ICS Lab 4 Y86Simulator(PIPE++) 实验报告

王鹏，卢力韬 {15307130185,15307130084}@fudan.edu.cn

2017 年 1 月 6 日

1 源代码说明

- HardwareUnits.h: 硬件单元相关代码
- LogicalUnits.h: 逻辑单元相关代码
- Const.h: 规定一些常数
- pipe.h: y86simulator 核心部分
- transfer.h: 将.yo 文件转成二进制表示的文本文件
- mainwindow.cpp/mainwendow.ui: 界面相关文件

2 环境及配置

- Ubuntu16.04
- g++ 5.4.0 (C++11)
- Qt 5.7.0 (64 bit)
- 由于 Qt 许可协议的限制，我们无法发布静态编译的程序。推荐在电脑上重新编译一遍。

3 使用方法

直接双击 release 目录下的 CPU_GUI 文件，会弹出一个对话框选择打开文件。可以打开 demo 文件夹下的一些示例文件作为演示。

4 界面说明

4.1 支持功能

支持查看任意时刻:

- 每个流水线寄存器的值
- 每个内存值
- 每条指令目前处于何种阶段
- 支持输入.yo(二进制文件)

在交互方面，支持无限步的单步前进与单步后退，也支持连续多步的执行与暂停执行，并支持调整程序运行的快慢。

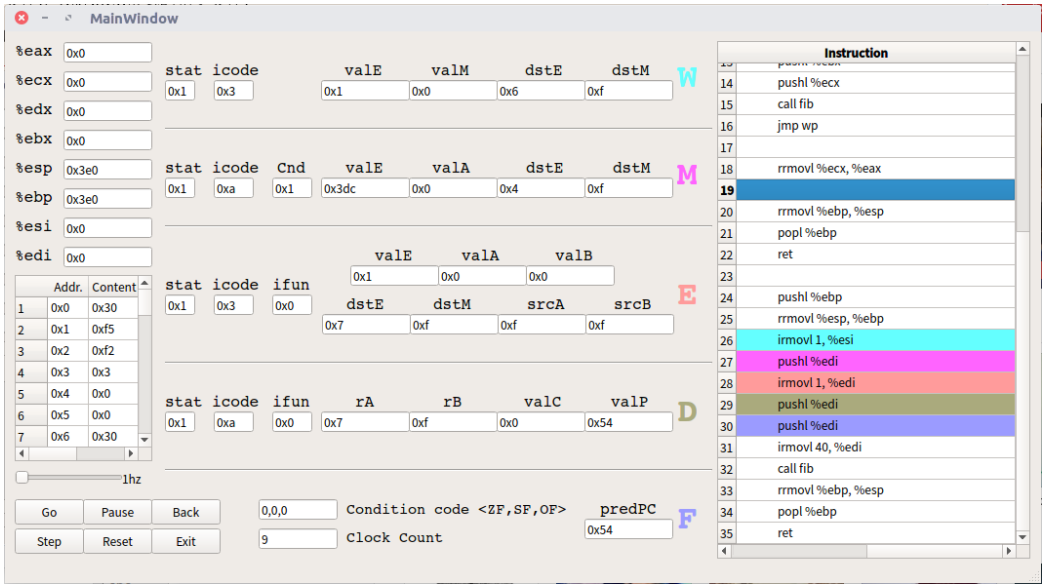

图 1: Y86Simulator 程序界面

4.2 界面实现

界面实现上采用了 Qt，Qt 是一个 C++ 的库，因此界面程序可以直接调用内核部分产生的代码。对于界面程序来说，内核部分的所有变量都是可见的。因此每个时刻将要显示什么就直接读取代码里面相应的变量就可以了。

并且使用 QtCreator 编写界面十分直观。界面设计可以直接拖空间就可以。每个空间在代码上体现的就是一个个对象。

4.2.1 每个流水线寄存器的显示

与内核的沟通用一句 #include <pipe.h> 就可以实现了。

在每一个 Clock Cycle，读取 pipe.h 内部相应的变量，刷新相应的 QLineinput 上面显示的值就可以了。

4.2.2 每个指令代码的显示

使用 QTableWidget 显示每一句指令。在读入.yo 文件时，预处理出每个指令开头地址与所在行数的对应关系。并显示在 QTableWidget 中。

为了跟踪每一条语句目前的执行阶段，在每一个阶段的寄存器中附带地传递了一个值 begin，表示当前时刻此阶段通过的指令所在的地址，这样我们就知道每一条语句目前处于流水线的何种阶段。在每一个 Clock Cycle，刷新对应 QTableWidget 行的颜色为对应阶段所代表的颜色。

4.2.3 每个内存值的显示

同样使用 QTableWidget 显示每一个内存值。这里我们查看的是每一个字节的值。显示的时候只需要到内核部分代码中的 HardwareUnits 中的 Memory 结构体中查看数组的值就可以了。

4.2.4 前进、暂停、后退等实现

我们只需要在 Qt 中将前进、暂停等按钮的按下事件与内核部分处理一步的函数绑定在一起就可以。

关于后退的实现：我们在代码中存储了每一个 ClockCycle 下 CPU 内所有的状态。因此，每一次后退，我们只需要将 CPU 状态恢复到上一个 Clockcycle 即可。

5 内核说明

5.1 支持指令：

1. nop halt ret
2. rrmovl irmvol rmovl mrmovl
3. pushl popl
4. opl cmpl iropl
5. jxx call

对任意的 y86 指令有效。

5.2 冒险问题的解决：

数据冒险：

转发：通过流水线寄存器＋一些额外的数据通路和逻辑选择单元，将前面指令所未完成的写通过 wire 转发，从而知道最新的值，能使当前指令通过流水线寄存器而不需要任何的暂停。能解决绝大部分的冒险问题。

控制以及数据冒险：

完备的流水线控制逻辑。

主要需要解决的问题：

- 处理 return：此时流水线必须暂停知道写回阶段。
- 加载/使用冒险：在一条从存储器 (内存) 中读出一个值的指令和一条使用该值的指令之间，流水线必
- 须暂停一个周期。
- 错误预测的条件分支：需要撤销掉错误的指令
- 异常：当指令导致异常，需要禁止后面的指令更新程序员可见的状态 & 停止执行

具体的处理方法：

条件	Fetch	Decode	Execyute	Memory	WriteBack
处理 RET	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常

相关冒险的测试代码：

```
// Ret Hazard
  0x000: 30f340000000 |   irmovl mem,%ebx
  0x006: 504300000000 |   mrmovl  0(%ebx),%esp
  0x00c: 90           |   ret
  0x00d: 00           |   halt
  0x00e: 30f605000000 | rtnpt:  irmovl $5,%esi
  0x014: 00           |   halt
  0x040:              | .pos 0x40
  0x040: 50000000     | mem:  .long stack  0x050:
  0x050:              | .pos 0x50
  0x050: 0e000000     | stack:    .long rtnpt

//Mispredicted Branch:
  0x000: 6300         | xorl %eax, %eax
  0x002: 7413000000   | jne target
  0x007: 30f202000000 | irmovl $2, %edx
  0x00d: 30f303000000 | irmovl $3, %ebx
  0x013:              | target:
  0x013: 30f001000000 | irmovl $1, %eax
  0x019: 00           | halt

//Data Hazard:
  0x000: 30f0e8030000 | irmovl $1000, %eax
  0x006: 30f39a020000 | irmovl $666, %ebx
  0x00c: 403000000000 | rmmovl %ebx, (%eax)
  0x012: 501000000000 | mrmovl (%eax), %ecx
  0x018: 2012         | rrmovl %ecx, %edx
  0x01a: 00           | halt
```

5.3 新增指令（Bonus）

cmpl,iropl 和 ircmpl。

cmpl: 寄存器值与寄存器值的比较

iropl: 立即数与寄存器之间的运算并赋值到寄存器中 (包括 add sub and xor)

ircmpl: 立即数与寄存器之间的比较

具体的处理方法：

```
//cmpl rA, rB
Fetch: icode:ifun<-M1[PC]   rA:rB<-M1[PC+1]  valP <-PC + 2
Decode: valA<-R[rA]  valB<-R[rB]
Execute: valE<-valB - valA   Set CC
Memory:
Writeback:
PC update: PC<-valP

//iropl V, rB
Fetch: icode:ifun<-M1[PC]   rA:rB<-M1[PC+1]  valC<-M4[PC+2]   valP
   <-PC + 6
Decode: valB<-R[rB]
Execute: valE<-valB OP valC   Set CC
Memory:
Writeback: R[rB]<-valE
PC update: PC<-valP

//ircmpl V, rB
Fetch: icode:ifun<-M1[PC]   rA:rB<-M1[PC+1]  valC<-M4[PC+2]   valP
   <-PC + 6
Decode: valB<-R[rB]
Execute: valE<-valB - valC   Set CC
Memory:
Writeback:
PC update: PC<-valP
```

5.4 效率上的优化（Bonus）

课程中中 PIPE 对 ret 指令的处理：将 ret 的“下一条”指令阻塞在取指阶段，直至上一条指令访存阶段结束 (即进入写回阶段)，这样做有一个缺陷就是：效率不高，因为对于每次的 ret，都会出现三个周期的时间的浪费。

因此我们采取了这样的做法：加入程序员不可见的硬件栈，对 ret 的返回地址在 Fetch 阶段进行预测，这样就没有暂停。

在 call 的时候将下一条指令压入栈中；在 ret 的时候将栈顶弹出并作为 ret 的返回地址的预测值。这样做的原理是一般情况下函数的调用和返回是成对出现的，基本上都能预测正确。但是这样子做是可能会出现预测错误的情况的，比如这两种情况：

1. jxx(条件跳转) 预测错误带来的连锁效应。

2. 有时候会出现手动压返回地址的情况，没用 call 就 ret。

所以我们需要一个恢复机制：

对 ret 预测正确性的判定，可以在 ret 完成访存阶段 (Memory) 后得到。如果预测错误，那么给 M,E,D 流水线寄存器插入 bubble 信号来撤销错误的操作即可。

还有一个问题，就是如果 ret 返回值预测错误的话，我们加入的硬件栈会被污染。所以我们需要一个可以快速恢复的栈：每次操作只新建，不修改（其实是一个树的结构）

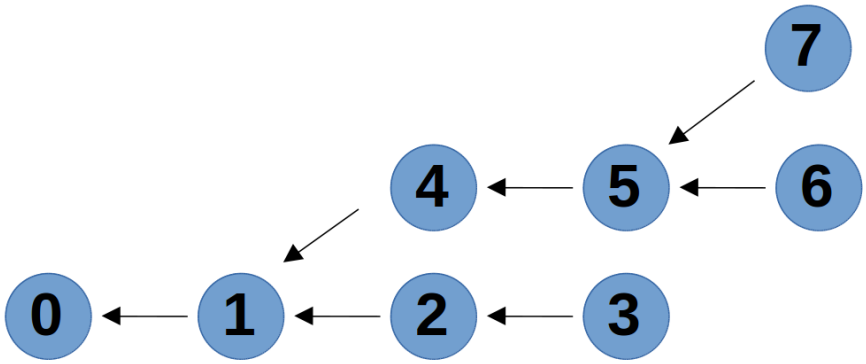

图 2: 恢复机制示意图

如图2所示: 在时刻 1,2,3,5,6 对栈进行的是 push 操作, 在时刻 4 进行的是 pop 操作, 在时刻 7 没有操作。如果需要恢复, 只需要将时间戳 -3(ret 预测错误) 或者-2(jxx 预测错误) 即可。

还有一个缺陷: 由于最早的错误指令会执行完 Execute 阶段, 程序员可见的条件码可能会被污染, 但是注意到 SelectPC 在需要用到条件码的时候, 条件码已经被正确的指令更新了, 不会影响整体的逻辑 (不过这的确是个缺陷, 其实如果把条件码也改成可持久化的就可以解决这个问题)

测试结果 (递归求 fibonacci 数列第 n 项):

初始值	10	20	30	40
加入硬件栈前	217(周期)	417	617	817
加入硬件栈后	184	354	524	694

可以看出效率提升了很多。

6 lab 过程和感想

王鹏：在这次的 project 中，我负责的是 kernel 部分。

最开始对于这个 kernel 我毫无头绪，不知道从何下手，所以只能按照自己的理解先把大概的框架弄出来，后来根据教材提供的 hcl 语言和 pipeline 的配图，了解了每个单元具体所做的事情以后就比较好些了。因为我对于每个阶段的 pipeline 都开了一个 struct，中间过程中有有些地方是调用当前的 pipe(组合逻辑) 还是上一个阶段的 pipe(时序逻辑) 我没有分得太清楚，结

果调 bug 调了很长的时间，我认为这是不应该的，最好是一开始就把组合逻辑和时序逻辑分清楚，这样就不会出那么多 bug。后期的难点是控制逻辑的设计，因为这是一个很烧脑的事情，要自己先模拟好冒险的情况和处理方法，才能清楚自己在写什么东西。

但是光完成教程上的东西是远远不够的，于是我想在写好的 pipe 的基础上加入一些其他的东西。首先是加入新的指令，cmpl 这 iropl 这两个指令加上去不算很难，后面我还想加入一个比较奇葩的 swapl(交换两个寄存器的值) 的操作，但是后面发现 register files 只有一个写端口，后来这个想法就报废了。

然而我在书上翻到了一个扩展阅读，里面介绍了现代的 cpu 对于 call/ret 操作是有一个专门的硬件栈进行预测的，我想了想觉得这是个很好的 feature，而且感觉是可以实现的。于是我尝试去实现了这个功能，后来还调了整整一天的 bug 才把教材提供的测试文件们给调过去。最终测试是效率的确提高了不少，非常开心。

总而言之，我觉得这次的 y86-simulator 的设计非常的有意思，而且能让我们更加深入地理解计算机系统，虽然中间过程很坎坷，但是这对于我而言是非常有意义的事情。

卢力韬：在这次的 project 中，我负责的是界面部分。并且还实现了一个 Y86 的 as（编译器）(虽然后来没用到……)。

这次写 PJ 也是我第一次写有界面的程序。刚开始我也不知道如何实现一个界面。通过网上查找一些资料，最后选定了 Qt 库。在编写的过程中，要看到很多队友的代码，以及有稍稍修改队友的代码。这对我也是一种比较新的体验，因为要将自己的代码嵌入到别人代码中。还不能将别人和自己的代码改错。特别提一下，因为是两个人写代码，我们用 github 实现我们代码的同步（虽然用的还不是很熟练）。

自己实现的 Y86 编译器也带给我挺多收获，首先，因为程序输入格式约束并不严格，所以程序要应对所有的情况，所以编写起来比较繁，后来看到有人用 regex 先预处理一下输入，这是一个很好的方法。

通过这次写 PJ，我学习了一下 Qt 的用法。并且因为要看懂队友的代码，我们也复习了很多 CPU 流水线的知识。总的来说，这次 PJ 还是很有意思的。