计算机体系结构 - Exp4 Report

1、实验目的和要求

实验目的

- 了解cache 在CPU 中的作用
- 了解cache management unit(CMU) 与cache 和memory 之间的交互机制
- 在CPU 中集成cache

实验要求

- 要求cache 实现2路组相联（cache分为两组，块可以放到对应组号的任意一行当中）
- 要求采用write-back（在数据更新时只写入缓存Cache。只在数据被替换出缓存时，被修改的缓存数据才会被写到后端存储）,write-allocate（先把要写的数据载入到Cache中，写Cache，后续通过flush方式写入到内存中）的数据更新策略。总结之，如果有读写miss，先将块传入，如果要替换的块是dirty的，则把dirty块传入，然后再换块，对这一块进行读或写；如果是hit，那么直接对这个hit的块进行读写。
- 要求采用LRU(最近最少使用，把最近最没使用此处cache块给置换出去) 的替换策略
- 此外，本次实验实现的Cache大小为1024B，每个块有16B，因此合计有1024/16/2=32 个set

LRU	V	D	Tag	Data

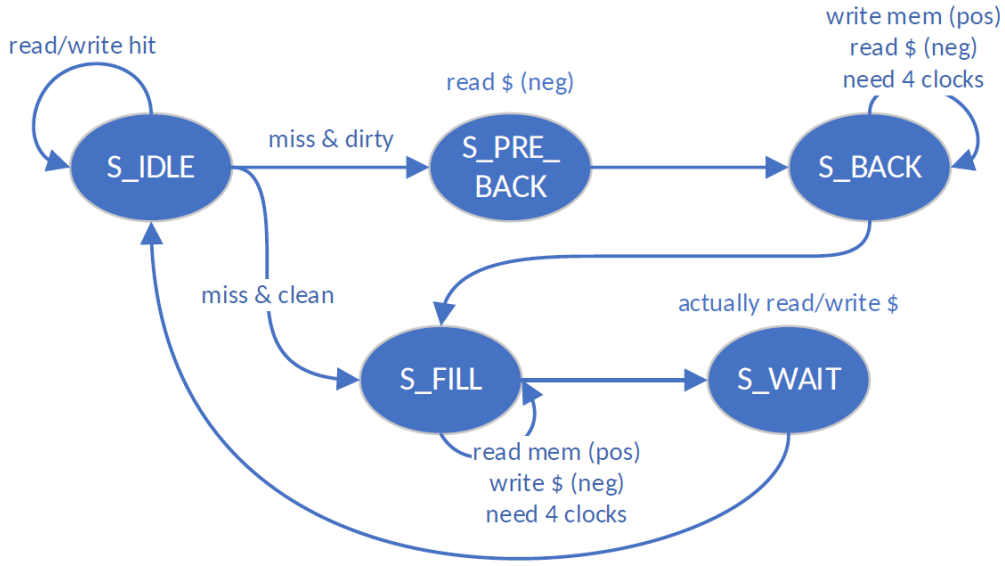
2、实验内容和原理

2.1 CMU设计

2.1.1 概念简述

CMU 是一个专门的控制模块，用于控制数据的读写，对于其控制逻辑一般采用状态机的模式来管理，当然也是可以像CPU 的datapath 一样用流水线来实现CMU。状态机一共有5个状态，如下为：

1. S_IDLE: cache 正常读写，即load/store 命中或者未使用cache。
2. S_PRE_BACK: 为了写回，先进行一次读cache
3. S_BACK: 上升沿将上个状态的数据写回到memory，下降沿从cache 读下次需要写回的数据（因此最后一次读无意义），由计数器控制直到整个cache block 全部写回。由于memory设置为4个周期完成读写操作，因此需要等待memory给出ack信号，才能进行状态的改变。
4. S_FILL: 上升沿从memory 读取数据，下降沿向cache 写入数据，由计数器控制直到整个cache block全部写入。与S_BACK 类似，需要等待ack信号。
5. S_WAIT: 执行之前由于miss 而不能进行的cache 操作



2.1.2 具体实现

状态定义

```
// 定义了5个状态，对应上面的状态图
localparam
    S_IDLE = 0,
    S_PRE_BACK = 1,
    S_BACK = 2,
    S_FILL = 3,
    S_WAIT = 4;
```

状态传递

```
// 用两个reg实现状态的传递
reg [2:0]state = 0;
reg [2:0]next_state = 0;
// 用两个reg实现word_count的传递，这里word_count的作用是记录当前已经读或写的字的个数，
cache的一个element里的data段有4个字，需要记录当前已经读或写到哪个字
reg [ELEMENT_WORDS_WIDTH-1:0]word_count = 0;
reg [ELEMENT_WORDS_WIDTH-1:0]next_word_count = 0;
always @ (posedge clk) begin
    if (rst) begin
        state <= S_IDLE; // 初始状态是S_IDLE
        word_count <= 2'b00;
    end
    else begin
        state <= next_state;
        word_count <= next_word_count;//下一个字
    end
end
```

状态转移

本模块实现几个状态之间的转换，实现的是有限状态机。

```
// 初始在S_IDLE状态，读写命中时无需额外操作，一直在该状态
// 读写miss时，需要把mem的数据写到cache的一个element
```

```
// 若此时分配的element是dirty的，则说明需要先将这个element原来的数据写到mem，此时会经过S_PRE_BACK和S_BACK两个状态把cache的数据先写到mem
// 若分配的element是clean的或者已经把dirty的数据写回mem，则进入S_FILL状态，这个状态会把miss的数据从mem写到cache，完成后进入S_WAIT状态
// S_WAIT状态时，cache中已经有需要的数据，此时cache进行读写，完成后会回到S_IDLE状态
always @ (*) begin
    if (rst) begin
        next_state = S_IDLE;//首次，初始化
        next_word_count = 2'b00;
    end
    else begin
        case (state)
            S_IDLE: begin
                if (en_r || en_w) begin // 如果此时有读或写操作，则需要进行处理
                    if (cache_hit) // 命中就仍在IDLE状态
                        next_state = S_IDLE;
                    else if (cache_valid && cache_dirty)
                        // 没命中且分配的+cache是脏的+已经valid有效的
                        //就去S_PRE_BACK状态,准备写回再写入
                        next_state = S_PRE_BACK;
                    else //干净的话就直接到S_FILL状态，准备直接写入
                        next_state = S_FILL;
                end
                next_word_count = 2'b00;//方便后面从头开始读
            end

            S_PRE_BACK: begin
                next_state = S_BACK; // 无条件转移到S_BACK
                next_word_count = 2'b00;//方便下面从头开始读
            end

            S_BACK: begin
                if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
                    //此时4个word都已经写回内存了，并且都接收了，这个cache块不是脏的了
                    next_state = S_FILL; // 转到S_FILL,准备把新的cache block写入
                else
                    next_state = S_BACK; // 要等到数据全部写入内存，继续等

                if (mem_ack_i)////如果目前已接收，得到消息，准备接收下一个
                    next_word_count = next_word_count + 1;
                else
                    next_word_count = word_count;
            end

            S_FILL: begin
                if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
                    //此时4个word都已经写到cache了，可以转移至S_WAIT
                    next_state = S_WAIT;
                else
                    next_state = S_FILL;

                if (mem_ack_i)//如果目前已接收，准备接收下一个
                    next_word_count = next_word_count + 1;
                else
                    next_word_count = word_count;
            end

            S_WAIT: begin
```

```
                next_state = S_IDLE; // 无条件转移至S_IDLE
                next_word_count = 2'b00;
            end
        endcase
    end
end
```

Cache控制

S_IDLE, S_WAIT: 在 S_IDLE/S_WAIT 模式下，cache 正常运行即可，cache_store 为 1 是因为无需从mem向cache中放入数据

```
S_IDLE, S_WAIT: begin
    cache_addr = addr_rw;
    cache_load = en_r;
    cache_edit = en_w;
    cache_store = 1'b0;
    cache_u_b_h_w = u_b_h_w;
    cache_din = data_w;
end
```

S_BACK, S_PRE_BACK: 由于需要从cache向Mem中写回脏数据，此时 cpu不可以对cache进行操作，设置cache_load， cache_edit，cache_store 为 0，同时不再向 cache 中放入

```
S_BACK, S_PRE_BACK: begin
    cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH], next_word_count,
{ELEMENT_WORDS_WIDTH{1'b0}}};
    cache_load = 1'b0;
    cache_edit = 1'b0;
    cache_store = 1'b0;
    cache_u_b_h_w = 3'b010;
    cache_din = 32'b0;
end
```

S_FILL: S_FILL 是从 mem 中读取数据放入 cache 中，为了防止意外如数据的写覆盖，需要停止 cpu 对 cache 内数据的写和修改操作，若 mem_ack_i 为 1 即 cache 从 mem 成功读取数据可以进行cache数据写回mem,即cache_store 置为1，由于是从 mem 中读出数据所以 cache_din 为 mem_data_i

```
S_FILL: begin
    cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH], word_count,
{ELEMENT_WORDS_WIDTH{1'b0}}};
    cache_load = 1'b0;
    cache_edit = 1'b0;
    cache_store = mem_ack_i;
    cache_u_b_h_w = 3'b010;
    cache_din = mem_data_i;
end
```

stall

cmu中的 stall 会暂停 cpu 中指令的执行，当 cache 与 mem 进行沟通时需要设置 stall 为 1， 故除 S_IDLE 状态均需进行停顿

```
assign stall = (next_state != S_IDLE);  // need to fill in
```

2.2 Cache设计

2-way-associate cache设计

地址解析： 一个地址有 32 位，其中 23 位为 Tag，5 位为 index，说明该 cache 共有 32 行， 剩余有 4 位，可知每行有 4words，16bytes，每行两路，可以计算出一路为 8bytes，即64 位(bit)

具体实现

输入输出信号

```
input wire clk, // clock
input wire rst, // reset
input wire [ADDR_BITS-1:0] addr, // address
input wire load, // read refreshes recent bit
input wire store, // set valid to 1 and reset dirty to 0
input wire edit, // set dirty to 1
input wire invalid, // reset valid to 0
input wire [2:0] u_b_h_w, // select signed or not & data width
// u_b_h_w具体含义如下：
// u_b_h_w[] = 000 -> unsigned byte
// 001 -> signed half_word
// 010 -> word
// 011 -> word
// 100 -> signed byte
// 101 -> unsigned half_word
// 110 -> word
// 111 -> word
input wire [31:0] din, // data write in
output reg hit = 0, // hit or not
output reg [31:0] dout = 0, // data read out
output reg valid = 0, // valid bit
output reg dirty = 0, // dirty bit
output reg [TAG_BITS-1:0] tag = 0 // tag bits
```

cache定位

addr_tag、addr_index根据它们在addr 中的位置将其截取出来；addr_element2 指向一个index下两路中的第二个存储对象，所以只需要在addr_index 后补一位1即可。addr_word2 可仿照addr_word1 完成，其意义为所寻找数据的所在word 位置

```
wire [TAG_BITS-1:0] addr_tag; // tag位
wire [SET_INDEX_WIDTH-1:0] addr_index; // index位
// index可以定位到cache到某个set，但是由于是2-way-associate，所以还需要一个bit来确定是这个set中的哪个element
wire [ELEMENT_INDEX_WIDTH-1:0] addr_element1;
wire [ELEMENT_INDEX_WIDTH-1:0] addr_element2;
// 具体定位到cache到某一个element时，还需要由于数据段有4个words，所以需要word offset(2bits)来定位到具体的word
wire [ELEMENT_INDEX_WIDTH+ELEMENT_WORDS_WIDTH-1:0] addr_word1;
wire [ELEMENT_INDEX_WIDTH+ELEMENT_WORDS_WIDTH-1:0] addr_word2; // element index +word index
```

```
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};
//addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]其实取出了word
offset
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]};
```

从cache取数据

对word、half_word、byte 进行填空，只需要根据addr 中对应的byte 位置、word 位置以及前面所求的addr_word1、2 进行赋值即可。

word2 被存储在inner_data 之中，我们利用之前计算出的addr_word2 即可获取。

half_word2 即word2 的前后部分，byte2即half_word2的前后部分，只要根据addr里的值即可取出

recent2、valid2、dirty2、tag2 的补充也很简单只需按照其对应的1填写即可，含义也比较简单

```
reg [ELEMENT_NUM-1:0] inner_recent = 0;
reg [ELEMENT_NUM-1:0] inner_valid = 0;
reg [ELEMENT_NUM-1:0] inner_dirty = 0;
reg [TAG_BITS-1:0] inner_tag [0:ELEMENT_NUM-1];
// 64 elements, 2 ways set associative => 32 sets
// each element has 4 words
reg [31:0] inner_data [0:ELEMENT_NUM*ELEMENT_WORDS-1]; //[0:64*4-1]
wire [31:0] word1, word2;
wire [15:0] half_word1, half_word2;
wire [7:0] byte1, byte2;
wire recent1, recent2, valid1, valid2, dirty1, dirty2;
wire [TAG_BITS-1:0] tag1, tag2;

//就是 那个地方可能存储的是不同类型的，这里我们假定可能是三种
// 取word
assign word1 = inner_data[addr_word1];
assign word2 = inner_data[addr_word2];

// 取half_word
assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];

// 取byte
// addr[1:0] = 00 -> word[7:0]
// 01 -> word[15:8]
// 10 -> word[23:16]
// 11 -> word[31:24]
assign byte1 = addr[1] ?
addr[0] ? word1[31:24] : word1[23:16] ://以单个位来取数
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];

// 取recent位
```

```
assign recent1 = inner_recent[addr_element1];
assign recent2 = inner_recent[addr_element2];

// 取valid位
assign valid1 = inner_valid[addr_element1];
assign valid2 = inner_valid[addr_element2];

// 取dirty位
assign dirty1 = inner_dirty[addr_element1];
assign dirty2 = inner_dirty[addr_element2];

// 取tag位
assign tag1 = inner_tag[addr_element1];
assign tag2 = inner_tag[addr_element2];
```

命中判断

```
wire hit1, hit2;
// 当element的valid bit为1且tag位和addr的tag部分相同时命中
assign hit1 = valid1 & (tag1 == addr_tag);
assign hit2 = valid2 & (tag2 == addr_tag);
```

读数据(load)

模仿1中即可写出。

```
// read $ with load==0 means moving data from $ to mem
// no need to update recent bit
// otherwise the refresh process will be affected
// 命中时，将数据给dout输出，并将命中的element对应的recent位置1，同一个set的另外一个
element对应的recent位置0
if (load) begin
    if (hit1) begin
        dout <=//dout是每次传递的数据块
            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[addr_element1] <= 1'b0;
        inner_recent[addr_element2] <= 1'b1;
    end
end
// 如果不是读数据，那么dout是最近没有用访问的数据
```

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

写数据（edit）

```
// 命中后将din写到命中的element数据段
// 写完以后将命中的element对应的recent位置1，同一个set的另外一个element对应的recent位置0
// 将element的dirty位置1，因为采用的是write back，先不写内存，所以此时内存和cache数据不同
if (edit) begin
        if (hit2) begin
            //这里的赋值就是进行不同的组装
            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_element2] <= 1'b1;
            inner_recent[addr_element1] <= 1'b0;
    end
        else 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], word2[15:0]}
                    :
                        {word1[31:16], din[15:0]}
                    : // byte
                    addr[1] ?
                        addr[0] ?
                            {din[7:0], word2[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
```

将内存数据写进cache(store)

```
// miss时，需要从内存取数据并且写到cache，此时需要替换，如果element是最近访问过那么就替换另一个
// 因为是刚从内存写进来的数据，那么此时这个element是clean的，dirty位置0
// valid位需要置1
if (store) begin
    if (recent1) begin // replace 2
        inner_data[addr_word2] <= din;//data write in
        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
        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
```

输出信号赋值

```
// 在命中的情况下，valid，dirty和tag是不用考虑的
// valid，dirty，tag三个信号是对于没有命中时，需要分配一个element，给出这个element的状态，取element时还是取最近没有访问的
valid <= (~recent1) & valid1 |
        (recent1) & valid2;
dirty <= (~recent1) & dirty1 |
        recent1 & dirty2;
tag <= {23{~recent1}} & tag1 |
        {23{recent1}} & tag2;
hit <= hit1 | hit2;
```

3、实验过程和数据记录

仿真解析

仿真代码Test Bench

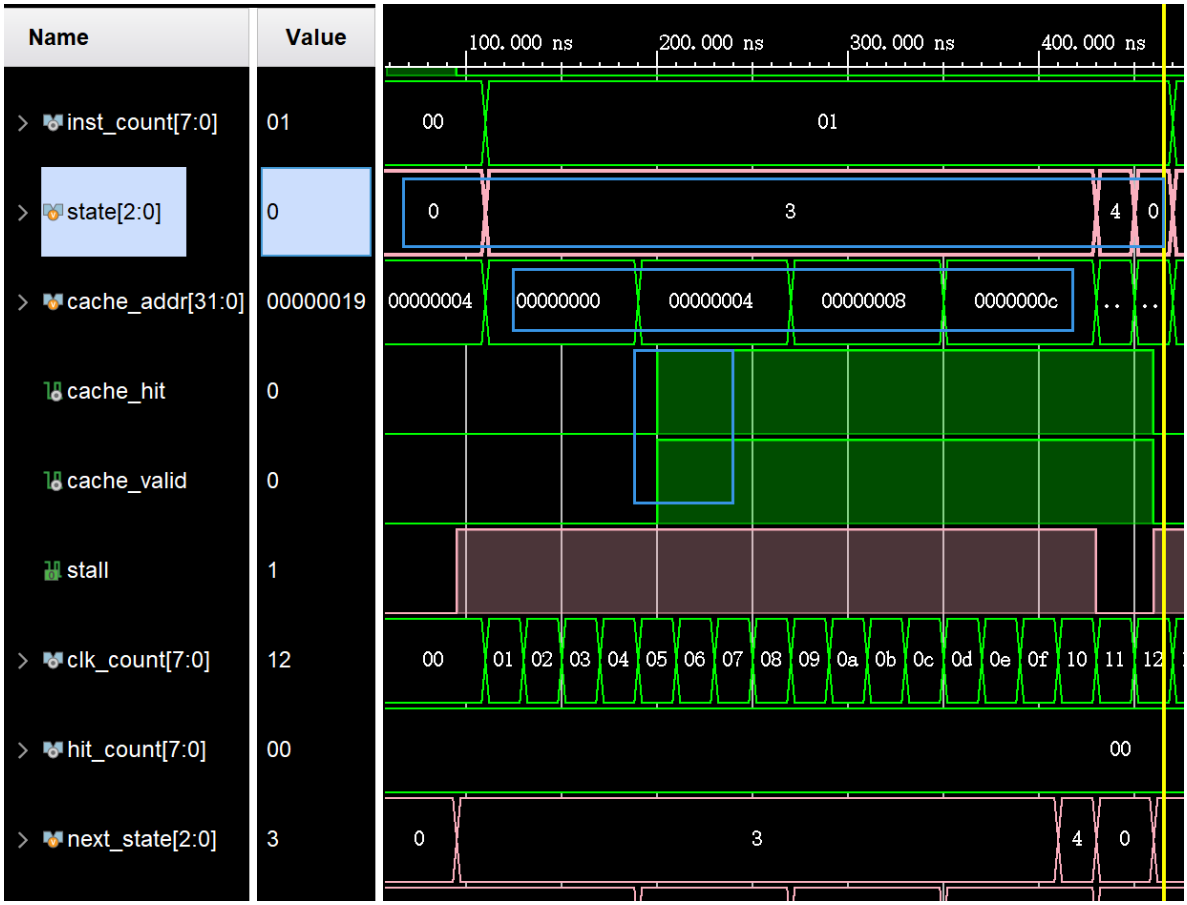
```
initial begin
    data[0] = 40'h0_2_00000004; // read miss 1+17
    data[1] = 40'h0_3_00000019; // write miss 1+17
    data[2] = 40'h1_2_00000008; // read hit 1
    data[3] = 40'h1_3_00000014; // write hit 1
    data[4] = 40'h2_2_00000204; // read miss 1+17
    data[5] = 40'h2_3_00000218; // write miss 1+17
    data[6] = 40'h0_3_00000208; // write hit 1
    data[7] = 40'h4_2_00000414; // read miss + dirty 1+17+17
    data[8] = 40'h1_3_00000404; // write miss + clean 1+17
    data[9] = 40'h0; // end total: 128
end
```

第一次读取的 read miss(clean)

对应实现：data[0] = 40'h0_2_00000004; // read miss 1+17

解释：在read miss发生的情况下，且cache中有空闲可替换位置的情况下，状态转换为 0->3->4->0，即 S_IDLE->S_FILL->S_WAIT->S_IDLE，为正常的read miss且clean情况下的，mem数据写入cache。

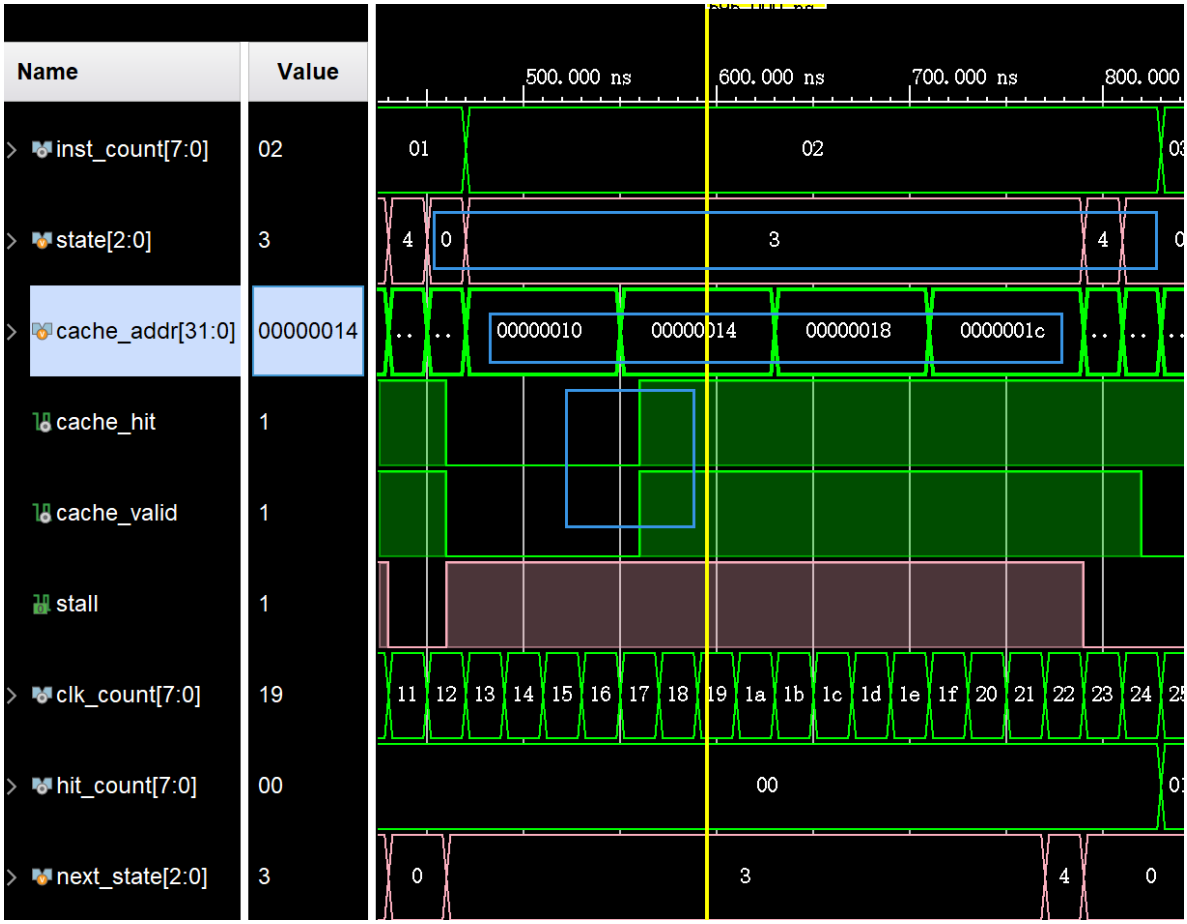
并且在 S_FILL 过程中，由于 cpu 需要地址0x4 处的数据,cache读取了 mem中从 0x0到0xc 的 block，在读取了 0x4 处数据后 cache_hit 与 cache_valid 变为1。



第一次写入的 write miss(clean)

对应实现：data[1] = 40'h0_3_00000019; // write miss 1+17

解释：状态转换依旧为S_IDLE->S_FILL->S_WAIT->S_IDLE，代表write miss且clean情况下将mem 中对应数据写入 cache。并且，写数据地址为 0x19，故将 0x10-0x1c 的块放入 cache 中，cache hit与cache valid也变为1.

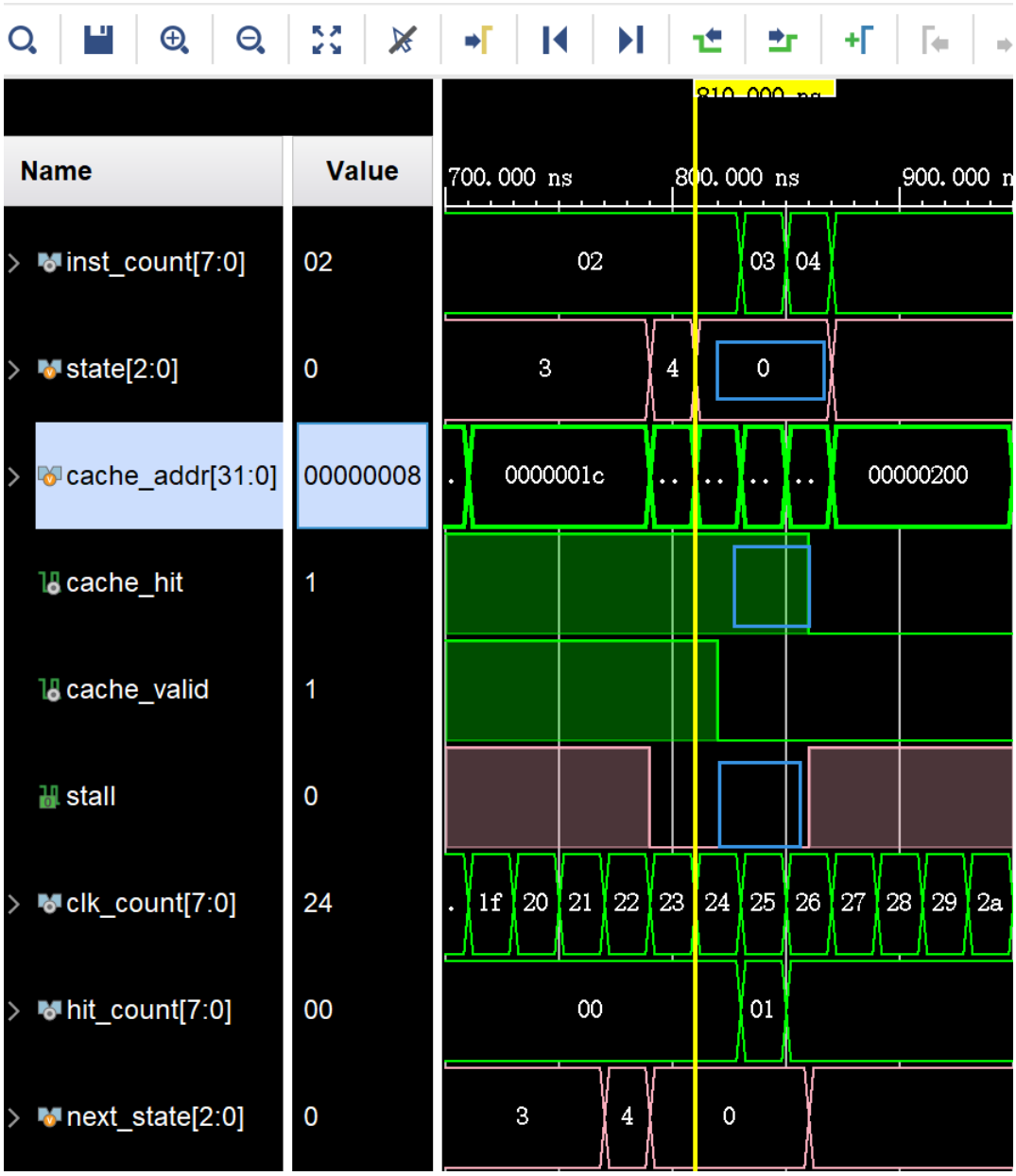


Read hit 与 Write hit

对应实现：

```
data[2] = 40'h1_2_00000008; // read hit 1
data[3] = 40'h1_3_00000014; // write hit 1
```

解释：read hit 与 write hit 发生时状态机始终在 S_IDLE 状态下，cpu 不会发生停顿故第 3、4 条指令直接执行完成，且在该过程中 stall 为 0,cache_hit 也为1



dirty 情况下的miss替换

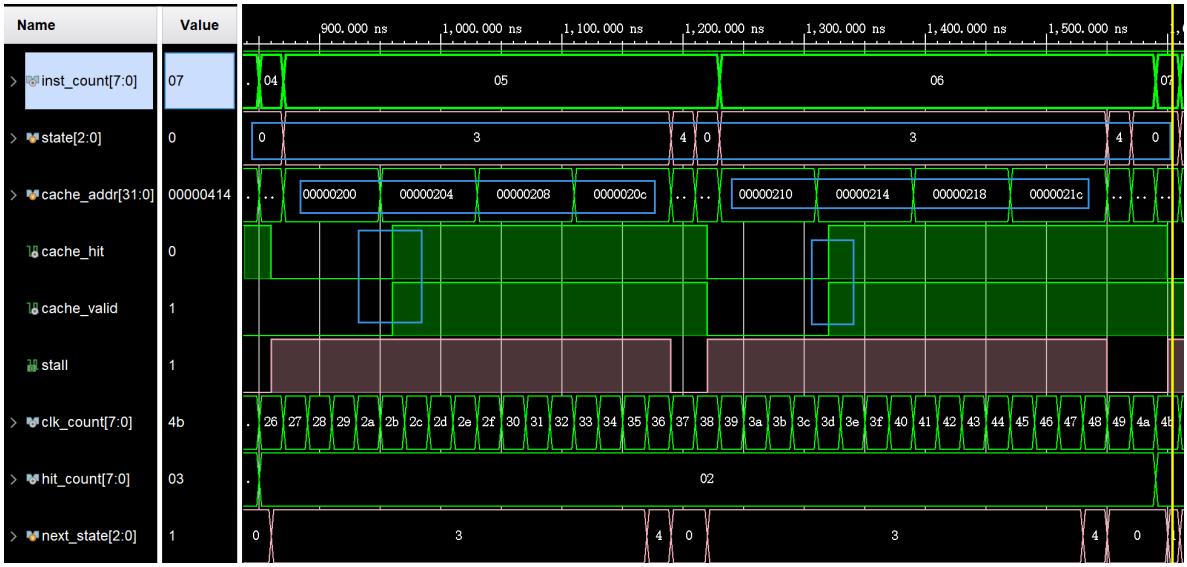
对应实现：

data[4] = 40'h2_2_00000204; // read miss 1+17

data[5] = 40'h2_3_00000218; // write miss 1+17

data[6] = 40'h0_3_00000208; // write hit 1

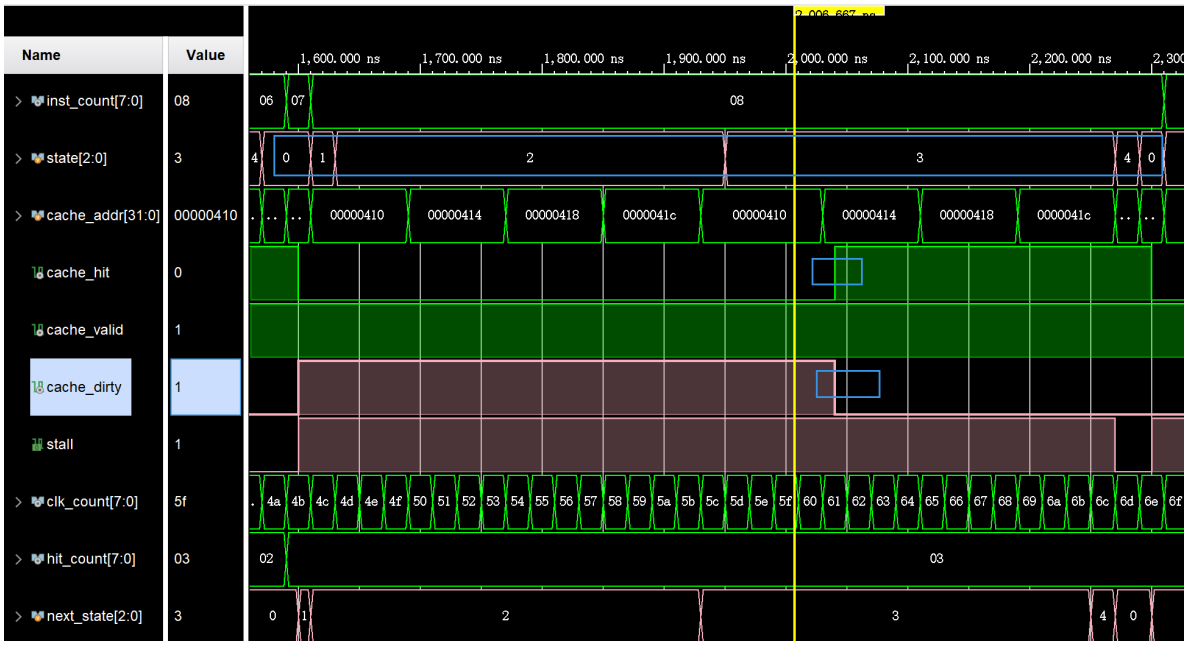
解释：首先这几条，依旧是执行正常替换，经历S_IDLE->S_FILL->S_WAIT->S_IDLE的状态，并装入cache，在对应的遇到时转换cache_hit为1，接着wite hit不停顿，正常为0.



对应实现：

data[7] = 40'h4_2_00000414; // read miss + dirty 1+17+17

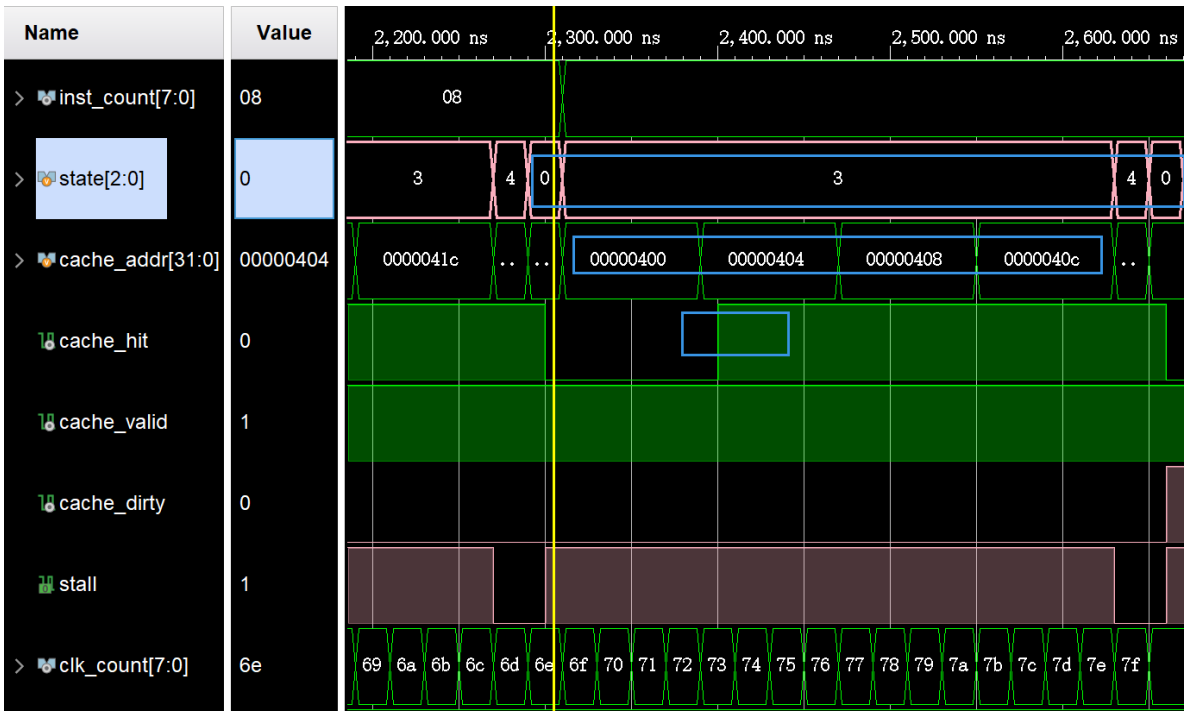
解释：由于所写cache为二路组相联，为了实现dirty情况下的替换需要填满cache中所要操作位置的数据并对其中一个数据进行修改使之dirty为1，在写入时替换掉改数据块。data[7]前面的几个访问正是为了创造这样的条件，可以看到此时的状态变化为0->1->2->3->4->0，正是S_IDLE->S_PRE_BACK->S_BACK->S_FILL->S_WAIT，符合dirty情况下的miss替换



clean 情况下的 miss 替换

对应实现：data[8] = 40'h1_3_00000404; // write miss + clean 1+17

解释：再次对clean的块进行替换，经历S_IDLE->S_FILL->S_WAIT->S_IDLE



上板验证

```
x01=0xABCDE71C
x02=0xFFFFF0F0
x03=0xF0F0F0F0
x04=0x000000F0
x05=0x0000F0F0
x06=0x00000000
x07=0x00008000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x000000ED
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000040
WB_INST=0x00000013
MEMADDR=0xFFFFFFFF
```

4、实验心得和体会

心得

这次的实验主要是一个填空和理解的实验，从原理上来说，和cache紧密联系，想要完成本次实验需要对基础概念（比如write back）这些很清楚，这样才能明白对dirty的处理是怎么进行的，另一方面，对于cmu的处理上，状态转移其实并不难理解，比较需要注意的是什么时候转移，要在这里等多久，这个是由计数器来决定的，需要在读的时候+1，在处理完毕之后再清0。另外，原理上我一开始还对word，half-word和byte有点不理解，后来想了一下是取出正确的位数然后截取。

从实现方面而言的话，这次的实验大部分是填空，很多都是基于前者的模拟，因此比较重要的是对原理的理解，大多我都已经注明好了放在了代码说明里。

思考题

1. 在实验报告分别展示缓存命中、不命中的波形，分析时延差异。

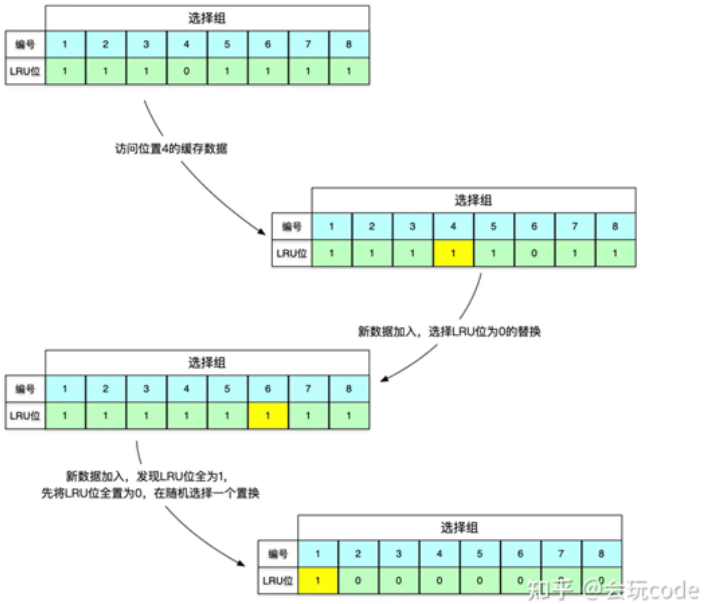
   答：如上图分析

2. 在本次实验中，cache 采取的是2 路组相联，在实现LRU 替换的时候，每一个set 需要用多少bit 来用于真正的LRU 替换实现?

   答：需要2个bit实现LRU即可。每次将最晚访问的块设置成1，将更早访问的块设置成0。即实验中的recent，每次替换recent=0的块即可。每一个块需要1个bit，一共需要2*1=2bit

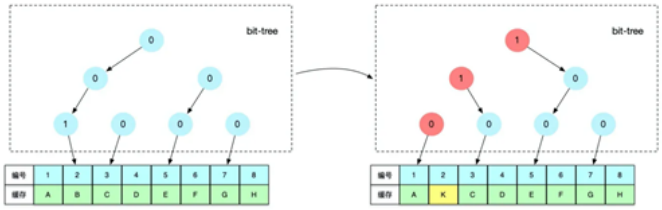
3. 如果要实现cache 的4路组相联，请描述一种真LRU 和一种Pseudo-LRU 的实现方式，并给出实现过程中每一个set 需要用到多少bit 来实现LRU。关于Pseudo-LRU，实现方式可以在网上查阅

   答：如果是LRU替换策略，需要两位age字段，每次将未访问的age减1，替换最小的age即可，共需要4*2=8bit。更经济的情况可以只用1个bit，置0代表可以进行LRU，置1表示暂时不能LRU，过一段时间将1改成0，共需要8bit。

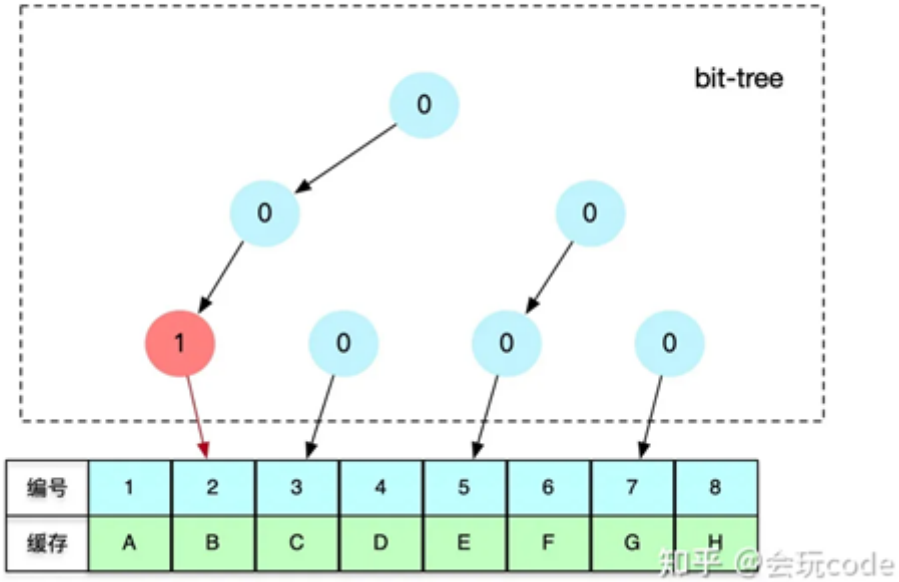


如果采用伪LRU，只需要2位。构建一个高度为2的二叉树，最下面两个叶子节点指向对应的块。每次删除顺着二叉树走即可，每次遍历后，将二叉树的0和1替换。每次访问时，将叶子节点最底下置换。

发生缓存置换时，会从根节点开始寻找，顺着箭头方向找到需要淘汰替换的缓存条目。在寻找过程中，会将路径上的节点箭头全部反转，0变成1，1变成0。比如，要写入新缓存“K”，结果如下。



读取缓存时，将改变指向读取的缓存的节点的箭头指向，比如，读取A时，会将指向A的箭头会被翻转，结果如下图。



读取缓存时，将改变指向读取的缓存的节点的箭头指向，比如，读取A时，会将指向A的箭头会被翻转，结果如下图。