中国地质大学研究生培养模式与教学改革基金项目资助

计算机体系结构新讲

JISUANJI TIXI JIEGOU XINJIANG

罗忠文 杨林权 陈 亮 龚君芳 编著

中国地质大学出版社
ZHONGGUO DIZHI DAXUE CHUBANSHE

中国地质大学研究生培养模式与教学改革基金项目资助

计算机体系结构新讲

JISUANJI TIXI JIEGOU XINJIANG

罗忠文　杨林权　陈　亮　龚君芳　编　著

CUGP
中国地质大学出版社
ZHONGGUO DIZHI DAXUE CHUBANSHE

图书在版编目(CIP)数据

计算机体系结构新讲/罗忠文等编著. —武汉:中国地质大学出版社,2015.12
ISBN 978-7-5625-3418-1

Ⅰ.计…
Ⅱ.①罗…
Ⅲ.①计算机体系结构-研究-
Ⅳ.①TP303

中国版本图书馆 CIP 数据核字(2015)第 257669 号

计算机体系结构新讲 罗忠文 杨林权 陈 亮 龚君芳 编 著

责任编辑:阎 娟 责任校对:张咏梅

出版发行:中国地质大学出版社(武汉市洪山区鲁磨路 388 号) 邮政编码:430074
电 话:(027)67883511 传 真:67883580 E-mail:cbb@cug.edu.cn
经 销:全国新华书店 http://www.cugp.cug.edu.cn

开本:787 毫米×1092 毫米 1/16 字数:227 千字 印张:8.875
版次:2015 年 12 月第 1 版 印次:2015 年 12 月第 1 次印刷
印刷:武汉市籍缘印刷厂 印数:1—500 册

ISBN 978-7-5625-3418-1 定价:35.00 元

前　言

计算机结构与组成是计算机及信息相关专业的一门重要基础课，主要是讨论底层软、硬件系统的设计，对于深入学习软、硬件系统有着重要意义。对于计算机及相关专业的学生，在学习了基本的高级语言之后，渴望进一步深入了解高级语言为什么能够实现人们所期望的功能，以及计算机硬件到底是如何工作的原理。但通常的计算机专业把这些隐藏在了多门课程中，除了极少数同学能够通过多年的学习和总结厘清其中的关系外，很多同学即使本科毕业了也未能完全理解计算机的工作原理。本书试图用比较精简的篇幅来概括性地介绍计算机从高级语言编写的程序到最终硬件执行的整个过程，以便计算机及信息相关专业的学生对计算机软、硬件结构及构成有一个总体了解。因此，该书一方面可以作为掌握了基本计算机高级语言的学生了解计算机深层次工作原理的书籍，同时对于经过多年计算机相关知识的学习，希望从总体上认识计算机结构的研究生及其他读者也有参考价值。

本书以设计和实现 RISC 结构的 MIPS 处理器作为主线，重点介绍计算机的总体轮廓和处理器设计中的一些主要原则及重要的思想方法，而不拘泥于细节。这样处理的优点是使读者能很快对计算机动作结构有一个总体认识，从而对计算机不再陌生，同时通过理解和掌握计算机设计中的一些核心思想方法，对未来的工作有所启迪。而对于那些追求细节的读者，可以参考业内其他重要的著作。

本书内容大致上分为以下几个方面。其一是计算机的系统软件，拟从计算机高级语言出发，介绍更低级的汇编语言，最后到机器语言。在整个介绍过程中，将重点关注语言间的转换。通过这个过程，让同学理解编译器及解释器的工作原理。其二是对处理器设计的具体过程进行分析。其三是介绍外设的访问方法及一些重要的思想，如轮循、中断、RAID 等。其四是通过计算机体系结构来提高计算的性能，包括利用流水线结构提高处理器的性能，通过高速缓存来提高内存的访问速度，通过虚拟内存技术来扩大内存空间，并且更重要的是提供了多个程序访问内存的一种良好机制。同时讨论了面向大数据时代的仓库式数据中心等。

当然，作为一本引论性质的教科书必须有所取舍，这些取舍一定程度上体现了笔者的偏好和笔者看问题的角度。因此，就选材及其他方面难免存在缺点和错误，欢迎读者对本书提出批评建议。

罗忠文

2015 年 8 月

目　录

第一章　汇编语言与汇编指令 …… (1)
第一节　概述 …… (1)
第二节　汇编指令 …… (1)
第三节　汇编指令中的操作数:寄存器 …… (2)
第四节　汇编指令中的操作数:立即数 …… (3)
第五节　汇编指令中的操作数:内存 …… (4)
第六节　MIPS 程序控制指令 …… (6)
第七节　函数调用 …… (10)
第八节　逻辑运算 …… (15)
第二章　指令表示 …… (18)
第一节　以数的形式出现的指令 …… (18)
第二节　反汇编 …… (23)
第三节　伪指令 …… (24)
第三章　浮点数 …… (27)
第一节　浮点数的表示 …… (29)
第二节　IEEE754 浮点数标准 …… (30)
第三节　特殊数的表示法 …… (31)
第四节　MIPS 浮点数结构 …… (33)
第四章　程序的运行 …… (34)
第一节　解释和翻译 …… (34)
第二节　汇编器 …… (36)
第三节　链接器 …… (37)
第四节　装入器 …… (38)
第五节　综合例子 …… (38)
第五章　电路基础与基本计算模块 …… (44)
第一节　同步数字系统 …… (44)
第二节　信号与波形 …… (45)
第三节　状态单元 …… (47)
第四节　组合逻辑电路的表示 …… (51)
第五节　布尔代数 …… (52)
第六节　组合逻辑块 …… (53)
第六章　CPU 设计 …… (57)
第一节　CPU 设计引论 …… (57)
第二节　单周期指令 CPU 的数据通道设计 …… (60)
第三节　单周期指令 CPU 的控制通道设计 …… (67)

第七章　流水线改进性能 …… (73)
第一节　流水线结构概述 …… (73)
第二节　结构困境 …… (74)
第三节　控制困境 …… (75)
第四节　数据困境 …… (78)
第八章　存储设计 …… (81)
第一节　高速缓冲存储器(cache) …… (81)
第二节　cache 索引 …… (83)
第三节　内存读写 …… (89)
第四节　虚拟内存 …… (97)
第九章　输入输出 …… (101)
第一节　输入输出概述 …… (101)
第二节　网络 …… (104)
第三节　磁盘 …… (107)
第十章　基于 SOPC 设计 CPU …… (110)
第一节　QuartusII 概览 …… (110)
第二节　使用库中模块设计费波拉契数计算器 …… (111)
第三节　分层设计实现一个计数器 …… (115)
第四节　用 FPGA 完成一个 16 位 CPU 的设计概述 …… (118)
第五节　基于 FPGA 设计寄存器文件 …… (120)
第六节　基于 FPGA 设计 ALU 和程序记数器 …… (123)
第七节　基于 FPGA 设计存储器 …… (125)
第八节　基于 FPGA 设计控制逻辑 …… (127)
第九节　相关问题的讨论 …… (130)
附录 …… (133)
主要参考文献 …… (135)

第一章　汇编语言与汇编指令

第一节　概　述

在日常生活中我们要使用各种语言来进行交流。在和计算机交流时，同样需要使用计算机自己的语言，这种语言计算机能懂，且能控制计算机执行的语言称为机器语言。正如人类使用多种语言且差异很大一样，不同的计算机系统，其机器语言也不一样。但相对来说，不同的机器语言之间的差异并没有人类语言差异那么大。因此，学会一种机器语言后，理解其他机器语言也比较容易。

对学习和使用者来说，总希望语言简单而高效。但针对不同的主体，简单具有不同的含义，即以计算机作为主体来看待程序的简单性和以人作为主体来看待程序的简单性完全不同。对人类而言，由于多年的发展和学习，有了高度的抽象能力，因此，抽象的语言对其而言是简单的。而对计算机，或者说是实现处理器来说，其简单则主要表现在语言元素少，如就单词而言仅仅只有 0 和 1，而操作也只有开和关。事实上，计算机就其根本上来讲正是由电路的开、关（或者开关电路）所组成的。

人类用于表示计算的高级语言和计算机本身表示计算的开关电路有很大的区别，将两者联系起来理解整个计算机的工作原理并非易事。而本书的目的之一，就是通过自上而下和自下而上两种方式来展现其间的细节，从而将其有效地联系起来。

在本章中，将采取自顶向下的方法，以 MIPS 计算机及其汇编语言为例，具体介绍如何将计算机高级语言编写的程序转换成低级汇编语言的程序。

第二节　汇编指令

MIPS 汇编语言是一种比较接近计算机底层的低级语言。在高级语言中，有各种语句和运算，汇编语言同样是通过指令语句来实现的。如对应于 C 语言的下列语句：

a＝b＋c

对应有下面的汇编语言语句：

add a，b，c

对应于普通语言的语法和单词，汇编语言有规定的格式，称为指令格式。一般来说，每条汇编指令与计算机能执行的一种基本操作相对应。在以上汇编指令语句中，add 称为汇编指令运算符，也称作操作符，而后面的 a，b，c 称为操作数。

为了简化处理器的设计，MIPS 处理器采用了固定结构的汇编指令，每个指令由 4 个部分组成，包含 1 个操作符和 3 个操作数。MIPS 汇编指令的格式如下：

指令代码 操作数 1，操作数 2，操作数 3

在高级语言中，使用变量前，首先要声明并给定一个类型。每个变量只能表示其所声明

的类型的值，不能混用，如比较整型和字符型变量。但是在汇编语言中，寄存器没有数据类型，运算（符）确定将寄存器内容当成什么数据类型来处理。

当前使用的计算机都是有精度的。目前主流的台式机是 32 位的，而一些低端的计算机则多是 8 位的。无论怎样，在进行算术运算时，都可能超出计算机字长所能表示的范围，从而产生溢出。例如对于在 4 位处理器上进行下列无符号数加法运算：

```
  +15        1111
   +3        0011
  ---       ----
  +18       10010
```

二进制数 1111 和 0011 相加的正确结果应是 10010，但是因为 4 位计算机没有地方保存结果的第 5 位数，所以计算结果是 0010，即十进制的+2，从而产生溢出错误。对于这种溢出错误，高级语言通常有两种处理方法：一种是忽略，即当什么事都没有发生一样，C 语言采用的是这种方式；另一种是检测出这种错误，并以某种方式（如异常）通知程序，ADA 语言采用此种方式。

为此，MIPS 提供了两种算术指令，检测溢出的指令和不检测溢出的指令，以适应不同高级语言的需求。具体指令如下：检测溢出的加（add）、减（sub）法指令，不检测溢出的加（addu）、减（subu）法指令。因此对于加法运算，MIPS C 编译器将产生 addu，而对 MIPS ADA 编译器则产生 add。

第三节　汇编指令中的操作数：寄存器

MIPS 处理器的汇编语言中，算术运算指令的操作数只能是寄存器。这样做的目的是使硬件设计更简单。而规定指令由 4 部分组成，即规则性，也在确保软件指令语法具有统一规则，同样也使得硬件设计较简单。

在 MIPS 计算机中设计了 32 个 32 位的通用寄存器来作为汇编指令的操作数，分别编号为 0，1，2，…，31。在汇编语言程序中可以通过 $0，$1，$2，…，$31 来指定这些寄存器。而为了汇编语言程序设计人员的方便，每个寄存器还分别定义了一个名字，因此也可以通过寄存器名来指定要访问的寄存器。

前面介绍的汇编语言加法指令，正确的写法可以是：

```
add $8, $9, $10      (用寄存器号指定要访问的寄存器)
```

或者

```
add $s0, $s1, $s2    (用寄存器名指定要访问的寄存器)
```

为了方便汇编语言程序设计，通常将 32 个寄存器分成若干类，其中第一类是 $s 寄存器，如 $s0，$s1，…，$s7。通常对应于 C 语言的变量，如要翻译 C 语言程序：

```
c=a-b
```

可将 C 语言的变量和汇编语言的 $s? 寄存器作如下对应：

C 语言变量名	a	b	c
汇编语言寄存器名	$s0	$s1	$s2

则上面的 C 程序可翻译成汇编程序：

```
sub   $s2, $s0, $s1
```

其中 sub 是减法指令。上面介绍的 C 语言语句，正好是对 2 个数进行运算，得到 1 个结果，一共涉及 3 个操作数。但对于其他 C 语言指令语句，其操作数不一定正好是 3 个，可能更多，也可能更少，如对如下的 C 语言指令语句：

```
v= (a+b) - (c+d)
```

可以使用多条汇编语句，把中间结果存储在临时寄存器 $t? 中来实现翻译，一种可能的结果如下：

```
add   $t0, $s1, $s2     # t0=a+b
add   $t1, $s3, $s4     #t1=c+d
sub   $s0, $t0, $t1     #v= (a+b) - (c+d)
```

对于其他更多操作的指令可以类似地实现。但对于具有更少操作数的指令，如赋值语句：f=g，该如何处理呢？MIPS 处理器作了一个聪明的处理，定义了一个稍微特殊一点的寄存器 $0（零号寄存器，名字表示为 $zero）。该寄存器的值永远都是 0，利用该寄存器即可实现赋值运算如下：

```
add   $s0, $s1, $zero
```

指令执行完后，寄存器 $s0 中的值与 $s1 中的值相同。

另外，由于计算机运算速度很快，对于一些要求计算机速度慢下来的应用需求，经常需要进行延时，因此要求在处理器中有一条不作任何处理，仅延时一个时钟周期的指令 noop，该指令没有任何操作数。利用 $0，我们可以实现该指令：

```
add   $0, $0, $0
```

第四节　汇编指令中的操作数：立即数

在 C 语言中，除了对变量相加之外，还经常需要加 1 个常数，如 a=a+1。当然，该指令可以通过把常数放到寄存器中，然后利用前面的 add 指令来实现，但这样做的结果是需要用两条指令才能完成一个运算。由于指令运算在 C 语言中出现非常频繁，因此，MIPS 设计了专门的指令来实现它，从而能加快计算的速度。这里，把数值常数称为立即数，意即该数在代码中可以立即获得，而不必先放到寄存器中，再从寄存器中取出。

对应于 C 语言中的 f=g+10 的汇编指令如下：

```
addi   $s0, $s1, 10
```

这里操作符是 addi，是一种新的运算，表示将数值常数 10 与存储在寄存器 $s1 中的数相加，其语法和 add 指令的区别在于最后一个操作数是数，而不是寄存器。

一种自然的推广是接着定义立即数减法指令。但事实上，在 MIPS 中没有立即数减法指令。原因在于可以通过加负常数来实现减正常数。如指令 subi $s0, $s1, 10 可以通过 addi $s0, $s1, -10 来实现。

这是一个 MIPS 设计的重要准则，让最少量的指令来实现功能。如果指令可以分解（或者减化）为更简化的形式，就略去该指令。

思考：既然不需要 subi 指令，是否也不需要 sub 指令呢？

第五节 汇编指令中的操作数：内存

第三节介绍了如何把 C 变量映射到寄存器，同时 MIPS 处理器仅有 32 个寄存器，如何处理超过 32 个元素的数组等数据结构呢？可以将其存储到空间足够大的内存（memory）中，但 MIPS 算术指令不能直接操作内存，仅能直接操作寄存器，因此要操作内存中数据，必须先将其从内存中装入到寄存器内，然后再对寄存器中的数据进行运算。因此需要指令实现内存和寄存器间的数据交换。

内存与寄存器数据交换指令有两种：一种是从内存中装入（load）数据到寄存器；另一种是将寄存器中的数据存储（store）到内存中。由图 1-1 可知，寄存器位于处理器的数据通道（datapath）中。如果操作数在内存中，必须将其传送到处理器中来处理，处理完后再传送回内存中。如果数据存放在寄存器中，MIPS 的一条算术指令就可以完成：读两个数，进行运算，并写结果；而 MIPS 的一条数据传送指令就仅仅只能读或写一个操作数，不能同时进行其他任何运算。

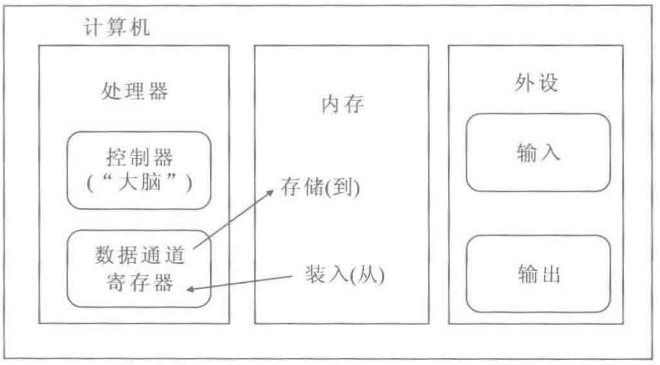

图 1-1 计算机组成部分示意图

一、数据交换：内存到寄存器

将数据从内存复制到寄存器中，需要指定将内存中具体某个位置的数据复制到指定的寄存器中。因此必须指定寄存器和内存的位置或地址。指定寄存器可以通过其编号（$0～$31）或者名字（$s0，…，$t0，…）来完成。指定内存则更复杂一些。

首先，可以把内存看成一个一维数组，这样就能简单地通过一个指向内存的指针来访问，可以通过一个寄存器来存储该指针。另外，通常访问某数据时，还会访问该数据附近的数据，当然这可以通过不断改变地址寄存器来实现。但这样完成一个读写内存需要两条指令：一条指令是改变地址寄存器，另一条指令实现从内存读入数据。一种更好的方案是，提供一个偏移量（offset）来实现访问附近的数据，从而只用改变偏移量常数即可访问附近的数据，这样一条指令即可实现数据访问。此时，内存地址为寄存器（包含指向内存的指针）和数值偏移量（以字节为单位）之和。

例如：8（$t0），当 $t0 的值为 1000 时，表示的内存地址为 1000+8=1008

从内存装入数据的指令语法格式如下：

1　2，3（4）

其中：1 表示指令名字；2 表示接收内存传入值的寄存器名；3 表示数值偏移量（单位：字节）；4 是寄存器名，该寄存器中存储的是内存地址，其中括号表示地址。MIPS 中装入字数据的指令为：lw（即 load word，每次装入 32 位即一个字）。

例如：lw $t0，12（$s0）

该指令所执行的操作是：取出寄存器 $s0 中的数据与立即数 12 相加，将结果作为内存地址，从内存中取值，把取得的值放入到 $t0 中。一般称 $s0 为基寄存器，常数 12 称为偏移量。

值得指出的是此处偏移量必须是常数（即在编译时已知）。因此，可以通过将基寄存器指向结构体的起始位置，改变偏移量来访问结构体数据中的各个元素。但对于动态循环访问数组的各个元素，则需要改变基寄存器的值来实现。

二、数据传送：寄存器到内存

如果希望把寄存器的值存储（store）到内存中去，则其语法和 load 类似。在 MIPS 中存储数据的指令为：sw（即 store word，每次存储 32 位即一个字）。

例如：sw $t0，12（$s0）

该指令所执行的是：取出 $s0 中的指针，加上 12 字节，得到内存地址的值，然后把寄存器 $t0 中的值存储到该内存地址中。在这里强调一个重要概念，就是寄存器可以保存任意 32 位数值。该值可以是有符号整数（signed int），无符号整数（unsigned int），指针 pointer（内存地址）等。但注意不要混用，对于指令 add $t2，$t1，$t0 而言，$t0 和 $t1 中存放的是数值；而对于指令 lw $t2，0（$t0）而言，$t0 中存放的是指针。

三、寻址（编址）模式

内存中的每个字都有地址，形式上和数组中的下标类似。一种内存的编址方法和 C 语言的数组下标一样：

Memory [0]，Memory [1]，Memory [2]，…

其中 Memory [i] 表示内存中的第 i 个字。而另一种方法，是以字节作为单位来编址，即 Memory [i] 表示内存中的第 i 个字节。由于 4 个字节表示一个字，因此，在字节编址的计算机中，表示各个字的地址是：

Memory [0]，Memory [4]，Memory [8]，…

在现代计算机中，既要访问字节，也要访问字，主要是采用第二种方式即字节编址，其相邻两个 32 位（4 字节）字的地址相差 4 个字节。

例：确定 C 语言中整型变量 A [5] 的偏移量。

需要把变量 A [5] 的地址转换为字节编址，相当于把 5 个字转化成为 20（5 * 4 = 20）字节。具体实例如下：

g = h + A [5]

其中，g 表示 $s1，h 表示 $s2，$s3 表示 A 的基地址。

手工编译以上语句的结果为（编译结果首先将数据从内存传到寄存器中）：

```
lw   $t0, 20 ($s3)      #装入数据到寄存器$t0，从内存地址20+（$s3）中装入
add  $s1, $s2, $t0      #$t0中的值与h相加，结果存在g中
```

需要注意的是，计算机操作的单位是“字”，而机器的编址是字节序列，所以相邻字之间的地址相差4个字节，而不是1个字节，这经常造成汇编程序员犯错，取下一个地址时简单加1。另外，谨记对于lw和sw，基址和偏移量之和必须是4的倍数，即需要字对齐，如果字没有对齐，则读数据操作将需要两次才能完成，因为硬件本身是字对齐逐字读取的。

四、字节数据的传送

除了传送字数据的指令lw，sw外，MIPS还有字节传送指令：lb（load byte）和sb（store byte）。其格式与lw，sw是相同的。

例如：lb $s0, 3 ($s1)

该指令表示为把位于（$s1）+3内存地址的内容复制到寄存器s0中，但寄存器有32位，传送的数据只有8位，这8位数据是放在寄存器的低字节中的。同样，对于指令sb，也是把寄存器中的低8位数据传送到指定的内存地址中，只是传送方向变了。如：

```
sb  $s0, 3 ($s1)
```

对于装入指令，面临的另一个问题是对于32位寄存器的高24位该如何处理。MIPS的lb是采用符号扩展方式填充高24位的，这样可以保证通常的有符号整数装入后的32位数与之前的8位数是同一个数。但是在把字符编码看成数时，通常无符号数，因此不应该做符号扩展，对于这种需求，MIPS提供了另一条指令：lbu（load byte unsigned），用来装入字节，并对高位数据进行零扩展。即装入8位数据到寄存器的低字节中，其余24位补0。

第六节 MIPS程序控制指令

一、基本MIPS分支指令

到目前为止，所学的指令只能操作数据，这样构建出来的只是某种形式的计算器，如果要建立计算机，就需要能进行判断（决策）。在C语言中提供了丰富的判断语句，但在汇编语言中，我们希望使用尽量少的判断语句，来实现C语言中丰富的判断功能。首先，考查C语言中最基本的条件语句，在C语言中有两种形式的if语句：

```
if (condition) clause; (1)
if (condition) clause1 else clause2; (2)
```

我们可以使用“goto”语句，将第二种形式的if语句转换为第一种形式：

```
if (condition) goto L1;
    clause2;
    goto L2;
L1: clause1;
L2:
```

这里“goto”语句实现跳转到语句标号（labels）处。对应于第一种形式的if语句，有下面的MIPS的分支（决策）指令：

beq　register1，register2，L1

其含义是：如果 register1 与 register2 相等，则跳转到标号 L1 处的语句，否则不执行。该语句对应的 C 语句是 if（register1＝＝register2）goto L1。

此外，还有一条与上面指令对应的进行不相等跳转的 MIPS 分支指令：

bne　register1，register2，L1

它和 C 语言的 if（register1！＝register2）goto L1 具有相同的含义，即如果 register1 与 register2 不相等，则跳转到 L1 标号处。

上面介绍的两条指令称为条件分支指令，除了条件分支外，MIPS 还有无条件分支：

j　label

该指令又称为跳转指令，即：不必满足任何条件而直接跳转（或分支）到指定的标号处。它和 C 语言的 goto label 意义相同。该语句形式上可以用条件跳转语句来实现：

beq　$0，$0，label｜　（由于$0 恒为零，因此条件总是满足）

下面来看一个将 C 语言条件语句转换为汇编语言的真实实例，程序流程图如图 1－2 所示。

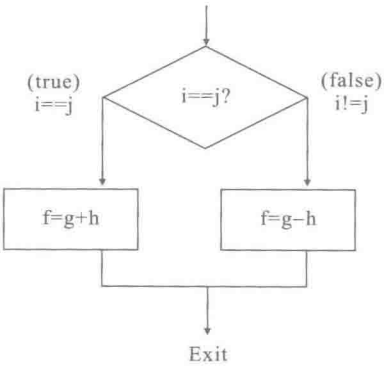

图 1－2　分支结构示意图

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

定义如下变量映射关系：

f：$s0　g：$s1　h：$s2　i：$s3　j：$s4

编译成的最终 MIPS 代码为：

```
beq $s3, $s4, True      # branch i==j
sub $s0, $s1, $s2       # f=g-h (false)
j  Fin                  # goto Fin
True: add $s0, $s1, $s2 # f=g+h (true)
Fin:
```

注意：在高级语言代码中是没有标号的，但为了处理分支，编译器会自动生成标记（labels）。

二、用分支指令实现循环控制

1. 两个“逻辑”指令

逻辑左移指令是 sll，具体例子如：

```
sll  $s1, $s2, 2      #s1=s2≪2
```

该指令的功能是将寄存器 $s2 的值左移 2 位，并将结果存放到 $s1 中，在右边空出来的位中补 0，和 C 语言的“≪”运算符等价。假定 $s2 的值是 2，则

$s2 的值为：

0000 0002（16 进制）　0000 0000 0000 0000 0000 0000 0000 0010（二进制）

执行之后为 $s1 的值，为：

0000 0008（16 进制）　0000 0000 0000 0000 0000 0000 0000 1000（二进制）

从以上结果可以看出，左移二位实现了乘 4（$=2^2$），推广到左移 n 位的情形，其结果是实现乘以 2^n。与逻辑左移指令类似，逻辑右移指令为：srl，它相当于 C 语言中的“≫”

运算符。

2. 循环

下面是一段很简单的C语言循环程序，其中A [] 是一个整型数组：

```
do {
        g=g+A [i];
        i=i+j;
} while (i ! = h);
```

要把它翻译成MIPS程序。首先，重写以上程序为条件语句的形式：

```
Loop: g=g+A [i];
        i=i+j;
        if (i ! =h)
        goto Loop;
```

然后，使用如下映射关系：

g -> $ s1；h -> $ s2；i -> $ s3；j -> $ s4；A的基地址-> $ s5

则可把上面语句翻译成为如下的MIPS代码：

```
Loop: sll $ t1, $ s3, 2          # $ t1=4 * i，乘4得字节地址
        add $ t1, $ t1, $ s5     # $ t1=A的地址
        lw $ t1, 0 ($ t1)        # $ t1=A [i]
        add $ s1, $ s1, $ t1     #g=g+A [i]
        add $ s3, $ s3, $ s4     #i=i+j
        bne $ s3, $ s2, Loop     # if i! =h goto Loop
```

在C语言中有3种循环方式：while；do... while；for。这3种循环中的任何一种形式都可以用其他两种表达，因此前面例子中所用的方法同样可以应用于while和for循环。从上面的讨论中知道，尽管有多种流程控制方式，但其关键均是条件分支。

三、MIPS汇编中的不等式判断

到目前为止，仅能对是否相等进行判断，也就是C语言中的“==”和“!=”。通常程序还需要对小于“<”和大于“>”进行判断。因此，需要有对应的分支语句。一种可能的做法是提供4个条件判断语句：

小于分支：　　blt（branch less than）

大于分支：　　bgt（branch great than）

小于等于分支：　ble（branch less or equal）

大于等于分支：　bge（branch great or equal）

但在处理器设计中通常采用一种更聪明的做法，通过提供一条指令来实现4种条件分支。MIPS中的该不等式判断指令是：slt（Set on Less Than）。

其句法为：slt　reg1，reg2，reg3

与指令对应的C语句是：reg1= （reg2<reg3）；

逻辑更清晰的C语句是：if（reg2 < reg3）reg1=1；else　reg1=0；

利用slt指令，可以实现小于分支，如C语言的小于分支语句：

if (g<h) goto Less;　　#变量映射为 g：$s0，h：$s1

可以翻译成 MIPS 代码：

```
slt   $t0, $s0, $s1      # 如果 g<h，则 $t0=1
bne   $t0, $0, Less      # if $t0!=0 goto Less 即 (if (g<h)) Less
```

此处，寄存器 $0 的值恒为 0。前一句话，实现当条件成立时，$t0 赋值为 1，然后 1 不等于 0 实现分支。因此，指令 slt 与 bne 的组合实现了小于分支。

现在实现了对小于“<”的判断，但又如何实现“>”“≤”和“≥”呢？一种做法是新加 3 个指令：sgt，sle，sge，但这样与我们开始的设想不一致。事实上也违背了 MIPS 设计的准则，即指令数目越少越好。由于小于判断与大于等于判断互逆，而等于和不等于互逆，因此只需将前面的 bne 改为 beq 即可实现大于等于的判断，具体指令如下：

```
slt   $t0, $s0, $s1      # $t0=1 if a<b
beq   $t0, $0, GreatE    # GreatE if a>=b
```

对应的实现 a 小于等于 b 的指令，只用在上式中交换 a，b 的位置即可，指令如下：

```
slt   $t0, $s1, $s0      # $t0=1 if b<a
beq   $t0, $0, GreatE    # LessE if b>=a
```

同样的道理可以实现大于指令。

另外，为了与常数比较，MIPS 也设计了 slt 的立即数版本 slti，用来判断是否小于常数，即 slti。此运算可应用于循环中，如下面的 C 语句：

if (g>=1) goto Loop

转化成 MIPS 为：

```
slti  $t0, $s0, 1        # $t0 = 1 if $s0<1 (g<1)
beq   $t0, $0, Loop      # goto Loop
                         # if $t0==0 (if (g>=1))
```

同样，还有针对无符号数的不等式判断指令：sltu，sltiu，该指令将操作数看成无符号数来进行比较。需要注意的是，寄存器中的数是二进制串，本身是不存在“有符号”或“无符号”的定义的，其含义决定于指令，如对于如下寄存器数中的十六进制数：

$s0=FFFF FFFA　　$s1=0000 FFFA

不难看出，当看成是无符号数时 $s0 大一些，而看成有符号数时 $s0 是负数，就小一些。因此对于语句

```
slt  $t0, $s0, $s1
sltu $t1, $s0, $s1
```

$t0，$t1 将取不同的值。

另外，在 MIPS 的指令中无符号标识 u 具有不同的含义。如对于装入字节 lbu，是进行符号扩展；对于 addu 是不检测溢出，而 sltu 是进行无符号数比较。

四、将 switch 语句编译成汇编指令

以下 C 语言 switch 语句，实现根据 k 的取值选择 4 种可能的运算：

```
switch (k) {
        case 0: f=i+j; break;        /* k=0 */
```

```
        case 1: f=g+h; break;          /* k=1 */
        case 2: f=g-h; break;          /* k=2 */
        case 3: f=i-j; break;          /* k=3 */
        }
```

为了方便使用前面介绍的汇编指令，改写为如下 if-else 语句：

```
if (k==0) f=i+j;
    else if (k==1) f=g+h;
        else if (k==2) f=g-h;
            else if (k==3) f=i-j;
```

对于该语句的编译，使用以下变量映射：f -> $s0，g -> $s1，h -> $s2，i -> $s3，j -> $s4，k -> $s5。则可将以上代码转换为如下的 MIPS 指令：

```
    bne   $s5, $0, L1        # branch k! =0
    add   $s0, $s3, $s4      #k==0 so f=i+j
    j     Exit               # end of case, so Exit
L1: addi  $t0, $s5, -1       # $t0=k-1
    bne   $t0, $0, L2            # branch k! =1
    add   $s0, $s1, $s2      #k==1 so f=g+h
    j     Exit               # end of case, so Exit
L2: addi  $t0, $s5, -2       # $t0=k-2
    bne   $t0, $0, L3        # branch k! =2
    sub   $s0, $s1, $s2      #k==2 so f=g-h
    j     Exit               # end of case, so Exit
L3: addi  $t0, $s5, -3       # $t0=k-3
    bne   $t0, $0, Exit          # branch k! =3
    sub   $s0, $s3, $s4      #k==3 so f=i-j
Exit:
```

第七节　函数调用

在结构化程序设计中，函数起着非常重要的作用，本节就将讨论汇编语言中如何实现函数调用功能。

一、函数的跳转指令

逻辑上，函数调用是要跳转到函数实现的代码处执行函数，执行完后，就返回主调程序，然后继续执行下面的语句。很显然函数需要有跳转指令来支持，下面以这段 C 程序为例子，考虑如何将函数调用转化成 MIPS 语句。

```
main () { c=sum (a, b);}    /* a: $s0, b: $s1, c: $s2 */
int sum (int x, int y) {  return x+y;}
```

在 MIPS 处理器中，所有指令都占 4 个字节，和数据一样存储在内存中。因此一种可能

的翻译及在内存中的存储情况如下（第一列表示内存地址）：

```
1000          j     sum        #jump to sum
1004 end:     exit
...
2000     sum: add   $s2, $s0, $s1
2004          j     end
```

但以上代码有几个问题。其一是主程序中的变量 a，b 和函数中 x，y 直接共享了 $s0，$s1，没有体现函数的值传递过程。其二是函数 sum 执行完成后返回到固定位置，这对多次调用函数会出现问题，如对程序：

```
main () { c=sum (a, b); sum (b, c)}    /* a: $s0, b: $s1, c: $s2 */
int sum (int x, int y) {   return x+y;}
```

可能翻译成：

```
1000          j     sum        #jump to sum
1004 end1:    add   $s0, $s1, $0
1008          add   $s1, $s2, $0
1012          j     sum        #jump to sum
1016 end2:    exit
...
2000   sum:   add   $s2, $s0, $s1
2004          j     end?
```

最后一行的返回语句，当第一次调用 sum 时，应该返回到 end1，而第二次调用 sum 时，应该返回到 end2。但 j 语句只能跳转到固定位置，一种可能的解决此问题的方法是让返回值地址可变，即存到对应于 C 变量的寄存器中，并设计一条跳转到寄存器的指令。事实上 MIPS 处理器确实设计了此指令 jr（jump to register），其格式如下：

```
jr   $ra   #跳到寄存器所指定的地址
```

这里，MIPS 规定使用通用寄存器 $ra 来存储返回值地址。另外，实现函数之间参数传递的方式，在 MIPS 中采用的是将要传递的参数共享到寄存器变量中，具体使用的是 $a0，$a1，$a2，$a3 四个变量，其中 a 的含义是参数（argument）。上面程序正确翻译如下：

```
1000     add      $a0, $s0, $zero   #参数传递，将变量 $s0 的值赋给实参 $a0
1004     add      $a1, $s1, $zero
1008     addi     $ra, $zero, 1016  #存储函数返回地址
1012     j        sum               #调用函数 sum
1016     add      $a0, $s1, $zero
1020     add      $a1, $v0, $zero
1024     addi     $ra, $0, 1032     #存储函数返回地址
1028     jr       $ra               #调用函数 sum
1032     exit
...
2000     sum: add $v0, $a0, $a1
```