计算机体系结构



MIPS流水线实验报告

|  |  |
| --- | --- |
| 学 院： | 计算机与信息技术学院 |
| 专 业： | 计算机科学与技术 |
| 指导教师： | 吴为民 |
| 班 级： | 计科1602 |
| 姓 名： | 麻锦涛 |
| 学 号： | 16281262 |

2019年 5月 1日

1. 实验目的：

熟悉 MIPS 流水线以及冒险（hazards）、前送(forwarding)、及调度

（scheduling）等概念和技术。

1. 实验内容及要求
2. 熟悉 winMIPS64 模拟器，并确定指令格式中各个域的具体值。包

括如下内容：

* 1. 将附件中的 winmips64.zip 文件解压到你的电脑中。
  2. 阅读附件中的winmipstut-6-3.3.docx文件,并按其中的步骤操作、学习 winMIPS64 模拟器。

1. 了解指令在 MIPS 流水线中的运行过程：
   1. 通过阅读理解程序 incre.s，指出该程序在运行时可能会出现哪些数据相关和控制相关。
   2. 将 incre.s 读入到模拟器中，关闭前送(forwarding)，单步运行程序，考察一个循环内程序的各个指令在各个周期的运行情况。对最下面的状态栏上指示的 RAW 和 Branch taken 冒险，解释其产生的原因。
2. 考察前送(forwarding)技术对流水线性能的影响：
   1. 在打开前送(forwarding)功能的情况下，单步运行 incre.s 并考察一个循环内程序的各个指令在各个周期的运行情况，解释前送是如何解决了很多冒险情况的。
   2. 指出哪些仍未消失的冒险为什么前送也无法解决。
3. 通过调度减少冒险：
   1. 考察 incre.s , 看看能否在不改变程序运行结果的情况下，通过改变指令的执行顺序来进一步减少冒险产生的 stall。要求：可以修改指令的操作数，但不能增加或减少指令。
   2. 解释你的做法为什么能减少冒险。
4. 总结：

对以上的三个步骤，分别考察统计信息（显示在 winMIPS64 的 statistics 窗口中）。通过列表的方式对本次实验做一个总结。

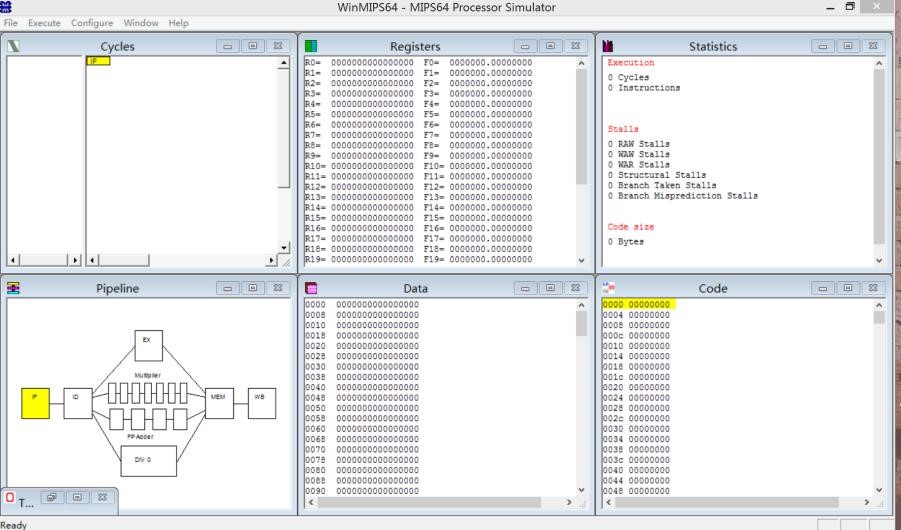
1. 实验过程

**3.1** 学习使用 **winmips64.exe**

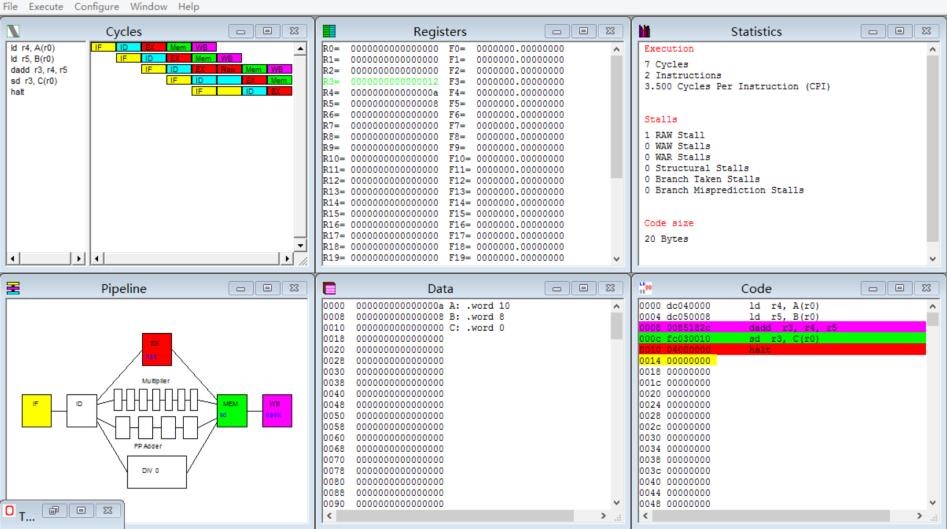
1. 将实验所给材料 lab3.rar 解压到电脑上，找到 winmips64.exe 并运行，模拟器界面如下图 3 所示：

 图 1

图 2

 图 3

1. 根据附件中的 winmipstut-6-3.3.docx 文件所提供的帮助，掌握了 winMIPS64 模拟器基本操作，图 4 为在模拟器中打开 sum.s 文件，单步运行的过程截图：

图 4

**3.2**了解指令在MIPS流水线中的运行过程

1. 程序incre.s的主要功能是：对于给定的数字序列，对前n个数字进行加1操作（程序中的n设置为6）。
2. 数据相关：流水线改变了对操作数读、写访问的顺序，导致与在非流水线机器上按顺序执行指令相比，对操作数读、写访问的顺序发生了改变。
3. 程序在运行时可能会出现的数据相关：
4. 指令“daddi r2, r0, number”与指令“daddi r3, r2, 24”

指令“daddi r2, r0, number”在WB阶段写寄存器r2的值，而指令“daddi r3, r2, 24”在ID阶段就需要读寄存器r2的值（此时对应前一条指令的EX阶段），因此如果指令“daddi r3, r2, 24”不停下来等待，就会读到错误的值，即为数据冒险。

1. 指令“lw r1, 0(r2)”与指令“daddi r1, r1, 1”

指令“lw r1, 0(r2)”在WB阶段写寄存器r1的值，而指令“daddi r1, r1, 1”在ID阶段就需要读寄存器r1的值（此时对应前一条指令的EX阶段），因此如果指令“daddi r1, r1, 1”不停下来等待，就会读到错误的值，即为数据冒险。

1. 指令“daddi r1, r1, 1”与指令“sw r1, 0(r2)”

指令“daddi r1, r1, 1”在WB阶段写寄存器r1的值，而指令“sw r1, 0(r2)”在ID阶段就需要读寄存器r1的值（此时对应前一条指令的EX阶段），因此如果指令“sw r1, 0(r2)”不停下来等待，就会读到错误的值，即为数据冒险。

1. 指令“daddi r2, r2, 4”与指令“dsub r4, r3, r2”

指令“daddi r2, r2, 4”在WB阶段写寄存器r2的值，而指令“dsub r4, r3, r2”在ID阶段就需要读寄存器r2的值（此时对应前一条指令的EX阶段），因此如果指令“dsub r4, r3, r2”不停下来等待，就会读到错误的值，即为数据冒险。

1. 指令“dsub r4, r3, r2”与指令“bnez r4, loop”

指令“dsub r4, r3, r2”在WB阶段写寄存器r4的值，而指令“bnez r4, loop”在ID阶段就需要读寄存器r4的值（此时对应前一条指令的EX阶段），因此如果指令“bnez r4, loop”不停下来等待，就会读到错误的值，即为数据冒险。

1. 控制相关：从微观的角度，在流水线处理器中，指令是并行处理的，在当前指令正在执行时，后面的很多条指令已经完成了取指和译码等步骤。然而，在程序中会存在很多的跳转语句，如果程序的实际执行路径是要跳转到其他的地址去执行，那么流水线中已经做的这些取指和译码工作就白做了，这就是流水线的控制冒险。
2. 程序在运行时可能会出现的控制相关：指令“bnez r4, loop”。

在前几次循环中，指令“bnez r4, loop”均转移成功，而在MEM阶段结束后PC的值才会更新为转移地址，因此该指令在ID、EX阶段时，后面的指令所进行的工作都白做了，即为控制冒险。

* 存在的数据相关的语句组 1：



这两个语句中的寄存器 r2 存在数据相关，第一条语句的 r2 为目的寄存器，相加的结果要存入寄存器 r2 中；第二条语句的 r2 为源寄存器，计算前需要先读取寄存器 r2 中的内容作为其中一个操作数。

* 存在的数据相关的语句组 2：



这两个语句中的寄存器 r1 存在数据相关，其中 load 指令中的 r1 为目的寄存器，它实现的是将内存地址为 0+r2 的内存单元中的内容读出，并存入寄存器 r1 中；addi 中的 r1 既是源寄存器又是目的寄存器，计算过程是先将寄存器r1中的内容读出作为一个操作数，最后再将运算结果重新存入寄存器 r1 中。

* 存在的数据相关的语句组 3：



这两个语句中的寄存器 r1 存在数据相关，addi 中的 r1 既是源寄存器又是目的寄存器，计算过程是先将寄存器 r1 中的内容读出作为一个操作数，最后再将运算结果重新存入寄存器 r1 中；sw 中 r1 为源操作数，它实现的是将寄存器 r1 中的内容读出存入到内存地址为 0+r2 的内存单元中。

* 存在的数据相关的语句组 4：



这两个语句中的寄存器 r2 存在数据相关，第一条语句中的 r2 既是源寄存器又是目的寄存器，计算过程是先将寄存器 r2 中的内容读出作为一个操作数，最后再将运算结果重新存入寄存器 r2 中； sub 中 r2 为源操作数，它实现的是将寄存器 r2 中的内容读出作为一个操作数，之后进行运算。

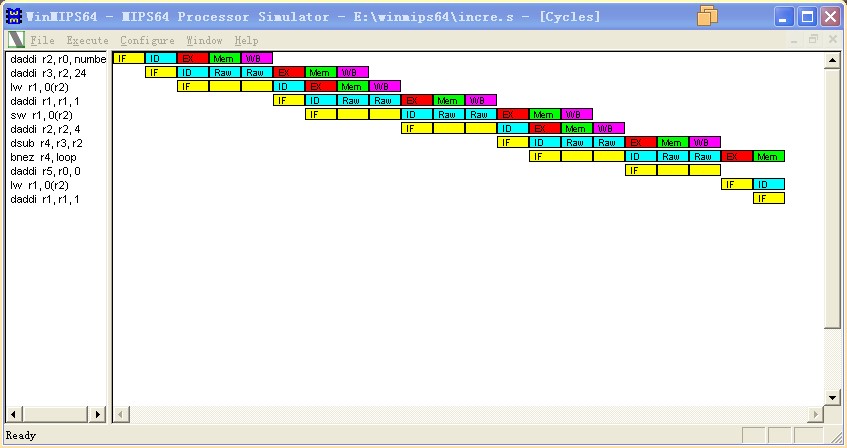
* 存在的数据相关和控制相关的语句组 5：



这两个语句中的寄存器 r4 存在数据相关，转移指令存在控制相关，第一条语句中的 r4 是目的寄存器，将运算结果存入寄存器 r4 中；条件转移语句中 r4 为源操作数，它根据寄存器 r4 中的内容是否等于 0 来决定是否执行条件转移。

**3.3** 关闭前送时程序运行情况

关闭前送时，第一个循环内程序的各个指令在各个周期的运行情况如下图：

程序运行中出现的冒险及其产生原因：

1. 语句组 

状态栏中的指示为：Raw stall in ID(r2)。因为第一条语句在 WB 阶段才能计算结果写会到寄存器 r2 中，而第二条语句则在 ID 阶段就需要读取寄存器 r2 中的内容作为运算的一个操作数，因此会出现 RAW 冒险，第二条语句会在 ID 阶段等待 2 个 stall.

1. 语句组 

状态栏中的指示为：Raw stall in ID(r1)。因为第一条语句在 WB 阶段才能将 LMD 中的内容写会寄存器 r1 中，而第二条语句则在 ID 阶段就需要读取寄存器 r1 中的内容作为运算的一个操作数，因此会出现 RAW 冒险，第二条语句会在 ID 阶段等待 2 个 stall.

1. 语句组 

状态栏中的指示为：Raw stall in ID(r1)。同样因为第一条语句 addi 在经过 WB 后才把运算结果存到 r1 寄存器，sw 需要在 ID 阶段获取 r1 中的数据。所以会出现 RAW 冒险，因为整数流水线的周期性质，所以后者会在 ID 阶段等待两个 stall.

1. 语句组 

状态栏中的指示为：Raw stall in ID(r2)。因为前一条语句在 WB 阶段才写入 r2 寄存器，sub 的 ID 阶段就需要获得 r2 的值。因此会出现 RAW 冒险，第二条语句会在 ID 阶段等待 2 个 stall.

1. 语句组 

状态栏中出现的指示为：Raw stall in ID(r1)和 Branch taken stall.

因为 sub 语句在 WB 阶段才能将运算结果写入寄存器 r4 中，转移指令则在 ID 阶段需要读取寄存器 r4 中的值，在 EX 阶段计算 Cond 是否为 0，因此会出现 RAW 冒险，条件转移语句会在 ID 阶段等

待 2 个 stall.在转移指令运行到 EX 阶段才得出 Cond != 0 的结论，因此是需要跳转的，所以需要重新取值，即出现

**3.4** 打开前送时程序运行情况

打开前送时，第一个循环内程序的各个指令在各个周期的运行情况如下图：



打开前送后

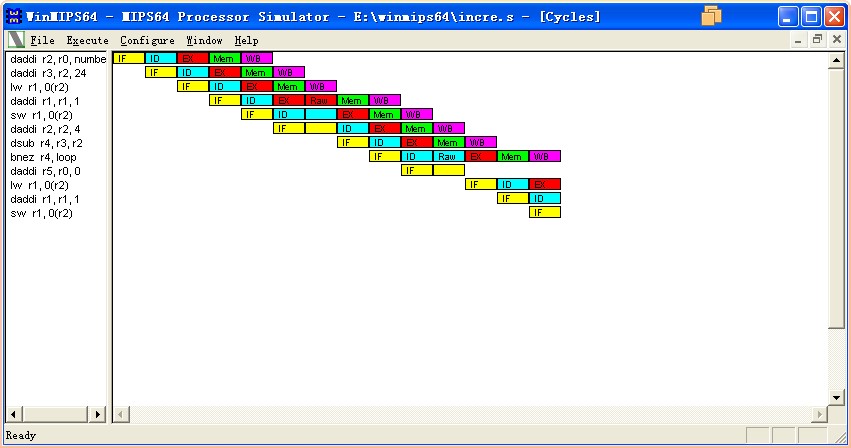
，解决了

一部分

RAW

类型

冒险，原因如下：



* + - 冒险组 1：

使用 forwarding 技术，使得前一条语句在 EX 阶段产生的数据直接送到下个指令的 EX 阶段，因此后一条语句不用等到 WB 阶段再运行，消除了 RAW 造成的 stall.

* + - 冒险组 3：原因与冒险组 1 相似。
    - 冒险组4：原因与冒险组1相似。

 程序中通过打开前送仍然没有解决的冒险有：

* + - 冒险组 2：

状态栏冒险提示： Load 指令产生的可用数据在 MEM 阶段，而后一条语句在 EX 阶段是在前一条语句的 MEM 之前，所以需要等待一个周期，会产生一个 stall.

* + - 冒险组 5：

状态栏冒险提示：

和



RAW 冒险无法解决的原因与冒险组 2 类似。而转移指令所造成的冒险是无法通过前送彻底解决的原因是，通过前送，最早可以在 EX 阶段可以将转移地址送给 PC，由 PC 指向的地址取出转移到的第一条指令，然而，后一条指令的 IF 阶段是在跳转语句的 EX 阶段之前，所以必然会造成转移成功需要重新取指的局面。

**3.5** 通过调度减少冒险

修改后 incre.s 的内容如下：

.data

number: .word32 1, 2, 3, 4, 5, 6, 7, 8, 9

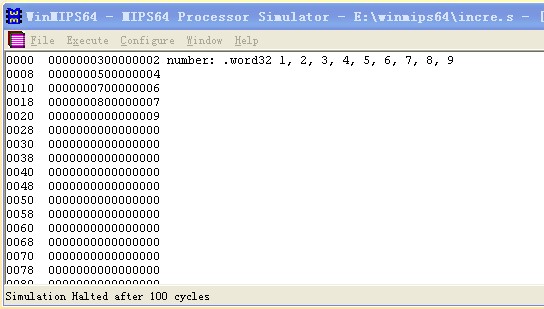
.text main:

daddi r2, r0, number daddi r3, r2, 24 loop: lw r1, 0(r2) daddi r2, r2, 4 daddi r1, r1, 1 dsub r4, r3, r2 sw r1, -4(r2) bnez r4, loop

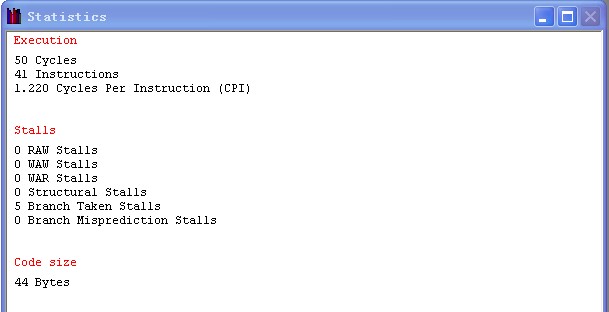
daddi r5, r0, 0 daddi r6, r0, 0 halt

调度后的程序的运行情况：

 运行结果与未调度之前的相同：



#  除了 Branch Taken stall 外，其余的 RAW 型冒险已经全部消除：

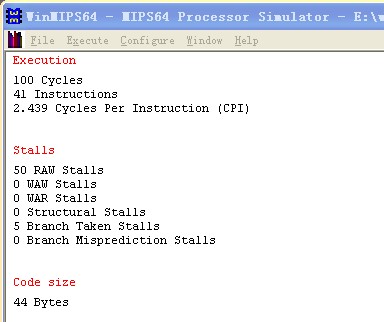


之所以可以减少冒险，是因为通过适当的调度策略，减少了资源冲突的机会，减少程序在同一时间内访问相同单元的可能，例如: 在 dsub r4, r3, r2 和 bnez r4, loop 两条语句中插入 sw r1, -4(r2)语句，就可以使程度对于寄存器 r4 的访问不再发生冲突，那么也就消除了冒险。

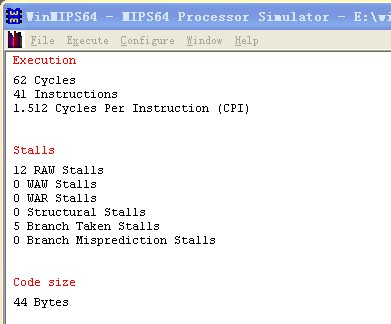
当然这种调度的前提一定是在不改变程序功能和结果的情况。

**3.6** 总结

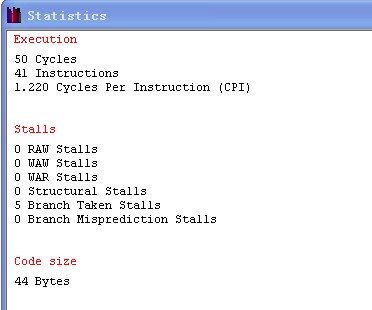
* 关闭前送：



* 打开前送：



* 通过调度减少冒险：

 对比三个步骤，信息总结如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 关闭前送 | 打开前送 | 通过调度减少冒险 |
| Cycle | 100 | 62 | 50 |
| Instruction | 41 | 41 | 41 |
| CPI | 2439 | 1512 | 1220 |
| stall | 50 RAW stalls  5 Branch taken stalls | 12 RAW stalls  5 Branch taken stalls | 0 RAW stall  5 Branch taken stalls |
| Code size | 44 bytes | 44 bytes | 44 bytes |

通过三个步骤的对比，可以发现，通过前送可以大大减少 RAW 冒险，进而大大降低了 CPI。之后通过调度，更进一步减少了冒险。可以发现有了前送技术再加上适当的调度策略，与关闭前送时对比，CPI 显著 降低，有效的提高了流水线效率。