浙江大學

本科实验报告

课程名称：	计算机体系结构
姓　　名：	
学　　院：	计算机科学与技术学院
系：	计算机科学与技术系
专　　业：	计算机科学与技术
学　　号：	
指导教师：	卜凯

2022 年　　11 月　　23 日

浙江大学实验报告

课程名称：计算机体系结构	实验类型：综合
实验项目名称：Lab3：Cache Design	
学生姓名：	专业：计算机科学与技术　　学号：
同组学生姓名：	指导老师：卜凯
实验地点：曹西 301	实验日期：2022 年 11 月 23 日

1 Tasks and requirements

1.1 Tasks

The main tasks of Lab-3 are:

1. Design the cache line.
2. Verify the cache line.
3. Observe the waveform of simulation.

As this lab is only finished and verified on Vivado, we do not need on-board experiment results.

1.2 Requirements

This experiment would be based on Vivado 2020.2.

We are given a Verilog project simulating a 2-way set associative cache. The project is only consisted of a simulation module and a cache module we need to finish:

cache.v, to implement the cache operations by some cache control signals on a 2-way set associative LRU replaced write-back cache.

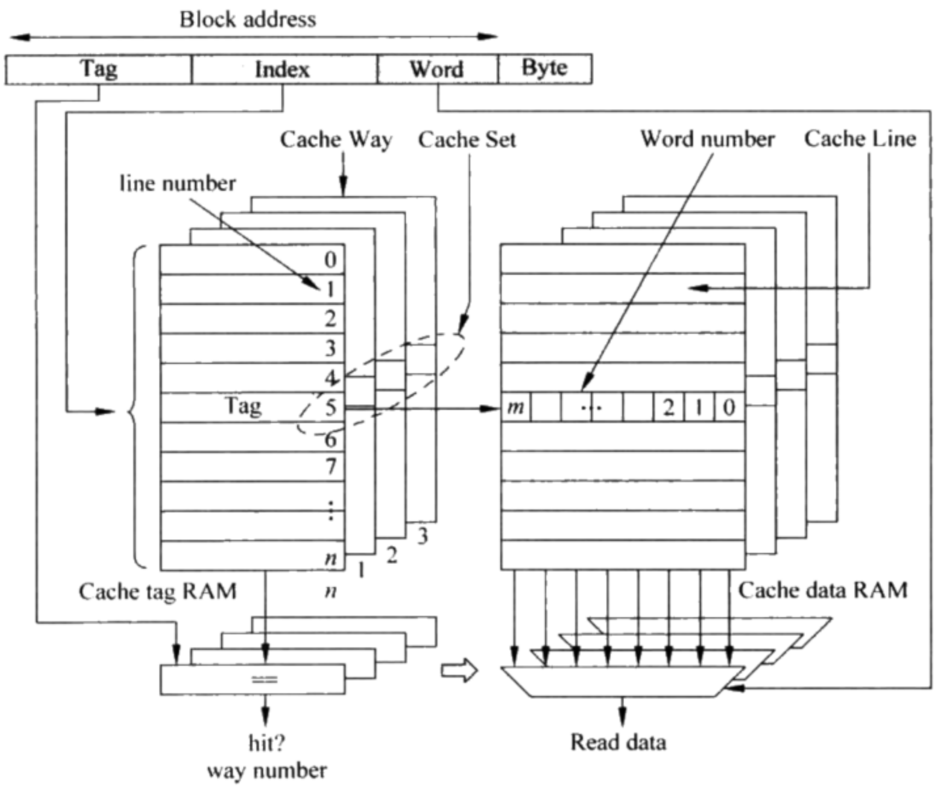
When this work is finished, we shall verify the simulation results and give out some analysis.

2 Contents and principles

All of the following principles are based on 2-way set associative LRU replaced write-back cache.

2.1 Cache

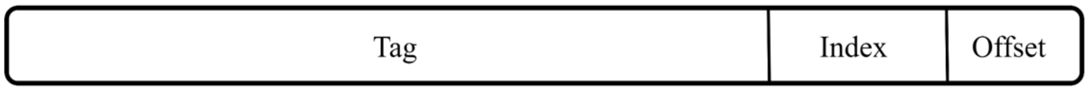
Cache is an important component used in computer logic design, useful for accelerating data access by CPU instructions. A cache stores data and some other signals concerning data validation and status, and may work as the following graph shows.



2.2 Address Decoding

In order to use the cache to optimize data read and write, we need to map the program-used memory address into cache blocks.

In cache we divide the original address into 3 fields: **tag**, **index**, and **offset**. Their placement is shown in the next graph.



Offset bits are used to locate the required data in cache lines, usually one specific bit. They are the last bits in the original address.

Index bits are used to determine where the data should be stored in cache. They are usually got by a **mod** operation, so they are the middle bits of an address.

The remining are **tag** bits. Tag bits are used to check whether the data in a cache block matches the required address. They are stored in a cache structure.

To locate an original address, we first use its **index** bits to find its corresponding block, and then use **tag** bits to check the data of that address is stored in cache (a **hit**) or not (a **miss**). If miss, then we do some other operations to load the data from main memory or another cache (not implemented in this lab). At last, the **offset** bits are used to find the specific data segment.

2.3 Cache Operations

In cache design, aside from address decoding problems, we shall consider how to deal with **hit**, **clean miss** and **dirty miss** conditions of cache.

If the cache is to experience a **hit** event, then it need not much thing to do. The cache would just send out the data required.

If the cache is to experience a **clean miss** event, then it needs to first load some data from memory. We just give out the demanded address and let cache controller do the job remaining.

If the cache is to experience a **dirty miss** event, then it needs to write the data in cache back into memory (write-back) by given out the demanded address to cache controller. And the following operations will be similar to the **clean miss** process: load again and send the data.

3 Steps and data records

3.1 Completed Verilog source files

3.1.1 cache.v

For clarity, we will omit some pre-set code segments, and only analysis the code blocks that we need to fill.

As this is a 2-way associative cache, we first read out the two cache lines according to **index** bits and assign these retrieved data to some wires that we will use afterwards. Note that **recent** field has only one bit; Because that the replacement would concern only one of the two cache lines, one bit is enough.

```
assign addr_tag = addr[31:9];
assign addr_index = addr[8:4];
assign addr_element1 = {addr_index, 1'b0};
assign addr_element2 = {addr_index, 1'b1};
assign addr_word1 = {addr_element1,
    addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-1
    :WORD_BYTES_WIDTH]};
assign addr_word2 = {addr_element2,
    addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-1
    :WORD_BYTES_WIDTH]};

assign word1 = inner_data[addr_word1];
assign word2 = inner_data[addr_word2];
assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];
assign byte1 = addr[1] ?
    addr[0] ? word1[31:24] : word1[23:16] :
   addr[0] ? word1[15:8] :    word1[7:0];
assign byte2 = addr[1] ?
   addr[0] ? word2[31:24] : word2[23:16] :
   addr[0] ? word2[15:8] :    word2[7:0];

assign recent1 = inner_recent[addr_element1];
assign recent2 = inner_recent[addr_element2];
assign valid1 = inner_valid[addr_element1];
assign valid2 = inner_valid[addr_element2];
assign dirty1 = inner_dirty[addr_element1];
assign dirty2 = inner_dirty[addr_element2];
assign tag1 = inner_tag[addr_element1];
assign tag2 = inner_tag[addr_element2];
```

The hit signal is assigned by comparing **tag** bits and the stored tag.

```
assign hit1 = valid1 & (tag1 == addr_tag);
assign hit2 = valid2 & (tag2 == addr_tag);
```

In the always@ block, we first get information about the least recently used cache line; These data may be used to write the data back to memory.

```
always @ (posedge clk) begin
    valid <= recent1 ? valid2 : valid1;
    dirty <= recent1 ? dirty2 : dirty1;
    tag <= recent1 ? tag2 : tag1;
    hit <= hit1 | hit2;
```

According to the control signals that the cache controller gives, we should do different operations in cache. First, if a **load** signal is valid, then we will give out the data stored if **hit**.

```
  if (load) begin
  if (hit1) begin
    dout <=
    u_b_h_w[1] ? word1 :
    u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word1[15]}}, half_word1} :
    {u_b_h_w[2] ? 24'b0 : {24{byte1[7]}}, byte1};
    inner_recent[addr_element1] <= 1'b1;
    inner_recent[addr_element2] <= 1'b0;
  end
  else if (hit2) begin
    dout <=
    u_b_h_w[1] ? word2 :
    u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word2[15]}}, half_word2} :
        {u_b_h_w[2] ? 24'b0 : {24{byte2[7]}}, byte2};
        inner_recent[addr_element1] <= 1'b0;
        inner_recent[addr_element2] <= 1'b1;
    end
end
```

If not a **load**, then we output the word that the address located to in the least recently used cache line.

```
else dout <= inner_data[ recent1 ? addr_word2 : addr_word1 ];
```

If an edit, then we adjust the demanded cache line by the input **din**.

```
if (edit) begin
    if (hit1) begin
        inner_data[addr_word1] <=
        u_b_h_w[1] ?
        din
        :
        u_b_h_w[0] ?
        addr[1] ?
        {din[15:0], word1[15:0]}
        :
        {word1[31:16], din[15:0]}
        :
        addr[1] ?
        addr[0] ?
        {din[7:0], word1[23:0]}
        :
        {word1[31:24], din[7:0], word1[15:0]}
        :
        addr[0] ?
        {word1[31:16], din[7:0], word1[7:0]}
        :
```

```
    {word1[31:8], din[7:0]}
    ;
    inner_dirty[addr_element1] <= 1'b1;
    inner_recent[addr_element1] <= 1'b1;
    inner_recent[addr_element2] <= 1'b0;
  end
  else if (hit2) begin
    inner_data[addr_word2] <=
    u_b_h_w[1] ?
    din
    :
    u_b_h_w[0] ?
    addr[1] ?
    {din[15:0], word2[15:0]}
    :
    {word2[31:16], din[15:0]}
    :
    addr[1] ?
    addr[0] ?
    {din[7:0], word2[23:0]}
    :
    {word2[31:24], din[7:0], word2[15:0]}
    :
    addr[0] ?
    {word2[31:16], din[7:0], word2[7:0]}
    :
    {word2[31:8], din[7:0]}
    ;
    inner_dirty[addr_element2] <= 1'b1;
    inner_recent[addr_element1] <= 1'b0;
```

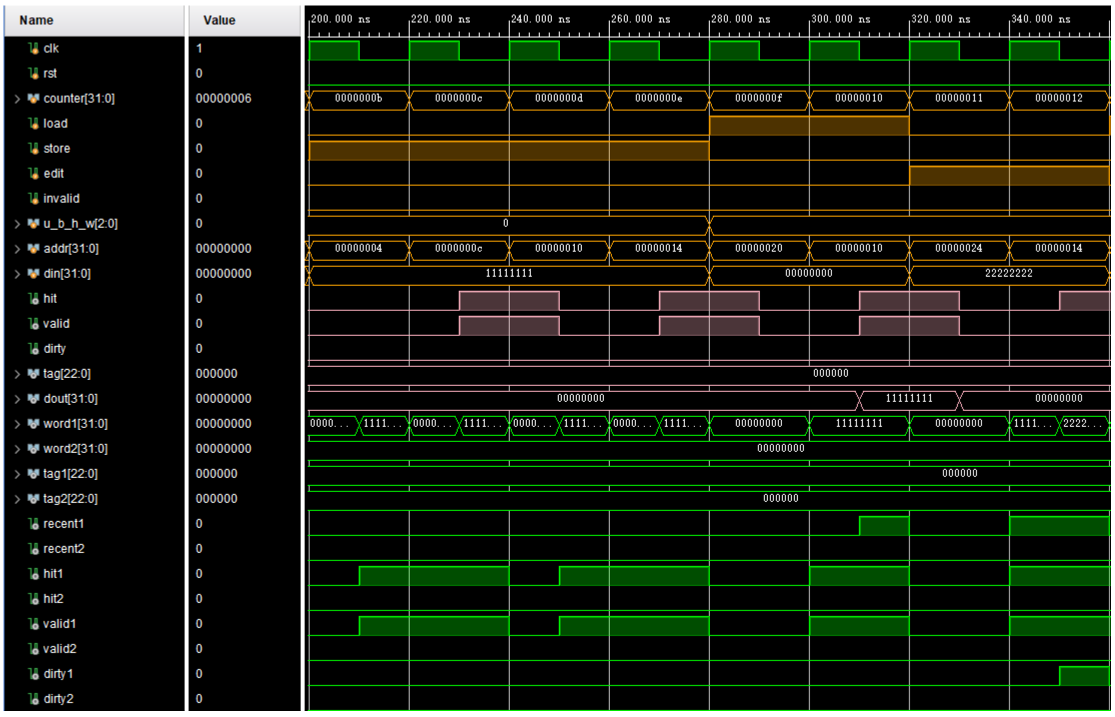
```
        inner_recent[addr_element2] <= 1'b1;
        end
    end
```

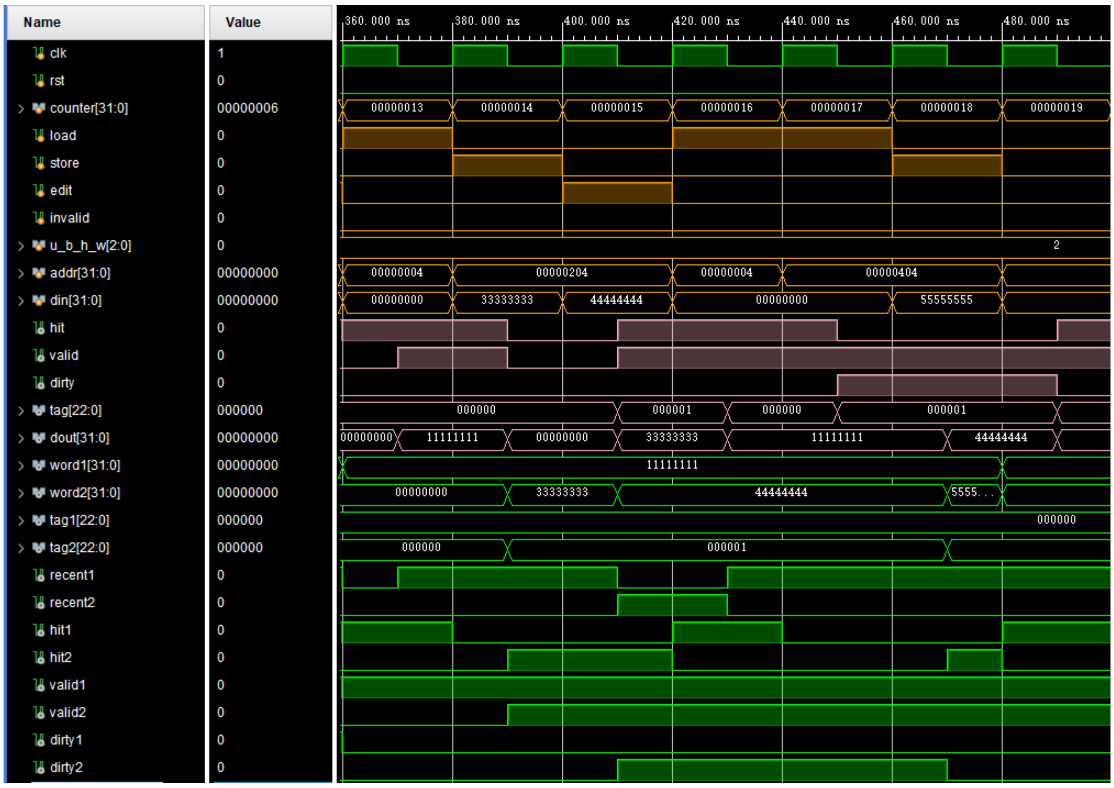
If **store** signal is valid, then we clear the least recently used cache line in that block, and modify the data by **din**.

```
if (store) begin
    if (recent1) begin
        inner_data[addr_word2] <= din;
        inner_valid[addr_element2] <= 1'b1;
        inner_dirty[addr_element2] <= 1'b0;
        inner_tag[addr_element2] <= addr_tag;
    end else begin
        inner_data[addr_word1] <= din;
        inner_valid[addr_element1] <= 1'b1;
        inner_dirty[addr_element1] <= 1'b0;
        inner_tag[addr_element1] <= addr_tag;
    end
end
```

3.2 Implementation results

Here we record all our behavioral simulation results. Most will not be used for analysis.

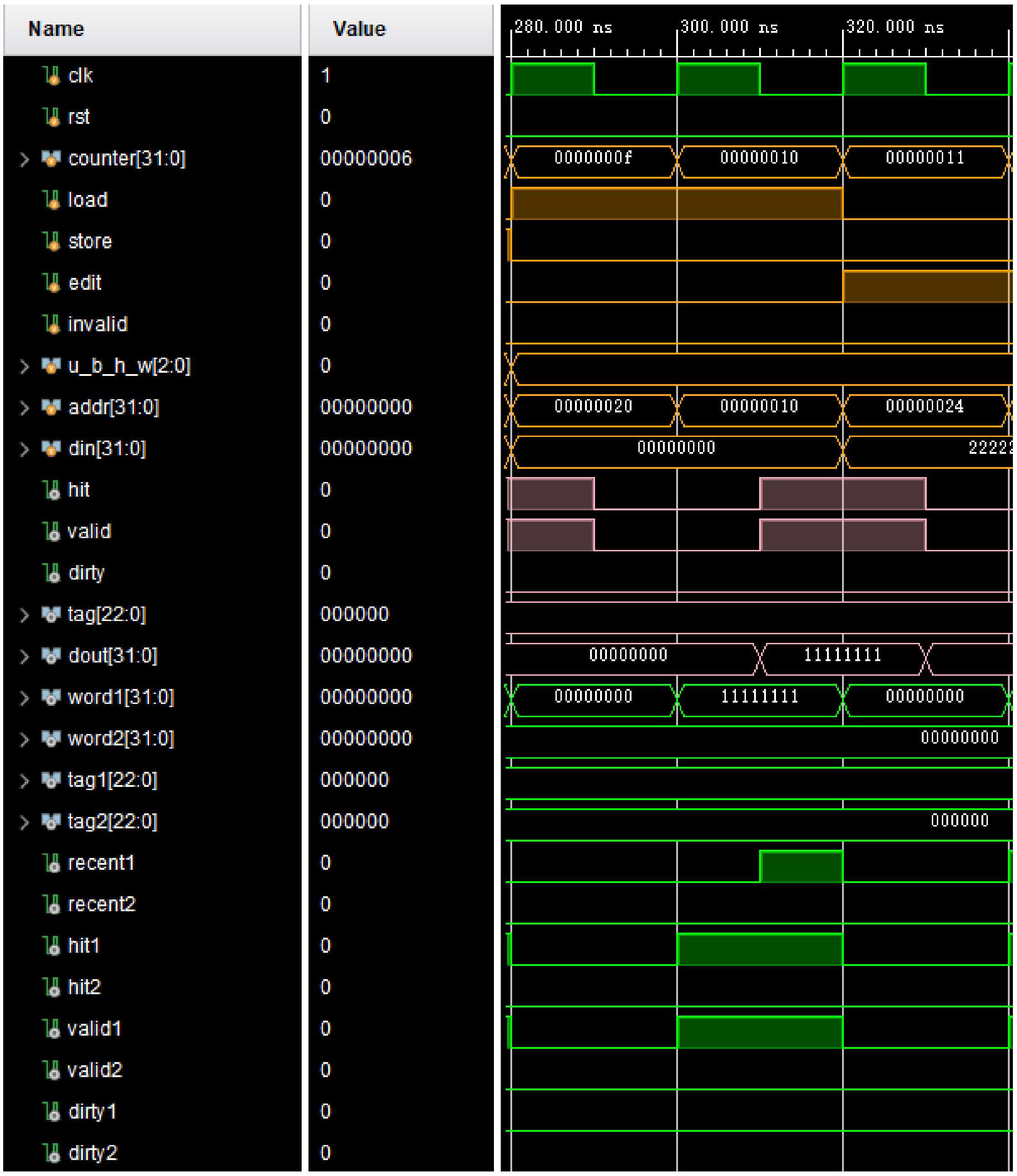




4 Analysis of the results

Here we only consider some typical results.

4.1 Read miss & hit



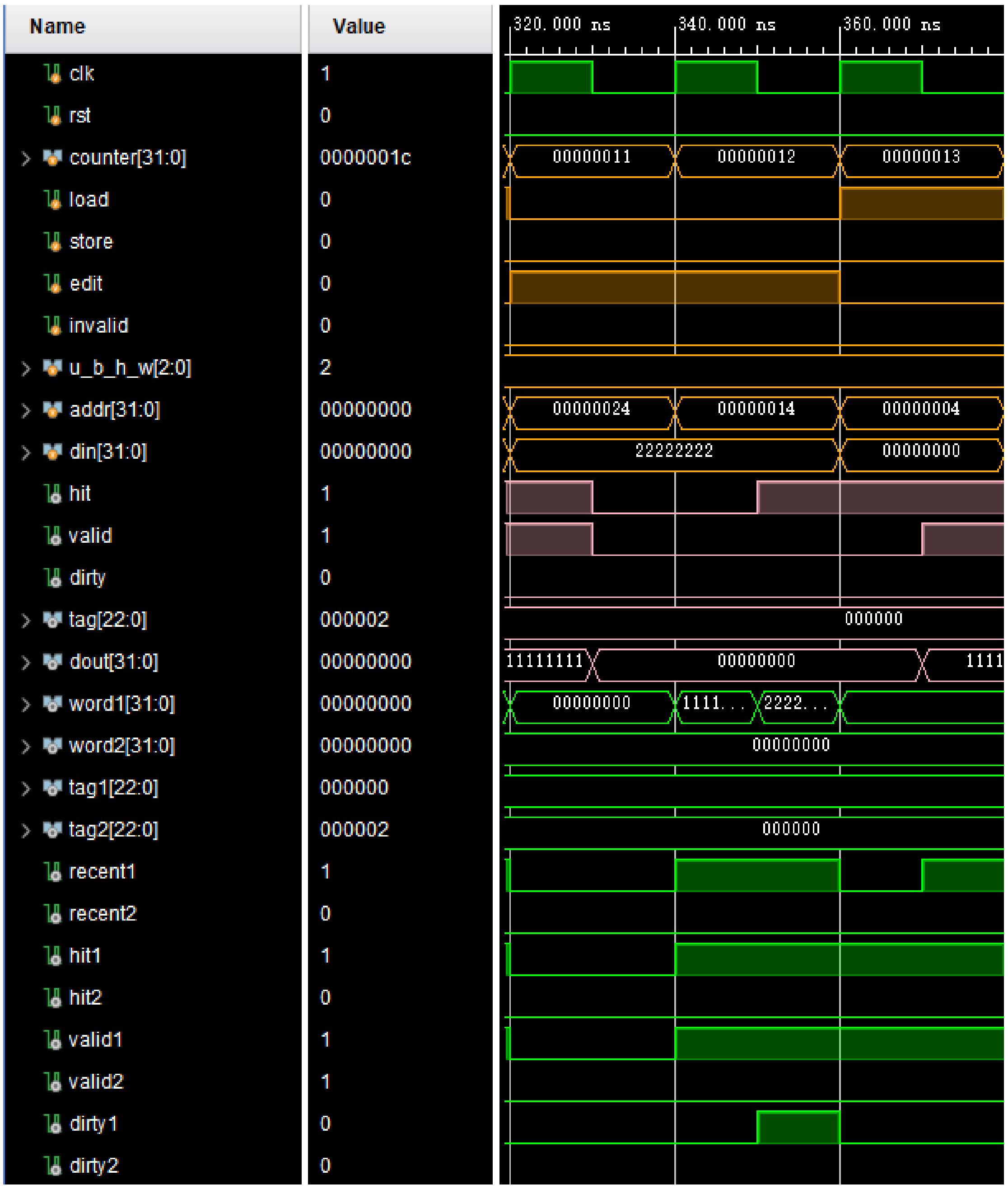
Name	Value
clk	1
rst	0
counter[31:0]	00000006
load	0
store	0
edit	0
invalid	0
u_b_h_w[2:0]	0
addr[31:0]	00000000
din[31:0]	00000000
hit	0
valid	0
dirty	0
tag[22:0]	000000
dout[31:0]	00000000
word1[31:0]	00000000
word2[31:0]	00000000
tag1[22:0]	000000
tag2[22:0]	000000
recent1	0
recent2	0
hit1	0
hit2	0
valid1	0
valid2	0
dirty1	0
dirty2	0

We will see the simulation in 280-340ns.

In 280-300ns, the cache should read from 0x00000020, however the data is not in cache, so it produces a **miss**.

Comparably, in 300-320ns, the cache is experiencing a **hit**. The results could be easily seen when we notice the **hit** signal.

4.2 Write miss & hit



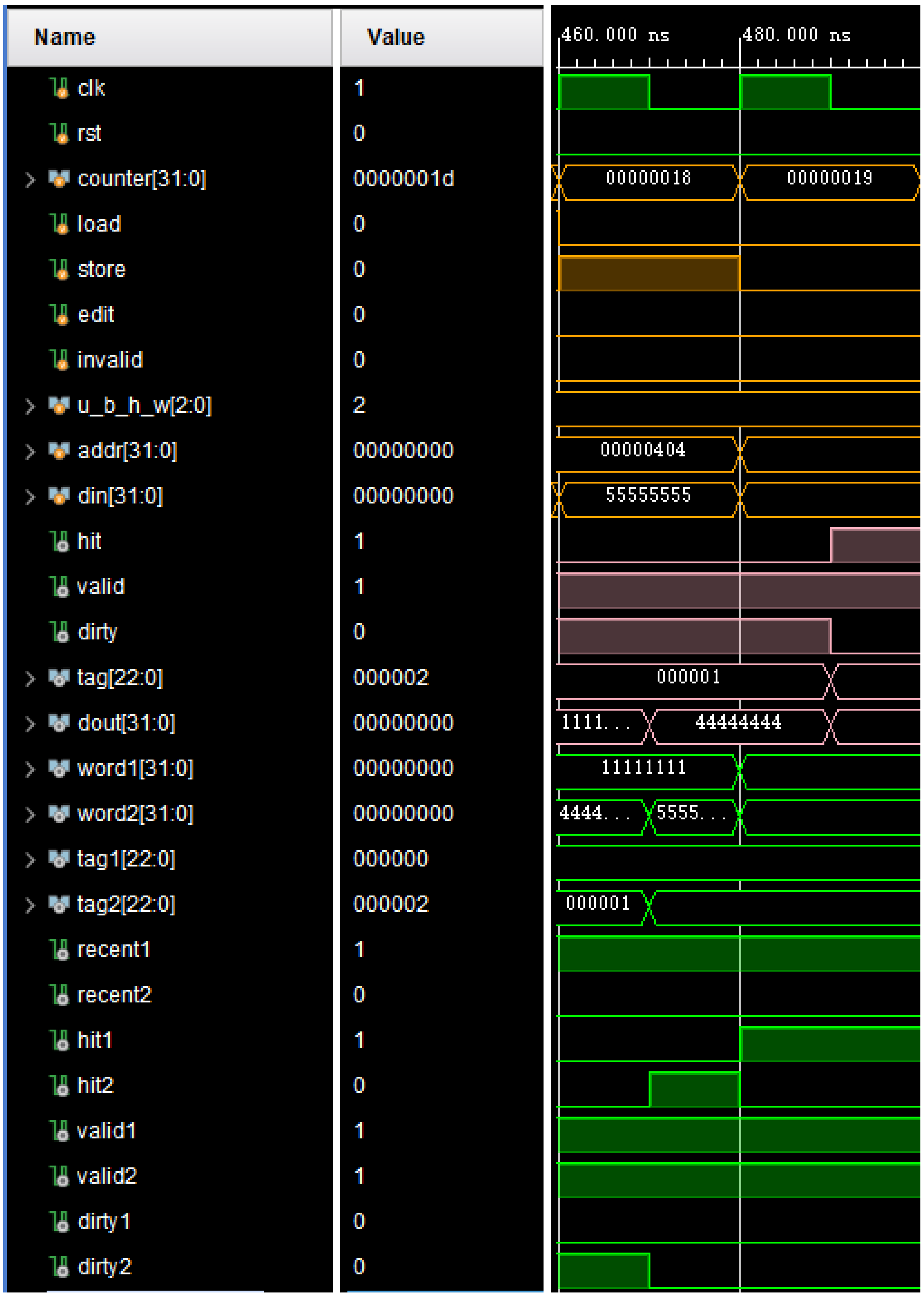
Name	Value
clk	1
rst	0
counter[31:0]	0000001c
load	0
store	0
edit	0
invalid	0
u_b_h_w[2:0]	2
addr[31:0]	00000000
din[31:0]	00000000
hit	1
valid	1
dirty	0
tag[22:0]	000002
dout[31:0]	00000000
word1[31:0]	00000000
word2[31:0]	00000000
tag1[22:0]	000000
tag2[22:0]	000002
recent1	1
recent2	0
hit1	1
hit2	0
valid1	1
valid2	1
dirty1	0
dirty2	0

We will see the simulation in 320-380ns, consisting of 2 operations giving out the **write** signal.

In 320-340ns, the cache should write 0x00000024, however the data is not in cache, so it produces a **miss**.

Comparably, in 340-360ns, the cache is experiencing a **hit**. The results could be easily seen when we notice the **hit** signal, and the data 0x22222222 is written to the cache as **word1** field shows. Besides, we can see that the **dirty** bit has been set.

4.3Block Replacement



Name	Value
clk	1
rst	0
counter[31:0]	0000001d
load	0
store	0
edit	0
invalid	0
u_b_h_w[2:0]	2
addr[31:0]	00000000
din[31:0]	00000000
hit	1
valid	1
dirty	0
tag[22:0]	000002
dout[31:0]	00000000
word1[31:0]	00000000
word2[31:0]	00000000
tag1[22:0]	000000
tag2[22:0]	000002
recent1	1
recent2	0
hit1	1
hit2	0
valid1	1
valid2	1
dirty1	0
dirty2	0

We will see the simulation in 460-500ns.

In 460-480ns, the cache should write 0x00000404. However, the corresponding cache lines have been both occupied, so it must find a cache line to replace.

We can see before that operation **recent1** is valid, so it would replace line 2 according to **LRU principle**. From the **word2** field we know the process correct.

5 Discussion and Conclusion

5.1 Problems

This lab is quite easy and there is no much problem to be discussed.

5.2 Achievements and conclusion

In this experiment, I implemented a 2-way set associative LRU replaced write-back cache, and learned the basic knowledge about cache automatons. And the experiment is successfully finished.