18th july 2025

VLSI System Project #1254

Design and Verification of a Parameterized Memory Controller using Verilog

Done this project under the vision of Deep Logic Labs.
Vinayak V P

MEMORY CONTROLLER.

ABOUT THE PROJECT(PROBLEM STATEMENT)

We have a single port RAM (random Access Memory), having R number of rows and C number of columns having each word of N bits. Here R, C and N are parameters that you can set with any number of your choice. This operates at 100Mhz frequency and has a reset signal. A read request is issued to it by asserting req signal asserted along with rw signal high for at least one cycle, read data is returned on output bus Qa along with valid signal 1 for at least one clock cycle from a given row address ar and column address ac which are driven as inputs on address buses. In case a write request is issued by asserting req signal asserted along with rw signal low, write data on input bus Qi is written on a given row address ar and column address ac which are driven as inputs on address buses. A controller block having 100MHz as clock, reset and chip enable signals, -Take address to be read/written “addr” as input -Generates row and column addresses for the above RAM model -Generates data to be written and ready signals -Take any other assumptions as required.

MEMORY CONTROLLER.

KEY UNDERSTANDING & WHAT WE NEED TO DO:

1. Parameterized Single-Port RAM Design:
    - Create a RAM module with parameters R (rows), C (columns), and N (bit-width per word).
    - Data is accessed via 2D addressing: row (ar) and column (ac) inputs.
    - Support read/write operations controlled via req and rw signals.
2. Read/Write Operation Logic:
    - Read: When req = 1 and rw = 1, output the data from given ar and ac with valid = 1.
    - Write: When req = 1 and rw = 0, write input Qi to the given location.
    - Both operations are synchronous with a 100 MHz clock and should respond after at least 1 clock cycle.
3. Controller Module Requirements:
    - Generates:
        - ar, ac (row/col addresses)
        - Qi (write data)
        - ready signal (indicating it's ready for new request)
    - Takes addr as a single input and decodes it into row and column (ar, ac).
    - Controlled by clk, rst, and chip select cs.
4. Assumptions & Constraints:
    - You can make reasonable assumptions about:
        - Size and format of addr (e.g., split high bits for row, low bits for column).
        - Timing (at least 1 clock cycle for valid or ready signal).
    - Design should be synchronous and operate at 100 MHz.
5. Integration & Testability:
    - Combine the controller and RAM modules in a top-level module.
    - Write a testbench to simulate multiple read and write cycles to test correctness of address generation, data flow, and control signals.

MEMORY CONTROLLER.

WHAT IS SINGLE PORT RAM

Single-port RAM is a type of memory that allows either a read or a write operation at a time using a single interface for address, data, and control signals. It is synchronous, meaning all operations are triggered on the rising or falling edge of a clock signal. The user provides an address and either reads data from that location (rw = 1) or writes data to it (rw = 0) based on control signals like req (request), rw (read/write), and clk (clock). Data read from memory is available on the Qa output with a valid signal, while data to be written is applied to the Qi input. The entire access is coordinated by a controller using proper timing.
This RAM is organized as a 2D array with R rows and C columns, where each memory cell stores a word of N bits. The use of single-port RAM is common in systems where memory access conflicts are minimal and cost or area needs to be optimized. Since only one access is allowed per cycle, it’s simpler than dual-port RAM and consumes less silicon. However, it cannot perform read and write simultaneously, which can be a limitation in high-throughput systems.

USE CASE OF SINGLE-PORT RAM

Single-port RAM is commonly used in embedded systems, microcontrollers, FPGAs, and digital signal processing (DSP) blocks where memory access is controlled and sequential.
A typical example is a buffer or register file inside a processor, where only one instruction accesses memory per cycle—either to load (read) or store (write) data. It’s ideal for applications where simplicity, low power, and minimal area are more important than parallel memory access.

MEMORY CONTROLLER.

DATAPATH LEVEL

```
`timescale 1ns / 1ps

module datapath #(
  parameter R = 4,
  parameter C = 4,
  parameter N = 4
)(
  input wire clk,
  input wire rst,
  input wire req,
  input wire rw,
  input wire cs,
  input wire [N-1:0] Qi,
  input wire [$clog2(R)-1:0] ar,
  input wire [$clog2(C)-1:0] ac,
  output reg [N-1:0] Qa,
  output reg valid
);

  reg [N-1:0] mem [0:R-1][0:C-1];

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      Qa  <=  {N{1'b0}};
      valid <= 1'b0;
    end else if (cs && ~rst) begin
      if (req && !rw) begin
        mem[ar][ac] <= Qi;
        valid <= 1'b0;
      end else if (req && rw) begin
        Qa <= mem[ar][ac];
        valid <= 1'b1;
      end else begin
        valid <= 1'b0;
      end
    end else begin
      valid <= 1'b0;
    end
  end
endmodule
```

MEMORY CONTROLLER.

EXPLANATION OF DATAPATH LEVEL

Purpose:

The datapath implements the actual RAM array and handles the read/write operations. It is the main data-processing unit.

Memory Structure:

2D array of registers: reg [N-1:0] mem[0:R-1][0:C-1]

For R=4, C=4, and N=4, this creates a 16-location, 4-bit wide memory.

Functions:

On req=1 and rw=0 → data Qi is written to mem[ar][ac].

On req=1 and rw=1 → data is read from mem[ar][ac] and output on Qa.

valid is asserted only during a successful read.

Supports rst to clear outputs and initialize system state.

Signal Flow:

Inputs: clk, rst, cs, req, rw, Qi, ar, ac

Outputs: Qa, valid

MEMORY CONTROLLER.

CONTROL PATH LEVEL

```
`timescale 1ns / 1ps

module controller #(
  parameter R = 4,
  parameter C = 4,
  parameter N = 4
)(
  input wire clk,
  input wire rst,
  input wire cs,
  input wire req,
  input wire rw,
  input wire [$clog2(R*C)-1:0] addr,
  input wire valid,
  output reg ready,
  output wire [$clog2(R)-1:0] ar,
  output wire [$clog2(C)-1:0] ac
);

  assign ar = addr[$clog2(R*C)-1 -: $clog2(R)];
  assign ac = addr[$clog2(C)-1:0];

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      ready <= 1'b0;
    end else begin
      if (cs) begin
        if (req && !rw)
          ready <= 1'b1;
        else if (valid)
          ready <= 1'b1;
        else
          ready <= 1'b0;
      end else begin
        ready <= 1'b0;
      end
    end
  end

endmodule
```

MEMORY CONTROLLER.

EXPLANATION OF CONTROL PATH LEVEL

Purpose:

The controller interprets the flat memory address and generates the appropriate row (ar) and column (ac) signals to access memory cells. It also controls the ready signal for system coordination.

Functions:

Converts the 4-bit address (addr) into 2-bit row and column values using bit slicing and $clog2.

Monitors req, rw, and valid to determine whether the system is ready to process a request.

Implements simple combinational logic to manage the read/write handshake.

Signal Flow:

Inputs: clk, rst, cs, req, rw, addr, valid

Outputs: ar, ac, ready

MEMORY CONTROLLER.

TOP MODULE

```
`timescale 1ns / 1ps

module top_module #(
  parameter R = 4,
  parameter C = 4,
  parameter N = 4
)(
  input wire clk,
  input wire rst,
  input wire cs,
  input wire req,
  input wire rw,
  input wire [$clog2(R*C)-1:0] addr,
  input wire [N-1:0] Qi,                 // Input data
  output wire [N-1:0] Qa,                // Output data
  output wire valid,
  output wire ready
);

  wire [$clog2(R)-1:0] ar;
  wire [$clog2(C)-1:0] ac;
  wire valid_internal;

  controller #(
    .R(R),
    .C(C),
    .N(N)
  ) c1 (
    .clk(clk),
    .rst(rst),
    .cs(cs),
    .req(req),
    .rw(rw),
    .addr(addr),
    .valid(valid_internal),
    .ready(ready),
    .ar(ar),
    .ac(ac)
  );
  datapath #(
  .R(R),
  .C(C),
  .N(N)
  ) d1 (
  .clk(clk),
  .rst(rst),
  .req(req),
  .rw(rw),
  .cs(cs),
  .Qi(Qi),
  .ar(ar),
  .ac(ac),
  .Qa(Qa),
  .valid(valid_internal)
  );

  assign valid = valid_internal;

endmodule
```

MEMORY CONTROLLER.

EXPLANATION OF TOP MODULE

Top Module (top_module)

Purpose:

This module integrates the controller and datapath blocks. It acts as the system"s backbone, ensuring that control and data signals are properly routed.

Functions:

Takes external inputs: clk, rst, cs, req, rw, addr, Qi.

Connects the controller output (row/column address: ar, ac) to the datapath.

Links valid and ready signals to external interfaces for status monitoring.

Modularizes the design to support scalability and parameterization (R, C, N).

Parameterization:

R = 4 rows

C = 4 columns

N = 4 bits data width

MEMORY CONTROLLER.

TEST BENCH

```
`timescale 1ns / 1ps

module tb_sp;

  parameter R = 4;
  parameter C = 4;
  parameter N = 4;

  reg clk = 0;
  reg rst = 1;

  reg cs = 0, req = 0, rw = 0;
  reg [$clog2(R*C)-1:0] addr = 0;
  reg [N-1:0] Qi = 0;
  wire [N-1:0] Qa;
  wire valid, ready;

  top_module #(.R(R), .C(C), .N(N)) dut (
    .clk(clk), .rst(rst), .cs(cs),
    .req(req), .rw(rw), .addr(addr),
    .Qi(Qi), .Qa(Qa), .valid(valid), .ready(ready)
  );

  always #5 clk = ~clk;

  initial #25 rst = 0;

  initial begin
    @(negedge rst);

    cs = 1; addr = 4'd3; Qi = 4'h01; rw = 0; req = 1;
    @(posedge clk);

    req = 1; @(posedge clk);

    cs = 1; addr = 4'd3; rw = 1; req = 1;
    @(posedge clk);

    cs = 1; addr = 4'd5; Qi = 4'd06; rw = 0; req = 1;
    @(posedge clk);
```

```
`
      @(posedge  clk);

      req  =  1;  @(posedge  clk);

      cs  =  1;  addr  =  4'd5;  rw  =  1;  req  =  1;
      @(posedge  clk);

      $finish;
    end

endmodule
```

EXPLANATION OF TEST BENCH

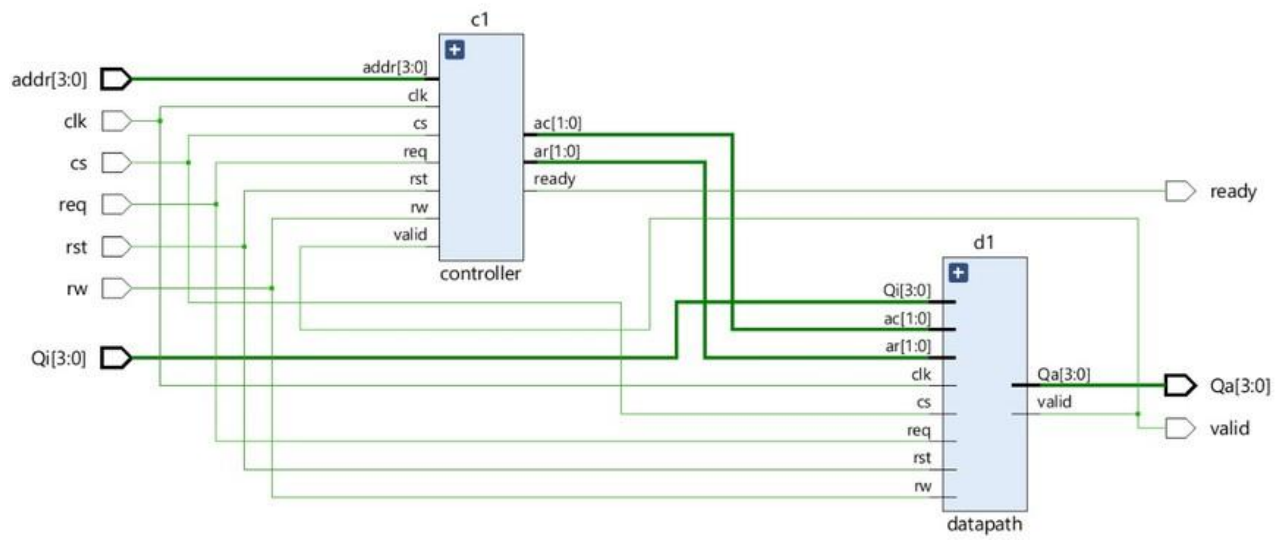
This testbench instantiates top_module (your DUT) with a 4×4 memory of 4-bit words. The clk toggles every 5 ns, and rst is held high for the first 25 ns before being released. After reset deassertion, the test sequence begins:

1. Write to address 3: cs=1, addr=3, Qi=0x1, rw=0 (write mode), req=1, then clock.
2. Optional hold: req=1 is held through the next cycle (though redundant).
3. Read from address 3: rw=1 (read mode), req=1, then clock—should return Qa=0x1, and valid should go high.
4. Write to address 5: addr=5, Qi=0x6, rw=0, req=1, then clock.
5. Hold, then
6. Read from address 5: rw=1 with req=1, then clock—should return Qa=0x6.

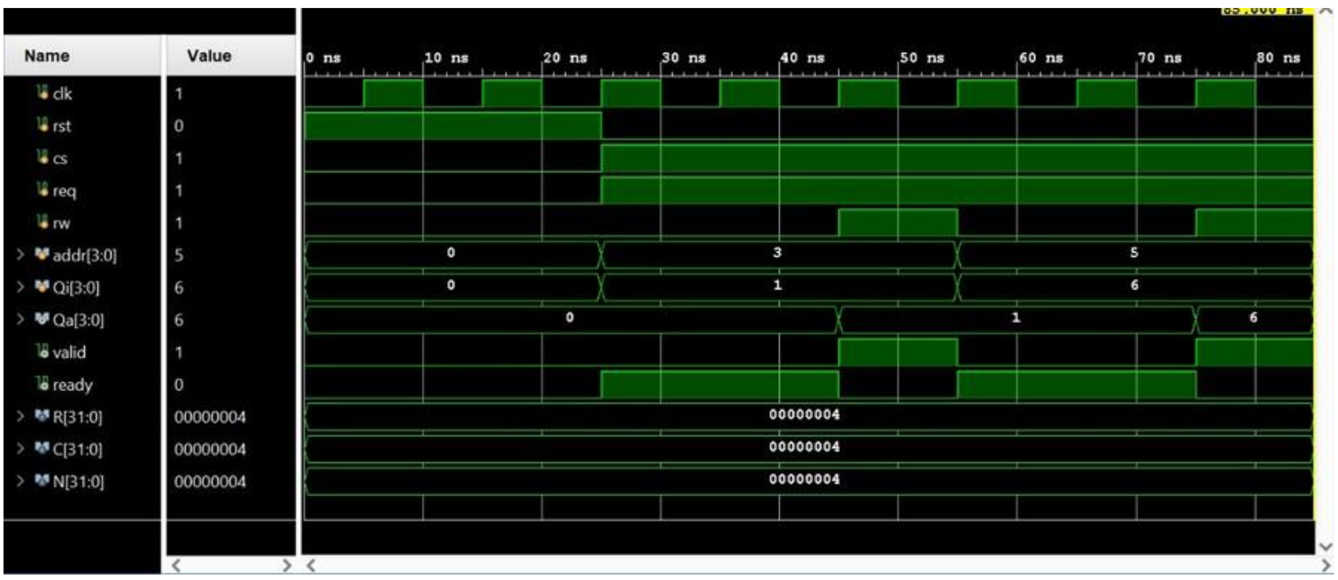
The test then calls $finish, ending simulation. This sequence verifies correct write and read operations at two different addresses, checking data integrity and valid/ready handshake behavior.

MEMORY CONTROLLER.

SCHEMATIC DIAGRAM



SIMULATION OF TEST BENCH



MEMORY CONTROLLER.

CONSTRAINTS FILE (NEXYS 4 DDR)

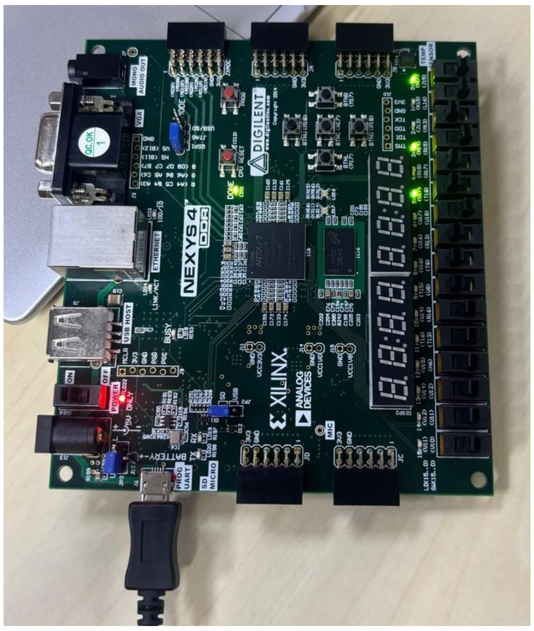
```
## Clock
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clk }];

## Switches for control signals
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { rst }];
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { req }];
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { rw }];
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { cs }];

## Data input (Qi[3:0])
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { Qi[0] }];
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { Qi[1] }];
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { Qi[2] }];
set_property -dict { PACKAGE_PIN T8  IOSTANDARD LVCMOS18 } [get_ports { Qi[3] }];

## Address input (addr[3:0])
set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { addr[0] }];
set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { addr[1] }];
set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { addr[2] }];
set_property -dict { PACKAGE_PIN H6  IOSTANDARD LVCMOS33 } [get_ports { addr[3] }];

## Flow-control signal
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { ready }];

## Data output (Qa[3:0]) and valid
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { Qa[0] }];
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { Qa[1] }];
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { Qa[2] }];
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { Qa[3] }];
set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports { valid }];
```

MEMORY CONTROLLER.

FPGA IMPLEMENTATION (NEXYS 4 DDR)

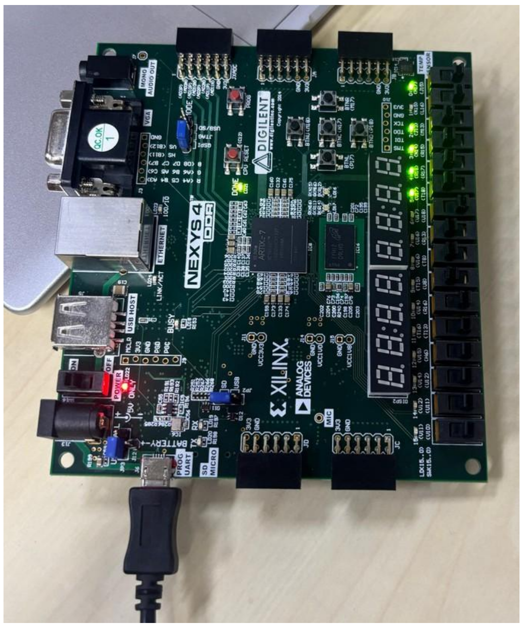
1101

WRITE



- rst = 0
- req =1
- rw =0
- cs=1
- Qi[0]=1
- Qi[1]=0
- Qi[2]=1
- Qi[3]=1
- addr[0]=1
- addr[1]=0
- addr[2]=0
- addr[3]=0

READ

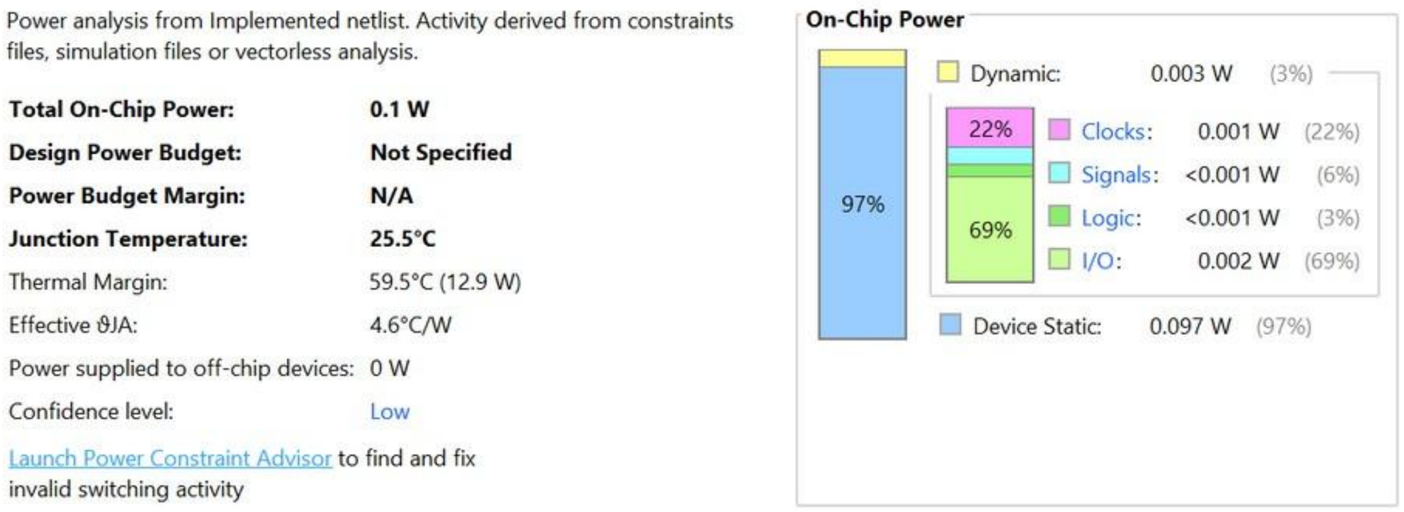


- rst = 0
- req =1
- rw =1
- cs=1
- Qa[0]=1
- Qa[1]=0
- Qa[2]=1
- Qa[3]=1
- addr[0]=1
- addr[1]=0
- addr[2]=0
- addr[3]=0

MEMORY CONTROLLER.

POWER CONSUMPTION REPORT

Parameter	Value
Total On-Chip Power	**0.101 W**
Dynamic Power	0.003 W (3%)
Static Power	0.097 W (97%)
Junction Temperature	25.5°C
Thermal Margin	59.5°C (12.9W)



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

Total On-Chip Power:	**0.1 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (12.9 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

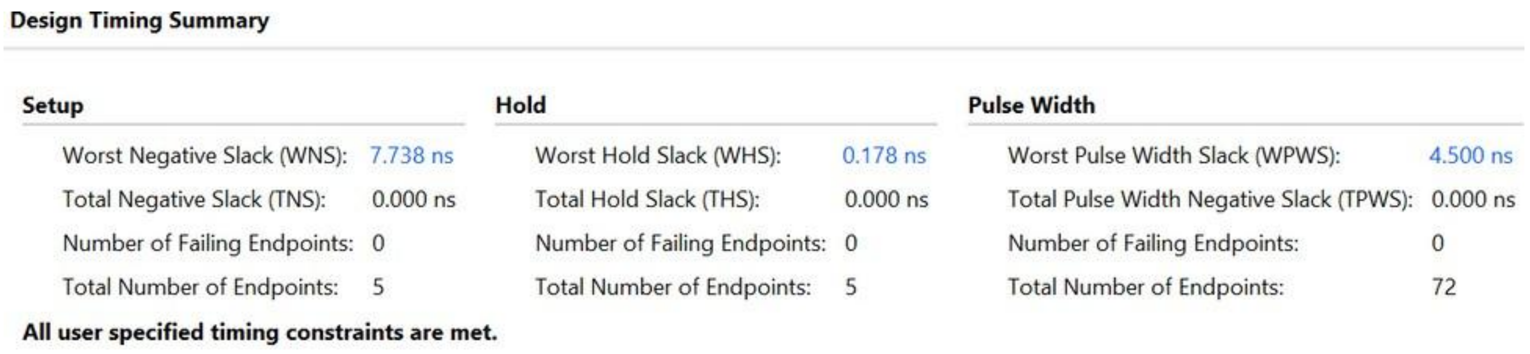
On-Chip Power

Dynamic:		0.003 W	(3%)
	Clocks:	0.001 W	(22%)
	Signals:	<0.001 W	(6%)
	Logic:	<0.001 W	(3%)
	I/O:	0.002 W	(69%)
Device Static:		0.097 W	(97%)

MEMORY CONTROLLER.

TIMING REPORT SUMMARY

Timing Parameter	Value
Worst Negative Slack (WNS)	7.738 ns
Total Negative Slack (TNS)	0.000 ns
Worst Hold Slack (WHS)	0.178 ns
Worst Pulse Width Slack (WPWS)	4.500 ns
Total Pulse Width Violations	0
Total Endpoints (Setup/Hold)	5
Total Endpoints (Pulse Width)	72



Design Timing Summary

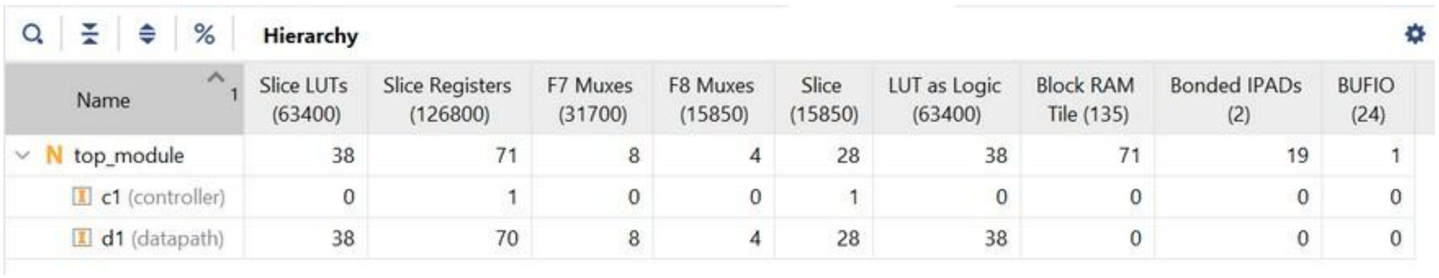
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.738 ns	Worst Hold Slack (WHS):	0.178 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5	Total Number of Endpoints:	5	Total Number of Endpoints:	72

All user specified timing constraints are met.

MEMORY CONTROLLER.

HARDWARE UTILIZATION REPORT

Resource Type	Used	Available	Utilization (%)
Slice LUTs	38	63,400	0.06%
Slice Registers	70	126,800	0.06%
F7 Muxes	8	31,700	0.03%
F8 Muxes	4	15,850	0.03%
Bonded IOBs	19	210	9.05%
BUFGCTRL	1	32	3.13%



Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	Block RAM Tile (135)	Bonded IPADs (2)	BUFIO (24)
top_module	38	71	8	4	28	38	71	19	1
c1 (controller)	0	1	0	0	1	0	0	0	0
d1 (datapath)	38	70	8	4	28	38	0	0	0

MEMORY CONTROLLER.

WHAT WE LEARNED

- How to slice bits of a bus to derive row/column addresses—a practical example of parameterized addressing.
- How to design a minimal controller for data synchronization with the memory, including reset handling.
- The mechanics of the ready/valid handshake, which ensures data is only transferred when both parties agree—and why it matters in hardware interfaces
- How to set up a self-driven testbench, using event controls like @(posedge clk) and @(negedge rst), to validate your design.
- How to refine your FPGA constraints (.xdc) to only include the signals you need—helping ensure clean synthesis and placement

CONCLUSION

In this project, we built a small but fully functional memory interface on the Nexys4_DDR FPGA board. It comprises:

- A datapath: a 4 × 4 memory array (parameterized), accessed via sliced addr signals and controlled through a ready/valid handshake.
- A controller: implemented in Verilog, that decodes the 4-bit addr into row and column indices, then drives the ready signal based on cs, req, rw, and valid.
- A testbench: automating writing and reading values at different addresses to verify correct behavior.
- A cleaned-up XDC constraints file: assigning only the relevant pins (clock, switches, LEDs, data lines), streamlining FPGA implementation.