体系结构 lab3 - 实验报告

PB18111707 吕瑞

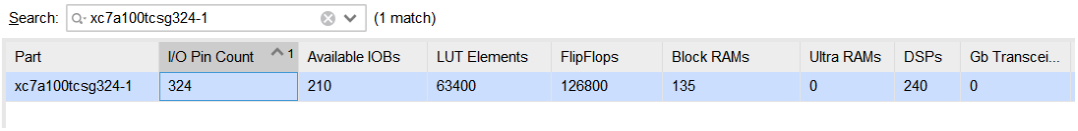
2021/5/31

实验目的

1. 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
2. 权衡选择合适的组相连度（相连度增大cache size也会增大，但是冲突miss会减低）
3. 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）
4. 理解写回法的优劣

实验环境

- 操作系统版本：Windows 10
- 处理器：Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz　1.80 GHz
- Vivado 2019.1
- FPGA 开发板型号：xc7a100tcsg324-1

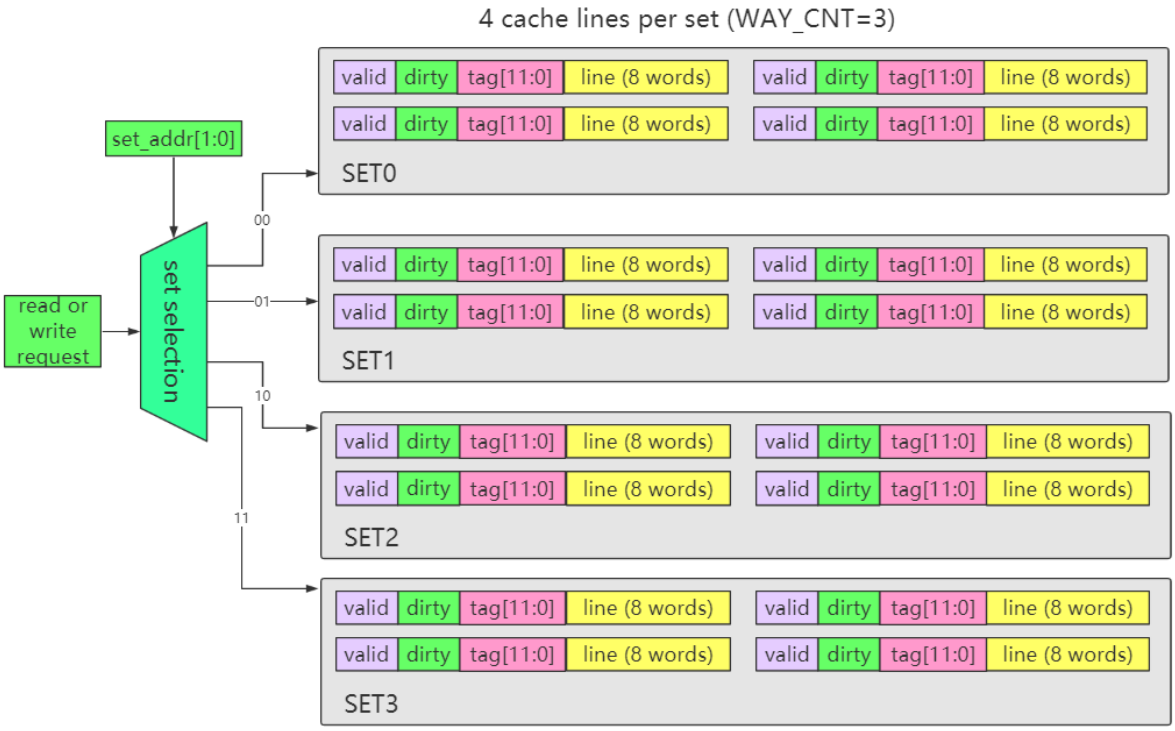


实验内容

组相连 Cache 的实现

直接相连（直接映射）可以看做组相连的特殊情况，即组内只有一路（line）。

K 路组相连即为：在使用 set 定位到对应的组后，需要的 cache 块可能在该组中的任意一个位置上，需要并行比较 cache 组中的每一个块的 tag 位和 valid 位，来确定最终引用的 cache line 地址。下面是 4 路组相连的结构图。



相比直接相连 cache，组相连需要修改的地方有：

1. 修改 cache_mem, cache_tags, vaild, dirty 这些 cache 变量，分别给他们增加一个维度，维度大小为 WAY_CNY

```
reg [            31:0] cache_mem    [SET_SIZE][WAY_CNT][LINE_SIZE]; // SET_SIZE个line，每个 SET 中有 WAY_CNT 个 line，每个line有LINE_SIZE个word
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE][WAY_CNT]; // SET_SIZE个TAG
reg                    valid        [SET_SIZE][WAY_CNT]; // SET_SIZE个valid(有效位)
reg                    dirty        [SET_SIZE][WAY_CNT]; // SET_SIZE个dirty(脏位)
```

2. 实现并行命中判断：为了判断是否命中，直接相连 cache 每次只需要判断一个 valid，一个 dirty，一个 TAG 是否命中，但组相连 cache 则需要在组内并行的判断每路 line 是否命中。

   首先添加一些 N 路选择信号：

```
reg [31:0]way_index; // 若命中，则值为命中的 line 的下标，否则为被替换的 line 下标
reg [31:0]mem_way_index; // 未命中，需要从内存中替换的块的下标
reg [31:0]history[SET_SIZE][WAY_CNT]; // 记录每个 line 的访问历史信息。在 FIFO 中，记录 line 被换入的时间；LRU 中，是记录上次访问的时间。
// 当一个块被访问（换入 or 命中）后，其history清零，其余块++。故而两种策略中，每次都选择 history 最大的块进行替换
```

   然后设计命中判断模块：根据 set_addr 确定组之后， for 循环遍历组中的每一路，判断是否满足命中条件，若不满足，按照替换策略，应该选择 history 最大的路作为换出块，下标存在 way_index 中。

```
// ---------- 判断 输入的 address 是否在 cache 中命中 ----------
always @ (*) begin
```

```
    cache_hit = 1'b0;
    way_index = 32'b0;
    for (integer i = 0;i < WAY_CNT;i++) begin
        if(valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) begin   // 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;
            way_index = i;
        end
    end
    if (cache_hit == 1'b0) begin
    // 未命中，选择 history 最大的块作为换出块，下标存在 way_index 中
        for (integer i = 0; i < WAY_CNT;i++) begin
            if (history[set_addr][way_index] < history[set_addr][i]) begin
                way_index = i;
            end
        end
    end

end
```

3. 实现替换策略：

   LRU：最近最少使用替换。即把访问次数最少的块替换掉。

   FIFO：先入先出替换。即把最先进入 cache 中的块替换掉。

   上述两种策略我们都可以用一个二维数组 history[set_addr][line_index] 来维护历史访问信息。针对不同的策略，用不同的方法修改 history 中的值即可。

   history 数组记录每个 line 的访问历史信息。在 FIFO 中，记录 line 被换入的时间；LRU 中，是记录上次访问的时间。当一个块被访问（换入 or 命中）后，其history清零，其余块++。故而两种策略中，每次都选择 history 最大的块进行替换

```
// ---------- 时序电路，维护 history[SET][WAY_CNT] 数组 ----------
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        for(integer i = 0; i < SET_SIZE; i++) begin
            for (integer j = 0;i < WAY_CNT; i++) begin
                history[i][j] <= 32'b0;
            end
        end
    end
    else begin
        if(LRU == 1'b1) begin
        // cache 替换策略选择 LRU
            if (en_signal && miss == 1'b0) begin
            // cache 命中
                for (integer i = 0; i < WAY_CNT;i++) begin
                    if(i == way_index) begin
                        history[set_addr][i] <= 32'b0;
                    end
                    else begin
                        history[set_addr][i] <= history[set_addr][i] + 1;
                    end
```

```
                end
            end
            else if (cache_stat == SWAP_IN_OK) begin
            // cache 未命中，要从内存写回
                for (integer i = 0; i < WAY_CNT;i++) begin
                    if(i == mem_way_index) begin
                        history[set_addr][i] <= 32'b0;
                    end
                    else begin
                        history[set_addr][i] <= history[set_addr][i]
+ 1;
                    end
                end
            end
        end
        else begin
        // cache 替换策略选择 FIFO
            if (cache_stat == SWAP_IN_OK) begin
            // 只有在块换入时修改 history 信息
                for (integer i = 0; i < WAY_CNT;i++) begin
                    if(i == mem_way_index) begin
                        history[set_addr][i] <= 32'b0;
                    end
                    else begin
                        history[set_addr][i] <= history[set_addr][i]
+ 1;
                    end
                end
            end
        end
    end
end
```

替换策略的改进：

用 for 循环从 WAY_CNT 个数中找最大值，至少需要三个时钟周期，电路执行效率较低。故可以做以下的策略改进。

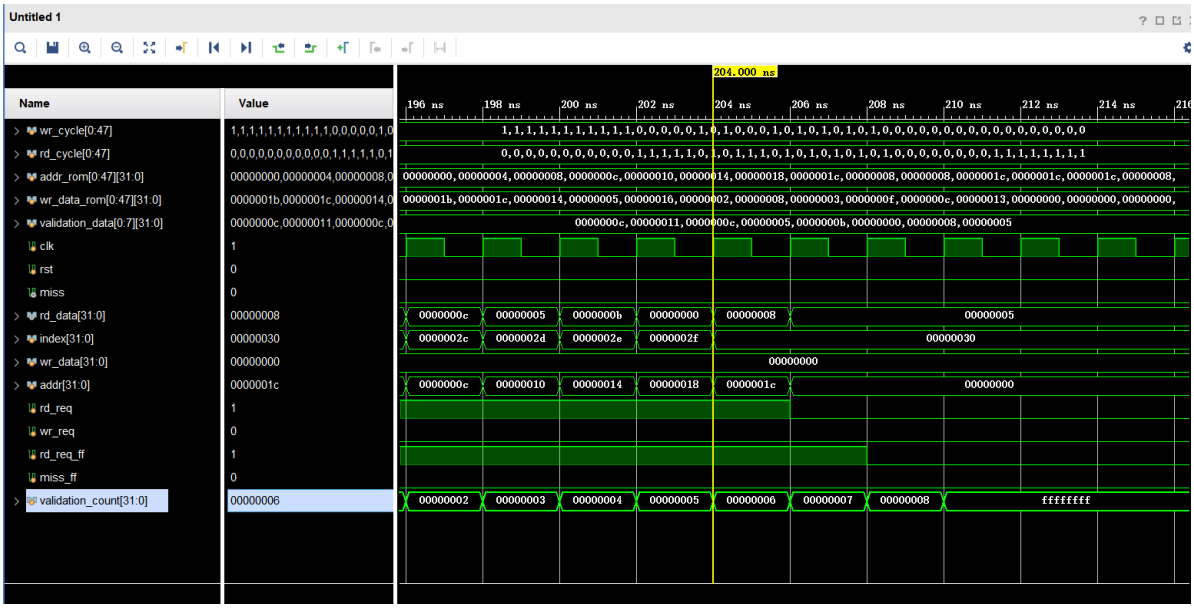
FIFO：保存下一个被换出块的指针，可以不用处理是否未被使用逻辑
LRU：当块被访问时，移到队尾，每次换成去队首。可以在一个时钟周期内完成

4. 实现一个边沿检测电路，来正确控制 cache 的引用。因为在实际的 cpu 使用中，cache miss 时，读（写）请求会长时间有效，但我们只需要在引用 cache 时更新状态。如果不加入边沿检测电路，会导致 cache 信息维护错误，从而导致 cache 不能正确工作。

```
// ---------- 对输入的每个 rd 或者 wr 只会产生一个周期的高平信号 en_signal
---------
// 因为 rd 和 wr 信号不止持续一个时钟周期，这里是防止 history 的值在一次访问
cache 的操作中被多次修改
reg en_signal,rec_signal;
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        en_signal <= 1'b0;
        rec_signal <= 1'b0;
    end
    else begin
        if(rd_req|wr_req) begin
            if (en_signal == 1'b0 && rec_signal == 1'b0) begin
```

```
                en_signal <= 1'b1;
                rec_signal <= 1'b1;
            end
            else if (en_signal == 1'b1 && rec_signal == 1'b1) begin
                en_signal <= 1'b0;
                rec_signal <= 1'b1;
            end
        end
        else begin
            en_signal <= 1'b0;
            rec_signal <= 1'b0;
        end
    end
end
```

cache测试的仿真波形：



图中可见，validation_count 最后结果为 -1，证明 cache 已经正确实现。

Cache 性能分析

在 WBSegReg 中，增添两个接口 miss_count，hit_count 用来统计 cache miss 的指令和 cache hit 的指令个数。

$$cache\_miss\_rate = miss\_count/(miss\_count + hit\_count)$$

Cache 资源占用

$$Cache\_Size = WAY\_CNT * SET\_SIZE$$

单独对 cache 文件综合，测试在其他参数不变，仅改变 cache 组相连度 `WAY_CNT` 时，列出 LRU 和 FIFO 两种策略的资源消耗。综合比较结果列在下表中。

电路资源	1路组相连，8组	2路组相连，8组	4路组相连，8组
LRU	Resource / Estimation / Available / Utilization % LUT 1129 63400 1.78 FF 3055 126800 2.41 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 2448 63400 3.86 FF 5674 126800 4.47 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization... LUT 5634 63400 8.89 FF 10398 126800 8.20 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13
FIFO	Resource / Estimation / Available / Utilization % LUT 1129 63400 1.78 FF 3055 126800 2.41 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 2448 63400 3.86 FF 5674 126800 4.47 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 5542 63400 8.74 FF 10397 126800 8.20 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13

固定 cache 大小不变，对组相连数和组数一起调整：

电路资源	1路组相连，8组	2路组相连，4组	4路组相连，2组
LRU	Resource / Estimation / Available / Utilization % LUT 1129 63400 1.78 FF 3055 126800 2.41 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 2140 63400 3.38 FF 3309 126800 2.61 BRAM 2 135 1.48 IO 80 210 38.10 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 2727 63400 4.30 FF 3314 126800 2.61 BRAM 1 135 0.74 IO 79 210 37.62 BUFG 1 32 3.13
FIFO	Resource / Estimation / Available / Utilization % LUT 1129 63400 1.78 FF 3055 126800 2.41 BRAM 4 135 2.96 IO 81 210 38.57 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 2129 63400 3.36 FF 3305 126800 2.61 BRAM 2 135 1.48 IO 80 210 38.10 BUFG 1 32 3.13	Resource / Estimation / Available / Utilization % LUT 2497 63400 3.94 FF 3306 126800 2.61 BRAM 1 135 0.74 IO 79 210 37.62 BUFG 1 32 3.13

从上述统计可以看出，LRU 策略和 FIFO 策略对 CPU 的资源占用并没有什么明显差别。这是因为我在实现两种策略时，都是采用 history 计数的方法，使得两种策略的电路设计相近，导致他们的电路资源消耗也相近。

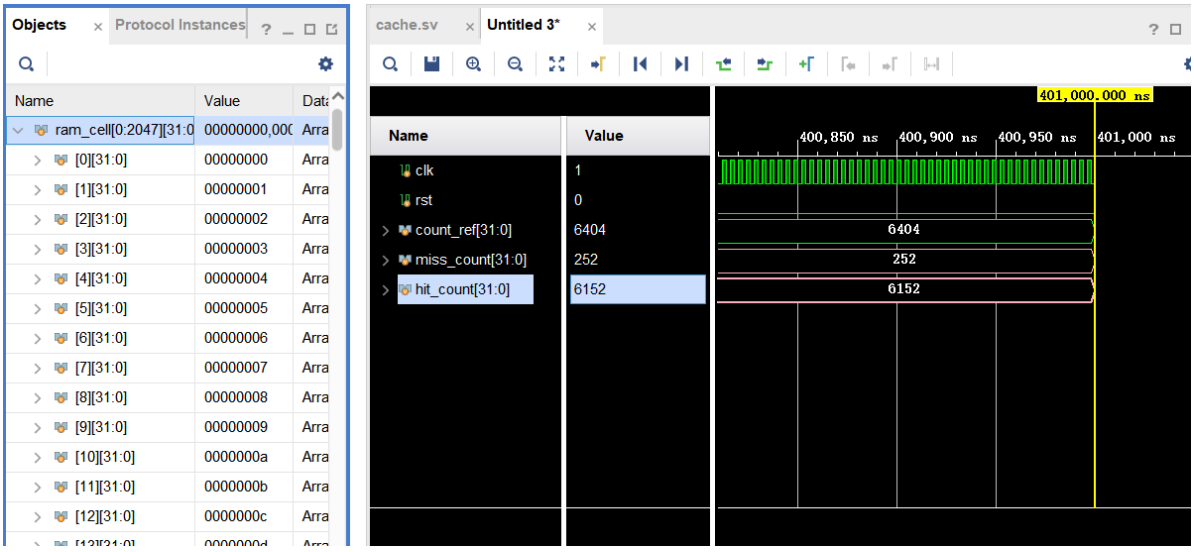
CPU Cache 缺失率统计

给实验结果全部截图不方便比较，在这里统一只各截一个仿真实例图作为程序正确执行的验证。

后续将计算好的缺失率以表格形式列出，方便观察和比较结果。

快速排序 256

结果仿真实例：



访存次数：6404

改变组相连度（cache 大小随之改变）：

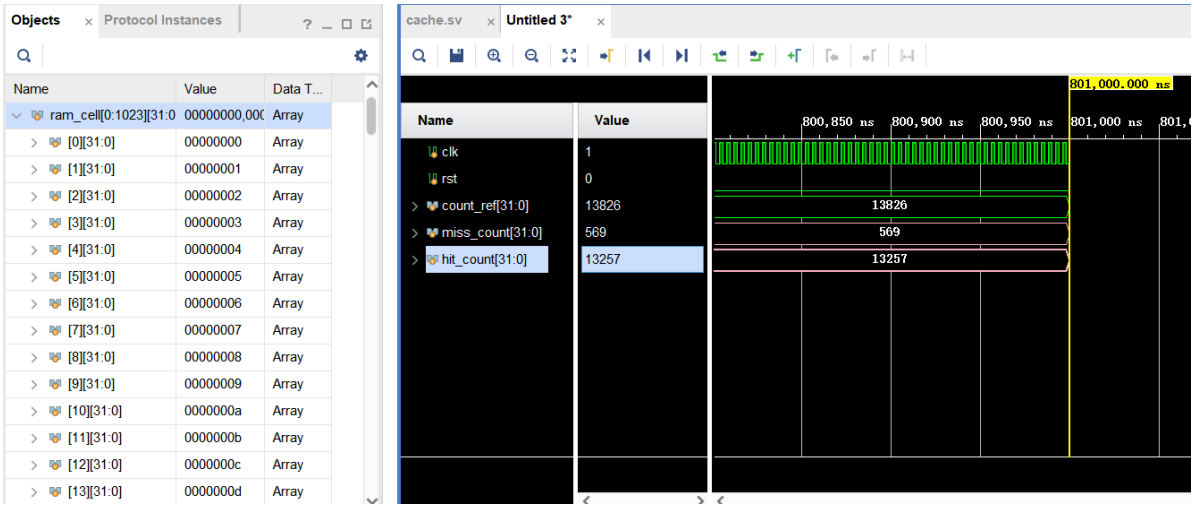
缺失率 （%）	1路组相连，8组	2路组相连，8组	4路组相连，8组
LRU	5.20	2.45	1.17
FIFO	5.20	2.14	1.03

保证 cache_size 不变，调整组相连度和组数

缺失率 （%）	1路组相连，8组	2路组相连，4组	4路组相连，2组
LRU	5.20	3.94	3.83
FIFO	5.20	3.51	3.58

快速排序 512

仿真结果实例：



访存次数：13826

改变组相连度（cache 大小随之改变）：

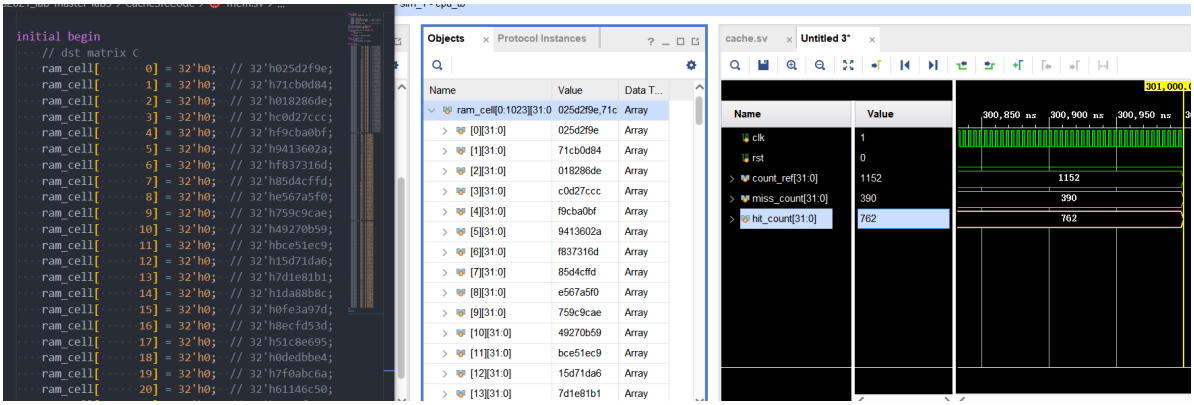
缺失率（%）	1路组相连，8组	2路组相连，8组	4路组相连，8组
LRU	5.77	2.75	1.87
FIFO	5.77	2.68	1.92

保证 cache_size 不变，调整组相连度和组数

缺失率（%）	1路组相连，8组	2路组相连，4组	4路组相连，2组
LRU	5.77	4.44	4.04
FIFO	5.77	4.03	4.12

矩阵乘法 8

仿真结果实例：



访存次数：1152

改变组相连度（cache 大小随之改变）：

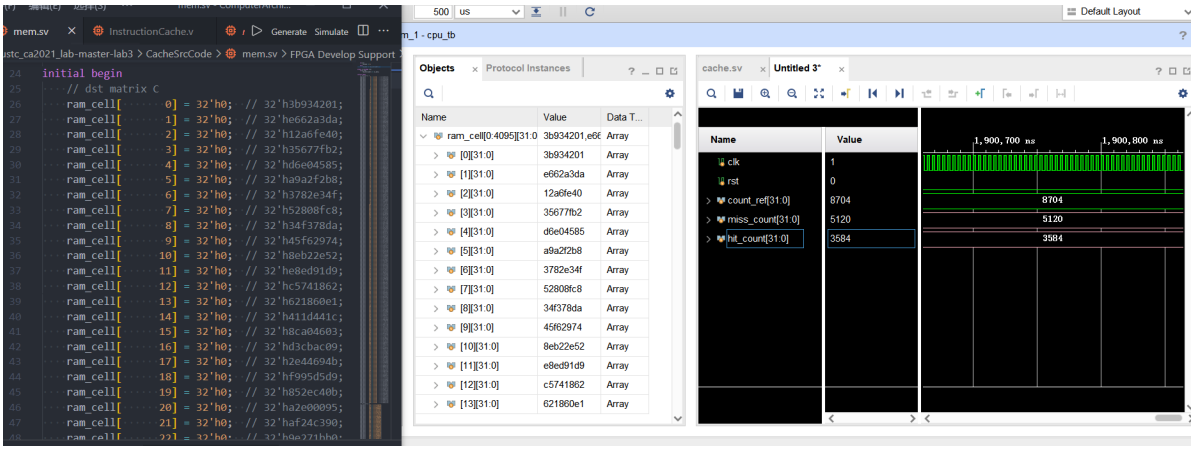
缺失率（%）	1路组相连，8组	2路组相连，8组	4路组相连，8组
LRU	23.35	17.19	2.60
FIFO	23.35	16.23	2.08

保证 cache_size 不变，调整组相连度和组数

缺失率（%）	1路组相连，8组	2路组相连，4组	4路组相连，2组
LRU	23.35	44.01	33.87
FIFO	23.35	26.39	33.74

矩阵乘法 16

仿真结果实例：



访存次数：8704

改变组相连度（cache 大小随之改变）：

缺失率（%）	1路组相连，8组	2路组相连，8组	4路组相连，8组
LRU	58.82	55.96	22.23
FIFO	58.82	55.88	19.98

保证 cache_size 不变，调整组相连度和组数

缺失率 （%）	1路组相连，8组	2路组相连，4组	4路组相连，2组
LRU	58.52	58.09	53.84
FIFO	58.52	58.09	49.38

总结

通过上面四个 benchmark 的测试，可以得出以下结论：

- 在只改变 cache 组相连度时，cache_size 随着组相连度的增大而增大，两个应用的cache 缺失率随着 cache 容量的增大而有大幅下降。
- 在保证 cache 容量不变，调整组相连度和组数时，随着组相连度的增加，两个应用的 cache 缺失率也有下降，但下降幅度较小。
- 在矩阵乘法中，FIFO 略好于 LRU，在快速排序中，两种策略的缺失率各有高低。

结论

不管是从理论层面还是实验层面，cache 的缺失率都和 cache 的容量、组相连度有关。增大 cache 容量能明显降低 cache 缺失率。在 cache 容量有限的情况下，增大组相连度，减少组数，也能在小范围内降低缺失率。

对快速排顺序应用，建议使用 “4路组相连，8组，FIFO”的策略，可以有效降低 cache 缺失率，且 cache 资源占用较少。

对矩阵乘法应用，建议使用“4路组相连，8组，FIFO”的策略，可以有效降低 cache 缺失率，且 cache 资源占用较少。