ECSE 425: Project #2 Report

ECSE 425: Computer Architecture
Group Number: 23
Group Members: Karl Doumar, Edward Latulipe-Kang
Student ID: 260733160, 260746475
Professor: Amin Emad
March 9th, 2021

ECSE 425 Project #2: Direct-Mapped Cache

Introduction

For this project, we are tasked with implementing a basic direct-mapped cache circuit. In further detail, the project enforces the concepts learned in class regarding cache and focuses on the technicals of implementing a memory hierarchy given some constraints.

Cache Design

To begin, we have chosen to implement our cache with 5 total states: idle, read, write, writeback, and replace. The 5 states were created to most accurately describe the behaviour of the system at any time. The implementation of our states can be seen in the appendix at the end of the report. The idle state reflects the cache system when it is idling or in the process of determining its next state. This is where the bulk of the conditional statements take place. Said conditional statements are important for assessing bit streams and delegating the cache. The read and write states are self-explanatory, and are accessed whenever a read from or a write to cache is required. The writeback state is required if data needs to be saved into main memory. This can occur when a valid dirty cache block is accessed. This access may be due to a read or a write. Nonetheless, data must be stored. Finally, the replace state is needed to replace a cache block in cache. This state will retrieve a cache block in main memory to store in some designated position in cache and is arguably the most crucial state. All states are designed to be modular, and relatively stand-alone, but given the nature of cache, these states will switch among themselves and in some specific order. For instance, a read may happen on its own, after a replace, or after a writeback and a replace. Of course, this depends on the cache hits or misses that our implementation experiences. Please look at the state machine for more detail in Figure 5A of the appendix.

Once we established the actual behaviour of our cache system, we needed to appropriately divide our incoming CPU source address into tag bits, block indices, and block offsets. To do this, we simply followed the cache structure that was provided to us. Please note that the block offset contains the word and byte offset. We used 2 bits for the byte offset because we are dealing with 32-bit words. This indicates that each word will contain 4 bytes, and so we need to be able to identify all 4 bytes. In terms of the word offset, we also required 2 bits because a cache block can only contain 4 words. Following a similar reasoning, to be able to switch between the 4 words, 2 bits are needed. Therefore, in total 4 bits are required for the block offset. In terms of the block index, given that we have 4096 bits for data storage and 128-bit blocks, we see that we can only fit 32 blocks in cache. To represent all 32 blocks we require 5 bits. In total, the block index and block offset will require 9 bits. The remaining bits will be considered tag bits. Here, it is important to note that we are to only use the least-significant, or lower, 15 bits. Because 9 bits are already occupied, we are left with 6 bits to represent the tag portion of the instruction or source address. These assumptions were adhered to for the remainder of the project.

With the constraints of an address defined, we must consider how data is organized in the contents of cache. When we are looking at this data, and not the source address to navigate to some location, the data is divided into 8 bits of actual data (allocated as per memory.vhd), while 6 other bits are used for tag matching purposes. This requires 14 bits and is unique to the data in a given cache line.

The 15th and 16th bits are used for the dirty and valid bit respectively. These 2 bits apply only to the entire cache block, and can only be observed at a block index without any offsets. This is important given the direct-mapped nature of our cache. Please note that we do not use the entire 32-bit address space, as we are only concerned with 15 bits. Observe Figure 1 for our implementation.

```
-- For storing CPU data address
temp_s_addr(3 downto 0) <= (others => '0');
temp_s_addr(14 downto 4) <= s_addr (14 downto 4);
tag_bits <= s_addr (14 downto 9);
block_index <= s_addr (8 downto 4);
block_index_int <= 4*to_integer(unsigned(s_addr (8 downto 4)));
word_index <= s_addr (3 downto 2);
word_index_int <= to_integer(unsigned(s_addr (3 downto 2)));

-- This blocks ensures that the block index isn't used when it's null and just being initialized
if block_index_int /= -2147483648 then
        retrieved_address <= cache_memory(block_index_int + word_index_int);
        in_cache_valid_bit <= cache_memory(block_index_int)(15);
        in_cache_dirty_bit <= cache_memory(block_index_int)(14);
        in_cache_tag_bits <= cache_memory(block_index_int)(13 downto 8);
end if;
```

Figure 1. Source Address Sub-division

States and Bit Combinations

Given all the combinations of valid, dirty, and tag bits, as well as whether we are in the read state or the write state, we have determined that there are 16 total possible combinations. Of those combinations, not all of them are useful. For instance, should a cache block be invalid, then it does not matter what combination it's in with respect to the other bits. The cache system will simply proceed to operate the same way each time, given that the data in cache can simply be overwritten. That is to say, our cache implementation will not bother with setting the dirty bit or checking the tag, since the data is invalid anyways. This, in turn, halves our combinations to 9 that are of interest, as opposed to 16 since 8 of them can be combined into a single outcome. Furthermore, given a valid address with matching tags, whether a cache block is labelled as dirty or not is irrelevant, and a read will still occur. Therefore, we can combine these outcomes. The same can be said for a write. In doing so, we can further reduce the number of outcomes that are observed to 7 combinations. Moreover, let's assume the cases where the tagging bits do not match and both the valid and dirty bits are enabled. This will require a sequence of writing back to main memory, replacing the cache block, and finally reading from or writing to the block as desired. Alternatively, if the dirty bit is not set, then we do not need to perform the described sequence. Instead, we replace the desired cache block into cache memory to be read or written to as desired.

Testing and Discussion

Given all of the defined states and possible outcomes of interest described above, we can begin to test the individual cases. Notably, we want to test if writing to cache works, if reading from cache works, if something from cache memory is actually written back to memory during an eviction, and finally if the appropriate cache blocks are retrieved. It is important to note here that not all features of the implementation are tested, as they may be rudimentary, judged to be irrelevant, or the success of our tests implies the success of other functional features.

To begin, we will first approach testing the rudimentary read and write capabilities of our implementation. To do so, we have written "1" to a given source address, denoted as s_writedata as per Figure 2.

```
report "Test #1";
s_read <= '0';
s_write <= '1';
s_writedata <= "00000000000000000000000000000001";
s_addr <= "11111111111111111111111111111111";
wait until rising_edge(s_waitrequest);
s_read <= '1';
s_write <= '0';
wait until rising_edge(s_waitrequest);
assert s_readdata(7 downto 0) = s_writedata(7 downto 0) report "DATA NOT IN CACHE (#1)";
s_read <= '0';
s_write <= '0';
wait for clk_period;
```

Figure 2. Test Case 1 for Read and Writes

This test conveniently enables us to test writes and reads at the same time. It is important to note that we must first write something to cache, before we can read it. The fact that we can read "1", as indicated by s_readdata in Figure 3, is indicative of our success with respect to those features.

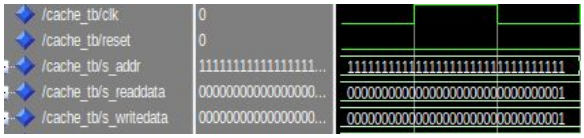

Figure 3. Test 1 Timing Diagram

Next we wanted to test the writeback feature of our system. To do this, we simply use a source address that matches the previous source address, in terms of block index, but whose tag differs. This ensures that we are writing different things to the same block in cache. It is important to note that the tests are done in the order that they appear to ensure the integrity of our tests.

```
report "Test #2";
s_read <= '1';
s_write <= '0';
s_addr <= "11111111111111111111100111111111";
wait until rising_edge(s_waitrequest);
assert s_readdata(7 downto 0) = m_writedata(7 downto 0) report "DATA NOT IN CACHE (#2)";
s_read <= '0';
s_write <= '0';
wait for clk_period;
```

Figure 4. Test Case 2 for Writeback

We can confirm that our writeback does, in fact, perform as expected. We can observe this to be true given the fact that m_writedata writes to memory the same value that was read by s_readdata previously, as per Figure 5. This demonstrates continuity of our writeback feature.

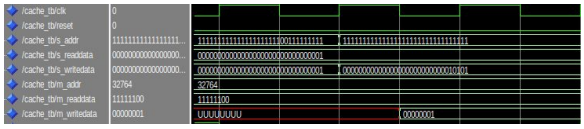

Figure 5. Test 2 Timing Diagram

To test the functionality of our dirty bits, we retrieved the cache line we had initially defined in our first test and stored into main memory. Upon doing so, we can observe that, once we attempt to write new data into this cache block, we are able to do so. In fact, this can be seen in Figure 6. Notice here, that s_readdata and s_writedata are identical. However, the m_writedata does not match. This is consistent with the fact that, despite writing new data onto the cache block and enabling the dirty bit, there is no need for a writeback since there is no eviction.

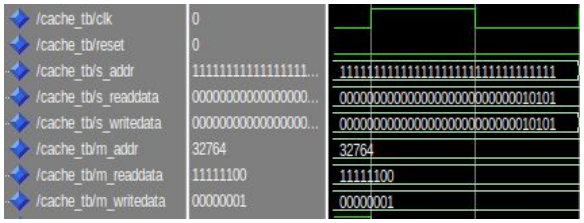

Figure 6. Test 3 Timing Diagram

The fourth test writes a different value into a previous address for further verification of the writeback and dirty bits. As we can see in Figure 7, the value “01010101” is written back to memory via m_writedata when the cache block is evicted.

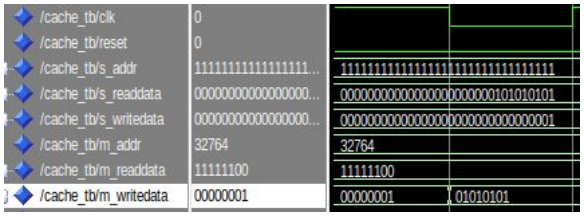

Figure 7. Test 4 Timing Diagram

The fifth test, which is the final test, as per Figure 8, checks to see what happens should a read miss occur without a write. That is to say, there is already data in memory which we would like to access.

```
report "Test #5";
s_read <= '1';
s_write <= '0';
s_addr <= "11111111111111111111100101111111";
wait until rising_edge(s_waitrequest);
s_read <= '1';
s_write <= '0';
s_addr <= "11111111111111111111100101111111";
wait until rising_edge(s_waitrequest);
assert s_readdata(7 downto 0) = m_readdata(7 downto 0) report "DATA NOT IN CACHE (#5)";
s_read <= '0';
s_write <= '0';
wait for clk_period;
```

Figure 8. Test Case 5 for Read Miss of RAM data

We can confirm that the data is indeed retrieved from memory despite the read miss. In fact, both m_readdata and s_readdata are matching, which indicates that the cache block was properly replaced by what was in main memory. Observe Figure 9. it is important to

note that the beginning portion of this test is repeated twice so as to ensure that all the wait requests and signal delays are properly accounted for. Doing so prevents undefined values, which would render our cache dysfunctional.

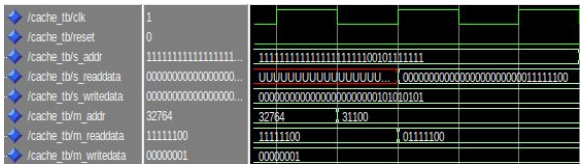

Figure 9.

Lastly, since the tests continue to run in a loop, the writeback behaviour, depending on the dirty and tag bits, will operate continuously. Although the later writes may be redundant, we have not explicitly defined our implementation to handle writing the same data into the same cache location. Therefore, the system operates as it should by overwriting the data. It should be stated that the testbench should be run and observed for a more descriptive representation of the success of our implementation. Although the pictures complement the explanations provided, they are not as faithful as scrolling through the actual generated timing diagram.

In conclusion

In conclusion, this project was an overall success. Moreover, the project helped solidify various concepts discussed in class regarding memory hierarchy, cache, reads, writes, as well as cache conflicts such as misses and hits. Furthermore, this project allowed for a comprehensive analysis of memory accesses and the way the cache system operates in computer architecture.

Appendix

```
when read =>

        -- Simply reads the data in cache
        s_readdata<= cache_memory(block_index_int + word_index_int);
        nextState <= idle;

        s_waitrequest <= '0';
```

Figure 1A. Read State Implementation

```
when write =>
        -- Writes the data to cache
        cache_memory(block_index_int + word_index_int) <= s_writedata;
        cache_memory(block_index_int)(13 downto 8) <= tag_bits;
        cache_memory(block_index_int)(14) <= '1';
        cache_memory(block_index_int)(15) <= '1';
        cache_memory(block_index_int)(31 downto 16) <= (others => '0');

        nextState <= idle;

        s_waitrequest <= '0';
```

Figure 2A. Write State Implementation

```
when writeBack =>
        -- Writes the data to main memory
        m_write <= '1';

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0)));
        m_writedata <= cache_memory(block_index_int)(7 downto 0);

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0))) + 4;
        m_writedata <= cache_memory(block_index_int + 1)(7 downto 0);

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0))) + 8;
        m_writedata <= cache_memory(block_index_int + 2)(7 downto 0);

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0))) + 12;
        m_writedata <= cache_memory(block_index_int + 3)(7 downto 0);

        m_write <= '0';

        nextState <= replace;
```

Figure 3A. Writeback State Implementation

```
when replace =>
        -- Replaces or adds a block from main memory to cache
        m_read <= '1';

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0)));
        cache_memory(block_index_int)(7 downto 0) <= m_readdata;
        cache_memory(block_index_int)(31 downto 8) <= (others => '0');

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0)))+4;
        cache_memory(block_index_int+1)(7 downto 0) <= m_readdata;
        cache_memory(block_index_int+1)(31 downto 8) <= (others => '0');

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0)))+8;
        cache_memory(block_index_int+2)(7 downto 0) <= m_readdata;
        cache_memory(block_index_int+2)(31 downto 8) <= (others => '0');

        m_addr <= to_integer(unsigned(temp_s_addr(14 downto 0)))+12;
        cache_memory(block_index_int+3)(7 downto 0) <= m_readdata;
        cache_memory(block_index_int+3)(31 downto 8) <= (others => '0');
        m_read <= '0';

        cache_memory(block_index_int)(13 downto 8) <= s_addr (14 downto 9);
        cache_memory(block_index_int)(14) <= '0';
        cache_memory(block_index_int)(15) <= '1';
        cache_memory(block_index_int)(31 downto 16) <= (others => '0');

        -- If Read go to read state
        if s_read = '1' then
                nextState <= read;
        -- If Write go to write state
        elsif s_write = '1' then
                nextState <= write;
        else
                nextState <= idle;
        end if;
```

Figure 4A. Replace State Implementation

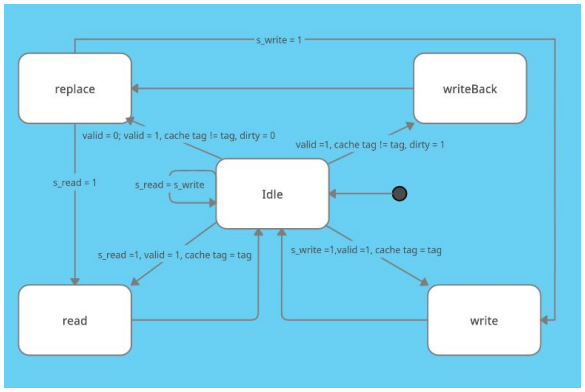

Figure 5A. Finite State Machine