

lab3 Cache

实验目标

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

实验环境和工具

- Windows10 Professional
- Vivado 2019.2

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

将助教提供的直接映射、写回带写分配的 cache 修改为 N 路组相连的（要求组相连度使用宏定义可调）、写回并带写分配的cache。

- 写分配法(write-allocate)：当CPU对Cache写不命中时，把主存中的块调入Cache，在Cache中修改。通常搭配写回法使用。
- 写回法(write-back)——当CPU对Cache写命中时，只修改Cache的内容，而不立即写入主存，只有当此块被换出时才写回主存
- 增加一个 STRATEGY 参数，为 0 代表使用 LRU 换出策略，为 1 代表使用 FIFO 换出策略

```
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
    parameter STRATEGY       = 1 // 换出策略，0 对应 LRU，1 对应 FIFO
)
```

LRU

- 将 cache 改为 N 路组相连接，需要将 cache_mem、cache_tags、valid、dirty 增加一个维度

```
// 增加一个维度
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)
```

后续使用它们的时候也需要增加对应的维度

- 判断 cache 是否命中，与 set 内的每个 line 的 tag 并行比较

```
always @ (*)
begin
    // 判断 address 是否在 cache 中命中
    cache_hit = 1'b0; // 默认不命中
    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
            cache_hit = 1'b1;
    end
end
```

- 设置一个 time_stamp，用来标识全局的时间

```
integer time_stamp;
```

- 在 rst 时，time_stamp 置零。
- 每个时钟上升沿到来时，time_stamp 加一
 - 这里 time_stamp 可能会溢出，可以增加一个判断语句来避免溢出。当然也可以只在每次访存时才将 time_stamp 加一，从而降低溢出的可能性
- 设置一个 order 数组，记录每个 line 的使用时间。LRU 策略每次都换出当前 set 中时间值最小的 line（最长时间未使用）。同时设置一个 way_addr 用来记录地址：如果 cache hit，则记录对应的 way_addr；如果 cache miss，则记录当前 set 中 order 值最小的 line 的 way_addr

```
// LRU
// 实现换出策略：记录每个line的使用时间，LRU选择同个set中时间最小的line换出
reg [WAY_ADDR_LEN-1:0] way_addr;
reg [31:0] order [SET_SIZE][WAY_CNT];

always @ (*)
begin
    if(cache_hit)
    begin // hit
        // 在组中并行寻找，记录组中命中的addr
        for (integer i = 0; i < WAY_CNT; i++)
        begin
            if(cache_tags[set_addr][i] == tag_addr)
```

```

        begin
            way_addr = i;
            break;
        end
    end
end
else if(rd_req | wr_req)
begin // miss
    way_addr = 0;
    // 在组中并行寻找，记录时间最小的line的addr
    // 初始化为0，所以这里从1开始即可
    for (integer i = 1; i < WAY_CNT; i++)
    begin
        if (order[set_addr][i] < order[set_addr][way_addr])
        begin
            way_addr = i;
        end
    end
end
end
end

```

- 更新 order

- 当 cache 命中时，需要更新对应 line 的 order 值，即更新为当前时间

```

// 每一次cache命中时，更新对应line的order
if(cache_hit) begin
    if(wr_req | rd_req) begin
        // FIFO 和 LRU 的唯一区别
        if(STRATEGY == 0) begin
            order[set_addr][way_addr] <= time_stamp;
        end
    end
end

```

- 当从内存中换入新的 line 时，也需要更新对应 line 的 order 值，即更新为当前时间

```

        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][mem_rd_way_addr]
[i] <= mem_rd_line[i];
            cache_tags[mem_rd_set_addr][mem_rd_way_addr] <=
mem_rd_tag_addr;
            valid [mem_rd_set_addr][mem_rd_way_addr] <=
1'b1;
            dirty [mem_rd_set_addr][mem_rd_way_addr] <=
1'b0;
            // 换入新的line时，也要更新line对应的order
            order[mem_rd_set_addr][mem_rd_way_addr] <=
time_stamp;
            cache_stat <= IDLE; // 回到就绪状态
        end
    end

```

FIFO

- FIFO 和 LRU 大体类似，主要区别在于 LRU 策略需要考虑 cache 是否被访问，记录对应的访问时间，根据记录换出最近最长时间未被访问的 line；而 FIFO 只根据 line 进入 cache 的时间来决定被换出的 line。所以在实现上，如果为 FIFO 策略，在 cache 命中时不更新 order 即可，只在 line 被换入 cache 时更新 order

```
if(cache_hit) begin
    if(wr_req | rd_req) begin
        // FIFO 和 LRU 的唯一区别
        if(STRATEGY == 0) begin
            order[set_addr][way_addr] <= time_stamp;
        end
    end
end
```

结果

利用助教提供的命令，生成对应的 testbench

```
python .\generate_cache_tb.py 16 > cache_tb.sv
```

cache_tb.sv

```
`timescale 1ns/100ps
//correct read result:
// 00000038 00000021 0000003b 0000001c 00000009 00000039 0000003c 00000017
00000029 00000000 0000003a 0000000d 0000002d 0000003c 00000037 00000011

module cache_tb();

`define DATA_COUNT (16)
`define RDWR_COUNT (6*`DATA_COUNT)

reg wr_cycle      [`RDWR_COUNT];
reg rd_cycle      [`RDWR_COUNT];
reg [31:0] addr_rom  [`RDWR_COUNT];
reg [31:0] wr_data_rom [`RDWR_COUNT];
reg [31:0] validation_data [`DATA_COUNT];

initial begin
    // 16 sequence write cycles
    rd_cycle[ 0] = 1'b0; wr_cycle[ 0] = 1'b1; addr_rom[
0]='h00000000; wr_data_rom[ 0]='h0000001b;
    rd_cycle[ 1] = 1'b0; wr_cycle[ 1] = 1'b1; addr_rom[
1]='h00000004; wr_data_rom[ 1]='h0000003d;
    rd_cycle[ 2] = 1'b0; wr_cycle[ 2] = 1'b1; addr_rom[
2]='h00000008; wr_data_rom[ 2]='h00000020;
    rd_cycle[ 3] = 1'b0; wr_cycle[ 3] = 1'b1; addr_rom[
3]='h0000000c; wr_data_rom[ 3]='h00000016;
    rd_cycle[ 4] = 1'b0; wr_cycle[ 4] = 1'b1; addr_rom[
4]='h00000010; wr_data_rom[ 4]='h00000033;
    rd_cycle[ 5] = 1'b0; wr_cycle[ 5] = 1'b1; addr_rom[
5]='h00000014; wr_data_rom[ 5]='h0000001c;
    rd_cycle[ 6] = 1'b0; wr_cycle[ 6] = 1'b1; addr_rom[
6]='h00000018; wr_data_rom[ 6]='h0000002b;
    rd_cycle[ 7] = 1'b0; wr_cycle[ 7] = 1'b1; addr_rom[
7]='h0000001c; wr_data_rom[ 7]='h0000002b;
```

```

rd_cycle[ 8] = 1'b0; wr_cycle[ 8] = 1'b1; addr_rom[
8]='h00000020; wr_data_rom[ 8]='h0000003b;
rd_cycle[ 9] = 1'b0; wr_cycle[ 9] = 1'b1; addr_rom[
9]='h00000024; wr_data_rom[ 9]='h00000000;
rd_cycle[10] = 1'b0; wr_cycle[10] = 1'b1; addr_rom[
10]='h00000028; wr_data_rom[10]='h00000005;
rd_cycle[11] = 1'b0; wr_cycle[11] = 1'b1; addr_rom[
11]='h0000002c; wr_data_rom[11]='h0000000d;
rd_cycle[12] = 1'b0; wr_cycle[12] = 1'b1; addr_rom[
12]='h00000030; wr_data_rom[12]='h00000006;
rd_cycle[13] = 1'b0; wr_cycle[13] = 1'b1; addr_rom[
13]='h00000034; wr_data_rom[13]='h00000003;
rd_cycle[14] = 1'b0; wr_cycle[14] = 1'b1; addr_rom[
14]='h00000038; wr_data_rom[14]='h00000037;
rd_cycle[15] = 1'b0; wr_cycle[15] = 1'b1; addr_rom[
15]='h0000003c; wr_data_rom[15]='h0000003a;
// 48 random read and write cycles
rd_cycle[16] = 1'b0; wr_cycle[16] = 1'b1; addr_rom[
16]='h00000034; wr_data_rom[16]='h0000000c;
rd_cycle[17] = 1'b0; wr_cycle[17] = 1'b1; addr_rom[
17]='h0000000c; wr_data_rom[17]='h00000020;
rd_cycle[18] = 1'b0; wr_cycle[18] = 1'b1; addr_rom[
18]='h00000010; wr_data_rom[18]='h0000002f;
rd_cycle[19] = 1'b0; wr_cycle[19] = 1'b1; addr_rom[
19]='h0000003c; wr_data_rom[19]='h0000001c;
rd_cycle[20] = 1'b0; wr_cycle[20] = 1'b1; addr_rom[
20]='h00000020; wr_data_rom[20]='h0000001c;
rd_cycle[21] = 1'b1; wr_cycle[21] = 1'b0; addr_rom[
21]='h00000010; wr_data_rom[21]='h00000000;
rd_cycle[22] = 1'b0; wr_cycle[22] = 1'b1; addr_rom[
22]='h00000004; wr_data_rom[22]='h00000008;
rd_cycle[23] = 1'b1; wr_cycle[23] = 1'b0; addr_rom[
23]='h00000010; wr_data_rom[23]='h00000000;
rd_cycle[24] = 1'b0; wr_cycle[24] = 1'b1; addr_rom[
24]='h00000000; wr_data_rom[24]='h0000001c;
rd_cycle[25] = 1'b1; wr_cycle[25] = 1'b0; addr_rom[
25]='h00000028; wr_data_rom[25]='h00000000;
rd_cycle[26] = 1'b1; wr_cycle[26] = 1'b0; addr_rom[
26]='h00000038; wr_data_rom[26]='h00000000;
rd_cycle[27] = 1'b0; wr_cycle[27] = 1'b1; addr_rom[
27]='h00000014; wr_data_rom[27]='h00000039;
rd_cycle[28] = 1'b0; wr_cycle[28] = 1'b1; addr_rom[
28]='h00000028; wr_data_rom[28]='h00000016;
rd_cycle[29] = 1'b0; wr_cycle[29] = 1'b1; addr_rom[
29]='h0000003c; wr_data_rom[29]='h0000000c;
rd_cycle[30] = 1'b1; wr_cycle[30] = 1'b0; addr_rom[
30]='h00000000; wr_data_rom[30]='h00000000;
rd_cycle[31] = 1'b0; wr_cycle[31] = 1'b1; addr_rom[
31]='h00000000; wr_data_rom[31]='h00000034;
rd_cycle[32] = 1'b1; wr_cycle[32] = 1'b0; addr_rom[
32]='h0000000c; wr_data_rom[32]='h00000000;
rd_cycle[33] = 1'b1; wr_cycle[33] = 1'b0; addr_rom[
33]='h00000004; wr_data_rom[33]='h00000000;
rd_cycle[34] = 1'b0; wr_cycle[34] = 1'b1; addr_rom[
34]='h00000004; wr_data_rom[34]='h00000006;
rd_cycle[35] = 1'b0; wr_cycle[35] = 1'b1; addr_rom[
35]='h00000018; wr_data_rom[35]='h0000002b;

```

```

rd_cycle[ 36] = 1'b0; wr_cycle[ 36] = 1'b1; addr_rom[
36]='h00000010; wr_data_rom[ 36]='h0000003a;
rd_cycle[ 37] = 1'b1; wr_cycle[ 37] = 1'b0; addr_rom[
37]='h00000014; wr_data_rom[ 37]='h00000000;
rd_cycle[ 38] = 1'b0; wr_cycle[ 38] = 1'b1; addr_rom[
38]='h00000000; wr_data_rom[ 38]='h00000038;
rd_cycle[ 39] = 1'b0; wr_cycle[ 39] = 1'b1; addr_rom[
39]='h00000008; wr_data_rom[ 39]='h0000002a;
rd_cycle[ 40] = 1'b1; wr_cycle[ 40] = 1'b0; addr_rom[
40]='h00000034; wr_data_rom[ 40]='h00000000;
rd_cycle[ 41] = 1'b0; wr_cycle[ 41] = 1'b1; addr_rom[
41]='h00000008; wr_data_rom[ 41]='h00000025;
rd_cycle[ 42] = 1'b0; wr_cycle[ 42] = 1'b1; addr_rom[
42]='h0000003c; wr_data_rom[ 42]='h00000011;
rd_cycle[ 43] = 1'b0; wr_cycle[ 43] = 1'b1; addr_rom[
43]='h00000010; wr_data_rom[ 43]='h00000009;
rd_cycle[ 44] = 1'b0; wr_cycle[ 44] = 1'b1; addr_rom[
44]='h00000028; wr_data_rom[ 44]='h0000003a;
rd_cycle[ 45] = 1'b0; wr_cycle[ 45] = 1'b1; addr_rom[
45]='h0000001c; wr_data_rom[ 45]='h00000017;
rd_cycle[ 46] = 1'b0; wr_cycle[ 46] = 1'b1; addr_rom[
46]='h00000020; wr_data_rom[ 46]='h00000029;
rd_cycle[ 47] = 1'b1; wr_cycle[ 47] = 1'b0; addr_rom[
47]='h00000018; wr_data_rom[ 47]='h00000000;
rd_cycle[ 48] = 1'b0; wr_cycle[ 48] = 1'b1; addr_rom[
48]='h00000008; wr_data_rom[ 48]='h0000003b;
rd_cycle[ 49] = 1'b0; wr_cycle[ 49] = 1'b1; addr_rom[
49]='h00000030; wr_data_rom[ 49]='h0000002d;
rd_cycle[ 50] = 1'b1; wr_cycle[ 50] = 1'b0; addr_rom[
50]='h00000004; wr_data_rom[ 50]='h00000000;
rd_cycle[ 51] = 1'b1; wr_cycle[ 51] = 1'b0; addr_rom[
51]='h00000030; wr_data_rom[ 51]='h00000000;
rd_cycle[ 52] = 1'b0; wr_cycle[ 52] = 1'b1; addr_rom[
52]='h0000000c; wr_data_rom[ 52]='h0000001c;
rd_cycle[ 53] = 1'b1; wr_cycle[ 53] = 1'b0; addr_rom[
53]='h0000000c; wr_data_rom[ 53]='h00000000;
rd_cycle[ 54] = 1'b1; wr_cycle[ 54] = 1'b0; addr_rom[
54]='h00000010; wr_data_rom[ 54]='h00000000;
rd_cycle[ 55] = 1'b1; wr_cycle[ 55] = 1'b0; addr_rom[
55]='h00000018; wr_data_rom[ 55]='h00000000;
rd_cycle[ 56] = 1'b0; wr_cycle[ 56] = 1'b1; addr_rom[
56]='h00000018; wr_data_rom[ 56]='h0000003c;
rd_cycle[ 57] = 1'b1; wr_cycle[ 57] = 1'b0; addr_rom[
57]='h00000010; wr_data_rom[ 57]='h00000000;
rd_cycle[ 58] = 1'b1; wr_cycle[ 58] = 1'b0; addr_rom[
58]='h0000003c; wr_data_rom[ 58]='h00000000;
rd_cycle[ 59] = 1'b1; wr_cycle[ 59] = 1'b0; addr_rom[
59]='h00000038; wr_data_rom[ 59]='h00000000;
rd_cycle[ 60] = 1'b1; wr_cycle[ 60] = 1'b0; addr_rom[
60]='h00000000; wr_data_rom[ 60]='h00000000;
rd_cycle[ 61] = 1'b0; wr_cycle[ 61] = 1'b1; addr_rom[
61]='h00000034; wr_data_rom[ 61]='h0000003c;
rd_cycle[ 62] = 1'b1; wr_cycle[ 62] = 1'b0; addr_rom[
62]='h00000038; wr_data_rom[ 62]='h00000000;
rd_cycle[ 63] = 1'b0; wr_cycle[ 63] = 1'b1; addr_rom[
63]='h00000004; wr_data_rom[ 63]='h00000021;

```

```
// 16 silence cycles
```

```

rd_cycle[ 64] = 1'b0; wr_cycle[ 64] = 1'b0; addr_rom[
64]='h00000000; wr_data_rom[ 64]='h00000000;
rd_cycle[ 65] = 1'b0; wr_cycle[ 65] = 1'b0; addr_rom[
65]='h00000000; wr_data_rom[ 65]='h00000000;
rd_cycle[ 66] = 1'b0; wr_cycle[ 66] = 1'b0; addr_rom[
66]='h00000000; wr_data_rom[ 66]='h00000000;
rd_cycle[ 67] = 1'b0; wr_cycle[ 67] = 1'b0; addr_rom[
67]='h00000000; wr_data_rom[ 67]='h00000000;
rd_cycle[ 68] = 1'b0; wr_cycle[ 68] = 1'b0; addr_rom[
68]='h00000000; wr_data_rom[ 68]='h00000000;
rd_cycle[ 69] = 1'b0; wr_cycle[ 69] = 1'b0; addr_rom[
69]='h00000000; wr_data_rom[ 69]='h00000000;
rd_cycle[ 70] = 1'b0; wr_cycle[ 70] = 1'b0; addr_rom[
70]='h00000000; wr_data_rom[ 70]='h00000000;
rd_cycle[ 71] = 1'b0; wr_cycle[ 71] = 1'b0; addr_rom[
71]='h00000000; wr_data_rom[ 71]='h00000000;
rd_cycle[ 72] = 1'b0; wr_cycle[ 72] = 1'b0; addr_rom[
72]='h00000000; wr_data_rom[ 72]='h00000000;
rd_cycle[ 73] = 1'b0; wr_cycle[ 73] = 1'b0; addr_rom[
73]='h00000000; wr_data_rom[ 73]='h00000000;
rd_cycle[ 74] = 1'b0; wr_cycle[ 74] = 1'b0; addr_rom[
74]='h00000000; wr_data_rom[ 74]='h00000000;
rd_cycle[ 75] = 1'b0; wr_cycle[ 75] = 1'b0; addr_rom[
75]='h00000000; wr_data_rom[ 75]='h00000000;
rd_cycle[ 76] = 1'b0; wr_cycle[ 76] = 1'b0; addr_rom[
76]='h00000000; wr_data_rom[ 76]='h00000000;
rd_cycle[ 77] = 1'b0; wr_cycle[ 77] = 1'b0; addr_rom[
77]='h00000000; wr_data_rom[ 77]='h00000000;
rd_cycle[ 78] = 1'b0; wr_cycle[ 78] = 1'b0; addr_rom[
78]='h00000000; wr_data_rom[ 78]='h00000000;
rd_cycle[ 79] = 1'b0; wr_cycle[ 79] = 1'b0; addr_rom[
79]='h00000000; wr_data_rom[ 79]='h00000000;
// 16 sequence read cycles
rd_cycle[ 80] = 1'b1; wr_cycle[ 80] = 1'b0; addr_rom[
80]='h00000000; wr_data_rom[ 80]='h00000000;
rd_cycle[ 81] = 1'b1; wr_cycle[ 81] = 1'b0; addr_rom[
81]='h00000004; wr_data_rom[ 81]='h00000000;
rd_cycle[ 82] = 1'b1; wr_cycle[ 82] = 1'b0; addr_rom[
82]='h00000008; wr_data_rom[ 82]='h00000000;
rd_cycle[ 83] = 1'b1; wr_cycle[ 83] = 1'b0; addr_rom[
83]='h0000000c; wr_data_rom[ 83]='h00000000;
rd_cycle[ 84] = 1'b1; wr_cycle[ 84] = 1'b0; addr_rom[
84]='h00000010; wr_data_rom[ 84]='h00000000;
rd_cycle[ 85] = 1'b1; wr_cycle[ 85] = 1'b0; addr_rom[
85]='h00000014; wr_data_rom[ 85]='h00000000;
rd_cycle[ 86] = 1'b1; wr_cycle[ 86] = 1'b0; addr_rom[
86]='h00000018; wr_data_rom[ 86]='h00000000;
rd_cycle[ 87] = 1'b1; wr_cycle[ 87] = 1'b0; addr_rom[
87]='h0000001c; wr_data_rom[ 87]='h00000000;
rd_cycle[ 88] = 1'b1; wr_cycle[ 88] = 1'b0; addr_rom[
88]='h00000020; wr_data_rom[ 88]='h00000000;
rd_cycle[ 89] = 1'b1; wr_cycle[ 89] = 1'b0; addr_rom[
89]='h00000024; wr_data_rom[ 89]='h00000000;
rd_cycle[ 90] = 1'b1; wr_cycle[ 90] = 1'b0; addr_rom[
90]='h00000028; wr_data_rom[ 90]='h00000000;
rd_cycle[ 91] = 1'b1; wr_cycle[ 91] = 1'b0; addr_rom[
91]='h0000002c; wr_data_rom[ 91]='h00000000;

```

```

    rd_cycle[ 92] = 1'b1; wr_cycle[ 92] = 1'b0; addr_rom[
92]='h00000030; wr_data_rom[ 92]='h00000000;
    rd_cycle[ 93] = 1'b1; wr_cycle[ 93] = 1'b0; addr_rom[
93]='h00000034; wr_data_rom[ 93]='h00000000;
    rd_cycle[ 94] = 1'b1; wr_cycle[ 94] = 1'b0; addr_rom[
94]='h00000038; wr_data_rom[ 94]='h00000000;
    rd_cycle[ 95] = 1'b1; wr_cycle[ 95] = 1'b0; addr_rom[
95]='h0000003c; wr_data_rom[ 95]='h00000000;
end

initial begin
    validation_data[ 0] = 'h00000038;
    validation_data[ 1] = 'h00000021;
    validation_data[ 2] = 'h0000003b;
    validation_data[ 3] = 'h0000001c;
    validation_data[ 4] = 'h00000009;
    validation_data[ 5] = 'h00000039;
    validation_data[ 6] = 'h0000003c;
    validation_data[ 7] = 'h00000017;
    validation_data[ 8] = 'h00000029;
    validation_data[ 9] = 'h00000000;
    validation_data[10] = 'h0000003a;
    validation_data[11] = 'h0000000d;
    validation_data[12] = 'h0000002d;
    validation_data[13] = 'h0000003c;
    validation_data[14] = 'h00000037;
    validation_data[15] = 'h00000011;

end

reg clk = 1'b1, rst = 1'b1;
initial #4 rst = 1'b0;
always #1 clk = ~clk;

wire miss;
wire [31:0] rd_data;
reg [31:0] index = 0, wr_data = 0, addr = 0;
reg rd_req = 1'b0, wr_req = 1'b0;
reg rd_req_ff = 1'b0, miss_ff = 1'b0;
reg [31:0] validation_count = 0;

always @ (posedge clk or posedge rst)
    if(rst) begin
        rd_req_ff <= 1'b0;
        miss_ff    <= 1'b0;
    end else begin
        rd_req_ff <= rd_req;
        miss_ff    <= miss;
    end
end

always @ (posedge clk or posedge rst)
    if(rst) begin
        validation_count <= 0;
    end else begin
        if(validation_count>=`DATA_COUNT) begin
            validation_count <= 'hfffffff;
        end else if(rd_req_ff && (index>(4*`DATA_COUNT))) begin

```



```

        if(~miss_ff) begin
            if(validation_data[validation_count]==rd_data)
                validation_count <= validation_count+1;
            else
                validation_count <= 0;
        end
    end else begin
        validation_count <= 0;
    end
end

always @ (posedge clk or posedge rst)
    if(rst) begin
        index    <= 0;
        wr_data  <= 0;
        addr     <= 0;
        rd_req   <= 1'b0;
        wr_req   <= 1'b0;
    end else begin
        if(~miss) begin
            if(index<`RDWR_COUNT) begin
                if(wr_cycle[index]) begin
                    rd_req  <= 1'b0;
                    wr_req  <= 1'b1;
                end else if(rd_cycle[index]) begin
                    wr_data <= 0;
                    rd_req  <= 1'b1;
                    wr_req  <= 1'b0;
                end else begin
                    wr_data <= 0;
                    rd_req  <= 1'b0;
                    wr_req  <= 1'b0;
                end
                wr_data <= wr_data_rom[index];
                addr    <= addr_rom[index];
                index <= index + 1;
            end else begin
                wr_data <= 0;
                addr    <= 0;
                rd_req  <= 1'b0;
                wr_req  <= 1'b0;
            end
        end
    end

cache #(
    .LINE_ADDR_LEN  ( 3 ),
    .SET_ADDR_LEN   ( 2 ),
    .TAG_ADDR_LEN   ( 12 ),
    .WAY_CNT        ( 3 )
) cache_test_instance (
    .clk             ( clk ),
    .rst             ( rst ),
    .miss            ( miss ),
    .addr            ( addr ),
    .rd_req          ( rd_req ),
    .rd_data         ( rd_data ),
    .wr_req          ( wr_req ),

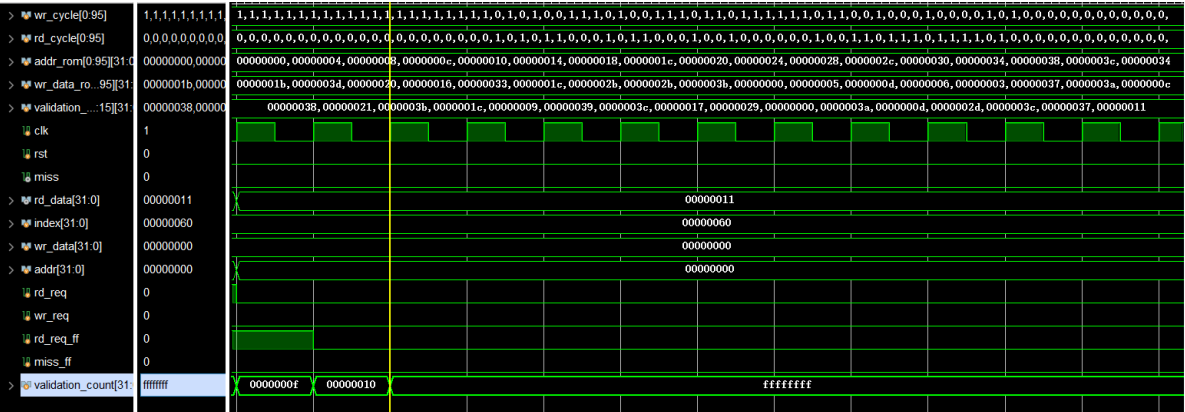
```

```
        .wr_data      ( wr_data      )
    );

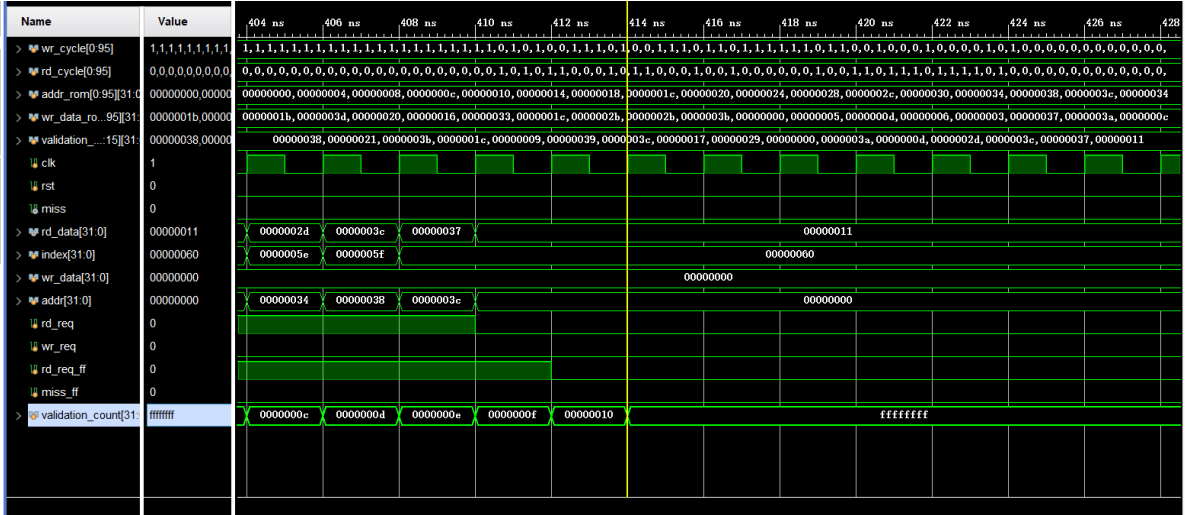
endmodule
```

LRU 和 FIFO 仿真结果

LRU



FIFO



可以看出，validation_count 从 0 递增到16，最后变为 -1，保持不变

阶段2

将阶段 1 的 cache 接入 lab2 的 CPU，修改 WBSegReg.v、IDSegReg.v、HazardUnit.v、RV32Core.v

WBSegtReg.v

- 增加 CacheMiss 信号

```
output wire CacheMiss
```

- 将之前的 DataRam 替换为 cache

```
cache #(
```

```

.LINE_ADDR_LEN ( 3 ),
.SET_ADDR_LEN ( 3 ),
.TAG_ADDR_LEN ( 6 ),
.WAY_CNT ( 4 ),
.STRATEGY ( 1'b0 )
) cache_test_instance (
.clk ( clk ),
.rst ( clear ),
.miss ( CacheMiss ),
.addr ( A ),
.rd_req ( MemToRegM[0] ),
.rd_data ( RD_raw ),
.wr_req ( |WE ),
.wr_data ( WD )
);

```

- 统计cache hit 和 cache miss 的次数

用 last_addr 记录上一次访存的地址。如果此次访存地址和上次不同，则根据 cache 的 cache_miss 信号将 hit_count 或 miss_count 加一。注意判断条件 `if(cache_rd_wr & (last_addr != A))` 主要是为了保证一次访存只被统计一次

```

// 统计 Cache 缺失率
reg [31:0] hit_count = 0, miss_count = 0;
// 最后读or写的地址
reg [31:0] last_addr = 0;
// 是否有对 Cache 的读or写请求
wire cache_rd_wr = (|WE) | MemToRegM[0];
always @(posedge clk or posedge clear)
begin
    if(clear)
    begin
        last_addr <= 0;
    end
    else
    begin
        if(cache_rd_wr)
        begin
            // 如果命中，则 last_addr == A
            last_addr <= A;
        end
    end
end

always @(posedge clk or posedge clear)
begin
    if(clear)
    begin
        hit_count <= 0;
        miss_count <= 0;
    end
    else
    begin
        // 如果有对 Cache 的读写请求
        if(cache_rd_wr & (last_addr != A))
        begin
            if(CacheMiss)

```

```

        miss_count <= miss_count + 1;
    else
        hit_count <= hit_count + 1;
    end
end
end
end

```

IDSegReg.v

- 首先修改群里提出的源代码中的小错误

```

always @ (posedge clk)
begin
    stall_ff<=~en;
    clear_ff<=clear;
    // lab3 modify
    RD_old<=RD;
end

```

- 将原本的 InstructionRam 替换为 InstructionCache，接口主要通过观察助教提供的 Python 脚本生成的 InstructionCache.v 来填写，和 InstructionRam 接口类似

```

InstructionCache InstructionRam (
    .clk      ( clk      ),
    .write_en( |WE2      ),
    .addr     ( A[31:2]  ),
    .data     ( RD_raw   )
);

```

HazardUnit.v

- 增加 CacheMiss 相关的信号和冲突处理。当发生 cache miss 时，bubble 当前指令及之后的指令

```

else if(DCacheMiss | ICacheMiss)
begin
    StallF <= 1'b1;
    FlushF <= 1'b0;
    StallD <= 1'b1;
    FlushD <= 1'b0;
    StallE <= 1'b1;
    FlushE <= 1'b0;
    StallM <= 1'b1;
    FlushM <= 1'b0;
    StallW <= 1'b1;
    FlushW <= 1'b0;
end

```

RV32Core.v

- 连接和 cache 有关的线路, 包括 `WBSegReg`、`HazardUnit`

```
WBSegReg WBSegReg1(  
    .clk(CPU_CLK),  
    .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),  
    //CSR  
    .CSRAddrM(CSRAddrM),  
    .CSRAddrW(CSRAddrW),  
    .CSROutM(CSROutM),  
    .CSROutW(CSROutW),  
    .CSRWriteM(CSRWriteM),  
    .CSRWriteW(CSRWriteW)  
);
```














```
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),  
    .StallF(StallF),
```

```
.FlushF(FlushF),
.StallD(StallD),
.FlushD(FlushD),
.StallE(StallE),
.FlushE(FlushE),
.StallM(StallM),
.FlushM(FlushM),
.StallW(StallW),
.FlushW(FlushW),
.Forward1E(Forward1E),
.Forward2E(Forward2E),
//CSR
.Forward3E(Forward3E),
.CSRSrcE(CSRAddrE),
.CSRSrcM(CSRAddrM),
.CSRSrcW(CSRAddrW),
.CSRReadE(CSRReadE),
.CSRWriteM(CSRWriteM),
.CSRWriteW(CSRWriteW)
);
```

结果





















使用实验指导文档中的指令生成 testbench

- FIFO 矩阵乘

▼  ram_cell[0:4095][31:0]	b77c7952,990
>  [0][31:0]	b77c7952
>  [1][31:0]	996142f5
>  [2][31:0]	968cdec9
>  [3][31:0]	06e2db2f
>  [4][31:0]	a8192a9e
>  [5][31:0]	721a6192
>  [6][31:0]	ad036e69
>  [7][31:0]	9d42a8f3
>  [8][31:0]	1d7b5c10
>  [9][31:0]	d2cab7bb
>  [10][31:0]	527b2abe
>  [11][31:0]	d961cf2c
>  [12][31:0]	d73618f5
>  [13][31:0]	14dfb33f
>  [14][31:0]	17e4b572
>  [15][31:0]	8a978fc5
>  [16][31:0]	77e5359a
>  [17][31:0]	704b342f
>  [18][31:0]	aac4dd09
>  [19][31:0]	d66493aa

























mem.sv中的ram_cell变量与注释中相同

- LRU 矩阵乘

>  [0][31:0]	b77c7952
>  [1][31:0]	996142f5
>  [2][31:0]	968cdec9
>  [3][31:0]	06e2db2f
>  [4][31:0]	a8192a9e
>  [5][31:0]	721a6192
>  [6][31:0]	ad036e69
>  [7][31:0]	9d42a8f3
>  [8][31:0]	1d7b5c10
>  [9][31:0]	d2cab7bb
>  [10][31:0]	527b2abe
>  [11][31:0]	d961cf2c
>  [12][31:0]	d73618f5
>  [13][31:0]	14dfb33f
>  [14][31:0]	17e4b572
>  [15][31:0]	8a978fc5
>  [16][31:0]	77e5359a
>  [17][31:0]	704b342f
>  [18][31:0]	aac4dd09
>  [19][31:0]	d66493aa

























mem.sv中的ram_cell变量与注释中相同

- FIFO QuickSort

▼  ram_cell[0:4095][31:0]	00000000,000
>  [0][31:0]	00000000
>  [1][31:0]	00000001
>  [2][31:0]	00000002
>  [3][31:0]	00000003
>  [4][31:0]	00000004
>  [5][31:0]	00000005
>  [6][31:0]	00000006
>  [7][31:0]	00000007
>  [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

mem.sv 中原本乱序的 ram_cell 最终变为有序

- LRU QuickSort

▼  ram_cell[0:4095][31:0]	00000000,00
>  [0][31:0]	00000000
>  [1][31:0]	00000001
>  [2][31:0]	00000002
>  [3][31:0]	00000003
>  [4][31:0]	00000004
>  [5][31:0]	00000005
>  [6][31:0]	00000006
>  [7][31:0]	00000007
>  [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

mem.sv 中原本乱序的 ram_cell 最终变为有序

阶段3

使用我们提供的快速排序和矩阵乘法的benchmark进行实验，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）。提示：为了方便进行性能评估，建议用上阶段二的缺失率统计功能

cache 资源占用对比

使用阶段 1 的代码进行资源占用对比，将cache模块作为顶层进行综合，修改 cache.sv 中的各个 cache 参数为要综合的参数

修改参数时，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，需要固定主存的大小，主存大小是 $2^{(LINE_ADDR_LEN+SET_ADDR_LEN+TAG_ADDR_LEN)}$ 个字。

LRU

下表第一行

Resource	Utilization	Available	Utilization %
LUT	3775	63400	5.95
FF	6014	126800	4.74
BRAM	4	135	2.96
IO	81	210	38.57

下表第二行

Resource	Utilization	Available	Utilization %
LUT	2613	63400	4.12
FF	5717	126800	4.51
BRAM	4	135	2.96
IO	81	210	38.57

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

FIFO

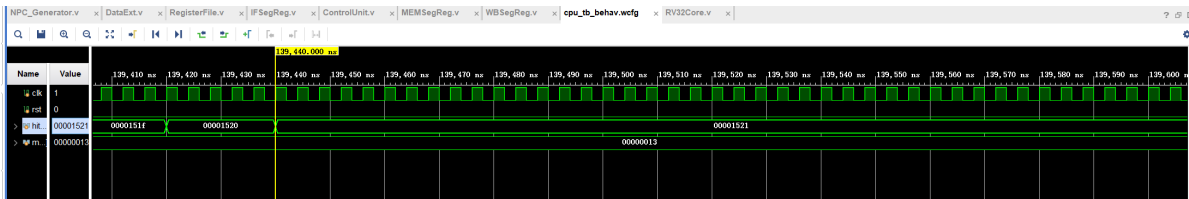
LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	LUT	FF
2	3	7	4	2429	6004
3	3	6	2	1986	5717
3	3	6	4	3986	10451
3	3	6	8	9030	20006
3	2	7	4	3587	5731
3	4	5	4	7199	19872
4	3	5	4	8421	19402

从上面的表可以观察到

- 在所有参数相同的情况下，两者所占用的 FF 数量相差不大，但LRU 占用的 LUT 比 FIFO 多，也就是说相同参数情况下 LRU 策略的 cache 电路面积更大。
- 主存大小不变的前提下
 - 组相连度越大，占用的 LUT 和 FF 越多，电路面积越大
 - 组数越大，占用的 LUT 和 FF 越多，电路面积越大
 - 每个 line 越大，占用的 LUT 和 FF 越多，电路面积越大

cache 性能对比

主要观察的波形如下，其余类似填入表中



LRU + QuickSort 256

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Times(ns)	Miss	Hit	Miss Rate
2	3	7	4	233,008	276	5152	5.08%
3	3	6	2	186,360	139	5289	2.56%
3	3	6	4	161,540	78	5350	1.44%
3	3	6	8	143,760	39	5389	0.72%
3	2	7	4	187,000	141	5287	2.60%
3	4	5	4	143,760	39	5389	0.72%
4	3	5	4	139,440	19	5409	0.35%

LRU + MatrixMul 16 * 16

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Times(ns)	Miss	Hit	Miss Rate
2	3	7	4	1,337,224	4744	3960	54.50%
3	3	6	2	1,321,640	4672	4032	53.68%
3	3	6	4	646,432	1547	7157	17.77%
3	3	6	8	291,484	123	8581	1.41%
3	2	7	4	1,321,640	4672	4032	53.68%
3	4	5	4	289,788	119	8584	13.67%
4	3	5	4	273,056	56	8648	0.64%

FIFO + QuickSort 256

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Times(ns)	Miss	Hit	Miss Rate
2	3	7	4	241,696	296	5132	5.45%
3	3	6	2	189,328	146	5282	2.69%
3	3	6	4	157,300	68	5360	1.25%
3	3	6	8	143,760	39	5386	0.72%
3	2	7	4	192,936	155	5273	2.86%
3	4	5	4	143,760	39	5386	0.72%
4	3	5	4	139,440	19	5409	0.35%

FIFO + MatrixMul 16 * 16

LINE_ADDR_LEN	SET_ADDR_LEN	TAG_ADDR_LEN	WAY_CNT	Times(ns)	Miss	Hit	Miss Rate
2	3	7	4	1,340,680	4760	3944	54.69%
3	3	6	2	1,363,112	4864	3840	55.88%
3	3	6	4	687,904	1739	6965	19.98%
3	3	6	8	294,576	146	8558	1.68%
3	2	7	4	1,349,288	4800	3904	55.14%
3	4	5	4	293,728	144	8560	1.65%
4	3	5	4	276,512	72	8632	0.83%

- 整体来看，当其他参数相同时，由于 MatrixMul 的局部性比 QuickSort，所以 MatrixMul 的 Cache 缺失率和运行时间都远大于 QuickSort
- 组数、line 的大小、组相连度的影响（相同的测试程序且主存大小不变的情况下）
 - 组数越大，运行时间越短，访存效果越好，Cache 缺失率越小
 - 每个 line 越大，运行时间越短，访存效果越好，Cache 缺失率越小
 - 组相连度越大，运行时间越短，访存效果越好，Cache 缺失率越小
- 另外，通过表格可以看出：加倍组数、增加 line 大小、加倍组相连度都可以优化访存性能。效果各异
- 权衡性能（运行仿真时的时钟周期数）和电路面积给出一个较优的cache参数和策略：
 - 对于 QuickSort，由于其局部性较好，所以 cache 缺失率普遍较低，使用 LRU 和 FIFO 区别不大，但是考虑到 LRU 使用的 cache 电路面积更大，因此采用 FIFO 策略。
 - 性能：参数组合为 (4,3,5,4) 时性能最优，cache 缺失率也最低。另外参数组合 (3,3,6,8) 和 (3,4,5,4) 性能和 cache 缺失率也较好
 - 电路面积：上述三者基本相同（(3,4,5,4) 使用的 LUT 在三者中最小）

因此选择参数: LINE_ADDR_LEN=4, SET_ADDR_LEN=3, TAG_ADDR_LEN=5, WAY_CNT=4
, 策略: FIFO

- 对于 MatrixMul, LRU 的整体性能优于 FIFO, 虽然 LRU 会消耗更多的资源 (LUT使用略多, FF 区别不大), 但是综合考虑还是使用 LRU 策略
 - 性能: 参数组合为 (4,3,5,4) 时性能最优, cache 缺失率最低。另外参数组合 (3,3,6,8) 性能和缺失率也较好
 - 电路面积: 上述二者基本相同

因此选择参数: LINE_ADDR_LEN=4, SET_ADDR_LEN=3, TAG_ADDR_LEN=5, WAY_CNT=4
, 策略: LRU

实验总结

- 通过本次实验, 我对 cache 的实现、LRU 和 FIFO 的区别有了更加深刻的理解。
- 在阶段 3 分析中, 可以直观地看出由于不同程序具有不同的局部性, 导致 cache 性能上存在差异。
- 在阶段 3 分析中, 可以看出 cache 不同的配置参数也会很大程度上影响 cache 的性能
- 在阶段 3 分析中, 对不同的策略、不同的程序、不同的参数组合进行了多次仿真和模拟, 对比了资源占用和性能上的差异。可以发现 LRU 的资源占用要比 FIFO 多, 因为 LRU 在每次 cache 命中时都需要更新时间戳。但是相应的 LRU 的性能一般也优于 FIFO。所以不同的策略有不同的优点和侧重, 本质上都是一种 trade off
- 通过本次实验, 我对 Verilog 的使用也更加熟悉。对 CPU 中 cache 的工作原理也有了实践上的认识, 收获颇丰

改进意见

- 阶段 3 分析时做了很多重复的测试, 如果能有什么脚本可以一体化批量测试就好了