ECSE 425 Project 2 - Direct-Mapped Cache

Haoran Du
260776911
McGill University
Montreal, Canada
haoran.du@mail.mcgill.ca

Carlo D'Angelo
260803454
McGill University
Montreal, Canada
carlo.dangelo@mail.mcgill.ca

***Abstract*—In this project, the team implemented and tested a basic, direct-mapped cache circuit with a write-back policy. The team designed the cache's structure (i.e., the storage for data, tags, and flags) and defined the cache's behaviour by drawing a state diagram and using the Altera Avalon interface as a foundation.**

I. Cache Implementation

The project description outlines specific characteristics that the cache implementation needs to have and describes what kind of main memory the cache will be interacting with. These details will directly affect the design of the cache structure and the decomposition of the memory addresses (i.e., what will be the tag, index and offset bits). These details can be seen below:

- 32-bit words
- 128-bit cache blocks (4 words)
- 4096-bit of cache data storage
- flag bits 'valid' and 'dirty' in the cache
- 32-bit memory addresses
- 32,768 bytes (32,768/4 = 8192 words) of total main memory

A. Memory Address Decomposition

Since the main memory only has a total of 32,768 = 2^{15} bytes, the cache only has to use the lower 15 bits of the memory addresses and ignore the rest. This reduction will result in fewer tag bits needing to be stored in the cache. We can also ignore the last two bits because the addresses are word-aligned. A more specific description of the address decomposition can be seen in Fig. 1. According to the cache specifications, the data storage is 4096-bit and each block is 128-bit (4 words). Therefore, the number of blocks in the cache must be 32:

$$\frac{4096}{128} = 32 \text{ blocks} \tag{1}$$

Furthermore, since we know the cache is a direct-mapped cache, it means that the block index must use 5-bits because every set contains only one block and because 32 = 2^5. As for the block offset, since there are 4 words per block and since each word is 4 bytes, it is clear that the word offset uses 2 bits and that the byte offset also uses 2 bits because 4 = 2^2. Therefore, the block offset uses 4 bits in total. The rest of the address bits are marked as tag bits. So, the tag uses 6 bits because of the following equation:

$$15 - 4 - 5 = 6 \text{ bits} \tag{2}$$

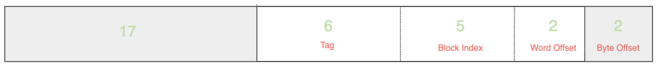

Fig. 1: 32-bit memory address

B. Cache Structure

Every entry in the cache uses 136 bits (see Fig. 2). 2 bits are used for the flags, 6 bits are used for the tag and 128 bits are used for the data itself. As a result, although the cache has a maximum of 4096 bits of data storage, it must actually store a total of 4352 bits:

$$136 * 32 = 4352 \text{ bits} \tag{3}$$

Therefore, there is an overhead of 4352 - 4096 = 256 bits in the cache.

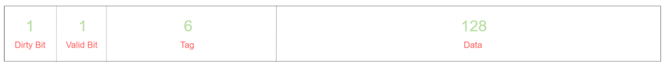

Fig. 2: Design of each entry in cache

C. State Diagram

Based on the study of the Altera Avalon interface, the team implemented the cache based on the state diagram shown in Fig. 3. There are 4 states in total, namely:

1) idle: the initial and default state

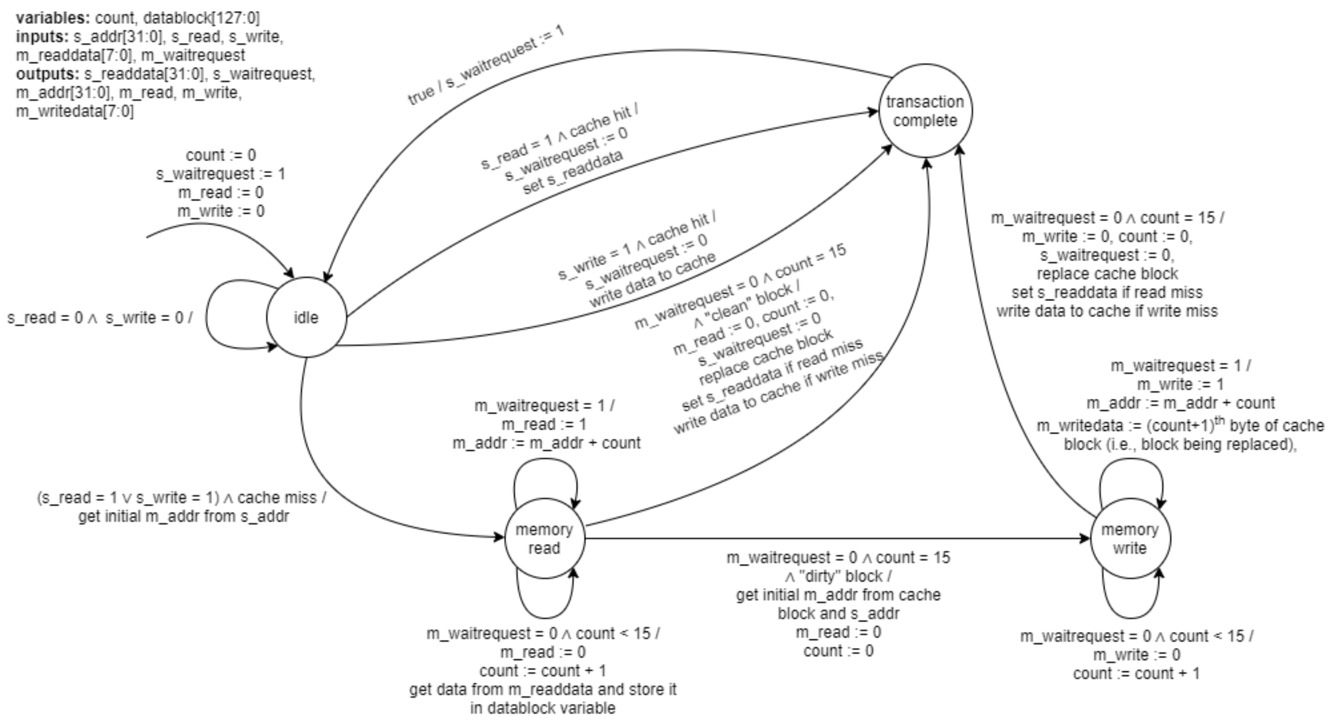
variables: count, datablock[127:0]
inputs: s_addr[31:0], s_read, s_write, m_readdata[7:0], m_waitrequest
outputs: s_readdata[31:0], s_waitrequest, m_addr[31:0], m_read, m_write, m_writedata[7:0]
true / s_waitrequest := 1
transaction complete
s_read = 1 ∧ cache hit / s_waitrequest := 0 set s_readdata
count := 0 s_waitrequest := 1 m_read := 0 m_write := 0
s_write = 1 ∧ cache hit / s_waitrequest := 0 write data to cache
s_read = 0 ∧ s_write = 0 /
idle
m_waitrequest = 0 ∧ count = 15 ∧ "clean" block / m_read := 0, count := 0, s_waitrequest := 0 replace cache block set s_readdata if read miss write data to cache if write miss
m_waitrequest = 0 ∧ count = 15 / m_write := 0, count := 0, s_waitrequest := 0, replace cache block set s_readdata if read miss write data to cache if write miss
m_waitrequest = 1 / m_read := 1 m_addr := m_addr + count
m_waitrequest = 1 / m_write := 1 m_addr := m_addr + count m_writedata := (count+1)th byte of cache block (i.e., block being replaced),
(s_read = 1 ∨ s_write = 1) ∧ cache miss / get initial m_addr from s_addr
memory read
memory write
m_waitrequest = 0 ∧ count = 15 ∧ "dirty" block / get initial m_addr from cache block and s_addr m_read := 0 count := 0
m_waitrequest = 0 ∧ count < 15 / m_read := 0 count := count + 1 get data from m_readdata and store it in datablock variable
m_waitrequest = 0 ∧ count < 15 / m_write := 0 count := count + 1

Fig. 3: State Diagram for Cache Implementation

2) memory read: if there is a cache miss, read the correct block from main memory and put it in the cache
3) memory write: if there is a cache miss and the current data in the cache is dirty, write the cache block to memory before replacing it.
4) transaction complete: handles the reassertion of the *waitrequest* signal

In this implementation, one important point to take note of is what happens when there is a cache miss. Since the addresses that are fed into the main memory (i.e., *m_addr[31:0]*) are byte aligned, we must access the main memory 16 times for both reads and writes respectively because the block size is 128 bits.

$$128/8 = 16 \text{ memory accesses} \tag{4}$$

II. Testbench

Since a cache block can be valid/invalid, dirty/not dirty, and an access to a block can be read/write, tag equal/tag not equal, there are theoretically $2^4 = 16$ different cases that the FSM must handle.

A. Impossible Cases

The following 4 cases are determined impossible:

1) invalid + dirty + read + tag not equal
2) invalid + dirty + read + tag equal
3) invalid + dirty + write + tag not equal
4) invalid + dirty + write + tag equal

These test case are impossible because it is impossible for a block to be invalid and dirty at the same time.

B. Example Test Case

In the *cache_tb.vhd* file, a complete testbench covering all possible combinations of valid/invalid, dirty/not dirty, read/write, tag equal/tag not equal for a cache access can be found. Listing 1 shows one of these test cases, namely the cache access that is valid, not dirty, read, and tag not equal.

```
-- Test case 3: Cache Miss (Read)
report "Test case 3, valid + not dirty +
  read + tag not equal";
       s_addr <= x"00002010";
       s_read <= '1';
       wait until rising_edge(s_waitrequest);
assert s_readdata = x"13121110" report "read
   data is wrong" severity error;
s_read <= '0';
       wait for clk_period;
-- The address tells us that the tag of the
  desired block is "010000" and that its
  index in the cache is 1.
-- The current cache block at index 1 is
  valid.
```

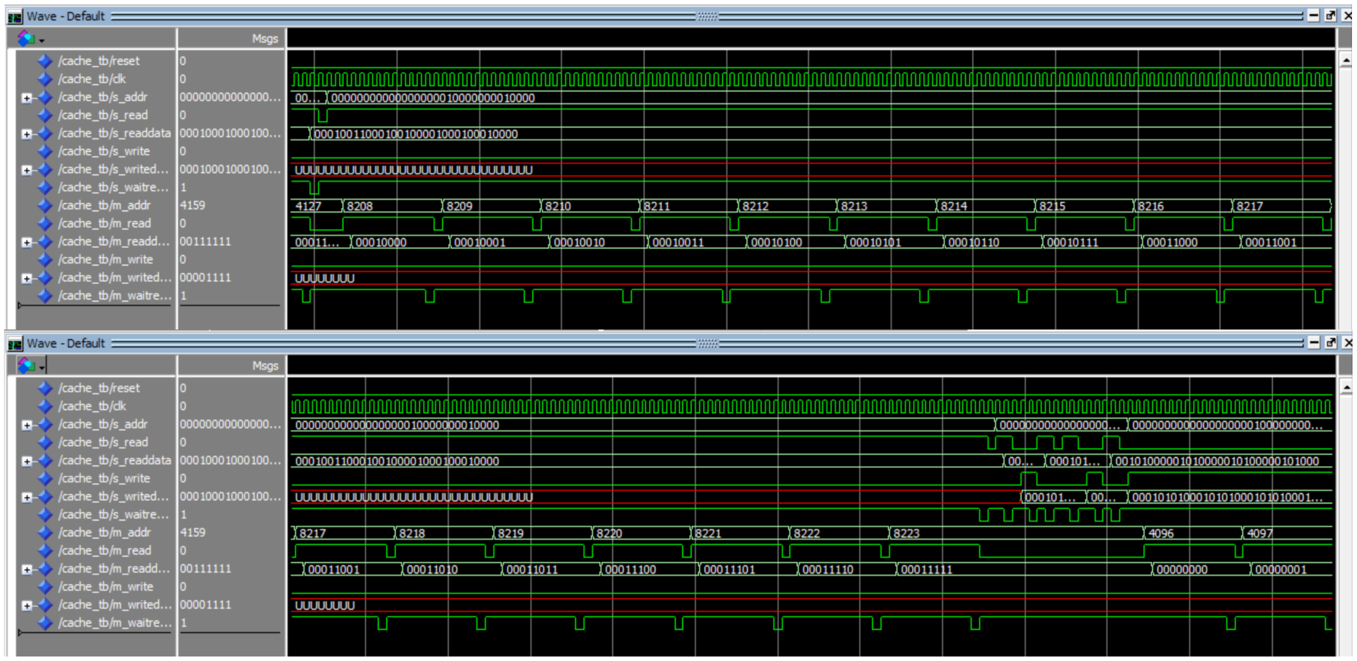
```
Wave - Default
                          Msgs
/cache_tb/reset           0
/cache_tb/clk             0
/cache_tb/s_addr          00000000000000...
/cache_tb/s_read          0
/cache_tb/s_readdata      0001000100010 0...
/cache_tb/s_write         0
/cache_tb/s_writed...     0001000100010 0...
/cache_tb/s_waitre...     1
/cache_tb/m_addr          4159
/cache_tb/m_read          0
/cache_tb/m_readd...      00111111
/cache_tb/m_write         0
/cache_tb/m_writed...     00001111
/cache_tb/m_waitre...     1
```
```
Wave - Default
                          Msgs
/cache_tb/reset           0
/cache_tb/clk             0
/cache_tb/s_addr          00000000000000...
/cache_tb/s_read          0
/cache_tb/s_readdata      0001000100010 0...
/cache_tb/s_write         0
/cache_tb/s_writed...     0001000100010 0...
/cache_tb/s_waitre...     1
/cache_tb/m_addr          4159
/cache_tb/m_read          0
/cache_tb/m_readd...      00111111
/cache_tb/m_write         0
/cache_tb/m_writed...     00001111
/cache_tb/m_waitre...     1
```

Fig. 4: Wave for the example test case from ModelSim

```
  -- The current cache block at index 1 is not
     dirty.
  -- The current cache block at index 1 will
    have tag not equal.
  -- The cache will access main memory and
    replace this block.
  -- The new cache block at index 1 is valid.
  -- The new cache block at index 1 is not
    dirty.
  -- The new cache block at index 1 will have
    a tag of "010000".
```
Listing 1: Test case covering the valid, not dirty, read, tag not equal cache access

The code comments in Listing 1 describe how the test memory access relates to this specific test case. Fig. 4 displays the test results for the example test case, which were obtained by running a simulation in ModelSim. These results reflect the test case accurately because the cache reads from main memory 16 times and writes nothing to memory, which is supposed to happen after a read miss with a clean block in this implementation.

III. Conclusion

By running the *cache.tcl* file and confirming all the test results make sense like the test result of the example test case shown above, it can be concluded that the cache functions correctly and follows all the design requirements.