ECSE 425 Project 2 Report

Junjian Chen 260909101 *Shichang Zhang 260890019*

Abstract

In this project, our group designed and tested a basic direct-mapped cache circuit, which is able to handle various cases of memory access such as read and write. We also perform tests on the cache with cases. Finally, we found that our implementations behaved correctly in all possible cases.

Introdcution

The cache we implemented is a 4096-bit data storage direct-mapped cache with write-back policy. It links with a 32768 bytes main memory. The design is based on Altera Avalon interface, which defines the timing with which data should be transferred between the master device and the slave device. We designed and implemented the cache by using a finite-state machine (FSM) to control the flow of operations of the cache, which will be explained in detail in the next part.

Method

A. Bit calculation

The MIPS processor uses a 32-bit address, but in this project we only use the lower 15 bits. So we only use 0-15 bits and keep 16-31 bits '0'.

Within the lower 15 bits, 0-1 bits are byte offset, 2-3 bits are word offset, 4-8 bits are block index and the rest 9-14 bits are tag. This is calculated by:

$$Blocks\ Number = \frac{4096-bit\ data\ storage}{128-bit/blocks} = 32$$

$$Words\ per\ Block = \frac{128-bits\ block}{32-bit/word} = 4$$

$$Bytes\ per\ word = \frac{32-bit\ word}{8-bit/byte} = 4$$

So 5 bits are required for block index, 2 bits for word offset and 2 bits for byte offset. The rest of the bits are for tag, so it is 15-5-2-2=6 bits.

B. Cache structure

Our cache consists of 128 sets and each set has 40 bits which contains the following information:

- 0-31 bits: data of a 32-bit word
- 32-37 bits: tag bits
- 38 bit: dirty bit
- 39 bit: valid bit

C. State diagram

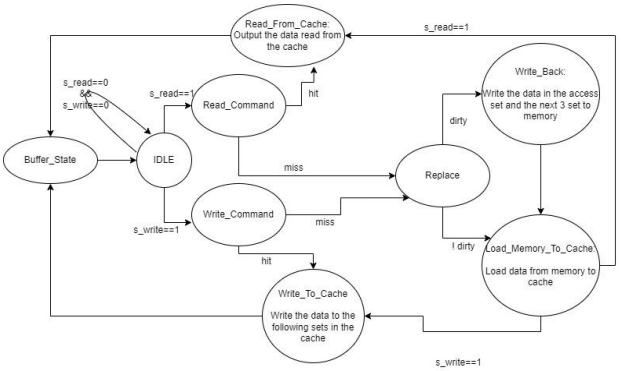

Figure 1: The State Diagram

The state diagram of our implementation is shown in Figure 1. There are nine states in our finite-state machine:

1. **IDLE**: Do nothing.
2. **Read_Command**: The system will enter this state from *IDLE* when the request is a read.
3. **Write_Command**:The system will enter this state from *IDLE* when the request is a write.
4. **Replace**: When the access is a read miss/write miss. It will go to *Write_Back* if the data is dirty.
5. **Write_Back**: In this state, the system writes the data in the access block to memory. Because we have one word per set and four words per block, we need to write four times. After writing, it enters *Load_Memory_To_Cache*.
6. **Load_Memory_To_Cache**: In this state, the system loads data from memory to cache. Similar to *Write_Back*, we read four times.
7. **Write_To_Cache**: In this state, the system will directly write the data to the cache.
8. **Read_From_Cache**: In this state, the system will directly read the data from the cache.
9. **Buffer_State**: Set waitrequest to '1' and go back to *IDLE* state. Ensuring waitrequest in the *IDLE* state is '1'.

D. Explanation of implementation

The cache is initialized by setting all bits of to 0. When the state machine starts, it enters the *IDLE* state and waits for a read or write command. If s_read or s_write is 1, it will move to *Read_Command* or *Write_Command*.

In *Read_Command* or *Write_Command*, we identify whether the access is a hit or a miss by checking whether the valid bit is '1' and tags are equal. If it is a hit, according to whether s_read='1' or s_write='1', the system goes to the *Read_From_Cache* or *Write_To_Cache* to directly read or write the data from or to the request block in the cache. If it is a miss, FSM moves to *Replace* state.

In *Replace* state, if the access block in cache is not dirty, we enter the *Load_Memory_To_Cache* state to load the desire block from main memory to cache. If the block is dirty, we go to *Wirte_Back* state to write dirty data back to the main memory and then go to the state *Load_Memory_To_Cache*.

After replacing the block from memory, for the read command, we go to *Read_From_Cache* state and output data from the cache. For the write command, we go to *Write_To_Cache* state and directly write data into the desire word in the cache. After operation, both of them go to *Buffer_State* to make s_waitrequest be low for 1 clock cycle then go to *IDLE*.

E. Analysis of impossible cases

There are four key factors affecting the operations of caches and memory:

- Valid/Invalid
- Dirty/Clean
- Read/Write
- TagEqual/TagNotEqual

So there are $2^4 = 16$ different cases:

1. Read, Valid, Clean, TagEqual
2. Read, Valid, Clean, TagNotEqual
3. Read, Valid, Dirty, TagEqual
4. Read, Valid, Dirty, TagNotEqual
5. Read, Invalid, Clean, TagEqual
6. Read, Invalid, Clean, TagNotEqual
7. Read, Invalid, Dirty, TagEqual
8. Read, Invalid, Dirty, TagNotEqual
9. Write, Valid, Clean, TagEqual
10. Write, Valid, Clean, TagNotEqual
11. Write, Valid, Dirty, TagEqual
12. Write, Valid, Dirty, TagNotEqual
13. Write, Invalid, Clean, TagEqual
14. Write, Invalid, Clean, TagNotEqual
15. Write, Invalid, Dirty, TagEqual
16. Write, Invalid, Dirty, TagNotEqual

Cases 7, 8, 15, 16 are impossible.

Since we are not able to set the valid bit of the cache set to 0, the cases that we encounter a invalid bit is only at the first read or write to the cache. In this case, due to the initialization, all sets in the cache have '000000' tag bits, '0' dirty bit, '0' valid bit. So we cannot have the case that the data is both invalid and dirty.

Result

The test cases number mentioned below refer to the test cases shown above. When we refer to a default cache set, it means the set has not been changed after the cache initialization. We modify the initialization of main memory in memory.vhd with a mod operation. The memory initialization code is ram_block(i) <= std_logic_vector(to_unsigned(i mod 256,8));.

A. Testing of case 5

```
--5. read, !valid, clean, tagEqual
s_addr <= X"00000051";--at 0 word and 00101=5block,tag 000000
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"53525150" report "read unsuccessful case 5" severity error;
s_read<='0';
```

Figure 2: Test code of case 5

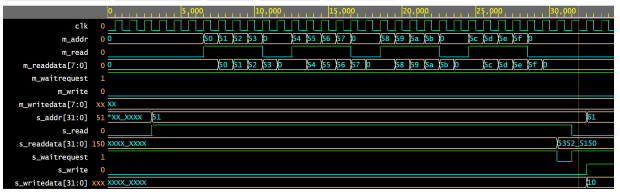

Figure 3: Test result of case 5

Test case 5 is the first test after the cache initialization. As shown in Figure 2, the access tag is 0x000000 so equal to the default tag. The data in the cache set is invalid and clean in default, so it takes 16 memory read operations to replace 4 words that belong to the block. Finally, we read from the cache and s_readdata is 0x53525150, which meets expectations.

B. Testing of cases 13 and 3

```
--13. write, !valid, clean, tagEqual
s_addr <= X"00000061";--at 0 word and 00110=6block,tag 000000
s_writedata <= X"00000010";
s_write <= '1';
wait until rising_edge(s_waitrequest);
s_write <= '0';
```

Figure 4: Test code of case 13

```
--3. read, valid, !clean, tagEqual
s_addr <= X"00000061";--at 0 word and 00110=6block,tag 000000
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"00000010" report "write unsuccessful case 4 or read unsuccessful case 3" severity error;
s_read <= '0';
```

Figure 5: Test code of case 3

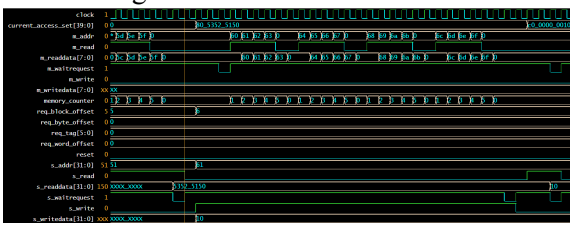

Figure 6: Test result of cases 13 and 3

For case 13, as shown in Figure 4, it is a write to an invalid cache set and tags equal, so it is a write miss. As the set is clean, it will only load data from memory to cache. After loading, we write data 0x10 to the first word in the block and set the dirty bit of the block to '1'. After case 13, we use case 3 to validate the result of the write. As shown in Figure 5, we read from the address 0x00000061, which is the same address in case 13. So the operation will be a read hit and unclean access. Since we have written 0x10 to this cache set, the expected s_readdata should be 0x10. As shown in Figure 6, our test result is correct.

C. Testing of cases 6 and 1

```
-- 6. read, !valid, clean, !tagEqual
s_addr <= X"00002091";--at 0 word and 01001=9block,tag 010000
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"93929190" report "read unsuccessful case 6" severity error;
s_read<='0';
```

Figure 7: Test code of case 6

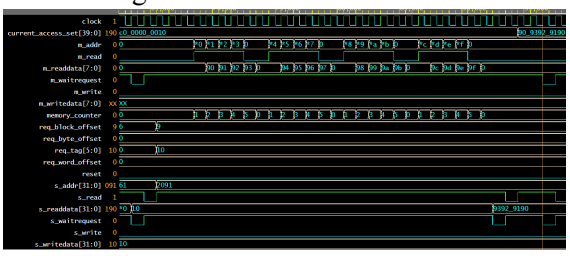

Figure 8: Test result of case 6

```
-- 1. read, valid, clean, tagEqual
s_addr <= X"00002091";--at 0 word and 01001=9block,tag 010000
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"93929190" report "read unsuccessful case 1" severity error;
s_read<='0';
```

Figure 9: Test code of case 1

As shown in Figure 7, we read from the first word in block 9, which is a default cache set, so it is invalid and tags are unequal. Since it is a read miss, it will load data from memory and read it. The expected output is 0x93929190. As shown in Figure 8, our test result is correct.

After that, for case 1 shown in Figure 9, we read the first word of block 9 again. So it will be a hit the data is valid and clean. As shown in Figure 8, the s_readdata signal does not change since we obtain 0x93929190 again. So our test of case 1 succeeds.

D. Testing of case 2

```
--2. read, valid, clean, !tagEqual
s_addr <= X"00002891";--at 0 word and 01001=9block,tag 010100
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"93929190" report "read unsuccessful case 2" severity error;
s_read <= '0';
```

Figure 10: Test code of case 2

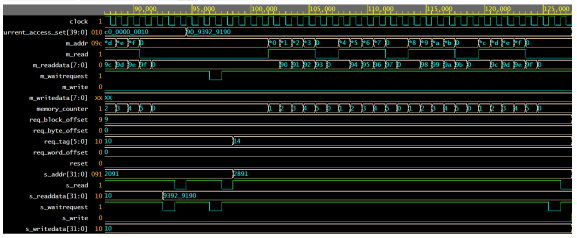

Figure 11: Test result of case 2

For case 2 in Figure 10, we read from block 9 again in case 1, but tags are different. So it is a read miss and will load data from memory. Since the memory is initialized using a module method, the value of data with the same block but different tags in the main memory are the same. As is shown in m_readdata in Figure 11, we get data from a different address, but coincidently the same value. Hence the s_readdata should be 0x93929190 again. As shown in Figure 1, our test result is correct.

E. Testing of cases 14, 9 and 4

```
-- 14. write, !valid, clean, !tagEqual
s_addr <= X"00001111";--at 0 word and 10001=17block,tag 001000
s_writedata <= X"00000011";
s_write <= '1';
wait until rising_edge(s_waitrequest);
s_write <= '0';

wait for clk_period;

-- 3. read, valid, !clean, tagEqual
s_addr <= X"00001111";--at 0 word and 10001=17block,tag 001000
s_write <= '0';
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"00000011" report "write unsuccessful case 14" severity error;
s_read <= '0';
```

Figure 12: Test code of case 14

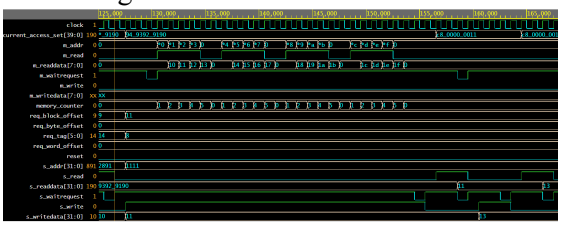

Figure 13: Test result of case 14

As shown in Figure 12, we write to block 17 which is a default set (clean, valid, tag unequal). It is a write miss, so we load data

from memory and write 0x11 to the first word. Again, we use case 3 to check the result of this write. We obtain 0x11 as shown in Figure 13, which is what we just wrote, hence correct.

```
-- 9. write, valid, !clean, tagEqual
s_addr <= X"00001111";--at 0 word and 10001-17block,tag 001000
s_writedata <= X"00000013";
s_read <= '0';
s_write <= '1';
wait until rising_edge(s_waitrequest);
s_write <= '0';

wait for clk_period;

-- 3. read, valid, !clean, tagEqual
s_addr <= X"00001111";--at 0 word and 10001-17block,tag 001000
s_write <= '0';
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"00000013" report "write unsuccessful case 9" severity error;
s_read <= '0';
```

Figure 14: Test code of case 9

For case 9 in Figure 14, we write 0x13 to the same address in test case 14. It is a write hit, we directly write the data to the cache. Again, we check by case 3 and obtain 0x13, which is what we just wrote. The result is correct.

```
-- 4. read, valid, !clean, !tagEqual
s_addr <= X"00001311";--at 0 word and 10001=17block,tag 001001
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"13121110" report "write unsuccessful case 9 or read unsuccessful case 4" severity error;
s_read <='0';
```

Figure 15: Test code of case 4

For case 4 in Figure 15, we read the first word of block 17 again but with different tags. Now the data is dirty. Hence, as shown in Figure 16, it takes 32 cycles each to write back and load data from memory to cache. Finally, the output data is 0x13121110, which is the correct result.

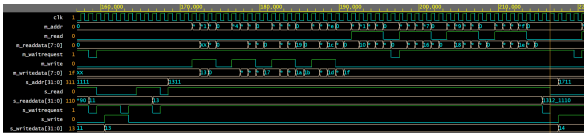

Figure 16: Test result of case 4

F. Testing of cases 10 and 11

```
-- 10. write, valid, clean, !tagEqual
s_addr <= X"00001711";--at 0 word and 10001=17block,tag 001011
s_writedata <= X"00000014";
s_write <= '1';
wait until rising_edge(s_waitrequest);
s_write <= '0';

wait for clk_period;

--3. read, valid, !clean, tagEqual
s_addr <= X"00001711";--at 0 word and 10001=17block,tag 001010
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"00000014" report "write unsuccessful case 10" severity error;
s_read <= '0';
```

Figure 17: Test code of case 10

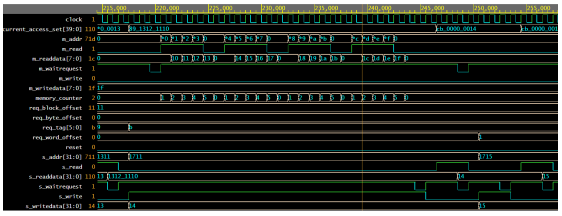

Figure 18: Test result of case 10

As shown in Figure 17, case 10 writes 0x14 to block 17 but with unequal tags. So it is a write miss and the data is valid and clean. Again, case 3 is used to examine the write. The output data is 0x14 and it matches the data we wrote.

```
--11. write, valid, !clean, tagEqual
s_addr <= X"00001715";--at 0 word and 10001=17block,tag 001011, word 1
s_writedata <= X"00000015";
s_write <= '1';
wait until rising_edge(s_waitrequest);
s_write <= '0';

wait for clk_period;

--3. read, valid, !clean, tagEqual
s_addr <= X"00001715";--at 0 word and 10001=17block,tag 001010
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"00000015" report "write unsuccessful case 10" severity error;
s_read <= '0';
```

Figure 19: Test code of case 11

Figure 19 shows case 11 writes the second word of block 17. The tags are equal, so it is a write hit and directly writes 0x15 to the cache. As shown in Figure 18, case 3 validates the data just written is 0x15. Hence correct.

G. Testing of case 12

```
-- 12. write, valid, !clean, !tagEqual
s_addr <= X"00001511";--at 0 word and 10001=17block,tag 001010
s_writedata <= X"00000016";
s_write <= '1';
wait until rising_edge(s_waitrequest);
s_write <= '0';

wait for clk_period;

--3. read, valid, !clean, tagEqual
s_addr <= X"00001511";--at 0 word and 10001=17block,tag 001010
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"00000016" report "write unsuccessful case 10" severity error;
s_read <= '0';

wait for clk_period;

--3. read, valid, !clean, tagEqual
s_addr <= X"00001515";--at 0 word and 10001=17block,tag 001010, word 1
s_read <= '1';
wait until rising_edge(s_waitrequest);
assert s_readdata = x"17161514" report "write unsuccessful case 10" severity error;
s_read <= '0';
```

Figure 20: Test code of case 12

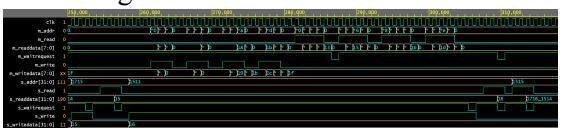

Figure 21: Test result of case 12

As shown in Figure 20, case 12 writes 0x16 to the first word of block 17. Case 12 is similar to case 4, but it is a write instead of a read. Both of them trigger write back and load data from memory to cache. Again, case 3 is used to check this write. From Figure 21, the data in the first word is 0x16, which is what we just wrote. The reading of the second word is not 0x15 but 0x17161514, since we load the whole block from memory. So the result is correct.

Conclusion

In conclusion, we implement the cache with a bug-free FSM. The cache functions correctly in all possible test cases and achieves all design requirements.