主流 CPU 架构

ARM 架构

简介：

ARM 处理器是英国 Acorn 有限公司设计的低功耗成本的第一款 RISC 微处理器。全称为 Acorn RISC Machine。ARM 处理器本身是 32 位设计，但也配备 16 位指令集，一般来讲比等价 32 位代码节省达 35%，却能保留 32 位系统的所有优势。

体系结构：

体系结构：

（一） CISC（Complex Instruction Set Computer，复杂指令集计算机）

在 CISC 指令集的各种指令中，大约有 20%的指令会被反复使用，占整个程序代码的 80%。而余下的指令却不经常使用，在程序设计中只占 20%。

（二）RISC（Reduced Instruction Set Computer，精简指令集计算机）

RISC 结构优先选取使用频最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻址方式种类减少；以控制逻辑为主，不用或少用微码控制等

RISC 体系结构应具有如下特点：

（1）采用固定长度的指令格式，指令归整、简单、基本寻址方式有 2～3 种。

（2）使用单周期指令，便于流水线操作执行。

（3）大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以访问存储器，以提高指令的执行效率。

（4）所有的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率。

（5）可用加载/存储指令批量传输数据，以提高数据的传输效率。

（6）可在一条数据处理指令中同时完成逻辑处理和移位处理。

（7）在循环处理中使用地址的自动增减来提高运行效率

寄存器结构：

ARM 处理器共有 37 个寄存器，被分为若干个组（BANK），这些寄存器包括：

（1）31 个通用寄存器，包括程序计数器（PC 指针），均为 32 位的寄存器。

（2）6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位，只使用了其中的一部分。

指令结构：

ARM 微处理器的在较新的体系结构中支持两种指令集：

ARM 指令集和 Thumb 指令集。其中，ARM 指令为 32 位的长度，Thumb 指令为 16 位长度。Thumb 指令集为 ARM 指令集的功能子集，但与等价的 ARM 代码相比较，可节省 30%～40%以上的存储空间，同时具备 32 位代码的所有优点。

体系结构扩充：

当前 ARM 体系结构的扩充包括：

（1） Thumb 16 位指令集，为了改善代码密度。

（2） DSP DSP 应用的算术运算指令集。

（3） Jazeller 允许直接执行 Java 字节码。

（4） ARM 处理器系列提供的解决方案有。

（5） 线、消费类电子和图像应用的开放平台。

（6） 存储、自动化、工业和网络应用的嵌入式实时系统。

（7） 智能卡和 SIM 卡的安全应用。

处理器工作模式：

（1） 用户模式(usr)： ARM 处理器正常的程序执行状态

（2） 系统模式(sys)： 运行具有特权的操作系统任务

（3） 快中断模式(fiq) ：支持高速数据传输或通道处理

（4） 管理模式(svc) ：操作系统保护模式

（5） 数据访问终止模式(abt) ：用于虚拟存储器及存储器保护

（6） 中断模式(irq)： 用于通用的中断处理

（7） 未定义指令终止模式(und)： 支持硬件协处理器的软件仿真

注：除用户模式外，其余 6 种模式称为非用户模式或特权模式；用户模式和系统模式之外的 5 种模式称为异常模式。ARM 处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。

X86 架构

简介：

x86 是一个 intel 通用计算机系列的标准编号缩写，也标识一套通用的计算机指令集合，X 与处理器没有任何关系，它是一个对所有*86 系统的简单的通配符定义。

基本特点:

（一） CISC 指令集。

（二） 分别是真实模式、保护模式、系统管理模式以及虚拟 V86 模式。

（三） 可变的指令长度 X86 指令的长度是不定的，而且有几种不同的格式。

（四） 寄存器的贫乏 X86 指令集架构只有 8 个通用寄存器，而且实际只能使用 6 个。

（五） 内存访问 X86 指令可访问内存地址，而现代 RISC CPU 则使用 LOAD/STORE 模式，只有 LOAD 和 STORE 指令才能从内存中读取数据到寄存器，所有其他指令只对寄存器中的操作数计算。

（六） 浮点堆栈 X87 FPU 是目前最慢的 FPU，主要的原因之一就在于 X87 指令使用一个操作数堆栈。

（七） 芯片变大 所有用于提高 X86 CPU 性能的方法，如寄存器重命名、巨大的缓冲器、乱序执行、分支预测、X86 指令转化等等，都使 CPU 的芯片面积变得更大，也限制了工作频率的进一步提高，而额外集成的这些晶体管都只是为了解决 X86 指令的问题。

拓展能力：

X86 结构的电脑采用“桥”的方式与扩展设备（如：硬盘、内存等）进行连接，而且 x86 结构的电脑出现了近 30 年，其配套扩展的设备种类多、价格也比较便宜，所以 x86 结构的电脑能很容易进行性能扩展，如增加内存、硬盘等。

MIPS 架构

简介：

MIPS 架构（英语：MIPS architecture），是一种采取精简指令集（RISC）的处理器架构，1981 年出现，由 MIPS 科技公司开发并授权，广泛被使用在许多电子产品、网络设备、个人娱乐装置与商业装置上。最早的 MIPS 架构是 32 位，最新的版本已经变成 64 位。

基本特点：

MIPS 架构 20 多年前由斯坦福大学开发，是一种简洁、优化、具有高度扩展性的 RISC 架构。它的基本特点是：包含大量的寄存器、指令数和字符、可视的管道延时时隙，这些特性使 MIPS 架构能够提供最高的每平方毫米性能和当今 SoC 设计中最低的能耗。

发展历史：

（一）1981 年，斯坦福大学教授约翰·轩尼诗领导他的团队，实作出第一个 MIPS 架构的处理器。他们原始的想法是通过指令管线化来增加 CPU 运算的速度。

（二）1984 年，约翰·轩尼诗教授离开斯坦福大学，创立 MIPS 科技公司。于 1985 年，设计出 R2000 芯片，1988 年，将其改进为 R3000 芯片。

（三）2002 年，中国科学院计算所开始研发龙芯处理器，采用 MIPS 架构，但未经 MIPS 公司的授权，遭到侵权的控告。

（四）2009 年，中国科学院与 MIPS 公司达成和解，得到正式授权。

应用范围：

MIPS32 和 MIPS64 指令集架构，可以无缝兼容。

特定应用扩展（Application Specific Extension，ASE），可提升特定类型应用的性能，其包括：

（一） 业界标准 MIPS32 和 MIPS64 架构的 MIPS DSP ASE 信号处理扩展，能够提升客户 SoC 的媒体性能。

（二）SmartMIPS ASE，可在智能卡及其它安全数据应用中实现前所未有的安全性。

（三）MIPS16e 代码压缩 ASE，能减少多达 40%的存储器使用量。

（四）MIPS-3D ASE，可在数字娱乐和多媒体产品中实现高性能三维图像处理的一种具成本效益的解决方案。

体系分类：

MIPS32 位架构：

（一） MIPS32®架构刷新了 32 位嵌入式处理器的性能标准。它是 MIPS 科技公司下一代高性能 MIPS-Based™处理器 SoC 发展蓝图的基础，并向上兼容 MIPS64®64 位架构。MIPS 架构拥有强大的指令集、从 32 位到 64 位的可扩展性、广泛的软件开发工具以及众多 MIPS 科技公司授权厂商的支持，是领先的嵌入式架构。

（二） MIPS32 架构基于一种固定长度的定期编码指令集，并采用导入/存储（load/store）数据模型。经改进，这种架构可支持高级语言的优化执行。其算术和逻辑运算采用三个操作数的形式，允许编译器优化复杂的表达式。此外，它还带有 32 个通用寄存器，让编译器能够通过保持对寄存器内数据的频繁存取进一步优化代码的生成性能。

（三） MIPS32 架构从流行的 R4000/R5000 类 64 位处理器衍生出特权模式异常处理和存储器管理功能。它采用一组寄存器来反映缓存器、MMU、TLB 及各个内核中实现的其它特权功能的配置。通过对特权模式和存储器管理进行标准化，并经由配置寄存器提供信息，MIPS32 架构能够使实时操作系统、其它开发工具和应用代码同时被执行，并在 MIPS32 和 MIPS64 处理器系列的各个产品之间复用。

（四） 指令和数据缓存器的大小可以从 256byte 到 4Mbyte。数据缓存可采用回写或直写策略。无缓存也是可选配置。存储器管理机制可以采用 TLB 或块地址转换（BAT）策略。利用 TLB，MIPS32 架构可满足 Windows CE 和 Linux 的存储器管理要求。

（五） 增加了密集型数据处理、数据流和断言操作（predicated operations）。

MIPS64 位架构：

（一） MIPS64®架构刷新了 64 位 MIPS-Based™嵌入式处理器的性能标准。它代表着下一代高性能 MIPS®处理器的基础，并兼容 MIPS32®32 位架构。

（二） MIPS64 架构是以前的 MIPS IV™ 和 MIPS V™指令集架构（ISA）的扩展集，整合了专门用于嵌入式应用的功能强大的新指令，通过整合强大的新功能、标准化特权模式

指令、支持前代 ISA，以及提供从 MIPS32 架构升级的路径，MIPS64 架构为未来基于 MIPS 处理器的开发提供了一个坚实的高性能基础。

（三） MIPS64 架构基于一种固定长度的定期编码指令集，并采用导入/存储（load/store）数据模型，可支持高级语言的优化执行。其算术和逻辑运算采用三个操作数的形式，允许编译器优化复杂的表达式。此外，它还带有 32 个通用寄存器，让编译器能够通过保持对寄存器内数据的频繁存取进一步优化代码的生成性能。

（四） 条件数据移动和数据预取指令被标准化，以提高通信及多媒体应用的系统级数据吞吐量。

microMIPS 架构：

（一） microMIPS™是一种在单个统一的指令集架构中集成了 16 位和 32 位优化指令的高性能代码压缩技术。它支持 MIPS32® 和 MIPS64® Release 2 架构，整合了可变长度重新编码 MIPS 指令集和新增的代码量优化 16 位和 32 位指令，可提供高性能和高代码密度。

（二） microMIPS 是一个完整的 ISA，既能单独工作，也能与原有的 MIPS32 兼容指令解码器共同工作，允许程序混合 16 位和 32 位代码，无需模式切换。microMIPS 的程序代码量较小，因此可获得更好的缓存利用率和更小的取指带宽（fetch bandwidth），从而有助于提升性能，降低功耗。

（三） microMIPS 包含所有 MIPS ASE 指令，支持 CorExtend™/UDI 接口。

MIPS 指令集汇编

MIPS32 指令集

MIPS 指令分类详情：

- 空操作 no-op
- 寄存器 / 寄存器传输：用得很广，包括条件传输在内
- 常数加载：作为数值和地址的整型立即数
- 算术 / 逻辑指令
- 整数乘法、除法和求余数、整数乘加
- 加载和存储
- 跳转、子程序调用和分支
- 断点和自陷
- CP0 功能：CPU 控制指令
- 浮点
- 用户态的受限访问：rdhwr 和 synci

数据类型

指令的主要任务就是对操作数进行运算，操作数有不同的类型和长度，MIPS32 提供的基本数据类型如下:

（1）位（b）：长度是 1bit。

（2）字节（Byte）：长度是 8bit。

（3）半字（Half Word）：长度是 16bit。

（4）字（Word）：长度是 32bit。

（5）双字（Double Word）：长度是 64bit。

（6）此外，还有 32 位单精度浮点数、64 位双精度浮点数等。

寄存器名称	约定命名	用途
$0	zero	总是为 0
$1	at	留作汇编器生成一些合成指令
$2、$3	v0、v1	用于存放子程序返回值
$4~$7	a0~a3	调用子程序时使用这 4 个寄存器传输前 4 个非浮点参数
$8~$15	t0~t7	临时寄存器，子程序使用时不用存储和恢复
$16~$23	s0~s7	子程序寄存器变量，改变这些寄存器值的子程序必须存储旧的值并在退出前恢复，对调用程序来说值不变
$24、$25	t8、t9	临时寄存器，子程序使用时不用存储和恢复
$26、$27	$k0、$k1	由异常处理程序使用
$28 或$gp	gp	全局指针
$29 或$sp	sp	堆栈指针
$30 或$fp	s8/ft	子程序可用来做堆栈帧指针
$31	ra	存放子程序返回地址

寄存器

MIPS32 中的寄存器分为两类：通用寄存器（GPR：General Purpose Register）、特殊寄存器。

（一） 通用寄存器

MIPS32 架构定义了 32 个通用寄存器，使用$0、$1……$31 表示，都是 32 位。其中$0 一般用做常量 0。在硬件上没有强制指定寄存器的使用规则，但是在实际使用中，这些寄存器的用法都遵循一系列约定，例如：寄存器$31 一般存放子程序的返回地址。MIPS32 中通用寄存器的约定用法如表 1-1 所示。在本书大部分章节中，测试程序都是直接使用汇编指令编写的，对寄存器的约定用法还不需要十分在意，但是本书的最后一章移植 μC/OS-II 时，因为涉及到 C 语言、汇编混合编程，对寄存器的约定用法就需要十分在意了。

MIPS32 中通用寄存器约定用法

（二） 特殊寄存器

MIPS32 架构中定义的特殊寄存器有三个：

PC（Program Counter 程序计数器）

HI（乘除结果高位寄存器）

LO（乘除结果低位寄存器）

（1） 进行乘法运算时，HI 和 LO 保存乘法运算的结果，其中 HI 存储高 32 位，LO 存储低 32 位。

（2） 进行除法运算时，HI 和 LO 保存除法运算的结果，其中 HI 存储余数，LO 存储商。

字节次序

数据在存储器中是按照字节存放的，处理器也是按照字节访问存储器中的指令或数据，但是如果需要读出一个字，也就是 4 个字节，比如读出的是 mem[n]、mem[n+1]、mem[n+2]、mem[n+3]这四个字节，那么最终交给处理器的有两种结果：

（1） 大端模式（Big-Endian）,也称 MSB(Most Significant Byte)
{mem[n],mem[n+1],mem[n+2],mem[n+3]}

（2） 小端模式（Little-Endian），也称为 LSB（Least Significant Byte）
{mem[n+3],mem[n+2],mem[n+1],mem[n]}

在大端模式下，数据的高位保存在存储器的低地址中，而数据的低位保存在存储器的高地址中

	低地址			高地址
大端模式	12	34	56	78
小端模式	78	56	34	12

指令格式

MIPS32 架构中的所有指令都是 32 位，也就是 32 个 0、1 编码连在一起表示一条指令，有三种指令格式。如图 1-5 所示。其中 op 是指令码、func 是功能码。

MIPS32 架构中的 3 种指令格式

（1） R 类型：具体操作由 op、func 结合指定，rs 和 rt 是源寄存器的编号，rd 是目的寄存器的编号，比如：假设目的寄存器是$3，那么对应的 rd 就是 00011（此处是二进制）。MIPS32 架构中有 32 个通用寄存器，使用 5 位编码就可以全部表示，所以 rs、rt、rd 的宽度都是 5 位。sa 只有在移位指令中使用，用来指定移位位数。

（2） I 类型：具体操作由 op 指定，指令的低 16 位是立即数，运算时要将其扩展至 32 位，然后作为其中一个源操作数参与运算。

（3） J 类型：具体操作由 op 指定，一般是跳转指令，低 26 位是字地址，用于产生跳转的目标地址。

指令集分类

（1）逻辑操作指令

有 8 条指令：and、andi、or、ori、xor、xori、nor、lui，实现逻辑与、或、异或、或非等运算。

（2）移位操作指令

有 6 条指令：sll、sllv、sra、srav、srl、srlv。实现逻辑左移、右移、算术右移等运算。

（3）移动操作指令

有 6 条指令：movn、movz、mfhi、mthi、mflo、mtlo，用于通用寄存器之间的数据移动，以及通用寄存器与 HI、LO 寄存器之间的数据移动。

（4）算术操作指令

有 21 条指令：add、addi、addiu、addu、sub、subu、clo、clz、slt、slti、sltiu、sltu、mul、mult、multu、madd、maddu、msub、msubu、div、divu，实现了加法、减法、比较、乘法、乘累加、除法等运算。

（5）转移指令

有 14 条指令：jr 、jalr 、j 、jal、b、bal、beq、bgez、bgezal、bgtz、blez、bltz、bltzal、bne，其中既有无条件转移，也有条件转移，用于程序转移到另一个地方执行。

（6）加载存储指令

有 14 条指令：lb、lbu、lh、lhu、ll、lw、lwl、lwr、sb、sc、sh、sw、swl、swr，以“l”开始的都是加载指令，以“s”开始的都是存储指令，这些指令用于从存储器中读取数据，或者向存储器中保存数据。

（7）协处理器访问指令

有 2 条指令：mtc0、mfc0，用于读取协处理器 CP0 中某个寄存器的值，或者将数据保存到协处理器 CP0 中的某个寄存器。

（8）异常相关指令

有 14 条指令，其中有 12 条自陷指令，包括：teq、tge、tgeu、tlt、tltu、tne、teqi、tgei、tgeiu、tlti、tltiu、tnei，此外还有系统调用指令 syscall、异常返回指令 eret。

（9）其余指令

有 4 条指令：nop、ssnop、sync、pref，其中 nop 是空指令，ssnop 是一种特殊类型的空指令，sync 指令用于保证加载、存储操作的顺序，pref 指令用于缓存预取。

32 位 MIPS 指令值常用指令格式

注：64 位版本开头以“d”表示，无符号数以“u”结尾，立即数通常以“i”结尾，字节操作以“b”结尾，双字操作以“d”结尾，字操作以“w”结尾

助	指令格式	示例	示例含义	操作及其解释

记符									
Bit #	31……26	25……21	20……16	15……11	10……6	5……0			
R-type	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	add $1,$2,$3	rd <- rs + rt ;其中 rs ＝$2，rt=$3, rd=$1
addu	000000	rs	rt	rd	00000	100000	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt;其中 rs ＝ $2 ， rt=$3, rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt ;其中 rs ＝$2，rt=$3, rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-$3	rd <- rs – rt;其中 rs ＝ $2 ， rt=$3, rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2 & $3	rd <- rs & rt;其中 rs ＝$2，rt=$3,rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2 | $3	rd <- rs | rt ;其中 rs ＝$2，rt=$3,rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2 ^ $3	rd <- rs xor rt ;其中 rs ＝ $2 ， rt=$3, rd=$1(异或)
nor	000000	rs	rt	rd	00000	101010	nor $1,$2,$3	$1=~($2 | $3)	rd <- not(rs | rt);其中 rs ＝ $2,rt=$3, rd=$1(或非）
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ； 其中 rs＝ $2， rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ； 其中 rs＝ $2， rt=$3, rd=$1 (无符号数）
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt ;shamt 存放移位的位数，也就是指令中的立即数，其中 rt=$2,

									rd=$1
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ; (logical) ， 其 中 rt=$2, rd=$1
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	rd <- rt >> shamt ;(arithmetic) 注意符号位保留 其中 rt=$2, rd=$1
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	rd <- rt << rs;其中 rs ＝ $3 ， rt=$2, rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	rd <- rt >> rs;(logical)其中 rs ＝$3，rt=$2, rd=$1
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ;(arithmetic) 注意符号位保留 其中 rs＝$3，rt=$2, rd=$1
jr	000000	rs	00000	00000	00000	001000	jr $31	jr $31	PC <- rs
I-type	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend) immediate 其 中 rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,100	$1=$2+100	rt <- rs + (zero-extend)immediate ;其中 rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ;其中 rt=$1,rs=$2
ori	001101	rs	rt	immediate			ori $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ； 其中 rt=$1,rs=$2
xori	001110	rs	rt	immediate			xori $1,$2,10	$1=$2 ^ 10	rt <- rs xor (zero-extend)immediate ； 其中 rt=$1,rs=$2

lui	001111	rs	rt	immediate	lui $1,100	$1=100*65536	rt <- immediate*65536；将 16 位立即数放到目标寄存器高 16 位，目标寄存器的低 16 位填 0
lw	100011	rs	rt	immediate	lw $1,10($2)	$1=memory[$2+10]	rt<-memory[rs+(sign-extend)immediate]；rt=$1,rs=$2
sw	101011	rs	rt	immediate	sw $1,10($2)	memory[$2+10]=$1	memory[rs + (sign-extend)immediate] <- rt ；rt=$1,rs=$2
beq	000100	rs	rt	immediate	beq $1,$2,10	if($1==$2)go to PC+4+40	if (rs == rt) PC <- PC+4 + (signextend)immediate<<2
bne	000101	rs	rt	immediate	bne $1,$2,10	if($1!=$2)go to PC+4+40	if (rs != rt) PC <- PC+4 + (sign-extend)immediate <<2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if (rs <(sign-extend)immediate) rt=1 else rt=0 ;其中 rs＝$2，rt=$1
sltiu	001011	rs	rt	immediate	sltiu$1, $2,10	if($2<10) $1=1 else $1=0	if (rs <(zero-extend)immediate) rt=1 else rt=0;其中 rs＝$2，rt=$1
J-type	op	address					
j	000010	address			j 10000	goto 10000	PC <- (PC+4)[31..28],address,0,0；address=10000/4
jal	000011	address			jal 10000	$31<-PC+4;goto 10000	$31<-PC+4；PC <- (PC+4)[31..28],address,0,0；address=10000/4

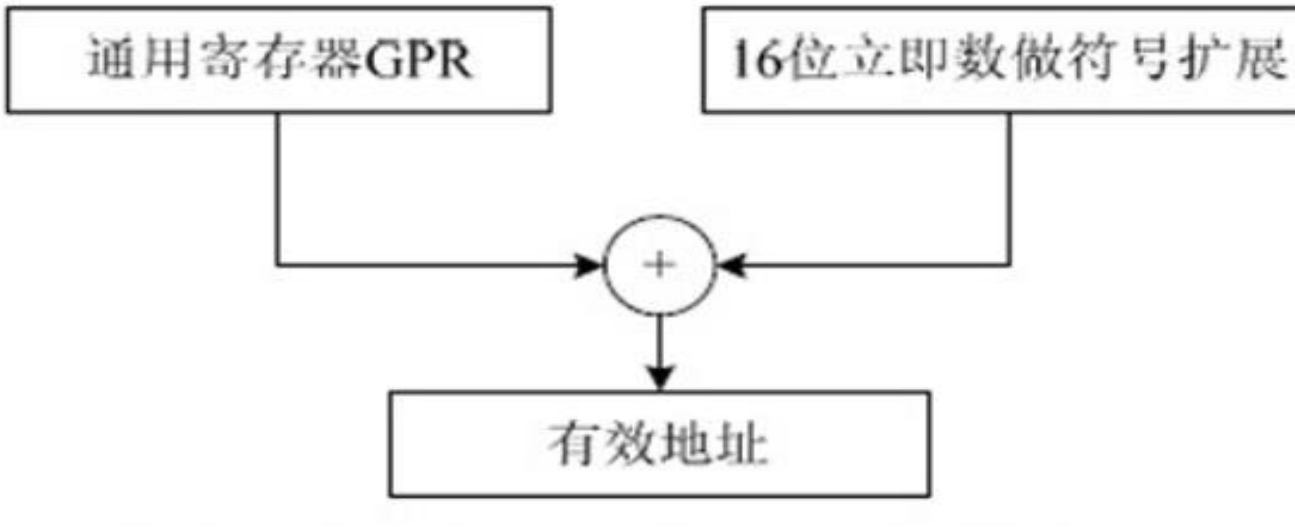
寻址方式

MIPS32 架构的寻址模式有寄存器寻址、立即数寻址、寄存器相对寻址和 PC 相对寻址四种

（1）寄存器相对寻址

这种寻址模式主要是加载/存储指令使用，其将一个 16 位的立即数做符号扩展，然后与指定通用寄存器的值相加，从而得到有效地址

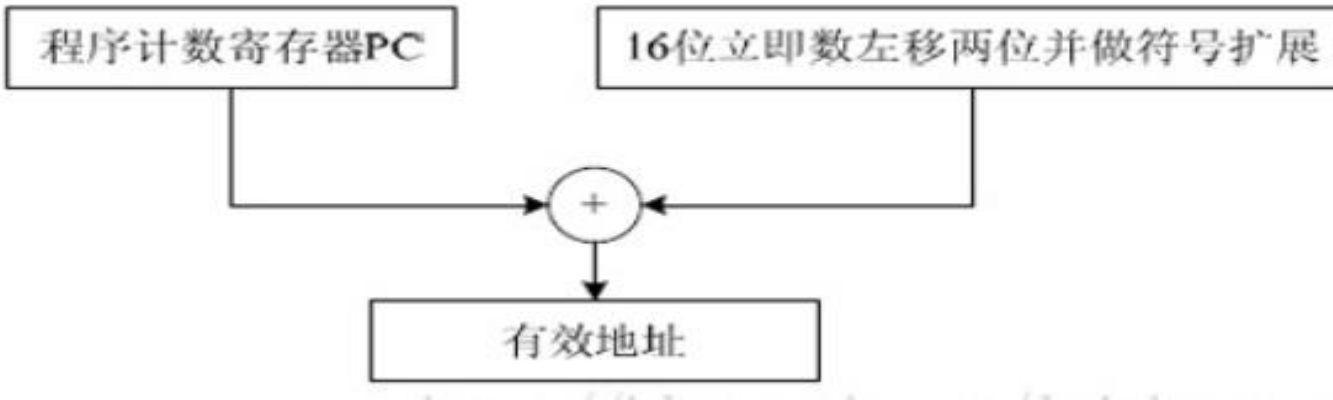
16 位寄存器相对寻址



（2）PC 相对寻址

这种寻址模式主要是转移指令使用，在转移指令中有一个 16 位的立即数，将其左移两位并作符号扩展，然后与程序计数寄存器 PC 的值相加，从而得到有效地址。

PC 相对寻址



协处理器 CP0

协处理器一词通常用来表示处理器的一个可选部件，负责处理指令集的某个扩展，拥有与处理器相独立的寄存器。MIPS32 架构提供了最多 4 个协处理器，分别是 CP0-CP3。协处理器 CP0 用作系统控制，CP1、CP3 用作浮点处理单元，而 CP2 被保留用于特定实现。除 CP0 外的协处理器都是可选的。

协处理器 CP0 的具体作用有：配置 CPU 工作状态、高速缓存控制、异常控制、存储管理单元控制等。CP0 通过配置内部的一系列寄存器来完成上述工作。

异常

在处理器运行过程中，会从存储器中依次取出指令，然后执行，但是有一些事件会打断正常的程序执行流程，这些事件有中断（Interrupt）、陷阱（Trap）、系统调用（System Call）等等，统称为异常。异常发生后，处理器会转移到一个事先定义好的地址，在那个地址有异常处理例程，在其中进行异常处理，这个地址称为异常处理例程入口地址。异常处理完成后，使用异常返回指令 eret，返回到异常发生前的状态继续执行。

MIPS 汇编语言

一．指令格式

[标签] 操作符 [操作数] [#注释]
标签：可选 标记内存地址，必须跟冒号 通常在数据和代码段出现
操作符：定义操作（如 add、sub 等）
操作数：指明操作需要的数据，可以是寄存器、内存变量或常数，大多数指令有 3 个操作数。
示例： L1：addiu $t0,$t1,1

```
#Title:              Filename:
#Author:             Date:
#Description:
#Input:
#OutPut:
################## Data segment ##################
.data
......
################## Code segment ##################
.text
.globl main
main:                                  # main program entry
.......
li $v0,10                              #Exit program
syscall
```

.data 伪指令：定义程序的数据段，程序的变量需要在该伪指令下定义，汇编程序会分配和初始化变量的存储空间。
.text 伪指令：定义程序的代码段。
.globl 伪指令：声明一个全局的符号位，全局符号可以被其他的文件引用，用该伪指令声明一个程序的 main 过程。

二．数据定义

[名字] ：伪指令 初始值[，初始值] ……

三．伪指令

（一）字符串伪指令

- .ASCII 伪指令：为一个 SACLL 字符串分配字节序列
- .ASCIIZ 伪指令：与.ASCLL 伪指令类似，但是在字符串末尾增加 NULL 字符。字符串以 NULL 结尾，类似 C 语言。
- .SPACE n 伪指令：为数据段中 n 个未初始化的字节分配空间。
- 字符串中的特殊字符（按照 C 语言约定）：新行: \n 、Tab: \t、引用：\”。

（二）数据伪指令

- .BYTE 伪指令：以 8 位字节存储数值表。
- .HALF 伪指令：以 16 位（半字长）存储数值表。
- .WORD 伪指令：以 32（一个字长）存储数值表。
- .WORd w:n 伪指令：将 32 位数值 w 存入 n 个边界对齐连续的字中。
- .FLOAT 伪指令：以单精度浮点数存储数值表。
- .DOUBLE 伪指令：以双精度存储数值表.

示例:

```
1   .DATA
2   var1: .BYTE    'A','E', 127 -1 '\n'
3   var2: .HALF    10 ,0xffff
4   var3: .WORD    0x12345678          #如果初始值超过了值域上界,
5   var4: .WORD    0:10
6   var5: .FLOAT   12.3,-0.10
7   var6: .DOUBLE  1.5e-10
8   str1: .ASCII   "A String\n"
9   str2: .ASCIIZ  "NULL Terminated String"
10  str3: .SPACE   100
```

四．跳转（绝对地址）指令

- JAL（lump-and-Link）调用指令：
  jal label #$31=PC+8, jump 在$ra=PC+8 中保存返回地址并跳转到相应的过程中。
- JR（jump Register）返回指令：
  jr Rs #PC=Rs 控制转移到任意地址 跳转到在寄存器 Rs(PC=Rs)中存储的地址所在指令
  j label #只能到达 2^28 个字节的页内指令 目标地址的高 4 位是当前 PC 的高 4 位值
- JALR（Jump-and-Link Register）
  在 Rd=PC+8 中存储返回地址，跳转到寄存器 Rs(PC=Rs)中存储的地址所在过程，地址仅在运行时可知。

五．分支（PC 相对寻址）指令：PC+label

- b label #goto label

- beq s, t, label # if(s==t) goto label
  beql s, t, label # if(s==t) goto label)beq 可能分支变体，仅当分支放生时才执行延迟槽指令。
- beqz s, label # if(s==0) goto label 可能分支变体 beqzl。
- bgez s, label # if(s>=0) goto label
- bgtz s, label # if(s>0) goto label
- bne s, label # if(s!=t) goto label，可能分支变体 bnel。
- bnez s, label # if(s!=0) goto label，可能分支变体 bnezl。
- blez s, label # if(s<=0) goto label)，可能分支变体 blezl。
- bltz s, label #if (s<0) goto label)，可能分支变体 bltl。

六．存储器访问

- lw t, adddr # t=*((int*)addr)，32 位加载，64 位 CPU 上符号扩展。
- lwl t, addr # 向左加载一个字，非对齐加载。
- lwr t, addr # 向右加载一个字。
- lh d, addr # 16 位加载，符号扩展到整个寄存器，d=*((signed short*)addr)。
- lhu d, addr # 16 位加载，零扩展到整个寄存器，d=*((unsigned short*)addr)。
- lbu d, addr # 8 位加载，零扩展到整个寄存器 d=*((unsigned char*)addr)。
- ld d , addr # 8 位加载，符号扩展到整个寄存器 d=*((signed char*)addr)。
- lwc2 cd , addr #32 位加载到协处理器 2 寄存器，不常见。
- lwxc1 fd , t(b) #采用索引（寄存器+寄存器）地址加载 32 位浮点，常写为 l.s，fd=*((float*(t+b))
- li d, j #将常数 j 的值放入寄存器 d 中。
- lui t ,u #(Load Upper Immediat)上位加载立即数（常数 u 符号扩展到 64 位寄存器），t=u<<16。
- ldc1 d, addr # 64 位加载协处理器 1,（浮点）寄存器，常写成 l.d;
- ldc2 d, addr #64 位加载协处理器 2 寄存器，如果采用协处理器 2 并且宽度为 64，那么 ll（Load Linked）和 sc （store conditional）组成原子操作：ll d off(b)……sc t,off(b)。ll 从内存读取一个字， 以实现接下来的 RMW 操作，sc 向内存中写一个字，以完成前面的 RMW 操作。ll d off(b)指令执行后，处理器会记住 ll 操作，sc t off(b)会检查上次 ll 指令执行后的 RMW 操作是否是原子操作（不存在其他对这个地址的从中），若是，t 的值会是被更新到内存中，同时 t 的值变为 1，表示操作成功；若不是，t 的值不会被更新到内存，同时 t 的值变为 0，表示操作失败。
- sw t , addr # *((int*)addr)=t 存储一个字。
- sb t , addr # *((char*)addr)=t。
- sh t ,addr # *((short*)addr =t。
- swc1 ft, addr # 存储单精度浮点数到内存。
- swc2 ft, addr # 存储协处理器 2 寄存器的 32 位数据
- sdc1 ft, addr #存储双精度寄存器到存储器，通常写成 s.d。
- sdc2 ft,addr # 存储 64 位协处理器 2 寄存器到存储器。

七．算术及逻辑指令

- addu d,s,t #d=s+t。
- addiu d,s,j #d=s+(signed)j。

- sub d,s,t　　#d=s-t 溢出时自陷。
- subu d,s,j　#d=s-j 。
- div d,s,t　　#有符号 32 位除法指令，在被零除和溢出的情况下异常。
- divu $zero,s,t #no checks,lo=s/t,hi=s%t。
- mult s,t　# hilo=(signed)s*(signed)t。
- and d,s,t #d=s&t。
- and d,s,j　(andi d,s,j) #d=s&(unsigned)j，仅用于 0<=j<65535,对于更大的数要生成额外的指令。
- maddu　d,s,t　#32 位整数乘法累加，两个寄存器以全精度相乘并累加 hilo=hilo+(long long)s*(long long)t
- or d,s,t　　# d=s|t。
- ori d,s,j　# d=s|(unsigned)j 跟一个常数执行“或”操作 OR
- sllv d,t,s　# d=t<<(s%32) (shift left logic by varriable)
- sll d,s,shf # d=s<<shf;sll d,t,s　d=t<<(s%32)
- sra d,s,shf # d=(signed)s>>shf,shift-right arithmetic,算术右移，最高位填充，社用于有符号数
- srl d,s,shf # d=(unsigned)s>>shf,shift-right logical,逻辑右移，类似 C 的无符号量的移位
- srl d,s,t(srlv d,s,t) # d=((unsigned)s>>(t%32)
- slt d,s,t　# d=((signed)s<<(signed)t)?1:0(set on less than)
- sltu d,s,t # d=((unsigned)s<(unsigned)t)?1:0
- slti d,s,j # d=((signed)s<(signed)j)?1:0
- sltiu d,s,t # d =((unsigned)s<(unsigned)j)?1:0
- break code　#调试器用的断点指令，code 值对硬件没有效果，但是断点异常例程可通过读取异原因指令检索到它的值。
- cache k,addr #对高速缓存进行操作
- ehb　#（执行遇险防护-当你需要保证上面指令的任何协处理器 0 的副作用在随后的指令执行之前已经完成的时候所用的指令）
- ext d,s,shf,sz　#从 32 位寄存器提取位域。Shf 是位域在 S 中的移位到第 0 位所需要的位移量，sz 是位域包含的位的个数。mask=(2**sz-1)<<shf,d=(s&mask)>>shf
- ssnop　#(超标量空操作，是个空操作，但是在同一个时钟周期 CPU 不得发送其他指令，用于达到时序目地)
- move d,s d=s
- movt d,s,$fccN # if($fcc(N))d=s ($fccN 或 fcc(N)是 FCSR 寄存器中的浮点条件位之一)
- movf d,s,t　# if(t)d=s
- movz d,s,t　# if(!t)d=s
- movn d,s,t　# if(t<0)d=s
- negu d,s　　# d=-s,没有溢出
- perf hint,addr;pref hint,t(b) #针对访存进行优化的预处理指令。事先知道可能需要的数据的程序可以进行安排，让所需数据提前进入高速缓存而不产生副作用，具体实现可以把 pref 当成空操作。Hint 定义这是哪种类型的预取。
- syscall B　#exception(SYSCALL,B),产生一个“系统调用”异常。
- teq s,t # if(s==t) exception(TRAP),条件自陷指令，如果相应的条件满足，生成一个 TRAP 异常

- tge s,t # if((signed)s>=(signed)t)exception(TRAP)
- tge s,j #if((signed)s>=(signed)j)exception(TRAP)
- tgeu s,t #if((unsigned)s>=(unsigned)t)exception(TRAP)
- tgeu s,j #if((unsigned)s>=(unsigned)j)exception(TRAP)
- tne s,t #if(t!=s)exception(TRAP)
- tnei s,j #if(s!=j)exception(TRAP)
- tlt s,t #if((signed)s<(signed)t) exception(TRAP)
- tlti s,j #if((signed)s<(signed)j) exception(TRAP)
- tltu s,j #if((unsigned)s<(unsigned)j)exception(TRAP)
- udi9 d,r,s,uc 在为用户自定义指令保留的指令编码空间内构建指令。这种指令可采用三个通用寄存器指令，还可以有用于用户逻辑的 5 位辅助操作码 uc。
- wait #MIPS32 位指令，进入某种断电状态，通常通过中止执行直到检测到中断实现。Wait 指令应该从空转循环中调用。

八．溢出（有符号数）

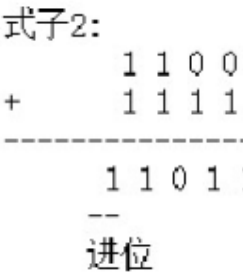
（1）两个正数相加，如果结果为负数，就产生了溢出

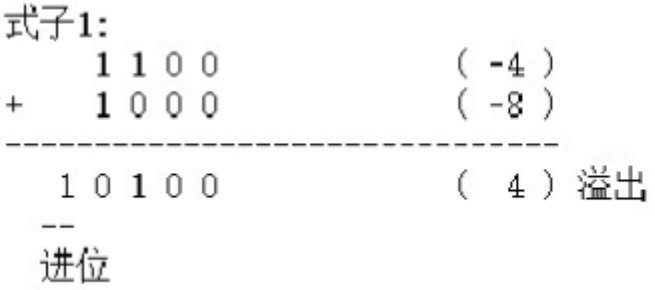
（2）两个负数相加，如果结果为正数，就产生了溢出

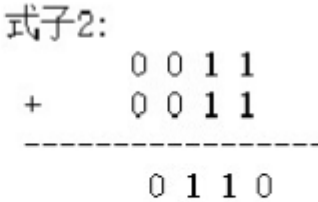
例如：

正数-负数=负数 就产生溢出

负数-正数=正数 就产生溢出



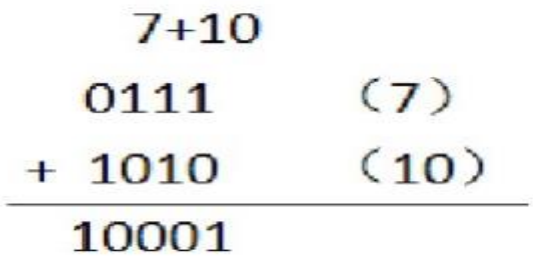


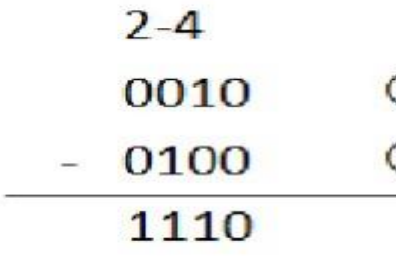


无符号数易出错：

（1） 两个数相加超出了数所能表示的大小，产生进位

（2） 小整数减大整数，会借位





九．MIPS 协处理器（CP0）主要操作指令

- mfhi / mflo rt　# (Move From Hi /Lo) 将 CP0 的 hi /lo 寄存器内容传输到 rt 通用寄存器中。
- mthi /mtlo rt　#将 rt 通用寄存器内容传输到 CP0 的 hi /lo 寄存器中，当 MIPS 体系结构演进到 MIPS IV 的 64 位架构后，新增了两条指令 dmfc0 和 dmtc0,向 CP0 的寄存器中 读/写一个 64bit 的数据。
- mtc0　t, cd　#把 32 位数据从通用寄存器传到协处理器寄存器 cd。mtc0 访问 CPU 控制寄存器，mtcl 把整数单元数据放到浮点寄存器，mtc2 是 CPU 采用协处理器 2 指令时候用。如果协处理器寄存器是 64 位宽度，数据加载进低位，但是高 32 位的状态没有定义。
- mfc0　t,cs　#把 32 位数据从协处理器 cs 传送到通用寄存器 t,如果 cs 是 64 位宽度，则传送的是低 32 位 CPU 控制寄存器。