Abstract **2**
Introduction **2**
Specifications **3**
Behavioral / Functional 3
Write to Cache (Hit) 3
Read from Cache (Hit) 3
Read/Write on Miss (Dirty Bit = 0) 3
Read/Write on Miss (Dirty Bit = 1) 3
Idle and Ready States 4
Synchronization 4
Device design **4**
Symbol Diagram 4
Block Diagram 6
Process Diagram 7
State Diagram 8
Timing Diagram 9
Results **10**
Case 1 Hit: Write Word to Cache 10
Case 2 Hit: Read Word from Cache 10
Case 3 Miss: Read/Write, Dbit is equal to 0 11
Case 4 Miss: Read/Write, Dbit is equal to 1 11
Conclusion **12**
References **12**
Appendix **12**
CPU.vhd 12
SDRAM.vhd 15
SRAM.vhd 16
Constraints.ucf 22

Abstract

This project involves the design and implementation of a simplified Cache Controller using VHDL in the Xilinx ISE environment, with a Spartan-3E FPGA. The objective was to build and understand cache memory operation and its interaction with both SRAM (cache memory) and SDRAM (main memory) within a hierarchical memory system. The cache stores 256 bytes, organized into eight blocks of 32 bytes, and operates with a 16-bit CPU address divided into Tag (8 bits), Index (3 bits), and Offset (5 bits) fields for direct-mapped cache operation.

The controller manages four main behaviors: read/write hits, misses with clean blocks (Dirty Bit = 0), and misses with dirty blocks (Dirty Bit = 1). On hits, data is accessed directly from cache for minimal delay. On misses, clean blocks trigger a direct fetch from SDRAM, while dirty blocks are first written back to main memory before loading the new data. A finite state machine governs these operations, maintaining synchronization between CPU, cache, and memory.

Simulation waveform results confirmed correct controller behavior and timing. Cache hits achieved a 60 ns access time, while miss penalties were 1220 ns (clean block) and 2500 ns (dirty block). These results show that the controller effectively reduces average memory access time and improves performance by exploiting temporal and spatial locality. Overall, the project demonstrates how efficient cache management bridges the speed gap between the CPU and main memory in digital systems.

Introduction

Modern computer systems rely on a hierarchical memory structure to balance speed, cost, and capacity, with cache memory serving as a high-speed intermediary between the CPU and main memory. Since processors operate faster than main memory, the cache reduces access latency by storing recently accessed data that is likely to be reused, exploiting temporal and spatial locality. The COE758 Digital System Engineering Project 1 focuses on the design and implementation of a simplified cache controller using VHDL in the Xilinx ISE environment, targeting the Spartan-3E FPGA. The project provides practical experience in digital system design, logic control, and memory hierarchy implementation. The cache controller manages communication between the CPU, local cache (SRAM), and main memory (SDRAM), determining whether a data request results in a hit or miss and performing the corresponding read, write, or block replacement operation. Through this design, students gain a deeper understanding of key computer architecture concepts such as tag comparison, valid and dirty bit management, and synchronous data transfer. The controller operates on a 16-bit CPU address divided into tag, index, and offset fields, implementing direct-mapped cache behavior with eight blocks of 32 bytes each. Performance parameters such as hit time, miss penalty, and block replacement time are analyzed to evaluate efficiency. Ultimately, this project bridges theory and practice by applying memory hierarchy principles to real hardware, reinforcing the critical role of cache control in achieving high-speed and efficient system performance.

Specifications

Behavioral / Functional

The Cache Controller is designed to manage data transfer between the CPU, local cache memory (SRAM), and main memory (SDRAM). Its primary function is to handle read and write requests from the CPU efficiently by determining whether the requested data is already in cache (a hit) or must be fetched from main memory (a miss). Based on this determination, the controller performs one of several defined behaviors.

Write to Cache (Hit)

When the CPU issues a write request and the corresponding data block is already present in cache, the controller writes the data to the appropriate cache address using the index and offset fields. The dirty and valid bits for that block are set to 1, indicating that the cache data is valid and has been modified relative to main memory.

Read from Cache (Hit)

If the CPU issues a read request and the required data is found in cache, the controller retrieves the data from the SRAM using the index and offset and sends it back to the CPU. This operation provides the fastest data access path, as no main memory interaction is required.

Read/Write on Miss (Dirty Bit = 0)

When the requested block is not found in cache and the dirty bit for the target cache line is 0, the controller initiates a block replacement. It reads the new block (32 bytes) from main memory using the SDRAM controller, stores it in the cache, updates the tag and valid bits, and then completes the requested CPU operation (read or write).

Read/Write on Miss (Dirty Bit = 1)

If the requested block is not in cache and the dirty bit is 1, the controller first writes back the modified cache block to main memory before fetching the new block. The write-back uses the concatenated address [Tag & Index & 00000] as the base address. After writing the old block to main memory, the new block is fetched from SDRAM, stored in cache, the tag is updated, and the CPU's request is completed.

Idle and Ready States

When no transaction is in progress, the controller asserts the RDY signal to indicate readiness for the next operation. During a transaction, RDY is de-asserted until the requested operation completes.

Synchronization

All operations are synchronized with the clock (CLK). The CPU, cache controller, SRAM, and SDRAM controller share this clock to maintain consistent timing and reliable data transfer.

Device design

Symbol Diagram

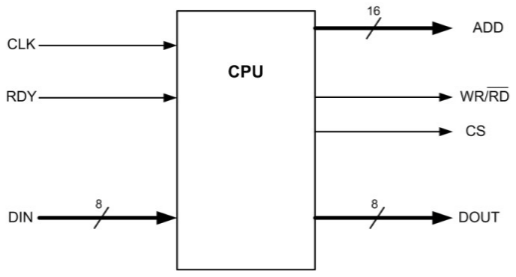


Figure 1: CPU Diagram

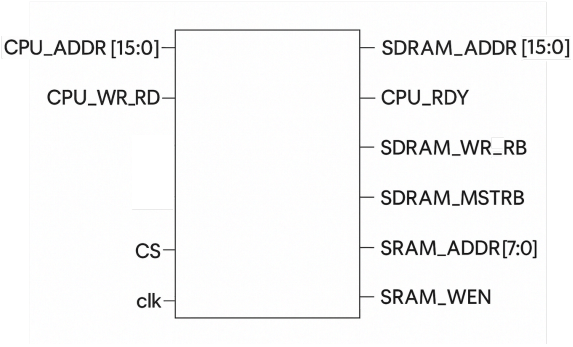


Figure 2: Cache Controller Diagram

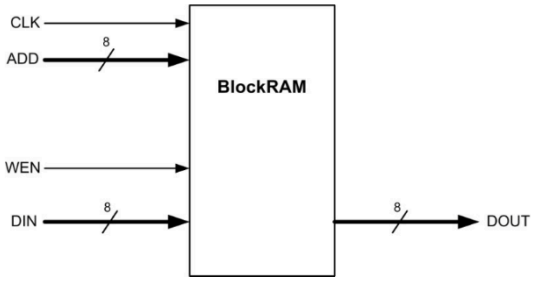


Figure 3: BlockRAM Diagram

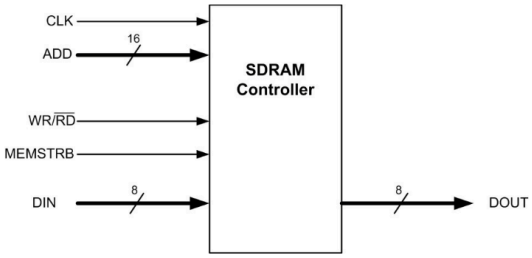


Figure 4: SDRAM Controller Diagram

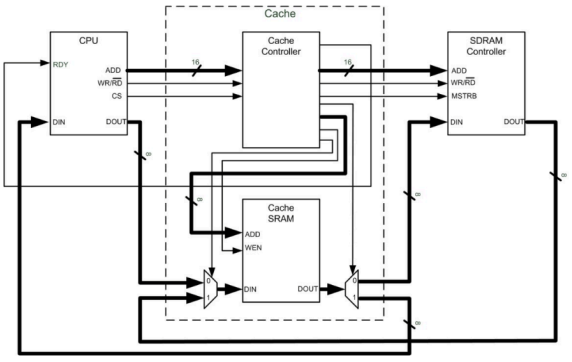


Figure 5: Complete Cache System Diagram

Figure 1 is the diagram for the CPU that communicates with the cache controller through a shared clock (CLK) and control/data signals, chip strobe (CS), read/write indicator (WR/RD), 16-bit address bus (ADD),

8-bit data ports (DIN, DOUT), and a ready input (RDY). During a transaction, the CPU sets the address, selects the operation (WR/RD = 0 for read, 1 for write), and, for writes, places data on DOUT. Once all signals are stable, it asserts CS for four clock cycles to complete the operation. *Figure 9* illustrates an example of a write request.

Figure 2 is the diagram for the cache controller that manages data flow between the CPU, cache (SRAM), and main memory (SDRAM), deciding whether to access data from the cache or fetch it from memory. It connects to the CPU through CPU_ADDR, CPU_WR_RD, CS, and CLK, and interfaces with memory using SDRAM_ADDR, SDRAM_MSTRB, and SRAM_ADDR. On a cache hit, data is served directly from SRAM; on a miss, it's fetched from SDRAM, stored in the cache, and then sent to the CPU.

Figure 3, is the diagram for the cache memory (BlockRAM) stores frequently accessed data and interfaces. It includes an 8-bit address bus (ADD), data ports (DIN, DOUT), a write enable signal (WEN), and a shared clock (CLK) with the cache controller. All operations occur on the rising clock edge, for a read, the selected data appears on DOUT after the next rising edge, for a write, the address, data, and WEN are set, and the data is written on the next rising edge. *Figure 10* illustrates the read and write processes.

Figure 4 is the diagram for the SDRAM controller that manages data transfers between the cache and main memory, handling block read/write requests from the cache controller. Its interface includes ADD, WR/RD, MEMSTRB, DIN, DOUT, and a shared clock (CLK). For a write, the cache sets the address, asserts WR/RD = 1, and pulses MEMSTRB after one cycle, which repeats 32 times for a full block. For a read, WR/RD = 0 and data appears on DOUT. *Figure 11* shows the block read and write operations.

Figure 5 is the overall system architecture where the CPU interfaces with the cache controller to perform read and write operations. The cache controller first checks the local SRAM for the requested data, and if it is not found, the data is fetched from main memory (SDRAM) via the SDRAM controller. This hierarchical design significantly enhances memory access speed. Therefore, the architecture is built with the CPU, the SDRAM controller, and the cache, which acts as the intermediary between them. The cache includes both a control unit and a local SRAM block.

Block Diagram

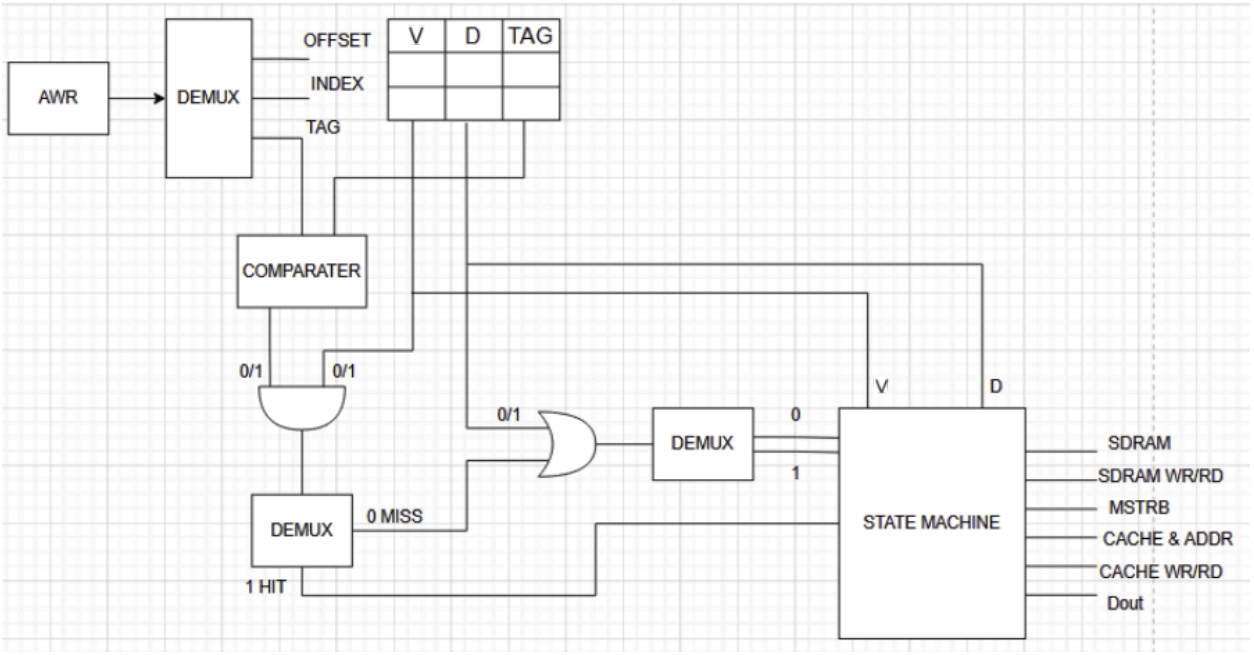


Figure 6: Block Diagram

The cache controller block diagram operation, shown in *Figure 6*, manages how the CPU accesses data between the cache (SRAM) and main memory (SDRAM). The CPU's 16-bit address is divided into Tag, Index, and Offset fields, which are separated by a DEMUX and sent to the Tag Table containing Valid (V) and Dirty (D) bits. A comparator checks if the Tag matches the stored one for the indexed block. If it matches and U = 1, it's a cache hit, and data is accessed directly from the cache. Otherwise, it's a cache miss, and the state machine handles data transfer. On a miss with D = 0, the required block is fetched from SDRAM and written to the cache, while if D = 1, the dirty block is written back before loading the new one. Overall, the controller ensures fast, efficient memory access by prioritizing cache hits and automating miss handling. In short, when Hit data is accessed instantly from cache, when Miss (D = 0) from SDRAM and update cache, when Miss (D = 1) writes back dirty blocks, fetches new blocks, and updates cache.

Process Diagram

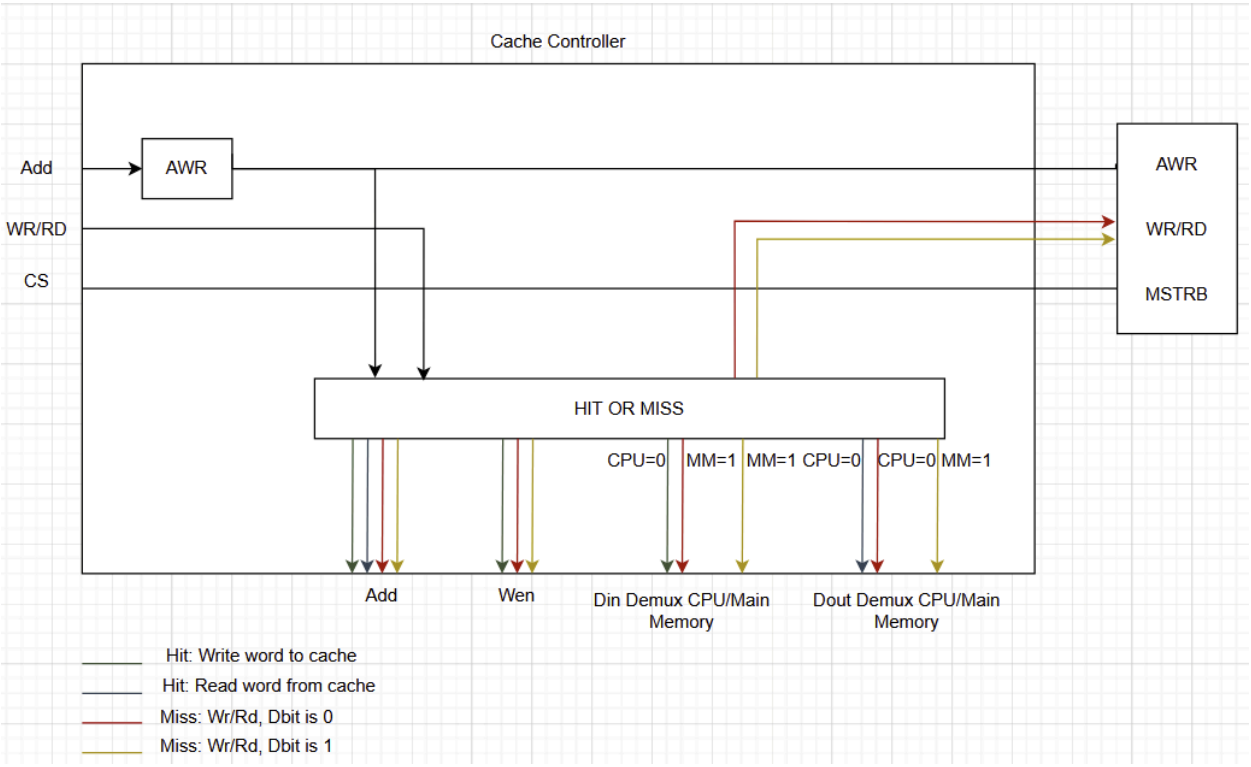


Figure 7: Process Diagram

Figure 7 shows the Cache Controller operation between the CPU and main memory. It determines whether a memory access results in a hit or miss using the address and control signals. On a hit, data is read from or written directly to the cache, while on a miss, the controller accesses main memory, writing back dirty data if necessary. The demultiplexers manage data flow between the CPU, cache, and main memory based on the hit/miss outcome and the dirty bit status.

State Diagram

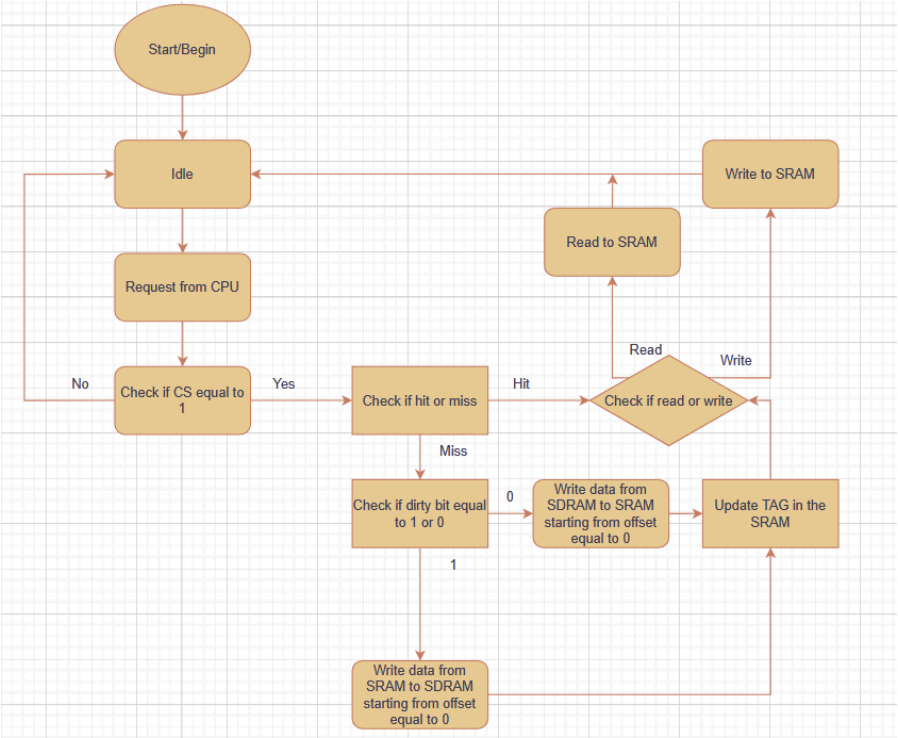


Figure 8: State Diagram

The state diagram shows how the cache controller manages CPU memory requests. The process begins in the idle state, waiting for a CPU request. When the chip select (CS) signal is active, the controller checks for a cache hit or miss. If it's a hit, the controller performs a read from or write to SRAM depending on the operation. If it's a miss, it checks the dirty bit, if set, the modified data in SRAM is written back to SDRAM before fetching the new block from SDRAM into SRAM. After updating the TAG to reflect the new data, the controller returns to the idle state, ready for the next CPU request.

Timing Diagram

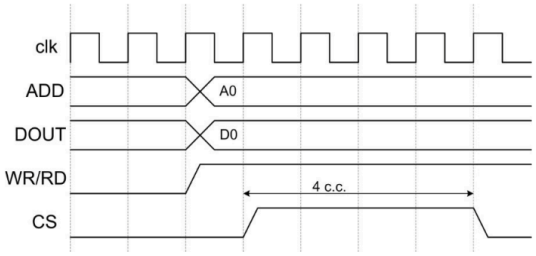


Figure 9: CPU Transaction Example Waveform

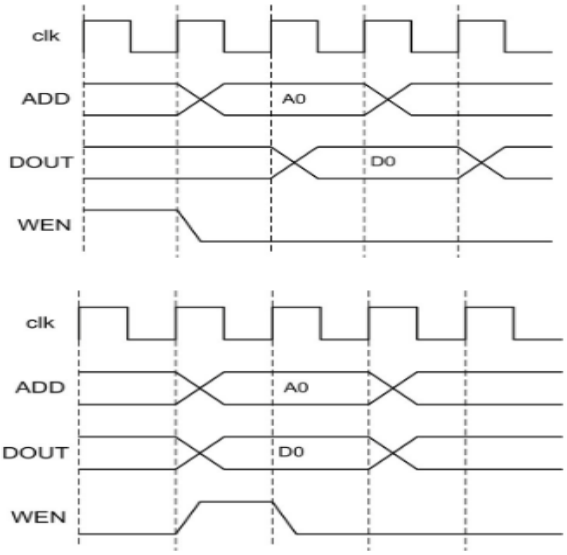


Figure 10: BlockRAM Read/Write Timing Diagram

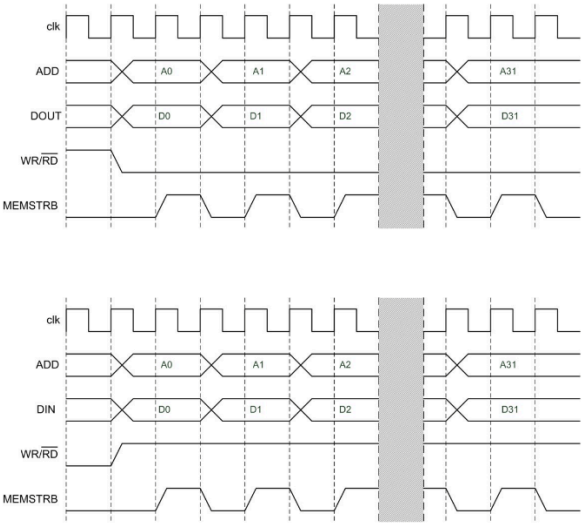


Figure 11: SDRAM Read/Write Timing Diagram

Figure 9 shows CPU transaction waveform shows how the processor communicates with memory or cache during a read or write cycle. When the chip select (CS) signal goes low, the CPU starts a memory transaction lasting about four clock cycles. The address (ADD) and read/write (WR/RD) signals define whether it's a read or write, while DOUT carries or receives the data (D0) during the transaction. This timing ensures proper synchronization between the CPU and memory, allowing stable data transfer within the defined clock cycles.

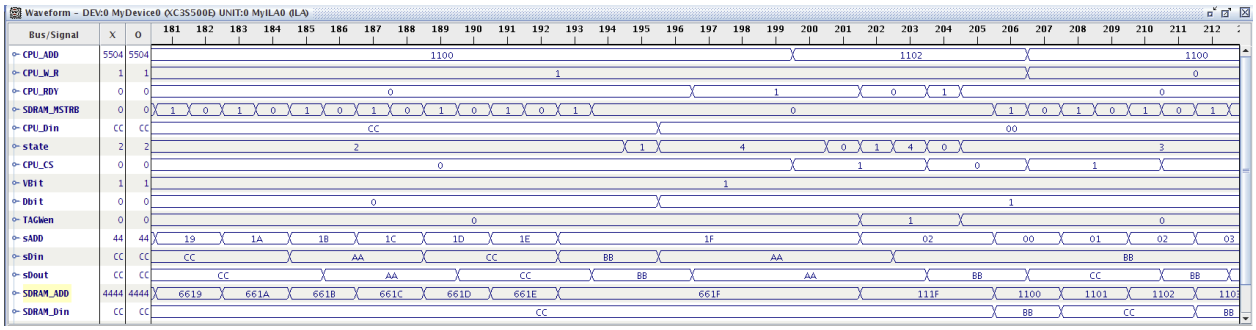
Figure 10 shows the BlockRAM timing diagram shows how read and write operations occur synchronously with the clock. During a write, the address and data are set before the rising clock edge while WEN is high, the data is stored in memory on that edge. During a read, WEN is low and the address

is applied before the rising edge, the corresponding data appears on the output after one clock cycle. This behavior ensures predictable, clock-synchronized data access for the cache controller.

Figure 11, shows the SDRAM timing diagram shows how data is read and written in bursts, synchronized with the clock. During a read operation (WR/RD low), a starting address is given, and after a short delay, consecutive data words (D0-D31) appear on DOUT for each clock cycle while MEMSTRB controls valid data timing. In a write operation (WR/RD high), data words (D0-D31) on DIN are written to consecutive addresses on each rising edge of the clock. This burst-based timing allows SDRAM to transfer large data blocks efficiently once the initial access latency is overcome.

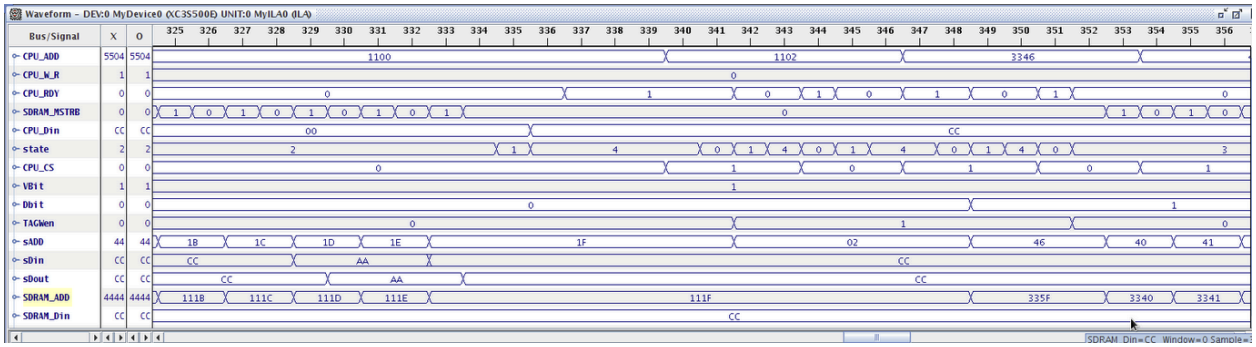
Results

Case 1 Hit: Write Word to Cache



When the CPU issues a write request (CPU_W_R = 1) to an address already present in the cache (a hit), the cache controller detects a valid tag match (Vbit=1, TAGwen=1). The data from CPU_Din (AA, BB, CC, etc.) is written directly to the cache SRAM (sDin changes) without accessing SDRAM (SDRAM_MSTRB stays idle). The state machine (state signal) moves briefly through “write” states, updates the dirty bit (Dbit=1), and completes the write once CPU_RDY asserts.

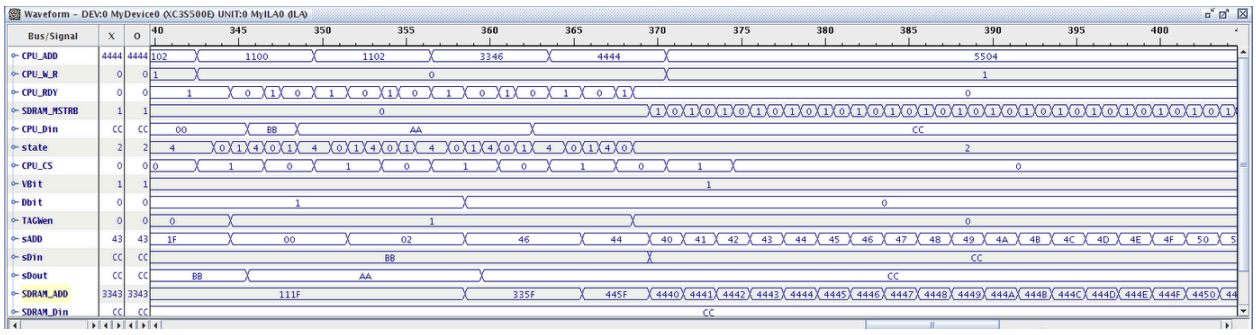
Case 2 Hit: Read Word from Cache



The CPU issues a read (CPU_nR = 1) with an address (CPU_ADD = 5504), and the cache controller checks its tag and valid bit. Since it’s a hit, the Dbit remains 0 (no write-back needed), and the data is served directly from the cache SRAM (CPU_Din = CC). The SDRAM lines (SDRAM_MSTRB,

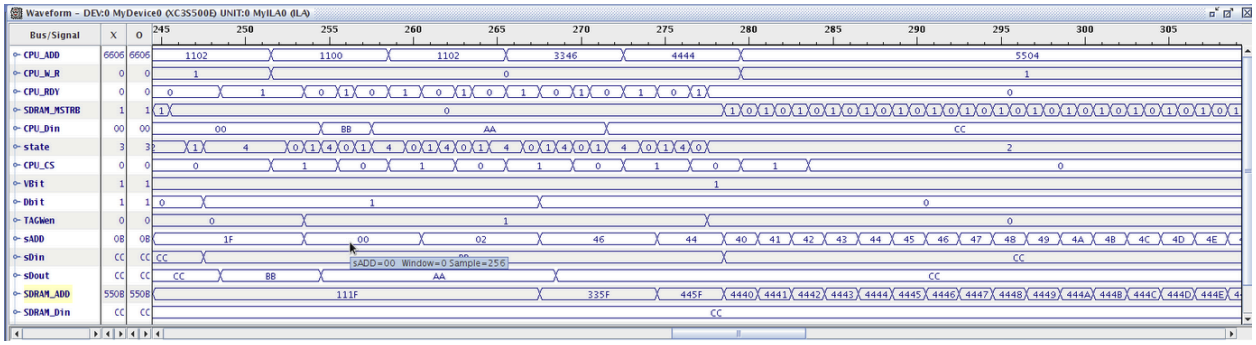
SDRAM_ADD, SDRAM_Din) remain mostly idle, confirming no access to main memory. The CPU receives valid data quickly (CPU_RDY = 1 soon after request), indicating low latency consistent with a hit. Therefore, data is supplied from cache without SDRAM involvement.

Case 3 Miss: Read/Write, Dbit is equal to 0



The CPU issues a read/write to address 4444, but the cache tag doesn't match, causing a miss. Since Dbit = 0, the old block in cache is clean, so it's not written back to SDRAM. The controller asserts SDRAM_MSTRB and starts fetching a new block from main memory (SDRAM_ADD and SDRAM_Din active). Once data arrives from SDRAM, the cache is updated (TAGwen = 1) and the CPU finally receives the data (CPU_RDY = 1). Therefore, data fetched from SDRAM, no write-back needed

Case 4 Miss: Read/Write, Dbit is equal to 1



The CPU requests address 6666, but the cache tag check fails -> cache miss. Since Dbit = 1, the current cache block is dirty and must be written back to SDRAM first (SDRAM_MSTRB and sDout active). After the old block is written to main memory, the controller fetches the new block from SDRAM (SDRAM_ADD, SDRAM_Din active). Once the new block is stored and the tag updated (TAGwen = 1), data is supplied to the CPU (CPU_RDY = 1). Therefore, write-back to SDRAM occurs before fetching the new data.

N	Cache performance parameter	Time in nS
1	Hit / Miss determination time	20ns
2	Data access time	40ns
3	Block replacement time	Dbit=0, 1280ns and Dbit=1, 2560ns
4	Hit time (Case 1 and 2)	60
5	Miss penalty for Case 3 (when D-bit = 0)	1220ns
6	Miss penalty for Case 4 (when D-bit = 1)	2500ns

Conclusion

This project demonstrated how a cache controller improves memory access speed by efficiently managing data transfers between the CPU, cache, and main memory. The controller successfully handled different conditions, such as cache hits when data was already stored locally and cache misses, when data had to be fetched from main memory. Cache hits provided quick access, while misses, especially when involving modified (dirty) data, took longer but were managed correctly through proper block replacement and write-back operations. Therefore, the cache controller reduced CPU wait times and enhanced system performance, showing how effective cache management is essential for improving the efficiency of modern computer systems.

References

[1] COE758 Digital System Engineering, *Project #1 – Memory Hierarchy: Cache Controller*, Toronto Metropolitan University, 2025. [Online]. Available: https://courses.torontomu.ca/d2l/le/content/1072788/viewContent/6603170/View. [Accessed: Oct. 21, 2025].

Appendix

CPU.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CPU is
```

```
    Port (
        clk     : in  STD_LOGIC;
        rst     : in  STD_LOGIC;
        trig    : in  STD_LOGIC;
        Address : out STD_LOGIC_VECTOR (15 downto 0);
        wr_rd   : out STD_LOGIC;
        cs      : out STD_LOGIC;
        DOut    : out STD_LOGIC_VECTOR (7 downto 0)
    );
end CPU;

architecture Behavioral of CPU is
    signal patOut  : std_logic_vector(24 downto 0);
    signal patCtrl : std_logic_vector(2 downto 0) := "111";
    signal updPat  : std_logic;
    signal st1     : std_logic_vector(2 downto 0) := "000";
    signal st1N    : std_logic_vector(2 downto 0);
    signal rReg1, rReg2 : std_logic;
    signal trig_r  : std_logic;
begin

    process(clk, rst, st1N)
    begin
        if(rst = '1')then
            st1 <= "000";
        else
            if(clk'event and clk = '1')then
                st1 <= st1N;
            end if;
        end if;
    end process;

    process(st1, trig_r)
    begin
        if(st1 = "000")then
            if(trig_r = '1')then
                st1N <= "001";
            else
                st1N <= "000";
            end if;
        elsif(st1 = "001")then
            st1N <= "010";
        elsif(st1 = "010")then
            st1N <= "011";
        elsif(st1 = "011")then
            st1N <= "100";
        elsif(st1 = "100")then
            st1N <= "101";
        elsif(st1 = "101")then
            st1N <= "000";
```

```
    else
        st1N <= "000";
    end if;
end process;

process(st1)
begin
    if(st1 = "000")then
        updPat <= '0';
        cs <= '0';
    elsif(st1 = "001")then
        updPat <= '1';
        cs <= '0';
    elsif(st1 = "010")then
        updPat <= '0';
        cs <= '1';
    elsif(st1 = "011")then
        updPat <= '0';
        cs <= '1';
    elsif(st1 = "100")then
        updPat <= '0';
        cs <= '1';
    elsif(st1 = "101")then
        updPat <= '0';
        cs <= '1';
    end if;
end process;

process(clk, rst, updPat, patCtrl)
begin
    if(rst = '1')then
        patCtrl <= "111";
    else
        if(clk'event and clk = '1')then
            if(updPat = '1')then
                patCtrl <= patCtrl + "001";
            else
                patCtrl <= patCtrl;
            end if;
        end if;
    end if;
end process;

process(patCtrl)
begin
    if(patCtrl = "000")then
        patOut <= "00010001000000001010101010";
    elsif(patCtrl = "001")then
        patOut <= "00010001000000101011101111";
    elsif(patCtrl = "010")then
```

```
            patOut <= "000100010000000000000000";
        elsif(patCtrl = "011")then
            patOut <= "000100010000010000000000";
        elsif(patCtrl = "100")then
            patOut <= "001100110100011000000000";
        elsif(patCtrl = "101")then
            patOut <= "010001000100010000000000";
        elsif(patCtrl = "110")then
            patOut <= "010101010000010011001l001";
        else
            patOut <= "011001100000011000000000";
        end if;
    end process;

    process(clk, trig)
    begin
        if(clk'event and clk = '1')then
            rReg1 <= trig;
        end if;
    end process;

    process(clk, rReg1)
    begin
        if(clk'event and clk = '1')then
            rReg2 <= rReg1;
        end if;
    end process;

    trig_r <= rReg1 and (not rReg2);
    Address(15 downto 0) <= patOut(24 downto 9);
    DOut(7 downto 0) <= patOut(8 downto 1);
    wr_rd <= patOut(0);

end Behavioral;
```

SDRAM.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SDRAM is
    Port (
        clk     : in  STD_LOGIC;
        ADD     : in  STD_LOGIC_VECTOR (15 downto 0);
```

```
      WrRd    : in  STD_LOGIC;
      MEMSTRB : in  STD_LOGIC;
      DIN     : in  STD_LOGIC_VECTOR (7 downto 0);
      DOUT    : out STD_LOGIC_VECTOR (7 downto 0)
   );
end SDRAM;

architecture Behavioral of SDRAM is
   type ramemory is array (7 downto 0, 31 downto 0) of std_logic_vector(7 downto 0);
   signal RAM_SIG : ramemory;
   signal counter : integer := 0;
begin
   process (clk)
   begin
      if CLK'event and CLK = '1' then
         if counter = 0 then
            for I in 0 to 7 loop
               for J in 0 to 31 loop
                  RAM_SIG(I, J) <= "11110000";
               end loop;
            end loop;
            counter <= 1;
         end if;

         if MEMSTRB = '1' then
            if WrRd = '1' then
               RAM_SIG(
                  to_integer(unsigned(ADD(7 downto 5))),
                  to_integer(unsigned(ADD(4 downto 0)))
               ) <= DIN;
            else
               DOUT <= RAM_SIG(
                  to_integer(unsigned(ADD(7 downto 5))),
                  to_integer(unsigned(ADD(4 downto 0)))
               );
            end if;
         end if;
      end if;
   end process;
end Behavioral;
```

SRAM.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;

entity SRAM is
    Port (
        clk       : in  STD_LOGIC;
        ADDR      : out STD_LOGIC_VECTOR(15 downto 0);
        DOUT      : out STD_LOGIC_VECTOR(7 downto 0);
        sAddra    : out STD_LOGIC_VECTOR(7 downto 0);
        sDina     : out STD_LOGIC_VECTOR(7 downto 0);
        sDouta    : out STD_LOGIC_VECTOR(7 downto 0);
        sD_Addra  : out STD_LOGIC_VECTOR(15 downto 0);
        sD_Dina   : out STD_LOGIC_VECTOR(7 downto 0);
        sD_Douta  : out STD_LOGIC_VECTOR(7 downto 0);
        cacheAddr : out STD_LOGIC_VECTOR(7 downto 0);
        WR_RD, MEMSTRB, RDY, CS : out STD_LOGIC
    );
end SRAM;

architecture Behavioral of SRAM is
    signal CPU_ADD       : STD_LOGIC_VECTOR (15 downto 0);
    signal CPU_W_R, CPU_CS, CPU_RDY : STD_LOGIC;
    signal cpu_tag       : STD_LOGIC_VECTOR(7 downto 0);
    signal index         : STD_LOGIC_VECTOR(2 downto 0);
    signal offset        : STD_LOGIC_VECTOR(4 downto 0);
    signal Tag_index     : STD_LOGIC_VECTOR(10 downto 0);
    signal CPU_Dout, CPU_Din : STD_LOGIC_VECTOR(7 downto 0);

    signal Dbit          : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
    signal Vbit          : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
    signal sADD, sDin, sDout : STD_LOGIC_VECTOR(7 downto 0);
    signal sWen          : STD_LOGIC_VECTOR(0 DOWNTO 0);
    signal TAGWen        : STD_LOGIC := '0';

    signal SDRAM_Din, SDRAM_Dout : STD_LOGIC_VECTOR(7 downto 0);
    signal SDRAM_ADD     : STD_LOGIC_VECTOR(15 downto 0);
    signal SDRAM_MSTRB, SDRAM_W_R : STD_LOGIC;
    signal counter       : integer := 0;
    signal sdoffset      : integer := 0;

    type cachememory is array (7 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
    signal memtag        : cachememory := ((others => (others => '0')));

    signal control0      : STD_LOGIC_VECTOR(35 downto 0);
    signal ila_data      : std_logic_vector(99 downto 0);
    signal trig0         : std_logic_vector(0 TO 0);

    TYPE state_value IS (state0, state1, state2, state3, state4);
    signal state_current  : state_value;
    signal state         : STD_LOGIC_VECTOR(3 downto 0);
```

```
   COMPONENT SDRAM
      Port (
         clk     : in  STD_LOGIC;
         ADD     : in  STD_LOGIC_VECTOR (15 downto 0);
         WrRd    : in  STD_LOGIC;
         MEMSTRB : in  STD_LOGIC;
         DIN     : in  STD_LOGIC_VECTOR (7 downto 0);
         DOUT    : out STD_LOGIC_VECTOR (7 downto 0)
      );
   END COMPONENT;

   COMPONENT SRAM
      PORT (
         clka  : IN STD_LOGIC;
         wea   : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
         addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         dina  : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
      );
   END COMPONENT;

   COMPONENT CPU
      Port (
         clk     : in  STD_LOGIC;
         rst     : in  STD_LOGIC;
         trig    : in  STD_LOGIC;
         Address : out STD_LOGIC_VECTOR (15 downto 0);
         wr_rd   : out STD_LOGIC;
         cs      : out STD_LOGIC;
         Dout    : out STD_LOGIC_VECTOR (7 downto 0)
      );
   END COMPONENT;

   COMPONENT icon
      PORT (
         CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0)
      );
   END COMPONENT;

   COMPONENT ila
      PORT (
         CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
         CLK     : IN STD_LOGIC;
         DATA    : IN STD_LOGIC_VECTOR(99 DOWNTO 0);
         TRIG0   : IN STD_LOGIC_VECTOR(0 TO 0)
      );
   END COMPONENT;

BEGIN
   myCPU : CPU Port Map (clk, '0', CPU_RDY, CPU_ADD, CPU_W_R, CPU_CS,
```

```
CPU_Dout);
   SDRAM     : SDRAM Port Map (clk, SDRAM_ADD, SDRAM_W_R, SDRAM_MSTRB,
SDRAM_Din, SDRAM_Dout);
   mySRAM    : SRAM Port Map (clk, sWen, sADD, sDin, sDout);
   myIcon    : icon Port Map (CONTROL0);
   myILA     : ila  Port Map (CONTROL0, CLK, ila_data, TRIG0);

   process(clk, CPU_CS)
   begin
      if (clk'event AND clk = '1') then
         if (state_current = state0) then
            CPU_RDY <= '0';
            cpu_tag <= CPU_ADD(15 downto 8);
            index <= CPU_ADD(7 downto 5);
            offset <= CPU_ADD(4 downto 0);
            SDRAM_ADD(15 downto 5) <= CPU_ADD(15 downto 5);
            sADD(7 downto 0) <= CPU_ADD(7 downto 0);
            sWen <= "0";

            if(Vbit(to_integer(unsigned(index))) = '1'
               AND memtag(to_integer(unsigned(index))) = cpu_tag) then
               TAGWen <= '1';
               state_current <= state1;
               state <= "0001";
            else
               TAGWen <= '0';
               if (Dbit(to_integer(unsigned(index))) = '1'
                  AND Vbit(to_integer(unsigned(index))) = '1') then
                  state_current <= state3;
                  state <= "0011";
               else
                  state_current <= state2;
                  state <= "0010";
               end if;
            end if;

         elsif(state_current = state1) then
            if (CPU_W_R = '1') then
               sWen <= "1";
               Dbit(to_integer(unsigned(index))) <= '1';
               Vbit(to_integer(unsigned(index))) <= '1';
               sDin <= CPU_Dout;
               CPU_Din <= "00000000";
            else
               CPU_Din <= sDout;
            end if;
            state_current <= state4;
            state <= "0100";

         elsif(state_current = state2) then
```

```
            if (counter = 64) then
               counter <= 0;
               Vbit(to_integer(unsigned(index))) <= '1';
               memtag(to_integer(unsigned(index))) <= cpu_tag;
               sdoffset <= 0;
               state_current <= state1;
               state <= "0001";
            else
               if (counter mod 2 = 1) then
                  SDRAM_MSTRB <= '0';
               else
                  SDRAM_ADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(sdoffset,
offset'length));
                  SDRAM_W_R <= '0';
                  SDRAM_MSTRB <= '1';
                  sADD(7 downto 5) <= index;
                  sADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(sdoffset,
offset'length));
                  sDin <= SDRAM_Dout;
                  sWen <= "1";
                  sdoffset <= sdoffset + 1;
               end if;
               counter <= counter + 1;
            end if;

         elsif(state_current = state3) then
            if (counter = 64) then
               counter <= 0;
               Dbit(to_integer(unsigned(index))) <= '0';
               sdoffset <= 0;
               state_current <= state2;
               state <= "0010";
            else
               if (counter mod 2 = 1) then
                  SDRAM_MSTRB <= '0';
               else
                  SDRAM_ADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(sdoffset,
offset'length));
                  SDRAM_W_R <= '1';
                  sADD(7 downto 5) <= index;
                  sADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(sdoffset,
offset'length));
                  sWen <= "0";
                  SDRAM_Din <= sDout;
                  SDRAM_MSTRB <= '1';
                  sdoffset <= sdoffset + 1;
               end if;
               counter <= counter + 1;
            end if;
```

```
        elsif(state_current = state4) then
            CPU_RDY <= '1';
            if (CPU_CS = '1') then
                state_current <= state0;
                state <= "0000";
            end if;
        end if;
    end if;
end process;

MEMSTRB <= SDRAM_MSTRB;
ADDR <= CPU_ADD;
WR_RD <= CPU_W_R;
DOUT <= CPU_Din;
RDY <= CPU_RDY;
CS <= CPU_CS;

sAddra <= sADD;
sDina <= sDin;
sDouta <= sDout;

sD_Addra <= SDRAM_ADD;
sD_Dina <= SDRAM_Din;
sD_Douta <= SDRAM_Dout;

cacheAddr <= CPU_ADD(15 downto 8);

ila_data(15 downto 0) <= CPU_ADD;
ila_data(16) <= CPU_W_R;
ila_data(17) <= CPU_RDY;
ila_data(18) <= SDRAM_MSTRB;
ila_data(26 downto 19) <= CPU_Din;
ila_data(30 downto 27) <= state;
ila_data(31) <= CPU_CS;
ila_data(32) <= Vbit(to_integer(unsigned(index)));
ila_data(33) <= Dbit(to_integer(unsigned(index)));
ila_data(34) <= TAGWen;
ila_data(42 downto 35) <= sADD;
ila_data(50 downto 43) <= sDin;
ila_data(58 downto 51) <= sDout;
ila_data(74 downto 59) <= SDRAM_ADD;
ila_data(82 downto 75) <= SDRAM_Din;
ila_data(90 downto 83) <= SDRAM_Dout;
ila_data(98 downto 91) <= CPU_ADD(15 downto 8);

end Behavioral;
```

Constraints.ucf

```
NET  "clk" LOC  = "C9";
```