

# 实验5 指令调度与延迟分支

## 1. 实验目的

- (1) 加深对指令调度技术的理解。
- (2) 加深对延迟分支技术的理解。
- (3) 熟练账务用指令调度技术解决流水线中的数据冲突的方法。
- (4) 进一步理解指令调度技术对 CPU 性能的改进。
- (5) 进一步理解延迟分支技术对 CPU 性能的改进。

## 2. 实验平台

实验平台采用指令级和流水线操作级模拟器 MIPSsim。

## 3. 实验原理

- (1) 启动 MIPSsim。
- (2) 根据实验 2 的相关知识中关于流水线各段操作的描述，进一步理解流水线窗口中各段的功能，掌握各流水线寄存器的含义（双击各段，就可以看到各流水线寄存器中的内容）。
- (3) 选择“配置”→“流水方式”选项，使模拟器工作在流水方式下。
- (4) 用指令调度技术解决流水线中的结构冲突与数据冲突：
- (5) 用延迟分支技术减少分支指令对性能的影响：

## 4. 实验过程

### 4.1

- 1) 启动 MIPSsim。
- 2) 用 MIPSsim 的“文件”->“载入程序”选项来加载 `schedule.s`（在模拟器所在文件夹下的“样例程序”文件夹中）。

```
.text
main:
ADDIU $r1,$r0,A # r1 = A = 56
LW    $r2,0($r1) # r2 = A[0] = 4
ADD   $r4,$r0,$r2
SW    $r4,0($r1) # A[0] = r4 = 0
LW    $r6,4($r1) # r6 = A[1] = 6
ADD   $r8,$r6,$r1 # r8 = 6 + 56 = 62
MUL   $r12,$r10,$r1 # r12 = 0 * 56 = 0
ADD   $r16,$r12,$r1 # r16 = 0 + 56 = 56
ADD   $r18,$r16,$r1 # r18 = 56 + 56 = 112
SW    $r18,16($r1) # A[4] = r18 = 112
LW    $r20,8($r1) # r20 = A[2] = 8
MUL   $r22,$r20,$r14 # r22 = 8 * 0 = 0
MUL   $r24,$r26,$r14 # r24 = 0 * 0 = 0
TEQ   $r0,$r0

.data
A:
.word 4,6,8
```

3) 关闭定向功能，这是通过“配置”->“定向”选项来实现的。

4) 执行所载入的程序，通过查看统计数据和时钟周期图，找出并记录程序执行过程中各种冲突发生的次数，发生冲突的指令组合以及程序执行的总时钟周期数。

## 运行结果1

汇总：

执行周期总数：33

ID段执行了15条指令

硬件配置：

内存容量：4096 B

加法器个数：1      执行时间（周期数）：6

乘法器个数：1      执行时间（周期数）7

除法器个数：1      执行时间（周期数）10

定向机制：不采用

停顿（周期数）：

RAW停顿：16      占周期总数的百分比：48.48485%

其中：

load停顿：6      占有所有RAW停顿的百分比：37.5%

浮点停顿：0      占有所有RAW停顿的百分比：0%

WAW停顿：0      占周期总数的百分比：0%

结构停顿：0      占周期总数的百分比：0%

控制停顿：0      占周期总数的百分比：0%

自陷停顿：1      占周期总数的百分比：3.030303%

停顿周期总数：17      占周期总数的百分比：51.51515%

分支指令：

指令条数：0      占指令总数的百分比：0%

其中：

分支成功：0      占分支指令数的百分比：0%

分支失败：0      占分支指令数的百分比：0%

load/store指令：

指令条数：5      占指令总数的百分比：33.33333%

其中：

load：3      占load/store指令数的百分比：60%

store：2      占load/store指令数的百分比：40%

浮点指令：

指令条数：0      占指令总数的百分比：0%

其中：

加法：0      占浮点指令数的百分比：0%

乘法：0      占浮点指令数的百分比：0%

除法：0      占浮点指令数的百分比：0%

自陷指令：

指令条数：1      占指令总数的百分比：6.666667%

## 分析1

RAW停顿: 16, 自陷停顿: 1, 执行周期总数: 33。

发生 RAW 冲突的指令组合:

ADDIU \$r1,\$r0,A 和 LW \$r2,0(\$r1)

LW \$r2,0(\$r1) 和 ADD \$r4,\$r0,\$r2

ADD \$r4,\$r0,\$r2 和 SW \$r4,0(\$r1)

LW \$r6,4(\$r1) 和 ADD \$r8,\$r6,\$r1

MUL \$r12,\$r10,\$r1 和 ADD \$r16,\$r12,\$r1

ADD \$r16,\$r12,\$r1 和 ADD \$r18,\$r16,\$r1

ADD \$r18,\$r16,\$r1 和 SW \$r18,16(\$r1)

LW \$r20,8(\$r1) 和 MUL \$r22,\$r20,\$r14

5) 自己采用调度技术对程序进行指令调度, 消除冲突(自己修改源程序)。将调度(修改)后的程序重新命名为 afer-schedule.s。(注意: 调度方法灵活多样, 在保证程序正确性的前提下自己随意调度, 尽量减少冲突即可, 不要求要达到最优。)

```
.text
main:
    ADDIU $r1,$r0,A    # $r1 = A = 56
    MUL $r24,$r26,$r14 # $r24 = 0 * 0 = 0
    LW $r2,0($r1)      # $r2 = A[0] = 4
    MUL $r12,$r10,$r1  # $r12 = 0 * 56 = 0
    LW $r6,4($r1)      # $r6 = A[1] = 6
    LW $r20,8($r1)     # $r20 = A[2] = 8
    ADD $r16,$r12,$r1  # $r16 = 0 + 56 = 56
    ADD $r4,$r0,$r2    # $r4 = 0 + 4 = 4
    ADD $r8,$r6,$r1    # $r8 = 6 + 56 = 62
    ADD $r18,$r16,$r1  # $r18 = 56 + 56 = 112
    SW $r4,0($r1)      # A[0] = 4
    MUL $r22,$r20,$r14 # $r22 = 8 * 0 = 0
    SW $r18,16($r1)    # A[4] = 112
    TEQ $r0,$r0

.data
A:
.word 4,6,8
```

6) 载入 afer-schedule.s, 执行该程序, 观察程序在流水线中的执行情况, 记录程序执行的总时钟周期数。

## 运行结果2

汇总:

执行周期总数: 18

ID段执行了15条指令

硬件配置：

内存容量：4096 B

加法器个数：1      执行时间（周期数）：6

乘法器个数：1      执行时间（周期数）7

除法器个数：1      执行时间（周期数）10

定向机制：不采用

停顿（周期数）：

RAW停顿：1      占周期总数的百分比：5.555555%

其中：

load停顿：0      占有RAW停顿的百分比：0%

浮点停顿：0      占有RAW停顿的百分比：0%

WAW停顿：0      占周期总数的百分比：0%

结构停顿：0      占周期总数的百分比：0%

控制停顿：0      占周期总数的百分比：0%

自陷停顿：1      占周期总数的百分比：5.555555%

停顿周期总数：2      占周期总数的百分比：11.111111%

分支指令：

指令条数：0      占指令总数的百分比：0%

其中：

分支成功：0      占分支指令数的百分比：0%

分支失败：0      占分支指令数的百分比：0%

load/store指令：

指令条数：5      占指令总数的百分比：33.333333%

其中：

load：3      占load/store指令数的百分比：60%

store：2      占load/store指令数的百分比：40%

浮点指令：

指令条数：0      占指令总数的百分比：0%

其中：

加法：0      占浮点指令数的百分比：0%

乘法：0      占浮点指令数的百分比：0%

除法：0      占浮点指令数的百分比：0%

自陷指令：

指令条数：1      占指令总数的百分比：6.666667%

7) 比较调度前和调度后的性能，论述指令调度对提高 CPU 性能的作用。

## 分析2

调度后的性能提升了  $\frac{33}{18} \approx 1.83$  倍。指令调度可以消除部分的数据冲突，减少了停顿的周期数，提高了 CPU 使用率。

## 4.2

1) 在 MIPSSim 中载入 `branch.s` 样例程序（在本模拟器目录的“样例程序”文件夹中。

```
.text
main:
ADDI $r2,$r0,1024 # $r2 = 1024
ADD $r3,$r0,$r0 # $r3 = 0
```

```

ADDI $r4,$r0,8 # $r4 = 8
loop:
LW $r1,0($r2) # $r1 = MEM[$r2] = MEM[1024]
ADDI $r1,$r1,1 # $r1 = $r1 + 1
SW $r1,0($r2) # MEM[$r2] = $r1 = MEM[1024] = MEM[1024] + 1
ADDI $r3,$r3,4 # $r3 = $r3 + 4
SUB $r5,$r4,$r3 # $r5 = $r4 - $r3 = 8 - $r3
BGTZ $r5,loop # if $r5 > 0, goto loop
ADD $r7,$r0,$r6 # $r7 = $r0 + $r6 = $r6
TEQ $r0,$r0

```

2) 关闭延迟分支功能。这是通过在“配置”->“延迟槽”选项来实现的。

3) 执行该程序，观察并记录发生分支延迟的时刻，记录该程序执行的总时钟周期数。

## 运行结果1

汇总：

执行周期总数：38

ID段执行了18条指令

硬件配置：

内存容量：4096 B

加法器个数：1      执行时间（周期数）：6

乘法器个数：1      执行时间（周期数）7

除法器个数：1      执行时间（周期数）10

定向机制：不采用

停顿（周期数）：

RAW停顿：16      占周期总数的百分比：42.10526%

其中：

load停顿：4      占有所有RAW停顿的百分比：25%

浮点停顿：0      占有所有RAW停顿的百分比：0%

WAW停顿：0      占周期总数的百分比：0%

结构停顿：0      占周期总数的百分比：0%

控制停顿：2      占周期总数的百分比：5.263158%

自陷停顿：1      占周期总数的百分比：2.631579%

停顿周期总数：19      占周期总数的百分比：50%

分支指令：

指令条数：2      占指令总数的百分比：11.11111%

其中：

分支成功：1      占分支指令数的百分比：50%

分支失败：1      占分支指令数的百分比：50%

load/store指令：

指令条数：4      占指令总数的百分比：22.22222%

其中：

load：2      占load/store指令数的百分比：50%

store：2      占load/store指令数的百分比：50%

浮点指令：

指令条数：0      占指令总数的百分比：0%

其中：

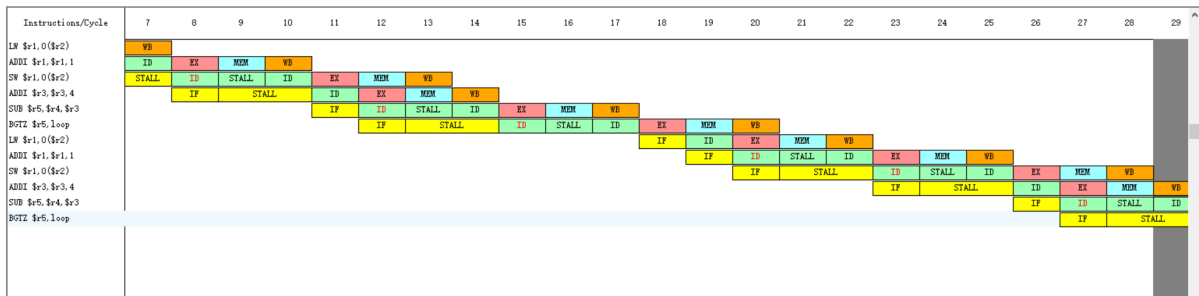
加法：0      占浮点指令数的百分比：0%

乘法：0          占浮点指令数的百分比：0%  
除法：0          占浮点指令数的百分比：0%

自陷指令：

指令条数：1          占指令总数的百分比：5.555555%

分支指令是 `BGTZ $r5,loop`。当这条指令执行时，下一条指令 `ADD $r7,$r0,$r6` 可能已经在流水线中了。所以，即使 `$r5` 的值大于 0，CPU 也需要执行 `ADD $r7,$r0,$r6` 这条指令，然后才能跳转到 `loop`。这就是发生分支延迟的时刻。



发生分支延迟的时刻：15，执行周期总数：38。

4) 假设延迟槽为一个，自己对 `branch.s` 程序进行指令调度（自己修改源程序），将调度后的程序重新命名为 `delayed-branch.s`。

```
.text
main:
    ADDI    $r2,$r0,1024 # r2 = 1024
    ADD     $r3,$r0,$r0 # r3 = 0
    ADDI    $r4,$r0,8 # r4 = 8
loop:
    LW      $r1,0($r2) # r1 = mem[r2] = mem[1024]
    ADDI    $r3,$r3,4 # r3 = r3 + 4
    ADDI    $r1,$r1,1 # r1 = r1 + 1
    SUB     $r5,$r4,$r3 # r5 = r4 - r3
    SW      $r1,0($r2) # mem[r2] = r1
    BGTZ    $r5,loop # if r5 > 0 goto loop

    ADD     $r7,$r0,$r6 # r7 = r6
    TEQ     $r0,$r0
```

5) 载入 `delayed-branch.s`，打开延迟分支功能，执行该程序，观察其时钟周期图，记录程序执行的总时钟周期数。

## 运行结果2

汇总：

执行周期总数：25  
ID段执行了19条指令

硬件配置：

内存容量：4096 B  
加法器个数：1          执行时间（周期数）：6  
乘法器个数：1          执行时间（周期数）7  
除法器个数：1          执行时间（周期数）10  
定向机制：不采用

停顿（周期数）：

RAW停顿：4            占周期总数的百分比：16%

其中：

load停顿：2            占有所有RAW停顿的百分比：50%

浮点停顿：0            占有所有RAW停顿的百分比：0%

WAW停顿：0            占周期总数的百分比：0%

结构停顿：0            占周期总数的百分比：0%

控制停顿：0            占周期总数的百分比：0%

自陷停顿：1            占周期总数的百分比：4%

停顿周期总数：5        占周期总数的百分比：20%

分支指令：

指令条数：2            占指令总数的百分比：10.52632%

其中：

分支成功：1            占分支指令数的百分比：50%

分支失败：1            占分支指令数的百分比：50%

load/store指令：

指令条数：4            占指令总数的百分比：21.05263%

其中：

load：2            占load/store指令数的百分比：50%

store：2            占load/store指令数的百分比：50%

浮点指令：

指令条数：0            占指令总数的百分比：0%

其中：

加法：0            占浮点指令数的百分比：0%

乘法：0            占浮点指令数的百分比：0%

除法：0            占浮点指令数的百分比：0%

自陷指令：

指令条数：1            占指令总数的百分比：5.263158%

## 实验结论

对比不采用延迟分支和采用延迟分支两种情况下的时钟周期图，比较两种情况下的性能之间的不同，论述延迟分支对提高 CPU 性能的作用。

从给出的两次运行结果来看，我们可以看到以下几点差异：

1. 执行周期总数：第一次运行的执行周期总数为38，而第二次运行的执行周期总数为25。这表明第二次运行的程序在执行过程中花费的时间更少，性能更高。
2. ID段执行的指令数：第一次运行的ID段执行了18条指令，而第二次运行的ID段执行了19条指令。这表明第二次运行的程序在同样的时间内执行了更多的指令，性能更高。
3. 停顿（周期数）：第一次运行的停顿周期总数为19，占周期总数的百分比为50%，而第二次运行的停顿周期总数为5，占周期总数的百分比为20%。这表明第二次运行的程序在执行过程中的停顿时间更少，性能更高。
4. 分支指令、load/store指令、浮点指令和自陷指令的执行情况在两次运行中基本相同。

综上所述，采用延迟分支的情况下，程序的执行性能更高。延迟分支是一种优化CPU性能的技术，它通过改变指令的执行顺序，使得在分支指令执行后的一段时间内，CPU可以继续执行其他指令，从而避免了因分支预测错误而导致的CPU空闲时间，提高了CPU的利用率和执行效率。

