Table of Contents

1. Abstract

The modern computer has gone through multiple iterations with improvements occurring every step of the way. The effective use of many different memory types has effectively increased processor efficiency, while keeping costs low. As data gets transported to and from the processor, the data travels through different memory components. The cache utilizes an SRAM, and expensive but fast memory module, only used in small quantities due to its cost. The main memory, or hard disk, utilizes the slower, but cheaper DRAM memory, and can be used in larger quantities due to its cost. The balance between cost and efficiency is where true engineering comes into play. One of the breakthroughs is the concept of spatial and temporal locality. Spatial locality is the idea that a piece of data which is used in a computational process will often be physically close to other pieces of data which will be needed soon. Temporal locality is the idea that a piece of data which is used in a computational process will be reused soon. These two attributes allowed computer engineers to design a memory hierarchy that improves computation speed while keeping costs low.

The hierarchy consists of a small amount of cache memory close to the CPU, which allows fast data transfers between the CPU and the cache. Data in the cache SRAM is stored in a system of blocks, with a certain number of words per block. The data in the SRAM is controlled by the cache controller, which uses identifiers to determine the state of the memory, and using that information, controls the movement of data throughout the system. The main memory, connected via busses to the cache, contains the slower but higher in quantity DRAM memory. Loading and writing in blocks between the cache and main memory allows for data to be quickly communicated between the CPU and cache, while also allowing for large amounts of data to be stored in memory. The high-level memory hierarchy is shown in *Figure 1.1*.

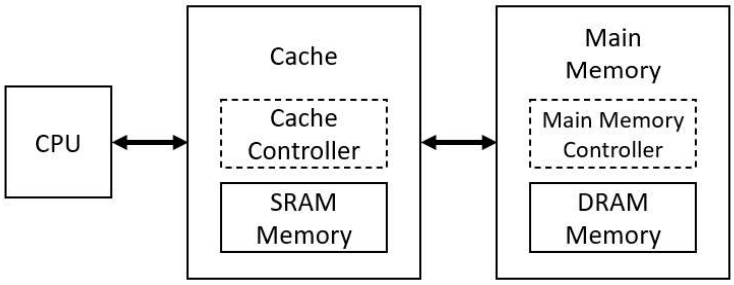

Figure 1.1: Memory Hierarchy

2. Introduction

The project described in this report explains and showcases a simplified cache controller, which controls the overall data movement between the CPU, cache SRAM, and main memory DRAM. The CPU issues instructions to the cache, and the cache controller will use the instruction to decide how to move the data of the system around in the most efficient way. The cache contains a SRAM which is organized in 8 blocks of 32 words each, for a total of 256 bytes of storage and is shown in *Figure 2.1*.

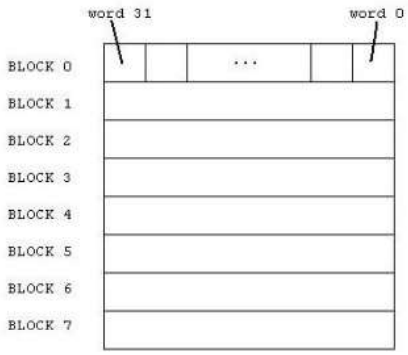

Figure 2.1: Memory Hierarchy

The CPU only interacts with the cache, so the cache controller will also need to decide on the instructions to send to the main memory to complete the CPU request. The complete memory hierarchy specification is a more detailed expansion of *Figure 1.1* and is shown in *Figure 2.2*.

The CPU sends instructions to the cache controller, and the cache controller will need to execute the instruction while also maintaining data concurrency between the SRAM and the DRAM based on the valid-bit, and dirty-bit identifiers. If changes have been made to memory in the SRAM, the changes need to be reflected in the associated memory block in the DRAM to maintain concurrency.

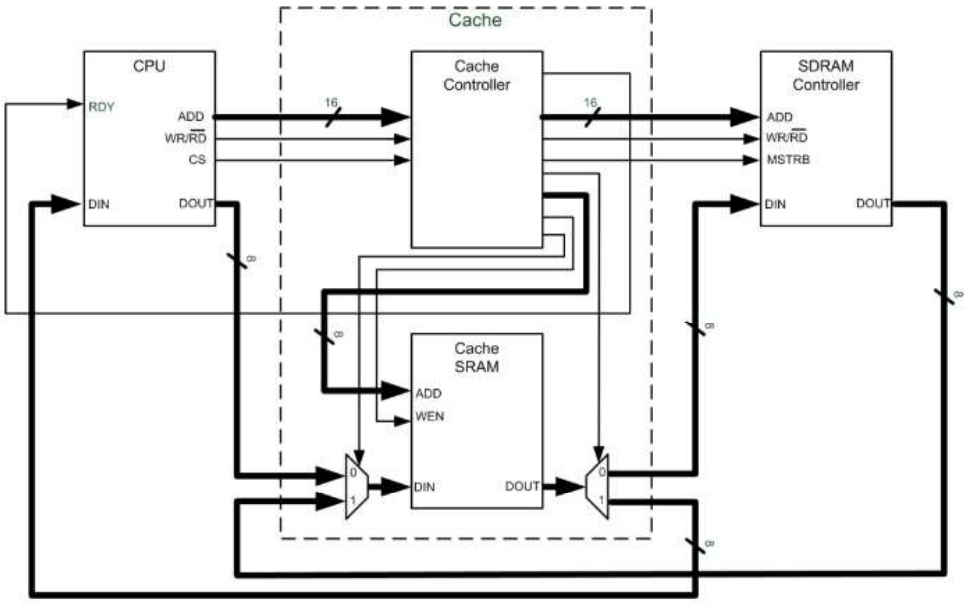

Figure 2.2: Expanded Memory Hierarchy

3. Specifications

This section outlines the functional and behavioural specifications of the cache controller to be implemented.

3.1 CPU Address Word Organization

The CPU sends out a 16-bit long address word to the cache controller. The cache controller then breaks down the address into 3 fields, the TAG, INDEX, and OFFSET. The address breakdown is shown in *Figure 3.1.1*.

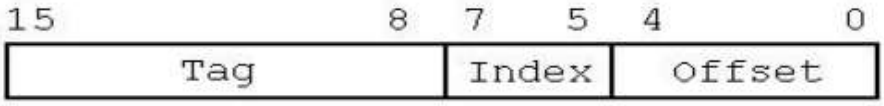

Figure 3.1.1: Address Word Register Fields

3.2 Cache Controller Behaviour

The cache controller controls the movement of data across the whole system. The CPU sends an instruction to the cache controller, and the controller will execute one of 4 main cases depending on the instruction and current states of the memory block. The CPU instruction will include either a WRITE or READ command, accompanied by a 16-bit address block. The data exchange must happen between the CPU and the cache, so if the data does not exist in the cache, the cache controller will load in the data from the main memory.

When the cache controller receives an instruction, it will first determine if the cache contains the specified address block, this is called a HIT. The incoming address block gets broken down into its respective fields as explains in section 3.1. The INDEX portion is used to locate a specific address inside the TAG register. The 8-bit TAG found in the register gets compared to the TAG portion of the address received from the CPU. If the two TAGs match, then the valid bit is checked, if it is a 1, then the entire address is a HIT, and the corresponding address location is in the cache and the data transfer can commence. If the TAGs do not match, or the valid bit is 0, then it is a MISS. The INDEX field specifies which of the 8 blocks the requested data location exists in. The OFFSET specifies which of the 32 words within the block is needed.

In the event of a MISS, the dirty bit is checked. The details of the results can be found in the corresponding cases 3 and 4 listed below:

1. **WRITE a word to the cache from the CPU [HIT]**

If the steps listed above were followed and the result is a WRITE command with a HIT, then the cache controller takes the data that comes from the CPU and writes it into the specified address location. The dirty bit of the block specified by the INDEX of the address gets set to 1 which means that the data of the block in the cache is different from the same block in the main memory.

2. **READ a word from the cache to the CPU [HIT]**

If the steps listed above were followed and the result is a READ command with a HIT, the cache controller will output the data located in the specified address provided by the CPU.

3. **WRITE/READ to/from the cache from/to the CPU [MISS] and dirty bit is 0**

If the steps listed above were followed and the result is either a WRITE or READ command with a MISS, this means that the specified data is not located in the cache and must be loaded in from the main memory. The dirty bit is checked to see if the current block in the cache that needs to be replaced can be safety overwritten by the new block, or if there is a mismatch. In this case, the dirty bit is 0, meaning the specified block can just be loaded in from the main memory. The cache controller sends the 16-bit address that was received from the CPU in the format [TAG+INDEX+OFFSET] to the main memory however, the OFFSET portion starts with a value of "00000", which results in [TAG+INDEX+00000]. This specifies the first word in the necessary block in the main memory that needs to be loaded into the cache. Once that word is loaded, the OFFSET increments by one, and in the next clock cycle the next word 00001 is loaded in, and so on and so forth until the last word FFFFF is loaded in. The tag register is updated at the INDEX location with the TAG received from the CPU, the valid bit is set to 1, and the dirty bit is set to 0.

4. **WRITE/READ to/from the cache from/to the CPU [MISS] and dirty bit is 1**

If the result is a MISS and the dirty bit of the block in the cache is a 1, that means there is a mismatch. A mismatch means that the block currently in the cache contains data that is not in the block in the main memory, and the block must be loaded back into the main memory before the new necessary block can be loaded in from the main memory into the cache. To do so, the block with the word located at OFFSET 00000 is written into the main memory, once done, the OFFSET increments by 1 to 00001, and that word gets written into the main memory. The OFFSET continues incrementing until the last word located at OFFSET FFFFF is written in the main memory. Once the affected block is written back into the main memory from the cache, the new block can be loaded in from the cache to the main memory, and the flags and registers are updated using the steps outlined in case 3.

4. Device Designs

This section explains the different components designed and introduced into the project, their symbols, diagrams, and processes. The CPU has been excluded from this section because it is treated as a working black box and has not been designed in the scope of this project.

4.1 Cache

Referring to Figure 2.2, the cache is comprised of 4 different components, the SRAM, which is the memory component, the data in mux, the data out mux, and the cache controller. All the components were created separately and connected in the cache. The VHDL code for the cache component can be found in the Appendix, Section 8.1.

Data in Mux

The data in multiplexer symbol is shown in *Figure 4.1.1.*

Data can be loaded into the SRAM from two sources, the CPU or the main memory. To switch between the two, a dual-input single-output mux is implemented, with the cache controller sending a selector signal to the mux to set the source going into the SRAM

memory block. The port connections between the mux, cache controller, CPU, main memory, and SRAM can be seen in *Figure 2.2.*

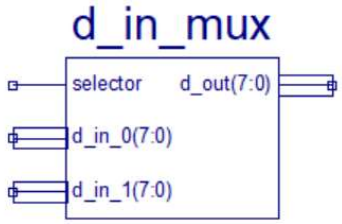

Figure 4.1.1: Data in Multiplexer Symbol

Data out Mux

The data out multiplexer symbol is shown in *Figure 4.1.2*.

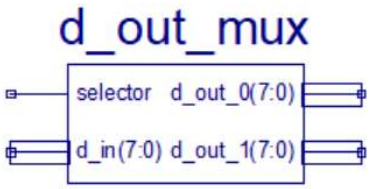

Figure 4.1.2: Data out Multiplexer Symbol

Data from the SRAM can be loaded into two locations, the main memory or the CPU. To switch between the two, a single-input, dual-output mux is implemented, with the cache controller sending a selector signal to the mux to set the output that the SRAM data will be sent to. The port connections between the mux, cache controller, CPU, main memory, and SRAM can be seen in *Figure 2.2.*

Cache Controller

Symbol

The cache controller symbol is shown in *Figure 4.1.3.*

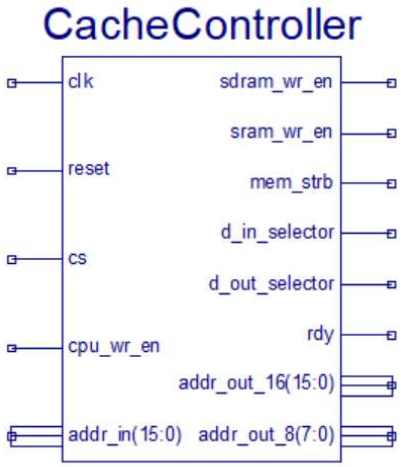

Figure 4.1.3: Cache Controller Symbol

The cache controller is clocked with the rest of the system, receives the strobe signal (*cs*) along with the write enable signal (*cpu_wr_en*), and the 16-bit address (*addr_in*) from the CPU. The cache controller outputs a write enable signal (*sdram_wr_en*) as well as an 8-bit address (*addr_out_8*) to the SRAM block within the cache. The controller also outputs a ready signal (*rdy*) to the CPU to signify when it is ready for the next instruction. The controller also outputs selector signals (*d_in_selector* and *d_out_selector*) to control the two multiplexers that control which of either the CPU or main memory gets to input or output data from/to the cache. The controller also outputs a write enable (*sdram_wr_en*), memory strobe (*mem_strb*), and 16-bit address signal (*addr_out*_16) to the main memory.

Finite State Machine

The cache controller has a state machine hierarchy implemented internally in VHDL. The state machine flow is provided and displayed in *Figure 4.1.4.*

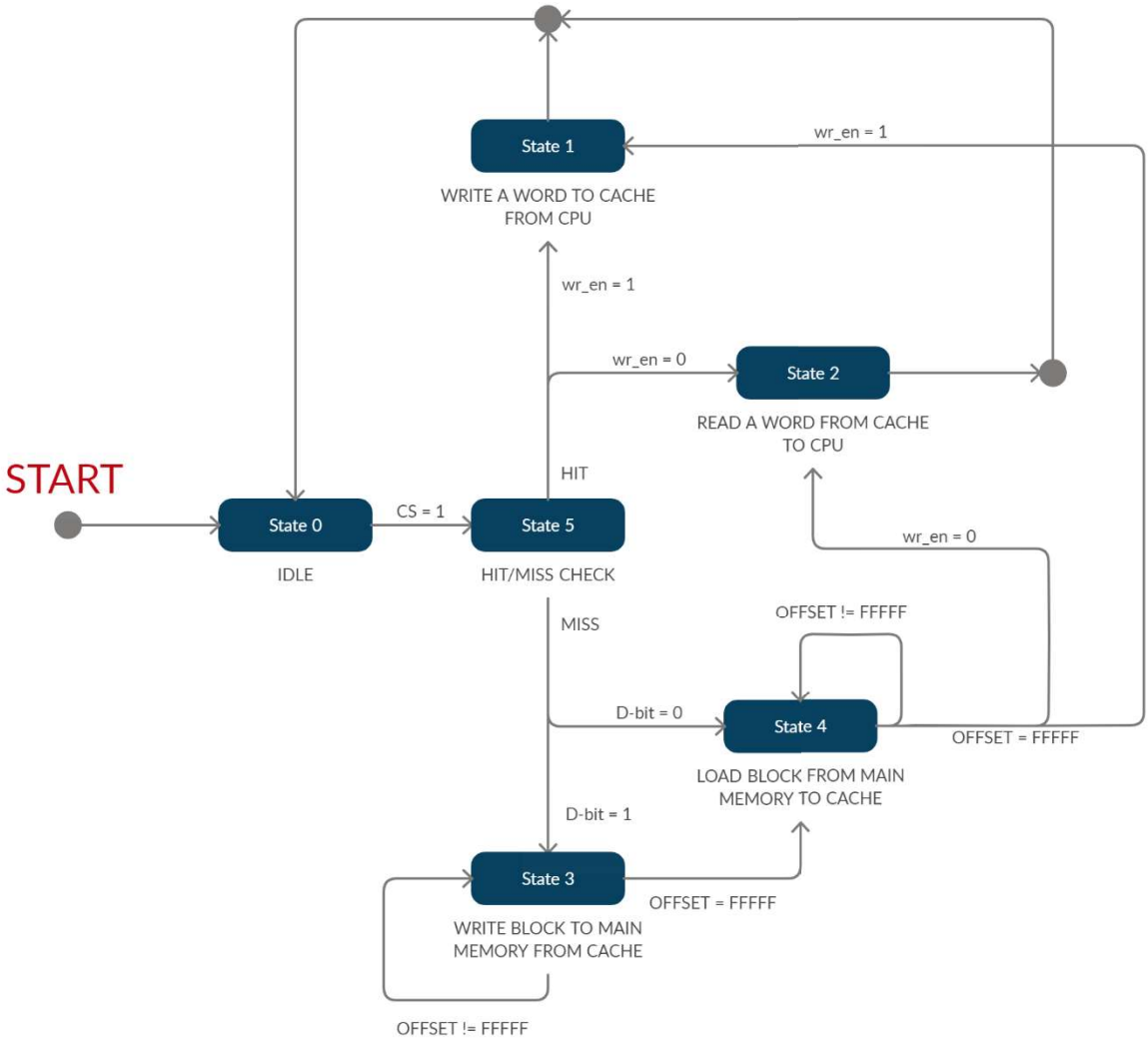

Figure 4.1.4: Cache Controller's State Machine Flow

Explanation of the states:

State 0 - IDLE:

The cache controller resets to this idle state. The cache controller waits until the CPU sends a CS trigger which will move the state to State 5.

Output signals:

rdy = 1

State 1 – WRITE a word to cache from CPU [HIT]:

The cache controller will set the data in multiplexer to load in data from the CPU into the SRAM and will send the INDEX+OFFSET portion of the CPU address to the SRAM to locate the right word location, as well as set the SRAM to write mode. The associated dirty bit of the block affected will be set to 1.

Output signals:

rdy = 0
d_in_selector = 0
sram_wr_en = 1
addr_out_8 = INDEX+OFFSET

State 2 – READ a word from cache to CPU [HIT]

The cache controller will set the data out multiplexer to send data from the cache to CPU, and will send the INDEX+OFFSET portion of the CPU address to the SRAM to locate and read the right word location.

Output signals:

rdy = 0
d_out_selector = 1
sram_wr_en = 0
addr_out_8 = INDEX+OFFSET

State 3 – WRITE/READ a word to/from cache from/to CPU [MISS] and dirty bit = 1

The cache controller will preform a block replacement, sending the affected block back to the main memory. It will send a write signal to the main memory, as well as the 16-bit address provided by the CPU, with the OFFSET set to 00000. It will also set the data out multiplexer to send data from the SRAM to the main memory, the SRAM will be set to write mode, and finally when all the signals are ready, will send a memory strobe signal to clock the main memory. After, it will increment the OFFSET up by one, and resent the same signals to send the next word in the block. This will be repeated 32 times for all 32 words in the affected block. Afterwards, the state will change to 4.

Output signals:

rdy = 0
d_out_selector = 0
sram_wr_en = 0
sdram_wr_en = 1
mem_strb = cycles from 0 to 1 in between address increments
addr_out_8 = iterate through from INDEX+00000 to INDEX+FFFFF

addr_out_16 = iterate through from TAG+INDEX+00000 to TAX+INDEX+FFFFF

State 4 – WRITE/READ a word to/from cache from/to CPU [MISS] and dirty bit = 0

The cache controller will need to load in the requested block from the main memory into the SRAM inside the cache. The entire CPU address will be sent to the main memory, with the OFFSET set to 00000. The SRAM will be set to read mode, with the same address sent to the main memory minus the first 8 bits which is associated with the TAG. Once all the signals are ready, the controller will send a memory strobe signal, and increment the OFFSET by 1 and repeat the process 32 times until all words in the affected block are loaded into the cache. Once done, the valid bit will be set to 1, and the dirty bit to 0. The cache will complete the READ or WRITE instruction by jumping to either state 1 if the incoming wr_en signal is 1 or state 2 otherwise.

Output signals:

rdy = 0
d_in_selector = 1
sram_wr_en = 1
sdram_wr_en = 0
mem_strb = cycles from 0 to 1 in between address increments
addr_out_8 = iterate through from INDEX+00000 to INDEX+FFFFF
addr_out_16 = iterate through from TAG+INDEX+00000 to TAX+INDEX+FFFFF

State 5 – Check CPU instruction for HIT or MISS and dirty bit = 1

The cache controller takes in the 16-bit CPU address and splits it into its three fields, the 8-bit TAG, the 3-bit INDEX, and the 5-bit OFFSET. It checks the internal TAG register at the location specified by the INDEX and compares the data inside with the TAG given by the CPU. If these match, it then checks the valid bit of the associated block, if it is a 1, then the instruction is considered a HIT, and the cache will complete the READ or WRITE instruction by jumping to either state 1 if the incoming wr_en signal is 1 or state 2 otherwise. If either of the two checks fail, the instruction is a MISS, and the dirty bit of the affected block to be replaced is checked. If the dirty bit is a 1, it jumps to state 3, otherwise it jumps to state 4.

Output signals:

rdy = 0

Process Diagram

Figure 4.1.5 shows the process diagram for the cache controller.

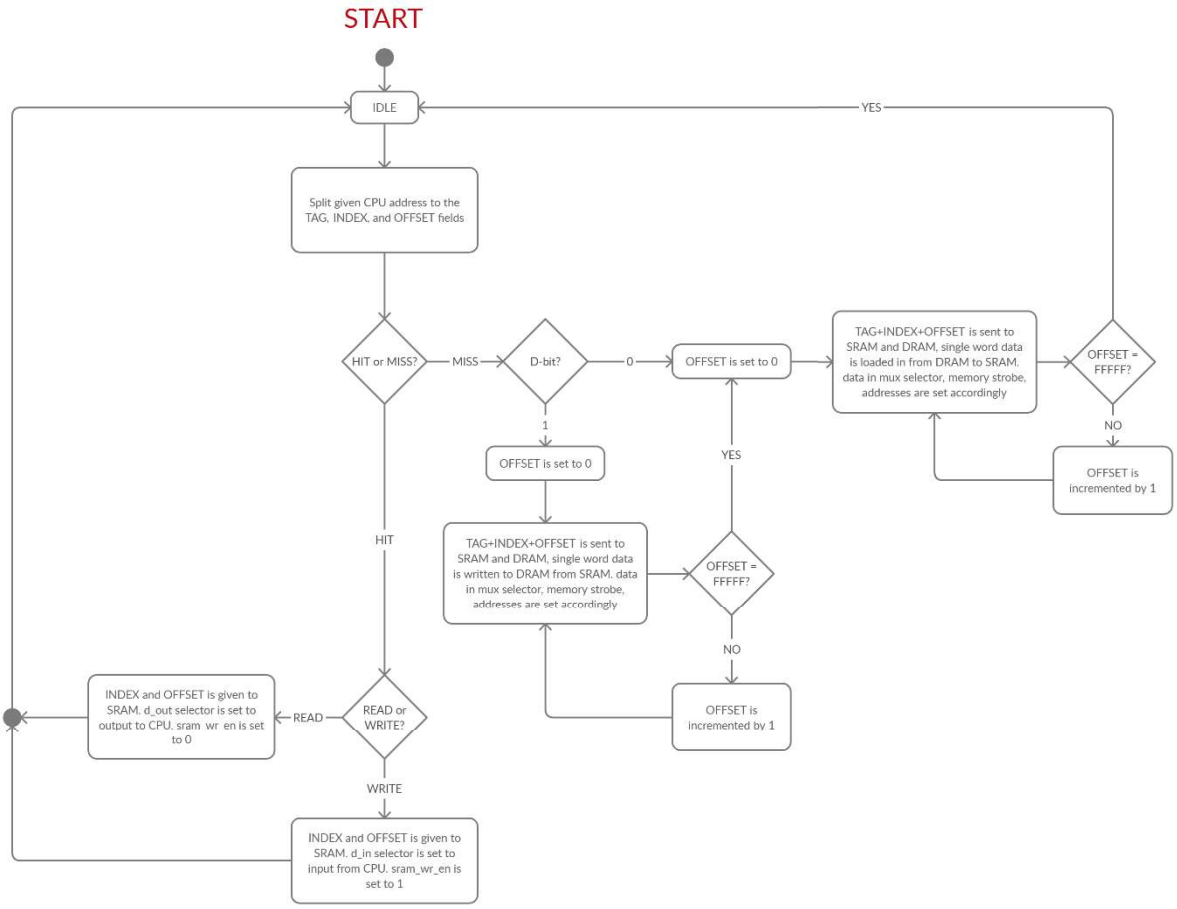

Figure 4.1.5: Cache Controller's Process Diagram

SRAM

The SRAM symbol can be found in *Figure 4.1.5*.

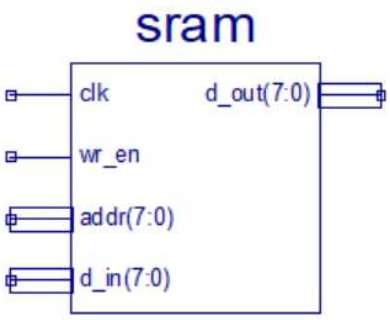

Figure 4.1.5: SRAM Symbol

The SRAM holds 8 blocks of 32 words of data. It takes in an 8-bit address that it breaks down into a 3-bit INDEX and 5-bit OFFSET to pick one of the 8 blocks and one of the 32 words within the block to locate the necessary data.

4.2 Main Memory

For simplicity, the main memory has the DRAM controller and the DRAM implemented in the same symbol, as the focus is on the cache for this project. The main memory symbol can be found in *Figure 4.2.1.*

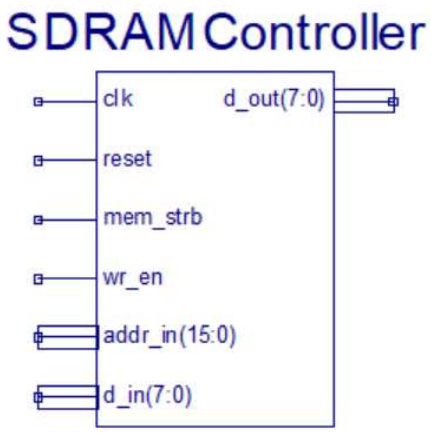

Figure 4.2.1: Main Memory Symbol

The main memory has a 16-bit address in, which it splits into the 3 fields, the 8-bit TAG, 3-bit INDEX, and 5-bit OFFSET, meaning the main memory can store $2^8 \times 2^3$ blocks of 2^5 words each = 256 × 8 blocks of 32 words each in the DRAM.

The DRAM controller, like the cache controller, waits for an incoming address, data, and a write enable signal to determine what to do. There is no state machine for the DRAM controller, as it is essentially a larger capacity SRAM, and completes an action using the incoming memory strobe signal (*mem_strb*).

5. Results

5.1 Timing Diagrams

All the components described in the previous sections were connected as outlined in *Figure 2.2*, and the simulation was run producing the timing diagrams as shown and explained below.

Figure 5.1.1 showcases that it takes 15ns to switch from the initial IDLE (000) state to the MISS state. Since the entire program is reset when initially run, both the cache and main memory is empty. Therefore, the diagram shows the program jumping to the MISS D-BIT = 0 (100) state. Notice that the memory strobe signal switches from high to low every clock cycle, along with the OFFSET portion of the address being sent to the SRAM and main memory starting at 0 and iterating by one every strobe cycle. This represents the loading in of data from the DRAM to the cache.

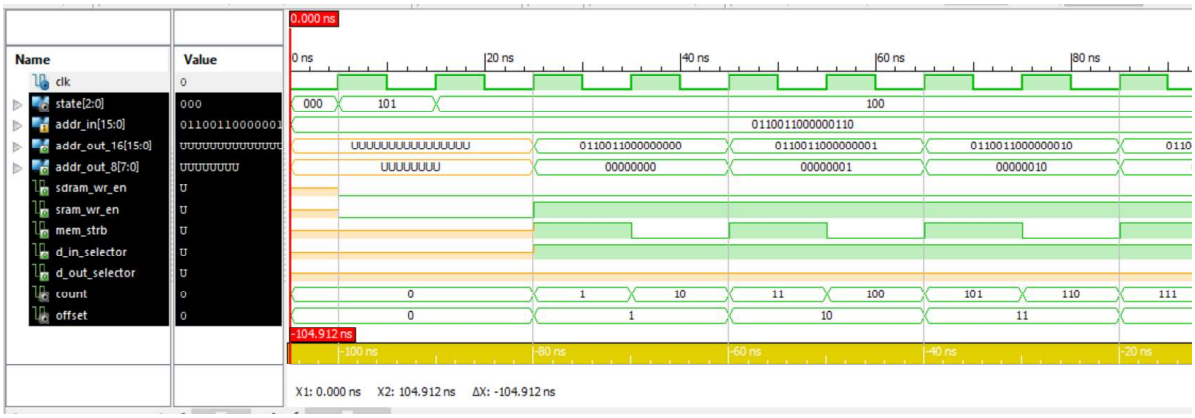

Figure 5.1.1: Initial State

Figure 5.1.2: End of first instruction

Figure 5.1.2 shows the end of the load-in state, taking a total time of 650ns to complete. As shown in the finite state diagram, the program then jumps to the READ [HIT] (010) state since the CPU sent a read signal (wr_en = 0). The READ state takes one clock cycle (10ns) to complete. The cache controller then jumps back to the IDLE (000) state as shown in the finite state machine since it fully completed the CPU instruction and waits for the next instruction.

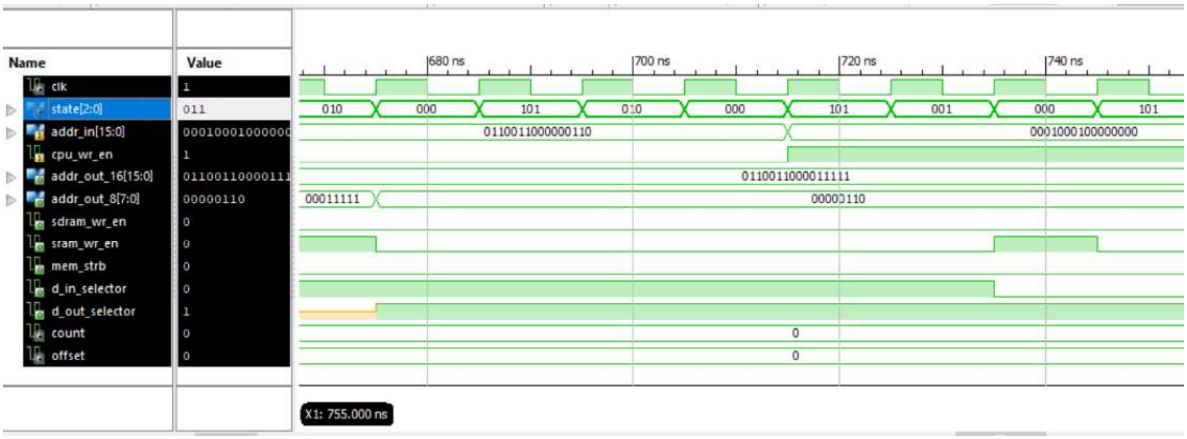

Figure 5.1.3: WRITE instruction

Figure 5.1.3 shows the CPU sending a WRITE instruction for the specified address. The timing diagram shows the state change from IDLE (000) to HIT/MISS CHECK (101) then to the WRITE [HIT] (001) state which takes one clock cycle (10ns) to complete.

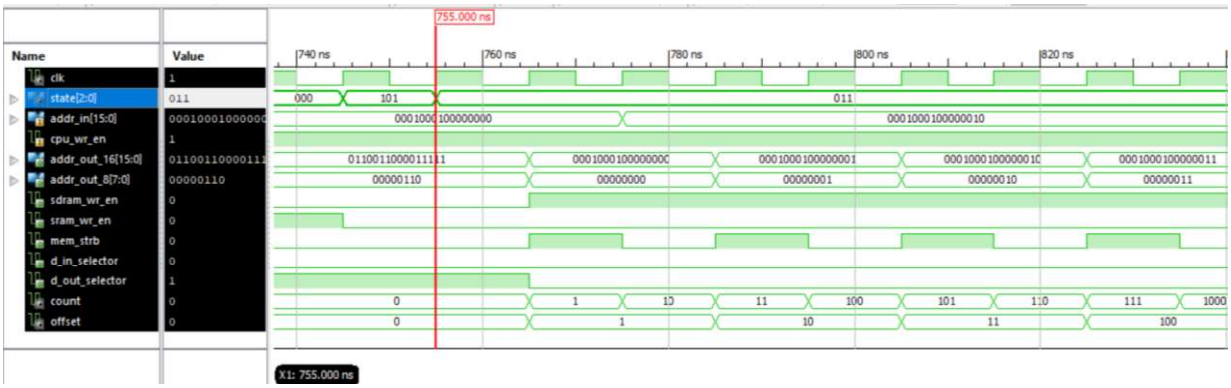

Figure 5.1.4: Block Replacement

Figure 5.1.4 shows the MISS D-BIT = 1 (011) state, where the cache performs a block replacement. The specified block gets written back to main memory from the cache. Notice the memory strobe, address out, SDRAM/SRAM write enable signals reacting accordingly.

Figure 5.1.5: End of Block Replacement

Figure 5.1.5 shows the end of the block replacement (011) state, jumping directly to load the specified block in from the DRAM to the cache (100).

5.2 Calculations

According to the times read in the timing diagrams in the previous section, the following table of values were calculated:

N	CACHE PERFORMANCE PARAMETER	TIME (NS)
1	Hit/Miss determination time	15
2	Data access time	10
3	Block replacement time	650
4	Hit time (Case 1 and 2)	10
5	Miss penalty for Case 3 (D-bit = 0)	665
6	Miss penalty for Case 4 (D-bit =1)	1315

Hit/Miss determination time: Equal to the time it takes to go from the initial state to a MISS or HIT action state = 15ns.

Data access time: Equal to the time it takes the data to be prepped and is equal to one clock cycle = 10ns.

Block replacement time: Data access time + beginning to end of the block replacement state = 650ns. Each memory strobe cycle is 2 clock cycles (20ns) and is the time it takes to write one word. (32 words × 2 clock cycles) + Data access time = 32 × 20ns + 10ns = 650ns.

Hit time: A HIT process takes one clock cycle = 10ns.

Miss penalty for Case 3 (D-bit = 0): Hit/Miss determination time + load in from main memory to SRAM = 15ns + 650ns = 665ns.

Miss penalty for Case 4 (D-bit =1): Hit/Miss determination time + write to main memory from cache + load in block from main memory to cache = 15ns + 650ns + 650ns = 1315ns.

6. Conclusions

This project explored the hierarchies of memory and how they can be structured and utilized to perform at the highest efficiency and speed while keeping costs low, by utilizing the fast and expensive SRAM cache with the slow but cheap DRAM main memory. A cache controller was designed with 6 states, and each state was explored, tested, and measured. As shown, data transfer between main memory and cache is a lot slower than cache to CPU, therefore utilizing the concept of spatial and temporal locality, blocks of data can be loaded in from the main memory to be used in data transfers between CPU and cache, speeding up the data movement process while still able to effectively use the cheaper and slower DRAM memory. An overall deeper understanding of the inner workings of a cache system was obtained through this project.

7. References

[1] COE 758 Project 1 Manual, “Memory Hierarchy – Cache Controller”. Ryerson University. Nov 3,
2020. [Online]. Available:
https://www.ee.ryerson.ca/~lkirisch/ele758/labs/Cache%20Project[12-09-10].pdf

8. Appendix

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.  use IEEE.NUMERIC_STD.ALL;
4.
5.  entity Project1 is
6.      Port (  clk                 : in  STD_LOGIC;
7.                  reset               : in    STD_LOGIC
8.              );
9.  end Project1;
10.
11. architecture Behavioral of Project1 is
12.
13.     -- Signals from Cache to CPU
14.     signal tmp_cpu_rdy      : STD_LOGIC;
15.     signal tmp_cpu_d_in     : STD_LOGIC_VECTOR(7 downto 0);
16.
17.     -- Signals from CPU to Cache
18.     signal tmp_cache_addr_in    : STD_LOGIC_VECTOR(15 downto 0);
19.     signal tmp_cache_wr_en      : STD_LOGIC;
20.     signal tmp_cache_cs         : STD_LOGIC;
21.     signal tmp_cache_mux_in     : STD_LOGIC_VECTOR(7 downto 0);
22.
23.     -- Signals from Cache to SDRAM Controller
24.     signal tmp_sdram_addr_in    : STD_LOGIC_VECTOR(15 downto 0);
25.     signal tmp_sdram_wr_en      : STD_LOGIC;
26.     signal tmp_sdram_memstrb    : STD_LOGIC;
27.     signal tmp_sdram_d_in       : STD_LOGIC_VECTOR(7 downto 0);
28.
29.     -- Signals from SDRAM Controller to Cache
30.     signal tmp_cache_sdram_mux_in       :STD_LOGIC_VECTOR(7 downto 0);
31.
32. begin
33.
34.     -- Instantiating the CPU
35.     CPU_gen: entity work.CPU_gen PORT MAP(
36.         clk => clk,
37.         rst => reset,
38.         trig => tmp_cpu_rdy,
39.         Address => tmp_cache_addr_in,
40.         wr_rd => tmp_cache_wr_en,
41.         cs => tmp_cache_cs,
42.         DOut => tmp_cache_mux_in
43.     );
44.
45.     -- Instantiating the Cache
46.     Cache: entity work.Cache PORT MAP(
47.         clk => clk,
48.         reset => reset,
49.         cs => tmp_cache_cs,
50.         cpu_wr_en => tmp_cache_wr_en,
51.         addr_in => tmp_cache_addr_in,
52.         cpu_d_in => tmp_cache_mux_in,
53.         sdram_d_in => tmp_cache_sdram_mux_in,
54.         cpu_d_out => tmp_cpu_d_in,
55.         sdram_d_out => tmp_sdram_d_in,
56.         addr_out => tmp_sdram_addr_in,
57.         wr_en => tmp_sdram_wr_en,
58.         mem_strb => tmp_sdram_memstrb,
59.         rdy => tmp_cpu_rdy
60.     );
61.
62.     -- Instantiating the SDRAMController
63.     SDRAMController: entity work.SDRAMController PORT MAP(
```

```
        clk => clk,
        reset => reset,
        mem_strb => tmp_sdram_memstrb,
        wr_en => tmp_sdram_wr_en,
        addr_in => tmp_sdram_addr_in,
        d_in => tmp_sdram_d_in,
        d_out => tmp_cache_sdram_mux_in
    );

end Behavioral;
```

8.1 Cache

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

entity Cache is
    Port (  clk                 : in  STD_LOGIC;
                reset               : in    STD_LOGIC;
                cs              : in    STD_LOGIC;
                cpu_wr_en       : in    STD_LOGIC;
                addr_in         : in    STD_LOGIC_VECTOR (15 downto 0);
                cpu_d_in            : in    STD_LOGIC_VECTOR (7 downto 0);
                sdram_d_in      : in    STD_LOGIC_VECTOR (7 downto 0);
                cpu_d_out       : out STD_LOGIC_VECTOR (7 downto 0);
                sdram_d_out     : out STD_LOGIC_VECTOR (7 downto 0);
                addr_out        : out   STD_LOGIC_VECTOR (15 downto 0); -- To SDRAM controller
                wr_en               : out STD_LOGIC; -- SDRAM write enable
                mem_strb        : out STD_LOGIC;
                rdy                 : out STD_LOGIC);
end Cache;

architecture Behavioral of Cache is

    -- Declaring internal signals of the cache that connect between the
    -- Cache Controller and Cache SRAM
    signal tmp_addr     : STD_LOGIC_VECTOR (7 downto 0);
    signal tmp_wr_en        : STD_LOGIC;

    -- Internal signals to connect the data in mux to SDRAM and Cache Controller
    signal tmp_mux_in_selector      : STD_LOGIC;
    signal tmp_mux_in_output        : STD_LOGIC_VECTOR (7 downto 0);

    -- Internal signals to connect the data out mux to SDRAM and Cache Controller
    signal tmp_mux_out_selector     : STD_LOGIC;
    signal tmp_mux_out_input            : STD_LOGIC_VECTOR (7 downto 0);


begin
    -- Instantiating the data in mux to determine which input (CPU
    -- or SDRAM) will be sent to SRAM
    d_in_mux: entity work.d_in_mux PORT MAP(
        selector => tmp_mux_in_selector,
        d_in_0 => cpu_d_in,
        d_in_1 => sdram_d_in,
        d_out => tmp_mux_in_output
    );

    -- Instantiating the data out mux to determine which output (CPU
    -- or SDRAM) will the SRAM data be sent to
    d_out_mux: entity work.d_out_mux PORT MAP(
        selector    => tmp_mux_out_selector,
        d_in            => tmp_mux_out_input,
```

```
        d_out_0         => sdram_d_out,
        d_out_1         => cpu_d_out
    );

    -- Instantiating the CacheController
    CacheController: entity work.CacheController PORT MAP(
        clk                     => clk,
        reset               => reset,
        cs                  => cs,
        cpu_wr_en           => cpu_wr_en,
        addr_in                 => addr_in,
        addr_out_16         => addr_out,
        addr_out_8          => tmp_addr,
        sdram_wr_en         => wr_en,
        sram_wr_en          => tmp_wr_en,
        mem_strb            => mem_strb,
        d_in_selector       => tmp_mux_in_selector,
        d_out_selector  => tmp_mux_out_selector,
        rdy                     => rdy
    );

    -- Instantiating the SRAM
    sram : entity work.sram PORT MAP (
    clk             => clk,
    wr_en       => tmp_wr_en,
    addr        => tmp_addr,
    d_in        => tmp_mux_in_output,
    d_out       => tmp_mux_out_input
    );

end Behavioral;
```

Data in Mux

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

entity d_in_mux is
    Port ( selector     : in    STD_LOGIC;
           d_in_0   : in    STD_LOGIC_VECTOR (7 downto 0);
           d_in_1   : in    STD_LOGIC_VECTOR (7 downto 0);
           d_out        : out   STD_LOGIC_VECTOR (7 downto 0)
    );
end d_in_mux;

architecture Behavioral of d_in_mux is

begin
    d_out <= d_in_0 when selector = '0' else d_in_1;
end Behavioral;
```

Data out Mux

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

entity d_out_mux is
    Port ( selector     : in  STD_LOGIC;
             d_in       : in  STD_LOGIC_VECTOR (7 downto 0);
           d_out_0  : out  STD_LOGIC_VECTOR (7 downto 0);
           d_out_1  : out  STD_LOGIC_VECTOR (7 downto 0)
             );
end d_out_mux;
```

```
11.
12.  architecture Behavioral of d_out_mux is
13.
14.  begin
15.      if (selector = '0') then
16.          d_out_0 <= d_in;
17.      else
18.          d_out_1 <= d_in;
19.      end if;
20.  end Behavioral;
```

Cache Controller

```
1.   library IEEE;
2.   use IEEE.STD_LOGIC_1164.ALL;
3.   use IEEE.NUMERIC_STD.ALL;
4.
5.   entity CacheController is
6.       Port (  clk                 : in  STD_LOGIC;
7.                   reset               : in    STD_LOGIC;
8.                   cs              : in    STD_LOGIC;
9.                   cpu_wr_en       : in    STD_LOGIC;
10.                  addr_in         : in    STD_LOGIC_VECTOR (15 downto 0);
11.                  addr_out_16     : out   STD_LOGIC_VECTOR (15 downto 0);
12.                  addr_out_8      : out STD_LOGIC_VECTOR (7 downto 0);
13.                  sdram_wr_en     : out STD_LOGIC;
14.                  sram_wr_en      : out STD_LOGIC;
15.                  mem_strb        : out STD_LOGIC;
16.                  d_in_selector   : out STD_LOGIC;
17.                  d_out_selector  : out STD_LOGIC;
18.                  rdy                 : out STD_LOGIC);
19.  end CacheController;
20.
21.  architecture Behavioral of CacheController is
22.
23.      -- States
24.      -- 0: 000: Idle state
25.      -- 1: 001: Write a word to cache from CPU [hit]
26.      -- 2: 010: Read a word from cache to CPU [hit]
27.      -- 3: 011: [miss] and dirty bit (d_bit) is 1 - perform block replacement + run state 4
28.      -- 4: 100: [miss] and dirty bit (d_bit) is 0 - read in from SDRAM (main memory) to SRAM
29.      --      + run state 1 or 2 based on the CPU instruction
30.      -- 5: 101: When cs is triggered, check if hit or miss
31.
32.      signal state        : STD_LOGIC_VECTOR(2 downto 0)  := "000";
33.
34.      -- Address word register
35.      signal cpu_tag      : STD_LOGIC_VECTOR(7 downto 0)  := "00000000";
36.      signal cpu_index    : STD_LOGIC_VECTOR(2 downto 0)  := "000";
37.      signal cpu_offset   : STD_LOGIC_VECTOR(4 downto 0)  := "00000";
38.
39.      signal vbit         : STD_LOGIC_VECTOR(7 downto 0)  := "00000000"; -
    - Represents array of vbits (valid bits), 1 per block (total 8 blocks)
40.      signal dbit         : STD_LOGIC_VECTOR(7 downto 0)  := "00000000"; -- Dirty bit(s)
41.
42.      -- Build the tag register block, 8 addresses of  8 bit tags will be stored
43.      type typeRAM is array (7 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
44.      signal memTAG : typeRAM := ((others => (others => '0')));
45.
46.      -- Iterator through the whole block for data transfer from/to main memory
47.      signal count : integer := 0;
48.      signal offset : integer := 0;
49.
50.  begin
51.      process(clk)
```

```
52.
53.     begin
54.         if (clk'EVENT and clk='1') then
55.             case state is
56.                 when "000" => -- IDLE
57.                     rdy             <= '1';
58.                     sdram_wr_en <= '0';
59.                     sram_wr_en  <= '0';
60.
61.                     -- Set next state
62.                     --if (cs = '1') then
63.                         -- CPU strobed, meaning an instruction is coming in
64.
65.                         -- Save the CPU address into it's parts for reference
66.                         cpu_tag         <= addr_in(15 downto 8);
67.                         cpu_index   <= addr_in(7 downto 5);
68.                         cpu_offset  <= addr_in(4 downto 0);
69.
70.                         state <= "101";
71.                     --end if;
72.                 when "001" => -- WRITE WORD TO CACHE FROM CPU
73.                     sram_wr_en                              <= '1';
74.                     sdram_wr_en                             <= '0';
75.                     addr_out_8(7 downto 5)              <= cpu_index(2 downto 0);
76.                     addr_out_8(4 downto 0)              <= cpu_offset(4 downto 0);
77.                     d_in_selector                           <= '0';
78.                     dbit(to_integer(unsigned(cpu_index)))   <= '1';
79.
80.                     state <= "000";
81.                 when "010" => -- READ WORD TO CPU FROM CACHE
82.                     sdram_wr_en                 <= '0';
83.                     sram_wr_en                  <= '0';
84.                     addr_out_8(7 downto 5)  <= cpu_index(2 downto 0);
85.                     addr_out_8(4 downto 0)  <= cpu_offset(4 downto 0);
86.                     d_out_selector              <= '1';
87.
88.                     state <= "000";
89.                 when "011" => -- BLOCK REPLACEMENT (WRITE BLOCK TO MAIN MEMORY), THEN STATE 4 (100)
90.                     vbit(to_integer(unsigned(cpu_index)))   <= '0';
91.                     sdram_wr_en <= '1';
92.                     sram_wr_en <= '0';
93.                     d_out_selector <= '0';
94.
95.                     if (count = 64) then
96.                         count <= 0;
97.                         offset <= 0;
98.                         state <= "100";
99.                     else
100.                        if (count mod 2 = 1) then
101.                            mem_strb <= '0';
102.                        else
103.                            mem_strb <= '1';
104.
105.                        -- Load 1 block, one word at a time
106.                        -- Sets the address of the main memory to the block location with offset from 0-
    31 to grab whole block
107.                        addr_out_16(15 downto 8)    <= cpu_tag(7 downto 0);
108.                        addr_out_16(7 downto 5)     <= cpu_index(2 downto 0);
109.                        addr_out_16(4 downto 0)     <= STD_LOGIC_VECTOR(to_unsigned(offset, 5));
110.                        -- Sets the address of the SRAM to the index location to write in the data from main memory
111.                        addr_out_8(7 downto 5)      <= cpu_index(2 downto 0);
112.                        addr_out_8(4 downto 0)      <= STD_LOGIC_VECTOR(to_unsigned(offset, 5));
113.
114.                        offset <= offset + 1;
115.                    end if;
116.                    count <= count + 1;
```

```
                    end if;
                when "100" => -- LOAD BLOCK FROM MAIN MEMORY
                vbit(to_integer(unsigned(cpu_index))) <= '0';
                sdram_wr_en <= '0';
                sram_wr_en <= '1'; --enable sram for receiving data
                d_in_selector <= '1';
                if (count = 64) then
                        count <= 0;
                        offset <= 0;
                        vbit(to_integer(unsigned(cpu_index)))       <= '1';
                        memTAG(to_integer(unsigned(cpu_index))) <= cpu_tag(7 downto 0);
                        if (cpu_wr_en = '1') then
                            state <= "001";
                        else
                            state <= "010";
                        end if;
                    else
                        if (count mod 2 = 1) then --strobe on every other clck cycle
                            mem_strb <= '0';
                        else
                            mem_strb <= '1';

                            -- Load 1 block, one word at a time
                            -- Sets the address of the main memory to the block location with offset from 0-31 to grab whole block
                            addr_out_16(15 downto 8)    <= cpu_tag(7 downto 0);
                            addr_out_16(7 downto 5)     <= cpu_index(2 downto 0);
                            addr_out_16(4 downto 0)     <= STD_LOGIC_VECTOR(to_unsigned(offset, 5));
                            -- Sets the address of the SRAM to the index location to write in the data from main memory
                            addr_out_8(7 downto 5)      <= cpu_index(2 downto 0);
                            addr_out_8(4 downto 0)      <= STD_LOGIC_VECTOR(to_unsigned(offset, 5));

                            offset <= offset + 1;
                        end if;
                        count <= count + 1;
                    end if;
                when "101" => -- CHECK CPU INSTRUCTION
                    rdy <= '0';
                    sdram_wr_en <= '0';
                    sram_wr_en  <= '0';

                    -- Check if it's a HIT or MISS
                    -- Check to see if it's a hit or miss by going to the tag memory block at address [cpu_index]
                    --   and comparing it with the [cpu_tag] to see if they match
                    -- Also check if the valid bit of the block is 1
                    if ((memTAG(to_integer(unsigned(cpu_index))) = cpu_tag) and (vbit(to_integer(unsigned(cpu_index))) = '1')) then
                        -- HIT
                        -- Check if it's a write or read instruction
                        if (cpu_wr_en = '1') then
                            -- HIT-WRITE instruction
                            state <= "001";
                        else
                            -- HIT-READ instruction
                            state <= "010";
                        end if;
                    else
                        -- MISS, need to read in from the main memory to cache
                        -- Check if d-bit = 1 to see if cache needs to perform block replacement (write to main mem from cache)
                        if (dbit(to_integer(unsigned(cpu_index))) = '1') then
                            -- Perform block replacement, then read in cpu address memory block from main memory
                            -- MISS d-bit = 1
                            state <= "011";
                        else
                            -- No block replacement, just read in cpu address memory block from main memory
```

```
                            -- MISS d-bit = 0
                            state <= "100";
                        end if;
                    end if;
                when others =>
            end case;
        end if;
    end process;

end Behavioral;
```

SRAM

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

entity sram is
    Port ( clk : in  STD_LOGIC;
           addr : in  STD_LOGIC_VECTOR (7 downto 0);
           d_in : in  STD_LOGIC_VECTOR (7 downto 0);
           d_out : out  STD_LOGIC_VECTOR (7 downto 0);
           wr_en : in  STD_LOGIC);
end sram;

architecture Behavioral of sram is

    type ram_type is array (7 downto 0, 31 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
    signal SRAM: ram_type;
    signal initializer : integer := 0;

begin

process (CLK)
begin
    if (CLK'event and CLK = '1') then

        if initializer = 0 then
            for I in 0 to 7 loop
                for J in 0 to 31 loop
                    SRAM(i,j) <= "00000000";
                end loop;
            end loop;
            initializer <= 1;
        end if;


        if (wr_en = '1') then
            SRAM(to_integer(unsigned(addr(7 downto 5))), to_integer(unsigned(addr(4 downto 0)))) <= d_in;
        else
            d_out <= SRAM(to_integer(unsigned(addr(7 downto 5))), to_integer(unsigned(addr(4 downto 0))));
        end if;

    end if;
end process;
end Behavioral;
```

8.2 Main Memory

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

```

```
entity SDRAMController is
    Port (  clk                 : in  STD_LOGIC;
                reset               : in    STD_LOGIC;
                mem_strb        : in    STD_LOGIC;
                wr_en               : in    STD_LOGIC;
                addr_in         : in    STD_LOGIC_VECTOR (15 downto 0);
                d_in                : in  STD_LOGIC_VECTOR (7 downto 0);
                d_out               : out STD_LOGIC_VECTOR (7 downto 0));
end SDRAMController;

architecture Behavioral of SDRAMController is

    type ram_type is array (7 downto 0, 31 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
    signal SDRAM: ram_type;

begin

process (CLK)
begin
  if (CLK'event and CLK = '1') then
        if (mem_strb = '1') then
            if (wr_en = '1') then
                SDRAM(to_integer(unsigned(addr_in(7 downto 5))), to_integer(unsigned(addr_in(4 downto 0)))) <= d_in;
            else
                d_out <= SDRAM(to_integer(unsigned(addr_in(7 downto 5))),to_integer(unsigned(addr_in(4 downto 0))));
            end if;
        end if;
  end if;
end process;
end Behavioral;
```