浙江大学

本科实验报告

课程名称：计算机体系结构

设计名称：Cache Design

姓　　名：曾帅王异鸣

学　　号：3190105729 3190102780

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：图灵 1901

指导老师：姜晓红

2021 年 10 月 30 日

目录

1 实验目的

本次实验要求我们实现一个简单的二路主关联 Data Cache

1. 理解 Cache Line
2. 理解缓存管理单元及其状态机的原理
3. 掌握缓存管理单元的设计方法
4. 掌握 Cache Line 的设计方法
5. 掌握 Cache Line 的验证方法

2 实验内容

实验的基本要求是实现 RISC-V 架构下，支持 sb，sw，sd，lb，lw，ld 的 Cache 模块。本实验已给出主要框架，需要完成的实验内容如下：

1. 设计 Cache Line 与缓存管理模块
2. 验证 Cache Line 与缓存管理模块
3. 利用测试程序验证 Cache 的执行情况

3 实验原理

3.1 Data Line 设计

实现 Cache 的最基本单元就是 Data Line，在本次实验中 Data Line 含有与 LRU 替换策略有关的 recent 位，与替换与读写策略有关的 Vaild 位和 Dirty 位，还有数据相关的 Tag 位与 Data 位，基本结构显示如下

recent(1bit)	Valid(1bit)	Dirty(1bit)	Tag(23bits)	Data

了解完 Data Line 的结构之后，我们就需要对 Data Line 中的 Data 结构进行分析，在本次实验中，一共有 64 个元素，二路主关联将 64 个数据分为两组，$64 \div 2 = 32 = 2^5$ 所以索引的位数是 5 位，又由于我们需要实现三种 load/store 指令，所以这就要求此 Cache 是字节寻址的，这就意味着需要有 2 位的 byte address(1 word 等于 $4 = 2^2$bytes)。综上所述，一个元素的结构如下：

Tag(23bits)	Index(5bits)	Word(2bits)	Byte(2bits)

通过上述结构的说明，我们就能根据分析出地址各部分的含义

3.2 Cache Mode

本次实验主要采用双路主关联，一次内存的存取会涉及到两个数据块，通过同时对比两个数据块的 Tag 值确定是否命中，具体的结构图如下所示：

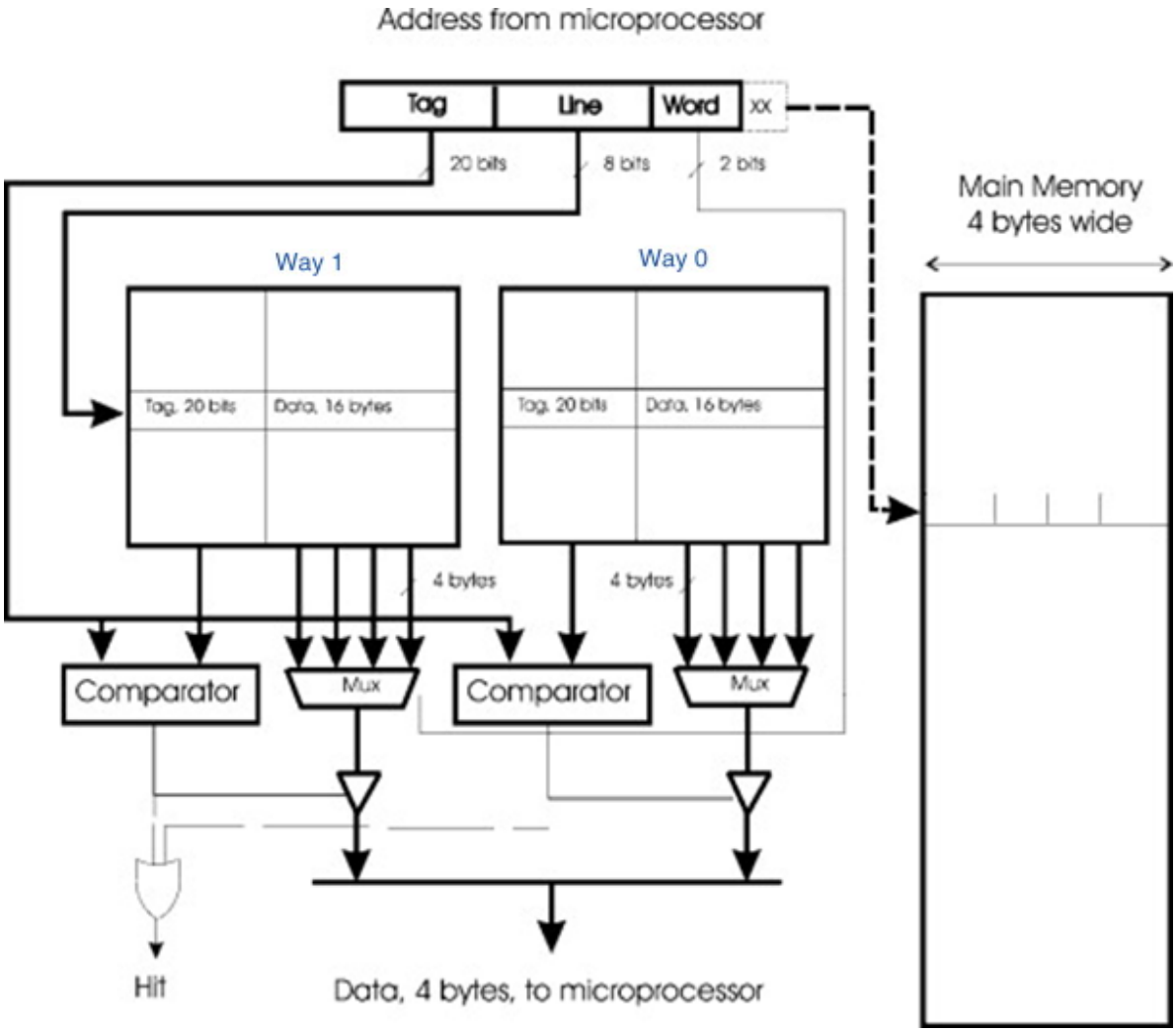

图 3.1: Cache 结构

通过同时比较两个数据块是否命中，来确定整个 Cache 的此次读写是否命中，所以整个 Cache 的命中信号就可以表示为两个子部分的 hit 信号的并。

Write Back 本次实验中实现的是 Write Back 策略，也即修改 Cache 的内容后，Cache 并不会立即更新内存中的内容，而是等到这个 Cache Line 因为某种原因需要从 Cache 中移除时，Cache 才会更新内存中的数据。此策略在本次实验中通过 Dirty 位与 Valid 位进行实现，对缓存中的 valid 为 1(也即有效数据) 的数据进行修改之后，需要将 Dirty 位置为 1，并不直接写入内存，而当此块需要被替换时，才将 Valid 为 1 且 Dirty 为 1 的数据块写入内存，并且 Valid 位与 Dirty 位重新置位。同时，为了对于整个 Cache 的 Dirty 信号与 Valid 信号，我们只需要将其置为下次需要被替换的 Data Line 的 Dirty 位与 Valid 位。

Write Allocate 本次实验中实现了 Write Allocate。Write Allocate 是用于实现 Write miss 之后的处理，如果在写失效之后直接将需要写入的内容写入内存，那么这是 no allocate 的做法，而我们在此实验中是把要写的地址所在的块先从 main memory 调入 cache 中，然后写 cache，这一过程同样依赖于 Valid 位与 Dirty 位，但此过程还涉及到替换策略，我们需要决定哪一块实验被新读入的数据块给替换掉的。

LRU 策略 本次实验中用到的替换策略是 LRU 策略，并且由于此 Cache 仅是二路主关联，所以我们可以只用一位来实现这一策略，但我们读写两个数据块之一时我们就将它的 recent 位置为 1，并且将另一块的 recent 位置为 0，这就可以表明最近访问的是 recent 位为 1 的那一块数据，替换的时候仅需检查 recent 位即可。

3.3 源代码

综合上述设计思想，我们组得设计代码与注释如下：

```
1  // | ———————— address 32 ———————— |
2  // | 31    9 | 8      4 | 3     2 | 1     0 |
3  // | tag 23 | index 5 | word 2 | byte 2 |
4
5  module cache (
6      input wire clk,           // clock
7      input wire rst,           // reset
8      input wire [ADDR_BITS-1:0] addr,  // address
9      input wire load,                  //  read refreshes recent bit
10     input wire store,                 // set valid to 1 and reset dirty to
           0
11     input wire edit,                  // set dirty to 1
12     input wire invalid,               // reset valid to 0
13     input wire [2:0] u_b_h_w,         // select signed or not & data width
14                               // please refer to definition of LB, LH, LW,
                                    LBU, LHU in RV32I Instruction Set
```

```
15      input wire [31:0] din,              // data write in
16      output reg hit = 0,                 // hit or not
17      output reg [31:0] dout = 0,         // data read out
18      output reg valid = 0,               // valid bit
19      output reg dirty = 0,               // dirty bit
20      output reg [TAG_BITS-1:0] tag = 0 // tag bits
21      );
22
23      `include "addr_define.vh"
24
25      wire [31:0] word1, word2;
26      wire [15:0] half_word1, half_word2;
27      wire [7:0]  byte1, byte2;
28      wire recent1, recent2, valid1, valid2, dirty1, dirty2;
29      wire [TAG_BITS-1:0] tag1, tag2;
30      wire hit1, hit2;
31
32      //ELEMENT_NUM = 64, 一个Line有4个WORD
33      reg [ELEMENT_NUM-1:0] inner_recent = 0;
34      reg [ELEMENT_NUM-1:0] inner_valid = 0;
35      reg [ELEMENT_NUM-1:0] inner_dirty = 0;
36      //总共64个元素的tag数组
37      reg [TAG_BITS-1:0] inner_tag [0:ELEMENT_NUM-1];
38      // 64 elements, 2 ways set associative => 32 sets
39      reg [31:0] inner_data [0:ELEMENT_NUM*ELEMENT_WORDS-1];
40
41      // initialize tag and data with 0
42      integer i;
43      initial begin
44          for (i = 0; i < ELEMENT_NUM; i = i + 1)
45              inner_tag[i] = 23'b0;
46
47          for (i = 0; i < ELEMENT_NUM*ELEMENT_WORDS; i = i + 1)
48              inner_data[i] = 32'b0;
49      end
50
51      // the bits in an input address:
52      wire [TAG_BITS-1:0] addr_tag;
53      wire [SET_INDEX_WIDTH-1:0] addr_index;              // idx of set
54      wire [ELEMENT_INDEX_WIDTH-1:0] addr_element1;
55      wire [ELEMENT_INDEX_WIDTH-1:0] addr_element2;       // idx of element
56      wire [ELEMENT_INDEX_WIDTH+ELEMENT_WORDS_WIDTH-1:0] addr_word1;
```

```
57      wire [ELEMENT_INDEX_WIDTH+ELEMENT_WORDS_WIDTH-1:0] addr_word2; //
            element index + word index
58
59      assign addr_tag = addr[31:9];                       //need to fill in
60      //index = set number, 32 = 2 ^ 5
61      assign addr_index = addr[8:4];                      //need to fill in
62      //根据结构赋值
63      assign addr_element1 = {addr_index, 1'b0};
64      assign addr_element2 = {addr_index, 1'b1};          //need to fill in
65      assign addr_word1 = {addr_element1, addr[ELEMENT_WORDS_WIDTH+
            WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]};
66      assign addr_word2 = {addr_element2, addr[ELEMENT_WORDS_WIDTH+
            WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]};          //need to fill in
67
68      //与数据类型有关的赋值
69      assign word1 = inner_data[addr_word1];
70      assign word2 = inner_data[addr_word2];              //need to fill in
71      assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
72      assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];     //need
            to fill in
73      assign byte1      = addr[1] ?
74                          addr[0] ? word1[31:24] : word1[23:16] :
75                          addr[0] ? word1[15:8] :  word1[7:0]   ;
76      assign byte2      = addr[1] ?
77                          addr[0] ? word2[31:24] : word2[23:16] :
78                          addr[0] ? word2[15:8] :  word2[7:0]   ;
                                        //need to fill in
79
80      assign recent1 = inner_recent[addr_element1];
81      assign recent2 = inner_recent[addr_element2];              //need to
            fill in
82      assign valid1 = inner_valid[addr_element1];
83      assign valid2 = inner_valid[addr_element2];               //need to
            fill in
84      assign dirty1 = inner_dirty[addr_element1];
85      assign dirty2 = inner_dirty[addr_element2];               //need to
            fill in
86      assign tag1 = inner_tag[addr_element1];
87      assign tag2 = inner_tag[addr_element2];                  //need to fill
             in
88
89      assign hit1 = valid1 & (tag1 == addr_tag);
```

```
90      assign hit2 = valid2 & (tag2 == addr_tag);                  //need to
            fill in
91
92      always @ (posedge clk) begin
93          valid <= recent1 ? valid2 : valid1;
94          dirty <= recent2 ? dirty1 : dirty2;     //need to fill in
95          tag <= hit1 ? tag1 :
96                  hit2 ? tag2 : 0;                 //need to fill in
97          hit <= hit1 | hit2;                      //need to fill in
98
99          // read $ with load==0 means moving data from $ to mem
100         // no need to update recent bit
101         // otherwise the refresh process will be affected
102         if (load) begin
103             if (hit1) begin
104                 dout <=
105                     u_b_h_w[1] ? word1 :
106                     u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word1
                            [15]}}, half_word1} :
107                     {u_b_h_w[2] ? 24'b0 : {24{byte1[7]}}, byte1};
108
109                 // inner_recent will be refreshed only on r/w hit
110                 // (including the r/w hit after miss and replacement)
111                 inner_recent[addr_element1] <= 1'b1;
112                 inner_recent[addr_element2] <= 1'b0;
113             end
114             else if (hit2) begin
115                 dout <=
116                     u_b_h_w[1] ? word2 :
117                     u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word2
                            [15]}}, half_word2} :
118                     {u_b_h_w[2] ? 24'b0 : {24{byte1[7]}}, byte2};
119
120                 // inner_recent will be refreshed only on r/w hit
121                 // (including the r/w hit after miss and replacement)
122                 inner_recent[addr_element1] <= 1'b0;
123                 inner_recent[addr_element2] <= 1'b1;    //need to fill in
124             end
125         end
126         else dout <= inner_data[ recent1 ? addr_word2 : addr_word1 ];
127
128         if (edit) begin
```

```
129             if (hit1) begin
130                 inner_data[addr_word1] <=
131                     u_b_h_w[1] ?        // word?
132                         din
133                     :
134                         u_b_h_w[0] ?    // half word?
135                             addr[1] ?       // upper / lower?
136                                 {din[15:0], word1[15:0]}
137                             :
138                                 {word1[31:16], din[15:0]}
139                         :   // byte
140                             addr[1] ?
141                                 addr[0] ?
142                                     {din[7:0], word1[23:0]}    // 11
143                                 :
144                                     {word1[31:24], din[7:0], word1[15:0]}
                                        // 10
145                             :
146                                 addr[0] ?
147                                     {word1[31:16], din[7:0], word1[7:0]}
                                          // 01
148                                 :
149                                     {word1[31:8], din[7:0]} // 00
150                 ;
151                 inner_dirty[addr_element1] <= 1'b1;
152                 inner_recent[addr_element1] <= 1'b1;
153                 inner_recent[addr_element2] <= 1'b0;
154             end
155             else if (hit2) begin
156             //need to fill in
157                 inner_data[addr_word2] <=
158                     u_b_h_w[1] ?        // word?
159                         din
160                     :
161                         u_b_h_w[0] ?    // half word?
162                             addr[1] ?       // upper / lower?
163                                 {din[15:0], word2[15:0]}
164                             :
165                                 {word2[31:16], din[15:0]}
166                         :   // byte
167                             addr[1] ?
168                                 addr[0] ?
```

```
169                                     {din[7:0], word2[23:0]}    // 11
170                                 :
171                                     {word2[31:24], din[7:0], word2[15:0]}
                                            // 10
172                             :
173                                 addr[0] ?
174                                     {word2[31:16], din[7:0], word2[7:0]}
                                            // 01
175                                 :
176                                     {word2[31:8], din[7:0]} // 00
177                 ;
178                 inner_dirty[addr_element2] <= 1'b1;
179                 inner_recent[addr_element2] <= 1'b1;
180                 inner_recent[addr_element1] <= 1'b0;
181
182             end
183         end
184
185         if (store) begin
186             if (recent1) begin  // replace 2
187                 inner_data[addr_word2] <= din;
188                 inner_valid[addr_element2] <= 1'b1;
189                 inner_dirty[addr_element2] <= 1'b0;
190                 inner_tag[addr_element2] <= addr_tag;
191             end else begin
192                 // recent2 == 1 => replace 1
193                 // recent2 == 0 => no data in this set, place to 1
194                 //need to fill in
195                 inner_data[addr_word1] <= din;
196                 inner_valid[addr_element1] <= 1'b1;
197                 inner_dirty[addr_element1] <= 1'b0;
198                 inner_tag[addr_element1] <= addr_tag;
199             end
200         end
201
202         // not used currently, can be used to reset the cache.
203         if (invalid) begin
204             inner_recent[addr_element1] <= 1'b0;
205             inner_recent[addr_element2] <= 1'b0;
206             inner_valid[addr_element1] <= 1'b0;
207             inner_valid[addr_element2] <= 1'b0;
208             inner_dirty[addr_element1] <= 1'b0;
```

```
209                 inner_dirty[addr_element2] <= 1'b0;
210             end
211         end
212
213 endmodule
```

4 实验结果与分析

本次实验比较特殊，仅通过仿真即可完成实验结果的正确性检测。

4.1 仿真代码设计

为了验证各项功能与各种情况的正确性，我组使用了如下仿真代码

```
1  module cache_sim;
2
3  // Inputs
4  reg clk;
5  reg rst;
6  reg [31:0] addr;
7  reg load;
8  reg store;
9  reg edit;
10 reg invalid;
11 reg [2:0] u_b_h_w;
12 reg [31:0] din;
13
14 // Outputs
15 wire hit;
16 wire [31:0] dout;
17 wire valid;
18 wire dirty;
19 wire [22:0] tag;
20
21 // Instantiate the Unit Under Test (UUT)
22 cache uut (
23         .clk(~clk),
24         .rst(rst),
25         .addr(addr),
26         .load(load),
27         .store(store),
```

```
28          .edit(edit),
29          .invalid(invalid),
30          .u_b_h_w(u_b_h_w),
31          .din(din),
32          .hit(hit),
33          .dout(dout),
34          .valid(valid),
35          .dirty(dirty),
36          .tag(tag)
37  );
38
39  initial begin
40          clk = 1;
41          forever #10 clk = ~clk ;
42  end
43
44  reg [31:0]counter = 0;
45
46  always @(posedge clk) begin
47          counter <= counter + 32'b1;
48
49          case (counter)
50                  // Initialize Inputs
51                  32'd0: begin
52                          rst <= 0;
53                          addr <= 0;
54                          load <= 0;
55                          store <= 0;
56                          edit <= 0;
57                          invalid <= 0;
58                          u_b_h_w <= 0;
59                          din <= 0;
60                  end
61
62                  // init
63                  32'd10: begin
64                          load <= 0;
65                          store <= 1;
66                          edit <= 0;
67
68                          din <= 32'h11111111;
69                          addr <= 32'h00000004;
```

```
70                  end
71
72                  32'd11: begin
73                          addr <= 32'h0000000C;
74                  end
75
76                  32'd12: begin
77                          addr <= 32'h00000010;
78                  end
79
80                  32'd13: begin
81                          addr <= 32'h00000014;
82                  end
83
84                  // read miss
85                  32'd14: begin
86                          load <= 1;
87                          store <= 0;
88                          edit <= 0;
89
90                          u_b_h_w <= 3'b010;
91                          din <= 0;
92                          addr <= 32'h00000020;
93                  end
94
95                  // read hit
96                  32'd15: begin
97                          u_b_h_w <= 3'b010;
98                          addr <= 32'h00000010;
99                  end
100
101                 // write miss
102                 32'd16: begin
103                         load <= 0;
104                         store <= 0;
105                         edit <= 1;
106
107                         u_b_h_w <= 3'b010;
108                         din <= 32'h22222222;
109                         addr <= 32'h000000024;
110                 end
111
```

```
112                 // write hit
113                 32'd17: begin
114                         u_b_h_w <= 3'b010;
115                         addr <= 32'h00000014;
116                 end
117
118
119                 // read line 0 of set 0, set recent bit
120                 32'd18: begin
121                         load <= 1;
122                         store <= 0;
123                         edit <= 0;
124
125                         u_b_h_w <= 3'b010;
126                         din <= 0;
127                         addr <= 32'h00000004;
128                 end
129
130                 // store to line 1 of set 0 due to line 0 recent
131                 32'd19: begin
132                         load <= 0;
133                         store <= 1;
134                         edit <= 0;
135
136                         u_b_h_w <= 3'b010;
137                         din <= 32'h33333333;
138                         addr <= 32'h00000204;
139                 end
140
141                 // edit line 1 of set 0, set dirty & recent
142                 32'd20: begin
143                         load <= 0;
144                         store <= 0;
145                         edit <= 1;
146
147                         u_b_h_w <= 3'b010;
148                         din <= 32'h44444444;
149                         addr <= 32'h00000204;
150                 end
151
152                 // read line 0 of set 0, set recent bit
153                 32'd21: begin
```

```
154                         load <= 1;
155                         store <= 0;
156                         edit <= 0;
157
158                         u_b_h_w <= 3'b010;
159                         din <= 0;
160                         addr <= 32'h00000004;
161                 end
162
163                 // read miss, tag mismatch. output tag (of line 1), valid
                       and dirty == 1
164                 32'd22: begin
165                         load <= 1;
166                         store <= 0;
167                         edit <= 0;
168
169                         u_b_h_w <= 3'b010;
170                         din <= 32'h0;
171                         addr <= 32'h00000404;
172                 end
173
174                 // auto replace line 1 of set 0
175                 32'd23: begin
176                         load <= 0;
177                         store <= 1;
178                         edit <= 0;
179
180                         u_b_h_w <= 3'b010;
181                         din <= 32'h55555555;
182                         addr <= 32'h00000404;
183                 end
184
185                 // clear
186                 default: begin
187                         load <= 0;
188                         store <= 0;
189                         edit <= 0;
190                         din <= 0;
191                         addr <= 0;
192                 end
193         endcase
194 end
```

```
195
196 endmodule
```

4.2 仿真结果分析

根据仿真代代码，我们组对仿真结果的详细分析如下所示：首先我们需要对整个 cache 进行初始化，清空所有缓存并重置所有控制信号，这个过程持续十个周期

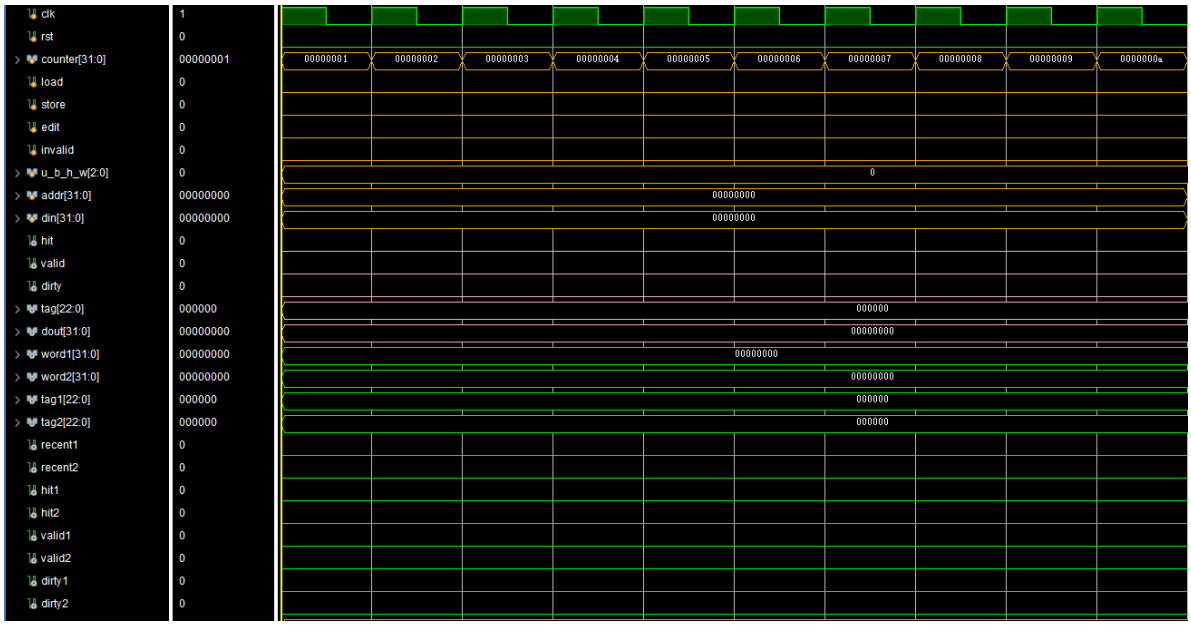

图 4.2: 初始化过程

Store 实现 Cache 的初始化 状态初始化之后，首先验证 store 的正确性，首先输入的数据为 32'h11111111，地址为 32'h00000004，此时 hit 信号为 0，说明是写缺失。然后紧接着保持需要写入的数据不变，将写入地址改为 32'h0000000C，此时我们可以发现 32'h0000000C 的 Index 与 Tag 均与 32'h00000004 相同，仿真结果上 hit 信号也升起，结果正确。继续保持写入数据不变，将写入地址改为 32'h00000010，此时 Index 与 cache 中的已经写入的 Data Line 均不同，hit 信号没有升起，Cache 将对应块读入。最后将写入地址改变为 32'h00000014 此地址对应的 Data Line 与 32'h00000010 相同，故又会产生 hit 信号，仿真结果正确。至此，Cache 的 Index 为 0, 1 的两个 Data Line 都被填满。

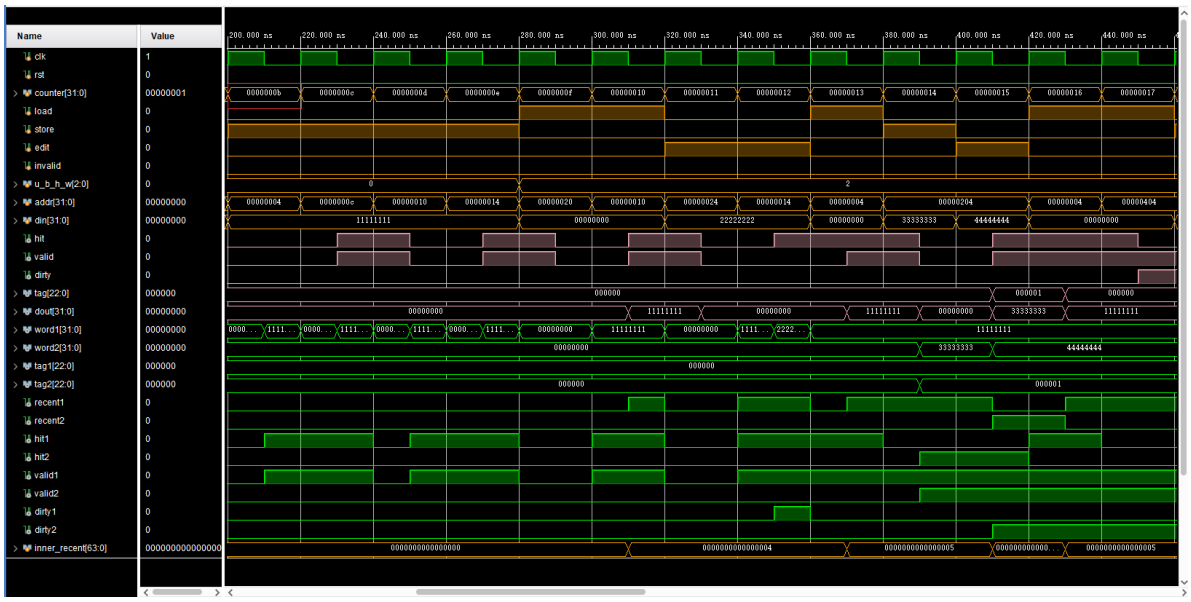

图 4.3: Store 过程

Read 验证 通过上述过程，Cache 已经被初始化完毕，我们写下来验证 read miss 的情形，我们采用 load 读取 32'h00000020 地址所对应的一个 Word 的形式来验证，32'h00000020 的 Index 为 2，会产生 read miss，hit 信号没有升起。在下一个周期，当我们读取 32'h00000010 的一个 Word 时，会发现 Index 为 1 的 Data Line 中有 Tag 为 0 的 Data Line，此时 hit 信号升起，并且 Tag 位 0 的数据块由于产生了读取操作，属于最近被访问，它的 recent 位会被置为 1，与之对应的另一 Data Line 的 recent 位被置为 0 从仿真图的对应位置来看，仿真结果正确。

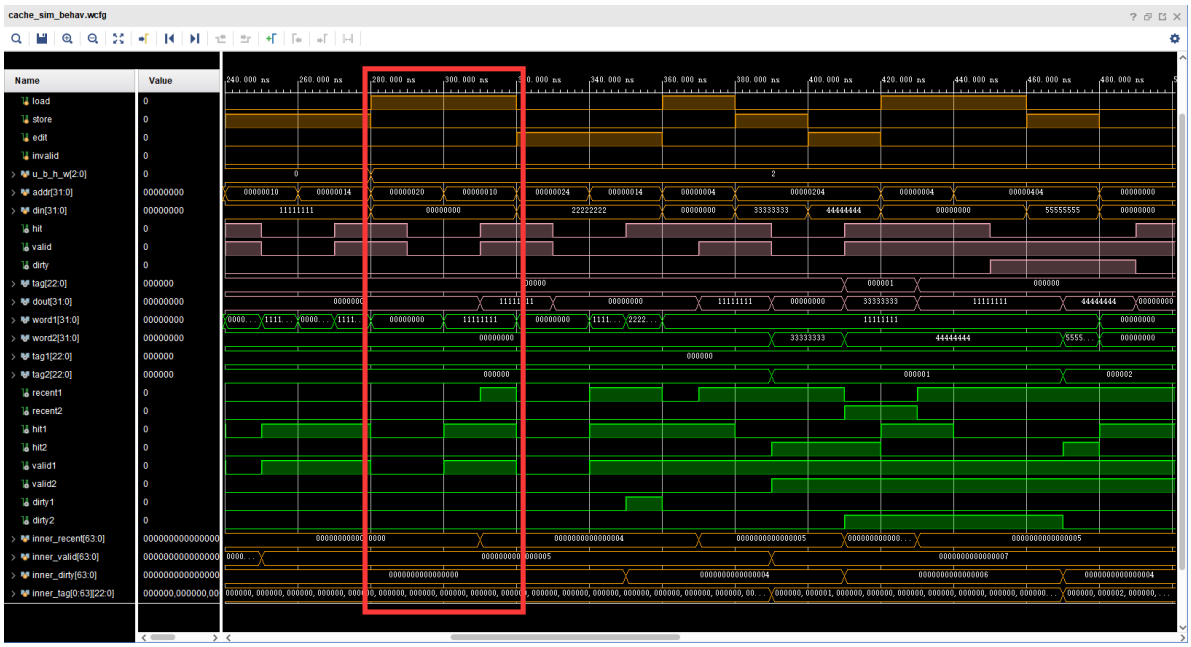

图 4.4: Read 过程

Write 验证　下面我们继续来验证 Write 操作的正确性，同样通过 Write hit 与 Write Miss 两种情形分别验证，首先我们使用 edit 向地址为 32'h000000024 的地方修改一个数据，此时 Index 为 2，显然是 Miss 的，此时 hit 信号没有升起，结果正确。但需要修改的地址为 32'h00000014 时，由于前面的读写操作，此地址对应的数据块应该加载进了 Cache 中，所以此时 hit 信号会升起同时我们可以直到是两路中的第一路命中，命中产生修改之后，此块的 recent 位与 Dirty 位都会被置为 1，说明它最近被修改过。

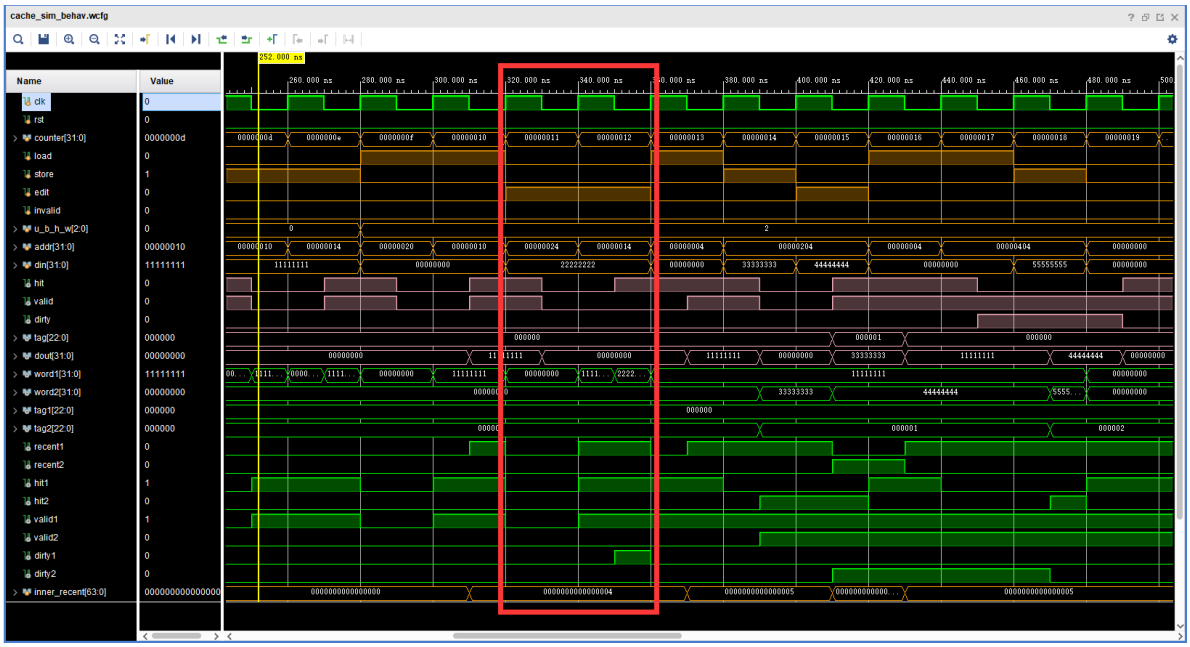

图 4.5: Write 过程

5 讨论与心得

本实验要求在上一个实验的基础上实现中断与异常的相关功能。在本次实验中，我们再次复习了上学期与中断部分相关的内容，并学会了如何在使用流水线的情况下进行中断与异常的相关处理。在实验过程中，我们也遇到了一些问题，如中断与异常时处理 mepc 到底有哪些区别，中断长开时应该如何处理。许多内容在课上都没有提及，我们在查阅了相关资料后得到了答案。