Toronto Metropolitan University

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

Course Number	COE758
Course Title	Digital System Engineering
Semester/Year	F2024
Instructor	Dr. Lev Kirischian
Section No.	01
Group No.	NA
Submission Date	2024/10/15
Due Date	2024/10/15

Assignment Title	Cache Controller Project Report

Name	**Student ID**	**Signature***
Vaishali Jadon	501051835	V.J.
Atiya Azeez	501052595	A.A.

TABLE OF CONTENTS

Abstract	2
Introduction	2
Purpose	3
Theory	4
Internal Specifications	4
Device Design Description	9
Timing Results	12
Qualitative Results	16
Conclusion	16
References	17
Appendix	18

Abstract

Objective

The main goals of this project were to understand how a cache controller works and how it can access data from the SRAM memory (used for the cache) and SDRAM controllers (main memory). It focused on designing custom logic controllers and interfacing them with the SRAM and other logic devices. The project aimed to build skills in VHDL coding using the Xilinx ISE CAD environment, with the Xilinx Spartan-3E FPGA as the hardware platform. This experience helped us learn about memory management and controller design in digital systems.

Approach

The Cache Controller was developed using VHDL. The cache project implemented the CPU's VHDL implementation provided as well as the ICON, ILA, and BRAM cores taught in previous tutorials. In the initial stages, a schematic diagram, block diagram, and conceptual finite state machine (FSM) were created. Following this, the Cache Controller symbol was generated and the VHDL code for it was written. The hardware emulation was observed using the Chipscore Analyzer which displayed the different cases that the cache supports.

Introduction

Modern computer systems are designed with a variety of memory types that are organized in a memory hierarchy to optimize performance and cost efficiency. In such hierarchies, as the distance from the CPU increases, both the capacity and the access time of the memory increase. This structure enables processors to effectively utilize slower, larger, and less expensive memory. This reduces the delays associated with accessing large, slow memories by using a faster, smaller memory close to the processor.

At the top of this hierarchy is the cache memory, a small, high-speed storage area used to hold data that has been accessed recently and is likely to be needed again soon. Data in the cache is stored in blocks, each containing multiple words. To manage these blocks and ensure efficient data access, a cache controller is required. It maintains necessary information, including the block's index, tag, valid bit, and dirty bit, which helps the controller track the contents of the cache and how they map to the main memory. Since there are multiple words in a block, an offset is used to access a specific word.

The cache controller handles read and write requests from the CPU by either retrieving or updating data within the cache or transferring whole blocks of data to and from the main memory as needed. This report details the implementation of a cache controller in VHDL, designed to interface with a pre-existing CPU model. The report covers the development process, including design diagrams, VHDL coding, hardware emulation results, and analysis of the cache's performance.

Purpose

The purpose of this report is to understand how the concept of memory hierarchies improves the performance and efficiency of computer systems. Memory hierarchies help bridge the gap between the fast processing power of CPUs and the slower speed of main memory. By organizing memory into levels, from the small and fast cache memory close to the CPU to the larger and slower main memory, computer systems can work more efficiently. This report aims to show how good memory hierarchy design can reduce delays in accessing data and make data transfer faster. The goal is to gain a better understanding of how memory management is critical for improving performance.

Theory

Cache memory stores data in blocks, each containing multiple words. These blocks are identified by several key components: the index, tag, valid bit, dirty bit, and offset. The index helps the system locate which block in the cache corresponds to the requested data, while the tag is used to verify that the block contains the correct data. The valid bit indicates whether the block contains valid data, and the dirty bit shows whether the data in the block has been modified and needs to be written back to the main memory. The offset is used to access individual words within the block.

The Cache Controller is responsible for managing data transfers between the CPU, cache memory, and main memory. It handles both read and write requests from the CPU by determining whether the requested data is in the cache (a cache hit) or not (a cache miss). If a cache hit occurs, the controller directly accesses or updates the data in the cache. If a cache miss occurs, the controller fetches the appropriate data block from the main memory or writes modified blocks back to the main memory before updating the cache.

For this project, the cache can store 256 bytes of data. It is structured into 8 blocks, each holding 32 one-byte words. The CPU communicates with the cache controller by sending 16-bit addresses, which are divided into the tag, index, and offset. The Cache Controller uses this address information to compare the tags of the cache blocks, check the valid and dirty bits, and determine the appropriate action for each read or write request.

The behavioral operation of the Cache Controller can be divided into four main cases:

1. **Write a Word to Cache (Cache Hit):** When the CPU issues a write request for data already stored in the cache, it is categorized as a cache hit. The Cache Controller uses the index and offsets from the CPU's address to locate the correct position in the SRAM. The new data is then written to this location, and both the dirty and valid bits for the corresponding block are set to 1. This indicates that the data has been modified and the cache block has been used.
2. **Read a Word from Cache (Cache Hit):** If the CPU requests to read data that exists in the cache, the Cache Controller identifies it as a cache hit. It retrieves the data using the index and offset from the CPU's address and sends the requested data back to the CPU.
3. **Read/Write from/to Cache (Cache Miss, Dirty Bit = 0):** When a read or write request is made but the corresponding block is not found in the cache (the tags do not match), a cache miss occurs. If the dirty bit is 0, the data in the cache has not been modified, thus the Cache Controller can replace the old block with the new block from the main memory. The CPU address is sent with the initial offset set to “00000” to read the entire 32-byte block. This block is then written to the SRAM, and the tag register is updated with the new tag from the CPU address. The valid bit is set to 1, and the procedure for a hit operation is performed.
4. **Read/Write from/to Cache (Cache Miss, Dirty Bit = 1):** When a cache miss occurs and the dirty bit is set to 1, the Cache Controller must first write the modified data back to the main memory. The data block from the SRAM is written to the following address: tag & index & 00000. After writing back, the Cache Controller retrieves the new block from the main memory using the same address format. The cache is then updated with the new data block and the original CPU request is executed.

Internal Specifications

1. CPU Interface

The CPU requests write and read transactions to the cache. The CPU itself has a chop select strobe (CS), a read/write enable (WR/RD), a 16-bit address (ADD), 8-bit data input and output ports: DIN and DOUT, and a ready indicator (RDY). The clock signal (CLK) is synchronized with the Cache Controller and sends read or write instructions to it. The ready signal (RDY) is used to indicate whether the cache is in an idle or busy state. The ports are shown in Figure 1. The CPU interface has an 8-bit data input pin (DIN) that gets data from the Cache (SRAM). It also sends an address to the Controller, along with a read/write signal and a CS strobe. The CS strobe tells the SRAM that the address, read/write signal, and data output are ready to be used for the instruction. The CS signal stays asserted for four clock cycles. The RDY signal is used by the Cache Controller to communicate the completion of a transaction. When the Cache Controller is idle and ready to accept new transactions, it asserts the RDY signal. When a transaction is received, the RDY signal is deasserted and remains low until the requested operation is fully completed. An example of a write operation is shown in Figure 2.

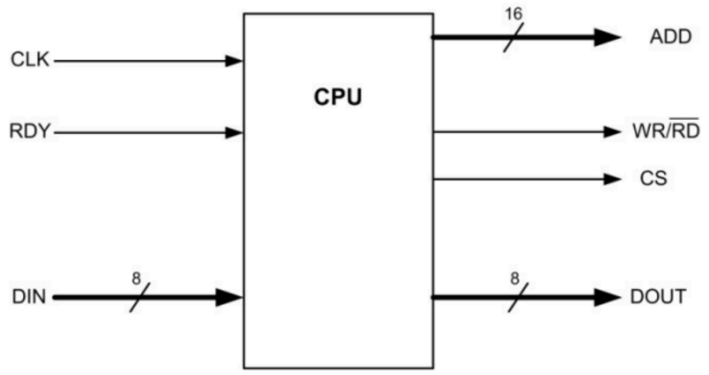


Figure 1. CPU symbol interface.

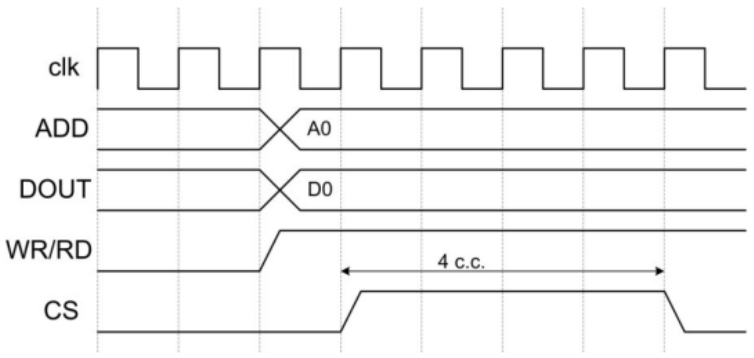


Figure 2. CPU transaction example.

2. SDRAM Controller

The SDRAM Controller processes main memory block read and write requests from the cache. Its block symbol is shown in Figure 7. It consists of a 16-bit address (ADD), a read/write signal (WR/RD), a strobe signal (MEMSTRB), and data input and output ports (DIN and DOUT). The SDRAM Controller and Cache Controller are synchronized using the same clock.

To write a word to the main memory, the Cache Controller sends the block address (tag & index & offset) and sets the WR/RD signal to 1 to indicate a write operation. After ensuring all signals are stable for one clock cycle, the MEMSTRB signal is asserted for that cycle. Writing a complete block of data to memory requires repeating this process 32 times, with incrementing the offset of the address and adjusting the data accordingly.

When reading a full block from memory, the procedure is similar, except the WR/RD signal is cleared to 0 so no data gets written. Instead, the requested data will be output on the DOUT port. Like writing, reading an entire block takes 32 clock cycles to complete. Figures 8 and 9 below demonstrate the read and write processes for an entire block of data.

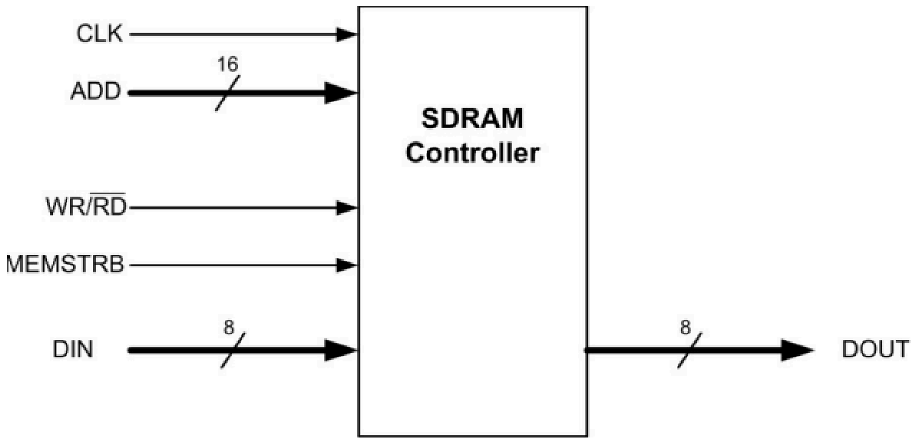


Figure 3. SDRAM controller interface.

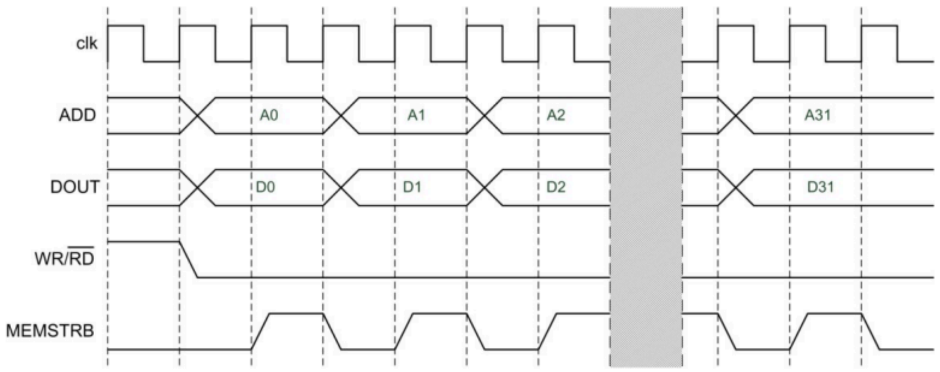


Figure 4. SDRAM block read.

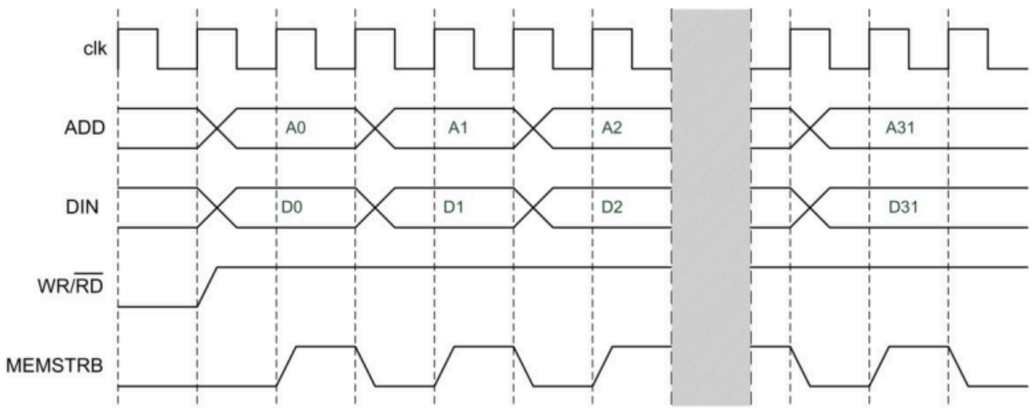


Figure 5. SDRAM block write.

3. Local SRAM

The BlockRAM memory (Figure 6) is used for local cache storage. This interface consists of an 8-bit address (ADD), 8-bit data input and output lines (DIN and DOUT), and a write enable signal (WEN). Like the other components, the BlockRAM operates using the same clock signal as the controller. All commands sent to the cache controller are synchronized with the clock's rising edge. For read operations, the correct address is placed on the address bus, and the corresponding data will be available on the DOUT line after the next clock-rising edge (Figure 7). Write operations involve specifying both the address and input data, followed by activating the write enable signal (WEN). The data will then be written to the designated address during the next rising edge of the clock (Figure 8).

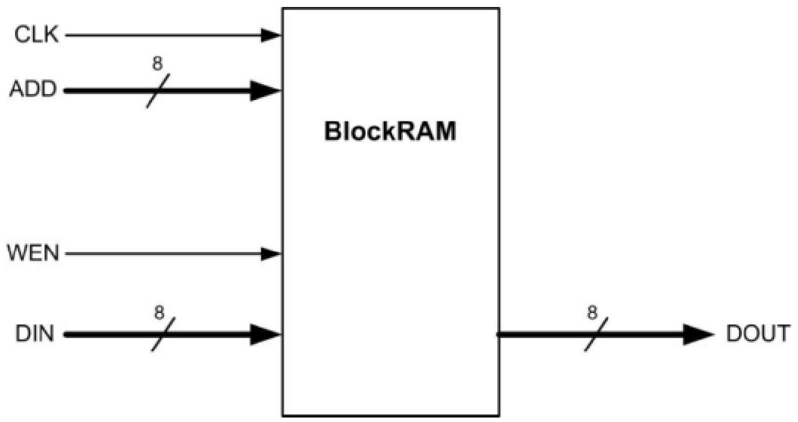


Figure 6. BlockRAM interface

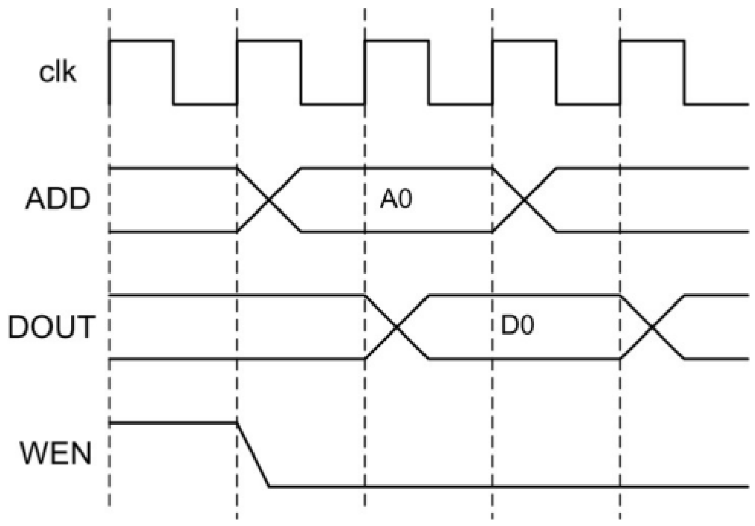


Figure 7. BlockRAM read operation

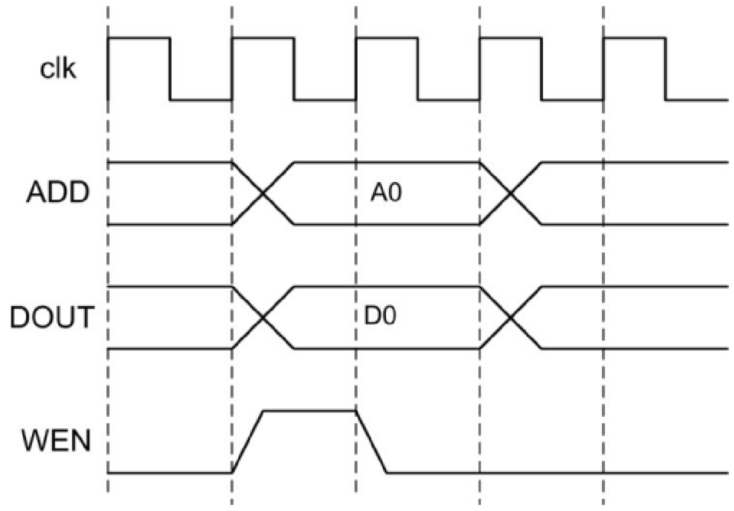


Figure 8. BlockRAM write operation

Device Design Description

Cache symbol:

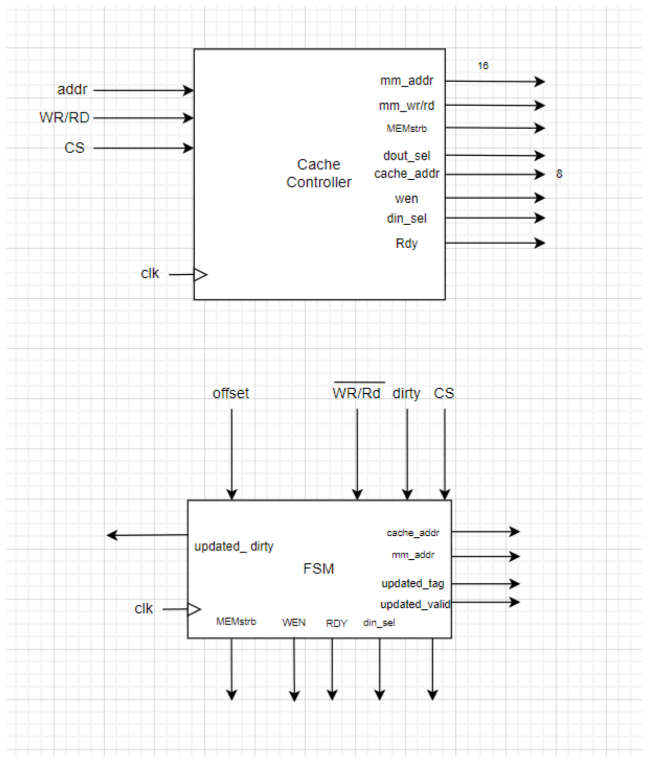


Figure 9. Cache controller

Block diagram:

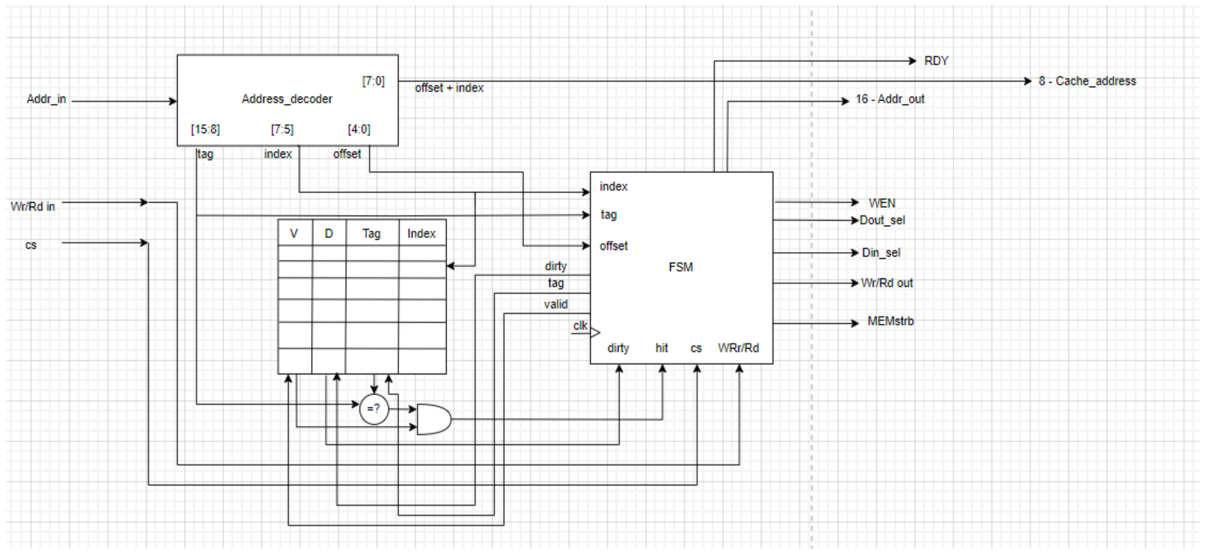


Figure 9. Block diagram of the cache controller

The block diagram of the Cache Controller in Figure 9 illustrates the component's internal workings. The 16-bit Input Address is connected to both the Address Decoder and FSM. The Address Decoder processes the incoming address by breaking it down into three fields: the tag field, the index field, and the offset field. The tag field identifies the block in the cache. The index field is used to locate the corresponding cache block within the cache memory that may contain the requested data, while the offset identifies a specific location within that block. Additionally, there are two values: dirty and valid used to check the data. The valid bit indicates whether data is stored in the cache entry and the dirty bit signifies whether the byte has been modified since it was loaded into memory. The valid bit value is combined with the results of comparing the address and tag assigned to that cache index in a logical AND gate. The outcome of this AND operation is then sent as the "hit" of the FSM. Subsequently, the FSM control generates several output signals: RDY, Addr_out, WEN, Din_sel, Dout_sel, WR/RD, and MEMSTRB.

State Machine Diagram

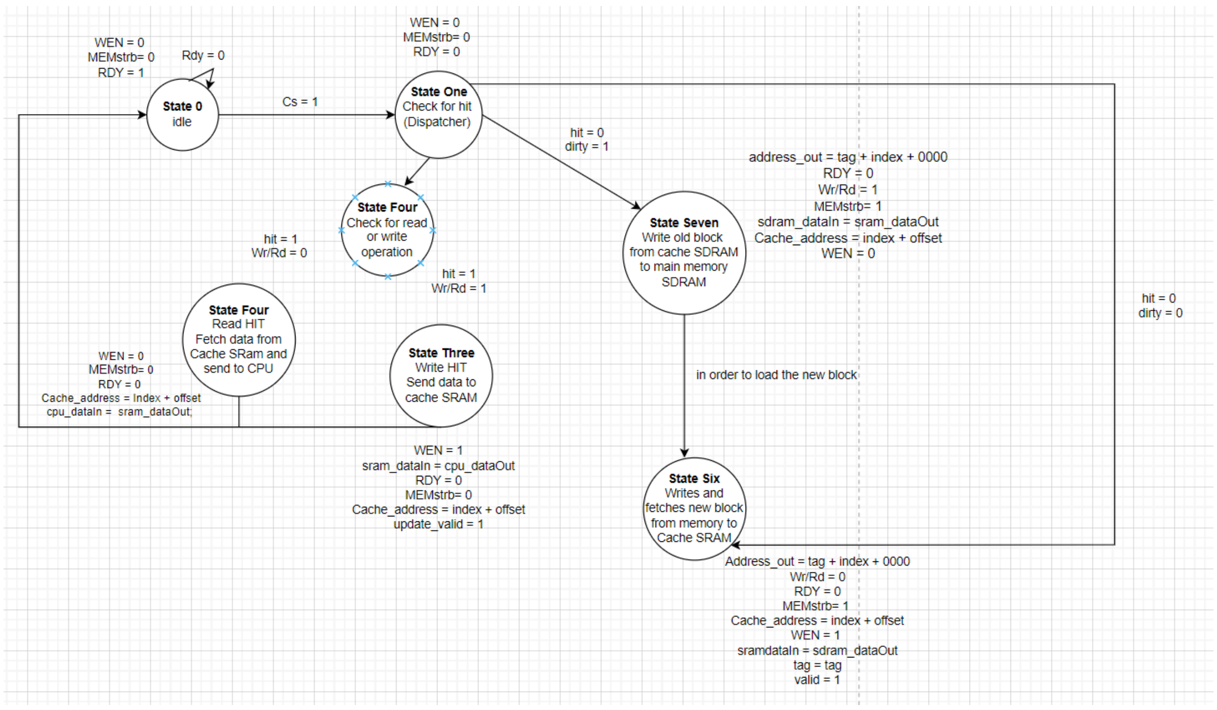


Figure 10. Finite state machine

The FSM in Figure 10, outlines the different states and transitions used by the controller. It consists of seven states, each accompanied by specific conditions for transitions and control signals. The states are outlined as follows:

State 0: Check if the cache is idle. If the CPU sends a request to the cache (CS=1), then the cache can accept this request. The ready signal is deasserted once the cache starts to process the CPU request.

State 1: Decodes the address sent from the cpu into its respective tag, index, and offset values and checks for a hit or miss by comparing the tags and valid bits. The cache tag is taken from the tag table and the valid bit is taken from the valid signal. The WEN signal is set to zero, to ensure that no data is written to cache before the checking occurs. The MEMSTRB signal is deasserted since no data is being read or written from the main memory.

State 2: If a hit occurs, this state determines whether it is a read or write operation that is performed. It uses the read and write enable signal (wr/rd) to determine this.

State 3: Performs the write operation, writing the data provided from the CPU to the specified cache address. Since a write is performed, the WEN signal needs to be asserted. In addition, since the cache data is updated, the valid is set to one for that cache block. Since the CPU request is completed, the cache returns to an idle state (state zero) making it ready to accept other requests.

State 4: Performs the read operation from the address that the CPU requested from the cache. In this case, the data out from the SRAM is outputted to the CPU Data In port. Since the CPU request is completed, the cache returns to an idle state ready to accept other requests.

State 7: If there is a miss and a dirty bit is one, then the cache block gets loaded into the memory. Since the SDRAM is written to, the SDRAM wr/rd enable signal must be asserted. To ensure that data is only written on the rising edge, the data is only read when the MEMSTRB is set to one.

State 6: If the dirty bit is zero, then the new memory block that the CPU address is included in, gets written to the cache. For this operation, the SDRAM wr/rd enable signal is deasserted since only data is being read. Once again, to ensure data output is synchronized, the SDRAM will only output data on the rising edge, when MEMSTRB is one. Moreover, the WEN for SRAM is set to one since the new block is being written and the valid bit is set to 1 to indicate that this cache block has data. Once this is completed, this state goes to state 2 to determine whether a read or write is performed.

Timing Results

1. Cache read operation

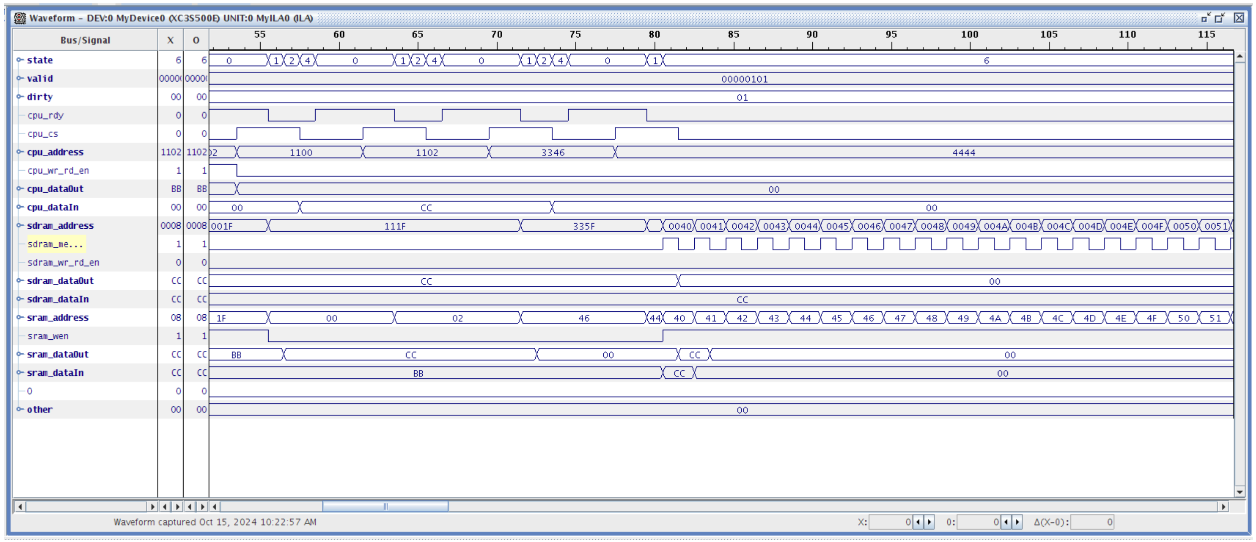


Figure 11. Cache operation (state 4) waveform

The waveform in Figure 11 shows a cache read operation (x=60). The waveform starts from state 0, which is the idle state. Once cpu_cs is set to one, the FSM proceeds to state 1 where cpu_rdy is 0 because the cache is not available. In state 1, the cache controller checks whether the requested address from the CPU results in a cache hit or miss. The cpu_address is split into the tag, index, and offset: 0001 001, 000, and 00010. Hence, the sram_address is 00000010 (0x02). This time around, it's a hit, because the valid bit is 1, and the tag matches. Since a cache hit occurs, the FSM moves to state 2. In this state, the controller checks if the CPU is requesting to read or write and this will be based on the cpu_wr_rd_en signal. From this waveform, since cpu_wr_rd_en is 0, the FSM moves onto state 4 which will handle reading data from the cache (SRAM) and sending that information to the CPU. The cpu_dataIn signal is loaded with the value from sram_dataout, in this case, it is 0xCC. Since no writes are required, sram_wen and sdram_wr_rd_en are 0. After finishing the read operation, the FSM returns to state 0, reasserting cpu_ready as it waits for the next request.

2. Cache write operation

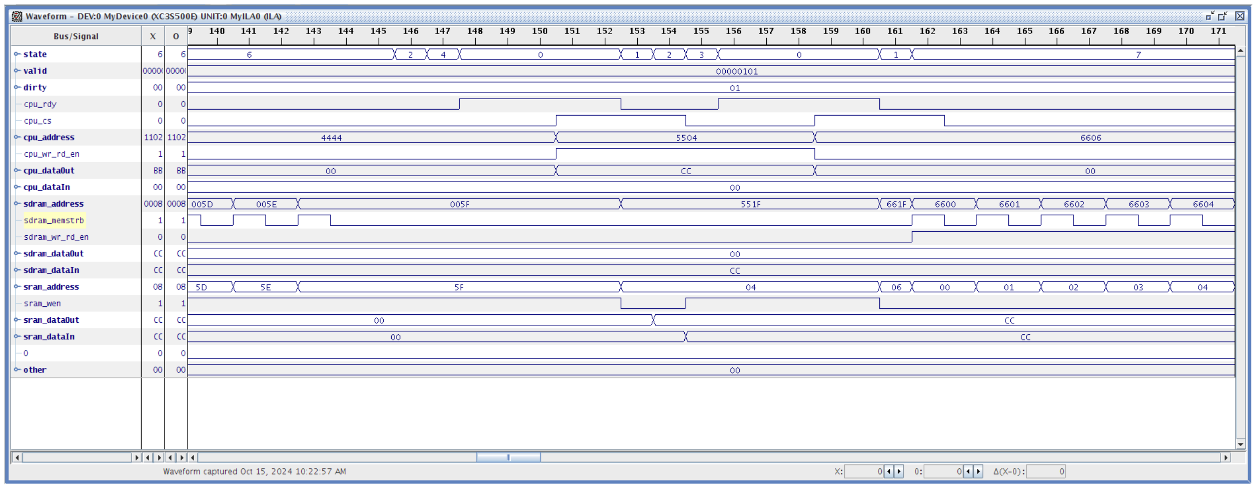


Figure 12. Cache operation (state 3) waveform

The following waveform shows the write operation (starting from x=148). Like the read operation, the waveform goes from state 0, which is the idle state, and then proceeds to state 1. In state 1, the cache controller checks whether the requested address from the CPU results in a cache hit or miss using the tag and index from the cpu_address: 0101 0101, 000, 00100. In this case, it is a hit, because the valid bit is 1 and the tag matches. This means that the FSM moves to state 2 to determine the read or write operation. Since cpu_wr_rd_en is 1, the CPU is requesting a write operation. Now we move to state 3 where the CPU is writing into the cache. From the CPU address fields, the sram_address being written to is 00100 (0x04). The controller will enable write to cache by setting sram_wen to 1. The CPU's data (cpu_dataout) will be written to the cache block (sram_dataIn). In this case, the value 0xCC is being written. It will update the valid bit to 1 and this will indicate that the cache block contains valid data. The dirty bit is also set to 1 indicating that the cache block has been changed. After state 3, the FSM moves to state 0 and waits for the next request

3. Dirty Bit is 0

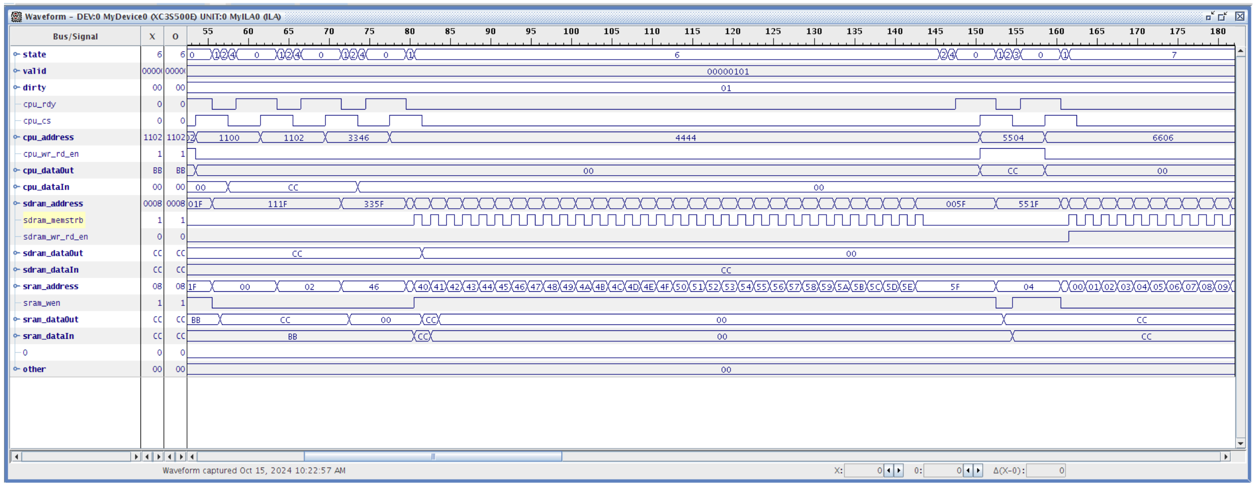


Figure 13. Cache operation (state 6) waveform

The waveform above is when the dirty bit is 0. At x = 75, the FSM is at state 0 and the cache is waiting for the CPU to send a signal command (cpu_cs). Once the signal is received, the FSM asserts cpu_rdy to 1 and that indicates the cache is performing the request. In state 1, the cpu_address is loaded and the FSM extracts the tag, index, and offset from the address: 0100 0100, 010, and 000100. The FSM will also check if the requested data is valid in the cache and whether the block is dirty. In this waveform, the valid bit is 0 (the second bit of valid) and the dirty bit is 0 (the second bit of diary). Thus, it is a cache miss. Since the dirty bit is 0, it proceeds to state 6 to load the missing block from the main memory (SDRAM) into the cache (SRAM). A 64-cycle counter is used to track the 32-byte data transfers since a full block write takes 32 clock cycles (one block per cycle). The cache controller loads a new block from SDRAM to SRAM, byte by byte over multiple clock cycles. This is seen as the SRAM address increments by one each cycle as it writes each byte from memory. In addition, the waveform shows that data outputs from the SDRAM when the sdram_memstrb is on the rising edge. Since the SRAM is being written to, sram_wen is set to one and since SDRAM is being read from, sdram_wr_rd_en is set to 0. Once the counter hits 64, the transfer is complete and the FSM transitions to state 2. Now at state 2, it will handle the CPU's original request. FSM will either read or write operations depending on the value of the cpu_w_rd_en signal. Since cpu_wr_rd_en = 0, the FSM will move to state 4 read data from the cache, and move it forward to the CPU. The FSM will then return to state 0 in its idle state.

4. Dirty bit is 1

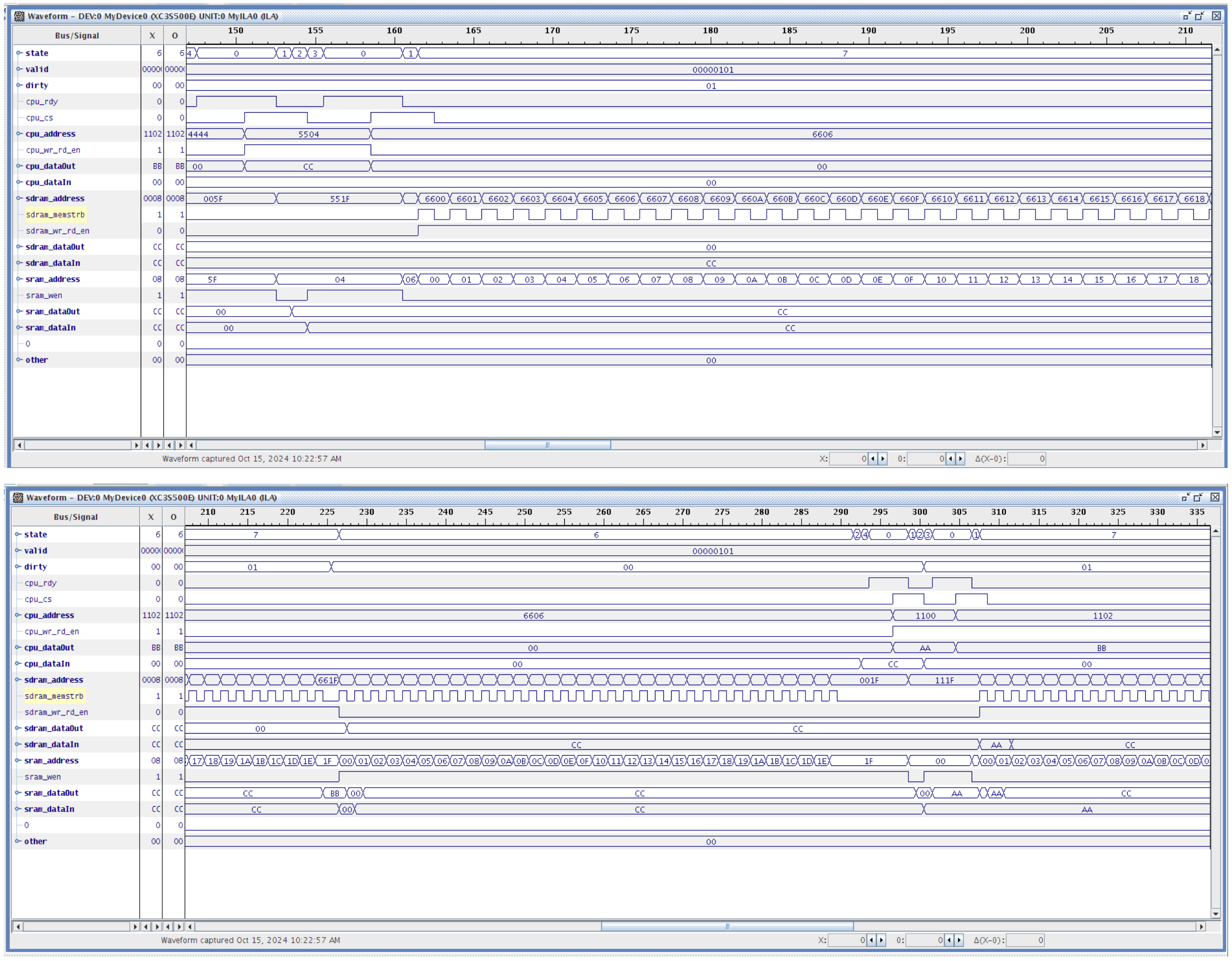


Figure 14. Cache operation (state 7) waveform

This waveform above represents when dirty bit = 1. Looking at x= 155, we can see that the cache controller is in an idle state, and the cache is waiting for a signal from the CPU to initiate memory operation. Once the signal is received, FSM will move on to state 1 (setting cpu_rdy to 0), where the cache controller checks whether it is a hit or a miss. State 1 concludes that it is a miss. Since it is a miss, it checks to see if the block is dirty. In this case, the dirty bit is 1(the lsb of dirty is 1) and it will move onto state 7. In state 7, the cache controller writes the dirty block from the SRAM back to the SDRAM. In this case, it is the block at index 000. This state is responsible for writing back a cache block to the main memory (SDRAM) when a cache miss occurs and the cache block is dirty. It will write all 32 bytes starting with offset 0x00 and incrementing by one until 0x1F each clock cycle. This will ensure data consistency between the cache and the SDRAM. From the waveform, the SRAM loaded the value 0xCC to the SDRAM. Since the SDRAM is being written to, the sdram_wr_rd_en is set to 1. The new memory block then needs to be loaded from SDRAM into the SRAM after the write-back operation. This is done by going to state 6. The controller initiates the process of fetching the new block of data from SDRAM and storing it in SRAM. This block will be written into the cache replacing the older cache line. The data will now perform its intended purpose by going to state 2 and determining whether the CPU is reading from or writing to the cache. It transitions to state 4 indicating that data the cache reads data to the CPU After it transitions to state 0.

Qualitative Results

N	**Cache performance parameter**	**Time in nS**
1	Hit / Miss determination time	10 ns
2	Data access time	10 ns
3	Block replacement time	650 ns
4	Hit time (Case 1 and 2)	30 ns
5	Miss penalty for Case 3 (when D-bit = 0)	Hit/Miss determination time + Block replacement time + hit time = 10ns + 650ns + 30ns = 690 ns
6	Miss penalty for Case 4 (when D-bit = 1)	Hit/Miss determination time + 2*Block replacement time + hit time = 10ns + 2*650ns + 30ns = 1340 ns

Figure 15. Cache performance parameters table

Conclusion

In this design project, the behavior of a cache controller was explored by implementing VHDL code using the Xilinx ISE CAD. The results of the cache controllers performance and its behaviour is detailed in Figure 15. These results agree with the theoretical principles of implementing cache controllers and their interactions with other system components.

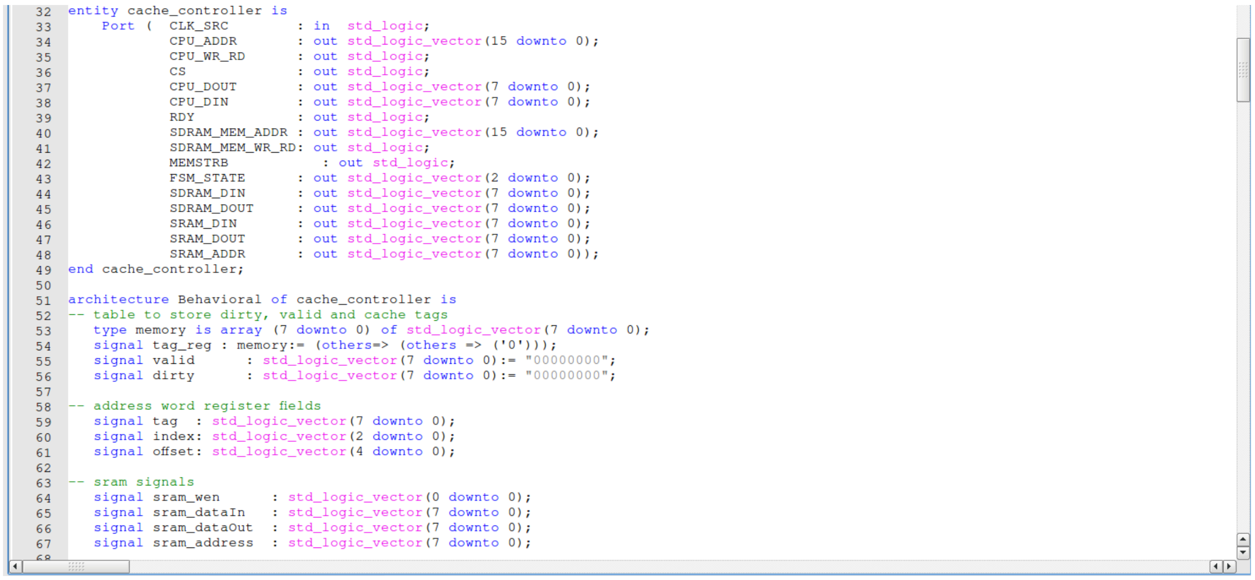
During the project, we encountered numerous challenges, many of which were related to Chipsope errors, largely caused by human error of implementing the designed FSM in VHDL to correctly transition between states for each case. We also experienced difficulties with understanding how to use the ICON and ILA to trigger the CPU so it can send requests to the cache. Overall, the project provided a deep understanding on how the cache is used and greater exposure to using VHDL in designing digital systems.

References

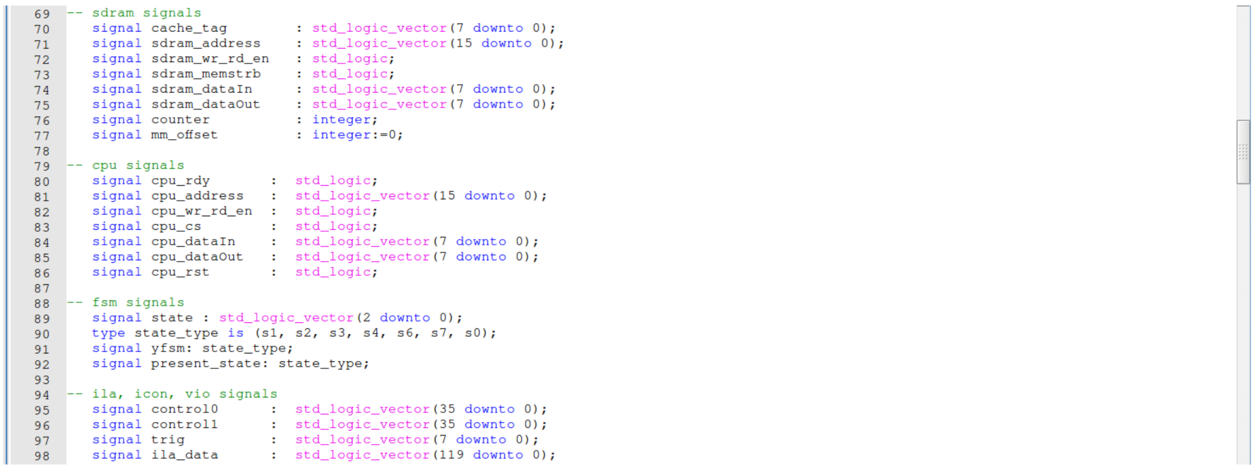
1. COE758 Digital Systems Engineering Project #1 - Memory Hierarchy: Cache Controller, TMU, Toronto, ON, Canada, 2023. Accessed: Nov. 5, 2023. [Online]. https://www.ecb.torontomu.ca/~lkirisch/ele758/labs/Cache%20Project[12-09-10].pdf

Appendix

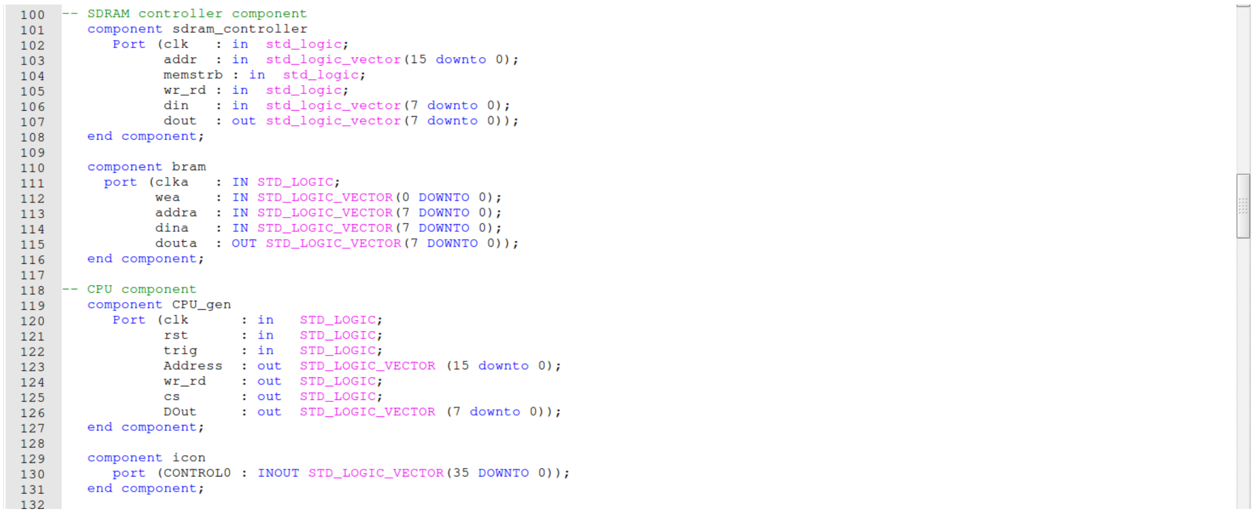
CACHE_CONTROLLER



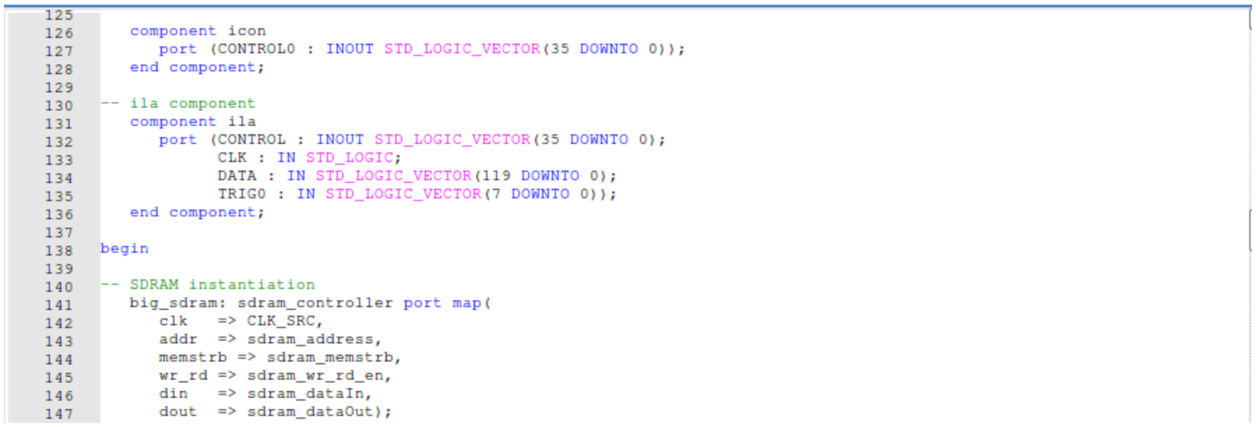
```vhdl
entity cache_controller is
    Port (  CLK_SRC         : in  std_logic;
            CPU_ADDR        : out std_logic_vector(15 downto 0);
            CPU_WR_RD       : out std_logic;
            CS              : out std_logic;
            CPU_DOUT        : out std_logic_vector(7 downto 0);
            CPU_DIN         : out std_logic_vector(7 downto 0);
            RDY             : out std_logic;
            SDRAM_MEM_ADDR  : out std_logic_vector(15 downto 0);
            SDRAM_MEM_WR_RD : out std_logic;
            MEMSTRB            : out std_logic;
            FSM_STATE       : out std_logic_vector(2 downto 0);
            SDRAM_DIN       : out std_logic_vector(7 downto 0);
            SDRAM_DOUT      : out std_logic_vector(7 downto 0);
            SRAM_DIN        : out std_logic_vector(7 downto 0);
            SRAM_DOUT       : out std_logic_vector(7 downto 0);
            SRAM_ADDR       : out std_logic_vector(7 downto 0));
end cache_controller;

architecture Behavioral of cache_controller is
-- table to store dirty, valid and cache tags
   type memory is array (7 downto 0) of std_logic_vector(7 downto 0);
   signal tag_reg : memory:= (others=> (others => ('0')));
   signal valid      : std_logic_vector(7 downto 0):= "00000000";
   signal dirty      : std_logic_vector(7 downto 0):= "00000000";

-- address word register fields
   signal tag   : std_logic_vector(7 downto 0);
   signal index: std_logic_vector(2 downto 0);
   signal offset: std_logic_vector(4 downto 0);

-- sram signals
   signal sram_wen      : std_logic_vector(0 downto 0);
   signal sram_dataIn   : std_logic_vector(7 downto 0);
   signal sram_dataOut  : std_logic_vector(7 downto 0);
   signal sram_address  : std_logic_vector(7 downto 0);
```



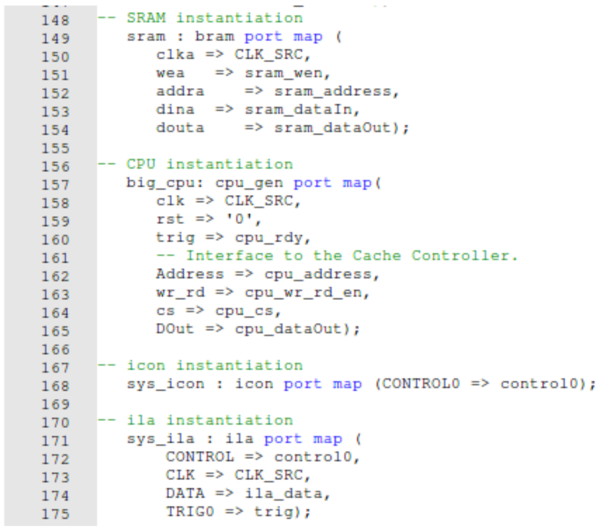
```vhdl
-- sdram signals
   signal cache_tag         : std_logic_vector(7 downto 0);
   signal sdram_address     : std_logic_vector(15 downto 0);
   signal sdram_wr_rd_en    : std_logic;
   signal sdram_memstrb     : std_logic;
   signal sdram_dataIn      : std_logic_vector(7 downto 0);
   signal sdram_dataOut     : std_logic_vector(7 downto 0);
   signal counter           : integer;
   signal mm_offset         : integer:=0;

-- cpu signals
   signal cpu_rdy       :  std_logic;
   signal cpu_address   :  std_logic_vector(15 downto 0);
   signal cpu_wr_rd_en  :  std_logic;
   signal cpu_cs        :  std_logic;
   signal cpu_dataIn    :  std_logic_vector(7 downto 0);
   signal cpu_dataOut   :  std_logic_vector(7 downto 0);
   signal cpu_rst       :  std_logic;

-- fsm signals
   signal state : std_logic_vector(2 downto 0);
   type state_type is (s1, s2, s3, s4, s6, s7, s0);
   signal yfsm: state_type;
   signal present_state: state_type;

-- ila, icon, vio signals
   signal control0      :  std_logic_vector(35 downto 0);
   signal control1      :  std_logic_vector(35 downto 0);
   signal trig          :  std_logic_vector(7 downto 0);
   signal ila_data      :  std_logic_vector(119 downto 0);
```



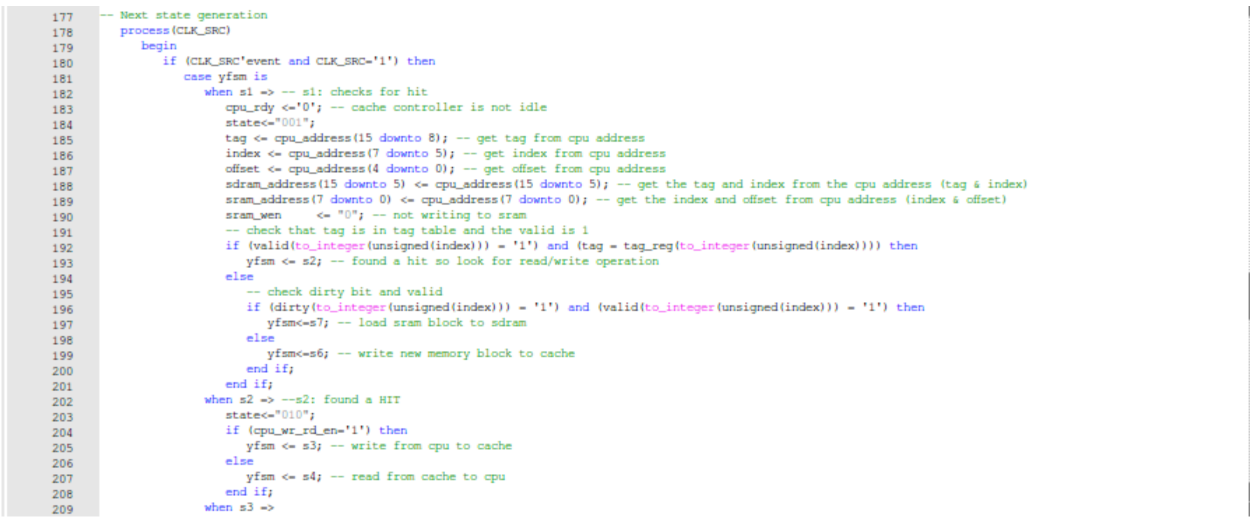
```
-- SDRAM controller component
   component sdram_controller
      Port (clk    : in  std_logic;
            addr   : in  std_logic_vector(15 downto 0);
            memstrb : in  std_logic;
            wr_rd : in  std_logic;
            din    : in  std_logic_vector(7 downto 0);
            dout   : out std_logic_vector(7 downto 0));
   end component;

   component bram
     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;

-- CPU 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));
   end component;
```



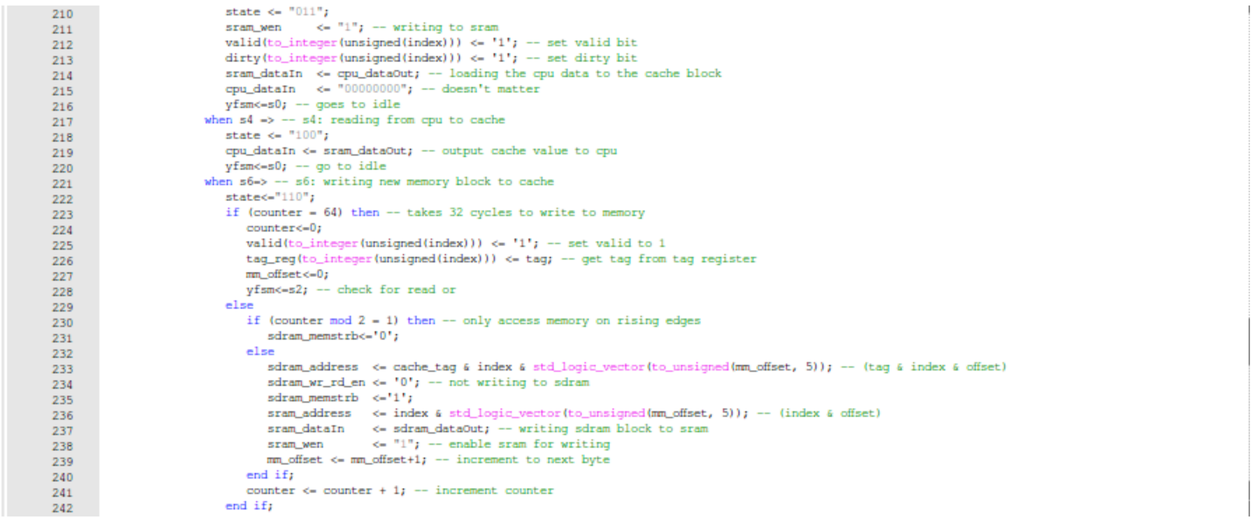
```
   component icon
      port (CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0));
   end component;

-- ila component
   component ila
      port (CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
            CLK : IN STD_LOGIC;
            DATA : IN STD_LOGIC_VECTOR(119 DOWNTO 0);
            TRIG0 : IN STD_LOGIC_VECTOR(7 DOWNTO 0));
   end component;

begin

-- SDRAM instantiation
   big_sdram: sdram_controller port map(
      clk    => CLK_SRC,
      addr   => sdram_address,
      memstrb => sdram_memstrb,
      wr_rd => sdram_wr_rd_en,
      din    => sdram_dataIn,
      dout   => sdram_dataOut);
```



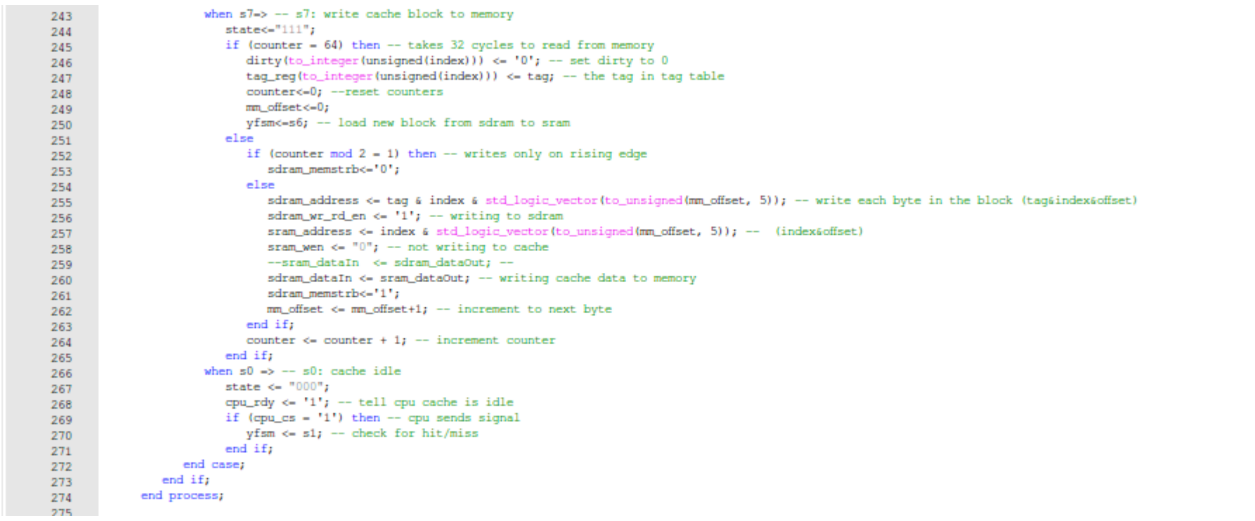
```
-- SRAM instantiation
   sram : bram port map (
      clka => CLK_SRC,
      wea    => sram_wen,
      addra     => sram_address,
      dina   => sram_dataIn,
      douta     => sram_dataOut);

-- CPU instantiation
   big_cpu: cpu_gen port map(
      clk => CLK_SRC,
      rst => '0',
      trig => cpu_rdy,
      -- Interface to the Cache Controller.
      Address => cpu_address,
      wr_rd => cpu_wr_rd_en,
      cs => cpu_cs,
      DOut => cpu_dataOut);

-- icon instantiation
   sys_icon : icon port map (CONTROL0 => control0);

-- ila instantiation
   sys_ila : ila port map (
       CONTROL => control0,
       CLK => CLK_SRC,
       DATA => ila_data,
       TRIG0 => trig);
```



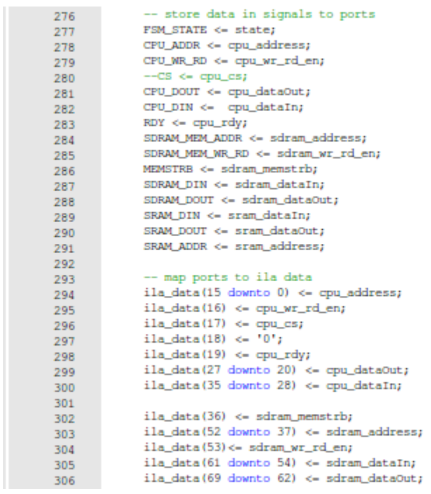
```vhdl
-- Next state generation
   process(CLK_SRC)
      begin
         if (CLK_SRC'event and CLK_SRC='1') then
            case yfsm is
               when s1 => -- s1: checks for hit
                  cpu_rdy <='0'; -- cache controller is not idle
                  state<="001";
                  tag <= cpu_address(15 downto 8); -- get tag from cpu address
                  index <= cpu_address(7 downto 5); -- get index from cpu address
                  offset <= cpu_address(4 downto 0); -- get offset from cpu address
                  sdram_address(15 downto 5) <= cpu_address(15 downto 5); -- get the tag and index from the cpu address (tag & index)
                  sram_address(7 downto 0) <= cpu_address(7 downto 0); -- get the index and offset from cpu address (index & offset)
                  sram_wen     <= "0"; -- not writing to sram
                  -- check that tag is in tag table and the valid is 1
                  if (valid(to_integer(unsigned(index))) = '1') and (tag = tag_reg(to_integer(unsigned(index)))) then
                     yfsm <= s2; -- found a hit so look for read/write operation
                  else
                     -- check dirty bit and valid
                     if (dirty(to_integer(unsigned(index))) = '1') and (valid(to_integer(unsigned(index))) = '1') then
                        yfsm<=s7; -- load sram block to sdram
                     else
                        yfsm<=s6; -- write new memory block to cache
                     end if;
                  end if;
               when s2 => --s2: found a HIT
                  state<="010";
                  if (cpu_wr_rd_en='1') then
                     yfsm <= s3; -- write from cpu to cache
                  else
                     yfsm <= s4; -- read from cache to cpu
                  end if;
               when s3 =>
```



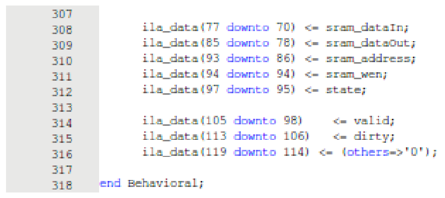
```vhdl
                  state <= "011";
                  sram_wen     <= "1"; -- writing to sram
                  valid(to_integer(unsigned(index))) <= '1'; -- set valid bit
                  dirty(to_integer(unsigned(index))) <= '1'; -- set dirty bit
                  sram_dataIn  <= cpu_dataOut; -- loading the cpu data to the cache block
                  cpu_dataIn   <= "00000000"; -- doesn't matter
                  yfsm<=s0; -- goes to idle
               when s4 => -- s4: reading from cpu to cache
                  state <= "100";
                  cpu_dataIn <= sram_dataOut; -- output cache value to cpu
                  yfsm<=s0; -- go to idle
               when s6=> -- s6: writing new memory block to cache
                  state<="110";
                  if (counter = 64) then -- takes 32 cycles to write to memory
                     counter<=0;
                     valid(to_integer(unsigned(index))) <= '1'; -- set valid to 1
                     tag_reg(to_integer(unsigned(index))) <= tag; -- get tag from tag register
                     mm_offset<=0;
                     yfsm<=s2; -- check for read or
                  else
                     if (counter mod 2 = 1) then -- only access memory on rising edges
                        sdram_memstrb<='0';
                     else
                        sdram_address  <= cache_tag & index & std_logic_vector(to_unsigned(mm_offset, 5)); -- (tag & index & offset)
                        sdram_wr_rd_en <= '0'; -- not writing to sdram
                        sdram_memstrb  <='1';
                        sram_address   <= index & std_logic_vector(to_unsigned(mm_offset, 5)); -- (index & offset)
                        sram_dataIn    <= sdram_dataOut; -- writing sdram block to sram
                        sram_wen       <= "1"; -- enable sram for writing
                        mm_offset <= mm_offset+1; -- increment to next byte
                     end if;
                     counter <= counter + 1; -- increment counter
                  end if;
```



```vhdl
               when s7=> -- s7: write cache block to memory
                  state<="111";
                  if (counter = 64) then -- takes 32 cycles to read from memory
                     dirty(to_integer(unsigned(index))) <= '0'; -- set dirty to 0
                     tag_reg(to_integer(unsigned(index))) <= tag; -- the tag in tag table
                     counter<=0; --reset counters
                     mm_offset<=0;
                     yfsm<=s6; -- load new block from sdram to sram
                  else
                     if (counter mod 2 = 1) then -- writes only on rising edge
                        sdram_memstrb<='0';
                     else
                        sdram_address <= tag & index & std_logic_vector(to_unsigned(mm_offset, 5)); -- write each byte in the block (tag&index&offset)
                        sdram_wr_rd_en <= '1'; -- writing to sdram
                        sram_address <= index & std_logic_vector(to_unsigned(mm_offset, 5)); --  (index&offset)
                        sram_wen <= "0"; -- not writing to cache
                        --sram_dataIn  <= sdram_dataOut; --
                        sdram_dataIn <= sram_dataOut; -- writing cache data to memory
                        sdram_memstrb<='1';
                        mm_offset <= mm_offset+1; -- increment to next byte
                     end if;
                     counter <= counter + 1; -- increment counter
                  end if;
               when s0 => -- s0: cache idle
                  state <= "000";
                  cpu_rdy <= '1'; -- tell cpu cache is idle
                  if (cpu_cs = '1') then -- cpu sends signal
                     yfsm <= s1; -- check for hit/miss
                  end if;
            end case;
         end if;
      end process;
```

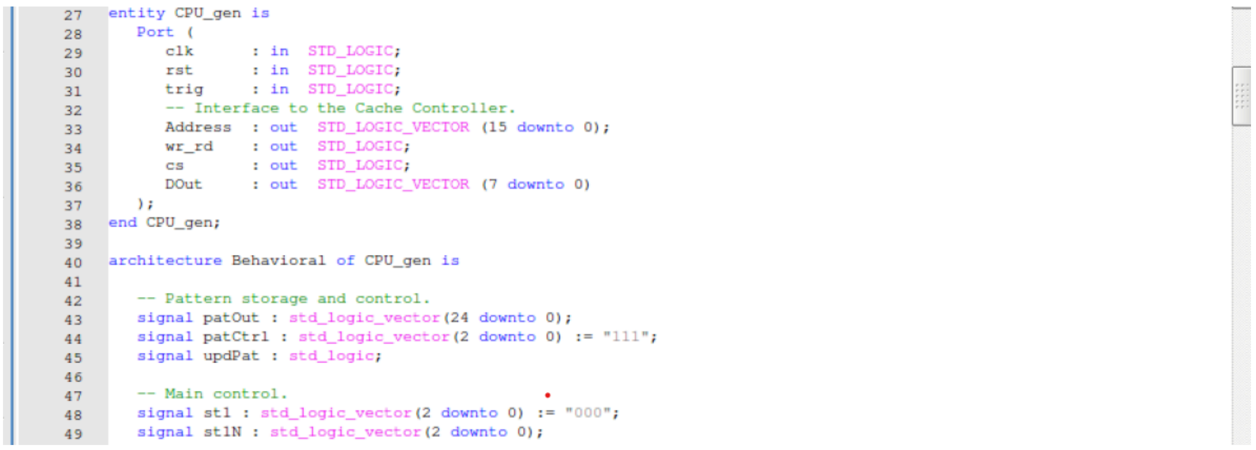


```vhdl
    -- store data in signals to ports
    FSM_STATE <= state;
    CPU_ADDR <= cpu_address;
    CPU_WR_RD <= cpu_wr_rd_en;
    --CS <= cpu_cs;
    CPU_DOUT <= cpu_dataOut;
    CPU_DIN <=  cpu_dataIn;
    RDY <= cpu_rdy;
    SDRAM_MEM_ADDR <= sdram_address;
    SDRAM_MEM_WR_RD <= sdram_wr_rd_en;
    MEMSTRB <= sdram_memstrb;
    SDRAM_DIN <= sdram_dataIn;
    SDRAM_DOUT <= sdram_dataOut;
    SRAM_DIN <= sram_dataIn;
    SRAM_DOUT <= sram_dataOut;
    SRAM_ADDR <= sram_address;

    -- map ports to ila data
    ila_data(15 downto 0) <= cpu_address;
    ila_data(16) <= cpu_wr_rd_en;
    ila_data(17) <= cpu_cs;
    ila_data(18) <= '0';
    ila_data(19) <= cpu_rdy;
    ila_data(27 downto 20) <= cpu_dataOut;
    ila_data(35 downto 28) <= cpu_dataIn;

    ila_data(36) <= sdram_memstrb;
    ila_data(52 downto 37) <= sdram_address;
    ila_data(53)<= sdram_wr_rd_en;
    ila_data(61 downto 54) <= sdram_dataIn;
    ila_data(69 downto 62) <= sdram_dataOut;
```

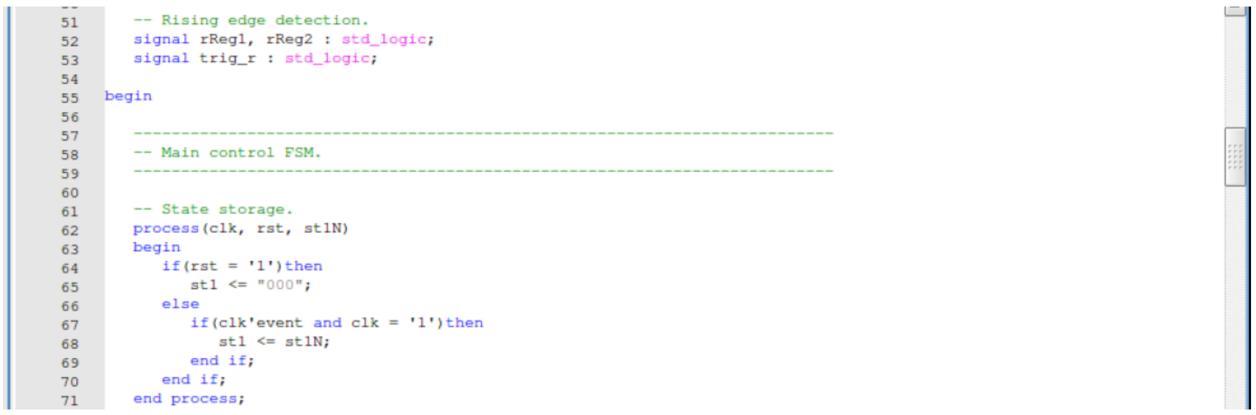


```vhdl

    ila_data(77 downto 70) <= sram_dataIn;
    ila_data(85 downto 78) <= sram_dataOut;
    ila_data(93 downto 86) <= sram_address;
    ila_data(94 downto 94) <= sram_wen;
    ila_data(97 downto 95) <= state;

    ila_data(105 downto 98)    <= valid;
    ila_data(113 downto 106)   <= dirty;
    ila_data(119 downto 114) <= (others=>'0');

end Behavioral;
```

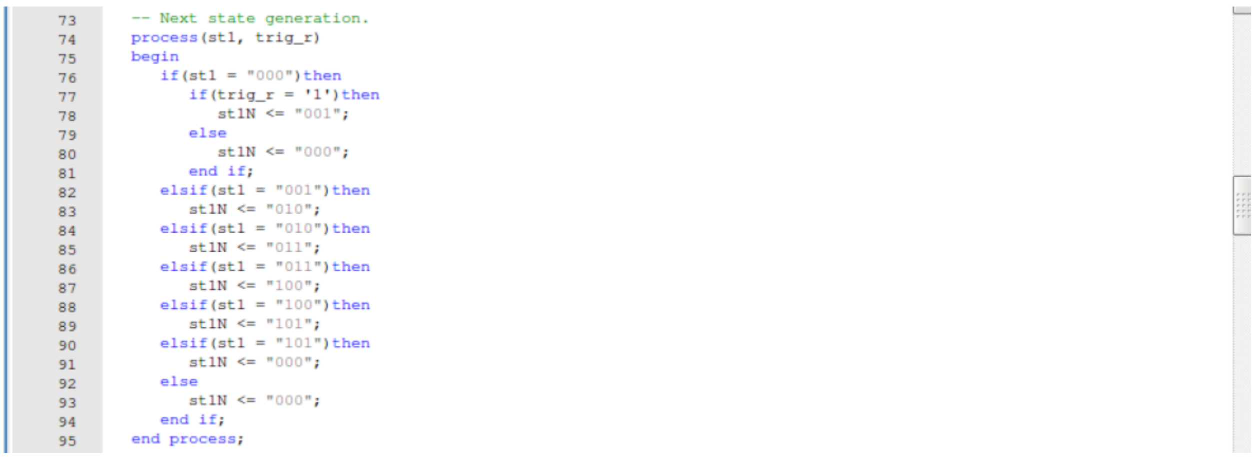
CPU_GEN



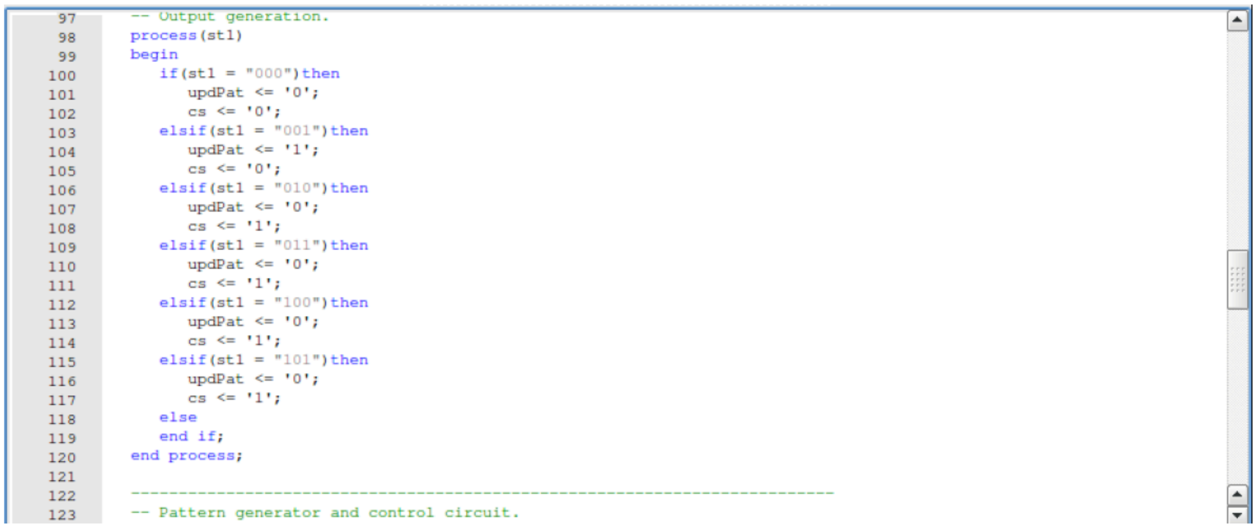
```vhdl
entity CPU_gen is
   Port (
      clk       : in  STD_LOGIC;
      rst       : in  STD_LOGIC;
      trig      : in  STD_LOGIC;
      -- Interface to the Cache Controller.
      Address   : out  STD_LOGIC_VECTOR (15 downto 0);
      wr_rd     : out  STD_LOGIC;
      cs        : out  STD_LOGIC;
      DOut      : out  STD_LOGIC_VECTOR (7 downto 0)
   );
end CPU_gen;

architecture Behavioral of CPU_gen is

   -- Pattern storage and control.
   signal patOut : std_logic_vector(24 downto 0);
   signal patCtrl : std_logic_vector(2 downto 0)  := "111";
   signal updPat : std_logic;

   -- Main control.
   signal st1 : std_logic_vector(2 downto 0)  := "000";
   signal st1N : std_logic_vector(2 downto 0);
```



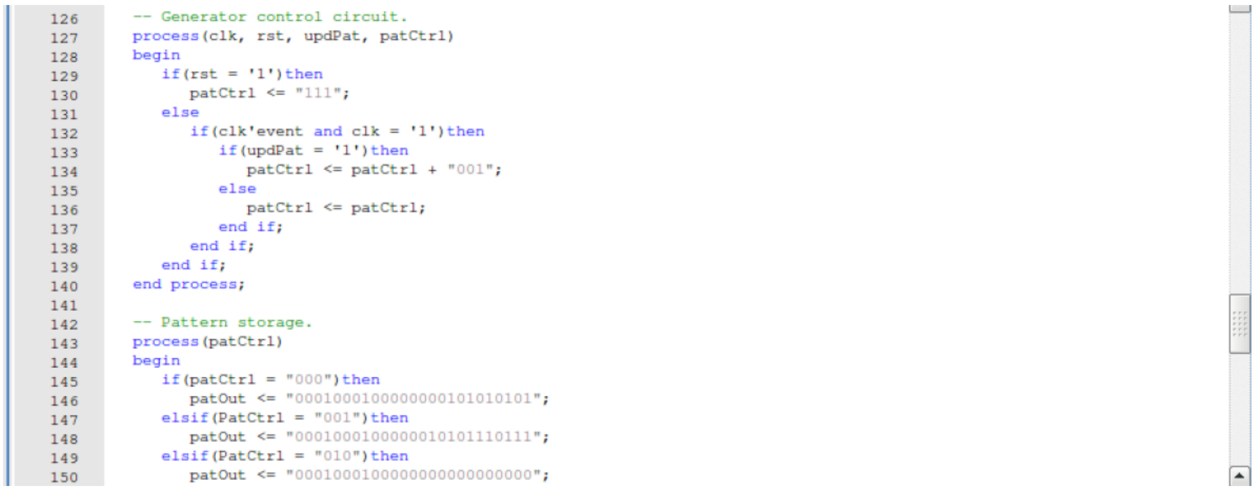
```vhdl
   -- Rising edge detection.
   signal rReg1, rReg2 : std_logic;
   signal trig_r : std_logic;

begin

   ---------------------------------------------------------------------------
   -- Main control FSM.
   ---------------------------------------------------------------------------

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



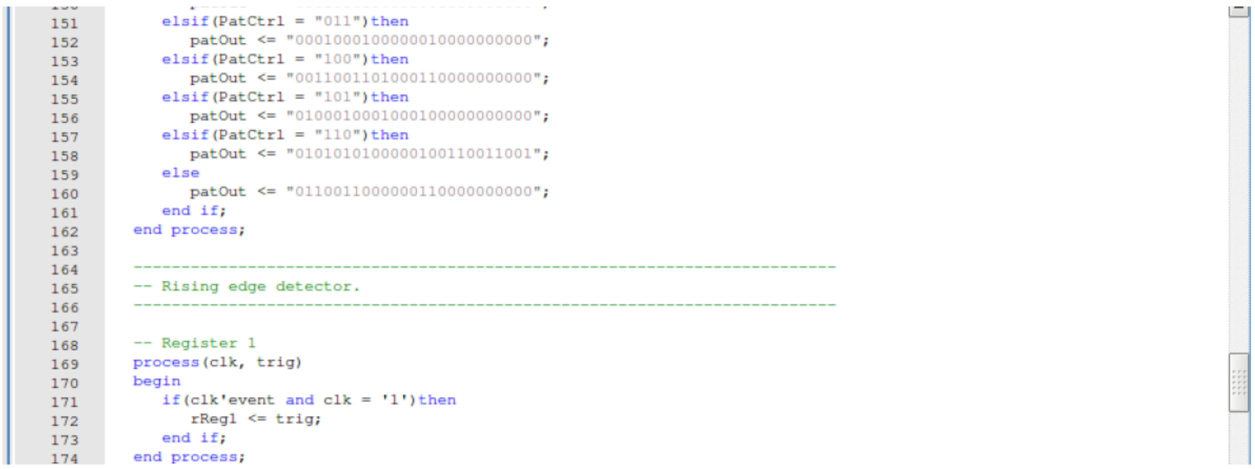
```vhdl
-- Next state generation.
process(stl, trig_r)
begin
   if(stl = "000")then
      if(trig_r = '1')then
         stlN <= "001";
      else
         stlN <= "000";
      end if;
   elsif(stl = "001")then
      stlN <= "010";
   elsif(stl = "010")then
      stlN <= "011";
   elsif(stl = "011")then
      stlN <= "100";
   elsif(stl = "100")then
      stlN <= "101";
   elsif(stl = "101")then
      stlN <= "000";
   else
      stlN <= "000";
   end if;
end process;
```



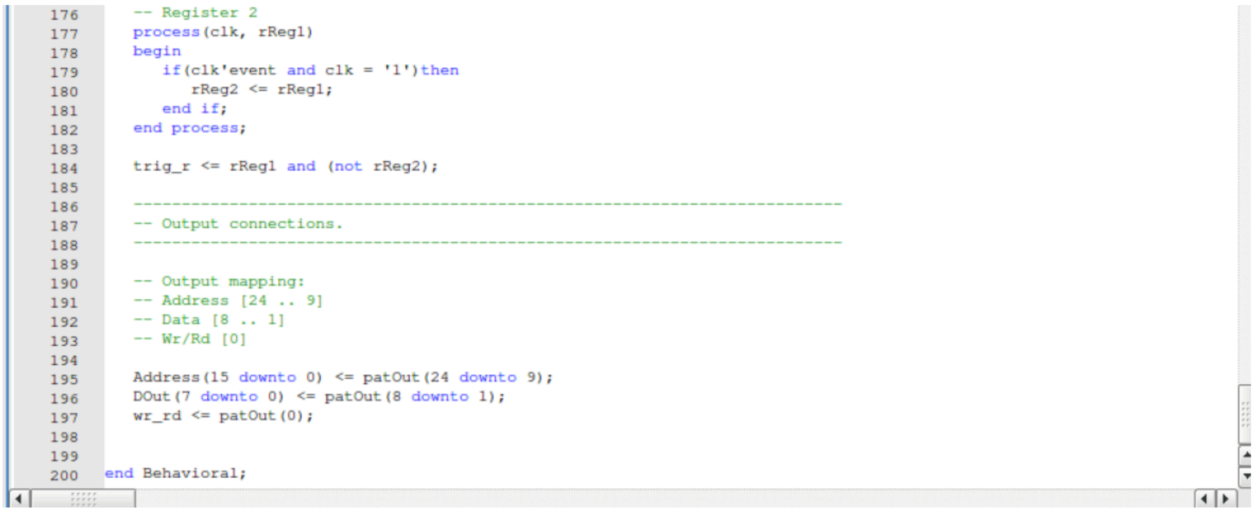
```vhdl
-- Output generation.
process(stl)
begin
   if(stl = "000")then
      updPat <= '0';
      cs <= '0';
   elsif(stl = "001")then
      updPat <= '1';
      cs <= '0';
   elsif(stl = "010")then
      updPat <= '0';
      cs <= '1';
   elsif(stl = "011")then
      updPat <= '0';
      cs <= '1';
   elsif(stl = "100")then
      updPat <= '0';
      cs <= '1';
   elsif(stl = "101")then
      updPat <= '0';
      cs <= '1';
   else
   end if;
end process;

--------------------------------------------------------------------------
-- Pattern generator and control circuit.
```



```vhdl
-- Generator control circuit.
process(clk, rst, updPat, patCtrl)
begin
   if(rst = '1')then
      patCtrl <= "111";
   else
      if(clk'event and clk = '1')then
         if(updPat = '1')then
            patCtrl <= patCtrl + "001";
         else
            patCtrl <= patCtrl;
         end if;
      end if;
   end if;
end process;

-- Pattern storage.
process(patCtrl)
begin
   if(patCtrl = "000")then
      patOut <= "0001000100000000101010101";
   elsif(PatCtrl = "001")then
      patOut <= "0001000100000010101110111";
   elsif(PatCtrl = "010")then
      patOut <= "0001000100000000000000000";
```

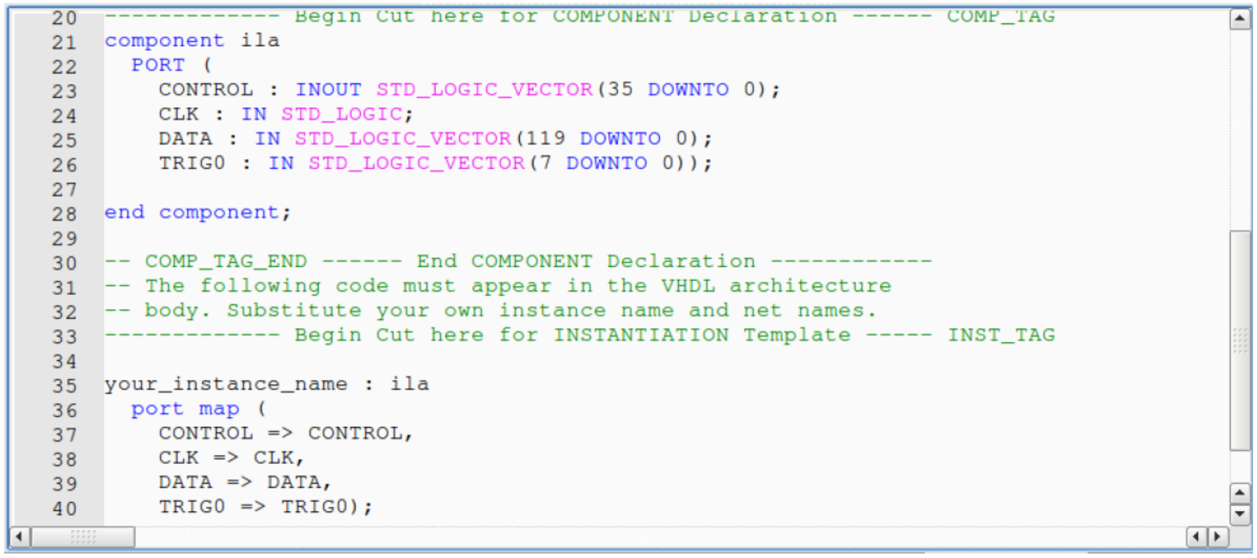


```
      elsif(PatCtrl = "011")then
         patOut <= "0001000100000001000000000";
      elsif(PatCtrl = "100")then
         patOut <= "0011001101000110000000000";
      elsif(PatCtrl = "101")then
         patOut <= "0100010001000100000000000";
      elsif(PatCtrl = "110")then
         patOut <= "0101010100000100110011001";
      else
         patOut <= "0110011000000110000000000";
      end if;
   end process;

   --------------------------------------------------------------------------
   -- Rising edge detector.
   --------------------------------------------------------------------------

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



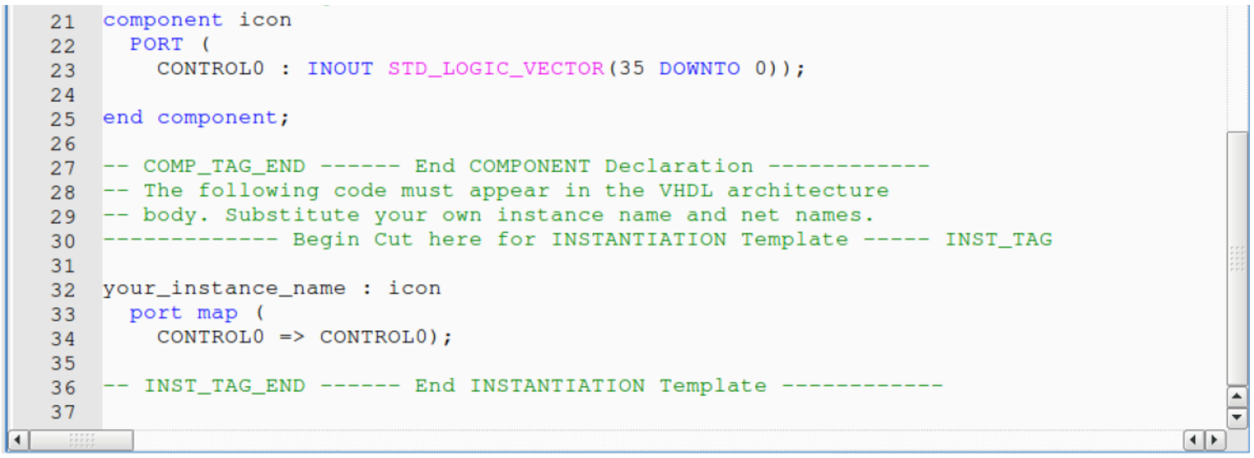
```
   -- Register 2
   process(clk, rReg1)
   begin
      if(clk'event and clk = '1')then
         rReg2 <= rReg1;
      end if;
   end process;

   trig_r <= rReg1 and (not rReg2);

   --------------------------------------------------------------------------
   -- Output connections.
   --------------------------------------------------------------------------

   -- Output mapping:
   -- Address [24 .. 9]
   -- Data [8 .. 1]
   -- Wr/Rd [0]

   Address(15 downto 0) <= patOut(24 downto 9);
   DOut(7 downto 0) <= patOut(8 downto 1);
   wr_rd <= patOut(0);


end Behavioral;
```

ILA



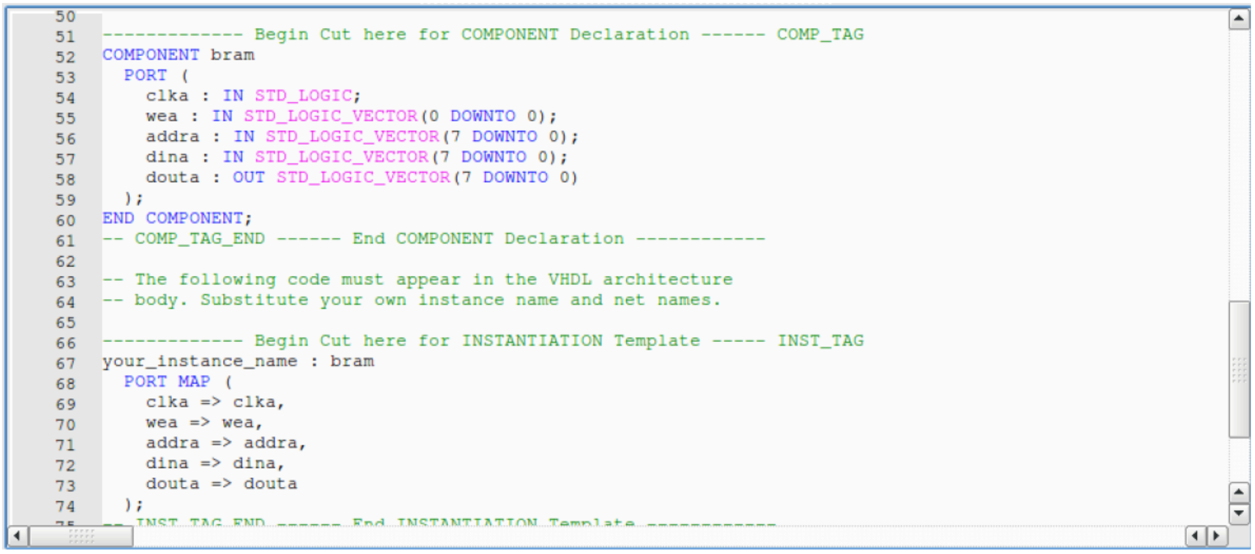
```
------------- Begin Cut here for COMPONENT Declaration ------ COMP_TAG
component ila
  PORT (
    CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    CLK : IN STD_LOGIC;
    DATA : IN STD_LOGIC_VECTOR(119 DOWNTO 0);
    TRIG0 : IN STD_LOGIC_VECTOR(7 DOWNTO 0));

end component;

-- COMP_TAG_END ------ End COMPONENT Declaration ------------
-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.
------------- Begin Cut here for INSTANTIATION Template ----- INST_TAG

your_instance_name : ila
  port map (
    CONTROL => CONTROL,
    CLK => CLK,
    DATA => DATA,
    TRIG0 => TRIG0);
```

ICON



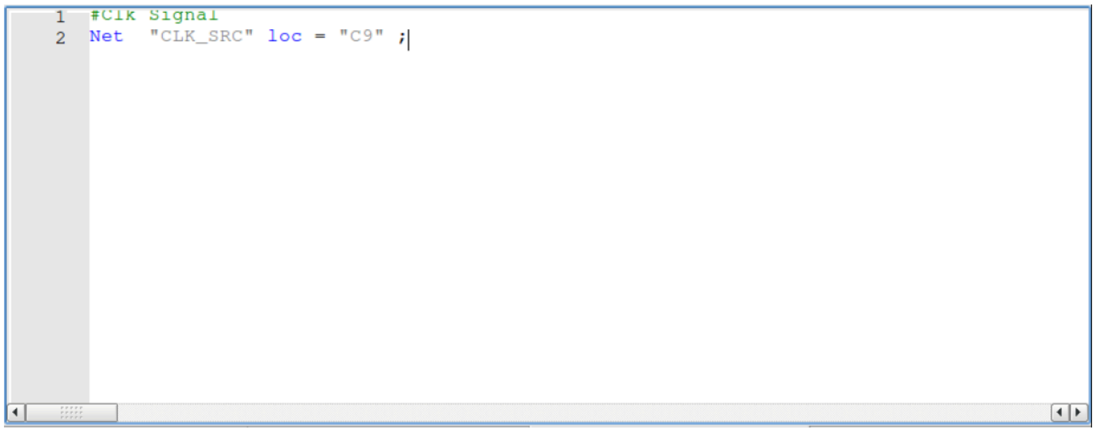
```
component icon
  PORT (
    CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0));

end component;

-- COMP_TAG_END ------ End COMPONENT Declaration ------------
-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.
------------- Begin Cut here for INSTANTIATION Template ----- INST_TAG

your_instance_name : icon
  port map (
    CONTROL0 => CONTROL0);

-- INST_TAG_END ------ End INSTANTIATION Template ------------
```

BRAM



```

------------- Begin Cut here for COMPONENT Declaration ------ COMP_TAG
COMPONENT bram
  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;
-- COMP_TAG_END ------ End COMPONENT Declaration ------------

-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.

------------- Begin Cut here for INSTANTIATION Template ----- INST_TAG
your_instance_name : bram
  PORT MAP (
    clka => clka,
    wea => wea,
    addra => addra,
    dina => dina,
    douta => douta
  );
-- INST_TAG_END ------ End INSTANTIATION Template ------------
```

CACHE CONSTRAINTS



```
#Clk Signal
Net   "CLK_SRC" loc = "C9" ;
```