Implementation Report Analysis

Vivado

September 4, 2024

Chapter 1

Introduction

This document is an analysis of the implementation report generated by Vivado for the `top` module. The report is generated after the implementation process is completed. The report contains information about the resources used by the design, the timing constraints, and the timing performance of the design. The report is used to analyze the design and to identify any potential issues that may need to be addressed.

Chapter 2

Resource Utilization

The resource utilization section of the report provides information about the resources used by the design. This includes information about the number of LUTs, FFs, BRAMs, and DSPs used by the design. The resource utilization section also provides information about the number of IOBs used by the design.

2.1 Common Resource Types

Here are some common resource types used in FPGA designs and Verilog codes that utilizes these resources:

2.1.1 LUTs

LUTs (Look-Up Tables) are fundamental building blocks in FPGA designs. They are used to implement combinational logic functions. Each LUT has a fixed number of inputs and a single output. The Verilog code snippet below shows an example of a 2-input LUT implementation:

```
module lut2 (input wire a, b, output wire y);
    assign y = a & b;
endmodule
```

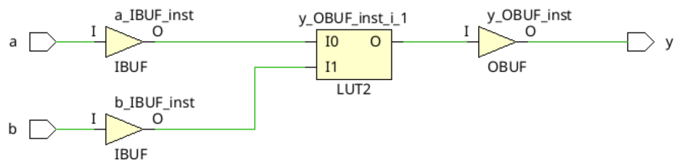


Figure 2.1.1.1: LUT Implemented

2.1.2 FFs

FFs (Flip-Flops) are sequential elements used to store and propagate data in FPGA designs. They are commonly used to implement registers and memory elements. The Verilog code snippet below shows an example of a D flip-flop implementation:

```
module dff (input wire clk, reset, input wire d, output reg q);
    always @(posedge clk or posedge reset)
        if (reset)
            q <= 1'b0;
        else
            q <= d;
endmodule
```

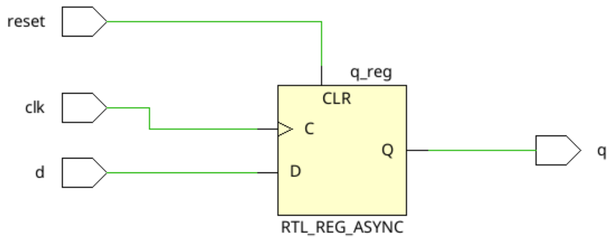


Figure 2.1.2.1: FF Synthesised

2.1.3 BRAMs

BRAMs (Block RAMs) are memory elements in FPGA designs. They provide large storage capacity and high-speed access. They can be coded into FPGA, or The Verilog code snippet below shows an example of a simple dual-port RAM implementation:

```
module dual_port_ram (
    input wire clk,
    input wire [7:0] addr_a, addr_b,
    input wire [7:0] data_a, data_b,
    input wire we_a, we_b,
    output reg [7:0] q_a, q_b
);
    reg [7:0] mem [0:255];
    always @(posedge clk) begin
        if (we_a)
            mem[addr_a] <= data_a;
        q_a <= mem[addr_a]; // Synchronous read for port A
    end
    always @(posedge clk) begin
```

```
        if (we_b)
            mem[addr_b] <= data_b;
        q_b <= mem[addr_b]; // Synchronous read for port B
    end
endmodule
```

* Vivado can pick BRAM, LUTRAM, or FFs to implement your design. If you want to force Vivado to use BRAM, you can use `(* ram_style = "block" *)` in the module. If your design fits Bram conditions, Vivado will use BRAMs to implement your design. If you do not explicitly specify, Vivado will use LUTRAMs or FFs to implement your design depending on size required access speed etc.

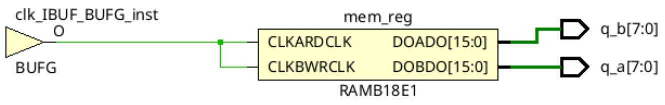


Figure 2.1.3.1: BRAM Implemented

2.1.4 DSPs

DSPs (Digital Signal Processors) are specialized hardware blocks in FPGA designs used for high-performance signal processing operations. They are commonly used in applications such as image and audio processing. The Verilog code snippet below shows an example of a simple multiplier using DSP blocks:

```
module multiplier (input wire [9:0] a, b, output wire [19:0] result);
    assign result = a * b;
endmodule
```

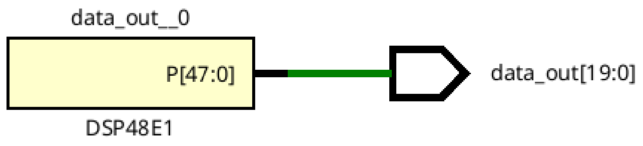


Figure 2.1.4.1: DSP Block Implemented

*For Basys3 Vivado prefers to use DSP blocks 10 and more bits multiplication. For 9 bits and less, it uses LUTs and FFs, but adding `(* use_dsp = "yes" *)` to the module can force Vivado to DSP for arithmetic operations.

2.1.5 IOBs

IOBs (Input/Output Blocks) are used to interface the FPGA design with external devices. They provide the necessary logic to drive and receive signals from the external world. The Verilog code snippet below shows an example of an IOB implementation for a simple GPIO (General Purpose Input/Output) interface:

```
module gpio (input wire clk, input wire reset, input wire [7:0] data_in,
             output wire [7:0] data_out, inout wire [7:0] io_pins);
    reg [7:0] internal_reg;

    always @(posedge clk or posedge reset)
        if (reset)
            internal_reg <= 8'b0;
        else
            internal_reg <= data_in;

    assign data_out = internal_reg;
    assign io_pins = internal_reg;
endmodule
```

2.2 Resource Utilization Report

The resource utilization report provides information about how many of these components are available in the FPGA and how many are used by your design. It helps you understand how efficiently your design is utilizing the available resources and whether any adjustments or optimizations are needed.

There are 8 parts in the resource utilization report.

1. Slice Logic: How many LUTs and FFs are used in the design. How many of LUTs are used as Logic or Memory and how many of FFs are used as Registers or Latches.
2. Slice Logic Distribution: How the LUTs and FFs are distributed across the slices.
3. Memory: Usage report of build in memory blocks. There can be different types for different FPGA families.
4. DSPs: Usage report of DSP blocks.
5. IOs
6. BUFG/BUFGCTRL (Clocking): Usage report of clocking resources.

7. Specific Feature: Usage report of specific featured blocks of the FPGA card.
8. Primitive: How many of each primitive type are utilized in your design. A more detailed report of the sources mentioned in the other tables used in the design. It doesn't give the utilization ratio.

Example scenarios to chose which part to look at:

- If you wanted to know exactly how many flip-flops or LUTs of a particular configuration (like FDRE for a flip-flop with enable) were used, you would consult the Primitive Types Report.
- If the design is using a lot of memory resources as LUT, you may want to look at the Memory section to see if there are any optimizations that can be made.
- If the design is using a lot of math operations, you may want to look at the DSP and LUT section to see if the design is close to the limits.
- *****The synthesis tool might not be able to utilize the design as desired. A different block of FPGA can be used to implement your design. For example a buffer can be utilized in BRAMs, LUTRAMs, or FFs. Simple changes in your design can make implementation tool to choose a different resource. *****

An example table from resource utilization report for the `top` module is shown below:

```
8. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| FDRE     |  178 |        Flop & Latch |
| LUT6     |   79 |                 LUT |
| LUT4     |   48 |                 LUT |
| LUT5     |   22 |                 LUT |
| LUT2     |   21 |                 LUT |
| LUT3     |   18 |                 LUT |
| CARRY4   |   11 |          CarryLogic |
| FDSE     |   10 |        Flop & Latch |
| OBUF     |    7 |                  IO |
| RAMB18E1 |    4 |        Block Memory |
| IBUF     |    3 |                  IO |
| LUT1     |    2 |                 LUT |
| BUFG     |    2 |               Clock |
+----------+------+---------------------+
```

General file names for this report are `Module_Name_utilization_routed.rpt` or `Module_Name_utilization_placed.rpt`.

2.2.1 TCL Commands

The most useful setting for utilization analysis when calling TCL option is hierarchical. This option gives a single table report, but with primitive types used by all submodules and theirs. This is useful to see which module is using the most of a specific resource. You can set this setting in the TCL command as follows:

```
report_utilization -hierarchical
```

*This command can help you to see which module is using the most of a specific resource. If you see a module is using a lot of a specific resource, you can optimize that module to reduce the resource usage.

Chapter 3

Power Analysis

Power analysis is an important aspect of FPGA design as it helps in understanding the power consumption of the implemented design. By analyzing the power utilization, we can optimize the design to reduce power consumption and improve overall efficiency. Power analysis provides insights into the power consumption of different components in the design, such as LUTs, FFs, BRAMs, DSPs, and IOBs. It also helps in identifying any potential power-related issues that may need to be addressed. In this section, we will explore the power utilization of the `top` module and analyze the power consumption of various components in the design.

3.1 Power Utilization Report

The power utilization report provides information about the power consumption of different components in the design. It includes details about the dynamic power, static power, and total power consumption of the design. There are 3 parts in the power utilization report. The content of the report may vary depending on the FPGA family and the tool used.

However, The first two table 1.1 *and* 1.2 **and the last table** 3.1 **are the most useful table for power analysis.**

- 1.1 Power Summary: This table provides an overview of the power consumption of the design. It includes information about the dynamic power, static power, dynamic power, and total power consumption of the design. It also provides details about constraints (if you set).
- 1.2 Power by Component: This table provides detailed information about the power consumption of different components in the design. It includes information about the power consumption of LUTs, FFs, BRAMs, DSPs, and IOBs. It helps in understanding the power consumption of different components and optimizing the design to reduce power consumption.
- 3.1 Power by Hierarchy: This table provides a hierarchical view of the power consumption of the design. It includes information about the power consumption of different modules in the design and helps in identifying the power consumption of individual modules.

3.1.1 Power Summary (1.1)

An example table from power utilization report for the `top` module is shown below:

```
    +-------------------------+--------------+
| Total On-Chip Power (W)  | 0.082        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 0.010        |
| Device Static (W)        | 0.072        |
| Effective TJA (C/W)      | 5.0          |
| Max Ambient (C)          | 84.6         |
| Junction Temperature (C) | 25.4         |
| Confidence Level         | Medium       |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
```

An explanation of the fields and tips about some fields in the table is as follows:

- Total On-Chip Power (W): The total power consumption of the design on the FPGA chip.
- Design Power Budget (W): The specified power budget for the design. **If you set a power budget, you can compare the actual power consumption with the specified power budget to ensure that the design meets the power requirements. If you need to run on battery power, this might be important.**
- Power Budget Margin (W): The margin between the actual power consumption and the specified power budget.
- Dynamic (W): The dynamic power consumption of the design. **High dynamic power drawn when performance required. It is best to keep power drawn low when not required. For example, if your design uses multiple modules used as needed, you can disable their clocks when they are not used. This should reduce dynamic power.**
- Device Static (W): The static power consumption of the FPGA device. **This should be reduced to keep power consumption and temperature low. Power gating can be used to reduce static power. This is not available for all FPGAs please research.**
- Effective TJA (C/W): The thermal junction-to-ambient resistance of the FPGA device.
- Max Ambient (C): The maximum ambient temperature for the design.
- Junction Temperature (C): The junction temperature of the FPGA device.
- Confidence Level: The confidence level of the power analysis results. This depends on the user specifications. **If you want a more accurate result, you can increase the confidence level by setting IO and clock activity using GUI or constraints file.**

- Setting File: The settings file used for the power analysis.
- Simulation Activity File: The simulation activity file used for the power analysis.
- Design Nets Matched: The number of design nets matched during the power analysis.

3.1.2 Power by Component (1.2)

An example table from power utilization report for the `top` module is shown below:

```
+----------------+-----------+----------+-----------+-----------------+
| On-Chip        | Power (W) | Used     | Available | Utilization (%) |
+----------------+-----------+----------+-----------+-----------------+
| Clocks         |     0.001 |        5 |       --- |             --- |
| Slice Logic    |    <0.001 |      406 |       --- |             --- |
|   LUT as Logic |    <0.001 |      164 |     20800 |            0.79 |
|   Register     |    <0.001 |      188 |     41600 |            0.45 |
|   CARRY4       |    <0.001 |       11 |      8150 |            0.13 |
|   Others       |     0.000 |       17 |       --- |             --- |
| Signals        |    <0.001 |      363 |       --- |             --- |
| Block RAM      |    <0.001 |        2 |        50 |            4.00 |
| I/O            |     0.007 |       10 |       106 |            9.43 |
| Static Power   |     0.072 |          |           |                 |
| Total          |     0.082 |          |           |                 |
+----------------+-----------+----------+-----------+-----------------+
```

Using this table and information from the previous section, you can analyze the power consumption of different components in the design and identify areas where power optimization is needed. For example, if the design is using a large number of LUTs, you can optimize the design to reduce the number of LUTs used and reduce power consumption. Similarly, if the design is using a large number of a component, and if it is more efficient to use other blocks you can force using other blocks for your design.

3.1.3 Power by Hierarchy (3.1)

An example table from power utilization report for the `top` module is shown below:

```
+---------------+-----------+
| Name          | Power (W) |
+---------------+-----------+
| MemController |     0.010 |
+---------------+-----------+
```

This table is not a great example, but if yours include your submodules you can analyze the power consumption of different modules in the design and identify areas where power optimization is needed. For example, if a particular module is consuming a large amount of power, you can optimize that module to reduce power consumption. This because of the hierarchical depth setting. At next session you will see how to set this setting.

3.1.4 TCL Commands

The most useful setting for power analysis when calling TCL option is hierarchical_depth. When implementation run it usually set to 1 by default. However, if you set it to 0, you can see the power consumption of the submodules and their submodules until the least piece. This is useful to see which module is consuming the most power, or you can set the depth you desire You can set this setting in the TCL command as follows:

```
report_power -hierarchical_depth 0
```

Chapter 4

Methodology Warnings

The methodology warnings section of the report provides information about any potential issues or warnings that may need to be addressed. This section includes information about the timing constraints, clock constraints, and other design constraints that may impact the performance of the design. The methodology warnings section helps in identifying any potential issues that may need to be addressed to improve the performance of the design.

4.1 Methodology Warnings Report

Example issues from methodology warnings report for the `top` module is shown below:

```
[Timing 38-282] The design failed to meet the timing requirements.
Please see the timing summary report for details on the timing violations.

[Place 30-58] IO placement is not user specified.

TIMING-18#1 Warning
Missing input or output delay
An input delay is missing on reset relative to the rising and/or falling clock edge(s)
of sys_clk_pin.
Related violations: <none>
```

Some of these issues are about the constraints that you did not set. Some can be ignored if you are confident about timing latency of those pins. However, if you are not sure about the latency of the pins, you can set the constraints for those pins. Then, if you get timing failed warning, your system will be high probably not working as desired stably. IO placement is more important, there might be bitstream write issues even if it is implemented. Also, if the tool can place as it wants, the design may not function as desired on the device. You can check the IO placement in `Module_Name_io_placed.rpt` file.

Chapter 5

Timing Analysis

The timing analysis section of the report provides information about the timing performance of the design. This includes information about the clock constraints, setup and hold times, clock-to-q delays, and other timing parameters that impact the performance of the design. The timing analysis section helps in identifying any potential timing issues that may need to be addressed to improve the performance of the design. This is the most complex and important part of the report.

5.1 Definitions and Explanations

These are some common terms used in the timing analysis report in order to understand the report better:

- **Setup Time:** The amount of time the input signal must be stable before the active edge of the clock signal. If the input signal changes too close to the active edge of the clock signal, the flip-flop may not capture the correct value.
- **Hold Time:** The amount of time the input signal must be stable after the active edge of the clock signal. If the input signal changes too soon after the active edge of the clock signal, the flip-flop may not capture the correct value.
- **Slack:** The slack is the amount of time by which the actual timing of the design exceeds the required timing. A positive slack indicates that the design meets the timing requirements, while a negative slack indicates that the design fails to meet the timing requirements.
- **Clock-to-Q Delay:** The amount of time it takes for the output signal to change after the active edge of the clock signal. This is the delay through the flip-flop.
- **Clock Skew:** The difference in arrival times of the clock signal at different flip-flops. Clock skew can cause timing violations if the clock signal arrives at different flip-flops at different times.
- **Clock Uncertainty:** The amount of uncertainty in the clock signal arrival time. Clock uncertainty can cause timing violations if the clock signal arrives at different flip-flops at different times.

- **Critical Path:** The path in the design with the longest delay. The critical path determines the maximum clock frequency of the design.
- **Timing Violation:** A timing violation occurs when the design fails to meet the timing requirements specified in the constraints file. Timing violations can cause the design to malfunction or fail to meet the desired performance.
- **Clock Domain Crossing:** Clock domain crossing occurs when signals from one clock domain are transferred to another clock domain. Clock domain crossing can cause timing violations if the signals are not synchronized properly.
- **False Path:** A false path is a path in the design that is not required to meet the timing requirements. False paths are ignored by the timing analysis tool to reduce the analysis time.
- **TNS:** Total Negative Slack. The sum of all negative slack values in the design.
- **WNS:** Worst Negative Slack. The worst negative slack value in the design.
- **THS:** Total Hold Slack. The sum of all hold slack values in the design.
- **WHS:** Worst Hold Slack. The worst hold slack value in the design.
- **WPWS:** Worst Pulse Width Slack. The worst pulse width slack value in the design.
- **Data Path Delay:** The amount of time it takes for the data signal to propagate through the combinational logic in the design. Data path delay is a critical parameter that impacts the overall performance of the design.
- **Clock Jitter:** The amount of variation in the clock signal arrival time. System jitter can cause timing violations if the clock signal arrives at different flip-flops at different times.

5.2 Calculation of Slack

The setup slack calculation as follows:

$$\text{Time Required} = T_r = \text{Maximum time that the signal must arrive at the destination before FF setup}$$

$$\text{Arrival Time} = T_a = \text{Time that the signal actually arrives at the destination FF}$$

$$\text{Clock Period Difference} = T_p = \text{Time difference between the triggering clock edges of the source and destination FFs}$$

$$\text{Clock Path Delay} = T_{ds}, T_{dd} = \text{Delay of the clock signal to source and destination FF}$$

$$\text{Data Path Delay} = T_{dp} = \text{Delay of the data signal to the destination FF}$$

$$\text{Setup Time} = T_s = \text{Setup time of the destination FF}$$

$$T_r = T_{dd} + T_p - T_s \text{ (T_p is for current and next clock cycle, period, in the same clock domain)}$$

$$T_a = T_{ds} + T_{dp} \text{ (Source FF Clock to Q delay included)}$$

$$\text{Setup Slack} = T_r - T_a$$

The hold slack calculation as follows:

Time Required $= T_r =$ Minimum time that the signal must arrive at the destination after FF setup

Arrival Time $= T_a =$ Time that the signal actually arrives at the destination FF

Clock Period Difference $= T_p =$ Time difference between the triggering clock edges of the source and destination FFs

Clock Path Delay $= T_{ds}, T_{dd} =$ Delay of the clock signal to source and destination FF

Data Path Delay $= T_{dp} =$ Delay of the data signal to the destination FF

Hold Time $= T_h =$ Hold time of the destination FF

$$T_r = T_{dd} + T_p + T_h \text{ (}T_p\text{ is 0 in the same clock domain)}$$

$$T_a = T_{ds} + T_{dp} \text{ (Source FF Clock to Q delay included)}$$

$$\text{Hold Slack} = T_a - T_r$$

Negative values of slacks are the timing violations.

5.3 Timing Analysis Report

The timing analysis doesn't have explicit sections like the other reports. It is a long report that includes all the required timing information of the design. However, we can break it down into 4 parts:

5.3.1 Timing Checks

At the start of the timing analysis report, there is a summary of the timing checks. These checks make sure that an accurate timing report is produced. The summary provides timing warnings for your design. This suggestions might be useful to fix the timing issues. An example from the timing analysis report for the `top` module is shown below:

```
Table of Contents
-----------------
1. checking no_clock (0)
2. checking constant_clock (0)
3. checking pulse_width_clock (0)
4. checking unconstrained_internal_endpoints (0)
5. checking no_input_delay (2)
6. checking no_output_delay (6)
7. checking multiple_clock (0)
8. checking generated_clocks (0)
9. checking loops (0)
10. checking partial_input_delay (0)
11. checking partial_output_delay (0)
12. checking latch_loops (0)

1. checking no_clock (0)
```

```
------------------------
 There are 0 register/latch pins with no clock.


2. checking constant_clock (0)
------------------------------
 There are 0 register/latch pins with constant_clock.

...

5. checking no_input_delay (2)
------------------------------
 There are 2 input ports with no input delay specified. (HIGH)

 There are 0 input ports with no input delay but user has a false path constraint.


6. checking no_output_delay (6)
-------------------------------
 There are 6 ports with no output delay specified. (HIGH)

 There are 0 ports with no output delay but user has a false path constraint

 There are 0 ports with no output delay but with a timing clock defined on it
 or propagating through it


7. checking multiple_clock (0)
------------------------------
 There are 0 register/latch pins with multiple clocks.


...


12. checking latch_loops (0)
----------------------------
 There are 0 combinational latch loops in the design through latch input
```

Using this information, you can identify any potential timing issues in the design and take appropriate actions to address them. For example, if there are input ports with no input delay specified, you can set the input delay constraints to ensure that the design meets the timing requirements. If the design has a negative slack, the input delay that is not specified might be causing this result. These warnings effect the detailed design timing summary which comes after.

5.3.2 Design Timing Summary

After the checks, there is a timing report of the whole module. An example from the timing analysis report for this part is shown below:

```
------------------------------------------------------------------------------------
| Design Timing Summary                                                            |
| ---------------------                                                            |
------------------------------------------------------------------------------------

WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)
-------      -------  ---------------------  -------------------      -------
3.864        0.000                        0                  561        0.058

THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)     TPWS(ns)
-------  ---------------------  -------------------     --------     --------
0.000                        0                  561       4.500        0.000

TPWS Failing Endpoints  TPWS Total Endpoints
----------------------  --------------------
                     0                   198
```

The titles are already explained in the previous part. If there is no negative value or close to zero, the design is working as desired. If there is a negative value, you should check the detailed timing report to see which part of the design is causing the issue. Hold times are close to zero, but they are how they are supposed to be as hold time is already a small value.

5.3.3 Clock Timing Summary

After the design timing summary, there is a clock timing summary. This part provides information about the clock constraints, slacks for each clock's own domain, slacks between clock domains, and clock uncertainty of the design. It helps in understanding the clock performance of the design and identifying any potential clock-related issues that may need to be addressed. An example from the timing analysis report for this part is shown below:

```
--------------------------------------------------------------------------------------------------
| Clock Summary
| -------------
--------------------------------------------------------------------------------------------------

Clock        Waveform(ns)          Period(ns)      Frequency(MHz)
-----        ------------          ----------      --------------
sys_clk_pin  {0.000 5.000}         10.000          100.000
    rx_clk     {0.000 270.000}       540.000         1.852
    tx_clk     {0.000 4340.000}      8680.000        0.115


--------------------------------------------------------------------------------------------------
```

```
| Intra Clock Table
| -----------------
-----------------------------------------------------------------------------------------------

Clock             WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints
-----             -------      -------  ---------------------  -------------------
sys_clk_pin         4.227        0.000                      0                  445
rx_clk            535.110        0.000                      0                   74
tx_clk           8675.541        0.000                      0                   34

WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints     WPWS(ns)
-------      -------  ---------------------  -------------------     --------
0.058        0.000                      0                  445        4.500
0.177        0.000                      0                   74      269.500
0.191        0.000                      0                   34     4339.500

TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
--------  ----------------------  --------------------
0.000                      0                   140
0.000                      0                    40
0.000                      0                    18
-------------------------------------------------------------
| Inter Clock Table                                         |
| -----------------                                         |
-------------------------------------------------------------

From Clock    To Clock          WNS(ns)      TNS(ns)  TNS Failing Endpoints
----------    --------          -------      -------  ---------------------
rx_clk        sys_clk_pin         3.864        0.000                      0
tx_clk        sys_clk_pin         5.181        0.000                      0
sys_clk_pin   tx_clk              6.771        0.000                      0

TNS Total Endpoints      WHS(ns)      THS(ns)  THS Failing Endpoints
-------------------      -------      -------  ---------------------
                69        0.945        0.000                      0
                 5        0.747        0.000                      0
                25        0.131        0.000                      0

                    THS Total Endpoints
                    -------------------
                                    69
                                     5
                                    25
```

The first table in this part gives wave specifications of the clocks. The second table gives the slack values of the clocks. The third table gives the slack values between the clocks. If there is a negative value in the slack, you should check the detailed timing report to see which part of the design is causing the issue.

As you can realize slower clocks has no problem with timing as they have a lot of slack. However, the faster clock has a tight timing. This is a common issue in the designs.

Inter Clock table is formed by the clocks that are crossing each other. If two systems with different clocks are not connected at all, they are not in this table. If they are connected in a way (output/input), it is in this table. From this report, we can understand that the module with main clock both reads and writes to the module with tx_clk in always @(clk) block.

When a design slack problem is detected, the first thing to do is to check the detailed timing report to see which part or module interaction of the design is causing the issue.

5.3.4 Detailed Timing Report

For each clock domain and cross domain, for each path the timing report is placed here ordered slack worst to best. All component's delay on the path can be seen with input and clock jitter calculation available. An example part from the timing analysis report for this part is shown below (For single path):

```
Slack (MET) :             4.227ns  (required time - arrival time)
Source:                  time_out_counter_reg[12]/C
                         (rising edge-triggered cell FDRE clocked by sys_clk_pin  {rise@0.000ns fall@5.000ns pe
Destination:             time_out_counter_reg[24]/R
                         (rising edge-triggered cell FDRE clocked by sys_clk_pin  {rise@0.000ns fall@5.000ns pe
Path Group:              sys_clk_pin
Path Type:               Setup (Max at Slow Process Corner)
Requirement:             10.000ns  (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay:         5.188ns  (logic 1.014ns (19.546%)  route 4.174ns (80.453%))
Logic Levels:            4  (LUT2=1 LUT3=1 LUT4=1 LUT6=1)
Clock Path Skew:         -0.026ns (DCD - SCD + CPR)
Destination Clock Delay (DCD):    4.857ns = ( 14.857 - 10.000 )
Source Clock Delay      (SCD):    5.156ns
Clock Pessimism Removal (CPR):    0.273ns
Clock Uncertainty:       0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter     (TSJ):    0.071ns
Total Input Jitter      (TIJ):    0.000ns
Discrete Jitter          (DJ):    0.000ns
Phase Error              (PE):    0.000ns

Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
-------------------------------------------------------------------    -------------------
                         (clock sys_clk_pin rise edge)
                                                    0.000     0.000 r
W5                                                0.000     0.000 r  clk (IN)
                         net (fo=0)                   0.000     0.000    clk
W5                   IBUF (Prop_ibuf_I_O)         1.458     1.458 r  clk_IBUF_inst/O
                         net (fo=1, routed)           1.967     3.425    clk_IBUF
BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)         0.096     3.521 r  clk_IBUF_BUFG_inst/O
```

```
                         net (fo=139, routed)         1.635     5.156    clk_IBUF_BUFG
SLICE_X6Y4           FDRE                                           r  time_out_counter_reg[12]/C
-------------------------------------------------------------------    -------------------
SLICE_X6Y4           FDRE (Prop_fdre_C_Q)         0.518     5.674 f  time_out_counter_reg[12]/Q
                         net (fo=2, routed)           0.653     6.328    nolabel_line102/time_out_counter[0]_i_
SLICE_X7Y5           LUT4 (Prop_lut4_I0_O)        0.124     6.452 f  nolabel_line102/time_out_counter[0]_i_6/O
                         net (fo=1, routed)           1.154     7.605    nolabel_line102/time_out_counter[0]_i_
SLICE_X7Y5           LUT6 (Prop_lut6_I3_O)        0.124     7.729 f  nolabel_line102/time_out_counter[0]_i_2/O
                         net (fo=3, routed)           0.597     8.327    nolabel_line102/time_out_counter[0]_i_
SLICE_X9Y5           LUT2 (Prop_lut2_I0_O)        0.124     8.451 f  nolabel_line102/FSM_sequential_state[2]_i
                         net (fo=4, routed)           0.898     9.349    nolabel_line102/time_out_counter_reg[0
SLICE_X7Y4           LUT3 (Prop_lut3_I2_O)        0.124     9.473 r  nolabel_line102/time_out_counter[26]_i_1/
                         net (fo=26, routed)          0.871    10.344    nolabel_line102_n_21
SLICE_X6Y7           FDRE                                           r  time_out_counter_reg[24]/R
-------------------------------------------------------------------    -------------------

                         (clock sys_clk_pin rise edge)
                                                     10.000    10.000 r
W5                                                    0.000    10.000 r  clk (IN)
                         net (fo=0)                   0.000    10.000    clk
W5                   IBUF (Prop_ibuf_I_O)         1.388    11.388 r  clk_IBUF_inst/O
                         net (fo=1, routed)           1.862    13.250    clk_IBUF
BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)         0.091    13.341 r  clk_IBUF_BUFG_inst/O
                         net (fo=139, routed)         1.516    14.857    clk_IBUF_BUFG
SLICE_X6Y7           FDRE                                           r  time_out_counter_reg[24]/C
                         clock pessimism              0.273    15.130
                         clock uncertainty           -0.035    15.095
SLICE_X6Y7           FDRE (Setup_fdre_C_R)       -0.524    14.571    time_out_counter_reg[24]
-------------------------------------------------------------------
                         required time                         14.571
                         arrival time                         -10.344
-------------------------------------------------------------------
                         slack                                  4.227
```

Let's break down the information in the report:

This is a setup timing path. The slack calculated is 4.227ns. This means that the path is 4.227ns faster than the required time. This is a good slack value. The source and destination of the path are given. The source is the register named `time_out_counter_reg[12]` that is sending the signal, and the destination is the register named `time_out_counter_reg[24]` that is receiving the signal. There are calculation logic between these two registers. Now lets look at times:

- Event: Clock rose at 0.000ns
- Clock signal reached the source register at 5.156ns (Source Clock Delay), starting logic flow.
- Then the signal of source register is processed by the logic between and reached the destination register at 10.344ns. (Data Path Delay: 10.344ns - 5.156ns = 5.188ns)
- The data already ready for destination to setup.

- Event: Clock rose at 10.000ns again
- Clock signal reached the destination register at 14.857 (Destination Clock Delay: 14.857ns - 10ns = 4.857ns), starting setup time.
- The setup is ready at 14.571ns, with calculation of clock pessimism and uncertainty. (Slack: 14.571ns - 10.344ns = 4.227ns)

*Tip: When you see a borderline (seperator) after a row in the path timings, a key event happened which are in the graph below.

To sum up, time required is when the second rise of the clock signal is reaches the destination register. Time arrival is the exact time data gets to D pin. Data path delay is Q to D. The clock reached at different times to the two registers. This is a clock skew example.

A graphical representation of the path as follows:

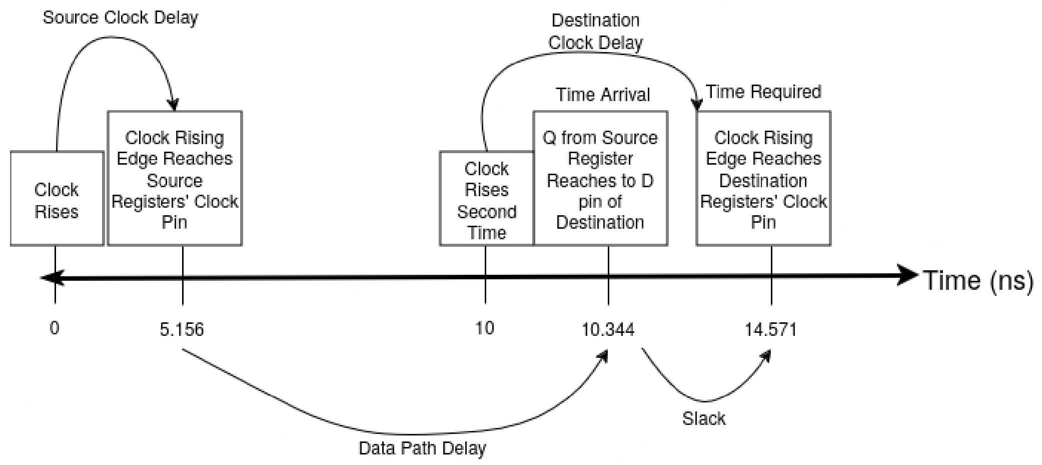

Source Clock Delay
Destination Clock Delay
Time Arrival
Time Required
Clock Rises
Clock Rising Edge Reaches Source Registers' Clock Pin
Clock Rises Second Time
Q from Source Register Reaches to D pin of Destination
Clock Rising Edge Reaches Destination Registers' Clock Pin
Time (ns)
0
5.156
10
10.344
14.571
Slack
Data Path Delay

Figure 5.3.4.1: Setup Timing Path

When it is a hold slack timing path, the time that data reaches the destination register is calculated. Then the hold time is calculated by the time that data is stable after the clock edge. Unlike it is given as two flow starting at t = 0ns, one is data path, the other flow includes parts inside the register, which is to find hold time. An example hold slack timing path is shown below:

```
Min Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :               0.177ns  (arrival time - required time)
Source:                   nolabel_line85/data_reg_reg[1]/C
                          (rising edge-triggered cell FDSE clocked by rx_clk  {rise@0.000ns fall@270.000ns perio
Destination:              nolabel_line85/data_out_reg[1]/D
                          (rising edge-triggered cell FDRE clocked by rx_clk  {rise@0.000ns fall@270.000ns perio
Path Group:               rx_clk
Path Type:                Hold (Min at Fast Process Corner)
```

```
Requirement:            0.000ns  (rx_clk rise@0.000ns - rx_clk rise@0.000ns)
Data Path Delay:        0.266ns  (logic 0.141ns (52.911%)  route 0.125ns (47.089%))
Logic Levels:           0
Clock Path Skew:        0.013ns (DCD - SCD - CPR)
Destination Clock Delay (DCD):    3.331ns
Source Clock Delay      (SCD):    2.476ns
Clock Pessimism Removal (CPR):    0.842ns

Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
-------------------------------------------------------------------    -------------------
                     (clock rx_clk rise edge)     0.000     0.000 r
W5                                                0.000     0.000 r  clk (IN)
                     net (fo=0)                   0.000     0.000    clk
W5                   IBUF (Prop_ibuf_I_O)         0.226     0.226 r  clk_IBUF_inst/O
                     net (fo=1, routed)           0.631     0.858    clk_IBUF
BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)         0.026     0.884 r  clk_IBUF_BUFG_inst/O
                     net (fo=139, routed)         0.562     1.445    clk_div/clk_IBUF_BUFG
SLICE_X35Y45         FDRE (Prop_fdre_C_Q)         0.141     1.586 r  clk_div/clk_rx_uart_out_reg/Q
                     net (fo=2, routed)           0.268     1.854    rx_clk
BUFGCTRL_X0Y1        BUFG (Prop_bufg_I_O)         0.026     1.880 r  rx_clk_BUFG_inst/O
                     net (fo=39, routed)          0.596     2.476    nolabel_line85/rx_clk_BUFG
SLICE_X3Y1           FDSE                                         r  nolabel_line85/data_reg_reg[1]/C
-------------------------------------------------------------------    -------------------
SLICE_X3Y1           FDSE (Prop_fdse_C_Q)         0.141     2.617 r  nolabel_line85/data_reg_reg[1]/Q
                     net (fo=2, routed)           0.125     2.742    nolabel_line85/data_reg_reg_n_0_[1]
SLICE_X2Y1           FDRE                                         r  nolabel_line85/data_out_reg[1]/D
-------------------------------------------------------------------    -------------------

                    (clock rx_clk rise edge)     0.000     0.000 r
W5                                               0.000     0.000 r  clk (IN)
                     net (fo=0)                   0.000     0.000    clk
W5                   IBUF (Prop_ibuf_I_O)         0.414     0.414 r  clk_IBUF_inst/O
                     net (fo=1, routed)           0.685     1.099    clk_IBUF
BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)         0.029     1.128 r  clk_IBUF_BUFG_inst/O
                     net (fo=139, routed)         0.831     1.958    clk_div/clk_IBUF_BUFG
SLICE_X35Y45         FDRE (Prop_fdre_C_Q)         0.175     2.133 r  clk_div/clk_rx_uart_out_reg/Q
                     net (fo=2, routed)           0.302     2.436    rx_clk
BUFGCTRL_X0Y1        BUFG (Prop_bufg_I_O)         0.029     2.465 r  rx_clk_BUFG_inst/O
                     net (fo=39, routed)          0.867     3.331    nolabel_line85/rx_clk_BUFG
SLICE_X2Y1           FDRE                                         r  nolabel_line85/data_out_reg[1]/C
                        clock pessimism            -0.842     2.489
SLICE_X2Y1           FDRE (Hold_fdre_C_D)         0.076     2.565    nolabel_line85/data_out_reg[1]
-------------------------------------------------------------------
                        required time                        -2.565
                        arrival time                          2.742
-------------------------------------------------------------------
```

```
            slack                                     0.177
```

This report tries to calculate when destination register's data pin is changed after the clock edge for a path. To break down the information in the report:

- Time axis analyzed twice, one for data path, the other for hold time.
- The clock signal reached the source register at 2.476ns (Source Clock Delay), starting data path flow.
- Then the signal of source register is processed by the logic between and reached the destination register at 2.742ns. (Data Path Delay: 2.742ns - (2.476ns) = 0.266ns)
- The data pin is changed 2.742ns after the clock edge.
- Time axis analyzed again for hold time. Starting from 0ns, the clock signal reached the destination register at 2.489ns (Destination Clock Delay), starting hold time flow (for previous data that analyzed in setup time, this report looks for future change not back).
- After adding the hold time of the destination register, it is found that the data pin must not change before t=2.489ns+0.076ns(Hold Time).
- But we already found that the data pin is changed at t=2.742ns. Therefore, the hold slack is calculated as 0.177ns. (Time Arrival - Time Required).

To sum up, time required is when the destination register completed hold (the data before the current clock rise). Time arrival is the exact time data changes D pin. Data path delay is Q to D.

A graphical representation of the path as follows:

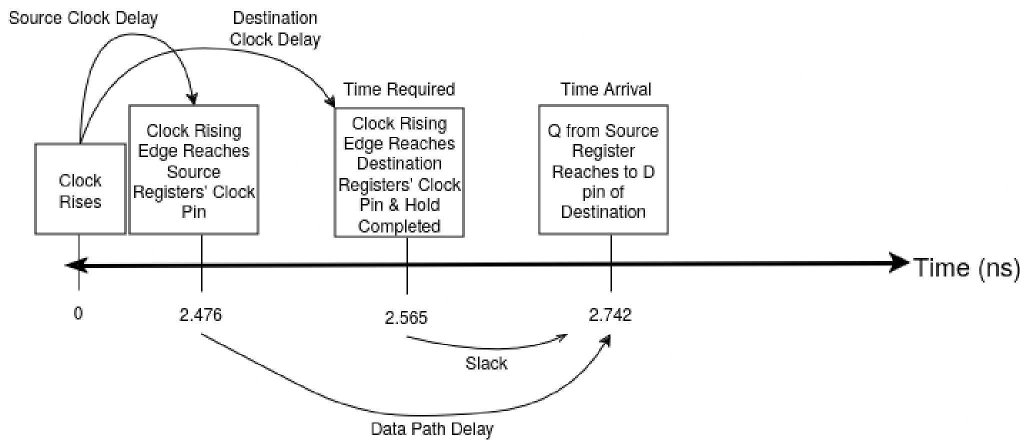

Source Clock Delay
Destination Clock Delay
Time Required
Time Arrival
Clock Rises
Clock Rising Edge Reaches Source Registers' Clock Pin
Clock Rising Edge Reaches Destination Registers' Clock Pin & Hold Completed
Q from Source Register Reaches to D pin of Destination
Time (ns)
0
2.476
2.565
2.742
Slack
Data Path Delay

Figure 5.3.4.2: Hold Timing Path

As this path reports are categorized by clock, hold and setup, you can find the problematic sequence by inspecting the data path report like above. There are several solutions available for negative setup slack:

- Add clock buffer to destination clock / Fix clock skew
- Optimizing the data path delay
- Pipeline the design (insert extra stages/states)
- Decrease the clock frequency

**In timing analysis, clock pessimism is initially introduced to ensure safety. However, when calculating timing slack (e.g., setup slack), the tools may apply Clock Pessimism Removal (CPR) to reduce or eliminate some of this conservative estimation. CPR adjusts the clock arrival times so that the analysis is more realistic. Therefore, the slack value is calculated by subtracting the arrival time from the required time. Don't get confused.*

*In order have a path in this table the clock related to the path must be registered in constraints file:

For External Clocks:

```
create_clock -period 2.225 -name clk_p -waveform {0.000 1.1125} [CLOCK PORT]
```

For Internal Clocks:

```
create_generated_clock -name clk -source [SOURCE CLOCK PIN/PORT] -divide_by 1 [OUTPUT PIN]
```

5.3.5 Input Delays and Importance to Timing Report

When input delay is not specified in the constraints file, the tool assumes that the input signal arrives at a perfect time, therefore slacks are given infinite. This can cause timing violations if the input signal arrives too close to the clock edge. Therefore, it is important to specify the input delay in the constraints file to ensure that the design meets the timing requirements. This can change:

- Your design might have a negative hold slack because of this issue, **which is not a real issue.**
- Your design meet the timing requirements, but **with an input delay setup time might be violated in practice.**

Therefore, to ensure that design meets the timing requirements in the real world, you should set the input delay in the constraints file. To set the input delay in the constraints file, you can use the following syntax:

```
set_input_delay -clock [clock_name] -max [delay_value] [port_name]
set_input_delay -clock [clock_name] -min [delay_value] [port_name]
```

This is range of delay that input signal can arrive any time inside. For hold time analysis -min is used. For setup time analysis -max is used by Vivado. **Testing the effect of input delay to the report** For this purpose a simple module is created as follows:

- Makes calculation using input and writes to a register.

```
module InputDelayTest(
    input wire clk,
    input wire reset,
    input wire [7:0] data_in,
    output reg [11:0] data_out
);
    reg [7:0] prev; // Data Store

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            data_out <= 9'b00000000;  // Initialize the register
            prev <= 8'b00000000;  // Initialize the register
        end else begin
            prev <= data_in;
            data_out <= prev * 14;
        end
    end
endmodule
```

The design is implemented, and the timing report is analyzed without constraints at first. Let's take the paths going into the prev register.

```
report_timing -from [get_ports *data_in*] -to [get_cells *prev_reg*]
-path_type summary -max_paths 10 -setup

Timing Report
Startpoint                     Endpoint                       Slack(ns)
----------------------------------------------------------------------------
data_in[0]                     prev_reg[6]/D                  inf
data_in[0]                     prev_reg[7]/D                  inf
data_in[0]                     prev_reg[5]/D                  inf
data_in[0]                     prev_reg[4]/D                  inf
data_in[0]                     prev_reg[3]/D                  inf
data_in[0]                     prev_reg[2]/D                  inf
data_in[0]                     prev_reg[1]/D                  inf

report_timing -from [get_ports *data_in*] -to [get_cells *prev_reg*]
-path_type summary -max_paths 10 -hold

Timing Report
Startpoint                     Endpoint                       Slack(ns)
----------------------------------------------------------------------------
data_in[6]                     prev_reg[7]/D                  inf
data_in[4]                     prev_reg[5]/D                  inf
data_in[3]                     prev_reg[4]/D                  inf
data_in[4]                     prev_reg[6]/D                  inf
data_in[1]                     prev_reg[2]/D                  inf
```

```
data_in[2]                  prev_reg[3]/D                 inf
data_in[0]                  prev_reg[1]/D                 inf
```

As you can see, the design meets the timing requirements without any constraints. Now, let's set the input delay to 5ns at maximum and 2ns at minimum and analyze the timing report again.

```
set_input_delay -clock [get_clocks sys_clk_pin] -max 5 [get_ports *data_in*]
set_input_delay -clock [get_clocks sys_clk_pin] -min 2 [get_ports *data_in*]
```

After setting the input delay, the timing report is analyzed again:

```
report_timing -from [get_ports *data_in*] -to [get_cells *prev_reg*]
-path_type summary -max_paths 10 -setup

Timing Report
Startpoint                  Endpoint                      Slack(ns)
---------------------------------------------------------------------
data_in[0]                  prev_reg[6]/D                 3.541
data_in[0]                  prev_reg[5]/D                 3.573
data_in[0]                  prev_reg[7]/D                 3.573
data_in[0]                  prev_reg[4]/D                 3.634
data_in[0]                  prev_reg[3]/D                 3.660
data_in[0]                  prev_reg[2]/D                 3.713
data_in[0]                  prev_reg[1]/D                 3.761

report_timing -from [get_ports *data_in*] -to [get_cells *prev_reg*]
-path_type summary -max_paths 10 -hold

Timing Report
Startpoint                  Endpoint                      Slack(ns)
---------------------------------------------------------------------
data_in[2]                  prev_reg[6]/D                 0.275
data_in[0]                  prev_reg[4]/D                 0.335
data_in[2]                  prev_reg[7]/D                 0.543
data_in[0]                  prev_reg[5]/D                 0.601
data_in[1]                  prev_reg[2]/D                 0.636
data_in[1]                  prev_reg[3]/D                 0.904
data_in[0]                  prev_reg[1]/D                 1.319
```

As you can see, the slack values are now finite, and the design meets the timing requirements with the input delay constraints. Let's examine the timings one more time by changing input time values that we can assume that it will create violation:

- As the worst setup slack is 3.541ns, if we set input delay to max 9ns, it is assumed to be violating the setup time.
- As the hold slack is 0.275ns, if we set input delay to min 0ns, it is assumed to be violating the hold time.

```
set_input_delay -clock [get_clocks sys_clk_pin] -max 9 [get_ports *data_in*]
set_input_delay -clock [get_clocks sys_clk_pin] -min 0 [get_ports *data_in*]
```

After setting the input delay, the timing report is analyzed again:

```
report_timing -from [get_ports *data_in*] -to [get_cells *prev_reg*] -path_type summary -max_paths 1
Timing Report

Startpoint                    Endpoint                      Slack(ns)
----------------------------------------------------------------------------
data_in[1]                    prev_reg[6]/D                 -2.161
data_in[1]                    prev_reg[7]/D                 -2.066
data_in[1]                    prev_reg[5]/D                 -2.050
data_in[1]                    prev_reg[4]/D                 -1.917
data_in[1]                    prev_reg[3]/D                 -1.857
data_in[1]                    prev_reg[2]/D                 -1.504
data_in[0]                    prev_reg[1]/D                 -0.752
```

```
report_timing -from [get_ports *data_in*] -to [get_cells *prev_reg*] -path_type summary -max_paths 1
Timing Report

Startpoint                    Endpoint                      Slack(ns)
----------------------------------------------------------------------------
data_in[5]                    prev_reg[6]/D                 0.067
data_in[2]                    prev_reg[3]/D                 0.139
data_in[3]                    prev_reg[4]/D                 0.163
data_in[0]                    prev_reg[1]/D                 0.276
data_in[5]                    prev_reg[7]/D                 0.335
data_in[2]                    prev_reg[5]/D                 0.403
data_in[0]                    prev_reg[2]/D                 0.410
```

As the reports indicate, the worst setup slack is dropped down by almost 5.7 which is more than the input delay we increased by 4. The hold slack is also decreased by 0.2ns. This is a good example to show the importance of input delay constraints in the timing analysis. **Even though the slack changes exact the amount of the delay due to required and arrival time factors, our hypothesis is correct. The design is not meeting the timing requirements anymore.**

5.4 TCL Commands

The most useful settings for timing analysis when calling TCL option are:

- -from [pin]: The starting point of the path to be analyzed.
- -to [pin]: The ending point of the path to be analyzed.
- -setup: The setup time analysis for the paths.
- -hold: The hold time analysis for the paths.

- -nworst or max_paths [#]: The number of the worst paths per clock domain if grouped or first # paths to be analyzed.
- -through [names]: The paths that are crossing a specific nets/pins/cells.
- -path_type summary: This is useful only to see slack and path.

You can combine these options, adding them to options place `report_timing [options]` to get the report you desire. To find pin/path/cell names, you can use `get_pins`, `get_paths`, `get_cells` commands, with search option `*keyword*`. Example: `get_pins *clk*`. You can combine these commands like this:

```
report_timing -to [get_cells *prev_reg*] -path_type summary -max_paths 20 -setup
```

*It is useful when the original report only gives worse 10 slacks, and you need to know the exact path slack for improvement. For example, you can find and get timings of all paths to a single register with these commands.

Chapter 6

Examining Reports of Common Designs & Questions

In this section, we will examine the reports of some common designs to understand the information provided in the reports, and we will see how LUT/DSP arithmetic, shifting registers effect power, timing, and resource utilization.

6.1 Multiplication

In this design, we will try to find the best way to multiply two 8,16,32-bit numbers. We will compare LUTs, DSPs, and using Multiply IP and '*' operator.

6.1.1 First Case

In this case, we will use the '*' operator and Multiplier IP (No Stages) to multiply two numbers, using only LUTs.

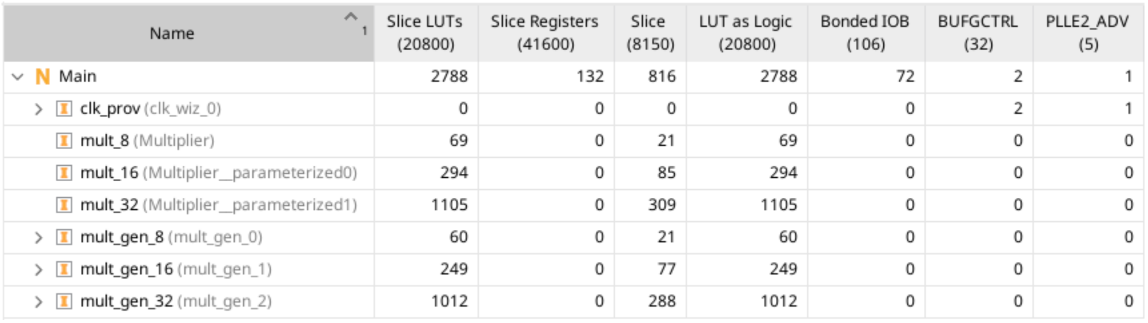
The modules used are below:

```
    (* USE_DSP="no", keep_hierarchy = "yes" *) module Multiplier #(parameter A_WIDTH = 8,
     B_WIDTH = 8)(input wire clk,
    input wire [A_WIDTH-1:0] a,
    input wire [B_WIDTH-1:0] b,
    output reg [A_WIDTH+B_WIDTH-1:0] product
);

    always @(clk) begin
        product <= a * b;
    end

endmodule
```

```
module Main(
    input wire clk, input wire [31:0] raw_a_32, input wire [31:0] raw_b_32,
    input wire rst, output reg [5:0] fake_out
    );

    wire AHundredMHz_clk;
    wire TwoHundredMHz_clk;
    wire locked;

    reg [31:0] a_32;
    reg [31:0] b_32;
    reg [15:0] product_16;
    reg [31:0] product_32;
    reg [63:0] product_64;
    reg [15:0] product_ip_16;
    reg [31:0] product_ip_32;
    reg [63:0] product_ip_64;

    clk_wiz_0 clk_prov(.clk_out1(AHundredMHz_clk), .clk_out2(TwoHundredMHz_clk), .reset(rst), .locke

    wire [15:0] product_w_16;
    wire [31:0] product_w_32;
    wire [63:0] product_w_64;
    wire [15:0] product_ip_w_16;
    wire [31:0] product_ip_w_32;
    wire [63:0] product_ip_w_64;

    Multiplier #(.A_WIDTH(8), .B_WIDTH(8)) mult_8(.clk(AHundredMHz_clk), .a(a_32[7:0]), .b(b_32[7:0
    Multiplier #(.A_WIDTH(16), .B_WIDTH(16)) mult_16(.clk(AHundredMHz_clk), .a(a_32[15:0]), .b(b_32
    Multiplier #(.A_WIDTH(32), .B_WIDTH(32)) mult_32(.clk(AHundredMHz_clk), .a(a_32[31:0]), .b(b_32

    mult_gen_0 mult_gen_8(a_32[7:0], b_32[7:0], product_ip_w_16);
    mult_gen_1 mult_gen_16(a_32[15:0], b_32[15:0], product_ip_w_32);
    mult_gen_2 mult_gen_32(a_32[31:0], b_32[31:0], product_ip_w_64);

    always @(posedge AHundredMHz_clk or posedge rst) begin
        if (rst) begin
            product_16 <= 16'b0;
            product_32 <= 32'b0;
            product_64 <= 64'b0;
            product_ip_16 <= 16'b0;
            product_ip_32 <= 32'b0;
            product_ip_64 <= 64'b0;
        end
        else begin
            product_16 <= product_w_16;
```

```
                product_32 <= product_w_32;
                product_64 <= product_w_64;
                product_ip_16 <= product_ip_w_16;
                product_ip_32 <= product_ip_w_32;
                product_ip_64 <= product_ip_w_64;
                fake_out[0] <= product_16[15];
                fake_out[1] <= product_32[31];
                fake_out[2] <= product_64[63];
                fake_out[3] <= product_ip_16[15];
                fake_out[4] <= product_ip_32[31];
                fake_out[5] <= product_ip_64[63];
                a_32 <= raw_a_32;
                b_32 <= raw_b_32;
            end
        end
endmodule
```

The results of the first case are shown below: **Utilization:**

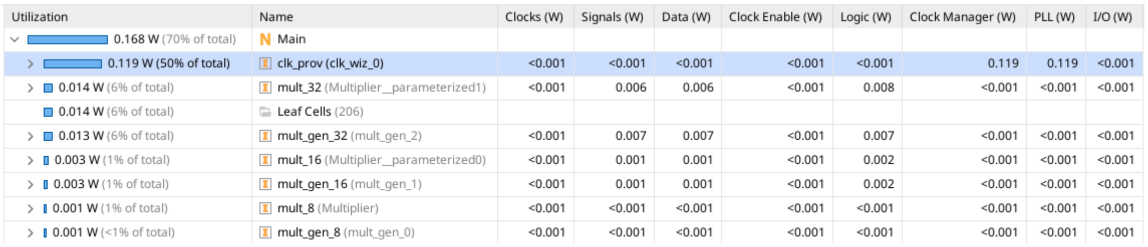


Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)	PLLE2_ADV (5)
Main	2788	132	816	2788	72	2	1
clk_prov (clk_wiz_0)	0	0	0	0	0	2	1
mult_8 (Multiplier)	69	0	21	69	0	0	0
mult_16 (Multiplier__parameterized0)	294	0	85	294	0	0	0
mult_32 (Multiplier__parameterized1)	1105	0	309	1105	0	0	0
mult_gen_8 (mult_gen_0)	60	0	21	60	0	0	0
mult_gen_16 (mult_gen_1)	249	0	77	249	0	0	0
mult_gen_32 (mult_gen_2)	1012	0	288	1012	0	0	0

Figure 6.1.1.1: Only LUT Multiplier Utilization Hierarchical

The submodules that have 'gen' in their names are IP multipliers build with only LUTs. The results indicate that the IP has better resource utilization than the '*' operator.

Power:



Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Logic (W)	Clock Manager (W)	PLL (W)	I/O (W)
0.168 W (70% of total)	Main								
0.119 W (50% of total)	clk_prov (clk_wiz_0)	<0.001	<0.001	<0.001	<0.001	<0.001	0.119	0.119	<0.001
0.014 W (6% of total)	mult_32 (Multiplier__parameterized1)	<0.001	0.006	0.006	<0.001	0.008	<0.001	<0.001	<0.001
0.014 W (6% of total)	Leaf Cells (206)								
0.013 W (6% of total)	mult_gen_32 (mult_gen_2)	<0.001	0.007	0.007	<0.001	0.007	<0.001	<0.001	<0.001
0.003 W (1% of total)	mult_16 (Multiplier__parameterized0)	<0.001	0.001	0.001	<0.001	0.002	<0.001	<0.001	<0.001
0.003 W (1% of total)	mult_gen_16 (mult_gen_1)	<0.001	0.001	0.001	<0.001	0.002	<0.001	<0.001	<0.001
0.001 W (1% of total)	mult_8 (Multiplier)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
0.001 W (<1% of total)	mult_gen_8 (mult_gen_0)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Figure 6.1.1.2: Only LUT Multiplier Power Hierarchical

The results indicate that the IP has better power utilization than the '*' operator.

Timing: Critical path timing (Violating) of the design as follows:

```
a_32_reg Register -> LUT 32 IP -> product_ip_64 Register

Slack (VIOLATED) :        -0.973ns  (required time - arrival time)
  Source:                 a_32_reg[8]/C
                            (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@0.
  Destination:            product_ip_64_reg[63]/D
                            (rising edge-triggered cell FDCE clocked by clk_out1_clk_wiz_0  {rise@0.
  Path Group:             clk_out1_clk_wiz_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_out1_clk_wiz_0 rise@10.000ns - clk_out1_clk_wiz_0 rise@0.00
  Data Path Delay:        10.936ns  (logic 6.760ns (61.812%)  route 4.176ns (38.188%))
  Logic Levels:           23  (CARRY4=18 LUT2=4 LUT4=1)
  Clock Path Skew:        -0.031ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -2.062ns = ( 7.938 - 10.000 )
    Source Clock Delay      (SCD):    -2.457ns
    Clock Pessimism Removal (CPR):    -0.425ns
  Clock Uncertainty:      0.068ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.116ns
    Phase Error              (PE):    0.000ns

b_32_reg Register -> LUT 32 '*' Logic -> product_64 Register

Slack (VIOLATED) :        -0.776ns  (required time - arrival time)
  Source:                 b_32_reg[4]/C
                            (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@0.
  Destination:            product_64_reg[63]/D
                            (rising edge-triggered cell FDCE clocked by clk_out1_clk_wiz_0  {rise@0.
  Path Group:             clk_out1_clk_wiz_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_out1_clk_wiz_0 rise@10.000ns - clk_out1_clk_wiz_0 rise@0.00
  Data Path Delay:        10.851ns  (logic 5.250ns (48.381%)  route 5.601ns (51.619%))
  Logic Levels:           21  (CARRY4=15 LUT3=2 LUT4=1 LUT5=1 LUT6=2)
  Clock Path Skew:        0.034ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -2.002ns = ( 7.998 - 10.000 )
    Source Clock Delay      (SCD):    -2.462ns
    Clock Pessimism Removal (CPR):    -0.425ns
  Clock Uncertainty:      0.068ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.116ns
    Phase Error              (PE):    0.000ns

a_32_reg Register -> LUT 16 '*' Logic -> product_32 Register
Slack (VIOLATED) :        -0.131ns  (required time - arrival time)
```

```
  Source:                 a_32_reg[12]/C
                            (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@0.
  Destination:            product_32_reg[31]/D
                            (rising edge-triggered cell FDCE clocked by clk_out1_clk_wiz_0  {rise@0.
  Path Group:             clk_out1_clk_wiz_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_out1_clk_wiz_0 rise@10.000ns - clk_out1_clk_wiz_0 rise@0.00
  Data Path Delay:        10.134ns  (logic 3.673ns (36.244%)  route 6.461ns (63.756%))
  Logic Levels:           12  (CARRY4=7 LUT3=1 LUT4=1 LUT5=1 LUT6=2)
  Clock Path Skew:        -0.038ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -2.067ns = ( 7.933 - 10.000 )
    Source Clock Delay      (SCD):    -2.455ns
    Clock Pessimism Removal (CPR):    -0.425ns
  Clock Uncertainty:      0.068ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.116ns
    Phase Error              (PE):    0.000ns

a_32_reg Register -> LUT 32 '*' IP -> product_ip_32 Register
Slack (VIOLATED) :        -0.104ns  (required time - arrival time)
  Source:                 a_32_reg[0]/C
                            (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@0.
  Destination:            product_ip_32_reg[31]/D
                            (rising edge-triggered cell FDCE clocked by clk_out1_clk_wiz_0  {rise@0.
  Path Group:             clk_out1_clk_wiz_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_out1_clk_wiz_0 rise@10.000ns - clk_out1_clk_wiz_0 rise@0.00
  Data Path Delay:        10.107ns  (logic 5.046ns (49.927%)  route 5.061ns (50.073%))
  Logic Levels:           14  (CARRY4=10 LUT2=4)
  Clock Path Skew:        -0.038ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -2.070ns = ( 7.930 - 10.000 )
    Source Clock Delay      (SCD):    -2.458ns
    Clock Pessimism Removal (CPR):    -0.425ns
  Clock Uncertainty:      0.068ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.116ns
    Phase Error              (PE):    0.000ns
```

The results indicate that 32-bit multiplication has timing violation for both. The '*' operator seems to more close to 0 slack than the IP multiplier for 32-Bit. However, for the 16-bit violation, the IP multiplier has better slack than the '*' operator. Using LUT multipliers ('*' or IP) over 8-bit is not recommended because of inefficient resource utilization and timing violations.

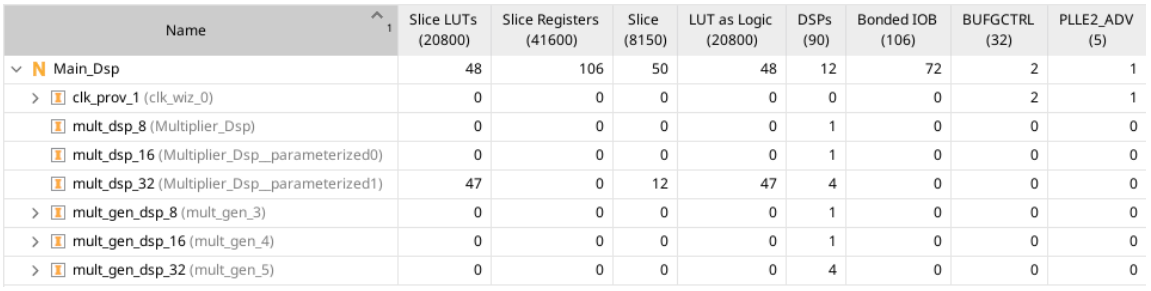
6.1.2 Second Case

In this case, we will use the '*' operator and Multiplier IP (No Stages) to multiply two numbers, using DSPs.

The modules used are below:

```
(* USE_DSP="yes", keep_hierarchy = "yes" *) module Multiplier_Dsp #(parameter A_WIDTH = 8,
     B_WIDTH = 8)(input wire clk,
    input wire [A_WIDTH-1:0] a,
    input wire [B_WIDTH-1:0] b,
    output reg [A_WIDTH+B_WIDTH-1:0] product
);

    always @(clk) begin
        product <= a * b;
    end
endmodule
module Main_Dsp(
input wire clk, input wire [31:0] raw_a_32, input wire [31:0] raw_b_32, input wire rst, output reg
    );

    wire AHundredMHz_clk;
    wire TwoHundredMHz_clk;
    wire locked;

    reg [31:0] a_32;
    reg [31:0] b_32;
    reg [15:0] product_16;
    reg [31:0] product_32;
    reg [63:0] product_64;
    reg [15:0] product_ip_16;
    reg [31:0] product_ip_32;
    reg [63:0] product_ip_64;

    clk_wiz_0 clk_prov_1(.clk_out1(AHundredMHz_clk), .clk_out2(TwoHundredMHz_clk), .reset(rst), .lo

    wire [15:0] product_w_16;
    wire [31:0] product_w_32;
    wire [63:0] product_w_64;
    wire [15:0] product_ip_w_16;
    wire [31:0] product_ip_w_32;
    wire [63:0] product_ip_w_64;

    Multiplier_Dsp #(.A_WIDTH(8), .B_WIDTH(8)) mult_dsp_8(.clk(AHundredMHz_clk), .a(a_32[7:0]), .b(
    Multiplier_Dsp #(.A_WIDTH(16), .B_WIDTH(16)) mult_dsp_16(.clk(AHundredMHz_clk), .a(a_32[15:0]),
    Multiplier_Dsp #(.A_WIDTH(32), .B_WIDTH(32)) mult_dsp_32(.clk(AHundredMHz_clk), .a(a_32[31:0]),
    mult_gen_3 mult_gen_dsp_8(a_32[7:0], b_32[7:0], product_ip_w_16);
    mult_gen_4 mult_gen_dsp_16(a_32[15:0], b_32[15:0], product_ip_w_32);
    mult_gen_5 mult_gen_dsp_32(a_32[31:0], b_32[31:0], product_ip_w_64);

    always @(posedge AHundredMHz_clk or posedge rst) begin
        if (rst) begin
            product_16 <= 16'b0;
            product_32 <= 32'b0;
            product_64 <= 64'b0;
            product_ip_16 <= 16'b0;
            product_ip_32 <= 32'b0;
            product_ip_64 <= 64'b0;
        end
        else begin
            product_16 <= product_w_16;
            product_32 <= product_w_32;
            product_64 <= product_w_64;
            product_ip_16 <= product_ip_w_16;
            product_ip_32 <= product_ip_w_32;
            product_ip_64 <= product_ip_w_64;
            fake_out[0] <= product_16[15];
            fake_out[1] <= product_32[31];
            fake_out[2] <= product_64[63];
            fake_out[3] <= product_ip_16[15];
            fake_out[4] <= product_ip_32[31];
            fake_out[5] <= product_ip_64[63];
            a_32 <= raw_a_32;
            b_32 <= raw_b_32;
        end
    end
endmodule
```

The results of the second case are shown below: **Utilization:**

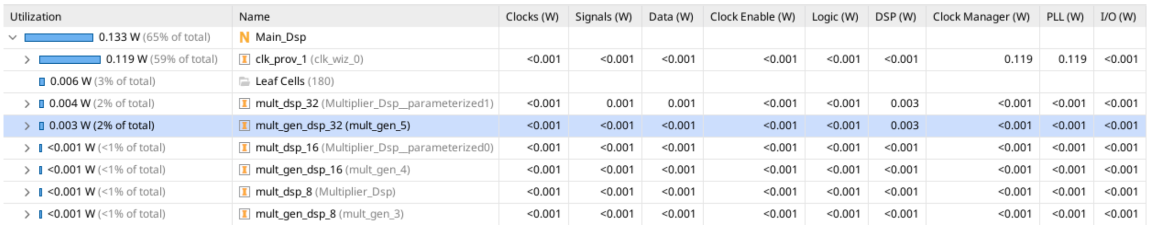


Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)	PLLE2_ADV (5)
Main_Dsp	48	106	50	48	12	72	2	1
clk_prov_1 (clk_wiz_0)	0	0	0	0	0	0	2	1
mult_dsp_8 (Multiplier_Dsp)	0	0	0	0	1	0	0	0
mult_dsp_16 (Multiplier_Dsp__parameterized0)	0	0	0	0	1	0	0	0
mult_dsp_32 (Multiplier_Dsp__parameterized1)	47	0	12	47	4	0	0	0
mult_gen_dsp_8 (mult_gen_3)	0	0	0	0	1	0	0	0
mult_gen_dsp_16 (mult_gen_4)	0	0	0	0	1	0	0	0
mult_gen_dsp_32 (mult_gen_5)	0	0	0	0	4	0	0	0

Figure 6.1.2.1: DSP Multiplier Utilization Hierarchical

The submodules that have 'gen' in their names are IP multipliers built with only DSPs. The results indicate that the IP has better resource utilization than the '*' operator for only 32-bit multiplication. Otherwise, same.

Power:



Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Logic (W)	DSP (W)	Clock Manager (W)	PLL (W)	I/O (W)
0.133 W (65% of total)	Main_Dsp									
0.119 W (59% of total)	clk_prov_1 (clk_wiz_0)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.119	0.119	<0.001
0.006 W (3% of total)	Leaf Cells (180)									
0.004 W (2% of total)	mult_dsp_32 (Multiplier_Dsp__parameterized1)	<0.001	0.001	0.001	<0.001	<0.001	0.003	<0.001	<0.001	<0.001
0.003 W (2% of total)	mult_gen_dsp_32 (mult_gen_5)	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	mult_dsp_16 (Multiplier_Dsp__parameterized0)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	mult_gen_dsp_16 (mult_gen_4)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	mult_dsp_8 (Multiplier_Dsp)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
<0.001 W (<1% of total)	mult_gen_dsp_8 (mult_gen_3)	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

Figure 6.1.2.2: DSP Multiplier Power Hierarchical

For 32-bit, IP multiplier has better power utilization than the '*' operator. Otherwise, they are similar.

Timing: Critical path timing (Violating) of the design as follows:

```
Slack (VIOLATED) :        -0.644ns  (required time - arrival time)
  Source:                 a_32_reg[10]_replica/C
                            (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@0.
  Destination:            product_ip_64_reg[63]/D
                            (rising edge-triggered cell FDCE clocked by clk_out1_clk_wiz_0  {rise@0.
  Path Group:             clk_out1_clk_wiz_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_out1_clk_wiz_0 rise@10.000ns - clk_out1_clk_wiz_0 rise@0.00
  Data Path Delay:        10.540ns  (logic 9.436ns (89.525%)  route 1.104ns (10.475%))
  Logic Levels:           4  (DSP48E1=4)
  Clock Path Skew:        -0.020ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -2.061ns = ( 7.939 - 10.000 )
    Source Clock Delay      (SCD):    -2.454ns
    Clock Pessimism Removal (CPR):    -0.412ns
  Clock Uncertainty:      0.068ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.116ns
    Phase Error              (PE):    0.000ns


Slack (VIOLATED) :        -0.068ns  (required time - arrival time)
  Source:                 b_32_reg[6]_replica/C
                            (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@0.
  Destination:            product_64_reg[63]/D
                            (rising edge-triggered cell FDCE clocked by clk_out1_clk_wiz_0  {rise@0.
  Path Group:             clk_out1_clk_wiz_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk_out1_clk_wiz_0 rise@10.000ns - clk_out1_clk_wiz_0 rise@0.00
  Data Path Delay:        10.043ns  (logic 7.611ns (75.785%)  route 2.432ns (24.215%))
  Logic Levels:           10  (CARRY4=7 DSP48E1=2 LUT2=1)
  Clock Path Skew:        -0.019ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -2.061ns = ( 7.939 - 10.000 )
    Source Clock Delay      (SCD):    -2.455ns
    Clock Pessimism Removal (CPR):    -0.412ns
  Clock Uncertainty:      0.068ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.116ns
    Phase Error              (PE):    0.000ns
```

The results indicate that 32-bit multiplication has timing violation for both. The '*' operator seems to more close to 0 slack than the IP multiplier for 32-Bit. When other met max delay paths are inspected, the IP multiplier data path delay is larger than '*' in general. For a no stage multiplication using DSP a '*' operator might be the choice for better slack given a bit more area is used.

Given the results using DSP blocks are much more efficient than using LUTs for multiplication for larger bit arrays. If the design intends to use no pipeline, the '*' operator might be the choice for better slack and usage simplicity as the results are close with no stage IP.

6.2 Slack / Clock / Pipeline Relationship Investigation

In this part we will try to answer following questions:

1. Can we assume that if we increase clock period by X seconds, slack will increase by X seconds?
2. How does Multiplier IP stages effect slack? (Even delay paths between stages or not)
3. Using the information from first two questions, can we fine tune the clock period given to IP multiplier to have a single clock cycle result in the top module?
4. Why does the IP setup suggests a stage level? Is it related to the maximum clock speed of the device?

6.2.1 First Question

In this part, we will try to understand the relationship between slack and clock period. We will try to increase the clock period by X seconds and see if the slack increases by X seconds. We will use the same module from the DSP part of the multiplication part using only IP Multipliers.

The clock period was set to 10ns and worst slack was around -0.644ns. New clock period will be:

$$\text{Closest integer} = \left\lfloor \frac{1}{10\,\text{ns} + 0.644} + \frac{1}{2} \right\rfloor = 93896714 \approx 93.75\text{MHZ}$$

The hypothesis is that if we set the clock period to 10.667ns, the slack will be positive close to 0. The results are shown below:

```
-------------------------------------------------------------------
    required time                         8.073
    arrival time                         -8.073
```

```
-------------------------------------------------------------------
    slack                                   -0.001
```

It failed again. It is observed that the slack is close to 0. The hypothesis is close to be correct. The slack is increased by 0.649ns when the clock period is increased by 0.667ns. Given the phase error and jitter of the clock it is more safe to increase the clock period more than required.

Trying again with 93.333MHZ, 10.71ns:

```
    -------------------------------------------------------------------
    required time                            8.117
    arrival time                            -8.076
-------------------------------------------------------------------
    slack                                    0.041
```

Slack is tested again with other clocks as data path delay is changing with the clock When the results are combined (Note that the non IP parts are removed therefore the slack at 100MHZ is different from the previous results):

Table 6.1: Timing Analysis at Different Clock Frequencies

Path Delay	Clk frq (MHz)	Period (ns)	Slack
11.093	50	20.000	8.657
10.538	93.333	10.714	0.041
10.535	93.750	10.535	-0.001
10.373	100	10.000	-0.495
10.361	200	5.000	-5.488

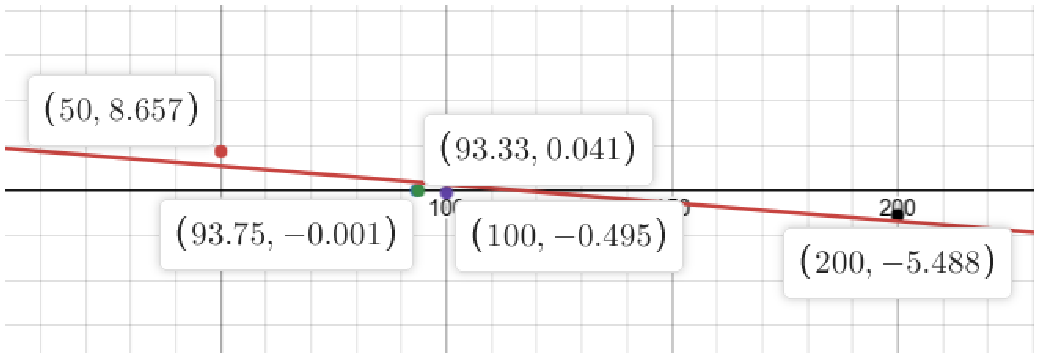


Figure 6.2.1.1: Frequency vs Slack Graph

By looking and plotting the data we can see that the data path delay and thus the slack values are not linearly related to the clock period. As frequency increases, change of the path delay decreases. By doing this analysis we can conclude that there is no linear relationship between slack and clock period, but there is a relation close to linear at higher clock speeds as data path delay tends to be constant.

6.2.2 Second Question

The second question requires us to find how does Multiplier IP pipelines multiple stages. To find a relationship, we will use critical path and how it is reduced after each stage. Same module will be used with different stages of IP multipliers.

During this test we will use 10ns clock period. It has initially the worst slack as -0.495ns. There kind of paths are critical:

- any a or b register to IP (it can have a weird name like mult_gen_dsp_32/U0/i_mult/gDSP.gDSP_only.iDSP/ use_prim.appDSP[1].bppDSP[1].use_dsp.use_dsp48e1.iDSP48E1/PCIN[0]) which refers to a flip-flop inside the DSP block)
- any path from IP to output register
- any path from IP register to IP register

1. **Stage 0:** Slowest path is from a_32[i] register to the product register through the IP. **The max data path delay is 10.373 ns.**
2. **Stage 1:** Slowest path is from a_32[i] register to a DSP register inside the IP. **Data path delay is 8.503 ns.** Searching a path from an IP to output register, **the longest path is 1.119 ns**, and it is in the 32-bit multiplier. It suggests that the IP doing most of the job in the first cycle.
3. **Stage 2:** Slowest path is from a DSP register inside the IP to a DSP register inside the IP (mult_gen_dsp_32/U0/i_mult/gDSP.gDSP_only.iDSP/use_prim.appDSP[0].bppDSP[0].use_dsp.use_dsp48e1.iDSP48 to mult_gen_dsp_32/U0/i_mult/gDSP.gDSP_only.iDSP/use_prim.appDSP[1].bppDSP[1]). **Data path delay is 6.633 ns**. It suggests that most of the job is done in the intermediate cycle. In the first cycle the maximum data path is to the 32-bit multiplier IP and **Its data path delay is 2.815 ns.** The maximum data path delay from the IPs to output register is 1.180 ns and in the 32-bit multiplier.
4. **Stage 3:** From timing report couldn't detect pipelining paths due to hidden information, but intermediate maximum data path delay is **4.208 ns**. Input to the IP paths has maximum delay **2.815 ns**. The maximum data path delay from the IPs to output register is **1.119 ns.** These are the maximum delay paths, and they are inside the 32-bit multiplier IP.

From this information we can come up with a table:

	Critical Path Delays (ns)		
# of Stages	Input	Intermediate Step/s	Output
0	Combinational*		10.373
1	8.503	-	1.119
2	2.815	6.633	1.180
3	2.815	4.208	1.119

Table 6.2: Critical Path Delays

Input refers to the delay from the input register to the IP, intermediate step/s refers to the delay inside the IPs, and output refers to the delay from the IP to the output register. A few things can be inferred from the table:

- The maximum delay path is reduced after each stage by around 2ns.
- Stages are not divided equally. An intermediate step is doing most of the job, or first one if it is a stage 0 and stage 1 IP.
- After 2 stages data path that gets input and gives output stays similar as job is done in the intermediate steps.

To be sure that the data path delay is reduced by 2ns after each stage, the reports for 4 stages and check the data path delays is checked. The maximum delay path in the IP and the whole system reduced to 2.428 as expected. At 5th stage, intermediate step path delays are below input and output path delays and **almost all of their delays are net delay**, so we can't say at every stage 2ns reduced.

6.2.3 Third Question

The investigation indicates that we can't assume that the stages are equally divided, and we cannot multiply clock by 2 as we increase the pipeline stages by one.

6.2.4 Fourth Question

Another mystery that why and how the optimum stage level of multiplier IP suggested by Vivado. The hypothesis is that it is related to the minimum clock period that the device, Basys3, can handle. Basys3 can handle maximum **463MHZ** clock frequency as documented, which corresponds to 2.159ns. The IP suggests a stage level to be 6 for 32-bit unsigned multiplier. We will check:

- The maximum delay path and the slack in the IP at stage level suggested.
- If it is below 2.159ns, and we have a slack around 8ns, we will check slack given the clock period is 2.159ns because given clock related time delays data path delay doesn't directly mean time required.
- To finalize the hypothesis, we will check again at stage level 5 if it is failing timing.

Checking at Stage Level 6: The result showed that the worst data path delay in the whole system is 2.585ns. Minimum slack is 6.852ns. However, higher clock speeds can make the DSP even faster as we already found. Therefore, it is worth to check the slack at 2.222ns clock period.

Checking at 2.222ns Clock Period: The results indicate that the design got faster. The worst path delay is 1.802ns. However, due to clock effects it is slightly negative, -0.045ns. Which means that we can try to work with 400MHZ clock.

Checking at 2.5ns Clock Period: After decreasing frequency to 400MHZ, the worst slack is:

```
      Slack (MET) :             0.014ns  (required time - arrival time)
      -------------------------------------------------------------------
                         required time                        -0.667
                         arrival time                          0.681
-------------------------------------------------------------------
                         slack                                 0.014
```

This test indicted that the optimum suggested stage level is related to the maximum clock speed of the device. The IP suggests a stage level that can handle the maximum clock speed of the device. Due to clock uncertainty it is same to use a smaller clock period than the maximum clock speed of the device as slack report showed. The report showed clock uncertainty and adds it to time domain to find the slack:

```
Clock Uncertainty:      0.057ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
Total System Jitter     (TSJ):    0.071ns
Discrete Jitter          (DJ):    0.090ns
Phase Error              (PE):    0.000ns
```

This is why at 450MHZ clock period, the slack was negative. The clock uncertainty is 0.057ns and the slack was -0.045ns.

6.3 Double Data Rate (DDR)

In this part, we will how to get maximum speed of an FPGA using double data rate.

6.3.1 Introduction

Double Data Rate (DDR) is a technique that allows to transfer data at **both rising and falling edges** of the clock. It is used in many applications to increase the data transfer rate. In this part, we will try to implement a DDR module and check the maximum clock speed that can be achieved. The use cases and drawbacks of DDR are:

- When limited by maximum generate-able clock speed of the device.
- As clock speed stays the same, power consumption can be reduced while speed is twice.
- However, FFs are used twice as much as in a normal design. As they can't work for both edges.
- Clock Multiplexing Logic is used to select the right data path to work for the specific edge.
- Another utilization is that a clock signal is divided into two and used for different data paths.
- By doing same operations in a parallel way in vice versa edges, the speed is doubled.

6.3.2 Implementation

In this part, we will implement a DDR module that will take two 8-bit input and output the multiplied 16-bit data. The process is divided to falling and rising edges of the clock. The module is implemented below:

```
(* USE_DSP="yes", keep_hierarchy = "yes" *) module Multiplier_Dsp #(parameter A_WIDTH = 8, B_WIDTH =
    input wire [A_WIDTH-1:0] a,
    input wire [B_WIDTH-1:0] b,
    output wire [A_WIDTH+B_WIDTH-1:0] product
);
    assign product = a * b;
endmodule

module OutMux(
    input selector, input [7:0] rise, fall, output reg [7:0] mul_out
    );

    always @(selector, rise, fall) begin
        case(selector)
            1'b0: mul_out <= fall;
            1'b1: mul_out <= rise;
            default: mul_out <= fall;
        endcase
    end
endmodule

module DDR4_Module(
    input wire clk, // Clock
    input wire [3:0] data_in, // Data input
    input wire [3:0] data_in_2, // Data input
    output wire [7:0] data_out // Data output;
    );

    reg [3:0] data_reg;
    reg [3:0] data_2_reg;
    reg [3:0] data_reg_fall;
    reg [3:0] data_2_reg_fall;

    wire [7:0] out_mul;
    wire [7:0] out_mul_fall;

    (* keep = "true" *) reg [7:0] data_out_reg;
    (* keep = "true" *) reg [7:0] data_out_reg_fall;

    OutMux out_mux (
        .selector(clk),
        .rise(data_out_reg),
        .fall(data_out_reg_fall),
        .mul_out(data_out)
    );

    Multiplier_Dsp #(
        .A_WIDTH(4),
        .B_WIDTH(4)
    ) dsp_mul_inst (
        .clk(clk),
        .a(data_reg),
        .b(data_2_reg),
        .product(out_mul)
    );
    Multiplier_Dsp #(
        .A_WIDTH(4),
        .B_WIDTH(4)
    ) dsp_mul_inst_2 (
        .clk(clk),
        .a(data_reg_fall),
        .b(data_2_reg_fall),
        .product(out_mul_fall)
    );
    always @(posedge clk) begin
        data_2_reg <= data_in_2;
        data_reg <= data_in;
        data_out_reg <= out_mul;
    end
    always @(negedge clk) begin
        data_2_reg_fall <= data_in_2;
        data_reg_fall <= data_in;
        data_out_reg_fall <= out_mul_fall;
    end
endmodule
```

By using the DDR module, we will try to fit multiplication of 4-bit numbers into a half clock cycle. The chosen clock period is 10ns. Parallel data paths are used to multiply the numbers in both edges of the clock. A mux with selector as clock is used to select the right data path to output. The results are shown below:

6.3.3 Utilization Hierarchical:

```
+------------------+-------------------+------------+------------+---------+-----+------------+
|     Instance     |       Module      | Total LUTs | Logic LUTs | LUTRAMs | FFs | DSP Blocks |
+------------------+-------------------+------------+------------+---------+-----+------------+
| DDR4_Module      |             (top) |          4 |          4 |       0 |  16 |          2 |
|   (DDR4_Module)  |             (top) |          0 |          0 |       0 |  16 |          0 |
|   dsp_mul_inst   | Multiplier_Dsp__1 |          0 |          0 |       0 |   0 |          1 |
|   dsp_mul_inst_2 |    Multiplier_Dsp |          0 |          0 |       0 |   0 |          1 |
|   out_mux        |            OutMux |          4 |          4 |       0 |   0 |          0 |
+------------------+-------------------+------------+------------+---------+-----+------------+
```

Instead of using twice frequency, we used twice the utilization of primitive types. The results indicate that the DDR module uses 2 DSP blocks and 16 FFs. The FFs are used twice as much as in a normal design. The DSP blocks are used to multiply the numbers in both edges of the clock.

6.3.4 Timing Summary:

We will look timing summary of the module and check the slowest path and slack to see if the limiting factor is DSP block.

```
| Design Timing Summary
| ---------------------
    WNS(ns)      TNS(ns)    WHS(ns)      THS(ns)
    -------      -------    -------      -------
    0.072        0.000        1.480        0.000
```

Worst setup slack is close to zero. This indicates that the module is almost running its maximum speed that it can run. The worst slack path is shown below:

```
Slack (MET) :       0.072ns  (required time - arrival time)
Source:             data_2_reg_reg[2]data_2_reg_fall_reg[2]/C
                    (rising edge-triggered cell IDDR clocked by sys_clk_pin  {rise@0.000ns fall@5.0
Destination:        data_out_reg_reg[3]/D
                    (rising edge-triggered cell FDRE clocked by sys_clk_pin  {rise@0.000ns fall@5.0
```

This means that the path is going from input buffer register's Q to output buffer register's data pin. The path goes through DSP as we know. **So the 4-bit multiplication can not be done using single DSP and no staging faster.** We can also claim that this module can't run if had pipelined the buffering and multiplication as buffer blocks add significant delays.

6.3.5 Power:

The power consumption of the module is shown below:

```
+----------------+-----------+----------+-----------+-----------------+
| On-Chip        | Power (W) | Used     | Available | Utilization (%) |
+----------------+-----------+----------+-----------+-----------------+
| Clocks         |     0.002 |        3 |       --- |             --- |
| Slice Logic    |    <0.001 |       45 |       --- |             --- |
|   LUT as Logic |    <0.001 |        4 |     20800 |            0.02 |
|   Register     |    <0.001 |       16 |     41600 |            0.04 |
|   Others       |     0.000 |       21 |       --- |             --- |
| Signals        |    <0.001 |       56 |       --- |             --- |
| DSPs           |    <0.001 |        2 |        90 |            2.22 |
| I/O            |     0.031 |        25 |       106 |           23.58 |
| Static Power   |     0.070 |           |           |                 |
| Total          |     0.104 |           |           |                 |
+----------------+-----------+-----------+-----------+-----------------+
| Dynamic (W)              | 0.034        |
| Device Static (W)        | 0.070        |
```

As we can see the utilization difference we have made does not affect the power consumption significantly unlike clocks. The 200MHZ clock would significantly dynamic power consumption. If clock gating is used, the power consumption can be more efficient for 200MHZ as static power is higher in this device.

6.3.6 Conclusion

DDR is used to achieve speed that can be achieved using twice clock speed. This is a requirement choice. Clock generation limitations, or power efficiency preferences, pipelining choices would affect the choice of using DDR. The results indicate that DDR can be used to achieve higher speeds. However, the design should be carefully implemented as the FFs are used twice as much as in a normal design.

6.4 Clock Shifting

In this section we will investigate clock phase shifting techniques: Resampling with a faster clock, PLL, Buffers.

6.4.1 Introduction

Clock phase can be shifted for the following purposes

- Negative setup slacks can be fixed **delaying clock of destination register.**
- Negative hold slacks can be fixed **delaying clock of source register.**

6.4.2 Module to Test

There is module that has setup slack and uses clock sampler module that will be explained in the next section:

```
(* USE_DSP = "yes" *) module DDR_Module(
input wire clk, // Clock
input wire [7:0] data_in, // Data input
input wire [7:0] data_in_2, // Data input
output reg [15:0] data_out // Data output;
);

reg [7:0] data_reg;
reg [7:0] data_2_reg;
wire [15:0] out_mul;
```

```
    Multiplier_Dsp #(
        .A_WIDTH(8),
        .B_WIDTH(8)
    ) dsp_mul_inst (
        .clk(clk_100_shifted_mhz),
        .a(data_reg),
        .b(data_2_reg),
        .product(out_mul)
    );

    wire clk_400_mhz;
    wire clk_100_mhz;
    wire clk_100_shifted_mhz;

    clk_wiz_0 clk_gen_0
   (
    // Clock out ports
    .clk_out1(clk_100_mhz),     // output clk_out1
    .clk_out2(clk_400_mhz),     // output clk_out2
    // Status and control signals
    .reset(1'b0), // input reset
   // Clock in ports
    .clk_in1(clk)      // input system_clk
    );

    ClockSampler clk_sampler(.clk(clk_400_mhz), .clk_raw(clk_100_mhz), .clk_sampled(clk_100_shifted_

    always @(posedge clk_100_mhz) begin
        data_2_reg <= data_in_2;
        data_reg <= data_in;
    end

    always @(negedge clk_100_shifted_mhz) begin
        data_out <= out_mul;
    end

endmodule
```

`data_out` register has a negative slack around 1.5ns. We will be shifting its clock by 2.5ns to fix this issue.

6.4.3 Clock Resampling

A clock can be delayed by a serial input shift register with a fast clock and the source clock to be shifted as serial input.

The verilog module for the intented design is below:

```
module ClockSampler(input wire clk, input wire clk_raw, output reg clk_sampled);
always @(posedge clk) begin
    clk_sampled <= clk_raw;
end
endmodule
```

Suppose that you have a 100MHZ clock, and you want to shift it 90 degrees. You should connect a 400MHZ clock to clk input of this module and 100MHZ to clk_raw pin. The output will be a 2.5ns shifted 100MHZ signal. To formulate:

$$Tp = \text{Original Clock Period}$$

$$N = \text{Number of Shifts}$$

$$Ts = \text{Sampling Clock Period}$$

$$\phi = \frac{Ts \cdot N}{Tp} \cdot 360^\circ$$

Timing Summary: There is no negative slack in the design. The results indicate that the clock shifting is successful. The worst slack is inside the clock sampler in the whole design The results are shown below:

```
Slack (MET) :       0.008ns  (required time - arrival time)
Source:             clk_gen_0/inst/plle2_adv_inst/CLKOUT0
    (clock source 'clk_out1_clk_wiz_0'  {rise@0.000ns fall@5.000ns period=10.000ns})
Destination:        clk_sampler/clk_sampled_reg/D
Path Group:         clk_out2_clk_wiz_0
Path Type:          Setup (Max at Slow Process Corner)


Slack (MET) :       1.350ns  (arrival time - required time)
Source:             clk_gen_0/inst/plle2_adv_inst/CLKOUT0
    (clock source 'clk_out1_clk_wiz_0'  {rise@0.000ns fall@5.000ns period=10.000ns})
Destination:        clk_sampler/clk_sampled_reg/D
    (rising edge-triggered cell FDRE clocked by clk_out2_clk_wiz_0  {rise@0.625ns fall@1.875ns peri
Path Group:         clk_out2_clk_wiz_0
Path Type:          Hold (Min at Fast Process Corner)
```

This path had a negative hold as resampler use a single register without a logic between. We shifted the sampling 400MHZ clock by 0.625 (90 degrees) to get this successful slack result.

*There is no way to exactly define a relationship between source and destination clock path in the clock path using constraints. **Time report tool can't understand D to Q relationship in the clock sampler module**. Therefore, the destination clock path delay is not showing up in the report properly. We will manually investigate this delay:

To see clock path delay of the clock sampler register, we will use the hold analysis's details:

```
                                                (clock clk_out2_clk_wiz_0 rise edge)
                                                0.625     0.625 r
W5                                              0.000     0.625 r  clk (IN)
```

```
net (fo=0)                                   0.000     0.625    clk_gen_0/inst/clk_in1
W5                  IBUF (Prop_ibuf_I_O)       1.388     2.013 r  clk_gen_0/inst/clkin1_ibufg/O
net (fo=1, routed)                           1.181     3.194    clk_gen_0/inst/clk_in1_clk_wiz_0
PLLE2_ADV_X1Y0      PLLE2_ADV (Prop_plle2_adv_CLKIN1_CLKOUT1)
                                            -7.750    -4.556 r  clk_gen_0/inst/plle2_adv_inst/CLK
net (fo=1, routed)                           1.576    -2.979    clk_gen_0/inst/clk_out2_clk_wiz_0
BUFGCTRL_X0Y2       BUFG (Prop_bufg_I_O)       0.091    -2.888 r  clk_gen_0/inst/clkout2_buf/O
net (fo=1, routed)                           1.512    -1.376    clk_sampler/clk_400_mhz
SLICE_X5Y13         FDRE                                         r  clk_sampler/clk_sampled_reg/C
                   clock pessimism          -0.590    -1.966
                   clock uncertainty        -0.188    -2.154
```

This is the clock path of the clock sampler register. At t = -2.154 450MHZ clock enters.

Now we should get the timing report of the register that we inserted delay for which is data_out register:

```
Slack (MET) :             3.846ns  (required time - arrival time)
Source:                 data_reg_reg[6]/C
                          (rising edge-triggered cell FDRE clocked by clk_out1_clk_wiz_0  {rise@
Destination:            data_out_reg[3]/D
                          (falling edge-triggered cell FDRE clocked by shifted_100  {rise@2.500n
Path Group:             shifted_100
Path Type:              Setup (Max at Slow Process Corner)
Requirement:            7.500ns  (shifted_100 fall@7.500ns - clk_out1_clk_wiz_0 rise@0.000ns)
Data Path Delay:        6.539ns  (logic 4.297ns (65.713%)  route 2.242ns (34.287%))
Logic Levels:           1  (DSP48E1=1)
Clock Path Skew:        2.886ns (DCD - SCD + CPR)
  Destination Clock Delay (DCD):    0.504ns = ( 8.004 - 7.500 )
  Source Clock Delay      (SCD):    -2.382ns
  Clock Pessimism Removal (CPR):    0.000ns
Clock Uncertainty:      0.084ns  ((TSJ^2 + TIJ^2 + DJ^2)^1/2) / 2 + PE
  Total System Jitter     (TSJ):    0.071ns
  Total Input Jitter      (TIJ):    0.100ns
  Discrete Jitter          (DJ):    0.116ns
  Phase Error              (PE):    0.000ns

  Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
-------------------------------------------------------------------    -------------------
                      (clock clk_out1_clk_wiz_0 rise edge)
                                                   0.000     0.000 r
  W5                                               0.000     0.000 r  clk (IN)
                      net (fo=0)                   0.000     0.000    clk_gen_0/inst/clk_in1
  W5                  IBUF (Prop_ibuf_I_O)         1.458     1.458 r  clk_gen_0/inst/clkin1_ibu
                      net (fo=1, routed)           1.253     2.711    clk_gen_0/inst/clk_in1_cl
  PLLE2_ADV_X1Y0      PLLE2_ADV (Prop_plle2_adv_CLKIN1_CLKOUT0)
                                                  -8.482    -5.770 r  clk_gen_0/inst/plle2_adv_
                      net (fo=1, routed)           1.655    -4.115    clk_gen_0/inst/clk_out1_c
  BUFGCTRL_X0Y0       BUFG (Prop_bufg_I_O)         0.096    -4.019 r  clk_gen_0/inst/clkout1_bu
                      net (fo=17, routed)          1.637    -2.382    clk_100_mhz
  SLICE_X0Y7          FDRE                                          r  data_reg_reg[6]/C
-------------------------------------------------------------------    -------------------
  SLICE_X0Y7          FDRE (Prop_fdre_C_Q)         0.456    -1.926 r  data_reg_reg[6]/Q
                      net (fo=1, routed)           0.987    -0.938    dsp_mul_inst/a[6]
  DSP48_X0Y4          DSP48E1 (Prop_dsp48e1_A[6]_P[3])
                                                   3.841     2.903 r  dsp_mul_inst/product__0/P
                      net (fo=1, routed)           1.255     4.157    out_mul[3]
  SLICE_X0Y16         FDRE                                          r  data_out_reg[3]/D
-------------------------------------------------------------------    -------------------

                      (clock shifted_100 fall edge)
                                                   7.500     7.500 f
  SLICE_X5Y13         FDRE                         0.000     7.500 f  clk_sampler/clk_sampled_r
                      net (fo=16, routed)          0.504     8.004    clk_100_shifted_mhz
  SLICE_X0Y16         FDRE                                          r  data_out_reg[3]/C  (IS_IN
                      clock pessimism              0.000     8.004
                      clock uncertainty           -0.084     7.920
  SLICE_X0Y16         FDRE (Setup_fdre_C_D)        0.083     8.003    data_out_reg[3]
-------------------------------------------------------------------
                      required time                          8.003
                      arrival time                          -4.157
-------------------------------------------------------------------
                      slack                                  3.846
```

The second part is "(clock shifted_100 fall edge) 7.500 7.500" assumes the clock rises at t=7.5 at Q of clock_resampling register. Using the information we get from the clock path above we fix this by adding -2.154 to required time as this clock delay path of the clock sampler register.

```
    -------------------------------------------------------------------    -------------------

                      (clock shifted_100 fall edge)
                                                   7.500     7.500 f
                      (Clock Path Delay)          -2.154     5.346
  SLICE_X5Y13         FDRE                         0.000     5.346 f  clk_sampler/clk_sampled_r
                      net (fo=16, routed)          0.504     5.850    clk_100_shifted_mhz
  SLICE_X0Y16         FDRE                                          r  data_out_reg[3]/C  (IS_IN
                      clock pessimism              0.000     5.850
                      clock uncertainty           -0.084     5.766
  SLICE_X0Y16         FDRE (Setup_fdre_C_D)        0.083     5.849    data_out_reg[3]
-------------------------------------------------------------------
                      required time                          5.849
                      arrival time                          -4.157
-------------------------------------------------------------------
                      slack                                  1.692
```

The actual worst slack in this previously problematic path is 1.692ns. The results indicate that the clock shifting is successful. This path had a negative setup slack, by shifting the clock by 2.5ns, the slack is positive as seen.

The constraint line for the shifted clock:

```
create_generated_clock -name shifted_100 -source [get_pins *clk_sampler/clk_sampled_reg/D*] -edges
```

For source section, set the clock to be sampled (to match frequency), for destination section set the clock to be shifted. The edges are rise fall and rise in order as we shift them all 2.5ns we shifted whole signal. Using this property it is possible to change duty cycle of the clock signal.

***This constraint will give a message that the clock is not connected to source by logic. This because the source and generated clock has D to Q relationship.** The timing reports won't include clock path delay for the generated clock. Therefore, we should manually investigate the clock path delay of the generated clock. This is not a problem for same domain analysis, but for cross-domain analysis, this should/must be investigated.

6.4.4 Buffer Insertion

Buffers can be inserted to fix negative slack. The buffer insertion can be done by like below:

```
    wire clk_buf1;
    wire clk_buf2;
    wire clk_buf3;
    wire clk_buf4;
    wire clk_buf5;

    BUF buf1 (.I(clk), .O(clk_buf1)); // First buffer
    BUF buf2 (.I(clk_buf1), .O(clk_buf2)); // Second buffer
    BUF buf3 (.I(clk_buf2), .O(clk_buf3)); // Third buffer
    BUF buf4 (.I(clk_buf3), .O(clk_buf4)); // Forth buffer
    BUF buf5 (.I(clk_buf4), .O(clk_buf_delayed)); // Fifth buffer
```

using clk_buf_delayed wire at the destination register clock, an improvement is done. However, not enough to change the slack positive even though 5 of them are used (The slack on the critical path is -0.709 after 5 LUT based buffer). The buffer itself give 100 picoseconds of delay average. **Most of the delay caused by net delay between slice logic and buffers as they are placed far from each other. However, this is not stable after every implementation, net delay changes.** Adding them each other does not add net delay like the first time. Therefore, only using one is effective.

For little slacks below 0.5ns adding an extra (BUFG, BUF or BUFR, their logic delays changes according to the board) buffer can fix the issue. **However, if negative slack is large enough to add more than one buffer, "not recommended."**

The example we used was not suitable for this solution as the results were not stable.

*Implementation can remove your extra buffers in opt_design stage, try adding `(* keep = "true" *)` before the buffer instantiation. If this is not working, add following line to the constraints file:

```
set_property DONT_TOUCH true [get_nets clk_buf1]
set_property DONT_TOUCH true [get_nets clk_buf2]
set_property DONT_TOUCH true [get_nets clk_buf3]
```

Replace net name with the name of net that goes out of your buffer. This will prevent the tool from removing the buffer.

6.4.5 PLL Insertion

Phase-Locked Loop (PLL) is block in the FPGA that can generate clocks. A PLL can be used to shift the clock. The GUI exists for the PLL configuration. The drawback of this method is that if there are already many multiple clocks in the design, and you might be not wanting to propagate a shifted clock through submodules in order to fix a setup time delay if you do this for every violation many PLL's and a complex module definitions appear.

6.4.6 Conclusion

- A PLL is recommended for a small design with a few clocks and one or two violations in different submodules.
- For a large design with many violations, a clock sampler with PLL is recommended. Example: A PLL to create: 10/1 sampling signal, and you can add sampler module to problematic submodule and shift N/10 of the clock as desired.
- For small violations like 0.5ns, 0.7ns, a buffer is recommended. It is simple solution with one block utilization.

6.5 Multi-Cycle Paths

In this section, we will investigate multi-cycle paths and use case of them.

In designs such as memory interfaces, pipelined designs, complex calculations, state machines that design can not be sure validation of data, setup time and hold time requirements can not be met. For such cases, multi-cycle paths can be used. Multi-cycle paths are paths that can not be validated in a single clock cycle. The paths can be validated in multiple clock cycles.

A multicycle module example is below:

```
module SimpleCounter (
    input wire clk,        // Clock input
    input wire [7:0] in_1,      // Input 1
    input wire [7:0] in_2,      // Input 2
    output reg [15:0] done        // Output
);
    reg [7:0] in_1_reg;       // input 1 buffer
    reg [7:0] in_2_reg;       // input 2 buffer
    reg [1:0] count;          // Counter
    assign wire [15:0] multiplied = in_1_reg * in_1_reg;

    localparam MAX_COUNT = 2'd11; // Define the maximum count value

    // Counter and multiplier signal logic
    always @(posedge clk) begin
        if (count == 0) begin
            in_1_reg <= 4'd0;
            in_2_reg <= 1'b0;
            count <= 2'd01;
        end
        else if (count == 4) begin
            done <= multiplied;
            count <= 2'b00;
        end
        else begin
            count <= count + 1'b1;
        end
    end

endmodule
```

The module is a simple multi cycle multiplier that counts from 1 to 4 for taking result. The counter is started by the start signal. The counter is reset by the reset signal. The counter is stopped when it reaches 4. The done signal is set to 1 when the counter is stopped. We need to clarify that `done` pins changes after 3 clock cycles source changed. Therefore, **when timing analysis is done, the paths from input buffers to done can be seen as timing violated.** If there is a timing violation exists in the path, the tool can ignore it as it is a multi-cycle path. The source won't change for 3 clocks, so output register have plenty of time to stably setup.

*To define a multi-cycle path, the following constraints can be used:

```
    set_multicycle_path <path_multiplier> [-setup|-hold] [-start|-end]
    [-from <startpoints>] [-to <endpoints>] [-through <pins|cells|nets>]

    -path_multiplier: The number of cycles that the path is valid for
    -start: take source's clock as reference
    -end: take destination's clock as reference

    For our case:
    set_multicycle_path 3 -setup -through [get_cells dsp_block]
```

This constraint is important as if we need to check if logic is arriving in the desired number of clock cycles instead of assuming it is done. Also, it prevents from timing violation message. Vivado can detect multi-cycle paths and ignore them in the timing analysis. However, it is recommended to define them manually to prevent any misunderstanding.

6.6 Differential Inputs

In this section, we will investigate differential inputs and their use cases.

6.6.1 Introduction

Differential inputs are used to reduce noise and have a better integrity in the signal. The noise is reduced by taking the difference of the two signals. The noise is common in the two signals. The noise is canceled out by taking the difference of the two signals. It enables us to use low voltage to eliminate power loss and inductance related latency issues. It allows us to have faster transmission rates and longer distances as the side effects of high speed signals like cable impedance, lower precision, vulnerabiliyy to EM interference are reduced. The differential inputs are used in the following cases:

- High speed inputs
- Long distance signal inputs
- External fast clocks

6.6.2 Implementation and Module

The differential inputs are implemented by adding a IBUFDS primitive to the design. The IBUFDS primitive is used to take the differential input and convert it to a single ended signal. The IBUFDS primitive is used as below:

```
IBUFDS #(
    .DIFF_TERM("TRUE"),  // Enable on-chip differential termination
    .IBUF_LOW_PWR("TRUE"), // Low power setting
    .IOSTANDARD("DEFAULT") // Specify the I/O standard
) IBUFDS_inst (
    .I(clk_p),   // Differential positive input
    .IB(clk_n),  // Differential negative input
    .O(clk_out)  // Single-ended output
);
```

The most important variable when instantiating IBUFDS (Input Buffer Differential Signaling) is the DIFF_TERM property. This property enables on-chip differential termination, which is crucial for properly handling differential signals.

Differential termination is used to terminate the differential signal by adding internal resistors to the signal path. This termination matches the impedance of the transmission line to prevent reflections and ensure signal integrity. By properly matching the impedance, differential termination helps in maintaining signal quality and reducing noise. This is especially important for high-speed data transmission where signal integrity is critical.

The other properties of IBUFDS are usually set to default values, but DIFF_TERM is essential for enabling the internal termination that aligns with the impedance of the transmission line and improves overall signal performance.

However, you should not set it true if you added external termination resistors.

The module to run at Basys 3 board's max speed 463MHZ as below:

```
module AModule(
    input wire clk_p, // Clock signal
    input wire clk_n, // Inverted clock signal
    input wire [7:0] data_in, // Data input
    input wire [7:0] data_in_2, // Data input 2
    output reg [8:0] data_out // Data output
    );

    wire clk;                // Buffered clock signal (non-delayed)

    IBUFDS clkbuf (.I(clk_p), .IB(clk_n), .O(clk) ); // Differential to single-ended clock

    always @(posedge clk) begin
        data_out <= data_in + data_in_2;
    end

endmodule
```

The module is a simple adder that adds two 8-bit inputs. The clock is buffered by the IBUFDS primitive. Other than the IBUFDS next most important thing is the constraints file for the differential inputs.

You should check pinout of the specific FPGA model you have to select coupled pins, for XC7A35T-1CPG236C model, pinout can be found here:

https://www.physics.umd.edu/hep/drew/495/xc7a35tcpg236pkg.txt. You can understand the coupled pins like below:

```
Pin  Pin Name                    Memory Byte Group  Bank  VCCAUX Group  I/O Type  No-Connect
U3   IO_L9P_T1_DQS_34            1                  34    NA            HR        NA
U2   IO_L9N_T1_DQS_34            1                  34    NA            HR        NA
```

Look at Pin Name column the names will be identical except P and N suffixes after IO_[PINNUMBER]. In the constraints file, a master clock for p signal and generated clock for the output of IBUFDS should be defined. The constraints file is below:

```
set_property PACKAGE_PIN L17 [get_ports clk_p ]
set_property IOSTANDARD LVDS_25 [get_ports clk_p ]
set_property PACKAGE_PIN K17 [get_ports clk_n ]
set_property IOSTANDARD LVDS_25 [get_ports clk_n ]

create_clock -period 2.225 -name clk_p -waveform {0.000 1.1125} [get_ports clk_p]
create_generated_clock -name clk -source [get_ports clk_p] -divide_by 1 [get_pins clkbuf/O]


set_input_delay -clock [get_clocks clk] -max 0 [get_ports *data_in*]
set_input_delay -clock [get_clocks clk] -min 0.2 [get_ports *data_in*]


set_input_delay -clock [get_clocks clk] -max 0 [get_ports *data_in_2*]
set_input_delay -clock [get_clocks clk] -min 0.2 [get_ports *data_in_2*]
```

Another important information gathered from this code's implementation is that the tool can have difficulty to route the design. **The input buffer connected to clock trans buffers (BUFG) must be on the same side of the FPGA in order comply design rules.** If the implementation gives such placing error you can use the following procedure:

- Open the synthesied design
- On the device open "find" (CTRL-F) tab
- The message code will include both input buffer's tile name and BUFG's site name ex:"X0Y0"
- Search these, you will see that one is on the upper side and the other is on the lower side.
- Chose another couple ports by listing the ports and make sure that they are on the same side with BUFG in the message.
- Fix constraints and run implementation again.

Example placement on the same side is as below:

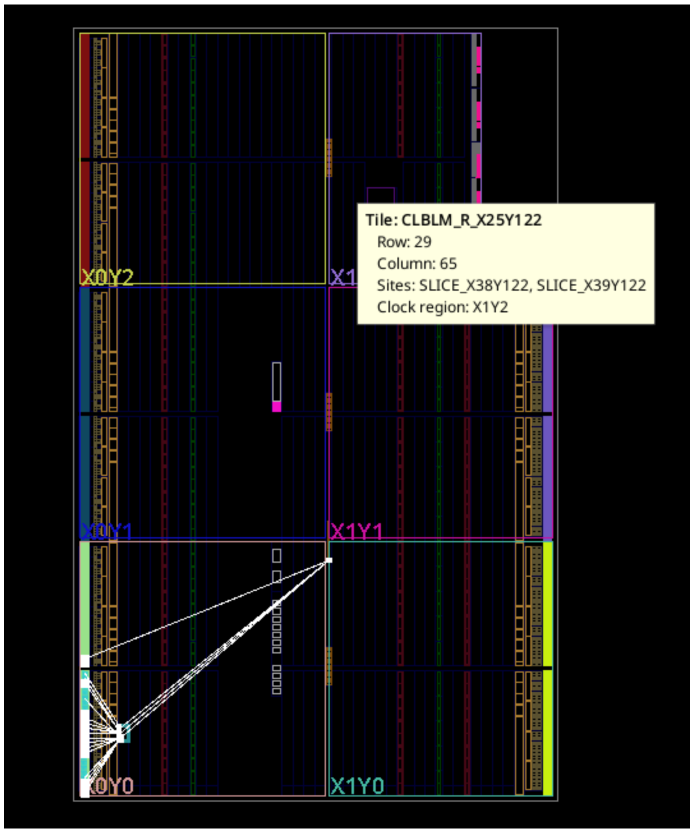


Figure 6.6.2.1: Design Placed

6.6.3 Results and Conclusion

The general timing summary of the design is below:

```
------------------------------------------------------------------------
| Design Timing Summary
| ----------------------
------------------------------------------------------------------------

WNS(ns)      TNS(ns)      WHS(ns)      THS(ns)     WPWS(ns)     TPWS(ns)
-------      -------      -------      -------     --------     --------
0.171        0.000        0.130        0.000        0.070        0.000
```

The results indicate that the design is successful. The worst slack is 0.171ns. The WPWS is 0.070ns, and it is on a BUFG, which means that we are running at the maximum clock speed that clock distribution buffer can handle.