中国科学技术大学计算机学院
《计算机体系结构》实验报告

2021.05.28

实验题目：Cache 的设计和实现

学生姓名：胡毅翔

学生学号：PB18000290

计算机实验教学中心制

2019 年 9 月

目录

1 实验目的

1. 实现组相联度可变的 Cache。
2. 实现 FIFO 和 LRU 替换策略。
3. 连接 Lab2 中实现的 CPU 并运行测试程序。
4. 统计程序运行过程中的 miss 和 hit。
5. 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。
6. 权衡选择合适的组相连度。
7. 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。
8. 理解写回法的优劣。

2 实验环境

1.PC 一台

2.Windows 10 操作系统

3.Vivado 2019.1

4.Visual Studio Code 1.56.2

3 Cache 设计和实现

3.1 Cache 结构

Cache 结构在课本《计算机体系结构：量化研究方法》(中文第五版) 的附录 B：存储器层次结构回顾中有所描述。本实验全部采用“写回 + 写入分派”的 cache 策略，这种策略在读或写命中时，直接从 cache 中读写数据，只需要一个时钟周期，不需要对 CPU 流水线进行 stall；在发生缺失时，读缺失和写缺失的处理方法是相同的，都是从主存中换入缺失的 line (line 即块) 到 cache 中 (当然，如果要换入的 line 已经被使用了，并且脏，则需要在换入之前进行换出)，再从 cache 中读写数据。总结下来，cache 应该维护如下的状态机：

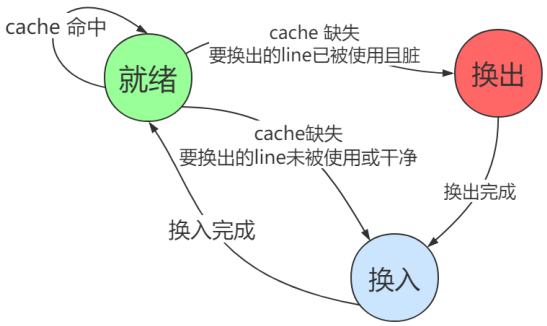

图 1: cache 状态机

当没有读/写请求时，cache 保持就绪状态，当 CPU 发出读/写请求时，cache 检查是否命中，如果命中则立刻响应读/写请求，并仍保持就绪状态。如果缺失，则进行换入（换入之前可能需要先换出），在 cache 进行换出换入时，cache 无法响应 CPU 当前的读写请求，因此需要向 CPU 发出 miss=1 的信号，CPU 需要使用该信号控制所有流水段进行 stall。直到 cache 完成换出换入后重回就绪状态，此时 cache 就能响应这个读写请求。

3.2 Cache 对外接口与时序

cache 的对外接口如下:

```
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 = 4 // 组相连度，决定了每组中有多少路line
)(
    input clk, rst,
    output miss,  // 对CPU发出的miss信号
    input [31: 0] addr,  // 读写请求地址
    input rd_req,  // 读请求信号
    output reg [31: 0] rd_data,  // 读出的数据，一次读一个word
    input wr_req,  // 写请求信号
    input [31: 0] wr_data // 要写入的数据，一次写一个word
);
```

当读/写命中时，时序与以往的 dataRam 完全一样，如图:

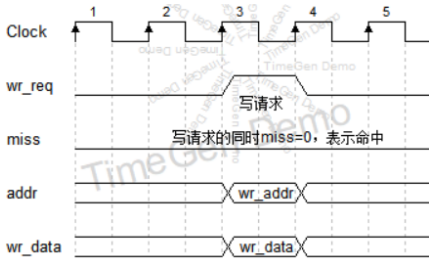

图 2: 写命中时序

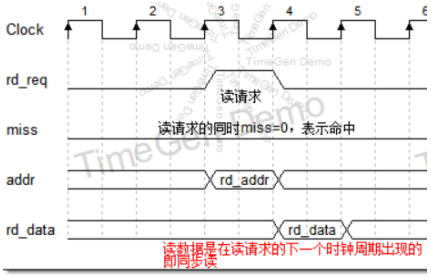

图 3: 读命中时序

当读/写缺失时，随着请求信号的出现，miss 信号同样变为 1，请求信号要一直保持 1，直到一个周期，miss 变为 0，请求信号仍为 1，就完成了一次读/写。另外，在请求信号保持 1 的过程中，addr 和 wr_data 也要保持。

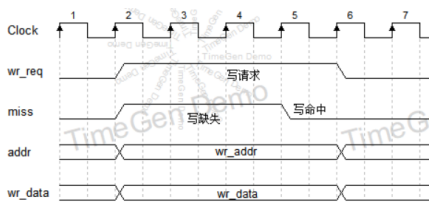

图 4: 写缺失时序

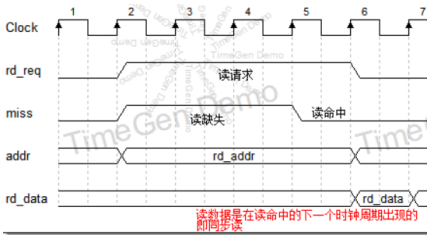

图 5: 读缺失时序

rd_req 与 miss，wr_req 与 miss，实际上构成了两对握手信号，这种握手信号时序广泛的应用于总线技术中。

在以上的时序图中，缺失只持续了 3 个时钟周期，这只是为了方便演示。在本实验中，由于主存需要 50 个周期进行一次读/写，所以 cache 缺失会持续 50 多个时钟周期或 100 多个时钟周期。当只进行换入时，缺失持续 50 多个时钟周期。当先换出后换入时，缺失持续 100 多个时钟周期。

3.3 主存对外接口与时序

主存被 cache 所调用，是使用 BRAM 模仿的 DDR。包括 main_mem.sv 与 mem.sv 两个文件，顶层文件是 main_mem.sv，它以 line 为读写单元（而不是以 word 为读写单元），且读写周期很长，本实验设置为 50 个时钟周期。

main_mem 的输入输出接口定义如下：

```
module main_mem #(          // 主存，每次读写以line为单位，并会延时固定的50
    个周期
  parameter  LINE_ADDR_LEN =  3,  // line内地址的长度，决定了每个line具有2
    ^3=8个word
  parameter  ADDR_LEN  = 8          // 主存一共有2^8=256个line
)(
  input  clk, rst,
  output gnt,                                   // 读写响应信号
  input  [ADDR_LEN-1:0] addr,                   // 读写地址
  input  rd_req,                                // 读请求信号
  output reg [31:0] rd_line [1<<LINE_ADDR_LEN], // 读出的line数据，这是一个
    二维数组，即8个word = 8*32bit
  input  wr_req,                                // 写请求信号
  input  [31:0] wr_line [1<<LINE_ADDR_LEN]      // 要写入的line数据，这是
    一个二维数组，是8个word = 8*32bit
);
```

main_mem 的读写时序与 cache 的读写缺失时序非常相似，也就是说，主存可以看作一个永远都会缺失，并且一缺失就缺失 50 个周期的 cache。不同的是 cache 的 miss 信号和 main_mem 的 gnt 信号的逻辑相反：cache 的 miss=0 时代表命中；而 main_mem 的 gnt=1 时代表命中。如图：

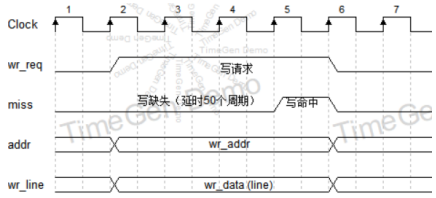

图 6: 主存写时序

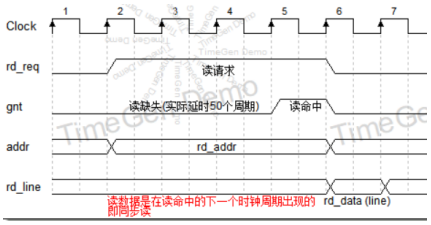

图 7: 主存读时序

3.4 直接映射 cache 的实现

理解 cache 首先要看 32bit addr 是如何分割的，如下图：

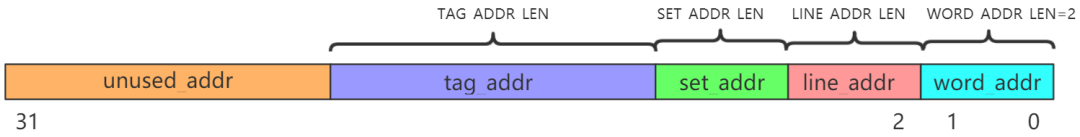

图 8: 32bit 地址的分割

word_addr[1:0]：字节地址，即指定字节是 word(字) 中的第几个。固定为 2bit。本次实验不要求处理独热码，因此 word_addr 不需要处理。

line_addr: line 内地址，其长度由参数 LINE_ADDR_LEN 决定。例如，如果希望每个 line 中有 16 个 word，则 LINE_ADDR_LEN 应设为 4，因为 $2^4 = 16$。在 cache 读写过程中，line_addr 用于指示要读写的 word 是 line 中的哪一个 word。

set_addr: line 地址，其长度由参数 SET_ADDR_LEN 决定。例如，如果希望 cache 中有 4 个 cache 组，则 SET_ADDR_LEN 应该设置为 2，因为 $2^2 = 4$。在 cache 读写过程中，set_addr 负责将读写请求路由到正确的组。

tag_addr: 是该 32 位地址的 TAG。当发生读写请求时，cache 应该把 32 位地址中的 tag_addr 取出，与 cache 中的 TAG 比较，如果相等则命中。如果不等则缺失。

unused_addr: 32 位地址中的高位，直接丢弃。

在我们提供的代码中，使用一句 assign 完成 32bit 地址的分割：

```
assign {unused_addr, tag_addr, set_addr, line_addr, word_addr} = addr;
```

在简单 cache 中，line size 可以通过调节 LINE_ADDR_LEN 去改变，组数可以通过调节 SET_ADDR_LEN 去改变。这里，以 LINE_ADDR_LEN=3，SET_ADDR_LEN=2, TAG_ADDR=12 为例，给出直接相连 cache 的结构图：

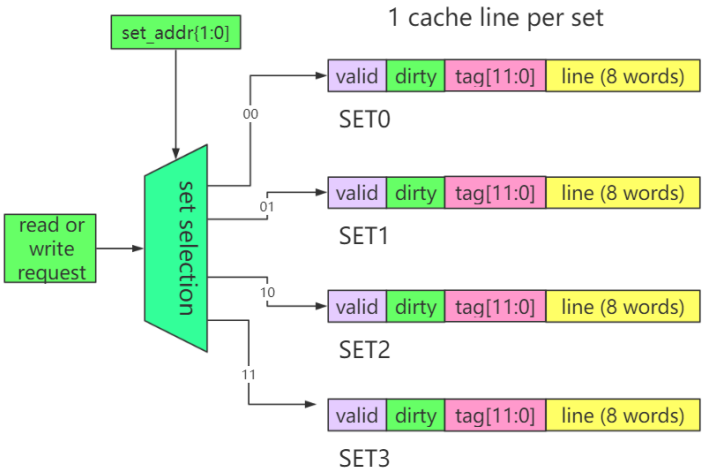

图 9: 直接相连 cache 的结构

实际上，直接相连是组相连的特殊情况，相当于 1 路组相连，因此每个 set 中只有 1 个 line。每个 line 是 8 个 word，除此之外，每个 line 还需要 1 个 TAG，一个 dirty（脏位），一个 valid（有效位）。这些在 system verilog 代码里如下：

```
reg [           31:0] cache_mem    [SET_SIZE][LINE_SIZE]; // SET_SIZE个line，每个line有LINE_SIZE个word
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE];            // SET_SIZE个TAG
reg                    valid        [SET_SIZE];            // SET_SIZE个valid(有效位)
reg                    dirty        [SET_SIZE];            // SET_SIZE个dirty(脏位)
```

图 10: 直接相连 cache 中的一些变量

当有读写请求时，根据地址中的 set_addr 字段，决定要到哪个 line 中读写数据。然后，查看该 line 是否 valid，如果 valid=0 则一定是缺失，如果 valid=1，说明这个 line 是有效的，需要比较这个 line 的 tag 和地址中的 tag 是否相同，相同则命中，不同则缺失。如果命中，则立即响应读写请求。当然，如果是写请求，要把 dirty 置 1。

如果 cache 缺失，要从主存中换入该块到这个 cache line 中。在换入前，也需要考虑是否需要先换出。如果 valid=1 且 dirty=1，说明该 cache 块是有效的并且已经被修改过，则需要先进行换出。此时需要控制 cache 状态机的状态转移。cache 状态机的状态如下：

```
enum {IDLE, SWAP_OUT, SWAP_IN, SWAP_IN_OK} cache_stat = IDLE;
```

相比图 1，这里多出一个状态 SWAP_IN_OK，该状态一定出现在 SWAP_IN 状态之后，只占用一个时钟周期，负责把主存中读出的数据写入 cache line。

3.5 组相连 cache 的实现

下图是组相连 cache 的结构图。取 LINE_ADDR_LEN=3，SET_ADDR_LEN=2，TAG_ADDR=12，WAY_CNT=4 为例。

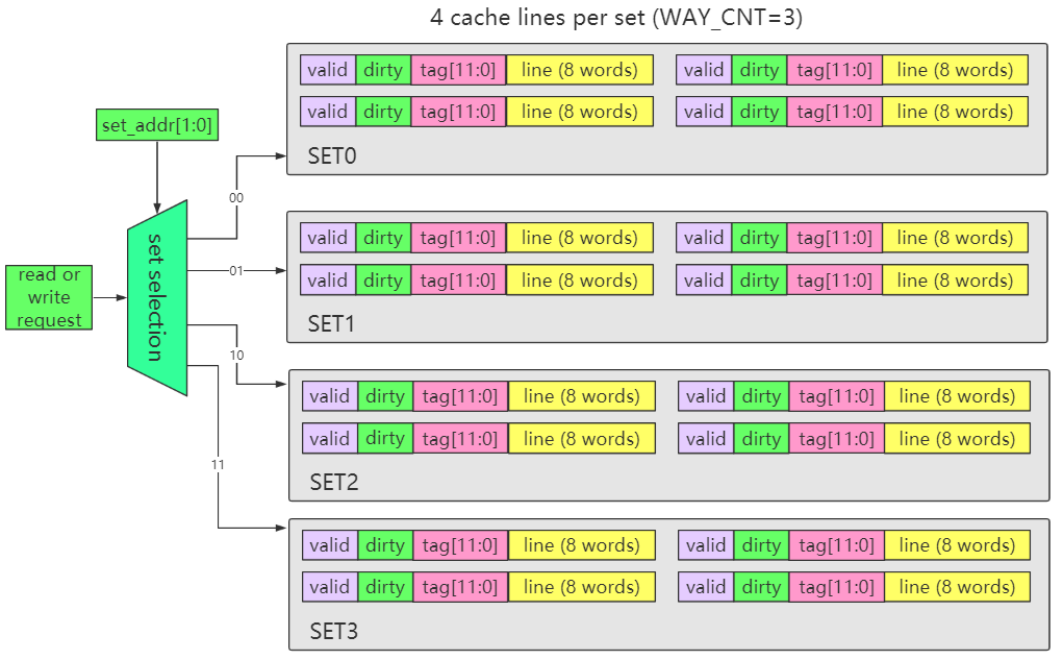

图 11: 4 路组相连 cache

相比直接相连 cache，组相连 cache 需要加入的有：

1. 将图 10 中所示的数组添加一个维度，该维度的大小为 WAY_CNT。
2. 实现并行命中判断：为了判断是否命中，直接相连 cache 每次只需要判断一个 valid，一个 dirty，一个 TAG 是否命中，但组相连 cache 则需要在组内并行的判断每路 line 是否命中。
3. 实现替换策略：当 cache 需要换出时，直接相连 cache 没有选择，因为每个组中只有 1 个 line，只能换出换入这唯一的 line。但组相连 cache 需要决策换出哪个 line。本实验要求实现 FIFO 换出策略与 LRU 换出策略（请见《计算机体系结构：量化研究方法》(中文第五版）附录 B)。为了实现 FIFO 策略和 LRU 策略，还需要加入一些辅助的 wire 和 reg 变量。

4 替换策略的实现

4.1 FIFO

FIFO（First In First Out）算法，即先入先出，替换规则为将最早存入的块换出。在代码中，增加如下数据结构：

```
reg [WAY_CNT: 0] FIFO_record[SET_SIZE][WAY_CNT]; //每一组内的FIFO排位记录
```

每次存入新块时，更新 FIFO_record, 替换时的算法为：

```
begin
    for (integer i = 0;i < WAY_CNT ;i = i + 1)
    begin //此时说明是最早进去的
        if (FIFO_record[set_addr][i] == WAY_CNT)
        begin
            out_way = i;
            FIFO_record[set_addr][i] = 0;
            break;
        end
    end
end
```

4.2 LRU

LRU（Least Recent Use）算法，即最近最少使用, 替换规则为将最近使用时间最早的块换出。在代码中，增加如下数据结构：

```
integer time_cnt; //LRU时间记录
reg [15: 0] LRU_record[SET_SIZE][WAY_CNT]; //每一项的LRU记录
```

每个时钟周期更新 time_cnt，使用块时，更新 LRU_record。选择换出的块时使用以下算法：

```
if (swap_out_strategy == LRU)
begin
    for (integer i = 0; i < WAY_CNT; i = i + 1 )
    begin
        out_way = 0;
        if (LRU_record[set_addr][i] < LRU_record[set_addr][out_way])
        begin
            out_way = i;
        end
    end
end
```

5 Cache 资源消耗评估

在 cache.sv 中修改 Cache 的参数（组数、组相连度、line 大小等），进行综合得到资源占用报告。其中 LUT、FF 是我们最在意的资源，因为 cache 的逻辑均使用 LUT 和 FF 实现。这两个参数的使用量就

代表了 cache 所占用电路的资源量。多次修改参数得到下表：

组数	组相联度	**line size**	**LUT**	**FF**
8	4	8	1128	3056
8	2	8	1131	3057
8	1	8	1141	3057
4	4	8	1997	3210
16	4	8	4075	9912
8	4	4	1532	3030
8	4	16	4138	10295

表 1: 不同 Cache 的资源占用情况

根据表中数据，我们可以得出以下结论：

1. 修改组相联度对 cache 所占资源数变化不大。
2. 增加组数或增大 line size 都会增加资源消耗。

6 Cache 与 CPU 组合测试

设置组数为 8，组相联度为 4，每个 line 内有 8 个 word，分别测试在规模为 256 的快速排序，规模为 16×16 矩阵乘法上对 FIFO，LRU 两种替换策略进行测试。

替换策略	测试样例	组相联度	**miss**	**hit**	命中率
FIFO	矩阵乘法	4	4800	3904	44.85%
LRU	矩阵乘法	4	4696	4008	46.04%
LRU	矩阵乘法	8	4696	4008	46.04%
FIFO	快速排序	4	142	5322	97.40%
LRU	快速排序	4	247	5217	95.48%
LRU	快速排序	1	247	5217	95.48%
LRU	快速排序	2	247	5217	95.48%
LRU	快速排序	8	247	5217	95.48%

表 2: 不同替换策略的缓存命中情况

根据表中数据，我们可以得出以下结论：

1. LRU 和 FIFO 在测试程序中的表现差异不大。
2. FIFO 算法虽然更为简单，但在快速排序中有着更好的效果。

7 总结

本次实验完成了组相联 Cache 的设计，实现了 FIFO 和 LRU 两种替换策略，并与 CPU 连接，运行测试程序。通过数据实验加深了对课程内容的理解。