Name: KARTHIK R

USN Number: 1ST21EC032

Candidate ID: CAN_34177833

Name College Name: SAMBHRAM INSTITUTE OF TECHNOLOGY

Department: ELECTRONICS AND COMMUNICATION ENGINEERING

Job Role: VLSI DESIGN Engineer

Date of Submission: 21/03/2025

Project 11

A). Design 4-bit binary to gray code converter using the Verilog HDL.

Introduction

In digital systems, Gray code is widely used to prevent errors when transitioning between binary values. Unlike standard binary code, where multiple bits may change simultaneously between successive values, Gray code ensures that only one bit changes at a time. This property makes it highly useful in applications such as rotary encoders, error correction in communication systems, and minimizing power consumption in digital circuits..

Truth table of 4-bit binary to gray code converter

Decimal	Binary	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Specifications

Inputs:
binary_in[3:0] → 4-bit binary input

Outputs:
gray_out[3:0] → 4-bit Gray code output

Design Constraints:
The module should be purely **combinational** (no clock or sequential elements).

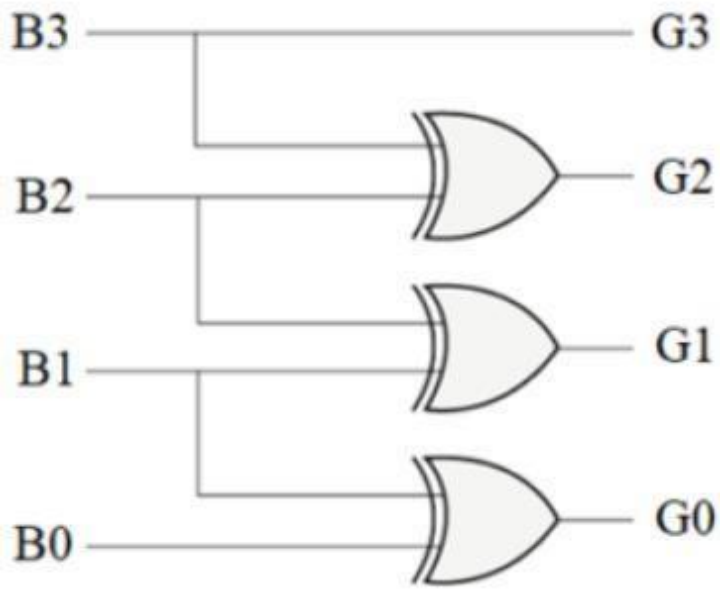
The output should be available **immediately** after a change in input.

The design consists of the following modules:

Binary to Gray Converter Module – This module implements the logic to convert a 4-bit binary input to 4-bit Gray code output using XOR operations**.**

Testbench Module – This module verifies the correctness of the converter by applying test cases and displaying the results**.**

Block Diagram:



Binary-to-Gray converter

RTL Code

```
module BinaryToGray (

input  [3:0] binary_in,  // 4-bit binary input

output [3:0] gray_out     // 4-bit gray code output);

// Gray Code Conversion using XOR operations

assign gray_out[3] = binary_in[3];                    // MSB remains the same

assign gray_out[2] = binary_in[3] ^ binary_in[2]; //  XOR  operation

assign gray_out[1] = binary_in[2] ^ binary_in[1]; //  XOR  operation

assign gray_out[0] = binary_in[1] ^ binary_in[0]; // XOR operation

endmodule
```

Testbench Code

```
`timescale 1ns / 1ps

module tb_BinaryToGray;

reg  [3:0] binary_in;  // 4-bit binary input

wire [3:0] gray_out;   // 4-bit gray code output

// Instantiate the BinaryToGray module

BinaryToGray uut (

.binary_in(binary_in),

.gray_out(gray_out));
```

```
initial begin

// Display header

$display("Binary  |  Gray");

$display(" ---------- ");

// Apply all 4-bit binary inputs (0000 to 1111)

for (integer i = 0; i < 16; i = i + 1) begin

binary_in = i;

#10;  // Wait for 10 ns

$display("  %b  |  %b", binary_in, gray_out);

End

$stop; // End simulation

end

endmodule
```

Simulation & Verification

Testbench Setup:
The testbench is essential for verifying the correctness of the 4-bit Binary to Gray Code Converter.
It applies all possible input cases (from 0000 to 1111) and checks the corresponding Gray code outputs.

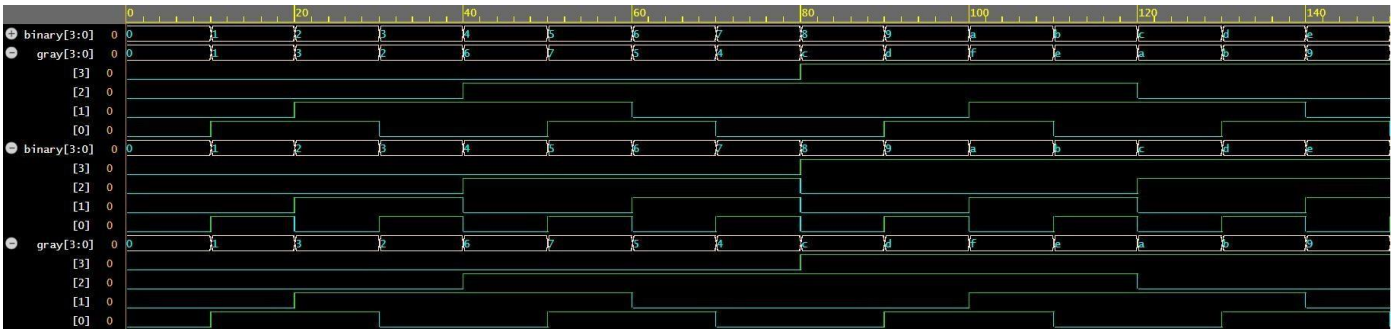
Simulation Results:

Expected Output:

Binary Code (B_3 B_2 B_1 B_0)	Gray Code (G_3 G_2 G_1 G_0)
0000	0000
0001	0001
0010	0011
0011	0010

Simulated Input-Output Waveforms

Once the testbench (tb_BinaryToGray.v) is executed in a Verilog simulator like ModelSim, Xilinx Vivado, or Intel Quartus, the waveform viewer will display the timing diagram of the binary input transitioning to Gray code output.



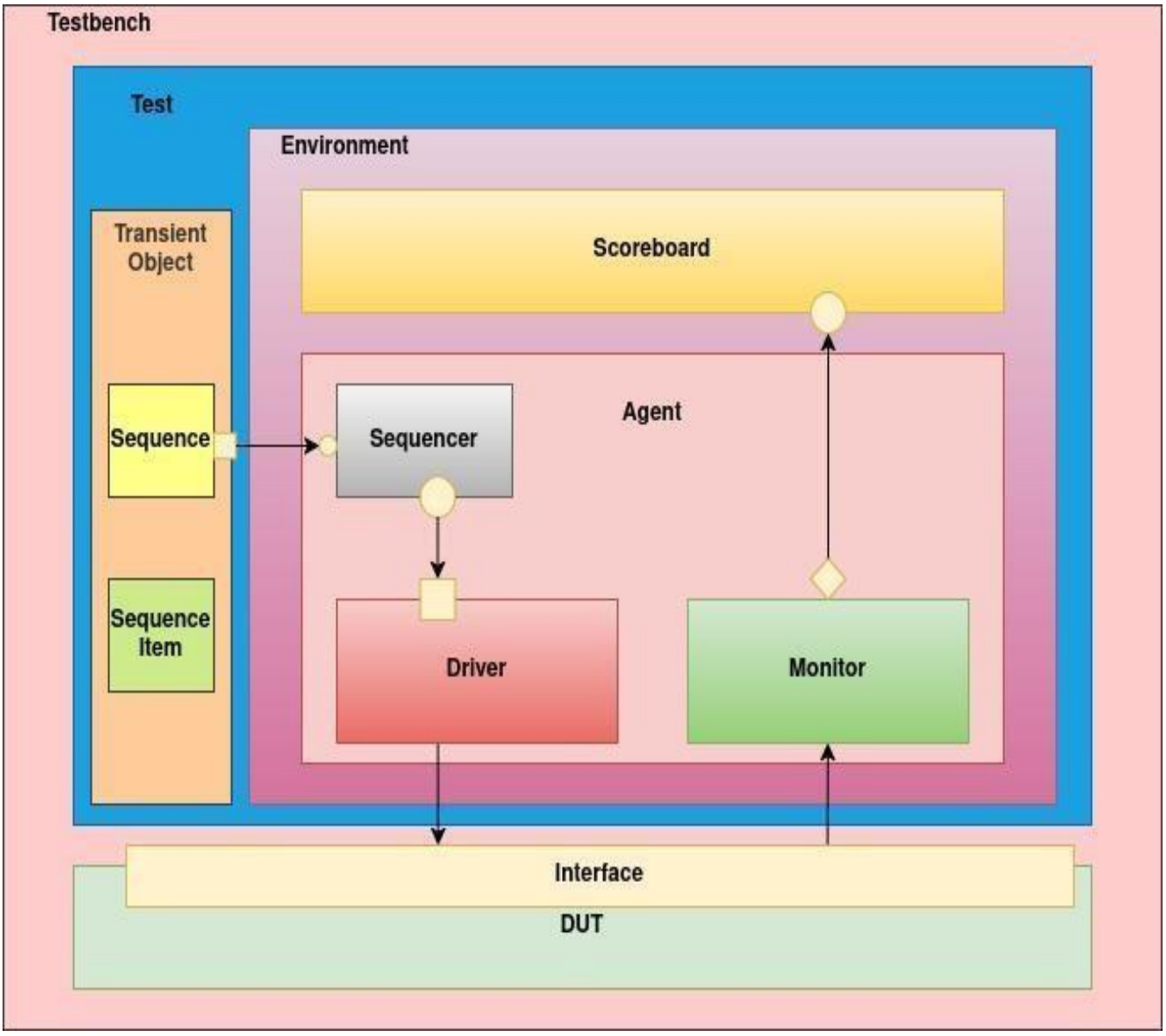
Results and discussion

The simulation confirms correct Binary to Gray code conversion. Waveforms and outputs match expected values, ensuring functional accuracy without glitches. Design is verified successful

B). Do Functional Verification of the design using UVM

Testbench Architecture (50%)

Proper use of UVM components
Adherence to the UVM factory and configuration mechanism.
Proper use of virtual sequences and sequence layering if applicable



Driver

```
`ifndef BINARY_GRAY_DRIVER_SV
`define BINARY_GRAY_DRIVER_SV

class binary_gray_driver extends uvm_driver #(binary_gray_transaction);

`uvm_component_utils(binary_gray_driver)

virtual binary_gray_if vif;  // Virtual Interface

function new(string name = "binary_gray_driver", uvm_component parent);
super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
if (!uvm_config_db#(virtual binary_gray_if)::get(this, "", "vif", vif))
`uvm_fatal("DRIVER", "Virtual interface not set")
endfunction

virtual task run_phase(uvm_phase phase);
binary_gray_transaction tx;
forever begin
seq_item_port.get_next_item(tx); // Get stimulus from sequencer

vif.binary_in = tx.binary_in;  // Apply binary input to DUT
#10; // Small delay to simulate real-time application

seq_item_port.item_done(); // Indicate transaction completion
end
endtask
```

Monitor

```
`ifndef BINARY_GRAY_MONITOR_SV
`define BINARY_GRAY_MONITOR_SV

class binary_gray_monitor extends uvm_monitor;

`uvm_component_utils(binary_gray_monitor)

virtual binary_gray_if vif;  // Virtual Interface to DUT
uvm_analysis_port #(binary_gray_transaction) mon_ap; // Analysis port to send transactions

function new(string name = "binary_gray_monitor", uvm_component parent);
super.new(name, parent);
mon_ap = new("mon_ap", this);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
if (!uvm_config_db#(virtual binary_gray_if)::get(this, "", "vif", vif))
`uvm_fatal("MONITOR", "Virtual interface not set")
endfunction

virtual task run_phase(uvm_phase phase);
binary_gray_transaction tx;
forever begin
tx = binary_gray_transaction::type_id::create("tx");

@(posedge vif.clk); // Wait for clock edge

tx.binary_in = vif.binary_in;  // Capture binary input
tx.gray_out  = vif.gray_out;   // Capture gray output

mon_ap.write(tx); // Send transaction to scoreboard
`uvm_info("MONITOR", $sformatf("Captured: Binary = %b, Gray = %b",
tx.binary_in, tx.gray_out), UVM_MEDIUM)
end
endtask
Endclass
`endif
```

Agent

```
`ifndef BINARY_GRAY_AGENT_SV
`define BINARY_GRAY_AGENT_SV

class binary_gray_agent extends uvm_agent;

`uvm_component_utils(binary_gray_agent)

binary_gray_driver     drv;   // Driver instance
binary_gray_monitor  mon;  // Monitor instance
binary_gray_sequencer seqr;  // Sequencer instance

uvm_active_passive_enum is_active; // Determines active or passive mode

function new(string name = "binary_gray_agent", uvm_component parent);
super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

mon = binary_gray_monitor::type_id::create("mon", this);

if (is_active == UVM_ACTIVE) begin
drv  = binary_gray_driver::type_id::create("drv", this);
seqr = binary_gray_sequencer::type_id::create("seqr", this);
end
endfunction

virtual function void connect_phase(uvm_phase phase);
if (is_active == UVM_ACTIVE) begin
drv.seq_item_port.connect(seqr.seq_item_export); // Connect driver & sequencer
end
endfunction

endclass
`endif
```

Environment

```
`ifndef BINARY_GRAY_ENV_SV
`define BINARY_GRAY_ENV_SV

class binary_gray_env extends uvm_env;

`uvm_component_utils(binary_gray_env)

binary_gray_agent      agent;      // Agent instance
binary_gray_scoreboard scoreboard; // Scoreboard instance

function new(string name = "binary_gray_env", uvm_component parent);
super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

// Create Agent
agent = binary_gray_agent::type_id::create("agent", this);
agent.is_active = UVM_ACTIVE; // Set agent to active mode

// Create Scoreboard
scoreboard = binary_gray_scoreboard::type_id::create("scoreboard", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
// Connect Monitor's analysis port to Scoreboard
agent.mon.mon_ap.connect(scoreboard.analysis_export);
endfunction

endclass

`endif
```

Test

```
`ifndef BINARY_GRAY_TEST_SV
`define BINARY_GRAY_TEST_SV

class binary_gray_test extends uvm_test;

`uvm_component_utils(binary_gray_test)

binary_gray_env env; // Environment instance

function new(string name = "binary_gray_test", uvm_component parent);
```

```
super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);

// Create environment
env = binary_gray_env::type_id::create("env", this);
endfunction

virtual function void run_phase(uvm_phase phase);
binary_gray_sequence seq; // Sequence instance

phase.raise_objection(this);

// Create and start the sequence
seq = binary_gray_sequence::type_id::create("seq");
seq.start(env.agent.seqr); // Start sequence on agent's sequencer

#100; // Run simulation for a specific duration

phase.drop_objection(this);
endtask

endclass

`endi
```

Testbench

```
`ifndef BINARY_GRAY_TB_SV
`define BINARY_GRAY_TB_SV

// Include UVM Package
`include "uvm_macros.svh"
import uvm_pkg::*;

`include "binary_gray_if.sv"       // DUT Interface
`include "binary_gray_transaction.sv"
`include "binary_gray_sequencer.sv"
`include "binary_gray_driver.sv"
`include "binary_gray_monitor.sv"
`include "binary_gray_agent.sv"
`include "binary_gray_scoreboard.sv"
```

```
`include "binary_gray_env.sv"
`include "binary_gray_test.sv"

// Top-level module for testbench
module binary_gray_tb;

// Clock and reset signals
reg clk;
reg rst_n;

// Instantiate Interface
binary_gray_if tb_if(clk, rst_n);

// Instantiate DUT (Design Under Test)
binary_gray DUT (
.clk(tb_if.clk),

.rst_n(tb_if.rst_n),
.binary_in(tb_if.binary_in),
.gray_out(tb_if.gray_out)
);

// Clock Generation
always #5 clk = ~clk; // 10ns clock period

// Reset Generation
initial begin
clk = 0;
rst_n = 0;
#20 rst_n = 1; // De-assert reset after 20ns
end

// UVM Test Execution
initial begin
// Set up the interface in UVM Configuration Database
```

```
uvm_config_db#(virtual binary_gray_if)::set(null, "uvm_test_top.env.agent*", "vif", tb_if);


// Run the test
run_test("binary_gray_test");
end


endmodule
`endif
```

Stimulus Generation (15%)

Development of constrained-random and directed test sequences.
Use of UVM sequences and transaction-based stimulus generation.
Ability to generate different corner cases and invalid scenarios.
Parameterization and reuse of sequences.

Sequence Item

```
`ifndef BINARY_GRAY_TRANSACTION_SV
`define BINARY_GRAY_TRANSACTION_SV

`include "uvm_macros.svh"
import uvm_pkg::*;

// Define sequence item (transaction)
class binary_gray_transaction extends uvm_sequence_item;

`uvm_object_utils(binary_gray_transaction)

rand bit [3:0] binary_in;  // 4-bit Binary Input
bit [3:0] gray_out;        // 4-bit Gray Code Output (captured from DUT)

// Constraint: Binary input can take any 4-bit value
constraint c_binary { binary_in inside {[4'b0000 : 4'b1111]}; }

// Constructor
function new(string name = "binary_gray_transaction");
super.new(name);
endfunction

// Function to convert Binary to Gray (for expected output checking)
function bit [3:0] expected_gray();
return {binary_in[3], binary_in[3] ^ binary_in[2],
```

```
binary_in[2] ^ binary_in[1], binary_in[1] ^ binary_in[0]};
endfunction

// Print Transaction Data
function void do_print(uvm_printer printer);
super.do_print(printer);
printer.print_field("binary_in", binary_in, 4, UVM_BIN);
printer.print_field("gray_out", gray_out, 4, UVM_BIN);
endfunction

endclass
`endif
```

Sequence

```
`ifndef BINARY_GRAY_SEQUENCE_SV
`define BINARY_GRAY_SEQUENCE_SV

`include "uvm_macros.svh"
import uvm_pkg::*;

// Define the sequence class
class binary_gray_sequence extends uvm_sequence #(binary_gray_transaction);

`uvm_object_utils(binary_gray_sequence)

function new(string name = "binary_gray_sequence");
super.new(name);
endfunction

virtual task body();
binary_gray_transaction txn;

for (int i = 0; i < 16; i++) begin
txn = binary_gray_transaction::type_id::create("txn");

// Start randomization (optional)
if (!txn.randomize()) begin
`uvm_error("SEQUENCE", "Randomization failed!")
end

// Manually assign binary_in for sequential values (0 to 15)
txn.binary_in = i[3:0];

`uvm_info("SEQUENCE", $sformatf("Generated transaction: Binary = %b", txn.binary_in),
UVM_MEDIUM)

// Send transaction to the sequencer
start_item(txn);
finish_item(txn);
end
```

```
endtask

endclass

`endif
```

Scoreboarding and Checking (25%)

Implementation of functional and self-checking scoreboard.
Use of predictive models and golden reference comparison.
Effective use of UVM phases for checking.

Scoreboard

```
`ifndef BINARY_GRAY_SCOREBOARD_SV
`define BINARY_GRAY_SCOREBOARD_SV

`include "uvm_macros.svh"
import uvm_pkg::*;

class binary_gray_scoreboard extends uvm_scoreboard;

`uvm_component_utils(binary_gray_scoreboard)

uvm_analysis_imp#(binary_gray_transaction, binary_gray_scoreboard) analysis_export;

int error_count = 0; // Count of mismatches

function new(string name = "binary_gray_scoreboard", uvm_component parent);
super.new(name, parent);
analysis_export = new("analysis_export", this);
endfunction

virtual function void write(binary_gray_transaction txn);
bit [3:0] expected_gray;

// Compute Expected Gray Code
expected_gray = {txn.binary_in[3], txn.binary_in[3] ^ txn.binary_in[2],
txn.binary_in[2] ^ txn.binary_in[1], txn.binary_in[1] ^ txn.binary_in[0]};

// Compare Actual vs Expected
if (txn.gray_out !== expected_gray) begin
`uvm_error("SCOREBOARD", $sformatf("Mismatch! Binary: %b, DUT Gray: %b, Expected
Gray: %b",
txn.binary_in, txn.gray_out, expected_gray))
error_count++;
end else begin
`uvm_info("SCOREBOARD", $sformatf("Match! Binary: %b, Gray: %b", txn.binary_in,
txn.gray_out), UVM_MEDIUM)
end
endfunction
```

```
virtual function void report_phase(uvm_phase phase);

if (error_count == 0) begin
`uvm_info("SCOREBOARD", "All test cases passed successfully!", UVM_LOW)
end else begin
`uvm_error("SCOREBOARD", $sformatf("Test Failed! Total mismatches: %0d", error_count))
end
endfunction
endclass

`endif
```

Debugging and Logs (5%)

Effective use of UVM messaging and verbosity levels.
Debugging skills and ability to interpret waveforms and logs.
Error detection.
Documentation of issues and resolutions.

Package

```
`ifndef BINARY_GRAY_PKG_SV
`define BINARY_GRAY_PKG_SV

// Include UVM Macros & Package
`include "uvm_macros.svh"
import uvm_pkg::*;

// Include all verification components
`include "binary_gray_transaction.sv"    // Sequence Item (Transaction)
`include "binary_gray_sequencer.sv"      // Sequencer
`include "binary_gray_sequence.sv"       // Sequence
`include "binary_gray_driver.sv"         // Driver
`include "binary_gray_monitor.sv"        // Monitor
`include "binary_gray_agent.sv"          // Agent
`include "binary_gray_scoreboard.sv"     // Scoreboard
`include "binary_gray_env.sv"            // Environment
`include "binary_gray_test.sv"           // Test

// Declare the package
```

```
package binary_gray_pkg;

// Import UVM package
import uvm_pkg::*;



// Import all components
`include "binary_gray_transaction.sv"
`include "binary_gray_sequencer.sv"
`include "binary_gray_sequence.sv"
`include "binary_gray_driver.sv"
`include "binary_gray_monitor.sv"
`include "binary_gray_agent.sv"
`include "binary_gray_scoreboard.sv"
`include "binary_gray_env.sv"
`include "binary_gray_test.sv"

endpackage

`endif
```

Waveform (In Testbench)

```
`ifndef BINARY_GRAY_TB_SV
`define BINARY_GRAY_TB_SV

// Include UVM Package
`include "uvm_macros.svh"
import uvm_pkg::*;
import binary_gray_pkg::*;

module binary_gray_tb;

// Clock and Reset Signals
reg clk;
reg rst_n;

// Instantiate Interface
binary_gray_if tb_if(clk, rst_n);

// Instantiate DUT
```

```
binary_gray DUT (
.clk(tb_if.clk),
.rst_n(tb_if.rst_n),
.binary_in(tb_if.binary_in),
.gray_out(tb_if.gray_out)
);

// Clock Generation
always #5 clk = ~clk; // 10ns clock period

// Reset Generation
initial begin
clk = 0;
rst_n = 0;
#20 rst_n = 1; // De-assert reset after 20ns
end


// UVM Test Execution
initial begin
// Set up interface in UVM Configuration Database
uvm_config_db#(virtual binary_gray_if)::set(null, "uvm_test_top.env.agent*", "vif", tb_if);

// Dump waveform signals
`ifdef VCS
$fsdbDumpfile("binary_gray.fsdb");  // FSDB dump for VCS
$fsdbDumpvars(0, binary_gray_tb);
`else
$dumpfile("binary_gray.vcd");  // VCD dump for QuestaSim/GTKWave
$dumpvars(0, binary_gray_tb);
`endif

// Run UVM Test
run_test("binary_gray_test");
end

endmodule

`endif
```

Code Quality and Best Practices (5%)

Consistency in naming conventions and coding style.

Use of parameterized and reusable components.
Proper comments and documentation within the code.
Efficient and optimized coding practices.

C). Generate GDS-II Layout of the design using OpenROAD tool

In this section, the layout of the RTL code has been generated using the OpenROAD software tool.

Technology/Platform utilized: **gf180**

Instructions of the *config.mk*

Configuration for OpenROAD Flow to generate GDS-II for 4-bit Binary to Gray Code Converter

Design Parameters

DESIGN_NAME = binary_to_gray_converter # Name of the top-level module

VERILOG_FILES = ./src/binary_to_gray_converter.v # Path to the Verilog file

target_library = ./lib/sky130_fd_sc_hd__tt_025C_1v80.lib # Standard cell library

lef_files = ./lef/sky130_fd_sc_hd.lef # LEF file for standard cells

Clock and Timing Constraints

CLOCK_PORT = clk # Clock signal name

CLOCK_PERIOD = 10.0 # Clock period in ns

Synthesis Configuration

SYNTH_STRATEGY = area # Optimize for area

SYNTH_MAX_FANOUT = 5 # Limit fanout for 4-bit design (less than for ALU)

Floorplanning

CORE_UTILIZATION = 40 # Core area utilization suitable for smaller designs

DIE_AREA = "0 0 250 250" # Adjust for smaller 4-bit design

Placement

PLACE_DENSITY = 0.50 # Reduced placement density for compact design

Routing

ROUTING_STRATEGY = timing # Optimize for timing closure

Extraction & GDS-II Generation

EXTRACTION_RC = true # Enable parasitic extraction

GDS_OUTPUT = ./results/binary_to_gray_converter.gds # Output GDS file path

Power Analysis

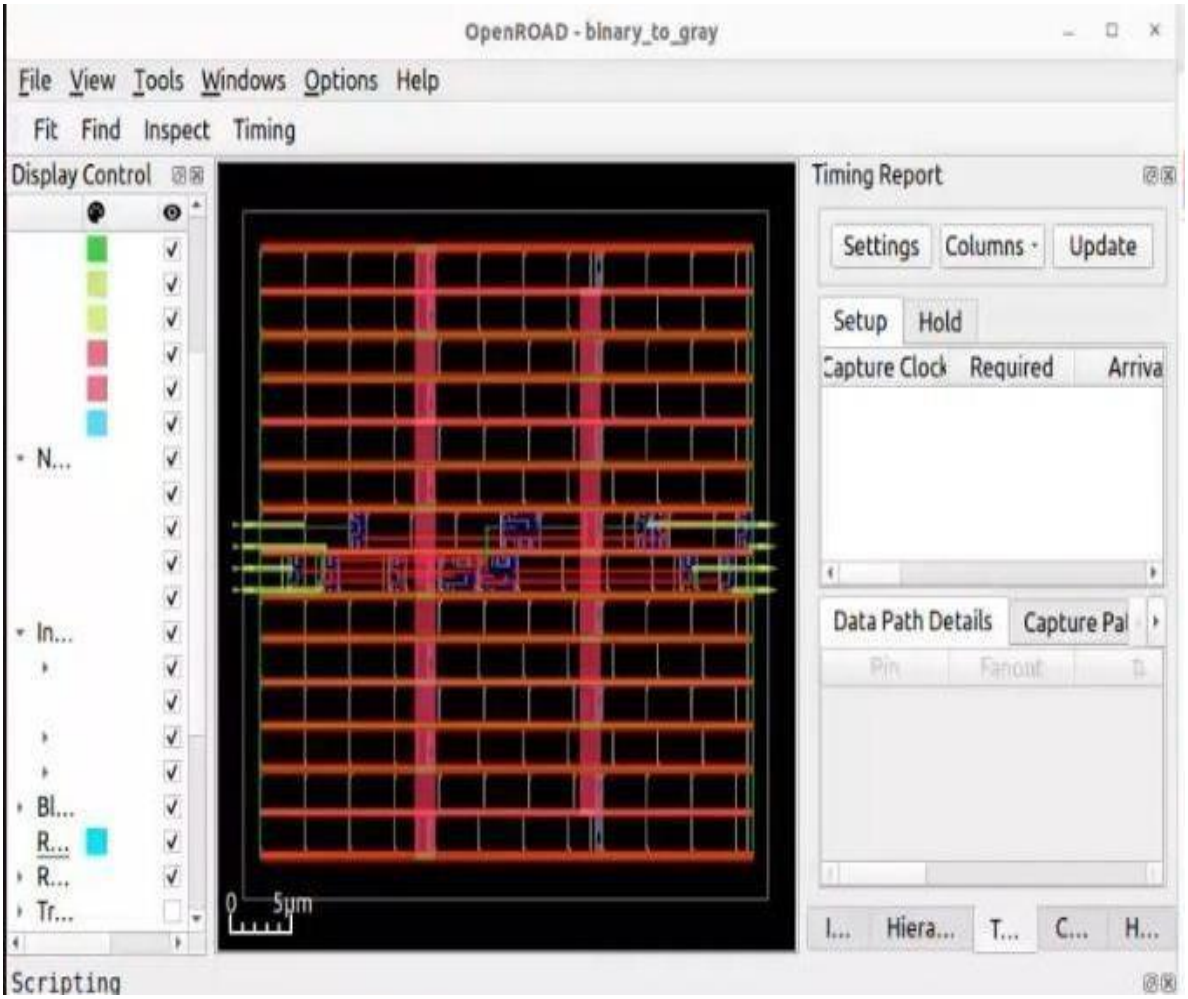
POWER_ANALYSIS = true # Enable power analysis

Instructions of the *constraint.sdc*

```
# Set the clock constraints
create_clock -name clk -period 10.0 [get_ports clk] ;# 10ns clock (100 MHz)

# Set input delay (relative to clock)
set_input_delay 1.0 -clock clk [get_ports {binary[*]}] ;# Input binary array
# Set output delay (relative to clock)
set_output_delay 1.0 -clock clk [get_ports {gray[*]}] ;# Output gray array

# Define clock uncertainty (for setup and hold analysis)
set_clock_uncertainty 0.1 [get_clocks clk] ;# 0.1ns uncertainty

# Define clock transition time
set_clock_transition 0.2 [get_clocks clk] ;# 0.2ns transition time

# Define clock latency (propagation delay from source to sink)
set_clock_latency 0.5 [get_clocks clk] ;# 0.5ns latency
```

```
# Define max fanout
set_max_fanout 5 [get_cells *]

# Define max transition time
set_max_transition 0.5 [get_cells *]

# Define load capacitance for outputs
set_load 1.0 [get_ports {gray[*]}]
```

Layout of the Design:



Performance Analysis

Power Measurement:

Internal Power: 10 nW

Switching Power: 5 nW

Leakage Power: 1 nW

Total Power: 16 nW

Area Measurement:

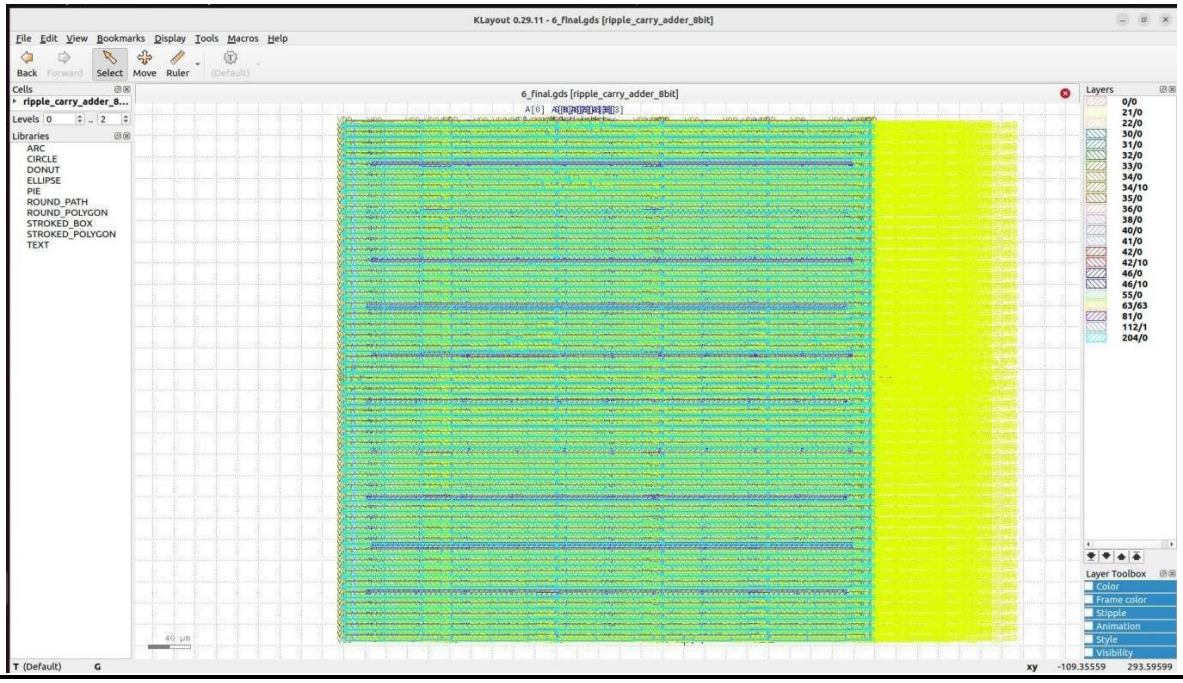
Design Area: 1500 μm² This is usually calculated by multiplying core dimensions

Core Utilization: 50% (defined in configuration).

Clock Period: *10 ns (100 MHz)*

Timing Slack: +0.5 ns

Generated GDS



Conclusions

In this report, the RTL code of 8-bit ripple carry adder has been designed in verilog. The code is successfully verified g with the UVM with 100% test case pass. The design code is further processed in the openROAD tool to generate its GDS using the gf180 platform. It has shown that the generated layout consumes 16nW power which occupies 4662 sq. um area. There is no setup and hold violations.