Ryerson University

Department of Electrical & Computer Engineering
Faculty of Engineering & Architectural Science

Course Title:	Digital Systems Engineering
Course Number:	COE758
Semester/Year (e.g. F2017)	F2024

Instructor	Karim Soubra

Assignment/Lab Number:	Project 1
Assignment/Lab Title:	Cache Controller

Submission Date:	Wednesday, October 16, 2024
Due Date:	Wednesday, October 16, 2024

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Kelsey	Charles		06	C.K.

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Contents

Abstract

Objective

This project has several objectives. The first objective was to learn the functionality of a cache controller and its interaction with block-memory (SRAM based) and SDRAM-controllers. The second objective was to get practical experience in the design and implementation of custom logic controllers and interfacing to SRAM memory units and other logic devices. The final objective was to earn VHDL-coding technique within the Xilinx ISE CAD environment of the system and a hardware evaluation platform based on the Xilinx Spartan-3E FPGA.

Approach

The Cache Controller was implemented in VHDL. The VHDL implementation for the CPU was already given, while everything else was designed from scratch. Prior to VHDL implementation, the schematic diagram, block diagram, and a conceptual finite state machine were drawn by hand. This resulted in the creation of three reference diagrams to help understand the nature of the project. After that, ip cores were created and VHDL implementation of the Cache Controller and the SDRAM were created. Several debugging variables and testbenches were created within each file to test functionality and correctness. Chipscope was used to demo while testbenches were used to simulate and analyze timing diagrams for this lab.

Results

The results of this design project are summarized in Table 1. Most of the timing parameters were measured using the testbenches of the Xilinx software. The only timing parameters calculated were the Miss Penalties for different values of D-bits.

Introduction

Motivation

Memory hierarchies in digital systems are essential for balancing the fast processing speed of CPUs with the slower access times of main memory. In modern computer architecture, memory hierarchy is a key factor driving improvements in system performance and energy efficiency. As technology advances, the need for efficient memory management and faster data communication becomes even more critical. It's important for students to understand how memory hierarchies help reduce delays associated with slower, high-capacity memory, while

also considering the trade-offs between memory capacity, access speed, and their proximity to the processor.

Objective

The main objectives of this lab were to:

- To learn the functionality of a cache controller and its interaction with block-memory (SRAM based) and SDRAM-controllers.
- To get practical experience in the design and implementation of custom logic controllers and interfacing to SRAM memory units and other logic devices.
- To learn VHDL-coding technique within the Xilinx ISE CAD environment of the system and a hardware evaluation platform based on the Xilinx Spartan-3E FPGA.

These objectives establish a clear framework for the experiment, directing the exploration of cache controller operations, memory interactions, custom logic design, and VHDL coding, while also highlighting hands-on experience with the Xilinx Spartan-3E FPGA platform.

Theory

Computer systems are built with a memory hierarchy to optimize memory management. At the top of this hierarchy is the Cache Memory, located closest to the CPU. Cache Memory holds small amounts of recently accessed data, organized into blocks, with each block containing multiple words. A Cache Controller manages the blocks in Cache by storing key identifiers such as the index, tag, valid, and dirty bits, all associated with each block of data. The goal of this project was to design and implement a Cache Controller for a hypothetical computer system with a total capacity of 256 bytes.

The different behavioral cases involved with the CPU and Cache Controller are as follows:

1. Write a word to cache [hit]
   When the cache receives a write request from the CPU, and the requested data is found in the cache (a cache hit), the index and offset parts of the CPU-supplied address are used as the write address for the new data in the local SRAM. The dirty and valid bits associated with the target block are set to 1, and the updated data is written to the local SRAM.

2. Read a word from cache [hit]
   When the cache receives a read request from the CPU, and the requested data is found in the cache (a cache hit), the index and offset parts of the CPU-supplied address are sent to the local SRAM. The corresponding data is then retrieved and routed back to the CPU.

3. Read/Write from/to cache [miss] and dirty bit = 0
   When a read or write request is received and the corresponding block is not found in the cache (a cache miss), a block replacement is necessary. First, the dirty bit of the relevant block must be checked. If the dirty bit is 0, the full CPU address with its offset portion set to "00000" is sent to the SDRAM memory controller as the base address for the block to be fetched. The entire block (32 bytes) is then read from SDRAM and written to the local Cache SRAM. The tag from the CPU address replaces the value in the corresponding tag register, and the valid bit is set to 1. Afterward, the requested read or write operation is carried out following the same steps as in a cache hit.

4. Read/Write from/to cache [miss] and dirty bit = 1
   When a read or write request results in a cache miss, triggering a block replacement, the dirty bit of the corresponding block must be checked. If the dirty bit is 1, the recently used block (which is about to be replaced) must be written back to main memory. The block in the local SRAM is written to main memory via the SDRAM memory controller, using the base address: [Tag & Index & 00000].

   Next, the new block (corresponding to the address requested by the CPU) must be loaded into the cache. This involves reading the block from the SDRAM memory controller by issuing the address: [Tag & Index & 00000], where the Tag is the one from the CPU's current request. The entire block fetched from the SDRAM is then written to the local cache SRAM. The tag associated with this block in the cache must be updated with the new tag from the CPU's request. Once this is done, the original CPU transaction (read or write) can be completed using the procedure for cache hits.

As stated above, the cache controller interacts with the CPU in several behavioral cases. The CPU itself consists of a strobe CS, a read/write indicator WR/RD, a 16-bit address ADD, 8-bit data input and output ports DIN and DOUT, and a ready indicator input RDY.

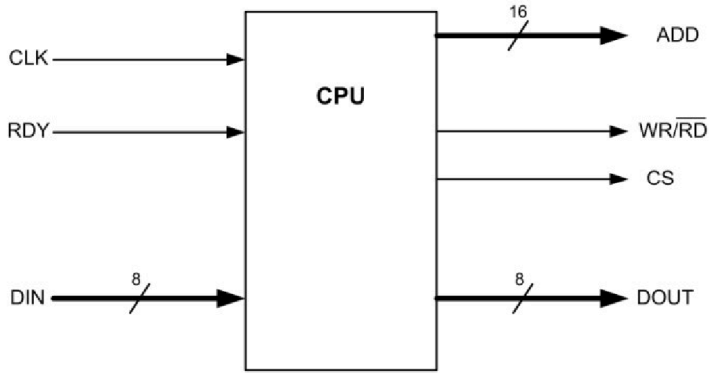

Figure 1: CPU Interface

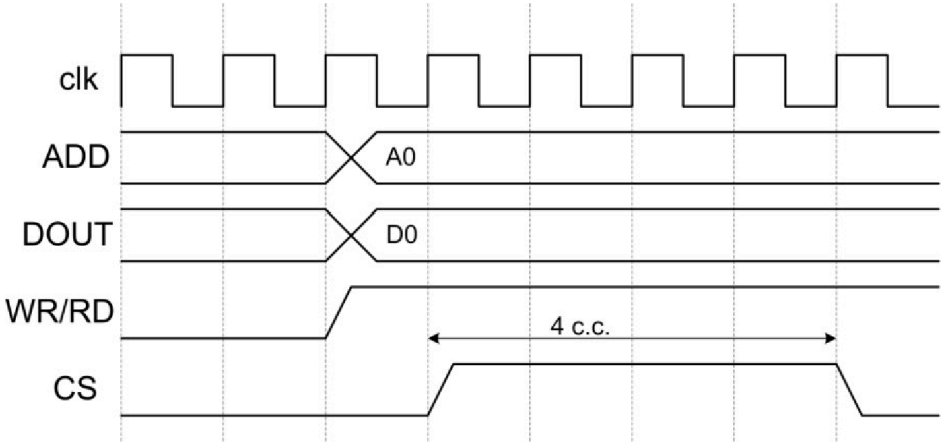

Figure 2: CPU Transaction Example

As mentioned previously, the Cache Controller, shown in Figure 3, is situated between the CPU and the SDRAM Controller. The SDRAM Controller is synchronized with the Cache Controller and responds to block read and write requests from it. It consists of a 16-bit address ADD, a read/write indicator WR/RD, a strobe MEMSTRB, and data input and output ports DIN and DOUT.

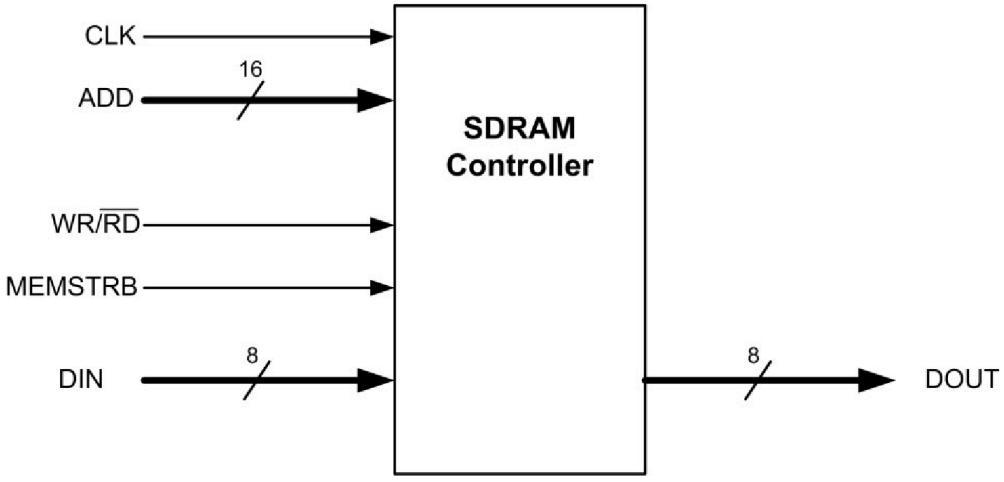

Figure 3: SDRAM Controller Interface

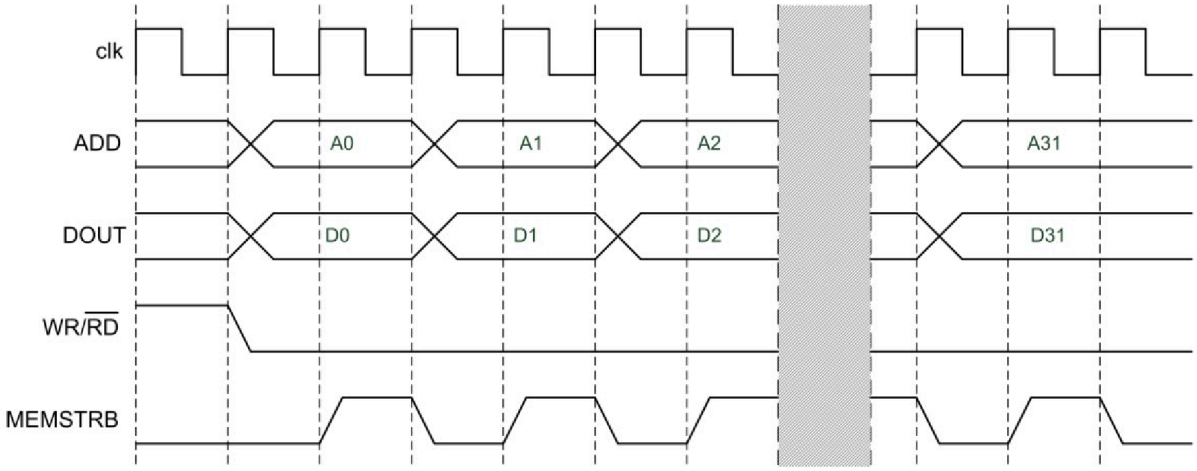

Figure 4: SDRAM Block Read

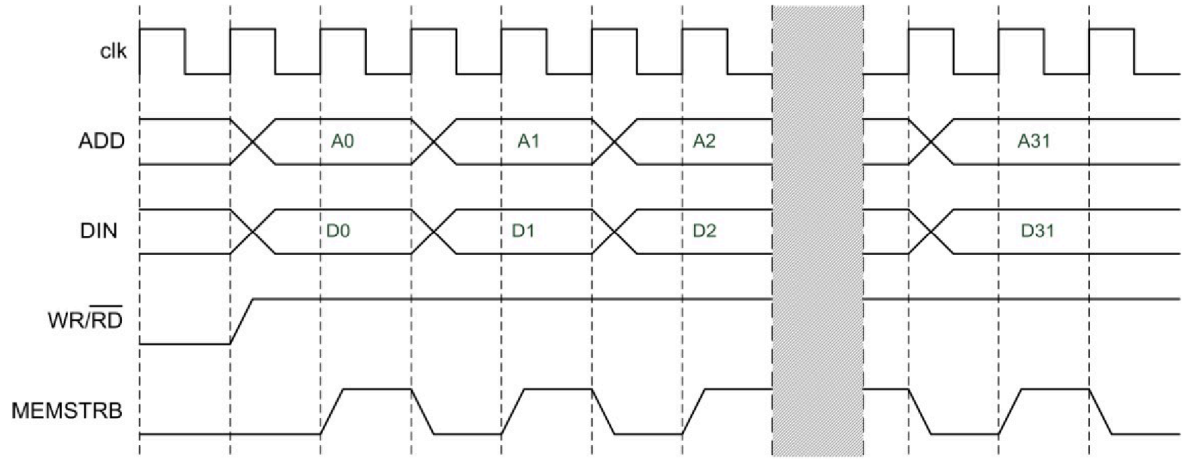

Figure 5: SDRAM Block Write

In Figures 4 and 5, an example of an SDRAM Block Read and Block Write operation is shown. Graphically, the difference between the two operations is that for Block Reads, WR/RD is held low, with no data being needed in the DIN port, whereas for Block Writes, the WR/RD signal is held high with the MEMSTRB being asserted for one clock cycle, while also repeating the process 32 times to write a full block to memory.

Finally, the BlockRAM, as shown below in Figure 6. is used to implement the local memory for the cache. It is synchronized with the Cache Controller, sharing the same clock, and consists of an 8-bit address ADD, 8-bit data input and output DIN and DOUT, and a write enable signal WEN.

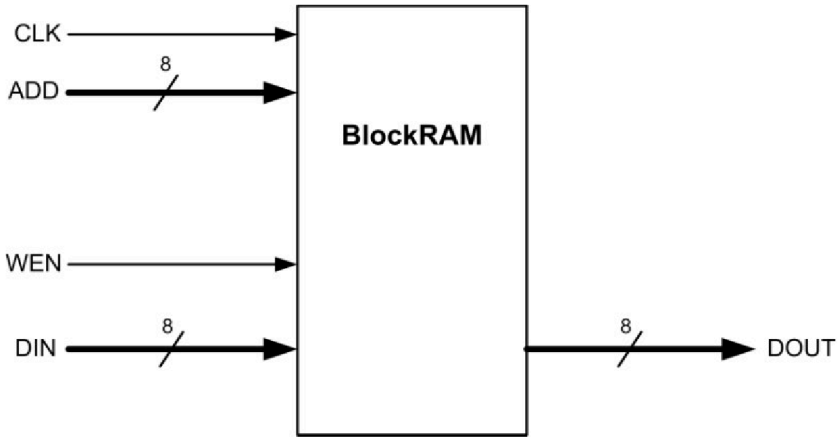

Figure 6: BlockRAM Interface

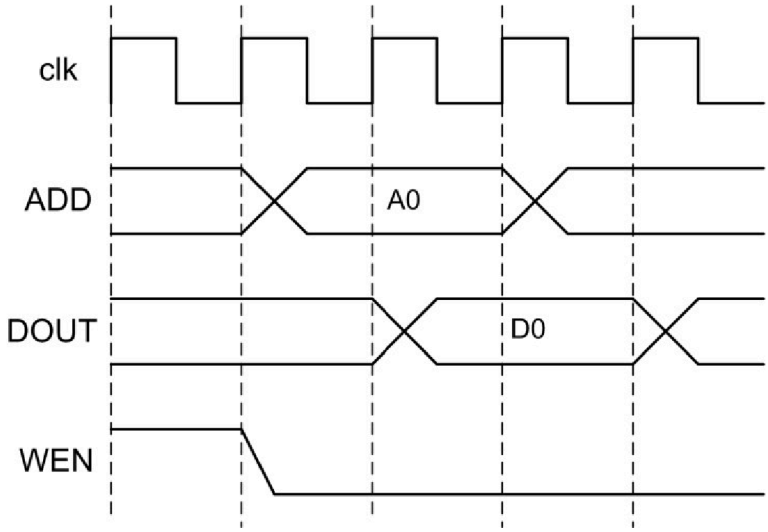

Figure 7: BlockRAM Read Example

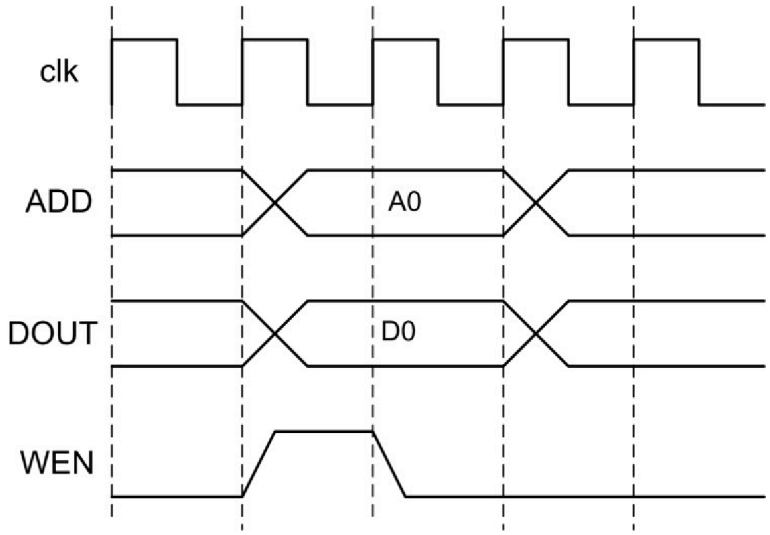

Figure 8: BlockRAM Write Example

In Figure 7 and Figure 8, BlockRAM Read and Write operations are shown. Graphically, the Read operations are done by first setting the appropriate address on the address bus, then propagating the addressed data to the output DOUT after the next rising edge of the clock, whereas Write operations are performed by setting a specific address and data on the appropriate ports, and then asserting the write enable signal WEN, with the data being written to the specified address on the next rising edge.

Device Design (Description)

Symbol Diagram

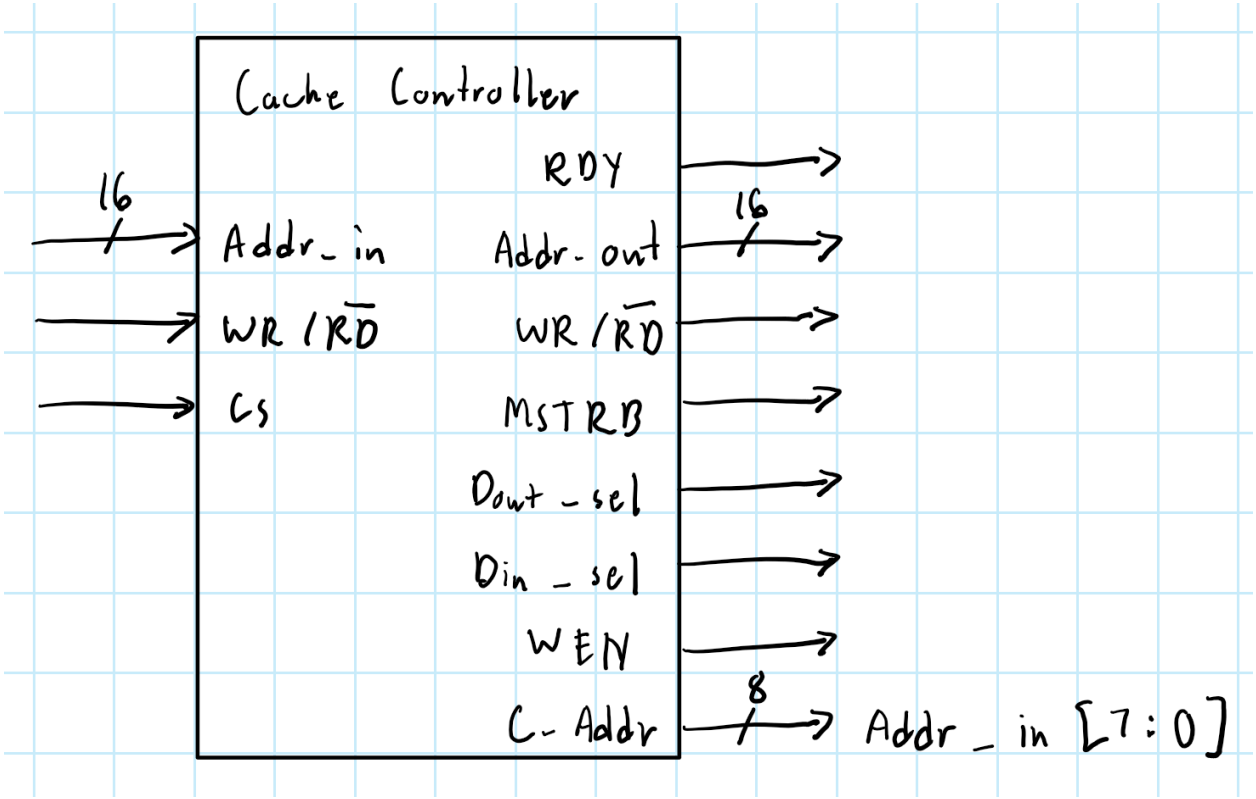

Figure 9: Cache Controller Symbol

In Figure 9, shown above is the symbol for the Cache Controller. The inputs consist of a 16-bit address input, a WR/RD input, and a strobe CS. The output ports to the SDRAM controller are: a 16-bit Address Output, a WR/RD output, and a Memory Strobe (MSTRB). Some outputs are headed towards the Cache SRAM which are: the 8-bit C_Addr and the Write Enable indicator. The other output includes a Dout_sel towards a Demultiplexer.

Block Diagram

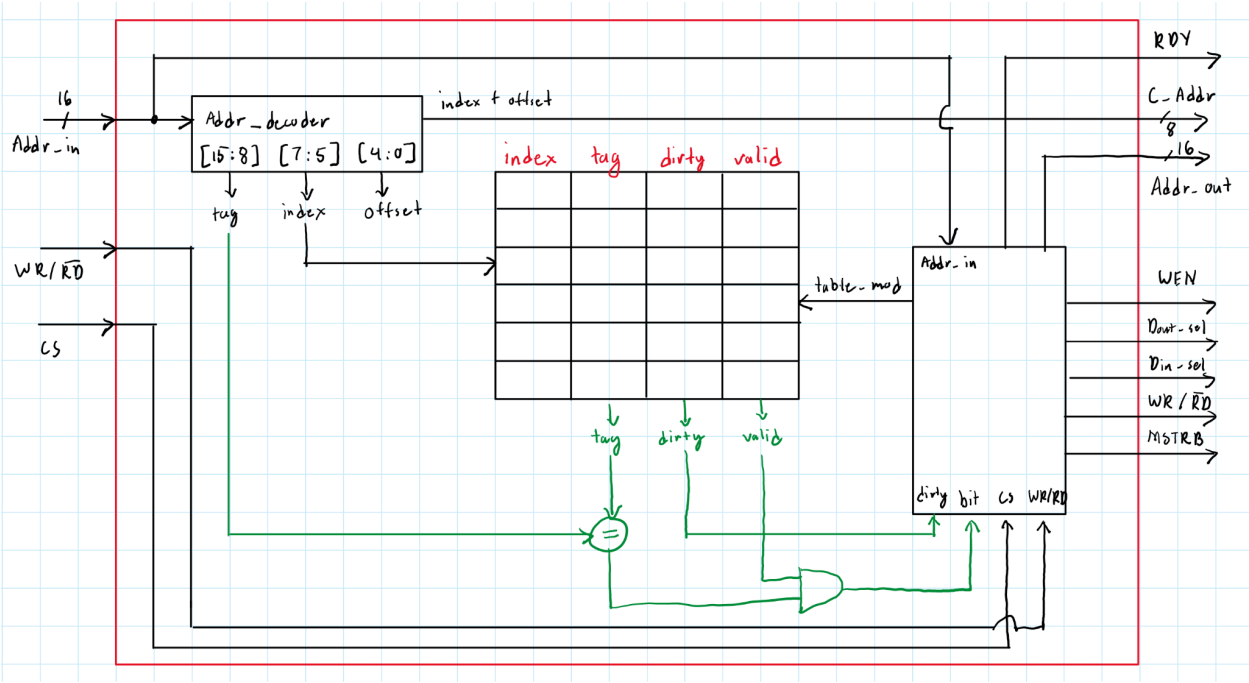

Figure 10: Block Diagram of Cache Controller

The block diagram of the Cache Controller (shown in Figure 10) illustrates the component's internal behavior. The 16-bit input address is connected to both the Address Decoder and the FSM (Finite State Machine) control. The Address Decoder processes the input address by dividing it into three fields: the tag, index, and offset. The tag field acts as a unique identifier for a group of data, while the index is used to locate the cache line in memory that may contain the requested data. The offset specifies the exact location within the cache line.

The index is sent to a table that contains columns for "dirty" and "valid" bits. The valid bit indicates whether a cache entry contains valid data, while the dirty bit shows whether the cache entry has been modified since it was loaded into memory. The valid bit is then fed into a logical AND gate, along with the result of the comparison between the address's tag and the stored tag. The result of this AND operation is sent to the FSM control's "bit" input port.

The FSM control then generates the following signals: RDY, Addr_out, WEN, Din_sel, Dout_sel, WR/RDo, and MSTRB, which are used to manage the interactions between the CPU, cache, and memory.

State Diagram

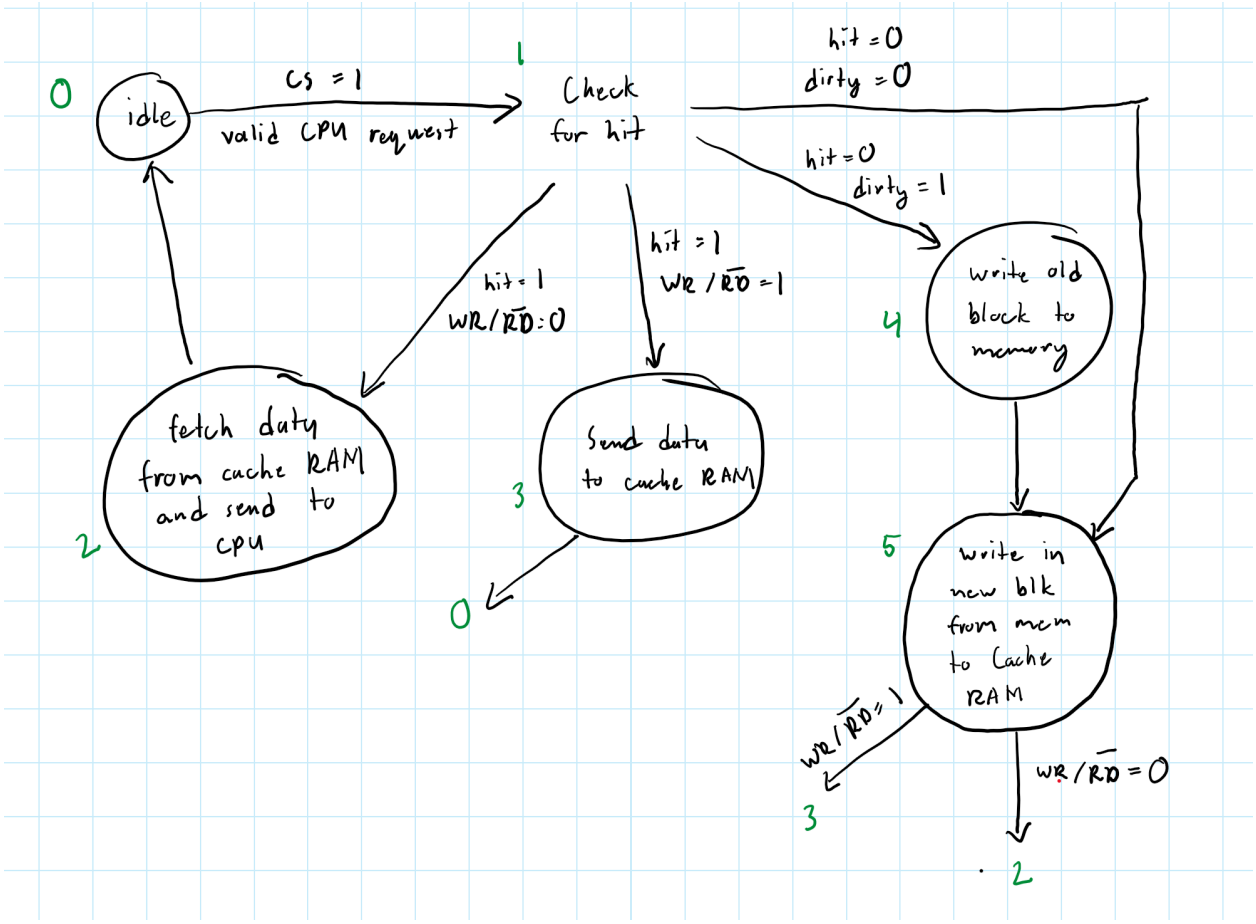

Figure 11: Finite State Machine (FSM) for Cache Controller

The Finite State Machine (FSM) shown in Figure 11 illustrates the various states and transitions involved in the caching process. In this diagram, there are six states, labeled from 0 through 5, each with corresponding conditions for each transition. The FSM largely corresponds to the Behavioral Cases discussed in the Theory section of this lab report.

Timing Diagram

Example 1: Tag (0001 0001), Index(000), Offset (00000), and R/W (W)

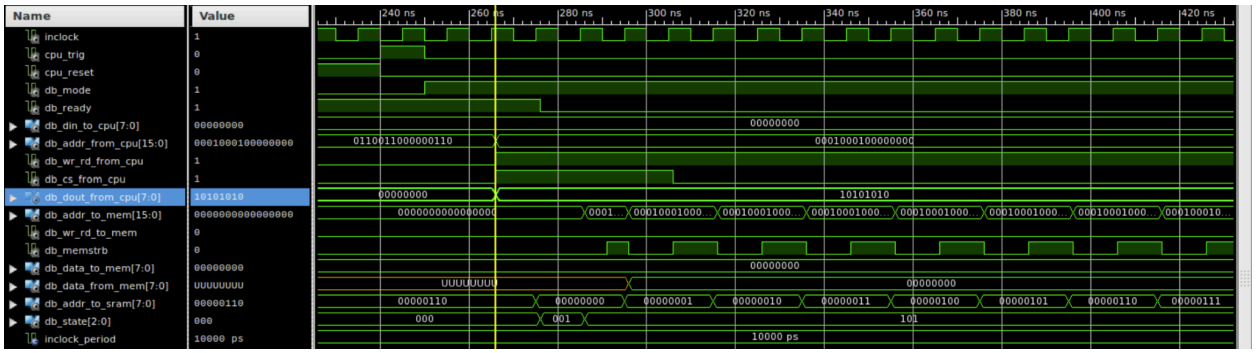

Figure 12: Functional Simulation for the process: State 0 to State 1, State 1 to State 5

The cache begins in state 0 (idle) from which it goes into state 1 (hit/miss determination). Since the cache is fresh, it's a miss (dirty bit = 0), and so the cache goes into state 5. This triggers 32 memory transfers from memory into the cache SRAM. Once this process is done, the simulation diagram continues as:

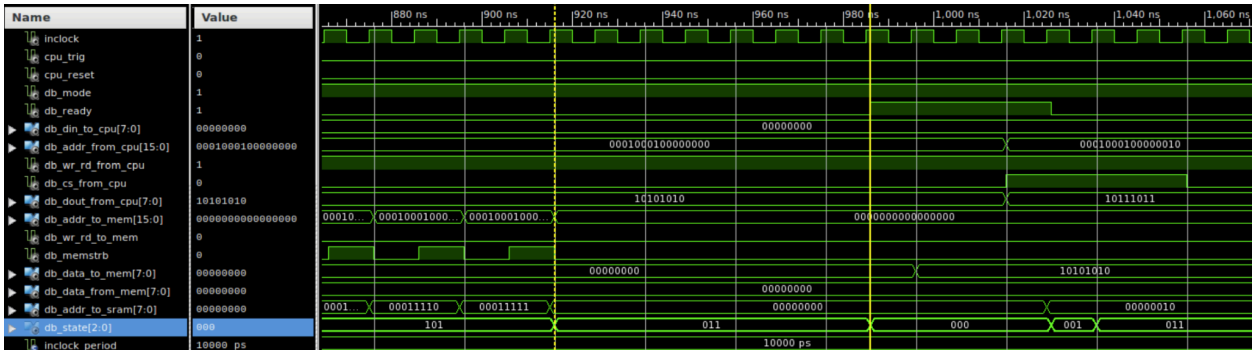

Figure 13: Functional Simulation for State 5 to State 3, then State 3 to State 0

Once the memory block transfer is complete, the write needs to be serviced, which is done in state 2. Finally, the cache returns to state 0.

Example 2: Tag (0001 0001), Index(000), Offset (00010), and R/W (W)

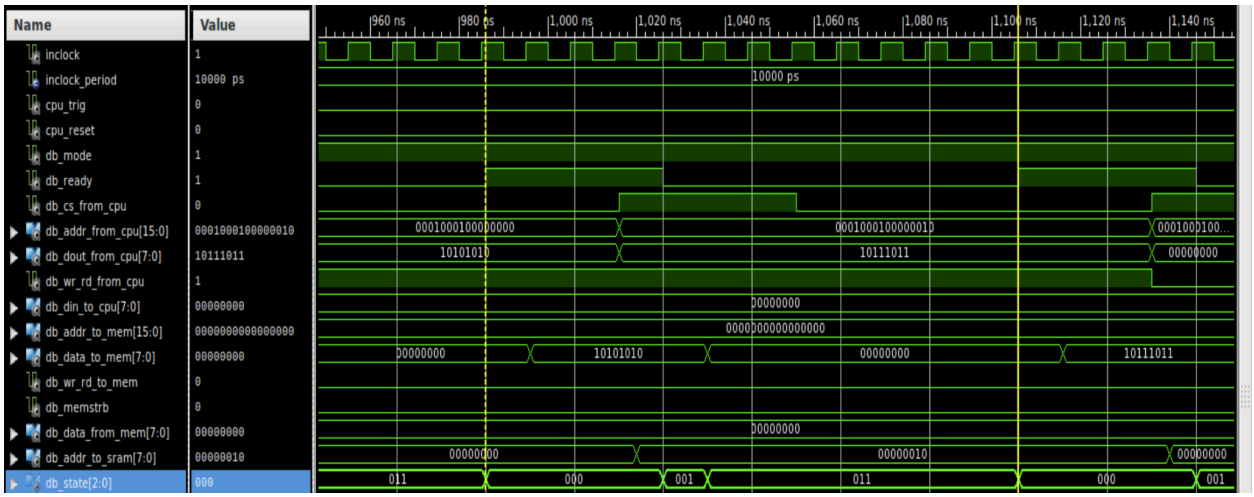

Figure 14: Functional Simulation for State 1 to State 3.

The cache begins in state 0 (idle) from which it goes into state 1 (hit/miss determination). Due to the results of example 1, this is a hit, and so the cache enters state 3 (write to cache).

Example 3: Tag (0101 0101), Index(000), Offset (00100), and R/W (W)

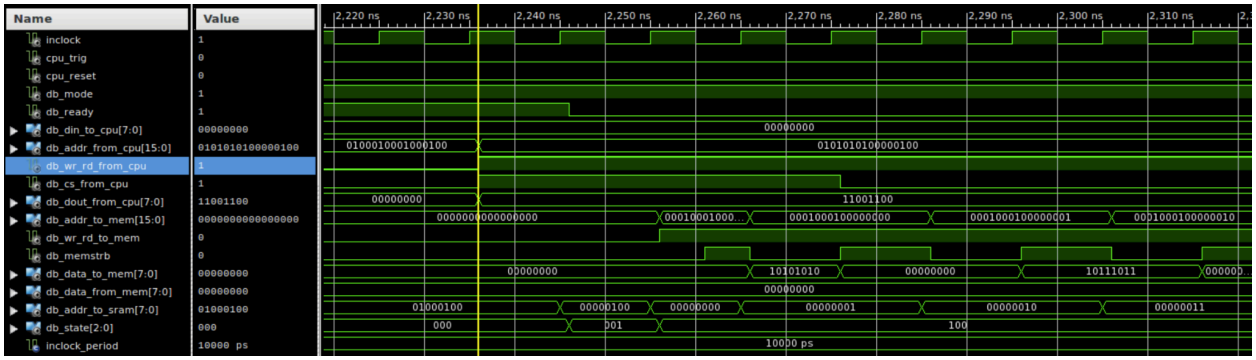

Figure 15: Functional Simulation for State 0 to State 1, State 1 to State 4, State 4 to State 5

The cache begins in state 0 (idle) from which it goes into state 1 (hit/miss determination). Due to the results of example 1 & 2, this is a miss with dirty bit =1, and so the cache enters state 4 (transfer block to memory). This triggers 32 transfers to occur, starting at the base address of the old block. Once this is complete the cache enters state 5 (transfer block from memory) which again triggers 32 transfers, starting at the base address of the new block. The tail end of these 64 transfers, as well as the caches service of the write (state 3), and return to idle (state 0) can be seen below:

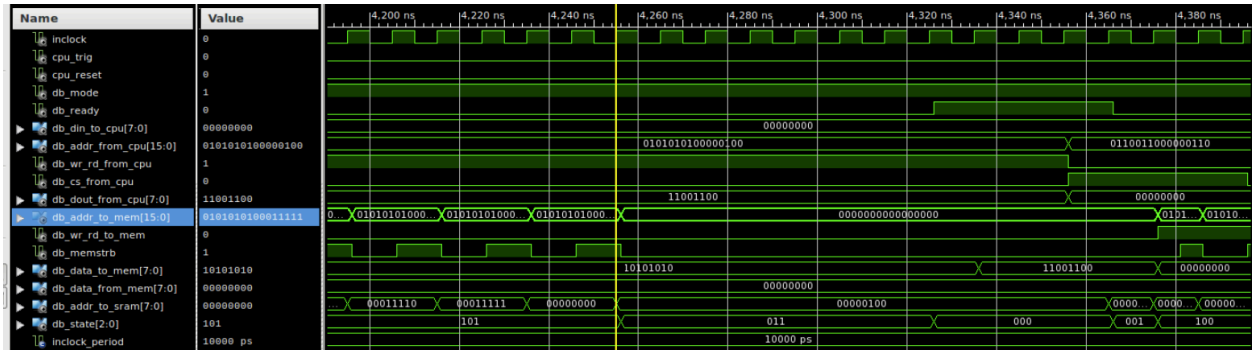

Figure 16: Functional Results for State 5 to State 3 and then State 3 to State 0

Results

Table 1: Cache Performance Parameters

N	Cache performance parameter	Time in ns
1	Hit / Miss determination time	10
2	Data access time	20
3	Block replacement time	650
4	Hit time (Case 1 and 2)	30.5
5	Miss penalty for Case 3 (when D-bit = 0)	705
6	Miss penalty for Case 4 (when D-bit = 1)	1352

With a 20 ns clock period, the system can assess cache hits or misses at each clock edge, which occurs every 10 ns. The hit/miss determination time is capped at 10 ns due to the system's sampling rate limitations.

The rest of the parameters can be determined from the simulation results or ChipScope. For this report, I used the functional simulation as I didn't have enough time to go to the lab and capture waveforms on ChipScope. Also, the functional simulation provided a clearer picture of the waveform and it's easier to calculate the time in between each signal. The block replacement time is calculated from the waveform when the MEMSTRB signal started and stopped, which is 650 ns.

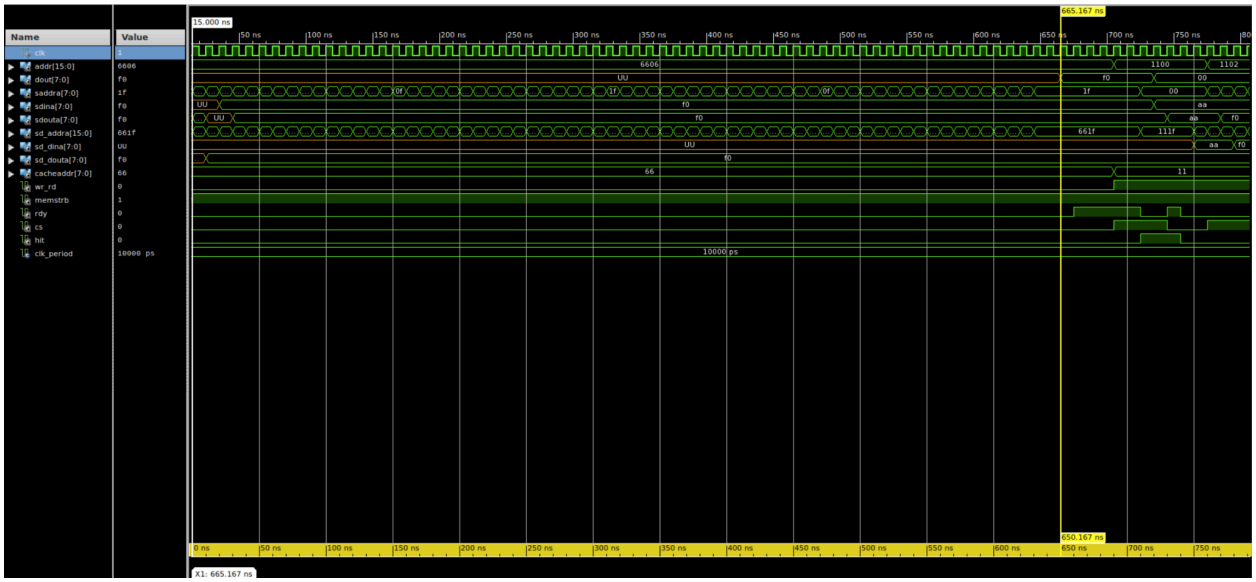

Figure 17: The block replacement time in simulation result

The hit time (Case 1 and 2) is indicated by the length of state 0001 (or state 1) in the waveform, which is 30.5 ns.

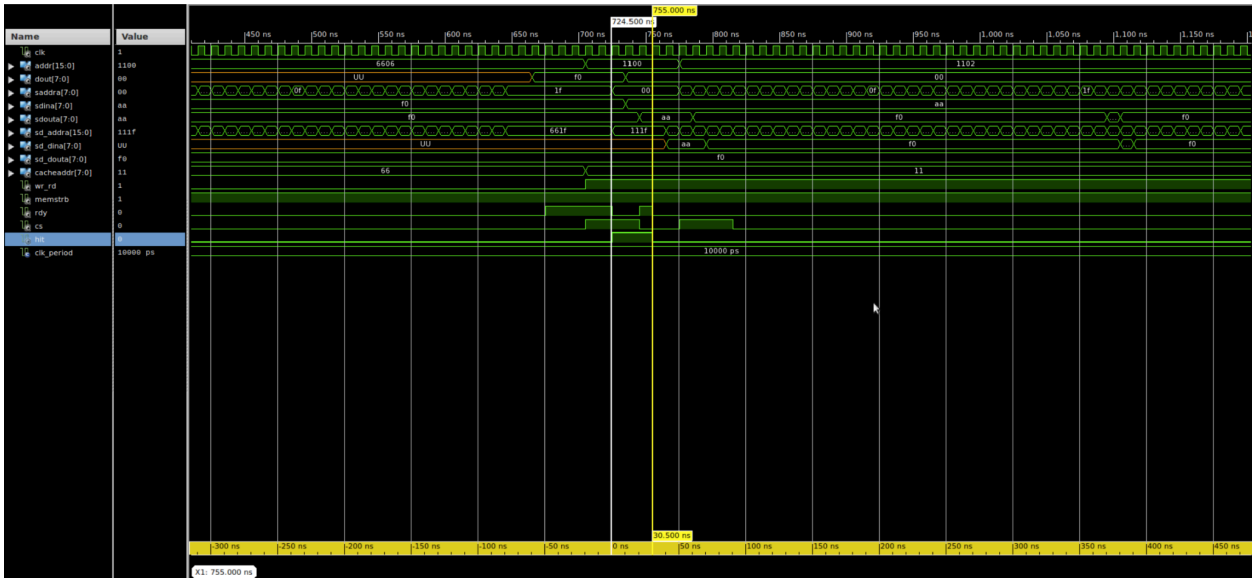

Figure 18: The hit time in simulation result

The miss penalty for Case 3 (when D-bit is 0) is given by the hit/miss determination time, state 0010 time, and hit time, which is 795 ns.

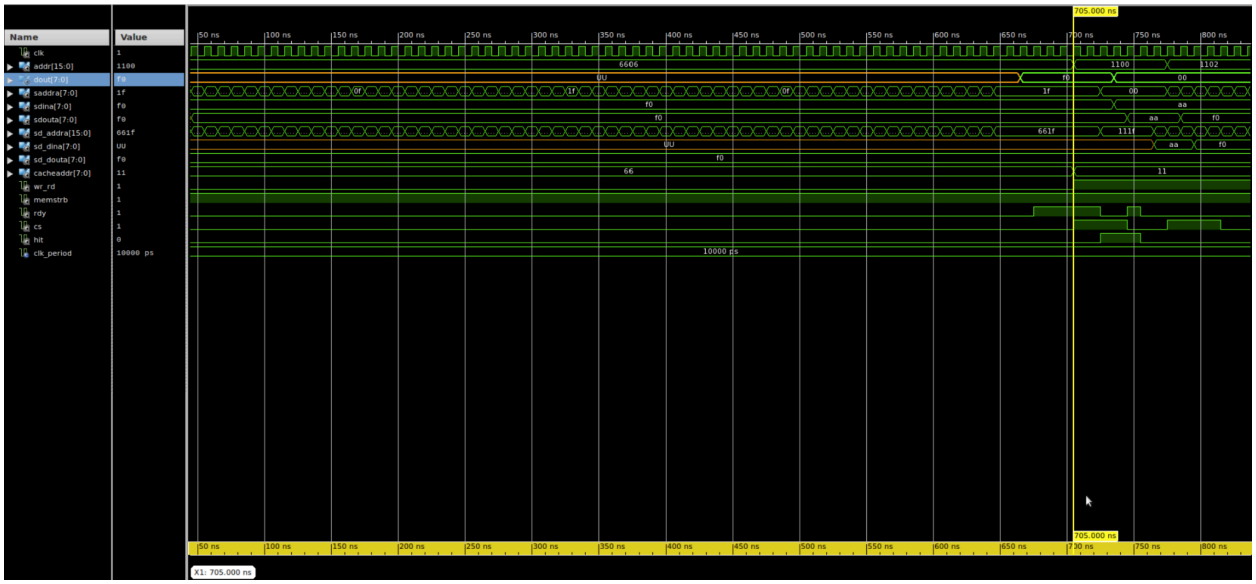

Figure 19: The miss penalty for Case 3 in simulation result

The miss penalty for Case 4 (when D-bit is 1) is given by the hit/miss determination time, state 0100 time, and hit time, which is 1352 ns.

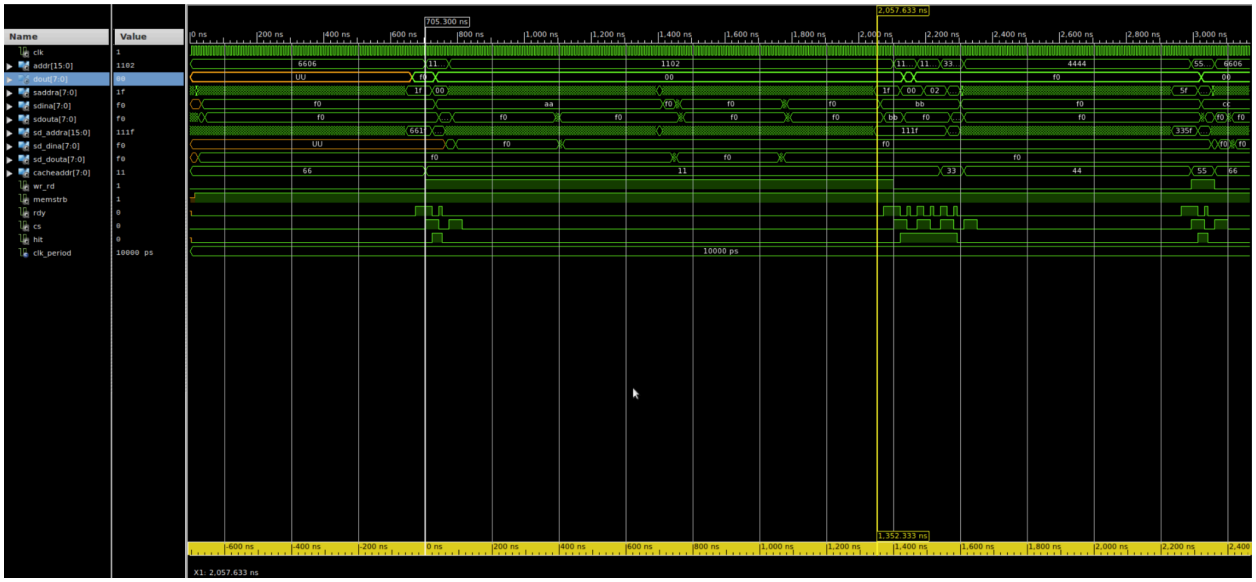

Figure 20: The miss penalty for Case 4 in simulation result

Conclusion

In this lab, our goal was to design a cache controller and simulate its functionality using VHDL. I have successfully developed a state-driven system to manage cache hits and misses, memory operations, and overall cache control. The system accounts for CPU requests, valid and dirty bits, and memory readiness. On cache hits, it quickly supplies the CPU with data from the cache, improving data access speed. On cache misses, it handles cache allocation and ensures that any dirty data is written back to main memory. Overall, this lab enhanced our understanding of cache controller design using the Xilinx ISE CAD environment and gave us insights into its interaction with block-memory (SRAM-based) and SDRAM controllers.

P.S: As mentioned earlier, I used functional simulation for my timing diagrams and results. My ChipScope works but it depends on what machine I use. There are times when a machine only has limited ports, there are also times when the VIO console does not show up, and there are times when my waveform does not show up. Due to these problems and my chaotic schedule this midterm season, I was unable to use ChipScope simulation for my timing diagrams and results.

Appendix

Figure A1: VHDL Implementation of SDRAM Controller

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

entity sdram_controller is
    port (
        clk     : in std_logic;
        memstrb : in std_logic;
        wea     : in std_logic_vector(0 downto 0);
        address : in std_logic_vector(11 downto 0);
        d_in    : in std_logic_vector(7 downto 0);
        d_out   : out std_logic_vector(7 downto 0)
    );
end sdram_controller;

architecture Behavioral of sdram_controller is

    -- Signal declarations without 'in' or 'out'
    signal sdram_rd_wr    : std_logic_vector(0 downto 0);
    signal sdram_add_sig  : std_logic_vector(11 downto 0);
    signal sdram_din_sig  : std_logic_vector(7 downto 0);
    signal sdram_dout_sig : std_logic_vector(7 downto 0);

    signal empty          : std_logic := '0';
    signal counter        : std_logic_vector(11 downto 0) := "000000000000";

    -- SDRAM component declaration
    component sdram
        port (
            clka  : in std_logic;
            wea   : in std_logic_vector(0 downto 0);
            addra : in std_logic_vector(11 downto 0);
            dina  : in std_logic_vector(7 downto 0);
            douta : out std_logic_vector(7 downto 0)
        );
    end component;

begin
    -- Instantiate the SDRAM component and map signals correctly
    sys_sdram : sdram
        port map (
            clka  => clk,             -- Clock signal mapping
            wea   => sdram_rd_wr,     -- Write enable signal
            addra => sdram_add_sig,   -- Address signal
            dina  => sdram_din_sig,   -- Data input signal
            douta => sdram_dout_sig   -- Data output signal
        );

    -- Main process for controlling SDRAM read/write
    main_process : process(clk)
    begin
        if rising_edge(clk) then
            -- Assign address signal
            sdram_add_sig <= address;

            -- Memory strobe logic
            if memstrb = '1' then
                -- If write enable is low, it's a read operation
                if wea(0) = '0' then
                    sdram_rd_wr(0) <= '0';  -- Read mode
```

```
                    d_out <= sdram_dout_sig; -- Output data from SDRAM
                else
                    -- Write operation
                    sdram_rd_wr(0) <= '1';  -- Write mode
                    sdram_din_sig <= d_in;  -- Input data to SDRAM
                end if;
            end if;
        end if;
    end process;

end Behavioral;
```

Figure A2: VHDL Implementation of Cache Controller

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

entity cache_controller is
    Port ( clk : in  STD_LOGIC);
end cache_controller;

architecture Behavioral of cache_controller is

   -- ICON --
   component icon
   PORT (
      CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
      CONTROL1 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0));
   end component;

   signal control0 : STD_LOGIC_VECTOR(35 downto 0);
   signal control1 : STD_LOGIC_VECTOR(35 downto 0);

   -- ILA --
   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 downto 0)
   );
   end component;

   signal ila_data : STD_LOGIC_VECTOR(99 downto 0);
   signal trig : STD_LOGIC_VECTOR(0 downto 0);

   -- VIO --
   component vio
   PORT (
    CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    ASYNC_OUT : OUT STD_LOGIC_VECTOR(25 DOWNTO 0)
    );
   end component;
```

```
59
60    signal vio_sig : STD_LOGIC_VECTOR(25 downto 0);
61
62    -- CACHE MEMORY --
63    component cache_mem
64      PORT (
65        clka : IN STD_LOGIC;
66        wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
67        addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
68        dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
69        douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
70      );
71    end component;
72
73    -- SDRAM CONTROLLER --
74    component sdram_controller
75    PORT(
76       clk : IN std_logic;
77       memstrb : IN std_logic;
78       wea : IN std_logic_vector(0 to 0);
79       address : IN std_logic_vector(11 downto 0);
80       d_in : IN std_logic_vector(7 downto 0);
81       d_out : OUT std_logic_vector(7 downto 0)
82       );
83    end component;
84
85    -- CACHE CONTROLLER REGISTERS --
86    type tag is array(0 to 7) of STD_LOGIC_VECTOR(3 downto 0);
87
88    signal tag_reg : tag;
89    signal vbit : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
90    signal dbit : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
91
92
93    -- CACHE CONTROLLER SIGNALS --
94    signal clk_counter : STD_LOGIC_VECTOR(7 downto 0);
95    signal state_sig : STD_LOGIC_VECTOR(2 downto 0) := "000";
96    signal next_state : STD_LOGIC_VECTOR(2 downto 0);
97    signal hit_sig, dirty_sig : STD_LOGIC;
98    signal cache_to_main, main_to_cache : STD_LOGIC;
99    signal offset : STD_LOGIC_VECTOR(4 downto 0) := "00000";
100   signal counter : integer;
101
102   -- CPU SIGNALS --
103   signal cpu_address : STD_LOGIC_VECTOR(11 downto 0);
104   signal cpu_tag : STD_LOGIC_VECTOR(3 downto 0);
105   signal cpu_index : STD_LOGIC_VECTOR(2 downto 0);
106   signal index : integer;
107   signal cpu_dout : STD_LOGIC_VECTOR(7 downto 0);
108   signal cpu_din : STD_LOGIC_VECTOR(7 downto 0);
109   signal cpu_rd_wr, cpu_cs, cpu_rdy : STD_LOGIC;
110
111   -- CACHE MEMORY SIGNALS --
112   signal cache_add_sig : STD_LOGIC_VECTOR(7 downto 0);
113   signal cache_rd_wr : STD_LOGIC_VECTOR(0 downto 0);
114   signal cache_din_sig, cache_dout_sig : STD_LOGIC_VECTOR(7 downto 0);
115
116   -- MAIN MEMORY SIGNALS --
117   signal main_add_sig : STD_LOGIC_VECTOR(11 downto 0);
118   signal main_din_sig : STD_LOGIC_VECTOR(7 downto 0);
```

```
119     signal main_dout_sig : STD_LOGIC_VECTOR(7 downto 0);
120     signal main_rd_wr : STD_LOGIC_VECTOR(0 downto 0);
121     signal main_memstrb : STD_LOGIC;
122
123  begin
124
125     -- PORT MAP --
126
127     sys_icon : icon port map(control0, control1);
128     sys_ila : ila port map(control0, clk, ila_data, trig);
129     sys_vio : vio port map(control1, vio_sig);
130     sys_cache : cache_mem port map(clk, cache_rd_wr, cache_add_sig, cache_din_sig, cache_dout_s
131     sys_sdram : sdram_controller port map(clk, main_memstrb, main_rd_wr, main_add_sig, main_din
132
133     -- ACTUAL PROCESSES --
134
135     -- CHECK HIT OR MISS --
136     hit_or_miss : process(cpu_address, tag_reg)
137     begin
138        if(tag_reg(index) = cpu_tag and vbit(index) = '1') then
139           hit_sig <= '1';
140        else
141           hit_sig <= '0';
142        end if;
143
144     end process;
145
146     -- UPDATE STATE --
147     update_state : process(clk, state_sig, next_state)
148     begin
149        if (clk'event and clk='1') then
150           state_sig <= next_state;
151        end if;
152     end process;
153
154     -- GENERATE THE NEXT STATE --
155     fsm : process(clk, cpu_cs, state_sig, cache_to_main, main_to_cache)
156     begin
157        if (clk'event and clk='1') then
158           if (state_sig = "000") then
159           -- IDLE STATE --
160           -- LET CPU KNOW THAT IT IS READY FOR TRANSACTION --
161              if (cpu_cs = '1') then
162           -- IF THE CPU TURNS ON CS THEN CHECK READ WRITE --
163                 next_state <= "001";
164              end if;
165           elsif (state_sig = "001") then
166              -- GO TO:
167              --    STATE 2 : READ
168              --    STATE 3 : WRITE
169              --    STATE 4 : WRITE CACHE TO MAIN
170              --    STATE 5 : WRITE MAIN TO CACHE
171              if (hit_sig = '1') then
172                 if (cpu_rd_wr = '0') then
173                 -- READ --
174                    next_state <= "010";
175                 else
176                 -- WRITE --
177                    next_state <= "011";
178                 end if;
```

```
            else
               if (dbit(index) = '0') then
               -- CACHE TO MAIN --
                  next_state <= "100";
               else
               -- MAIN TO CACHE --
                  next_state <= "101";
               end if;
            end if;
         elsif (state_sig = "010") then
         -- IN READ STATE --
            next_state <= "000";
         elsif (state_sig = "011") then
         -- IN WRITE STATE --
            next_state <= "000";
         elsif (state_sig = "100") then
         -- IN MAIN TO CACHE STATE --
            if (main_to_cache = '1') then
               next_state <= "001";
            end if;
         elsif (state_sig = "101") then
         -- IN CACHE TO MAIN STATE --
            if (cache_to_main = '1') then
               next_state <= "100";
            end if;
         end if;
      end if;
   end process;


   -- GENERATE OUTPUT SIGNALS OF STATES --
   gen_output : process(clk, state_sig)
   begin
      if (clk'event and clk='1') then
         if (state_sig = "000") then
         -- IDLE STATE --
            cpu_rdy <= '1';
         elsif (state_sig = "001") then
         -- TRANSITION STATE --
            offset <= "00000";

            cache_to_main <= '0';
            main_to_cache <= '0';
            cpu_rdy <= '0';
            counter <= 0;
         elsif (state_sig = "010") then
         -- READ STATE --
            cache_rd_wr(0) <= '0';
            cache_add_sig <= cpu_address(7 downto 0);
            cpu_din <= cache_dout_sig;
         elsif (state_sig = "011") then
         -- WRITE STATE --
            cache_rd_wr(0) <= '1';
            cache_add_sig <= cpu_address(7 downto 0);
            cache_din_sig <= cpu_dout;
            dbit(index) <= '1';
         elsif (state_sig = "100") then
         -- MAIN TO CACHE --
            if (counter = 64) then
               counter <= 0;
               offset <= "00000";
```

```
239                tag_reg(index) <= cpu_tag;
240                vbit(index) <= '1';
241                dbit(index) <= '0';
242                main_to_cache <= '1';
243             else
244
245                if (counter mod 2 = 1) then
246                   main_memstrb <= '0';
247                else
248                   -- MAIN SIGNALS --
249                   main_memstrb <= '1';
250                   main_rd_wr(0) <= '0';
251                   --main_memstrb <= '1';
252                   main_add_sig(11 downto 8) <= cpu_tag;
253                   main_add_sig(7 downto 5) <= cpu_index;
254                   main_add_sig(4 downto 0) <= offset;
255
256
257                   -- CACHE SIGNALS --
258                   cache_rd_wr(0) <= '1';
259                   cache_add_sig(7 downto 5) <= cpu_address(7 downto 5);
260                   cache_add_sig(4 downto 0) <= offset;
261
262                   -- WRITE MAIN DOUT INTO CACHE DIN --
263                   cache_din_sig <= main_dout_sig;
264
265                   -- INCREMENT THE OFFSET --
266                   offset <= std_logic_vector(unsigned(offset) + 1);
267
268                end if;
269
270                counter <= counter + 1;
271             end if;
272
273          elsif (state_sig = "101") then
274             vbit(index) <= '0';
275          -- CACHE TO MAIN --
276             if (counter = 64) then
277                counter <= 0;
278                offset <= "00000";
279                cache_to_main <= '1';
280             else
281                -- OSCILLATE THE MAIN_MEMSTRB SIGNAL TO ENSURE THAT THE DATA IS STABLE --
282                if (counter mod 2 = 1) then
283                   main_memstrb <= '0';
284                else
285                   -- MAIN SIGNALS --
286                   --    WRITE TO MAIN --
287                   main_memstrb <= '1';
288                   main_rd_wr(0) <= '1';
289                   --main_memstrb <= '1';
290                   main_add_sig(11 downto 8) <= tag_reg(index);
291                   main_add_sig(7 downto 5) <= cpu_index;
292                   main_add_sig(4 downto 0) <= offset;
293
294                   -- CACHE SIGNALS --
295                   cache_rd_wr(0) <= '0';
296                   cache_add_sig(7 downto 5) <= cpu_index;
297                   cache_add_sig(4 downto 0) <= offset;
298
```

```
                    -- WRITE CACHE DOUT TO MAIN DIN --
                    main_din_sig <= cache_dout_sig;

                    -- INCREMENT THE OFFSET --
                    offset <= std_logic_vector(unsigned(offset) + 1);

                end if;
                counter <= counter + 1;
            end if;
        end if;
    end if;
end process;



-- WIRES --
cpu_tag <= cpu_address(11 downto 8);
cpu_index <= cpu_address(7 downto 5);
index <= to_integer(unsigned(cpu_index));

-- VIO --
cpu_address <= vio_sig(11 downto 0);
cpu_dout <= vio_sig(19 downto 12);
cpu_cs <= vio_sig(20);
cpu_rd_wr <= vio_sig(21);

-- ILA --
trig(0) <= cpu_cs;

-- CPU DATA
ila_data(0) <= cpu_rdy;
ila_data(1) <= cpu_cs;
ila_data(2) <= cpu_rd_wr;
ila_data(14 downto 3) <= cpu_address;
ila_data(22 downto 15) <= cpu_din;
ila_data(31 downto 24) <= cpu_dout;

-- CACHE CONTROLLER DATA --
ila_data(32) <= hit_sig;
ila_data(33) <= vbit(index);
ila_data(34) <= dbit(index);
ila_data(37 downto 35) <= state_sig;

-- CACHE MEMORY DATA --
ila_data(38) <= cache_rd_wr(0);
ila_data(46 downto 39) <= cache_add_sig;
ila_data(54 downto 47) <= cache_din_sig;
ila_data(62 downto 55) <= cache_dout_sig;

-- MAIN MEMORY DATA --
ila_data(63) <= main_rd_wr(0);
ila_data(64) <= main_memstrb;
ila_data(76 downto 65) <= main_add_sig;
ila_data(84 downto 77) <= main_din_sig;
ila_data(92 downto 85) <= main_dout_sig;


end Behavioral;
```