Project Report: N-Way Round-Robin Arbiter for GPU Shared Memory

Introduction

The increasing demand for high-performance computing in fields such as video game rendering and scientific simulations has driven the development of massively parallel architectures like GPUs. Within a GPU's Streaming Multiprocessor (SM), multiple threads execute in parallel to perform compute-intensive tasks. These threads often need to access a shared, fast on-chip memory known as shared memory for temporary data and inter-thread communication. A significant challenge arises when multiple threads simultaneously request access to this single-banked shared memory, creating a resource contention problem.

This report details the design and implementation of an N-way round-robin arbiter, a hardware-based solution to manage this contention. The arbiter's primary function is to ensure fair, non-blocking access to the shared memory bank, which is critical for maximizing the efficiency and predictability of GPU-accelerated applications. By using a round-robin approach, the arbiter prevents any single thread or process from monopolizing the shared resource, a condition known as starvation, thereby maintaining a high degree of temporal equality in resource sharing.

Theory and Arbitration

N-way round-robin arbiter is a digital logic circuit designed to manage requests from N different sources for a single shared resource. Its core principle is to grant access in a sequential, cyclic order, ensuring fairness and preventing starvation.

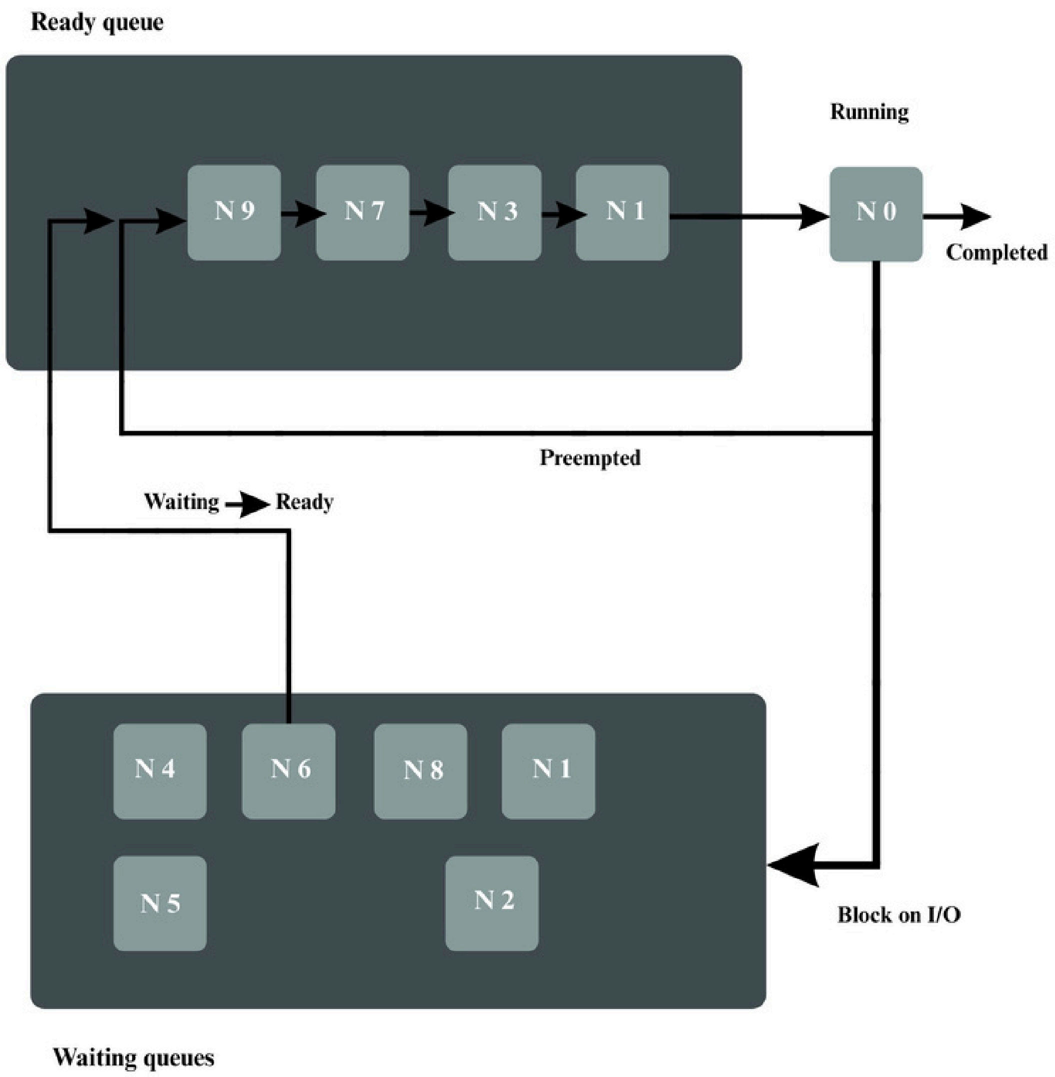
Key Principles:

- **ArbitrationDecision** : In each arbitration cycle, the arbiter grants access to only one requestor from the N competing requesters.

- **Round-Robin Sequence**: The arbiter maintains an internal pointer that tracks which requestor was last served. In the next cycle, it starts its search for an active requestor from the next position in the sequence. If a requestor in line does not have an active request, the arbiter simply skips them and moves to the next one in the round. The sequence then wraps back to the first requestor, creating a continuous cycle.

- **Fairness and Starvation Prevention**: This cyclic mechanism guarantees that every requestor gets a periodic opportunity to access the resource, as no single requestor can monopolize it. This is a crucial feature for parallel computing, as it ensures that all threads can make progress.

- **N-Way Extension**: The design is scalable and can handle any number of requesters, N. The logic remains simple regardless of N, as it primarily involves tracking a pointer and iterating through the requesters in a modular fashion.

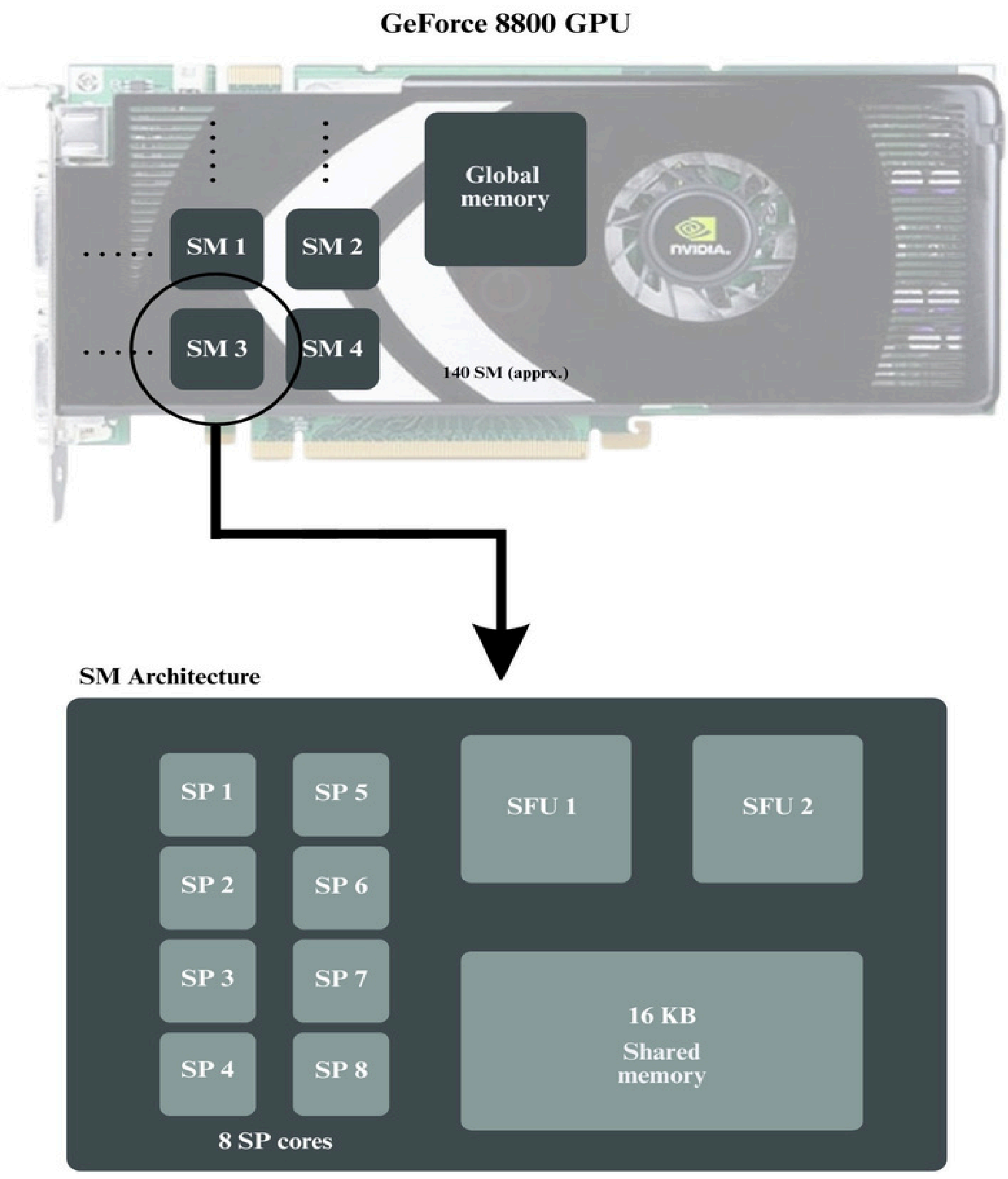


An example with three requesters (Req0, Req1, Req2) illustrates this process:

1. **Cycle 1**: Req0 is granted access.
2. **Cycle 2**: The pointer advances, and Req1 is granted.
3. **Cycle 3**: The pointer moves, and Req2 is granted.
4. **Cycle 4**: The cycle repeats, and the pointer wraps back to Req0, which is granted again.

Design Diagram

This project's design can be understood by examining its application within a GPU architecture and the underlying state machine concepts.



GPU Architecture

The arbiter is implemented within a GPU's Streaming Multiprocessor (SM). As shown in the diagram, a GPU can have numerous SMs (e.g., SM1, SM2, etc.). Each SM contains multiple Streaming Processors (SP cores) and a bank of **shared memory**. The round-robin arbiter is the critical component that sits between these multiple SP cores and the single shared memory bank, managing their access requests.

Process Scheduling and Resource Contention

The need for a round-robin arbiter is analogous to managing a **ready queue** in an operating system. Threads (represented as blocks like N1, N3, etc.) are in a "Ready" state, waiting for their turn to be scheduled onto a processor. When multiple threads from the ready queue simultaneously try to access a shared resource (like shared memory), an arbiter is needed to decide which one gets to run. The round-robin approach ensures that a thread waiting in a "Waiting" queue due to resource contention will eventually get its turn, preventing it from being preempted indefinitely.

Research Insights

The concept of a round-robin arbiter is well-established in hardware design, especially in parallel computing and networking. Academic papers highlight its importance for achieving fairness and scalability.

- **RAM Arbiter Architectures**: Research on RAM arbiter architectures confirms that round-robin schemes are widely used for managing access to a single RAM module from multiple systems or ports. These papers emphasize that round-robin arbitration provides a predictable and fair way to handle resource contention, preventing any single client from monopolizing the memory bus.
- **High-Speed Arbitration**: In high-speed systems like network switches and routers, advanced versions like **Parallel Round-Robin Arbiters (PRRA)** and **Improved Parallel Round-Robin Arbiters (IPRRA)** are employed. These designs are optimized for speed and fairness, demonstrating that the fundamental round-robin principle is a robust foundation for more complex, high-performance arbitration schemes. The simplicity and scalability of the round-robin logic make it an ideal choice for hardware implementation, as it avoids complex priority calculations that can add latency.
- **Application in Modern Systems**: The principles of round-robin arbitration are not limited to GPUs; they are applied in various parallel computing contexts, including multiprocessor systems and NoC (Network-on-Chip) designs, where multiple agents must share common resources.

HDL Implementation

Implemented this project in ModelSim version 20.1.

The N-way round-robin arbiter was implemented using VHDL (Very High-Speed Integrated Circuit Hardware Description Language) to define its behavior. The code defines an entity arbiter with a generic parameter N for the number of requestors.

Arbiter.vhd:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


entity arbiter is

   generic(

      N : integer := 4 -- Number of requestors

   ); port

   (    clk

   rst          : in std_logic;

                : in std_logic;

      enable : in std_logic;

      mask     : in std_logic_vector(N-1 downto 0);

      req      : in std_logic_vector(N-1 downto 0);

      grant    : out std_logic_vector(N-1 downto 0)

   );

end entity arbiter;


architecture Behavioral of arbiter is

   signal pointer    : integer range 0 to N-1 := 0;

   signal grant_reg : std_logic_vector(N-1 downto 0) := (others => '0');
```

```
begin
    process(clk, rst)
    begin
        if rst = '1' then
            pointer <=0;
            grant_reg<=(others => '0');
        elsifrising_edge(clk) then
            ifenable='1'then
                grant_reg<=(others => '0');
                foriin0toN-1 loop
                    ifreq((pointer + i) mod N) = '1' and mask((pointer + i) mod N) = '0' then
                        grant_reg((pointer + i) mod N) <= '1';
                        pointer<= ((pointer + i + 1) mod N);
                        exit;
                    end if;
                end loop;
            else
                grant_reg<=(others => '0');
            end if;
        end if;
    end process;

    grant <= grant_reg;

endarchitectureBehavioral;
```

Arbiter_tb.vhd

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


entity arbiter_tb is

end entity;


architecture tb of arbiter_tb is

    constant N : integer := 4;

    signal clk      : std_logic := '0';

    signal rst      : std_logic := '0';

    signal enable : std_logic := '1';

    signal mask     : std_logic_vector(N-1 downto 0) := (others => '0');

    signal req      : std_logic_vector(N-1 downto 0) := (others => '0');

    signal grant    : std_logic_vector(N-1 downto 0);


    type int_vec is array (0 to N-1) of integer;

    signal grant_count : int_vec := (others => 0);


    constant clk_period : time := 10 ns;


begin


    uut: entity work.arbiter
    generic map (N => N)
    port map (
        clk      => clk,
        rst      => rst,
        enable => enable,
        mask     => mask,
        req      => req,
        grant    => grant
    );

clk_process : process
begin
    while now < 400 ns loop
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end loop;
    wait;
end process;

stim_proc: process
begin
    rst <= '1'; wait for 12 ns;
    rst <= '0'; wait for 12 ns;
```

```
-- Simple request, no mask

req <= "1000"; wait for 15 ns;


--Multi-request,somemasked,enable ON

req<="1100";mask<="1000";wait for 15 ns;


--Allrequestorsactive,enableON, then OFF

req<=(others=>'1');mask<=(others => '0'); enable <= '1'; wait for 30 ns;

enable<='0';waitfor16ns;enable <= '1';


--Removedgrantholdfeaturehere


-- Edge case: all masked

mask<=(others=>'1');waitfor8ns; mask <= (others => '0');


--Randomizedrequestsandmasks for coverage

for i in 0 to 8 loop

   req<=std_logic_vector(to_unsigned(i, N));

   mask<=std_logic_vector(to_unsigned(i * 3, N));

   wait for 10 ns;

end loop;


-- Test reset again

rst<='1';waitfor8ns;rst<='0';wait for 12 ns;


-- Single requester, then none
```

```
    req<="0001";mask<=(others => '0'); wait for 16 ns;
    req<="0000";waitfor6ns;


    wait for 24 ns;
    wait;
end process;

cov_process: process(clk)
begin
    if rising_edge(clk) then
        for i in 0 to N-1 loop
            if grant(i) = '1' then
                grant_count(i)<=grant_count(i) + 1;
            end if;
        end loop;
    end if;
end process;

assert_onehot: process(clk)
    variable sum : integer;
begin
    ifrising_edge(clk)andnow > 30 ns then
        sum := 0;
        for i in 0 to N-1 loop
            ifgrant(i)='1'thensum := sum + 1; end if;
        end loop;
```

```
            assert(sum=1orsum= 0)
               report"ERROR:Grant not one-hot or all-zero!" severity error;
            for i in 0 to N-1 loop
               ifgrant(i)='1'and(mask(i) = '1') then
                  assertfalsereport"ERROR: Grant given to masked requestor!" severity error;
               end if;
            end loop;
         end if;
      end process;


      end_print: process
      begin
         wait for 390 ns;
         report"========GrantCoverage ========";
         for i in 0 to N-1 loop
            report"Requestor"&integer'image(i) & ": " & integer'image(grant_count(i)) & "
grants.";
         end loop;
         wait;
      end process;


endarchitecture tb;
```

Code Description (`arbiter.vhd`)

The core logic resides within a single synchronous process, triggered by a clock (

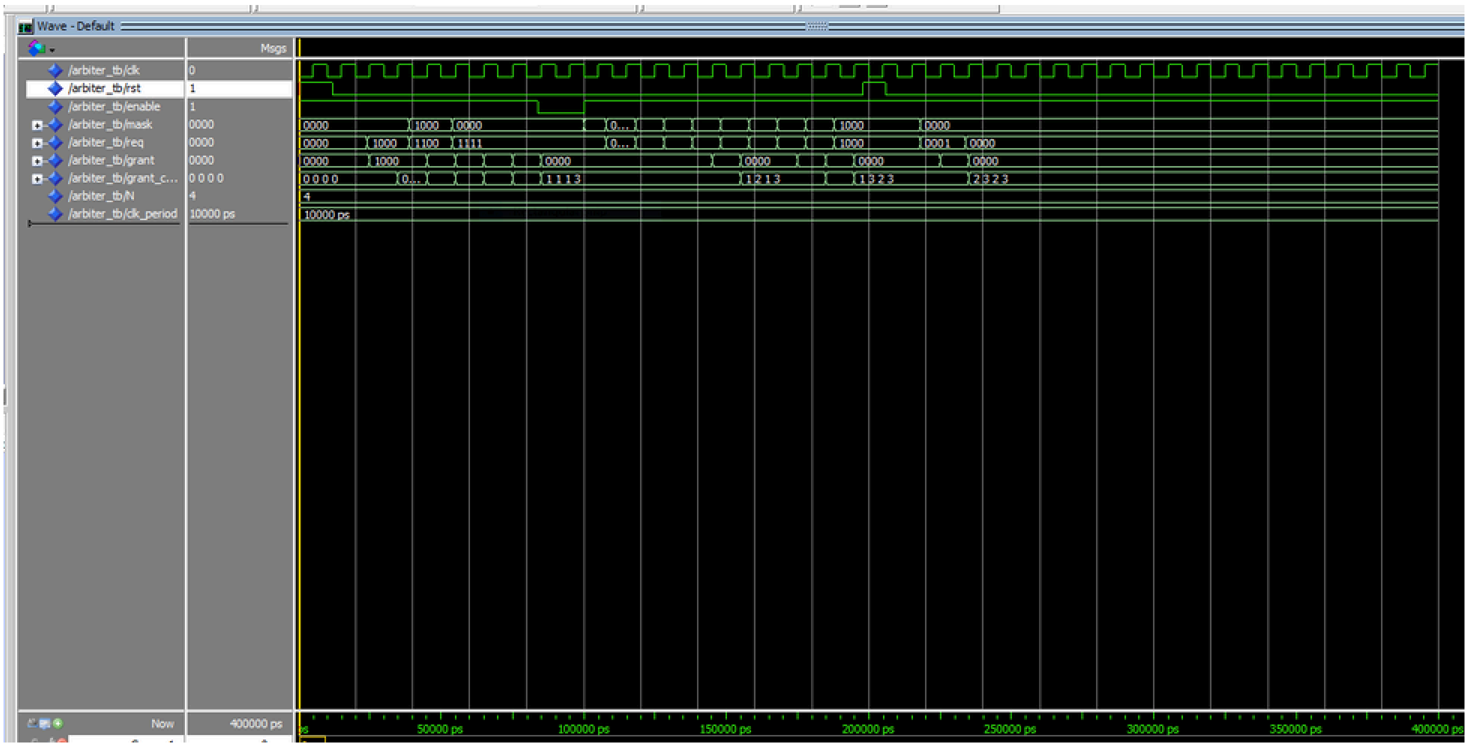
clk) and a reset (rst) signal.

- **Inputs**: The arbiter takes clk, rst, an enable signal, a mask vector to disable specific requestors, and a req vector representing the N requests.
- **Outputs**: It provides a grant vector, where only one bit can be high at any time, indicating which requestor has been granted access.
- **Internal State**: A pointer variable tracks the last granted requestor, and a grant_reg signal holds the output value.
- **Logic**:
  1. Upon a rising clock edge, if enable is high, the arbiter iterates through all possible requesters, starting from the current pointer position.
  2. It uses the modulo operator (pointer + i) mod N to create a circular, round-robin search.
  3. When it finds the first unmasked and active request (req = '1'), it sets the corresponding bit in grant_reg to '1'.
  4. The pointer is then updated to point to the requestor immediately after the granted one, ensuring the next cycle starts from the next-in-line.
  5. The exit statement ensures that only a single grant is issued per cycle, as required by the shared memory single-bank architecture.

Simulation and Results:

The functionality of the arbiter was verified using a testbench (arbiter_tb.vhd) and simulated with a ModelSim waveform viewer. The simulation confirmed that the arbiter operates as expected, granting requests in a sequential, round-robin fashion.

The provided waveform image shows the key signals and their behavior:



- **clk** and **rst**: The clock signal is active, and the reset signal is asserted briefly at the beginning of the simulation to initialize the arbiter's state.
- **req**: The request vector shows several test cases. At 80ns, all four requesters (req <= "1111") are active.
- **grant**: In response to the req signals, the grant signal shows a cyclic pattern. When all requesters are active, the grant is given to Req0, then Req1, then Req2, and then Req3, and the cycle repeats. This visually confirms the round-robin mechanism.
- **pointer**: The pointer signal clearly increments with each grant, demonstrating that the arbiter is correctly tracking the last granted requestor and updating its state for the next cycle.
- **grant_count**: The testbench also tracks the number of grants per requestor. The final report shows that each requestor received a roughly equal number of grants over the simulation, confirming the fairness of the design.

Application & Reflection

Application: This N-way round-robin arbiter is a fundamental building block in parallel computing systems. Its most direct application is within a **GPU's Streaming Multiprocessor (SM)**, where it manages access to the shared memory from multiple threads or warps. This is crucial for performance in GPU-accelerated tasks like video game rendering, scientific simulations, and machine learning, where thousands of threads might concurrently try to access shared data. Beyond GPUs, similar arbiters are used in:

- **Multiprocessor Systems**: For multiple CPU cores to access shared caches or memory banks.
- **Network-on-Chip (NoC)**: To manage packet traffic and resource allocation between different processing units on a single chip.
- **Bus Arbitration**: For multiple devices on a system bus to get a turn to transmit data.

Conclusion and Reflection:

Designing and implementing this arbiter reinforced the importance of synchronous digital logic and the use of state machines for controlling complex processes. I learned how to translate a theoretical concept (round-robin) into a practical, cycle-accurate hardware description using VHDL. The simulation and testbench development process highlighted the value of robust verification, ensuring that the design not only works but also adheres to critical constraints like one-hot granting and masking.

Possible Improvements:

While the current design is functional, several improvements could be explored:

- **Scalability**: For a very large number of requestors, a tree-based or hierarchical arbitration scheme could reduce the critical path latency compared to a simple loop.
- **Fairness with Different Priorities**: The design could be extended to a priority-based round-robin system, where certain requestors receive a higher chance of being granted access without being completely starved.
- **Low-Latency Implementation**: For extremely high-frequency applications, the arbiter could be optimized to minimize the clock-to-grant delay. This might involve exploring more complex combinational logic or pipelined architectures.

References:

NVIDIA Tesla: A Unified Graphics and Computing Architecture

https://ieeexplore.ieee.org/document/4523358