CAIC Lab5 Report

1 设计思路

本Lab的设计是在Lab4的基础上，于MEM模块与Main Memory之间加入了一级Cache构成的。

cache的设计大致与tutorial中所给一致。

- 地址的翻译由`assign`对addr的部分进行截取并连续赋值；
- 判断是否命中使用generate以及组合逻辑来进行；miss信号作为是否需要流水线进行stall的标志，在有request且request没有完成时置为高值：

```
    wire hit;
    wire [WAY_CNT-1:0] hit_i;
    wire [WAY_CNT-1:0] way_ok[7:0];

    genvar way;
    generate
        for (way = 0; way < WAY_CNT; way = way + 1)begin:hitloop
            assign hit_i[way] = (valid[set_addr][way]&&(tag[set_addr][way] ==
tag_addr));
        end
    endgenerate
    assign hit = |hit_i;
    integer way_i;
    always@(*) begin
        // always@(hit_i) begin
        for(way_i = 0 ; way_i < WAY_CNT; way_i++)begin
            if(hit_i[way_i])
                hit_way = way_i;
        end
    end

    assign miss = ((read_request || write_request)&&(!request_finish));
```

- 状态机的更新与tutorial上给出的基本一致。尤其注意的是`request_finish`信号的状态。
  - `READY`状态下，若hit，有`read_request`可以直接读，有`write_request`时写进cache时注意置dirty；若没有hit，如果替换的行valid且dirty，下个周期则进入`REPLACE_OUT`状态将这个数据换入主存；否则直接在下个周期进入`REPLACE_IN`状态；
  - `REPLACE_OUT`状态下，`mem_write_request`保持为1，等主存`mem_request_finish`后进入换入状态；
  - `REPLACE_IN`状态下，`mem_read_request`保持为1,等主存`mem_request_finish`后将主存中的数据写入cache，并更新tag、valid和dirty，下个周期回到`READY`状态；
  - hit时若有`(write_request || read_request)`，则`request_finish`信号与原来相反。这既是为了让hit信号发出时`request_finish`信号置1，又是为了避免在连续两个周期`request_finish`为1时对连续两个hit命令的完成判断造成混淆；`REPLACE_OUT`与`REPLACE_IN`两个状态下，`request_finish`

置0；

- 对于LRU替换策略，在`READY`状态下进行如下的选择；`REPLACE_IN`状态下则不再需要进行其他的操作。注意更新age的时候的条件是`(read_request||write_request)&&!request_finish`，若不带上!request_finish的条件，cache的age将会是参考的两倍。

```
if ((read_request||write_request)&&!request_finish)begin//更新age
        for(i = 0; i < WAY_CNT; i++)
            if(i == hit_way)
                way_age[set_addr][i] <= 0;
        else if(way_age[set_addr][i] < `MAX_AGE)
            way_age[set_addr][i] <= way_age[set_addr][i]+1;//为了不越界
        else way_age[set_addr][i] <= way_age[set_addr][i];
    end
    age_max = 0;//寻找最大age
    for (i = 0; i <WAY_CNT; i++)begin:oldlop
        if(way_age[set_addr][i] > age_max)
            begin
                age_max = way_age[set_addr][i];
                age_max_way = i;
            end
    end
    replace_way[set_addr] <= age_max_way;
```

在`data_cache.v`文件之外，对`riscv.v`与`hazard_detect_unit`相应进行了修改。其中，`hazard_detect_unit`在`data_cache.v`的miss信号发出时会对所有五个阶段进行stall；stall与bubble信号同lab4中的逻辑判断一致。

```
    assign stall_if = stall||miss;
    assign stall_id = stall||miss;
    assign stall_ex = miss;
    assign stall_mem = miss;
    assign stall_wb = miss;
```

2 Debug过程

这个lab的debug过程的痛苦程度不亚于pipeline。主要是时序的问题。以下是一些遇到的问题与解决的方法：

- hit的判断最好还是用组合逻辑来进行，否则可能遇到不少的时序问题；
- miss时的stall是要对整个pipeline进行的，而不是只对if和id两个阶段进行的；
- 一定要注意`request_finish`不能连续两个周期为1；
- 在筛选最大age的时候，循环中使用的赋值应为阻塞(=)，否则会出现混乱的问题；
- `mem_read_addr`的长度是32而不是`MEM_ADDR_LEN`，这一点在本地的编译器中语法一直没报错但结果一直不对，这说明有的时候高级的编译器不一定是好的……
- debug的时候可以通过建立一些flag变量，去知晓哪个分支被运行了哪个没有……

最后完成了各个case的测试，提交序号44443e81d986。

答案正确

作者: zzq0219

Performance

score: 100

3 设计探索

对各个ADDR_LEN进行调整时，有：

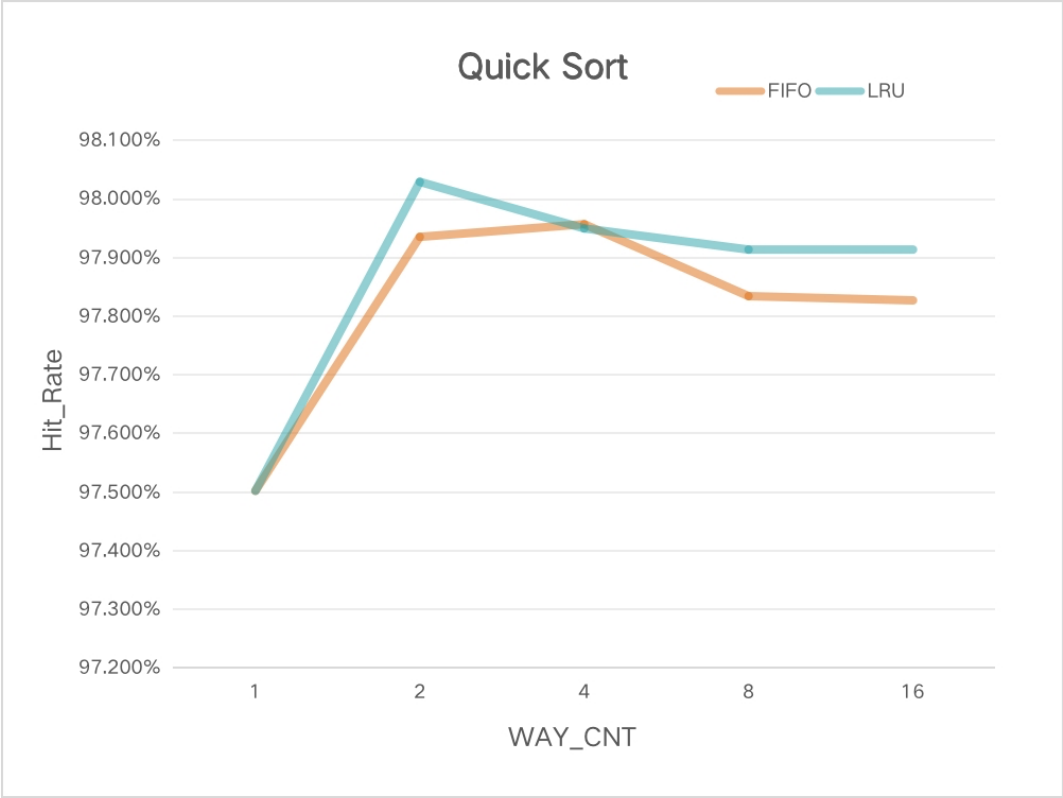
- `WORD_ADDR_LEN` 恒为2；
- `LINE_ADDR_LEN` 恒为3，因为在此lab中Cache_Line长度恒为64Byte；
- `SET_ADDR_LEN` 与 `WAY_CNT` 相互制约，满足$WAY_CNT \times 2^{SET_ADDR_LEN}(=4\times 2^3)=32$不变；
- 在改变 `SET_ADDR_LEN` 时，要满足 `SET_ADDR_LEN` 与 `TAG_ADDR_LEN` 之和，也就是 `main_memory_wrapper.v` 中的 `ADDR_LEN` 为13不变。这是因为Data RAM的存储容量为2^{18}字节，因此主存接受的地址为18位； `WORD_ADDR_LEN` 恒为2、 `LINE_ADDR_LEN` 恒为3，那么剩下的 `SET_ADDR_LEN` 与 `TAG_ADDR_LEN` 之和为13。如果设置不对，出现地址错误，会对运行结果产生致命影响。

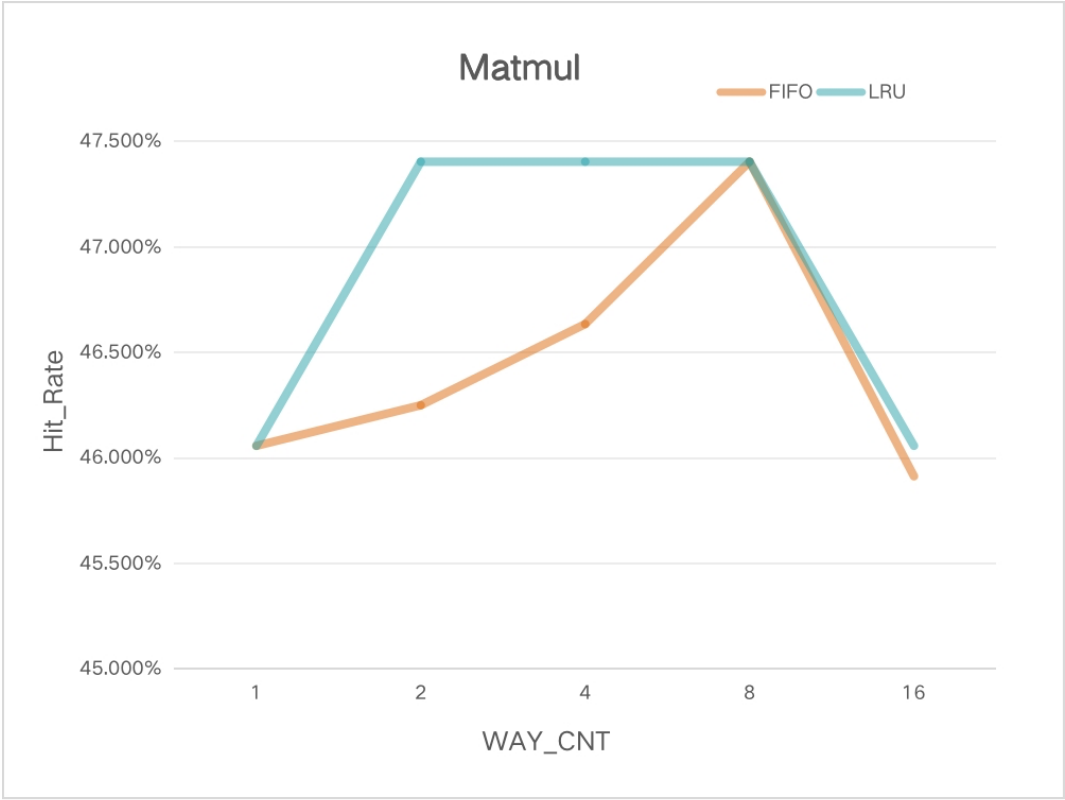
对Hit与Miss以及总的Total Count进行计数是如此进行的：

- 在未hit时如果有 `(write_request || read_request)`，那么 `miss_cnt` 加一；
- 在hit时，如果有 `(write_request || read_request)` 且 `request_finish`，那么 `total_cnt` 加一。不是对 `hit_cnt` 进行+1，这是因为在指令miss后，还是会进行替换后跳转到这一步，所以这一步进行计数的除了hit还有miss，故应对total+1；
- `hit_cnt` 在每个上升沿开始时由 `total_cnt` 减去 `miss_cnt` 得到。

No.	Replace_Policy	Test_Case	WAY_CNT	SET_ADDR_LEN	总次数	Hit次数	Miss次数	Hit_Rate
1	FIFO	Quicksort	1	5	13853	13507	346	97.502%
2	FIFO	Quicksort	2	4	13853	13567	286	97.935%
3	FIFO	Quicksort	4	3	13853	13570	283	97.957%
4	FIFO	Quicksort	8	2	13853	13553	300	97.834%
5	FIFO	Quicksort	16	1	13853	13552	301	97.827%
6	LRU	Quicksort	1	5	13853	13507	346	97.502%
7	LRU	Quicksort	2	4	13853	13580	273	98.029%
8	LRU	Quicksort	4	3	13853	13569	284	97.950%
9	LRU	Quicksort	8	2	13853	13564	289	97.914%
10	LRU	Quicksort	16	1	13853	13564	289	97.914%
11	FIFO	Matmul	1	5	66560	30656	35904	46.058%
12	FIFO	Matmul	2	4	66560	30784	35776	46.250%
13	FIFO	Matmul	4	3	66560	31040	35520	46.635%
14	FIFO	Matmul	8	2	66560	31552	35008	47.404%
15	FIFO	Matmul	16	1	66560	30560	36000	45.913%
16	LRU	Matmul	1	5	66560	30656	35904	46.058%
17	LRU	Matmul	2	4	66560	31552	35008	47.404%
18	LRU	Matmul	4	3	66560	31552	35008	47.404%
19	LRU	Matmul	8	2	66560	31552	35008	47.404%
20	LRU	Matmul	16	1	66560	30656	35904	46.058%

以下是两项不同的任务下，FIFO和LRU两种不同的替换策略对于不同的WAY_CNT下处理时的Hit Rate结果。





对于图表数据进行分析，可知：

- 对于同样的任务与替换策略，随着WAY_CNT的增加，HitRate先增加后减少。这是因为，WAY_CNT过小时缓存出现冲突进行替换的概率大，过大时则同周期内需要同时进行比较的标签数量多，而仅当WAY_CNT在适当区间时，对于空间与时间局部性的利用才是较为平衡、合理的；
- 对于同样的任务、同样的结构参数，选择LRU替换策略相较于FIFO而言能得到的Hit_Rate更高，这是因为LRU对于时间局部性的利用比FIFO更好；
- 对于不同的任务而言：QuickSort的HitRate在各种情况下都远高于Matmul，这可能是因为问题本身具有的空间与时间局部性造成的，在算法保持不变时，硬件能对其进行的优化是有限的。