data bits against the received parity bit to detect any transmission errors or data corruption.
    - In case the parity check process suggests there is an error in the received data the module raises an rx_err signal flag in order to notify the controller module that the frame is corrupted and thus discard it.

4. STOP State:

    - In the STOP state, the receiver reads 1 to 2 stop bits from the RX line. These stop bits provide a brief pause between successive data frames, ensuring proper synchronization between the transmitter and receiver.
    - The stop bits also allow the receiver to prepare for receiving the next frame of data.
    - The stop_counter suggests how many stop bits must be received before switching back to the IDLE state.
    - After receiving the set amount of stop bits the module raises the rx_rdy signal flag to notify the control module that a frame was received successfully.

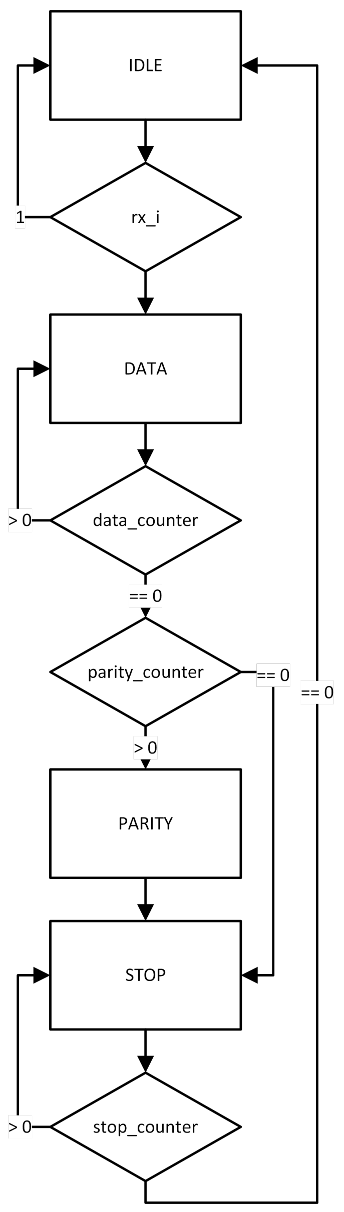


Figure 3.2: UART Receiver Flow Diagram

3.2.2 Protocol Parameters and State Machine Controller

The state machine controller takes into account these protocol parameters to determine the appropriate state transitions and actions to be performed. Let's explore how these parameters influence the state transitions:

1. Payload Size:

   - The payload size, determined by the number of data bits, affects the duration of the DATA state. The state machine controller needs to ensure that the correct number of data bits are received and stored in the shift register before transitioning to the next state.
   - Depending on the payload size, the state machine controller will need to wait for a specific number of bits to be received before proceeding to the PARITY state or directly transitioning to the STOP state.

2. Parity:

    - If parity checking is enabled, the state machine controller incorporates the parity setting to determine whether the received data is valid or contains errors. The PARITY state is entered when the receiver expects to receive a parity bit.
    - The state machine controller compares the received parity bit with the calculated parity based on the received data bits. If they match, the received data is considered valid. Otherwise at the end of the whole process an error flag will be raised.

3. Number of Stop Bits:

    - The number of stop bits impacts the duration of the STOP state. The state machine controller waits for the specified number of stop bits to be received before transitioning back to the IDLE state.

By considering the payload size, parity, and number of stop bits, the state machine controller orchestrates the transitions between the IDLE, DATA, PARITY, and STOP states.

This coordination ensures that the receiver accurately processes the incoming data according to the specified UART protocol parameters, enabling reliable data communication between devices.

3.2.3 Receiver Module RTL Definition

```
module uart_rx (
    input wire clk_i,
    input wire clk_en_i,
    input wire rst_ni,
    input wire en,
    input wire rx_i,
    input wire [3:0] data_size_i,
    input wire parity_size_i,
    input wire parity_type_i,
    input wire [1:0] stop_size_i,
    output wire [8:0] data_o,
    output wire rx_rdy_o,
    output wire rx_err_o,
    output wire [1:0] rx_state_o
);
```

Listing 3.1: Verilog Implementation of Receiver Module

3.2.4 Signals

The module utilizes several internal signals that play a vital role in determining state transitions and controlling flag behavior. These signals are:

- [3:0] data_counter, counts data bits received
- [3:0] data_size, protocols size of data packet

- parity_counter, counts parity bits received
- parity_size, protocols size of parity
- [1:0] stop_counter, counts number of stop bits received
- [8:0] data_buf, shift register for received data
- parity_buf, parity bit buffer
- parity_type, protocols parity type (Even/Odd)
- [1:0] stop_buf, stop bits buffer
- [1:0] state, synchronous state register (on positive edge)
- [1:0] next_state, combinational logic to determine next state
- reg [8:0] data_d, combinational logic for formatting the data
- rx_rdy_d, received a new packet (flag)
- rx_err_d, corruption detected (flag)

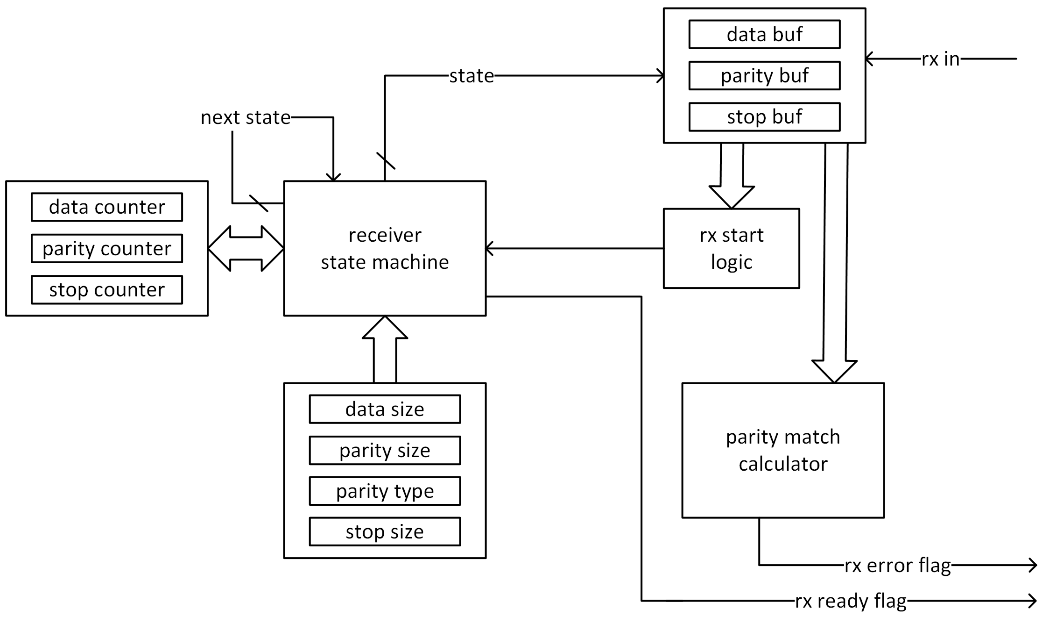


Figure 3.3: UART Receiver Block Diagram

3.3 Transmitter

Same as the receiver module, the transmitter circuit operates as a shift register and is accompanied by a state machine that controls its operation.

3.3.1 Transmitter State Machine

The UART transmitter state machine consists of just two states

1. IDLE State:

    - In the IDLE state the transmitter's TX line is held high (1). This indicates to the receiver on the other end that no data is currently being transmitter.
    - While on the IDLE state, when a new data frame needs to be sent, the module assembles it accordingly based on the current protocol's parameters before loading it to the shift register.
    - In order to determine the parity bit for the frame, a dedicated parity calculation circuit is employed. This circuit performs a XOR or NOT(XOR) operation on the data bits, depending on the selected parity type. If the None parity option is chosen, the parity calculation circuit replaces it with a logic 1.

2. WRITE State:

    - In the WRITE state the transmitter operates solely as a shift register. The assembled frame is transmitted bit-by-bit on the TX line of the module, starting with the least significant bit (LSB) first.
    - A counter is being used to to keep track of the progress of the transmission. It signals the end of the frame once all bits have been transmitted, triggering a transition back to the IDLE state.

3.3.2 Transmission Process of the UART Transmitter

The UART transmitter generates a start bit to mark the beginning of the character transmission.

It then proceeds to shift out the necessary data bits from the shift register onto the communication line. If parity checking is enabled, the UART generates and transmits the parity bit based on the selected parity type.

Finally, the transmitter sends the stop bits to signal the end of the character transmission. The UART transmitter operates efficiently, ensuring reliable and effective data transmission.

This simplicity contributes to the overall robustness of the UART transmitter circuit.

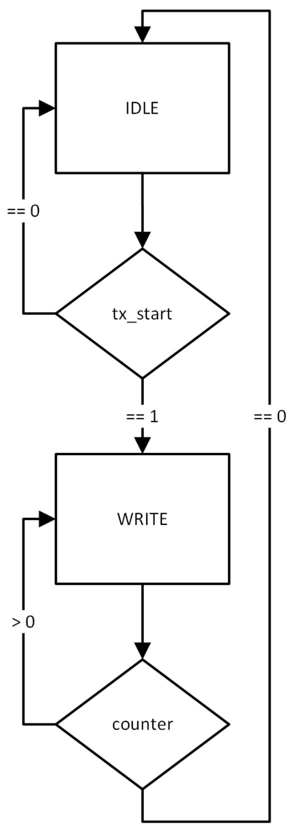


Figure 3.4: UART Transmitter Flow Diagram

3.3.3 Transmitter Module RTL Definition

```
module uart_tx (
    input wire clk_i,
    input wire clk_en_i,
    input wire rst_ni,
    input wire en,
    input wire tx_start_i,
    input wire [3:0] data_size_i,
    input wire parity_size_i,
    input wire parity_type_i,
    input wire [1:0] stop_size_i,
    input wire [8:0] data_i,
    output wire tx_o,
    output wire tx_rdy_o,
    output wire tx_state_o
);
```

Listing 3.2: Verilog Implementation of Transmitter Module

3.3.4 Signals

The transmitter module utilizes the following internal signals:

- [3:0] frame_counter, total number of bits to be transmitted
- [12:0] frame_buffer, shift register for transmitting data on the serial line

- state, synchronous state register
- next_state, combinational logic to determine next state
- there is also an unnamed signal which is used to calculate the parity bit to be transmitted.

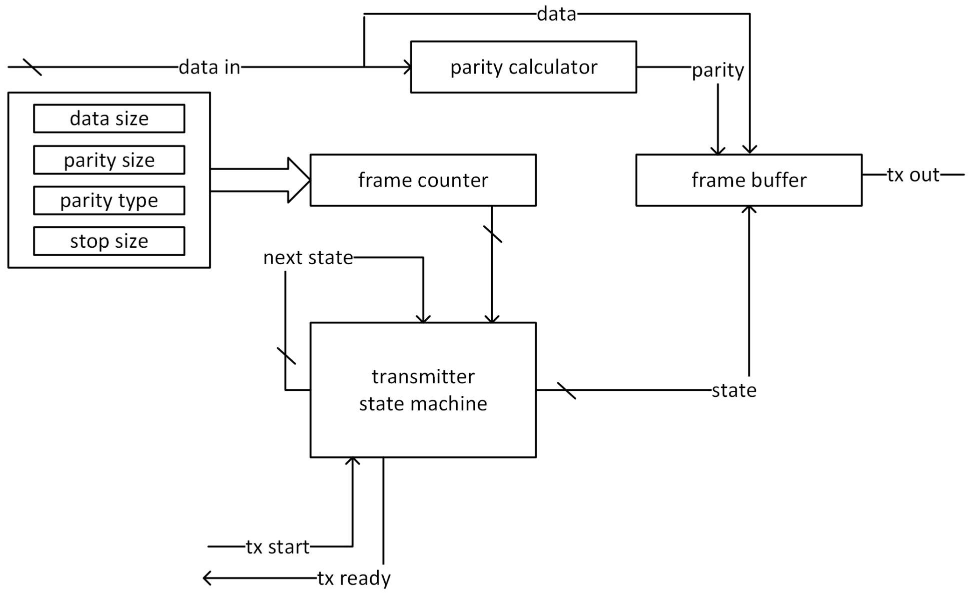


Figure 3.5: UART Transmitter Block Diagram

3.3.5 Parity Calculation

The parity bit of the frame is calculated during the IDLE state of the transmitter using the following case statement in Verilog:

```
case(data_size_i)
    6 : begin
        frame_buffer <= {4'b1111, (~|parity_size_i)? 1'b1 :
        (parity_type_i)? ^data_i :
        ~^data_i, data_i[5:0], 2'b01};
    end
    7 : begin
        frame_buffer <= {3'b111, (~|parity_size_i)? 1'b1 :
        (parity_type_i)? ^data_i :
        ~^data_i, data_i[6:0], 2'b01};
    end
    8 : begin
        frame_buffer <= {2'b11, (~|parity_size_i)? 1'b1 :
        (parity_type_i)? ^data_i :
        ~^data_i, data_i[7:0], 2'b01};
    end
    default : begin
        frame_buffer <= {1'b1, (~|parity_size_i)? 1'b1 :
        (parity_type_i)? ^data_i :
        ~^data_i, data_i, 2'b01};
    end
endcase
```

Listing 3.3: Verilog Implementation of Parity Calculation Logic

This Verilog implementation demonstrates the calculation of the parity bit based on the parity size (parity_size_i), and the parity type (parity_type_i). The resulting value is stored in the frame_buffer, which represents the shift register for transmitting data on the serial line.

3.3.6 Frame Assembly

In the IDLE state of the transmitter, the frame to be transmitted is assembled following a predefined format.

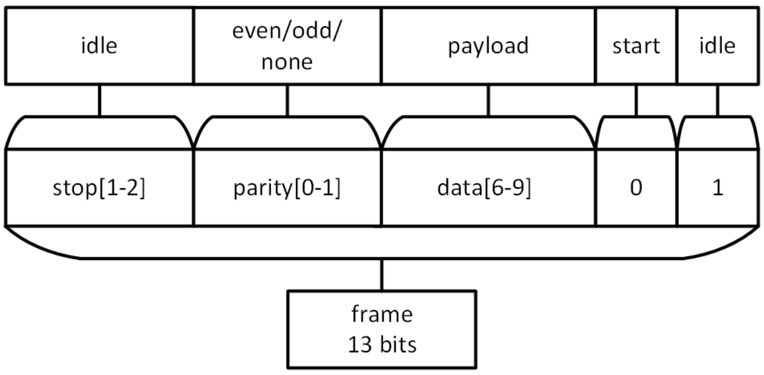


Figure 3.6: Frame Format

3.4 Controller

The controller serves as the central control unit of the UART module, acting as an interface that coordinates the data flow between the receiver and transmitter circuits with the connected microprocessor.

When the controller receives a memory request from the microprocessor, whether it is a request to fetch data from or store data into memory, it takes on the responsibility of translating these operations into corresponding local UART commands. These commands are specific instructions that govern the operation of the UART module and facilitate the reliable transfer of data. We will analyze these operations later on.

So far the supported features of the UART controller are as follows,

- Read received data.
- Write data to be transmitted.
- Read the state of the UART module.
- Write UART enable signals.
- Write UART protocol parameters.

- Utilizing FIFO queues for buffering and smooth data flow.
- Control the timing of data transmission and reception using clock enable and tick counter mechanisms.
- Handle requests and responses from the core.
- Manage internal signals and registers for state management and data flow control.
- Throw a *byte received* interrupt. Generates an interrupt signal when a byte of data is received by the UART module. This interrupt can be used to notify the core or processor about the availability of new data for processing.

3.4.1 Efficient Data Buffering through Synchronous FIFO Queues

The module houses two synchronous FIFO queues for data buffering and smooth data flow. The width of these queues is 9 bits because the size of the payload can vary from 6 to 9 bits. Additionally, it is noteworthy that, the depth of the queues can only be altered on synthesis time by using the RX/TX_FIFO_DEPTH parameter of the module.

Each FIFO queue is equipped with the *full* and *empty* signals, which provide the module with valuable information about the queue's status. The *full* signal indicates when the queue is full and cannot store additional packets, while the *empty* signal indicates that the queue currently holds no data.

Operations of a synchronous FIFO

1. Write Operation: The operation involves in writing or storing the data in to the FIFO memory till it rises the *full* flag for not to write anymore.
2. Read Operation: Read operation is performed when we want to get data out from the FIFO memory until it informs there is no more data to be read from the memory, using the *empty* flag.

Pointers and Structure

In our implementation of a synchronous FIFO queue, we employ two n-bit synchronous counters to track the positions for reading and writing data. The write pointer is incremented by one during a write operation, while the read pointer is incremented during a read operation. To determine the memory address for writing or reading data, we utilize the n-1 least significant bits from the respective counter.

The structure of our FIFO queue follows that of a circular buffer. This means that when a pointer reaches its maximum value, it wraps around to zero, allowing for continuous reuse of the same memory locations for read and write operations.

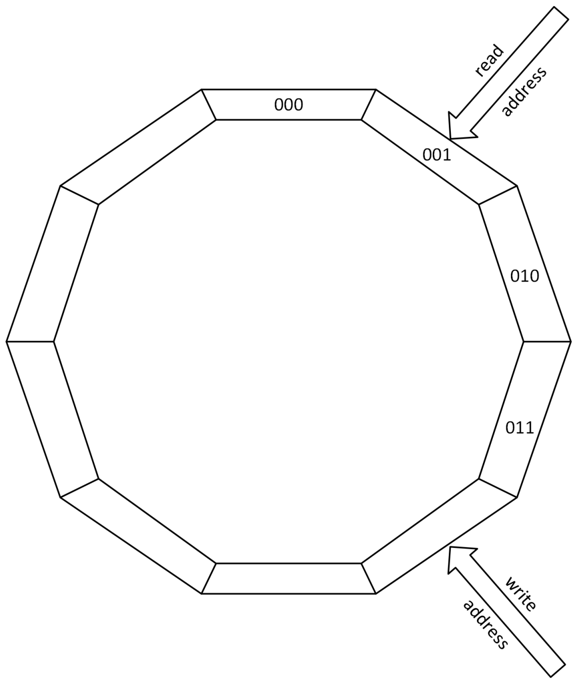


Figure 3.7: Circular Buffer

FIFO Depth and Address Space

The number of memory block available for the FIFO queue must strictly be a power of two for the flags to work seamlessly. The address space of the queue is determined using $\log_2 DEPTH$

Full Flag

The full flag of the FIFO queue is determined by comparing the read pointer with the write pointer after inverting the most significant bit of the write pointer. If these two signals have the same value, it indicates that the buffer is full, and the full flag is raised. When the full condition is true, no more write operations are allowed to execute.

To understand why the most significant bit of the write pointer is inverted during the comparison with the read pointer, let's consider an example using 3 bits.

Example. *Initially, both pointers are set to zero.*

During the comparison to calculate the full flag, we use the operands: $000 == ((\neg 0)00)$, which is false.

After a write operation, the write pointer increases by one. The operands become: $000 == ((\neg 0)01)$, which is false.

This process continues until the write pointer reaches 011. On the next write operation, the write pointer increases by one and becomes 100.

Assuming no read operations were performed, the compared operands are now: $000 == ((\neg 1)00)$, which is true. This indicates that the buffer is full.

Near full condition is calculated the same way except that the write pointer used for the comparison is (write pointer + 1).

Empty Flag

The calculation of the empty flag in the FIFO queue is straightforward.

It involves comparing the read pointer with the write pointer. If the values of these two counters are identical, it indicates that the buffer is empty, and the empty flag is raised. Subsequently, when the empty condition is true, it implies that no further read operations can be performed.

On start both counters are initialized to zero and the circular buffer is considered empty and thus the empty signal is raised.

The near empty signal is calculated comparing the write pointer with the (read pointer + 1).

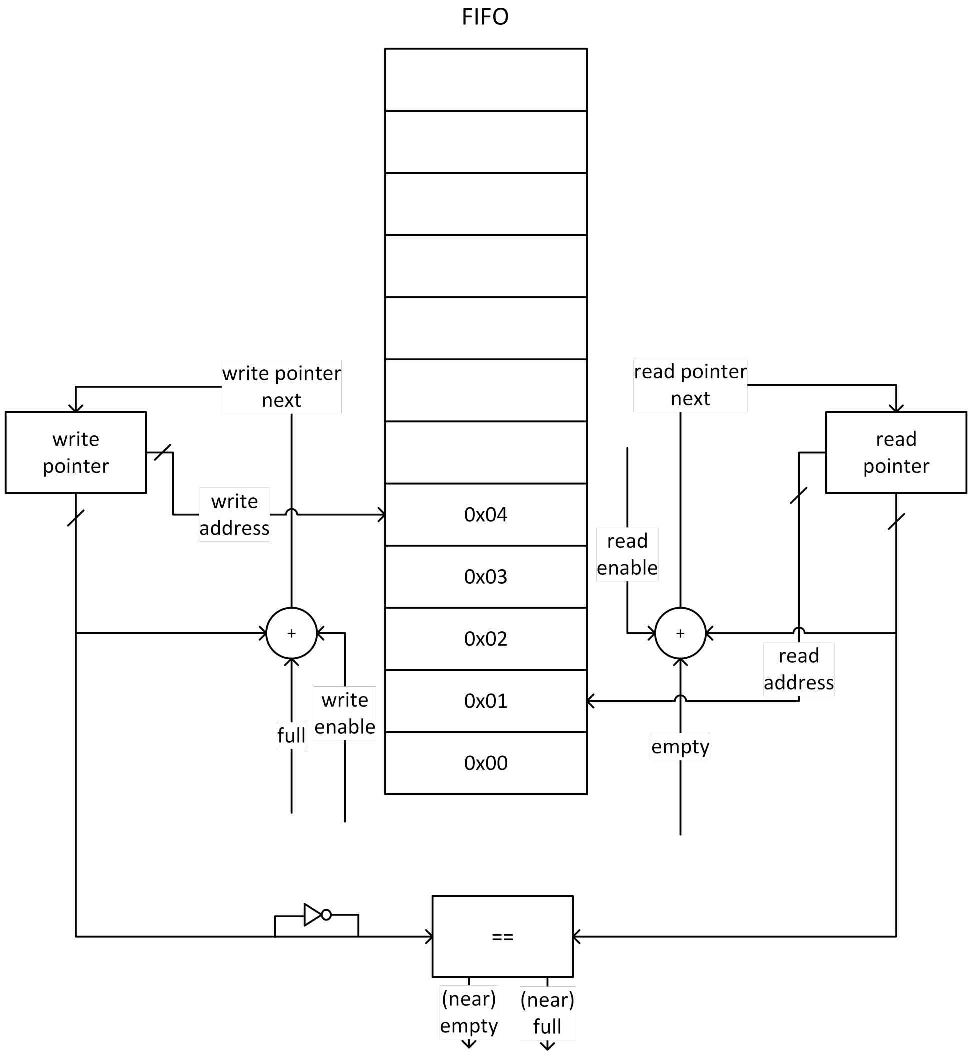


Figure 3.8: FIFO Block Diagram

Receiver FIFO

One FIFO queue is utilized by the receiver circuit to store received data until the processor reads them. A new data packet is being stored in the queue if the receiver circuit raises the rx_rdy flag and the rx_err flag is lowered.

Transmitter FIFO

The second queue serves as a buffer for packets intended to be transmitted by the transmitter circuit. This feature proves highly advantageous as the core operates at a significantly higher frequency compared to the UART module's capability to transmit data packets through the serial line.

The FIFO queue buffer ensures that the processor can execute multiple write operations on the peripheral without experiencing delays caused by the transmission of packets. Consequently, this greatly enhances the module's throughput and stability.

FIFO Module RTL Definition

```
module sync_fifo #(
    parameter WIDTH = 32,
    parameter DEPTH = 128
) (
    input wire clk_i,
    input wire rst_ni,
    input wire [WIDTH-1:0] wdata_i,
    input wire we_i,
    input wire re_i,
    output wire [WIDTH-1:0] rdata_o,
    output reg full_o,
    output reg empty_o
);

// address width for buffer
localparam ADDR_BITS = $clog2(DEPTH);
```

Listing 3.4: Verilog Implementation of Synchronous FIFO Queue

3.4.2 UART Interrupts

The UART module incorporates the use of interrupts to enhance its functionality and provide efficient communication with the microprocessor. Currently, the module supports a specific interrupt known as the *byte received* interrupt.

The *byte received* interrupt is generated when the internal signal NOT(rx_fq_empty) is active. This signal indicates that the receiver's FIFO (First-In-First-Out) queue contains a new packet of data that has been received. The interrupt serves as a notification mechanism to inform the microprocessor that new data is available and waiting to be processed.

3.4.3 Timing control using Clock Enable signals

The UART module utilizes clock enable signals for both the transmitter and the receiver to ensure accurate and reliable communication.

Transmitter Clock Enable

The transmitter clock enable signal operates using a 32-bit counter that increments by 1 with each clock tick. When the counter value matches the baud rate minus 1, it is reset and the clock enable signals is set to 1 for exactly one clock cycle.

```
always @(posedge clk_i, negedge rst_ni) begin
    if(~rst_ni) begin
        tx_tick_counter <= 32'b0;
        tx_clk_en <= 1'b0;
    end
    else begin
        tx_tick_counter <= (tx_clk_en)? 32'b0 :
        tx_tick_counter + 1'b1;

        tx_clk_en <= (tx_tick_counter == (baud_rate - 1'b1));
    end
end
```

Listing 3.5: Verilog Implementation of Transmitter Tick Counter Logic

Receiver Clock Enable

On the other hand, the design of the receiver's clock enable signal is slightly more intricate. The receiver continuously samples the incoming signal on the RX line with each clock tick. The observed value is then shifted into a 3-bit shift register. When the value in the shift register reaches the pattern 3'b100 and the receiver's state machine is in the IDLE state, the receiver loads a value equal to the baud rate divided by 2 into its tick counter and initiates counting.

The receiver's clock enable signal is activated each time the counter reaches a value of baud rate minus 1. This ensures that the clock enable signal aligns precisely with the center of each transmitted bit's duration.

```
always @(posedge clk_i, negedge rst_ni) begin
    if(~rst_ni) begin
        rx_buf <= 3'b111;
    end
    else begin
        rx_buf <= {rx_buf[1:0], uart_rx_i};
    end
end
assign rx_start = rx_buf[2] & ~rx_buf[1] & ~rx_buf[0] & rx_state == rx0.IDLE;
always @(posedge clk_i, negedge rst_ni) begin
    if(~rst_ni) begin
        rx_tick_counter <= 32'b0;
        rx_clk_en <= 1'b0;
    end
    else begin
        rx_tick_counter <= (rx_clk_en)? 32'b0 :
        (rx_start)? baud_rate >> 1 :
        rx_tick_counter + 1'b1;
        rx_clk_en <= (rx_tick_counter == (baud_rate - 1'b1));
    end
end
```

Listing 3.6: Verilog Implementation of Receiver Tick Counter Logic

This feature significantly enhances the stability and robustness of the module by minimizing the possibility of reading false values from the transmission line due to noise or timing inconsistencies.

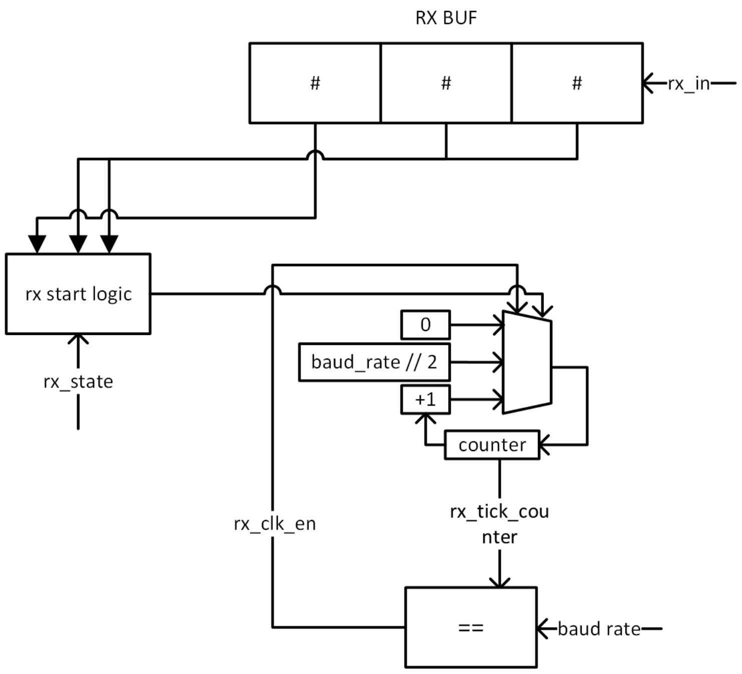


Figure 3.9: Receiver Tick Count Block Diagram

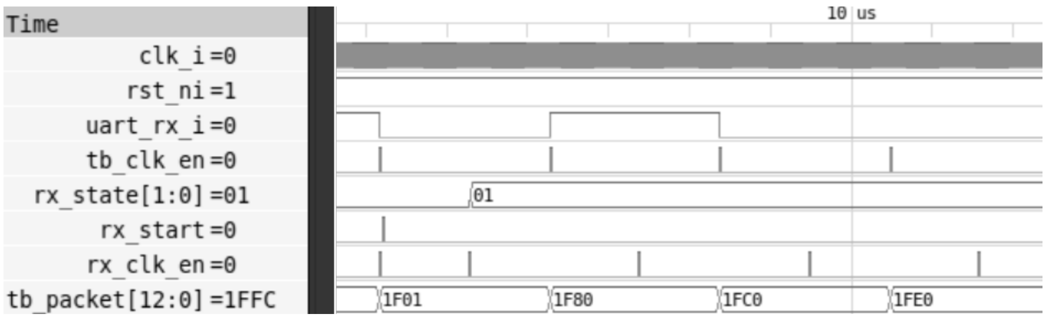


Figure 3.10: Receiver Tick Count Waveform

Chapter 4

UART Microcode

When the UART Controller module receives a valid memory request from the microprocessor (we will discuss later what is considered valid and what is not), it translates the RISC-V instruction into local microcode.

These microcode instructions provide the necessary control and communication capabilities for the UART Controller module, enabling the microprocessor to interact with external devices or systems using the UART interface.

The UART microcode consists of 5-bit instructions encoded using a one-hot encoding format to mitigate possible meta-stability issues at higher frequencies.

The UART Controller module supports the following microcode instructions, each serving a specific purpose in facilitating communication and control

The following instructions are being supported so far:

(4.1) OP_WRITE_UART_PARAMETERS

(4.2) OP_WRITE_UART_EN

(4.3) OP_READ_UART_STATE

(4.4) OP_WRITE_UART_DATA

(4.5) OP_READ_UART_DATA

(4.6) OP_DO_NOTHING

Since we focus on a memory mapped implementation of the peripheral our instructions make use of a base 32-bit address plus a prespecified offset also referred as BASE_UART.

4.1 Writing Protocol Parameters

This instruction is used to write the UART protocol parameters, such as data size, parity, and stop bits, which define the format of transmitted and received data. In order to write the parameters to the memory mapped registers with software we use several predefined masks, to which we perform a bit-wise OR operation to get the final vector of parameters.

The address of the memory mapped registers in hexadecimal is BASE_UART + 0x10

Assuming data bus of 32 bits wdata[31:0]
wdata[1:0] bits are used to determine the bit size of the data

- 00 => 6 Data Bits
- 01 => 7 Data Bits
- 10 => 8 Data Bits
- 11 => 9 Data Bits

Following wdata[3:2] bits are used to determine the parity type used in the protocol

- 00 => None Parity
- 01 => Even Parity

! 10 => (Not Used)

- 11 => Odd Parity

wdata[4] bit is used to determine the number of stop bits after receiving a byte

- 0 => 1 Stop Bits
- 1 => 2 Stop Bits

Finally wdata[6:5] bits are used to determine the baud rate of the protocol

- 00 => 4800 Baud per Second
- 01 => 9600 Baud per Second
- 10 => 57600 Baud per Second
- 11 => 115200 Baud per Second

4.2 Writing Enable Register

This instruction enables or disables the UART transmitter and/or receiver submodules, controlling the transmission and reception of data.

The address of the memory mapped registers in hexadecimal is BASE_UART + 0x0C

During this operation, the wdata bus is employed to specify the desired result, using the following bit patterns:

- 00 => Both UART TX and RX disabled
- 01 => UART TX disabled but RX enabled
- 10 => UART TX enabled but RX disabled
- 11 => Both UART TX and RX enabled

Upon startup, both the rx enable and tx enable registers are initialized to 0, indicating that the UART modules are disabled by default. Therefore, to enable the UART functionality, it is necessary to set the appropriate bits in the registers.

4.3 Reading FIFO States

This instruction reads the current state of the UART's FIFO queues. It assembles a 32 bit vector with all bits set to 0 except the four less significant bits, which hold the (near) full flag of the transmitter FIFO and the (near) empty flag of the receiver FIFO queue.

The memory mapped register address that needs to be read in order to read the FIFO statuses is BASE_UART + 0x08

In Verilog code, the result of this instruction can be represented as the 32-bit vector (28'b0, tx_fq_near_full, rx_fq_near_empty, tx_fq_full, rx_fq_empty). This code indicates that the first 28 bits are set to 0. The first bit represents the transmitter FIFO full flag (tx_fq_full), and the least significant bit (bit 0) represents the receiver FIFO empty flag (rx_fq_empty). The third bit represents the near empty flag and the fourth bit the near full flag.

4.4 Writing Data to Transmitter FIFO

This instruction is used to write data to be transmitted through the UART interface.

To write data into the transmitter FIFO, the corresponding memory mapped register address is BASE_UART + 0x04.

When using this instruction, the 9 least significant bits of the data are written into the transmitter FIFO register. The remaining bits, are ignored.

4.5 Reading Data from Receiver FIFO

When executing this instruction, the contents of the receiver FIFO are accessed, allowing the microprocessor to read the received data.

To retrieve the received data, the corresponding memory mapped register address is BASE_UART + 0x00.

The retrieved result is a 32-bit vector, where the least significant 9 bits contain the received data. The remaining 23 bits are filled with zeros, so that the result is not sign-extended.

4.6 No Operation

This instruction is a no-operation instruction that does not carry out any specific action. It is used as a placeholder or filler instruction when there is no need for any meaningful operation or functionality.

Chapter 5

RISCV ISA and the IBEX Microprocessor

5.1 IBEX Microprocessor

Ibex is a production-quality open source 32-bit RISC-V CPU core written in SystemVerilog. The CPU core is heavily parametrizable and well suited for embedded control applications. Ibex is being extensively verified and has seen multiple tape-outs. Ibex supports the Integer (I) or Embedded (E), Integer Multiplication and Division (M), Compressed (C), and B (Bit Manipulation) extensions. Ibex offers several configuration parameters to meet the needs of various application scenarios. The options include different choices for the architecture of the multiplier unit, as well as a range of performance and security features. In summary, IBEX is an incredibly versatile microcontroller-class CPU core that seamlessly integrates into designs with ease.

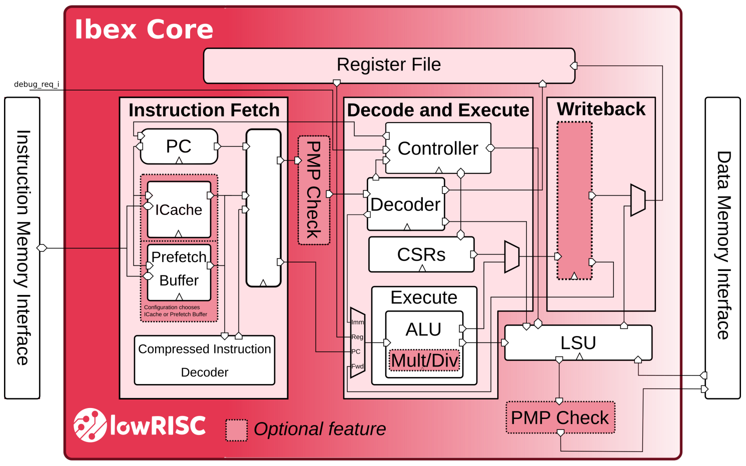


Figure 5.1: IBEX Block Diagram

As shown on the diagram 5.1 the core consists of two pipeline stages (three optional for Writeback stage). Firstly we have the Instruction Fetch (IF) stage which is responsible for fetching the instructions from the memory. Then there is the Decode and Execute stage where the instructions fetched are being decoded (ID) and executed (EX) by the core. Ibex can be configured to have a third pipeline stage (Writeback) which has major effects on performance and instruction behaviour. [7]

5.2 Load Store Unit

The IBEX microprocessor has many features and consists of many sub-modules. However in this thesis we are only going to focus on the LSU circuit. The Load-Store Unit (LSU) of the core takes care of accessing the data memory. Loads and stores of words (32 bit), half words (16 bit) and bytes (8 bit) are supported. Any load or store will stall the ID/EX stage for at least a cycle to await the response (whether that is awaiting load data or a response indicating whether an error has been seen for a store). [3]

The protocol that is used by the LSU to communicate with a memory works as follows:

- The LSU provides a valid address in data_addr_o and sets data_req_o high. In the case of a store, the LSU also sets data_we_o high and configures data_be_o and data_wdata_o. The memory then answers with a data_gnt_i set high as soon as it is ready to serve the request. This may happen in the same cycle as the request was sent or any number of cycles later.

- After receiving a grant, the address may be changed in the next cycle by the LSU. In addition, the data_wdata_o, data_we_o and data_be_o signals may be changed as it is assumed that the memory has already processed and stored that information.

- The memory answers with a data_rvalid_i set high for exactly one cycle to signal the response from the bus or the memory using data_err_i and data_rdata_i (during the very same cycle). This may happen one or more cycles after the grant has been received. If data_err_i is low, the request could successfully be handled at the destination and in the case of a load, data_rdata_i contains valid data. If data_err_i is high, an error occurred in the memory system and the core will raise an exception.

- When multiple granted requests are outstanding, it is assumed that the memory requests will be kept in-order and one data_rvalid_i will be signalled for each of them, in the order they were issued.

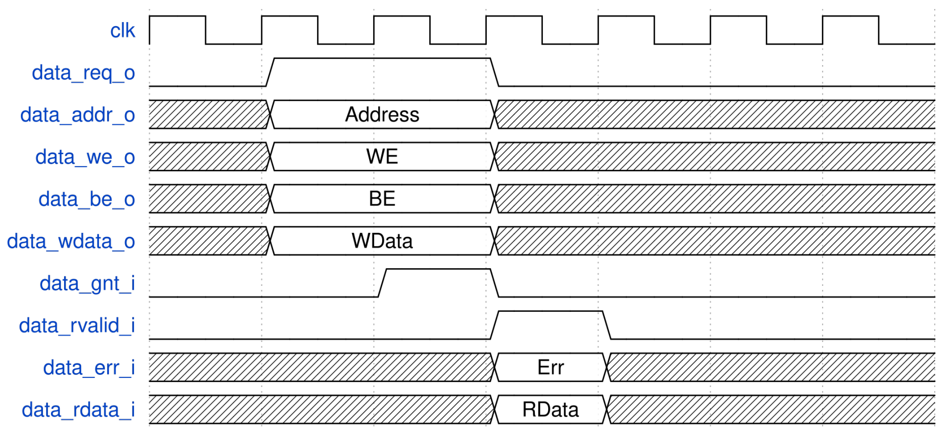


Figure 5.2: Basic Memory Transaction

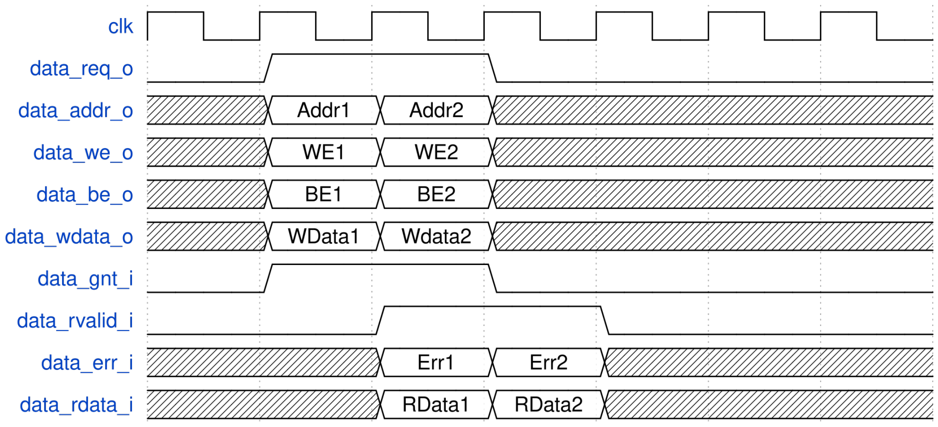


Figure 5.3: Back to Back Memory Transaction

5.3 Exceptions and Interrupts

Ibex implements trap handling for interrupts and exceptions according to the RISC-V Privileged Specification, version 1.11. Within our circuit, the irq_fast_i bus line plays a crucial role in managing interrupts. This bus line comprises 15 fast, local interrupts, with each interrupt represented by a specific bit. In our project we made use of the least significant bit of the irq bus (uart_irq).

However, it is important to note that the detailed implementation and handling of these signals by the processor itself are beyond the scope of this thesis. Therefore, we will not delve into the specific intricacies of how the processor manages or processes these signals.

5.4 IBEX Demo System

A more complete implementation can be found in the Ibex Demo System repository. In particular it includes a integration of the PULP RISC-V debug module. It targets the Arty A7 FPGA board from Digilent and supports debugging via OpenOCD and GDB over USB (no external JTAG probe required). The Ibex Demo System is maintained by lowRISC but is not an official part of Ibex.

Within the Ibex Demo System, a UART module is already implemented using System Verilog. However, it should be mentioned that the specific feature of updating the protocol's parameters through software, which distinguishes our variation, is not yet implemented at this time in the Ibex Demo System.

The UART module Controller we have developed is fully compatible with the Load Store Unit (LSU) Protocol of the Ibex microprocessor. All the essential signals required for successful communication between the peripheral and the LSU, such as req_gnt and data_rvalid, have been implemented and thoroughly tested within the context of the Ibex Demo System. This ensures seamless integration and reliable operation between the UART module and the Ibex microprocessor.[4]

It comprises the lowRISC Ibex core along with the following features:

- RISC-V debug support (using the PULP RISC-V Debug Module)

- A UART

- GPIO (output only for now)
- Timer
- SPI
- A basic peripheral to write ASCII output to a file and halt simulation from software

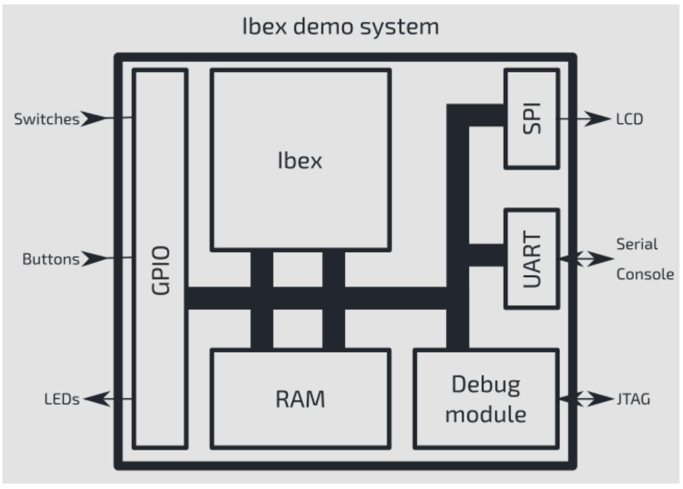


Figure 5.4: IBEX Demo System

Chapter 6

Controlling the UART Module: C Code Implementation and Integration

6.1 Implementation

In the process of controlling the UART module, a series of C routines were developed. These routines incorporated both the existing code from the IBEX Demo System and new code that was written specifically for our variation of the UART module.

The implementation of the UART control routines benefited greatly from the existing infrastructure provided by the IBEX Demo System repository. This solid foundation served as a robust framework for building and integrating the new routines.

We contributed to the IBEX Demo System software ecosystem by adding the following routines along with the mandatory defined constants

- **void uart_enable(uart_t uart, char en)**: This routine is responsible for enabling or disabling the UART transmitter and receiver sub-modules. It takes two parameters: uart, which specifies the in memory address of the UART module to be controlled, and en, which determines the enable status. The function uses the DEV_WRITE macro to write the corresponding value to the UART enable register.

- **void uart_disable(uart_t uart)**: This routine is used to disable the UART transmitter and receiver sub-modules. It takes the uart parameter, specifying the in memory address of the UART module to be controlled. By utilizing the DEV_WRITE macro, the function writes a value of 0x00 to the UART enable register, effectively disabling both the transmitter and receiver.

- **void uart_setup(uart_t uart, char parameters)**: This routine is responsible for configuring the UART protocol parameters. The uart parameter specifies the in memory address of the UART module and the parameters parameter represents the desired protocol parameters, such as data size, parity, and stop bits. The function utilizes the DEV_WRITE macro to write the parameters value to the UART parameters register, setting up the desired protocol configuration.

- **int uart_status(uart_t uart)**: This suite is used to read the status registers of the FIFO queues. The reason this function was implemented was to preserve the encapsulation structure of the dev header file macros.

6.2 Software Configuration and Initialization

In order to allow for the writing to the new addressable registers inside the UART module, the following memory addresses have been added:

UART Memory Mapped Address Offset	
UART_RX_REG	0x00
UART_TX_REG	0x04
UART_STATUS_REG	0x08
UART_ENABLE_REG	0x0C
UART_PARAMETERS_REG	0x10

Table 6.1: UART Memory Mapped Registers Address Offset

These addresses allow direct access to the corresponding registers for controlling the UART module's behavior and parameters of the protocol.

Additionally, to simplify the usage and configuration of the UART module, valid parameter values have been defined:

UART Data Size Constants Bit Maps	
DATA_SIZE_6	0x00
DATA_SIZE_7	0x01
DATA_SIZE_8	0x02
DATA_SIZE_9	0x03

Table 6.2: UART Defined Data Size Constants

UART Parity Type Constants Bit Maps	
PARITY_NONE	0x00
PARITY_EVEN	0x04
PARITY_ODD	0x0C

Table 6.3: UART Parity Type Constants

UART Stop Bits Constants Bit Maps	
STOP_BITS_ONE	0x00
STOP_BITS_TWO	0x10

Table 6.4: UART Stop Bits Constants

UART Baud Rate Constants Bit Maps	
BAUD_RATE_4800	0x00
BAUD_RATE_9600	0x20
BAUD_RATE_57600	0x40
BAUD_RATE_115200	0x60

Table 6.5: UART Baud Rate Constants

To configure the UART module to the desired parameters now it is as simple as performing a bit-wise OR operation between the desired parameter values and calling the uart_setup routine.

For example, let's configure the UART module with a data size of 8 bits, no parity, one stop bit, and a baud rate of 9600, also known as 8N1-9600. We can use the defined parameter values mentioned earlier and perform a bit-wise OR operation to combine them:

```
char parameters = DATA_SIZE_8 |
                  PARITY_NONE |
                  STOP_BITS_ONE |
                  BAUD_RATE_9600;
uart_setup(uart, parameters);
```

Listing 6.1: C Parameter Write Example

To enable the receiver and transmitter modules of the UART on startup, you can use the uart_enable routine and the defined constants for UART module enable bit maps. By performing a bit-wise OR operation between the desired enable constants, you can activate both modules in a single operation.

First, let's take a look at the defined constants for the UART module enable bit maps:

UART Module Enable Constants Bit Maps	
UART_RX_EN	0x01
UART_TX_EN	0x02

Table 6.6: UART Module Enable Constants

To enable both the receiver and transmitter modules, you can perform a bit-wise OR operation on the enable constants and pass the resulting value to the uart_enable routine. Here's an example:

```
char en  =  UART_TX_EN | UART_RX_EN;
uart_enable(uart, en);
```

Listing 6.2: C Module Enable Example

In the above example, the bit-wise OR operation combines the constants UART_TX_EN and UART_RX_EN, enabling both the transmitter and receiver modules. The resulting value is then passed to the uart_enable routine, activating the modules and making them operational.

In Listing 6.3, we provide the implementation of the configuration routines that were developed to enable dynamic UART parameter settings. These routines allow for seamless control over the UART module by providing functions to enable or disable the UART and to set up the desired parameters.

```
void uart_enable(uart_t uart, char en) {
  DEV_WRITE(uart + UART_ENABLE_REG, en);
}

void uart_disable(uart_t uart) {
  DEV_WRITE(uart + UART_ENABLE_REG, 0x00);
}

void uart_setup(uart_t uart, char parameters) {
  DEV_WRITE(uart + UART_PARAMETERS_REG, parameters);
}
```

Listing 6.3: C Configuration Routines

With the integrated UART routines, we can now dynamically configure the UART parameters, such as data size, parity, stop bits, and baud rate, through software. This flexibility allows for greater control and adaptability in UART communication.

6.3 Read and Write Operations

In the context of the system's memory-mapped addresses, the registers can be classified into two categories: read-only registers and write-only registers.

To enable the reading and writing of data from and to memory addresses within the system, two macros have been defined: DEV_READ and DEV_WRITE.

6.3.1 Writing to Memory Mapped Registers

Firstly, we have the write-only registers, which are designed to hold specific configuration or control signals for the submodules within the system and the protocol overall. As an example, consider the registers that hold the enable signal for each submodule. These registers are categorized as write-only registers because their functionality is to enable or disable the receiver/transmitter submodules.

The DEV_WRITE macro is used for writing data to a specific memory address. By providing the desired memory address and the data to be written as parameters to the DEV_WRITE macro, the system software can store the data at the specified address.

6.3.2 Reading from Memory Mapped Registers

On the other hand, we have the read-only registers, which provide information and status updates about various components of the system. These registers allow the system software to retrieve specific data or monitor the state of certain modules. The FIFOs state registers are an example of read-only registers. These registers provide information about the current state of the FIFOs such as the full flag of the transmitter and the empty flag of the receiver. The system software can read the

Memory Mapped Register	Register Type
UART_RX_EN_REG	W
UART_TX_EN_REG	W
UART_FIFO_STATUS_REGs	R
UART_PARAMETERS_REGs	W
UART_RX_FIFO_REG	R
UART_TX_FIFO_REG	W

Table 6.7: UART Registers Classes

values from these registers to obtain insights into the state of the FIFO but cannot modify their contents.

Similarly the DEV_READ macro is designed for reading data from a specific memory address. By passing the desired memory address as a parameter to the DEV_READ macro, the system software can retrieve the data stored at that address.

6.4 Writing and Reading Data over Serial using Software

In order to read and write data over serial using software several functions have been implemented. These functions are fully compatible with our changes on the UART module hardware.

6.4.1 Fundamental Routines of Serial IO

Firstly taking a look on the fundamentals we have the uart_in and uart_out functions.

```
int uart_in(uart_t uart) {
  int res = UART_EOF;
  if (!(DEV_READ(uart + UART_STATUS_REG) & UART_STATUS_RX_EMPTY)) {
    res = DEV_READ(uart + UART_RX_REG);
  }
  return res;
}
```

Listing 6.4: Fundamental Routine on Serial Read

The uart_in function is responsible for receiving data from the serial line. It checks if data is available in the receive buffer of the UART module and then reads the data from the buffer.

```
void uart_out(uart_t uart, char c) {
  while(DEV_READ(uart + UART_STATUS_REG) & UART_STATUS_TX_FULL);
  DEV_WRITE(uart + UART_TX_REG, c);
}
```

Listing 6.5: Fundamental Routine on Serial Write

Additionally the uart_out function is used for transmitting data over the serial line. It waits until the transmit buffer of the UART module is ready to accept data and then writes the data to the buffer.

6.4.2 Higher Level Functions of Serial IO

In this subsection we take a look at the implemented functions for handling data received on the serial line and sending data over it.

These functions are build upon the fundamental uart_in and uart_out functions mentioned earlier, providing a more convenient and user-friendly interface for serial I/O operations.

Sending a Byte Over Serial

The function presented in Listing 6.6 demonstrates how to transmit a single character over the serial line. This function, named putchar, serves as an interface for sending a C character to the serial interface.

```
int putchar(int c) {
#ifdef SIM_CTRL_OUTPUT
  DEV_WRITE(SIM_CTRL_BASE + SIM_CTRL_OUT, c);
#else
  if (c == '\n') {
    uart_out(DEFAULT_UART, '\r');
  }
  uart_out(DEFAULT_UART, c);
#endif
  return c;
}
```

Listing 6.6: Transmitting a C character over Serial

Receiving a Byte Over Serial

The code snippet in Listing 6.7 presents a simple routine for receiving a single character from the serial line.

The function, named getchar, provides an interface to retrieve a character from the serial interface and returns it as an integer value.

```
int getchar(void) {
  return uart_in(DEFAULT_UART);
}
```

Listing 6.7: Receiving a C character over Serial

Transmitting a String Over Serial

The provided code snippet in Listing 6.8 illustrates a function named puts that is responsible for transmitting a null-terminated string over the serial line.

The puts function takes a pointer to a character array (char str*) as its parameter. It utilizes a while loop to iterate through the characters of the string until it encounters a null character ('\0'), indicating the end of the string.

Within each iteration, the function calls the putchar function to transmit the current character to the serial line. The putchar function, as previously explained, handles the transmission of individual characters over the serial interface.

```
int puts(const char *str) {
  while (*str) {
    putchar(*str++);
  }
  return 0;
}
```

Listing 6.8: Transmitting a C string over Serial

Transmitting a Hexadecimal Value Over Serial

The provided code snippet in Listing 6.9 showcases a function named puthex that facilitates the transmission of a 32-bit hexadecimal value over the serial line.

The puthex function takes a 32-bit unsigned integer (uint32_t h) as its parameter, representing the value to be transmitted. It employs a for loop to iterate through each hex digit of the value, starting from the most significant digit.

```
void puthex(uint32_t h) {
  int cur_digit;
  // Iterate through h taking top 4 bits each time and outputting ASCII of hex
  // digit for those 4 bits
  for (int i = 0; i < 8; i++) {
    cur_digit = h >> 28;

    if (cur_digit < 10)
      putchar('0' + cur_digit);
    else
      putchar('A' - 10 + cur_digit);

    h <<= 4;
  }
}
```

Listing 6.9: Transmitting a C 32-bit unsigned integer over Serial

6.5 Software side of Interrupts

In the previous sections, we have explored the conventional polling method for reading the states of the UART receiver FIFO.

While this method is widely used, it is considered objectively slow and inefficient in most scenarios. To address this limitation, our UART implementation incorporates a software interrupt mechanism, also previously referenced on the hardware description as ”on-packet-receive-interrupt”. This interrupt is designed to notify the microprocessor when a packet is received over the serial line and stored in the receiver’s sub-module FIFO queue.

To use the provided interrupt there is are certain routines that need to be called beforehand.

In order to configure the routine to be executed when the UART exception occurs, we utilize the function ”install_exception_handler” with the appropriate IRQ number and a valid function pointer of the routine to be executed. In our case, the UART exception has an IRQ number of 16. To create the interrupt mask, we shift the value 1 to the left by the IRQ_NUM times.

The reason behind using 16 as the interrupt mask is rooted in the hardware architecture. At the hardware level, the fast interrupt request vector occupies the 15 most significant bits of the 32-bit interrupt vector. The UART IRQ is hooked at the 17th bit of the 32-bit vector, thus the corresponding interrupt mask is calculated.

A simple yet effective example of installing and utilizing the ”on-packet-receive” interrupt is showcased in the demo program section of this thesis. The purpose of this example is to demonstrate the practical implementation of the interrupt mechanism and illustrate its functionality within a real-world context. In the example provided on the code of the demo program the exception triggers the ”test_uart_irq_handler routine which when executed reads the received byte and echoes it back on the serial line to the receiver of the other device.

Also we created an interrupt to activate on the near full flag of the transmitter. If the FIFO is about to raise the full flag on the next request we stall the microprocessor using an empty for block and then check again on the FIFO’s status. This technique ensures that no time will be lost on handling failed memory requests.

6.6 Demo Program

In this section, we present the integration of the C routines developed as part of this thesis into the hello_world program from the ibex-demo-system repository.

The hello_world program was modified to incorporate the implemented routines, showcasing their practical usage and demonstrating their functionality within a real-world context.

Firstly, the mandatory header files are included, and the necessary constants are defined.

```
#include "uart.h"
#include "demo_system.h"
#include "timer.h"
#include "gpio.h"
#include "pwm.h"
#include <stdbool.h>
#include <stdint.h>

#define USE_GPIO_SHIFT_REG 0
```

Listing 6.10: C Demo Program Constants

Next, the interrupt handler routine is defined. This handler is responsible for handling UART interrupts and processing the received data.

```
void test_uart_rx_irq_handler(void) __attribute__((interrupt));

void test_uart_rx_irq_handler(void) {
    unsigned char u;
    u = getchar();
    putchar(u);
}
```

Listing 6.11: C Demo Program Interrupt

Finally, the main function is where the microprocessor is instructed to alter the colors of some RGB LEDs and send or receive data from the serial line. The example code below is trimmed to discard code not regarding the UART serial communication.

```
int main(void) {
  install_exception_handler(UART_RX_IRQ_NUM, &test_uart_rx_irq_handler);
  uart_enable_rx_int();

  uart_enable(DEFAULT_UART, UART_RX_EN | UART_TX_EN);
  uart_setup(DEFAULT_UART, DATA_SIZE_8 | PARITY_NONE | STOP_BITS_ONE |
    BAUD_RATE_115200);

  // This indicates how often the timer gets updated.
  timer_init();
  timer_enable(5000000);

  uint64_t last_elapsed_time = get_elapsed_time();

  const char str[] = " Hello World ";
  uint32_t c = 0;
  while(1) {
    uint64_t cur_time = get_elapsed_time();

    if (cur_time != last_elapsed_time) {
      last_elapsed_time = cur_time;

      set_global_interrupt_enable(0);
      puts(str);
      puthex(c++);
      puts("\r\n");
      set_global_interrupt_enable(1);
    }
    asm volatile ("wfi");
  }
}
```

Listing 6.12: C Demo Program Main

At the top of the main function we enable the UART on-packet-receive interrupt and then we enable both the receiver and transmitter sub-modules. Then we set the protocols parameters to a data size of 8 bits, no parity bit, one stop bit and a baud rate of 115200 baud per second.

After that we initialize a constant character array to broadcast, the classic Hello World string, and a counter variable named c.

In the infinite loop at of the main function we transmit the Hello World string accompanied by a numerical value in hexadecimal representation of the counter variable. When transmitting we disable the global interrupt to ensure the stability of the transmission procedure. We enable the interrupt after we are done transmitting.

The entirety of this code was provided by the ibex-demo-system and is under the Apache-2.0 license.

Chapter 7

Code Coverage and Functional Verification through Simulation

7.1 Understanding Code Coverage and its Limitations

Coverage is a metric to assess the progress of functional verification activity. This plays a major role to get a clear picture on how well the design has been verified and also to identify the uncovered areas in verification.

Code coverage is a basic coverage type which is collected automatically. It tells us how well our HDL code has been exercised by our test bench. In other words, how thoroughly the design has been executed by the simulator using the tests used in the regression.

Code coverage is often classified into different types:

- Statement Coverage /Line Coverage: This gives an indication of how many statements (lines) are covered in the simulation, by excluding lines like module, endmodule, comments, timescale etc. This is important in all kinds of design. Statement coverage includes procedural statements, continuous assignment statements, conditional statement and Branches for conditional statements etc.

- Block coverage: A group of statements which are in the begin-end or if-else or case or wait or while loop or for loop etc. is called a block. Block coverage gives the indication that whether these blocks are covered in simulation or not. The nature of the block coverage & line coverage looks similar. Block coverage looks for a group of statements called block and line coverage/statement coverage checks whether each statement is covered.

- Conditional/Expression Coverage: This gives an indication how well variables and expressions (with logical operators) in conditional statements are evaluated. Conditional coverage is the ratio of number of cases evaluated to the total number of cases present. If an expression has Boolean operations like XOR, AND ,OR as follows, the entries which is given to that expression to the total possibilities are indicated by expression coverage.

- Toggle Coverage: Toggle coverage gives a report that how many times signals and ports are toggled during a simulation run. It also measures activity in the design, such as unused signals or signals that remain constant or less value changes.

- State/FSM Coverage: FSM coverage reports, whether the simulation run could reach all of the states and cover all possible transitions or arcs in a given state machine. This is a complex coverage type as it works on behaviour of the design, that means it interprets the synthesis semantics of the HDL design and monitors the coverage of the FSM representation of control logic blocks.

Code coverage is an important indication for the verification engineer on how well the design code has been executed by the tests.

But it does not know anything about the design and what the design is supposed to do. There is no way to find what is missing in the RTL code, as code coverage can only evaluate the quality of the implemented code.

By achieving high code coverage, developers can gain confidence in the correctness and reliability of their software system. However, it's important to note that achieving 100% code coverage does not guarantee the absence of bugs or errors, as it only indicates the extent of code execution during testing.

7.2 Vivado Code Coverage

Code coverage is automatically extracted by the simulator when enabled. AMD Vivado™ simulator currently supports 4 types of code coverage, line, branch, condition, and toggle. When the user enables code coverage for any of the code coverage types, the tool will automatically generate a code coverage database.

To view the coverage of design, AMD Vivado™ simulator provides a standalone executable named as xcrg (Xilinx Coverage Report Generator), which can be used to generate coverage reports by reading the coverage database.

The code coverage functionality supported by the simulator is :

- Line/Statement coverate with exact execution count of statements

- Branch Coverage for if-else, if-elseif-else, switch case, ternary operators

- Detection and highlighting of missing else and missing default in code coverage report

- Condition Coverage with exact count of the number of times a condition has been checked and evaluated to TRUE/FALSE

- packed/unpacked reg, bit, logic, wire data types

- int, shorting, integer, byte datatypes and non-dynamic struct members

- Generation of code coverage HTML report by using xcrg

- Dashboard view of code coverage of design

- List of file, module and hierarchical instances for whole design

- File specific code coverage view, module, and instance specific views
- Merging of Line/Statement, Branch, Condition, and Toggle coverage from different runs using xcrg

Note: Currently, Vivado simulator only supports these features for SystemVerilog and Verilog code. VHDL is not supported yet.

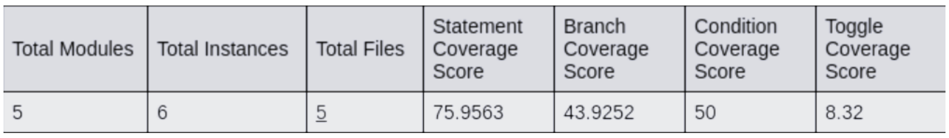
7.3 Simulating Packet Reception

In order to produce the waveform 3.10 showing the receiver tick count and receiver clock enable signals we implemented a test suite in Verilog. This test script simulates a packet being transmitted on the serial line and how the uart circuit behaves to receive it.

To produce the code coverage report file the following tcl commands were utilized

1. xvlog <verilog absolute file(s) path>
2. xelab <tb_module_name> -cc_type sbct -cc_db DB1 -cc_dir ./cRun1 -R
3. xcrg -cc_db DB1 -cc_dir ./cRun1 -cc_report ./cReport1

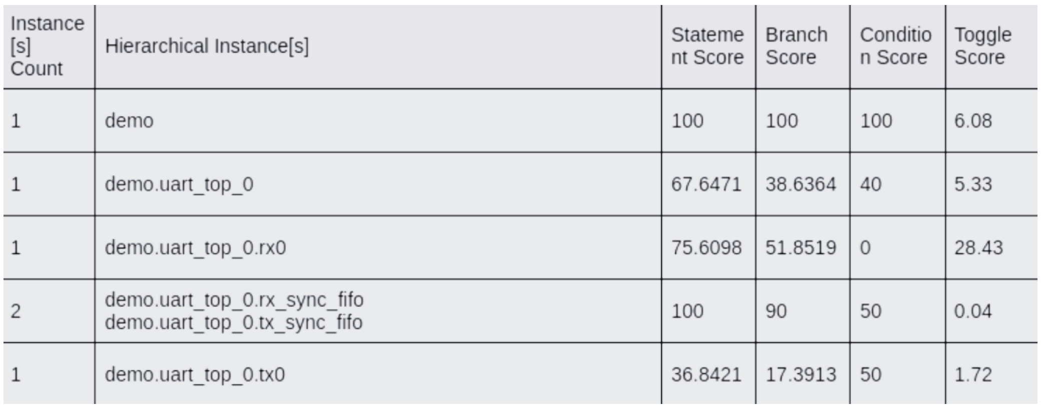
When examining the produced dashboard.html file we observe the following metrics results.



Total Modules	Total Instances	Total Files	Statement Coverage Score	Branch Coverage Score	Condition Coverage Score	Toggle Coverage Score
5	6	5	75.9563	43.9252	50	8.32

Figure 7.1: Code Coverage Summary

Vivado also allows for a more detailed view of the code coverage metrics produced by a test suite by calculating the coverage score for each module separately. This feature allows for further insight of how well our HDL code has been exercised by our test bench. These results are available at the modules.html file produced.



Instance [s] Count	Hierarchical Instance[s]	Stateme nt Score	Branch Score	Conditio n Score	Toggle Score
1	demo	100	100	100	6.08
1	demo.uart_top_0	67.6471	38.6364	40	5.33
1	demo.uart_top_0.rx0	75.6098	51.8519	0	28.43
2	demo.uart_top_0.rx_sync_fifo demo.uart_top_0.tx_sync_fifo	100	90	50	0.04
1	demo.uart_top_0.tx0	36.8421	17.3913	50	1.72

Figure 7.2: Code Coverage Module

The procedure described in this test suite follows a series of simple steps to verify the functionality of the circuit.

1. Initialize all signals to either logic zero or logic 1, according to what the active ”on start” value should be and define the packet to be transmitted over the serial line.
2. Reset the circuit by pulling the rst signal to active low and the after a short delay set it back to active 1.
3. Start the clock with a period of 10 nano seconds.
4. Enable both the receiver and the transmitter sub-modules.
5. Shift the value (packet) stored in the shift register and extract each bit over to the serial line.

The initial block of the demo test suite sets up the initial conditions and configurations for the test.

```
initial begin
    $dumpfile("test_uart.vcd");
    $dumpvars(0, demo);

    tb_tick_counter <= 0;
    tb_packet <= {3'b111, 8'b0000_0001, 2'b01};
    tb_clk_en <= 0;
    clk_i <= 'b0;
    we <= 'b0;
    be <= 'b0;
    uart_wdata_i <= 'b0;
    addr_i <= 'b0;
    uart_req_i <= 'b0;
    uart_rx_i <= 'b1;
    rst_ni <= 'b0;
    #1
    rst_ni <= 'b1;
    #1
    uart_wdata_i <= 32'b11;
    we <= 1'b1;
    uart_req_i <= 1'b1;
    addr_i <= 32'h0c;
    #13
    uart_wdata_i <= 'b0;
    we <= 1'b0;
    uart_req_i <= 1'b0;
    addr_i <= 'b0;
    #100000
    $finish;
end
```

Listing 7.1: Initial Block of Demo Test Suite

- $dumpfile(”test_uart.vcd”) and $dumpvars(0, demo) are VCD dump commands that specify the name of the waveform dump file and the variables to be included in the dump, respectively.
- tb_tick_counter <= 0 initializes the tick counter to 0, which is used to track the number of clock cycles.

- tb_packet <= 3'b111, 8'b0000_0001, 2'b01 defines the packet to be transmitted over the serial line.

- tb_clk_en <= 0 sets the clock enable signal to 0, initially disabling the clock.

- clk_i <= 'b0 sets the clock signal to logic 0.

- we <= 'b0, be <= 'b0, uart_wdata_i <= 'b0, addr_i <= 'b0, uart_req_i <= 'b0 and uart_rx_i <= 'b1 sets the control signals to their default initial values.

- rst_ni <= 1'b0 #1 rst_ni <= 'b1 effectively resets the circuit.

- uart_wdata_i <= 32'b11, we <= 1'b1, uart_req_i <= 1'b1, and addr_i <= 32'h0c sets the values of the control signals required to enable both the receiver and transmitter sub-modules.

- uart_wdata_i <= 'b0, we <= 1'b0, uart_req_i <= 1'b0, and addr_i <= 'b0 reset the control signals to their default values after the modules have been enabled.

- #100000 introduces a significant delay to allow the test to run for a specific duration.

- $finish terminates the simulation.

In addition to that we generate a clock signal with a period of 10 nano seconds using the following always block.

```
always #10 clk_i <= ~clk_i;
```

Listing 7.2: Clock Signal Genrator Block

The baud rate generator block is responsible for generating the clock signal that determines the transmission rate of the test suite, simulating the functionality of the actual baud rate generator in the UART module.

By using the tb_clk_en signal to enable and disable the clock, the baud rate generator block ensures that the shift register operates at the desired transmission rate.

It simulates the behavior of the UART's baud rate generator, allowing accurate testing and evaluation of the UART module's functionality.

```
always @(posedge clk_i, negedge rst_ni) begin
    if(~rst_ni) begin
        tb_tick_counter <= 0;
    end
    else begin
        tb_tick_counter <= (tb_clk_en)? 0 : tb_tick_counter + 1;
        tb_clk_en <= (tb_tick_counter == ((CLOCK_FREQUENCY / 9600) - 1));
    end
end
```

Listing 7.3: Baud Rate Generator Block

Additionally, the shift register component plays a crucial role in the test suite by extracting the packet data from the least significant bit (LSB) and shifting it on each positive edge of the baud rate clock.

```
always @(posedge tb_clk_en) begin
    uart_rx_i <= tb_packet[0];
    tb_packet <= {1'b1, tb_packet[12:1]};
end
```

Listing 7.4: Shift Register Component Block

Finally, at the end of the test suite, an instance of our UART module is declared and connected to the signals previously used in our program.

```
uart_top #(
    .CLOCK_FREQUENCY(CLOCK_FREQUENCY),
    .RX_FIFO_DEPTH(128),
    .TX_FIFO_DEPTH(128)
) uart_top_0 (
    clk_i,            // clock
    rst_ni,           // reset not
    we,               // write enable
    be,               // byte enable
    uart_wdata_i,     // data bus
    addr_i,           // addr bus
    uart_req_i,       // request from core (IBEX LSU)
    uart_rx_i,        // rx line
    uart_tx_o,        // tx line
    uart_rdata_o,     // data bus
    uart_req_gnt_o,   // request granted to core (IBEX LSU)
    uart_rvalid_o,    // request valid to core (IBEX LSU)
    uart_irq_o,       // interrupt request (CSR)
    uart_err_o        // error to core (IBEX LSU)
);
```

Listing 7.5: UART Module Instantiation Block

7.4 Microcode Oriented Testing

In this test, we aimed to simulate the behavior of the IBEX microprocessor by creating an instruction read-only memory (ROM) and emulating the interaction with the UART module as if it were integrated into the actual microprocessor. The test covered all supported micro-functions and aimed to verify their functionality.

The test procedure consisted of the following steps:

1. Initialization: The test initialized the clock signal and the instruction memory address. The instruction memory was defined as a 2D array, representing the ROM content.

2. Instruction Setup: The ROM was populated with instructions representing different test scenarios. Each instruction had a specific format containing the 32-bit address, 32-bit data, write enable and the LSU request flag.

3. Execution: The test executed the instructions stored in the ROM by sequentially accessing them based on the instruction memory address. Each clock

cycle, the instruction memory address was incremented, and the corresponding instruction was fetched and executed.

4. Verification: Throughout the execution of the test, we verified the expected behavior of the UART module by examining the results of the executed instructions and their corresponding waveforms. This includes checking the successful configuration of protocol parameter registers, enabling the transmitter and receiver sub-modules, writing data to the transmitter's FIFO queue, reading the UART state, reading the receiver's FIFO, and handling addresses that are out of the module's memory address pool range.

Taking a more thorough look at the waveforms produced, we are going to examine every instruction executed using the GTK-Wave waveform visualizer application.

7.4.1 Writing Protocol Parameters

The first instruction executed is responsible for altering the baud rate of the UART module's protocol from the default value of 9600 to 115200. In this test scenario, the clock frequency is set to 1MHz.

To achieve a baud rate of 9600, the frequency divisor is calculated as 1,000,000 / 9600, resulting in a divisor of 104 (0x68 in hexadecimal representation), as seen in the waveform. The instruction calculates the new clock divisor value 0x08, which is derived from the desired baud rate of 115200, as seen in the waveform 7.3.

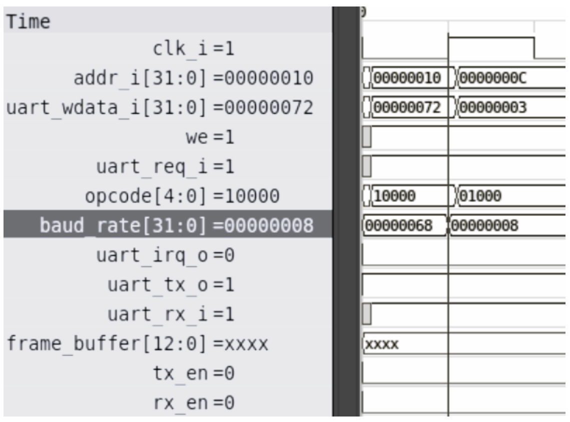


Figure 7.3: Write Protocol Parameters Waveform

7.4.2 Enabling the Receiver and Transmitter Sub-Modules

The second instruction fetched from the instruction memory enables both the receiver and transmitter sub-modules by writing the hexadecimal value 0x03 to the enable registers. As shown in waveform 7.4, the enable registers are updated to logic one after executing this instruction.

7.4.3 Writing to the Transmitters FIFO Queue

The third instruction in our test suite involves writing the value 0xa1 to the transmitter's FIFO queue. This action initiates the process of storing data in the queue and

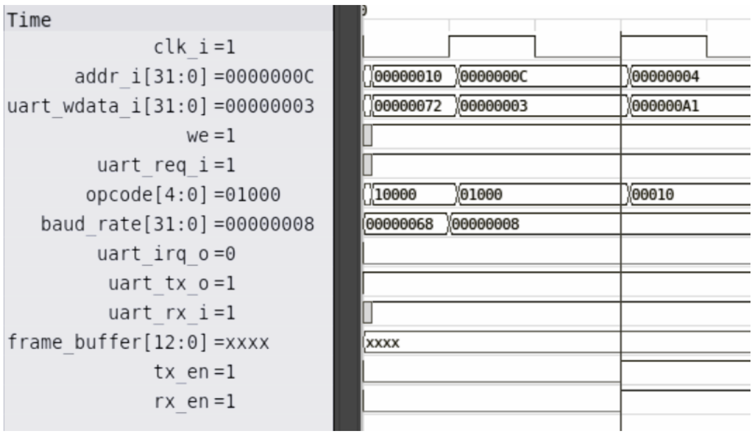


Figure 7.4: Write Enable Waveform

triggers the UART controller to notify the transmitter of the new data and update the queue status accordingly. The waveform 7.5 provides a visual representation of the signal transitions during the execution of this instruction. Specifically, the signal wdata_i holds the value that is written to the transmitter's FIFO queue, and the tx_fq_empty signal transitions from a logic one state to a logic zero, indicating that the FIFO is no longer empty and now contains a value.

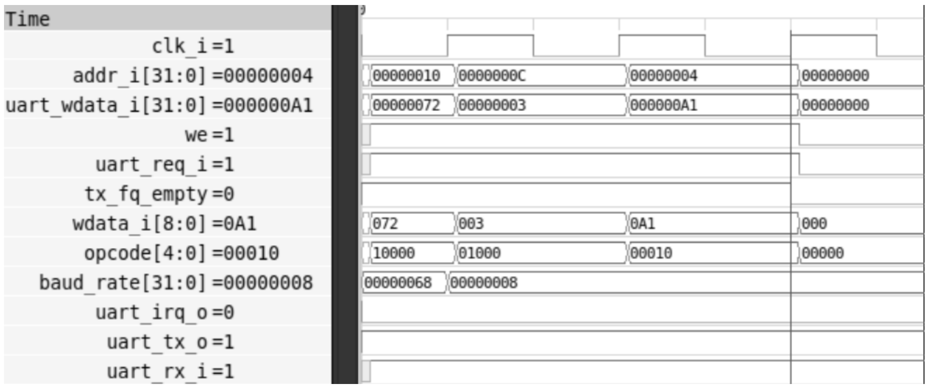


Figure 7.5: Write Data to Transmitter FIFO Waveform

7.4.4 The no-op Instruction

Another instruction stored in the read-only memory is the no-op instruction, also known as the do-nothing instruction. When this instruction is executed, the UART module simply performs no operation and ignores any input signals. The opcode corresponding to the no-op instruction is 0b0000 as displayed on the waveform 7.6 on the opcode signal.

7.4.5 Reading UART State

This instruction is responsible for reading the values of the transmitter's FIFO full flag and the receiver's FIFO empty flag. The waveform 7.7 shows that the uart_rdata_o signal of the bus line holds the value 0x01, indicating that the trans-

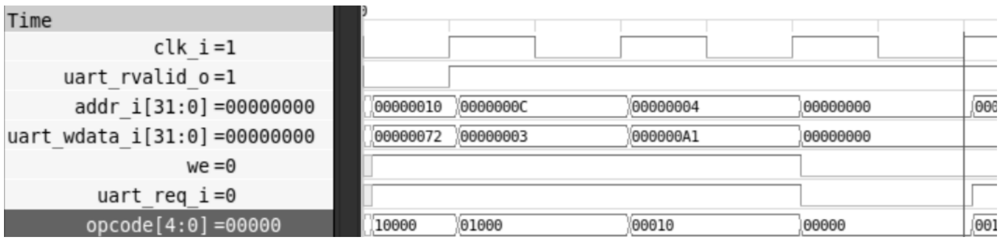


Figure 7.6: NO-OP Waveform

mitter's FIFO is not full, allowing us to store data if needed and the receiver's FIFO is empty, indicating that there is no unread data available in the queue.

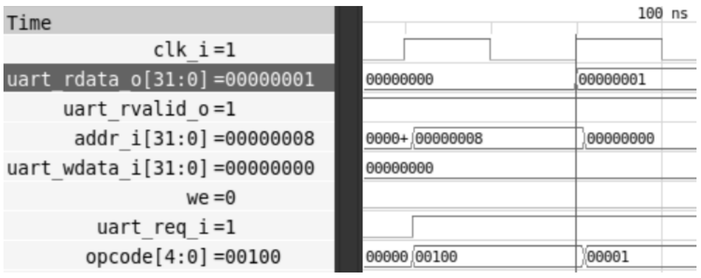


Figure 7.7: Read UART FIFO States Waveform

7.4.6 Invalid Address

Our design is equipped with the capability to recognize out-of-range addresses or invalid signal combinations that do not correspond to any valid coded instruction. In such cases, the design automatically transforms these invalid combinations into a no-op instruction, as depicted in Figure 7.8. Given the example, when the address value is 0xa1 and the data value is 0x03, with the write enable signal set to logic one, this particular combination does not translate to any valid instruction. However, our module intelligently recognizes this and executes a no-op instruction instead. This mechanism ensures that the module behaves correctly, even when provided invalid inputs.

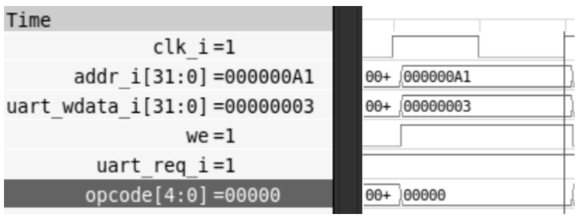


Figure 7.8: Invalid to No-Op Instruction Waveform

7.4.7 Reading Receiver FIFO Queue

This particular instruction is designed to read the received data stored in the receiver's FIFO queue. During simulation, we have implemented a short-circuit mechanism that connects the module's RX and TX lines together. As a result, every bit of data sent is automatically received, eliminating the need for emulating a separate transmitter within the test suite. As the suite progresses, the uart_irq_o signal transitions from logic zero to logic one, indicating to our imaginary microprocessor that new data has arrived. On the next clock cycle, the microprocessor reads this data using the read instruction. The successful reading of data is evident in waveform 7.9, where the uart_rdata_o bus holds the value 0xa1, which matches the data transmitted earlier during the test.

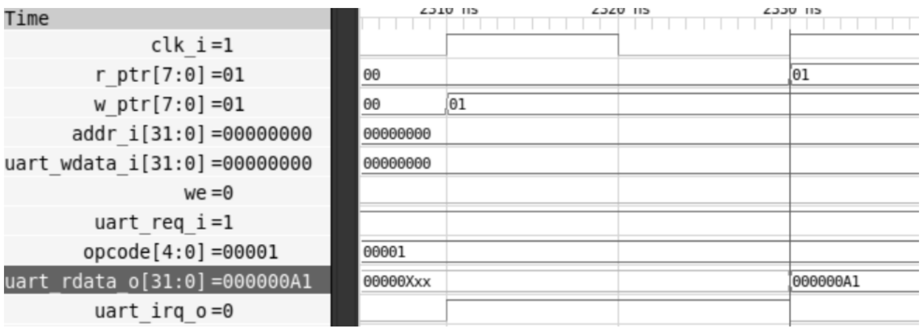


Figure 7.9: Reading Receiver FIFO Queue Waveform

7.4.8 Instruction Format and Initial Block of Suite

The provided Verilog code snippet represents the initialization and instruction setup of the microcode test suite. The initial block of the code sets the clock signal to a logic zero and initializes the instruction memory address to zero. The subsequent instructions are then stored in the instruction memory. The instructions are formatted as follows: {32-bit address, 32-bit data, 1-bit write enable, 1-bit uart_req_i}.

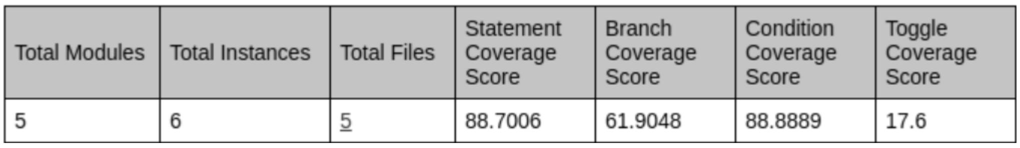
```
clk_i <= 1'b0;
instr_mem_addr = 'b0;
/* Format {32-bit address, 32-bit data, 1-bit write enable, 1-bit uart_req_i}; */
/* Alter Protocol Parameters */
/* stop-size_parity-type_parity-size_data-size */
/* 115200-2-NONE-0-8 */
i=0;    instr_mem[i] <= {32'h10, 32'b11_1_0_0_10, 1'b1, 1'b1};
/* Enable receiver and transmitter sub-modules */
i=i+1;  instr_mem[i] <= {32'h0c, 32'h3, 1'b1, 1'b1};
/* Write a1 hex value to transmitter FIFO */
i=i+1;  instr_mem[i] <= {32'h04, 32'ha1, 1'b1, 1'b1};
/* Idle */
i=i+1;  instr_mem[i] <= 'b0;
/* Read UART State */
i=i+1;  instr_mem[i] <= {32'h08, 32'h00, 1'b0, 1'b1};
/* Read receivers FIFO */
i=i+1;  instr_mem[i] <= {32'h00, 32'h00, 1'b0, 1'b1};
/* Out of range address translated to Idle */
i=i+1;  instr_mem[i] <= {32'ha1, 32'h03, 1'b1, 1'b1};
for(i=i+1; i < INSTR_MEM_DEPTH; i=i+1)
    instr_mem[i] <= {32'h00, 32'h00, 1'b0, 1'b1};
rst_ni <= 1'b0; #2
rst_ni <= 1'b1; #10000
```

Listing 7.6: Initialization and Instruction Setup of Microcode Test Suite

7.4.9 Coverage Scores of Microcode Suite

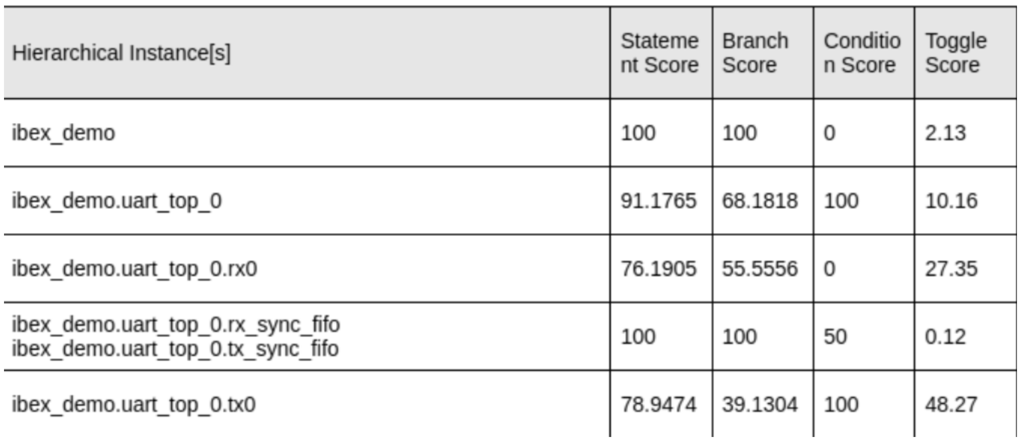
In the verification process of the microcode suite, coverage scores were obtained to evaluate the effectiveness of the test in terms of code coverage. The scores are presented in two tables: Table 7.10 shows the summary or total coverage score of the test suite, while Table 7.11 provides the scores for each individual module.

Most metrics exceed or are around 90%, indicating that the developed suite provides more than enough information about the expected behavior of our design. While Branch Coverage score falls slightly above the 50% threshold, it still indicates that a significant portion of the module's branches have been executed and tested. Toggle Score is the only low metric, at around 8%. The lower metric score does not necessarily indicate a lack of coverage or inadequate testing but rather reflects the toggling behavior within the design.



Total Modules	Total Instances	Total Files	Statement Coverage Score	Branch Coverage Score	Condition Coverage Score	Toggle Coverage Score
5	6	5	88.7006	61.9048	88.8889	17.6

Figure 7.10: Microcode Test Suite Coverage Score



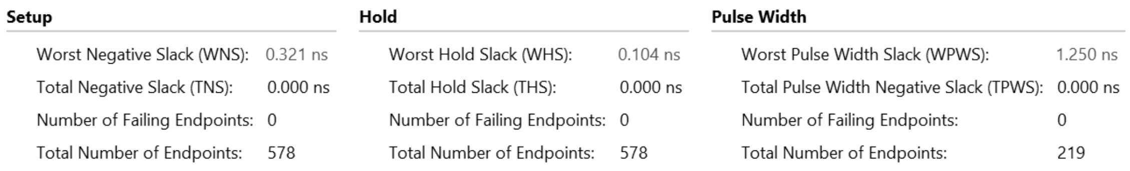
Hierarchical Instance[s]	Stateme nt Score	Branch Score	Conditio n Score	Toggle Score
ibex_demo	100	100	0	2.13
ibex_demo.uart_top_0	91.1765	68.1818	100	10.16
ibex_demo.uart_top_0.rx0	76.1905	55.5556	0	27.35
ibex_demo.uart_top_0.rx_sync_fifo ibex_demo.uart_top_0.tx_sync_fifo	100	100	50	0.12
ibex_demo.uart_top_0.tx0	78.9474	39.1304	100	48.27

Figure 7.11: Microcode Test Suite Coverage Per Module Score

7.5 Evaluation

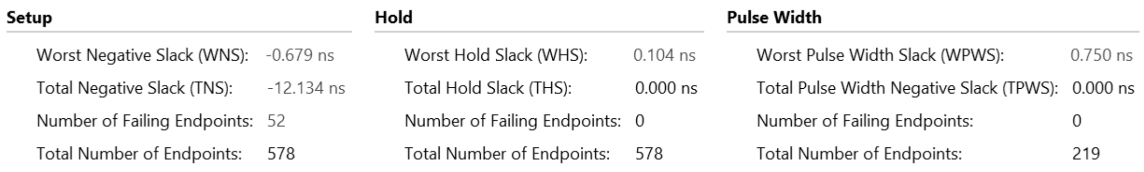
To determine the maximum operating frequency of our UART module, we conducted a series of tests by modifying the timing constraints file in Vivado and varying the clock period. Through extensive experimentation, we established that the standalone UART module can operate at a maximum frequency of 200 Mega Hertz, or a period of 5 nano seconds (Figure 7.12).

The last period value tested was that of 4 nanoseconds. However, upon using this frequency, instructing Vivado EDA to generate the corresponding circuit, computed a negative Worst Negative Slack (WNS) metric, indicating that the specified timing constraints are not met (Figure 7.13).



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.321 ns	Worst Hold Slack (WHS):	0.104 ns	Worst Pulse Width Slack (WPWS):	1.250 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	578	Total Number of Endpoints:	578	Total Number of Endpoints:	219

Figure 7.12: Vivado Timing Report



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.679 ns	Worst Hold Slack (WHS):	0.104 ns	Worst Pulse Width Slack (WPWS):	0.750 ns
Total Negative Slack (TNS):	-12.134 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	52	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	578	Total Number of Endpoints:	578	Total Number of Endpoints:	219

Figure 7.13: Vivado Timing Report with Invalid Clock

Furthermore, area usage is an important design consideration, comparing the original design from the ibex-demo-system repository (Figure 7.14) to our modified design (Figure 7.15) we observe the following changes in the utilization bar diagrams of the arty7a100 FPGA board.

In our modified design, we note a minimal 1% decrease in LUT utilization, indicating a slight decrease in logic complexity. Furthermore, we observe a 2% decrease in FF utilization, suggesting that our modifications have led to a slightly more efficient use of flip-flops within the design. An overall satisfying result as we managed to keep the same utilization metrics while expanding the circuits features.

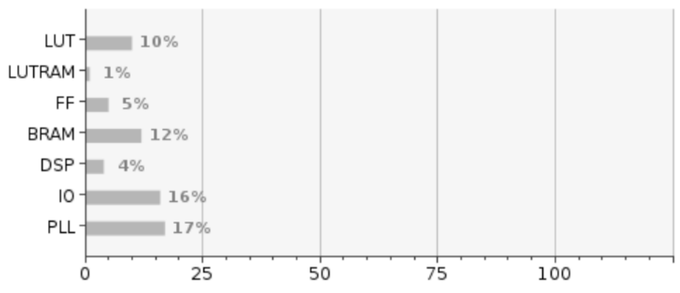


Figure 7.14: Original Design Board Utilization

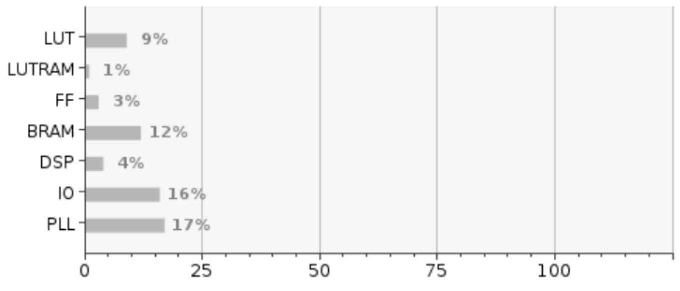


Figure 7.15: Modified Design Board Utilization

Chapter 8

Into the Realm of Asynchronous FIFOs

8.1 Background

An Asynchronous FIFO, is a FIFO (dual-port SRAM) memory structure that implements the transfer of data between different clock domains. It is characterized by the fact that the read and write clocks of the FIFO are not synchronized using the same clock domain. Asynchronous FIFOs play a crucial role in scenarios where data flows at different rates between systems, requiring synchronization between the clock domains.

8.1.1 Clock Domain

A clock domain is defined as part of the design that is driven by either one clock or more clocks that have related to each other. For example, a clock with frequency 10MHz and a divide by 2 clock driven from 10MHz clock are treated as a single clock domain design. However, designs which have two unrelated clocks (different clock frequencies) or clocks from two different sources (even with same frequency) are treated as multiple clock domain designs. [1]

8.1.2 Read and Write Pointers of an Asynchronous FIFO

In an asynchronous FIFO, two counters are employed: one for the write address, determining the location for the next data to be written, and another for the read address, determining the next address from which data will be read. This structure is a lot similar to the synchronous FIFO discussed earlier.

The key distinction between the Synchronous FIFO and the Asynchronous version is that these two counters belong to separate clock domains. We refer to them as the **w_clk** domain for the write counter and the **r_clk** domain for the read counter, respectively.

This difference causes an extremely crucial synchronization issue when we need to calculate the *full* and *empty* flags of the queue because the write pointer is aligned to the w_clock domain whereas the read pointer is aligned to the r_clock domain. Hence, it requires domain crossing to calculate the queue's flags.

To address this problem, synchronizers must be implemented to safely pass the write and read pointers between the different clock domains. By employing synchronizers, we can mitigate the risks associated with metastability and ensure the reliable operation of the asynchronous FIFO flags.

8.1.3 Flip-Flop Synchronizers in Clock Domain Crossing

The most used synchronizer is the multi-flop synchronizer as shown in figure 8.1.

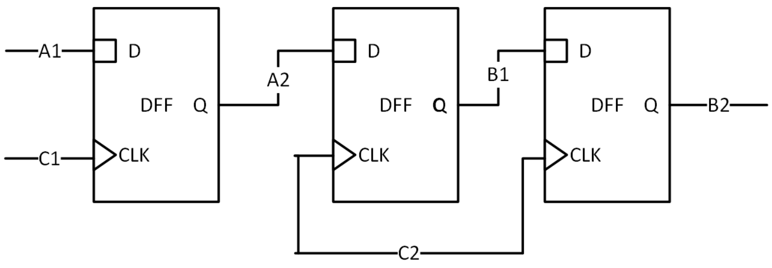


Figure 8.1: Two Flip-Flop Synchronizer

The first flip-flop, captures the signal in the source clock domain, while the other two flip-flops operate in the destination clock domain. [5]

This two-stage process helps to stabilize the signal by introducing an additional delay and reducing the risk of metastability. Flip-flop synchronizers are essential components in designs that require reliable and safe data transfer across different clock domains, ensuring proper synchronization and minimizing the potential for data corruption or loss.

When dealing with more than one bit of data we replicate this technique a time of n-bits.

8.1.4 Gray Code on Mitigating Metastability

Metastability is a critical issue that arises when signals from different clock domains are synchronized. It occurs when a signal changes close to the edge of a flip-flop's clock input, leading to an uncertain state that can cause erroneous behavior.

Another approach to mitigate metastability is by using Grey code encoding. The reflected binary code (RBC), also known as reflected binary (RB) or Gray code after Frank Gray, is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit).

For example, the representation of the decimal value "1" in binary would normally be "001" and "2" would be "010". In Gray code, these values are represented as "001" and "011". That way, incrementing a value from 1 to 2 requires only one bit to change, instead of two.

By employing Gray code in clock domain crossing, representing our data to be transmitted over the synchronizers using this encoding, we can ensure that only one bit changes at a time during the transition, reducing the likelihood of metastability.

8.1.5 The Asynchronous FIFO Queue

Summarizing all the above points of this section, the asynchronous FIFO queue's behavior closely resembles that of its synchronous counterpart, with one notable difference: the synchronization of the read and write pointers for calculating the *full* and *empty* flags. To ensure reliable synchronization and minimize the probability of errors, we employ a two-step approach involving Gray code conversion and a 2-Flip-Flop synchronizer circuit.

The read and write pointers, belonging to different clock domains, undergo a transformation into their respective Gray code representations. Gray code encoding provides the advantage of minimal state transitions, reducing the chances of metastability and misinterpretation during the domain crossing process. The Gray code encoded values of the pointers are then fed into an n-bit 2-Flip-Flop synchronizer circuit.

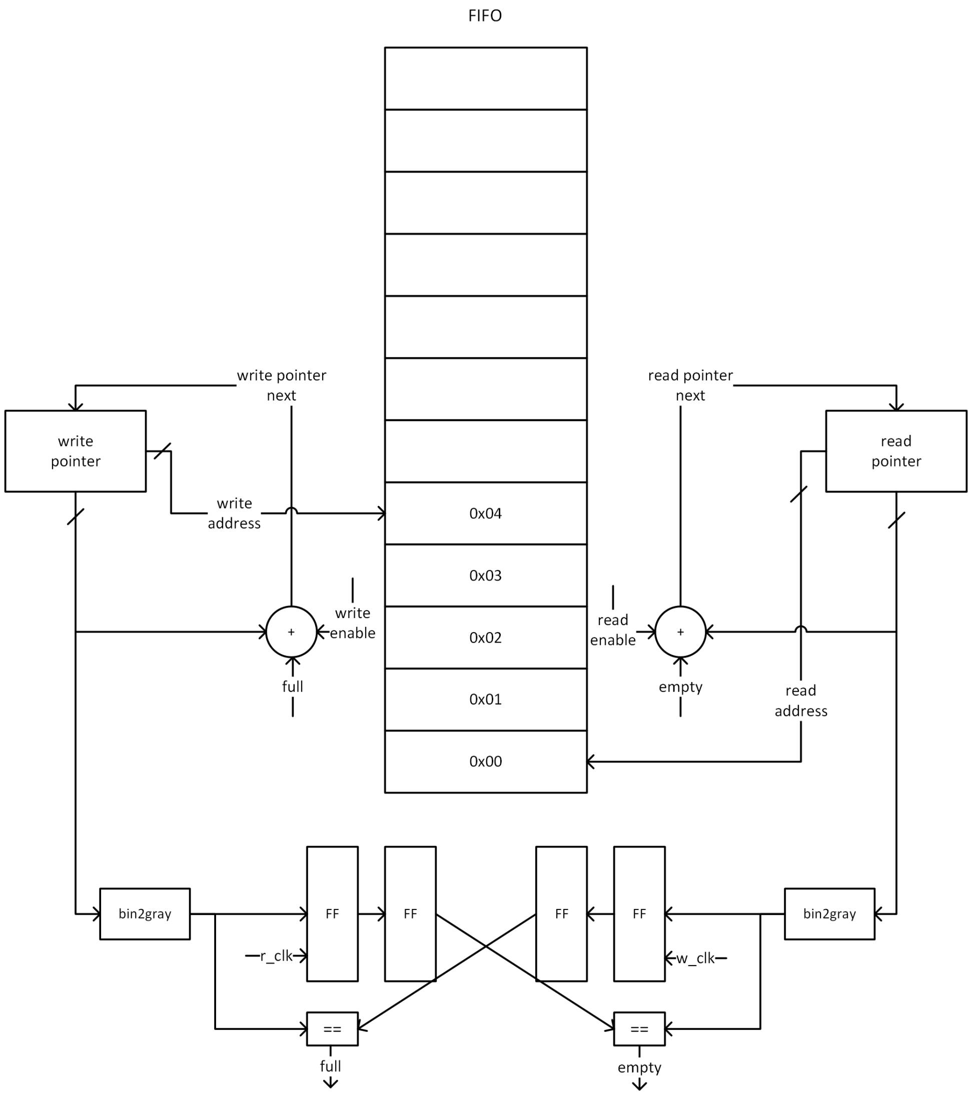


Figure 8.2: Asynchronous FIFO Block Diagram

8.2 Application and Purpose

The need for an Asynchronous FIFO Queue arises when we are dealing with systems with different data rates. For the rate of data flow being different, we will be needing an Asynchronous FIFO to synchronize the data flow between the systems. The main work of an Asynchronous FIFO is to pass data from one clock domain to another clock domain. Our UART module is fully compatible with the ibex-demo-system and fully complies with the the RISC-V instruction set architecture without the need for this extension. However, with the rapid evolution of integrated circuits and the growing demand for multi-clock domain designs, incorporating an Asynchronous FIFO became crucial to enhance the versatility and cross-compatibility of our UART module, enabling it to seamlessly integrate with more sophisticated and state of the art designs.

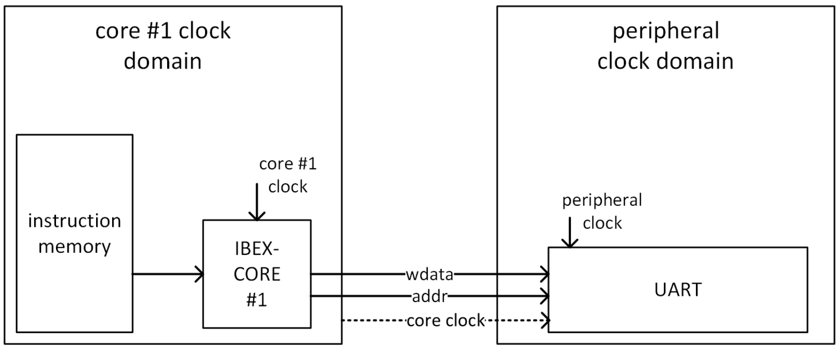


Figure 8.3: UART Integration on Different Clock Domains

8.2.1 Implementation, Integration and Design Changes

To make the option for an asynchronous FIFO queue available while maintaining a simple and flexible design, several changes needed to be made.

Firstly a header file was created to include necessary dependencies and define the **ASYNC** macro. By using the preprocessor, the top module's signals were modified accordingly to create instances of either synchronous or asynchronous FIFO components.

The resulting implementation includes the multi clock domain functionality without introducing significant architectural or structural complexities.

```
`ifndef UART
`define UARTS
`include "sync_fifo.v"
`include "async_fifo.v"
`include "uart_rx.v"
`include "uart_tx.v"

// If need be to use asynchronous FIFOs, uncomment.
`define ASYNC
`endif
```

Listing 8.1: UART Header File

Chapter 9

Conclusion and Results

9.1 Implementation

Throughout this thesis, we have explored the UART communication protocol and its circuit. We began by obtaining a solid understanding of the basics of the UART communication protocol, including the format of data and error detection mechanisms.

Then we analyzed the architecture of our circuit, examining its registers, signals, and components in hierarchical order and evaluating its significance in the whole circuit. A hierarchical design was adopted to contribute to the modules expandabiltiy and versatility. This approach created a simple and easy to understand design, covering an insignificant area on the arty7a FPGA. Comparing it to the original design, which is of monolithic architecture, provided by the lowrisc ibex demo system, our circuit manages to cover the same FPGA area while allowing for configurable protocol parameters, a feature missing from the original design.

Next, we shifted our focus to the software side, analyzing the microcode and the C software routines and functions that enable the configuration and efficient utilization of our UART module. We successfully integrated our design into the ibex demo system and performed exhausting functional verification through simulation to ensure its reliability. Additionally, we evaluated the code coverage metrics of our testing approach, ensuring extensive testing coverage.

Recognizing the significance of multi-clock domain designs, we explored the importance of asynchronous FIFO queues in synchronizing data flow between systems with different data rates, we discussed the implementation and seamless integration of these queues into our UART peripheral, effectively adding even more functionality to an already popular circuit by essentially being the only design that allows for the peripheral and the processor to belong to different clock domains published on github.

In summary, this thesis has provided a thorough explanation of the UART module, effectively covering both hardware and software aspects of an advanced UART peripheral that offers for reliable serial communication.

9.2 Further Work

A project is never truly complete, there is always room for improvement, and that's the beauty of the process when building something. Having accepted that, let's discuss some further features that could be considered future work for our project.

1. Oversampling of input rx to enhance noise tolerance over an unreliable channel with low SNR (signal to noise ratio).

2. Integration of a DMA (Direct Memory Access Controller) to improve the response speed of the module without overloading the data bus when loading data from memory to be transmitted or storing received data to the memory.

3. Adding even more interrupts to improve processor signaling regarding error detection.

4. Handshake mechanism for the modules to decide over the values of the protocol's parameters automatically.

5. Developing a low power mode. We approached this issue by adding two "en" registers that having their value changed from logic 1 to logic 0 disable the state machines of the modules and lock them on IDLE mode along with their corresponding FIFO queues ignoring any incoming bits. This is a very simple approach that leaves a lot of space for improvement on the design.

Bibliography

[1] anysilicon.com. clock-domain-crossing-cdc. `https://anysilicon.com/clock-domain-crossing-cdc/`, 2021, (accessed January, 2023). [Online; *anysilicon.com*].

[2] Arm. Rm0433 reference manual. `https://www.st.com/resource/en/reference_manual/dm00314099-stm32h742-stm32h743-753-and-stm32h750-value-line-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf`, 2021, (accessed October 2022). Online; *st.com*.

[3] lowrisc.org. Load store unit. `https://ibex-core.readthedocs.io/en/latest/03_reference/load_store_unit.html`, 2018, (accessed November, 2022). [Online; *ibex-core.readthedocs.io*].

[4] GregAC marnovandermaas. Readme. `https://github.com/lowRISC/ibex-demo-system/blob/main/README.md`, 2022, (accessed February 2023). GitHub; *github.com*.

[5] maven silicon.com. clock-domain-crossing. `https://www.maven-silicon.com/blog/clock-domain-crossing/`, 2022, (accessed January, 2023). [Online; *maven-silicon.com*].

[6] Rohde-Schwarz. Understanding-uart. `https://www.rohde-schwarz.com/us/products/test-and-measurement/essentials-test-equipment/digital-oscilloscopes/understanding-uart_254524.html#:~:text=UART%20stands%20for%20universal%20asynchronous,also%20have%20a%20ground%20connection.`, Not mentioned, (accessed November, 2022). [Online; *https://www.rohde-schwarz.com/*].

[7] Philipp Wagner. An update on ibex our microcontroller class cpu core. `https://lowrisc.org/blog/2019/06/an-update-on-ibex-our-microcontroller-class-cpu-core/`, 2019, (accessed November, 2022). [Online; *lowrisc.org*.

[8] Wikipedia. Universal asynchronous receiver-transmitter. `https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter`, 2002, (accessed October, 2022). [Online; *https://en.wikipedia.org/*].