Ryerson University

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

Course Title:	Digital Systems Engineering
Course Number:	COE758
Semester/Year (e.g.F2016)	F2023

Instructor:	Dr. Geurkov

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

Submission Date:	2023-10-21
Due Date:	2023-10-25

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Zelenovic	Danilo	501032542	11	DZ

*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

Abstract

The goal of this design project is to design and create an SDRAM controller as well as a cache memory system, using a cache controller and an SRAM cache. The CPU is to issue 16-bit address words (may be less if not possible), to the cache controller, and it is to act accordingly. The cache controller must be capable of performing 4 cases:

1. Write a word to the cache (hit)
2. Read a word from the cache (hit)
3. Read or write from or to the cache (miss) with the dirty bit set to 0
4. Read or write from or to the cache (miss) with the dirty bit set as 1

The proper functionality can be tested with the simulated waveforms in order to ensure the project is functional.

Introduction

In nearly all modern computers, there are multiple different memories, all of which have a purpose of storing, reading, and processing data and information, however, each level has a different purpose or usage within the system. As an example, RAM provides very fast read/write but this memory is lost or reset when the system is turned off. ROM is also a form of memory; however, it is non-volatile and stores read-only data which is permanently stored. Withing RAM, there are many categories such as SRAM (static RAM), DRAM (dynamic RAM), SDRAM (synchronized DRAM), and more. These memories, as well as others such as main memory, secondary storage are placed into a memory hierarchy. As a general rule, the closer the memory is to the CPU, the faster the access speed, but smaller the size of available storage, and more expensive it is.

System Specifications

There are 4 behavioral cases we must implemented in the cache controller, these being:

1. When the cache controller receives a request to write from the CPU, a cache hit is detected. In this situation, the index and offset data provided by the CPU is then moved over to the SRAM as the address where the data will be written. Next, the multiplexer writes data to the SRAM received from the CPU only if/when the write bit is enabled, otherwise it is not able to do so.

2. The second case is the cache receiving a read request from the CPU. Here, a cache hit should be happening. The index and offset are transmitted to SRAM and the data read from that address is then sent back to the CPU.
3. The third case occurs when a request is received from the CPU, but the requested block is not found in the cache, as well as the dirty bit being set to 0. The block must be fetched from memory and is placed in the cache. The valid bit is then set to 1 as the tag register is replaced. Then, the request can operate without issues.
4. The final case occurs when a situation similar to the previous case occurs, but the dirty bit is set to 1 instead. In this case, the appropriate block is required to be written to the SDRAM before the system continues reading from the main memory.

Device Design

a. Symbols

CPU:

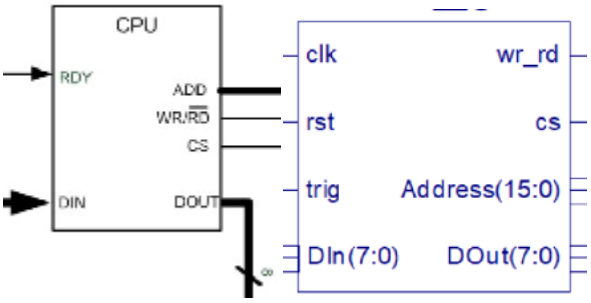

Figure 1: CPU Symbol

Cache Controller:

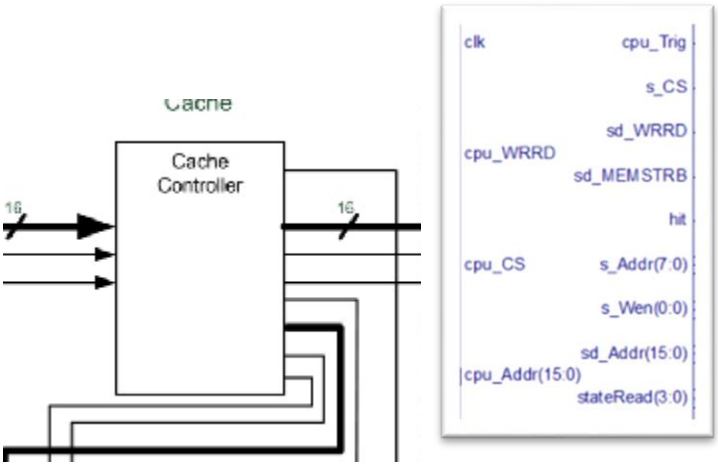

Figure 2: Cache controller symbol

SDRAM Controller:

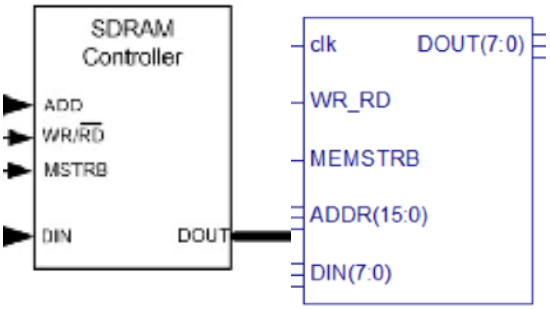

Figure 3: SDRAM controller symbol

Cache SRAM:

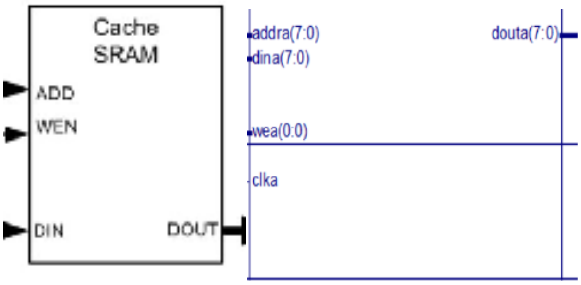

Figure 4: Cache SRAM symbol

2-to-1 Multiplexer:

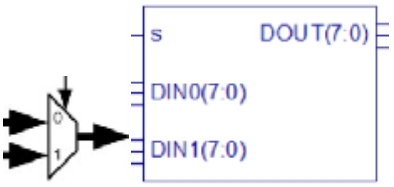

Figure 5: 2-to-1 multiplexer symbol

1-to-2 Demultiplexer:

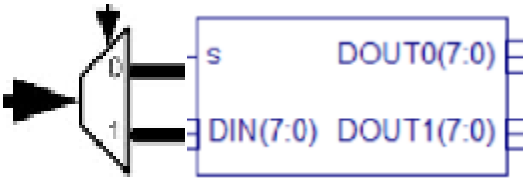

Figure 6: 1-to-2 demultiplexer symbol

The **figures** above **(1-6)** illustrate the individual components used in the block diagram used to create the entire cache controller. Each symbol was created by stating components in each file. After each component was verified, the block was created using “create schematic symbol”. After this part, we opened a schematic file to link each symbol together with wiring between the appropriate inputs and outputs. This can be seen in the block diagram **Figure 7 and 8**.

b. Block diagrams

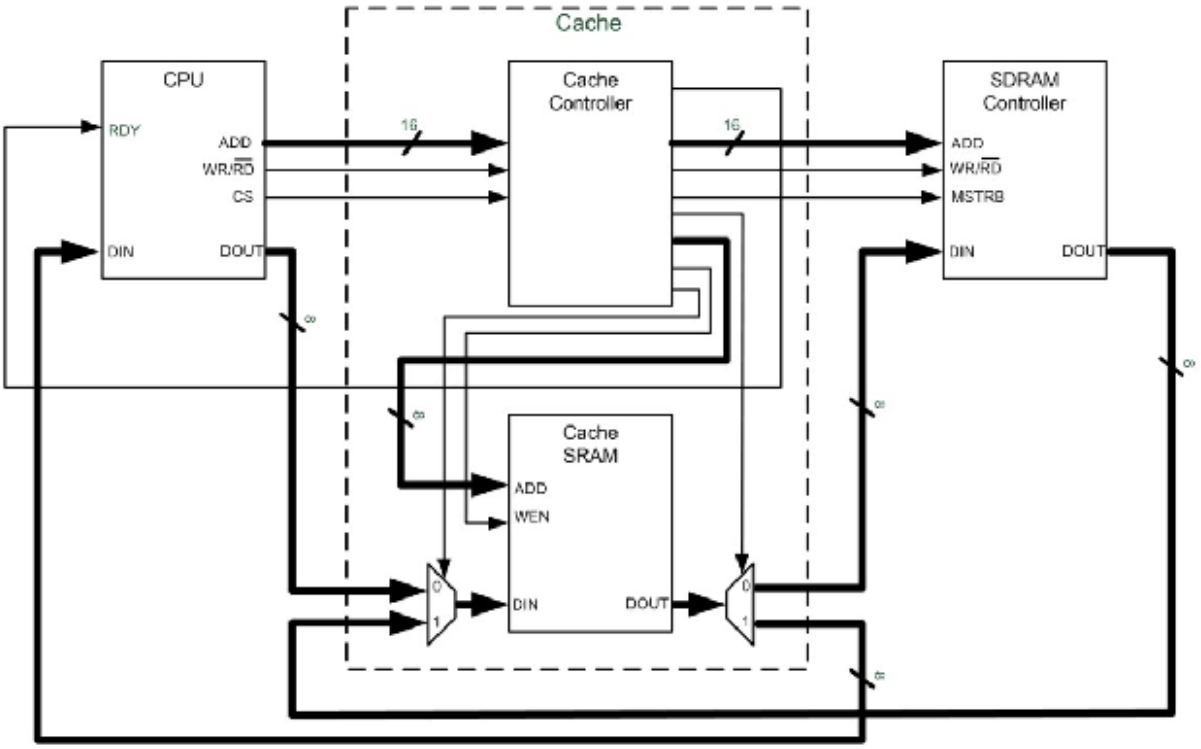

Figure 7: Complete cache system

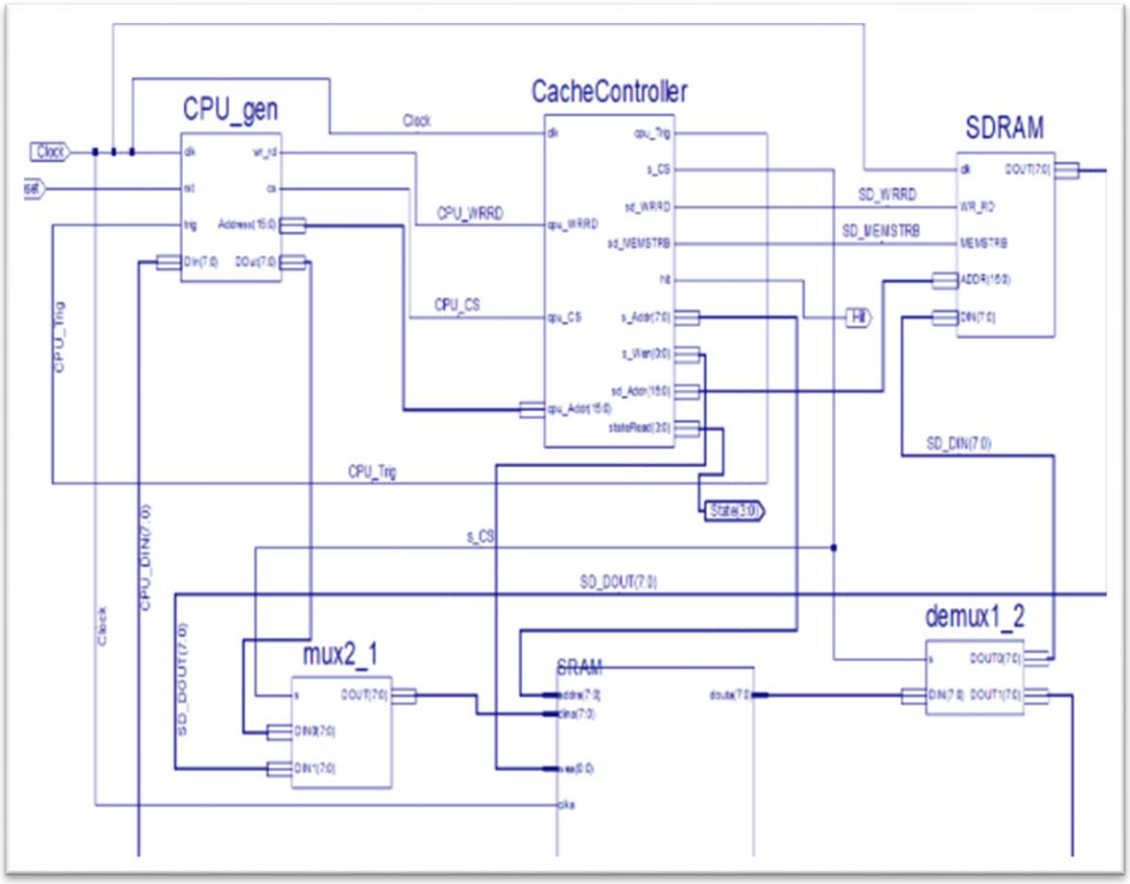

Figure 8: Block Diagram

Figure 8 shows the full block diagram of connections for the cache controller with each component wired, as per the manual instructions shown in **Figure 7.** The thin wires have only 1 bit connection, while thicker wires represent multiple bits to be transmitted. The main inputs are clock and reset. Clock signal is used to synchronize the CPU, cache controller, SDRAM and SRAM. The CPU transmits an address to cache controller. The cache controller then must read or write to SRAM, which depends on statuses in cache controller. Along with this process, the CPU will also transmit an address to the SRAM using the 2-to-1 multiplexer with the selector bit coming from the cache controller itself. The multiplexer and demultiplexer are used to transmit data to CPU, data from the SRAM and comparing values in order to determine whether a cache hit or miss has occurred. The SRAM which is connected to the (de)multiplexers is the cache's memory component which stores data. This component receives signals from all other components. The SDRAM's job is to evaluate data transmitted by the cache controller and also taking input from the clock and demultiplexer.

c. State diagrams

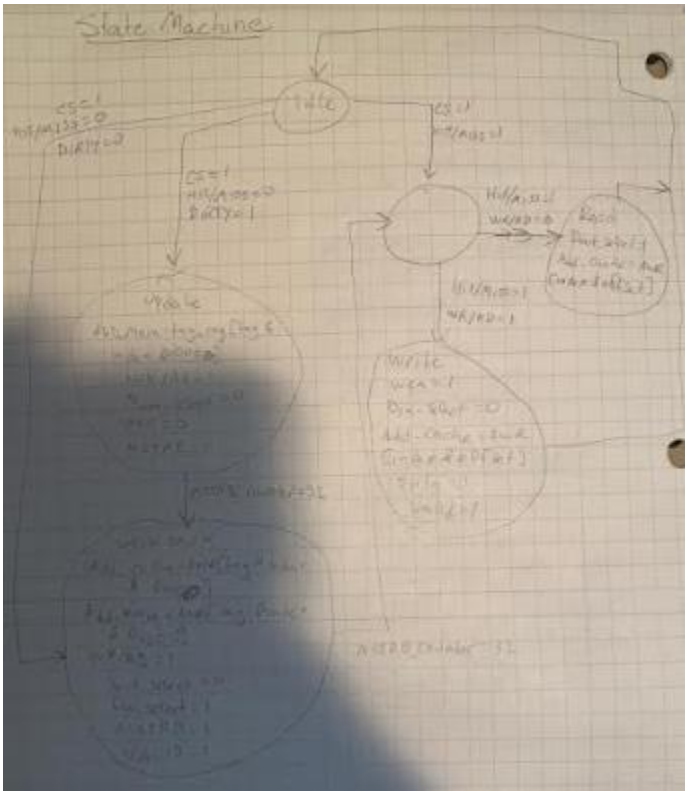

Figure 9: State diagram

The figure above indicated a finite state machine (FSM) used by our cache controller. First, the controller must fetch an address from the CPU. The CPU then issues the read/write enable bits as well as the control signals required such as addresses to be fetched by the cache controller. Then, the cache controller will search the SRAM and see if there is data in that specified location (hit) or if there is not any (miss). If a hit is detected by the cache controller, it will then begin to read/write the data from SRAM and send that information to the CPU. Afterwards, the signal will be switched to indicate that the cache controller is in an idle state, and it is awaiting further instructions from the CPU. Once it receives instructions, it will return to the first state of checking whether this value is in the cache or not. If a miss is detected by the cache controller, the cache controller moves to a state that reflects this, depending on the value of the dirty bit. If the dirty bit is 1, it will enter state where it writes the block to SDRAM, then the controller will switch states to read/write to/from SRAM. However, if the miss was detected, and dirty bit is 0, the cache controller will skip the write operation and will read directly to SDRAM. Finally, it will switch states after retrieving data from SDRAM in order to store it in SRAM cache memory. It will then wait for another instruction from the CPU.

d. Process diagrams

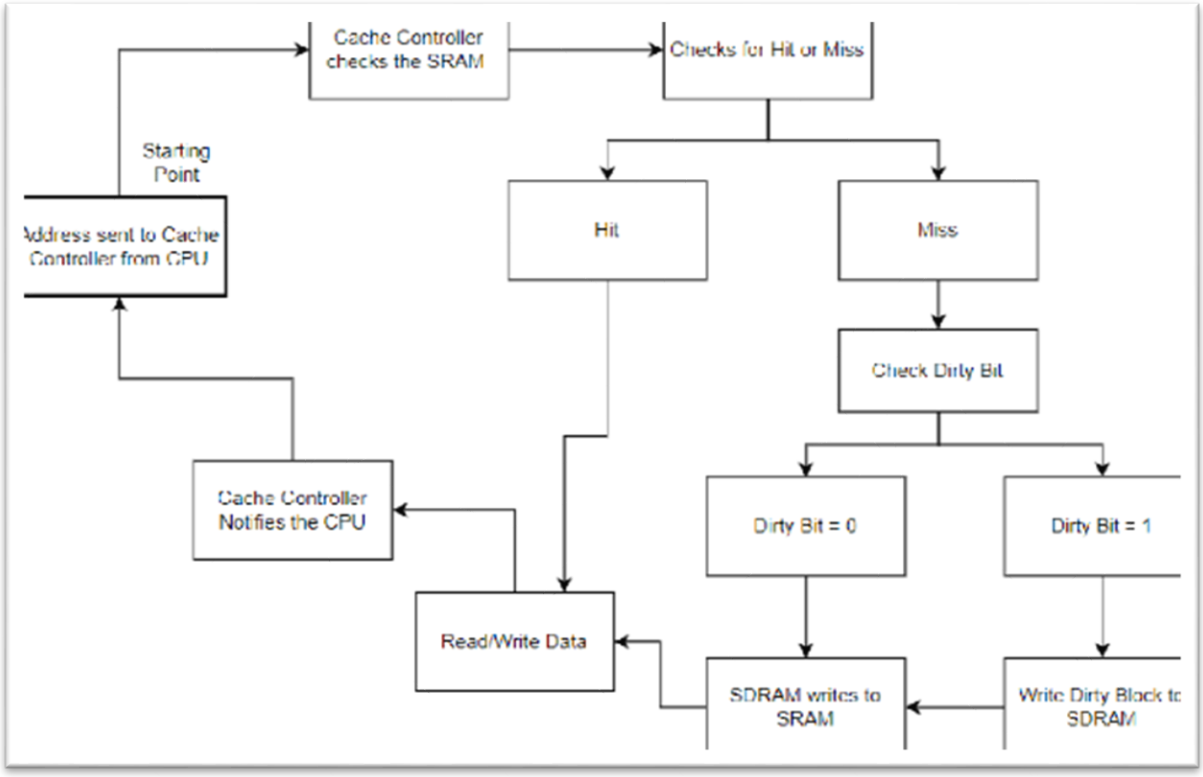

Figure 10: Process diagram

Figure 10 above illustrates the process diagram of our cache controller. The process starts when the CPU sends instructions as well as an address to the cache controller. Next, the cache controller will check the SRAM memory to determine if there is data in that location of the cache (hit or miss). If a hit is detected, the controller is able to read/write to/from the SRAM. But, if it a miss, the controller must read the dirty bit first, in order to determine its next step. If the dirty bit is a 0, the SDRAM will write its information at that address to the SRAM and the cache controller is able to continue with the original request. The controller then lets the CPU know that it is awaiting further instructions. However, if the dirty bit is a 1, then the cache controller will copy the block back to the SDRAM before the new information is given to SRAM.

Results

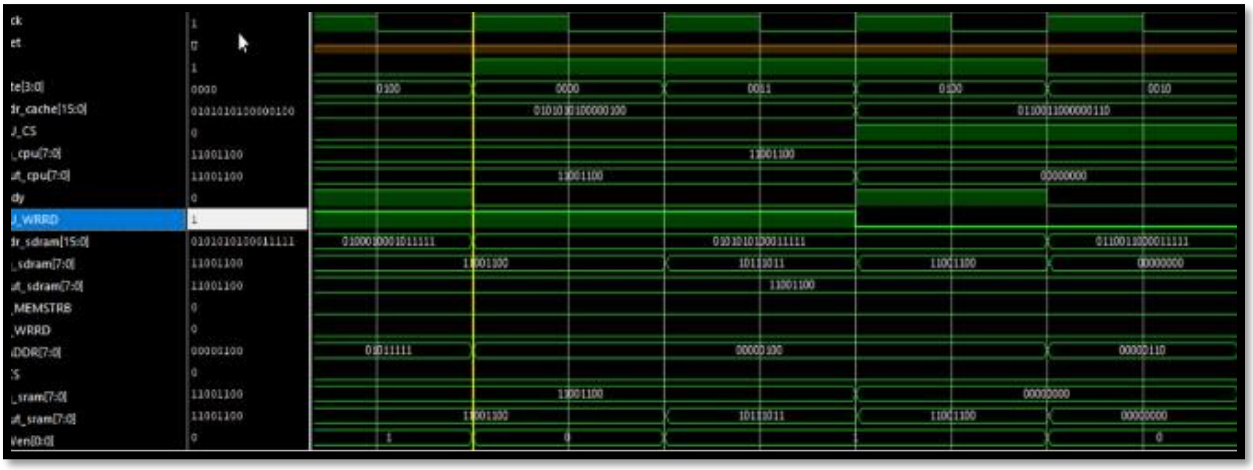

Figure 11: Functional simulation results

Figure 11 is a demonstration of the simulation depicted. At the 630 ns mark, we see the state switch, where instructions are fetched from CPU. The chip select signal is at 0 and the read/write signal is high, meaning a write operation is to be executed. Hit is a 1, meaning that a hit has been detected and a state change is done. Because of this, we see at 640 ns that Dout_SRAM is made to be same as Din_CPU, which works as required as the read operation to CPU was done properly. The controller goes back to next state and sits idly waiting for next instructions form CPU. It then repeats a similar process.

Conclusion

It appears that the cache controller designed was implemented as should be. Thanks to this project, greater knowledge, and insight into the workings of a cache controller and the cache system was developed. The importance was also showcased greatly. Why dirty bits and valid bits are used and why they increase optimization of data flow and control was also showcased during this project. This project greatly showed how memory controllers can affect time and efficiency of programs and information to be shared with other components.

Appendix/References

References:

https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5322259/View

https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5323145/View

https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5322257/View

https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5322256/View

https://courses.torontomu.ca/d2l/le/content/792271/viewContent/5325205/View

Sources:

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity CacheController is
    Port (
        cpu_Addr : in STD_LOGIC_VECTOR(15 downto 0);
        clk : in STD_LOGIC;
        cpu_WRRD : in STD_LOGIC;
        cpu_CS : in STD_LOGIC;
        cpu_Trig : 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;

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

        stateRead : out STD_LOGIC_VECTOR(3 downto 0);
        );
end CacheController;

architecture Behavior of CacheController is
--CPU Signals
    signal tag : STD_LOGIC_VECTOR(7 downto 0);
    signal index : STD_LOGIC_VECTOR(2 downto 0);
    signal offset : STD_LOGIC_VECTOR(4 downto 0);

--Dirty Bit Signal
    signal dBit : STD_LOGIC_VECTOR(7 downto 0):= "00000000";

-- Valid Bit Signal
    signal vBit : STD_LOGIC_VECTOR(7 downto 0):= "00000000";

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

-- SDRAM Signals
    signal counter : integer := 0;
    signal sd_Offset : integer := 0;
```

```
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";
```

```
 else
    state_current <= state1;
    state <= "0001";
    stateRead <= "0001";
 end if;
 end if;
ate 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";
ate 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) <=
```

```
        STD_LOGIC_VECTOR(to_unsigned(sd_Offset, offset'length));
        s_Wen <= "1";
        sd_Offset <= sd_Offset + 1;
     end if;
     counter <= counter + 1;
     end if;
  State 2
     elsif(state_current = state2) then

     if (counter = 64) then
        counter <= 0;
        dBit(to_integer(unsigned(index))) <= '0';
        sd_Offset <= 0;
        state_current <= state1;
        state <= "0001";
        stateRead <= "0001";
     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 <= '1';
        s_Addr(7 downto 5) <= index;
        s_Addr(4 downto 0) <=
        STD_LOGIC_VECTOR(to_unsigned(sd_Offset, offset'length));
        s_Wen <= "0";
        sd_MEMSTRB <= '1';
        sd_Offset <= sd_Offset + 1;
     end if;
     counter <= counter + 1;
     end if;
  State 3
     elsif(state_current = state3) then
     cpu_Trig <= '1';
     state_current <= state4;
     state <= "0100";
     stateRead <= "0100";
     end if;
     end if;
  d process;
```

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity SDRAM is
   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 SDRAM;
architecture Behavior of SDRAM is
   -- SDRAM Array
   type sdmemory is array (7 downto 0, 31 downto 0) of std_logic_vector(7 downto 0);
   signal sd_SIG: sdmemory;
   signal initialized : integer := 0;
   begin
      process (clk)
   begin
      if (clk'event AND clk = '1') then
      if (initialized = 0) then
         for I in 0 to 7 loop
            for J in 0 to 31 loop
               sd_SIG(i,j) <= "11110000";
            end loop;
         end loop;
      initialized <= 1;
      end if;
      if (MEMSTRB = '1') then
      if (WR_RD = '1') then
         sd_SIG(to_integer(unsigned(ADDR(7 downto 5))),to_integer(unsigned(ADDR(4 downto 0)))) <= DIN;
      else
         DOUT <= sd_SIG(to_integer(unsigned(ADDR(7 downto 5))),to_integer(unsigned(ADDR(4 downto 0))));
      end if;
      end if;
      end if;
end process;
end Behavior;
```

```
-----, -----
se IEEE.STD_LOGIC_1164.ALL;

ntity 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)
  );
nd topLevel;

rchitecture 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;
```

```
      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);
```

```
SRAM_1: SRAM port map(
   clka => CLOCK,
   addra => addrSRAM,
   wea => wenSRAM,
   dina => dinSRAM,
   douta => doutSRAM
);


CacheController_1: CacheController port map(
   CLK => CLOCK,
   cpu_Addr => addrCache,
   cpu_WRRD => wenCache,
   cpu_CS => cs,

   sd_Addr => addrSDRAM,
   sd_WRRD => wenSDRAM,
   sd_MEMSTRB => memstrb,

   s_Addr => addrSRAM,
   s_Wen(0) => wenSRAM(0),
   s_CS => muxIn,

   cpu_Trig => rdy
);

sdramC: sdram port map(
   CLK => CLOCK,
   ADDR => addrSDRAM,
   WR_RD => wenSDRAM,
   MEMSTRB => memstrb,
   DIN => dinSDRAM,
   DOUT => doutSDRAM
);

CPU: CPU_gen port map(
   clk => CLOCK,
   rst => RESET,
   trig => trig,
   DIn => dinCPU,
```

```
    Address => addrCache,
    wr_rd => wenCache,
    cs => cs,
    DOut => doutCPU
 );

 trigMux: process(START, rdy)
 begin
    if(START = '1') then
       trig <= '1';
    else
       trig <= rdy;
    end if;
 end process;

 dataInputSRAM: process(muxIn, doutCPU, doutSRAM)
 begin
    if(muxIn = '0') then
       dinSRAM <= doutCPU;
    else
       dinSRAM <= doutSDRAM;
    end if;
 end process;
```

```
dataOutputSRAM: process(muxOut, doutSRAM)
begin
   if(muxOut ='0') then
      dinSDRAM <= doutSRAM;
   else
      dinCPU <= doutSRAM;
   end if;
end process;

debug: process(addrCache, addrSDRAM, addrSRAM, dinCPU, doutCPU, dinSDRAM, doutSDRAM,
begin
   ADDR_CACHE <= addrCache;
   ADDR_SDRAM <= addrSDRAM;
   ADDR_SRAM <= addrSRAM;
   DIN_CPU <= dinCPU;
   DOUT_CPU <= doutCPU;
   DIN_SDRAM <= dinSDRAM;
   DOUT_SDRAM <= doutSDRAM;
   DIN_SRAM <= dinSRAM;
   DOUT_SRAM <= doutSRAM;
   O_RDY <= trig;
   O_CS <= cs;
   WEN_CACHE <= wenCache;
```

```
begin
   ADDR_CACHE <= addrCache;
   ADDR_SDRAM <= addrSDRAM;
   ADDR_SRAM <= addrSRAM;
   DIN_CPU <= dinCPU;
   DOUT_CPU <= doutCPU;
   DIN_SDRAM <= dinSDRAM;
   DOUT_SDRAM <= doutSDRAM;
   DIN_SRAM <= dinSRAM;
   DOUT_SRAM <= doutSRAM;
   O_RDY <= trig;
   O_CS <= cs;
   WEN_CACHE <= wenCache;
   MUX_IN <= muxIn;
   MUX_OUT <= muxOut;
   WEN_SRAM <= wenSRAM(0);
   WEN_SDRAM <= wenSDRAM;
   MEM_STRB <= memstrb;
end process;
i Behavioral;
```

```
COMPONENT CPU_gen
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;

Inst_CPU_gen: CPU_gen PORT MAP(
        clk => ,
        rst => ,
        trig => ,
        Address => ,
        wr_rd => ,
        cs => ,
        DOut =>
);
```

```
    Port ( clk : in  STD_LOGIC;
           WR_RD : in  STD_LOGIC;
           CS : in  STD_LOGIC;
           CPU_ADD : in  STD_LOGIC_VECTOR (15 downto 0);
                   SDRAM_ADD : out STD_LOGIC_VECTOR(15 DOWNTO 0);
                   CACHE_ADD : out STD_LOGIC_VECTOR(7 DOWNTO 0);
           CACHE_WEN : out  STD_LOGIC;
           CACHE_DIN_WEN : out  STD_LOGIC;
           CACHE_DOUT_WEN : out  STD_LOGIC;
           WEN_SDRAM : out  STD_LOGIC;
           MEMSTRB : out  STD_LOGIC;
           RDY : out  STD_LOGIC;
                   DEBUG : out STD_LOGIC_VECTOR(31 DOWNTO 0));
end CacheControllerFSM;

architecture Behavioral of CacheControllerFSM is
        -- Array of 8 blocks of dirty and valid bits
        type dirty_bits is array (7 downto 0) of STD_LOGIC;
        signal dbits: dirty_bits := (others => '0');
        type valid_bits is array (7 downto 0) of STD_LOGIC;
        signal vbits: valid_bits := (others => '0');
        -- CPU signals
        signal cpu_tag : STD_LOGIC_VECTOR(7 DOWNTO 0);
        signal cpu_index : STD_LOGIC_VECTOR(2 DOWNTO 0);
        signal cpu_offset : STD_LOGIC_VECTOR(4 DOWNTO 0);
        signal index_and_offset : STD_LOGIC_VECTOR(7 DOWNTO 0);
        -- Tag compare component
        COMPONENT TagCompareDirectMapping
        PORT(
                CPU_ADD : IN std_logic_vector(15 downto 0);
                clk : IN std_logic;
```

```
PORT(
        CPU_ADD : IN std_logic_vector(15 downto 0);
        clk : IN std_logic;
        HIT_MISS : OUT std_logic
        );
END COMPONENT;

begin
        -- Continuous assignment of CPU signals
        cpu_tag <= CPU_ADD(15 DOWNTO 8);
        cpu_index <= CPU_ADD(7 DOWNTO 5);
        cpu_offset <= CPU_ADD(4 DOWNTO 0);
        index_and_offset <= CPU_ADD(7 DOWNTO 0);
        sys_tag_compare: TagCompareDirectMapping PORT MAP(
                CPU_ADD => CPU_ADD,
                clk => clk,
                HIT_MISS => hit_miss_signal
        );
```

```
        begin
                if(clk'Event AND clk='1') then
                        case yfsm is
                                -- Processing s0 - Idle
                                when s0 =>
                                        if(CS = '1') then
                                                yfsm <= s1;
                                                DEBUG(2 DOWNTO 0) <= "001";
```

```
when s1 =>
        if(hit_miss_signal='1' AND WR_RD='1') then
                yfsm <= s2;
                DEBUG(2 DOWNTO 0) <= "010";
        elsif(hit_miss_signal='1' AND WR_RD='0') then
                yfsm <= s3;
                DEBUG(2 DOWNTO 0) <= "011";
        elsif(hit_miss_signal='0' AND dbits(to_integer(unsigned(cpu_index)))='0') then
                yfsm <= s4;
                DEBUG(2 DOWNTO 0) <= "100";
        elsif(hit_miss_signal='0' AND dbits(to_integer(unsigned(cpu_index)))='1') then
                yfsm <= s5;
                DEBUG(2 DOWNTO 0) <= "101";
        end if;
-- Processing s2 - Cache hit and write
when s2 =>
        -- if(cpu_rdy='1') then
```

```
            DEBUG(2 DOWNTO 0) <= "000";
    -- Processing s3 - Cache hit and read
    when s3 =>
            -- if(cpu_rdy='1') then
            --      yfsm <= s0;
            -- else
            --      yfsm <= s3;
            -- end if;
            yfsm <= s0;
            DEBUG(2 DOWNTO 0) <= "000";
    -- Processing s4 - Cache miss and dirty bit=0
    when s4 =>
            if(WR_RD='0') then
                    yfsm <= s3;
                    DEBUG(2 DOWNTO 0) <= "011";
            elsif(WR_RD='1') then
                    yfsm <= s2;
                    DEBUG(2 DOWNTO 0) <= "010";
            -- elsif(memstrb='0') then
            --      yfsm <= s4;
            end if;
```

```
when s4 =>
        if(WR_RD='0') then
                yfsm <= s3;
                DEBUG(2 DOWNTO 0) <= "011";
        elsif(WR_RD='1') then
                yfsm <= s2;
                DEBUG(2 DOWNTO 0) <= "010";
        -- elsif(memstrb='0') then
        --      yfsm <= s4;
        end if;
-- Processing s5 - Cache miss and dirty bit=1
when s5 =>
        -- if(memstrb='1') then
        --      yfsm <= s4;
        -- else
        --      yfsm <= s5;
        -- end if;
        yfsm <= s4;
        DEBUG(2 DOWNTO 0) <= "100";
```

```
end process;
process(yfsm)
begin
        case yfsm is
                -- Generating outputs for s0
                when s0 =>
                        RDY <= '1';
                -- Generating outputs for s1
                when s1 =>
                        RDY <= '0';
                -- Generating outputs for s2
                when s2 =>
                        CACHE_ADD <= index_and_offset;
                        CACHE_WEN <= '1';
                        CACHE_DIN_WEN <= '0';
                        dbits(to_integer(unsigned(cpu_index))) <= '1'; --dirty bit
                        vbits(to_integer(unsigned(cpu_index))) <= '1'; --valid bit
                        -- RDY <= '1'; -- This should happen when going back to s0
                -- Generating outputs for s3
                when s3 =>
                        CACHE_ADD <= index_and_offset;
                        CACHE_WEN <= '0';
                        CACHE_DOUT_WEN <= '1';
                        -- RDY <= '1'; -- This should happen when going back to s0
                -- Generating outputs for s4
                when s4 =>
                        SDRAM_ADD <= (cpu_add AND "1111111111100000");
                        WEN_SDRAM <= '0';
                        CACHE_DIN_WEN <= '1';
```

```
                        SDRAM_ADD <= (cpu_add AND "1111111111100000");
                        WEN_SDRAM <= '0';
                        CACHE_DIN_WEN <= '1';
                        CACHE_WEN <= '1';
                        vbits(to_integer(unsigned(cpu_index))) <= '1';
                -- Generating outputs for s5
                when s5 =>
                        SDRAM_ADD <= (cpu_add AND "1111111111100000");
                        WEN_SDRAM <= '1';
                        CACHE_DOUT_WEN <= '0';
                        MEMSTRB <= '1';
        end case;
end process;

DEBUG(3) <= dbits(to_integer(unsigned(cpu_index)));
DEBUG(4) <= vbits(to_integer(unsigned(cpu_index)));

end Behavioral;
```