Y86 流水线模拟器的实现
计算机原理项目报告

游沛杰 13307130325

June 9, 2015

Contents

1 背景介绍

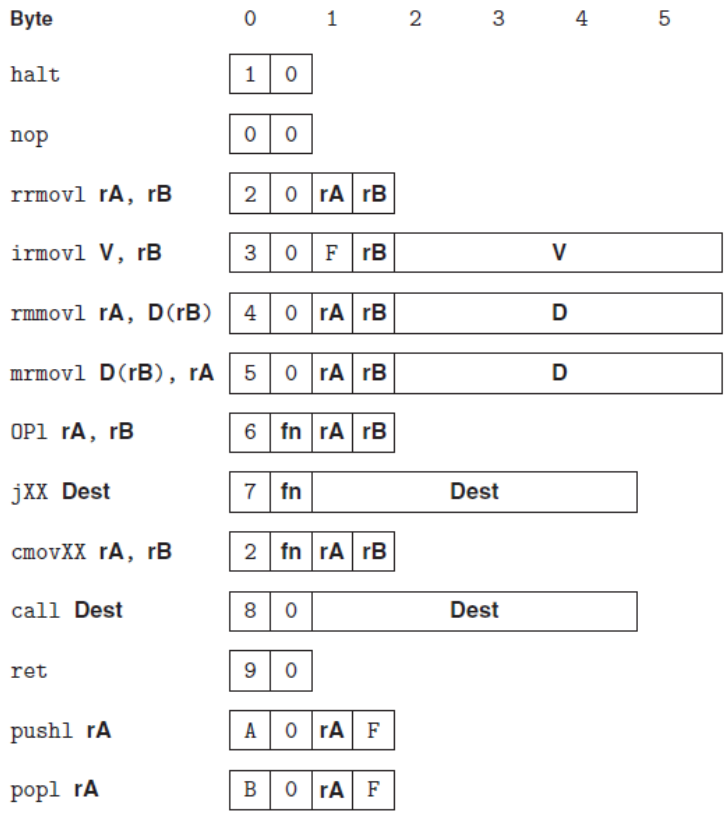
在本次项目实验中，我们需要实现一个 Y86 模拟器，能够正确的流水线化模拟执行课本 (CS:APP[1]) 第 4 章介绍的 Y86 指令，并将执行过程和程序运行结果通过图形界面的方式可视化。同时可以帮组我们更好理解课本介绍的知识。我认为这次项目的目的是忠实还原 CPU 处理 Y86 汇编程序时候的状态，所以有时候考虑问题会从硬件方面出发。

2 工作原理

这里分指令集，流水线阶段，流水线异常处理等部分来介绍

2.1 指令集

实现了书本上介绍的全部 Y86 指令，如下



其中，地址的形式为 D(rB)，而指令跳转使用绝对地址的方式（比如 jmp 0x123，那么下一条要执行的指令就是 0x123 开始的指令，而不是 pc+0x123 或者 pc+0x123+0x2）。

数据存储使用小端法。[3]
另外需要说明几点：

1. 书本上的 halt 和 nop 对应的机器码不明确（见第二版英文书 P338 和 P384，前后矛盾）。所以在这里，我把 10 看成是 halt 指令对应机器码，00 看成是 nop 指令。
2. 样例给出的 RNONE=0x8，而书本上是 0xF，在这里我们使用的是 0xF
3. 当 fetch 到的指令发现不需要常数 valC 的时候，程序不会读取后面的内容，并把 valC 设成 0，而样例结果会把后面的指令当成常数读入，并没有什么用。

2.2 PIPELINE

我实现的流水线分成五个阶段 (Fetch, Decode, Execute, Memory, Write back)

1. Fetch 从指令内存中取出指令以及相关信息 (rA、rB 编号, valC)
2. Decode 读取相关的寄存器的值 (rA, rB, RESP 的值)
3. Execute 进行算术/逻辑运算
4. Memory 读写内存
5. Write back 将需要更新的寄存器值放回 register file 中

使用 forward+stall/bubble 来处理异常情况。

- forward 将前面阶段中已经计算好的值直接传输到 decode 阶段
- stall 不执行更新，但是保留原来流水线寄存器中的值 (延迟到以后再更新)
- bubble 把 pipeline register 中保存的值重置置为默认值 (舍弃当前已在进行的指令操作)

2.3 Memory

同步 RAM，后面有介绍

3 我的项目概况

全部代码均作者本人独立完成，并没有任何抄袭和复制粘贴的成分。

项目的设计和准备阶段从 5 月 16 日开始，经历两个星期（其中有另外 2 个项目需要完成）。从 6 月 1 日左右开始投入实际开发。其中核心部分历时 4 天编写并完成调试，UI 部分做了 4-5 天。

在这个项目之前几天我们还完成了数据库的课程项目 (HTML+PHP 实现)，所以作者这次不想再写 web 端的 UI，试图使用不会的 Qt 来实现一个桌面端应用程序。虽然界面不一定好看，但是也是一种挑战。

Table 1: 开发环境

主要使用语言	内核	Python 2.7.9
	GUI	PyQt 4.11.3
开发平台	IDE	PyCharm
	操作系统	Windows 8.1
辅助工具	GUI 界面绘制	Qt Creator

4 内核具体实现

4.1 中间变量的实现

这里的中间变量指的的 stage output，通常以小写字母 fdemw 开头 (比如 e_valE)。

对于这些结果，在硬件上相当于一个组合电路，那么我们就用一个函数来实现，其中函数的输入就相当于硬件层面上的接线，组合电路就在函数内部实现其功能，比如下面例子：

```
def f_stat(f_icode, imem_error):
    #   DONE
    #   有优先级？
    if imem_error: return SADR
    if not instr_valid(f_icode): return SINS
    if f_icode == IHALT: return SHLT
    return SAOK
```

其中部分 stage output 只用到了一次，我们在需要的时候再计算这个结果，以保证程序整体运行速度，而对于在不止一个地方用到的值，我们采用中间变量来保存这些值。

我们可以发现，Python 的语法与课本上的 HCL 有很多相似之处，比如有 in 的用法，Python 的 list 理解起来也比较简单，如下语句：

```
def d_srcB(D_icode, D_rB):
    #   DONE
    if D_icode in [IOPL, IRMMOVL, IMRMOVL]: return D_rB
    if D_icode in [IPUSHL, IPOPL, ICALL, IRET]: return RESP;
    return RNONE;
```

我们发现 Python 在这里看起来就和真的硬件描述语言一样，同时十分接近自然语言，给人带来愉悦感！这也是其他语言（比如 C++）所没有的。

使用 Python 是一个正确的选择。

4.2 内存的实现

下面介绍我的“内存”的实现方式。

在 Python 中，我用一个 list 对象来保存内存，假设的大小为 4k 字节（0x1000 个地址），对于绝大部分的 Y86 汇编程序（包括样例程序）足够了。

其中指令以及常数存储从低地址开始存储，程序运行时从高地址开始堆栈，如图。

同时我们假定这是一个同步的内存，也就是在时钟上升沿的时候我们才进行写入操作。

在具体的读取内存操作时候，有两种情况

1. 读指令，这时候程序会判断当前读的是否合法的指令地址

```
def inst_valid(addr):
  return addr in range(0, max_instr_addr + 1)
```

思考一下发现并没有在指令区间中，而读出来其实不是指令的例子（只能是 Y86 汇编代码写错了）。所以指令译码错误我们放在模拟器主程序中判断，如果指令非法返回一个错误提示。

2. 读数据。在程序开始运行之前，我们甚至都无法判断一个地址放的是指令还是数据，参见样例汇编程序中从地址 0x14 开始有一个数组：

```
  0x014: 0d000000     | array:  .long 0xd
  0x018: c0000000     |   .long 0xc0
  0x01c: 000b0000     |   .long 0xb00
  0x020: 00a00000     |   .long 0xa000
```

说明数据和指令是混在一起的，所以在读取数据时，我们能只判断地址是否越界，不能很好判断这个地址是否落在“数据”的地址范围。

在写入内存的时候，考虑到这是一个同步的 RAM，所以不能异步的写入。我实现的办法是先把这个写操作记录到一个类似于缓冲区的东西里，当在时钟上升沿的时候再修改。

有点类似于 Git 的 stage 和 commit 操作，如果需要修改，我先加入到 stage 里面，等时钟上升沿再一起 commit。如下例子：

需要注意的是，这种实现方式从原理上与以下程序有所不同

```
next = [0] * 233
while True:
  this = next
  do_something()
  # 从this里面读数据，写到next里面
```

这里是有两个内存，并不很符合硬件的实际，同时如果内存较大，则需要两倍的空间。

而我每次只记录需要写入的地址和数据信息，每个时钟周期都会清空。

同时也和下面程序不同

```
this = [0] * 233
while True:
  do_something()
  # 排列读写this的顺序，保证写入以后不会在同一个时钟周期读到。
```

上面的简直就是组合逻辑电路，并没有反应真实情况。

另外我们还支持把内存保存到文件，和从文件载入内存的操作。可以方便同学们汇编程序调试到一半的时候关闭电脑，在下次继续之前的进度调试。

对应具体的实现方式，我们使用 Python 的一个叫 pickle 的库，他可以把一个对象保存到文件（不是单纯的转成字符串再写入）对应函数如下：

```
pickle.dump(memory, file)   # 保存
memory = pickle.load(file)  # 载入
```

4.3 寄存器的实现

寄存器，包括流水线寄存器，实际上也是一个同步写入的元件。

所以我们可以用和内存相似的方法来实现，不过需要考虑 stall 和 bubble，要给每个元件一个默认值（比如 D_icode = 0, D_rA = 0xF）。

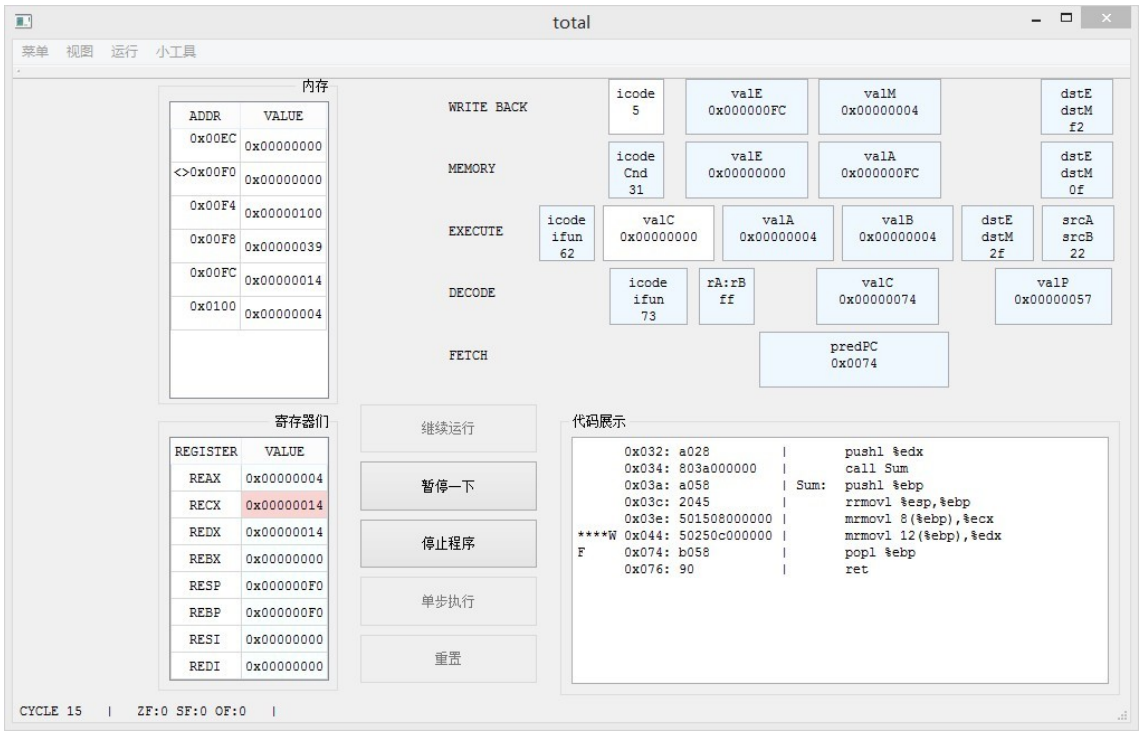
在实现的时候，我是直接在内存的最后面分出一小块空间来存储寄存器的值，同时保证只能通过寄存器名称访问他们，而不能直接根据地址读出值（因为上面提到读数据有地址越界判断）。

5 UI 介绍

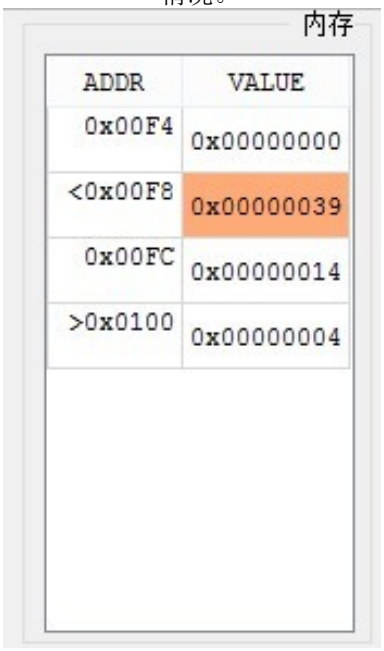
5.1 界面介绍

我们可以很清楚看到寄存器/流水线寄存器的值，以及当前程序堆栈等情况。

在对应的值改变的时候，都会有颜色变化的动画效果。



程序运行时栈，用“<>”显示当前的栈帧 (callee)，同时还会显示 caller 的栈情况。

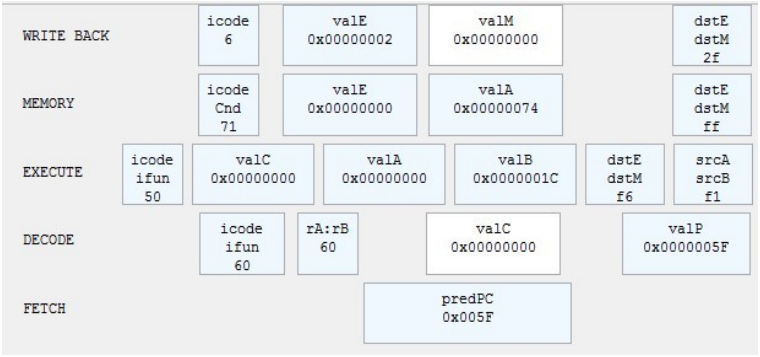


8 个寄存器对应的值。

寄存器们

REGISTER	VALUE
REAX	0x00000004
RECX	0x00000014
REDX	0x00000004
REBX	0x00000000
RESP	0x000000F0
REBP	0x000000F0
RESI	0x00000000
REDI	0x00000000

流水线寄存器，这个周期没有用到的 pipeline register 会保持白色。

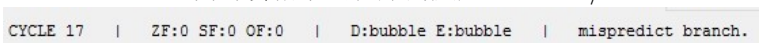


代码展示，可以看到各个阶段当前执行到什么指令（下图初始全是在 0x000 处）

代码展示

```
FDEMW 0x000: 308400010000 | init: irmovl Stack, %esp
      0x006: 308500010000 |       irmovl Stack, %ebp
      0x00c: 7024000000   |       jmp Main
      0x014: 0d000000     | array:.long 0xd
      0x018: c0000000     |       .long 0xc0
      0x01c: 000b0000     |       .long 0xb00
```

下方有状态栏，显示出当前执行到第几个周期。如果遇到冒险会提示是怎样的 hazard，同时会指示出程序的反应（bubble/stall）。



5.2 怎样运行程序

可以通过按屏幕中的按钮运行，也可以在菜单中调不同速率。

其中要说明的是停止的时候，虽然界面显示的值并不更新，但是下次继续运行的时候会从头开始。

另外并没有“后退一步”这种功能，因为普通的 IDE 都不会有这种功能。(一个 CPU 怎么能重置成历史时刻的数据? 不符合实际，只能从头开始运行)

5.3 菜单栏其他功能

另外用户界面还有以下功能

1. 文件操作
   - 载入新的 Y86 指令文件
   - 保存当前进度 (内存和寄存器情况) 到指定文件 (我表示为.pk 格式)
   - 载入进度文件
   - 导出结果到文件中，以 txt 的格式
2. 视图，提供几种不同的视图，能让用户看到他们只关心的内容
   - 全局视图
   - 只看流水线寄存器
   - 只看寄存器和内存
3. 运行
   - 以不同速率来运行，为了图形界面的显示效果 (慢速运行的时候有颜色，如果颜色闪烁太快会对用户造成精神污染，所以我在快速运行的时候去掉颜色变化)，这里只提供几种速度，单位是 instruction per second(IPS)
   - 运行，单步，暂停，停止等控制，同上。
4. 其他小工具，包括
   - 根据地址，读取当前内存对应的数据
   - 直接修改内存的值

6 UI 的关键实现

6.1 模拟运行

为了后面程序更好的操作，我们把模拟器封装成一个类 Simulator()。Simulator 包括以下方法

1. init() 初始化
2. step(update_fun = None) 单步执行，同时把结果通过传入的 update_fun 函数反馈到界面上
3. run_all() 一次执行全部代码并输出

在程序 commit 内存修改的时候，会调用 update_fun(addr, value)，告诉 UI 界面需要更新 addr 这个内存/寄存器，刷新界面。

6.2 颜色的更新

在值改变的时候 (可能) 会有颜色变化，然后过了一段时间又会重新变成白色（类似于 cool down），这样的功能使用 CSS 和 HTML 可以简单实现 (比如说使用-webkit-transform) 等，然而我这次要用的是 Qt，所以我手动实现了这个功能。

7 其他功能

8 遇到的问题

9 项目总结

10 Thank You

截至本报告完成的时候，代码已经没有任何 bug。考虑到今天到明天还会继续整理和优化代码，可能在最终应用程序会引入一些小问题，请谅解。

View this project on GitHub[2]

Report is written in LaTeX

References

[1] `http://www.csapp.cs.cmu.edu/`

[2] https://github.com/kjkszpj/AE86

[3] http://en.wikipedia.org/wiki/Endianness#Little-endian