ECSE 425 Project Report 2 Direct-Mapped Cache

Edem Nuviadenu, 260779440
Nathanael Lemma, 260779759
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec

I. Introduction

In this deliverable, we built and tested a finite-state machine that simulates a basic, direct-mapped cache circuit in VHDL. Our cache has two set of ports - the processor-cache interface ports and the main_memory-cache interface ports. Each port is further divided into input and output ports depending on the data flow with respect to the cache. Some of the ports such as s_write and s_read served as signal ports where as s_writedata and m_readdata served as data ports. A detailed implementation of the system along side with a discussion of our finite state machine diagram will be discussed in this report. Additionally, calculation of the number of bits required for tag, block index, and block offset and including results from our tests will also be discussed.

II. Cache

A. Bit Calculation

In order to determine how many bits are required for the processor to determine the block offset, we deduce that given 128 bit blocks (4 words = 2^2) in the cache data block, then we can represent the **block offset** by 2 bits. Because words are 32 bit, there are 4 bytes per word hence 2 bits to represent **word offset**, and 2 bits to represent **byte offset** within the word. To determine the **block index**, we know that the total data storage bit of the cache is 4096 and a block bit = 128, thus number of addresses = $\frac{4096}{128} = \frac{2^{12}}{2^7}$ = 32 blocks (ie 31 - 0) = 2^5. Thus 5 bits for **block index**. Given that main memory has only 2^{15} bytes, our **tag** can be addressed by 15 - block address bit - block offset bit = $15 - 5 - 2 = 8$ bits. Assumptions: last two bits of input address ignored, cache should only use lower 15 bits of address.

B. State Diagram Implementation

In building the State diagram, we identified the various mutually exclusive states the direct mapped cache would need to be in in order to process a request from the processor. As such we identified 5 states namely **c_idle, c_read, c_write, write-back, m_fetch**. The diagram for our finite state machine implementation can be found in Figure 1.

The notations used in the diagram are: *r* = *s_read*, *w* = *s_write*, *t* = *tag*, *v* = *valid bit* and *d* = *dirty bit*. Additionally, *byte_count* == 15 implies the end of block fetch.

The system is triggered to leave the idle state whenever any of the *r* or *w* signals are high. After that, the next 4 possible states it could end up in are determined based on other parameters that will be explained below. Implicitly, we are assuming that both the *r* and *w* signals cannot be high at the same time.

In the case of the processor wanting to write (w=1), the system would go to the c_write state if the specified block that the data is to be written on exists in the cache (v=1) and our tag matches our address. Similarly, if the processor wants to read data from a specific address in the memory and if the specified block exists in the cache we go to the c_read state.

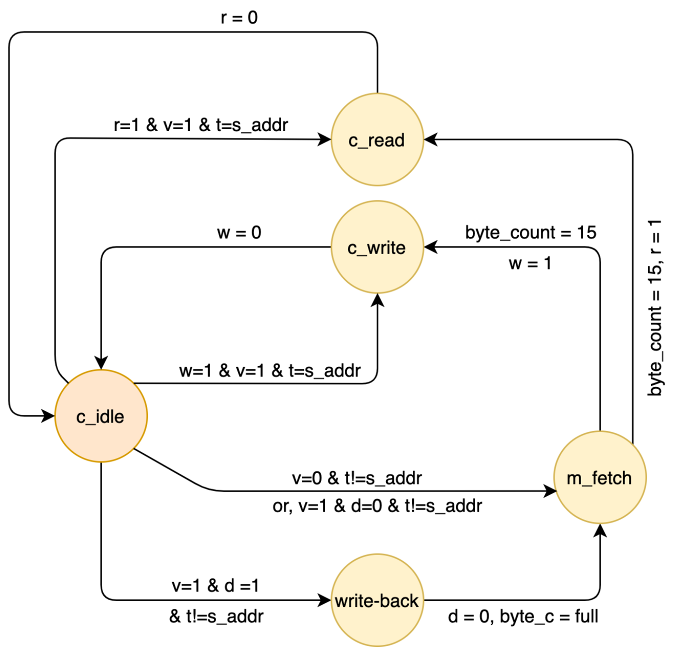


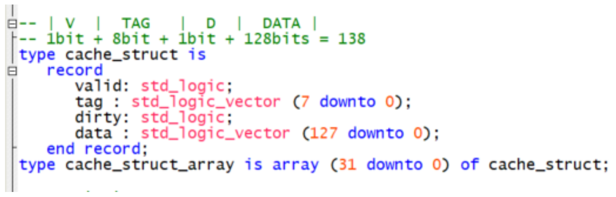
Fig. 1. State Diagram

However, not always is it the case that the the requested block exists in the cache. When this happens, that means the tag does not match the specified address (t!=s_address). In this scenario, whether we want to write (w=1) or read (r=1), the requested block should be loaded to the cache. If the area of the cache that the requested block is to be written on is free (v=0) or is preoccupied by another block (v=1) but the data there has not been modified yet (d=0), the system transitions to the m_fetch state. This is where we fetch date from main memory to the corresponding address on the cache. After this is done, the system can go back to the c_read if (r=1) or c_write if (w=1) state.

In the case of the data being preoccupied by another block (v=1) and the data there has been modified (d=1), the system transitions to the write-back state to save the content of the block to be replaced to memory. Then, we can proceed to the m_fetch state and carry on tasks as discussed above.

C. Code Design

The state machine was implemented in VHDL by first identifying the port map of the structure, which involved clk, reset, an Avalon interface. In defining the architecture of the cache we created a cache structure record to represent a row we might have in a typical cache structure with columns representing the valid bit = 1 bit, tag bit = 8 bits (7 downto 0), dirty bit = 1 bit, data block = (127 downto 0). This is shown in Fig 2.

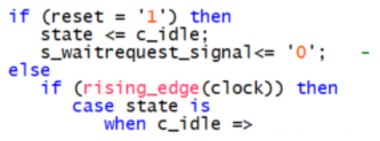


```
-- | V  |  TAG   |  D  |  DATA |
-- 1bit + 8bit + 1bit + 128bits = 138
type cache_struct is
   record
      valid: std_logic;
      tag : std_logic_vector (7 downto 0);
      dirty: std_logic;
      data : std_logic_vector (127 downto 0);
   end record;
type cache_struct_array is array (31 downto 0) of cache_struct;
```

Fig. 2. Cache Structure declaration

1) Process Block:

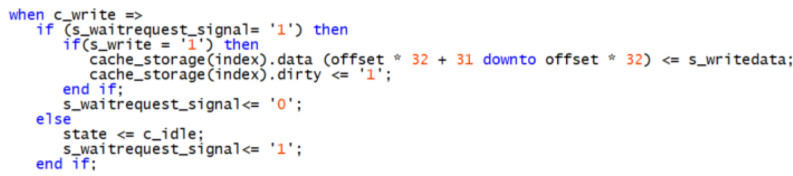
1) Process Block: This contains the key logic of cache and sequential statements can be seen. Prior to beginning the sequential statements, we obtain the respective values of the cache/block index, the data offset and the tag from the s_addr signal. Our sensitivity list consists of the clock, reset, state, tag, wb_address, index, offset, address, s_addr. We further implement an asynchronous reset on active high which changes our state to the idle state and resets s_waitrequest signal.



```
if (reset = '1') then
   state <= c_idle;
   s_waitrequest_signal<= '0';   -
else
   if (rising_edge(clock)) then
      case state is
         when c_idle =>
```

Fig. 3. reset

The s_waitrequest is always high by default and we do this to avoid indefinite waiting. However, upon the rising edge of a clock we check the current state of the cache and implement the appropriate state logic. Most transitions are done using the NSL (Next State Logic) approach. A sample of state is shown in Fig 4.



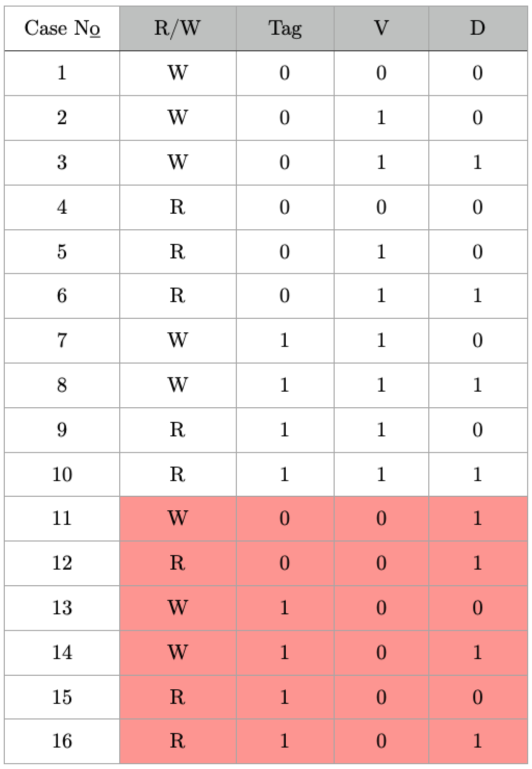
```
when c_write =>
   if (s_waitrequest_signal= '1') then
      if(s_write = '1') then
         cache_storage(index).data (offset * 32 + 31 downto offset * 32) <= s_writedata;
         cache_storage(index).dirty <= '1';
      end if;
      s_waitrequest_signal<= '0';
   else
      state <= c_idle;
      s_waitrequest_signal<= '1';
   end if;
```

Fig. 4. Sample state c-write

III. Results

A. Possible & ImpossibleCases

Prior to testing out our implementation we made sure to highlight which cases are possible to occur in a direct mapped cache. In total there are 6 scenarios that we identified will not occur these are highlighted in red in fig 5. A hit is represented by Tag = 1 and miss by Tag = 0. We see that for case 11, where we have a write, it is impossible to have the valid bit = 0 and the dirty bit high (D = 1), since the valid bit represents the fact that there is an appropriate block in memory and it only when valid = 1 that the block may become dirty. This logic applies to case 12 as well. For cases 13 and 15, its impossible to have a hit (Tag = 1) when there is no valid (V = 0) data block in the cache since the tag is set upon a valid fetch from memory, and for cases 14 and 16, it follows an earlier logic that an invalid block cannot be set dirty with a hit (Tag = 1) since only a valid block can be set dirty and a tag associated to it.



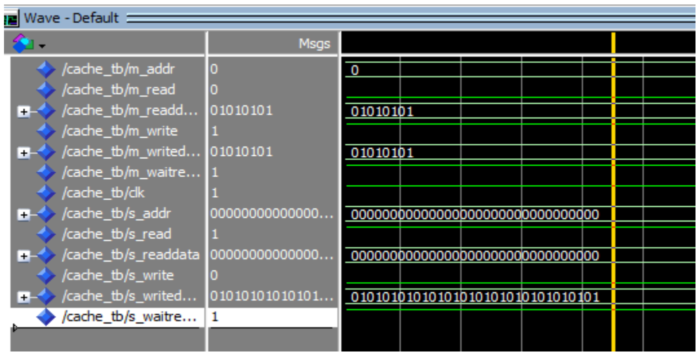
Case No	R/W	Tag	V	D
1	W	0	0	0
2	W	0	1	0
3	W	0	1	1
4	R	0	0	0
5	R	0	1	0
6	R	0	1	1
7	W	1	1	0
8	W	1	1	1
9	R	1	1	0
10	R	1	1	1
11	W	0	0	1
12	R	0	0	1
13	W	1	0	0
14	W	1	0	1
15	R	1	0	0
16	R	1	0	1

Fig. 5. Possible and Impossible cases.

B. ModelSim Tests and WaveForm

Our complete test run on 10 test cases using 5 main tests, due to eliminating the possible scenarios as shown above.

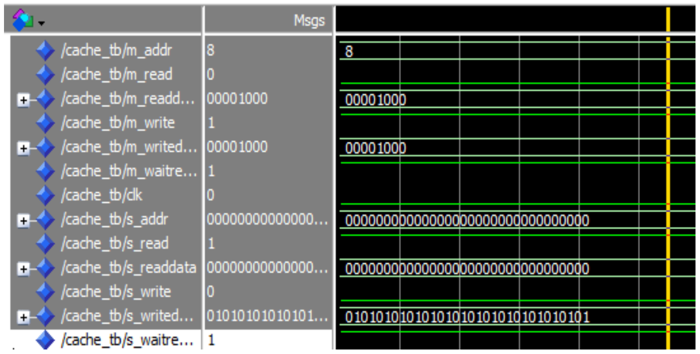
1) Test 1: This test handles the **scenarios 1 and 9** as shown in Fig 5. Scenario 1 is where the command is a write, block is clean, we have a miss and invalid. Scenario 9 is when we read and the block is clean and there is a hit and block is valid. The initial address of ”00000000000000000000000000000000” is passed in s_addr and since there is no value in memory on the first iteration, case 1 is tested where the memory block is brought back into index ”00000” and then clean data read. Value written VAL1 = ”01010101010101010101010101010101”. Image below shows the wave form from ModelSim.



```
Wave - Default
                          Msgs
/cache_tb/m_addr          0                  0
/cache_tb/m_read          0
/cache_tb/m_readd...      01010101           01010101
/cache_tb/m_write         1
/cache_tb/m_writed...     01010101           01010101
/cache_tb/m_waitre...     1
/cache_tb/clk             1
/cache_tb/s_addr          00000000000000...  00000000000000000000000000000000
/cache_tb/s_read          1
/cache_tb/s_readdata      00000000000000...  00000000000000000000000000000000
/cache_tb/s_write         0
/cache_tb/s_writed...     01010101010101...  01010101010101010101010101010101
/cache_tb/s_waitre...     1
```

Fig. 6. Sample wave simulation Test 1.

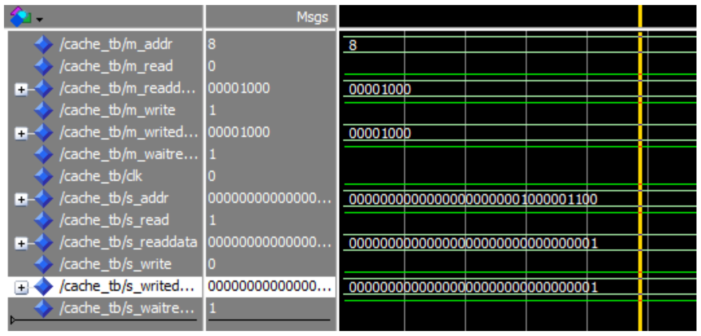
2) Test 2: This test handles the **scenarios 7 and 10** as shown in table Fig 5. Scenario 7 is where prior to a write, the block is clean, we have a hit and block is valid. Scenario 10 is when we read and the block is dirty and there is a hit and block is valid. Following the first test, the same s_addr is used as such we write the dirty data back to main memory, bring in clean data and write VAL2 = ”00000000000000000000000000000001” at that location, then we have a subsequent read hit to the dirty block. The results shown below.



```
                          Msgs
/cache_tb/m_addr          8                  8
/cache_tb/m_read          0
/cache_tb/m_readd...      00001000           00001000
/cache_tb/m_write         1
/cache_tb/m_writed...     00001000           00001000
/cache_tb/m_waitre...     1
/cache_tb/clk             0
/cache_tb/s_addr          00000000000000...  00000000000000000000000000000000
/cache_tb/s_read          1
/cache_tb/s_readdata      00000000000000...  00000000000000000000000000000000
/cache_tb/s_write         0
/cache_tb/s_writed...     01010101010101...  01010101010101010101010101010101
/cache_tb/s_waitre...     1
```

Fig. 7. Sample wave simulation Test 2.

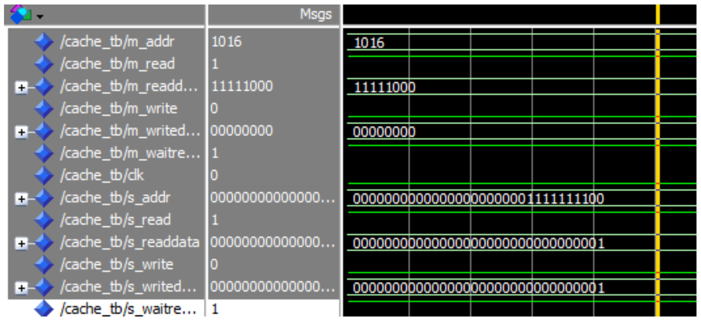
3) Test 3: This test handles the scenario where we have a read miss, but data block is dirty from earlier write and valid and a read miss where the data block is still clean and now valid. This covers **scenarios 3 and 5**. Given the different tag address with the same index ie s_addr = ”00000000000000000000001000001100”, we initially encounter scenario 3 and upon a write back of VAl3 = X”0F0E0D0C” we follow up with a read for Val2 at the same address which subsequently puts us in scenario 5 such that there will be a miss although the data block is valid and clean.



```
                          Msgs
/cache_tb/m_addr          8                  8
/cache_tb/m_read          0
/cache_tb/m_readd...      00001000           00001000
/cache_tb/m_write         1
/cache_tb/m_writed...     00001000           00001000
/cache_tb/m_waitre...     1
/cache_tb/clk             0
/cache_tb/s_addr          00000000000000...  00000000000000000000001000001100
/cache_tb/s_read          1
/cache_tb/s_readdata      00000000000000...  00000000000000000000000000000001
/cache_tb/s_write         0
/cache_tb/s_writed...     00000000000000...  00000000000000000000000000000001
/cache_tb/s_waitre...     1
```

Fig. 8. Sample wave simulation Test 3.

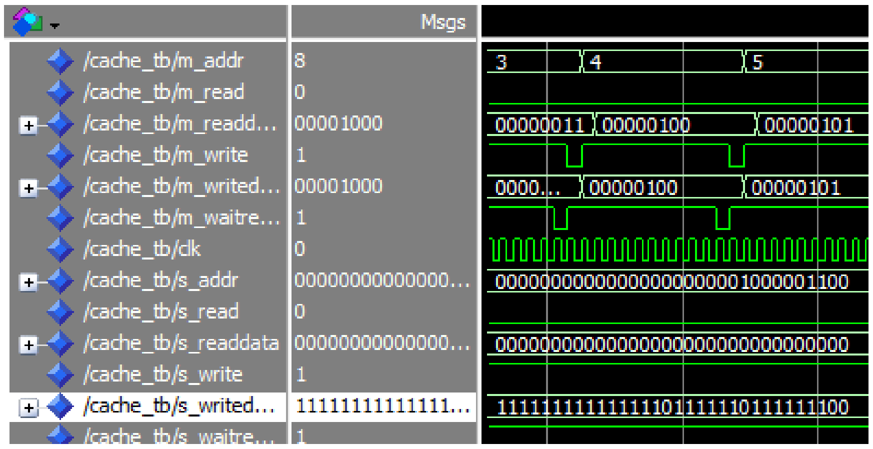
4) Test 4: This test handles the **scenarios 4 and 6**. In scenario 4 we have a read miss, but data block is clean and invalid. In scenario 6 we have a read miss where the data block is dirty and now valid. Given the different tag address with the same index, s_addr=”00000000000000000000001111111100” we initially encounter scenario 4 and upon a write back of VAl3=X”0F0E0D0C” we follow up with a read for Val2 at the same address which subsequently puts us in scenario 6 such that there will be a miss although the data block is valid and clean.



```
                          Msgs
/cache_tb/m_addr          1016               1016
/cache_tb/m_read          1
/cache_tb/m_readd...      11111000           11111000
/cache_tb/m_write         0
/cache_tb/m_writed...     00000000           00000000
/cache_tb/m_waitre...     1
/cache_tb/clk             0
/cache_tb/s_addr          00000000000000...  00000000000000000000001111111100
/cache_tb/s_read          1
/cache_tb/s_readdata      00000000000000...  00000000000000000000000000000001
/cache_tb/s_write         0
/cache_tb/s_writed...     00000000000000...  00000000000000000000000000000001
/cache_tb/s_waitre...     1
```

Fig. 9. Sample wave simulation Test 4.

5) Test 5: This test handles the remaining **scenarios 2 and 8** from Fig 5. where there is a write for VAL3 at an address with a different tag (miss) and different index to trigger a write back that brings in a block that is valid and clean as seen in scenario 2, s_addr=”00000000000000000000001000001100” following this, we test the scenario where we have a write hit fro VAL4 = X”FFFEFDFC” at the same s_addr used, such that the data block is dirty before the write, and valid, causing another write back as seen in scenario 8.



	Msgs	
/cache_tb/m_addr	8	3 · 4 · 5
/cache_tb/m_read	0	
/cache_tb/m_readd...	00001000	00000011 · 00000100 · 00000101
/cache_tb/m_write	1	
/cache_tb/m_writed...	00001000	0000... · 00000100 · 00000101
/cache_tb/m_waitre...	1	
/cache_tb/clk	0	
/cache_tb/s_addr	00000000000000...	00000000000000000000001000001100
/cache_tb/s_read	0	
/cache_tb/s_readdata	00000000000000...	00000000000000000000000000000000
/cache_tb/s_write	1	
/cache_tb/s_writed...	11111111111111...	11111111111111101111110111111100
/cache_tb/s_waitre...	1	

Fig. 10. Sample wave simulation.