Ryerson University

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

Course Title:	Digital systems
Course Number:	COE 758
Semester/Year (e.g.F2016)	F2023

Instructor:	Dr. Vadim Geurkov

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

Submission Date:	November 3rd 2023
Due Date:	November 3rd 2023

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Akhtar	Abbad	500953405	12	A.A
Melegrito	Nyle Fernan	500974255	12	N.M

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

Table of contents:

Abstract-

The design project aims to develop a Cache Controller in VHDL for enhancing the performance and efficiency of memory access in embedded systems and microprocessor architectures. While caches play a pivotal role in bridging the gap between the CPU and main memory by storing frequently accessed data and data that have a high probability of being used, they are only memory modules. Most of the logic that determines what data and actions are required to process a CPU request from the cache is done by the cache controller. This project focuses on creating a robust and flexible cache controller to manage data in the cache memory.

The approach we will be focusing on is separated by Cache organization and Cache interfacing. The cache organization is designed using the direct mapped addressing method taught in the lectures. The cache is implemented using a Block RAM (BRAM). It has 8 blocks/entries accessible with a 3-bit index, with each block containing 32 addressable words that are accessed with a 5-bit offset. The cache interface involves an address bus from the cache controller, data buses that go in and out of the cache, and a read/write signal from the cache controller. With its limited interface it depends on the cache controller which has a more complex interface that controls the flow of data in and out of the cache. When the CPU wants to access data it sends an address to the cache controller where the controller separates it into a tag, index and address. The cache controller then sends the index portion to its own private service BRAM module to find the associated tag and valid and dirty bits. It then compares the tag given by the CPU and the saved tag associated with the given index. The cache controller checks for a match and a valid bit for a hit, then the dirty bit if it is a miss. The cache controller then sets the correct signals to the surrounding components to properly process the request. For example, setting the read or write signals for the cache and SDRAM controller. More detail will be covered in the later sections.

We ended up with a correctly functioning cache controller that was able to handle requests from the given CPU_gen file that imitated a real CPU in terms of cache interactions.

Introduction-

The goal of this design project is to implement a cache controller, this is a fundamental idea in digital systems because speed and efficiency is what progresses technology, understanding the foundation of how a cache works and why it is important is best seen in practice. This also sheds light on the memory hierarchy and why it is there in the first place. Other important theories that will be discussed are hit and misses, direct mapped addressing, blocks, read and write operations.

Every computer works off of the same principle, where the cpu sends instructions throughout the motherboard in order to accomplish certain tasks, a vital part of this process is memory. Every microprocessor or computer needs memory to either hold instructions or data to use in computations. The ideal memory module would be one that holds infinite data and instantaneous data transfer. However in real life, holding more data while being fast is expensive. Therefore, most computers use slower and physically bigger memory modules to hold large amounts of data as a cheaper alternative. The problem is the computer would not be able to take advantage of the extremely fast CPU when it is held back by a slower memory module. Also a physically larger memory module cannot be put near or inside the CPU, and the physical distance away from the CPU adds to more delay. The solution is a memory hierarchy that

puts faster memory modules that hold less data between the CPU and a larger and slower memory module.

The cache is the closest to the CPU and the fastest out of the whole memory hierarchy, excluding CPU registers. Since the cache has such limited space, it must only store data that has a high chance of being requested by the CPU to either read or edit. The cache takes advantage of spatial locality and temporal locality. Spatial locality is the idea that when an item is referenced, items near will soon be referenced as well. The cache takes advantage of this by storing 32 consecutive words into a block. Meaning when one word in a block is accessed, there is a good chance another word in the same block will be referenced as well. Temporal locality is the idea that when an item is referenced, it will soon be accessed again. The cache takes advantage of this by storing tags to quickly refer to blocks of data and storing data requested by the CPU until it needs to be replaced by newly requested data. Another fundamental point is that latency increases as distance increases from memory to processor (referring to the memory hierarchy) knowing this, the cache is also placed close to the processor.

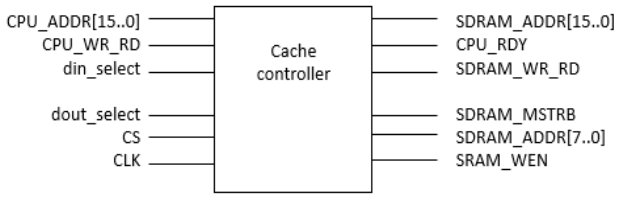
The cache controller is what allows the cpu to interface with the cache. It is responsible for calculating and retrieving direct mapped addresses. This is a method that separates the cache into blocks and places things in memory into these blocks in the cache. We use this formula (Block address)MOD(#of blocks in the cache) to find the block of interest and then we use the lowest n bits to address them in the cache (256 entries = 2^3 , n = 3). So if we had an address 10110 the index would be 110 and the tag would be 10, we can use this to recover the data in the cache. In the design of our cache controller we had our tag to be 6 bits, index to be 3 bits, block offset to be 5 bits and address to be 14 bits. Along with the cache controller there are other components in the design such as the service SRAM and SDRAM controller.

A cache controller is required because the cache only stores data and outputs the required data. To enable its quick data transfers, it must not be hindered by computing logic and other mechanisms like figuring out where to send data to, where to receive data from, or comparing tags. This will all be done by the cache controller.

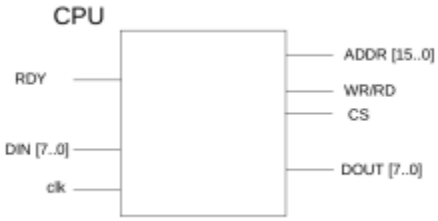
System specification-

Symbols:

Cache controller:

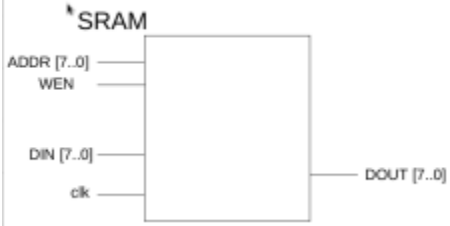


CPU:



SDRAM controller:

SRAM:



Note : our newly implemented design of the cache controller our address is split up as follows -> Tag = 6 bits Index = 3 bits and Block offset = 5 bits. All address buses for each component are actually [13..0] to reflect the change of making the Tag 6 bits instead of 8. Also the CPU_gen.vhd file given does not actually have a DIN as one of its inputs as it is not a real CPU, although our cache controller treats it as such.

Block diagram:

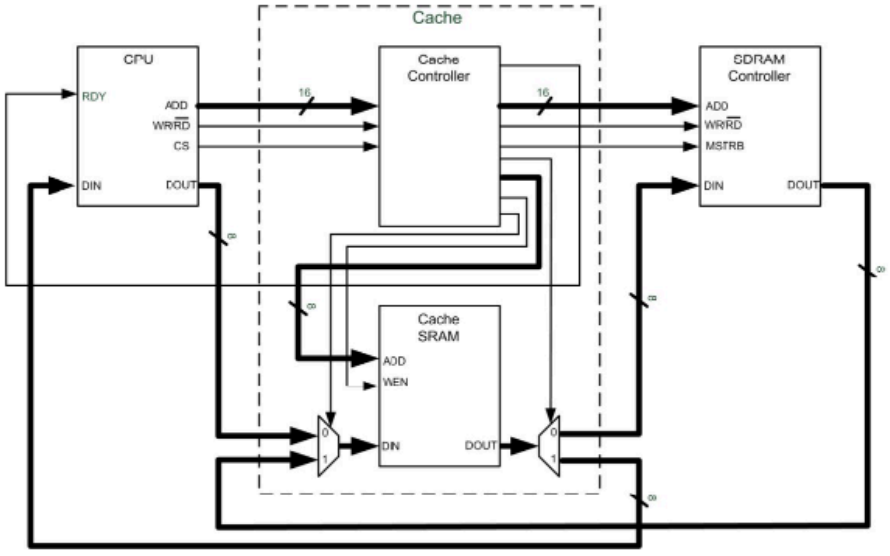
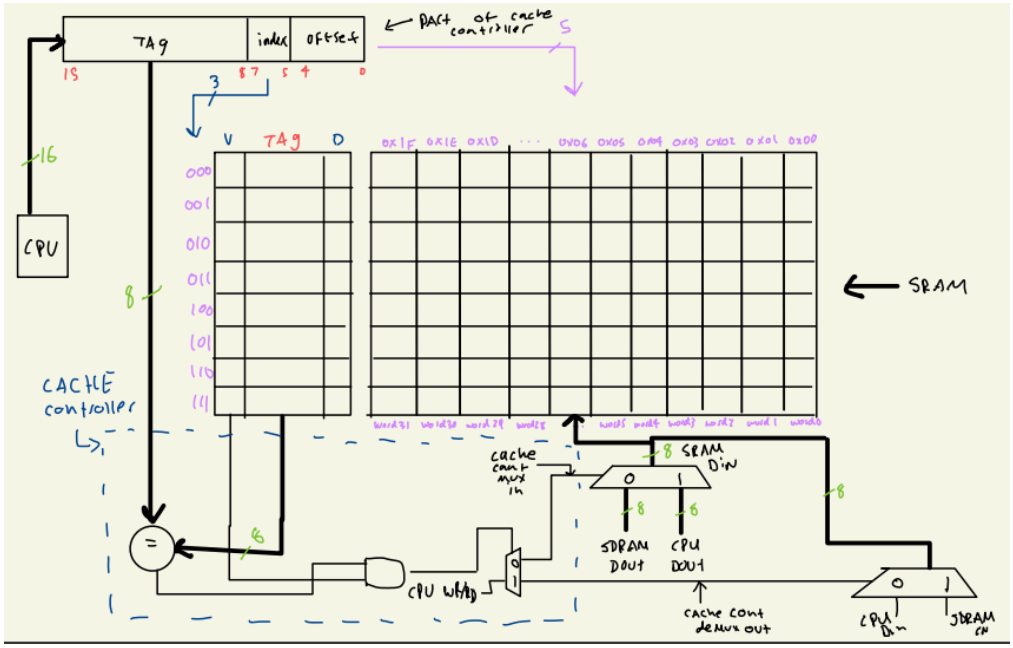
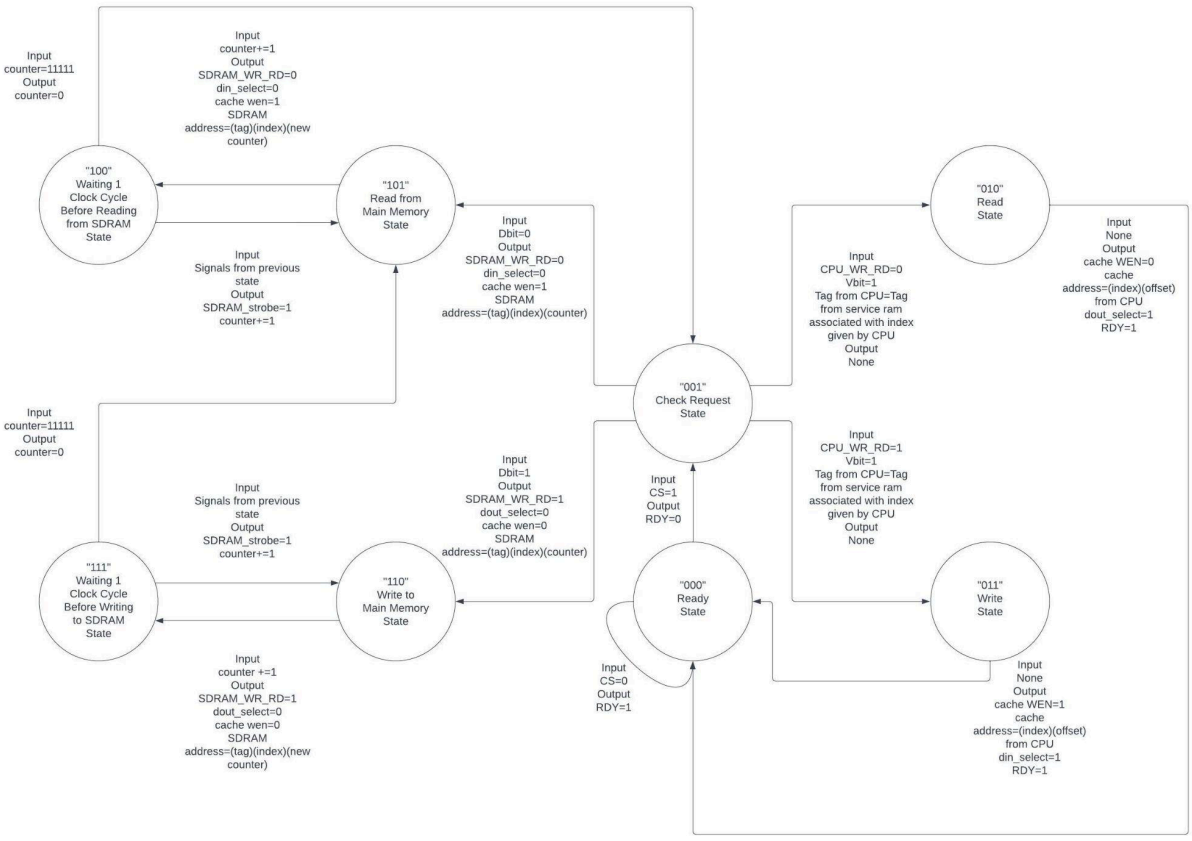


Figure 3: Complete Cache System

Process diagram:



State diagram:



Note: More details on each state in the next section.

Device Description/ design-

Ready State

The cache controller is waiting for a CPU request. If there is no CPU request it makes sure that the RDY signal is set to let the CPU know that the cache is ready for another request. The cache controller knows there is a CPU request when the CS signal is set, in which the cache controller moves on to the Check Request State.

Check Request State

The cache controller sends the index it decoded from the whole CPU address it received to its service BRAM. The BRAM gives the associated tag and valid and dirty bits if it exists. The cache controller then takes this tag and compares it with the tag it decoded from the CPU address. If the tags match and the valid bit is 1, then the request is deemed a hit. If the tag does not exist, does match, or the valid bit is 0 - then the request is deemed a miss.

When a hit occurs, the CPU_WR_RD signal from the CPU is checked. If the signal is a high 1, then it is a write request and the cache controller moves to the Write state. If the signal is a low 0, then it is a read request and the cache controller moves to the Read state.

When a miss occurs, the dirty bit from the service BRAM is checked. If it does not exist or is 0, then the cache does not need to write anything back to main memory and can go straight to the Read from Main Memory state. Contrary if the dirty bit is 1, then that means the data for associated address has been changed and this change must be reflected back to main memory. The cache controller must tell the cache to write back this data back to main memory, moving to the Write to Main Memory state.

Read State

The cache controller sets the cache wen signal to 0 and sends it the index and offset portion taken from the given CPU address. The cache then takes this address and outputs the corresponding data. The cache itself does not know where the data goes but just outputs it. The data goes to a demux that directs the data to either the CPU or to the SDRAM controller. In this case, the cache controller sets the dout_select signal to 0 to direct the data back to the CPU. The read request is completed and the cache controller goes back to the Ready state.

Write State

The cache controller sets the cache wen signal to 1 and sends it the index and offset portion taken from the given CPU address. The cache would then accept data coming in and storing it into the corresponding address. Since the cache cannot choose where the data comes from, it takes data from a mux. The mux directs data either from the CPU or the SDRAM into the cache. In this case, the cache controller sets the din_select signal to 1 to direct the CPU data into the cache. The write request is completed and the cache controller goes back to the Ready state.

Read from Main Memory State

Data from the SDRAM must be read into the cache when the requested data is not found in the cache. The cache controller takes the address given by the CPU and relays it to the SDRAM controller. However, it does set the offset portion to “00000” first. The SDRAM_WR_RD signal is set low to 0 to read data from main memory. To direct data from main memory to the cache, the cache controller sets the din_select signal to 0. The cache wen signal is also set to 1 enabling writing data into. The index and offset portion from the same CPU address given to the SDRAM controller. This associates directly maps the current data being written to the appropriate address. The signals mentioned must also be held stable for 1 clock cycle, therefore the cache controller moves on to the Waiting 1 Clock Cycle Before Reading from

SDRAM state. Since only one word can be written at a time this process is repeated 32 times to write a whole new block into the cache.

Write from Main Memory State

This state occurs when the requested data is not found in the cache but the block the index refers to has been edited or written into prior - dirty bit = 1. The cache must write this back to the SDRAM before changing the data inside the cache. The cache wen is set to 0 to read data from it, while supplying the index given by the CPU address and the offset set to “00000”. The dout_select signal is set to 0 to direct data from the cache to the SDRAM controller. Similar to the previous state, the signals must be held stable for 1 clock cycle before trying to access the SDRAM and another state is used to fulfill this purpose - Waiting 1 Clock Cycle Before Writing to SDRAM state. This process must also be repeated 32 times to completely transfer each word in one block into the SDRAM.

Waiting 1 Clock Cycle Before Reading from SDRAM State

This state sets the main memory strobe to 1 after the signals in the previous state have been held for 1 clock cycle. The current word is read from main memory and the offset portion of the address is incremented by 1. This goes back to the Read from Main Memory state to repeat the process as needed. When the whole block has been written into the cache, the cache controller goes back to the Check Request state to attempt processing the initial CPU request again.

Waiting 1 Clock Cycle Before Writing to SDRAM State

This state sets the main memory strobe to 1 after the signals in the previous state have been held for 1 clock cycle. The current word is written into main memory and the offset portion if the address is incremented by1. This goes back to the Write to Main Memory state to repeat the process as needed. When the whole block has been written into main memory, the cache controller goes to the Read from Main Memory state. This happens because writing back to main memory only updates the data, but the request data must still be fetched to service the initial CPU request.

This design uses the CLK signal to synchronize all the components. The SDRAM just takes an address from the cache controller and read or write signal. The SDRAM is then implemented with an array that has 16, 384 (2^14) cells that each hold an 8-bit word. The demo.vhd file combines all the components together to simulate a working memory hierarchy and to demonstrate our project. The component interfaces are mapped exactly as seen in the block diagram. The SRAM cache and the service ram inside the cache controller uses the BRAM Chipscope core we learned about in tutorial 2.

Results-

N	Cache performance parameter	Time in nS
1	Hit / Miss determination time	10
2	Data access time	20
3	Block replacement time	640
4	Hit time (case 1 and 2)	20

5	**Miss penalty for Case 3 (D bit = 0)**	**690**
6	**Miss penalty for Case 4 (D bit = 1)**	**1350**

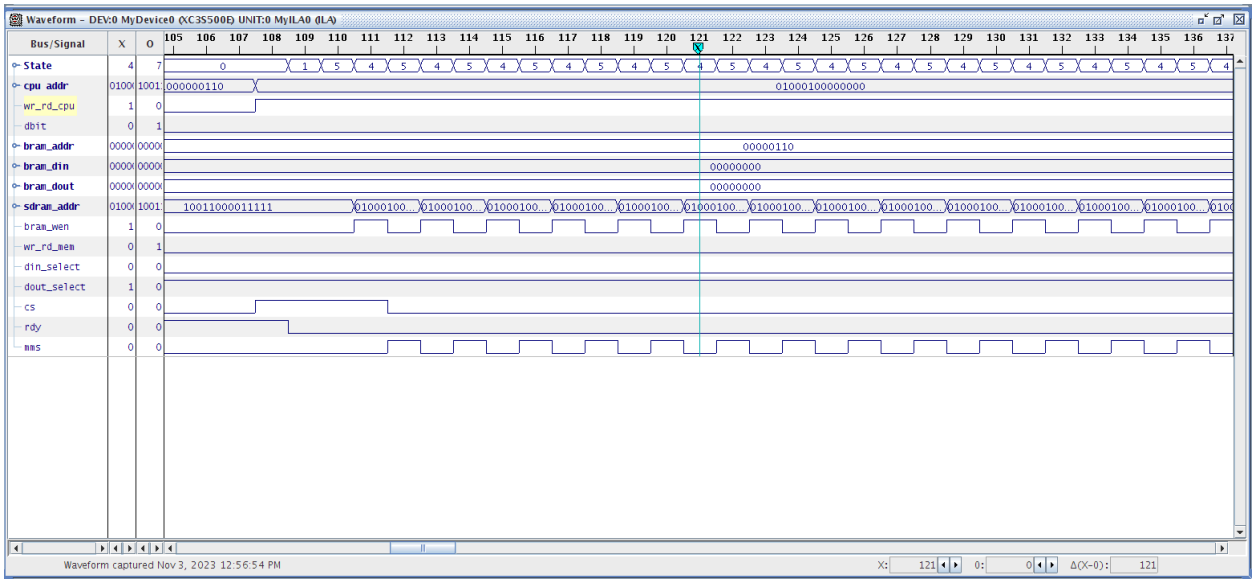
CPU Frequency : 50MHz
CPU period : 20ns
Data access time : subtracting a miss read from a miss write = 20ns
Block replacement time : each state interval is 10ns and it switches between states 32 times (32 state 5 + 32 state 4) = 64*10 = 640ns
Hit time : when a hit read request is sent going from state 0 back to state 0 takes 20 state switches so 20ns
Miss penalty for case 3 : hit determination time + state length(sum of #of sums* their length in time) + HIT write/read = 690ns
Miss penalty for case 4 : hit determination time + state length(sum of #of sums* their length in time) + HIT write/read = 1350ns

Timing diagrams-

6 Cases:

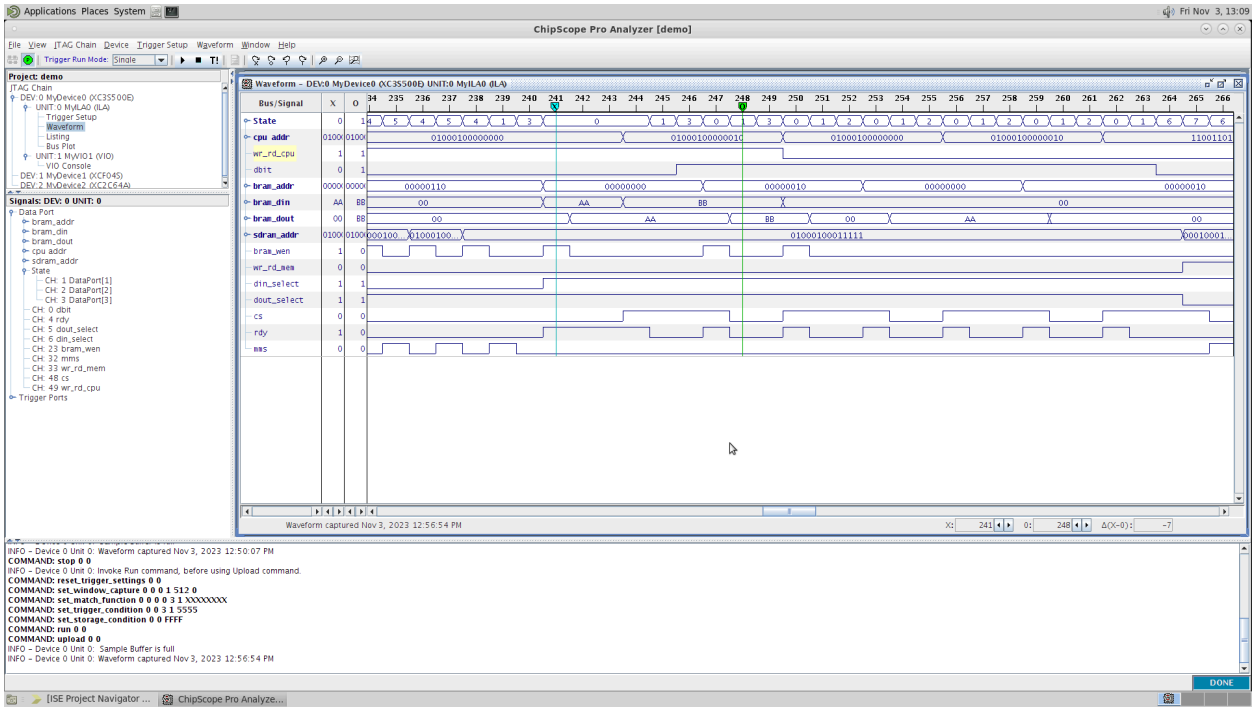
- Write to cache when there is a hit
- Read from cache when there is a hit
- Read when a miss and D bit = 0
- Write when a miss and D bit = 0
- Read when a miss and D bit = 1
- Write when a miss and D bit = 1

Write request miss with dbit=0



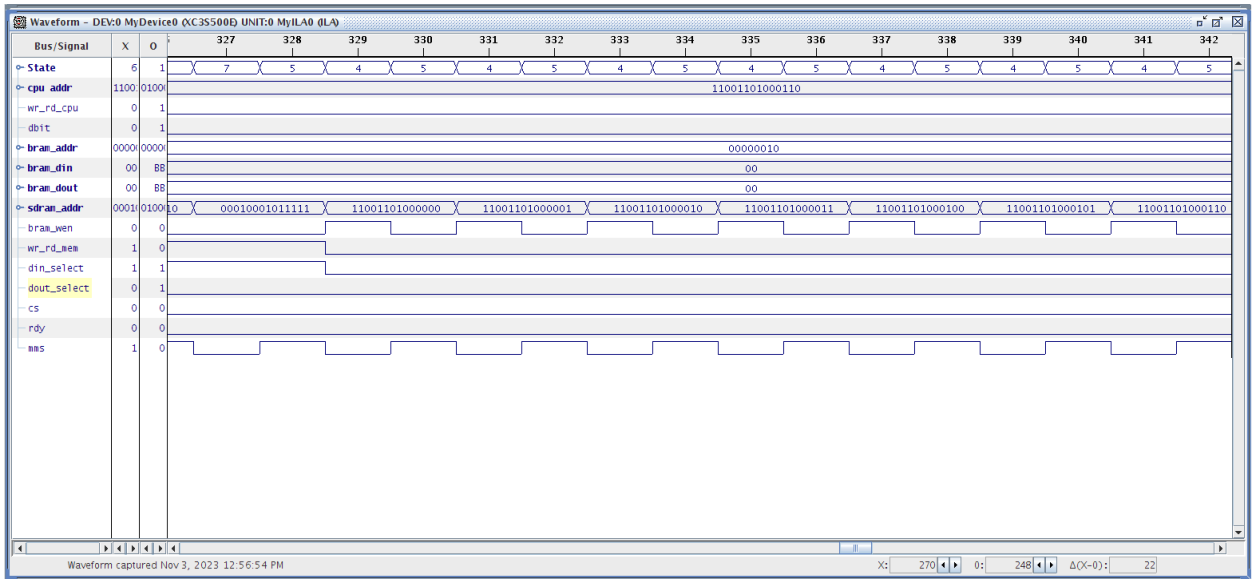
Since this is the first request, the cache is empty. This results in a miss with dbit=0 since no data has been changed. The state is seen to cycle above between 5 and 4 which are the read from main memory state and its corresponding delay state. The SDRAM address is set to the same CPU address given except the

offset portion was set to “00000”. Then we see it get incremented by one each time to read 32 words into one block. BRAM (cache) wen is set to 1 periodically to write each block and the read/write signal for the main memory is set to 0 for a read. The din_select is also set to 0 to direct SDRAM data to the cache.

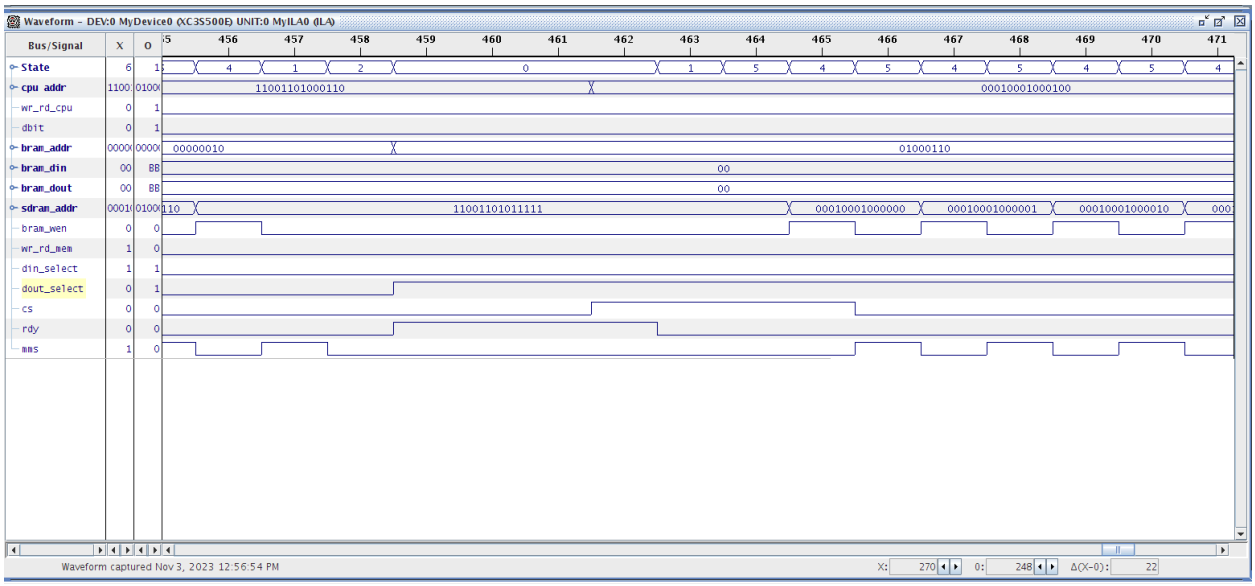


After reading the whole block into the cache the initial request is serviced again. The BRAM (cache) address is set to the index and offset portion of the CPU address, BRAM wen is set to 1 to write. The din_select is 1 to direct data from the CPU to the cache. The din data is seen to be 0xAA which is correct when checking the CPU_gen.vhd file. The next instruction is also process and becomes a hit because the index with the matching tag has already been written into the cache from the previous instruction. The same signals are set but the offset is different and the data written is seen to be 0xBB. The next two requests are also seen and are read requests for the previous two addresses. This is a hit because tag matches and already exists in cache. We see 0xAA in the dout data section, with BRAM wen set to 0 and dout_select set to 1 to direct cache data to the CPU.

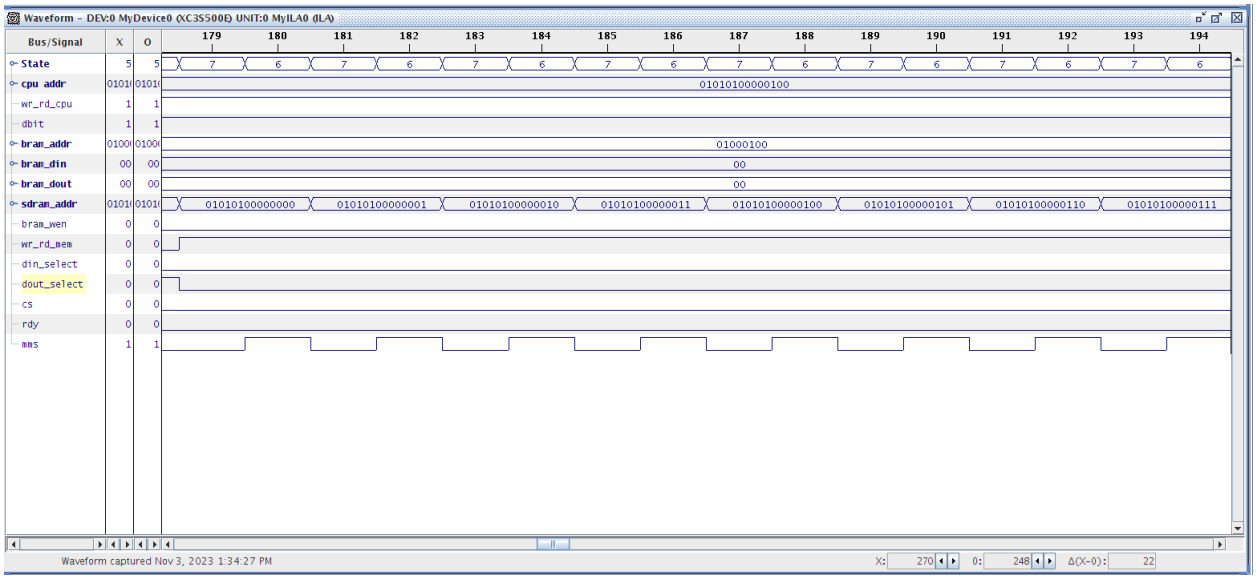
Read miss with dbit=0



This is the 5th instruction from the CPU. It tries to read from an address that is not in the cache yet, so it becomes a miss. We see the wr_rd_cpu and dbit signal as zero in the waveform. The sdram_addr is being incremented by 1 for the offset and writing each word one by one. The mms and bram_wen signals are alternating between 1 and 0, to write properly and make sure the signals are stable before data enters either the SDRAM or cache. The bram_din is 0x00 because I initialized the SDRAM with all zero values. This alternation between state 4 and 5 (read from main state and waiting 1 clock cycle) repeats 32 times.



We the state go back to state 1 then 2 (check request state then the read state), after reading from main memory the cache is able to fulfill the CPU's initial read request. It goes back to state 0 (ready state) and takes the next CPU instruction and we see the rdy signal set to 1. The 6th instruction is another read request from the same index, but with a different tag. It goes to read from main memory again, with dbit=0 because the CPU did not alter the data from the previous instruction.



This is the 7th instruction from the CPU. It is a write instruction as we the wr_rd_cpu signal is set to 1. Since the CPU address has the same index as the first 4 instructions but a different tag, the data from the first 2 write instructions must be written back to the SDRAM. We see dbit=1 and the alternation between states 6 and 7 (writing to main memory state and waiting 1 clock cycle). Bram_wen=0 to read from the cache, wr_rd_mem is set to 1 to write to main memory, and dout_select=0 to direct cache data into the SDRAM controller. The sdram_addr is all being incremented by 1 each write until the whole block is written.

The next three images involve the last CPU instruction where it gives the same index but different tag. Since the last instruction for that index was a write, dbit = 0. It first writes to main memory, then reads from it. Then it finally services the last CPU read instruction before going back to the ready state.

Some Errors in the Timing Diagrams

For parts that have to do with writing to and reading from main memory, you will notice that the bram_addr is wrong. This is because in the code, we forgot to feed the index and offset portions to the cache address port in the cache controller. Below you see us only set the address being fed to the SDRAM controller but not the cache.

```
elsif(state = "101") then --read from main mem
    rdy <= '0';
    mms <= '0';
    addr_out(13 downto 8) <= tag;
    addr_out(7 downto 5) <= index;
    addr_out(4 downto 0) <= counter;
    wen <= "1"; --write to cache
    wrRDmem <= '0'; --read from main memory
    din_select <= '0'; --direct main memory data to cache

    sBram_wen <= "0";
    next_state <= "100";
--*********************************************************************************************
elsif(state = "110") then --write to main mem
    rdy <= '0';
    sBram_wen <= "0"; --read from service bram to get tag to be replaced
    mms <= '0';
    addr_out(13 downto 8) <= sBram_dout(5 downto 0);
    addr_out(7 downto 5) <= index;
    addr_out(4 downto 0) <= counter;
    wen <= "0"; --read from cache
    wrRDmem <= '1'; --write to main memory
    dout_select <= '0'; --direct cache data to main memory

    next_state <= "111";
```

However, when reading from and writing to cache when it is a hit the bram_addr is set correctly. Below you see us include the section where we set the cache address.

```
elsif(state = "010") then --process read request
    --ready
        dout_select <= '1'; --direct cache data to cpu
        wen <= "0";           --read from cache
        mms <= '0';

        cacheAddr(7 downto 5) <= index;
        cacheAddr(4 downto 0) <= offset;

        sBram_wen <= "0";
        next_state <= "000";
        rdy <= '1';
--*****************************************************************************************
elsif(state = "011") then --process write request
    --write
        wen <= "1"; --write to cache
        mms <= '0';
        din_select <= '1'; --direct cpu data to cache
        cacheAddr(7 downto 5) <= index;
        cacheAddr(4 downto 0) <= offset;

        --update tag info
        sBram_wen <= "1";
        sBram_din(7) <= '1'; --sets dbit to 1
        sBram_din(6) <= '1';    --sets vbit to 1
        sBram_din(5 downto 0) <= tag;

        next_state <= "000";
        rdy <= '1';
--*****************************************************************************************
```

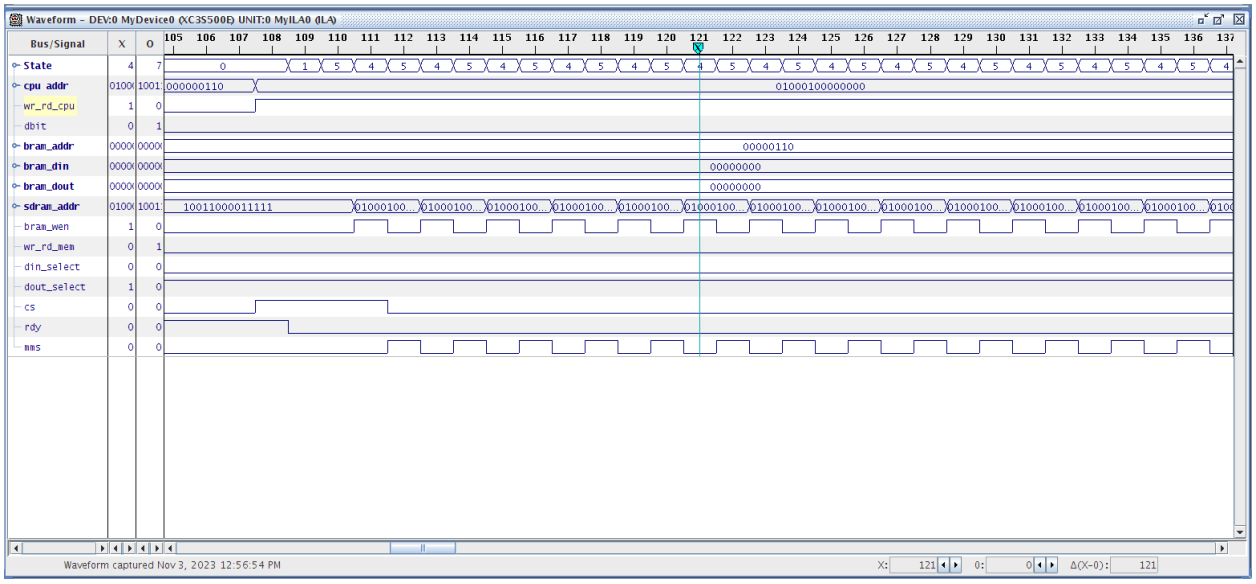
Otherwise most signals are correctly set according to the request.

Conclusions-

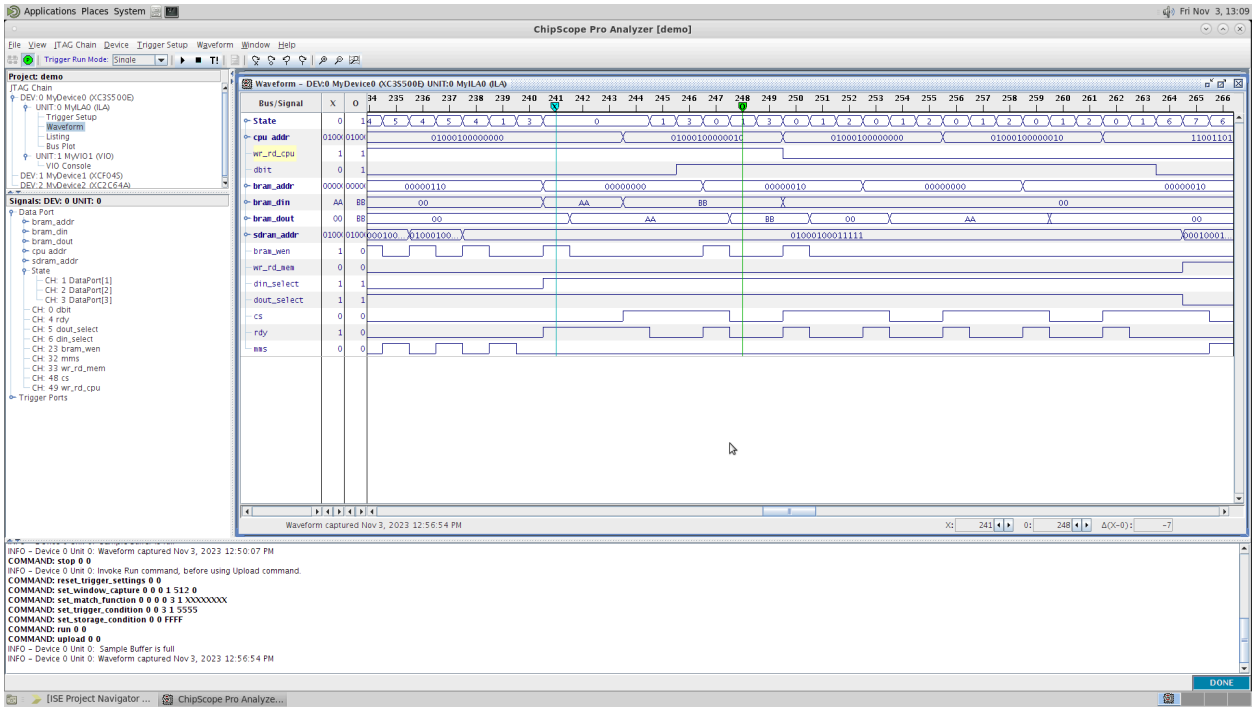
The main concept for this design project was to create a cache controller. This allowed us to conceptualize how a cache controller works and more specifically how the cpu interacts with the cache. It gives a good understanding of how hits and misses are determined and handled by the cache controller and how data is transferred from main memory into the cache. This project lets us analyze different cases where a cpu sends an instruction that could be a hit or miss, and also changes certain words in a block. We also saw how index, tag, block offset and V&D bits are crucial in a working cache system.

All Screenshots for every situation in reading the CPU req:

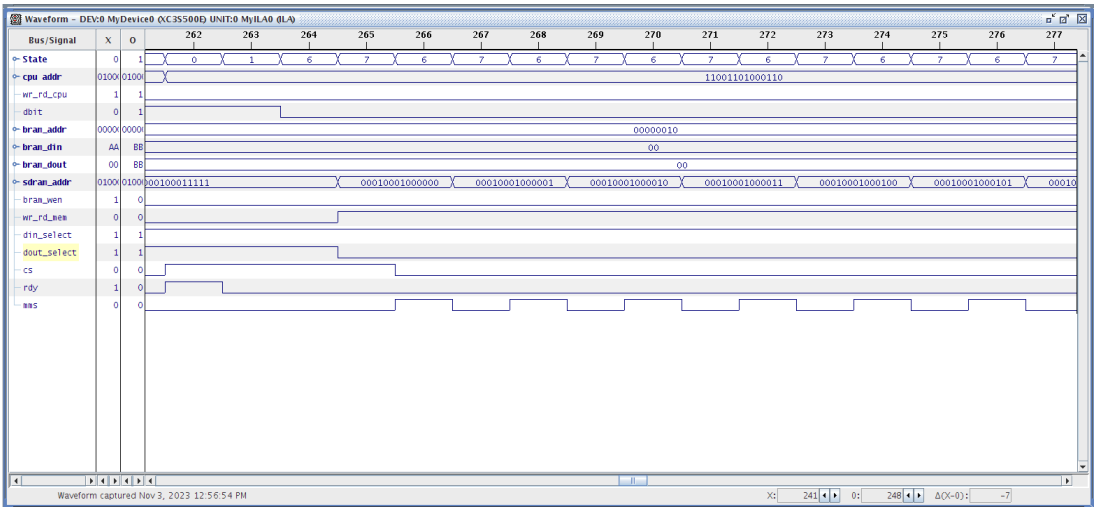
First write request - reading from main memory



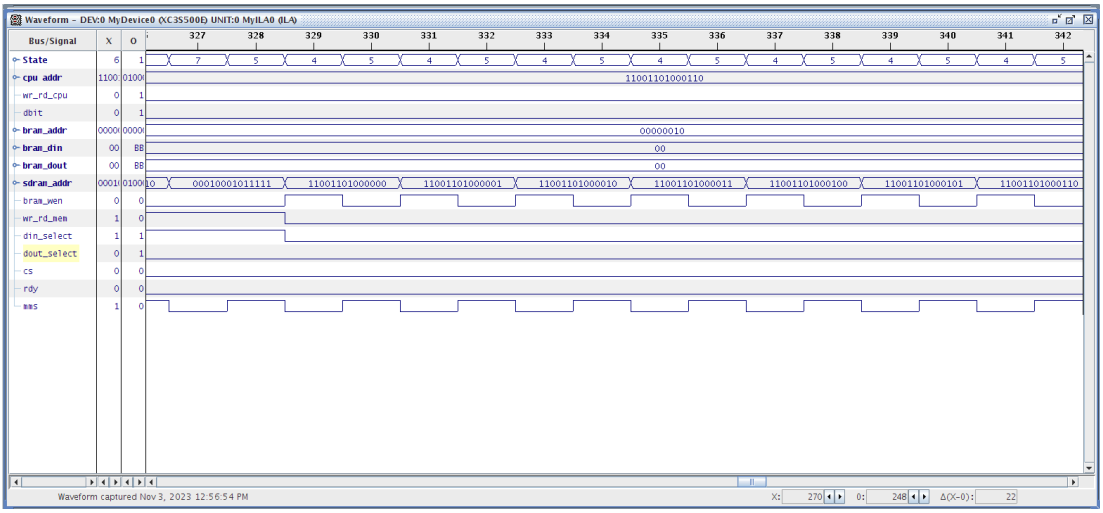
Showing first and second write request



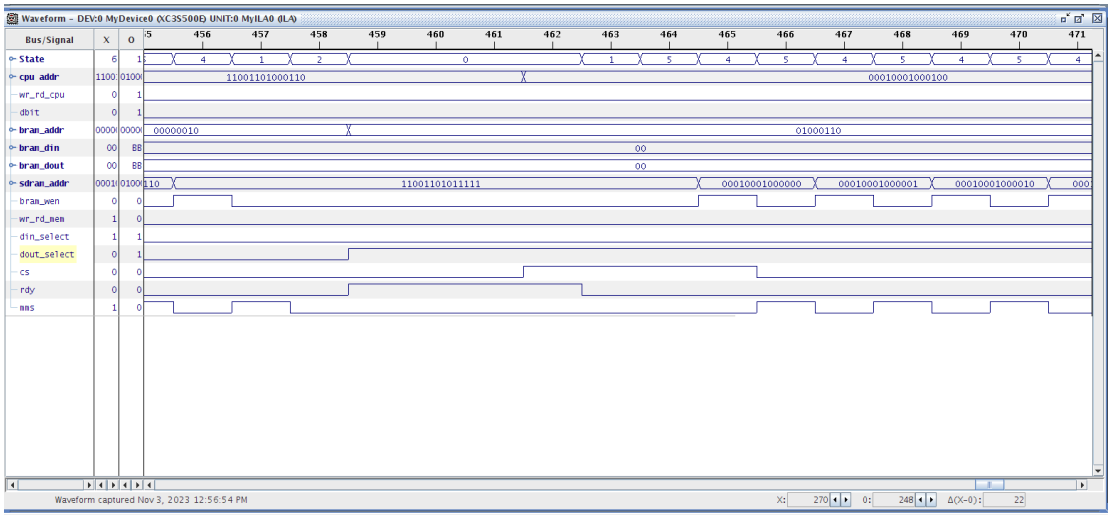
Read request with different address (miss)



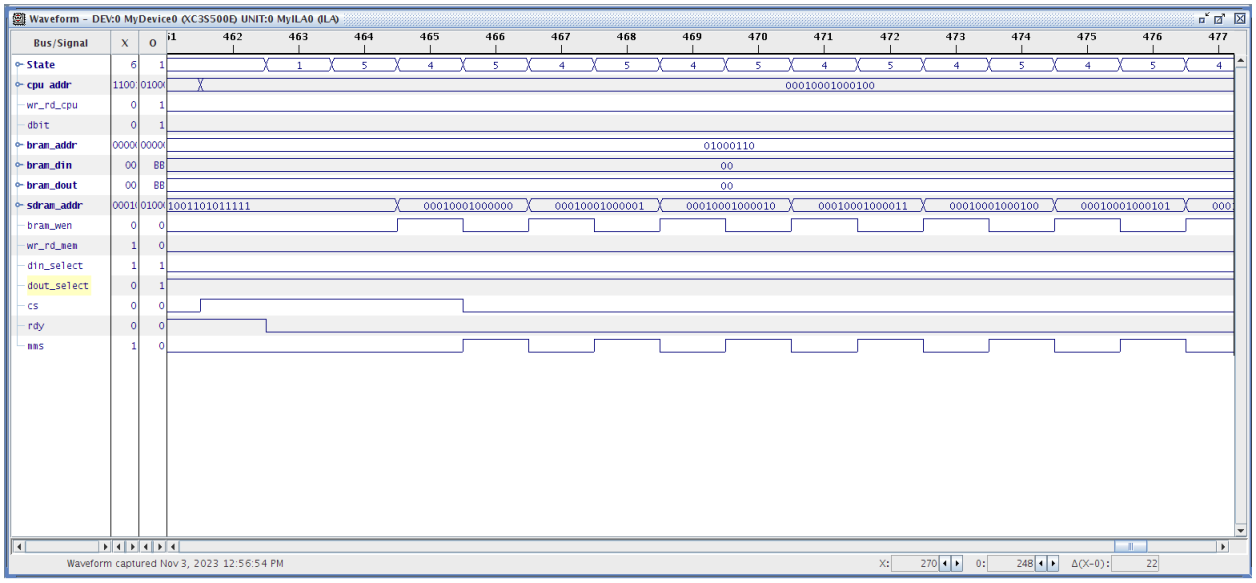
Continuing the miss, reading from main memory



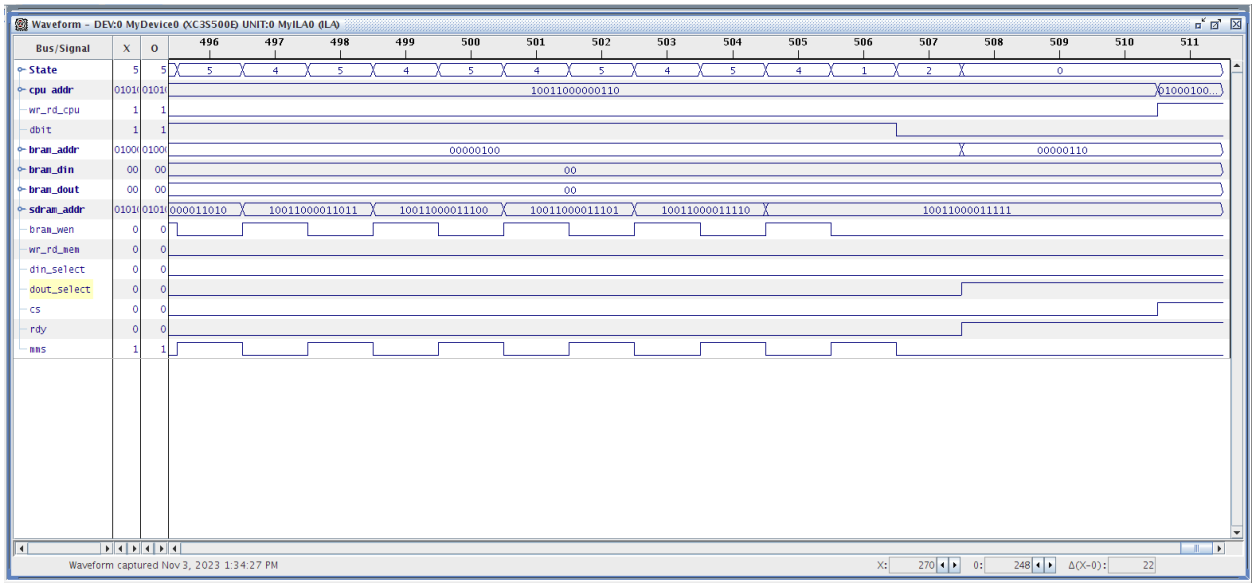
Request read



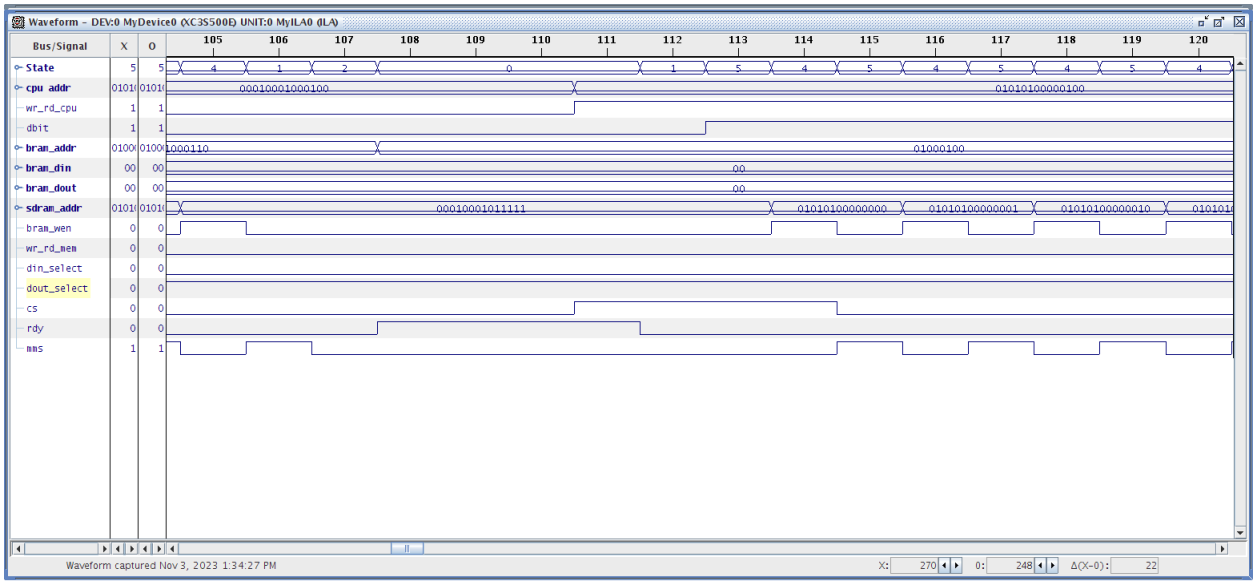
Next read request from main memory



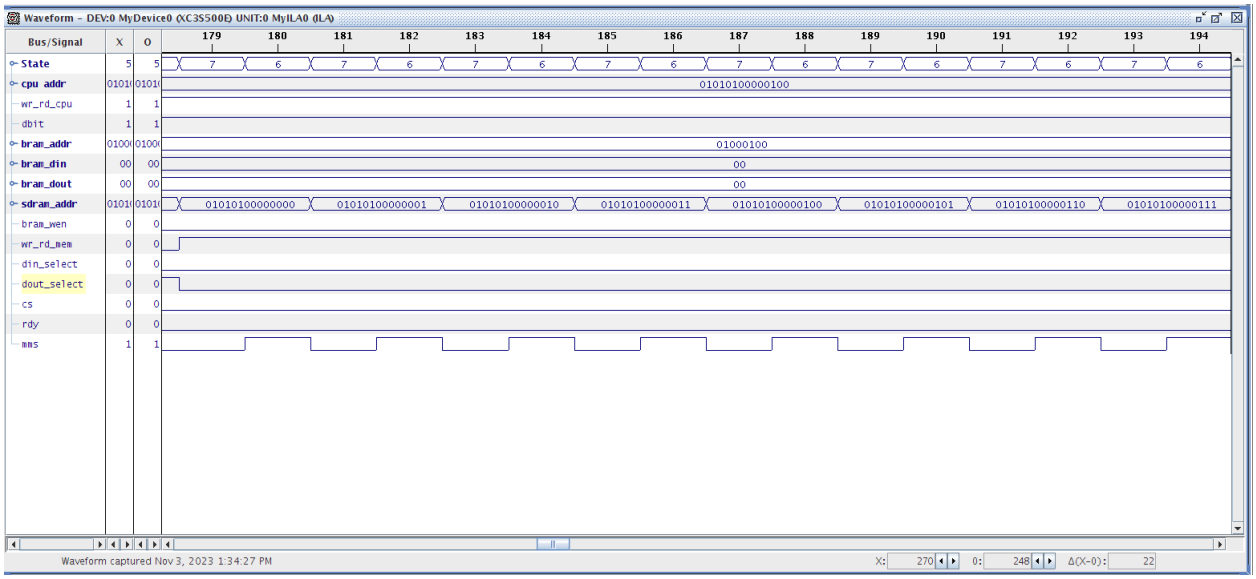
Services Read



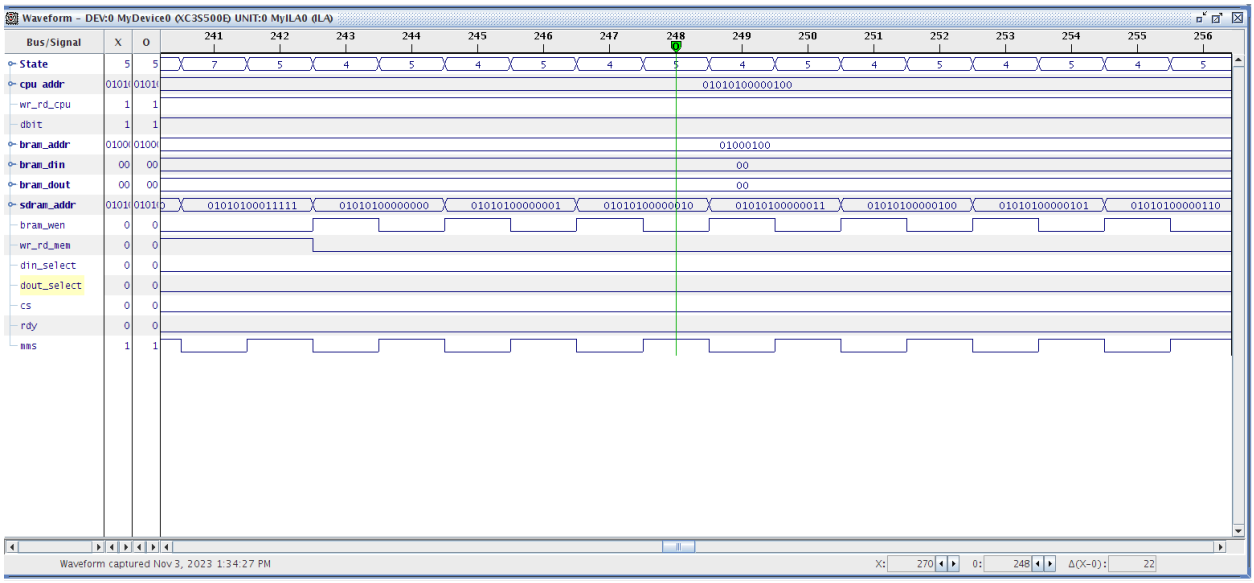
Last write (reads from main memory) (miss - D bit 1)

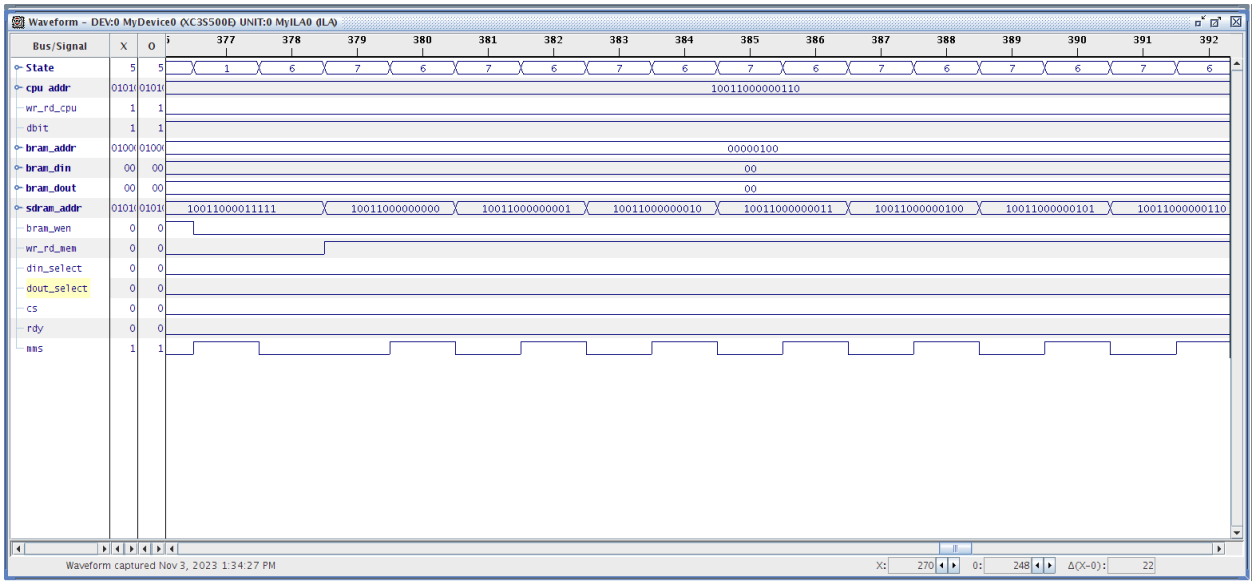


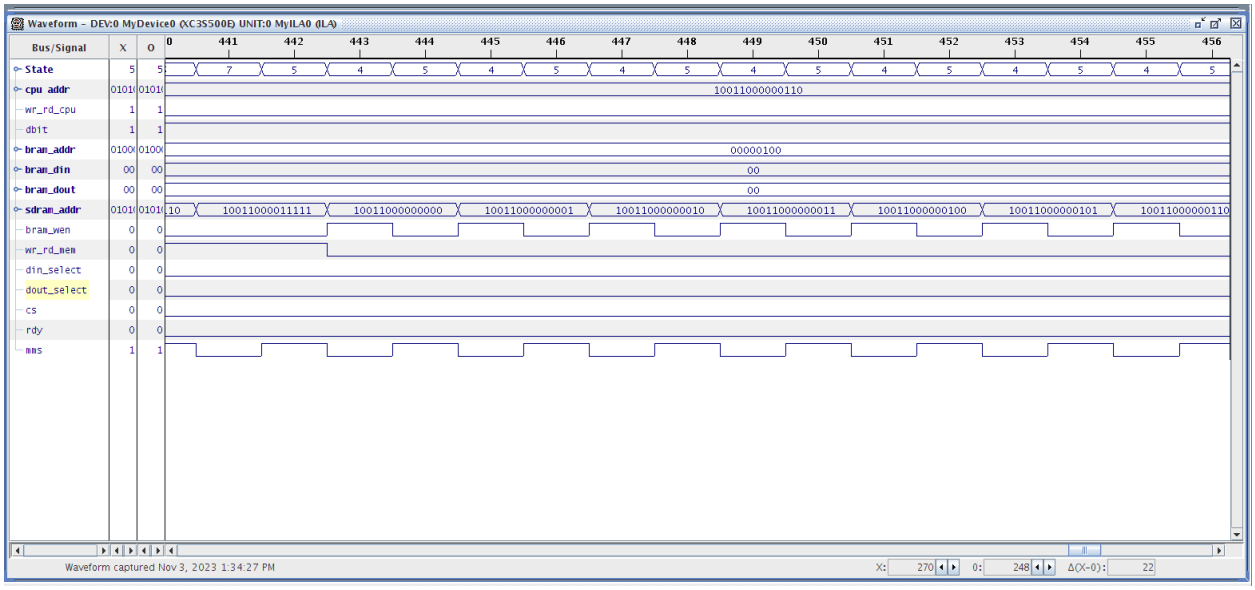
Writes to main

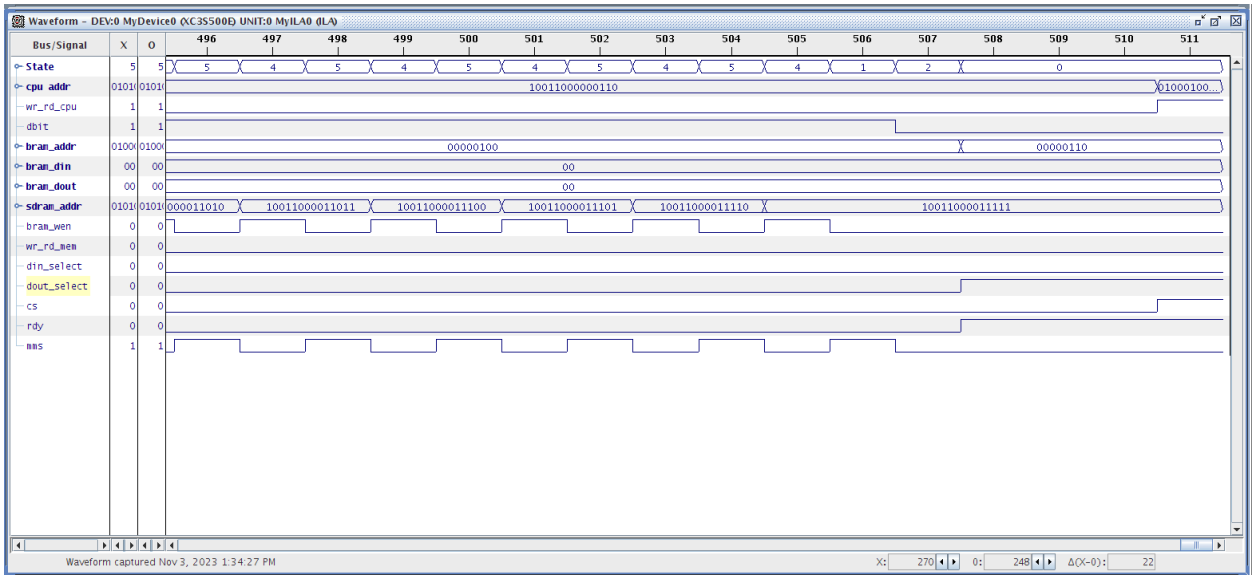


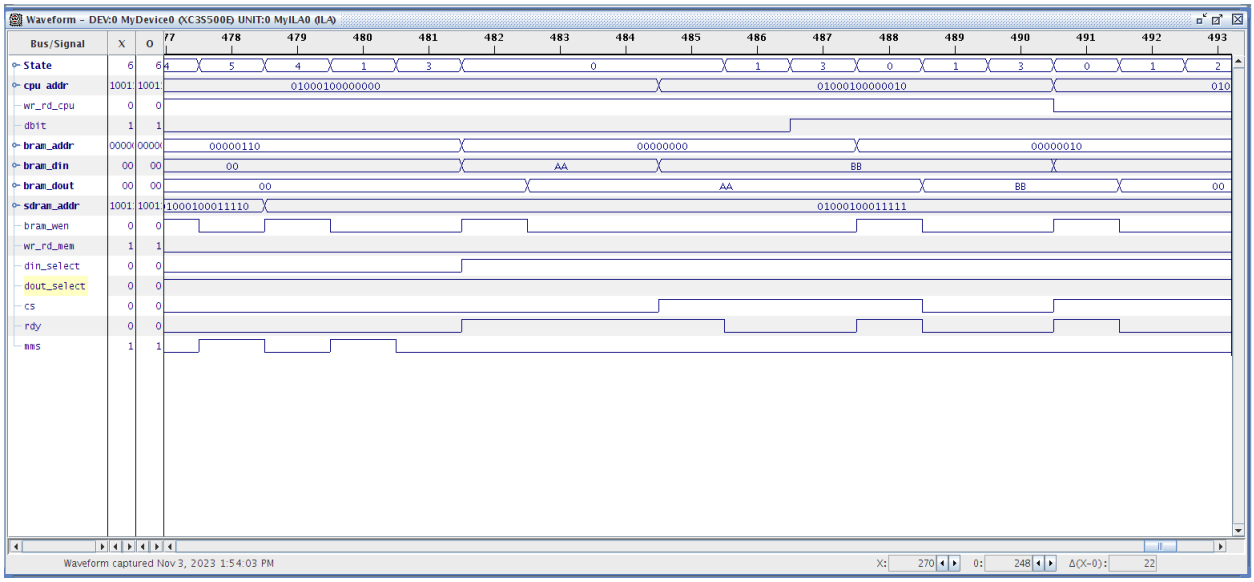
Reads from main memory






Waveform - DEV:0 MyDevice0 (XC3S500E) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
State
cpu_addr
wr_rd_cpu
dbit
bram_addr
bram_din
bram_dout
sdram_addr
bram_wen
wr_rd_mem
din_select
dout_select
cs
rdy
mms
10011000000110
00000100
00
00
10011000011111
10011000000000
10011000000001
10011000000010
10011000000011
10011000000100
10011000000101
10011000000110
Waveform captured Nov 3, 2023 1:34:27 PM
X: 270
0: 248
Δ(X-0): 22


Waveform - DEV:0 MyDevice0 (XC3S500E) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
State
cpu_addr
wr_rd_cpu
dbit
bram_addr
bram_din
bram_dout
sdram_addr
bram_wen
wr_rd_mem
din_select
dout_select
cs
rdy
mms
10011000000110
00000100
00000110
00
00
0000011010
10011000011011
10011000011100
10011000011101
10011000011110
10011000011111
Waveform captured Nov 3, 2023 1:34:27 PM
X: 270
0: 248
Δ(X-0): 22


Waveform – DEV:0 MyDevice0 (XC3S500E) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
State
cpu_addr
wr_rd_cpu
dbit
bram_addr
bram_din
bram_dout
sdram_addr
bram_wen
wr_rd_mem
din_select
dout_select
cs
rdy
mms
00000110
00000000
00000010
01000100000000
01000100000010
01000100011110
01000100011111
Waveform captured Nov 3, 2023 1:54:03 PM

Appendix-

Cache controller :

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cache_controller is
    Port ( clk : in  STD_LOGIC;
           addr_in : in  STD_LOGIC_VECTOR (13 downto 0);
           wrRDcpu : in  STD_LOGIC;
           cs : in  STD_LOGIC;
                        addr_out : out STD_LOGIC_VECTOR (13 downto 0);
           rdy : out  STD_LOGIC;
           wrRDmem : out  STD_LOGIC;
           mms : out  STD_LOGIC;
                        cacheAddr: out STD_LOGIC_VECTOR (7 downto 0);
                        wen: out STD_LOGIC_VECTOR (0 DOWNTO 0);
           din_select : out  STD_LOGIC;
           dout_select : out  STD_LOGIC;
                        state_a         :out STD_LOGIC_VECTOR (2 downto 0);
                        valid   : out std_logic;
                        dirty   : out std_logic
                        );
end cache_controller;

architecture Behavioral of cache_controller is
          signal tag : STD_LOGIC_VECTOR (5 downto 0);
          signal index: std_logic_vector(2 downto 0);
          signal offset: std_logic_vector(4 downto 0);
          signal state: std_logic_vector(2 downto 0) := "000";
          signal next_state: std_logic_vector(2 downto 0) := "000";
          --signal hit: std_logic;
          signal vbit: std_logic;
```

```
	signal dbit: std_logic;
	signal counter: std_logic_vector(4 downto 0) := (others => '0');
	signal try : std_logic;

	--signal sBram_addr: std_logic_vector(2 downto 0); --using index signal
	signal sBram_din, sBram_dout: std_logic_vector(7 downto 0);
	signal sBram_wen: std_logic_vector(0 downto 0) := "0";

	COMPONENT bram
	  PORT (
		clka : IN STD_LOGIC;
		wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
		addra : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
		dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
		douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
	  );
	END COMPONENT;
begin
	service_bram : bram
	  PORT MAP (
		clka => clk,
		wea => sBram_wen,
		addra => index,
		dina => sBram_din,
		douta => sBram_dout
	  );

	tag <= addr_in(13 downto 8);
	index <= addr_in(7 downto 5);
	offset <= addr_in(4 downto 0);
	--fsm states
	--000 - 0 ready
	--001 - 1 checking tag
	--010 - 2 read
	--011 - 3 write
	--100 - 4 delay read from main memory
	--101 - 5 read from main memory
	--110 - 6 write to main memory
	--111 - 7 delay write from main memory

	process(clk)
	begin
		if(clk'Event and clk = '1') then
			if(state = "000") then --checks for cpu request
```

```
                        if(cs = '1') then
                                rdy <= '0';
                                next_state <= "001";
                        else
                                rdy <= '1';
                                next_state <= "000";
                        end if;
                        wen <= "0";
                        mms <= '0';
                        sBram_wen <= "0";

--*******************************************************************************
                elsif(state = "001") then --if hit check read/write
                        --check tag
                        sBram_wen <= "0";
                        vbit <= sBram_dout(6);
                        dbit <= sBram_dout(7);

                        if(sBram_dout(5 downto 0) = tag and vbit = '1') then --if hit check if
read or write request
                                --hit
                                if(wrRDcpu = '0') then
                                        next_state <= "010"; --go to read state
                                else
                                        next_state <= "011"; --go to write state
                                end if;
                        else
                                                        --if miss check if dbit is 0 or 1
                                --miss
                                if(dbit = '0') then
                                        next_state <= "101"; --go to dbit=0 state
                                else
                                        next_state <= "110"; --go to dbit=1 state
                                end if;
                        end if;
                        rdy <= '0';
                        wen <= "0";
                        mms <= '0';

--*******************************************************************************
                elsif(state = "010") then --process read request
                        --ready
                                dout_select <= '1'; --direct cache data to cpu
                                wen <= "0";                     --read from cache
```

```
                                        mms <= '0';

                                        cacheAddr(7 downto 5) <= index;
                                        cacheAddr(4 downto 0) <= offset;

                                        sBram_wen <= "0";
                                        next_state <= "000";
                                        rdy <= '1';

--*****************************************************************************
                        elsif(state = "011") then --process write request
                                --write
                                        wen <= "1"; --write to cache
                                        mms <= '0';
                                        din_select <= '1'; --direct cpu data to cache
                                        cacheAddr(7 downto 5) <= index;
                                        cacheAddr(4 downto 0) <= offset;

                                        --update tag info
                                        sBram_wen <= "1";
                                        sBram_din(7) <= '1'; --sets dbit to 1
                                        sBram_din(6) <= '1';   --sets vbit to 1
                                        sBram_din(5 downto 0) <= tag;

                                        next_state <= "000";
                                        rdy <= '1';

--*****************************************************************************
                        elsif(state = "100") then --waited 1 clock cycle before asserting main mem
strobe
                                rdy <= '0';
                                mms <= '1';
                                wen <= "0";
                                wrRDmem <= '0';
                                din_select <= '0'; --direct data from main memory to cache

                                if(counter = "11111") then
                                        --update tag info
                                        sBram_wen <= "1";
                                        sBram_din(7) <= '0'; --sets dbit to 1
                                        sBram_din(6) <= '1';   --sets vbit to 1
                                        sBram_din(5 downto 0) <= tag; --change tag to the one
given by CPU
                                        next_state <= "001"; --service initial request
```

```
                        counter <= "00000";
                    else
                        counter <= counter + '1';
                        next_state <= "101";
                    end if;

--*******************************************************************************
                elsif(state = "101") then --read from main mem
                    rdy <= '0';
                    mms <= '0';
                    addr_out(13 downto 8) <= tag;
                    addr_out(7 downto 5) <= index;
                    addr_out(4 downto 0) <= counter;
                    wen <= "1"; --write to cache
                    wrRDmem <= '0'; --read from main memory
                    din_select <= '0'; --direct main memory data to cache

                    sBram_wen <= "0";
                    next_state <= "100";

--*******************************************************************************
                elsif(state = "110") then --write to main mem
                    rdy <= '0';
                    sBram_wen <= "0"; --read from service bram to get tag to be
replaced
                    mms <= '0';
                    addr_out(13 downto 8) <= sBram_dout(5 downto 0);
                    addr_out(7 downto 5) <= index;
                    addr_out(4 downto 0) <= counter;
                    wen <= "0"; --read from cache
                    wrRDmem <= '1'; --write to main memory
                    dout_select <= '0'; --direct cache data to main memory

                    next_state <= "111";

                elsif(state = "111") then --waited 1 clock cycle before asserting main mem
strobe
                    rdy <= '0';
                    mms <= '1';
                    wen <= "0";
                    wrRDmem <= '1';
                    dout_select <= '0';
                    sBram_wen <= "0";
```

```
                                if(counter = "11111") then
                                        next_state <= "101"; --read block requested by cpu from
main memory
                                        counter <= "00000";
                                else
                                        counter <= counter + '1';
                                        next_state <= "110";
                                end if;

                        else
                                next_state <= state;
                        end if;
                end if;
                state_a <= state;
                state <= next_state;
                valid <= vbit;
                dirty <= dbit;

        end process;
end Behavioral;
```

SDRAM controller:

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


entity sdram_controller is
   Port ( clk     : std_logic;
                        addr : in  STD_LOGIC_VECTOR (13 downto 0);
          wrRD : in  STD_LOGIC;
          mms : in  STD_LOGIC;
          din : in  STD_LOGIC_VECTOR (7 downto 0);
          dout : out  STD_LOGIC_VECTOR (7 downto 0));
end sdram_controller;

architecture Behavioral of sdram_controller is

        type RAM is array (16384 downto 0) of std_logic_vector (7 downto 0);

        signal SDRAM: RAM := (others => (others => '0'));
```

```
begin
        process(clk)
        begin
                if(clk'Event and clk='1') then
                        if(wrRD = '1' and mms = '1') then
                                SDRAM(conv_integer(unsigned(addr))) <= din;
                        elsif(wrRD = '0' and mms = '1') then
                                dout <= SDRAM(conv_integer(unsigned(addr)));
                        end if;
                end if;
        end process;
end Behavioral;
```

MUX:

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


entity mux is
   Port ( sdram_dout : in  STD_LOGIC_VECTOR (7 downto 0);
          cpu_dout : in  STD_LOGIC_VECTOR (7 downto 0);
          din_select : in  STD_LOGIC;
          bram_din : out  STD_LOGIC_VECTOR (7 downto 0));
end mux;

architecture Behavioral of mux is

begin
         WITH din_select SELECT
                bram_din <=   sdram_dout WHEN '0',
                                          cpu_dout WHEN OTHERS;

end Behavioral;
```

DEMUX:

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


entity demux is
   Port ( bram_dout : in  STD_LOGIC_VECTOR (7 downto 0);
          dout_select : in  STD_LOGIC;
          sdram_din : out  STD_LOGIC_VECTOR (7 downto 0);
          cpu_din : out  STD_LOGIC_VECTOR (7 downto 0));
```

```
end demux;

architecture Behavioral of demux is

begin
	WITH dout_select SELECT
		sdram_din <=  bram_dout WHEN '0',
                                                "00000000" WHEN OTHERS;
	WITH dout_select SELECT
		cpu_din <=	bram_dout WHEN '1',
                                                "00000000" WHEN OTHERS;

end Behavioral;
```

CPU Gen:

```
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);

	-- 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;

-- Next state generation.
process(st1, trig_r)
begin
	if(st1 = "000")then
		if(trig_r = '1')then
			st1N <= "001";
		else
			st1N <= "000";
		end if;
	elsif(st1 = "001")then
		st1N <= "010";
	elsif(st1 = "010")then
		st1N <= "011";
	elsif(st1 = "011")then
		st1N <= "100";
	elsif(st1 = "100")then
		st1N <= "101";
	elsif(st1 = "101")then
		st1N <= "000";
	else
		st1N <= "000";
	end if;
end process;

-- Output generation.
process(st1)
begin
```

```
		if(st1 = "000")then
			updPat <= '0';
			cs <= '0';
		elsif(st1 = "001")then
			updPat <= '1';
			cs <= '0';
		elsif(st1 = "010")then
			updPat <= '0';
			cs <= '1';
		elsif(st1 = "011")then
			updPat <= '0';
			cs <= '1';
		elsif(st1 = "100")then
			updPat <= '0';
			cs <= '1';
		elsif(st1 = "101")then
			updPat <= '0';
			cs <= '1';
		else
		end if;
end process;

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

-- 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 <= "000100010000000101010101"; --write 0xAA into
address:010001 000 00000                                        miss dbit=0
		elsif(PatCtrl = "001")then
			patOut <= "000100010000010101110111"; --write 0xBB into
adress:010001 000 00010                                         hit
		elsif(PatCtrl = "010")then
			patOut <= "000100010000000000000000"; --read from address: 010001
000 00000 should be 0xAA		hit
		elsif(PatCtrl = "011")then
			patOut <= "000100010000010000000000"; --read from address: 010001
000 00010 should be 0xBB		hit
		elsif(PatCtrl = "100")then
			patOut <= "001100110100011000000000"; --read from address: 110011
010 00110 should be 0x00		miss dbit=0
		elsif(PatCtrl = "101")then
			patOut <= "010001000100010000000000"; --read from address: 000100
010 00100 should be 0x00		miss dbit=0
		elsif(PatCtrl = "110")then
			patOut <= "010101010000010011001001"; --write 0xCC to address:
010101 000 00100                                                miss dbit=1
		else
			patOut <= "011001100000011000000000"; --read from address: 100110
000 00110                                                       miss dbit=1
		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;
```

Demo:

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

entity demo is
    Port (
        clk  : in  STD_LOGIC;
        rst  : in  STD_LOGIC;
        data : out STD_LOGIC_VECTOR (63 downto 0)
    );
end demo;

architecture Behavioral of demo is
    -- chipscope cores
    component icon
        PORT (
            CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
            CONTROL1 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0)
```

```
    );
end component;

component mux
    Port (
        sdram_dout : in  std_logic_vector(7 downto 0);
                                    cpu_dout   : in  std_logic_vector(7 downto 0);
        din_select : in  std_logic;
        bram_din   : out std_logic_vector(7 downto 0)
    );
end component;

component demux
    Port (
        bram_dout   : in  std_logic_vector(7 downto 0);
        dout_select : in  std_logic;
        sdram_din   : out std_logic_vector(7 downto 0);
        cpu_din     : out std_logic_vector(7 downto 0)
    );
end component;

component ila
    PORT (
        CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
        CLK     : IN STD_LOGIC;
        DATA    : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
        TRIG0   : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
end component;

component vio
    PORT (
        CONTROL    : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
        ASYNC_OUT  : OUT STD_LOGIC_VECTOR(17 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;

  -- cache controller
  component cache_controller
    Port (
      clk        : in  STD_LOGIC;
      addr_in    : in STD_LOGIC_VECTOR (13 downto 0);
      wrRDcpu    : in  STD_LOGIC;
      cs         : in  STD_LOGIC;
                          addr_out    : out STD_LOGIC_VECTOR (13 downto 0);
      rdy        : out STD_LOGIC;
      wrRDmem    : out STD_LOGIC;
      mms        : out STD_LOGIC;
      cacheAddr  : out STD_LOGIC_VECTOR (7 downto 0);
      wen        : out STD_LOGIC_VECTOR (0 downto 0);
      din_select  : out STD_LOGIC;
      dout_select : out STD_LOGIC;
                          state_a                  : out std_logic_vector (2 downto 0);
                          valid    : out std_logic;
                          dirty    : out std_logic
    );
  end component;

  -- cpu
  component CPU_gen
    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 component;

  -- sdram controller with sdram
  component sdram_controller
    Port (
      clk   : std_logic;
      addr  : in  STD_LOGIC_VECTOR (13 downto 0);
      wrRD  : in  STD_LOGIC;
```

```
        mms   : in  STD_LOGIC;
        din   : in  STD_LOGIC_VECTOR (7 downto 0);
        dout  : out STD_LOGIC_VECTOR (7 downto 0)
      );
  end component;

  -- signals
        signal address     : std_logic_vector(13 downto 0) := (others => '0'); --connects cpu
address and CC addr in
        signal cpu_address : std_logic_vector(15 downto 0) := (others => '0');
        signal sdram_address : std_logic_vector(13 downto 0) := (others => '0'); --connects CC
addr out to sdram CC

        signal state                : std_logic_vector (2 downto 0);
        signal bram_addr   : std_logic_vector(7 downto 0) := (others => '0'); --connect cacheAddr
and sram addra
        signal bram_din    : std_logic_vector(7 downto 0) := (others => '0');
        signal bram_dout   : std_logic_vector(7 downto 0) := (others => '0');
        signal bram_wen    : std_logic_vector(0 downto 0) := (others => '0');
        signal cpu_dout    : std_logic_vector(7 downto 0) := (others => '0');
        signal sdram_dout  : std_logic_vector(7 downto 0) := (others => '0');
        signal cpu_din     : std_logic_vector(7 downto 0) := (others => '0');
        signal sdram_din   : std_logic_vector(7 downto 0) := (others => '0');
        signal din_select  : std_logic := '0';
        signal dout_select : std_logic := '0';
        signal wr_rd_cpu   : std_logic := '0';
        signal rdy         : std_logic := '0';
        signal wr_rd_mem   : std_logic := '0';
        signal mms         : std_logic := '0';
        signal cs          : std_logic := '0';
        signal ila_data    : std_logic_vector(63 downto 0) := (others => '0');
        signal trig0       : std_logic_vector(7 DOWNTO 0) := (others => '0');
        signal vio_out     : std_logic_vector(17 downto 0) := (others => '0');
        signal control0    : std_logic_vector(35 DOWNTO 0) := (others => '0');
        signal control1    : std_logic_vector(35 DOWNTO 0) := (others => '0');
        signal vbit                         : std_logic := '0';
        signal dbit        : std_logic := '0';

begin

  --control0 <= (others => '0');
  --control1 <= (others => '0');
        address <= cpu_address(13 downto 0);
```

```
  -- icon
  iconA : icon
     port map (
        CONTROL0 => control0,
        CONTROL1 => control1
     );

  -- ila
  ilaA : ila
     port map (
        CONTROL => control0,
        CLK     => clk,
        DATA    => ila_data,
        TRIG0   => trig0
     );

  -- vio
  system_vio : vio
     port map (
        CONTROL   => control1,
        ASYNC_OUT => vio_out
     );

  -- bram
  cache : sram
     PORT MAP (
        clka  => clk,
        wea   => bram_wen, -- Matched signal types
        addra => bram_addr,
        dina  => bram_din,
        douta => bram_dout
     );

  -- cpu
  CPU : CPU_gen
     port map (
        clk     => clk,
        rst     => rst,
        trig    => rdy,
        Address => cpu_address,
        wr_rd   => wr_rd_cpu,
        cs      => cs,
        DOut    => cpu_dout
     );
```

```
-- MUX and DEMUX
MUX_inst : MUX
   port map (
      cpu_dout   => cpu_dout,
      sdram_dout => sdram_dout,
      din_select => din_select,
      bram_din   => bram_din
   );

DEMUX_inst : DEMUX
   port map (
      bram_dout   => bram_dout,
      dout_select => dout_select,
      sdram_din   => sdram_din,
      cpu_din     => cpu_din
   );

-- cache controller
cache_ctrl : cache_controller
   port map (
      clk         => clk,
      addr_in     => address,
      wrRDcpu     => wr_rd_cpu,
      cs          => cs,
                     addr_out         => sdram_address,
      rdy         => rdy,
      wrRDmem     => wr_rd_mem,
      mms         => mms,
      cacheAddr   => bram_addr,
      wen         => bram_wen,
      din_select  => din_select,
      dout_select => dout_select,
                     state_a          => state,
                     valid       => vbit,
                     dirty       => dbit
   );

-- connecting to the sdram controller
sdram_ctrl : sdram_controller
   port map (
      clk  => clk,
      addr => address,
      wrRD => wr_rd_mem,
```

```
        mms  => mms,
        din  => sdram_din,
        dout => sdram_dout
    );

  -- Output data from the ILA
        ila_data(63 downto 50) <= address; --address from cpu to cache controller
        ila_data(49) <= wr_rd_cpu;
        ila_data(48) <= cs;
        ila_data(47 downto 34) <= sdram_address; --address cache controller sends to main
mem
        ila_data(33) <= wr_rd_mem;
        ila_data(32) <= mms;
        ila_data(31 downto 24) <= bram_addr; --address cache controller sends to cache
        ila_data(23) <= bram_wen(0);
        ila_data(22 downto 15) <= bram_din; --data entering cache
        ila_data(14 downto 7) <= bram_dout; --data leaving cache
        ila_data(6) <= din_select;
        ila_data(5) <= dout_select;
        ila_data(4) <= rdy;
        ila_data(3 downto 1) <= state;
        ila_data(0) <= dbit;

  data <= ila_data;

end Behavioral;
```