Lab 3 实验报告

实验过程及截图

阶段一

1. 增加维度

修改如下：

```
reg [            31:0] cache_mem    [SET_SIZE][WAY_CNT][LINE_SIZE];
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE][WAY_CNT];
reg                    valid        [SET_SIZE][WAY_CNT];
reg                    dirty        [SET_SIZE][WAY_CNT];
```

2. 增加并行判断

需要知道是否hit，并且知道hit的是哪一个。

组合逻辑中通过for循环的方式一一查找是否命中，综合后会生成一个并行判断的语句。

```
reg cache_hit = 0;
reg [WAY_CNT - 1:0] way_addr = 0;

// old
always @ (*) begin              // 判断 输入的address 是否在 cache 中命中
    if(valid[set_addr] && cache_tags[set_addr] == tag_addr)   // 如果 cache line有效，并且tag与输入地址中的tag相等，则命中
        cache_hit = 1'b1;
    else
        cache_hit = 1'b0;
end

// new
always @ * // cache_hit & way_addr
begin
    cache_hit = 0;
    for (integer i = 0; i < WAY_SIZE; i++)
    begin
        if (valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr)
        begin
            cache_hit = 1;
            way_addr = set_addr;
        end
    end
end
```

3. FIFO

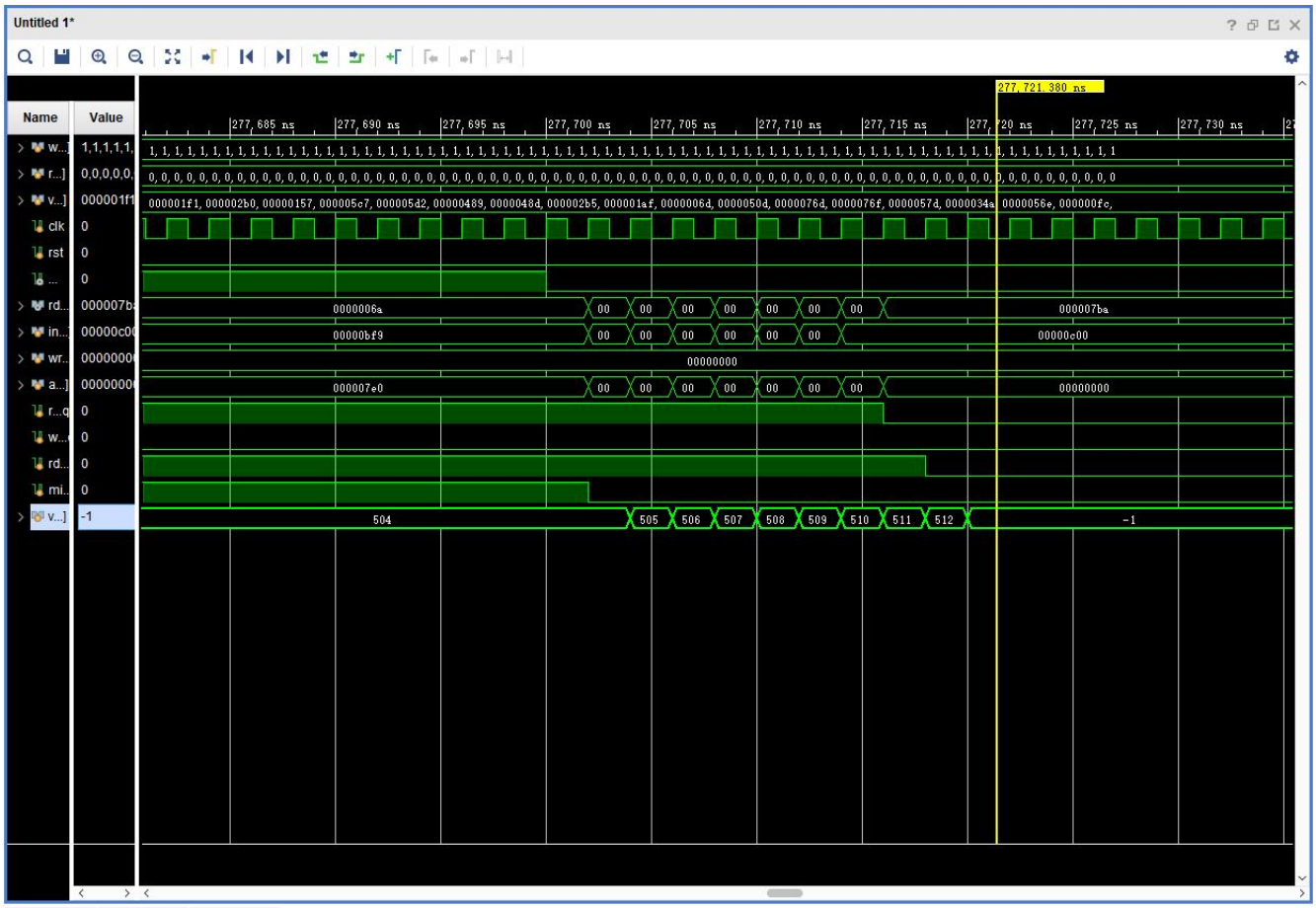
设寄存器：`reg [WAY_ADDR_LEN - 1:0] fifo_head[SET_SIZE]`

从而为每个set维护一个游标，初始值化为零。

每次需要换出这个块之后，对应SET的游标加一，从而下次优先换后面的块，这样循环前进，可以实现FIFO。

具体实现过程中只需要对每个状态内的语句均添加way_addr维度，并且换出块前加入`mem_rd_way_addr <= fifo_head[set_addr]`将参数传递到SWAP_IN_OK状态中。

截图：



最终变为-1，仿真成功。

4. LRU

加入变量`reg [WAY_CNT-1:0] lru [SET_SIZE];`为每个set中对应的位记录bit值从而实现伪LRU算法。在组合逻辑中加入统计0的个数和第一个不为0的bit的代码，如下：

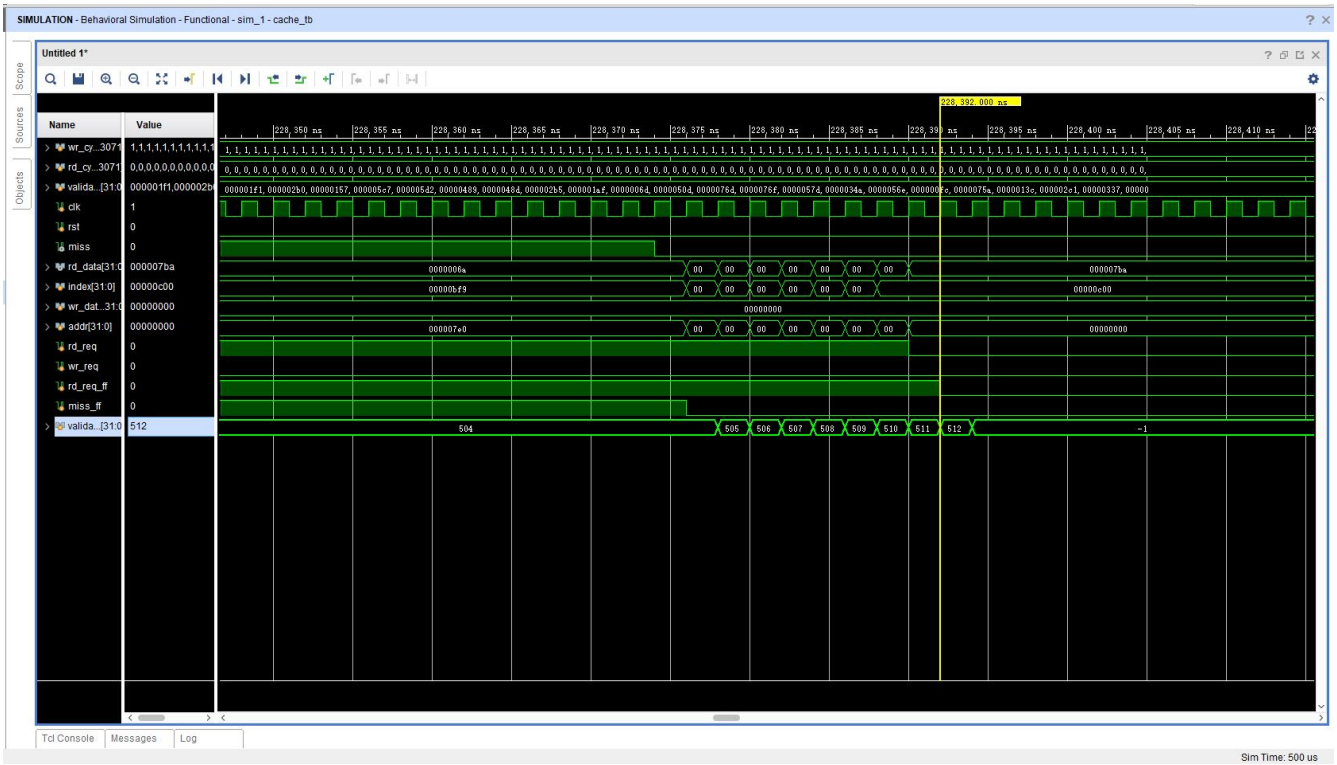
```
    // 寻找最小为0的line，并记录0的个数
    zero_count = 0;
    for (integer i = 0;i < WAY_CNT; i++)
    begin
        if (lru[set_addr][i] == 0)
        begin
            out_sel = i;
            zero_count = zero_count + 1;
        end
    end
```

out_sel即为换出块的选择，zero_count为选择的set中为0的bit的个数。

其余类似FIFO，但是还要在IDLE状态并且命中时加入判断1的个数是否为WAY_CNT-1的语句，如果为是，则需要将所有比特翻转，否则将当前读写位置bit置为1。

```
if (zero_count == 1 && lru[set_addr][way_addr] == 0)
for (integer i = 0; i < WAY_CNT; i++)
    lru[set_addr] <= ~lru[set_addr];
else
    lru[set_addr][way_addr] <= 1;
```

截图



阶段二

将WbData中之前的DataCache注释掉，然后实例化一个新的cache，改动部分如下：

```
    // lab3: add cache
    wire rd_req, wr_req;
    assign rd_req = (load_type == `NOREGWRITE) ? 1'b0 : 1'b1;
    assign wr_req = (write_en == 4'b0000) ? 1'b0 : 1'b1;

    cache #(
        .LINE_ADDR_LEN (3),
        .SET_ADDR_LEN  (3),
        .TAG_ADDR_LEN  (6),
        .WAY_CNT       (4)
    ) cache1 (
        .clk(clk),
        .rst(rst),
        .miss(miss),
```

```
        .addr(addr),
        .rd_req(rd_req),
        .rd_data(data_raw),
        .wr_req(wr_req),
        .wr_data(in_data)
    );
```

同时，需要把reset信号从top模块引入WbData.v，把miss信号一直引出到hazard中，并且生成对应的流水线停顿信号。

```
// hazard.v
        if (miss)
            begin
                flushF <= 0;
                flushD <= 0;
                flushE <= 0;
                flushM <= 0;
                flushW <= 0;
                bubbleF <= 1;
                bubbleD <= 1;
                bubbleE <= 1;
                bubbleM <= 1;
                bubbleW <= 1;
            end
```

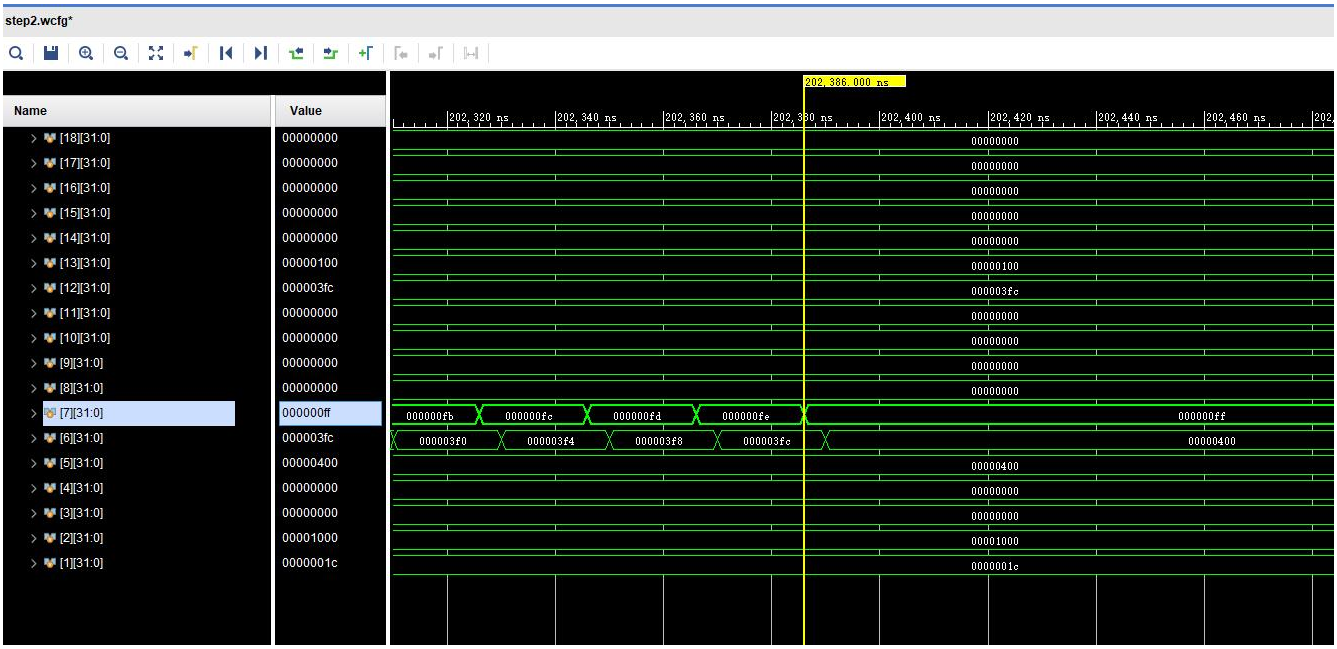
> 源代码中并未对阶段二创建两个工程，故如果想测试需要使用阶段一中对应的实现方式文件cache.sv进行替换。

DEBUG

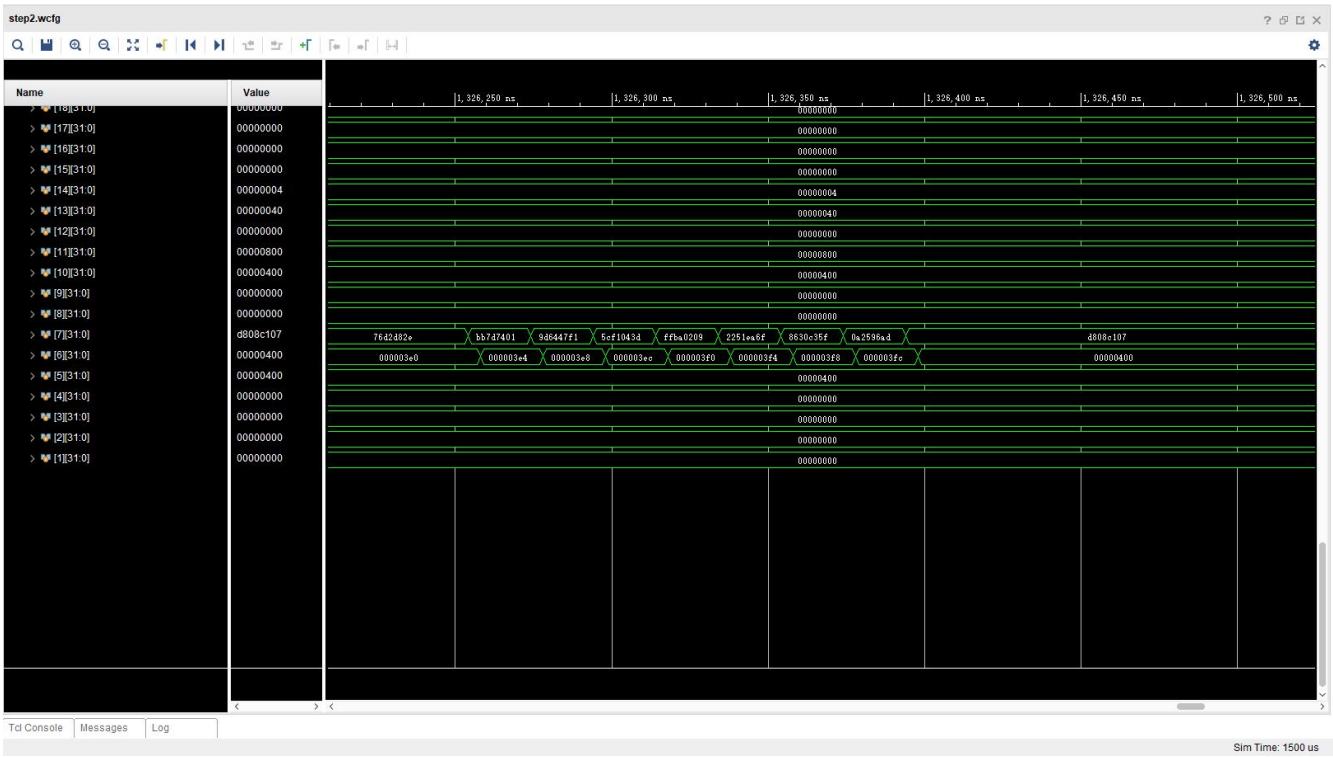
> **bug 1:**
>
> ```
> // WbData.v 上方代码块中Line 37 access_ff -> access; miss_ff -> miss
> if (access && !bubble_ff)
>     if (miss)
>         miss_count  <= miss_count + 1;
>     else
>         hit_count <= hit_count + 1;
> ```

截图

快速排序



矩阵相乘



结果分析

电路面积分析

固定SET_LEN = 3, TAG_LEN = 6

	组相联度	LUT	FF	BRAM	IO
FIFO	2	**2007**(3.17%)	**5175**(4.08%)	**4**(2.96%)	**81**(38.57%)
	4	**4464**(7.04%)	**9419**(7.43%)	**4**(2.96%)	**81**(38.57%)
	8	**7430**(11.72%)	**17877**(14.10%)	**4**(2.96%)	**81**(38.57%)
	16	**12690**(20.02%)	**34784**(27.43%)	**4**(2.96%)	**81**(38.57%)
LRU	2	**2139**(3.37%)	**5188**(4.09%)	**4**(2.96%)	**81**(38.57%)
	4	**4170**(6.58%)	**9427**(7.43%)	**4**(2.96%)	**81**(38.57%)
	8	**7632**(12.04%)	**17934**(14.14%)	**4**(2.96%)	**81**(38.57%)
	16	**13180**(20.79%)	**34928**(27.55%)	**4**(2.96%)	**81**(38.57%)

可以发现，电路面积与组相联度成正比关系，且FIFO和LRU算法使用资源的规模类似，LRU要稍微略大一点点，区别并不大。

固定组相联度为4，有：

> Bram和IO与上方一致，在此不列出了。

	SET_LEN	TAG_LEN	LUT	FF
FIFO	2	7	4066	5193
	3	6	4464	9149
	4	5	7726	17819
LRU	2	7	3992	5203
	3	6	4170	9427
	4	5	7911	17853

可以看出，在固定总场不变时，FF资源使用量和SET_LEN成正比，和TAG_LEN成反比。LUT使用量在SET_LEN较小时看不出正比关系，但是在较大时也与SET_LEN成正比趋势。

缺失率分析

快速排序

> 左侧为FIFO，右侧为LRU

组相联度	数据大小	命中次数	缺失次数	缺失率
2	256	6216 / 6222	172 / 167	2.69% / 2.61%
4	256	6304 / 6288	85 / 101	1.33% / 1.58%
8	256	6347 / 6347	42 / 42	0.66% / 0.66%
16	256	6347 / 6347	42 / 42	0.66% / 0.66%

FIFO和LRU表现类似，在组相联度为8以下之后，数据缺失率变化不明显，所以我认为取8的性价比较高。

矩阵相乘

组相联度	数据大小	命中次数	缺失次数	缺失率
2	16	3840 / 4010	4864 / 4694	55.9% / 46.1%
4	16	6965 / 7050	1739 / 1654	19.98% / 19.00%
8	16	8558 / 8560	146 / 144	1.67% / 1.65%
16	16	8608 / 8608	96 / 96	1.10% / 1.10%

和上面类似，在组相联度8-16的优化远远不如4-8的优化，故我认为在这组中还是取8更好。

总结

由缺失率的分析可以看出，将组相联度控制在8可以使得优化最有效，而SET_LEN和TAG_LEN在和一定（按照原来设定，为9）时，控制SET_LEN=3, TAG_LEN=6可以控制在LUT快速增长的起点，资源利用率较高。所以我认为对于我的测试，应当将组相联度控制在8附近，而在SET_LEN和TAG_LEN为9时，控制SET_LEN=3, TAG_LEN=6，可以有效利用资源。