山东大学 计算机科学与技术 学院

计算机体系结构 课程实验报告

学号：201900130133	姓名：施政良	班级： 四班
实验题目： 指令调度		
实验学时：2	实验日期： 2022-05-24	

实验目的：

（1） 通过本实验，加深对指令调度的理解

（2） 了解指令调度技术对 CPU 性能改进 的好处

硬件环境：

WinDLX (一个基于 Windows 的 DLX 模拟器)

软件环境：

Windows 7

实验步骤与内容：

实验内容

本次实验主要涉及指令调度,具体的实验步骤可以划分为如下几个步骤

（1） 通过 Configuration 菜单中的“Floating point stages”选项，把除法单 元数设置为 3，把加法、乘法、除法的延迟设置为 3 个时钟周期。

（2） 用 WinDLX 模拟器运行调度前的程序 sch-before.s 。记录程序执行过程中各种相关发生的次数以及程序执行的总时钟周期数。

（3） 用 WinDLX 模拟器运行调度后的程序 sch-after.s ，记录程序执行过 程中各种相关发生的次数以及程序执行的总时钟周期数。

（4） 根据记录结果，比较调度前和调度后的性能。

（5） 论述指令调度对于提高 CPU 性能的意义。

具体实验过程如下所示

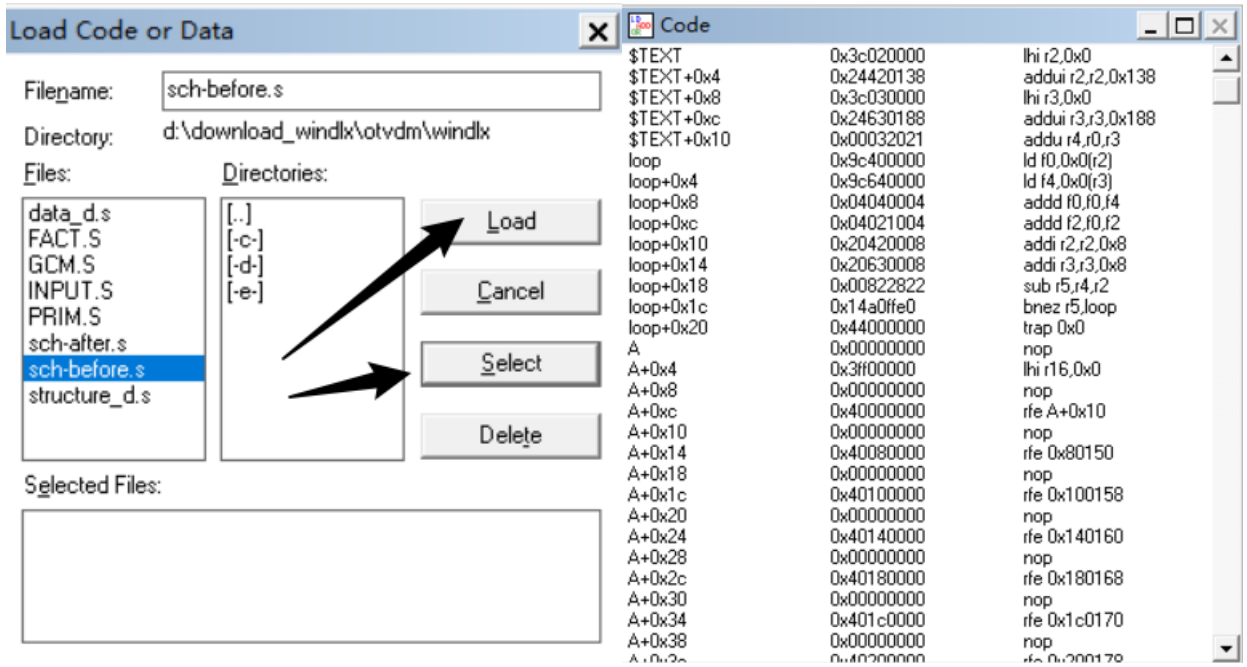
具体实验过程

1. 运行 sch-before.s 文件

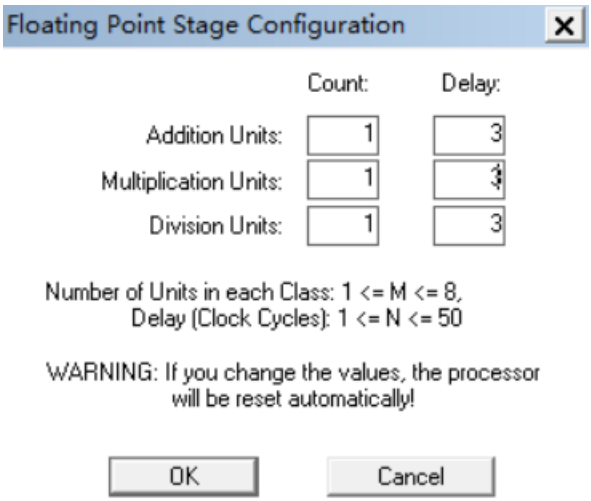
首先运行 sch-before.s 文件，观察指令调度之前，指令流水线的执行情况。

（1） 重置 WinDLX 模拟器并导入对应的文件。到入之后点击 code 窗口可以看到对

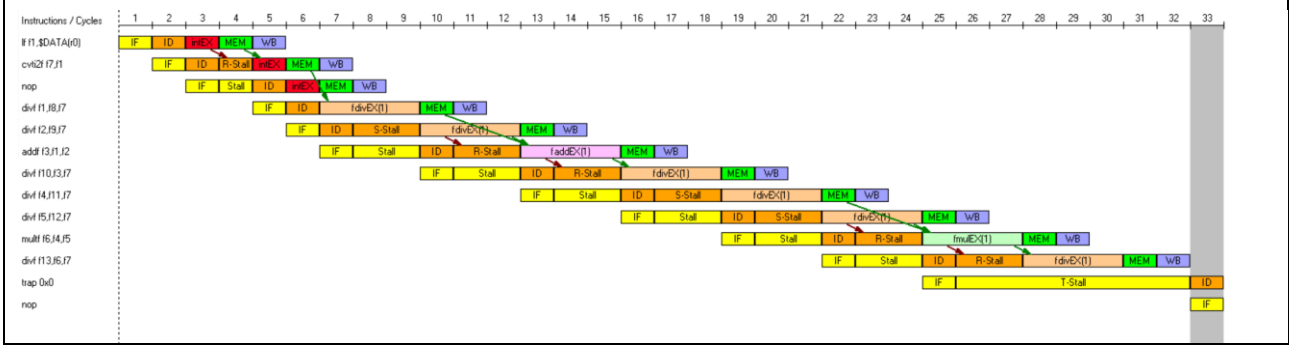
应的代码



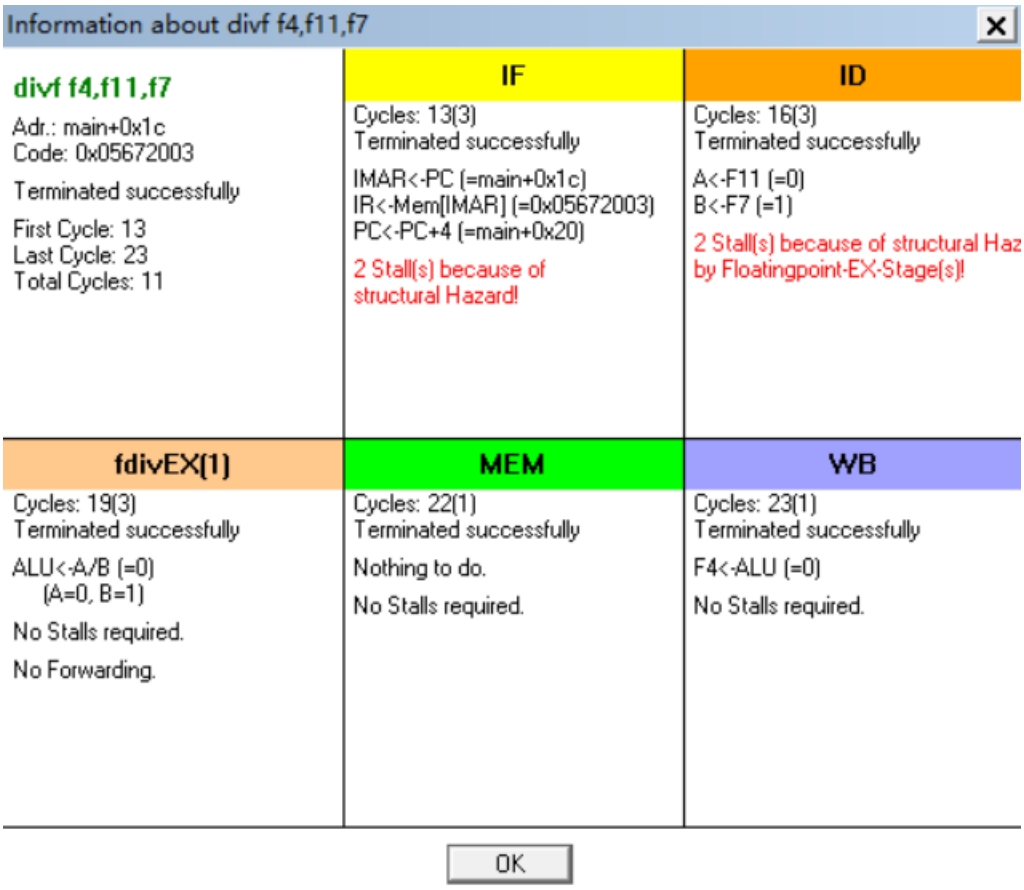
（2） 通过 Configuration 菜单中的“Floating point stages”选项，把除法单元数设 置为 3，把加法、乘法、除法的延迟设置为 3 个时钟周期



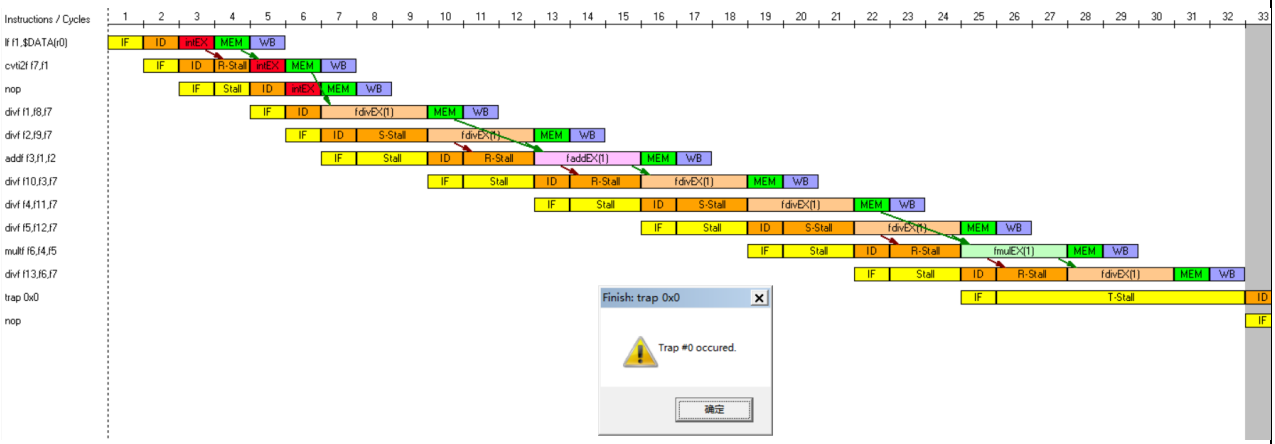
（3） 单步跟踪程序的执行，观察指令执行时各个功能部件的指令流水。



可以看到，上图中红色的箭头说明出现了结构相关和数据相关。例如指令 $div\ f_1\ f_0\ f_7$在译码后由于上一条指令占用了执行部件，因此该指令被阻塞在译码阶段，如下图所示：



（4） 按下 F5, 连续执行指令直到程序结束，如下图所示



可以看到，sch-before 程序在执行时共花费了 33 个指令周期。之后点击 statistics 窗口，观察指令执行过程中的统计信息

```
Total:
    33 Cycle(s) executed.
    ID executed by 12 Instruction(s).
    2 Instruction(s) currently in Pipeline.

Hardware configuration:
    Memory size: 32768 Bytes
    faddEX-Stages: 1, required Cycles: 3
    fmulEX-Stages: 1, required Cycles: 3
    fdivEX-Stages: 1, required Cycles: 3
    Forwarding enabled.

Stalls:
    RAW stalls: 9 (27.27% of all Cycles), thereof:
        LD stalls: 1 (11.11% of RAW stalls)
        Branch/Jump stalls: 0 (0.00% of RAW stalls)
        Floating point stalls: 8 (100.00% of RAW stalls)
    WAW stalls: 0 (0.00% of all Cycles)
    Structural stalls: 6 (18.18% of all Cycles)
    Control stalls: 0 (0.00% of all Cycles)
    Trap stalls: 7 (21.21% of all Cycles)
    Total: 22 Stall(s) (66.67% of all Cycles)

Conditional Branches):
    Total: 0 (0.00% of all Instructions), thereof:
        taken: 0 (0.00% of all cond. Branches)
        not taken: 0 (0.00% of all cond. Branches)

Load-/Store-Instructions:
    Total: 1 (8.33% of all Instructions), thereof:
        Loads: 1 (100.00% of Load-/Store-Instructions)
        Stores: 0 (0.00% of Load-/Store-Instructions)

Floating point stage instructions:
    Total: 8 (66.67% of all Instructions), thereof:
        Additions: 1 (12.50% of Floating point stage inst.)
        Multiplications: 1 (12.50% of Floating point stage inst.)
        Divisions: 6 (75.00% of Floating point stage inst.)

Traps:
    Traps: 1 (8.33% of all Instructions)
```

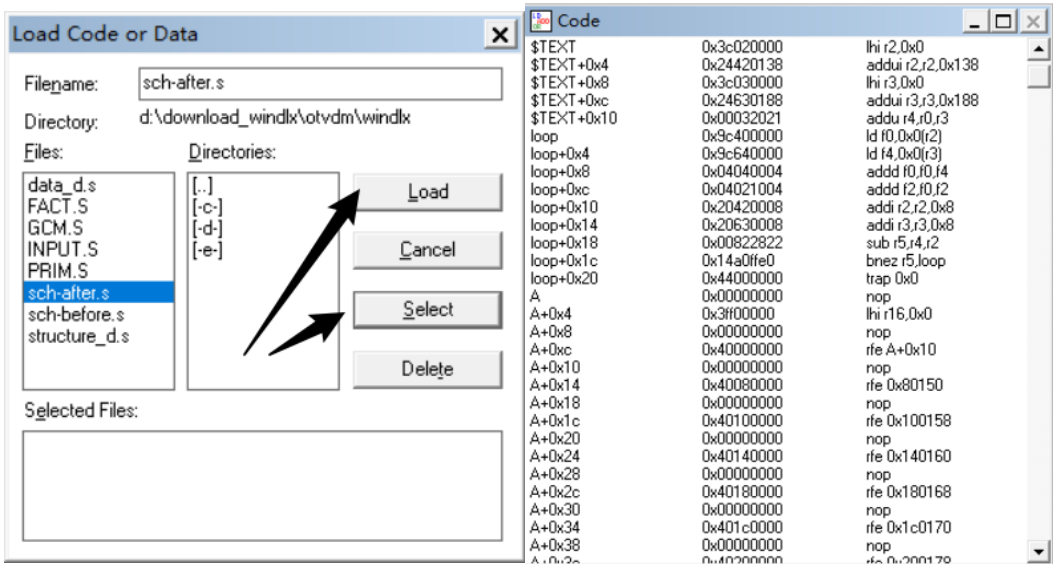
通过上图可知，程序执行过程中的流水线冒险涉及到数据相关和结构相关。

1. 先写后读 (RAW)产生的数据相关次数共 9 次，其中有 8 次为浮点数运算所导致。
2. 结构相关发生次数为 6 次
3. 因 Trap 暂停导致的流水线暂停周期数为 7
4. 这个程序的流水线周期暂停数量为 22

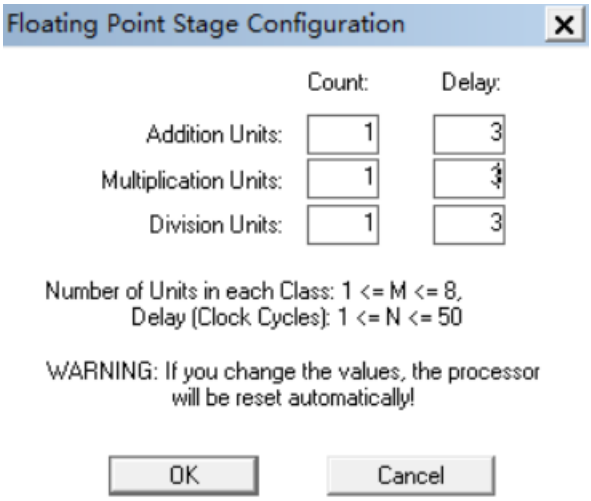
2. 运行 sch-after.s 文件

运行调度后的程序 sch-after.s ，记录程序执行过程中各种相关发生的次数以及程 序执行的总时钟周期数

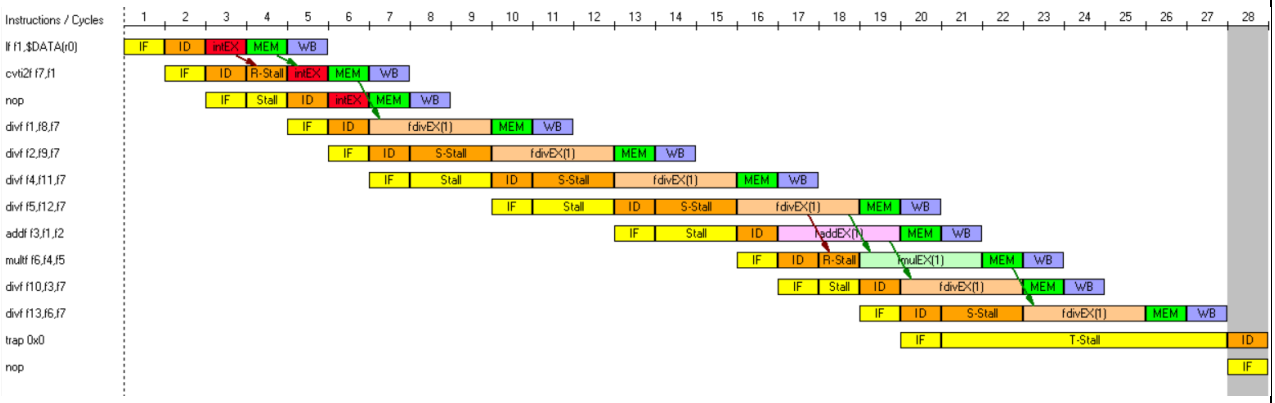
（1） 重置 WinDLX 模拟器并导入对应的文件。到入之后点击 code 窗口可以看到对应的代码



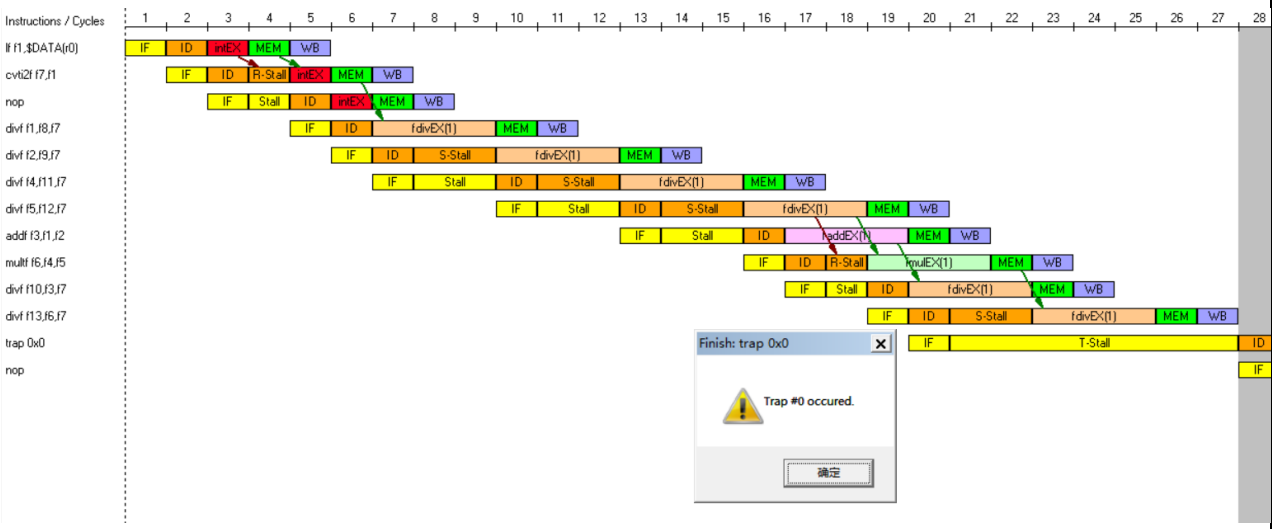
（5） 通过 Configuration 菜单中的“Floating point stages”选项，把除法单元数设 置为 3，把加法、乘法、除法的延迟设置为 3 个时钟周期



（6） 单步跟踪程序的执行，观察指令执行时各个功能部件的指令流水。

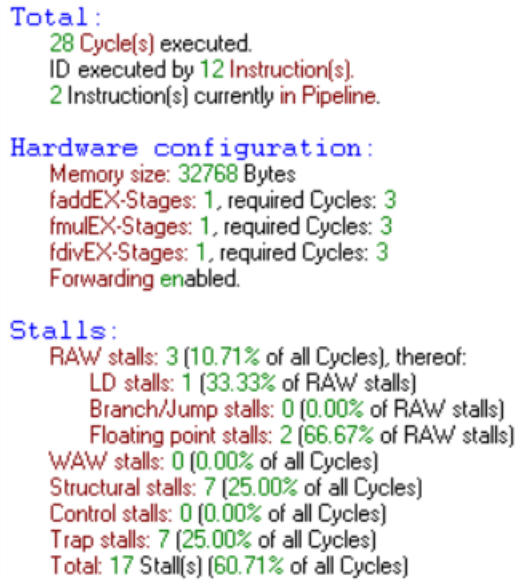


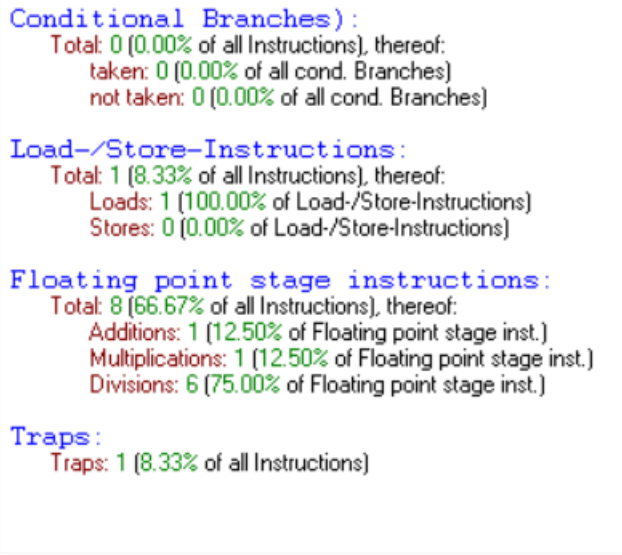
（7） 按下 F5,连续执行指令直到程序结束，如下图所示



可以看到，sch-after 程序在执行时共花费了 28 个指令周期。

之后点击 statistics 窗口，观察指令执行过程中的统计信息





通过上图可知，程序执行过程中的流水线冒险涉及到数据相关和结构相关。

1. 先写后读 (RAW)产生的数据相关次数共 3 次，其中有 2 次为浮点数运算所导致。
2. 结构相关发生次数为 7 次
3. 因 Trap 暂停导致的流水线暂停周期数为 7
4. 这个程序的流水线周期暂停数量为 17

结论分析与体会：

结论分析

1. 指令调度的分析

分析：

通过实验可知，将指令的执行顺序进行适当的调度可以减少数据相关和结构相关导致的流水线的断流。从汇编代码的角度进行分析，分析指令的调整情况。如下图所示（左图为调整前，右图为调整后）

```
.data
.global ONE
ONE: .word 1
.text
.global main
main:
lf f1,ONE ;turn divf into a move
cvti2f f7,f1 ;by storing in f7 1 in
nop ;floating-point format
divf f1,f8,f7 ;move Y=(f8) into f1
divf f2,f9,f7 ;move Z=(f9) into f2
addf f3,f1,f2
divf f10,f3,f7 ;move f3 into X=(f10)
divf f4,f11,f7 ;move B=(f11) into f4
divf f5,f12,f7 ;move C=(f12) into f5
multf f6,f4,f5
divf f13,f6,f7 ;move f6 into A=(f13)
Finish:
trap 0
```

图 1 调整前汇编指令

```
lf f1,ONE ;turn divf into a move
cvti2f f7,f1 ;by storing in f7 1 in
nop ;floating-point format
divf f1,f8,f7 ;move Y=(f8) into f1
divf f2,f9,f7 ;move Z=(f9) into f2
divf f4,f11,f7 ;move B=(f11) into f4
divf f5,f12,f7 ;move C=(f12) into f5
addf f3,f1,f2
multf f6,f4,f5
divf f10,f3,f7 ;move f3 into X=(f10)
divf f13,f6,f7 ;move f6 into A=(f13)
Finish:
trap 0
```

图 2 调整后汇编指令

对比上述代码可以清晰的发现，调整后的汇编代码中将具有数据相关的指令进行了分离，例如向原来代码中三条连续执行的指令（如下表一）中插入了三条除法指令（如下表二），且新插入的除法指令在数据上与之前的指令不存在读写冲突。由于数据之间不存在相关关系，因此在指令执行时无需等待上一条除法指令执行完毕即可继续向下执行，避免了流水线的断流，提高了指令的执行效率。

表一 原始代码执行顺序

```
divf f1,f8,f7 ;move Y=(f8) into f1
divf f2,f9,f7 ;move Z=(f9) into f2
addf f3,f1,f2
```

表格 2 插入不相关指令后的代码（红色为插入的指令）

```
divf f1,f8,f7 ;move Y=(f8) into f1
divf f2,f9,f7 ;move Z=(f9) into f2
divf f4,f11,f7 ;move B=(f11) into f4
divf f5,f12,f7 ;move C=(f12) into f5
addf f3,f1,f2
```

2. 调度前后指令执行的定量分析

分析：

通过对比两次程序的运行结果可知，程序 sch-before 和 sch-after 执行的功能相同，但是由于指令执行的顺序不同，在具体实行过程中各个功能部件的流水线执行情况不同。

具体来说，sch-after 将指令进行调度，在相关性较强的指令之间插入额外的指令，从而减少了流水线冒险，缓解了流水线的断流。

通过定量计算，指令调度之前程序执行的总时钟周期数为 32，吞吐率为 10/32=0.30，指令调度之后程序执行的总时钟周期数为 28，吞吐率为 10/28=0.35，提高性能的倍数为 $\frac{0.35}{0.30} = 1.16.$

3. 论述指令调度对于提高 CPU 性能的意义（对流水线相关的理解）

分析：

由于相关的存在，使得指令中的下一条指令不能在指定的时钟周期执行。流水线冲突 会给指令在流水线中的执行带来很多问题，如果不能很好的解决冲突，轻则影响流水线的 性能，重则导致错误的执行结果。对于各种冲突，如果可以很好的利用指令调度，就会得 到很好的解决。 在使用调度之后不管是结构相关还是数据相关，发生的次数都明显减少很多，而且程 序执行的总周期数目也大大的减少了。编译器通过重新组织代码的顺序来实现“指令调度”， 消除了部分暂停周期，暂停周期数目的减少有助于提高流水线的性能，使得流水线能够更 加高效的完成工作。这样极大的减少了暂停周期的数目，从而减少了执行周期数目，使得 CPU 工作更加高效，提高了系统的性能。

体会

本次实验主要涉及了指令调度的基本内容。在实验中，通过对比执行调度前后各个功能部件的执行情况，我分析了指令执行顺序对程序执行性能的影响。从本质上来分析，数据相关和结构相关都是由于上一条指令延迟占用功能部件，导致本条指令无法正常的进入指令流水，从而导致流水线断流。而具体来分析，结构相关强调当硬件资源满足不了指令重叠执行的要求，而发生资源冲突时，就发生了结构相关；数据相关是指当一条指令需要用到前面指令的执行结果，而这些指令均在流水

线中重叠执行时，因此引起数据相关；控制相关主要发生在分支指令。

通过实验可以发现，一旦流水线中出现相关，必然会给指令在流水线中的顺利执行带来许多问题，如果不能很好 地解决相关问题，轻则影响流水线的性能，重则导致错误的执行结果。消除相关的基本方法是让 流水线暂停执行某些指令，而继续执行其它一些指令。

附录

sch-before.s

```
;---------------------------------------------------------------------
; Example to illustrate instruction scheduling
;---------------------------------------------------------------------
        .data
        .global     ONE
ONE:    .word       1
        .text
        .global main
main:
        lf          f1,ONE      ;turn divf into a move
        cvti2f      f7,f1       ;by storing in f7 1 in
        nop                     ;floating-point format
        divf        f1,f8,f7    ;move Y=(f8) into f1
        divf        f2,f9,f7    ;move Z=(f9) into f2
        addf        f3,f1,f2
        divf        f10,f3,f7   ;move f3 into X=(f10)
        divf        f4,f11,f7   ;move B=(f11) into f4
        divf        f5,f12,f7   ;move C=(f12) into f5
        multf       f6,f4,f5
        divf        f13,f6,f7   ;move f6 into A=(f13)
Finish:
        trap        0
```

sch-after.s

```
1. ;-----------------------------------------------------------------
2. ; Example to illustrate instruction scheduling - reordered instructions
3. ;-----------------------------------------------------------------
4.          .data
5.          .global     ONE
6. ONE:     .word       1
7.          .text
8.          .global main
9. main:
10.         lf          f1,ONE      ;turn divf into a move
11.         cvti2f      f7,f1       ;by storing in f7 1 in
12.         nop                     ;floating-point format
13.         divf        f1,f8,f7    ;move Y=(f8) into f1
14.         divf        f2,f9,f7    ;move Z=(f9) into f2
15.         divf        f4,f11,f7   ;move B=(f11) into f4
16.         divf        f5,f12,f7   ;move C=(f12) into f5
17.         addf        f3,f1,f2
18.         multf       f6,f4,f5
19.         divf        f10,f3,f7   ;move f3 into X=(f10)
20.         divf        f13,f6,f7   ;move f6 into A=(f13)
21.Finish:
22.         trap        0
```