**MEMORY** **Controller**

A **memory controller** is the brain behind **seamless data transfer** between a processor and external memory devices like **SRAM, DRAM, Flash, or DDR memory**. This digital circuit not only manages data flow but also tackles timing constraints, arbitration, and optimizes memory access. Think of it as the unsung hero behind your system's performance!

*Acts as a "traffic cop" for memory access, ensuring efficient and reliable data transfers*.

**Why Should You Care About Memory Controllers?**

* Bridges the **speed gap** between processor and memory.
* Handles **synchronization and latency** with finesse.
* Crucial in **high-speed computing, embedded systems, and VLSI architectures**.
* Found in almost **every modern computing system**, from **CPUs and GPUs** to **SoCs and FPGAs**.

|  |  |
| --- | --- |
| Type | Use Case |
| SRAM Controllers | For cache and high-speed storage. |
| DRAM Controllers | Handles larger memory with refresh cycles. |
| Flash Memory Controllers | Found in SSDs and embedded storage devices. |
| DDR Memory Controllers | High-speed DDRx interfaces for data transfer. |
| High-Speed Bus Controllers | Embedded in AXI/AHB for SoC applications. |

**MEMORY INTERFACE**

Memory interfacing is the process of connecting a digital system (e.g., CPU, GPU, FPGA) to a memory device (e.g., SRAM, DRAM, Flash) and managing the flow of data between them. It involves **address decoding**, **timing control**, and **data transfer protocols**.

* **Address Decoding :** The processor generates memory addresses to access specific memory locations. If the processor has a K-bit address bus, it can address up to 2K,536 locations (64 KB). Then, **Address decoding** maps these addresses to physical memory regions.
* **Read/Write Operations :** Efficient reading and writing depend on satisfying timing constraints.

Timing Parameters:

tRC (Row Cycle Time): Minimum time between consecutive row accesses.

tRCD (RAS to CAS Delay): Time between activating a row and accessing a column.

tCL (CAS Latency): Delay between issuing a read command and receiving data.

tWR (Write Recovery Time): Time after a write operation before a new command can be issued.

|  |  |
| --- | --- |
| READ OPERATION | WRITE OPERATION |
| 1. Activate row (assert RAS). 2. Wait for tRCD. 3. Access column (assert CAS). 4. Wait for tCL to receive data. | 1. Activate row (assert RAS). 2. Wait for tRCD. 3. Access column (assert CAS) and provide data. 4. Wait for tWR before next operation. |

**MULTI-PORT MEMORY HANDLING(Arbitration Techniques)**

When multiple components (like processors or peripherals) access memory simultaneously, a memory controller ensures data integrity and fair usage:

|  |  |  |  |
| --- | --- | --- | --- |
| Arbitration Type | Fairness | Complexity | Use Cases |
| Fixed Priority | ❌ Low | ⭐ Simple | Real-time systems with critical tasks |
| Round-Robin | ✅ High | ⭐⭐ Moderate | Multi-core CPUs, memory controllers |
| Time-Sliced | ✅ High | ⭐⭐ Moderate | Systems with strict timing constraints |

**Fixed Priority Arbitration**

In fixed priority, one master always has the highest priority. If multiple requests come at the same time, the one with the highest priority gets access first.

module fixed\_priority\_arbiter (

input wire clk, reset,

input wire req1, req2, req3,

// Memory access requests

output reg grant1, grant2, grant3

// Granted access

);

always @(posedge clk or posedge reset) begin

if (reset) begin

grant1 <= 0;

grant2 <= 0;

grant3 <= 0;

end else begin

// Fixed priority: req1 > req2 > req3

if (req1) begin

grant1 <= 1; grant2 <= 0; grant3 <= 0;

end else if (req2) begin

grant1 <= 0; grant2 <= 1; grant3 <= 0;

end else if (req3) begin

grant1 <= 0; grant2 <= 0; grant3 <= 1;

end else begin

grant1 <= 0; grant2 <= 0; grant3 <= 0;

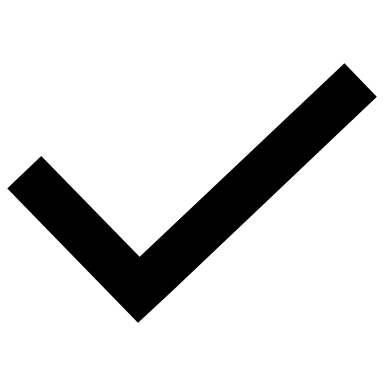
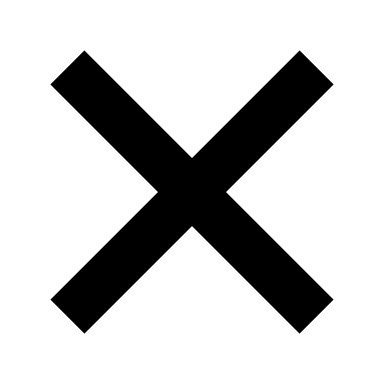
end

end

end

endmodule

**Pros & Cons**

Simple to implement  
 Lower priority requests may starve

**Round-Robin Arbitration**

In **round-robin arbitration**, each request gets a turn in a circular order, preventing starvation.

module round\_robin\_arbitration (

input wire clk,

input wire rst,

input wire [3:0] req, // 4 request lines

output reg [3:0] grant // 4 grant lines );

reg [1:0] pointer; // Tracks the current priority always @(posedge clk or posedge rst) begin

if (rst) begin

pointer <= 2'b00; // Reset the pointer

grant <= 4'b0000; // Reset all grants

end else begin

grant <= 4'b0000; // Default: no request

case (pointer)

2'b00: if (req[0]) grant <= 4'b0001;

else if (req[1]) grant <= 4'b0010;

else if (req[2]) grant <= 4'b0100;

else if (req[3]) grant <= 4'b1000;

2'b01: if (req[1]) grant <= 4'b0010;

else if (req[2]) grant <= 4'b0100;

else if (req[3]) grant <= 4'b1000;

else if (req[0]) grant <= 4'b0001;

2'b10: if (req[2]) grant <= 4'b0100;

else if (req[3]) grant <= 4'b1000;

else if (req[0]) grant <= 4'b0001;

else if (req[1]) grant <= 4'b0010;

2'b11: if (req[3]) grant <= 4'b1000;

else if (req[0]) grant <= 4'b0001;

else if (req[1]) grant <= 4'b0010;

else if (req[2]) grant <= 4'b0100; endcase

// Update the pointer to rotate priorities

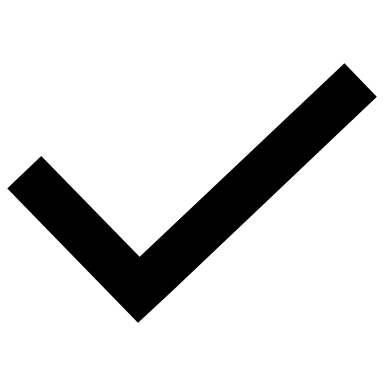
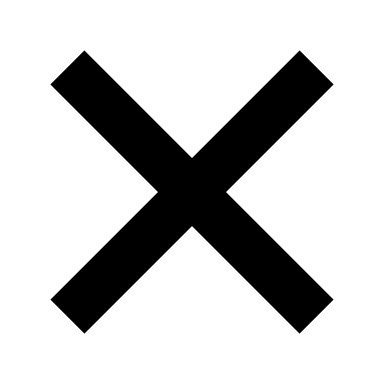
if (|grant) begin // Only rotate if a grant

pointer <= pointer + 2'b01;

end end end

endmodule

**Pros & Cons**

 Ensures fairness  
 Higher latency

**Time-Sliced Arbitration**

In **time-sliced arbitration**, each request gets access for a fixed time slice before moving to the next.  
module time\_sliced\_arbiter (

input wire clk, reset,

input wire req1, req2, req3,

output reg grant1, grant2, grant3

);

reg [3:0] counter;

// Time slice counter

reg [1:0] state;

always @(posedge clk or posedge reset) begin if (reset) begin

counter <= 0;

state <= 2'b00;

grant1 <= 0;

grant2 <= 0;

grant3 <= 0;

end else begin

counter <= counter + 1;

if (counter == 4'b1111) begin

// Change every 16 cycles

counter <= 0;

state <= state + 1;

end

case (state)

2'b00: begin grant1 <= req1; grant2 <= 0; grant3 <= 0; end

2'b01: begin grant1 <= 0; grant2 <= req2; grant3 <= 0; end

2'b10: begin grant1 <= 0; grant2 <= 0; grant3 <= req3; end

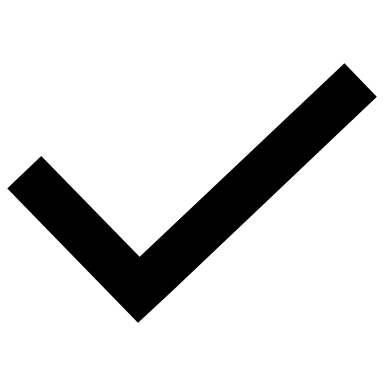
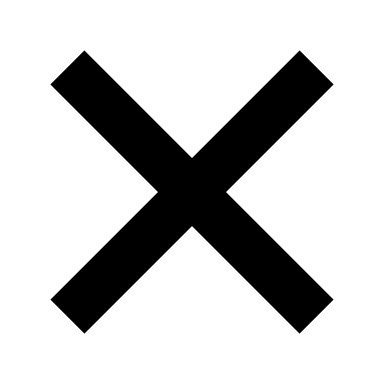
default: state <= 2'b00;

endcase

end

end

endmodule  
**Pros & Cons**

 Ensures each request gets fair access  
 May waste cycles if no request is present

**Advanced Design Consideration**

* **Handling Parallel Requests**: Use FIFO buffers, Round Robin arbitration.
* **Latency & Bandwidth Optimization**: Introduce burst transfers and pipelining.
* **Power Efficiency**: Employ techniques like dynamic frequency scaling.
* **Data Integrity**: Protect with **Error Correction Codes (ECC)** such as:
  + **Hamming Codes**: Corrects single-bit errors.
  + **Reed-Solomon Codes**: Corrects burst errors.

**//HAMMING CODE**

module hamming\_encoder (

input [3:0] data\_in, // 4 data bits

output [6:0] code\_out // 7 Hamming code bits

);

assign code\_out[0] = data\_in[0] ^ data\_in[1] ^ data\_in[3]; // P1

assign code\_out[1] = data\_in[0] ^ data\_in[2] ^ data\_in[3]; // P2

assign code\_out[2] = data\_in[0]; // D1

assign code\_out[3] = data\_in[1] ^ data\_in[2] ^ data\_in[3]; // P4

assign code\_out[4] = data\_in[1]; // D2

assign code\_out[5] = data\_in[2]; // D3

assign code\_out[6] = data\_in[3]; // D4

endmodule

module hamming\_decoder (

input [6:0] code\_in, // 7 Hamming code bits

output [3:0] data\_out, // 4 data bits

output reg error // Error flag

);

wire p1, p2, p4;

assign p1 = code\_in[0] ^ code\_in[2] ^ code\_in[4] ^ code\_in[6];

assign p2 = code\_in[1] ^ code\_in[2] ^ code\_in[5] ^ code\_in[6];

assign p4 = code\_in[3] ^ code\_in[4] ^ code\_in[5] ^ code\_in[6];

wire [2:0] syndrome = {p4, p2, p1}; // Error location index

always @(\*)

begin

if (syndrome != 3'b000) error = 1; // Error detected

else error = 0;

end

assign data\_out ={code\_in[6], code\_in[5],code\_in[4],code\_in[2]};

endmodule

USES

 Corrects **single-bit errors** and detects two-bit errors.

 Ideal for **memory systems** like RAM (error correction in hardware).

 Used in **low-complexity devices** such as sensors and embedded systems.

 Protects **data storage** in magnetic drives and communication protocols.

 Lightweight, perfect for systems with **limited computational resources**.

**//REED-SOLOMON CODES**

module reed\_solomon\_encoder (

input [7:0] data\_in, // 8-bit data

output [11:0] code\_out // 8 data bits + 4 parity bits

);

reg [3:0] parity; // Parity bits

always @(\*) begin

// Simplified parity calculation (example polynomial)

parity[0] = data\_in[0] ^ data\_in[1] ^ data\_in[2];

parity[1] = data\_in[1] ^ data\_in[3] ^ data\_in[4];

parity[2] = data\_in[2] ^ data\_in[5] ^ data\_in[6];

parity[3] = data\_in[3] ^ data\_in[6] ^ data\_in[7];

end

assign code\_out = {data\_in, parity};

endmodule

module reed\_solomon\_decoder (

input [11:0] code\_in, // 12-bit received code

output [7:0] data\_out, // Original data

output reg error // Error flag

);

reg [3:0] syndrome; // Error-checking syndrome

always @(\*) begin

// Simple syndrome computation (example polynomial)

syndrome[0] = code\_in[8] ^ code\_in[0] ^ code\_in[1] ^ code\_in[2];

syndrome[1] = code\_in[9] ^ code\_in[1] ^ code\_in[3] ^ code\_in[4];

syndrome[2] = code\_in[10] ^ code\_in[2] ^ code\_in[5] ^ code\_in[6];

syndrome[3] = code\_in[11] ^ code\_in[3] ^ code\_in[6] ^ code\_in[7];

error = (syndrome != 4'b0000); // Flag error if syndrome is non-zero

end

assign data\_out = code\_in[11:4]; // Extract original data

endmodule

USES

 Corrects **burst errors**, often seen in clustered data corruption.

 Integral to optical storage like **CDs, DVDs, and Blu-rays**.

 Deployed in **high-noise environments** (e.g., satellite communication).

 Key in **network protocols** (Ethernet, DSL) and broadcasting systems (DVB/DAB).

 Ensures reliability in **RAID arrays** and advanced storage solutions.

 Supports **QR code generation** and decoding.

**//SRAM CONTROLLER**

module sram\_controller (

input wire clk, reset,

input wire req, rw, // Read (1) / Write (0)

input wire [15:0] addr, // 16-bit Address

inout wire [7:0] data, // Bidirectional data bus

output reg mem\_ce, mem\_oe, mem\_we // Control signals

);

reg [7:0] data\_reg; reg data\_out\_enable;

assign data = (data\_out\_enable) ? data\_reg : 8'bz;

typedef enum logic [1:0]

{ IDLE = 2'b00,READ = 2'b01,WRITE = 2'b10,

DONE = 2'b11} state\_t;

state\_t state;

always @(posedge clk or posedge reset) begin

if (reset)

begin

state <= IDLE;

mem\_ce <= 1;mem\_oe <= 1;mem\_we <= 1;

data\_out\_enable <= 0;

end

else

begin

case (state)

IDLE: if (req) state <= (rw) ? READ : WRITE;

READ: begin

mem\_ce <= 0; mem\_oe <= 0; mem\_we <= 1;

state <= DONE;

end

WRITE: begin

mem\_ce <= 0; mem\_oe <= 1; mem\_we <= 0;

data\_reg <= data;

data\_out\_enable <= 1;

state <= DONE;

end

DONE: begin

mem\_ce <= 1; mem\_oe <= 1; mem\_we <= 1;

data\_out\_enable <= 0;

state <= IDLE;

end

endcase

end end

endmodule

**BASIC MEMORY CONTROLLER VERILOG CODE**

**//DRAM CONTROLLER**

module dram\_controller (

input wire clk, reset,

input wire req, rw, // Read (1) / Write (0)

input wire [15:0] addr, // 16-bit Address

inout wire [7:0] data, // Bidirectional data bus

output reg mem\_ce, mem\_oe, mem\_we, mem\_ras, mem\_cas, refresh ); reg [7:0] data\_reg;

reg data\_out\_enable;

assign data = (data\_out\_enable) ? data\_reg : 8'bz;

typedef enum logic [2:0] { IDLE = 3'b000, REFRESH = 3'b001, ACTIVATE = 3'b010, READ = 3'b011, WRITE = 3'b100,PRECHARGE = 3'b101,DONE = 3'b110 } state\_t;

state\_t state;

always @(posedge clk or posedge reset) begin

if (reset) begin

state <= IDLE; refresh <= 0; mem\_ce <= 1; mem\_ras <= 1; mem\_cas <= 1;

mem\_oe <= 1; mem\_we <= 1; data\_out\_enable <= 0;

end else begin

case (state)

IDLE: if (req) state<= (rw) ? READ : WRITE;

REFRESH: begin refresh <= 1; state <= IDLE;

end

ACTIVATE: begin mem\_ce <= 0; mem\_ras <= 0;

state <= (rw) ? READ : WRITE;

end

READ: begin mem\_cas <= 0; mem\_oe <= 0; mem\_we <= 1;

state <= DONE;

end

WRITE: begin mem\_cas <= 0; mem\_we <= 0;

data\_reg <= data; data\_out\_enable <= 1;

state <= DONE;

end

PRECHARGE: begin mem\_ce <=1; mem\_ras <= 1;

state <= IDLE;

end

DONE: begin mem\_ce <= 1; mem\_ras <= 1;

mem\_cas <= 1; mem\_oe <= 1; mem\_we <= 1;

data\_out\_enable <= 0;

state <= IDLE;

end

endcase

end

end

endmodule