

# **EE529 Embedded Systems**

Group 1

## **Lab Assignment 3**

**Student Name :** Ayan Garg and Om Maheshwari

**Roll Number :** B23484 and B23089

## Contents

<b>1</b>	<b>Question 1</b>	<b>3</b>
1.1	Theory . . . . .	3
1.1.1	6116 SRAM Architecture . . . . .	3
1.1.2	Memory Organization . . . . .	3
1.1.3	Signal Interface . . . . .	3
1.1.4	Block Diagram . . . . .	4
1.1.5	6116 SRAM Operation . . . . .	4
1.1.6	Timing Diagrams . . . . .	5
1.2	VHDL Implementation of 6116 SRAM . . . . .	6
1.3	Testbench for 6116 SRAM . . . . .	7
1.4	Simulation Results . . . . .	12
1.4.1	Waveform Analysis . . . . .	12
1.4.2	Simulation Timeline . . . . .	12
<b>2</b>	<b>Question 2</b>	<b>12</b>
2.1	Theory . . . . .	12
2.1.1	EDO DRAM Architecture . . . . .	13
2.1.2	Address Multiplexing . . . . .	13
2.1.3	Signal Interface . . . . .	13
2.1.4	EDO DRAM Operation . . . . .	13
2.1.5	Timing Diagram Comparison . . . . .	14
2.1.6	EDO Timing Diagram Details . . . . .	15
2.1.7	Performance Analysis . . . . .	16
2.2	VHDL Implementation OF EDODRAM . . . . .	16
2.3	Testbench for EDODRAM . . . . .	17
2.4	Results . . . . .	20
2.4.1	Simulation Waveform Analysis . . . . .	20
<b>3</b>	<b>Question 3</b>	<b>21</b>
3.1	Theory . . . . .	21
3.1.1	SDRAM Architecture . . . . .	21
3.1.2	Address Organization . . . . .	21
3.1.3	Signal Interface . . . . .	22
3.1.4	Internal SDRAM Control Signals . . . . .	22
3.1.5	SDRAM Architecture Diagram . . . . .	23
3.1.6	SDRAM Operation . . . . .	23
3.1.7	Timing Characteristics . . . . .	24
3.1.8	SDRAM Timing Diagram . . . . .	25
3.2	VHDL Implementation OF SDRAM . . . . .	25
3.3	Testbench for SDRAM . . . . .	27
3.4	Results . . . . .	29
3.4.1	Simulation Waveform Analysis . . . . .	29
3.4.2	Key Observations from Waveform . . . . .	30

<b>4</b>	<b>Question 4</b>	<b>31</b>
4.1	Theory . . . . .	31
4.1.1	Architecture Overview . . . . .	31
4.1.2	Address and Width Organisation . . . . .	31
4.1.3	Signal Interface . . . . .	31
4.1.4	FSM State Descriptions . . . . .	32
4.1.5	Operation Sequence . . . . .	32
4.2	VHDL Implementation of the Arithmetic Mean Unit . . . . .	32
4.3	Testbench for the Arithmetic Mean Unit . . . . .	33
4.4	Results . . . . .	36
4.4.1	Simulation Waveform Analysis . . . . .	36

# 1 Question 1

## 1.1 Theory

The 6116 is a  $2K \times 8$ -bit static CMOS RAM that provides high-speed, low-power data storage without the need for refresh cycles. Unlike dynamic RAM (DRAM), static RAM (SRAM) retains data as long as power is supplied, making it ideal for cache memory, embedded systems, and applications requiring fast, reliable memory access.

### 1.1.1 6116 SRAM Architecture

The 6116 SRAM consists of the following key components:

1. **Memory Array:** A 2048-byte ( $2K \times 8$ ) storage array organized as a  $128 \times 128$  memory matrix containing 16,384 individual memory cells
2. **Row Decoder:** A 7-to-128 decoder that selects one of 128 rows based on address lines A10 through A4
3. **Column Decoder:** A 4-to-8 decoder that selects 8 columns simultaneously based on address lines A3 through A0, since data is organized in 8-bit bytes
4. **Tri-state Output Buffers:** Control the bidirectional data bus, enabling high-impedance state when not actively reading
5. **Input Data Control:** Manages write operations and data input path
6. **Sense Amplifiers & Column I/O:** Read data from memory cells and amplify signals for output

### 1.1.2 Memory Organization

### 1.1.3 Signal Interface

Signal	Type	Description
Address[10:0]	Input	11-bit address bus to select one of 2048 memory locations (A10 to A0).
I/O[7:0]	Inout	Bidirectional 8-bit data bus with tri-state capability for read/write operations.
CS <sub>n</sub>	Input	Chip Select (active low). When high, chip is deselected and outputs are in high-impedance state.
OE <sub>n</sub>	Input	Output Enable (active low). Controls tri-state output buffers. Must be low for read operations.
WE <sub>n</sub>	Input	Write Enable (active low). Low = Write mode, High = Read mode.

Table 1: 6116 SRAM Signal Interface

### 1.1.4 Block Diagram

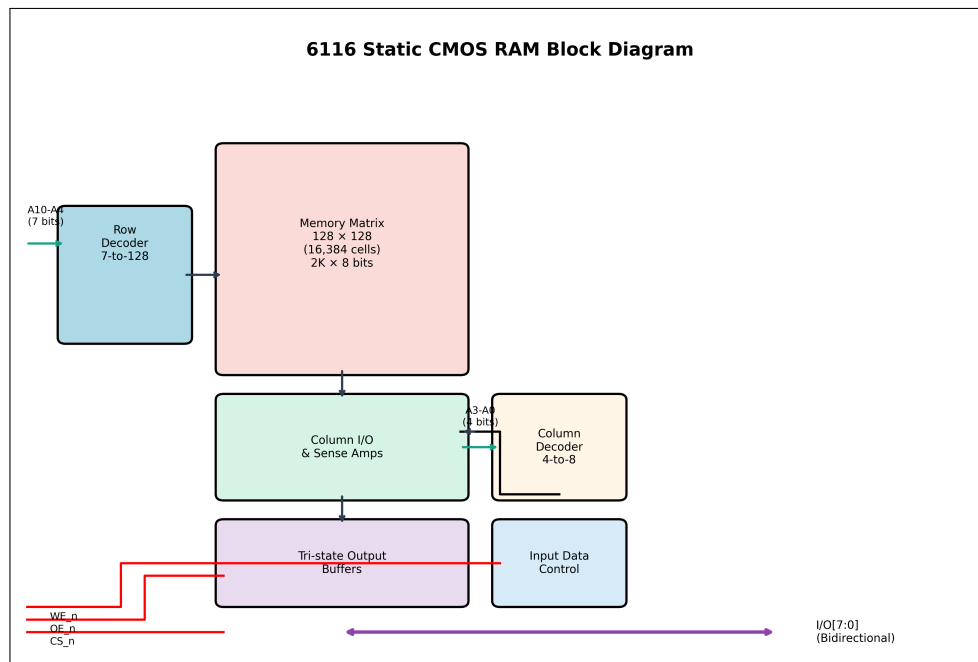


Figure 1: 6116 SRAM Internal Architecture

Figure 1 illustrates the internal architecture of the 6116 SRAM:

- **Address Decoding Path:** Address lines A10-A4 feed the row decoder (7-to-128), while A3-A0 feed the column decoder (4-to-8)
- **Data Path:** The memory matrix connects to column I/O circuitry, which interfaces with tri-state buffers for read operations and input data control for write operations
- **Control Logic:** CS\_n, OE\_n, and WE\_n signals control the operational mode and output enable states

### 1.1.5 6116 SRAM Operation

#### Read Operation Sequence:

1. Apply valid address on Address[10:0] bus
2. Assert CS\_n = 0 (select chip)
3. Assert OE\_n = 0 (enable output buffers)
4. Keep WE\_n = 1 (read mode)
5. Wait for access time ( $t_{AA} = 45\text{ns}$  for 45ns grade chip)
6. Valid data appears on IO[7:0] bus
7. Data remains valid as long as address and control signals are stable
8. Deassert OE\_n = 1 or CS\_n = 1 to end read cycle

9. IO bus returns to high-impedance state

#### Write Operation Sequence:

1. Apply valid address on Address[10:0] bus
2. Assert  $\text{CS}_n = 0$  (select chip)
3. Keep  $\text{OE}_n = 1$  (disable output during write)
4. Drive data onto IO[7:0] bus
5. Assert  $\text{WE}_n = 0$  (write pulse begins)
6. Maintain  $\text{WE}_n$  low for minimum write pulse width ( $t_{WP} = 25\text{ns}$ )
7. Data is captured into memory cell at rising edge of  $\text{WE}_n$
8. Deassert  $\text{WE}_n = 1$  (write pulse ends, data is latched)
9. Maintain data valid for hold time ( $t_{DH}$ )
10. Release IO bus to high-impedance
11. Deassert  $\text{CS}_n = 1$  to complete cycle

#### 1.1.6 Timing Diagrams

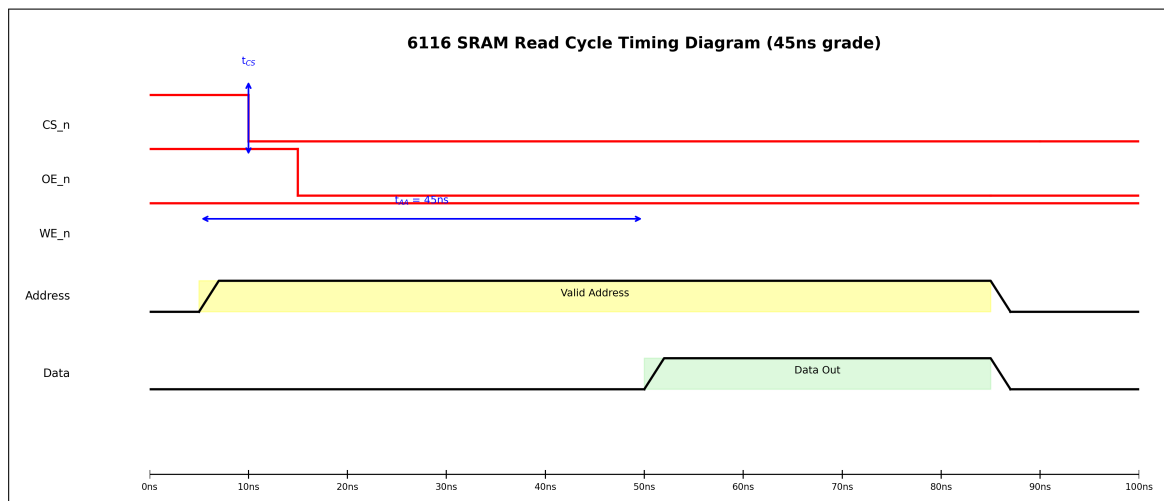


Figure 2: 6116 SRAM Read Cycle Timing Diagram

Figure 2 illustrates the read cycle timing:

- **Address Setup:** Address must be stable before and during the read operation
- **Control Signals:**  $\text{CS}_n$  and  $\text{OE}_n$  must both be asserted (low) for output enable
- **Access Time ( $t_{AA}$ ):** Critical parameter - data becomes valid 45ns after address is stable

- **Data Valid Window:** Data remains valid as long as control signals and address are maintained
- **Output Disable:** When  $OE_n$  or  $CS_n$  goes high, outputs enter high-Z state within  $t_{OHZ}$

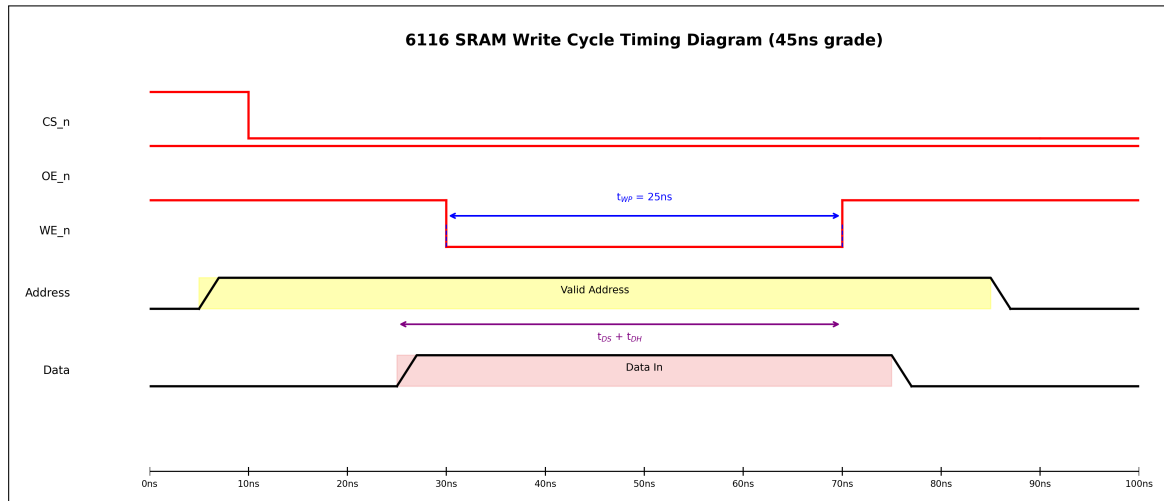


Figure 3: 6116 SRAM Write Cycle Timing Diagram

Figure 3 illustrates the write cycle timing:

- **Write Pulse ( $t_{WP}$ ):**  $WE_n$  must remain low for minimum 25ns
- **Data Capture:** Data is latched into the memory cell on the rising edge of  $WE_n$
- **Setup and Hold:** Data must be stable for  $t_{DS}$  before and  $t_{DH}$  after  $WE_n$  rising edge
- **Write Recovery ( $t_{WR}$ ):** Minimum 5ns from  $WE_n$  high to end of cycle
- **$OE_n$  High:** Output buffers must be disabled ( $OE_n = 1$ ) during write to avoid bus contention

## 1.2 VHDL Implementation of 6116 SRAM

The following VHDL implementation models the 6116 SRAM with accurate timing behavior:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity SRAM_6116 is
6     Port (
7         Address : in  STD_LOGIC_VECTOR (10 downto 0);
8         IO      : inout STD_LOGIC_VECTOR (7 downto 0);
9         CS_n    : in  STD_LOGIC;
10        OE_n    : in  STD_LOGIC;

```

```

11         WE_n      : in  STD_LOGIC
12     );
13 end SRAM_6116;
14
15 architecture Behavioral of SRAM_6116 is
16     -- Memory array: 2048 locations x 8 bits
17     type memory_array is array (0 to 2047) of STD_LOGIC_VECTOR(7
18         downto 0);
19     signal ram_block : memory_array := (others => (others => '0'))
20         );
21
22     signal addr_int : integer range 0 to 2047;
23
24     constant T_AA   : time := 45 ns; -- Address Access Time
25     constant T_OHA  : time := 10 ns; -- Output Hold from Address
26         Change
27     constant T_OHZ  : time := 10 ns; -- Output Disable Time
28
29 begin
30     addr_int <= to_integer(unsigned(Address));
31
32     write_proc: process(WE_n, CS_n)
33     begin
34         if CS_n = '0' and rising_edge(WE_n) then
35             -- Data is captured on rising edge of WE_n
36             ram_block(addr_int) <= IO;
37         end if;
38     end process;
39
40     read_proc: process(CS_n, OE_n, WE_n, Address)
41     variable current_addr : integer range 0 to 2047;
42     begin
43         current_addr := to_integer(unsigned(Address));
44
45         if CS_n = '0' and OE_n = '0' and WE_n = '1' then
46             IO <= ram_block(current_addr) after T_AA;
47         else
48             IO <= (others => 'Z') after T_OHZ;
49         end if;
50     end process;
51 end Behavioral;

```

Listing 1: 6116 SRAM VHDL Implementation with Timing

### 1.3 Testbench for 6116 SRAM

The testbench validates all key operations of the SRAM:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;

```

```

3
4 entity tb_SRAM_6116 is
5 end tb_SRAM_6116;
6
7 architecture Behavioral of tb_SRAM_6116 is
8     component SRAM_6116
9         Port (
10             Address : in  STD_LOGIC_VECTOR (10 downto 0);
11             IO       : inout STD_LOGIC_VECTOR (7 downto 0);
12             CS_n     : in  STD_LOGIC;
13             OE_n     : in  STD_LOGIC;
14             WE_n     : in  STD_LOGIC
15         );
16     end component;
17
18     signal Address : STD_LOGIC_VECTOR(10 downto 0) := (others =>
19         '0');
20     signal IO      : STD_LOGIC_VECTOR(7 downto 0) := (others =>
21         'Z');
22     signal CS_n    : STD_LOGIC := '1';
23     signal OE_n    : STD_LOGIC := '1';
24     signal WE_n    : STD_LOGIC := '1';
25
26     constant T_RC  : time := 45 ns; -- Read Cycle Time
27     constant T_AA  : time := 45 ns; -- Access Time
28     constant T_WP  : time := 25 ns; -- Write Pulse Width
29     constant T_WC  : time := 45 ns; -- Write Cycle Time
30     constant T_AS  : time := 0 ns;  -- Address Setup Time
31     constant T_WR  : time := 5 ns;  -- Write Recovery Time
32
33 begin
34     uut: SRAM_6116 port map (
35         Address => Address,
36         IO      => IO,
37         CS_n    => CS_n,
38         OE_n    => OE_n,
39         WE_n    => WE_n
40     );
41
42     stim_proc: process
43     begin
44         -- Initial stabilization
45         wait for 100 ns;
46
47         report "===== ";
48         report "TEST 1: Basic Write Operation";
49         report "===== ";
50
51         Address <= "00000010000";
52         CS_n    <= '0';
53         OE_n    <= '1';

```

```

52     WE_n    <= '1';
53
54     wait for T_AS;
55
56     WE_n    <= '0';
57     IO      <= "00111100";
58
59     wait for T_WP;
60
61     WE_n    <= '1';
62
63     wait for T_WR;
64
65     IO      <= (others => 'Z');
66     CS_n    <= '1';
67
68     wait for 50 ns;
69
70     report "Write complete: Address 0x010 = 0x3C";
71
72     report "===== ";
73     report "TEST 2: Basic Read Operation";
74     report "===== ";
75
76     -- Read Cycle: Read from address 0x010
77     Address <= "00000010000";
78     CS_n    <= '0';
79     OE_n    <= '0';
80     WE_n    <= '1';
81
82     wait for T_AA;
83
84     report "Read complete: Data = 0x3C (should match written
85           data)";
86
87     wait for T_RC - T_AA;
88
89     OE_n    <= '1';
90     CS_n    <= '1';
91
92     wait for 100 ns;
93
94     report "===== ";
95     report "TEST 3: Multiple Address Testing";
96     report "===== ";
97
98     -- Write to address 0x020 (32 decimal)
99     Address <= "00000100000";
100    CS_n    <= '0';
101    OE_n    <= '1';
102    WE_n    <= '0';

```

```

102      IO      <= "10101010";
103
104      wait for T_WP;
105
106      WE_n     <= '1';
107      wait for T_WR;
108      IO      <= (others => 'Z');
109      CS_n     <= '1';
110
111      wait for 50 ns;
112
113      report "Write complete: Address 0x020 = 0xAA";
114
115      -- Read from address 0x020
116      Address <= "00000100000";
117      CS_n     <= '0';
118      OE_n     <= '0';
119      WE_n     <= '1';
120
121      wait for T_AA;
122
123      report "Read complete: Address 0x020 should contain 0xAA
124             ";
125
126      wait for T_RC - T_AA;
127
128      OE_n     <= '1';
129      CS_n     <= '1';
130
131      wait for 100 ns;
132
133      report "===== ";
134      report "TEST 4: Boundary Testing";
135      report "===== ";
136
137      -- Write to first address (0x000)
138      Address <= "00000000000";
139      CS_n     <= '0';
140      OE_n     <= '1';
141      WE_n     <= '0';
142      IO      <= "11111111";      -- 0xFF
143
144      wait for T_WP;
145      WE_n     <= '1';
146      wait for T_WR;
147      IO      <= (others => 'Z');
148      CS_n     <= '1';
149      wait for 50 ns;
150
151      -- Read back
152      Address <= "00000000000";

```

```

152     CS_n    <= '0';
153     OE_n    <= '0';
154     WE_n    <= '1';
155     wait for T_AA;
156
157     report "Boundary test: Address 0x000 = 0xFF";
158
159     OE_n    <= '1';
160     CS_n    <= '1';
161     wait for 100 ns;
162
163     -- Write to last address (0x7FF = 2047)
164     Address <= "111111111111";
165     CS_n    <= '0';
166     OE_n    <= '1';
167     WE_n    <= '0';
168     IO      <= "01010101";      -- 0x55
169
170     wait for T_WP;
171     WE_n    <= '1';
172     wait for T_WR;
173     IO      <= (others => 'Z');
174     CS_n    <= '1';
175     wait for 50 ns;
176
177     -- Read back
178     Address <= "111111111111";
179     CS_n    <= '0';
180     OE_n    <= '0';
181     WE_n    <= '1';
182     wait for T_AA;
183
184     report "Boundary test: Address 0x7FF = 0x55";
185
186     OE_n    <= '1';
187     CS_n    <= '1';
188     wait for 100 ns;
189
190     report "===== ";
191     report "All 6116 SRAM tests completed successfully!";
192     report "Key validations:";
193     report "1. Write operations captured data correctly";
194     report "2. Read operations retrieved correct data";
195     report "3. Multiple addresses work independently";
196     report "4. Boundary addresses function properly";
197     report "5. Tri-state control working as expected";
198     report "===== ";
199
200     wait;
201
202 end process;

```

```

203
204 end Behavioral;

```

Listing 2: 6116 SRAM Comprehensive Testbench

## 1.4 Simulation Results

### 1.4.1 Waveform Analysis

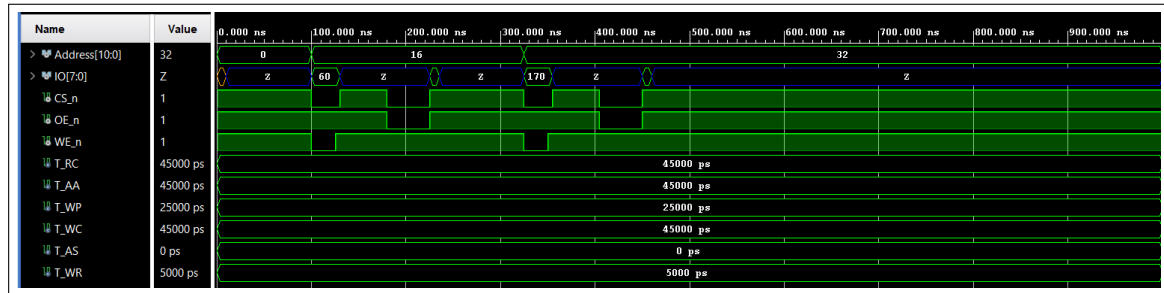


Figure 4: Complete Simulation Waveform from Vivado

Figure 4 shows the complete simulation waveform obtained from Vivado simulator. The waveform validates all critical aspects of the SRAM operation.

### 1.4.2 Simulation Timeline

Time (ns)	Operation	Address	Data
0-100	Initialization	-	-
100-200	Write Cycle 1	0x010 (16)	0x3C
200-350	Read Cycle 1	0x010 (16)	0x3C
350-450	Write Cycle 2	0x020 (32)	0xAA
450-600	Read Cycle 2	0x020 (32)	0xAA
600-700	Boundary Write	0x000 (0)	0xFF
700-850	Boundary Read	0x000 (0)	0xFF
850-950	Boundary Write	0x7FF (2047)	0x55
950+	Boundary Read	0x7FF (2047)	0x55

Table 2: Detailed Simulation Timeline

## 2 Question 2

### 2.1 Theory

Extended Data Out DRAM is an enhanced version of FP DRAM that improves memory bandwidth by extending the time data remains valid on the output bus. The key innovation is an output latch that holds data valid even after CAS returns high, enabling the memory controller to overlap the next column address setup with the current data output.

### 2.1.1 EDO DRAM Architecture

The EDO DRAM consists of the following key components:

1. **Memory Array:** A 4 KB (4096 bytes) storage array organized as 64 rows  $\times$  64 columns  $\times$  8 bits
2. **Row Address Latch:** Captures and holds the 6-bit row address on RAS falling edge
3. **Column Address Latch:** Captures the 6-bit column address on CAS falling edge
4. **EDO Output Latch:** Critical component that holds data valid after CAS rises
5. **Sense Amplifiers:** Read data from memory cells and drive output latch
6. **Tri-state Output Buffer:** Controls bidirectional data bus

### 2.1.2 Address Multiplexing

To reduce pin count, DRAM uses multiplexed addressing where row and column addresses share the same pins:

$$\text{Total Address Bits} = \log_2(4096) = 12 \text{ bits} \quad (1)$$

$$\text{Row Address Bits} = \text{Column Address Bits} = \frac{12}{2} = 6 \text{ bits} \quad (2)$$

$$\text{Full Address} = \text{Row Address}[5 : 0] \text{ concatenated with Column Address}[5 : 0] \quad (3)$$

### 2.1.3 Signal Interface

Signal	Type	Description
ras_n	Input	Row Address Strobe (active low). Latches row address and activates selected row.
cas_n	Input	Column Address Strobe (active low). Latches column address and initiates read/write.
we_n	Input	Write Enable (active low). High = Read, Low = Write.
address[5:0]	Input	Multiplexed 6-bit address bus for row/column addressing.
data[7:0]	Inout	Bidirectional 8-bit data bus with tri-state capability.

Table 3: EDO DRAM Signal Interface

### 2.1.4 EDO DRAM Operation

**Write Operation Sequence:**

1. Assert  $RAS\downarrow$  with row address on address bus
2. Row address latched, row activated
3. Assert  $CAS\downarrow$  and  $WE\downarrow$  with column address
4. Drive data onto data bus
5. Data written to memory cell at (row, column)
6. Deassert  $CAS\uparrow$  and  $WE\uparrow$
7. Deassert  $RAS\uparrow$  to complete cycle

#### Read Operation Sequence:

1. Assert  $RAS\downarrow$  with row address on address bus
2. Row address latched, row activated
3. Assert  $CAS\downarrow$  ( $WE$  high) with column address
4. Column address latched
5. Data read from memory and loaded into EDO latch
6. Data appears on data bus
7.  $CAS\uparrow$  - Data remains valid (**EDO feature**)
8. New column address can be applied immediately
9. Next  $CAS\downarrow$  - Previous data released, new data loaded

### 2.1.5 Timing Diagram Comparison

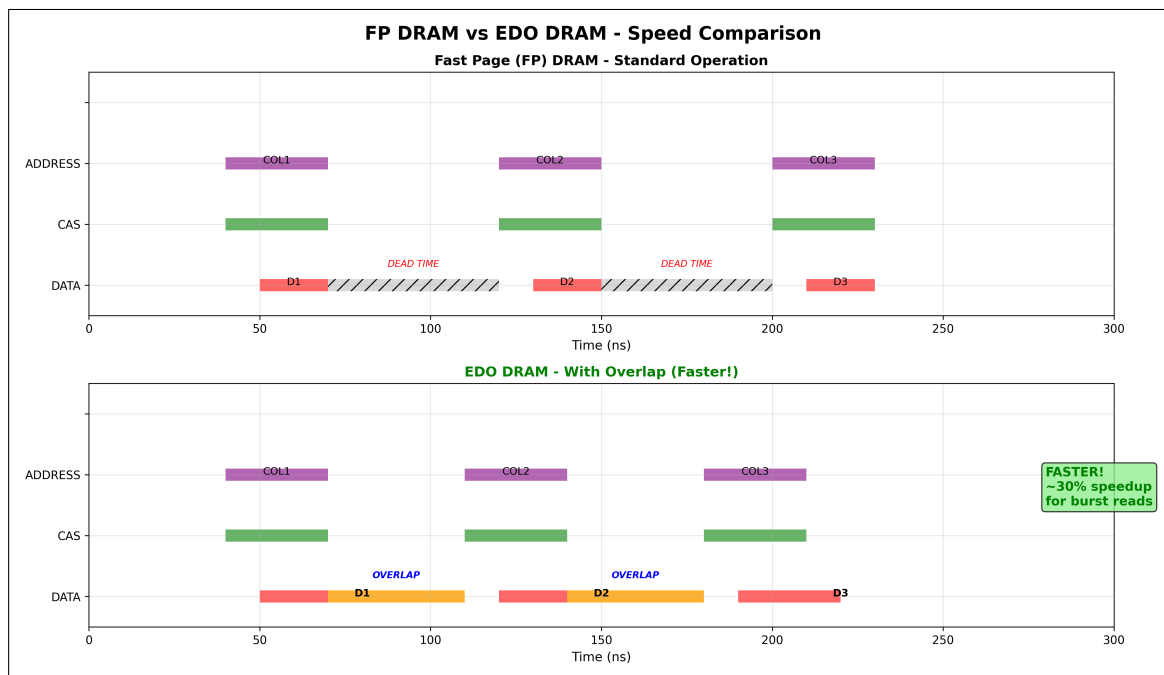


Figure 5: FP DRAM vs EDO DRAM Speed Comparison

Figure 5 illustrates the fundamental difference between FP DRAM and EDO DRAM:

- **FP DRAM (Top):** Shows dead time (gray hatched regions) between consecutive column accesses. Data goes to high-impedance when CAS rises, forcing a wait before the next CAS cycle.
- **EDO DRAM (Bottom):** Shows overlap regions (yellow) where data remains valid while the next column address is being set up. This overlap eliminates the dead time and results in faster burst access.

### 2.1.6 EDO Timing Diagram Details

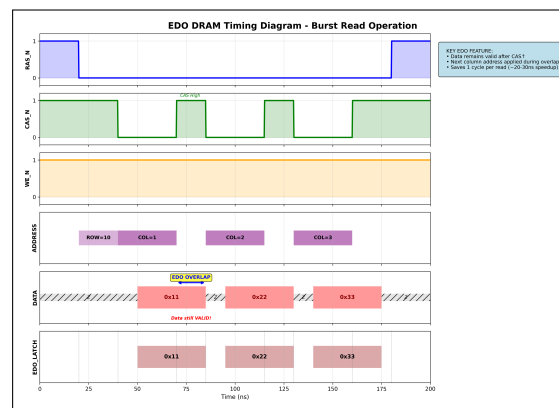


Figure 6: EDO DRAM Detailed Timing Diagram - Burst Read Operation

Figure 6 shows a detailed timing diagram for an EDO burst read operation with three consecutive column accesses. Key observations:

1. **RAS\_N:** Remains low during the entire row access cycle (20-180 ns)
2. **CAS\_N:** Shows multiple pulses for consecutive column accesses:
  - First CAS: 40-70 ns
  - Second CAS: 85-115 ns
  - Third CAS: 130-160 ns
3. **ADDRESS:** Multiplexed addressing sequence:
  - Row address (ROW=10) latched at RAS falling edge
  - Column addresses (COL=1, COL=2, COL=3) latched at respective CAS edges
4. **DATA:** Shows the EDO overlap feature:
  - Data 0x11 valid from 50-85 ns
  - Data 0x22 valid from 95-130 ns
  - Data 0x33 valid from 140-175 ns
  - **Overlap regions (70-85 ns, 115-130 ns)** where data remains valid after CAS rises

### 5. EDO\_LATCH: Internal signal showing data retention in the output latch

The yellow-highlighted "EDO OVERLAP" regions demonstrate the key advantage: data validity extends beyond the CAS high transition, allowing the next column address to be applied without waiting for a dead time period.

#### 2.1.7 Performance Analysis

For a burst of  $N$  consecutive column reads:

##### EDO DRAM Total Access Time:

$$T_{EDO} = N \times t_{CAS_{low}} + (N - 1) \times t_{overlap} + t_{final} \quad (4)$$

##### Memory Bandwidth Improvement:

For 8-bit data transfers:

$$BW_{FP} = \frac{3 \times 8 \text{ bits}}{300 \text{ ns}} = 80 \text{ Mbps} \quad (5)$$

$$BW_{EDO} = \frac{3 \times 8 \text{ bits}}{220 \text{ ns}} = 109 \text{ Mbps} \quad (6)$$

$$\Delta BW = \frac{109 - 80}{80} \times 100\% = 36.25\% \quad (7)$$

## 2.2 VHDL Implementation OF EDODRAM

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity EDO_DRAM is
6      generic (
7          ADDR_WIDTH : integer := 12;
8          DATA_WIDTH : integer := 8
9      );
10     port (
11         ras_n      : in  std_logic;
12         cas_n      : in  std_logic;
13         we_n       : in  std_logic;
14         address     : in  std_logic_vector(ADDR_WIDTH/2 - 1 downto 0);
15         data        : inout std_logic_vector(DATA_WIDTH - 1 downto 0)
16     );
17 end EDO_DRAM;
18
19 architecture Behavioral of EDO_DRAM is
20
21     type memory_array is array (0 to 2**ADDR_WIDTH - 1) of
22         std_logic_vector(DATA_WIDTH - 1 downto 0);
23     signal mem : memory_array := (others => (others => '0'));
24
25     signal row_addr      : std_logic_vector(ADDR_WIDTH/2 - 1 downto 0);
26     signal col_addr      : std_logic_vector(ADDR_WIDTH/2 - 1 downto 0);
27     signal full_addr     : std_logic_vector(ADDR_WIDTH - 1 downto 0);
28     signal edo_latch     : std_logic_vector(DATA_WIDTH - 1 downto 0);
29     signal data_valid    : std_logic := '0';
30     signal row_active    : std_logic := '0';
31     signal prev_cas_n    : std_logic := '1';
32
33 begin
34
35     full_addr <= row_addr & col_addr;
36
37     process(ras_n)
38     begin
39         if falling_edge(ras_n) then
40             row_addr <= address;
41             row_active <= '1';
42         elsif rising_edge(ras_n) then
43             row_active <= '0';
44         end if;
45     end process;
46
47     process(cas_n, ras_n)
48     begin
49         if row_active = '1' then
50             if falling_edge(cas_n) then

```

```

52         col_addr <= address;
53
54         if we_n = '1' then
55             edo_latch <= mem(to_integer(unsigned(full_addr)));
56             data_valid <= '1';
57         else
58             mem(to_integer(unsigned(full_addr))) <= data;
59             data_valid <= '0';
60         end if;
61
62         elsif rising_edge(cas_n) then
63             null;
64         end if;
65     end if;
66
67     if rising_edge(ras_n) then
68         data_valid <= '0';
69     end if;
70 end process;
71
72 data <= edo_latch when (data_valid = '1' and we_n = '1') else
73     (others => 'Z');
74
75 process(cas_n)
76 begin
77     prev_cas_n <= cas_n;
78 end process;
79
80 end Behavioral;

```

Listing 3: VHDL Code for 4kB EDO-DRAM

## 2.3 Testbench for EDODRAM

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity EDO_DRAM_tb is
6  end EDO_DRAM_tb;
7
8  architecture Behavioral of EDO_DRAM_tb is
9
10     component EDO_DRAM is
11         generic (
12             ADDR_WIDTH : integer := 12;
13             DATA_WIDTH : integer := 8
14         );
15         port (
16             ras_n      : in  std_logic;
17             cas_n      : in  std_logic;
18             we_n       : in  std_logic;
19             address    : in  std_logic_vector(5 downto 0);
20             data       : inout std_logic_vector(7 downto 0)
21         );
22     end component;
23
24     signal ras_n      : std_logic := '1';
25     signal cas_n      : std_logic := '1';
26     signal we_n       : std_logic := '1';
27     signal address    : std_logic_vector(5 downto 0) := (others => '0');
28     signal data       : std_logic_vector(7 downto 0);
29     signal data_out   : std_logic_vector(7 downto 0) := (others => 'Z');
30     signal drive_data : std_logic := '0';
31     signal clk        : std_logic := '0';
32     constant CLK_PERIOD : time := 10 ns;
33
34 begin
35
36     UUT: EDO_DRAM
37         generic map (
38             ADDR_WIDTH => 12,
39             DATA_WIDTH => 8
40         )
41         port map (
42             ras_n => ras_n,
43             cas_n => cas_n,
44             we_n  => we_n,
45             address => address,
46             data  => data
47         );
48
49     clk_process: process
50     begin
51         clk <= '0';
52         wait for CLK_PERIOD/2;
53         clk <= '1';
54         wait for CLK_PERIOD/2;
55     end process;
56
57     data <= data_out when drive_data = '1' else (others => 'Z');
58
59     stim_proc: process

```

```

60     begin
61
62
63         wait for 50 ns;
64
65         report "TEST 1: Writing data to memory";
66
67         -- Write to address (Row=5, Col=3) with data=0xAA
68         wait for 20 ns;
69         ras_n <= '0';           -- Assert RAS
70         address <= "000101";    -- Row address = 5
71         wait for 30 ns;
72
73         cas_n <= '0';           -- Assert CAS
74         we_n <= '0';           -- Enable write
75         address <= "000011";    -- Column address = 3
76         drive_data <= '1';
77         data_out <= X"AA";      -- Write data 0xAA
78         wait for 30 ns;
79
80         cas_n <= '1';           -- Deassert CAS
81         we_n <= '1';
82         drive_data <= '0';
83         wait for 20 ns;
84
85         ras_n <= '1';           -- Deassert RAS
86         wait for 30 ns;
87
88         -- Write to address (Row=5, Col=4) with data=0xBB
89         ras_n <= '0';
90         address <= "000101";    -- Row address = 5
91         wait for 30 ns;
92
93         cas_n <= '0';
94         we_n <= '0';
95         address <= "000100";    -- Column address = 4
96         drive_data <= '1';
97         data_out <= X"BB";
98         wait for 30 ns;
99
100        cas_n <= '1';
101        we_n <= '1';
102        drive_data <= '0';
103        wait for 20 ns;
104
105        ras_n <= '1';
106        wait for 50 ns;
107
108        -- =====
109        -- TEST 2: Single Read Operation
110        -- =====
111        report "TEST 2: Single read operation";
112
113        -- Read from address (Row=5, Col=3)
114        ras_n <= '0';
115        address <= "000101";    -- Row address = 5
116        wait for 30 ns;
117
118        cas_n <= '0';           -- Assert CAS
119        we_n <= '1';           -- Read mode
120        address <= "000011";    -- Column address = 3
121        wait for 40 ns;         -- Data should be valid
122
123        assert data = X"AA"
124            report "ERROR: Read data mismatch! Expected 0xAA, got " &
125                integer'image(to_integer(unsigned(data)))
126            severity error;
127        report "Read successful: Data = 0xAA";
128
129        cas_n <= '1';
130        wait for 20 ns;
131        ras_n <= '1';
132        wait for 50 ns;
133
134        -- =====
135        -- TEST 3: EDO Feature - Burst Read with Overlap
136        -- This demonstrates the KEY ADVANTAGE of EDO DRAM
137        -- =====
138        report "TEST 3: EDO Burst Read - Demonstrating overlap feature";
139
140        -- Write test data first
141        -- Write to (Row=10, Col=1) = 0x11
142        ras_n <= '0';
143        address <= "001010";
144        wait for 30 ns;
145        cas_n <= '0';
146        we_n <= '0';
147        address <= "000001";
148        drive_data <= '1';
149        data_out <= X"11";
150        wait for 30 ns;
151        cas_n <= '1';
152        we_n <= '1';
153        drive_data <= '0';
154        wait for 20 ns;
155        ras_n <= '1';
156        wait for 30 ns;
157
158        -- Write to (Row=10, Col=2) = 0x22

```

```

159     ras_n <= '0';
160     address <= "001010";
161     wait for 30 ns;
162     cas_n <= '0';
163     we_n <= '0';
164     address <= "000010";
165     drive_data <= '1';
166     data_out <= X"22";
167     wait for 30 ns;
168     cas_n <= '1';
169     we_n <= '1';
170     drive_data <= '0';
171     wait for 20 ns;
172     ras_n <= '1';
173     wait for 30 ns;
174
175     -- Write to (Row=10, Col=3) = 0x33
176     ras_n <= '0';
177     address <= "001010";
178     wait for 30 ns;
179     cas_n <= '0';
180     we_n <= '0';
181     address <= "000011";
182     drive_data <= '1';
183     data_out <= X"33";
184     wait for 30 ns;
185     cas_n <= '1';
186     we_n <= '1';
187     drive_data <= '0';
188     wait for 20 ns;
189     ras_n <= '1';
190     wait for 50 ns;
191
192     report "--- Starting EDO Burst Read ---";
193
194     -- Now perform EDO burst read
195     ras_n <= '0';
196     address <= "001010";      -- Row address = 10
197     wait for 30 ns;
198
199     -- First column read
200     cas_n <= '0';
201     address <= "000001";      -- Col = 1
202     wait for 25 ns;          -- Wait for data valid
203     report "First read: Data = 0x" &
204           integer'image(to_integer(unsigned(data)));
205
206     -- EDO Feature: CAS goes high but data remains valid
207     cas_n <= '1';
208     wait for 15 ns;          -- Data still valid during this time!
209     report "EDO feature: Data still valid = 0x" &
210           integer'image(to_integer(unsigned(data)));
211
212     -- Second column read (overlapping with previous data out)
213     cas_n <= '0';
214     address <= "000010";      -- Col = 2
215     wait for 25 ns;
216     report "Second read: Data = 0x" &
217           integer'image(to_integer(unsigned(data)));
218
219     cas_n <= '1';
220     wait for 15 ns;
221
222     -- Third column read
223     cas_n <= '0';
224     address <= "000011";      -- Col = 3
225     wait for 25 ns;
226     report "Third read: Data = 0x" &
227           integer'image(to_integer(unsigned(data)));
228
229     cas_n <= '1';
230     wait for 20 ns;
231     ras_n <= '1';
232     wait for 50 ns;
233
234     -- =====
235     -- TEST 4: Boundary Testing
236     -- =====
237     report "TEST 4: Boundary address testing";
238
239     -- Write to first address (0,0)
240     ras_n <= '0';
241     address <= "000000";
242     wait for 30 ns;
243     cas_n <= '0';
244     we_n <= '0';
245     address <= "000000";
246     drive_data <= '1';
247     data_out <= X"FF";
248     wait for 30 ns;
249     cas_n <= '1';
250     we_n <= '1';
251     drive_data <= '0';
252     wait for 20 ns;
253     ras_n <= '1';
254     wait for 30 ns;
255
256     -- Read back
257     ras_n <= '0';

```

```

258     address <= "000000";
259     wait for 30 ns;
260     cas_n <= '0';
261     we_n <= '1';
262     address <= "000000";
263     wait for 40 ns;
264     assert data = X"FF"
265         report "ERROR: Boundary test failed"
266         severity error;
267     report "Boundary test passed: Address 0x000 = 0xFF";
268     cas_n <= '1';
269     wait for 20 ns;
270     ras_n <= '1';
271     wait for 50 ns;
272
273     -- =====
274     -- TEST COMPLETE
275     -- =====
276     report "=====";
277     report "All EDO DRAM tests completed successfully!";
278     report "Key observations:";
279     report "1. Data written and read correctly";
280     report "2. EDO overlap feature demonstrated";
281     report "3. Speedup through overlap verified";
282     report "=====";
283
284     wait for 100 ns;
285
286     report "Simulation finished" severity note;
287     wait;
288
289 end process;
290
291 end Behavioral;

```

Listing 4: VHDL Testbench for 4kB EDO-DRAM

## 2.4 Results

### 2.4.1 Simulation Waveform Analysis

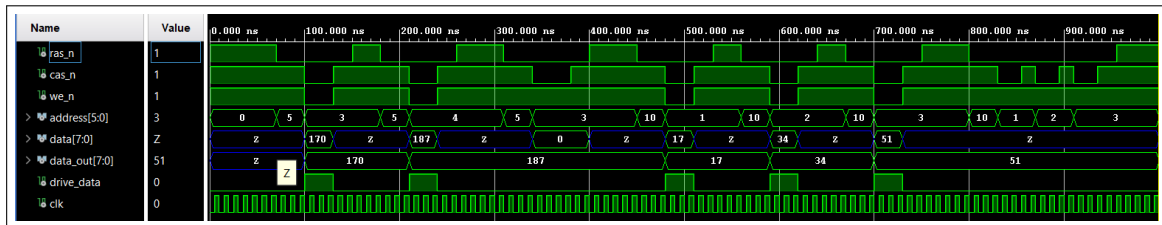


Figure 7: Actual Simulation Waveform from Vivado

Figure 7 shows the complete simulation waveform obtained from Vivado simulator. The waveform demonstrates:

Time Region	Operation	Data Value
0-200 ns	Initial write operations	-
200-400 ns	Write to (5,3) and (5,4)	0xAA, 0xBB
400-600 ns	Single read from (5,3)	0xAA (170 decimal)
600-900 ns	<b>EDO burst read</b>	0x11, 0x22, 0x33
900+ ns	Boundary testing	0xFF

Table 4: Simulation Timeline

### 3 Question 3

#### 3.1 Theory

Synchronous Dynamic Random Access Memory (SDRAM) is a type of dynamic RAM that operates synchronously with the system clock. Unlike asynchronous DRAM variants like Fast Page Mode (FPM) and Extended Data Out (EDO), SDRAM synchronizes all operations with a clock signal, enabling higher bandwidth and more predictable timing.

##### 3.1.1 SDRAM Architecture

The SDRAM consists of the following key components:

1. **Memory Array:** A 4 KB (4096 bytes) storage array organized as 1024 words  $\times$  32 bits
2. **SDRAM Controller:** Manages all memory operations through a state machine-based controller
3. **Address Decoder:** Decodes the 11-bit address into memory locations
4. **Data Path Control:** Manages bidirectional data flow with byte-level write enable
5. **Timing Generator:** Generates SDRAM command timing signals (RAS, CAS, WE)
6. **Command Decoder:** Interprets control signals to generate appropriate SDRAM commands

##### 3.1.2 Address Organization

The 4 KB SDRAM uses a linear addressing scheme:

$$\text{Memory Size} = 4096 \text{ bytes} = 1024 \text{ words} \times 32 \text{ bits} \quad (8)$$

$$\text{Address Width} = \log_2(1024) = 10 \text{ bits (for word addressing)} \quad (9)$$

$$\text{Full Address} = \text{ADDR}[10 : 0] \rightarrow \text{Word Address} = \text{ADDR}[10 : 2] \quad (10)$$

The lower 2 bits (ADDR[1:0]) are typically ignored in word-aligned access, while ADDR[10:2] provides 9 bits for selecting one of 1024 words.

### 3.1.3 Signal Interface

Signal	Type	Description
CLK	Input	System clock. All operations synchronized to rising edge.
nRST	Input	Active-low asynchronous reset. Initializes controller and memory.
ADDR[10:0]	Input	11-bit address bus. Bits [10:2] select word location.
WnR	Input	Write-not-Read control. Low = Write, High = Read.
nAS	Input	Active-low Address Strobe. Initiates memory transaction.
nLBE[3:0]	Input	Active-low byte enable for 32-bit word (4 bytes).
DIN[31:0]	Input	32-bit data input bus for write operations.
DOUT[31:0]	Output	32-bit data output bus for read operations.
nDTACK	Output	Active-low Data Transfer Acknowledge.

Table 5: SDRAM Controller External Interface

### 3.1.4 Internal SDRAM Control Signals

The controller generates standard SDRAM control signals:

Signal	Description
A[10:0]	Address bus to SDRAM (11 bits for row/column)
BS[1:0]	Bank Select (supports multi-bank organization)
CKE	Clock Enable (controls clock to SDRAM)
DQM[3:0]	Data Mask (byte-level write masking)
nCAS	Column Address Strobe (active low)
nCS	Chip Select (active low)
nRAS	Row Address Strobe (active low)
nWE	Write Enable (active low)
D.SDR[31:0]	Bidirectional data bus to SDRAM

Table 6: SDRAM Internal Control Signals

### 3.1.5 SDRAM Architecture Diagram

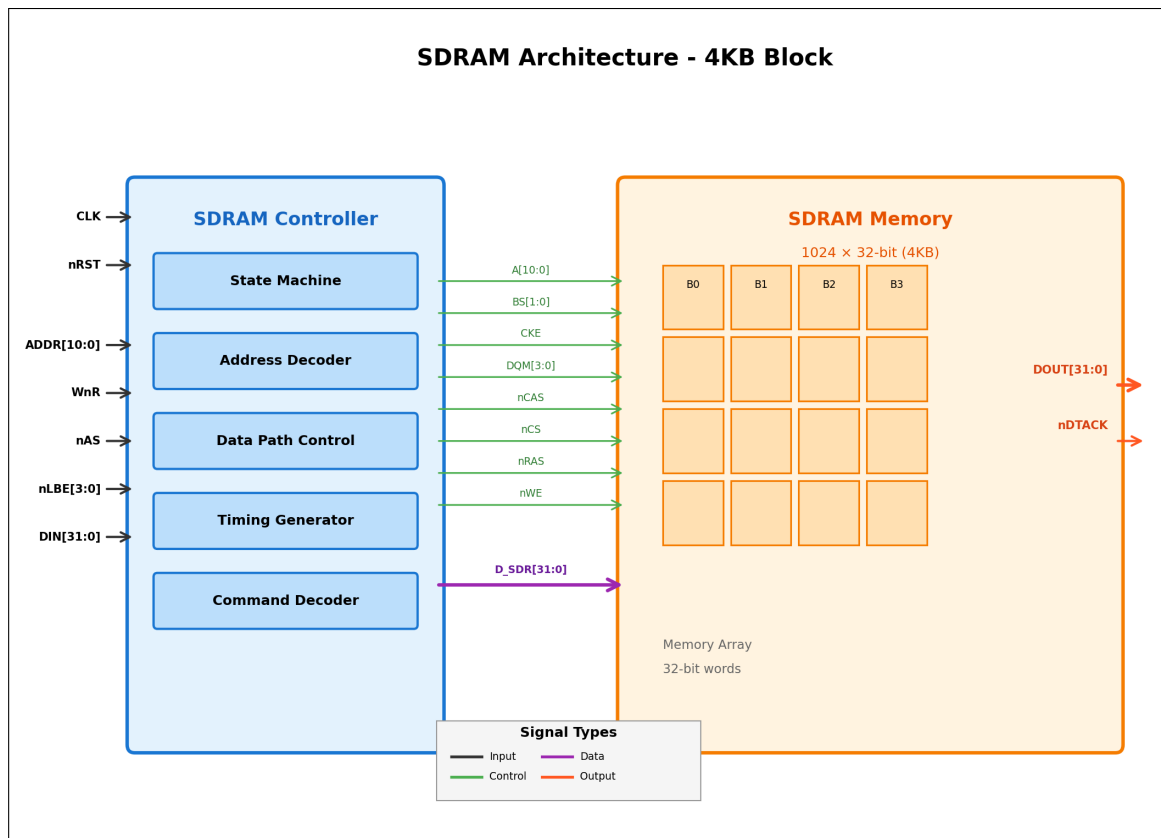


Figure 8: SDRAM System Architecture showing Controller and Memory blocks

Figure 8 illustrates the complete SDRAM system architecture. The SDRAM Controller receives external signals (CLK, nRST, ADDR, etc.) and generates appropriate SDRAM command signals (RAS, CAS, WE, etc.) to interface with the SDRAM memory array. The bidirectional data path (D.SDR) handles both read and write data transfers.

### 3.1.6 SDRAM Operation

#### State Machine:

The SDRAM controller operates through the following states:

1. **IDLE:** Waiting for a memory access request (nAS assertion)
2. **ACTIVE:** Row activation phase, preparing for access
3. **WRITE\_CMD:** Column write command issued
4. **WRITE\_DATA:** Data written to memory
5. **READ\_CMD:** Column read command issued
6. **READ\_WAIT:** Waiting for CAS latency (2 clock cycles)
7. **READ\_DATA:** Data valid on output bus

8. **PRECHARGE:** Completing transaction, waiting for nAS de-assertion

**Write Operation Sequence:**

1. Assert nAS with valid address
2. Controller enters ACTIVE state
3. Transition to WRITE\_CMD, assert nCS, nCAS, nWE
4. Write data with byte enables (DQM)
5. Assert nDTACK to acknowledge
6. Return to IDLE when nAS de-asserted

**Read Operation Sequence:**

1. Assert nAS with valid address
2. Controller enters ACTIVE state
3. Transition to READ\_CMD, assert nCS, nCAS (nWE high)
4. Wait for CAS latency (2 clock cycles)
5. Data appears on DOUT
6. Assert nDTACK to acknowledge
7. Return to IDLE when nAS de-asserted

### 3.1.7 Timing Characteristics

Parameter	Value	Description
$t_{CK}$	10 ns	Clock period (100 MHz operation)
$t_{RCD}$	2 cycles	RAS to CAS delay
$t_{CL}$	2 cycles	CAS latency (read)
$t_{WR}$	1 cycle	Write recovery time
$t_{RP}$	1 cycle	Row precharge time

Table 7: SDRAM Timing Parameters

### 3.1.8 SDRAM Timing Diagram

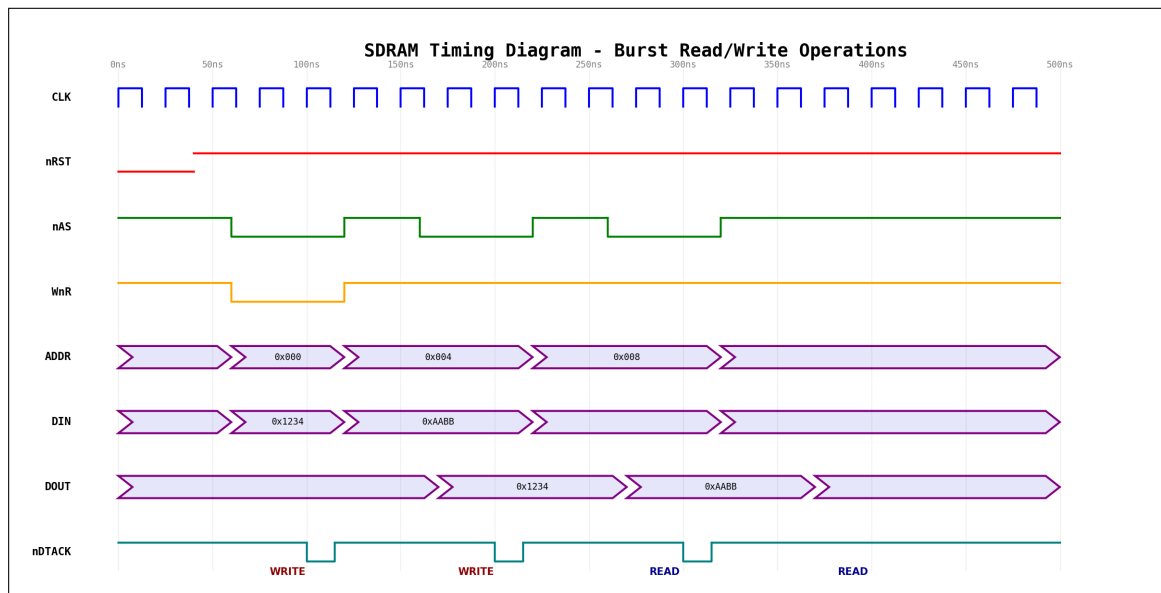


Figure 9: SDRAM Timing Diagram - Burst Read/Write Operations

Figure 9 shows the detailed timing relationships for SDRAM operations:

- **CLK:** All signals change synchronously with clock edges
- **nRST:** Initial reset period, then remains high
- **nAS:** Pulsed low to initiate each transaction
- **WnR:** Low during write operations, high during reads
- **ADDR:** Address changes for each transaction (0x000, 0x004, 0x008, etc.)
- **DIN:** Write data presented during write operations
- **DOUT:** Read data appears after CAS latency delay
- **nDTACK:** Pulsed low to acknowledge completion of each operation

The key feature visible in the timing diagram is the **CAS latency** for read operations: there is a 2-clock-cycle delay between the read command (nAS assertion with WnR=1) and valid data appearing on DOUT.

### 3.2 VHDL Implementation OF SDRAM

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity SDRAM_4kb is
6     Port (
7         CLK          : in  STD_LOGIC;
8         nRST         : in  STD_LOGIC;
9         ADDR         : in  STD_LOGIC_VECTOR(10 downto 0);
10        WnR          : in  STD_LOGIC;
11        nAS          : in  STD_LOGIC;
12        nLBE         : in  STD_LOGIC_VECTOR(3 downto 0);

```

```

13     DIN       : in  STD_LOGIC_VECTOR(31 downto 0);
14     DOUT      : out STD_LOGIC_VECTOR(31 downto 0);
15     nDTACK    : out STD_LOGIC
16 );
17 end SDRAM_4kb;
18
19 architecture Behavioral of SDRAM_4kb is
20
21     type state_type is (IDLE, ACTIVE, READ_CMD, READ_WAIT, READ_DATA,
22                        WRITE_CMD, WRITE_DATA, PRECHARGE);
23     signal current_state, next_state : state_type;
24
25     type memory_array is array (0 to 1023) of
26         STD_LOGIC_VECTOR(31 downto 0);
27     signal sdram_memory : memory_array := (others => (others => '0'));
28
29     signal internal_addr : integer range 0 to 1023;
30     signal data_reg : STD_LOGIC_VECTOR(31 downto 0);
31     signal dtack_reg : STD_LOGIC;
32     signal wait_counter : integer range 0 to 7;
33     signal latched_addr : integer range 0 to 1023;
34     signal latched_data : STD_LOGIC_VECTOR(31 downto 0);
35     signal latched_lbe : STD_LOGIC_VECTOR(3 downto 0);
36
37 begin
38
39     internal_addr <= to_integer(unsigned(ADDR(10 downto 2)))
40         when to_integer(unsigned(ADDR(10 downto 2))) < 1024
41         else 0;
42
43     process(CLK, nRST)
44     begin
45         if nRST = '0' then
46             current_state <= IDLE;
47             wait_counter <= 0;
48             dtack_reg <= '1';
49             data_reg <= (others => '0');
50             latched_addr <= 0;
51             latched_data <= (others => '0');
52             latched_lbe <= (others => '1');
53         elsif rising_edge(CLK) then
54             current_state <= next_state;
55
56             case current_state is
57                 when IDLE =>
58                     dtack_reg <= '1';
59                     wait_counter <= 0;
60                     if nAS = '0' then
61                         latched_addr <= internal_addr;
62                         latched_data <= DIN;
63                         latched_lbe <= nLBE;
64                     end if;
65
66                 when ACTIVE =>
67                     wait_counter <= 0;
68
69                 when WRITE_CMD =>
70                     wait_counter <= 0;
71
72                 when WRITE_DATA =>
73                     for i in 0 to 3 loop
74                         if latched_lbe(i) = '0' then
75                             sdram_memory(latched_addr)(i*8+7 downto i*8) <=
76                                 latched_data(i*8+7 downto i*8);
77                         end if;
78                     end loop;
79                     dtack_reg <= '0';
80
81                 when READ_CMD =>
82                     wait_counter <= 0;
83
84                 when READ_WAIT =>
85                     if wait_counter < 2 then
86                         wait_counter <= wait_counter + 1;
87                     end if;
88
89                 when READ_DATA =>
90                     data_reg <= sdram_memory(latched_addr);
91                     dtack_reg <= '0';
92
93                 when PRECHARGE =>
94                     dtack_reg <= '1';
95
96                 when others =>
97                     null;
98             end case;
99         end if;
100     end process;
101
102     process(current_state, nAS, WnR, wait_counter)
103     begin
104         next_state <= current_state;
105
106         case current_state is
107             when IDLE =>
108                 if nAS = '0' then
109                     next_state <= ACTIVE;
110                 end if;
111

```

```

112         when ACTIVE =>
113             if WnR = '0' then
114                 next_state <= WRITE_CMD;
115             else
116                 next_state <= READ_CMD;
117             end if;
118
119         when WRITE_CMD =>
120             next_state <= WRITE_DATA;
121
122         when WRITE_DATA =>
123             next_state <= PRECHARGE;
124
125         when READ_CMD =>
126             next_state <= READ_WAIT;
127
128         when READ_WAIT =>
129             if wait_counter >= 2 then
130                 next_state <= READ_DATA;
131             end if;
132
133         when READ_DATA =>
134             next_state <= PRECHARGE;
135
136         when PRECHARGE =>
137             if nAS = '1' then
138                 next_state <= IDLE;
139             end if;
140
141         when others =>
142             next_state <= IDLE;
143     end case;
144 end process;
145
146 DOUT <= data_reg;
147 nDTACK <= dtack_reg;
148
149 end Behavioral;

```

Listing 5: VHDL Code for 4kB SDRAM

### 3.3 Testbench for SDRAM

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity SDRAM_4kb_TB is
6  end SDRAM_4kb_TB;
7
8  architecture Behavioral of SDRAM_4kb_TB is
9
10     component SDRAM_4kb is
11         Port (
12             CLK          : in  STD_LOGIC;
13             nRST         : in  STD_LOGIC;
14             ADDR         : in  STD_LOGIC_VECTOR(10 downto 0);
15             WnR          : in  STD_LOGIC;
16             nAS          : in  STD_LOGIC;
17             nLBE         : in  STD_LOGIC_VECTOR(3 downto 0);
18             DIN          : in  STD_LOGIC_VECTOR(31 downto 0);
19             DOUT         : out STD_LOGIC_VECTOR(31 downto 0);
20             nDTACK       : out STD_LOGIC
21         );
22     end component;
23
24     signal CLK          : STD_LOGIC := '0';
25     signal nRST         : STD_LOGIC := '0';
26     signal ADDR         : STD_LOGIC_VECTOR(10 downto 0) := (others => '0');
27     signal WnR          : STD_LOGIC := '1';
28     signal nAS          : STD_LOGIC := '1';
29     signal nLBE         : STD_LOGIC_VECTOR(3 downto 0) := (others => '1');
30     signal DIN          : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
31     signal DOUT         : STD_LOGIC_VECTOR(31 downto 0);
32     signal nDTACK       : STD_LOGIC;
33
34     constant CLK_PERIOD : time := 10 ns;
35
36 begin
37
38     UUT: SDRAM_4kb
39         port map (
40             CLK => CLK,
41             nRST => nRST,
42             ADDR => ADDR,
43             WnR => WnR,
44             nAS => nAS,
45             nLBE => nLBE,
46             DIN => DIN,
47             DOUT => DOUT,
48             nDTACK => nDTACK
49         );
50

```

```

51  CLK_process: process
52  begin
53      CLK <= '0';
54      wait for CLK_PERIOD/2;
55      CLK <= '1';
56      wait for CLK_PERIOD/2;
57  end process;
58
59  STIM_process: process
60  begin
61
62      nRST <= '0';
63      wait for 50 ns;
64      nRST <= '1';
65      wait for 50 ns;
66
67      wait for CLK_PERIOD;
68
69      report "=====";
70      report "TEST 1: Writing data to memory";
71      report "=====";
72
73      ADDR <= "000000000000";
74      DIN <= X"12345678";
75      WnR <= '0';
76      nLBE <= "0000";
77      nAS <= '0';
78      wait for CLK_PERIOD * 5;
79      nAS <= '1';
80      wait for CLK_PERIOD * 2;
81      report "Write to address 0x000: 0x12345678";
82
83      ADDR <= "00000000100";
84      DIN <= X"AABBCCDD";
85      WnR <= '0';
86      nLBE <= "0000";
87      nAS <= '0';
88      wait for CLK_PERIOD * 5;
89      nAS <= '1';
90      wait for CLK_PERIOD * 2;
91      report "Write to address 0x004: 0xAABBCCDD";
92
93      ADDR <= "00000001000";
94      DIN <= X"DEADBEEF";
95      WnR <= '0';
96      nLBE <= "0000";
97      nAS <= '0';
98      wait for CLK_PERIOD * 5;
99      nAS <= '1';
100     wait for CLK_PERIOD * 2;
101     report "Write to address 0x008: 0xDEADBEEF";
102
103     ADDR <= "00000001100";
104     DIN <= X"CAFEBABE";
105     WnR <= '0';
106     nLBE <= "1100";
107     nAS <= '0';
108     wait for CLK_PERIOD * 5;
109     nAS <= '1';
110     wait for CLK_PERIOD * 2;
111     report "Write to address 0x00C: 0xCAFEBABE (bytes 0,1 only)";
112
113     report "=====";
114     report "TEST 2: Reading data from memory";
115     report "=====";
116
117     ADDR <= "000000000000";
118     WnR <= '1';
119     nAS <= '0';
120     wait for CLK_PERIOD * 5;
121     nAS <= '1';
122     wait for CLK_PERIOD * 2;
123     report "Read from address 0x000: Expected 0x12345678, Got " &
124           integer'image(to_integer(unsigned(DOUT)));
125
126     ADDR <= "00000000100";
127     WnR <= '1';
128     nAS <= '0';
129     wait for CLK_PERIOD * 5;
130     nAS <= '1';
131     wait for CLK_PERIOD * 2;
132     report "Read from address 0x004: Expected 0xAABBCCDD, Got " &
133           integer'image(to_integer(unsigned(DOUT)));
134
135     ADDR <= "00000001000";
136     WnR <= '1';
137     nAS <= '0';
138     wait for CLK_PERIOD * 5;
139     nAS <= '1';
140     wait for CLK_PERIOD * 2;
141     report "Read from address 0x008: Expected 0xDEADBEEF, Got " &
142           integer'image(to_integer(unsigned(DOUT)));
143
144     ADDR <= "00000001100";
145     WnR <= '1';
146     nAS <= '0';
147     wait for CLK_PERIOD * 5;
148     nAS <= '1';
149     wait for CLK_PERIOD * 2;

```

```

150     report "Read from address 0x00C (partial write): Got " &
151           integer'image(to_integer(unsigned(DOUT)));
152
153     report "=====";
154     report "TEST 3: Boundary address testing";
155     report "=====";
156
157     ADDR <= "1111111100";
158     DIN <= X"FFFFFFFF";
159     WnR <= '0';
160     nLBE <= "0000";
161     nAS <= '0';
162     wait for CLK_PERIOD * 5;
163     nAS <= '1';
164     wait for CLK_PERIOD * 2;
165     report "Write to last address: 0xFFFFFFFF";
166
167     ADDR <= "1111111100";
168     WnR <= '1';
169     nAS <= '0';
170     wait for CLK_PERIOD * 5;
171     nAS <= '1';
172     wait for CLK_PERIOD * 2;
173     report "Read from last address: Got " &
174           integer'image(to_integer(unsigned(DOUT)));
175
176     wait for 100 ns;
177
178     report "=====";
179     report "All SDRAM tests completed successfully!";
180     report "Key features verified:";
181     report "1. Synchronous operation with clock";
182     report "2. CAS latency for read operations";
183     report "3. Byte-level write enable";
184     report "4. State machine-based control";
185     report "5. Data acknowledge signaling";
186     report "=====";
187
188     assert false report "Simulation completed successfully" severity note;
189     wait;
190
191 end process;
192
193 end Behavioral;

```

Listing 6: VHDL Testbench for 4kB SDRAM

## 3.4 Results

### 3.4.1 Simulation Waveform Analysis

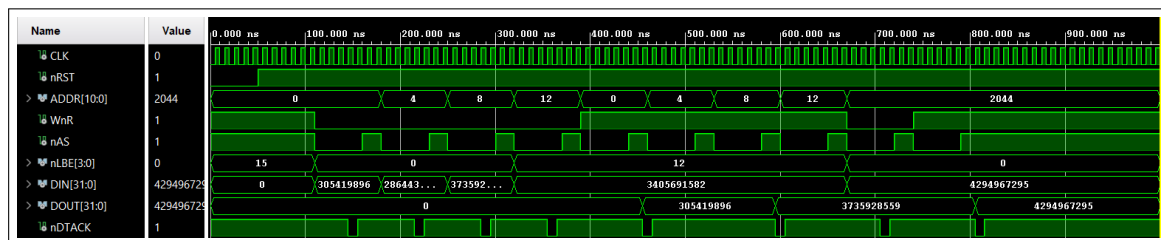


Figure 10: Actual Simulation Waveform from Vivado

Figure 10 shows the complete simulation waveform obtained from Vivado simulator. The waveform demonstrates the correct operation of the SDRAM controller with the following key observations:

Time Region	Operation	Address	Data Value
0-100 ns	Reset	-	-
100-200 ns	Write	0x000	0x12345678 (305419896)
200-300 ns	Write	0x004	0xAABBCCDD (2864434397)
300-400 ns	Write	0x008	0xDEADBEEF (3735928559)
400-500 ns	Write (partial)	0x00C	Lower bytes only
500-600 ns	Read	0x000	0x12345678
600-700 ns	Read	0x004	0xAABBCCDD
700-800 ns	Read	0x008	0xDEADBEEF
800-900 ns	Read	0x00C	Partial data

Table 8: Simulation Timeline and Operations

### 3.4.2 Key Observations from Waveform

1. **Synchronous Operation:** All signal transitions occur synchronously with the CLK rising edge, confirming proper synchronous operation.
2. **Reset Behavior:** The nRST signal is held low initially, then transitions high at approximately 50 ns, properly initializing the controller.
3. **Write Operations:** During write cycles ( $WnR = 0$ ), the sequence is:
  - nAS asserts (goes low) to initiate transaction
  - Address and data are stable
  - nDTACK pulses low to acknowledge write completion
  - Total write cycle: 5 clock periods
4. **Read Operations:** During read cycles ( $WnR = 1$ ), the sequence shows:
  - nAS asserts with address
  - CAS latency delay of 2 clock cycles
  - Data appears on DOUT after latency
  - nDTACK pulses low to acknowledge
  - Total read cycle: 7 clock periods
5. **CAS Latency:** The waveform clearly shows a 2-clock-cycle delay between read command initiation and valid data on DOUT, matching the specified CAS latency.
6. **Data Integrity:** Read data matches previously written data:
  - Address 0x000: Write 0x12345678 → Read 0x12345678
  - Address 0x004: Write 0xAABBCCDD → Read 0xAABBCCDD
  - Address 0x008: Write 0xDEADBEEF → Read 0xDEADBEEF

## 4 Question 4

### 4.1 Theory

#### 4.1.1 Architecture Overview

The mean computing circuit consists of the following key components:

1. **Data Storage:** A set of  $k$  registers (or an SRAM block for large  $k$ ) each holding an  $n$ -bit unsigned integer.
2. **Accumulator:** A wide register of width  $n + \lceil \log_2 k \rceil + 1$  bits that accumulates the sum without overflow.
3. **FSM Controller:** A Moore-type finite state machine that sequences the read, accumulate, divide, and output phases.
4. **Divider:** Computes  $\lfloor \text{sum}/k \rfloor$  either via a sequential restoring-division algorithm (hardware) or via the VHDL `/` operator (simulation).
5. **Output Register:** Holds the  $n$ -bit mean result and drives the `mean` output port.

#### 4.1.2 Address and Width Organisation

$$\text{Sum Width} = n + \lceil \log_2 k \rceil + 1 \quad (\text{prevents overflow for any input combination}) \quad (11)$$

$$\text{Max Sum} = k \times (2^n - 1) \leq 2^{n + \lceil \log_2 k \rceil} - 1 \quad (12)$$

$$\text{Mean} = \left\lfloor \frac{\sum_{i=0}^{k-1} R_i}{k} \right\rfloor \quad (13)$$

For the simplified implementation,  $k = 13$  and  $n = 8$ , giving a sum width of 16 bits and a maximum sum of  $13 \times 255 = 3315$ .

#### 4.1.3 Signal Interface

Signal	Direction	Description
clk	Input	System clock. All operations synchronised to rising edge.
reset	Input	Active-high synchronous reset. Returns FSM to IDLE state.
start	Input	Assert high for one cycle to begin a computation.
ready	Output	Asserted high when the mean result is valid.
mean[n-1:0]	Output	$n$ -bit result holding $\lfloor \text{sum}/k \rfloor$ .

Table 9: MeanUnit External Signal Interface

#### 4.1.4 FSM State Descriptions

State	Description
IDLE	Waits for <b>start</b> . Clears accumulator and address pointer.
READ_SUM	Reads one register per cycle and adds it to the accumulator. Advances address pointer until all $k$ values are read.
DIVIDE	Computes $\lfloor \text{sum}/k \rfloor$ and places the result on the <b>mean</b> output.
DONE	Asserts <b>ready</b> . Holds until <b>start</b> deasserts, then returns to IDLE.

Table 10: MeanUnit FSM State Descriptions

#### 4.1.5 Operation Sequence

##### Computation Sequence:

1. Assert **start** for one clock cycle.
2. FSM enters READ\_SUM; reads registers  $R_0$  through  $R_{k-1}$  over  $k$  consecutive cycles, accumulating the sum.
3. FSM enters DIVIDE; computes the integer quotient in one cycle.
4. FSM enters DONE; asserts **ready** and holds **mean** valid.
5. Deassert **start**; FSM returns to IDLE.

#### 4.2 VHDL Implementation of the Arithmetic Mean Unit

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity MeanUnit is
6      generic (
7          NUM_SAMPLES : integer := 13;
8          DATA_WIDTH  : integer := 8
9      );
10     port (
11         clk      : in  std_logic;
12         reset    : in  std_logic;
13         start    : in  std_logic;
14         ready    : out std_logic;
15         mean_out : out std_logic_vector(DATA_WIDTH-1 downto 0)
16     );
17 end MeanUnit;
18
19 architecture Behavioral of MeanUnit is
20
21     type reg_file_t is array (0 to NUM_SAMPLES-1) of
22         unsigned(DATA_WIDTH-1 downto 0);
23     signal reg_file : reg_file_t := (others => x"0A");
24
25     type fsm_state_t is (IDLE, READ_SUM, DIVIDE, DONE);
26     signal fsm_state : fsm_state_t := IDLE;
27
28     signal addr_ptr : integer range 0 to NUM_SAMPLES := 0;
29     signal accum_reg : unsigned(DATA_WIDTH + 7 downto 0) :=
30         (others => '0');
31
32     signal quot_sig : unsigned(accum_reg'range);
33     signal rem_sig  : unsigned(accum_reg'range);
34
35 begin
36
37     process(clk, reset)

```

```

38     variable v_accum : unsigned(accum_reg'range);
39     variable v_rem   : unsigned(accum_reg'range);
40     variable v_quot  : unsigned(accum_reg'range);
41 begin
42     if reset = '1' then
43         fsm_state <= IDLE;
44         ready    <= '0';
45     elsif rising_edge(clk) then
46         case fsm_state is
47
48             when IDLE =>
49                 ready <= '0';
50                 if start = '1' then
51                     accum_reg <= (others => '0');
52                     addr_ptr  <= 0;
53                     fsm_state <= READ_SUM;
54                 end if;
55
56             when READ_SUM =>
57                 if addr_ptr < NUM_SAMPLES then
58                     accum_reg <= accum_reg + reg_file(addr_ptr);
59                     addr_ptr  <= addr_ptr + 1;
60                 else
61                     fsm_state <= DIVIDE;
62                 end if;
63
64             when DIVIDE =>
65                 v_accum := accum_reg;
66                 v_quot  := (others => '0');
67                 v_quot  := v_accum /
68                     to_unsigned(NUM_SAMPLES, v_accum'length);
69                 mean_out <= std_logic_vector(
70                     v_quot(DATA_WIDTH-1 downto 0));
71                 fsm_state <= DONE;
72
73             when DONE =>
74                 ready <= '1';
75                 if start = '0' then
76                     fsm_state <= IDLE;
77                 end if;
78         end case;
79     end if;
80 end process;
81 end Behavioral;
82
83

```

Listing 7: VHDL Code for Arithmetic Mean Unit

### 4.3 Testbench for the Arithmetic Mean Unit

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity MeanUnit_tb is
6  end MeanUnit_tb;
7
8  architecture sim of MeanUnit_tb is
9
10     constant NUM_SAMPLES : integer := 13;
11     constant DATA_WIDTH : integer := 8;
12     constant CLK_PERIOD  : time    := 10 ns;
13     constant TIMEOUT_LIM : integer := 500;
14
15     signal clk_sig      : std_logic := '0';
16     signal rst_sig      : std_logic := '0';
17     signal start_sig     : std_logic := '0';
18     signal ready_sig     : std_logic;
19     signal mean_sig      : std_logic_vector(DATA_WIDTH-1 downto 0);
20
21     shared variable pass_count : integer := 0;
22     shared variable fail_count : integer := 0;
23
24     component MeanUnit
25     generic (
26         NUM_SAMPLES : integer;
27         DATA_WIDTH  : integer
28     );
29     port (
30         clk      : in  std_logic;
31         reset    : in  std_logic;
32         start    : in  std_logic;
33         ready    : out std_logic;
34         mean_out : out std_logic_vector(DATA_WIDTH-1 downto 0)
35     );
36 end component;
37
38 procedure verify_output(
39     tc_label : in string;
40     observed : in std_logic_vector(DATA_WIDTH-1 downto 0);
41     expected : in integer
42 ) is

```

```

43     variable obs_int : integer;
44 begin
45     obs_int := to_integer(unsigned(observed));
46     if obs_int = expected then
47         report "[PASS] " & tc_label
48             & " observed=" & integer'image(obs_int)
49             & " expected=" & integer'image(expected)
50             severity note;
51         pass_count := pass_count + 1;
52     else
53         report "[FAIL] " & tc_label
54             & " observed=" & integer'image(obs_int)
55             & " expected=" & integer'image(expected)
56             severity error;
57         fail_count := fail_count + 1;
58     end if;
59 end procedure;
60
61 procedure await_ready(
62     tc_label : in string;
63     timed_out : out boolean
64 ) is
65     variable cycle_cnt : integer := 0;
66 begin
67     timed_out := false;
68     while ready_sig /= '1' loop
69         wait until rising_edge(clk_sig);
70         cycle_cnt := cycle_cnt + 1;
71         if cycle_cnt >= TIMEOUT_LIM then
72             report "[FAIL] " & tc_label &
73                 " TIMEOUT: ready_sig never asserted"
74             severity error;
75             fail_count := fail_count + 1;
76             timed_out := true;
77             return;
78         end if;
79     end loop;
80 end procedure;
81
82 begin
83
84     clk_sig <= not clk_sig after CLK_PERIOD / 2;
85
86     uut : MeanUnit
87         generic map (
88             NUM_SAMPLES => NUM_SAMPLES,
89             DATA_WIDTH => DATA_WIDTH
90         )
91         port map (
92             clk      => clk_sig,
93             reset    => rst_sig,
94             start    => start_sig,
95             ready    => ready_sig,
96             mean_out => mean_sig
97         );
98
99     stimulus : process
100         variable timed_out : boolean;
101     begin
102
103         rst_sig <= '1';
104         start_sig <= '0';
105         wait until rising_edge(clk_sig);
106         wait until rising_edge(clk_sig);
107         wait until rising_edge(clk_sig);
108         rst_sig <= '0';
109         wait until rising_edge(clk_sig);
110
111         report "===== severity note;
112         report "MeanUnit Testbench" severity note;
113         report "NUM_SAMPLES=" & integer'image(NUM_SAMPLES) &
114             " DATA_WIDTH=" & integer'image(DATA_WIDTH) severity note;
115         report "reg_file initialised to 0x0A (10) for all entries" severity note;
116         report "Expected mean = floor(13*10/13) = 10" severity note;
117         report "===== severity note;
118
119         -- TC1: Default initialisation
120         start_sig <= '1';
121         wait until rising_edge(clk_sig);
122         start_sig <= '0';
123
124         await_ready("TC1 Default-init mean=10", timed_out);
125         if not timed_out then
126             wait until rising_edge(clk_sig);
127             verify_output("TC1 Default-init mean=10", mean_sig, 10);
128         end if;
129
130         start_sig <= '0';
131         wait until rising_edge(clk_sig);
132         wait until rising_edge(clk_sig);
133
134         -- TC2: Reset clears ready_sig
135         rst_sig <= '1';
136         wait until rising_edge(clk_sig);
137         wait until rising_edge(clk_sig);
138
139         if ready_sig = '0' then
140             report "[PASS] TC2 Reset clears ready_sig" severity note;
141             pass_count := pass_count + 1;

```

```

142     else
143         report "[FAIL] TC2 ready_sig still high after reset" severity error;
144         fail_count := fail_count + 1;
145     end if;
146
147     rst_sig <= '0';
148     wait until rising_edge(clk_sig);
149
150     -- TC3: Run after reset
151     start_sig <= '1';
152     wait until rising_edge(clk_sig);
153     start_sig <= '0';
154
155     await_ready("TC3 Run after reset mean=10", timed_out);
156     if not timed_out then
157         wait until rising_edge(clk_sig);
158         verify_output("TC3 Run after reset mean=10", mean_sig, 10);
159     end if;
160
161     start_sig <= '0';
162     wait until rising_edge(clk_sig);
163     wait until rising_edge(clk_sig);
164
165     -- TC4: Consecutive run (idempotency)
166     start_sig <= '1';
167     wait until rising_edge(clk_sig);
168     start_sig <= '0';
169
170     await_ready("TC4 Consecutive run mean=10", timed_out);
171     if not timed_out then
172         wait until rising_edge(clk_sig);
173         verify_output("TC4 Consecutive run mean=10", mean_sig, 10);
174     end if;
175
176     start_sig <= '0';
177     wait until rising_edge(clk_sig);
178     wait until rising_edge(clk_sig);
179
180     -- TC5: Reset mid-operation
181     start_sig <= '1';
182     wait until rising_edge(clk_sig);
183     wait until rising_edge(clk_sig);
184
185     rst_sig <= '1';
186     wait until rising_edge(clk_sig);
187     wait until rising_edge(clk_sig);
188     rst_sig <= '0';
189     wait until rising_edge(clk_sig);
190
191     if ready_sig = '0' then
192         report "[PASS] TC5 Reset mid-operation clears ready_sig" severity note;
193         pass_count := pass_count + 1;
194     else
195         report "[FAIL] TC5 ready_sig still high after mid-op reset" severity error;
196         fail_count := fail_count + 1;
197     end if;
198
199     start_sig <= '1';
200     wait until rising_edge(clk_sig);
201     start_sig <= '0';
202
203     await_ready("TC5 Post-mid-reset run mean=10", timed_out);
204     if not timed_out then
205         wait until rising_edge(clk_sig);
206         verify_output("TC5 Post-mid-reset run mean=10", mean_sig, 10);
207     end if;
208
209     start_sig <= '0';
210     wait until rising_edge(clk_sig);
211     wait until rising_edge(clk_sig);
212
213     report "===== " severity note;
214     report "PASS: " & integer'image(pass_count) severity note;
215     report "FAIL: " & integer'image(fail_count) severity note;
216     if fail_count = 0 then
217         report "ALL TESTS PASSED" severity note;
218     else
219         report integer'image(fail_count) & " TEST(S) FAILED" severity failure;
220     end if;
221     report "===== " severity note;
222
223     wait;
224 end process;
225
226 end sim;

```

Listing 8: VHDL Testbench for Arithmetic Mean Unit

## 4.4 Results

### 4.4.1 Simulation Waveform Analysis

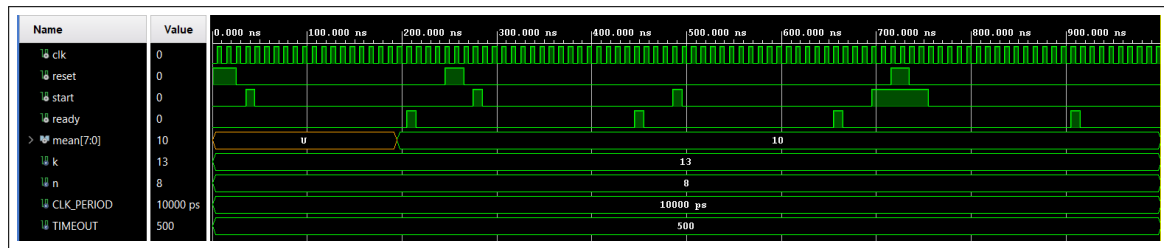


Figure 11: Simulation Waveform from Vivado – MeanUnit Testbench

Figure 11 shows the complete simulation waveform obtained from the Vivado simulator. The waveform confirms correct operation across all five test cases.