RAMAIAH
Institute of Technology

DEPARTMENT OF ELECTRONICS AND COMMUNICATION

MINI PROJECT REPORT ON

ROUND ROBIN ARBITER

SUBMITTED BY:

Abhay Swamy V.K (1MS23EC004)

Anaghashree (1MS23EC014)

Ashwith Rai N (1MS23EC029)

Deeksha Pandey (1MS23EC034)

Academic year:2025-2026

ABSTRACT

A Round Robin Arbiter is a key digital circuit used to allocate a shared resource among multiple requesters in a fair and cyclic manner. In this project, a Round Robin Arbiter was designed and implemented using Verilog HDL. The architecture uses a pointer-based rotating priority mechanism that ensures fairness and prevents starvation, allowing each requester to be served in turn. The Verilog RTL design was verified using a structured testbench, and simulation waveforms were analyzed to confirm correct grant sequencing, pointer rotation, and responsiveness to different request patterns. The results show that the arbiter is fully functional, synthesizable, and scalable, making it suitable for applications such as multi-master bus systems, DMA controllers, memory interfaces, and Network-on-Chip routers where fair resource allocation is essential.

INTRODUCTION

In digital systems, multiple modules often need to access a shared resource such as a bus, memory, or communication channel. When several requesters try to use the resource at the same time, an arbitration mechanism is required to decide which requester gets access. An arbiter ensures orderly communication, avoids conflicts, and improves system performance. Among the various arbitration techniques, the Round Robin method is widely used because it provides fairness and prevents any requester from being starved of access.

A Round Robin Arbiter works by serving each requester in a rotating order. After granting access to one requester, the priority shifts to the next requester in the sequence. This cyclic behavior ensures that every module gets an equal chance of accessing the shared resource. Compared to fixed-priority arbiters, which may starve lower-priority requesters, the Round Robin approach guarantees balanced resource allocation.

In this project, a Round Robin Arbiter is designed and implemented using Verilog HDL. The design is aimed at understanding digital arbitration mechanisms, hardware modeling in Verilog, and verification through simulation. A testbench is used to apply different request patterns, and waveform results are analyzed to verify correct functionality. The goal is to develop a fair, efficient, and synthesizable arbiter suitable for multi-master bus systems and modern digital architectures.

OBJECTIVE

The objective of this project is to design and implement a Round Robin Arbiter using Verilog HDL and to understand its role in fair resource allocation within digital systems. The specific goals are:

- To develop an arbiter that allocates access to multiple requesters in a cyclic and fair manner.

- To implement the rotating priority mechanism that prevents starvation.

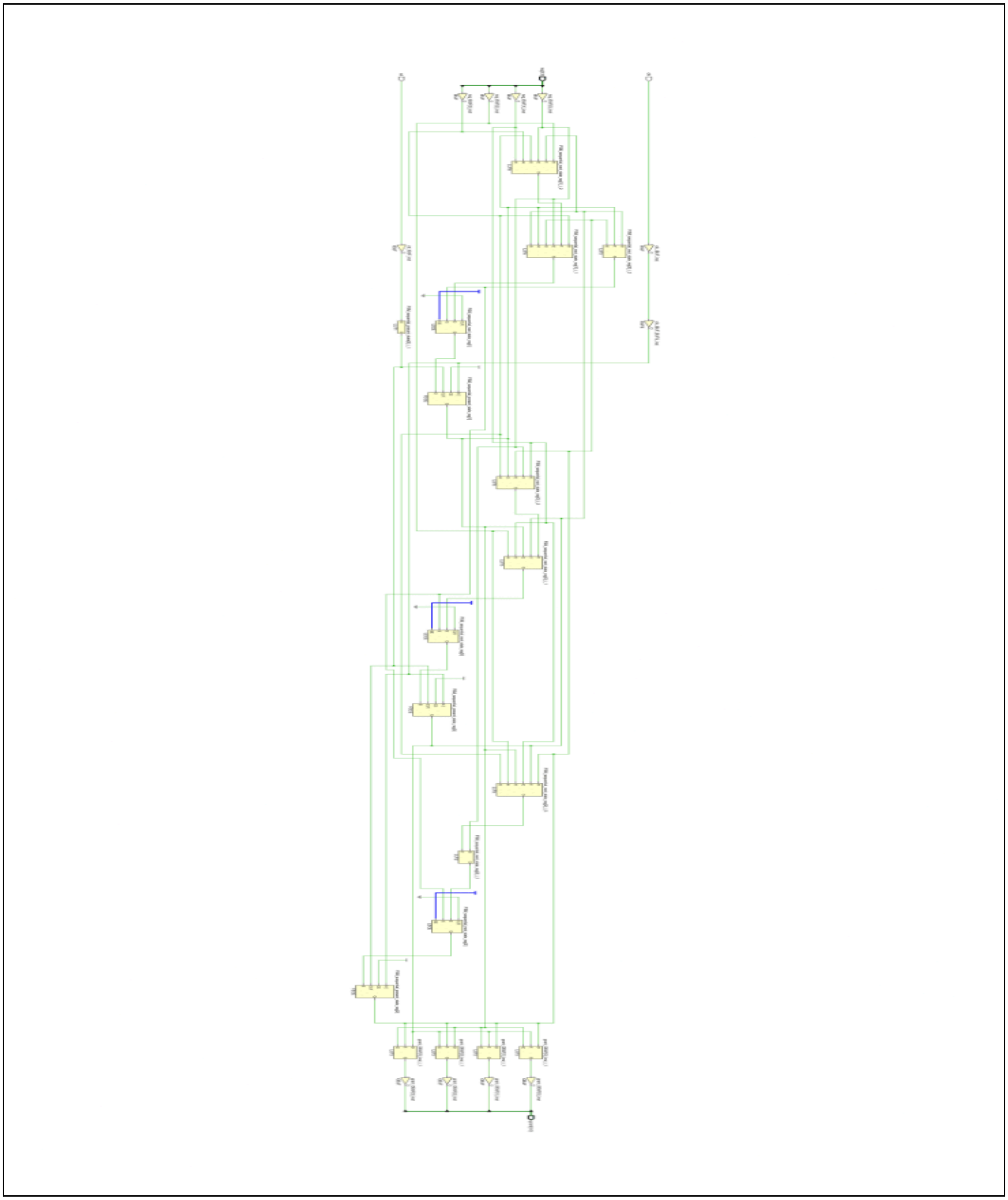
- To write synthesizable Verilog RTL code for the arbiter.

- To create a functional testbench for generating input request patterns.

- To verify the design through simulation and analyze the resulting waveforms.

- To demonstrate that the arbiter operates correctly and can be used in real digital applications such as bus controllers and communication modules.

METHODOLOGY



THE SCHEMATIC VIEW OF ROUND ROBIN ARBITER

The schematic diagram illustrates the overall architecture of the Round Robin Arbiter. The design consists of multiple request inputs, a rotating priority pointer, combinational grant logic, and a sequential block to update the pointer after each successful grant. The arbiter checks active requests, determines the next eligible requester based on the current pointer position, and outputs the corresponding grant signal. After servicing a requester, the priority pointer rotates to ensure fairness and prevent starvation.

WORKING PRINCIPLE:

The Round Robin Arbiter operates by allocating access to a shared resource among multiple requesters in a cyclic and fair sequence. At any given time, several request lines may be active, and the arbiter must decide which requester receives the grant signal. Unlike fixed-priority arbitration, the Round Robin method avoids starvation by rotating the highest priority after each successful grant.

The working principle is based on a priority pointer, which indicates the requester currently holding the highest priority. During each cycle, the arbiter scans the request lines starting from the pointer position. The first active request encountered is granted access. After the grant is issued, the pointer is updated to the next requester in the sequence, ensuring that the next cycle begins with a new highest priority.

If no request is active during a cycle, the pointer remains unchanged. This mechanism guarantees fairness, equal access opportunity, and predictable behavior. The design includes both combinational logic to generate grant signals and sequential logic to update the pointer on every clock cycle. The result is a robust arbitration system suitable for multi-master digital architectures.

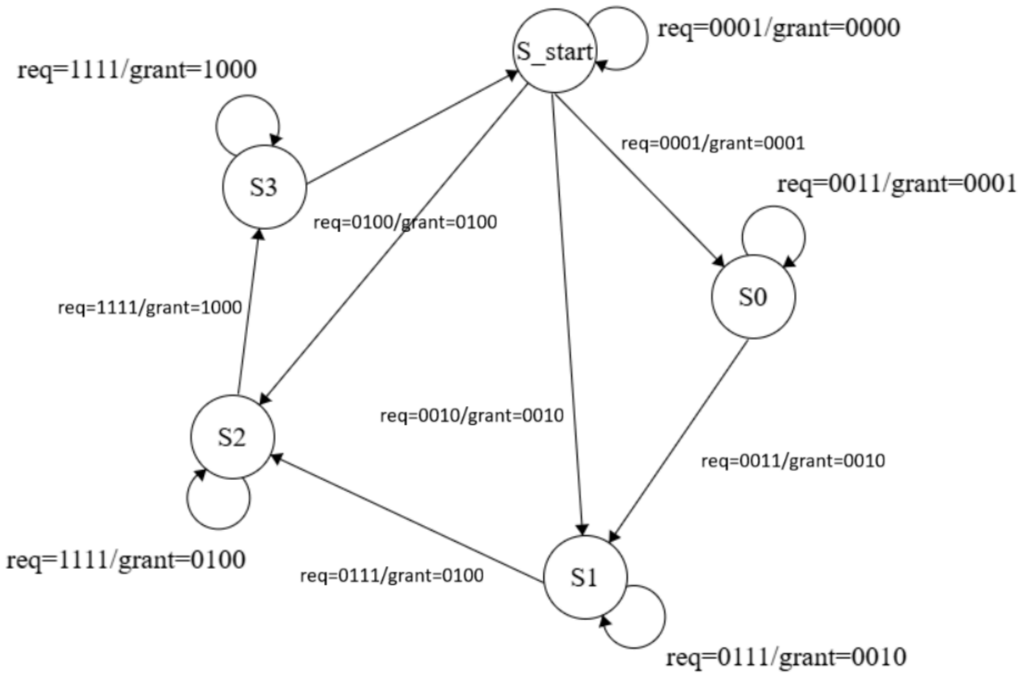
FSM in Round Robin Arbiter:

A Finite State Machine (FSM) is used in the Round Robin Arbiter to control how requests are checked, how the grant is issued, and how the priority pointer rotates. The FSM ensures that the arbitration process happens in an orderly and synchronous manner.

- It typically moves through simple states such as Idle, Check, Grant, and Rotate.
- In Idle, it waits for a request.
- In Check, it inspects requests starting from the current pointer.
- In Grant, it assigns the grant to the selected requester.
- In Rotate, it updates the pointer so the next requester gets priority.

Using an FSM makes the design more structured, prevents glitches, and ensures fairness without starvation.

State Diagram:



Assumption: Starting from S_start.
The state diagram will be similar when started from S0,S1,S2 or S3 state respectively.

VERILOG CODE :

```
module code(clk,rst,req,grant);
input clk;
input rst;
input [3:0]req;
output reg [3:0]grant;
reg [2:0] present_state,next_state;
parameter S_start=3'b000;
parameter S0=3'b001;
parameter S1=3'b010;
parameter S2=3'b011;
parameter S3=3'b100;
always @ (posedge clk or negedge rst) begin
if(!rst)
present_state<=S_start;
else
present_state<=next_state;
end
always @ (*) begin
case(present_state)
S_start:
begin
    if(req[0])
    begin
    next_state=S0;
    end
    else if(req[1])
    begin
    next_state=S1;
    end
    else if(req[2])
    begin
        next_state=S2;
    end
    else if(req[3])
    begin
        next_state=S3;
    end
    else
    begin
    next_state=S_start;
    end
end
S0:
begin
    if(req[1])
    begin
        next_state=S1;
    end
    else if(req[2])
    begin
        next_state=S2;
    end
    else if(req[3])
    begin
        next_state=S3;
    end
    else if(req[0])
    begin
        next_state=S0;
    end
    else
    begin
        next_state=S_start;
    end
end
S1:
begin
    if(req[2])
    begin
    next_state=S2;
    end
    else if(req[3])
    begin
        next_state=S3;
    end
    else if(req[0])
    begin
        next_state=S0;
    end
    else if(req[1])
    begin
        next_state=S1;
    end
    else
    begin
        next_state=S_start;
    end
end
S2:
begin
    if(req[3])
    begin
        next_state=S3;
    end
    else if(req[0])
    begin
        next_state=S0;
    end
    else if(req[1])
    begin
        next_state=S1;
    end
    else if(req[2])
    begin
        next_state=S2;
    end
    else
    begin
        next_state=S_start;
    end
end
S3:
begin
if(req[0])
    begin
        next_state=S0;
    end
    else if(req[1])
    begin
        next_state=S1;
    end
    else if(req[2])
    begin
        next_state=S2;
    end
    else if(req[3]) begin
        next_state=S3;
    end
    else
    begin
        next_state=S_start;
    end
```

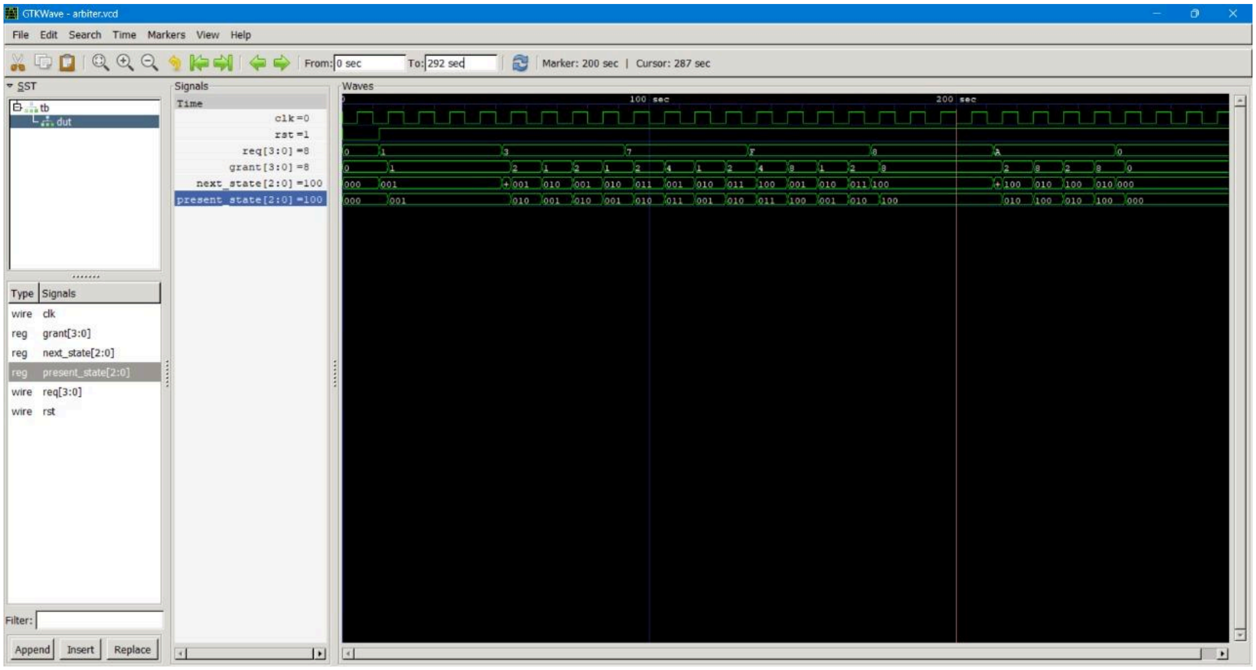
```
end
endcase
end
always @ (*)
begin
    case(present_state)
    S0: grant=4'b0001;
    S1: grant=4'b0010;
    S2: grant=4'b0100;
    S3: grant=4'b1000;
    default: grant=4'b0000;
    endcase
end
endmodule
```

TESTBENCH :

```
module tb_rr_arbiter();
reg clk;
reg rst;
reg [3:0] req;
wire [3:0] grant;

// Instantiate DUT
code dut (
    .clk(clk),
    .rst(rst),
    .req(req),
    .grant(grant)
);

// Generate clock: 10ns period
initial begin
    clk = 0;
    forever #5 clk = ~clk;  // toggles every 5ns -> 10ns clock
end

initial begin
    // Enable waveform dump (for EDA Playground / GTKWave)
    $dumpfile("arbiter.vcd");
    $dumpvars(0, tb_rr_arbiter);

    $monitor($time, " ns | req=%b | grant=%b | state=%d", req, grant, dut.present_state);

    // Apply reset
    rst = 0; req = 4'b0000;
    #12;            // wait a bit
    rst = 1;

    // ---- Requests Changing Every 40 ns ----

    // (0 - 40ns) Only req[0]
    req = 4'b0001;
    #40;

    // (40 - 80ns) req[0] and req[1] active → ping-pong
    req = 4'b0011;
    #40;

    // (80 - 120ns) req[0], req[1], req[2] active → rotates 0→1→2→0→...
    req = 4'b0111;
    #40;

    // (120 - 160ns) All requesting → rotates 0→1→2→3→...
    req = 4'b1111;
    #40;

    // (160 - 200ns) Only req[3]
    req = 4'b1000;
    #40;

    // (200 - 240ns) req[1] and req[3] → alternates 1↔3
    req = 4'b1010;
    #40;

    // (240+) No one requesting → idle
    req = 4'b0000;
    #40;

    $finish;
end

endmodule
```

SIMULATION AND OUTPUT

WAVEFORMS:

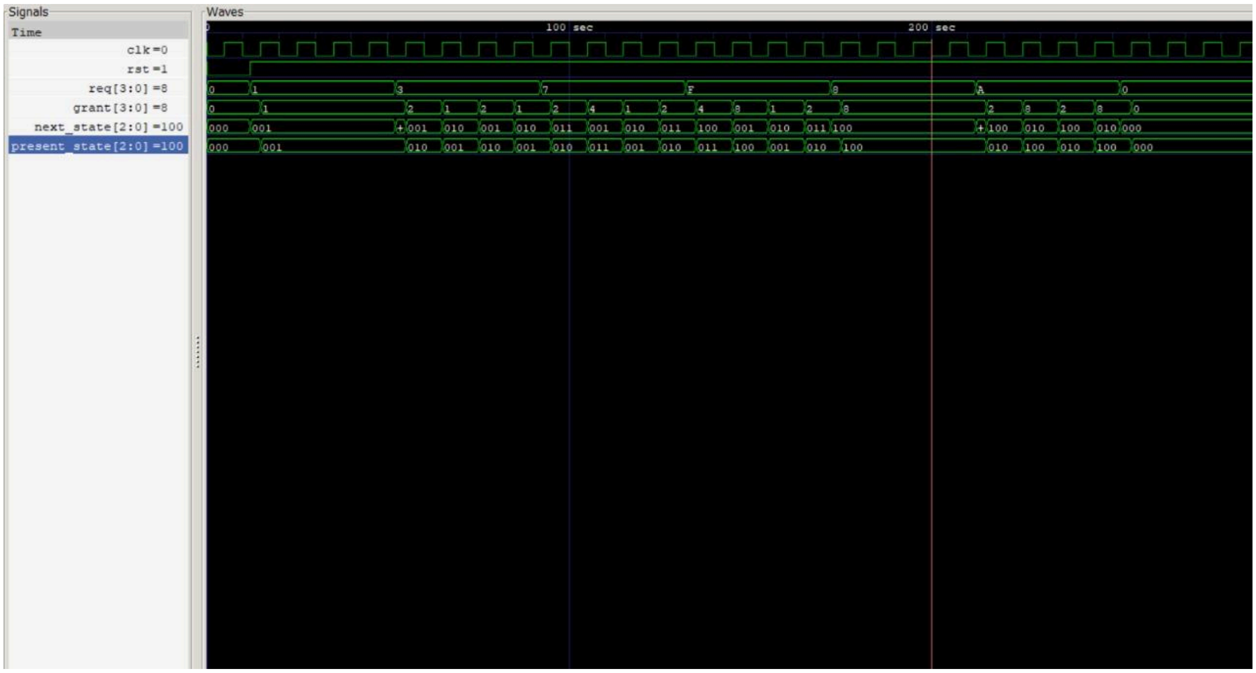


From: 0 sec To: 292 sec

Marker: 200 sec | Cursor: 287 sec

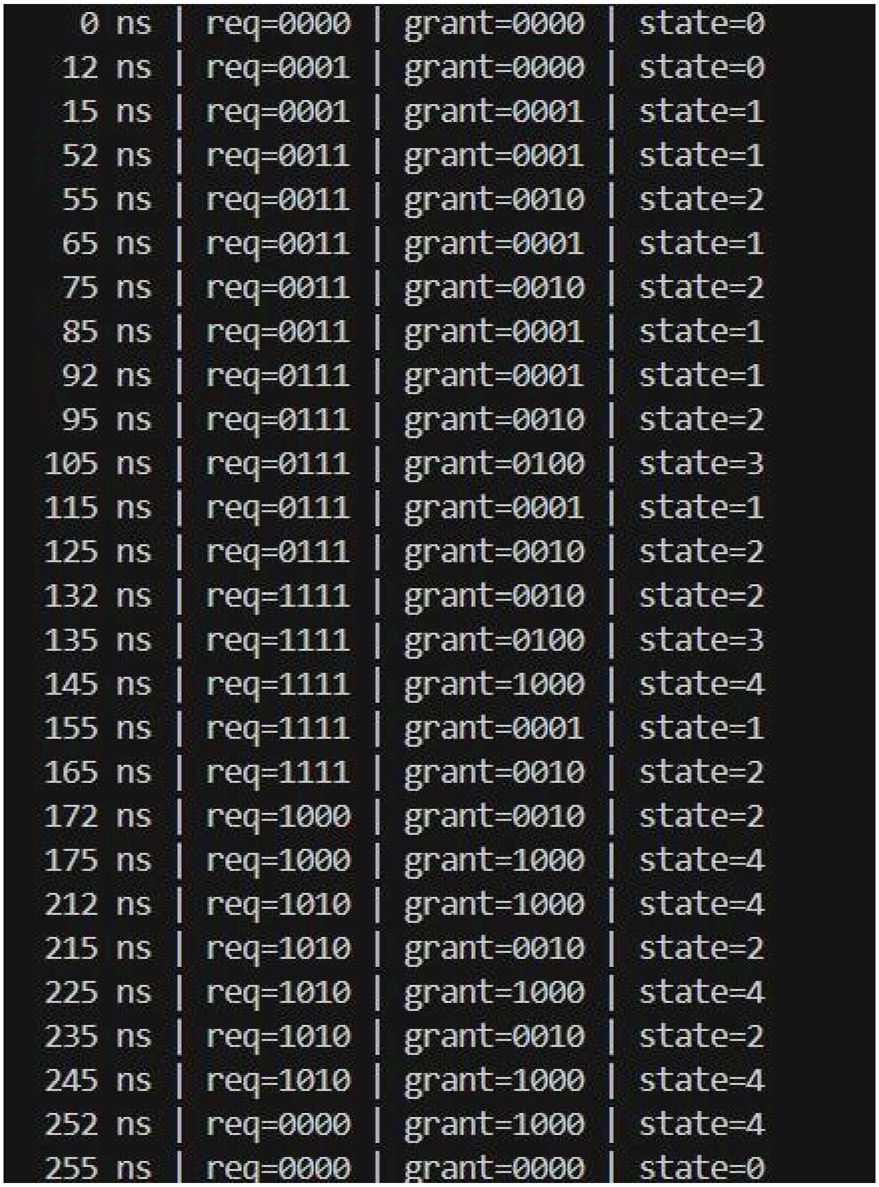
Type	Signals
wire	clk
reg	grant[3:0]
reg	next_state[2:0]
reg	present_state[2:0]
wire	req[3:0]
wire	rst

clk=0
rst=1
req[3:0]=8
grant[3:0]=8
next_state[2:0]=100
present_state[2:0]=100



clk=0
rst=1
req[3:0]=8
grant[3:0]=8
next_state[2:0]=100
present_state[2:0]=100

OUTPUT:

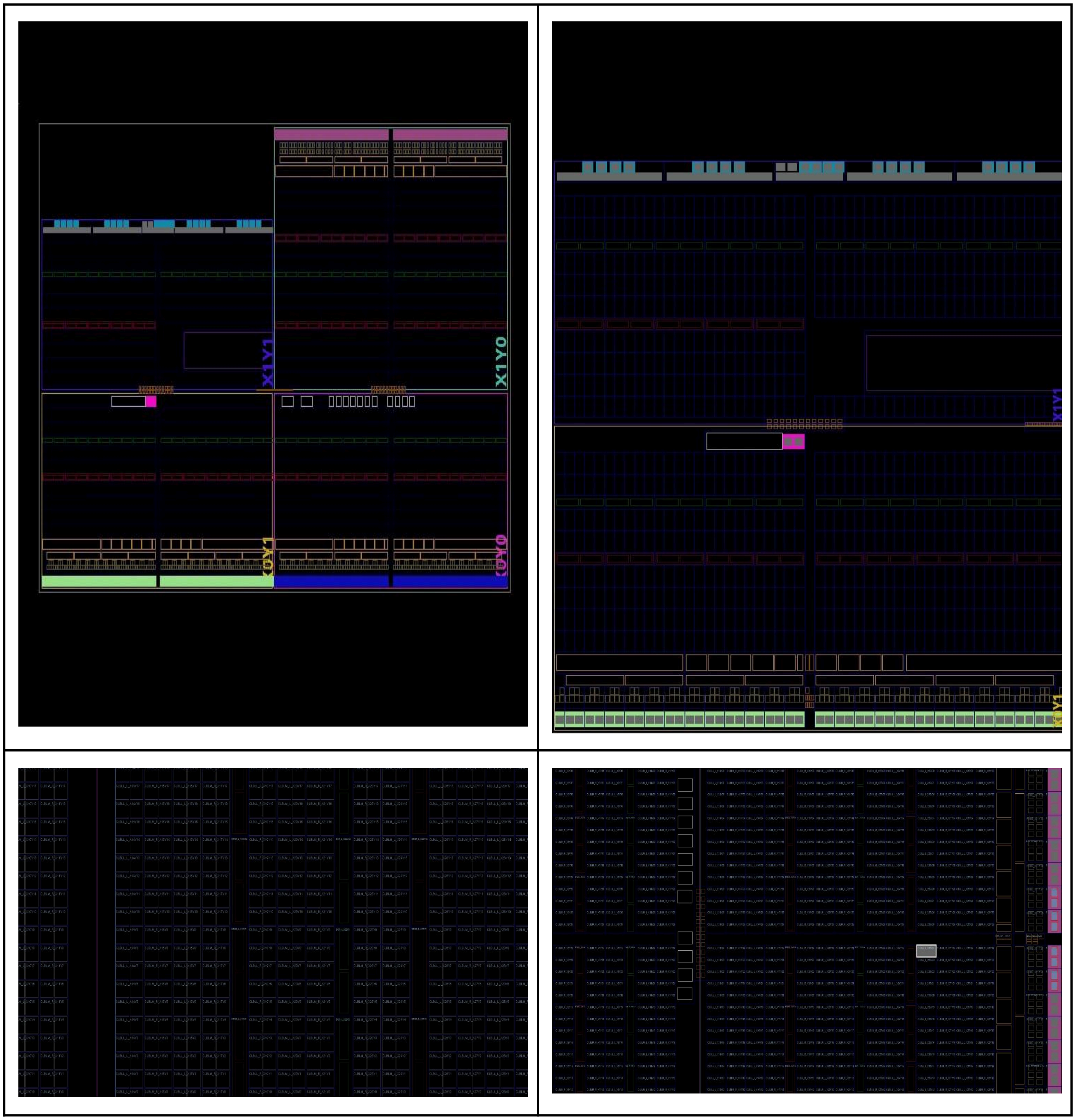


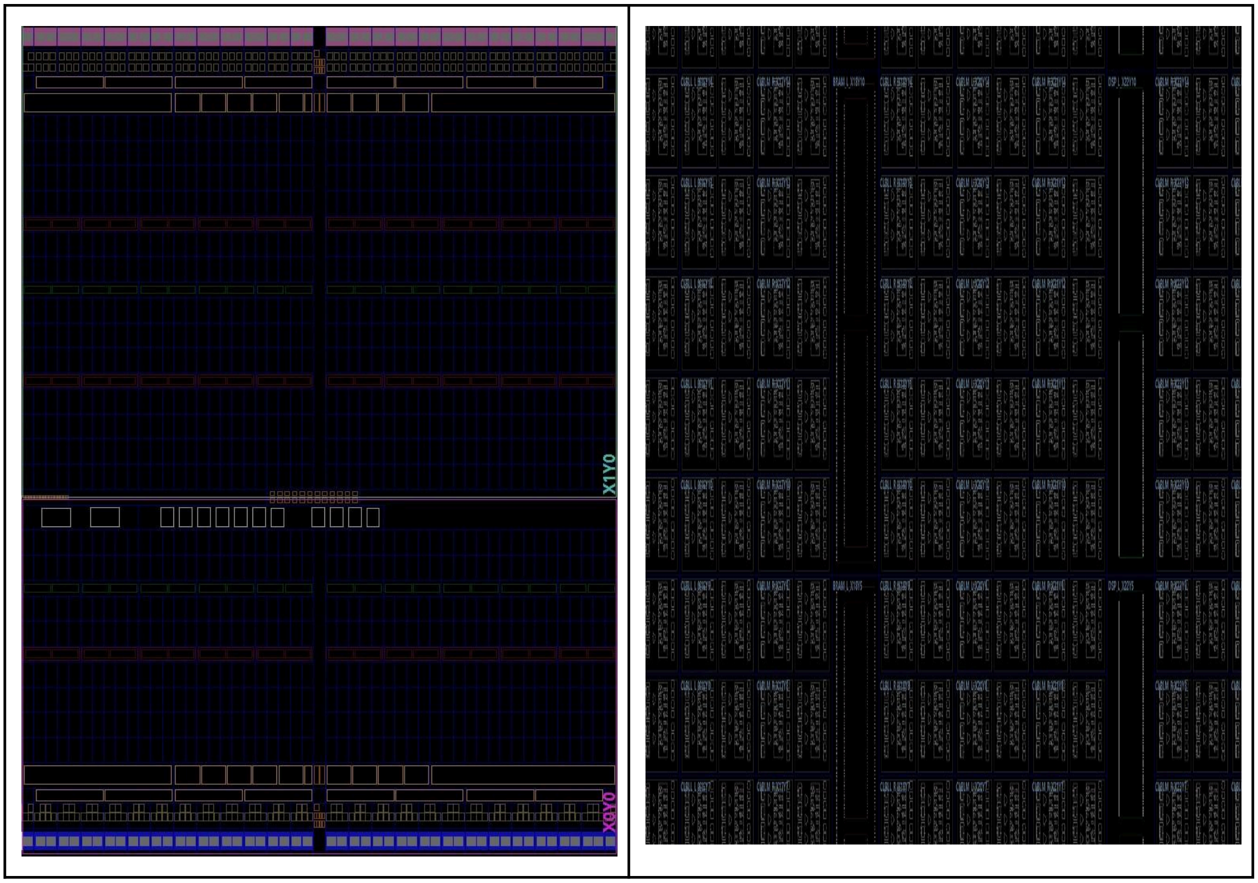
```
  0 ns | req=0000 | grant=0000 | state=0
 12 ns | req=0001 | grant=0000 | state=0
 15 ns | req=0001 | grant=0001 | state=1
 52 ns | req=0011 | grant=0001 | state=1
 55 ns | req=0011 | grant=0010 | state=2
 65 ns | req=0011 | grant=0001 | state=1
 75 ns | req=0011 | grant=0010 | state=2
 85 ns | req=0011 | grant=0001 | state=1
 92 ns | req=0111 | grant=0001 | state=1
 95 ns | req=0111 | grant=0010 | state=2
105 ns | req=0111 | grant=0100 | state=3
115 ns | req=0111 | grant=0001 | state=1
125 ns | req=0111 | grant=0010 | state=2
132 ns | req=1111 | grant=0010 | state=2
135 ns | req=1111 | grant=0100 | state=3
145 ns | req=1111 | grant=1000 | state=4
155 ns | req=1111 | grant=0001 | state=1
165 ns | req=1111 | grant=0010 | state=2
172 ns | req=1000 | grant=0010 | state=2
175 ns | req=1000 | grant=1000 | state=4
212 ns | req=1010 | grant=1000 | state=4
215 ns | req=1010 | grant=0010 | state=2
225 ns | req=1010 | grant=1000 | state=4
235 ns | req=1010 | grant=0010 | state=2
245 ns | req=1010 | grant=1000 | state=4
252 ns | req=0000 | grant=1000 | state=4
255 ns | req=0000 | grant=0000 | state=0
```

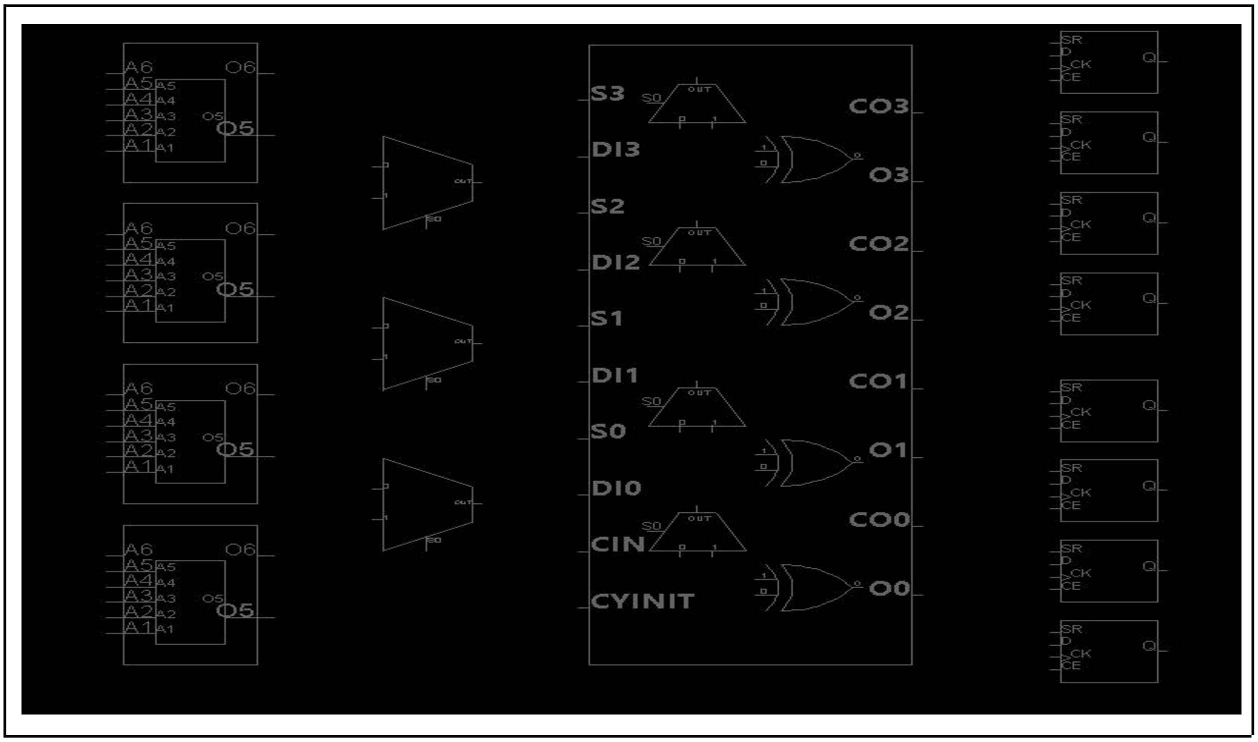
PHYSICAL DESIGN

LAYOUT AND FLOORPLANNING:

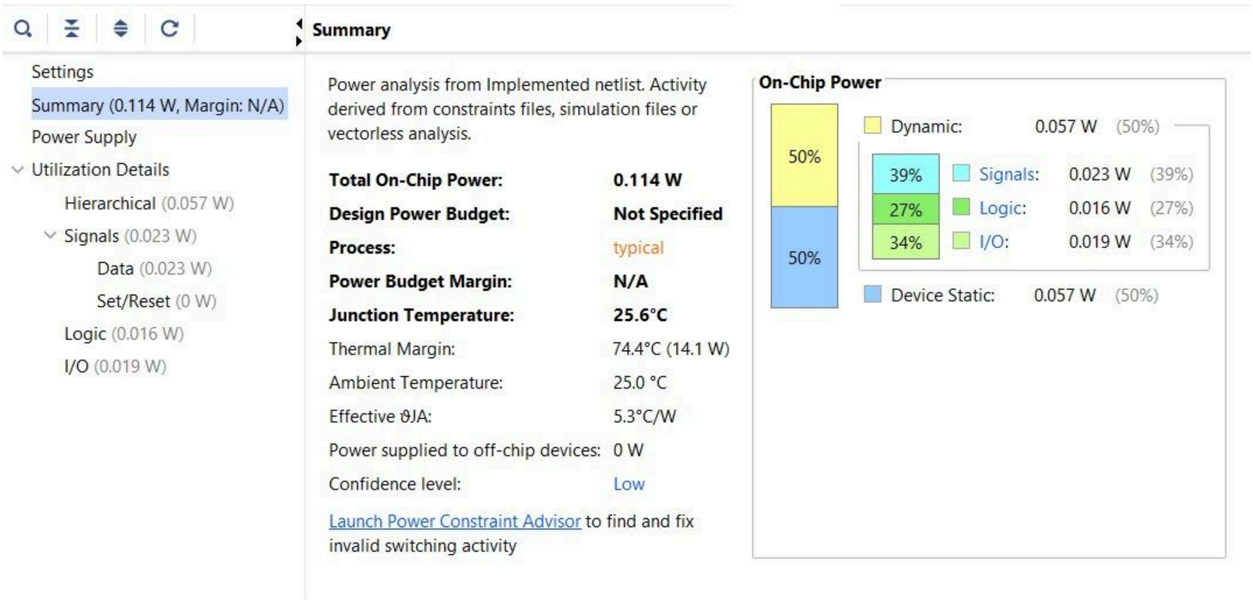
Layout and floorplanning involve arranging the circuit blocks and routing on a silicon chip to prepare it for fabrication. Floorplanning sets the initial placement of major functional units to optimize area, performance, and connectivity, while layout finalizes the detailed geometry and interconnections following design rules.




X1Y0
X0Y0


A6
A5
A4
A3
A2
A1
O6
O5
OUT
S0
S3
DI3
S2
DI2
S1
DI1
S0
DI0
CIN
CYINIT
CO3
O3
CO2
O2
CO1
O1
CO0
O0
SR
D
CK
CE
Q

POWER REPORT:



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.114 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	74.4°C (14.1 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.3°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

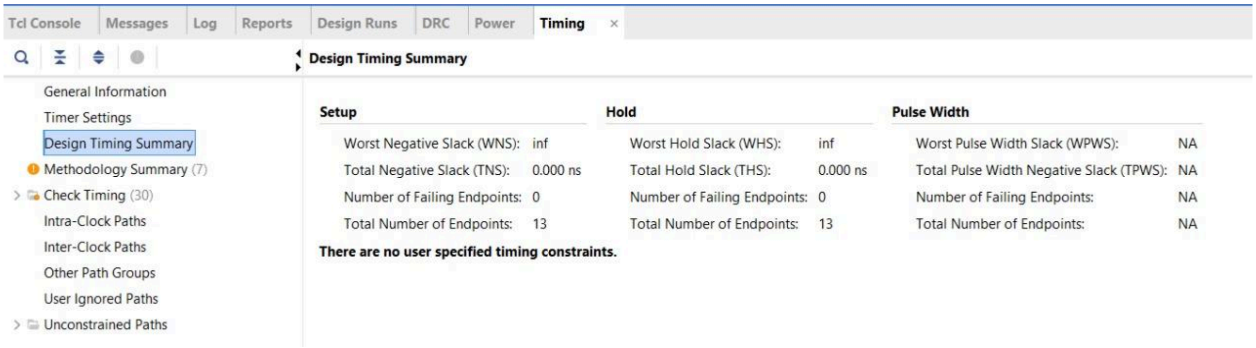
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.057 W	(50%)
Signals:	0.023 W	(39%)
Logic:	0.016 W	(27%)
I/O:	0.019 W	(34%)
Device Static:	0.057 W	(50%)

TIMING REPORT:

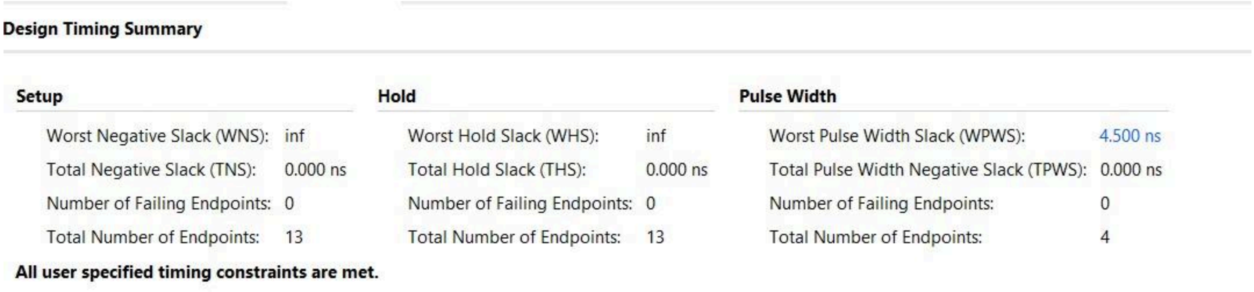
UNCONSTRAINED TIMING REPORT:



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	13	Total Number of Endpoints:	13	Total Number of Endpoints:	NA

There are no user specified timing constraints.

CONSTRAINED TIMING REPORT:

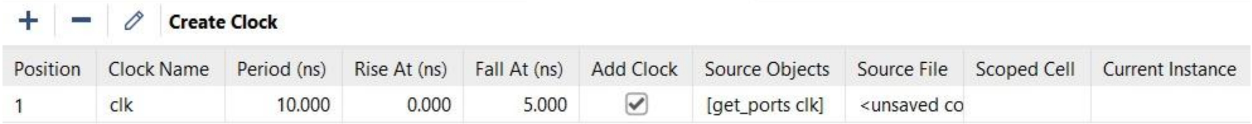


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	13	Total Number of Endpoints:	13	Total Number of Endpoints:	4

All user specified timing constraints are met.

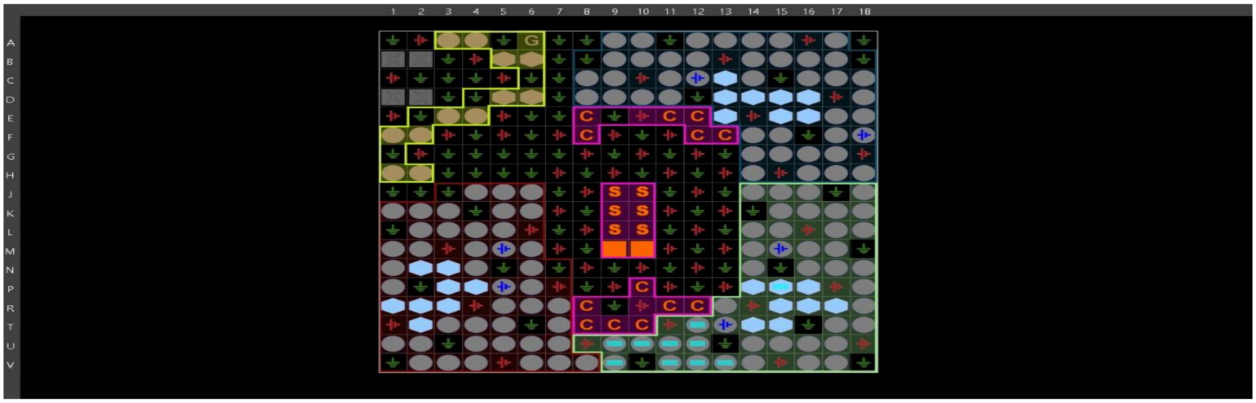
THE CONSTRAINT APPLIED:



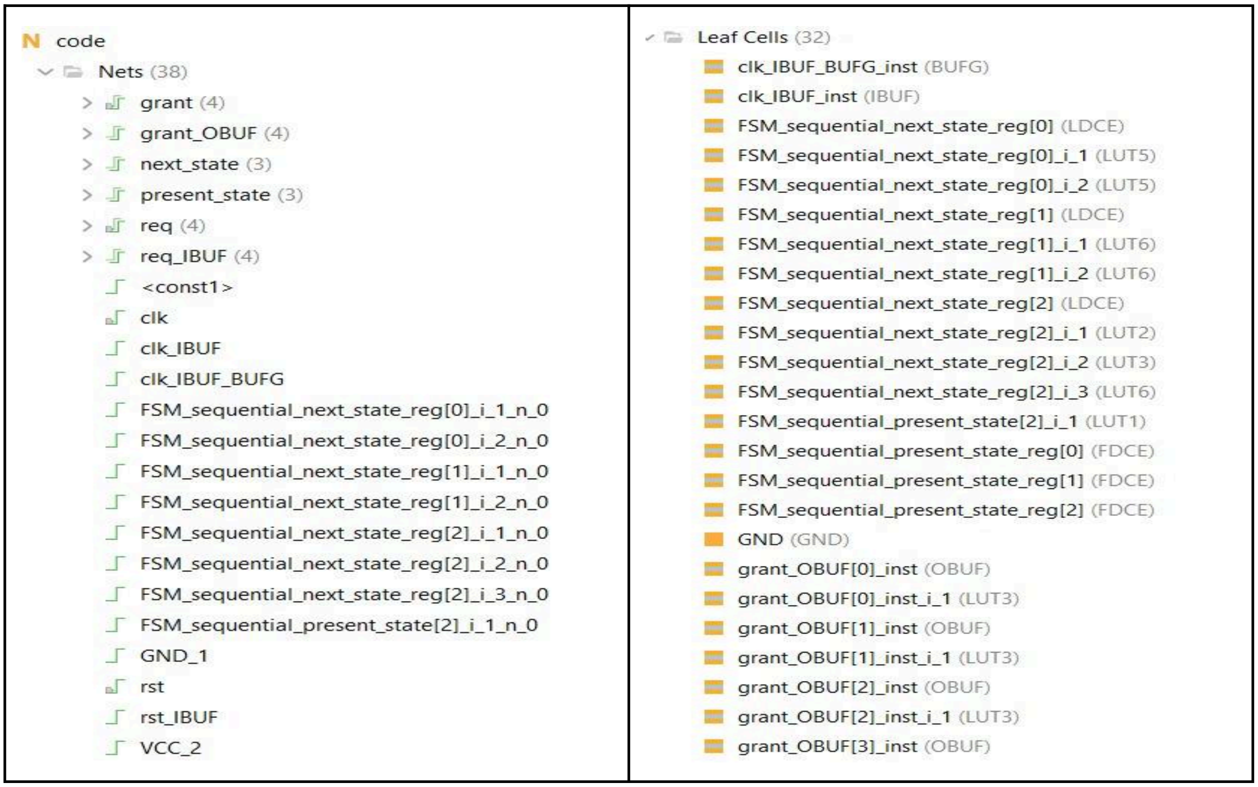
+ − ✎ **Create Clock**

Position	Clock Name	Period (ns)	Rise At (ns)	Fall At (ns)	Add Clock	Source Objects	Source File	Scoped Cell	Current Instance
1	clk	10.000	0.000	5.000	☑	[get_ports clk]	<unsaved co		

INPUT OUTPUT LAYOUT:

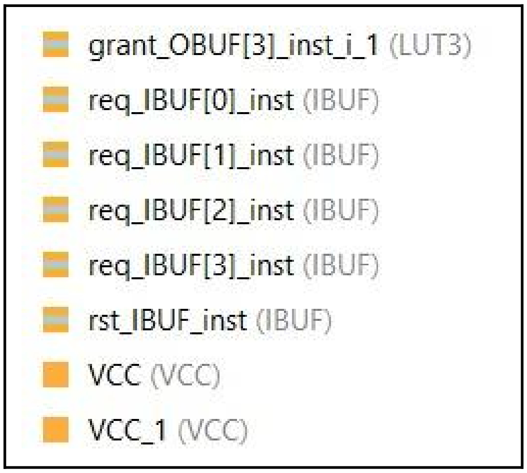


NETLIST:



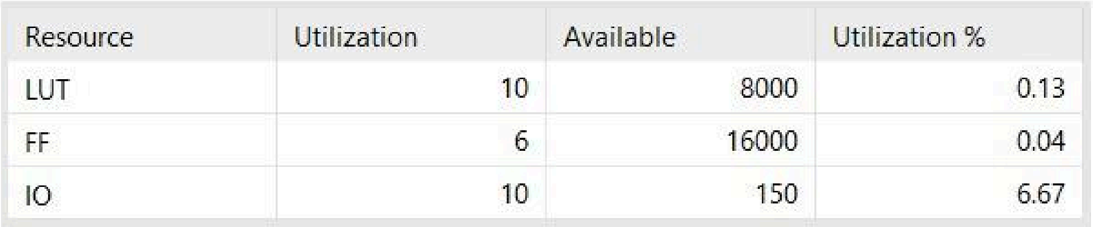
- N code
  - Nets (38)
    - grant (4)
    - grant_OBUF (4)
    - next_state (3)
    - present_state (3)
    - req (4)
    - req_IBUF (4)
    - <const1>
    - clk
    - clk_IBUF
    - clk_IBUF_BUFG
    - FSM_sequential_next_state_reg[0]_i_1_n_0
    - FSM_sequential_next_state_reg[0]_i_2_n_0
    - FSM_sequential_next_state_reg[1]_i_1_n_0
    - FSM_sequential_next_state_reg[1]_i_2_n_0
    - FSM_sequential_next_state_reg[2]_i_1_n_0
    - FSM_sequential_next_state_reg[2]_i_2_n_0
    - FSM_sequential_next_state_reg[2]_i_3_n_0
    - FSM_sequential_present_state[2]_i_1_n_0
    - GND_1
    - rst
    - rst_IBUF
    - VCC_2

- Leaf Cells (32)
  - clk_IBUF_BUFG_inst (BUFG)
  - clk_IBUF_inst (IBUF)
  - FSM_sequential_next_state_reg[0] (LDCE)
  - FSM_sequential_next_state_reg[0]_i_1 (LUT5)
  - FSM_sequential_next_state_reg[0]_i_2 (LUT5)
  - FSM_sequential_next_state_reg[1] (LDCE)
  - FSM_sequential_next_state_reg[1]_i_1 (LUT6)
  - FSM_sequential_next_state_reg[1]_i_2 (LUT6)
  - FSM_sequential_next_state_reg[2] (LDCE)
  - FSM_sequential_next_state_reg[2]_i_1 (LUT2)
  - FSM_sequential_next_state_reg[2]_i_2 (LUT3)
  - FSM_sequential_next_state_reg[2]_i_3 (LUT6)
  - FSM_sequential_present_state[2]_i_1 (LUT1)
  - FSM_sequential_present_state_reg[0] (FDCE)
  - FSM_sequential_present_state_reg[1] (FDCE)
  - FSM_sequential_present_state_reg[2] (FDCE)
  - GND (GND)
  - grant_OBUF[0]_inst (OBUF)
  - grant_OBUF[0]_inst_i_1 (LUT3)
  - grant_OBUF[1]_inst (OBUF)
  - grant_OBUF[1]_inst_i_1 (LUT3)
  - grant_OBUF[2]_inst (OBUF)
  - grant_OBUF[2]_inst_i_1 (LUT3)
  - grant_OBUF[3]_inst (OBUF)

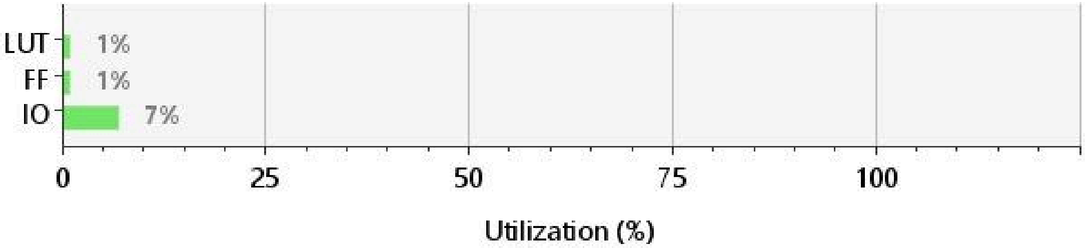


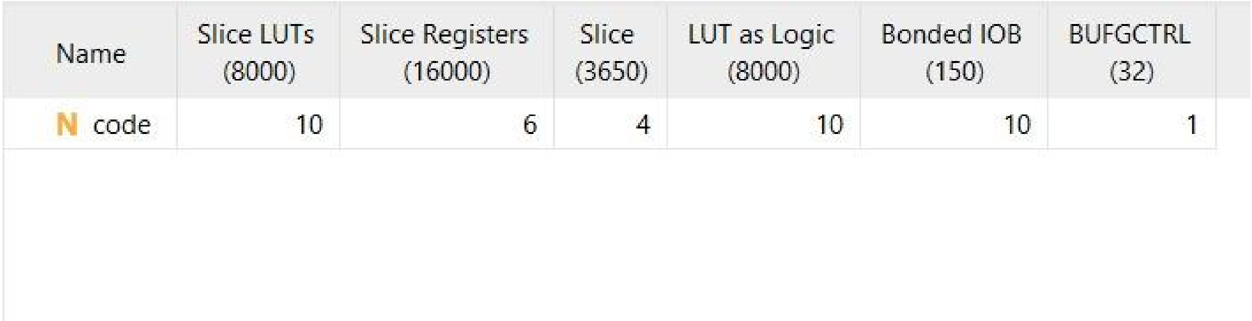
- grant_OBUF[3]_inst_i_1 (LUT3)
- req_IBUF[0]_inst (IBUF)
- req_IBUF[1]_inst (IBUF)
- req_IBUF[2]_inst (IBUF)
- req_IBUF[3]_inst (IBUF)
- rst_IBUF_inst (IBUF)
- VCC (VCC)
- VCC_1 (VCC)

AREA / UTILIZATION DETAILS :



Resource	Utilization	Available	Utilization %
LUT	10	8000	0.13
FF	6	16000	0.04
IO	10	150	6.67





Name	Slice LUTs (8000)	Slice Registers (16000)	Slice (3650)	LUT as Logic (8000)	Bonded IOB (150)	BUFGCTRL (32)
N code	10	6	4	10	10	1

APPLICATIONS

1. **Multi-processor systems**

Ensures each processor gets fair access to shared memory without starvation.

2. **Network-on-Chip (NoC)**

Used in routers/switches inside chips to decide which data packet is sent next.

3. **Bus arbitration**

Controls access to a shared system bus among multiple masters (CPU, DMA, I/O).

4. **Real-time embedded systems**

Provides predictable and fair scheduling for time-critical tasks.

5. **I/O controllers**

Manages multiple input/output requests (USB, PCIe, SPI peripherals).

6. **Memory controllers**

Arbitrates between CPU, GPU, DMA and other modules trying to access RAM.

7. **Communication systems**

Used in packet scheduling for routers, Wi-Fi modules, and MAC layers.

8. **Operating systems scheduling (conceptually)**

Similar principle to round robin CPU scheduling—fairness to processes.

CONCLUSIONS

The Round Robin Arbiter was successfully designed and implemented using Verilog, ensuring fair and starvation-free scheduling among multiple requestors.

The FSM and pointer-rotation logic worked effectively to guarantee that each requestor received equal priority over time.

Simulation results and waveforms verified correct functional behavior, matching the expected output from the testbench.

Physical design steps including floorplanning, placement, routing, power analysis, timing analysis, and layout generation were completed, confirming that the design meets area, power, and timing constraints.

The netlist, timing reports (constrained & unconstrained), and power utilization indicate that the implemented arbiter is efficient, reliable, and suitable for integration in digital systems like NoCs, multiprocessor architectures, and memory controllers.

Overall, the project demonstrates a complete RTL-to-GDSII flow, proving that the designed Round Robin Arbiter is both functionally correct and physically realizable.

REFERENCES

ROUND ROBIN ARBITER VERILOG IMPLEMENTATION - LINKEDIN POST,
Available at:
https://www.linkedin.com/posts/tamilarasi-kannan-341a23259_day-58-round-robin-arbiter-activity-7388816966913941506-vtO7?utm_source=share&utm_medium=member_android&rcm=ACoAAF5DImgByKWYt-inXbW8lrEiPwkQ07kO378
Accessed on : 09 - 11 - 2025

DESIGN OF ROUND ROBIN ARBITER IN VERILOG AND MODELSIM - LINKEDIN POST,
Available at:
https://www.linkedin.com/posts/mubashira-jamil_verilog-based-round-robin-arbiter-activity-7304780638543343616-gveR?utm_source=share&utm_medium=member_android&rcm=ACoAAF5DImgByKWYt-inXbW8lrEiPwkQ07kO378
Accessed on : 09 - 11 - 2025

ROUND ROBIN ARBITER (VARIABLE TIME SLICES) - YOUTUBE,
Available at:
https://youtu.be/d8E3CZhY_FM?si=YycHePIcB4dQLTMy
Accessed on : 04 - 11 - 2025