Lab3: Cache Design

3220106039

Hanxuan Li(李瀚轩)

1. Introduction

This report presents the implementation of a 2-way set associative cache in Verilog, designed for a 32-bit address space. The cache consists of 64 elements organized into 32 sets, where each set contains 2 cache lines. The cache supports operations such as read, write, edit, and invalidate, while managing valid and dirty bits for effective data storage and retrieval. The report details the functionality of the cache, the addressing scheme used, and the overall design approach.

2. Design Overview

Design Overview

Cache Structure

- **Cache Size:** 64 elements (32 sets, 2 ways)
- Address Format:

  32 bits

  - **Tag:** 23 bits
  - **Index:** 5 bits (for 32 sets)
  - **Word:** 2 bits
  - **Byte:** 2 bits

Input/Output Signals

- **Inputs:**
  - `clk` : Clock signal for synchronization.
  - `rst` : Reset signal for cache initialization.
  - `addr` : 32-bit address for data access.
  - `load` : Control signal for read operations.
  - `store` : Control signal for write operations.
  - `edit` : Control signal to mark data as dirty.
  - `invalid` : Control signal to invalidate cache entries.
  - `u_b_h_w` : Control signal for data width and signedness.
  - `din` : Data input for write operations.
- **Outputs:**
  - `hit` : Indicates whether the cache hit occurred.
  - `dout` : Data output from the cache.

- `valid` : Indicates the validity of the cache entry.
- `dirty` : Indicates whether the cache entry has been modified.
- `tag` : The tag of the currently accessed cache line.

Internal Signals and Registers

- **Recent Tracking:** `inner_recent` array to track the most recently accessed cache lines.
- **Validity Tracking:** `inner_valid` array to track the validity of each cache line.
- **Dirty Tracking:** `inner_dirty` array to track if the cache line has been modified.
- **Tag Storage:** `inner_tag` array for storing tag values associated with each cache line.
- **Data Storage:** `inner_data` array for storing the actual data in the cache.

Address Decoding

The address is broken down as follows:

- **Tag:** Extracted from the upper 23 bits of the address.
- **Index:** Bits 8 to 4 of the address determine which cache set to access.
- **Element Indices:** Derived from the index to identify the specific cache lines in the selected set.

```
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 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] ? word1[31:24] : word1[23:16] : addr[0] ?
word1[15:8] : word1[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
```

Cache Operations

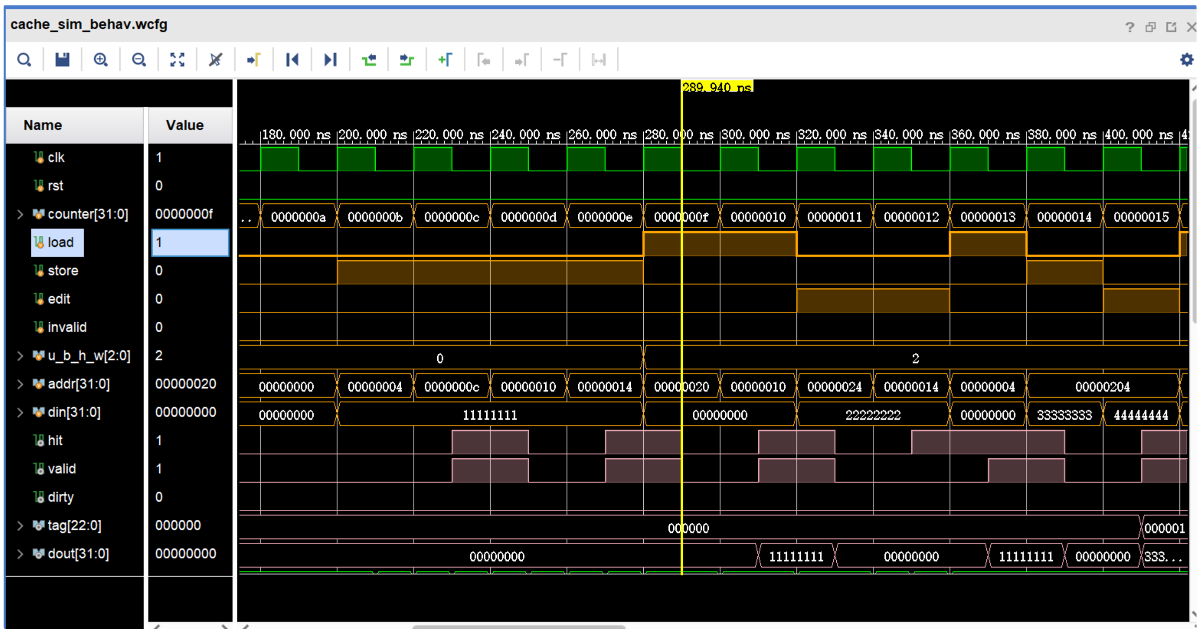
1. **Read Operations (`load`):**
   - Determine if the data is present in the cache (`hit1` and `hit2`).
   - Read data based on the control signal `u_b_h_w` (word, half-word, byte).
   - Update the recent access tracking bits.
2. **Write Operations (`store`):**
   - Write new data to the cache and update the valid and dirty bits accordingly.
   - Decide which cache line to replace based on the recent usage.
3. **Edit Operations (`edit`):**
   - Modify specific bytes/words in the cache data while marking it as dirty.
4. **Invalidate Operations (`invalid`):**
   - Reset the valid and dirty bits for specific cache lines, marking them as invalid.

```
always @ (posedge clk) begin
        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

        // read $ with load==0 means moving data from $ to mem
        // no need to update recent bit
        // otherwise the refresh process will be affected
        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
```

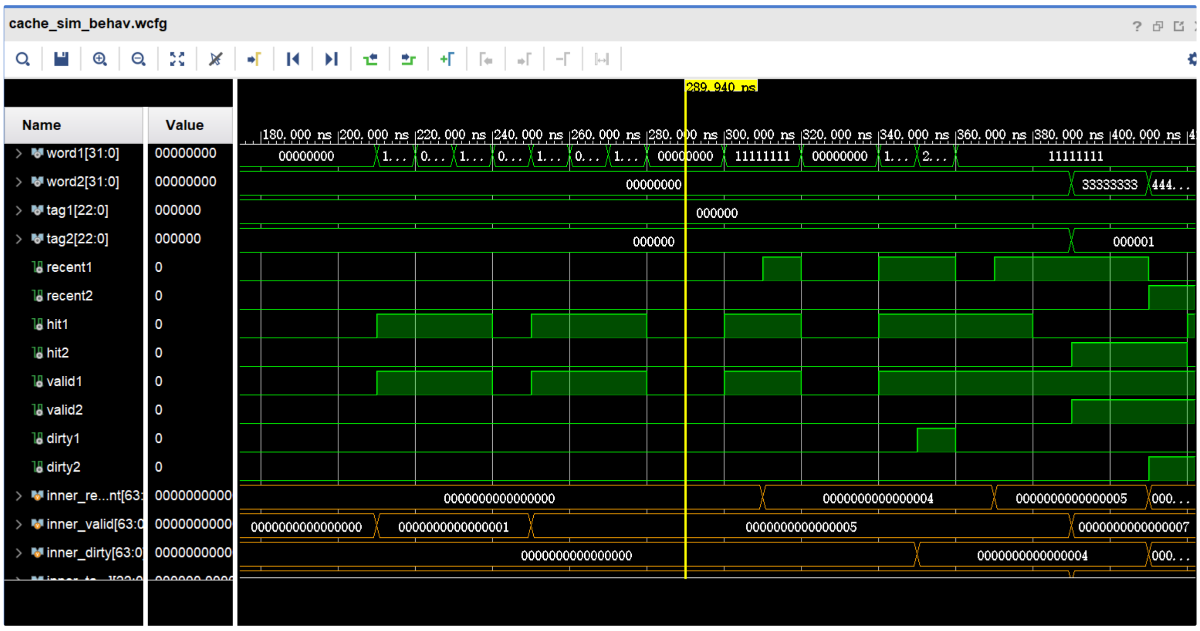
```
                    //need to fill in
                dout  ≤  u_b_h_w[1] ? word2:
                        u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 :
{16{half_word2[15]}}, half_word2} :
                        {u_b_h_w[2] ? 24'b0 : {24{byte2[7]}}, byte2};
                inner_recent[addr_element1]  ≤  1'b0;
                inner_recent[addr_element2]  ≤  1'b1;
            end
        end
        else dout  ≤  inner_data[ recent1 ? addr_word2 : addr_word1 ];

        if (edit) begin
            if (hit1) begin
                inner_data[addr_word1]  ≤
                    u_b_h_w[1] ?        // word?
                        din
                    :
                        u_b_h_w[0] ?    // half word?
                            addr[1] ?       // upper / lower?
                                {din[15:0], word1[15:0]}
                            :
                                {word1[31:16], din[15:0]}
                        :   // byte
                            addr[1] ?
                                addr[0] ?
                                    {din[7:0], word1[23:0]}   // 11
                                :
                                    {word1[31:24], din[7:0], word1[15:0]} //
10
                            :
                                addr[0] ?
                                    {word1[31:16], din[7:0], word1[7:0]}   //
01
                                :
                                    {word1[31:8], din[7:0]} // 00
                ;
                inner_dirty[addr_element1]  ≤  1'b1;
                inner_recent[addr_element1]  ≤  1'b1;
                inner_recent[addr_element2]  ≤  1'b0;
            end
            else if (hit2) begin
                    //need to fill in
                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) 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

        // not used currently, can be used to reset the cache.
        if (invalid) begin
            inner_recent[addr_element1] <= 1'b0;
            inner_recent[addr_element2] <= 1'b0;
            inner_valid[addr_element1] <= 1'b0;
            inner_valid[addr_element2] <= 1'b0;
            inner_dirty[addr_element1] <= 1'b0;
            inner_dirty[addr_element2] <= 1'b0;
        end
    end

endmodule
```

3. Evaluation

3.1 Read Hit/Miss


cache_sim_behav.wcfg

Name	Value
clk	1
rst	0
counter[31:0]	0000000f
load	1
store	0
edit	0
invalid	0
u_b_h_w[2:0]	2
addr[31:0]	00000020
din[31:0]	00000000
hit	1
valid	1
dirty	0
tag[22:0]	000000
dout[31:0]	00000000

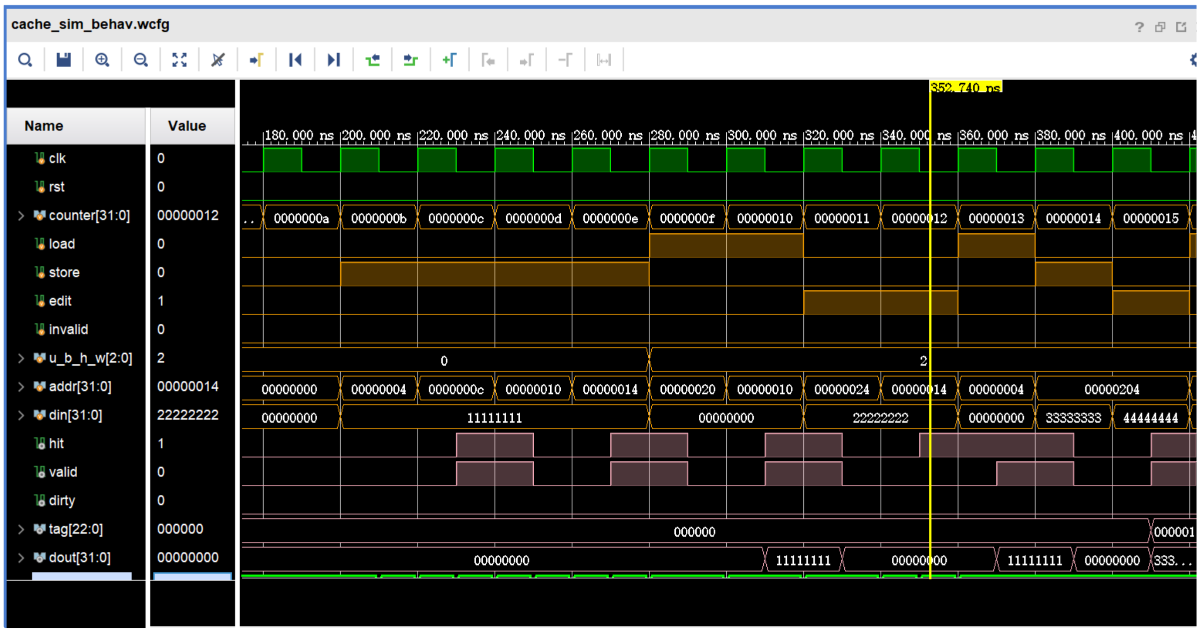


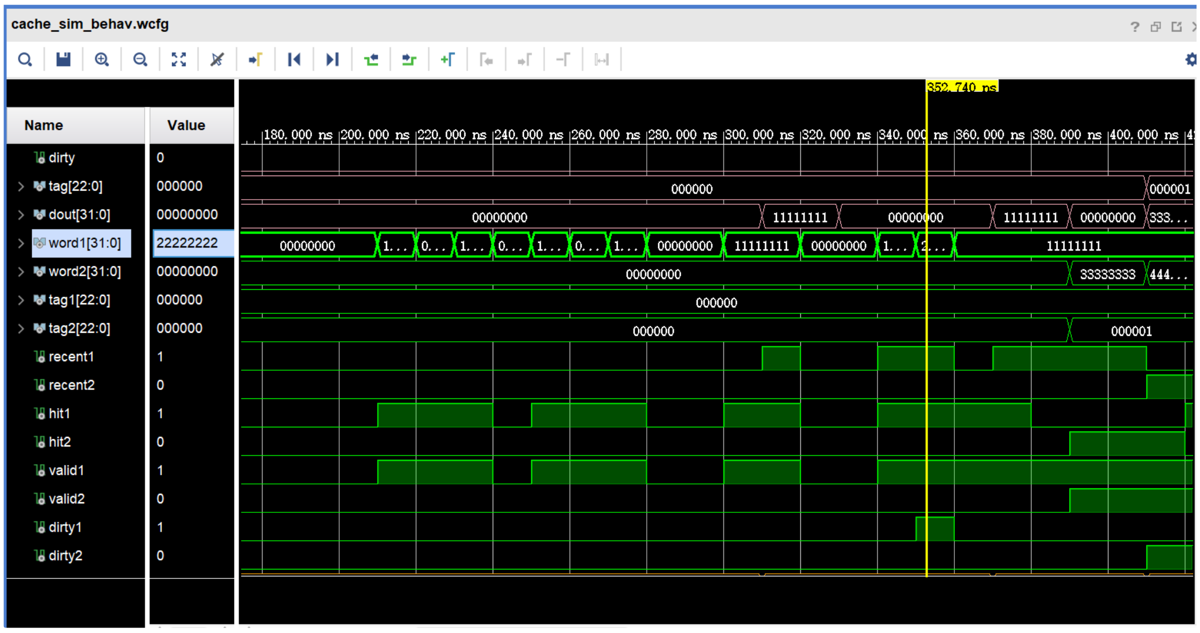
- **Miss:** For the address `addr = 0x20`, we calculate the following:
  - Offset = `0`
  - Index = `0x2`
  - Tag = `0`

  Given `load = 1`, we find `tag2 = tag = 0`, `valid2 = valid = 0`, and `u_b_h_w = 010`. Since `valid2` is `0`, we have a cache miss.
- **Hit:** For the address `addr = 0x10`, we calculate:
  - Offset = `0`
  - Index = `0x1`
  - Tag = `0`

  With `load = 1`, we have `tag1 = tag = 0`, `valid1 = valid = 1`, and `u_b_h_w = 010`. Since `valid1` is `1`, we have a cache hit, loading the word from `0x10` to block `1`, and we set `recent1` to `1`.

3.2 Write Miss/Hit



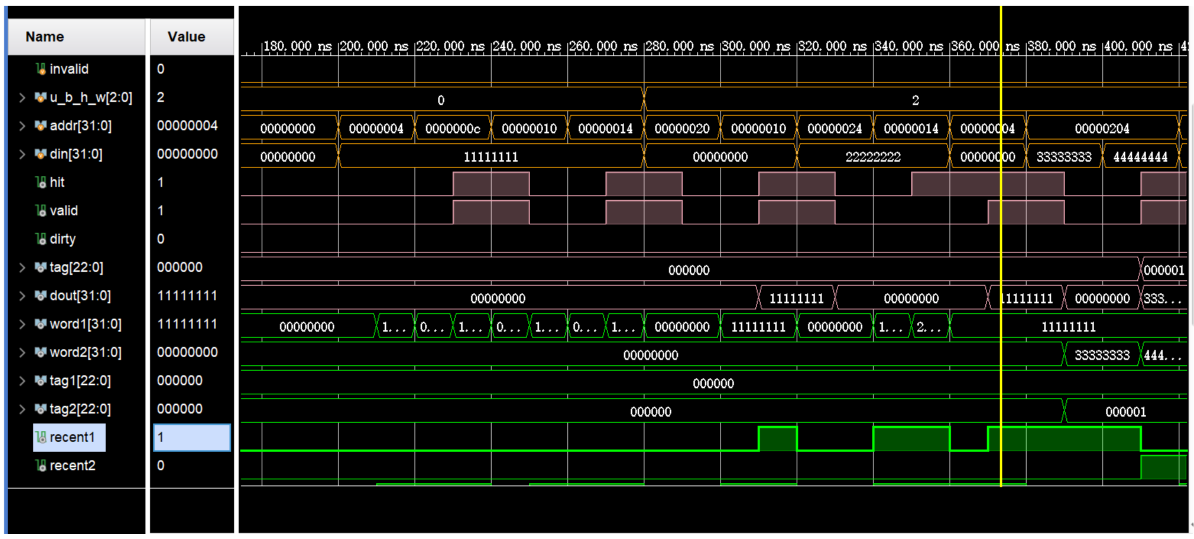


- **Write Miss:** For the address `addr = 0x24` , we have:
  - Offset = `0x4`
  - Index = `0x2`
  - Tag = `0`

  Given `edit = 1` , we find `tag2 = tag = 0` , `valid2 = valid = 0` , and `u_b_h_w = 010` . Since `valid2` is `0` , we have a write miss.
- **Write Hit:** For the address `addr = 0x14` , we have:
  - Offset = `0x4`
  - Index = `0x1`
  - Tag = `0`

  With `edit = 1` , we find `tag1 = tag = 0` , `valid1 = valid = 1` , and `u_b_h_w = 010` . Since `valid1` is `1` , we have a write hit, writing the word to block `1` , and we set `recent1` to `1` .

3.3 Recent bit related

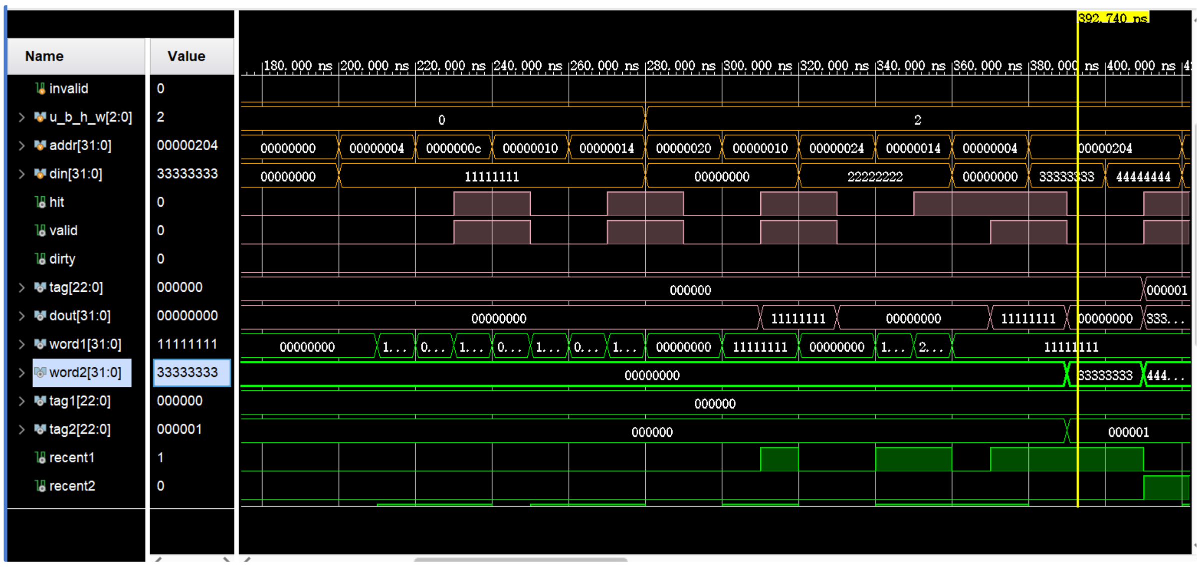


For the address `addr = 0x4` , we calculate:

- Offset = `0x4`
- Index = `0x0`

- Tag = `0x0`

In this case, we find `tag1 = tag` , `valid1 = valid = 1` , and `u_b_h_w = 010` . Therefore, we set `recent1` to `1` .

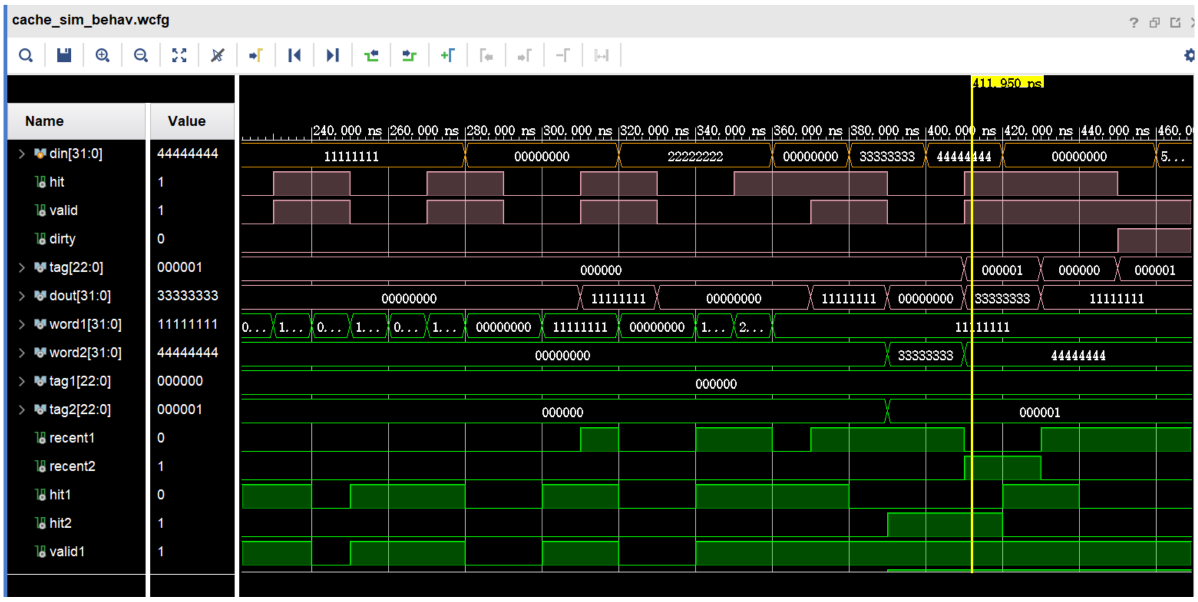


For the address `addr = 0x204` , we calculate:

- Offset = `0x4`
- Index = `0x0`
- Tag = `0x1`

Here, we have `store = 1` , `tag2 = tag` , and `valid2 = valid = 1` . We store data to line `1` of set `0` due to line `0` being the most recent.

3.4 edit

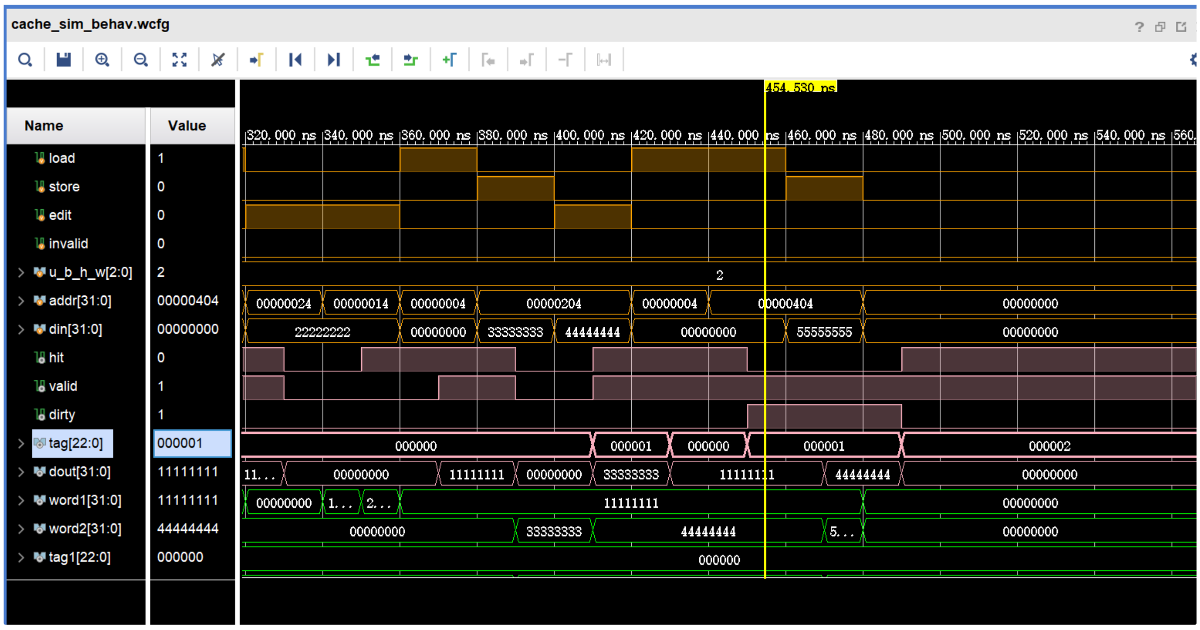


For the address `addr = 0x204` , we calculate:

- Offset = `0x4`
- Index = `0x0`
- Tag = `0x1`

We find `tag2 = tag` , `valid2 = valid = 1` , and `u_b_h_w = 010` . Consequently, we set `recent1` to `0` , `recent2` to `1` , and `dirty2` to `1` .

3.5 tag miss

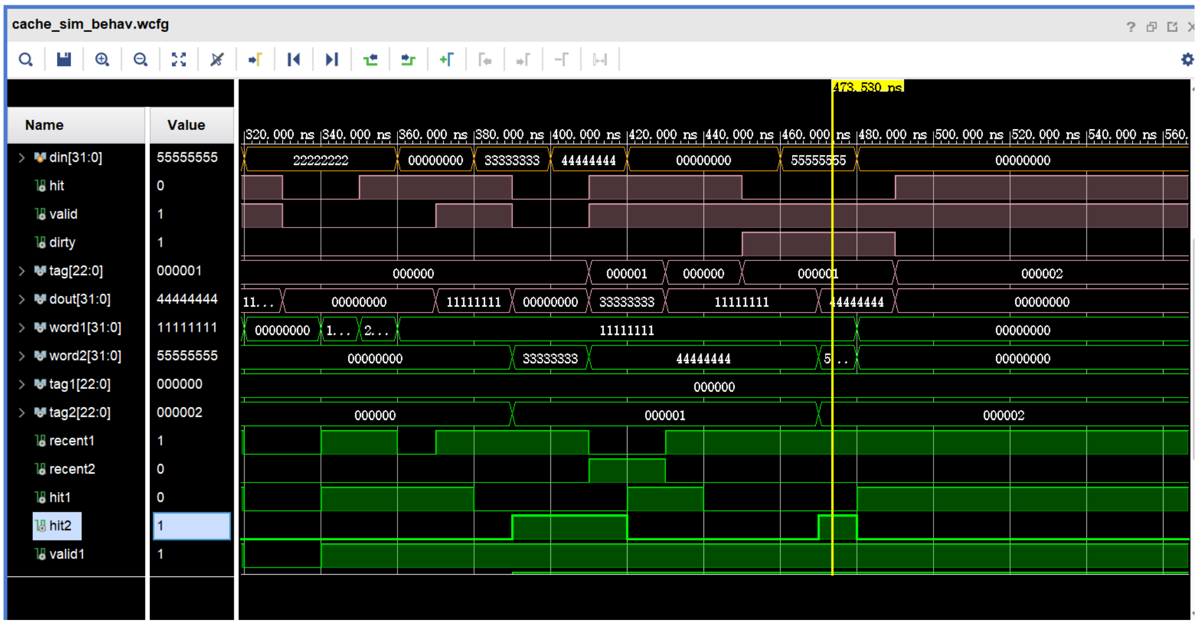


For the address `addr = 0x404` , we have:

- Offset = `0x4`
- Index = `0x0`
- Tag = `0x2`

Here, we find `tag2 = tag` , `valid2 = valid = 1` , and `u_b_h_w = 010` . Since the tag does not match, we experience a tag miss.

3.6 autoplace



For the address `addr = 0x404` , we calculate:

- Offset = `0x4`
- Index = `0x0`
- Tag = `0x2`

In this case, we have `tag2 = tag = 2` , `valid2 = valid = 1` , and `u_b_h_w = 010` . As a result, we execute an auto placement.

The correctness of the implementation has already been verified in the lab class.