A

Project Report

On

# ROUND ROBIN ARBITER USING VERILOG

Submitted to

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE AND TECHNOLOGIES

RK VALLEY, KADAPA

in partial fulfillment of the requirement for the award of Degree of

BACHELOR OF TECHNOLOGY

in

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by


B.HEMANJALI R200209

K.HARSHITHA R200376

J.SRAVYA R200366


Under the Guidance of

A.RAMESH, ASSISTANT PROFESSOR

RGUKT,RK VALLEY


DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE AND TECHNOLOGIES

RK VALLEY,KADAPA 516330


2024-2025

# RAJIV GANDHI UNIVERSITY OF KNOWLEDGE AND TECHNOLOGIES

## RK VALLEY,KADAPA 516330

## DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

## CERTIFICATE

This is to certify that the project report entitled **"ROUND ROBIN ARBITER USING VERILOG"** a bonafide record of the project work done and submitted by

**B.HEMANJALI R200209**

**K.HARSHITHA R200376**

**J.SRAVYA R200366**

for the partial fulfillment of the requirements for the award of B.TECH Degree in **ELECTRONICS AND COMMUNICATION ENGINEERING,** RGUKT,RK VALLEY

**GUIDE**

**Mr. A.RAMESH**

**Assistant Professor**

**RGUKT,RK VALLEY**

**Kadapa-516330**

**Head of the Department**

**Mr. Y.ARUN KUMAR**

**Assistant Professor**

**RGUKT,RK VALLEY**

**Kadapa-516330**

RAJIV GANDHI UNIVERSITY OF KNOWLEDGE AND TECHNOLOGIES

RK VALLEY,KADAPA 516330

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING


## DECLARATION

We hereby declare that the project report entitled **"ROUND ROBIN ARBITER USING VERILOG"** submitted to the Department of **ELECTRONICS AND COMMUNICATION ENGINEERING** in partial fulfilment of requirements for the award of the degree of **BACHELOR OF TECHNOLOGY.** This project is the result of our own effort and that it has not been submitted to any other University or Institution for the award of any degree or diploma other than specified above.

**B.HEMANJALI R200209**

**K.HARSHITHA R200376**

**J.SRAVYA R200366**

# ACKNOWLEDGEMENT

We are thankful to our guide **Mr. A.RAMESH**, for his valuable guidance and encouragement. His helping attitude and suggestions have helped us in the successful completion of the project.

We would like to express our gratefulness and sincere thanks to **Mr.Y.ARUN KUMAR**, Head of the Department of **ELECTRONICS AND COMMUNICATION ENGINEERING**, for his kind help and encouragement during the course of our study and in the successful completion of the project work.

Successful completion of any project cannot be done without proper support and encouragement. We sincerely thanks to the management for providing all the necessary facilities during the course of study.

We would like to thank our parents and friends, who have the greatest contributions in all our achievements, for the great care and blessings in making us successful in all our endeavors.

# ABSTRACT

The Round Robin Arbiter is a fundamental component in digital systems designed to manage access to shared resources, ensuring fairness in multi-user or multi-resource environments. This project focuses on the design, implementation, and simulation of a Round Robin Arbiter that allocates access to a shared resource, such as a bus or memory, in a cyclic manner. The primary goal is to develop an efficient and fair arbitration mechanism that ensures no entity monopolizes the resource, while avoiding starvation and ensuring equitable access for all components. The arbiter's operation is based on a simple yet effective rotating priority scheme, where requests are granted in a sequential, round-robin fashion. The design is simulated and tested under various scenarios, demonstrating its functionality in managing simultaneous requests, single requests, and idle states. The project provides insight into resource management in digital systems and highlights the importance of fair arbitration in complex systems.

# TABLE OF CONTENTS

**NAME OF THE CONTENT :** **PAGE NO:**

# 1. INTRODUCTION

## 1.1 BACKGROUND

In digital systems, efficient management of shared resources is crucial to ensure optimal system performance. A Round Robin Arbiter is a hardware mechanism designed to allocate resources fairly among multiple competing entities (such as processors, memory modules, or I/O devices). Unlike other arbitration schemes, such as priority-based or random, Round Robin operates in a cyclic order, ensuring that each entity gets a turn to access the shared resource in a systematic way. This eliminates the chances of starvation and maintains fairness, making it ideal for systems with equal priority requirements. The Round Robin Arbiter finds applications in multi-processor systems, multi-core CPUs, memory access control, and I/O device management.

This report presents the design, implementation, and simulation of a Round Robin Arbiter, focusing on its functionality, fairness, and performance in managing resource allocation.

## 1.2 OBJECTIVE

The primary objectives of this project are:

To design and implement a Round Robin Arbiter that efficiently allocates resources to multiple competing entities.

To simulate the arbiter in a controlled environment and verify its functionality in various operational scenarios.

To analyze the advantages and limitations of the Round Robin arbitration scheme.

To demonstrate how a Round Robin Arbiter ensures fair access to resources without starvation.

## 1.3 SCOPE

The scope of this project covers the following:

**Designing the Arbiter:**
Develop a hardware-based Round Robin Arbiter using a sequential logic approach that grants access to different requesters in a rotating order.

**Simulation & Testing:**
Implement the arbiter design in a simulation environment (e.g., ModelSim, Vivado) and test its performance under various conditions such as:

1. Simultaneous requests from multiple entities.
2. Single request scenarios.
3. No requests scenario.

**System Integration:**
Integrate the arbiter into a basic bus system and demonstrate its role in managing access to a shared resource.

**Performance Analysis:**
Assess the system's performance in terms of fairness, response time, and efficiency.

## 1.4 Working Principle

The Round Robin Arbiter operates based on a simple cyclic mechanism where access to the shared resource is granted in a rotating manner.

Working Steps:
**1. Request Inputs:** Multiple entities (e.g., devices, processors) make requests for resource access. Each entity has a dedicated request line.

**2. Grant Outputs:** The arbiter grants access to only one entity at a time, and this is communicated via grant lines.
**3. Round Robin Cycle:** When the arbiter detects multiple active requests, it grants access to the requesting entity that is next in the cycle, maintaining fairness by giving each entity an equal chance to access the resource.
**4. Reset/Refresh:** After granting access, the arbiter waits for the next round of requests, resetting the cycle when necessary.

## 1.5 OVERVIEW

One common arbitration scheme is the simple priority arbiter. Each requester is assigned a fixed priority, and the grant is given to the active requester with the highest priority. For example, if the request vector into the arbiter is request [N-1:0], request [0] is typically declared the highest priority. If request [0] is active, it gets the grant. If not, and request [1] is active,grant [1] is asserted, and so on. Simple priority arbiters are very common when hosing between just a few requesters.
For example, a maintenance port may always be lower priority than the functional port. ECC corrections may always be higher priority than all other requests. Priority arbiters are also often used as the basis for other types of arbiters. A more complexarbiter may reorder the incoming requests into the desired priority, run these scrambled requests through a simple priority arbiter, then unscramble the grants which come out.

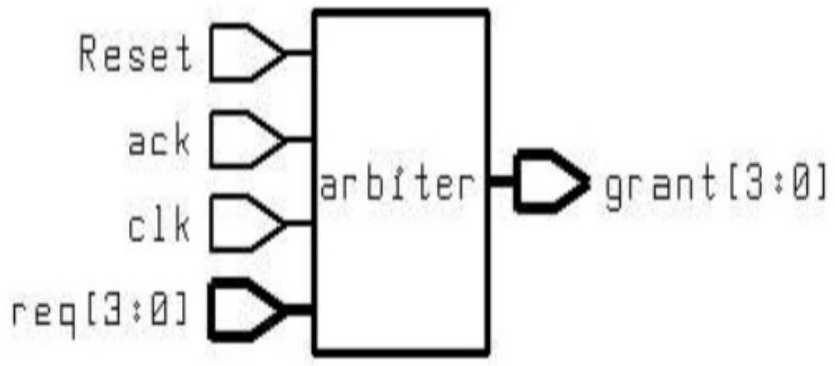



**Fig 1: Block Diagram for Four Bus Masters.**

## 2.THREE STEPS OF ARBITARATION:

**Step 1: Request.**

Each input sends a request to every output for which it has a queued cell.

**Step 2: Grant.**

If an output receives any requests, it chooses the one that appears next in a fixed, round robin schedule starting from the

highest priority element. The output notifies each input whether or not its request was granted. The pointer to the highest

priority element of the round-robin schedule is incremented (modulo to one location beyond the granted input.

**Step 3: Accept.**

If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer to the highest priority element of the round-robin schedule is incremented (modulo to one location beyond the accepted output .The requester that is pointed to by the round-robin pointer is shifted to the highest priority position, and the other requests are rotated in behind it. This rotated request vector is then sent. The grant vector from the priority arbiter is then “unrotated” to come up with the round-robin arbiter‟s final grant signal. This is shown in the block diagram.

A round-robin token passing bus or switch arbiter guarantees fairness (no starvation) among masters and allows any unused time slot to be allocated to a master whose round-robin turn is later but who is ready now. A reliable prediction of the worst-case wait time is another advantage of the round-robin protocol. The worst-case wait time is proportional to number of requestors minus one. The protocol of a round-robin token passing bus or switch arbiter works as follows. In each cycle, one of the masters (in round-robin order) has the highest priority (i.e., owns the token) for access to a shared resource.

If the token-holding master does not need the resource in this cycle, the master with the next highest priority who sends a request can be granted the resource, and the highest priority master then passes the token to the next master in round-robin order.

A round-robin token passing bus or switch arbiter guarantees fairness (no starvation) among masters and allows any unused time slot to be allocated to a master whose round-robin turn is later but who is ready now. A reliable prediction of the worst-case wait time is another advantage of the round-robin protocol. The worst-case wait time is proportional to number of requestors minus one. The protocol of a round-robin token passing bus or switch arbiter works as follows.

In each cycle, one of the masters (in round-robin order) has the highest priority (i.e., owns the token) for access to a shared resource. If the token-holding master does not need the resource in this cycle, the master with the next highest priority who sends a request can be granted the resource, and the highest priority master then passes the token to the next master in round-robin order.

In fig shows as ASM chat the working of proposed arbiter design. In this at single clock cycle itchecks for a request and assigns the grant to it and update the pointer. The pointer move in cyclic order as explained earlier.

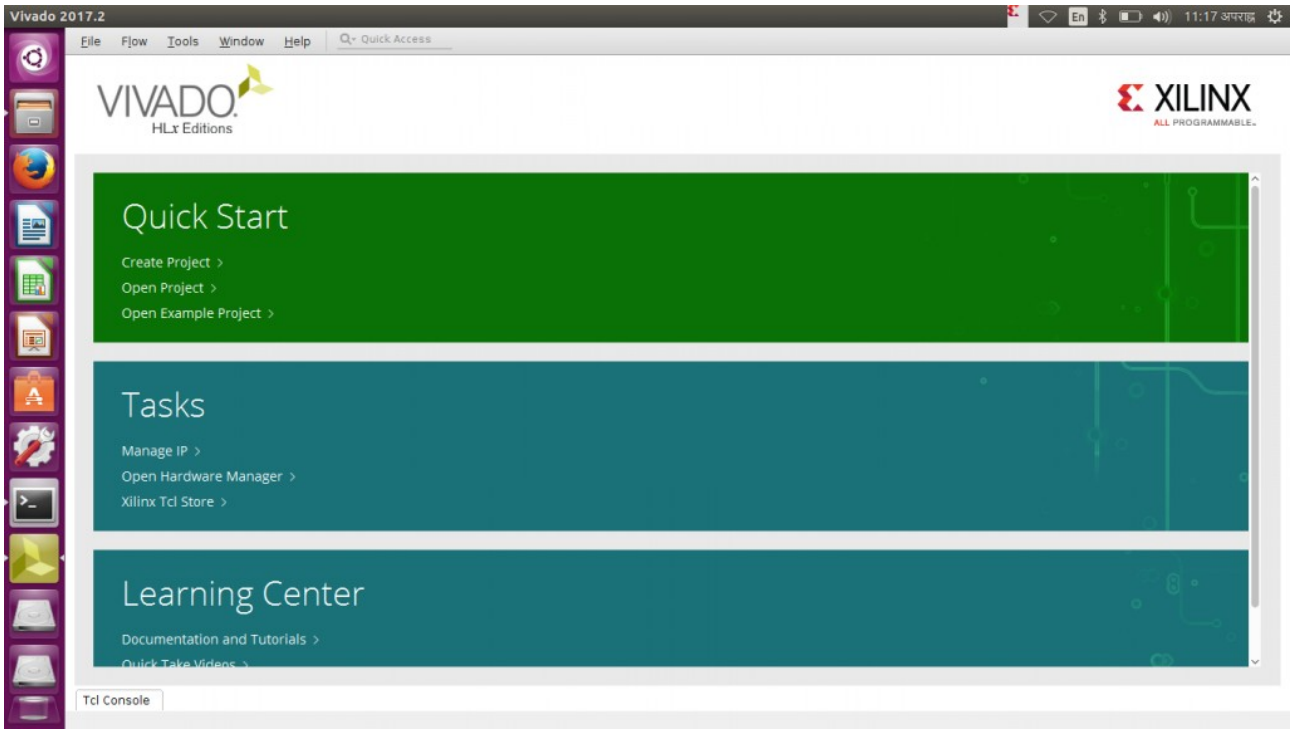
In the state diagram of the new arbiter the pointer is the extra port which is used. In single clock pulse pointer checks which requestor or user is asking for request it provide with the grant and update the pointer. Different cases is discusses for explaining the different condition of the arbiter

# 3. SOFTWARE USED

## 3.1 XILINX VIVADO

**Xilinx Vivado** is an advanced FPGA (Field Programmable Gate Array) design and analysis suite developed by **Xilinx** (now a part of AMD). It provides a comprehensive toolset for designing, simulating, and programming FPGAs, SoCs (System on Chips), and other hardware programmable devices. Vivado Design Suite is specifically built to replace the older ISE (Integrated Software

Environment) and to support the newer Xilinx FPGA architectures, especially the 7-series devices and beyond (e.g., UltraScale, UltraScale+, and Versal platforms).




**Key Features of Xilinx Vivado:**

1. **RTL Design and IP Integration**:
    - Vivado supports designing using **HDL** languages like **VHDL**, **Verilog**, and **SystemVerilog**.
    - It includes an **IP Integrator** that allows users to connect pre-built IP cores, including Xilinx's own IP, such as DSP cores, memory controllers, AXI bus interfaces, etc. The tool simplifies the integration of custom IP blocks with drag-and-drop interfaces.
2. **Block Design Tool**:
    - The **Block Design Tool** (BDT) enables users to visually design their hardware by connecting pre-verified IP blocks, similar to a schematic diagram. It’s very useful for designing complex systems like processors or SoC designs with peripherals and interconnects.
3. **Vivado High-Level Synthesis (HLS)**:

- High-Level Synthesis allows you to write C, C++, or SystemC code and convert it into hardware description language (HDL) for implementation in an FPGA. HLS enables quicker design exploration, validation, and development by abstracting away some of the low-level design complexity.

4. **Comprehensive Simulation and Debugging**:
    - **Vivado Simulator** supports detailed behavioral, post-synthesis, and post-implementation simulation, allowing users to verify and debug the behavior of their design before programming the hardware.
    - The **Integrated Logic Analyzer (ILA)** allows for on-chip debugging by inserting probes in the design that can capture and analyze internal signals in real-time.
5. **Synthesis and Implementation**:
    - **Vivado Synthesis** translates the high-level RTL design into a netlist of logic gates and performs optimizations for the specific Xilinx FPGA target architecture.
    - The **Implementation** stage handles placement and routing, ensuring that the design is efficiently mapped to the FPGA's logic elements, DSP blocks, BRAM, and interconnects.
6. **Tcl-based Scripting and Automation**:
    - Vivado uses **Tcl (Tool Command Language)** for automation, enabling designers to run repetitive tasks through scripts, automate design flows, and customize processes like project creation, synthesis, and bitstream generation.
7. **Vivado Design Checkpoints (DCP)**:
    - A **Design Checkpoint** is a snapshot of a partially implemented design. It allows users to save, review, and resume work from a specific point in the design flow, especially useful in large designs for modular or team-based development.
8. **Support for Xilinx FPGAs**:
    - Vivado supports the latest Xilinx FPGA families, such as:
        - **7-Series**: Artix-7, Kintex-7, Virtex-7

- **Zynq-7000 SoCs**: Combining ARM processors with programmable logic
- **UltraScale**: High-performance, low-power FPGAs
- **UltraScale+**: Advanced FPGAs and SoCs, targeting AI and 5G applications
- **Versal ACAP**: Adaptive Compute Acceleration Platforms, which combine programmable logic, CPUs, and specialized acceleration engines.

9. **Power Optimization**:
    - Vivado provides power analysis tools to estimate and optimize the power consumption of FPGA designs. This is particularly useful for power-sensitive applications like mobile devices or data centers.
10. **Partial Reconfiguration**:
    - Vivado supports **partial reconfiguration**, allowing a portion of the FPGA to be reconfigured while the rest of the FPGA remains operational. This is useful for applications that require hardware changes without powering down the entire system.

**Vivado Design Flow:**

The typical Vivado design flow includes the following steps:

1. **Create a New Project**:
    - Define the target FPGA device and set up the working environment.
2. **Design Entry**:
    - Design using HDL (Verilog, VHDL, or SystemVerilog) or use the **Block Design** tool for IP-based integration.
    - Alternatively, use **Vivado HLS** for high-level C/C++-based design.
3. **Synthesis**:
    - Convert the HDL or Block Design into a netlist format, optimize it, and check for design rules.
4. **Implementation**:
    - Place and route the design on the FPGA to map the logic, routing resources, and I/O pins.

5. **Simulation and Debugging**:
    - Simulate the design using the built-in simulator or third-party simulators like ModelSim.
    - Debug using tools like the **Integrated Logic Analyzer (ILA)** or **Virtual I/O (VIO)** to capture and analyze signals in real-time.
6. **Generate Bitstream**:
    - Create a bitstream that can be used to program the FPGA hardware.
7. **Programming and Configuration**:
    - Load the generated bitstream onto the FPGA or SoC device and verify its functionality.
8. **Post-Implementation**:
    - Perform timing analysis, power estimation, and any further optimizations.

**Vivado Editions:**

- **Vivado WebPACK**: A free version with limited functionality, supporting lower-end devices like the Artix-7 and Spartan series.
- **Vivado Design Edition**: A paid version with additional features such as advanced synthesis and implementation tools.
- **Vivado System Edition**: The most advanced edition, supporting all high-end features, including System Generator, HLS, and complete IP support.

**Applications of Vivado:**

- **Embedded Systems**: Using Zynq SoCs to create custom embedded systems with hardware and software co-design.
- **AI and Machine Learning**: Implementing AI inference engines on FPGAs using high-performance features like DSP slices and HBM memory.
- **Telecommunications**: Designing custom FPGA-based systems for 5G, optical networks, and high-speed data communications.
- **DSP and Signal Processing**: Optimizing signal processing algorithms on FPGAs for applications like radar, medical imaging, and video processing.

# 4.Design and Implementation

The design of the Round Robin Arbiter involves the following steps:

### 1. Request and Grant Signals:

The arbiter has multiple request inputs (e.g., R1, R2, R3 for three entities) and corresponding grant outputs (e.g., G1, G2, G3).

### 2. Cyclic Grant Logic:

A counter or state machine can be used to keep track of which entity's request should be granted next. For example, if there are three entities, the counter moves from 0 to 1 to 2, and then back to 0 after granting the access.

### 3. Edge Detection:

The arbiter continuously checks for active requests. If no requests are active, no grants are issued. When a request is made, the arbiter grants access based on the current counter/state value.

### 4. Implementation Details:

Input Signals: Request signals from entities, clock, and reset signals.

Output Signals: Grant signals, indicating which entity has been granted access.

Control Logic: A simple counter or state machine that tracks the next entity to be granted access.

# 5.SOURCE CODE

```
module arbiter (
  clk,
  rst,
  req3,
  req2,
  req1,
  req0,
  gnt3,
  gnt2,
  gnt1,
  gnt0
);
// PORT DECLARATION
input          clk;    //CLOCK
input          rst;    //RESET
input          req3;   //REQUEST SIGNALS
input          req2;
input          req1;
input          req0;
output          gnt3;   //GRANT SIGNALS
output          gnt2;
output          gnt1;
```

```
output         gnt0;

//INTERNAL REGISTERS
wire    [1:0]   gnt      ;
wire            comreq    ;
wire            beg       ;   // BEGIN SIGNAL
wire   [1:0]    lgnt      ;  // LATCHED ENCODED GRANT
wire            lcomreq   ;  // BUS STATUS
reg             lgnt0     ;  // LATCHED GRANTS
reg             lgnt1     ;
reg             lgnt2     ;
reg             lgnt3     ;
reg             mask_enable   ;
reg             lmask0    ;
reg             lmask1    ;
reg             ledge     ;

//--//

always @ (posedge clk)

  if (rst)  //if reset is true
  begin
    lgnt0 <= 0;
    lgnt1 <= 0;
    lgnt2 <= 0;
```

```
  lgnt3 <= 0;
end

else
  begin
lgnt0 <=(~lcomreq & ~lmask1 & ~lmask0 & ~req3 & ~req2 & ~req1 & req0)
      | (~lcomreq & ~lmask1 &  lmask0 & ~req3 & ~req2 &  req0)
      | (~lcomreq &  lmask1 & ~lmask0 & ~req3 &  req0)
      | (~lcomreq &  lmask1 &  lmask0 & req0  )
      | ( lcomreq & lgnt0 );

lgnt1 <=(~lcomreq & ~lmask1 & ~lmask0 &  req1)
      | (~lcomreq & ~lmask1 &  lmask0 & ~req3 & ~req2 &  req1 & ~req0)
      | (~lcomreq &  lmask1 & ~lmask0 & ~req3 &  req1 & ~req0)
      | (~lcomreq &  lmask1 &  lmask0 &  req1 & ~req0)
      | ( lcomreq &  lgnt1);

lgnt2 <=(~lcomreq & ~lmask1 & ~lmask0 &  req2  & ~req1)
      | (~lcomreq & ~lmask1 &  lmask0 &  req2)
      | (~lcomreq &  lmask1 & ~lmask0 & ~req3 &  req2  & ~req1 & ~req0)
      | (~lcomreq &  lmask1 &  lmask0 &  req2 & ~req1 & ~req0)
      | ( lcomreq &  lgnt2);

lgnt3 <=(~lcomreq & ~lmask1 & ~lmask0 & req3  & ~req2 & ~req1)
      | (~lcomreq & ~lmask1 &  lmask0 & req3  & ~req2)
      | (~lcomreq &  lmask1 & ~lmask0 & req3)
```

```
        | (~lcomreq &  lmask1 &  lmask0 & req3  & ~req2 & ~req1 & ~req0)
        | ( lcomreq & lgnt3);
end

 //BEGIN SIGNAL

  assign beg = (req3 | req2 | req1 | req0) & ~lcomreq;



 // comreq logic (BUS STATUS)

assign lcomreq = ( req3 & lgnt3 )
                | ( req2 & lgnt2 )
                | ( req1 & lgnt1 )
                | ( req0 & lgnt0 );
// Encoder logic

assign  lgnt =  {(lgnt3 | lgnt2),(lgnt3 | lgnt1)};



// lmask register.

always @ (posedge clk )
if( rst )
  begin
    lmask1 <= 0;
```

```
    lmask0 <= 0;
  end
 else if(mask_enable)
    begin
      lmask1 <= lgnt[1];
      lmask0 <= lgnt[0];
    end
 else
    begin
      lmask1 <= lmask1;
      lmask0 <= lmask0;
    end


assign comreq = lcomreq;
assign gnt    = lgnt;

// Drive the outputs

assign gnt3   = lgnt3;
assign gnt2   = lgnt2;
assign gnt1   = lgnt1;
assign gnt0   = lgnt0;

endmodule
```

**TEST BENCH CODE :**

```
module top ();

reg          clk;
reg          rst;
reg          req3;
reg          req2;
reg          req1;
reg          req0;
wire          gnt3;
wire          gnt2;
wire          gnt1;
wire          gnt0;

// Clock generator
always #1 clk = ~clk;

initial begin
  $dumpfile ("arbiter.vcd");
  $dumpvars();
  clk = 0;
  rst = 1;
  req0 = 0;
  req1 = 0;
  req2 = 0;
```

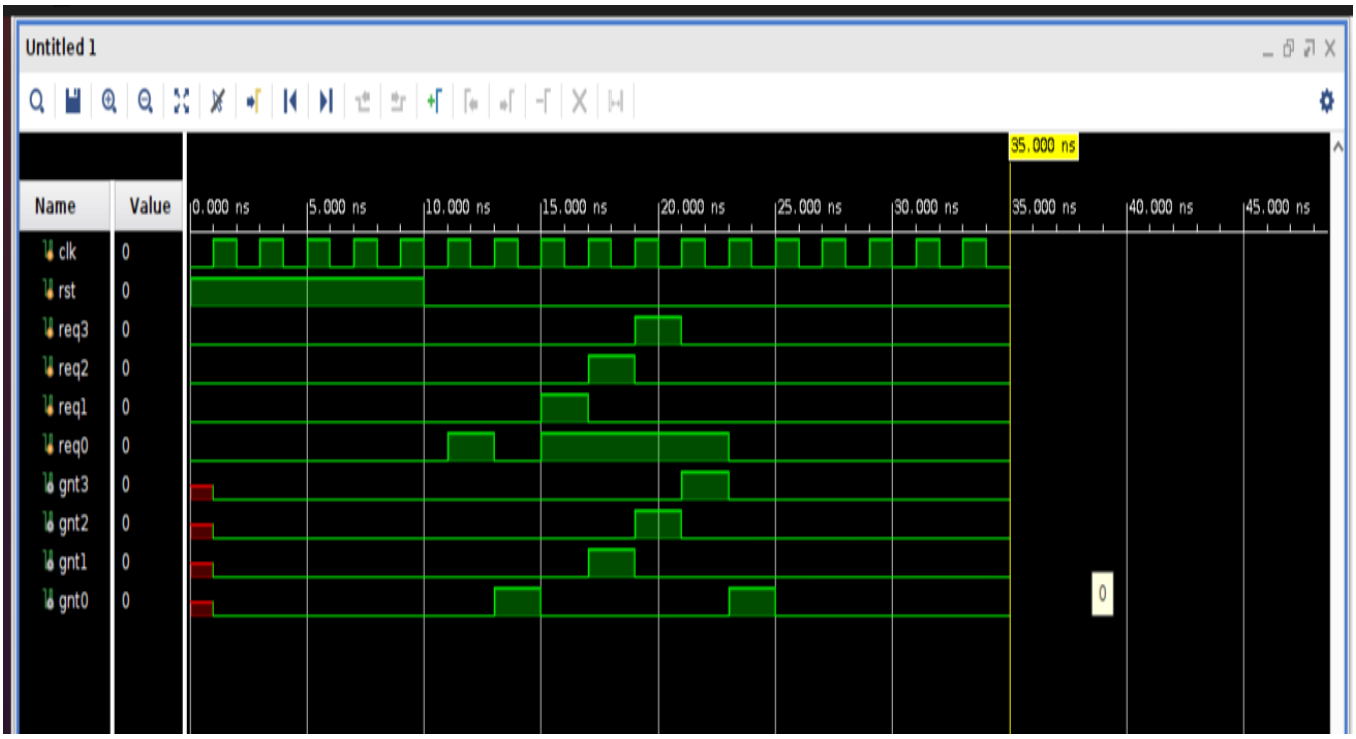
```
  req3 = 0;
  #10 rst = 0;
  repeat (1) @ (posedge clk);
  req0 <= 1;
  repeat (1) @ (posedge clk);
  req0 <= 0;
  repeat (1) @ (posedge clk);
  req0 <= 1;
  req1 <= 1;
  repeat (1) @ (posedge clk);
  req2 <= 1;
  req1 <= 0;
  repeat (1) @ (posedge clk);
  req3 <= 1;
  req2 <= 0;
  repeat (1) @ (posedge clk);
  req3 <= 0;
  repeat (1) @ (posedge clk);
  req0 <= 0;
  repeat (1) @ (posedge clk)
  #10 $finish;
end

// Connect the DUT
arbiter U (
 clk,
```

```
 rst,
 req3,
 req2,
 req1,
 req0,
 gnt3,
 gnt2,
 gnt1,
 gnt0
);

endmodule
```

# 6. WAVEFORMS



## 7.Simulation & Testing

The Round Robin Arbiter was simulated in [ModelSim/Vivado] to evaluate its functionality and verify its performance under different scenarios:

**Scenario 1:** Simultaneous Requests

All entities request access simultaneously. The arbiter grants access to the first requester and then continues granting access in a cyclic manner.

**Scenario 2:** Single Request

Only one entity makes a request. The arbiter grants access immediately.

**Scenario 3:** No Requests

No entity requests the resource, and the arbiter grants no access.

The simulation confirmed that the arbiter granted access in a round-robin fashion, with each entity receiving a fair turn, regardless of the order of arrival of requests.

# 8. Performance Analysis

## 8.1Advantages of Round Robin Arbitration:

**Fairness:** Every requesting entity receives a turn in the same order, ensuring no entity is favored over another.

**Simplicity:** The Round Robin scheme is easy to implement with minimal hardware resources (e.g., counters or state machines).

**No Starvation:** Each request gets serviced in a cyclic order, eliminating the risk of starvation.

## 8.2 Key Characteristics:

**Fairness:** Each requesting entity gets access in a rotating manner, preventing starvation.

**No Priority:** All entities are treated equally; the order of granting access is only dependent on the request cycle.

**Simplicity:** Round Robin is easy to implement with a straightforward state machine or counter-based design.

## 9.CONCLUSION :

The Round Robin Arbiter provides an effective and fair mechanism for resource allocation in digital systems. By cyclically granting access to entities, the system ensures fairness, minimizes starvation, and simplifies design. This project successfully demonstrated the functionality of a Round Robin Arbiter through design and simulation. Future work could explore more advanced arbitration techniques such as Weighted Round Robin, which introduces varying priorities to different entities, or enhancements to reduce latency in high-traffic systems.

# 10.REFERENCES

[1]. Ge, J., “Cost-effective Buffered Wormhole Routing”. Lecture Notes in Computer Science, Springer-Verlag, Heidelberg,

Germany, Vol. 3358, pp666-676, 2004.

[2]. Palnitkar, S., “Verilog HDL: A Guide to Digital Design and Synthesis”. SunSoft Press, Palo Alto, CA, 1996.

[3]. Nabeel Al-saber, Saurab Oberoi, Roberto Rojas-Cessa andSotirios G. Ziavras, "Concatenating Packets for Variable-Length

Input-Queue Packet Switches with Cell-Basedand Packet-Based Scheduling," Proc. IEEE Sarnoff Symposium, Princeton, NJ,

April 28-30, 2008.

[4]. M.J. Karol, M. G. Hluchyj, and S.P. Morgan, “Input Versus Output Queuing on a Space-Division Packet Switch,” IEEE

Transactions on Communications, 35:1347-56, 1997.

[5]. T. Anderson, S. Owicki, J. Saxe, and C. Thacker, “High Speed Switch Scheduling for Local Area Networks,” ACM Trans.

Comput. Syst., pp. 319-52, Nov. 1993.

[6]. M. A. Marsan, A. Bianco, E. Leonardi, and L. Milia, “RPA: A Flexible Scheduling Algorithm for Input Buffered Switches,”

IEEE Transactions on Communications, 47: 1921-33, Dec.1999.

[7]. N. Mckeown, A. Mekkittikul, V. Anantharam, and J. Walrand, “Achieving 100% Throughput in an Input-Queued Switch,” IEEE

Transactions on Communications, 47: 1260-67, Aug. 1999.

[8]. A. Mekkittikul, and N. Mckeown, “A Practical Scheduling Algorithm to Achieve 100% Throughput in Input-Queued Switches,”