

lab3 实验报告

雷雨轩 PB18111791 计算机学院

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

- Vscode 2021
- Vivado 2019.1

实验内容

- 阶段一：理解我们提供的直接映射策略的cache，将它修改为N路组相连的cache，并通过我们提供的cache读写测试。
- 阶段二：使用阶段一编写的N路组相连cache，正确运行我们提供的几个程序。
- 阶段三：对不同cache策略和参数进行性能和资源的测试评估，编写实验报告。

实验过程

阶段1

实现思路

- 首先要把cache.sv的代码原理读懂，然后在此基础上，来实现N路组相连cache，分为FIFO和LRU两种策略

```
module cache #(
    parameter LINE_ADDR_LEN = 3, // line内地址长度，决定了每个line具有2^3个word
    parameter SET_ADDR_LEN = 3, // 组地址长度，决定了一共有2^3=8组
    parameter TAG_ADDR_LEN = 6, // tag长度
    parameter WAY_CNT = 3 // 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
)()
```

输入参数里比较重要的即是WAY_CNT和SET_ADDR_LEN,表示每组中有多少路line以及共有多少个组

其次即是要知道代码里的主要逻辑部分即是状态机的转移过程

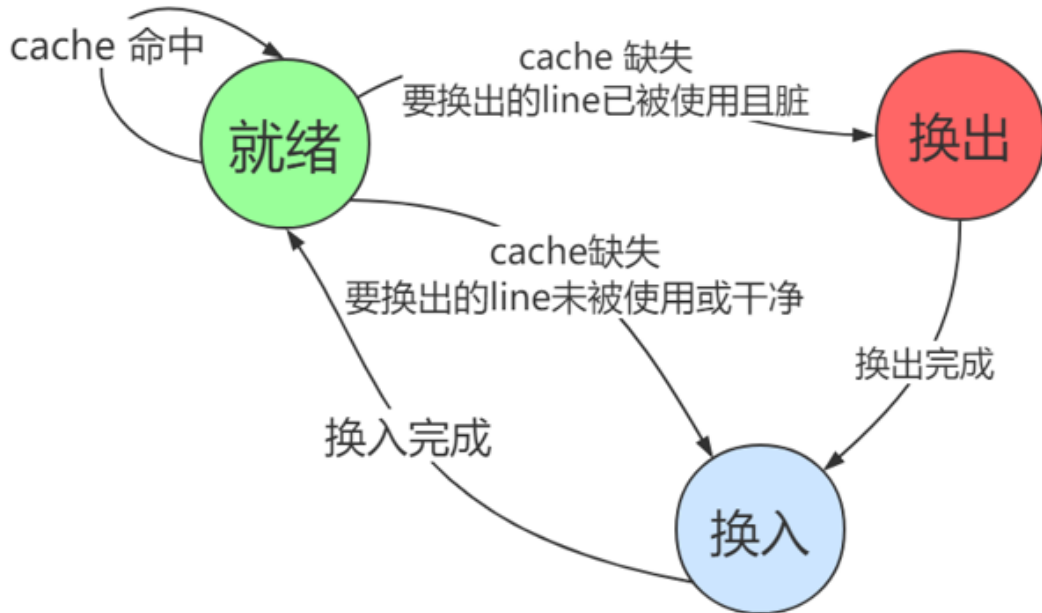


图 1: cache 状态机

- 首先, 要把直接相连改为N路组相连, 那么需要把cache_mem,cache_tags,valid,dirty变量都增加一个维度,这是因为现在在一个set里有多路。

```
reg [31:0] cache_mem [SET_SIZE][WAY_CNT][LINE_SIZE]; //
SET_SIZE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)
```

此后的访问里, 也需要根据维度的变化把这几个变量的访问代码做一定调整

- 此外, 还需要增加如下几个变量。其含义见注释

```
reg [31:0] way_idx; // way_idx 变量, 当命中时, 这个变量为该组中某个line的下标, 当未命中
时, 这个变量为要换出的line的下标
reg [31:0] mem_way_idx;
reg [31:0] history[SET_SIZE][WAY_CNT]; // 每个line的历史信息, 即该line最近是何时被
使用过(LRU), 或者是何时被换入(FIFO)
localparam LRU = 1'b0;
integer cnt; //记录当前是第几个周期
```

其中我们用cnt记录当前运行到第几个周期了, 然后用history记录每个line最近是在哪个周期被访问的, 这个需要根据策略是FIFO还是LRU来动态更新(策略即用LRU参数来标识, 为1的时候是LRU, 为0则是FIFO)

- 接着需要在原来的代码基础上, 做一些修改

这一块代码是在判断到底是否命中, 利用for循环实现并行比较cache_tags, 并且若命中, 那么cache_hit=1, way_idx记录是哪一路命中了;

若没命中, 那么找到history[set_addr]数组里, 数值最小的那个, 相当于其即为需要被替换出去的块

```

always @ (*) begin
    cache_hit = 1'b0;
    way_idx = 32'b0;
    for(integer i = 0; i < WAY_CNT; i++) begin
        if(valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) // 如果 cache line有效, 并且tag与输入地址中的tag相等, 则命中
            begin//
                cache_hit = 1'b1;
                way_idx = i;
            end
        end
    end
    if (cache_hit == 1'b0) begin
        // 若cache未命中, way_idx是history中选出的下标, 是LRU或者FIFO状态机选出的要替换的下标
        for (integer i = 0; i < WAY_CNT; ++i)
            begin
                if(history[set_addr][way_idx] > history[set_addr][i])
                    begin
                        way_idx = i;
                    end
            end
    end
end
end

```

- 另一块要修改的即是何时更新history变量

```

case(cache_stat)
IDLE: begin
    if(cache_hit) begin
        if(wr_req | rd_req)begin
            if(LRU==1'b1)begin //LRU
                history[set_addr][way_idx]<=cnt;
            end
        end
    end
end

```

如上图, 当在就绪阶段, 且cache命中, 并且有读或写请求, 此时说明该路数据会被访问, 此时LRU的记录需要更新。

此外, 当在SWAP_IN_OK阶段时, 因为此时相当于刚换入成功一个新的line, 所以无论FIFO还是LRU都需要更新history

```

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
    for(integer i=0; i<LINE_SIZE; i++) cache_mem[mem_rd_set_addr][way_idx][i] <= mem_rd_line[i];
    cache_tags[mem_rd_set_addr][mem_way_idx] <= mem_rd_tag_addr;
    valid [mem_rd_set_addr][mem_way_idx] <= 1'b1;
    dirty [mem_rd_set_addr][mem_way_idx] <= 1'b0;
    history[mem_rd_set_addr][mem_way_idx]<= cnt; //LRU或者FIFO
    cache_stat <= IDLE; // 回到就绪状态
end
endcase

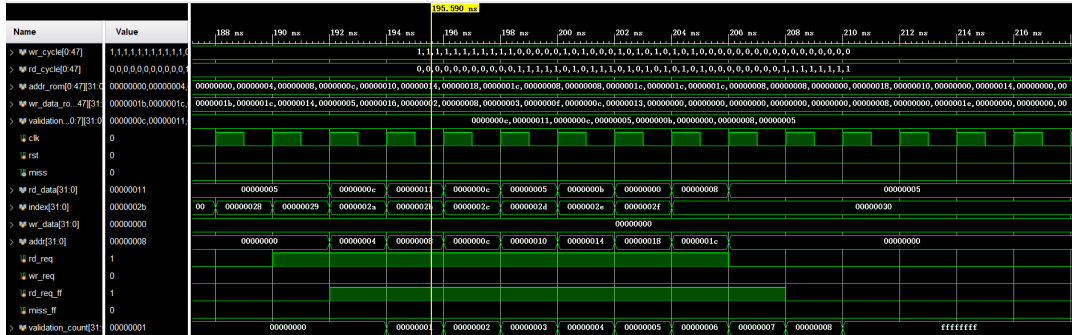
```

运行结果

- 由助教提供的generate_cache_tb.py文件, 可以生成不同N的cache_tb.sv文件, 我们可以取N=8、N=16等
- 仿真结果
 - LRU



- FIFO



- 可以看到，validation_count都是从0增加到8，最后变为0xffffffff，说明代码功能正确

阶段2

实现思路

- 即是要将阶段1里实现的cache与lab2里实现的CPU连接起来。经过分析，需要修改的文件有：

RV32Core.v IDSegReg.v HazardUnit.v WBSegReg.v

HazardUnit.v

- 比较简单，只需增加ICacheMiss和DCacheMiss，并且在有其中一个miss发生时，stall一个周期

```
always @ (*)
    if(CpuRst)//当CpuRst==1时CPU全局复位清零（所有段寄存器flush）
    {StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <= 10'b0101010101;
    else if(DCacheMiss | ICacheMiss)
    {StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <= 10'b1010101010;
```

IDSegReg.v

- 根据实验指导所述，生成对应的InstructionCache.v，对比其中的接口，修改IDSegReg.v文件即可

```
InstructionCache cache_2 (
    .clk          ( clk          ),
    .write_en     ( |WE2        ),
    .addr         ( A[31:2]     ),
    .debug_addr   ( A2[31:2]    ),
    .data         ( RD_raw      ),
    .debug_input  ( WD2         ),//write
    .debug_data   ( RD2         )//read
);
```

WBSegReg.v

- 这里，一方面要把原来的dataram换为新写的cache，另一方面要统计hit_count以及miss_count
- cache模块的调用只需对应cache.sv里的接口即可，不过需要在WBSegReg.v增加一个CacheMiss的输出信号

```
cache #(
    .LINE_ADDR_LEN    ( 3 ),
    .SET_ADDR_LEN     ( 3 ),
    .TAG_ADDR_LEN     ( 6 ),
    .WAY_CNT           ( 3 )
) cache_1 (
    .clk               ( clk ),
    .rst               ( rst ),
    .miss              ( CacheMiss ),
    .addr              ( A ),
    .rd_req            ( MemToRegM ),
    .rd_data            ( RD_raw ),
    .wr_req            ( |WE ),
    .wr_data            ( WD )
);
```

- 统计缺失率这一块

```
reg [31:0] hit_count = 0, miss_count = 0; // 计算cache命中和缺失次数
reg [31:0] last_addr = 0; //
wire cache_rd_wr = (|WE) | MemToRegM; // cache有读或写操作时置1
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        last_addr <= 0;
    end else begin
        if( cache_rd_wr ) begin
            last_addr <= A;
        end
    end
end

always @ (posedge clk or posedge rst) begin
    if(rst) begin
        hit_count <= 0;
        miss_count <= 0;
    end else begin
        if( cache_rd_wr & (last_addr!=A) ) begin //仅在对cache有读或写操作时的第一个周期才会计算，此后last_addr==A，不会重复计数
            if(CacheMiss)
                miss_count <= miss_count+1;
            else
                hit_count <= hit_count +1;
        end
    end
end
```

```
end  
end
```

注意到，一旦cacheMiss发生时，那么cache_rd_wr信号会一直有效，所以为了使得miss_count不重复计数，引入一个变量last_addr，其仅在对cache有读或写操作时的第一个周期会发生值变化，更新为此次访存的地址，并且仅在这一刻才会进行miss_count和hit_count的计数。之后因为cacheMiss造成访存等待的周期里，last_addr==A，所以不会再重复计数。

RV32Core.v

- 顶层模块，只需要修改对应的接口的参数之类的即可，主要是调用WBSegReg和HazardUnit时
- 由实验说明，不考虑指令miss，所以默认设其为0

```
HarzardUnit HarzardUnit1(  
    .CpuRst(CPU_RST),  
    .BranchE(BranchE),  
    .JalrE(JalrE),  
    .JalD(JalD),  
    .Rs1D(Rs1D),  
    .Rs2D(Rs2D),  
    .Rs1E(Rs1E),  
    .Rs2E(Rs2E),  
    .RegReadE(RegReadE),  
    .MemToRegE(MemToRegE),  
    .RdE(RdE),  
    .RdM(RdM),  
    .RegWriteM(RegWriteM),  
    .RdW(RdW),  
    .RegWriteW(RegWriteW),  
    .ICacheMiss(1'b0),  
    .DCacheMiss(DCacheMiss),
```

- WBSegReg则需要增加一个输出信号，即DCacheMiss
-

























```

WBSegReg WBSegReg1(
    .clk(CPU_CLK),
    .rst(CPU_RST),
    .en(~StallW),
    .clear(FlushW),
    .CacheMiss(DCacheMiss),
    .A(AluOutM),
    .WD(StoreDataM),
    .WE(MemWriteM),
    .RD(DM_RD),
    .LoadedBytesSelect(LoadedBytesSelect),
    .A2(CPU_Debug_DataRAM_A2),
    .WD2(CPU_Debug_DataRAM_WD2),
    .WE2(CPU_Debug_DataRAM_WE2),
    .RD2(CPU_Debug_DataRAM_RD2),
    .ResultM(ResultM),
    .ResultW(ResultW),
    .RdM(RdM),
    .RdW(RdW),
    .RegWriteM(RegWriteM),
    .RegWriteW(RegWriteW),
    .MemToRegM(MemToRegM),
    .MemToRegW(MemToRegW)
);

























```

运行结果

- 根据实验指导，对快排和矩阵乘进行测试
- FIFO快速排序，可以看到ram里结果已排好序

| | | |
|---|---------------------|-------|
| ▼  ram_cell[0:4095][31:0] | 00000000,00000001,C | Array |
| >  [0][31:0] | 00000000 | Array |
| >  [1][31:0] | 00000001 | Array |
| >  [2][31:0] | 00000002 | Array |
| >  [3][31:0] | 00000003 | Array |
| >  [4][31:0] | 00000004 | Array |
| >  [5][31:0] | 00000005 | Array |
| >  [6][31:0] | 00000006 | Array |
| >  [7][31:0] | 00000007 | Array |
| >  [8][31:0] | 00000008 | Array |
| >  [9][31:0] | 00000009 | Array |
| >  [10][31:0] | 0000000a | Array |
| >  [11][31:0] | 0000000b | Array |
| >  [12][31:0] | 0000000c | Array |
| >  [13][31:0] | 0000000d | Array |
| >  [14][31:0] | 0000000e | Array |
| >  [15][31:0] | 0000000f | Array |
| >  [16][31:0] | 00000010 | Array |
| >  [17][31:0] | 00000011 | Array |
| >  [18][31:0] | 00000012 | Array |
| >  [19][31:0] | 00000013 | Array |
| >  [20][31:0] | 00000014 | Array |
| >  [21][31:0] | 00000015 | Array |
| >  [22][31:0] | 00000016 | Array |

- LRU矩阵乘：可以对比mem.sv里的注释，发现矩阵乘结果是正确的

| | | |
|---|---------------------|-------|
| ▼  ram_cell[0:4095][31:0] | 068b5899,b078a2d2,f | Array |
| >  [0][31:0] | 068b5899 | Array |
| >  [1][31:0] | b078a2d2 | Array |
| >  [2][31:0] | f0d2f875 | Array |
| >  [3][31:0] | 45b8f052 | Array |
| >  [4][31:0] | f27c7987 | Array |
| >  [5][31:0] | 74430d0b | Array |
| >  [6][31:0] | e1739a30 | Array |
| >  [7][31:0] | 9b8c4edc | Array |
| >  [8][31:0] | ee8873f1 | Array |
| >  [9][31:0] | c49045fa | Array |
| >  [10][31:0] | 7cf9ed4d | Array |
| >  [11][31:0] | c0126625 | Array |
| >  [12][31:0] | 4aa649e1 | Array |
| >  [13][31:0] | 0161e4b2 | Array |
| >  [14][31:0] | fbbf9b9e | Array |
| >  [15][31:0] | 3759b2f8 | Array |
| >  [16][31:0] | dfdcf7ed | Array |
| >  [17][31:0] | 36794929 | Array |
| >  [18][31:0] | c7961eb6 | Array |
| >  [19][31:0] | 35f5f318 | Array |
| >  [20][31:0] | c729c403 | Array |
| >  [21][31:0] | 71da519c | Array |
| >  [22][31:0] | 03261e4d | Array |

阶段3

体会**cache size**、**组相连度**、**替换策略**针对不同程序的优化效果，以及策略改变带来的电路面积的变化。针对不同程序，**权衡性能和电路面积给出一个较优的cache参数和策略**。其中“性能”参数使用运行仿真时的**时钟周期数量**进行评估。“资源占用”参数使用**vivado或其它综合工具给出的综合报告进行评估**。进行这一步时需要用阶段一的结果进行一些实验，不能仅仅进行理论分析，实验报告中需要给出实验结果（例如仿真波形的截图、vivado综合报告等）。

cache资源占用对比

这一部分是使用阶段1里的代码进行比较。通过修改各个参数来做资源占用的对比。

- 注意到，当修改这些参数时，cache规模会发生变化，主存也会。在进行实验时，为了排除主存大小对资源占用的影响，可能需要固定主存的大小。主存大小是 $2^{(\text{LINE_ADDR_LEN} + \text{SET_ADDR_LEN} + \text{TAG_ADDR_LEN})}$ 个字。当你将SET_ADDR_LEN或LINE_ADDR_LEN改大时，TAG_ADDR_LEN就要改小，这样就能保证主存的大小不变。

- LUT、FF：是我们最在意的资源，因为cache的逻辑均使用LUT和FF实现。这两个参数的使用量就代表了你的cache所占用电路的资源量。

BRAM：主存main_mem被综合成了BRAM。由于我们不对主存进行修改，所以这一项不需要在意。

LRU

对应下表第一行

| Resource | Utilization | Available | Utilization % |
|----------|-------------|-----------|---------------|
| LUT | 3626 | 63400 | 5.72 |
| FF | 6015 | 126800 | 4.74 |
| BRAM | 4 | 135 | 2.96 |
| IO | 81 | 210 | 38.57 |

对应下表第二行

| Resource | Utilization | Available | Utilization % |
|----------|-------------|-----------|---------------|
| LUT | 2951 | 63400 | 4.65 |
| FF | 5716 | 126800 | 4.51 |
| BRAM | 4 | 135 | 2.96 |
| IO | 81 | 210 | 38.57 |

其余依次测量，结果如下表

| LINE_ADDR_LEN | SET_ADDR_LEN | TAG_ADDR_LEN | WAY_CNT | LUT | FF |
|---------------|--------------|--------------|---------|------|-------|
| 2 | 3 | 7 | 4 | 3626 | 6015 |
| 3 | 3 | 6 | 2 | 2951 | 5716 |
| 3 | 3 | 6 | 4 | 5312 | 10455 |
| 3 | 3 | 6 | 8 | 9602 | 19885 |
| 3 | 2 | 7 | 4 | 4112 | 5821 |
| 3 | 4 | 5 | 4 | 9520 | 19849 |
| 4 | 3 | 5 | 4 | 9560 | 19401 |

FIFO

对应下表第一行

| Resource | Utilization | Available | Utilization % |
|----------|-------------|-----------|---------------|
| LUT | 2947 | 63400 | 4.65 |
| FF | 6008 | 126800 | 4.74 |
| BRAM | 4 | 135 | 2.96 |
| IO | 81 | 210 | 38.57 |

对应下表第二行

| Resource | Utilization | Available | Utilization % |
|----------|-------------|-----------|---------------|
| LUT | 2369 | 63400 | 3.74 |
| FF | 5713 | 126800 | 4.51 |
| BRAM | 4 | 135 | 2.96 |
| IO | 81 | 210 | 38.57 |

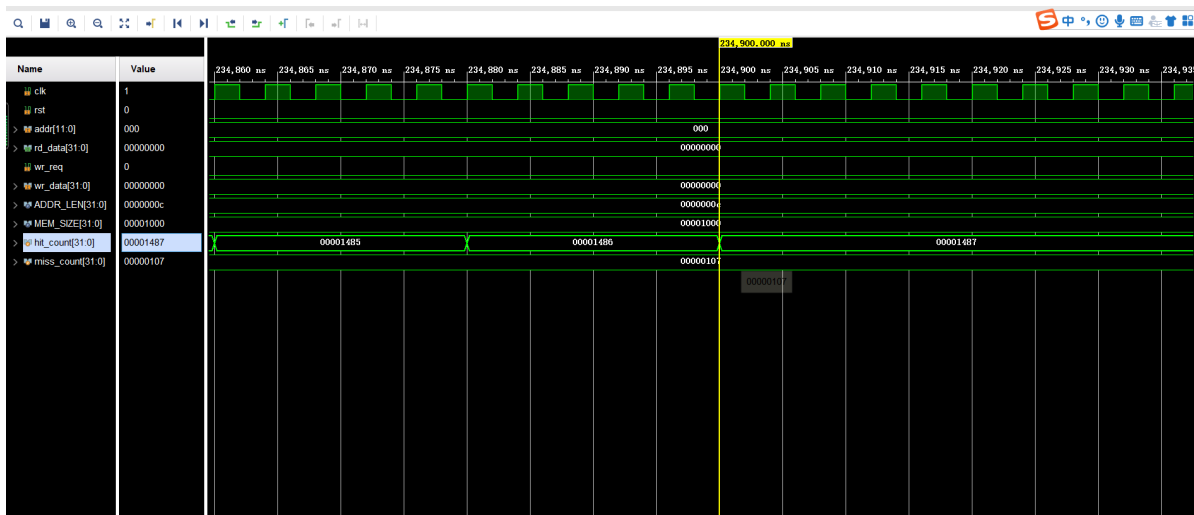
其余依次测量，结果如下表

| LINE_ADDR_LEN | SET_ADDR_LEN | TAG_ADDR_LEN | WAY_CNT | LUT | FF |
|---------------|--------------|--------------|---------|------|-------|
| 2 | 3 | 7 | 4 | 2947 | 6008 |
| 3 | 3 | 6 | 2 | 2369 | 5713 |
| 3 | 3 | 6 | 4 | 4001 | 10450 |
| 3 | 3 | 6 | 8 | 9030 | 20005 |
| 3 | 2 | 7 | 4 | 3585 | 5730 |
| 3 | 4 | 5 | 4 | 7102 | 19800 |
| 4 | 3 | 5 | 4 | 8420 | 19432 |

- 由上述表可知
 - 主存大小不变时
 - 组相连度WAY_CNT越大，占用的LUT和FF越多，电路面积越大
 - 每个line越大（LINE_ADDR_LEN），占用的LUT和FF越多，电路面积越大
 - 组数SET_ADDR_LEN越大，占用的LUT和FF越多，电路面积越大
 - 在同样的参数条件下，LRU和FIFO所占用的FF数基本一致，但LRU的LUT更大，说明其cache电路面积更大一些

cache性能对比

以FIFO+快排256为例，其表里第一行的仿真结果如下

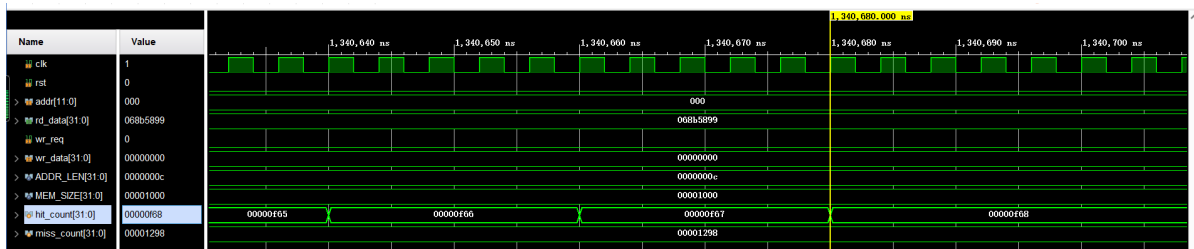


我们需要观察的即是运行时间，以及稳定时的hit_count和miss_count

FIFO+快排 256

| LINE_ADDR_LEN | SET_ADDR_LEN | TAG_ADDR_LEN | WAY_CNT | Times(ns) | Miss | Hit | Miss Rate |
|---------------|--------------|--------------|---------|-----------|------|------|-----------|
| 2 | 3 | 7 | 4 | 234900 | 263 | 5255 | 4.77% |
| 3 | 3 | 6 | 2 | 190300 | 140 | 5378 | 2.54% |
| 3 | 3 | 6 | 4 | 156800 | 60 | 5458 | 1.09% |
| 3 | 3 | 6 | 8 | 140980 | 45 | 5473 | 0.82% |
| 3 | 2 | 7 | 4 | 193920 | 150 | 5368 | 2.72% |
| 3 | 4 | 5 | 4 | 143800 | 41 | 5477 | 0.74% |
| 4 | 3 | 5 | 4 | 140108 | 20 | 5498 | 0.36% |

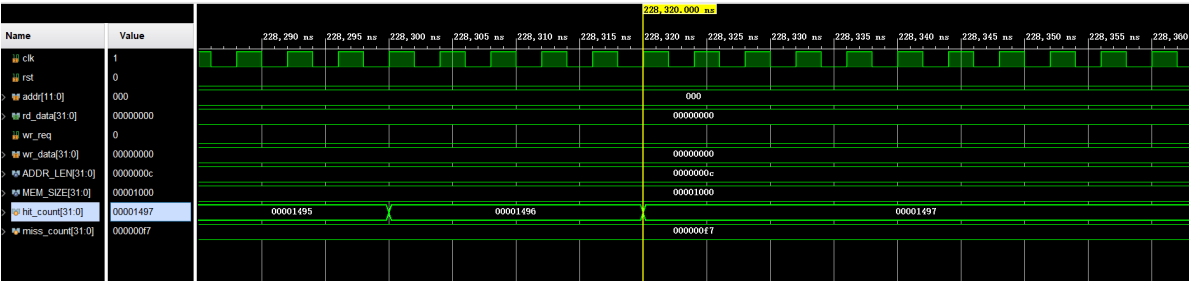
FIFO+矩阵乘 16*16



| LINE_ADDR_LEN | SET_ADDR_LEN | TAG_ADDR_LEN | WAY_CNT | Times(ns) | Miss | Hit | Miss Rate |
|---------------|--------------|--------------|---------|-----------|------|------|-----------|
| 2 | 3 | 7 | 4 | 1340680 | 4760 | 3944 | 54.69% |
| 3 | 3 | 6 | 2 | 1363112 | 4864 | 3840 | 55.88% |
| 3 | 3 | 6 | 4 | 687904 | 1739 | 6965 | 19.98% |
| 3 | 3 | 6 | 8 | 294576 | 146 | 8558 | 1.68% |
| 3 | 2 | 7 | 4 | 1349288 | 4800 | 3904 | 55.14% |
| 3 | 4 | 5 | 4 | 293728 | 144 | 8560 | 1.65% |
| 4 | 3 | 5 | 4 | 276512 | 72 | 8632 | 0.83% |

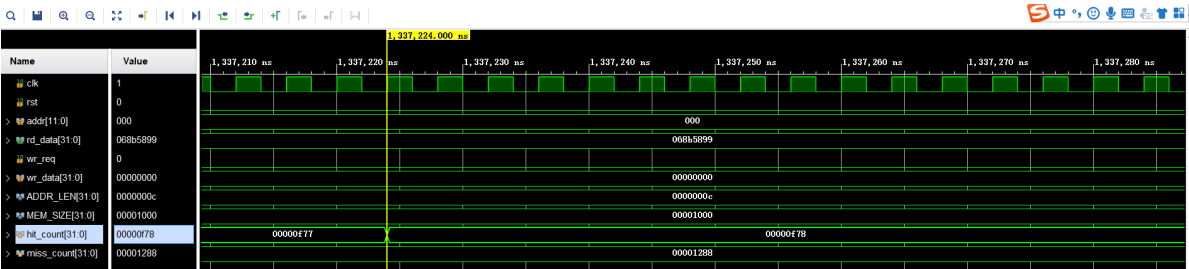
LRU+快排 256

第一行结果



| LINE_ADDR_LEN | SET_ADDR_LEN | TAG_ADDR_LEN | WAY_CNT | Times(ns) | Miss | Hit | Miss Rate |
|---------------|--------------|--------------|---------|-----------|------|------|-----------|
| 2 | 3 | 7 | 4 | 228320 | 247 | 5271 | 4.48% |
| 3 | 3 | 6 | 2 | 186300 | 130 | 5388 | 2.36% |
| 3 | 3 | 6 | 4 | 160500 | 80 | 5438 | 1.45% |
| 3 | 3 | 6 | 8 | 143600 | 45 | 5473 | 0.82% |
| 3 | 2 | 7 | 4 | 189003 | 146 | 5372 | 2.65% |
| 3 | 4 | 5 | 4 | 143740 | 45 | 5473 | 0.82% |
| 4 | 3 | 5 | 4 | 139440 | 24 | 5494 | 0.43% |

LRU+矩阵乘 16*16



| LINE_ADDR_LEN | SET_ADDR_LEN | TAG_ADDR_LEN | WAY_CNT | Times(ns) | Miss | Hit | Miss Rate |
|---------------|--------------|--------------|---------|-----------|------|------|-----------|
| 2 | 3 | 7 | 4 | 1337224 | 4744 | 3960 | 54.50% |
| 3 | 3 | 6 | 2 | 1321640 | 4672 | 4032 | 53.68% |
| 3 | 3 | 6 | 4 | 646432 | 1547 | 7157 | 17.77% |
| 3 | 3 | 6 | 8 | 291484 | 123 | 8581 | 1.41% |
| 3 | 2 | 7 | 4 | 1321640 | 4672 | 4032 | 53.68% |
| 3 | 4 | 5 | 4 | 289788 | 119 | 8584 | 13.67% |
| 4 | 3 | 5 | 4 | 273056 | 56 | 8648 | 0.64% |

- 首先，其他参数相同时，矩阵乘的cache缺失率和运行时间都远大于快排，这可能是因为两种应用程序的时间复杂度、规模以及程序局部性不同导致的。
- 保持主存大小不变时，可以发现
 - 组相连度WAY_CNT越大，运行时间越短，cache缺失率越小
 - 每个line越大（LINE_ADDR_LEN），运行时间越短，cache缺失率越小
 - 组数SET_ADDR_LEN越大，运行时间越短，cache缺失率越小
- 权衡性能和电路面积：
 - 对于矩阵乘程序：局部性很差，再观察表格发现，LRU性能整体优于FIFO，所以优先考虑LRU

- 在性能方面，比较直观的可以看到，参数组合(4,3,5,4)的性能最优，其运行时间最低，且cache缺失率也最低
- 唯一能与其相提并论的只有(3,3,6,8)组合，并且可以发现两者的电路面积非常相近
- 所以，综合考虑，选择参数组合(4,3,5,4),以及LRU策略
- 对于快排程序，其局部性本身较好，cache缺失率都很低，使用LRU和FIFO区别不大，并且由于LRU电路面积更大，所以综合考虑，应采用FIFO策略
- 在性能方面，比较直观的可以看到，参数组合(4,3,5,4)的性能最优，其运行时间最低，且cache缺失率也最低
- 当然，（3,4,5,4）和（3,3,6,8）的组合也能取得不错的性能
- 再结合电路面积来考虑：上述3种组合的电路面积差不多，但比其他参数组合的电路面积都大一截
- 所以说，综合考量，应该考虑参数：(4,3,5,4)组合，且使用FIFO策略。（当然，如果对电路面积有限制，那可能需要再看其余的组合）

总结

- 通过本次实验，自己对于cache的内部实现有了更深刻的认识，也完全区分了set、way等概念的区别
- 通过阶段3的分析，自己也是发现，不同程序因为存在不同的局部性，所以会导致cache性能有所不同；此外，cache不同的配置参数也会影响cache性能
- 通过阶段3的多次模拟、测试、记录、思考，明显发现LRU策略资源占用要多一些，但性能会更好一些，这也与理论分析一致，可以看出不同策略都有不同的特点，要根据实际情况灵活选择。

改进意见

暂无