# 北京郵電大學


# 实验三：
# 使用 MIPS 指令实现求两个数组的点积

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

学号：2022211683

姓名：张晨阳

2025 年 4 月 21 号

# 目录

# 1. 实验目的

（1） 通过实验熟悉实验 1 和实验 2 的内容

（2） 增强汇编语言编程能力

（3） 学会使用模拟器中的定向功能进行优化

（4） 了解对代码进行优化的方法

# 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

## 3.1. MIPS 汇编语言实现

MIPS 汇编语言是一种简化的 RISC 架构，用于进行低级编程。在本实验中，我们将编写一个 MIPS 汇编程序，计算两个向量的点积。

具体步骤如下：

- **数据加载**：程序开始时，将向量 A 和 B 的地址加载到寄存器中。
- **循环计算**：通过一个循环，程序按顺序将向量的每一对元素加载到寄存器中，计算它们的乘积，并将结果累加到最终的点积中。
- **结束计算**：当循环遍历完所有向量元素时，将累加结果存储到指定的内存位置，并结束程序执行。

## 3.2. 定向功能与静态调度

定向功能用于跟踪程序的每个阶段，确保程序执行的每个步骤都能按预期进行。通过定向功能，我们可以识别出程序中的潜在瓶颈，尤其是指令间的数据依赖。

静态调度是通过手动重排指令序列，避免不必要的等待和资源冲突，从而减少相关性。例如，可以将不相关的指令移动到其他指令之间，以减少处理器的空闲时间，从而提高程序的执行效率。

## 3.3. 优化原理

在实验中，我们使用静态调度方法优化程序。通过分析程序中的指令依赖性，识别并消除不必要的延迟，使得程序在执行时更高效。优化后的程序通过减少流水线冲突和空闲时间，实现了更快的执行速度。

- **优化目标**：减少指令间的等待时间，优化数据访问顺序，避免数据冒险。
- **优化策略**：通过重排指令，确保在计算当前向量元素的乘积时，其他操作不会干扰执行。通过调整循环内的指令顺序，优化向量元素的访问顺序，减少不必要的内存访问和寄存器操作。

# 4. 源程序代码及注释说明

```
.text
main:
    ADDIU $r1, $r0, A      # 将向量 A 的地址加载到寄存器$r1
    ADDIU $r2, $r0, B      # 将向量 B 的地址加载到寄存器$r2
    ADDIU $r3, $r0, ans    # 将结果存储地址加载到寄存器$r3
    ADDIU $r4, $r0, n      # 将向量维度 n 的地址加载到寄存器$r4
    LW $r4, 0($r4)         # 从内存中加载向量的维度（n 值）
    ADDIU $r5, $r0, 0      # 初始化循环索引 i 为 0
    ADDIU $r6, $r0, 0      # 初始化累加器$r6 为 0，用于存储结果

loop:
    LW $r7, 0($r1)         # 从向量 A 中加载第 i 个元素到$r7
    LW $r8, 0($r2)         # 从向量 B 中加载第 i 个元素到$r8
    MUL $r9, $r7, $r8      # 计算 A[i] * B[i]，结果存储到$r9
    ADD $r6, $r6, $r9      # 将计算结果累加到累加器$r6 中

    ADDI $r1, $r1, 4       # 将$r1 移动到向量 A 的下一个元素
    ADDI $r2, $r2, 4       # 将$r2 移动到向量 B 的下一个元素
    ADDI $r5, $r5, 1       # 索引 i 加 1

    BNE $r5, $r4, loop     # 如果索引 i 不等于向量的维度 n，继续循环

    SW $r6, 0($r3)         # 将最终的点积结果存储到内存地址 ans 中

    # 程序结束标志
    TEQ $r0, $r0           # 比较寄存器$r0 与自身，结束程序

# 数据部分
.data
    A: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10   # 向量 A 的数据
    B: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1   # 向量 B 的数据
    n: .word 10                               # 向量的维度 n
    ans: .word 0                              # 用于存储计算结果的内存地址
```

## 1. 程序概述

本程序使用 MIPS 汇编语言编写，目的是计算两个向量 A 和 B 的点积。程序假设向量 A 和 B 的维度 n = 10。向量数据存储在内存中，每个向量元素占用 4 字节（32 位）。程序通过循环逐个计算每个元素的乘积并累加，最终将点积结果存储在内存中的指定位置。

## 2. 程序结构

**(1) 初始化阶段**

- **加载向量数据地址**：首先，程序通过 ADDIU 指令将向量 A 和 B 的地址加载到寄存器 $r1 和 $r2 中，准备访问向量元素。$r3 被用来存储点积的结果地址。
- **加载维度**：程序使用 LW 指令从内存中加载向量维度 n 的值到寄存器 $r4，这个值将用于控制循环次数。

**(2) 循环计算**

- **加载向量元素**：每次循环，程序会使用 LW 指令将向量 A 和 B 的当前元素加载到寄存器 $r7 和 $r8。
- **乘法计算**：程序使用 MUL 指令计算当前元素的乘积 $A[i] \times B[i]$，结果存储在寄存器 $r9。
- **累加结果**：将每次的乘积结果加到累加器 $r6 中，最终 $r6 将存储所有乘积之和，即点积结果。

**(3) 循环控制**

- **更新指针和索引**：通过 ADDI 指令更新指针 $r1 和 $r2，移动到下一个向量元素。寄存器 $r5 用来记录当前元素的索引，每次循环加 1。
- **判断是否继续**：BNE 指令判断索引 $r5 是否等于维度 $r4，如果不相等，则继续执行循环。

**(4) 结果存储与结束**

- **存储结果**：在循环结束后，程序通过 SW 指令将最终的点积结果存储到内存中的 ans 位置。
- **程序结束**：通过 TEQ 指令比较寄存器 $r0 与自身，作为程序的结束标志。

## 3. 数据部分

- **向量 A**：存储了 10 个整数，表示向量 A 的元素。
- **向量 B**：存储了 10 个整数，表示向量 B 的元素。
- **维度 n**：存储向量的维度，值为 10。
- **结果存储区 ans**：用于存储计算后的点积结果。

# 5. 优化后程序说明

```
.text
main:
    ADDIU $r1, $r0, A      # 将向量 A 的地址加载到寄存器$r1
    ADDIU $r2, $r0, B      # 将向量 B 的地址加载到寄存器$r2
    ADDIU $r3, $r0, ans    # 将结果存储的地址加载到寄存器$r3
    ADDIU $r4, $r0, n      # 将向量维度 n 的地址加载到寄存器$r4
    LW $r4, 0($r4)         # 从内存中加载向量维度 n 的值到寄存器$r4
    ADDIU $r5, $r0, 0      # 初始化循环索引 i 为 0，存储在寄存器$r5
    ADDIU $r6, $r0, 0      # 初始化结果寄存器$r6 为 0，用于存储点积结果

loop:
    LW $r7, 0($r1)         # 从向量 A 中加载第 i 个元素（A[i]）到寄存器$r7
    LW $r8, 0($r2)         # 从向量 B 中加载第 i 个元素（B[i]）到寄存器$r8

    MUL $r9, $r7, $r8      # 计算 A[i] * B[i]，结果存储在寄存器$r9

    ADDI $r1, $r1, 4   # 移动到向量 A 的下一个元素，4 字节为一个整数的大小
    ADDI $r2, $r2, 4       # 移动到向量 B 的下一个元素，同样是 4 字节
    ADDI $r5, $r5, 1       # 索引 i 加 1

    ADD $r6, $r6, $r9      # 将计算的 A[i] * B[i]的结果累加到寄存器$r6
中，得到点积结果

    BNE $r5, $r4, loop     # 如果索引 i 不等于向量维度 n，继续循环，执行下
一次迭代

    SW $r6, 0($r3)         # 将点积结果存储到内存地址 ans 中

    # 结束程序
    TEQ $r0, $r0           # 比较寄存器$r0 与自身，作为程序结束的标志

# 数据部分
.data
    A: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  # 向量 A 的元素
    B: .word 10, 9, 8, 7, 6, 5, 4, 3, 2, 1  # 向量 B 的元素
    n: .word 10                             # 向量的维度 n
    ans: .word 0                            # 存储点积结果的内存地址
```

## 1. 程序结构

### (1) 初始化阶段

- **加载向量数据地址：**程序首先使用 ADDIU 指令将向量 A 和 B 的地址加载到寄存器 $r1 和 $r2 中。寄存器 $r3 用于存储最终的点积结果的内存地址。
- **加载向量维度：**通过 LW 指令将向量的维度 n 从内存中加载到寄存器 $r4，用来控制循环的次数。

### (2) 循环计算

优化前的程序会在每次循环迭代中，等待乘法结果才能进行加法计算，并且每次都需要更新内存地址指针。

优化后的程序将加载数据、计算乘法、累加结果以及指针更新进行了合理安排，尽量减少指令之间的等待时间。

- **加载向量元素：**每次循环中，使用 LW 指令分别从向量 A 和 B 中加载当前元素到寄存器 $r7 和 $r8。
- **乘法计算：**计算当前元素的乘积 $A[i] \times B[i]$，结果存储在寄存器 $r9。
- **指针更新与索引增加：**通过 ADDI 指令分别更新向量 A 和 B 的地址，移动到下一个元素，同时更新索引寄存器 $r5。
- **累加结果：**将每次计算的乘积加到结果寄存器 $r6 中，通过 ADD 指令将乘积累加到之前的结果中。

### (3) 循环控制

- **继续循环判断：**通过 BNE 指令检查索引 $r5 是否与维度 $r4 相等。如果不相等，继续执行循环，直到所有元素计算完毕。

### (4) 结果存储与程序结束

- **存储结果：**在循环结束后，程序通过 SW 指令将计算的点积结果存储到内存地址 ans 中。
- **结束程序：**通过 TEQ 指令比较寄存器 $r0 与自身，标记程序结束。

## 2. 数据部分

- **向量 A：**存储了 10 个整数，表示向量 A 的元素。
- **向量 B：**存储了 10 个整数，表示向量 B 的元素。
- **维度 n：**存储向量的维度，值为 10。

- **结果存储区 ans**：用于存储计算后的点积结果。

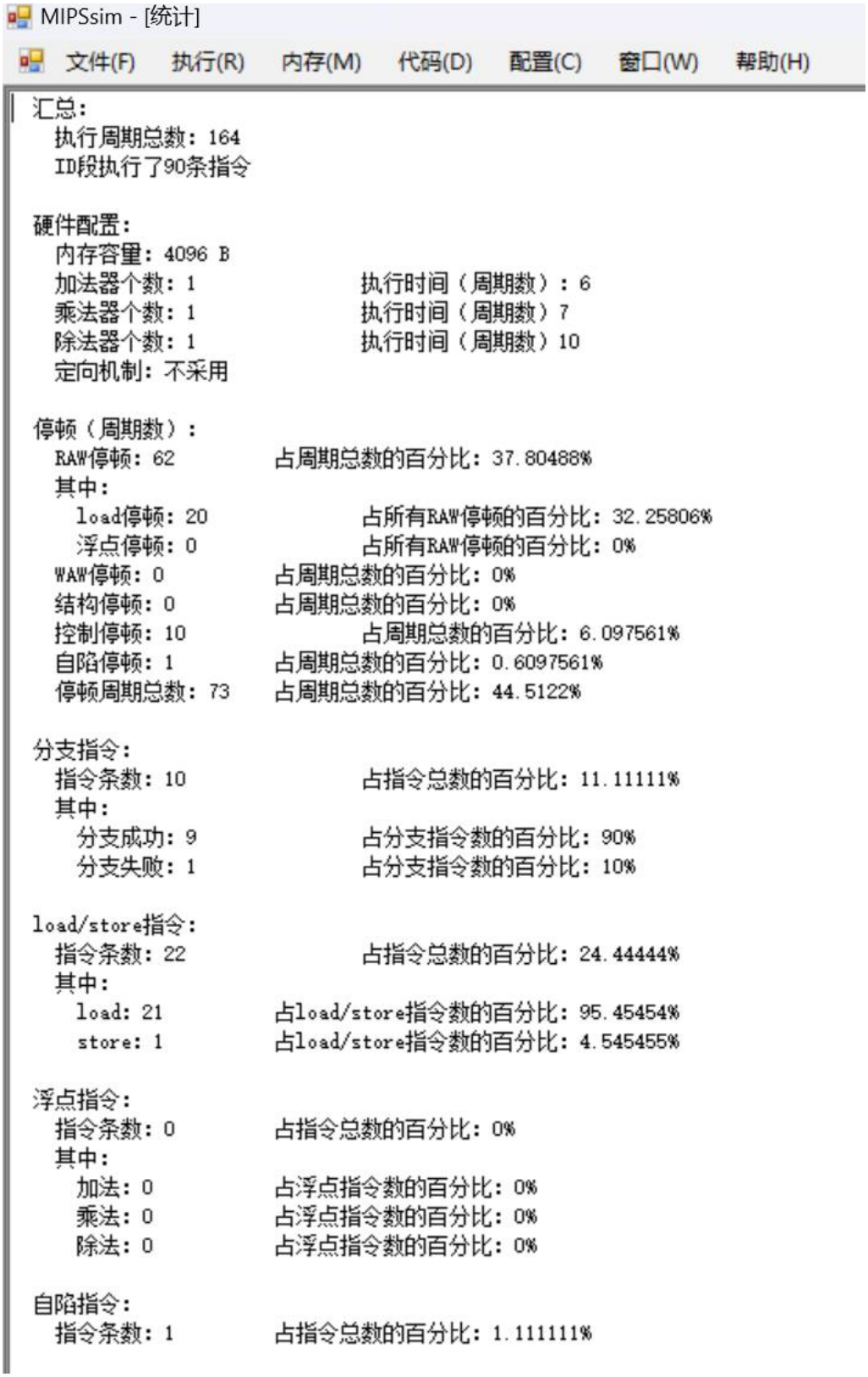
## 3. 优化内容与策略

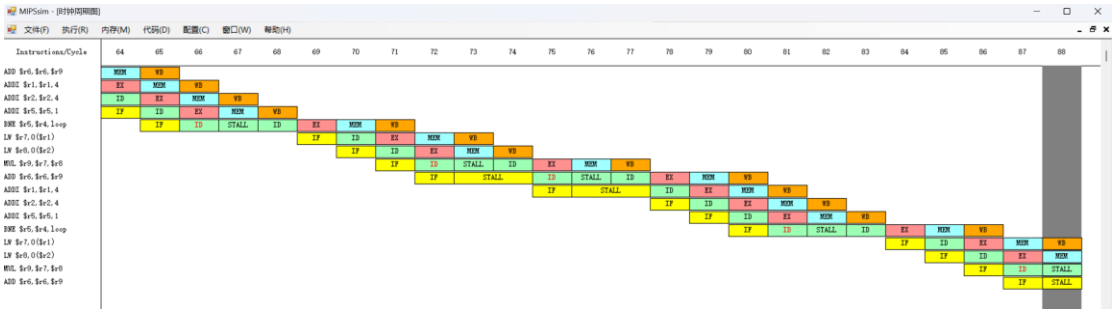
优化后的代码在指令的重排上做了以下优化：

- **减少数据依赖和等待**：通过将加载数据、乘法计算和累加等操作分开并重排顺序，尽量避免了指令间的依赖等待。例如，在计算乘积的同时更新指针，使得下一轮的计算可以尽快进行，减少了流水线等待。
- **优化指令顺序**：通过将不相关的指令重排，减少了乘法操作与加法操作之间的依赖，使得计算过程更加流畅，提高了指令的并行性。
- **指令间并行执行**：通过合理安排指令，允许不同的指令在不同的流水线阶段并行执行，最大程度地减少了流水线冲突和指令延迟。

# 6. 性能分析与比较

## 6.1. 优化前-未开启定向

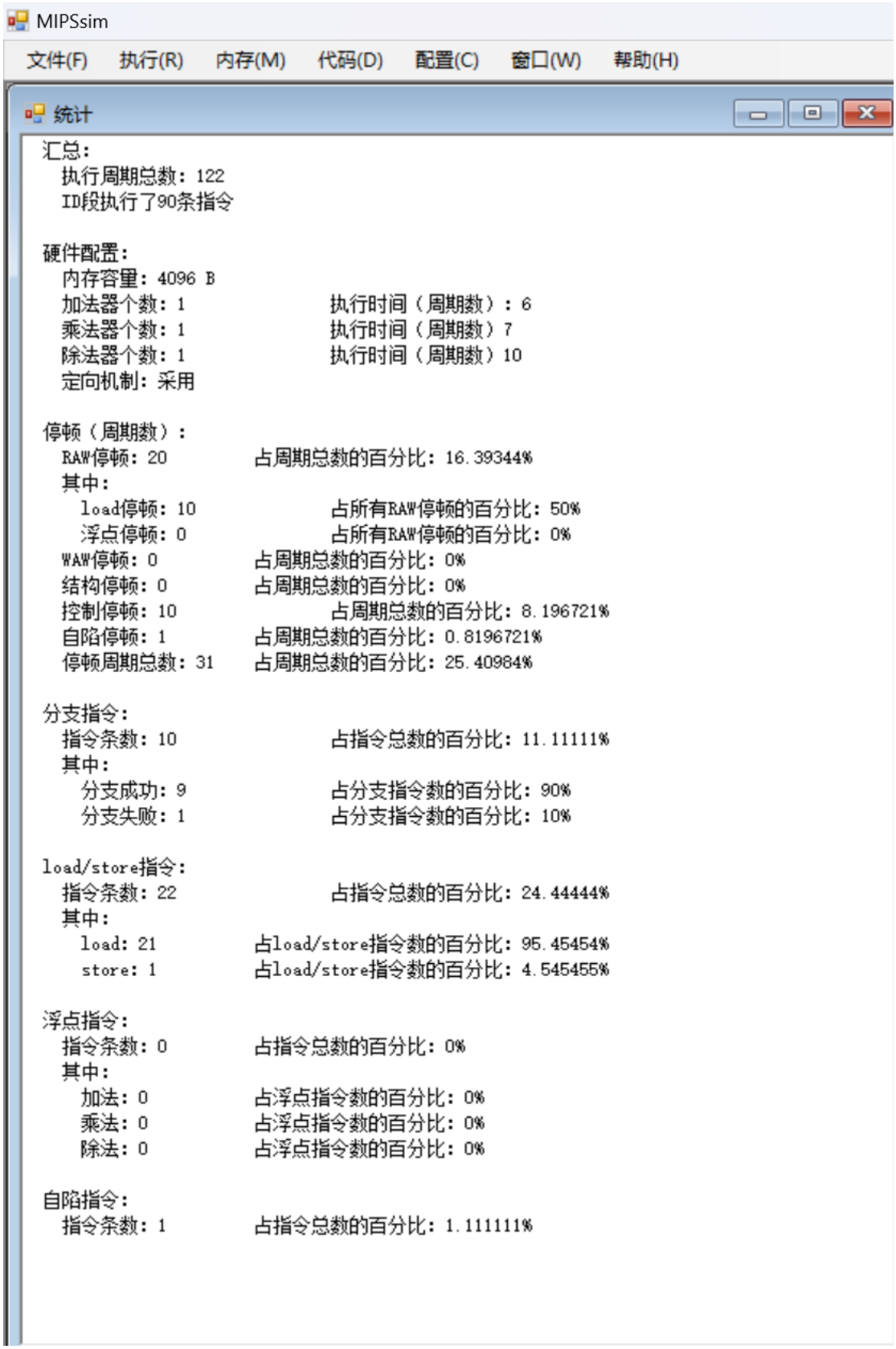


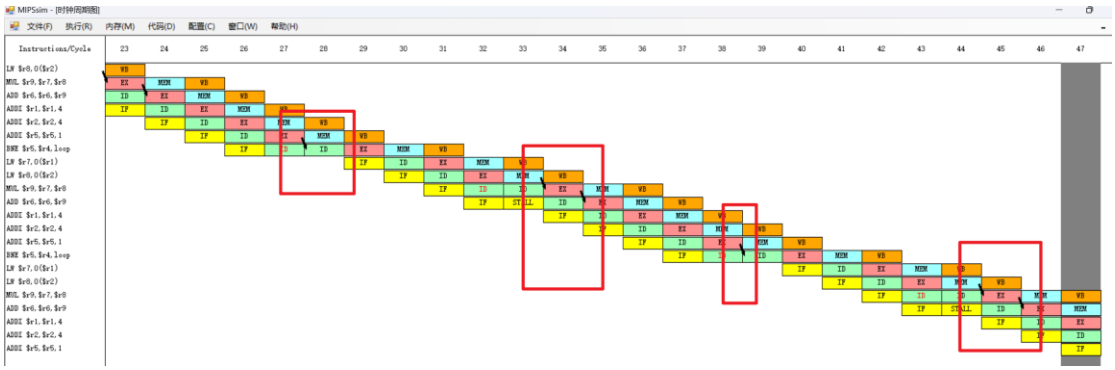



根据 STALL 的时钟周期，不难发现，主要的数据相关来自：

- MUL $r9, $r7, $r8 和 ADD $r6, $r6, $r9
- LW $r8, 0($r2) 和 MUL $r9, $r7, $r8

## 6.2. 开启定向功能

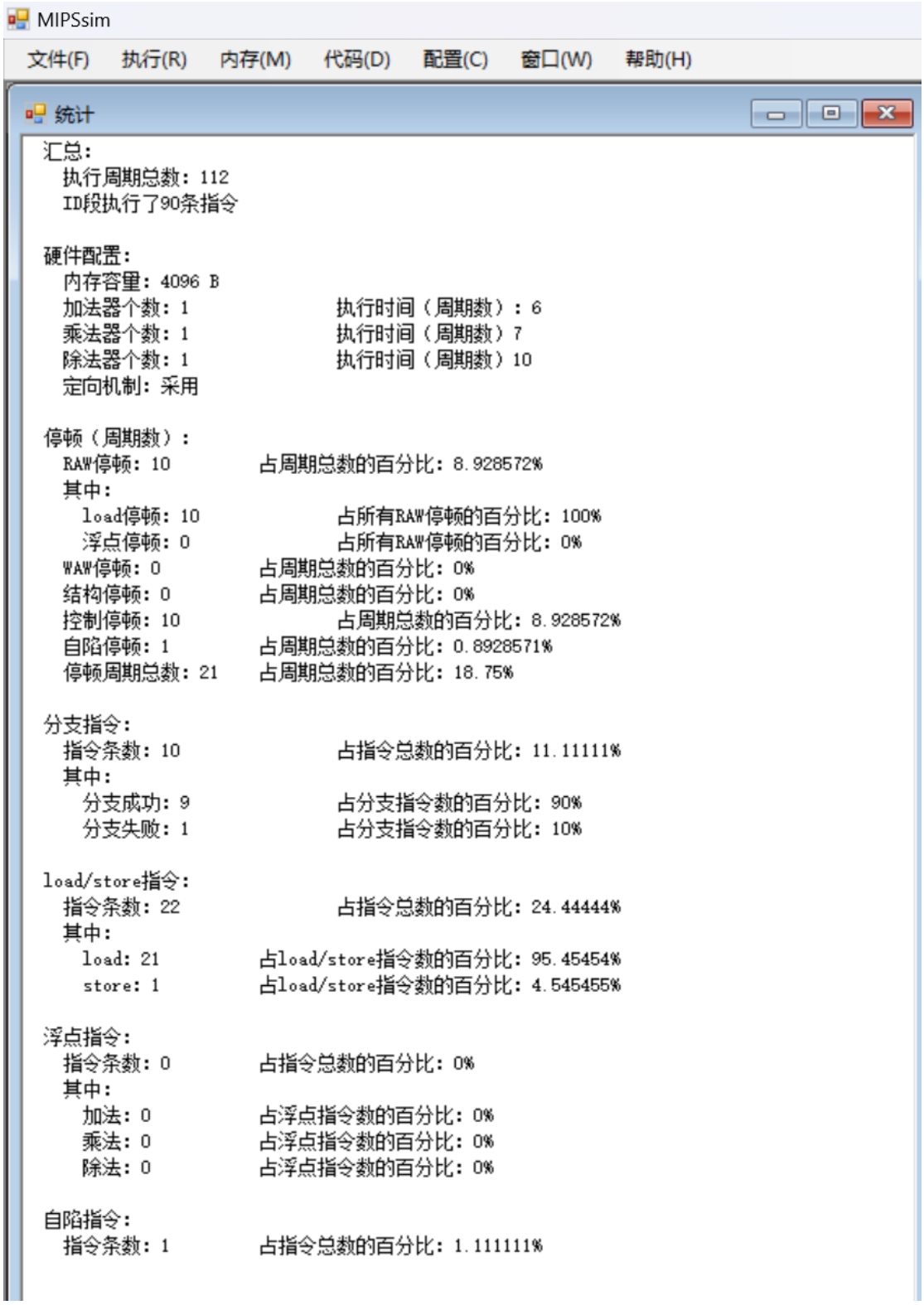

MIPSsim
文件(F) 执行(R) 内存(M) 代码(D) 配置(C) 窗口(W) 帮助(H)
统计
汇总:
执行周期总数: 122
ID段执行了90条指令
硬件配置:
内存容量: 4096 B
加法器个数: 1 执行时间（周期数）: 6
乘法器个数: 1 执行时间（周期数）7
除法器个数: 1 执行时间（周期数）10
定向机制: 采用
停顿（周期数）:
RAW停顿: 20 占周期总数的百分比: 16.39344%
其中:
load停顿: 10 占所有RAW停顿的百分比: 50%
浮点停顿: 0 占所有RAW停顿的百分比: 0%
WAW停顿: 0 占周期总数的百分比: 0%
结构停顿: 0 占周期总数的百分比: 0%
控制停顿: 10 占周期总数的百分比: 8.196721%
自陷停顿: 1 占周期总数的百分比: 0.8196721%
停顿周期总数: 31 占周期总数的百分比: 25.40984%
分支指令:
指令条数: 10 占指令总数的百分比: 11.11111%
其中:
分支成功: 9 占分支指令数的百分比: 90%
分支失败: 1 占分支指令数的百分比: 10%
load/store指令:
指令条数: 22 占指令总数的百分比: 24.44444%
其中:
load: 21 占load/store指令数的百分比: 95.45454%
store: 1 占load/store指令数的百分比: 4.545455%
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: 1.111111%

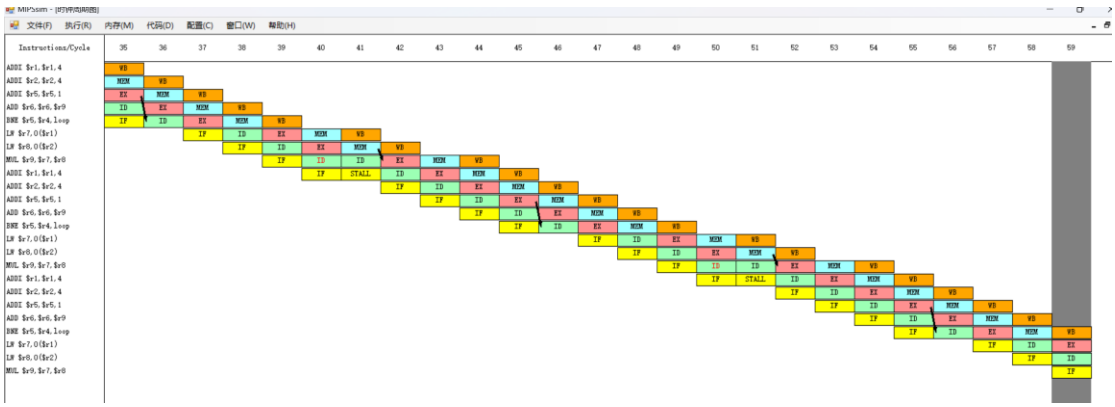


主要的定向内容已在图中标出。

定向功能消除了部分数据冲突，性能提升了约 $164 \div 122 \approx 1.34$ 倍。

## 6.3. 优化后程序


MIPSsim
文件(F) 执行(R) 内存(M) 代码(D) 配置(C) 窗口(W) 帮助(H)
统计
汇总:
执行周期总数: 112
ID段执行了90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: 采用
停顿（周期数）:
RAW停顿: 10 占周期总数的百分比: 8.928572%
其中:
load停顿: 10 占所有RAW停顿的百分比: 100%
浮点停顿: 0 占所有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: 10 占周期总数的百分比: 8.928572%
自陷停顿: 1 占周期总数的百分比: 0.8928571%
停顿周期总数: 21 占周期总数的百分比: 18.75%
分支指令:
指令条数: 10 占指令总数的百分比: 11.11111%
其中:
分支成功: 9 占分支指令数的百分比: 90%
分支失败: 1 占分支指令数的百分比: 10%
load/store指令:
指令条数: 22 占指令总数的百分比: 24.44444%
其中:
load: 21 占load/store指令数的百分比: 95.45454%
store: 1 占load/store指令数的百分比: 4.545455%
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: 1.111111%



与最开始的程序相比，效率大概是 $164 \div 112 \approx 1.46$ 倍，

与仅仅开启定向的程序相比，效率大概是 $122 \div 112 \approx 1.09$ 倍。

从三个执行过程的时钟周期图可以发现，性能提升的原因主要有两点：

（1） 定向技术的使用优化了数据的流动过程；

（2） 数据冲突的避免减少了执行过程中的 STALL。

# 7. 实验总结

通过本次实验，我对 MIPS 汇编语言的编程、流水线模拟以及程序优化有了更深刻的理解。整个实验过程不仅帮助我掌握了如何实现向量点积计算，还使我认识到程序执行效率和优化的重要性。

总之，本次实验不仅提升了我的汇编编程能力，还让我认识到优化对程序性能的影响。在今后的学习中，我将继续深入研究程序优化，力求在设计和开发过程中最大化地提高程序的运行效率。