记分牌指令动态调度算法模拟程序  
设计实验报告

计算机学院 宋子恒

**2016140402**

1 实验目的：

模拟记分牌算法，实现指令的动态调度，加深对指令相关性的理解。  
2 记分牌指令调度算法的原理：  
 尽可能提早指令的执行。当一条指令暂停执行时，如果其他后继指令与暂停指令  
及已发射的指令无任何相关，则仍然可以发射，执行。  
每一条指令都要经过记分牌（记录在“指令状态表”中），即发射到记分牌数据结构  
中； 纪录数据相关性，把源和目的操作数记录在“功能单元状态表”中； 决定指令何时  
可读出操作数（检测是否有 RAW 竞争）决定指令何时可以开始执行（一旦操作数就  
绪，立即开始执行），若某条指令不能立即执行，则由记分牌监控硬件的每一变化，并决定何时才能开始执行；控制指令何时能将结果写入目的寄存器（即检测是否有 WAR 竞争）

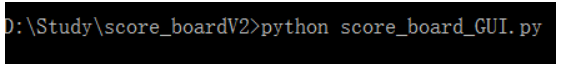
3 实验内容：

该实验通过 Python 来实现了对记分牌实验的模拟 。程序运行说明如下：

(1)打开 cmd 命令行 通过命令行来使 Python 解释器来运行脚本。 命令如下



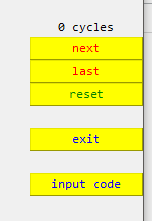
图。 要运行的脚本是 score\_board\_GUI.py (另外注意cmd的工作目录要切换到score\_boardV2 目录下)：



图： 运行对应脚本程序的命令行

(3) 运行上述命令后会弹出相应的程序对话框，此时可以点击按钮进行程序的执行并

在表格中观察程序执行的结果。



图：程序的按钮（位于弹出的界面的右上角）

单击击 next 可以执行一个时钟周期。

单击last 可以查看上一条指令的执行情况。

单击 reset 可以恢复到 0 时钟周期 重新从源文件中载入代码并清除原有的内部执

行状态。

单击 exit 可以退出程序。

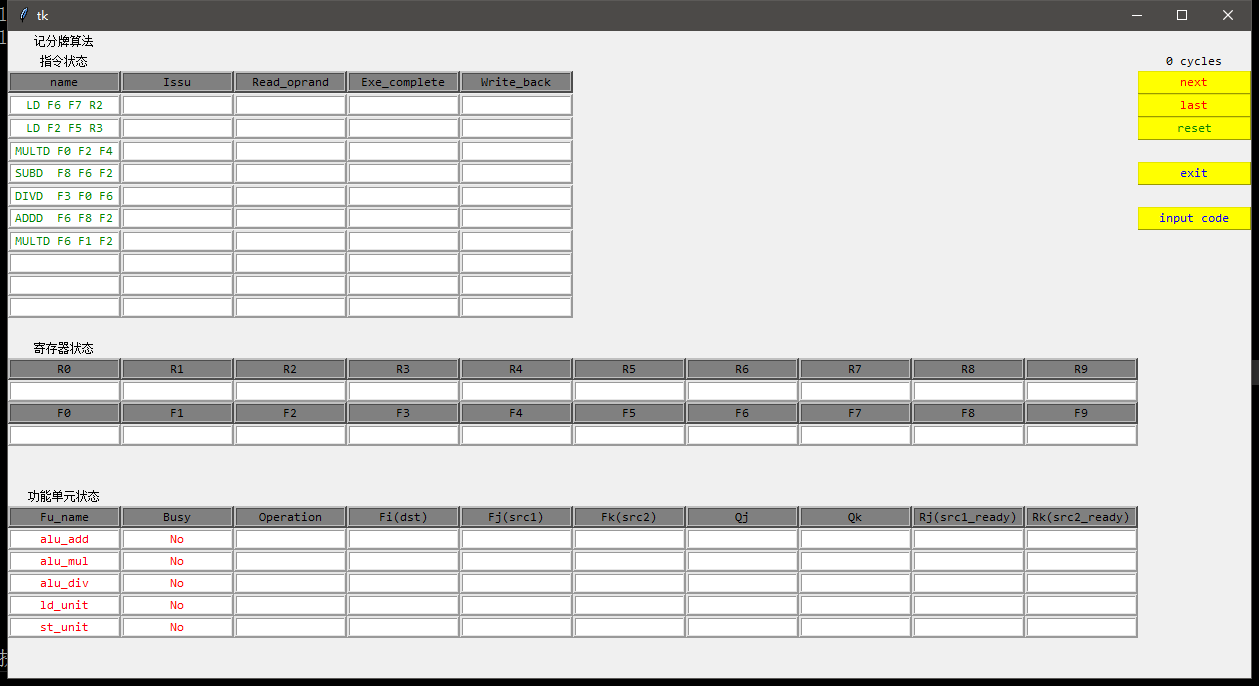
单击input code按钮会弹出代码输入框 可以输入十条汇编代码，输入后点击下方的save按钮 点击后 如果输入的代码有误将会提示语法错误”syntax error”保存无效(输入整行无效的空格也认为是错误) 必须输入正确的代码后才会显示save complete！在save complete之后关闭代码输入框。再次到主界面点击reset 重新载入代码即可运行

需要注意的是：

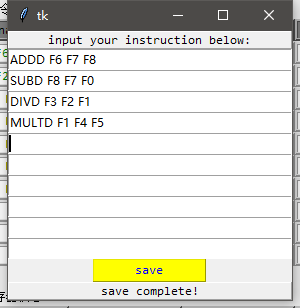
(1)在按钮上方的字显示的是当前执行到了第几个时钟周期。

(2)如果所有指令执行完毕 按钮的左侧会出现提示信息 提示“指令已经执

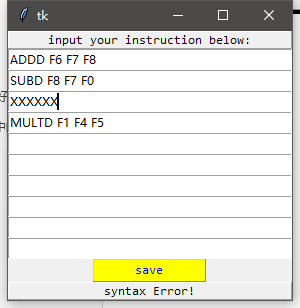
行完毕需要重新复位程序 或点击last按钮倒回去”



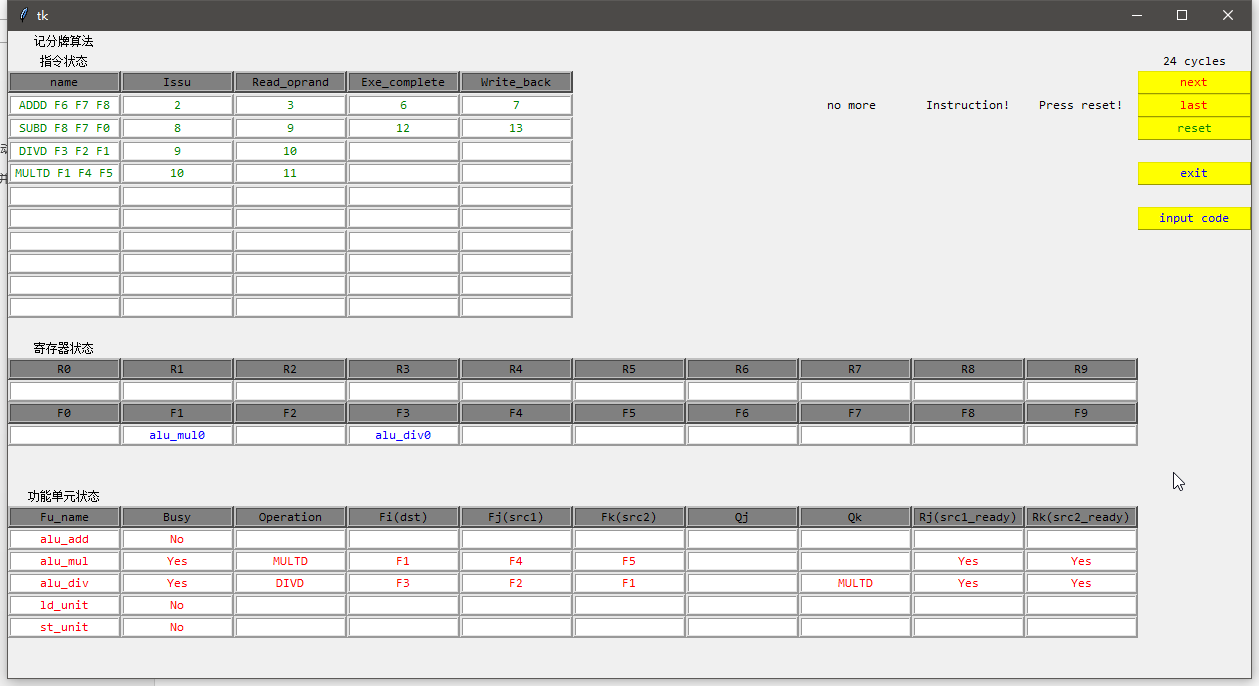
图：程序刚打开时的状态(表格内会载入程序默认的代码)



图：弹出的代码输入框（保存成功后）



图：弹出的代码输入框（保存失败后）



图：程序执行过程中的状态

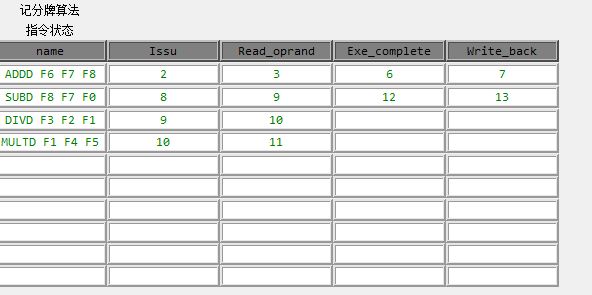
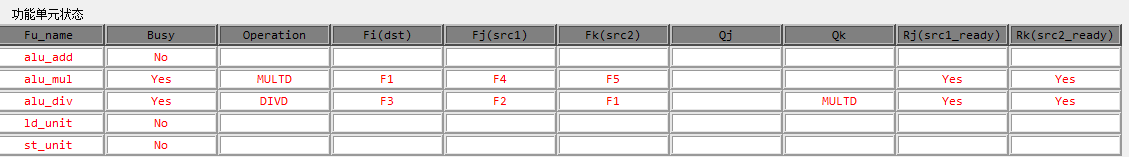
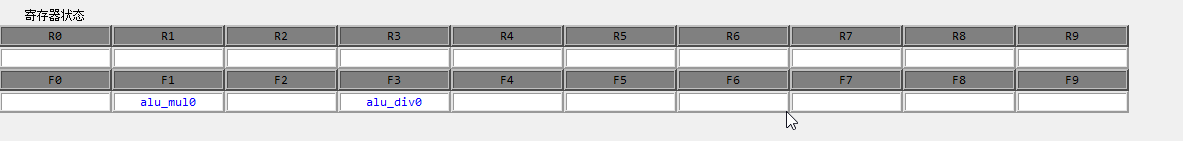


图 指令状态表的状态

  
图 执行过程中功能单元的状态



图：执行过程中的寄存器状态



图：所有指令执行完成后的提示信息

(4) 另外本程序支持自定义指令集 以兼容不同种类的处理器的模拟

定义指令集需要修改 score\_boardV2 下 的 instruction.xml 文件

该文件的格式如下：

<instruction> <!-- an instruction type-->

<name>ADDD</name>

<execute\_cycle>2</execute\_cycle>

<function\_unit>alu\_add</function\_unit>

</instruction>

<instruction> <!-- an instruction type-->

<name>SUBD</name>

<execute\_cycle>2</execute\_cycle>

<function\_unit>alu\_add</function\_unit>

</instruction>

<instruction> </instruction> 之间的内容为一条指令所需的信息

<name> </name> 之间的内容为指令的名称

（注意提醒那些的内容与标签之间不能有空隙以及其他字符）

<execute\_cycle> </execute\_cycle> 之间的内容为该条指令执行所需的时

钟周期 （注意提醒那些的内容与标签之间不能有空隙以及其他字符）

<function\_unit> </function\_unit> 之间的内容为该条指令执行所需用到

的功能单元（注意提醒那些的内容与标签之间不能有空隙以及其他字符）

4 数据结构及说明

数据结构主要在于程序中定义的三个类

class Fu:#功能单元类

def \_\_init\_\_(self,fu\_type="No\_type",fu\_index\_in\_type=0):

self.fu\_type=fu\_type #功能单元的类型

self.fu\_index\_in\_type=fu\_index\_in\_type #功能单元的序号

self.is\_busy=False#功能单元忙标志

self.busy\_inst\_ID=-1# 指明是那一条指令占用了该 FU

self.op="No\_op"#当前功能单元上正在执行的指令操作

class Reg:#寄存器类

def \_\_init\_\_(self,reg\_type="No\_type",reg\_index\_in\_type=0):

self.reg\_type=reg\_type #寄存器的类型 是整数寄存器还是浮点寄存器

self.reg\_index\_in\_type=reg\_index\_in\_type

#寄存器在一个类型中的序号

self.tobe\_read=False #寄存器是否正在被读取

self.tobe\_read\_inst\_ID=[]#等待读取该寄存器的指令的 ID 的队列

self.tobe\_write=False #该寄存器正在被写的标志

self.tobe\_write\_inst\_ID=[]#要写该寄存器的指令 ID 列表

class Instruction:# 指令类

#指令自有变量

self.inst\_ID=inst\_ID #指令学了中指令的唯一标号

self.full\_instruction=full\_instruction#源文件中的原始指令

self.exe\_state=exe\_state#指令当前的执行状态

#UN\_LOAD(未执行)

#IF (已取指)

#ID1(已发射到功能部件等待操作数)

#ID2(已等到所有操作数待执行)

#EX0(正在执行)

#WB0(已达到执行周期表明已经执行完毕等待写回)

#FIN(执行完毕 可以从指令状态表中清除)

self.full\_exe\_cycles=full\_exe\_cycles#完成 EX 步骤所需要的

#的周期数

self.current\_exe\_cycles=current\_exe\_cycles#当前已经执行的

#周期

#记分牌相关变量

self.fu=self.convert\_str\_to\_wiget(fu) #需要使用的功能部件

self.dst\_reg=self.convert\_str\_to\_wiget(dst\_reg)

#结果的目标寄存器

#type tuple (type,in\_type\_index)

self.src\_reg1=self.convert\_str\_to\_wiget(src\_reg1)

#操作数寄存器 1

self.src\_reg2=self.convert\_str\_to\_wiget(src\_reg2)

5 编程思想：

主要的编程思想源于对CPU内部硬件结构的模拟，所以使用Python的面向对象的优点，将CPU的内部结构用python中的类与对象来表示对应的结构。 CPU的内部是许多的功能单元以及寄存器组成的，记分牌算法的主要思想就是对各种资源进行标记，保证资源使用的互斥性。

另一方面，将每一条指令看做是一个对象，每条指令都有其内部的执行状态，然后指令执行的下一个状态将由指令的当前状态以及当前CPU中资源的占用情况来决定，下面以一个例子来说明：

假设指令I现在已经取指成功（处于IF状态），它所期望进入Issu状态。 这时，在当前的一个时钟周期内，它会检测他所需要的功能单元Fu是否被其他的指令所占用。如果没有被占用，那么它就通过修改自己内部的状态变量来表示自己进入了Issu状态，否则继续保持IF状态 等待下一个时钟周期再对自己所请求的资源进行检查。整个的流程如下图：



图 指令的IF状态转移图