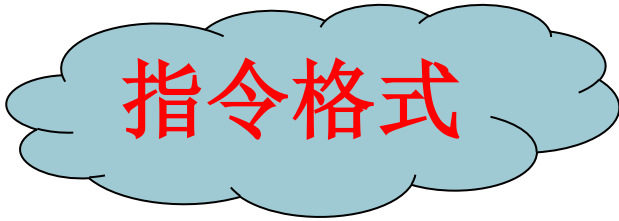
Chapter 2

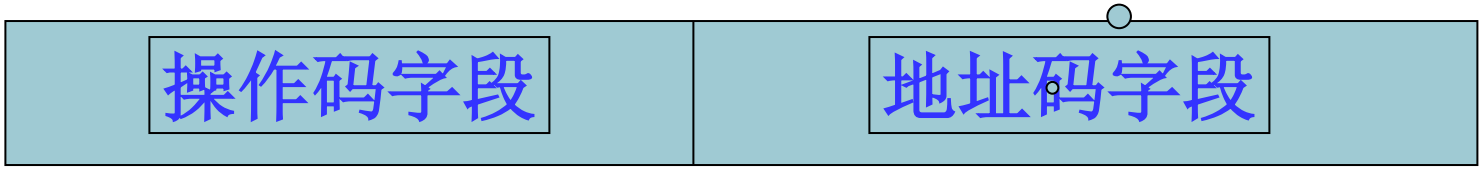
指令：计算机的语言

- 本章的主要目的，学习指令的基本格式、掌握指令的寻址方式，以MIPS指令集为例，理解指令的设计原则、MIPS指令与编程语言中结构的相关性、软/硬件间的接口。

2.0认识指令

- 什么是指令、指令中应该包含哪些内容？
- 知道要做什么
- 知道要处理哪些数，数在哪里
- 知道处理的结果放在哪里
- 知道下一条指令在哪里





2.0.1 操作码

- 每一条指令都要规定一个操作码。指令的操作码OP表示该指令应进行什么性质的操作，如进行加法、减法、乘法、除法、取数、存数等等。不同的指令用操作码字段的不同编码来表示，每一种编码代表一种指令。
- 组成操作码字段的位数一般取决于计算机指令系统的规模。较大的指令系统就需要更多的位数来表示每条特定的指令。
  - 等长（指令规整，译码简单）
    - 例如IBM 370机，该机字长32位，16个通用寄存器R0～R15，共有183条指令；指令的长度可以分为16位、32位和48位等几种，所有指令的操作码都是8位固定长度。
    - 固定长度编码的主要缺点是：信息的冗余极大，使程序的总长度增加。

Why?

2.0.2 地址码

- 根据一条指令中有几个操作数地址，可将该指令称为几操作数指令或几地址指令。
  - 三地址指令
  - 二地址指令
  - 单地址指令
  - 零地址指令

不是所有指令都需要3个操作数

操作码（4位）	A1（6位）	A2（6位）	A3（6位）
操作码（4位）	A1（6位）	A2（6位）	
操作码（4位）	A1（6位）		
操作码			

2.0.3指令操作码的扩展

通过扩展操作码来解决定长指令遇到的问题

操作码（4位）	A1（6位）	A2（6位）	A3（6位）
操作码（4位）	OP（6位）	A2（6位）	A3（6位）
操作码（4位）	OP（6位）	OP（6位）	A3（6位）
操作码（4位）	OP（6位）	OP（6位）	OP（6位）

给出这个格式的指令系统的能力？

2.0.4影响计算机指令格式的因素

- 机器的字长（反映了机器的能力）
- 存储器的容量（多少个单元、单元的容量）
- 指令的功能（简单、复杂）

2.0.5 指令在计算机系统中的地位

指令是软件和硬件分界面的一个主要标志，也是软硬件设计人员之间沟通的桥梁：

- 硬件设计人员采用各种手段实现它；
- 软件设计人员则利用它编制各种各样的系统软件和应用软件。

2.1引言　　指令集

- 一台计算机的全部指令
- 不同计算机有不同的指令集
  - 但有许多共同的方面
- 早期计算机有很简单的指令集（RISC）
  - 实现简单
  - 目前RISC-V
- 许多现代计算机也有简单指令集
- 　例如：ARM系列计算机
- 存储程序。

MIPS指令集

- 用于编写书中的例子使用的是MIPS语言
- 斯坦福大学的MIPS来自MIPS公司（www.mips.com）
- 在嵌入式处理机市场中占有很大的份额
  - 广泛地应用在家用电器、网络/存储设备，照相机，打印机...
- 许多现代指令集的代表
  - 参见MIPS参考数据卡和附录B和E。
  - RISC-V

2.2计算机硬件的操作

- 加法和减法，都是三操作数
  - 两个源操作数，一个目的操作数

```
add  a,  b,  c   # a = b + c
```

- 注释用#开头
- 所有的算术运算都是这样的形式
- *设计原则一：简单源于**规整***
  - 规整使实现简单
  - 简单能获得低成本高性能

算术操作例子

- C 语言的语句:

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

- 编译成MIPS代码:

```
add t0, g, h    #临时变量t0 = g + h
add t1, i, j    #临时变量t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

2.3 计算机硬件的操作数：寄存器操作数

- 算术运算指令使用寄存器操作
- MIPS有32个32位 寄存器
  - 用于存储频繁使用的数据
  - 编号0 到31
  - 32位数码称为一个字
- 编译时名称的约定
  - $t0, $t1, …, $t9 表示**临时寄存器**
  - $s0, $s1, …, $s7 用于存储变量
- *设计原则二：**越少越快***
  - 对照主存：数以百万计的存储位置

寄存器操作数举例

- C 语言编写的代码:

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

  - f, ..., j in $s0, ..., $s4

- 编译成MIPS代码:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

2.3.1存储器操作数

- 主存可以存储复杂数据
  - 数组，结构，动态数据
- 使用算术运算操作数
  - 从主存把数读入到寄存器
  - 把结果从寄存器存储到主存
- 存储器*按字节编址*
  - 每个地址表示一个8位字节
- *按字存放*在内存
  - 每个地址必须是4个字节
- MIPS按大端编址
  - 高位存放在低地址
  - *对照小端模式：低位放到低地址*

如果将一个16位的整数0x1234存放到一个短整型变量（short）中。这个短整型变量在内存中的存储在大小端模式由下表所示。

地址偏移	大端模式	小端模式
0x00	12（OP0）	34（OP1）
0x01	34（OP1）	12（OP0）

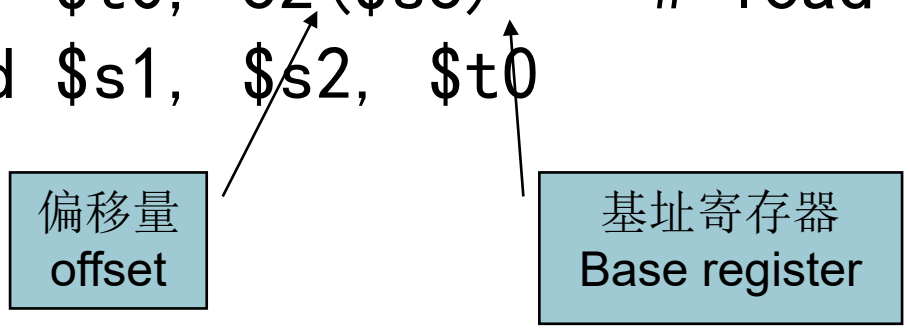
存储器操作数举例1

- C code:

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

  - g 在 $s1, h 在 $s2, A的基址在 $s3
- 编译成MIPS代码:
  - 下标8（数组第8个分量）需要32的偏移
    - 每个字对应4个字节

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```



存储器操作数举例2

- C code:

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

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

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

寄存器和主存储器

- 寄存器的访问速度比主存快得多
- 对主存储器数据的操作要用
- 取数指令lw（load word）
- 存数指令sw（store word）。
  - 需要执行更多的指令
- 编译器**必须尽量使用寄存器访问变量**
  - 仅当寄存器不够用时才把不经常使用的变量放到内存；
  - 寄存器的高效利用对系统优化非常重要。

2.3.2常数或立即数操作数

- 指令中使用常数

  addi $s3, $s3, 4（immediate）

- 没有减去立即数的减法指令
  - 可以使用负常数，即“加负数”实现减法

    addi $s2, $s1, -1

- *加速执行常用操作*
  - 小常数操作出现的频率高
  - 立即数操作不用到内存取数

常数零

- MIPS寄存器0（$zero）表示常数0
  - 不能被改写
- 在常用的操作中，很有用
  - 例如，可在寄存器之间传送数据

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

2.4有符号数和无符号数：无符号数

- 给定一个n位数

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围: 0 to $+2^n - 1$
- 举例
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- 32位数
  - 0 to +4,294,967,295

二进制补码表示有符号整数

- x是一个n位数

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围: -2^{n-1} to $+2^{n-1} - 1$
- 举例
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- 32位数
  - –2,147,483,648 to +2,147,483,647

二进制补码表示有符号整数

- 31位是符号位
  - 1表示负数
  - 0表示非负
- $-(-2^{n-1})$ 不能表示
- 非负数的无符号数和二进制补码是相同的
- 一些特殊的数
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - 绝对值最大负数: 1000 0000 … 0000
  - 最大正数: 0111 1111 … 1111

有符号数（二进制补码）的取反

- 取反加1
  - 取反的含义1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- 举例：对+2的求反
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

符号扩展

- 使用更多的位表示一个数
  - 数值保持不变
- 在MIPS指令集中
  - addi: extend immediate value扩展到立即数
  - lb, lh: extend loaded byte/halfword 扩展到取字节/半字
  - beq, bne: 扩展到跳转后的位置extend the displacement 相等/不相等跳转
- 复制符号位到左侧
  - 对照无符号数：用0扩充
- 举例: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

2.5计算机中指令的表示

- 指令用二进制编码
  - 称为机器代码
- MIPS 指令
  - 32位指令字编码
  - 指令格式中若干字段分别用于表示操作码，寄存器编号，…
  - 非常规整
- 寄存器编号【参考教材71页图2-14】
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

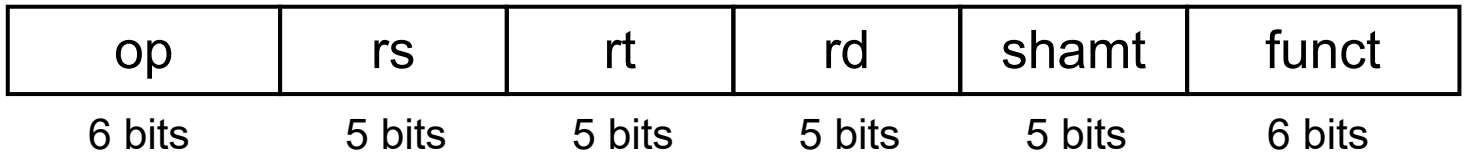
指令格式的演变

- 四地址指令；
- 三地址指令；
- 二地址指令；
- 一地址指令；
- 零地址指令。
- 地址的显式和隐含式

常用的寻址方式

- 寄存器型：直接、间接；
- 存储器型：直接、间接；
- 计算型：相对、基址、变址；
- 立即数型；

MIPS中寄存器指令



- 指令字段
  - op：操作码(opcode)
  - rs：第一个源寄存器编号
  - rt：第二个源寄存器编号
  - rd：目的寄存器编号
  - shamt：移位位数(00000 表示不移位)
  - funct：功能码（扩展操作码）（extends opcode)

R-型（寄存器）操作举例

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

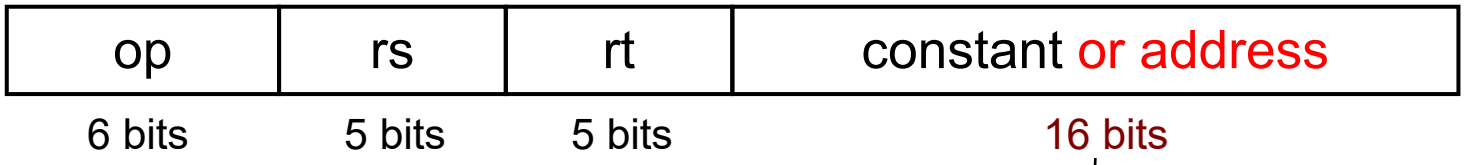
十六进制

- 基底是16
  - 二进制串的压缩表示（使用16进制的原因）
  - 四位二进制组成一个十六进制数

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I型（立即数指令）



- 立即数的算术和读数/存数指令
  - rt：目的或源寄存器编号
  - 常数的取值范围：-2^{15} to $+2^{15} - 1$
  - 地址：偏移加上rs中的基址
- *设计原则3：优秀的设计需要适宜的折中方案*
  - 不同类型指令采用不同的解码方式，但都是32位相同的指令长度
  - 尽可能保持相似的指令格式

指令格式举例1

- 如果数组A的基址放在$t1中，h存放在$s2中，这C语言写的语句：
- A[300]=h+A[300]编译成汇编语言，则为：
- lw　　$t0, 1200($t1) # $t0=A[300]
- Add　　$t0, $s2, $t0　# $t0=h+A[300]
- Sw　　$t0, 1200($t1)　#把h+A[300]存到原来A[300]所在的单元。

指令格式举例1-2

汇编语言编写的程序:

```
lw    $t0, 1200($t1)
Add   $t0, $s2, $t0
Sw    $t0,1200($t1)
```

寄存器编号

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

变成机器语言，如下:

op	rs	rt	rd	Address /shamd	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

指令格式举例1-3

op	rs	rt	rd	Address /shamd	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

汇编语言编写的程序:

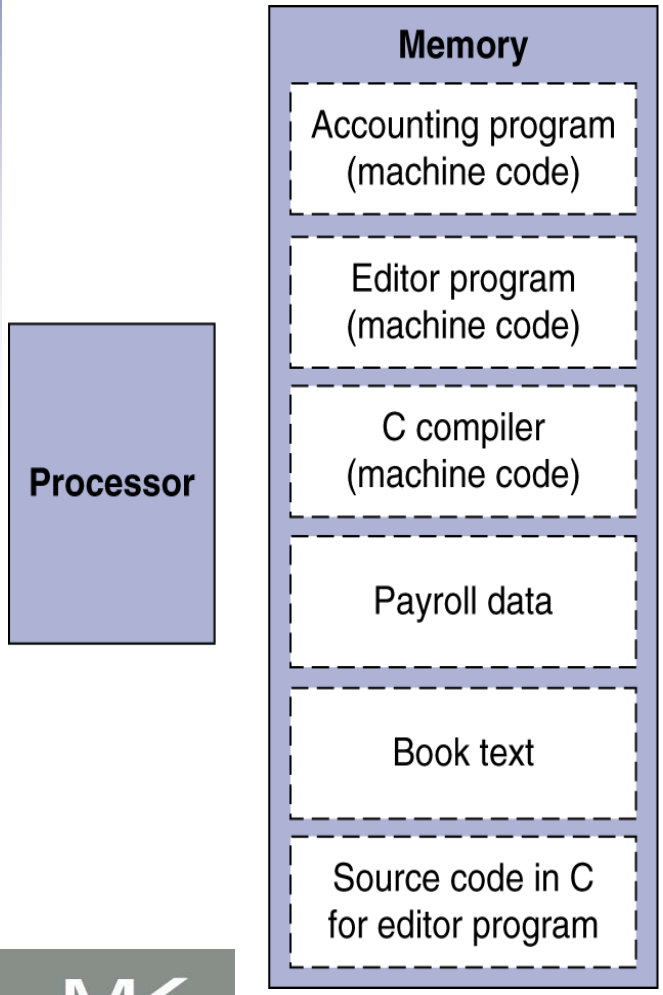
```
lw    $t0, 1200($t1)
Add   $t0, $s2, $t0
Sw    $t0,1200($t1)
```

存在存储器中的形式为上表的2进制

: op	rs	rt	rd	Address /shamd	funct
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

存储程序

The BIG Picture



- 指令用二进制表示，像数一样
- 指令和数据存储在存储器中
- 程序可以被其他程序操作
  - 例如，编译，连接，…
- “二进制兼容”可以让编译后的程序运行在不同的计算机上
  - 标准的ISAs（软件编程的标准）

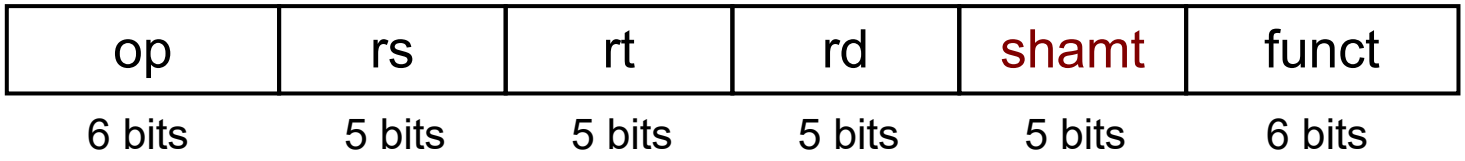
2.6逻辑操作

- 对位进行处理的指令

逻辑操作	C	Java	MIPS
左移	<<	<<	sll
右移	>>	>>>	srl
按位与	&	&	and, andi
按位或	|	|	or, ori
按位取反	~	~	nor

- 用于对字中的若干 *“位”*
- 打包和拆包的操作

移位操作

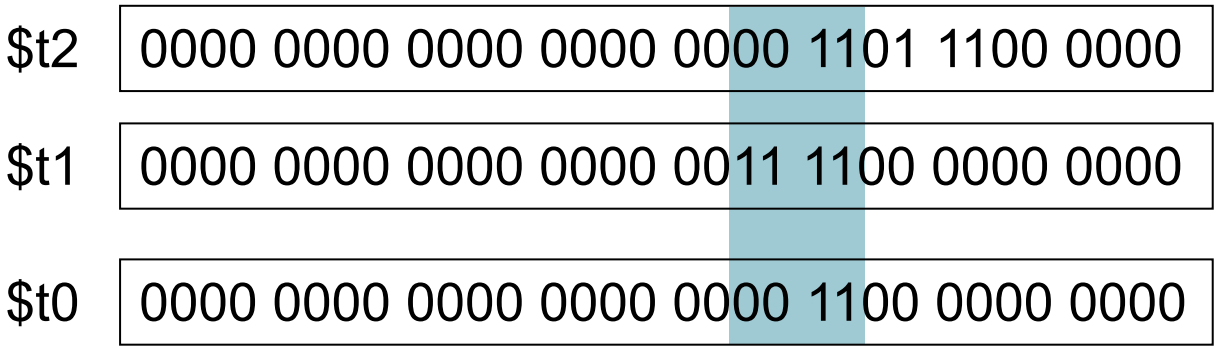


- shamt：移多少位
- 逻辑左移sll
  - 左移空位填0
  - 逻辑左移i位相当于乘2^i
- 逻辑右移srl
  - 逻辑右移空位填0
  - 逻辑右移i位相当于除2^i（仅对无符号数）

“与”操作

- 可用于一个字中的掩码操作
  - 选择某些位，其他位清零

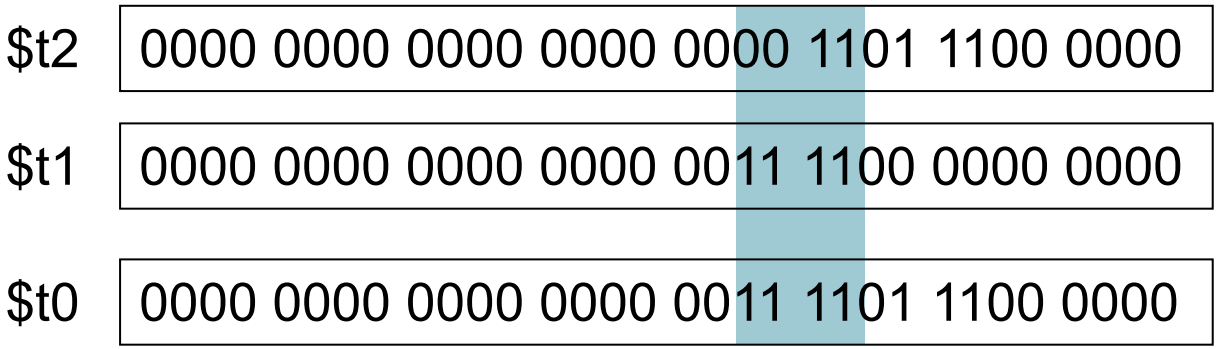
```
and $t0, $t1, $t2
```



“或”操作

- 用于把包含字中的一些位置1，其他位不变

```
or $t0, $t1, $t2
```



按位“取反”操作

- 用于改变字中的一些位
  - 0变成1, 1变成0
- MIPS 3-操作数指令NOR
  - a NOR b == NOT (a OR b) “或非”

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

2.7 决策指令

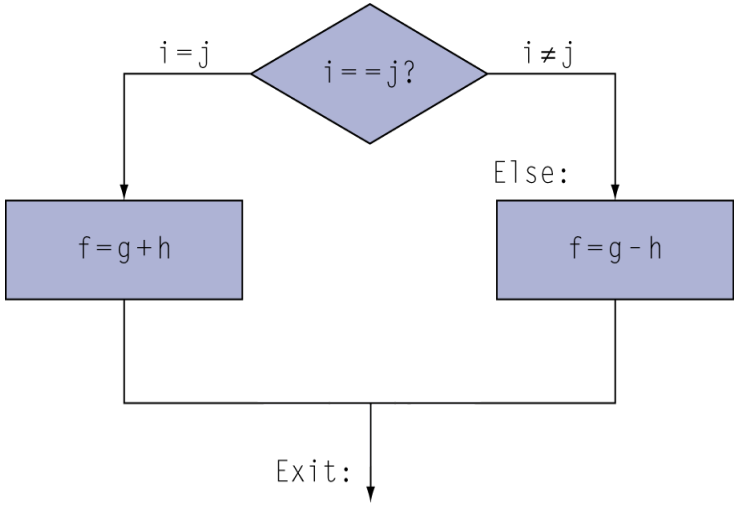
- 如果条件为真，跳转到被标签的指令执行
  - 否则，继续执行
- beq rs, rt, L1
  - if (rs == rt)转到标签为L1的指令执行
- bne rs, rt, L1
  - if (rs != rt)转到标签为L1的指令执行;
- j L1
  - 无条件跳转到标签为L1的指令执行

编译IF语句

- C code:

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

  - f, g, … in $s0, $s1, …



- 编译成MIPS 代码:

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

Assembler calculates addresses

编译循环语句

- C code:

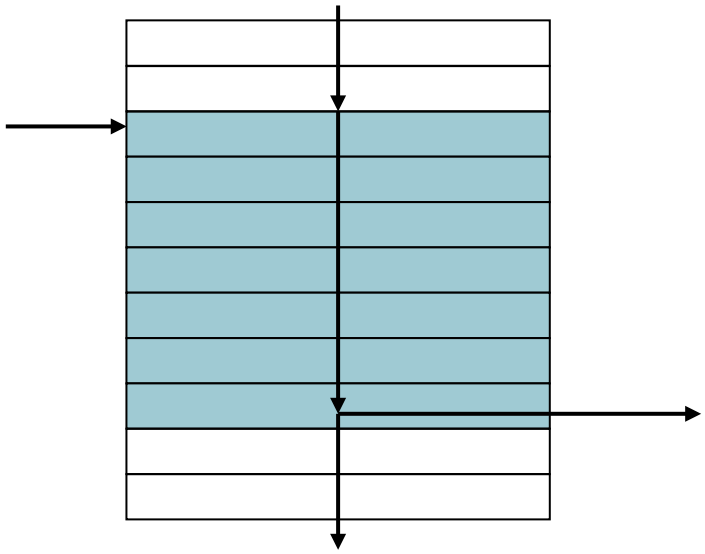
```
while (save[i] == k)
i += 1;
```

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

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

基本块

- 一个基本块是一个指令序列，其中
  - 内部没有跳出的指令（结束指令除外）
  - 也没有被跳转到的指令（开始指令除外）



- 编译器标识基本快用于优化
- 高级处理机能够加速基本块的执行

更多的条件操作

- 如果条件为真置1，否则置0
- slt rd, rs, rt；小于则置位
- if (rs < rt) rd = 1; else rd = 0;
- slti rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;
- beq，bne可以和其他指令结合使用

```
slt $t0, $s1, $s2  # if ($s1 < $s2)$t0=1;
bne $t0, $zero, L  # $t0 not equal to
zero  branch to L
```

分支指令设计

- 为什么没有blt等指令？
- 硬件执行<，≥，…比=，≠慢
  - 指令中结合分支指令包含更多工作，需要更慢的时钟
  - 所有指令都受到了影响
- Beq和bne是较常用的
- 这是一个很好的设计折中方案

有符号数和无符号数对比

- 有符号数比较：slt，slti
- 无符号数比较：sltu，sltui
- 举例
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - slt $t0, $s0, $s1 # signed
    - –1 < +1 ⇒ $t0 = 1
  - sltu $t0, $s0, $s1 # unsigned
    - +4,294,967,295 > +1 ⇒ $t0 = 0

过程调用

- 遵循步骤
  1. 将参数放在过程可以访问的寄存器里
  2. 将控制转移给过程
  3. 获得过程所需要的存储资源
  4. 执行过程的操作（请求的任务）
  5. 将结果的值放在调用程序可以访问到的寄存器
  6. 将控制返回到调用点

寄存器的使用

- $a0 – $a3: 传递参数 (reg's 4 – 7)
- $v0, $v1: 返回结果值 (reg's 2 and 3)
- $t0 – $t9: 临时寄存器
  - 可以被调用者改写
- $s0 – $s7: 保存参数
  - 必须被调用者保存和恢复
- $gp: 静态数据的全局指针寄存器(reg 28)
- global pointer for static data (reg 28)
- $sp: 堆栈指针寄存器stack pointer (reg 29)
- $fp: 帧指针寄存器(frame pointer)，
- 保存过程帧的第一个字 (reg 30)
- $ra:返回地址寄存器 return address (reg 31)

过程调用指令

- 过程调用：跳转和链接

  jal ProcedureLabel

  - 下一条指令的地址在寄存器$ra中
  - 跳转到目标地址
- 过程返回:寄存器跳转 jump register

  jr $ra

  - 拷贝$ra到程序计数器
  - 也被用于运算后跳转
  - 例如用于case/switch分支语句e.g

不调用其他过程（叶过程）例子

- C code:

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

  - 参数 g, ..., j 在 $a0, ..., $a3中
  - f 在 $s0 (因此，需要存储$s0到堆栈)
  - 结果在$v0

不调用其他过程的例子

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

存储 $s0 到堆栈

g+h----t0

i+j---t1　过程体
(g + h) - (i + j)

结果　(s0----v0)

恢复 $s0

返回

参数 g, …, j 在 $a0, …, $a3中;f 在 $s0 (因此，需要存储$s0到堆栈);结果在 $v0

嵌套过程—非叶过程(Non-Leaf Procedures)

- 过程调用其他过程
- 对于嵌套调用，调用者需要存储到堆栈的信息:
  - 它的返回地址
  - 调用后还需要用的任何参数寄存器和临时寄存器
- 调用后返回，寄存器会从堆栈中恢复

嵌套过程举例 Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

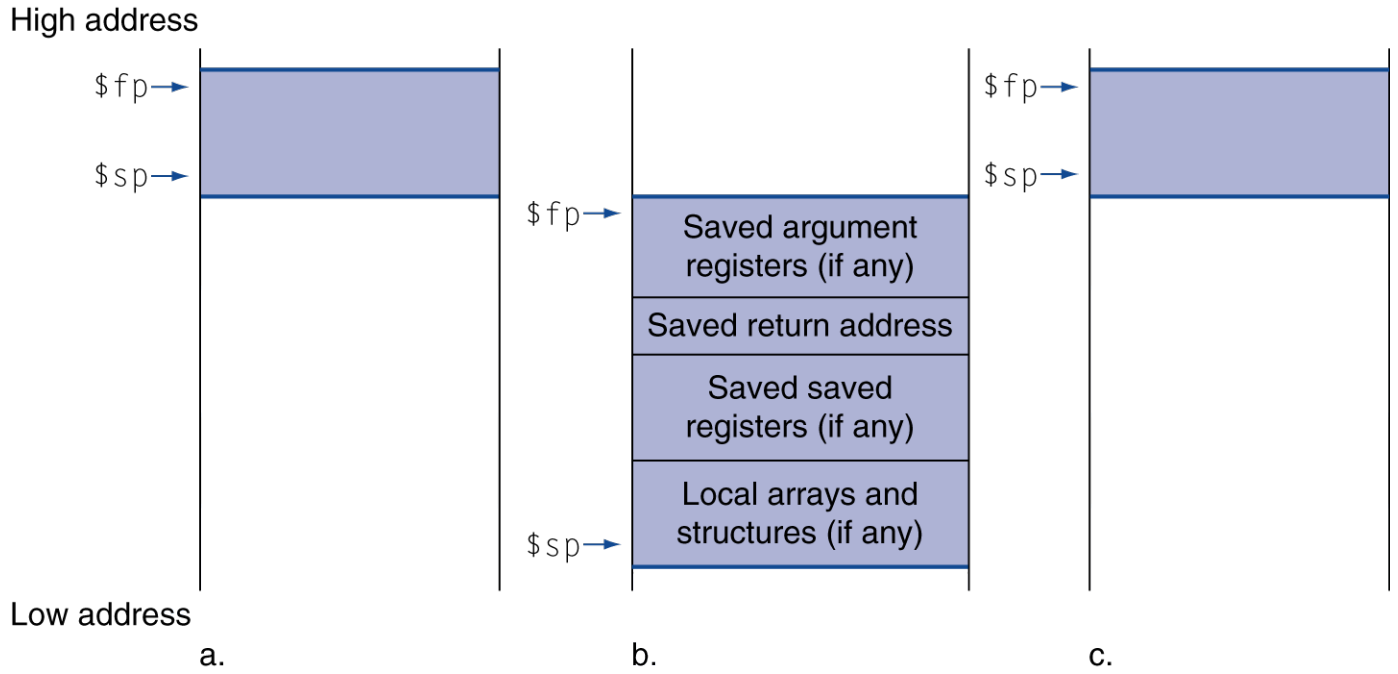
  - 参数n放在 $a0
  - 结果放在 $v0

嵌套过程举例 Non-Leaf Procedure Example

- MIPS code:

```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument（存参数）
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call(递归调用)
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```

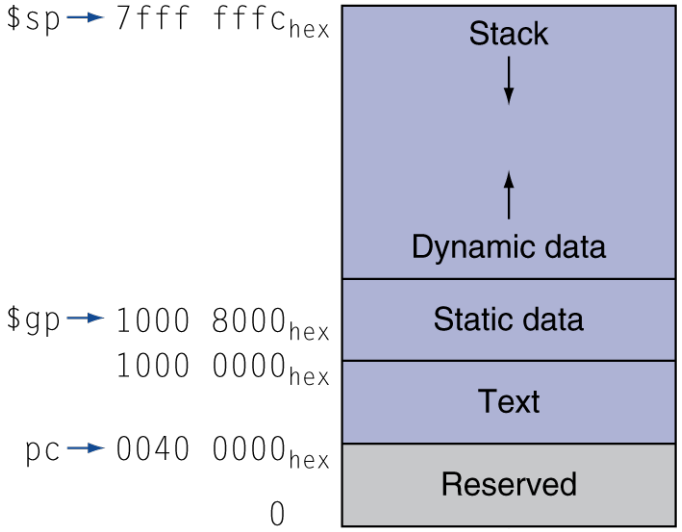
堆栈中的局部数据 Local Data on the Stack



- 局部数据有调用者分配
- e.g., C 自动分配变量
- 过程帧（活动记录）
  - 被一些编译器使用控制堆栈存储

内存布局Memory Layout

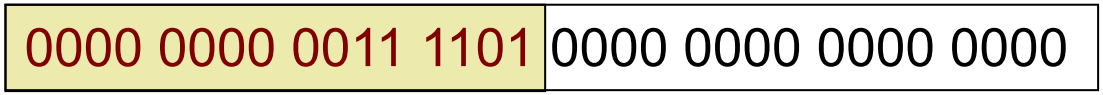
- 正文：程序代码
- 静态数据：全局变量
- e. g.， C语言静态变量，常数数组和字符串
  - $gp 初始化地址，允许段内的± 偏移
- 动态数据：堆
- 堆栈：自动存储



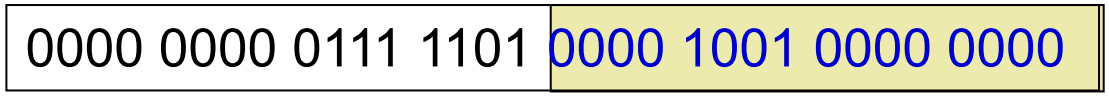
32位立即数和寻址

- 大部分常数都比较小，16位表示立即数足够了，偶尔使用32位常数。
- `lui rt, constant`
  - 取立即数并放到高16位

```
lui $s0, 61
```



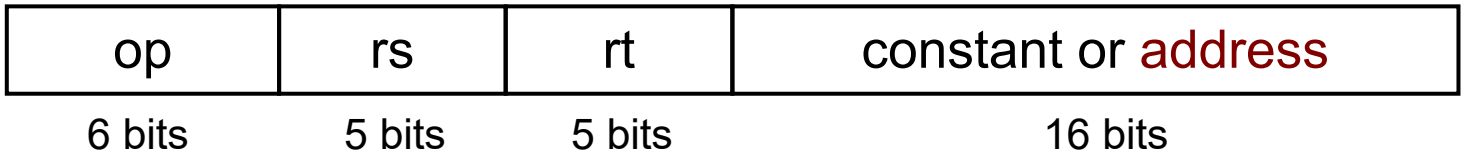

```
ori $s0, $s0, 2304
```



Ori立即数或， $s0与常数2304“或”，结果放在$s0形成一个32位的常数

分支地址

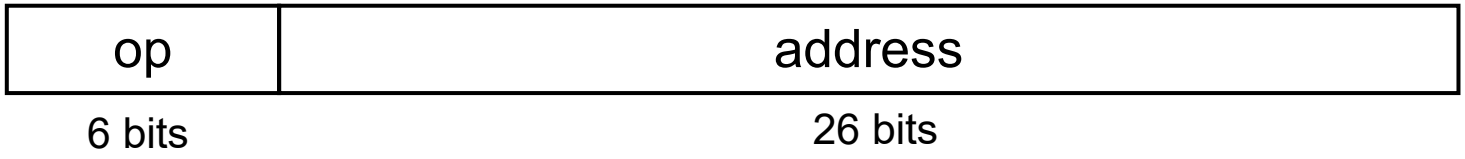
- 分支指令说明
  - 操作码，两个寄存器，两个地址
  - 大多数跳转目标离跳出的位置较近
  - 向前或向后



- PC相对寻址
  - 目标地址 = PC + offset × 4
  - 此时PC的增加量是4的倍数

跳转地址

- 跳转（j和jal）的目标地址可以在代码段的任何位置
  - 指令除op外，指令其它字段都是地址



- 直接跳转到地址
- Target address = $PC_{31...28}$: (address × 4)

4位　　28位

目标地址

- 早期例子的循环代码
- 设循环的起始地址是8000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

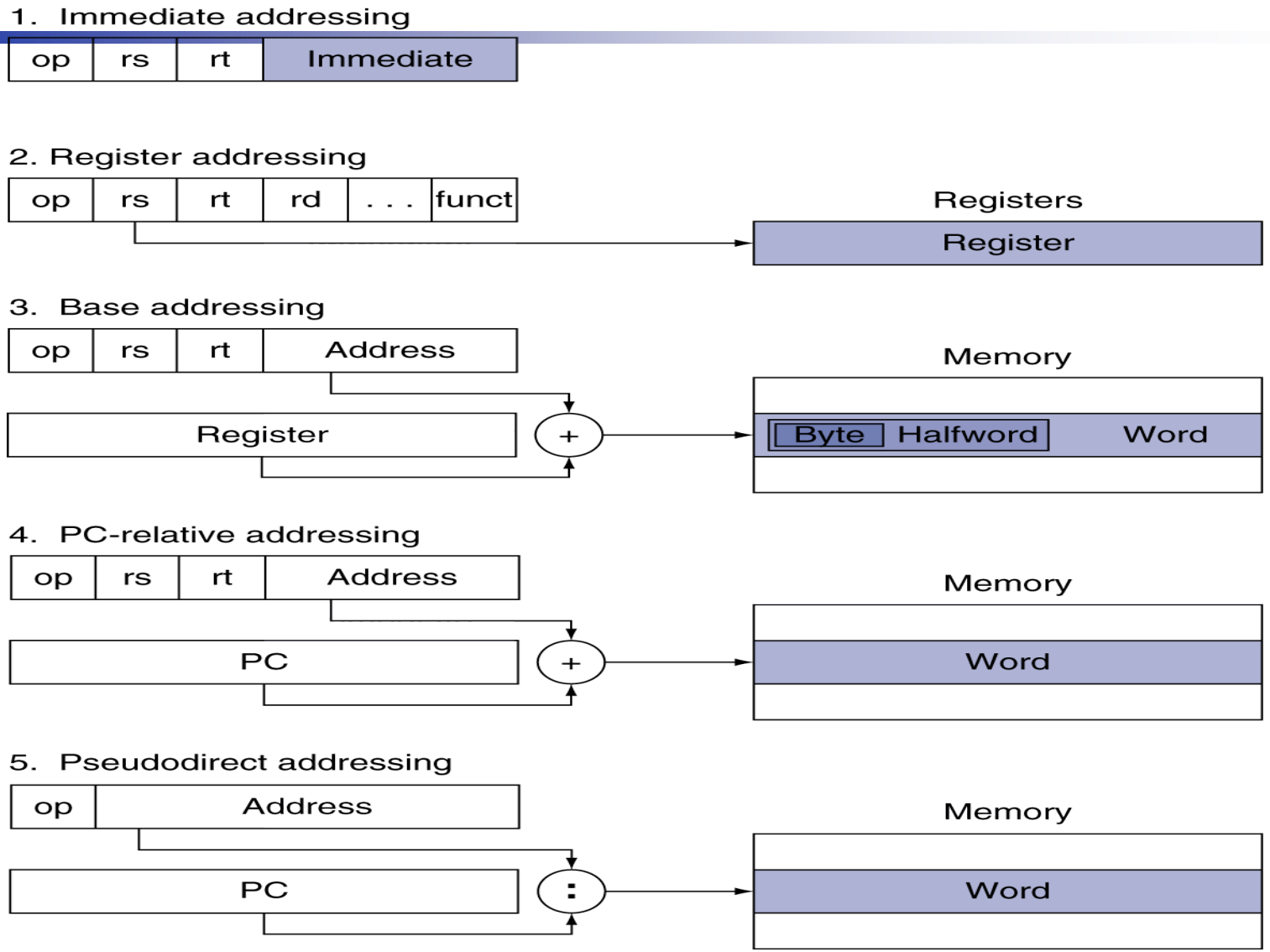
0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

远程分支

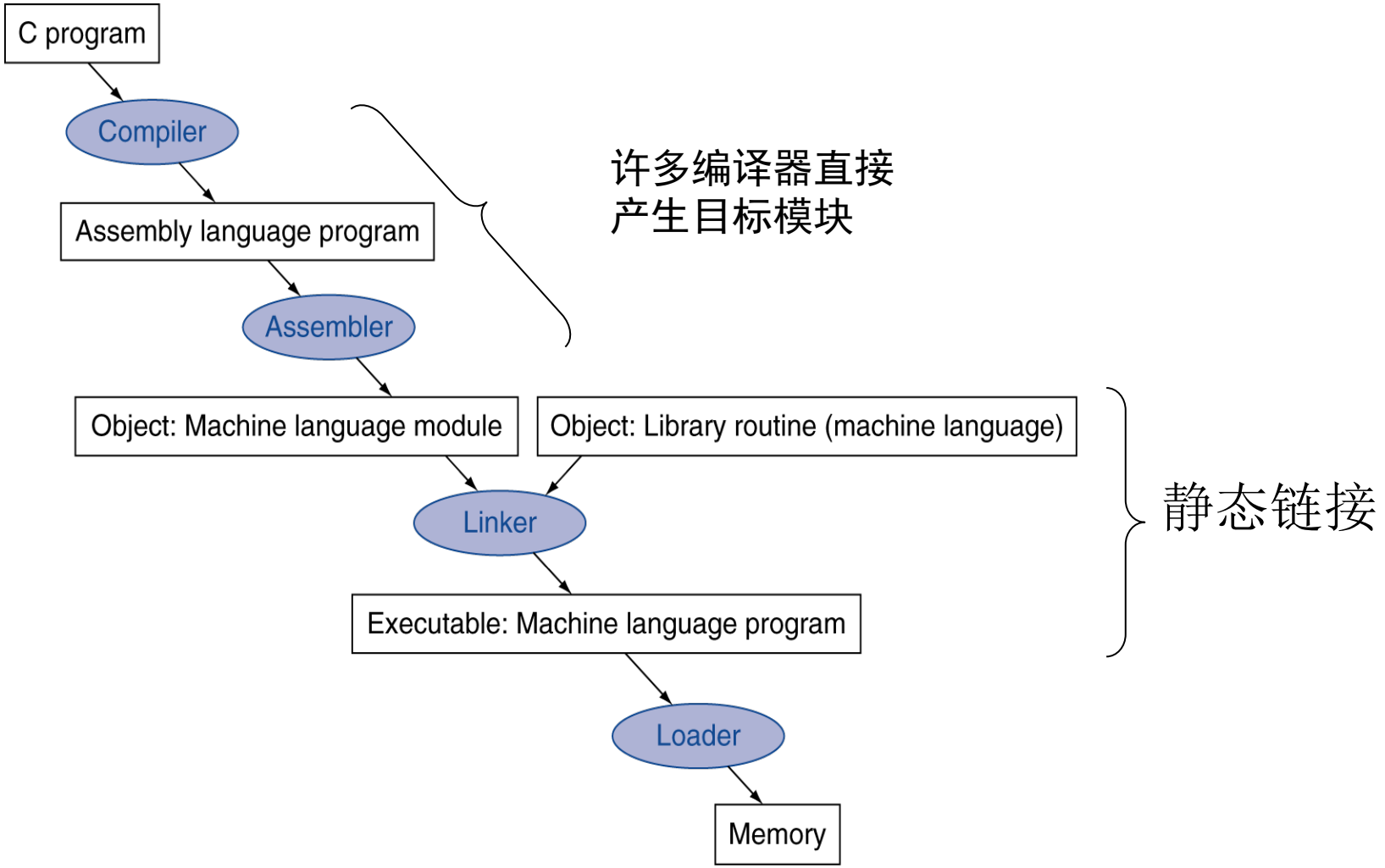
- 如果跳转对象地址太大无法用16位的偏移表示，汇编将重写代码
- 【把短跳转（2^{16} 范围）
- 变成长跳转（2^{26} 范围）】
- Example

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

地址模式总结 Addressing Mode Summary



编译并执行程序 Translation and Startup



汇编伪指令 Assembler Pseudo Instructions

- 大多数汇编指令和机器指令是一对一的
- 特殊的是伪指令
- 伪指令：汇编指令的变种
- `move $t0, $t1` → `add $t0, $zero, $t1`

  `blt $t0, $t1, L` → `slt $at, $t0, $t1`
  `bne $at, $zero, L`
  - $at (register 1): 汇编程序的临时寄存器（使用汇编语言时候的硬件额外的开销）

生成目标模块Producing an Object Module

- 汇编器（或编译器）把程序翻译成机器语言
- 提供从部分构建完整程序的信息
- 目标文件头：描述目标文件其他部分的大小和位置
  - 正文段：翻译后的指令，包含机器语言代码
  - 静态数据段：包含在程序生命周期内分配的数据
  - 重定位信息，标记了一些程序加载进内存时依赖于绝对地址的指令和数据
  - 符号表，全局定义和外部引用
  - 调试信息：用于关联源文件

链接目标模块 Linking Object Modules

- 产生一个可执行的映像
  1. 合并段（代码和数据数据库象征性放入内存）
  2. 决定数据和指令标签的地址
  3. 修补引用（内部和外部引用）

- 可以留下依靠重定位程序修复的部分
  - 但虚拟内存，不需要做这些
  - 虚拟内存空间，程序必须以绝对地址装入

加载程序Loading a Program

把待执行的程序从硬盘的镜像文件读入内存

1. 读取可执行文件头来确定正文段和数据段的大小
2. 为正文和数据创建一个足够大的地址空间
3. 把指令和初始数据拷贝到内存或者设置页表项，使它们可用
4. 把主程序的参数复制到栈顶
5. 初始化寄存器（包括堆栈指针$sp，帧指针$fp，全局指针$gp ）
6. 跳转到启动进程
   - 复制参数到寄存器并调用主函数main
   - 主函数返回时，通过系统调用exit终止程序