第二章 指令系统原理与实例

- 2.1 简介
- 2.2 指令集系统结构的分类
- 2.3 存储器寻址
- 2.4 操作数的类型
- 2.5 指令系统的操作
- 2.6 控制流指令
- 2.7 指令系统的编码
- 2.8 编译器的角色
- 2.9 MIPS系统结构
- 2.10 谬误和易犯的错误
- 2.11 结论

2.6控制流指令

- **控制流指令分类**

  条件转移，跳转，过程调用，过程返回

- **“跳转”（Jump）：当控制指令为无条件改变控制流时，我们称之为“跳转”。**
- **“分支”（Branch）：而当控制指令是有条件改变控制流时，我们称之为“分支”。**
- **控制流程的改变情况：**
  - 条件分支（conditional branch）；
  - 跳转（jump）；
  - 过程调用（call）；
  - 过程返回（return）。

MIPS Branch Operations

- **Branch to a labeled instruction if a condition is true**
  - Otherwise, continue sequentially

  - `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;

  - `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;

Example *if* Statement if.s

- Assuming translations below, compile *if* block

  f → `$s0` g → `$s1` h → `$s2`

  i → `$s3` j → `$s4`

```
if (i == j)          bne $s3,$s4,Exit
  f = g + h;         add $s0,$s1,$s2
                Exit:
```

- May need to negate branch condition

Example *if* simple_branch.asm

```
# C code:
#
# x = 0
 # if (x == 0) {
#   printf("x is zero")
# }
# exit()
```

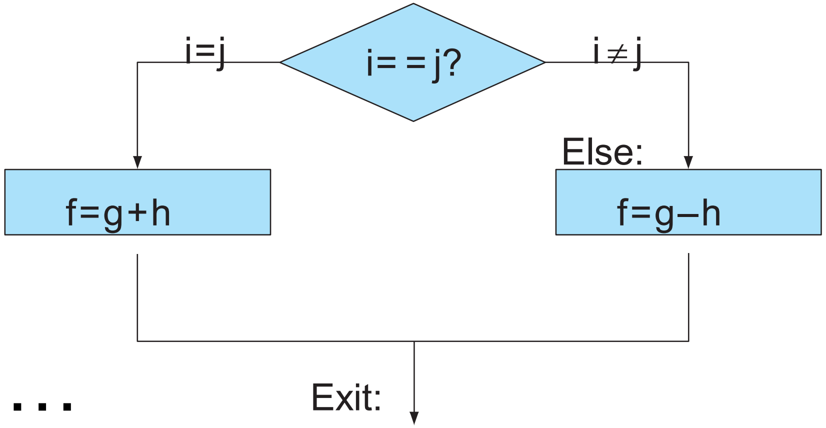
```
# MIPS assembly code

        move $t0, $zero
        bne $t0, $zero, after_print
        li $v0, 4
        la $a0, x_is_zero
        syscall
after_print:
        li $v0, 10
        syscall
```

Compiling *if-else* Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```



  - f, g, h, i, j in $s0, $s1, …

- Compiled MIPS code: **if_then_else.s**

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:**while.s**

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Example add_0_to_n.s

```
// Adds up all numbers between 0 and N.
// For example, if N = 3, then it would find
//0 + 1 + 2 + 3 = 6

sum = 0 ;

while (N != 0) {
  sum += N ;
   N--;
 }
```

```
.text
main:

        li $t0, 3      # value of N

        li $t1, 0      # running sum

loop:
        beq $t0, $zero, loop_exit
        addu $t1, $t1, $t0
        addi $t0, $t0, -1
        j loop

loop_exit:
```

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

demo: for_loop.s

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
       addi $s1, $0, 0
       add  $s0, $0, $0
       addi $t0, $0, 10
for:   beq  $s0, $t0, done
       add  $s1, $s1, $s0
       addi $s0, $s0, 1
       j    for
done:
```

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Demo: slt.s

C Code

```c
// add the powers of 2 from 1
// to 10
int sum = 0;
int i,j;

for (i=1,j=2; i < 11; i= i+1,j=j*2)
{
  sum = sum + j;

}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum, $s2 = j

        addi $s1, $0, 0

        addi $s0, $0, 1
        addi $s2, $0, 2

loop:   slti $t1, $s0, 11
        beq  $t1, $0, done

        add  $s1, $s1, $s2

        addi $s0, $s0, 1
        sll  $s2, $s2, 1
        j    loop
done:
```

Example max.s

```
# stores the max of two numbers
        .text
main:
        # two initial numbers
        li $t0, 5
        li $t1, 10
        # put max in $t3
        #
        # max = $t0
        # if ($t0 < $t1) {
        #   max = $t1
        # }
        move $t3, $t0
        slt $t2, $t0, $t1       # $t0 < $t1?
        beq $t2, $zero, have_max  # if not, we are done
        move $t3, $t1
have_max:
        # print out the result
        li $v0, 1
        move $a0, $t3
        syscall
        # exit the program
        li $v0, 10
        syscall

    .end main
```

COD 5e Exercise 2.23

- Assume $t0 holds the value 0x00101000. What is the value of $t2 after the following instructions?
- 假设$t0中存放数值 0x00101000, 在执行下列指令后$t2的值是多少?

```
        slt     $t2, $0, $t0
        bne     $t2, $0, ELSE
j       DONE
ELSE: addi $t2, $t2, 2
DONE:
```

COD 5e Exercise 2.23

- Assume $t0 holds the value 0x00101000. What is the value of $t2 after the following instructions?

```
        slt     $t2, $0, $t0
        bne     $t2, $0, ELSE
j       DONE
ELSE: addi $t2, $t2, 2
DONE:
```

2.23 (Answer)

- Assume \$t0 holds the value 0x00101000. What is the value of \$t2 after the following instructions?

```
      slt     $t2, $0, $t0
      bne     $t2, $0, ELSE
j       DONE
ELSE: addi $t2, $t2, 2
DONE:
```

\$t2 = 3

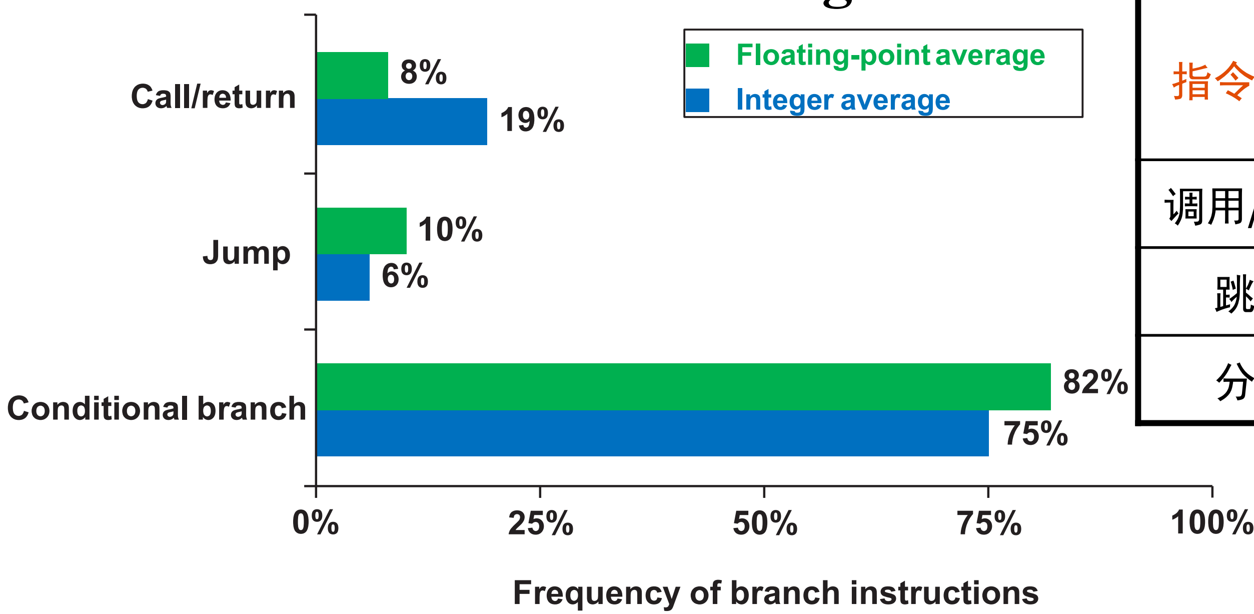
2.6控制流指令

- 控制流指令分类

  条件转移，跳转，过程调用，过程返回

- 控制流指令不同类型出现的相对频率（ Alpha 上测试）

Figure A.14



指令类型	使用频度	
	整型平均	浮点平均
调用/返回	19%	8%
跳转	6%	10%
分支	75%	82%

2.6控制流指令

❖ 不同类型**条件转移**中的频率（ **Alpha 上测试**）

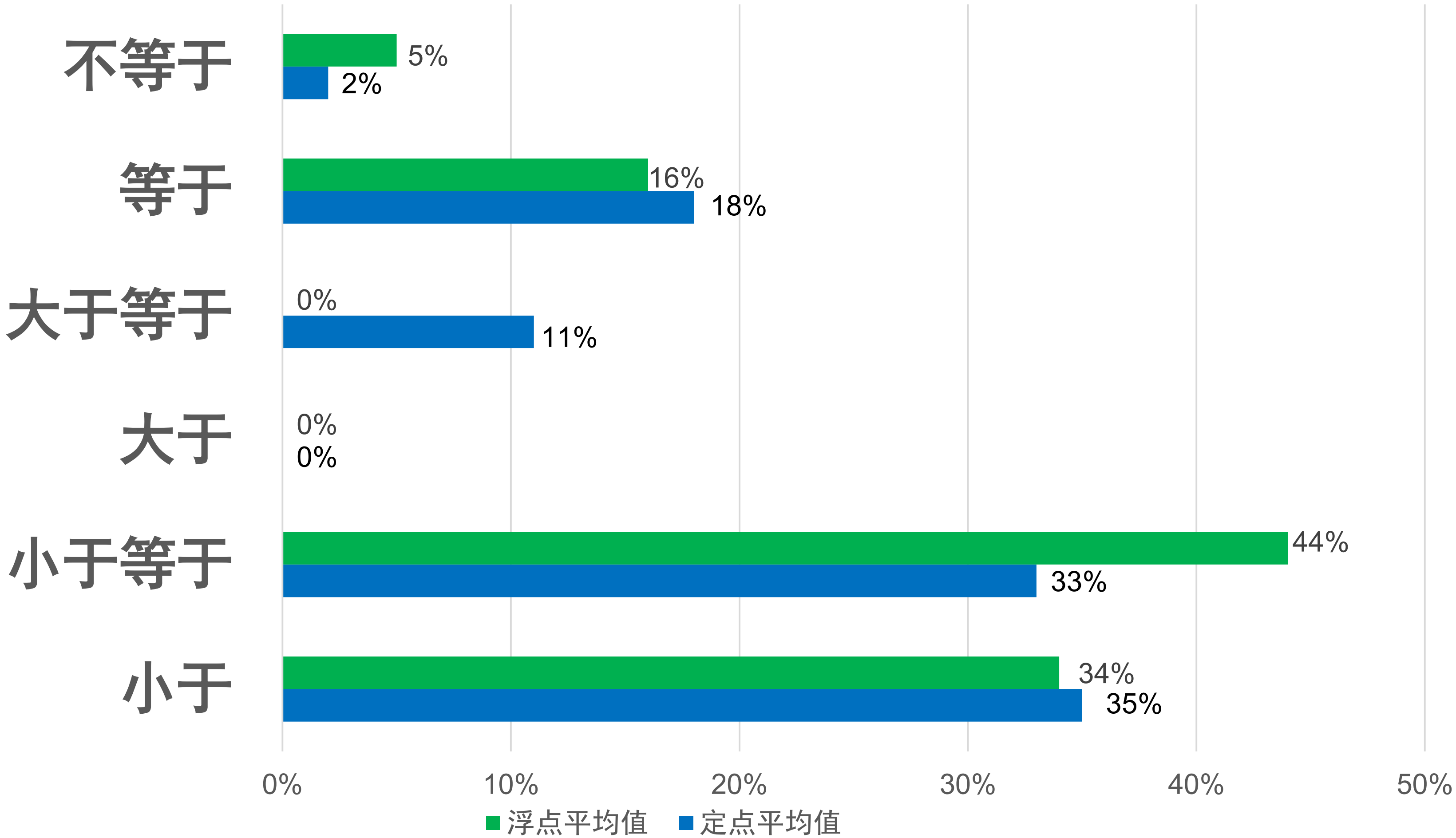


Figure A.17

2.6控制流指令

- 控制流指令的寻址方式

一般要指明转移的目标地址。

过程返回是例外，因为编译时不知道返回地址。

- PC相对寻址，即使用基于程序计数器（PC）的位移量来指定目标地址。

PC相对寻址指令的优点：

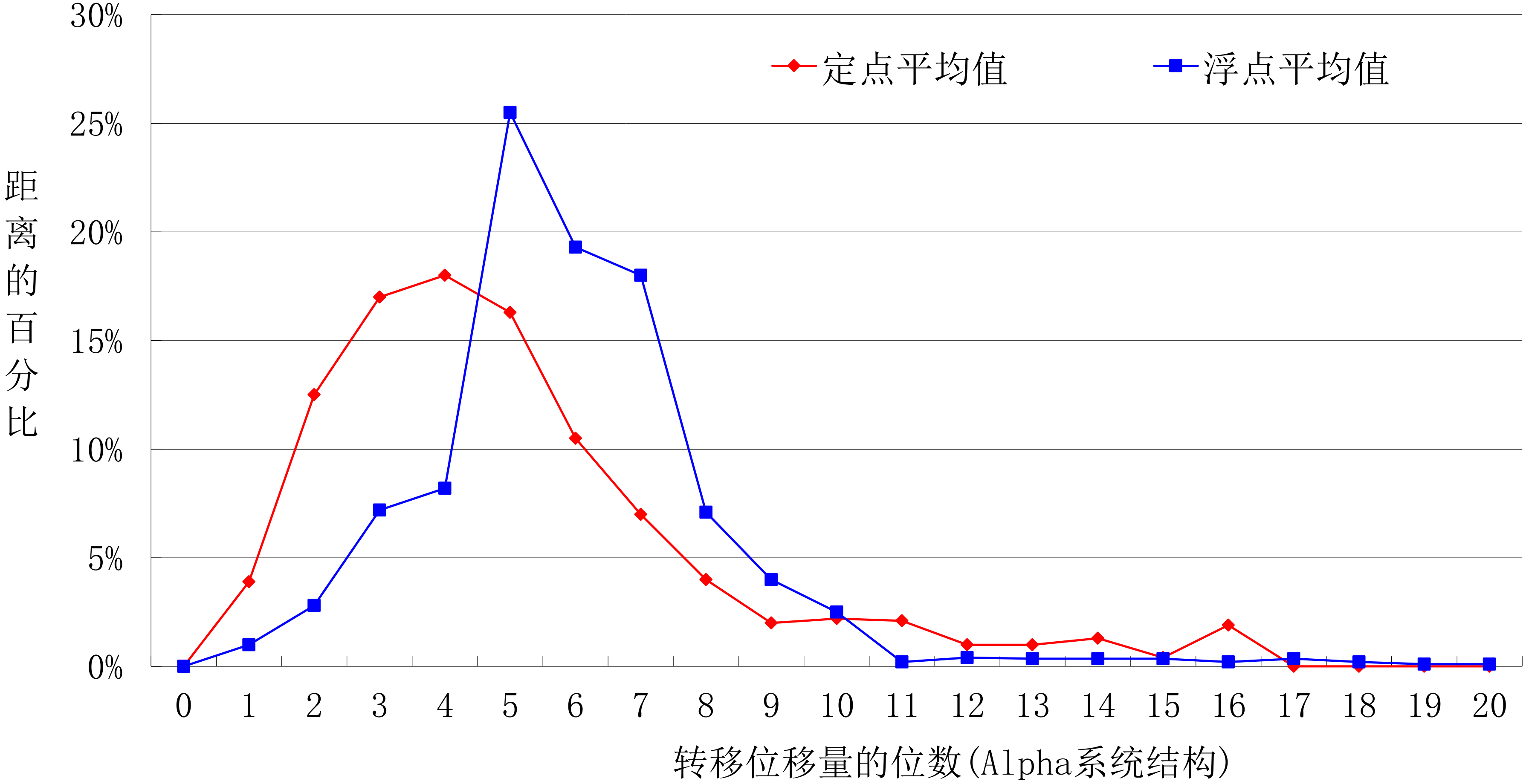
目标与当前指令离得不远；使用相对偏移地址可以缩减指令长度。

使用相对寻址的程序可以载入到主存任何位置，称为位置无关，对在执行时才链接的程序可以减少工作量。

2.6控制流指令

PC相对寻址转移距离：转移指令与目标之间的指令数

（ Alpha 上测试） **Figure A.15**



定点相对寻址转移位移量：至少8位。MIPS是16位

例如：64位MIPS主要控制流指令

❖ MIPS控制流指令 Figure A.25

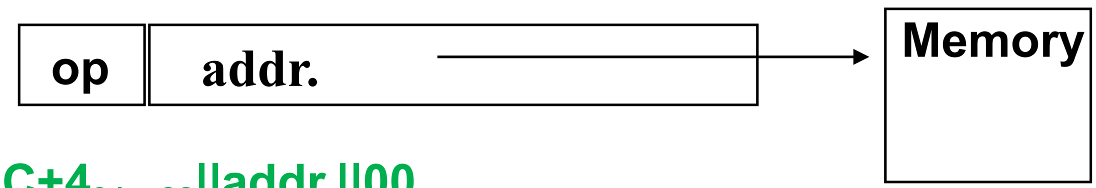
指令举例	指令名称	含义
j name	跳转	PC←$\mathbf{PC+4_{GPRLENGTH-1..28}}$ **|| name || 2b'00**
jal name	跳转并链接	Regs[r31]←PC+**8** PC←$\mathbf{PC+4_{GPRLENGTH-1..28}}$ **|| name || 2b'00**
jalr r2	寄存器跳转并链接	Regs[r31]←PC+**8** PC←Regs[r2]
jr r3	寄存器跳转	PC←Regs[r3]
beqz r1，name	等于零时分支	If(Regs[r1]==0) PC←**PC+4 + name << 2**
bne r3，r4，name	不等于时分支	If(Regs[r3]!=(Regs[r4]) PC←**PC+4 + name << 2**
movz r1，r2，r3	等于零时传送	if(Regs[r3]==0) Regs[r1]←Regs[r2]

分支的目标地址=16位带符号位移量左移2位 +（PC+4）

MIPS指令是32位,4个字节长

J-Format Instructions

J-format: OP=000010H (j) or 000011H (jal)



$\text{Result}=\text{PC}+4_{31\sim28}||\text{addr.}||00$

2.6控制流指令

- **寄存器间接跳转**

编译时不知道目标位置，为了实现返回和间接跳转，需要使用寄存器间接跳转：给出包含目标地址的寄存器名称。

寄存器间接跳转还支持：

* 分支选择语句case或者switch
* 面向对象语言中的虚拟函数或者方法
* 高阶函数或者函数指针
* 动态共享库

以上四种情况，编译时都不知道目标地址，因此通常在寄存器间接跳转之前，才把地址从存储器载入到寄存器中。

COD 5e Exercise 2.24

- Suppose the program counter (PC) is set to 0x2000 0000. Is it possible to use one jump (j) MIPS assembly instruction to set the PC to the address as 0x4000 0000? Is it possible to use one branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?
- 假设程序计数器 (PC) 被设置为 0x2000 0000, 是否可以使用MIPS的跳转(j)指令将 PC 设置为地址 0x4000 0000? 是否可以使用MIPS的相等则分支(beq)指令将 PC 设置为该地址？

2.24(Answer)

- Suppose the program counter (PC) is set to 0x2000 0000. Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as 0x4000 0000? Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?

jump: no, beq: no

COD 5e Exercise 2.26

- Consider the following MIPS loop:
- 考虑如下的MIPS循环：
- LOOP: slt $t2, $0, $t1
- beq $t2, $0, DONE
- addi $t1, $t1, -1
- addi $s2, $s2, 2
- j LOOP
- DONE:

COD 5e Exercise 2.26

- Consider the following MIPS loop:
- LOOP: slt $t2, $0, $t1
- beq $t2, $0, DONE
- addi $t1, $t1, -1
- addi $s2, $s2, 2
- j LOOP
- DONE:

2.26

- [1] Assume that the register $t1 is initialized to the value 10. What is the value in register $s2 assuming $s2 is initially zero?假设寄存器$t1的初始值为10,假设$t2初始值为0, 则循环执行完毕时寄存器$t2的值是多少？
- [2] For each of the loops above, write the equivalent C code routine. Assume that the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively.对于上面的循环体，写出等价的C代码例程。假定寄存器$s1、 $s2、 $t1和$t2分别为整数A、B 、 i和temp。
- [3] For the loops written in MIPS assembly above, assume that the register $t1 is initialized to the value N. How many MIPS instructions are executed?假定寄存器$t1的初始值为N,上面的MIPS汇编循环执行了多少条指令？

2.26

- [1] Assume that the register \$t1 is initialized to the value 10. What is the value in register \$s2 assuming \$s2 is initially zero?
- [2] For each of the loops above, write the equivalent C code routine. Assume that the registers \$s1, \$s2, \$t1, and \$t2 are integers A, B, i, and temp, respectively.
- [3] For the loops written in MIPS assembly above, assume that the register \$t1 is initialized to the value N. How many MIPS instructions are executed?

2.26 Answer (1)

- Consider the following MIPS loop:
- LOOP: slt $t2, $0, $t1
- beq $t2, $0, DONE
- addi $t1, $t1, -1
- addi $s2, $s2, 2
- j LOOP
- DONE:

[1] Assume that the register $t1 is initialized to the value 10. What is the value in register $s2 assuming $s2 is initially zero?

- [1] 20

2.26 Answer(2)

- Consider the following MIPS loop:
- LOOP: slt $t2, $0, $t1
- beq $t2, $0, DONE
- addi $t1, $t1, -1
- addi $s2, $s2, 2
- j LOOP
- DONE:

[2] For each of the loops above, write the equivalent C code routine. Assume that the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively.

```
i = 10;
While (0 < i) {
i += – 1;
B += 2;
}
```

2.26 Answer(3)

- Consider the following MIPS loop:
- LOOP: slt $t2, $0, $t1
- beq $t2, $0, DONE
- addi $t1, $t1, -1
- addi $s2, $s2, 2
- j LOOP
- DONE:

[3] For the loops written in MIPS assembly above, assume that the register $t1 is initialized to the value N. How many MIPS instructions are executed?

- [3] 5*N+2 (when N > 0)
- 2 (when N<=0)

2.26 Answer

- [1] 20
- [2]

```
i = 10;
While (0 < i ) {
i += – 1;
B += 2;
}
```

- [3] 5*N+2 (when N > 0)
-   2 (when N<=0)

COD 5e Exercise 2.29

- Translate the following loop into C. Assume that the C-level integer i is held in register $t1, $s2 holds the C-level integer called result, and $s0 holds the base address of the integer MemArray.
- 将下面的循环翻译成C代码。假定寄存器$t1中存放C语言的整数i, $s2中存放C语言的整数result, $s0存放整数数组MemArray的基地址。

```
        addi   $t1, $0, 0
LOOP:   lw     $s1, 0($s0)
        add    $s2, $s2, $s1
        addi   $s0, $s0, 4
        addi   $t1, $t1, 1
        slti   $t2, $t1, 100
        bne    $t2, $s0, LOOP
```

COD 5e Exercise 2.29

- Translate the following loop into C. Assume that the C-level integer i is held in register $t1, $s2 holds the C-level integer called result, and $s0 holds the base address of the integer MemArray.

```
       addi  $t1, $0, 0
LOOP:  lw    $s1, 0($s0)
       add   $s2, $s2, $s1
       addi  $s0, $s0, 4
       addi  $t1, $t1, 1
       slti  $t2, $t1, 100
       bne   $t2, $s0, LOOP
```

2.29

- Translate the following loop into C. Assume that the C-level integer i is held in register $t1, $s2 holds the C-level integer called result, and $s0 holds the base address of the integer MemArray.

```
        addi   $t1, $0, 0
LOOP:   lw     $s1, 0($s0)
        add    $s2, $s2, $s1
        addi   $s0, $s0, 4
        addi   $t1, $t1, 1
        slti   $t2, $t1, 100
        bne    $t2, $0, LOOP
```

```
for (i=0; i<100; i++) {
    result += MemArray[i];
}
```

Or

```
i = 0 ;
do
{
 result += MemArray[i];
 i++ ;
} while( i < 100 )
```

2.6控制流指令

- **过程调用的可选方案**

  有两种基本、传统的方法用来保存子程序使用的寄存器：调用者保存和被调用者保存。

- 调用者保存：调用者调用其他过程时，必须保存在调用过程后还要使用的寄存器，被调用者则无须维护这些寄存器。
- 被调用者保存：被调用的过程必须保存它要使用的寄存器，调用者则不受这种限制。

有时候，如果两个不同的过程都要访问相同的全局变量，则必须使用调用者保存方法。大多数实际使用的编译器会结合这两种方法。

8086 中断处理子程序（被调用者保存）

```
PRT_INT  PROC  FAR
        PUSH    AX
        PUSH    BX
        PUSH    DX
        PUSH    DS
        LDS   BX，POINT          ；装入地址指针BX，DS
        MOV  AL，[BX]            ；取打印数据
        .
        .
        POP     DS
        POP     DX
        POP     BX
        POP     AX
        IRET                    ；中断返回
PRT_INT  ENDP
```

保护现场

恢复现场

Six Steps in Execution of a Procedure

1. **Main routine (caller) places parameters in a place where the procedure (callee) can access them**
   - `$a0` - `$a3`: four argument registers
2. **Caller transfers control to the callee (`jal Dest`)**
3. **Callee acquires the storage resources needed**
4. **Callee performs the desired task**
5. **Callee places the result value in a place where the caller can access it**
   - `$v0` - `$v1`: two value registers for result values
6. **Callee returns control to the caller (`jr $ra`)**
   - `$ra`: one return address register to return to the point of original

Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Register Usage

Register Name	Software Name	Register Usage	Saver
$0	zero	Constant zero- Always returns 0	
$1	at	Assembler temp(Reserved for assembler)	
$2..$3	v0,v1	Function return	Caller
$4..$7	a0-a3	Incoming Arguments	Caller
$8..$15	t0-t7	Temporary Registers	Caller
$16..$23	s0-s7	Saved Registers	Callee
$24..$25	t8,t9	Temporary Registers	Caller
$26..$27	k0,k1	Exception Handling (Reserved for OS)	Callee
$28	gp	Global Pointer	Callee
$29	sp	Stack Pointer	Callee
$30	s8 or fp	Save register or Frame pointer if needed	Callee
$31	ra	Return Address	Caller

Procedure Call Instructions

- **Procedure call: jump and link**

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
- **Procedure return: jump register**

  `jr $ra`

  - Copies $ra to program counter
- **jump register can also be used for computed jumps**
  - e.g., for switch/case statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in \$a0, …, \$a3
  - f in \$s0 (hence, need to save \$s0 on stack)
  - Result in \$v0

Leaf Procedure Example

- MIPS code: **leaf_example.s**

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)        Save $s0 on stack
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3      Procedure body
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero    Result
    lw   $s0, 0($sp)
    addi $sp, $sp, 4        Restore $s0
    jr   $ra                Return
```

Non-Leaf Procedures

- **Procedures that call other procedures**
- **For nested call, caller needs to save on the stack:**
  - Its return address
  - Any arguments and temporaries needed after the call
- **Restore from the stack after the call**

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100*4
		Equal test; PC relative branch
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100*4
		Not equal test; PC relative
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
		Compare less than; 2's comp.
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
		Compare < constant; 2's comp.
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
		Compare less than; natural numbers
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
		Compare < constant; natural numbers
jump	j 10000	go to **PC+4$_{31\sim28}$**|| 10000*4
		Jump to target address
jump register	jr $31	go to $31
		For switch, procedure return
jump and link	jal 10000	$31 = PC + 4; go to **PC+4$_{31\sim28}$**||10000*4
		For procedure call

2.6控制流指令

- **条件转移的可选方案** **Figure A.16**

名称	举例	如何测试条件	优点	缺点
条件码（CC）	80x86，ARM，PowerPC，SPARC，SuperH	由ALU操作设定的某些特定位，可能是由程序控制的	有时条件可自由设置	CC是附加状态。条件码强制限制了指令顺序，因为它把信息从一条指令传送给一个转移
条件寄存器	Alpha，MIPS	用比较结果测试任意寄存器	简单	占用一个寄存器
比较并转移	PA-RISC，VAX	比较是转移的一部分，通常比较只限于子集内部	一个转移是一条而不是两条指令	对流水线执行来说，一条指令要做的事情可能太多了

上图列出了目前使用的三种方法和它们各自的优缺点。

Discussion

- 考虑条件分支指令的两种不同设计方法：

（1）CPU1：通过比较指令设置条件码，然后测试条件码进行分支。

（2）CPU2：在分支指令中包括比较过程。

在这两种CPU中，条件分支指令都占用2个时钟周期，而所有其它指令占用1个时钟周期。对于CPU1，执行的指令中分支指令占30%；由于每条分支指令之前都需要有比较指令，因此比较指令也占30%。由于CPU1在分支时不需要比较，因此CPU2的时钟周期时间是CPU1的1.35倍。问：哪一个CPU更快？如果CPU2的时钟周期时间只是CPU1的1.15倍，哪一个CPU更快呢？

Discussion

- 考虑条件分支指令的两种不同设计方法：

  （1）CPU1：通过比较指令设置条件码，然后测试条件码进行分支。

  （2）CPU2：在分支指令中包括比较过程。

在这两种CPU中，条件分支指令都占用2个时钟周期，而所有其它指令占用1个时钟周期。对于CPU1，执行的指令中分支指令占30%；由于每条分支指令之前都需要有比较指令，因此比较指令也占30%。由于CPU1在分支时不需要比较，因此CPU2的时钟周期时间是CPU1的1.35倍。问：哪一个CPU更快？如果CPU2的时钟周期时间只是CPU1的1.15倍，哪一个CPU更快呢？

Discussion

考虑条件分支指令的两种不同设计方法：

（1）CPU_1：通过比较指令设置条件码，然后测试条件码进行分支。

（2）CPU_2：在分支指令中包括比较过程。

在这两种CPU中，条件分支指令都占用2个时钟周期，而所有其它指令占用1个时钟周期。对于CPU_1，执行的指令中分支指令占30%；由于每条分支指令之前都需要有比较指令，因此比较指令也占30%。由于CPU_1在分支时不需要比较，因此CPU_2的时钟周期时间是CPU_1的1.35倍。问：哪一个CPU更快？如果CPU_2的时钟周期时间只是CPU_1的1.15倍，哪一个CPU更快呢？

解 采用CPU时间公式。占用2个时钟周期的分支指令占总指令的30%，剩下的指令占用1个时钟周期。所以

$$CPI_1 = 0.3 \times 2 + 0.70 \times 1 = 1.3$$

则CPU_1性能为：

$$\text{总CPU时间}_1 = IC_1 \times 1.3 \times \text{时钟周期}_1$$

根据假设，有：

$$\text{时钟周期}_2 = 1.35 \times \text{时钟周期}_1$$

在CPU_2中没有独立的比较指令，所以CPU_2的程序量为CPU_1的70%，分支指令的比例为：

30%/70% = 42.8%

$$30\%/70\% = 42.8\%$$

这些分支指令占用2个时钟周期，而剩下的57.2%的指令占用1个时钟周期，因此：

$$CPI_2 = 0.428 \times 2 + 0.572 \times 1 = 1.428$$

因为CPU_2不执行比较，故：

$$IC_2 = 0.7 \times IC_1$$

因此CPU_2性能为：

$$\begin{aligned} \text{总CPU时间}_2 &= IC_2 \times CPI_2 \times \text{时钟周期}_2 \\ &= 0.7 \times IC_1 \times 1.428 \times (1.35 \times \text{时钟周期}_1) \\ &= 1.349 \times IC_1 \times \text{时钟周期}_1 \end{aligned}$$

前面计算 $\text{总CPU时间}_1 = IC_1 \times 1.3 \times \text{时钟周期}_1$

在这些假设之下，尽管CPU_2执行指令条数较少，CPU_1因为有着更短的执行时间，所以更快。

如果CPU_2的时钟周期时间仅仅是CPU_1的1.15倍，则

$$时钟周期_2 = 1.15 \times 时钟周期_1$$

CPU_2的性能为：

$$
\begin{aligned}
总CPU时间_2 &= IC_2 \times CPI_2 \times 时钟周期_2 \\
&= 0.7 \times IC_1 \times 1.428 \times (1.15 \times 时钟周期_1) \\
&= 1.15 \times IC_1 \times 时钟周期_1
\end{aligned}
$$

前面计算 $总CPU时间_1 = IC_1 \times 1.3 \times 时钟周期_1$

因此CPU_2更快。