

计算机体系结构 lab3 实验报告

肖桐 PB18000037

1. FIFO 多路组相联 Cache

假设组相联数为 `WAY_CNT`。

相比全相联的 Cache，不同点在于每个组的 line 个数多于 1。因此相应的 `cache_mem`、`cache_tag`、`dirty`、`valid` 域都得增加一个维度，大小为 `WAY_CNT`。同时对于每一个 line 需要一个 `fifo` 数组用来记录入队出队信息，用于 `cache` 替换。

因此有：

```
1 reg [31:0] cache_mem [SET_SIZE][WAY_CNT]
  [LINE_SIZE]; // SET_SIZE个line, 每个line有LINE_SIZE个word
2 reg [TAG_ADDR_LEN-1:0] cache_tags [SET_SIZE][WAY_CNT];
  // SET_SIZE个TAG
3 reg valid [SET_SIZE][WAY_CNT];
  // SET_SIZE个valid(有效位)
4 reg dirty [SET_SIZE][WAY_CNT];
  // SET_SIZE个dirty(脏位)
5 reg [31:0] fifo [SET_SIZE][WAY_CNT];
  // 记录 fifo 信息
```

而在对某一个 set 进行访问时，因为对于多路 Cache 而言，在同一个 set 内的 line 无法通过地址值确定，因此需要对所有的 line 都遍历一遍，比较每个 line 的 tag 域。然后返回命中的 line。

实现为：

```
1 reg cache_hit = 1'b0;
2 always @ (*) begin // 判断 输入的address 是否在 cache
  命中
3   for (integer i = 0; i < WAY_CNT; i++) begin
4     if (valid[set_addr][i] == 1'b1 && cache_tags[set_addr]
  [i] == tag_addr) begin
5       cache_hit = 1'b1;
```

```

6         hit_way = i;
7         break;
8     end
9     else begin
10         cache_hit = 1'b0;
11         hit_way = 0;
12     end
13 end
14 end

```

这里 `hit_way` 整数记录命中的 line 编号，用于从 Cache 中返回读取的字、或者向 Cache 中写入字：

```

1     case(cache_stat)
2     IDLE:     begin
3         if(cache_hit) begin
4             if(rd_req) begin // 如果cache命中，
// 并且是读请求，
5                 rd_data <= cache_mem[set_addr]
[hit_way][line_addr]; //则直接从cache中取出要读的数据
6             end else if(wr_req) begin // 如果
// cache命中，并且是写请求，
7                 cache_mem[set_addr][hit_way]
[line_addr] <= wr_data; // 则直接向cache中写入数据
8                 dirty[set_addr][hit_way] <=
1'b1; // 写数据的同时置脏位
9             end
10        end

```

对于需要换入换出的情况，选取对应 set 中 fifo 值最小的 line 进行换入，同时对所有的 line，将 fifo 值减一。换入之后再换入的 line 的 fifo 值置为最大 `WAY_CNT`。如下所示：

```

1         else begin
2             if(wr_req | rd_req) begin // 如果
// cache 未命中，并且有读写请求，则需要换入 // 上一个周期换入成
// 功，这周期将主存读出的line写入cache，并更新tag，置高valid，置低dirty
3                 static integer min = WAY_CNT;
// 找出 fifo 最小的 且 不脏的 line 换入
4                 selected_way = hit_way;
// 若没有满足要求的 line，则将换入到 hit_way 自身

```

```

5           for (integer i = 0; i < WAY_CNT;
i++) begin
6           if (valid[set_addr][i] ==
1'b0) begin
7           selected_way = i;
8           break;
9           end
10          if (min > fifo[set_addr][i])
begin
11          min = fifo[set_addr][i];
12          selected_way = i;
13          end
14          fifo[set_addr][i] <=
fifo[set_addr][i] - 1; //队列递减
15          end

```

这里 `selected_way` 用于记录要被换入的 line 的编号，在 `SWAP_IN_OK` 状态下再实质上将来自 `mem` 的数据写入 `cache` 的该 line 中：

```

1      SWAP_IN_OK: begin
2          for(integer i=0; i<LINE_SIZE; i++)
cache_mem[mem_rd_set_addr][selected_way][i] <= mem_rd_line[i];
3          cache_tags[mem_rd_set_addr][selected_way]
<= mem_rd_tag_addr;
4          valid [mem_rd_set_addr][selected_way]
<= 1'b1;
5          dirty [mem_rd_set_addr][selected_way]
<= 1'b0;
6          fifo [mem_rd_set_addr][selected_way]
<= WAY_CNT; //置为最大
7          cache_stat <= IDLE; // 回到就绪状态
8          end

```

换入之后同时将 `dirty` 置 0，`valid` 置 1。

2. LRU 多路组相联 Cache

LRU Cache 相比 FIFO Cache 只有换入换出时选择换入 line 的方式不同。LRU Cache 维护一个 `lru` 数组，与 FIFO Cache 目的类似，但是是在选择换入的 line 时选择 `lru` 值最小的 line。

同时每次 `cache_hit` 时都需要将 `hit` 的 `line` 的 `lru` 值重置为 0，以表示该 `line` 最近被使用过。同时需要将所有的其他 `line` 的 `lru` 值加一。

实现如下：

```
1      case(cache_stat)
2      IDLE:      begin
3                  if(cache_hit) begin
4                      for (integer i = 0; i < WAY_CNT;
5 i++) begin
6                          lru[set_addr][i] = lru[set_addr]
7 [i] + 1;
8                      end
9                      lru[set_addr][hit_way] = 0;      //最近使用，lru 置为 0
10                     if(rd_req) begin      // 如果cache命中，
11                         rd_data <= cache_mem[set_addr]
12 [hit_way][line_addr];      //则直接从cache中取出要读的数据
13                     end else if(wr_req) begin // 如果
14                         cache_mem[set_addr][hit_way]
15 [line_addr] <= wr_data;      // 则直接向cache中写入数据
16                         dirty[set_addr][hit_way] <=
17 1'b1;      // 写数据的同时置脏位
18                     end
19                 end
20             end
```

而在选择换入得块时与 FIFO Cache 操作是相同的，只不过此时是选择 `lru` 值最大的 `line`，实现如下：

```
1      else begin
2          if(wr_req | rd_req) begin      // 如果
3              cache 未命中，并且有读写请求，则需要进行换入      // 上一个周期换入成
4              功，这周期将主存读出的line写入cache，并更新tag，置高valid，置低dirty
5              static integer max = 0;      //
6              找出 lru 最大的 且 不脏的 line 换入
7              selected_way = hit_way;      //
8              若没有满足要求的 line，则将换入到 hit_way 自身
9              for (integer i = 0; i < WAY_CNT;
10 i++) begin
```

```

6             if (valid[set_addr][i] ==
1'b0) begin
7                 selected_way = i;
8                 break;
9             end
10            if (max < lru[set_addr][i])
begin
11                max = lru[set_addr][i];
12                selected_way = i;
13            end
14        end

```

这里 `selected_way` 用于记录要被换入的 line 的编号，在 `SWAP_IN_OK` 状态下再实质上将来自 `mem` 的数据写入 `cache` 的该 line 中：

```

1    SWAP_IN_OK: begin
2        for(integer i=0; i<LINE_SIZE; i++)
cache_mem[mem_rd_set_addr][selected_way][i] <= mem_rd_line[i];
3        cache_tags[mem_rd_set_addr][selected_way]
<= mem_rd_tag_addr;
4        valid    [mem_rd_set_addr][selected_way]
<= 1'b1;
5        dirty    [mem_rd_set_addr][selected_way]
<= 1'b0;
6        lru      [mem_rd_set_addr][selected_way]
<= 0; //置为最小
7        cache_stat <= IDLE; // 回到就绪状态
8    end

```

换入之后同时将 `dirty` 置 0，`valid` 置 1，同时将 `lru` 值置 0。

3. 接入 CPU 后执行结果

选取 `cache` 大小为：

```

1    my_cache_FIFO #(
2        .LINE_ADDR_LEN ( 2 ),
3        .SET_ADDR_LEN   ( 3 ),
4        .TAG_ADDR_LEN   ( 5 ),
5        .WAY_CNT         ( 4 )
6    ) DCacheInst

```

```

7      (
8          .clk(clk),
9          .rst(rst),
10         .miss(DCacheMiss),
11         .addr(A),
12         .rd_req(MemReadM),
13         .rd_data(RD_raw),
14         .wr_req(|WE),
15         .wr_data(WD)
16     );

```

保证 `mem` 中有 1024 words，能够存下所有的矩阵元素和快排元素。

(1). 矩阵乘法

执行 16 阶矩阵相乘，即每个矩阵的大小为 256 word。

(a). FIFO Cache

执行结果为：

initial begin	
// dst matrix C	
ram_cell[0]	= 32'h0; // 32'hd91d1d3c;
ram_cell[1]	= 32'h0; // 32'h20081f05;
ram_cell[2]	= 32'h0; // 32'hcf071f74;
ram_cell[3]	= 32'h0; // 32'h0720ca21;
ram_cell[4]	= 32'h0; // 32'h4fea5bae;
ram_cell[5]	= 32'h0; // 32'hc6ac9ece;
ram_cell[6]	= 32'h0; // 32'hac85927e;
ram_cell[7]	= 32'h0; // 32'h28100cca;
ram_cell[8]	= 32'h0; // 32'h586cb62c;
ram_cell[9]	= 32'h0; // 32'h19666a50;
ram_cell[10]	= 32'h0; // 32'h2492b4cb;
ram_cell[11]	= 32'h0; // 32'hf5b8296c;
ram_cell[12]	= 32'h0; // 32'h95c0ef52;
ram_cell[13]	= 32'h0; // 32'hc6f00229;
ram_cell[14]	= 32'h0; // 32'he3978956;
ram_cell[15]	= 32'h0; // 32'h6a395b34;
ram_cell[16]	= 32'h0; // 32'hcd68fd38;
ram_cell[17]	= 32'h0; // 32'ha48a9820;
ram_cell[18]	= 32'h0; // 32'hde77a8e0;
ram_cell[19]	= 32'h0; // 32'heac27f18;
ram_cell[20]	= 32'h0; // 32'hdcad4fd4;
ram_cell[21]	= 32'h0; // 32'hc241bc4f;
ram_cell[22]	= 32'h0; // 32'he2a331cc;
ram_cell[23]	= 32'h0; // 32'h345c805d;
ram_cell[24]	= 32'h0; // 32'hea809987;
ram_cell[25]	= 32'h0; // 32'h94366e8b;
ram_cell[26]	= 32'h0; // 32'h211f149d;
ram_cell[27]	= 32'h0; // 32'h1003b418;
ram_cell[28]	= 32'h0; // 32'h9564b871;
ram_cell[29]	= 32'h0; // 32'hd86649be;

> mem_count[31:0]	8704
> miss_count[31:0]	3535
> hit_count[31:0]	5169

左上为实际运行结果，右边是预期运行结果。下面是该次运行的 cache miss 和 cache hit 次数。

其中：mem_count 是一共的访存次数，miss_count 为 cache miss 次数，hit_count 为 cache hit 次数。

(b). LRU Cache

执行结果为：

ram_cell[0:1023][31:0]	d91d1d3c,20081f	initial begin
> [2][31:0]	cf071f74	// dst matrix C
> [3][31:0]	0720ca21	ram_cell[0] = 32'h0; // 32'hd91d1d3c;
> [4][31:0]	4fea5bae	ram_cell[1] = 32'h0; // 32'h20081f05;
> [5][31:0]	c6ac9ece	ram_cell[2] = 32'h0; // 32'hcf071f74;
> [6][31:0]	ac85927e	ram_cell[3] = 32'h0; // 32'h0720ca21;
> [7][31:0]	28100cca	ram_cell[4] = 32'h0; // 32'h4fea5bae;
> [8][31:0]	586cb62c	ram_cell[5] = 32'h0; // 32'hc6ac9ece;
> [9][31:0]	19666a50	ram_cell[6] = 32'h0; // 32'hac85927e;
> [10][31:0]	2492b4cb	ram_cell[7] = 32'h0; // 32'h28100cca;
> [11][31:0]	f5b8296c	ram_cell[8] = 32'h0; // 32'h586cb62c;
> [12][31:0]	95c0ef52	ram_cell[9] = 32'h0; // 32'h19666a50;
> [13][31:0]	c6f00229	ram_cell[10] = 32'h0; // 32'h2492b4cb;
> [14][31:0]	e3978956	ram_cell[11] = 32'h0; // 32'hf5b8296c;
> [15][31:0]	6a395b34	ram_cell[12] = 32'h0; // 32'h95c0ef52;
> [16][31:0]	cd68fd38	ram_cell[13] = 32'h0; // 32'hc6f00229;
> [17][31:0]	a48a9820	ram_cell[14] = 32'h0; // 32'he3978956;
> [18][31:0]	de77a8e0	ram_cell[15] = 32'h0; // 32'h6a395b34;
> [19][31:0]	eac27f18	ram_cell[16] = 32'h0; // 32'hcd68fd38;
> [20][31:0]	dcad4fd4	ram_cell[17] = 32'h0; // 32'ha48a9820;
> [21][31:0]	c241bc4f	ram_cell[18] = 32'h0; // 32'hde77a8e0;
> [22][31:0]	e2a331cc	ram_cell[19] = 32'h0; // 32'heac27f18;
> [23][31:0]	345c805d	ram_cell[20] = 32'h0; // 32'hdcad4fd4;
> [24][31:0]	ea809987	ram_cell[21] = 32'h0; // 32'hc241bc4f;
		ram_cell[22] = 32'h0; // 32'he2a331cc;
		ram_cell[23] = 32'h0; // 32'h345c805d;
		ram_cell[24] = 32'h0; // 32'hea809987;
		ram_cell[25] = 32'h0; // 32'h94366e8b;
		ram_cell[26] = 32'h0; // 32'h211f149d;
		ram_cell[27] = 32'h0; // 32'h1003b418;
		ram_cell[28] = 32'h0; // 32'h9564b871;
		ram_cell[29] = 32'h0; // 32'hd86649be;

> mem_count[31:0]	8704
> miss_count[31:0]	3595
> hit_count[31:0]	5109

左上为实际运行结果，右边是预期运行结果。下面是该次运行的 cache miss 和 cache hit 次数。

可见运行结果都满足预期。

执行 256 words 的快速排序。

执行结果为:

▼ ram_cell[0:1023][31:0]	00000000,00000000	> [233][31:0]	0000000b
> [0][31:0]	00000000	> [234][31:0]	0000000c9
> [1][31:0]	000000001	> [235][31:0]	00000005f
> [2][31:0]	000000002	> [236][31:0]	0000000c3
> [3][31:0]	000000003	> [237][31:0]	00000007
> [4][31:0]	000000004	> [238][31:0]	00000005a
> [5][31:0]	000000005	> [239][31:0]	00000007
> [6][31:0]	000000006	> [240][31:0]	0000000f0
> [7][31:0]	000000007	> [241][31:0]	0000000f1
> [8][31:0]	000000008	> [242][31:0]	0000000f2
> [9][31:0]	000000009	> [243][31:0]	0000000f3
> [10][31:0]	00000000a	> [244][31:0]	0000000f4
> [11][31:0]	00000000b	> [245][31:0]	0000000f5
> [12][31:0]	00000000c	> [246][31:0]	0000000f6
> [13][31:0]	00000000d	> [247][31:0]	0000000f7
> [14][31:0]	00000000e	> [248][31:0]	0000000f8
> [15][31:0]	00000000f	> [249][31:0]	0000000f9
> [16][31:0]	000000010	> [250][31:0]	0000000fa
> [17][31:0]	000000011	> [251][31:0]	0000000fb
> [18][31:0]	000000012	> [252][31:0]	0000000fc
> [19][31:0]	000000013	> [253][31:0]	0000000fd
> [20][31:0]	000000069	> [254][31:0]	0000000fe
> [21][31:0]	0000000b3	> [255][31:0]	0000000ff
> [22][31:0]	000000086		

cache_mem[0.7][0.3][0.3][31.0]	'(000000bd,000000be,000000bf,000000c0),(00000020,00000021,00000022,00000023),(000000df,000000e0,000000e1,000000e2),(00000040,00000041,00000042,00000043),'
> [0][0.3][0.3][31.0]	'(000000bd,000000be,000000bf,000000c0),(00000020,00000021,00000022,00000023),(000000df,000000e0,000000e1,000000e2),(00000040,00000041,00000042,00000043),'
> [1][0.3][0.3][31.0]	'(000000c1,000000c2,000000c3,000000c4),(00000024,00000025,00000026,00000027),(0000000c,000000e4,000000e5,000000e6),(00000044,00000045,00000046,00000047),'
> [2][0.3][0.3][31.0]	'(000000c5,000000c6,000000c7,000000c8),(000000e7,000000e8,000000e9,000000ea),(00000028,00000029,0000002a,0000002b),(00000048,00000049,0000004a,0000004b),'
> [3][0.3][0.3][31.0]	'(0000000a,0000000b,0000000c,0000000d),(000000eb,000000ec,000000ed,000000ef),(00000002,00000002,00000002e,0000002f),(00000004,00000004d,00000004e,0000004f),'
> [4][0.3][0.3][31.0]	'(000000f0,000000f1,000000f3),(0000000c,0000000d),(00000003,00000003,00000003),(00000003,00000003,00000003),(00000005,00000005,00000005),'
> [5][0.3][0.3][31.0]	'(000000f4,000000f5,000000f6,000000f7),(00000014,00000015,00000016,00000017),(00000034,00000035,00000036,00000037),(000000d2,000000d3,000000d4,000000d5),'
> [6][0.3][0.3][31.0]	'(000000f8,000000f9,000000fa,000000fb),(00000018,00000019,0000001a,0000001b),(00000038,00000039,0000003a,0000003b),(000000d7,000000d8,000000d9,000000da),'
> [7][0.3][0.3][31.0]	'(000000fc,000000fd,000000fe,000000ff),(0000001c,0000001d,0000001e,0000001f),(0000003c,0000003d,0000003e,0000003f),(000000db,000000dc,000000dd,000000de),'
> cache_tags[0.7][0.3][4.0]	'{06,01,07,02},{06,01,07,02},{06,07,01,02},{06,07,01,02},{07,06,01,02},{07,00,01,06},{07,00,01,06},{07,00,01,06},'
> valid[0.7][0.3]	'{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},'
> dirty[0.7][0.3]	'{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},'

上图为局部运行结果，下图为运行结束之后 **Cache** 中的值。易看出运行结果中局部无序的数据都在 **cache** 中。

> mem_count[31:0]	6291
> miss_count[31:0]	438
> hit_count[31:0]	5853

上图为缺失率统计信息。

(b). LRU Cache

执行结果如下：

ram_cell[0:1023][31:0]		00000000	> [233][31:0]	000000cb
> [0][31:0]		00000000	> [234][31:0]	000000c9
> [1][31:0]		00000001	> [235][31:0]	00000051
> [2][31:0]		00000002	> [236][31:0]	000000c3
> [3][31:0]		00000003	> [237][31:0]	000000c7
> [4][31:0]		00000004	> [238][31:0]	0000005a
> [5][31:0]		00000005	> [239][31:0]	000000f7
> [6][31:0]		00000006	> [240][31:0]	000000f0
> [7][31:0]		00000007	> [241][31:0]	000000f1
> [8][31:0]		00000008	> [242][31:0]	000000f2
> [9][31:0]		00000009	> [243][31:0]	000000f3
> [10][31:0]		0000000a	> [244][31:0]	000000f4
> [11][31:0]		0000000b	> [245][31:0]	000000f5
> [12][31:0]		0000000c	> [246][31:0]	000000f6
> [13][31:0]		0000000d	> [247][31:0]	000000f7
> [14][31:0]		0000000e	> [248][31:0]	000000f8
> [15][31:0]		0000000f	> [249][31:0]	000000f9
> [16][31:0]		00000010	> [250][31:0]	000000fa
> [17][31:0]		00000011	> [251][31:0]	000000fb
> [18][31:0]		00000012	> [252][31:0]	000000fc
> [19][31:0]		00000013	> [253][31:0]	000000fd
> [20][31:0]		00000069	> [254][31:0]	000000fe
> [21][31:0]		000000b3	> [255][31:0]	000000ff
> [22][31:0]		00000086		
cache_mem[0:7][0:3][31:0]		'{'000000e0,000000e1,000000e2,000000e3},{000003ec,000003f0,000003ec,000000c4},{000000a0,000000a1,000000a2,000000a3},{00000040,00000041,00000042,00000043},{000000e0,000000e1,000000e2,000000e3},{000003ec,000003f0,000003ec,000000c4},{000000a0,000000a1,000000a2,000000a3},{00000040,00000041,00000042,00000043},{000000c4,000000c5,000000c6,000000c7},{00000024,00000025,00000026,00000027},{000000e4,000000e5,000000e6,000000e7},{00000044,00000045,00000046,00000047},{000000c8,000000c9,000000ca,000000cb},{000000e8,000000e9,000000ea,000000eb},{00000028,00000029,0000002a,0000002b},{00000048,00000049,0000004a,0000004b},{000000cc,000000cd,000000ce,000000cf},{000000ec,000000ed,000000ee,000000ef},{0000002c,0000002d,0000002e,0000002f},{0000004c,0000004d,0000004e,0000004f},{000000f0,000000f1,000000f2,000000f3},{000000d0,000000d1,000000d2,000000d3},{00000030,00000031,00000032,00000033},{00000050,00000051,00000052,00000053},{000000f4,000000f5,000000f6,000000f7},{00000014,00000015,00000016,00000017},{00000034,00000035,00000036,00000037},{000000d4,000000d5,000000d6,000000d7},{000000f8,000000f9,000000fa,000000fb},{00000018,00000019,0000001a,0000001b},{00000038,00000039,0000003a,0000003b},{000000d8,000000d9,000000da,000000db},{000000fc,000000fd,000000fe,000000ff},{0000001c,0000001d,0000001e,0000001f},{0000003c,0000003d,0000003e,0000003f},{000000dc,000000dd,000000de,000000df}'		
> cache_tags[0:7][0:3][4:0]		'{07,1f,05,02},{06,01,07,02},{06,07,01,02},{08,07,01,02},{07,06,01,02},{07,00,01,06},{07,00,01,06},{07,00,01,06}'		
> valid[0:7][0:3]		'{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1}'		
> dirty[0:7][0:3]		'{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1},{0,1,1,1}'		

上图为局部运行结果，下图为运行结束之后 **Cache** 中的值。易看出运行结果中局部无序的数据都在 **cache** 中。

> mem_count[31:0]	6309
> miss_count[31:0]	422
> hit_count[31:0]	5887

上图是缺失率统计信息。

4. 综合结果

(1). FIFO Cache

```

1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 1, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN = 2, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 7, // tag长度
5     parameter WAY_CNT      = 2 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 )(
7     input  clk, rst,
8     output miss,           // 对CPU发出的miss信号
9     input  [31:0] addr,    // 读写请求地址
10    input  rd_req,         // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input  wr_req,         // 写请求信号
13    input  [31:0] wr_data   // 要写入的数据，一次写一个word
14 );

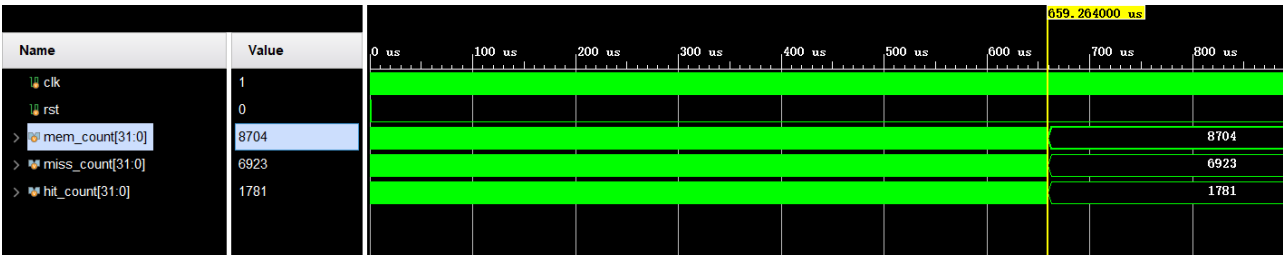
```

Resource	Utilization	Available	Utilization %
LUT	1080	63400	1.70
FF	1236	126800	0.97
BRAM	1	135	0.74
IO	79	210	37.62

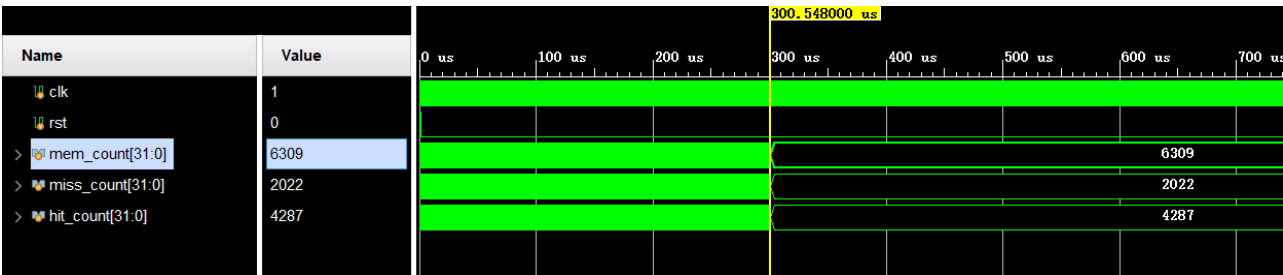
对于 `LINE_ADDR_LEN = 1`, `SET_ADDR_LEN = 2`, `TAG_ADDR_LEN = 7`, `WAY_CNT = 2` 的 FIFO Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



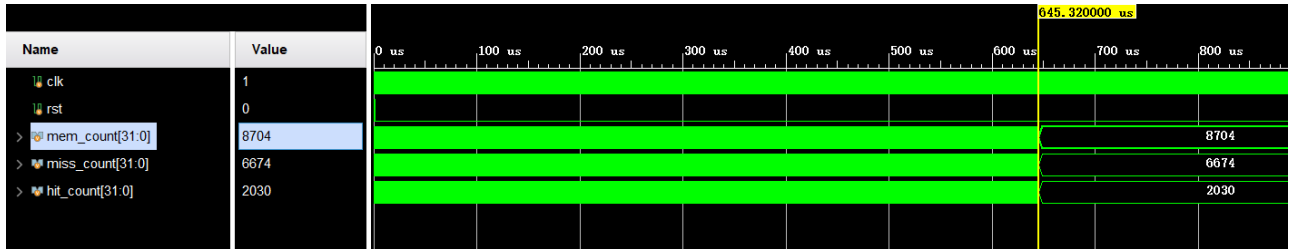
```
1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 1, // line内地址长度，决定了每个line
      具有2^3个word
3     parameter SET_ADDR_LEN = 2, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 7, // tag长度
5     parameter WAY_CNT = 4 // 组相连度，决定了每组中有多少路
      line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input [31:0] addr, // 读写请求地址
10    input rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input wr_req, // 写请求信号
13    input [31:0] wr_data // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	2290	63400	3.61
FF	2082	126800	1.64
BRAM	1	135	0.74
IO	79	210	37.62

对于 `LINE_ADDR_LEN = 2, SET_ADDR_LEN = 3, TAG_ADDR_LEN = 5, WAY_CNT = 2` 的 FIFO Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



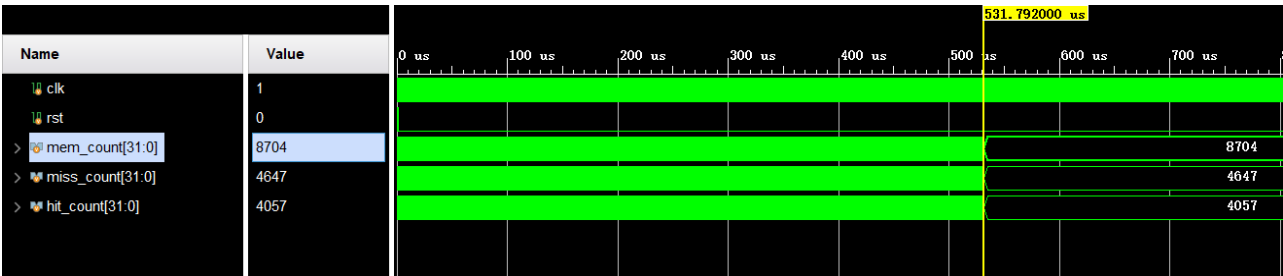
```
1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 2, // line内地址长度，决定了每个line
    具有 $2^3$ 个word
3     parameter SET_ADDR_LEN = 3, // 组地址长度，决定了一共有 $2^3=8$ 组
4     parameter TAG_ADDR_LEN = 5, // tag长度
5     parameter WAY_CNT = 2 // 组相连度，决定了每组中有多少路
    line, 这里是直接映射型cache，因此该参数没用到
6 ) (
7     input clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input [31:0] addr, // 读写请求地址
10    input rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input wr_req, // 写请求信号
13    input [31:0] wr_data // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	2137	63400	3.37
FF	3260	126800	2.57
BRAM	1	135	0.74
IO	79	210	37.62

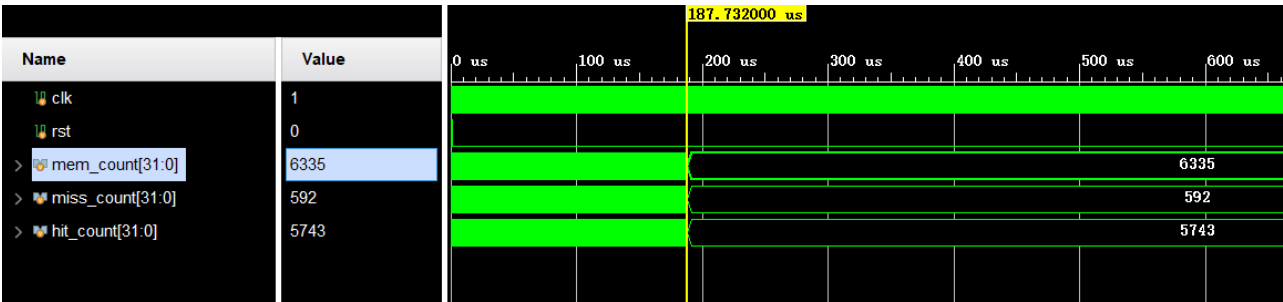
对于 `LINE_ADDR_LEN = 2`, `SET_ADDR_LEN = 3`, `TAG_ADDR_LEN = 5`, `WAY_CNT = 4` 的 FIFO Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



```
1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 2, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN = 3, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 5, // tag长度
5     parameter WAY_CNT = 4 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input [31:0] addr, // 读写请求地址
10    input rd_req, // 读请求信号
```

```

11     output reg [31:0] rd_data, // 读出的数据，一次读一个word
12     input  wr_req,           // 写请求信号
13     input  [31:0] wr_data    // 要写入的数据，一次写一个word
14 );

```

Resource	Utilization	Available	Utilization %
LUT	3658	63400	5.77
FF	5935	126800	4.68
BRAM	1	135	0.74
IO	79	210	37.62

对于 $\text{LINE_ADDR_LEN} = 2$, $\text{SET_ADDR_LEN} = 3$, $\text{TAG_ADDR_LEN} = 5$, $\text{WAY_CNT} = 2$ 的 FIFO Cache 而言，资源占用率如上图所示。

```

1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 2, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN  = 3, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN  = 5, // tag长度
5     parameter WAY_CNT       = 4 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input  clk, rst,
8     output miss,           // 对CPU发出的miss信号
9     input  [31:0] addr,    // 读写请求地址
10    input  rd_req,         // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input  wr_req,         // 写请求信号
13    input  [31:0] wr_data   // 要写入的数据，一次写一个word
14 );

```

Resource	Utilization	Available	Utilization %
LUT	3658	63400	5.77
FF	5935	126800	4.68
BRAM	1	135	0.74
IO	79	210	37.62

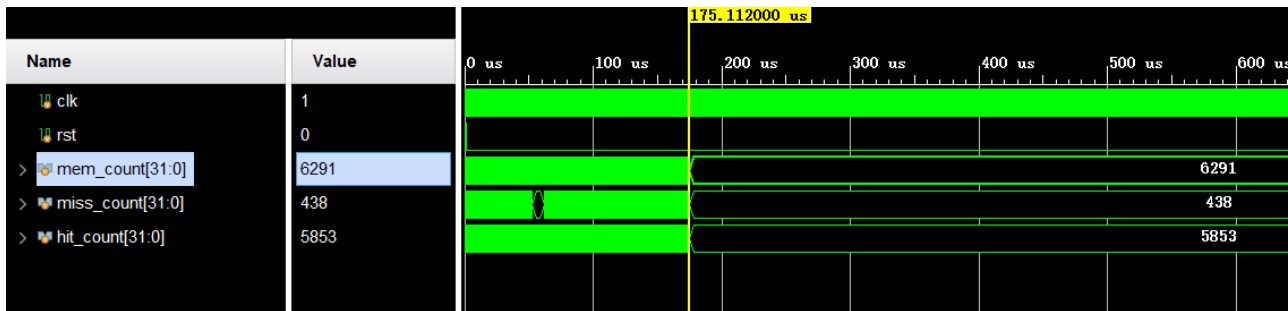
对于 `LINE_ADDR_LEN = 2, SET_ADDR_LEN = 3, TAG_ADDR_LEN = 5, WAY_CNT = 4` 的 FIFO Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



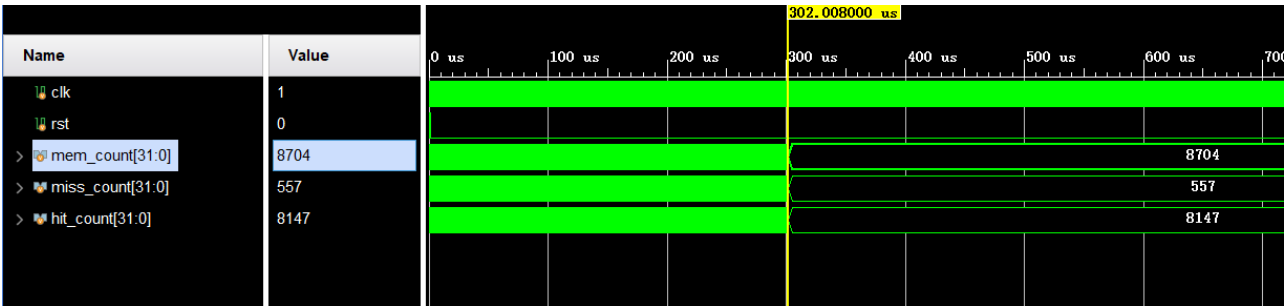
```
1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 4, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN = 4, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 2, // tag长度
5     parameter WAY_CNT = 2 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input [31:0] addr, // 读写请求地址
10    input rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input wr_req, // 写请求信号
13    input [31:0] wr_data // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	8822	63400	13.91
FF	18165	126800	14.33
BRAM	1	135	0.74
IO	79	210	37.62

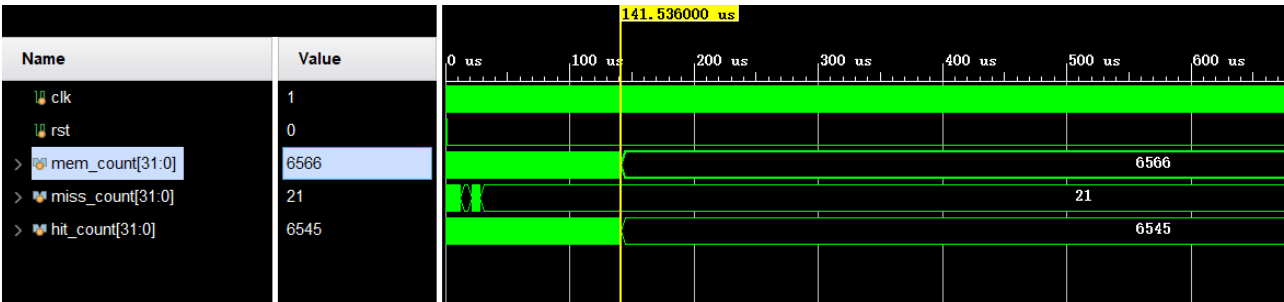
对于 `LINE_ADDR_LEN = 4`, `SET_ADDR_LEN = 4`, `TAG_ADDR_LEN = 2`, `WAY_CNT = 2` 的 FIFO Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



```
1 module my_cache_FIFO #(
2     parameter LINE_ADDR_LEN = 4, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN = 4, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 2, // tag长度
5     parameter WAY_CNT = 4 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input [31:0] addr, // 读写请求地址
10    input rd_req, // 读请求信号
```



```

11     output reg [31:0] rd_data, // 读出的数据，一次读一个word
12     input  wr_req,           // 写请求信号
13     input  [31:0] wr_data    // 要写入的数据，一次写一个word
14 );

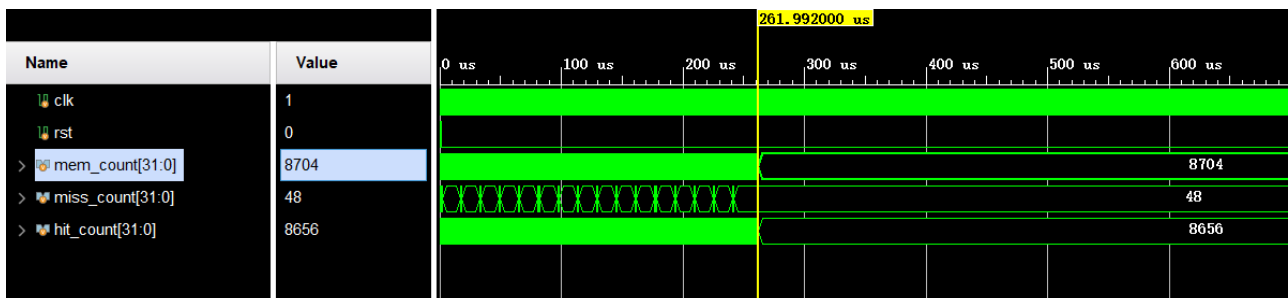
```

Resource	Utilization	Available	Utilization %
LUT	15475	63400	24.41
FF	35689	126800	28.15
BRAM	1	135	0.74
IO	79	210	37.62

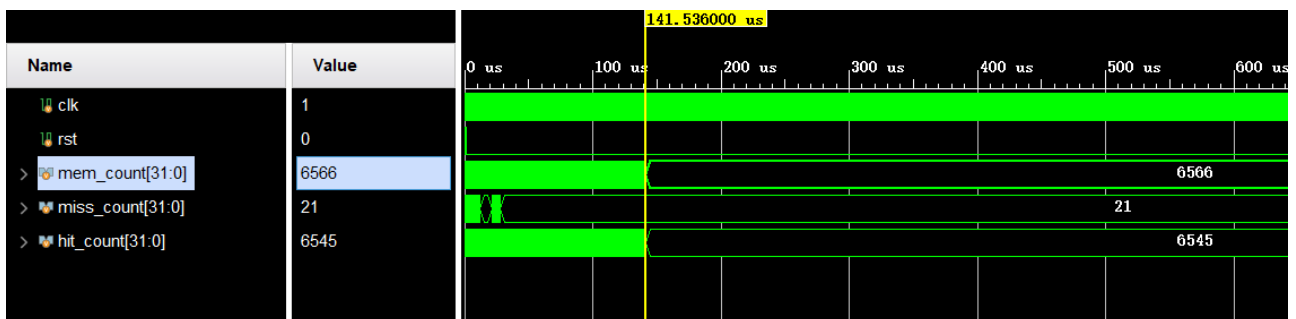
对于 `LINE_ADDR_LEN = 4`, `SET_ADDR_LEN = 4`, `TAG_ADDR_LEN = 2`, `WAY_CNT = 4` 的 FIFO Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



(2). LRU Cache

```

1 module my_cache_LRU #(
2     parameter LINE_ADDR_LEN = 1, // line内地址长度，决定了每个line
   具有2^3个word

```

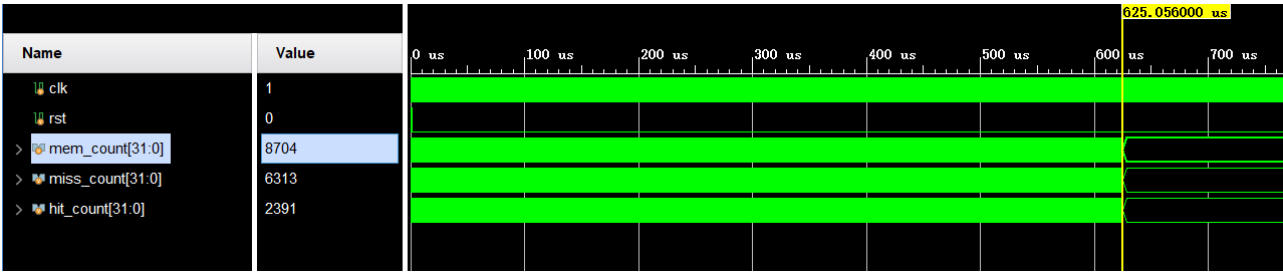
```
3     parameter SET_ADDR_LEN = 2, // 组地址长度，决定了一共有 $2^3=8$ 组
4     parameter TAG_ADDR_LEN = 7, // tag长度
5     parameter WAY_CNT      = 2 // 组相连度，决定了每组中有多少路
    line, 这里是直接映射型cache，因此该参数没用到
6 )(
7     input  clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input  [31:0] addr, // 读写请求地址
10    input  rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input  wr_req, // 写请求信号
13    input  [31:0] wr_data // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	1050	63400	1.66
FF	1236	126800	0.97
BRAM	1	135	0.74
IO	79	210	37.62

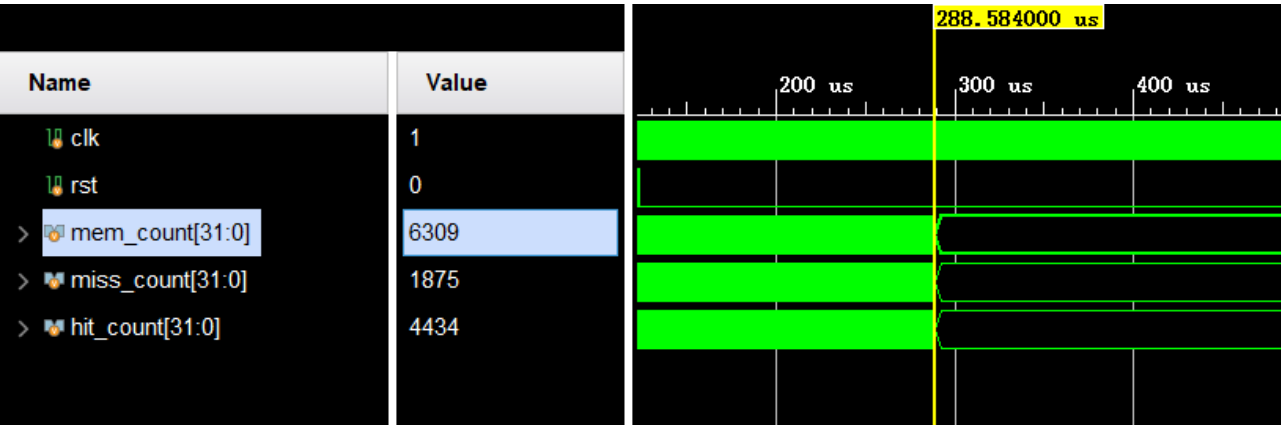
对于 `LINE_ADDR_LEN = 1, SET_ADDR_LEN = 2, TAG_ADDR_LEN = 7, WAY_CNT = 2` 的 LRU Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



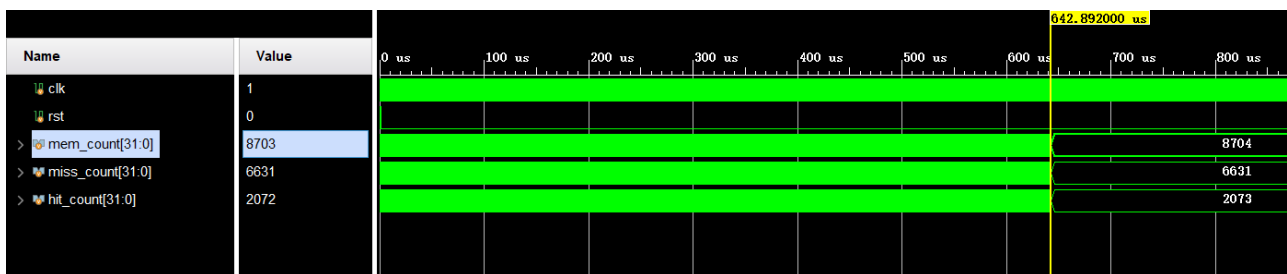
```
1 module my_cache_LRU #(
2     parameter LINE_ADDR_LEN = 1, // line内地址长度，决定了每个line
    具有2^3个word
3     parameter SET_ADDR_LEN = 2, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 7, // tag长度
5     parameter WAY_CNT = 4 // 组相连度，决定了每组中有多少路
    line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input  clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input  [31:0] addr, // 读写请求地址
10    input  rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input  wr_req, // 写请求信号
13    input  [31:0] wr_data // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	2093	63400	3.30
FF	2077	126800	1.64
BRAM	1	135	0.74
IO	79	210	37.62

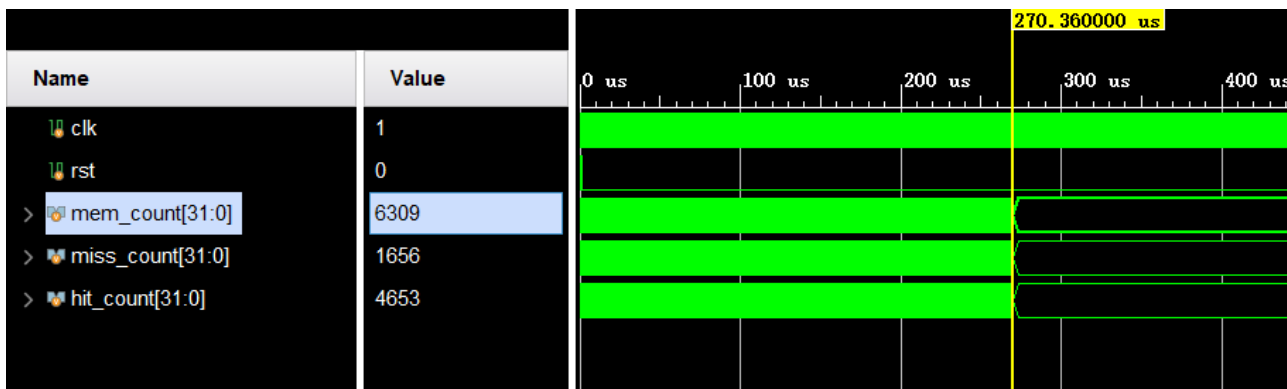
对于 `LINE_ADDR_LEN = 1, SET_ADDR_LEN = 2, TAG_ADDR_LEN = 7, WAY_CNT = 4` 的 LRU Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排:



```

1 module my_cache_LRU #(
2     parameter LINE_ADDR_LEN = 2, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN  = 3, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN  = 5, // tag长度
5     parameter WAY_CNT       = 2 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 )(
7     input  clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input  [31:0] addr, // 读写请求地址
10    input  rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input  wr_req, // 写请求信号
13    input  [31:0] wr_data // 要写入的数据，一次写一个word
14 );

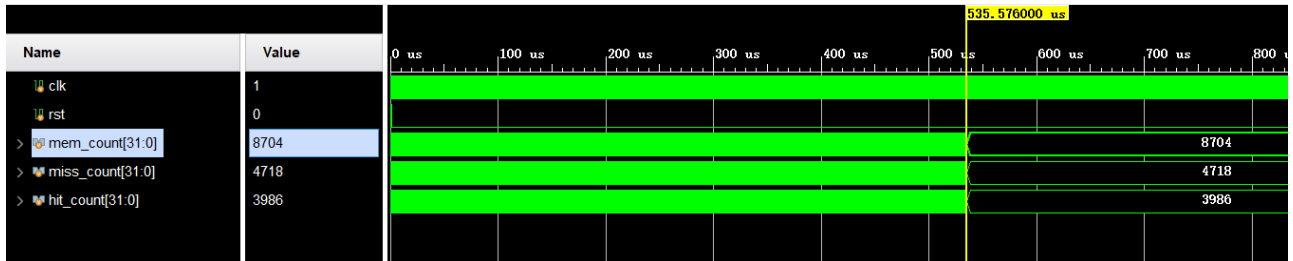
```

Resource	Utilization	Available	Utilization %
LUT	1846	63400	2.91
FF	3258	126800	2.57
BRAM	1	135	0.74
IO	79	210	37.62

对于 `LINE_ADDR_LEN = 2, SET_ADDR_LEN = 3, TAG_ADDR_LEN = 5, WAY_CNT = 2` 的 LRU Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



```
1 module my_cache_LRU #(
2     parameter LINE_ADDR_LEN = 2, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN = 3, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 5, // tag长度
5     parameter WAY_CNT = 4 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input clk, rst,
8     output miss, // 对CPU发出的miss信号
9     input [31:0] addr, // 读写请求地址
10    input rd_req, // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input wr_req, // 写请求信号
13    input [31:0] wr_data // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	4026	63400	6.35
FF	5942	126800	4.69
BRAM	1	135	0.74
IO	79	210	37.62

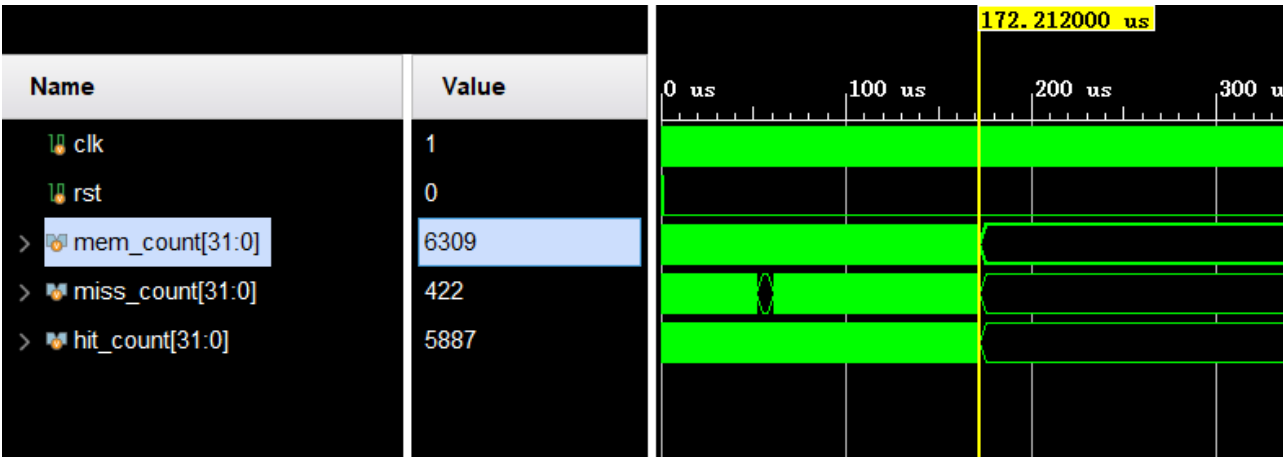
对于 `LINE_ADDR_LEN = 2`, `SET_ADDR_LEN = 3`, `TAG_ADDR_LEN = 5`, `WAY_CNT = 4` 的 LRU Cache 而言, 资源占用率如上图所示。

缺失率统计:

矩阵乘法:



快排:



```
1 module my_cache_LRU #(
2     parameter LINE_ADDR_LEN = 4, // line内地址长度, 决定了每个line
      具有2^3个word
3     parameter SET_ADDR_LEN = 4, // 组地址长度, 决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN = 2, // tag长度
5     parameter WAY_CNT = 2 // 组相连度, 决定了每组中有多少路
      line, 这里是直接映射型cache, 因此该参数没用到
6 ) (
7     input clk, rst,
```

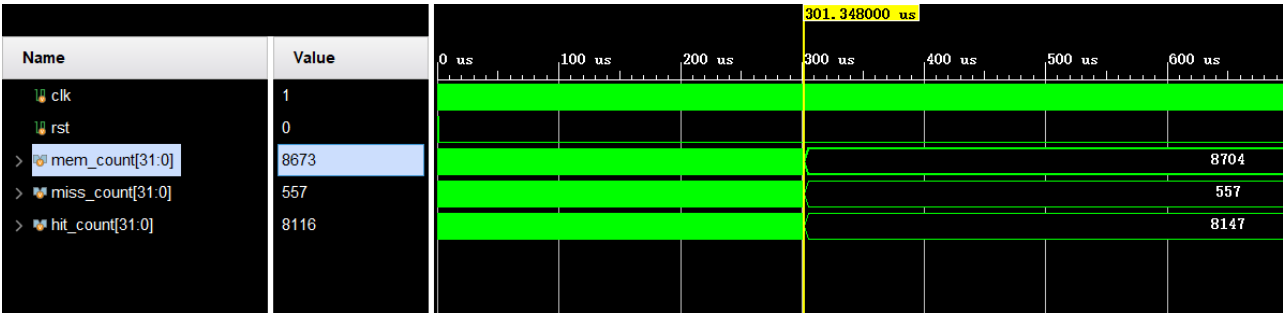
```
8      output miss,           // 对CPU发出的miss信号
9      input  [31:0] addr,    // 读写请求地址
10     input  rd_req,         // 读请求信号
11     output reg [31:0] rd_data, // 读出的数据，一次读一个word
12     input  wr_req,         // 写请求信号
13     input  [31:0] wr_data  // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	11015	63400	17.37
FF	20257	126800	15.98
BRAM	1	135	0.74
IO	79	210	37.62

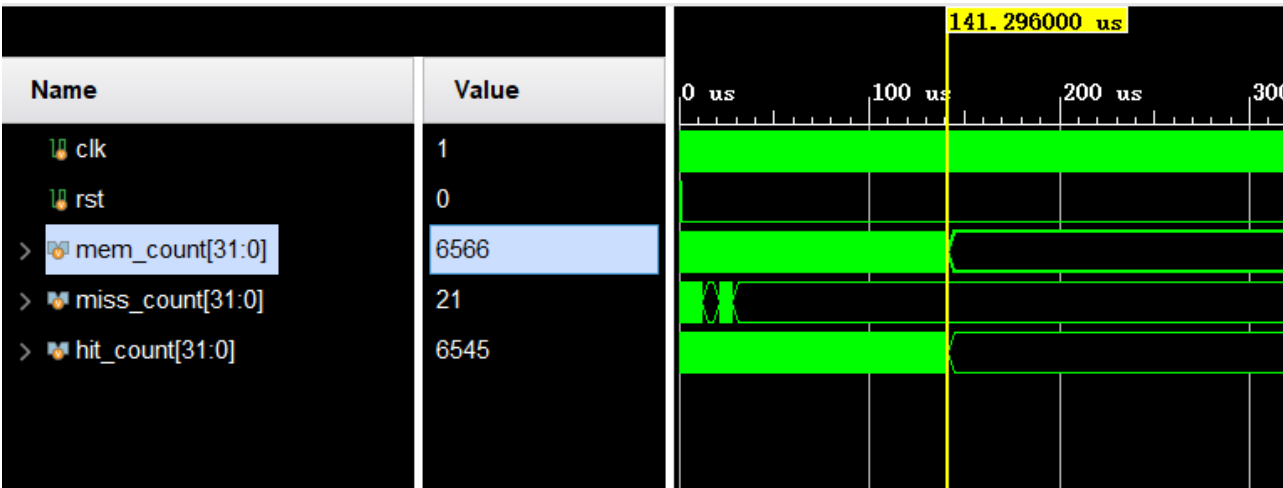
对于 `LINE_ADDR_LEN = 4, SET_ADDR_LEN = 4, TAG_ADDR_LEN = 2, WAY_CNT = 2` 的 LRU Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：



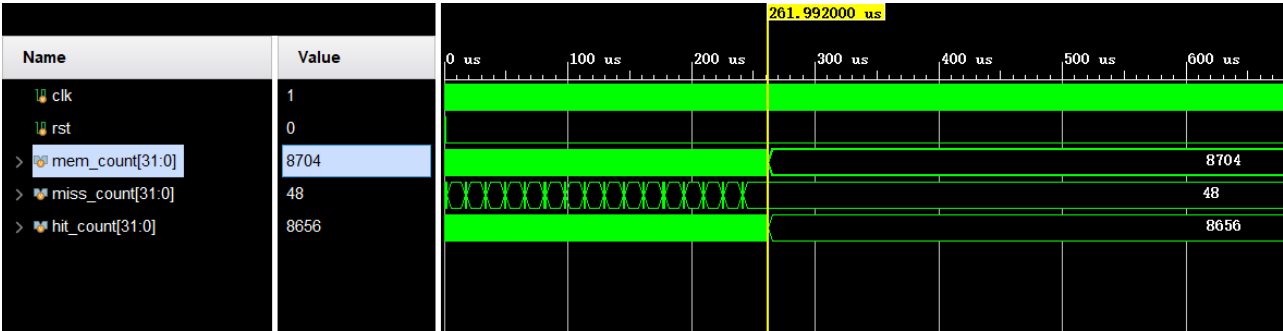
```
1 module my_cache_LRU #(
2     parameter LINE_ADDR_LEN = 4, // line内地址长度，决定了每个line
   具有2^3个word
3     parameter SET_ADDR_LEN  = 4, // 组地址长度，决定了一共有2^3=8组
4     parameter TAG_ADDR_LEN  = 2, // tag长度
5     parameter WAY_CNT       = 4 // 组相连度，决定了每组中有多少路
   line，这里是直接映射型cache，因此该参数没用到
6 ) (
7     input  clk, rst,
8     output miss,           // 对CPU发出的miss信号
9     input  [31:0] addr,    // 读写请求地址
10    input  rd_req,         // 读请求信号
11    output reg [31:0] rd_data, // 读出的数据，一次读一个word
12    input  wr_req,         // 写请求信号
13    input  [31:0] wr_data  // 要写入的数据，一次写一个word
14 );
```

Resource	Utilization	Available	Utilization %
LUT	21890	63400	34.53
FF	39820	126800	31.40
BRAM	1	135	0.74
IO	79	210	37.62

对于 `LINE_ADDR_LEN = 4`，`SET_ADDR_LEN = 4`，`TAG_ADDR_LEN = 2`，`WAY_CNT = 4` 的 LRU Cache 而言，资源占用率如上图所示。

缺失率统计：

矩阵乘法：



快排：

