RTL Design : (Training)

Module 1: Basics of Verilog and FPGA

Assignment-1

https://hdlbits.01xz.net/wiki/Special:VlgStats/FB9B937037FEF2F0

Shot Note:

Always Block:

For synthesizing hardware, two types of always blocks are relevant:

- Combinational: `always @(*)`
- sequential: `always @(posedge clk)`

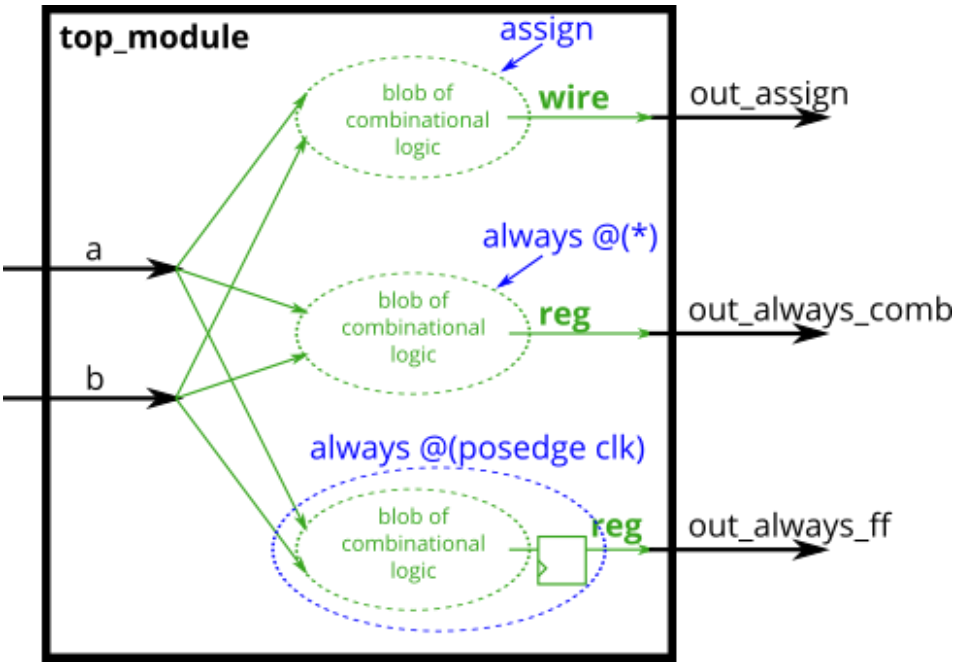
- Combinational always blocks are equivalent to assign statements

For example, the assign and combinational always block describe the same circuit. Both create the same blob of combinational logic. Both will recompute the output whenever any of the inputs (right side) changes value.

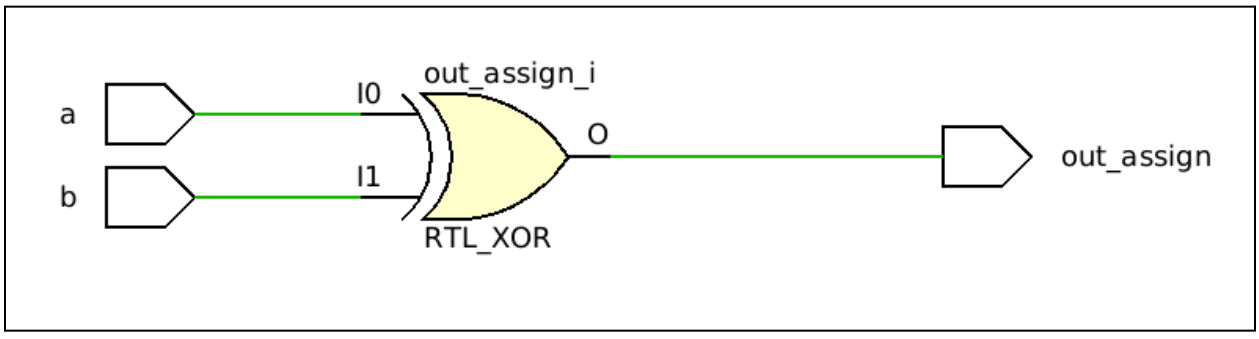
```
assign out1 = a & b | c ^ d;
always @(*) out2 = a & b | c ^ d;
```

- Thus there is always a way to express a combinational circuit both ways. The choice between which to use is mainly an issue of which syntax is more convenient.
- Procedural blocks have a richer set of statements (e.g., if-then, case), cannot contain continuous assignments
- Procedural continuous assignments do exist, but are somewhat different from continuous assignments, and are not synthesizable.

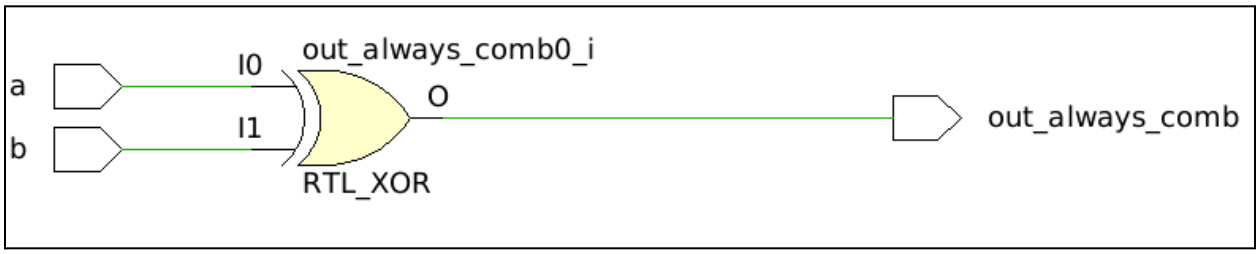
A note on wire vs. reg: The left-hand-side of an assign statement must be a *net* type (e.g., `wire`), while the left-hand-side of a procedural assignment (in an always block) must be a variable type (e.g., `reg`)



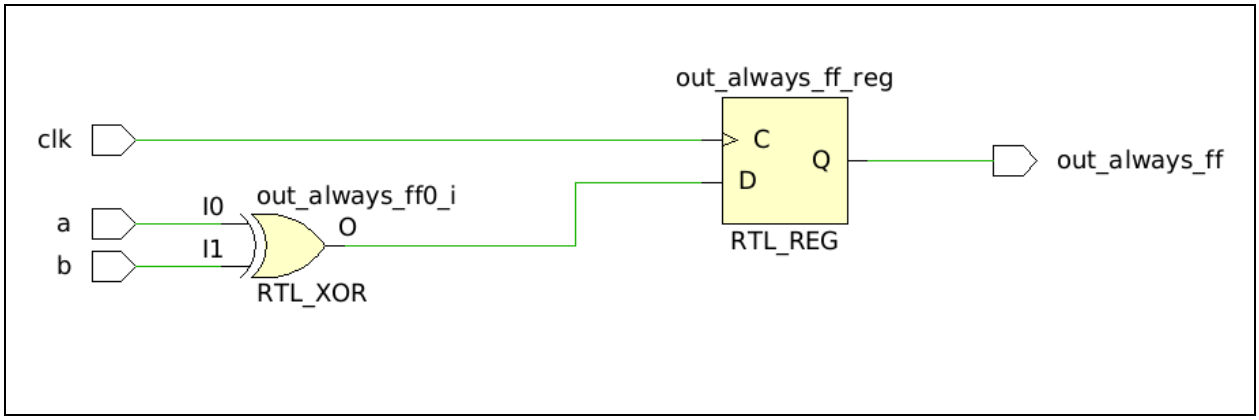

```
`timescale 1ns / 1ps

module test2(
    input a,
    input b,
    output wire out_assign
    );


    assign out_assign =a^b;

endmodule
```



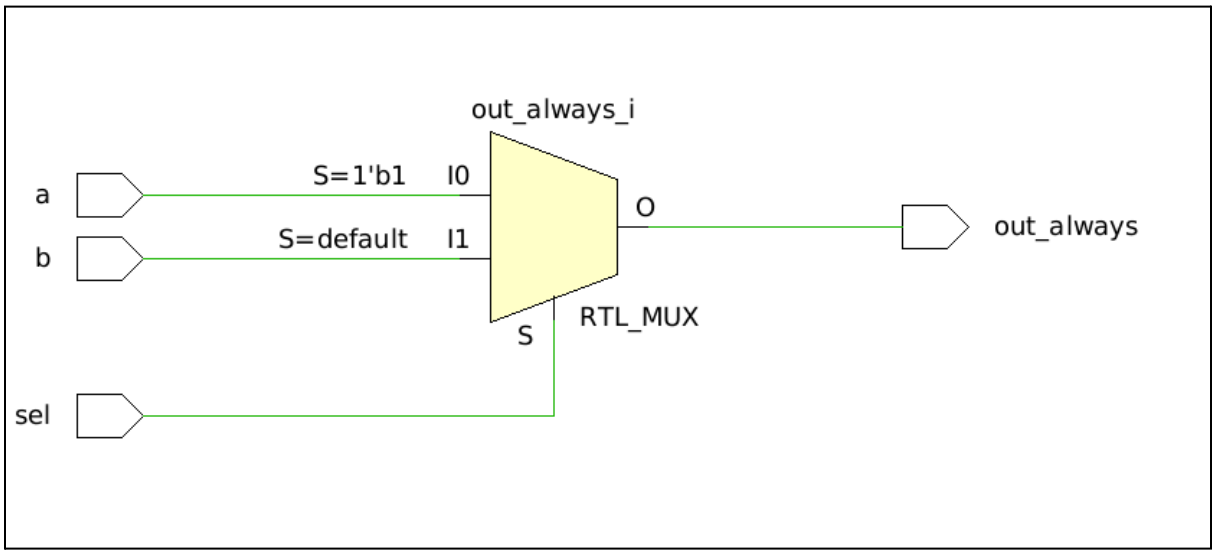
```
`timescale 1ns / 1ps

module test2(

    input a,
    input b,
    output reg out_always_comb
    );

 always @(*)
    begin
    out_always_comb=a^b;
    end

endmodule
```



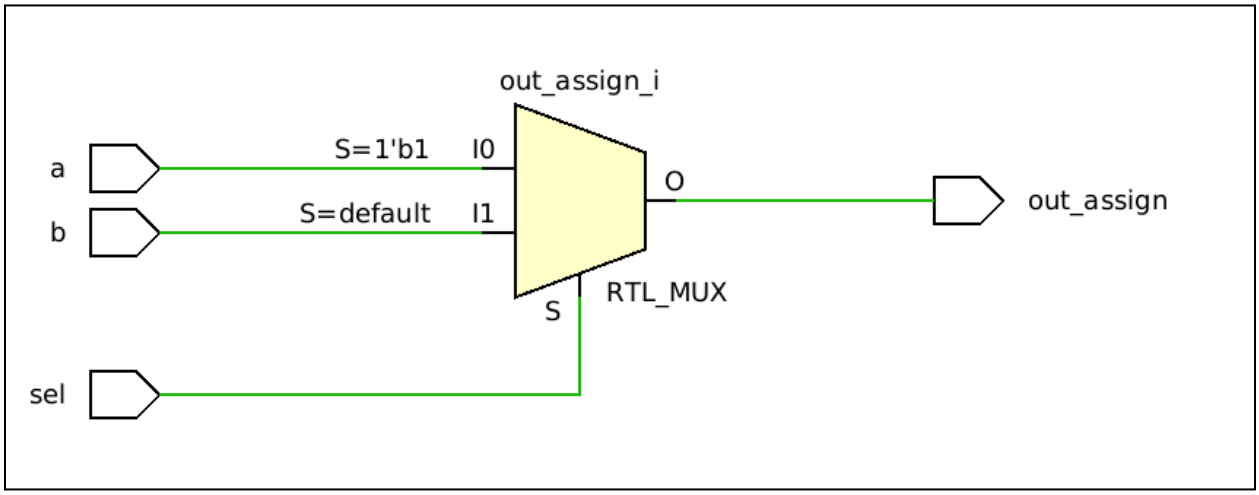

```
`timescale 1ns / 1ps

module test2(
    input clk,
    input a,
    input b,
    output reg out_always_ff
    );

  always@(posedge clk)
    begin
    out_always_ff<=a^b;
    end
endmodule
```




```
`timescale 1ns / 1ps
module test1(
    input a,
    input b,
    input sel,
    output reg out_always
    );

    always @(*) begin
    if (sel) begin
        out_always = a;
    end
    else begin
        out_always = b;
    end
    end
endmodule
```



```
`timescale 1ns / 1ps
module test1(
    input a,
    input b,
    input sel,
    //output reg out_always,
    output wire out_assign
    );
   /*
    always @(*) begin
    if (sel) begin
        out_always = a;
    end
    else begin
        out_always = b;
    end
    end
    */
    assign out_assign = (sel) ? a : b;
endmodule
```



If Statement Latches:

When designing circuits, you *must* think first in terms of circuits:

- I want this logic gate
- I want a *combinational* blob of logic that has these inputs and produces these outputs
- I want a combinational blob of logic followed by a set of flip-flops

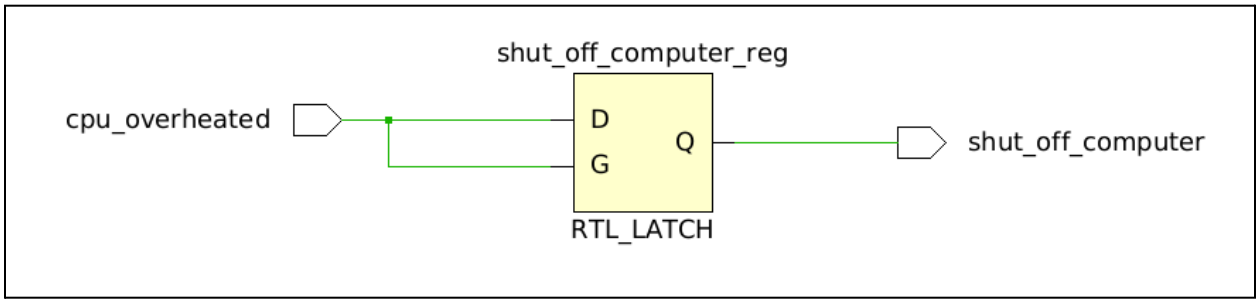
Syntactically-correct code does not necessarily result in a reasonable circuit (combinational logic + flip-flops). The usual reason is: "What happens in the cases other than those you specified?". Verilog's answer is: Keep the outputs unchanged.

This behavior of "keep outputs unchanged" means the current state needs to be *remembered*, and thus produces a *latch*. Combinational logic (e.g., logic gates) cannot remember any state.

Combinational circuits must have a value assigned to all outputs under all conditions. This usually means you always need else clauses or a default value assigned to the outputs.

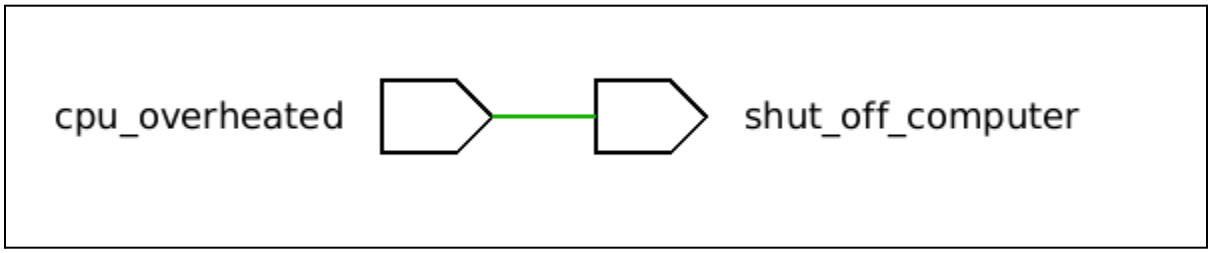
```
`timescale 1ns / 1ps

module test3(
input cpu_overheated,
output reg shut_off_computer
    );

    always @(*)
    begin
    if (cpu_overheated)
       shut_off_computer = 1;
    end

endmodule
```

Non Ideal:




```
`timescale 1ns / 1ps

module test3(
input cpu_overheated,
output reg shut_off_computer
    );

    always @(*)
    begin
    if (cpu_overheated)
       shut_off_computer = 1;
    else
    shut_off_computer = 0;
    end

endmodule
```

Ideal:



Case Statement Latches:

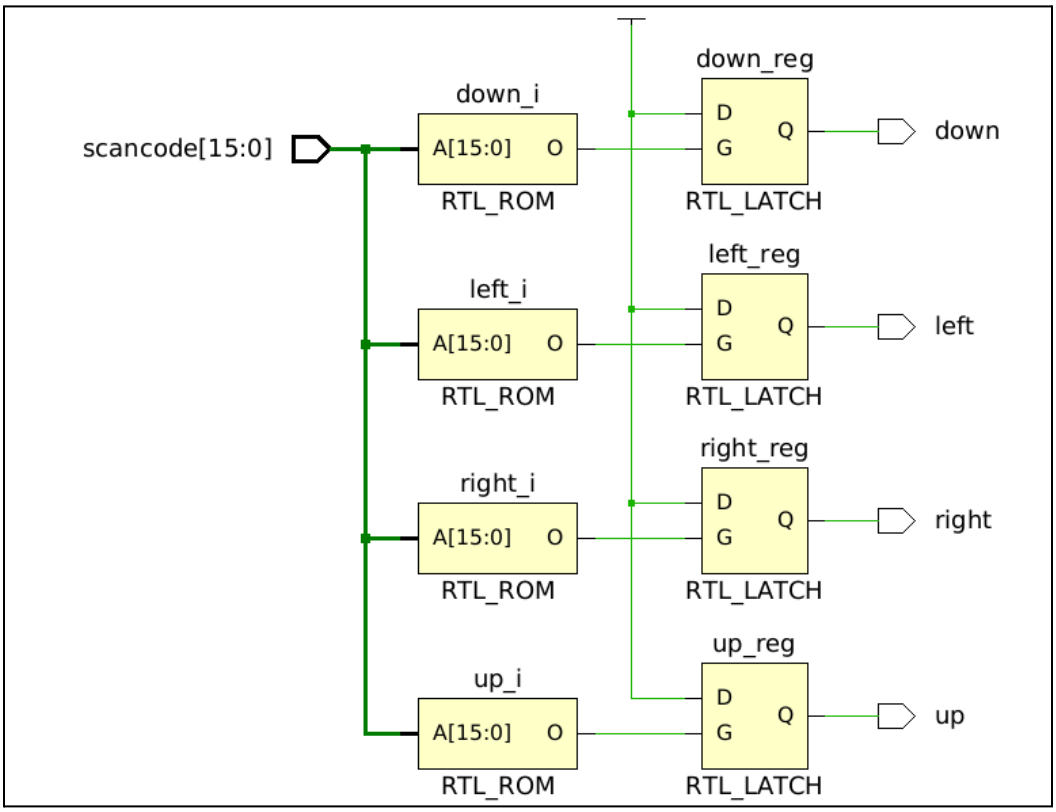
To avoid creating latches, all outputs must be assigned a value in all possible conditions . Simply having a `default` case is not enough. You must assign a value to all four outputs in all four cases and the default case. This can involve a lot of unnecessary typing. One easy way around this is to assign a "default value" to the outputs *before* the case statement:

```
always @(*) begin
    up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b0;
    case (scancode)
        ... // Set to 1 as necessary.
    endcase
end
```

- This style of code ensures the outputs are assigned a value (of 0) in all possible cases unless the case statement overrides the assignment. This also means that a `default:` case item becomes unnecessary.

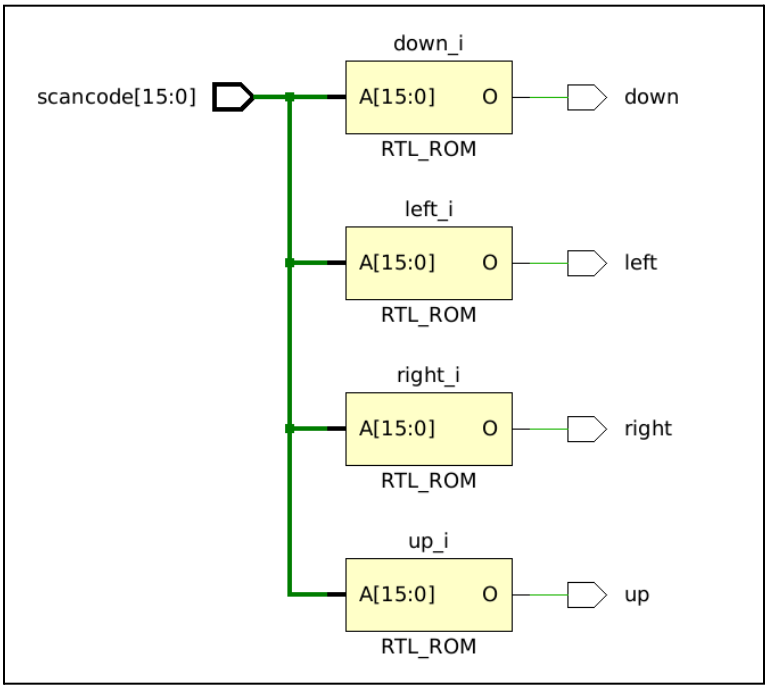
```
`timescale 1ns / 1ps

module test4 (
    input [15:0] scancode,
    output reg left,
    output reg down,
    output reg right,
    output reg up  );
    always @(*) begin
        // up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b0;
        case (scancode)
            16'he06b:left = 1'b1;
            16'he072:down = 1'b1;
            16'he074:right = 1'b1;
            16'he075:up = 1'b1;
            default: ;
        endcase
    end
endmodule
```

Non Ideal:

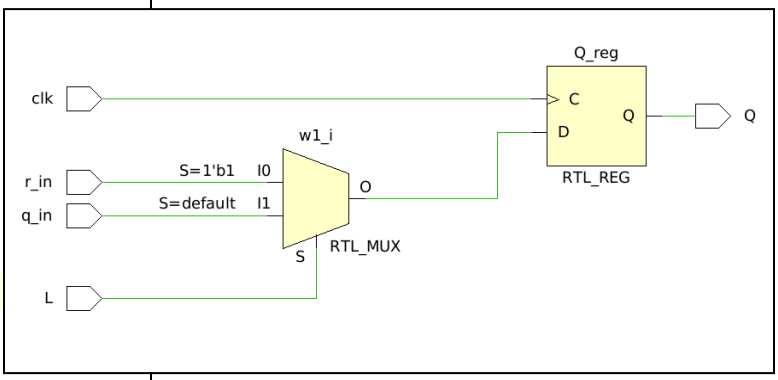



```
`timescale 1ns / 1ps

module test4 (
    input [15:0] scancode,
    output reg left,
    output reg down,
    output reg right,
    output reg up  );
    always @(*) begin
        up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b0;
        case (scancode)
            16'he06b:left = 1'b1;
            16'he072:down = 1'b1;
            16'he074:right = 1'b1;
            16'he075:up = 1'b1;
        endcase
    end
endmodule
```

Ideal:




```
`timescale 1ns / 1ps

module test6 (
    input clk,
    input L,
    input r_in,
    input q_in,
    output reg Q);
    wire w1;
    always@(posedge clk)

            Q<=w1;

    assign w1=(L==1)?r_in:q_in;



endmodule
```



Assignment-2

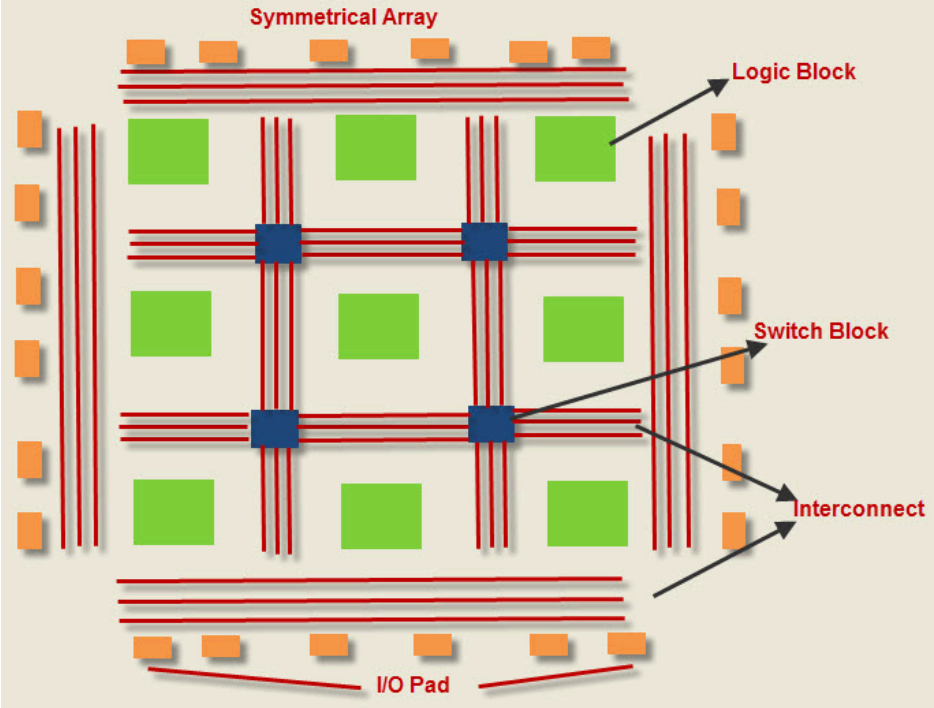
FPGA Architecture

- Programmable Logic (1980's to present)
    - A chip that be reprogrammed after it has been fabricated
      Examples: PALs, EPROM, EEPROM, PLDs, FPGAs
    - Excellent support for mapping from Verilog

- FPGA stands for Field Programmable Gate Array

- semiconductor logic chip which can be programmed to become almost any kind of system or digital circuit

- The configuration of the FPGA architecture is generally specified using a language, i.e., HDL (Hardware Description language)

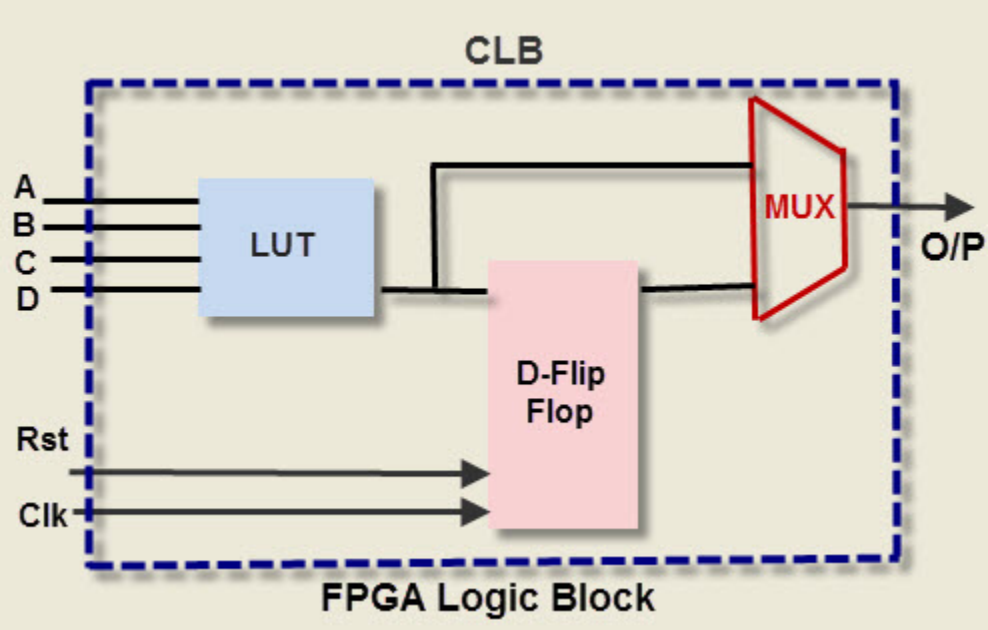


The general FPGA architecture consists of three types of modules.

- They are I/O blocks or Pads,
    - I/O Pads used for the outside world to communicate with different Applications.

- Switch Matrix/ Interconnection Wires and
    - Interconnects provide direction between the logic blocks to
    - implement the user logic.
    - Depending on the logic, the switch matrix provides switching between interconnects.

- Configurable logic blocks (CLB).
    - The basic FPGA architecture has two dimensional arrays of logic blocks
    - CLB (Configurable Logic Block) includes digital logic,inputs,outputs.
    - It implements the user logic.



Logic Block contains MUX (Multiplexer), D flip flop and LUT.

- LUT implements the combinational logical functions;
- the MUX is used for selection logic,
- and D flip flop stores the output of the LUT

Xilinx is the most popular FPGA that contains a Lookup Table (LUT) which is connected with MUX, and a flip flop as discussed above. Present FPGA consists of about hundreds or thousands of configurable logic blocks.

For configuring the FPGA, Vivado and Quartus softwares are used to generate a bitstream file and for development.

Reconfigurable Logic

Logic Blocks

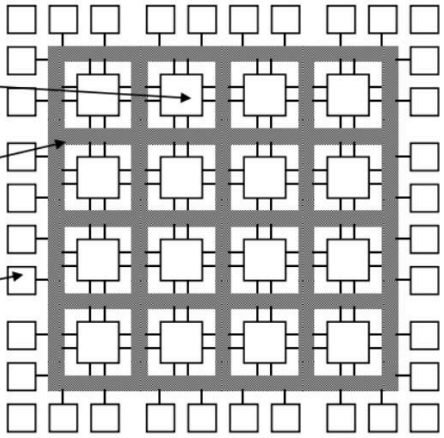
- To implement combinational and sequential logic

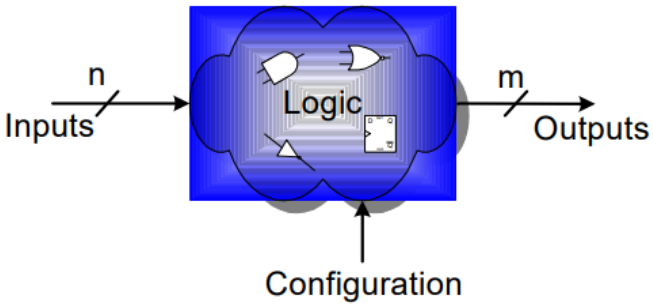
Interconnect

- Wires to connect inputs and outputs to logic blocks

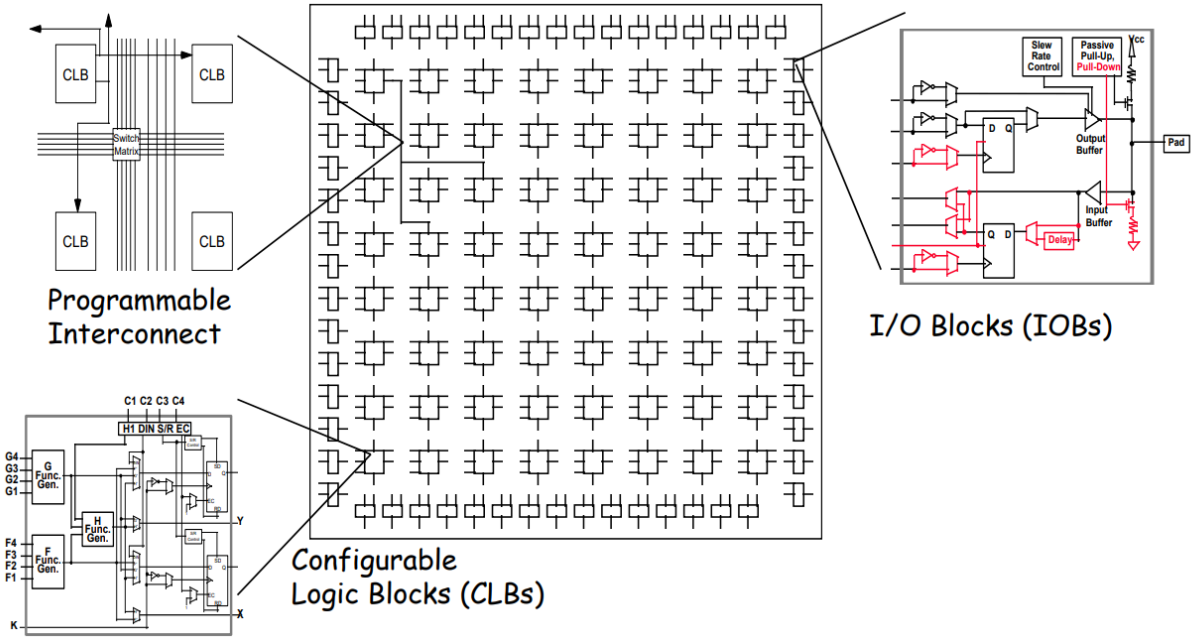
I/O blocks

- Special logic blocks at periphery of device for external connections





Field Programmable Logic - Xilinx



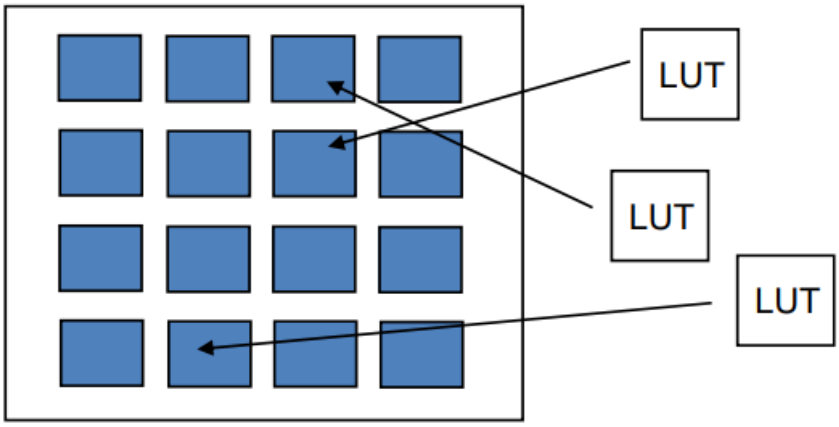
Design Flow - Mapping

- Technology Mapping: Schematic/HDL to Physical Logic units
- Compile functions into basic LUT-based groups (function of target architecture)

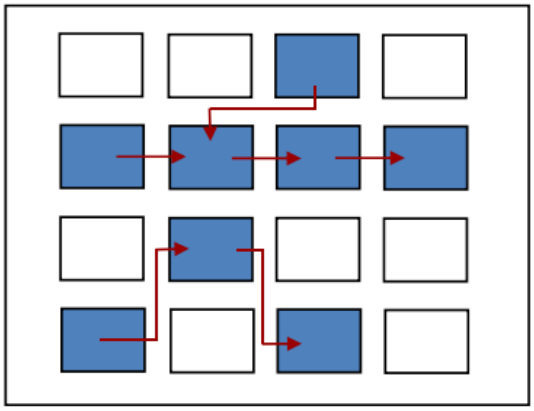

```
always @(posedge clock or negedge reset)
  begin
    if (! reset)
       q <= 0;
    else
       q <= (a&b&c)||(b&d);
  end
```

Design Flow – Placement & Route

- Placement – assign logic location on a particular device



- Routing – iterative process to connect CLB inputs/outputs and IOBs. Optimizes critical path delay – can take hours or days for large, dense designs



Sample Test Bench :

```
module sample_tf;
   // Inputs
   reg bit_in;
   reg [3:0] bus_in;

   // Outputs
   wire out_bit;
   wire [7:0] out_bus;

   // Instantiate the Unit Under Test (UUT
   sample uut (
      .bit_in(bit_in),
      .bus_in(bus_in),
      .out_bit(out_bit),
      .out_bus(out_bus)
   );

   initial begin
      // Initialize Inputs
      bit_in = 0;
      bus_in = 0;

      // Wait 100 ns for global reset to finish
      #100;

      // Add stimulus here

   end

endmodule
```

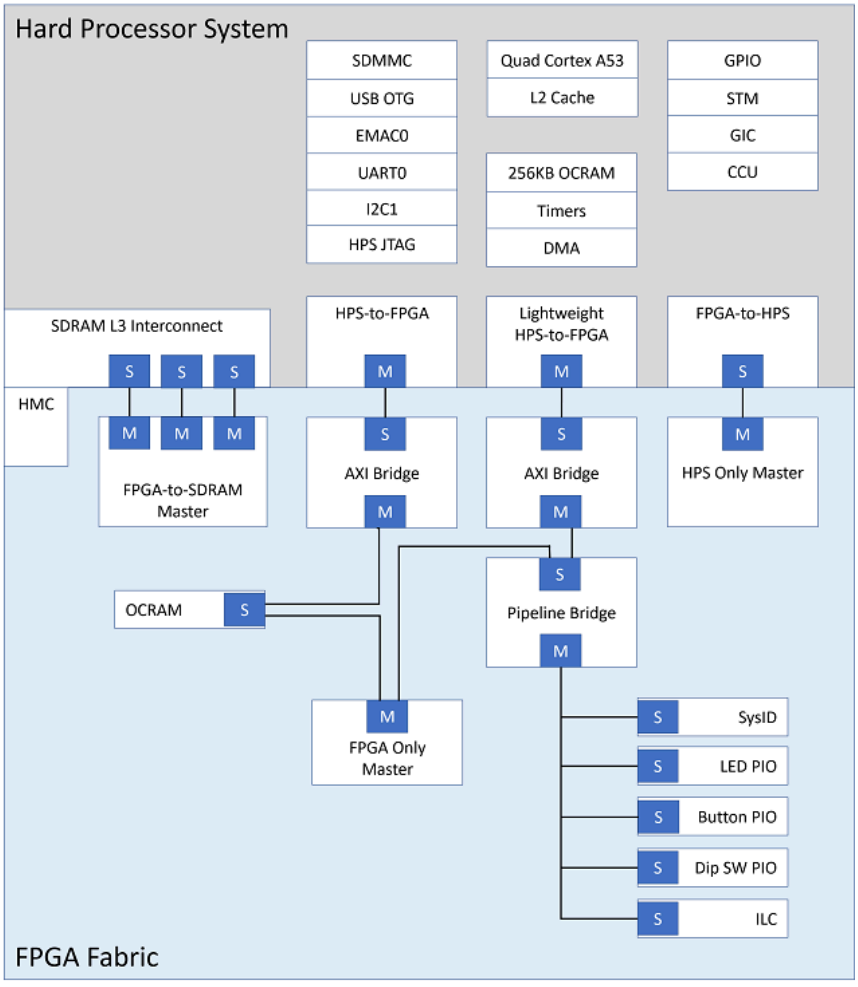
```
module sample(
   input bit_in,
   input [3:0] bus_in,

   output out_bit,
   output [7:0] out_bus
   );
  . . . Verilog . . .

endmodule
```

Summary

- FPGA provide a flexible platform for implementing digital computing
- A rich set of macros and I/Os supported (multipliers, block RAMS, ROMS, high-speed I/O)
- A wide range of applications from prototyping (to validate a design before ASIC mapping) to high-performance spatial computing
- Interconnects are a major bottleneck (physical design and locality are important considerations)

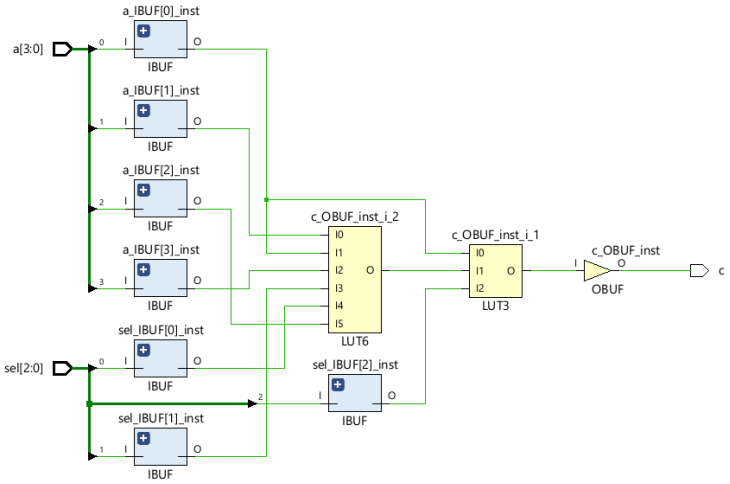
Intel SoC FPGA: (PS & PL)

Reference Design:



FPGA Resources:

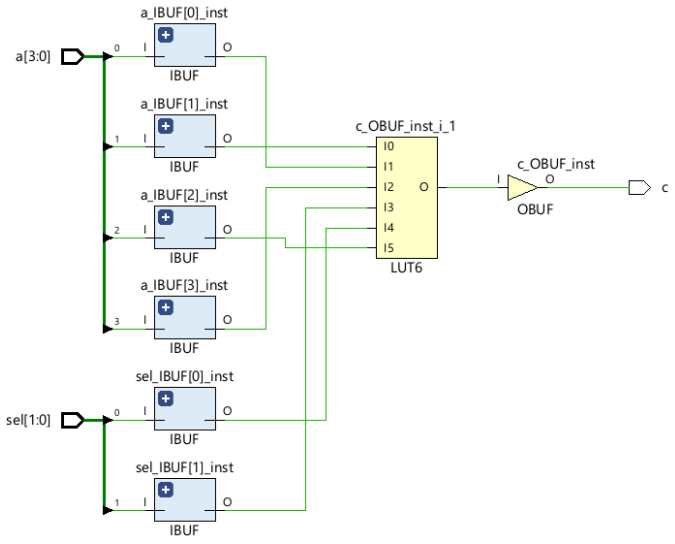
```
`timescale 1ns / 1ps
module test2(
input [3:0] a,
input [2:0] sel,
output reg c
);

always@(*)
case(sel)
3'b000:c=a[0];
3'b001:c=a[1];
3'b010:c=a[2];
3'b011:c=a[3];
default c=a[0];
endcase

endmodule
```



(a=4,sel=3)
7-inputs ⇒ 1 LUT6, 1 LUT3

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												2	0	0	0	0

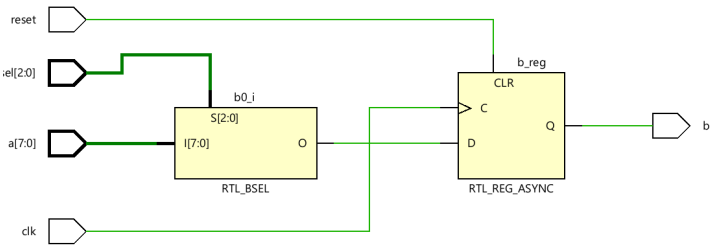
```
`timescale 1ns / 1ps
module test2(
input [3:0] a,
input [1:0] sel,
output reg c
);

always@(*)
case(sel)
2'b00:c=a[0];
2'b01:c=a[1];
2'b10:c=a[2];
2'b11:c=a[3];
default c=a[0];
endcase

endmodule
```



(a=4,sel=2)
6-inputs ⇒ 1 LUT6

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM
✓ synth_1	constrs_1	synth_design Complete!												1	0	0	0

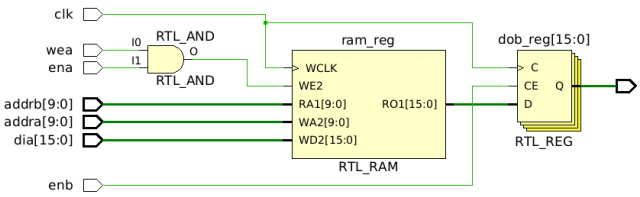
```
`timescale 1ns / 1ps
`timescale 1ns / 1ps
module mux_8_1(
input clk,
input reset,
input [7:0] a,
input[2:0] sel,
output reg b
    );

  always@(posedge clk or posedge reset) begin
  if(reset)
  b<=0;
  else
  b<=a[sel];
  end

endmodule
```



Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												2	1	0	0	0

(a=8,sel=3)
11-inputs ⇒ 2x LUT6, 1 Flip Flop (procedural sequential block)

```
module bram (clk,ena,enb,wea,addra,addrb,dia,dob);

input clk,ena,enb,wea;
input [9:0] addra,addrb;
input [15:0] dia;
output [15:0] dob;
reg [15:0] ram [1023:0];
reg [15:0] doa,dob;

always @(posedge clk) begin
if (ena) begin
if (wea)
ram[addra] <= dia;
end
end

always @(posedge clk) begin
if (enb)
dob <= ram[addrb];
end

endmodule
```



Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												0	0	0.5	0	0

0.5 - BRAM (Store large data sets on an FPGA target more efficiently than RAM built from look-up tables)

```
module simple_bram (
    input wire clk,
    input wire [9:0] addr,
    input wire [7:0] data_in,
    input wire we,
    output reg [7:0] data_out
);

reg [7:0] memory [0:64];

always @(posedge clk) begin
        if (we) begin
            memory[addr] <= data_in;
        end
        data_out <= memory[addr];
    end

endmodule
```

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												26	8	0	0	0

Uses LUTS & FF's (if less Array size to be stored)

```
module squarediffmult # (parameter SIZEIN = 16)
 (
  input clk,
  input ce,
  input rst,
  input signed [SIZEIN-1:0] a, b,
  output signed [2*SIZEIN+1:0] square_out
 );

 reg signed [SIZEIN-1:0] a_reg, b_reg;
 reg signed [SIZEIN:0] diff_reg;
 reg signed [2*SIZEIN+1:0] m_reg, p_reg;

always @(posedge clk)
 if (rst)
  begin
     a_reg <= 0;
     b_reg <= 0;
     diff_reg <= 0;
     m_reg <= 0;
     p_reg <= 0;
  end
 else
 if (ce)
   begin
     a_reg <= a;
     b_reg <= b;
     diff_reg <= a_reg - b_reg;
     m_reg <= diff_reg * diff_reg;
     p_reg <= m_reg;
   end
 assign square_out = p_reg;
```

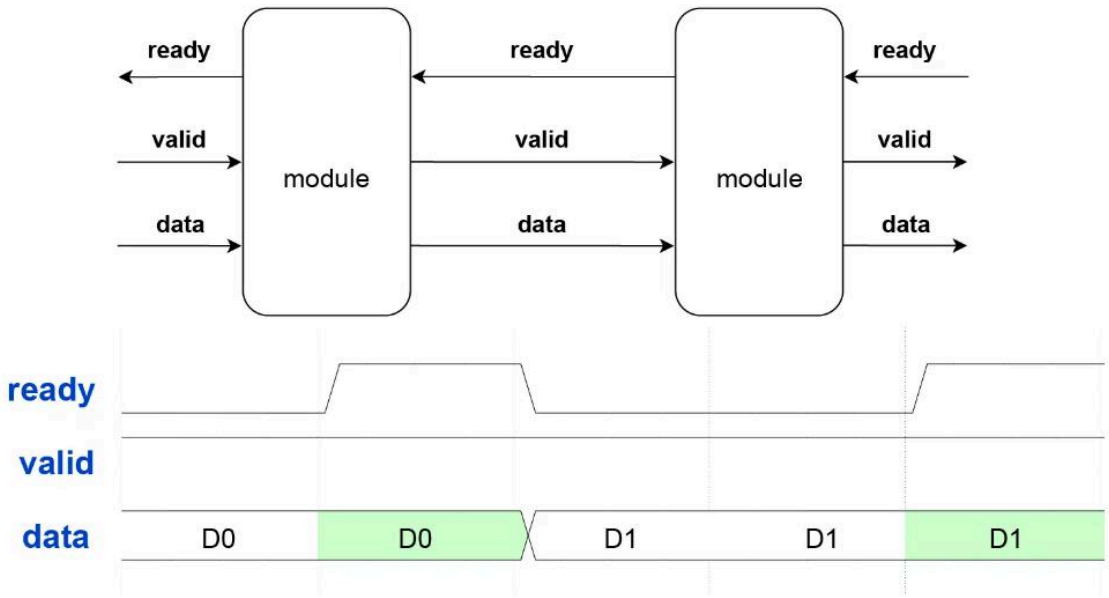
Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												0	0	0	0	1

Uses 1-DSP (For complex Arithmetic operations, advantage of the pipelining capabilities of DSP blocks)

Module 2: RTL Designs with AXIS Interfaces

Short Note:

How does the AXI Handshake Works..?

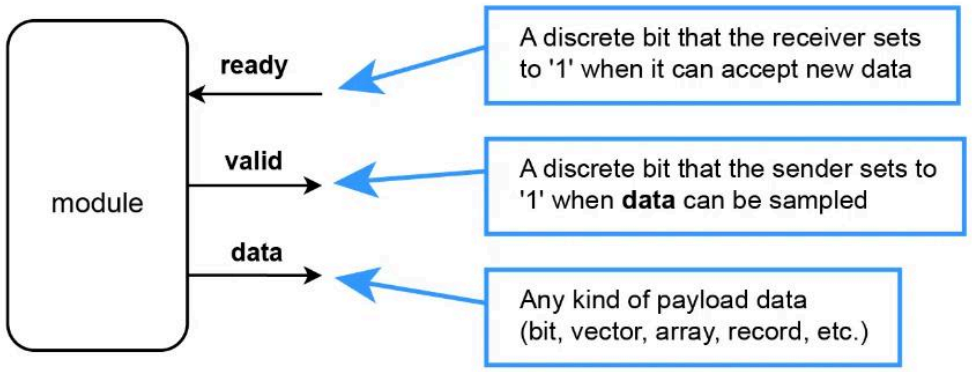


- The ready/valid hardware data transfer protocol is simple, providing flow control with only two control signals. The rules are straightforward: data transfer only happens when both ready and valid are '1' during the same clock cycle.

- The AMBA AXI protocol uses the ready/valid handshake signals for flow control on all its hardware data buses. Therefore, many developers associate the ready/valid naming scheme with AXI, but it's a fundamental flow control mechanism that you should know about even if you're not using AXI.

- Furthermore, both parties must operate synchronously and read the control signals on the same clock edge. Because of that, ready/valid isn't appropriate for clock domain crossing.

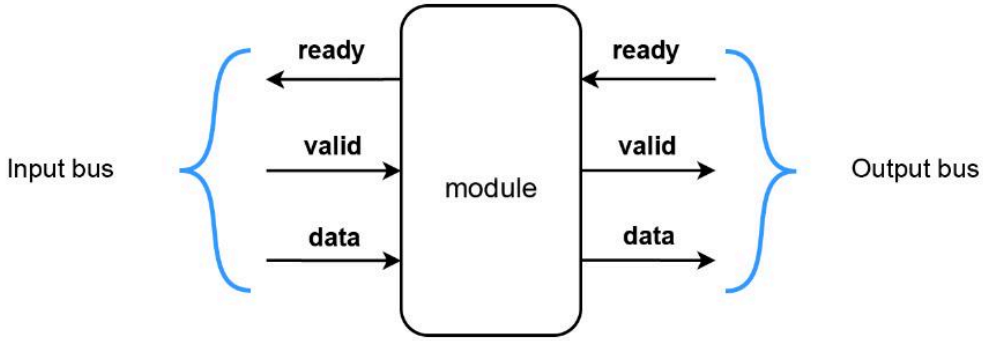
- The diagram below shows a module with one output bus that uses ready/valid handshaking. While ready is the receiver's key to limiting the data flow into it, the sender controls the valid and data signals. Both parties can throttle the data rate, and transfers only happen when both agree.



- Stream processing modules typically have input and output interfaces. That's because they sit on the data path and do various transformations on the data before passing it on to downstream modules.

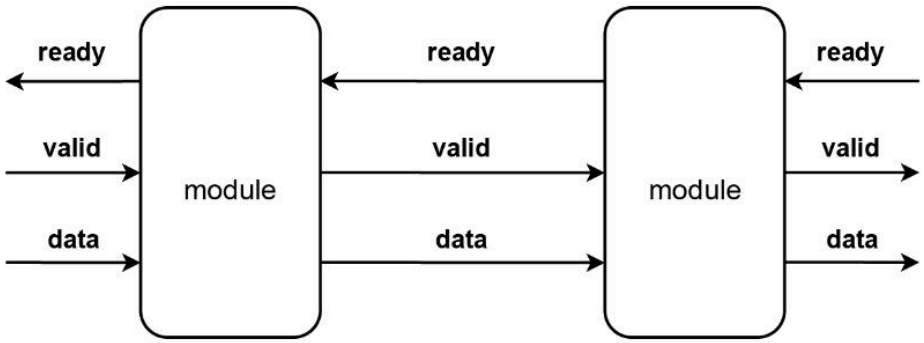
- The diagram below shows the outline of such a module. Notice that the input and output buses have identical names but with the data directions swapped.

(Also called transmitter/receiver, data source/sink, or master/slave interface)



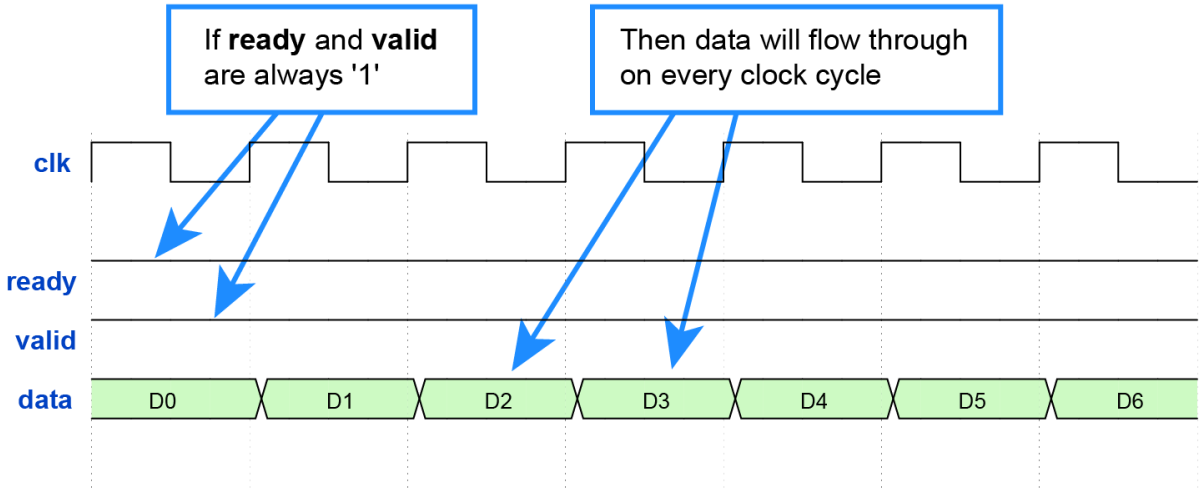
- ready/valid signals go to the signals with matching names when connecting sender and receiver modules.

- You can see the benefit in the diagram below, showing two interconnected modules with identical interfaces.



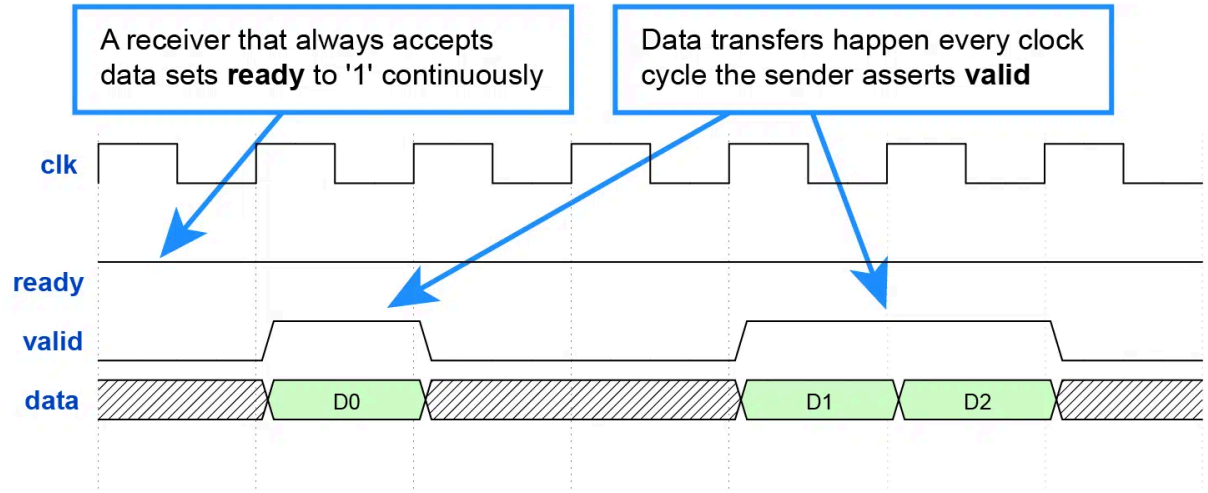
Sending and Receiving at full speed

The simplest example you can think of is when ready and valid are '1' continuously. Then, data flows through the interface unhindered, with one transaction on every clock cycle. It's as if there was no flow control.



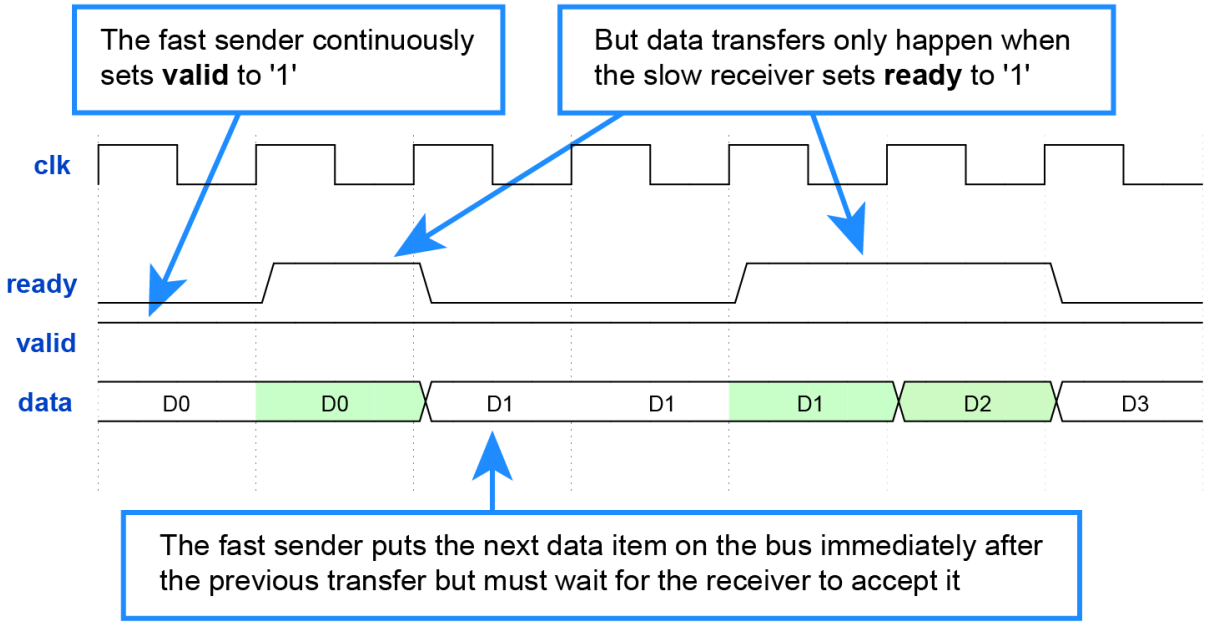
Slow writer and fast reader

In this example, the ready signal is always '1' while the sender asserts valid occasionally.



Fast writer and slow reader

This waveform shows a situation where the reader module is throttling the data rate. We say that the downstream module exerts back pressure when it pauses the data stream like that.

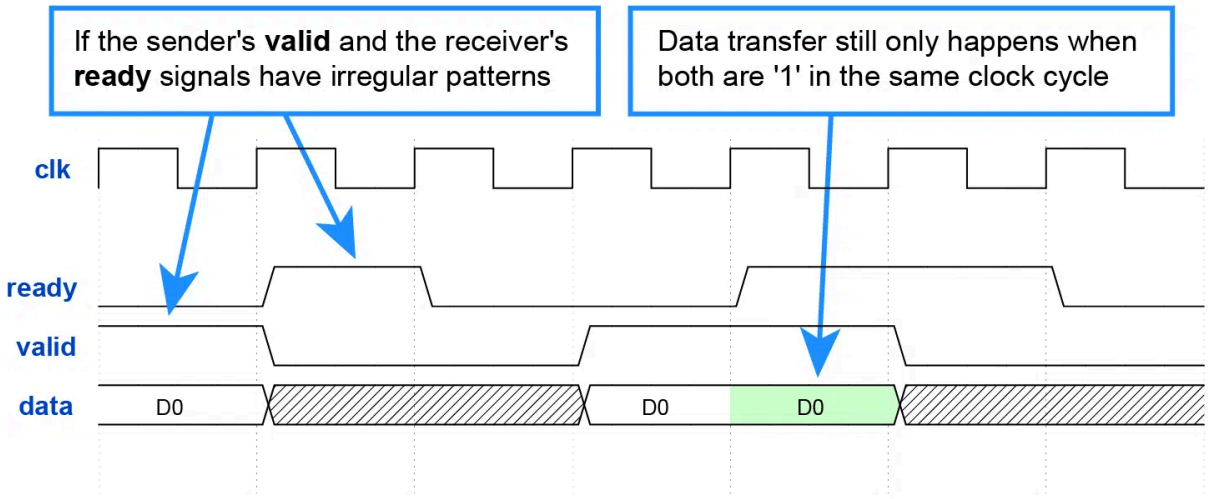


Irregular read/write pattern

The writer has no obligation to keep the data on the bus until it's read. It can put it on the bus on one clock cycle and remove it on the next. It's legal* in the ready/valid handshake.

The AXI Stream protocol doesn't permit this behavior.

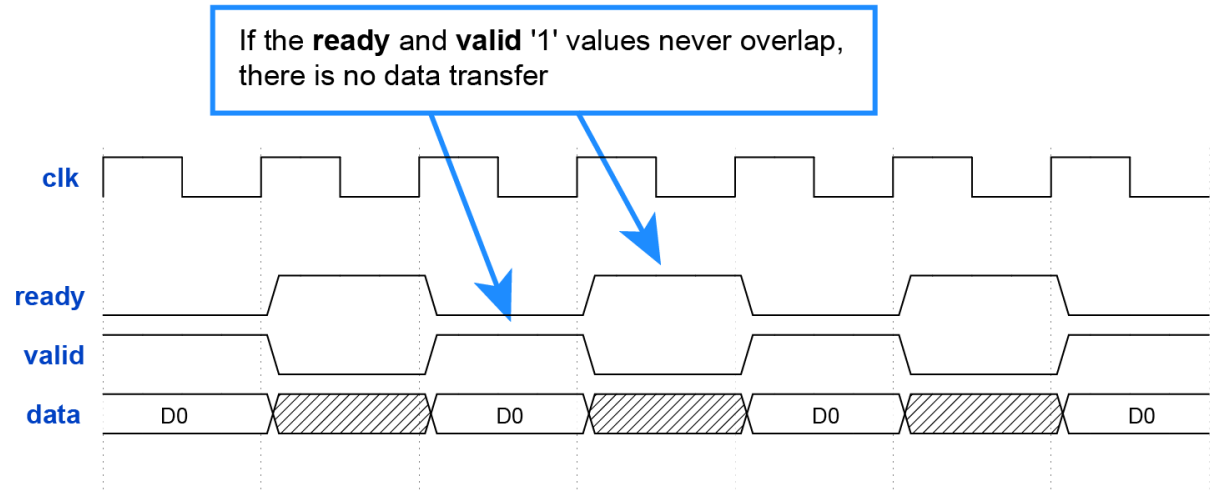
A master is not permitted to wait until TREADY is asserted before asserting TVALID. Once TVALID is asserted it must remain asserted until the handshake occurs.



Non-overlapping read/write pattern

Finally, let's look at an example where ready and valid switches continuously, but no data passes through.

Hopefully, you won't see this behavior in an RTL design, but it's useful to understand why it's like that.



Assignment-1

AXI-Stream Register:

Introduction:

AXI-Stream Register provides the infrastructure to insert a pipeline stage between AXI4-Stream master and slave

Functionality:

AXI-Stream Register acts as a buffer which could able to store one sample of data, abiding the Hand Shaking protocols rules of AXI-Stream.

Whenever slave is ready with the valid samples and also master is ready to accept the samples this AXI-Stream register acts as a single stage pipeline mediator passing the data via it

Implementation:

HANDSHAKE MECHANISM:

Firstly due to the initial reset " valid_out " internal signal was assigned a initial value of '0'
Due to this, ready becomes high and makes the "s_axis_tready" high making the module to accept the data at all the cases of valid_out is zero.

```
always @ (posedge clk or posedge reset)
  begin
   if (reset == 1)
    begin
      valid_out <= 0;
          .
          .
assign ready = (valid_out == 0) | ((m_axis_tready == 1) & (valid_out == 1));
assign s_axis_tready= (ready == 1) & !(reset);
```

On the next clock cycle “valid_out” was assigned a value of enable signal value because of “ready” high (see below else)

Internal enable signal is dependent on ready and s_axis_tvalid. If enable was “0” because of the s_axis_tvalid low then valid_out becomes “0”, making the module to accept the signal continuously until valid_out becomes “HIGH”

```
.
.
else if (ready == 1)
      begin
        valid_out <= enable;
      end
   end

 assign enable = ((ready == 1) & (s_axis_tvalid == 1));
```

Once if valid_out becomes high because of s_axis_tvalid then it checks for m_axis_tready

If the master was not in a position to accept the data the ready becomes low it in turn makes s_axis_tready low making the module to not accept the new data because we only have room for one sample unlike FIFO, if not deasserted our module is flood with the data and we may lose valid samples

Whenever the master is ready to accept the data ready becomes high and it in turn makes the s_axis_tready high making the module to accept the new data by passing the internal stored data to the master.

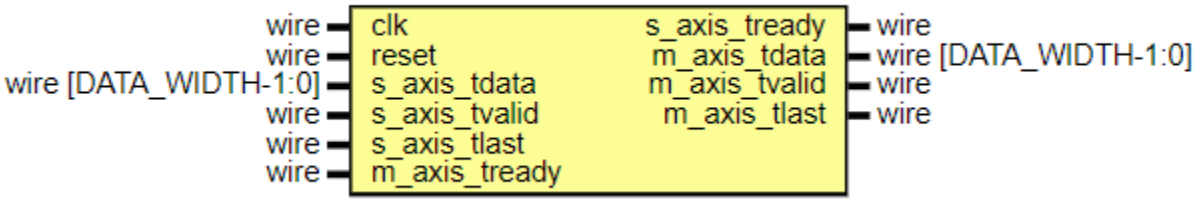
As Long as the valid_out is HIGH s_axis_tdata will be store in the internal register

```
       .
       .
       .
    else if (valid_out == 1)
       begin
         reg_data <=  s_axis_tdata;
       end
     end
```

Internal reg_data will be continuously assigned to the out port m_axis_tdata whenever it was available

```
assign m_axis_tdata  =  reg_data;
```

Module Block design

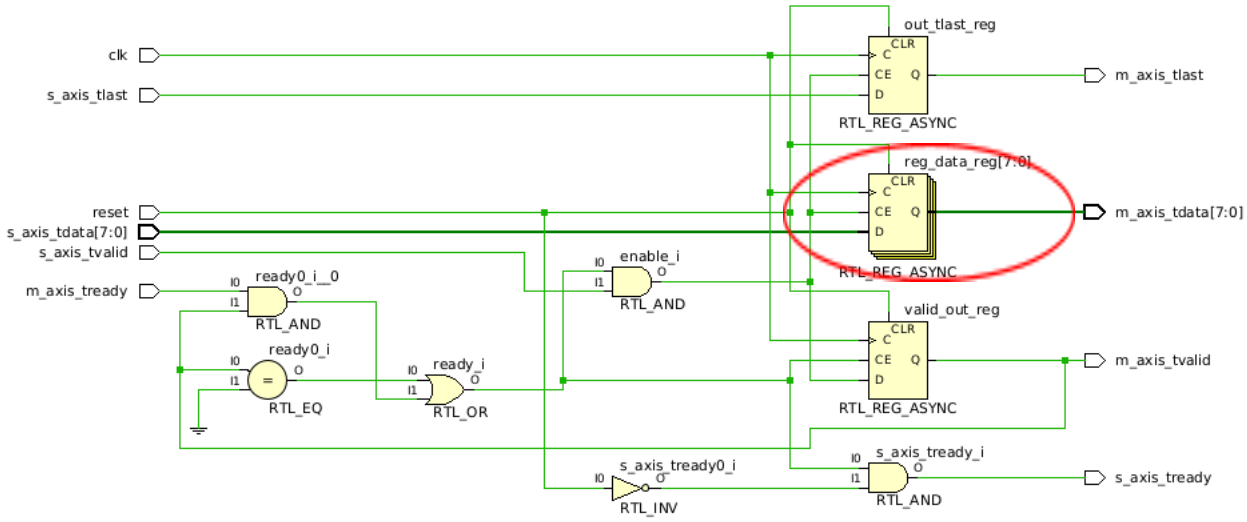


I/O Details :

Port name	Direction	Type	Description
clk	input	wire	For providing the clock for the sequential logic
reset	input	wire	For asynchronous reset
s_axis_tdata	input	wire [DATA_WIDTH-1:0]	For passing the upstream data inside the module
s_axis_tvalid	input	wire	To prompt the module as valid data
s_axis_tready	output	wire	For indicating the module is not busy and ready to accept the signal
s_axis_tlast	input	wire	For indicating the last sample of the packet

m_axis_tdata	output	wire [DATA_WIDTH-1:0]	Master data port to send the data out for the other module
m_axis_tvalid	output	wire	Valid signal to prompt the data as either as valid or invalid
m_axis_tready	input	wire	To get the ready signal from master side indication of weather it is busy or free to accept
m_axis_tlast	output	wire	Last sample of the packet

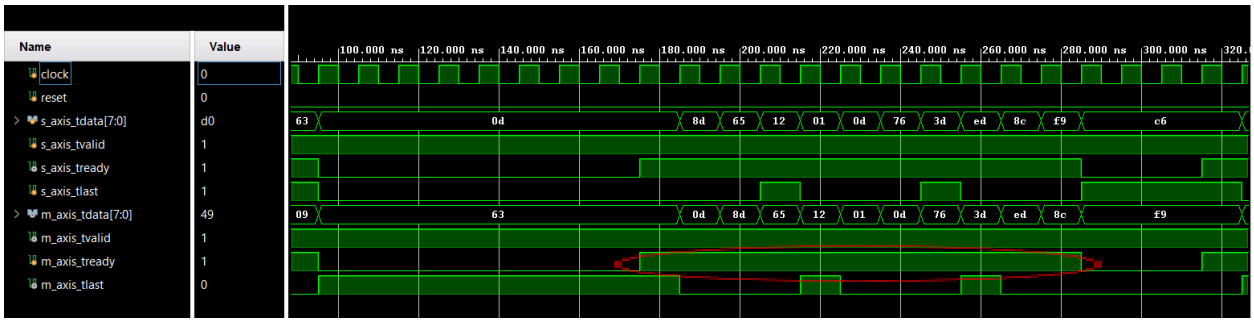
Schematic:



Verification:

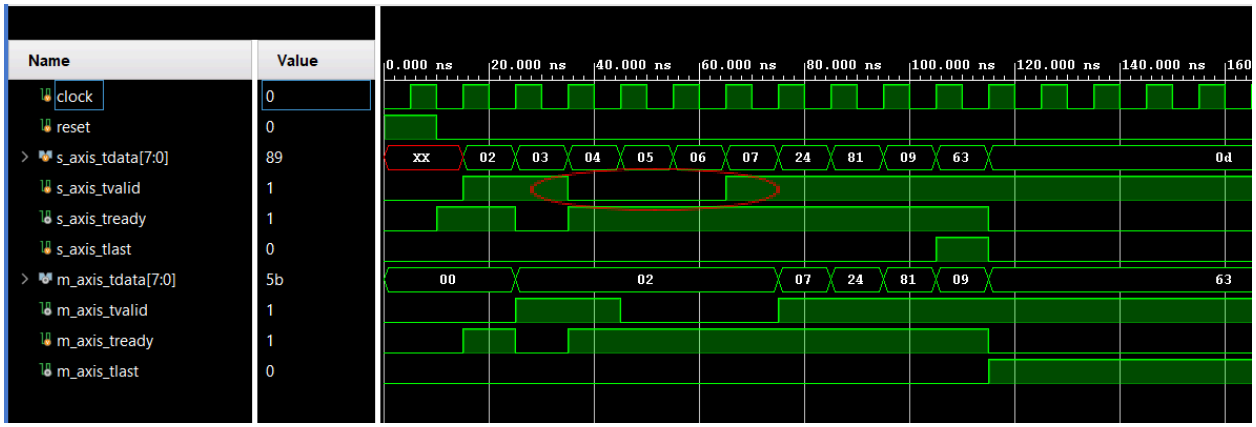
Sending and Receiving at full speed: (Operating at Normal Mode)

m_axis_tready (↑) → s_axis_tready (↑), s_axis_tvalid (↑), s_axis_tdata (increment's):



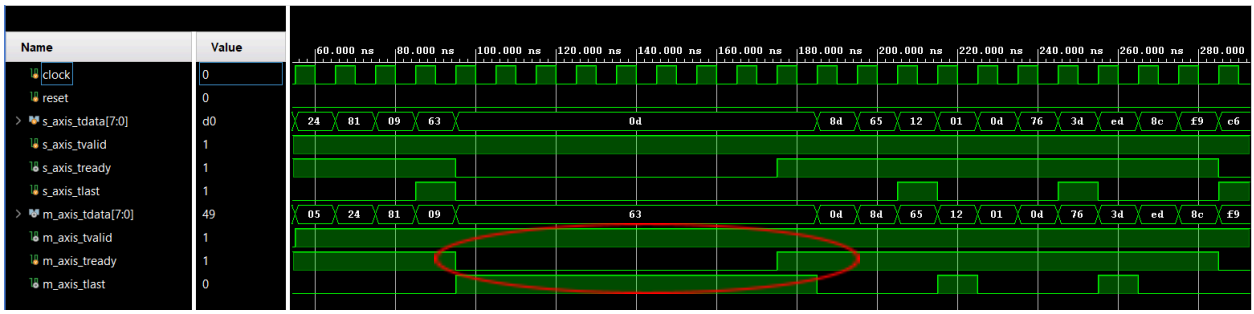
Slow Writer Fast Reader: (Hand Shake test)

m_axis_tready (↑) and s_axis_tvalid (↓) → m_axis_tvalid (↓)



Fast Writer Slow Reader: (Exerting Back Pressure)

m_axis_tready (↓) → s_axis_tready (↓)
m_axis_tvalid (pulled high)
m_axis_tdata (freeze): → s_axis_tdata (freezes)
s_axis_tvalid (pulled high)



Performance Details:

Name	Waveform	Period (ns)	Frequency (MHz)
clk1	{0.000 5.000}	10.000	100.000
clk2	{0.000 2.000}	4.000	250.000
clk3	{0.000 1.000}	2.000	500.000
clk4	{0.000 0.500}	1.000	1000.000

Worst Negative slack (WNS) :

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)
clk1	rise - rise	9.303	0.000	0	34
clk2	rise - rise	3.303	0.000	0	34
clk3	rise - rise	1.303	0.000	0	34
clk4	rise - rise	0.303	0.000	0	34

Worst Hold Slack (WHS) :

Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)
rise - rise	0.046	0.000	0	34
rise - rise	0.046	0.000	0	34
rise - rise	0.046	0.000	0	34
rise - rise	0.046	0.000	0	34

Worst Pulse Width Slack (WPWS) :

WPWS (ns)	TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
4.725	0.000	0	35
1.725	0.000	0	35
0.710	0.000	0	35
-0.290	-0.290	1	35

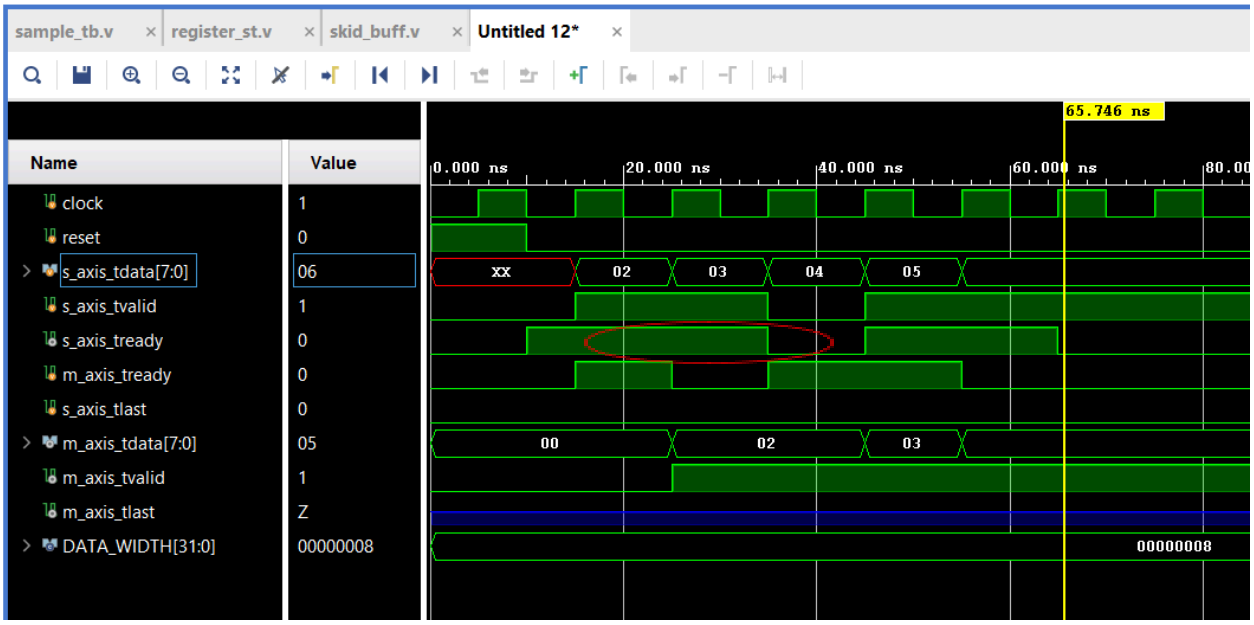
Resources Utilization:

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
synth_1	constrs_1	synth_design Complete!												3	10	0	0	0

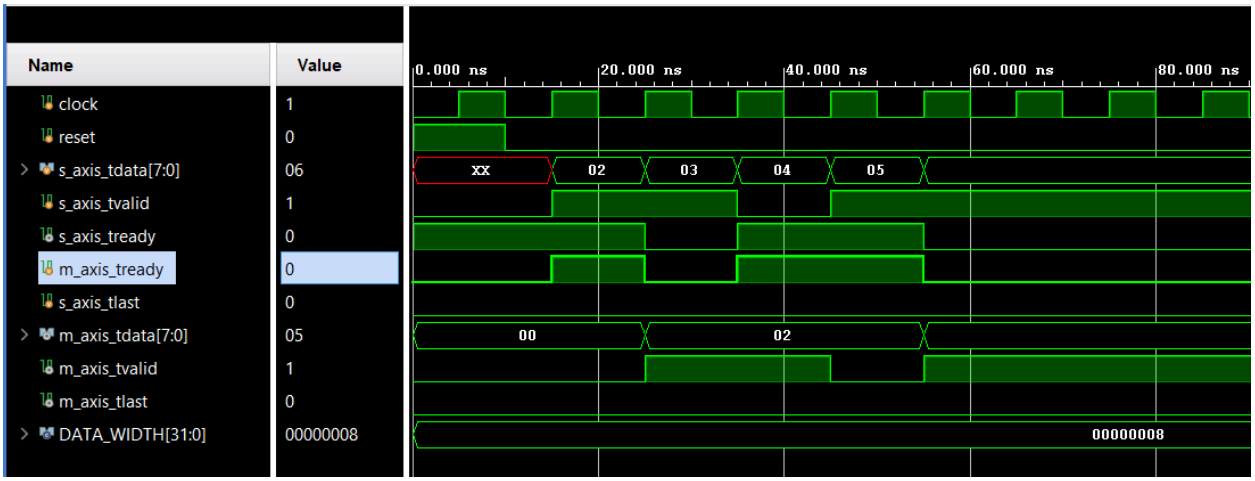
https://github.com/velicharlagokulkumar/vivado/tree/main/Axis_Reg

Skid Buffer vs AXIS Reg

Skid Buffer:



AXIS-Reg:



Assignment-2

AXI-Stream 2X1 Mux :

Introduction:

AXI-Stream 2x1 MUX is useful when we had a need to select and route two input stream data to the output stream based on the select line signal

Functionality:

AXI-Stream 2x1 MUX connects the s_axis_tdata_A to m_axis_tdata when the input sel is '0' And routes the s_axis_tdata_B to m_axis_tdata when sel is '1' following the AXI-stream Protocol standards

Whenever the slave is ready with the valid samples and also the master is ready to accept the data this AXI-Stream 2x1 MUX acts as a single stage pipeline mediator passing the input stream from the slave to the output as a stream to master based on the select line input.

Implementation:

HANDSHAKE MECHANISM:

Firstly due to the initial reset " valid_out " internal signal was assigned a initial value of '0' Due to this, ready becomes high and makes the "s_axis_tready" high making the module accept the data at all the cases of valid_out is zero.

```
always @ (posedge clk or posedge reset)
  begin
   if (reset == 1)
    begin
      valid_out <= 0;
          .
          .
assign ready = (valid_out == 0) | ((m_axis_tready == 1) & (valid_out == 1));
assign s_axis_tready= (ready == 1) & !(reset);
```

On the next clock cycle “valid_out” was assigned a value of enable signal value because of “ready” high (see below else)

If enable was “0” because of the s_axis_tvalid_A or s_axis_tvalid_B low or invalid select then valid_out becomes “0”, making the module to accept the signal continuously until valid_out becomes “HIGH”

```
.
.
else if (ready == 1)
     begin
       valid_out <= enable;
     end
   end
 assign enable = ((input_select == 0) & (ready == 1) & (s_axis_tvalid_A == 1)) |
                     ((input_select == 1) & (ready == 1) & (s_axis_tvalid_B == 1));
```

Once valid_out becomes high because of s_axis_tvalid_A or s_axis_tvalid_B based on input_select value (0 or 1) then it checks for m_axis_tready high (see above assign ready = ...) in order to pull and hold the ready high.

If the master was not in a position to accept the data the ready becomes low it in turn makes s_axis_tready_A or s_axis_tready_B low based on sel making the module to not accept the new data because we only have room for one sample unlike FIFO, if not deasserted our module is flood with the data and we may lose valid samples

Whenever the master is ready to accept the data (m_axis_tready == 1) ready becomes high as we already have valid_out == 1 (why….? because we already have a valid sample inside a reg as for as the AXIS protocol whenever we have valid data we should freeze the valid as high as so as data) and it in turn makes the s_axis_tready high making the module to accept the new data by passing the internal store data to the master through m_axis_tdata.

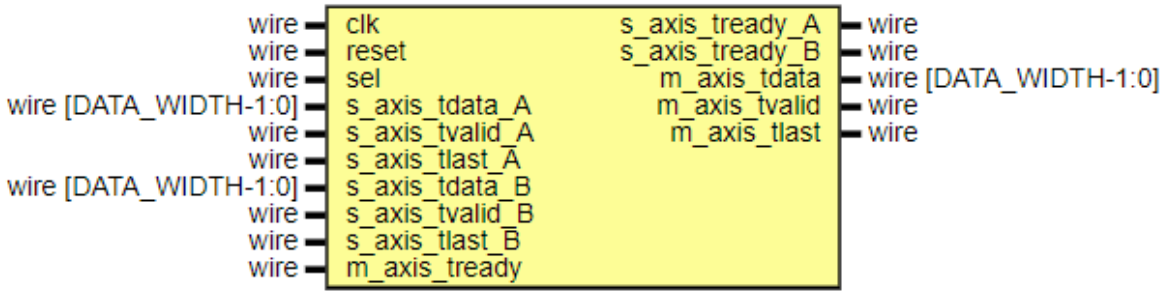
As Long as the valid_output is HIGH s_axis_tdata_A or s_axis_tdata_B will be store in the internal register

```
. .
 . . . .
   else if (valid_output == 1)
    begin
      data_out <= (input_select == 0)? s_axis_tdata_A : s_axis_tdata_B;
    end
```

Internal data_out will be continuously assigned to the out port m_axis_tdata whenever it was available

```
assign m_axis_tdata = data_out;
```

Module Block design



I/O Details:

Port name	Direction	Type	Description
clk	input	wire	Clock input.
reset	input	wire	Reset input.
sel	input	wire	Input select to select between A or B
s_axis_tdata_A	input	wire [DATA_WIDTH-1:0]	Input data A to be routed to the mux.
s_axis_tvalid_A	input	wire	Signal indicating valid input data A.
s_axis_tready_A	output	wire	Signal indicating readiness to accept input data A.

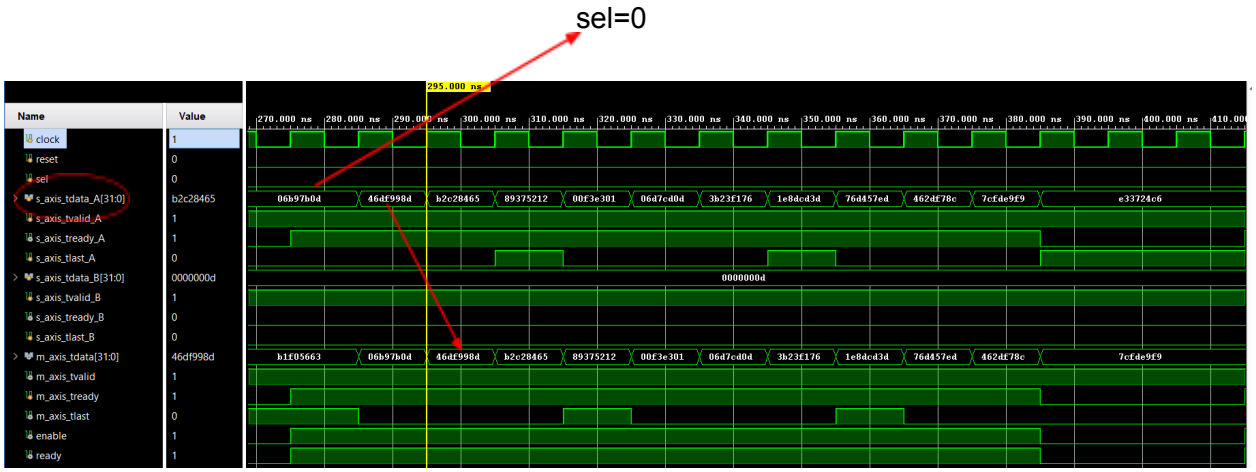
s_axis_tlast_A	input	wire	Signal indicating the last sample in the packet of input data A
s_axis_tdata_B	input	wire [DATA_WIDTH-1:0]	Input data B to be routed to the mux.
s_axis_tvalid_B	input	wire	Signal indicating valid input data B.
s_axis_tready_B	output	wire	Signal indicating readiness to accept input data B.
s_axis_tlast_B	input	wire	Signal indicating the last sample in the packet of input data B
m_axis_tdata	output	wire [DATA_WIDTH-1:0]	Output data read from the 2x1 MUX.
m_axis_tvalid	output	wire	Signal indicating valid output data.
m_axis_tready	input	wire	Signal indicating readiness to accept output data.
m_axis_tlast	output	wire	Signal indicating the last sample in a packet

Verification:

Sel: 0

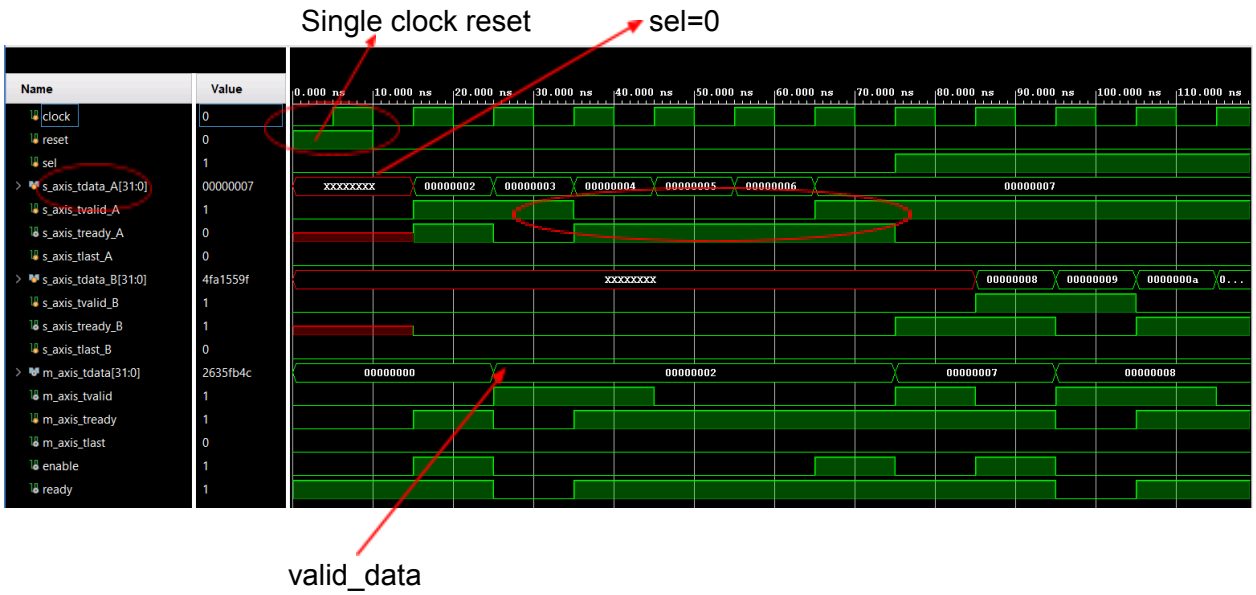
Sending and Receiving at full speed:(Operating at Normal Mode)

m_axis_tready (↑) → s_axis_tready_A (↑), s_axis_tvalid_A (↑), s_axis_tdata_A (increment's):



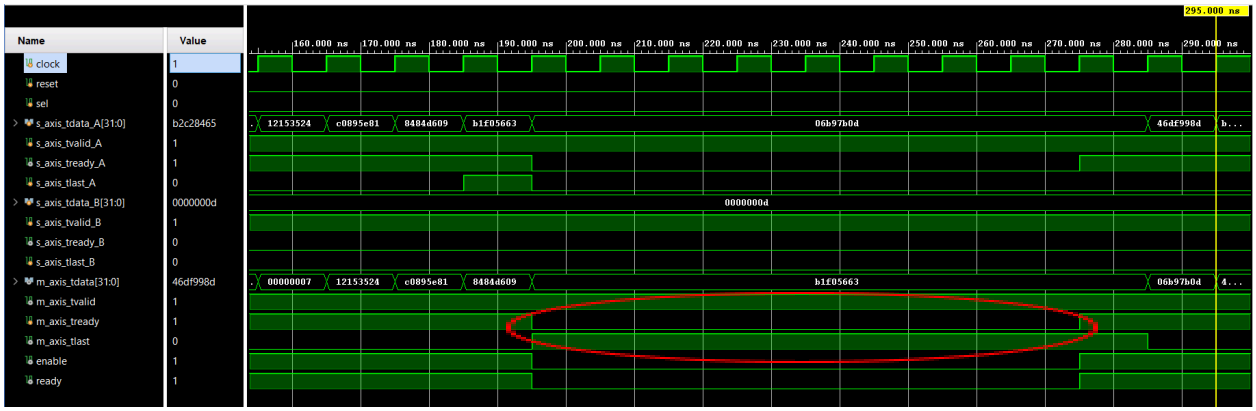
Slow Writer Fast Reader:(Hand Shake test)

m_axis_tready (↑) and s_axis_tvalid_A (↓) → m_axis_tvalid (↓)



Fast Writer Slow Reader: (Exerting Back Pressure)

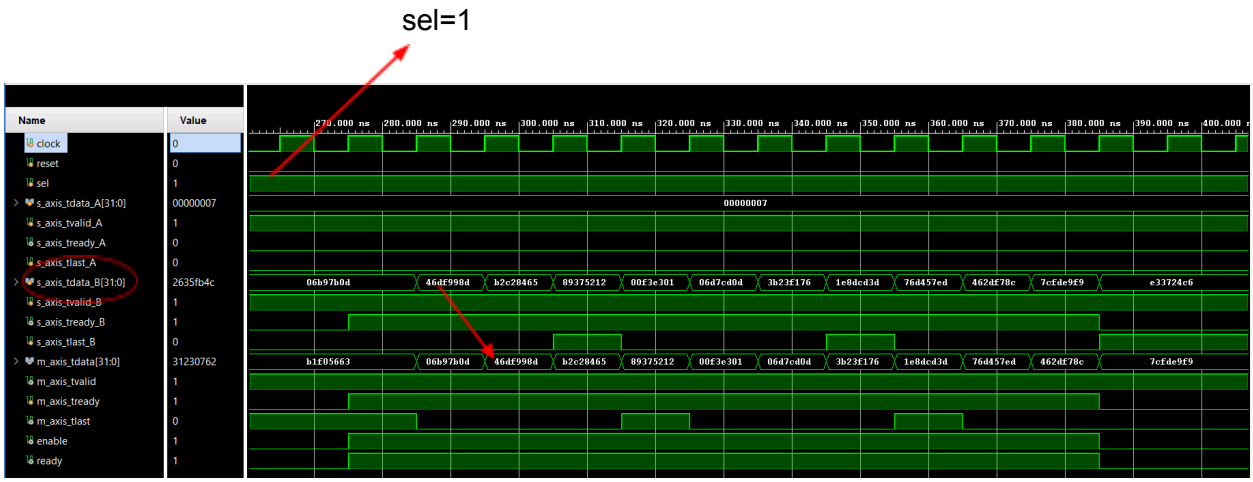
m_axis_tready (↓) → s_axis_tready_A (↓)
m_axis_tvalid (pulled high)
m_axis_tdata (freeze): → s_axis_tdata_A (freezes)
s_axis_tvalid_A (pulled high)



Sel: 1

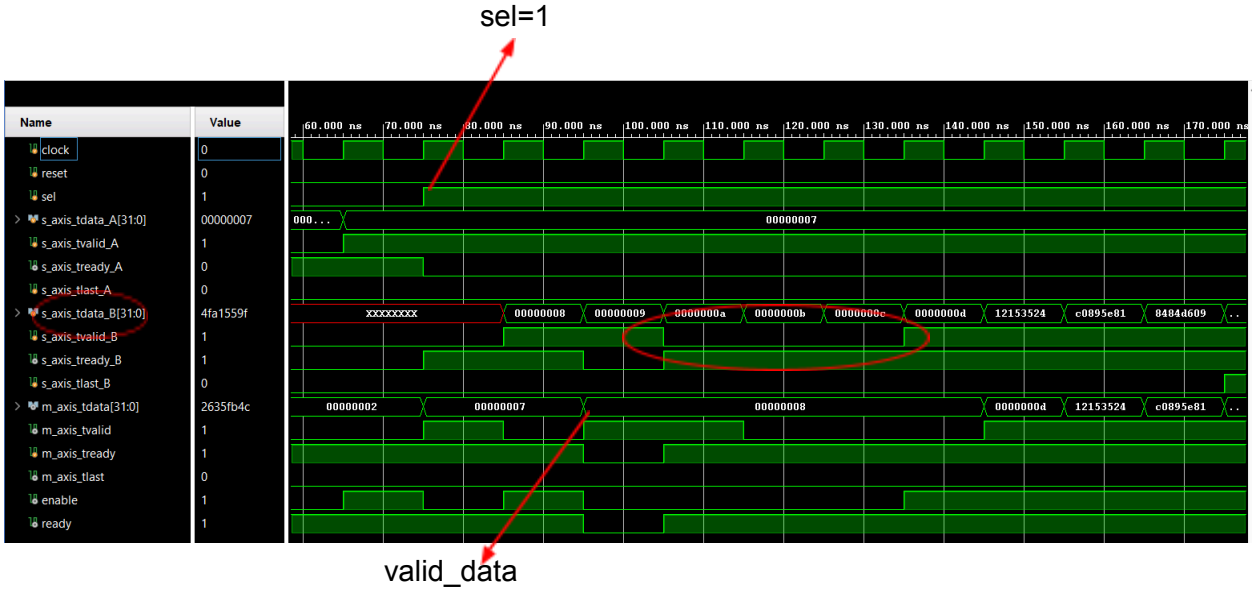
Sending and Receiving at full speed: (Operating at Normal Mode)

m_axis_tready (↑) → s_axis_tready_B (↑) and s_axis_tvalid_B (↑):



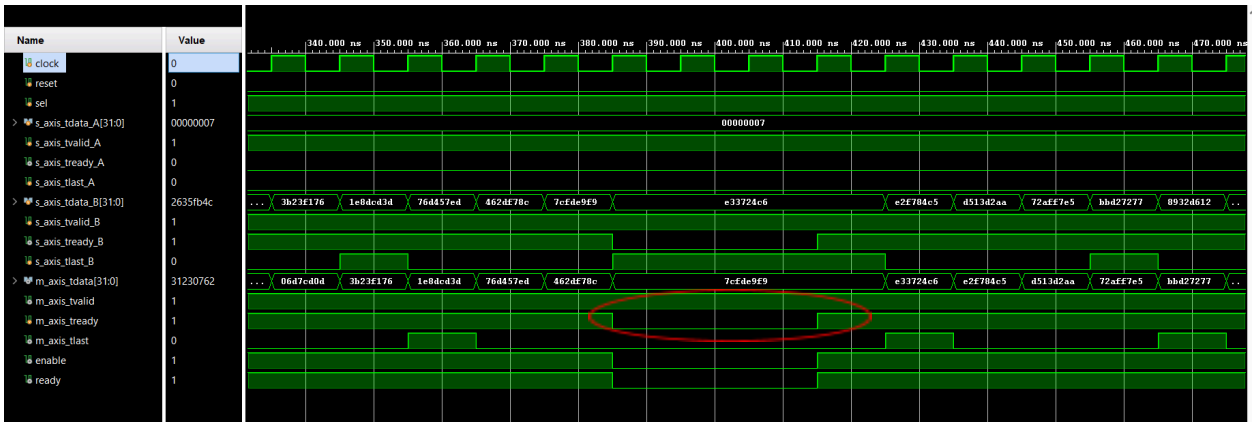
Slow Writer Fast Reader: (Hand Shake test)

m_axis_tready (↑) and s_axis_tvalid_B (↓) → m_axis_tvalid (↓)



Fast Writer Slow Reader: (Exerting Back Pressure)

m_axis_tready (↓) → s_axis_tready_B (↓)
m_axis_tvalid (pulled high)
m_axis_tdata (freeze): → s_axis_tdata_B (freezes)
s_axis_tvalid_B (pulled high)



Performance Details:

Name	Waveform	Period (ns)	Frequency (MHz)
clk1	{0.000 5.000}	10.000	100.000
clk2	{0.000 2.000}	4.000	250.000
clk3	{0.000 1.000}	2.000	500.000
clk4	{0.000 0.500}	1.000	1000.000

Worst Negative slack (WNS) :

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Edges (WHS)
clk1	rise - rise	9.304	0.000	0	66	rise - rise
clk2	rise - rise	3.304	0.000	0	66	rise - rise
clk3	rise - rise	1.304	0.000	0	66	rise - rise
clk4	rise - rise	0.304	0.000	0	66	rise - rise

Worst Hold Slack (WHS) :

WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)
0.055	0.000	0	66
0.055	0.000	0	66
0.055	0.000	0	66
0.055	0.000	0	66

Worst Pulse Width Slack (WPWS) :

TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
0.000	0	36
0.000	0	36
0.000	0	36
-0.290	1	36

Resources Utilization:

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM
synth_1	constrs_1	Synthesis Out-of-date												36	35	0

https://github.com/velicharlagokulkumar/vivado/tree/main/Axis_Mux

Assignment-3

(2 X 2048) Depth AXI Stream FIFO:

Introduction:

AXI Stream FIFO refers to a FIFO (First-In-First-Out) that is useful when we have a requirement of buffering the input data of certain data width and depth between AXIS master and AXIS slave.

FIFO buffer, meaning it stores data in the order it arrives, and data is retrieved in the same order it was stored—first in, first out.

This is useful for managing data flow between different components or modules in a system where there might be variations in data arrival rates or processing times.

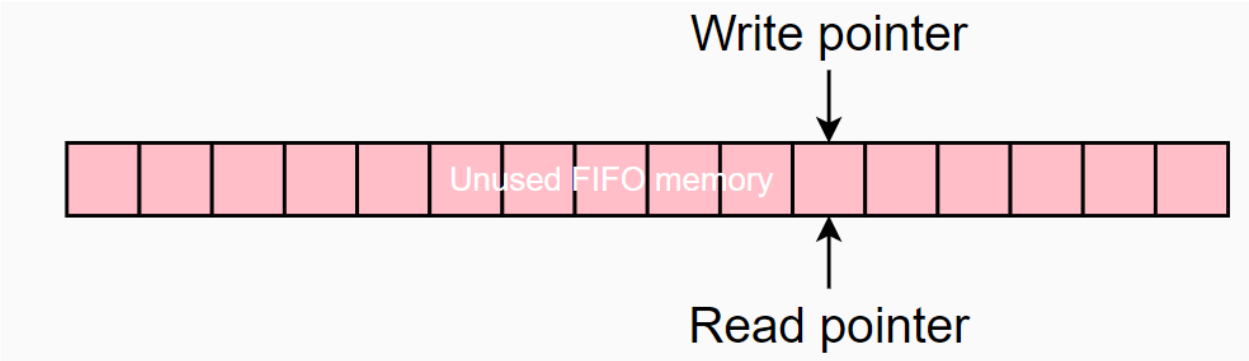
The FIFO includes mechanisms for flow control to ensure that data is transferred at a rate that both the sender and receiver can handle. This helps prevent data loss

The FIFO typically includes some level of buffering to store incoming data temporarily until it can be processed or transferred to the next stage in the system. This buffering helps smooth out any discrepancies in data arrival rates or processing speeds between different components.
Input samples

Functionality:

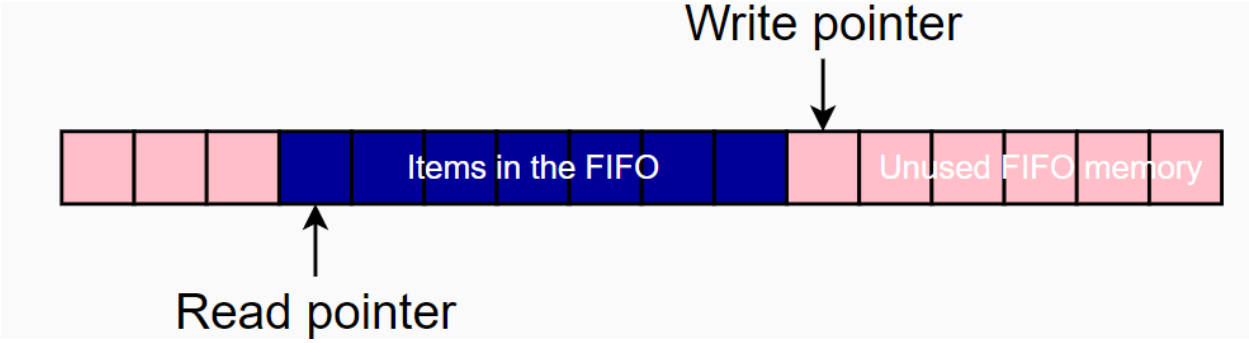
Circular Buffer:

Whenever the read and write pointers are identical, as in Fig 1 below, we'll use that as the indication that the buffer is empty. Initially, both pointers will point to the same address in the buffer (zero)



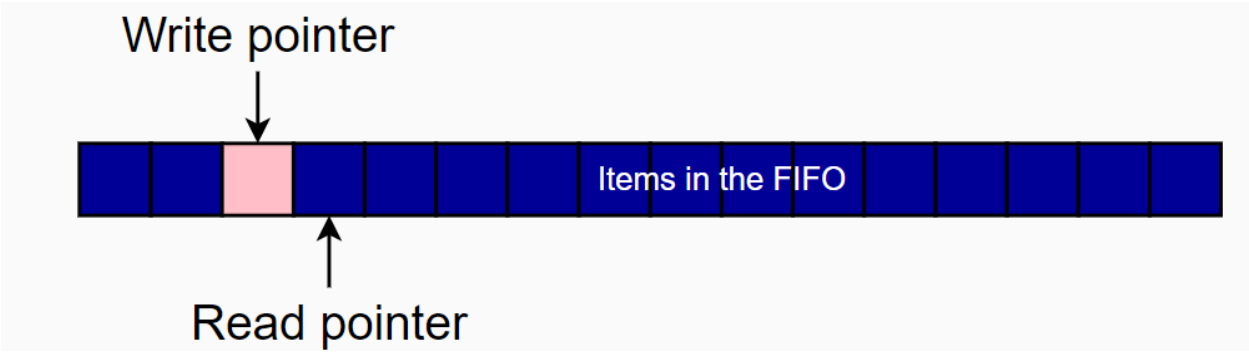
When an item is written to the buffer, the write pointer is incremented. This pointer then always points to the item not yet written to.

When an item is read from the buffer, the read pointer is incremented. This pointer always references the next item to be read.



What happens when the write or read pointer gets to the end of the buffer? The pointer in question simply wraps around to the beginning of the buffer. Because the buffer pointers just wrap around, this type of buffer is called a circular buffer.

You can tell this FIFO is full, because write pointer one plus will equal the read pointer, as



Implementation:

Inside the module, there is an array of memory elements that stores the data in the FIFO. There are two pointer variables, `wrPtr` and `rdPtr`, that keep track of the write and read positions in the FIFO, respectively. These pointers have one more bit than the `PtrWidth` parameter, known as the wrap bit. This is because when the pointer increments and reaches the maximum value, it wraps around to the beginning of the FIFO

The `always_comb` block updates the next values of the write and read pointers based on the current values and the write and read enable signals. If the write enable signal is high, the write pointer is incremented by one. If the read enable signal is high, the read pointer is incremented by one.

The `always_ff` block updates the write and read pointers on the rising edge of the clock. If the reset signal is low, the pointers are set to 0. Otherwise, the pointers are updated to their next values. Additionally, the `writeData` input is written to the memory element at the position specified by the write pointer, `readData` output is read from memory element at the position specified by the readpointer

Since input and output ports are operating independently we need to somehow keep track of the t_last signal of input, so wrPtr at where the t_last becomes high will be store in t_last_reg memory and when the rdPtr reaches the Ptr value stored inside the memory it will make t_last high at the output side

The `empty` output is computed by checking if the write and read pointers are equal. The `full` output is computed by checking if the write and read pointers differ in their most significant bit (the wrap bit) but are equal in their lower bits.

Empty:

```
(wrPtr[PtrWidth]==rdPtr[PtrWidth]) &&
(wrPtr[PtrWidth-1:0]==rdPtr[PtrWidth-1:0]);
```

Let us start wrPtr and rdPtr and stop the wrptr and a some point of time when ever the above condition satisfies that means we read all the samples in FIFO and it was empty.

Let PtrWidth == 5	wrPtr	rdPtr
	00000	00000
	00001	00000
	00010	00001
	00011	00010
	00011	00011 ⇒ empty

```
mem[wrPtr[PtrWidth-1:0]] <= writeData;
```

Full:

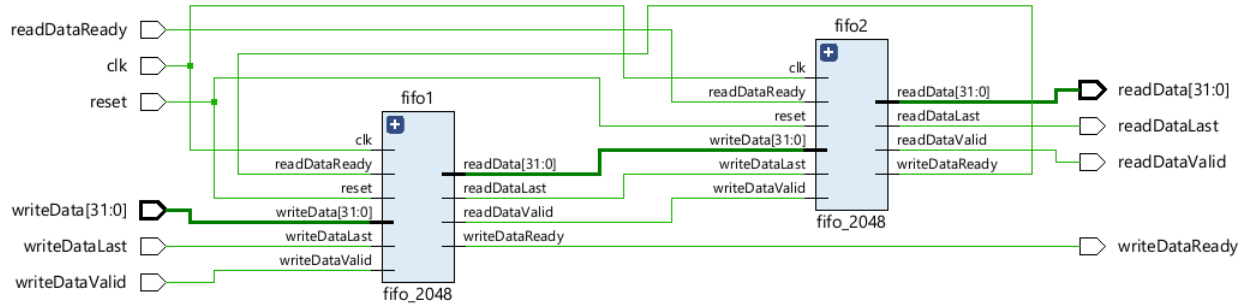
```
(wrPtr[PtrWidth]!= rdPtr[PtrWidth]) &&
(wrPtr[PtrWidth-1:0]==rdPtr[PtrWidth-1:0]);
```

Let us start wrPtr and don't increment rdPtr and some point of time above condition satisfies that means FIFO was full.

Let PtrWidth == 5	wrPtr	rdPtr
	00000	00000
	00001	00000
	00010	00000
	00011	00000
	00100	00000
	.	.
	.	.
	.	.

10000 00000 ⇒ full

Module Block Design:



I/O Details:

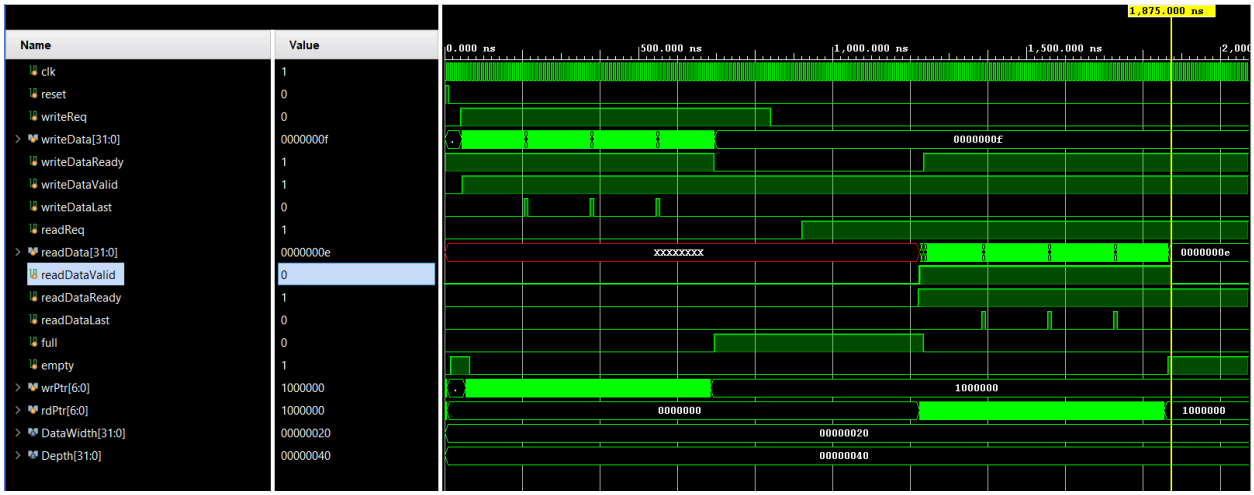
Port name	Direction	Type	Description
clk	input	wire	Clock input.
reset	input	wire	Reset input.
writeData	input	Wire [DataWidth-1:0]	Input data to be written into the FIFO.
writeDataValid	input	wire	Signal indicating valid input data.
writeDataReady	output	wire	Signal indicating readiness to accept input data.
writeDataLast	input	wire	Signal indicating the last data of the packet
readData	output	Wire [DataWidth-1:0]	Output data read from the FIFO.

readDataValid	output	wire	Signal indicating valid output data.
readDataReady	input	wire	Signal indicating readiness to accept output data.
readDataLast	output	wire	Signal indicating the last data in a packet

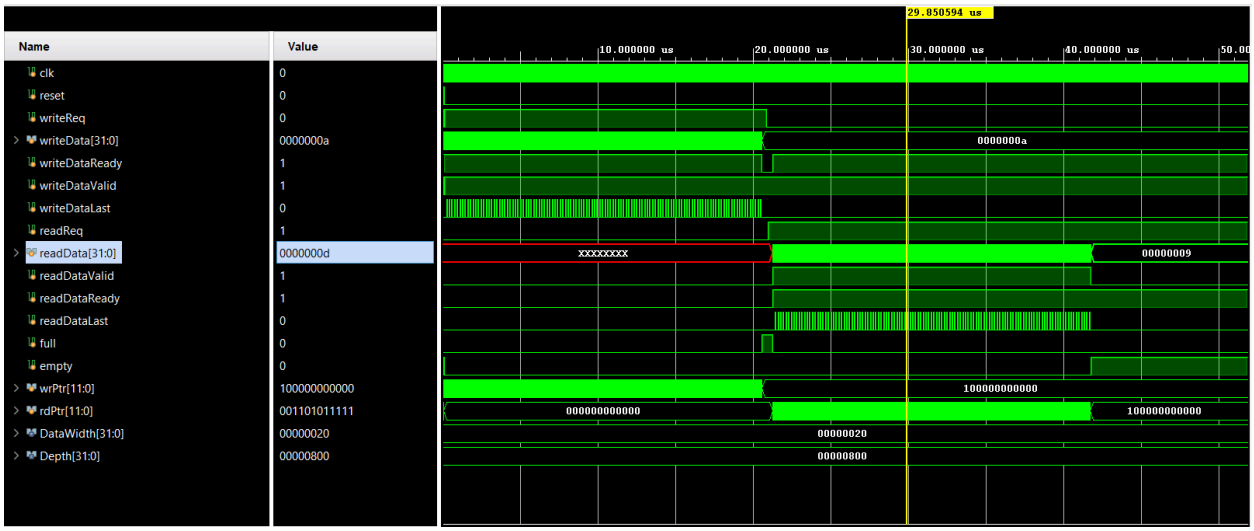
Verification:

Single FIFO: (With WriteReq, ReadReq, Full, Empty)

Parameter (Depth=64 samples):



Parameter (Depth=2048 samples):



Name	Waveform	Period (ns)	Frequency (MHz)
clk1	{0.000 5.000}	10.000	100.000
clk2	{0.000 2.000}	4.000	250.000
clk3	{0.000 1.000}	2.000	500.000
clk4	{0.000 0.500}	1.000	1000.000

Performance Details: (single FIFO)

Worst Negative slack (WNS) :

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)
clk1	rise - rise	8.107	0.000	0	2134
clk2	rise - rise	2.107	0.000	0	2134
clk3	rise - rise	0.107	0.000	0	2134
clk4	rise - rise	-0.893	-110.381	625	2134

Worst Hold Slack (WHS) :

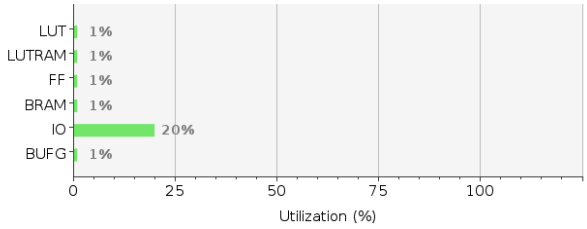
Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)
rise - rise	0.049	0.000	0	2134
rise - rise	0.049	0.000	0	2134
rise - rise	0.049	0.000	0	2134
rise - rise	0.049	0.000	0	2134

Worst Pulse Width Slack (WPWS) :

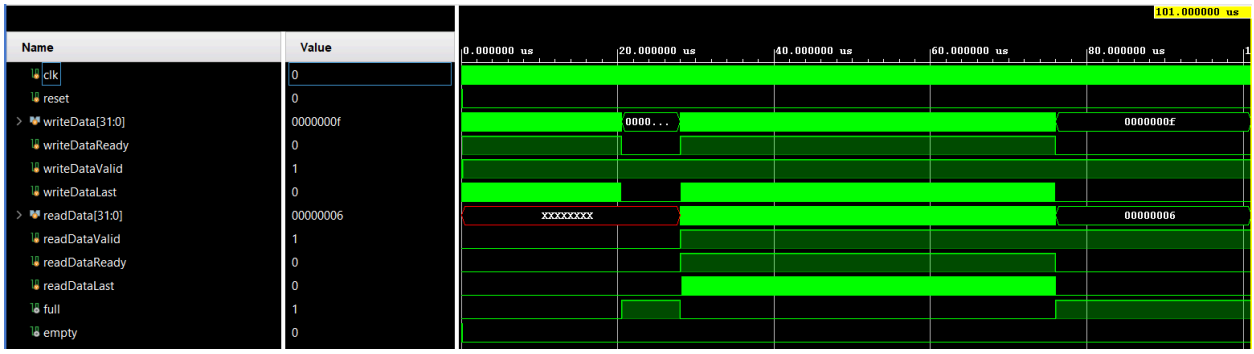
TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
0.000	0	317
0.000	0	317
0.000	0	317
-18.522	261	317

Resources Utilization:

Resource	Utilization	Available	Utilization %
LUT	371	230400	0.16
LUTRAM	256	101760	0.25
FF	56	460800	0.01
BRAM	2	312	0.64
IO	72	360	20.00
BUFG	1	544	0.18

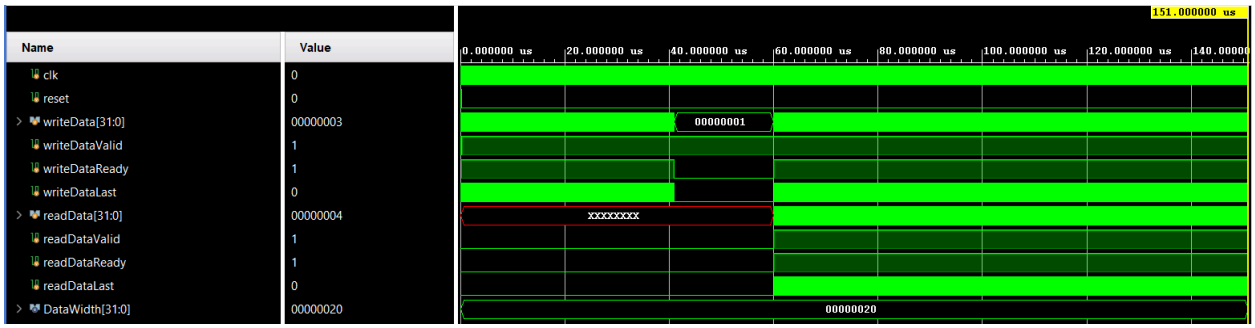


Single FIFO 2048 : (Without WriteReq, ReadReq)

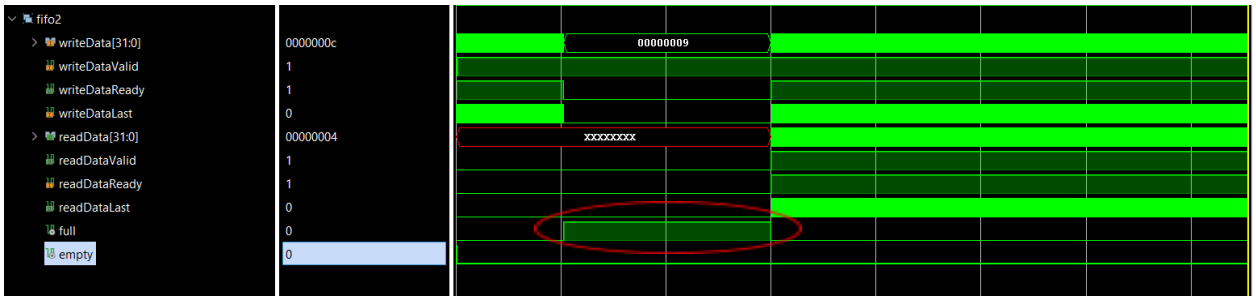


Cascaded (2 x 2048) :

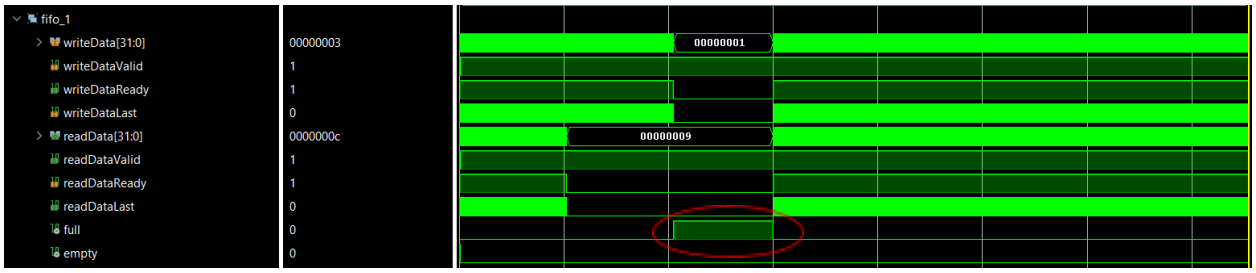
fifo_top



fifo_2:

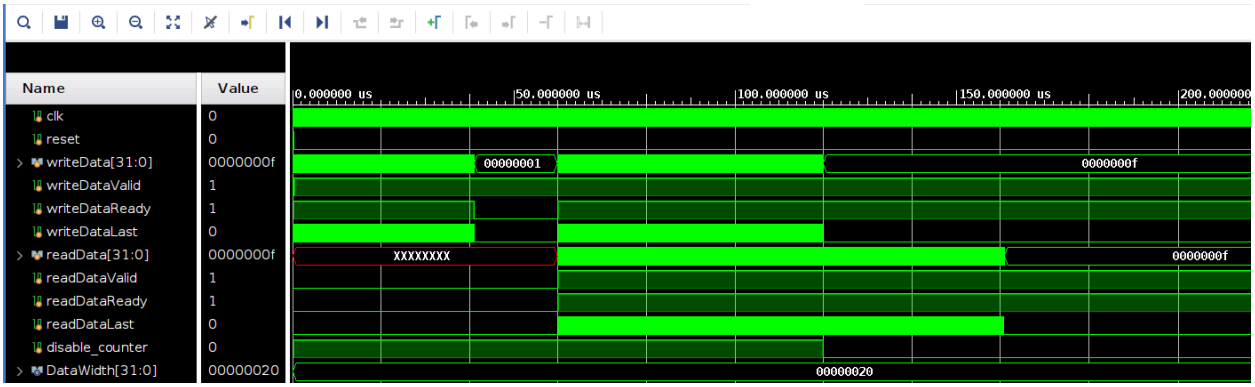


fifo_1:



Throttled Response:

fifo_top



Performance Details: (Dual FIFO)

Name	Waveform	Period (ns)	Frequency (MHz)
clk1	{0.000 5.000}	10.000	100.000
clk2	{0.000 2.000}	4.000	250.000
clk3	{0.000 1.000}	2.000	500.000
clk4	{0.000 0.500}	1.000	1000.000

Worst Negative slack (WNS) :

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)
clk1	rise - rise	8.062	0.000	0	4314
clk2	rise - rise	2.062	0.000	0	4314
clk3	rise - rise	0.062	0.000	0	4314
clk4	rise - rise	-0.938	-525.479	2310	4314

Worst Hold Slack (WHS) :

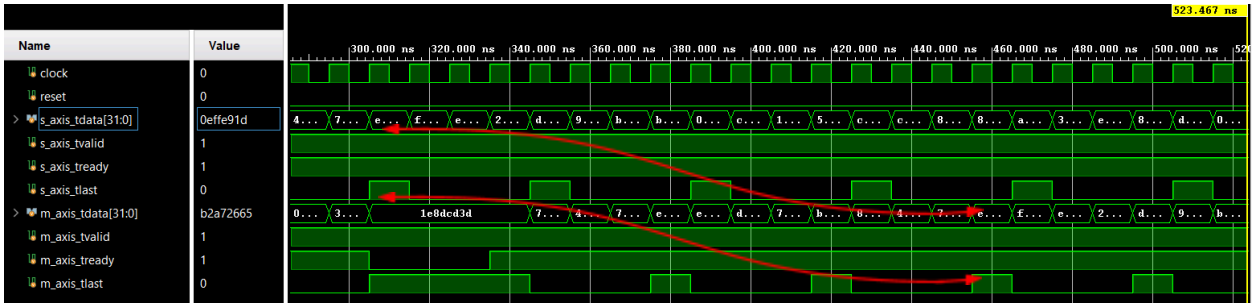
Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)
rise - rise	0.032	0.000	0	4314
rise - rise	0.032	0.000	0	4314
rise - rise	0.032	0.000	0	4314
rise - rise	0.032	0.000	0	4314

Worst Pulse Width Slack (WPWS) :

WPWS (ns)	TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
4.458	0.000	0	633
1.458	0.000	0	633
0.431	0.000	0	633
-0.569	-36.754	521	633

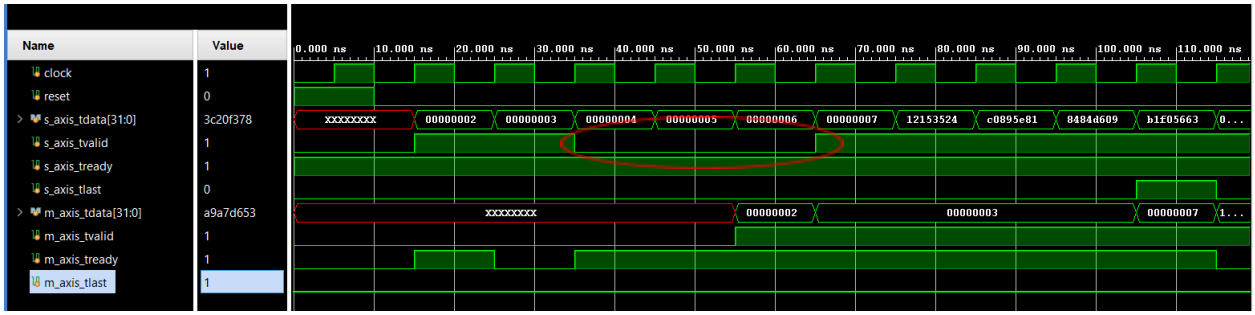
Sending and Receiving at full speed: (Operating at Normal Mode)

m_axis_tready (↑) → s_axis_tready_B (↑) and s_axis_tvalid_B (↑):

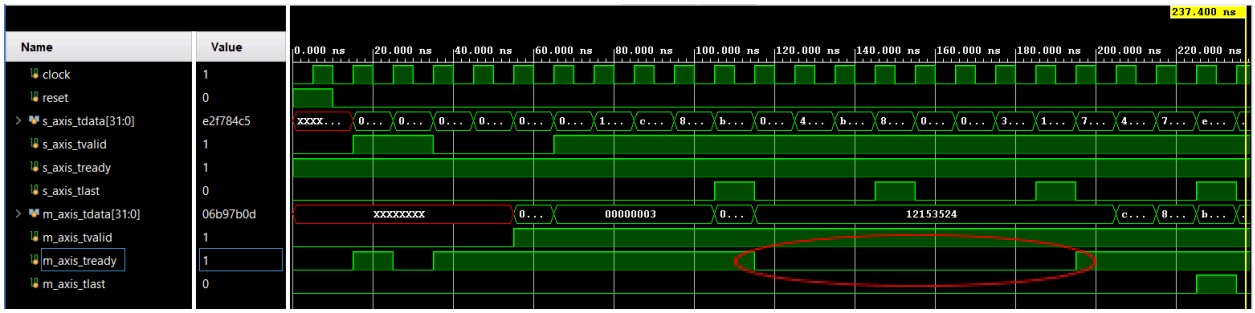


Slow Writer Fast Reader: (Hand Shake test)

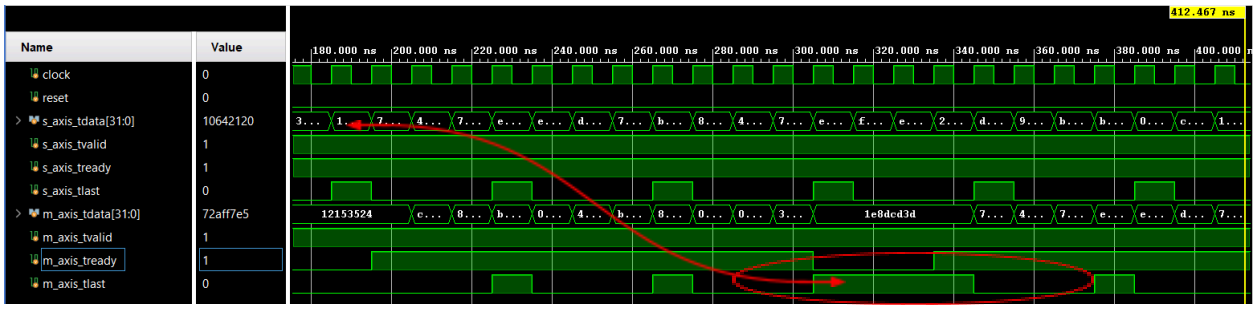
m_axis_tready (↑) and s_axis_tvalid_B (↓) → m_axis_tvalid (↓)



Fast Writer Slow Reader: (Exerting Back Pressure)

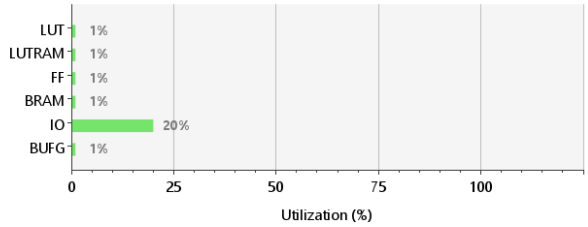


m_axis_tready (↓) for the t_last output sample:



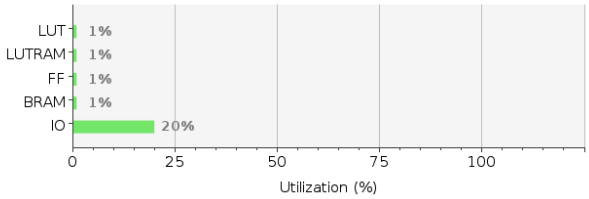
Resources Utilization: (Design vs XILINX FIFO_4096)

Resource	Utilization	Available	Utilization %
LUT	1444	230400	0.63
LUTRAM	1024	101760	1.01
FF	140	460800	0.03
BRAM	4	312	1.28
IO	72	360	20.00
BUFG	1	544	0.18



Design

Resource	Utilization	Available	Utilization %
LUT	60	230400	0.03
LUTRAM	1	101760	0.00
FF	110	460800	0.02
BRAM	4	312	1.28
IO	71	360	19.72

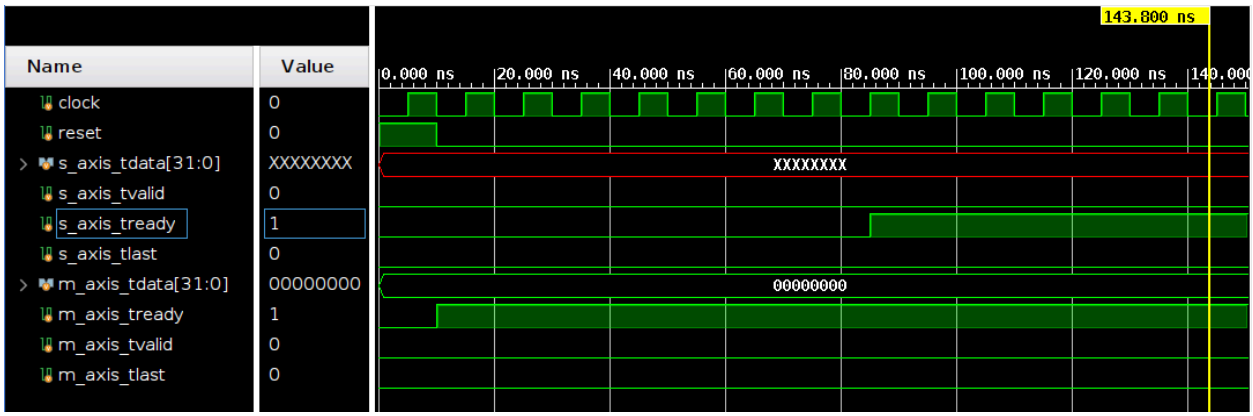


Xilinx FIFO

Can be reduced….

https://github.com/velicharlagokulkumar/vivado/tree/main/Axis_fifo

xilinx fifo ....?



Module 3: RTL Designs with pipelining

Short Note:

- When the module's operation is complex then it is suggested that it should be split over multiple clock cycles to ease timing which we call it as pipelining.

- 3 stage pipelined design that uses three clock cycles instead of one should not delimit the throughput.

Assignment-1

Square of Difference:

```
module squarediffmult # (parameter SIZEIN = 16)
 (
  input clk,
  input ce,
  input rst,
  input signed [SIZEIN-1:0] a, b,
  output signed [2*SIZEIN+1:0] square_out
 );

 reg signed [SIZEIN-1:0] a_reg, b_reg;
 reg signed [SIZEIN:0] diff_reg;
 reg signed [2*SIZEIN+1:0] m_reg, p_reg;

always @(posedge clk)
 if (rst)
  begin
     a_reg <= 0;
     b_reg <= 0;
     diff_reg <= 0;
     m_reg <= 0;
     p_reg <= 0;
  end
 else
 if (ce)
   begin
     a_reg <= a;
     b_reg <= b;
     diff_reg <= a_reg - b_reg;
     m_reg <= diff_reg * diff_reg;
     p_reg <= m_reg;
   end
 assign square_out = p_reg;
```

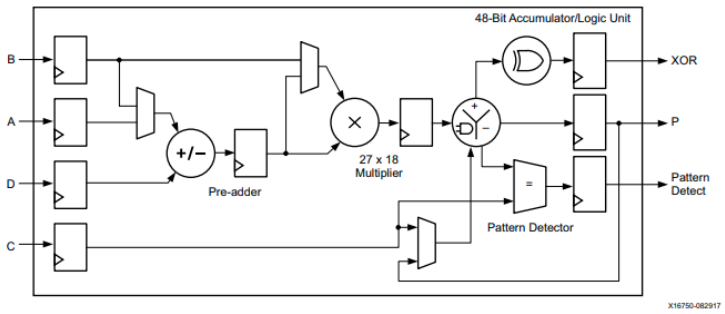
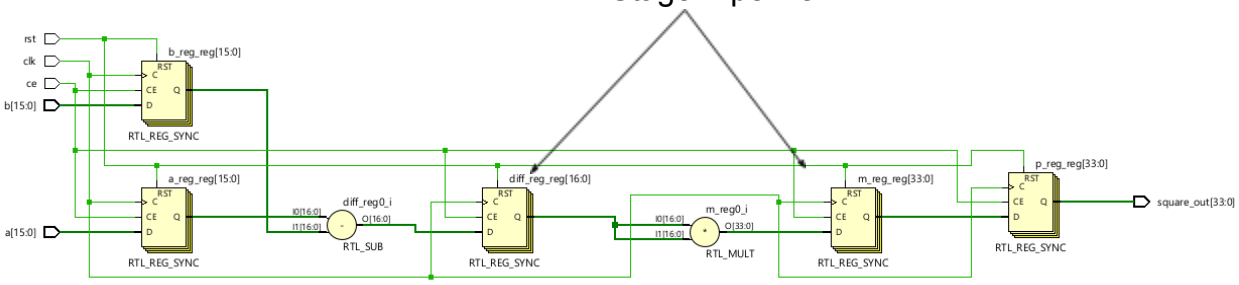


Figure 1-1: **Basic DSP48E2 Functionality**

2-Stage Pipeline

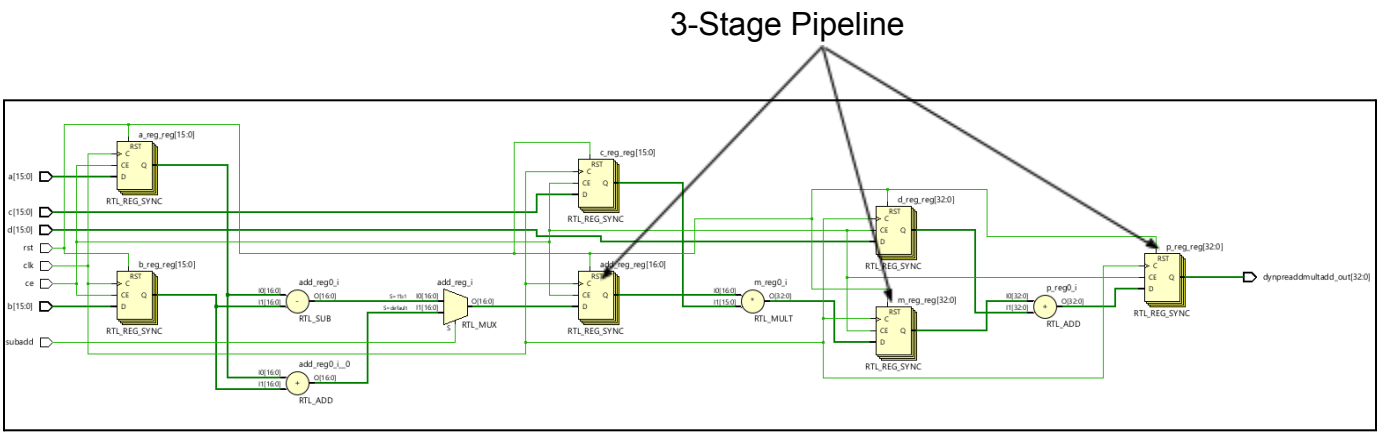


Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												0	0	0	0	1

Uses 1-DSP (For complex Arithmetic operations, advantage of the pipelining capabilities of DSP blocks)

Pre-Adder Dynamically Configured Followed by Multiplier and Post-Adder

```
module test # (parameter SIZEIN = 16)
(
input clk, ce, rst, subadd,
input signed [SIZEIN-1:0] a, b, c, d,
output signed [2*SIZEIN:0] dynpreaddmultadd_out
);
// Declare registers for intermediate values
reg signed [SIZEIN-1:0] a_reg, b_reg, c_reg;
reg signed [SIZEIN:0] add_reg;
reg signed [2*SIZEIN:0] d_reg, m_reg, p_reg;

always @(posedge clk)
begin
if (rst)
begin
a_reg <= 0;
b_reg <= 0;
c_reg <= 0;
d_reg <= 0;
add_reg <= 0;
m_reg <= 0;
p_reg <= 0;
end
else if (ce)
begin
a_reg <= a;
b_reg <= b;
c_reg <= c;
d_reg <= d;
if (subadd)
add_reg <= a_reg - b_reg;
else
add_reg <= a_reg + b_reg;
m_reg <= add_reg * c_reg;
p_reg <= m_reg + d_reg;
end
end

assign dynpreaddmultadd_out = p_reg; // Output accumulation result

endmodule
```

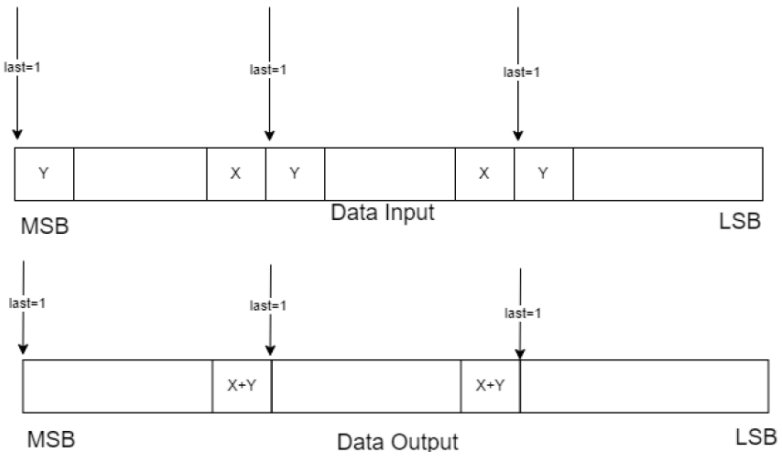


Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
synth_1	constrs_1	synth_design Complete!												0	0	0	0	1

Assignment-2

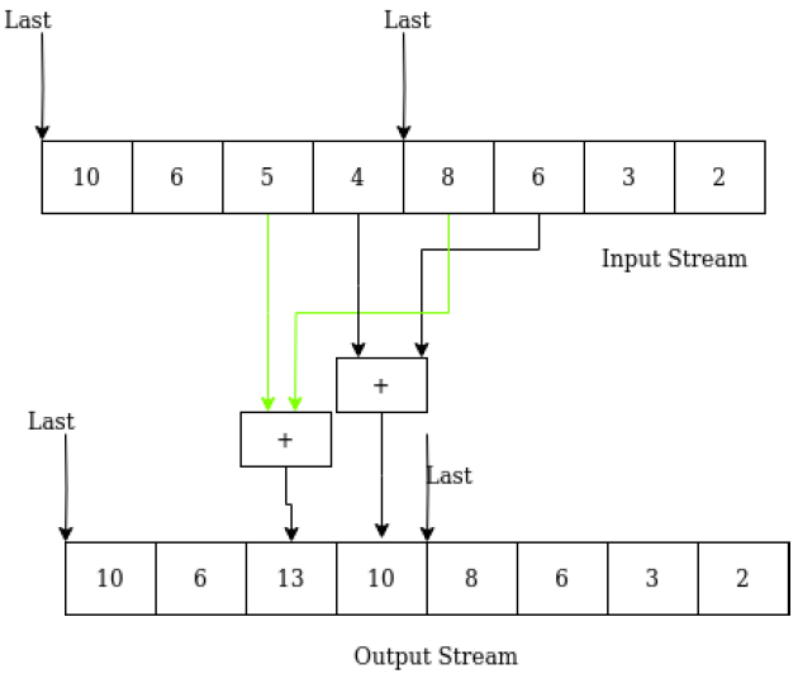
Packet Processor:

Functionality:



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 of 16 bits, in which the first 8 LSB bits indicate the length of the packet (n) and the MSB 8 bits specify “k” value which will be used for packet processing.
- Configuration can be read only once when the system comes out of reset state.



- 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.
- Assume the maximum memory depth is 512 and the packet length always should be less than or equal to 512.
- 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 first 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

Implementation:

If the config is given as 16'h0410;
Then 16 bytes are treated as a One Packet (Packet Length) and k = 4

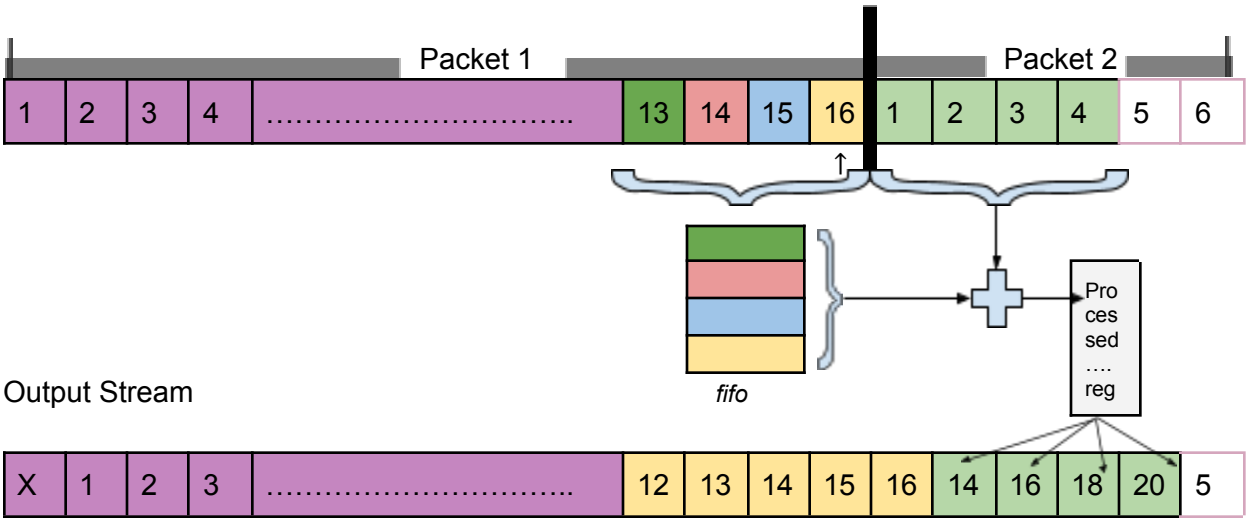
Module has an internal counter which increments receiving every valid sample coming from the upstream and Buffer will be used to store for processing with next packet and it will be initialized only when count reaches (Packet Length - K), since we are interested in processing the last K bytes with the first K bytes of the Next Packet, Last K bytes are stored in a Buffer.

Whenever we hit the last sample (t_last HIGH). Then we start retrieving our DATA from the Buffer to process the samples with the next packet, since Reading from the buffer takes one clock cycle we are initializing it at the t_last sample received clock edge itself so that we are ready with the data on the next edge where our next packet samples are arriving.

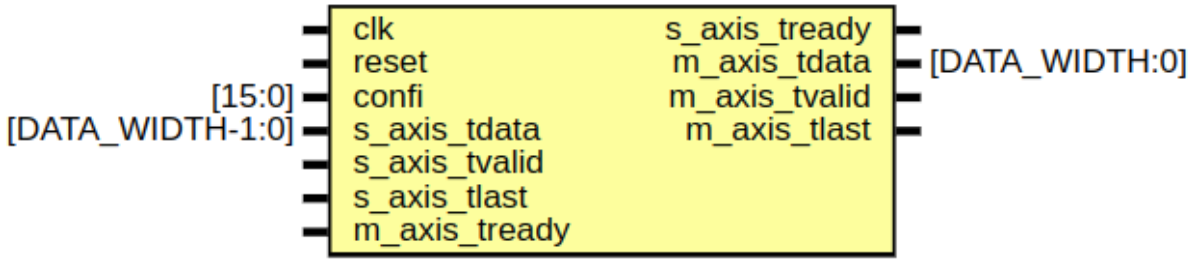
Buffer will be controlled to give the next stored sample only when the handshake happens to ensure that we are not processing with invalid receiving upstream samples.

When we received the next packet first sample after handshake we add that stream data with the buffer data that is already handy and store it in a processed register (Flopped) we get the processed data in the next clock cycle, However we are having the output delayed by one clock cycle with respect to the input this one clock cycle delay will compensate there simple by switching the output stream to this Flip Flop. There is another counter which increments when processing happens and when this count reaches K that means we are done with processing all samples and we need to send the remaining samples as it is and the output stream immediately switches to the input stream by some control signals internally.

Input Stream



Module Block Design:



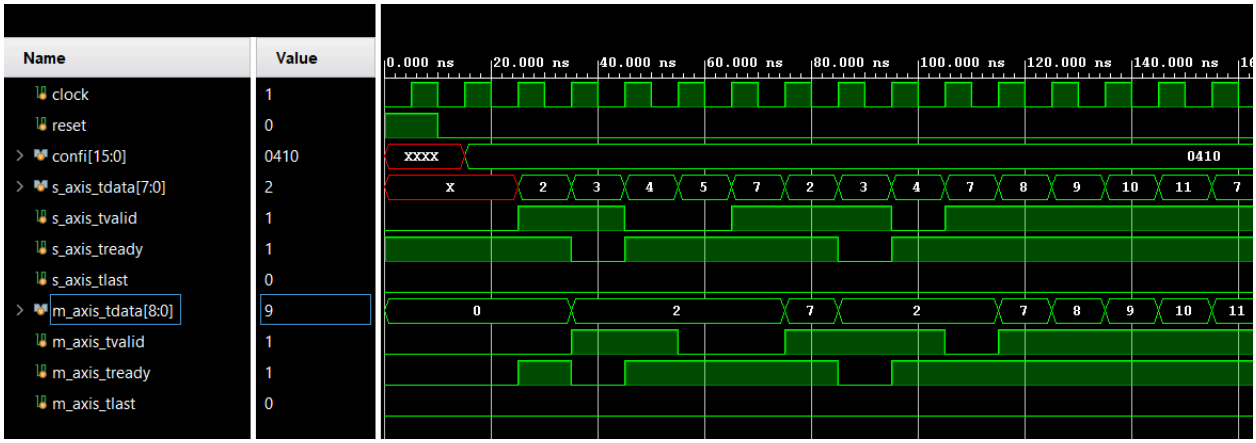
I/O Details:

Port name	Direction	Type	Description
clk	input	wire	Clock input.
reset	input	wire	Reset input.
confi	input	Wire [15:0]	Input configuration
s_axis_tdata	input	Wire [DATA_WIDTH-1:0]	Input data to be written into the module.
s_axis_tvalid	input	wire	Signal indicating valid input data.
s_axis_tready	output	wire	Signal indicating readiness to accept input data.
s_axis_tlast	input	wire	Signal indicating the last data of the packet
m_axis_tdata	output	Wire [DATA_WIDTH:0]	Output data read from the module.

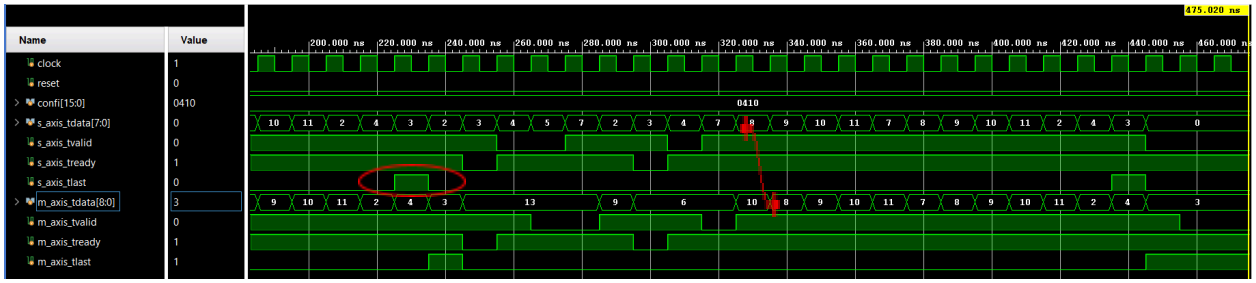
m_axis_tvalid	output	wire	Signal indicating valid output data.
m_axis_tready	input	wire	Signal indicating readiness to accept output data.
m_axis_tready	output	wire	Signal indicating the last data in a packet

Verification:

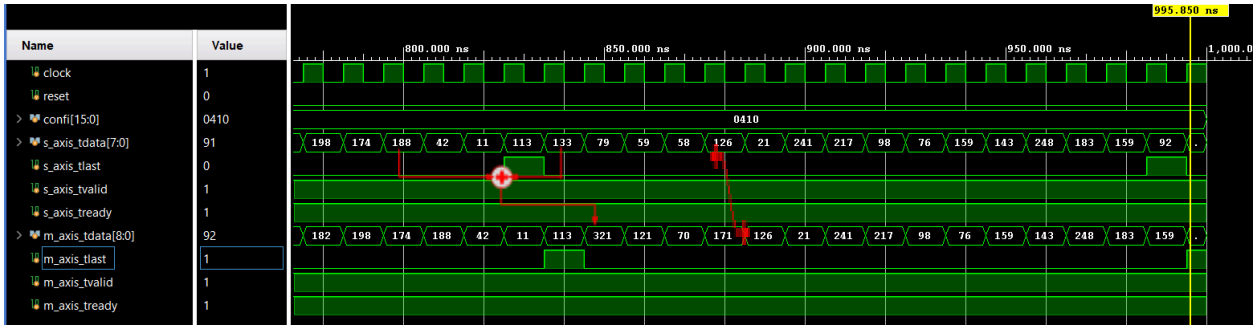
Basic Handshake Test:



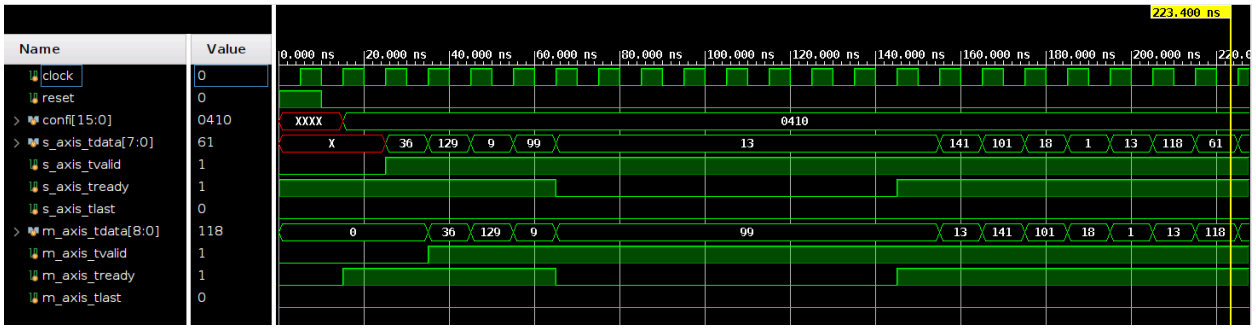
Functionality Test:



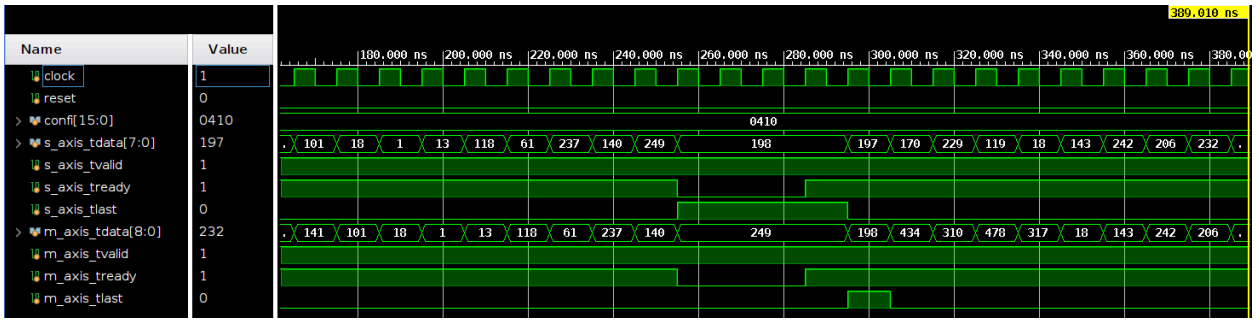
Sending and Receiving at full speed: (Operating at Normal Mode)



Fast Writer Slow Reader: (Exerting Back Pressure)



m_axis_tready (↓) for the t_last input sample



Performance Details:

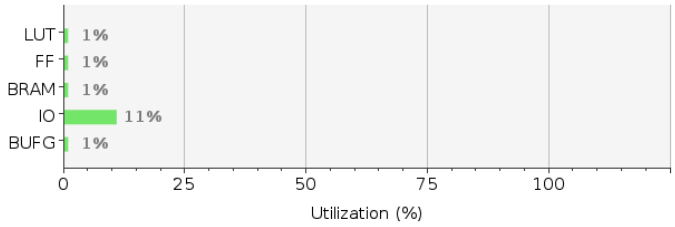
Worst Negative slack (WNS) & Worst Hold Slack (WHS) :

Name	Waveform	Period (ns)	Frequency (MHz)
clk1	{0.000 5.000}	10.000	100.000
clk2	{0.000 2.000}	4.000	250.000
clk3	{0.000 1.000}	2.000	500.000
clk4	{0.000 0.500}	1.000	1000.000

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Edges (WHS)	WHS (ns)
clk1	rise - rise	8.410	0.000	0	162	rise - rise	0.031
clk2	rise - rise	2.410	0.000	0	162	rise - rise	0.031
clk3	rise - rise	0.410	0.000	0	162	rise - rise	0.031
clk4	rise - rise	-0.590	-13.960	65	162	rise - rise	0.031

Resources Utilization :

Resource	Utilization	Available	Utilization %
LUT	157	230400	0.07
FF	77	460800	0.02
BRAM	0.50	312	0.16
IO	41	360	11.39
BUFG	1	544	0.18



➔ https://github.com/velicharlagokulkumar/vivado/tree/main/Data_packer

Module 4: RTL Designs for fixed point arithmetic

Short Notes

- An XQN format number is an 1+X+N bit two's complement binary number; a sign bit followed by X integer bits followed by an N bit mantissa (fraction).
- XQN format can be used to express numbers in the range (-2^X) to $(2^X - 2^{(-N)})$.
- An equivalent notation using the System Generator Fix format, defined as Fix***word_length _fractional_length***, would be Fix(1+X+N)_N.
- A number using Q15 format is equivalent to a number using Fix16_15 representation, and a number in 1Q15 format is equivalent to a number using Fix17_15 representation.

2QN Format: Example of a 2Q6 (or Fix9_6) Format Number

	(Sign) Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
+1	0	0	1	0	0	0	0	0	0
-1	1	1	1	0	0	0	0	0	0
+pi (3.140625)	0	1	1	0	0	1	0	0	1
-pi	1	0	0	1	1	0	1	1	1
				Fractional Bits					

Signed Arithmetic:

- The keyword “signed” is used to declare verilog object data types as signed types.
- This is for better modeling a signed arithmetic when the MSB is assumed as the sign bit
- When adding two signed numbers, the addend and augend may have different lengths. In this case, we have to extend the sign bit of the shorter number otherwise the result may not be correct.
- Also, Verilog makes a **sign extension** for objects of different sizes that are declared as signed.

eg..,

```
assign temp = $signed(A);
```

```
initial
begin
  //A = 16'b1010_010000000000;
  A = 4'b1010;
```

A= 4 bit’s, but the temp = 9 bit’s is assigned $signed(A) observe sign bit ‘1’ is extended.

```
assign temp = (A);
```

⇒

```
initial
begin
  //A = 16'b1010_010000000000;
  A = 4'b1010;
```

A= 4 bit's, but the temp = 9 bit's is assigned (A), observe sign '1' is not extended when not used $signed.

Assignment-1

Implement Addition of any two numbers of $Q_{2.14}$ format

- $Q_{2.14} \rightarrow$ 2 bits dedicated to Integer part and remaining bits dedicated to fractional part
- Design should include overflow and underflow logic
- Verify the functionality of the design using testbench
- Check the Resource utilization for the same design and justify it.

Introduction:

To add two numbers in Q format, we should first align the binary point of the two numbers and sign extend the number that has shorter integer part

Functionality :

Addition in Q format:

eg:

N_x	+	N_y	=	N
$Q_{2.3}$		$Q_{1.4}$		$Q_{2.4}$

- First thing to be noticed is they are not in the same Q format.
- We can't add two numbers unless we match their size, even though both are 5 bits we can't simply add because they are not in the same format.

$Q_{n1.m1} + Q_{n2.m2} = Q_{n.m}$ n → maximum of n1 and n2, m → maximum of m1 and m2

- Rest of LSBs padded with zeros
- Rest of MSBs extended by MSB

Sign extension: (sign extending MSB does not change value)

Consider $\mathbf{N_y} = Q_{1.4} = 1\_1100$ convert it into $\mathbf{N} = Q_{2.4}$ => $Q_{2.4} = 11\_1100$

$Q_{2.2}(dec) = -1 + 0.5 + 0.25 = -0.25$
$Q_{3.2}(dec) = -2 + 1 + 0.5 + 0.25 = -0.25$

Aligning the binary point: (padding 0's at LSB does not change value)

Consider $\mathbf{N_x} = Q_{2.3} = 10\_111$ convert it into $\mathbf{N} = Q_{2.4}$ => $Q_{2.4} = 10\_1110$

$Q_{2.3}(dec) = -2 + 0.5 + 0.25 + 0.125 = -1.875$
$Q_{2.4}(dec) = -2 + 0.5 + 0.25 + 0.125 + 0 = -1.875$

Overflow:

In certain circumstances, when an adder/subtractor circuit is employing signed arithmetic, there is arithmetic overflow from the most significant magnitude bit into the sign bit. This will occur for example, if a 4-bit arithmetic result is required when two 3-bit numbers are added together and where the fourth bit in the circuit has been assigned the task of indicating the sign of the answer.

The consequences of overflow
when it occurs are:

1. The addition of two positive numbers gives a negative answer
2. The addition of two negative numbers gives a positive answer

An example of four possible situations that may arise is given below for a 4-bit word (n=4) and for each case the carries from the $(n-1)^{th}$ and n^{th} stages have been displayed.

$C_n = 0, C_{n-1} = 0$

+1	0,001	- -A
+3	0,011	- -B
+4	0,100	

$C_n = 0, C_{n-1} = 1$

+5	0,101	
+6	0,110	
+11	1,011	**interpreted as -5**

$C_n = 1, C_{n-1} = 1$

-3	1,101	
-3	1,101	
-6	1,010	

$C_n = 1, C_{n-1} = 0$

-5	1,011	
-6	1,010	
-11	0,101	**interpreted as +5**

Observe that when either a positive or negative sum of $(11)_{10}$ is required, the magnitude of this number, either in its positive or negative form, cannot be expressed in terms of three binary digits, and the resulting answer is both incorrect and has the wrong sign.

C_{n-1}	C_n	*O*
0	0	0
0	1	1
1	0	1
1	1	0

Truth table for the overflow function

A truth table for C_{n-1} and C_n, for the above four cases is shown and it is clear from this table that the overflow function is the XOR of C_n, C_{n-1}.

Hence the equation for the overflow flag is:

$$O = C_{n-1} \wedge C_n$$

A_3	B_3	S	*O*
0	0	0	0
0	0	1	1
1	1	0	0
1	1	0	1

Alternative expression, After K-map overflow flag in terms of A_3,B_3,S is

$$O = A_3'B_3'S + A_3B_3S'$$

where S is the sign of the result and A_3 and B_3 are the sign digits of the two 4-bit numbers.

Implementation:

```
assign temp_a = $signed(A); //for sign extending
assign temp_b = $signed(B); //for sign extending
```

temp_a, temp_b stores the signed extended versions of A,B that have short integer part.

```
    if(m1 > m2) begin
       temp_1 = temp_a;
        temp_2 = temp_b << shift;
      end
      else
       if (m1 < m2) begin
       temp_1 = temp_a << shift;
       temp_2 = temp_b;
```

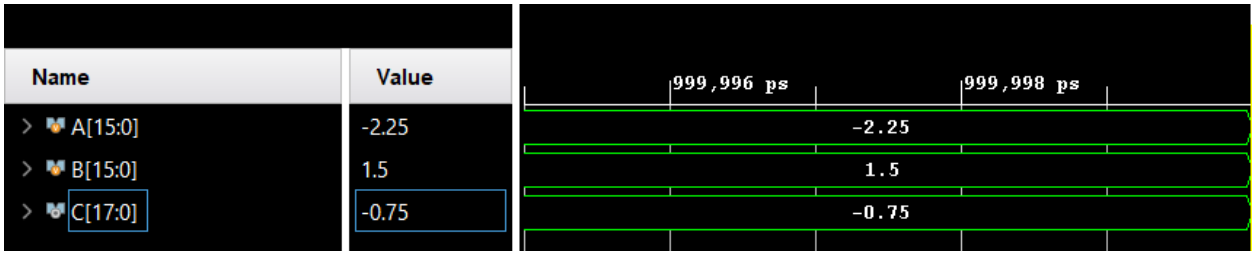
temp_1, temp_2 have the content of binary point aligned, based on short fractional part LSB are padded with 0's by a shift of difference in fractional length.

```
assign C = temp_1 + temp_2; //fixed point addition
 assign overflow = ~(temp_1[MAX_size-1])& ~(temp_2[MAX_size-1]) &
                    (C[MAX_size-1]) | (temp_1[MAX_size-1])&
                     (temp_2[MAX_size-1]) & ~(C[MAX_size-1]);
   //overflow function theory see doc
```

C has the content of the fixed point addition and the condition for the overflow flag was implemented based on the above derived expression.

Verification:

Functionality Test:

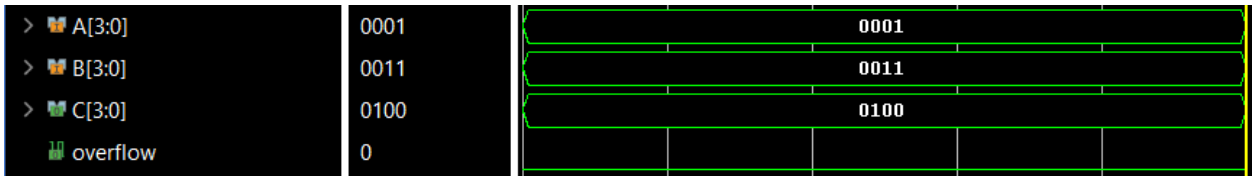


Different Q format Test: (with same input values)

Overflow Test: (aforementioned cases)

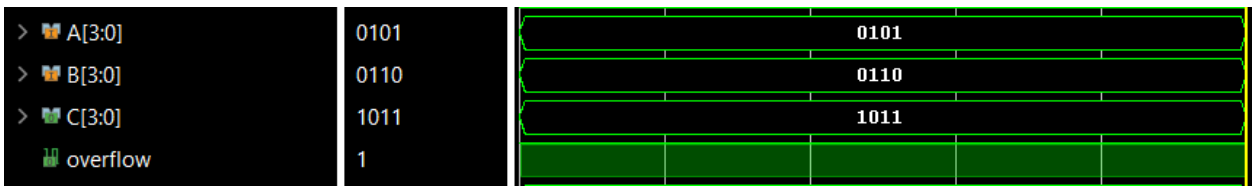
C_n = 0, C_{n-1} =0

+1	0,001
+3	0,011
+4	0,100



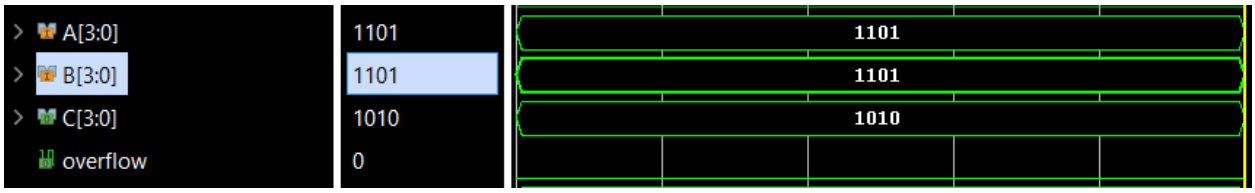
C_n = 0, C_{n-1} =1

+5	0,101	
+6	0,110	
+11	1,011	**interpreted as -5 (because of overflow)**



C_n = 1, C_{n-1} =1

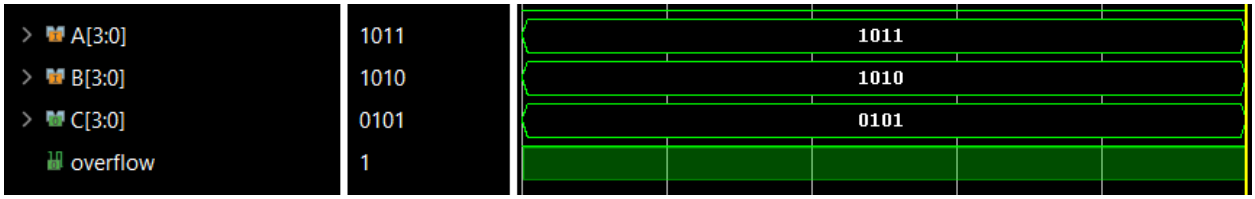
-5	1,011
-3	1,101
-11	0,101



C_n = 1, C_{n-1} =0

-5	1,011
-6	1,010
-11	0,101

interpreted as +5 (because of overflow)



Resources Utilization :

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!												17	0	0	0	0

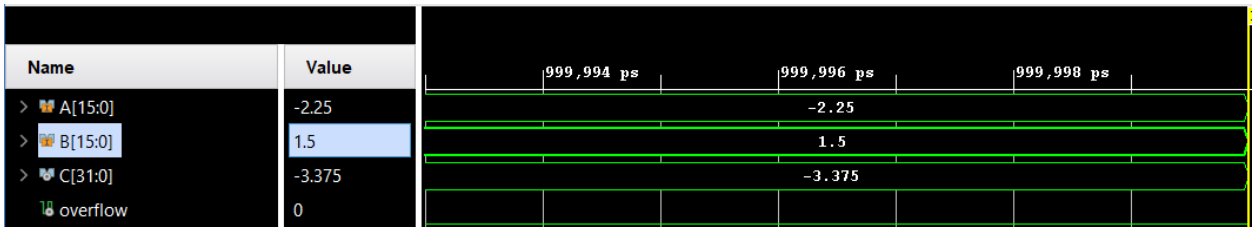
➔ https://github.com/velicharlagokulkumar/vivado/tree/main/fp_addition

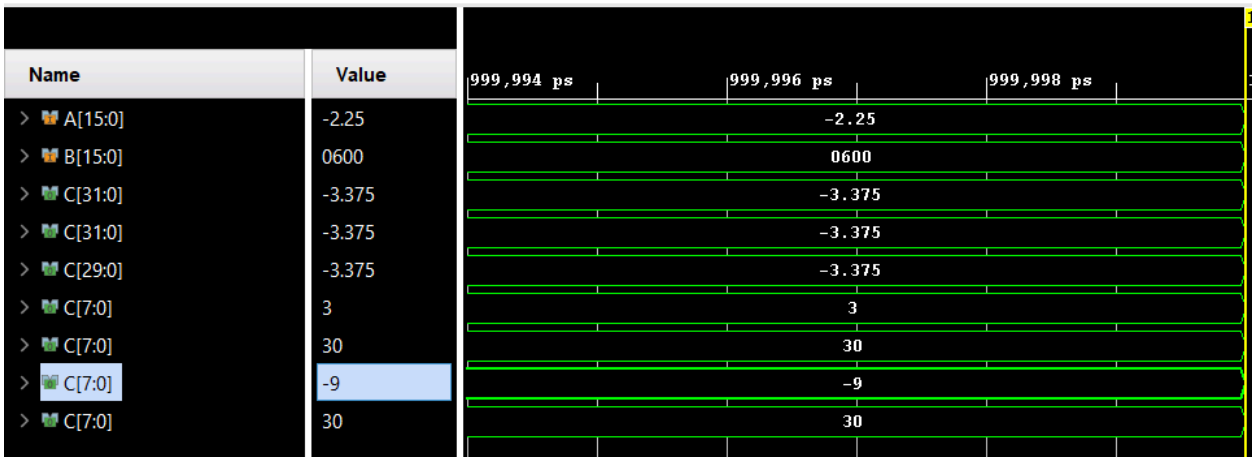
Assignment-2

Implement multiplication of any two numbers of Q2.14 format

- Q2.14 → 2 bits dedicated to Integer part and remaining bits dedicated to fractional part
- Design should include overflow and underflow logic
- Verify the functionality of the design using testbench
- Check the Resource utilization for the same design and justify it.

Verification:





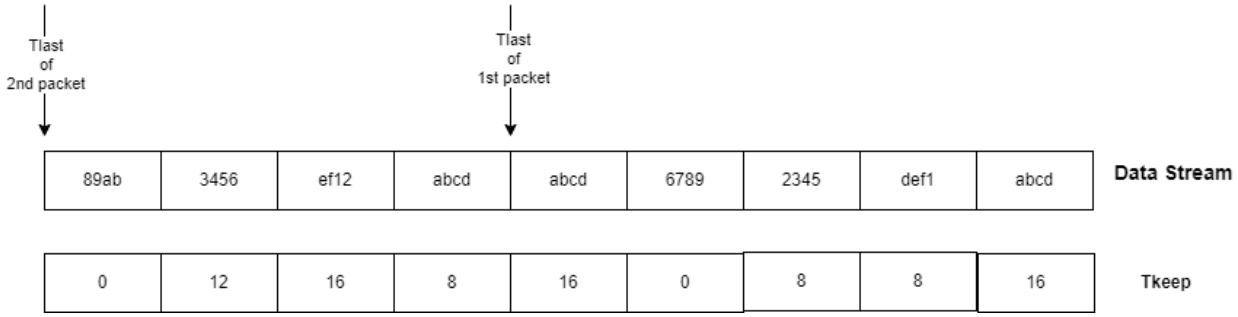
➔ https://github.com/velicharlagokulkumar/vivado/tree/main/fp_multiplication

Module 5: FSM Based RTL Designs

Assignment 1

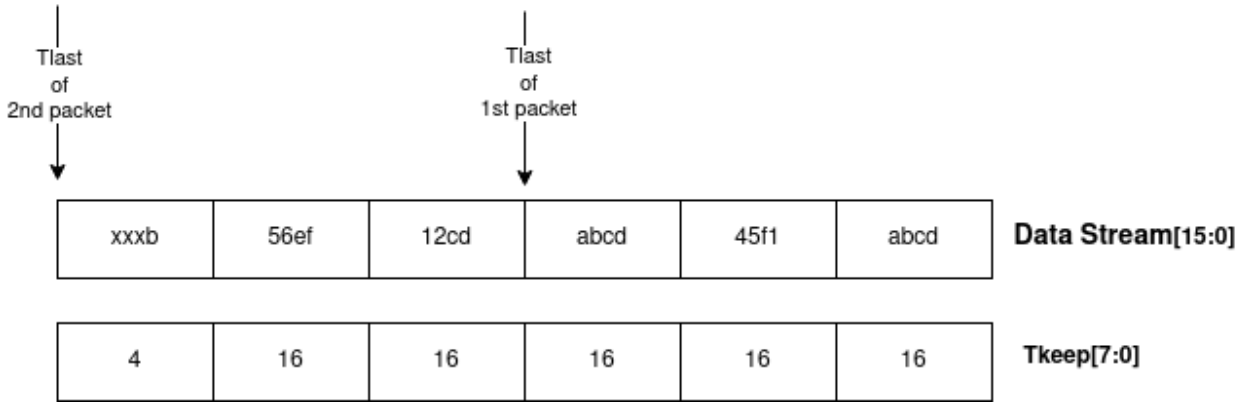
Functionality:

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

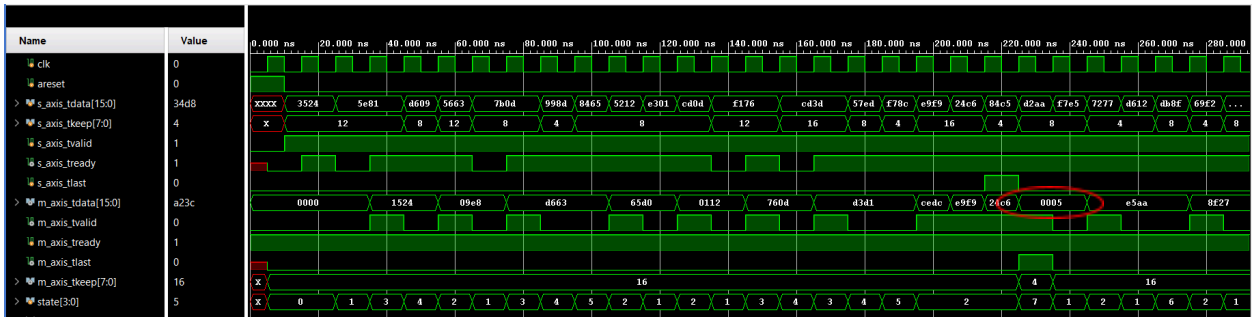


- Design should give 16 valid bits in every clock cycle exception may be valid for the last cycle in the packet based on the number of valid bits present in the packet.
- Each packet will end with the last signal asserted high and immediately next packet data will start coming.
- If the current clock cycle does not have 16 valid bits of data it will take enough bits from the succeeding samples to make 16 valid bits.

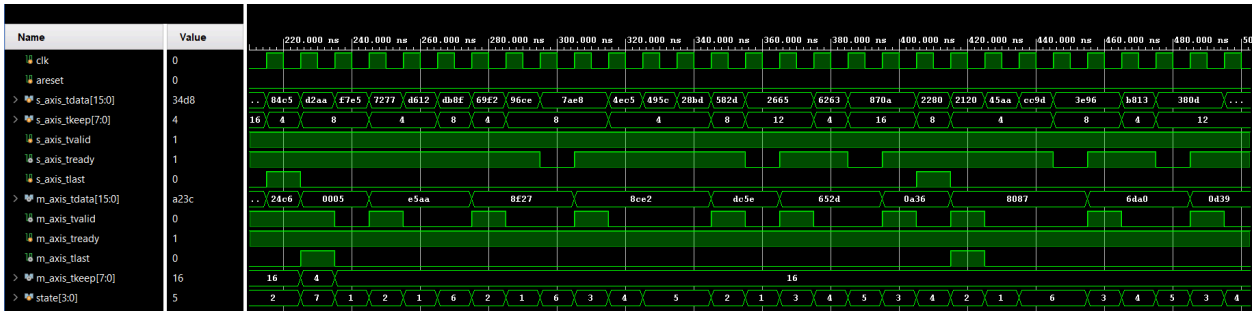
- Input : 16 bits and keep is of 8 bits
- output : 16 bits

Design should comply with AXI Stream protocol.(Use necessary signals only like, tdata, tvalid, tready and tkeep)

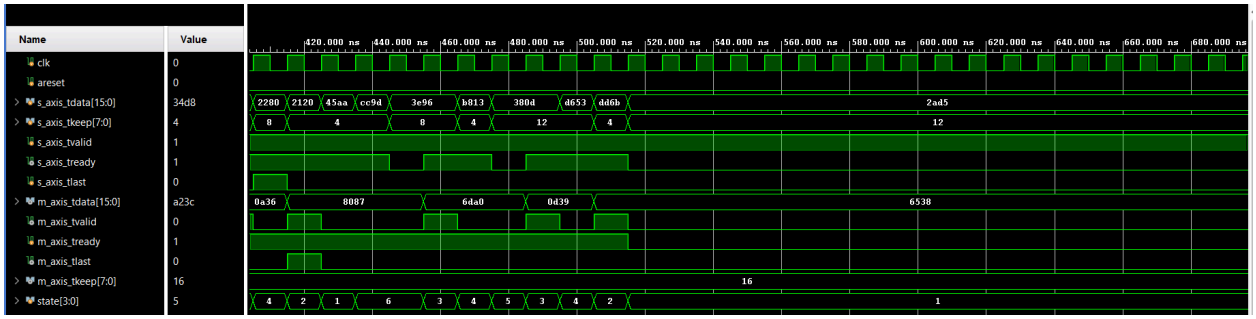
Verification:

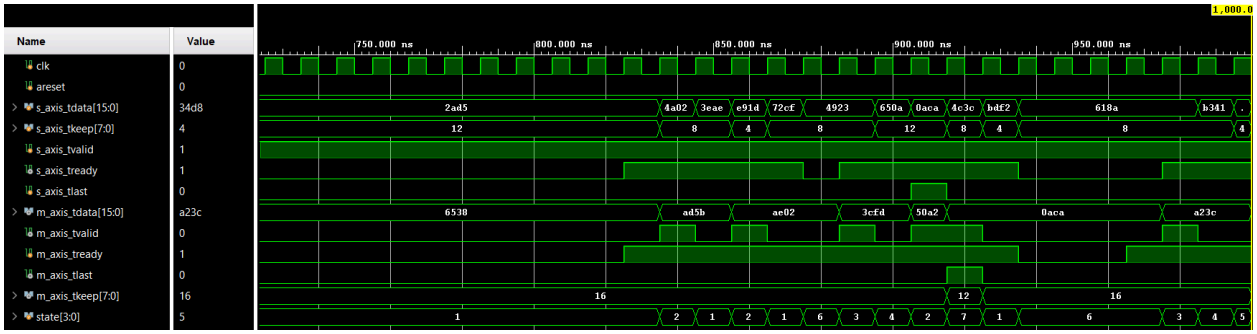


Tlast sample:



Handshake Test:

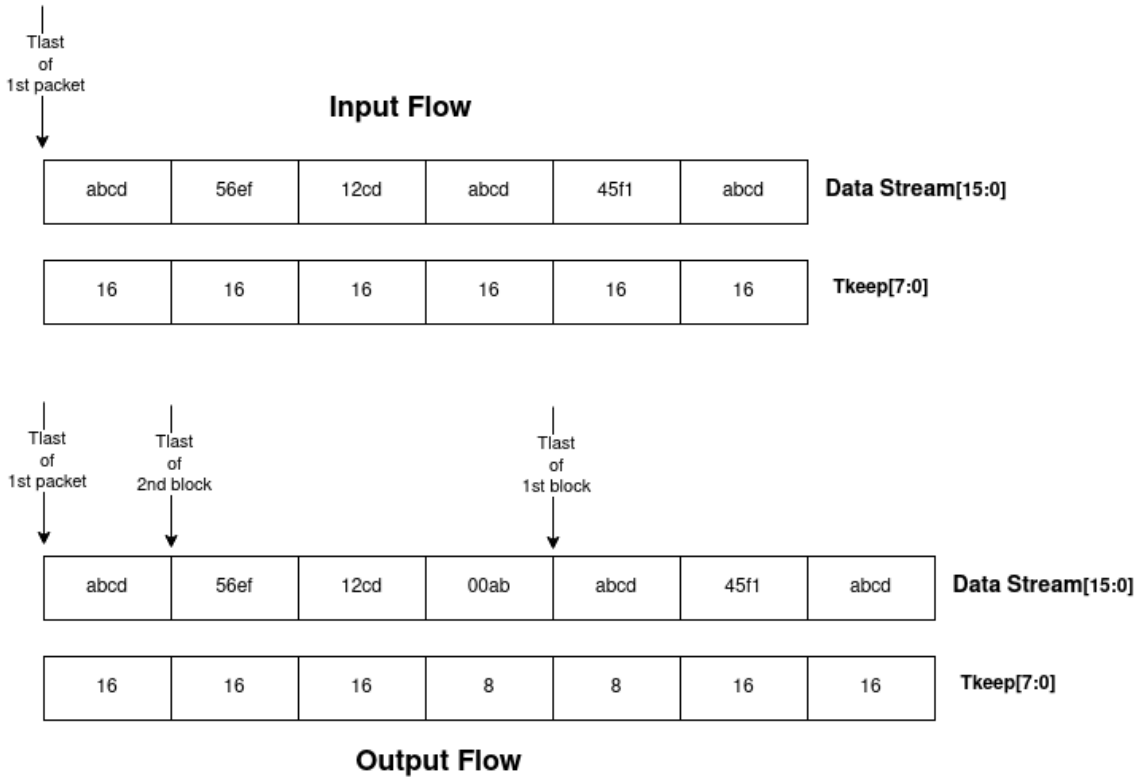




➔ https://github.com/velicharlagokulkumar/vivado/tree/main/fsm_1

Assignment 2

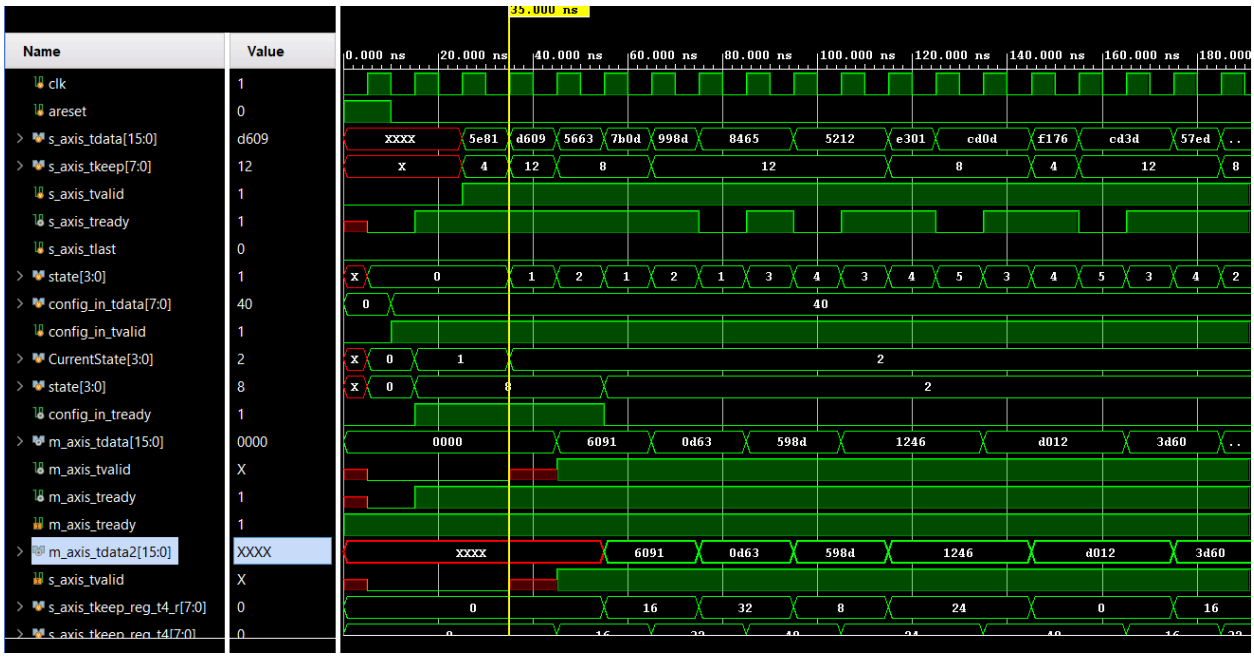
Functionality:



- 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

Verification:



➔ https://github.com/velicharlagokulkumar/vivado/tree/main/fsm_2