Abstract

The purpose of this project was to simulate a simple cache controller and attain practical experience in the design and implementation of custom logic controllers and interfacing to SRAM memory units and other logic devices. The functionality and interactions between the cache controller, SRAM and SDRAM controller was designed and then studied. Another goal of the project was 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. The cache controller, SRAM memory and SDRAM controller were all implemented using VHDL while the CPU generation VHDL file was provided.

Introduction

The CPU block was pre-built and provided a series of hard-coded outputs. These outputs included, Address[16](made up of Tag[8], Index[3] and Offset[5]), Data[8], and Write/Read[1]. These outputs are summarized below in Table 1.

Table 1: CPU Output Values in Hexadecimal

Address				
Tag	**Index**	**Offset**	**Data**	**Write/Read**
11	0	00	AA	Write
11	0	02	BB	Write
11	0	00	xx	Read
11	0	02	xx	Read
33	2	06	xx	Read
44	2	04	xx	Read
55	0	04	CC	Write
66	0	06	xx	Read

The purpose of these outputs are to simulate Write and Read commands being sent to the cache controller. The CPU block also output a CS[1] bit which was to indicate to the cache controller that it has completed loading its new output values (from table 1) and they are ready to be processed.

A Bram block was used to act as the SRAM Component.

An SDRAM block was created which simulated the Synchronous Dynamic RAM by storing bits in an array of 6 Blocks of 32 Addresses of 8 bits. This array was initialized with values 0x14, 0x8F, 0x67 sequentially throughout.

The cache controller was to be designed as a Finite State Machine which would be able to receive the signal from the CPU, then determine if the requested memory address's block is already in SRAM (hit) or if it is not and needs to be retrieved from SDRAM. If the block to be replaced in SRAM is 'dirty', that is to say it has had a write operation performed on it, it must first be written back to SDRAM. This cache controller was then evaluated on its performance timings for each type of memory access.

System Specifications

CPU block (Functional Specifications):

1. Simulates Write and Read commands being sent to the cache controller
2. CPU block also output a CS[1] bit which was to indicate to the cache controller that it has completed loading its new output values (from table 1) and they are ready to be processed.
3. Once CS is de-asserted, the CPU block must be ready to repeat the process.

CPU block (Technical Specifications):

1. The CPU is synchronized to the rest of the system via a clock frequency
2. Periodically issues read or write transaction requests
3. When the CPU issues a transaction, it first sets the appropriate address on the address bus and sets the read/write indicator to the correct value
   a) Read : the WR/RD signal is low (0)
   b) Write: the WR/RD signal is high (1), appropriate data is set on the DOUT
4. When all transaction signals are stable, the strobe CS is asserted, and stays asserted for 4 clock cycles

Cache Controller block (Functional Specifications):

1. Receive Write and Read requests from the CPU
2. Determine whether the requested data is currently in SRAM (hit)
3. If necessary, move data from SDRAM to SRAM and vice-versa to maintain data integrity
4. Manage and control block Tag[8], Valid[1] and Dirty[1] bits corresponding to SRAM data
5. Perform read and write operations to SRAM based on CPU requests.

Cache Controller block (Technical Specifications):

1. When CS is asserted by the CPU the Address, Data and WR/RD signals are received by the cache controller from the CPU based on Table 1.
2. The given Address Block's tag and index values are looked for in SRAM.
3. If the block does not exist in SRAM, the dirty bit of the block currently at that index position in SRAM is checked.
4. If the dirty bit is 1, that block is written back to SDRAM; over a period of 64 clock cycles by toggling memstrb every clock cycle and pushing SRAM's Dout to SDRAM's Din. During this process SDRAM's WR/RD (write/read) signal is asserted and an offset value is incremented every 2 clock cycles such that 32 incrementations occur, and 32 sequential address values are passed to both the SDRAM and SRAM.
5. If the dirty bit is 0, or if step 4 has been performed the block is written to SRAM by toggling memstrb every clock cycle for 64 clock cycles and pushing SDRAM's Dout to SRAM's Din. During this process SRAM's Wea (write enable) signal is asserted and an offset value is incremented every 2 clock cycles such that 32 incrementations occur, and 32 sequential address values are passed to both the SDRAM and SRAM.
6. In the case of a write operation, the Wea (write enable) signal of the SRAM is asserted, and Dout, index and offset signals from the CPUare passed to the SRAM Din and Address respectively. The corresponding tag value is updated to match the first 8 bits of the CPU address and the corresponding Valid and Dirty bits are set to '1'
7. In the case of a read operation, the Wea (write enable) signal of the SRAM is set to 0, and Dout, index and offset signals from the CPU are passed to the SRAM Din and Address respectively. The corresponding tag value is updated to match the first 8 bits of the CPU address and the corresponding Valid bit is set to '1'
8. The ready signal is then asserted and sent to the CPU
9. The Cache Controller then remains in a ready state waiting for the CS strobe to be re-asserted by the CPU

Cache SRAM block (Functional Specifications):

1. Acts as local memory for the cache controller

Cache SRAM block (Technical Specifications):

1. All operations issued to the cache controller are synchronized on the rising edge of the clock
2. Write Operation:
   a) Correct address is set on the Address bus by the Cache Controller
   b) WEN signal is asserted (1)
   c) On subsequent rising edge, data is written
3. Read Operation:
   a) Correct address is set on the Address bus by the Cache Controller
   b) addressed data will propagate to the output DOUT after the next rising edge of the clock

SDRAM Controller block (Functional Specifications):

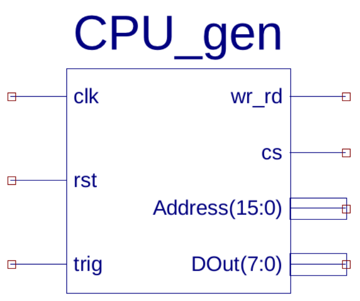
1. Responds to block read and write requests from the Cache Controller
2. Data can be written from the cache to the SDRAM through the Din bus when the WR/RD signal is asserted
3. Data can be read from the SDRAM to the cache through the Dout bus when the WR/RD signal is de-asserted

SDRAM Controller block (Technical Specifications):

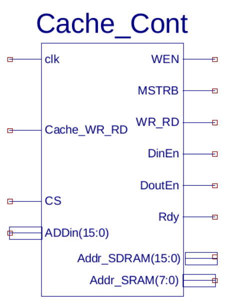
1. Synchronized to Cache and the rest of the system via a clock frequency
2. Write Operation:
   a) Correct base address is set on the ADD port by the Cache Controller
   b) WR/RD signal is asserted (1)
   c) once all other signals are stable (one clock cycle later), the strobe MEMSTRB is asserted for one clock cycle
   d) Process repeated 32 times for 1 full block
3. Read Operation:
   a) Correct base address is set on the ADD port by the Cache Controller
   b) WR/RD signal is de-asserted (0)
   c) once all other signals are stable (one clock cycle later), the strobe MEMSTRB is asserted for one clock cycle
   d) Process repeated 32 times for 1 full block

Device Descriptions

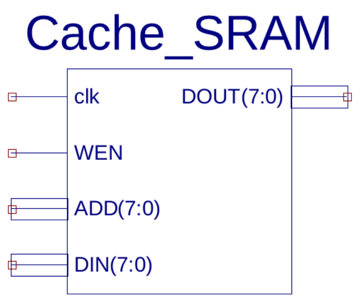
Symbols



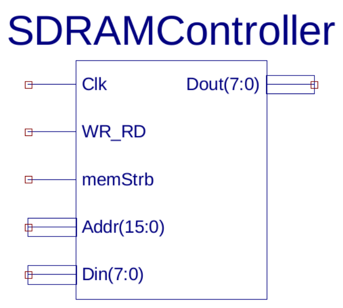
(a) CPU Generator Symbol



(b) Cache Controller Symbol



(c) Cache Static RAM Symbol



(d) Synchronous Dynamic RAM Symbol

Figure 1: Symbol Diagrams for Cache Controller Project

Block Diagrams

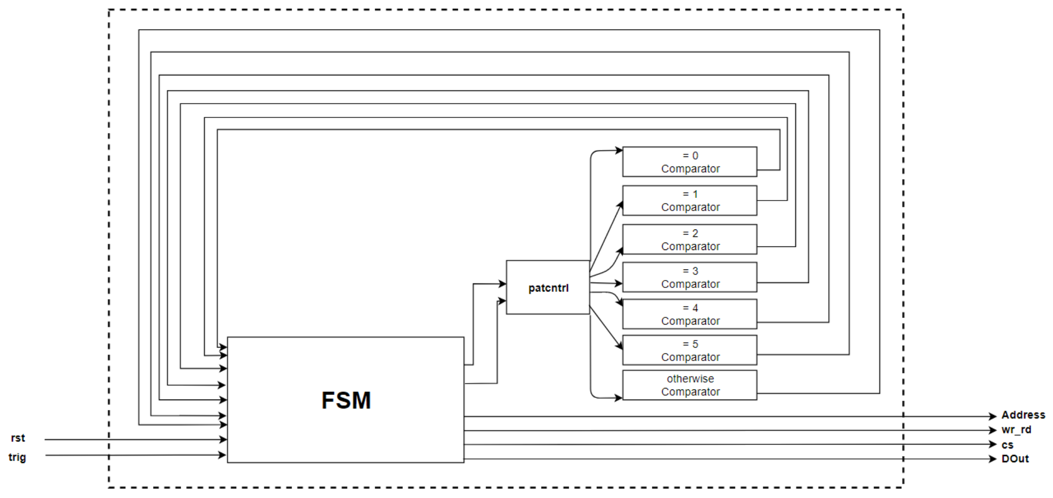


Figure 2: CPU Block Diagram

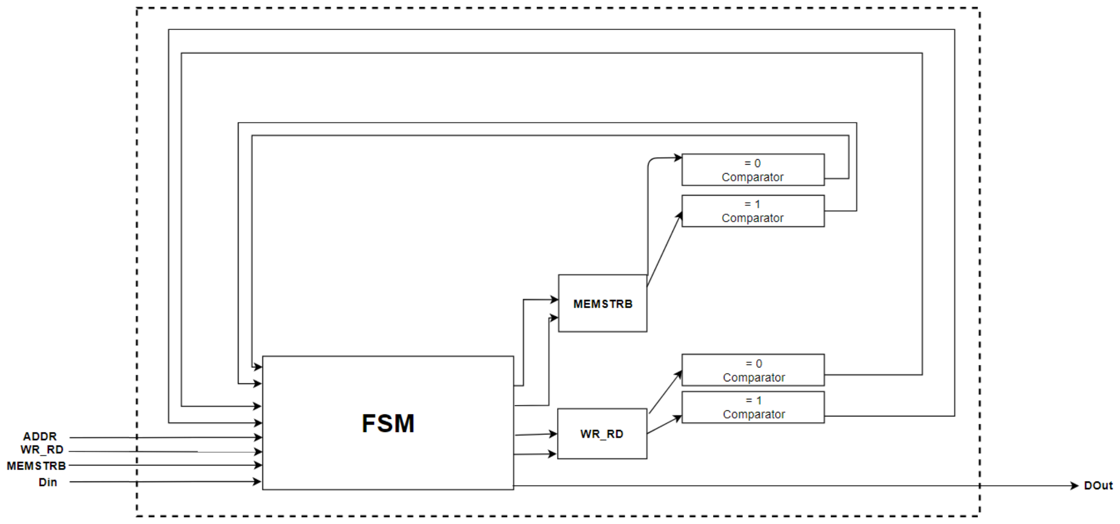


Figure 3: SDRAM Block Diagram

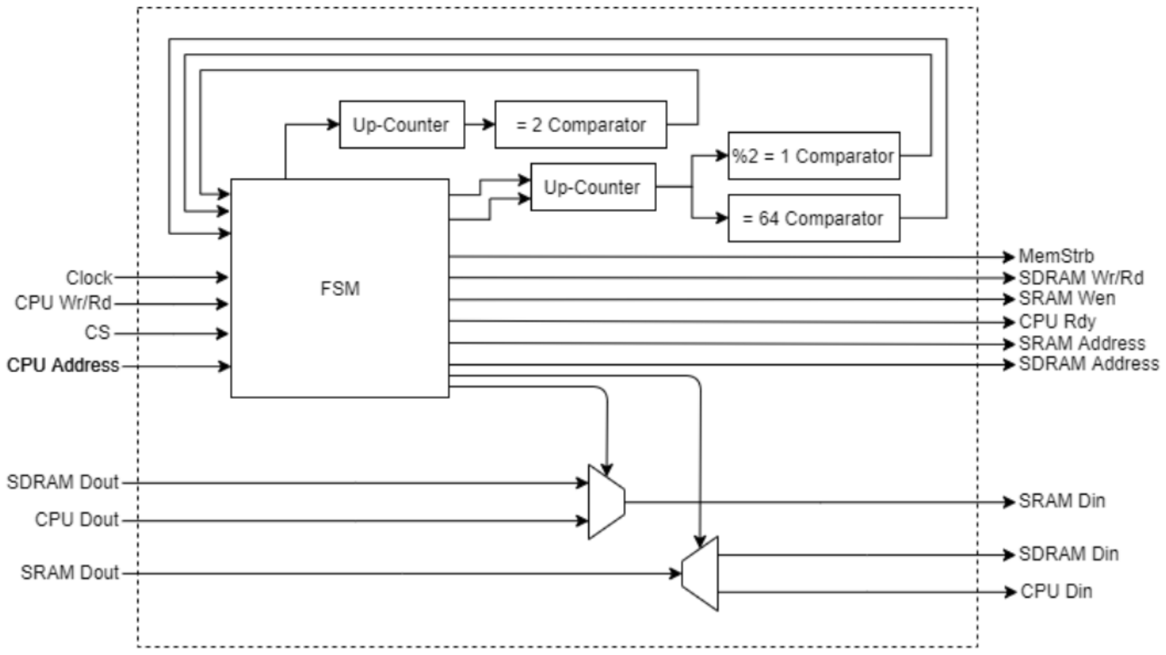


Figure 4: Cache Controller Block Diagram

State Diagrams

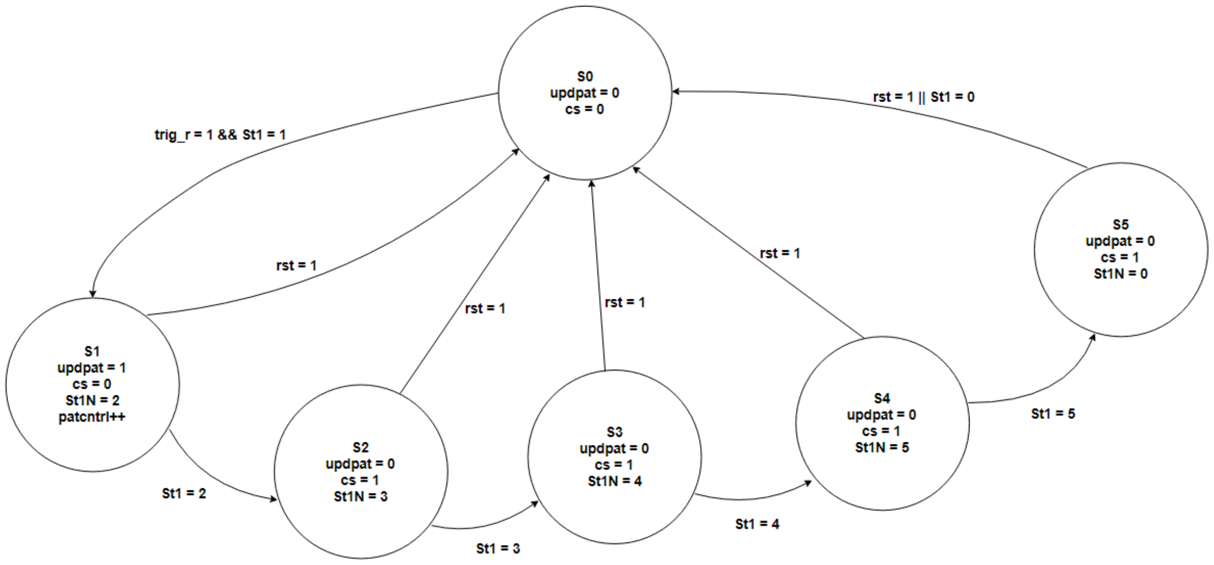


Figure 5: CPU State Diagram

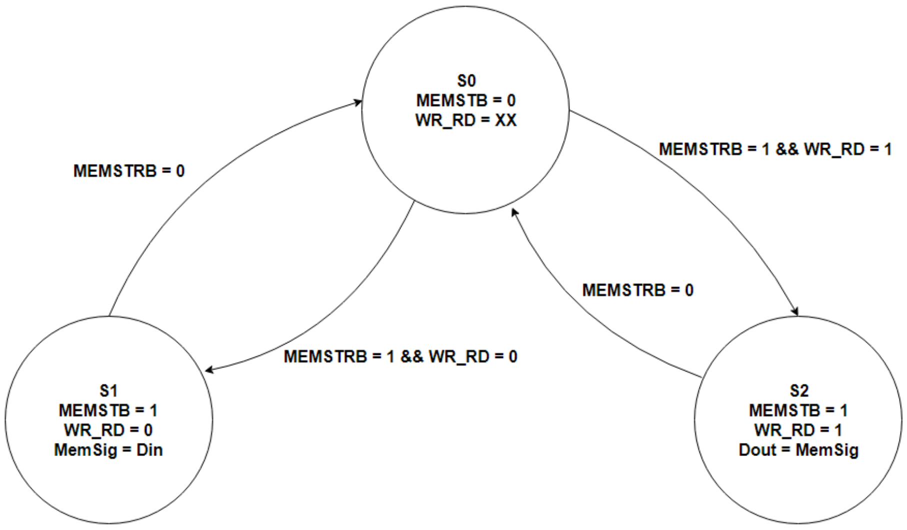


Figure 6: SDRAM State Diagram

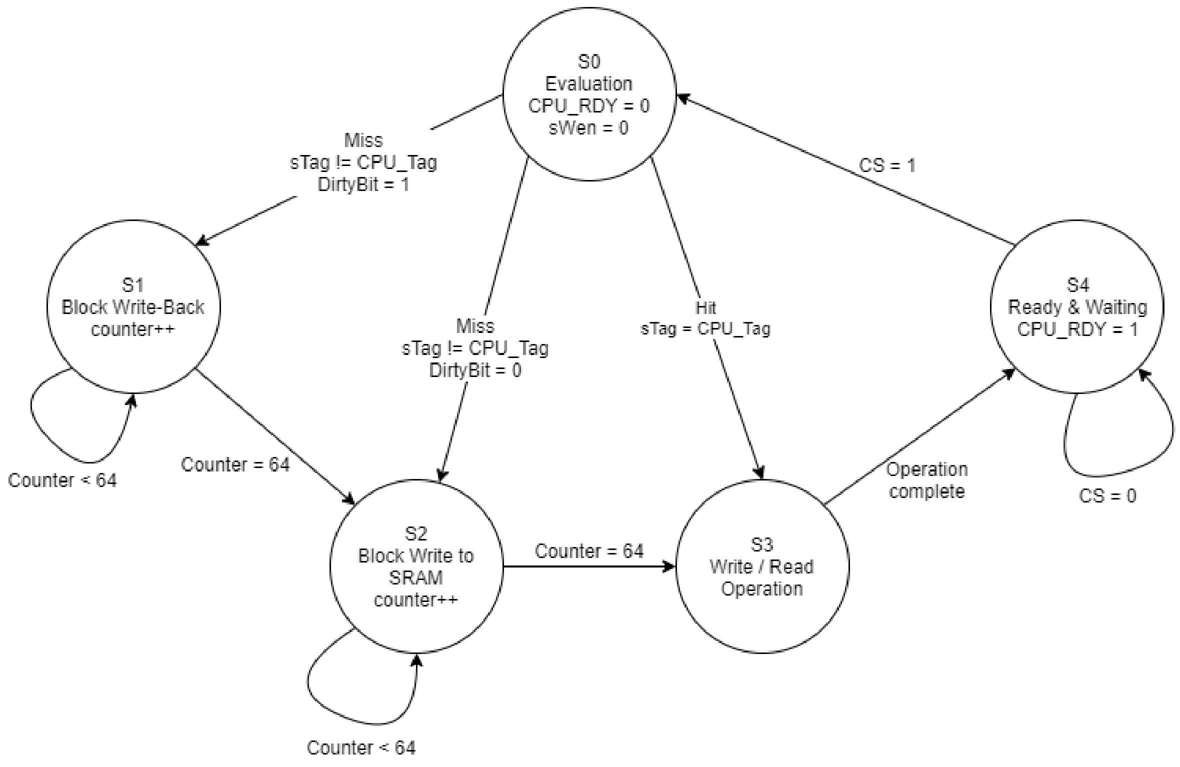


Figure 7: Complete Cache Controller State Diagram

Table 2: Complete Cache State Operation Table

State	Operation
0	Evaluation of CPU ADDR
1	Write block to SDRAM
2	Read block from SDRAM
3	Write/Read To/From SRAM From/To CPU
4	Ready and waiting

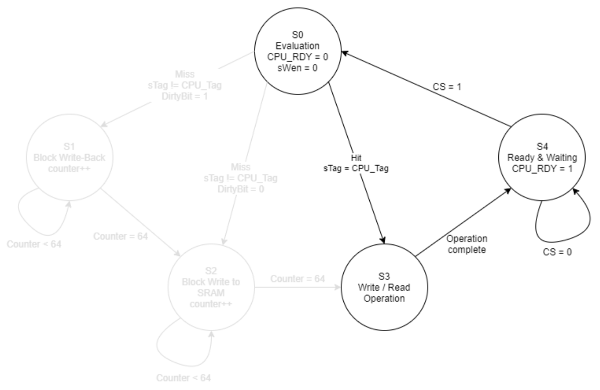


Figure 8: Case 1 and 2

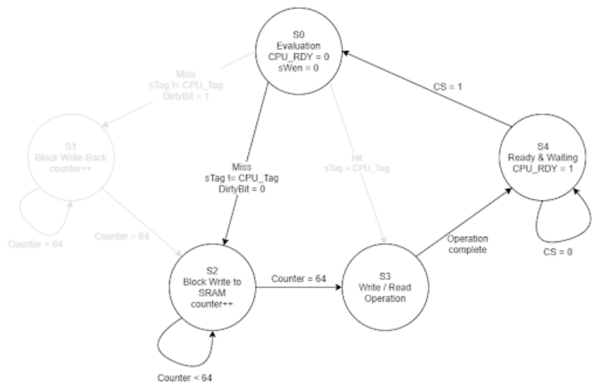


Figure 9: Case 3

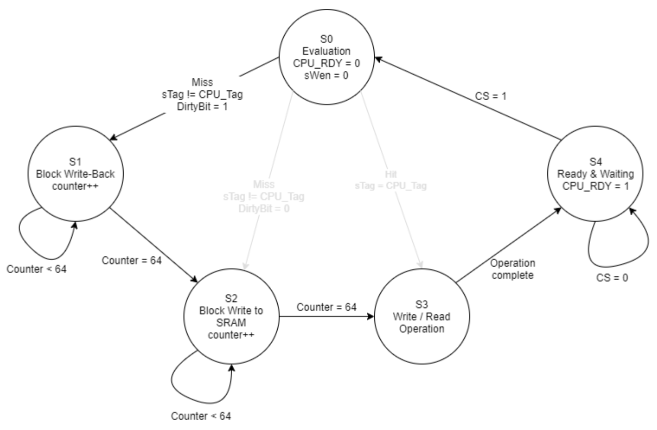
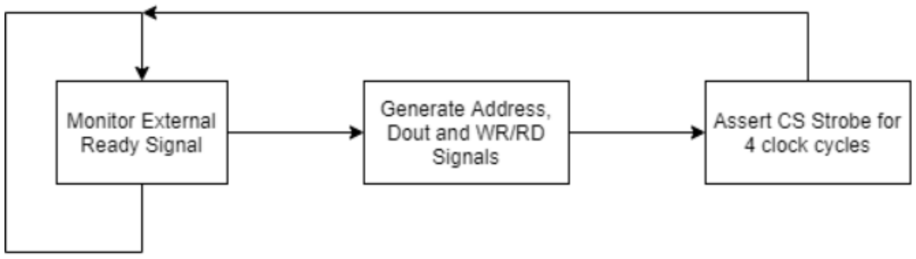


Figure 10: Case 4

Table 3: Complete Cache Behavior Table

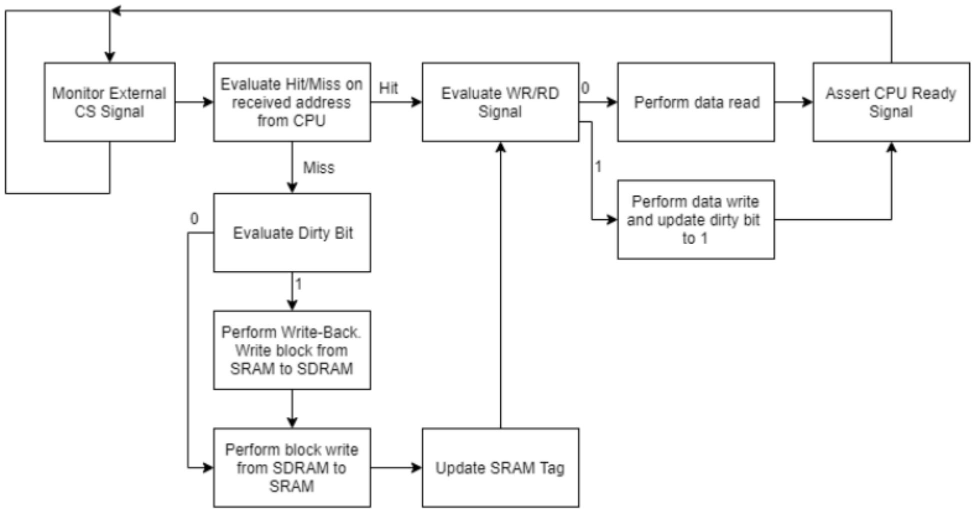
Case	**Behavior**	**State Progression**
1	Write a word to cache [hit]	0 - 3 - 4 - 0
2	Read a word from cache [hit]	0 - 3 - 4 - 0
3	Read/Write from/to cache [miss] and dirty bit = 0	0 - 2 - 3 - 4 - 0
4	Read/Write from/to cache [miss] and dirty bit = 1	0 - 1 - 2 - 3 - 4 - 0

Process Diagram



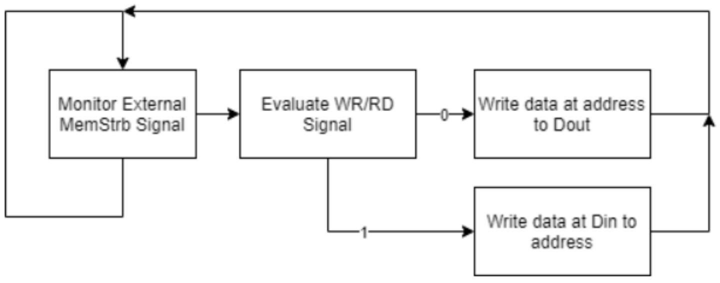
Monitor External Ready Signal
Generate Address, Dout and WR/RD Signals
Assert CS Strobe for 4 clock cycles

Figure 11: CPU Process Diagram



Monitor External CS Signal
Evaluate Hit/Miss on received address from CPU
Hit
Evaluate WR/RD Signal
0
Perform data read
Assert CPU Ready Signal
Miss
1
Perform data write and update dirty bit to 1
0
Evaluate Dirty Bit
1
Perform Write-Back. Write block from SRAM to SDRAM
Perform block write from SDRAM to SRAM
Update SRAM Tag

Figure 12: Cache Controller Process Diagram



Monitor External MemStrb Signal
Evaluate WR/RD Signal
0
Write data at address to Dout
1
Write data at Din to address

Figure 13: SDRAM Process Diagram

Results

Timing Diagrams

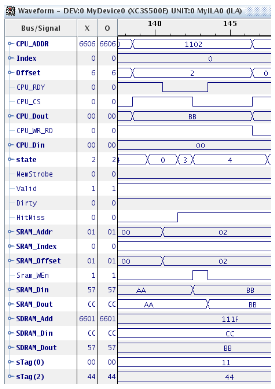


Figure 14: Write a word to cache [hit]

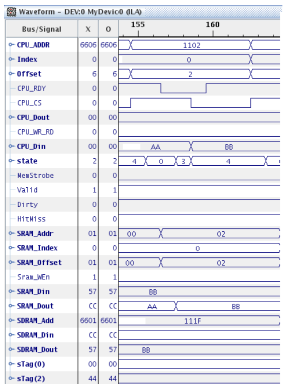


Figure 15: Read a word from cache [hit]

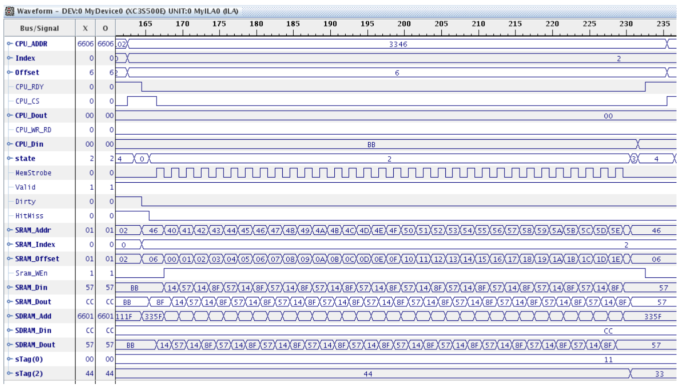


Figure 16: Read/Write from/to cache [miss] and dirty bit = 0

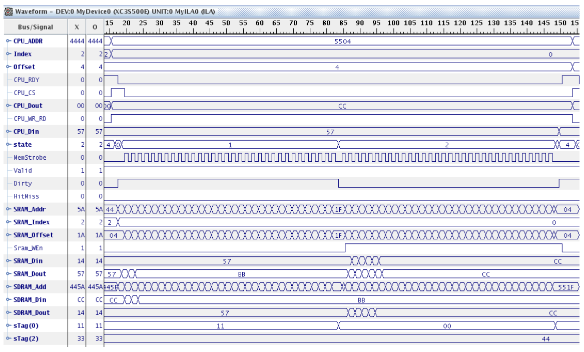


Figure 17: Read/Write from/to cache [miss] and dirty bit = 1 (See Appendix for Detailed)

Cache Parameters

* Assuming Board frequency = 5MHz and therefore 1 Clock Cycle = 20ns

N	Cache performance parameter	Time (ns)	Clock Cycles
1	Hit/Miss Determination Time	40	2
2	Data Access Time	20	1
3	Block Replacement Time	1280	64
4	Hit Time (Case 1 and Case 2)	160	8
5	Miss Penalty for Case 3 (when D-bit = 0)	1300	65
6	Miss Penalty for Case 4 (when D-bit = 1)	2600	130

Table 4: Cache Performance Table

Brief Explanation

Hit/Miss Determination Time = Time spent in state 0. Requires 1 clock cycle to propagate data and a second clock cycle to compare tag and compare valid bit. Seen above in Figure 11.

Data Access Time = Time spent in state 3. Requires 1 clock cycle to perform write or read action. Seen above in Figure 11.

Block replacement time = Time required to write a block to SRAM. Memstrobe pulses 32 times with 1 clock cycle at 1 and 1 clock cycle at 0. This is a total of 54 cycles. Seen above in Figure 13.

Hit Time = Sum of States 0, 3 and 4. Note that in our program we add intentional delay to compensate for the fact that the cpu holds the CS strobe for 4 clock cycles. We could operate state 4 on only 1 clock cycle however then the hit time would be 4 clock cycles only, which means that the CS strobe asserted for the previous CPU data set could still be asserted by the time we returned back to state 4, causing old data to be propagated through the system. The delay we included is not an ideal case and could optimally be reduced. Seen above in Figure 11.

Miss Penalty for Case 3 (when D-bit = 0) = Block replacement time + 1 clock cycle to set new output signals. This is the time spent in State 2. Seen above in Figure 13.

Miss Penalty for Case 4 (when D-bit = 0) = 2 x (Block replacement time + 1 clock cycle to set new output signals). This is the time spent in State 1 + State 2. Seen above in Figure 14.

Conclusion

In this lab, the objective benefits of creating and utilizing a simple cache controller was accomplished. The theoretical operation of the cache controller matched expected and predicted results based on in class material. The only exceptions were when we had to manually delay the program in order to wait for the CS strobe to be set back to zero. We also were unable to accurately extract values from the Xilinx generated BRAM block. When writing data to the BRAM we can see that for each change of address there is a change in data value as shown in Figure 13 and Figure 14 above (in State 2). However when writing from the BRAM block we can see that despite the address value changing for each MemStrobe pulse, the SRAM_Dout value does not change across all values. This can be seen above in Figure 14 (during State 1). To correct this issue we would have used a simple array to store the SRAM data, just as we did for the SDRAM. This would allow us to have greater control over the inner workings of the SRAM block, and not have to rely on the prebuilt BRAM.

Outside of this small issue our cache controller performed as expected, being able to perform all 4 behavioral cases consistently as can be seen in Figures 11, 12, 13 and 14. It clearly demonstrated the large difference in time cost between a cache hit - 8ns total - versus a cache miss - 73ns if clean, 138ns if dirty. In most cache controllers, the hit rate is upwards of 95%, therefore the overall access time of a digital system which utilizes a cache memory will be much less than those that do not. This undoubtedly shows the necessity and utility of using a cache based system versus constantly writing to main memory.

The secondary goals of learning and expanding the knowledge of VHDL within the Xilinx ISE CAD environment and gaining experience in the design and implementation of custom logic controllers was also achieved.

References

1. D. A. Patterson, J. L. Hennessy, and P. Alexander, Computer organization and design: the hardware/ software interface. Amsterdam: Morgan Kaufmann, 2015.

2.[Online]. Available: https://www.ee.ryerson.ca/ lkirisch/ele758/labs/Cache Project[12-09-10].pdf. [Accessed: 03-Nov-2019].

Appendix

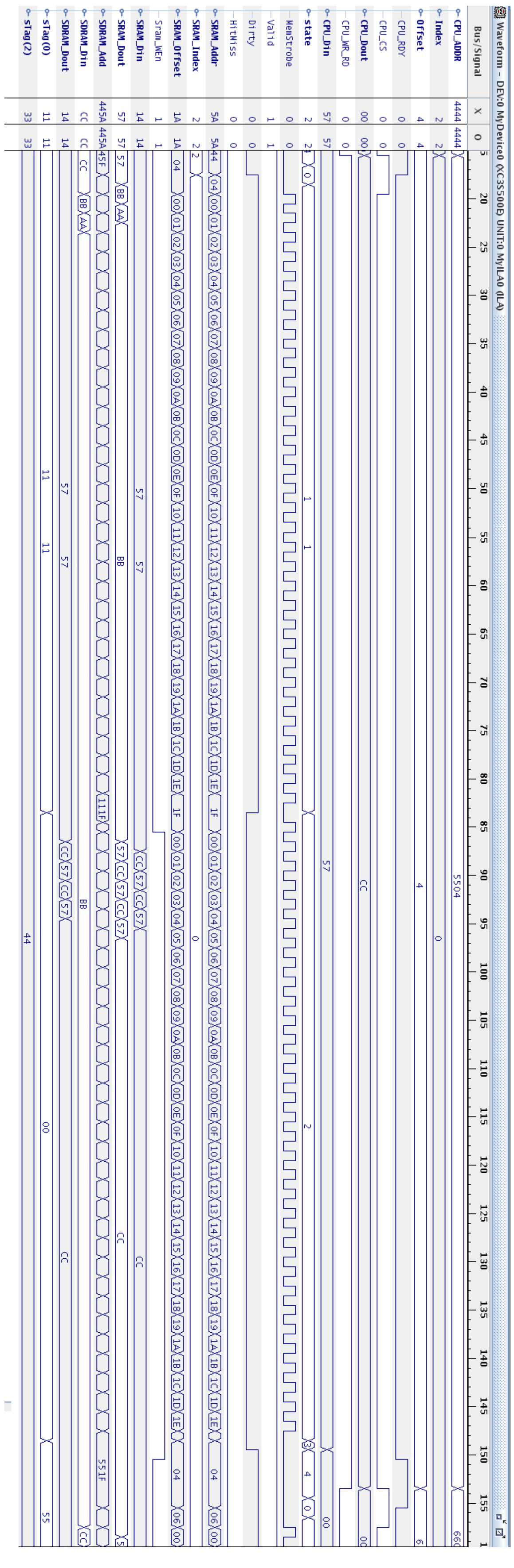


Figure 18: Read/Write from/to cache [miss] and dirty bit = 1

```vhdl
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    12:35:06 10/22/2019
-- Design Name:
-- Module Name:    CacheController - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Upper Level Cache Controller Entity
entity CacheController is
   Port (
      clk                     : in STD_LOGIC;
      Cache_Addr_Comp         : out STD_LOGIC_VECTOR(15 downto 0);
      Cache_Dout_Comp         : out STD_LOGIC_VECTOR(7 downto 0);
      SRAM_Addr_Comp          : out STD_LOGIC_VECTOR(7 downto 0);
      SRAM_Din_Comp           : out STD_LOGIC_VECTOR(7 downto 0);
      SRAM_Dout_Comp          : out STD_LOGIC_VECTOR(7 downto 0);
      SDRAM_Addr_Comp         : out STD_LOGIC_VECTOR(15 downto 0);
      SDRAM_Din_Comp          : out STD_LOGIC_VECTOR(7 downto 0);
      SDRAM_Dout_Comp         : out STD_LOGIC_VECTOR(7 downto 0);
      Cache_SRAMAddr_Comp     : out STD_LOGIC_VECTOR(7 downto 0);
      Cache_WR_RD_Comp        : out STD_LOGIC;
      Cache_memStrb_Comp      : out STD_LOGIC;
      Cache_RDY_Comp          : out STD_LOGIC;
      Cache_CS_Comp           : out STD_LOGIC
   );
end CacheController;

architecture Behavioral of CacheController is
   --CPU Signals
   signal Cpu_Dout_Sig, Cpu_Din_Sig     : STD_LOGIC_VECTOR(7 downto 0);
   signal Cpu_Addr_Sig                  : STD_LOGIC_VECTOR (15 downto 0);
   signal Cpu_WR_RD_Sig,Cpu_CS_Sig      : STD_LOGIC;
   signal Cpu_Rdy_Sig                   : STD_LOGIC;
   signal Cpu_Tag_Sig                   : STD_LOGIC_VECTOR(7 downto 0);
   signal Cpu_Index_Sig                 : STD_LOGIC_VECTOR(2 downto 0);
   signal offset                        : STD_LOGIC_VECTOR(4 downto 0);

   --SRAM(cache memory) Signals
   signal DirtyByte             : STD_LOGIC_VECTOR(7 downto 0):= "00000000";
   signal ValidByte             : STD_LOGIC_VECTOR(7 downto 0):= "00000000";
   signal sADD, sDin, sDout     : STD_LOGIC_VECTOR(7 downto 0);
```

```vhdl
	signal sWen                    : STD_LOGIC_VECTOR(0 DOWNTO 0);
	signal HitMiss                    : STD_LOGIC := '0';
	type cachemem is array (7 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
	signal sTAG: cachemem := ((others=> (others=>'0')));

	--SDRAM Signals
	signal SDRAM_Din,SDRAM_Dout : STD_LOGIC_VECTOR(7 downto 0);
	signal SDRAM_ADD            : STD_LOGIC_VECTOR(15 downto 0);
	signal SDRAM_MSTRB,SDRAM_W_R: STD_LOGIC;

	--Counters
	signal counter              : integer := 0;
	signal delay_counter        : integer := 0;
	signal delay_counterb       : integer := 0;
	signal startup              : integer := 0;
	signal offset_counter       : integer := 0;


	--ICON & VIO & ILA Signals
	signal control0             : STD_LOGIC_VECTOR(35 downto 0);
	signal control1             : STD_LOGIC_VECTOR(35 downto 0);
	signal ila_data             : std_logic_vector(149 downto 0);
	signal trig0                : std_logic_vector(0 DOWNTO 0);

	--State
	signal state : STD_LOGIC_VECTOR(3 downto 0);

	--Components
	COMPONENT SDRAMController
	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;

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

	COMPONENT CPU_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;

      COMPONENT icon
      PORT (
         CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
         CONTROL1 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0)
      );
      END COMPONENT;

      COMPONENT ila
      PORT (
         CONTROL  : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
         CLK      : IN STD_LOGIC;
         DATA     : IN STD_LOGIC_VECTOR(149 DOWNTO 0);
         TRIG0    : IN STD_LOGIC_VECTOR(0 TO 0)
      );
      END COMPONENT;

      component vio
      PORT (
      CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
      ASYNC_OUT : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
      );

      end component;
      signal vioOut : std_logic_vector(1 downto 0);
      signal ilaTrig : std_logic_vector(7 downto 0);

      BEGIN

      --PORT MAPS:
      myCPU_gen : CPU_gen
      Port Map (clk => clk,
                rst => vioOut(0),
                trig => Cpu_Rdy_Sig,
                Address => Cpu_Addr_Sig,
                wr_rd   => Cpu_WR_RD_Sig,
                cs      => Cpu_CS_Sig,
                DOut    => Cpu_Dout_Sig
      );


      SDRAM     : SDRAMController
      Port Map (clk  => clk,
                ADDR => SDRAM_ADD,
                WR_RD=> SDRAM_W_R,
                MEMSTRB=> SDRAM_MSTRB,
                DIN=> SDRAM_Din,
                DOUT=> SDRAM_Dout
      );

      mySRAM    : SRAM
      Port Map (clka => clk,
                wea => sWen,
                addra => sADD,
                    dina => sDin,
                    douta=> sDout
        );

        myIcon    : icon
        Port Map (CONTROL0 => control0,
                    CONTROL1 => control1
        );

        myILA     : ila
        Port Map (CONTROL => control0,
                    CLK => clk,
                    DATA => ila_data,
                    TRIG0 => trig0
        );


        myvio : vio
        port map (
           CONTROL => control1,
           ASYNC_OUT => vioOut
        );

        process(clk, Cpu_CS_Sig)
        begin
           if (clk'event AND clk = '1') then

              if (startup = 0) then
              state <= "0100";
              startup <= 1;
              end if;

                 --evaluation 0000 0
                 --Write      0001 1
                 --Load       0010 2
                 --CPU wr/rd  0011 3
                 --READY      0100 4

              -- STATE 0
              if (state = "0000") then
                 delay_counter <= 0;
                 Cpu_Rdy_Sig <= '0';
                 Cpu_Tag_Sig <= Cpu_Addr_Sig(15 downto 8);
                 Cpu_Index_Sig <= Cpu_Addr_Sig(7 downto 5);
                 offset <= Cpu_Addr_Sig(4 downto 0);
                 SDRAM_ADD(15 downto 5) <= Cpu_Addr_Sig(15 downto 5);
                 sADD(7 downto 0) <= Cpu_Addr_Sig(7 downto 0);
                 sWen <= "0";
                 if (delay_counterb >= 1) then


                    --Evaluating HIT/MISS
                    if((ValidByte(to_integer(unsigned(Cpu_Index_Sig))) = '1') AND (sTAG(
     to_integer(unsigned(Cpu_Index_Sig))) = Cpu_Tag_Sig)) then -- HIT
                       HitMiss <= '1';
                       state <= "0011";
                    else --MISS
```

```
                     HitMiss <= '0';
                     --Dirty and Valid bit check
                     if ((DirtyByte(to_integer(unsigned(Cpu_Index_Sig))) = '1') AND (
ValidByte(to_integer(unsigned(Cpu_Index_Sig))) = '1')) then
                        state <= "0001"; --Switching to write state
                     else
                        state <= "0010"; --Switch to load state
                     end if;
                  end if;
               end if;
               delay_counterb <= delay_counterb + 1;

            -- STATE 1 MISS, write-back to SDRAM
            elsif(state = "0001") then
               SDRAM_ADD(15 downto 8) <= sTAG(to_integer(unsigned(Cpu_Index_Sig)));
               SDRAM_ADD(7 downto 5) <= Cpu_Index_Sig;
               if (counter = 64) then
                  counter <= 0;
                  DirtyByte(to_integer(unsigned(Cpu_Index_Sig))) <= '0';
                  sTAG(to_integer(unsigned(Cpu_Index_Sig))) <= (others=>'0');
                  offset_counter <= 0;
                  state <= "0010"; -- switch to load state, write complete
               else
                  if (counter mod 2 = 1) then
                     SDRAM_MSTRB <= '0';
                  else
                     SDRAM_MSTRB <= '1';
                     SDRAM_ADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(offset_counter
, 5));
                     SDRAM_W_R <= '1';
                     SDRAM_Din <= sDout;

                     sADD(7 downto 5) <= Cpu_Index_Sig;
                     sADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(offset_counter, 5));
                     sWen <= "0";

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

            -- STATE 2 MISS read
            elsif(state = "0010") then
               SDRAM_ADD(15 downto 8) <= Cpu_Tag_Sig;
               SDRAM_ADD(7 downto 5) <= Cpu_Index_Sig;
               if (counter = 64) then
                  counter <= 0;
                  ValidByte(to_integer(unsigned(Cpu_Index_Sig))) <= '1';
                  sTAG(to_integer(unsigned(Cpu_Index_Sig))) <= Cpu_Tag_Sig;
                  offset_counter <= 0;
                  state <= "0011"; -- switch to cpu rd/wr
                  SDRAM_MSTRB <= '0';
                  sADD(7 downto 0) <= Cpu_Addr_Sig(7 downto 0);
               else
                  if (counter mod 2 = 0) then
                     SDRAM_MSTRB <= '1';
                  else
```

```vhdl
                  SDRAM_MSTRB <= '0';
                  SDRAM_ADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(offset_counter, 5));
                  SDRAM_W_R <= '0';

                  sADD(7 downto 5) <= Cpu_Index_Sig;
                  sADD(4 downto 0) <= STD_LOGIC_VECTOR(to_unsigned(offset_counter, 5));
                  sDin <= SDRAM_Dout;
                  sWen <= "1";

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

         -- STATE 3 CPU read/write
         elsif(state = "0011") then
            if (Cpu_WR_RD_Sig = '1') then
               sWen <= "1";
               DirtyByte(to_integer(unsigned(Cpu_Index_Sig))) <= '1';
               ValidByte(to_integer(unsigned(Cpu_Index_Sig))) <= '1';
               sDin <= Cpu_Dout_Sig;
               Cpu_Din_Sig <= "00000000";
            else
               Cpu_Din_Sig <= sDout;
            end if;
         state <= "0100";

         -- STATE 4  send ready, wait for CS
         elsif(state = "0100") then
            delay_counterb <= 0;
            sWen <= "0";
            Cpu_Rdy_Sig <= '1';
            if ((Cpu_CS_Sig = '1') AND (delay_counter >= 2)) then
               state <= "0000";
            end if;
            delay_counter <= delay_counter + 1;
         end if;
      end if;
   end process;

   -- Signals
   Cache_memStrb_Comp <= SDRAM_MSTRB;
   Cache_Addr_Comp    <= Cpu_Addr_Sig;
   Cache_WR_RD_Comp   <= Cpu_WR_RD_Sig;
   Cache_Dout_Comp    <= Cpu_Din_Sig;
   Cache_RDY_Comp     <= Cpu_Rdy_Sig;
   Cache_CS_Comp      <= Cpu_CS_Sig;
   SRAM_Addr_Comp     <= sADD;
   SRAM_Din_Comp      <= sDin;
   SRAM_Dout_Comp     <= sDout;
   SDRAM_Addr_Comp    <= SDRAM_ADD;
   SDRAM_Din_Comp     <= SDRAM_Din;
   SDRAM_Dout_Comp    <= SDRAM_Dout;
   Cache_SRAMAddr_Comp<= Cpu_Addr_Sig(15 downto 8);

   -- ILA outputs
```

```
        ila_data(15 downto 0)  <= Cpu_Addr_Sig;
        ila_data(16)           <= Cpu_WR_RD_Sig;
        ila_data(17)           <= Cpu_Rdy_Sig;
        ila_data(18)           <= Cpu_CS_Sig;
        ila_data(26 downto 19) <= Cpu_Din_Sig;
        ila_data(30 downto 27) <= state;
        ila_data(31)           <= SDRAM_MSTRB;
        ila_data(32)           <= ValidByte(to_integer(unsigned(Cpu_Index_Sig)));
        ila_data(33)           <= DirtyByte(to_integer(unsigned(Cpu_Index_Sig)));
        ila_data(34)           <= HitMiss;
        ila_data(42 downto 35) <= sADD;
        ila_data(50 downto 43) <= sDin;
        ila_data(58 downto 51) <= sDout;
        ila_data(74 downto 59) <= SDRAM_ADD;
        ila_data(82 downto 75) <= SDRAM_Din;
        ila_data(90 downto 83) <= SDRAM_Dout;
        ila_data(98 downto 91) <= Cpu_Dout_Sig;
        ila_data(99)           <= sWen(0);
        ila_data(107 downto 100) <= sTag(0);
        ila_data(115 downto 108) <= sTag(1);
        ila_data(123 downto 116) <= sTag(2);
        ila_data(131 downto 124) <= sTag(3);
        ila_data(139 downto 132) <= sTag(4);
        ila_data(147 downto 140) <= sTag(5);


        -- Trigger
        ilaTrig(0) <= vioOut(0);
        ilaTrig(7 downto 1) <= (others => '0');
    end Behavioral;

```

```vhdl
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    12:37:55 10/22/2019
-- Design Name:
-- Module Name:    SDRAMController - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SDRAMController 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 SDRAMController;

architecture Behavioral of SDRAMController is
   type mem is array (6 downto 0, 31 downto 0) of std_logic_vector(7 downto 0);
   signal first4  : STD_LOGIC_VECTOR(3 downto 0);
   signal last4   : STD_LOGIC_VECTOR(4 downto 0);
   signal mem_sig: mem;
   signal counter : integer := 0;
   begin
      process (CLK)
      begin
      if CLK'event and CLK = '0' then
         if counter = 0 then
            for I in 0 to 6 loop
               for J in 0 to 31 loop
                  if (J mod 3 = 1) then
                     mem_sig(I,J) <= "00010100"; --14
                  elsif (J mod 3 = 2) then
                     mem_sig(I,J) <= "10001111"; --8F
                  else
                     mem_sig(I,J) <= "01010111"; --67
                  end if;
               end loop;
            end loop;
            counter <= 1;
         end if;
         if MEMSTRB = '1' then
            if WR_RD = '1' then
               mem_sig(to_integer(unsigned(ADDR(15 downto 12))),to_integer(unsigned(
ADDR(4 downto 0)))) <= DIN;
            ELSE
               DOUT <= mem_sig(to_integer(unsigned(ADDR(15 downto 12))),to_integer(
unsigned(ADDR(4 downto 0))));
            end if;
         end if;
      end if;
      end process;
end Behavioral;
```