实验四 流水线 MIPS CPU Cache

张海斌*

2019 年 6 月

目 录

1 概要

实验四添加 Cache 模块充当 MIPS CPU 和内存之间数据通信的中介，目的是了解 Cache 的基本原理和构造，以及程序如何对时间、空间的局部性进行优化。Cache 采用的是四路组相联高速缓存，Cache 包括四个组，每个组包含四个块，每个块包含四个字的数据。块替换策略使用 LRU 算法。由于同时有指令内存和数据内存，所以我总共使用了两个完全相同的独立的 Cache 作为 CPU 与内存数据交流的中介。对于 CPU 的控制，当缓存没有命中的时候，CPU 会处于暂停的状态，这个暂停状态我通过控制 CPU 的时钟延迟上升沿的到来来控制。

2 功能展示

与之前的流水线处理器相比，七段数码管的显示和开关的控制几乎是完全相同的，提供了当前 Fetch 阶段 PC 和下一个周期 PC 的显示和当前正在读取的机器码的显示，以及通用寄存器和数据内存数据的显示控制。只有显示内存数据需要的内存地址位数增加了一位。而

* 学号 17307130118

LED 则显示了一些更多的内容：LED[15] 同样是显示处理器的时钟信号，LED[14] 显示的是两个 Cache 是否都已就绪，LED[13] 表示 CPU 实际接收的时钟信号（Cache 未就绪时时钟上升沿信号会被屏蔽掉），LED[12] 和 LED[11] 分别表示指令内存和数据内存对应的缓存是否命中，LED[10:7] 和 LED[6:3] 分别显示指令内存和数据内存对应的缓存控制器中有限状态机的状态值。

3 代码结构

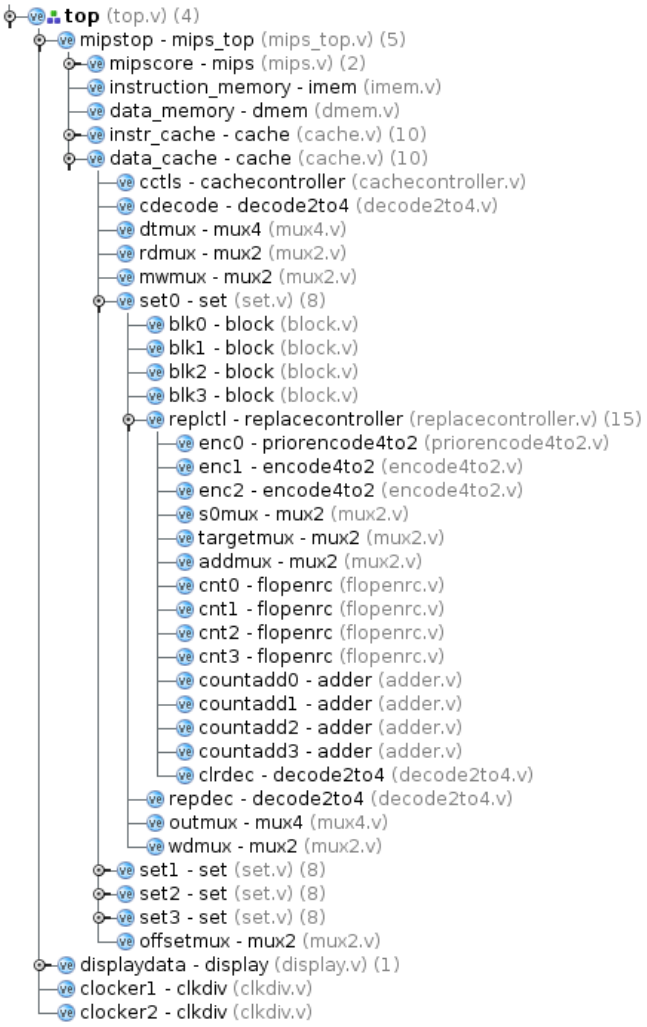

Figure 1: Cache 模块层次结构图

Figure 1 展现了 MIPS 的结构和 `Cache` 模块中的各个具体模块结构。总体上来说，MIPS 的结构没有太大的变化，只在 `mips_top` 模块内增加了两个 `cache` 模块。 `cache` 模块的组成主要有两个部份：Cache 控制器 `cachecontroller` 模块和四个完全相同的组 `set` 模块。在每个组 `set` 模块下，又包含了块 `block` 模块和块替换策略控制模块 `replacecontroller`。

4 模块设计

4.1 `mem` 模块

```
    assign nextstate = state + 1;

    always @(posedge CLK)
        if (Reset)
            state <= 2'b00;
        else
            state <= nextstate;

    always @(*)
        if (state == 2'b11)
        begin
            Ready <= 1'b1;
            RD <= RAM[A[7:2]];
        end
        else
        begin
            Ready <= 1'b0;
            RD <= 32'b0;
        end
```

Listing 1: `mem` 模块实现

为了模拟真实情况下内存慢于 CPU 和缓存的情况，我修改了同是内存的 `imem` 模块和 `dmem` 模块，为它们的端口添加了 `Ready` 输出信号表示内存数据读写是否完成（代码见 Listing 1 ）。读取内存时，内存输出数据只有在 `Ready` 信号为 1 的时候是有效的。向内存写入数据时，也只有在 `Ready` 信号为 1 的时候会写入。

在内存内部实现上，我使用了一个有限状态机（计数器）来控制内存的延时每四个周期读写一次。当状态机状态为 11 时，`Ready` 信号为 1。

4.2 `block` 缓存块

`block` 模块的功能从接口上（见 Listing 2 ）就基本可以看出来了。因为一个块中有四个字的数据，所以需要一个两位的偏移量 `Offset` 来表示需要操作的数据是哪一个字。同时，还有写入控制信号 `WE` 控制是否写入所有 `Set*` 数据，1 为写入，0 为读取数据。`Valid` `Dirty` `Tag` `RD` 分别表示块的有效位（是否已经被使用)、是否已经写入的脏位、Tag 标签和读取的数据。而加了前缀 `Set` 的表示对应数据的写入信号值。

4.3 `replacecontroller` 组替换策略模块

组替换策略我实现了完整的 LRU 算法。该算法通过用计数器记录最近时间内各个块的使用情况决定替换的对象。每个块都对应有一个两位的计数器，所以共有四个计数器。这在 `replacecontroller` 模块中对应于两位数据的数组 `count[3:0]`。当每次对组的请求访问开始的时候（ `Inti` 信号为 1 ）对计数器进行更新。更新共分为三种情况：

1. 当请求的 Tag 与四个块中的某一个块的 Tag 相同，即 Hit 时，将 Hit 的块的计数值清零，同时比 Hit 的块计数值小的块对应计数值全部加 1。

```
module block(
    input CLK,
    input Reset,
    input [1:0] Offset,
    input WE, SetValid, SetDirty,
    input [25:0] SetTag,
    input [31:0] WD,
    output reg Valid,
    output Dirty,
    output reg [25:0] Tag,
    output [31:0] RD);
```

Listing 2: `block` 模块接口

2. 当请求未命中 Miss 且四个块不是所有都被使用时，从剩余未使用的块中取出一个使用（实现中使用优先级编码器进行选择），将它的计数器置零，并将所有有效的块的计数器全部加 1。

3. 当请求未命中且所有块都被使用时，选择计数值为 3 的块替换，并且将它的计数器清零。这与加 1 溢出变为 0 效果相同，而其它的计数器也全部要加 1，所以实现时我选择将所有的计数器都加 1。

`replacecontroller` 模块的输出是选中的块编号 `S` 和是否命中的信号 `Hit`。输入是所有组的有效位 `Valid`，和各组 Tag 是否与请求的 Tag 相同的信号 `Eq` 以及 Cache 是否处于初始状态的 `Init` 信号。

4.4 `set` 组模块

`set` 模块中，有四个 `block` 模块表示组中的四路，即四个块，还有一个替换策略模块用来从四个块中选出使用的块。同时，`set` 模块通过译码器将块控制信号传输到 `replacecontroller` 模块所选出的块上。对于块的写入数据是从内存读取的数据还是 CPU 写给 Cache 的数据也需要一个多路选择器进行选择。最后，`set` 模块需要向上一层 `cache` 模块输出所选择的块的相关数据，以方便 Cache 控制器根据相关信息判断之后的操作。相关代码见 Listing 3。

4.5 `cache` 高速缓存模块

`cache` 模块将外部 CPU 的数据传输到访问地址所对应的组上。`cache` 模块中除四个组外，还包括了一个控制器模块 `cachecontroller`。控制器模块内有一个有限状态机，并且根据有限状态机的状态和选中的块的状态决定了各种控制信号。为了统一 MIPS 中数据和指令内存对应的两个 Cache 的工作进程，我添加了一个 `Suspense` 控制信号，用来控制 Cache 在 Hit 后是否立即读取或写入数据。不过，最后我发现，其实并不需要这个控制信号，它的有无不会影响到替换策略选择块。

```
    block blk0(CLK, Reset, offset, we[0], setValid, setDirty, setTag, wd,
               valid[0], dirty[0], tag[0], rd[0]);
    block blk1(CLK, Reset, offset, we[1], setValid, setDirty, setTag, wd,
               valid[1], dirty[1], tag[1], rd[1]);
    block blk2(CLK, Reset, offset, we[2], setValid, setDirty, setTag, wd,
               valid[2], dirty[2], tag[2], rd[2]);
    block blk3(CLK, Reset, offset, we[3], setValid, setDirty, setTag, wd,
               valid[3], dirty[3], tag[3], rd[3]);

    // LRU 替换算法控制
    replacecontroller replctl(CLK, Reset, init, valid, eq, s, Hit);

    // 写使能端信号译码器
    decode2to4 #(1) repdec(wen, s, we[0], we[1], we[2], we[3]);
    // 输出信号选择
    mux4 #(59) outmux({dirty[0], rd[0], tag[0]},
                      {dirty[1], rd[1], tag[1]},
                      {dirty[2], rd[2], tag[2]},
                      {dirty[3], rd[3], tag[3]},
                      s, {Dirty, ReadData, OutTag});
    // 写入块的数据选择
    mux2 #(32) wdmux(MemReadData, WD, offsetSW, wd);
```

Listing 3: `set` 模块部份关键代码

4.6 `cachecontroller` 缓存控制模块

Figure 2是 Cache 控制器的有限状态机的状态转移图。其中，`mem` 表示内存 `Ready` 信号。状态 0 到状态 1 和 5 的线上的 `wb` 表示 `hit` 信号为 0 且 `en` 信号为 1 的时候 Cache 是否需要写回，也就是命中的块对应的 `Dirty` 信号。`hit` 信号和 `Dirty` 信号分别表示 Cache 是否命中以及命中的块是否需要写回内存。图中的 `!en` 表示 `en` 信号为 0，即当前时钟周期内不使用 Cache。状态 9 到状态 0 的 `susp` 表示控制 Cache 的 `Suspense` 信号，在代码中可能有些不同，因为后来我发现代码中有些控制是多余的。

4.7 `mips_top` 模块

```
    assign ready = (~iEn | iHit) & (~dEn | dHit);
    assign cpuclk = cpumask & CLK;
    assign cpumask = ready & state;
    assign suspense = ~ready;
```

Listing 4: `mips_top` 模块信号控制

相比流水线 CPU，这个加了 Cache 的版本的 `mips_top` 模块有一些变化。我加了一个

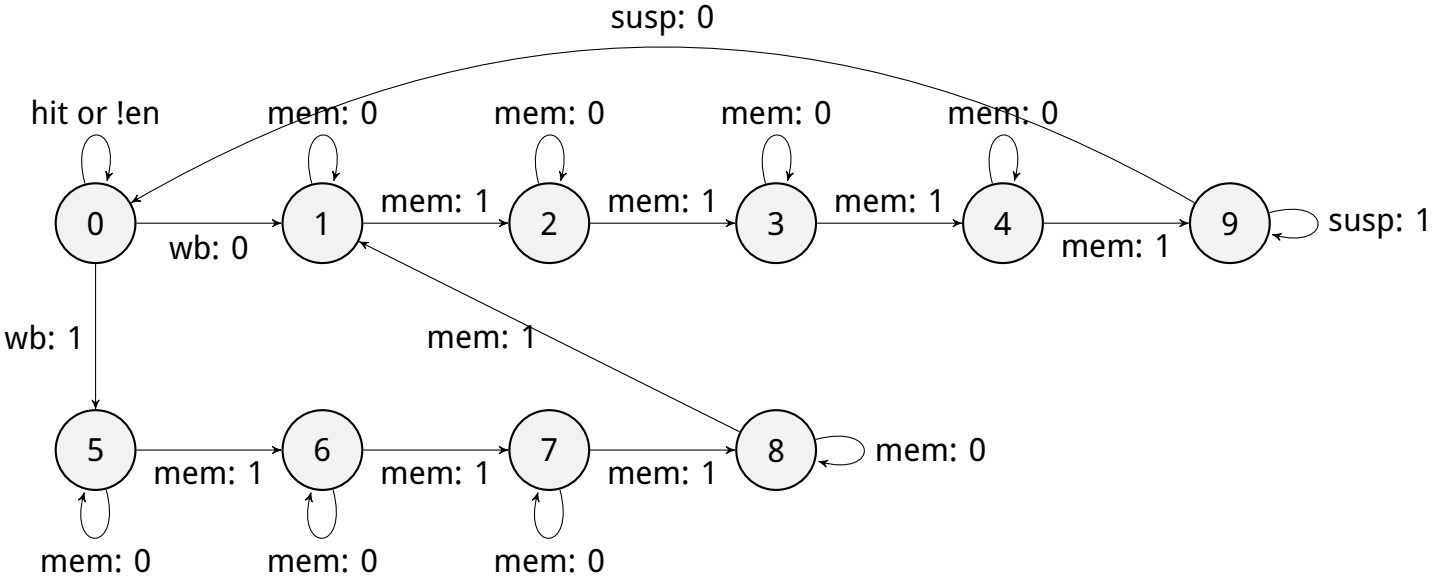

Figure 2: Cache 控制器有限状态机的状态转移图

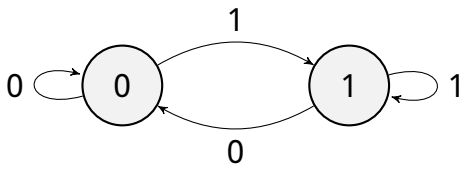

Figure 3: `mips_top` 模块中有限状态机的状态转移图

有限状态机来控制是否屏蔽 MIPS CPU 的时钟信号从而控制 CPU 的暂停与运行。其状态转移图见 Figure 3。其中有向边上的数字表示 `ready` 信号。而 `ready` 信号由指令内存和数据内存的 Cache 的 `Hit` 信号和 `En` 信号决定，它表示两个缓存是否已经同时就绪了。相关信号控制的代码见 Listing 4。 `cpuclk` 表示 MIPS CPU 的时钟信号， `cpumask` 表示时钟屏蔽信号。 `suspense` 是传输到两个 Cache 的 `Suspense` 信号。

5 仿真结果

添加了 Cache 的 MIPS 我先后在两个程序上仿真通过了，一个是以前的书上测试代码，另一个是我自己写的一个矩阵“乘法”。由于没有实现乘法指令，所以我用或运算和与运算分别代替矩阵乘法中的加法运算和乘法运算。在程序的开头有一个循环初始化矩阵数据，中间就是矩阵“乘法”，最后有一个循环大量读取内存从而让计算得到的矩阵数据从缓存中写回到内存中。由于没有乘法，矩阵乘法中对矩阵数据对应内存地址的计算就无法直接用乘法和加法计算得到。我通过增加寄存器临时变量的使用，实现了只用加法且不需要额外进行循环的情况下计算出数据内存地址。

从程序 1 的仿真结果可以看到每隔一段时间，指令内存的缓存就会从指令内存中读取四个字的指令，在最后对数据内存的写入使数据内存的缓存从数据内存读取数据。

程序 2 的汇编代码和机器码见 `code2.s` 文件。程序 2 最初可以在仿真跑完，而在板子上跑不完，我找到原因是在 `dmem` 模块中，Reset 内存清零和数据更新发生了 multi-driven，所以上板子上会跑出问题。在删去了 Reset 从而解决了 multi-driven 后，上板子就可以正常跑了。由于循环时间太长了，我没有看着它在板子上全部跑完。在仿真中的波形图也是非常长，就不放图了。在仿真图中，可以看到 PC 在某些区域变化非常频繁，也就是缓存基本上没有

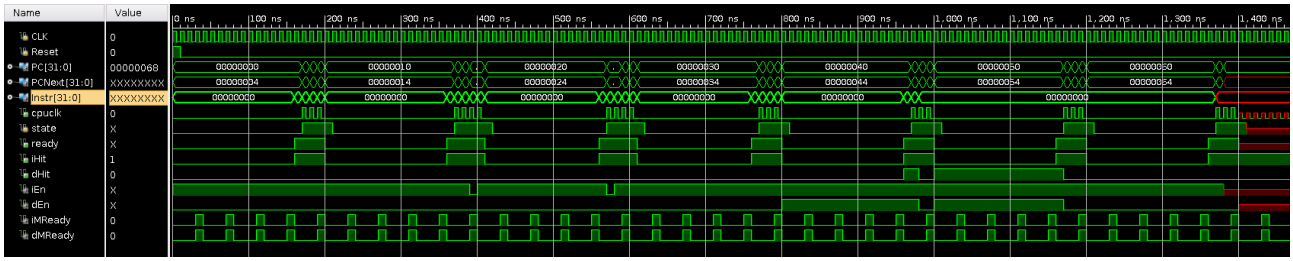

Figure 4: 程序 1 仿真结果

发生 Miss 的情况，在另一些区域 PC 不变，就说明此时缓存正在与内存进行数据的交互。在波形图上，中间一部分的循环在大部份时间下 PC 都在变化，所以可以看出 Cache 对程序运行速度的提升有很大的作用。