RYERSON UNIVERSITY

Faculty of Engineering, Architecture and Science

Department of Electrical and Computer Engineering

Course Number	**COE 758**
Course Title	**Digital Systems Engineering**
Semester/Year	**Fall 2021**
Instructor	**Dr. Lev Kirischian**

Lab/Tutorial Report NO. 1

Report Title	**Design Project 1**

Section No.	01
Group No.	01
Submission Date	Oct. 29th, 2021
Due Date	Oct. 29th, 2021

Name	Student ID	Signature*
Darien Lee	xxxxx8176	DL
Taimoor Farooqi	xxxxx3873	TF

(Note: remove the first 4 digits from your student ID)

**By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission 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/policies/pol60.pdf.*

Taimoor Farooqi
Darien Lee

Design Project 1

Abstract

The goal of this design project is to design and implement an SDRAM controller and a 256-byte cache consisting of a cache controller and SRAM cache. The CPU will issue 16-bit address words to the cache controller which responds to the instructions received accordingly. The behavioral cases that the cache controller must carry out include 4 cases: *1)* write a word to cache [hit], *2)* read a word from cache [hit], *3)* read/write from/to cache [miss] and dirt bit = 0, and *4)* read/write from/to cache [miss] and dirt bit = 1.

This design project resulted in a cache controller that meets the 4 behavioral requirements stated above by examining the simulation waveforms, and the dirty and valid bits to ensure the proper functionality of the implemented project.

Introduction

Different types of memory in modern computers serve similar functionality of storing, reading and processing data but serve different purposes within the system. For example, random access memory (RAM) provides high-speed read/write access to information but this information is stored only temporarily and will be lost when the system is turned off or reset. Read-only memory (ROM) is another form of memory but is non-volatile unlike RAM and stores data permanently only to be read which is not lost upon system reset. Even within RAM, there are subcategories of memory such as static random access memory (SRAM), dynamic random access memory (DRAM), and many others. All these different types of memories fall within a memory hierarchy with respect to the processor. In general, the more distance away from the processor the greater the volume and access time of memory.

System Specifications

Behavioral/Functional:

_____There are 4 distinct behavioral cases found available for the cache controller. These 4 cases include:

1. The cache controller receives a write request from the CPU resulting in a cache hit being detected. In this case, the index and offset data provided by the CPU is passed to the SRAM as the write address for the new data. In addition, the multiplexer writes the data to the SRAM received from the CPU only when the write bit is enabled.
2. The cache received a read request from the CPU. In this scenario, we should also see a cache hit occurring. Similar to the first case, the index and offset data are then transmitted to the SRAM and the read data is routed back to the CPU.
3. The third behavioral case occurs when a read/write request is received but the associated block is not found in the cache/SRAM, and the dirty bit of that address being

set to “00000”. In order for the system to continue, the tag value must replace the value in the corresponding tag register and the valid bit set to 1. Afterwards, the read/write operation can continue without delay.

4. The fourth and final case is similar to the third case however after the analysis of the dirty bit, it is found to be set to 1. This will require the appropriate block to be rewritten to the SDRAM prior to the system continuing to read from the main memory.

Device Description / Design

Symbols

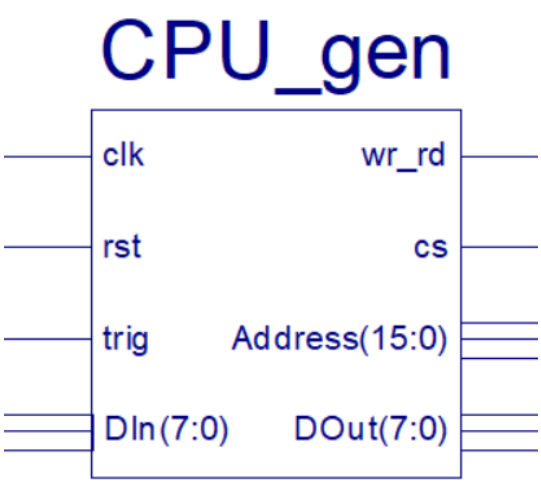


Figure 1: CPU Symbol

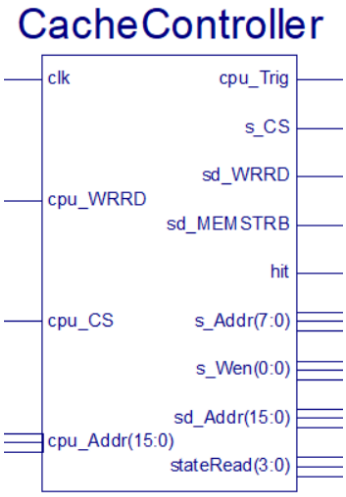


Figure 2: Cache Controller Symbol

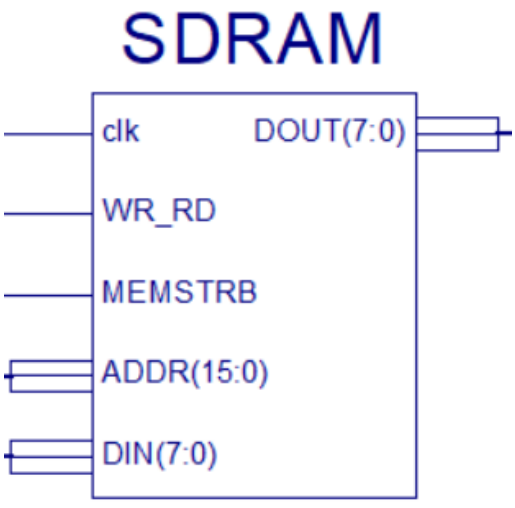


Figure 3: SDRAM Symbol

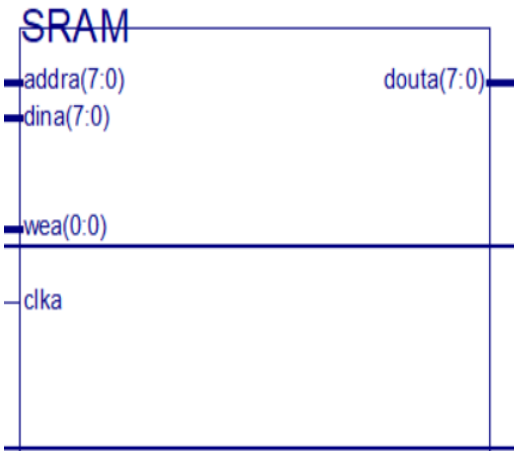
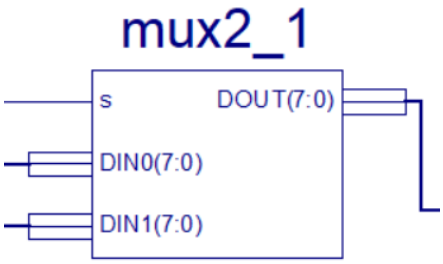


Figure 4: SRAM Symbol

Taimoor Farooqi
Darien Lee



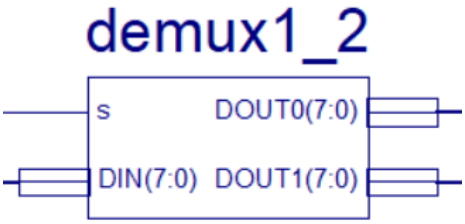


Figure 5: 2 to 1 Multiplexer Symbol **Figure 6: 1 to 2 Demultiplexer Symbol**

Symbols (Cont'd)

Figures 1 to **Figure 6** represent the individual blocks that were used in our compiled block diagram showcasing the entire Cache Controller in **Figure 7**. These symbols were created after declaring the components in our individual .VHD files in Xilinx ISE 13.4 program. After these components are declared and our code's syntax is verified, we are able to create the block by selecting the "Create Schematic Symbol" option inside the Design tab of our implementation view. After this, we create a new source file specifying that we need a new schematic file type. The program then opens up the schematic workbench and we are able to individually add each block and connect wires between the appropriate buses/links.

Block Diagram

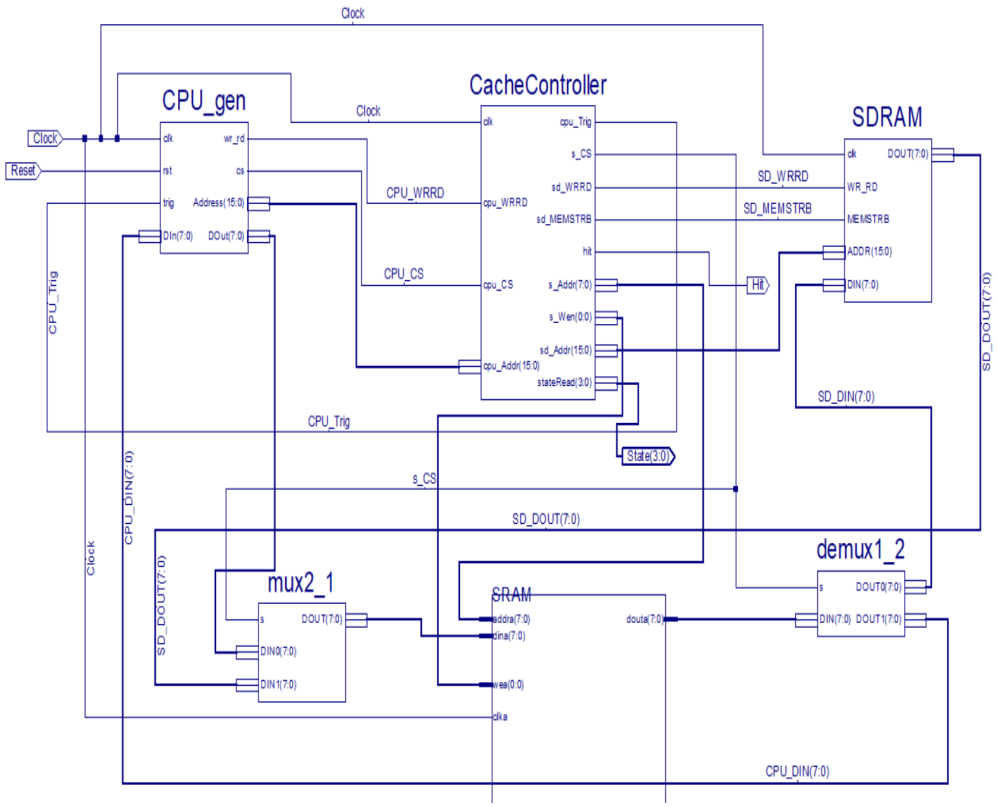


Figure 7: Block Diagram

Block Diagram (Cont'd)

Figure 7 showcases the entire block diagram of our Cache Controller with all the individual components connected with the appropriate wire connections. It is worth noting that the thicker/bolder wires represent buses, which have multiple vectors associated with them (i.e 7:0 representing an 8 input vector), whereas thin wires represent a simple connection between source and destination with just 1 bit being transmitted.

Our main inputs begin with the clock and reset blocks. Clock synchronizes the CPU, cache controller block, SDRAM and SRAM components. The CPU then proceeds to transmit a 16-bit address over to the cache controller. The cache controller then is responsible for either reading or writing to local memory (SRAM), which is dependent on the statuses of each component in the cache controller. In addition to the cache controller transmitting data to the SRAM, the CPU also outputs an 8 bit address to the SRAM via a 2-to-1 multiplexer with the selector bit coming from the cache controller component.

The 2-to-1 multiplexer and 1-to-2 demultiplexer are used for transmitting data over to the CPU, processing data from the SRAM, and evaluating the data to determine whether or not a Cache Miss has been detected. In between these 2 components, the SRAM is also known as the Cache storage component or simply the Cache of the system. It receives information from all components including the CPU, SDRAM, cache controller and the 2 muxes. The final component is the SDRAM, which is responsible for evaluating data transmitted by the cache controller which includes data transmitted via a 16-bit address bus, a Write/Read input and a MEMSTRB input provided by the cache controller, whilst also taking input from the clock and the 1-to-2 demultiplexer.

State Diagram

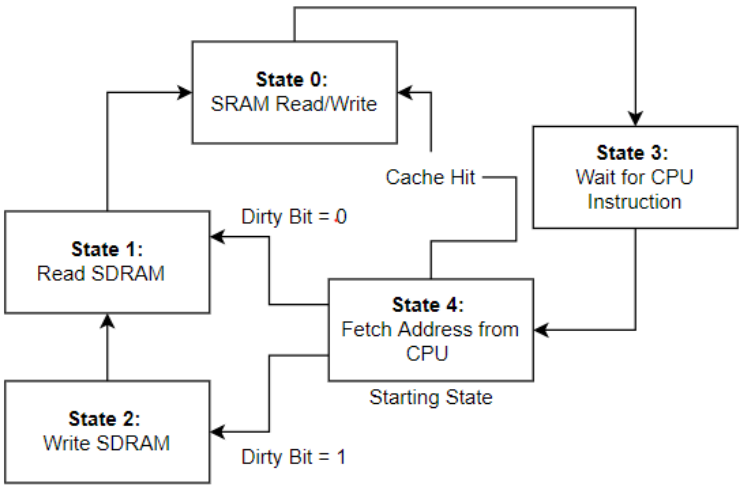


Figure 8: State Diagram

Above in **Figure 8** is a state diagram of the finite state machine within our cache controller. The initial state is State 4 as the controller fetches the address from the CPU. The CPU issues the read/write enable bits and controls signals including addresses to be fetched by the cache controller. In State 4, the cache controller checks the SRAM cache for a Cache Hit or a Cache Miss.

When a Cache Hit is detected by the cache controller, the cache controller will enter State 0 and proceed to read/write this data from the SRAM to the CPU's data in port. Then, the state machine will enter State 3 and set the cpu_trig signal to high which indicates that the cache controller is now idle and awaiting instruction from the CPU. Upon receiving instructions from the CPU, the cache controller will then enter back into State 4.

When a Cache Miss is detected by the cache controller, the cache controller will move to State 1 or State 2 depending on the value of the dirty bit. If dirty bit = 1, the controller will enter State 2, and the dirty block is written to the SDRAM memory. Then, the cache controller moves to State 1 to retrieve the block missing from the cache. After that, the cache controller moves to State 0 and reads/writes to the SRAM. If dirty bit = 0, the cache controller moves from State 4 to State 1, skipping the write operation and applying the read operation to the SDRAM. Finally, it moves from State 1 to State 0 after retrieving the data from SDRAM to store in the SRAM cache. From State 0, the controller will move to State 3 and then back to State 4 in the same manner as described in the previous section with the Cache Hit.

Process Diagram

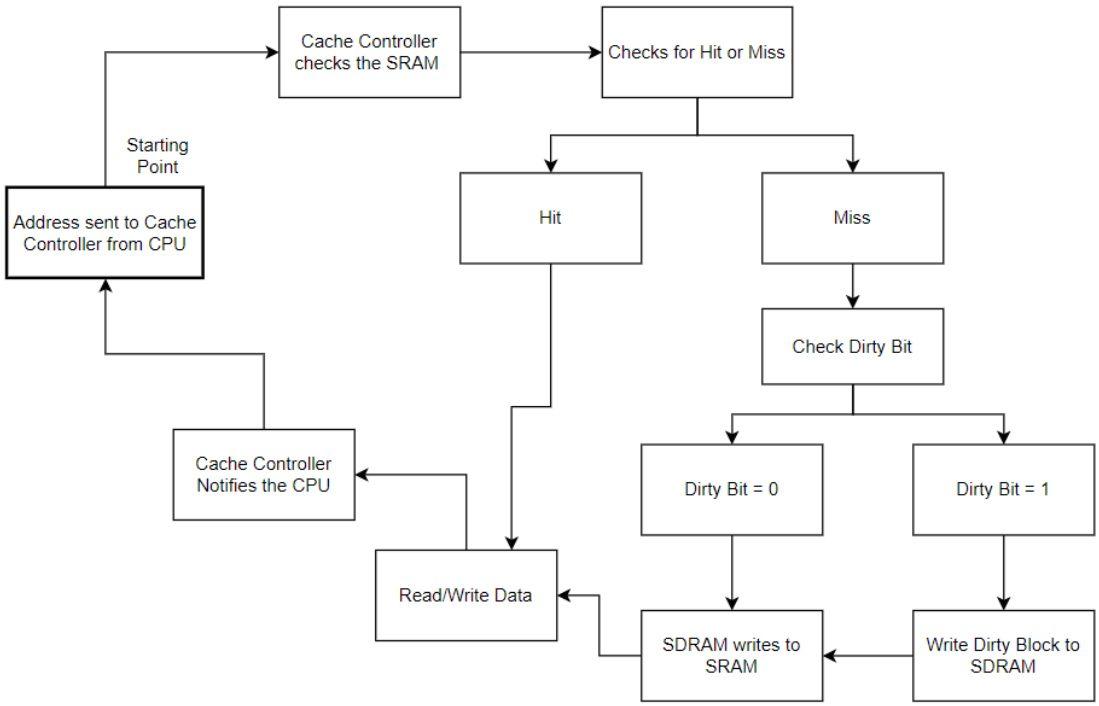


Figure 9: Process Diagram

Taimoor Farooqi
Darien Lee

Figure 9 depicts the process diagram of the cache controller. The process begins with the CPU sending instructions and an address to the cache controller. The cache controller checks the SRAM for a hit or miss. If it is a hit, the controller can read/write to the SRAM. If it is a miss, the controller will check the dirty bit to determine the next steps. If the dirty bit is 0, the SDRAM will write to the SRAM then the cache controller continues to the original instructions of reading/writing the data. Finally, the cache controller notifies the CPU and awaits for new instructions to begin the cycle again. If the dirty bit is 1, then the cache controller will proceed to write the block back to the SDRAM before the SDRAM writes to the SRAM. The remainder steps follow the same final steps of the cycle of reading/writing the data as per original instructions and so on until the beginning of a new cycle.

Results

Functional Simulation

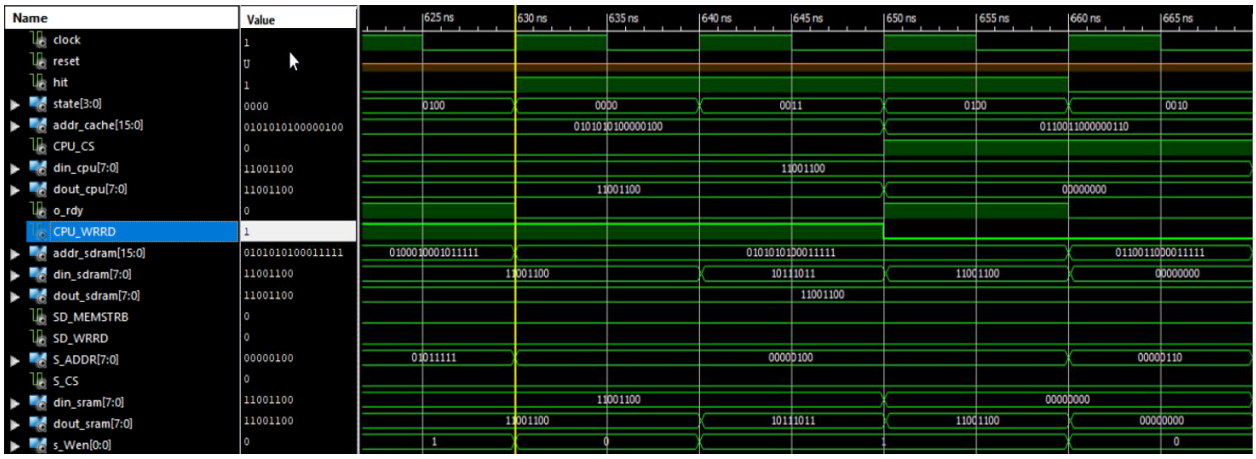


Figure: 10: Functional Simulation Diagram

In **Figure 10**, a functional simulation of the designed program is depicted. At 630ns, we can see the operation of the cache controller from State 4 in which an address and other instructions are fetched from the CPU. The “CPU_CS” (control signal) is at 0 and the “CPU_WRRD” (write/read signal) is at 1, indicating a write operation to the SRAM. The “hit” signal is at 1 which determines that the controller detected a hit and a state change from State 4 to State 0 occurs as seen in “state[3:0]”. As a result, we see here at 640ns that the “dout_sram” is made equal to the “din_cpu”, which works as intended indicating that the read operation to the CPU was successful. The controller then moves into State 3 at 640ns and sits idle awaiting further instructions from the CPU to enter State 4 again. This cycle repeats with next clocks.

Taimoor Farooqi
Darien Lee

Conclusions

The cache controller designed in the project was successfully implemented. This project provided knowledge and insight into the importance of a well-designed memory controller. The use of dirty and valid bits resulted in the optimization of the flow of data through memory and the controller helped by enabling/disabling the writing/reading to cache when appropriate. Overall, the project showed how the design of memory controllers can greatly contribute to the ease of use and time efficiency of the user.

References

- Kirischian, Lev. “Project #1 – Memory Hierarchy: Cache Controller.” COE758 ,www.ee.ryerson.ca/~lkirisch/ele758/handouts/COE758_Digital_Design_Tutorial.pdf.

- Kirischian, Lev. “Project 1: Cache Controller – Secondary Component Specifications.” COE758 , www.ee.ryerson.ca/~lkirisch/ele758/handouts/P1_interfaces.pdf.

Appendix

CacheController.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity CacheController is
5     Port (
6        cpu_Addr : in STD_LOGIC_VECTOR(15 downto 0);
7        clk : in STD_LOGIC;
8        cpu_WRRD : in STD_LOGIC;
9        cpu_CS : in STD_LOGIC;
10       cpu_Trig : out STD_LOGIC;
11
12       s_Addr : out STD_LOGIC_VECTOR(7 downto 0);
13       s_Wen : out STD_LOGIC_VECTOR(0 downto 0);
14       s_CS : out STD_LOGIC;
15
16       sd_Addr : out STD_LOGIC_VECTOR(15 downto 0);
17       sd_WRRD : out STD_LOGIC;
18       sd_MEMSTRB : out STD_LOGIC;
19
20       stateRead : out STD_LOGIC_VECTOR(3 downto 0);
21       );
22 end CacheController;
23
24 architecture Behavior of CacheController is
25 --CPU Signals
26    signal tag : STD_LOGIC_VECTOR(7 downto 0);
27    signal index : STD_LOGIC_VECTOR(2 downto 0);
28    signal offset : STD_LOGIC_VECTOR(4 downto 0);
29
30 --Dirty Bit Signal
31    signal dBit : STD_LOGIC_VECTOR(7 downto 0):= "00000000";
32
33 -- Valid Bit Signal
34    signal vBit : STD_LOGIC_VECTOR(7 downto 0):= "00000000";
35
36 -- SRAM Cache Array
37    type cachememory is array (7 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
38    signal memtag: cachememory := ((others=> (others=>'0')));
39
40 -- SDRAM Signals
41    signal counter : integer := 0;
42    signal sd_Offset : integer := 0;
```

Taimoor Farooqi
Darien Lee

```
-- FSM State Signals --
   -- State 0: sram Read/Write
   -- State 1: main memory read
   -- State 2: main memory write
   -- State 3: Idle State
   -- State 4: Hit/Miss Detection and Address Retrieval
   TYPE state_value IS (state4, state0, state1, state2, state3);
   signal state_current : state_value ;
   signal state : STD_LOGIC_VECTOR(3 downto 0);

   begin
      process(clk, cpu_CS)
   begin
      if (clk'event AND clk = '1') then -- rising edge of clock
-- State 4
      if (state_current = state4) then
         cpu_Trig <= '0';
         tag <= cpu_Addr(15 downto 8);
         index <= cpu_Addr(7 downto 5);
         offset <= cpu_Addr(4 downto 0);
         sd_Addr(15 downto 5) <= cpu_Addr(15 downto 5);
         s_Addr(7 downto 0) <= cpu_Addr(7 downto 0);
         s_Wen <= "0";
      if (vBit(to_integer(unsigned(index))) = '1' AND memtag(to_integer(unsigned(index))) = tag) then
         hit <= '1';
         state_current <= state0;
         state <= "0000";
         stateRead <= "0000";
-- Cache Miss
      else
         hit <= '0';
-- If Dirty Bit and Valid Bit in Block are 1
-- Switches to State 2 to write back to SDRAM
      if (dBit(to_integer(unsigned(index))) = '1' AND
         vBit(to_integer(unsigned(index))) = '1') then
         state_current <= state2;
         state <= "0010";
         stateRead <= "0010";
-- If Dirty Bit is 0 in Block and a Cache Miss occurs
-- Switches to State 1 to read from SDRAM
      else
         state_current <= state1;
         state <= "0001";
         stateRead <= "0001";
      end if;
      end if;
-- State 0
      elsif(state_current = state0) then
      if (cpu_WRRD = '1') then
         s_Wen <= "1";
         s_CS <= cpu_CS;
         dBit(to_integer(unsigned(index))) <= '1';
         vBit(to_integer(unsigned(index))) <= '1';
      else
      s_Wen <= "0";
      s_CS <= cpu_CS;
      end if;
      state_current <= state3;
      state <= "0011";
      stateRead <= "0011";
-- State 1
      elsif(state_current = state1) then
      if (counter = 64) then
         counter <= 0;
         vBit(to_integer(unsigned(index))) <= '1';
         memtag(to_integer(unsigned(index))) <= tag;
         sd_Offset <= 0;
         state_current <= state0;
         state <= "0000";
         stateRead <= "0000";
      else
      if (counter mod 2 = 1) then
         sd_MEMSTRB <= '0';

      else
         s_CS <= cpu_CS;
         sd_Addr(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(sd_Offset, offset'length));
         sd_WRRD <= '0';
         sd_MEMSTRB <= '1';
         s_Addr(7 downto 5) <= index;
         s_Addr(4 downto 0) <=
```

```
124         STD_LOGIC_VECTOR(to_unsigned(sd_Offset, offset'length));
125         s_Wen <= "1";
126         sd_Offset <= sd_Offset + 1;
127      end if;
128      counter <= counter + 1;
129      end if;
130   -- State 2
131      elsif(state_current = state2) then
132
133      if (counter = 64) then
134         counter <= 0;
135         dBit(to_integer(unsigned(index))) <= '0';
136         sd_Offset <= 0;
137         state_current <= state1;
138         state <= "0001";
139         stateRead <= "0001";
140      else
141      if (counter mod 2 = 1) then
142         sd_MEMSTRB <= '0';
143
144      else
145         s_CS <= cpu_CS;
146         sd_Addr(4 downto 0) <=
147         STD_LOGIC_VECTOR(to_unsigned(sd_Offset, offset'length));
148         sd_WRRD <= '1';
149         s_Addr(7 downto 5) <= index;
150         s_Addr(4 downto 0) <=
151         STD_LOGIC_VECTOR(to_unsigned(sd_Offset, offset'length));
152         s_Wen <= "0";
153         sd_MEMSTRB <= '1';
154         sd_Offset <= sd_Offset + 1;
155      end if;
156      counter <= counter + 1;
157      end if;
158   -- State 3
159      elsif(state_current = state3) then
160      cpu_Trig <= '1';
161      state_current <= state4;
162      state <= "0100";
163      stateRead <= "0100";
164      end if;
165      end if;
166   end process;
167   end Behavior;
168
```

Taimoor Farooqi
Darien Lee

SDRAM.vhd:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  entity SDRAM is
5     Port (
6        clk : in STD_LOGIC;
7        ADDR : in STD_LOGIC_VECTOR (15 downto 0);
8        WR_RD : in STD_LOGIC;
9        MEMSTRB : in STD_LOGIC;
10       DIN : in STD_LOGIC_VECTOR (7 downto 0);
11       DOUT : out STD_LOGIC_VECTOR (7 downto 0)
12       );
13    end SDRAM;
14 architecture Behavior of SDRAM is
15    -- SDRAM Array
16    type sdmemory is array (7 downto 0, 31 downto 0) of std_logic_vector(7 downto 0);
17    signal sd_SIG: sdmemory;
18    signal initialized : integer := 0;
19    begin
20       process (clk)
21    begin
22       if (clk'event AND clk = '1') then
23       if (initialized = 0) then
24          for I in 0 to 7 loop
25             for J in 0 to 31 loop
26                sd_SIG(i,j) <= "11110000";
27             end loop;
28          end loop;
29       initialized <= 1;
30       end if;
31       if (MEMSTRB = '1') then
32       if (WR_RD = '1') then
33          sd_SIG(to_integer(unsigned(ADDR(7 downto 5))),to_integer(unsigned(ADDR(4 downto 0)))) <= DIN;
34       else
35          DOUT <= sd_SIG(to_integer(unsigned(ADDR(7 downto 5))),to_integer(unsigned(ADDR(4 downto 0))));
36       end if;
37       end if;
38       end if;
39 end process;
40 end Behavior;
```

Taimoor Farooqi
Darien Lee

TopLevel.vhd:

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

entity topLevel is
   PORT(
      CLOCK : in std_logic;

      RESET : in std_logic;
      START  : in std_logic;

      ADDR_CACHE, ADDR_SDRAM : out std_logic_vector(15 downto 0);
      ADDR_SRAM : out std_logic_vector(7 downto 0);
      DIN_CPU, DOUT_CPU, DIN_SDRAM, DOUT_SDRAM, DIN_SRAM, DOUT_SRAM : out std_logic_vector(7 downto 0);
      O_RDY, O_CS, WEN_CACHE, MUX_IN, MUX_OUT, WEN_SRAM, WEN_SDRAM, MEM_STRB : out std_logic;
      CPU_STATE : OUT std_logic_vector(3 downto 0)
   );
end topLevel;

architecture Behavioral of topLevel is
   --SRAM 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;

   --Cache Controller Component
   component CacheController
   Port (
         CLK : in  STD_LOGIC;
         cpu_Addr : in  STD_LOGIC_VECTOR (15 downto 0);
         cpu_WRRD : in  STD_LOGIC;
         cpu_CS : in  STD_LOGIC;

         sd_Addr : out  STD_LOGIC_VECTOR (15 downto 0);
         sd_WRRD : out  STD_LOGIC;
         sd_MEMSTRB : out  STD_LOGIC;

          s_Addr : out  STD_LOGIC_VECTOR (7 downto 0);
          s_Wen : out  STD_LOGIC_VECTOR(0 downto 0);
          s_CS : out  STD_LOGIC;

          cpu_Trig : out std_logic

   );
   end component;

   --SDRAM Component
   component sdram
   Port (CLK      : in  STD_LOGIC;
         ADDR     : in  STD_LOGIC_VECTOR (15 downto 0);
         WR_RD       : 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;

   --CPU Component
   component CPU_gen
   Port (
         clk      : in  STD_LOGIC;
         rst      : in  STD_LOGIC;
         trig     : in  STD_LOGIC;
         DIn      : in  STD_LOGIC_VECTOR (7 downto 0);
         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;

   signal addrCache, addrSDRAM : std_logic_vector(15 downto 0) := (others => '0');
   signal addrSRAM : std_logic_vector(7 downto 0) := (others => '0');
   signal dinCPU, doutCPU, dinSDRAM, doutSDRAM, dinSRAM, doutSRAM : std_logic_vector(7 downto 0) := (others => '0');
   signal rdy, cs, wenCache, muxIn, muxOut, wenSDRAM, memstrb : std_logic := '0';
   signal trig : std_logic := '0';
   signal wenSRAM : STD_LOGIC_VECTOR(0 downto 0);

```

```
83  begin
84     SRAM_1: SRAM port map(
85        clka => CLOCK,
86        addra => addrSRAM,
87        wea => wenSRAM,
88        dina => dinSRAM,
89        douta => doutSRAM
90     );
91
92
93     CacheController_1: CacheController port map(
94        CLK => CLOCK,
95        cpu_Addr => addrCache,
96        cpu_WRRD => wenCache,
97        cpu_CS => cs,
98
99        sd_Addr => addrSDRAM,
100       sd_WRRD => wenSDRAM,
101       sd_MEMSTRB => memstrb,
102
103       s_Addr => addrSRAM,
104       s_Wen(0) => wenSRAM(0),
105       s_CS => muxIn,
106
107       cpu_Trig => rdy
108    );
109
110    sdramC: sdram port map(
111       CLK => CLOCK,
112       ADDR => addrSDRAM,
113       WR_RD => wenSDRAM,
114       MEMSTRB => memstrb,
115       DIN => dinSDRAM,
116       DOUT => doutSDRAM
117    );
118
119    CPU: CPU_gen port map(
120       clk => CLOCK,
121       rst => RESET,
122       trig => trig,
123       DIn => dinCPU,
124        Address => addrCache,
125        wr_rd => wenCache,
126        cs => cs,
127        DOut => doutCPU
128     );
129
130    trigMux: process(START, rdy)
131    begin
132       if(START = '1') then
133          trig <= '1';
134       else
135          trig <= rdy;
136       end if;
137    end process;
138
139    dataInputSRAM: process(muxIn, doutCPU, doutSRAM)
140    begin
141       if(muxIn = '0') then
142          dinSRAM <= doutCPU;
143       else
144          dinSRAM <= doutSDRAM;
145       end if;
146    end process;
147
148    dataOutputSRAM: process(muxOut, doutSRAM)
149    begin
150       if(muxOut ='0') then
151          dinSDRAM <= doutSRAM;
152       else
153          dinCPU <= doutSRAM;
154       end if;
155    end process;
156
157    debug: process(addrCache, addrSDRAM, addrSRAM, dinCPU, doutCPU, dinSDRAM, doutSDRAM,
158    begin
159       ADDR_CACHE <= addrCache;
160       ADDR_SDRAM <= addrSDRAM;
161       ADDR_SRAM <= addrSRAM;
162       DIN_CPU <= dinCPU;
163       DOUT_CPU <= doutCPU;
164       DIN_SDRAM <= dinSDRAM;
165       DOUT_SDRAM <= doutSDRAM;
166       DIN_SRAM <= dinSRAM;
167       DOUT_SRAM <= doutSRAM;
168       O_RDY <= trig;
169       O_CS <= cs;
170       WEN_CACHE <= wenCache;
```

```
158     begin
159         ADDR_CACHE <= addrCache;
160         ADDR_SDRAM <= addrSDRAM;
161         ADDR_SRAM <= addrSRAM;
162         DIN_CPU <= dinCPU;
163         DOUT_CPU <= doutCPU;
164         DIN_SDRAM <= dinSDRAM;
165         DOUT_SDRAM <= doutSDRAM;
166         DIN_SRAM <= dinSRAM;
167         DOUT_SRAM <= doutSRAM;
168         O_RDY <= trig;
169         O_CS <= cs;
170         WEN_CACHE <= wenCache;
171         MUX_IN <= muxIn;
172         MUX_OUT <= muxOut;
173         WEN_SRAM <= wenSRAM(0);
174         WEN_SDRAM <= wenSDRAM;
175         MEM_STRB <= memstrb;
176     end process;
177 end Behavioral;
```