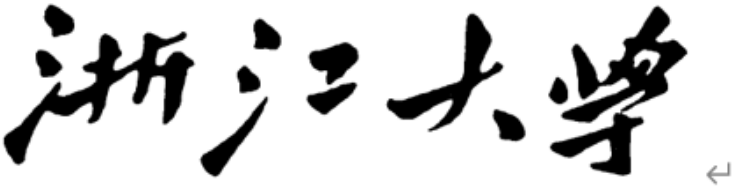


课程名称:	**计算机体系结构**
姓 名:	**杨吉祥**
学 院:	**计算机科学与技术学院**
系:	**竺可桢学院图灵班**
专 业:	**计算机科学与技术**
学 号:	**3230106222**
指导教师:	**常瑞**

一、设计思路

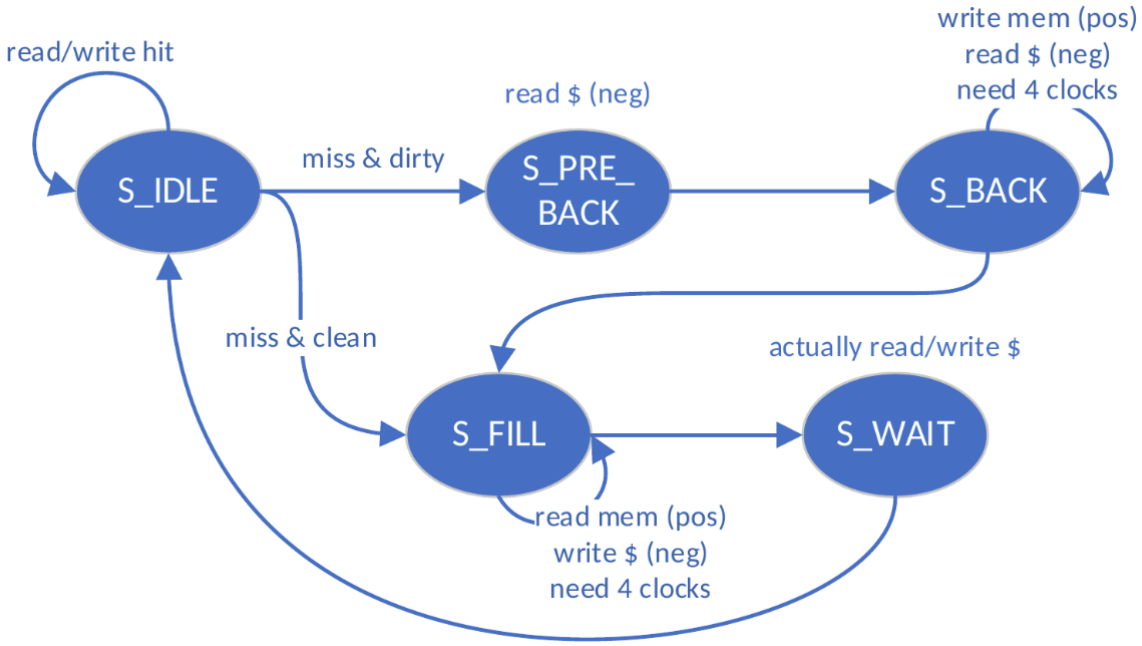
- **补全cache模块**

  地址的31-9为`tag`,8-4为`index`,由于是二路组相联,所以组里的第二个块地址就是组地址加上最后一位为1,一个字为四个字节,所以字地址就是块地址加上地址的3-2位。根据字地址得到缓存中的字,再根据地址的低两位得到对应的半字和字节。根据块地址找到对应块的`recent`,`valid`,`dirty`位,块合法且`tag`相同表示命中。`recent`为1表示最近被使用的,而下面这几个信号是要被替换的块的信号,所以`recent1`为1则要替换块2,也就是要用块2的信号,当块1或块2命中时表示命中。其余的就模仿他给的另一个例子就行了。

```
assign addr_tag = addr[31:9];              //need to fill in
assign addr_index = addr[8:4];            //need to fill in
assign addr_element2 = {addr_index, 1'b1};      //need to fill in
assign addr_word2 = {addr_element2, addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-
1:WORD_BYTES_WIDTH]};           //need to fill in
assign word2 = inner_data[addr_word2];                 //need to fill in
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];           //need to
fill in
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 recent2 = inner_recent[addr_element2];              //need to fill in
assign valid2 = inner_valid[addr_element2];               //need to fill in
assign dirty2 = inner_dirty[addr_element2];               //need to fill in
assign tag2 = inner_tag[addr_element2];                 //need to fill in
assign hit2 = valid2 & (tag2 == addr_tag);                 //need to fill in


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
```

- **补全cmu模块**

根据下方的状态图将代码补全即可,当`next_state`不是`S_IDLE`的时候,说明miss后,内存数据还没写回cache中,此时需要`stall`。



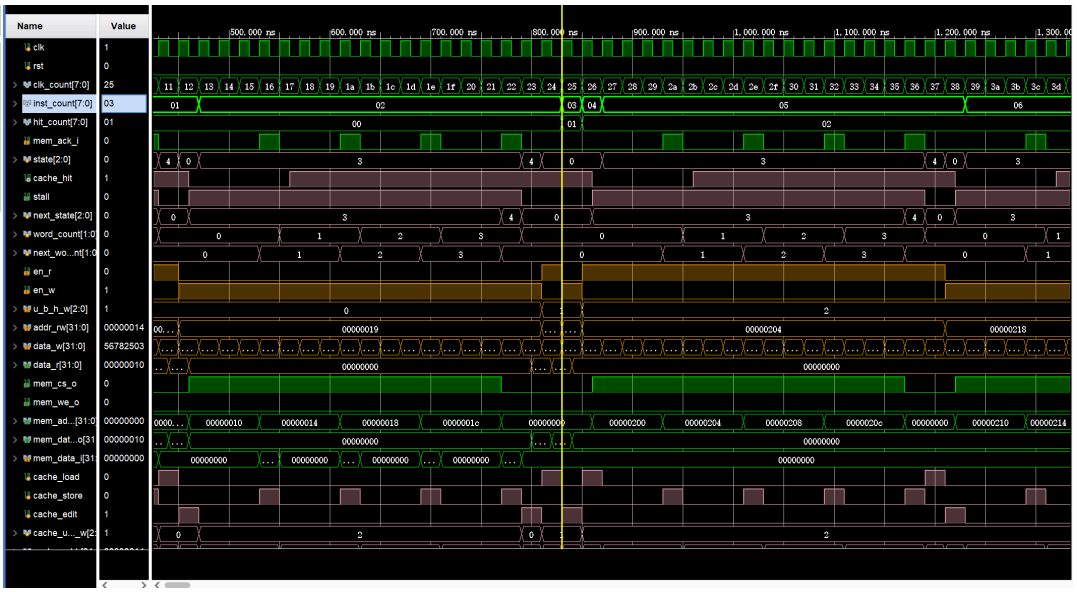

```
always @ (*) begin
        if (rst) begin
            next_state = S_IDLE;
            next_word_count = 2'b00;
        end
        else begin
            case (state)
                S_IDLE: begin
                    if (en_r || en_w) begin
                        if (cache_hit)
                            next_state = S_IDLE;
                        else if (cache_valid && cache_dirty)
                            next_state = S_PRE_BACK;
                        else
                            next_state = S_FILL;
                    end
                    next_word_count = 2'b00;
                end

                S_PRE_BACK: begin
                    next_state = S_BACK;
                    next_word_count = 2'b00;
                end

                S_BACK: begin
                    if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
 // wrote back all words, 1 cache line = 4 words
                        next_state = S_FILL;
                    else
                        next_state = S_BACK;

                    if (mem_ack_i)
                        next_word_count = word_count + 2'b01;
                    else
                        next_word_count = word_count;
                end
```

```
                S_FILL: begin
                    if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
                        next_state = S_WAIT;
                    else
                        next_state = S_FILL;

                    if (mem_ack_i)
                        next_word_count = word_count + 2'b01;
                    else
                        next_word_count = word_count;
                end

                S_WAIT: begin
                    next_state = S_IDLE;
                    next_word_count = 2'b00;
                end
            endcase
        end
    end

    // cache ctrl
    always @ (*) begin
        case(state)
            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: 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: 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
        endcase
    end


assign stall = next_state != S_IDLE ;
```

二、思考题

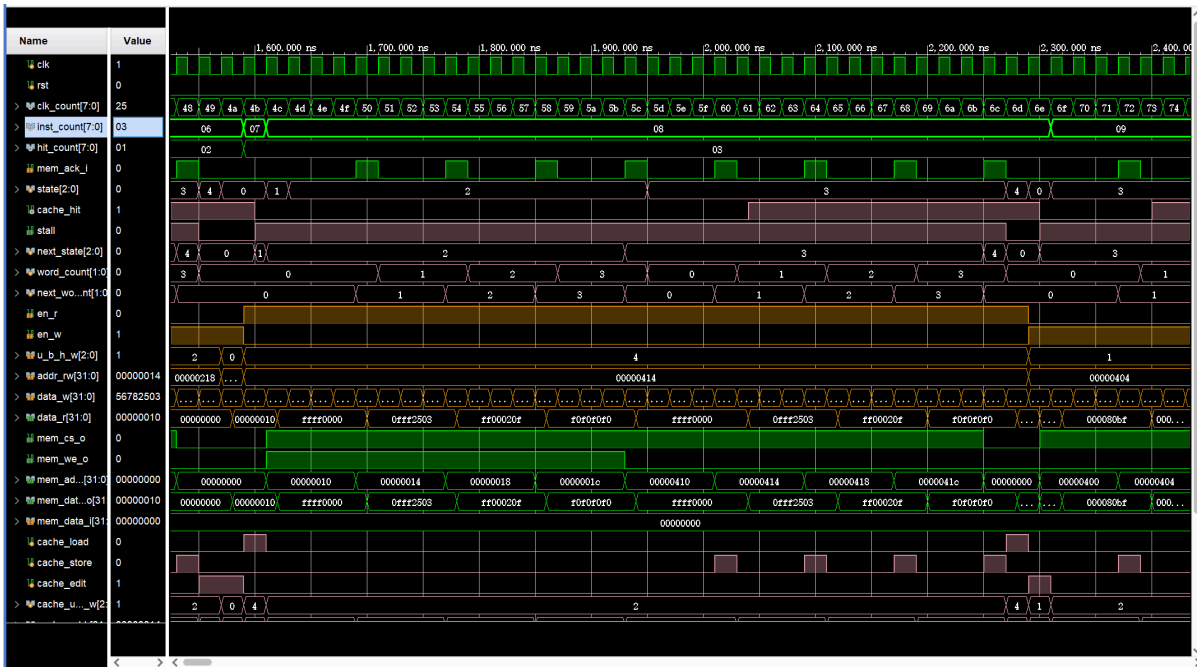
1. - **Cache hit**

     当 `inst_count` 为0x3时,可以看到 `hit_count` 由0变成1,表示命中,此时 `state` 一直为0,表示 `S_IDLE` 状态,只需要一个时钟周期。



   - **Cache miss+dirty**

     当 `inst_count` 为0x8时,可以看到 `hit_count` 没变,表示未命中,并且此时需要 `write_back` ,此时 `state` 变化依次为 `S_PRE_BACK` , `S_BACK` , `S_FILL` , `S_WAIT` , `S_IDLE` ,从图中 `state` 变化为1,2,3,4,0也说明了状态变化,所需要的时钟周期为 `S_PRE_BACK` 1个时钟周期, `S_BACK` 4X4=16个时钟周期, `S_FILL` 4X4=16个时钟周期, `S_WAIT` , `S_IDLE` 各1个时钟周期,一共需要35个时钟周期。



2. 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2bit来用于真正的LRU替换实现,因为 `inner_recent` 的每一位记录了每一个块最近是否被使用,而一个set有两个块,所以在替换的时候,只需要看这两个块对应的那两个位是否为1,为1的就是最近使用的,然后将另一个块替换。

三、感想

本次实验应该是目前最轻松的了,因为要填的空很多都给了提示或参考,只需要模仿就基本填完了,以至于写完以后有的地方还不是很理解就实现了,最后还是将不懂的地方给搞懂了。一下子就过了半个学期了,希望继续努力,顺利过完剩下的半个学期。