第3章

处理器体系结构

处理器的体系结构定义了指令集和基于这一体系结构下处理器的程序设计模型。尽管相同体系结构不同型号的处理器性能不同，所面向的应用也不同，但每个处理器的实现都要遵循这一体系结构。在本章接下来的内容中，将以龙芯架构 LoongArch 为主讲解相关的指令集和汇编程序设计知识。

指令集的概念与作用

3.1 指令集体系结构

指令集架构(Instruction Set Architecture，ISA)，又称指令集或指令集体系，是计算机体系结构中与程序设计有关的部分，包含基本数据类型、指令集、寄存器、寻址模式、存储体系、中断、异常处理及外部 I/O。指令集架构包含一系列的 opcode(机器语言中的操作码)，以及由特定处理器执行的基本命令。

常见的指令集架构包括**复杂指令集**(Complex Instruction Set Computing，CISC)、**精简指令集**(Reduced Instruction Set Computing，RISC)、**显式并行指令计算**(Explicitly Parallel Instruction Computing，EPIC)及**超长指令**(Very Long Instruction Word，VLIW)指令集。

指令集的划分

3.1.1 精简指令集与复杂指令集

1. 复杂指令集介绍

早期的计算机部件比较昂贵，主频低，运算速度慢。为了提高运算速度，人们不得不将越来越多的复杂指令加入指令系统中，以提高计算机的处理效率，这就逐步形成复杂指令集计算机体系。

CISC 指令系统的主要特点如下。

(1) 指令系统庞大，指令功能复杂，指令条数一般多于 100 条。

(2) 指令格式多，一般大于 4 种。

(3) 指令寻址方式多，一般大于 4 种。

(4) 指令字长不固定。

(5) 各种指令均可访问内存。

(6) 不同指令使用频率相差很大。

(7) 大多数指令需要多个机器周期才能完成。

（8）指令系统由微程序控制。

2. 精简指令集介绍

通过对 CISC 的测试，科研人员发现 CISC 指令系统中存在指令使用频率相差悬殊的问题，即一些最常用的指令仅占指令总数的 20%，但是这些指令在程序中的使用频率却占到了惊人的 80%；而剩下的复杂指令占到了指令总数的 80%，但是其使用频率却只占到 20%。同时，复杂的指令系统也带了结构的复杂性，增加了设计的时间成本和设计失误的可能性。此外，尽管超大规模集成电路技术即 VLSI 技术已经达到了很高的水平，但想要把所有 CISC 硬件集成在一个芯片上仍是极其困难的。上述种种原因均限制了计算机芯片速度的进一步提升。

针对 CISC 技术存在的一些弊病，1975 年，IBM 公司的 John Cocke 首先提出了精简指令系统的设想。1979 年，美国加州大学伯克利分校由 Patterson 教授领导的研究组，首先提出了精简指令集系统这一术语，并先后研制了 RISC-Ⅰ和 RISC-Ⅱ计算机。1981 年，美国斯坦福大学在 Hennessy 教授领导下的研究小组研制了 MIPS RISC 计算机，强调高效的流水和采用编译的方法进行流水调度，使得 RISC 技术设计风格得到很大补充和发展。

有学者认为，精简指令集计算机的出现可能是存储程序式计算机诞生以来最有意义且最重要的变革。与 CISC 相比，RISC 在指令系统的设计上采取了决然不同的方法。其主要体现在：

（1）指令系统简单，指令条数少。

（2）寻址方式少。

（3）指令格式简单，指令长度固定，操作码字段位置固定。

（4）拥有更多的通用寄存器，寄存器操作较多，减少了对存储器的访问。

3. 精简指令集与复杂指令集的对比

RISC 和 CISC 是目前设计制造微处理器的两种典型技术，虽然它们都试图在体系结构、操作运行、软件硬件、编译时间和运行时间等诸多因素中做出某种平衡，以求达到高效的目的，但采用的方法不同，在很多方面差异很大，两种指令集的主要不同如下。

（1）指令系统：RISC 设计者把主要精力放在那些经常使用的指令上，尽量使它们简单、高效。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，但可以利用流水技术和超标量技术加以改进和弥补。而 CISC 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，处理特殊任务效率相对较高。

（2）存储器操作：RISC 对存储器操作有限制，使控制简单化；而 CISC 的存储器操作指令多，操作直接。

（3）程序：RISC 汇编语言程序一般需要较大的内存空间，实现特殊功能时程序复杂，不易设计；而 CISC 汇编语言程序编程相对简单，科学计算及复杂操作的程序设计相对容易，效率较高。

（4）中断：RISC 在一条指令执行的适当地方可以响应中断，而 CISC 是在一条指令执行结束后响应中断。

（5）CPU：RISC CPU 包含较少的电路单元，因而面积小、功耗低；而 CISC CPU 包含丰

富的电路单元,因而功能强、面积大、功耗大。

(6) 设计周期:RISC 微处理器结构简单,布局紧凑,设计周期短,且易于采用最新技术;CISC 微处理器结构复杂,设计周期长。

(7) 用户使用:RISC 微处理器结构简单,指令规整,性能容易控制,易学易用;CISC 微处理器结构复杂,功能强大,实现特殊功能容易。

(8) 应用范围:基于 RISC 指令集的设备功能强大且功耗较低,常应用在移动设备领域;而基于 CISC 指令集的设备则主要集中在功能强大且功耗较高的计算机市场中。

4. 指令集架构发展历程

1) x86 指令集架构

x86 泛指一系列基于 Intel 8086 且向后兼容的中央处理器指令集架构,是一种采取复杂指令集计算机(CISC)的指令集架构。最早的基于该指令集架构的 8086 处理器于 1978 年由 Intel 推出,为 16 位处理器。该系列较早期的处理器名称以数字来表示,由于以“86”作为结尾,包括 Intel 8086、80186、80286、80386 及 80486,因此其架构被称为“x86”。而随着计算机技术及应用领域的不断发展,出现了 32 位的 x86 指令集架构和 64 位的 x86 指令集架构,其中以 IA-32 和 x86-64 指令集架构最为著名。

2) MIPS 指令集架构

MIPS 是一种采取精简指令集计算机(RISC)的指令集架构,于 1981 年出现,由 MIPS 公司开发并授权,被广泛使用在许多电子产品、网络设备、个人娱乐设备与商业设备上。最早的 MIPS 架构是 32 位,最新的版本已经变成 64 位。其中包括 MIPS Ⅰ、MIPS Ⅱ、MIPS Ⅲ、MIPS Ⅳ,以及 MIPS Ⅴ,它们是 MIPS32/64(分别是 32 位、64 位的实现)发布的五个版本。MIPS32/64 与 MIPS Ⅰ~Ⅴ 的主要区别在于它除了用户态架构外,还定义了特权内核模式系统控制协处理器。2021 年 3 月,MIPS 宣布 MIPS 架构的开发已经结束,因为该公司正在向 RISC-V 过渡。

3) ARM 指令集架构

ARM 指令集是一个基于精简指令集计算机(RISC)的指令集架构。自 1985 年 ARMv1 指令集架构诞生以来,ARM 公司已经推出了第九代 ARM 指令集架构 ARMv9。目前,ARM 处理器已经占据了移动和嵌入式领域芯片的绝大多数市场份额,基于 ARM 指令集的 Cortex-M 与 Cortex-A 两个系列的处理器已分别在低功耗微控制器与手持移动设备领域得了巨大的成功。

4) RISC-V 指令集架构

RISC-V 是一个基于精简指令集计算机(RISC)的开源指令集架构,该项目于 2010 年始于加州大学伯克利分校,但许多贡献者是该大学以外的志愿者和行业工作者。与大多数指令集相比,RISC-V 指令集可以自由地用于任何目的,允许任何人设计、制造和销售 RISC-V 芯片和软件而不必支付给任何公司专利费。虽然这不是第一个开源指令集,但它具有重要意义,因为其设计使其适用于现代计算设备(如仓库规模云计算机、高端移动电话和微小嵌入式系统)。该指令集还具有众多支持的软件,这克服了新指令集通常的弱点。

5) LoongArch 指令集架构

LoongArch 指令集架构,简称 LA,是龙芯中科研发的完全自主的国产指令集架构。该

架构包含架构翻译(Architecture Translate)的指令子集,可在软硬配合下高效率翻译诸如x86-64、ARM 架构、MIPS 架构、RISC-V 架构等指令集架构。其拥有基础指令 337 条、虚拟机扩展 10 条、二进制翻译扩展 176 条、128 位向量扩展 1024 条、256 位向量扩展 1018 条,共计 2565 条原生指令。LoongArch 的发布标志着中国指令集系统架构承载的软件生态走向完全自主。目前,主流开源软件在 LoongArch 上都已经完成移植,龙芯中科将依托 CPU 底层核心技术及龙芯 CPU 核心优势,形成更加完善的生态体系。

3.1.2　二进制翻译

当今最具商业意义的微处理器仍然牢牢地依附于传统指令集架构,其中一些已经有数十年的历史了。尽管这些指令集架构存在公认的缺点,但制造商不愿开发全新的指令集架构,因为他们可能会失去其产品现有软件基础的商业优势。另一方面,软件开发人员发现将代码移植到新的体系结构非常困难和耗时,如果架构不能获得足够的市场份额,他们就像硬件开发人员一样也面临失去大量投资的风险。这两个因素共同阻碍了处理器设计的创新,因此**二进制翻译**(Binary Translation)技术应运而生,它是一种可以直接翻译二进制程序的技术,可以把某种处理器上的二进制程序翻译到另一种处理器上运行。利用二进制翻译技术,可以在不同处理器之间移植二进制程序,扩大了硬件/软件的适用范围,有助于打破处理器与支持软件之间相互制约影响创新的局面。

1. 二进制翻译技术概述

二进制翻译也是一种编译技术,它与传统编译技术的差别在于其编译器的处理对象。二进制翻译处理的对象是二进制机器代码,该二进制机器代码是经过传统编译器编译生成的,经过二进制翻译处理后成为另一种机器的二进制代码,而传统编译器的处理对象是某种高级语言,经过编译器处理后生成某种机器的目标代码。二进制翻译在概念上可以分成三个部分:前端解码器、中端分析优化器和后端翻译器,它们在二进制翻译的不同阶段起着不同的作用,共同完成从源到目标的翻译。

2. 二进制翻译技术分类及比较

基于软件的二进制翻译,可以分为三类:解释执行、静态翻译、动态翻译,这三种技术的介绍与优缺点如下。

(1) 解释执行是利用本地处理器代码对源处理器代码逐条实时解释并执行的过程,该过程中系统不对解释的结果进行保存或缓存,也不进行任何优化。解释器比较容易实现,且可以较容易地与老的体系结构进行兼容,但是效率很低。

(2) 静态翻译指的是在离线状态下,对源代码进行整体翻译,产生本地可执行文件。例如,将某源机器上的二进制可执行程序文件完全翻译成目标机器上的二进制可执行程序文件,然后在目标机器上执行该程序,一次的翻译结果可以多次使用,离线翻译也会提供充足的时间进行优化,产生高质量代码,提高运行时效率。但是静态翻译无法摆脱执行期解释器的支持,也需要终端用户的参与,整个过程缺乏透明,给用户造成了不便。

(3) 动态翻译在程序运行时对执行到的代码片段进行实时翻译,克服了静态翻译的一些缺点,如无法实现代码挖掘、动态信息收集、自修改代码和精确中断等问题。并且动态翻

译器对用户完全透明,不需要用户的干涉。但是动态翻译由于在翻译过程中受到动态执行的限制,不能像静态翻译那样进行细致的优化,使得翻译生成代码的执行效率相较于静态翻译较差。三种二进制翻译技术的优缺点如表 3-1 所示。

表 3-1 三种二进制翻译技术的比较

翻译方法名称	优 点	缺 点
解释执行	容易开发,不需要用户干涉,高度兼容	代码执行效率很差
静态翻译	离线翻译,可以进行更好的优化,代码执行效率较高	依赖解释器、运行环境的支持,需要终端用户的参与,给用户使用造成了不便
动态翻译	无须解释器和运行环境支持,无须用户参与,可利用动态信息发掘优化机会	翻译的代码执行效率不如静态翻译高,对目标机器有额外的空间开销

3. 具有代表性的二进制翻译系统

目前,二进制翻译已经得到了广泛的重视和研究,一些有代表性的二进制翻译系统如表 3-2 所示。

表 3-2 具有代表性的二进制翻译系统

名 称	研究单位	源 平 台	目的平台
FX! 32	Digital	Windows/x86	Windows/Alpha
CodeMorphing	Transmeta	x86	VLIM
Rosetta2	Apple	macOS/x86	macOS/Arm
BOA	IBM	UNIX V/PowerPC	UNIX V/PowerPC
QuickTrans-it	Transmeta	MIPS PowerPC x86	Itanium x86 PowerPC Opteron
Daisy	IBM	UNIX V/PowerPC	VLIW
BOA	IBM	UNIX V/PowerPC	UNIX V/PowerPC
Houdini	Intel	Android/ARM	Windows/x86
LAT	龙芯中科	Linux/MIPS Linux/x86 Windows/x86	Linux/LoongArch Linux/LoongArch Linux/LoongArch

由于静态二进制翻译器的局限性,所有的实用系统都不采用纯静态的翻译,而是选择动态模拟或动态翻译再加上动态优化的方式。这样就可以在保证程序能够正确执行的基础上,尽量提高效率。而且动态翻译对用户透明,无须用户对其过程进行干预。

4. 龙芯二进制翻译技术

龙芯为了兼容 x86 和 ARM 应用程序,对二进制翻译技术进行了多年的研究和实验,并把成果整合在了自主设计的 LoongArch 架构中。

目前，龙芯二进制翻译技术的产出主要分为两大部分，分别是应用级二进制翻译器、跨指令集的系统级虚拟机。在应用级二进制翻译器层面，龙芯推出了 Linux 到 Linux 的翻译器及 Windows 到 Linux 的翻译器。其中，Linux 到 Linux 的翻译器主要包括 LATM（Linux/MIPS 到 Linux/LoongArch）、LATX（Linux/x86 到 Linux/LoongArch），而 Windows 到 Linux 的翻译器主要是由开源系统 Wine 完成，其主要作用是用 Linux 的系统调用来模拟 Windows 的系统调用。在推进自主 CPU 国产化应用的过程中，有大量的打印机无法在 Linux 平台正常工作，只能运行于 Windows 平台下，而龙芯通过二进制翻译技术，使基于 Windows 开发的打印机驱动流畅运行于 LoongArch 平台的 Linux 上。在系统级虚拟机层面，通过二进制翻译技术，龙芯 CPU 上可运行 Linux 和 Windows 2000/XP/7/10 等操作系统，实现了基于 QEMU 的 LoongArch 后端及更高效的系统 LATX/SYS。同时，通过二进制翻译技术，龙芯也完成了对 Windows 打印驱动的支持，使得老旧的打印机可以正常工作运行。

针对二进制翻译目前存在的一些问题，龙芯也给出了自己的技术路线。龙芯从持续完备测试与调试环境和软硬件深度优化两个方面入手，运用面向二进制翻译的随机测试生成技术、高强度的持续集成系统技术来支持龙芯 CPU 去模拟各种 CPU 的行为以应对完备性挑战，运用从指令集开始的协同和不断迭代的方法来应对性能挑战，不断提升自身二进制翻译技术的效率。

3.2　LoongArch 指令系统概述

龙芯架构 LoongArch 是一种精简指令集计算机风格的指令系统架构。龙芯架构具有 RISC 指令架构的典型特征——指令长度固定且编码格式规整。LoongArch 的指令集采用 load/store 架构，且绝大多数指令只有两个源操作数和一个目的操作数。龙芯架构分为 32 位（LA32）和 64 位（LA64）两个版本，LA64 架构支持应用级向下二进制兼容 LA32 架构，即在应用软件范围内，采用 LA32 架构的软件的二进制可执行文件可以直接运行在兼容 LA64 架构的机器上并能够获得相同的运行结果。但对于系统软件来说，能够在 LA32 架构上运行的系统软件（如操作系统内核）的二进制并不一定能在 LA64 架构的机器上获得相同的运行结果。

龙芯架构采用基础部分（Loongson Base）加扩展部分的组织形式。其中扩展部分包括二进制翻译扩展（Loongson Binary Translation，LBT）、虚拟化扩展（Loongson Virtualization，LVZ）、向量扩展（Loongson SIMD Extension，LSX）和高级向量扩展（Loongson Advanced SIMD Extension，LASX）。在这一节中将介绍 LoongArch 指令系统的具体内容。

3.2.1　LoongArch 指令的编码与汇编助记格式


指令格式

1. 指令编码格式

首先龙芯架构采用的是定长编码，其中的所有指令都是以 32 位固定长度编码的，且所有指令的地址都被要求按照 4 字节边界对齐。若访问的指令地址不对齐，就证明该指令错

误,从而因地址错误触发异常。

在龙芯指令集中,指令编码的特点是所有的操作码都是从指令的第 31 比特(最高位)开始从高到低依次摆放,而所有的寄存器操作数域都是从第 0 比特(最低位)开始从低到高依次摆放。在这种情况下,如果指令中存在立即数操作数,立即数域就要位于寄存器域和操作码域之间,而且根据不同的指令类型立即数域会有不同的长度。大体上来说,共包含 9 种典型的指令编码格式,即 3 种只以寄存器为操作对象不含立即数的编码格式 2R、3R、4R,以及 6 种含有立即数的编码格式 2RI8、2RI12、2RI14、2RI16、1RI21、I26,其中,*R 代表指令中有 * 个寄存器操作数,I* 则表示指令中的立即数占用的数位长度。在图 3-1 中列举了这 9 种典型编码格式的具体定义。

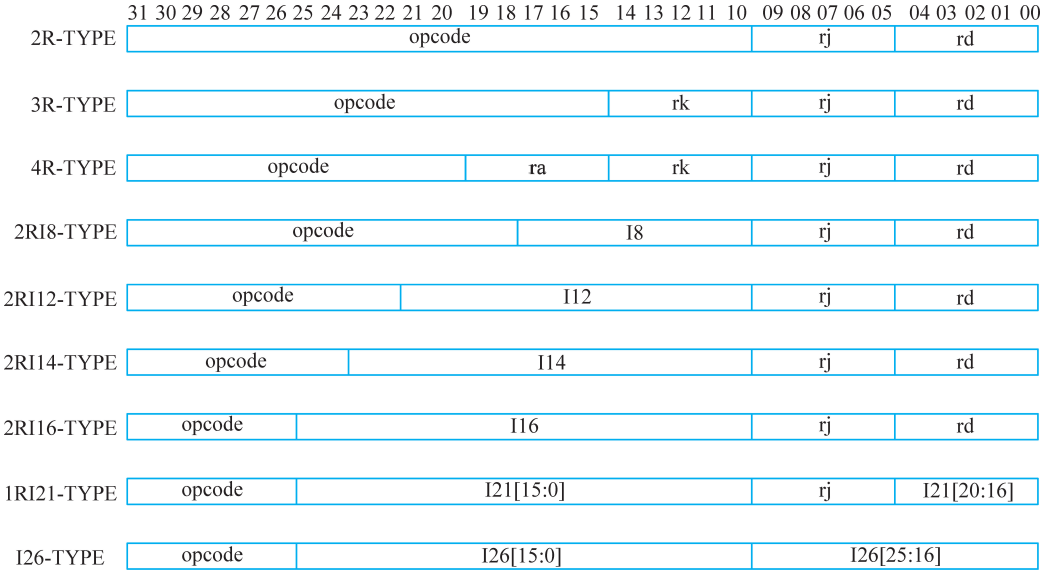

图 3-1 LoongArch 典型指令编码格式

此外,在 LoongArch 的指令系统中还存在少数指令的指令编码域并不完全等同于以上列出的 9 种典型指令编码格式,而是在其基础上存在一些变化,但这种指令的数目并不多且指令编码变化的幅度也不大。

2. 指令汇编助记格式

由于机器指令以二进制的形式呈现,对于人类来说比较难以记忆与使用,便催生了便于人们记忆并能描述指令功能和指令操作数的符号——助记符。助记符一般是表明指令功能的英语单词缩写。在龙芯架构当中,指令的汇编助记格式主要包括指令名和操作数两部分,龙芯架构也对指令名和操作数的前缀、后缀进行了统一考虑与规范来使得汇编编程人员和编译器开发人员可以更方便地使用。下面将对 LoongArch 的指令汇编助记格式进行介绍,但是出于理解难度的考虑,为了帮助初学者更容易地了解指令系统,本书采用的是指令数较少的 32 位 LoongArch 指令集,之后章节中暂不涉及向量指令的具体分析。

对于前缀,龙芯架构通过指令名的前缀字母来区分整数和浮点数指令。例如,指令无前缀默认为非向量整数运算指令,以字母“F”开头的是非向量浮点数指令;以“VF”开头的是

128 位向量浮点指令；以“XVF”开头的是 256 位向量浮点指令；以字母“V”开头的是 128 位向量指令；以字母“XV”开头的是 256 位向量指令。如指令“ADD.W rd,rj,rk”是整数运算指令，而指令“FADD.S fd,fj,fk”则是针对浮点数的操作指令。

在后缀部分，指令系统中的绝大多数指令都通过指令名中“.XX”形式的后缀来表明指令的操作对象类型。例如，对于以整数类型数据为操作对象的指令，指令名的后缀为 B、H、W、BU、HU、WU 时分别表示该指令操作的数据类型为有符号字节、有符号半字、有符号字、无符号字节、无符号半字、无符号字。但是针对操作数为有符号数和无符号数两种情况，无论操作数是哪种类型均不会对运算结果的正确性造成影响时，指令名中携带的后缀均不带 U，但此时并不限制操作对象只能是有符号数，即使后缀中没有无符号标识“U”，操作数也可以是无符号数，但此时操作数是无符号数与否对于指令的执行结果没有影响。而对于操作对象是浮点数类型的，即那些指令名是以“F”“VF”和“XVF”开头的指令，其指令名后缀为 H、S、D、W、L、WU、LU 分别表示该指令操作的数据类型是半精度浮点数、单精度浮点数、双精度浮点数、有符号字、有符号双字、无符号字、无符号双字。此外，在涉及向量操作的指令中，指令名后缀.V 表示该指令是将整个向量数据作为一个整体进行操作。如指令“ADD.W rd,rj,rk”是将 rj＋rk 后得到结果的[31:0]位写入通用寄存器 rd 中，即按字操作。而指令“FADD.S fd,fj,fk”则表明指令设计的三个浮点寄存器中的数据都是单精度浮点数。

还需要指出的是，用“.XX”形式的后缀来指示指令的操作对象的情况并不适用于所有的指令。例如，像 SLT 和 SLTU 这种操作对象的数据位宽由所执行处理器决定的指令是不加后缀的。另外，针对 CSR、TLB 和 Cache 进行操作的特权态指令以及在不同寄存器文件之间移动数据的指令也不会添加后缀来表明操作对象的类型。

此外，在一般情况下，一条指令往往有多个操作数。当源操作数和目的操作数的数据位宽和有无符号情况一致时，指令名只有一个后缀。如果所有源操作数的数据位宽和有无符号情况一致，但是与目的操作数的不一致，那么指令名将有两个后缀依次从左往右排列，第一个后缀表明目的操作数的情况，第二个后缀表明源操作数的情况。而如果不同操作数之间的数据类型区别情况更复杂，那么指令名将有多个后缀从左往右依次列出目的操作数和每个源操作数的情况，其次序与指令助记符中后面操作数的顺序一致。例如，指令“MUL.W.HU rd,rj,rk”中，后缀 W 对应的为目的操作数 rd，HU 对应的为源操作数 rj 和 rk，表示将两个无符号半字相乘，得到的字结果写入 rd 中。

寄存器操作数通过不同的首字母表明其属于哪个寄存器文件。以“r*N*”来标记通用寄存器，以“f*N*”来标记浮点寄存器，以“v*N*”来标记 128 位向量寄存器，以“x*N*”来标记 256 位向量寄存器。其中，*N* 是数字，表示操作的是该寄存器文件中第 *N* 号寄存器。

3.2.2　LoongArch 的寄存器组

由于 LoongArch 采用的是 load/store 架构，除数据存取指令外，所有的指令都直接对寄存器和立即数进行操作，无须内存参与，而寄存器和内存的通信就由数据存取指令来完成。所以架构中会存在较多的寄存器，这也是 RISC 指令集明显区别于 CISC 指令集的一个特点。

在 LoongArch 架构中设置有 32 个通用寄存器(General－purpose Register，GR)，记为 r0～r31，其中，第 0 号寄存器 r0 的值恒为 0。GR 的位宽记作 GRLEN，在本书中介绍的

LA32 架构下 GRLEN 为 32,而在 LA64 中 GRLEN 则是 64。基础整数指令与通用寄存器是正交关系,即指令中的寄存器操作数都可以采用 32 个 GR 中的任意一个。但是除了 r0 寄存器恒为 0 之外,r1 寄存器在标准的龙芯架构应用程序二进制接口中固定作为存放函数调用返回地址的寄存器,且 BL 指令中隐含的目的寄存器也一定是第 1 号寄存器 r1。

程序计数器(Program Counter,PC)则只有一个,PC 负责记录当前取指指令的地址。为了确保程序运行的稳定性与安全性,PC 寄存器中的值不能被指令直接修改,只能通过转移指令、调用陷入和调用返回指令间接修改。但是 PC 可以作为部分非转移类指令的源操作数被直接读取,PC 的宽度总是与 GR 的宽度一致。

基础整数指令涉及的寄存器如图 3-2 所示,包括通用寄存器(GR)与程序计数器(PC)。此外,对于浮点操作来说,浮点数指令涉及浮点寄存器(Floating-point Register,FR)、条件标志寄存器(Condition Flag Register,CFR)和浮点控制状态寄存器(Floating-point Control and Status Register,FCSR)。

图 3-2 LA32 下的通用寄存器和 PC

浮点寄存器 FR 共有 32 个,其位宽与其操作的数据有关,仅当实现操作单精度浮点数和字整数的浮点指令时,FR 的位宽为 32b,通常情况下,FR 的位宽为 64b。同时,基础浮点指令与浮点寄存器也存在正交关系。当位宽为 32b 时,数据总是出现在浮点寄存器的[31:0]位上,此时浮点寄存器的[63:32]位可以是任意值。LA32 下的浮点寄存器如图 3-3 所示。

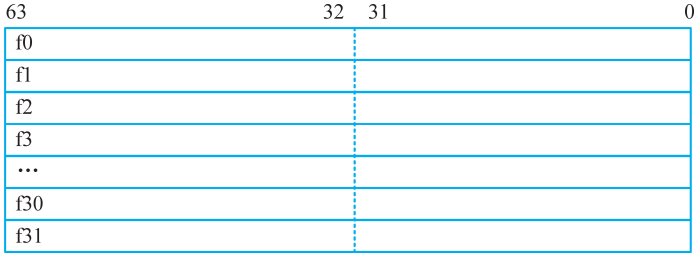

图 3-3 LA32 下的浮点寄存器

而条件标志寄存器 CFR 共有 8 个,分别记为 fcc0～fcc7,每个寄存器均支持用户进行读写。CFR 的位宽仅为 1b,用于存放浮点比较的结果,当比较结果为真则置 1,否则置 0,是浮点分支指令的判断条件。

浮点控制状态寄存器 FCSR 共有 4 个,分别记为 fcsr0～fcsr3,位宽都是 32b。其中,fcsr1～fcsr3 是 fcsr0 中部分域的别名,即访问 fcsr1～fcsr3 其实是访问 fcsr0 的某些域,进

行写操作时对应的域也会被修改。

同时，龙芯架构下定义了一系列的状态控制寄存器 CSR，这些寄存器用于控制指令的执行行为，通常每个 CSR 都会包含若干域。这些寄存器负责维护机器的运行状态信息，处理异常情况等任务。例如，在 CRMD 寄存器中的第 0 位和第 1 位是负责维护当前特权等级的寄存器域，在执行特权指令之前，都需要对该域进行查询操作以确定当前用户是否具备执行指令的权限。而在异常处理的过程中也需要将 CRMD 中的 PLV，IE 分别存到 PRMD 的 PPLV，PIE 中，再将触发异常的指令的 PC 值记录到 ERA 寄存器中，以供异常处理之后返回程序执行时使用。

在运行过程中，这些状态寄存器为软件提供四种不同的读写操作模式：可读可写，只读，读取永远返回 0，唯写入 1 有效。这些操作有效保护了系统安全并使得寄存器功能得以正常实现。龙芯架构中状态控制寄存器最多可以有 2^{14} 个，目前常用的控制状态寄存器如表 3-3 所示。通常采用 CSR.%%%%.####的形式来指称名称缩写为%%%%的状态控制寄存器中名为####的域。例如，CSR.MISC.0 就是指杂项寄存器 MISC 的保留域 0。所有控制状态寄存器的位宽，或者固定为 32 位，或者与机器所实现的具体架构是 LA32 还是 LA64 相关。对于第一种类别的寄存器，其在 LA64 架构下被 CSR 指令访问时，读返回的是符号扩展至 64 位后的值，写的时候高 32 位的值自动被硬件忽略。

此外，如果软件要操作当前系统中不存在的 CSR，即软件使用 CSR 指令访问的 CSR 对象是架构规范中未定义的，或者是架构规范中定义的可实现项但是具体硬件未实现的，此时读返回的值可以是任意值，但是写动作不应改变软件可见的处理器状态。尽管软件写这些未定义或未实现的控制状态寄存器不会改变软件可见的处理器状态，但如果想确保向后兼容，则软件不应主动写这些寄存器。

表 3-3 给出了 LA32 架构下常用的状态控制寄存器的名称与缩写标识。

表 3-3　控制状态寄存器一览表

名　称	缩　写	名　称	缩　写
当前模式信息	CRMD	异常前模式信息	PRMD
扩展部件使能	EUEN	异常配置	ECFG
异常状态	ESTAT	异常返回地址	ERA
出错虚地址	BADV	异常入口地址	EENTRY
TLB 索引	TLBIDX	TLB 表项高位	TLBEHI
TLB 表项低位 0	TLBELO0	TLB 表项低位 1	TLBELO1
地址空间标识符	ASID	低半地址空间全局目录基址	PGDL
高半地址空间全局目录基址	PGDH	全局目录基址	PGD
处理器编号	CPUID	数据保存	SAVEn(0～3)
定时器编号	TID	定时器配置	TCFG
定时器值	TVAL	定时中断清除	TICLR

续表

名　称	缩　写	名　称	缩　写
LLBit 控制	LLBCTL	TLB 重填异常入口地址	TLBRENTRY
高速缓存标签	CTAG	直接映射配置窗口	DMW0～DMW1

3.2.3 LoongArch 的寻址方式


指令的寻址方式

1. 寄存器寻址

在寄存器寻址方式(如图 3-4 所示)下,操作数就存放在 CPU 内部的寄存器中。指令中指定的寄存器号所对应的寄存器中的值即为操作数。由于无须通过访问存储器来取得操作数,因此采用寄存器寻址方式的指令具有较高的执行效率。同时寄存器寻址方式的指令字段短且执行速度快,还支持向量矩阵运算。其缺点是,CPU 中可供使用的寄存器总个数相对有限,因此寄存器使用的代价较高。

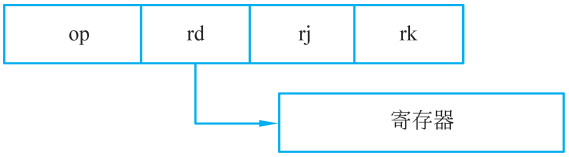

图 3-4　寄存器寻址方式

具体过程如指令:

```
add.w r4, r5, r6  ;  r4=r5+r6==4+1
```

该指令中两个操作数都是由寄存器寻址的方式获得的。在指令中指定了寄存器 r5,r6,则被指定寄存器中的内容就是指令的操作数。假设 r5 中存放的内容为 4,r6 中存放的内容为 1。在指令执行过程中,取指之后先确认指令给出的寄存器编号,然后访问寄存器 r5,r6,从中取出所需要的操作数并进行相应操作,指令执行结果存入通用寄存器 r4 中,指令执行完毕。

2. 立即数寻址

在立即数寻址方式(如图 3-5 所示)中,操作数作为指令的一部分,被包含在指令中,紧跟在操作码后面,不需要到其他地址单元中去取,取出指令也就取出了操作数,这种操作数就被称为立即数。立即数寻址方式在执行阶段不需要再取出操作数,是常见的寻址方式中速度最快的寻址方式。但是立即数寻址方式下立即数的表示范围受到指令长度的限制。

图 3-5　立即数寻址方式

具体过程如指令:

```
addi.w  r4, r5, 0x001  ; r4=r5+1==4+1
```

在指令中，r5 中存放了一个源操作数，假设为 4，对于该操作数的寻址方式就是寄存器寻址；而同时另外一个源操作数为立即数 0x001，该操作数在指令中直接给出，不需要再到其他地址单元中寻址，对于该操作数的寻址方式就是立即数寻址。指令执行过程中，首先通过寄存器寻址的方式获取 r5 中存放的操作数 4，同时通过立即数寻址的方式取得第二个操作数 0x001，将二者相加后得到的结果 5 存放到通用寄存器 r4 中，指令执行完成。

3. 基址寻址

基址寻址方式（如图 3-6 所示）下，指令中会给出一个寄存器号和一个形式地址，寄存器的内容为基准地址，而形式地址是地址偏移量。寻址操作数时需要将寄存器中的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。这样一来就可以有效扩大寻址范围（基址寻址的位数大于形式地址的位数），从而可以通过加上一个基地址在更大范围的空间内设计程序。例如，原本使用形式地址只能寻址到 0～99 的地址空间，但加上一个 100 的基址之后，采用基址寻址的方式，就可以将 100～199 的地址空间也纳入寻址范围。这样整个寻址空间就变成了 0～199，有效扩大了寻址范围，在进行程序设计时能够有更多可用的地址空间。

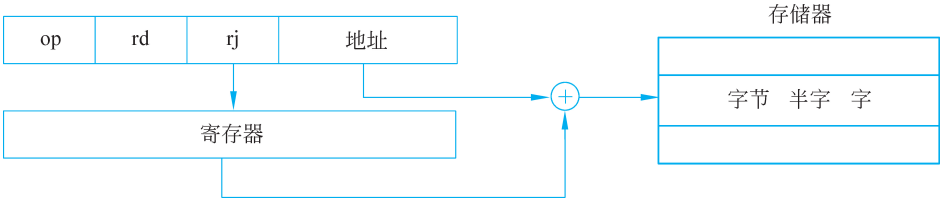

图 3-6　基址寻址方式

具体过程如指令：

```
ld.b   r4, r5, 0x001   ;从内存地址为(通用寄存器 r5 的+0x001)的位置取回一个字节的数据符
                       ;号扩展后写入通用寄存器 r4
```

指令中除目的寄存器 r4 外还给出了一个寄存器 r5，一个 12b 长的立即数 0x001，假设 r5 中存放的值为 4，该指令采取基址寻址的寻址方式。指令执行过程中首先读取给出的寄存器 r5 的内容，读出为 4，将其作为寻址操作数的基址；然后获得指令中携带的立即数 0x001，并将其进行符号扩展，得到的结果与基址相加，得到操作数的实际地址 5。然后指令进行访存操作，并从地址为 5 的位置取回一个字节的数据，符号扩展后写入通用寄存器 r4，指令执行完毕。

4. 相对基址变址寻址方式

在相对基址变址寻址方式（如图 3-7 所示）下，指令中存在两个寄存器，一个作为基址寄存器，一个作为变址寄存器。当需要操作数时将两个寄存器中的值相加求和，获得的结果就是目标地址。在典型的基址变址寻址方式中，往往还需要确定一个段寄存器作为地址基准，那么相加的结果就是地址的偏移量，偏移量再加上基准地址才能获得操作数的地址。但是龙芯架构中，虚拟地址空间是线性平整的，即每个特权级在其能够访问的地址空间中没有分

段,也就相当于龙芯架构的段基址就是整个内存空间的最小地址,无须再确定段寄存器。在龙芯架构中这种寻址方式主要用于普通访存指令中。

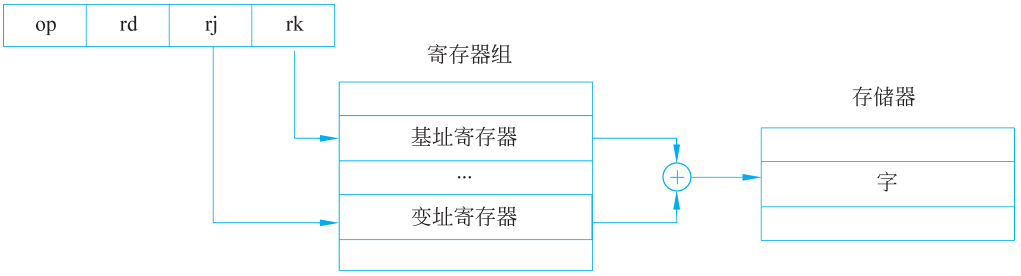

图 3-7 基址变址寻址

具体过程如指令:

```
ldx   r4, r5, r6   ;内存中某处取出一个字节的数据符号扩展后写入通用寄存器 r4,取数的地址
                   ;值等于 r5 的内容加上 r6 的内容
```

假设 r5 中的内容为 0x00000001,r6 中的内容为 0x00000002;则目标地址为 0x00000003。即指令从地址为 0x00000003 的内存位置取出一个字节的数据并进行符号扩展后存入通用寄存器 r4 中。

5. PC 相对寻址

PC 相对寻址方式(如图 3-8 所示)下,在 LoongArch 指令中会存在 16 位、21 位或 26 位三种长度的立即数。LA 的转移指令一般都会采用 PC 相对寻址方式来获得跳转的目标地址。PC 相对寻址方式中,一般先将指令码中的立即数逻辑左移两位后再进行符号扩展得到地址的偏移量,然后将获得的偏移量加上当前程序计数器 PC 的内容,获得的结果就是目标地址。PC 相对寻址的优点是其操作数地址不是固定的,会随着 PC 值的变化而变化,并且与指令地址之前总是相差一个固定值,因此便于程序浮动。

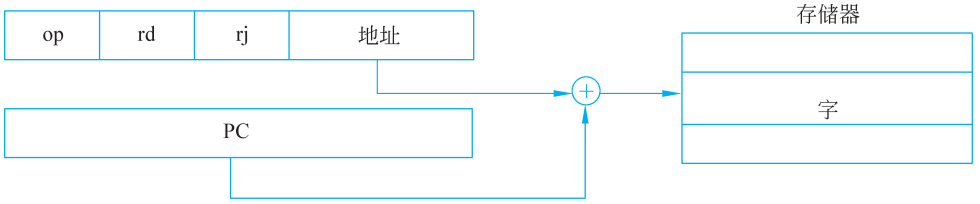

图 3-8 PC 相对寻址

具体过程如指令:

```
beq   r4, r5, 0x0001   ; 比较 r4,r5 中的内容,若两者相等则跳转到地址为(PC 寄存器的值+立
                       ;即数 0x0001 左移两位后符号扩展的值)的指令处
```

假设 r4 中的值为 1,等于 r5 中的值,当前指令的 PC 值为 4。指令中指定的两个寄存器的值相同,则进行跳转,跳转的目标地址采用 PC 相对寻址的方式进行确认。指令中给出的

立即数为 0x0001，在逻辑左移两位之后变为 0x0004，接着进行符号扩展，结果为 0x00000004，将其与 PC 值 4 相加，得到跳转的目标地址 8，跳转到该位置，指令执行结束。

3.3　LoongArch32 指令集

对于一个兼容龙芯架构的 CPU，架构中的基础指令部分必须完整实现，扩展部分可以有选择地进行实现。而对于大多数的应用场景来说，基础部分已经足以支持应用。同时，龙芯架构的基础部分包含非特权指令集和特权指令集两个部分，其中，非特权指令集部分定义了常用的整数和浮点数指令。本节将根据指令的功能划分来逐一介绍龙芯架构 LoongArch32(LA32)中基础部分的指令。

3.3.1　数据处理指令

1. ADD.W 寄存器加法指令

指令格式：

```
add.w   rd, rj, rk
```

指令功能：ADD.W 将通用寄存器 rj 中的[31:0]位数据加上通用寄存器 rk 中的[31:0]位数据，所得结果的[31:0]位符号扩展后写入通用寄存器 rd 中，即 rd＝rj＋rk。

例如：

```
add.w   r5, r6, r7          ;r5=r6+r7
```

2. SUB.W 寄存器减法指令

指令格式：

```
sub.w   rd, rj, rk
```

指令功能：SUB.W 将通用寄存器 rj 中的[31:0]位数据减去通用寄存器 rk 中的[31:0]位数据，所得结果的[31:0]位符号扩展后写入通用寄存器 rd 中，即 rd＝rj－rk。

例如：

```
sub.w   r5, r6, r7          ;r5=r6-r7
```

3. ADDI.W 带立即数的加法指令

指令格式：

```
addi.w    rd, rj, si12
```

指令功能：ADDI.W 将通用寄存器 rj 中的[31:0]位数据加上 12b 立即数 si12 符号扩

展后的 32 位数据,所得结果的[31:0]位符号扩展后写入通用寄存器 rd 中。该指令执行时不会对溢出情况做任何特殊处理。

例如:

```
addi.w  r5, r6, 5          ;r5=r6+5
```

4. ALSL.W 移位相加指令

指令格式:

```
alsl.w    rd, rj, rk, sa2
```

指令功能:ALSL.W 将通用寄存器 rj 中的[31:0]位数据逻辑左移(sa2+1)位后加上通用寄存器 rk 中的[31:0]位数据,所得结果的[31:0]位符号扩展后写入通用寄存器 rd 中。指令执行时不会对溢出情况做任何特殊处理。

例如:

```
alsl.w  r5, r6, r7, 3    ;tmp=(GR[r6][31:0]<<4)+GR[rk][31:0]将其移位后加上 r5=
                         ;SignExtend(tmp[31:0],GRLEN)所得结果的低 32 位存入 r5
```

5. LU12I.W 立即数装载指令

指令格式:

```
lu12i.w  rd, si20
```

指令功能:LU12I.W 将 20b 立即数 si20 最低位连接上 12b 的 0,然后符号扩展后写入通用寄存器 rd 中。该指令和 LA64 中的 LU32I.D,LU52I.D 指令都可以与 ORI 指令一起,用于将超过 12 位的立即数装载到通用寄存器中。

例如:

```
lu12i.w  r5, 10001000000000000000  ;在 10001000000000000000 最低位连接 12 个 0 之后
                                   ;进行符号扩展,并将结果存入 r5
```

6. SLT[U]数据比较指令

指令格式:

```
slt rd, rj, rk
sltu  rd, rj, rk
```

指令功能:SLT 将通用寄存器 rj 中的数据与通用寄存器 rk 中的数据视作有符号整数进行大小比较,如果前者小于后者,则将通用寄存器 rd 的值置为 1,否则置为 0。SLTU 则是将数据视为无符号数进行相同操作。同时,SLT 和 SLTU 比较的数据位宽与所执行机器

的通用寄存器的位宽一致。即在 LA32 下比较的数据位宽就是 32 位，而在 LA64 下比较的位宽是 64 位。

例如：

```
slt   r5, r6, r7          ;假设 r6=8,r7=9,则满足 r6 的值小于 r7,
                          ;将 r5 置为 1;否则若 r6=9,r7=8,则 r6 的
                          ;值不小于 r7,将 r5 置为 0
```

7. SLT[U]I 带立即数的数据比较指令

指令格式：

```
slti rd, rj, si12
sltui  rd, rj, si12
```

指令功能：SLTI 将通用寄存器 rj 中的数据与 12b 立即数 si12 符号扩展后所得的数据视作有符号整数进行大小比较，如果前者小于后者，则将通用寄存器 rd 的值置为 1，否则置为 0。而 SLTUI 则是将 rj 中的数据与 12b 立即数 si12 符号扩展后所得的数据都视为无符号数进行大小比较，再根据关系对 rd 进行赋值。注意 SLTUI 在对立即数进行扩展时仍然使用符号扩展方式。同时，SLTU 与 SLTUI 比较的数据位宽与所执行的机器的通用寄存器的位宽保持一致。

例如：

```
slti   r5, r6, 10         ;假设 r6=9,10 是正数,符号位为 0 则进行
                          ;扩展之后的值仍然为 10,r6<10,将 r5
                          ;置为 1;否则若 r6=11,则 r6 的值大于 10,
                          ;将 r5 置为 0
```

8. PCADDU12I 指令地址计算指令

指令格式：

```
pcaddu12i   rd, si20
```

指令功能：PCADDU12I 指令在 20b 长度的立即数 si20 的最低位连接上 12b 的 0，然后进行符号扩展，所得的数据加上该指令的 PC 值并将得到的结果写入通用寄存器 rd 中。指令操作的数据位宽与所执行机器的通用寄存器位宽一致。即符号扩展之后数据的长度在 LA32 中应该为 32b。这条指令通常可以用来计算当前指令往后第 N 条指令的地址。

9. AND，ANDI 逻辑与指令

指令格式：

```
and   rd, rj, rk
andi   rd, rj, ui12
```

指令功能：AND 指令将通用寄存器 rj 与 rk 中的数据进行按位逻辑与运算，结果写入通用寄存器 rd 中。ANDI 指令将通用寄存器 rj 中的数据与 12b 长度的立即数 ui12 零扩展之后的数据进行按位与运算，结果写入通用寄存器 rd 中。指令操作的数据位宽与所执行机器的通用寄存器位宽一致。

例如：

```
and r5, r6, r7      ;理论上寄存器中的值应该为 32 位，这里出于简便考虑，
                    ;只设 4 位，后面的逻辑指令同理。设 r6 的值为 1011，r7 的值为 0101，则两
                    ;个值按位与之后得到的数据为 0001，并将该值存入通用寄存器 r5 中
andi r5, r6, 0101   ;这里立即数也假设只有 4 位，r6 仍为 1011，结果相同
```

10. OR，ORI 逻辑或指令

指令格式：

```
or rd, rj, rk
ori rd, rj, ui12
```

指令功能：OR 指令将通用寄存器 rj 与 rk 中的数据进行按位逻辑或运算，结果写入通用寄存器 rd 中。ORI 指令将通用寄存器 rj 中的数据与 12b 长度的立即数 ui12 零扩展之后的数据进行按位或运算，结果写入通用寄存器 rd 中。指令操作的数据位宽与所执行机器的通用寄存器位宽一致。

例如：

```
or r5, r6, r7       ;仍设 r6 值为 1011，r7 为 0101，则按位或之后获得数据
                    ;1111 并将该值存入通用寄存器 r5 当中
ori r5, r6, 0101    ;r6 仍为 1011，结果相同
```

11. XOR，XORI 逻辑异或指令

指令格式：

```
xor  rd, rj, rk
xori  rd, rj, ui12
```

指令功能：XOR 指令将通用寄存器 rj 与 rk 中的数据进行按位逻辑异或运算，结果写入通用寄存器 rd 中。XORI 指令将通用寄存器 rj 中的数据与 12b 长度的立即数 ui12 零扩展之后的数据进行按位异或运算，结果写入通用寄存器 rd 中。指令操作的数据位宽与所执行机器的通用寄存器位宽一致。

例如：

```
xor r5, r6, r7        ;仍设 r6 值为 1011，r7 为 0101，则按位异或之后获得数
                      ;据 1110 并将该值存入通用寄存器 r5 当中
xori r5, r6, 0101     ;r6 仍为 1011，结果相同。
```

12. NOR 逻辑或非指令

指令格式：

```
nor  rd, rj, rk
```

指令功能：NOR 指令将通用寄存器 rj 与 rk 中的数据进行按位逻辑或非运算，结果写入通用寄存器 rd 中。指令操作的数据位宽与所执行机器的通用寄存器位宽一致。

例如：

```
nor r5, r6, r7     ;仍设 r6 值为 1011,r7 为 0101,则按位或非之后获得数
                   ;据 0000 并将该值存入通用寄存器 r5 当中
```

13. NOP 空指令

指令格式：

```
nop
```

指令功能：NOP 指令在 LoongArch 中，其实是“andi r0,r0,0”指令的别名，NOP 指令的作用仅为占据 4B 的指令码位置并将 PC 加 4，除此之外不会改变其他任何软件可见的处理器状态。NOP 指令可以用于程序延时或精确计时。

14. MUL.W，MULH.W[U]乘法指令

指令格式：

```
mul.w    rd, rj, rk
mulh.w   rd, rj, rk
mulh.wu  rd, rj, rk
```

指令功能：MUL.W 指令将通用寄存器 rj 中的数据与通用寄存器 rk 中的数据进行相乘，乘积结果的[31:0]位数据写入通用寄存器 rd 中。MULH.W 将寄存器 rj 和 rk 中的数据视为有符号数进行相乘，乘积结果的[63:32]位数据写入通用寄存器 rd 中。MULH.WU 则是将 rj 和 rk 中的数据视作无符号数进行相乘，乘积结果的[63:32]位写入通用寄存器 rd 中。

例如：

```
mul.w    r4, r5, r6     ;将 r5 和 r6 中的数据进行相乘,乘积结果的[31:0]位写入 r4
mulh.w   r4, r5, r6     ;将 r5 和 r6 中的数据当作有符号数进行相乘,乘积的[63:32]
                        ;位写入 r4
mulh.wu  r4,r5,r6       ;r5,r6 当作无符号数进行相乘,结果的[63:32]位存入 r4
```

15. DIV.W[U]除法指令

指令格式：

```
div.w  rd, rj, rk
  div.wu rd, rj, rk
```

指令功能：两种指令都是将通用寄存器 rj 中的数据除以通用寄存器 rk 中的数据，所得的商写入通用寄存器 rd 中。区别就是 div.wu 将数据当作无符号数进行计算。而 div.w 将数据当作有符号数进行计算.同时当除数是 0 时，指令的运行结果可以是任意值，且不会因此触发任何异常。

例如：

```
div.w  r4, r5, r6        ;假设 r5=4,r6=1,则指令运行结束后 r4=4
```

16. MOD.W[U]取余指令

指令格式：

```
mod.w   rd, rj, rk
mod.wu  rd, rj, rk
```

指令功能：MOD.W 指令将 rj 和 rk 中的数据当作有符号数进行操作，MOD.WU 将其当作无符号数进行操作。两条指令都会将 rj 中的数据除以通用寄存器 rk 中的数据，所得的余数写入通用寄存器 rd 中。同时每一对求商/余数的指令对(DIV.W/MOD.W，DIV，WU/MOD.WU)的运算结果都满足余数与被除数的符号一致且余数的绝对值小于除数的绝对值。

例如：

```
mod.w  r4, r5, r6        ;假设 r5=5,r6=6,则指令运行结束后 r4=5
```

17. SLL.W，SLLI.W 逻辑左移指令

指令格式：

```
sll.w    rd, rj, rk
slli.w   rd, rj, ui5
```

指令功能：SLL.W 和 SLLI.W 两条指令都是将通用寄存器 rj 中的数据逻辑左移，然后将移位结果写入通用寄存器 rd 中。不同之处在于 SLL.W 的移位位数是由 rk 中[4:0]位的数据决定的，而 SLLI.W 的移位位数是由指令中给出的无符号立即数 ui5 决定的。

例如：

```
sll.w r4, r5, r6     ;假设 r5=4,r6=2,则指令运行结束后 r4=16
```

18. SRL.W，SRLI.W 逻辑右移指令

指令格式：

```
srl.w     rd, rj, rk
srli.w    rd, rj, ui5
```

指令功能：SRL.W 和 SRLI.W 两条指令都是将通用寄存器 rj 中的数据逻辑右移，然后将移位结果写入通用寄存器 rd 中。不同之处在于 SRL.W 的移位位数是由 rk 中[4:0]位的数据决定的，而 SRLI.W 的移位位数是由指令中给出的 5 位无符号立即数 ui5 决定的。

例如：

```
srl.w   r4, r5, r6   ;假设 r5=4,r6=2,则指令运行结束后 r4=1
```

19. SRA.W,SRAI.W 算术右移指令

指令格式：

```
sra.w     rd, rj, rk
srai.w    rd, rj, ui5
```

指令功能：SRA.W 和 SRAI.W 两条指令都是将通用寄存器 rj 中的数据算术右移，然后将移位结果写入通用寄存器 rd 中。不同之处在于 SRA.W 的移位位数是由 rk 中[4:0]位的数据决定的，而 SRAI.W 的移位位数是由指令中给出的 5 位无符号立即数 ui5 决定的。

例如：

```
srai.w   r4, r5, 1       ;假设 r5=4,则指令运行结束后 r4=2
```

20. EXT.W.{B/H}符号扩展指令

指令格式：

```
ext.w.b     rd, rj
ext.w.h   rd, rj
```

指令功能：EXT.W.B 将通用寄存器 rj 中[7:0]位数据符号扩展后写入通用寄存器 rd 中；EXT.W.H 将通用寄存器 rj 中[15:0]位数据符号扩展后写入通用寄存器 rd 中。

例如：

```
ext.w.h   r4, r5     ;假设 r5 中[15:0]位的值为-1,则符号扩展后 32 位数为
                     ;-2147450881,将结果存入 r4 中
```

21. CL{O/Z}.W,CT{O/Z}.W 按位计数指令

指令格式：