计算机组成与原理实验报告

计科 40 黄志翱 2014011345

May 27, 2016

1 实验要求

1.1 一

在 Y86 流水线处理器中增加 IADDL (iaddl 立即数，目标寄存器) 与 LEAVE 指令。

1.2 二

设计实现两个共享内存的 Y86 流水线处理器 (各带私有的 L1 Cache) 间的数据通信。

2 实验一

2.1 题意及解法分析

2.1.1 iaddl

iaddl 的等价代码为。

2.1.2 leave

leave 的等价代码为，基本操作与 pop 类似，故照抄。

3 实验二

3.1 题目分析

显然该问题是由两个完全不同的问题拼凑而成的。

- 实现具有 write-allocation-write-back 的两个 l1 cache，并实现两者通讯以达成缓存一致性。
- 基于共享内存实现 Y86 流水线之前的通讯。

我们只需要分别解决这两个问题即可。

3.2 l1 cache

首先详细地定义我们要实现的问题：在一个 unix-like system 下，我们需要实现两个进程。每个进程，需要支持从内存中读取与向内存中写入。

缓存是一组键值对应的表，键指示内存中的位置，值则对应于存储在内存上的内容。

由于需要维护 cache，所以向内存读取的方式被限定为：查找被访问的地址是否在缓存中，若在，则读取，否则从内存将相应的内容加载到缓存，读取。

由于要求 write-allocation-write-back，所以向内存写入的方式被限定为：查找被访问的地址是否在缓存中，若在，则直接在缓存中写入，否则从内存将相应的内容加载到缓存，写入。另外，当缓存满时，需要将某个缓存中的元素写入内存。

对于多进程的情况下，我们需要加入如下两个约束：

- 对于内存中同一位置的写入不能同时进行。
- 从缓存中读取时所得到的值必须是正确的。

为了优雅，我们还要求不能直接锁死内存，以及尽可能少减少向内存的写入和读取操作。为了实现这个目的，我采取了经典的 MESI 协议。

3.2.1 MESI

在 MESI 中，我们要求如果有某个元素存在于两个缓存上，则这两个缓存上的内容必须一致。
具体而言，对于一个缓存的一个键值，存在四种状态：

- M，该位置有效，键被该进程独占，值与内存中不同。
- E，该位置有效，键被该进程独占，值与内存中相同。
- S，两个进程都存有此值，且两个缓存，内存中的值都是一致的。
- I，此位置的缓存无效。

假定两个进程之间能够通讯，现在进程 A 想做一点事情，那么我们可以很容易地凭借这些状态对内存进行操作：

- 若当前位置是 M，则 A 可以直接操作而当做 B 不存在。
- 若当前位置是 E，则 A 可以直接操作而当做 B 不存在。
- 若当前位置是 S，如果 A 是 Read 操作，可以直接读取而当 B 不存在，如果 A 要 Write，则 A 需要通知 B 放弃该值，将状态改为 I。
- 若当前位置是 I，A 需要通知 B，若该值在 B 存在，则 B 需要将值写入内存，A 将其加载进入缓存。如果 A 是读取，则 A，B 状态都是 S，如果 A 是写入，则 A 状态变为 M，B 状态变为 I。

3.2.2 cache 通讯

A, B 之间通讯依赖于 bus 的存在。基本方法是：当 A 想要对 B 进行任何信息交流时，A 发出信号。A 和 B 在闲暇时刻随时监听对方的信号，一旦 B 空闲下来了，就发出信号通知 A 表示应答。此时 A 即可将所要发出的信息提供给 B，A 继续等待 B 接收信息处理。这样通讯直到最后 A 发出结束信号，继续各干各活。
如果 A 和 B 同时发出请求，则让编号大的回答标号较小的防止死锁。

3.2.3 实现细节

所有相关代码在 misc/cache/中。
server.c 提供两个 cache 共享的内存 mem 和总线 bus。内存通过 shmat 相关函数获得。内存 id 由 ftok 函数指定，id 信息被存放在 shareinfo.h 以供所有文件访问。
cache.c 则实现了一个 cache。首先根据 shareinfo.h 申请访问 mem 和 bus。然后利用 id 得知自己是 A 还是 B (从硬件角度上来讲，两个 cache 本身一定是有区分的，所以这个 id 在程序启动时就指定了)，开一个共享内存 msg_{id}，监听 msg_{id} 中是否有读写请求。当收到一个请求后，就利用 MESI 协议进行操作。监听 msg 和监听 bus 交替进行。无论何时收到请求则立刻回答。

3.3 对 psim 的修改

3.3.1 同时运行两个处理器

假定两个 cache 进程已经运行，则 cpu 往内存进行的读写操作都通过 msg 发出请求进行即可。从这个角度看，只要 cpu 通过编号确定与哪个 cache 连接即可，剩下的操作与单核 cpu 完全一致。
所以第一步修改就是将 isa.c 中所有涉及到内存读写的 $get_word_val, get_byte_val, set_word_val, set_byte_val, load_mem$ 中所有对 content 的读写替换成通过 msg 向 cache 发出读写请求。
值得注意的是，此时如果用两个模拟器同时运行程序的话，$load_mem$ 会将两个代码加载在同一个地方，所以我还修改了相应的函数和读取代码阶段 valp 的值将代码放置在不同的内存地址。

3.3.2 原子操作 test

为了使得运行相同代码时代码能够得知自己的身份，我加入了原子操作 test:

$$test\ addr, reg$$

效果为读取 addr 的值，将值赋给 reg，同时将 addr 处的值加 1。

改代码格式与 mrmovl 基本类似，所以在修改 pip-full.hcl 时只要在所有有 mrmovl 的地方添加上该操作即可。

但是为了使得这个操作在最低层面是原子的，我们需要让 cache 也支持这个操作——这是非常容易的，我们可以将对内存地址的 test 看成 read 和 write 的组合，为了保持这两个操作是连续的，只需要在 A,B 通讯的时候让 B 再等一下即可。

为了在 psim 中使得该操作是连续的，理论上只需要将 mrmovl 读取内存的时候，改为 test 即可。但是 psim.c 的实现非常奇怪，操作发生在 update 函数中，故而我们需要记录下要写入的寄存器的值，在 update 时进行 test 并对寄存器赋值。

为了视觉效果，我在 test 中加入了往屏幕输出 test 结果的功能。

3.4 y86 code

代码流程如下：根据对指定内存的 test 决定是 A 还是 B，跳到指定的代码段。

A 会进行多次循环，每次循环是往连续的内存段中写入 1，1 的个数从 1024 到 1024-100。写完之后会通过一个内存 byte 通知 B 已经写完。等待 B 回信，继续循环。

B 会等待 A 发出询问，当 A 询问完毕，会从特定内存位置开始遍历，数 1 的个数，直到内存中的值为 0 结束。代码如下：

```
.pos 0
irmovl 0, %ecx
test 0x00000f08(%ecx), %eax
addl %eax, %eax
je A
jmp B

A:
irmovl 0, %ecx
rmmovl %ecx, 0x00001001(%ecx)
test 0x00001005(%ecx), %ebx
irmovl 100, %ebx
ALOOP:
    irmovl 0, %ecx
    rmmovl %ebx, 0x00000f00(%ecx)
    AWAIT:
        mrmovl 0x00001005(%ecx), %ebx
        addl %ebx, %ebx
    je AWAIT
    test 0x00001010(%ecx), %edx

    irmovl 1024, %edx
    irmovl 0, %ecx
    irmovl 0x00002000, %eax

    INITA:
        rmmovl %ecx, (%eax)
        irmovl 4, %ebx
        addl %ebx, %eax
        irmovl 1, %ebx
```

```
        subl %ebx, %edx
    jne INITA

    mrmovl 0x00000f00(%ecx), %ebx
    rrmovl %ebx, %edx
    irmovl 924, %ebx
    addl %ebx, %edx
    irmovl 0x00002000, %eax
    irmovl 1, %ebx
    irmovl 4, %ecx

    ALOOP3:
        rmmovl %ebx, (%eax)
        addl %ecx, %eax
        subl %ebx, %edx
    jne ALOOP3

    rmmovl %edx, 0x00001005(%edx)
    rmmovl %edx, 0x00001015(%edx)
    irmovl 1, %ecx
    rmmovl %ecx, 0x00001001(%edx)

    irmovl 0, %ecx
    mrmovl 0x00000f00(%ecx), %ebx
    irmovl 1, %ecx
    subl %ecx, %ebx
jne ALOOP
irmovl 0, %ecx
irmovl 2, %ebx
rmmovl %ebx, 0x00001015(%ecx)
halt

B:
BLOOP:
    irmovl 0, %ecx
    BWAIT:
        mrmovl 0x00001001(%ecx), %ebx
        addl %ebx, %ebx
    je BWAIT

    mrmovl 0x00001015(%ecx), %edx
    irmovl -2, %ecx
    addl %ecx, %edx
    je HALT

    irmovl 0, %ecx
    rmmovl %ecx, 0x00001001(%ecx)

    irmovl 4, %ebx
    irmovl 0, %edx
    irmovl 0x00002000, %eax
    BLOOP2:
        addl %edx, %ecx
        mrmovl (%eax), %edx
```

```
        addl %ebx, %eax
        addl %ebx, %edx
        subl %ebx, %edx
    jne BLOOP2

    irmovl 0, %edx
    rmmovl %ecx, 0x00001010(%edx)
    irmovl 1, %ecx
    rmmovl %ecx, 0x00001005(%edx)
jmp BLOOP

HALT:
    halt
```

3.5 代码的运行

进入 misc/cache/ make 之后，再在 pipe/进行 make 即可运行。
在终端键入 $python3run_two_sim.py$ 即可运行。
程序会依次从 1024 输出到 926。