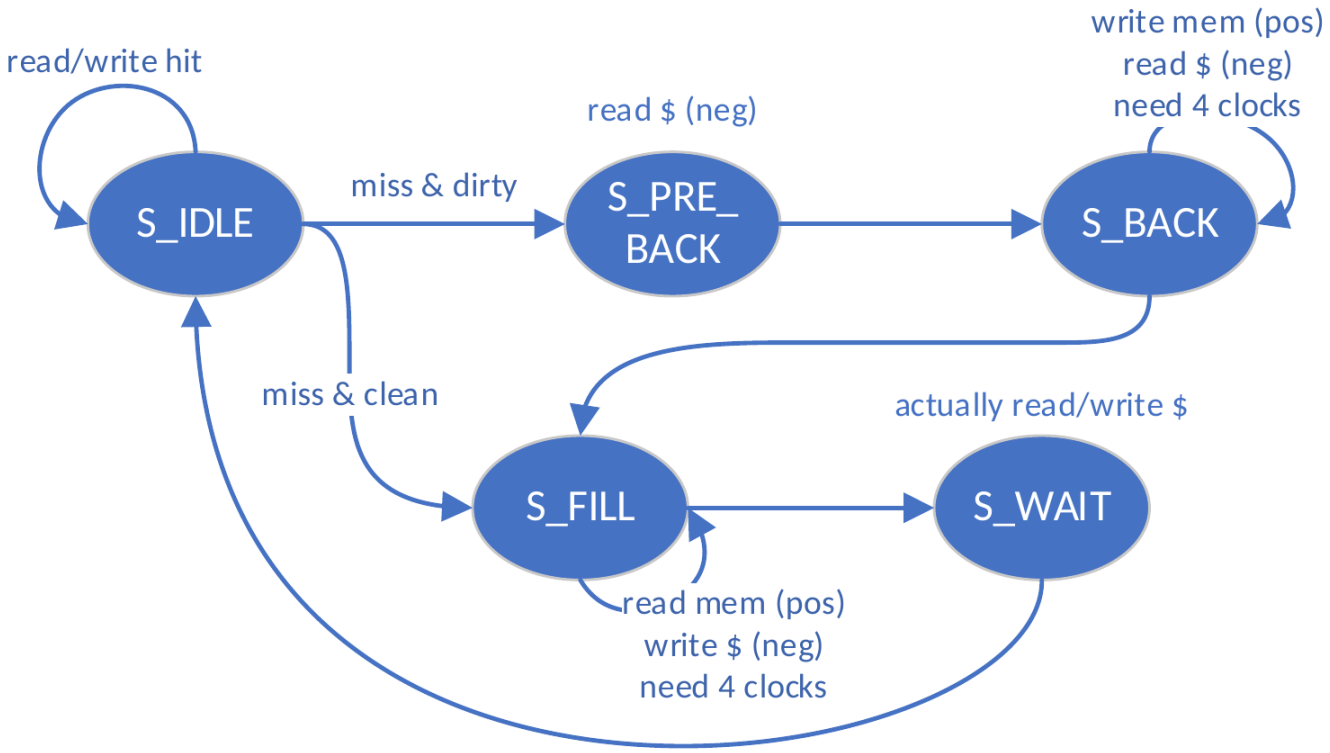
实验 4 - L1 cache设计

1 实验步骤

1.1 补全 CMU 模块

根据实验指导，CMU 模块关键在于完成状态机的填写，根据指导里的图



补全后的结果如下：

```
    // state ctrl
    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}})
// 2'b11 in default case
                        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
```

还剩下 `stall` 信号的输出，这里我们直接判断 `next_state` 和 `S_IDLE` 是否相等，如果相等则说明需要 stall。

```
    assign stall = next_state != S_IDLE;
```

1.2 补全 Cache 模块

本次实验要求 cache 实现 2 路组相联，采用 write-back,write-allocate 的数据更新策略，采用 LRU 的替换策略。

- 根据 `addr` 的划分，我们可以得到 `tag` 和 `index` 的部分：

```
    assign addr_tag = addr[31:9];
    assign addr_index = addr[8:4];
```

- `addr_element2` 表示第二路的块地址。

```
addr_element2 = {addr_index, 1'b1};
```

- `addr_word2` 表示第二路的字对应在 `inner_data` 中的索引。

```
assign addr_word2 = {addr_element2,
addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]};
```

- `word2, half_word2, byte2` 分别表示第二路的字、半字、字节，直接照搬第一路的写法即可。
- `recent2, valid2, dirty2, tag2, hit2` 分别表示第二路的最近访问位、有效位、脏位、标记位、命中位，直接照搬第一路的写法即可。
- `valid, dirty, tag, hit`：这里 `valid` 表示当前要写入的位置是否被用，判断方法是看 `recent1` 是否为 1，为 1 说明第一路最近被用过，则使用第二路，否则使用第一路；`dirty, tag` 的思路类似。`hit` 表示本次访问是否命中，那么两路只要有一个命中了 `hit` 就为 1。

```
    valid <= recent1 ? valid2 : valid1;
    dirty <= recent1 ? dirty2 : dirty1;
    tag <= recent1 ? tag2 : tag1;
    hit <= hit1 | hit2;
```

- `load, edit` 的情况处理类似，直接照搬第一路的写法即可。
- `store` 的情况略有不同，注释里要求根据 `recent2` 值分类讨论，但实际上这两种情况可以写在一起，直接照搬第一路的写法即可。

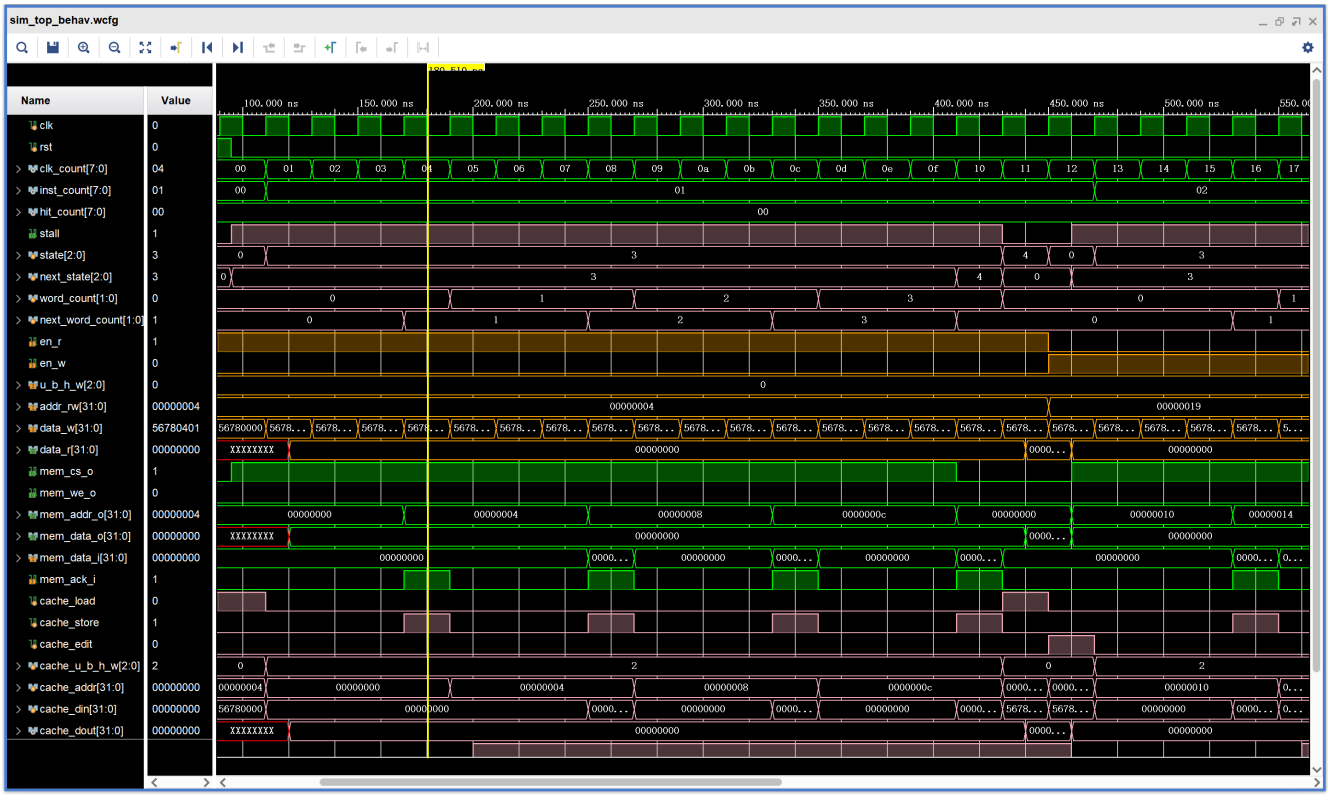
```
    inner_data[addr_word1] <= din;
    inner_valid[addr_element1] <= 1'b1;
    inner_dirty[addr_element1] <= 1'b0;
    inner_tag[addr_element1] <= addr_tag;
```

2 实验评估

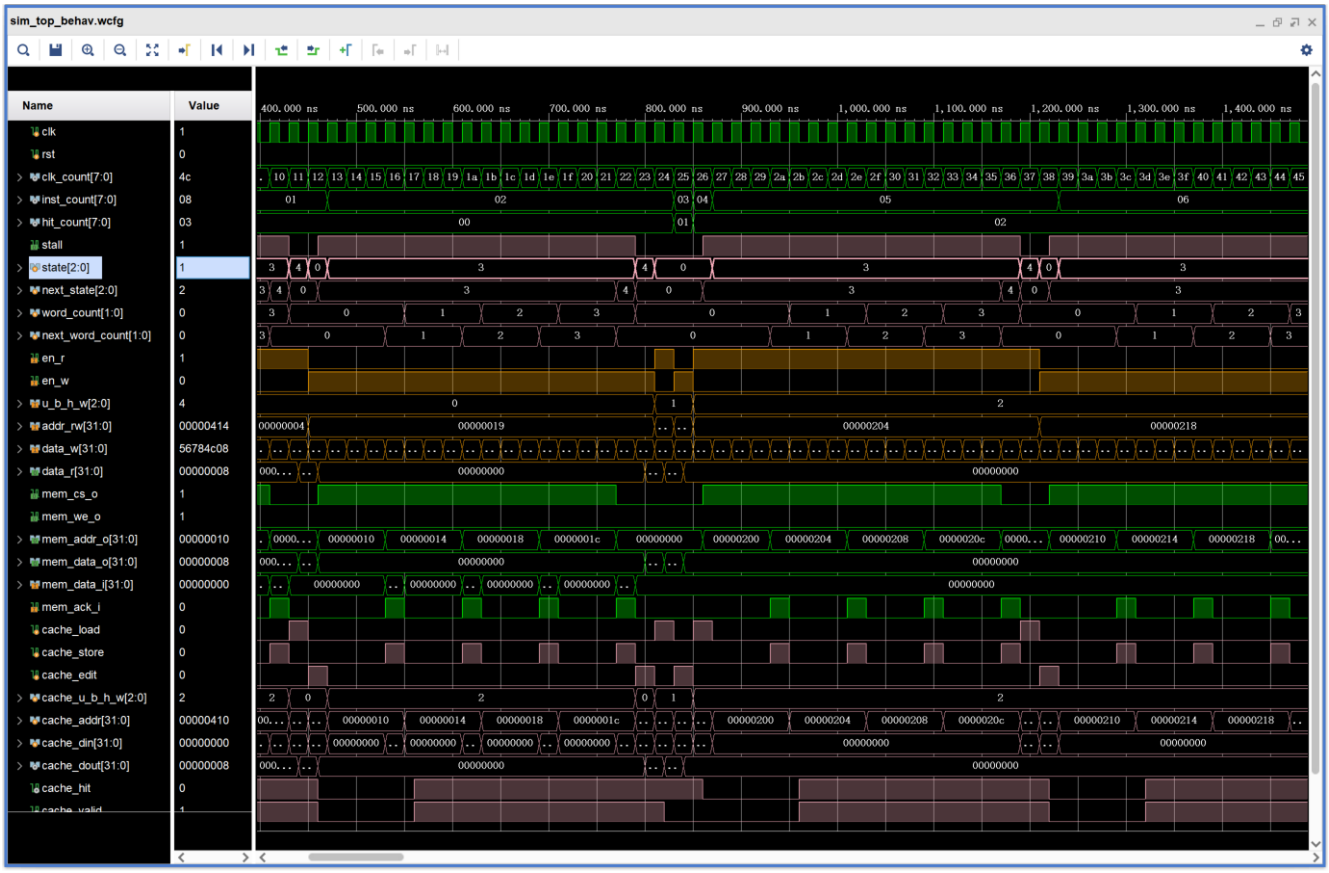
2.1 仿真

2.1.1 对 Cache & CMU 模块进行仿真

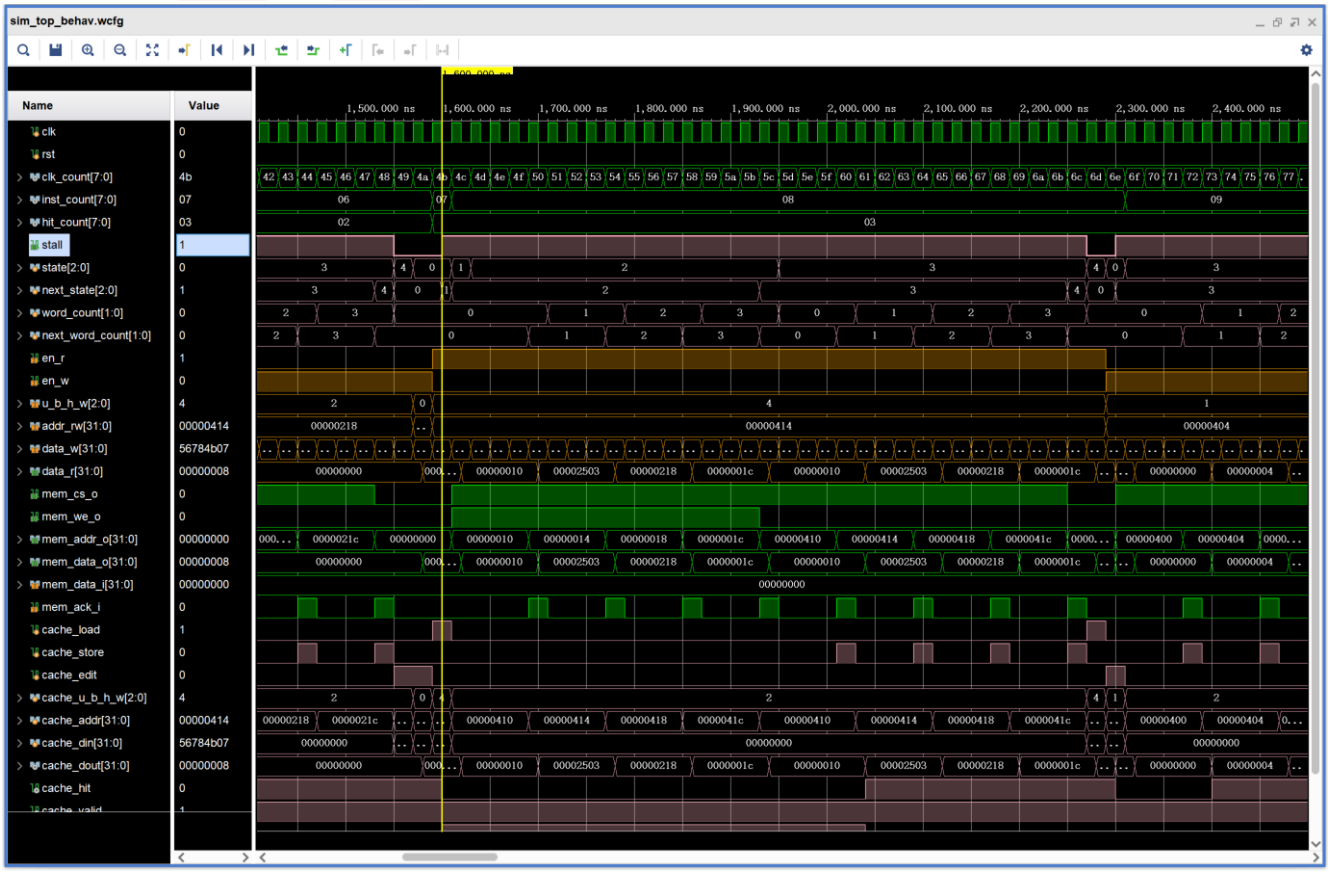
0x4 的是一条 load 指令，此时 cache 为空会发生 read miss，因此可以看到 `mem_addr_o` 从 0x0，0x4，0x8，0xC 依次变化（对应 `cache_store=1`），把对应的数据读入到缓存中。这里可以看到 `S_FILL` 阶段需要 4*4=16 个周期。（可以看到 `clk_count` 从 0 变为了 0x10）



第三、第四条指令时，cache hit，830ns、850ns 可以看到一次是 read hit，一次是 write hit，均各只用了一个周期。

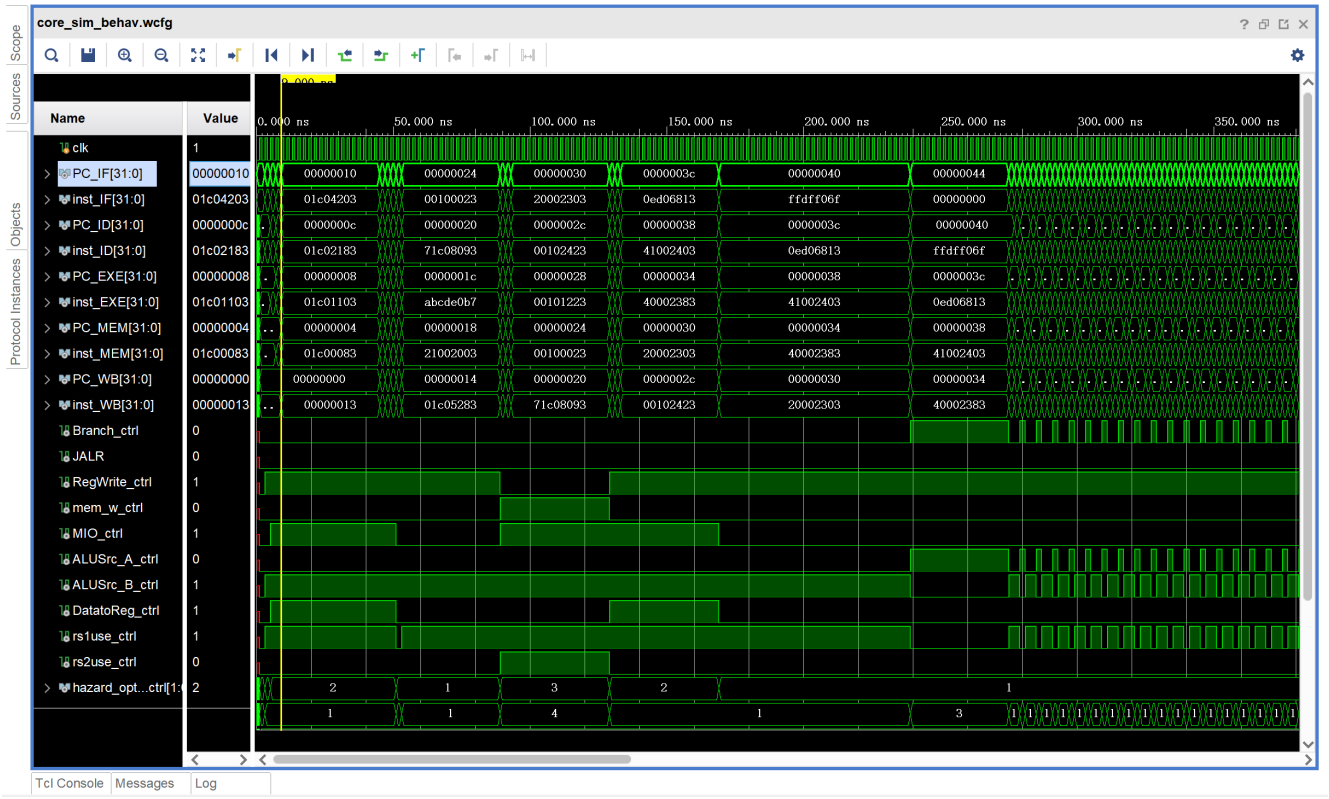


第八条指令时，会发生 miss 而且需要 write back。可以看到 1610ns 时状态从 0 变成了 1（即 `S_PRE_BACK`），下个周期变成了 2（即 `S_BACK`），随后先写回缓存数据，再进入 `S_FILL` 状态写入新数据。这里可以看到 `S_BACK` 和 `S_FILL` 阶段各需要 4*4=16 个周期。



2.1.2 对 Core 进行仿真

对 core 进行仿真可以看到，前四次 cache miss 时只需要将数据读到 Cache 里即可，只需要 18 个周期。（4*4（`S_FILL`）+ 1（`S_WAIT`）+ 1（`S_IDLE`）= 18）第五次 cache miss 需要先把脏数据写回内存，因此需要 35 个周期。（1（`S_PRE_BACK`）+ 4*4（`S_BACK`）+ 4*4（`S_FILL`）+ 1（`S_WAIT`）+ 1（`S_IDLE`）= 35）



2.2 上板结果

验收已通过，此处略去。

3 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。

   见仿真波形分析，分别展示了 cache miss，miss & dirty，hit 的波形。可以看到缓存命中则一个周期即可完成，而缓存不命中则需要 16/35 个周期（取决于是否有脏数据需要写回）。

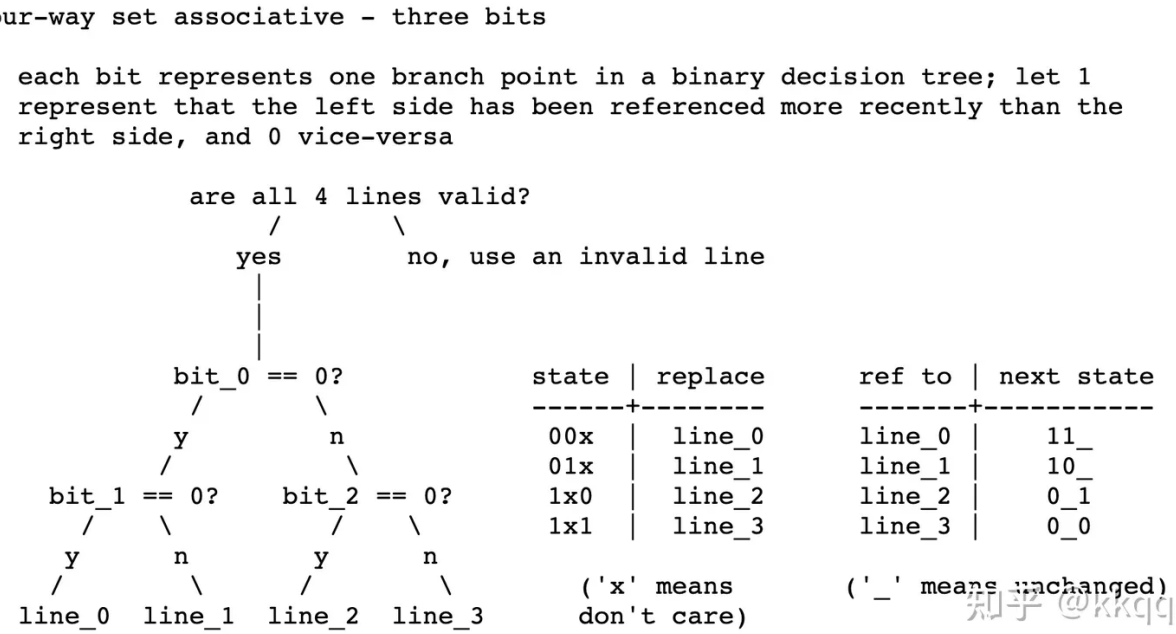
2. 在本次实验中，cache采取的是2路组相联，在实现LRU替换的时候，每一个set需要用多少bit来用于真正的LRU替换实现？

   2 bit，将最晚访问的块对应的位设为 1，例如初始状态为 00，最近访问了第一路则变为 01，最近访问了第二路则变为 10，相当于实验中的 `recent1, recent2`。每次替换 `recent=0` 对应的块即可。

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。

   真 LRU：我们用上课讲的栈的方式来模拟，即栈有 4 个元素，每个元素是一个数字，唯一表示当前的块编号。则编号需要 2 位，因此栈总共有 8 位。每次访问我们就将对应的块编号放在栈底，需要替换时就将栈顶元素替换掉即可。

Pseudo-LRU：一个 set 里有 w 个 cache line，tree-PLRU 会使用 w-1 位来表示近似访问历史顺序的二叉树。tree-PLRU 将这 w 个 cache line 划分成不同的区块，并用 0/1 表示区块的访问时间的远近，如果该 bit 位为 1，说明最近访问的是左边的区块，反之最近访问的是右边的区块。



因此 4 路组相联需要 3 个 bit 来表示近似访问历史顺序的二叉树。