# RISC-V 指令集模拟器项目开发文档

张嘉乐 518030910319

2019 年 7 月 10 日

## 目录

# 1 开发历程

本项目输入一个 C 程序对应的 RISV-V 指令，解析指令最终输出 C 程序的主函数返回值。在此基本功能基础上尽量模拟 CPU 处理指令时的五级流水并行结构。

一个多星期的开发历程中，主要经历了以下四步：

## 1.1 第一阶段 7.2-7.4

写出基本版，仅包括 IF,ID,EX 三个阶段，EX 阶段直接读写寄存器和内存。每个指令类对基类继承，多态实现 EX，实现了该程序的基本功能，评测通过。

## 1.2 第二阶段 7.5

写出串行五级流水结构。重构代码，由类继承多态实现改为在一个基类中使用选择语句，评测通过，用时缩短到 40

## 1.3 第三阶段 7.8

写出并行五级流水，对于控制冲突和数据冲突均采用停机等待。由于同时执行多条指令，用 stl 队列存储不同阶段的指令，运行结果正确，评测超时。

## 1.4 第四阶段 7.9-7.10

改用 forwarding 处理数据冲突并寻求加速。用 gprof 分析程序用时发现是 stl 队列占用大量时间，改用手写队列，采用 O2 编译优化，评测通过，时间与之前的串行流水相仿。

# 2 项目框架

从程序过程上看，主程序结构分六步：读入程序到内存，循环执行五级流水 IF,ID,EX,MA,WB。

从数据结构上看，有三个类：内存，寄存器，指令。内存类中封装有模拟内存的数组和读程序函数；寄存器类中封装有模拟寄存器的数组，pc 寄存器，寄存器锁，数据传递区；指令类中封装有指令的 EX,MA,WB 三个过程。

以下先介绍串行结构的运行过程，再讨论 forwarding 处理冲突的并行结构实现。

## 2.1 串行五级流水结构

### 2.1.1 读程序

程序输入的以 RISC-V 指令二进制码的十六进制表示给出，其间混有表示程序地址的数字。这部分解析十六进制数并将其存到模拟的内存中。模拟的内存为字符数组，一个位置一个字节。

### 2.1.2 指令获取 IF

根据 pc 寄存器的值从内存相应地址获取 32bits 长度二进制数作为 RISC-V 指令。

### 2.1.3 指令解析 ID

先根据 RISC-V 手册将 32 位的二进制机器码转化为一条汇编指令结构体，包括指令名称 (枚举类型)，立即数，寄存器 1，寄存器 2，目标寄存器，再把寄存器里面的值读取到运算器临时变量里面。有些指令不含有的部分取默认值即可。

具体实现方法是先分析操作码和函数码,得到指令的类型和格式 (R,I,S,B,J,U 六种)，根据格式进一步解析出 rs1,rs2,imm。

### 2.1.4 指令执行 EX

为了模拟真实情况，运算器不应直接访问寄存器，因此使用 ID 阶段已经从寄存器里读取出的变量进行运算，结果仍然存在运算器临时变量中。具

体运算内容由指令说明而定。

#### 2.1.5 内存访问 MA

仅针对 Store 指令和 Load 指令，根据 EX 阶段计算好的内存地址，从内存中获取变量到缓存或者将结果写入内存。

#### 2.1.6 寄存器回写 WB

将从内存读取的数据或者计算得到的数据写到相应寄存器中。

### 2.2 并行处理的结构和冲突

#### 2.2.1 并行处理的结构

C++ 程序是串行运行的，模拟并行时，认为一次 while 循环表示一个时间单位，其中的几个过程是同时执行的。

实际程序中模拟的每一个时间单位先执行靠前指令的后阶段，再执行靠后指令的前阶段，因此循环中的执行顺序是 WB,MA,EX,ID,IF。

#### 2.2.2 并行处理的冲突及其处理

同时执行多条指令能导致前一条指令还没有将结果写回寄存器，后一条指令就已经读取了寄存器的值进行运算，这样导致的错误叫数据冲突；条件跳转 Branch 指令改变了程序指令的获取顺序，如果 IF 仍然向下读取指令就会出错，这是控制冲突。

最简单处理控制冲突的方法是给寄存器，包括 pc 寄存器上锁，pc 寄存器上锁后 IF 就不再读取指令，等待跳转完成解锁后再读取指令；其他寄存器上锁后访问到被锁寄存器的指令暂停运行，直到解锁为止。

经常性的暂停会导致并行结构向串行靠近，在实际 CPU 中效率下降。为了保持并行，针对数据冲突可以采取 Forwarding 处理方法，控制冲突可以采取分支预测。

Forwarding 是指令的寄存器回写 WB 阶段之前就将运算结果放到缓冲区传给 ID 阶段读寄存器过程中，ID 阶段根据上锁情况在寄存器和缓冲区数据中选择。

分支预测是在得到 Branch 指令后根据该 pc 位置的历史跳转情况进行位置跳转，继续执行后续指令，Branch 指令的 EX 阶段执行后发现预测正

确则继续执行，预测错误就把预测错误的运行结果删去，重新跳转 pc 获取指令。

#### 2.2.3 Forwarding 处理在本模拟器中的具体实现

针对控制冲突，加入一个 pc 锁标记，ID 阶段发现跳转指令，分支指令就将其置 1，等到该条指令执行完在 WB 阶段将其置 0.

针对数据冲突，加入寄存器锁和寄存器修改标记及其修改值，所有有目标寄存器的非 Load 指令均在 EX 阶段得到结果后增加相应寄存器修改标记并存入修改值，ID 阶段查看 rs1,rs2 两个寄存器是否有修改标记，没有则读取相应寄存器，若有则读取相应修改值。

Load 类指令的 MA 阶段会晚于其下一条指令的 ID 阶段，所以 Load 类指令还是需要在其 ID 阶段给 rd 寄存器上锁，WB 阶段回写后解锁。ID 阶段遇到寄存器上锁就暂停指令执行，等到解锁了再重新读取寄存器的值。

# 3 重要调试

模拟器工程量不太大，但程序中数据处理比较容易出错，也不好调试，开发过程中经历了以下三个重要调试过程：

## 3.1 读取程序到内存

以字符串形式读入程序码，多次输出“内存”与读取的文件相比较调试。之后出现的很多运行错误其实是这个阶段出了一些问题。

## 3.2 ID 阶段解析立即数

有些指令的立即数存储方式比较奇怪，解析过程易错，且不易输出结果调试。整个 ID 阶段是否正确只能让程序完全跑起来后输出指令来和 Dump 文件比较才能发现。这步完成后基本版模拟器也就完成了。

## 3.3 并行结构的冲突调试

基本版到串行五级流水是比较容易的，但到并行就有一定的困难。各种冲突处理方案都需要大量调试。

幸好之前已经有正确的串行五级流水模拟器，可以输出每一步的寄存器内容进行比对调试，得以发现其中出现数据冲突、控制冲突的指令。

# 4 重要加速

用高级语言模拟低级语言的代价就是程序运行效率大幅下降，为了让运行时间在可控范围内，就需要尽量减少封装和复杂的结构，在整个开发过程中主要经历了三次重要加速。

## 4.1 代码重构：继承多态到选择语句

基本版模拟器中对每个指令开一个类，从基类继承来，均含有同名虚函数，在五级流水版中将这些子类删除，在基类函数中用 switch 语句处理，评测时间由 11s 降到 4s。

## 4.2 舍弃 stl 队列

刚开始写并行结构时没有多想直接用 stl 的队列装指令，评测超时后用 gprof 分析程序运行时间发现 queue 的调用时间较多，自写裸装队列后速度提高 3 倍。

## 4.3 O2 编译优化

这是个很好的编译优化项，使用后本机上运行速度提高 3 倍左右。到此为止停机并行和 forwarding 并行评测时间均为 5s 左右，得以通过。

# 5 总结

本项目最难在开头规划模拟器整体结构和具体汇编指令的学习上，因此基本版用时最长。此后的并行结构在调试和加速上会遇到一些困难，但有串行程序作为调试工具，可以解决。

制作模拟器最大的收获是了解了 CPU 执行汇编指令的过程，对计算机指令执行的低层结构有了一些了解。