Lab 3 report

PB21020485 吴敌

实验内容

- 将直接映射DCache改写为组相连DCache，使用FIFO和LRU两种替换算法进行改写
- 将改写好的DCache接入流水线
- 调整DCache各项参数，统计不同配置的DCache的性能

DCache设计

组相连DCache设计

直接相联的DCache已经给了：

- 命中逻辑：需要对每一路Tag进行并行比较

```
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；好像不能综合的写法，但事实不能使用
parameter
    begin
        cache_hit = 1'b1;
        way_addr = i;
    end
```

- FIFO替换算法：为了方便起见，同时也没有过多影响资源使用，我们选择维护一个时间寄存器，每放入就将时间置为0；当然也可以使用寄存器记录位置，不过资源没有节约很多

```
begin
        if (cache_stat == SWAP_IN_OK)
        begin
            for (integer k = 0; k < WAY_CNT; k++)
            begin
                if (k == way_addr_last)
                    LFtime[set_addr][k] <= 32'b0;
                else LFtime[set_addr][k] <= LFtime[set_addr][k] + 1;
            end
        end
    end
```

值得注意的是，我们需要对这个时间赋予一个初值。一开始我没有赋给初值，cache测试也能通过，但是事实上只用了第一个块进行存储，很难debug，浪费了很多时间

```
initial begin
    for (integer j = 0; j < SET_SIZE; j++)begin
    for (integer i = 0; i < WAY_CNT; i++)
    begin
        LFtime[j][i] = 32'b0;
    end
end
end
```

- LRU替换算法：我们使用了时间寄存器，在命中和换入时会将时间置为0

```
begin
        if (rw && cache_stat == IDLE && cache_hit)
        begin
            for (integer k = 0; k < WAY_CNT; k++)
            begin
                if (k == way_addr)
                    LFtime[set_addr][k] <= 32'b0;
                else LFtime[set_addr][k] <= LFtime[set_addr][k] + 1;
            end
        end
        else if (cache_stat == SWAP_IN_OK)
        begin
            for (integer k = 0; k < WAY_CNT; k++)
            begin
                if (k == way_addr_last)
                    LFtime[set_addr][k] <= 32'b0;
                else LFtime[set_addr][k] <= LFtime[set_addr][k] + 1;
            end
        end
```

```
    end
```

DCache加入流水线CPU

DCache接入流水线后，需要使流水线能够在Cache Miss时能停下。由此，我们需要做如下修改：

- 将miss信号接入Hazard，并在miss时，停顿所有段间寄存器
- 做miss和读写信号的统计，我们做了一个边沿检测来实现统计数量

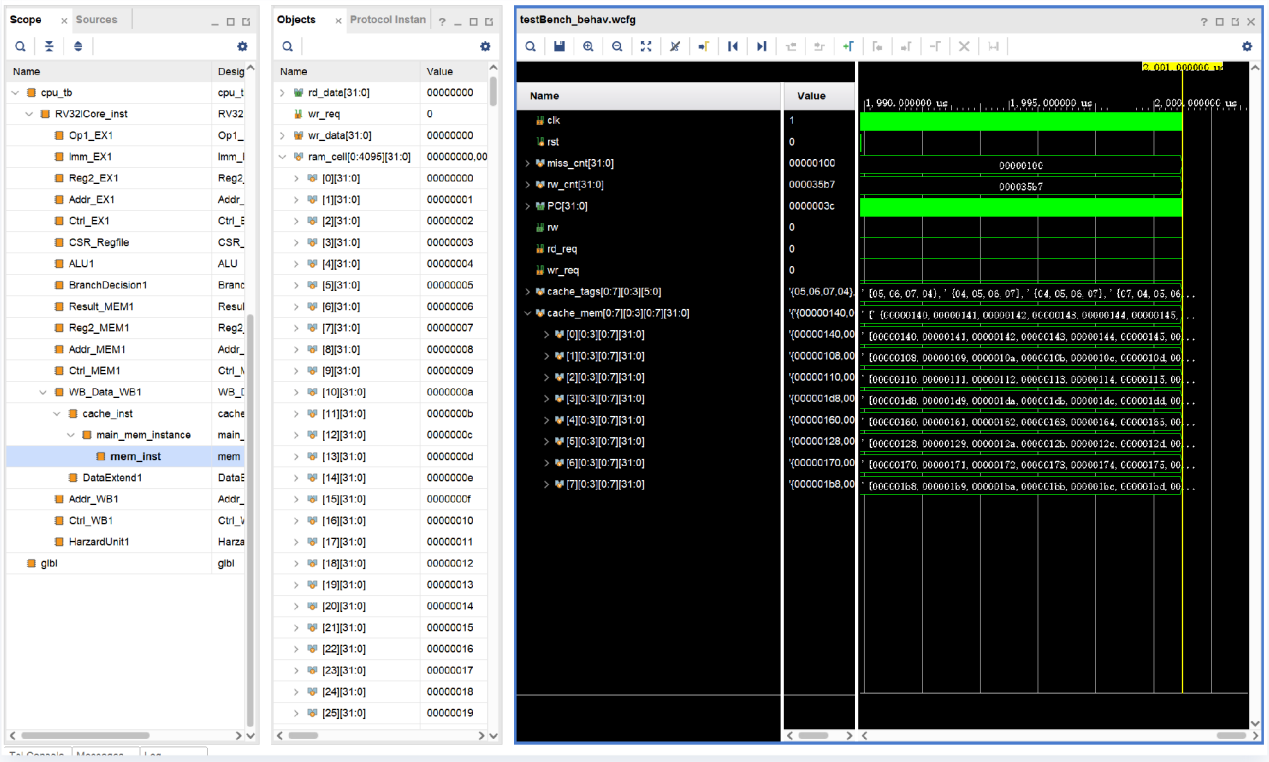
```
assign miss_edge = miss & ~miss_last;
//统计
always @ (posedge CPU_CLK or posedge CPU_RST)
begin
    if(CPU_RST)
    begin
        miss_cnt <= 0;
        rw_cnt <= 0;
    end
    else
    begin
        miss_last <= miss;
        if(miss_edge)
            miss_cnt <= miss_cnt + 1;
        if(rw)
            rw_cnt <= rw_cnt + 1;
    end
end
```

DCache性能测试

QuickSort

算法对命中率的影响

首先是仿真结果，我们生成了一个512数字的快排进行测试，下面是仿真结果。很容易看到结果的mem都被排成了有序的数：



测试替换算法对于Cache性能的影响：

No	替换算法	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
1	LRU	4	3	3	6	248	13751	0.0180
1	FIFO	4	3	3	6	256	13751	0.0186
2	LRU	2	3	3	6	357	13751	0.0260
2	FIFO	2	3	3	6	373	13751	0.0271

可以基本得出结论：LRU算法略强于FIFO算法，这也符合我们算法实现之初的估计：

容量对命中率的影响

下面的实验中，我们都使用比较好的LRU算法来进行探索：

改变LINE_LEN来统计：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
1	2	1	3	8	2545	13751	0.1851
2	2	2	3	7	927	13751	0.0674
3	2	3	3	6	357	13751	0.0260
4	2	4	3	5	132	13751	0.0096

这时可以得出结论：当容量逐渐增大时，命中率逐渐提升，但提升速度逐渐变得缓慢。当容量大到可以囊括内存中大部分数的时候，这个时候cache基本全是强制缺失和冲突缺失。

组相连度和组数、行长度对命中率的影响

我们将Cache容量设置为0+3+3，改变其余几个参数进行实验：

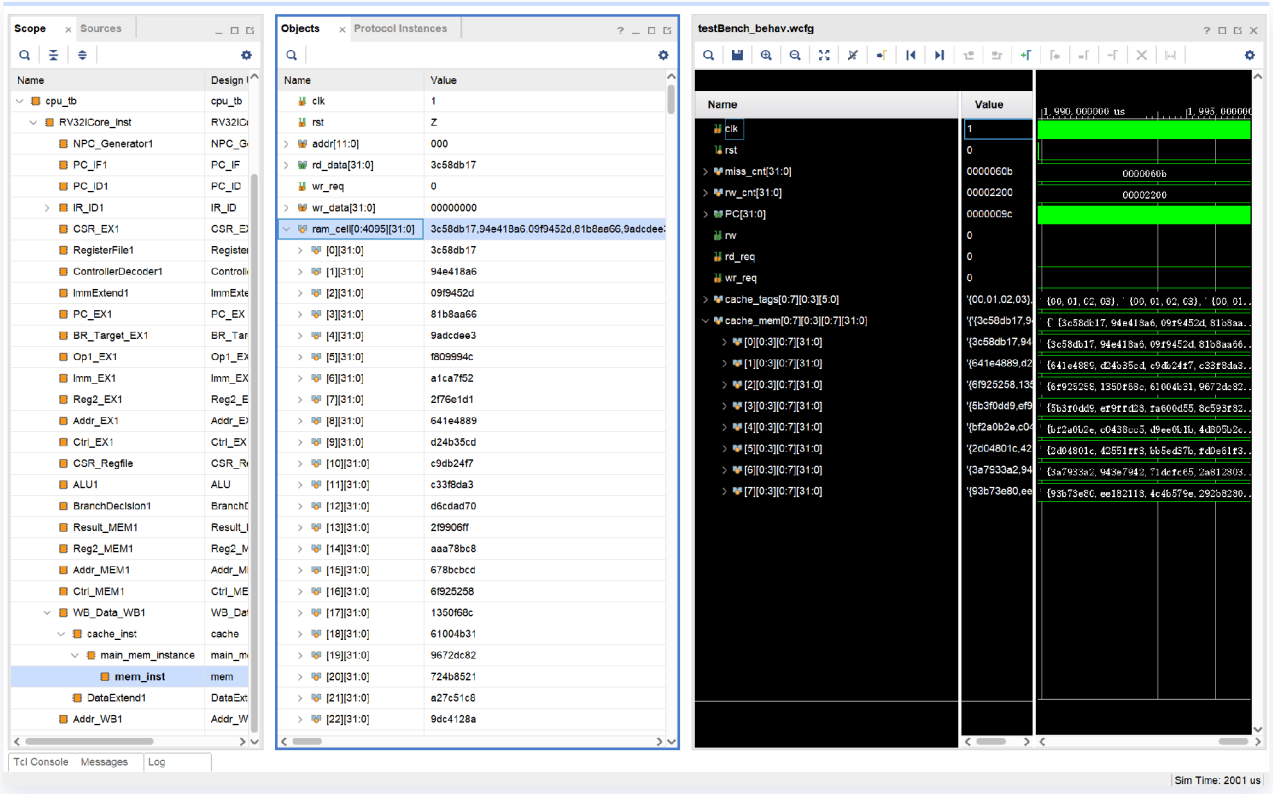
No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
1	1	1	5	6	2098	13751	0.1526
1	1	2	4	6	1140	13751	0.0829
1	1	3	3	6	824	13751	0.0599
1	1	4	2	6	742	13751	0.0539
2	2	1	4	7	1826	13751	0.1327
2	2	2	3	7	927	13751	0.0674
2	2	3	2	7	512	13751	0.0372
2	2	4	1	7	322	13751	0.0234
3	4	1	3	8	1859	13751	0.1352
3	4	2	2	8	942	13751	0.0685
3	4	3	1	8	519	13751	0.0377

可以看出，该cache容量下**2路+4位行地址+1位组地址**构成了QuickSort程序的最佳Cache配置。

同时我们也可以发现：**当Cache容量一定时，LINE_SIZE越大，命中率就会越高。**

MatMul

我们先将inst和mem代码放入，容易对比发现，执行结果和给出python结果相同，验证了我们cache的正确性：



```
initial begin
// dst matrix C
ram_cell[       0] = 32'h0;  // 32'h3c58db17;
ram_cell[       1] = 32'h0;  // 32'h94e418a6;
ram_cell[       2] = 32'h0;  // 32'h09f9452d;
ram_cell[       3] = 32'h0;  // 32'h81b8aa66;
ram_cell[       4] = 32'h0;  // 32'h9adcdee3;
ram_cell[       5] = 32'h0;  // 32'hf809994c;
ram_cell[       6] = 32'h0;  // 32'ha1ca7f52;
ram_cell[       7] = 32'h0;  // 32'h2f76e1d1;
ram_cell[       8] = 32'h0;  // 32'h641e4889;
ram_cell[       9] = 32'h0;  // 32'hd24b35cd;
ram_cell[      10] = 32'h0;  // 32'hc9db24f7;
ram_cell[      11] = 32'h0;  // 32'hc33f8da3;
ram_cell[      12] = 32'h0;  // 32'hd6cdad70;
ram_cell[      13] = 32'h0;  // 32'h2f9906ff;
ram_cell[      14] = 32'h0;  // 32'haaa78bc8;
ram_cell[      15] = 32'h0;  // 32'h678bcbcd;
ram_cell[      16] = 32'h0;  // 32'h6f925258;
ram_cell[      17] = 32'h0;  // 32'h1350f68c;
ram_cell[      18] = 32'h0;  // 32'h61004b31;
ram_cell[      19] = 32'h0;  // 32'h9672dc82;
ram_cell[      20] = 32'h0;  // 32'h724b8521;
ram_cell[      21] = 32'h0;  // 32'ha27c51c8;
ram_cell[      22] = 32'h0;  // 32'h9dc4128a;
ram_cell[      23] = 32'h0;  // 32'h24d0160a;
ram_cell[      24] = 32'h0;  // 32'h5b3f0dd9;
ram_cell[      25] = 32'h0;  // 32'hef9ffd28;
ram_cell[      26] = 32'h0;  // 32'hfa600d55;
ram_cell[      27] = 32'h0;  // 32'h8c593f82;
ram_cell[      28] = 32'h0;  // 32'h1994d141;
ram_cell[      29] = 32'h0;  // 32'h8c92456c;
ram_cell[      30] = 32'h0;  // 32'h7be5d474;
ram_cell[      31] = 32'h0;  // 32'hd9181b06;
ram_cell[      32] = 32'h0;  // 32'hbf2a0b2e;
ram_cell[      33] = 32'h0;  // 32'hc0438cc5;
ram_cell[      34] = 32'h0;  // 32'hd9ee0b1b;
```

算法对命中率的影响

测试替换算法对于Cache性能的影响：（使用规模为16的矩阵乘法）

No	替换算法	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
1	LRU	4	3	3	6	1547	8704	0.1777
1	FIFO	4	3	3	6	1739	8704	0.1998
2	LRU	2	3	3	6	4672	8704	0.5368
2	FIFO	2	3	3	6	4864	8704	0.5588

可以基本得出结论：LRU算法略强于FIFO算法，不过相差并不大。

容量对命中率的影响

我们使用比较好的LRU算法来进行探究：

改变LINE_LEN来统计：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
1	2	1	3	8	4960	8704	0.5699
2	2	2	3	7	4768	8704	0.5478
3	2	3	3	6	4672	8704	0.5367
4	2	4	3	5	873	8704	0.1003

与快速排序类似：当容量逐渐增大时，命中率逐渐提升。

而且，当一行可以容纳16个字的时候，命中率有了显著的提升，从理论上分析，这可能是和矩阵乘法的性质有关：每一行都有16个字，Cache的一行可以容纳完整的矩阵一行，可以对乘法有显著的加速。

组相连度和组数、行长度对命中率的影响

我们将Cache容量设置为1+4+3，改变其余几个参数进行实验：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
1	1	1	7	4	1319	8704	0.1515
1	1	2	6	4	1071	8704	0.1230
1	1	3	5	4	995	8704	0.1143
1	1	4	4	4	1053	8704	0.1210
1	1	5	3	4	1525	8704	0.1752
2	2	1	6	5	1621	8704	0.1862
2	2	2	5	5	1313	8704	0.1509
2	2	3	4	5	1159	8704	0.1332
2	2	4	3	5	873	8704	0.1003
2	2	5	2	5	865	8704	0.0994
3	4	1	5	6	2308	8704	0.2652
3	4	2	4	6	1930	8704	0.2217
3	4	3	3	6	1739	8704	0.1998

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	MISS_TIMES	TOTAL_TIMES	MISS_RATE
3	4	4	2	6	1671	8704	0.1920
3	4	5	1	6	1639	8704	0.1883

可以看出，**2路+5位行地址+2位组地址**构成了MatMul程序的最佳Cache配置。

对于这样访问空间较小的程序，更多的组导致了更多的冲突，随着组数的增加，每一路的组数也不得不缩小，这很有可能导致一个频繁被使用的块因为地址在组相连情况下和另一个块冲突，而不停被换入换出，从而导致miss更多。

资源使用情况

在这里，我们比较一下LRU算法和FIFO算法在最佳配置下的资源使用情况（4+3+3+6）

- LRU

```
1. Slice Logic
--------------

+-------------------------+--------+-------+------------+-----------+--------+
|        Site Type        |  Used  | Fixed | Prohibited | Available |  Util% |
+-------------------------+--------+-------+------------+-----------+--------+
| Slice LUTs*             |  43559 |     0 |          0 |     63400 |  68.71 |
|   LUT as Logic          |  43559 |     0 |          0 |     63400 |  68.71 |
|   LUT as Memory         |      0 |     0 |          0 |     19000 |   0.00 |
| Slice Registers         | 138348 |     0 |          0 |    126800 | 109.11 |
|   Register as Flip Flop | 138348 |     0 |          0 |    126800 | 109.11 |
|   Register as Latch     |      0 |     0 |          0 |    126800 |   0.00 |
| F7 Muxes                |  17808 |     0 |          0 |     31700 |  56.18 |
| F8 Muxes                |   8736 |     0 |          0 |     15850 |  55.12 |
+-------------------------+--------+-------+------------+-----------+--------+
```

- FIFO

```
1. Slice Logic
--------------

+-------------------------+--------+-------+------------+-----------+--------+
|        Site Type        |  Used  | Fixed | Prohibited | Available |  Util% |
+-------------------------+--------+-------+------------+-----------+--------+
| Slice LUTs*             |  42852 |     0 |          0 |     63400 |  67.59 |
|   LUT as Logic          |  42852 |     0 |          0 |     63400 |  67.59 |
|   LUT as Memory         |      0 |     0 |          0 |     19000 |   0.00 |
| Slice Registers         | 137825 |     0 |          0 |    126800 | 108.69 |
|   Register as Flip Flop | 137825 |     0 |          0 |    126800 | 108.69 |
|   Register as Latch     |      0 |     0 |          0 |    126800 |   0.00 |
| F7 Muxes                |  18274 |     0 |          0 |     31700 |  57.65 |
```

```
| F8 Muxes                |   8970 |     0 |          0 |     15850 |  56.59 |
+-------------------------+--------+-------+------------+-----------+--------+
```

以上是完整CPU的资源使用，可以看到LRU算法的资源消耗量要比FIFO算法的资源消耗量大，这主要是由于LRU的LRU表的更新策略相对于FIFO的更新策略更加复杂,同时使用寄存器数量也多很多。

实验问题记录

- 写好cache后由于没有进行initial，导致无法正常记录time值，同时cache还可以正常运行，难以发现这个错误
- cache写时没有注意换入后way_addr改变了，所以需要一个寄存器寄存一下未改变的值

实验总结

- 本次实验让我理解了Cache和流水线的适配方法，以及对于流水线冲突的处理方法
- 通过若干数据的测试，我了解了Cache性能变化和各项参数的关系