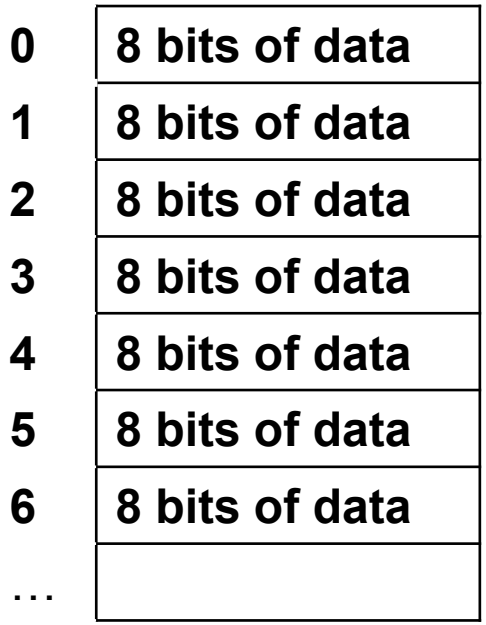
第二章　指令系统原理与实例

- 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

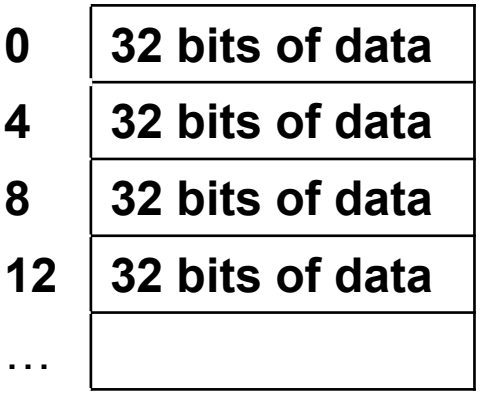
2.3 存储器寻址

存储器地址表示

我们讨论的所有指令系统都是字节寻址的，都提供了字节（8位）、半字（16位）和字（32位）寻址，大多数的计算机还提供了双字（64位）寻址。



内存中的字节



内存中的字

2.3存储器寻址

- 小端模式&大端模式

小端模式把最低有效字节存放在地址为“X…XX00”的位置上（低地址存低字节）。12345678H的存放方式如下左图。

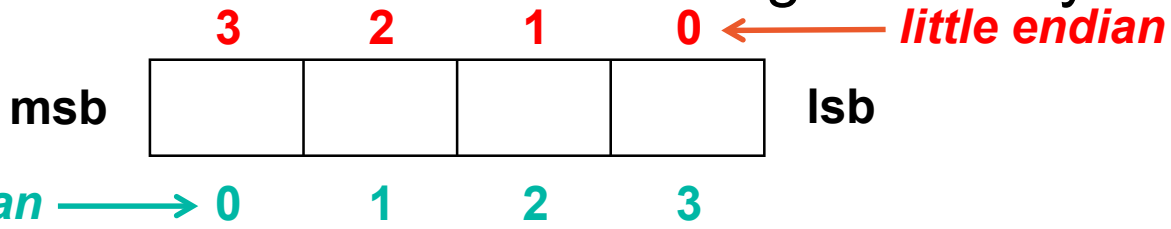
小端模式

大端模式

大端模式把最高有效字节存放在地址为“X…XX00”的位置上（低地址存高字节）。12345678H的存放方式如上右图。

Endianness

- **Big Endian:** **Most-significant byte at least address of word**
  - word address = address of most significant byte
- **Little Endian:** **Least-significant byte at least address of word**
  - word address = address of least significant byte



- **MIPS is can go either way**
  - Using QtSpim simulator in our lab, which is little endian
- **Why is this confusing?**
  - Data stored in reverse order than you write it out!
  - Data `0x01020304` stored as `04 03 02 01` in memory

Little endian : Increasing address

COD 5e Exercise 2.7

- Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.
- 分别画出数据0xabcdef12在小端编址和大端编址的机器上是如何分布在存储器中的。假定数据从地址0开始存储。

2.7 Answer

- Show how the value 0xabcdef12 would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

Little-Endian		Big-Endian	
Address	**Data**	**Address**	**Data**
3	ab	3	12
2	cd	2	ef
1	ef	1	cd
0	12	0	ab

`12 ef cd ab` →

Little endian : Increasing address

`ab cd ef 12` →

Big endian : Increasing address

Memory Operand Example 1

- **C code:**

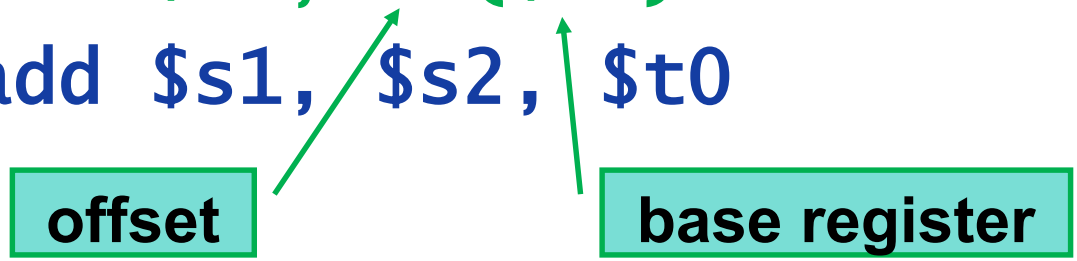
```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- **Compiled MIPS code: memory_1.s**

  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)  #load word $t0=A[8]
add $s1, $s2, $t0
```



Memory Operand Example 2

- **C code:**

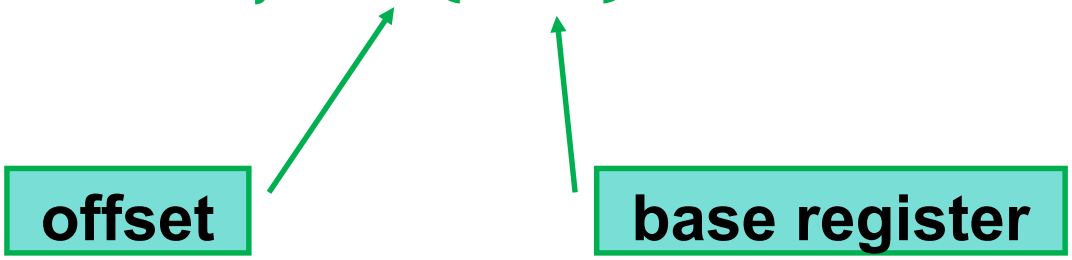
```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3

- **Compiled MIPS code: memory_2.s**

  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)  #load word $t0=A[8]
add $t0, $s2, $t0 # $t0  = h + A[8]
sw  $t0, 48($s3)  # store word A[12]=$t0
```



Example lw_sw.s

- Suppose each integer contains 32-bit value. A is an array with 400 integer.
- A[300] = h + A[300];

is compiled into

- lw $t0,1200($t1) # Temporary reg $t0 gets A[300]
- add $t0,$s2,$t0 # Temporary reg $t0 gets h + A[300]
- sw $t0,1200($t1) # Stores h + A[300] back into A[300]

COD 5E Exercise 2.4

- For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.
- 下面的MIPS汇编语言程序段对应的C语言表达式是什么？假设变量f、g、h、i和j分别赋值给寄存器$s0、 $s1、 $s2、 $s3和$s4。假设数组A和B的基地址分别在寄存器$s6和$s7中。

```
sll $t0, $s0, 2      # $t0 = f * 4
add $t0, $s6, $t0    # $t0 = &A[f]
sll $t1, $s1, 2      # $t1 = g * 4
add $t1, $s7, $t1    # $t1 = &B[g]
lw  $s0, 0($t0)      # f = A[f]
addi $t2, $t0, 4
lw  $t0, 0($t2)
add $t0, $t0, $s0
sw  $t0, 0($t1)
```

COD 5E Exercise 2.4

- **For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.**

```
sll $t0, $s0, 2      # $t0 = f * 4
add $t0, $s6, $t0    # $t0 = &A[f]
sll $t1, $s1, 2      # $t1 = g * 4
add $t1, $s7, $t1    # $t1 = &B[g]
lw  $s0, 0($t0)      # f = A[f]
addi $t2, $t0, 4
lw  $t0, 0($t2)
add $t0, $t0, $s0
sw  $t0, 0($t1)
```

Answer: 2.4

```
sll   $t0 , $s0 , 2        # $t0 = f * 4
add   $t0 , $s6 , $t0      # $t0 = &A[f]
sll   $t1 , $s1 , 2        # $t1 = g * 4
add   $t1 , $s7 , $t1      # $t1 = &B[g]
lw    $s0 , 0($t0)         # f= A[f]

addi  $t2 , $t0 , 4        # $t2 = &A[1+f]
lw    $t0 , 0($t2)         # $t0 = A[1+f]
add   $t0 , $t0 , $s0      # $t0 = A[1+f] + A[f]
sw    $t0 , 0($t1)         # B[g] = A[1+f] + A[f]
```

2.3存储器寻址

- 对齐

假设一个s字节数据的地址是A,如果A mod s=0,访问该地址就是对齐的。

字节地址的低三位（2进制）								
数据宽度	000	001	010	011	100	101	110	111
1字节（字节）	对齐	对齐	对齐	对齐	对齐	对齐	对齐	对齐
2字节（半字）	对齐		对齐		对齐		对齐	
2字节（半字）		未对齐		未对齐		未对齐		未对齐
4字节（字）	对齐				对齐			
4字节（字）		未对齐				未对齐		
4字节（字）			未对齐				未对齐	
4字节（字）				未对齐				未对齐
8字节（双字）	对齐							
8字节（双字）		未对齐						
8字节（双字）			未对齐					
8字节（双字）				未对齐				
8字节（双字）					未对齐			
8字节（双字）						未对齐		
8字节（双字）							未对齐	
8字节（双字）								未对齐

Figure A.5

2.3存储器寻址

- 为什么要有对齐限制？

- 字或双字整数倍对齐访问存储器：简化硬件实现的复杂性。
- 一次不对齐的存储器访问：导致多次对齐存储器访问。因此，即使是在没有对齐限制的计算机里面，对齐访问的程序也会运行得比较快。

2.3存储器寻址

- 寻址方式

  寻址方式：指令中如何指定所要访问操作数的地址。寻址方式要指定常量、寄存器和存储器操作数的位置。

- 后图列出了近期计算机中使用的常用数据寻址方式。

  * 立即数通常也被认为是一种存储器寻址方式（尽管它们要访问的数值在指令流里）。

  * 寄存器不属于存储器寻址。

  * 把依赖于程序计数器的PC相对寻址（后面详细讨论）也分离出来。

2.3存储器寻址

❖表示方法

- ←：赋值操作
- Mem：存储器
- Regs：寄存器组
- 方括号：表示内容
  - Mem[ ]：存储器的内容
  - Regs[ ]：寄存器的内容
  - Mem[Regs[R1]]：以寄存器R1中的内容作为地址的存储器单元中的内容

采用多种寻址方式可以显著地减少程序的指令条数，但可能增加计算机的实现复杂度以及指令的CPI。

2.3存储器寻址

寻址方式

寻址方式	指令举例	含义	何时使用
寄存器寻址	Add R4，R3	Regs[R4]←Regs[R4]+Regs[R3]	数值在寄存器中
立即数寻址	Add R4，#3	Regs[R4]←Regs[R4]+3	数值是常量
位移量寻址	Add R4，100（R1）	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	存取局部变量（+模拟寄存器间接、直接寻址）
寄存器间接寻址	Add R4，（R1）	Regs[R4]←Regs[R4]+Mem[Regs[R1]]	使用指针或者计算出的地址进行寻址
索引寻址	Add R3，(R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+[Regs[R2]]	有时用于数组中，R1是数组的基址，R2是索引值
直接寻址	Add R1，（1001）	Regs[R1]←Regs[R1] +Mem[1001]	用来存取静态数据；地址常量可能需要很大
存储器间接寻址	Add R1，@（R3）	Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]	如果R3是指针p的地址，那么就得到*p
自动递增寻址	Add R1，（R2）+	Regs[R1]←Regs[R1]+Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	用在循环中递增变量，R2是数组的起始地址，每次增加d
自动递减寻址	Add R1，-(R2)	Regs[R2]←Regs[R2]-d Regs[R1]←Regs[R1]+Mem[Regs[R2]]	和自动递增类似，自动递增/递减用来实现类似栈的push/pop功能
比例寻址	Add R1，100(R2)[R3]	Regs[R1]←Regs[R1] +Mem[100+Regs[R2]+Regs[R3]*d]	用来对数组寻址。一些计算机可以用任意的索引（间接）寻址方式

Figure A.6

2.3存储器寻址

寄存器寻址

- 指令实例：Add R4，R3
- 用途：数值在寄存器中
- 含义：
  - Regs[R4]←Regs[R4]+Regs[R3]

寄存器寻址	Add R4，R3	Regs[R4]←Regs[R4]+Regs[R3]	数值在寄存器中

MIPS register 0 ($zero) is the constant 0

- **Cannot be overwritten**

Useful for common operations

- **E.g., move between registers**

```
add $t2, $s1, $zero
```

2.3存储器寻址

寻址方式

后图给出了VAX系统结构的计算机上用三个测试程序所测试出的**存储器寻址方式**（*不包括寄存器寻址*）的结果。

采用VAX系统结构的计算机来进行测试，是因为它有丰富的寻址方式，且对存储器寻址限制很少。

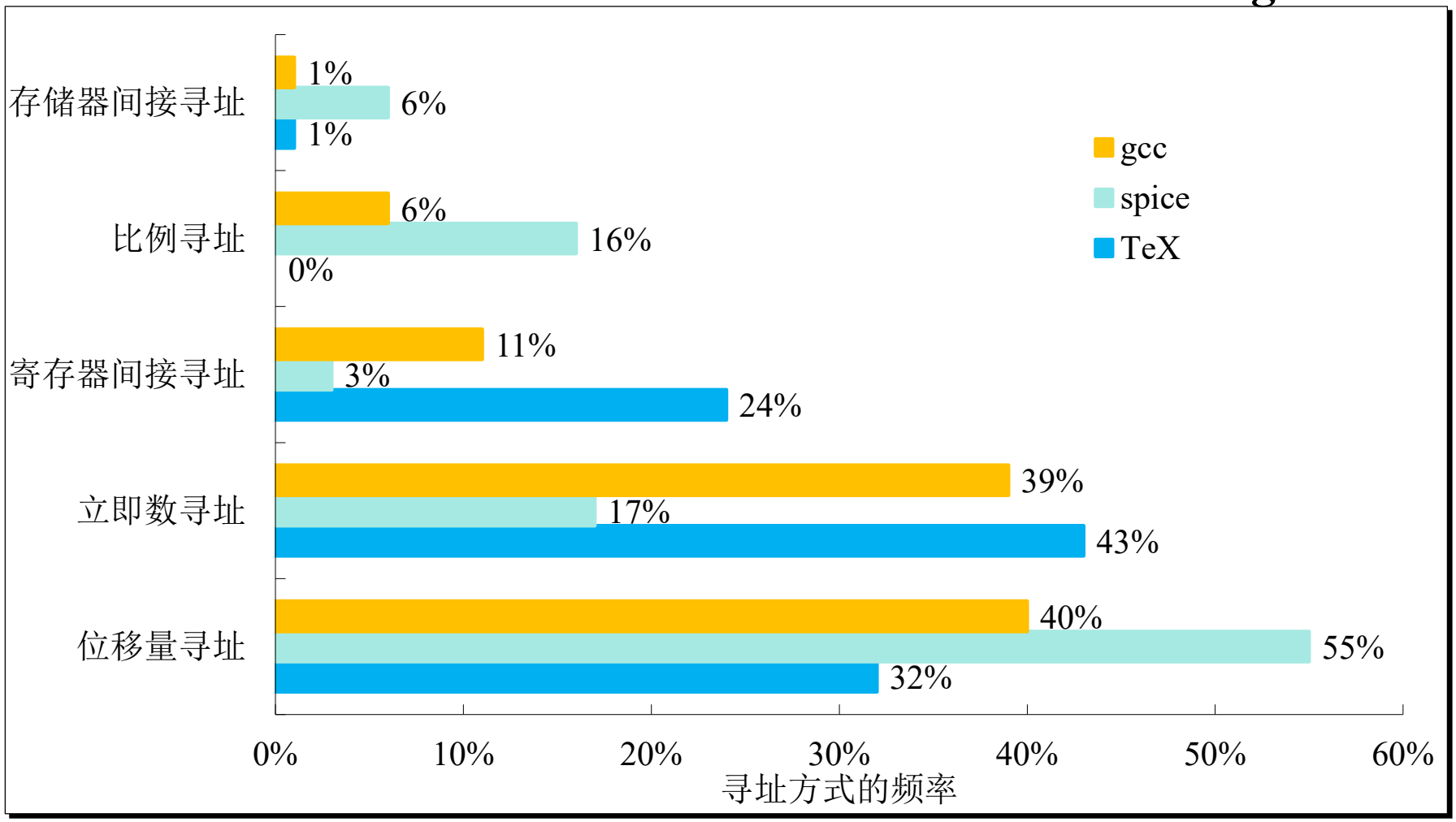
如下所示，**位移量寻址**和**立即数寻址**是用得**最多**的寻址方式。

寻址方式	使用频度	
	gcc	Tex
位移量寻址	40%	32%
立即数寻址	39%	43%
寄存器间接寻址	11%	24%

2.3 存储器寻址

寻址方式（VAX上测试）

Figure A.7



位移量寻址和立即数寻址的使用频度最高。

2.3存储器寻址

位移量寻址方式

- 指令实例：Add R4，100（R1）
- 用途：存取局部变量（+模拟寄存器间接、直接寻址）
- 含义：
  - Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]

位移量寻址	Add R4，100（R1）	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	存取局部变量（+模拟寄存器间接、直接寻址）

2.3存储器寻址

- 位移量寻址方式

主要问题是位移的范围，即多长的位移量。

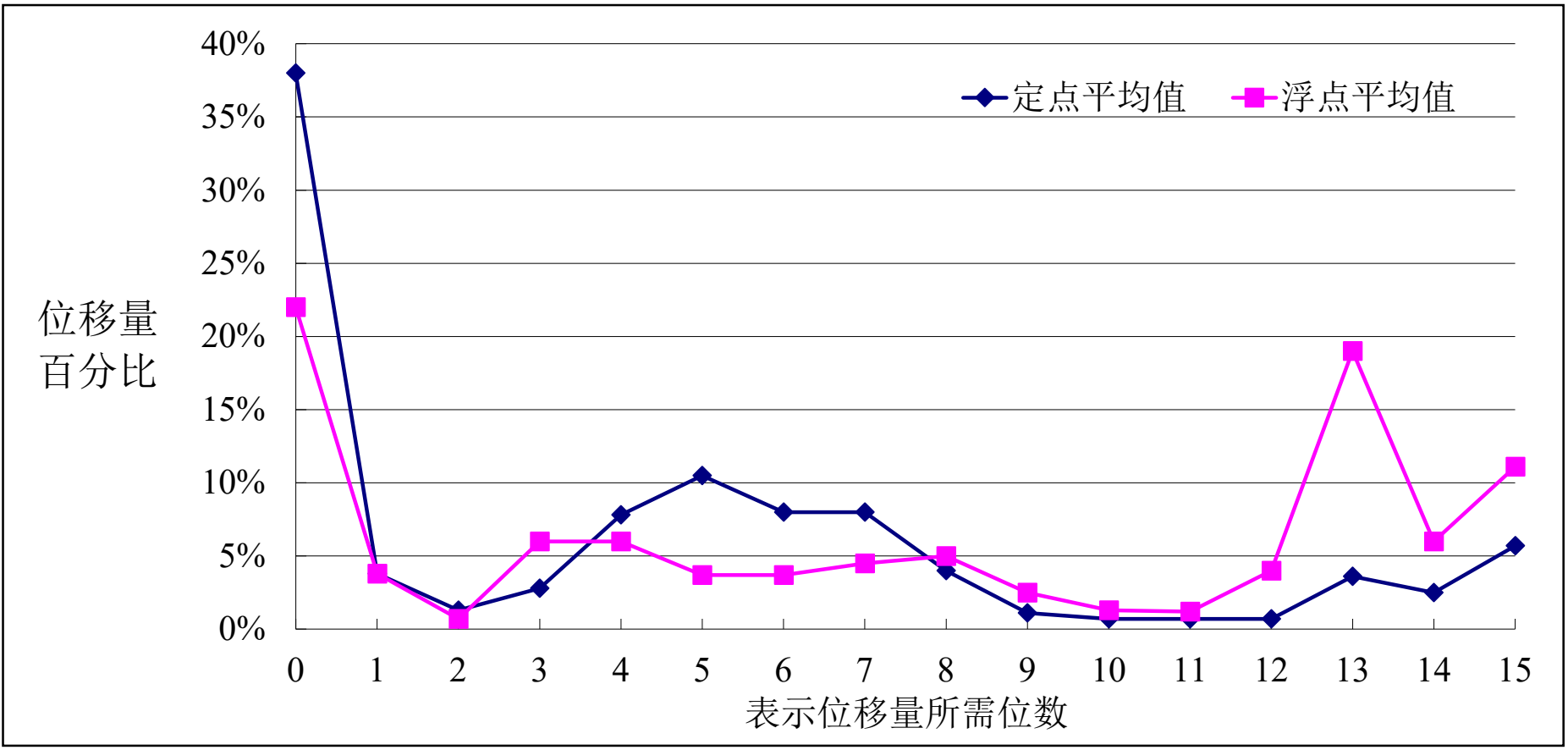
选择位移量的长度：直接影响到指令的长度。

后图列出了Alpha（16位位移量）上运行SPEC CPU2000的CINT2000和CFP2000测试程序数据访问测试结果的平均值。

位移量寻址	Add R4，100（R1）	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	存取局部变量（+模拟寄存器间接、直接寻址）

2.3存储器寻址

❖ 位移量寻址方式（Alpha 上测试） Figure A.8



位移量寻址	Add R4，100（R1）	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	存取局部变量（+模拟寄存器间接、直接寻址）

2.3存储器寻址

位移量寻址方式

如上图所示，位移量值分布范围很广。由于变量的存储位置和存取变量方式的不同以及编译器使用的整个寻址方式的原因，位移量分布范围很广。

结论：13位~16位位移量是有必要的。

如MIPS采用16位位移量。

位移量寻址	Add R4，100（R1）	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	存取局部变量（+模拟寄存器间接、直接寻址）

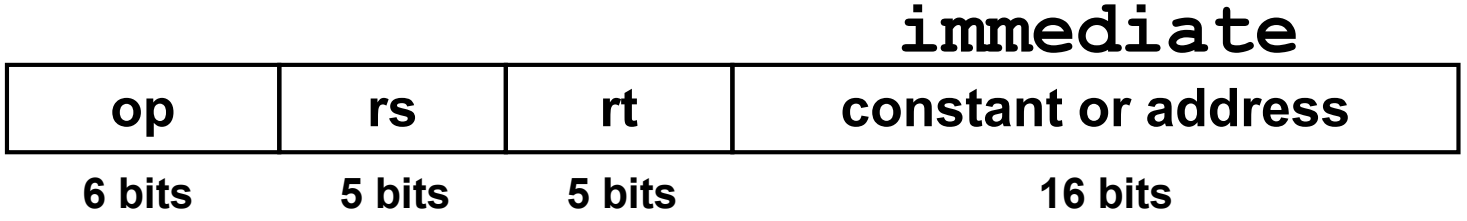
2.3存储器寻址

- 立即数寻址方式(不同指令集的具体格式可能不一样）
  - 指令实例（通常）：Add R4，#3
  - 用途：数值是常量
  - 含义：
    - Regs[R4]←Regs[R4]+3

立即数寻址	Add R4，#3	Regs[R4]←Regs[R4]+3	数值是常量

2.3存储器寻址

- **立即数寻址方式(MIPS）**
  - 指令实例（通常）：addi rt, rs, -50
  - 用途：数值是常量
  - 含义：
    - Regs[rt]←Regs[rs] + (-50)



- **Immediate arithmetic and load/store instructions**
  - Immediate: constant operand, or offset added to base address in rs

Arithmetic Example

complex_assign_4.s (MIPS)

- C code:

```
f = (3 + 4) - (1 + 2);
```

- Compiled MIPS code:

```
addi $t1 , $0, 1
addi $t2 , $0, 2
addi $t3 , $0, 3
addi $t4 , $0, 4
add $t5  , $t1 , $t2
add $t6  , $t3 , $t4
sub  $t0   , $t6  , $t5
```

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Used to add constants

2.3存储器寻址

立即数寻址方式

立即数的取值范围：与位移量相同，立即数的数值大小会影响指令的长度。

如后图所示的立即数分布中，小立即数是最常用的。有时也使用大立即数，特别是在寻址计算中。

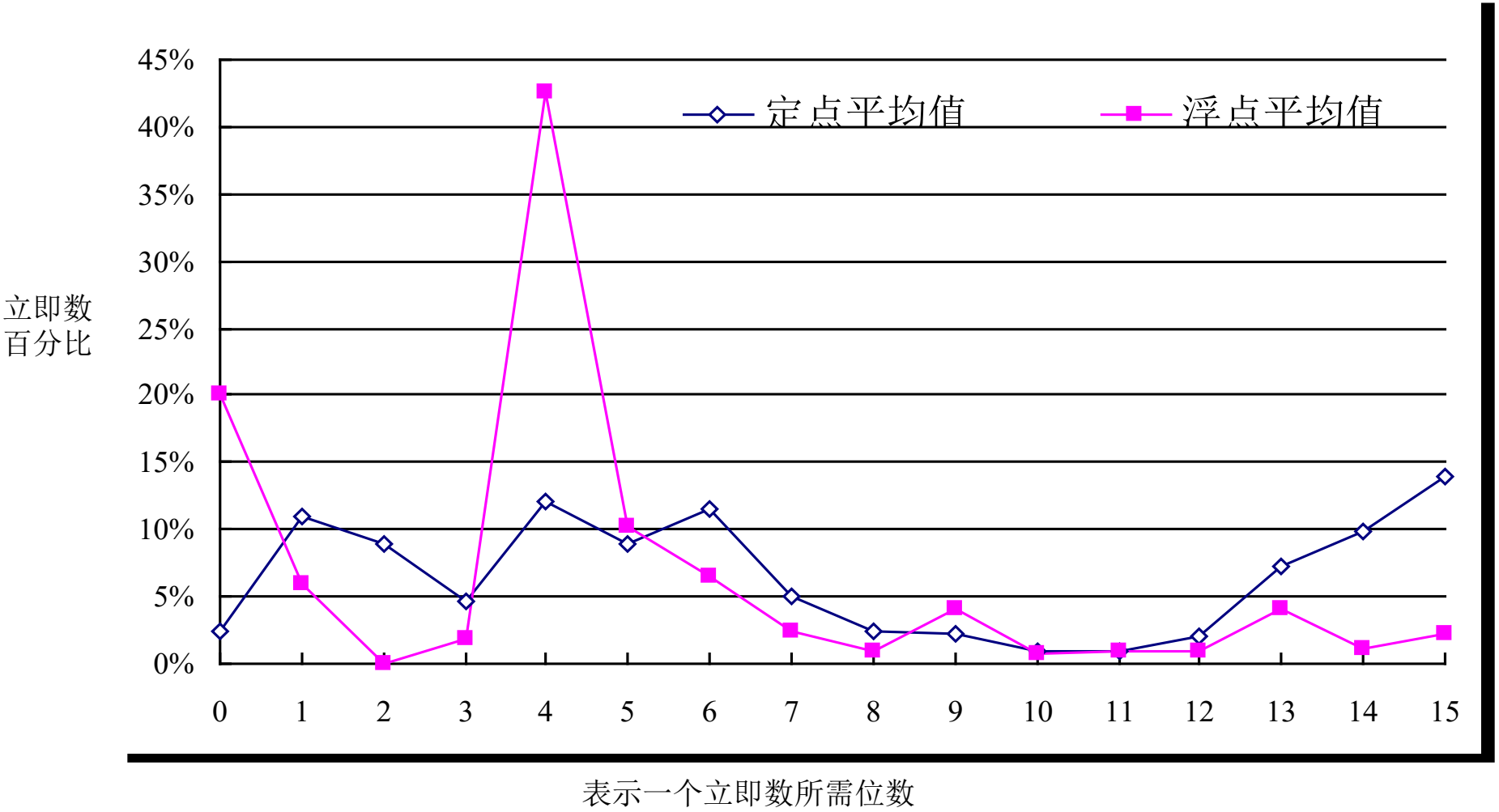
后图列出了Alpha（最大16位立即数）上运行SPEC CPU2000的CINT2000和CFP2000测试程序结果的平均值。

立即数寻址	Add R4，#3	Regs[R4]←Regs[R4]+3	数值是常量

2.3存储器寻址

立即数寻址方式（Alpha 上测试）

Figure A.10



立即数寻址	**Add R4，#3**	**Regs[R4]←Regs[R4]+3**	**数值是常量**

2.3存储器寻址

立即数寻址方式

X轴代表要表示一个立即数所需要数值的位数（0表示立即数的数值为0）。大多数立即数的值是正的，在CINT2000中20%的立即数是负的，CFP2000中为30%。

在一台支持32位立即数的VAX计算机上进行相同的测试，结果显示有20%-25%的立即数大于16位。这样，16位及小于16位的大约占80%，8位及小于8位的大约占50%。

结论：16位立即数是有必要的。

如，MIPS和ALPHA采用16位立即数。

立即数寻址	Add R4，#3	Regs[R4]←Regs[R4]+3	数值是常量

32-bit Constants

i_32bit.s

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant **4,000,000**

  `lui rt, constant` **=61*65536+2304**
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
                # 61 decimal = 0000 0000 0011 1101 binary
lui $s0, 61     0000 0000 0011 1101 0000 0000 0000 0000

ori $s0, $s0, 2304  0000 0000 0011 1101 0000 1001 0000 0000
                # 2304 decimal = 0000 1001 0000 0000  binary
```

Immediate Operands

- **Constant data specified in an instruction**

```
addi $s3, $s3, 4
```

- **No subi (subtract immediate) instruction**
  - Just use a negative constant

```
addi $s2, $s1, -1
```

2.3存储器寻址

- 立即数寻址方式

立即数寻址常用于算术运算指令、载入常数到寄存器指令、比较指令（条件转移指令/置条件转移指令中的条件）。

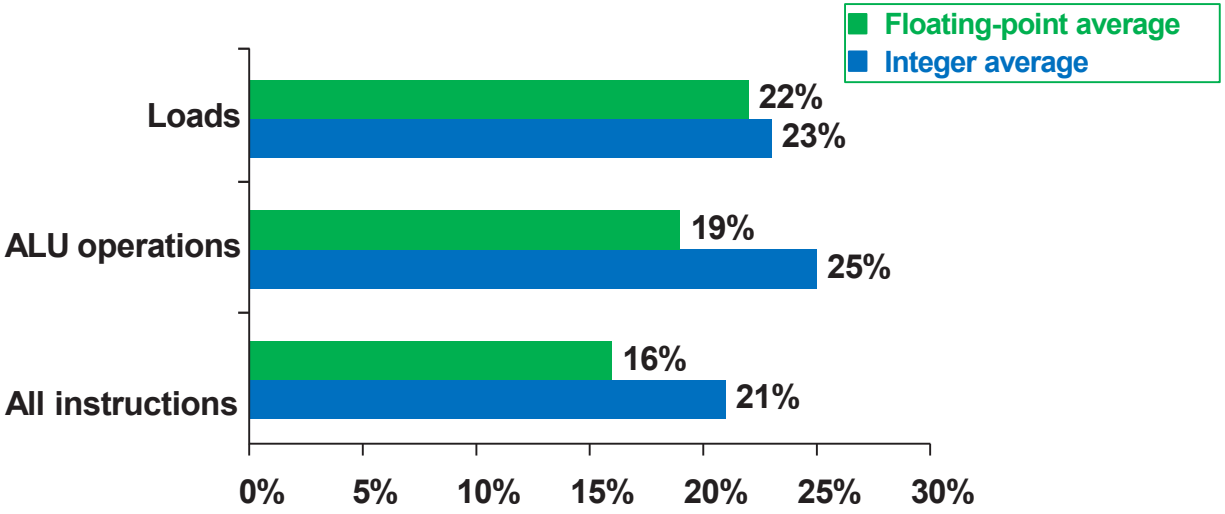
立即数寻址：是支持所有操作还是只支持一部分操作对设计指令系统很重要。下面的统计图表列出了Alpha运行SPEC2000测试程序使用立即数的频率。

立即数寻址	Add R4，#3	Regs[R4]←Regs[R4]+3	数值是常量

2.3存储器寻址

- 立即数寻址方式（Alpha 上测试）

Figure A.9



指令类型	使用频度	
	整型平均	浮点平均
Load指令	23%	22%
ALU指令	25%	19%
所有指令	21%	16%

2.3存储器寻址

- 立即数寻址方式
- 大约1/4的数据传输和定点ALU操作有一个立即数操作数。
- 定点程序中有约1/5的指令用到立即数，在浮点程序中这个比例约为1/6。

指令类型	使用频度	
	整型平均	浮点平均
Load指令	23%	22%
ALU指令	25%	19%
所有指令	21%	16%

立即数寻址	Add R4，#3	Regs[R4]←Regs[R4]+3	数值是常量

2.3存储器寻址

❖寄存器间接寻址

- 指令实例：Add R4 , (R1)
- 用途：使用指针或者计算出的地址进行寻址
- 含义：
  - Regs[R4]←Regs[R4]＋Mem[Regs[R1]]

寄存器间接寻址	Add R4，（R1）	Regs[R4]←Regs[R4]+Mem[Regs[R1]]	使用指针或者计算出的地址进行寻址

2.3存储器寻址

❖索引寻址

- 指令实例：Add R3 , (R1 + R2)
- 用途：有时用于数组中，R1是数组的基址，R2是索引值
- 含义：
  - Regs[R3]←Regs[R3]＋Mem[Regs[R1]+Regs[R2]]

索引寻址	Add R3，(R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+[Regs[R2]]	有时用于数组中，R1是数组的基址，R2是索引值

2.3存储器寻址

❖ 直接寻址或绝对寻址

- 指令实例：Add R1 , (1001)
- 用途：用来存取静态数据；地址常量可能需要很大
- 含义：
  - Regs[R1]←Regs[R1]＋Mem[1001]

直接寻址	Add R1，（1001）	Regs[R1]←Regs[R1] +Mem[1001]	用来存取静态数据；地址常量可能需要很大

2.3存储器寻址

❖ 存储器间接寻址

- 指令实例：Add R1，@(R3)
- 用途：如果R3是指针p的地址，那么就得到*p
- 含义：
  - Regs[R1]←Regs[R1]＋Mem[Mem[Regs[R3]]]

存储器间接寻址	Add R1，@（R3）	Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]	如果R3是指针p的地址，那么就得到*p

2.3存储器寻址

❖ 自动递增寻址

- 指令实例：Add R1 , (R2)+
- 用途：用在循环中递增变量，R2是数组的起始地址，每次增加d
- 含义：
  - Regs[R1]←Regs[R1]＋Mem[Regs[R2]]
  - Regs[R2]←Regs[R2]＋d

自动递增寻址	Add R1，（R2）+	Regs[R1]←Regs[R1]+Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	用在循环中递增变量，R2是数组的起始地址，每次增加d

2.3存储器寻址

❖ 自动递减寻址

- 指令实例：Add R1, -(R2)
- 用途：和自动递增类似，自动递增/递减用来实现类似栈的push/pop功能
- 含义：
  - Regs[R2]←Regs[R2]－d
  - Regs[R1]←Regs[R1]+Mem[Regs[R2]]

自动递减寻址	Add R1，-(R2)	Regs[R2]←Regs[R2]-d Regs[R1]←Regs[R1]+Mem[Regs[R2]]	和自动递增类似，自动递增/递减用来实现类似栈的push/pop功能

2.3存储器寻址

❖比例寻址或缩放寻址

- 指令实例：Add R1，100(R2)[R3]
- 用途：用来对数组寻址。一些计算机可以用任意的索引（间接）寻址方式
- 含义：
  - Regs[R1]←Regs[R1]＋Mem[100＋Regs[R2]＋Regs[R3]*d]

比例寻址	Add R1，100(R2)[R3]	Regs[R1]←Regs[R1] +Mem[100+Regs[R2]+Regs[R3]*d]	用来对数组寻址。一些计算机可以用任意的索引（间接）寻址方式

2.3存储器寻址

- 寻址方式的确定：为了能区分出各种不同寻址方式，必须在指令中给出标识。标识的方式通常有两种：显式和隐式。

显式

OP	MOD	A

隐式

OP	A

2.3存储器寻址

- 小结

- 一般ISA支持的基本寻址方式：位移量寻址、立即数寻址、寄存器寻址、寄存器间址。
- 位移量为13~16位；立即数为16位。
- 立即寻址通常用于：运算类指令、置常数到寄存器指令

Classroom Test

- (COD 5E Exercise 2.5) Rewrite the assembly code to minimize the number of MIPS instructions needed to carry out the same function.
- `addi $t2, $t0, 4`
- `lw  $t0, 0($t2)`