BUILDING SYNTHESIZABLE MODEL AND VERIFICATION ENVIRONMENT FOR AXI4-STREAM FIFO

By
Ahmed Mohamed Yassien
Aly Essam Aly
Arwa Abdelaziz Abdelghany
Aya Mahmoud Mahmoud
Salma Khaled Hanafy
Shadwa Mohsen Abdellatif

Under the Supervision of Associate Prof. Hassan Mostafa

A Graduation Project thesis submitted to Faculty of Engineering, Cairo University in Partial Fulfilment of the Requirements for the Degree of Bachelor of Science in Electronics and Electrical Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
JULY 2019

Abstract

Streaming is a way of sending data from one block to another. The idea on streaming devices is to provide a steady flow of high speed data and to reduce overhead by using point to point connections which eliminate the need of addressing. In order to allow memory mapped access to an AXI4-Stream interface, the LogiCORE™ IP AXI4-Stream FIFO core is needed.

The core can be used to interface to AXI Streaming IPs, Similar to the LogiCORE IP AXI Ethernet core without the need to use a full Direct Memory Access (DMA) solution. The principal operation of this core allows the write or read of data packets to or from a device without any concern over the AXI Streaming interface. The AXI Streaming interface is transparent to the user.

In this thesis, the design, synthesis and verification using Universal Verification Method (UVM) of the AXI4-Stream FIFO core (Vivado design suite) are discussed.

Keywords: AXI4-Stream FIFO core, Universal Verification Methodology

Acknowledgments

We are using this opportunity to express our gratitude to everyone who supported us throughout the graduation project. We are thankful for their aspiring guidance and friendly advice.

First, we want to thank our major advisor Dr. Hassan Mostafa for his encouragement through the whole year, his caring about following up of each stage in the project and his suggestions to solve some problems we faced during the project work.

We want to thank Mentor Graphics team; Eng. Wael Mahmoud and Eng. Haytham Shoukry, QA managers, Eng. Amr Abbas, QA team leader, Eng. Ziad Abdelati, QA Engineer and finally Eng. Michael Hany, Research assistant at ONE lab at Cairo University, for providing their time, experience to help us overcome some obstacles we faced during some stages especially when dealing with new concepts and tools.

Finally, we want to thank our families for their support, tolerance and love during this year especially during the hard times they were always there having faith in what we do. We are grateful to our families, colleagues and friends for always motivating us, without them we wouldn't have come so far.

Table of Contents

Abstract ii
Achnowledgements..... iii
Table of Contents.....iv
List of Figures vi
List of Tables viii
List of Acronyms ix

Chapter 1.Introduction 1
1.1.Motivation 1
1.2.Problem Statement 3
1.3.Solution Approach 4
1.4.Organization 5
Chapter 2.Background 6
2.1.AXI4 Protocol 6
2.2.AXI4 – Lite Protocol.....9
2.3.AXI4 – Stream 9
Chapter 3.Vivado AXI4 Stream FIFO Design.....11
3.1.Block Introduction 11
3.2.Design Overview.....13
3.2.1.Transmit Path 14
3.2.2.Receive Path 15
3.3.Individual Building Blocks 16
3.3.1.AXI4-Lite Interface 16
3.3.2.AXI4 Interface 24
.3.3.3Register Space 35
3.3.4.Transmit Control 44
.3.3.5Transmit FIFO Unit 52
3.3.6.Stream Receive Interface 75
.3.3.7Receive FIFO 81
.3.3.8Receive Control 86
3.3.9.Calculation Unit 99

3.3.10.Interrupt Interface 105
Chapter 4.FPGA Synthesis and Implementation 108
4.1.Introduction 108
4.2.Results 109
4.2.1.Synthesis Results 109
4.2.2.Implementation Results 111
Chapter 5.Verification 112
5.1.Introduction 112
5.2.Direct Testing 112
.5.2.1Unit Testing 113
5.2.2.Integration Testing 115
5.3.Universal Verification Methodology (UVM) 127
5.3.1.Introduction 127
5.3.2.Overview 129
5.3.3.UVM Sequences 133
Chapter 6.Simulation Results 138
6.1.Transmit Path 138
6.2.Receive Path 143
Chapter 7.Conclusion and Future Work 146
7.1.Conclusion 146
7.2.Future Work 147
7.2.1.Completion of UVM testing 147
7.2.2.Verification using equivalence checking 147
7.2.3.Verification using Questa Verification IP 147
Chapter 8.References 148

List of Figures

Figure 2-1 VALID before READY handshake 7
Figure 2-2 READY before VALID handshake 7
Figure 3-1 Schematic of Vivado AXI4-Stream FIFO 13
Figure 3-2 Transmit Path 14
Figure 3-3 Receive Path 15
Figure 3-4 AXI4_Lite Write Operation Flow Chart 19
Figure 3-5AXI4-Lite Read Operation Flow Chart 20
Figure 3-6 Block Diagram of AXI4-Lite 21
Figure 3-7 write operation FSM of AXI4-lite 22
Figure 3-8 Read Operation FSM of AXI4-Lite 23
Figure 3-9AXI4 Write tranasaction FlowChart 29
Figure 3-10 AXI4 Read transaction Flow Chart 30
Figure 3-11AXI4 block diagram 31
Figure 3-12Write Operation FSM of AXI4 32
Figure 3-13 AXI4 Read Operation FSM 34
Figure 3-14 Interrupt Service Register (offset 0x00) 37
Figure 3-15 Interrupt Service Register (offset 0x00) 38
Figure 3-16Transmit Data FIFO Reset Register (offset 0x8) 39
Figure 3-17 Transmit Data FIFO Vacancy Register (offset 0xC) 39
Figure 3-18Transmit Data FIFO Data Write Port (offset 0x10) 39
Figure 3-19 Receive Data FIFO Reset Register (offset 0x18) 40
Figure 3-20 Receive Data FIFO Occupancy Register (offset 0x1C) 40
Figure 3-21 Receive Data FIFO Data Write Port (offset 0x20) 40
Figure 3-22 Transmit Length Register (offset 0x14) 41
Figure 3-23 Receive Length Register (offset 0x14) 41
Figure 3-24 AXI4-Sream Reset Register (offset 0x28) 42
Figure 3-25 Transmit Destination Register (offset 0x2C) 42
Figure 3-26 Receive Destination Register (offset 0x30) 42
Figure 3-27 Register Space block diagram 43
Figure 3-28 Transmit Control functionality flow chart 48
Figure 3-29 Transmit Control block diagram 49
Figure 3-30 Transmit Control FSM 50
Figure 3-31 Detecting Next position read/write flow chart 56
Figure 3-32 Check Full and Empty of FIFO flow chart 57
Figure 3-33Write /Read operation in Transmit FIFO flow chart 58
Figure 3-34 Store and forward mode flow chart 61
Figure 3-35 Cut-through mode flow chart 62
Figure 3-36 FIFO and Stream Mapper block 63
Figure 3-37 Transmit Data FIFO 64
Figure 3-38 Store and forward mode finite state machine 65
Figure 3-39 Cut-through mode finite state machine 66
Figure 3-40 Stream Mapper finite state machine 67
Figure 3-41 AXI4-Stream interface flow chart 78

Figure 3-42Receive AXI4 Stream block 79
Figure 3-43 Receive AXI4 Stream FSM 79
Figure 3-44Receive FIFO block 85
Figure 3-45 Receive Control functionality flow chart 94
Figure 3-46 Receive Control FSM of both store and forward and cut-through modes 96
Figure 3-47 TDFV functionality flow chart 101
Figure 3-48 RDFO Functionality flow chart 102
Figure 3-49 Calculation Unit block diagram 103
Figure 3-50 Interrupt Interface 107
Figure 4-1Total utilization of board resources (Post-Synthesis) 109
Figure 4-2Total utilization of board resources for each module (Post-Synthesis) 110
Figure 4-3 Number of bonded IOB in design (post-synthesis) 110
Figure 4-4Total Utilization of board resources (Post-Implementation) 111
Figure 4-5Utilization of board resources for each module (Post-Synthesis) 111
Figure 5-1UVM Architecture 129
Figure 6-1 Register Space and Control Unit simulation results1 138
Figure 6-2 Stream Interface simulation results1 138
Figure 6-3 Register Space and Control Unit simulation results 2 139
Figure 6-4 Stream Interface simulation results 2 139
Figure 6-5 Register Space and Control Unit simulation results 3 140
Figure 6-6 Stream Interface simulation results 3 140
Figure 6-7 Transmit path simulation result 4 141
Figure 6-8 Transmit Path simulation result 5 142
Figure 6-9 Writing and reading random packets 143
Figure 6-10 First word written to the core 143
Figure 6-11 RC bit 26 goes high after TLAST 144
Figure 6-12 First word written to the core 144
Figure 6-13 First word read from the core 145
Figure 6-14 RC bit 26 goes high after TLAST 145

List of Tables

Table 2-1 Supported Signals by AXI4 8
Table 2-2 Supported signals by AXI4 - Lite 9
Table 2-3 Supported Signalsby AXI4-Stream..... 10
Table 3-1Parameters that can be configured in design..... 12
Table 3-2AXI4-Lite I/O Signals..... 18
Table 3-3Write Interface States Illustration 22
Table 3-4Read Operation States Illustration 23
Table 3-5AXI4 I/O signals..... 26
Table 3-6Write operation states illustration 32
Table 3-7AXI4 Write Interface FSM transition conditions illustration 33
Table 3-8AXI4 write operation FSM outputs..... 33
Table 3-9AXI4 Read operation states 34
Table 3-10Register Space I/O signals 36
Table 3-11Interrupt Service Register structure 38
Table 3-12Transmit Control I/O signals..... 45
Table 3-13Transmit Control FSM states illustration..... 50
Table 3-14Transmit Control FSM conditions 51
Table 3-15Trasmit Contro FSM outputs 51
Table 3-16Transmit FIFO Unit I/O signals..... 54
Table 3-17 Stream Mapper FSM states..... 65
Table 3-18 Receive AXI4 Stream I/O signals..... 76
Table 3-19Receive AXI4 Stream FSM states..... 79
Table 3-20Receive AXI4 Stream FSM conditions..... 80
Table 3-21 Receive AXI4 Stream FSM outputs..... 80
Table 3-22Receive FIFO I/O signals..... 82
Table 3-23Receive Control I/O signals 87
Table 3-24 Receive Control FSM states..... 95
Table 3-25Receive Control FSM transition conditions..... 97
Table 3-26 Receive Control FSM outputs..... 98
Table 3-27Calculation Unit I/O signals..... 100
Table 3-28 Interrupt Interface I/O signals..... 106

List of Acronyms

AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
DMA	Direct Memory Access
DSP	Digital Signal Processing
DUT	Design Under Test
eRM	e Reuse Methodology
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IEEE	Institute of Electrical and Electronics Engineers
IER	Interrupt Enable Register
IP	Intellectual property
ISR	Interrupt Status Register
OVM	Open Verification Methodology
PAR	Place and Route
RC	Receive Complete
RDFD	Receive Data FIFO Data Register
RDFO	Receive Data FIFO Occupancy Register
RDFR	Receive Data FIFO Reset Register
RDR	Receive Destination Register
RFPE	Receive FIFO Programmable Empty
RFPF	Receive FIFO Programmable Full
RLR	Receive Length Register
RPORE	Receive Packet Overrun Read Error

RPUE	Receive Packet Underrun Error
RPURE	Receive Packet Underrun Read Error
RRC	Receive Reset Complete
RTL	Register Transfer Logic
RX	Receive
SoC	System-on-Chip
SRR	AXI4-Stream Reset Register
TC	Transmit Complete
TDFD	Transmit Data FIFO Data Write Port
TDFV	Transmit Data FIFO Vacancy Register
TDR	Transmit Destination Register
TFDR	Transmit FIFO Data Register
TFPE	Transmit FIFO Programmable Empty
TFPF	Transmit FIFO Programmable Full
TLM	Transaction-Level Modeling
TLR	Transmit Length Register
TPOE	Transmit Packet Overrun Error
TRC	Transmit Reset Complete
TSE	Transmit Size Error
TX	Transmit
UUT	Unit Under Test
UVM	Universal Verification Methodology

Chapter 1.
Introduction

In this thesis, we are going to design and verify using UVM the AXI4 Stream FIFO that deals with some of AMBA 4.0 protocols such as AXI4, AXI4-Lite and AXI4-Stream.

1.1. Motivation

The Advanced Microcontroller Bus Architecture (AMBA) protocol is a registered trademark of ARM Limited. It's an open standard, on-chip interconnect requirement for the connection and management of functional blocks in a System-On-Chip (SoC). It facilitates easy implementation of different macro functions operating at different frequencies (High frequency), and also facilitates right-first-time development of multi-processor designs with large numbers of controllers and peripherals. The AMBA protocol is technology independent, since it is implemented for any operating frequency range [1].

The scope of AMBA has gone far beyond microcontroller devices, and is at the present widely used on a range of ASIC and SoC parts including applications processors used in modern portable mobile devices like Smartphones.

AMBA was introduced by ARM Ltd in 1996. The first AMBA buses were highly developed Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). In its 2nd version in 1999, AMBA 2.0, ARM added AMBA High-performance Bus (AHB) that is a single clock-edge procedure. In 2003, ARM introduced the 3rd generation, AMBA 3.0, with Advanced Extensible Interface (AXI) to reach even high performance inter-connects. In 2010, ARM introduced the 4th generation, AMBA 4.0, these includes the second version of AXI, AXI4.

AXI4 provides improvements and enhancements to any Intellectual Property (IP) as it provides benefits to productivity, flexibility and availability [2]:

- **Productivity:** Developers need to learn only a single protocol for IP by standardizing on the AXI interface.
- **Flexibility:** There are three types of AXI4 which depends on the application:
    1. AXI4 is for memory mapped interfaces and allows burst up to 256 data transfer cycles with just a single address phase.
    2. AXI4-Lite is a light-weight, single transaction memory mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage.
    3. AXI4-Stream is for high streaming data that can burst an unlimited amount of data.
- **Availability:** Many IP providers support the AXI protocol.

1.2. Problem Statement

There are a lot of differences between AXI4, AXI-Lite and AXI4-Stream as AXI4-Stream removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped, unlike AXI4 and AXI4-Lite that are used for memory mapped interfaces.

Both AXI4 and AXI4-Lite interfaces consist of five different channels: Write Address Channel, Write Data Channel and Write Response Channel for Write transactions. Read Address Channel and Read Data Channel for Read transactions. Data can move in both directions between the master and the slave simultaneously, and data transfer sizes can vary. The limit in AXI4 is a burst transaction of up to 256 data transfers but AXI4-Lite allows only one data transfer per transaction as bursting is not supported.

The AXI4-Stream protocol defines a single channel for transmission of streaming data. The AXI4-Stream channel is modeled after the Write Data channel of the AXI4. Unlike AXI4, AXI4-Stream interfaces can burst an unlimited amount of data. There are additional, optional capabilities described in AXI Stream Specification [3]. The specification describes how AXI4-Stream-compliant interfaces can be split, merged, interleaved, upsized, and downsized. Unlike AXI4, AXI4-Stream transfers cannot be reordered.

The AXI4-Stream interface also supports a wide variety of different stream types such as: Transfer which is a single transfer of data across an AXI4-Stream interface, Packet which is a group of bytes that is similar to an AXI4 burst and Frame which is the highest level of byte grouping in an AXI-Stream that contains an integer number of packets unlike AXI4 that supports only bursts.

As the transactions are different between AXI4/AXI4-Lite and AXI4-Stream, the communication between these blocks would be difficult if they are in the same design unless there is an intermediate block to facilitate communication between them.

Note: Further information about each protocol will be discussed in chapter2.

1.3. Solution Approach

If the communication is required between AXI4/AXI4-Lite and AXI4-Stream, Conversion of transactions between these interfaces (in both directions) is essential to make communication between them possible. At this point, the need for an intermediate block arises to make this mission, which is the role of AXI4-Stream FIFO.

AXI4 Stream FIFO converts from AXI4/AXI4-Lite transactions to AXI4-Stream transactions in one direction which is called the transmit path and converts from AXI4-Stream transactions to AXI4/AXI4-Lite transactions in the other direction which is called the receive path. These two paths are completely independent.

AXI4 Stream FIFO also allows memory mapped access to an AXI-Stream interface. The principal operation is to allow the write or read of data packets to or from a device without any concern over the AXI-Stream interface signaling. The AXI-Stream interface is transparent to the user.

1.4. Organization

The following chapters discuss Vivado AXI4-Stream architecture, the AXI4 protocols, the design blocks and its implementation, and testing of the design. The remainder of this thesis is organized as follows:

Chapter 2 provides background information on AXI protocols such as AXI4, AXI4-Lite and AXI4-Stream, discusses its architecture and the most important signals used.

Chapter 3 discusses Vivado AXI4-Stream FIFO design, provides information about the blocks and its implementation, block diagrams, flow charts and finite state machines that explain the functionality of each block in the transmit and receive paths.

Chapter 4 provides results of the synthesis and implementation of Vivado AXI4-Stream FIFO using Vivado suite showing the number of resources utilized on the chosen kit which is XCVU440-FLGA2892-1-C.

Chapter 5 discusses Verification part as it includes direct testing on each block and integration testing on the whole design then constrained random testing using Universal Verification Methodology (UVM) with further explanation of the blocks used in UVM and sequences that have been tested on transmit and receive paths.

Chapter 6 provides the results of the simulation of the whole design after constrained random testing using UVM on Questa-Sim (Behavioral Simulation).

Chapter 7 provides a brief overview of the findings, draws conclusions, and recommends directions for future work.

Chapter 2.
Background

As the AXI4-Stream FIFO core deals with AXI4/AXI4-Lite and AXI4-Stream interfaces, and each of them uses a different protocol then to fully understand the functionality of the core these protocols must be illustrated first in more details before getting into the design process.

In this chapter, the important general concerns in each protocol are illustrated such as the handshaking procedure, important features, etc. and further information can be found in each protocol standard mentioned in the References.

2.1. AXI4 Protocol

The AXI4 protocol is a transaction based protocol which relies on the handshake of the VALID and READY signal to complete any data or information transfer, a transaction is defined as the process of exchanging of the complete set of required information between master and slave.

It is also a burst-based which means it allows the transfer of multiple words of data in one transaction unlike the AXI4 – Lite which will be discussed later, the supported burst types are as follows:

The AXI4 protocol defines five independent channels, two for read transactions and three for write transactions which are as follows:

- Write Data Channel
- Write Response Channel
- Read Address Channel
- Read Data Channel
- Write Address Channel

Where Address channels are responsible for carrying all required signals and control information to define the transaction, while the Data channels carry the data to be exchanged between source and destination. The Write Response channel carries the response of the destination to the write transaction whether the write was a success or an error took place.

The AXI4 relies on the handshake between the VALID and READY signals between the source and destination as mentioned above, the handshake is explained as follows:

The VALID signal is generated by the source which indicates the presence of valid control information or data on the bus to the destination.

The READY signal is generated by the destination to indicate that it is ready to accept information at this moment, the transfer only takes place when both the VALID and READY signals are HIGH as shown in Figure 2-1 and 2-2.

As shown, there is no restriction on which of the two signals comes high first except for only one constraint, that the source is not allowed to wait for the READY signal to be high to assert its VALID signal, and for the READY signal, the destination can either wait for the VALID signal to be high to assert the READY or have the READY high whenever it's ready to receive information which saves one clock cycle.

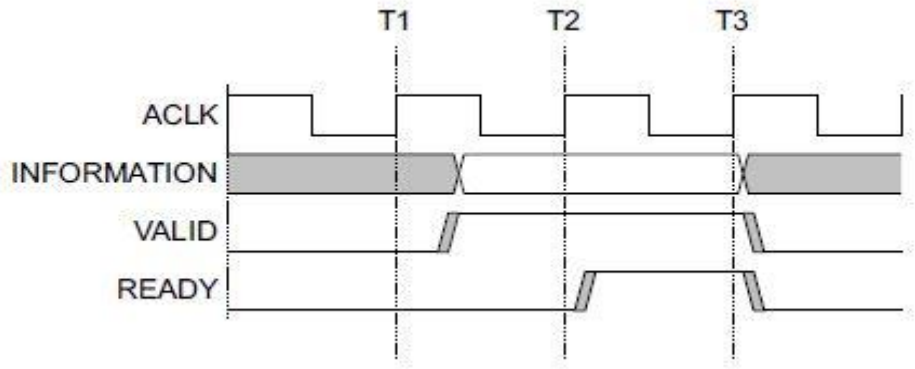


Figure 2-1 VALID before READY handshake

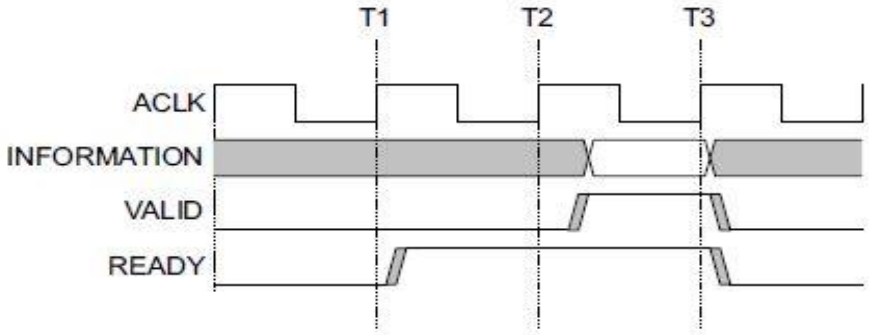


Figure 2-2 READY before VALID handshake

The Signals supported by the AXI4 are as follows in Table 2-1

Write address channel	**Read address channel**	**Write data channel**	**Read data channel**	**Write response channel**
AWADDR	ARADDR	WDATA	RDATA	BRESP
AWLEN	ARLEN	WSTRB	RRESP	BVALID
AWSIZE	ARSIZE	WLAST	RLAST	BREADY
AWBURST	ARBURST	WVALID	RVALID	
AWVALID	ARVALID	WREADY	RREADY	
AWREADY	ARREADY			

Table 2-1 Supported Signals by AXI4

The following signals are what define the burst properties of the transaction

AxBURST: Determines the burst type of the transaction

AxSIZE: Determines the number of transferred bytes in each data transfer in a burst

AxLEN: Determines the number of data transfers in each transaction

Note: The “x” means that it could be for read or write channel, as an example AxBURST can be ARBURST or AWBURST.

The AXI4 protocol supports strobing on write transactions where each bit in the WSTRB signal corresponds to one byte of the WDATA (Write Data) bus, if the bit is pulled high it indicates the corresponding byte in the WDATA bus carries valid data to be stored otherwise the data on the corresponding byte is invalid, such feature is useful in unaligned memory access and in narrow bus width transfers.

2.2. AXI4 – Lite Protocol

Unlike AXI4, The AXI4 – Lite supports transactions of burst length of one only, thus the AxBurst, AxSIZE, AxLEN and LAST signals are not supported. The AXI4 – Lite supports the same channels and the same handshake process as the AXI4 protocol.

The Signals supported by the AXI4 - Lite are as follows in Table 2-2

Write address channel	**Read address channel**	**Write data channel**	**Read data channel**	**Write response channel**
AWADDR	ARADDR	WDATA	RDATA	BRESP
AWVALID	ARVALID	WSTRB	RRESP	BVALID
AWREADY	ARREADY	WVALID	RVALID	BREADY
		WREADY	RREADY	

Table 2-2 Supported signals by AXI4 - Lite

2.3. AXI4 –Stream

The AXI4–Stream is suitable to continuous transfer of data with low complexity, unlike the two previous protocols it only supports one channel, it also depends on the same handshake procedure between the TVALID and TREADY signals, where each time both the TVALID and TREADY are asserted together, an exchange of a beat or word of the packet to be transmitted can take place, the TLAST signal marks the packet boundary meaning that at the last beat of a packet the TREADY, TVALID and TLAST are all high.

The following byte definitions are used in this specification:

Data byte: A byte of data that contains valid information that is transmitted between the source and destination.

Position byte: A byte that indicates the relative positions of data bytes within the stream. This is a placeholder that does not contain any relevant data values that are transmitted between the source and destination.

Null byte: A byte that does not contain any data information or any information about the relative position of data bytes within the stream.

TSTRB signal acts as the WSTRB signal in the AXI4 protocol, where each bit of the TSTRB corresponds to one byte of the TDATA, if the bit is pulled high it indicates the corresponding byte in the TDATA bus is a Data byte to be stored otherwise the data on the corresponding byte is considered position byte (control data).

However, TSTRB is in effect only when TKEEP is asserted, otherwise if the TKEEP is low the corresponding byte is considered to be a NULL byte and must be removed from the stream bytes by the interconnect if the slave can't handle the NULL bytes.

The Signals supported by the AXI4 -Stream in our design are shown in Table 2-3

Signal	Description
TDATA	Carries the data transmitted from the source to destination
TVALID	Generated from the source to indicate the presence of valid data on the bus
TREADY	Generated from the destination to indicate that it's ready to receive data at this point
TLAST	Generated from the source to mark the packet boundary
TKEEP	A byte qualifier used to indicate whether the content of the associated byte must be transported to the destination.
TSTRB	A byte qualifier used to indicate whether the content of the associated byte is a data byte or a position byte.

Table 2-3 Supported Signalsby AXI4-Stream

Chapter 3.
Vivado AXI4 Stream FIFO Design

3.1.Block Introduction

The AXI4-Stream FIFO v4.1 (Vivado Design Suite) is a logic core IP owned by Xilinx which is used to convert AXI4/AXI4-lite transactions to and from AXI4-Stream transactions, which is useful for interfacing between processors and data flow applications as DSP applications.

Features supported by the core

- Independent TX and RX paths
- Simultaneous operations for both Transmit and Receive paths
- Two modes of operation: Store and Forward and Cut-Through
- Configurable FIFO depth
- Two Data interfaces for the user either AXI4 or AXI4-Lite
- Interrupts to show status of the core and occurrence of errors

Modes of operation

The core supports two modes of operation

- **Store and Forward Mode**
  In store and forward mode, the core must wait for the packet to be fully stored and its length of the full packet is written to the core before the start of packet transmission whether it was on TX or RX path, in this case the FIFO must be large enough to hold the complete packet.

- **Cut-Through Mode**

  In Cut-Through Mode transmission behavior changes depending on whether the packet is being transmitted on the TX or RX path as follows:

  In case of TX: The packet transmission starts when the Transmit FIFO is not empty however the last beat is transmitted only after the length of the whole packet is written to the core

  In case of RX: The packet transmission starts as soon as a part of the packet is written into the Receive FIFO, meanwhile the packet length is stored and updated accordingly with total partial packet length, however last part of the packet is not transmitted till the length of the full packet is updated in the core

  The FIFO doesn't need to be large enough to hold the complete packet

Interfaces

The Core offers AXI4 – Lite interface or AXI4 interface for the user to use to write or read data from the Core.

The AXI4 – Lite interface is mandatory for register access even if AXI4-interface option is selected to deal with FIFO access.

This means that if AXI4 – Lite is selected then it has full access to all the registers in the register space, while if AXI4 option is selected then the AXI4 – Lite interface has access to all the registers in the register space except for the two registers responsible of writing and reading from the FIFOs while the AXI4 interface has access to those two registers only. Further information about these registers will be explained in details in section 3.3.3.

Parameters

	Parameter Name	**Default value**
AXI4 – Lite data width	C_S_AXI_DATA_WIDTH	32
RX FIFO Depth	C_RX_FIFO_DEPTH	512
TX FIFO Depth	C_TX_FIFO_DEPTH	512
AXI4 Data Width	C_S_AXI4_DATA_WIDTH	32
AXI4 - Stream Data Width	C_S_S_AXI4_DATA_WIDTH	32
Base Address for the Register Space	C_BASEADDR	0
Full threshold for the FIFOs (Amount of locations left before the FIFO is considered Full)	full_threshold_data	0
Empty Threshold for the FIFOs	empty_threshold_data	0
Mode of operation 1 for cut-through 0 for store and forward	enable_cut_through	0

Table 3-1Parameters that can be configured in design

Assumptions

The user is aware of the presence of the core and in order to interact with it the user needs to use correct addresses specified by the AXI4-Stream FIFO (Vivado Design Suit) to access each register via normal AXI4-Lite read or write transactions.

3.2. Design Overview

As mentioned in the features supported, the design is meant to be bidirectional which means that it has two independent paths. Both paths have nearly the same building blocks which are a Transmit/Receive control blocks, Transmit/Receive FIFOs and finally the Transmit/Receive Stream interfaces.

Clearly, the control blocks are monitoring the user's programming sequence and ensuring correct operation, while the FIFOs are the storing units for packets, and the stream interface is meant to communicate with Stream sides. The implementation of each path blocks is totally different as the Transmit and Receive logic are different. However, there may be some similar concepts between both paths such as FIFOs.

Each of the Transmit FIFO and Receive FIFO is split into 3 different FIFOs

- **Data FIFO:** The largest FIFO of them all and used to store the packets
- **Length FIFO:** Used to store the length of the packets in bytes
- **Destination FIFO:** Used to store destination of packets

The remaining block is the AXI4/AXI4-Lite interface which participates in both Receive and Transmit paths, as it has different channels for reading and writing operations and thus supporting full duplex operation.

In order to fully understand the core's correct operation and data flow from both interfacing sides, we need to separately study each path, which is illustrated in the upcoming two sections.

The schematic of the core design is shown in figure 3-1.

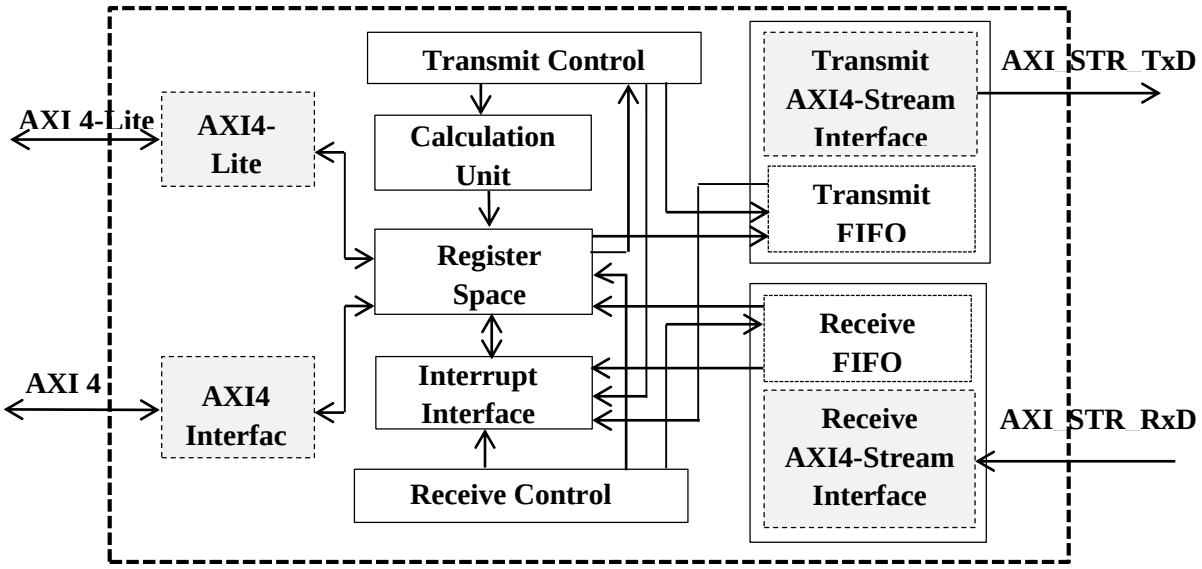


Figure 3-1 Schematic of Vivado AXI4-Stream FIFO

3.2.1. Transmit Path

Transmit path is responsible for storing AXI4 or AXI4–Lite data write transactions in the Transmit FIFO (through passing by the Register Space first) till read transactions are initiated on the AXI4 stream interface, this is done as follows:

1- Programming sequence is applied from the user to the AXI4-Lite interface, passing information about the packet to the corresponding registers as destination, length and the written data.
2- By the end of this process the packet should be fully stored in the transmit FIFO and the Vacancy register is updated with number of vacant locations available to be written by the upcoming packets.
3- The packet is then transmitted over the AXI4-Stream Transmit interface when a handshake is initiated by the "application" on the other end of the interface.

The Transmit control block is responsible for coordinating the transfer of written data by the user from the Register Space to the Transmit FIFO and is also responsible for monitoring the programming sequence to allow a correct operation and to detect any error in the programming sequence that requires a reset for the Transmit Logic.

In case of Cut-Through mode of operation, the data packets that are written from the AXI4/AXI4-Lite don't need to be completely stored in the Transmit FIFO to start the transmission of the data into the AXI4-Stream interface, meaning that the transmission can start as soon as the Transmit FIFO is not empty.

Transmit path is illustrated in figure 3-2, where packet path and blocks engaged in transmission operation are highlighted.

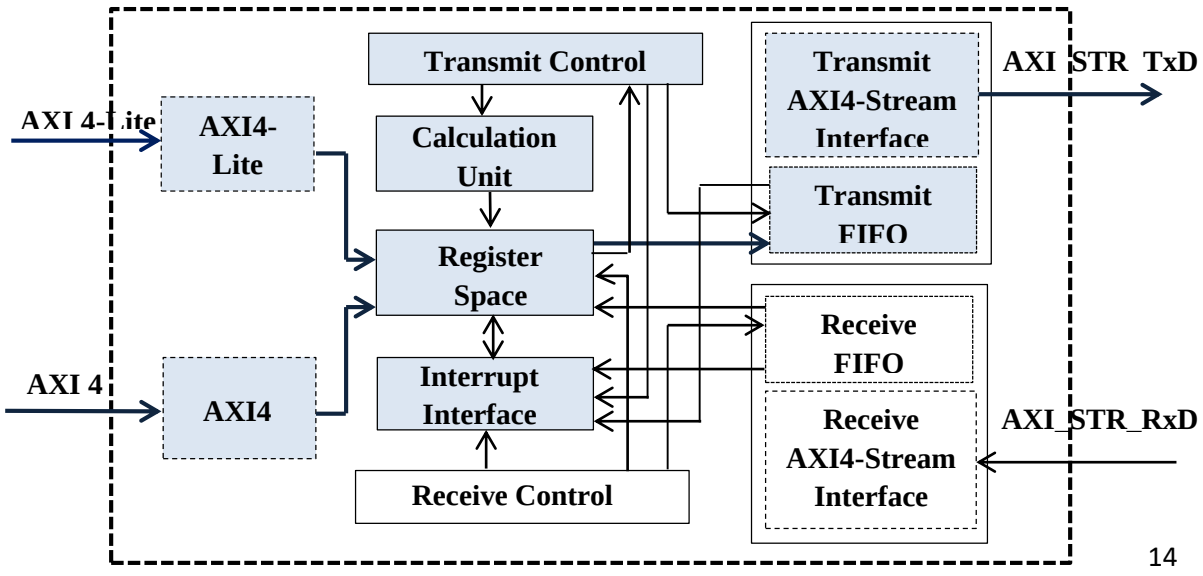


Figure 3-2 Transmit Path

3.2.2. Receive Path

The Receive path stores incoming data from the AXI4-Stream interface in the Receive FIFO till read transactions are initiated from the user on AXI4-Lite interface, that's when the data is transferred from the Receive FIFO into AXI4/AXI4-Lite interface through the Register space, this is done as follows:

1- Packet received via the AXI4-Stream Receive interface is written to the Receive FIFO, and the length of the packet is calculated and written to the length FIFO, same goes for the destination FIFO.
2- By the end of this process the packet should be fully stored in the Receive FIFO and the Occupancy register is updated with number of locations occupied by the last successfully received packet.
3- The packet is then transmitted over the AXI4/AXI4-Lite interface when a handshake is initiated by the user on the other end of the interface, passing by the Register Space (the data reaches the interface through the Register Space that the user access)

The Receive control block is responsible for coordinating the transfer of written data from the Receive FIFO to the Register Space and is also responsible for monitoring the programming sequence to allow a correct operation and to detect any error in the programming sequence that requires a reset for the Receive Logic.

In case of Cut-Through mode of operation, the data packets that are written from the AXI4 Stream side don't need to be completely stored in the Receive FIFO to start the transmission of the data into the AXI4/AXI4-Lite interface, meaning that the transmission can start as soon as the Receive FIFO is not empty.

Receive path is illustrated in figure 3-3 where packet path and block engaged in the reception operation are highlighted.

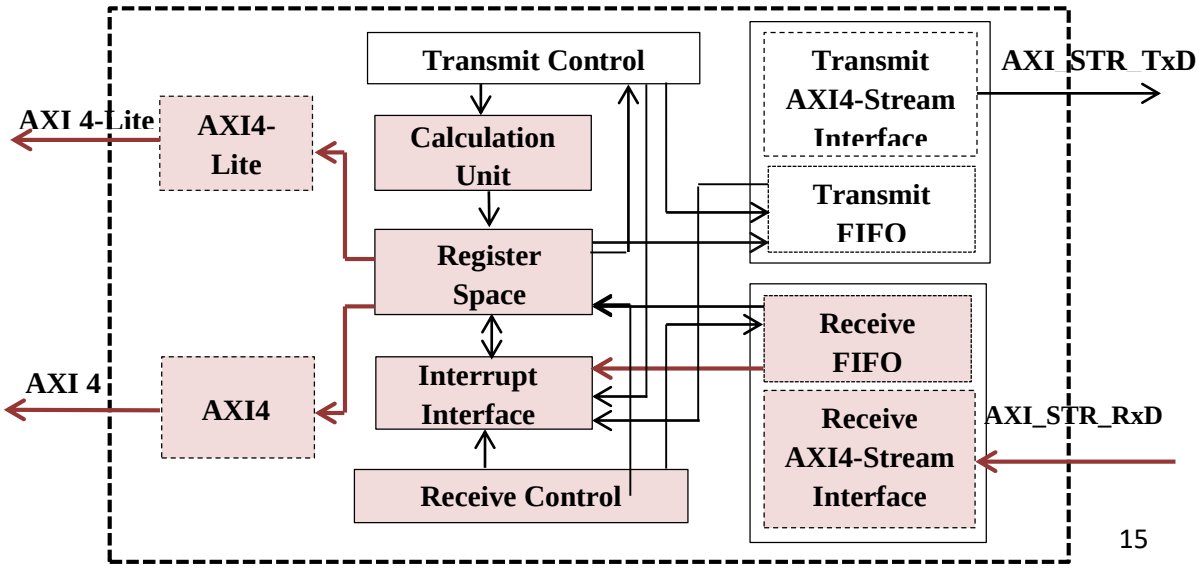


Figure 3-3 Receive Path

3.3. Individual Building Blocks

3.3.1. AXI4-Lite Interface

Description

AXI4-Lite interface is the main interface with the core, it provides independent read and write channels where transactions mainly take place on a two stage handshake, first defines the all the required signals and control information to define the transaction and takes place when both the **AxVALID** and **AxREADY** signals are pulled high at the same time, while the second passes the data from the source to destination which takes place when both **xVALID** and **xREADY** signals are high at the same time.

The AXI4-Lite protocol supports only transactions of burst length 1, meaning that each transaction whether it was read or write is responsible for delivering one word only from the source to the destination. The supported data bus width in the core is 32 bits.

All data accesses use the full bus width, however only bytes containing valid data is stored in write transactions. Valid data is indicated by the strobe signal where each bit corresponds to one byte of the data bus.

The AXI4-Lite interface mainly deals with the Register Space within the core where all read/write operations are passed to the Register Space via the interface except when the AXI4 interface is chosen for data access to the core, then the AXI4-Lite doesn't have access to the Receive Data FIFO Data Read Port (RDFD) nor the Transmit Data FIFO Data Write Port (TDFD) in the Register Space.

Interfacing

The AXI4-lite block contains I/O signals listed in Table 3-2.

Port name	Port mode	Connection	Description
Clk	Input	User	Global interface clock
AXI4_lite_ARESET	Input	User	Resets entire core
AXI4_lite_AWADDR	Input	User	Address of a write transaction
AXI4_lite_AWPROT	Input	User	Not used
AXI4_lite_AWVALID	Input	User	A valid write address is available
AXI4_lite_WDATA	Input	User	Data of a write transaction
AXI4_lite_WSTRB	Input	User	Indicates which byte lanes have valid data
AXI4_lite_WVALID	Input	User	A valid write data is available
AXI4_lite_BREADY	Input	User	Master can accept response information
AXI4_lite_ARADDR	Input	User	Address of a read transaction
AXI4_lite_ARPROT	Input	User	Not used
AXI4_lite_ARVALID	Input	User	A valid read address is available
AXI4_lite_RREADY	Input	User	Master can accept read data
AXI4_lite_AWREADY	Output	User	Slave can accept read information
AXI4_lite_WREADY	Output	User	Slave is ready to receive read data
AXI4_lite_BRESP	Output	User	Status of a write transaction
AXI4_lite_BVALID	Output	User	A valid write response is available
AXI4_lite_ARREADY	Output	User	Slave can accept read address
AXI4_lite_RDATA	Output	User	Data of a read transaction
AXI4_lite_RRESP	Output	User	Status of a read transaction
AXI4_lite_RVALID	Output	User	A valid read data is available
read_enable	Output	Register space	Read enable signal
write_enable	Output	Register space	Write enable signal
write_address	Output	Register space	Address of a write operation

Memory_address	Output	Register space	Address of a read operation
data_in	Output	Register space	Data of a write operation
data_out	Input	Register space	Data of a read operation
STROBE_OUT	Output	Transmit control	Indicates which byte lanes has valid data

Table 3-2AXI4-Lite I/O Signals

Functionality

The AXI4-Lite interface is responsible for read/write operations from/into Register Space; this is done through five channels:

- Write Address Channel: carries all required signals and control information to define a write transaction
- Write Data Channel: carries data written by master (source) into slave (destination)
- Write Response Channel: carries write response which gives information about write transaction statue.
- Read Address Channel: carries all required signals and control information to define a read transaction
- Read Data Channel: carries data to be read from slave (source) by master (destination)

The Write operation is done as follows

- The Address of the intended register to be written is carried out on the Write Address Channel, when the handshake between the AWVALID and AWREADY takes place, the core has read the address of the register to be written successfully
- Then the actual data is written on the Write Data Channel, when the handshake between the WVALID and WREADY takes place, this indicates that the core has read the data successfully
- The core then replies with a Response signal to feedback the write transaction statues and similarly BREADY and BVALID perform the handshake

The Read operation is done as follows

- The Address of the intended register to be read is written on the Read Address Channel, and the handshake between the ARVALID and ARREADY takes place, which indicates that the core has read the address of the register to be read successfully
- Then the data to be read is written from the core on the Read Data Channel, and the handshake between the RVALID and RREADY takes place, which indicates that the user has read the written data by the core successfully
- The RRESP signal is signaled from core to feedback the read transaction statues

Write operation in AXI4-lite is illustrated by the flow chart in figure 3-4

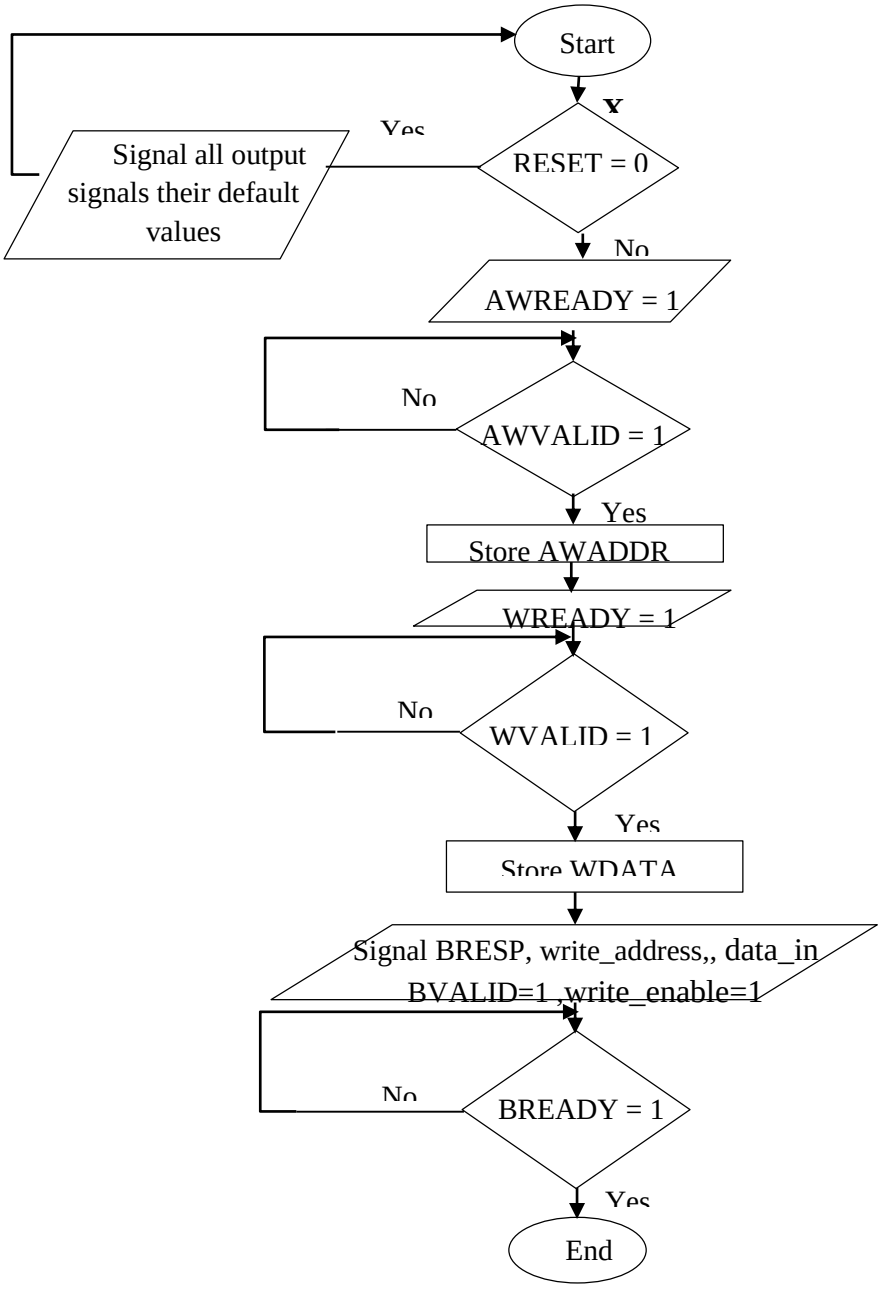


Figure 3-4 AXI4_Lite Write Operation Flow Chart

Note: Resetting AXI4-lite at any point will result in going to start point (marked **X** at flow chart)

Read operation in AXI4-lite is illustrated by the flow chart in figure 3-5

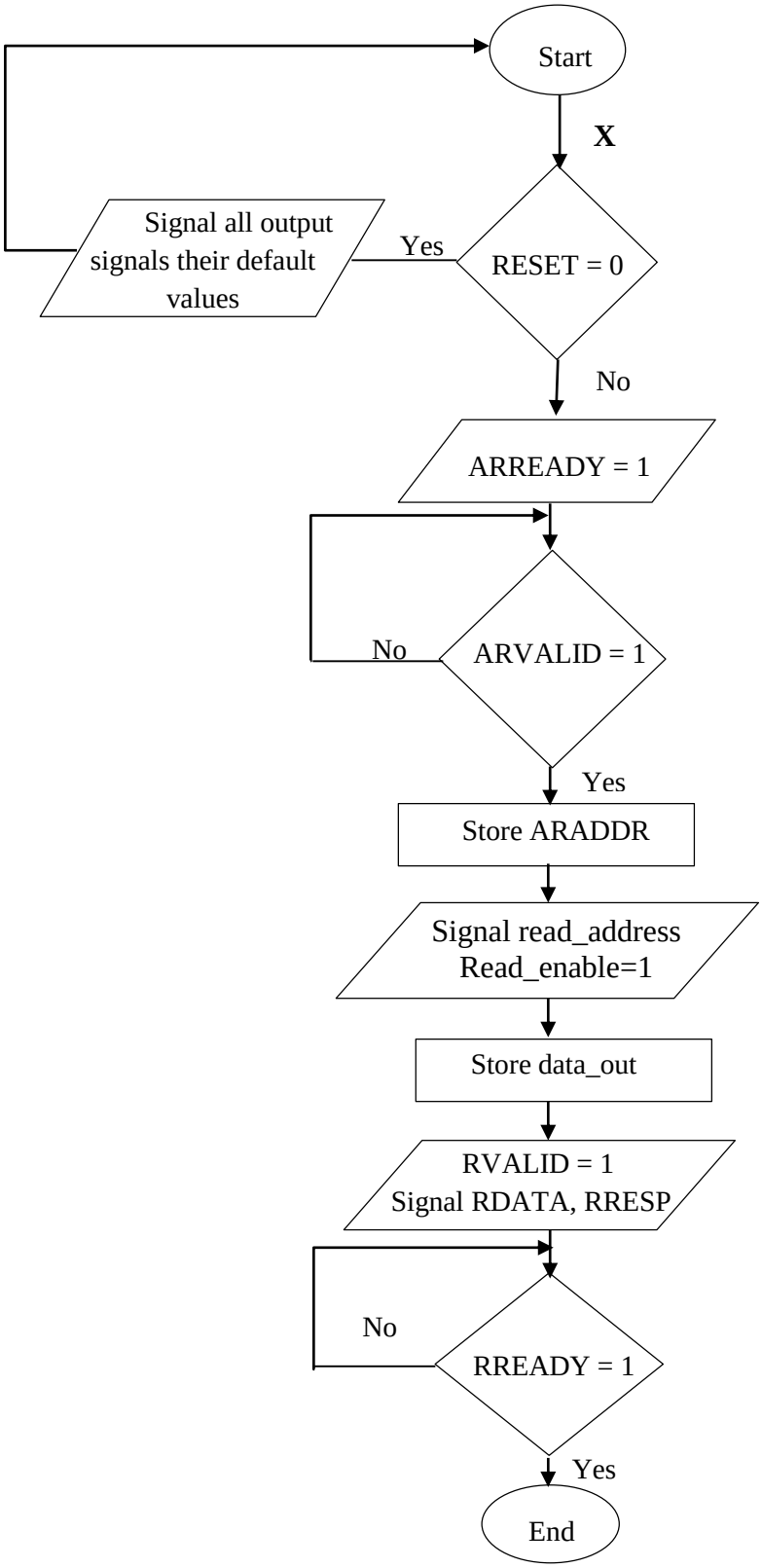


Figure 3-5AXI4-Lite Read Operation Flow Chart

Note: Resetting AXI4-lite at any point will result in going to start point (marked **X** at flowchart)

Implementation

AXI4-lite unit is divided into two main blocks as illustrated in the figure 3-6; **Write interface** is responsible of write transactions and **Read interface** is responsible of read transactions, both of blocks are implemented by a Mealy FSM.

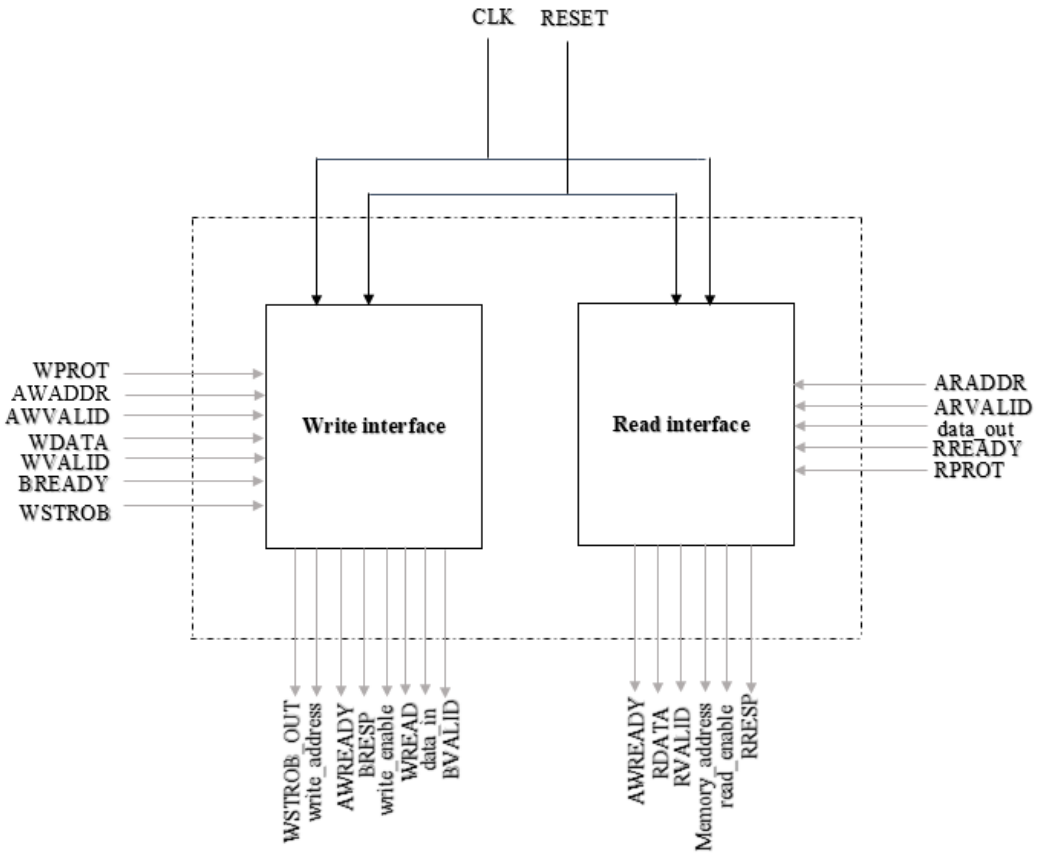


Figure 3-6 Block Diagram of AXI4-Lite

Write Interface

The finite state machine states of write interface are illustrated in the table 3-3 and in figure 3-7

State Number	State	Description
1	IDLE	Idle state when AXI4-lite is not active
2	ADDR WAIT	Wait for AWVALID signal to be high
3	DATA WAIT	Wait for WVALID signal to be high
4	DATA	Handshaking is done and data is written in the given address
5	RESP	Response of slave to indicate correct and incorrect write operation

Table 3-3Write Interface States Illustration

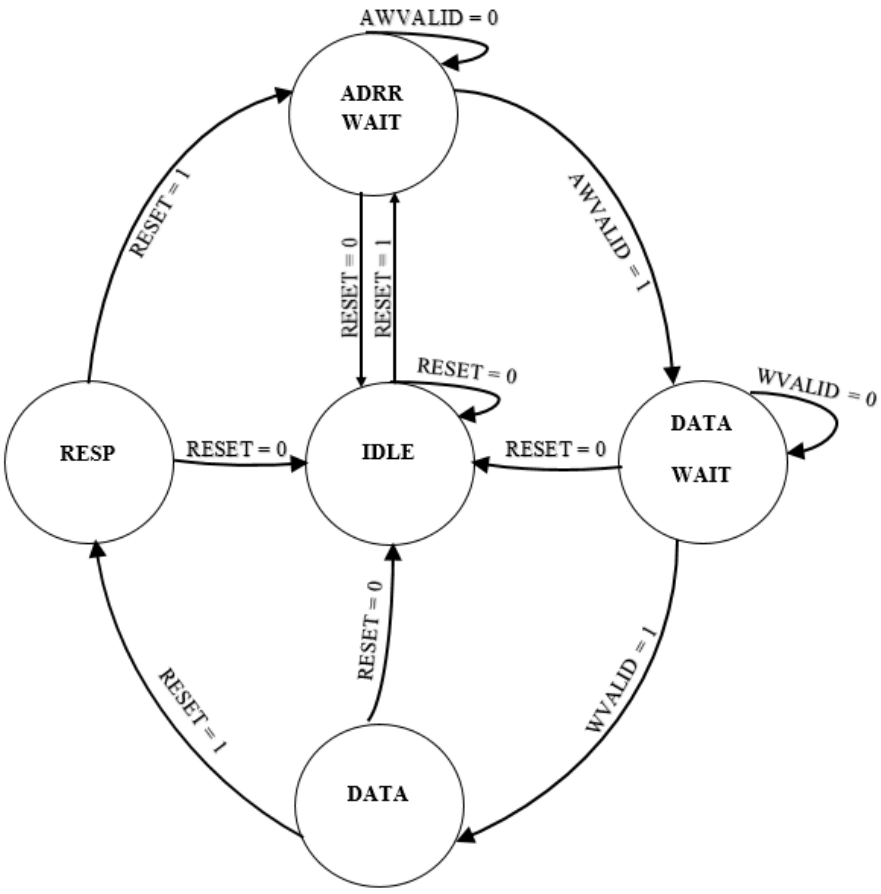


Figure 3-7 write operation FSM of AXI4-lite

Read Interface

The finite state machine states of read interface are illustrated in the table 3.4 and figure 3-8

State Number	State	Description
1	IDLE	Idle state when AXI4-lite is not active
2	ADDR WAIT	Wait for ARVALID signal to be high
3	DATA	Handshaking is done and data is read from the given address

Table 3-4Read Operation States Illustration

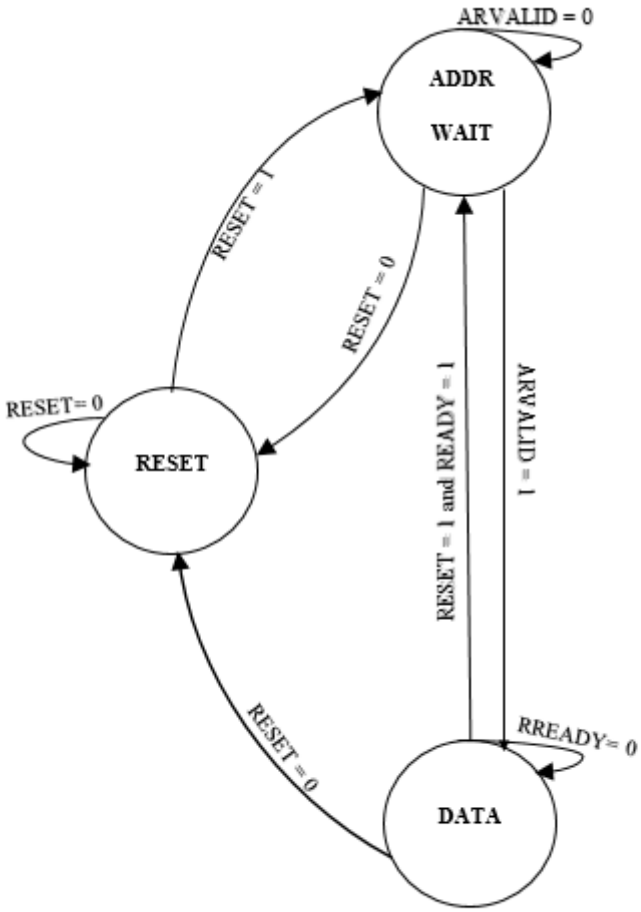


Figure 3-8 Read Operation FSM of AXI4-Lite

3.3.2. AXI4 Interface

Description

AXI4 interface is the secondary interface with the core, this interface has access only to two data registers in the Register Space; Transmit Data FIFO Data Write Port (TDFD) and Receive Data FIFO Data read Port (RDFD), it provides independent read and write channels where transactions take place on a two stage handshake which is very similar to AXI4-Lite interface.

The AXI4 protocol is very similar to AXI4-Lite interface except it supports more features; the AXI4 protocol supports three types of burst transactions specified by the signal **AxBURST** and a configurable burst length up to 256 transfers in incremental bursts and 16 transfers in fixed and wrapping bursts which is chosen by user and signaled by **AxLEN**.

The supported data bus width in the core is configurable and can be 32 bits or 64 bits.

The protocol supports full bus width or narrow transfers where the number of bytes used in data transfer are specified by the signal **AxSIZE**; however only bytes containing valid data is stored in write transactions. Valid data is indicated by the strobe signal where each bit corresponds to one byte of the data bus.

The AXI4 interface mainly deals with the data registers within the core where all read/write operations are passed to the Register Space via the interface and subsequently to data FIFOs if used has performed correct programming sequence, other information about data like packet destination and packet length must be written into core using AXI4-Lite interface.

Interfacing

The AXI4 block contains I/O signals listed in Table 3-5.

Port name	Port mode	Connection	Description
ACLK	Input	User	Global interface clock
ARESET	Input	User	Resets entire core
AWADDR	Input	User	Address of write transaction
AWLEN	Input	User	Number of transfers in a write transaction
AWSIZE	Input	User	Number of bytes used in each transfer in a write transaction
AWBURST	Input	User	Write transaction burst type
AWVALID	Input	User	Valid write address is available
BREADY	Input	User	Master ready to accept response
ARADDR	Input	User	Address of a read transaction
ARLEN	Input	User	Number of transfers in a read transaction
ARSIZE	Input	User	Number of bytes used in each transfer in a read transaction
ARBURST	Input	User	Read transaction burst type
ARVALID	Input	User	Valid read address is available
RREADY	Input	User	Master can accept read data
WDATA	Input	User	Data of a write transaction
WSTRB	Input	User	Indicates bytes lanes which contain valid data in a write transaction
WVALID	Input	User	Valid write data is available
WLAST	Output	User	Indicates last transfer in a write transaction
WREADY	Output	User	Slave can accept write data
RDATA	Output	User	Data of a read transaction
RRESP	Output	User	Status of a read transaction
RLAST	Output	User	Indicates last transfer in a read transaction
RVALID	Output	User	Valid data is available
ARREADY	Output	User	Slave can accept read address

BRESP	Output	User	Statues of a write transaction
BVALID	Output	User	Valid response is available in a write transaction
AWREADY	Output	User	Slave can accept write address
read_enable	Output	Register space	Read enable signal
write_enable	Output	Register space	Write enable signal
write_address	Output	Register space	Address of a write operation
Memory_address	Output	Register space	Address of a read operation
data_in	Output	Register space	Data of a write operation
data_out	Input	Register space	Data of a read operation
STROBE_OUT	Output	Transmit control	Indicates which byte lanes contains valid data in a write operation
WUSER	Input	User	Not used
AWQOS	Input	User	Not used
AWPROT	Input	User	Not used
AWLOCK	Input	User	Not used
AWUSER	Input	User	Not used
ARQOS	Input	User	Not used
ARPROT	Input	User	Not used
ARLOCK	Input	User	Not used
ARUSER	Input	User	Not used
BUSER	Output	User	Not used
BID	Output	User	Not used
WUSER	Input	User	Not used
RID	Output	User	Not used

Table 3-5AXI4 I/O signals

Functionality

The AXI4 interface is responsible for read/write operations from/into two register in the Register Space (TDFD and RDFD); this is done through five channels similar to AXI4-Lite interface.

This AXI4 protocol has 3 modes of operation (burst types) for write and read transactions

1. Fixed Burst
2. Incrementing Burst
3. Wrapping Burst

Although AXI4- interface can support the three mode of operation; only fixed bursts are used since user will have access to only one register in each path (transmit path and receive path). Write and read transaction are performed by the same concept illustrated in AXI4-Lite interface; however, since AXI4 supports more advanced features; the write/read transactions are slightly different.

The Write operation is done as follows

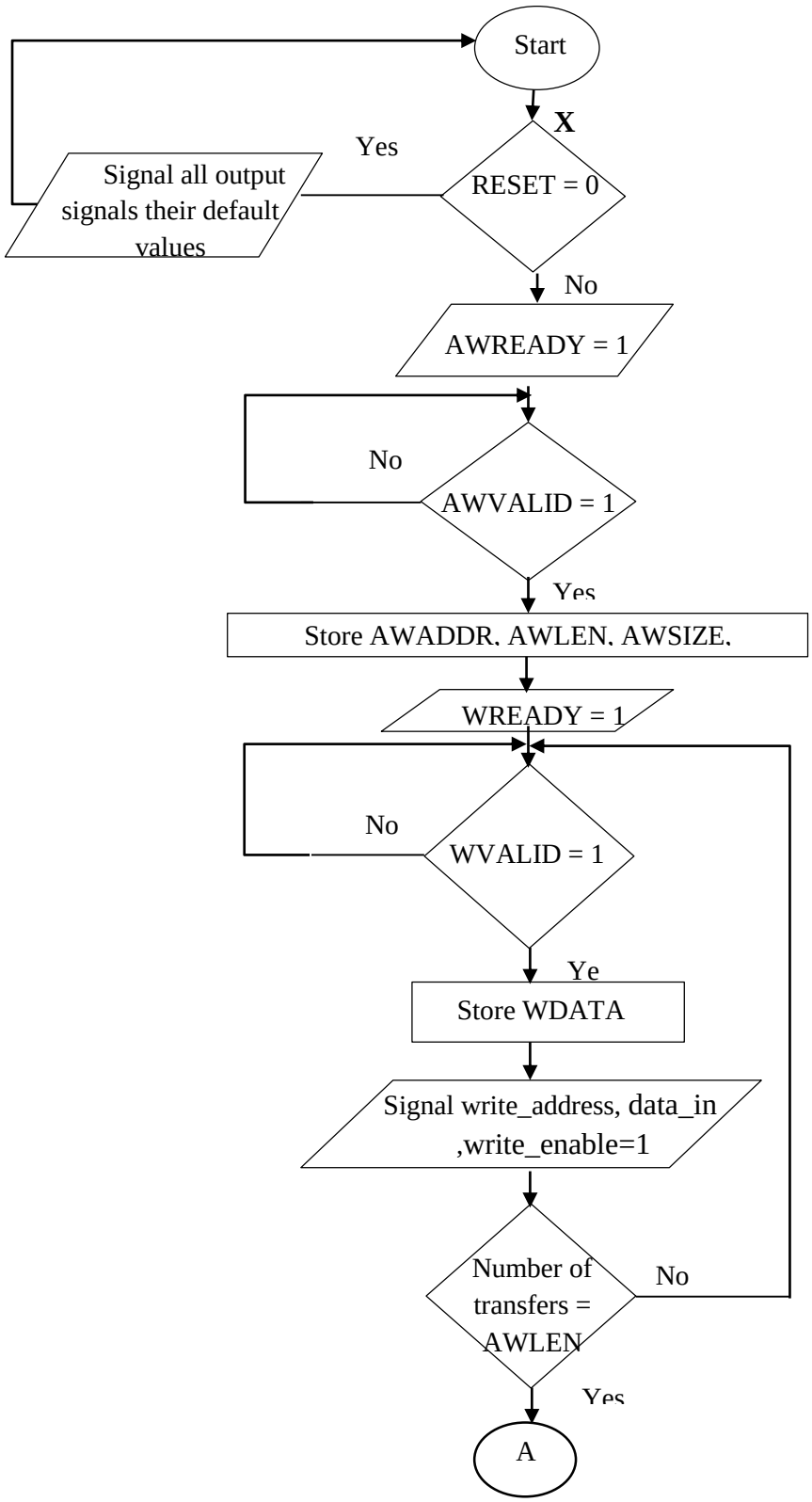
- The Address of the intended register to be written in and burst information are carried out on the Write Address Channel, when the handshake between the AWVALID and AWREADY takes place, the core has read the address of the register to be written in successfully and information about the transaction like transaction size, length and burst type are stored.
- The actual data is written on the Write Data Channel, when the handshake between the WVALID and WREADY takes place, this indicates that the core has read the data successfully, this step is performed as many times as transaction length.
- The core then replies with a Response signal to feedback the write transaction statues and similarly BREADY and BVALID perform the handshake.

The Read operation is done as follows

- The Address of the intended register to be read from and burst information are written on the Read Address Channel, and the handshake between the ARVALID and ARREADY takes place, which indicates that the core has read the control signals successfully.
- Data to be read is written from the core on the Read Data Channel, and the handshake between the RVALID and RREADY takes place, which indicates that the user has read the written data by the core successfully. This step is performed as many times as transaction length.

- The RRESP signal is signaled from core to feedback the read transaction statues.

Write transaction over AXI4 channel is illustrated by the flow chart in figure 3-9



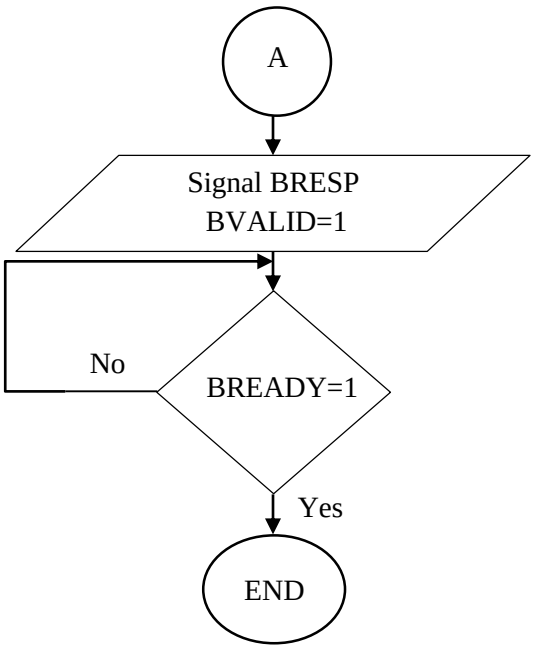


Figure 3-9AXI4 Write tranasaction FlowChart

Note: Resetting AXI4 at any point will result in going to start point (marked **X** at flow chart)

Read transaction over AXI4 channel is illustrated by the flow chart in figure 3.10

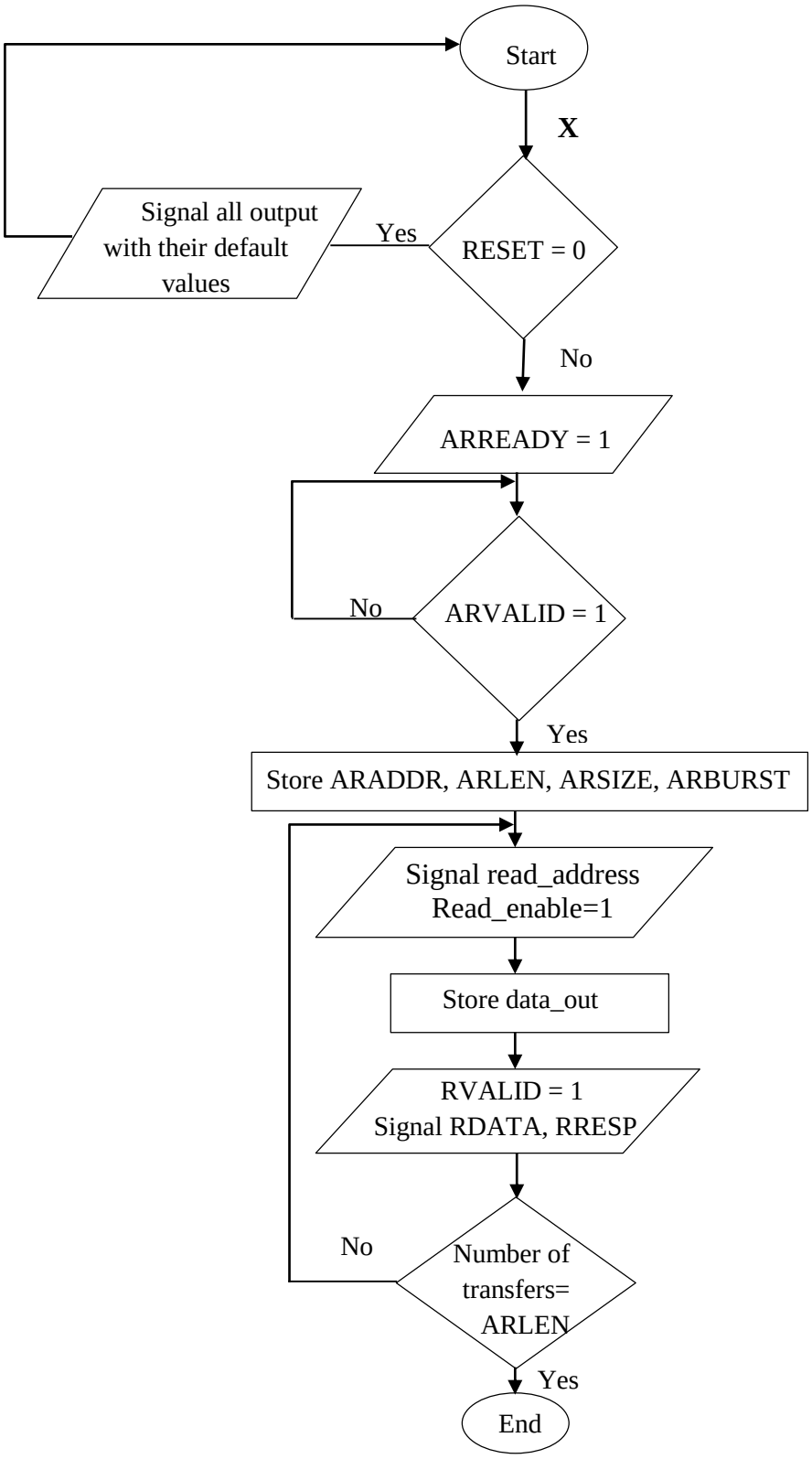


Figure 3-10 AXI4 Read transaction Flow Chart

Note: Resetting AXI4 at any point will result in going to start point (marked **X** at flow chart

Implementation

AXI4 unit is divided into two main blocks as illustrated in the figure 3.11; **Write interface** is responsible for write transactions and **Read interface** is responsible for read transactions, AXI4 Interface is implemented by two finite state machines.

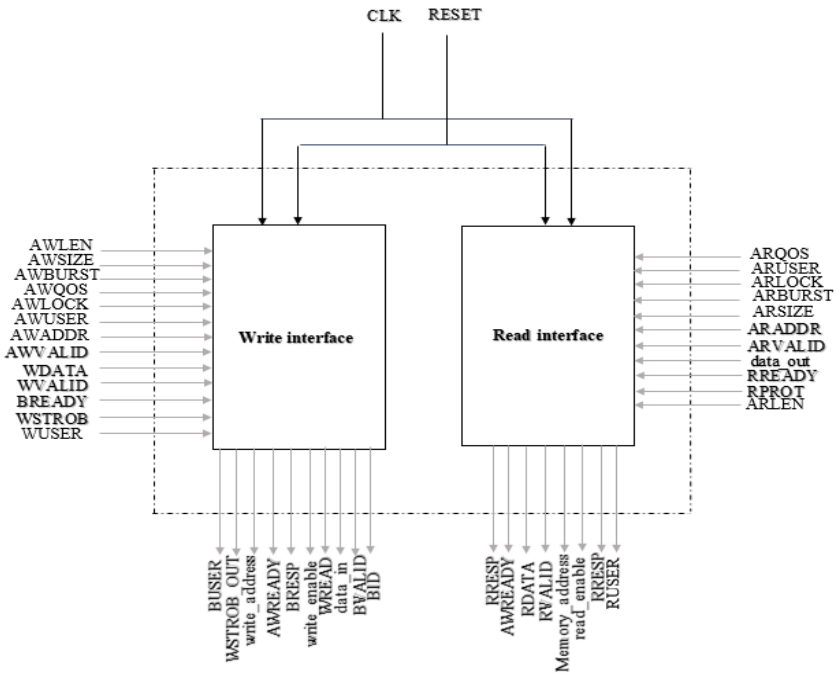


Figure 3-11AXI4 block diagram

Write Interface

The finite state machine of write operation states is illustrated in the table3-6 and figure3-12

State Number	State	Description
1	IDLE	Idle state when write interface is not active
2	Address wait	Valid address is stored
3	Wait for WVALID	Waiting state for data to be available
4	Data wait	Valid data is written
5	Response wait	Waiting state for master to accept write response
6	Slave error	Error state to feedback a slave error to master

Table 3-6Write operation states illustration

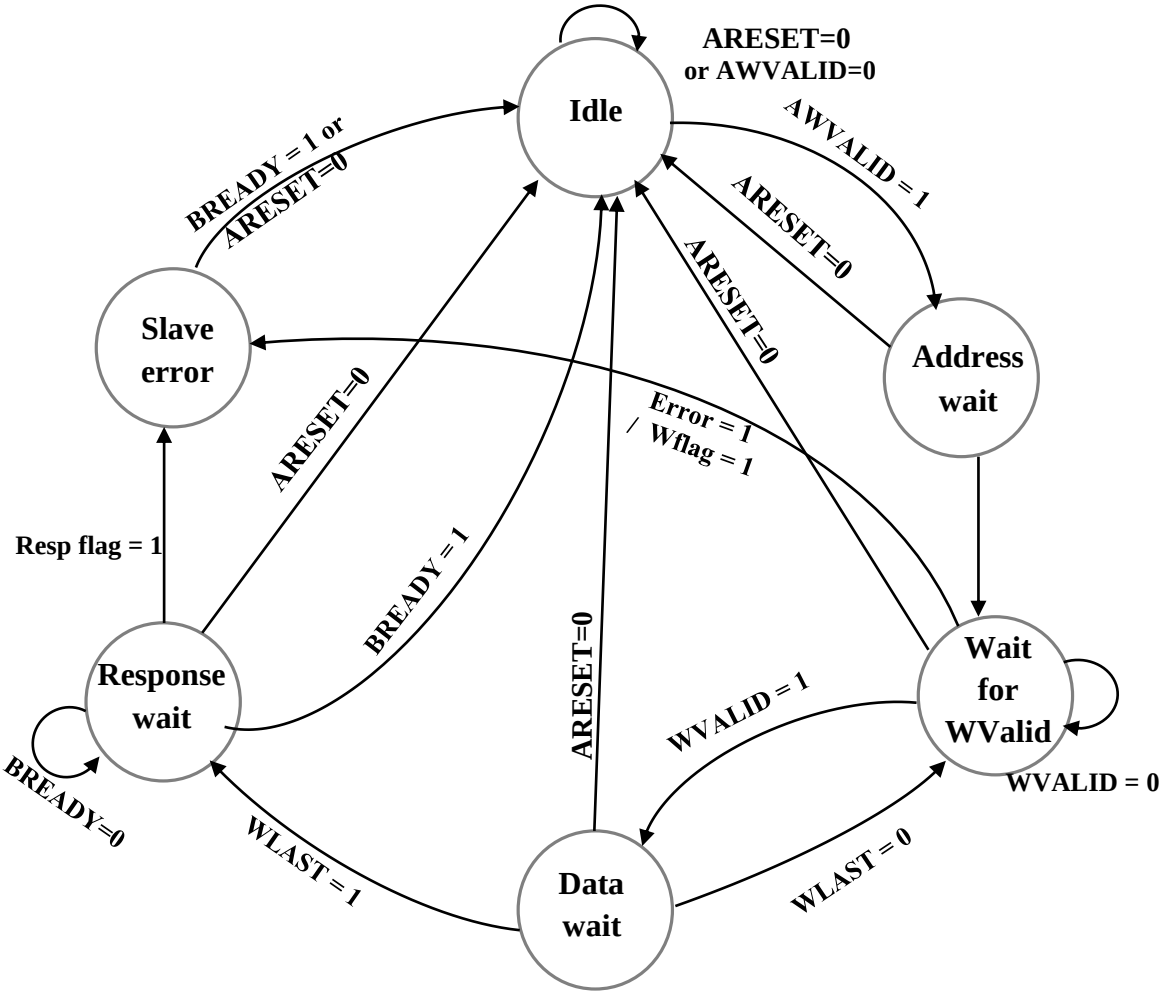


Figure 3-12Write Operation FSM of AXI4

The finite state machine transition conditions are illustrated in the table 3-7

Transition condition	Description	Condition
Reset	Reset AXI4 interface and entire core	ARREST=0
Valid address	Valid address and control signals are available	AWVALID=1
Valid data	Valid data is available	WVALID=1
Last	Last transfer in a write transaction	WLAST=1
Response ready	Master can accept write response	BREADY=1
Slave error	Slave wishes to return an error condition to the master	Error=1
Time out write error	Master has exceeded a parameterized maximum number of clock cycles without a data transfer	Wflag=1
Time out response error	Master has exceeded a parameterized maximum number of clock cycles without responding to valid data	Respflag =1

Table 3-7AXI4 Write Interface FSM transition conditions illustration

The finite state machine outputs corresponding to each state are illustrated in the table 3-8

Outputs	State number					
	1	**2**	**3**	**4**	**5**	**6**
AWREADY	0	1	0	0	1	1
WREADY	0	0	0	1	0	0
BRESP	00	00	00	00	00	10
BVALID	0	0	0	0	1	1

Table 3-8AXI4 write operation FSM outputs

Read Interface

The finite state machine of read operation states are illustrated in the table.3.9 and figure3.13

State Number	State	Description
1	IDLE	Idle state when AXI4-lite is not active
2	R_ADR_WAIT	Wait for AVALID signal to be high
3	R_DATA_WAIT	Wait for READY signal to be high
4	R_DATA_FIX	Move to this state if RBURST = “00” which indicates fixed mode
5	R_DATA_INCR	Move to this state if RBURST = “01” which indicates incrementing mode
6	R_DATA_WRAP	Move to this state if RBURST = “10” which indicates wrapping mode
7	R_DATA_END	Move to this state after ending read transaction
8	R_RES	Responsible for informing correct or incorrect read operation

Table 3-9AXI4 Read operation states

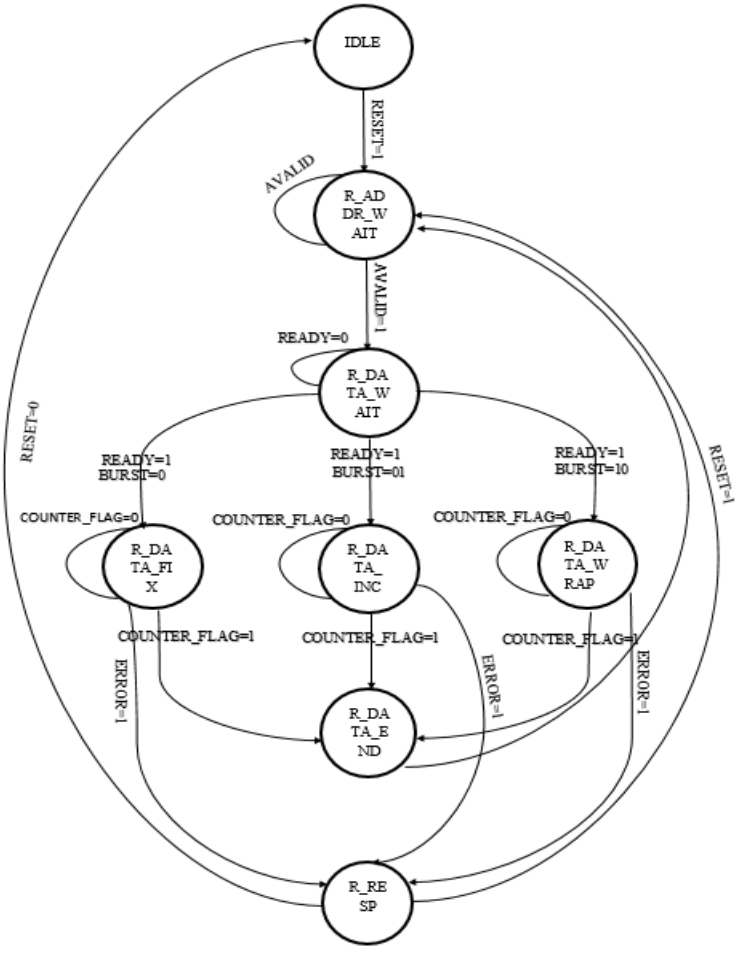


Figure 3-13 AXI4 Read Operation FSM

3.3.3. Register Space

Description

The Register Space contains all the registers that the user needs to interact with the core whether through write or read operations to operate the block, it contains thirteen registers where the AXI4-lite interface have access to the thirteen registers in case of using AXI4-lite as data interface, while in case of using AXI4 as data interface the AXI4-lite has access to eleven of the thirteen registers while the AXI4 has access to the data interface registers Transmit Data FIFO Write port (TDFD) and the Receive Data FIFO Read port (RDFD).

Interfacing

The I/O signal description is illustrated in Table 3-10

Port Name	Port mode	Connection	Description
user_read_data rg_read_address user_read_enable	Output Input Input	AXI4-lite interface	Outputs the stored value in system registers to the user
user_read_data_axi4 rg_read_address_axi4 user_read_enable_axi4	Output Input Input	AXI4 interface	Outputs the value stored in the RDFD register **ONLY** to the user
user_write_data rg_write_address user_write_enable	Input Input Input	AXI4-lite interface	Responsible of storing the data written by the user through the to the register of the corresponding address
user_write_data_axi4 rg_write_address_axi4 user_write_enable_axi4	Input Input Input	AXI4 interface	Responsible of storing the data written by the user to the TDFD register **ONLY**
Calc_TDFV Calc_RDFO	Input Input	Calculation unit	The values of the vacancy and occupancy to be stored in the corresponding registers
recieve_fifo_RDFD recieve_fifo_RLR recieve_fifo_RDR	Input Input Input	receive FIFO	The values of the data, length and destination to be stored in the corresponding registers
RDFD_en RLR_en RDR_en	Input Input Input Input	Receive Control and Transmit Control	The enable signals to save the incoming data to the corresponding register

TDFV_en RDFO_en	Input		
input_reset	Input	AXI4-lite interface	The reset to the whole core
Interrupt_service	Input	Interrupt Interface	The interrupt status of the system to be stored in the ISR
RLR_read_trial	Output	Interrupt Interface	Used to trigger receive underrun read interrupt
Read_op	Output	Interrupt Interface	Used to trigger receive underrun interrupt
Reset	Output	All other modules	The general reset to the whole core which can be triggered due to external reset or writing 0xA5 to SRR
Receive_reset	Output	Receive Control Receive FIFO	Reset to the receive blocks only which is triggered due to writing 0xA5 to the RDFR
Transmit_reset	Output	Transmit Control Transmit FIFO	Reset to the transmit blocks only which is triggered due to writing 0xA5 to the TDFR
ISR	Output	Interrupt Interface	To generate the interrupt bit
IER	Output	Interrupt Interface	To generate the interrupt bit
TLR	Output	Transmit Length FIFO	Outputs the length written by the user
TDR	Output	Transmit Destination FIFO	Outputs the destination written by the user
TDFD	Output	Transmit Data FIFO	Outputs the data written by the user

Table 3-10Register Space I/O signals

Functionality

Register Space has thirteen registers that the user needs to operate the core, in the following section a brief description of each register.

The description of each register is as follows:

Interrupt Service Register (ISR)

The System supports 13 different interrupts each is represented by one bit from bit 19 to bit 31, The ISR structure is shown in Figure 3.14. The supported interrupts and their corresponding bits in the ISR register are shown as follows in Table 3-11

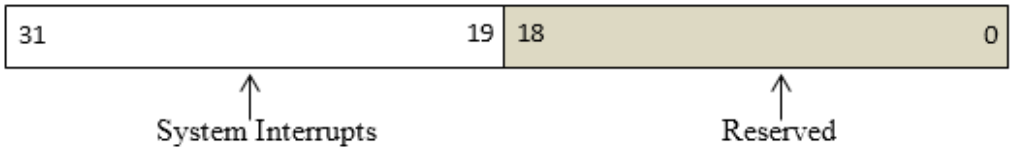


Figure 3-14 Interrupt Service Register (offset 0x00)

Core Access: Read/Clear on Write
Reset Value: 0x01D00000
Description: returns the interrupts generated by the system

Bits	Name	Core Access	Reset Value	Description
0-18	Reserve	Read	0x0	Reserved for future definition
19	RFPE	Read/ Clear on write of 1	0	**Receive FIFO Programmable Empty:** triggered when the difference between read and write pointer of receive FIFO equals the Empty threshold
20	RFPF	Read/ Clear on write of 1	1	**Receive FIFO Programmable Full:** triggered when the difference between read and write pointer of receive FIFO equals the Full threshold
21	TFPE	Read/ Clear on write of 1	0	**Transmit FIFO Programmable Empty:** triggered when the difference between read and write pointer of transmit FIFO equals the Empty threshold
22	TFPF	Read/ Clear on write of 1	1	**Transmit FIFO Programmable Full:** triggered when the difference between read and write pointer of transmit FIFO equals the Full threshold
23	RRC	Read/ Clear on write of 1	1	**Receive Reset Complete:** the reset of receive logic has completed
24	TRC	Read/ Clear on write of 1	1	**Transmit Reset Complete:** the reset of transmit logic has completed

25	TSE	Read/ Clear on write of 1	0	**Transmit Size Error:** triggered due to the mismatch of the written length by the user and the actual length of the written data in terms of number of words
26	RC	Read/ Clear on write of 1	0	**Receive Complete:** indicates the successful receive of one or more packets
27	TC	Read/ Clear on write of 1	0	**Transmit Complete:** indicates that at least one transmit has completed
28	TPOE	Read/ Clear on write of 1	0	**Transmit Packet Overrun Error:** triggered when an attempt to write to the transmit FIFO while it's full, reset of transmit logic is required to recover
29	RPUE	Read/ Clear on write of 1	0	**Receive Packet Underrun Error:** triggered when an attempt is made to read receive FIFO while it's empty, reset of receive logic is required to recover
30	RPORE	Read/ Clear on write of 1	0	**Receive Packet Overrun Read Error:** triggered when the read of the current packet from the FIFO exceeds the packet length, reset of receive logic is required to recover
31	RPURE	Read/ Clear on write of 1	0	**Receive Packet Underrun Read Error:** triggered when an attempt to read the RLR when it's empty, reset of receive logic is required to recover

Table 3-11Interrupt Service Register structure

Interrupt Enable Register (IER)

The Interrupt Enable Register acts as a mask for the ISR meaning only the interrupts which have their corresponding bit set in the IER will be able to trigger the interrupt bit

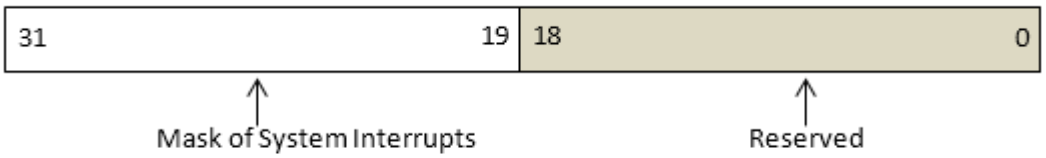


Figure 3-15 Interrupt Service Register (offset 0x00)

Core Access: Write
Reset Value: 0x00000000
Description: Masks the interrupts generated by the system from affecting the interrupt bit

Transmit FIFO Data register (TDFR)

TDFR is a write only register which when written with value 0xA5 generates a reset to all the transmit blocks in the core, the reset cannot interrupt an ongoing transmission of a packet and only takes place when the ongoing transmission is over.

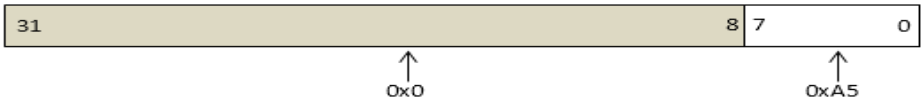


Figure 3-16Transmit Data FIFO Reset Register (offset 0x8)

Core Access: Write
Reset Value: 0x00000000
Description: generates a reset upon writing 0xA5, other values have no effect

Transmit Data FIFO Vacancy Register (TDFV)

The Transmit Data FIFO Vacancy Register shown in Figure 3.17 is a read-only register that gives the vacancy status of the Transmit Data FIFO. It stores the number of locations free for data storage in the Transmit Data FIFO. The value stored (N) in this register tells you that you can perform N writes to Transmit FIFO. The Register Value increments by one but decrements by two for every two write locations.

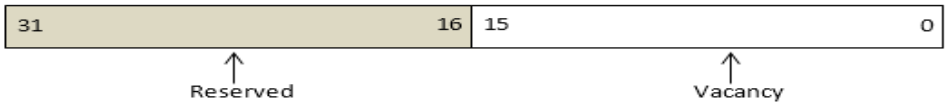


Figure 3-17 Transmit Data FIFO Vacancy Register (offset 0xC)

Core Access: Read
Reset Value: Transmit FIFO Depth - 4
Description: gives the vacancy status of the Transmit Data FIFO

Transmit Data FIFO Data Write Port (TDFD)

The Transmit Data FIFO Data Write Port shown in Figure 3.18 is an *N*-bit register for writing the data from the user into the Transmit Data FIFO. *N* is equal to AXI 4 Data width which is by default 32.

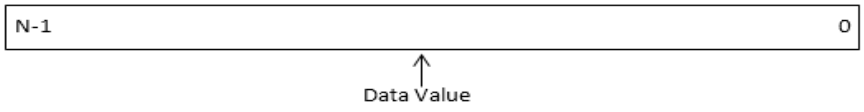


Figure 3-18Transmit Data FIFO Data Write Port (offset 0x10)

Core Access: Write
Reset Value: 0x00000000
Description: stores the data written by the user to be stored in the Transmit FIFO

Receive Data FIFO Reset Register (RDFR)

RDFR is a write only register which when written with value 0xA5 generates a reset to all the receive blocks in the core, the reset cannot interrupt an ongoing reception of a packet and only takes place when the ongoing reception is over.

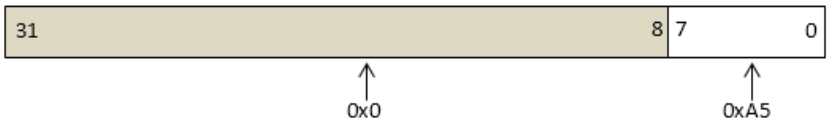


Figure 3-19 Receive Data FIFO Reset Register (offset 0x18)

Core Access: Write
Reset Value: 0x00000000
Description: generates a reset upon writing 0xA5, other values have no effect

Receive Data FIFO Occupancy Register (RDFO)

The Receive Data FIFO Occupancy Register shown in Figure 3.20 is a read-only register that gives the number of locations occupied in the receive FIFO, the value of RDFO is not updated except after full packet reception, returning 0 indicates that the FIFO is empty while other values indicate the number of locations used by the last successfully received packet.

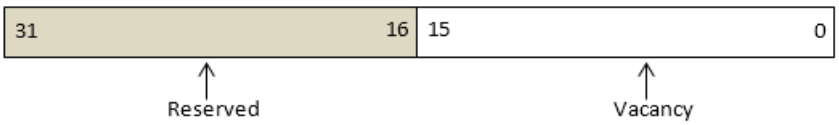


Figure 3-20 Receive Data FIFO Occupancy Register (offset 0x1C)

Core Access: Read
Reset Value: 0x00000000
Description: reflects number of locations occupied by the latest received packet, returns 0 if no packets received

Receive Data FIFO Data Register (RDFD)

The Receive Data FIFO Data Read Port shown in Figure 3.21 is an *N*-bit register for reading data from Receive Data FIFO. *N* equals AXI-4 Data width.

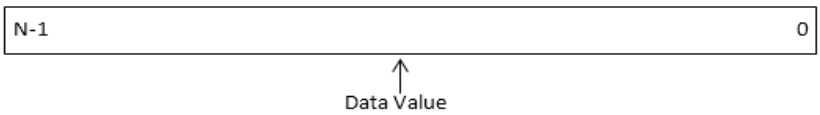


Figure 3-21 Receive Data FIFO Data Write Port (offset 0x20)

Core Access: Read
Reset Value: 0x00000000
Description: stores the data to be read by the user written in the Receive FIFO

Transmit Length Register (TLR)

The Transmit Length Register shown in Figure 3.22 stores the length of the packet to be transmitted in bytes, in case of store-and-forward mode the TLR is written with length of the to be transmitted before the actual transmission of the packet, while in cut-through mode the packet transmission begins as soon as the FIFO is not empty while the TLR is only written before the transmission of the last beat of the packet.

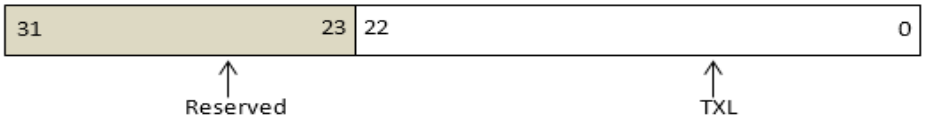


Figure 3-22 Transmit Length Register (offset 0x14)

Core Access: Read
Reset Value: 0x00000000
Description: stores the length of the packet written by the user in the Transmit FIFO

Receive Length Register (RLR)

The Receive Length Register shown in Figure 3.23 stores length of the received packet in bytes; value stored in RLR in store-and-forward mode represents packet length and is updated when the packet is completely received, while in cut-through mode the RLR stores the length of partial packets. Bit 31 is used to determine whether the value stored is the length of a partial packet or a complete packet, if bit 31 equals one then the stored value is that of a partial packet, while if bit 31 equals zero then stored value is that of a complete packet.

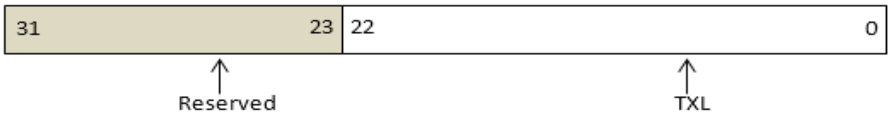


Figure 3-23 Receive Length Register (offset 0x14)

Core Access: Read
Reset Value: 0x00000000
Description: stores the length of the packet to be read by the user written in the Receive FIFO

AXI4-Stream Reset Register (SRR)

The AXI4-Stream Register shown in Figure 3.24 is a write-only address, which when written with a specific value, generates an immediate reset for the entire core as well as driving a reset on the external outputs, s2mm_prmry_reset_out_n, mm2s_prmry_reset_out_n, and mm2s_cntrl_reset_out_n, which can be used to reset the core on the other end of the AXI4-Stream.

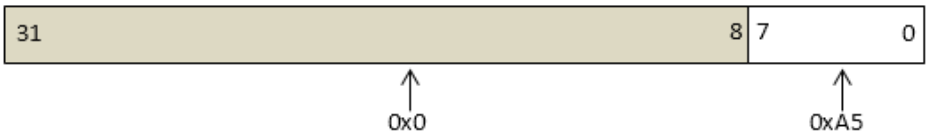


Figure 3-24 AXI4-Sream Reset Register (offset 0x28)

Core Access: Write
Reset Value: 0x00000000
Description: generates a reset upon writing 0xA5 to the whole core

Transmit Destination Register (TDR)

The Transmit Destination Register shown in Figure 3.25 stores the destination address corresponding to the packet to be transmitted.

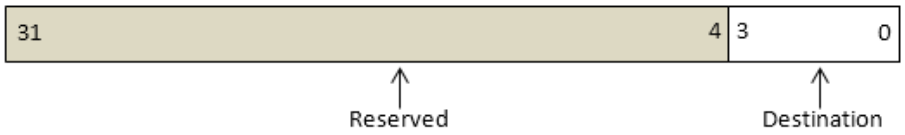


Figure 3-25 Transmit Destination Register (offset 0x2C)

Core Access: Write
Reset Value: 0x00000000
Description: contains the destination written by the user

Receive Destination Register (RDR)

The Receive Destination Register shown in Figure 3.26 retrieves the destination address corresponding to the valid packet received.

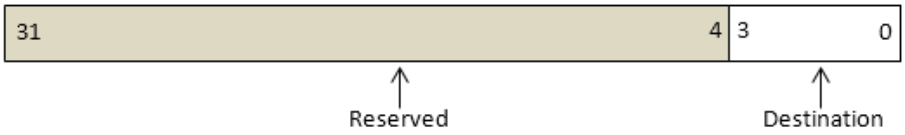


Figure 3-26 Receive Destination Register (offset 0x30)

Core Access: Read
Reset Value: 0x00000000
Description: contains the destination of the packet written from the AXI4-Stream Interface

Implementation

In the Register Space, Registers written by the user and Registers written by the system are mutually exclusive, same goes for Registers read by the user and Registers read by the system, this is the with the exception of the ISR.
The user accesses registers using Address, Data and Enable signals, while the system can access the registers directly by enable signals.
Not all Registers are fully accessible by the user; some Registers are read only while others are write only as follows

Read Only Registers to the user: TDFV, RDFO, RDFD, RLR, RDR

Write Only Registers to the user: TDFR**,** TDFD**,** TLR**,** RDFR**,** SRR**,** TDR

Read – Write Registers to the User: ISR**,** IER

- Any illegal attempt to write to read only registers has no effect on the registers value
- Any illegal attempt to read to write only registers returns output of zeros to the user on the data bus

The block design of the Register Space is illustrated as follows in Figure 3.27

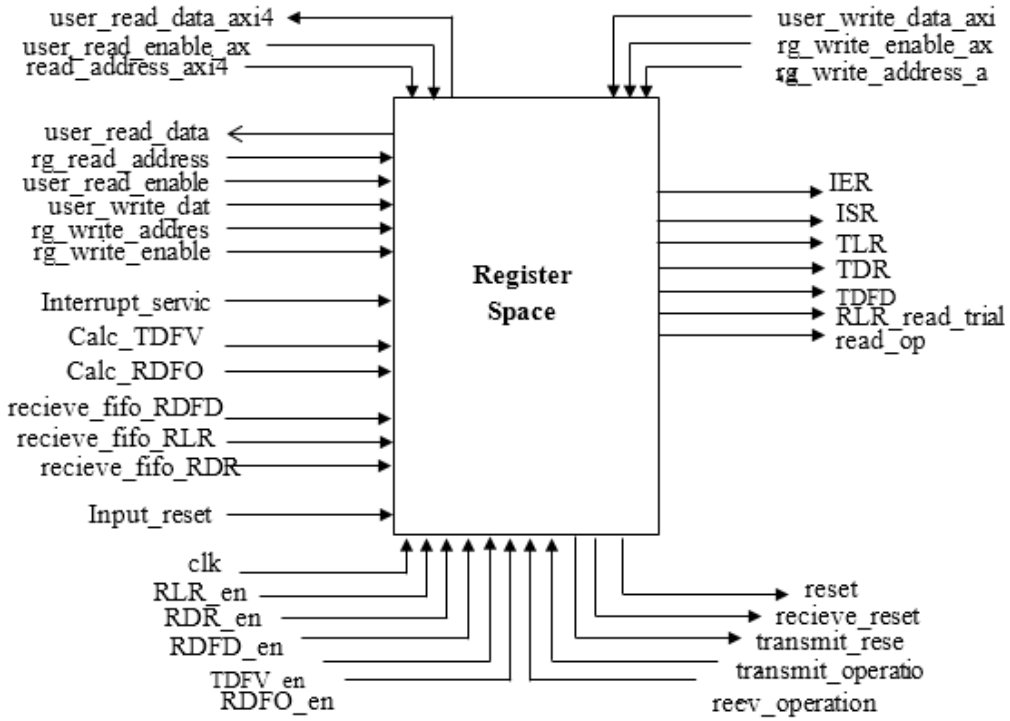


Figure 3-27 Register Space block diagram

3.3.4. Transmit Control

Description

Transmit Control is a block in the Control Unit and is the intermediate block between Transmit FIFO and Register Space. Block main responsibility is to allow data written by user to transfer from Register Space into Transmit FIFO and subsequently be transmitted over AXI-4 Stream interface; under the condition that user has performed a correct programming sequence to transmit a packet.

The block also calculates TSE bit (transmit size error) in the ISR register by calculating number of data bytes written in TDFD register and comparing this value to the value written by user in TLR register, if there's mismatch in number of words the block issues an error signal to Interrupt Interface.

Transmit Control block communicates with the following blocks; Transmit FIFO, Interrupt Interface, AXI4/AXI-Lite Interface and Register Space.

- Transmit FIFO block contains three FIFOs; destination FIFO to store data written in TDR register, Length FIFO to store packet length calculated internally in Transmit Control and data FIFO to store actual data written in TDFD register, wherefore Transmit Control block has three enable signals connected to the three FIFOs. The block also calculates number of data bytes in each data transfer and signals the value along with data enable signal; this value is used by data FIFO to update its pointers and latch in only valid data in TDFD register.
  (Since smallest packet can be 1 byte and TDFD register width equals data bus width which can take only two values; 32 bits and 64 bits).

- Interrupt Interface is signaled TSE signal from Transmit Control.

- AXI4/AXI-Lite Interface has two signals used by Transmit Control to monitor Register Space; write enable signal and write address signal, those two signals are used to detect write operations in the Register Space performed by user and therefore detect a correct programming sequence.

- Transmit Control is connected to some registers and bits in Register Space; TLR register to calculate TSE bit, TPOE bit in the ISR register which can push the block into lock state until user resets either the transmit circuitry or the entire core. Register Space is responsible for resetting Transmit Control when a user writes specific values in either SRR register or TDFR register.

Interfacing

The Transmit Control block contains I/O signals listed in Table 3.12

Port name	Port mode	Connection	Description
Clk	Input	Interface	Global interface clock
reset_all	Input	Register Space	Resets entire core
reset_tx	Input	Register Space	Resets transmit circuitry
rg_write_enable	Input	Interface	Enable write into register space
rg_write_address	Input	Interface	Write address to register space
Error	Input	Register space	Transmit packet overrun error
rs_rdata_TLR	Input	Register space	Data in the TLR register
Strobe	Input	AXI4 Interface	AXI4 Strobe signal
TSE_Error	Output	Interrupt controller	Transmit size error
FIFO_wdata_enable	Output	Transmit FIFO	Enable for transmit data FIFO
FIFO_wlength_enable	Output	Transmit FIFO	Enable for transmit data length FIFO
FIFO_wdestination_enable	Output	Transmit FIFO	Enable for destination FIFO
FIFO_len_data_Tx_fifo	Output	Transmit FIFO	Number of data bytes in one transfer
Active	Output	Register space	Indicates when transmit block logic is active
packet_length_seq	Output	Transmit FIFO	Calculated packet length

Table 3-12 Transmit Control I/O signals

Functionality

The block is responsible of performing the following two functions

1. Allow data flow from Register Space into Transmit FIFO when user performs correct transmit programming sequence
2. Handle transmit size error (TSE) bit in the ISR register

The first function is performed by monitoring write operations done in three registers in the Register Space; TDR register for packet destination, TDFD register for data and TLR register for packet length.

To perform a correct, transmit programming sequence user must access the three registers mentioned earlier in a specific order; first TDR, then TDFD which can be accessed many times in order to write the packet by many transfers, then TLR register. This functionality is performed by a Moore finite state machine.

When user performs correct programming sequence Transmit Control will signal write enable signals to Transmit FIFO to permit the FIFOs to latch the data in from Register Space.

The second function is performed by using strobe signal from AXI4/AXI4-Lite data interface to calculate number of valid data bytes in each transfer, and hence calculate packet length. If there's mismatch in number of words between calculated value and packet length written by user in TLR register taking into consideration partial words; the block will issue a transmit size error signal to Interrupt Interface.

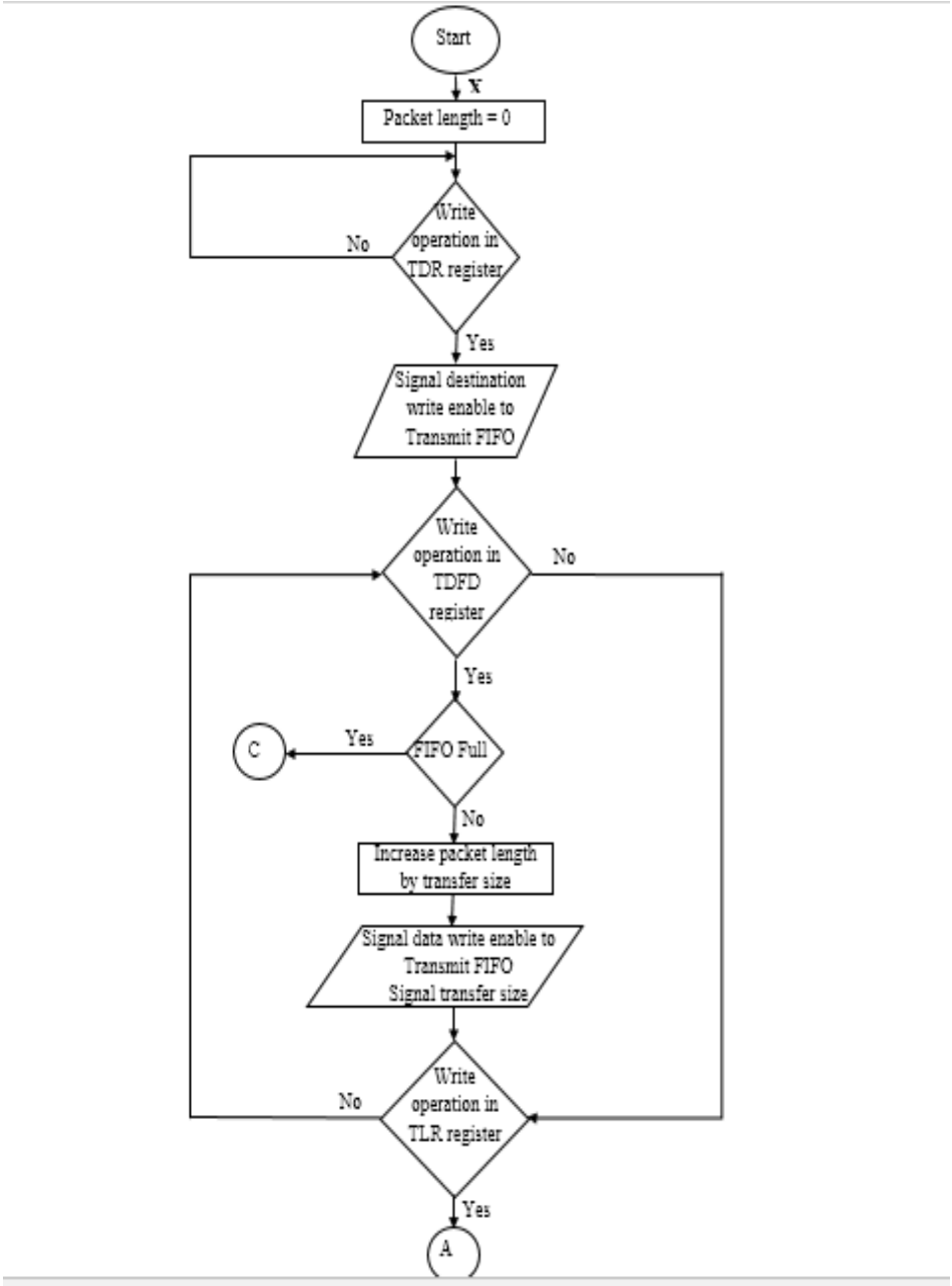
Since the standard doesn't specify a certain action in case of a transmit size error case, packet length stored in the Transmit FIFO is the calculated value not the value entered by the user, this solution will prevent improper operation of interconnect for the upcoming packets if the user doesn't take any action in case of a transmit size error.

- Transmit programming sequence and hence Transmit Control operation is the same in both modes of operation (store and forward mode and cut through mode) and both data interfaces (AXI4 and AXI4-Lite).

- Writing correct programming sequence is the responsibility of the user. Wrong programming sequence will push interconnect into stuck state and user will need to reset transmit circuitry or entire core to retrieve proper operation of interconnect.

- Transmit Control doesn't require from the user to perform programming sequence in minimum clock cycles.

Transmit Control block functionality is illustrated by the flow chart in figure 3.28



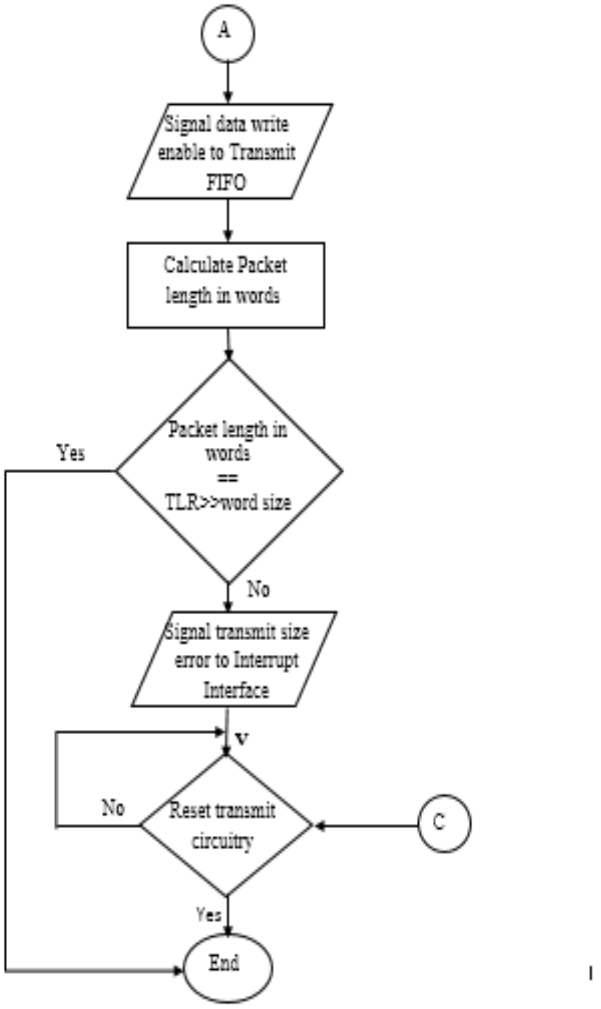


Figure 3-28 Transmit Control functionality flow chart

- Resetting transmit circuitry or entire core at any point will result in going to start point (marked **X** at flow chart)
- Wrong programming sequence will result in going to stuck point (marked **Y** at flow chart)

Implementation

Transmit Control unit is divided into two main blocks as illustrated in the figure 3.29; **length_calc** block is responsible for calculating packet length as number of bytes, and **Transmit_control** block is responsible for performing a Moore finite state machine to perform the two functions mentioned earlier with the help of **length_calc** unit to calculate packet length and subsequently calculate TSE bit.

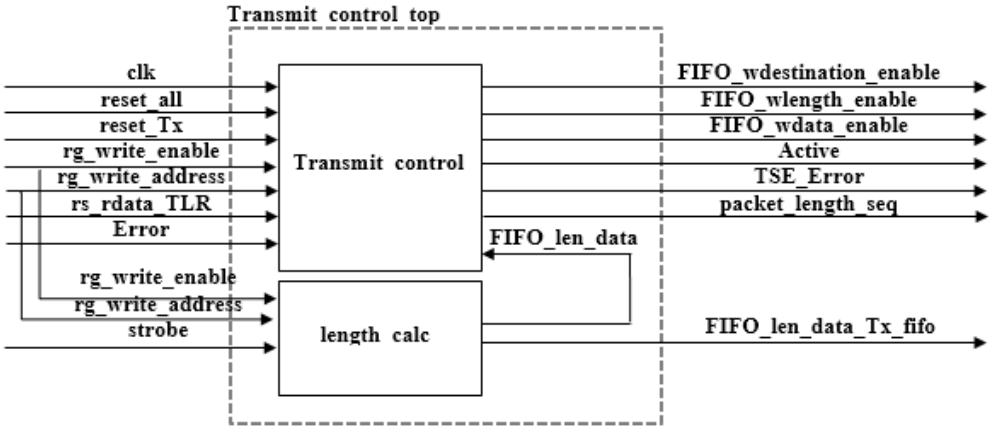


Figure 3-29 Transmit Control block diagram

length_calc Block

The block calculates number of valid data bytes in each transfer using strobe signal from data interface chosen by user (AXI4/AXI4-Lite). The block is parameterized by data bus width and calculates number of bytes by adding strobe signal bits.

The block transmits two versions of the calculated value; **FIFO_len_data** is connected to Transmit_control and represents actual number of data bytes and used to calculate packet length, **FIFO_len_data_Tx_fifo** represents number of data bytes -1 and is connected to Transmit FIFO and used to indicate number of valid data bytes in TDFD register and also write pointer of data FIFO is updated according to this value.

Transmit_Control Block

The Moore finite state machine states are illustrated in the table 3.13 and figure 3.30

State Number	State	Description
1	Idle	Idle state when transmit control is not active
2	State0	Signal destination write enable to Transmit FIFO
3	State1	Wait for write operation in TDFD register
4	State2	Signal data write enable to Transmit FIFO
5	State3	Wait for write operation in either TDFD or TLR registers
6	State4	Signal length write enable to Transmit FIFO
7	tranmsit_error	Signal TSE error to Interrupt Interface
8	Stuck	Stuck state when TPOE is high

Table 3-13Transmit Control FSM states illustration

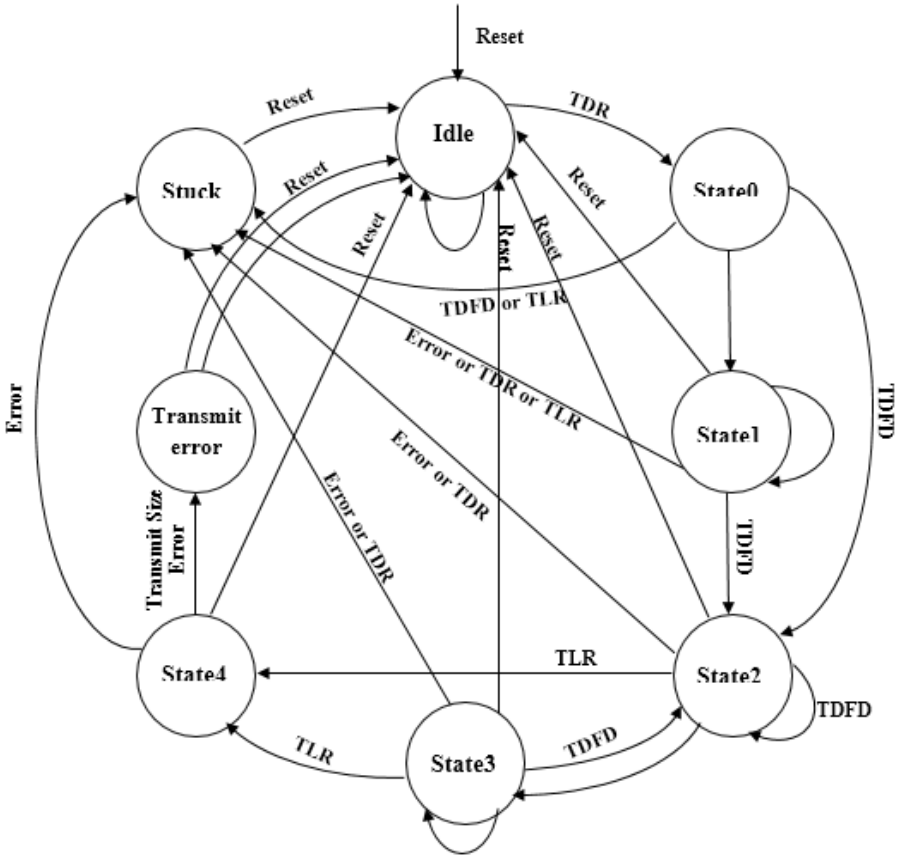


Figure 3-30 Transmit Control FSM

The finite state machine transition conditions are illustrated in the table 3.14

Transition condition	**Description**	**Condition**
Reset	reset for transmit circuitry	reset_all or reset_tx == 1
TDR	write operation in TDR register	rg_write_enable && rg_write_address == 0x2C
TDFD	write operation in TDFD register	**for AXI4-Lite interface** rg_write_enable && rg_write_address == 0x10 **for AXI4 interface** rg_write_enable && rg_write_address == 0x00
TLR	write operation in TLR register	rg_write_enable && rg_write_address == 0x14
Transmit size error	mismatch between TLR value and packet size calculates internally in words	(packet_length << Log_2(word size)) + partial_word_TC != (rs_rdata_TLR<< Log_2(word size)) + partial_word_TLR - Note: word size can take only two values; 32 bits and 64 bits
Error	Transmit packet overrun error	TPOE == 1

Table 3-14Transmit Control FSM conditions

The finite state machine outputs corresponding to each state are illustrated in the table 3.15

Outputs	**State number**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Active	0	1	1	1	1	1	1	0
TSE_Error	0	0	0	0	0	0	1	0
FIFO_wdata_enable	0	0	0	1	0	0	0	0
FIFO_wlength_enable	0	0	0	0	0	1	0	0
FIFO_wdestination_enable	0	1	0	0	0	0	0	0

Table 3-15Transmit Control FSM outputs

- Note: Whenever a user writes data into TDFD register during correct programming sequence **packet_length_seq** will be updated to represent a current snapshot of the packet length, but this value is latched in to Transmit FIFO only when **FIFO_wlength_enable** is high to represent the actual packet length.

3.3.5. Transmit FIFO Unit

Description

Transmit FIFO Unit is the intermediate unit between Transmit Control and AXI4-Stream Transmit Data Channel. Unit main responsibility is to allow data written by user to be stored when Transmit Control raises its enable signals and to allow data stored to be read and subsequently be transmitted over AXI4-Stream Transmit Data Channel.

Transmit FIFO Unit is divided into two Main blocks:

1. Transmit FIFO
   This block contains three circular FIFOs as follows,

   - Transmit Data FIFO (to store actual data written in TDFD register).
   - Transmit Length FIFO (to store packet length calculated internally in Transmit Control block).
   - Transmit Destination FIFO (to store data written in TDR register).

2. Stream Mapper

   - Responsible for transmitting packets over AXI4-Stream channel in the two modes of operation (Store-and-Forward mode and Cut-Through mode).

Interfacing

Transmit FIFO Unit contains I/O signals listed in Table 3.16

Port name	Port mode	Connection	Description
Tclk	Input	Interface	Global interface clock
Reset_All	Input	Register Space	Resets entire core
RESET	Input	Register Space	Resets transmit logic
TREADY	Input	AXI4-Stream Transmit Data Channel	Indicates that the slave can accept a transfer in the current cycle
WE_DATA	Input	Transmit Control	Enable reading data from register space (TDFD) & Write in Transmit Data
WE_L	Input	Transmit Control	Enable reading Length from Transmit Control & Write in Transmit Data
WE_D	Input	Transmit Control	Enable reading Destination from register space (TDR) & Write in Transmit Data
W_byte	Input	Transmit Control	Number of valid data bytes read from register space(TDFD) and stored into Transmit FIFO
data_in_F	Input	Register space (TDFD)	Data in the TDFD register Width [63:0]
data_in_L	Input	Register space (TLR)	Packet length Width [31:0]
data_in_D	Input	Register space (TDR)	Packet destination Width [3:0]
fifo_empty	Output	Interrupt controller	indicates if Data FIFO is empty =1 or not =0
fifo_full	Output	Interrupt controller	indicates if Data FIFO is full =1 or not =0
fifo_occpancy	Output	calc_unit	Represents number of empty bytes in Transmit Data FIFO
TVALD	Output	AXI4-Stream Transmit Data Channel	Indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted
TDES	Output	AXI4-Stream Transmit Data Channel	Destination AXI Stream Identifier and Provides routing information for the data stream

TDATA	Output	AXI4-Stream Transmit Data Channel	The primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes 32-bytes.
TKEEP	Output	AXI4-Stream Transmit Data Channel	The byte qualifier that indicates whether the content of the associated byte of TDATA is valid. For a 32-bit DATA, bit 0 corresponds to the least significant byte on DATA, and bit 3 corresponds to the most significant byte
TUSER	Output	AXI4-Stream Transmit Data Channel	TUSER: User-defined sideband information that can be transmitted with the data stream (not used)
TID	Output	AXI4-Stream Transmit Data Channel	The data stream identifier that indicates different streams of data **(not used)**
TSRB	Output	AXI4-Stream Transmit Data Channel	The byte qualifier that indicates whether the content of the associated byte of TDATA is processed as a data byte or a position byte **(not used)**
TLAST	Output	AXI4-Stream Transmit Data Channel	TLAST: Indicates the boundary of a packet
TRESET	Output	AXI4-Stream Transmit Data Channel	Reset for the AXI4-Stream Transmit data interface
RESET_COMPLETE	Output	Interrupt Interface / Transmit FIFO	Indicate that the reset operation has done

Table 3-16Transmit FIFO Unit I/O signals

Functionality

Transmit FIFO Block

Transmit FIFO block contains three circular FIFOs, the block is responsible for performing the following functions,

1. Allow data written in the Register Space (TDFD, TLR and TDR register) to be stored into Transmit FIFO when Transmit Control raises write enable signals

2. Allow data stored into Transmit FIFO to be read, and transmitted over AXI4 Stream Interface.

The first function is performed in three steps

1. By monitoring **WE_DATA** and **W_byte** raised by Transmit Control, when **WE_DATA** is high and there is enough space in Transmit data FIFO to perform a write operation; data is stored and **Next_Position_W** is updated to determine the location of the next write operation using **W_byte** which is used to determine number of data bytes stored into data FIFO and accordingly update write pointer.
2. Data written in TDR register is stored into destination FIFO when **WE_D** is high
3. Packet length calculated internally in Transmit Control is stored into length FIFO when **WE_L** is high

The second function is performed as follows,

By monitoring **R_en** and **number_byte_R** rised by AXI-4 Stream Mapper,

when **R_en** is high and there is enough data in Transmit FIFO to be read. Then **Next_Position_R** is used to determine the location of the next read operation.

Data stored into destination and length FIFOs is read similarly, but it’s a lot simpler since data stored in those FIFOs have fixed size.

Transmit FIFO block consist of:

1. **Memory** block
2. **Next_Position_W** block: Main Functionality is to calculate the location of the next write operation.
3. **Next_Position_R** block: Same as **Next_Position_W** block but calculates location of the next read operation by Stream Mapper.
4. **Check Full_empty** block which has the following functions,
   - Detect if Transmit data FIFO is Full or not.
   - Detect if Transmit data FIFO is Empty or not.

Next position Calculation for read/write operation from/into FIFO is illustrated by the flow chart in figure 3.31

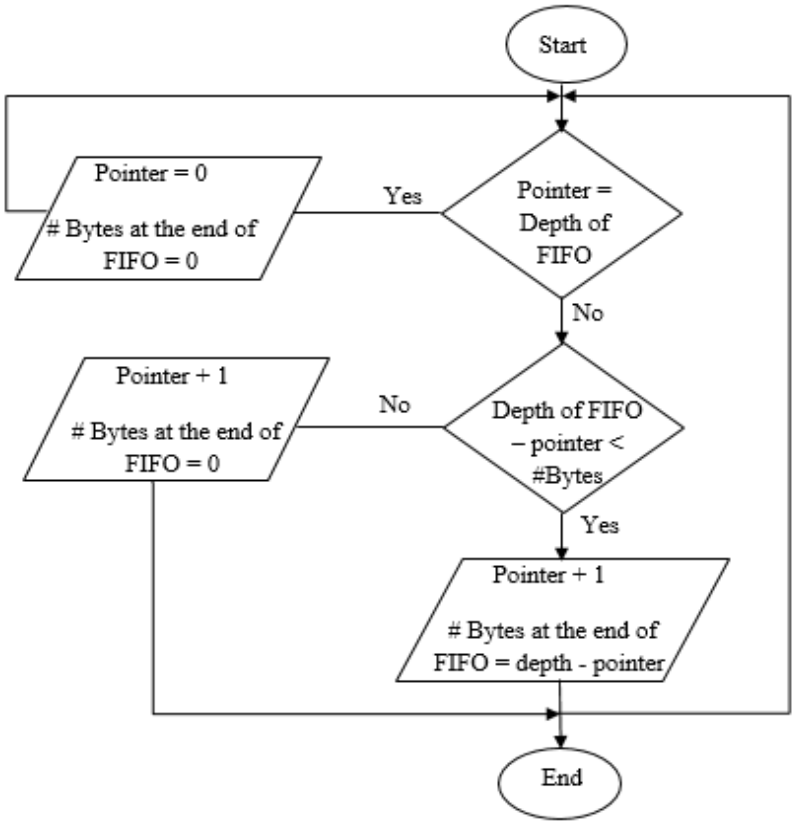


Figure 3-31 Detecting Next position read/write flow chart

Notes:

- # Bytes → number of bytes to be read/written from/into transmit FIFO.
- # Bytes at the end of FIFO → number of Bytes that can be read/written from/into the FIFO if it was non-circular FIFO, i.e. before read/write pointers are set to zero and write/read operations start from first location again.
- In the Transmit Length FIFO and Transmit Destination FIFO write and read operations have fixed size. Length FIFO read/write operation size is 32 bits and as for Destination FIFO, read/write operations are of size 4-bits. So #Bytes not used.
- In the Transmit Length FIFO and Transmit Destination FIFO when read enable signal is high data is read in the same clock cycle, while in transmit Data FIFO data is read in the next cycle.

Check Full and Empty block functionality is illustrated by the flow chart in figure 3.32

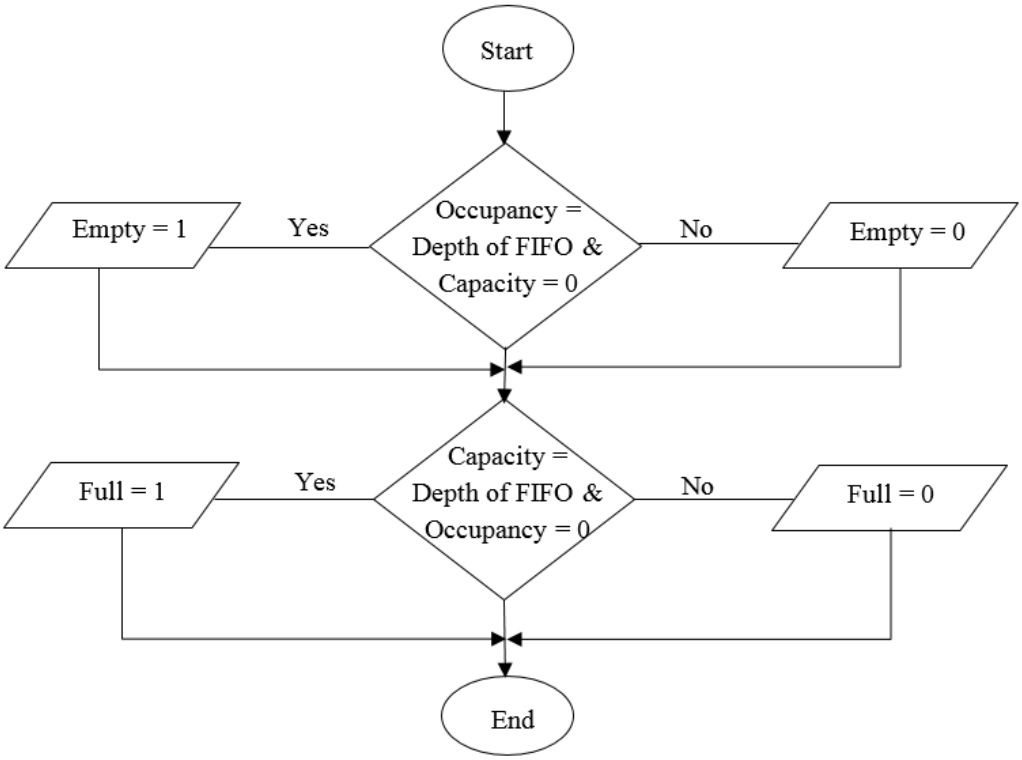


Figure 3-32 Check Full and Empty of FIFO flow chart

Note:

- Each cycle if read/write operations occurred; Capacity and Occupancy are updated and **check Full and Empty** operation is performed again.

Write /Read operations in Transmit FIFO is illustrated by the flow chart in figure 3.33

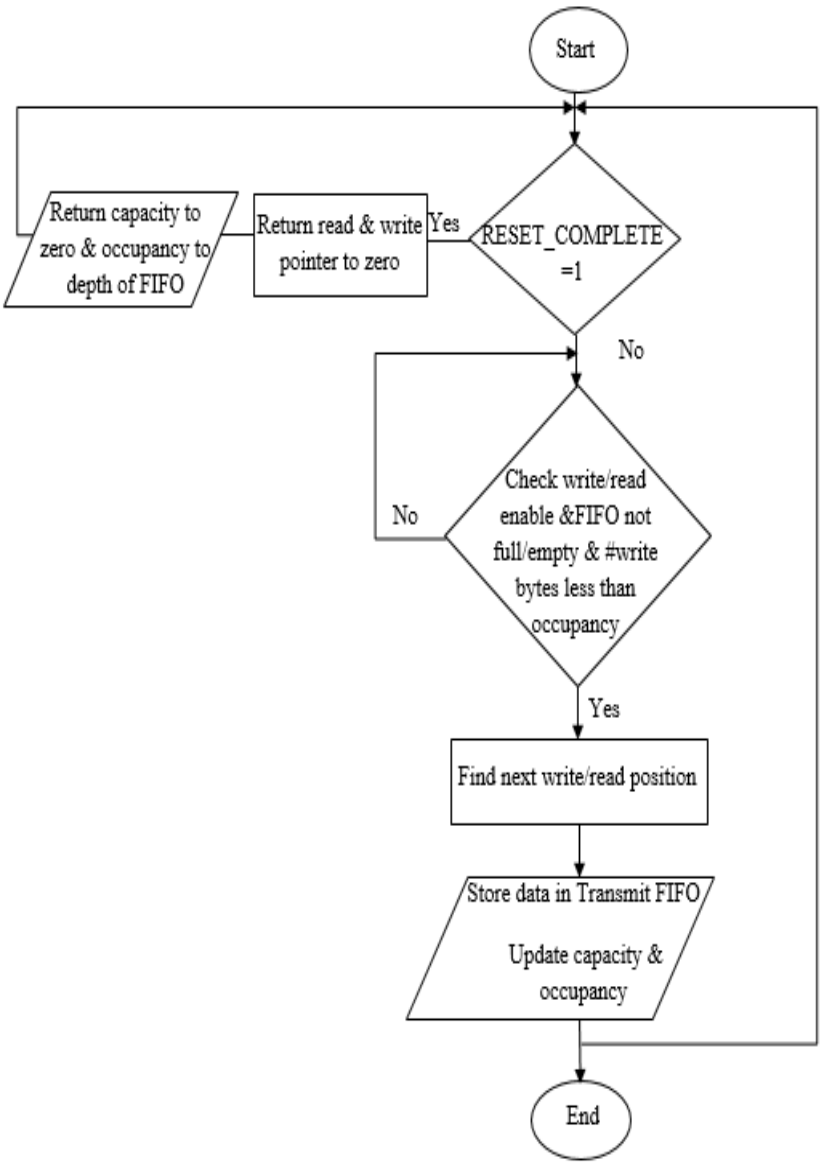


Figure 3-33Write /Read operation in Transmit FIFO flow chart

Notes

- If **RESET_COMPLETE** is high resetting operation is allowed to be performed, this can happen if the user resets entire core or if the user have issued a reset signal but a packet was being transmitted over stream interface at this time so when the packet is fully transmitted this signal is high to indicate that resetting operation is allowed to be performed

- Checking number of bytes required to be written is less than vacancy of transmit data FIFO is to make sure that there is enough space in data FIFO to store one transfer, while in case of writing data into transmit length/destination FIFO there's no need to check for this condition, as one transfer is fixed and equals one location; so it's sufficient to check that transmit length/destination FIFO is not full.

- In case of performing a read operation there's no need to check that number of bytes required to be read is less than occupancy of data FIFO; this is because packet length is first read from length FIFO and then data read from data FIFO corresponds to the packet length

Stream Mapper

The block is responsible for performing the following functions:

1- Read Destination, Data and Length from Transmit FIFO block
2- Enable data to be transmit over AXI4-Stream Transmit Data Channel in two modes:

- **Store-and-forward mode**, packet transmission begins on the AXI4-Stream interface in the following circumstances:

  - When the complete packet is written to the FIFO,
  - Length of packet is written to TX Length Register (TLR).

  In this mode, the size of the FIFO must be large enough to hold the complete packet. In this mode The main function of the block is to monitor the **empty_L** signal until it's low (this condition means a complete packet is stored in transmit data FIFO and the corresponding packet length is stored into length FIFO), when this happen Stream Mapper issues **TVALD** and wait until **TREADY** become high than Data can be Transmitted over AXI4-Stream Transmit Data Channel.

- **Cut-through mode**, packet transmission begins on the AXI4-Stream interface when there is enough data in the transmit data FIFO.

  In this mode, the FIFO does not need to hold the complete packet before transmission starts over stream interface. However, the block must ensure that the last beat of data is transmitted when the packet length is written by user into TLR register.

  The main function of the block in this mode is performed by monitoring **empty_Data** signal until it's low (this condition means there is data stored into Transmit Data FIFO), but transmission operation will not start until the data in the Transmit Data FIFO is enough (this condition is satisfied if capacity of Data FIFO is larger than width of **TDATA** or capacity of Data FIFO is smaller than or equal width of **TDATA** but **empty_L** is low and also **data_out_L** equals capacity), Then Stream Mapper raise **TVALD** and waits until **TREADY** become high to start data transmission over AXI4-Stream Transmit Data Channel. The last transfer can't be transmitted over AXI4-Stream Transmit Data Channel until the length of packet is written in Transmit Length FIFO.

If **Reset** signal is low and a packet is being transmitted over AXI4-Stream Transmit Data Channel, the resetting operation can’t be performed until the packet is fully transmitted, after this condition is satisfied **RESET_COMPLETE** signal

is raised high to indicate that the resetting operation is performed, while if **Reset_All** signal becomes low the reset operation is performed immediately even if a packet was being transmitted over stream interface.

Packet transmission over AXI4-Stream Transmit Data Channel in Store and Forward mode illustrated by the flow chart in figure 3.34

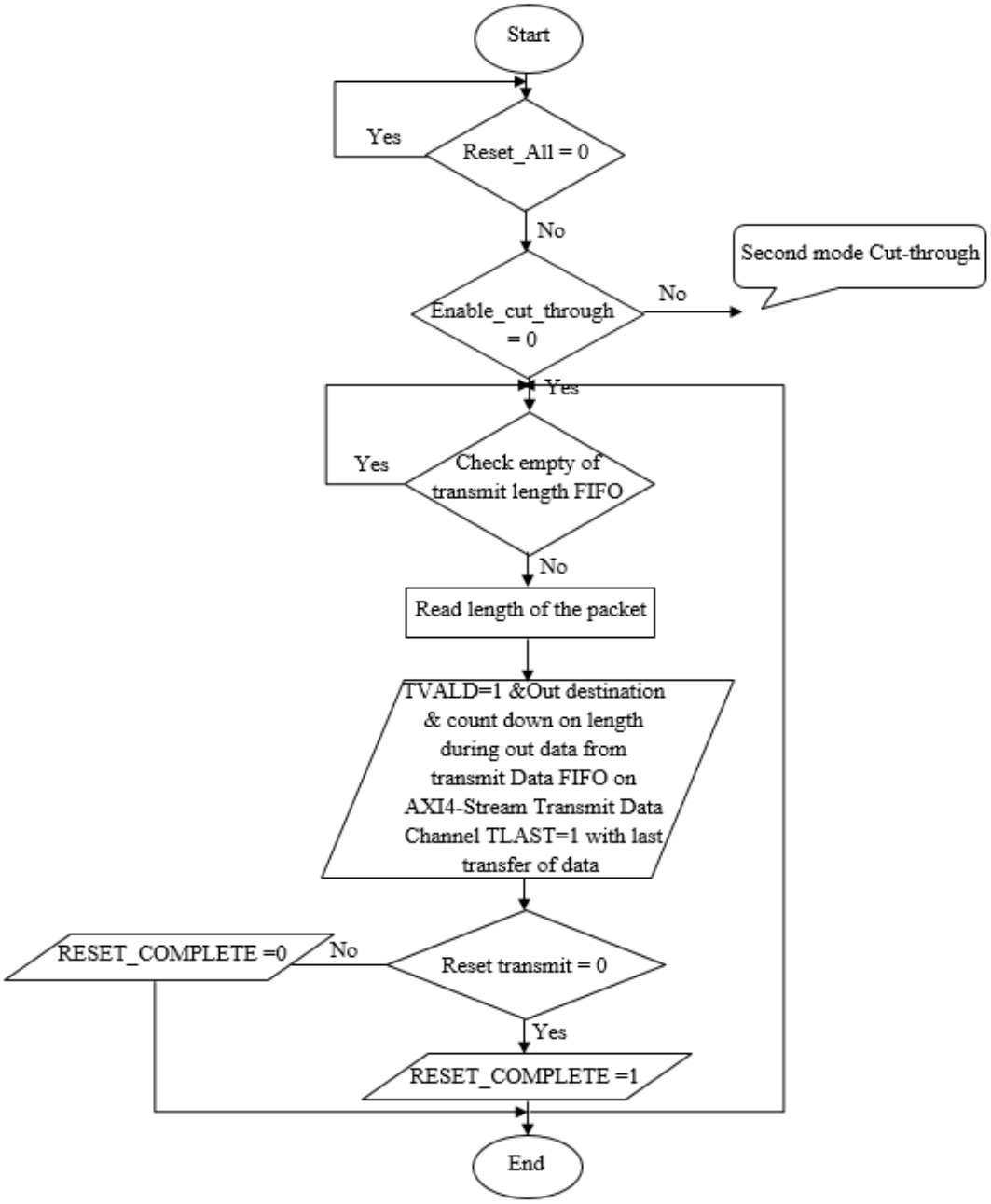


Figure 3-34 Store and forward mode flow chart

Acknowledged transmission over AXI4-Stream Transmit Data Channel in Cut-through mode illustrated by the flow chart in figure 3.35

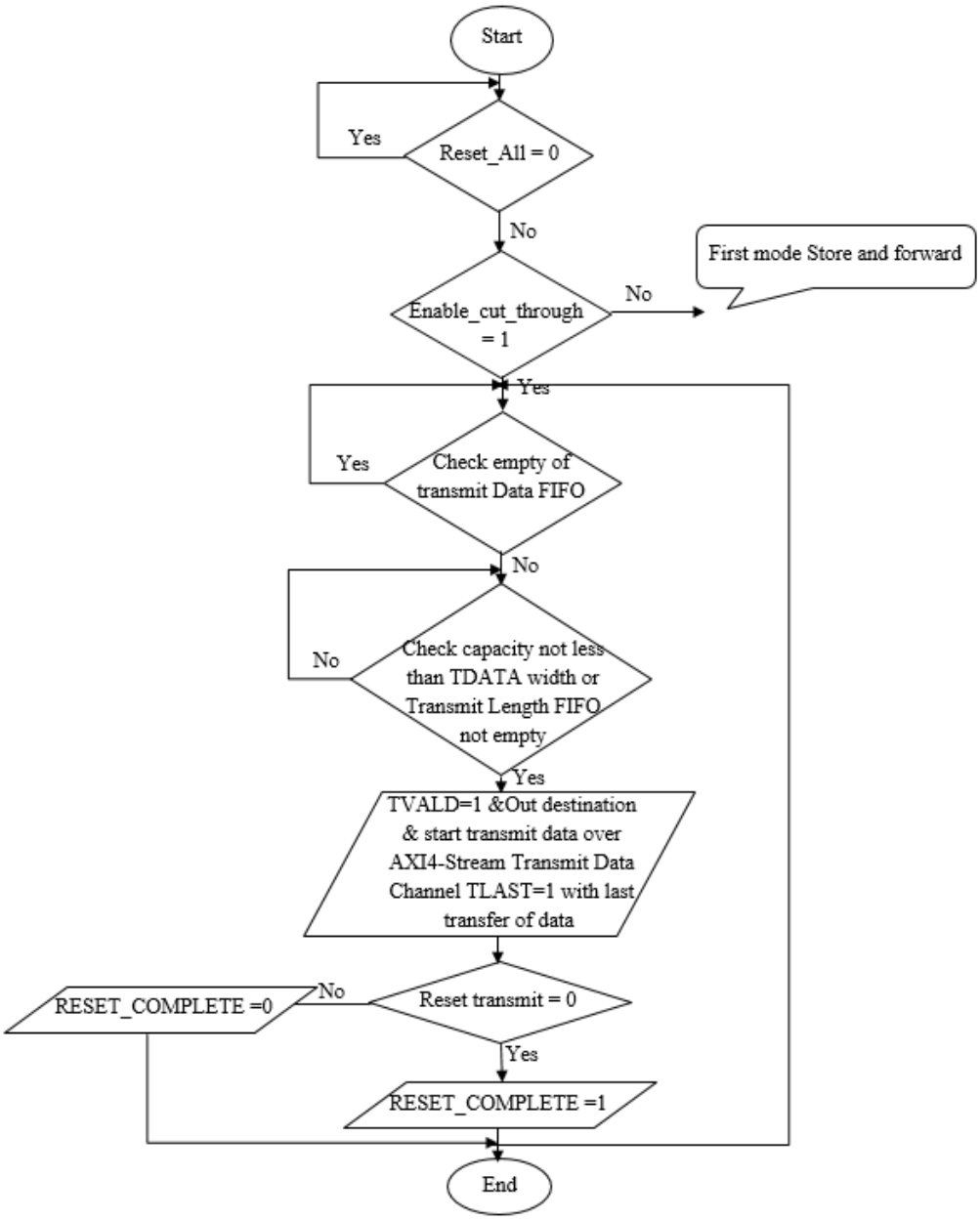


Figure 3-35 Cut-through mode flow chart

Implementation

Transmit FIFO Unit is divided into four main blocks as illustrated in the figure 3.36

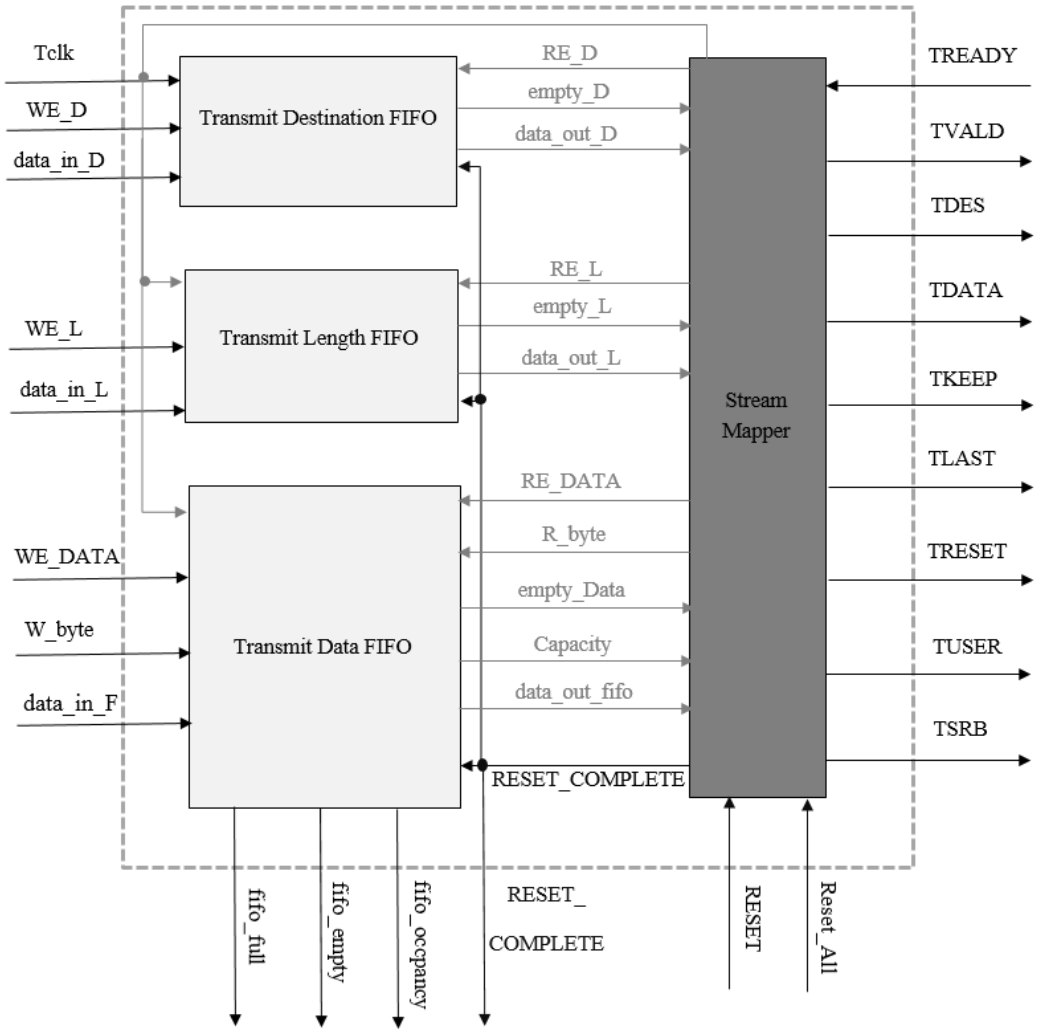


Figure 3-36 FIFO and Stream Mapper block

Transmit FIFO Block

Transmit Data FIFO block is divided into four main blocks as illustrated in the figure 3.37

1- Memory
2- Next_Position_W
3- Next_Position_R
4- Check full_empty

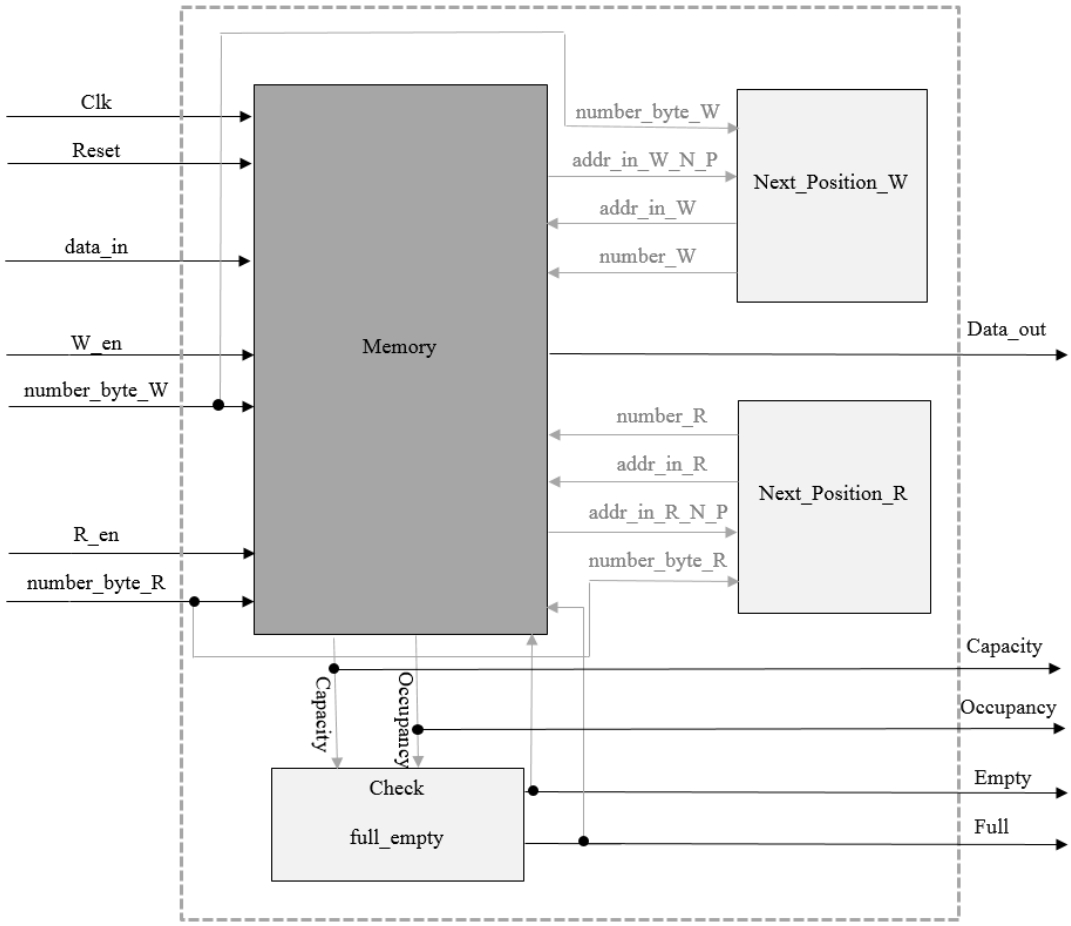


Figure 3-37 Transmit Data FIFO

Stream Mapper Block

Stream Mapper is implemented by a mealy finite state machine, the finite state machine supports two modes of operation (Store-and-Forward mode and Cut-Through mode). Figure 3.38 shows part of the finite state machine to illustrate block performance in store and forward mode.

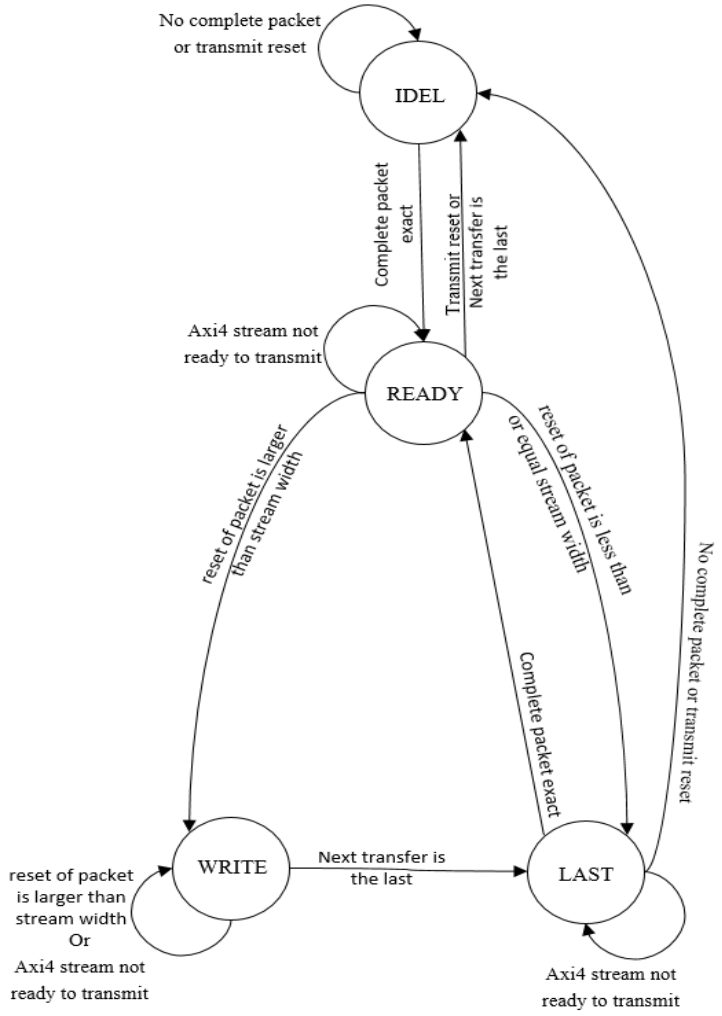


Figure 3-38 Store and forward mode finite state machine

Figure 3.39 shows part of the finite state machine to illustrate block performance in cut through mode.

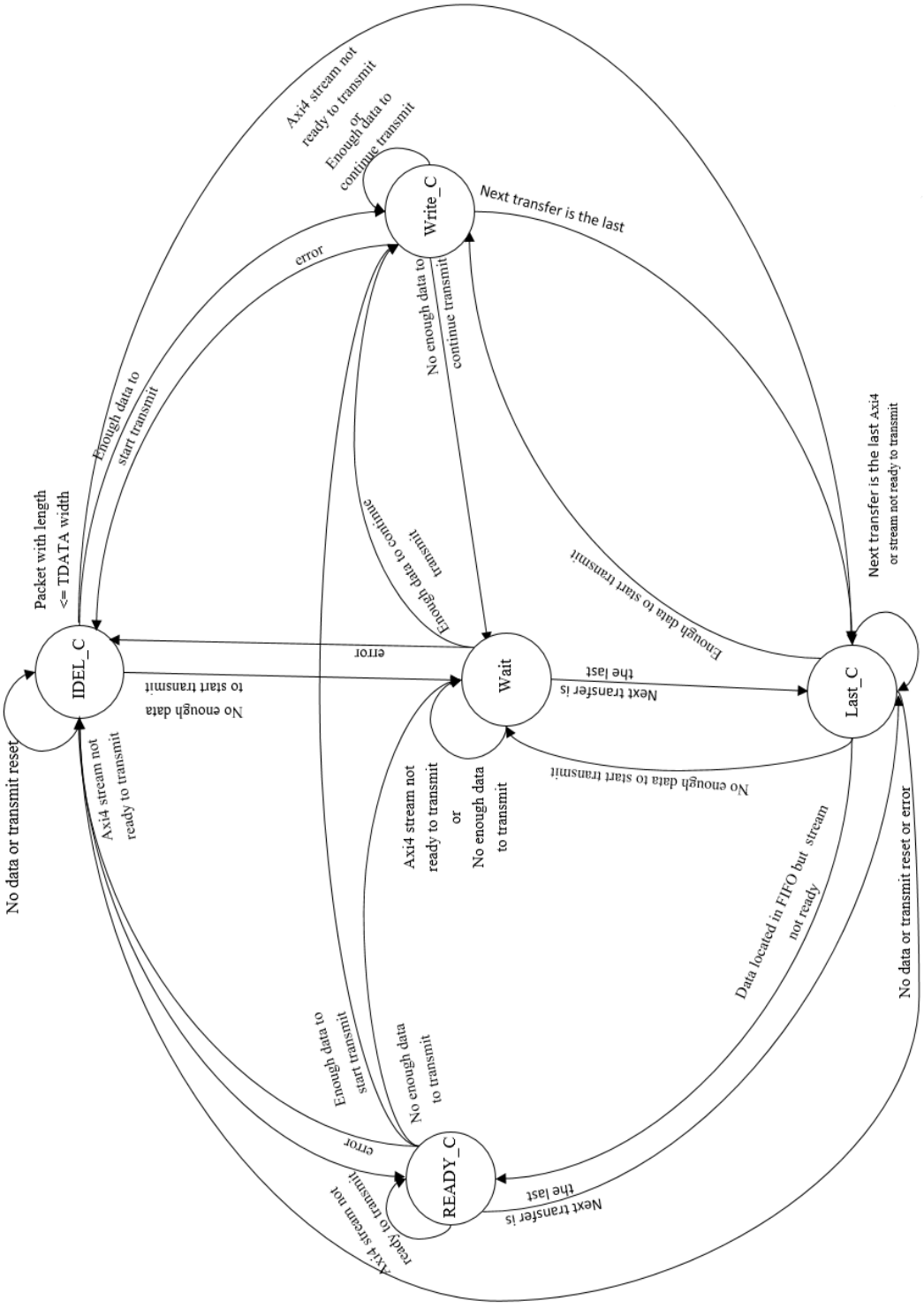


Figure 3-39 Cut-through mode finite state

Figure 3.40 shows Stream Mapper finite state machine in the two modes of operation

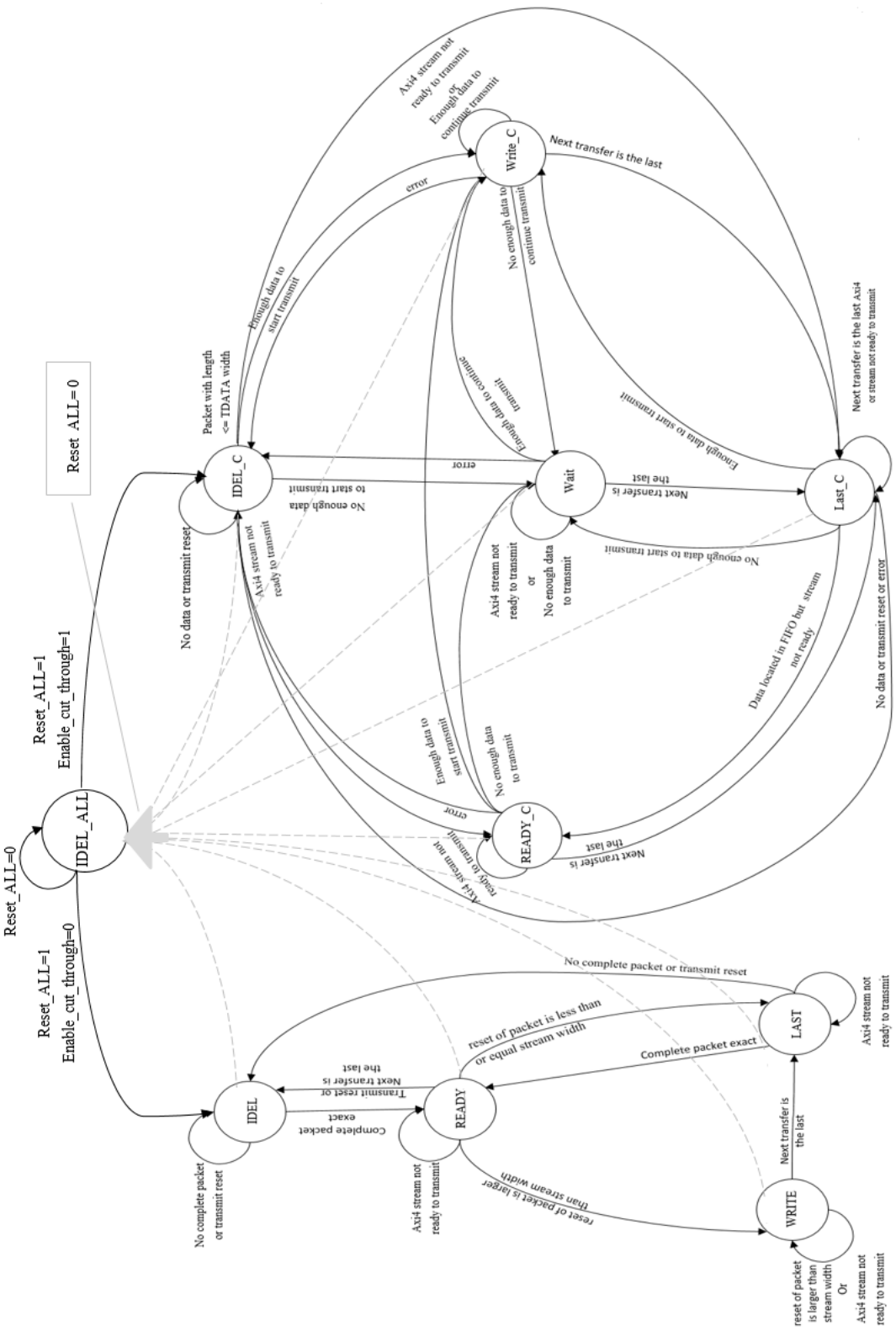


Figure 3-40 Stream Mapper finite state

Table 3.17 illustrates the states of the Stream Mapper FSM

State 0			
IDLE_All			
Description:			
Initial state of the system, in this state mode of operation is detected; at any state if **Reset_All** is low system will return to this state.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
Reset_All=1 enable_cut_through=0	Store and forward mode	IDLE	TVALD=0 TLAST=0
Reset_All=1 enable_cut_through=1	Cut through mode	IDLE_C	TVALD=0 TLAST=0

State 1			
IDLE			
Description			
This is the initial state for a Store-and-Forward mode of operation, When the transmit length FIFO is not empty this indicates that at least one complete packet is received and stored in the transmit data FIFO. In this state packet length is read bur transmission over AXI4-Stream Transmit Data Channel starts when **TREADY** is high.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
RESET=0	Transmit reset	IDLE	TVALD=0 TLAST=0 RESET_COMPLETE=1
RESET=1 empty_L=1	No complete packet	IDLE	TVALD=0 TLAST=0 RESET_COMPLETE=0
RESET=1 empty_L=0	Axi4 stream not ready to transmit	READY	TVALD=0 TLAST=0

State 2			
READY			
Description			
This is a waiting state until **TREADY** is high. When this condition is satisfied **TVALD** is raised, packet destination and first transfer of the packet is transmitted along with **TKEEP** signal.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0

RESET=0 Reg_TREADY=0	Transmit reset	IDLE	TVALD=0 TLAST=0
Len → 1:4 Reg_TREADY=1	Next transfer is the last	IDLE	TVALD=1 TLAST=1 Out TDES Out TDATA
RESET=1 Reg_TREADY=0	Axi4 stream not ready to transmit	READY	TVALD=0 TLAST=0
Len>8 Reg_TREADY=1	reset of packet is larger than stream width	WRITE	TVALD=1 TLAST=0 Out TDES Out TDATA
Len1>=0 Len1<=4 Reg_TREADY=1	Reset of packet is less than or equal stream width	LAST	TVALD=1 TLAST=0 Out TDES Out TDATA

State 3			
WRITE			
Description			
Rest of the packet is transmitted in this state except for the last beat of data, using packet length to decide number of bytes read from data FIFO.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
Len1>4 TREADY=1	Reset of packet is larger than stream width	WRITE	TVALD=1 Out TDATA TLAST=0
TREADY=0	Axi4 stream not ready to transmit	WRITE	TVALD=1 TLAST=0
Len1=1:4 TREADY=1	Next transfer is the last	LAST	TVALD=1 Out TDATA TLAST=0

State 4			
LAST			
Description			
Last beat of data in transmitted in this state along with **TLAST** signal, if reset signal was low **RESET_COMPLETE** signal is raised and Transmit Unit is reset. IF transmit length FIFO is not empty this means that a new packet is received and ready to be transmitted and next_state is **READY** if **TREADY** signal is high, if length FIFO is empty block will go back to IDLE state.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0

RESET=1 empty_L=1 Reg_TREADY=0	No complete packet transmit reset	IDLE	Out TDATA TVALD=1 TLAST=1
TREADY=0	Axi4 stream not ready to transmit	LAST	
RESET=1 empty_L=1 Reg_TREADY=1	Complete packet exact	READY	

State 5			
IDLE_C			
Description			
This is the initial state in Cut-Through mode of operation. If transmit data FIFO is not empty this indicates that a partial packet is stored in the transmit data FIFO, if the occupancy of transmit data FIFO is less than or equal **TDATA** Width, transmission over AXI4-Stream Transmit Data Channel will not start until occupancy of transmit data FIFO is higher than **TDATA** Width, or transmit length FIFO is not empty to ensure that the last beat of data in only transmitted when packet length is entered by user.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Resest core	IDLE_All	TVALD=0 TLAST=0
RESET=0	Transmit reset	IDLE_C	TVALD=0 TLAST=0 RESET_COMPLETE=1
RESET=1 empty_Data=1	No data	IDLE_C	TVALD=0 TLAST=0 RESET_COMPLETE=0
empty_Data=0 empty_D=0 TREADY=0	Axi4 stream not ready to transmit	READY_C	TVALD=1 TLAST=0
RESET=1 TREADY=1 empty_D=0 empty_Data=0 empty_L=1 capacity<= 4	No enough data to start transmit	Wait	TVALD=0 TLAST=0
RESET=1 TREADY=1 empty_D=0 empty_Data=0 capacity> 4	Enough data to start transmit	Write_C	TVALD=0 TLAST=0
RESET=1 TREADY=1 empty_D=0 empty_Data=0 empty_L=0	Packet with length <= TDATA width	Last_C	TVALD=1 TLAST=0

capacity<= 4			

State 6			
READY_C:			
Description			
This is a waiting state until **TREADY** signal becomes high.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
Error=1	Error	IDEL_C	TVALD=0
TREADY=0	Axi4 stream not ready to transmit	READY_C	TVALD=0 TLAST=0
capacity<= 4 empty_L=1	No enough data to transmit	Wait	TVALD=0 TLAST=0
capacity> 4 empty_L=0 data_out_L>4 Or capacity> 4 empty_L=1	Enough data to start transmit	Write_C	TVALD=0 TLAST=0
capacity> 4 empty_L=0 data_out_L<=4 or capacity<=4 empty_L=0	Next transfer is the last	Last_C	TVALD=0 TLAST=0

State 7			
Wait			
Description			
This is a waiting state until occupancy of transmit data FIFO is more than **TDATA** width or transmit length FIFO becomes not empty			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
TREADY=0 or RESET=1 TREADY=1 empty_L=1 capacity<=4	Axi4 stream not ready to transmit or No enough data to transmit	Wait	TVALD=0 TLAST=0
Error=1	Error	IDEL_c	TVALD=0 TLAST=0
RESET=1 TREADY=1	Enough data to	Write_C	TVALD=0 TLAST=0

capacity> 4	continue transmit		
RESET=1 TREADY=1 capacity> 4 (packet_len=data_out_L-4)|| (packet_len> data_out_L- 4) &packet_len< data_out_L Or RESET=1 TREADY=1 capacity<= 4 empty_L=0	Next transfer is the last	Last_C	TVALD=0 TLAST=0

State 8			
Write_c			
Description			
In this state **TVALD** is raised, destination and of data are transmitted along with **TKEEP** signal until occupancy of data FIFO is less than or equal **TDATA** width or transmit length FIFO becomes not empty, when this condition is satisfied block moves to **LAST_c** state under the condition that packet length is received.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
Error=1	Error	IDEL_C	TVALD=0
TREADY=0	Axi4 stream not ready to transmit	Write_c	
TREADY=1 capacity> 4 empty_L=1 length_c=0 Or TREADY=1 capacity> 4 empty_L=0 length_c=0 packet_len=4 packet_len<length_c-4 Or TREADY=1 capacity> 4 length_c>0 packet_len<length_c-4	Enough data to continue transmit	Write_c	TVALD=1 Out TDES Out TDATA
TREADY=1 capacity<= 4 empty_L=1 length_c=0	No enough data to continue transmit	Wait	TVALD=0 TLAST=0

TREADY=1 capacity<= 4 empty_L=1 length_c>0 Or TREADY=1 capacity> 4 length_c>0 packet_len<length_c- 4 Or TREADY=1 capacity<= 4 empty_L=0	Next transfer is the last	Last_C	TVALD=1 TLAST=0

State 9			
Last_c			
Description:			
In this state last beat of data is transmitted along with TLAST signal, if reset signal is low **RESET_COMPLETE** is raised and Transmit FIFO Unit is reset. If transmit data FIFO is not empty, a partial packet is stored in the transmit Data FIFO. If the capacity of transmit data FIFO is less than or equal **TDATA** width, transmission over AXI4-Stream Transmit Data Channel will not start until capacity of transmit data FIFO is higher than **TDATA** width, or transmit length FIFO not empty.			
Condition	**Description**	**Next state**	**Output**
Reset_All=0	Reset core	IDLE_All	TVALD=0 TLAST=0
TREADY=1 RESET=0 Or TREADY=1 RESET=1 empty_Data=1	No data or transmit reset	IDEL_C	TVALD=1 TLAST=1 Out TDES Out TDATA
Error=1	error	IDEL_C	TVALD=0 TLAST=0
RESET=1 TREADY=1 empty_D=0 empty_Data=0	Data located in FIFO but stream not ready	READY_C	TVALD=1 TLAST=1 Out TDES Out TDATA
RESET=1 TREADY=1 capacity<= 4 empty_D=0 empty_Data=0 empty_L=1 Or RESET=1 TREADY=1 capacity<= 4	No enough data to start transmit	Wait	TVALD=1 TLAST=1 Out TDES Out TDATA

empty_D=1 empty_Data=0 empty_L=1			
RESET=1 TREADY=1 capacity> 4 empty_L=0 empty_D=0 empty_Data=0 data_out_L>4 length_c=0 or RESET=1 TREADY=1 capacity> 4 empty_L=1 empty_D=0 empty_Data=0	Enough data to start transmit	Write_C	TVALD=1 TLAST=1 Out TDES Out TDATA
RESET=1 TREADY=1 capacity> 4 empty_L=0 empty_D=0 empty_Data=0 length_c=0 data_out_L<=4 Or RESET=1 TREADY=1 capacity<= 4 empty_L=0 empty_D=0 empty_Data=0	Next transfer is the last Axi4	Last_C	TVALD=1 TLAST=1 Out TDES Out TDATA
TREADY=0	stream not ready to transmit	Last_C	

Table 3-17 Stream Mapper FSM states

3.3.6. Stream Receive Interface

Description

Stream Receive Interface is the first block in the received data flow of the receive path of AXI4-Stream FIFO. It's the intermediate block between the external Stream interface and the Receive FIFO. The main responsibility of the Stream Receive Interface is to receive data in form of packets and their destination from the external Stream interface, calculate the received packet length and write data, destination and packet length into Receive FIFO.

Receive Stream Interface block communicates with external Stream interface and the Receive FIFO.

Receive Stream Interface has one enable signal connected to the Receive FIFO block. The block also calculates number of data bytes in each data transfer and signals this value along with the enable signal to the Receive FIFO (the calculation will be multiple of the data width in bytes as the data width of Receive FIFO and Receive AXI Stream is considered to be the same).

Interfacing

The I/O signals of Receive AXI Stream is listed in Table 3.1

Port name	Port mode	Connection	Description
TCLK	Input	Interface	Global interface clock
TReset	Input	Register Space	Resets entire core
TReset_rx	Input	Register Space	Resets receive logic
TVALID	Input	External stream Interface	Enables handshaking with Receive AXI Stream
TData_in	Input	External stream Interface	Receives data during handshaking
TDest	Input	External stream Interface	Receives destination during handshaking
TKeep	Input	External stream Interface	AXI4 Keep signal
TLast	Input	External stream Interface	Determines the end if the transaction
TReady	Output	External stream Interface	Enables handshaking with External stream Interface
Data_fifo_full	Input	Receive FIFO	Indication that data FIFO is full to stop handshaking
Packet_length	Output	Receive FIFO	Length written in length FIFO
TData	Output	Receive FIFO	Data written in data FIFO
Pass_length	Output	Receive FIFO	Enable for destination, data and length FIFOs
Length_reset	Output	Receive FIFO	Resets calculation of the packet length

Table 3-18 Receive AXI4 Stream I/O signals

Functionality

The block is responsible for performing the following functions

1. Receive data, destination and length from external stream interface
2. Calculate the packet length
3. Enable writing in Receive FIFO
4. Determining whether the packet is partial packet or complete packet in cut through mode.

External Stream interface as any AXI4 Stream will contain some essential signals:

- TValid: signal that will enable or start handshaking with Stream Receive Interface.
- TReady: signal that indicates that Stream Receive Interface is ready to send data and destination to it.
- TKeep: signal that determines the position and data bytes.
- TLast: signal that determines the end of the transaction.

So the first function is performed by handshaking with the external interface by TValid and TReady, so the data and destination will be transferred to Stream Receive Interface. The handshaking is stopped if data FIFO is full.

The second function is performed by checking the pass_length signal that will be equal to 1 during handshaking then packet length will be calculated as follows: Packet length (in bytes) = no of data transactions received * data width in bytes

The calculation will stop if length_reset signal is equal to 1 at the end of the transaction, means the packet has ended.

The third function is also performed by checking pass_length signal that will be equal to 1 during handshaking then it data, destination and length will be written in the Receive FIFO, then the writing will stop if length_reset signal is equal to 1 at the end of the transaction.

The fourth function is performed by assigning the 31st bit of packet length to 1 if the packet is not completed then assign the bit to 0 if the packet is completed which will happen at the end of the transaction (end of packet when TLast=1)

The functionality of Receive AXI Stream block is illustrated in the flow chart in figure 3.41

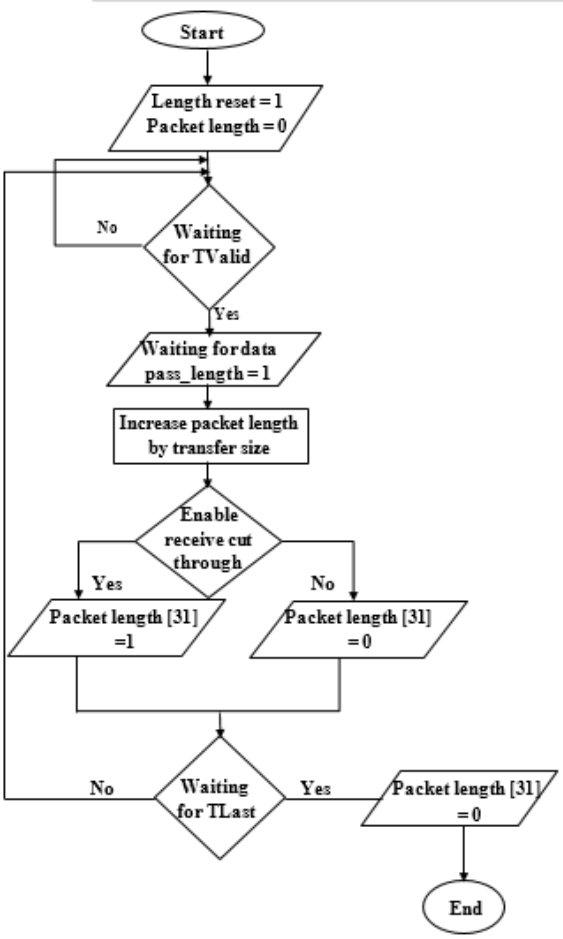


Figure 3-41 AXI4-Stream interface flow chart

Implementation

Receive AXI Stream block shown in figure 3.42 is using a Moore finite state machine to perform the functions mentioned earlier.

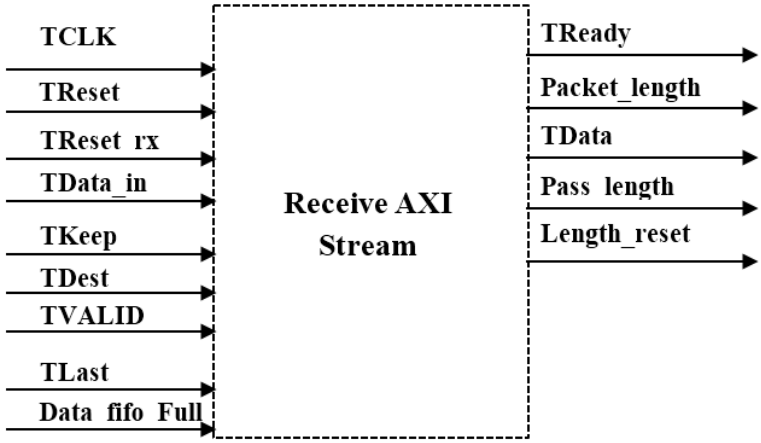


Figure 3-42Receive AXI4 Stream block

Receive AXI Stream Finite State Machine

The Moore finite state machine states are illustrated in the table 3.19 and figure 3.43

State	Description
Idle	Idle state when Receive Stream is not active
Data wait	Waiting for Data after TValid is asserted from external interface to start transaction
Data wait for Tvalid	Wait for TValid during transaction

Table 3-19Receive AXI4 Stream FSM states

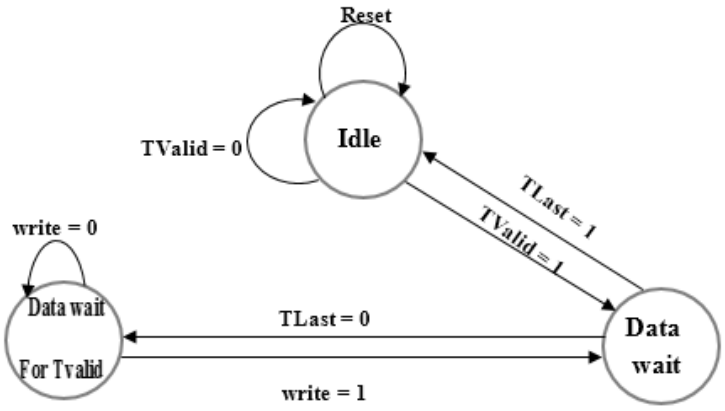


Figure 3-43 Receive AXI4 Stream FSM

The finite state machine transition conditions are illustrated in the table 3.20

Transition condition	Description	Condition
Reset	reset for receive circuitry	reset or reset_rx == 1
Write	Write in data FIFO	TValid and !data_fifo_full

Table 3-20Receive AXI4 Stream FSM conditions

The finite state machine outputs corresponding to each state are illustrated in table 3.21.

Table 1.14: Receive AXI Stream FSM outputs

Outputs	State number		
	Idle	Data wait	Data wait For TValid
TReady	0	1	0
Length_reset	1	0	0
Pass_length	0	1	0

Table 3-21 Receive AXI4 Stream FSM outputs

3.3.7. Receive FIFO

Description

Receive FIFO is the second block in the data flow of the receive path of AXI4-Stream FIFO and is the intermediate block between the Stream Receive Interface and other blocks in the receive flow. Block main responsibility is to store data, destination and length for each packet from Stream Receive interface, and it allows the Receive control block to read from FIFOs.

Receive FIFO block contains three FIFOs:

- Destination FIFO to store destination from Stream Receive Interface
- Length FIFO to store packet length calculated internally in Stream Receive Interface
- Data FIFO to store data from Stream Receive Interface.

Receive FIFO block communicates with three blocks which are:

- Receive control: Receive FIFOs enable Receive Control to read destination, data and length from them.
- Interrupt Interface: Interrupt Interface sets the Receive FIFO Programmable Empty (RFPE - bit 19 in ISR) to 1 if data FIFO is empty and sets Receive FIFO Programmable Full (RFPF – bit 20 in ISR) to 1 if data FIFO is full according to the Receive FIFO information.
- Register Space: Data from Receive data FIFO is passed to RDFD register, while destination is passed to RDR and length to RLR.

Interfacing

The I/O signals of Receive AXI Stream is listed in Table 3.22

Port name	Port	Connection	Description
CLK	Input	Interface	Global interface clock
Reset	Input	Register space	Resets entire core
Reset_rx	Input	Register space	Resets receive logic
Data_in	Input	Stream Receive Interface	Input data from Stream Receive Interface
Length_in	Input	Stream Receive Interface	Input length from Stream Receive Interface
Dest_in	Input	Stream Receive Interface	Input destination from Stream Receive Interface
Pass	Input	Stream Receive Interface	Enable to write in FIFOs
Length_reset	Input	Stream Receive Interface	Indication of the end of the transaction in Stream Receive Interface
Data_rd_enable	Input	Receive Control	Enable to read data from data FIFO
Length_rd_enable	Input	Receive Control	Enable to read length from length FIFO
Dest_rd_enable	Input	Receive Control	Enable to read destination from destination FIFO
Data_out	Output	Register space	Output data from data FIFO
Length_Out	Output	Register space Receive Control	Output length from length FIFO
Dest_Out	Output	Register space	Output destination from destination FIFO
Data_fifo_full	Output	Interrupt Interface	Indication that data FIFO Is full
Data_fifo_empty	Output	Interrupt Interface	Indication that data FIFO Is empty
Length_fifo_empty	Output	Receive Control	Indication that length FIFO is empty
Reset_rx_complete	Output	Interrupt Interface	Reset of the receive logic is completed
Prev_location	Output	Calculation Unit	Length of last completed packet

Table 3-22Receive FIFO I/O signals **Functionality**

The block is responsible for performing the following functions

1. Enable Stream Receive Interface to write in FIFOs
2. Enable Receive Control to read from FIFOs
3. Determining that FIFOs are full or empty
4. Determining the Receive data FIFO occupancy (RDFO)

As mentioned earlier. Receive FIFO contains three FIFOs; data, length and destination. Each FIFO has read and write pointers so locations can be read from and written into.

So the first function is performed by the pass signal from Stream Receive Interface that is an enable to write in the FIFOs. Writing in each FIFO will be explained in the following paragraphs.

- Writing in data FIFO is very simple as if it is not Full, the data is written to the location determined by data write pointer and data write pointer is increased by one.

- Writing in destination FIFO is similar to data FIFO as if it is not Full, the destination is written to the location determined by destination write pointer and destination write pointer is increased by one only at the end of the transaction determined by length_reset signal.

- Writing in length FIFO is different from data and destination FIFO as AXI4-Stream FIFO has two mode as follows:

    1. Store and forward: writing only happens at the end of the transaction determined by length_reset signal and length write pointer is increased by one if FIFO is not full.

    2. Cut through: writing in same location the length received from Stream Receive Interface during transaction then length write pointer is increased by one at the end of the transaction determined by length_reset signal if FIFO is not full.

The second function is performed by three enable signals; one signal for each FIFO from Receive Control. Reading from each FIFO will is explained as follows:

- Reading from data FIFO is as simple as writing as if data_rd_en signal is asserted from Receive Control, the data is read from location determined by data read pointer and data read pointer is increased by one only if data FIFO is not empty.

- Reading in length FIFO is also different from data FIFO as if length_rd_en signal is asserted from Receive Control, the length is read from location determined by length read pointer then determining that the packet is completed or not by 31st bit in length.

    1. If this bit is equal to 0 which means the packet is completed, length read pointer is increased by one only if length FIFO is not empty.

    2. If this bit is equal to 1 which means the packet is not completed yet, length read pointer is in the same location.

- Reading from destination FIFO is similar to length FIFO as If dest_rd_en signal is asserted from Receive Control, the destination is read from location determined by destination read pointer then determining that the packet is completed or not by 31st bit in length using the same technique mentioned above.

The Third function is performed by calculating the difference between write and read pointers. The full signal is asserted if the difference is equal or more than the full threshold and the Empty signal is asserted if the difference is equal or less than empty threshold.

The fourth function is performed by viewing the last complete packet written in length FIFO so that it will be an indication of the occupancy in data FIFO.

Implementation

The implementation of Receive FIFO is illustrated by the block diagram in figure 3.44

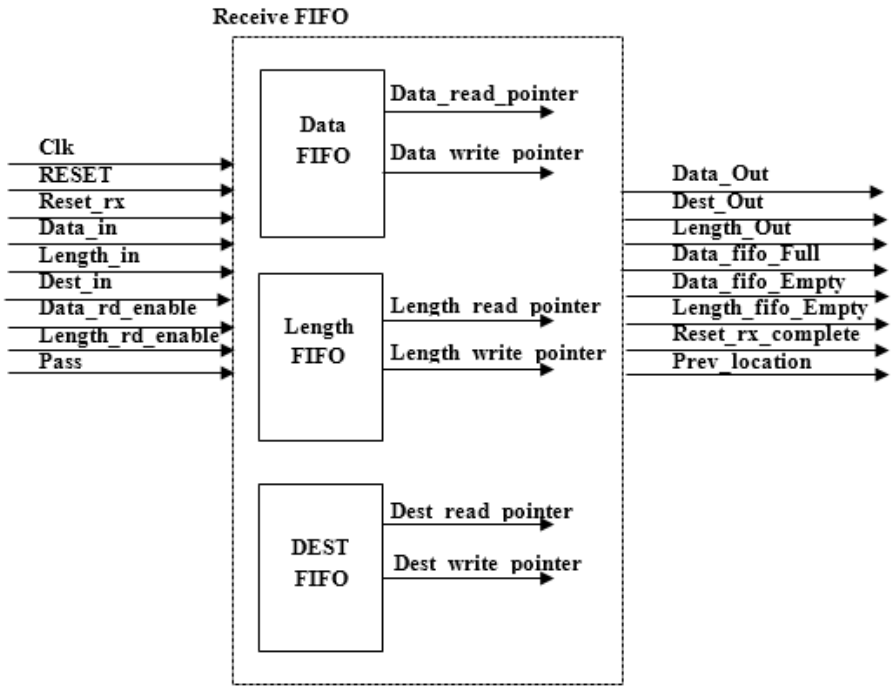


Figure 3-44Receive FIFO block

3.3.8. Receive Control

Description

Receive Control is the second block in the Control Unit and it is the block that handles communication between Receive FIFO and the registers related to the receive path in the Register Space. Block main responsibility is to allow data written by Stream interface to be transferred from Receive FIFO into Register Space and subsequently be passed to AXI4/AXI4-Lite interface; under the condition that user has performed a correct programming sequence to receive each packet correctly.

Basic idea of the receive control unit is to always be one step ahead the user. The user sequence is done by using the correct handshake of the AXI4-Lite interface and to access the register space using a write/read enable and register address. Every read or write in the register space through the interface is a step in the programming sequence and each performed step in the receive logic triggers the next one's data to be ready so that the user's data is ready before asking for it to enable faster operation.

Receive Control block communicates with the following blocks:

- AXI4/AXI-Lite interface: Communication with AXI4/AXI4-Lite interface is a one-way communication; means that the receive control block only use values from AXI4/AXI4-Lite interface to check for proper input programming sequence and to activate the receive logic sequence.

- Receive FIFO and Register Space: Communication with Receive FIFO, and Register Space is necessary to enable data transfer from the receive FIFO into the main receive registers in the register space; RDFD, RLR, RDR in the correct sequence. According to this, data types can be classified into three types; packet data (actual data), destination and length (exactly as receive FIFO classification specified earlier). Register space is responsible for resetting Receive Control when a user writes specific values in either SRR register or RDFR register,

- Interrupt Controller: Receive control calculates two interrupts for the Interrupt Interface to be added in the ISR; the first is RC (Receive Complete) that indicates that at least one successful receive has completed (from stream interface) and that the receive packet data and packet data length is available in the receive FIFO, the second is RPORE (Receive Packet Overrun Read Error) that indicates that more words are read from the receive data FIFO than are in the packet being processed.

Interfacing

The I/O signals of Receive Control is listed in Table 3.23

Port name	Port mode	Connection	Description
Clk	Input	Interface	Global interface clock
Reset	Input	Register Space	Resets entire core
recieve_reset	Input	Register Space	Resets receive logic
rg_read_enable	Input	AXI4 Lite Interface	Enables reading from register space
rg_read_address	Input	AXI4 Lite Interface	Read address for register space
rg_RDFD_enable	Output	Register space	Enable for writing in RDFD
rg_RLR_enable	Output	Register space	Enable for writing in RLR
rg_RDR_enable	Output	Register space	Enable for writing in RDR
on_off	Output	Interrupt interface	Indicates when receive sequence is active
rf_length_empty	Input	Receive FIFO	Indicates whether the length FIFO is empty
recieve_fifo_RLR	Input	Receive FIFO	[22:0]: Number of bytes in the packet that is being processed bit 31: indicates a partial or a complete packet The rest of bits are reserved
rf_length_enable	Output	Receive FIFO	Enable reading from receive length FIFO
rf_dest_enable	Output	Receive FIFO	Enable reading from receive destination FIFO
rf_data_enable	Output	Receive FIFO	Enable reading from receive data FIFO
ic_RC_26	Output	Interrupt interface	Indicates that at least one successful receive has completed and that the receive packet data and length are available.
ic_RPORE_30	Output	Interrupt interface	Indicates that more words are read from the receive data FIFO than are in the packet being processed.
ic_process_indication	Output	Interrupt interface	Indicates that the length or data is being written to register space from receive FIFO

Table 3-23Receive Control I/O signals

Functionality

The block is responsible of performing the following two functions

1. Allow data flow from Receive FIFO (written by stream interface) into Register space when user performs correct receive programming sequence, and then data can be transferred to the AXI4/AXI4-Lite interface with the proper handshaking
2. Handle Receive Packet Overrun Read Error (RPORE) bit in the ISR register

The first function is performed by monitoring read operations done in four registers in the Register Space:

- ISR: register that is responsible for interrupts, it's important to Receive Control block as reading operation to ISR triggers the programming sequence (either in the power up or receive packet sequences)
- RLR: register for received packet length
- RDR: register for received packet destination
- RDFD: register for received packet data

Generally, the receive logic can deal with more registers than that mentioned above which are:

- Receive ID Register: Not currently supported in the standard
- Receive USER Register: Not currently supported in the standard
- SSR and RDFR: The Register space deals with these registers and sends a reset signal to the Receive Control when specific values are written to any of these two registers.
- RDFO: Occupancy value is important for the user but it doesn't strongly affect the sequence so the Receive control won't focus on it, yet it's important for the user to read it in the right specified order to correctly read its value. RDFO value is calculated and updated separately from Receive control in the calculation unit as will be illustrated later.

A measure of success of the operation is that the data reach the other side which is the AXI4/AXI4-Lite and that can only be accomplished when user performs the programming sequence correctly, at this point the Receive Control will signal the appropriate enable signals (according to the sequence) to the Receive FIFO and the Register space to permit data flow from FIFO to the required registers.

The Receive control block doesn't transfer data itself from the receive FIFO to the Register space, instead it just allows (or enables) the data transfer between them to enable a faster operation.

Modes of operation

The cut-through mode can be considered a generalized mode as it contains the store and forward operation inside, this can be proven after illustrating the concept of a complete and partial packets:

- Complete Packet: a packet that is fully written in the receive data FIFO and thus the length corresponding to it in the receive length FIFO is the final length of the packet, and when it exists it sets the value of receive complete bit in the ISR (RC) into 1 indicating that at least one successful receive has completed.
- Partial Packet: a packet that is not yet fully written in the receive data FIFO and thus its corresponding length is continuously being updated in the receive length FIFO.

In Cut-Through mode, the user can start reading packet as soon as the stream interface starts writing it into the core, producing a partial packet scenario, or wait until the packet is fully written into the core producing a complete packet scenario which is the only option in the store and forward mode, and thus the Cut-through is the general mode as mentioned earlier.

Also in the Cut-through mode, a partial packet can be read from the receive data FIFO part by part and the last part means that end of packet is present and thus the value of RC bit must be set to 1 in this case representing a completion of a successful receive just as the case of complete packet in the store and forward mode.

Store and forward mode (complete packets only)

To perform a correct receive programming sequence user must access the four registers mentioned earlier in a specific order; first one is the ISR from which the user will read the RC bit which will tell him when a complete packet is ready which means that its length is also available which leads us to the second register, the RLR that will contain the length of packet available (in bytes), the third register is the RDR that contains packet's destination, and the final register is the RDFD which has to be accessed number of times related to the length specified in the RLR and the width of RDFD (RLR data/width of RDFD in bytes) in order to read all the data in the packet in process.

Cut-through mode (complete and partial packets)

The most important factor in this mode is whether the packet being processed is a partial packet or a complete packet and that is determined by the most significant bit (bit number 31) in the corresponding length of this packet in the receive length FIFO or in the RLR. The Receive stream interface and receive FIFO is responsible for determining and updating this value. If this bit is 1, it indicates that a partial packet is available in the receive data FIFO, and the sequence of order is different of the one that is mentioned above.

The sequence is started by first reading the ISR values, and as it's a partial packet so the RC bit value will be zero informing the user that there is no successful receive is completed yet, second the user reads the RLR register finding the bit 31 to be 1 indicating a partial packet and that the length existing in the RLR is not the final packet length, the third register will be the RDFD which has to be accessed a specific number of times as explained before. And the user starts to read the ISR again to check for the RC value and repeats this sequence until RC is found to be 1 which means that this is the final part of the packet, so it's now a complete packet and the remaining sequence will be just as the one in store and forward mode mentioned above.

Note: in the store and forward mode, we are not interested in the bit 31 in receive length FIFO or in the RLR as it'll always be zero as this mode doesn't allow partial packet operation.

The second function of Receive control is to handle Receive Packet Overrun Read Error (RPORE) bit in the ISR register. The RPORE indicates that more words are read from the receive data FIFO (user read it through RDFD) than that specified in the packet being processed (in the RLR).

Since the Receive control block monitors the sequence by checking for user reading certain registers at certain order, it's aware of the registers that user accesses and more important and unique to the receive control block, it's aware of the order of this access. So simply, the receive control counts the number that the user tries to read the RDFD and compares it with specified length in RLR, once the counter reaches the limit and user tries to read RDFD again, Receive control set the value of RPORE bit to 1 and stays in a stuck state without enabling any other data flow to prevent corrupting

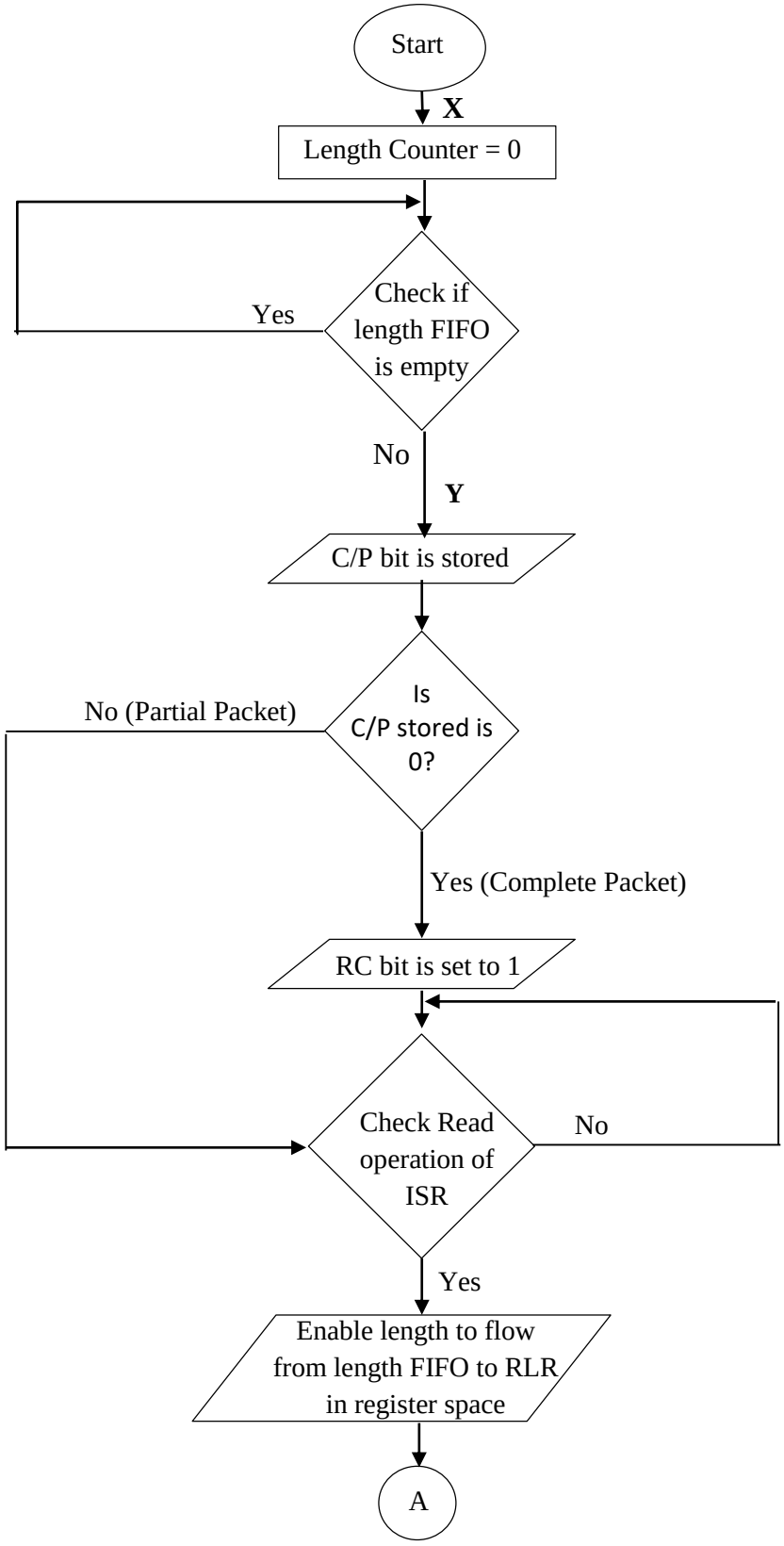
of any packet in the receive data FIFO, to recover from this stuck state, a reset signal must be sent to the Receive control block to allow further normal operation.

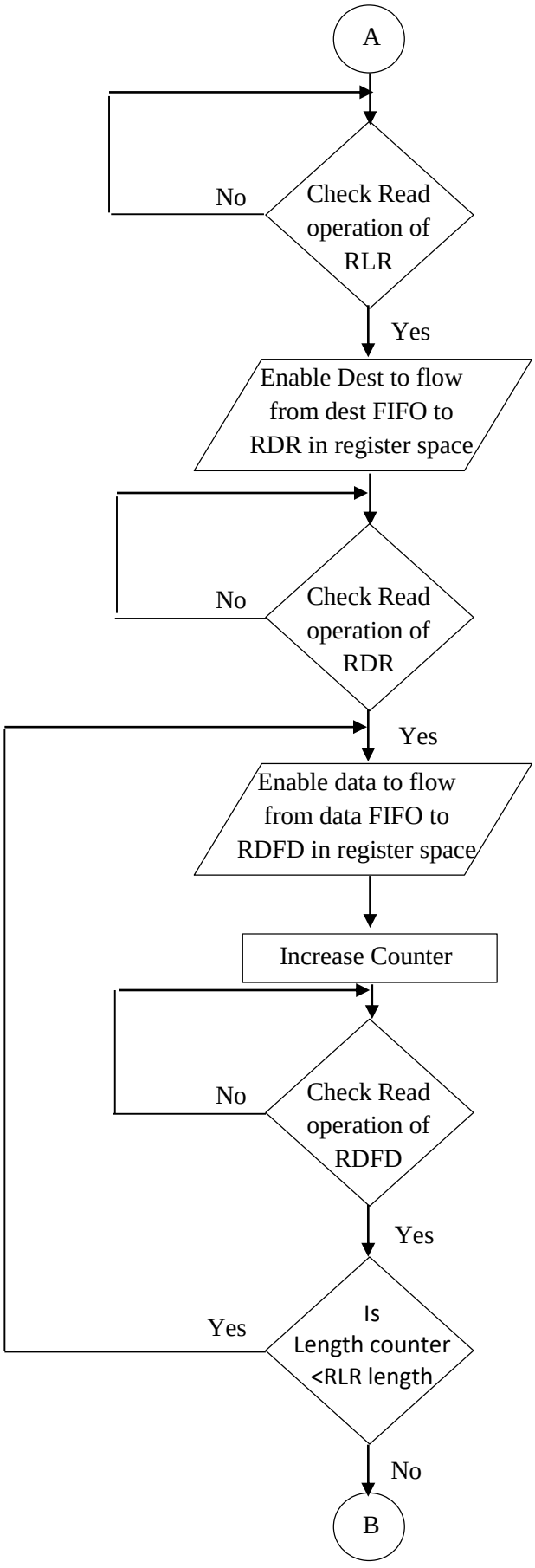
Notes:

- Writing correct programming sequence is the responsibility of the user. Wrong programming sequence will result in undesirable performance and user will need to reset receive circuitry to retrieve proper operation of interconnect.

- Receive Control doesn't require from the user to perform programming sequence in a specific number of clock cycles.

Receive Control block functionality is illustrated by the flow chart in figure 3.45




A
Check Read operation of RLR
No
Yes
Enable Dest to flow from dest FIFO to RDR in register space
Check Read operation of RDR
No
Yes
Enable data to flow from data FIFO to RDFD in register space
Increase Counter
Check Read operation of RDFD
No
Yes
Is Length counter <RLR length
Yes
No
B

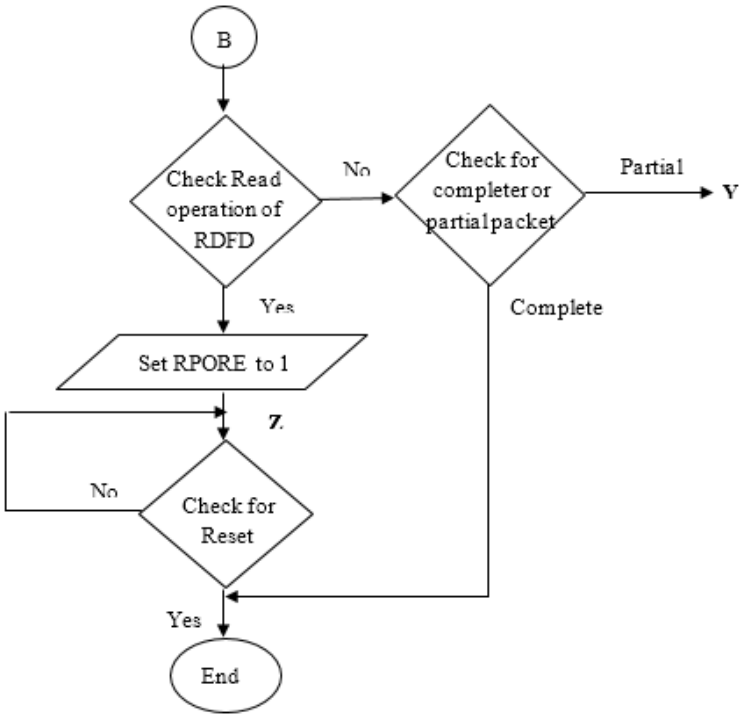


Figure 3-45 Receive Control functionality flow chart

- Resetting receive circuitry or entire core at any point will result in going to start point (marked **X** at flow chart)
- Wrong programming sequence will result in going to stuck point and ROPRE will make the Receive control stuck too (marked **Z** at flow chart)
- End of flow chart represents the end of an entire packet, after which the sequence must go back at the start point again to check for existence of more packets (marked **X** at flow chart

Implementation

Receive control block is constructed mainly from a Moore finite state machine (FSM) that is responsible for monitoring and supporting the programming sequence as mentioned earlier.

The two modes of operation (store and forward/cut-through) have the same states but with different order of sequence according to the packet type; complete packet or partial packet.

The Moore finite state machine states are illustrated in the table 3.24

Number	State	Description
0	Idle	Idle state when receive control is not active
1	RC	Receive Complete (RC) state indicates that at least one successful receive has completed and packet data and its length are available
2	Length	Length State that enables the length of packet (or part of it) to flow from receive length FIFO to the RLR in register space
3	Destination	Destination State that enables the destination of a packet to flow from receive destination FIFO to the RDR in register space
4	Data	Data State that enables the data in a packet to flow from receive data FIFO to the RDFD in register space
5	Wait	Waite state is almost the intermediate state between all states, as the receive logic moves to it whenever there's a delay in the programming sequence, providing the user freedom to choose the reading rate, the only constrain in this case is the depth of receive FIFO that can be totally full if the user waits tool long to perform the receive programming sequence (reading operation)
6	RPORE	RPORE state at which RPORE bit in the ISR is set to 1 indicating an error occurred in reading the packet (more words are read from the receive data FIFO than that specified in the packet being processed)
7	Stuck	Stuck state at which the receive logic moves to when an RPORE occur, and stays in it until a reset signal is sent to it to get it back to Idle state

Table 3-24 Receive Control FSM states

Color Coding for the FSM (as illustrated in the following figure 3.46**):**

- Represent the path of a complete packet (store and forward mode and the last beat of the cut-through mode partial packet that is considered to be a complete packet).
- Represents the path of a partial packet that only exists in the cut-through mode.
- Represents the scenarios that result in a RPORE
- Represents resetting the receive logic

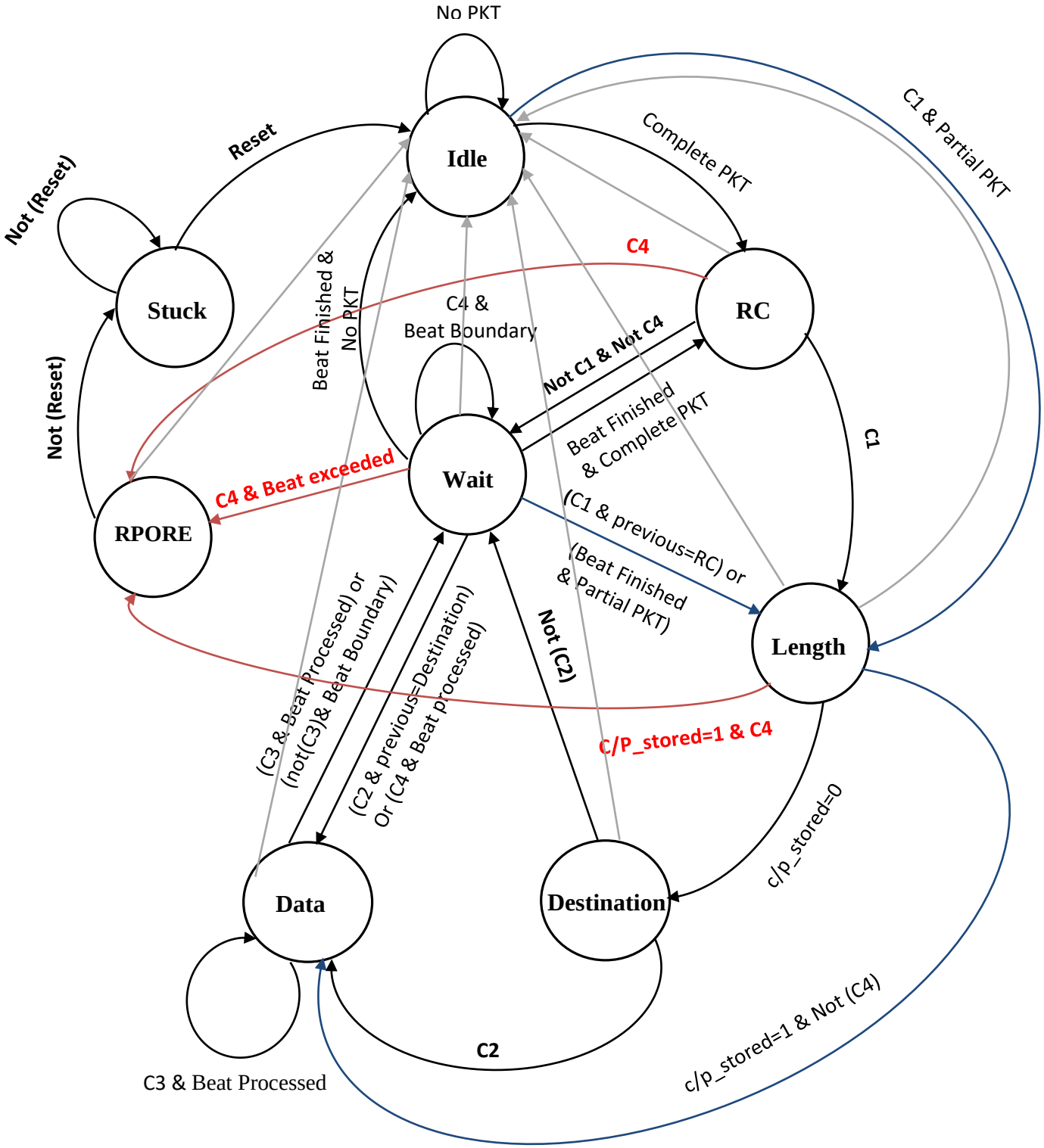


Figure 3-46 Receive Control FSM of both store and forward and cut-through modes

Note: At any state, if Reset conditions are satisfied, the next state will be the idle state, it's drawn in lite grey in the above FSM graph just for clarity of other states and conditions.

The finite state machine transition conditions are illustrated in the table 3.25

Transition condition	Description	Condition
Reset	It resets receive logic by forcing FSM to go to Idle state	reset=0 or recieve_reset=0
Partial PKT	Indicates a partial packet exists in the receive data FIFO	rf_length_empty=0 C1 & recieve_fifo_RLR[31]=1
Complete PKT	Indicates a complete packet exists in the receive data FIFO	rf_length_empty=0 C1 & recieve_fifo_RLR[31]=0
No PKT	Store and forward: Indicates that there's no complete packet in the receive data FIFO Cut-through: Indicates that there's no data at all in the receive data FIFO	rf_length_empty=1
C1	Read operation in ISR register	rg_read_enable=1 & rg_read_address=rg_ISR_addr
C2	Read operation in RLR register	rg_read_enable=1 & rg_read_address=rg_RLR_addr
C3	Read operation in RDFD register	rg_read_enable=1 & rg_read_address=rg_RDFD_addr
C4	Read operation in RDFD register while the previous step in the sequence was a read operation in RDFD too	previous_state==Data_state & rg_read_enable &&rg_read_address=rg_RDFD_addr
Beat Processed	Beat is being processed and data is still being read from RDFD	length_counter<(length_limit-1)
Beat Boundary	Last part of beat data is being read from RDFD	length_counter=(length_limit-1)
Beat Finished	The beat is finished	length_counter=length_limit & previous_state=Data_state
Beat Exceeded	The beat is exceeded and user tries to read a part from other packet	Length_counter= length_limit

Table 3-25Receive Control FSM transition conditions

Note: Beat is a name for the packet parts, and in case of store and forward mode the packet contains only one beat.

The finite state machine outputs corresponding to each state are illustrated in table 3.26

Outputs	**State number**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
On_off	0	1	1	1	1	1	1	0
ic_RPORE_30	0	0	0	0	0	0	1	0
ic_RC_26=0	0	1	0	0	0	0	0	0
rg_RLR_enable	0	0	1	0	0	0	0	0
rf_length_enable	0	0	1	0	0	0	0	0
rg_RDR_enable	0	0	0	1	0	0	0	0
rf_dest_enable	0	0	0	1	0	0	0	0
rg_RDFD_enable	0	0	0	0	1	0	0	0
rf_data_enable	0	0	0	0	1	0	0	0
ic_process_indication	0	0	1	As previous	1	As previous	0	0

Table 3-26 Receive Control FSM outputs

3.3.9. Calculation Unit

Description

Calculation unit is part of the Control Unit and doesn't provide any service for the internal blocks, so its outputs can only be accessed by user through AXI4-Lite interface. Unit is responsible for calculating vacancy of the transmit data FIFO and occupancy of the receive data FIFO and subsequently updating the registers; transmit data FIFO vacancy (TDFV) and receive data FIFO occupancy (RDFO) in the Register Space.

The unit interacts with the following blocks to perform its functionality; Transmit FIFO block and Receive FIFO block.

- Transmit FIFO calculates vacancy as number of bytes and this value is used by Calculation Unit to calculate vacancy as number of locations taking into consideration partial locations and updating TDFV register.

- Receive FIFO calculates packet length of each transaction through AXI4-Stream interface as number of locations; this value is used to update RDFO.

Interfacing

The Calculation Unit contains I/O signals listed in Table 3.27

Port name	Port mode	Connection	Description
Clk	Input	Interface	Global interface clock
reset_all	Input	Interrupt Interface	Resets entire core
reset_tx	Input	Interrupt Interface	Resets transmit circuitry
reset_rx	Input	Interrupt Interface	Resets receive circuitry
TX_fifo_vacancy	Input	Transmit FIFO	Vacancy as number of bytes
rg_fifo_vacancy	Output	Register Space	Transmit data FIFO vacancy
rg_TDFV_enable	Output	Register space	TDFV register enable
RX_fifo_occupancy	Input	Receive FIFO	Latest packet received length
rg_fifo_occupancy	Output	Register Space	Receive data FIFO occupancy
rg_RDFO_enable	Output	Register Space	RDFO register enable

Table 3-27Calculation Unit I/O signals

Functionality

Calculation unit is responsible for the following functions;

1. Update transmit data FIFO vacancy register (TDFV)
2. Update Receive data FIFO occupancy (RDFO)

First function is performed by comparing value of TDFV register with actual vacancy of transmit data FIFO; the value in TDFV register is incremented by one but decremented by two for each two write locations. Value in TDFV register doesn't give the exact number of bytes available for data storage in transmit data FIFO because the value in the register is the empty locations available so a vacant partial location is not considered.

TDFV register is updated as illustrated in the flow chart in figure 3.47

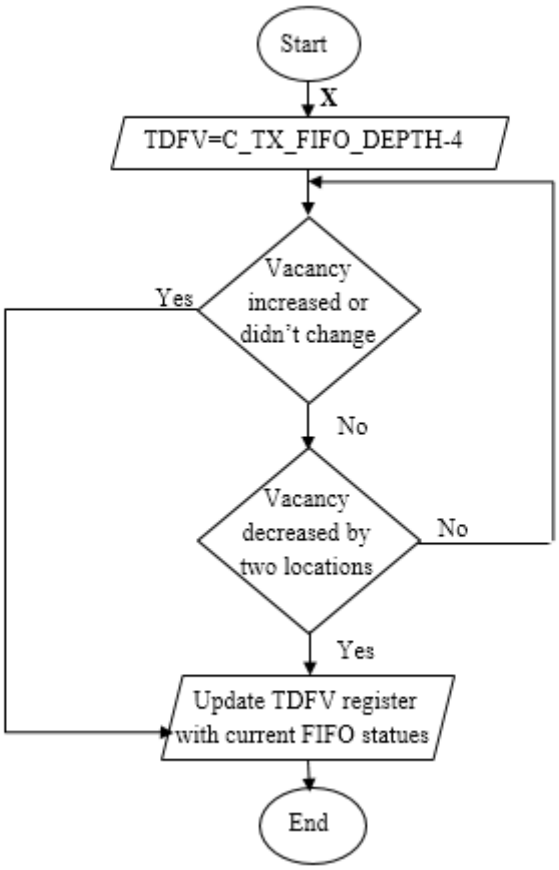


Figure 3-47 TDFV functionality flow chart

- Note: Resetting transmit circuitry or entire core will result in going to start point (marked **X** at flow chart)

The second function is performed by updating RDFO with the number of locations occupied by the latest received packet by AXI4-Stream interface. The value in the RDFO represents number of locations occupied by the last full packet stored at receive FIFO; hence if the receive FIFO is empty the value stored in RDFO register is zero. The register is only updated after a full packet is received.

RDFO is updated as illustrated in the flow chart in figure 3.48

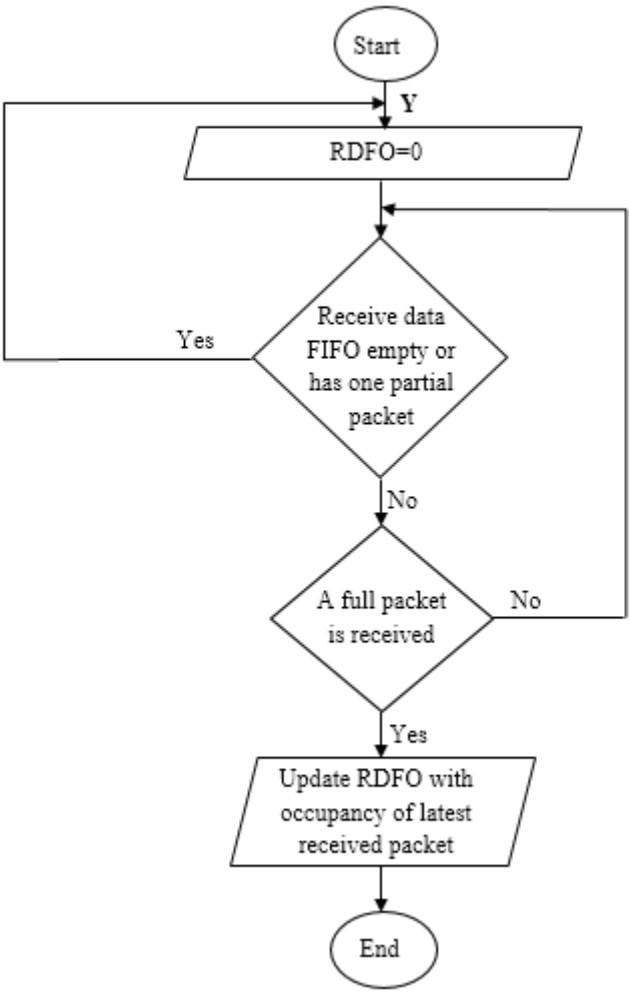


Figure 3-48 RDFO Functionality flow chart

- Note: Resetting receive circuitry or entire core at any time will result in going to start point (marked **Y** at flow chart)

Implementation

The Unit is divided into two main blocks to implement the two functions mentioned earlier. The block diagram of the Calculation Unit is shown in figure 3.49

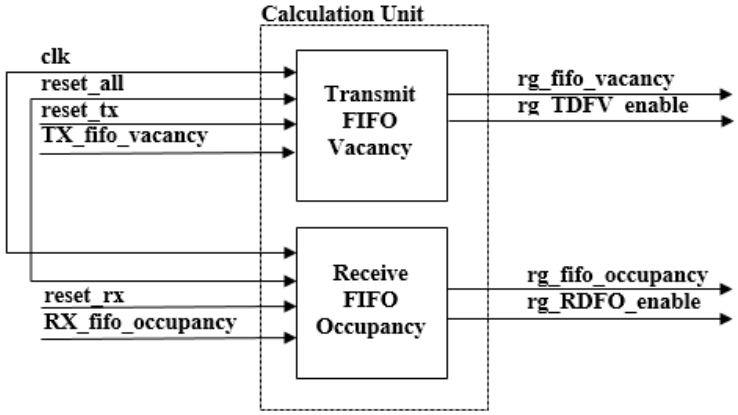


Figure 3-49 Calculation Unit block diagram

Transmit FIFO Vacancy

TX_fifo_vacancy is the vacancy of transmit data FIFO as number of bytes calculated by Transmit FIFO block; this value is used to calculate number of locations available for data storage in transmit FIFO and hence TDFV register is updated using this value by a couple of multiplexers to perform the following;

1. If Vacancy is increasing TDFV will represent current available locations in the transmit data FIFO is updated and
   - **rg_fifo_vacancy** = TX_fifo_vacancy >> $log_2\,(word\ size)$
     (word size can only take two values; 32 bits or 64 bits)

2. If vacancy is decreasing TDFV register is not updated with the actual vacancy value until vacancy decreases by two locations; TDFV will keep its value if the following condition is satisfied,
   - **TX_fifo_vacancy_temp**>>$log_2\,(word\ size)$ - **TX_fifo_vacancy**>>$log_2\,(word\ size)$ < 2

   Where **TX_fifo_vacancy_temp** is the last value of number of bytes used in updating TDFV register.

Receive FIFO Occupancy

Number of locations occupied is calculated for each packet stored in the receive data FIFO by the Receive FIFO block and stored in the receive length FIFO; hence number of locations occupied by the latest received packet can be retrieved by Receive FIFO block. **RX_fifo_occupancy** represents this value and is used to update RFDO register.

3.3.10. Interrupt Interface

Description

The interrupt Interface is responsible of generating the system interrupts to be stored in the ISR in the register space, the interrupts are either passed directly from other blocks or is generated from the block itself with the aid of external signals from the other blocks, the block is also responsible of the external interrupt bit to the user which is triggered when an interrupt is pulled high and the corresponding bit is set in the Interrupt Enable Register

Interfacing

The Interrupt Interface contains I/O signals listed in Table 3.28

Port Name	Port Mode	Connection	Description
ISR	Input	Register Space	The current status of the system interrupts, used to trigger the interrupt bit
IER	Input	Register Space	The mask of the enabled interrupts, used to trigger the interrupt bit
RLR_read_trial	Input	Register Space	Used to trigger receive underrun read interrupt
Receive_overrun	Input	Receive Control	Interrupt triggered from external block
Receive_empty	Input	Receive FIFO	Interrupt triggered from external block
Receive_full	Input	Receive FIFO	Interrupt triggered from external block
Transmit_empty	Input	Transmit FIFO	Interrupt triggered from external block
Transmit_full	Input	Transmit FIFO	Interrupt triggered from external block
Receive_reset_complete	Input	Receive FIFO and Stream Interface	Interrupt triggered from external block
Transmit_reset_complete	Input	Transmit Stream Interface	Interrupt triggered from external block
Receive_complete_signal	Input	Receive Control	Interrupt triggered from external block
Transmit_complete_Signal	Input	Transmit Stream Interface	Interrupt triggered from external block

Transmit_size_Error	Input	Transmit Control	Interrupt triggered from external block
Transmit_wr_en	Input	AXI4/AXi4-lite write interface	Used to trigger transmit overrun interrupt
Read_op	Input	Register Space	Used to trigger receive underrun interrupt
RLR_written	Input	Register Space	Used to trigger receive underrun read interrupt
Reset	Input	Register Space	To trigger Transmit and Receive Reset complete interrupts
Transmit_reset	Input	Register Space	To trigger Transmit Reset complete interrupt
Receive_reset	Input	Register Space	To trigger Receive Reset Complete interrupt
Interrupt_service	Output	Register Space	The status of the system interrupts to be stored in the ISR
Interrupt_bit	Output	External Signal	Alerts the user of the occurrence of an maskable Interrupt

Table 3-28 Interrupt Interface I/O signals

Functionality

Most of the interrupts triggered by other blocks in the form of a pulse for one clock cycle, the Interrupt interface, pulls the corresponding interrupt signal high till the user resets the interrupt, as

- Receive FIFO Empty
- Receive FIFO Full
- Transmit Empty
- Transmit Full
- Receive Reset Complete
- Transmit Reset Complete
- Receive Complete Signal
- Transmit Complete Signal
- Transmit Size Error

While the rest of the interrupts are calculated within the Interrupt Interface using external signals as

- Transmit Packet Overrun Error
- Receive Packet Underrun Error
- Receive Packet Overrun Read Error
- Receive Packet Underrun Read Error

Implementation

The Block Design of the Interrupt Interface is shown in Figure 3.50

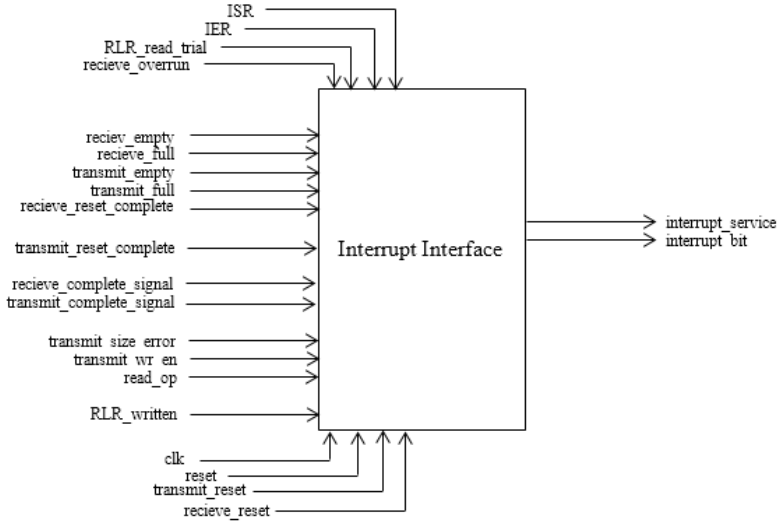


Figure 3-50 Interrupt Interface

Chapter 4.
FPGA Synthesis and Implementation

4.1.Introduction

Synthesis Overview

After design and behavioral simulation, It is not sufficient that the design can be implemented on ASIC or FPGA; design must be coded in such a way that it directs the synthesis tool to generate good hardware. A synthesis tool converts HDL (VHDL/Verilog) code into a gate-level net list, Synthesis report contains many useful information.

One should also pay attention to synthesis warnings since they can indicate hidden problems such as unintentional latches or unused signals, and after a successful synthesis. A gate-level schematic of the design can be produced by synthesis tool.

Implementation Overview

Implementation stage is intended to translate net list into the placed and routed FPGA design. For example, Xilinx design flow has three implementation stages: translate, map and place and route. (Implementation stages may be different for other tools)

- **Translate:** Combines all net lists and constraints into one large netlist

- **Map:** Compares the resources specified in the input netlist against the available resources of the target FPGA (insufficient or incorrectly specified resources generate errors). Divides net list circuit into sub-blocks to fit into the FPGA logic blocks.

- **Place & Route (PAR):** Iterative process, very time intensive. Places physically the sub-blocks into FPGA logic blocks. Routes signals between logic blocks such that timing constraints are met.

4.2.Results

The tools that have been used for the synthesis of the design are QLint tool which is tool supported by Mentor Graphics and Vivado design suite which is tool supported by Xilinx

4.2.1. Synthesis Results

After synthesizing the design on the board "XCVU440-FLGA2892-1-C" using the VIVADO synthesis tool, the utilization results of the resources are obtained as shown in the following figure 4-1.

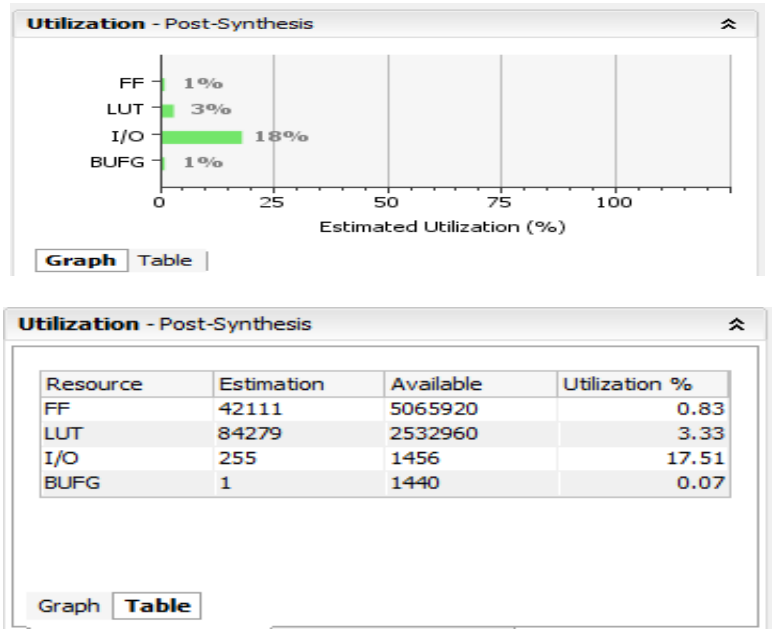


Figure 4-1Total utilization of board resources (Post-Synthesis)

For more details, the following figure shows how each individual module is synthesized and the number of its resources.

Name	CLB LUTs (2532960)	CLB Registers (5065920)	CARRY8 (316620)	F7 Muxes (1266480)	F8 Muxes (633240)	Bonded IOB (1456)	GLOBAL CLOCK BUFFERs (720)
top	84279	42111	177	6449	2431	255	1
calc_unit1 (calc_unit)	48	33	2	0	0	0	0
fifo_and_stream1 (fifo_and...	6048	17719	8	2342	1024	0	0
interrupt_controller1 (interr...	0	5	0	0	0	0	0
RC (Receive_Control)	163	64	16	0	0	0	0
register_file1 (register_file)	3286	619	0	0	0	0	0
stream_I (interface_s)	73619	23186	129	4107	1407	0	0
TC (Transmit_control_top)	851	276	22	0	0	0	0
top_LITE (top_a)	251	209	0	0	0	0	0

Figure 4-2Total utilization of board resources for each module (Post-Synthesis)

Analyzing the previous results of the main modules as shown in Figure 4-3:

Site Type	Used	Fixed	Available	Util%
Bonded IOB	255	0	1456	17.51

Figure 4-3 Number of bonded IOB in design (post-synthesis)

- Module 1: (calc_unit) → **33 Registers** are used
- Module 2: (interrupt_controller) → **5 Registers** are used
- Module 3: (top_lite) → **209 Registers** are used
- Module 4: (register_file) → **619 Registers** are used
- Module 5: (transmit_control_top) → **276 Registers** are used
- Module 6: (Receive_control) → **64 Registers** are used
- Module 7: (fifo_and_stream) → **17719 Registers** are used
- Module 8: (stream_interface) → **23186 Registers** are used

4.2.2. Implementation Results

After implementing the design on the board “XCVU440-FLGA2892-1-C” using the VIVADO implementation tool, the utilization results of the resources are obtained as shown in the following figure 4-4.

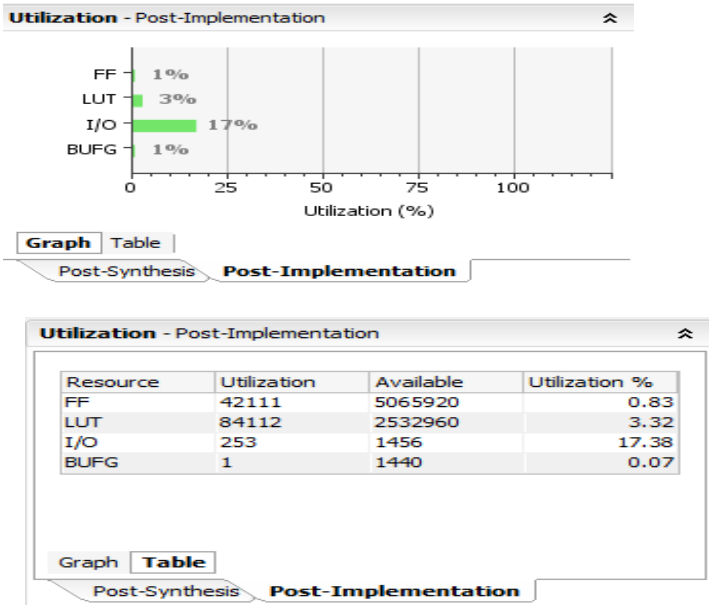


Figure 4-4Total Utilization of board resources (Post-Implementation)

For more details, the following figure shows how each individual module is synthesized and the number of its resources.

Name	CLB LUTs (2532960)	CLB Registers (5065920)	CARRY8 (316620)	F7 Muxes (1266480)	F8 Muxes (633240)	CLB (316620)	LUT as Logic (2532960)	LUT Flip Flop Pairs (2532960)	Bonded IOB (1456)	HPIOB (1404)	GLOBAL CLOCK BUFFERs (
top	84112	42111	177	6449	2431	18008	84112	94960	253	253	
calc_unit1 (calc_unit)	48	33	2	0	0	16	48	49	0	0	
fifo_and_stream1 (fifo_and...	6028	17719	8	2342	1024	4379	6028	13223	0	0	
interrupt_controller1 (interr...	0	5	0	0	0	5	0	5	0	0	
RC (Receive_Control)	144	64	16	0	0	35	144	176	0	0	
register_file1 (register_file)	3262	619	0	0	0	2103	3262	3601	0	0	
stream_I (interface_s)	73546	23186	129	4107	1407	14431	73546	78343	0	0	
TC (Transmit_control_top)	822	276	22	0	0	683	822	972	0	0	
top_LITE (top_a)	249	209	0	0	0	85	249	290	0	0	

Figure 4-5Utilization of board resources for each module (Post-Synthesis)

From both previous sections (4.2.1 and 4.2.2), we can see that the implementation utilization of resources report is almost the same as the synthesis utilization of resources report of but less resources specially LUTs due to optimization option.

Chapter 5.
Verification

5.1. Introduction

After finishing the design, the next step is to verify that the RTL code is doing what it's supposed to do. There are several types of testing such as [6], [7]:

1. **Directed:** It instantiates the DUT (design under test), then applies the inputs. Blocking assignments and delays are used to apply the inputs in the appropriate order, the user must view the results of the simulation and verify by inspection that the correct outputs are produced.
2. **Random:** Machine-generated random inputs, random inputs find cases that designers didn't consider. It's easy to write but may wastes simulation time on undesired cases.
3. **Constrained Random:** Randomized, but targeted, can quickly generate many interesting cases.

In this chapter, we will talk about each method and how we used it in our verification.

5.2. Direct Testing

Direct testing can be done through test benches. Test benches are pieces of code that are used during FPGA or ASIC simulation. Simulation allows you the ability to look at your FPGA or ASIC design and ensure that it does what you expect it to. A test bench provides the stimulus that drives the simulation.

A simple test bench will instantiate the Unit Under Test (UUT) and drive the inputs. One should attempt to create all possible input conditions to check every corner case of the project. A good test bench should be self-checking. A self-checking test bench is one that can generate inputs and automatically compare actual outputs to expected outputs.

5.2.1. Unit Testing

AXI4-Lite Interface

- Normal operation in write transactions
- Normal operation in read transactions

AXI4 Interface

The following test cases are carried out in the three modes of operation (burst types)

- Normal operation in write and read transactions
- Different cases for strobe signal
- Different cases for transaction size
- Different cases for transaction length
- Aligned and unaligned transfers
- Timeout condition error in the slave
- Unsupported transfer size attempt

Register Space

- Check read operations of read access registers upon applying the address of the register and read enable
- Check write operations of write access registers upon applying the address of the register and read enable and input data to be stored.
- Check for response of the register space due to illegal write access: read access registers are not affected by illegal attempt to overwrite on their stored value
- Check for response of the register space due to illegal read access: returns zeros upon illegal attempt to read write only registers
- Check for triggering reset upon writing 0xA5 to the SRR register
- Check for triggering Transmit reset upon writing 0xA5 to the TDFR register
- Check for triggering Receive reset upon writing 0xA5 to the RDFR register

Transmit Control

- Normal operation of block
- Wrong programming sequence
- Packet length calculations for different strobe values
- Transmit size error bit calculations in three scenarios:
    1. No mismatch between calculated packet value and value entered by user
    2. Mismatch in number of words
    3. Mismatch in number of bytes
- TPOE bit is forced high in the middle of normal operation
- Resetting transmit circuitry in the middle of normal operation

Transmit FIFO

- Write operation into FIFO (data FIFO, length FIFO or destination FIFO)
- Read operation from FIFO (data FIFO, length FIFO or destination FIFO)
- Circular operation of FIFO
- Write and read operations at the same time
- Triggering Empty and FULL signals
- Vacancy calculations

Transmit Stream Interface

- Handshake in normal operation
- Throttling data flow by controlling TREADY signal
- Varying TREADY signal in the middle of packet transmission
- Reset transmit circuitry or reset entire core

Receive Stream Interface

- Handshaking: Ensure a proper communication with the Stream side
- Receiving data and destination
- Calculating packet length using the TLAST of the Stream side

Receive FIFO

- Writing in FIFO in both modes (Store and Forward, Cut-Through)
- Reading from FIFO in both modes (Store and Forward, Cut-Through)
- Reading during and at the end of the packet (Partial and Complete Packets)
- Checking empty and full signals (for data FIFO and length FIFO)

Receive Control

- Testing both modes of operation
- Checking the effect of bit 31 in the Receive length FIFO and how it affects type of packet (partial or complete)
- Tracing how the length counter changes in both modes (in complete packet the length counter is initialized with zero while in partial packet initialization with zero takes place only in the first beat of packet and when a processing of new packet starts)
- Testing of wrong programming sequence entry and a reset logic for recovery
- Testing different reading rates (the user can start the reading operation whenever he wants and perform the sequence with no constraints on the number of cycles used)
- Testing scenario for the Receive Packet Overrun Read Error (RPORE)

Calculation Unit

- Increasing vacancy/decreasing vacancy
- Randomly varying vacancy

Interrupt Interface

- Main approach is to randomize the inputs to the interrupt interface to test different scenarios and make sure the correct interrupts are fired depending on the scenario.

5.2.2. Integration Testing

The main goals of the direct testing of the core are to

- Validate the correctness of operation of the core using common scenarios
- Investigate the behavior of core during wrong inputs/sequences
- Probing the design by forcing uncommon scenarios and corner cases

The features supported are to be tested using different test cases and scenarios, the upcoming test cases can be applied on any of the supported features to ensure a full correct functionality:

- Changing data interface: The writing or reading of data from the core can be done using AXI4 or AXI4-Lite
- The width of RDFD is changed in test cases, it can be 32 bits or 64 bits' wide
- Transmit/Receive FIFO depths are changed and tested
- Transmit/Receive Programmable Full and Empty Thresholds are also changed and tested

The following test cases cover the major scenarios in Transmit and Receive paths and are the main ideas for testing the core, and thus each of these general cases can have many other specific cases beyond them.

Transmit Path

The following test cases are carried out for both modes of operation.

Test Case 0

- **Title:** Normal operation of the core

- **Description:** The basic operation of the core. The test can be done by writing packets with different lengths over AXI4-Lite/AXI4 interface and allowing data to be transmitted over AXI4-Stream interface by forcing TREADY signal to be high the entire time of the test. Same test in conducted in two modes of operation (store-and-forward mode and cut-through mode).

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Positive test case

- **What to be tested:** The correct behavior of the core under normal operation in two modes of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data written in TDFD register by user, packet length is identical to packet length written by user in TLR register and packet destination is identical to data written by user in TDR register.

  In Store-And-Forward mode, packet is only transmitted over AXI4 Stream interface when packet length is written by user in TLR register.

  In Cut-Through mode, packet is only transmitted over AXI4 Stream interface when there are at least two transfers in the transmit data FIFO to guarantee that the last beat is transmitted only when length of the packet is written by user in TLR register.

Test Case 1

- **Title:** Transmit Size Error (TSE) bit in ISR register

- **Description:** This test can be conducted by writing a packet length value in TLR register which doesn't correspond to the actual length of data written in TDFD register. TSE bit is tested in three scenarios,

1. No mismatch in actual packet length and value written in TLR register
2. Mismatch in number of bytes
3. Mismatch in number of words

   Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Negative test case

- **What to be tested:** TSE bit value in three scenarios

- **Expected correct output**: TSE bit is expected to be high only in the third scenario. As for data transmitted over AXI4-Stream interface, in the three scenarios data will be identical to data entered by user in TDFD register and packet length is the actual packet length not the value stored in TLR register.

Test Case 2

- **Title:** Transmit packet overrun error (TPOE) bit in ISR register

- **Description:** This test can be done by decreasing depth of transmit data FIFO and write packets into core that exceeds capacity of the transmit FIFO while forcing TREADY signal to be low to guarantee that no data is transmitted over AXI4-Stream interface. Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Negative test case

- **What to be tested:** TPOE bit in ISR register

- **Expected correct output**: TPOE is expected to be high only when the user attempts to write data in a full FIFO. The transmit circuitry will be in a lock state until the user resets either transmit circuitry or entire core

Test Case 3

- **Title:** Rate of data entered into core is more than rate of output data stream
- **Description:** This test can be done by throttling output data stream by forcing TREADY signal to be low at the beginning of the test to allow packets to accumulate into transmit FIFO and then forcing TREADY signal to be high to allow accumulated packets to be transmitted over AXI4-Stream interface.

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.
- **Type of test case:** Positive test case
- **What to be tested:** The correct behavior of the core.
- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data written in TDFD register by user and packet length is identical to packet length written by user in TLR register.

Test Case 4

- **Title:** Obstruct flow of data over AXI4-Stream interface
- **Description:** This test is carried out by forcing TREADY signal to be low in the middle of a packet being transmitted over AXI4-Stream interface

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.
- **Type of test case:** Negative test case
- **What to be tested:** Data flow from AXI4-Stream interface
- **Expected correct output**: Stream interface will stop transmitting data when TREADY is low and will continue to transmit rest of the packet when TREADY is high again.

Test Case 5

- **Title:** Reset transmit circuitry

- **Description:** Resetting transmit circuitry can be done in two scenarios

1. Reset transmit circuitry only by writing specific value in TDFR register
2. Reset entire core by writing specific value in SRR register

   We are interested in the scenario when resetting occurs in the middle of a packet being transmitted over AXI4-Stream interface because the two scenarios mentioned earlier will give different results.
   Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Positive test case.

- **What to be tested:** The correct behavior of the core.

- **Expected correct output**: Resetting transmit circuitry only is not carried out until full packet is transmitted over AXI4-Stream interface, while resetting entire core will be executed immediately and this will result in a partial packet being transmitted over AXI4-Stream interface.

Test Case 6

- **Title:** Different cases for strobe signal.

- **Description:** This test can be done by varying strobe signal of data interface (AXI4-Lite/AXI4).
  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Positive test case.

- **What to be tested:** Data flow from AXI4-Stream interface.

- **Expected correct output**: Transmit FIFO stores only valid data indicated by strobe signal and thus data flow from AXI4-Stream interface is not identical to value written in TDFD register but will be identical to only byte lanes which contains valid data.

Test Case 7

- **Title:** Packet size is not multiple of Stream interface data bus width

- **Description:** This test is done by controlling strobe signal to make the packet length not a multiple of data bus width of stream interface such that the last beat of data transmitted over stream interface will not occupy the full data bus width.

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Positive test case

- **What to be tested:** TKEEP signal of AXI4-Stream interface.

- **Expected correct output**: TKEEP signal should be all ones in all transfers except for the last one; only byte lanes containing valid data will have the corresponding bits in TKEEP signal high.

Test Case 8

- **Title:** Vacancy of transmit FIFO

- **Description:** This test is conducted under the following conditions,

  - Packets written into core have different lengths
  - Strobe signal change from packet to another
  - TREADY signal is not always high so packets will accumulate at part of the test

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of test case:** Positive test case

- **What to be tested:** Transmit data FIFO vacancy (TDFV) register

- **Expected correct output**: TDFV increments by one whenever a beat of data is transmitted over AXI4-Stream interface but decrements by two for each two write locations into transmit FIFO.

Test Case 9

- **Title:** Wrong programming sequence
- **Description:** This test is done by accessing registers TDR, TDFD and TLR in any sequence except for the correct programming sequence. For example, writing data in TDFD register before writing packet destination in TDR register.
- **Type of test case:** Negative test case
- **What to be tested:** Operation of entire core.
- **Expected correct output**: Transmit Circuitry will be in lock state until user resets entire core or transmit circuitry, but if previous packets with correct programming sequence are still stored in transmit FIFO those packets will be allowed to be transmitted over AXI4-Stream interface.

Test Case 10

- **Title:** Circular operation of transmit FIFO
- **Description:** This test is done by decreasing depth of transmit FIFO while entering packets that exceeds capacity of data memory and allowing data to be transmitted over AXI4-Stream interface.
- **Type of test case:** Positive test case
- **What to be tested:** Data flow from AXI4-Stream interface and TPOE bit
- **Expected correct output**: Data transmitted over AXI4-Stream interface is identical to data entered by used in TDFD register. TPOE bit is low the entire test.

Receive Path

Store and Forward Mode

Test Case 0

- **Title:** Normal Operation
- **Description:** This is a basic test case for testing whether the Receive path is doing well under normal conditions, the test can be done by allowing the AXI Stream interface to write some packets with different lengths in the Receive FIFO, and Reading them from the other side of the FIFO block which is the AXI4-Lite. Reading is started as soon as a complete packet appears in the Receive FIFO.
  The handshake in both sides are done in a correct way, also the programming sequence written by the user is correct.
- **Test case classification:** Positive test case
- **What to be tested:** The correct behavior of the Receive logic of the core under normal conditions; i.e. no extremes.
- **Expected behavior:** The written packets from the Stream interface will be transferred successfully into the AXI4-Lite side appearing on the port AXI4_lite_RDATA, also the same port will carry the packet information such as the packet length and packet destination in the same order as the order of the programming sequence

Test Case 1

- **Title:** Wrong sequence that affects the operation
- **Description:** This test can be done by writing wrong major steps in the programming sequence from the AXI4-Lite side which affects the core operation of the Receive logic, such as trying to read the data in RDFD register before reading its length in RLR.
  The handshake is correct in both sides, but the order of input data through the read address port (AXI4_lite_ARADDR) is not correct.
- **Test case classification:** Negative test case
- **What to be tested:** The effect of a strongly wrong programming sequence on the Receive path operation and how to recover from it.
- **Expected behavior:** The starting of sequence of receiving a packet will trigger the Receive control block to be active, and the data out on the Read data port AXI4_lite_RDATA will be correct until the wrong step in the sequence, then it’s expected that the Receive logic will be at stuck state, and will need a reset signal to recover from this state and be ready again to transfer packets.

Test Case 2

- **Title:** Full Receive FIFO
- **Description:** This test can be done by performing a continuous write into the Receive FIFO from the Stream interface side, and not reading from the AXI4-Lite side, and thus the packets will continue to be written in the data FIFO until it becomes full, this is an extreme case at which the user may be busy doing something else and thus not available to read from the Receive FIFO.
  The handshake in both sides are done in a correct way, the programming sequence here is not the point of interest but if it's written, it shall be correct sequence.

- **Test case classification:** Destructive/Capacity test case

- **What to be tested:** The behavior of the core in one of its abnormal conditions which is fully utilizing the Receive FIFO, also the behavior of the Stream interface inside the core at this case to prevent any packet corruption.

- **Expected behavior:** The FIFO is filled with Packets, location by location, and when it reaches the Receive FIFO Programmable Full Threshold (indicating that data FIFO is full) the Receive FIFO should stop receiving packets from the Stream side by stopping the handshake (TREADY=0), also the Receive FIFO Programmable Full (RFPF) which is bit 20 in the ISR must be set to 1 to inform the user that the Receive FIFO is now full and that they can't receive any other packets from Stream side until a Reading operation is performed.

Test Case 3

- **Title:** Circular operation of Receive FIFO
- **Description:** This test can be done by performing a continuous write into the Receive FIFO from the Stream interface side, but with also a continuous reading from the AXI4-Lite side, reading rate doesn't need to be as writing rate but it should be continuous so that when the end of Receive data FIFO is reached, it won't be full as reading operation has been performed and thus circular movement for write and read pointers can be tested, providing more space for user to utilize in the Receive data FIFO. The handshake in both sides are done in a correct way, the programming sequence is written in a correct way.

- **Test case classification:** Positive test case

- **What to be tested:** The operation of the Receive FIFO, and how it handles large number of packets or data, while also watching the RFPF behavior.

- **Expected behavior:** The written packets from the Stream interface will be transferred successfully into the AXI4-Lite side appearing on the port AXI4_lite_RDATA, also the same signal will carry the packet information such as the packet length and packet destination in the same order as the order of the programming sequence, and this will continue to happen even though the end of Receive data FIFO is reached, also the RFPF shall not be set to 1 as the Receive FIFO is not considered to be Full.

Test Case 4

- **Title:** Receive Packet Overrun Read Error (RPORE)
- **Description:** This test can be performed by writing some packets in the Receive FIFO coming through the Stream interface and start reading them using a partially correct programming sequence, to be able to reach the RPORE scenario.
  The programming sequence will be correct at the first three steps: reading and writing in the ISR, Reading RLR, Reading RDR, then finally in the fourth step reading RDFD will take place but with a number of times that is larger than the number specified in the RLR.

- **Test case classification:** Negative test case

- **What to be tested:** The ability of the core of updating the ISR with the interrupts that indicate that an error has occurred, in our case it's the RPORE. Also how the Receive logic will recover from these errors.

- **Expected behavior:** As this is also considered to be a wrong sequence, it's expected that the receive control will be stuck until a reset signal is sent to it.
  But our point of interest here is watching the ISR, as it's expected to find the RPORE (bit 30) is set to 1, indicating that this error scenario has occurred.

Test Case 5

- **Title:** Writing only one packet in the Receive FIFO
- **Description:** This is case is just as the normal operation specified in case 1 but with only one packet to be written in the Receive FIFO. This case is considered to be a corner case as when performing a correct programming sequence, reading the RLR will cause the Receive length FIFO to be empty, and if not handled well, this may indicate an empty FIFO although there're still data bytes in the Receive data FIFO.

- **Test case classification**: Positive test case

- **What to be tested:** The internal signals that travel from individual blocks into the ISR in the interrupt interface which is the Receive FIFO Programmable Empty (RFPE).

- **Expected behavior:** Exactly as normal operation case, the written packets from the Stream interface will be transferred successfully into the AXI4-Lite side appearing on the port AXI4_lite_RDATA, also the same port will carry the packet information such as the packet length and packet destination in the same order as the order of the programming sequence.
  Also the expected correct behavior for the RFPE is to be set to zero during reading the packet data then is set to 1 when the entire data in packet is read (or until reaching the Receive FIFO Programmable Empty).

Cut-Through Mode

Test Case 6

- **Title:** Normal Operation – Writing and Reading in parallel
- **Description:** This is a basic test case for testing whether the Receive path is doing well under normal conditions in the Cut-Through mode. The test can be done by allowing the AXI Stream interface to write some packets with different lengths in the Receive FIFO, and Reading them from the other side of the FIFO block which is the AXI4-Lite. Reading must start as soon as writing of first packet in the Receive FIFO starts, to allow reading of a partial packet not a complete one.
  The handshakes in both sides are done in a correct way, also the programming sequence written by the user is correct.

- **Test case classification:** Positive test case

- **What to be tested:** The correct behavior of the Receive logic of the core under normal conditions in the cut-through mode, and how to deal with Partial Packets.

- **Expected behavior:** The written packets from the Stream interface will be transferred successfully into the AXI4-Lite side appearing on the port AXI4_lite_RDATA as soon as reading operation start without waiting for a packet to be totally received in the Receive data FIFO, also the same signal will carry the packet information such as the packet length in the same order as the order of the programming sequence, and this will continue until the last beat of the packet where it becomes a complete packet and sequence is changed according to this.

Test Case 7

- **Title:** Normal Operation – Writing then Reading
- **Description:** This is an another basic test of the Cut-through mode, this is a test of generality of this mode, as illustrated before in Chapter 3 the cut-through mode is more general than the store and forward mode, as if the reading operation is delayed until a complete packet is written, then the cut through mode will behave exactly as same as the store and forward mode. In this test the Stream interface is allowed to write into the Receive FIFO but the Reading operation is delayed to allow a complete packet formation.

- **Test case classification:** Positive test case

- **What to be tested:** Operation of the Cut-through mode under conditions similar to the store and forward mode.

- **Expected behavior:** It should be exactly behaving like the store and forward mode.

Test Case 8

- **Title:** Wrong Programming Sequence
- **Description:** This case is similar to case 7 with same operation that needs writing in the Receive FIFO and reading from it at the same time to allow a partial packet operation, the only difference is that the input programming sequence is wrong. Reading sequence of a partial packet is different from reading a complete one.

- **Test case classification:** Negative test case

- **What to be tested:** The behavior of the Receive logic of the core when the input programming sequence is wrong in the cut-through mode, and how to deal with Partial Packets.

- **Expected behavior:** The starting of sequence of receiving a packet will trigger the Receive control block to be active, and the data out on the Read data port AXI4_lite_RDATA will be correct until the wrong step in the sequence, then it's expected that the Receive logic will be at stuck state, and will need a reset signal to recover from this state and be ready again to transfer packets.

Test Case 9

- **Title:** Writing a very long packet
- **Description:** This test can be performed by writing a packet with length of the Receive data FIFO, and reading it while writing to check for the Cut-through mode correct operation.

- **Test case classification:** Destructive/Capacity test case

- **What to be tested:** The performance and capacity of system under large inputs and extreme conditions.

- **Expected behavior:** The written beats from the Stream interface will be transferred successfully into the AXI4-Lite side as soon as reading operation start appearing on the port AXI4_lite_RDATA, also the same port will carry each beat information such as the beat length and packet Destination in the same order as the order of the programming sequence, and this will continue until the last beat of the packet where it becomes a complete packet and sequence is changed according to this performing reading of a complete packet.

5.3. Universal Verification Methodology (UVM)

5.3.1. Introduction

When we deal with small designs or small blocks of larger designs, we can create a direct test bench that will produce correct simulations results, exercise all behaviors of the design, but in case of large design with complex functionality, most designer may miss some design behaviors while creating a direct test bench as discussed in [5], [8], [9] and [10].

So, for a proper functionality testing, we want to

1. Test all important design behaviors
2. Be sure that design behaves as intended
3. And direct test bench is not enough to do that as we need better coverage support and to check that whatever is planned and whatever is designed are the same. And that's why we go to a more effective verification method which is the Universal Verification Methodology (UVM).

UVM is a methodology for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. UVM was created by Accellera based on the OVM (Open Verification Methodology) version 2.1.1. The roots of these methodologies lie in the application of the languages IEEE 1800™ SystemVerilog, IEEE 1666™ SystemC, and IEEE 1647™ e.

UVM is a methodology for writing test benches, but it's used in constrained random verification.

We can summarize the objective of UVM in three word:

- **Check:** if you've got random vectors, you can automate checking, by write self-checking test benches in systemVerilog to answer the question "Does design work?" automatically.
- **Coverage:** functional coverage, record what goes on during a verification environment, identify how thoroughly we've exercised the design. If you have coverage holes, then you haven't exercised the design thoroughly enough.
- **Constraints:** specific constraints on the values of the inputs to increase test coverage.

UVM has several advantages and characteristics, some of them are as follows:

1. **Support constrained random verification:** the idea is to randomize at least some elements of the test vectors which is useful for finding unexpected bugs, and this way we push the design into state which we may have never thought of.

2. **Automating stimulus generation:** you can run long simulation without manually writing all the test vectors yourself, as they can be generated automatically.

3. **Base class library:** UVM has built in methods like compare, copy, print, also we have reporting mechanism; macros. Macros are not function calls, they are textual replacements.

4. **Configuration class:** config class is a database where we configure all the parameters needed throughout the hierarchy using set and get.

5. **Modularity and Reusability:** The methodology is designed as modular components (Driver, Sequencer, Agents, Environment, etc.) which enable reusing components across unit level to multi-unit or chip level verification as well as across projects.

6. **Phasing:** since all the components are derived from uvm_component , phasing helps in synchronization of each and every component before proceeding to next phase.

7. **Simulator independent**: The base class library and the methodology are supported by all simulators and hence there is no dependency on any specific simulator.

8. **Sequence methodology**: It gives good control on stimulus generation. There are several ways in which sequences can be developed which includes randomization, layered sequences, virtual sequences, etc. which provides a good control and rich stimulus generation capability.

9. **Factory:** factory is the class that manufactures components and objects which gives the ability to modify and number of objects that makes test bench hierarchy in more predictable manner. Overriding of components or objects becomes much more easy using factory concept using create() method instead of new() method.

10. **Virtual sequencer and sequences**: For keeping the independency between test bench writer and test case writer as test case writer doesn't have to worry about the path in order to start a sequence.

5.3.2. Overview

The **Universal Verification Methodology** (UVM) is a standardized methodology for verifying integrated circuit designs as mentioned before.

UVM General Architecture

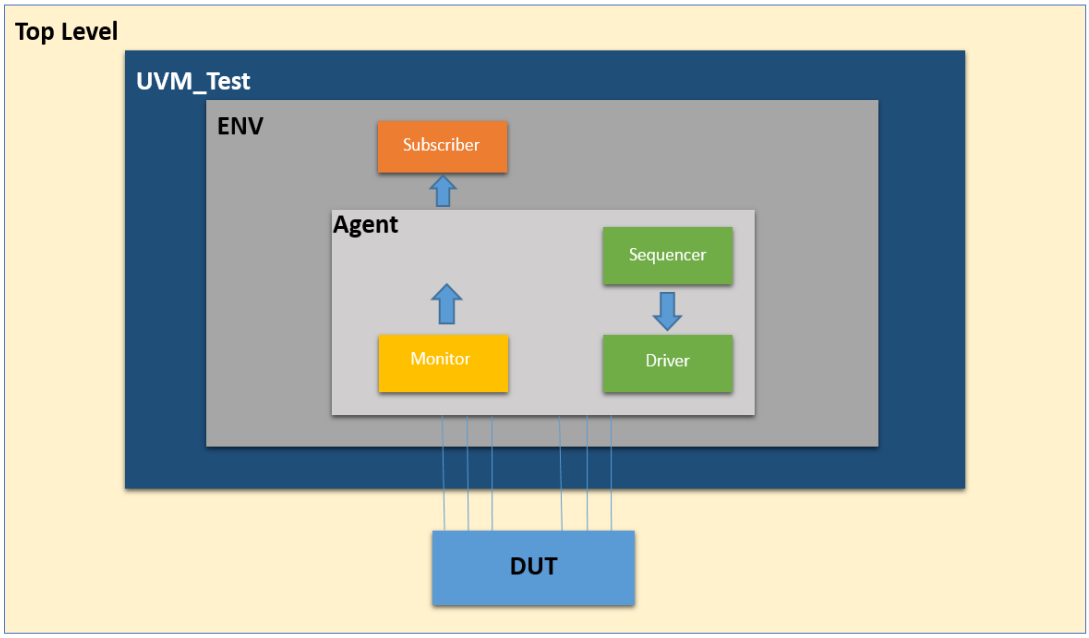


Figure 5-1UVM Architecture

1. UVM Test bench

The UVM Test bench typically instantiates the Design under Test (DUT) module and the UVM Test class, and configures the connections between them. If the verification includes module-based components, they are instantiated under the UVM Test bench as well. The UVM Test is dynamically instantiated at run-time, allowing the UVM Test bench to be compiled once and run with many different tests.

2. UVM Test

The UVM Test is the top-level UVM Component in the UVM Test bench. The UVM Test typically performs three main functions: Instantiates the top-level environment, configures the environment, and applies stimulus by invoking UVM Sequences through the environment to the DUT.

3. UVM Environment

The UVM Environment is a hierarchical component that groups together other verification components that are interrelated. Typical components that are usually instantiated inside the UVM Environment are UVM Agents, UVM Scoreboards, or even other UVM Environments. The top-level UVM Environment encapsulates all the verification components targeting the DUT.

4. UVM Scoreboard / Subscriber

The main function is to check the behavior of a certain DUT. Usually receives transactions carrying inputs and outputs of the DUT through UVM Agent analysis ports. Run the input transactions through some kind of a reference model to produce expected output, and then compares the expected output versus the actual output.

5. UVM Agent

The UVM Agent is a hierarchical component that groups together other verification components that are dealing with a specific DUT interface. A typical UVM Agent includes a UVM Sequencer to manage stimulus flow, a UVM Driver to apply stimulus on the DUT interface, and a UVM Monitor to monitor the DUT interface. UVM Agents might include other components, like coverage collectors, protocol checkers, a Transaction-Level Modeling (TLM) model, etc.

The UVM Agent needs to operate both in an active mode (where it is capable of generating stimulus) and a passive mode (where it only monitors the interface without controlling it).

6. UVM Sequencer

The UVM Sequencer controls the flow of UVM Sequence Items transactions generated by one or more UVM Sequences.

7. UVM Sequence

A UVM Sequence is an object that contains a behavior for generating stimulus. UVM Sequences are not part of the component hierarchy; each UVM Sequence is eventually bound to a UVM Sequencer. Multiple UVM Sequence instances can be bound to the same UVM Sequencer.

8. UVM Driver

UVM Driver receives individual UVM Sequence Item transactions from the UVM Sequencer and applies (drives) it on the DUT Interface. Thus, a UVM Driver spans abstraction levels by converting transaction-level stimulus into pin-level stimulus. It also has a TLM port to receive transactions from the Sequencer and access to the DUT interface in order to drive the signals.

9. UVM Monitor

Monitoring pin level activity, assemble transaction object from DUT and pass them to the verification environment for further analysis, it spans abstraction levels by converting pin-level activity to transactions. In order to achieve that, the UVM Monitor typically has access to the DUT interface and also has a TLM analysis port to broadcast the created transactions through.

UVM Functionality

The main Functionality of UVM is to generate constrained random verification

- Constrained verification to simulate the needs of the input variables of the DUT (design under test)
- Random to test (DUT) in a large critical and different test cases to discover unexpected bugs.

Now if we have a Design and want to test it using UVM, the following steps are needed:

1. Set up the UVM components properly, including the Test, Environment, Subscriber, Agent, Monitor, Driver and the Sequencer.
2. Connect both the Monitor and the Driver to the interface connecting the DUT to the UVM Test.
3. We need to write a sequence, sequences mainly describe a testing operation with random value, each sequence need sequencer to control the flow of UVM Sequence Items transactions, each UVM Sequence is eventually bound to a UVM Sequencer. Multiple UVM Sequence instances can be bound to the same UVM Sequencer.
4. Driver is driving stimulus to the DUT through an interface "Driver and sequencer shapes the downstream path to the DUT"

Now a constrained random sequence is the input to the DUT (design under test) and also input to the Scoreboard/Subscriber, Scoreboard/Subscriber are responsible for simulate the main Functionality needed to be tested. To use it in applying Automating check, the simplest way to do this is to compare the output from DUT and the out from Scoreboard / Subscriber.

Difference between Scoreboard and Subscriber is that Scoreboard a real model which can be a Matlab code, C++ code, etc. but Subscriber is a systemVerilog design that simulates the behavior of the DUT. To apply compare and check between output from DUT and output from Scoreboard/Subscriber we need monitor. The monitor recognizes pin level activity and assemble transaction object from DUT and pass them to the verification environment for further analysis. Using analysis port, sending transaction up to Subscriber can take place (Subscriber has analysis export) and monitor can perform internally some processing on the transactions produced such as coverage collection, checking, recording, etc.

5.3.3. UVM Sequences

Transmit Path

Sequence 1

- **Title:** Normal operation in Store-And-Forward Mode and Cut-Through mode
- **Description:** The basic operation of the core. The Sequence writes random number of packets with random length for each packet to the Transmit FIFO through the AXI4-Lite interface, and then after all the packets are written to the core, the written packets/they are read from the core through AXI4-Stream interface.
  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence
- **Constraints:**
  - Length of packets is constrained to be from 1 to 20 words
  - Number of packets is constrained to be from 1 to 20 packets
  - TREADY is always high
  - Full bus width contains valid data (Strobe signal constrained to the decimal value 15)

- **What to be tested:** The correct behavior of the core under normal operation the Store-And-Forward mode of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data written by the user over the AXI4-Lite interface, packet length read from the core is identical to the packet length written by the user over the AXI4-Lite interface.

Sequence 2

- **Title:** Throttling packet transmission path over AXI4-Stream interface in the two modes of operation (Store-and-Forward and Cut-Through mode)
- **Description:** The Sequence writes random number of packets with random length for each packet to the Transmit FIFO through the AXI4-Lite interface, and then after all the packets are written to the core, the written packets/they are read from the core through AXI4-Stream interface but with TREADY signal randomized each data transfer which means Stream interface is not always permitted to transmit data.
  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence
- **Constraints:**
  - Length of packets is constrained to be from 1 to 20 words
  - Number of packets is constrained to be from 1 to 20 packets
  - Full bus width contains valid data (Strobe signal constrained to the decimal value 15)

- **What to be tested:** Data flow from AXI4-Stream interface

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data written by the user over the AXI4-Lite interface, however stream of data is not continuous due to the randomization of TREADY, it's expected that when TREADY is low flow of data will strop and when TREADY is high again packet transmission will continue. Packet length read from the core is identical to the packet length written by the user over the AXI4-Lite interface.

Sequence 3

- **Title:** Resting transmit circuitry at random instants
- **Description:** Packets are written into core over AXI4-Lite interface while keeping **tready** signal random. Transmit circuitry is reset at random instants while keeping normal operation of the core.

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence
- **Constraints:**
  - Length of packets is constrained to be from 1 to 20 words
  - Number of packets is constrained to be from 1 to 20 packets
  - Full bus width contains valid data (Strobe signal constrained to the decimal value 15)

- **What to be tested:** Data flow from AXI4-Stream interface and reset operation

- **Expected correct output**: When user requires to reset transmit circuitry while a packet is being transmitted over AXI4 stream interface; reset operation is not performed until packet is fully transmitted over interface, even if tready signal becomes low; the reset operation will be performed after the last beat over AXI4-Stream interface. If reset operation is requested at any other scenario the reset operation is performed immediately.

Sequence 4

- **Title:** Allow packets to accumulate in the transmit data FIFO
- **Description:** A random number of packets is written into core while keeping **tready** signal low to let packets accumulates in transmit data FIFO, then **tready** signal is forced high while keeping normal operation of core by continuously writing packets into transmit data FIFO.

  Correct handshake over AXI4-Lite/AXI4 interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence
- **Constraints:**
  - Length of packets is constrained to be from 1 to 20 words
  - Number of packets is constrained to be from 1 to 20 packets
  - Full bus width contains valid data (Strobe signal constrained to the decimal value 15)

- **What to be tested:** Data flow from AXI4-Stream interface

- **Expected correct output**: Continuous stream of data transmitted over AXI4-Stream interface is identical to data entered by user into TDFD register.

Receive Path

Sequence 1

- **Title:** Store-And-Forward Mode normal operation

- **Description:** The basic operation of the core. The Sequence writes random number of packets with random length for each packet to the Receive FIFO through the AXI4-Stream interface, and then after all the packets are written to the core, the written packets/they are read from the core through AXI4-Lite interface.

  Correct handshake over AXI4-Lite interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence

- **Constraints:** Length of packets is constrained to be from 1 to 20 words
  Number of packets is constrained to be from 1 to 20 packets

- **What to be tested:** The correct behavior of the core under normal operation the Store-And-Forward mode of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data read by the user over the AXI4-Lite interface, packet length read from the core is identical to the length of the packet written through the AXI4-Stream interface

Sequence 2

- **Title:** Store-And-Forward Mode normal operation - 2
- **Description:** The basic operation of the core. The Sequence writes random number of packets with random length for each packet to the Receive FIFO through the AXI4-Stream interface, after the first packet is written to the core each subsequent write of a new packet to the core through the AXI4-Stream interface is accompanied by a read to the previous packet written in the core at the same time

  Correct handshake over AXI4-Lite interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence

- **Constraints:** Length of packets is constrained to be from 1 to 20 words
  Number of packets is constrained to be from 1 to 20 packets

- **What to be tested:** The correct behavior of the core under normal operation the Store-And-Forward mode of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data read by the user over the AXI4-Lite interface, packet length read from the core is identical to the length of the packet written through the AXI4-Stream interface

Sequence 3

- **Title:** Cut-Through Mode Normal operation – One Packet
- **Description:** The basic operation of the core. The Sequence writes one packet through the entire sequence over the AXI4-Stream interface, however the first half of the packet is written and then the second half of the packet is written while the first half is read at the same time over the AXI4 – Lite interface, and then the second half is read time over the AXI4 – Lite interface.

  Correct handshake over AXI4-Lite interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence

- **Constraints:** Length of packet is constrained to be 20 words

- **What to be tested:** The correct behavior of the core under normal operation the Store-And-Forward mode of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data read by the user over the AXI4-Lite interface, packet length read from the core is identical to the length of the packet written through the AXI4-Stream interface

Sequence 4

- **Title:** Cut-Through Mode Normal operation – Random Number Of Packets
- **Description:** The same sequence as sequence 3, however instead of writing one packet only the sequence writes a random number of packets

  Correct handshake over AXI4-Lite interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence

- **Constraints:** Length of packet is constrained to be 20 words
  Number of packets is constrained to be from 1 to 20 packets

- **What to be tested:** The correct behavior of the core under normal operation the Store-And-Forward mode of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data read by the user over the AXI4-Lite interface, packet length read from the core is identical to the length of the packet written through the AXI4-Stream interface

Sequence 5

- **Title:** Cut-Through Mode Normal operation – Random Number Of Packet
- **Description:** The sequence goes as follows
  1) The first half of the first is packet is written over the AXI4 – Stream interface
  2) The first half of the packet is read over the AXI4 - Lite while the second half is being written at the same time
  3) The second half of the packet is read while the first half of the next packet is being written at the same time

  Steps 2 and 3 are repeated for a random number of packets

  Correct handshake over AXI4-Lite interface and AXI4-Stream interface and correct programming sequence are guaranteed.

- **Type of sequence:** Positive sequence

- **Constraints:** Length of packet is constrained to be 20 words
  Number of packets is constrained to be from 1 to 20 packets

- **What to be tested:** The correct behavior of the core under normal operation the Store-And-Forward mode of operation

- **Expected correct output**: Data transmitted over AXI4 Stream interface is identical to data read by the user over the AXI4-Lite interface, packet length read from the core is identical to the length of the packet written through the AXI4-Stream interface

Chapter 6.
Simulation Results

6.1. Transmit Path

Simulation Result 1:

Description: Correct programming sequence is performed in Store-and-Forward mode to transmit a packet of length 10 bytes as illustrated in figure 6-1, in the first two transfers full data bus width contains valid data, while the third transfer only the first two bytes contain valid data.

When packet length is written data is transmitted over AXI4-Stream interface as illustrated in figure 6-2, the first two transfer **tkeep** signal indicates that all data bus width contain valid data while the third transfer **tkeep** signal indicates that only the first two bytes contain valid data.

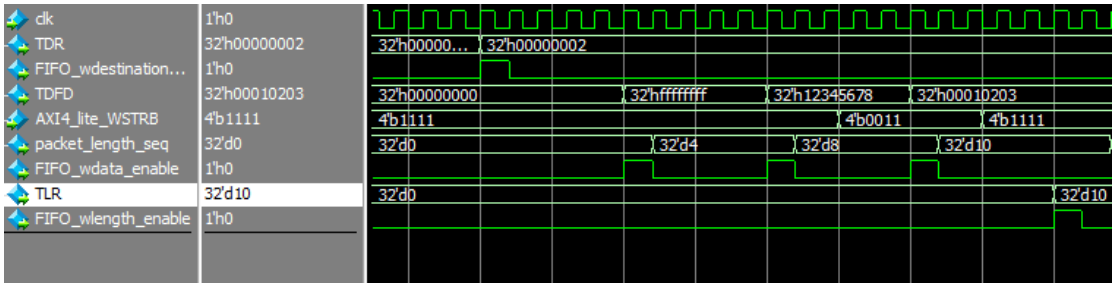


Figure 6-1 Register Space and Control Unit simulation results1

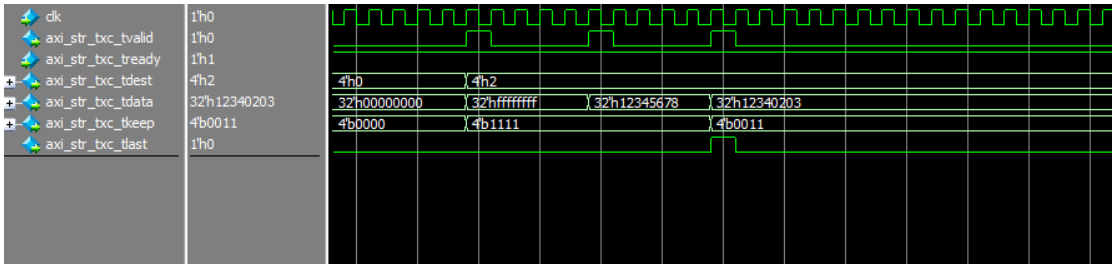


Figure 6-2 Stream Interface simulation results1

Simulation Result 2:

Description: Correct programming sequence is performed in Store-and-Forward mode to transmit a packet of length 10 bytes as illustrated in figure 6-3.

When packet length is written data is transmitted over AXI4-Stream interface as shown in figure 6-4, while the packet is being transmitted the **reset** signal is low to indicate that the user required to reset entire core, the core immediately performs the reset operation which means that the core on the other side of the stream interface will receive a partial packet.

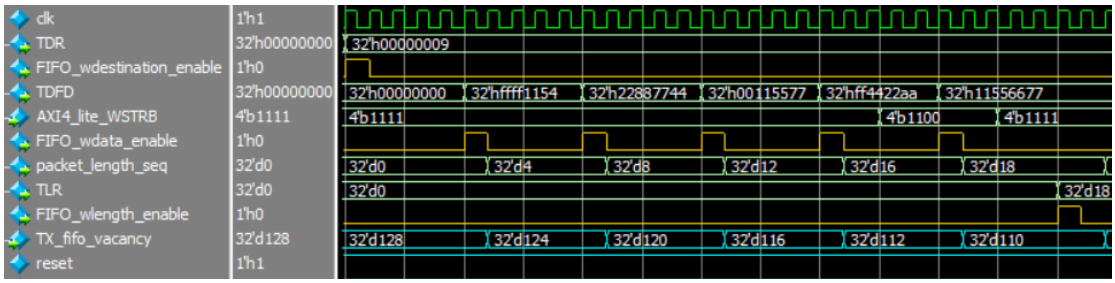


Figure 6-3 Register Space and Control Unit simulation results 2

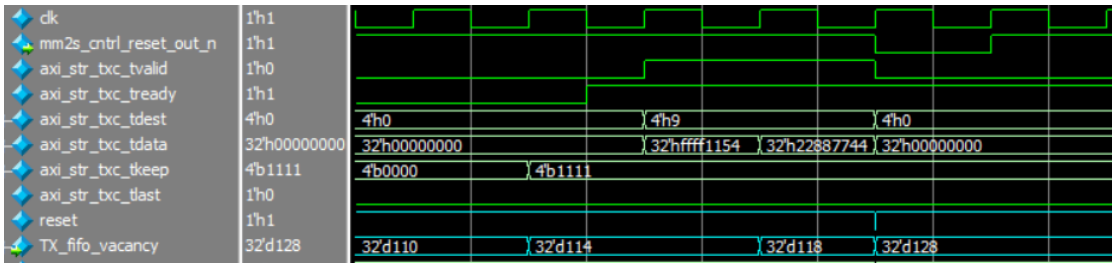


Figure 6-4 Stream Interface simulation results 2

Simulation Result 3:

Description: This case is very similar to previous one; the only difference is that **transmit_reset** signal is low to indicate that the user required to reset transmit circuitry, the core doesn't perform the reset operation until the packet is fully transmitted to prevent the core on the other side of the stream interface from receiving a partial packet. Waveform at figures 6-5 and 6-6 illustrated this scenario.

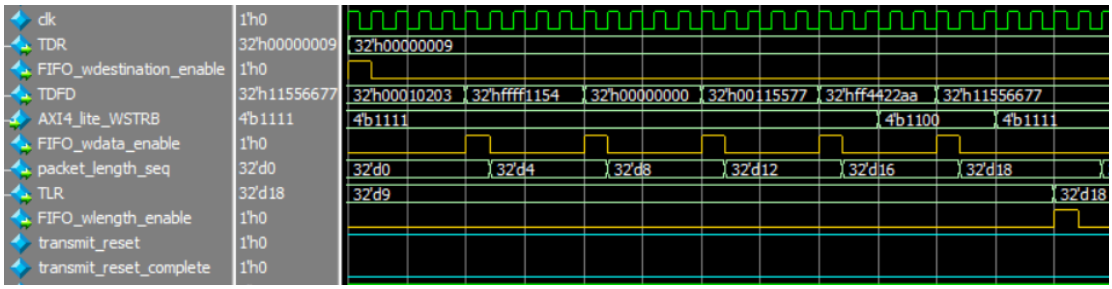


Figure 6-5 Register Space and Control Unit simulation results 3

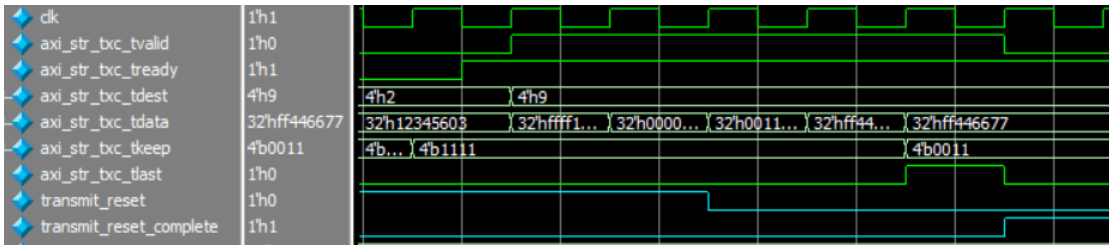


Figure 6-6 Stream Interface simulation results 3

Simulation Result 4:

Description: Correct programming sequence is performed in Cut-Through mode to transmit a packet of length 18 bytes as illustrated in the figure 6-7, When there is more than 4 bytes (width of stream data bus) stored in the data FIDO; data is transmitted over AXI4-Stream interface.

While the packet is being transmitted **tready** becomes low and stream interface pause transmission until **tready** is high.

The **reset** signal is low to indicate that the user required to reset entire core, the core immediately performs the reset operation which means that the core on the other side of the stream interface will receive a partial packet.

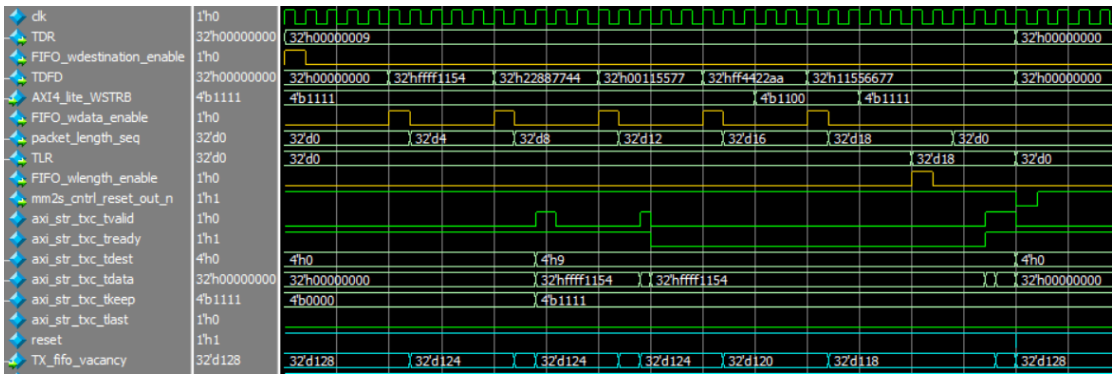


Figure 6-7 Transmit path simulation result 4

Simulation Result 5:

Description: This case is very similar to previous test case; the only difference is that user required to reset transmit circuitry; at this scenario reset operation is not performed until the packet is fully transmitted over AXI4-Stream interface. Waveform in figure 6-8 illustrates this scenario.

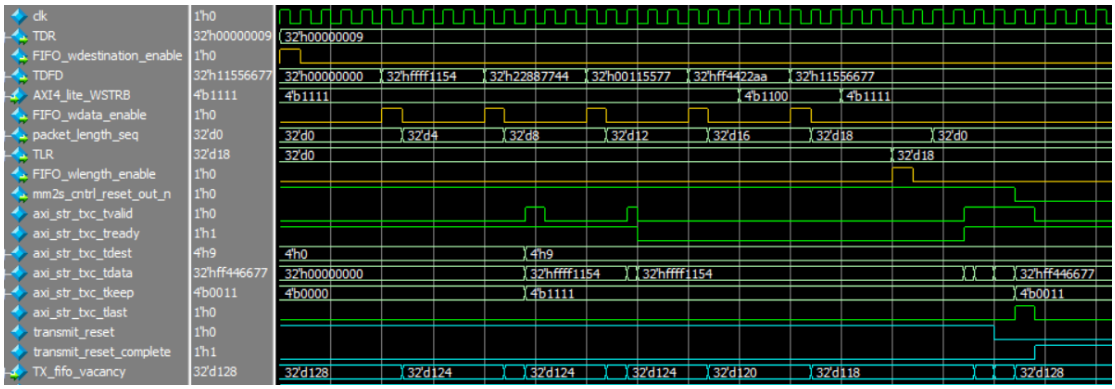


Figure 6-8 Transmit Path simulation result 5

6.2.Receive Path

Simulation Results 1:

Description: Store-And-Forward normal operation where random number of packets are written to the core via the AXI4-Stream interface and then the same number of packets is read from the core through the AXI4 – Lite interface, as shown in Figure 6-9 the first word of the packet written through the AXI4-Stream interface is 0x12249216 which is the same word read from the AXI4 – Lite interface when reading from the RDFD register of address 0x20 as shown in Figure 6-10, in Figure 6-11 the blue signal is bit 26 in the ISR is the receive complete interrupt which goes high after the end of writing of the first packet which is marked by the **TLast** signal with golden color.

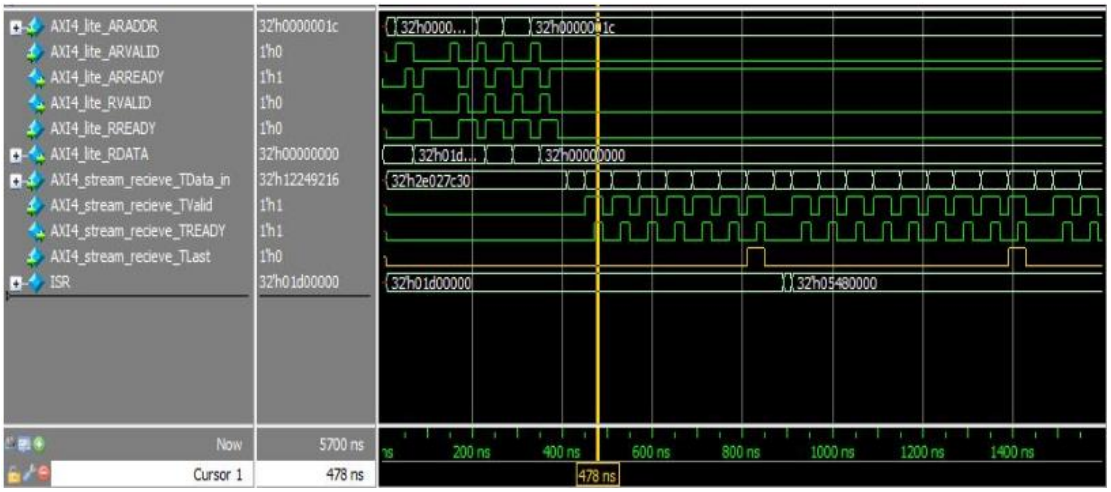


Figure 6-9 Writing and reading random packets

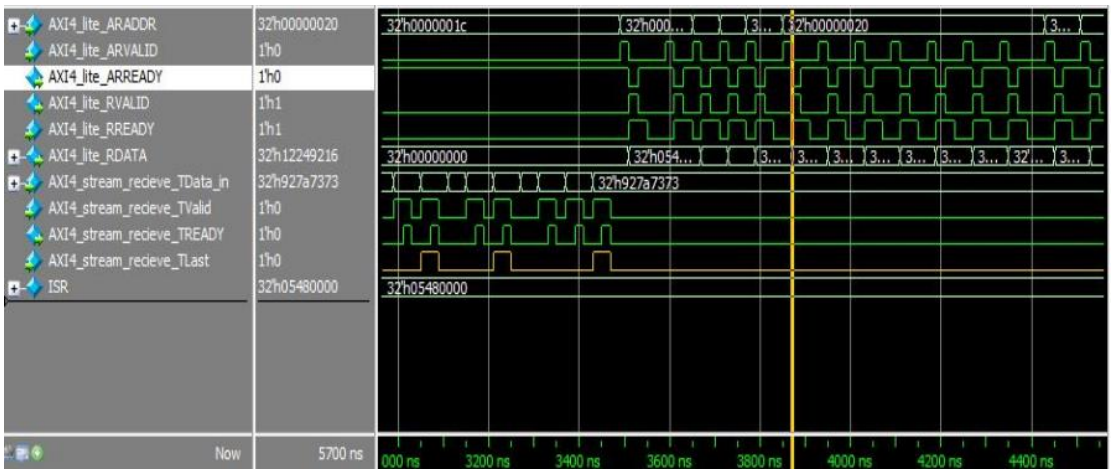


Figure 6-10 First word written to the core

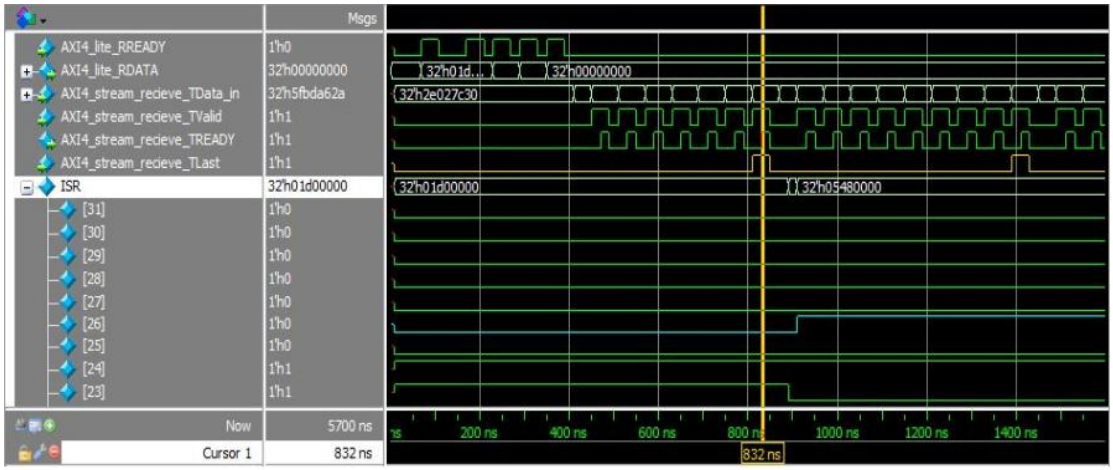


Figure 6-11 RC bit 26 goes high after TLAST

Simulation Result 2:

Description: Cut-Through mode operation where half of the first packet is written to the core via the AXI4-Stream and then the second half is written while the first half is being read via the AXI4 – Lite at the same time, as shown in Figure 1 the first word of the packet written through the AXI4-Stream interface is 0x8e33dc49 which is the same word read from the AXI4 – Lite interface when reading from the RDFD register of address 0x20 as shown in Figure 2,in Figure 3 the blue signal is bit 26 in the ISR is the receive complete interrupt which goes high after the end of writing of the first packet which is marked by the TLast signal with golden color

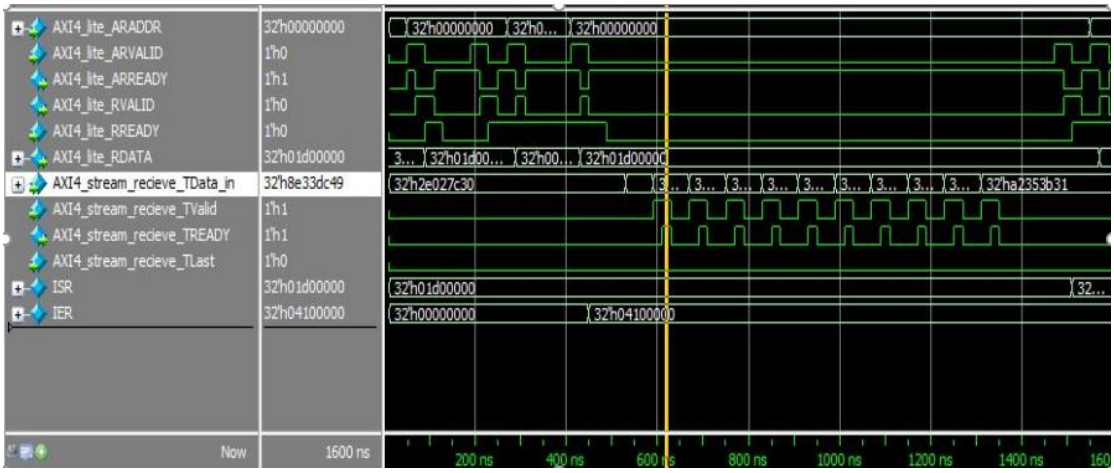


Figure 6-12 First word written to the core

Figure 6-13 First word read from the core

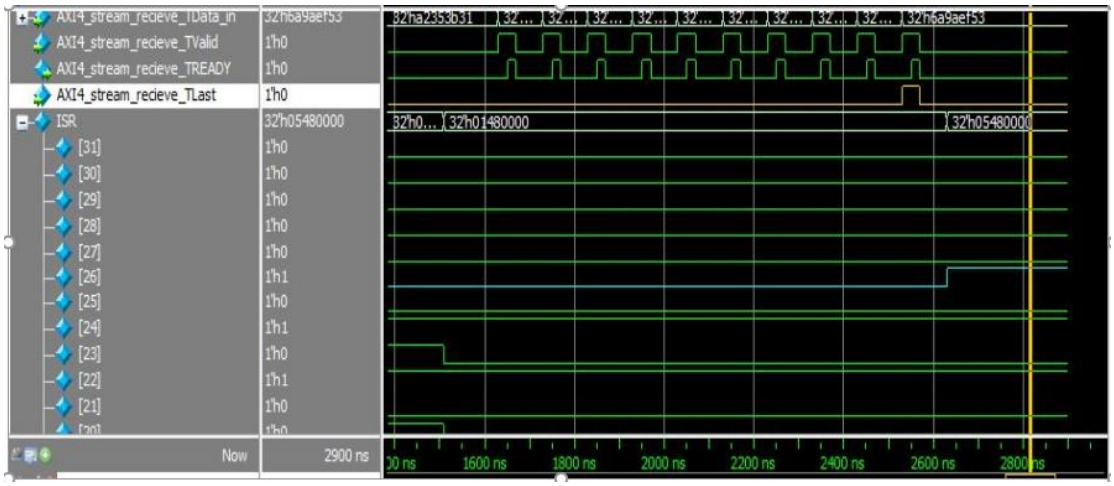


Figure 6-14 RC bit 26 goes high after TLAST

Chapter 7.
Conclusion and Future Work

7.1. Conclusion

In this thesis we demonstrate the design and verification of VIVADO AXI4 Stream FIFO core, introducing the design and verification techniques that can be used,

The proposed architecture was discussed in details. The designed VIVADO AXI4 stream FIFO core has two independent Transmit and Receive paths with two independent FIFOs with configurable width and depth. The core also has two data interfaces: AXI4 and AXI4-Lite. It supports two modes of operation that can be used in transmission or reception: Store-And-Forward and Cut-Through. And also has thirteen mask able interrupts to show the status of core and fault conditions.

Regarding the verification, The Direct testing was divided into two types of testing: Unit testing for each individual block in the design to test its different features and functionalities, and Integration Testing which led to more than twenty different test cases to explore the main features of the core using positive and negative scenarios to test the response of the core to resulting faults and check interrupts proper operation.

The Constrained Random Testing using Universal Verification Methodology (UVM) is also used to test the core using various sequences to explore different scenarios of normal mode of operation in the Cut-Through and Store-And-Forward modes and the AXI4 and AXI4 – Lite data interfaces.

In closing, The VIVADO AXI4-Stream FIFO core can be used to allow memory mapped access to AXI4-Stream Interface such as AXI Ethernet core without having to use a full Direct Memory Access (DMA) solution which is more complicated.

7.2. Future Work

Future work concerns deeper analysis of particular mechanisms, new proposals to try different methods, or simply curiosity. This thesis has been mainly focused on design and verification of VIVADO AXI4-Stream FIFO as we should give enough attention to (1) Completion of UVM testing, (2) Verification using equivalence checking, and (3) Verification using Questa Verification IP.

7.2.1. Completion of UVM testing

As The Constrained Random Testing using Universal Verification Methodology (UVM) is used to test the core, more sequences should be added to explore different scenarios to test for bugs and issues in design.

7.2.2. Verification using equivalence checking

Formal equivalence checking process is a part of electronic design automation (EDA) to formally prove that two representations of a circuit design exhibit exactly the same behavior, as it turns functional verification is more efficient, in terms of time (verification speed) and comprehensiveness (state space coverage). Simulation-based techniques are implemented with relatively low effort, and are good at quickly finding easy-to-spot bugs.

In future work, it is supposed to verify our design using equivalence checking versus Xilinx generated FIFO stream simulation module.

7.2.3. Verification using Questa Verification IP

In our thesis, The Verification has been tested using Questa Sim which has limited capabilities but Questa Verification IP integrates seamlessly into all advanced verification environments on all industry-standard simulators. With a consistent and easy-to-use UVM architecture across all protocols, Questa Verification IP ensures maximum productivity and flexibility for the verification of block level, subsystem, and SoC designs. Questa Verification IP includes AMBA® Questa Verification IP (VIP) family enables fast and accurate verification for designs that use any AMBA 3, AMBA 4 or AMBA 5 protocols.

Chapter 8.
References

---

[1]. K. Swetha & G. Ramakrishna. (2014). AMBA protocols for ALU.

[2]. AXI Reference guide. (2011).

[3]. AMBA AXI-Stream protocol specifications version 1.0. (2010).

[4]. MIT Course for Complex Digital Systems spring 2005 lecture slides [Online] Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-884-complex-digital-systems-spring-2005/lecture-notes/l15_testing.pdf

[5]. J. Aynsley. (June 2010). UVM Verification Primer

[6]. Testbench – an overview | ScienceDirect Topics [Online] Available: https://www.sciencedirect.com/topics/computer-science/testbench

[7]. Tutorial - What is a Testbench [Online] Available: https://www.nandland.com/articles/what-is-a-testbench-fpga.html

[8]. What is the advantage of UVM over systemverilog? [Online] Available: https://www.quora.com/What-is-the-advantage-of-UVM-over-systemverilog [Accessed 3 - Jun – 2019]

[9]. Verification Methodologies Made Easy — Aldec - Youtube [Online] Available: https://www.youtube.com/watch?v=ZlDTA9E5gEU

[10]. Introduction to Verification Methodology – Youtube [Online] Available: https://www.youtube.com/watch?v=-yO2ID-3dTk&t=183s

[11]. Introduction to UVM - The Universal Verification Methodology for SystemVerilog – YouTube [Online] Available: https://www.youtube.com/watch?v=imH4CFmVGWE&t=23s

[12]. Basic UVM | Universal Verification Methodology | Verification Academy [Online] Available: https://verificationacademy.com/courses/basic-uvm

[13]. https://verificationacademy.com

[14]. UVM Users guide 1.2 by Accellera