RTL Designs with pipelining

Contents

Revision History

Version	Date (DD/MM/YY)	Description	Author(s)	Reviewer(s)
1	1-04-2024		Alavala Chinnapa Reddy	

Objective

- Understanding the concept of pipelining and its use case scenarios in real applications.

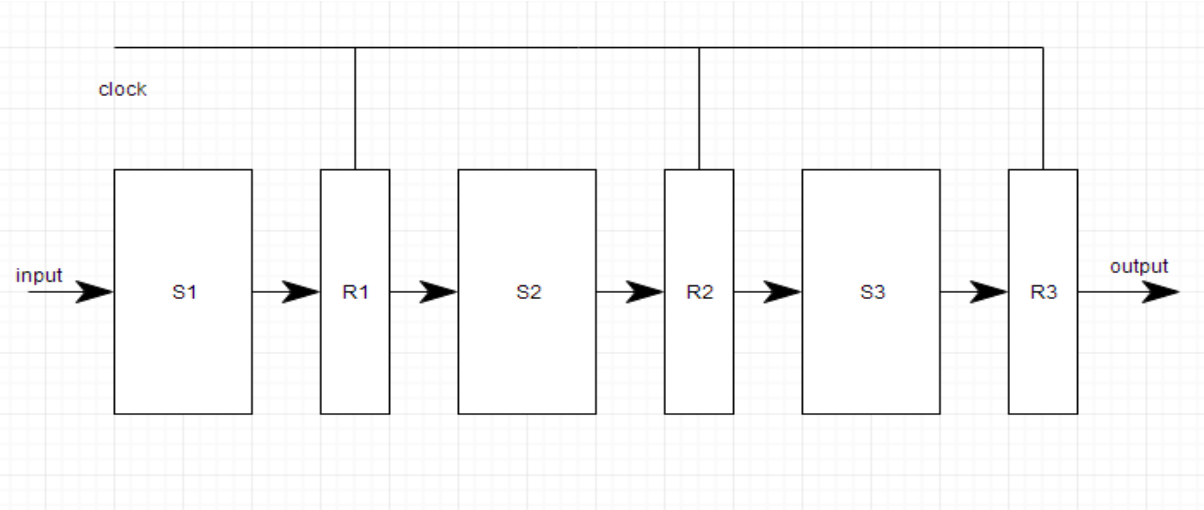
What is Pipelining

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages relate to one another to form a pipe like structure. Instructions enter from one end and exit from another end.

Pipelining increases the overall instruction throughput.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Pipeline system is like the modern-day assembly line setup in factories. For example, in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed-point numbers etc. For example: The input to the Floating-Point Adder pipeline is:

$$X = A*2^a$$

$$Y = B*2^b$$

Here A and B are mantissas (significant digit of floating-point numbers), while **a** and **b** are exponents.

The floating-point addition and subtraction are done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

In this a stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus, we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Pipeline Conflicts

There are some factors that cause the pipeline to deviate its normal performance. Some of these factors are given below:

1. Timing Variations

All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

2. Data Hazards

When several instructions are in partial execution, and if they reference same data then the problem arises. We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

3. Branching

To fetch and execute the next instruction, we must know what that instruction is. If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

4. Interrupts

Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

5. Data Dependency

It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Advantages of Pipelining

1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

Assignment -1

- Refer to the DSP architecture of any Xilinx - Ultra Scale FPGA / Intel-Agilex FPGA and mimic the same architecture with RTL coding of your choice
- It can be included with multiplication operations followed by additions (MAC) or in other ways
- In post implementation, make sure this design is invoking the DSP slices available on the FPGA.
- The design should comply with the AXI Stream Protocol

1. DSP48E2 Functionality

The DSP48E2 slice consists of a 27-bit pre-adder, 27 x 18 multiplier, and a flexible 48-bit ALU that serves as a post-adder/subtracter, accumulator, or logic unit.

The DSP48E2 supports many independent functions. These functions include:

• Multiply

• Multiply accumulate (MACC)

• Multiply add

• Four-input add

• Barrel shift

• Wide-bus multiplexing

• Magnitude comparator

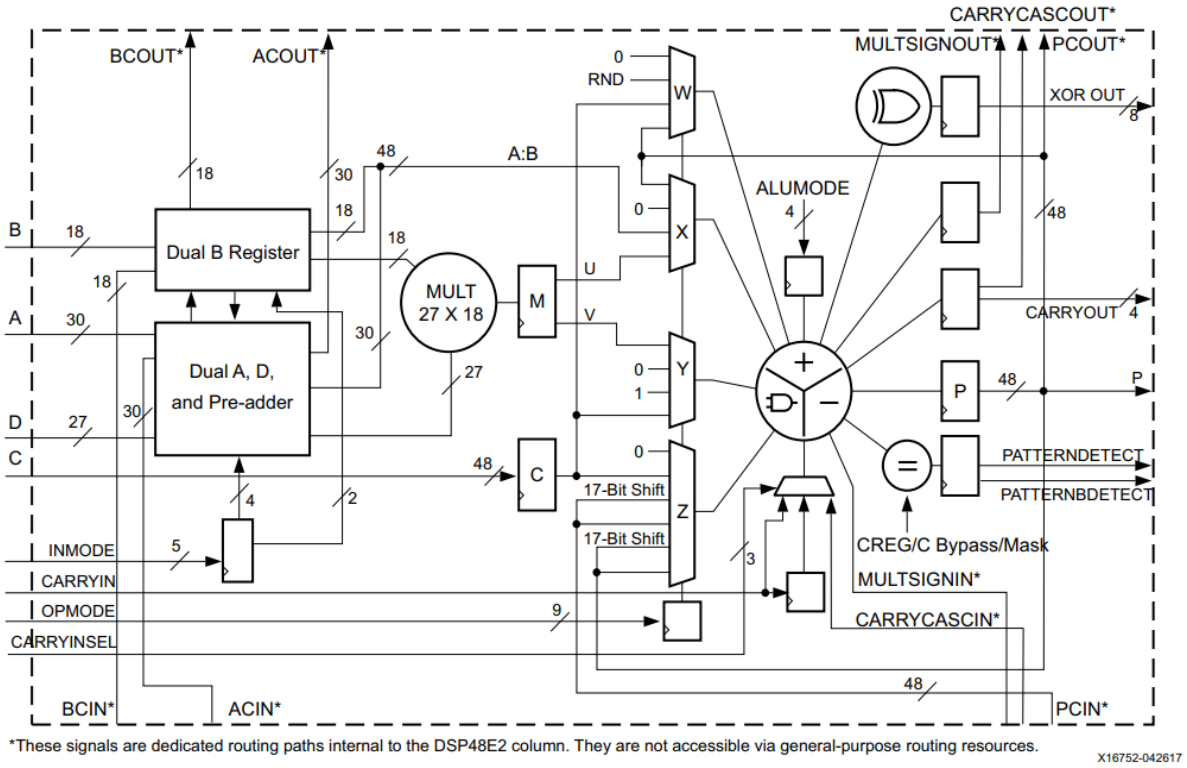
• Bitwise logic functions

• Wide XOR

• Pattern detects

• Wide counter

The architecture also supports cascading multiple DSP48E2 slices to form wide math functions, DSP filters, and complex arithmetic without the use of general logic



2. DSP48E2 Features

The features in the DSP48E2 slice are:

- 27-bit pre-adder with D register to enhance the capabilities of the A or B path
- A or B can be selected as pre-adder input to allow for wider multiplication coefficients
- The result of the pre-adder can be sent to both inputs of the multiplier to provide squaring capability
- INMODE control supports balanced pipelining when dynamically switching between multiply (A*B) and add operations (A+B)
- 27 x 18 multiplier
- 30-bit A input of which the lower 27 bits feed the A input of the multiplier, and the entire 30-bit input forms the upper 30 bits of the 48-bit A:B concatenated internal bus
- Cascading A and B input:
    - Semi-independently selectable pipelining between direct and cascade paths
    - Separate clock enables for two-deep A and B set of input registers
- Independent C input and C register with independent reset and clock enable

- CARRYCASCIN and CARRYCASCOUT internal cascade signals to support 96-bit accumulators/adders/subtracters in two DSP48E2 slices, and to support cascading more than two DSP slices

- MULTSIGNIN and MULTSIGNOUT internal cascade signals with special OPMODE setting to support a 96-bit MACC extension
- Single Instruction Multiple Data (SIMD) Mode for four-input adder/subtracter, which precludes use of multiplier in first stage:
    - Dual 24-bit SIMD adder/subtracter/accumulator with two separate CARRYOUT signals
    - Quad 12-bit SIMD adder/subtracter/accumulator with four separate CARRYOUT signals
- 48-bit logic unit:
    - Bitwise logic operations-two-input AND, OR, NOT, NAND, NOR, XOR, and XNOR
    - Logic unit mode dynamically selectable via ALUMODE
- 96-bit wide XOR logic selectable from eight 12-bit XORs to one 96-bit XOR
- Pattern detector:
    - Overflow/underflow support
    - Convergent rounding support
    - Terminal count detection support and auto resetting: auto resetting can give priority to clock enable
- Cascading 48-bit P bus supports internal low-power adder cascade: 48-bit P bus allows for 12-bit quad or 24-bit dual SIMD adder cascade support
- Optional 17-bit right shift to enable wider multiplier implementation
- Dynamic user-controlled operating modes:
    - 9-bit OPMODE control bus provides W, X, Y, and Z multiplexer select signals
    - 5-bit INMODE control bus provides selects for 2-deep A and B registers, pre-adder add-sub control as well as mask gates for pre-adder multiplexer functions
    - 4-bit ALUMODE control bus selects logic unit function and accumulator add-sub control
- Carry in for the second stage adder:
    - Support for rounding
    - Support for wider add/subtracts
    - 3-bit CARRYINSEL multiplexer
- Carry out for the second stage adder:
    - Support for wider add/subtracts
    - Available for each SIMD adder (up to four)
    - Cascaded CARRYCASCOUT and MULTSIGNOUT allows for MACC extensions up to 96 bits
- Single clock for synchronous operation
- Optional input, pipeline, and output/accumulate registers
- Optional registers for control signals (OPMODE, ALUMODE, and CARRYINSEL)
- Independent clock enables and synchronous resets with programmable polarity for greater flexibility
- Internal multiplier and XOR logic can be gated off when unused to save power

The DSP slice consists of a multiplier followed by an accumulator. At least three pipelines registers are required for both multiply and multiply-accumulate operations to run at full speed. The multiply operation in the first stage generates two partial products that need to be added together in the second stage.

When only one or two registers exist in the multiplier design, the M register should always be used to save power and improve performance.

Add/Sub and Logic Unit operations require at least two pipeline registers (input, output) to run at full speed.

The cascade capabilities of the DSP slice are extremely efficient at implementing high-speed pipelined filters built on the adder cascades instead of adder trees.

Multiplexers are controlled with dynamic control signals, such as OPMODE, ALUMODE, and

CARRYINSEL, enabling a great deal of flexibility. Designs using registers and dynamic opmodes are better equipped to take advantage of the DSP slice's capabilities than combinatorial multiplies.

In general, the DSP slice supports both sequential and cascaded operations due to the dynamic OPMODE and cascade capabilities. Fast Fourier Transforms (FFTs), floating point, computation (multiply, add/sub, divide), counters, and large bus multiplexers are some applications of the DSP slice.

Additional capabilities of the DSP slice include synchronous resets and clock enables, dual An input pipeline registers, pattern detection, Logic Unit functionality, single instruction/multiple data (SIMD) functionality, and MACC and Add-Acc extension to 96 bits.

The DSP slice supports convergent and symmetric rounding, terminal count detection and auto-resetting for counters, and overflow/underflow detection for sequential accumulators.

A 96-bit wide XOR function can be implemented as eight 12-bit wide XOR, four 24-bit wide XOR, or two 48-bit wide XOR.

3. Design-1(with no clk and rst)

A. Implementation

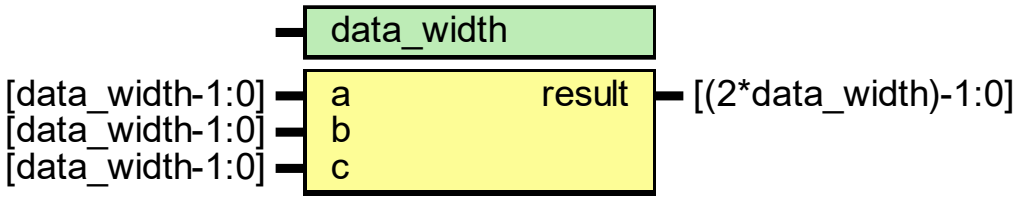
The dsp_slice module you provided is a simple design without clock and reset signals, and it does not include AXI stream interfaces.

This module is named dsp_slice and it takes three input vectors a, b, and c, each of width data_width. It also has an output vector result which is twice the width of the input data (2*data_width). The width of the output is calculated as twice the width of the input data because the multiplication of a and b might produce a result with double the width of the inputs.

Inside the always block, there is a combinational logic which computes the output result. It performs the multiplication of inputs a and b, then adds input c to the result. The multiplication result (a*b) is calculated as a product of two data_width-bit vectors, resulting in a vector with width 2*data_width. Finally, c is added to this result to produce the final output result.

This module essentially implements a simple DSP (Digital Signal Processing) slice, performing multiplication and addition operations on the input vectors a, b, and c.

B. Block diagram



C. Port Description

Generics

Generic name	Type	Value	Description
data_width		8 or 16	

Ports

Port name	Direction	Type	Description
a	input	[data_width-1:0]	Data input 1
b	input	[data_width-1:0]	Data input 2
c	input	[data_width-1:0]	Data input 3
result	output	[(2*data_width)-1:0]	Final output

D. Module code

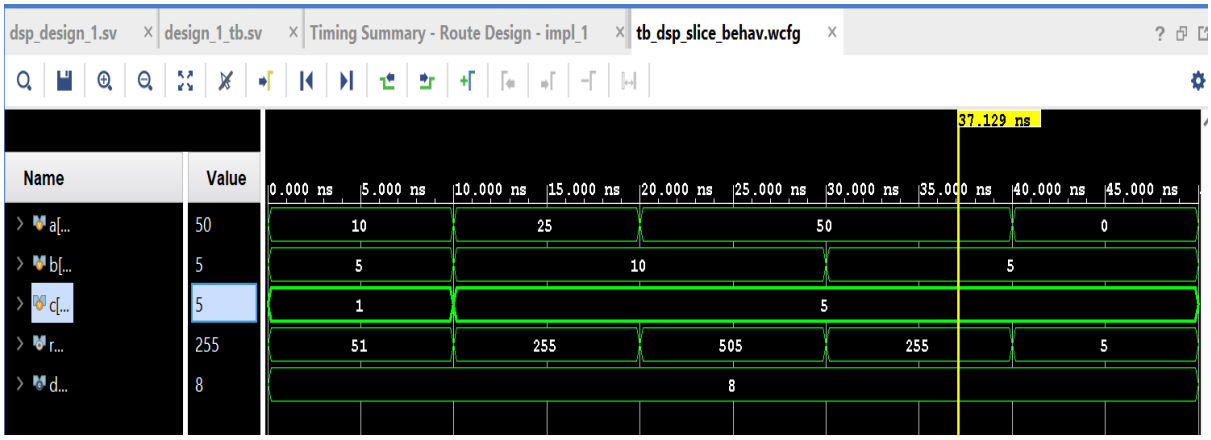
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/Design_1/dsp_design_1.sv

E. Test Bench code

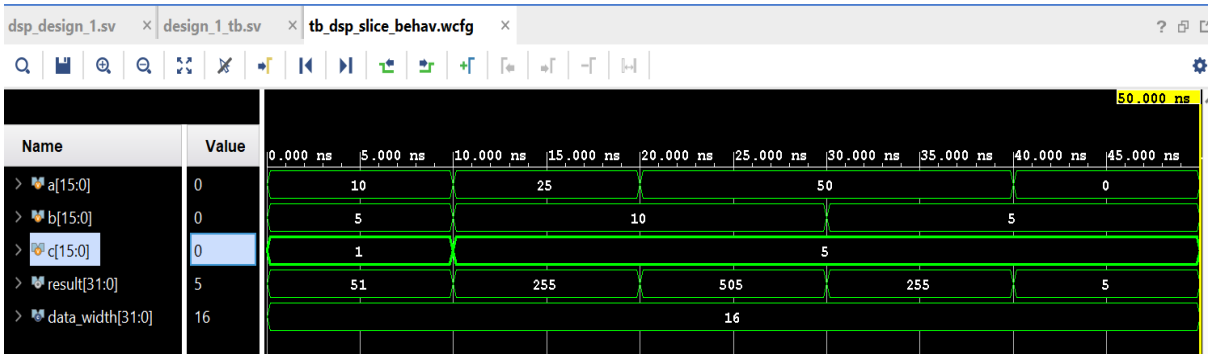
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/Design_1/design_1_tb.sv

F. Test cases

- **Data_width = 8**

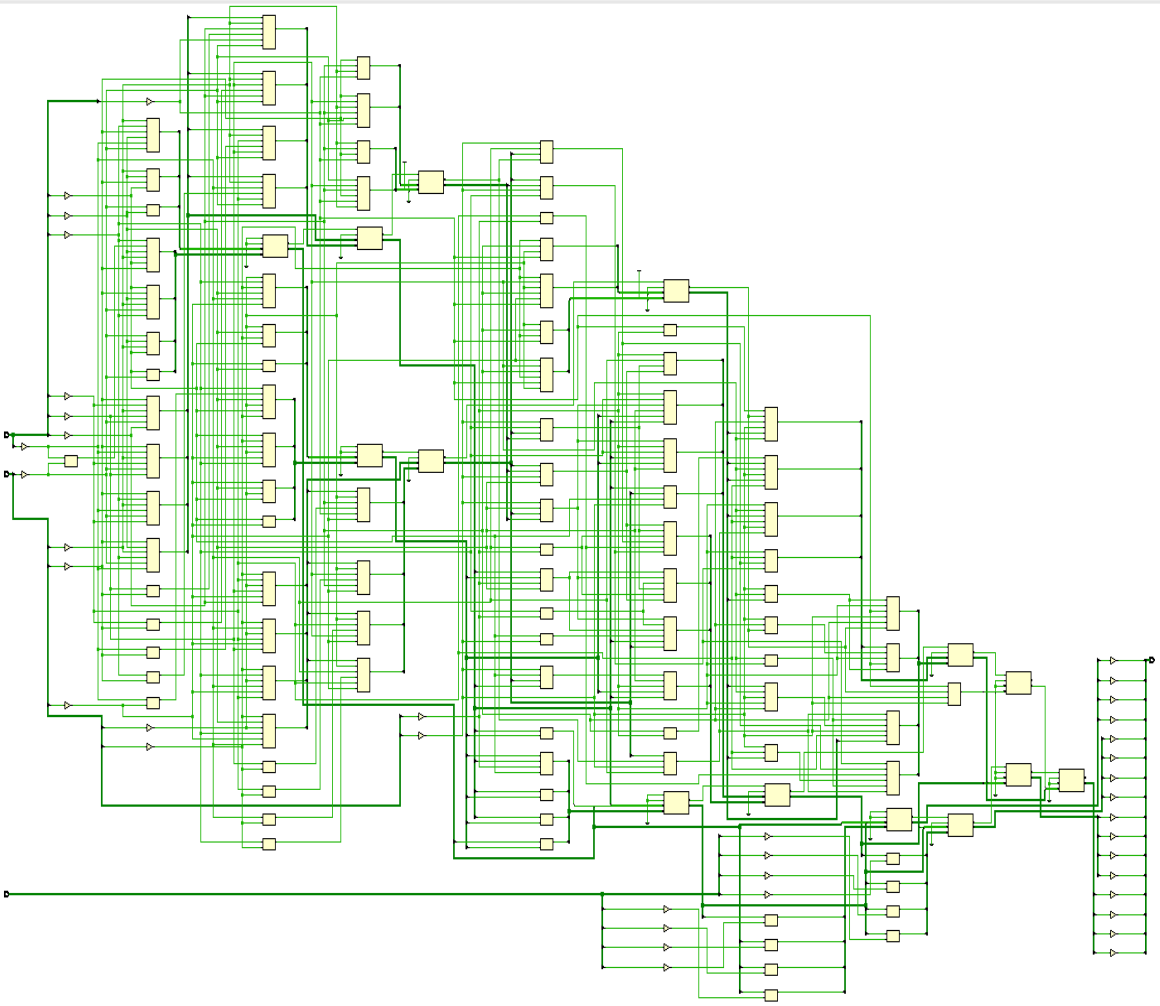


- **Data_width = 16**

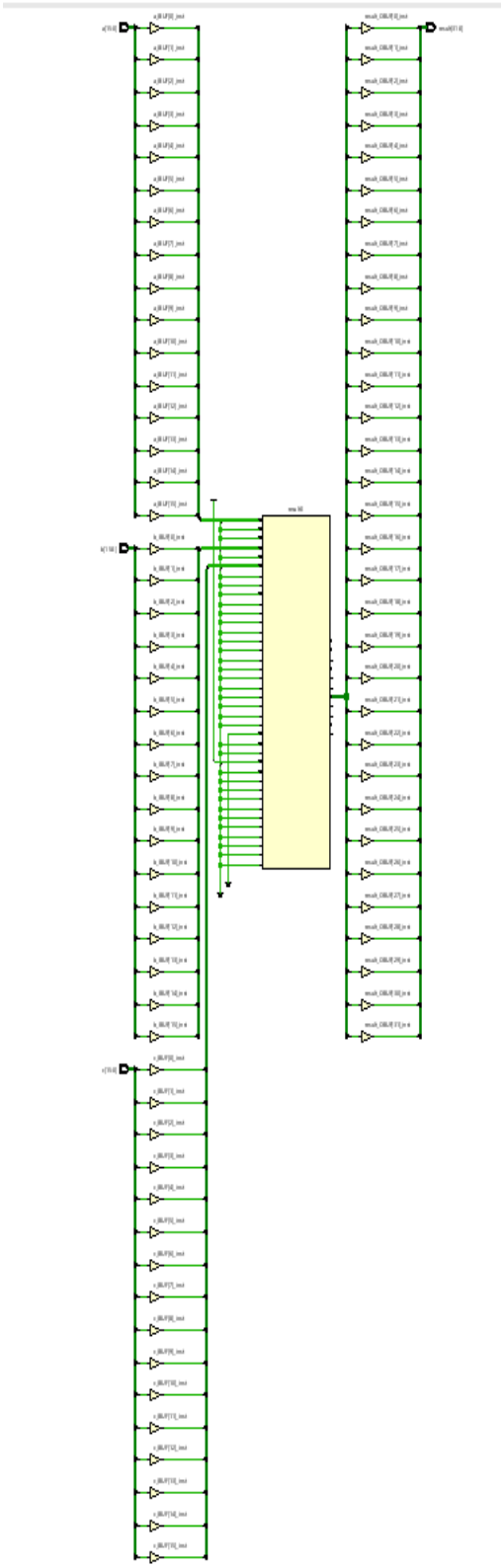


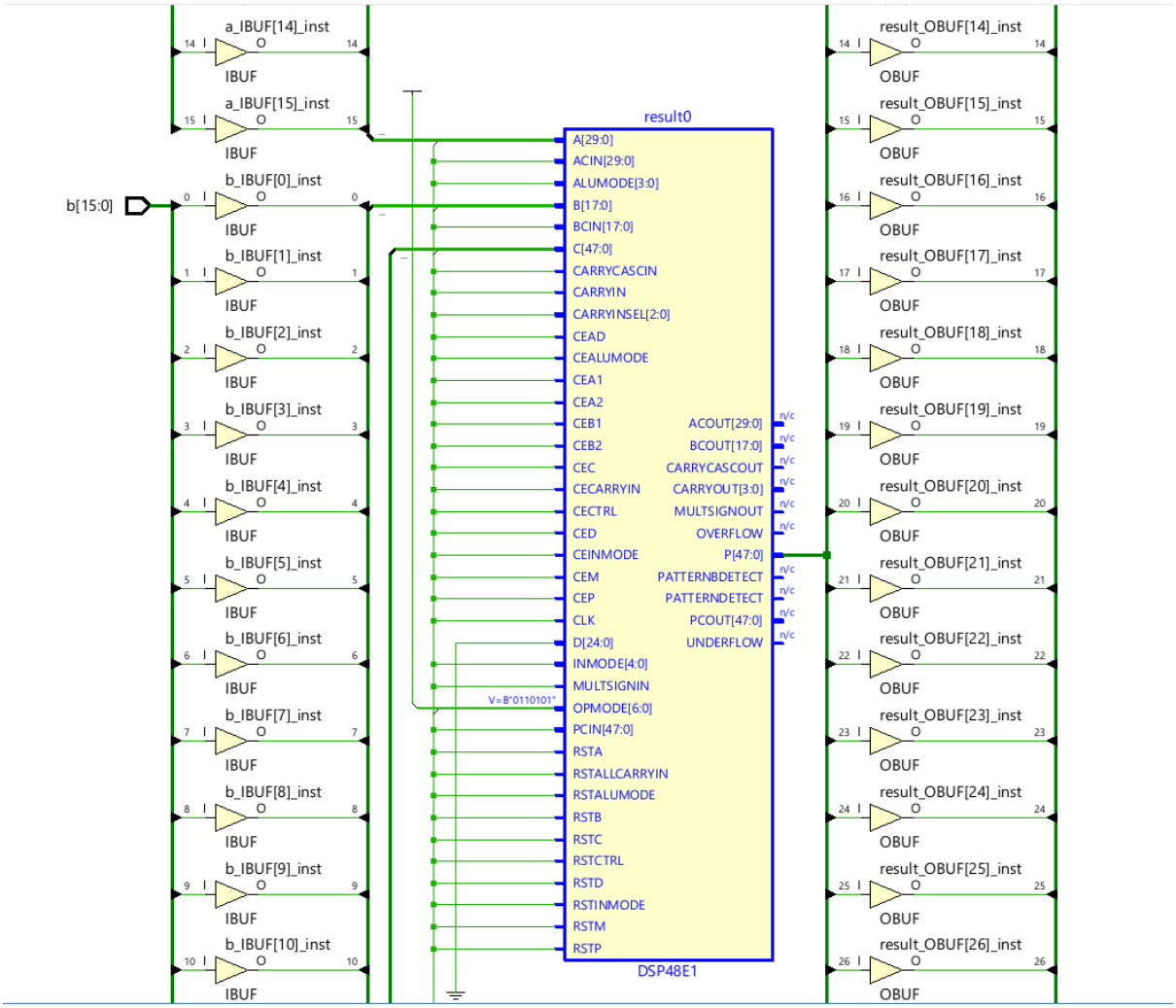
G. Schematic Diagram

- **Data_width = 8**



- **Data_width = 16**





H. Utilization

- **Data_width = 8**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	78	41000	0.19
IO	40	300	13.33

- **Data_width = 16**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
DSP	1	240	0.42
IO	80	300	26.67

4. Design 2 (with clk , reset and registering)

A. Implementation

This Verilog module represents a multiplier that computes the expression (a*b)+c.

This module is named dsp_mult and has a parameter WIDTH which determines the width of the input and output data. It includes clock (clk) and reset (rst) inputs along with AXI stream inputs (input_a_tdata, input_b_tdata, input_c_tdata) and their respective valid signals (input_a_tvalid, input_b_tvalid, input_c_tvalid). It also includes ready signals for each input stream (input_a_tready, input_b_tready, input_c_tready). Additionally, it has an AXI stream output (output_tdata, output_tvalid, output_tready).

Registers are declared to hold the current and previous values of input data (input_a_reg_0, input_a_reg_1, input_b_reg_0, input_b_reg_1, input_c_reg_0, input_c_reg_1) and output data (output_reg_0, output_reg_1).

A wire transfer is used to indicate that all input streams are valid and the output is ready to accept data.

Ready signals for input streams are assigned based on the validity of input data and the readiness of the output.

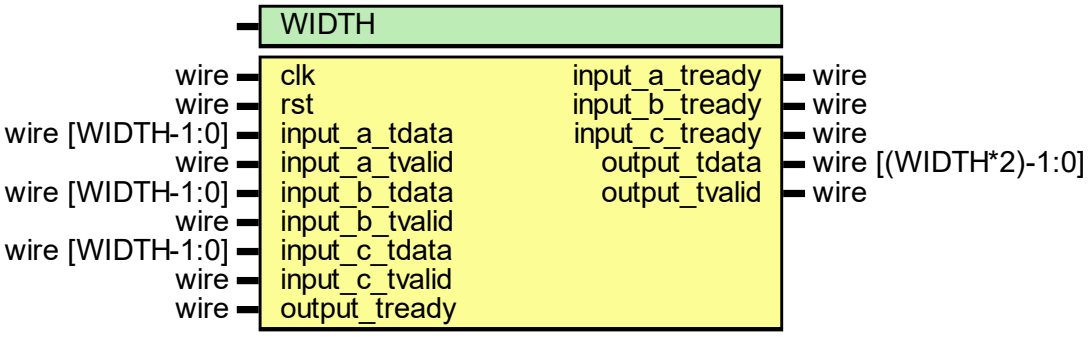
The output data and its validity are assigned based on the validity of all input streams.

Inside the always block, the module operates on the rising edge of the clock. When not in reset, if all input streams are valid and the output is ready, the pipeline and computation logic are executed.

The pipeline includes storing the current input data into registers, performing the multiplication and addition operation on the data, and storing the result into output registers.

Overall, this module implements a multiplier that computes (a*b)+c using AXI stream interfaces for input and output. It also includes pipelining to optimize performance.

B. Block diagram



C. Port Description

Generics

Generic name	Type	Value	Description
WIDTH		8 or 16	

Ports

Port name	Direction	Type	Description
clk	input	wire	Clock input for synchronous operation.
rst	input	wire	Reset input to initialize the circuit.
input_a_tdata	input	wire [WIDTH-1:0]	Data input for a
input_a_tvalid	input	wire	Valid signal for a
input_a_tready	output	wire	Ready signal for a
input_b_tdata	input	wire [WIDTH-1:0]	Data input for b

Port name	Direction	Type	Description
input_b_tvalid	input	wire	Valid signal for b
input_b_tready	output	wire	Ready signal for b
input_c_tdata	input	wire [WIDTH-1:0]	Data input for c
input_c_tvalid	input	wire	Valid signal for c
input_c_tready	output	wire	Ready signal for c
output_tdata	output	wire [(WIDTH*2)-1:0]	Data output for final signal
output_tvalid	output	wire	Valid signal for Output
output_tready	input	wire	Ready signal for Output

Signals

Name	Type	Description
input_a_reg_0 = 0	reg [WIDTH-1:0]	Registers for input 'a'.
input_a_reg_1 = 0	reg [WIDTH-1:0]	Registers for input 'a'.
input_b_reg_0 = 0	reg [WIDTH-1:0]	Registers for input 'b'.
input_b_reg_1 = 0	reg [WIDTH-1:0]	Registers for input 'b'.
input_c_reg_0 = 0	reg [WIDTH-1:0]	Registers for input 'c'.

Name	Type	Description
input_c_reg_1 = 0	reg [WIDTH-1:0]	Registers for input 'c'.
output_reg_0 = 0	reg [(WIDTH*2)-1:0]	Registers for output.
output_reg_1 = 0	reg [(WIDTH*2)-1:0]	Registers for output.
transfer = input_a_tvalid & input_b_tvalid &input_c_tvalid & output_tready	wire	Control signal indicating that valid data is available for all inputs and the output is ready to accept data.

D. Module code

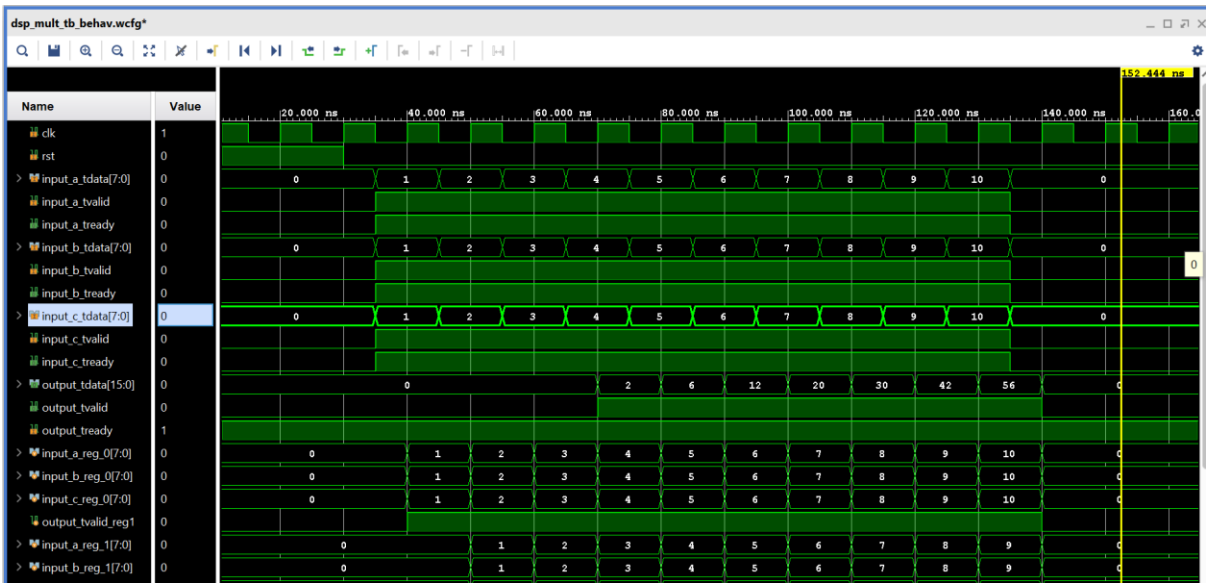
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/Design_2/dsp_mul.v

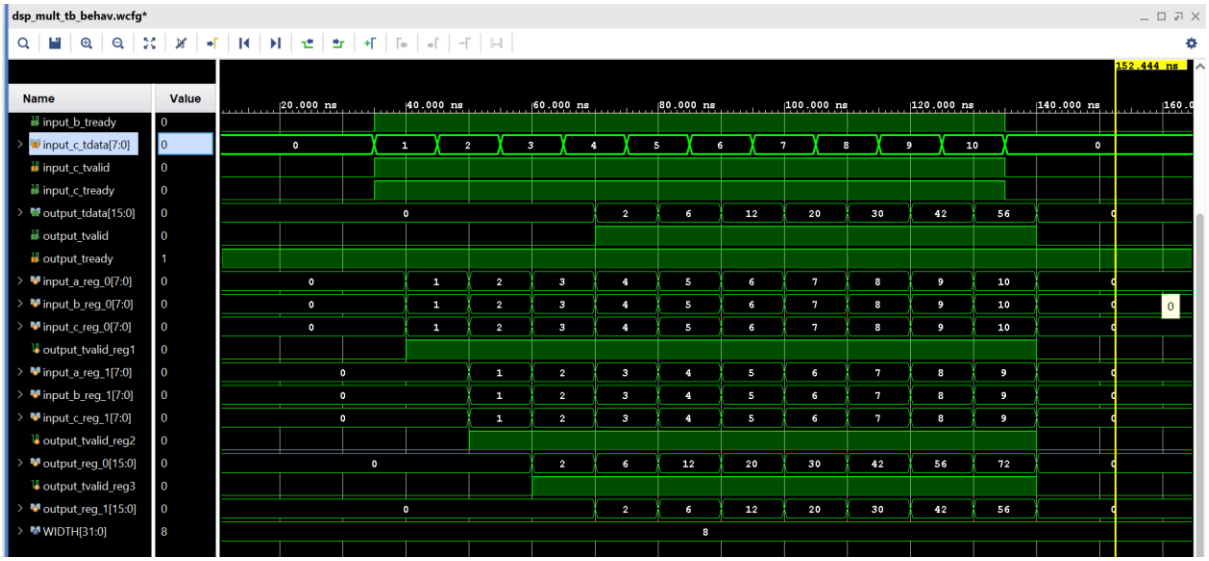
E. Test bench code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/Design_2/dsp_mul_tb.v

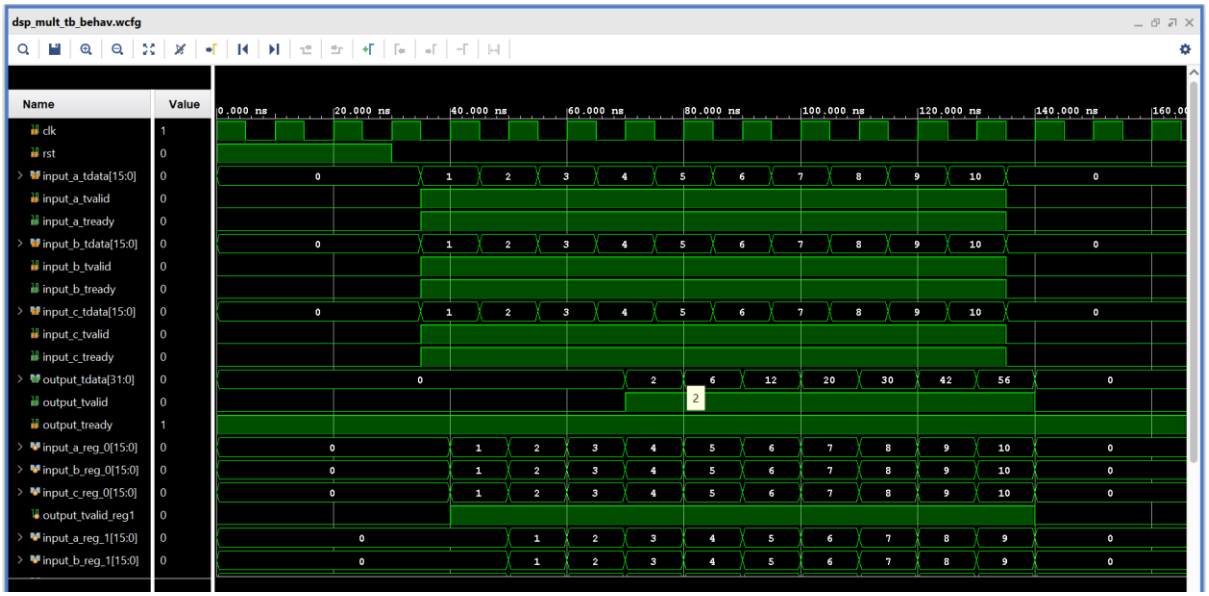
F. Test cases

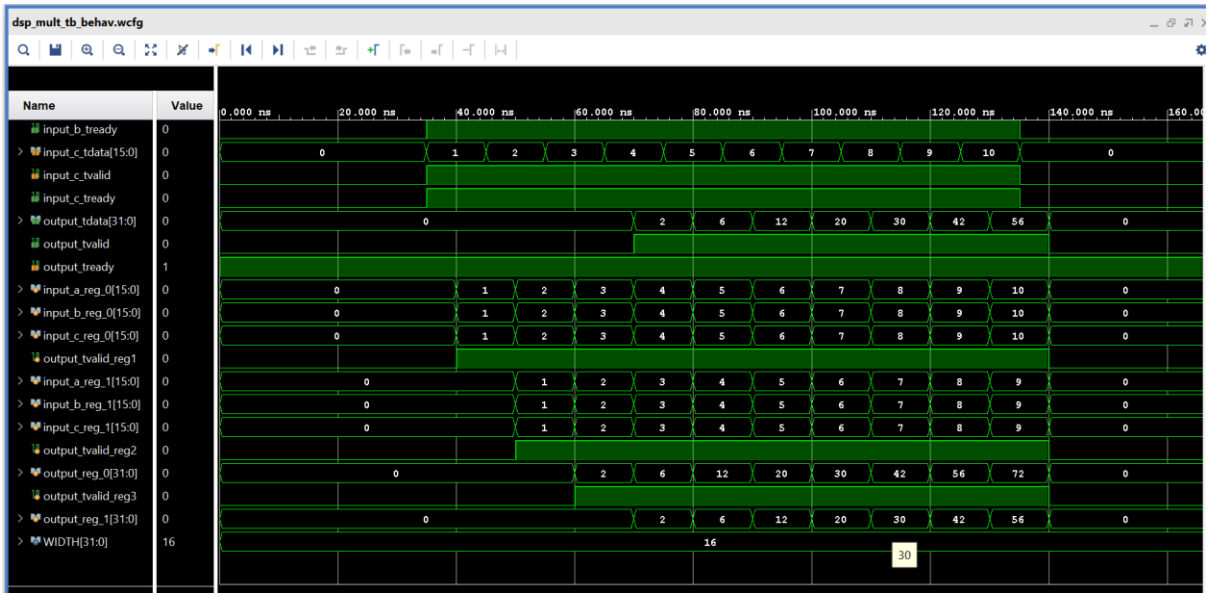
- **Data_width = 8**





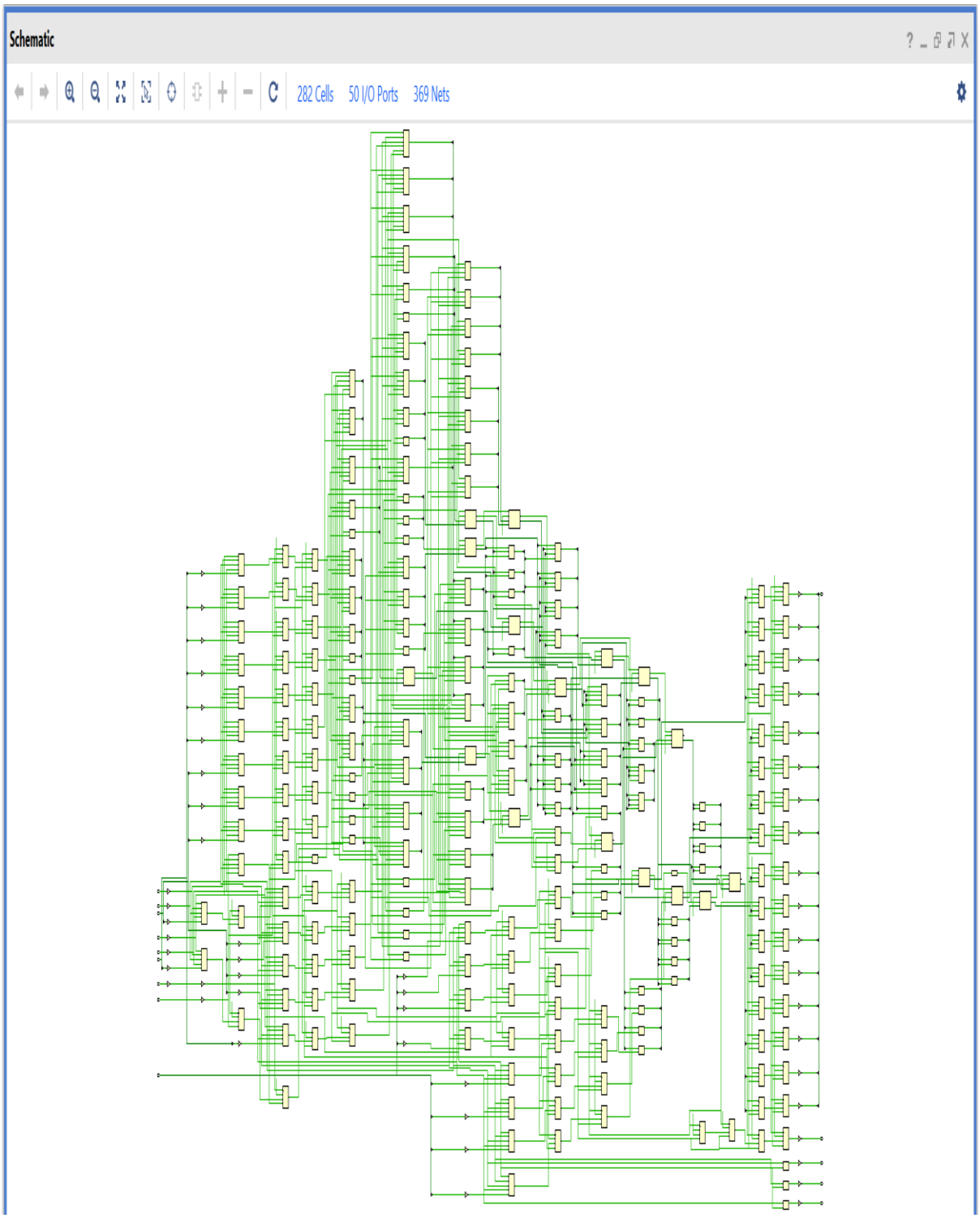
- **Data_width = 16**



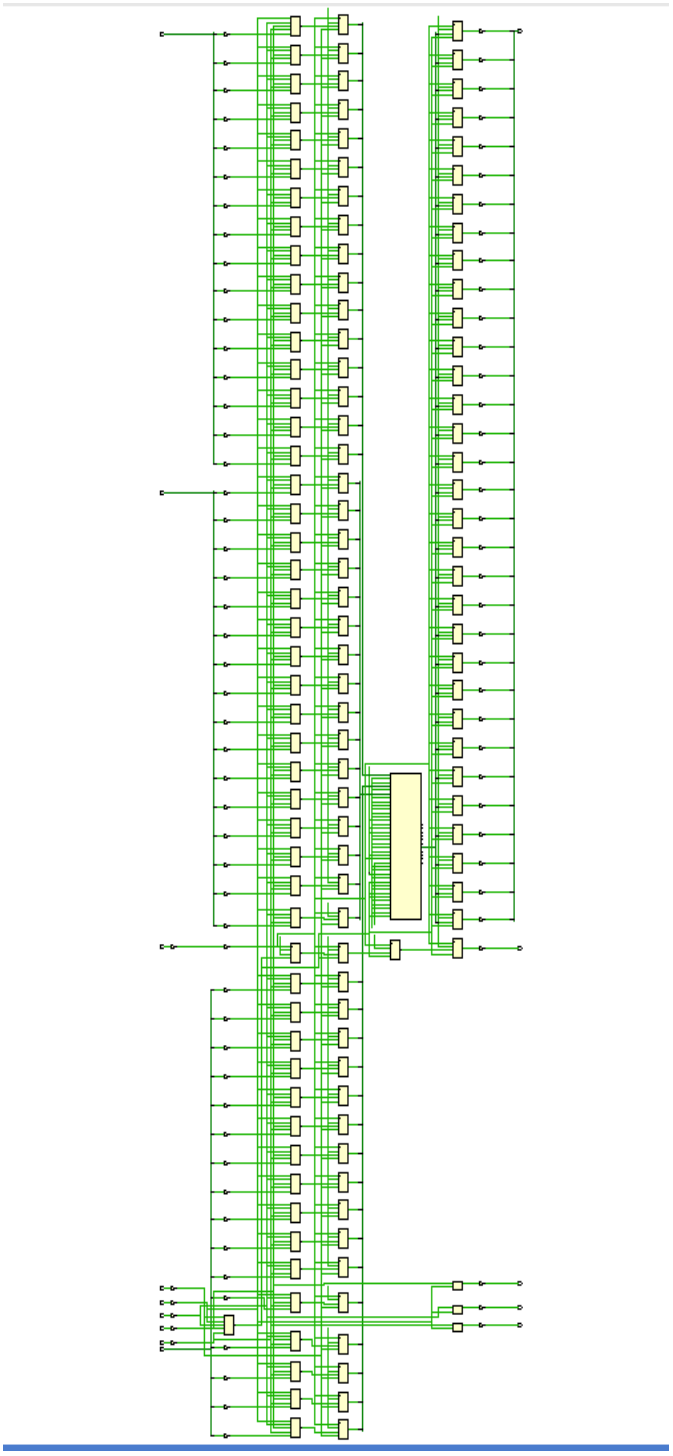


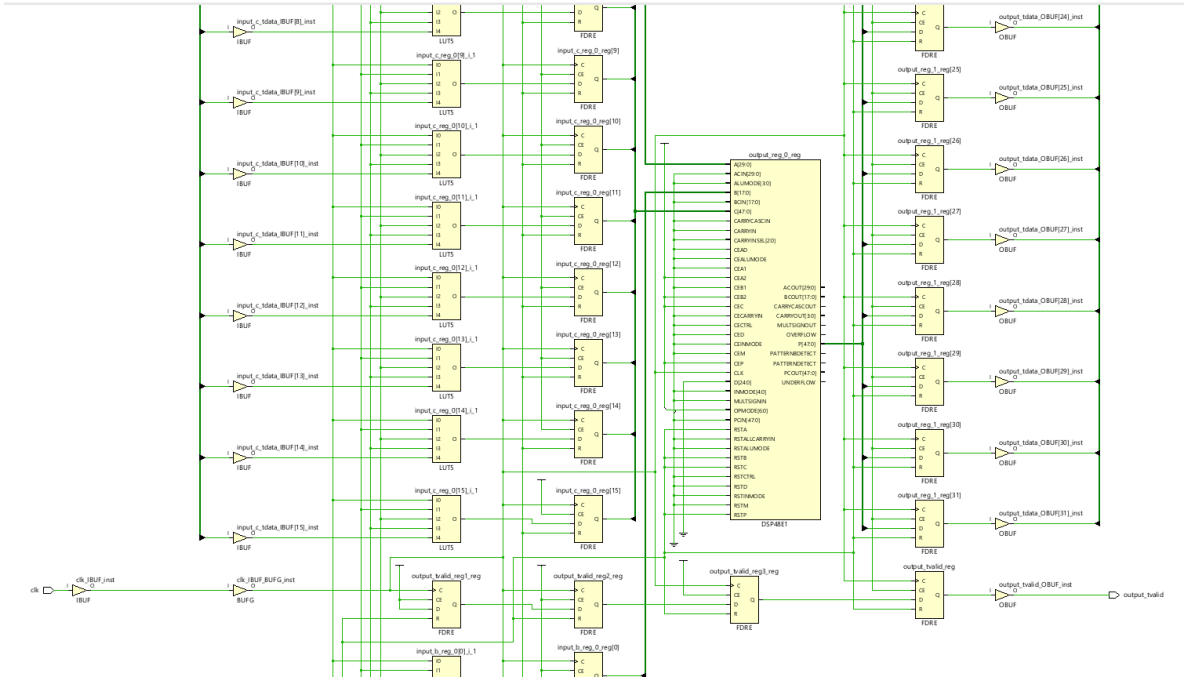
G. Schematic Diagram

- **Data_width = 8**



- **Data_width = 16**





H. Utilization

- **Data_width = 8**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	101	41000	0.25
FF	84	82000	0.10
IO	50	300	16.67
BUFG	1	32	3.13

- **Data_width = 16**

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

Graph | **Table**

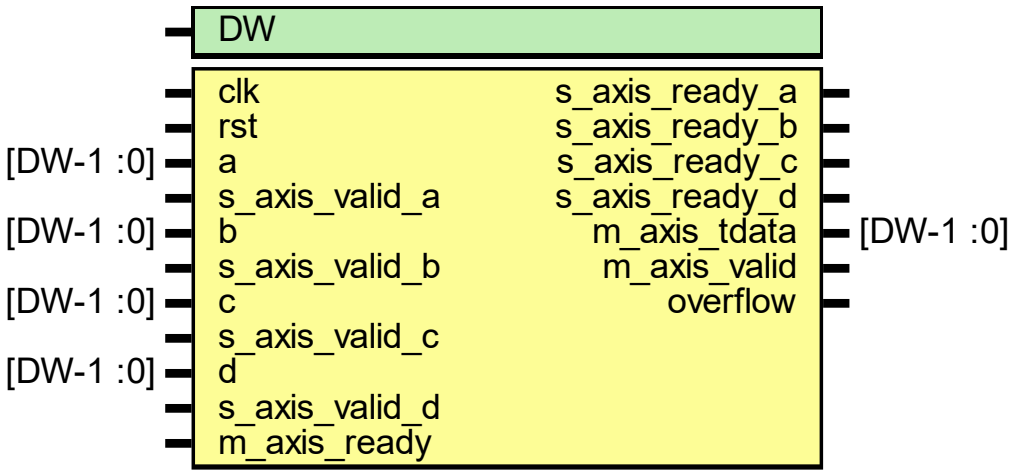
Resource	Utilization	Available	Utilization %
LUT	49	41000	0.12
FF	84	82000	0.10
DSP	1	240	0.42
IO	90	300	30.00
BUFG	1	32	3.13

5. Design 3 (with overflow condition)

A. Implementation

- On each rising edge of the clock (clk), if the reset signal (rst) is asserted, the internal registers are reset to zero, and output valid signal (m_axis_valid) is de-asserted.
- When not in reset state, if all input valid signals (s_axis_valid_a, s_axis_valid_b, s_axis_valid_c, s_axis_valid_d) are asserted and the corresponding ready signals (s_axis_ready_a, s_axis_ready_b, s_axis_ready_c, s_axis_ready_d) are also asserted, the module computes (a-d)*b + c using pipelined registers.
- The computed result is made available on output m_axis_tdata, and the output valid signal m_axis_valid is asserted when all inputs are valid and ready.
- The overflow output indicates whether overflow occurred during the computation.
- The module also controls the readiness of inputs (s_axis_ready_a, s_axis_ready_b, s_axis_ready_c, s_axis_ready_d) based on the readiness of the output interface (m_axis_ready).

B. Block diagram



C. Port Description

Generics

Generic name	Type	Value	Description
DW		16	Data width, specifying the bit width of input and output data.

Ports

Port name	Direction	Type	Description
clk	input		Clock signal.
rst	input		Reset signal.
a	input	[DW-1 :0]	Input operand a.
s_axis_valid_a	input		Valid signal for input a.
s_axis_ready_a	output		Ready signal for input a.
b	input	[DW-1 :0]	Input operand b.
s_axis_valid_b	input		Valid signal for input b.
s_axis_ready_b	output		Ready signal for input b.
c	input	[DW-1 :0]	Input operand c.

Port name	Direction	Type	Description
s_axis_valid_c	input		Valid signal for input c.
s_axis_ready_c	output		Ready signal for input c.
d	input	[DW-1 :0]	Input operand d.
s_axis_valid_d	input		Valid signal for input d.
s_axis_ready_d	output		Ready signal for input d.
m_axis_tdata	output	[DW-1 :0]	Output data containing the result of (a-d)*b + c.
m_axis_valid	output		Valid signal indicating the availability of output data.
m_axis_ready	input		Input ready signal
overflow	output		Signal indicating overflow in the computation result.

Signals

Name	Type	Description
reg1	logic signed [DW :0]	containing the result of (a-d).
reg2	logic signed [DW :0]	containing the result of reg3+c.
reg3	logic signed [2*DW-1:0]	containing the result of reg1*b.

D. Module code

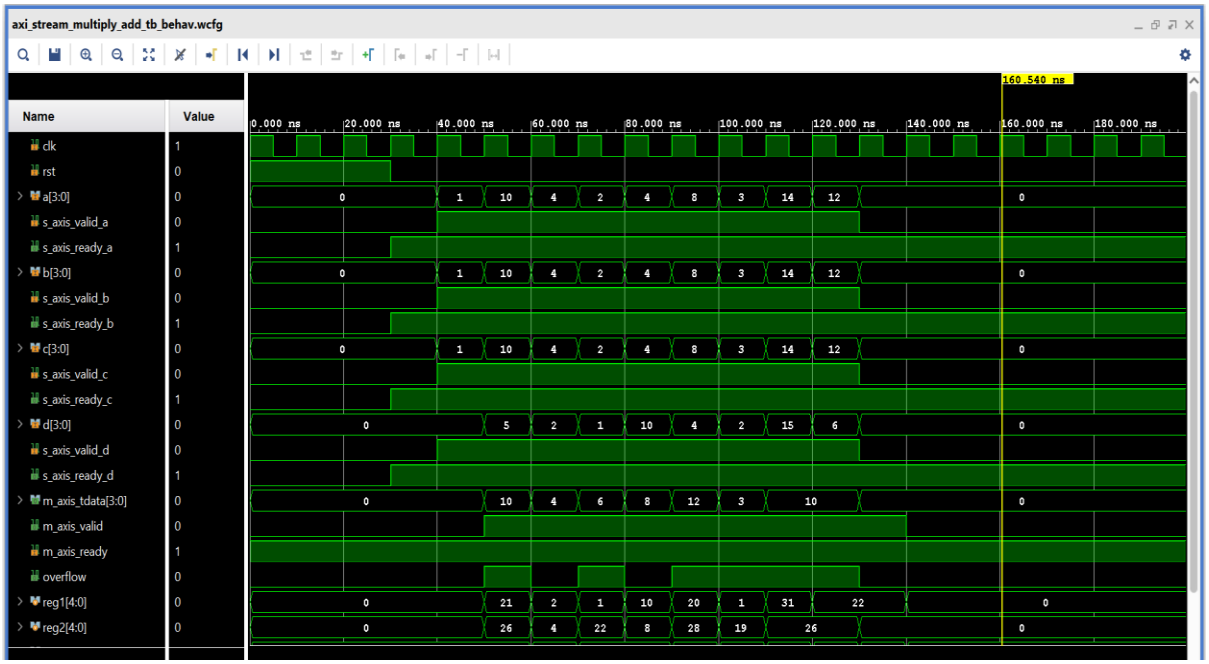
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/Design_3/dsp_design_3.sv

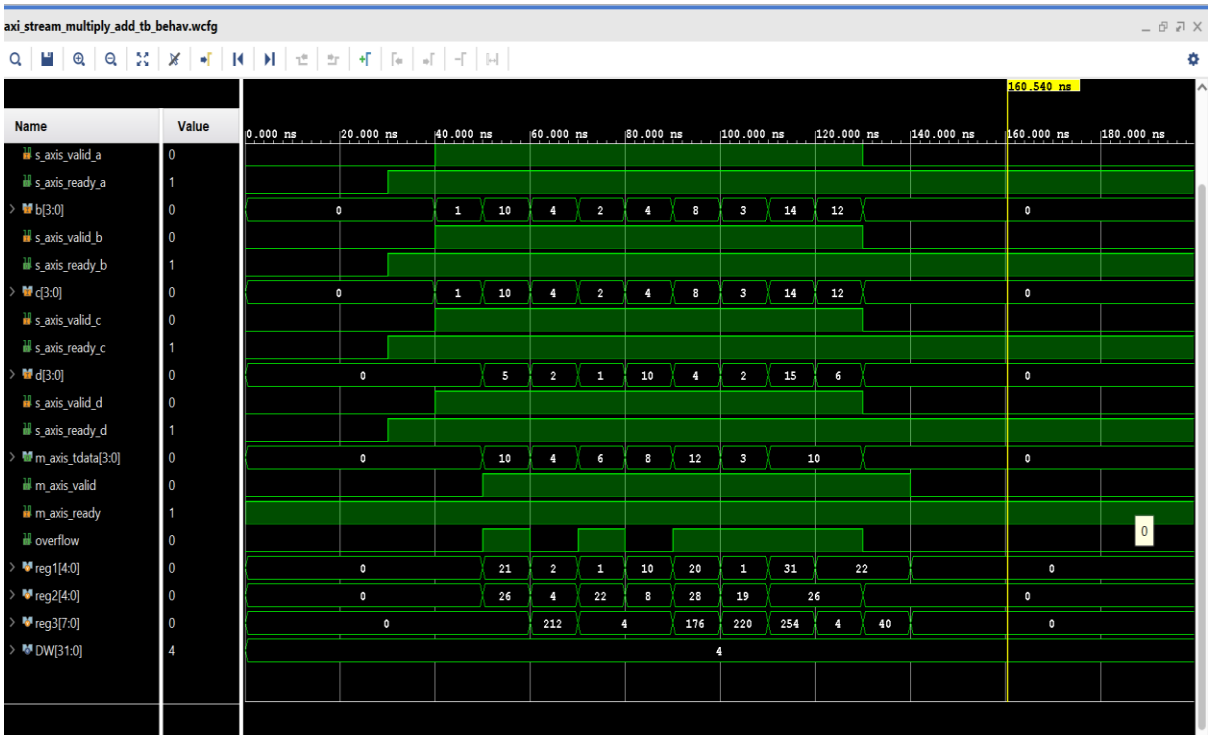
E. Test bench code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/Design_3/design3_tb.sv

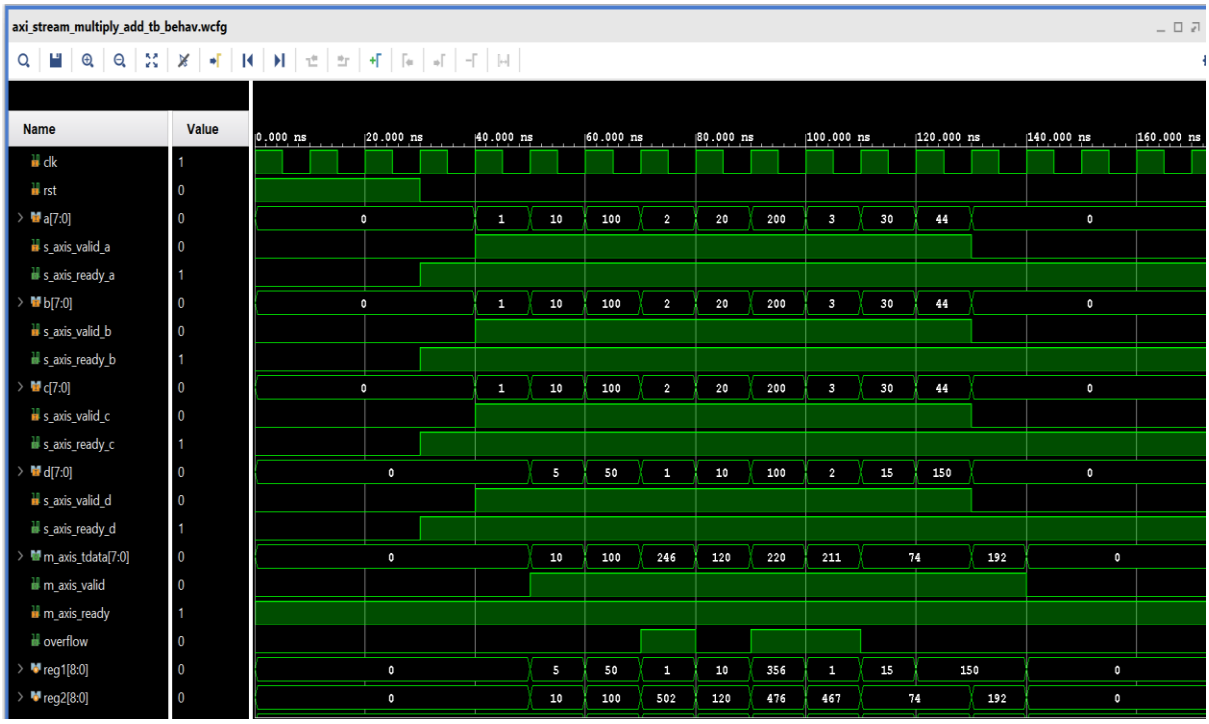
F. Test cases

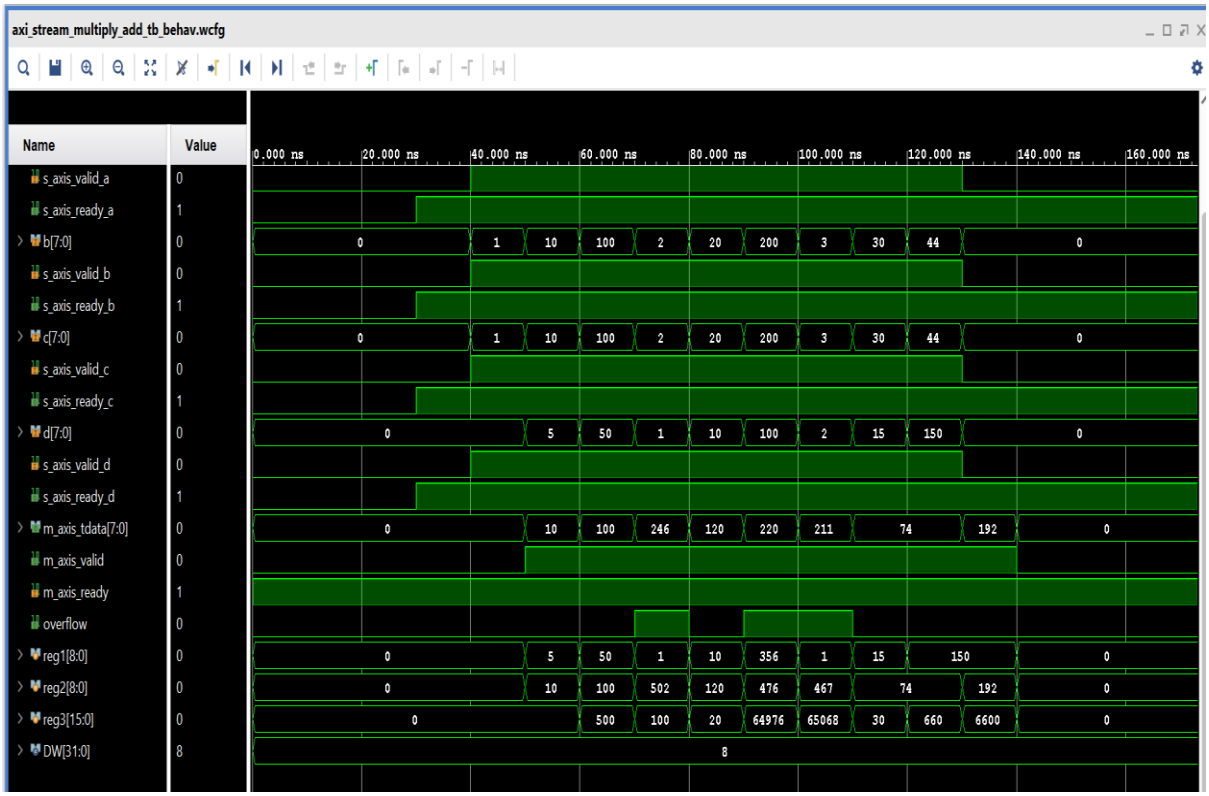
- **Data_width = 4**





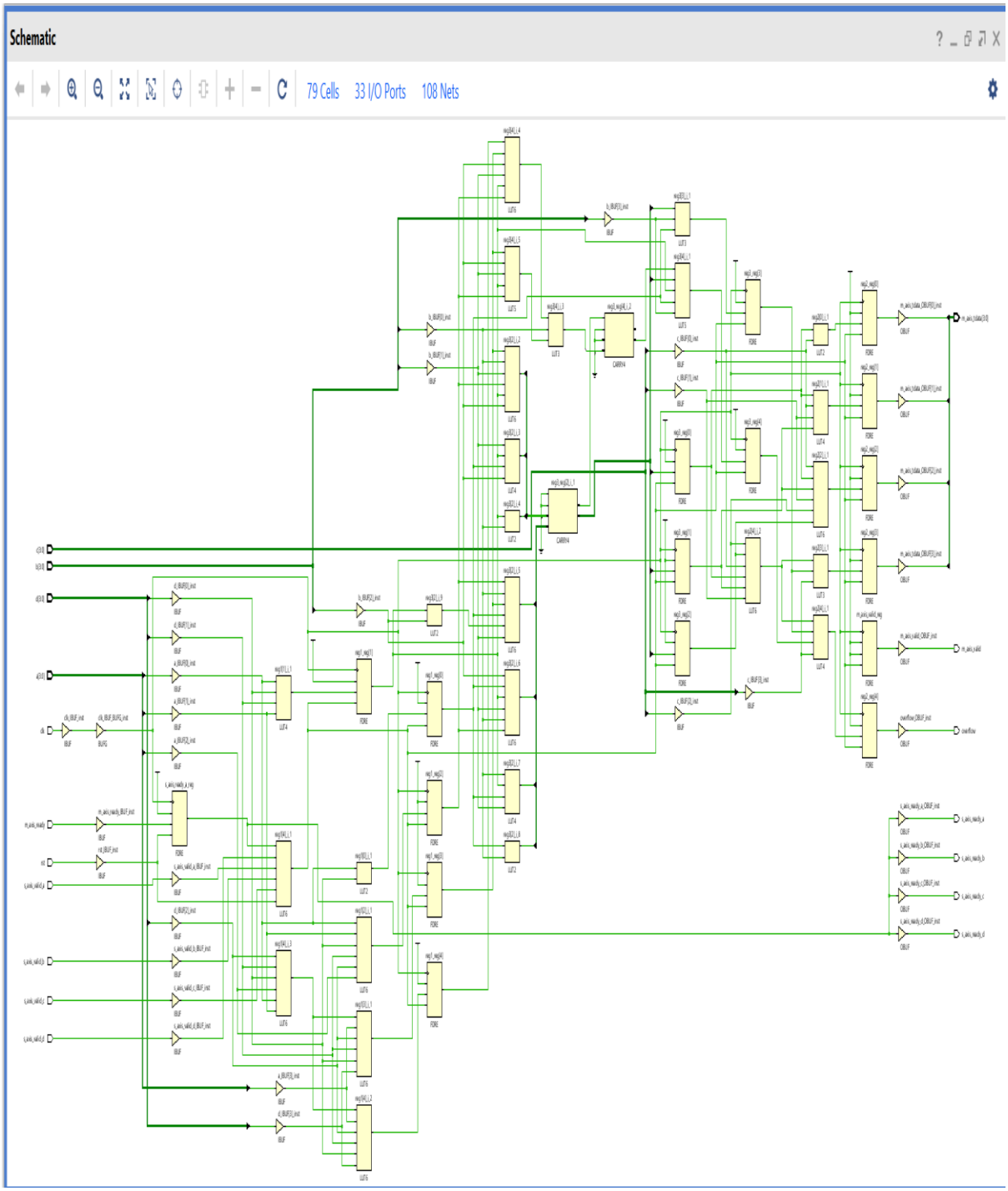
- **Data_width = 8**





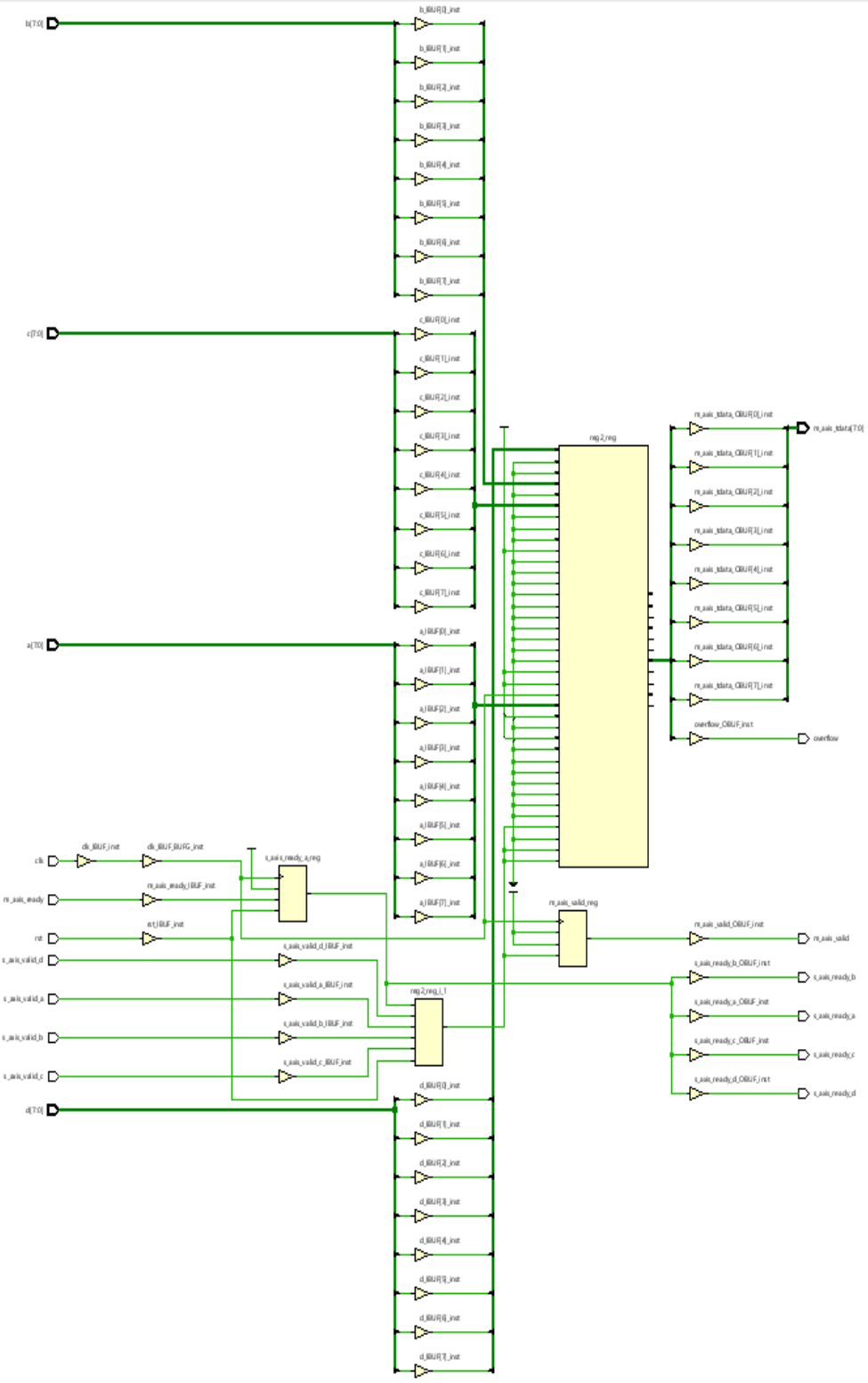
G. Schematic Diagram

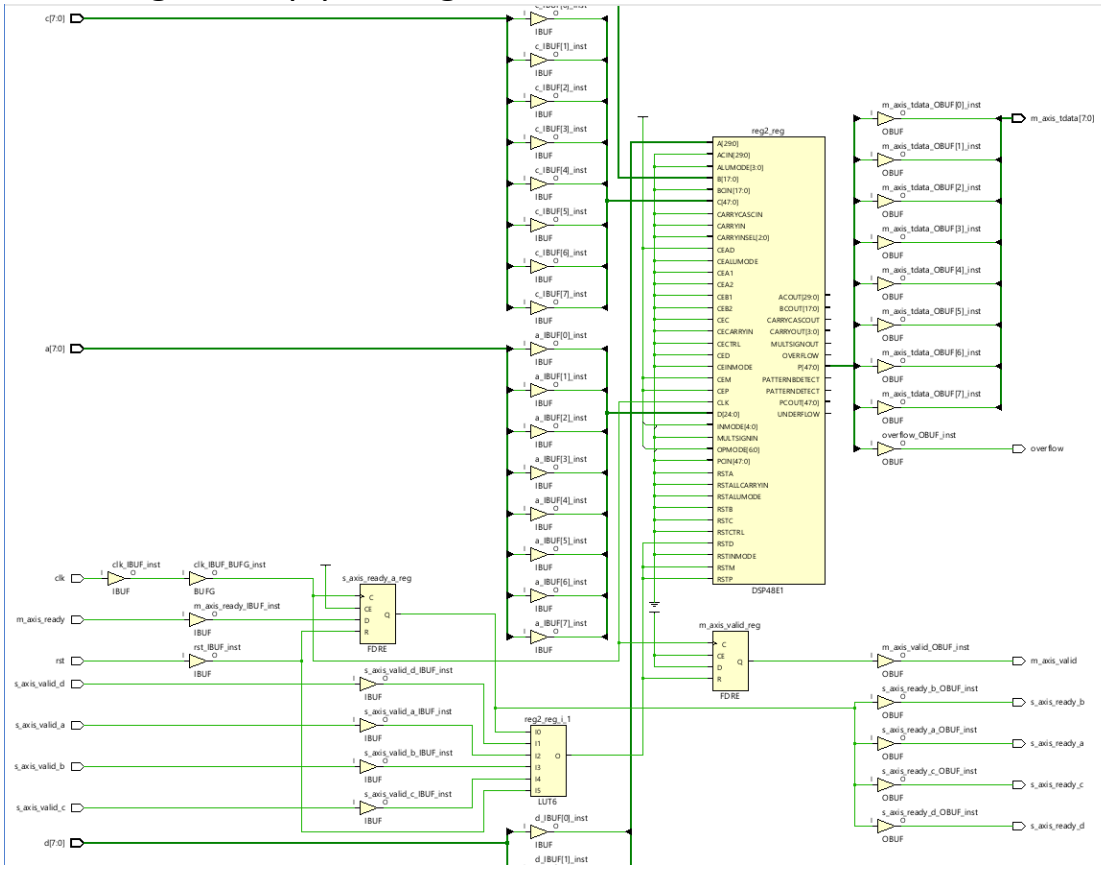
- **Data_width = 4**



RTL Design with pipelining

- **Data_width = 8**





H. Utilization

- **Data_width = 4**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	20	41000	0.05
FF	17	82000	0.02
IO	33	300	11.00
BUFG	1	32	3.13

- **Data_width = 8**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1	41000	0.01
FF	2	82000	0.01
DSP	1	240	0.42
IO	53	300	17.67
BUFG	1	32	3.13

6. Design-4 (pattern detect)

A. Implementation

PATTERNDETECT and PATTERNBDETECT Logic:

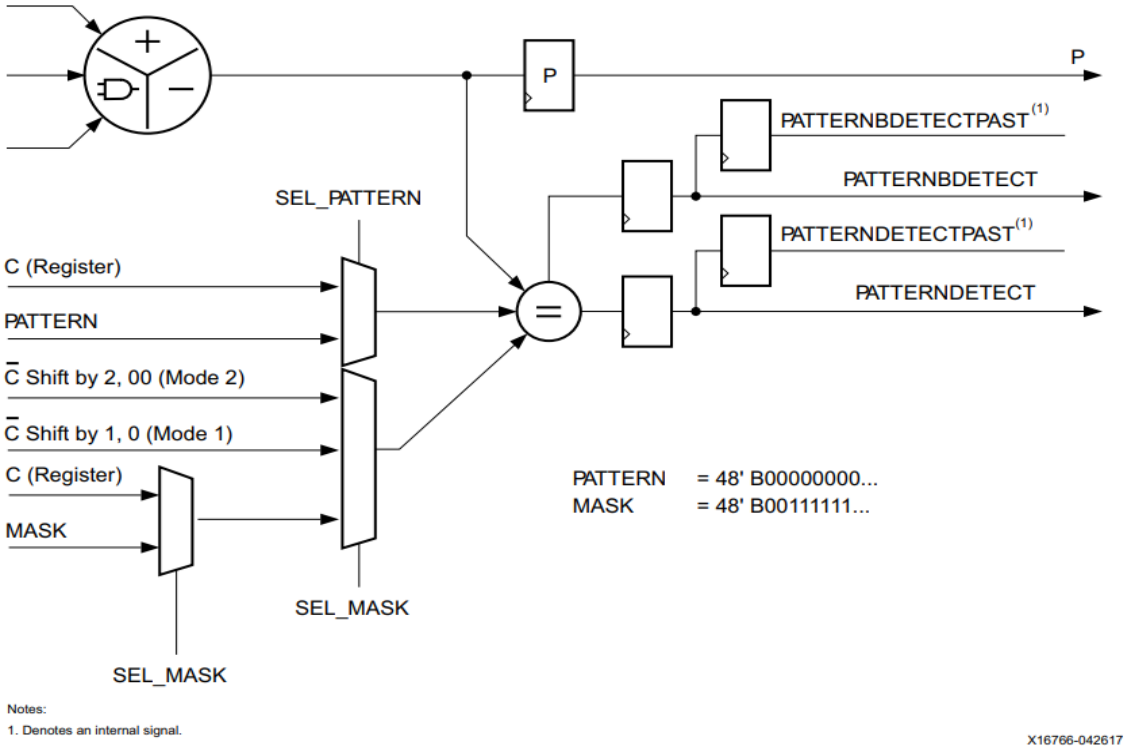
A pattern detector on the output of the DSP48E2 slice detects if the P bus matches a specified pattern or if it exactly matches the complement of the pattern. The PATTERNDETECT output goes High if the output of the adder matches a set pattern. The PATTERNBDETECT output goes High if the output of the adder matches the complement of the set pattern.

A mask field can also be used to hide certain bit locations in the pattern detector. PATTERNDETECT computes ((P == pattern)||mask) on a bitwise basis and then ANDs the results to a single output bit. Similarly, PATTERNBDETECT can detect if ((P == ~pattern)||mask).

The pattern and the mask fields can each come from a distinct 48-bit configuration field or from the (registered) C input. When the C input is used as the PATTERN, the OPMODE should be set to select a 0 at the input of the Z multiplexer. If all the registers are reset, PATTERNDETECT is High for one clock cycle immediately after the RESET is deasserted.

The pattern detector allows the DSP48E2 slice to support convergent rounding and counter auto reset when a count value has been reached as well as support overflow, underflow, and saturation in accumulators.

The pattern detector is best described as an equality check on the output of the adder/subtracter/logic unit that produces its result on the same cycle as the P output. There is no extra latency between the pattern detect output and the P output of the DSP48E2 slice. The use of the pattern detector leads to a moderate speed reduction due to the extra logic on the pattern detect path.



Some of the applications that can be implemented using the pattern detector are:

• Pattern detect with optional mask

• Dynamic C input pattern match with A x B

• Overflow/underflow/saturation past P[46]

• A:B == C and dynamic pattern match, e.g., A:B OR C == 0, A:B AND C == 1

• A:B {function} C == 0

• 48-bit counter auto reset (terminal count detection) with option for CEP priority

• Detecting mid points for rounding operations

If the pattern detector is not being employed, it can be used for other creative design implementations. These include:

• Duplicating a pin (e.g., the sign bit) to reduce fanout and thus increase speed.

• Implementing a built-in inverter on one bit (e.g., the sign bit) without having to route out to the CLBs.

• Checking for sticky bits in floating point, handling special cases, or monitoring the DSP48E2 slice outputs.

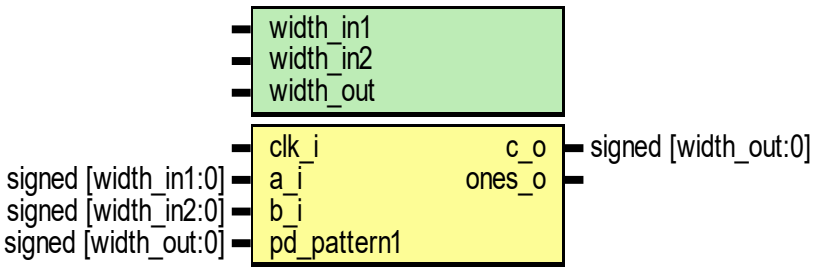
• Raising a flag if a certain condition is met or if a certain condition is no longer met.

A mask field can also be used to mask out certain bit locations in the pattern detector. The pattern field and the mask field can each come from a distinct 48-bit memory cell field or from the (registered) C input.

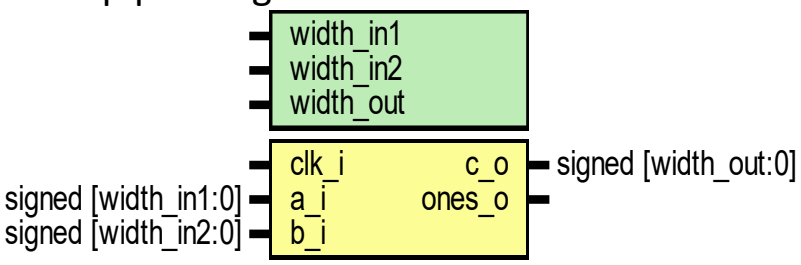
Note : I observed pattern detect pin in one case, but I am not sure complete understating of that logic. I tried with different cases and observed different Utilizations. I will try to understand that logic completely.

B. Block diagram

- **Pattern given from input side**



- **Pattern given inside the code through assign with some value.**



C. Port description

- Port description is similar all cases with small changes

Generics

Generic name	Type	Value	Description
width_in1		24	Input a width
width_in2		17	Input b width
width_out		42	Pattern and output width

Ports

Port name	Direction	Type	Description
clk_i	input		Clock signal.
a_i	input	signed [width_in1:0]	Input a.
b_i	input	signed [width_in2:0]	Input b.

Port name	Direction	Type	Description
pd_pattern1	input	signed [width_out:0]	Input pattern
c_o	output	signed [width_out:0]	Output signal after multiplication of a and b.
ones_o	output		Gives pattern detects output.

Signals

Name	Type	Description
ab	reg signed [width_out:0]	Multiplication from input a and b.
a_reg	reg [width_in1:0]	Registering input a.
b_reg	reg [width_in2:0]	Registering input b.
pd_pattern	reg [width_out:0]	Registering input pattern.
mask	wire [width_out:0]	Internal passing mask value.
mask1=1	wire	Internal passing mask value.

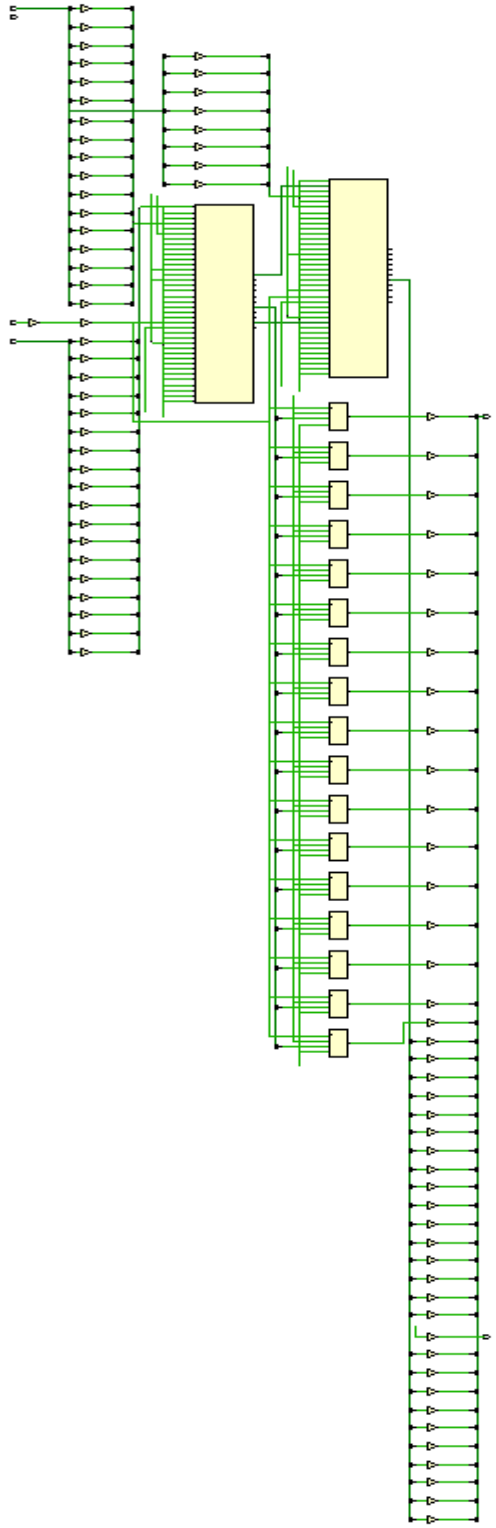
D. Module code

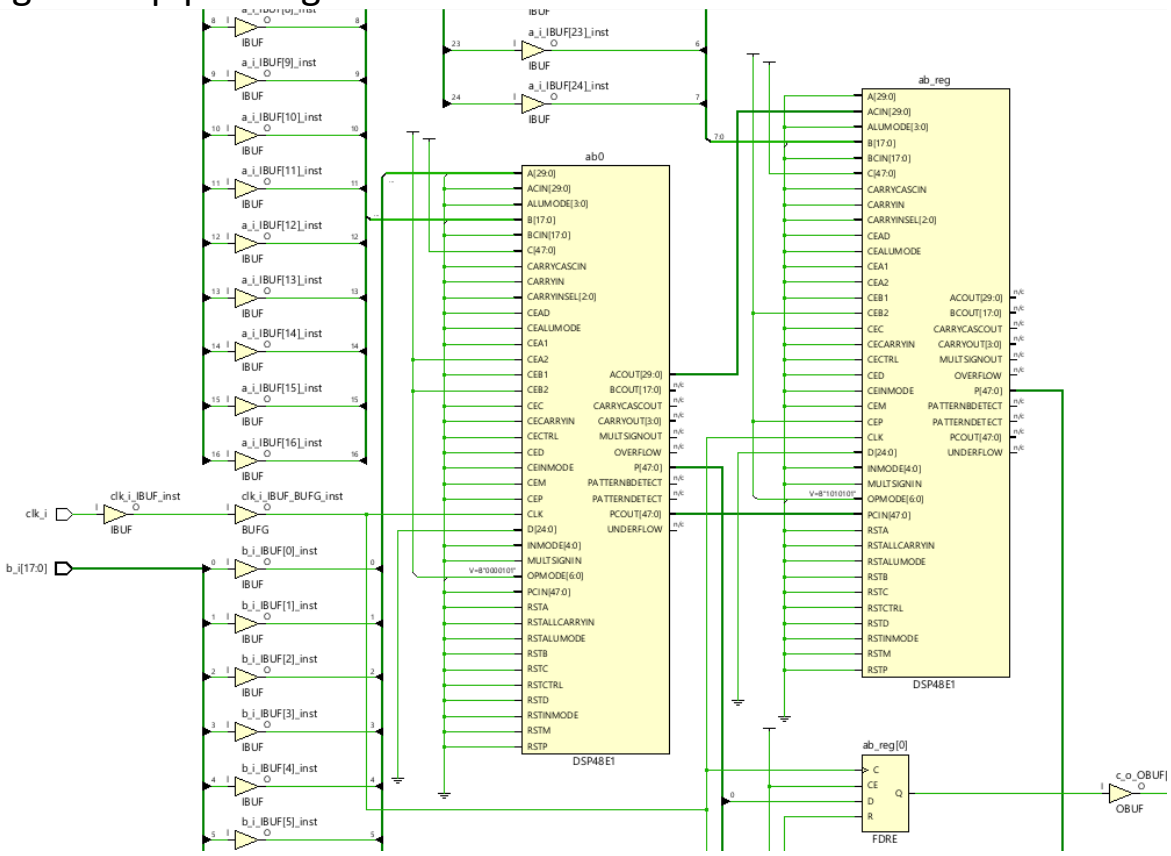
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_1/pattern_detect/comparator.v

E. Schematic Diagram

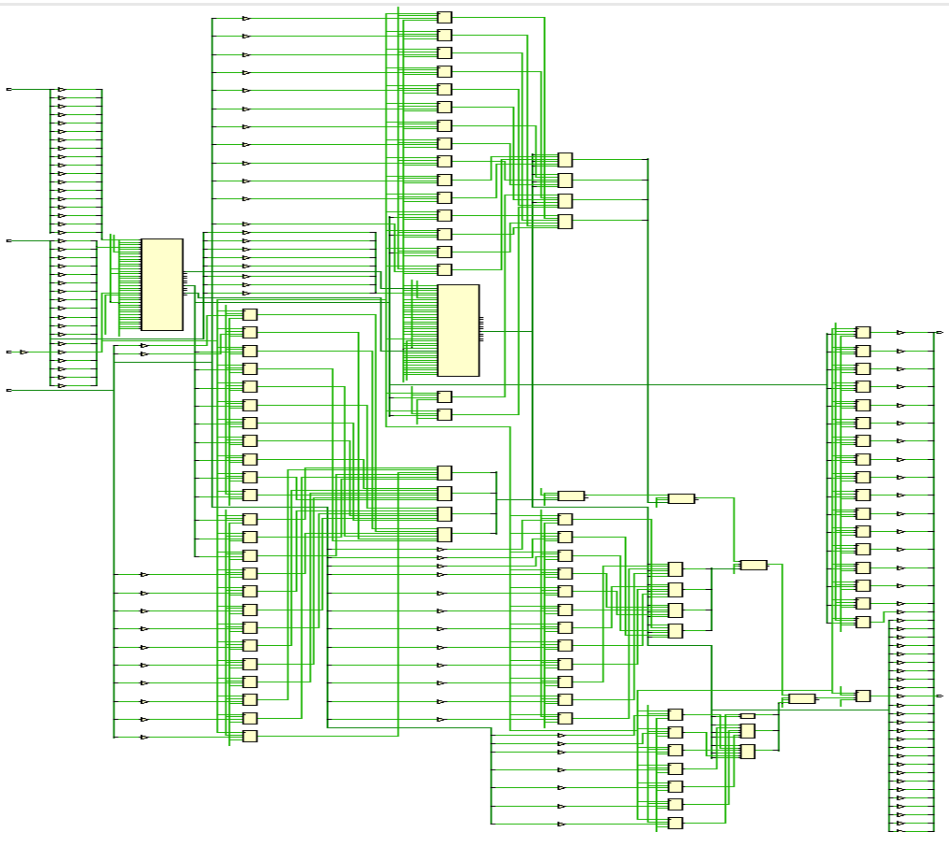
- **Pattern given from input side with mask.**

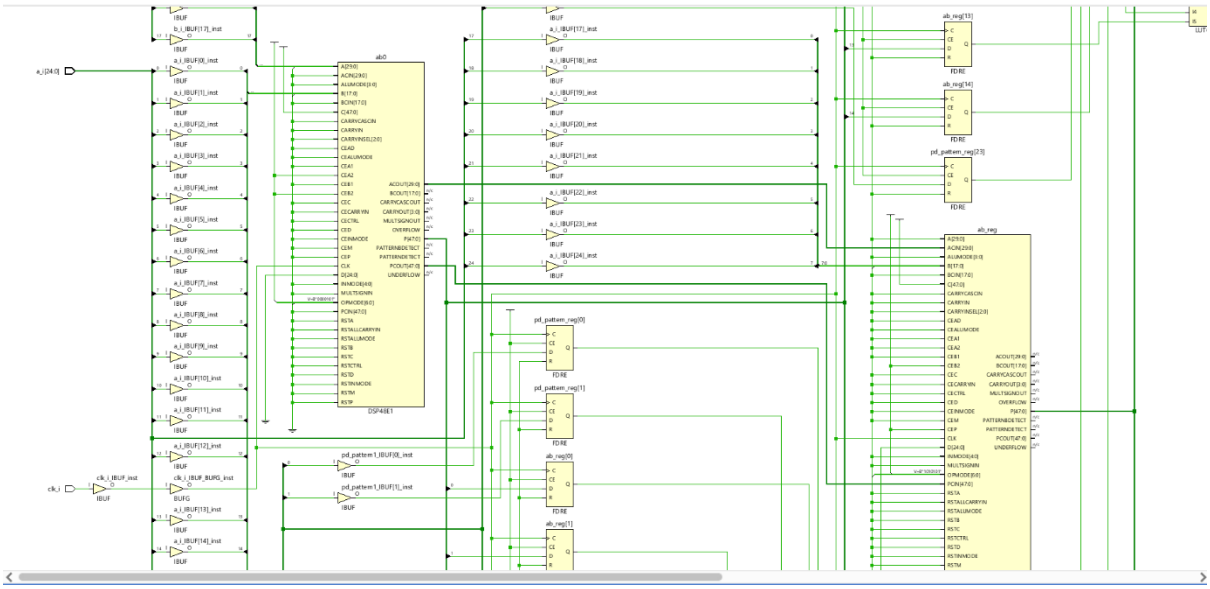
RTL Design with pipelining



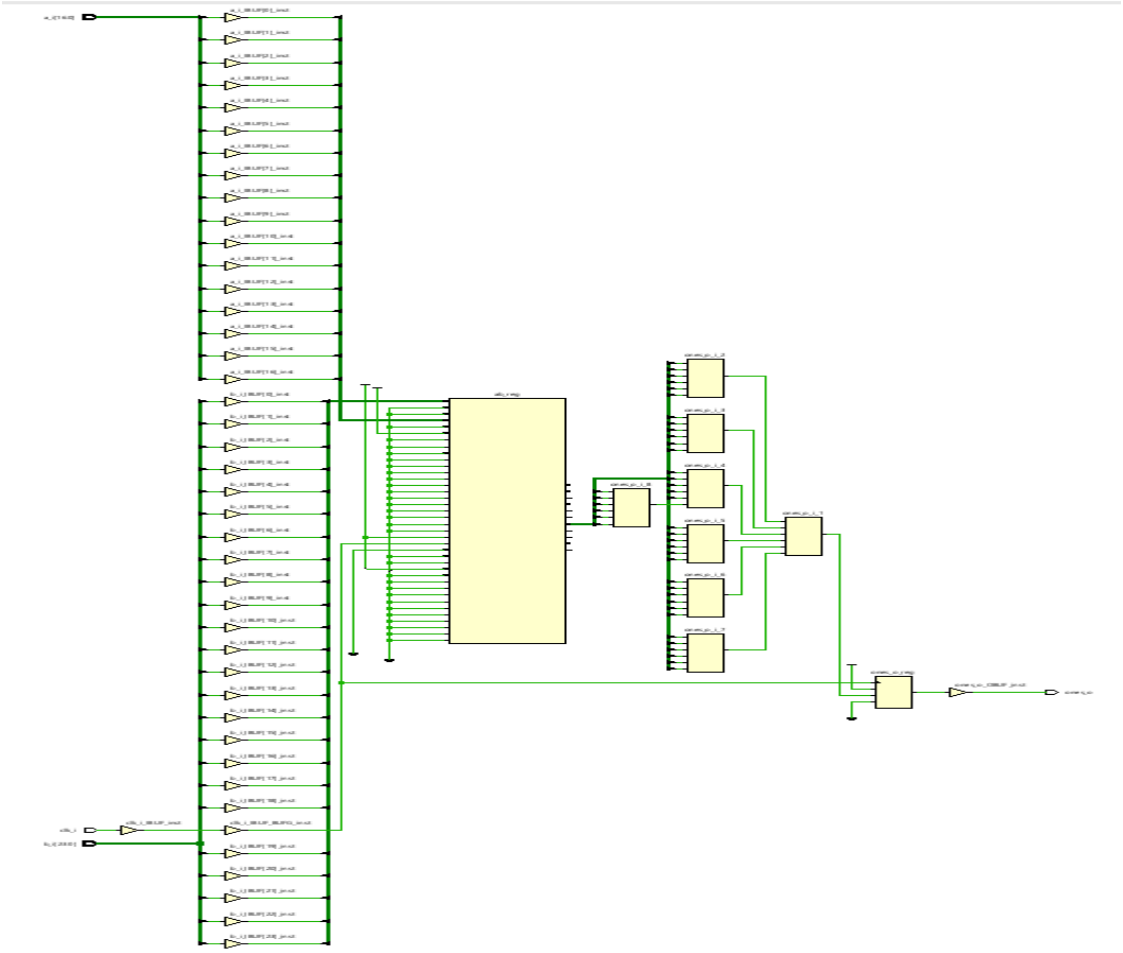


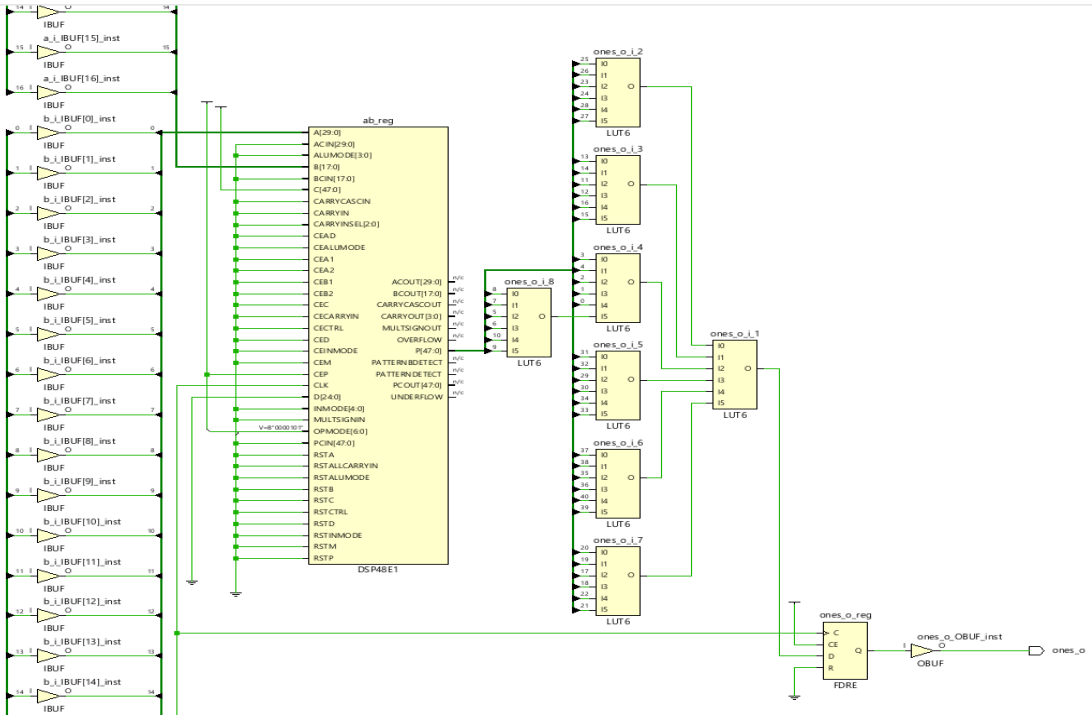
- **Pattern given from input side without mask.**



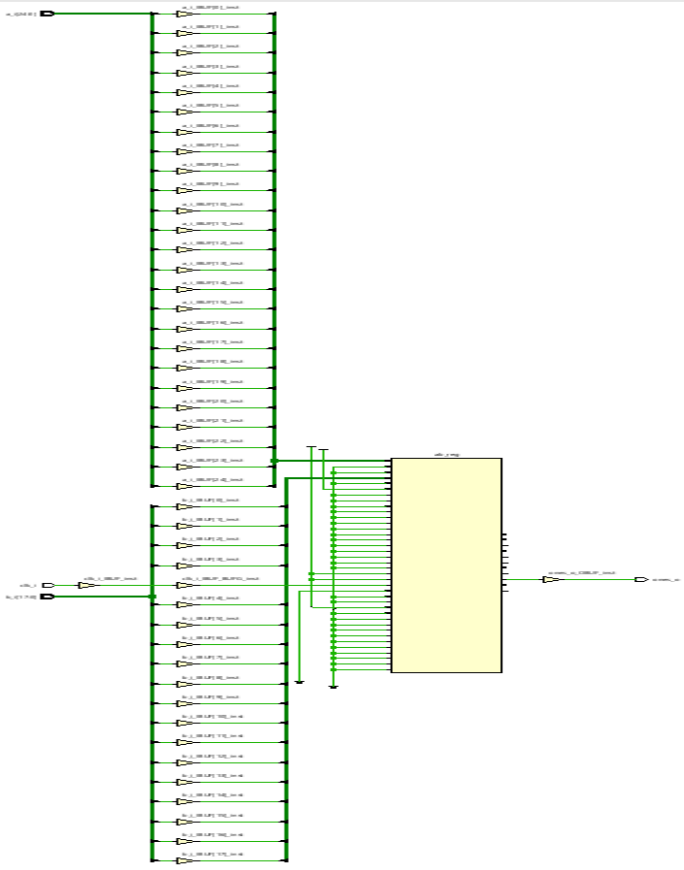


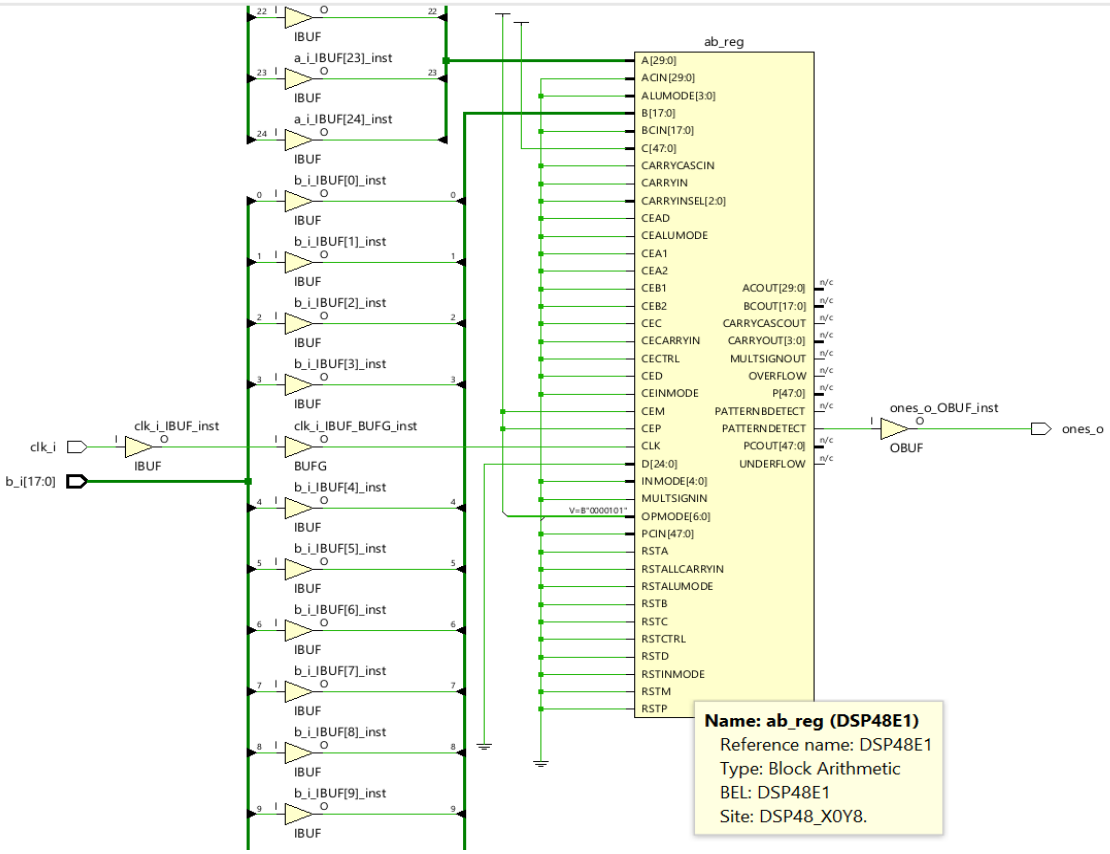
- **Pattern given inside the code through assign with some value.**





- **Pattern detect pin example.**





F. Utilization

- **Pattern given from input side with mask.**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
FF	17	82000	0.02
DSP	2	240	0.83
IO	88	300	29.33
BUFG	1	32	3.13

- **Pattern given from input side without mask.**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	15	41000	0.04
FF	78	82000	0.10
DSP	2	240	0.83
IO	131	300	43.67
BUFG	1	32	3.13

- **Pattern given inside the code through assign with some value.**

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	8	41000	0.02
FF	1	82000	0.01
DSP	1	240	0.42
IO	43	300	14.33
BUFG	1	32	3.13

- **Pattern detect pin example.**

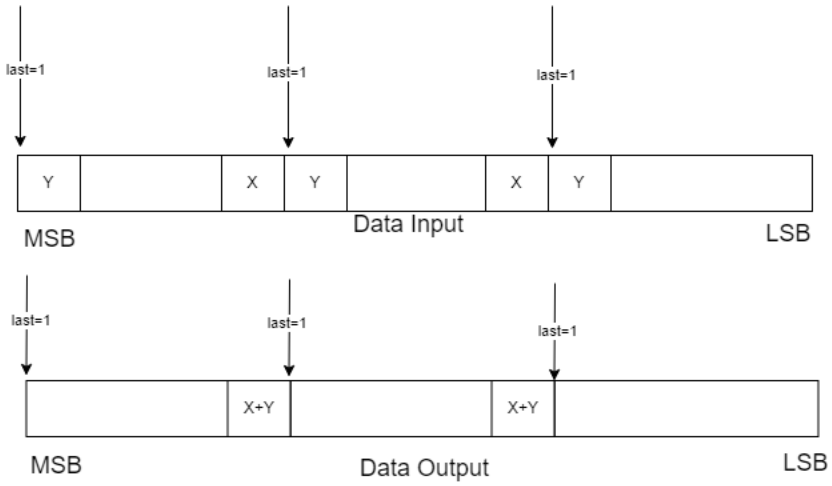
Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
DSP	1	240	0.42
IO	45	300	15.00
BUFG	1	32	3.13

Assignment-2

- Understand the below figure.

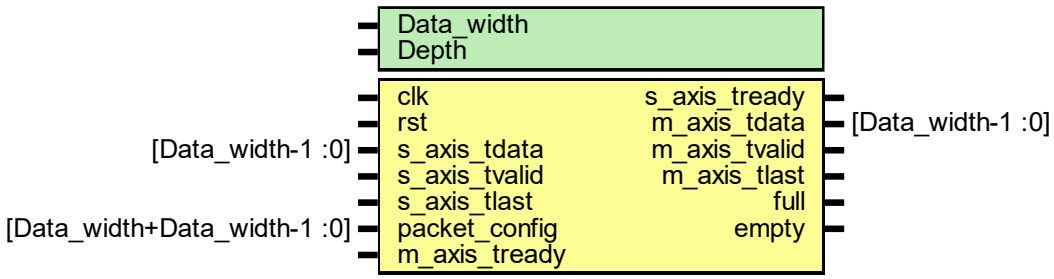


- Input data packet considerations:
  - One input data packet can have an “n” number of data bytes and the end of the packet would be specified with a “last” signal as illustrated in the above figure.
  - Design should accept one configuration input for every input data packet and that configuration specifies “k” value which will be used for packet processing.
  - There should not be any IDLE clock cycles between configuration reception and data reception.
  - Design should receive the data at a rate of one byte per clock cycle.
  - Also assume that there are continuous streams of input packets without any IDLE clock cycles in between them.
  - LSB will comes first to the design
- Packet processing:
  - For example, if k = 2 for a packet, 2 bytes from that corresponding packet should be taken and added to the 2 bytes of the next immediate packet.
  - ‘y’ indicates the last ‘k’ number of samples in a packet.
  - Resulted bytes should be placed at the ‘x’ samples position and send the next packet as output.
  - ‘x’ and ‘y’ represent the ‘k’ number of samples in corresponding packets.
  - Remaining samples need to be sent as it is.
- Design should have as much as less latency and initiation interval as possible.
- The design should comply with the AXI Stream Protocol.

1) Implementation

- On reset (rst), the module initializes all internal registers and memories.
- The input data (s_axis_tdata) is written into the buffer (mem_1) along with the corresponding last data indicator (s_axis_tlast).
- Additional data for partial packets is stored in mem_2.
- The read pointer (rd_ptr) controls the output data retrieval from the buffer.
- Output data (m_axis_tdata) is read from mem_1 and mem_2 and sent out through the output streaming interface when the interface is ready (m_axis_tready) and the buffer is not empty.
- Valid and last signals (m_axis_tvalid and m_axis_tlast) are generated based on the availability of output data.

2) Block Diagram



3) Port Description

Generics

Generic name	Type	Value	Description
Data_width		8	Specifies the bit width of the data.
Depth		64	Specifies the depth of the buffer, i.e., the maximum number of packets that can be stored in the buffer.

Port name	Direction	Type	Description
clk	input		Clock signal.
rst	input		Reset signal.
s_axis_tdata	input	[Data_width-1 :0]	Input data from the streaming interface
s_axis_tvalid	input		Valid signal indicating the availability of input data.
s_axis_tlast	input		Indicates the last data word of the input packet.
s_axis_tready	output		Ready signal for the input streaming interface.
packet_config	input	[Data_width+Data_width-1 :0]	Configuration data specifying the packet size (k) and the total length of the packet (len).
m_axis_tdata	output	[Data_width-1 :0]	Output data to the streaming interface.
m_axis_tvalid	output		Valid signal indicating the availability of output data.
m_axis_tlast	output		Indicates the last data word of the output packet.
m_axis_tready	input		Ready signal for the output streaming interface.
full	output		Signal indicating whether the buffer is full.

Port name	Direction	Type	Description
empty	output		Signal indicating whether the buffer is empty.

Signals

Name	Type	Description
mem_1 [Depth-1:0]	reg [Data_width-1:0]	Memory array to store the input data.
mem_2 [Depth-1:0]	reg [Data_width-1:0]	Memory array to store additional data for partial packets.
mem_3 [Depth-1:0]	reg	Memory array to store the last data indicator for each packet.
k	reg [Data_width-1:0]	Configuration parameters extracted from packet_config.
len	reg [Data_width-1:0]	Configuration parameters extracted from packet_config.
wr_ptr	reg [Data_width-1:0]	Write pointer to access the memory arrays.
rd_ptr	reg [Data_width-1:0]	Read pointer to access the memory arrays.
wr_ptr2	reg [Data_width-1:0]	Write pointer for mem_2 to handle partial packets.
i	integer	

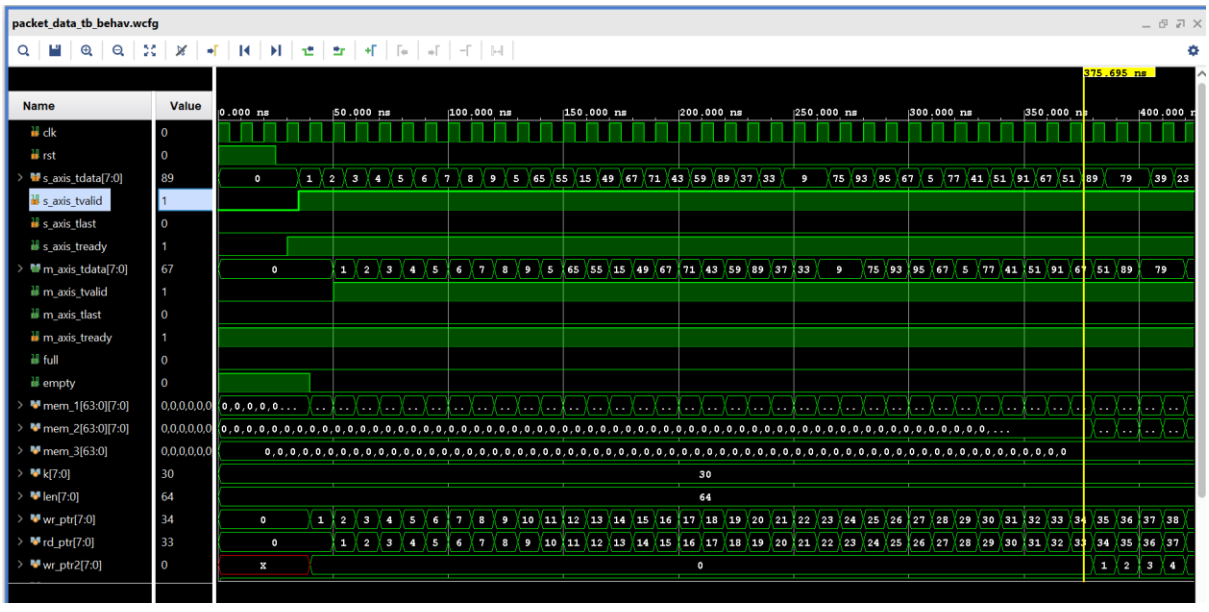
4) Module code

https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_2/packet_data.sv

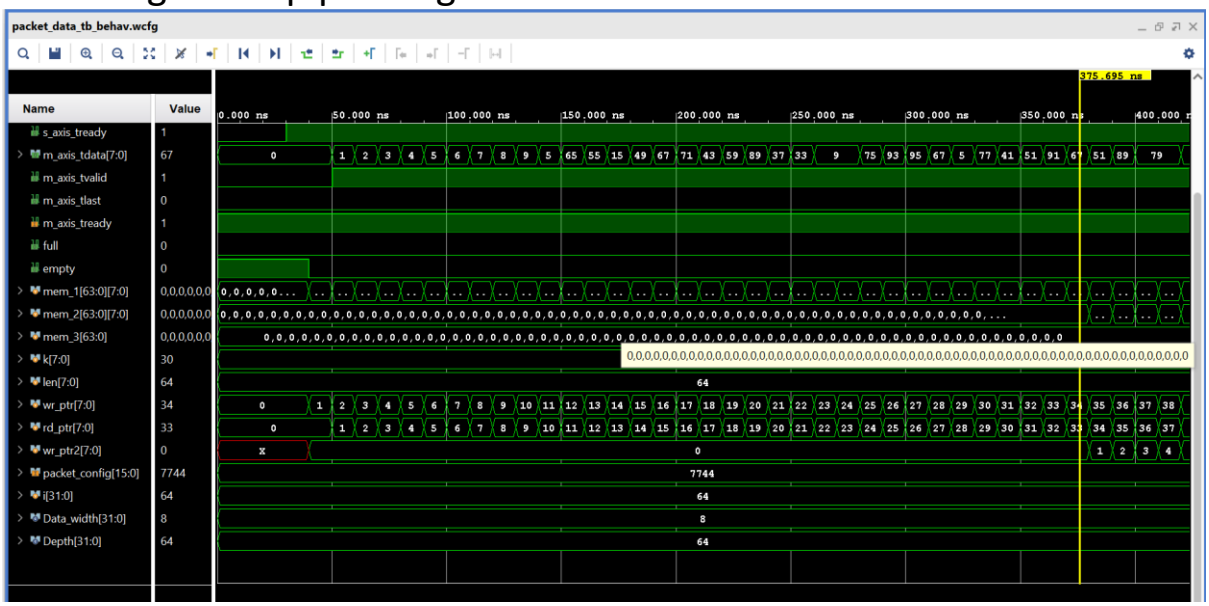
5) Test bench code

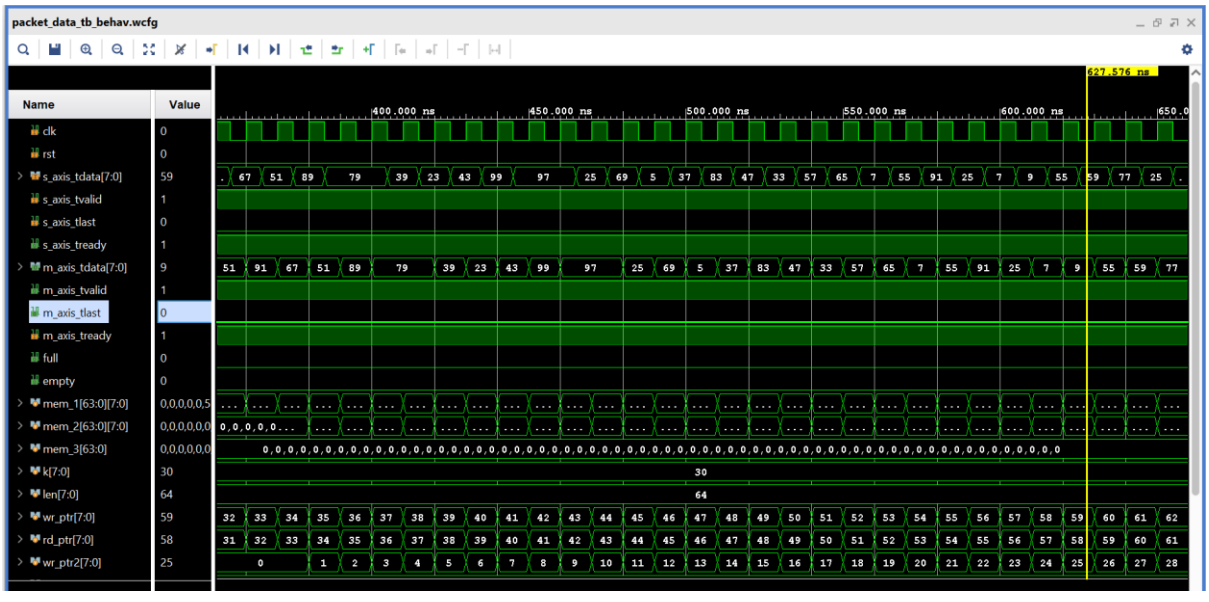
https://github.com/chinnapa5264/RTL_Training/blob/main/Module_3/Assignment_2/packet_data_tb.sv

6) Test cases

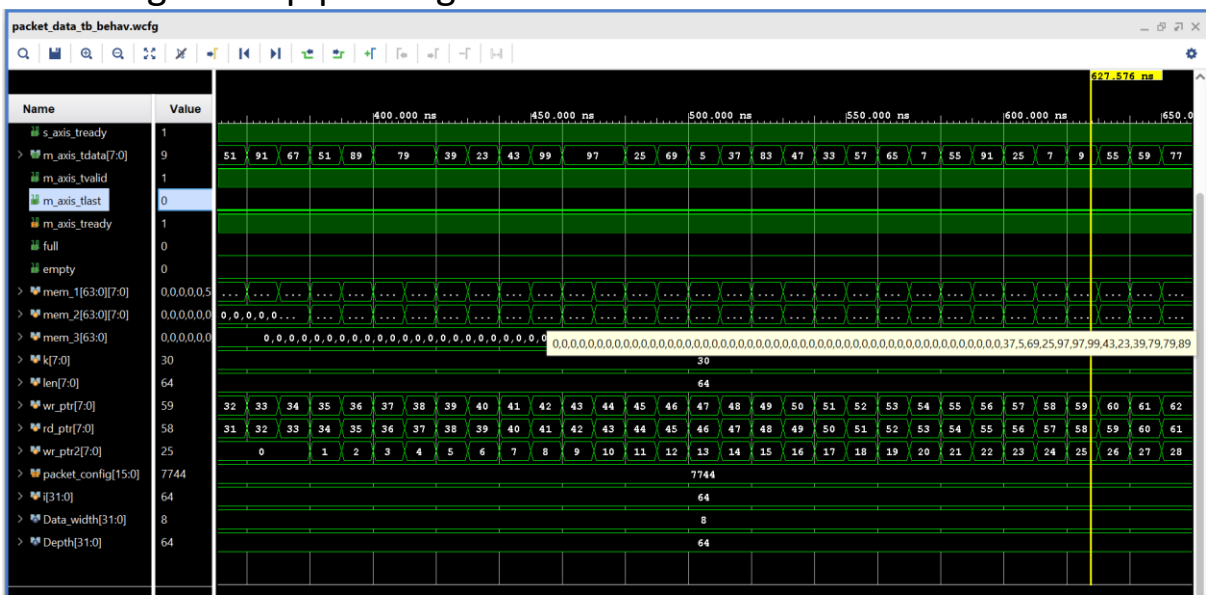


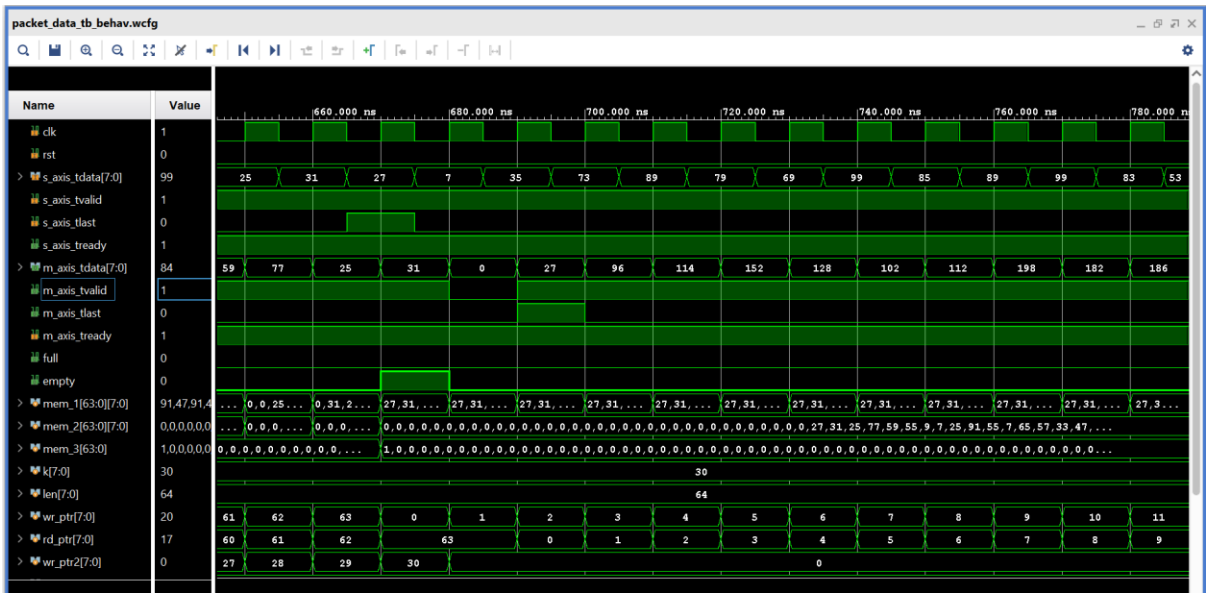
RTL Design with pipelining



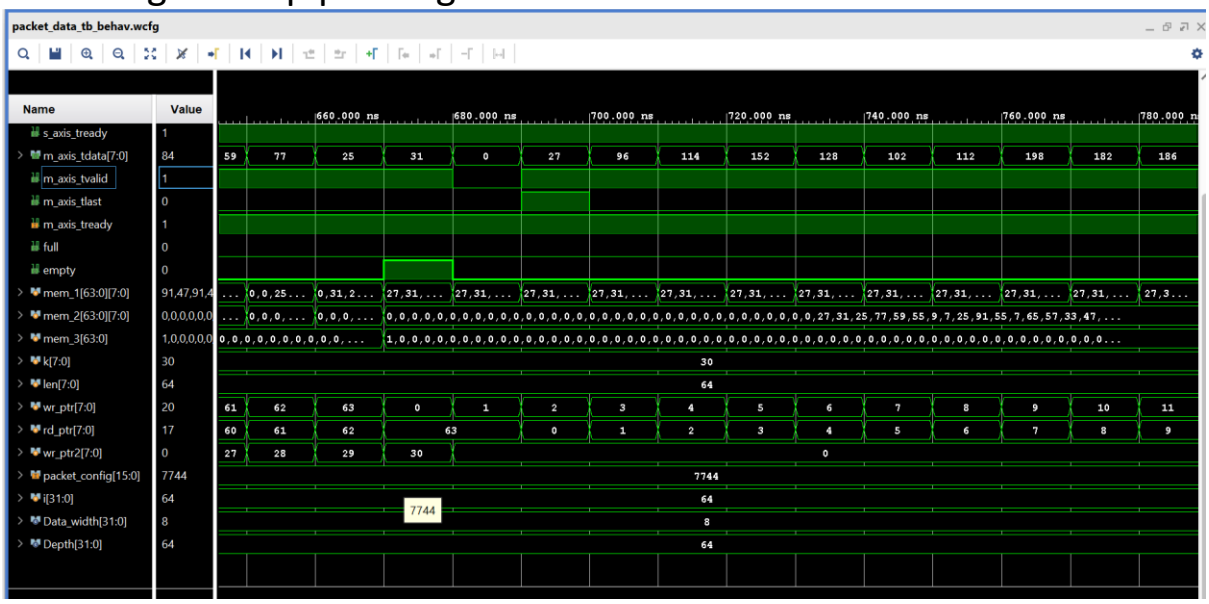


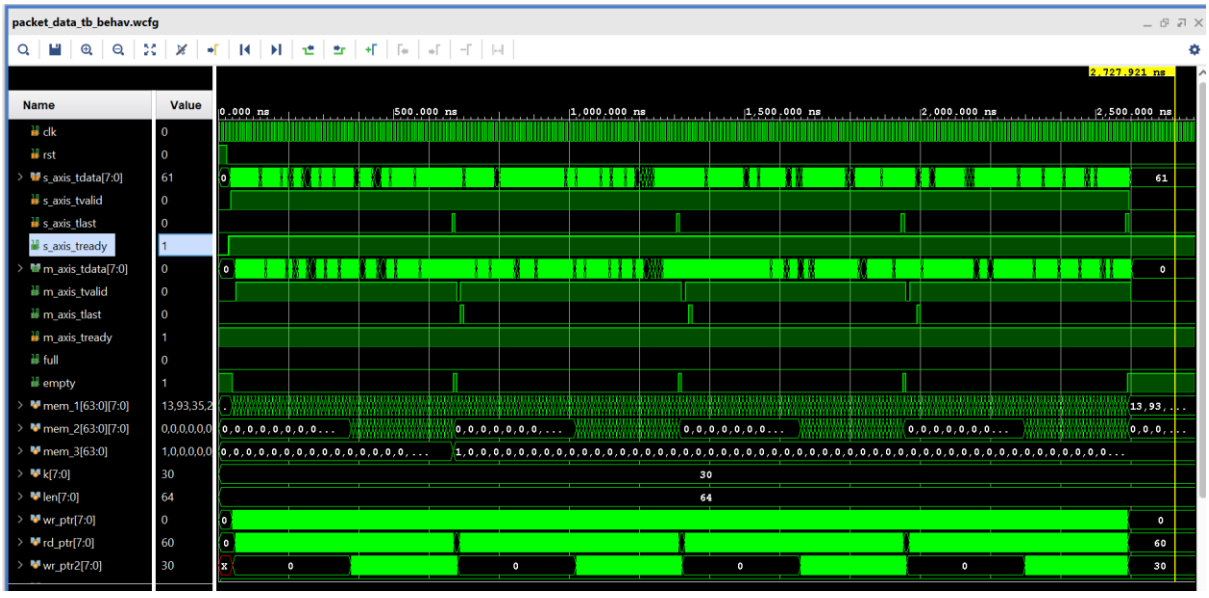
RTL Design with pipelining

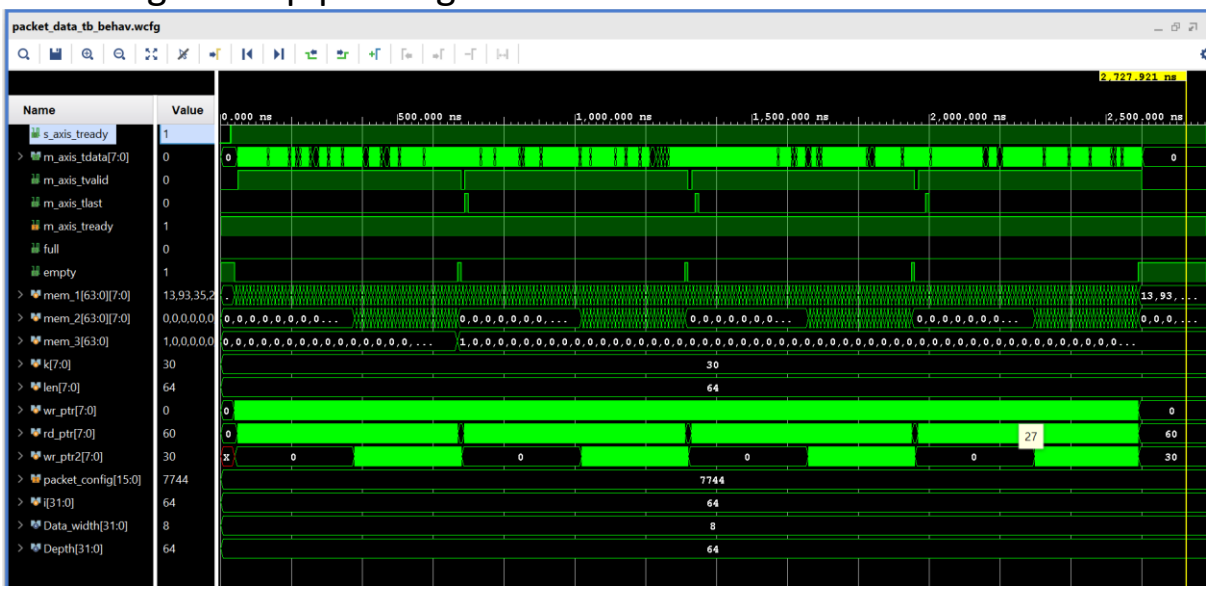




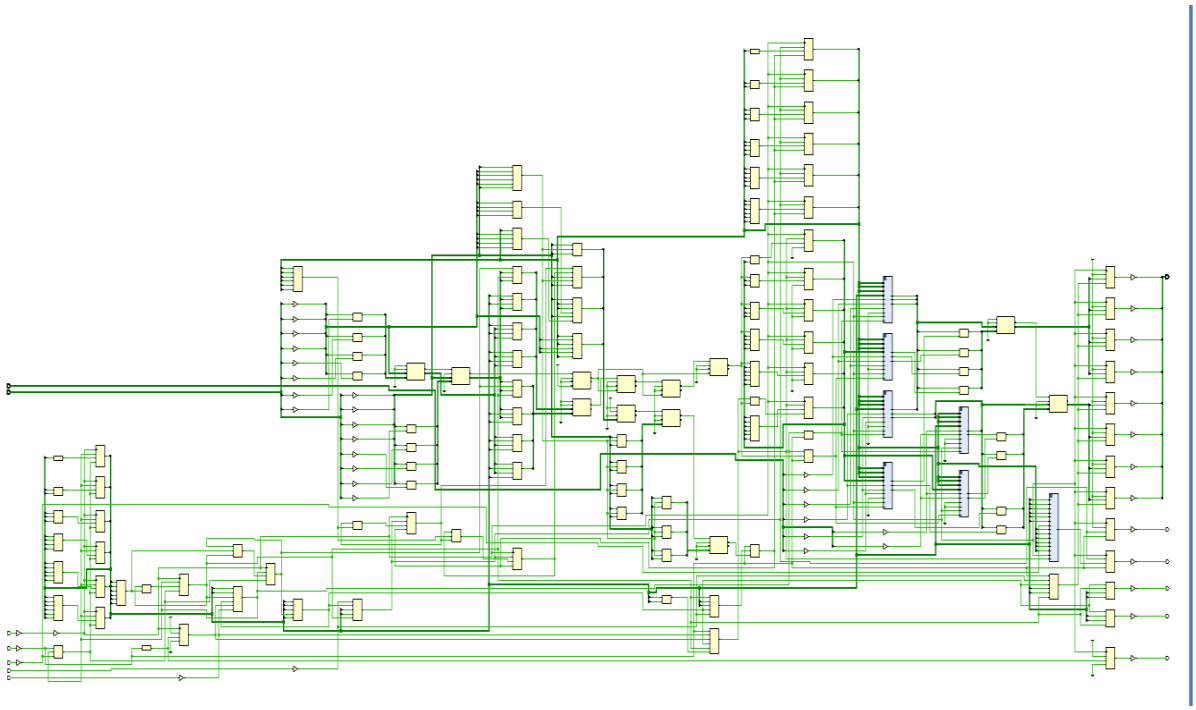
RTL Design with pipelining

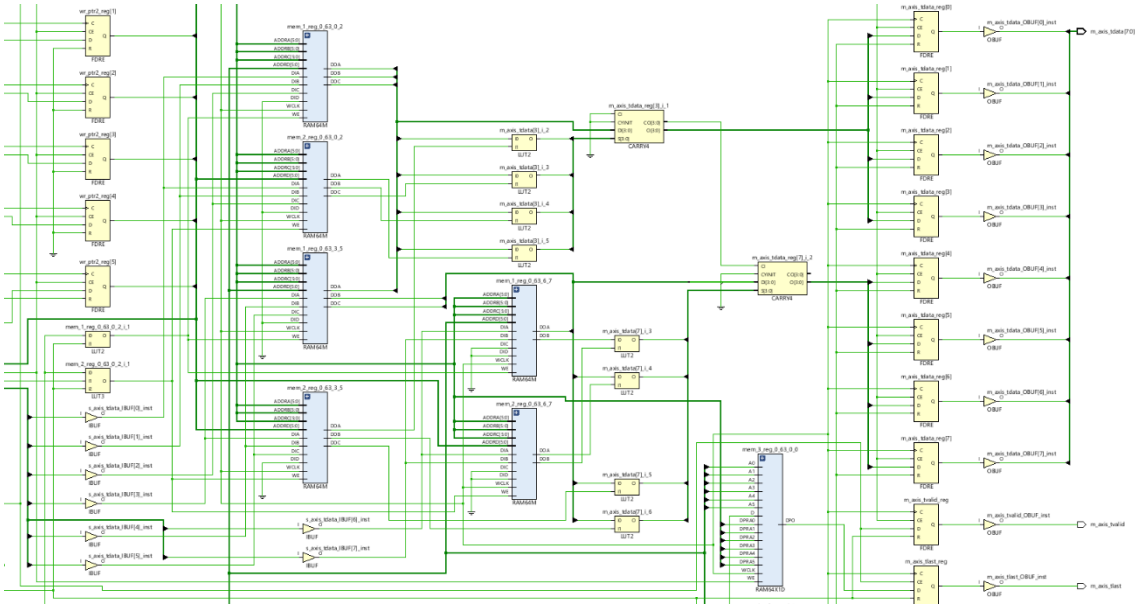






7) Schematic Diagram





8) Utilization

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

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	87	41000	0.21
LUTRAM	26	13400	0.19
FF	34	82000	0.04
IO	42	300	14.00
BUFG	1	32	3.13