FSM based RTL Designs

Contents

Revision History

Version	Date (DD/MM/YY)	Description	Author(s)	Reviewer(s)
1	1-05-2024		Alavala Chinnapa Reddy	

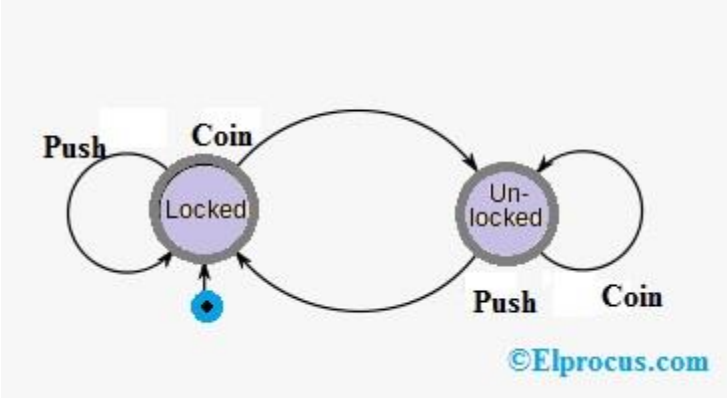
Objective

- Understanding the concept of fixed-point arithmetic and use cases.
- Learning the implementation insights of fixed-point arithmetic.

What is an FSM

The finite state machines (FSMs) are significant for understanding the decision-making logic as well as control the digital systems. In the FSM, the outputs, as well as the next state, are a present state and the input function. This means that the selection of the next state mainly depends on the input value and strength lead to more compound system performance. As in sequential logic, we require the past inputs history for deciding the output. Therefore, FSM proves very cooperative in understanding sequential logic roles. Basically, there are two methods for arranging a sequential logic design namely mealy machine as well as more machine.

The definition of a finite state machine is, the term finite state machine (FSM) is also known as finite state automation. FSM is a calculation model that can be executed with the help of hardware otherwise software. This is used for creating sequential logic as well as a few computer programs. FSMs are used to solve the problems in fields like mathematics, games, linguistics, and artificial intelligence. In a system where specific inputs can cause specific changes in state that can be signified with the help of FSMs.



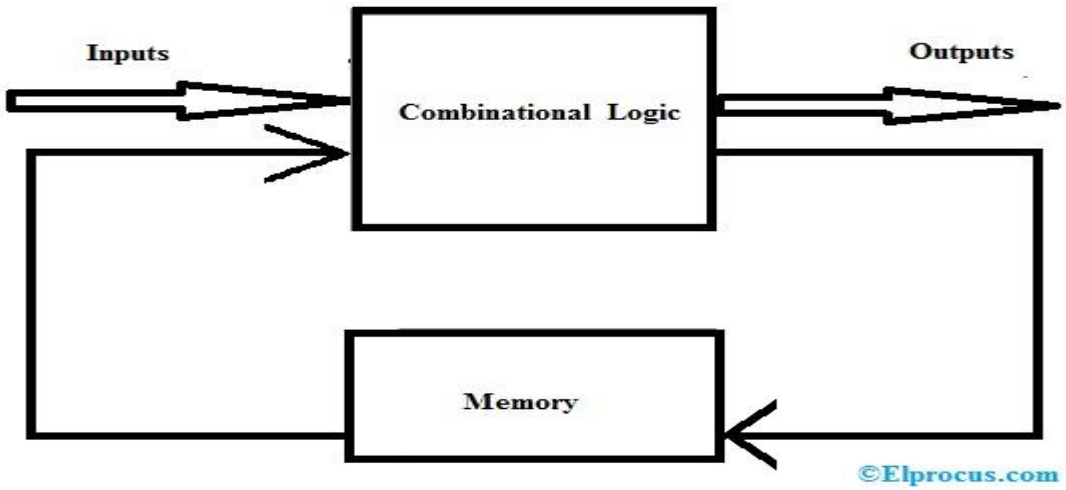
This finite state machine diagram explains the various conditions of a turnstile. Whenever placing a coin into a turnstile will unbolt it, and after the turnstile has been pressed, it bolts gain. Placing a coin into an unbolted turnstile, otherwise pressing against a bolted turnstile will not alter its state.

1. Types of Finite State Machine

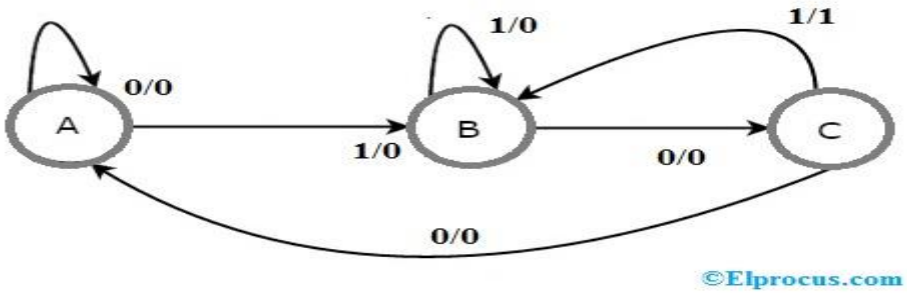
The finite state machines are classified into two types such as Mealy state machine and Moore state machine.

Mealy State Machine

When the outputs depend on the current inputs as well as states, then the FSM can be named to be a mealy state machine. The following diagram is the mealy state machine block diagram. The mealy state machine block diagram consists of two parts namely combinational logic as well as memory. The memory in the machine can be used to provide some of the previous outputs as combinational logic inputs.



Based on the current inputs as well as states, this machine can produce outputs. Thus, the outputs can be suitable only at positive otherwise negative of the CLK signal. The mealy state machine's state diagram is shown below.

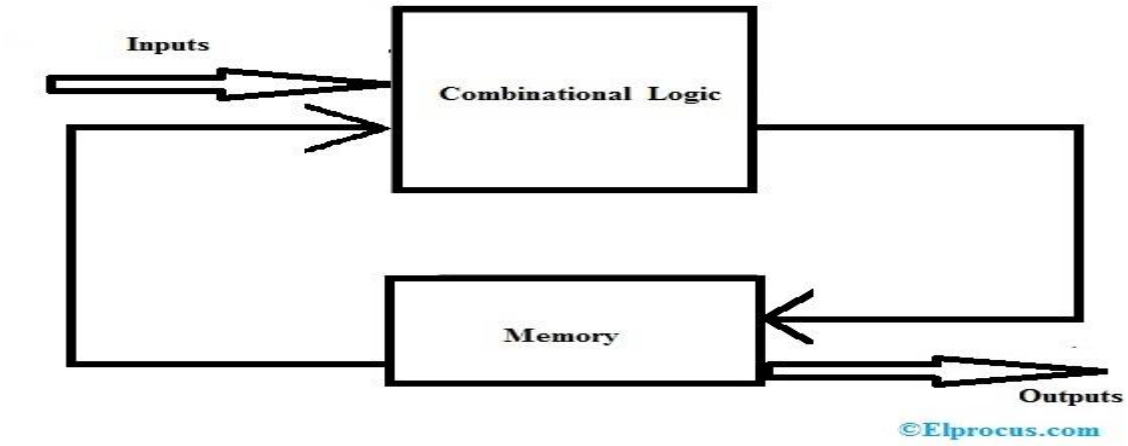


The state diagram of mealy state machine mainly includes three states namely A, B, and C. These three states are tagged within the circles as well as every circle communicates with one state. Conversions among these three states are signified by directed lines. In the above diagram, the inputs and outputs are denoted with 0/0, 1/0, and 1/1. Based on the input value, there are two conversions from every state.

Generally, the number of required states in the mealy machine is below or equivalent to the number of required states in Moore state machine. There is an equal Moore state machine for every Mealy state machine. As a result, based on the necessity we can employ one of them.

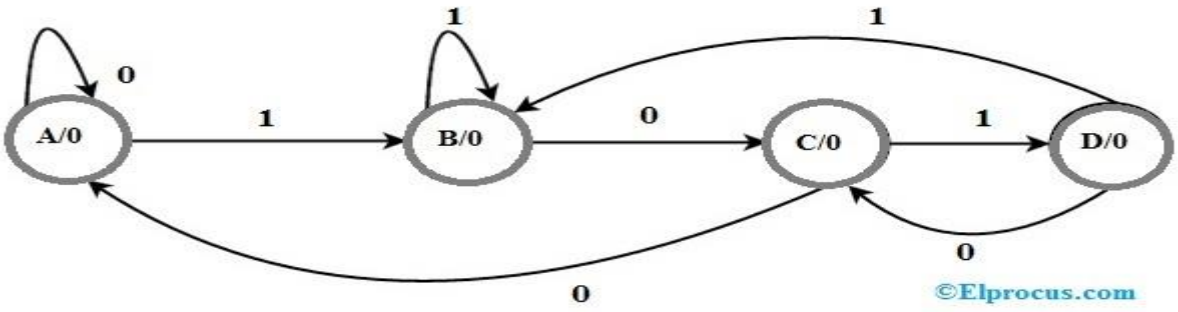
Moore State Machine

When the outputs depend on current states then the FSM can be named as Moore state machine. The Moore state machine's block diagram is shown below. The Moore state machine block diagram consists of two parts namely combinational logic as well as memory.



In this case, the current inputs, as well as current states, will decide the next states. Thus, depending on further states, this machine will generate the outputs. So, the outputs of this will be applicable simply after the conversion of the state.

The Moore state machine state diagram is shown below. In the above state, the diagram includes four states like a mealy state machine namely A, B, C, and D. the four states as well as individual outputs are placed in the circles.



In the above figure, there are four states, namely A, B, C & D. These states and the respective outputs are labelled inside the circles. Here, simply the input worth is marked on every conversion. In the above figure includes two conversions from every state depending on the input value.

Generally, the number of required states in this machine is greater than otherwise equivalent to the required number of states in the mealy state machine

Generally, the number of required states in this machine is more than otherwise equivalent to the required states in MSM (Mealy state machine). For every Moore state machine, there is a corresponding Mealy state machine. Consequently, depending on the necessity we can utilize one of them.

There is an equal mealy state machine for every Moore state machine. As a result, based on the necessity we can employ one of them.

2. Finite State Machine Applications

- The finite state machine applications mainly include the following.
- FSMs are used in games; they are most recognized for being utilized in artificial intelligence, and however, they are also frequent in executions of navigating parsing text, input handling of the customer, as well as network protocols.
- These are restricted in computational power; they have the good quality of being comparatively simple to recognize. So, they are frequently used by software developers as well as system designers for summarizing the performance of a difficult system.
- The finite state machines are applicable in vending machines, video games, traffic lights, controllers in CPU, text parsing, analysis of protocol, recognition of speech, language processing, etc.

3. Advantages of Finite State Machine

- The advantages of Finite State Machine include the following.
- Finite state machines are flexible
- Easy to move from a significant abstract to a code execution
- Low processor overhead
- Easy determination of reachability of a state

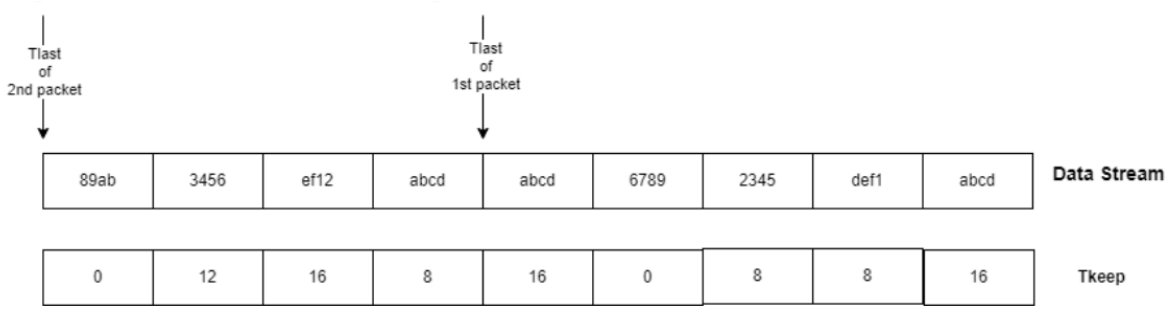
4. Disadvantages of Finite State Machine

- The disadvantages of the finite state machine include the following
- The expected character of deterministic finite state machines can be not needed in some areas like computer games
- The implementation of huge systems using FSM is hard for managing without any idea of design.
- Not applicable for all domains
- The orders of state conversions are inflexible.

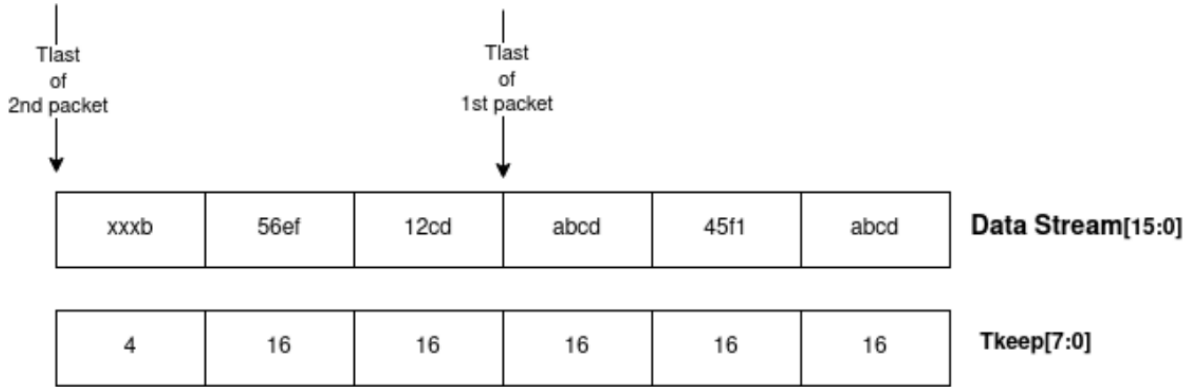
Thus, this is all about finite state machines. From the above information finally, we can conclude that synchronous sequential circuits affect their states for each positive otherwise negative conversion of the CLK signal depending on the input. So, this behaviour can be signified in the form of graphical which is known as a state diagram. Another name of a synchronous sequential circuit is FSM (finite state machine).

Assignment -1

- Implement the design having below specifications



- Data stream of 16 bits comes continuously infinite time along with the valid, last and keep signals.
- 'keep' signal will indicate the number of valid bits are present in that corresponding data stream starts from LSB.
- Output should follow the below stream flow



- Design should give 16 valid bits in every clock cycle exception may be valid for the last cycle in the packet based on the number of valid bits present in the packet.
- Each packet will end with the last signal asserted high and immediately next packet data will start coming.
- If the current clock cycle does not have 16 valid bits of data it will take enough bits from the succeeding samples to make 16 valid bits.
    - Input : 16 bits and keep is of 8 bits
    - output : 16 bits
    - Design should comply with AXI Stream protocol.(Use necessary signals only like tdata, tvalid, tready and tkeep)

1. Implementation

This FSM module processes data streams between two interfaces: the s_axis (source axis) and m_axis (master axis). The FSM handles the transfer of 16-bit data chunks and manages the validity and readiness signals between these interfaces. It ensures that data is correctly aligned and transferred based on control signals, specifically s_axis_tkeep.

Parameters and Local Parameters

DATA_WIDTH: Defines the width of the data, set to 16 bits.

States:

STATE_Initial: Initial state, waiting for data.

STATE_1: Intermediate state processing smaller chunks of data.

STATE_2: State processing full 16-bit data.

STATE_3: Additional state for handling carry-over data.

STATE_4: Final state for additional processing of carry-over data.

Registers

CurrentState: Holds the current state of the FSM.

NextState: Determines the next state based on the current conditions.

mem, mem_s: Temporary storage registers for data manipulation.

mem_1, mem_2, mem_3: Additional storage registers for intermediate data.

t_keep_out, t_keep_out_reg: Registers to manage the tkeep values.

m_axis_tlast_reg: Register to manage the tlast signal.

State Transition Logic

The FSM transitions between states based on the s_axis_tkeep values and the readiness of the master interface (m_axis_tready). The transitions are defined in the always @* block:

STATE_Initial: Waits for s_axis_tvalid to be asserted. Based on s_axis_tkeep, transitions to:

STATE_Initial if s_axis_tkeep is 0.

STATE_1 if s_axis_tkeep is 4, 8, or 12.

STATE_2 if s_axis_tkeep is 16.

STATE_1: Checks if m_axis_tready is asserted or if both s_axis_tvalid and s_axis_tready are asserted. Based on s_axis_tkeep, transitions to:

STATE_Initial if s_axis_tkeep is 0.

STATE_1 if s_axis_tkeep is 4, 8, or 12.

STATE_3 if s_axis_tkeep is 16.

STATE_2: Checks if m_axis_tready is asserted. Based on s_axis_tkeep, transitions to:

STATE_Initial if s_axis_tkeep is 0.

STATE_1 if s_axis_tkeep is 4, 8, or 12.

STATE_Initial if s_axis_tkeep is 16.

STATE_3: Checks if m_axis_tready is asserted. Based on s_axis_tkeep, transitions to:

STATE_Initial if s_axis_tkeep is 0.

STATE_1 if s_axis_tkeep is 4, 8, or 12.

STATE_4 if s_axis_tkeep is 12.

STATE_3 if s_axis_tkeep is 16.

STATE_4: Checks if m_axis_tready is asserted. Based on s_axis_tkeep, transitions to:

STATE_4 if s_axis_tkeep is 0.

STATE_1 if s_axis_tkeep is 4, 8, or 12.

STATE_3 if s_axis_tkeep is 16.

Signal Assignments

s_axis_tready:

Asserted (set to 1) in states STATE_1, STATE_2, STATE_3, and STATE_4.

Deasserted (set to 0) in STATE_Initial and any other default state.

Data Processing:

In STATE_Initial: Resets mem_s and t_keep_out.

In STATE_1:

If s_axis_tkeep is less than 16 and t_keep_out is 16, it processes the input data by left-shifting and combines it with the existing mem data, updating t_keep_out accordingly.

In STATE_2:

If s_axis_tkeep is 16, it directly assigns s_axis_tdata to mem_s and updates t_keep_out.

In STATE_3:

If s_axis_tkeep is 16 and t_keep_out is greater than 16, it processes the input data similar to STATE_1 but with more detailed handling of intermediate storage (mem_1).

In STATE_4:

If s_axis_tkeep is less than 16 and t_keep_out is 16, it processes the data by combining shifted s_axis_tdata with mem and mem_1 data.

Output Signal Management:

m_axis_tlast:

Updated in every clock cycle to reflect the correct end of the packet based on the current state and s_axis_tlast.

m_axis_tdata:

Driven by mem_s if m_axis_tvalid is asserted, otherwise by mem_3.

m_axis_tvalid:

Asserted when the current state is STATE_4 or when t_keep_out equals 16, provided m_axis_tready is also asserted.

m_axis_tkeep:

Set to t_keep_out if m_axis_tvalid is asserted, otherwise set to t_keep_out_reg.

Additional Logic

mem, mem_s, mem_1, mem_2, mem_3:

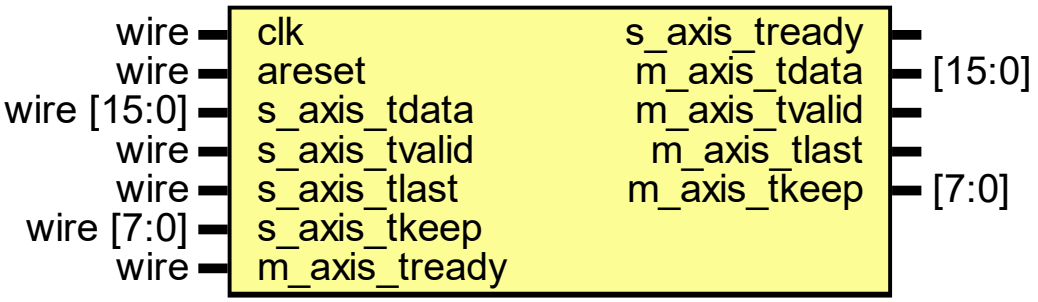
Used for intermediate data storage and handling shifts and concatenations.

t_keep_out, t_keep_out_reg:

Manage the alignment and size of the data chunks being processed and transferred.

This FSM design ensures the data is correctly processed and transferred between interfaces, handling various conditions dictated by the s_axis_tkeep control signal. It efficiently manages data alignment and ensures proper handshaking with the master interface through tready, tvalid, and tlast signals.

2. Block Diagram



3. Port Description

Ports

Port name	Direction	Type	Description
clk	input	wire	
areset	input	wire	
s_axis_tdata	input	wire [15:0]	
s_axis_tvalid	input	wire	
s_axis_tready	output		
s_axis_tlast	input	wire	
s_axis_tkeep	input	wire [7:0]	
m_axis_tdata	output	[15:0]	
m_axis_tvalid	output		

Port name	Direction	Type	Description
m_axis_tready	input	wire	
m_axis_tlast	output		
m_axis_tkeep	output	[7:0]	

Signals

Name	Type	Description
CurrentState	reg [3:0]	
NextState	reg [3:0]	
mem	reg [15:0]	
mem_s	reg [15:0]	
mem_1	reg [15:0]	
mem_2	reg [15:0]	
mem_3	reg [15:0]	
t_keep_out=0	reg [7:0]	
t_keep_out_reg=0	reg [7:0]	
m_axis_tlast_reg	reg	

Constants

Name	Type	Value	Description
DATA_WIDTH		16	
STATE_Initial		3'd0	
STATE_1		3'd1	
STATE_2		3'd2	
STATE_3		3'd3	
STATE_4		3'd4	

4. Module code

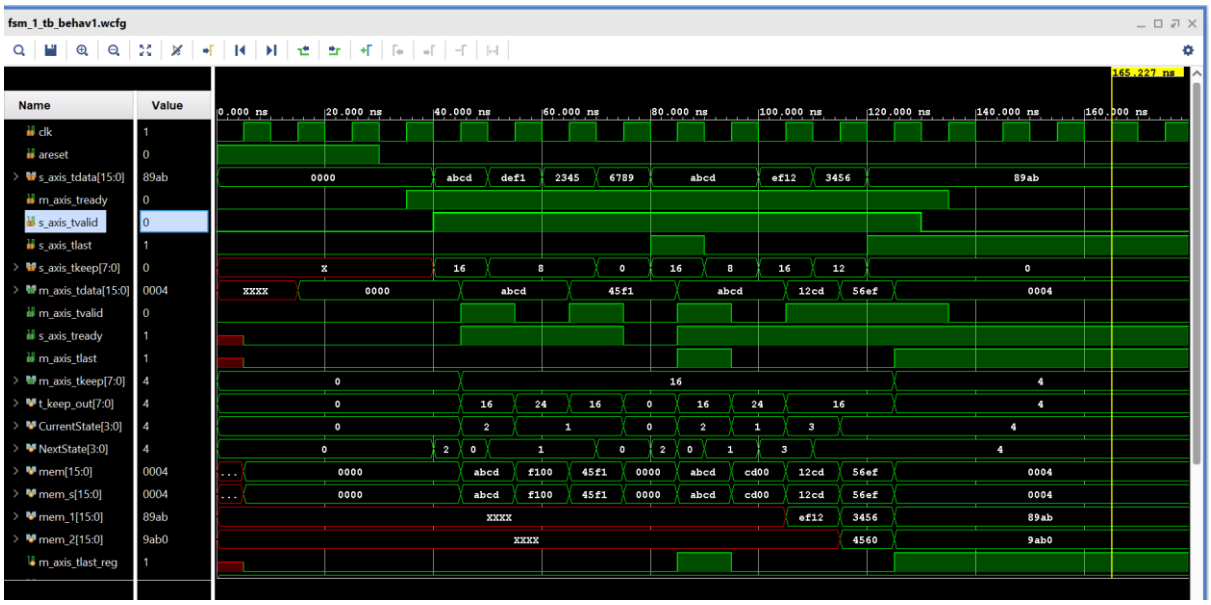
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_5/Assignment_1/FSM_1.sv

5. Test bench code

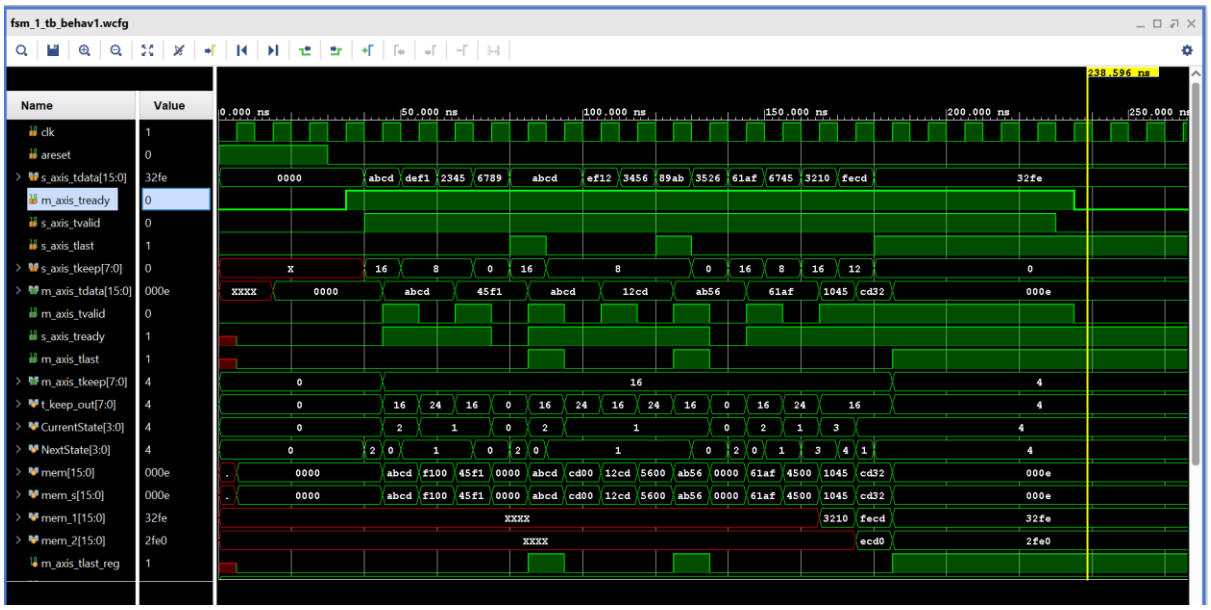
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_4/Assignment_1/temp.sv

6. Test cases

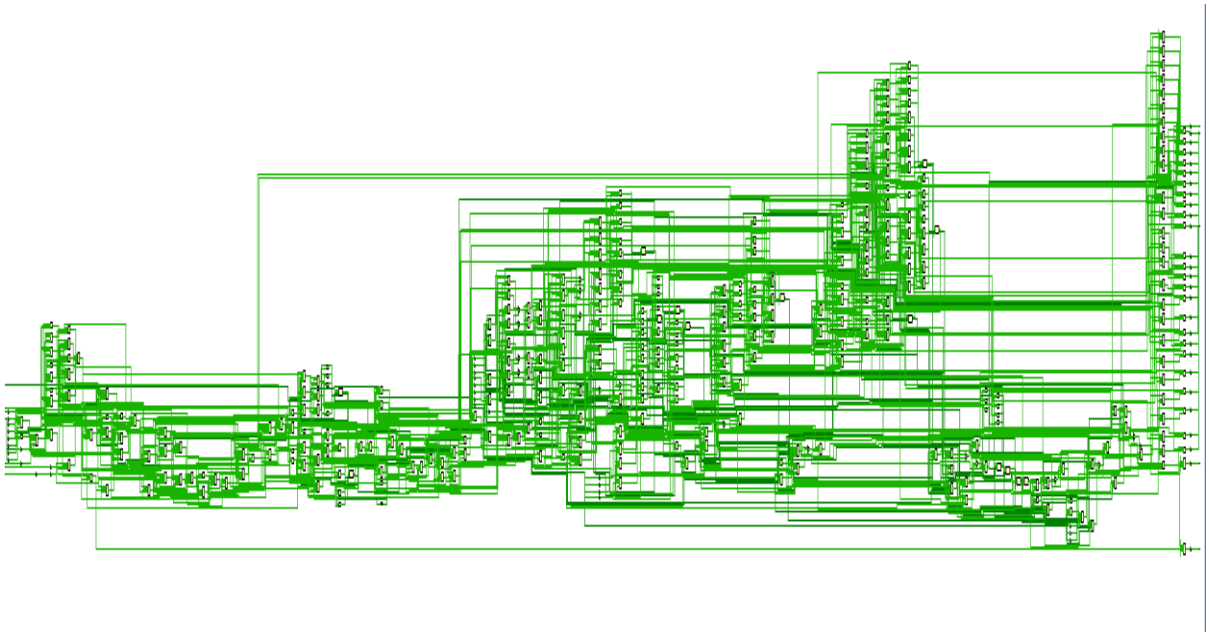
- **Input from given assignment data.**



- **Input data taken from different .**



7. Schematic Diagram



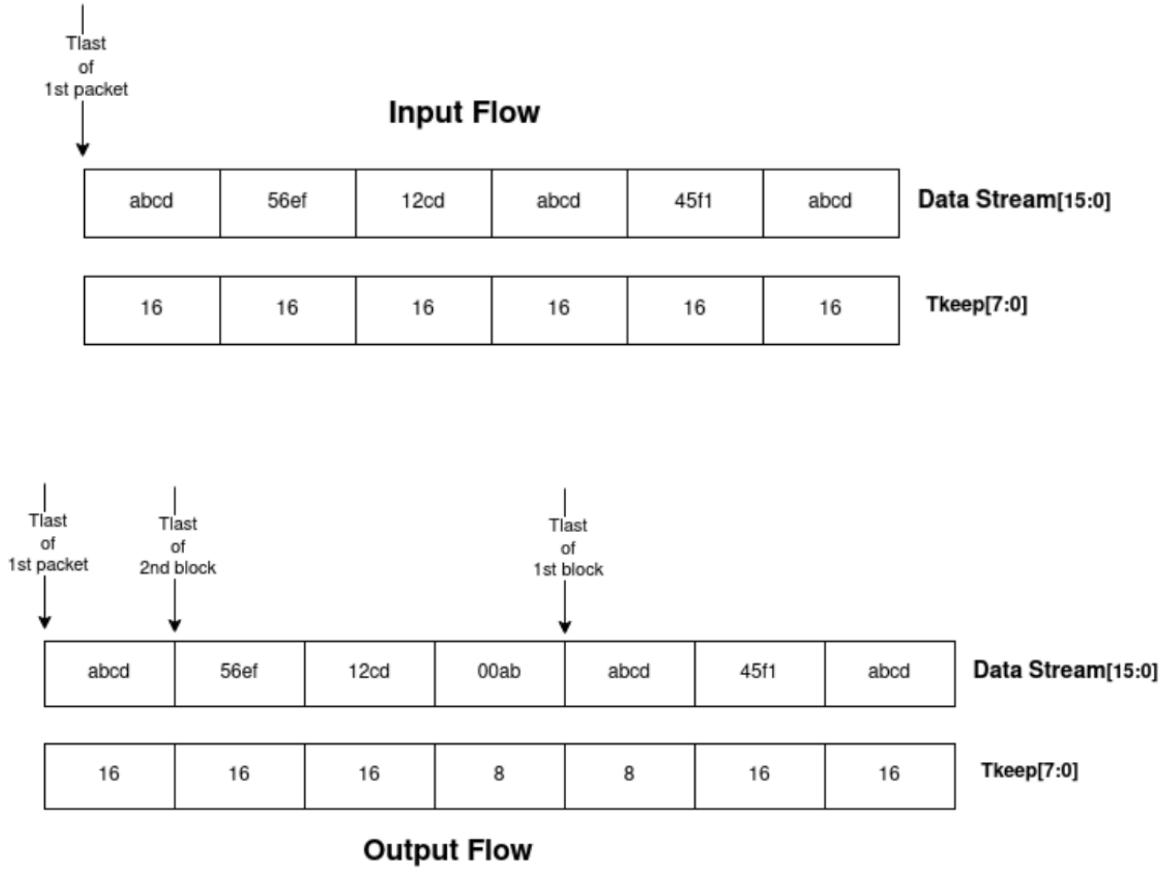
8. Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	227	41000	0.55
FF	68	82000	0.08
IO	56	300	18.67
BUFG	1	32	3.13

Assignment-2



- Input and output port are 16 bits along with keep signal of 8 bits and one-bit last signal
- Input is an infinite continuous stream of packets. Each packet will end with the last signal asserted high.
- Design should have 3 AXI Stream ports, one port for output, one port for input data and another port for input configuration.
- Design should always accept configuration first and then data. Configuration will give the size of the block.
- Based on the size of the block, the design should assert the last signal in the output port.
- Example input and output flow is shown above for the block size of 40.
- A keep signal will indicate the number of valid bits present in the current sample starting from the LSB.
- Once the input packet last signal receives, the design should again accept the configuration.

- Once the design receives input packet last signal, even though the block size data is not sent out, output last signal should be asserted.
- Each port should comply with the AXI stream protocol.
- Integrate this IP with the previous Assignment-1 IP and get the final output where the data shouldn't have any empty bits in any transaction (except the last transaction).
- NOTE: If the data width is 16, whole 16 bits should be occupied in a transaction else that transaction holds empty bits
- Write a System Verilog testbench for this design alone, integrated modules and check whether design meets expectations.

1. Implementation

The fsm_3 module is designed to process data streams based on a configurable packet configuration. This description details the module's operation, including state transitions and data handling.

Module Parameters

DW_IN: Input data width (default 16 bits).

DW_OUT: Output data width (default 16 bits).

DW_USER: User data width (default 8 bits).

Input Ports

clock: System clock.

reset_n: Active-low reset signal.

config_in_tdata: Configuration data input.

config_in_tvalid: Configuration data valid signal.

data_in_tdata: Input data.

data_in_tuser: User data associated with input data.

data_in_tlast: Last data indicator.

data_in_tvalid: Input data valid signal.

data_out_tready: Ready signal from the output interface.

Output Ports

config_out_tready: Configuration data ready signal.

data_in_tready: Input data ready signal.

data_out_tdata: Output data.

data_out_tuser: User data associated with output data.

data_out_tlast: Last data indicator for output.

data_out_tvalid: Output data valid signal.

Local Parameters (State Definitions)

CONFIG_DATA: Configuration data processing state.

RD_WR_DATA: Read/write data state.

MERGE: Merge input data with stored data state.

FILTER: Filter data state.

FLUSH: Flush remaining data state.

SEND_LAST0_SAMPLE: Send the last sample state.

Registers

data_in_tdata_reg: Registered input data.

data_in_tuser_reg: Registered user data.

data_in_tlast_reg: Registered last data indicator.

data_in_tvalid_reg: Registered input data valid signal.

config_in_tdata_reg: Registered configuration data.

config_in_tvalid_reg: Registered configuration data valid signal.

rem_bits, mem_1: Temporary storage for data bits.

rem_user, count_user, count: Counters for user data and data bits.

last: Indicator for the last data in the packet.

data_out_tdata_1d, data_out_tuser_1d: Registered output data and user data.

Operation and State Transitions

Input Registering

The REGISTERING_INPUT block registers the input data, user data, and control signals. It updates these registers on every clock edge if the corresponding ready and valid signals are asserted.

Next State Logic

The NEXT_STATE_SEQ block updates the current state based on the next state determined by the NEXT_STATE_DECODER block.

State Transition Logic

The NEXT_STATE_DECODER block determines the next state based on the current state and input signals:

CONFIG_DATA: Transition to RD_WR_DATA if the configuration data is valid and ready.

RD_WR_DATA: Transition to MERGE if the input data is valid and the output is ready.

MERGE: Transition to FILTER, SEND_LAST0_SAMPLE, or stay in MERGE based on the count_user and data_in_tlast signals.

FILTER: Transition to FLUSH if the data is ready.

FLUSH: Transition back to MERGE if the data is ready.

SEND_LAST0_SAMPLE: Transition back to RD_WR_DATA if the data is ready.

State Definitions and Data Handling

The STATE_DEFINITION block defines the operations performed in each state:

CONFIG_DATA: Registers the configuration data.

RD_WR_DATA: Reads input data and increments the user data counter.

MERGE: Merges input data with stored data and updates the counter and last data indicator.

FILTER: Filters the data based on the user data count and configuration data.

FLUSH: Flushes the remaining data based on the user data count.

SEND_LAST0_SAMPLE: Sends the last data sample with the last data indicator asserted.

Output Logic

The OUTPUT_DECODER block assigns the output data, user data, and control signals based on the current state:

RD_WR_DATA: Outputs the registered data and user data.

MERGE: Outputs merged data if the user data count is greater than 0.

FILTER: Outputs filtered data based on the user data count.

FLUSH: Outputs remaining data based on the user data count.

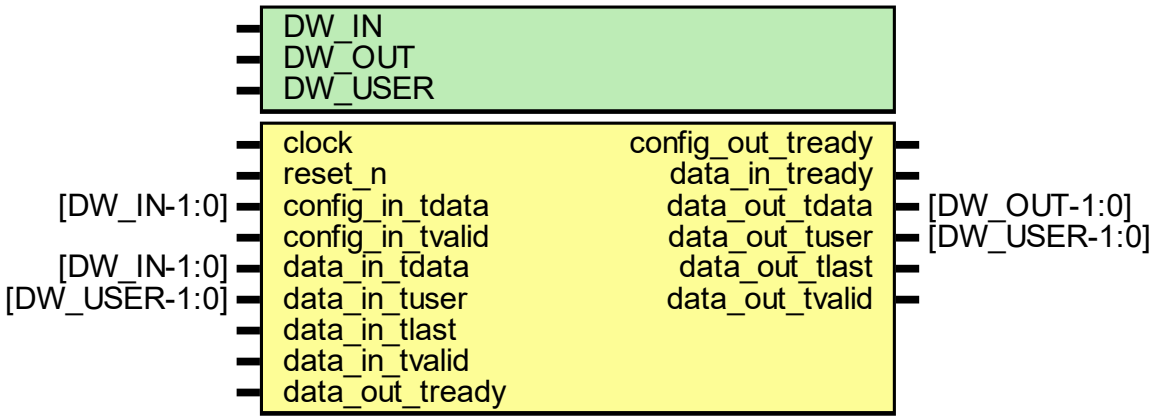
SEND_LAST0_SAMPLE: Outputs the last data sample.

Data Ready Signal

The data_in_tready signal is asserted if the FSM is in a state that is ready to accept input data (RD_WR_DATA, FLUSH, FILTER, MERGE, SEND_LAST0_SAMPLE) and the output is ready.

This FSM efficiently processes and transfers data based on configurable packet configurations, ensuring proper handling of data streams and control signals.

2. Block Diagram



3. Port description

Generics

Generic name	Type	Value	Description
DW_IN		16	
DW_OUT		16	
DW_USER		8	

Ports

Port name	Direction	Type	Description
clock	input		
reset_n	input		
config_in_tdata	input	[DW_IN-1:0]	
config_in_tvalid	input		

Port name	Direction	Type	Description
config_out_tready	output		
data_in_tdata	input	[DW_IN-1:0]	
data_in_tuser	input	[DW_USER-1:0]	
data_in_tlast	input		
data_in_tvalid	input		
data_in_tready	output		
data_out_tdata	output	[DW_OUT-1:0]	
data_out_tuser	output	[DW_USER-1:0]	
data_out_tlast	output		
data_out_tvalid	output		
data_out_tready	input		

Signals

Name	Type	Description
state	enum r	
next	enum r	
data_in_tdata_reg	reg [DW_IN-1:0]	
data_in_tuser_reg	reg [DW_USER-1:0]	

Name	Type	Description
data_in_tlast_reg	reg	
data_in_tvalid_reg	reg	
data_out_tready_reg	reg	
config_in_tdata_reg	reg [DW_IN -1:0]	
config_in_tuser_reg	reg [DW_USER-1:0]	
config_in_tvalid_reg	reg	
config_out_tready_reg	reg	
rem_bits	reg [DW_OUT -1:0]	
mem_1	reg [DW_OUT -1:0]	
rem_user=0	reg [DW_USER -1:0]	
count_user=0	reg [DW_USER -1:0]	
count=0	reg [DW_USER -1:0]	
last	reg	
data_out_tdata_1d	reg [DW_OUT-1:0]	
data_out_tuser_1d	reg [DW_USER-1:0]	

4. Module code

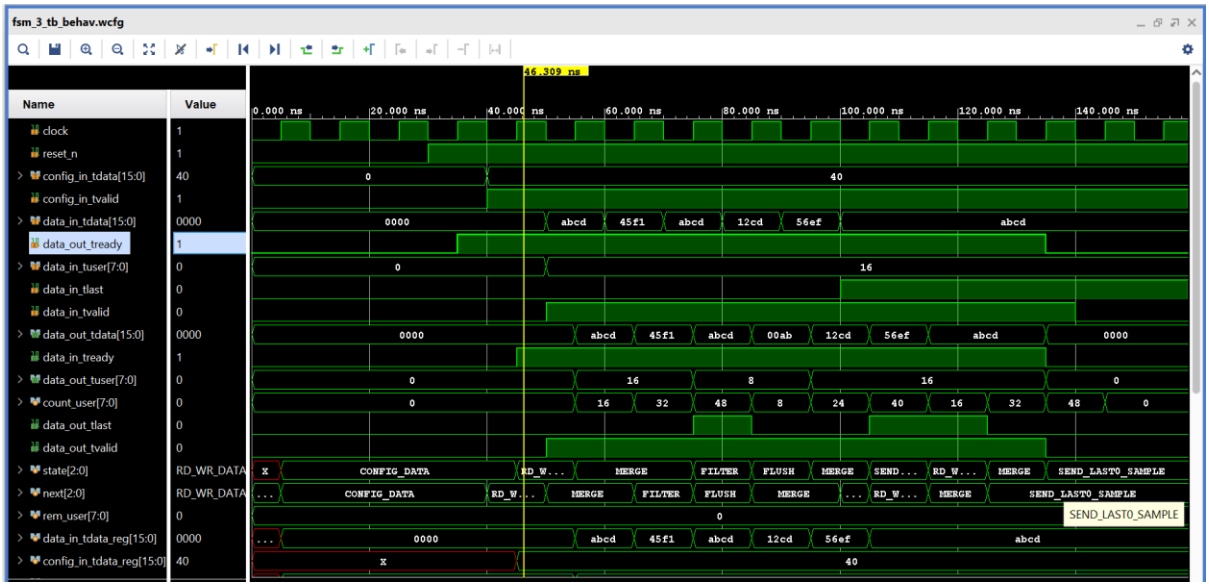
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_5/Assignment_2/fsm_3.sv

5. Testbench code

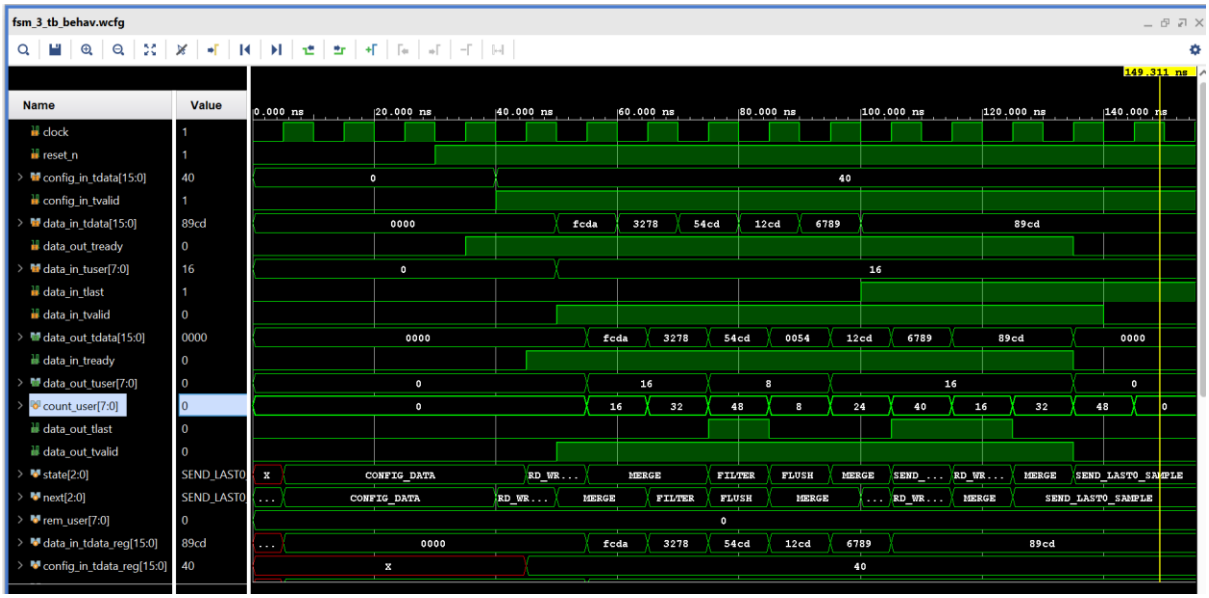
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_5/Assignment_2/fsm_3_tb.sv

6. Test cases

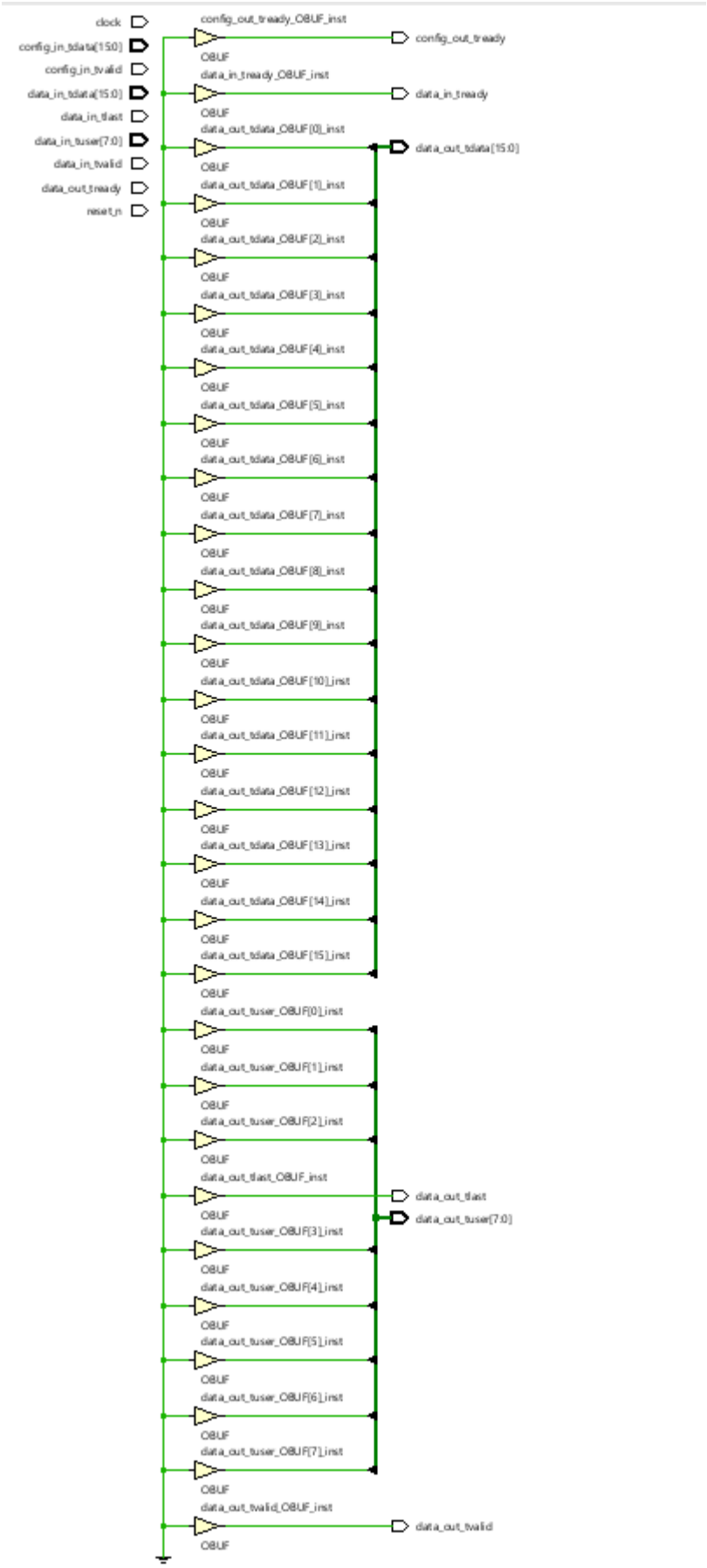
- Input data from assignment .



- Input data from my side .



7. Schematic Diagram


clock
config_in_tdata[15:0]
config_in_tvalid
data_in_tdata[15:0]
data_in_tlast
data_in_tuser[7:0]
data_in_tvalid
data_out_tready
reset_n
config_out_tready_OBUF_inst
OBUF
config_out_tready
data_in_tready_OBUF_inst
OBUF
data_in_tready
data_out_tdata_OBUF[0]_inst
OBUF
data_out_tdata[15:0]
data_out_tdata_OBUF[1]_inst
OBUF
data_out_tdata_OBUF[2]_inst
OBUF
data_out_tdata_OBUF[3]_inst
OBUF
data_out_tdata_OBUF[4]_inst
OBUF
data_out_tdata_OBUF[5]_inst
OBUF
data_out_tdata_OBUF[6]_inst
OBUF
data_out_tdata_OBUF[7]_inst
OBUF
data_out_tdata_OBUF[8]_inst
OBUF
data_out_tdata_OBUF[9]_inst
OBUF
data_out_tdata_OBUF[10]_inst
OBUF
data_out_tdata_OBUF[11]_inst
OBUF
data_out_tdata_OBUF[12]_inst
OBUF
data_out_tdata_OBUF[13]_inst
OBUF
data_out_tdata_OBUF[14]_inst
OBUF
data_out_tdata_OBUF[15]_inst
OBUF
data_out_tuser_OBUF[0]_inst
OBUF
data_out_tuser_OBUF[1]_inst
OBUF
data_out_tuser_OBUF[2]_inst
OBUF
data_out_tlast_OBUF_inst
data_out_tlast
OBUF
data_out_tuser[7:0]
data_out_tuser_OBUF[3]_inst
OBUF
data_out_tuser_OBUF[4]_inst
OBUF
data_out_tuser_OBUF[5]_inst
OBUF
data_out_tuser_OBUF[6]_inst
OBUF
data_out_tuser_OBUF[7]_inst
OBUF
data_out_tvalid_OBUF_inst
data_out_tvalid
OBUF

8. Utilization

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
IO	28	300	9.33