第5章

计算机指令集架构

前面介绍了以布尔逻辑与有限状态机为数学基础、CMOS电路为物理基础的数字电路分析与设计。如图5-1所示，从本章开始将讲解更高层、更通用也更复杂的计算系统：通用计算机的分析与设计。本章将从通用计算机的概念引入，以MIPS指令集为例介绍指令集架构的意义与组成，再介绍MIPS汇编指令程序设计方法，最后介绍一般的计算机系统的性能评价标准。

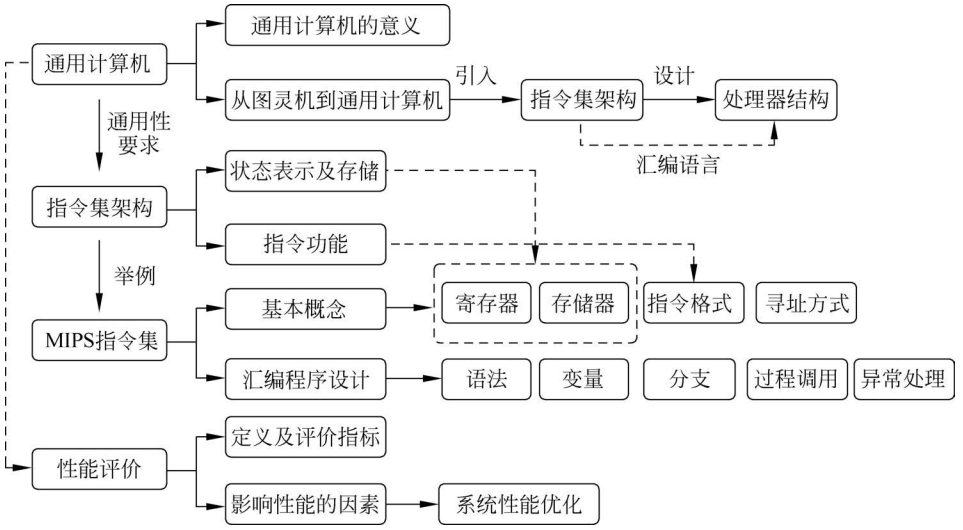

图5-1　本章思维导图

5.1　通用计算机与指令集

5.1.1　通用计算机的意义

计算问题的数学模型——$y=f(x)$以一种统一的形式体现了任何计算任务均可以抽象为输入数据到输出数据的转换，不同的计算任务仅仅是函数$f(\cdot)$的形式有所差别。在本书前面的内容中，针对一个特定任务会采取这样的工作流程设计电路以完成任务：一、分析任务并分解成子任务，二、设计不同的电路模块，三、测试每个电路模块并整合。但是这样

设计的电路对于不同任务的适应性比较差，换一种说法就是“高特定任务性能，低通用性”，即**专用集成电路**(application specific integrated circuits，ASIC)。有些情况下不希望每有一个新的需求就设计一套新的电路系统，而是希望拥有**一台通用机器可以完成各种各样的工作**。即使这台机器在每一个特定任务上可能比针对设计的 ASIC 性能差一些，但是却可以降低部署成本和任务切换的成本。图 5-2 以计算 $A+B$ 为例，直观说明了专用集成电路与通用集成电路的区别。

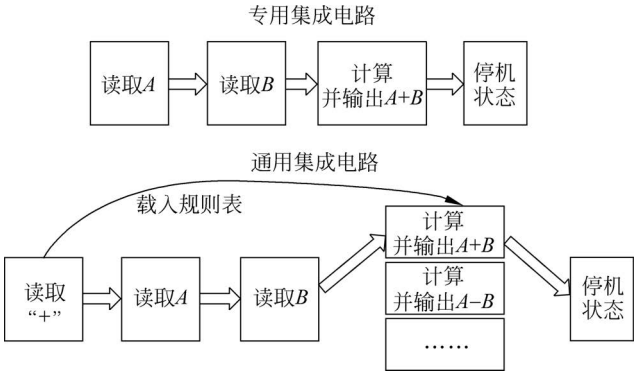

图 5-2　专用集成电路和通用集成电路对比

针对这种通用性的需求人们发明了通用计算机。通用计算机(general purpose computer)简称计算机(computer)，是一种可以根据指令序列自动完成一系列算术或逻辑操作的机器。联系前面介绍的内容，其中“算术或逻辑操作”是指以布尔代数为基础的逻辑运算和算术操作，“自动完成一系列算术或逻辑操作的机器”是指以数字逻辑为基础的电路系统，而**“指令序列”**就是让通用计算机与一般的数字电路系统不一样的关键点。这里的指令序列是指由各种特定功能的指令(instruction)构成的序列，不同的指令序列可以组成程序(program)。通过在计算机上执行各种程序，可以完成小到简单数学计算题、控制智能家电、查看文档内容，大到超大规模科学仿真、控制飞机火箭、处理海量交易信息等任务，这些内容迥异的任务都可以在同一台机器上完成，唯一不同的只是运行的指令序列①。

利用通用计算机，针对一个特定任务，本书会采用以下工作流程完成任务：一、分析任务并分解成子任务，二、设计不同的子程序，三、测试每个子程序并整合。相对于设计 ASIC，软件程序设计的成本更低，开发周期更短，可以低成本复制并通过互联网等载体传播，还可以通过物联网下载到各个终端结点完成功能升级，这使得各个行业都可以享受数字化带来的好处。值得注意的是，即使通用计算机可以在大部分的应用场景中发挥作用，但是由于性能、成本等约束，仍然存在一些场景需要使用特定的电路设计完成任务。实际上很多应用场景会使用同时包括 ASIC 和通用计算机的异构计算(heterogeneous computing)，达到高效率和通用性的均衡。

5.1.2　从图灵机到通用计算机

图灵机(Turing machine)，由数学家艾伦·麦席森·图灵于 1936 年提出，作为一种抽

① 实际考虑到任务规模、性能需求、成本约束等，使用小机器完成大任务和用大机器完成小任务都是不合适的，这里只是讨论理论上的可行性。

象计算模型去模拟人在数学计算中的行为。图灵机包括：①一条无限长的纸带，纸带被分成一个一个的小格，每个格子可以记录一个来自有限字母表的字母。②一个读写头，可以在纸带上左右移动，读取当前格子上的字母或者改变当前格子上的字母。③一个状态寄存器，用来记录图灵机当前所处的状态，状态的总数量是有限的，并且机器同一时间只能处于一个状态。④一套有限规则表，规则表中记录了若干条用于控制读写头的规则，读写头会根据图灵机当前的状态和当前格子上的符号决定读写头的下一个动作并改变状态寄存器的状态。以上部件构成了一个物理可实现[①]的有限状态机，通过状态机的状态跳转，图灵机可以在有限步骤内完成任意有限规模的可计算任务。

理论上任意一个特定的计算任务，都可以设计一套特定的规则表以及相应的寄存器状态和字母表去解决。而由于该状态机状态数、符号集、跳转规则都是有限的，所以总可以用一个有限长的符号串进行描述。因此可以设计一个特殊图灵机，以描述其他图灵机的符号串为输入，然后模拟其他图灵机的行为。这样一台图灵机可以以统一的状态寄存器和有限规则表完成不同任务并且只需要改变纸带上记录的符号。至此，本书依然在讨论一台没有超出图灵计算能力的机器，但是却在通用性上相对“针对特定任务设计的图灵机”迈出了重要的一步，即**在复用同一套状态寄存器和规则表的前提下，将一个任意图灵机的抽象符号序列作为输入并模拟其行为**。

普林斯顿架构(princeton architecture)是一种明确将描述任务的程序作为一种数据和其他输入数据统一存储在存储器(memory)中，使用中央处理器(CPU)进行数据操作和流程控制，并配上与人交互的输入设备(input device)输出设备(output device)组成的计算系统架构，如图 5-3(a)所示。在图 5-3(b)中展示了一种改进版本——哈佛架构，其中单独设计存储程序指令的存储器以提高运行效率。通过对比可以发现：图灵机中的状态寄存器＋跳转规则表与 CPU 对应，图灵机中纸带与存储器对应，描述其他的图灵机结构的编码与存储器内的指令对应，其他图灵机要解决的任务的输入与存储器内的其他数据对应，而存储单元和中央处理器读写数据的过程与读写头的行为一致。至此，如果还知道**一般性的用于描述其他图灵机的编码方法和各个组成单元的设计方法**，就可以用现有的数字电路技术制造一台与图灵机等价的通用计算系统，而这正是本章和后续章节将详细阐述的内容。

5.1.3　指令集架构——软硬件接口

指令代表指示处理器进行一项操作的指示与命令，体现为处理器的某种电路行为(比如加法运算)。而指令的集合，用于控制处理器运转的规则表，通常被称为指令集架构(instruction set architecture，ISA)。通用图灵机的理论只提供了最基础的理论限制，具体如何设计还需要结合现有技术的条件和限制。

总体来说，指令集架构需要描述硬件需要完成哪些功能，每一个步骤状态如何改变，同时也约束了软件只能使用描述的功能，并用这些功能组合成需要的完整的软件系统。可以说指令集架构约束了硬件所必须完成的功能，也约束了软件所能使用的功能范围。针对某个指令集架构编写的任意软件，需要在所有实现了该指令集架构的硬件上运行得到相同的

① 由于可计算任务要求在有限步骤内结束，磁头一次只能移动一格，所以实际使用的纸带是有限长的。无限长假设旨在说明图灵机可以有效解决任意有限规模的任务，这一点并不破坏物理可实现条件。

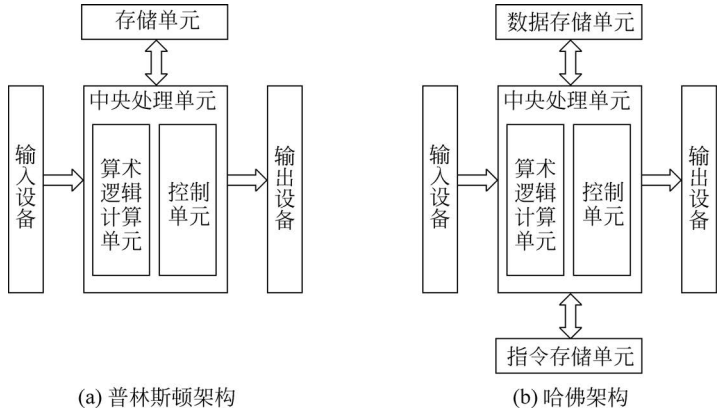

图 5-3　普林斯顿架构与哈佛架构

效果，但对具体如何实现以及所需要的资源没有限制。所以说指令集架构的重要意义在于将软件设计和硬件设计解耦，避免同时考虑两者带来的麻烦，同时也为计算机行业的发展提供了较为稳定的行业标准。如图 5-4 所示，指令集是计算机系统中硬件与软件的纽带。

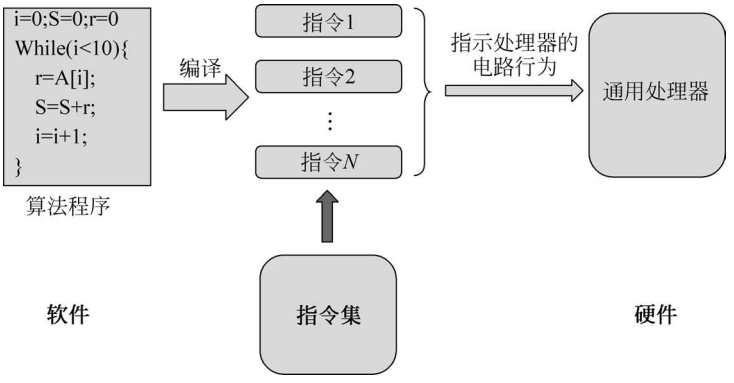

图 5-4　指令集是计算机系统中硬件与软件的纽带

同时也要看到，指令集架构的设计并不是一成不变的。各种具有不同特点的指令集被设计出来，并且随着时代的发展不断改进。指令集架构分为两类，复杂指令集（CISC）和精简指令集（RISC）。CISC（例如 x86）的特点是计算机的指令系统比较丰富，有专用指令来完成特定的功能。因此，处理特殊任务效率较高。而 RISC（例如 MIPS、ARM、RISC-V）设计者把主要精力放在那些经常使用的指令上，尽量使它们具有简单高效的特色。对不常用的功能，常通过组合指令来完成。因此，在 RISC 机器上实现特殊功能时，效率可能较低。但可以利用流水线技术和超标量技术加以改进和弥补，本书第 7 章会详细讨论这部分内容。

为了清楚地比较两种架构的特点和区别，采用两个数的相加运算来说明。如图 5-5 所示，假设待运算的两个数字分别存储在存储器地址 1 和地址 2 。计算单元只能对寄存器 A、B 进行操作。任务定义为将地址 1 和地址 2 两个位置的数字相加，然后存储在地址 1。

CISC 体系结构采用的方法是设计一条专门的指令，不妨记为“EXE”。在执行这条指令时，会将存储器中的这两个值分别加载到不同的寄存器中，计算单元会将操作数相加，然后将结果存储进目标存储位置中。因此，两个数的相加操作可以用一条指令完成：

```
EXE address1 address2
```

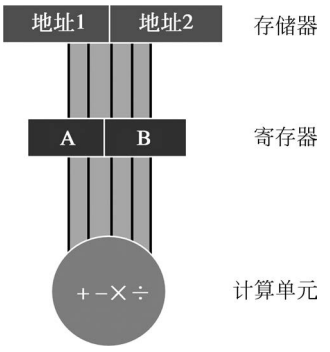

图 5-5　计算机中的加法计算

EXE 就是一条复杂指令，直接对计算机的存储器进行操作，不需要程序员显式地调用任何加载或者存储函数。该体系结构的主要优点之一是，编译器只需做很少的工作就可以将高级语言语句转换为汇编语言。由于代码的长度相对较短，只需要很少的内存来存储指令。实现的关键在于将复杂的指令直接构建到硬件中。

RISC 体系结构采用的方法是只使用可以在一个时钟周期内执行的简单指令。因此，上面描述的“EXE”命令可以分为三个单独的命令：“LOAD”，将数据从存储器移动到寄存器；“ADD”，计算位于寄存器内的两个操作数的加和；“STORE”，将数据从寄存器移动到存储器。为了执行 CISC 方法中描述的一系列步骤，程序员需要编写四行汇编代码：

```
LOAD   A,address1
LOAD   B,address2
ADD    A,B
STORE address1,A
```

这种方式有更多的代码行，需要更多的内存来存储汇编级指令。编译器还必须执行更多的工作，以将高级语言语句转换为这种形式。然而，RISC 也带来了一些非常重要的优势。因为每条指令只需要一个时钟周期来执行，所以整个程序的执行时间与多周期“EXE”命令大致相同。这些简化指令比复杂指令需要更少的晶体管硬件空间，为通用寄存器留下更多的空间。因为所有的指令都在一个时钟内执行，所以流水线是可行的，详情见本书第 7 章。将加载和存储指令分离实际上减少了计算机必须执行的工作量。在执行 CISC 风格的“EXE”命令后，处理器自动擦除寄存器。如果其中一个操作数需要用于另一个计算，处理器必须重新将数据从存储器加载到寄存器中。在 RISC 中，操作数将留在寄存器中，直到另一个值加载到它的位置。

虽然 CISC 和 RISC 都是图灵完备[①]的，甚至只有一条指令的最简指令集[②]也可以是图灵完备的，但是背后电路设计实现的复杂程度和完成同样任务的效率差别却很大。一个一般性的趋势是指令集所描述的功能越全面、越复杂，完成各种任务的效率越高，但是电路设计也越复杂。反之，指令集描述的功能越少、越简单，所对应的电路设计也越简单，但是完成各种任务的效率可能不高。所以在实际应用中需要根据具体应用场景去选择或设计对应的指令集架构，并相应地设计硬件系统和软件系统。

5.2　指令集架构

本节将沿着从图灵机向实际计算机的路径继续介绍指令集架构的相关知识，包括指令集架构如何定义状态表示，如何处理状态转移等知识。为了让读者有一个直观的认识，在介

① 如果一系列操作数据的规则(如指令集、编程语言)可以用来模拟任何图灵机，那么它是图灵完备的。

② 最简指令计算机，又称单一指令计算机。只包含一条指令即可图灵完备，例如“subleq a，b，c：mem[B]＝mem[b]-mem[a]，if (mem[b]<=0)goto c”。

绍每个知识点时，会将指令集架构的内容和读者相对熟悉的 C 语言进行简单对比，帮助读者理解其中的联系与区别。

5.2.1 状态表示及存储

在 C 语言程序中，所有的计算结果都是通过各种数据类型的变量进行存储的。如图 5-6 所示，为了完成一个“若 x 大于 0，则输出 2 倍 x，否则输出 x 的相反数”的计算任务。在这个程序中需要通过一次执行指令并分别计算 x>0，2 * x，－x 的值才能得到正确的结果。并且注意到 C 语言程序执行时，指令的执行顺序是根据控制流程进行跳转或者顺次向下执行，并且每条指令执行后各个变量的值可能发生变化。联系到图灵机模型，这对应着当前图灵机的内部状态加上纸带上的数据和读写头的位置。

```
一个简单的C语言程序的例子
int x = 2;
int temp;
int result;
temp = x>0;
if(temp)
  result = 2*x;
else
  result = -x;
```

图 5-6 一段 C 语言程序

目前主流的以 CPU 为核心的计算系统中用于记录状态的部件一般包括**寄存器**和**存储器**。其中寄存器一般是指存在于 CPU 芯片内部的寄存器部件，关于寄存器的电路实现细节，可以回顾本书 4.6.1 节。而存储器一般是指与 CPU 芯片分开并且需要通过总线等结构进行访问的存储部件，本书第 8、9 章将分别介绍存储器与总线的相关内容。

5.2.2 指令功能

有了程序状态的定义与表示之后，还需要根据一定的流程进行正确的状态跳转才能完成程序的功能。C 语言中通过各种运算指令去改变变量的值或改变程序执行的流程，对应到汇编指令集架构中的概念就是通过一系列的汇编指令修改寄存器堆、存储器（改变变量）和 PC 寄存器（计算流程）的值。一般的计算架构中包括三种类型的指令：计算指令、数据传送指令和流程控制指令。图 5-7 给出了 C 程序中三种类型指令的示例。

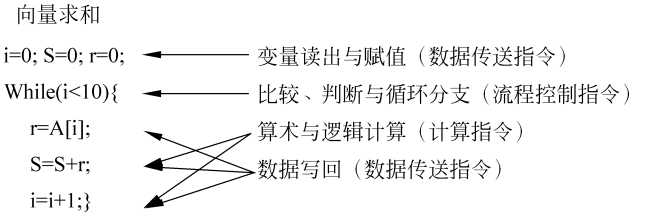

图 5-7 C 语言中三种类型指令的示例

（1）计算指令：其主要功能是根据输入值和一定的规则计算结果。例如整型数据的四则运算和大小比较，浮点数的四则运算和大小比较，数据左移右移，与或非等逻辑运算。

（2）数据传送指令：其主要功能是控制数据的流动，因为目前主流的冯·诺依曼架构中存储器和计算单元是分离的，需要通过一定的指令将数据从存储器读到计算单元内的寄存器中以及将计算单元中的计算结果存回存储器中。

（3）流程控制指令：其主要功能是控制程序的流程。因为在冯·诺依曼架构中程序中

的指令也是按照数据的方式依次排好进行存储的，但是实际执行时需要根据一定的规则跳转执行。这类指令用于支持过程调用和分支判断等功能。

5.3　MIPS 指令集

前面章节介绍了一般的指令集架构包含的状态表示、状态转移、指令编码等内容，以及与有限状态机、图灵机、冯·诺依曼架构等概念是如何联系的。本节将介绍 32 位的 MIPS 架构中的相应概念是如何规定的。MIPS 指令集具有简洁优雅的特点，其设计思想同样适用于 RISC-V、ARM 等其他 RISC 指令集。因此本书采用 32 位 MIPS 指令集作为教学范例。本节一方面让读者了解指令集架构的定义与重要意义，掌握汇编语言及汇编程序的设计方法，另一方面为后续课程中学习 MIPS 架构处理器打下基础。

5.3.1　寄存器

在 MIPS 指令集架构中，直接参与计算的数据是存储在寄存器堆中的，寄存器堆通常具有很低的读取延时和很高的带宽。处理器在进行数据计算时主要从寄存器堆读取数据并将计算结果存回寄存器堆中。

MIPS 指令集架构的寄存器堆通常包括 32 个 32 位通用寄存器。这 32 个寄存器分别记作 $0～$31。这 32 个寄存器都可以被各种指令当作操作数来源，也可以被当作目标寄存器写入数据。但是实际使用时通过制定一定的规范，这些寄存器被分配了不同的功能和使用场合并分别取了别称，从而提高程序的可读性。比如 $0 寄存器别称 $zero 寄存器，这个寄存器的值永远为 0 不会改变；$2，$3 别称 $v0，$v1，被用来存储函数的返回值；等等。别称和对应的功能见表 5-1。后文会详细分析具体如何使用如下寄存器。

表 5-1　寄存器编号、别称及功能

寄存器编号	别　　称	英 文 全 称	功　　能
0	zero	zero	永远存储 0
1	at	assembly temporary	保留用于组装 32 位数
2～3	v0，v1	value	存储子过程返回值
4～7	a0～a3	arguments	调用子过程的参数
8～15 24～25	t0～t7	temporaries	临时寄存器，子过程不需要保存
16～23	s0～s7	saved	保存寄存器，子过程修改前需要保存
26～27	k0，k1	kernel	用于处理中断和异常
28	gp	global pointer	存储全局数据的地址，方便程序读取
29	sp	stack pointer	栈指针，用于记录栈顶的位置
30	s8/fp	frame pointer	8 号保存寄存器，子过程需要时可以用作帧指针
31	ra	return address	子过程的返回地址

一般寄存器堆具有两个读取端口和一个写入端口，输入 5 位寄存器编号便可以读取或写入对应寄存器，后面会详细介绍寄存器堆的硬件结构。

除了寄存器堆中的 32 个通用寄存器外，MIPS 汇编架构中还定义了一个 PC（program

counter)寄存器,用于存储当前正要执行的指令对应的地址。对于一般的指令执行完成后PC←PC+4,对于分支或者跳转指令执行完成后PC会被更新为跳转后的指令对应的地址①。

5.3.2 存储器

寄存器虽然具有读取延时低、带宽大的优点,但是总容量有限,当需要执行的程序有较多变量或者需要分配大量存储空间时,仅仅使用寄存器是不够的。存储器具有较大的存储空间,但是读写数据需要较长的延时,带宽相对寄存器堆也较小。在MIPS32指令集架构中,存储器地址为32位,故存储器可以读写的最大范围为0～($2^{32}-1$)字节。输入32位地址可以读取或写入对应地址的数据。

存储器相对寄存器堆的存储空间大得多,但是延时更大、功耗更高。在调度时应尽量使用寄存器堆参与计算,仅在必要时通过存储器读写数据,这部分会在后面汇编程序结构中介绍。另外使用额外的硬件结构(缓存结构)可以减少读写存储器的平均代价,后面会详细介绍缓存技术。

5.3.3 指令格式

前面根据指令的功能进行分类并介绍了MIPS32指令集中的指令,计算机系统的主要功能是由数字电路组成的,为了执行这些指令需要将指令编码为二进制表示。MIPS32指令集的一个重要特征是所有的指令都被编码为32位的二进制编码,并且分为R型、I型和J型三种指令格式。本节将介绍这三种指令格式。

1. R型指令

一条MIPS中的R型指令按照6+5+5+5+5+6=32位的方式划分为6个字段,如表5-2所示。

表5-2 R型指令说明

位宽	6	5	5	5	5	6
含义	opcode	rs	rt	rd	shamt	funct
作用	操作码	第一个源操作数	第二个源操作数	目标寄存器	位移量	功能码

其中每个字段的具体作用解释如下:

操作码(opcode):用于区分不同的R型指令对应的操作,事实上包括后面提到的I型指令和J型指令都会包含6位的操作码,用于区分不同的指令。6位的操作码最多可以用于区分$2^6=64$种指令,这个数字并不足够,因此还需要与后面的6位的功能码一起确定不同的指令。

源操作数1、2(register source,rs; register target,rt):R型指令的两个操作数均来自寄存器,按照寄存器的编号确定使用哪两个寄存器。因为在MIPS当中一共只有32个寄存器,所以用5位足以编号。

目标寄存器(register destination,rd):与源操作数一样,按照寄存器的编号确定使用哪个寄存器,并用5位进行编号。

位移量(shamt):对寄存器内的数字进行位移,由于寄存器内的操作数不会超过32位,

① MIPS32指令集中,一条指令占4B,所以下一条指令的地址为当前地址+4。

因此用 5 位表示位移量足够。

功能码(funct)：正如前面介绍的，6 位的操作码能够区分的指令数太少，因此需要功能码在同一操作码下区分不同的操作。

【例 5-1】

汇编代码：

add $8，$9，$10

十进制表示：

0	9	10	8	0	32

二进制表示：

000000	01001	01010	01000	00000	100000

可以看到，加法指令的源操作数分别来自 $9 和 $10 两个寄存器，而目标寄存器是 $8。对应的操作码和功能码分别为 0 和 32。由于过程中没有涉及位移操作，因此位移量也是 0。

由于 R 型指令的所有操作数均来自寄存器，并且最终结果也会写回寄存器，因此表现在数据通路上为寄存器堆与算术与逻辑计算单元之间的数据交互。算术与逻辑计算单元从寄存器获取操作数，进行计算后将结果写回寄存器堆，其可能用到的数据通路如图 5-8 所示。

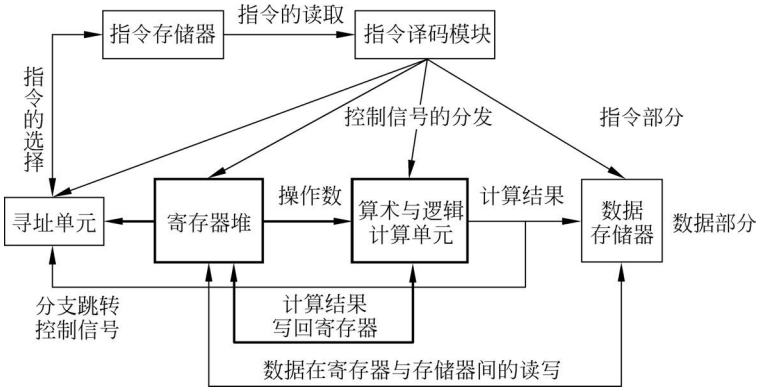

图 5-8　R 型指令数据通路

注意，核心指令集中 R 型指令主要为运算类指令，但是存在一个特例**跳转寄存器 jr** 指令，该指令具有 R 型指令的格式，jr $x 的功能为 PC←R[x]。

2. I 型指令

一条 MIPS 中的 I 型指令按照 6＋5＋5＋16＝32 位的方式划分为 4 个字段，如表 5-3 所示。

表 5-3　I 型指令说明

位宽	6	5	5	16
含义	opcode	rs	rt	Imm
作用	操作码	第一个源操作数	第二个源操作数或者目标寄存器	立即数

其中每个字段的具体作用解释如下：

操作码(opcode)：与 R 型指令的操作码含义相同，I 型指令不需要功能码进行辅助区分。

源操作数(rs，rt)：I 型指令的源操作数可能有一个，也可能有两个，其中第一个源操作数的寄存器编号存储在 rs 中。

目标寄存器(rt)：当 I 型指令没有第二个源操作数时，第二个寄存器编号代表目标寄存器。

立即数(Imm)：16 位数字，根据操作码的区别对应不同的含义，可以是地址偏移量，也可以是某个具体的数字。

MIPS 在设计之初，按照指令格式，将指令划分为 R 型、I 型和 J 型。如此设计指令格式的原则是什么，起到了什么样的作用，会在本节末尾做出详细解释。

【例 5-2】

汇编代码：

lw $s1，100($s2)

十进制表示：

35	18	17	100

二进制表示：

100011	10010	10001	0000000001100100

这是一个数据存入与装载的例子，$s1 是目标寄存器，将以 $s2 内存储数据作为基地址，位移量为 100 的存储器内存储的数据读出并存入 $s1 中。立即数 100 就是地址偏移量。

【例 5-3】

汇编代码：

addi $21，$22，－50

十进制表示：

8	22	21	－50

二进制表示：

001000	10110	10101	1111111111001110

这是一个立即数操作的例子，$22 是源寄存器，$21 是目标寄存器，将 $22 寄存器中的操作数减去 50 后的计算结果存入 $21 寄存器。立即数－50 作为计算源数字。

【例 5-4】

汇编代码：

beq $21，$22，addr

十进制表示：

4	22	21	addr

二进制表示：

000100	10110	10101	addr

这是一个有条件跳转指令的例子。$21 寄存器和 $22 寄存器均作为源寄存器，当两个寄存器中的数值相同时，下一条指令将根据 addr 进行跳转。分支指令采用的寻址方式为 PC 相对寻址——分支目标的地址是 PC+4 与指令中的位移量之和。包括这种寻址方式在内的寻址方式将在 5.3.4 节中做出详细解释。

I 型指令的操作数来自寄存器、存储器以及立即数(指令译码模块输出)，计算结果同样可能写回寄存器、存储器以及寻址单元。I 型指令的数据通路如图 5-9 所示。

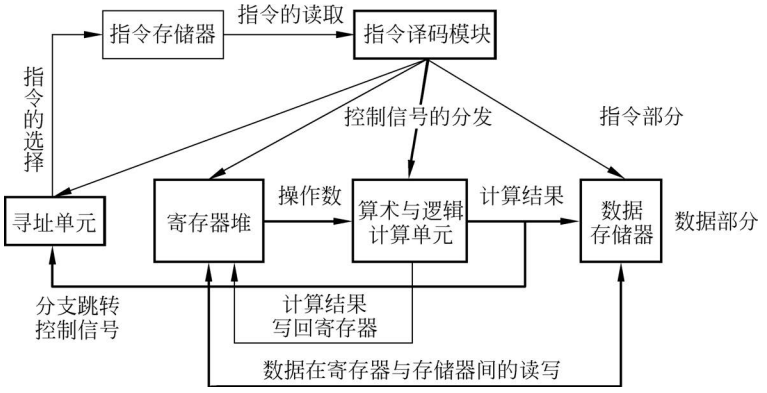

图 5-9　I 型指令数据通路

I 型指令的功能包括运算与数据传送指令、分支指令。大部分 I 型指令中立即数都是进行符号扩展的，即使 sltiu 也是在进行符号扩展后进行无符号比较；也存在例外，andi、ori 两个逻辑立即数运算是对立即数进行无符号扩展的，lui 加载高位立即数指令仅需要扩展低 16 位。

3. J 型指令

一条 MIPS 中的 J 型指令按照 6+26=32 位的方式划分为 2 个字段，如表 5-4 所示。

表 5-4　J 型指令说明

位宽	6	26
含义	opcode	target address
作用	操作码	目标地址

其中每个字段的具体作用解释如下：

操作码(opcode)：与 I 型指令的操作码含义相同，不需要功能码进行辅助区分。

目标地址(target address)：用于标识跳转的目标地址，这里的目标地址只有 26 位，相比较于指令存储器 32 位的地址线还差 6 位，后面会介绍如何用 26 位构造出 32 位的地址。

【例 5-5】

汇编代码：

j 10000

十进制表示：

2	10000

二进制表示：

000010	00000000000010011100010000

在本例中，J 型指令将跳转到 10000 所对应的地址，事实上 10000 所对应的地址并不是地址 10000，而是采用了伪直接寻址，伪直接寻址是在当前指令的一定的范围内进行寻址。

在 J 型指令中，跳转指令采用伪直接寻址——跳转地址由指令中的 26 位常数与 PC 中的高位拼接得到，也就是说：

新的 PC＝{ PC[31:28]，target address，00 }

其他字段都节省出来给跳转的目的地址以表示很大的跳转范围。即便如此，J 型指令也不能在指令存储器中进行任意寻址，后面会提到多种寻址方式。

J 型指令的数据通路如图 5-10 所示。

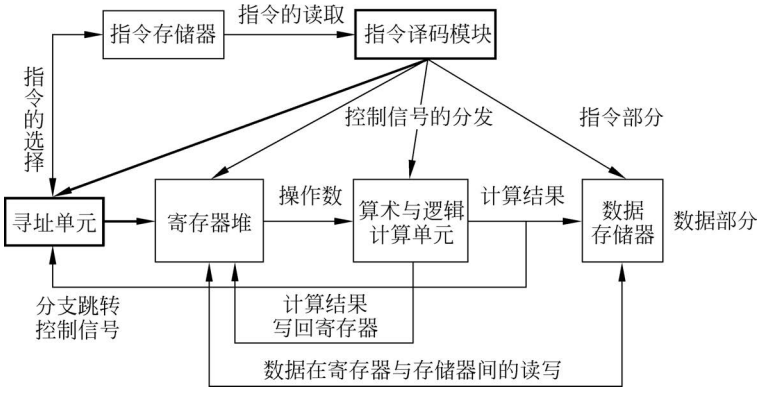

图 5-10　J 型指令数据通路

核心指令集中 J 型指令仅有两条，j 指令和 jal 指令，j 指令仅进行无条件跳转而 jal 指令会在跳转的同时令 R[31]←PC＋4。

学习完 MIPS 的全部指令格式后，可以回过头来思考，设计者为何要将指令划分为不同的指令格式。MIPS 指令集在设计过程中包含着三条重要的设计思想：一、规整性。MIPS 指令集具有所有指令长度统一、寄存器字段在每种指令格式中的位置相同等特点。例如，对应到 R 型指令的设计中，指令长度为固定的 32 位，包含 3 个寄存器操作数，寄存器操作数全部为 5 位。规整性的设计原则使得指令格式变得简单。二、折中设计思想。大量寄存器可能会使得时钟周期变长，因此 MIPS 将寄存器限制为 32 个。但是这条原则不是绝对的，设计者必须在期望更多寄存器和加快时钟周期之间进行权衡。这种思想还体现在不同指令

格式的引入。如果使用R型指令完成取字指令，必须指定两个寄存器和一个常数。这样取字指令的常数就会被限制在2^5(即32)以内。这个常数通常用来从数组或者数据结构中选择元素，通常比32大得多。因此5位字段过小，用处不大。设计者既希望所有指令长度相同，又希望有统一的格式，两者产生冲突。MIPS设计者选择了一种折中方案：保持所有指令长度相同，但是不同类型的指令采用不同的指令格式。R型指令用于处理寄存器，I型指令用于立即数。I型指令16位的地址字段意味着取字指令可以取相对于基址寄存器偏移$\pm 2^{15}$字节范围内的任意数据。虽然多种指令格式引入了复杂的硬件设计，但是保持指令格式的类似性可以降低复杂度。比如，R型与I型指令的前三个字段长度相同，名称一样。I型指令的第四个字段与R型指令的后三个字段之和相等。自然地，设计者可以采用第一个相同长度的字段区分指令格式，不同格式的指令在第一个字段(op)中占用不同的值区间。

5.3.4　寻址方式

前面已经介绍了MIPS指令集架构是如何定义状态表示(寄存器与存储器)和状态跳转(R,I,J三型指令)的。注意，无论是寄存器中的数据，存储器中的一般数据或者存储器中的指令数据，都是依照一定的顺序进行排列的。例如寄存器堆中的32个通用寄存器被编号为0～31，而存储器中的每个字节都被赋予一个32位的二进制地址，两个字节连起成一个半字，4个字节连起成一个字。

为了完成正确的状态跳转并得到正确的状态结果，需要选择其中的某些数据用于计算或进行修改。一般使用地址表示某个特定数据的位置，而通过一定的表示或计算得到地址的过程称为寻址。下面列举并辨析不同的寻址方式和它们的用途。

1. 寄存器寻址

在寄存器寻址(图5-11)中，操作数来源和目标的寻址，根据指令中编码(5位，32个寄存器)从寄存器中读取操作数，并将结果写回寄存器。

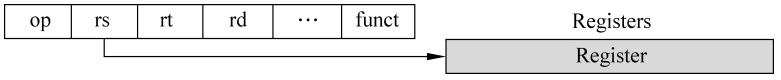

图5-11　寄存器寻址

涉及寄存器寻址的指令种类包括R型指令和I型指令。

2. 立即数寻址

在立即数寻址(图5-12)中，根据指令中的立即数进行寻址。之所以采用立即数寻址，是因为相较于先将立即数存入寄存器(存储器比较慢)，再用寄存器寻址，不如用立即数寻址。

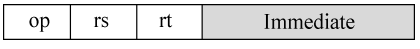

图5-12　立即数寻址

立即数只有16位，怎么把一个32位的常数装入寄存器?

可以将32位的数字拆成两个16位，然后使用两条指令，对高16位和低16位分别进行操作。

【例 5-6】

用立即数寻址的方式，将一个 32 位的数字(高 16 位是 61，低 16 位是 2304)：

0000 0000 0011 1101 0000 1001 0000 0000 装入 32 位的寄存器中。

具体操作为

```
Lui   $s0,61
addi  $s0,$s0,2304
```

第一条指令将 61(高 16 位)装入寄存器的高 16 位中，后一条指令将 2304(低 16 位)装入寄存器的低 16 位中。

涉及立即数寻址的指令种类一般为 I 型指令。

3. 基址或偏移寻址

在基址或偏移寻址(图 5-13)中，以寄存器存储数字作为基地址，在存储器中进行偏移量寻址，偏移量就是立即数。常见的基址或偏移寻址就是前述的 sw 和 lw 指令。

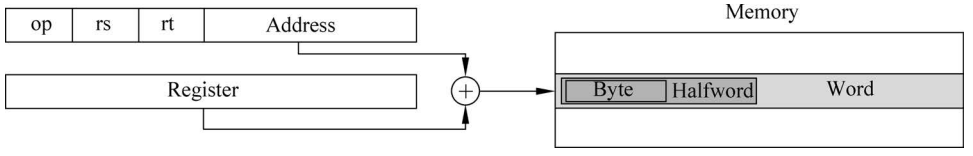

图 5-13　基址或偏移寻址

基址或偏移寻址常用于 I 型指令中的 sw 和 lw 指令中。

4. PC 相对寻址

在 PC 相对寻址(图 5-14)中，根据两个源寄存器的逻辑判断结果(相等、大、小等)，跳转到当前 PC 附近的指令，跳转偏移量为 I 型指令的立即数。

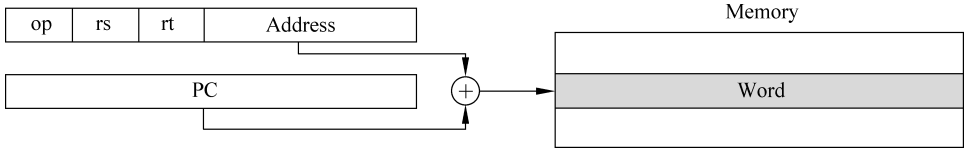

图 5-14　PC 相对寻址

立即数长度为 16 位，是有符号数，考虑到地址都是 4 的倍数，因此跳转范围为 -2^{15}～$2^{15}-1$ 个字(不是字节!)。

5. 伪直接寻址

伪直接寻址(图 5-15)的方式在 5.3.3 节介绍 J 型指令时已经介绍过了，32 位的地址由 26 位和其他数据拼接而成，拼接方式为

新的 PC = { PC[31:28], target address, 00 }

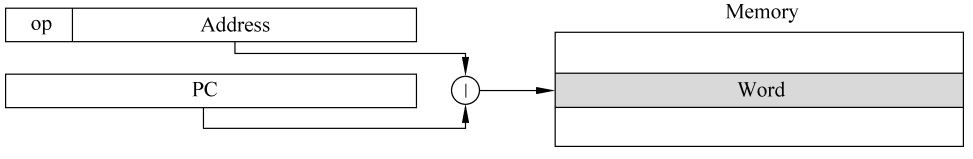

图 5-15　伪直接寻址

涉及伪直接寻址的指令种类只有 J 型指令。

5.4　汇编程序设计

前面介绍了 MIPS32 指令集是如何表示状态并实现状态跳转的，理论上已经可以进行各种计算了。但是从工程实践的角度来看，仅有指令集架构是远远不够的，还需要依照汇编程序编程规范进行编程，这样得到的程序更高效，鲁棒性强，可读性强，更容易维护。这些编程规范是大量的工程师在无数的工程实践过程中总结出来的，用于提高工程师自己的编程效率并降低与他人合作的成本。注意，在现在一般的编程场景中这些规范已经包含在编译工具链中，编程者仅需要遵循所使用的编程语言的规范而无需处理汇编层面的规范。

5.4.1　语法

以一段汇编代码为例：

```
      .data                 #将子数据项,存放到数据段中
Item: .word 1,2             #将 2 个 32 位数值送入地址连续的内存字中
      .text                 #将子串,即指令或字送入用户文件段
      .global main          #必须为全局变量
Main: lw $t0, item          #lw 指令
```

1. 基本的语法规范

(1) 注释行以“# ”开始。

(2) 标识符由字母、下画线(_)、点(.)构成，但不能以数字开头。指令操作码是保留字，不能用作标识符。例如：Item。

(3) 标识符放在行首，后跟冒号(:)。

2. MIPS 汇编语言语句格式

指令与伪指令语句：

```
[Label:] < op> Arg1, [Arg2], [Arg3] [ #comment]
```

例如：

```
AddFunc: add $a1 $a2 $a3     # a1 = a2 + a3
```

汇编命令 (directive) 语句：

```
[Label:] .Directive [arg1], [arg2],  . . . [ #comment]
```

例如：

```
word 0xa3
```

3. 常用汇编命令

汇编命令用来定义数据段、代码段以及为数据分配存储空间。

```
.data  [address]                    #定义数据段,[address]为可选的地址
.text  [address]                    #定义正文段(即代码段),[address]为可选的地址
```

```
.align n                          # 以 2^n 字节边界对齐数据,只能用于数据段
.ascii < string >                 # 在内存中存放字符串
.asciiz < string >                # 在内存中存放 NULL 结束的字符串
.word w1, w2,..., wn              # 在内存中存放 n 个字
.half h1, h2,..., hn              # 在内存中存放 n 个半字
.byte b1, b2,..., bn              # 在内存中存放 n 个字节
```

5.4.2 变量与数组

变量存储在主存储器中,而不是寄存器。通常使用这些变量,会使用 lw 语句将变量加载到寄存器,对寄存器进行操作,最后通过 sw 指令将结果写回主存储器。

使用.word 汇编命令为数组开辟空间。该命令在编译时会静态地开辟 n * 4 字节的空间。调用数组同样是通过 lw 和 sw 完成的,例如:

```
lw $t1,0( $A)                     # t1 = A[0],以 0 为地址偏移量
sw $t1,8( $B)                     # B[2] = t1,以 8 为地址偏移量
```

5.4.3 分支

分支常见于 if-then-else 结构中与循环结构中。例如:if then else; while 循环; do until 循环。Case 语句也可以实现分支,是通过枚举类型索引地址并跳转寄存器实现的。汇编指令中,通常使用 beq,bne,blez, bgez, bltz, bgtz, bnez,beqz 指令实现分支。例如:

```
        .data 0x10000000
        .word -6,0                  # x: -6,  y: 0
        .text
main:
        ori     $s6, $0, 0x1000     # 计算内存中数据存放地址
        sll     $s6, $s6, 16        #  $s6 = x
        addiu   $s5, $s6, 4         #  $s5 = y
        lw      $s0, 0( $s6)
        slt     $s2, $0, $s0        # if 0 < x, $s2 = 1
        beqz    $s2, else           #  $s2 = 0, 跳到 else, $s2 = 1, 跳到 done
        move    $s1, $s0
        j done
else:sub    $s1, $0, $s0
done:  sw    $s1, 0( $s5)
       jr    $ra
```

在上述求绝对值的汇编程序中,使用了 beqz 指令实现分支。

5.4.4 过程调用

将相对独立并需要重复使用的功能封装在一个单独的子过程中,在需要使用时进行过程调用是编程中最常见的编程范式。

一个过程包括入口、过程体和出口。调用过程时需要准备好过程参数(如果有)并跳转到过程入口,在执行完过程体中的代码后从出口离开并回到主调过程的调用点的下一条语句,同时获得该过程的返回值(如果有)。

在MIPS的过程调用遵循如下约定：①通过$a0～$a3四个参数寄存器传递参数；②通过$v0～$v1两个返回值寄存器传递返回值；③通过$ra寄存器保存返回地址(跳转前PC值+4)。

考虑过程嵌套，主调过程将调用后还需要使用的参数寄存器$a0～$a3和临时寄存器$t0～$t9压栈。被调过程将返回地址寄存器$ra和在被调过程中修改了的保存寄存器$s0～$s7压栈。如图5-16嵌套过程调用所示，程序A、B、C进行嵌套过程调用，上层程序调用下层程序时，将自己的变量压栈保存。在程序返回时，对应的变量会出栈。

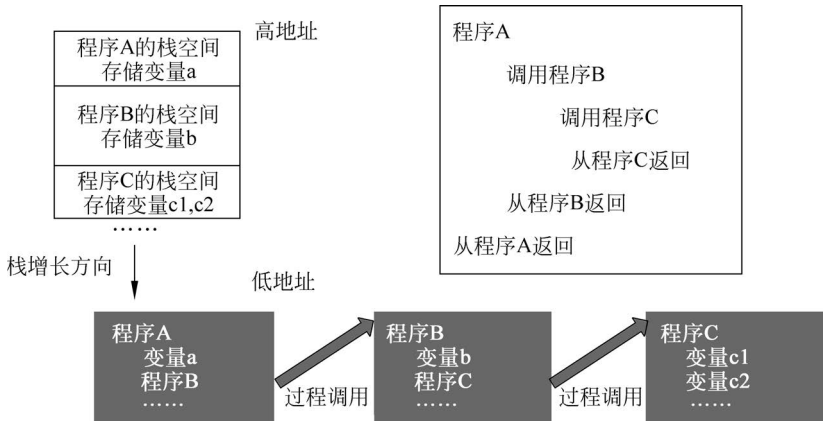

图5-16　嵌套过程调用

【例5-7】

考虑C语言程序段(图5-17)，计算n!，需要存储：(1)每个过程的返回地址；(2)fact(n)中的参数n；(3)在过程调用中临时/局部变量；(4)被破坏的寄存器。

```
int fact(int n)
{
    if(n>0)
    return fact(n-1)*n;
    else
    return 1;
}
```

图5-17　阶乘的C语言实现

以fact(2)为例，图5-18给出了模拟堆栈的变化情况。随着程序的调用，fact(2)、fact(1)、fact(0)的信息按照后进先出的规则进行入栈与出栈。

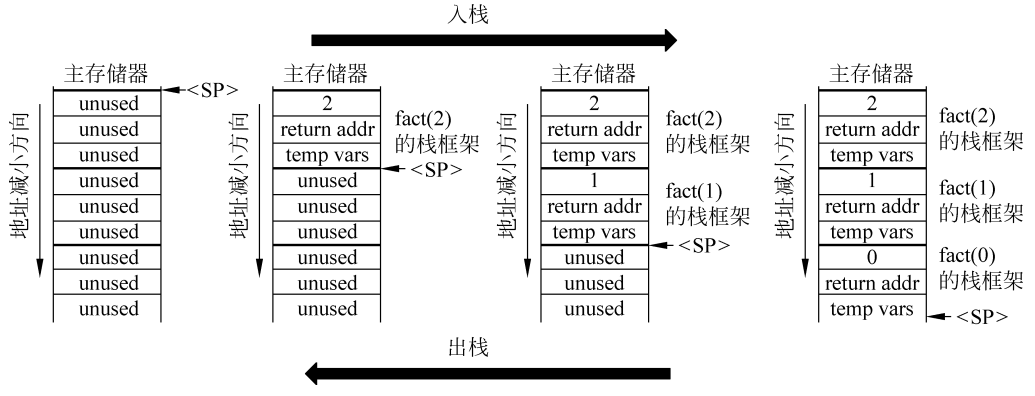

图5-18　fact(2)的入栈和出栈

5.4.5 异常处理

异常是指在程序运行过程中发生的异常事件，通常是由外部问题（如硬件错误、输入错误）所导致的。异常处理的流程主要包括以下步骤：

（1）保护现场，将每个寄存器的值入栈，以便处理完之后回到原来的指令流。

（2）判断是哪种异常类型，执行具体的异常处理函数。

（3）恢复现场，将保存的寄存器的值出栈并写回。

（4）跳转到正常指令流断点，回到 CPU 正常的指令流。

5.4.6 MARS 模拟器

MARS 是 MIPS Assembler and Runtime Simulator（MIPS 汇编器和运行时模拟器）的缩写，能够运行和调试 MIPS 汇编语言程序。MARS 采用 Java 开发，需要 JRE（Java Runtime Environment）执行，可以跨平台。更多的 MARS 相关问题可以参考官网 http://courses.missouristate.edu/KenVollmar/MARS/。

1. 伪指令

MIPS 标准在定义指令集的同时也定义了伪指令，伪指令可以使汇编语言可读性更好，更容易维护。每条伪指令都有对应的 MIPS 指令。汇编器负责将伪指令翻译成正式的 MIPS 指令。表 5-5 给出了 MARS 汇编器中使用到的常见的伪指令及其对应的功能。

表 5-5 伪指令与功能对应表

伪 指 令	功 能
move $t0, $t1	$t0 = $t1
li $t1, 100	将 $t1 设定为 16 位有符号数
la $t1, Label	将 $t1 设定为 label 的地址
abs $t1, $t2	将 $t2 的绝对值存入 $t1
bne $t1, 100000, label	如果 $t1 的值和 32 位立即数不相等，跳转到 label 的位置
ori $t1, $t2, 100000	将 $t1 设定为 $t2 与 32 位立即数的或
xori $t1, $t2, 100000	将 $t1 设定为 $t2 与 32 位立即数的异或

2. 实例

【例 5-8】 用 example_0.asm 作为例子演示 MARS 的用法。example_0.asm 中包含了一个从文件读取数据并写入另一个文件的例子，图 5-19 中给出了代码和注释。

运行 MARS 后的主要界面如图 5-20 所示，主要编辑区用于编写汇编指令。输出信息区可以查看程序运行过程中的输出和系统报错等。寄存器列表实时显示当前运行状态下各个寄存器存储的值。

打开汇编文件 example_0.asm。如图 5-21 所示，单击汇编按钮即可切换到执行页面，源代码汇编成基础指令和机器码，PC 置为 0x00400000，并等待执行。执行页面内可以看到汇编后的基础指令和对应的机器码，以及每条指令的指令地址。

MARS 为代码调试提供了多种功能。如图 5-22 所示，执行：从第一条指令开始连续执行直到结束。单步执行：执行当前指令并跳转到下一条。单步后退：后退到最后一条指令执行

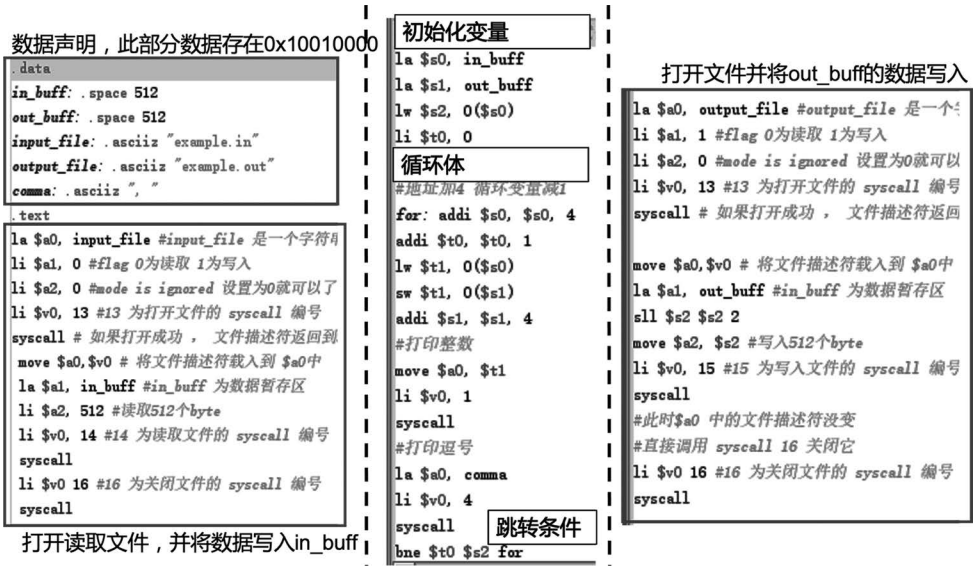

图 5-19　example_0.asm 代码和注释

图 5-20　MARS 功能分区

前的状态(包括寄存器和存储器)。暂停 & 停止：在连续执行时可以停下来，一般配合较慢的指令运行速度，不用于调试。调试通常使用断点功能。重置：重置所有寄存器和存储器。

图 5-21　汇编执行界面

图 5-22　不同的汇编执行方式

单击执行按钮后，所有指令执行完毕。如图 5-23 所示，可以看到各个寄存器内的值发生了变化。memory 中 in_buff、out_buff 地址对应的数据发生变化。输出区正确打印了对应的数据并提示，程序执行完地址最大的指令并且没有后续指令了(drop off bottom)。

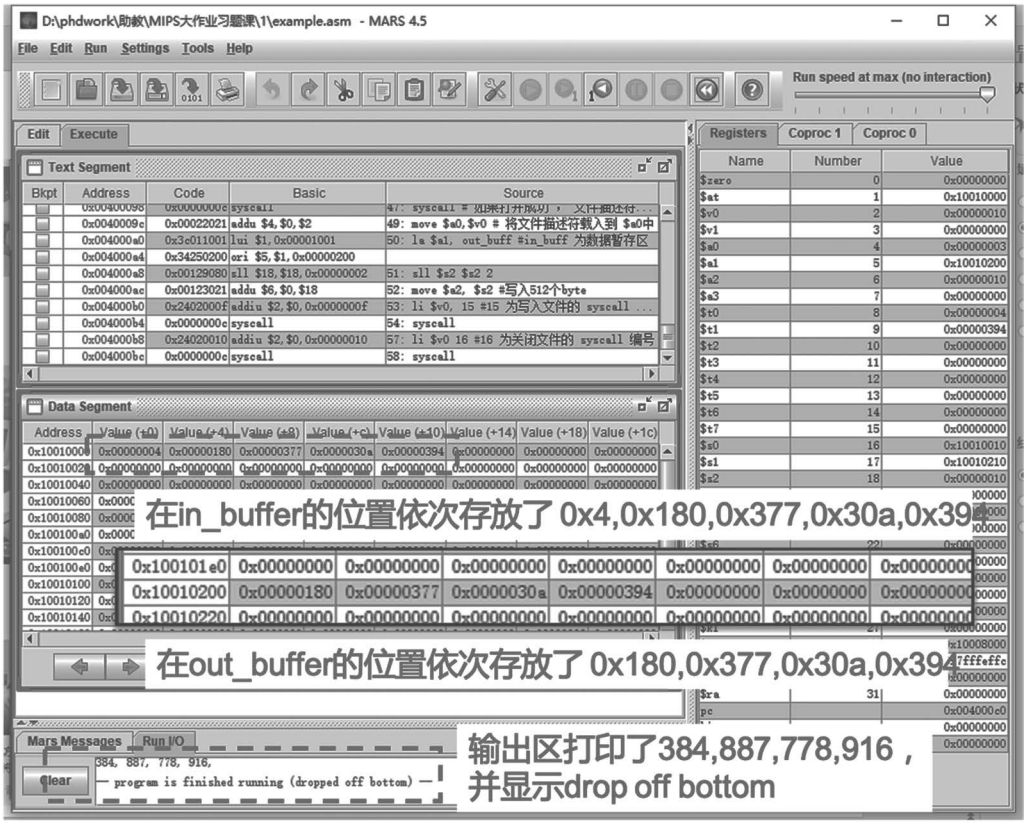

图 5-23　example_0. asm 的执行结果

5.5　性能评价

精确测量和比较不同计算机的性能对于购买者和设计者都至关重要。销售人员也需要了解相关知识，向用户突出展示产品表现最好的一面。对于购买者来说，对于计算机性能的评价标准通常包括成本、处理任务的速度、功耗等多方面。本节主要从“速度”这一角度出发，介绍计算机性能的定义，以及性能评价的不同方法，然后从计算用户和设计者的角度分别描述性能测试的度量标准，最后分析这些度量标准之间的联系。

5.5.1　性能的定义及评价指标

人们在评价一台计算机的性能优劣时，使用最频繁的标准是“这台计算机有多快”，处理任务的速度是评价计算机系统的核心指标之一。“速度”这一概念通常定义为某个量除以时间，选择不同的标准会获得不同的速度指标。以火车和轮船为例，通常情况下汽车行驶同样的距离要比轮船快很多，但是轮船一次能运输的货物是汽车的很多倍。在行驶距离相同的

情况下，如果要运送一些新鲜的食物，人们会选择用汽车运输，防止食物变质。而运输大量钢铁煤炭时，可能更倾向于使用轮船运输，虽然运输到港的时间长，但是单位时间内运输的量更大。

对应到计算机上，如果在两台个人计算机上运行同一个程序，可以说先完成作业的计算机更快。如果运行的是一个数据中心，它有好几台服务器供许多用户同时投放作业，那应该说一天之内完成作业最多的计算机最快。这两个评价标准分别称为响应延时(response time)和吞吐量(throughput)。响应延时表示系统从开始做一项任务到任务完成所需要的总时间，又称为执行时间(execution time)。通常个人计算机和智能手机对于降低响应延时更感兴趣，用户从发起任务到获得结果的时间越短，用户的体验越流畅。吞吐量则表示系统单位时间内处理的任务总数，服务器以及工作站更看重这一点，吞吐量更大的计算系统在面对大量任务请求时，能够更快地完成所有任务(这时大部分的任务都在队列中等待完成，等待的时间远大于任务的响应延时)。面对不同的应用场景，应该选择不同的评价标准去衡量系统的性能。

计算机在实际处理一个任务时，任务的响应时间包括 CPU 运算、磁盘读写、内存读写等时间。在评价一个 CPU 的性能时，应当将除了 CPU 处理任务之外的时间扣除，只考查 CPU 处理程序需要的时间。将一台计算机只处理一项任务时的总时间称为响应时间，对应系统性能，而将 CPU 处理程序的时间称为 CPU 执行时间，对应 CPU 的性能。

对于 CPU 而言，计算应用程序所需要的总时间有一个简单的公式：

$$\text{CPU 执行时间}=\frac{\text{CPU 执行的时钟周期数}}{\text{时钟频率}}=\text{CPU 执行的时钟周期数}\times\text{时钟周期}$$

这个公式表明，设计者减少程序的 CPU 时钟周期数，或者提高时钟频率就能提升性能。但是在实际设计过程中，需要对两者进行权衡。很多提升技术在减少时钟周期数的同时会导致时钟频率的降低。

【例 5-9】 某程序在一台时钟频率为 2GHz 的计算机 A 上运行需要 10s。现在希望将运行这段程序的时间缩短为 6s。设计者拟采用的方式是提高时钟频率，但是这会影响 CPU 其余部分的设计，使得计算机 B 在运行该程序时需要相当于计算机 A 的 1.2 倍时钟周期数。问设计者应该将时钟频率提升到多少？

解：首先要知道在计算机 A 上运行该程序需要多少时钟周期数

$$\text{CPU 时间}_A=\text{CPU 时钟周期数}_A/\text{时钟频率}_A$$

$$10\text{s}=\text{CPU 时钟周期数}_A/2\times10^9(\text{周期数/s})$$

$$\text{CPU 时钟周期数}_A=2\times10^{10}\ \text{周期数}$$

B 的 CPU 时间公式为

$$6\text{s}=1.2\times2\times10^{10}\ \text{时钟周期数}/\text{时钟频率}_B$$

$$\text{时钟频率}_B=4\text{GHz}$$

因此，要在 6s 内完成该程序，B 的时钟频率需要提高为 A 的两倍。

上述性能公式中没有涉及程序所需的指令数。CPU 执行程序时的周期数等于这个程序的所有指令所需要的周期数之和。每条指令所需要的周期数不一定相同，使用指令平均周期数(clock cycles per instruction，CPI)表示平均一条指令所需要的周期数。CPI 可以用

一个程序需要的总周期数除以总指令数计算，或者表示为

$$\mathrm{CPI}=\sum_{i=1}^{N}\mathrm{CPI}_i\times P_i$$

其中，CPI_i 表示第 i 种指令需要的周期数，P_i 表示这种指令出现的频度，通过加权平均得到总 CPI。通常会用性能测试程序的 CPI 进行处理器性能的比较。通过引入 CPI，CPU 的性能公式可以写为

CPU 执行时间＝指令数×CPI×时钟周期

上式表明，计算机的性能应该从三个方面考虑，片面地考虑一个因素往往会得到错误的结果。通过一个例子来说明这一点。

【例 5-10】 有两台计算机 A、B，它们使用了不同的处理器，指令集也不相同。现在有一个程序需要在它们上面运行，这个程序在两种指令集上的指令数、CPI 以及两种处理器的时钟频率如下。问：哪台机器处理该程序的性能更高？

	A	B
指令数	30	25
CPI	2.3	2
时钟频率	2GHz	1.5GHz

解：对于 A，执行任务的时间为 30×2.3÷2GHz＝34.5ns。

对于 B，执行任务的时间为 25×2÷1.5GHz＝33.3ns。

所以 B 的性能比 A 要高，如果单纯地看处理器的时钟频率，会得到错误的结论。考虑系统性能时，不但要考虑硬件的处理速度，还要考虑算法在该硬件对应的指令集上的指令数与 CPI。

5.5.2　影响性能的因素

对于计算机系统而言，影响其性能的因素有很多，例如算法设计、使用的程序设计语言、使用的编译器、使用的指令集架构以及进行运算的处理器架构等。这些因素从不同的方面影响了系统的性能。表 5-6 中给出了不同的因素对系统性能的影响。

表 5-6　影响性能的因素

影响因素	影　响	如何影响
算法	指令数、CPI	算法决定了源程序执行指令的数目，从而决定了 CPU 执行指令的数目。算法通过选用较快或者较慢的指令影响 CPI。例如，当算法使用较多的除法运算时，会导致 CPI 增大
程序设计语言	指令数、CPI	编程语言的语句需要翻译为指令，因此不同的编程语言会影响指令数。编程语言影响 CPI，例如 Java 语言充分支持数据抽象，因此在间接调用时，会使用 CPI 较高的指令
编译器	指令数、CPI	编译器决定了源程序到计算机指令是如何翻译的。不同的编译器会导致翻译后的指令数不同。编译器影响 CPI 的方式比较复杂

续表

影响因素	影　响	如何影响
指令集体系架构	指令数、CPI、时钟周期	指令集体系架构影响完成某功能所需的指令数、每条指令的周期数以及处理器的时钟频率
硬件实现	CPI、时钟周期	处理器时钟周期与硬件的具体实现相关。CPI 的计算公式中包括每个指令所需要的周期，因此也与硬件实现有关

从表 5-6 中可以看到，与指令数有关的因素往往偏上层，也就是靠近算法层。同样完成一个任务，设计优良的算法和注重性能的程序设计语言将会减少可能需要的指令数。同时编译器在编译的过程中也会对程序进行一些优化，减少指令数。指令数还取决于程序被编译到何种指令集上，功能强大的指令集架构会减少所需要的指令数。

CPI 是贯穿整个系统设计的一个重要因素。CPI 的计算公式中包括每个指令需要的周期和每个指令的频度，这说明 CPI 既受到来自软件设计的影响，又与硬件的具体实现有关。

时钟周期更接近于底层硬件层。处理器的时钟周期与硬件的具体实现和生产工艺有关。但是时钟周期是一项与系统功耗紧密联系的参数，无休止地提高时钟频率会使得系统遇到能量供给和散热方面的问题。

【例 5-11】 某 C++程序在桌面处理器上运行耗时 10s，一个新版本的 C++编译程序发行了，其编译产生的指令数量是旧版本编译程序的 0.5 倍，但是 CPI 增加为 1.2 倍。请问该程序在新版本 C++编译程序中运行时间是多少？

解：10s×0.5×1.2＝6s。只减少指令数，CPU 运行时间减少。只增加 CPI，CPU 运行时间增加，可以得出都是相乘的关系。

5.5.3　系统性能的优化

分析了影响性能的各种因素，再来考察有哪些提升系统性能的方法，重新回到 CPU 执行时间的计算公式：

$$\text{CPU 执行时间} = \text{指令数} \times \text{CPI} \times \text{时钟周期}$$

通过优化以上三项的任意一项都可以提升系统的性能，比如通过优化编译器减少指令总数，或者增加 CPU 的复杂性降低 CPI，或者优化 CPU 的关键路径降低时钟周期。但是对其中任意一项的优化，都有可能导致另外两项的提升（同时也有可能增加成本、功耗等参数），以致最终结果提升不大，或者反而性能下降，所以在进行优化设计时，要综合考虑多方面因素以评估最终的性能收益。

有一些技术可以在不影响其他两项的情况下对其中一项进行优化，例如：采用编译器优化技术，使得同样一段高级语言的代码翻译成汇编后的指令数更少，这样纯软件的改动不影响指令集和硬件架构，代价是增加了编译程序的时间。单纯从硬件工艺的角度出发，在不改变硬件实现的逻辑功能的基础上，选择更快的电路实现与生产工艺，可以减少时钟周期。由于处理器的逻辑功能没有变，所以不会影响 CPI 和指令数。代价是增加生产成本和功耗。

另一些技术可能在减少其中某一项的同时，增加另外两项。例如可以让指令集变得更加复杂，这样原本需要几条指令才能完成的一件事仅用一条指令即可完成，这也是 CISC 指令集架构的基本思想。但是这样的改动虽然减少了指令数量，处理器为了正确地处理这些

所有的指令，其硬件实现的复杂程度会上升，不但会使时钟周期更长，还可能导致设计成本的提高。反过来，为了降低 CPI，可以让指令集更加变得更加精简，减少指令的复杂性。这是 RISC 的基本思想。这么做会导致原本需要一条指令即可完成的任务需要多条指令才能完成。

另外，通过设计处理器的体系架构，在时钟周期变化不大的情况下，让原本只能一个周期完成一条指令的系统变为可以一个周期完成多条指令，也可以成倍地增加系统的性能。流水线技术、超标量技术等正是基于这样的思想设计的，第 7 章将详细讲解流水线处理器的设计原理。

无论采取何种优化技术，只有当优化结果中，运行时间的减少作用大于增加作用时，才能获得性能上的收益。如何寻找最佳的优化方案是系统优化的一个重要环节。

5.6　总结

本章首先概括地讲解了通用计算机的概念。希望读者通过对比专用电路和通用机器的差别，掌握硬件思路与软件思路的主要区别，着重理解引入指令集架构的必要性。

计算机指令集架构是通用计算电路的理论基础，本书以 MIPS 指令集作为代表讲解计算机架构的相关知识。MIPS 指令集架构有着指令简单、处理器电路易实现的特点。本章具体讲解了状态表示及存储、指令功能、指令格式以及寻址方式等方面内容，展示了指令集架构的设计思路。同时本章还讲解了 MIPS 汇编指令程序设计方法，希望读者掌握 MIPS 汇编指令的语法、变量与数组的调用方法、分支以及过程调用的编写方法。

最后介绍了一般的计算机系统的性能评价标准。希望读者通过 CPU 执行时间的计算公式，了解影响性能的因素，并且理解不同因素是如何影响计算机系统性能的。

本章只涉及 MIPS 指令集架构的知识，其他常用的指令集架构并不在本课程要求中。为了更全面地了解指令集架构的设计思路，拓展阅读列举了 x86 架构的相关知识，感兴趣的读者可以自行学习。

5.7　拓展阅读

5.7.1　符号扩展与无符号扩展

MIPS32 位指令集架构中，经常会有指令需要将其中的立即数进行符号扩展或者无符号扩展。即，将 n 位立即数扩展为 32 位。两种扩展方法定义如下。

无符号扩展：直接将扩展后的数据，高 $32-n$ 位设为 0。

符号扩展：将扩展后的数据的高 $32-n$ 位设置为立即数的最高位。

例如：

16 位立即数	0x8001	0x1002
符号扩展	0xFFFF8001	0x00001002
无符号扩展	0x00008001	0x00001002

算术运算中，addu、subu 等指令不输出溢出信号，为无符号扩展。addi、subi、addiu、subiu、slti、sltiu 等指令为符号扩展。逻辑运算中，andi、ori、xori 均为无符号扩展。

5.7.2 x86 指令集

MIPS 属于精简指令集，整个体系结构可以简洁地描述出来。而 x86 与之不同，属于复杂指令集。x86 是由一些相互独立的小组开发的，并且被持续改进了超过 35 年。这些改进在原来的指令集基础上增加了新的特性，使得整个指令集变得十分复杂。本节将介绍 80386 的 32 位指令子集，主要内容包括寄存器、寻址模式、整数操作和指令编码。

1. x86 寄存器和数据寻址模式

图 5-24 给出了 80386 使用的寄存器组，E 前缀代表 32 位寄存器。80386 把 16 位寄存器（除了段寄存器）扩展为 32 位，通常称为通用寄存器（general-purpose register，GPR）。80386 只有 8 个通用寄存器，与之对应，MIPS 使用了 4 倍数量的寄存器。

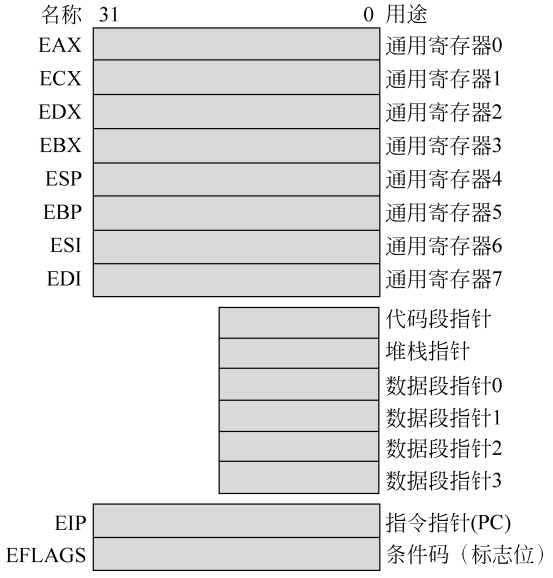

图 5-24 80386 寄存器组

表 5-7 展示了 x86 寻址模式和每个模式下哪个通用寄存器是不允许使用的。同时为了对比，给出了相对应的 MIPS 代码。

表 5-7 x86 寻址模式汇总

模　式	描　述	寄存器限制	等价的 MIPS 代码
寄存器间接寻址	地址存在寄存器	不能为 ESP 或者 EBP	lw $s0，0（$s1）
8 位或 32 位偏移寻址	地址是基址寄存器和偏移量之和	不能为 ESP	lw $s0，100（$s1）# <＝16bit 偏移
基址＋比例下标寻址	地址是基址＋（$2^{比例}$ × 下标），比例是 0、1、2 或者 3	基址：任何 GPR 下标：不能为 ESP	mul $t0，$s2，4 add $t0，$t0，$s1 lw $s0，0（$t0）
8 位或 32 位偏移量的基址＋比例下标寻址	地址是基址＋（$2^{比例}$ × 下标）＋偏移量，比例是 0、1、2 或者 3	基址：任何 GPR 下标：不能为 ESP	mul $t0，$s2，4 add $t0，$t0，$s1 lw $s0，100（$t0）# <＝16bit 偏移

2. x86 整数操作

x86 整数操作主要分为四类。

- 数据传送指令：包括 move、push、pop。
- 算术和逻辑指令：包括测试，整数和小数算术运算。
- 控制指令：包括条件分支、无条件跳转、调用和返回。
- 字符串指令：包括字符串传送和字符串比较。

算术和逻辑操作指令的结果既可以保存在寄存器又可以保存在存储器中。条件分支基于条件码(condition code)或者标志位(flag)。条件码用作结果与 0 的比较，然后使用分支指令测试条件码。PC 相对分支地址必须以字节数来指定，这与 MIPS 是不同的。80386 的指令并不都是 4 字节长。字符串指令在大部分程序中都不使用了，是 8080 的一部分。表 5-8 列出了一些 x86 的整数指令。

表 5-8　x86 整数指令

指　令	含　义
控制指令	**条件和无条件分支**
jnz，iz	条件成立跳转到 EIP+8 位偏移量；JNE(for JNZ)，JE(for JZ)两者之一
jmp	无条件跳转——8 位或 16 位偏移量
call	过程调用——16 位偏移量；返回地址压入栈中
ret	从栈中弹出返回地址并跳转到该地址处
loop	循环分支——递减 ECX；若 ECX 为 0，则跳转到 EIP+8 位偏移处
数据传输	**在两个寄存器之间或寄存器和存储器之间传递数据**
move	在两个寄存器之间或寄存器和存储器之间传递数据
push，pop	将源操作数压栈；将栈顶数据取到寄存器中
les	从存储器中取 ES 和一个 GPR
算术、逻辑	**使用数据寄存器和存储器的算术和逻辑操作**
add，sub	将源操作数与目的操作数相加；从目的操作数中减去源操作数；寄存器-存储器格式
cmp	比较源和目的操作数；寄存器-存储器格式
shl，shr，rcr	左移；逻辑右移；循环右移并用条件码填充
cbw	将 8 位带符号数进行符号位扩展至 16 位
test	将源操作数和目的操作数进行逻辑与，并设置条件码
inc，dec	递增目的操作数，递减目的操作数
or，xor	逻辑或；异或；寄存器-存储器格式
字符串	**在字符串操作数之间移动；由重复前缀给出长度**
movs	通过递增 ESI 和 EDI 从源字符串复制到目的字符串；可能使用重复
lods	从字符串中取字节、字或双字到寄存器

3. x86 指令编码

80386 有多种不同的指令格式。当没有操作数时，80386 的指令可以是 1～15 字节。图 5-25 展示了几条常见指令的格式。操作码字节中通常有一位用来表明操作数是 8 位还是 32 位。一些指令的操作码可能包含寻址模式和寄存器。例如，多数指令形式为“寄存器＝寄存器操作立即数”。其他指令使用寻址模式的“后置字节”或者额外的操作码字节，标记为“mod，reg，r/m”，分别代表模式、寄存器、寄存器/存储器。基址加比例下标的寻址模式

使用第二个后置字节，标记为“sc，index，base”，分别代表比例、下标和基址。

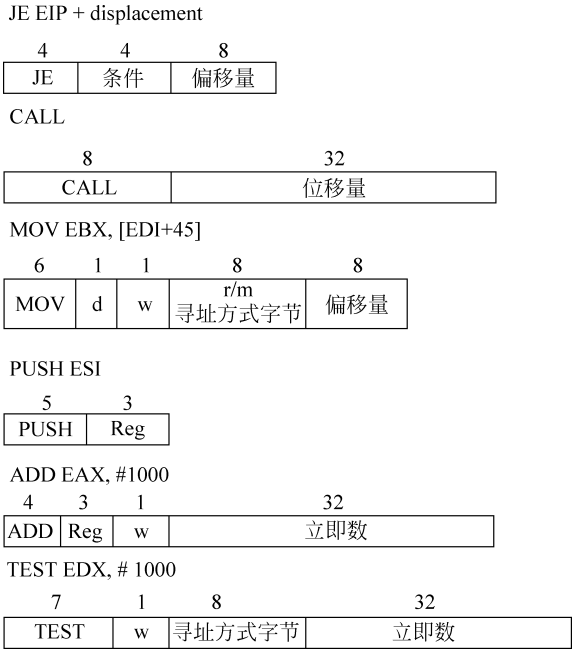

图 5-25　x86 常见指令格式

5.8　思考题

1. CISC 和 RISC 指令集的区别是什么？

2. 通用计算机和专用电路的根本区别是什么？为什么说指令集架构是硬件和软件之间的纽带？

3. 寄存器和存储器的差别是什么？举例说明 MIPS 指令集架构中哪些指令是访问数据存储器的。

4. MIPS 指令集架构中指令按照功能可以分为哪几类？

5. MIPS 指令集架构有哪些指令格式？分类别汇总常用的指令。

6. MIPS 指令集架构的寻址方式分为哪几类？

7. MIPS 分支跳转使用哪些指令可以实现？举例说明。

8. MIPS 过程调用的基本流程是什么？

9. CPU 执行时间如何计算？

10. 影响计算机系统性能的因素有哪些？分别是如何影响的？

5.9　习题

1. 写出下面的 MIPS 字段描述的指令类型、汇编语言指令和二进制表示：

op＝0，rs＝3，rt＝2，rd＝3，shamt＝0，funct＝34

2. 下面的 MIPS 汇编语言程序段对应的 C 语言表达式是什么？

```
add $t0, $a0, $a1
add $t0, $a2, $t0
```

3. 下面 C 语言表达式对应的 MIPS 汇编语言程序段是什么？假设 a、b、c 为 3 个 32 位整型数据，分别保存在寄存器 $a0、$a1 和 $a2 中。

```
a = b + (c - 5);
```

4. 对于 32 位 MIPS 而言，BEQ 指令相对于给定的地址（二进制描述）0x1234A000 的跳转地址范围有多大，具体范围的上下界地址是多少（请注明单位，是字还是字节）？如果想要前往该范围以外的地址，需要进行什么额外操作（说明一种可行方案即可）？

5. 请描述 J 型指令的格式，并说明 J 型指令跳转地址的范围。若 PC 为 0x005FCA90，请计算 J 指令的跳转范围（仅考虑理论范围）。

6. 两个 32 位的变量分别存放在 $t0 和 $t1 中，请给出 MIPS 指令序列交换两个变量的值，要求：不允许使用额外的寄存器。（注意：如果使用加减法，需要考虑溢出）

7. 下列是某个 CISC 指令集中的一条指令，请用 MIPS 指令集实现相同的功能

```
rpt   $t2, loop   # if(R[rs]> 0) R[rs] = R[rs] - 1, PC = PC + 4 + BranchAddr (loop)
```

8. 一些计算机有显式的指令从 32 位寄存器中取出任意字段并放在寄存器的最低有效位中，图 5-26 显示了需要的操作。

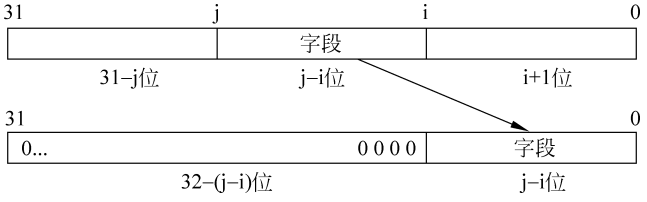

图　5-26

找出最短的 MIPS 指令序列能够在 i＝5 和 j＝22 的情况下从寄存器 $t5 中取出一个字段并放到寄存器 $t0 中。（提示：可以用两条指令实现）

9. 请用尽量少的 MIPS 汇编语句完成 C 语言语句 A[B[0]]＝0。其中数组 A，B 中的元素均为 32 位整数，$s0、$s1 分别存储了数组 A 和 B 的首地址。

10. 假设有如下寄存器内容：

```
$t0 = 0xAAAAAAAA, $t1 = 0x12345678
```

对于以上的寄存器内容，执行下面的指令后 $t2 的值是多少？

```
sll $t2, $t0,4
or $t2, $t2, $t1
```

对于以上的寄存器内容，执行下面的指令后 $t2 的值是多少？

```
sra   $t2, $t0,4
andi $t2, $t2, -1
```

11. 给下面的 MIPS 代码添加注释，并用一句话描述其功能。假设 $a0 和 $a1 用于输入，且在开始时分别包括整数 a 和 b。假设 $v0 用于输出。

```
        add $t0, $zero, $zero
loop:   beq $a1, $zero, finish
        add $t0, $t0, $a0
        sub $a1, $a1, 1
        j loop
finish: addi $t0, $t0, 100
        add $v0, $t0, $zero
```

12. 把下面的 MIPS 代码翻译成 C 代码。假定变量 f、g、h、i 和 j 分别赋值给寄存器 $s0、$s1、$s2、$s3 和 $s4。假定数组 A 和数组 B 的基地址分别存放在 $s6 和 $s7 中。

```
addi $t0, $s6, 8
add $t1, $s6, $0
sw $t1, 0( $t0)
lw $t0, 0( $t0)
add $s0, $t1, $t0
```

对于每条 MIPS 指令，写出操作码(op)、源操作数(rs)和目标操作数(rt)的值。对于 i 型指令，写出立即数字段的值。对于 r 型指令，写出目标寄存器(rd)字段的值。

13. 使用 MIPS 汇编语言编写程序将一个 32 位整型数据转换为对应的 ASCII 码十进制字符串。假设 32 位整型数据存在寄存器 $a0 中，将输出的 ASCII 码字符串保存在以寄存器 $v0 中数据为起始地址的内存中。

14. 通常 C 语言编译器为了方便结构体数据存取，默认设置数据 4 字节对齐。如图 5-27 所示，结构体中数据大小小于 4 字节的数据，也会占用 4 字节的空间。但是有些时候也可以设置不进行数据对齐，通过紧凑排列来节省数据存储的空间。

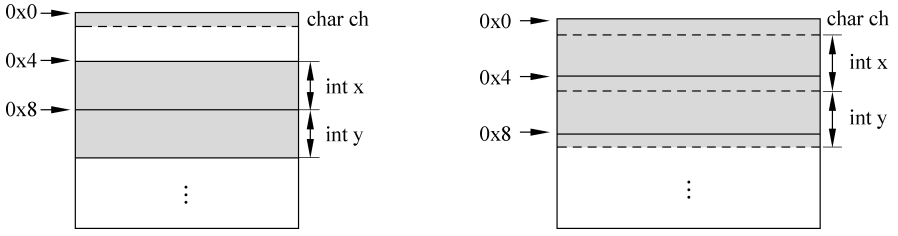

图 5-27

现在假设有如下结构体：

```
struct Foo{
    char ch;
    int x;
    int y;
}foo;
```

结构体变量 foo 的起始地址保存在 $a0 中。($a0 中数据是 4 的倍数，MIPS 规定 lhu/lw 中地址必须为 2/4 的倍数)

(1) 假设结构体采用4字节对齐，写出计算 foo.x+foo.y 的 MIPS 汇编代码，结果保存在 $v0 中。

(2) 假设结构体紧凑排列，写出计算 foo.x+foo.y 的 MIPS 汇编代码，结果保存在 $v0 中。

15. 将下述代码在时钟频率为 2GHz 的机器上运行，各指令要求的周期数如下：

指　令	周　期
add，addi，sll	1
lw，bne	2

$a2，$a3 中的值均为 2500，最坏情况下，将需要多少秒来执行下面这段代码？

```
            sll   $a2, $a2, 2
            sll   $a3, $a3, 2
            add   $v0, $zero, $zero
            add   $t0, $zero, $zero
outer:      add   $t4, $a0, $t0
            lw    $t4, 0( $t4)
            add   $t1, $zero, $zero
inner:      add   $t3, $a1, $t1
            lw    $t3, 0( $t3)
            bne   $t3, $t4, skip
            addi  $v0, $v0, 1
skip:       addi  $t1, $t1, 4
            bne   $t1, $a3, inner
            addi  $t0, $t0, 4
            bne   $t0, $a2, outer
```

16. 有以下一段汇编程序和对应的 C 程序：

地　址	汇编代码	注　释	指令代号
0x00400000	addi $s0 $zero 21	int a=21；	I1
0x00400004	addi $s1 $zero 0	int N=0；	I2
0x00400008	while： slti $t0 $s1 1000	while 开始	I3
0x0040000c	addi $at $zero 1		I4
0x00400010	bne $t0 $at end		I5
0x00400014	andi $t0 $s0 1		I6
0x00400018	slti $t0 $t0 1		I7
0x0040001c	beq $t0 $zero else		I8
0x00400020			I9
0x00400024	j endif		I10
0x00400028	else： add $t0 $s0 $s0		I11
0x0040002c	add $t0 $t0 $s0		I12
0x00400030	addi $s0 $t0 1		I13
0x00400034	endif： slti $t0 $s0 2		I14
0x00400038	bne $t0 $zero end		I15

续表

地　址	汇编代码	注　释	指令代号
0x0040003c	addi $s1 $s1 1		I16
0x00400040	j while	while 结束	I17
0x00400044	end: addi $v0 $s1 0	设置返回值	I18

```
int a = 21;
int N = 0;
while(N < 1000){
    if ((a&1) == 0){
        a = a >> 1;
    }
    else{
        a = a + a + a + 1;
    }
    if (a < 2) break;
    N = N + 1;
}
```

(1) 请根据 C 语言代码写出汇编指令 I9,它的指令格式类型是什么?

(2) I17 是 J 型指令,请写出该指令第 25～0 位(I17[25:0])的值是多少,用十六进制表示;

(3) 请计算该程序执行结束时 I16 指令一共执行了多少次;

(4) 请计算该汇编程序在一个主频为 1GHz 的单周期处理器上执行完成需要多少时间。

17. 假设 $t0 中存放数值 0x00011000,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:
```

18. 考虑如下的 MIPS 循环:

```
LOOP: slt   $t2, $0, $t1
      beq   $t2, $0, DONE
      subi  $t1, $t1, 1
      addi  $s2, $s2, 2
      j LOOP
DONE:
```

(1) 假设寄存器 $t1 的初始值为 20,假设 $t2 初始值为 0,循环完毕寄存器 $t2 的值是多少?

(2) 对于上述循环,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) 假定寄存器 $t1 的初始值为 N,上面的 MIPS 代码执行了多少条指令?

19. 从 CPU 性能或者说指令执行时间的角度考虑，举出两个例子，为什么说 CPU 的硬件设计需要和指令集、编译器，甚至算法之间协同优化设计？

20. 某处理器的算术指令 CPI 为 1，load/store 指令 CPI 为 10，分支指令 CPI 为 3。假设一段程序有 800 万条算术指令，500 万条 load/store 指令和 100 万条分支指令。

（1）计算该处理器运行这段程序的平均 CPI。

（2）假设为该处理器增加更加高效的指令，能够减少 20%的算术指令，但是会使得处理器的时钟频率降低为原来的 90%。从性能角度来说，是否应该增加这些指令？为什么？

（3）若能够加速 load/store 指令至原来的 2 倍，则该处理器执行这段程序总的性能提升多少？加速 load/store 指令至原来的 10 倍呢？