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：202222130195	姓名：阎发祥	班级：22.3班
实验题目：指令调度		
实验学时：2	实验日期：2025.5.21	

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 ;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
```

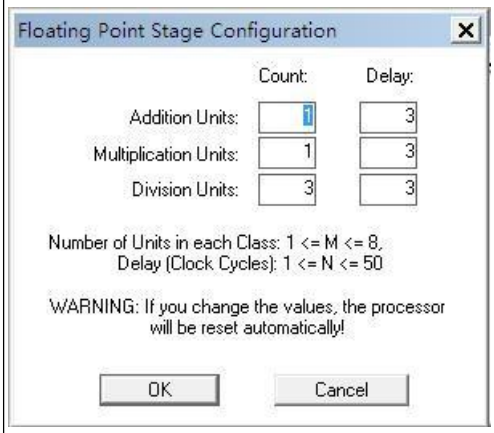
```
divf f10,f3,f7 ;move f3 into X=(f10)
divf f4,f11,f7 ;move B=(f11) into f4
divf f5,f12,f7 ;move C=(f12) into f5
multf f6,f4,f5
divf f13,f6,f7 ;move f6 into A=(f13)
Finish:
trap 0
```

sch-after.s 如下：

```
.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
    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
```

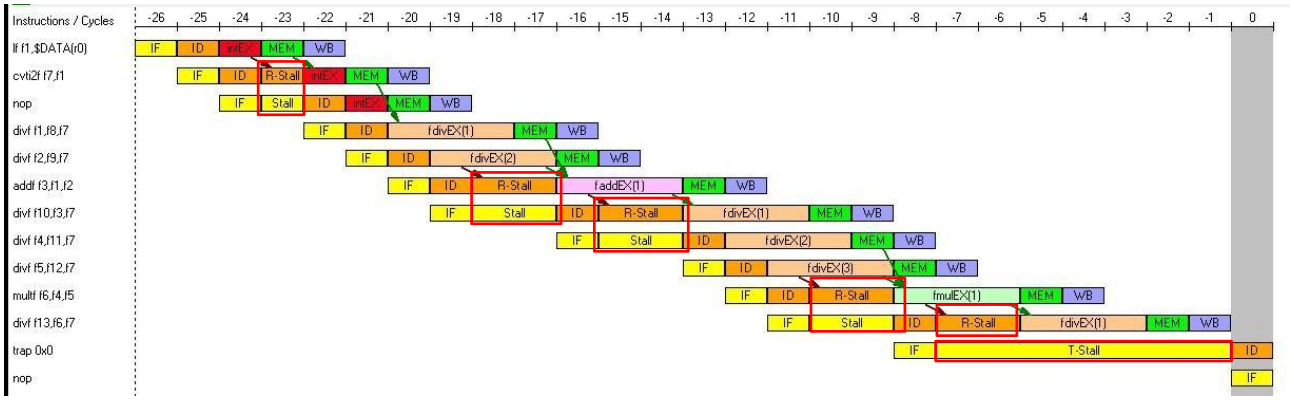
其中这三条指令顺序发生了变化

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：



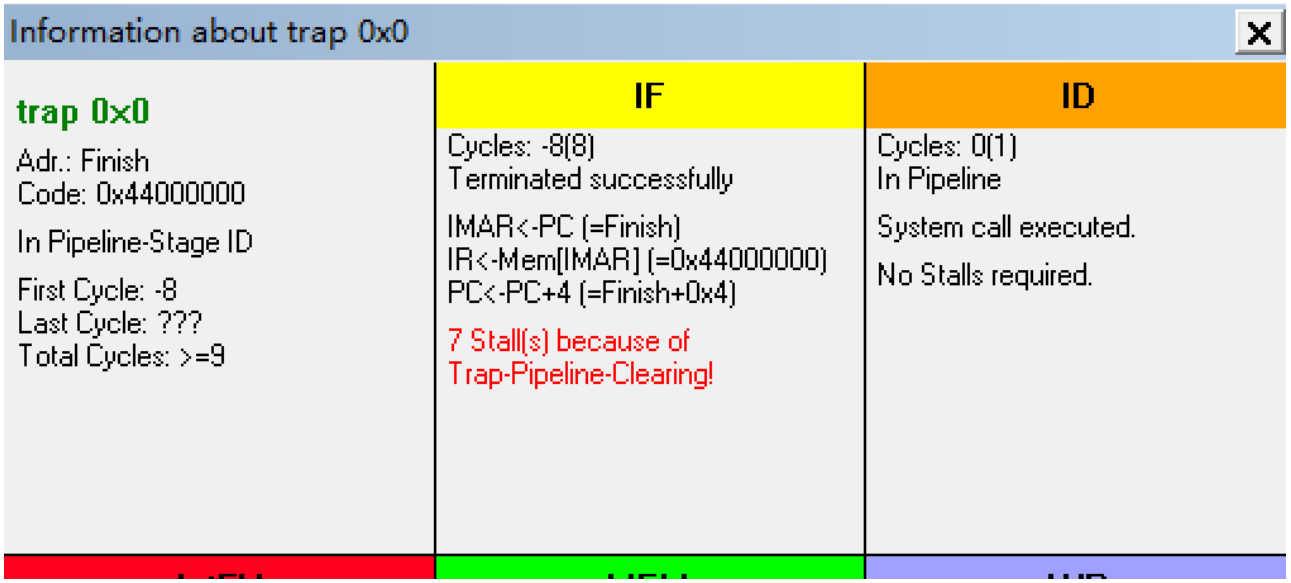
可以看到的是，在上图中出现了结构相关和数据相关，例如指令 lf f1,$DATA[0]与 cvti2f f7, f1 之间就存在数据相关，所以产生了一个 R-Stall。而由于指令 cvti2f f7, f1 在译码阶段停留了一个周期，故与下一条指令 nop 就产生了结构相关。

继续向后，指令 divf f2, f9, f7 与指令 addf f3, f1, f2 之间出现了数据相关，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。查看trap指令：



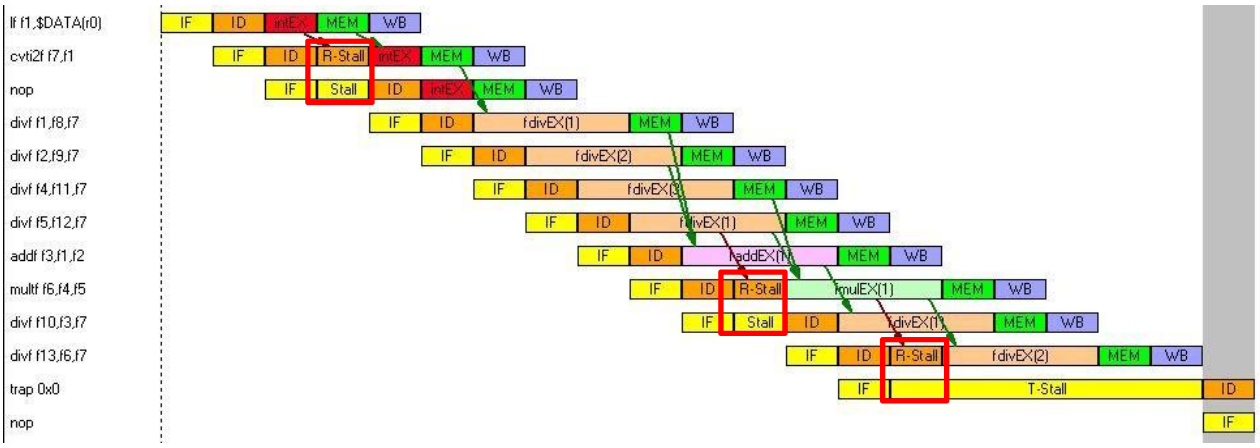
则整个程序执行过程中，一共发生 5 次数据相关、5 次结构相关。

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)
```

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



可以观察到，在指令 lf f1, $DATA[0] 与 cvti2f f7, f1 之间存在数据相关，进而导致 cvti2f f7, f1 与后续的 nop 指令之间产生结构相关性，这些相关性依然存在。

继续向后分析，divf f5, f12, f7 与 multf f6, f4, f5 之间存在数据相关，因为浮点除法需要三个周期完成，而 multf 指令依赖于 f5 的结果，因此会导致一个读取停顿（R-Stall）。由于 multf f6, f4, f5 在译码阶段引发了 R-Stall，进而使得 divf f10, f3, f7 产生结构相关。同理，multf f6, f4, f5 本身也存在数据相关，而 divf f13, f6, f7 与其后续指令之间同样存在结构相关。

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

```
Total:
    21 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: 3 (14.28% of all Cycles), thereof:
        LD stalls: 1 (33.33% of RAW stalls)
        Branch/Jump stalls: 0 (0.00% of RAW stalls)
        Floating point stalls: 2 (66.67% 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: 6 (28.57% of all Cycles)
    Total: 9 Stall(s) (42.86% 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)
```

此时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

$\frac{27}{21}$ = 1.29 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

分析：

由于存在数据相关、结构相关和控制相关，导致某些指令无法在预定的时钟周期内执行。流水线中的这些冲突会对指令的正常执行造成较大影响，若不能有效解决，不仅会降低流水线性能，甚至可能导致程序运行错误。针对不同类型的冲突，若通过优化指令调度策略，便能显著缓解这些问题。通过调度优化，无论是结构相关还是数据相关，其发生频率都有明显下降，同时程序的总执行周期也显著减少。编译器通过重新排列指令顺序实现“指令调度”，从而消除部分停顿周期，这有助于提升流水线的整体性能，使其运行更加高效。暂停周期的大幅减少也带来了执行周期的下降，从而提升了 CPU 的运算效率和整个系统的性能。

体会：

本次实验主要围绕指令调度的基本原理展开。通过对比指令调度前后各指令的执行情况，分析指令执行顺序对程序整体性能的影响。从根本上讲，数据相关和结构相关的问题，主要源于前一条指令由于延迟占用了功能部件，导致当前指令无法顺利进入流水线，从而造成流水线断流。

具体而言，结构相关是由于硬件资源不足以支持多条指令的重叠执行，导致发生资源冲突；数据相关则是指一条指令需要依赖前一条尚未完成的指令的结果，而这些指令在流水线中处于重叠状态，因此产生依赖冲突；控制相关问题通常出现在分支跳转指令的处理过程中。

实验表明，一旦在流水线中出现相关性问题，必然会对指令的顺利执行带来影响。如果不能有效处理这些相关问题，轻则降低流水线的效率，重则可能造成程序执行错误。为了解决这些问题，常用的方法是让流水线暂时停止部分指令的执行，同时允许其他指令继续执行，以此来消除相关带来的影响。