1. 实验目的和要求

Purpose

- Understand Cache Line.
- Understand the principle of Cache Management Unit (CMU) and State Machine of CMU.
- Master the design methods of CMU.
- Master the design methods of Cache Line.
- Master verification methods of Cache Line.

Task

- Design of Cache Line and CMU.
- Verify the Cache Line and CMU.
- Observe the Waveform of Simulation.

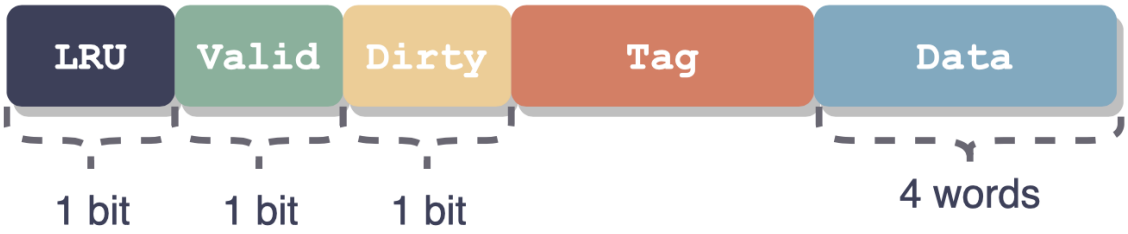
2. 实验内容和原理

Cache Configuration

- 64 cache lines
- 2-way set associative
- 4 words per cache line
- Write Back
- Write Allocate
- LRU Replacement Policy
    - In a 2-way associative cache, once one of the cache line is accessed, a bit indicating it is recently used is set. If there is need for an eviction, the another cache line is evicted.

Cache Line

A cache line consists of the following parts:



3. 实验过程和数据记录

There is only one module implementing cache to be filled out.

cache.v

At first we need to finish some definition. According to the format of cache line and the configuration of cache, we can easily fill out the blank. Since this is a two-way associative cache, we should define different variables for different ways respectively.

```
assign addr_tag = addr[31:9];              //need to fill in
assign addr_index = addr[8:4];            //need to fill in
assign addr_element1 = {addr_index, 1'b0};
assign addr_element2 = {addr_index, 1'b1};      //need to fill in
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]};           //need to fill in

assign word1 = inner_data[addr_word1];
assign word2 = inner_data[addr_word2];                //need to fill in
assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];           //need to
fill in
assign word1 = inner_data[addr_word1];
assign word2 = inner_data[addr_word2];                //need to fill in
assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];           //need to
fill in
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]   ;             //need to fill in

assign recent1 = inner_recent[addr_element1];
assign recent2 = inner_recent[addr_element2];              //need to fill in
assign valid1 = inner_valid[addr_element1];
assign valid2 = inner_valid[addr_element2];               //need to fill in
assign dirty1 = inner_dirty[addr_element1];
assign dirty2 = inner_dirty[addr_element2];               //need to fill in
assign tag1 = inner_tag[addr_element1];
```

```
assign tag2 = inner_tag[addr_element2];                 //need to fill in

assign hit1 = valid1 & (tag1 == addr_tag);
assign hit2 = valid2 & (tag2 == addr_tag);                  //need to fill in
```

In the sequential block, assign each property of the less recently used cache line to each output. In fact, there is an issue worth discussing here, which will be discussed in chapter 5.

```
        valid <= recent1 ? valid2 : valid1;                  //need to fill in
        dirty <= recent1 ? dirty2 : dirty1;                  //need to fill in
        tag <= recent1 ? tag2 : tag1;                      //need to fill in
        hit <= hit1 | hit2;                     //need to fill in
```

If there is a hit for a load instruction, just set the recently-used bit of the corresponding bit.

```
        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 will be refreshed only on r/w hit
                // (including the r/w hit after miss and replacement)
                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 will be refreshed only on r/w hit
                // (including the r/w hit after miss and replacement)
                inner_recent[addr_element1] <= 1'b0;
                inner_recent[addr_element2] <= 1'b1;    //need to fill in
            end
        end
        else dout <= inner_data[ recent1 ? addr_word2 : addr_word1 ];
```

If there is a hit for an instruciton modifying data, we just need to set the data of the cache line according to `u_b_h_w`, which indicates whether the data is signed and the data width, and correspondingly set the property bits.

```
if (edit) begin
            if (hit1) begin
                ...
            end
            else if (hit2) begin
                    //need to fill in
                inner_data[addr_word2] <=
                    u_b_h_w[1] ?        // word?
```

```
                        din
                    :
                        u_b_h_w[0] ?    // half word?
                            addr[1] ?       // upper / lower?
                                {din[15:0], word2[15:0]}
                            :
                                {word2[31:16], din[15:0]}
                        :   // byte
                            addr[1] ?
                                addr[0] ?
                                    {din[7:0], word2[23:0]}   // 11
                                :
                                    {word2[31:24], din[7:0], word2[15:0]} // 10
                            :
                                addr[0] ?
                                    {word2[31:16], din[7:0], word2[7:0]}   // 01
                                :
                                    {word2[31:8], din[7:0]} // 00
                ;
                inner_dirty[addr_element2] <= 1'b1;
                inner_recent[addr_element1] <= 1'b0;
                inner_recent[addr_element2] <= 1'b1;
            end
        end
```

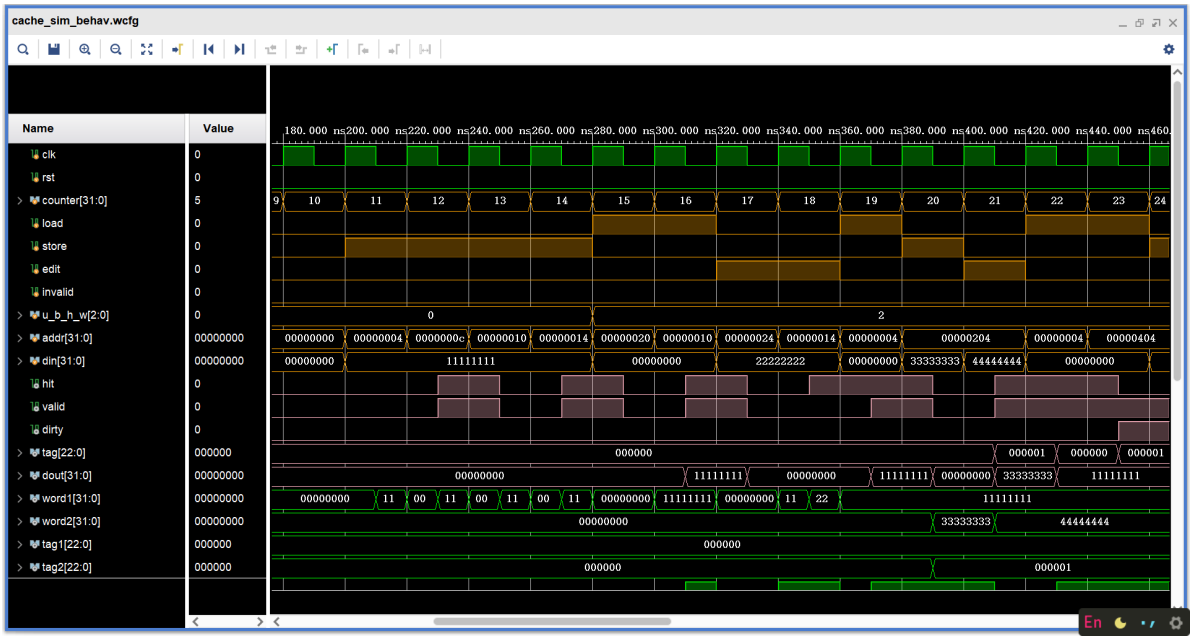
For a store instruciton, if `recent1` and `recent2` are both 0, there is no data in this set and thus the arriving data can be put into the first way. If `recent1` is 0 and `recent2` is set, which means the data in the first way is less recently used and should be replaced.

```
if (store) begin
            if (recent1) begin  // replace 2
                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
                // recent2 == 1 => replace 1
                // recent2 == 0 => no data in this set, place to 1
                //need to fill in
                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
```

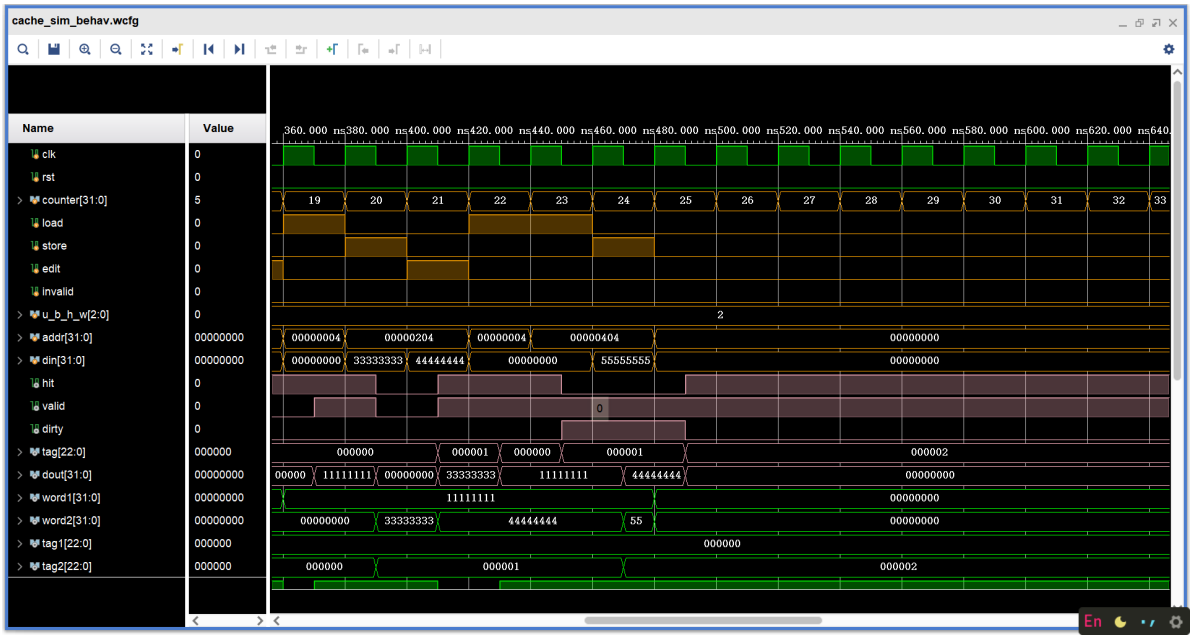
4. 实验结果分析

Simulation

Run the simulation and we can see the result below. As expected, we can see the property bits change at the negative edge. In the cycle where `counter` equals to 15, there is a load miss. We can see `hit` and `valid` are unset at the negative edge. In the next cycle, there is a load hit. We can see `hit` and `valid` are set at the negative edge. Correspondingly the data output `dout` and the bit `recent1` indicating the first way is accessed are set.



Then, in the following cycles, the result is completely the same as the result given by the guidance. Detailed analysis is omitted here, which is similar to the aforementioned part.



5. 讨论与心得

The most confusing part of this experiment is the following part.

```
    valid <= recent1 ? valid2 : valid1;                 //need to fill in
    dirty <= recent1 ? dirty2 : dirty1;                 //need to fill in
    tag <= recent1 ? tag2 : tag1;                     //need to fill in
    hit <= hit1 | hit2;                     //need to fill in
```

In fact my original answer is:

```
    valid <= valid1 ? valid2 : valid1;                  //need to fill in
    dirty <= valid1 ? dirty2 : dirty1;                  //need to fill in
    tag <= valid1 ? tag2 : tag1;                      //need to fill in
    hit <= hit1 | hit2;                     //need to fill in
```

However, such answer causes some difference between the result and the one given by the guidance. Having discussed with TA, we conclude that this is caused by different interpretations of the usage of the output. According to my interpretation, these outputs are for checking the correctness of the code, which should indicate the property of the current set. In fact, these three outputs is **for CMU to select a cache line to be replaced**, which is the explanation from TA and explains why the condition is `rencent1` rather than `valid1`. (I think it is better to explain this purpose in the guidance or the comment, or it's hard to speculate this purpose if we didn't relate it with the next experiment)