实验二　循环展开与指令流水

1. 实验目的

（1）加深对指令流水理解。.
（2）掌握利用流水功能的编程技巧。

2. 实验原理

参见《多路处理器实验教学系统实验指导书－计算机系统结构.pdf》P20
附件 1－C 语言嵌入 MIPS 汇编语法说明；
附件 2－MIPS 汇编 printf-scanf 汇编实现。
附件 1 为本实验紧密相关内容，
附件 2 为课外自愿练习 MIPS 汇编程序的“赠送”内容。

（1）无循环级并行

程序 C 代码

```
/* for.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
    float a[200000];
    int i = 0;
    int duration;
    clock_t start, finish;
    start = clock();
        for(i=0; i<200000; i++){
        a[i] = 5.0;
    }
    for(i=0; i<200000; i++){
        a[i] = a[i] + 10.0;
    }
    finish = clock();
    duration = (int)(finish - start)*1000;
    printf("the clock time is: %d\n", duration);
    return 0;
}
```

－－－－－－－－－－－－－－－

```
/* for_asm_s.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void main(){
```

嵌入汇编的 C 代码

```
    float a[200000];
    int i = 0;
    int x=4, c, len=200000;
    int ans;
    int    duration;
    clock_t start, finish;

    for(i=0; i<200000; i++){
        a[i] = 5;
    }
    start = clock();
    __asm__ (
    "li.s       $f2, 10.0\n"
    "Loop:    \n"
    "lwc1      $f0, 0(%[a])\n"
    "add.s     $f0, $f0, $f2\n"
    "swc1      $f0, 0(%[a])\n"
    "addiu    %[a], 4 \n"
    "addiu    %[ans], 0 \n"
    "addiu     %[len], -1\n"
    "bne %[len], $0, Loop\n"
    :[ans]"=r"(ans)
    :[a]"r"(a),[len]"r"(len)
    );
    finish = clock();
    duration = (int)(finish - start)*1000;
    printf("the clock time is: %d\n", duration);
}
```

```
/* for_asm_open.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main(){
    float a[200000];   // 定义数组变量
    int i = 0;
    int c, len=200000;
    int ans;
    int duration;
    clock_t start, finish;   // 定义时间变量

    for(i=0; i<200000; i++){
        a[i] = 5.0;
    }
    start = clock();              // 开始计时
    int iter=20;
    for(i=0;i<iter;i++){

    __asm__(
    "li.s $f2, 10.0\n\t"    // 采用指令 li.s 对 f2 进行赋值，并赋值为 10.0（随便赋值的），在整数类型里面是用 li 来赋值，浮点数类型则使用 li.s。
    "Loop:    \n\t"                  // 设置循环
    "lwc1     $f0,0(%[a])\n\t" // 这一句话是把值赋予给 f0；这里的 a 用于外部参数输入，用%表示；0(%[a])指的是对 a 这个外部参数偏移 0 个字节
    "lwc1     $f6,4(%[a])\n\t" // 这一句话是把值赋予给 f6；这里的 a 用于外部参数输入，用%表示；4(%[a])指的是对 a 这个外部参数偏移 4 个字节
    "lwc1     $f10,8(%[a])\n"   // 这一句话是把值赋予给 f10；这里的 a 用于外部参数输入，用%表示；8(%[a])指的是对 a 这个外部参数偏移 8 个字节
    "lwc1     $f14,12(%[a])\n" // 这一句话是把值赋予给 f14；这里的 a 用于外部参数输入，用%表示；12(%[a])指的是对 a 这个外部参数偏移 12 个字节
    "add.s $f4,$f0,$f2\n"       // 这一句话是把 f0 和 f2 相加，并赋值给 f4
    "add.s $f8,$f6,$f2\n"       // 这一句话是把 f0 和 f2 相加，并赋值给 f8
    "add.s $f12,$f10,$f2\n"    // 这一句话是把 f0 和 f2 相加，并赋值给 f12
    "add.s $f16,$f14,$f2\n"    // 这一句话是把 f0 和 f2 相加，并赋值给 f16
    "swc1     $f4,0(%[a])\n"     //保存结果
    "swc1     $f8,4(%[a])\n"     //保存结果
    "swc1     $f12,8(%[a])\n"   //保存结果
    "swc1     $f16,12(%[a])\n" //保存结果
    "addiu %[len],-4\n"          // 循环次数减 4，因为我们是做了 4 次循环展开
    "addiu %[a],16\n"             // 指针偏移 16 位，因为我们 4 次循环展开，每个数有 4 个字节，4*4=16
    "bne %[len],$0,Loop\n"// 判断是否跳出循环
    :[ans]"=r"(ans)                // 输出，这是 asm 编程需要的，这语句不能删除，但此程序不需要输出，所
```

以这个 ans 没有实际含义

```
    :[a]"r"(a),[len]"r"(len) //  外部输入，把数组和数组长度传入
    );
    }
    finish = clock();              //  结束计时
    duration = (int)(finish - start)/iter;
    printf("the clock time is: %d\n", duration);
}
```

说明：

"for.c"-----------------C 语言：循环数组加法计算

"for_asm_s.c"-------- 嵌入式汇编：无循环展开的数组加法计算

"for_asm_open.c"---- 嵌入式汇编：有四重循环展开的数组加法计算

编译

```
gcc for.c -o for
gcc for_asm_s.c -o forasm
gcc for_asm_open.c -o forasmopen
```

运行

```
./for    ./ forasm    ./ forasmopen
```

4. 实验分析

《多路处理器实验教学系统实验指导书-并行算法实践.pdf》

附件 1：C 语言嵌入 MIPS 汇编语法说明

一、内嵌汇编基本格式

```
asm(
        内嵌汇编指令
        ：输出操作数
        ：输入操作数
        ：破坏描述
);
```

内嵌汇编以 asm()或__asm__(); 格式表示，里面分成 4 个部分，内嵌汇编指令、输出操作数、输入操作数、破坏描述。各部分之间使用“:”分割。其中内嵌汇编指令是必不可少的，但可以为空。其他 3 部分根据程序需要可选。如果只有内嵌汇编指令时，后面的“:”可以省略。

再看下面的内嵌汇编：

```
asm("daddu %0,%1,%2\n\t"
        :"=r"(ret)
        :"r"(a),"r"(b)
);
```

其中"daddu %0,%1,%2\n\t" 就是内嵌汇编指令，指令由指令操作符和指令操作数组成。操作符就使用

MIPS 汇编指令中的助记符，操作数可以是%0,%1,%2 形式的占位符，来表示 c 语言中变量 ret、a 和 b。指令操作数也可以是寄存器。使用寄存器做指令操作数时，寄存器前面需要符号$。例如：

```
asm("move $31,%0\n\t"
    :           /*此处的：不能省略*/
    :"r"(a)
);
```

上面这条指令实现了把 c 语言变量 a 的值存入通用寄存器 ra（$31）。

注意：内嵌汇编程序中如果没有输出部分，但是有输入部分，那么输出部分的“:”不能省略。同时 asm 模板里面可以使用/**/或者//添加注释。

asm 模板里可以有多条内嵌汇编指令。每条指令都以" "为单位。多条指令可以使用" ;"号、\n\t 或者换行来分割。

```
asm("dadd %0,%1\n\t"
    "dsub %0,%2\n\t"
    :"=r"(ret)
    :"r"(a),"r"(b)
);
```

二、输入操作数和输出操作数

内嵌汇编中的操作数包括输出操作数和输入操作数，输出操作数和输入操作数里的每一个操作数都由一个约束字符串和一个带括号的 c 语言表达式或变量组成，比如“r”(src)。多个操作数之间使用“,”分割。内嵌汇编指令中使用%num 的形式依次表示每一个操作数，num 从 0 开始。比如：

```
asm("daddu %0,%1,%2\n\t"
    :"=r"(ret)          /* 输出操作数，也是第 0 个操作数%0 */
    :"r"(a),"r"(b)      /* 输入操作数，也是第 1 个操作数和第 2 个操作数 %1,%2 */
);
```

这里使用了 daddu 指令实现了 c 语言中 ret=a+b 的操作。两个输入操作数"r"(a)和"r"(b)之间使用“,”分割。%0 代表操作数"=r"(ret)、%1 代表操作数"r"(a)、%2 代表操作数"r"(b)。

每个操作数前面的约束字符串表示对后面 c 语言表达式或变量的限制条件。GCC 会根据这个约束条件来决定处理方式。比如"=r"(ret)中的"=g"表示有两个约束条件，"="表明此操作数是输出操作数，"r"(b)中的"r"表示将变量 b 放入通用寄存器（relation 相关联）。约束字符还有很多，有些还与特定体系结构相关，在下一节会详细列举。

输入操作数通常是 c 语言的变量，但是也可以是 c 语言表达式。比如：

```
asm("move %0,%1\n\t"
    :"=r"(ret)
    :"r"(&src+4)
);
```

这里输入操作数 &src+4 就是 c 语言表达式。执行的结果就是把&src+4 的地址赋给 ret。

输出操作数必须是左值，GCC 编译器会对此做检查。左值概念就是以赋值符号 = 为界，= 左边的就是左值，= 右边就是右值。输入操作数可以是左值也可以是右值。所以输出操作数必须使用"="标识自己。同时默认情况下输出操作数必须是只写（write-only）的，但是 GCC 不会对此做检查。这个特性有时会给我们带来麻烦。如果你要在内嵌汇编指令里把输出操作数当右值来操作，GCC 编译时不会报错，但是程序运行后你可能无法得到你想要的结果。为此我们可以使用限制符"+"来把输出操作符的权限改为可读可写。例如：

```
asm("daddu %0,%0,%1\n\t"
    :"+r"(ret)
    :"r"(a)
);
```

这就实现了 ret = ret+a 的操作。 "+r"中的"+"就表示 ret 为可读可写。同时我们也可以使用数字限制符"0"达到修改输出操作符权限的目的。

```
asm("daddu %0,%1,%2\n\t"
     :"=r"(ret)
     :"0"(ret),"r"(a)
);
```

这里数字限制符"0"意思是第 1 个输入操作数 ret 和第 0 个输出操作数使用同样的地址空间。数字限制符只能用在输入操作数部分，而且必须指向某个输出操作数。

三、破坏描述

破坏描述部分就是声明内嵌汇编中有些寄存器被改变。通常内嵌汇编程序中会使用到一些寄存器，并对其做修改。如果在破坏描述部分不做说明，那么 gcc 编译内嵌汇编时不会做任何的检查和保护。这可能就会导致程序出错或致命异常。例如：

```
asm("dadd %0,%1,%2\n\t"
    "move $31,%0\n\t"
    :"=g"(ret)
    :"r"(a),"r"(b)
);
```

上面程序完成 ret=a+b，然后 ret 的值写入寄存器 ra($31)。我们知道寄存器 ra 被用来做函数返回的。但是 ra 被改变，将导致函数无法正常返回。这时就需要在破坏描述部分添加声明来告诉编译器此寄存器的值被改变。MIPS 的内嵌汇编中寄存器的使用以$num 形式，num 代表寄存器编号。在破坏部分声明就使用"$num"形式，多个声明之间使用“,”分开。例如：

```
asm("dadd %0,%1,%2\n\t"
    "move $31,%0\n\t"
    :"=g"(ret)
    :"r"(a),"r"(b)
    :"$31"
);
```

破坏描述符除了寄存器还有“memory”。它的作用见本章最后一节。

四、有名操作数和指定寄存器

从 gcc 的 3.1 版本之后，内嵌汇编支持有名操作数。就是可以在内嵌汇编中为输入操作数、输出操作数取名字，名字形式是[name]，放在每个操作数的前面，然后汇编程序模板里面就可以使用%[name]的形式，而不是上面%num 形式。例如：

```
asm("daddu %[out],%[in1],%[in2]\n\t"
     :[out]"=g"(ret)
     :[in1]"r"(a),[in2]"r"(b)
);
```

这里给 c 语言变量 ret 取名为 out、变量 a 和 b 取名为 int1 和 in2。这里的别名要求可以是大小写字母、数字、下划线等，但是你必须确保同一汇编程序中没有两个操作数使用相同的别名。

当然，你可以仅输出或者输入中的部分操作数去名字。例如：

```
asm("daddu %[out],%1,%2\n\t"
    :[out]"=g"(ret)
    :"r"(a),"r"(b)
);
```

这里我只给第 0 个操作数"=g"(ret)取名字。那么后面的第 1 个操作数和第 2 个操作数在使用时还是序列号形式%1、%2。

有时候我们需要在指令中使用指定的寄存器；比如系统调用时需要将系统调用号放在 v0 寄存器，参数放入 a0-a7 寄存器。那么这是我们可以在 c 语言声明变量时使用指定寄存器功能。例如：

```
register int sys_id asm("$2") = 5001;
```

这里使用关键字 register 声明了一个寄存器变量 sys_id，并通知 GCC 编译器使用$2（v0）寄存器来加载 sys_id 变量。

五、操作数的修饰符：约束字符

约束字符就是输入操作数和输出操作数前面的修饰符。约束字符可以说明操作数是否可以在寄存器中，以及哪种寄存器；操作数是否可以是内存引用，以及哪种地址；操作数是否可以是立即常数，以及它可能具有的值。本节介绍常用的约束字符信息。

“r” 通知汇编器可以使用通用寄存器中的任意一个来加载操作数。最常用的一个约束。

“g” 允许使用任何通用寄存器、内存或立即整数操作数。

“i”通知汇编器这个操作数是个立即数（一个具有常量值）。例如：

```
#define DEFAULT 1
asm("li %0,%1\n\t"
    :"=r"(ret)
    :"i"(-TCP_MSS_DEFAULT)
);
```

此处的"i"也可以使用"g"代替。

“n”同约束字符“i”。

“m”内存操作数，用在访存指令的地址加载和存储。例如

“o”内存操作数，用在访存指令的地址加载和存储。在 MIPS 架构中功能同“m”。

“+”修改操作数的权限为可读可写，通常只修饰输出操作数。

“&”

“I”在 MIPS 架构下用于标识此操作数是一个有符号 16 位的常数。“I”用于算术指令。例如：

```
asm("daddiu $2,$2,%0\n\t" : :"I"(0x3) );
```

七、理解 asm volatile("": : :"memory")的含义

asm volatile("": : :"memory")是我们平时经常遇到的内嵌汇编格式。其中有一个关键字 volatile 和一个破坏描述“memory”。当然这两个关键字不是必须同时出现的，使用时要根据情况。

MIPS 是多级流水线架构。编译器优化就会依赖这个特性在编译时调整指令顺序，让没有相关性的指令可以乱序执行，以充分利用 CPU 的指令流水线，提高执行速度。如果内嵌汇编的指令中直接使用了某些寄存器或内存。GCC 编译器在优化后很可能带来错误。这种情况我们可以使用 volatile 关键字来修饰。volatile 用于告诉编译器，严禁将此处的 asm 汇编语句与它之前和之后的 c 语句在编译时重组合。

如果你的内嵌汇编中使用了一段未知大小的内存，或者使用的内存用于在多线程。那么请务必使用约束字符“memory”。“memory”就是通知 GCC 编译器，此段内嵌汇编修改了 memory 中的内容，asm 之前的 c 代码块和之后的 c 代码块看到的 memory 可能是不一样的，对 memory 的访问不能依赖之前的缓存，需要重新加载。

附件 2—MIPS 汇编 printf-scanf 汇编实现

```
/*helloworld.S:Hello world! from Loongson3A*/

#include <asm/unistd.h>
#define STDOUT 1
#define STDIN 0

/*print macro*/
   .macro print msg,n
   li $a0, STDOUT
   lui $a1,%hi(\msg)
   addiu $a1,$a1,%lo(\msg)
#  la $a1,\msg
   li $a2,\n
   li $v0,__NR_write
   syscall
   .endm

/*exit macro*/
   .macro exit n
   li $a0,0
   li $v0,__NR_exit
   syscall
   .endm

   .set nomips16
   .data
inMsg:.space 255
prompt:  .asciiz  "What's  your
name?\n"
Msg:  .asciiz  "Glad  to  meet
you!\n"

   .text
   .globl main
   .ent main
main:
   print prompt,18
   /*ssize_t     read(int     fd,voud
*buf,size)*/
   li $a0,STDIN
   lui $a1,%hi(inMsg)
   addiu $a1,$a1,%lo(inMsg)
   li $a2,255
   li $v0,__NR_read
   syscall
   print Msg,18
   print inMsg,255

exit: li $a0,0
   li $v0,__NR_exit
   syscall
   .end main

Linux 下编译命令：
gcc -o helloworld helloworld.S
./helloworld
```