

# 深圳大学实验报告

课程名称: 计算机系统(3)

实验项目名称: 处理器结构实验一

学院: 计算机与软件学院

专业: 计算机与软件学院所有专业

指导教师: 刘刚

报告人: \_\_\_ 学号: \_\_\_ 班级: \_\_\_

实验时间: 2025年11月5日

实验报告提交时间: 2025年11月6日

## 一、 实验目标：

了解 MIPS 的五级流水线，和在运行过程中的所产生的各种不同的流水线冒险  
通过指令顺序调整，或旁路与预测技术来提高流水线效率  
更加了解流水线细节和其指令的改善方法  
更加熟悉 MIPS 指令的使用

## 二、实验内容

观察一段代码并运行，观察其中的流水线冒险，并记录统计信息。  
对所给的代码进行指令序列的调整，以期避免数据相关，并记录统计信息。  
启动 forward 功能，以获得性能提升，并且记录统计信息。  
(选做：用 perf 记录 x86 中的数据相关于指令序列调整后的时间统计、  
调整指令，以避免连续乘法间的阻塞。)

## 三、实验环境

硬件：桌面 PC  
软件：Windows，WinMIPS64 仿真器

## 四、实验步骤及说明

首先，我们给出一段 C 代码，该段代码实现的是两个矩阵相加。

设有 4\*4 矩阵 A 和 4\*4 矩阵 B 相加，得到 4\*4 矩阵 C：

```
for(int i = 0; i < 4; i++)  
    For(int j = 0; j < 4; j++)  
        C[i][j] = A[i][j] + B[i][j];
```

根据上述的 C 代码，我们将其转换成 MIPS 语言，然后运行，并进行分析。

MIPS 代码如下：

```
.data  
a: .word 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4  
b: .word 4, 4, 4, 4, 3, 3, 3, 3, 2, 2, 2, 2, 1, 1, 1, 1  
c: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  
len: .word 4  
control: .word32 0x10000  
data: .word32 0x10008  
  
.text  
start:daddi r17, r0, 0  
       daddi r21, r0, a  
       daddi r22, r0, b  
       daddi r23, r0, c  
       ld r16, len(r0)  
loop1: slt r8, r17, r16  
       beq r8, r0, exit1
```

```
daddi r19, r0, 0  
loop2: slt r8, r19, r16  
       beq r8, r0, exit2
```

```
dsll r8, r17, 2  
dadd r8, r8, r19  
dsll r8, r8, 3
```

```
dadd r9, r8, r21  
dadd r10, r8, r22  
dadd r11, r8, r23
```

```
ld r9, 0(r9)  
ld r10, 0(r10)  
dadd r12, r9, r10  
sd r12, 0(r11)
```

```
daddi r19, r19, 1  
j loop2  
exit2:daddi r17, r17, 1  
j loop1  
exit1: halt
```

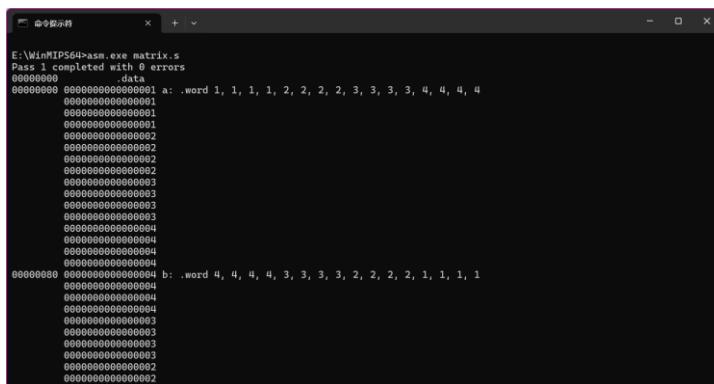
实验前请保证 winMIPS64 配置中“Enable Forwarding”没有选中。将这段代码加载到 WinMIPS64 中，运行后观察结果（提供 Statistic 窗口截图）。从 Statistic 窗口记录：本程序运行过程中总共产生了多少次 RAW 的数据相关。接下来，我们对产生数据相关的代码逐个分析，请列出产生数据相关的代码，并在下一步中进行分析和优化。

## 一、 调整指令序列

在这一部分，我们利用指令调整的方法对数据相关代码进行优化，规避数据相关。

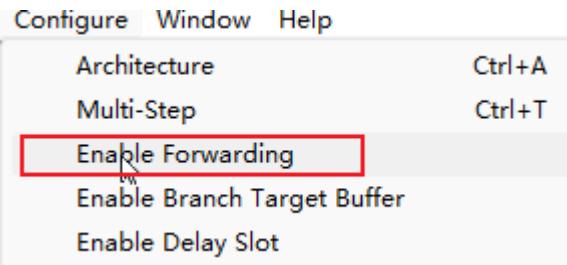
通过**调整序列**来规避这个数据相关，在 **statics** 窗口中记录其效果。将此结果与初始的结果进行对比，报告 **RAW 相关的次数减少**的数量。

1. 保存代码为 matrix.s，用 asm.exe 检查其是否正确，如下图所示。

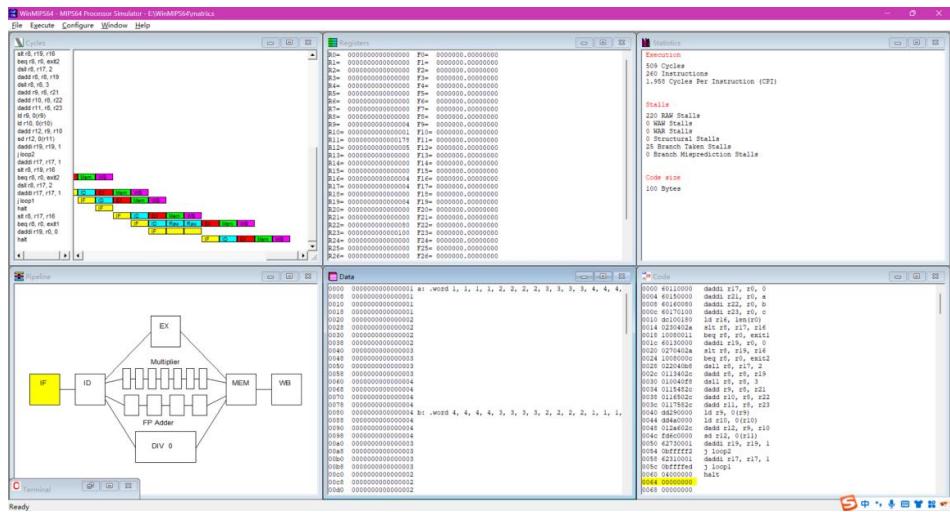


The screenshot shows the WinMIPS64 command-line interface. The command entered is "E:\WinMIPS64>asm.exe matrix.s". The output shows the assembly code and its corresponding binary representation. The assembly code includes instructions like daddi, slt, beq, dsll, dadd, and ld. The binary output shows memory locations starting from 00000000, with labels 'a' and 'b' pointing to specific memory addresses.

2. 将 matrix.s 载入 WinMIPS64 中，并关闭 Configure – Enable Forwarding，如下图所示



3. 运行后观察结果，发现有 220 次 RAW 数据相关，并且矩阵元素值都为 5，结果正确



4. 逐步执行程序，发现有以下数据相关的指令导致堵塞。

① 如下代码中，r16 需读取后再进行运算，发生数据冒险。

```
start: daddi r17,r0,0
daddi r21,r0,a
daddi r22,r0,b
daddi r23,r0,c
ld r16,len(r0) #有误
```

改进措施：将 ld 指令提前，以避免数据冲突。

```
start: ld r16,len(r0) #改正
daddi r17,r0,0
daddi r21,r0,a
daddi r22,r0,b
daddi r23,r0,c
```

②如下代码中，`beq` 指令读取 `r8` 时发生数据冒险。`slt r8,r17,r16` 指令后紧接着执行 `beq r8,r0,exit1` 指令。这两条指令之间存在数据依赖性，即 `beq` 指令需要使用 `slt` 指令的结果 `r8`，而 `slt` 指令的结果会在执行完后才被写入寄存器 `r8`。如果在此之间没有足够的延迟或者数据前推等技术来解决数据冒险，就有可能导致问题。

```
loop1: slt r8,r17,r16  
        beq r8,r0,exit1 #有误  
        addi r19,r0,0
```

改进措施：将 `beq` 指令滞后即可。

```
loop1: slt r8,r17,r16  
        addi r19,r0,0  
        beq r8,r0,exit1 #改正
```

③如下图代码中，`beq` 指令读取 `r8` 时候有数据相关

```
loop2: slt r8,r19,r16  
        beq r8,r0,exit2 #有误  
  
        dsll r8,r17,2  
        dadd r8,r8,r19  
        dsll r8,r8,3  
        dadd r9,r8,r21  
        dadd r10,r8,r22  
        dadd r11,r8,r23  
  
        ld r9,0(r9)  
        ld r10,0(r10)  
        dadd r12,r9,r10  
        sd r12,0(r11)  
  
        addi r19,r19,1  
        j loop2
```

改进措施：可将不冲突的指令 `daddi r19,r19,1` 提前。

```
loop2: slt r8,r19,r16
```

```
    daddi r19,r19,1 #改正
```

```
    beq r8,r0,exit2
```

```
    dsll r8,r17,2
```

```
    dadd r8,r8,r19
```

```
    dsll r8,r8,3
```

```
    dadd r9,r8,r21
```

```
    dadd r10,r8,r22
```

```
    dadd r11,r8,r23
```

```
    ld r9,0(r9)
```

```
    ld r10,0(r10)
```

```
    dadd r12,r9,r10
```

```
    sd r12,0(r11)
```

```
    j loop2
```

④如下图代码中，`dadd r12,r9,r10` 发生数据冒险（取数-使用型）

```
dadd r9,r8,r21
```

```
dadd r10,r8,r22
```

```
dadd r11,r8,r23
```

```
ld r9,0(r9)
```

```
ld r10,0(r10)
```

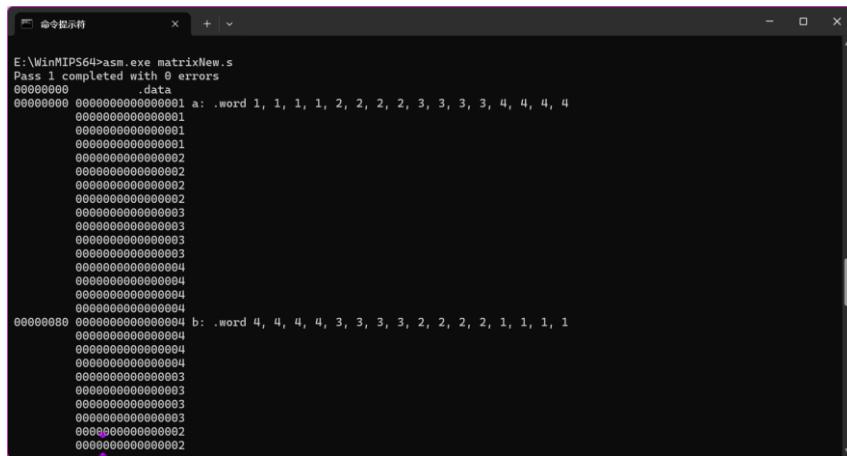
```
dadd r12,r9,r10 #有误
```

```
sd r12,0(r11)
```

改进措施：可将不冲突的指令 dadd r11,r8,r23 滞后。此时 r12 可能发生冲突，但开启前推可解决。

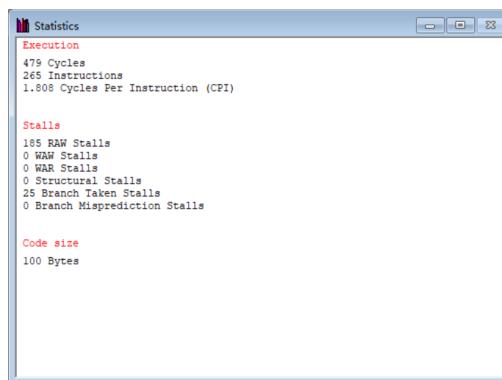
```
dadd r9,r8,r21  
dadd r10,r8,r22  
  
ld r9,0(r9)  
ld r10,0(r10)  
dadd r11,r8,r23 #改正  
  
dadd r12,r9,r10  
sd r12,0(r11)
```

## 5. 保存修改后的代码为 matrixNew.s，用 asm.exe 检查是否有误



```
E:\WinMIPS64>asm.exe matrixNew.s
Pass 1 completed with 0 errors
00000000 .data
00000000 0000000000000001 a: .word 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000000 0000000000000004 b: .word 4, 4, 4, 4, 3, 3, 3, 2, 2, 2, 1, 1, 1
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
```

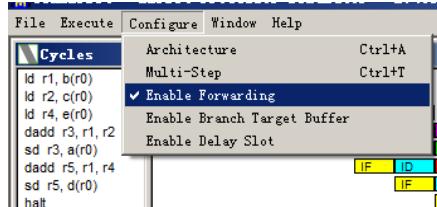
6. 将 matrixNew.s 载入 WinMIPS64 中，并关闭 Configure – Enable Forwarding，观察运行结果，发生了 185 次 RAW，与之前相比少了 35 次



## 二、Forwarding 功能开启

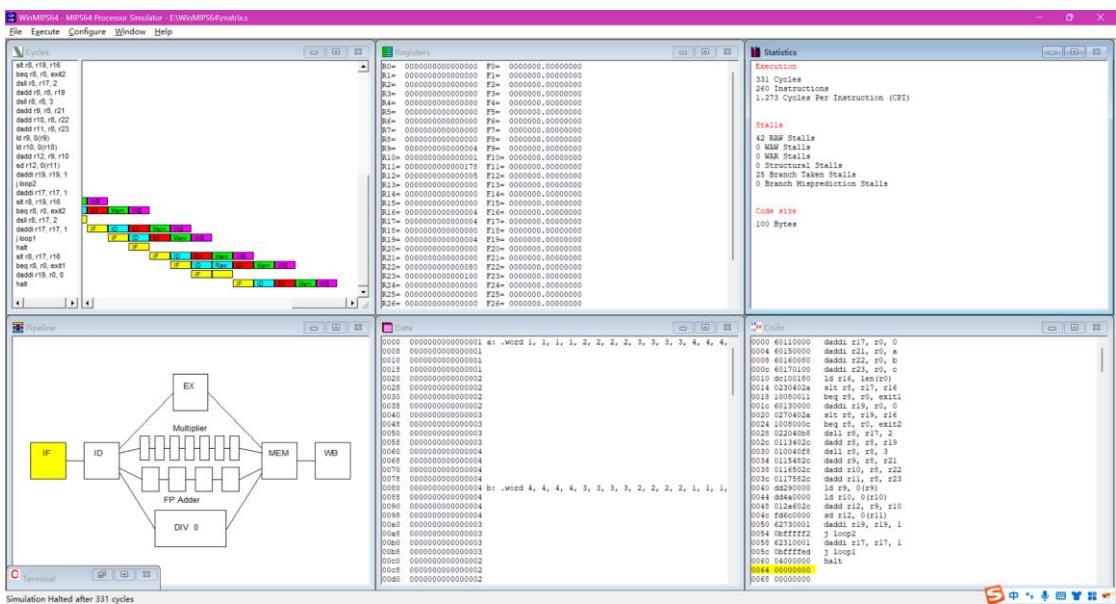
接下来，我们要展示 Forwarding 功能的优化效果。

首先，我们要知道如何开启 Forwarding 功能。方法如下：点开 **configure** 下拉窗口，给 **Enable Forwarding** 选项左侧点上勾。

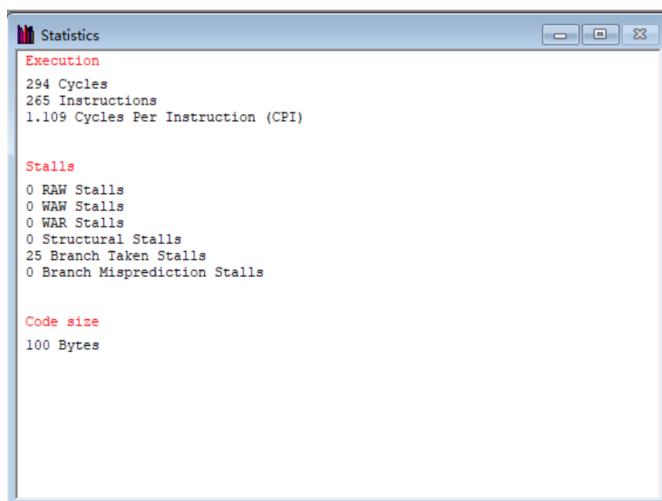


开启了 Forwarding 功能之后，我们再运行，查看结果，解释哪些数据相关的问题得到解决，并以截图说明问题解决前后的差异所在。

1. 运行 matrix 代码可以看到会有 42 个 RAW



2. 运行优化后的 matrixNew，可以观察到 RAW 下降至 0 次



### 三、 结构相关优化

流水线中的结构相关，指的是流水线中多条指令在同一时钟周期内争用同一功能部件现象。即因硬件资源满足不了指令重叠执行的要求而发生的冲突。

在 WinMIPS64 中，我们可以在除法中观察到这种现象。要消除这种结构相关，我们可以采取调整指令位置的方法进行优化。在这个部分，我们首先给出几条 C 代码，然后将该代码翻译成 MIPS 代码（为了观察的方便，我们这里 MIPS 代码并不是逐一翻译，而是调整代码，使得其他部分数据相关已经优化，而两条除法指令连续出现），运行并查看结果。接着，调整代码序列，重新运行。观察优化效果。

下面是给出的 C 代码：

```
a = a / b  
c = c / d  
e = e + 1  
f = f + 1  
g = g + 1  
h = h + 1  
i = i + 1  
j = j + 1
```

根据上述的 C 代码，我们给出数据相关优化的指令如下：

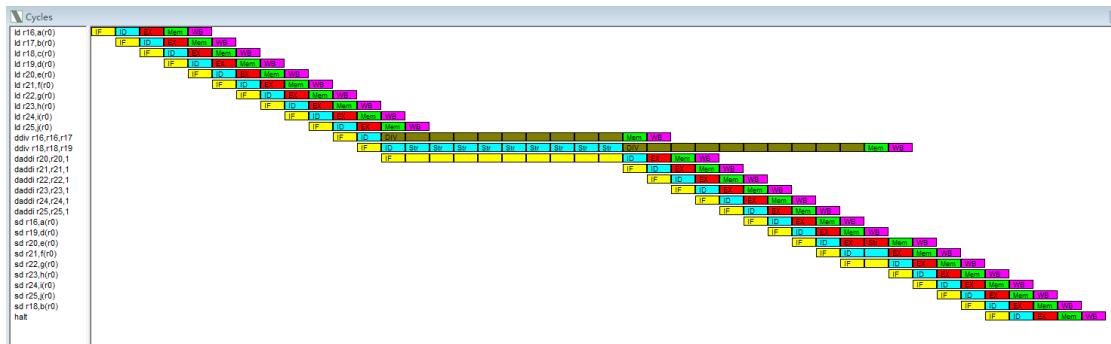
```
.data  
a:    .word    12  
b:    .word    3  
c:    .word    15  
d:    .word    5  
e:    .word    1  
f:    .word    2  
g:    .word    3  
h:    .word    4  
i:    .word    5  
  
.text  
start:  
    ld r16, a(r0)  
    ld r17, b(r0)  
    ld r18, c(r0)  
    ld r19, d(r0)  
    ld r20, e(r0)  
    ld r21, f(r0)  
    ld r22, g(r0)  
    ld r23, h(r0)  
    ld r24, i(r0)  
    ddiv r16, r16, r17  
    ddiv r18, r18, r19  
    addi r20, r20, 1  
    addi r21, r21, 1
```

```

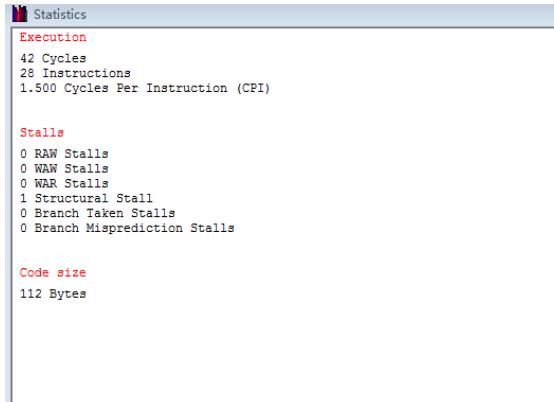
daddi r22, r22, 1
daddi r23, r23, 1
daddi r24, r24, 1
halt

```

上面的指令运行，在 *Cycle* 窗口结果如下（程序运行前请将 *configure->architecture->division latency 改为 10*）：



在 *Statistics* 窗口的结果如下：

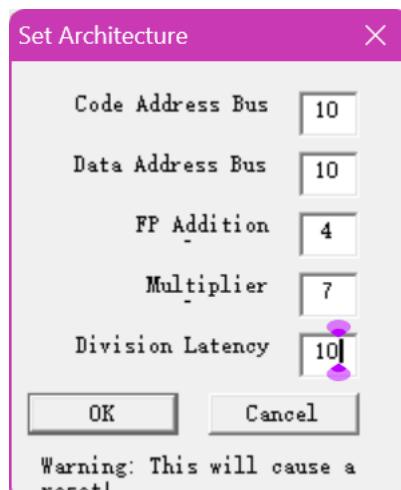


通过观察，我们可以发现，两个连续的除法产生了明显的结构相关，第二个除法为了等待上一个除法指令在执行阶段所占用的资源，阻塞了 9 个周期。

显然，这样的连续的除法所导致的结构相关极大的降低了流水线效率，为了消除结构相关，我们需要做的是调整指令序列，将其他无关的指令塞入两条连续的除法指令中。

给出指令序列的调整方案并给出流水线工作状态的截图，做出解释。

1. 将 *configure->architecture->division latency* 改为 10



## 2.编写 div.s

```
.data  
  
a: .word 12  
  
b: .word 3  
  
c: .word 15  
  
d: .word 5  
  
e: .word 1  
  
f: .word 2  
  
g: .word 3  
  
h: .word 4  
  
i: .word 5  
  
j: .word 6  
  
  
.text  
  
start:  
  
    ld r16, a(r0)  
  
    ld r17, b(r0)  
  
    ld r18, c(r0)  
  
    ld r19, d(r0)  
  
    ld r20, e(r0)  
  
    ld r21, f(r0)  
  
    ld r22, g(r0)  
  
    ld r23, h(r0)  
  
    ld r24, i(r0)  
  
    ld r25, j(r0)  
  
    ddiv r16, r16, r17  
  
    ddiv r18, r18, r19  
  
    addi r20, r20, 1  
  
    addi r21, r21, 1  
  
    addi r22, r22, 1
```

```

daddi r23, r23, 1
daddi r24, r24, 1
daddi r25, r25, 1
sd r16, a(r0)
sd r18, c(r0)
sd r20, e(r0)
sd r21, f(r0)
sd r22, g(r0)
sd r23, h(r0)
sd r24, i(r0)
sd r25, j(r0)
halt

```

用 asm.exe 检查是否有误

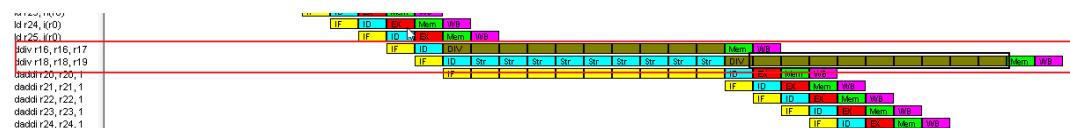
```

E:\WinMIPS64>asm.exe div.s
Pass 1 completed with 0 errors
00000000    .data
00000000 00000000000000c a: .word 12
00000000 000000000000003 b: .word 3
00000000 000000000000005 c: .word 5
00000000 000000000000001 d: .word 1
00000000 000000000000002 e: .word 2
00000000 000000000000003 f: .word 3
00000000 000000000000004 g: .word 4
00000000 000000000000005 h: .word 4
00000000 000000000000006 i: .word 5
00000000 000000000000006 j: .word 6

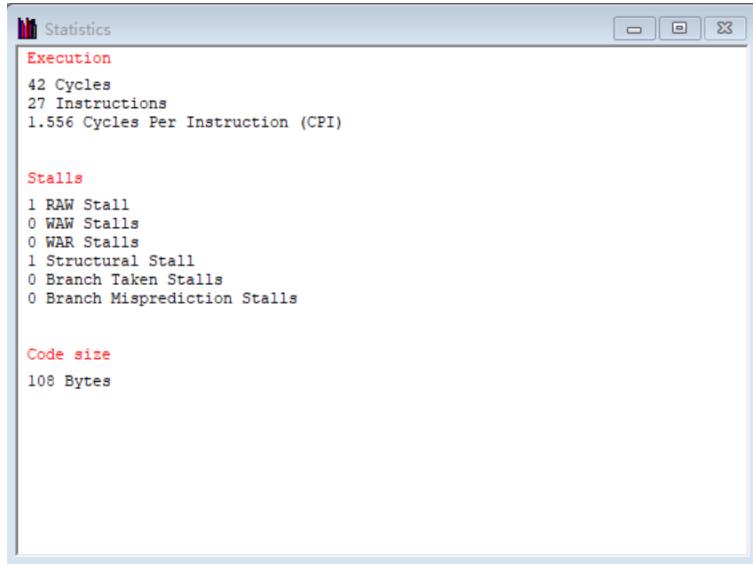
00000000    .text
00000000 start:
00000000 dcl00000  ld r16, a(r0)
00000000 dcl10008  ld r17, b(r0)
00000008 dcl10010  ld r18, c(r0)
00000010 dcl10018  ld r19, d(r0)
00000016 dcl10020  ld r20, e(r0)
00000014 dcl10028  ld r21, f(r0)
00000018 dcl10030  ld r22, g(r0)
0000001c dcl170038 ld r23, h(r0)
00000020 dcl180048 ld r24, i(r0)
00000024 dcl190048 ld r25, j(r0)
00000028 0211801e ddiv r16, r16, r17
0000002c 0253901e ddiv r18, r18, r19
00000030 62940001 daddi r20, r20, 1
00000034 62b50001 daddi r21, r21, 1

```

3. 将其载入 WinMIPS64 并运行，观察结果如下，第二个除法为了等待上一个除法指令在执行阶段所占用的资源，阻塞了 9 个周期。



此时程序所需的时钟周期数如下图所示。



在 MIPS 处理器中，除法操作（ddiv）会占用较长时间，因此如果在除法指令之间直接执行其他可能依赖于其结果的指令，会发生流水线阻塞。为了避免这种情况，我们需要插入一些无关的指令，确保除法指令的执行不会因前后数据依赖而阻塞。

第一条除法指令： ddiv r16, r16, r17 执行后，r16 和 r17 会影响到后续的运算结果。因此，在执行除法指令后，后续的计算依赖于除法结果，如果不等待除法结果就执行后续指令，就会发生取数-使用数据冒险。

解决办法是插入一些不依赖于除法结果的指令。我们插入了 7 条无关指令（加载操作），这些指令可以确保除法操作完成并且结果已经计算出来，从而避免流水线阻塞。

第二条除法指令： ddiv r18, r18, r19 在第一条除法指令执行完毕后，再执行。这是因为 r19 的加载和计算在第一条除法指令执行后完成，因此可以有效避免数据冒险。

指令延迟： 在第二条除法指令后，再执行递增操作（daddi），这些操作不会直接依赖于除法结果，因此可以自由插入。

存储指令延迟： 存储操作（sd）被延迟执行，确保所有的计算都完成后再写回内存，避免存储冲突。

#### 4. 编写优化代码 divNew.s

```
.data
a: .word 12
b: .word 3
c: .word 15
d: .word 5
```

```
e: .word 1
f: .word 2
g: .word 3
h: .word 4
i: .word 5
j: .word 6

.text
start:
    ld r16, a(r0)      # 取 a
    ld r17, b(r0)      # 取 b
    ld r18, c(r0)      # 取 c

    # 第一条除法指令
    ddiv r16, r16, r17 # r16 = r16 / r17

    # 插入不冲突的指令，避免流水阻塞
    ld r19, d(r0)      # 取 d
    ld r20, e(r0)      # 取 e
    ld r21, f(r0)      # 取 f
    ld r22, g(r0)      # 取 g
    ld r23, h(r0)      # 取 h
    ld r24, i(r0)      # 取 i
    ld r25, j(r0)      # 取 j

    # 递增操作
    addi r20, r20, 1   # r20 = r20 + 1
    addi r21, r21, 1   # r21 = r21 + 1
```

```
# 第二条除法指令

ddiv r18, r18, r19 # r18 = r18 / r19

# 继续递增操作

daddi r22, r22, 1 # r22 = r22 + 1
daddi r23, r23, 1 # r23 = r23 + 1
daddi r24, r24, 1 # r24 = r24 + 1
daddi r25, r25, 1 # r25 = r25 + 1

# 存储结果

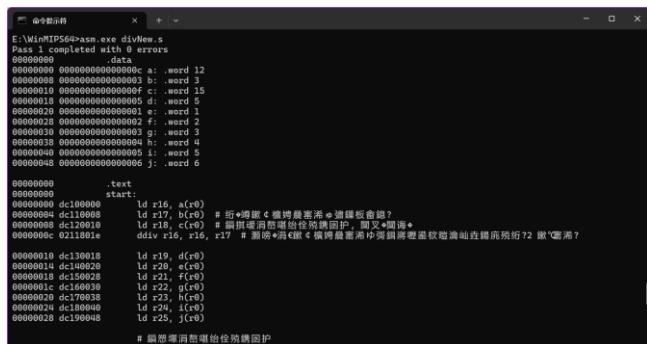
sd r20, e(r0)      # 存回 e
sd r21, f(r0)      # 存回 f
sd r22, g(r0)      # 存回 g
sd r23, h(r0)      # 存回 h
sd r24, i(r0)      # 存回 i
sd r25, j(r0)      # 存回 j

# 存回除法结果

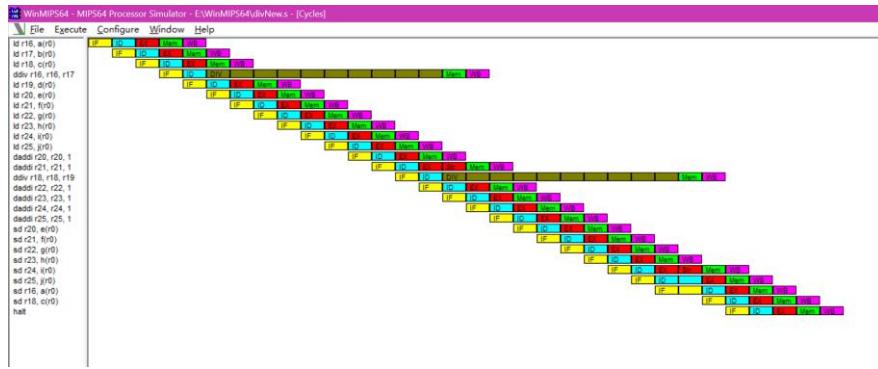
sd r16, a(r0)      # 存回 a
sd r18, c(r0)      # 存回 c

halt
```

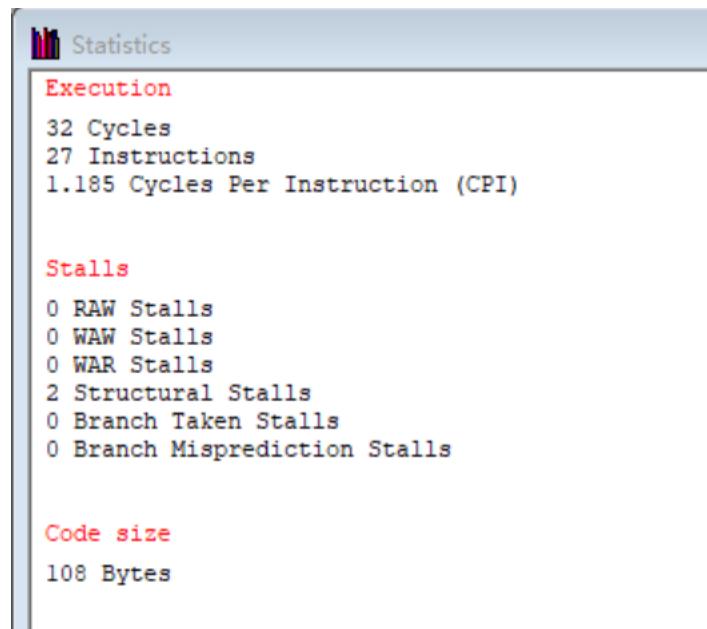
用 asm.exe 检查是否有误



4. 将其载入 WinMIPS64 并运行，观察结果如下，观察到两条除法指令的流水之间插入了其他指令，大大提升效率



此时程序所需的时钟周期数如下图所示。



#### 四、 提交报告

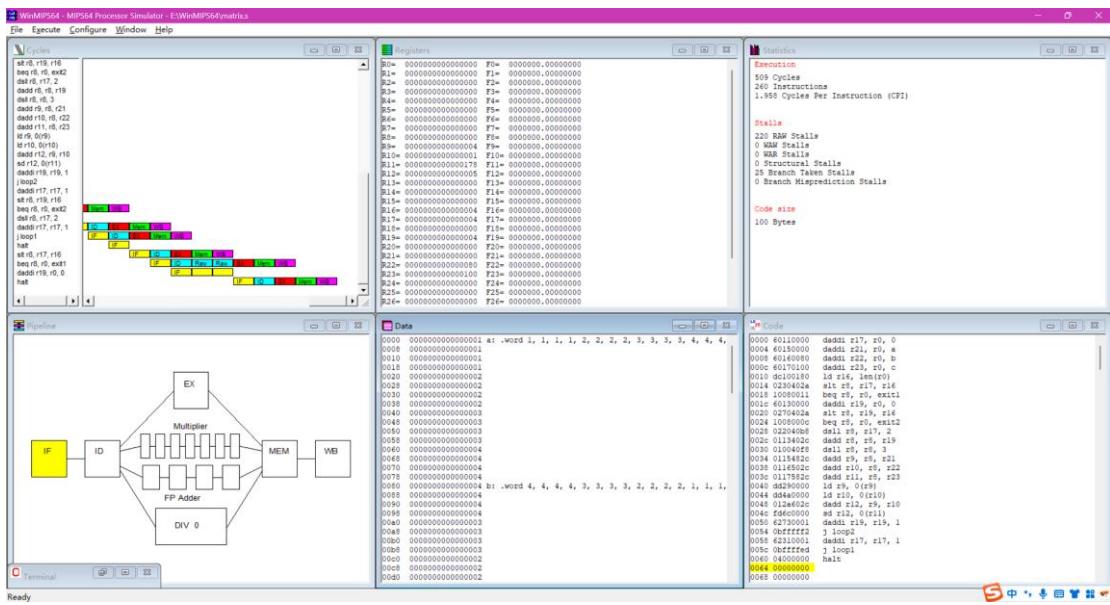
记录实验过程，保存实验截图，给出分析结果，形成实验报告。初始代码准备（10 分），后面每个优化方法各 30 分。

### 五、 实验结果

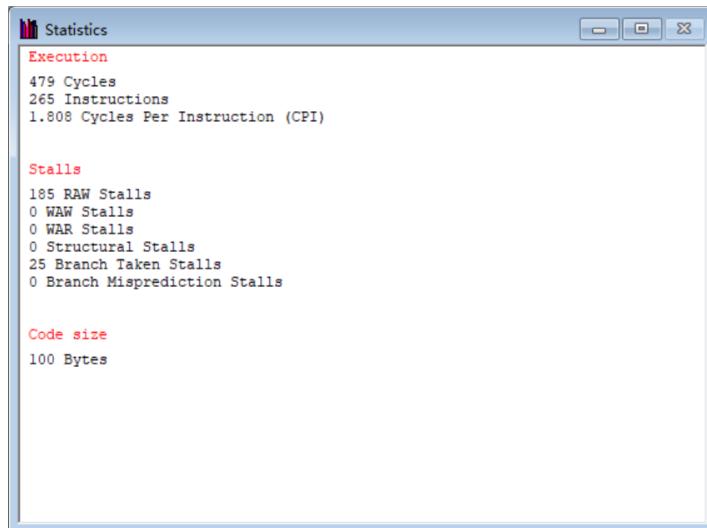
#### 1. 调整指令序列

修改前 220 次 RAW，修改后发生了 185 次 RAW，修改后减少了 35 次，如下图所示。

修改前

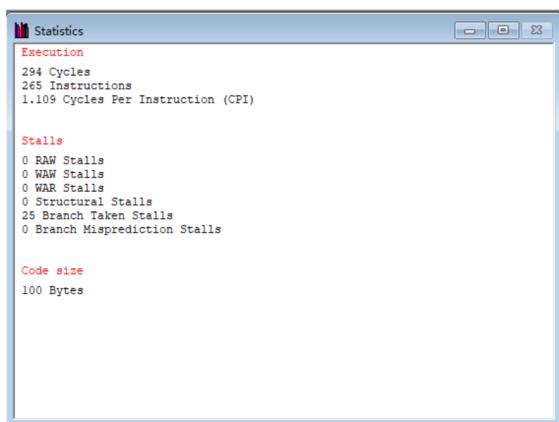


修改后



## 2. 开启前推

开启前推后运行程序， RAW 下降至 0 次，如下图所示。



### 3. 结构优化

修改前程序需 42 个时钟周期，修改后程序需 32 个时钟周期，如下图所示。

修改前

```
Statistics

Execution
42 Cycles
27 Instructions
1.556 Cycles Per Instruction (CPI)

Stalls
1 RAW Stall
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
0 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
108 Bytes
```

修改后

```
Statistics

Execution
32 Cycles
27 Instructions
1.185 Cycles Per Instruction (CPI)

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
2 Structural Stalls
0 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
108 Bytes
```

## 五、实验总结与体会

在完成处理器结构实验一的过程中，我深刻体会到了对于 MIPS 五级流水线的理解以及流水线冒险的重要性。通过实际操作观察到不同类型的流水线冒险如 RAW（读后写）冒险，深入了解了如何通过指令顺序的调整、旁路以及预测技术等手段来提高流水线效率。

在优化代码的过程中，我逐步分析了代码中导致数据相关的部分，并进行了调整和改进。通过调整指令序列以规避数据相关，我观察到了在性能提升方面的显著效果，并通过对初始结果和优化后的结果，清晰地看到了 RAW 相关次数减少的数量，这进一步验证了优化的有效性。

在实验中还发现了一些指令之间的数据冒险问题，如 beq 指令读取 r8 时发生数据冒险等情况。通过分析并采取相应的改进措施，如将关键指令滞后或提前，成功避免了数据冒险带来的性能损失，使得程序能够更加高效地执行。

通过这次实验，我不仅加深了对 MIPS 指令集的理解，还提升了对处理器结构和流水线优化的认识。这些经验不仅对我的学习有所裨益，也为将来在计算机系统领域的深入研究和应用打下了坚实的基础。

指导教师批阅意见:

成绩评定:

指导教师签字:

年 月 日

备注:

注: 1、报告内的项目或内容设置, 可根据实际情况加以调整和补充。  
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。