cache实验报告

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：win10

内存：16GB

编辑器：VSCode

仿真环境：Vivado 2019.1

实验内容

实验实现过程

对直接映射cache进行仿真；

实现FIFO策略；

实现LRU策略；

加入非CSR指令的CPU

全局信号的设计

缓存的ram设置如下：

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

于是所有的缓存的大小均为SET_SIZE*WAY_CNT*LINE_SIZE个word。

缓存中其他相关的寄存器如下：

```
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE][WAY_CNT];
reg                    valid        [SET_SIZE][WAY_CNT];
reg                    dirty        [SET_SIZE][WAY_CNT];
```

开一个寄存器数组：

```
int fifo_queue[SET_SIZE][3]; // for FIFO

int lru_queue[SET_SIZE][3]; // for LRU
```

注意这个queue每个元素的位数为WAY_CNT位，这可能存在冗余，事实上只要lg(WAY_CNT)位，但是不知道为什么计算lg的系统函数不被vivado 2019.1支持。于是设了这个冗余。

在组相连映射中，我设计了如下数据结构来记录信息：

```
reg [WAY_CNT-1:0] hit_flag; // 记录组内的哪一个块被命中，如果命中对应的位为1，否则为0
reg [WAY_CNT-1:0] line_hit_addr; // 记录命中的块的组内地址,在下面的循环中被赋值
wire cache_hit = | hit_flag; // 缓存命中信号=所有块命中信息的或
always @ (*) begin              // 判断 是否命中
    for (integer i=0; i<WAY_CNT; i++) begin
        if (valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) begin
            //判断每一路是否命中
            hit_flag[i]=1;
            line_hit_addr=i;
```

```
            end
            else begin
                hit_flag[i]=0;
            end
        end
    end
```

下面介绍实现的俩种置换策略。

注： 我用了俩种数据结构实现的FIFO，为了简单起见，主要介绍实现起来更简单的第二种。

FIFO算法的描述

1. 初始(rst==1)时，把每个组的所有的块的地址按照从小到大的顺序压入fifo_queue
2. 在状态`IDLE`中，如果成功命中了，则用`line_hit_addr`所代表的line参与读写；如果没有命中，则用`fifo_queue[set_addr][0]`来参与写回。注意，对于刚刚初始化之后的缓存来说，这里恒为0号块。对于运行了一段时间的缓存，这里的块号应该是最早被压入队列的块号。
3. 在状态`SWAP_IN_OK`中，块换入，用`fifo_queue[set_addr][0]`所代表的的line来参与写入cache。
4. 在状态`SWAP_IN_OK`中，根据FIFO策略，当一个块换入后，其是最新压入的块，故应该在其队列的队尾。如下实现：

   ```
   for(integer j =1;j<=WAY_CNT;j++)  begin
      fifo_queue[set_addr][(j-1)%WAY_CNT]<=fifo_queue[set_addr][j%WAY_CNT];
   end
   ```

   注意这里fifo_queue[set_addr][0]是最近被压入的块，其应该在队尾。按照非阻塞赋值的特点，上面的代码可以实现一个循环队列，每次可以将队头压到队尾，并且资源开销非常小。

LRU算法的描述

由于FIFO算法的实现中我使用了类似于数据结构中的队列的概念，在LRU算法实现时我开始考虑，也许根本就不需要计算复杂的每个块被命中之后经过了多少时间。前述做法大大增加了资源消耗和每个周期缓存的工作量，使缓存性能低下。我基于我的考虑，在FIFO的队列的基础上，实现了一个还有一些优化空间的优先级队列。利用这个优先级队列，我可以在不计算时间，仅仅记录每个块之间的优先级关系的基础上实现LRU算法。

1. 初始(rst==1)时，把每个组的所有的块的地址按照从小到大的顺序压入lru_queue
2. 在状态`IDLE`中，如果成功命中了，则用`line_hit_addr`所代表的line参与读写，同时需要更新这个刚刚被命中的块的时间(优先级最高，优先级表示了需要被保留的程度)。考虑在传统的LRU中，被命中的块的时间应该被设置为0，于是我把刚被命中的块的优先级设为最高(放在我的队尾)：

```
if(rd_req|wr_req) begin // 如果发生了缓存读写请求，才需要去判断
    for (integer i=0; i<WAY_CNT; i++) begin // 遍历每一个块
        if(lru_queue[set_addr][i]==int'(line_hit_addr)) begin
            // 如果命中了队列第i个元素指向的块
            for(integer j =i+1;j<WAY_CNT;j++) begin
                // 就把从这第i个元素开始，到这个组的优先级队列队尾视作一个循环队列
                lru_queue[set_addr][j-1]<=lru_queue[set_addr][j];
            end
            lru_queue[set_addr][WAY_CNT-1]<=lru_queue[set_addr][i];
            // 上面的几行代码实现了把之前命中的块压倒了队尾，之间的块前移一位(优先级下
降)
        end
   end
end
```

如果没有命中，则用`lru_queue[set_addr][0]`来参与写回。注意，对于刚刚初始化之后的缓存来说，这里恒为0号块。对于运行了一段时间的缓存，这里的块号应该是最近最远被使用的块。这是因为每有块被命中或者被写入，其就会被压在队尾，于是队头的块必然是最远被使用的块。

3. 在状态`SWAP_IN_OK`中，块换入，用`lru_queue[set_addr][0]`所代表的的line来参与写入cache。
4. 在状态`SWAP_IN_OK`中，根据LRU策略，当一个块换入后，其应该是优先级最高的块，故应该在其队列的队尾。如下实现：

```
for(integer j =1;j<=WAY_CNT;j++)  begin
   lru_queue[set_addr][(j-1)%WAY_CNT]<=lru_queue[set_addr][j%WAY_CNT];
end
```

注意这里lru_queue[set_addr][0]是最近被写入的块，其优先级最高，应该在队尾。按照非阻塞赋值的特点，上面的代码可以实现一个循环队列，每次可以将队头压到队尾，并且资源开销非常小。

把cache加入到CPU

将文件Wb_Data.v里面的DataCache模块替换为cache。为了安全，不要在Wb_Data.v内指定cache参数，因为并不清楚到底是哪里决定了参数的值。

将cache的miss信号接出去，在RV32ICore中，将miss传递到hazard中。由于是miss，所以在hazard.v中，将所有阶段都bubble。

另外在CPU的数据通路的MEM阶段实现了自己的访存和miss的统计模块。（公用的版本好像有问题）

```
reg miss_old;
reg cache_read_old,cache_write_old;

always@(posedge clk or posedge rst)
    begin
        if (rst)
            cache_access_count<=0;
        else if(cache_read&&!cache_read_old || cache_write&&!cache_write_old)
begin
            cache_access_count<=cache_access_count+1;
        end
    end

always@(posedge clk or posedge rst)
    begin
```

```
        if (rst)
            cache_read_old<=0;
        else begin
            cache_read_old<=cache_read;
        end
    end

always@(posedge clk or posedge rst)
    begin
        if (rst)
            cache_write_old<=0;
        else begin
            cache_write_old<=cache_write;
        end
    end


always@(posedge clk or posedge rst)
    begin
        if (rst)
            miss_count<=0;
        else if(miss&&!miss_old) begin
            miss_count<=miss_count+1;
        end
    end

always@(posedge clk or posedge rst)
    begin
        if (rst)
            miss_old<=0;
        else begin
            miss_old<=miss;
        end
    end
```

最终统计出miss多少次和一共访存了多少次。

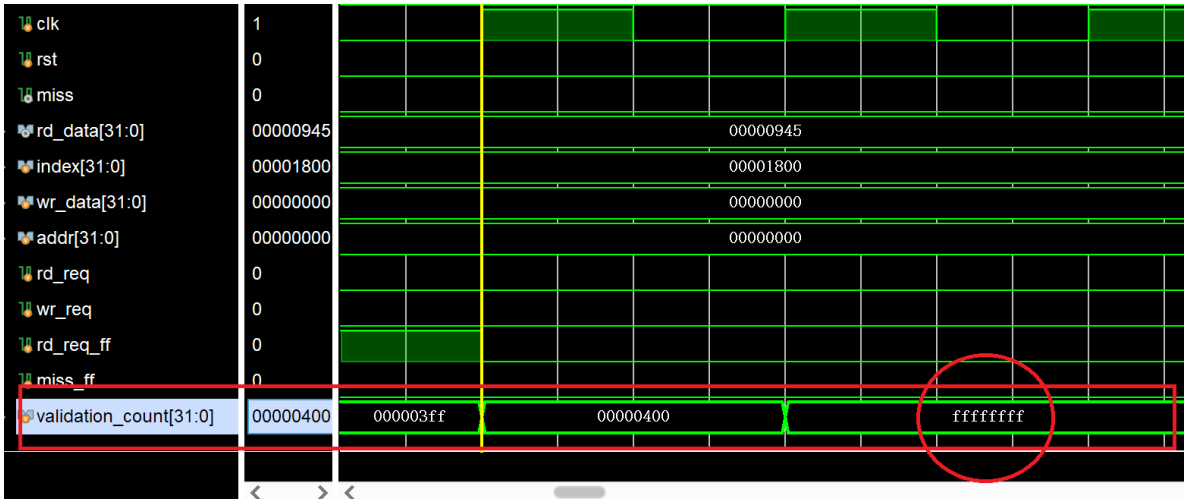
对于这个模块，我通过演算16*16的矩阵乘法一共需要进行多少次访存，和自己统计出来的访存次数进行了比较，结果是正确的。由于miss的统计和访存次数的统计是完全相似的，因此我的miss统计也肯定是正确的。

实验结果

正确性

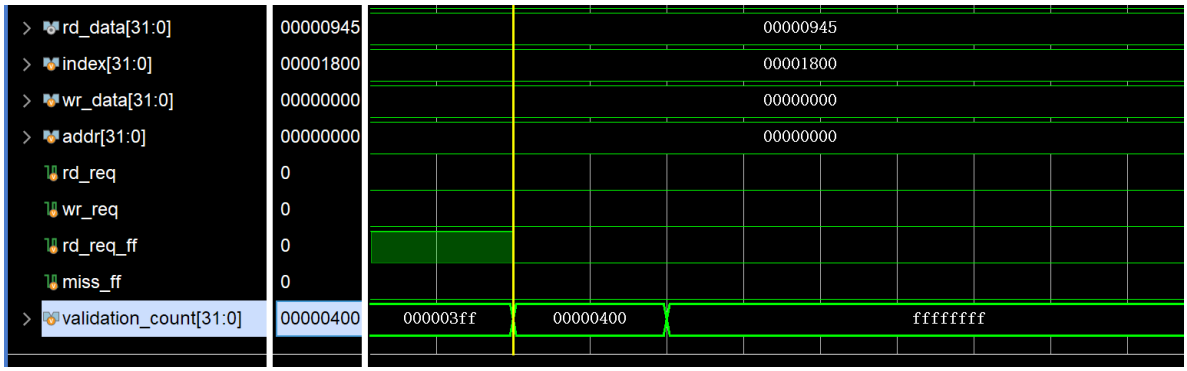
助教当时指定的规模是1024=`0x400`，运行后，结果为：

LRU



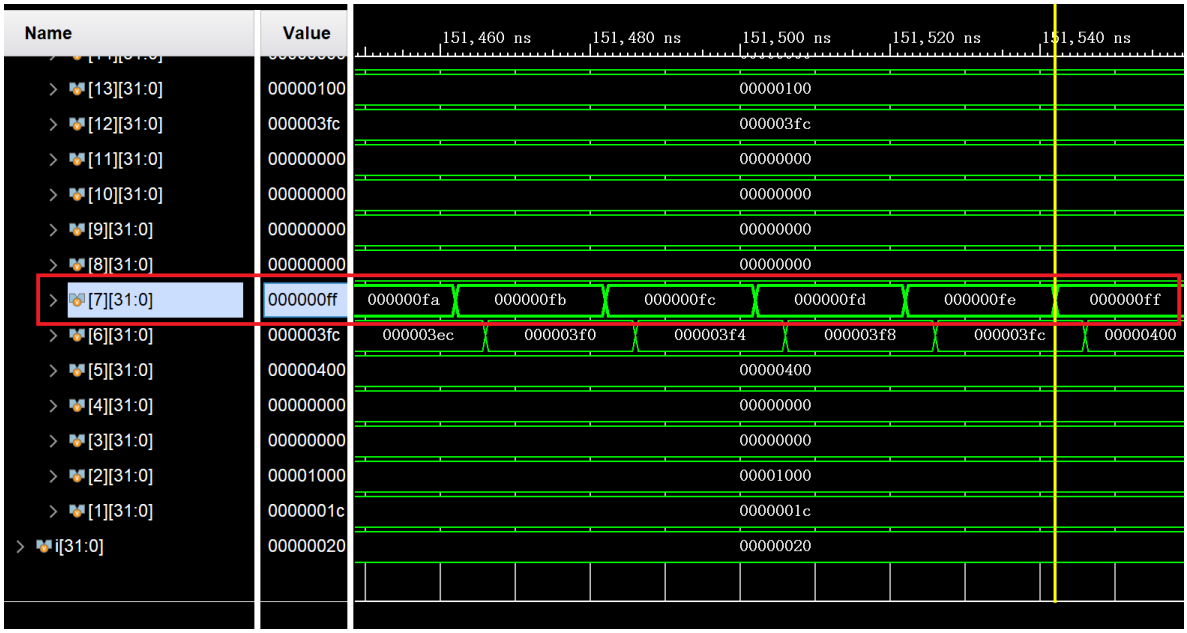
结果正确。

FIFO



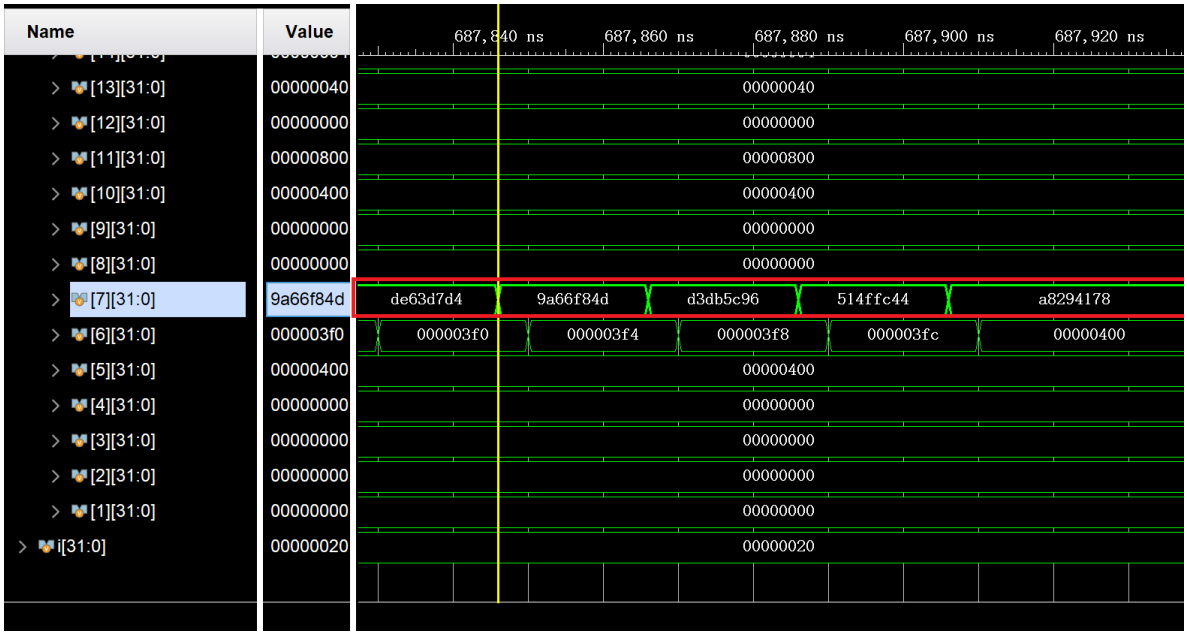
和上面现象相同。

QuickSort.S



正确。

MatMul.S



其中mem.sv：

```
    ...
    ...
    ram_cell[     248] = 32'h0;  // 32'h67214b9a;
    ram_cell[     249] = 32'h0;  // 32'hbb5e7a7b;
    ram_cell[     250] = 32'h0;  // 32'hea44ae80;
    ram_cell[     251] = 32'h0;  // 32'hde63d7d4;
    ram_cell[     252] = 32'h0;  // 32'h9a66f84d;
    ram_cell[     253] = 32'h0;  // 32'hd3db5c96;
    ram_cell[     254] = 32'h0;  // 32'h514ffc44;
    ram_cell[     255] = 32'h0;  // 32'ha8294178;
```

结果正确。

性能分析

下面进入对cache参数、策略等优劣的分析。注：我认为需要保留7号寄存器读数据的过程，以模拟正常计算过程中读取计算结果的过程。所以在以下分析中，我均保留了7号寄存器读取计算结果的过程。所以我的miss率和执行时间可能会比其它人表现的高一些。但是我认为这是应该的。尽管助教在群里说把这部分拿去。

在比较缓存大小、组相联度对缓存性能的影响时，我主要采用矩阵乘法作为待测试程序。因为矩阵乘法的读写次数是可以计算的，其指令执行的数据流与数据本身无关。因此对其的探究和比较更科学。

参数对缓存、主存大小影响的分析

主存大小=2^(LINE_ADDR_LEN+SET_ADDR_LEN+TAG_ADDR_LEN)。保证其不变。假设设置LINE_ADDR_LEN = 3，SET_ADDR_LEN = 3，TAG_ADDR_LEN = 6，为使得主存大小保持$2^{12} = 4096$个words，需要保证TAG_ADDR_LEN + LINE_ADDR_LEN + SET_ADDR_LEN =12；

cache大小=`WAY_CNT*2^(SET_ADDR_LEN+LINE_ADDR_LEN)`words。

那么如果需要改变cache大小，而且不改变组相连度的时候，应该增加`LINE_ADDR_LEN`，这会使得cache大小加倍；同时需要相应地减小`TAG_ADDR_LEN`。

那么如果需要改变组相连度，并保持cache大小不变的时候，若增加`WAY_CNT`，应当相应地减小`LINE_ADDR_LEN`，从而相应地增加`TAG_ADDR_LEN`；为了cache大小不变，`WAY_CNT`应当是2的幂，取`WAY_CNT=1,2,4,8,16,...`

选择初始的`WAY_CNT=4`。

快排时的性能对比

为了控制变量，只随机生成一次待排序的数据。以防止出现因为待排序数据不同而导致的额外的循环。

两种策略的比较

第一次比较

参数为：

参数名称	大小
`LINE_ADDR_LEN`	3
`SET_ADDR_LEN`	3
`TAG_ADDR_LEN`	6
`WAY_CNT`	4

cache大小为$4 \times 2^{3+3} = 2^8 = 256$words。（恰好是快排的规模）

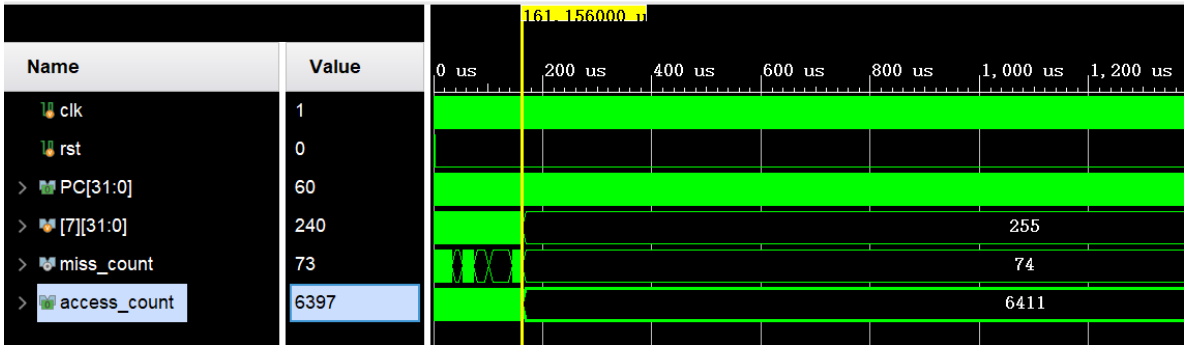
FIFO

cache总次数为6441次，$MissRate = \frac{164}{6441} = 5.55\%$。运行时间为197us。

资源占用：

Resource	Utilization	Available	Utilization %
LUT	4535	63400	7.15
FF	9437	126800	7.44
BRAM	4	135	2.96
IO	81	210	38.57

LRU



cache总次数为6441次，$MissRate = \frac{74}{6441} = 1.15\%$。运行时间为161us。

Resource	Utilization	Available	Utilization %
LUT	4343	63400	6.85
FF	9472	126800	7.47
BRAM	4	135	2.96
IO	81	210	38.57

资源仅多了10%，命中率就有了四倍的优化。

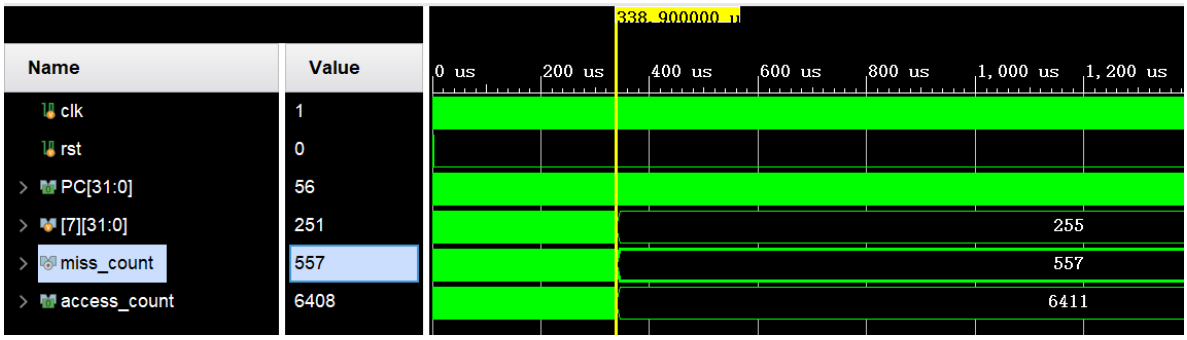
下面减小cache的大小。

第二次比较

参数名称	大小
`LINE_ADDR_LEN`	2
`SET_ADDR_LEN`	3
`TAG_ADDR_LEN`	7
`WAY_CNT`	4

cache大小为$4 \times 2^{2+3} = 2^8 = 128$words。（cache大小变为一半，应当会使得miss rate上升）

FIFO

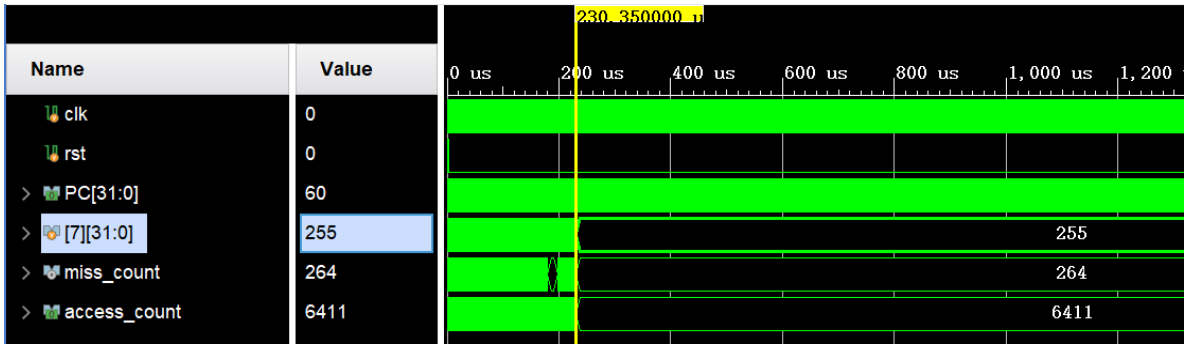


cache总次数为6411次，$MissRate = \frac{557}{6411} = 8.69\%$。运行时间为338us。

Resource	Utilization	Available	Utilization %
LUT	2216	63400	3.50
FF	4986	126800	3.93
BRAM	4	135	2.96
IO	81	210	38.57

资源占用降为一半。

LRU



cache总次数为6411，$MissRate = \frac{264}{6411} = 4.12\%$。运行时间230us。缺失率仅为FIFO的一半。

Resource	Utilization	Available	Utilization %
LUT	2516	63400	3.97
FF	5014	126800	3.95
BRAM	4	135	2.96
IO	81	210	38.57

这里资源和FIFO的资源占用相似。在资源近似的情况下，缺失率有很大下降。

比较结果

在我的实现中，FIFO的缺失率在上面的俩种情况下均要高于LRU，而后者在我的优化下资源和前者的资源占用相近。因此在这种情况下应该优先选择LRU。

矩阵乘法的分析

两种策略的比较

由于矩阵乘法涉及到的数据量大，每次需要循环读的数据多，故增加缓存。将cache size从256增加。

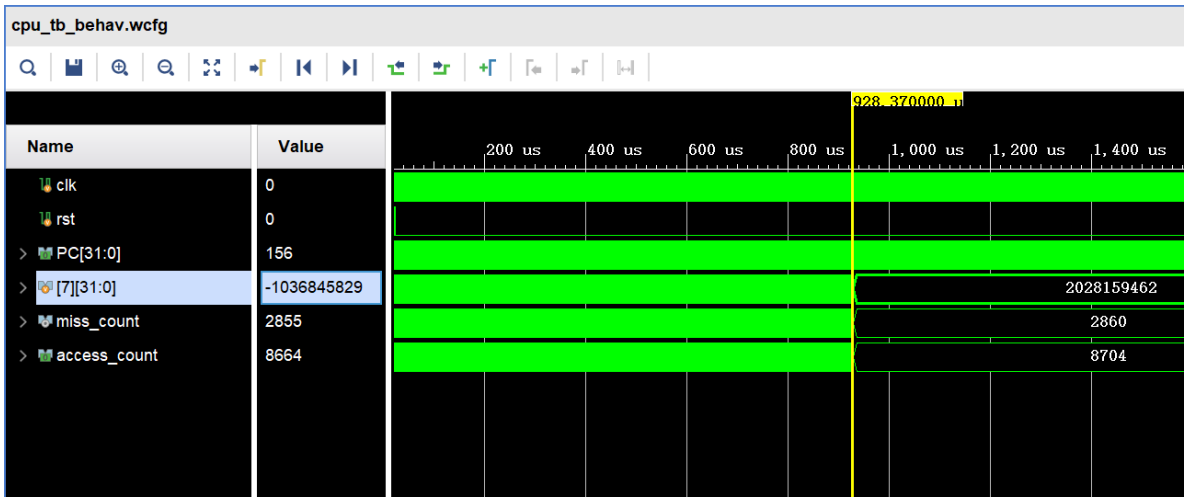
第一次比较

参数为：

参数名称	大小
`LINE_ADDR_LEN`	3
`SET_ADDR_LEN`	3
`TAG_ADDR_LEN`	6
`WAY_CNT`	4

cache大小为$4 \times 2^{3+3} = 2^8 = 256$words。

FIFO



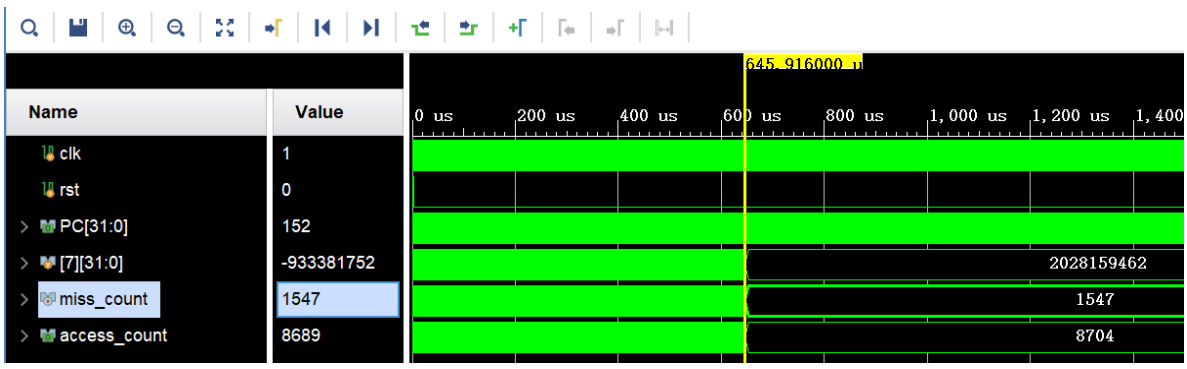
cache总次数为8704次。其中：

$MissRate = \dfrac{2860}{8704} = 32.859\%$。相比于快排，miss率看上去很高。运行时间为0.928ms。

此时的资源占用：

Resource	Utilization	Available	Utilization %
LUT	4535	63400	7.15
FF	9437	126800	7.44
BRAM	4	135	2.96
IO	81	210	38.57

LRU



cache总次数为8704次，$MissRate = \frac{1547}{8704} = 17.77\%$。它的miss率远远比FIFO策略小，我认为这是正确的实验现象。因为在此时出现了大量的连续读写，并且在矩阵运算中常常需要读最近刚刚读写在缓存中的块。这也侧面说明了我的LRU策略是正确的。

Resource	Utilization	Available	Utilization %
LUT	4343	63400	6.85
FF	9472	126800	7.47
BRAM	4	135	2.96
IO	81	210	38.57

可以看到由于我使用的LRU和FIFO的基础数据结构相似，这里的资源和FIFO相比没有增加很多。

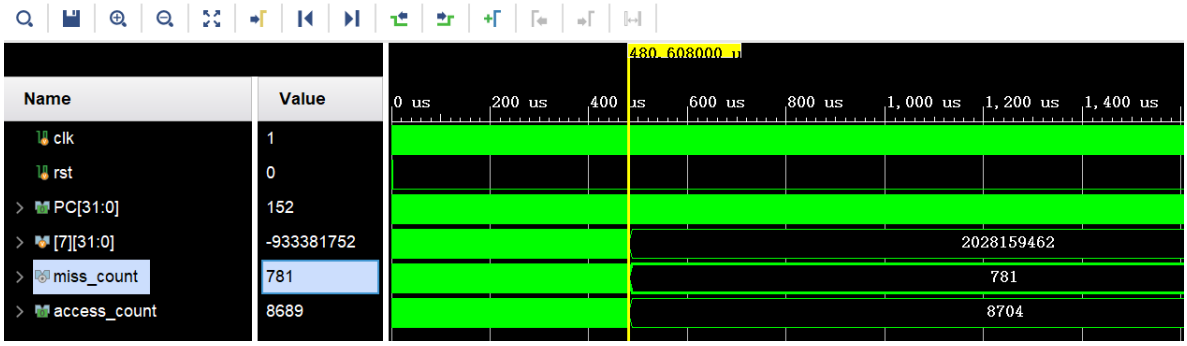
看来这次的实验的分析和个人的实现关系比较大。

第二次比较

参数名称	大小
`LINE_ADDR_LEN`	4
`SET_ADDR_LEN`	3
`TAG_ADDR_LEN`	5
`WAY_CNT`	4

cache的大小为$4 \times 2^{3+4} = 512$words。

FIFO

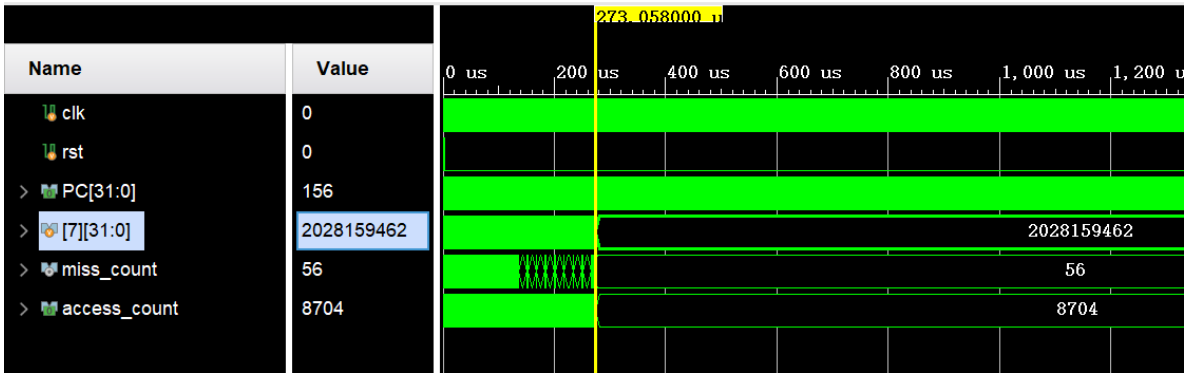


cache总次数为8704次，$MissRate = \frac{781}{8704} = 8.97\%$。运行时间为480us。

Resource	Utilization	Available	Utilization %
LUT	8012	63400	12.64
FF	18374	126800	14.49
BRAM	4	135	2.96
IO	81	210	38.57

比起256word资源翻倍。

LRU



cache总次数为8704次，$MissRate = \frac{56}{8704} = 0.64\%$。运行时间为273us。命中率和运行时间都有显著下降。

Resource	Utilization	Available	Utilization %
LUT	9058	63400	14.29
FF	18401	126800	14.51
BRAM	4	135	2.96
IO	81	210	38.57

比起256word资源翻倍。也比FIFO的资源多出约10%。

总结

替换策略	cache size(words)	WAY CNT	miss率(%)	周期数	LUT	FF
FIFO	256	4	20.2178	168968	4100	9409
LRU	256	4	22.8693	169379	21500	13603
FIFO	512	4	0.7576	67414	8524	18366
LRU	512	4	0.9706	68335	28358	22498

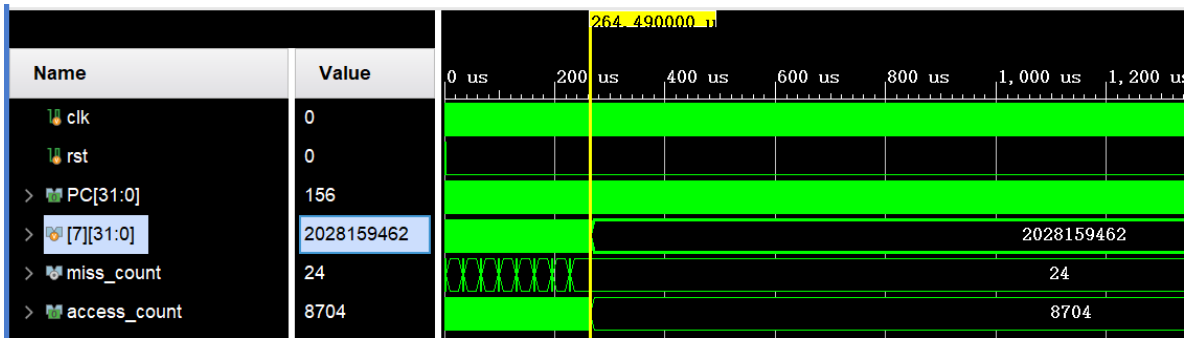
相比之下，FIFO策略全面优于LRU，miss率比LRU低，资源比LRU少。

cache大小对缓存性能的影响

选择LRU来探究cache大小对命中率和资源的影响。从cache大小为256word开始尝试增加cache size。用矩阵乘法来模拟程序。

参数名称	大小
LINE_ADDR_LEN	5
SET_ADDR_LEN	3
TAG_ADDR_LEN	4
WAY_CNT	4

cache size为$4 \times 2^{5+3} = 1024$words。



cache总次数为8704次，$MissRate = \frac{24}{8704} = 0.275\%$。比512word的缓存版本miss率下降1倍。

Resource	Utilization	Available	Utilization %
LUT	16583	63400	26.16
FF	36289	126800	28.62
BRAM	4	135	2.96
IO	81	210	38.57

相比与512word的缓存，这里的占用资源率增加了近1倍。说明缓存大小和资源占用率近似正比。

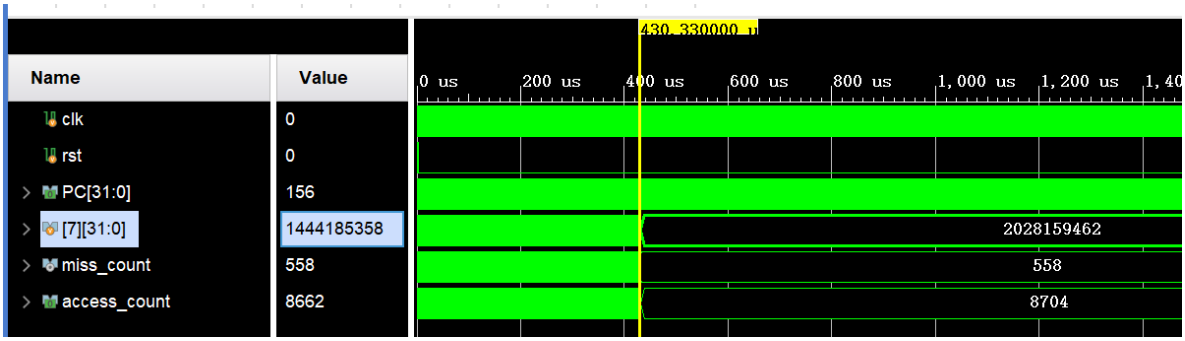
组相连度的影响

取cache size = 512，取LRU策略，取16*16矩阵乘法作为模拟程序。

第1次

参数名称	大小
LINE_ADDR_LEN	5
SET_ADDR_LEN	3
TAG_ADDR_LEN	4
WAY_CNT	2

cache size为$2 \times 2^{3+5} = 512$words。



cache总次数为8704次，$MissRate = \frac{558}{8704} = 7.53\%$。和512word，组相连度为4的缓存相比，miss率增加了约2%。

Resource	Utilization	Available	Utilization %
LUT	8385	63400	13.23
FF	19780	126800	15.60
BRAM	4	135	2.96
IO	81	210	38.57

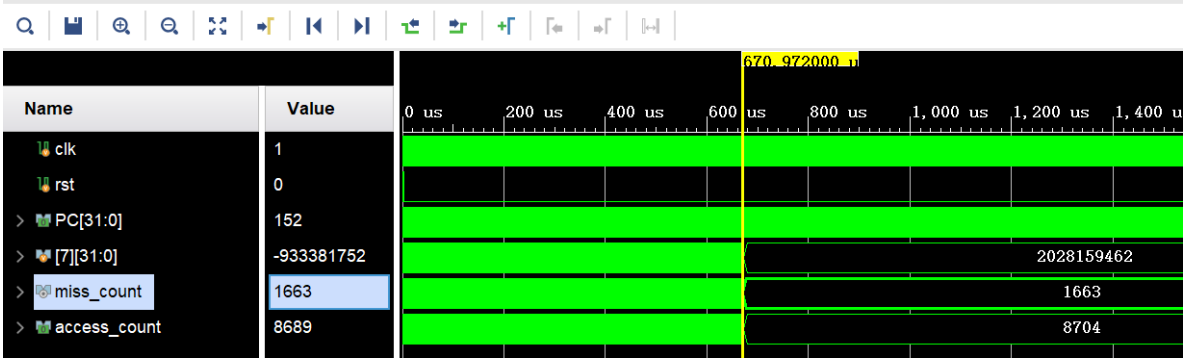
和组相连度为4的LRU策略缓存相比，这里LUT资源减少了约10%，FF资源略有增加。占用资源率相似。

第2次

尝试增加WAY CNT。

参数名称	大小
LINE_ADDR_LEN	3
SET_ADDR_LEN	3
TAG_ADDR_LEN	6
WAY_CNT	8

此时cache size为$8 \times 2^{3+3} = 512$words。



cache总次数为8704次，$MissRate = \frac{1663}{8704} = 19.11\%$。可以看到这里缺失率显著增加。因此组相连度不是越大越好。

Resource	Utilization	Available	Utilization %
LUT	4205	63400	6.63
FF	9533	126800	7.52
BRAM	4	135	2.96
IO	81	210	38.57

资源有所减少。

实验总结

1. cache大小越大，miss次数越少，周期数越少，性能越好；但是所用的资源也越多。应该按照自己程序的需求来设计cache的大小，如对于本次实验的矩阵乘法，设计512word的缓存已经足够了。
2. 在我的实现中，LRU的缺失率要远远小于FIFO的缺失率，在大部分情况下，FIFO的缺失率都比LRU的缺失率高出至少一倍。而在我的优化下，LRU策略可以以高出10%的资源为代价实现更好的缓存替换策略。
3. 如果在固定cache大小的情况下，随着组相连度的增加，性能先上升后下降，存在一个适度的组相连度使得性能最好。
4. 写回法有缺点，就是在程序结束了后，可能会有数据没有写回主存，依旧留在cache中。在实际生活中，如果计算机的故障比较多，写回法也许会造成数据丢失。这对于分布式系统而言，很有可能造成数据不一致的情况。因此要谨慎使用。