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：	姓名：	班级：
实验题目：数据相关		
实验学时：2	实验日期：2024.6.7	

实验目的：

通过本实验，加深对指令调度的理解，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：

VMware Workstation 16 Player

Windows 7

实验步骤与内容：

实验内容：

（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。

实验步骤：

1. 阅读汇编程序，比较 sch-before.s 与 sch-after.s 的区别

sch-before.s 如下：

```
.data
.global ONE
ONE: .word 1

.text
.global main
main:
    lf f1,ONE          ; 将整数1加载到浮点寄存器f1中
    cvti2f f7,f1       ; 将f1中的整数值转换为浮点数并存储到f7中
    nop                ; 无操作，占位符
    divf f1,f8,f7      ; 将f8寄存器中的值除以f7（即1），结果存入f1
    divf f2,f9,f7      ; 将f9寄存器中的值除以f7（即1），结果存入f2
    addf f3,f1,f2      ; 将f1和f2中的值相加，结果存入f3
```

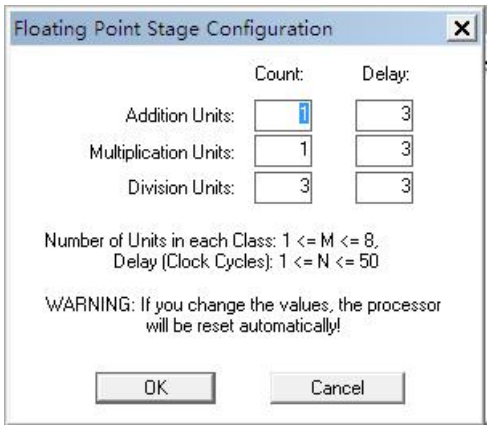
```
    divf f10,f3,f7    ; 将f3寄存器中的值除以f7（即1），结果存入f10
    divf f4,f11,f7    ; 将f11寄存器中的值除以f7（即1），结果存入f4
    divf f5,f12,f7    ; 将f12寄存器中的值除以f7（即1），结果存入f5
    multf f6,f4,f5    ; 将f4和f5中的值相乘，结果存入f6
    divf f13,f6,f7    ; 将f6寄存器中的值除以f7（即1），结果存入f13
Finish:
    trap 0            ; 程序结束
```

sch-after.s 如下：

```
.data
.global ONE
ONE: .word 1
.text
.global main
main:
51
    lf f1,ONE         ; 将整数1加载到浮点寄存器f1中
    cvti2f f7,f1      ; 将f1中的整数值转换为浮点数并存储到f7中
    nop               ; 无操作，占位符
    divf f1,f8,f7     ; 将f8寄存器中的值除以f7（即1），结果存入f1
    divf f2,f9,f7     ; 将f9寄存器中的值除以f7（即1），结果存入f2
    divf f4,f11,f7    ; 将f11寄存器中的值除以f7（即1），结果存入f4
    divf f5,f12,f7    ; 将f12寄存器中的值除以f7（即1），结果存入f5
    addf f3,f1,f2     ; 将f1和f2中的值相加，结果存入f3
    multf f6,f4,f5    ; 将f4和f5中的值相乘，结果存入f6
    divf f10,f3,f7    ; 将f3寄存器中的值除以f7（即1），结果存入f10
    divf f13,f6,f7    ; 将f6寄存器中的值除以f7（即1），结果存入f13
Finish:
    trap 0            ; 程序结束
```

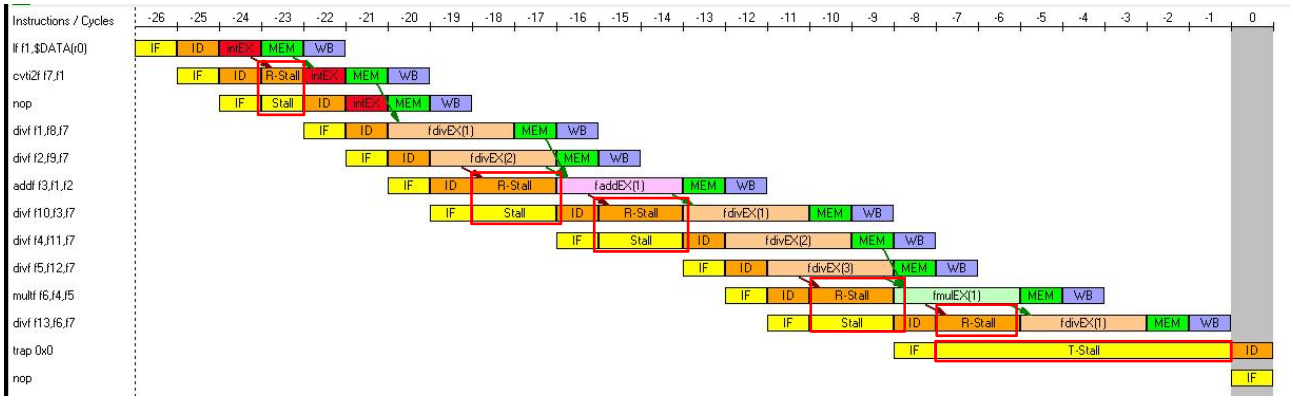
其中这三条指令顺序发生了变化

2. 设置加法、乘法、除法延迟，执行程序 sch-before.s



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。

通过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，如下图所示：



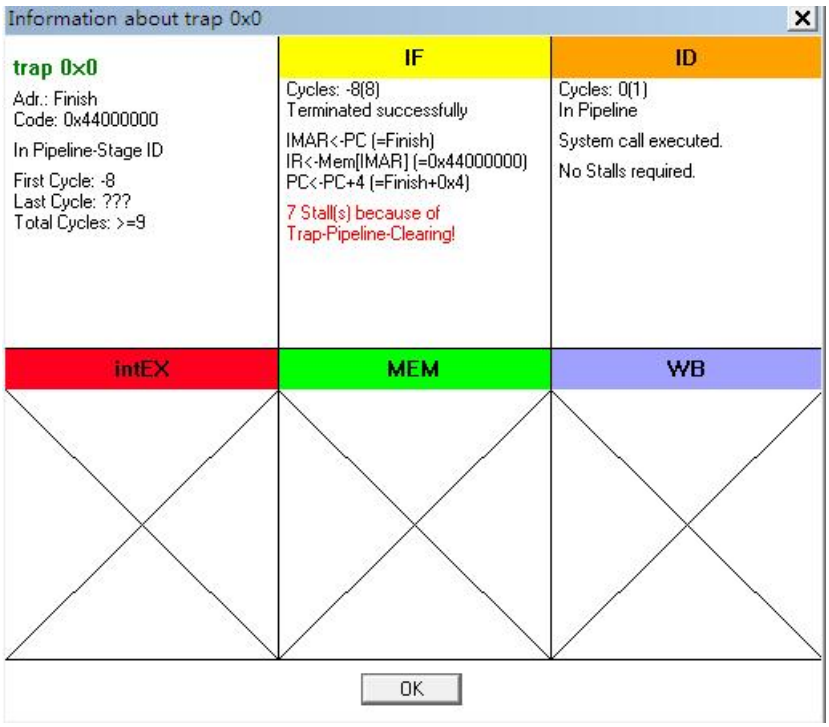
可以看到的是,在上图中出现了结构相关和数据相关,例如指令 lf f1,$DATA[0]与 cvti2f f7, f1 之间就存在数据相关，所以产生了一个 R-Stall。而由于指令 cvti2f f7，f1 在译码阶段停留了一个周期，故与下一条指令 nop 就产生了结构相关。

继续往后看，在指令 divf f2，f9，f7 与指令 addf f3，f1，f2 之间出现了数据相关，因为指令 addf f3，f1，f2 需要 f2 的值，而此时 f2 的值尚未计算得到 ，所以此处产生 R-Stall。而因为指令 addf f3，f1，f2 在译码阶段停留了一个 Stall，导致与下一条指令 divf f10，f3，f7 又产生了结构相关。

同时指令 divf f10，f3，f7 与上一条指令还有数据相关，因此该指令会在译码阶段停了两个 Stall，又导致与下一条指令 divf f4，f11，f7 出现结构相关。

继续单步执行，由于指令 multf f6,f4,f5 与指令 divf f5,f12,f7 关于寄存器 f5 产生了数据相关,所以产生了一个R-Stall。而由于指令divf f5,f12,f7 译码阶段产生了一个R-Stall，所以与下一条指令 divf f13,f6,f7 产生了结构相关。

最后由于指令 divf f13，f6，f7 需要上一条指令计算得到的 f6 的值，所以当上一条指令在执行时，该指令停留在译码阶段，也就导致了与 trap 指令产生了 T-Stall。



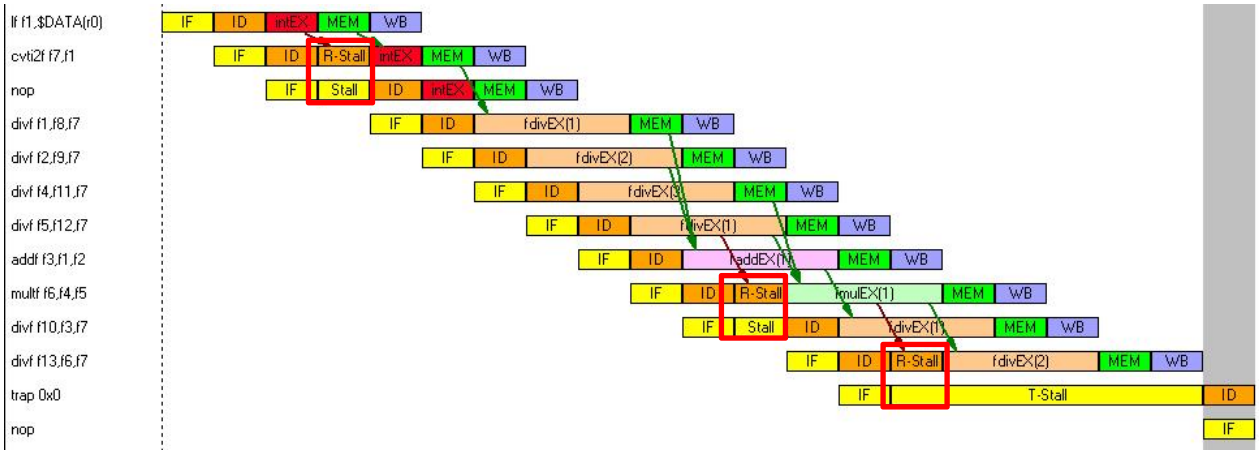
则整个程序执行过程中，一共发生 5 次数据相关、5 次结构相关。

3. 查看 Statistics 窗口

```
Total:
    27 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: 3, required Cycles: 3
    Forwarding enabled.

Stalls:
    RAW stalls: 9 (33.33% 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: 0 (0.00% of all Cycles)
    Control stalls: 0 (0.00% of all Cycles)
    Trap stalls: 7 (25.92% of all Cycles)
    Total: 16 Stall(s) (59.26% 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)
```

总执行时钟周期数为 27

4. 设置加法、乘法、除法延迟，执行程序 sch-after.s

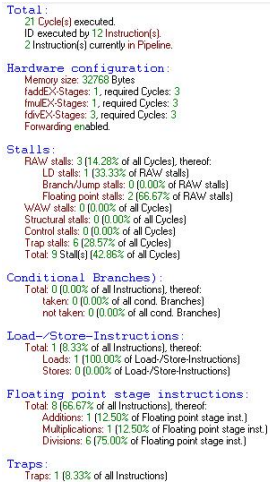


可以看到的是，一开始在指令 lf f1，$DATA[0]与 cvti2f f7，f1 之间存在的数据相关，以及因此导致指令 cvti2f f7，f1 与 nop 之间产生的结构相关，仍然存在。

接着往后看，指令 divf f5，f12，f7 与指令 multf f6，f4，f5 之间存在数据相关，因为浮点数除法需要三个周期，而指令 multf f6，f4，f5 需要 f5 的值，故该指令会产生一个 R-Stall。又因为指令 multf f6，f4，f5 在译码部件产生了一个 R-Stall，因此导致指令 divf f10，f3，f7 出现了结构相关。同理指令 multf f6，f4，f5 之间也存在数据相关，而指令 divf f13，f6，f7 也与其下一条指令存在结构相关。

故则整个程序执行过程中，一共发生 3 次数据相关、3 次结构相关。

5. 查看对应 Statistics 窗口



总执行时钟周期数为 21

6. 比较调度前和调度后的性能

在程序调度前所需的时钟周期数为 27，调度后所需时钟周期数为 21，性能提升到原来的

$\frac{27}{21}$ = 1.286 倍

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

通过对比可以发现，调整后的程序将会导致数据相关的指令往后调整。例如向原来代码中三条连续执行的指令（如下表一）中插入了两条除法指令（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数据相关，因此在指令执行时无需等待，这样就避免了流水线的断流，也能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.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：

由于数据、结构、控制相关的存在，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，如果我们对指令调度进行优化，问题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，而且程序执行的总时钟周期数也大大的减少了。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：

本次实验中主要涉及到指令调度的基本内容。在实验中，通过对比执行调度前后各条指令的执行情况，分析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；而控制相关则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。