Lab3 实验报告

【实验目标】

- 将提供的cache.sv代码由直接映射修改为可调整组相连度的N路相连Cache，并通过相关的读写测试
- 将修改过的Cache替换CPU中的datamem，并且执行快排和矩阵乘法的代码，观察主存中的数据情况是否符合预期，从而判断Cache编写是否正确
- 调整Cache的几个参数，对同一个程序统计缺失率和运行周期，分析Cache性能的影响因素
- 调整Cache的几个参数，对同一程序进行综合，分析Cache资源占用

【实验环境和工具】

- 实验环境：Windows 10操作系统
- 实验工具：Vivado 2019.1
- 实验方式：Vivado自带的仿真波形

【实验内容和过程】

一、代码及结果分析

1、Cache

```
reg [            31:0] cache_mem    [SET_SIZE][WAY_CNT][LINE_SIZE];
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE][WAY_CNT];
reg                    valid        [SET_SIZE][WAY_CNT];
reg                    dirty        [SET_SIZE][WAY_CNT];
```

原本的几个关于每个set的有效位、脏块位和标志位以及cache的存储都加上一维，用于区分每个set中的way。

```
always @ (*) begin
    for(integer i = 0; i < WAY_CNT; i++) begin
        if(valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) begin
            cache_hit[i] = 1'b1;
            way_hit_addr = i;
        end
        else
            cache_hit[i] = 1'b0;
    end
end
```

修改命中判断如上，N路组相联要对每个set中的各路判断是否命中，并记录命中的路的位置，用于后续使用。

```
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        cache_stat <= IDLE;
        tag <= 0;
        for(integer i = 0; i < SET_SIZE; i++) begin
            for(integer j = 0; j < WAY_CNT; j++) begin
```

```
                dirty[i][j] = 1'b0;
                valid[i][j] = 1'b0;
            end
        end
        for(integer k = 0; k < LINE_SIZE; k++)
            mem_wr_line[k] <= 0;
        mem_wr_addr <= 0;
        {mem_rd_tag_addr, mem_rd_set_addr} <= 0;
        rd_data <= 0;
    end else begin
        case(cache_stat)
        IDLE:       begin
                        if(|cache_hit) begin
                            if(rd_req) begin
                                rd_data <= cache_mem[set_addr][way_hit_addr]
[line_addr];
                            end else if(wr_req) begin
                                cache_mem[set_addr][way_hit_addr][line_addr] <=
wr_data;
                                dirty[set_addr][way_hit_addr] <= 1'b1;

                            end
                        end else begin
                            if(wr_req | rd_req) begin
                                if(valid[set_addr][choose_way] & dirty[set_addr]
[choose_way]) begin
                                    cache_stat  <= SWAP_OUT;
                                    mem_wr_addr <= {cache_tags[set_addr]
[choose_way], set_addr};
                                    mem_wr_line <= cache_mem[set_addr]
[choose_way];
                                end else begin
                                    cache_stat  <= SWAP_IN;
                                end
                                {mem_rd_tag_addr, mem_rd_set_addr} <= {tag_addr,
set_addr};
                            end
                        end
                        if(tag == 2'b10)
                            tag <= 2'b01;
                        else if(tag == 2'b01)
                            tag <= 2'b00;
                        else
                            tag <= 2'b00;
                    end
        SWAP_OUT:   begin
                        if(mem_gnt) begin
                            cache_stat <= SWAP_IN;
                        end
                    end
        SWAP_IN:    begin
                        if(mem_gnt) begin
                            cache_stat <= SWAP_IN_OK;
                        end
                    end
        SWAP_IN_OK: begin
                        for(integer i=0; i<LINE_SIZE; i++)
 cache_mem[mem_rd_set_addr][choose_way][i] <= mem_rd_line[i];
```

```
                        cache_tags[mem_rd_set_addr][choose_way] <=
mem_rd_tag_addr;
                        valid     [mem_rd_set_addr][choose_way] <= 1'b1;
                        dirty     [mem_rd_set_addr][choose_way] <= 1'b0;
                        cache_stat <= IDLE;
                        tag <= 2'b10;
                    end
        endcase
    end
end
```

整个cache的状态机修改如上，对比直接相连，仅在赋值时加上了way_hit_addr保证了所取的值对应的是正确的那一路；在换入中，用到了换入换出策略所选择的choose_way用于换出保证了换入换出策略的正常使用。

```
always@(*) begin
    choose_way = 32'b0;
    for(integer i = 0;i < WAY_CNT;i++) begin
        if(way_age[set_addr][choose_way] < way_age[set_addr][i])begin
            choose_way = i;
        end
    end
end
```

首先choose_way的选择都是根据各路的age值，选择其最大的为换出位。对于FIFO来说，age值越大表示换入的越早；而对于LRU来说，age值越大表示越长时间未被使用，从而完成了换出路的选择。

```
always@(posedge clk or posedge rst) begin
    if(rst) begin
        for(integer i = 0; i < SET_SIZE; i++) begin
            for(integer j = 0; j < WAY_CNT; j++) begin
                way_age[i][j] <= 32'b0;
            end
        end
    end
    else if(!miss) begin
        for(integer i = 0; i < WAY_CNT; i++) begin
            if((i == choose_way)&&(tag == 1)) begin
                way_age[set_addr][i] <= 32'b0;
            end
            else begin
                way_age[set_addr][i] <= way_age[set_addr][i] + 1;
            end
        end
    end
end
```

上为FIFO的源码，复位时，所有的age均为0，当发生了换入换出时，出现换入换出的那一路的age为0，而未发生换入换出的路的age同步增长，从而实现了先换入的age大于后换入的age。这里的tag用于确保上个状态为SWAP_IN_OUT即为换入换出。

```
always@(posedge clk or posedge rst) begin
    if(rst) begin
        for(integer i = 0; i < SET_SIZE; i++) begin
            for(integer j = 0; j < WAY_CNT; j++) begin
```

```
                way_age[i][j] <= 32'b0;
            end
        end
    end
    else if(!miss) begin
        for(integer i = 0; i < WAY_CNT; i++) begin
            if((wr_req | rd_req)&& i == way_hit_addr) begin
                way_age[set_addr][i] <= 32'b0;
            end
            else begin
                way_age[set_addr][i] <= way_age[set_addr][i] + 1;
            end
        end
    end
end
```

上为LRU的代码，和FIFO的区别在于age置位0的条件，即有读或者写信号时，对应的路的age置位0，保证了一旦使用该路，其age置为0，从而实现了未使用的路的age一直在增长，从而最久未被使用的age值最大，用于换出。

2、CPU代码修改

```
    cache #(
    .LINE_ADDR_LEN  ( 4             ),
    .SET_ADDR_LEN   ( 2             ),
    .TAG_ADDR_LEN   ( 4            ),
    .WAY_CNT        ( 4             )
) cache_test_instance (
    .clk            ( clk           ),
    .rst            ( clear         ),
    .miss           ( Dcachemiss    ),
    .addr           ( A       ),
    .rd_req         ( MemToRegM     ),
    .rd_data        ( RD_raw        ),
    .wr_req         ( |WE           ),
    .wr_data        ( WD            )
);
```

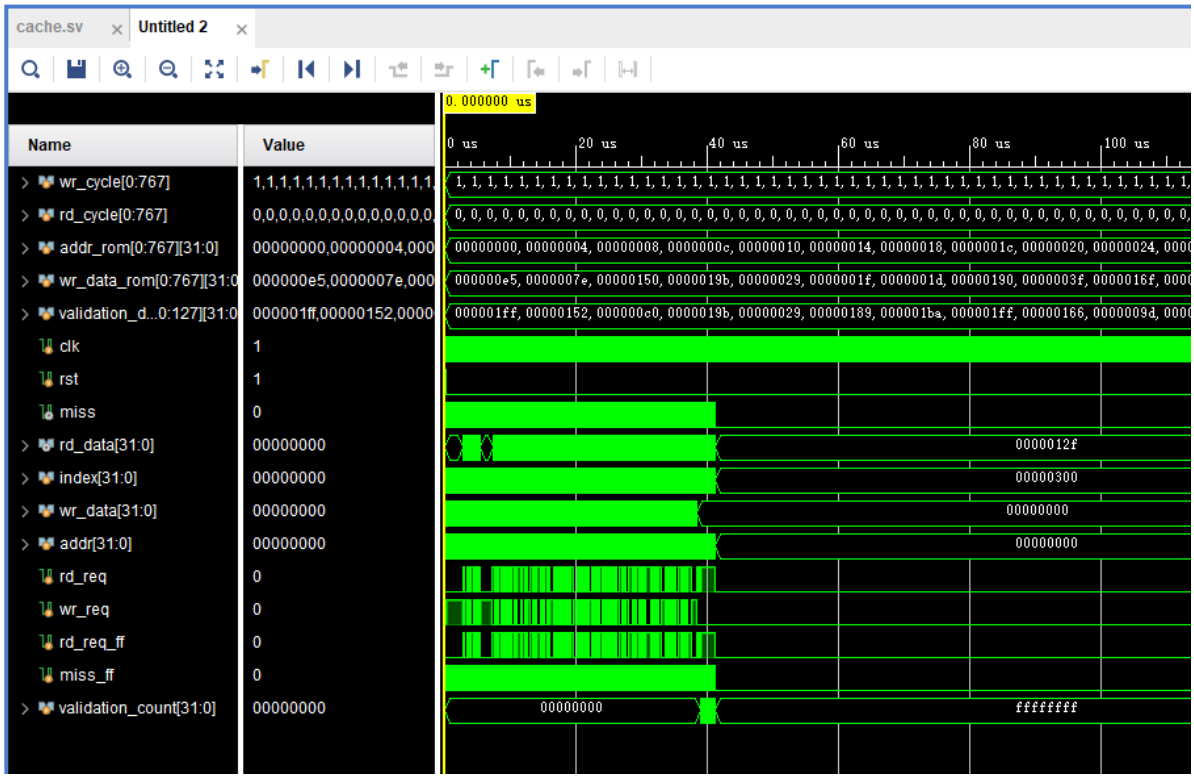
WB段寄存器中的dataram替换为cache，需要注意的是cache中的地址为字地址，所以需要直接连A，读使能仅在LW指令是会生效，因而可以直接使用MemToReg信号作为其输入，由于本次实验不用考虑非对齐写，所以要对WE信号进行按位或，不需要考虑非对齐写。

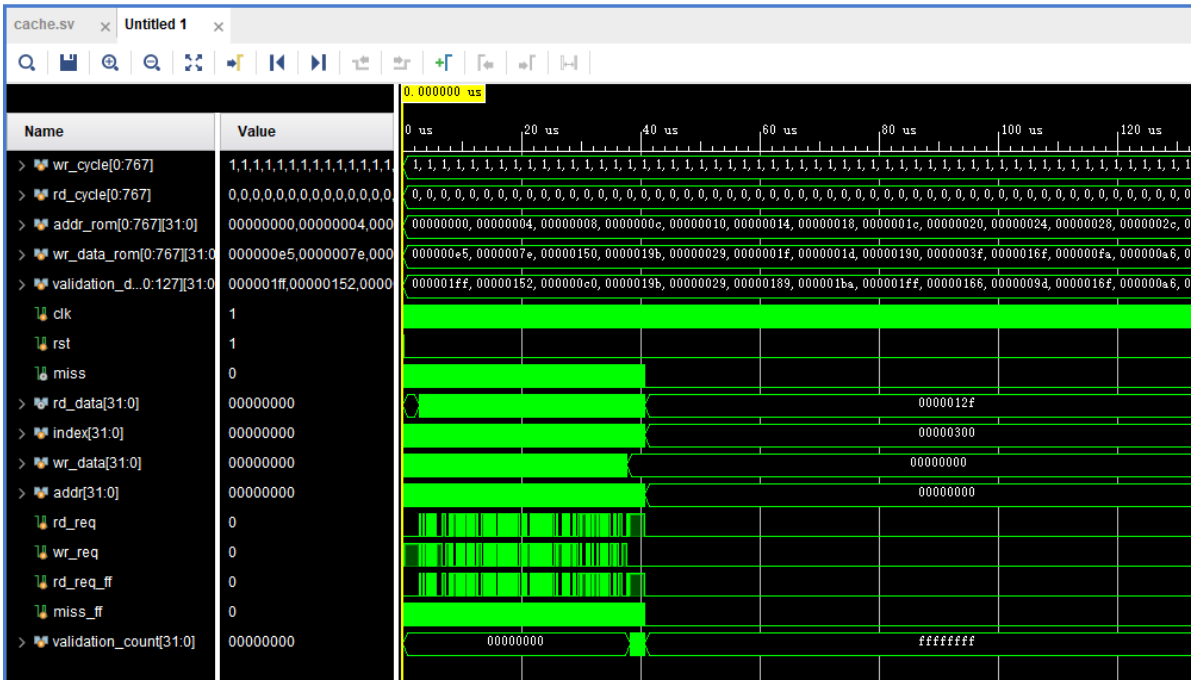
```
 else if(DCacheMiss)
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b1010101010;
```

CPU代码其次修改了地方为冒险模块，当cache未命中时，需要额外的周期从主存中换入换出相关的块，故当CacheMiss时，要停顿整个流水线，故所有的stall信号均生效。

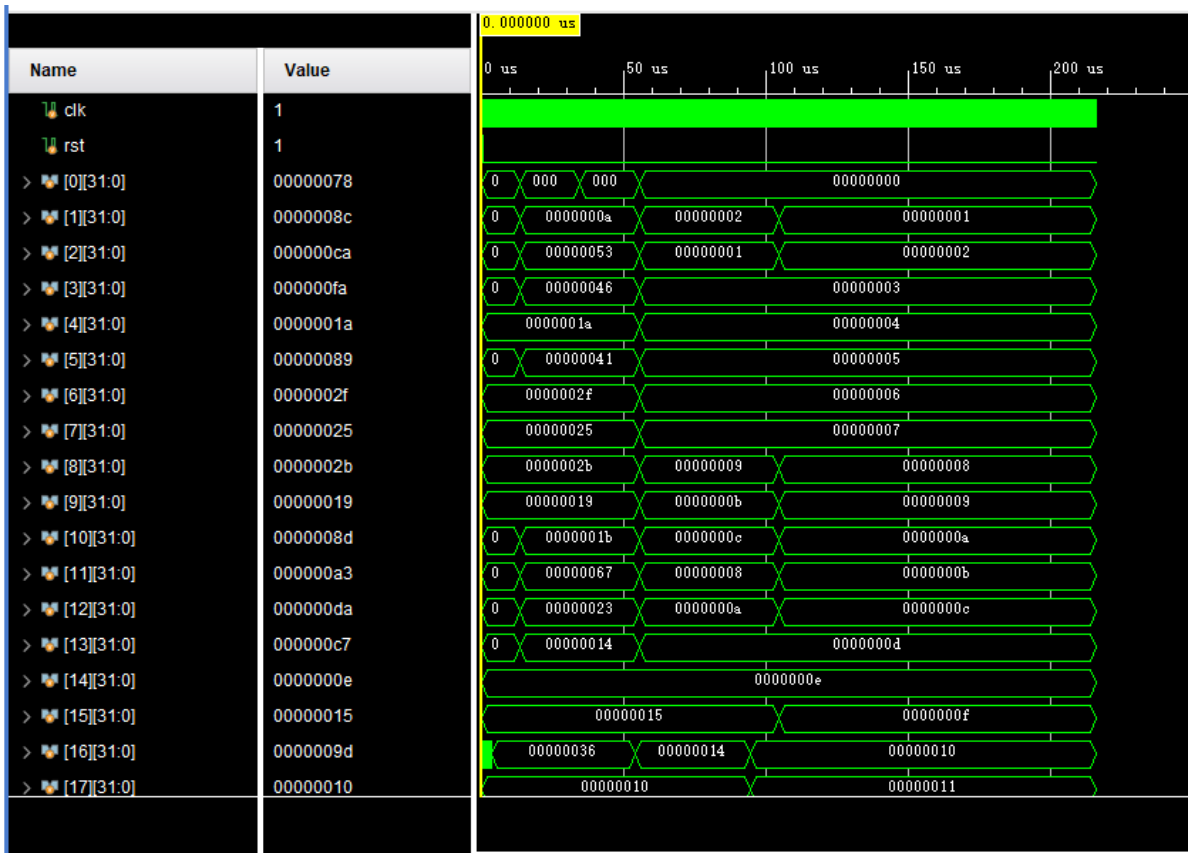
剩余修改了ID段寄存器的RD_old赋值，原本的值存在问题，会导致ID段寄存器未按照预期的情况stall导致整个cpu出现问题。

3、仿真结果分析

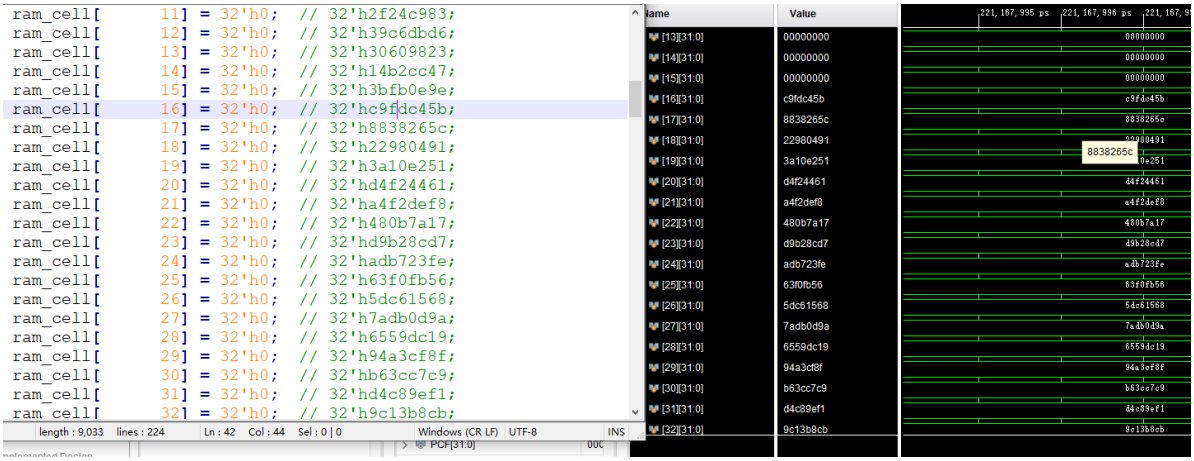




上图分别是FIFO和LRU策略下cache的仿真，根据给出的仿真文件说明，validation_count为0xffffffff，说明cache功能符合预期，完成了N路组相联Cache的功能实现。



快排的部分结果如下，可以观察到主存中的数据逐渐有序，最终主存绝大部分有序，只有一部分的值在cache中尚未换出所以无序。



上为矩阵乘法的最终结果，由于前面的数据未写入，故从ram_cell[16]开始观察， 得到结果符合预期，且在cache中也找到了未写入的数据，符合前面15个数的值，因而实现成功。

二、Cache性能评估

1、快速排序

（1）组数固定，改变组相连度

设置SET_ADDR_LEN = 2，LINE_ADDR_LEN = 3,分别调整相连度，得到结果如下

运行时间/ns	1路组相联	2路组相联	4路组相联	8路组相联
FIFO(256)	342904	243144	207072	185216
LRU(256)	342904	208092	180936	162068
FIFO(512)	821108	631936	557416	500864
LRU(512)	821108	539832	482492	427016

缺失率	1路组相联	2路组相联	4路组相联	8路组相联
FIFO(256)	10.06%	5.32%	3.65%	2.64%
LRU(256)	10.06%	3.61%	2.34%	1.53%
FIFO(512)	10.96%	7.01%	5.47%	4.23%
LRU(512)	10.96%	4.93%	3.74%	2.53%

（2）cache大小固定，改变组数和组相连度

设置LINE_ADDR_LEN = 3，且保证块大小不变

运行时间/ns	1路组相联，8组	2路组相联，4组	4路组相联，2组
FIFO(256)	270692	243144	241056
LRU(256)	270692	208092	210208
FIFO(512)	652400	631936	620932
LRU(512)	652400	539832	535800

缺失率	1路组相联，8组	2路组相联，4组	4路组相联，2组
FIFO(256)	6.49%	5.32%	5.32%
LRU(256)	6.49%	3.61%	3.70%
FIFO(512)	7.30%	7.01%	6.82%
LRU(512)	7.30%	4.93%	4.85%

对比两个四个表格，可以得到以下几个结论：

- 仅改变相连度时，相当于扩大了cache 的大小，可以看到运行时间明显下降且缺失率也随着组相联度增大而减小，说明此时cache的性能更优，但是，更大的cache意味着更复杂的工艺以及更高的成本，所以并不能无限的增长；
- 当cache大小固定的时候，直接映射的性能最差，运行时间和缺失率均高于后面两种，但是后两种的运行时间和缺失率相差不大，说明在某个范围内是更优的；
- 对比FIFO和LRU策略，可以很明显的看出来，除了直接相连外，其余情况下均为LRU优于FIFO，显然符合预期，LRU策略更符合局部性原理，能够更大程度上减少缺失率。

2、矩阵乘法

（1）组数固定，改变组相连度

设置SET_ADDR_LEN = 3，LINE_ADDR_LEN = 3,分别调整相连度，得到结果如下

运行时间/ns	1路组相联	2路组相联	4路组相联
FIFO(16*16)	1418408	1267864	933776
LRU(16*16)	1418408	1321640	646432

缺失率	1路组相联	2路组相联	4路组相联
FIFO(16*16)	58.82%	51.13%	33.40%
LRU(16*16)	58.82%	53.68%	17.77%

（2）cache大小固定，改变组数和组相连度

运行时间/ns	1路组相联，8组	2路组相联，4组	4路组相联，2组
FIFO(16*16)	1418408	1391208	1274152
LRU(16*16)	1418408	1321640	1321640

缺失率	1路组相联，8组	2路组相联，4组	4路组相联，2组
FIFO(16*16)	58.82%	58.00%	52.39%
LRU(16*16)	58.82%	53.68%	53.68%

从上述数据可以看出：

- 随着连通度的增大，cache增大，导致运行时间减少，缺失率下降，说明增大连通度对cache的性能有提高的作用，但受工艺和成本的因素不能无限增长；
- 当cache大小固定时，组相连度增大在一定程度上提高了cache的性能，但是幅度不是非常大；
- 观察到，除了4路组相联，8组时，FIFO性能均比LRU好，主要原因是其他组的cache大小相较于整个mem的规模偏小，导致了LRU无法正常发挥出作用。

三、Cache资源评估

一、组数固定，改变组相连度

SET_ADDR_LEN = 3, SET_ADDR_LEN = 2 , TAG_ADDR_LEN = 3

资源消耗	1路组相联	2路组相联	4路组相联
FIFO	LUT:1.71% FF:1.56%	LUT:3.38% FF:2.58%	LUT:7.47% FF:4.44%
LRU	LUT:1.71% FF:1.56%	LUT:3.38% FF:2.58%	LUT:7.58% FF:4.44%

二、cache大小固定，改变组相连度和组数

WAY_CNT = 1时，SET_ADDR_LEN = 3,SET_ADDR_LEN = 3, TAG_ADDR_LEN = 3

资源消耗	1路组相联	2路组相联	4路组相联
FIFO	LUT:1.82% FF:2.38%	LUT:3.38% FF:2.58%	LUT:4.27% FF:2.58%
LRU	LUT:1.82% FF:2.38%	LUT:3.38% FF:2.59%	LUT:4.25% FF:2.60%

三、改变line大小

SET_ADDR_LEN = 3时, SET_ADDR_LEN = 2 , TAG_ADDR_LEN = 3，WAY_CNT= 4

资源消耗	2	3	4
FIFO	LUT:3.99% FF:2.53%	LUT:7.47% FF:4.44%	LUT:14.17% FF:8.26%
LRU	LUT:4.07% FF:2.53%	LUT:7.58% FF:4.44%	LUT:14.20% FF:8.26%

从得到的数据可以看出：

- 当组数固定时，随着相连度增大，cache的大小增大， LUT和FF均增大且近似于线性增长，说明了cache大小需要的电路资源基本上是线性的；
- 当cache大小固定时，FF随相连度组数的变化基本上没有什么改变，而LUT的消耗随着相连度的增大而增大，可以推断出LUT受组数影响，随组数的减小而增大；
- 当改变line的大小时，其相当于增大cache的大小，可以看到LRU和FF的使用情况近似于线性增长，说明资源消耗随着cache的大小增加而增加
- 两种替换策略所需要的资源消耗差距不大，但LRU所需要的资源消耗要更多一些, 在算法实现上两种策略基本是差不多的，却别在于LRU中的age变化更为频繁。

【实验总结】

本次实验主要是对cache进行了比较综合性的实验，完成了N路组相联cache的设计，且将该cache替换了实验2中CPU的数据存储器，从整个实验的结果上来看，该cache很顺利的完成了相应的功能，能够正确的使用两种替换策略进行换入换出。在FIFO和LRU两种替换策略中，很清楚的可以看出LRU在性能上更优于FIFO，原因在于其更加符合局部性原理，替换时更多的换出使用频率上较低的块从而避免了缺失率的增高。

我在本次实验中主要遇到的问题在于将cache接入CPU中时，部分信号与原本的dataram有所差异，因而导致了部分的bug，另外是ID段寄存器中存在的一个固有bug前期未发现导致损耗了大量时间，但最终均得以解决，对cache有了更好的认识，清楚的了解了cache各项参数对其性能的影响。