计算机组成与原理实验报告

计科 40 黄志翱 2014011345

June 4, 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 的作用为将一个常数加到一个指定的寄存器上，为了添加这个操作，我们需要做如下修改：

- 在 instr_valid 中将该操作添加为合法操作。
- 设置为需要寄存器和常数。
- 目标寄存器和 B source 设为 D_rB。
- ALU 的两个操作数设为 E_valC 和 E_valB。
- 设置为 setcc。

2.1.2 leave

leave 的等价代码为：
rrmovl %ebp, %esp
popl %ebp
基本操作与 pop 类似，只不过是将%esp 寄存器的来源进行了修改，修改操作如下：

- 在 instr_valid 中将该操作添加为合法操作。
- 设置为需要寄存器。
- %ebp 作为 A source 和 d_dstM, %esp 作为 B source 和 d_dstM。
- ALU 的两个操作数设为%ebp 和 4。
- 将 mem_addr 设为 M_valA。
- 在所有有 popl 的异常处理代码上添加上 ileave。

2.2 代码的测试

对 pipe-full.hcl 进行修改后在 ptest 目录下进行 make TFLAGS=-il 进行测试，测试结果通过。

3 实验二

3.1 题目分析

显然该问题可以分为两个不同的问题。

- 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一个块（为了表述的简单，下面不妨设键由内存地址唯一指定，值就是一位）。

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以进行替换。

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。

在此前提下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 是读操作，可以直接读取而当 B 不存在，如果 A 要写，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 以 M 状态存在，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：A 和 B 随时监听对方的信号，当 A 想要对 B 进行任何信息交流时，A 发出信号。一旦 B 空闲下来了，就回答 A 的请求。此时 A 即可将所要发出的任意数量的信息提供给 B，A 每次等待 B 接收信息处理。这样通讯直到最后 A 发出结束信号，双方结束通讯。

具体实现上，bus 是一块共享内存，A 和 B 分别了 bus 的不同部位。A 发出的任何信号在 bus 上表现是将特定的位数置 0 或者置 1。A，B 通过轮询的方式等待对方的通讯：

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。

还有一个值得注意的细节是，由于 rmmovl 和 mrmovl 的设计，我们对内存的访问很多时候都是连续四个进行的，为了使得能够保证在读取的时候四个内存的值都不会修改，我采取了将对方一直锁住，然后进行连续四次操作的方法。

理论上而言，由于一个 block 的大小通常都会超过 4，故而实际上只要双方能够对至多两个 cache line 进行同步即可—但考虑到 block 为 1 的时候这个方法会退化到需要同步四个 cache line，于是我并未这么实现。

通讯的相关代码是 misc/cache/cache.h 的 ask 和 answer 函数。

3.2.3 block 的处理

假设 cache line 的 block 大小为 64，那么我们可以通过对地址右移六位求得对应的 block，之后所有 cache 到内存与从内存到 cache 的所有操作都对 block 同时操作即可。

理论上在 cache 中寻找指定的元素应该通过散列表进行，但考虑到实际上 cache 的大小通常非常小，为了写代码方便我使用 for 循环进行了替代。

3.2.4 实现细节

cache 相关的代码位于 misc/cache/中。

cache.h 中存放了缓存的实现与相关接口。

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。

在运行时，server, cache0, cache1，将会依次启动。之后只需要利用 msg 数组即可实现任何程序和 cpu 的交互接口：

```
//length表示我希望在一次操作中对连续多少个内存位进行处理。
//msg为通讯所使用的共享内存。
//answer表示回答另个一个缓存的询问。
//ONE_STEP为一次ask操作。

//server:
...
    while(1){
        answer(c);
        if(msg[0]){
            if(msg[0]==1){
                //Read
                int pos = msg[1];
                int ans = ONE_STEP(c, pos, 0, READ, msg[4]);
                msg[3] = ans;
                msg[0] = 0;
            }
...


//client
```

```
int read_request(volatile int* msg, int pos, int length){
    msg[1] = pos;
    msg[0] = 1;
    msg[4] = length;
    while(msg[0]){
    }
    return msg[3];
}
```

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。

为了实现这些操作，我在 cache_rec 类中添加了一个 msg 指针，指向共享内存（如果该指针为 NULL 则表示该对象是表示寄存器。）利用之前提供的交互，一段典型的 get_word_val 代码则可以写成：

```
bool_t get_word_val(mem_t m, word_t pos, word_t *dest)
{
    int i;
    word_t val;
    if(m->msg!=NULL){
        *dest = read_request(m->msg, pos, 4);
        return 1;
    }
    val = 0;
    for (i = 0; i < 4; i++)
        val = val | m->contents[pos+i]<<(8*i);
    *dest = val;
    return TRUE;
}
```

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。但是由于本题只会运行两份相同的代码，所以这么做没有任何意义。

基于对这些操作的修改，实际上我们已经完成了两个 cpu 的并行。

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，但是区别在于 test 还有一个叫做 mem_test 的信号量。

为了使得这个操作在是原子的，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，也就是进行连续 8 此的缓存操作即可。

为了在 psim 中使得该操作是连续的，需要同时修改 mem 阶段和 wb 阶段，因为这里面同时涉及到了对内存的读写与对寄存器的赋值。

对于 mem 阶段，需要加入如下代码：

```
...
    if (read && !mem_test) {
...
    if (mem_test && read){
        mem_test_address = mem_addr;
        int ans = test_memory(mem, mem_test_address);
        valm = ans;
    }
...
```

然后在写回阶段：

```
...
    if(wb_testM != REG_NONE){
        set_reg_val(reg, wb_testM, wb_valM);
        //为了视觉效果，我在test中加入了往屏幕输出test结果的功能。
        printf("PSIM_ID %d test_result:  %d on %d\n", PSIM_ID, wb_valM, mem_test_address);
    }
...
```

其中 wb_testM 的值与 wb_destM 一样，而 mem_test_address 是用于调试的全局变量，。

3.4 y86 code

当 psim 写好后，基本上 y86 代码就非常简单了。

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，对特定的内存位 test，输出 test 结果，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，累积求和，直到读取到内存中的值为 0 结束。

y86 测试代码为 pipe/ans.ys。

3.5 代码的运行已经运行结果

在总目录下进行 make 即可。

cd pipe，进入 pipe 文件夹。在终端键入 python3 run_two_sim.py 即可运行。run_two_sim.py 是一段 python 脚本，负责调用多个线程 (server, cache 0, cache 1, psim 0, psim 1)。

程序会依次从 1024 输出到 925，运行的视觉效果如图：

```
./gen-driver.pl -n 4 -f ncopy.ys > sdriver.ys
../misc/yas sdriver.ys
./gen-driver.pl -n 63 -f ncopy.ys > ldriver.ys
../misc/yas ldriver.ys
../misc/cache/server
738468442 721691226
mem: 0x10fc3a000, bus: 0x10fc42000
server begin ...
../misc/cache/cache 0
../misc/cache/cache 1
 cache begin ...
id: 0, mem: 0x107940000, bus: 0x107948000
cache 0 ready, msg_id: 196613
id: 1, mem: 0x103aeb000, bus: 0x103af3000
cache 1 ready, msg_id: 196614
psim ans.yo 0
psim ans.yo 1
msg id: 196613
msg id: 196614
PSIM_ID 1 test_result:  0 on 3848
PSIM_ID 0 test_result:  1 on 3848
PSIM_ID 1 test_result:  1024 on 4112
PSIM_ID 1 test_result:  1023 on 4112
PSIM_ID 1 test_result:  1022 on 4112
PSIM_ID 1 test_result:  1021 on 4112
PSIM_ID 1 test_result:  1020 on 4112
PSIM_ID 1 test_result:  1019 on 4112
PSIM_ID 1 test_result:  1018 on 4112
PSIM_ID 1 test_result:  1017 on 4112
```

4 感谢

感谢茅佳源同学，在做第二个实验时和他进行的讨论给了我莫大的帮助。
也感谢这门课布置了这么一个好玩的实验。