

# 计算机系统结构实验报告

学号: 19071125

姓名: 卢雨轩

同组人姓名: 孙天天

同组人学号: 19071110

实验报告结构	实验报告内容	实验理解	报告成绩	评语
<input type="checkbox"/> 完全符合要求	<input type="checkbox"/> 充实正确	<input type="checkbox"/> 明确		
<input type="checkbox"/> 基本符合要求	<input type="checkbox"/> 基本正确	<input type="checkbox"/> 较明确		
<input type="checkbox"/> 有比较多的缺陷	<input type="checkbox"/> 有一些问题	<input type="checkbox"/> 基本明确		
<input type="checkbox"/> 完全不符合要求	<input type="checkbox"/> 问题很大	<input type="checkbox"/> 有一定理解		

教师签字: \_\_\_\_\_

日期: \_\_\_\_\_

# 目录

<b>第一部分 流水线、指令调度与循环展开</b>	<b>3</b>
1.1 流水线中的相关 . . . . .	3
1.1.1 实验目的 . . . . .	3
1.1.2 实验内容与步骤 . . . . .	3
1.1.3 实验结果 . . . . .	3
1.2 循环展开和指令调度 . . . . .	4
1.2.1 实验目的 . . . . .	4
1.2.2 实验步骤 . . . . .	4
1.2.3 实验结果 . . . . .	6
1.2.4 循环展开、寄存器换名以及指令调度提高性能 . . . . .	7
1.3 总结与提高 . . . . .	11
<b>第二部分 Cache 性能分析</b>	<b>13</b>
2.1 实验目的 . . . . .	13
2.2 实验内容及步骤 . . . . .	13
2.3 实验方法 . . . . .	13
2.3.1 配置参数 . . . . .	13
2.3.2 测试方法 . . . . .	13
2.4 实验结果 . . . . .	14
2.4.1 块大小实验 . . . . .	14
2.4.2 相联度实验 . . . . .	14
2.4.3 总容量实验 . . . . .	15
2.4.4 替换方法实验 . . . . .	15
2.5 总结与提高 . . . . .	15

# 第一部分 流水线、指令调度与循环展开

## 1.1 流水线中的相关

### 1.1.1 实验目的

1. 熟练掌握 WinMIPS64 模拟器的操作和使用，熟悉 MIPS 指令集结构及其特点；
2. 加深对计算机流水线基本概念的理解；
3. 进一步了解 MIPS 基本流水线各段的功能以及基本操作；
4. 加深对数据相关、结构相关的理解，了解这两类相关对 CPU 性能的影响；
5. 了解解决数据相关的方法，掌握如何使用定向技术来减少数据相关带来的暂停。

### 1.1.2 实验内容与步骤

1. 用 WinMIPS64 模拟器执行三个程序，分别以步进、连续的方式运行程序，观察程序在流水线中的执行情况，观察 CPU 中寄存器和存储器的内容。熟练掌握 WinMIPS64 的操作和使用。
2. 用 WinMIPS64 运行程序 test\_for.s，通过模拟找出存在资源相关的指令对以及导致资源相关的部件；记录由资源相关引起的暂停时钟周期数，计算暂停时钟周期数占总执行周期数的百分比；论述资源相关对 CPU 性能的影响，讨论解决资源相关的方法。
3. 在不采用定向技术的情况下（去掉 Configuration 菜单中 Enable Forwarding 选项前的勾选符），用 WinMIPS64 运行程序 sum.s 和 test\_for.s。记录数据相关引起的暂停时钟周期数以及程序执行的总时钟周期数，计算暂停时钟周期数占总执行周期数的百分比。在采用定向技术的情况下（勾选 Enable Forwarding），用 WinMIPS64 再次运行程序 sum.s 和 test\_for.s。重复上述 3 中的工作，并计算采用定向技术后性能提高的倍数。

### 1.1.3 实验结果

#### test\_for.s 的数据相关

test\_for.s 的代码为：

```
1 .data
2 a: .space 48
3 b: .word 10,11,12,13,0,1
4 c: .word 1,2,3,4,5,6
5
6 .text
7 ;initialize registers
8 addi r1,r0,a
```

```

9  addi r2,r0,b
10 addi r3,r0,c
11 addi r4,r0,6
12 Loop: lw r5,0(r1) ; element of a
13     lw r6,0(r2) ; element of b
14     lw r7,0(r3) ; element of c
15     add r8,r5,r6 ; a[i] + b[i]
16     add r9,r7,r8 ; a[i] = a[i] + b[i] + c[i];
17     sw r9,0(r1) ; store value in a[i]
18     addi r1,r1,8 ; increment memory pointers
19     addi r2,r2,8
20     addi r3,r3,8
21     addi r4,r4,-1 ; i++
22     bnez r4,Loop
23 end: halt

```

运行结果见图 1.1 test\_for.s 运行结果。可以看到，程序一共运行了 86 周期，其中有 6 次数据相关 (Read after Write Stall) 的暂停，占总时钟周期的 7%。数据相关会导致 CPU 空置，会显著影响性能。

程序中第 21 和 22 行对 `r4` 寄存器存在先写后读的数据相关。只需要将 21 行和 20 行交换位置即可解决。修改后的运行结果见图 1.2 test\_for.s 修改后运行结果。

### 定向技术对程序运行的影响

关闭定向技术之后，分别运行 sum.s 和 test\_for.s，发现分别有 4 和 42 个先读后写的数据相关引起的暂停，分别占总周期数量  $6/86 = 30.8\%$  与  $42/122 = 34.4\%$ 。重新启用后运行，发现分别有 1 和 6 个先读后写的暂停，性能分别提高了  $13/10 = 1.3$  与  $122/86 = 1.418$  倍。程序运行结果见第 5 页图 1.3 启用和关闭 Forwarding 对程序运行的影响。

## 1.2 循环展开和指令调度

### 1.2.1 实验目的

1. 加深对循环级并行性、指令调度技术、循环展开技术以及寄存器换名技术的理解
2. 熟悉用指令调度技术来解决流水线中的数据相关的方法；
3. 了解循环展开、指令调度等技术对 CPU 性能的改进。

### 1.2.2 实验步骤

1. 用指令调度技术解决流水线中的结构相关与数据相关
  - (a) 用 DLX 汇编语言编写代码文件 \*.s，程序中应包括数据相关与结构相关 (假设：加法、乘法、除法部件各有 2 个，延迟时间都是 3 个时钟周期)
  - (b) 通过 Configuration 菜单中的“Floatingpointstages”选项，把加法、乘法、除法部件的个数设置为 2 个，把延迟都设置为 3 个时钟周期；

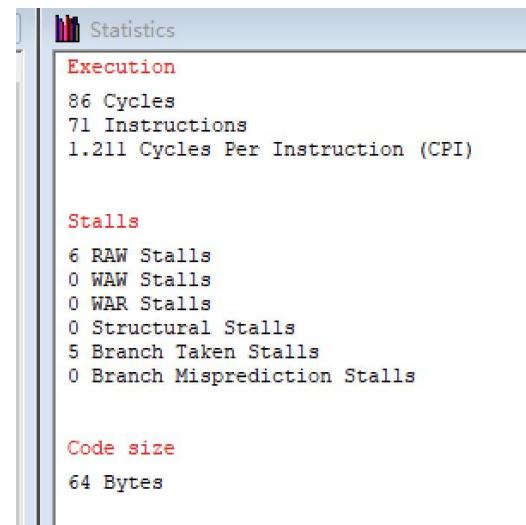


图 1.1: test\_for.s 运行结果

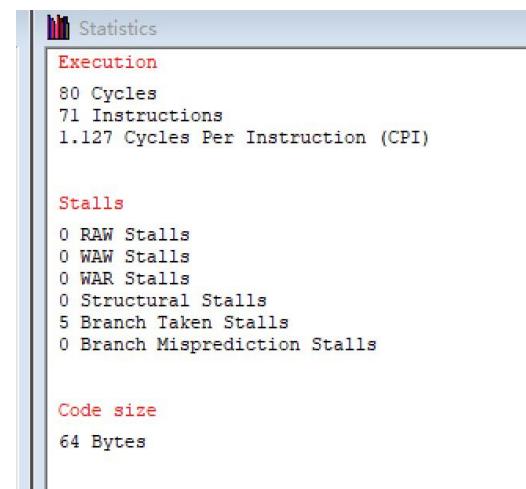


图 1.2: test\_for.s 修改后运行结果

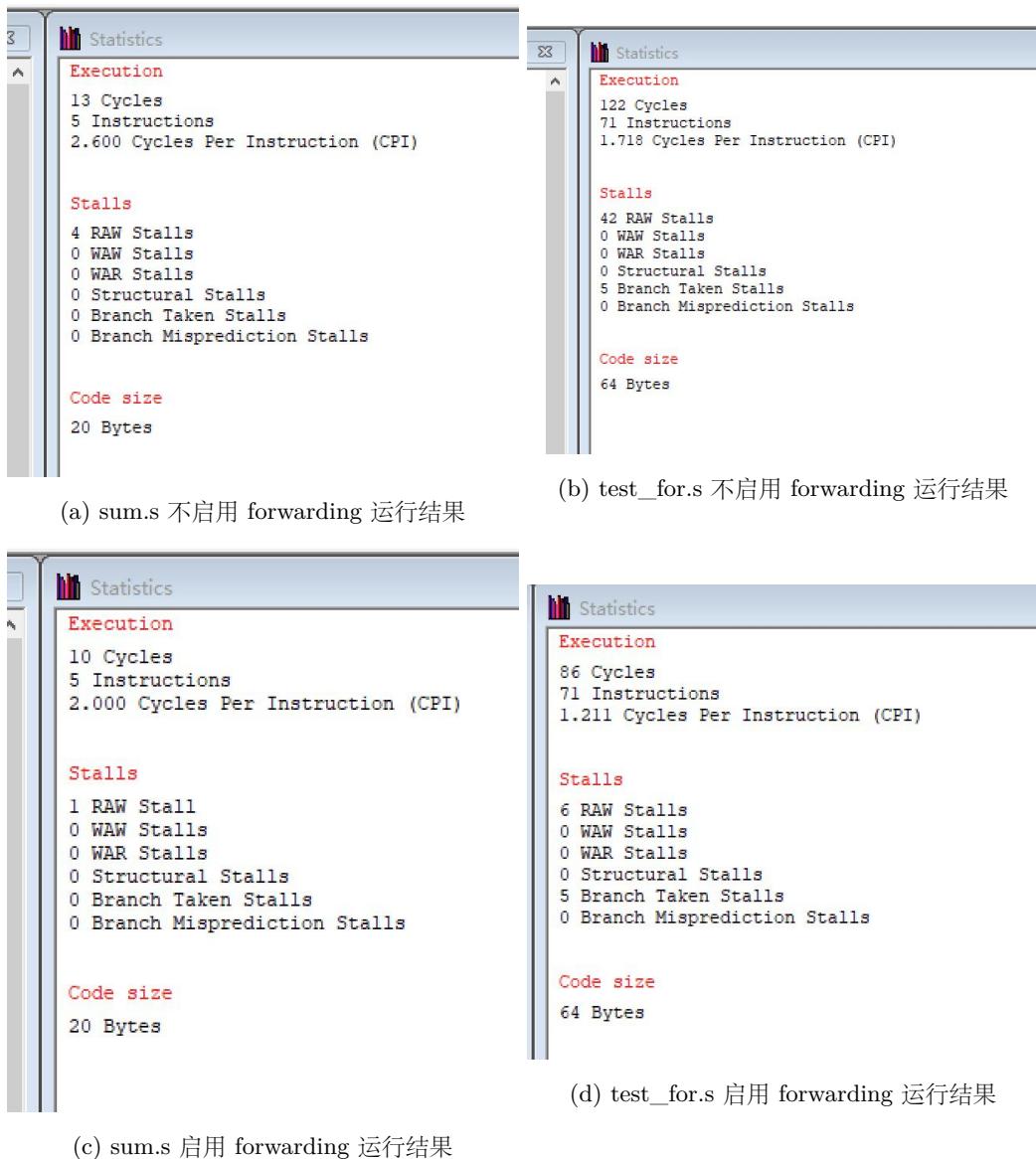


图 1.3: 启用和关闭 Forwarding 对程序运行的影响

- (c) 用 WinDLX/ WinMIPS64 运行程序。记录程序执行过程中各种相关发生的次数、发生相关的指令组合，以及程序执行的总时钟周期数；
- (d) 采用指令调度技术对程序进行指令调度，消除相关；
- (e) 用 WinDLX/ WinMIPS64 运行调度后的程序，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；
- (f) 根据记录结果，比较调度前和调度后的性能。论述指令调度对于提高 CPU 性能的意义。

## 2. 用循环展开、寄存器换名以及指令调度提高性能

- (a) 用 DLX 汇编语言编写代码文件 \*.s，程序中包含一个循环次数为 4 的整数倍的简单循环；
- (b) 用 WinDLX 运行该程序。记录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；
- (c) 将循环展开 3 次，将 4 个循环体组成的代码代替原来的循环体，并对程序做相应的修改。然后对新的循环体进行寄存器换名和指令调度；
- (d) 用 WinDLX/ WinMIPS64 运行修改后的程序，记录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；
- (e) 根据记录结果，比较循环展开、指令调度前后的性能。

### 1.2.3 实验结果

#### 指令调度技术解决结构相关与数据相关

实验中使用的程序为：

```

1 .data
2 title: .asciiz "Test instructions
3     ↳ reordering\n"
4 CONTROL: .word32 0x10000
5 DATA:    .word32 0x10008
6
7 .text
8
9 lwu r21,CONTROL(r0)
10 lwu r22,DATA(r0)
11
12 addi r1, r0, 0
13 addi r2, r0, 1
14 addi r3, r0, 2
15 addi r4, r0, 3
16 addi r5, r0, 4
17 addi r6, r0, 5
18 addi r7, r0, 6
19 addi r8, r0, 7
20
21 ; calculate (r1+r2) * r3 * r4 + r5 * r6 *
22     ↳ r7 * r8
23 dadd r11, r1, r2
24 dmulu r11, r11, r3
25 dmulu r11, r11, r4
26 dmulu r12, r5, r6
27 dmulu r13, r7, r8
28 dadd r11, r11, r12
29
30 addi r24,r0,1      ; integer output
31 sd r11,(r22)       ; should be 1562
32 sd r24,(r21)
33
34 halt

```

程序首先给 `r1 ~r8` 寄存器赋初值，接着计算表达式 $(r1+r2) * r3 * r4 + r5 * r6 * r7 * r8$  的值，最终输出计算结果。

程序运行过程中，在第 24 和 25 行之间有对 `r11` 寄存器的先读后写的数据相关。通过指令调度技术，将第 26 到 27 行移到 24 和 25 行中间，即可解决数据相关：

```

1 .data
2 title: .asciiz "Test instruction
3   ↳ reordering\n"
4 CONTROL: .word32 0x10000
5 DATA:     .word32 0x10008
6
7 .text
8
9 lwu r21,CONTROL(r0)
10 lwu r22,DATA(r0)
11
12 addi r1, r0, 0
13 addi r2, r0, 1
14 addi r3, r0, 2
15 addi r4, r0, 3
16 addi r5, r0, 4
17 addi r6, r0, 5
18 addi r7, r0, 6
19 addi r8, r0, 7
20
21 # calculate (r1+r2) * r3 * r4 + r5 * r6 *
22   ↳ r7 * r8
23 add r11, r1, r2
24 dmulu r11, r11, r3
25 dmulu r12, r5, r6
26 dmulu r13, r7, r8
27 dmulu r11, r11, r4
28 dmulu r12, r12, r13
29
30 # add r8, r6, r7
31
32 addi r24,r0,1      ; integer output
33 sd r11,(r22)    # should be 1562
34 sd r24,(r21)
35
36 halt

```

程序运行结果见第 8 页 图 1.4 指令调度运行结果。可以看到，运行结果相同，减少了 4 个先读后写的数据相关，加速了 3 个周期，同时执行结果正确，意味着指令重排前后程序逻辑一致。

### 1.2.4 循环展开、寄存器换名以及指令调度提高性能

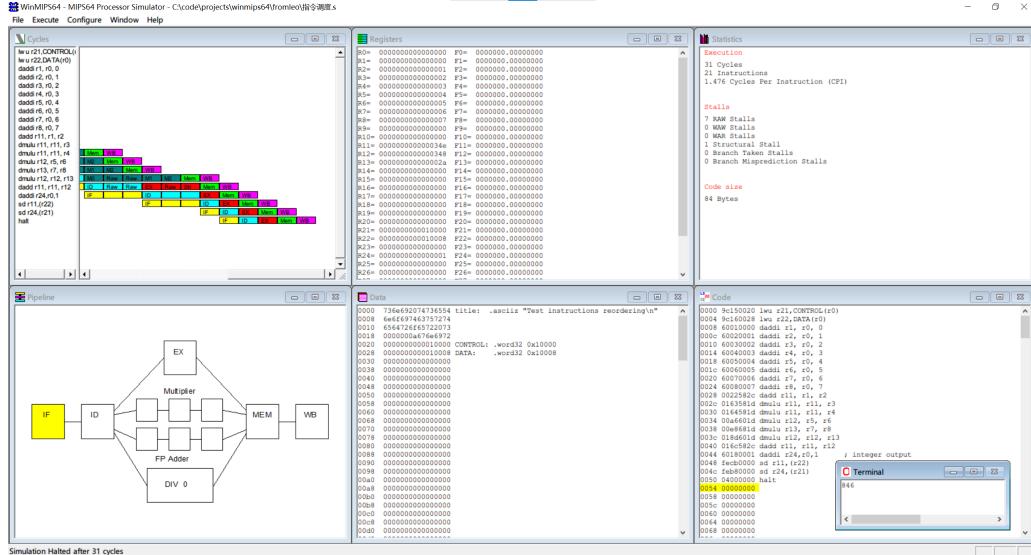
#### 测试程序

循环展开使用的测试程序为计算 fibonacci 数列的第 32 项，代码如下：

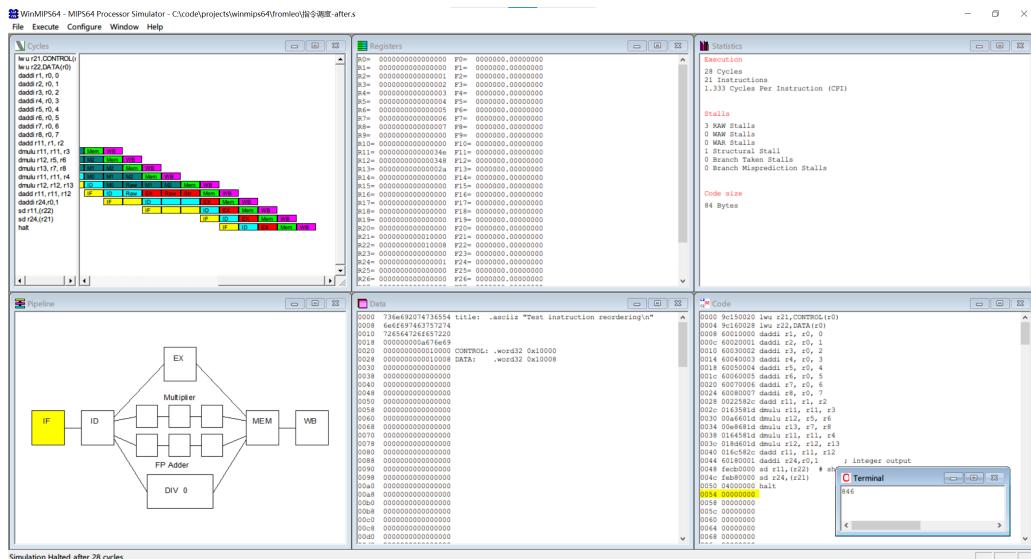
```

1       .data
2 title: .asciiz "Test loop unrolling\n"
3
4 CONTROL: .word32 0x10000
5 DATA:     .word32 0x10008
6
7 .text
8
9 lwu r21,CONTROL(r0)
10 lwu r22,DATA(r0)
11 addi r24,r0,4      ; ascii output
12 addi r1,r0,title
13 sd r1,(r22)
14 sd r24,(r21)
15
16 addi r11, r0, 0 ; f0 = 0
17 addi r12, r0, 1 ; f1 = 1
18
19

```



(a) 指令调度之前的运行结果



(b) 指令调度之后的运行结果

图 1.4: 指令调度运行结果

```

20  daddi r10, r0, 32 ; loop 30 times
21  loop:
22  dadd r13, r12, r11 ; r13 = r12 + r11
23
24  dadd r11, r0, r12 ; r11 = r12
25  dadd r12, r0, r13 ; r12 = r13
26  addi r10, r10, -1 ; r10 -= 1
27
28  bnez r10, loop
29
30  addi r24, r0, 1      ; integer output
31  sd r11, (r22)
32  sd r24, (r21)
33
34  halt

```

代码存在先读后写数据相关与控制语句。展开循环 3 次、寄存器换名、指令重排后的代码如下：

```

1      .data
2  title: .asciiz "Test loop unrolling\n"
3
4  CONTROL: .word32 0x10000
5  DATA:     .word32 0x10008
6
7  .text
8
9  lwu r21, CONTROL(r0)
10  lwu r22, DATA(r0)
11  addi r24, r0, 4      ; ascii output
12  addi r1, r0, title
13  sd r1, (r22)
14  sd r24, (r21)
15
16  addi r11, r0, 0 ; f0 = 0
17  addi r12, r0, 1 ; f1 = 1
18
19
20  addi r10, r0, 32 ; loop 30 times
21  loop:
22  # dadd r13, r12, r11 ; r13 = r12 + r11
23
24  # dadd r11, r0, r12 ; r11 = r12
25  # dadd r12, r0, r13 ; r12 = r13
26  # addi r10, r10, -1 ; r10 -= 1
27
28  dadd r13, r12, r11
29  dadd r14, r12, r13

```

```

30  dadd r15, r13, r14
31  dadd r16, r14, r15
32
33
34  addi r10, r10, -4
35
36  add r11, r0, r15
37  add r12, r0, r16
38
39
40  bnez r10, loop
41
42  addi r24, r0, 1      ; integer output
43  sd r11, (r22)
44  sd r24, (r21)
45
46  halt

```

通过循环展开 3 次，可以在执行过程中减少 75% 的循环控制语句，达到加速执行的目的。同时，使用指令重排技术解决了循环计数器 `r10` 寄存器的先读后写数据相关。

当前程序仍有 4 次的条件跳转等待，可以通过延迟槽技术解决：

```

1      .data
2  title: .asciiz "Test loop unrolling\n"
3
4  CONTROL: .word32 0x10000
5  DATA:    .word32 0x10008
6
7  .text
8
9  lwu r21, CONTROL(r0)
10  lwu r22, DATA(r0)
11  addi r24, r0, 4      ; ascii output
12  addi r1, r0, title
13  sd r1, (r22)
14  sd r24, (r21)
15
16  addi r11, r0, 0 ; f0 = 0
17  addi r12, r0, 1 ; f1 = 1
18
19
20  addi r10, r0, 32 ; loop 30 times
21  loop:
22  # add r13, r12, r11 ; r13 = r12 + r11
23
24  # add r11, r0, r12 ; r11 = r12
25  # add r12, r0, r13 ; r12 = r13

```

```
26 # daddi r10, r10, -1 ; r10 -= 1
27
28 dadd r13, r12, r11
29 dadd r14, r12, r13
30 dadd r15, r13, r14
31 dadd r16, r14, r15
32
33 daddi r10, r10, -4
34 dadd r11, r0, r15
35
36
37 bnez r10, loop
38 dadd r12, r0, r16
39
40 daddi r24,r0,1      ; integer output
41 sd r11,(r22)
42 sd r24,(r21)
43
44 halt
```

通过将一条循环内的计算指令移动到循环跳转指令的延迟槽中执行，可以消除掉条件跳转等待，加快程序运行速度。

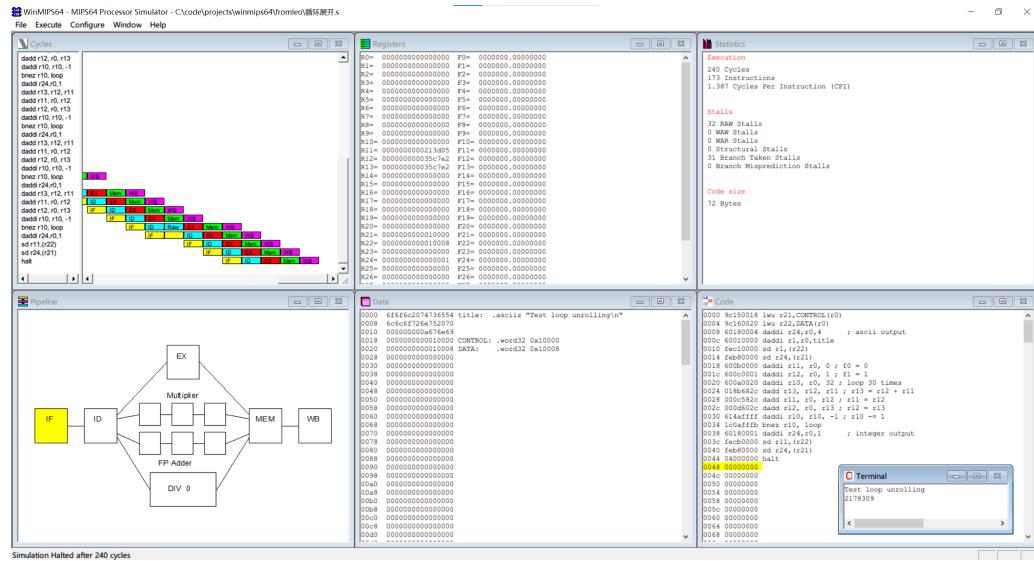
程序运行结果见 第 12 页 图 1.5 循环展开测试程序执行结果。可以看到，程序执行时间分别为 240、88、81 周期，同时执行结果正确，意味着展开前后程序逻辑一致。

### 1.3 总结与提高

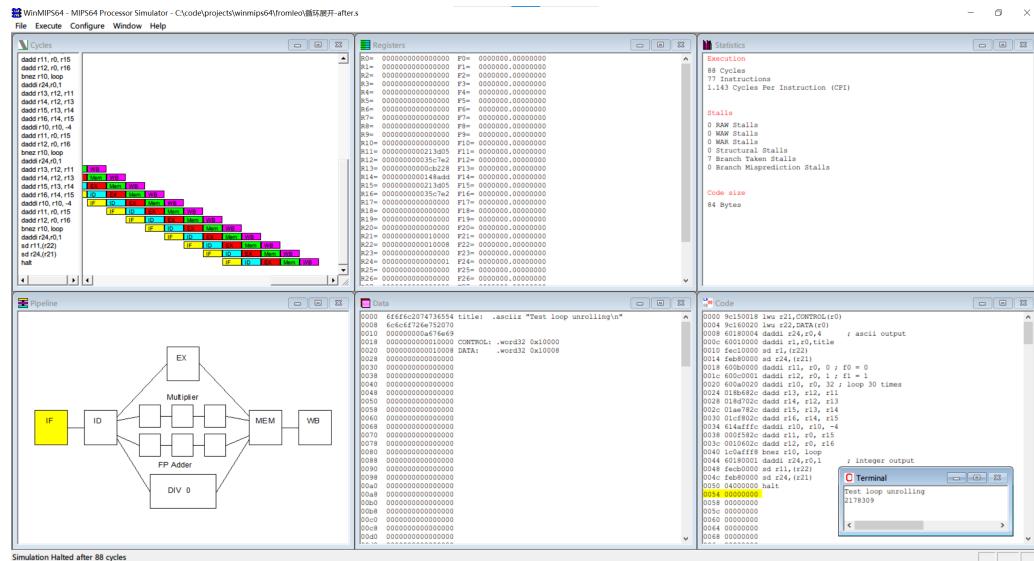
在这次实验中，我了解了各种汇编层面对于程序进行优化的方式。在高中参加信息学竞赛时，由于不开启编译器优化，我曾经简单了解过循环展开等优化方式，有的时候需要手动进行循环展开和函数内联。经过本次实验，我深入了解了循环展开等优化的原理和限制（如展开次数太多会导致寄存器不够用，从而导致效率降低）。

同时，本次实验过程中，我也了解了『过早优化是万恶之源』。在优化调整程序的过程中，多次出现优化后程序运行反而变慢或者没有变快的现象。因此，应该尽可能自动化、程序化的进行优化，这样可以避免人在优化时考虑不周全的问题，同时可以自动化的尝试多种优化方案，让执行效率最高。

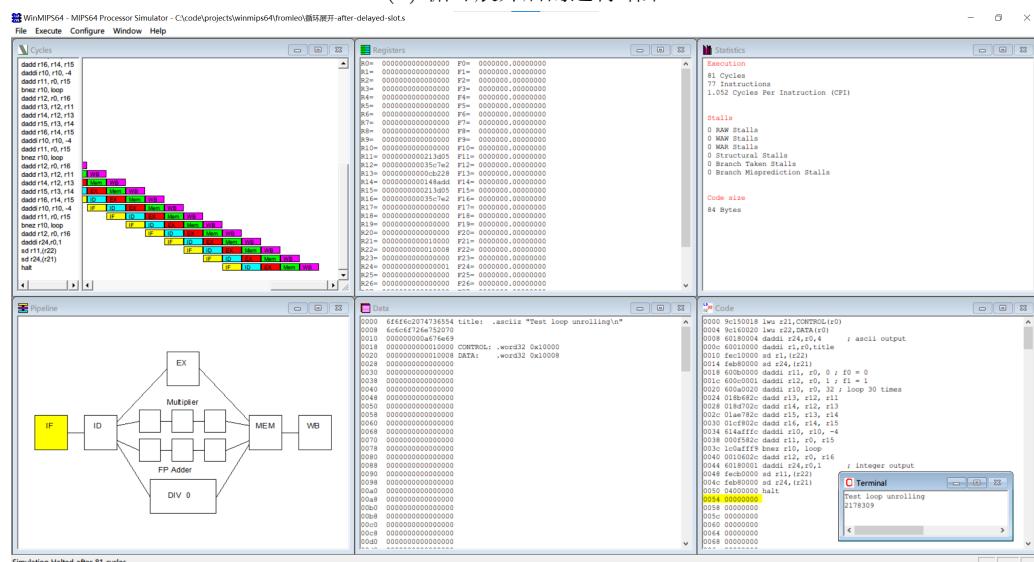
在编写程序的过程中，如果需要最高效率的执行，则可采用平台特定的、深度优化的编译器，如在 Intel CPU 的电脑上使用 `icc` 作为编译器并开启编译优化。这样，可以使得源代码的逻辑尽可能简单，同时让编译之后的程序运行结果尽可能变高。



(a) 循环展开前的运行结果



(b) 循环展开后的运行结果



(c) 循环展开后、启用延迟槽的运行结果

图 1.5: 循环展开测试程序执行结果

## 第二部分 Cache 性能分析

### 2.1 实验目的

1. 加深对 Cache 的基本概念、基本组织结构以及基本工作原理的理解;
2. 了解 Cache 的容量、相联度、块大小对 Cache 性能的影响;
3. 掌握降低 Cache 失效率的各种方法，以及这些方法对 Cache 性能提高的好处;
4. 理解 Cache 失效的产生原因以及 Cache 的三种失效;
5. 理解 LRU 与随机法的基本思想，及它们对 Cache 性能的影响。

### 2.2 实验内容及步骤

1. 在基本配置情况下运行程序 (请指明所选的测试程序)，统计 Cache 总失效次数、三种不同种类的失效次数;
2. 改变 Cache 容量 (\*2, \*4, \*8, \*64)，运行程序 (指明所选的测试程序)，统计各种失效的次数，并分析 Cache 容量对 Cache 性能的影响;
3. 改变 Cache 的相联度 (1 路, 2 路, 4 路, 8 路, 64 路)，运行程序 (指明所选的测试程序)，统计各种失效的次数，并分析相联度对 Cache 性能的影响;
4. 改变 Cache 块大小 (\*2, \*4, \*8, \*64)，运行程序 (指明所选的测试程序)，统计各种失效的次数，并分析 Cache 块大小对 Cache 性能的影响;
5. 分别采用 LRU 与随机法，在不同的 Cache 容量、不同的相联度下，运行程序 (指明所选的测试程序) 统计 Cache 总失效次数，计算失效率。分析不同的替换算法对 Cache 性能的影响。

### 2.3 实验方法

#### 2.3.1 配置参数

鉴于实验指导书中没有给出实验的初始参数，我们遍历了所有和里的参数，找到了一组实验效果较为显著的初始参数，如第 14 页 表 2.1 Cache 实验参数设置 所示。

#### 2.3.2 测试方法

我们使用 sim-cache 对多个程序进行测试，使用脚本自动化程序运行并绘制图表。

实验名称	参数名称	参数意义	初始值	合法范围	测试范围
块大小	块大小	每块的大小	8	$\geq 8$	$\times 1, \times 2, \times 4, \times 8$
相联度	相联度	每组的块数	2	$\geq 2$	$\times 1, \times 2, \times 4, \times 8, \times 16, \times 32, \times 64$
总容量	组数	组数	64	$\geq 2$	$\times 1, \times 2, \times 4, \times 8, \times 16, \times 32, \times 64$
替换算法	替换算法	替换算法	l	l, f, r	l, f, r

表 2.1: Cache 实验参数设置

## 2.4 实验结果

### 2.4.1 块大小实验

块大小实验结果见图 2.1 dl1-块大小实验结果。可以发现，随着块大小的提升，大部分程序的缓存失效次数下降。这可以解释为，块大小提升后，根据局部性原理，程序使用一端数据时，附近更多的数据被一起缓存了，使得使用附近的数据时不需要再去主存加载，使得失效次数降低。

对于失效次数先下降再上升的情况，我们认为这两个程序的内存访问模式和其他程序不太相同，访问的数据更为分散。块大小提升时，首先由于一次性缓存的数据更多使得缓存效率提升；然后则由于程序访问内存非常分散，而块大小提升后块的数量减少了，也就意味着无法处理分散的内存访问，失效次数提高。

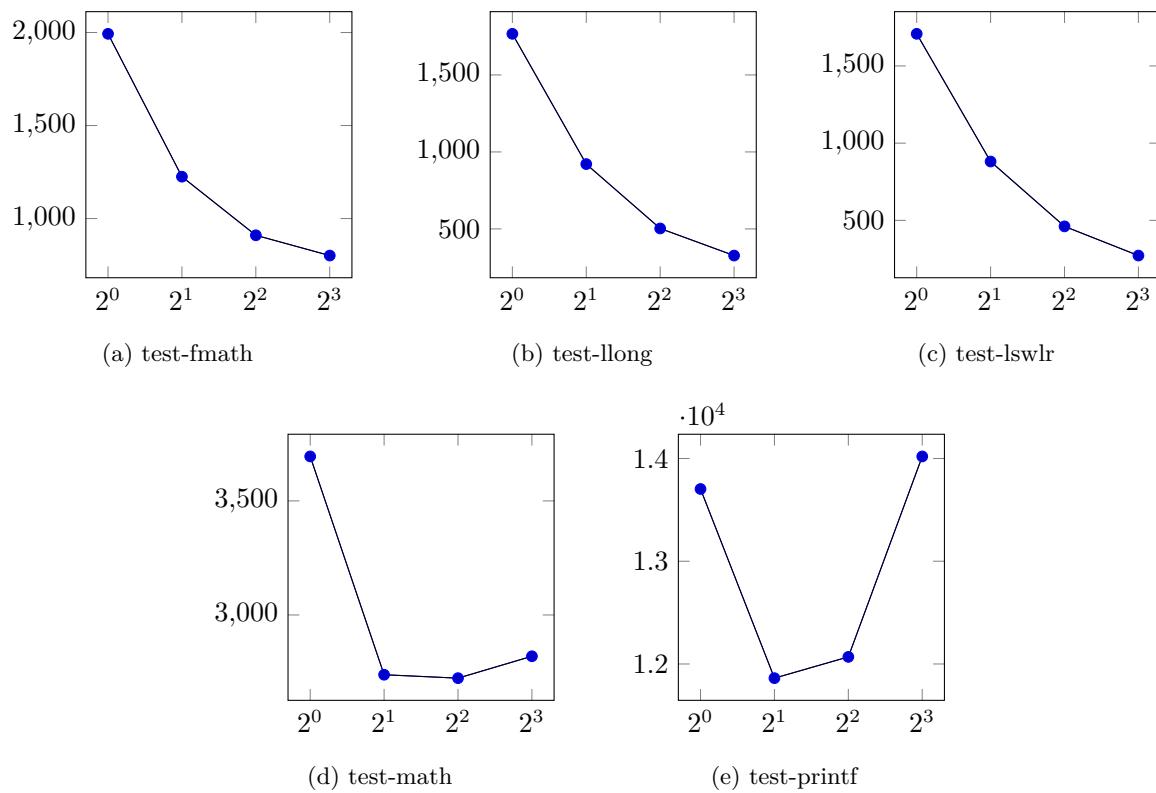


图 2.1: dl1-块大小实验结果

### 2.4.2 相联度实验

相联度试验结果见第 15 页 图 2.2 dl1-相联度实验结果。相联度即组内块数，在相联度实验中，我们为了控制变量，在增大块数的同时减少了总的组数以保证缓存总大小不变。

观察试验结果可以发现，随着相联度的提升，所有程序的命中率均显著提升。这是因为，由于采用组相连算法，组内全相连，即可以缓存内存中任意位置的数据，而组间采用直接相连，即只能缓存固定区域的内存。提升相联度，意味着每一块缓存可以对应更多的内存地址，即分配更为灵活。这就有效的提升了

缓存的利用率，使得失效次数降低。

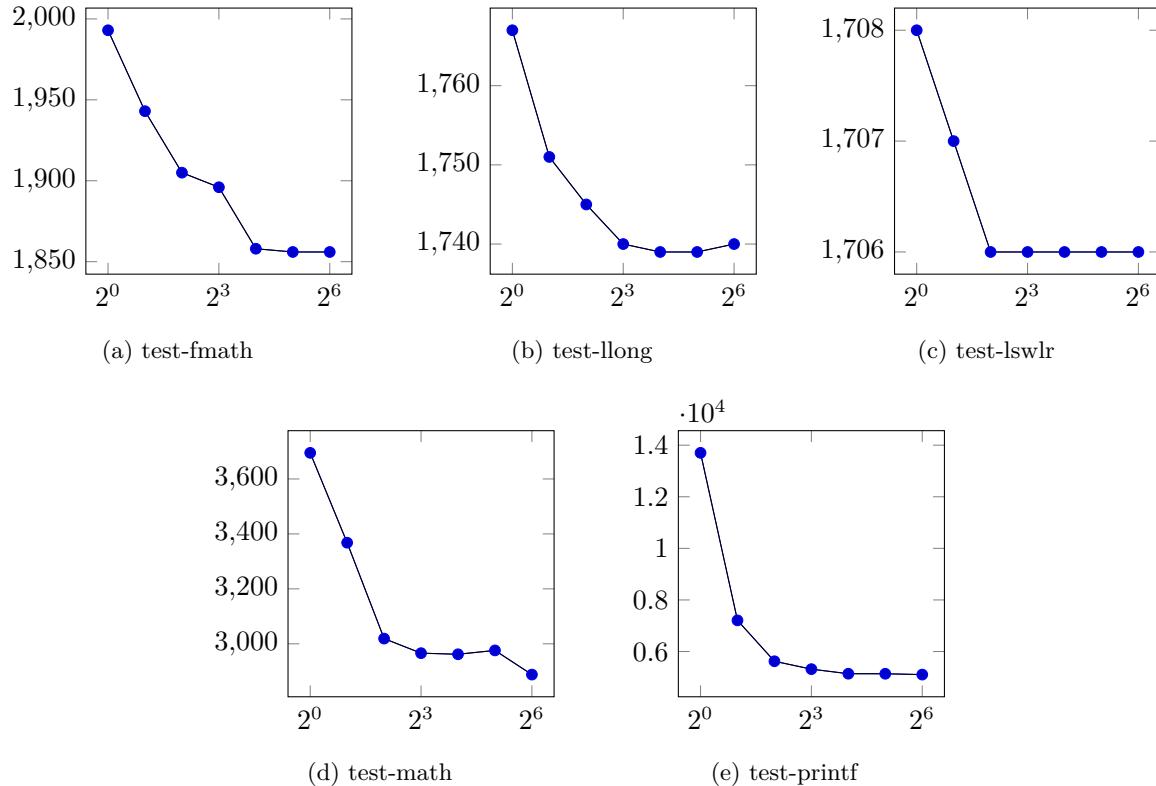


图 2.2: dl1-相联度实验结果

### 2.4.3 总容量实验

总容量试验结果见第 16 页 图 2.3 dl1-总容量实验结果。由图可见，随着总容量增大，程序的缓存失效效率基本都下降，可以看出缓存容量更大后可以缓存更多数据，自然失效次数更少。

对于极小部分情况，缓存提升失效效率并没有显著下降，甚至轻微上升，则是因为那些程序拥有较为特殊的内存访问模式。在数学上可以证明在全相联、使用 LRU 算法时没有 Belady 异常，即随着缓存大小的提升，缓存命中率一定提升。但是，实际使用的是组相联，使得 LRU 算法在极小数情况下也会产生 Belady 异常。

### 2.4.4 替换方法实验

替换方法实验结果见第 16 页 图 2.4 dl1-替换方法实验结果。大多数情况下，LRU、FIFO、Random 的效果依次变差。因为，这三种算法中，LRU、FIFO 都不同程度上考虑了程序的内存访问模式，而 Random 完全不考虑。LRU 认为，很久没用到的内存就不会再用了，某种意义上更符合现实的现象，也更符合局部性原理。FIFO 认为，程序每块内存只会访问一小段时间，和现实中程序访问内存的逻辑更为不符合。

## 2.5 总结与提高

经过本次实验，我印证了系统结构和操作系统课程中学到的程序局部性原理等规则，也充分理解了缓存的各个参数的意义以及对与缓存效率的影响。

在真正设计 CPU 的时候，要考虑到不同程序的内存访问模式。如，Intel 最新的服务器系列 CPU 就提供了搜索引擎优化、数据库优化等不同的版本。只有综合了目标程序的特点，选择对应的参数和算法，才能让整体执行效率最高。

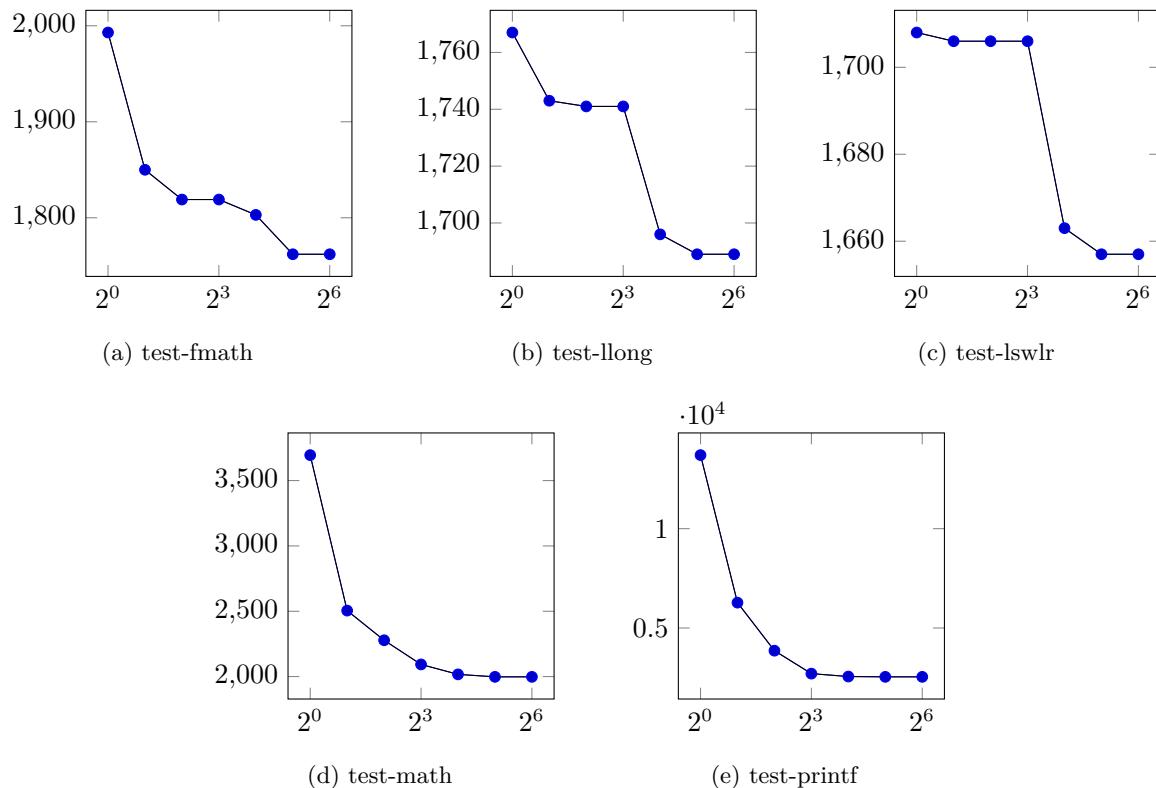


图 2.3: dl1-总容量实验结果

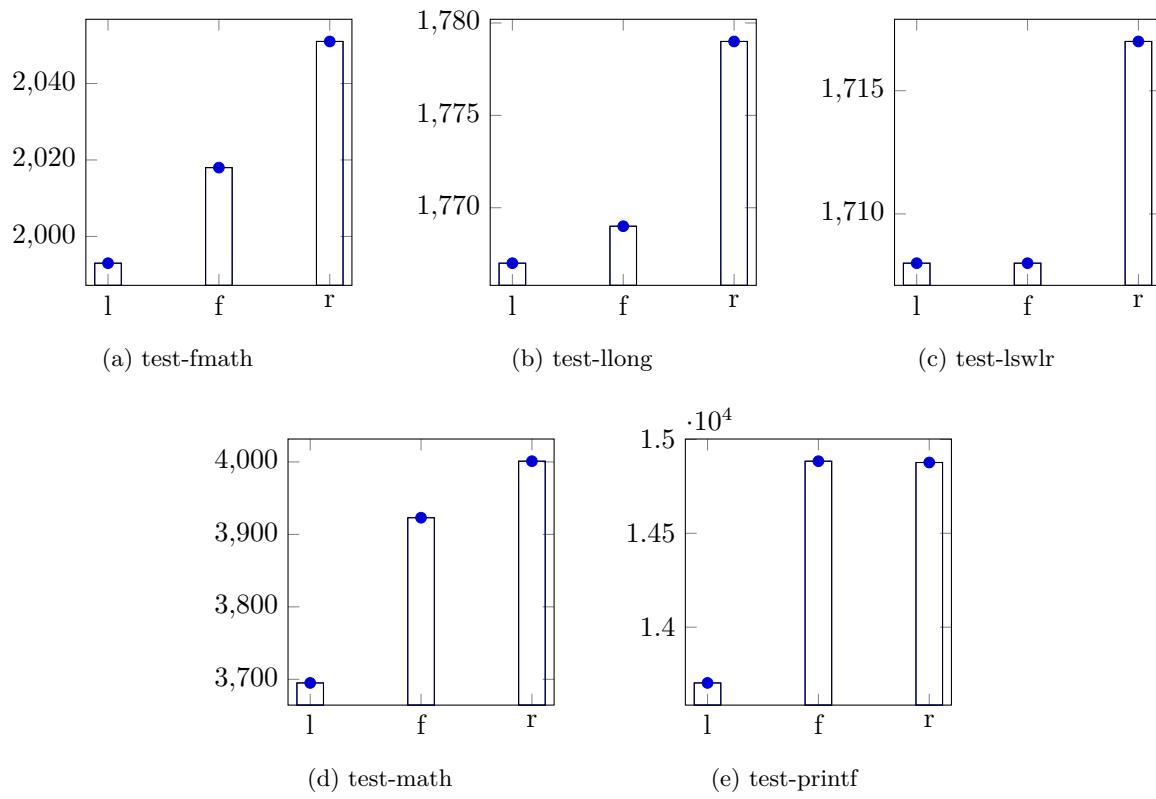


图 2.4: dl1-替换方法实验结果