计算机体系结构基础

胡伟武、苏孟豪

第02章 指令系统

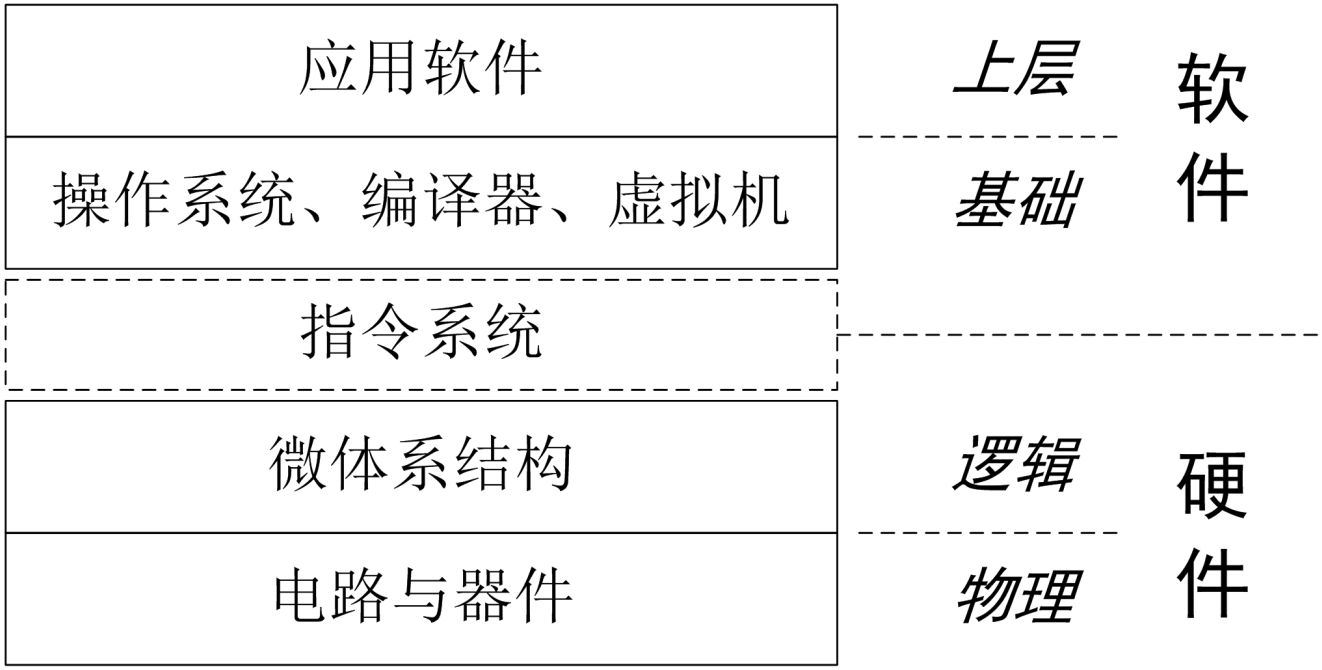
- 指令系统简介
- 指令系统的设计原则
- 指令系统的演变
- 指令系统组成
  - 地址空间
  - 操作数
  - 指令操作和编码
- RISC指令系统比较
- C语言的机器表示

指令系统简介

什么是指令系统

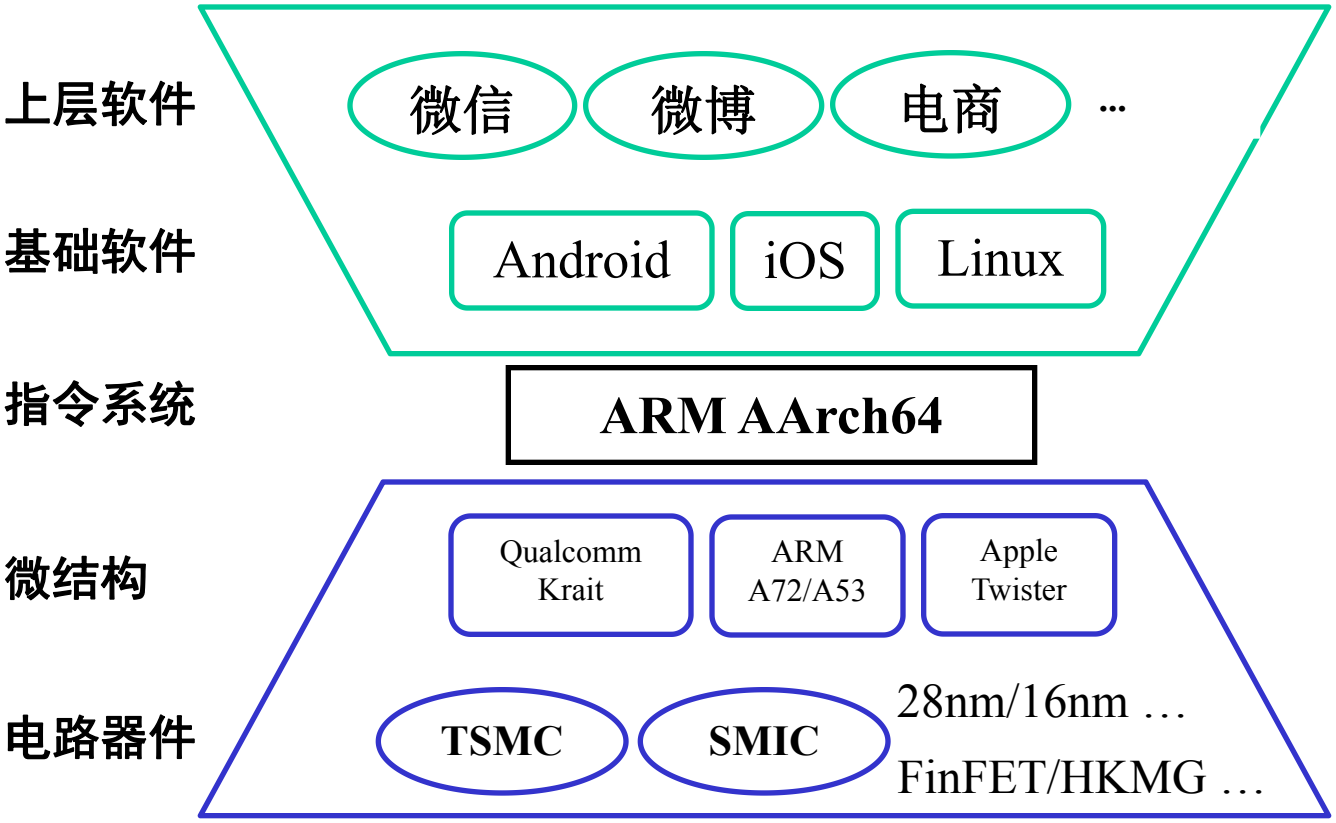
- 指令系统是计算机功能的抽象模型
  - 是软硬件的界面，所有软件最终都以指令的形式运行
  - 体现了结构设计者对应用的深刻理解
  - 一个指令系统可以有多种实现（低功耗、高性能、软件模拟...）
- 指令系统结构不仅仅是关于指令功能的编码
  - 运行环境：地址空间、异常和中断处理、存储管理、安全管理、Cache管理等（思考：应用软件是否能够为所欲为）
  - 运行环境差异比功能差异大，如MIPS有5组寄存器，PPC有14组

指令系统 – 承上启下



指令系统 – 承上启下

- 以ARM指令系统为例



指令系统为什么重要

- 指令系统是计算机产业的枢纽，产业生态的基础
  - 指令系统是计算机软硬件的重要标准
  - 决定应用程序的二进制兼容（Wintel和AA都做到）
  - 是操作系统二进制兼容（Wintel做到、AA没做到）的重要因素
  - 如技术上比X86做得好的指令系统都死得差不多了
- 指令系统影响系统性能和实现复杂性等
  - 如RISC/CISC，32位/64位，媒体指令，向量指令等
  - 微结构对系统复杂性影响更大

国际主流指令系统分析

- 目前三种较流行的指令系统：X86、ARM、MIPS
  - X86：每年几亿片，垄断PC和服务器市场，虽然受到ARM的一些威胁，但桌面的垄断地位难以动摇，并通过凌动开辟部分移动终端和嵌入式市场
  - ARM：每年50-100亿片，在手持终端市场处于垄断地位，并不断侵蚀MIPS的数字电视、机顶盒等市场，正在往高端发展，在云服务器等领域与X86竞争。
  - MIPS：每年5-10亿片，在X86和ARM的夹缝中艰难生存，但在传统优势市场如数字电视、打印机、网络等仍有一定势力
- 其它指令系统
  - PowerPC在汽车电子、工控、服务器领域还有一线生机
  - Alpha、PA-RISC、Sparc、IA64主流市场机会不大

指令系统是不断发展的

- 作为计算机软硬件的界面，指令系统是不断发展的
  - X86位宽从8位、16位、32位、64位不断发展
  - 功能从只支持定点、到浮点、到媒体指令、到向量指令
  - 向量指令又经历了MMX、SSE、SSE2、SSE3、SSE4、AVX等
- 技术和应用发展对指令系统提出新要求
  - 多核结构要求指令增加对多核同步、通信和数据一致性的支持
  - 向量部件需要指令系统增加专门的向量指令
  - 媒体类应用要求指令系统增加对媒体编解码的专门支持
  - 云计算要求指令系统增加对虚拟机的支持；等等

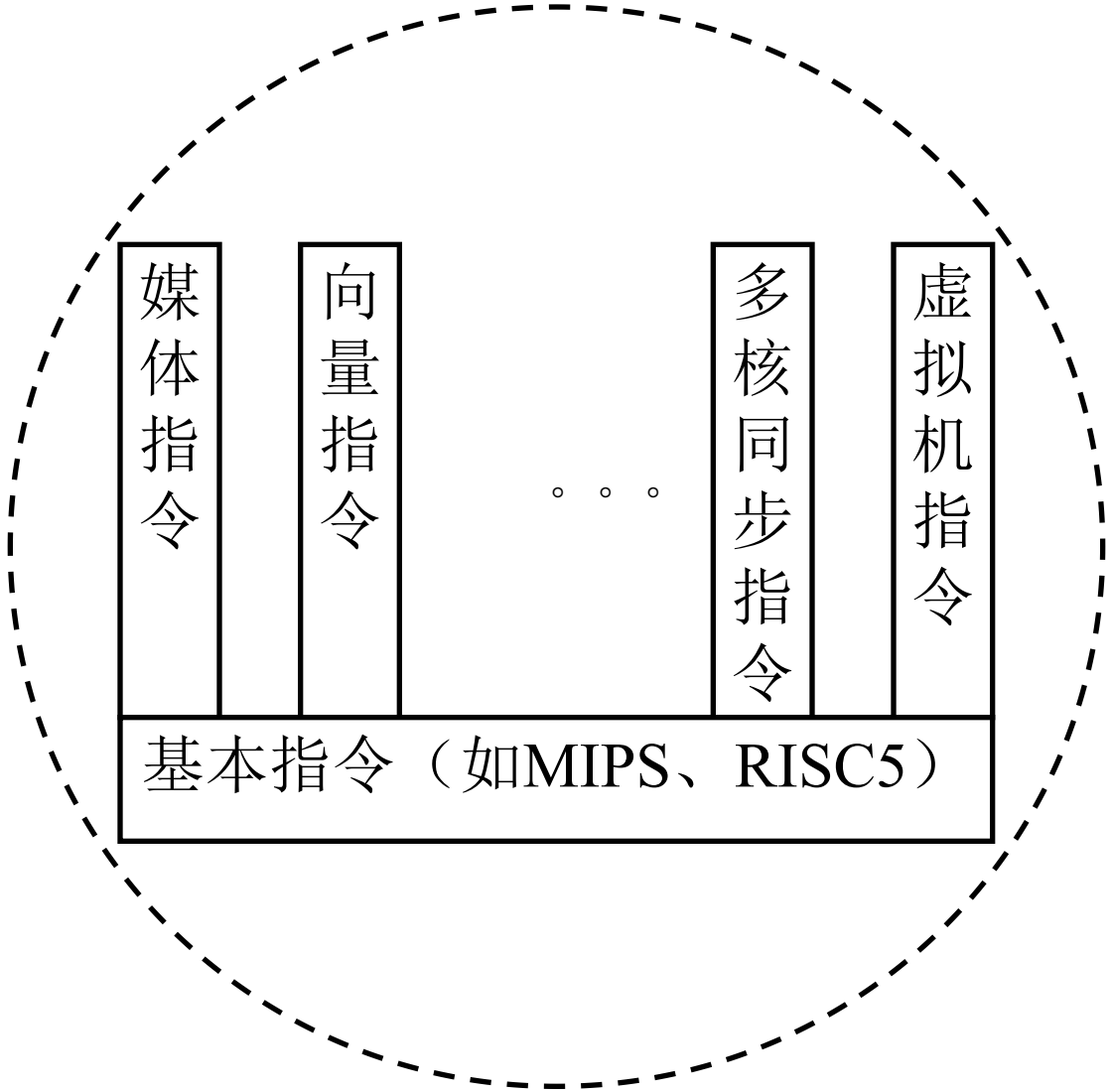
自主软硬件需要自主指令系统

- 我国需要发展自主可控的软硬件，已逐渐成为政府、军队、学术界、公众的共识
  - 在目前的IT产业体系中，国外垄断企业设置了严密的知识产权壁垒
  - 主要体现在包括指令系统在内的各类接口中
- 目前自主软硬件推进效果明显，但我国软硬件力量过于薄弱，通过统一的自主指令系统可以迅速形成合力
  - 指令系统是自主可控软硬件的重要标准
- 目前IT产业正从单极化向多极化发展，我国要争取在IT产业的多极世界中形成既开放又竞争的一极
  - 与X86和ARM三分天下

我国发展指令系统的可行路径

- 先兼容后自主
  - 与国外指令兼容（MIPS、RISC等），在此基础上自主发展
  - “可控”权：自主扩展和自主再授权
  - “对等”权：协商扩展和反向授权
- 先自主后兼容
  - 通过二进制翻译方式，运行主流指令系统的软件
  - 如Transmeta，IA64上运行IA32程序等
  - Intel推出的基于X86的智能手机能运行Android上ARM的应用
- 二者结合
  - 如基于MIPS/RISC5，并通过扩展实现对X86和ARM的兼容

自主指令系统示意图



指令系统的设计原则

指令系统的设计原则

- 指令系统在计算机系统中的位置
  - 硬件和软件的界面
  - 反映了结构设计者对计算机系统的认识
- 设计原则
  - 兼容性：对软件的包容性，长时间保持不变，如**X86**
  - 通用性：对软件的易用性，编译器和程序员觉得好用
  - 高效性：对硬件的易用性，便于**CPU**设计优化和不同性能的实现
  - 安全性：对软硬件安全的支持，支持通用操作系统，考虑不同的安全要求

影响指令系统设计的因素

- 工艺技术
  - 早期的硬件昂贵，简化硬件是指令系统设计的主要因素
  - 现在如何发挥存储层次的效率，如何利用芯片面积
- 系统结构
  - 增加指令功能还是提高主频？
  - 并行性： SIMD、向量、多发射（兼容性好）
- 操作系统
  - 多进程支持、虚地址空间、安全等级、虚拟机等
- 编译技术与程序设计语言
  - 指令的表达能力
- 应用程序
  - 应用适应性、兼容性等

工艺技术对指令系统的影响

- 早期的指令系统设计主要考虑如何减少硬件
- 后来集成度的提高使得系统结构的优化成为可能
  - TLB、从32位到64位、SIMD媒体运算
- 由于CPU与存储器的速度差距，指令系统应能较好地利用存储层次，如通过并行或流水容忍延迟
  - Cache管理指令、预取指令
- 随着工艺的进一步发展，由于主频极限和功耗问题引起的多核结构需要特殊指令支持
  - 多线程管理和同步

系统结构对指令系统的影响

- 指令系统本身是系统结构发展的结果
  - 从16位、到32位、到64位
  - SIMD指令、从单核到多核等
- 指令系统的兼容性要求与系统结构发展的矛盾关系
  - 尽量不改变指令系统的前提下提高性能，如流水、多发射等
  - 尽量保持兼容，如Intel的做法
- 增加指令功能还是提高主频
  - RISC vs. CISC vs. VLIW
- 指令中如何体现并行性？
  - SIMD、多线程。。。

操作系统对指令系统的影响

- 操作系统专用的核心态指令和运行环境
- 多进程和虚空间
  - 页表与TLB的关系
  - 页保护：读写权限
- 系统安全等级
  - 核心态和用户态管理
- 异常和中断的处理
  - 异常处理入口、ERET指令等
- 访存和访问I/O的区别
- 虚拟机：支持多操作系统的快速切换

编译技术对指令系统的影响

- 指令是编译器的工作结果
  - 早期的指令系统主要考虑如何便于编程
  - 后期（如RISC）兼顾便于编程和实现效率
- 指令功能
  - 只有简单指令，甚至乘法都由加法和移位来实现
  - 具有复杂指令，如除法、开方
  - 更复杂的函数由库函数实现（如C库）
- 寄存器和存储器分配
  - 堆栈存放局部变量，全局数据区存放静态数据，堆存放动态数据
  - 为有效使用图着色启发式算法，至少需要16个通用寄存器
- 简单规整，提高编译效率
  - 正交性，如所有访存指令都可用所有寻址方式
  - 简化编译器取舍，如允许编译时确定常量，只提供基本的通用操作（less is more）等

应用对指令系统的影响

- 指令系统归根到底是为应用设计的
  - 指令系统体系结构设计者对应用的精确理解
  - 指令系统随着应用的发展而发展，科学和工程技术、事务处理、网络和媒体处理等应用在指令系统上都有相应的体现
- 应用要求指令系统保持兼容
  - 更新计算机时，兼容老的应用

指令系统的演变

指令系统的演变 – 分类

- 依据指令长度的不同，指令系统分为
  - 复杂指令系统（Complex Instruction Set Computer，简称CISC），其指令长度可变（x86: 1~15字节）
  - 精简指令系统（Reduced Instruction Set Computer，简称RISC），其指令长度比较固定
  - 超长指令字（Very Long Instruction Word，简称VLIW），本质上是多条同时执行的指令的组合，其“同时执行”的特征由编译器指定，无需硬件进行判断

指令系统的演变 – CISC

- 早期的CPU都采用CISC结构
  - IBM System360、Intel 8080/8086、Motorola 68000
  - 一方面，昂贵的软硬件系统要求兼容性
  - 另一方面，指令集的不完善要求新增指令
- 兼容性 vs. 历史包袱
  - Intel的辉煌和包袱
  - 指令集臃肿导致实现复杂，降低常用指令的运行效率
- 简单指令有利于高效实现
  - 程序中80%的指令只占指令集的20%
  - X86指令通过内部译码后的微操作类似于RISC

指令系统的演变 - RISC

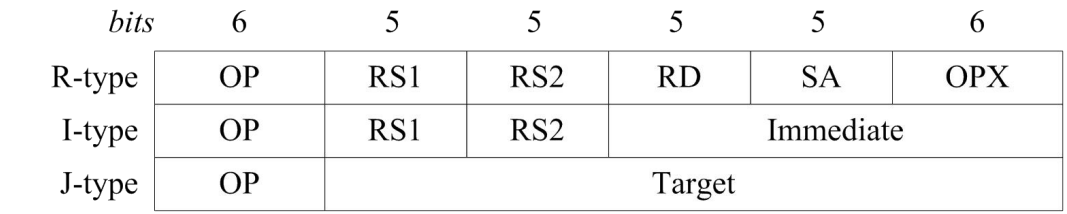
- 丢掉包袱，轻装上阵
- 核心思路：简化
  - 简化指令功能：执行时间短
  - 简化指令编码：译码简单
  - 简化访存类型：访存和运算分开
- **Power、MIPS、ARM、SPARC、Alpha……**
- 现代指令系统对**CISC**和**RISC**的融合
  - 核心流水线采用**RISC**
  - 包含复杂功能的宏指令

指令系统的演变 - VLIW

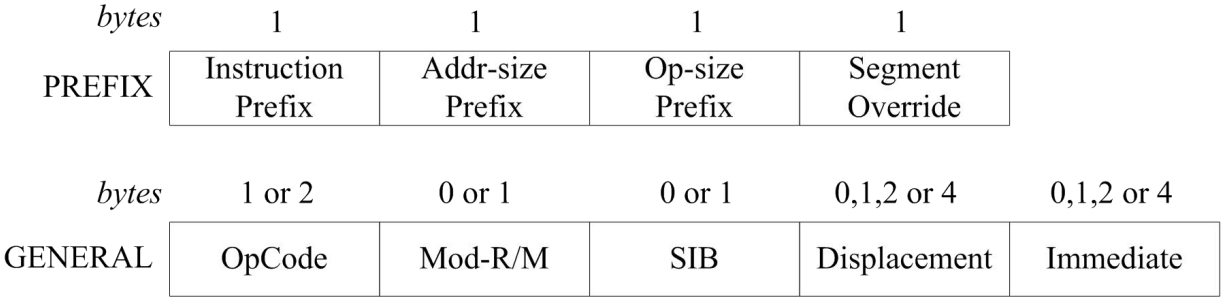
- 指令级并行（Instruction Level Parallelism，简称ILP）的极端
  - 同一个指令字中的多条指令不存在相关
  - 对编译器提出了更多的要求
- TRACE、Itanium（IA-64）
- 应用在早期的GPU中，GPGPU发展后遭弃用
  - AMD转向了SIMT的GCN架构

指令系统的演变 – 示例

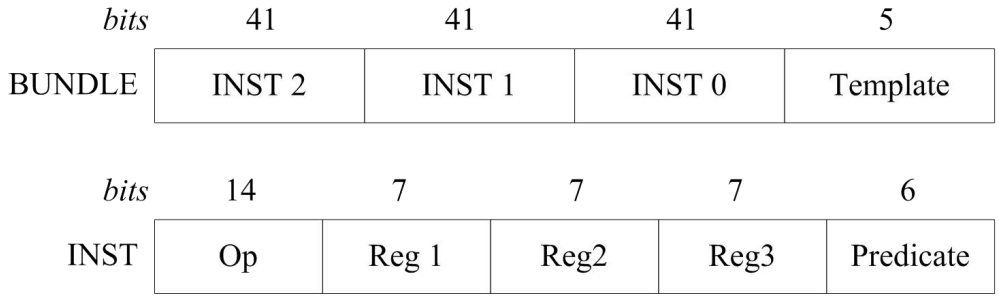
RISC
(MIPS)



CISC
(X86)



VLIW
(IA-64)

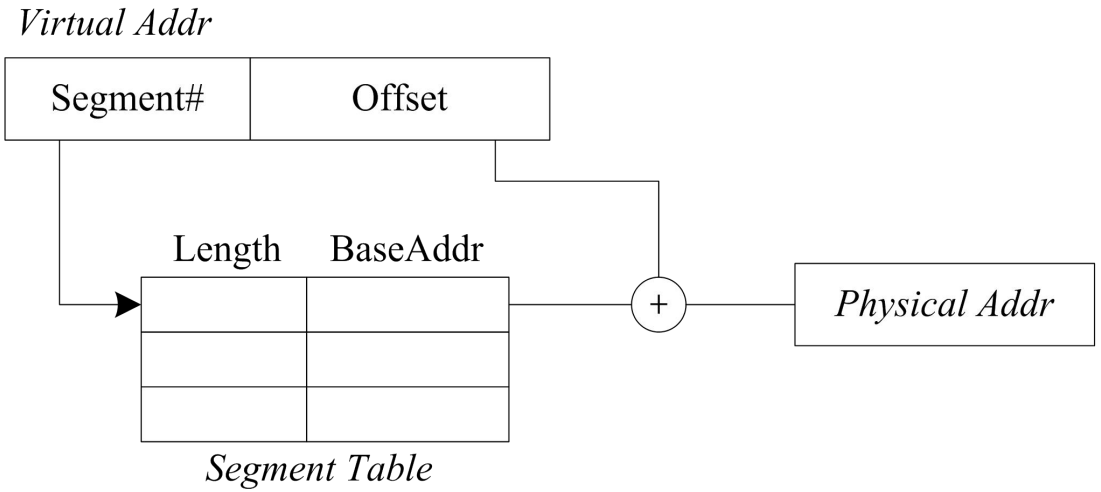


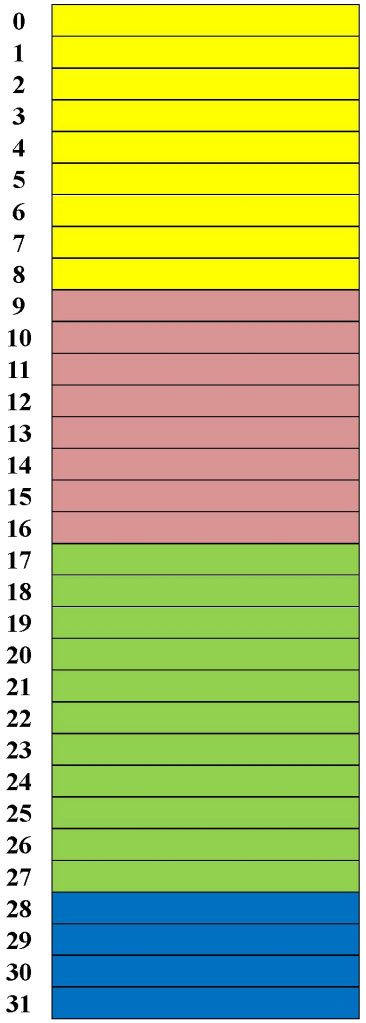
存储管理的演变 – 分类

- 连续实地址
  - 各程序数据连续存放，显式保证不冲突
  - 碎片多、管理难
- 段式
  - 分为多个段，通过相对段的偏移来访问
  - **Segment Fault**
- 页式虚拟存储
  - 将虚地址和实地址的对应关系组织为页表
  - 通过**TLB**进行硬件支持
- 段页式：融合段式和页式

存储管理的演变 – 段式

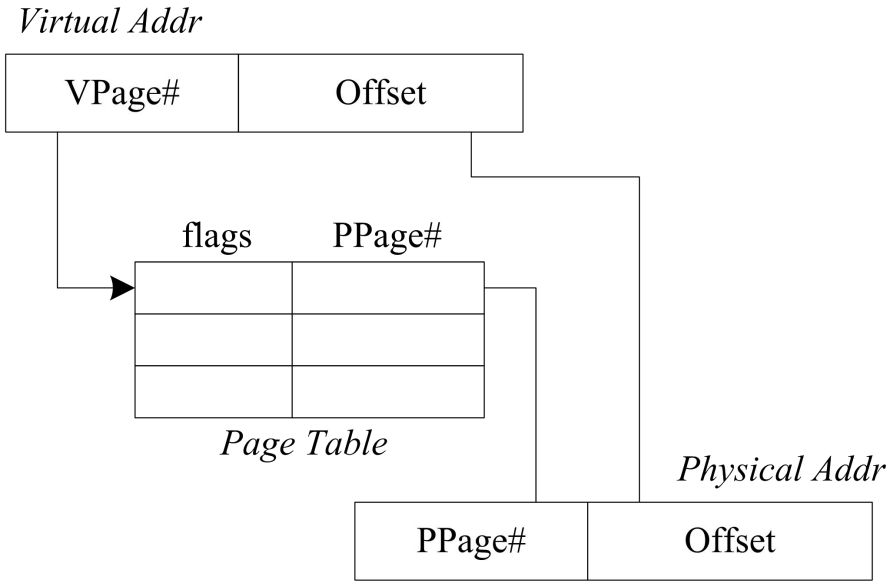
- 段表包含长度检查，偏移超过长度产生段错误
- 地址结果通过加法得到

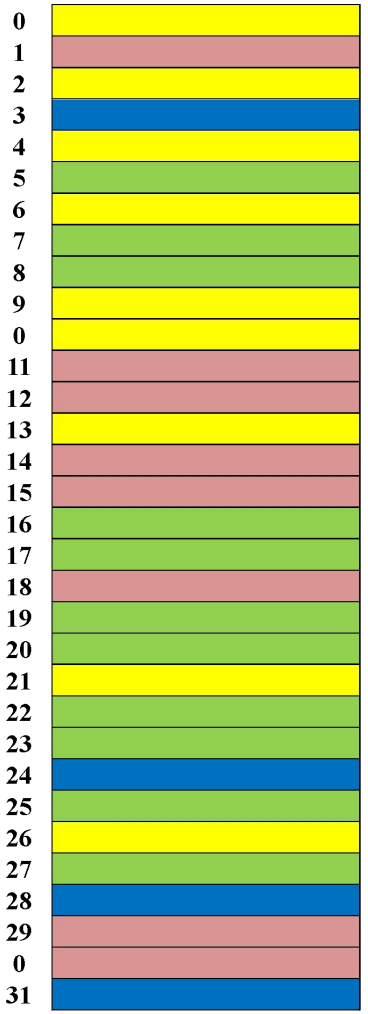




存储管理的演变 – 页式

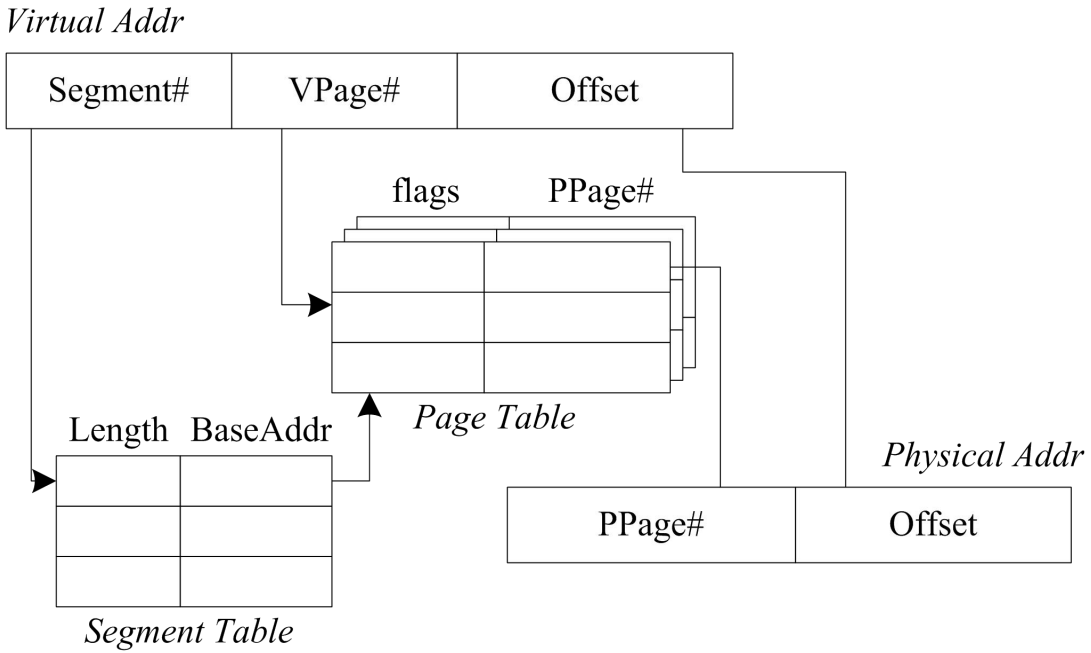
- 页表中包含有效、可写等标志信息
- 地址结果通过页地址和页内地址组合得到

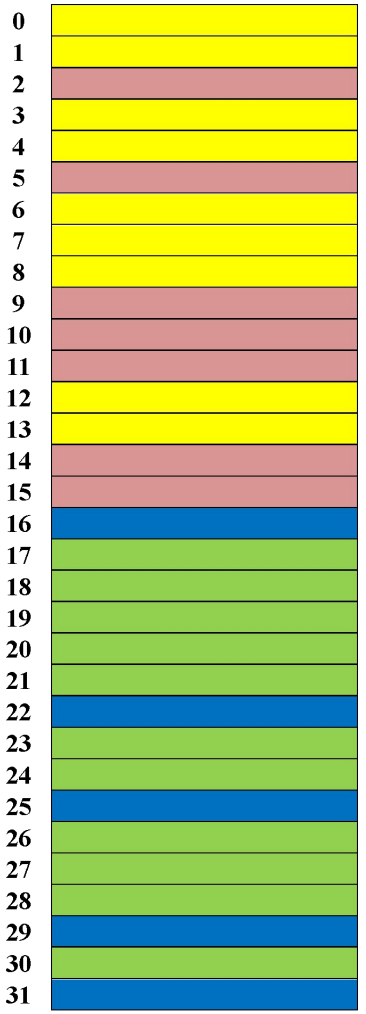




存储管理的演变 – 段页式

- 先分段，段内分页
  - 图中0-15页为第一段，16-31页为第二段
  - 根据段表得到本段的页表位置





运行级别的演变 – 分类

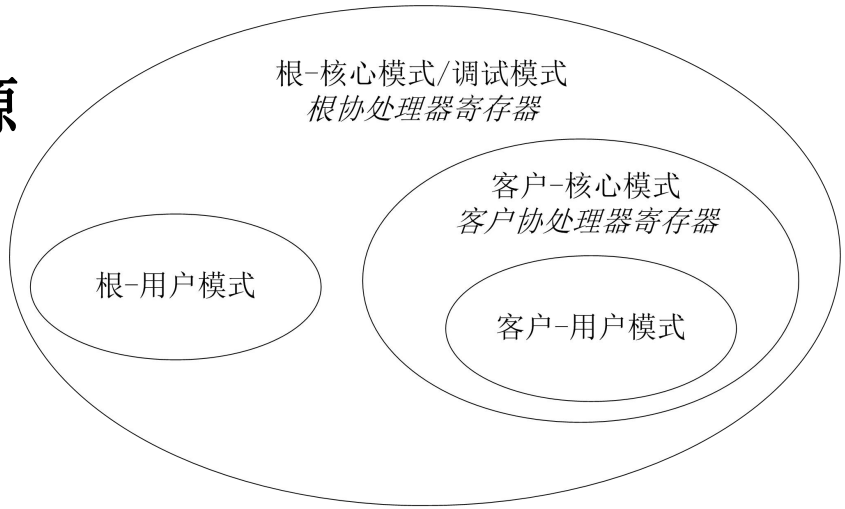
- 唯一实模式：无管理
- 保护模式：权限管理
  - 核心态：掌握全部资源
  - 用户态：只能访问受限的内存，不能访问I/O
- 调试模式：调试支持
  - ARM JTAG、MIPS EJTAG
- 客户模式：虚拟机支持
  - Intel非根模式、MIPS客户模式

运行级别的演变 - MIPS

<table>
<tr><th colspan="5">Root CP0</th><th colspan="3">Guest CP0</th><th rowspan="2">模式</th></tr>
<tr><th>Debug.
DM</th><th>Status.
ERL</th><th>Status.
EXL</th><th>GuestCtl
0.
GM</th><th>Status.
KSU</th><th>Status.
ERL</th><th>Status.
EXL</th><th>Status.
KSU</th></tr>
<tr><td>1</td><td colspan="7">Don’t care</td><td>调试模式</td></tr>
<tr><td rowspan="10">0</td><td>1</td><td colspan="6">Don’t care</td><td rowspan="3">根-核心模式</td></tr>
<tr><td rowspan="9">0</td><td>1</td><td colspan="5">Don’t care</td></tr>
<tr><td rowspan="3">0</td><td>0</td><td>00</td><td colspan="3">Don’t care</td></tr>
<tr><td>01</td><td colspan="3">Don’t care</td><td>根-监管模式</td></tr>
<tr><td>10</td><td colspan="3">Don’t care</td><td>根-用户模式</td></tr>
<tr><td rowspan="5">1</td><td rowspan="5">Don’t
care</td><td>1</td><td colspan="2">Don’t care</td><td rowspan="3">客户-核心模式</td></tr>
<tr><td rowspan="4">0</td><td>1</td><td>Don’t care</td></tr>
<tr><td rowspan="3">0</td><td>00</td></tr>
<tr><td>01</td><td>客户-监管模式</td></tr>
<tr><td>10</td><td>客户-用户模式</td></tr>
</table>

运行级别的演变 - MIPS

- 根模式和客户模式下各有一组协处理器寄存器用于操作系统上下文的快速切换
- 各个模式可访问/控制的资源呈包含关系：



- 多个层次的隔离与控制
  - 应用程序：独立虚地址空间、寄存器
  - 操作系统(OS)：独立客户物理地址空间、控制寄存器，可管理应用程序的地址映射，转换到客户物理地址
  - 虚拟机(Hypervisor)：可管理客户系统的物理地址映射，转换到最终的根物理地址

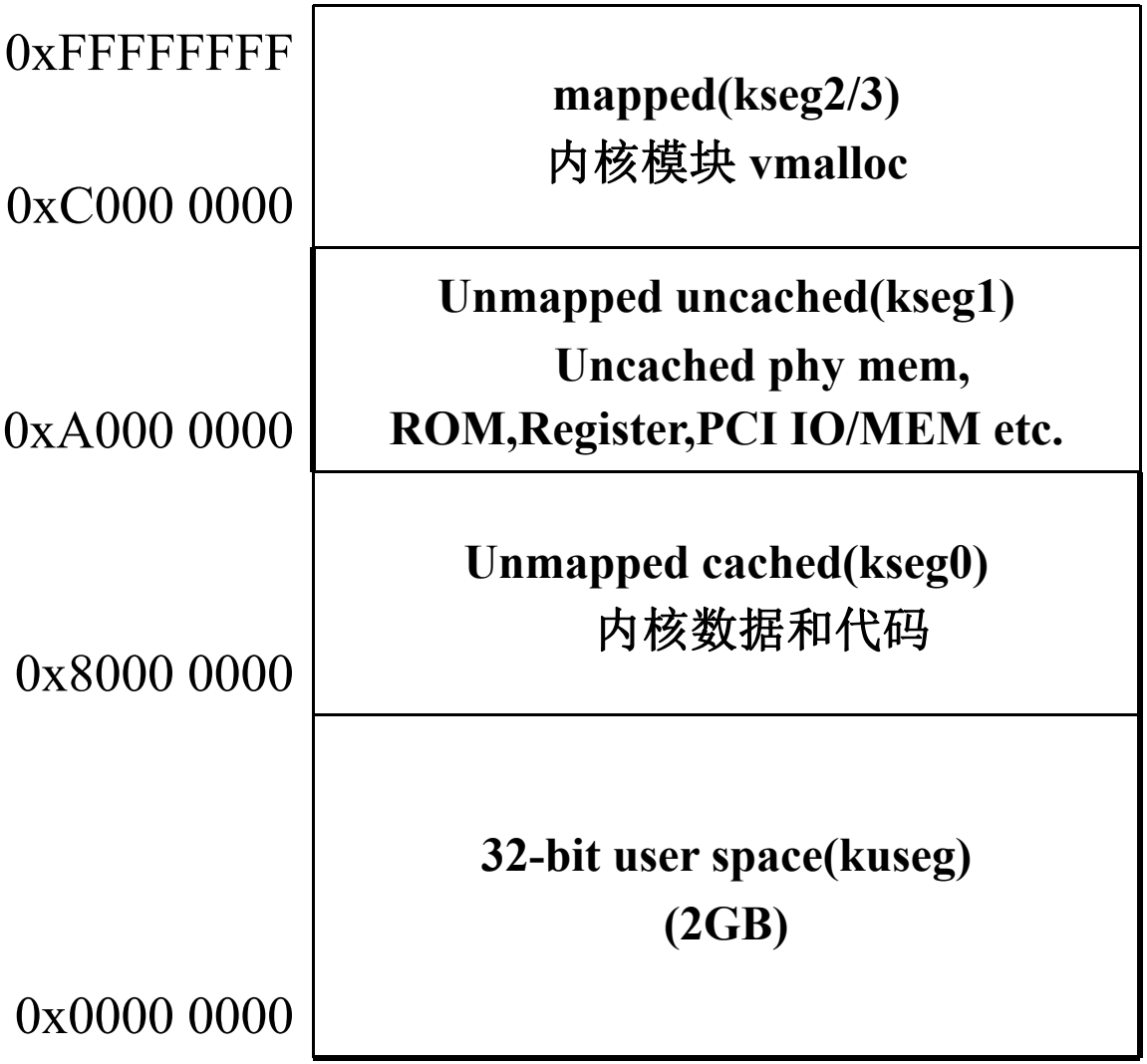
核心态的“特权”

- 有些地方只有核心态能访问
  - 控制寄存器、TLB、Cache、IO空间、特定地址空间
  - 访问上述空间的指令在用户态运行时硬件发例外
- 核心态和用户态的转换
  - 用户态下执行系统调用或发生例外时进入核心态
  - ERET从核心态回到用户态
- 例如：翻一页PPT过程中
  - 硬件响应中断从用户态到核心态，操作系统调用PPT后回到用户态
  - PPT要调用通过系统调用多次进出核心态以通过IO操作完成翻页（读硬盘、显示等）

MIPS32存储空间分段情况

地址范围	容量	映射方式	Cached	访问权限
0xe0000000-0xffffffff	0.5GB	查找TLB	Yes (TLB)	Kernel
0xc0000000-0xdfffffff	0.5GB	查找TLB	Yes (TLB)	Kernel, Supervisor
0xa0000000-0xbfffffff	0.5GB	地址-0xa0000000	No	Kernel
0x80000000-0x9fffffff	0.5GB	地址-0x80000000	Yes (Config)	Kernel
0x00000000-0x7fffffff	2GB	查找TLB	Yes (TLB)	Kernel, Supervisor, User

Linux/MIPS虚拟地址空间安排



指令系统组成

指令系统的组成

- 指令的主、谓、宾
  - CPU、操作、操作数
- 操作
  - 算术与逻辑运算、转移、访存、系统指令。。
- 操作数
  - 数据类型：定点/浮点，32位/64位。。
  - 访存对象：字节/半字/字/双字，大/小尾端。。
  - 寻址方式：寄存器、立即数、直接、间接。。
- 指令编码
  - 定长、变长

指令系统组成

---操作数的存储（地址空间）

地址空间的组成

- 寄存器空间
  - 整数通用寄存器
  - 浮点通用寄存器
  - 协处理器寄存器
- 系统内存空间
  - 内存空间
  - IO空间
- 访问方式
  - 寄存器：在指令中以寄存器号引用
  - 系统内存：访存指令

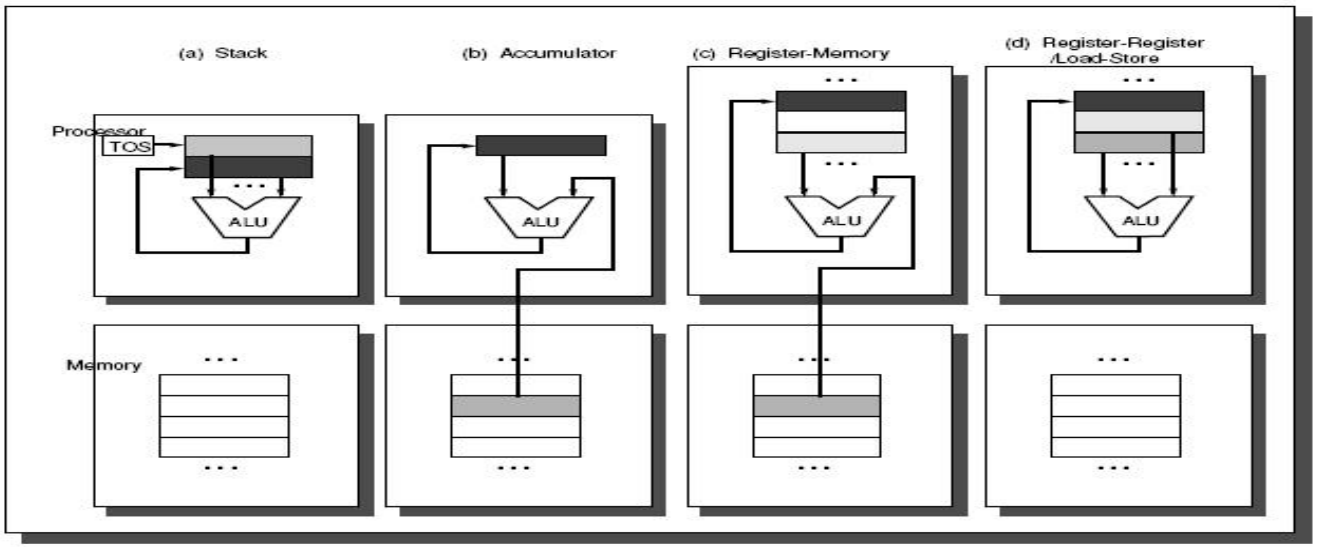
指令系统的地址空间的演变

- **堆栈型（Stack）：零地址指令**
  - 操作数在栈顶，运算操作不用指定操作数
- **累加器型（Accumulator）：单地址指令**
  - 一个操作数总在累加器中，结果也写回累加器
- **寄存器型（Register）：多地址指令**
  - **Register-Register型**
  - **Register-Memory型**
  - **Memory-Memory型**

不同类型指令功能举例

- 例子：不同指令系统完成C=A+B的指令序列
  - 假设A、B、C在内存中不同的单元

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1, A	Load R1,A
Push B	Add B	Add R1, B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C R3



指令系统类型的发展

- 早期的计算机多用堆栈和累加器型指令
  - 出于降低硬件复杂度的考虑
  - 现在已经不用（Intel有点例外）
- 1980年代后的机器主要是寄存器型
  - 访问寄存器比访问存储器快,便于编译器使用和优化
  - 寄存器可以用来存放变量，减少访存次数
  - 寄存器间的相关容易判断，易于实现流水线、多发射、乱序执行等
  - X86通过把复杂指令翻译成类似于RISC的内部操作并使用RISC指令流水线技术提高性能，X86的向量指令也是寄存器型
- RISC的不断复杂化
  - 内存离寄存器越来越远，以寄存器为中心的结构增加了不必要的数据搬运开销（如memcpy）
  - 向量指令、超越函数指令、Transacitonal memory等

不同指令系统通用寄存器数量

指令集	整数通用寄存器数
Itanium	128
VAX	16
ARMv8	31
PowerPC	32
Alpha	32(including ”zero”)
SPARC	32(including ”zero”)
MIPS	4-32(including ”zero”)
ARMv7	7 in 16bit thumb mode,14 in 32bit
X86	8 in 16/32bit,16 in 64bit

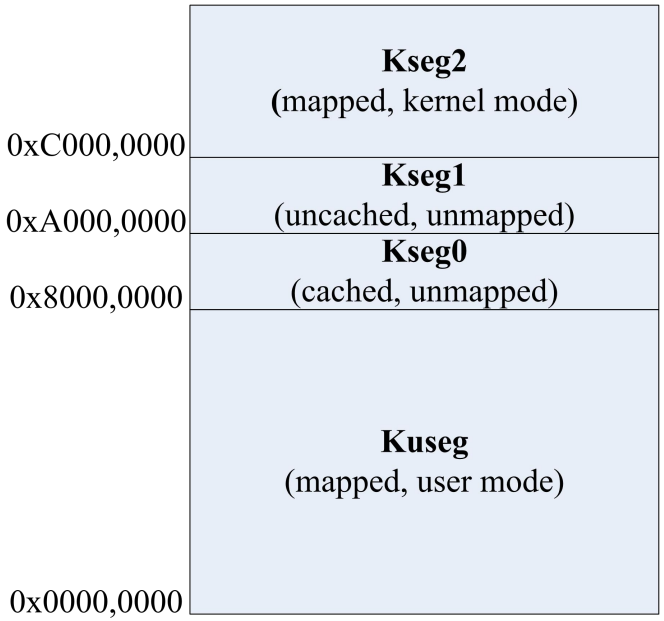
MIPS寄存器空间

- 32个整数通用寄存器
  - $0 - $31（$0总是返回0）
- 32个浮点通用寄存器
  - $f0 - $f31
- 控制寄存器
  - 仅OS使用
- 若干协处理器寄存器
  - DSP、MSA……

助记符	编号	说明
SR	12	状态寄存器，包含CPU特权等级、中断使能和其他模式配置。
Cause	13	标记异常和中断发生的原因。
EPC	14	异常程序计数器，异常和中断处理结束后的重新执行地址。
Count	9	组成一个简单但有用的高精度内部计数器。
Compare	11	
BadVAddr	8	存放导致地址相关异常的程序地址。
Context	4	存储管理（TLB）相关寄存器，将在第五章进行详细介绍。
EntryHi	10	
EntryLo0-1	2-3	
Index	0	
PageMask	5	
Random	1	
Wired	6	
PRId	15	CPU类型和版本标识。
Config	16	CPU设置参数，依据实现而不同。
Config1-3	16.1-3	
EBase	15.1	异常入口指针基地址，多处理器时作为CPU标识。
IntCtl	12.1	向量中断和中断优先级相关设置。
SRSCtl	12.2	影子寄存器相关控制。
SRSMap	12.3	
CacheErr	27	用于分析内存错误的寄存器。
ECC	26	
ErrorEPC	30	
TagLo	28.0	cache操作相关寄存器。
DataLo	28.1	
TagHi	29.0	
DataHi	29.1	
Debug	23.0	EJTAG调试单元相关寄存器。
DEPC	24.0	
DESAVE	31.0	
WatchLo	18.0	数据观测点寄存器，当CPU对该地址进行访存时会触发异常。
WatchHi	19.0	
PerfCtl	25.0	性能计数器寄存器。
PerfCnt	25.1	
LLAddr	17.0	存放LL指令的地址。
HWREna	7.0	决定哪些硬件寄存器对用户特权程序可访问。

系统内存空间-MIPS

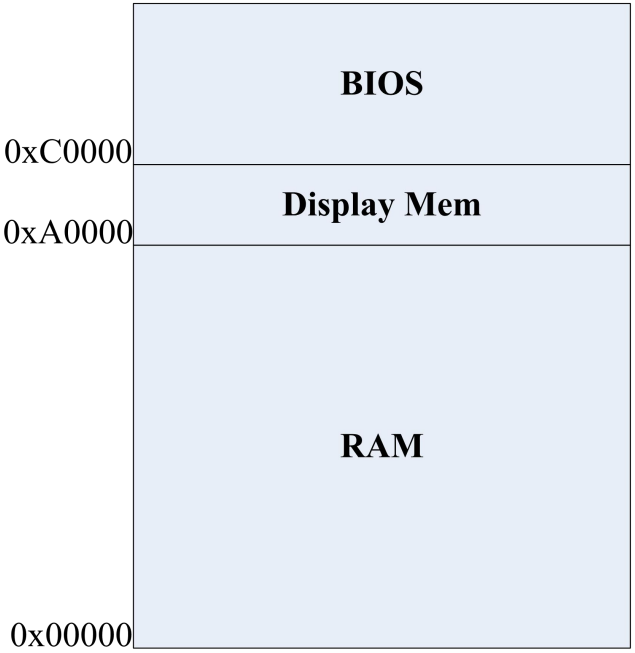
- **IO空间？**
  - **X86规定了独立的IO空间，使用专门的in/out指令来访问**
  - **MIPS/ARM不区分IO空间和内存空间，使用同样的访存指令**



(1) 32位MIPS内存空间

Kseg2/3：访问经过存储管理，只能由核心态访问
Kseg1：访问不经过存储管理，直接映射到从0开始的物理地址，且不使用cache
Kseg0：访问不经过存储管理，直接映射到从0开始的物理地址，但使用cache
Kuseg：访存经过存储管理，用户态程序只能访问这段内存

系统内存空间-X86



(2) 8086实模式内存空间

BIOS：
包括VGA BIOS ROM、IDE BIOS ROM、System BIOS Routine、BIOS Boot Block等内容

Display Mem：
包括Text Video Buffer、VGA/XGA Graphic Video Buffer等

RAM：
包括Interrupt Vector（低1K）、Boot Sector（31KB起的512B）等

指令系统组成

---操作数的表示

数据类型和大小

- 类型：
  - 整数、实数、字符、十进制数
  - 字节、半字、字、双字
  - **IEEE 754格式**
- 类型的表示
  - 一般由操作码来区分不同类型
  - 专门的类型标志

数据类型的分布

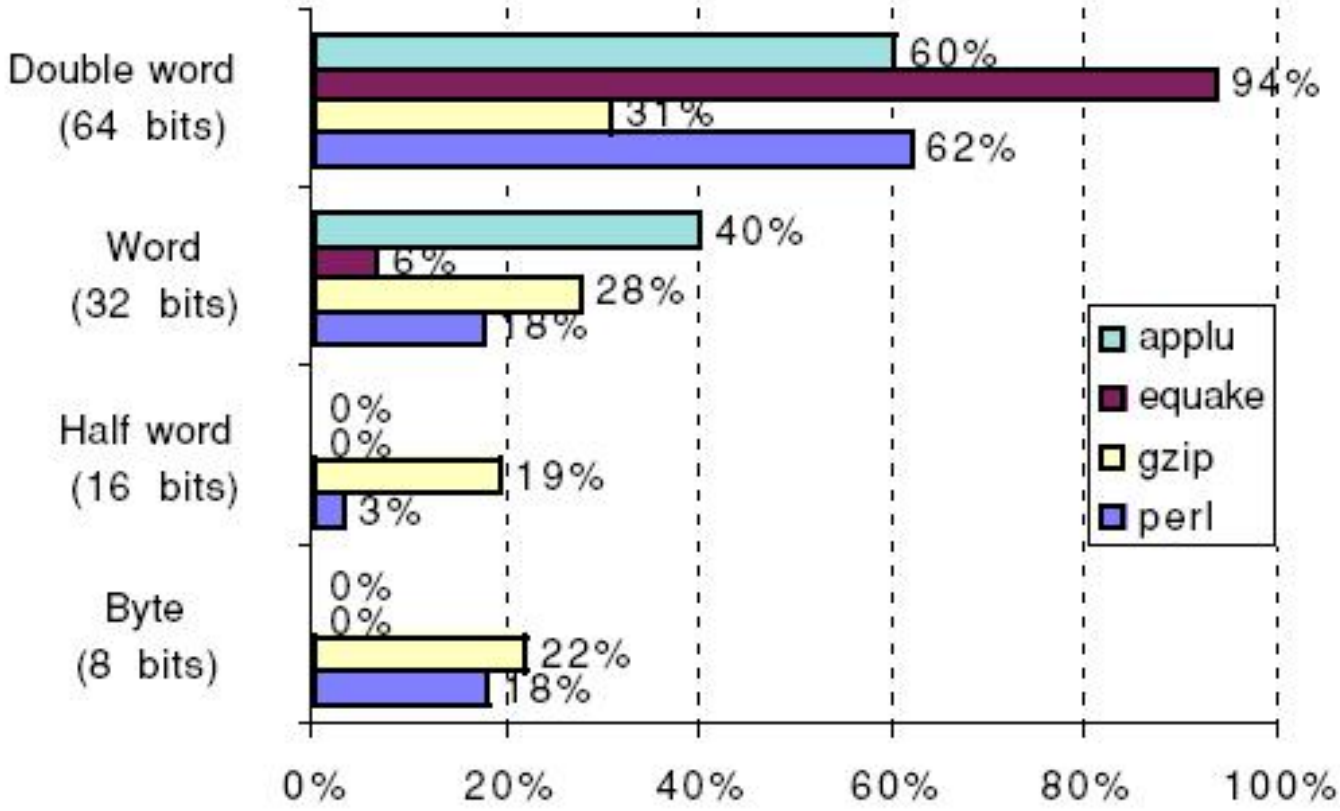


FIGURE 2.12 Distribution of data accesses by size for the benchmark programs. The double word data type is used for double-precision floating-point in floating-point programs and for addresses, since the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single word accesses.

访存对象

- 存储器按字节编址
  - 所有地址都是字节地址
  - 访问长度：字节、半字、字、双字
- 访存地址是否对齐（ Aligned vs. Misaligned ）
  - 地址对齐简化硬件设计：如字地址最低两位为0
  - 跨数据通路边界的访问可能需要访问两次RAM
  - 如何支持不对齐访问（如串操作）
- 大尾端（Big Endian）和小尾端（Little Endian）
  - Little Endian地址指向一个字的最右字节
  - Big Endian 反之

寻址方式

寻址方式	格式	含义
Register	ADD R1, R2	regs[R1]=reg[R1]+reg[R2]
Immediate	ADD R1, #2	regs[R1]=reg[R1]+2
Displacement	ADD R1, 100(R2)	regs[R1]=reg[R1]+mem[100+reg[R2]]
Reg. Indirect	ADD R1, (R2)	regs[R1]=reg[R1]+mem[reg[R2]]
Indexed	ADD R1, (R2+R3)	regs[R1]=reg[R1]+mem[reg[R2]+reg[R3]]
Absolute	ADD R1, (100)	regs[R1]=reg[R1]+mem[100]
Mem. Indirect	ADD R1, @(R2)	regs[R1]=reg[R1]+mem[mem[reg[R2]]]
Autoincrement	ADD R1, (R2)+	regs[R1]=reg[R1]+mem[reg[R2]], reg[R2]=reg[R2]+d
Autodecrement	ADD R1, -(R2)	reg[R2]=reg[R2]-d, regs[R1]=reg[R1]+mem[reg[R2]]
Scaled	ADD R1, 100(R2)[R3]	regs[R1]=reg[R1]+ mem[100+reg[R2]+reg[R3]*d]

常用寻址方式

- 三个程序在VAX机上的统计
  - 寄存器访问占一半，存储器访问占一半
  - 简单寻址方式占存储器访问的的97%

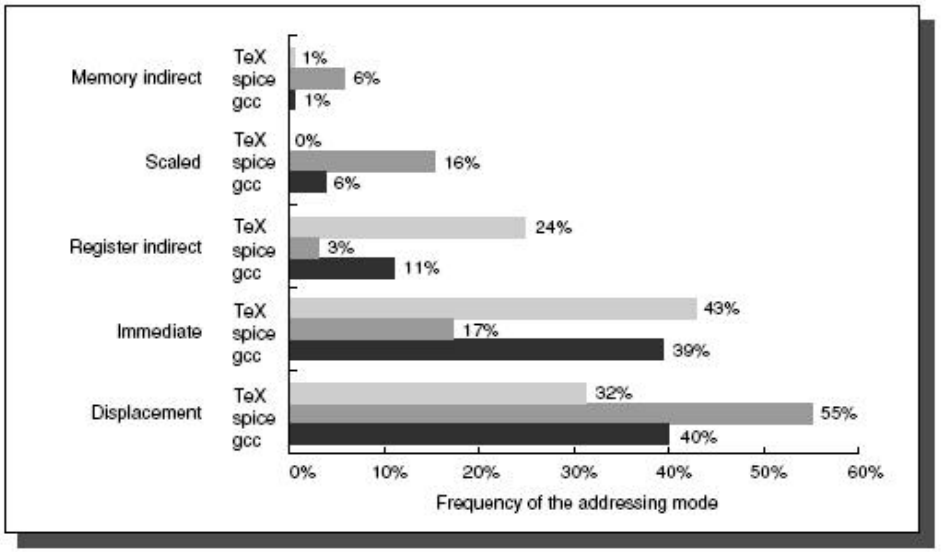


FIGURE 2.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see section 2.11. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Displacement mode includes all displacement lengths (8, 16, and 32 bit). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown. The data are from a VAX using three SPEC89 programs.

偏移量值的分布

- **SPEC CPU2000在Alpha结构（最大偏移为16位）上的统计**
  - **小偏移和大偏移较多，大偏移（14位以上）多数为负数**
  - **跟数据在内存中的分布有关**
- **现代虚拟机技术需要大偏移相对跳转指令**

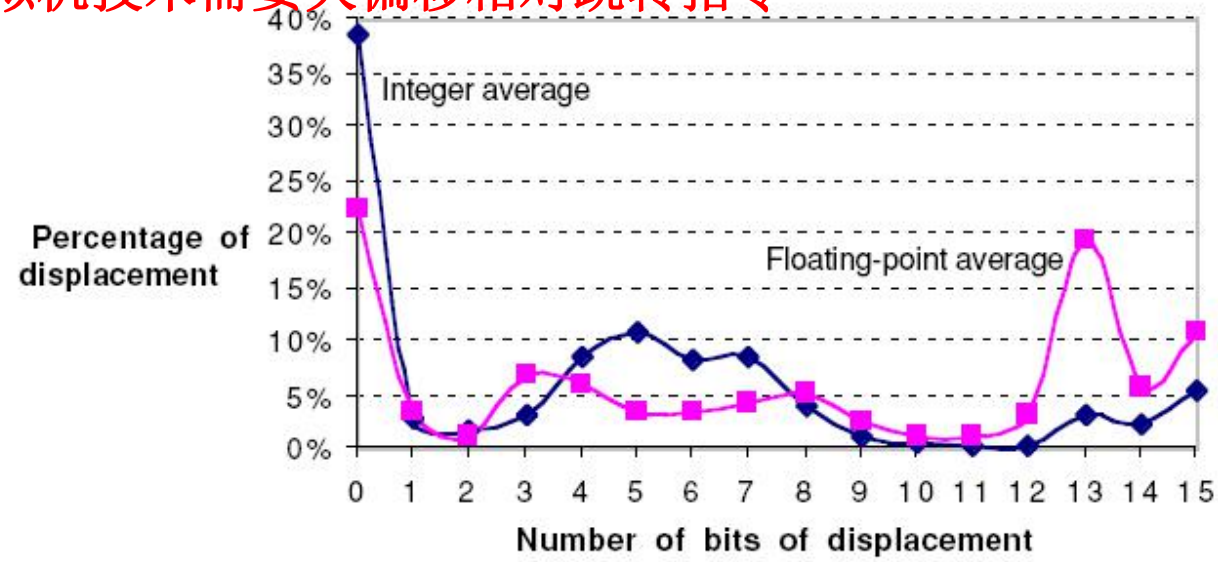


FIGURE 2.8 **Displacement values are widely distributed.** There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see section 2.11) as well as the overall addressing scheme the compiler uses. The x axis is $\log_2$ of the displacement; that is, the size of a field needed to represent the magnitude of the displacement. Zero on the x axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) is negative. Since this data was collected on a computer with 16-bit displacements, it cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization (see section 2.11) for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).

立即数的比例

- **SPEC CPU2000在Alpha结构上的统计**
  - **ALU操作定点1/4、浮点1/5需要立即数**
  - **Load操作有近1/4是取立即数（没有真正访存）**
  - **平均定点1/5、浮点1/6的指令需要立即数**

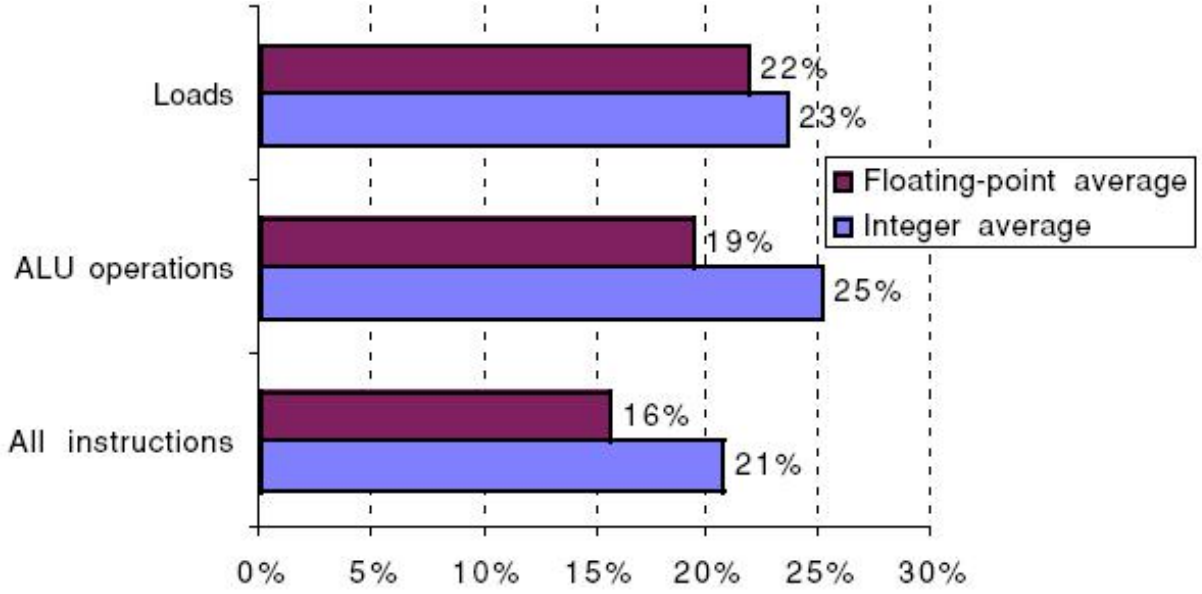


FIGURE 2.9 **About one-quarter of data transfers and ALU operations have an immediate operand.** The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) These measurements as in Figure 2.8.

立即数值的分布

- **SPEC CPU2000在Alpha结构上的统计**
  - **CINT2000中20%、CFP2000中30%的立即数是负的**
  - **在支持32位立即数的 VAX上统计表明，20%-30%立即数大于16位**

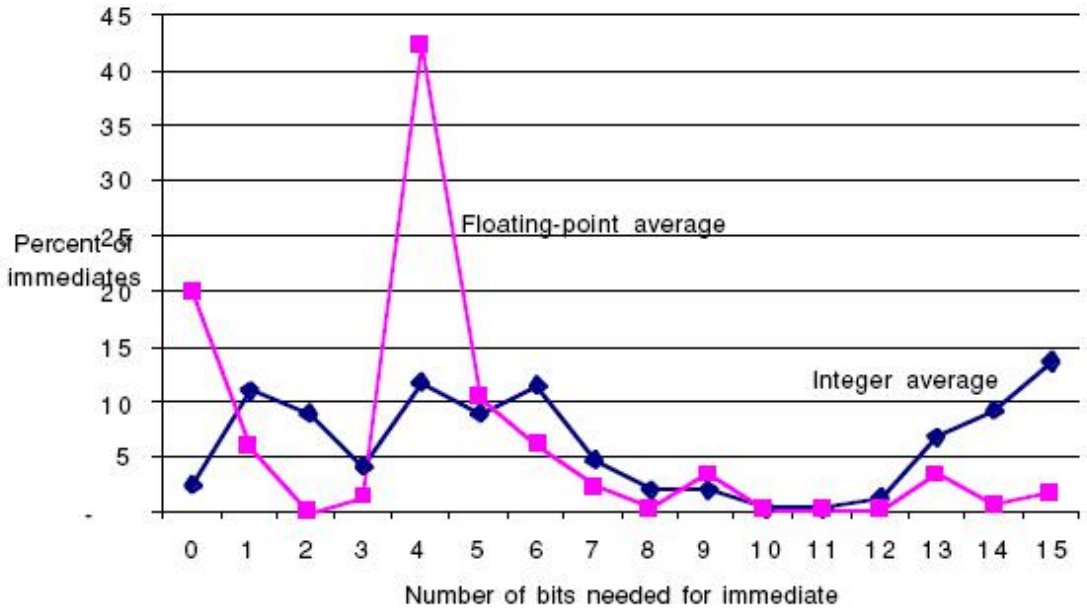


FIGURE 2.10 **The distribution of immediate values.** The *x* axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000 and about 30% were negative for CFP2000. These measurements were taken on a Alpha, where the maximum immediate is 16 bits, for the same programs as in Figure 2.8. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20% to 25% of immediates were longer than 16 bits.

寻址方式小结

- 至少支持以下寻址方式
  - **Register**
  - **Immediate**
  - **Displacement**
  - **Register indirect**
- 指令中常数位数
  - 地址偏移量位数**12-16**位
  - 立即数位数**8-16**位

指令系统组成

---指令操作和编码

指令操作

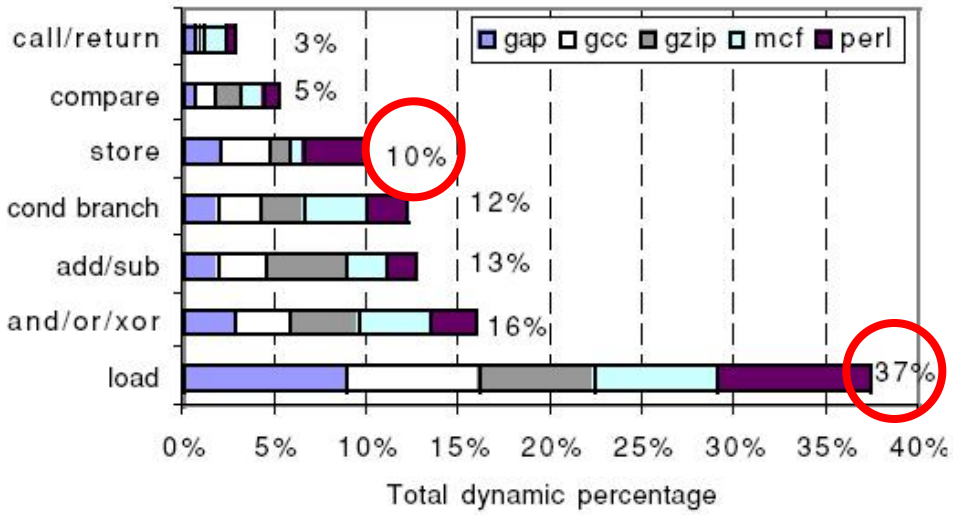
- 算术和逻辑运算指令
  - 加、减、乘、除、开方。。
  - 移位：左移与右移、逻辑移位与算术移位。。
  - 与、或、非、异或。。
  - 格式转换。。
- 访存指令：取数、存数
  - 不同长度和不同类型：定点/浮点，字节/半字/字/双字
  - 不同寻址方式
- 转移指令
  - 相对/绝对、直接/间接、条件/无条件
- 系统管理指令
  - TLB管理、Cache管理、异常管理、安全管理

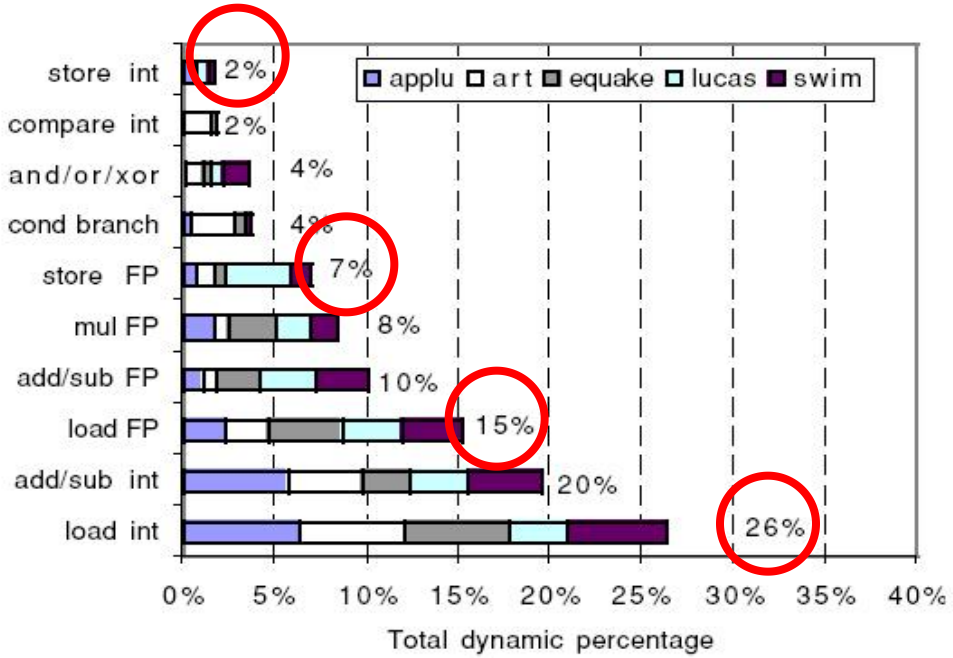
常见指令操作

- **最常用的指令是简单指令**
  - **SPECint92的X86指令统计**
  - **把这些简单指令做得快一点，其他慢一点没关系**

编号	指令	比例
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move reg-reg	4%
9	Call	1%
10	Return	1%
总计		96%

SPEC CPU2000动态指令分布





转移指令

- 转移指令类型
  - 条件转移/无条件转移
  - 过程调用/过程返回
- 转移地址类型
  - 相对：PC+偏移量
  - 绝对：指令中给出转移地址
  - 间接：根据寄存器内容转移（编译器不知道目标地址），如Switch语句、函数指针、动态链接、过程返回等

转移指令特点

- **SPEC CPU2000在Alpha结构上的统计**
  - 条件转移最多

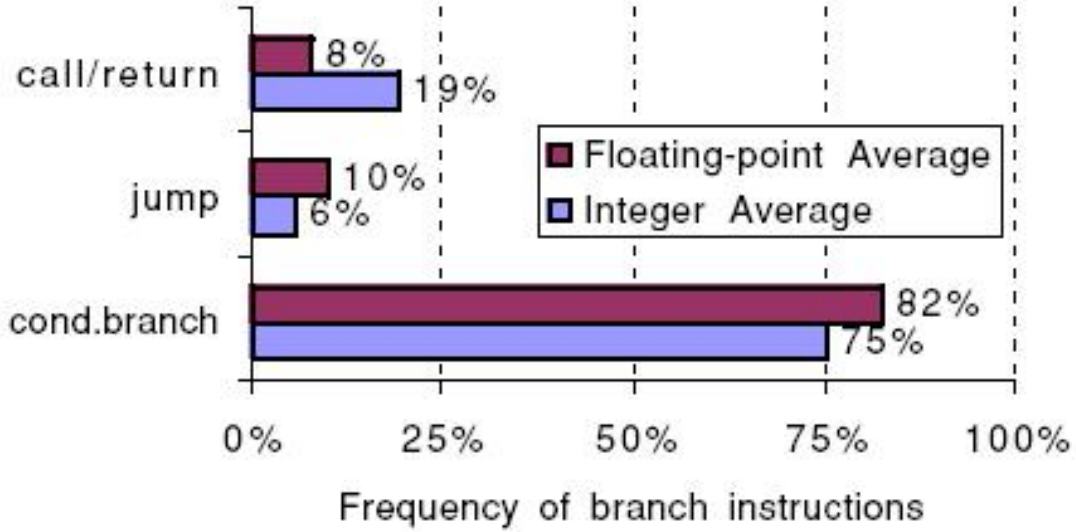


FIGURE 2.19 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure 2.8.

转移指令偏移量位数

- **SPEC CPU2000在Alpha结构上的统计**
  - **多数是4-8位**
- **虚拟机生成的代码放得较远，转移指令偏移量大**

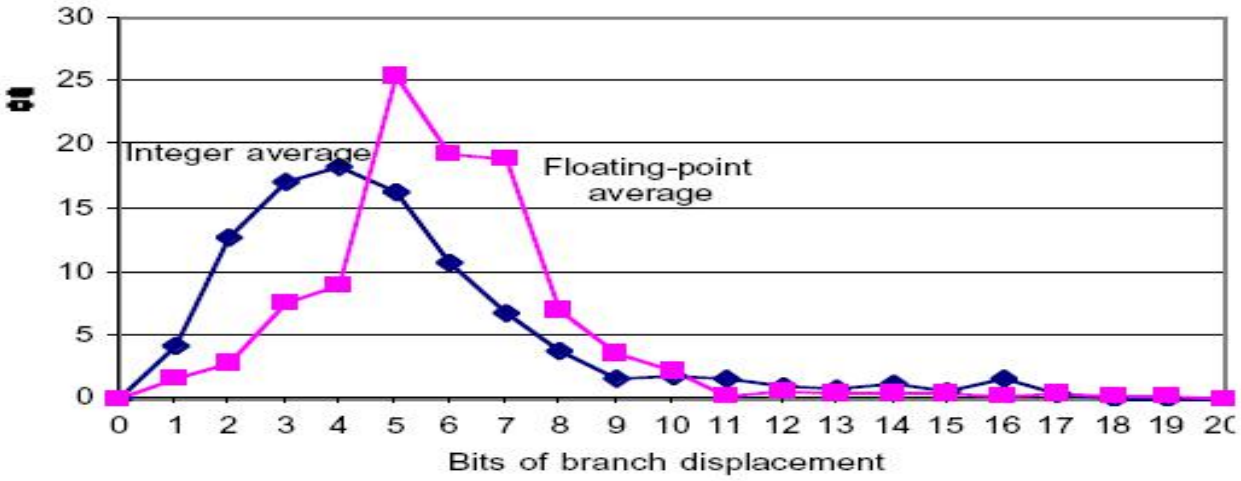


FIGURE 2.20 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in four to eight bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable length instructions to be aligned on any byte boundary. Exercise 2.1 shows the accumulative distribution of this branch displacement data (see Figure 2.42 on page 173). The programs and computer used to collect these statistics are the same as those in Figure 2.8.

转移条件的分布

- **SPEC CPU2000在Alpha结构上的统计**
  - 小于、等于、小于或等于比较最多

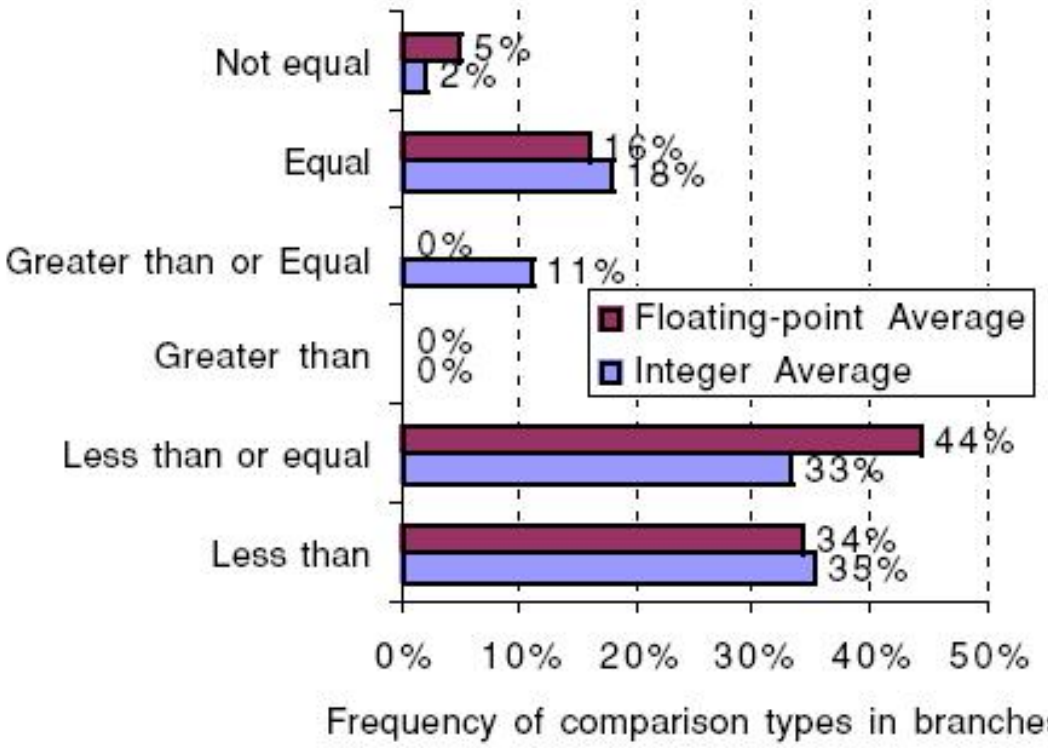


FIGURE 2.22 Frequency of different types of compares in conditional branches. Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure 2.8

转移条件的表达

- 根据条件位判断转移
- 直接比较寄存器内容转移

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

FIGURE 2.21 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

指令编码

- 需要考虑的因素
  - 操作码部分比较简单
  - 操作数的个数、类型对指令长度影响很大
  - 变长指令程序代码短、定长指令实现简单
- 编码方法
  - 定长：**RISC**
  - 变长：**VAX的指令1-53字节，其中ADD指令3-19字节，Intel的X86指令1-17字节**
  - 混合：**IBM 360/370，MIPS16，Thumb，TI TMS320C54x**

不定长、定长和混合指令编码

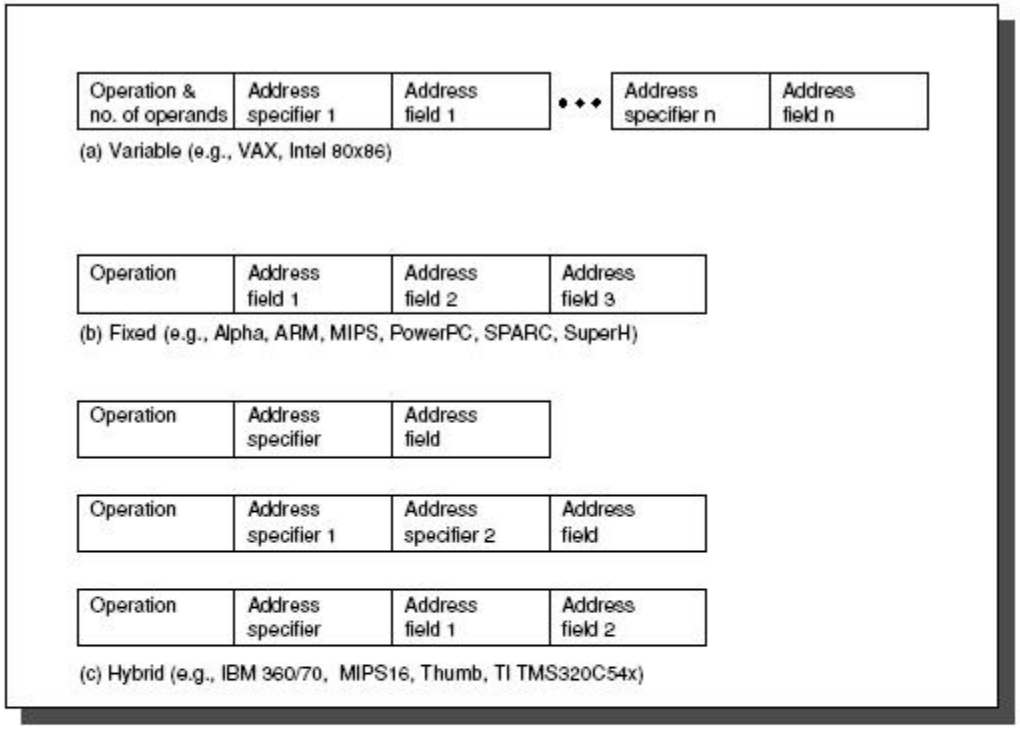


FIGURE 2.23 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode (see also Figure C.3 on page C-4). It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address (see also Figure D.7 on page D-12).

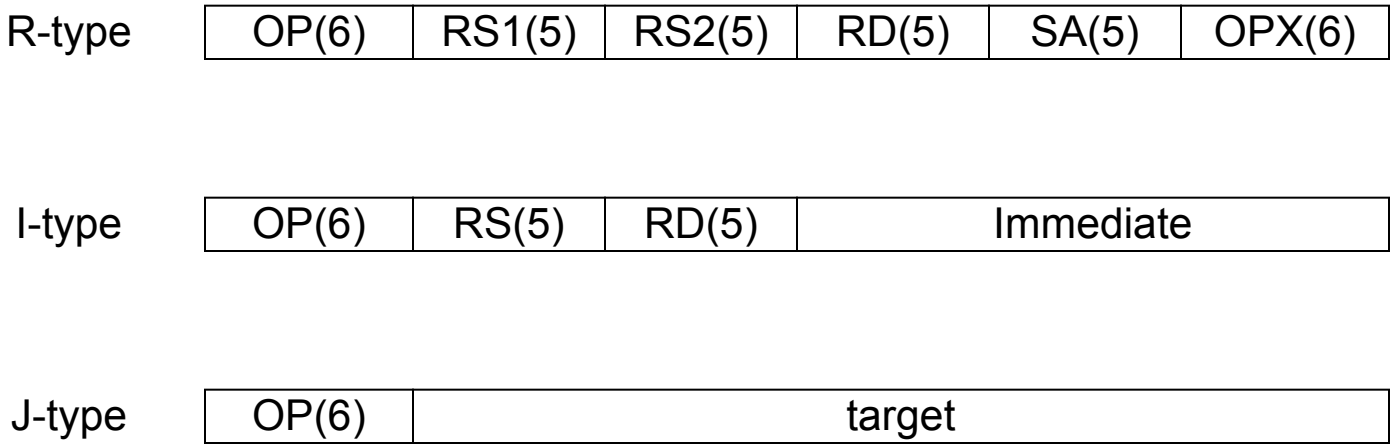
从上述分析可以看出

- 简单操作和简单寻址方式用得最多
  - 10种简单操作指令占96%
  - 寄存器, 立即数, 偏移寻址, 寄存器间接寻址四种寻址方式
- 简单指令便于高效实现和使用
  - load-store结构简化硬件设计，提高主频
  - 定长简化译码
  - 符合编译器“常用的做得快，少用的只要对”的原则
- 硬件优化应充分考虑兼容性
  - 流水、多发射不改变指令系统
  - 流水、多发射技术在load-store指令系统上容易实现
- 上述原因呼唤RISC系统结构
  - 简单是最复杂的创新

一个“典型”的RISC

- **32位定长指令**
- **32个32位通用寄存器**
- **三寄存器操作数运算指令**
- **Load-Store指令，基址+偏移量寻址方式**
- **简单转移条件**

MIPS指令格式


R-type
OP(6)
RS1(5)
RS2(5)
RD(5)
SA(5)
OPX(6)
I-type
OP(6)
RS(5)
RD(5)
Immediate
J-type
OP(6)
target

MIPS指令类型

- 访存指令（包括定点和浮点）
- 运算指令 （包括定点和浮点）
- 比较和转移指令（包括定点和浮点）
- 系统管理指令
  - **TLB、CACHE、例外处理**
  - **TRAP、Breakpoint**

RISC指令集比较

RISC发展历史

RISC发展过程（1）

- 1964年CDC公司推出的CDC 6600是第一台超级计算机，具备了RISC的一些基本特征
  - CDC 6600的设计者认识到为了实现有效的流水技术，需要简化体系结构：Load-Store结构
  - 记分板（Score-Boarding）动态流水线调度
  - 乱序执行（Out-of-Order）技术
- 1976年的Cray-1向量机使用了与CDC 6600类似想法
  - Cray是CDC 6600的主要设计者之一
- 上述简化结构以高效实现的想法在60-70年代没有受到小型机和微处理器设计者的重视

RISC发展过程（2）

- 1968年John Cocke在IBM的San Jose研究中心开始ASC（Advanced Scientific Computer）项目的研究
  - 基本思想是让编译器做更多的指令调度以减少硬件复杂度
  - 还提出了每个周期发射多条指令的思想
  - ASC计划后来被取消，Cocke在1971年到Future System
- 1975年Cocke到IBM的Yorktown研究中心开始研制IBM 801，801是最早开始设计的RISC处理器
  - Cocke获得了Eckert-Mauchly和Turing奖
  - 801是PowerPC的前身
- 比801稍晚开始的有Patterson在Berkeley的RISC-I及RISC-II与Hennessy在Standford的MIPS项目
  - 这两个大学的研究生曾参与801项目的研究，后来返回大学
  - RISC-II是SPARC的前身，MIPS项目是MIPS处理器前身

RISC发展过程（3）

- 801的项目经理Joel Birnbaum到HP创立了PA-RISC
- DEC在推出Alpha之前曾经使用MIPS处理器三年
- 从上述发展过程不难解释刚开始时五个RISC处理器的相似性
- 后来每个RISC处理器有了不同的发展
  - 如Alpha的指令简单，容易高主频实现，“a speed demon”
  - PowerPC指令功能强，灵活，架构上追求ILP ，“a brainiac ”
- 最近Berkeley推出的RISC-V
  - Berkeley在RISC-I/II演变成SPARC后又发展了SOAR、SPUR
  - RISC-V是Berkeley的第五代指令系统，V还代表Victory
- 现代X86处理器内部的核心也是RISC结构

RISC指令系统的发展

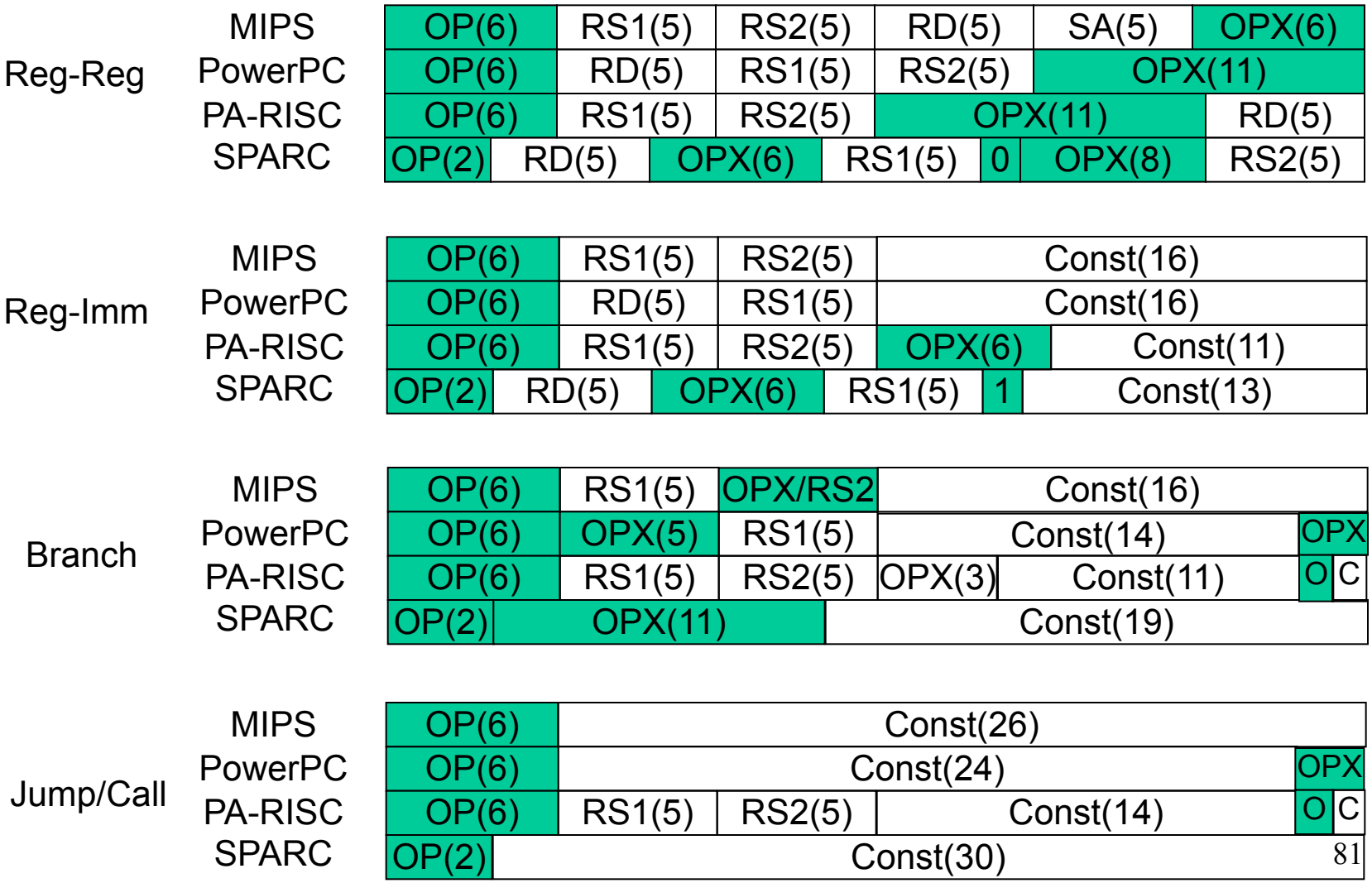
	MIPS	ALPHA	PA-RISC	SPARC	PowerPC
1986	MIPS I		PA-RISC 1.0		RT/PC
1987				SPARC v8	
1988					
1989	MIPS II				
1990			PA-RISC 1.1		Power 1
1991					
1992	MIPS III (64b)	Alpha (64b)			
1993					Power 2& Power PC
1994	MIPS IV (64b)			SPARC v9 (64b)	
1995					Power PC (64b)
1996			PA-RISC 2.0 (64b)		

常见RISC指令系统的比较

常见RISC指令系统比较

- 通过比较常见**RISC**处理器的指令加深对**RISC**的了解
  - **MIPS、PA-RISC、PowerPC、SPARC**
- 通过以下方面进行比较
  - 指令格式
  - 寻址方式
  - 指令功能

指令格式比较


Reg-Reg
MIPS
PowerPC
PA-RISC
SPARC
OP(6) RS1(5) RS2(5) RD(5) SA(5) OPX(6)
OP(6) RD(5) RS1(5) RS2(5) OPX(11)
OP(6) RS1(5) RS2(5) OPX(11) RD(5)
OP(2) RD(5) OPX(6) RS1(5) 0 OPX(8) RS2(5)
Reg-Imm
MIPS
PowerPC
PA-RISC
SPARC
OP(6) RS1(5) RS2(5) Const(16)
OP(6) RD(5) RS1(5) Const(16)
OP(6) RS1(5) RS2(5) OPX(6) Const(11)
OP(2) RD(5) OPX(6) RS1(5) 1 Const(13)
Branch
MIPS
PowerPC
PA-RISC
SPARC
OP(6) RS1(5) OPX/RS2 Const(16)
OP(6) OPX(5) RS1(5) Const(14) OPX
OP(6) RS1(5) RS2(5) OPX(3) Const(11) O C
OP(2) OPX(11) Const(19)
Jump/Call
MIPS
PowerPC
PA-RISC
SPARC
OP(6) Const(26)
OP(6) Const(24) OPX
OP(6) RS1(5) RS2(5) Const(14) O C
OP(2) Const(30)

寻址方式比较

寻址方式	MIPS IV	PA-RISC 1.0	PowerPC	SPARC v9
Register	√	√	√	√
Imm.	√	√	√	√
Disp. (reg+offser)	√	√	√	√
Indexed(reg+reg)	√ (FP)	√	√	√
Scaled(reg+scaled reg)		√		
(reg+offset+update reg)		√	√	
(reg+reg+update reg)		√	√	

指令功能比较

- 所有RISC处理器都有一些公共指令
  - load/store指令
  - 算术运算及逻辑指令
  - 控制流指令
  - 系统管理指令
- 不同处理器在发展过程中形成的特色举例
  - MIPS的非对齐访问
  - SPARC的寄存器窗口
  - PowerPC的Link和Count寄存器
  - HP的Nullification
  - ……

Load/Store指令

- 对任何GPR和FPR进行存取操作
  - 通常R0总是为0
- MIPS的例子

OpCode	Description
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right

ALU指令

- 所有ALU指令都是寄存器型的
- ALU的常见操作有加、减、与、或、异或、移位、比较，乘除法在专门的部件进行
- MIPS的例子

OpCode	Description
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive OR Immediate
LUI	Load Upper Immediate

OpCode	Description
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive OR
NOR	NOR

OpCode	Description
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move To HI
MFLO	Move From LO
MTLO	Move To LO

控制流指令

- **绝对跳转jump和相对转移branch**
- **MIPS的例子**

OpCode	Description
J	Jump
JAL	Jump And Link
JR	Jump Register
JALR	Jump And Link Register
BEQ(L)	Branch on Equal (Likely)
BNE	Branch on Not Equal
BLEZ	Branch on Less Than or Equal to Zero
BGTZ	Branch on Greater Than Zero
BLTZ	Branch on Less Than Zero
BGEZ	Branch on Greater Than or Equal to Zero
BLTZAL	Branch on Less Than Zero And Link
BGEZAL	Branch on Greater Than or Equal to Zero And Link

条件转移的条件判断

- SPARC v8使用4位条件码(CC)，该条件码在程序状态字中
  - 整数运算指令设置CC，条件转移指令检测CC
  - 浮点运算有另外两位CC
  - v9为了支持64位运算增加了4位整数CC，3位浮点CC
- MIPS直接比较寄存器内容判断是否转移
  - MIPS III浮点部件有一位条件码，记录cmp指令的结果
  - MIPS IV有多位浮点条件码
- PowerPC有4位CC，一个条件寄存器中有8份4位CC
  - 整数和浮点运算各1位，其它用于比较指令。
  - Branch指令需指定根据哪一位进行转移
  - 运算指令中有一位指定该指令是否影响CC
- PA-RISC有多种选择，最常用的是比较两个寄存器的值并根据结果决定是否转移

系统管理指令

- 原子操作指令
- 存储管理指令
- 例外管理指令
- 共享存储同步指令
- 等等

原子操作指令

- 以MIPS的LL和SC指令 为例
  - LL(Load Linked)取数且置系统中LLbit为1
  - LL为1时，处理器检查相应单元是否被修改，如果其它处理器或设备访问了相应单元或执行了ERET操作，LLbit置为0
  - 执行SC（Store Conditional）时若LLbit为1，则成功，目标寄存器为1；否则存数不成功，目标寄存器为0

```
L1:  LL    R1, (R3)
     ADD   R2, R1, 1
     SC    R2, (R3)
     BEQ   R2, 0, L1
     NOP
```

MIPS特色—非对齐访存指令

- 边界不对齐的数据传送（小尾端）

初始值

	3	2	1	0
M[100]	A	V	E	
M[104]				D
R2	J	O	H	N

初始值

	3	2	1	0
M[200]	E			
M[204]		D	A	V
R2	J	O	H	N

执行“LWR R2, 0x101”后

R2	J	A	V	E

执行“LWR R2, 0x203”后

R2	J	O	H	E

执行“LWL R2, 0x104”后

R2	D	A	V	E

执行“LWL R2, 0x206”后

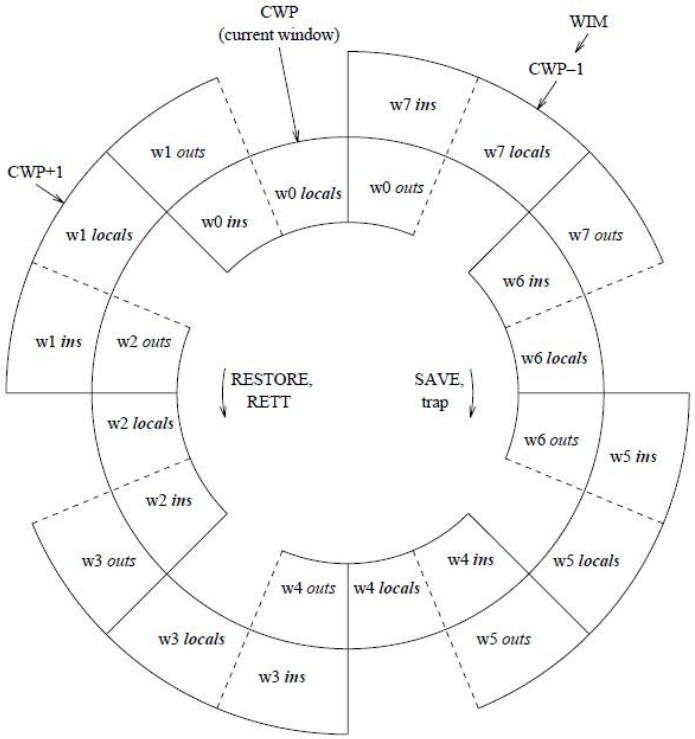
R2	D	A	V	E

LWL/LWR指令举例

- 如果没有它们,取一个不对齐的字需要10条指令
  - 包括4个load/store并需要一个临时寄存器:

```
lbu   rt,3(base)
sll   rt,rt,24
lbu   rtmp,2(base)
sll   rtmp,rtmp,16
or    rt,rt,rtmp
lbu   rtmp,1(base)
sll   rtmp,rtmp,8
or    rt,rt,rtmp
lbu   rtmp,rtmp,0(base)
or    rt,rt,rtmp
```

SPARC特色—寄存器窗口



- **2-32个寄存器窗口，用于不同过程**
  - **8个全局寄存器用于存放全局变量**
  - **24个局部寄存器**
    - **8个输入参数，8个局部变量，8个输出参数**
  - **过程调用和退出不用把现场保留到存储器**
- **SAVE和RESTORE指令**
  - **SAVE：功能同ADD，源寄存器来自调用过程(caller)，目标寄存器来自被调用过程(callee)，该指令自动修改寄存器窗口指针CWP--**
  - **RESTORE：功能同ADD，源寄存器来自被调用过程(callee)，目标寄存器来自调用过程(caller)，该指令自动修改寄存器窗口指针CWP++**
- **AMD AM29000的局部寄存器窗口大小可变，全局寄存器64个**

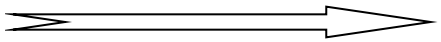
PowerPC特色－Link和Count寄存器

- **Link寄存器用于保存返回地址，实现快速过程调用**
- **Count寄存器用于循环计数，每次自动递减**
- **这两个寄存器还可以放转移地址**
- **PowerPC不用Delay Slot**
- **其它特色**
  - **Load和Store指令同时存取多个寄存器（多达32个）**
  - **Load和Store字符串（变长或定长、对齐或不对齐）**

PA-RISC的Nullification

- 根据当前指令执行结果确定下一条指令是否执行
  - 所有的转移指令和多数算术指令可用
  - 如ADDB（Add and branch）指令在完成加法后，检查加法结果是否满足条件，如果不满足就转移。同时判断下一条指令（延迟槽）是否执行
  - 可以消除一些简单的转移指令

```
if (a>b) s=a;
else     s=b;
```



```
ADD     %r1, %r0, %r3 // r3 = r1(a)
SUB,*>  %r1, %r2, %r0 // r1(a) – r2(b)
ADD     %r2, %r0, %r3 // r3 = r2(b)
```

  - 其它指令系统有条件移数指令如“CMOV”
- 后来演化为IA64的谓词技术
  - 运算指令通过64个谓词寄存器决定结果是否保存

Alpha和PowerPC指令功能举例

源代码：for (k=0; k<512; k++) x[k] = r * x[k] + t * y[k];

PowerPC代码	Alpha代码

r3+8指向x
r4+8指向y
fp1内容为t
fp3内容为r
CTR内容为512
LOOP:
LFU fp0=y(r4=r4+8) //FP load with update
FMUL fp0=fp0, fp1 //FP multiply
LF fp2=x(r3,8) //FP load
FMADD fp0=fp0,fp2,fp3 //FP multiply-add
STFU x(r3=r3+8)=fp0 //FP load with update
BC LOOP, CTR>0 //decrement CTR, branch if>0
	r1指向x
	r2指向y,
	r6指向y的末尾
	fp2内容为t
	fp4内容为r
	LOOP:
	LDT fp3=y(r2,0)
	LDT fp1=x(r1,0)
	MULT fp3=fp3,fp2 // t*y
	ADDQ r2=r2,8
	MULT fp1=fp1,fp4 // r*x
	SUBQ r4=r2,r6
	ADDT fp1=fp3,fp1 //t*y+r*x
	STT x(r1,0)=fp1
	ADDQ r1=r1,8
	BNE r4, LOOP

Alpha和PowerPC比较

- 从这个例子可以看出
  - **PowerPC的load-with-update和store-with-update指令适合于数组运算，Alpha没有寄存器加寄存器的寻址方式，指向数组的指针每次分别递增**
  - **在Alpha中，循环次数由指针实现，在PowerPC中，有CTR专门用于保存循环次数**
  - **PowerPC只需两条浮点指令：乘及乘加**
  - **Alpha需要10条指令，比PowerPC多4条**
  - **Alpha指令简单容易高效实现，主频也高**

RISC指令系统小结

	PA-RISC			SPARC		MIPS				Power		
	1.0	1.1	2.0	v8	v9	I	II	III	IV	1	2	PC
Interlocked loads	√	√	√	√	√		√	√	√	√	√	√
Load/store FP double	√	√	√	√	√		√	√	√	√	√	√
Semaphore	√	√	√	√	√		√	√	√	√	√	√
Square root	√	√	√	√	√		√	√	√		√	√
Single-precision FP ops	√	√	√	√	√	√	√	√	√			√
Memory synchronization	√	√	√	√	√		√	√	√	√	√	√
Coprocessor	√	√	√	√		√	√	√	√			
Base+index addressing	√	√	√	√	√				√	√	√	√
32 64-bit FP registers		√	√		√			√	√	√	√	√
Annulling delayed branch	√	√	√	√	√		√	√	√			
Branch register contents	√	√	√		√	√	√	√	√			
Big or little Endian		√	√		√	√	√	√	√			√
Branch prediction bit					√		√	√	√	√	√	√
Conditional move					√				√	√	√	
Prefetch data into cache			√		√				√	√	√	√
64-bit addressing/int. op			√		√			√	√			√
32-bit multiply, divide		√	√		√	√	√	√	√	√	√	√
Load/store FP quad					√						√	
Fused FP mul/add			√						√	√	√	√
String instruction	√	√	√							√	√	

- 对于一个程序在上述四种RISC结构上的执行，平均90%以上的指令为四个指令系统共有的

C语言的机器表示

过程调用

- 封装好的一段代码序列
- **JAL/JALR/BAL用于调用，JR用于返回**
  - **“AL”=and Link，将下一条指令地址（不含延迟槽）自动存入$31(RA)寄存器中**
  - **JR的操作数通常为RA，也可以是其他寄存器**
- 过程调用流程
  1. **调用者（Caller）将实参放入寄存器或栈中**
  2. **使用调用指令调用被调用者（Callee）**
  3. **Callee在栈中分配自己所需的局部变量空间**
  4. **执行callee过程**
  5. **Callee释放局部变量空间（将栈指针还原）**
  6. **Callee使用JR返回调用者**

过程调用 – 示例

- **MIPS过程调用的寄存器约定**
  - **$4-$7（记为a0-a3）作为参数输入**
  - **$2、$3（记为v0、v1）作为返回值**
  - **$8-$15（记为t0-t7）作为子程序暂存器**
  - **$28(记为sp）作为栈指针寄存器**

```
int add(int a,int b)
{
return a+b;
}
int ref()
{
int t1 = 12;
int t2 = 34;
return add(t1,t2);
}
```

```
add:
  jr    ra
  addu  v0, a0, a1
ref:
  addiu sp, sp, -32
  sw    ra, 28(sp)
  li    a0, 12
  jal   add
  li    a1, 34
  lw    ra, 28(sp)
  nop
  jr    ra
  addiu sp, sp, 32
```

依据MIPS函数调用规范保存RA到SP+28

流程控制语句

- C语言的控制流语句
  - 选择语句：if~else， switch~case
  - 循环语句：for，while，do~while
  - 辅助控制语句：break，continue，goto，return
- C语言的控制流语句在MIPS汇编中映射为各种分支指令（B类）或其组合
  - 选择语句和循环语句映射为条件分支
  - 辅助控制语句映射为无条件分支或JR（return语句）

流程控制语句 – 示例1

- 当条件表达式t0等于0（BEQZ）时，跳转到标号1
- 标号2为公用的程序出口
- 本例的延迟槽使用空指令（NOP）进行填充

```
// C if ~ else
if (cond_exp)
  <then_statement>
else
  <else_statement>
```

```
## ASM if ~ else
    move t0, cond_exp
    beqz t0, 1f          # label1 forward
    nop                  # delay slot
    <then_statement>
    b    2f              # label2 forward
    nop                  # delay slot
1:
    <else_statement>
2:
```

流程控制语句 – 示例2

- **switch-case采用一串比较实现**

```
int st(int a,int b,int c)
{
  switch(a){
    case 15:
      c=b&0x0f;
    case 10:
      return c+50;
    case 12:
    case 17:
      return b+50;
    case 14:
      return b;
    default:
      return a;
  }
}
```

```
st:
    li   t0, 15
    beq  t0, a0, 1f
    li   t0, 10
    beq  t0, a0, 2f
    li   t0, 12
    beq  t0, a0, 3f
    li   t0, 17
    beq  t0, a0, 3f
    li   t0, 14
    beq  t0, a0, 4f
    nop
    jr   ra
    move v0, a0
1:  andi a2, a1, 0x0f
2:  jr   ra
    addi v0, a2, 50
3:  jr   ra
    addi v0, a1, 50
4:  jr ra
    move v0, a1
```

流程控制语句 – 示例3

- **switch-case采用跳转表实现**

```
int st(int a,int b,int c)
{
  switch(a){
    case 15:
      c=b&0x0f;
    case 10:
      return c+50;
    case 12:
    case 17:
      return b+50;
    case 14:
      return b;
    default:
      return a;
  }
}
```

```
st:
    addiu   v0, a0, -10
    sltiu   v1, v0, 8
    beqz    v1, default
    li      v1, jr_table
    sll     v0, v0, 0x2
    addiu   v1, v1, v0
    lw      v0, 0(v1)
    jr      v0
    nop
case_15:
    andi    a2, a1, 0x0f
case_10:
    jr      ra
    addiu   v0, a2, 50
case_12o17:
    jr      ra
    addiu   v0, a1, 50
case_14:
    jr      ra
    addiu   v0, a1, 0
default:
    jr      ra
    addiu   v0, a0, 0
```

```
jr_table:
00: case_10
04: default
08: case_12o17
0c: default
10: case_14
14: case_15
18: default
1c: case_12o17
```

流程控制语句 – 示例4

// C for
for(i=0; i<8; i++)
  sum += i;
	## ASM for
	  move t0, $0
	  li t1, 8
	1:
	  add t2, t0, t2
	  addi t0, t0, 1
	  bne t0, t1, 1b # label1 backward
	  nop

// C while
i = 0;
while (i < 8)
  i++;
	## ASM while
	  move t0, $0
	  li t1, 8
	1:
	  bge t0, t1, 2f
	  nop
	  b 1b
	  addi t0, t0, 1 # delay slot
	2:

// C do ~ while
i = 0;
do {
  i++;
}while (i < 8)
	## ASM do ~ while
	  move t0, $0
	  li t1, 8
	1:
	  blt t0, t1, 1b
	  addi t0, t0, 1 # delay slot

作 业