RTL Designs with AXIS Interfaces

Objective

- Understanding the AXI stream interface protocol
- Learning the implementation insights of AXIS interfaces using Verilog/System Verilog

Introduction to the module and its functionality

➢ AXI stream interface

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.

Understanding the AXI stream interface protocol. 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:

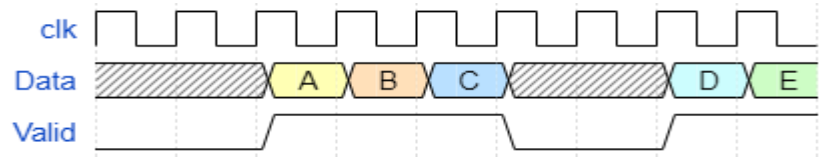
- Data
- Valid

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

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.

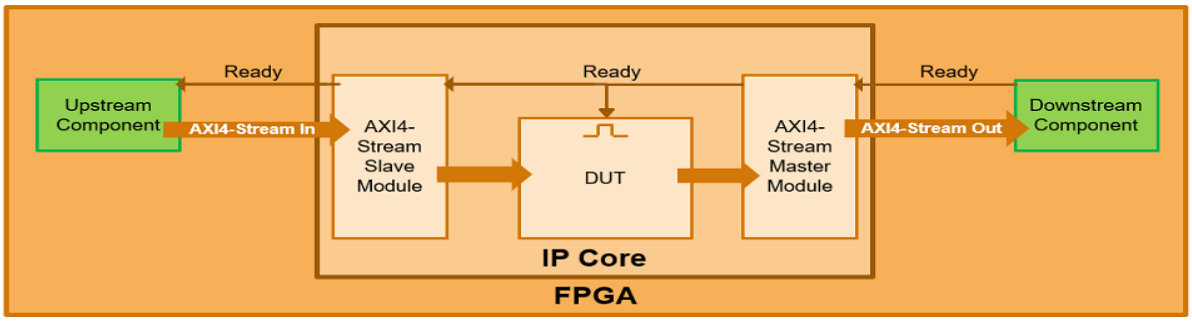


Ready Signal (Optional)

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.

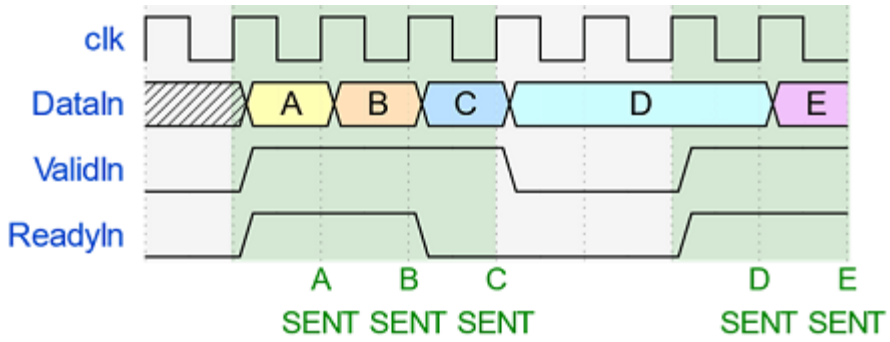
The absence of a back pressure logic might result in data being dropped. To avoid data loss and to apply back pressure on the slave interface or respond to back pressure from the master interface in your design:

- 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

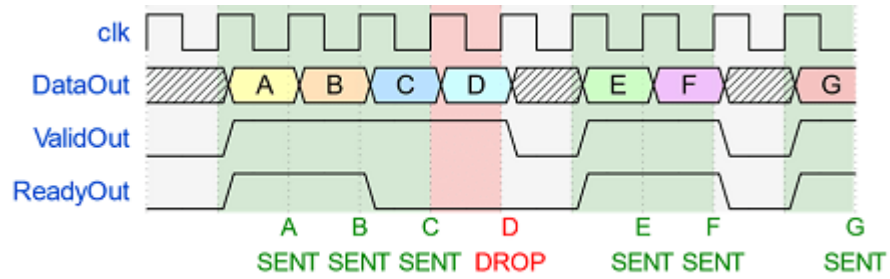
This image illustrates the timing relationship between the Data, Valid, and Ready signals according to the simplified streaming protocol. In this image clock is clk. The AXI4-Stream Slave module sends the DataIn and ValidIn signals after asserting the ReadyIn signal from the DUT. This is represented 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.

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

- 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 AXI stream interface

A computer consists of 3 basic components viz., a central processing unit (CPU), Memory to temporarily store results, and Storage to store data permanently. CPU in turn contains three main components namely a. The arithmetic logic unit, b. Control unit and c. Register Memory.

The Central Processing Unit (CPU) is the heart of a computer and it executes program codes, does arithmetic calculations, logical comparisons as instructed, and store the outcome in storage. It takes data and executable instructions from the main memory and processes them. While doing so it needs some working space to store intermediate results and special instructions and the stored values should be retrievable faster. Register does this function effectively.

Functions of Register

It is typically a tiny memory unit, not part of the main memory of the computer (Random Access Memory (RAM) or Read-only Memory (ROM)) resides in the CPU. They are positioned in the computer hierarchy a level above the main memory. The Control Unit of the computer takes the data from the disk storage (Secondary storage) and Program codes from the library and stores the relevant instruction and data in the main memory and instructs CPU to process it. CPU is the brain of the computer that processes the instruction and data and delivers the result.

The process may involve multiple steps and the results of the intermediate steps and other parameters like address, data will have to be stored in memory units for smooth continuity. The main memory of the computer cannot fulfill this requirement as the speed of storing and retrieval is not fast. Register memory fills the gap and provides faster storage and retrieval of the contents.

They store data, addresses, and instructions with a size of 32 bits to 64 bits, and the power of the CPU is determined by the number of Registers and its size. The big-sized can be split into smaller sized units to hold multiple data. One dimensional array or vector can be operated simultaneously using these registers and such processors are called vector processors. There are different types that are categorized by their contents, instructions, and uses. Some of the categories are accumulating values, data storage, address storage, next instruction, etc.

Registers can be grouped functionally under 2 groups:

Registers Accessible by Users: Contents in these can be altered by the instructions in the flow of program execution e.g. Data and address Register

Internal Registers: Used exclusively by CPU for internal operations and these are not accessible by instructions.

Importance of Registers

It plays a critical role in storing instructions, addresses, data, and results in tiny quickly retrievable memory units, and enhances the program execution speed. Though each has a specific function to perform, they are easily accessible to CPU, memory and other components of computers and the storing contents into and out of these registers are fast.

➢ Implementation Target

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.

Inputs:

clk: Clock signal used for synchronous operation. The module updates its internal state on the rising edge of this clock.

reset: Asynchronous reset signal. When asserted (high), it resets the module's internal state to a default condition.

s_axis_tdata: Input data bus of width 8 bits from the AXI Stream source.

s_axis_tvalid: Input valid signal from the AXI Stream source, indicating the availability of valid data on the input data bus.

s_axis_tlast: Input last signal from the AXI Stream source, indicating the last data beat of a packet.

Outputs:

s_axis_tready: Output ready signal to the AXI Stream source, indicating the readiness of the module to accept new data.

m_axis_tdata: Output data bus of width 8 bits to the AXI Stream sink, providing the stored data.

m_axis_tvalid: Output valid signal to the AXI Stream sink, indicating the availability of valid data on the output data bus.

m_axis_tlast: Output last signal to the AXI Stream sink, indicating the last data beat of a packet.

Internal Signals:

reg_data: Internal register to store the incoming data from the AXI Stream source.

valid_out: Internal signal to track the validity of the data stored in the register.

out_tlast: Internal signal to track the last beat of the packet.

ready: Internal signal indicating the readiness of the module to accept new data from the AXI Stream source.

enable: Internal signal enabling the storage of incoming data when valid and ready.
t_last: Internal signal indicating the last beat of the packet.

Behaviour:

On the rising edge of the clock signal (clk) or the rising edge of the asynchronous reset signal (reset), the module updates its internal state.

During reset (reset asserted), the internal register (reg_data) is set to all zeros (8'b0), and the output valid signal (valid_out) is de-asserted.

When not in reset, the module captures incoming data (s_axis_tdata) from the AXI Stream source if both the AXI Stream ready signal (m_axis_tready) and the internal valid signal (valid_out) are asserted.

The output data (m_axis_tdata) provided to the AXI Stream sink is the stored data in the internal register.

The output valid signal (m_axis_tvalid) is asserted when valid data is available, and the output ready signal (s_axis_tready) is asserted, indicating readiness to accept new data.

The output last signal (m_axis_tlast) reflects the internal last signal (out_tlast), indicating the last beat of the packet.

➢ VERIFICATION APPROCH

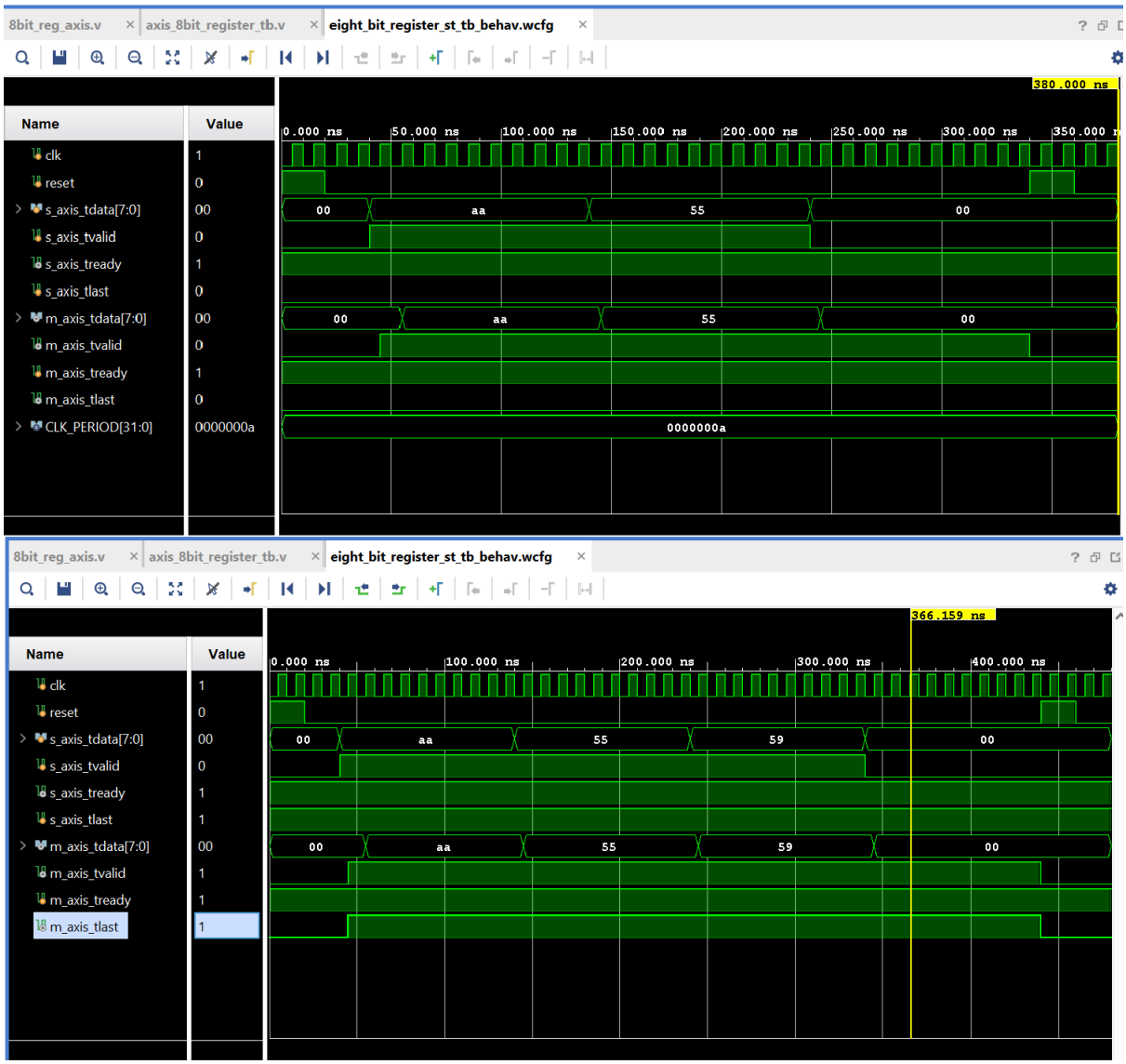
FPGA Resources occupancy

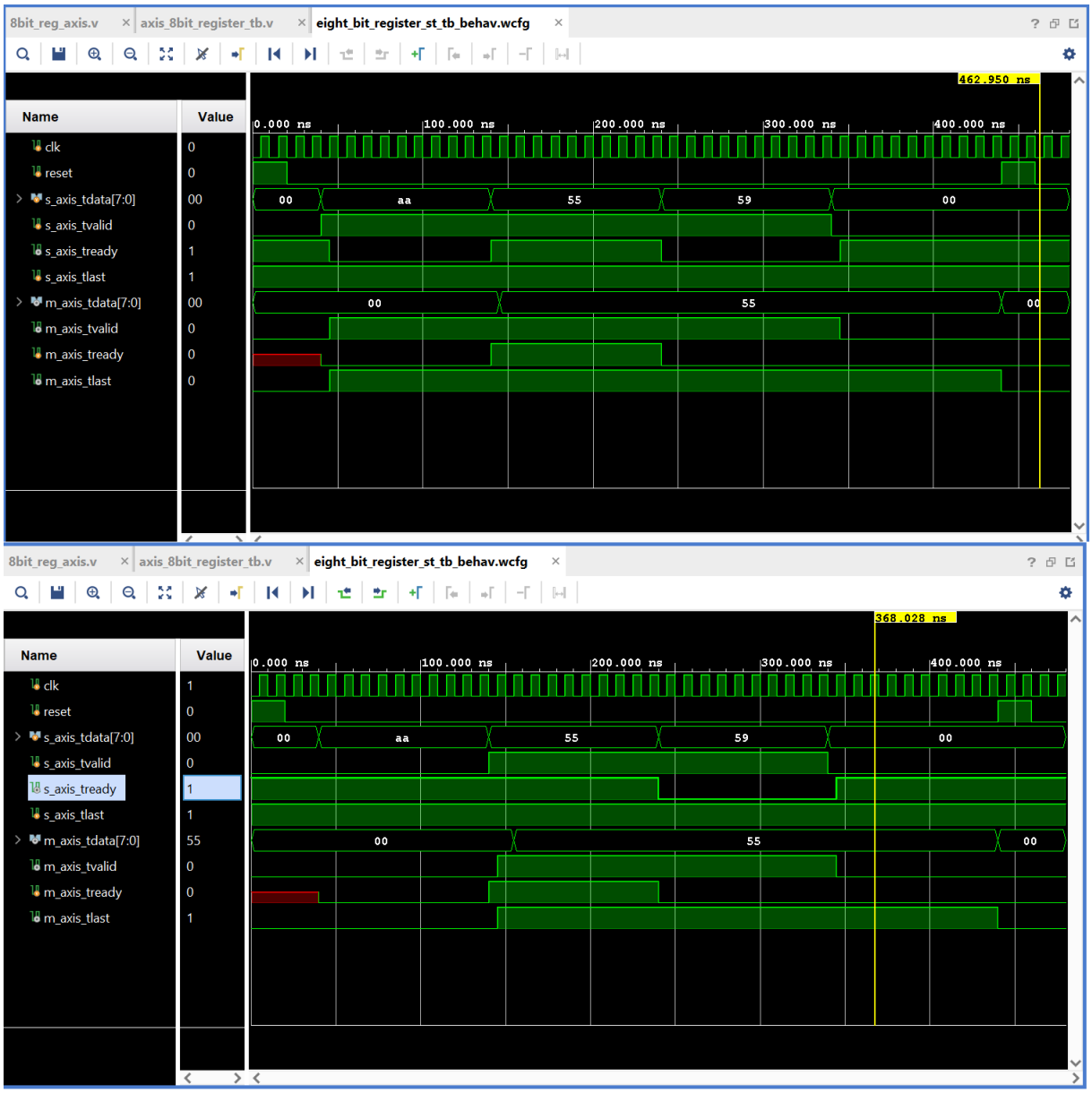
Resource	Utilization	Available	Utilization %
LUT	3	230400	0.01
FF	10	460800	0.01
IO	24	360	6.67
BUFG	1	544	0.18

Source code and TB link to the GIT repository

https://github.com/chinnapa5264/RTL_Training/tree/main/Module_2/Assignment_1

Simulation




8bit_reg_axis.v
axis_8bit_register_tb.v
eight_bit_register_st_tb_behav.wcfg
Name
Value
clk
reset
s_axis_tdata[7:0]
s_axis_tvalid
s_axis_tready
s_axis_tlast
m_axis_tdata[7:0]
m_axis_tvalid
m_axis_tready
m_axis_tlast
462.950 ns
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
00
aa
55
59
00
00
55
00
8bit_reg_axis.v
axis_8bit_register_tb.v
eight_bit_register_st_tb_behav.wcfg
Name
Value
clk
reset
s_axis_tdata[7:0]
s_axis_tvalid
s_axis_tready
s_axis_tlast
m_axis_tdata[7:0]
m_axis_tvalid
m_axis_tready
m_axis_tlast
368.028 ns
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
00
aa
55
59
00
00
55
00

Assignment - 2

- 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

➢ 2X1 MUX with AXI stream interface

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.

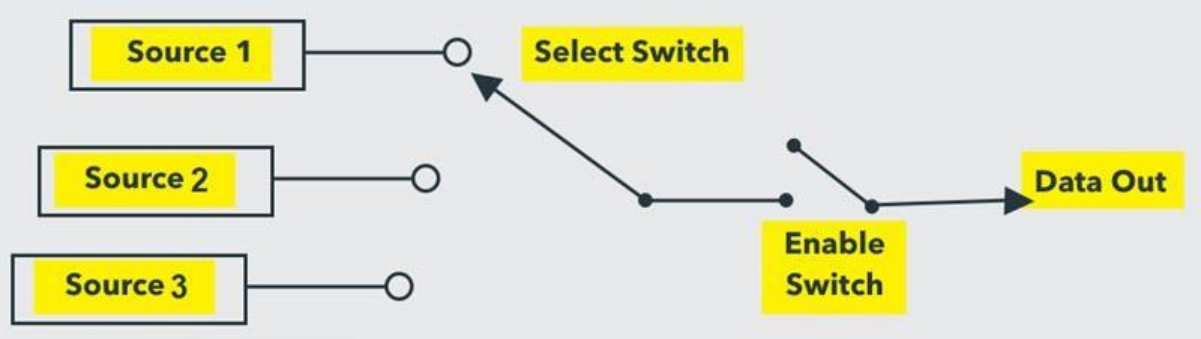
The device which is responsible for Multiplexing is known as Multiplexer. Multiplexers are used for both Analog and Digital signals. Let us focus on digital signals in this tutorial, to keep things simple. A multiplexer is the most frequently used combinational circuit and it is an important building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

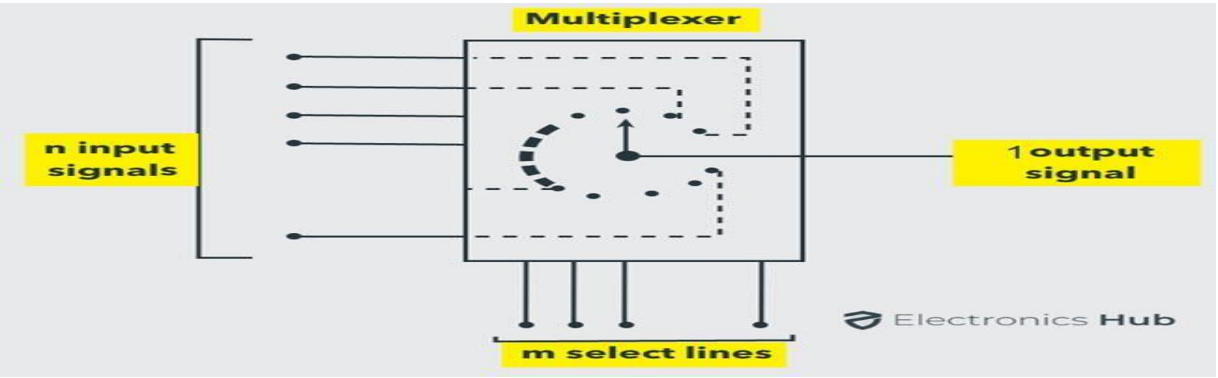
The multiplexer or MUX is a digital switch, also called as data selector. It is a Combinational Logic Circuit with more than one input line, one output line and more than one selects line. It accepts the binary information from several input lines or sources and depending on the set of select lines,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



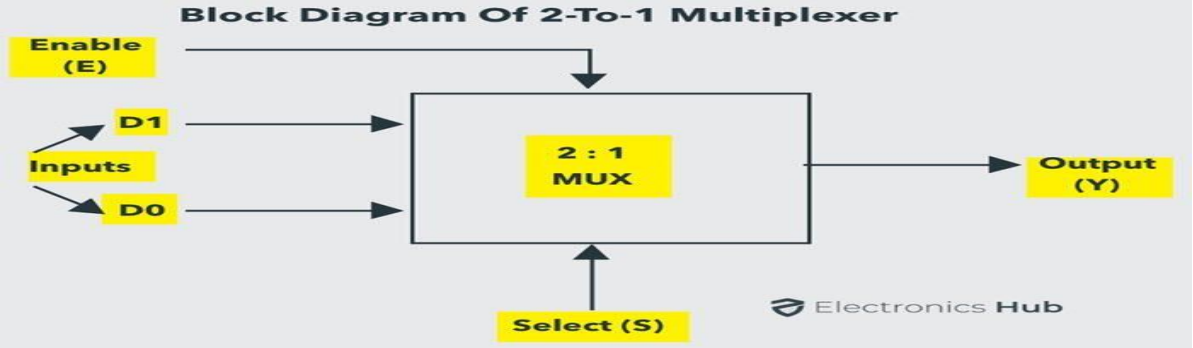
Generally, the number of data inputs to a multiplexer is a power of two such as 2, 4, 8, 16, etc. Some of the most frequently used multiplexers include 2-to-1, 4-to-1, 8-to-1 and 16-to-1 multiplexers.

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.

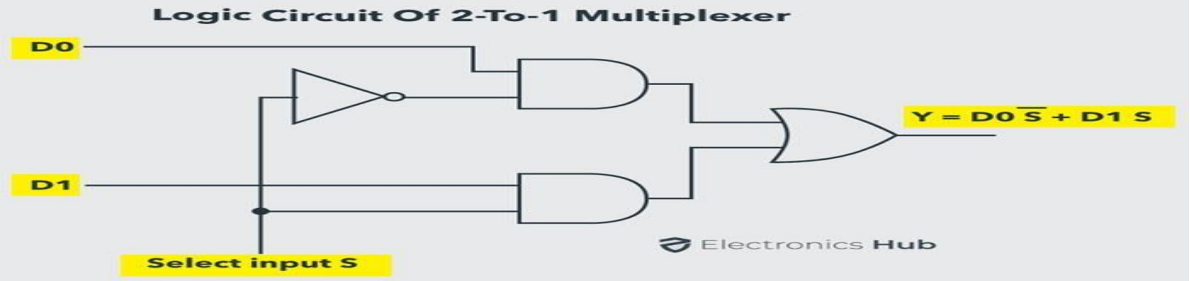
S	D0	D1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

'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:

$Y = SD0 + SD1$

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.

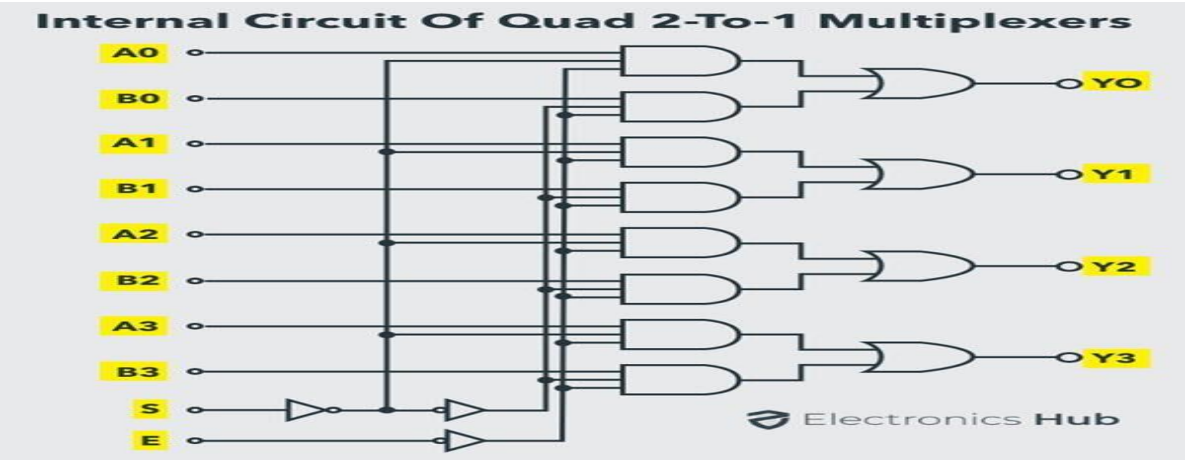
Similarly, when S=1, the output of the upper AND gate is zero, but the output of lower AND gate is D1. Therefore, the output of the OR gate is D1. Thus, the above given Boolean expression is satisfied by this circuit.



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.

➢ Implementation Target

The axis_mux module is an AXI4-Stream multiplexer designed to merge multiple AXI stream inputs into a single output stream. Here is a description of its functionality:

Inputs:

clk: Clock signal used for synchronization.

rst: Asynchronous reset signal to reset the multiplexer.

s_axis_tdata: Concatenated data inputs of the AXI4-Stream interfaces, representing the data to be multiplexed.

s_axis_tvalid: Valid signals of the AXI4-Stream interfaces, indicating when valid data is present on each input.

s_axis_tlast: Last signals of the AXI4-Stream interfaces, indicating the end of packets for each input.

enable: Control signal to enable the multiplexer.

select: Control signal to select the input port to be forwarded to the output.

Outputs:

s_axis_tready: Ready signals of the AXI4-Stream interfaces, indicating the readiness of each input to accept data.

m_axis_tdata: Data output of the multiplexer, representing the selected input data forwarded to the output.

m_axis_tvalid: Valid signal of the output AXI4-Stream interface, indicating when valid data is present on m_axis_tdata.

m_axis_tlast: Last signal of the output AXI4-Stream interface, indicating the end of packets.

Behaviour:

The module selects one of the input streams based on the select signal and forwards its data to the output.

It generates the s_axis_tready signals for each input based on the readiness of the downstream module to accept data and the current selection.

The module includes logic for frame detection based on the last signal of each input stream (s_axis_tlast) and manages the control signals accordingly.

It includes reset logic to initialize internal registers and control signals on reset (rst).

The multiplexer ensures that the output stream (m_axis_tdata, m_axis_tvalid, m_axis_tlast) reflects the selected input stream and handles flow control to prevent data loss.

Overall, the axis_mux module serves as a flexible AXI4-Stream multiplexer capable of merging multiple input streams into a single output stream based on control signals.

➢ VERIFICATION APPROCH

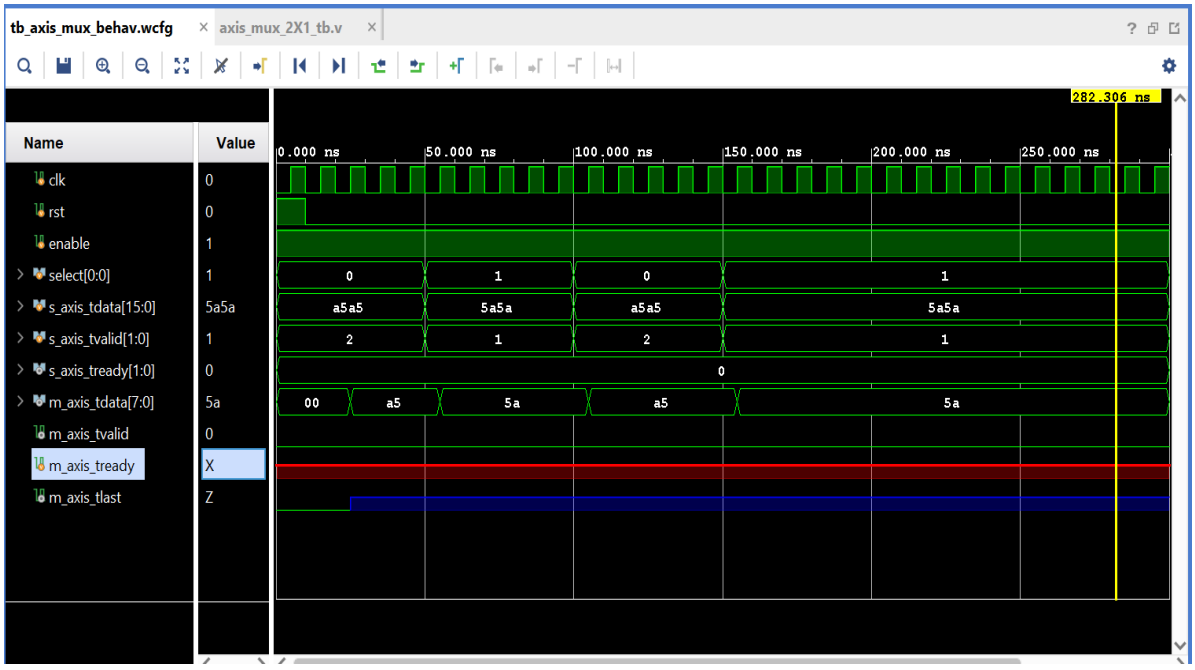
FPGA Resources occupancy

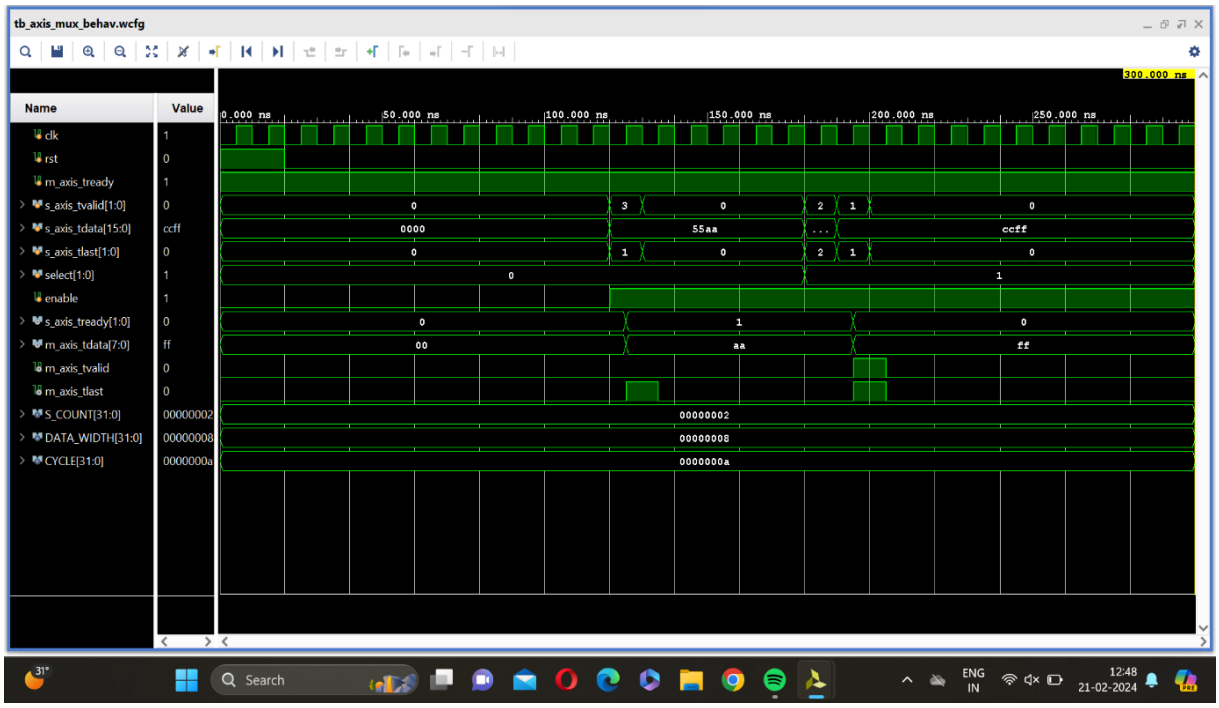
Resource	Utilization	Available	Utilization %
LUT	27	41000	0.07
FF	25	82000	0.03
IO	37	300	12.33
BUFG	1	32	3.13

Source code and TB link to the GIT repository

https://github.com/chinnapa5264/vivado_new/tree/main/axis_mux_2X1

Simulation




tb_axis_mux_behav.wcfg
Name
Value
clk
rst
m_axis_tready
s_axis_tvalid[1:0]
s_axis_tdata[15:0]
s_axis_tlast[1:0]
select[1:0]
enable
s_axis_tready[1:0]
m_axis_tdata[7:0]
m_axis_tvalid
m_axis_tlast
S_COUNT[31:0]
DATA_WIDTH[31:0]
CYCLE[31:0]
00000002
00000008
0000000a
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
Search
ENG
IN
12:48
21-02-2024

Assignment - 3

● Implement a 4096 depth FIFO using two 2048 FIFOs 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
● Observe the resource consumption of this implementation and make sure it is using **BRAM**
slices available on the FPGA

➢ FIFO with AXI stream interface

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.

➢ Implementation Target

Here is a detailed description of the provided Verilog code for the fifo_top module and the fifo_2048 submodule:

fifo_top Module:

Parameters:

DataWidth: Width of each data element in the FIFO (default: 32 bits).

Depth: Total depth of the FIFO (default: 4096 elements).

PtrWidth: Width of the pointers used to address memory locations in the FIFO (calculated based on Depth).

Inputs:

clk: Clock signal used for synchronization.

reset: Asynchronous reset signal for resetting the FIFO.
writeData: Data to be written into the FIFO.

writeDataValid: Valid signal indicating the presence of valid data to be written.

writeDataLast: Last signal indicating the end of a packet of data to be written.

readDataReady: Ready signal indicating the readiness of the consumer to accept data from the FIFO.

Outputs:

readData: Data read from the FIFO.

readDataValid: Valid signal indicating the presence of valid data read from the FIFO.

readDataLast: Last signal indicating the end of a packet of data read from the FIFO.

writeDataReady: Ready signal indicating the readiness of the FIFO to accept more data for writing.

full: Signal indicating whether the FIFO is full.

empty: Signal indicating whether the FIFO is empty.

Behaviour:

The fifo_top module instantiates two instances of the fifo_2048 module (fifo1 and fifo2) to create a larger FIFO with a depth of 4096 elements.

fifo1 is responsible for writing data into the FIFO, and fifo2 is responsible for reading data from the FIFO.

Data is written into fifo1, and then read from fifo2.

Control signals (writeDataReady and readDataValid) are managed to ensure proper flow of data between the two FIFOs.

Synchronization with the clock (clk) and reset (reset) signals is implemented to ensure reliable operation of the FIFO.

The fifo_top module provides a unified interface to the user while managing the interaction between the two FIFO modules internally.

fifo_2048 Module:

Parameters:

DataWidth: Width of each data element in the FIFO (default: 32 bits).

Depth: Total depth of the FIFO (default: 2048 elements).

PtrWidth: Width of the pointers used to address memory locations in the FIFO (calculated based on Depth).

Inputs:

clk: Clock signal used for synchronization.

reset: Asynchronous reset signal for resetting the FIFO.

writeData: Data to be written into the FIFO.

writeDataValid: Valid signal indicating the presence of valid data to be written.

writeDataLast: Last signal indicating the end of a packet of data to be written.

readDataReady: Ready signal indicating the readiness of the consumer to accept data from the FIFO.

Outputs:

readData: Data read from the FIFO.

readDataValid: Valid signal indicating the presence of valid data read from the FIFO.

readDataLast: Last signal indicating the end of a packet of data read from the FIFO.

full: Signal indicating whether the FIFO is full.

empty: Signal indicating whether the FIFO is empty.

Behaviour:

The fifo_2048 module implements a FIFO with a depth of 2048 elements.

It utilizes a memory array mem to store the data elements.

Pointers wrPtr and rdPtr are used to track the write and read positions in the FIFO, respectively.

Control signals (writeDataReady and readDataValid) are managed to control the flow of data into and out of the FIFO.

The module includes logic to handle the writeDataLast signal to identify the end of a packet of data and ensure synchronization.

Full and empty status of the FIFO is monitored and managed accordingly.

Internal logic ensures proper synchronization and operation of the FIFO with respect to clock and reset signals.

➢ VERIFICATION APPROCH

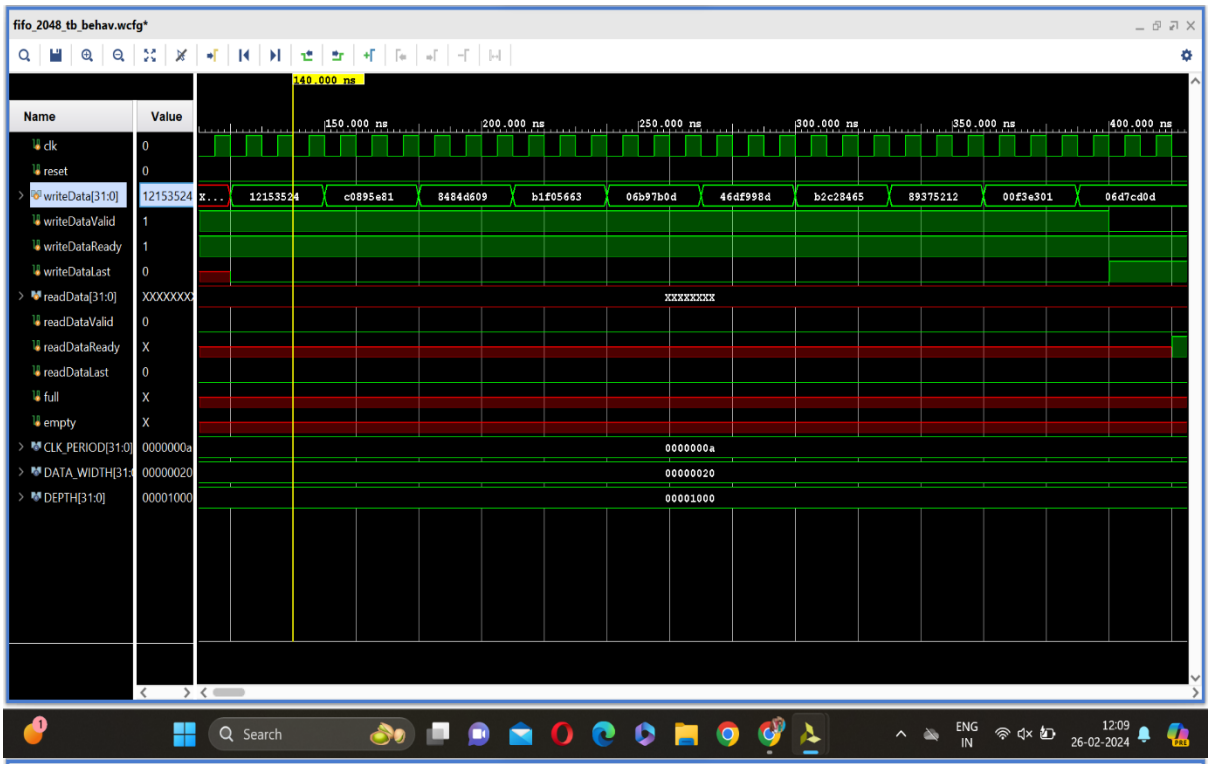
FPGA Resources occupancy

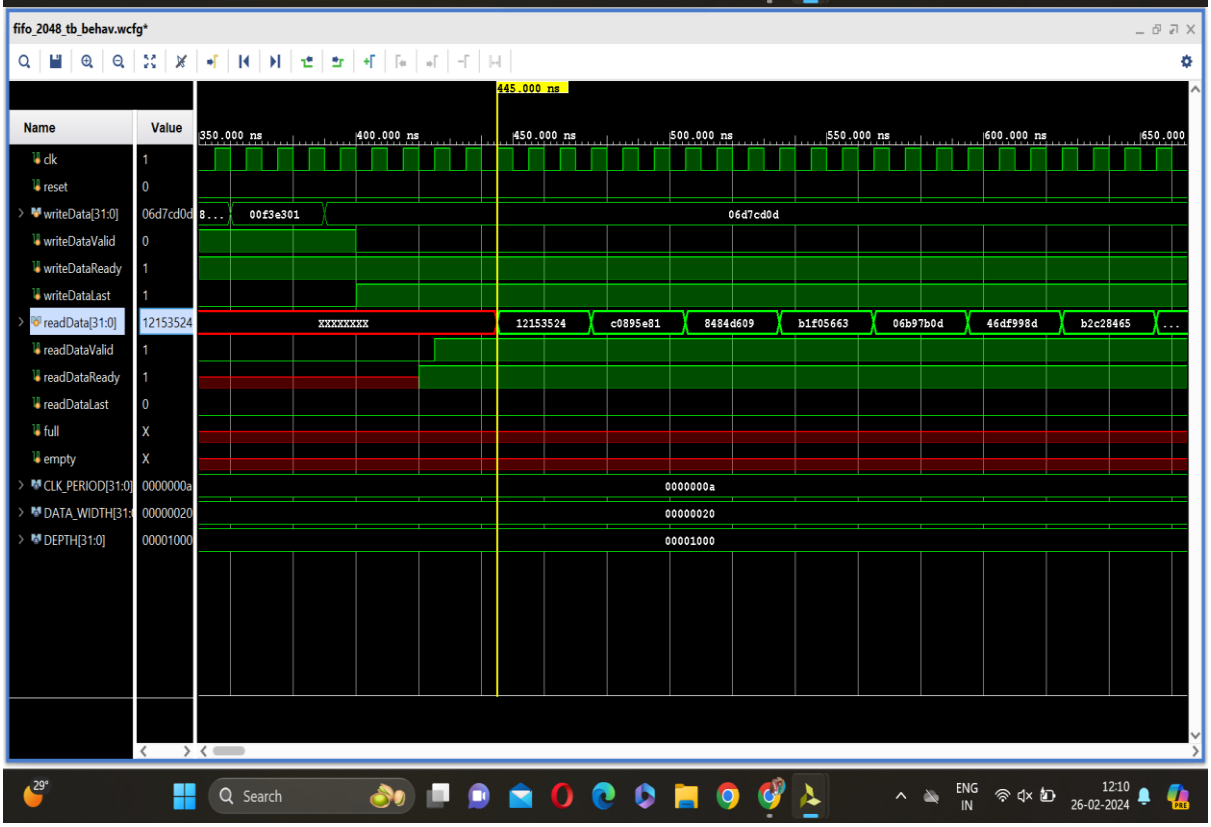
Resource	Utilization	Available	Utilization %
LUT	721	41000	1.76
LUTRAM	512	13400	3.82
FF	134	82000	0.16
BRAM	4	135	2.96
IO	74	300	24.67
BUFG	1	32	3.13

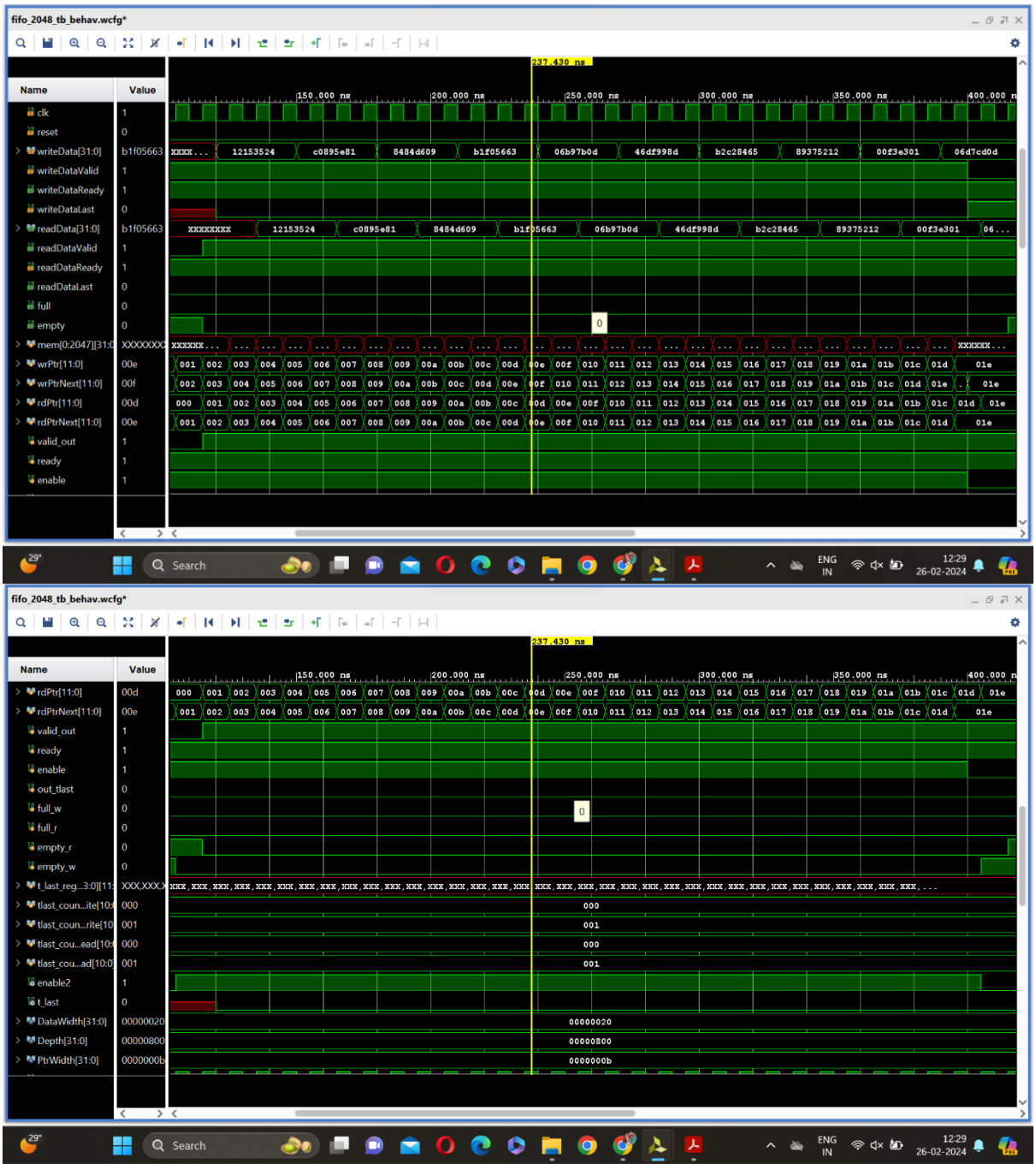
Source code and TB link to the GIT repository

https://github.com/chinnapa5264/vivado_new/tree/main/axis_fifo_sv

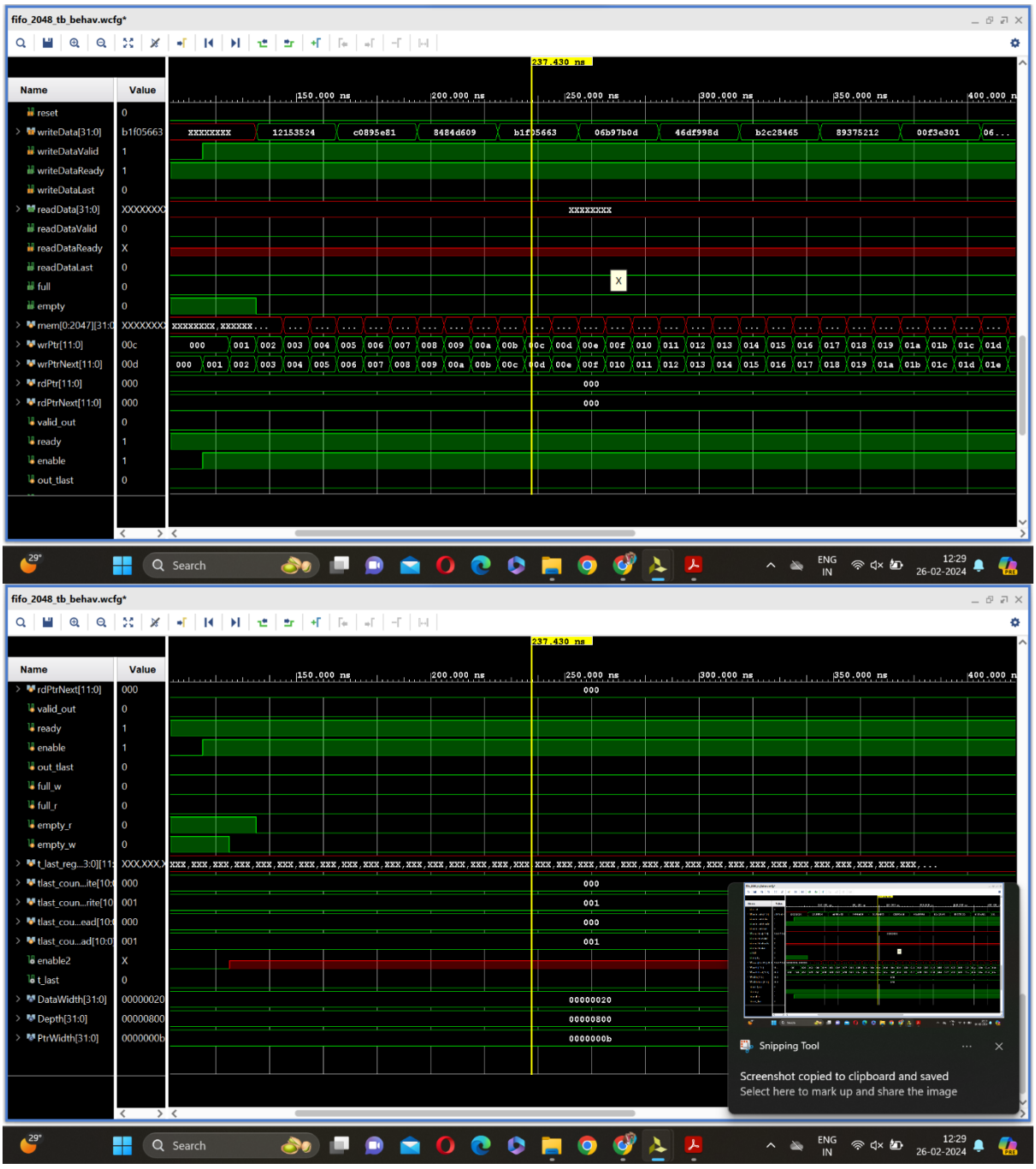
Simulation:

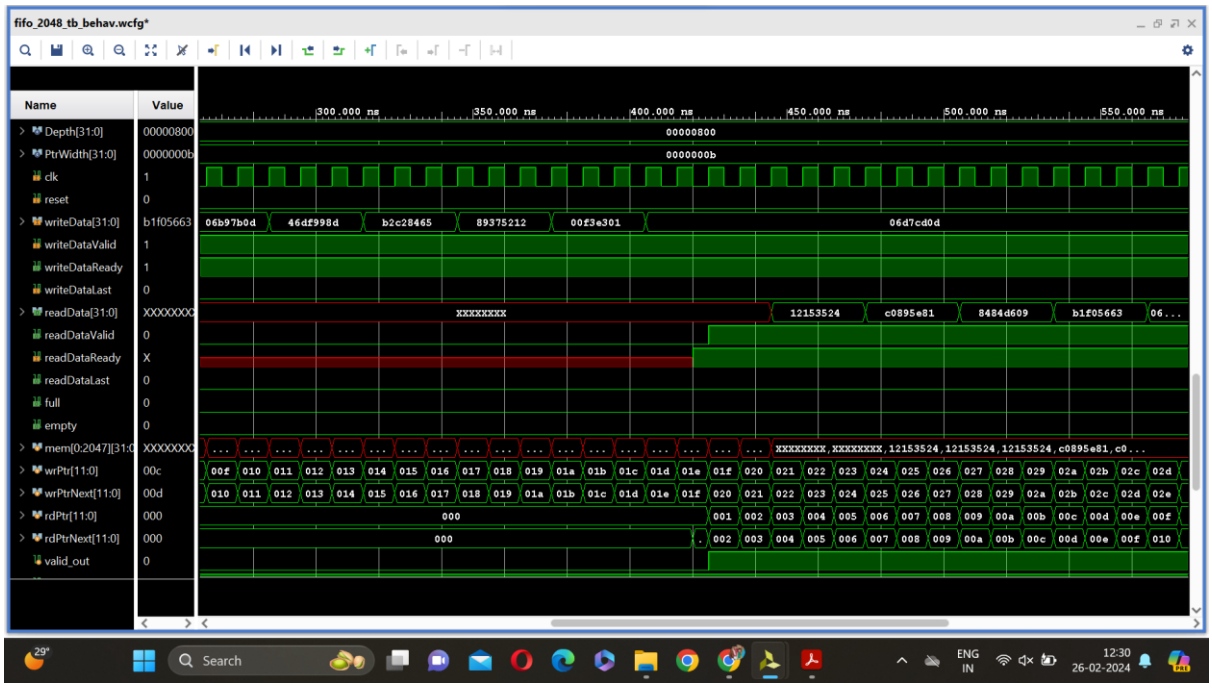

fifo_2048_tb_behav.wcfg*
140.000 ns
Name
Value
clk
reset
writeData[31:0]
writeDataValid
writeDataReady
writeDataLast
readData[31:0]
readDataValid
readDataReady
readDataLast
full
empty
CLK_PERIOD[31:0]
DATA_WIDTH[31:
DEPTH[31:0]
12153524
c0895e81
8484d609
b1f05663
06b97b0d
46df998d
b2c28465
89375212
00f3e301
06d7cd0d
0000000a
00000020
00001000
Search
ENG
IN
12:09
26-02-2024


fifo_2048_tb_behav.wcfg*
445.000 ns
Name
Value
clk
reset
writeData[31:0]
writeDataValid
writeDataReady
writeDataLast
readData[31:0]
readDataValid
readDataReady
readDataLast
full
empty
CLK_PERIOD[31:0]
DATA_WIDTH[31:
DEPTH[31:0]
00f3e301
06d7cd0d
12153524
c0895e81
8484d609
b1f05663
06b97b0d
46df998d
b2c28465
0000000a
00000020
00001000
Search
ENG
IN
12:10
26-02-2024


fifo_2048_tb_behav.wcfg*
Name
Value
clk
reset
writeData[31:0]
writeDataValid
writeDataReady
writeDataLast
readData[31:0]
readDataValid
readDataReady
readDataLast
full
empty
mem[0:2047][31:0]
wrPtr[11:0]
wrPtrNext[11:0]
rdPtr[11:0]
rdPtrNext[11:0]
valid_out
ready
enable
237.430 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 n
12153524
c0895e81
8484d609
b1f05663
06b97b0d
46df998d
b2c28465
89375212
00f3e301
06d7cd0d
Search
ENG
IN
12:29
26-02-2024

fifo_2048_tb_behav.wcfg*
Name
Value
rdPtr[11:0]
rdPtrNext[11:0]
valid_out
ready
enable
out_tlast
full_w
full_r
empty_r
empty_w
t_last_reg...3:0][11:
tlast_coun...ite[10:
tlast_coun...rite[10
tlast_cou...ead[10:
tlast_cou...ad[10:0]
enable2
t_last
DataWidth[31:0]
Depth[31:0]
PtrWidth[31:0]
00000020
00000800
0000000b
237.430 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 n
Search
ENG
IN
12:29
26-02-2024


fifo_2048_tb_behav.wcfg*
237.430 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 n
Name
Value
reset
writeData[31:0]
writeDataValid
writeDataReady
writeDataLast
readData[31:0]
readDataValid
readDataReady
readDataLast
full
empty
mem[0:2047][31:0]
wrPtr[11:0]
wrPtrNext[11:0]
rdPtr[11:0]
rdPtrNext[11:0]
valid_out
ready
enable
out_tlast
Search
ENG
IN
12:29
26-02-2024
fifo_2048_tb_behav.wcfg*
237.430 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 n
Name
Value
rdPtrNext[11:0]
valid_out
ready
enable
out_tlast
full_w
full_r
empty_r
empty_w
t_last_reg...3:0][11:
tlast_coun...ite[10:
tlast_coun...rite[10
tlast_cou...ead[10:
tlast_cou...ad[10:0]
enable2
t_last
DataWidth[31:0]
Depth[31:0]
PtrWidth[31:0]
Snipping Tool
Screenshot copied to clipboard and saved
Select here to mark up and share the image
Search
ENG
IN
12:29
26-02-2024



References

https://semiconductorclub.com/what-is-fifo-synchronous-fifo-asynchronous-fifo/

https://in.mathworks.com/help/hdlcoder/ug/model-design-for-axi4-stream-interface-generation.html#mw_c86ee525-dac6-4267-b4c0-85660a126c49