WiSig Networks

RTL Designs with AXIS Interfaces

Table of Contents

Copyright (c) 2016-2024, WiSig Networks Pvt Ltd. All rights reserved.

All information contained herein is property of WiSig Networks Pvt. Ltd., unless otherwise explicitly mentioned. The intellectual and technical concepts in this file are proprietary to WiSig Networks and may be covered by grants or in process of national and international patents and are protected by trade secrets and copyright law. Redistribution and use in source and binary forms of the content in this file, with or without modification are not permitted unless permission is explicitly granted by WiSig Networks.

Revision History

Version	**Date (dd/mm/yy)**	**Description**	**Author(s)**	**Reviewer(s)**
1	2/03/2024		Alavala chinnapa reddy	

Objective

- This document consists of the design approaches related to AXI Stream Interface.
- Understanding the AXI stream interface protocol.
- Learning the implementation insights of AXIS interfaces using Verilog/System Verilog.

AXI Stream Interface

1. Introduction

The **Advanced eXtensible Interface** (**AXI**) is an on-chip communication bus protocol and is part of the Advanced Microcontroller Bus Architecture specification (AMBA). AXI had been introduced in 2003 with the AMBA3 specification. In 2010, a new revision of AMBA, AMBA4, defined the AXI4, AXI4-Lite and AXI4-Stream protocols. AXI is royalty-free and its specification is freely available from ARM.

AMBA AXI4, AXI4-Lite and AXI4-Stream have been adopted by **Xilinx** and many of its partners as a main communication bus in their products. For designs that require high speed data transfers use AXI4-Stream interfaces.

Use the simplified AXI4-Stream protocol for write and read transactions. When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement these signals:

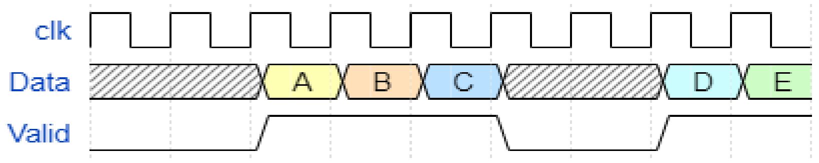
- Data
- Valid

Optionally, when you map scalar DUT ports to an AXI4-Stream interface, you can model these signals:

- Ready
- Other protocol signals, such as:
  - TSRTB
  - TKEEP
  - TLAST
  - TID
  - TDEST
  - TUSER

2. Data and Valid Signals

When the Data signal is valid, the Valid signal is asserted. This diagram illustrates the Data and Valid signal relationship according to the simplified streaming protocol. When you run the IP core generation workflow, HDL Coder adds a streaming interface module in the HDL IP core that translates the simplified protocol to the full AXI4-stream protocol. In this image, the clock signal is clk.



3. Ready Signal(Optional)

Downstream components use back pressure to tell upstream components they are not ready to receive data. The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. Use the Ready signal to:

- Apply back pressure in an AXI4-Stream slave interface. For example, drop the Ready signal when the downstream component is not ready to receive data.
- Respond to back pressure in an AXI4-Stream master interface. For example, stop sending data when the downstream component Ready signal is low.

When you use a single streaming channel, by default, HDL Coder generates the Ready signal and the logic to handle the back pressure. The back pressure logic ties the Ready signal to the DUT Enable signal. When the input master Ready signal is low, the DUT is disabled, and the output slave Ready signal is driven low. Because HDL Coder generates the back pressure logic and Ready signal, when you use a single streaming channel, the Ready signal is optional and you do not have to model this signal at the DUT port.

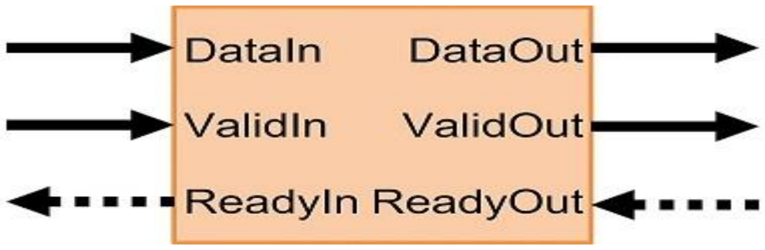
When you use multiple streaming channels, HDL Coder generates a ready signal and does not generate the back pressure logic. In a DUT that has multiple streaming channels:

- The master channel ignores the Ready signal from downstream components.
- The slave channel Ready signal is high, which causes upstream components to continue sending data.

The absence of a back pressure logic might result in data being dropped. To avoid data loss and to apply back pressure on the slave interface or respond to back pressure from the master interface in your design:

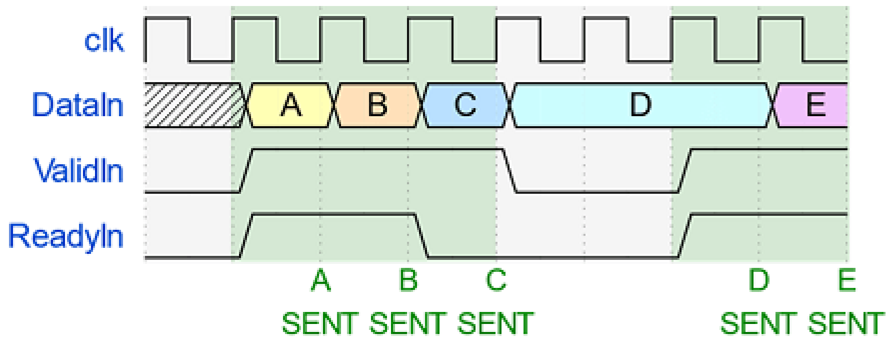
- Model the Ready signal for each additional stream interface.
- Map the modelled Ready signal to a DUT port for the additional interface.

When you do not model the Ready signal, the **Set Target Interface** task displays a warning that provides names of interfaces that require a Ready port. If your design does not require applying or responding to back pressure, ignore this warning.



AXI4-Stream Input

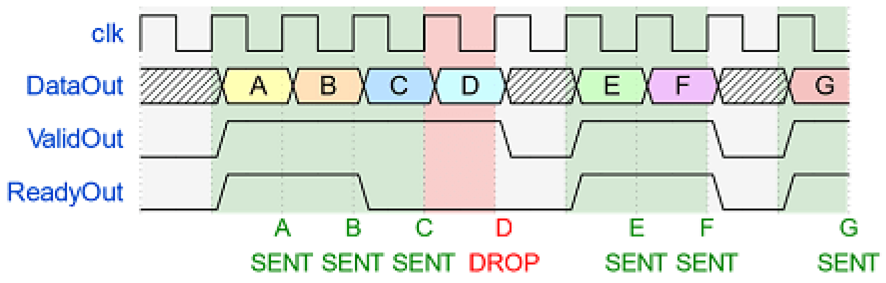
This image illustrates the timing relationship between the Data, Valid, and Ready signals according to the simplified streaming protocol. In this image clock is clk. The AXI4-Stream Slave module sends the DataIn and ValidIn signals after asserting the ReadyIn signal from the DUT. This is represented by data packets A,B,D, and E in the image. When you drop the ReadyIn signal, the module always sends one more DataIn and ValidIn signal. This is represented by data packet C in the image. When you model the ReadyIn signal, the DUT must be able to accept one more value after de-asserting the ready signal.



For example, if you have a first in first out (FIFO) in your DUT to store a frame of data, to apply backpressure to the upstream component, model the Ready signal based on the FIFO almost full signal.

AXI4-Stream Output

This image illustrates the timing relationship between the Data, Valid, and Ready signals according to the simplified streaming protocol. In this image clock is clk. You can send DataOut and ValidOut signals to the AXI4-Stream Master module after you assert the ReadyOut signal. This is represented by data packets A, B,E,F,andG in the image. You can optionally send one more DataOut and ValidOut signal after the ReadyOut signal drops. This is represented by data packet C in the image. You can only send one additional packet after the ReadyOut signal drops, subsequent data packets will be dropped until the Ready signal is asserted again. This is represented by data packet D in the image.



The optional one cycle latency between the Valid and Ready signals of the simplified streaming protocol allows you to use a classic or first word fall through (FWFT) FIFO to handle the backpressure from downstream components.

4. TLAST Signal (optional)

The AXI4-Stream interface on your DUT can optionally model a TLAST signal, which is used to indicate the end of a frame of data. If you do not model this signal, HDL Coder generates it for you. On the AXI4-Stream Slave interface, the incoming TLAST signal is ignored. On the AXI4-Stream Master interface, the autogenerated TLAST signal is asserted when the number of valid samples counts to the default frame length value. The default frame length value can be set by using the AXI4-Stream interface options in the Target Interface.

When the IP core has an AXI4 Slave interface, the default frame length value is stored in a programmable register in the IP core. You can change the default frame length during run time. When the default frame length register is changed in the middle of a frame, the TLAST counter state is reset to zero and the TLAST signal is asserted early. You can find the address for the programmable TLAST register in the IP core generation report.

Assignment-1

8-bit Register with AXI Stream interface

1. Introduction

- Implement an 8-bit register with AXI stream interface on both inputs and outputs
- Use only the DATA, VALID, READY and LAST signals in all the interfaces
- Verify its functionality using testbench in simulation

8-bit register with an AXI Stream interface. It serves as a data storage element that can receive data from a streaming source (such as an AXI Stream master) and provide the stored data to a streaming sink (such as an AXI Stream slave). The module operates synchronously with a clock signal (clk) and asynchronously with a reset signal (reset), following the AXI Stream protocol for data communication.

This axis_reg module is used to minimize Propagation delay caused by Combinational Path. Thus, input is registered and then given as output which has 1 clock cycle delay. We can also use registers to get more than 1 cycle delay.

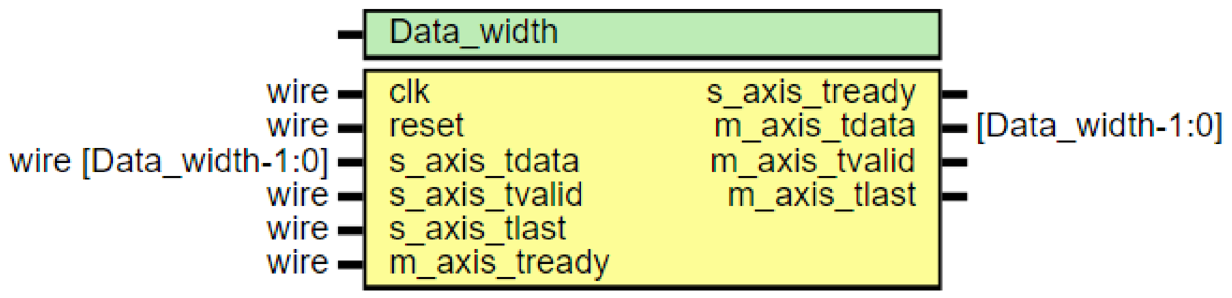
2. Implementation

The axis_reg module has a Parameter (Data_width) which tells input and output data widths and uses AXI Streaming Interface.

The AXI Streaming ports used in this module are **Data, Valid, Ready and Last.**

Connect input ready to output ready by registering because whenever output is ready to accept data input will also process some data and output will be seen in the next clock cycle because it has been registered. Therefore, the module has 1 cycle latency.

3. Block Diagram



3.1 Port Description

Generic name	Type	Value	Description
Data_width		8	

Port name	Direction	Type	Description
clk	input	wire	Clock signal used for synchronous operation. The module updates its internal state on the rising edge of this clock.
reset	input	wire	Asynchronous reset signal. When asserted (high), it resets the module's internal state to a default condition.
s_axis_tdata	input	wire [Data_width-1:0]	Input data bus of width 8 bits from the AXI Stream source.
s_axis_tvalid	input	wire	Input valid signal from the AXI Stream source, indicating the availability of valid data on the input data bus.
s_axis_tready	output	reg	Output ready signal to the AXI Stream source, indicating the readiness of the module to accept new data.
s_axis_tlast	input	wire	Input last signal from the AXI Stream source, indicating the last data beat of a packet.
m_axis_tdata	output	reg [Data_width-1:0]	Output data bus of width 8 bits to the AXI Stream sink, providing the stored data.

m_axis_tvalid	output	reg	Output valid signal to the AXI Stream sink, indicating the availability of valid data on the output data bus.
m_axis_tready	input	wire	Input ready signal indicates that another module is ready to accept data.
m_axis_tlast	output	reg	Output last signal to the AXI Stream sink, indicating the last data beat of a packet.

Internal Signals

Name	Type	Description
reg_data=8'd0	reg [Data_width-1:0]	Internal register to store the incoming data from the AXI Stream source.
ready_out	reg	Internal signal indicating the readiness of the module to accept new data from the AXI Stream source.
valid_out	reg	Internal signal to track the validity of the data stored in the register.
tlast_out	reg	Internal signal to track the last beat of the packet.

Module Code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_2/Assignment_1/8bit_reg_axis.v

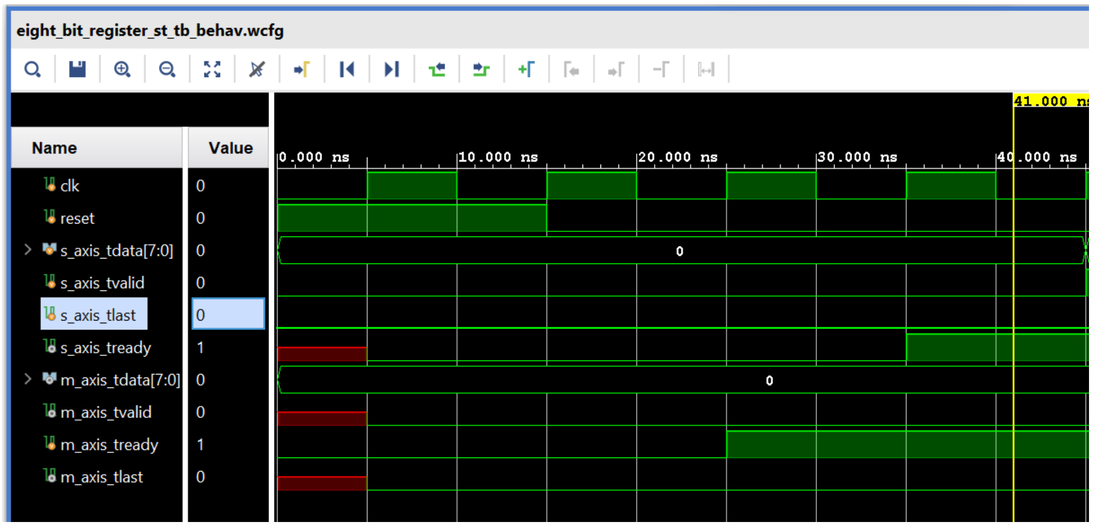
- ➢ Whenever m_axis_tready is high module will start processing data
- ➢ Whenever s_axis_tvalid and s_axis_tready is high data is being stored into a register reg_data.
- ➢ In the next cycle stored value is given out to m_axis_tdata.

Test bench Code

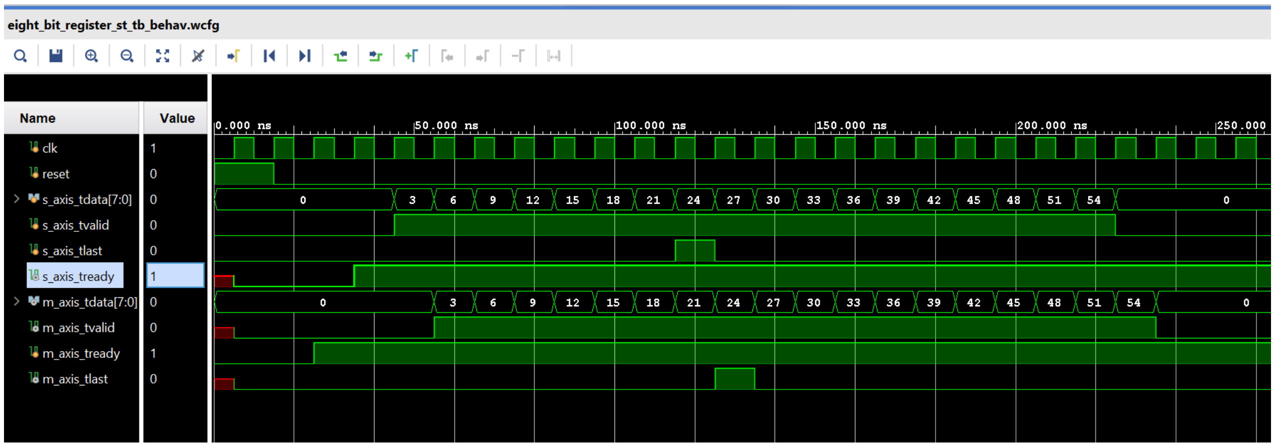
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_2/Assignment_1/axis_8bit_register_tb.v

4. Test Cases

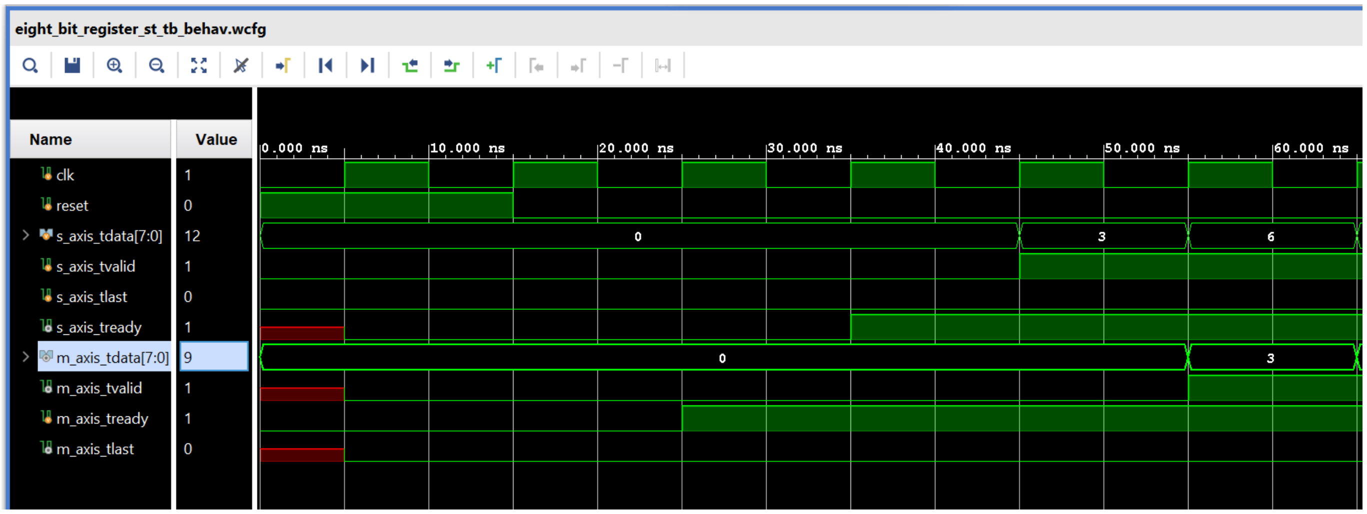
- **When reset is asserted no input and no output transfer takes place.**



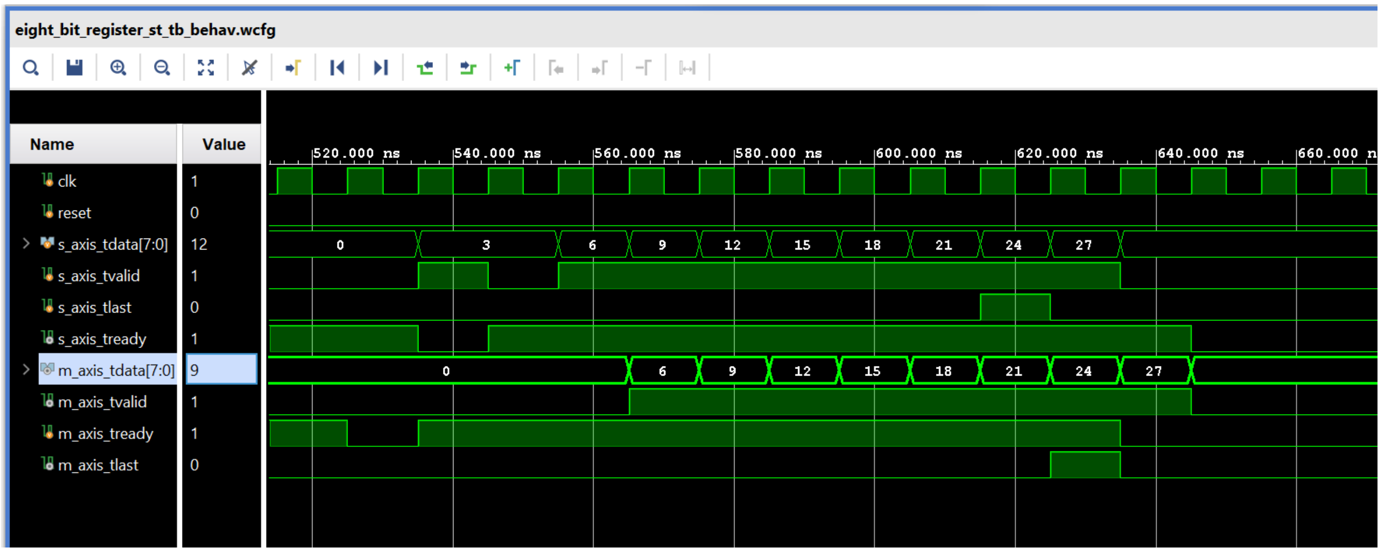
- **TVALID and TREADY asserted simultaneously**



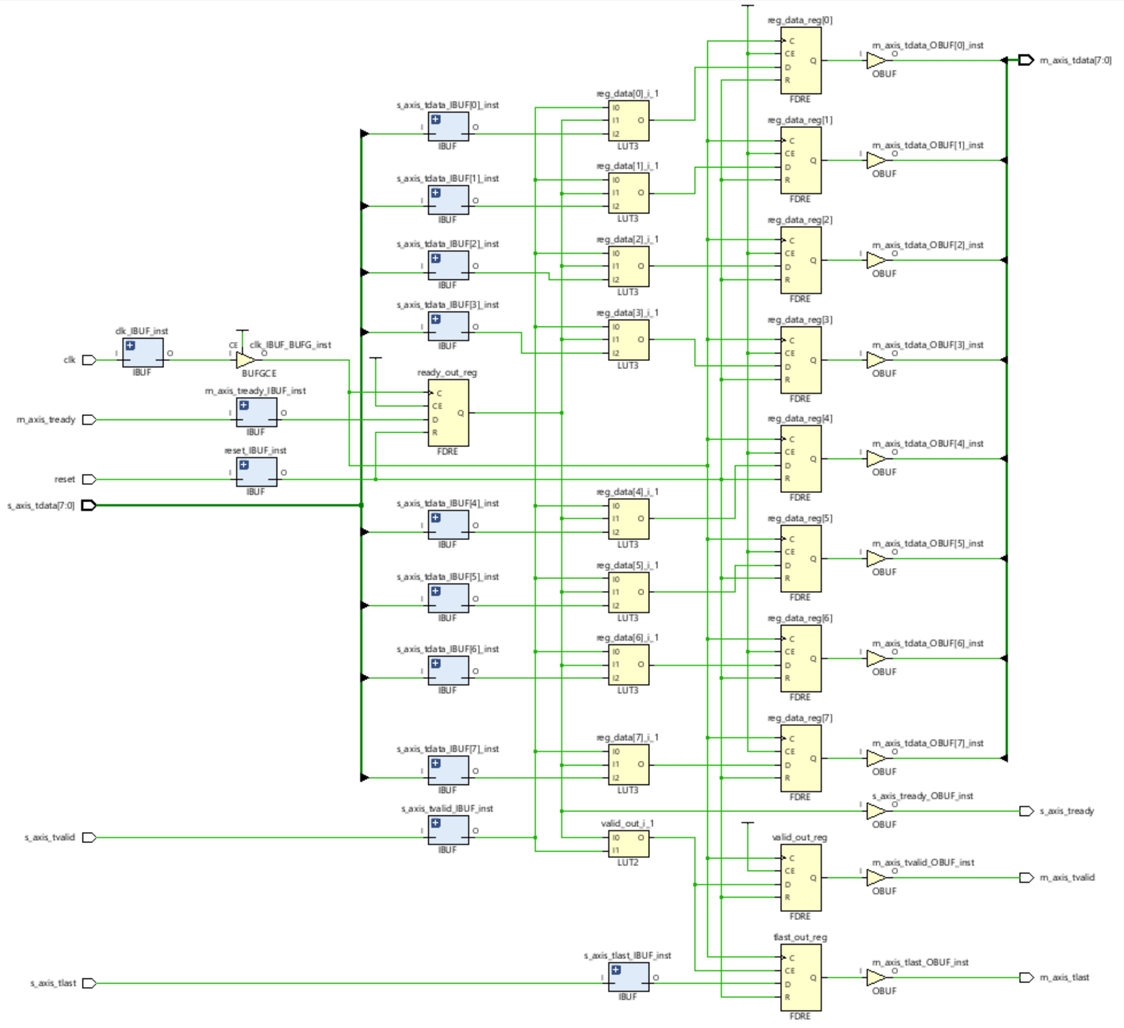
- **TREADY asserted before TVALID**



- **TREADY Low and TVALID High**



5. Schematic diagram



6. Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	5	230400	0.01
FF	11	460800	0.01
IO	24	360	6.67
BUFG	1	544	0.18

Assignment – 2

2X1 MUX with AXI stream interface

- Implement a 2X1 mux with AXI stream interface on both inputs and outputs
- Use only the DATA, VALID, READY and LAST signals in all the interfaces
- Verify its functionality using testbench in simulation

1. What is Multiplexing?

Multiplexing is the process of combining one or more signals and transmitting on a single channel. In analog communication systems, a communication channel is a scarce quantity, which must be properly used. For cost-effective and efficient use of a channel, the concept of Multiplexing is very useful as it allows multiple users to share a single channel in a logical way.

The three common types of Multiplexing approaches are:

- Time
- Frequency
- Space

Two of the best examples of Multiplexing Systems used in our day-to-day life are the landline telephone network and the Cable TV.

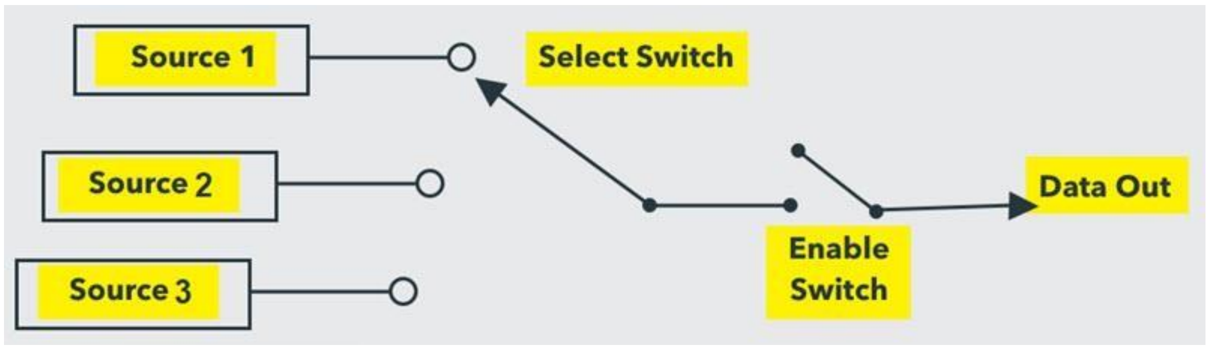
The device which is responsible for Multiplexing is known as Multiplexer. Multiplexers are used for both Analog and Digital signals. Let us focus on digital signals in this tutorial, to keep things simple. A multiplexer is the most frequently used combinational circuit and it is an important building block in many in digital systems.

These are mostly used to form a selected path between multiple sources and a single destination. A basic multiplexer has various data input lines and a single output line. These are found in many digital system applications such as data selection and data routing, logic function generators, digital counters with multiplexed displays, telephone network, communication systems, waveform generators, etc. In this article we are going to discuss about types of multiplexers and its design.

What is a Multiplexer?

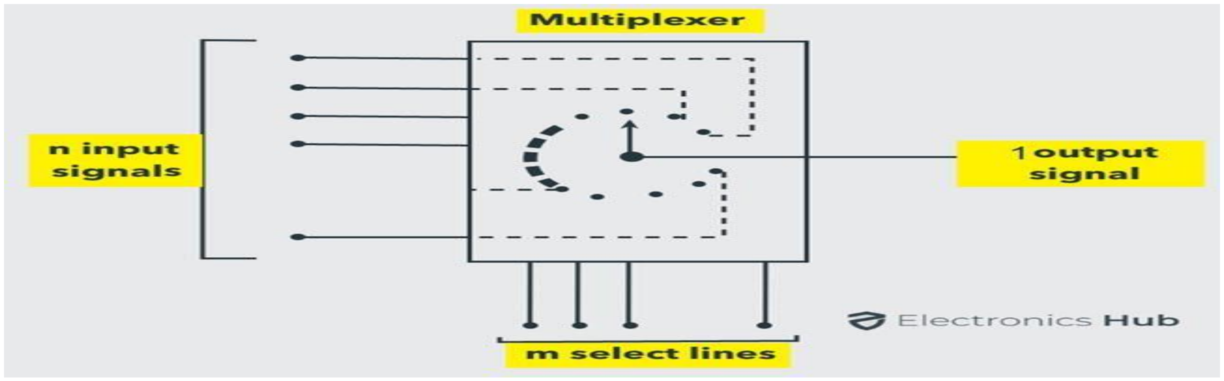
The multiplexer or MUX is a digital switch, also called as data selector. It is a Combinational Logic Circuit with more than one input line, one output line and more than one selects line. It accepts the binary information from several input lines or sources and depending on the set of select lines, a particular input line is routed onto a single output line.

The basic idea of multiplexing is shown in figure below in which data from several sources are routed to the single output line when the enable switch is ON. This is why, multiplexers are also called as 'many to one' combinational circuits.



The below figure shows the block diagram of a multiplexer consisting of n input lines, m selection lines and one output line. If there are m selection lines, then the number of possible input lines is 2^m. Alternatively, we can say that if the number of input lines is equal to 2^m, then m selection lines are required to select one of n (consider $2^m = n$) input lines.

This type of multiplexer is referred to as $2^n \times 1$ multiplexer or 2^n-to-1 multiplexer. For example, if the number of input lines is 4, then two select lines are required. Similarly, to select one of 8 input lines, three select lines are required.

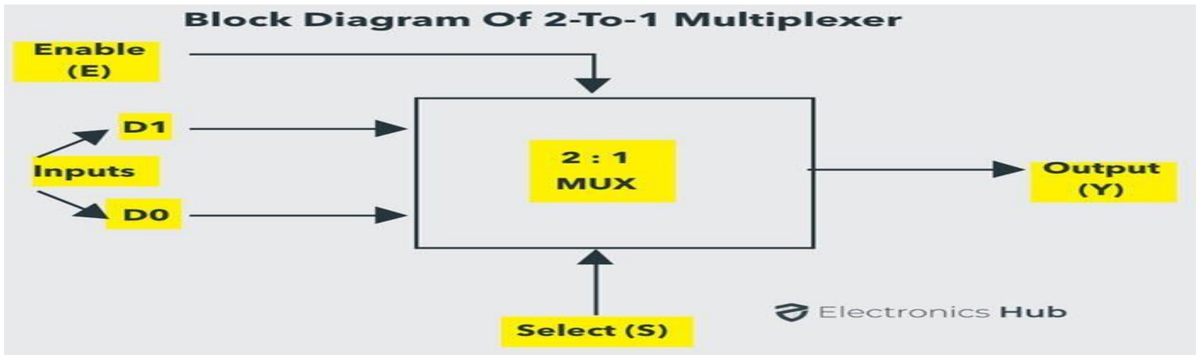


Generally, the number of data inputs to a multiplexer is a power of two such as 2, 4, 8, 16, etc. Some of the most frequently used multiplexers include 2-to-1, 4-to-1, 8-to-1 and 16-to-1 multiplexers.

These multiplexers are available in IC forms with different input and select line configurations. Some of the available multiplexer ICs include 74157 (Quad 2-to-1 MUX), 78158 (Quad 2-to-1 MUX with inverse output), 74153 (4-to-1 MUX), 74152 (8-to-1 MUX) and 74150 (16-to-1 MUX).

2-to-1 Multiplexer
A 2-to-1 multiplexer consists of two inputs D0 and D1, one selects input S and one output Y. Depending on the select signal, the output is connected to either of the inputs. Since there are two input signals, only two ways are possible to connect the inputs to the outputs, so one selects is needed to do these operations.

If the select line is low, then the output will be switched to D0 input, whereas if select line is high, then the output will be switched to D1 input. The figure below shows the block diagram of a 2-to-1 multiplexer which connects two 1-bit inputs to a common destination.



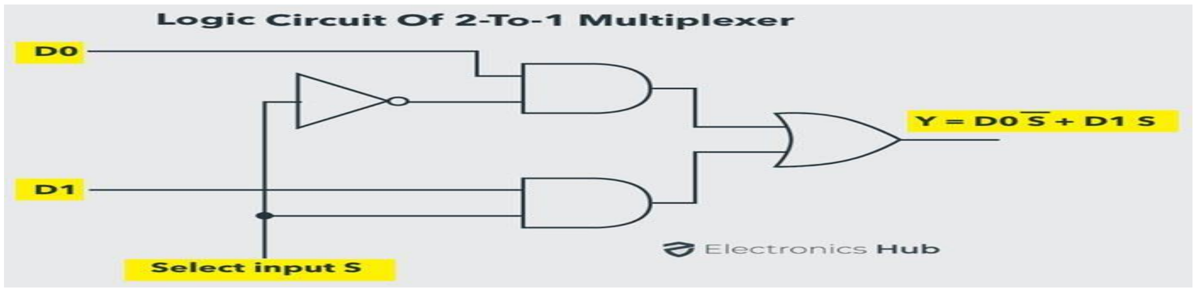
The truth table of the 2-to-1 multiplexer is shown below. Depending on the value of the select input, the inputs i.e., D0, D1 are produced at outputs. The output is D0 when Select value is S = 0 and the output is D1 when Select value is S = 1.

S	**D0**	**D1**	**Y**
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

'X' in the above truth table denotes a do not care condition. So, ignoring the do not care conditions, we can derive the Boolean Expression of a typical 2 to 1 Multiplexer as follows:

From the above output expression, the logic circuit of 2-to-1 multiplexer can be implemented using logic gates as shown in figure. It consists of two AND gates, one NOT gate and one OR gate. When the select line, S=0, the output of the lower AND gate is zero, but the output of upper AND gate is D0. Thus, the output generated by the OR gate is equal to D0.

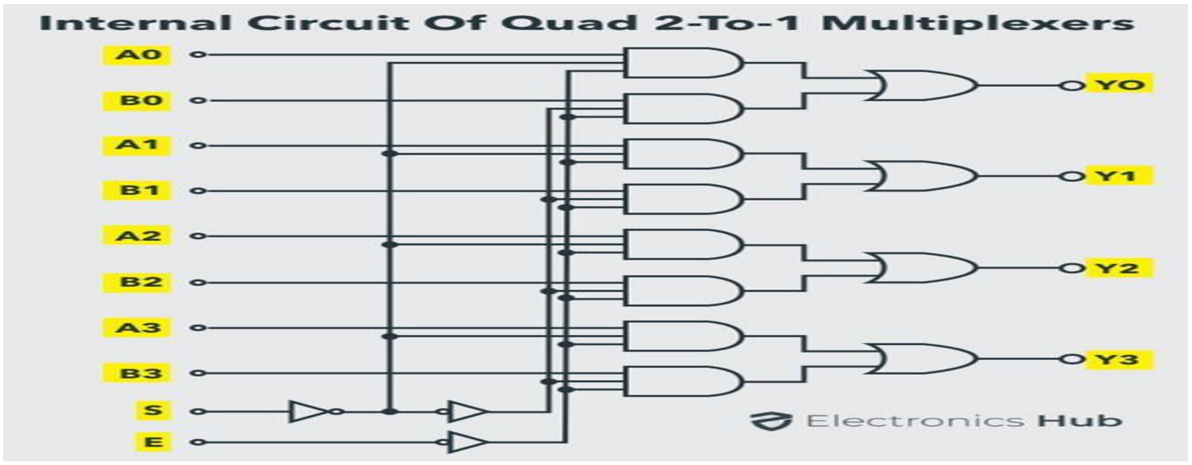
Similarly, when S=1, the output of the upper AND gate is zero, but the output of lower AND gate is D1. Therefore, the output of the OR gate is D1. Thus, the above given Boolean expression is satisfied by this circuit.


Logic Circuit Of 2-To-1 Multiplexer
D0
D1
Select input S
$Y = D0\,\overline{S} + D1\,S$
Electronics Hub

To efficiently use the Silicon, IC Manufacturers fabricate multiple Multiplexers in a single IC. Generally, four 2 lines to 1-line multiplexers are fabricated in a single IC. Some of the popular ICs of 2 to 1 multiplexer include IC 74157 and IC 74158.

Both these ICs are Quad 2-to-1 Multiplexers. While IC 74157 has a normal output, the IC74158 has an inverted output. There is only one selection line, which controls the input lines to the output in all four multiplexers.

The output Y0 can be either A0 or B0 depending on the status of the select line. Similarly, Y1 can be either A1 or B1, Y2 can be either A2 or B2 and so on. There is an additional Strobe or Enable control input E/Strobe, which enables and disables all the multiplexers, i.e., when E=1, outputs of all the multiplexer is zero irrespective of the value of S.



All the multiplexers are activated only when the E / Strobe input is LOW.

2. Implementation

The Verilog module axis_mux is a basic multiplexer for AXI4-Stream interfaces. It selects between two input AXI streams based on a control signal called select. When select is low, it forwards data from the first input stream to the output, and when select is high, it forwards data from the second input stream to the output. The module includes logic to synchronize the control and data signals with the clock signal, ensuring proper operation in synchronous digital systems.

Clock Domain Crossing and Reset Handling:

- The module synchronizes all inputs and internal signals with the clock signal (clk).
- Asynchronous reset (reset) is used to initialize the internal state of the module.

Data Selection:

- When select is low, the module forwards data from the first input stream to the output.
- When select is high, the module forwards data from the second input stream to the output.

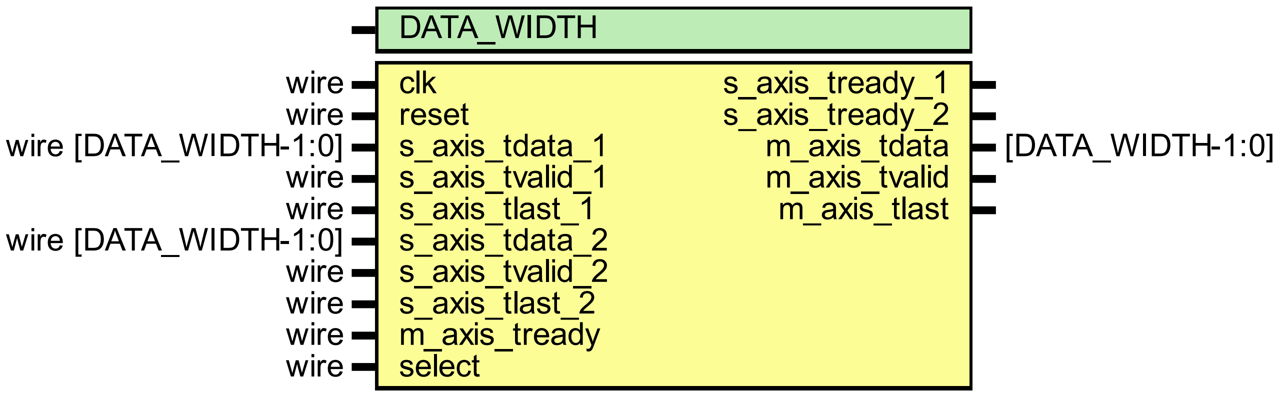
Handshaking:

- Proper handshaking is maintained between the selected input stream and the output stream to ensure correct data transmission.
- The output valid signal (m_axis_tvalid) is controlled based on the valid signals of the selected input stream.
- The output ready signal (m_axis_tready) is connected to the ready signals of both input streams based on the select signal.

Packet End Detection:

- The module ensures that the m_axis_tlast signal is correctly set based on the selected input stream's s_axis_tlast signal.

3. Block Diagram



3.1 Port Description

Generic name	Type	Value	Description
DATA_WIDTH		8	

Ports

Port name	Direction	Type	Description
clk	input	wire	Clock signal used for synchronizing the internal operations of the module.
reset	input	wire	Asynchronous reset signal used to initialize the internal state of the module.
s_axis_tdata_1	input	wire [DATA_WIDTH-1:0]	Data input of the first AXI stream.
s_axis_tvalid_1	input	wire	Valid signal of the first AXI stream indicating that valid data is present on s_axis_tdata_1.
s_axis_tready_1	output	Reg	Ready signal of the first AXI stream indicating that the receiver is ready to accept data.
s_axis_tlast_1	input	wire	Last signal of the first AXI stream indicating the end of a

Port name	Direction	Type	Description
			packet.
s_axis_tdata_2	input	wire [DATA_WIDTH-1:0]	Data input of the second AXI stream.
s_axis_tvalid_2	input	wire	Valid signal of the second AXI stream indicating that valid data is present on s_axis_tdata_2.
s_axis_tready_2	output	Reg	Ready signal of the second AXI stream indicating that the receiver is ready to accept data.
s_axis_tlast_2	input	wire	Last signal of the second AXI stream indicating the end of a packet.
m_axis_tdata	output	Reg [DATA_WIDTH-1:0]	Data output of the selected AXI stream, forwarded from either s_axis_tdata_1 or s_axis_tdata_2.
m_axis_tvalid	output	Reg	Valid signal indicating that valid data is present on m_axis_tdata.
m_axis_tready	input	wire	Ready signal indicating that the receiver is ready to accept data from m_axis_tdata.
m_axis_tlast	output	Reg	Last signal indicating the end of a packet on m_axis_tdata.
select	input	wire	Control signal used to select between the two input streams. When select is low, the first

Port name	Direction	Type	Description
			input stream is selected; when select is high, the second input stream is selected.

Signals

Name	Type	Description
reg_data=8'd0	reg [DATA_WIDTH-1:0]	
ready_out_1	reg	
ready_out_2	reg	
valid_out	reg	
tlast_out	reg	

3.2 Module Code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_2/Assignment_2/axis_mux_2X1.v

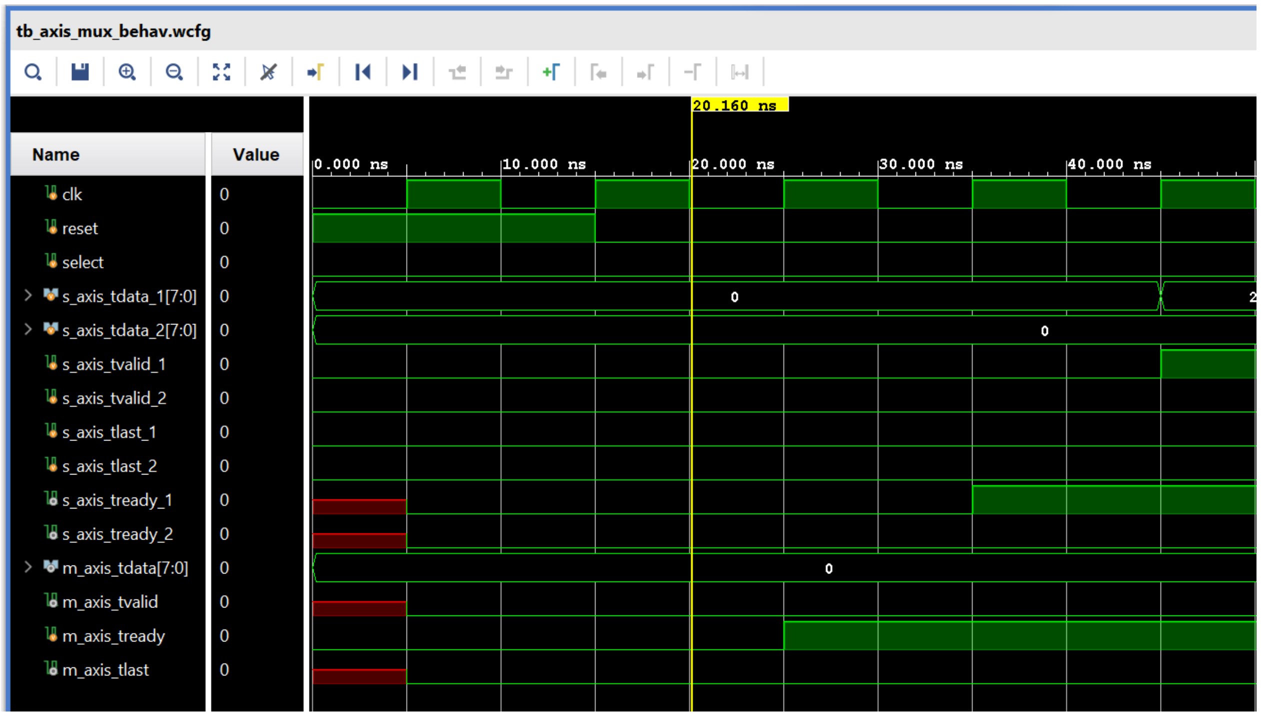
4.Testing Procedure

4.1 Testbench code

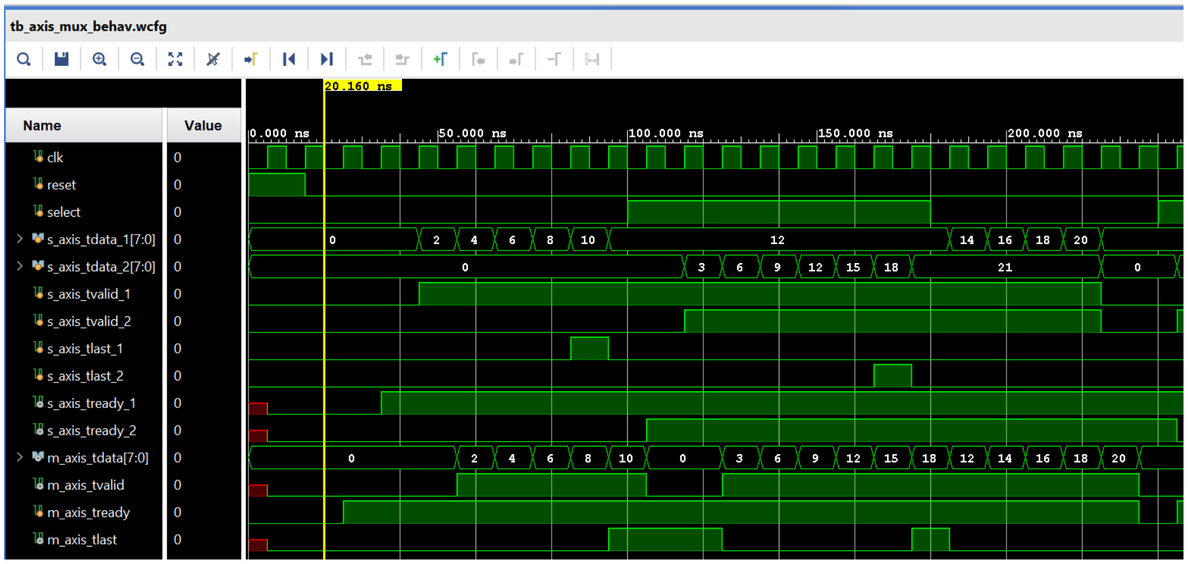
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_2/Assignment_2/axis_mux_2X1_tb.v

4.2 Test Cases

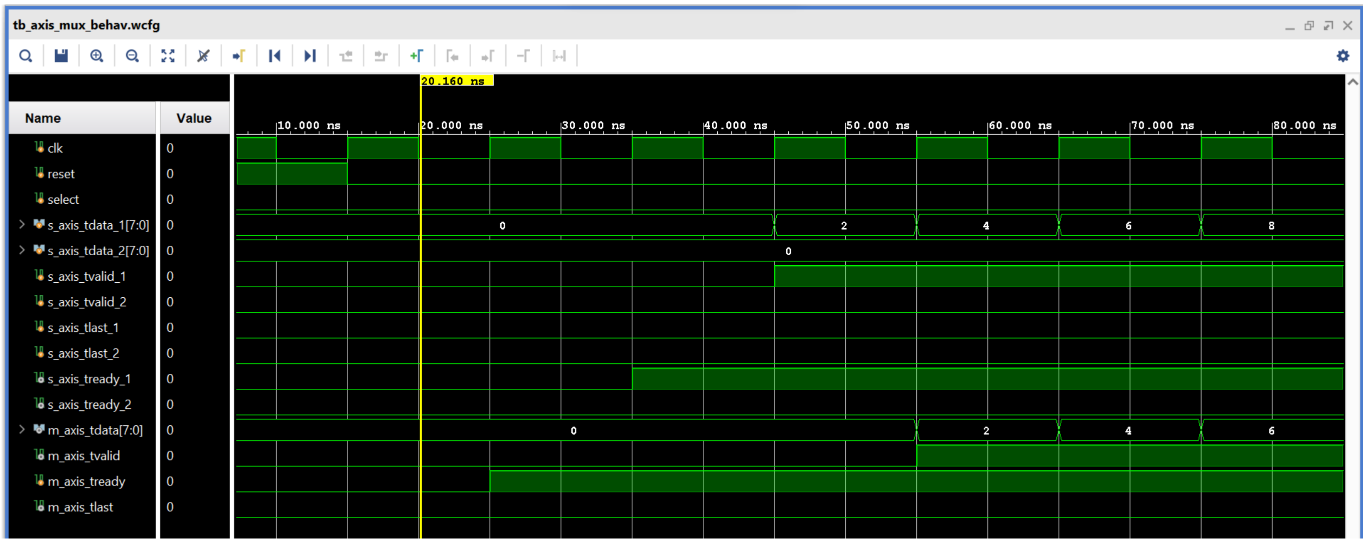
- **When reset is asserted no input and no output transfer takes place.**



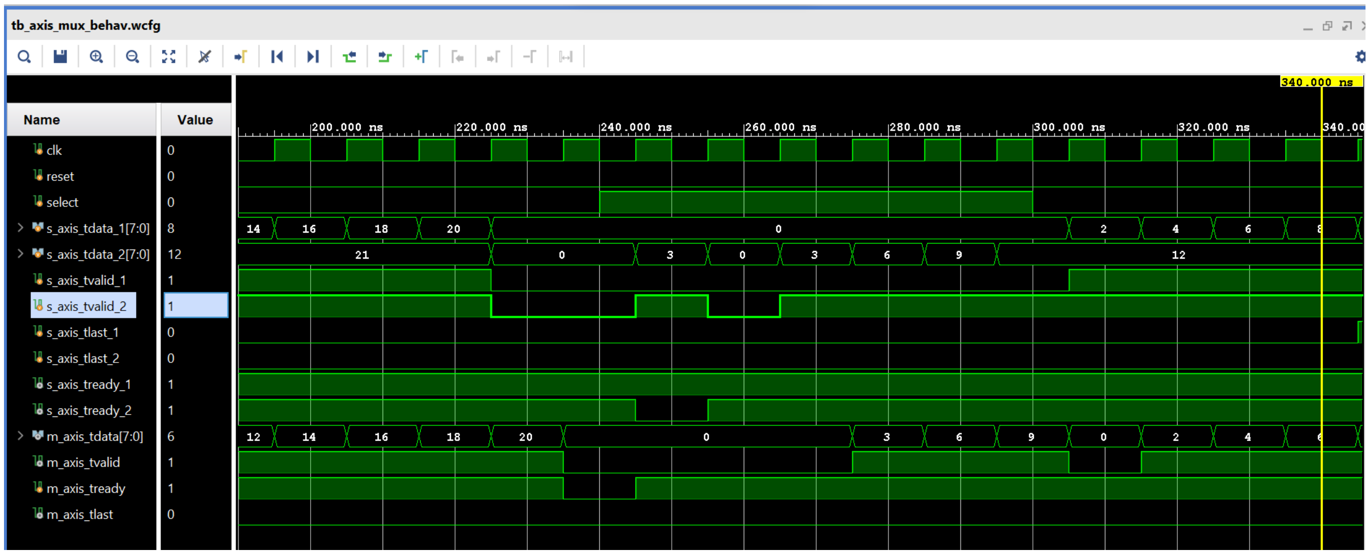
- **TVALID and TREADY asserted simultaneously**



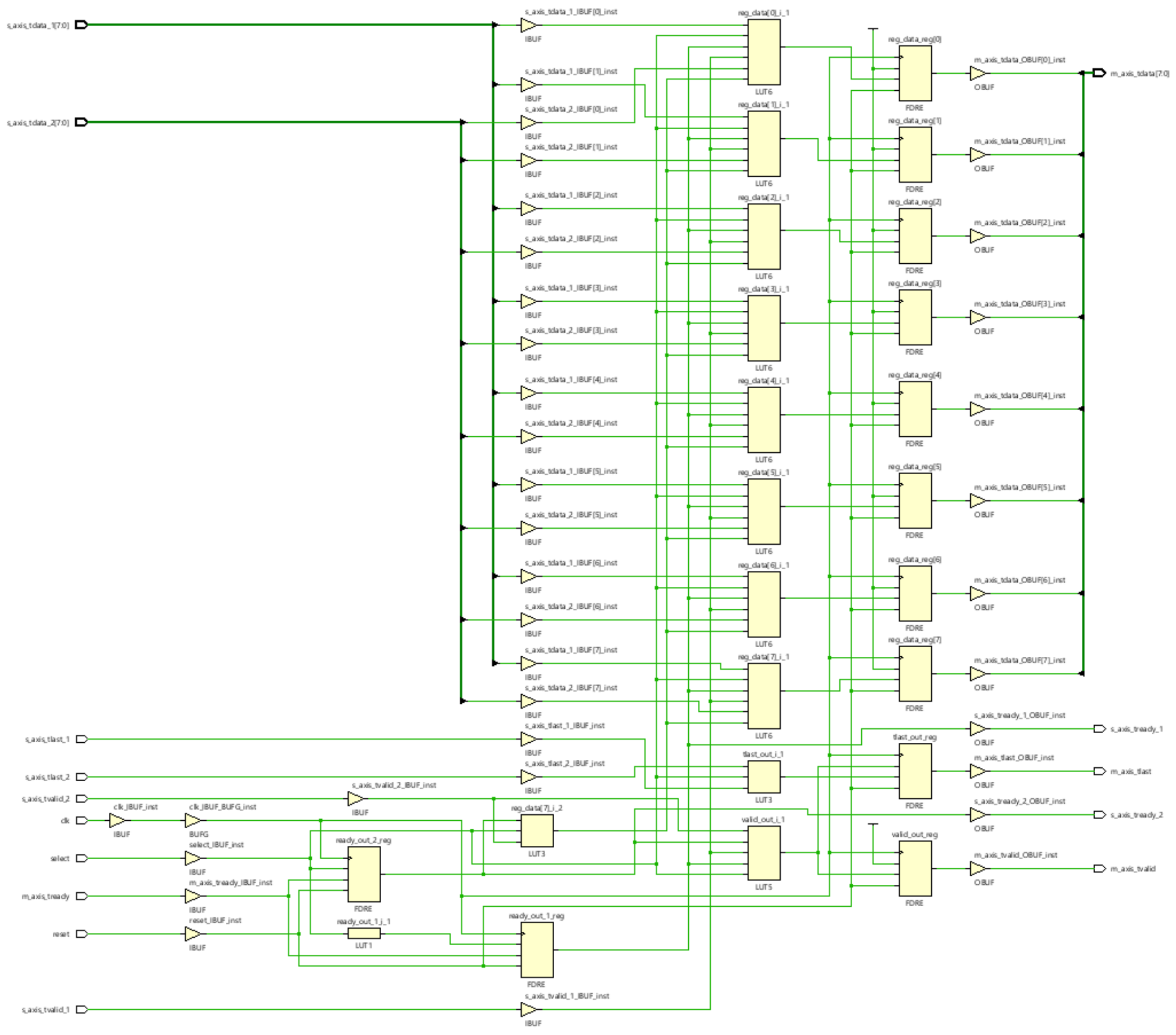
- **TREADY asserted before TVALID**



- **What will be happening when we give valid data when input is not ready.**



5. Schematic diagram



6. Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	11	41000	0.03
FF	12	82000	0.01
IO	36	300	12.00
BUFG	1	32	3.13

Assignment - 3:

4096 FIFO with 2 2048 FIFOs using AXI interface

1. Introduction

This module is designed to store and retrieve data in fifo of depth 4096 using 2 2048 fifo modules which are cascaded. It provides functionality for both writing data into the FIFO and reading data from fifo simultaneously.

Synchronous FIFO

A **Synchronous FIFO** is a First-In-First-Out queue in which there is a single clock pulse for both data write and data read. In Synchronous FIFO the read and write operations are performed at the same rate. The number of rows is called depth or number of words of FIFO and number of bits in each row is called as width or word length of FIFO. This kind of FIFO is termed as Synchronous because the rate of read and write operations are same.

Basically, Synchronous FIFO are used for High-speed systems because of their high operating speed. Synchronous FIFO are easier to handle at high speed because they use free running clocks whereas in case of Asynchronous FIFO, they use two different clocks for read and write. Synchronous FIFO is more complex than the Asynchronous FIFO.

Operations in the synchronous FIFO:

1. **Write Operation:** The operation involves in writing or storing the data in to the FIFO memory till it rises any flag conditions for not to write anymore.

- To perform a write operation the data to be written is given at **DIN** port and **write EN** is to be set high then at the next rising edge the data will be written.

2. **Read Operation:** Read operation performed when we want to get data out from the FIFO memory until it informs there is no more data to be read from the memory, the condition called empty condition. Empty conditions are generated using empty flags.

- To perform read operation, we need to set read EN high then at next rising edge the data to be read will be at **DOUT**

Pointers to control operations:

1. **Write Pointer:** This pointer controls the write operation of the FIFO. It used to point to the FIFO memory location where the data will be written.

2. **Read Operation**: The read operation is controlled by the read pointer. It will be pointing the location from where next data is to be read.

Flags in FIFO:

Synchronous FIFO provides us with few flags ,to determine the status or to interrupt the operation of FIFO.

1. **EMPTY flag:** This flag is useful to avoid the case of the invalid request of read operation when the FIFO is already empty.
2. **FULL flag:** This flag is useful to avoid the case of the invalid request of write operation when the FIFO is already full.

Here we use a variable named count, which will count the total number of words in the FIFO. Based on value of count the flag logic will assign values to flags. If the count reaches a value equal to the size of FIFO it will assign FULL flag as logic high and if the count becomes zero it will assign EMPTY flag as logic high.

Asynchronous FIFO

An **Asynchronous FIFO** refers to a FIFO where the data values are written to the FIFO at a different rate and data values are read from the same FIFO at a different rate, both at the same time. The reason for calling it Asynchronous FIFO, is that the read and write clocks are not Synchronized.

The basic need for an Asynchronous FIFO arises when we are dealing with systems with different data rates. For the rate of data flow being different, we will be needing Asynchronous FIFO to synchronize the data flow between the systems. The main work of an Asynchronous FIFO is to pass data from one clock domain to another clock domain.

How to use it?

Here for example consider two systems- system A and system B. Let the data from system A is to be fed to system B, and the data output rate of system A is different than the data input rate of system B. Here we can use an Asynchronous FIFO to Synchronize system A with system B. The data from system A will be taken as input in FIFO at a rate of system A and when FIFO is full the data will be given to system B at its respective rate.

Operations in Asynchronous FIFO:

1. **Write Operation:**

This operation involves writing or storing the data into FIFO memory till it rises any flag conditions for not to write anymore.

- To perform a write operation the data to be written is given at the **DIN** port and the **write EN** is to be set high then at the next rising edge of the **write clock** the data is written.

2. **Read Operation:**

Read operation performed when we must get data out from the FIFO memory until it informs there is no more data to be read from the memory. Empty conditions are generated using empty flags.

- To perform read operation, we need to set **Read EN** high then at the next rising edge of **reading clock** the data to be read is at **DOUT**

Pointers to control Operations:

1. **Write Pointer:** This pointer controls the write operation of the FIFO. It points to the memory location where next data is to be written.
2. **Read Pointer:** The read operation is controlled by read pointer. It points to the location from where next data is to be read.

Flags in FIFO:

Asynchronous FIFO provides us with following two flags, to determine the status and to interrupt the operation of FIFO.

1. **EMPTY flag:** This flag is useful to avoid the case of invalid request of read operation when the FIFO is already empty.
2. **FULL flag:** This flag is useful to avoid the case of invalid request of write operation when the FIFO is already full.

2. Implementation

FIFO

This Verilog module implements a synchronous FIFO (First-In-First-Out) buffer with the specified parameters. Here's a breakdown of the logic:

Parameter Declaration: Parameters like DataWidth, Depth, and PtrWidth are declared to define the width of data, depth of the FIFO, and pointer width respectively.

Port Declaration: Inputs include datain (data input), clk (clock), rst (reset), wr_en (write enable), rd_en (read enable). Outputs are data_out (data output), full, and empty.

Internal FIFO Declaration: fifo is an array of registers representing the FIFO buffer, with its depth determined by the parameter Depth.

Initialization: The initial block initializes the FIFO and pointers. It sets all elements of the FIFO to zero, and initializes the read and write pointers to zero.

Clocked Always Block: The always @(posedge clk) block handles operations on the rising edge of the clock.

Reset Handling: If rst is asserted, all internal signals including data_out, write_ptr, read_next, read_ptr, and counter are reset to initial values.

Write Operation: If wr_en is asserted and the FIFO is not full, incoming data (datain) is written to the FIFO at the current write_ptr. write_ptr is then incremented, and counter is incremented to track the number of elements in the FIFO. If write_ptr reaches the depth, it wraps around to zero.

Read Operation: If rd_en is asserted and the FIFO is not empty, read_next is incremented to point to the next element to be read. If it reaches the depth, it wraps around to zero. Then, data_out is updated with the data at read_ptr and read_ptr is updated to read_next.

Full and Empty Checking: Using assign statements, full is set if the FIFO is full (i.e., counter equals Depth), and empty is set if the FIFO is empty (i.e., write_ptr equals read_next).

FIFO_AXIS

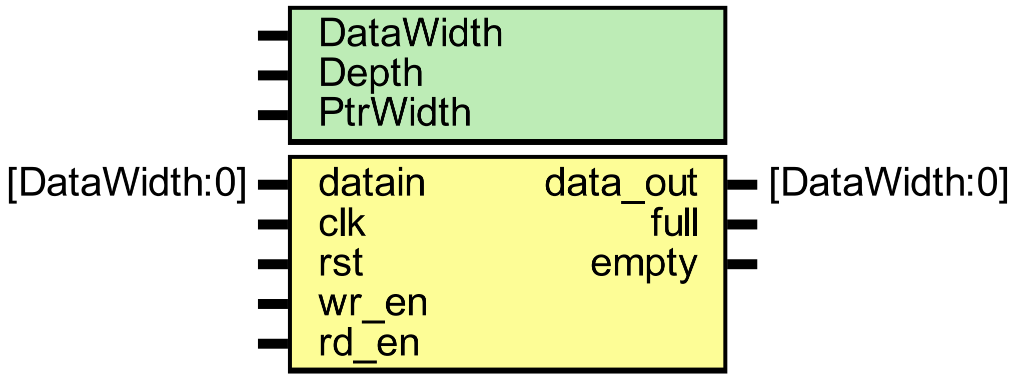
Instantiation of sync_fifo module: This module represents the synchronous FIFO memory. It's instantiated with the specified parameters and connected to the inputs and outputs of the main module.

Combinational logic for output ready signal (ready_out): This logic determines the value of ready_out based on the reset signal or FIFO full condition.

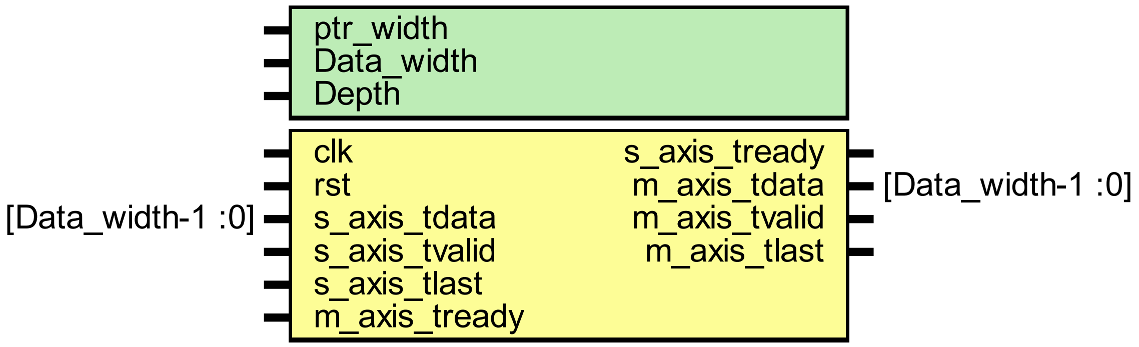
Combinational logic for output valid signal (valid_out): This logic determines the value of valid_out based on the reset signal or FIFO empty condition.

3. Block Diagram

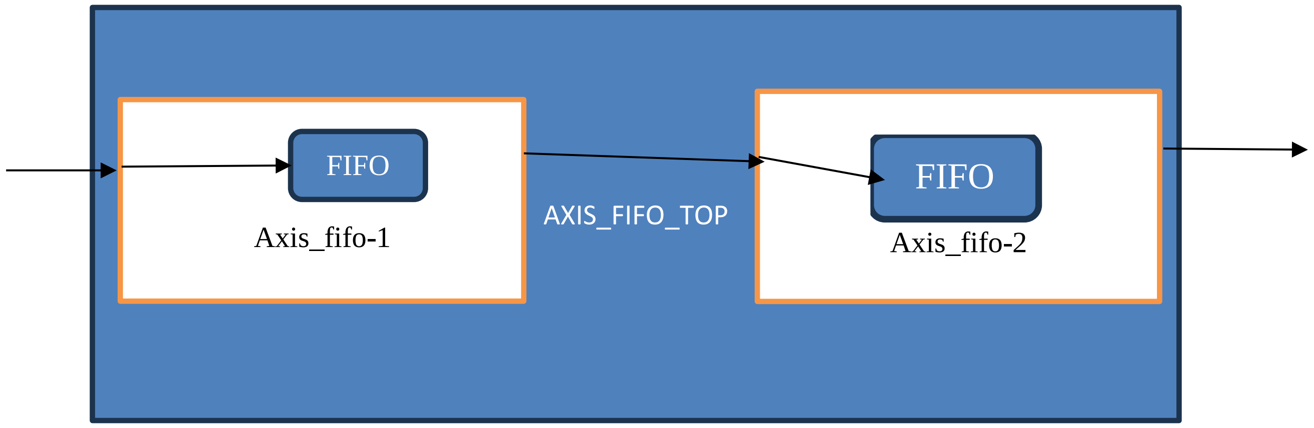
FIFO



FIFO_AXIS



Axis_fifo_top:



3.1 Port Description

FIFO

Generics

Generic name	Type	Value	Description
DataWidth		31	Width of the data bus.
Depth		2048	Depth of the FIFO buffer, i.e., the number of elements it can store.
PtrWidth		12	Width of the pointers used to address the FIFO buffer. It's calculated as the ceiling of the base-2 logarithm of the depth ($clog2(Depth)).

Ports

Port name	Direction	Type	Description
datain	input	[DataWidth:0]	Input data bus.
clk	input		Clock input.
rst	input		Reset input.
wr_en	input		Write enable input.

Port name	Direction	Type	Description
rd_en	input		Read enable input.
data_out	output	[DataWidth:0]	Output data bus.
full	output		Output indicating whether the FIFO is full.
empty	output		Output indicating whether the FIFO is empty.

Signals

Name	Type	Description
fifo[Depth-1:0]	reg [DataWidth:0]	Array of registers representing the FIFO buffer.
write_ptr	reg [PtrWidth:0]	Pointers for writing into the FIFO.
write_next	reg [PtrWidth:0]	Pointers for writing into the FIFO.
read_ptr	reg [PtrWidth:0]	Pointers for reading from the FIFO.
read_next	reg [PtrWidth:0]	Pointers for reading from the FIFO.
counter	reg [PtrWidth:0]	Counter to keep track of the number of elements currently stored in the FIFO.

FIFO_AXIS

Generic name	Type	Value	Description
Data_width		32	Width of the data bus.
Depth		2048	Depth of the FIFO buffer, i.e., the number of elements it can store.
ptr_width		12	Width of the pointers used to address the FIFO buffer. It's calculated as the ceiling of the base-2 logarithm of the depth ($clog2(Depth)).

Ports

Port name	Direction	Type	Description
clk	input		Clock input.
rst	input		Reset input.
s_axis_tdata	input	[DataWidth-1 :0]	Input data bus.
s_axis_tvalid	input		Input data valid signal.
s_axis_tlast	input		Input data last signal.
s_axis_tready	output		Output data ready signal.
m_axis_tdata	output	[DataWidth-1 :0]	Output data bus.

Port name	Direction	Type	Description
m_axis_tvalid	output		Output data valid signal.
m_axis_tlast	output		Output data last signal.
m_axis_tready	input		Input data ready signal
full	output		Full flag indicating FIFO is full.
empty	output		Empty flag indicating FIFO is empty.

Signals

Name	Type	Description
s_axis_tdata_reg	logic [DataWidth:0]	Register to hold input data.
m_axis_tdata_reg	logic [DataWidth:0]	Register to hold output data.
ready_out	logic	Signal indicating output ready status.
valid_out	logic	Signal indicating output valid status.

FIFO_AXIS_TOP

Generics

Generic name	Type	Value	Description
DataWidth		32	Width of the data bus.
Depth		2048	Depth of the FIFO buffer, i.e., the number of elements it can store.
ptrWidth		12	Width of the pointers used to address the FIFO buffer. It's calculated as the ceiling of the base-2 logarithm of the depth ($clog2(Depth)).

Ports

Port name	Direction	Type	Description
clk	input		
rst	input		
s_axis_tdata	input	[DataWidth-1 :0]	
s_axis_tvalid	input		
s_axis_tlast	input		

Port name	Direction	Type	Description
s_axis_tready	output		
m_axis_tdata	output	[DataWidth-1 :0]	
m_axis_tvalid	output		
m_axis_tlast	output		
m_axis_tready	input		

Signals

Name	Type	Description
m_axis_tdata_1	logic [DataWidth-1 :0]	
m_axis_tdata_2	logic [DataWidth-1 :0]	
m_axis_tdata_reg	logic [DataWidth-1 :0]	
m_axis_tvalid_1	logic	
m_axis_tlast_1	logic	
s_axis_tready_1	logic	
m_axis_tvalid_2	logic	

Name	Type	Description
m_axis_tlast_2	logic	
s_axis_tready_2	logic	
m_axis_tvalid_reg	logic	
m_axis_tlast_reg	logic	
s_axis_tready_reg	logic	
empty1	logic	
empty2	logic	
full1	logic	
full2	logic	

3.3 Module Code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_2/Assignment_3/fifo_axis.sv

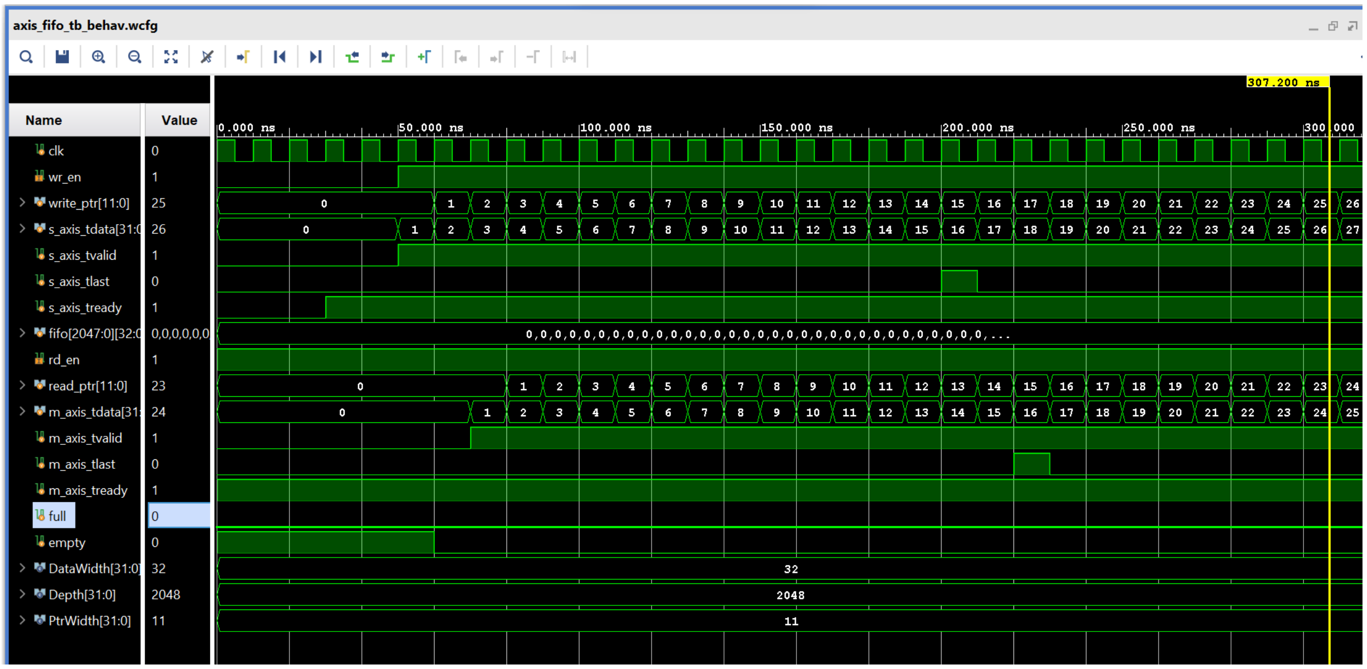
4. Testing Procedure

4.1.1 Testbench code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_2/Assignment_2/axis_mux_2X1_tb.v

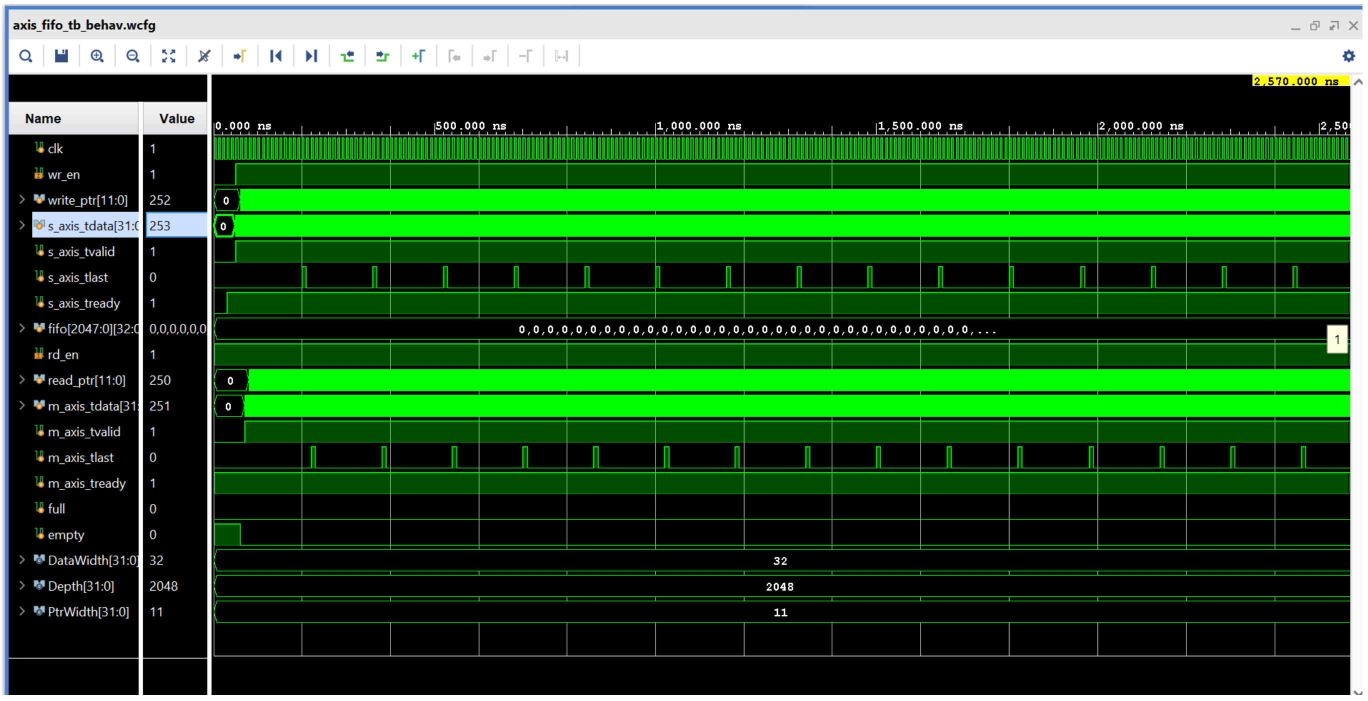
4.1.2 Test Cases

- **Write and Read at same time (Depth =16)**



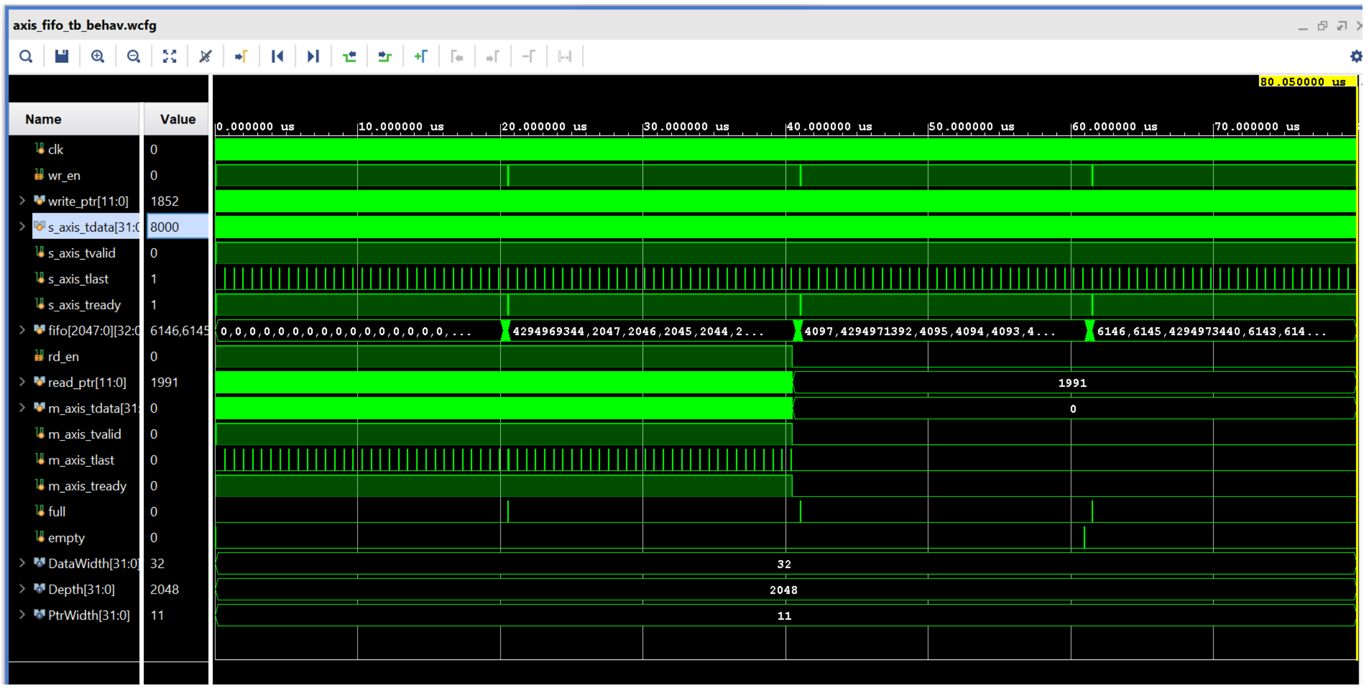
Name	Value
clk	0
wr_en	1
write_ptr[11:0]	25
s_axis_tdata[31:0]	26
s_axis_tvalid	1
s_axis_tlast	0
s_axis_tready	1
fifo[2047:0][32:0]	0,0,0,0,0,0
rd_en	1
read_ptr[11:0]	23
m_axis_tdata[31:0]	24
m_axis_tvalid	1
m_axis_tlast	0
m_axis_tready	1
full	0
empty	0
DataWidth[31:0]	32
Depth[31:0]	2048
PtrWidth[31:0]	11

- **Write and Read at same time with Last signal (Depth =16)**


axis_fifo_tb_behav.wcfg

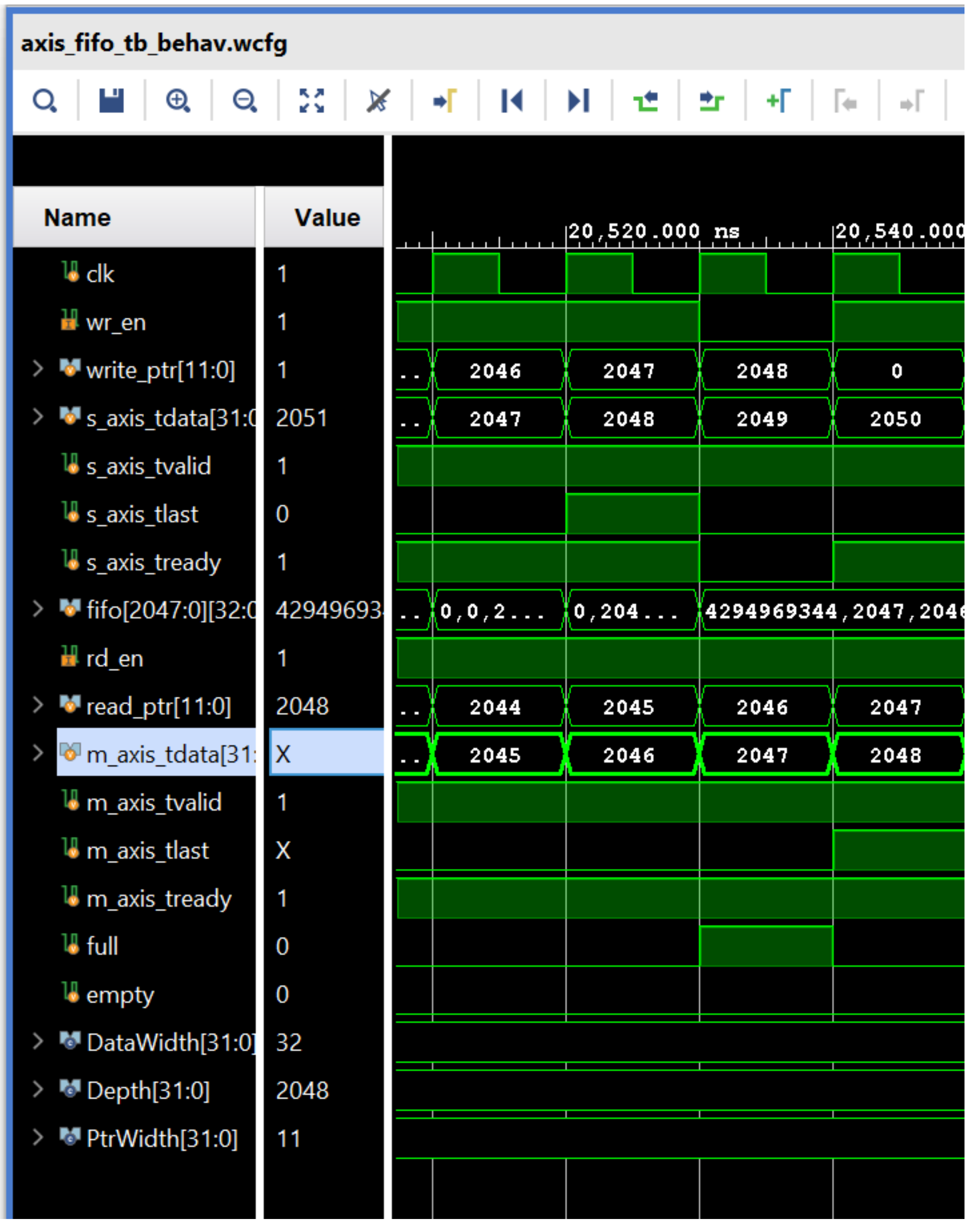
Name	Value
clk	1
wr_en	1
write_ptr[11:0]	252
s_axis_tdata[31:(	253
s_axis_tvalid	1
s_axis_tlast	0
s_axis_tready	1
fifo[2047:0][32:0	0,0,0,0,0,0
rd_en	1
read_ptr[11:0]	250
m_axis_tdata[31:	251
m_axis_tvalid	1
m_axis_tlast	0
m_axis_tready	1
full	0
empty	0
DataWidth[31:0]	32
Depth[31:0]	2048
PtrWidth[31:0]	11

- **Write and Read at same time with Full and Empty signals (Depth =16)**

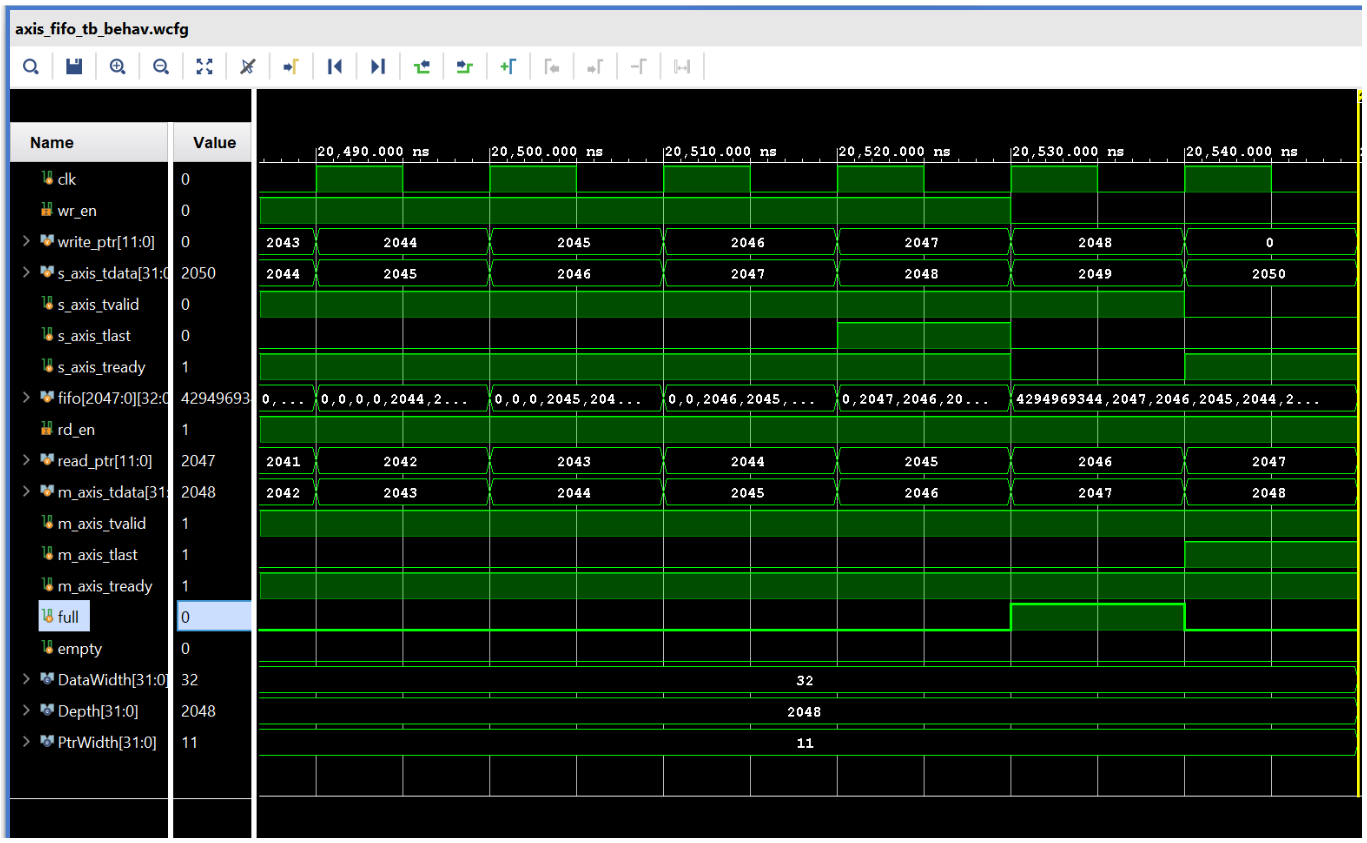

axis_fifo_tb_behav.wcfg

Name	Value
clk	0
wr_en	0
write_ptr[11:0]	1852
s_axis_tdata[31:(	8000
s_axis_tvalid	0
s_axis_tlast	1
s_axis_tready	1
fifo[2047:0][32:0	6146,6145
rd_en	0
read_ptr[11:0]	1991
m_axis_tdata[31:	0
m_axis_tvalid	0
m_axis_tlast	0
m_axis_tready	0
full	0
empty	0
DataWidth[31:0]	32
Depth[31:0]	2048
PtrWidth[31:0]	11

- **Full and Last signals (Depth =2048)**

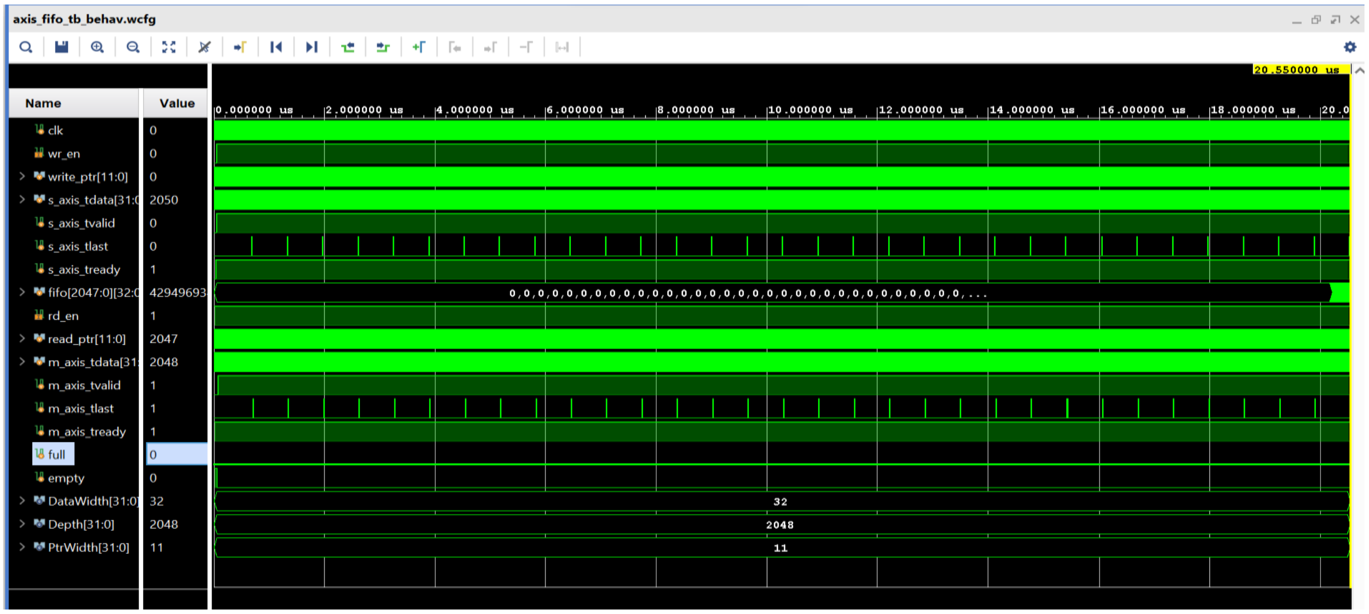


Name	Value				
clk	1				
wr_en	1				
write_ptr[11:0]	1	2046	2047	2048	0
s_axis_tdata[31:0]	2051	2047	2048	2049	2050
s_axis_tvalid	1				
s_axis_tlast	0				
s_axis_tready	1				
fifo[2047:0][32:0]	42949693	0,0,2...	0,204...	4294969344,2047,2046	
rd_en	1				
read_ptr[11:0]	2048	2044	2045	2046	2047
m_axis_tdata[31:0]	X	2045	2046	2047	2048
m_axis_tvalid	1				
m_axis_tlast	X				
m_axis_tready	1				
full	0				
empty	0				
DataWidth[31:0]	32				
Depth[31:0]	2048				
PtrWidth[31:0]	11				



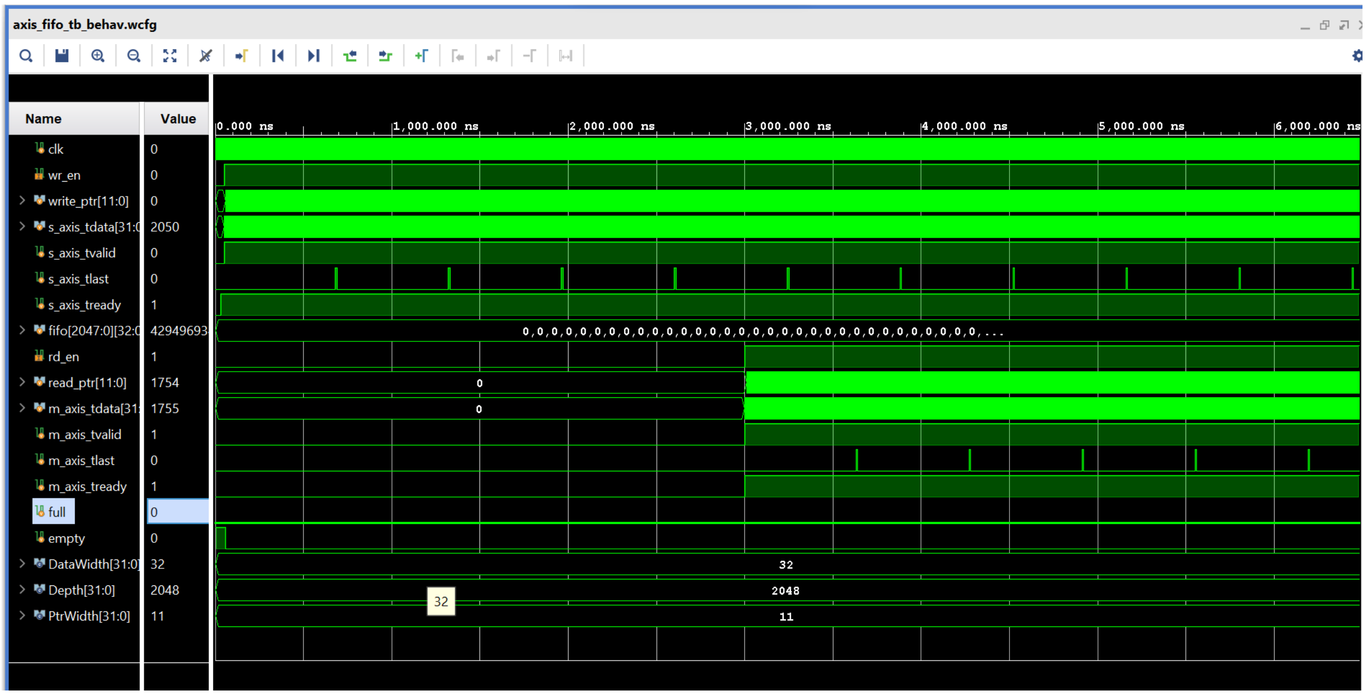
Name	Value
clk	0
wr_en	0
write_ptr[11:0]	0
s_axis_tdata[31:0]	2050
s_axis_tvalid	0
s_axis_tlast	0
s_axis_tready	1
fifo[2047:0][32:0]	42949693
rd_en	1
read_ptr[11:0]	2047
m_axis_tdata[31:	2048
m_axis_tvalid	1
m_axis_tlast	1
m_axis_tready	1
full	0
empty	0
DataWidth[31:0]	32
Depth[31:0]	2048
PtrWidth[31:0]	11

- **Write and Read at with Delay and Last signal (Depth =2048)**



Name	Value
clk	0
wr_en	0
write_ptr[11:0]	0
s_axis_tdata[31:0]	2050
s_axis_tvalid	0
s_axis_tlast	0
s_axis_tready	1
fifo[2047:0][32:0]	42949693
rd_en	1
read_ptr[11:0]	2047
m_axis_tdata[31:	2048
m_axis_tvalid	1
m_axis_tlast	1
m_axis_tready	1
full	0
empty	0
DataWidth[31:0]	32
Depth[31:0]	2048
PtrWidth[31:0]	11

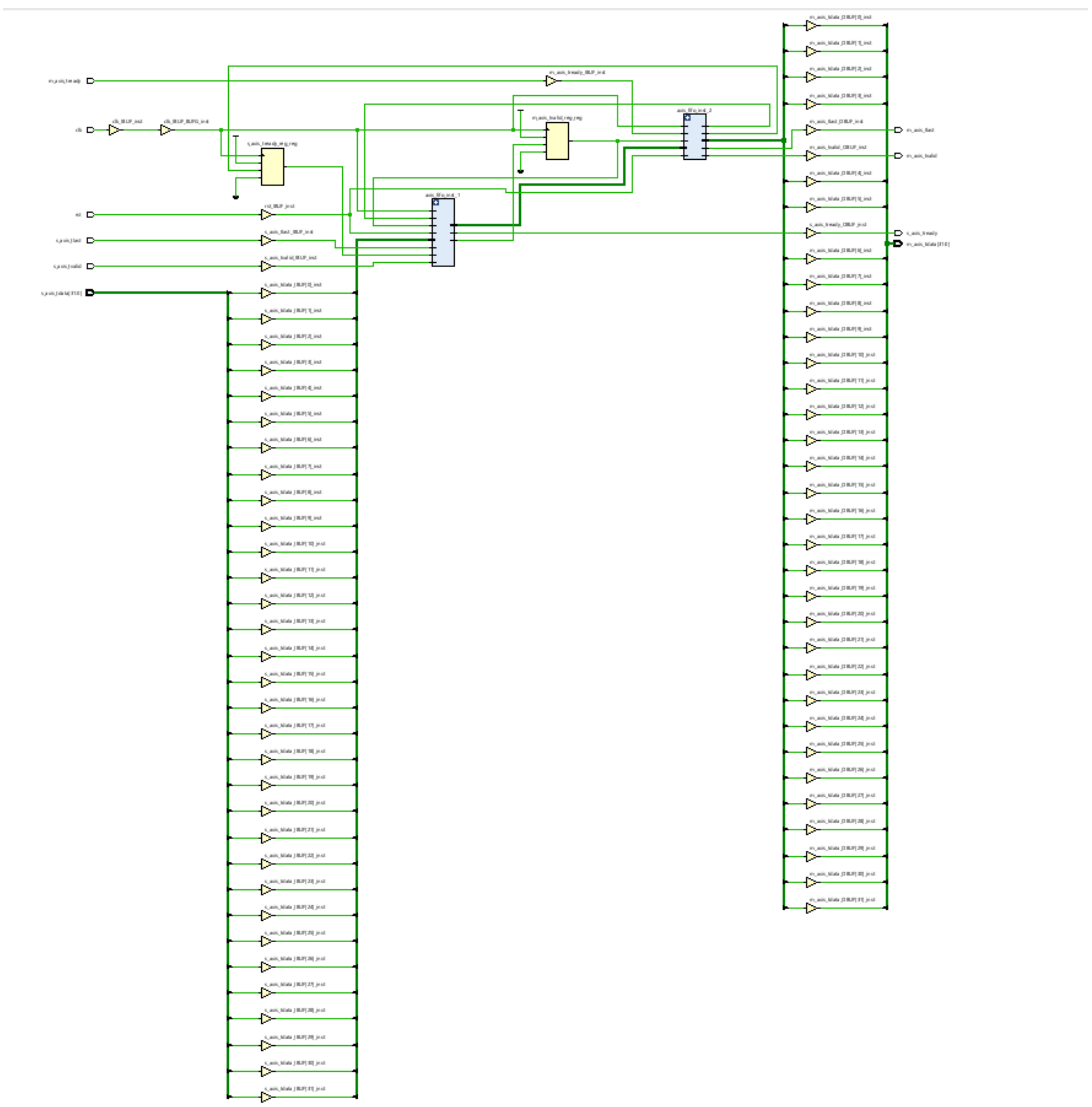
- **Write and Read at with Delay (Depth =2048)**

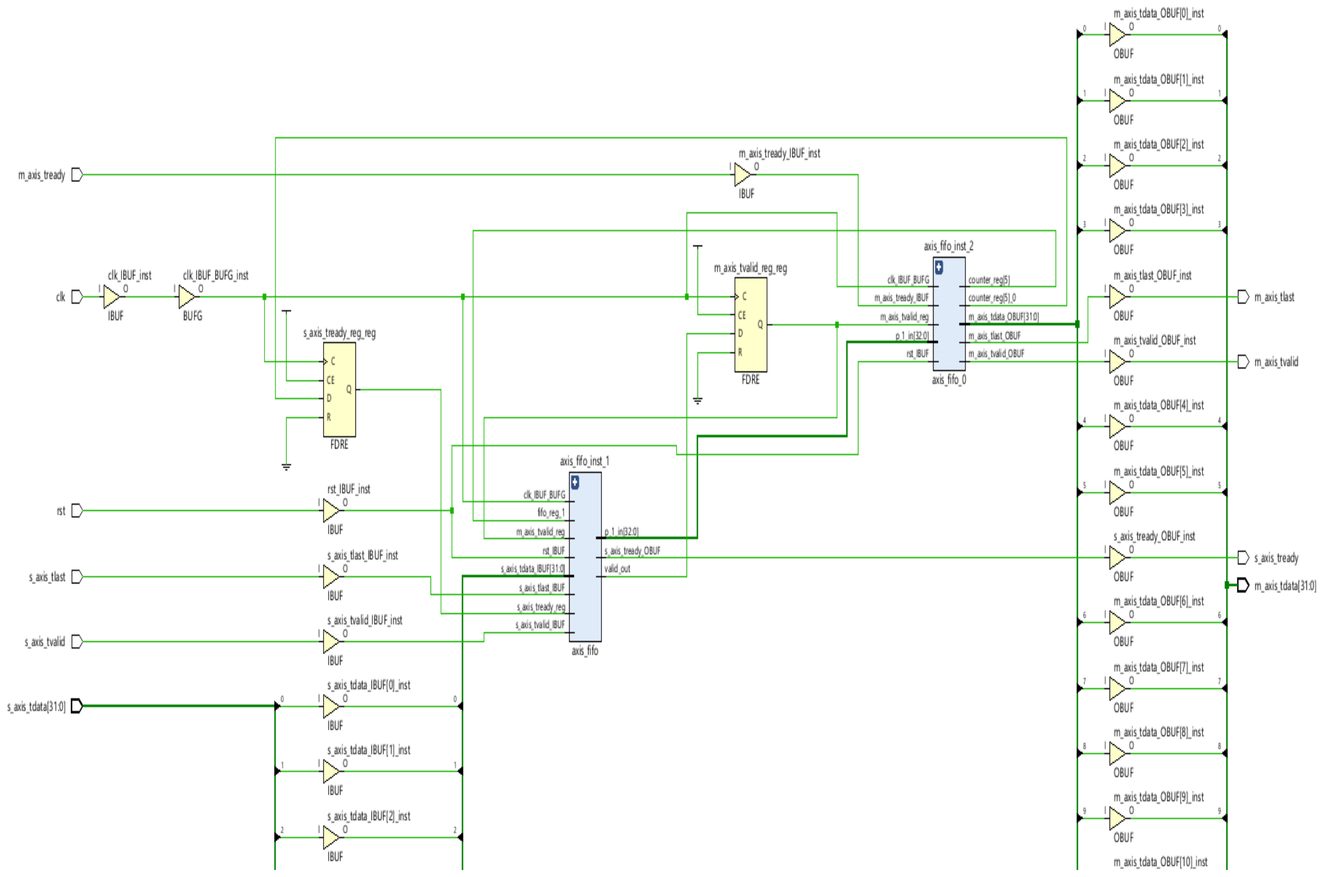


Name	Value
clk	0
wr_en	0
write_ptr[11:0]	0
s_axis_tdata[31:0]	2050
s_axis_tvalid	0
s_axis_tlast	0
s_axis_tready	1
fifo[2047:0][32:0]	42949693
rd_en	1
read_ptr[11:0]	1754
m_axis_tdata[31:0]	1755
m_axis_tvalid	1
m_axis_tlast	0
m_axis_tready	1
full	0
empty	0
DataWidth[31:0]	32
Depth[31:0]	2048
PtrWidth[31:0]	11

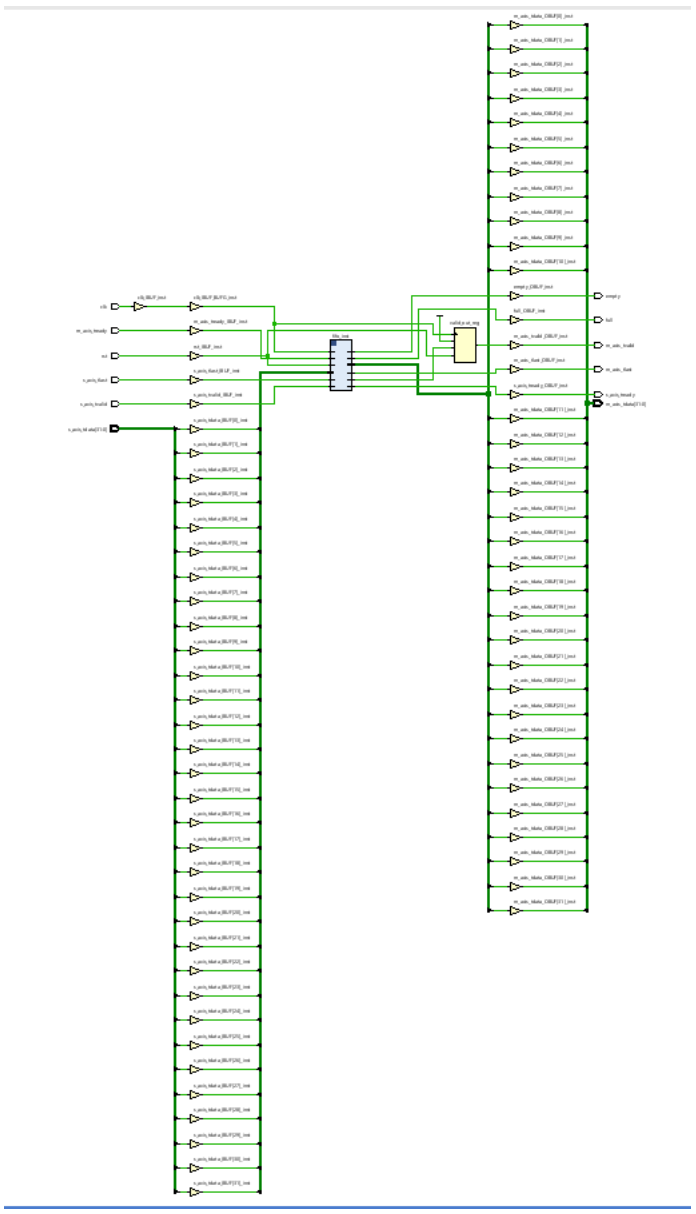
5. Schematic diagram

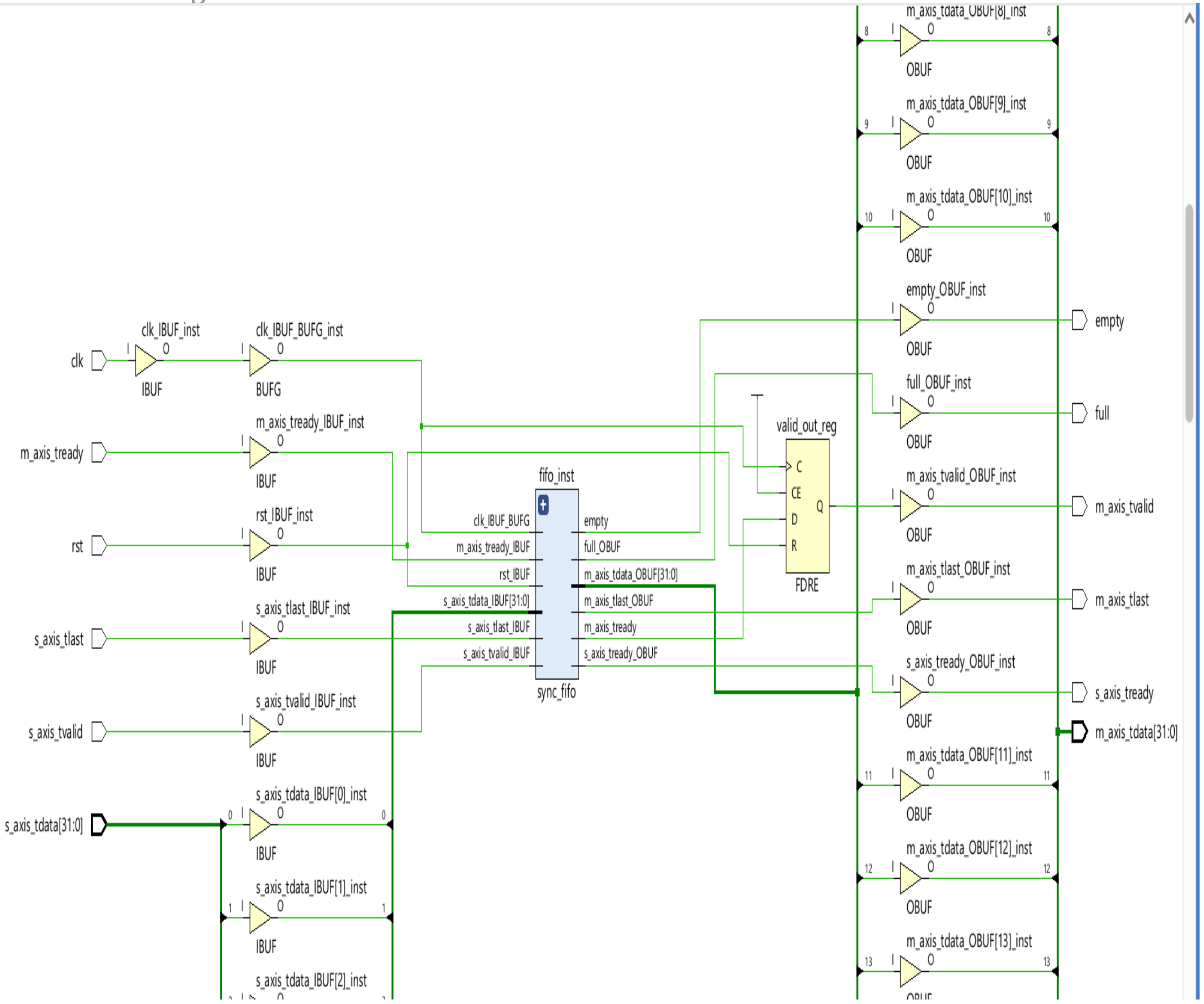
FIFO_AXIS_TOP



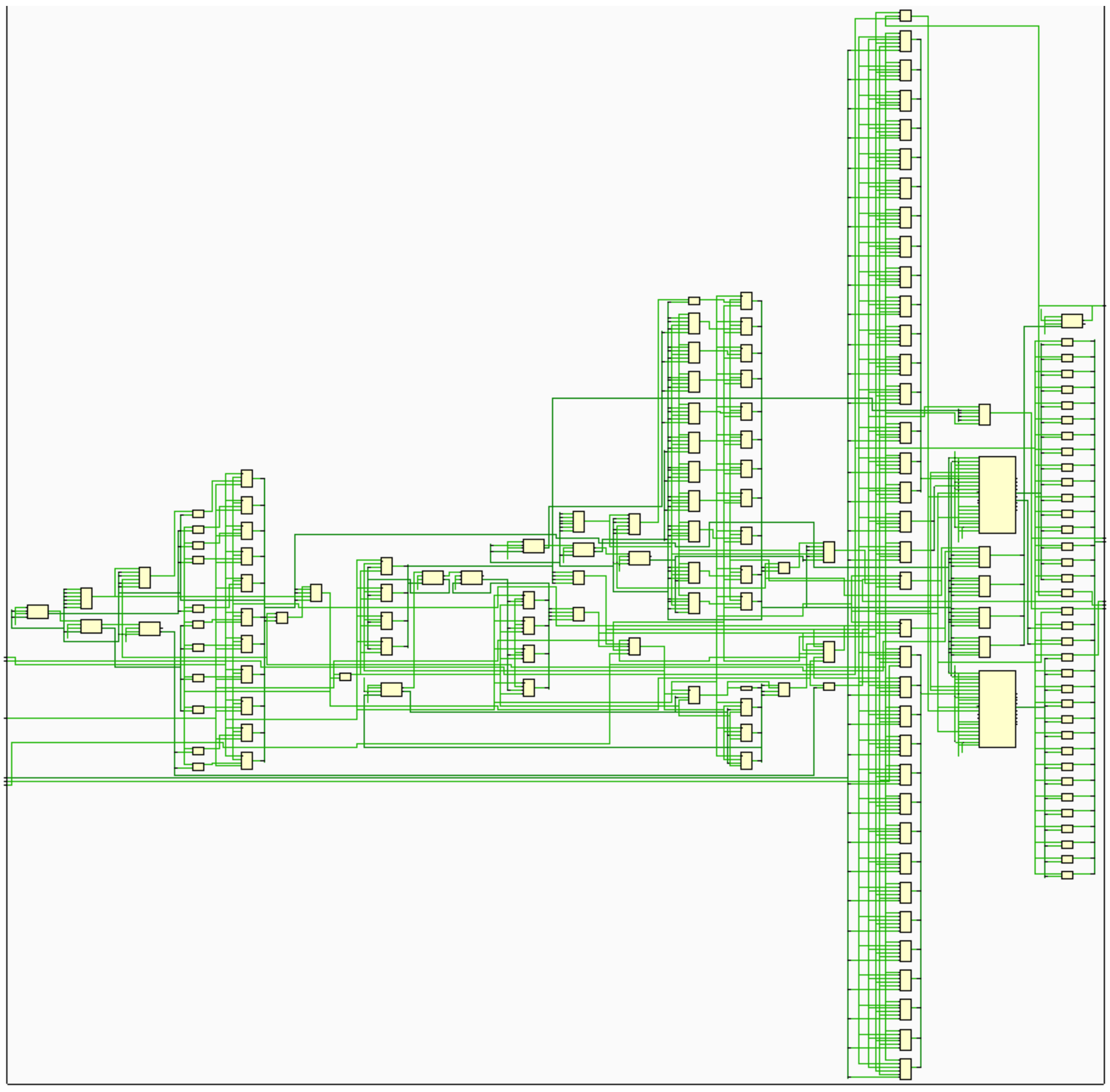

m_axis_tready
clk
rst
s_axis_tlast
s_axis_tvalid
s_axis_tdata[31:0]
clk_IBUF_inst
clk_IBUF_BUFG_inst
IBUF
BUFG
s_axis_tready_reg_reg
FDRE
m_axis_tready_IBUF_inst
m_axis_tvalid_reg_reg
axis_fifo_inst_2
axis_fifo_0
axis_fifo_inst_1
axis_fifo
rst_IBUF_inst
s_axis_tlast_IBUF_inst
s_axis_tvalid_IBUF_inst
s_axis_tdata_IBUF[0]_inst
s_axis_tdata_IBUF[1]_inst
s_axis_tdata_IBUF[2]_inst
m_axis_tdata_OBUF[0]_inst
m_axis_tdata_OBUF[1]_inst
m_axis_tdata_OBUF[2]_inst
m_axis_tdata_OBUF[3]_inst
m_axis_tlast_OBUF_inst
m_axis_tvalid_OBUF_inst
m_axis_tdata_OBUF[4]_inst
m_axis_tdata_OBUF[5]_inst
s_axis_tready_OBUF_inst
m_axis_tdata_OBUF[6]_inst
m_axis_tdata_OBUF[7]_inst
m_axis_tdata_OBUF[8]_inst
m_axis_tdata_OBUF[9]_inst
m_axis_tdata_OBUF[10]_inst
OBUF
m_axis_tlast
m_axis_tvalid
s_axis_tready
m_axis_tdata[31:0]

FIFO_AXIS



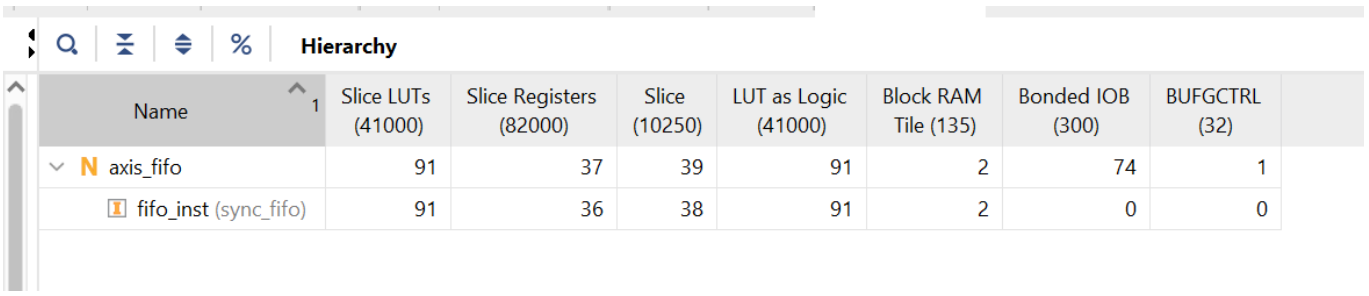

m_axis_tdata_OBUF[8]_inst
OBUF
m_axis_tdata_OBUF[9]_inst
OBUF
m_axis_tdata_OBUF[10]_inst
OBUF
empty_OBUF_inst
OBUF
empty
full_OBUF_inst
OBUF
full
clk_IBUF_inst
IBUF
clk_IBUF_BUFG_inst
BUFG
clk
m_axis_tready_IBUF_inst
IBUF
m_axis_tready
valid_out_reg
FDRE
m_axis_tvalid_OBUF_inst
OBUF
m_axis_tvalid
fifo_inst
sync_fifo
clk_IBUF_BUFG
m_axis_tready_IBUF
rst_IBUF
s_axis_tdata_IBUF[31:0]
s_axis_tlast_IBUF
s_axis_tvalid_IBUF
empty
full_OBUF
m_axis_tdata_OBUF[31:0]
m_axis_tlast_OBUF
m_axis_tready
s_axis_tready_OBUF
rst_IBUF_inst
IBUF
rst
s_axis_tlast_IBUF_inst
IBUF
s_axis_tlast
m_axis_tlast_OBUF_inst
OBUF
m_axis_tlast
s_axis_tready_OBUF_inst
OBUF
s_axis_tready
s_axis_tvalid_IBUF_inst
IBUF
s_axis_tvalid
m_axis_tdata[31:0]
s_axis_tdata_IBUF[0]_inst
IBUF
s_axis_tdata[31:0]
s_axis_tdata_IBUF[1]_inst
IBUF
s_axis_tdata_IBUF[2]_inst
m_axis_tdata_OBUF[11]_inst
OBUF
m_axis_tdata_OBUF[12]_inst
OBUF
m_axis_tdata_OBUF[13]_inst

FIFO



6. Resource Utilization

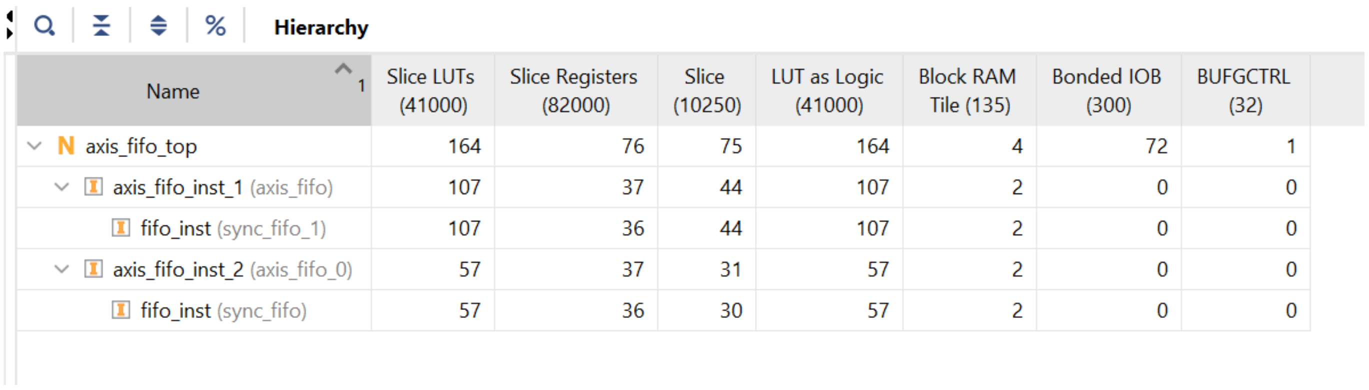
FIFO_AXIS



Hierarchy

Name	Slice LUTs (41000)	Slice Registers (82000)	Slice (10250)	LUT as Logic (41000)	Block RAM Tile (135)	Bonded IOB (300)	BUFGCTRL (32)
axis_fifo	91	37	39	91	2	74	1
fifo_inst (sync_fifo)	91	36	38	91	2	0	0

FIFO_AXIS



Hierarchy

Name	Slice LUTs (41000)	Slice Registers (82000)	Slice (10250)	LUT as Logic (41000)	Block RAM Tile (135)	Bonded IOB (300)	BUFGCTRL (32)
axis_fifo_top	164	76	75	164	4	72	1
axis_fifo_inst_1 (axis_fifo)	107	37	44	107	2	0	0
fifo_inst (sync_fifo_1)	107	36	44	107	2	0	0
axis_fifo_inst_2 (axis_fifo_0)	57	37	31	57	2	0	0
fifo_inst (sync_fifo)	57	36	30	57	2	0	0