

# Computer Architecture Lab3

PB19071501 李平治

## 实验目的

- 掌握直接相连和组相连Cache的原理并用VHDL实现
- 权衡cache size增大带来的命中率提升收益和存储资源电路面积的开销，权衡选择合适的组相连度
- 体会使用复杂电路实现复杂替换策略带来的收益和简单替换策略的优势（有时候简单策略比复杂策略效果不差很多甚至可能更好）
- 理解写回法的优劣

## 实验环境

- Vivado 2019.1
- VMWard Fusion12.2.3虚拟机中的WIndows 10

## 实验内容

- 阶段一：将直接相连Cache代码修改为组相连Cache，并通过testbench测试
- 阶段二：将阶段一的组相连Cache添加到实验一的CPU中，并运行快速排序和矩阵乘法二进制代码，统计Cache缺失率
- 阶段三：修改Cache size、组相连度、替换策略，体会针对不同程序的优化效果和电路面积的变化，并给出较优的Cache参数和策略

## 阶段一

增加WAY\_CNT路数维度、命中位置way\_addr、当前时间time\_count和每块的时间相关数组record，并修改状态机：

```

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 [WAY_ADDR_LEN - 1 : 0] way_addr;
6 reg [31:0] record [SET_SIZE][WAY_CNT];

```

```

1 always @ (posedge clk or posedge rst) begin // ?? cache ???
2     if(rst) begin
3         cache_stat <= IDLE;
4         time_count <= 0;
5         for(integer i = 0; i < SET_SIZE; i++) begin
6             for(integer j = 0; j < WAY_CNT; j++) begin
7                 dirty[i][j] = 1'b0;
8                 valid[i][j] = 1'b0;
9                 record[i][j] = 1'b0;
10            end
11        end
12        for(integer k = 0; k < LINE_SIZE; k++)
13            mem_wr_line[k] <= 0;
14        mem_wr_addr <= 0;
15        {mem_rd_tag_addr, mem_rd_set_addr} <= 0;
16        rd_data <= 0;
17    end else begin
18        time_count = time_count + 1;
19        case(cache_stat)
20        IDLE: begin
21            if(cache_hit) begin
22                if(wr_req | rd_req) begin
23                    if(CACHE_POLICY == 0)begin //LRU
24                        record[set_addr][way_addr] <= time_count;
25                    end
26                end
27                if(rd_req) begin // 如果cache命中, 并且是读请求,
28                    rd_data <= cache_mem[set_addr][way_addr]
29                    [line_addr]; //则直接从cache中取出要读的数据
30                end else if(wr_req) begin // 如果cache命中, 并且是写请求,
31                    cache_mem[set_addr][way_addr][line_addr] <=
32                    wr_data; // 则直接向cache中写入数据
33                    dirty[set_addr][way_addr] <= 1'b1;
34                    // 写数据的同时置脏位
35                end
36            end else begin
37                if(wr_req | rd_req) begin // 如果 cache 未命中, 并且有
38                读写请求, 则需要换入
39                    if(valid[set_addr][way_addr] & dirty[set_addr]
40                    [way_addr]) begin // 如果 要换入的cache line 本来有效, 且脏, 则需要先将它换出
41                        cache_stat <= SWAP_OUT;
42                    end
43                end
44            end
45        end
46    end
47 end

```

```

37         mem_wr_addr <= {cache_tags[set_addr]
[way_addr], set_addr};
38         mem_wr_line <= cache_mem[set_addr][way_addr];
39     end else begin
// 反之，不需要换出，直接换入
40         cache_stat <= SWAP_IN;
41     end
42     {mem_rd_tag_addr, mem_rd_set_addr,
mem_rd_way_addr} <= {tag_addr, set_addr, way_addr};
43     end
44     end
45     end
46     SWAP_OUT: begin
47         if(mem_gnt) begin // 如果主存握手信号有效，说明换出成
功，跳到下一状态
48             cache_stat <= SWAP_IN;
49         end
50     end
51     SWAP_IN: begin
52         if(mem_gnt) begin // 如果主存握手信号有效，说明换入成
功，跳到下一状态
53             cache_stat <= SWAP_IN_OK;
54         end
55     end
56     SWAP_IN_OK: begin // 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
57         for(integer i=0; i<LINE_SIZE; i++)
58             cache_mem[mem_rd_set_addr][mem_rd_way_addr][i] <= mem_rd_line[i];
59             cache_tags[mem_rd_set_addr][mem_rd_way_addr] <=
mem_rd_tag_addr;
60             valid [mem_rd_set_addr][mem_rd_way_addr] <= 1'b1;
61             dirty [mem_rd_set_addr][mem_rd_way_addr] <= 1'b0;
62             record[mem_rd_set_addr][mem_rd_way_addr] <= time_count; //
Both LRU and FIFO
63             cache_stat <= IDLE; // 回到就绪状态
64         end
65     endcase
66 end
end

```

其中LRU算法和FIFO算法的唯一区别在于：LRU仅在换入块时用time\_count更新对应的record，但FIFO在换入块和命中时都用time\_count更新对应的record

根据时间记录数组record来选择替换块：

```

1 always @ (*) begin
2     if(cache_hit) begin // hit
3         for (integer i = 0; i < WAY_CNT; i = i + 1)
4             begin
5                 if(cache_tags[set_addr][i] == tag_addr)
6                     begin
7                         way_addr = i;

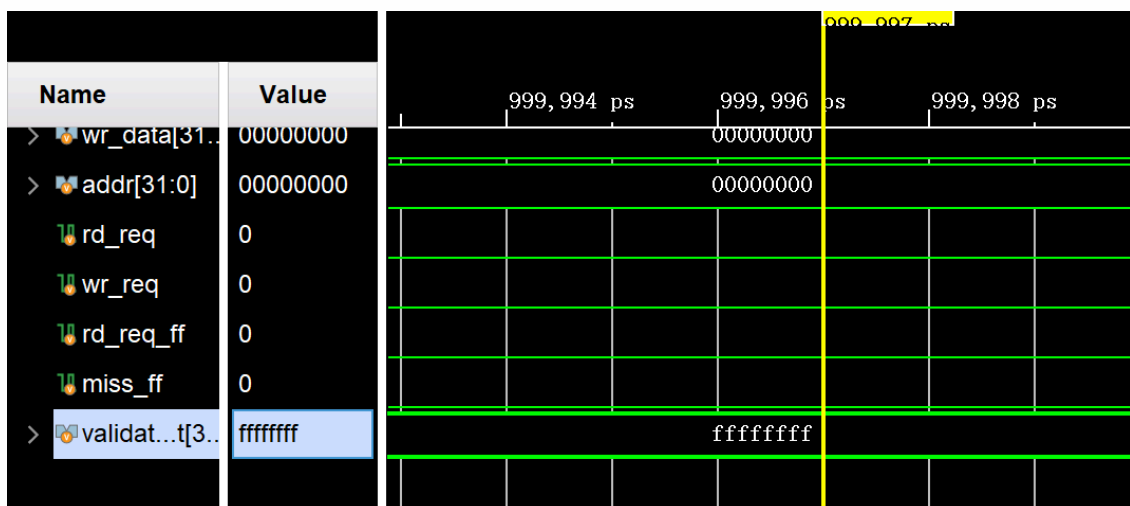
```

```

8         break;
9     end
10 end
11 end else if(rd_req | wr_req) // miss
12 begin
13     way_addr = 0;
14     // search for the minit order way
15     for (integer i = 1; i < WAY_CNT; i = i + 1)
16     begin
17         if (record[set_addr][i] < record[set_addr][way_addr])
18             way_addr = i;
19     end
20 end
21 end

```

使用cache\_tb.v进行测试，通过：



## 阶段二

- 将CPUSrcCode/MemWbSegReg/WbData.v中的DataCache用本次实现的Cache替换，并加入miss和hit次数统计 (miss\_count, hit\_count)：

```

1 wire cache_write_en;
2 assign cache_write_en = (write_en == 4'b1111) ? 1'b1 : 1'b0;
3 cache cache1(
4     .clk(clk),
5     .rst(rst),
6     .addr(addr),
7     .rd_req(wb_select),
8     .rd_data(data_raw),
9     .wr_req(cache_write_en),
10    .wr_data(in_data),
11    .miss(miss)
12 );

```

```

1  always @(posedge clk or posedge rst)
2  begin
3      if(rst) begin
4          last_addr <= 0;
5          hit_count <= 32'b0;
6          miss_count <= 32'b0;
7      end else begin
8          if(cache_rd_wr)begin
9              last_addr <= addr;
10         end else begin
11             last_addr <= last_addr;
12         end
13     end
14 end
15 always @(posedge clk or posedge rst)
16 begin
17     if(rst) begin
18         hit_count <= 32'b0;
19         miss_count <= 32'b0;
20     end else begin
21         if((last_addr != addr) & cache_rd_wr)begin
22             if(miss)
23                 miss_count <= miss_count + 1;
24             else
25                 hit_count <= hit_count + 1;
26         end else begin
27             miss_count <= miss_count;
28             hit_count <= hit_count;
29         end
30     end
31 end

```

- 修改Hazard，当miss时进行stall:

```

1  else if (cache_miss==1) begin
2      bubbleF = 1;
3      flushF = 0;
4      bubbleD = 1;
5      flushD = 0;
6      bubbleE = 1;
7      flushE = 0;
8  end

```

```

1  else if (cache_miss==1) begin
2      bubbleW = 1;
3      flushW = 0;
4  end

```

```

























1 else if (cache_miss==1) begin
2     bubbleM = 1;
3     flushM = 0;
4 end

```

- 将QuickSort二进制代码和数据放入CPU内存中，并运行。结果表明排序成功。

Name	Value
▼ ram_cell[0:4095][31:0]	00000063,00C
> [8][31:0]	00000008
> [9][31:0]	00000009
> [10][31:0]	0000000a
> [11][31:0]	0000000b
> [12][31:0]	0000000c
> [13][31:0]	0000000d
> [14][31:0]	0000000e
> [15][31:0]	0000000f
> [16][31:0]	00000010
> [17][31:0]	00000011
> [18][31:0]	00000012
> [19][31:0]	00000013
> [20][31:0]	00000014
> [21][31:0]	00000015
> [22][31:0]	00000016
> [23][31:0]	00000017
> [24][31:0]	00000018
> [25][31:0]	00000019
> [26][31:0]	0000001a
> [27][31:0]	0000001b
> [28][31:0]	0000001c
> [29][31:0]	0000001d
> [30][31:0]	0000001e
> [31][31:0]	0000001f
> [32][31:0]	00000020
> [33][31:0]	00000021

- 将MatrixMultiply二进制代码和数据放入CPU内存中，并运行，结果表明计算成功

Name	Value
▼  ram_cell[0:4095][31:0]	d2de7cac,30aaa1
>  [0][31:0]	d2de7cac
>  [1][31:0]	30aaa1ea
>  [2][31:0]	6655f6b7
>  [3][31:0]	70224917
>  [4][31:0]	678c55ec
>  [5][31:0]	d40d418e
>  [6][31:0]	59593ce5
>  [7][31:0]	ab564f3c
>  [8][31:0]	2b4a21cc
>  [9][31:0]	232d7d98
>  [10][31:0]	d0194190
>  [11][31:0]	4ae6e03b
>  [12][31:0]	8c5da8a6
>  [13][31:0]	b94041a0
>  [14][31:0]	adfacc478
>  [15][31:0]	dd95a879
>  [16][31:0]	f9f3490c
>  [17][31:0]	c3fccd41
>  [18][31:0]	54a35f8f
>  [19][31:0]	2a7f50e0
>  [20][31:0]	ba7b7dac
>  [21][31:0]	ec2fb562
>  [22][31:0]	9f03e1db
>  [23][31:0]	23c658b7
>  [24][31:0]	458e447d
>  [25][31:0]	9abbdc5e
>  [26][31:0]	d07f2bd9
>  [27][31:0]	1e963a4f
>  [28][31:0]	fbee3003
>  [29][31:0]	42b336d2
>  [30][31:0]	6500cabe
>  [31][31:0]	bc0d6eed
>  [32][31:0]	02635574
>  [33][31:0]	b40dbc36

```

initial begin
    // dst matrix C
    ram_cell[ 0] = 32'h0; // 32'hd2de7cac;
    ram_cell[ 1] = 32'h0; // 32'h30aaa1ea;
    ram_cell[ 2] = 32'h0; // 32'h6655f6b7;
    ram_cell[ 3] = 32'h0; // 32'h70224917;
    ram_cell[ 4] = 32'h0; // 32'h678c55ec;
    ram_cell[ 5] = 32'h0; // 32'hd40d418e;
    ram_cell[ 6] = 32'h0; // 32'h59593ce5;
    ram_cell[ 7] = 32'h0; // 32'hab564f3c;
    ram_cell[ 8] = 32'h0; // 32'h2b4a21cc;
    ram_cell[ 9] = 32'h0; // 32'h232d7d98;
    ram_cell[10] = 32'h0; // 32'hd0194190;
    ram_cell[11] = 32'h0; // 32'h4ae6e03b;
    ram_cell[12] = 32'h0; // 32'h8c5da8a6;
    ram_cell[13] = 32'h0; // 32'hb94041a0;
    ram_cell[14] = 32'h0; // 32'hadfac478;
    ram_cell[15] = 32'h0; // 32'hdd95a879;
    ram_cell[16] = 32'h0; // 32'hf9f3490c;
    ram_cell[17] = 32'h0; // 32'hc3fccd41;
    ram_cell[18] = 32'h0; // 32'h54a35f8f;
    ram_cell[19] = 32'h0; // 32'h2a7f50e0;
    ram_cell[20] = 32'h0; // 32'hba7b7dac;
    ram_cell[21] = 32'h0; // 32'hec2fb562;
    ram_cell[22] = 32'h0; // 32'h9f03e1db;
    ram_cell[23] = 32'h0; // 32'h23c658b7;
    ram_cell[24] = 32'h0; // 32'h458e447d;
    ram_cell[25] = 32'h0; // 32'h9abbd5e;
    ram_cell[26] = 32'h0; // 32'hd07f2bd9;
    ram_cell[27] = 32'h0; // 32'h1e963a4f;
    ram_cell[28] = 32'h0; // 32'hfbec3003;
    ram_cell[29] = 32'h0; // 32'h42b336d2;
    ram_cell[30] = 32'h0; // 32'h6500cabe;
    ram_cell[31] = 32'h0; // 32'hbc0d6eed;
    ram_cell[32] = 32'h0; // 32'h02635574;
    ram_cell[33] = 32'h0; // 32'hb40dbc36;

```

## 阶段三

### ■ Cache资源占用分析

将Cache模块作为顶层，修改Cache的LINE\_ADDR\_LEN, SET\_ADDR\_LEN, TAG\_ADDR\_LEN, WAY\_CNT参数，同时为了消除主存大小的影响，保持LINE\_ADDR\_LEN+SET\_ADDR\_LEN+TAG\_ADDR\_LEN不变，测试在两种策略下的资源占用情况。

例如，当参数采取



```
1 parameter LINE_ADDR_LEN = 3,
2 parameter SET_ADDR_LEN  = 3,
3 parameter TAG_ADDR_LEN  = 6,
4 parameter WAY_CNT       = 4,
5 parameter CACHE_POLICY  = 0
```

时，综合资源占用分析结果为

Resource	Utilization	Available	Utilization %
LUT	5315	63400	8.38
FF	10461	126800	8.25
BRAM	4	135	2.96
IO	81	210	38.57

为报告从简，其他类似实验截图不再放上，结果以表格方式呈现，如下：

LRU：

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	LUT	FF
3	3	6	2	2613	5717
3	3	6	4	5315	10461
3	3	6	8	9660	19982
2	3	7	4	3775	6014
3	2	7	4	4121	5729
3	4	5	4	9540	19879
4	3	5	4	9571	19398

FIFO：

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	LUT	FF
3	3	6	2	1986	5717
3	3	6	4	3986	10451
3	3	6	8	9030	20006
2	3	7	4	2429	6004
3	2	7	4	3587	5731
3	4	5	4	7199	19872
4	3	5	4	8421	19402

分析表中结论可知：

- 在参数都相同的情况下，FIFO占用资源一般高于LRU
- LUT和FF（即电路面积）关于组相连度、组数、Line大小呈正相关

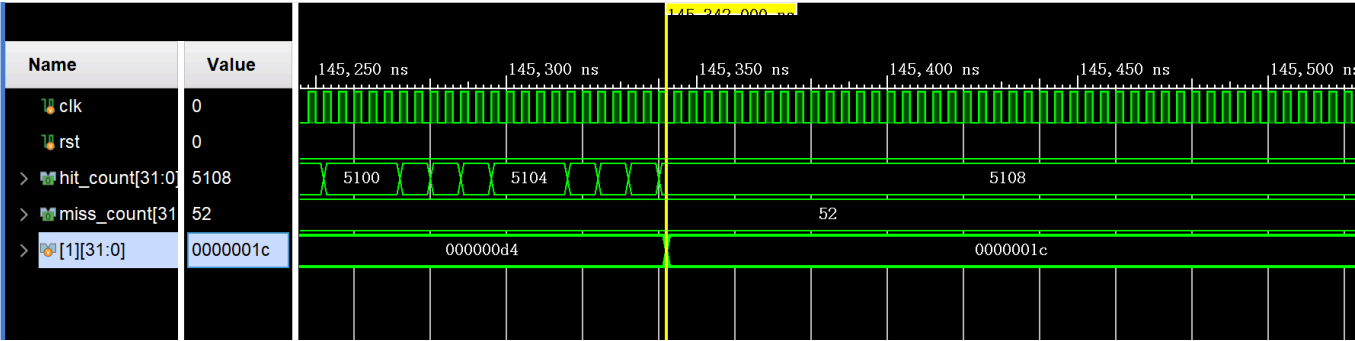
Cache性能分析

对Cache的不同参数和策略，用QuickSort和MatMul进行命中率、运行时间等指标的性能分析

例如，当参数采取

```
1 parameter LINE_ADDR_LEN = 3,
2 parameter SET_ADDR_LEN = 3,
3 parameter TAG_ADDR_LEN = 6,
4 parameter WAY_CNT = 4,
5 parameter CACHE_POLICY = 0
```

时，运行结果如下：



表明其命中率为  $\frac{5108}{5108+52} = 98.99\%$ ，运行时间为145240ns

为报告从简，其他类似实验截图不再放上，结果以表格方式呈现，如下

LRU-QuickSort256:

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Time(ns)	MissCount	HitCount	HitRate
3	3	6	2	168282	99	5061	98.08%
3	3	6	4	145350	52	5108	98.99%
3	3	6	8	137850	38	5122	99.26%
2	3	7	4	200800	190	4967	96.31%
3	2	7	4	166686	95	5065	98.16%
3	4	5	4	137850	38	5122	99.26%
4	3	5	4	133846	19	5141	99.63%

FIFO-QuickSort256:

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Time(ns)	MissCount	HitCount	HitRate
3	3	6	2	171350	106	5054	97.95%
3	3	6	4	142798	46	5114	99.11%
3	3	6	8	137956	38	5122	99.26%
2	3	7	4	207710	205	4955	96.03%
3	2	7	4	170078	103	5057	98.00%
3	4	5	4	137056	38	5122	99.26%
4	3	5	4	133848	19	5141	99.63%

LRU-MatMul16x16:

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Time(ns)	MissCount	HitCount	HitRate
3	3	6	2	1321640	4672	4032	46.32%
3	3	6	4	646422	1547	7157	82.23%
3	3	6	8	291484	123	8581	98.59%
2	3	7	4	1337224	4744	3960	45.50%
3	2	7	4	1321640	4672	4032	46.32%
3	4	5	4	289788	119	8585	98.63%
4	3	5	4	273056	56	8648	99.36%

FIFO-MatMul16x16:

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Time(ns)	MissCount	HitCount	HitRate
3	3	6	2	1363112	4864	3840	44.12%
3	3	6	4	664742	1682	6625	79.75%
3	3	6	8	294576	146	8558	98.32%
2	3	7	4	1340680	4760	3944	45.31%
3	2	7	4	1349288	4800	3904	44.85%
3	4	5	4	293728	144	8560	98.35%
4	3	5	4	276512	72	8632	99.17%

由表可知：

- 组数越大，运行时间越短，Cache命中率越大
- Line越大，运行时间越短，Cache命中率越大
- 组相连度越大，运行时间越短，Cache命中率越大
- 总体来说，FIFO策略和LRU策略在QuickSort和MatMul这两个测试样例下，效果差距较小，小于样例区别导致的差距

综合选择较优参数组：

- LRU和FIFO的Cache性能相差不大，但FIFO的电路面积显著小于LRU
- 参数组(3,3,6,8), (3,4,5,4), (4,3,5,4)的命中率较高 (QuickSort中略高，MatMul中显著高于其他参数组)，资源占用率相差不大

因此可以选择

```
1 parameter LINE_ADDR_LEN = 3,
2 parameter SET_ADDR_LEN  = 4,
3 parameter TAG_ADDR_LEN  = 5,
4 parameter WAY_CNT       = 4,
5 parameter CACHE_POLICY  = 1 //FIFO
```

作为Cache参数组

# 实验总结

---

本次试验框架设计文档友好，代码易读懂。通过具体实践，加深了本人对Cache及其参数、运行程序与性能之间关系的理解，进一步理解了CPU的设计思想与实践方法。

本次试验共花费约5h:

- 0.5h阅读文档
- 0.5h阅读简单cache代码
- 1h将简单cache修改为两种策略的组相连cache
- 1h将cache添加到cpu中
- 2h运行综合程序+书写实验报告