BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

EEE 468 (December 2024)

VLSI Laboratory

Final Project Report

Section: G1 Group: 06

12 Bit Synchronous Counter

Course Instructors:

Nafis Sadik (Lecturer)
Rafid Hassan Palash (PT)

Signature of Instructor: _____

Academic Honesty Statement:

IMPORTANT! Please carefully read and sign the Academic Honesty Statement, below. Type the student ID and name, and put your signature. *You will not receive credit for this project experiment unless this statement is signed in the presence of your lab instructor.*

"In signing this statement, We hereby certify that the work on this project is our own and that we have not copied the work of any other students (past or present), and cited all relevant sources while completing this project. We understand that if we fail to honor this agreement, We will each receive a score of ZERO for this project and be subject to failure of this course."

Signature: Bihan
Full Name: **Subhan Zawad Bihan**
Student ID: **1906037**

Signature: Nilotpaul
Full Name: Nilotpaul Kundu Dhrubo
Student ID: 1906060

Signature: Arghya
Full Name: Archishman Sarkar Arghya
Student ID: 1906051

Signature: Mahir
Full Name: Abrar Md. Mahir
Student ID: 1906039

1. Abstract

In this project we design and implement a **12-bit synchronous counter** with a parallel load feature. The counter is controlled by a clock signal and incorporates a synchronous reset, enabling prioritized reset and load operations. The design supports two primary modes of operation: **up-counting** and **down-counting**, which are controlled by an input signal. A parallel load feature allows the counter to load a user-defined 12-bit value when activated. We also verify the code using directed and layered verification. Finally, we synthesize the circuit and design the layout in Innovus. We also optimize the circuit to increase its cost effectiveness.

2. Introduction

Counters are fundamental building blocks in digital systems, commonly used for applications such as timing, event counting, frequency division, and memory address generation. Among various types of counters, **synchronous counters** are preferred for their ability to mitigate clock skew and ensure synchronized operations across all flip-flops.

This report describes a **12-bit synchronous counter** with an additional parallel load feature. The counter operates under a clock signal and prioritizes synchronous reset and load functionalities.

3. Design

3.1. RTL Block Diagram

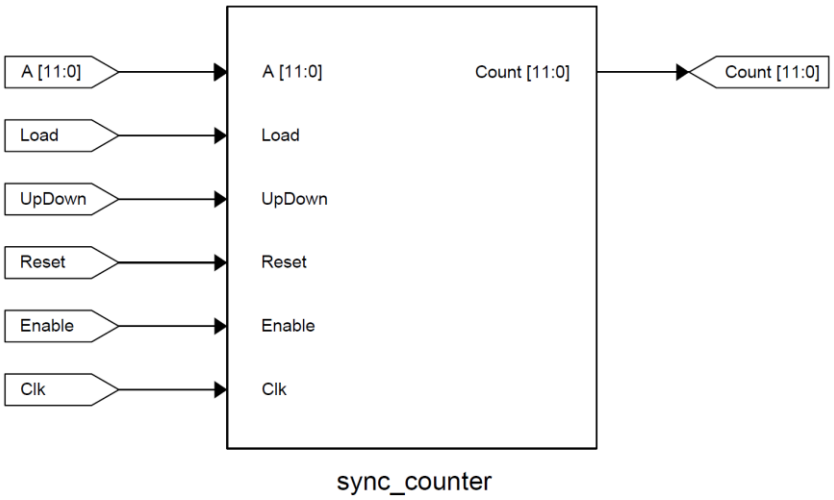


Figure 1: RTL module top view

4. Implementation

4.1. RTL code

```
module sync_counter (
    input wire [11:0] A,          // Number to be loaded
    input wire Load,              // Parallel load enable
    input wire UpDown,            // Up/Down control
    input wire Reset,             // Synchronous reset
    input wire Enable,            // Enable signal
    input wire Clk,               // Clock signal
    output reg [11:0] Count       // 12-bit counter output
);

    // Always block triggered on the rising edge of the clock signal
    always @(posedge Clk) begin
      if (Reset) begin
        // Reset has the highest priority
        Count <= 12'b0;
      end
      else if (Load) begin
        // Load A into Count when Load is high
        Count <= A;
      end
      else if (Enable) begin    // If Enable is high, allow Up/Down counting
        Count <= UpDown ? Count + 1 : Count - 1;    // Up-count if UpDown = 1,
Down-count if UpDown = 0
      end
      // If Enable is low, Count remains unchanged
    end

endmodule
```

Figure 2: RTL code

The sync_counter module takes a 12 bit input defined by the bus A[11:0]. If the Reset signal is TRUE, output Count is set to 12b'0. If Reset is FALSE, and Load is TRUE, value of A is passed to Count. If both Reset and Load is FALSE, and Enable is TRUE, Count is incremented by 1, if UpDown is TRUE, and decremented by 1 if UpDown is FALSE. The input and output signals are synchronized with the Clk signal.

4.2. Verification

4.2.1 Directed Verification

```
`timescale 1ns / 1ps

module directed_tb();
    reg [11:0] A;           // Number to be loaded
    reg Load;               // Parallel load enable
    reg UpDown;             // Up/Down control
    reg Reset;              // Synchronous reset
    reg Enable;             // Enable signal
    reg Clk;                // Clock signal
    wire [11:0] Count;      // 12-bit counter output

    initial begin
        Clk = 1;
        forever #5 Clk = ~Clk;
    end

    sync_counter DUT (
        .A(A),
        .Load(Load),
        .UpDown(UpDown),
        .Reset(Reset),
        .Enable(Enable),
        .Clk(Clk),
        .Count(Count)
    );
```

The testbench module has a 1ns time unit and 1ps precision unit. All the input and output signals along with a 10ns Clock cycle is declared in the beginning of the tb code. A DUT is also declared which is connected to the RTL module.

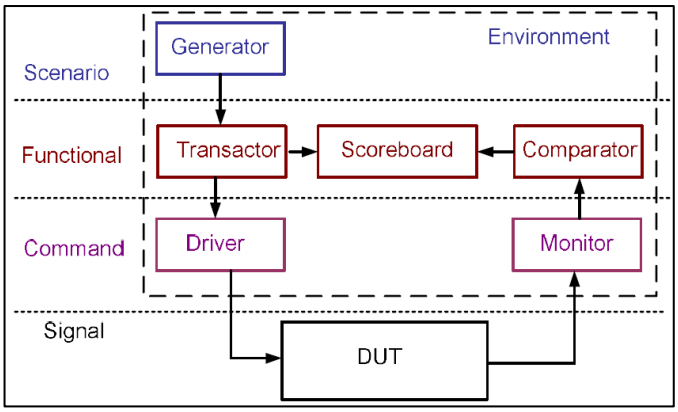
```
    initial begin
      A = 12'b0;
      Load = 0;
      UpDown = 1;
      Reset = 0;
      Enable = 0;

      // Apply reset and check Count
      Reset = 1; #10;
      Reset = 0;
      $display("Test Reset: Count = %d (Expected: 0)\n", Count);

      // Test minimum wrap-around
      Enable = 1;
      UpDown = 0;
      #10;  // Wait for 1 clock cycle
      $display("Test minimum wrap-around: Count = %d (Expected: 4095)\n", Count); // Should increment by 4 counts

      // Test maximum wrap-around
      UpDown = 1;
      #10;  // Wait for 1 clock cycle
      $display("Test maximum wrap-around: Count = %d (Expected: 0)\n", Count); // Should increment by 4 counts

      // Load a value into Count and check
      A = 100;
      Load = 1;     // Enable still 1, but that shouldn't matter
      #10;
      Load = 0;
      $display("Test Load: Count = %d (Expected: 100)\n", Count);

      // Test Up-Counting for few cycles
      #10; $display("Test Up-counting: Count = %d (Expected: 101)", Count);
      #10; $display("Test Up-counting: Count = %d (Expected: 102)", Count);
      #10; $display("Test Up-counting: Count = %d (Expected: 103)", Count);

      // Test Enable. UpDown still 1, but count should stay same
      Enable = 0;
      #10; $display("Test Enable: Count = %d (Expected: 103)\n", Count);

      // Load another value into Count and check
      A = 2024;
      Load = 1;
      #10;
      Load = 0;
      $display("Test Load: Count = %d (Expected: 2024)\n", Count);
```

We use directed verification to check the RTL for various testcases. Such as – Applying reset, Minimum and Maximum wrap-around, Loading a value, upcounting and downcounting.

```
      // Test Down-Counting for few cycles
      Enable = 1;
      UpDown = 0;
      #10; $display("Test Down-counting: Count = %d (Expected: 2023)", Count);
      #10; $display("Test Down-counting: Count = %d (Expected: 2022)", Count);
      #10; $display("Test Down-counting: Count = %d (Expected: 2021)", Count);

      // Test Enable. UpDown still 0, but count should stay same
      Enable = 0;
      #10; $display("Test Enable: Count = %d (Expected: 2021)\n", Count);

      // Apply reset again and check Count
      Reset = 1; #10;
      Reset = 0;
      $display("Test Reset: Count = %d (Expected: 0)\n", Count);
      $finish;
    end

endmodule
```

We also check whether the Enable and Reset signals are functioning correctly or not. Both the testbench and counter module outputs are shown using $display() command for debugging purpose.

4.2.2 Layered Verification



4.2.2.1 Code

transaction.sv

```
class transaction;
  rand bit [11:0] A;
  rand bit Load;
  rand bit UpDown;
  rand bit Reset;
  rand bit Enable;
  bit [11:0] Count;

endclass:transaction
```

Transaction class encapsulates the ports of the module. Inputs are set to “rand” so that the built-in “randomize” class method can operate on them.

generator.sv

```
`include "transaction.sv"

class generator;
  mailbox gen2driv;
  transaction g_trans, custom_trans;

  function new(mailbox gen2driv);
    this.gen2driv=gen2driv;
  endfunction

  task main(input int count);
    repeat(count) begin
      g_trans=new();
      g_trans=new custom_trans;
      assert(g_trans.randomize());
      gen2driv.put(g_trans);
    end
  endtask:main

endclass:generator
```

Generator class creates a transaction-type object, randomizes the inputs (defined as rand), and puts the data into gen2driv mailbox.

interface.sv

```
interface counter_if(input clk);
  logic [11:0] A, Count;
  logic Load, UpDown, Reset, Enable;

  clocking driver_cb @(negedge clk);
    default input #1 output #1;
    output A, Load, UpDown, Reset, Enable;
  endclocking

  clocking mon_cb @(negedge clk);
    default input #1 output #1;
    input A, Load, UpDown, Reset, Enable, Count;
  endclocking

  modport DRIVER (clocking driver_cb, input clk);
  modport MONITOR (clocking mon_cb, input clk);

endinterface
```

Interface defines the direction of ports as seen by driver and monitor. It also specifies the sampling instance with respect to the testbench clock.

driver.sv

```
class driver;
  mailbox gen2driv, driv2sb;
  virtual counter_if.DRIVER counterif;
  transaction d_trans;
  event driven;

  function new(mailbox gen2driv, driv2sb , virtual counter_if.DRIVER
counterif, event driven);
    this.gen2driv=gen2driv;
    this.counterif=counterif;
    this.driven=driven;
    this.driv2sb=driv2sb;
  endfunction


  task main(input int count);
    repeat(count) begin
      d_trans=new();
      gen2driv.get(d_trans);

      @(counterif.driver_cb);
      counterif.driver_cb.A <= d_trans.A;
      counterif.driver_cb.Load <= d_trans.Load;
      counterif.driver_cb.Enable <= d_trans.Enable;
      counterif.driver_cb.Reset <= d_trans.Reset;
      counterif.driver_cb.UpDown <= d_trans.UpDown;
      driv2sb.put(d_trans);
      -> driven;
    end

  endtask:main

endclass:driver
```

Driver makes use of the DRIVER modport of counterif interface. It fetches data from the gen2driv mailbox, waits for trigger of driver_cb, then sends the data along counterif (leading to DUT).

monitor.sv

```
class monitor;
  mailbox mon2sb;
  virtual counter_if.MONITOR counterif;
  transaction m_trans;
  event driven;

  function new(mailbox mon2sb, virtual counter_if.MONITOR counterif, event driven);
    this.mon2sb=mon2sb;
    this.counterif=counterif;
    this.driven=driven;
  endfunction

  task main(input int count);
    @(driven);
    @(counterif.mon_cb);
    repeat(count) begin
      m_trans=new();    // It seems we only care about DUT output Count from Monitor
      @(posedge counterif.clk);
      m_trans.Count = counterif.mon_cb.Count;
      mon2sb.put(m_trans);
    end
  endtask:main


endclass:monitor
```

Monitor waits for both trigger of 'driven' event (driver sends data to DUT) and counterif.mon_cb. It then collects data of all ports from DUT, and puts it into mon2sb mailbox

coverage.sv

```
class counter_coverage;
  // Variables to sample
  bit [11:0] A, Count;
  bit Load, Reset, Enable, UpDown;

  covergroup cg;
    option.per_instance = 1;
    option.comment = "Counter coverage";

    // Cover basic input values
    A_cp: coverpoint A {
      bins min_edge = {12'h000};
      bins max_edge = {12'hFFF};
      bins others[4] = {[12'h001:12'hFFE]};  // Split other values into 4 bins
    }

    Load_cp: coverpoint Load;
    Reset_cp: coverpoint Reset;
    Enable_cp: coverpoint Enable;
    UpDown_cp: coverpoint UpDown;


    // Cross coverage
    LoadA_cross: cross Load_cp, A_cp {
      option.weight = 2;  // Important to verify Load with different A values
    }

  endgroup
  function new();
    cg = new;
    cg.start();  // Explicitly start coverage collection
  endfunction


  function void sample(transaction trans);
    this.A = trans.A;
    this.Count = trans.Count;
    this.Load = trans.Load;
    this.Reset = trans.Reset;
    this.Enable = trans.Enable;
    this.UpDown = trans.UpDown;
    cg.sample();
  endfunction
endclass
```

Coverage class defines the bins and coverpoints to evaluate coverage. Edge/corner cases are in separate bins of their own. The cross coverage is given double weight to raise its importance.

Scoreboard.sv

```
`include "coverage.sv"

class scoreboard;
  mailbox driv2sb;
  mailbox mon2sb;

  transaction d_trans;
  transaction m_trans;

  event driven;

  int pass_count, fail_count;
  bit test_result;  // 1 for pass, 0 for fail

  // Add state tracking
  bit [11:0] expected_count;
  counter_coverage coverage;

  function new(mailbox driv2sb, mon2sb);
    this.driv2sb = driv2sb;
    this.mon2sb = mon2sb;
    this.pass_count = 0;
    this.fail_count = 0;
    this.expected_count = 0; // Initialize counter
    coverage = new();        // Initialize coverage
  endfunction

  task main(input int count);
    $display("------------------Scoreboard Test Starts--------------------");
    repeat(count) begin
      d_trans = new();
      m_trans = new();

      // Get both transactions
      driv2sb.get(d_trans);
      mon2sb.get(m_trans);

      // Calculate expected value based on inputs
      if (d_trans.Reset) begin
        expected_count = 12'b0;
      end
      else if (d_trans.Load) begin
        expected_count = d_trans.A;
      end

       else if (d_trans.Enable) begin
         if (d_trans.UpDown) begin  // Count up
           if (expected_count == 12'hFFF)
             expected_count = 12'b0;
           else
             expected_count = expected_count + 1'b1;
         end
         else begin  // Count down
           if (expected_count == 12'b0)
             expected_count = 12'hFFF;
           else
             expected_count = expected_count - 1'b1;
         end
       end

       // Compare with actual value
       test_result = (m_trans.Count == expected_count);

       // Sample coverage after each transaction
       coverage.sample(d_trans);
```

```
      if(test_result) begin
        pass_count++;
        $display("Passed : A=0x%h, Reset=%b, Load=%b, Enable=%b, UpDown=%b, Expected
Count=0x%h, Actual Count=0x%h",
                 d_trans.A, d_trans.Reset, d_trans.Load, d_trans.Enable, d_trans.UpDown,
expected_count, m_trans.Count);
      end else begin
        fail_count++;
        $display("Failed : A=0x%h, Reset=%b, Load=%b, Enable=%b, UpDown=%b, Expected
Count=0x%h, Actual Count=0x%h",
                 d_trans.A, d_trans.Reset, d_trans.Load, d_trans.Enable, d_trans.UpDown,
expected_count, m_trans.Count);
      end
    end
    $display("------------------Scoreboard Test Ends--------------------");
    $display("Total Passes: %0d, Total Fails: %0d", pass_count, fail_count);

    $display("------------------Coverage Summary--------------------");
    $display("Coverage: %.2f%%", coverage.cg.get_coverage());
  endtask

endclass
```

Scoreboard receives data from both driver and monitor. It calculates the expected output (from driver data) and compares with actual output (reported by monitor). At the end, it displays the total passes and fails, plus the coverage.

environment.sv

```
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "scoreboard.sv"

class environment;
  mailbox gen2driv;
  mailbox driv2sb;
  mailbox mon2sb;

  generator gen;
  driver drv;
  monitor mon;
  scoreboard scb;

  event driven;

  virtual counter_if counterif;

  function new(virtual counter_if counterif);
    this.counterif = counterif;
    gen2driv = new();
    driv2sb = new();
    mon2sb = new();

    gen = new(gen2driv);
    drv = new(gen2driv, driv2sb, counterif.DRIVER, driven);
    mon = new(mon2sb, counterif.MONITOR, driven);
    scb = new(driv2sb, mon2sb);
  endfunction
```

```
    task main(input int count);
      fork
        gen.main(count);
        drv.main(count);
        mon.main(count);
        scb.main(count);
      join
    endtask

    function int get_pass_count();
      return scb.pass_count;
    endfunction

    function int get_fail_count();
      return scb.fail_count;
    endfunction
  endclass
```

Environment class encapsulates the entire testbench – including generator, driver, monitor, scoreboard.

testcases.sv

```
`include "environment.sv"

program test(input int count, counter_if counterif, output int pass_count, output int
fail_count, output bit test_done);
  environment env;

  class testcase01 extends transaction;
    constraint c_A {
      A dist {
        12'h000 :/ 20,  // 20% probability
        12'hFFF :/ 20,  // 20% probability
        [12'h001:12'hFFE] :/ 60  // 60% probability for any other random value
      };
    }

    constraint c_edge {
      Count == 12'hFFF -> UpDown dist { 0 := 10, 1 := 90 };
      Count == 12'h000 -> UpDown dist { 0 := 90, 1 := 10 };
    }

    constraint c_others {
      Reset dist {
        0 := 96, 1 := 4
      };
      Load dist {
        0 := 94, 1 := 6
      };
      Enable dist {
        0 := 15, 1 := 85
      };
    }

  endclass

  initial begin
    testcase01 testcase01handle;
    testcase01handle = new();

    env = new(counterif);
    env.gen.custom_trans = testcase01handle;
    env.main(count);

    pass_count = env.get_pass_count();
    fail_count = env.get_fail_count();
    test_done = 1'b1;
  end

endprogram:test
```

Testcases places additional restrictions/constraints on DUT inputs for suitable testing. Edge-cases occur to at 20% probability each (high importance compared to any other random value). Reset and load should occur at low probability (here 4% and 6% respectively). Enable

should be on most of the time (here 85%).

testbench.sv

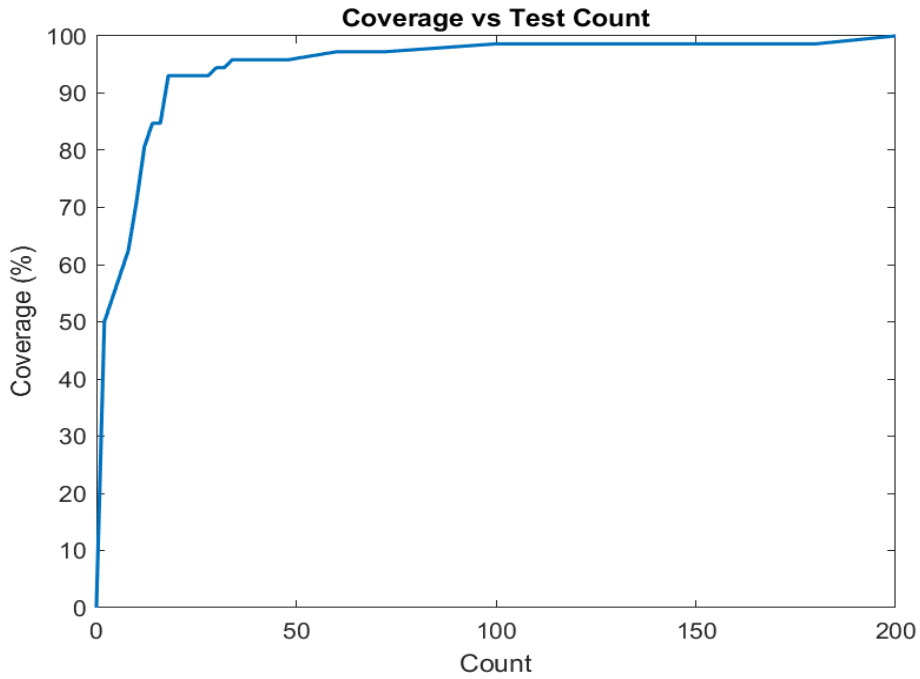
```
`include "testcases.sv"
`include "interface.sv"

module testbench;
  bit clk;
  int pass_count, fail_count;
  bit test_done;

  always #5 clk = ~clk;

  int count = 200;  // Increased test count for better coverage
  counter_if counterif(clk);

  test test01(count, counterif, pass_count, fail_count, test_done);

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
  end

  sync_counter DUT (
    .A(counterif.A),
    .Load(counterif.Load),
    .UpDown(counterif.UpDown),
    .Reset(counterif.Reset),
    .Enable(counterif.Enable),
    .Clk(clk),
    .Count(counterif.Count)
  );

  initial begin
    #4;
    counterif.Reset = 1;
    #6;
    counterif.Reset = 0;
  end

endmodule
```

The top module of the layered tesbench. Instantiates the DUT, runs the test program, and sets reset to high initially (so that we get proper results).

4.2.2.2 Results

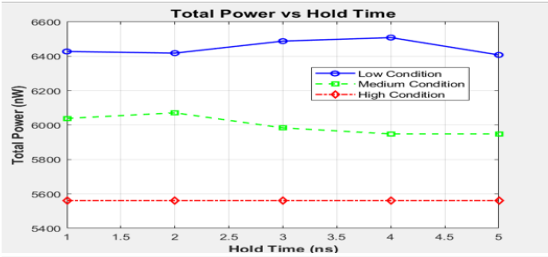
```
Chronologic VCS simulator copyright 1991-2023
Contains Synopsys proprietary information.
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Dec 15 23:57 2024
------------------Scoreboard Test Starts--------------------
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xfff, Actual Count=0xfff
Passed : A=0xfff, Reset=0, Load=0, Enable=0, UpDown=0, Expected Count=0xfff, Actual Count=0xfff
Passed : A=0xd6e, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffe, Actual Count=0xffe
Passed : A=0xfff, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffd, Actual Count=0xffd
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffc, Actual Count=0xffc
    ...
    ...
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xfff, Actual Count=0xfff
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffe, Actual Count=0xffe
Passed : A=0x6ab, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffd, Actual Count=0xffd
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffc, Actual Count=0xffc
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=1, Expected Count=0xffd, Actual Count=0xffd
Passed : A=0x000, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffc, Actual Count=0xffc
Passed : A=0x224, Reset=0, Load=0, Enable=1, UpDown=1, Expected Count=0xffd, Actual Count=0xffd
Passed : A=0x8e4, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffc, Actual Count=0xffc
Passed : A=0xfff, Reset=0, Load=0, Enable=1, UpDown=0, Expected Count=0xffb, Actual Count=0xffb
------------------Scoreboard Test Ends--------------------
Total Passes: 200, Total Fails: 0
------------------Coverage Summary--------------------
Coverage: 100.00%
$finish at simulation time                2015
           V C S   S i m u l a t i o n   R e p o r t
Time: 2015 ns
CPU Time:      0.400 seconds;       Data structure size:   0.0Mb
Sun Dec 15 23:57:30 2024
```

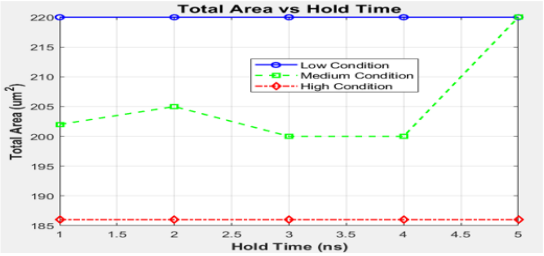
We have also plotted coverage against test count by varying the count

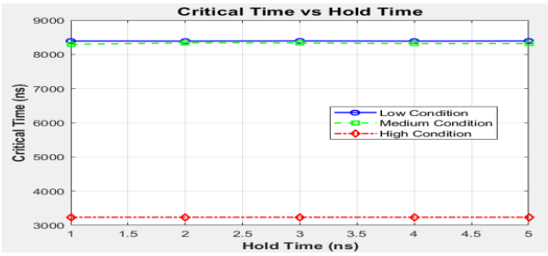


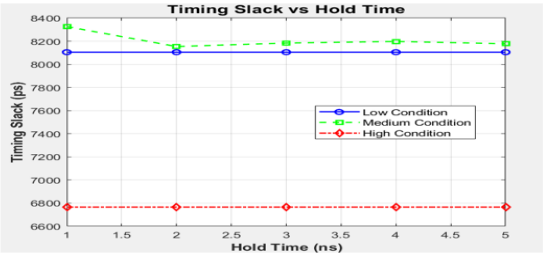
4.3. Metrics vs Different Parameters:

4.3.1 Metrics vs Hold Time

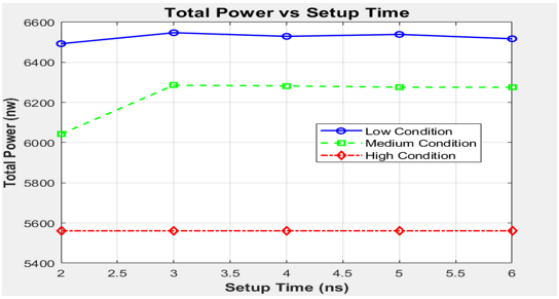


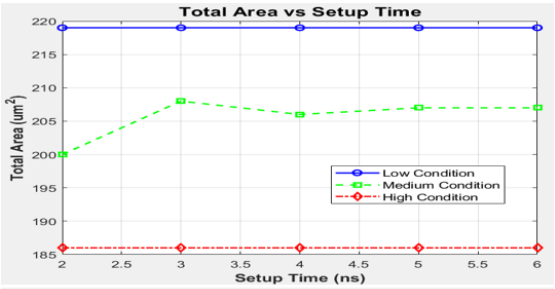


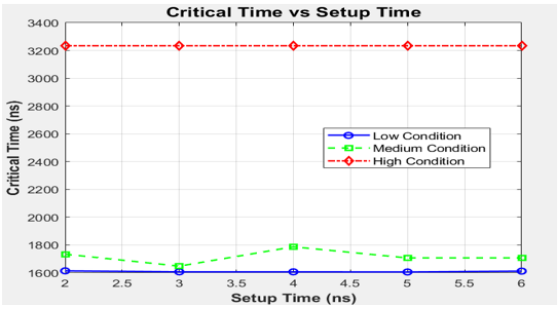


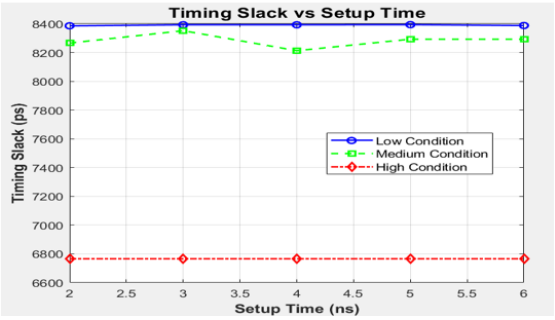


4.3.2 Metrics vs Setup Time

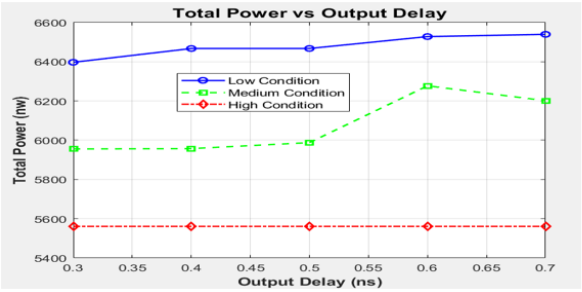


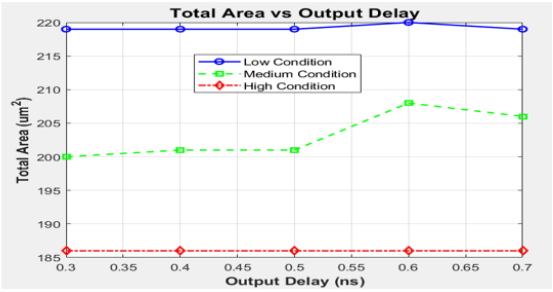


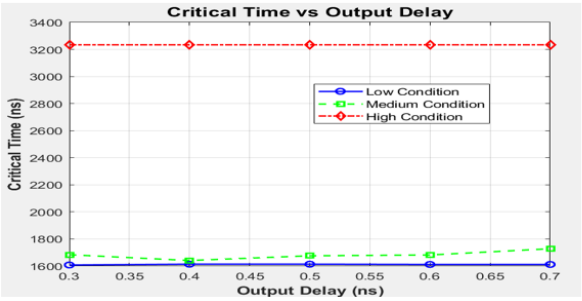


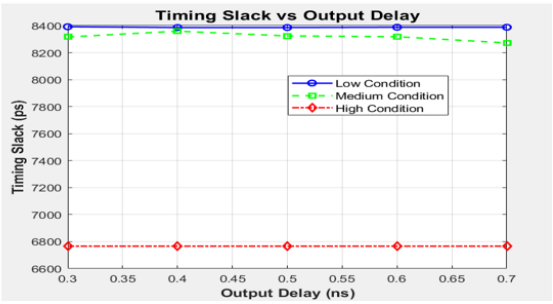


4.3.3 Metrics vs Output Delay

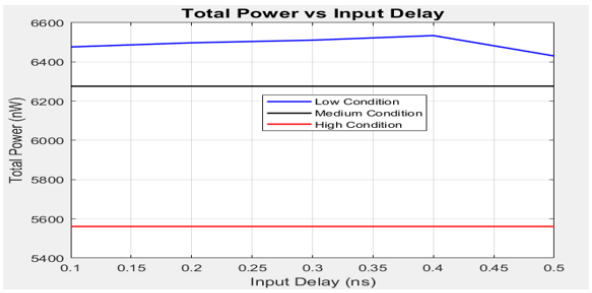


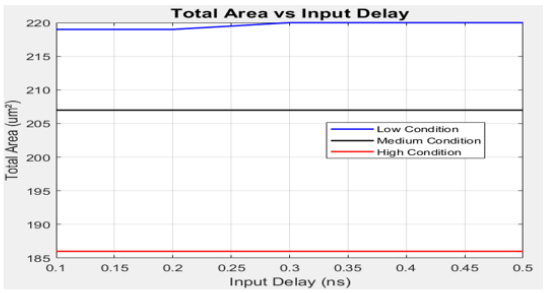


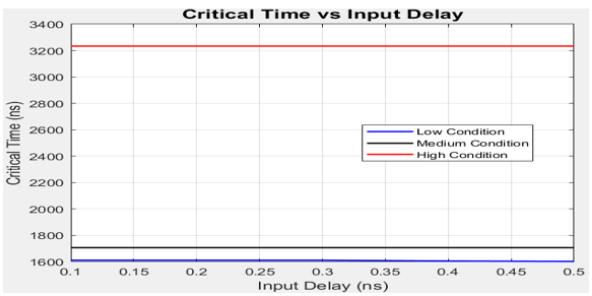


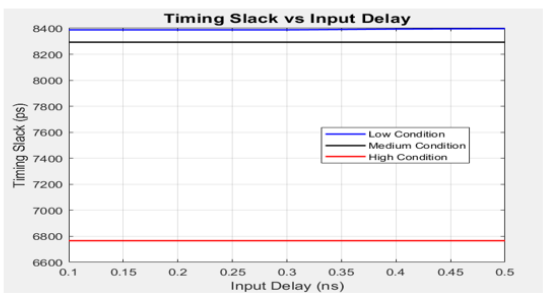


4.3.4 Metrics vs Input Delay









4.3.5 Conditions for Optimisation:

Minimum Total Area and Minimum Total Power
As long as timing slack is positive, critical time doesn't matter

Using these Conditions the optimized value for 3 different conditions are given below:

	Low	Medium	High
Hold Time(ns)	5	4	1
Setup Time(ns)	2	2	2
Input Delay(ns)	0.5	0.1	0.1
Output Delay(ns)	0.3	0.3	0.3
Clock Period(ns)	10	10	10

We have taken clock period 10 ns for every condition. For low clock period circuit became faster but total power became very high. 10 ns is a good compromise between latency and consumed power.

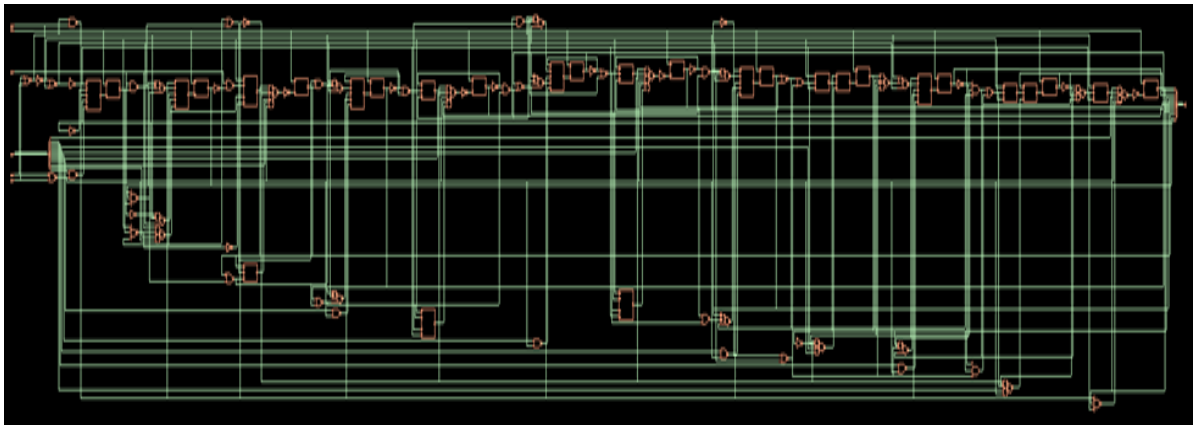
Among the 3 different conditions high condition gives the best results. So, we synthesized our circuit using the high condition values. And our final synthesized results are given below:

```
                    Leakage   Internal    Net        Dynamic          Total
                                                  (Internal+Net) (Leakage+Dynamic)
  Instance   Cells Power(nW) Power(nW) Power(nW)     Power(nW)       Power(nW)
-------------------------------------------------------------------------------
sync_counter   109     4.625  4430.371  1133.599       5563.970         5568.594
```

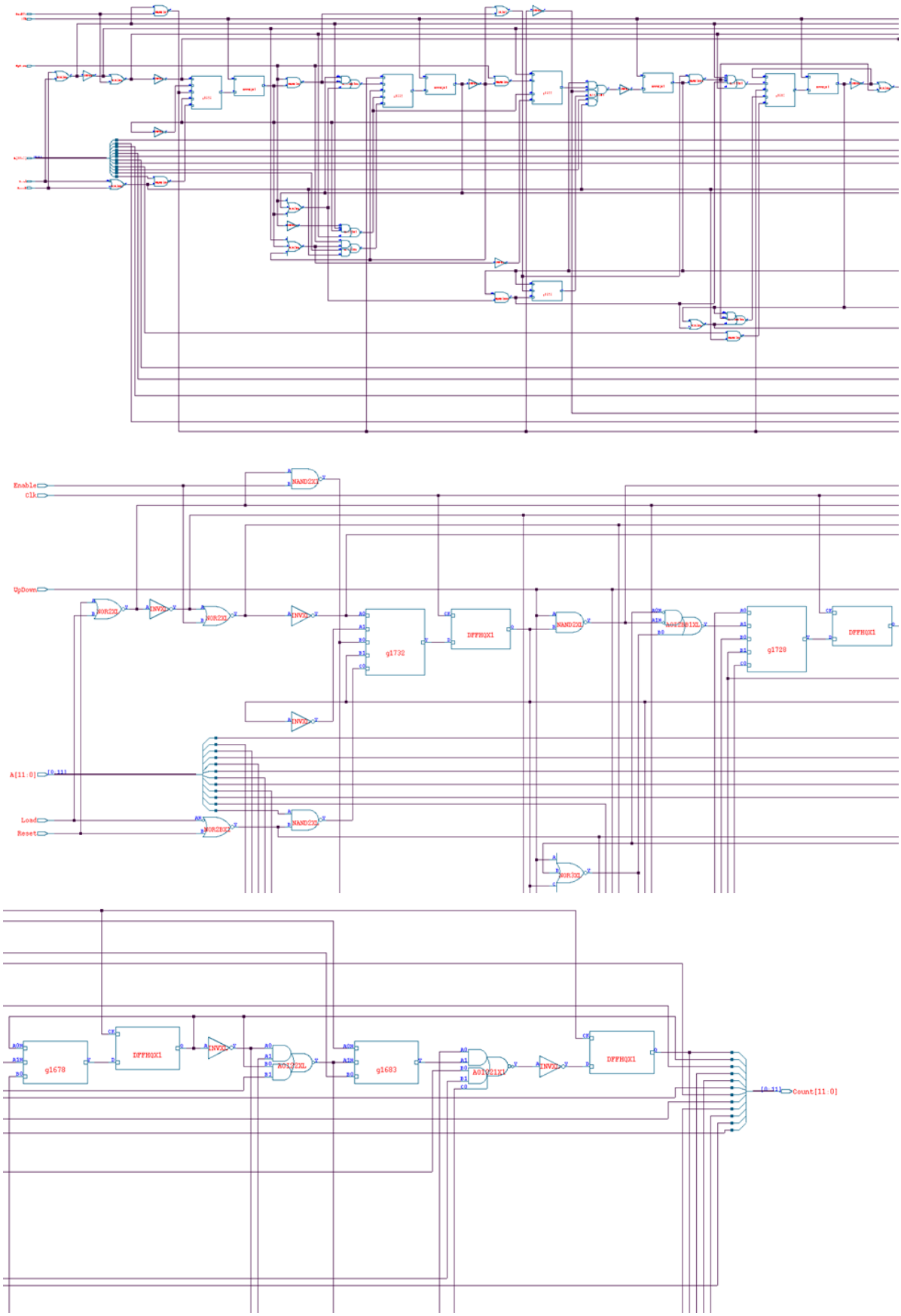
```
  Instance   Module  Cells  Cell Area  Net Area   Total Area  Wireload
-------------------------------------------------------------------------
sync_counter           109        203         0          203     <none> (D)
```

```
Count_reg[10]/CK      setup                       0  +119    1774 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock func_clk)      capture                            10000 R
-------------------------------------------------------------------------
Cost Group   : 'func_clk' (path_group 'func_clk')
Timing slack :    8226ps
```

4.3.6 Final Synthesized Circuit:



4.3.7 Zoomed Version of Different Portion of the Circuit:



4.4. Physical Design

Steps to make the IC or we can say placing the logic elements in the silicon wafer

4.4.1 Floor Planning

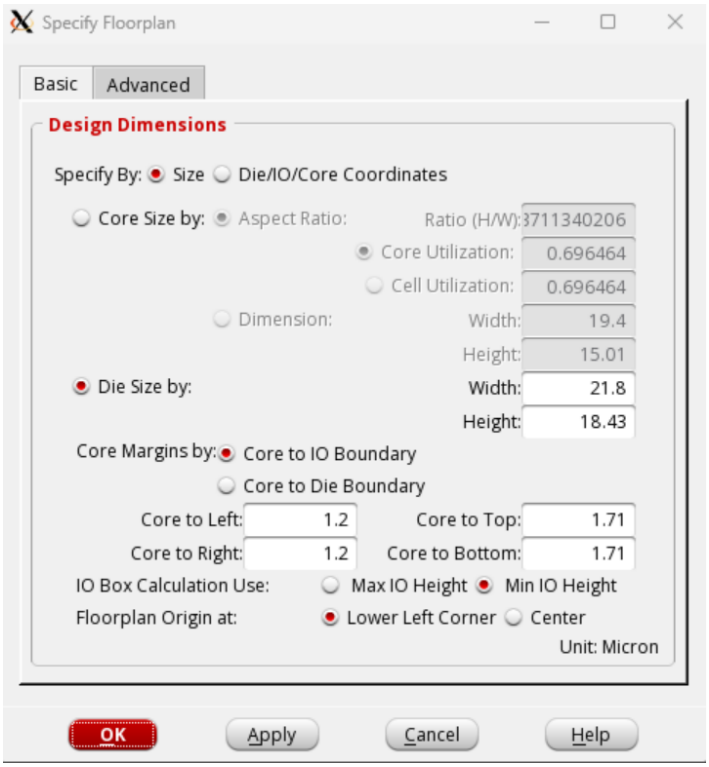


Figure 4: Floorplan Specification

4.4.2 Creating Power Mesh

A network of metal lines laid out in a regular grid pattern across the chip's surface. The primary purpose is to distribute power (voltage and current) evenly and efficiently to all areas of the chip

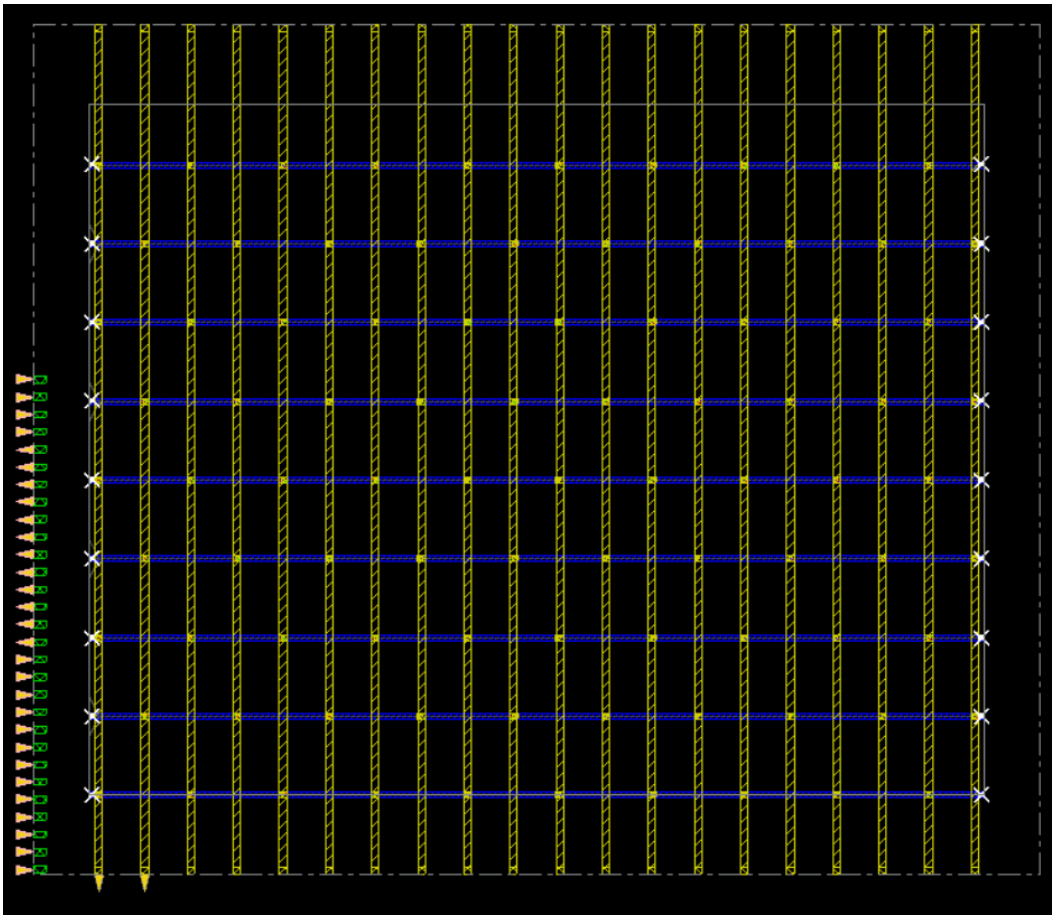


Figure 5: Generated power mesh

4.5.3 Placement

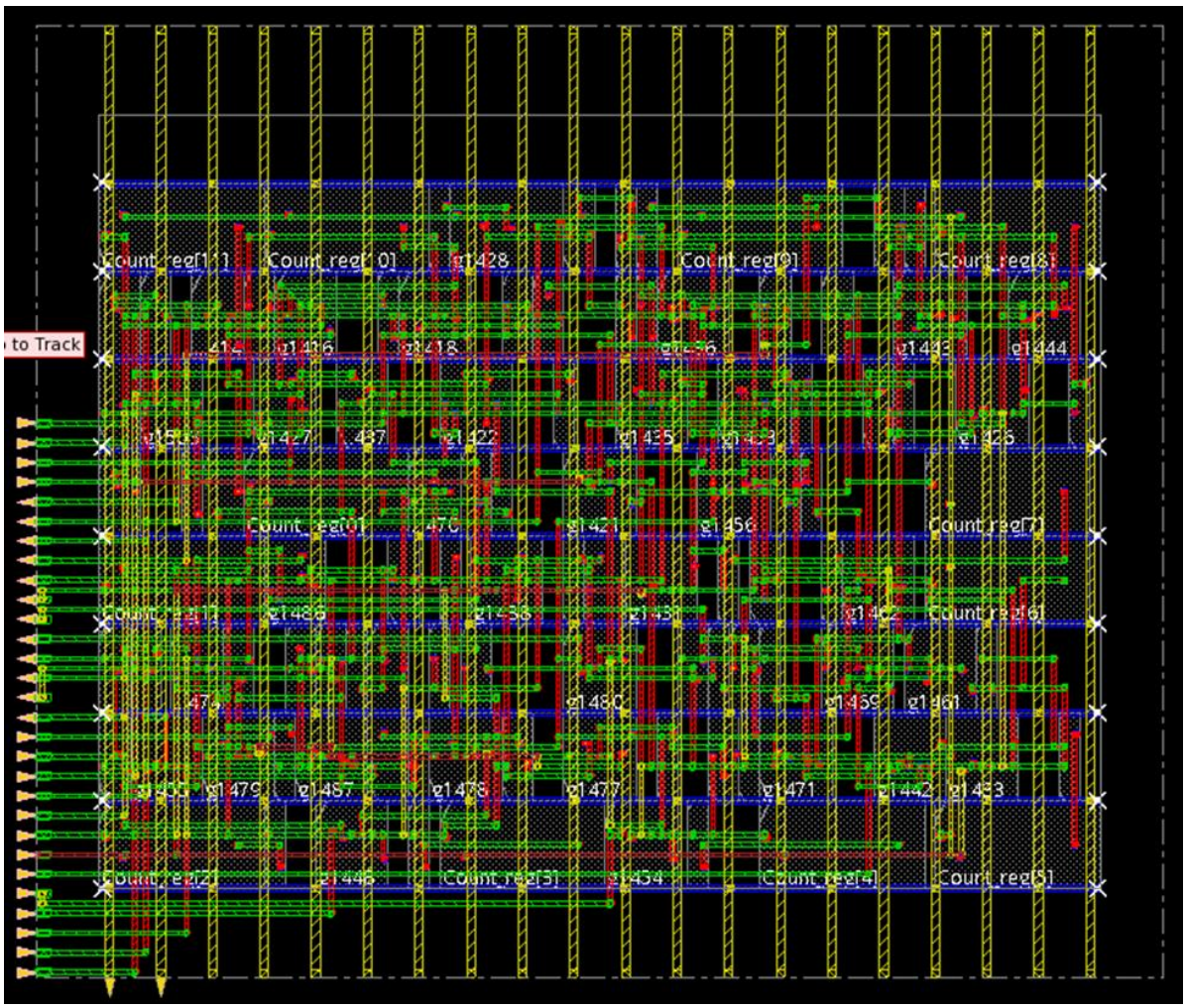


Figure 6: Layout after optimizing placement

4.5.4 Post-Route timing and SI optimization

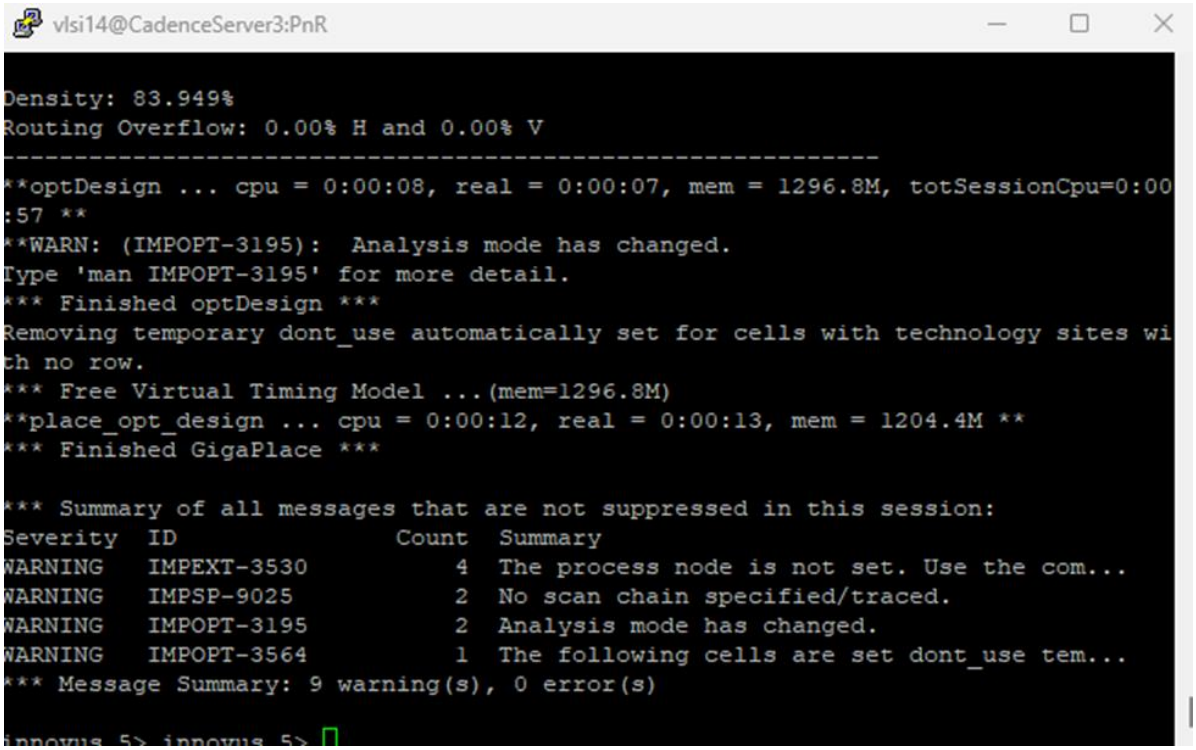


Figure 7: Command Line Report after optimizing placement

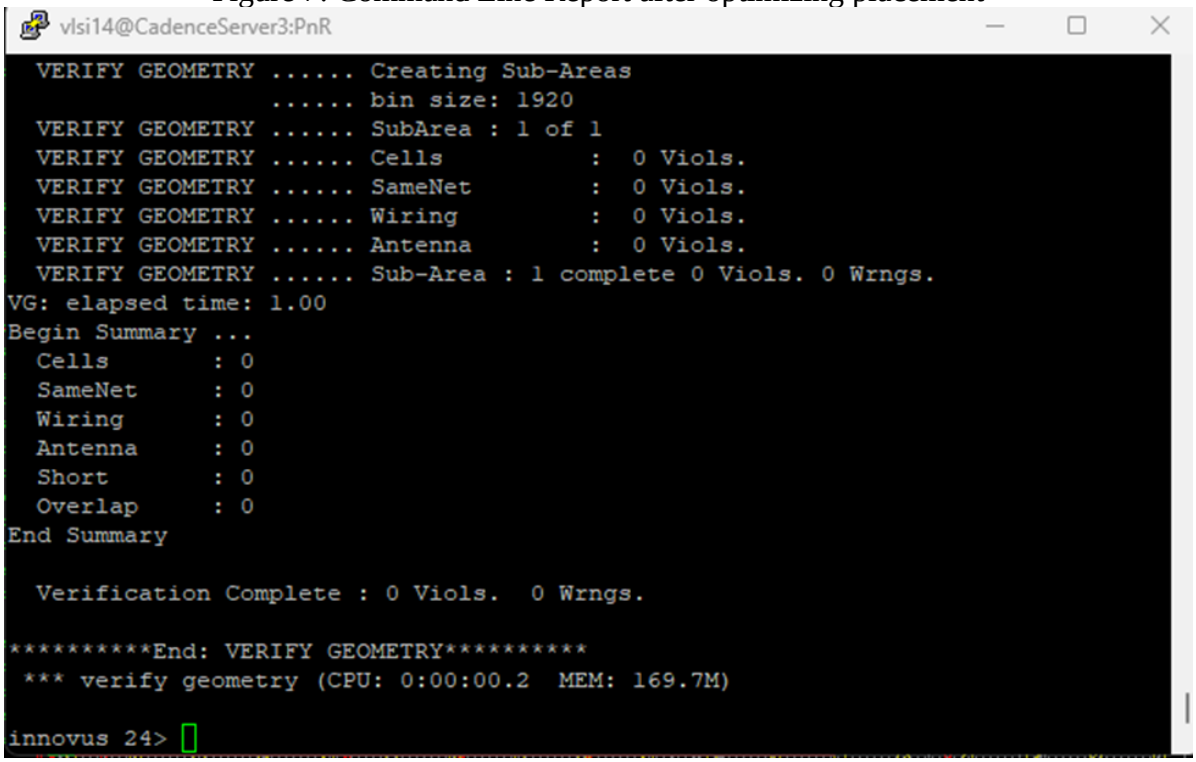


Figure 8: Geometry Verification

4.5.5 Adding Filler Cell and Metal Filling

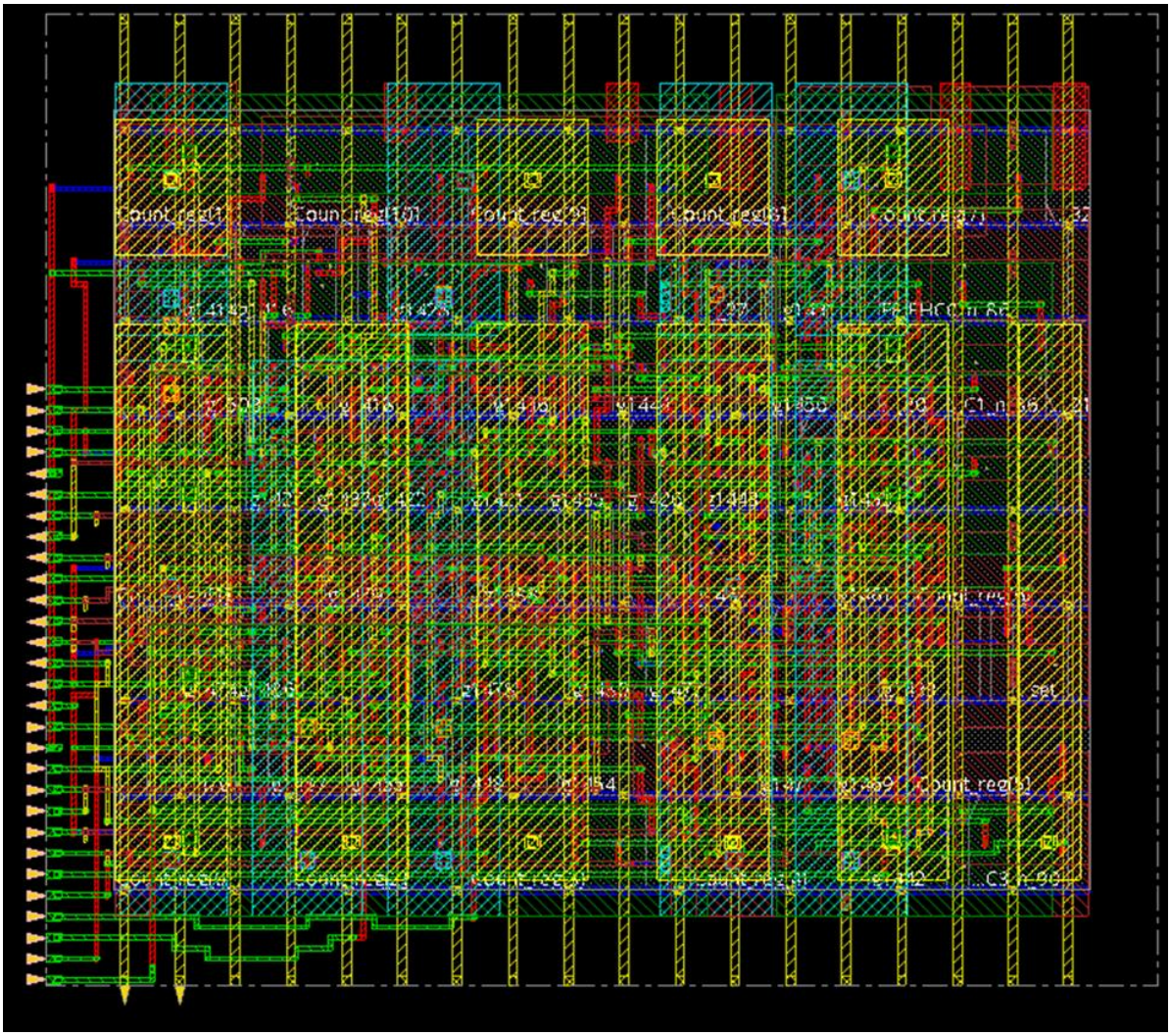


Figure 9: Layout view after adding metal fill

4.5.6 Final View

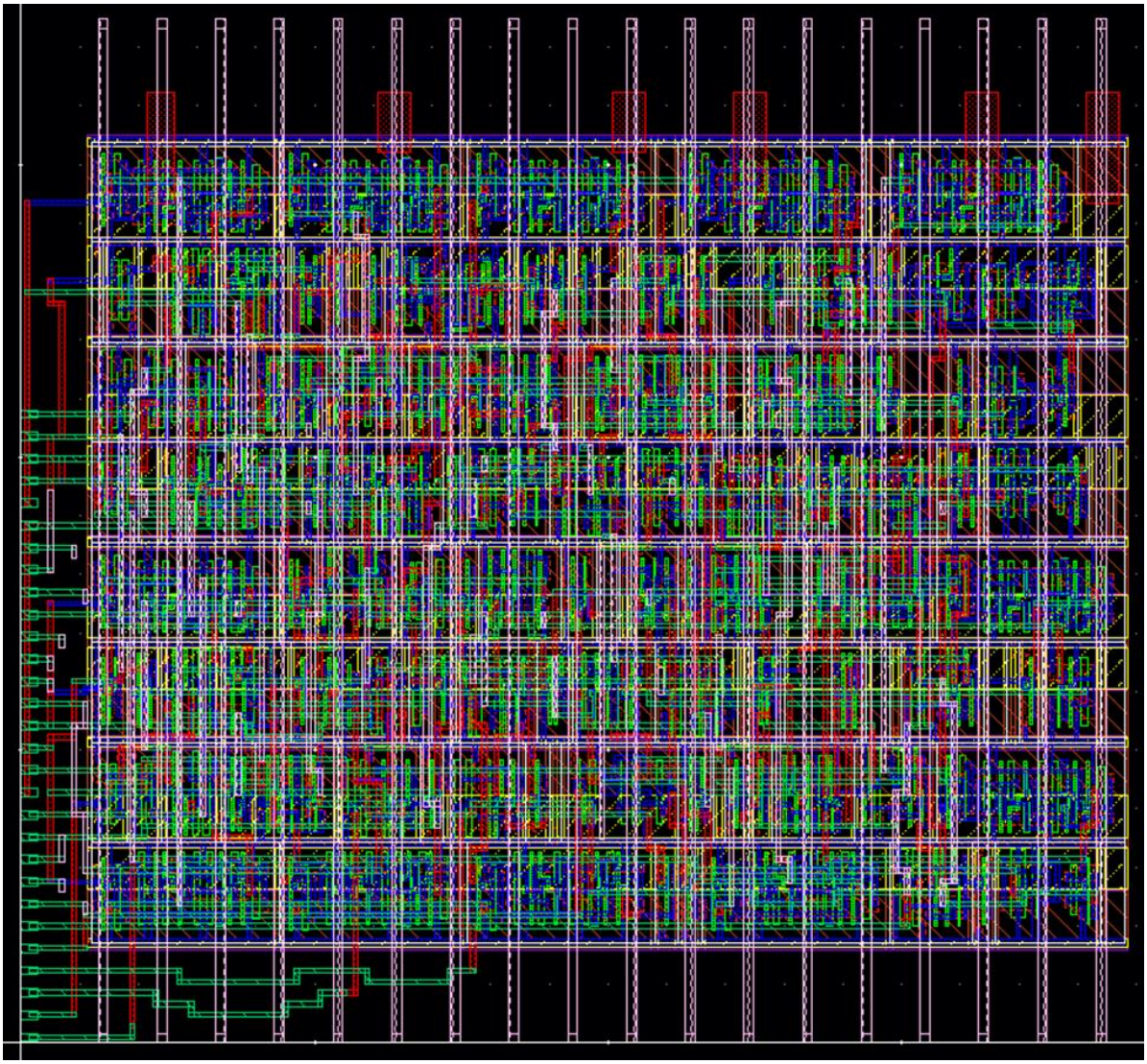


Figure 10: Final view in virtuoso

4.5.7 Design Rule Check

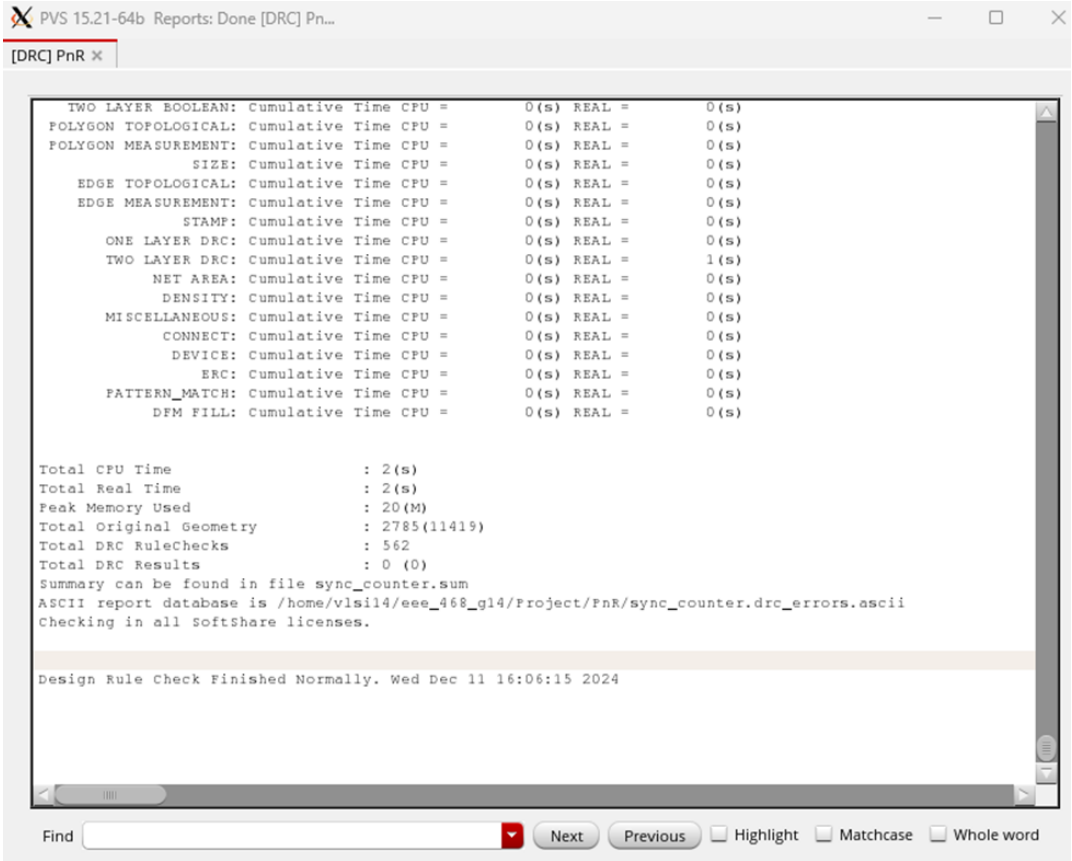


Figure 11: DRC Check with clean result

5. Design Analysis and Evaluation

5.1. Novelty

The 12-bit counter designed in Cadence RTL stands out for its high-performance capabilities, operating at an impressive 100 MHz frequency, which makes it exceptionally well-suited for applications requiring fast and efficient counting, such as digital signal processing, high-speed communication systems, and embedded devices. The counter's 12-bit resolution enables it to handle a wide range of counting operations up to 4096, ensuring versatility across a variety of scenarios while maintaining precision. What further enhances its novelty is the optimization across multiple criteria, including speed, power efficiency, and area utilization, achieving a balance that ensures high reliability even at high clock rates. The design's compact size and high cell density allow seamless integration into space-constrained systems, making it an excellent choice for modern, high-performance electronic projects. Such a well-rounded combination of speed, efficiency, and adaptability highlights the innovation behind its development.

5.2. Design Considerations

Optimization was our highest priority. We have tried to follow the guidelines of VLSI Laboratory to fulfill the requirement for this designed counter. Here the project has been optimized with respect to setup time, hold time, input delay and output delay whereas we have taken 10ns time clock as it is best for this project. The higher clock frequency will require higher power almost 6-7 times. So we can see this is a fair trade between power consumption and speed.

6. Reflection on Individual and Team work

ID	Contribution
1906037	System verilog code and test bench development
1906051	Project optimization and physical design
1906056	System verilog code and test bench development
1906060	Project optimization and physical design

7. Summary

This report presents the successful design and implementation of a 12-bit counter optimized for high-speed operation and efficient integration into modern digital systems. The counter supports high-frequency operation at 100 MHz and features a compact, block-based structure to enhance performance and adaptability for various applications. Key highlights of the project include the following:

Design and Implementation:

The 12-bit counter was architected with a modular design, leveraging compact logic units to achieve a balance between speed and area efficiency. The high-frequency operation was ensured through the careful optimization of logic paths and clock synchronization.

Verification and Simulation:

Directed and layered testbenches were developed to rigorously verify functionality and performance across a wide range of input conditions. Simulations confirmed the counter's correctness, stability, and reliability at the target operating frequency.

Optimization:

The design underwent extensive Power, Performance, and Area (PPA) optimizations. Iterative refinements reduced propagation delays, minimized power consumption, and maximized area efficiency, ensuring the counter met stringent design specifications.

Physical Design:

Synthesis, floor planning, and routing adhered to industry-standard design practices. Clean Design Rule Check (DRC) and Layout Versus Schematic (LVS) reports validated the correctness, manufacturability, and readiness for fabrication.

In conclusion, this project demonstrates the successful application of advanced digital design techniques in creating a high-speed, efficient 12-bit counter. Its performance and compact size make it a strong candidate for integration into modern digital systems requiring reliable and precise counting operations.

8. Conclusion

Here we have designed a small IC but the experience of design will give us knowledge as well as confidence for designing complex circuits. IC fabrication technology is the most critical technology in the world, and we have just learned the previous step for IC fabrication. Again, the way of optimization has helped us to understand how to improve performance in the compact area. In future we hope the knowledge we have gathered from this project will help us a lot