计算机体系结构

周学海

xhzhou@ustc.edu.cn

0551-63492271,0551-63601556

中国科学技术大学计算机系

03-06-13-Review

- 设计发展趋势

	Capacity	Speed
Logic	2x in 3 years	2x in 3 years
DRAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years

- 运行任务的时间
  - Execution time, response time, latency
- 单位时间内完成的任务数
  - Throughput, bandwidth
- “X性能是Y的n倍”：

$$\frac{\text{ExTime(Y)}}{\text{ExTime(X)}} = \frac{\text{Performance(X)}}{\text{Performance(Y)}}$$

- **Amdahl's 定律:**

$$\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- **CPI Law:**

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

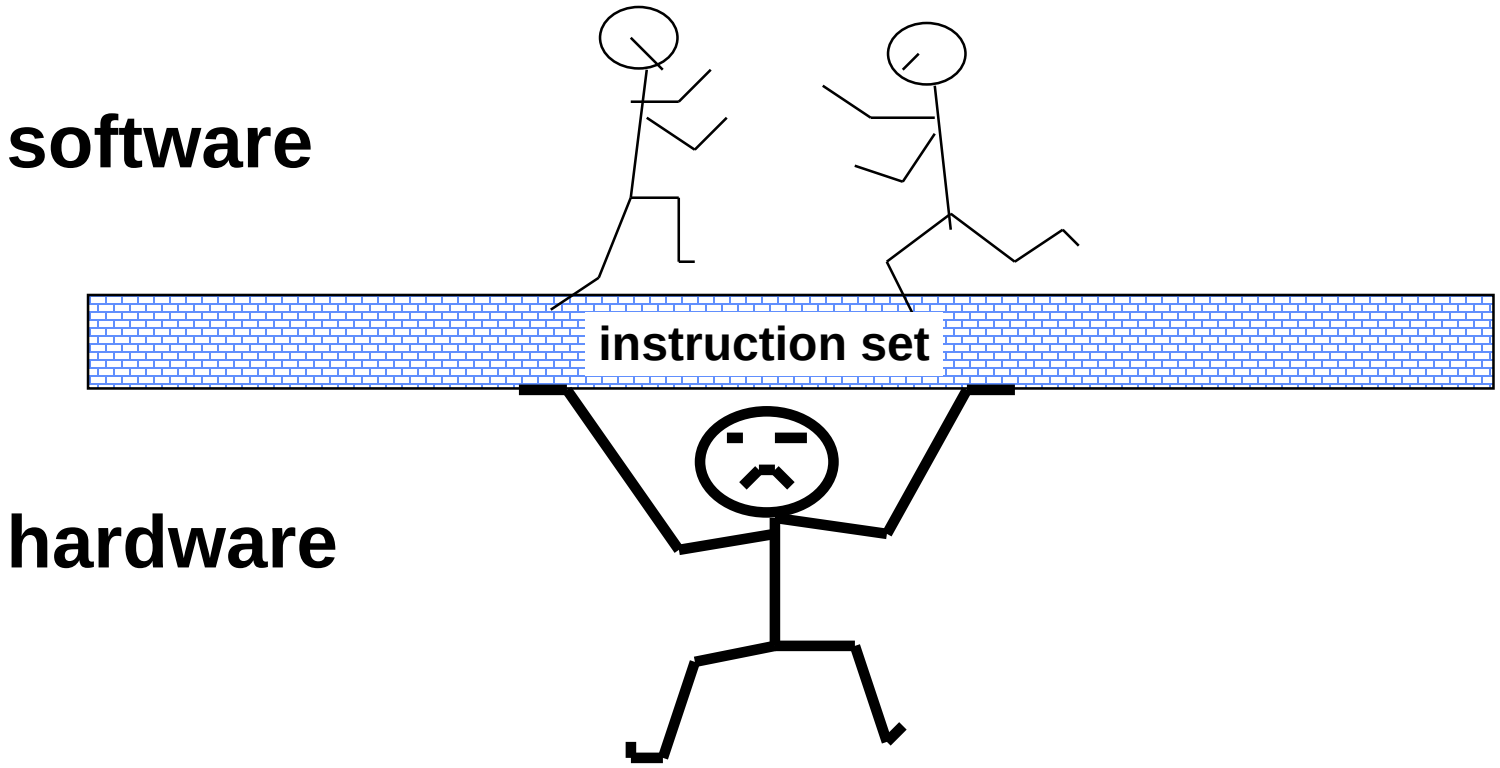
- 执行时间是计算机系统度量的最实际，最可靠的方式

第2章 指令集结构设计

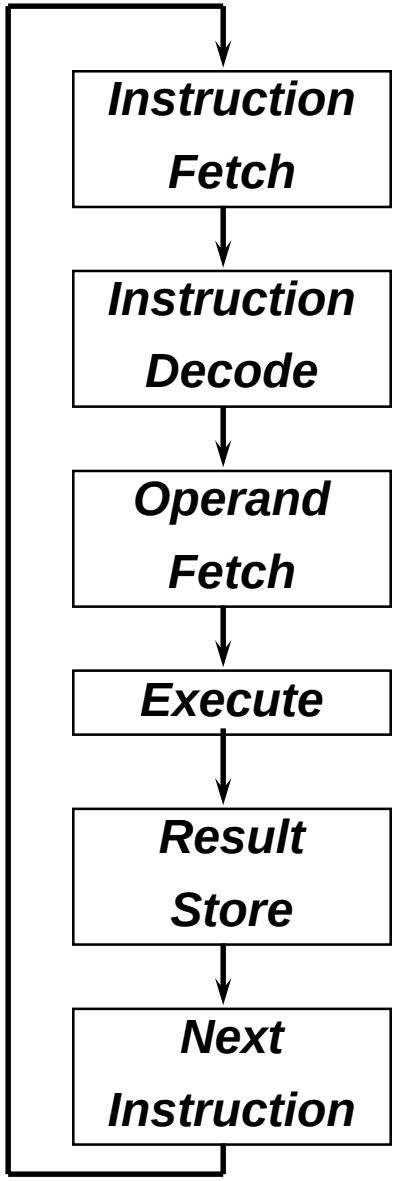
- ✓ ISA的分类
- ✓ 寻址方式
- ✓ 操作数的类型、表示和大小
- ✓ ISA的操作
- ✓ 控制类指令
- ✓ 指令编码

- ✓ 编译技术与计算机体系结构
- ✓ MIPS指令集

指令集设计



指令集结构：我们必须说明哪些东西?

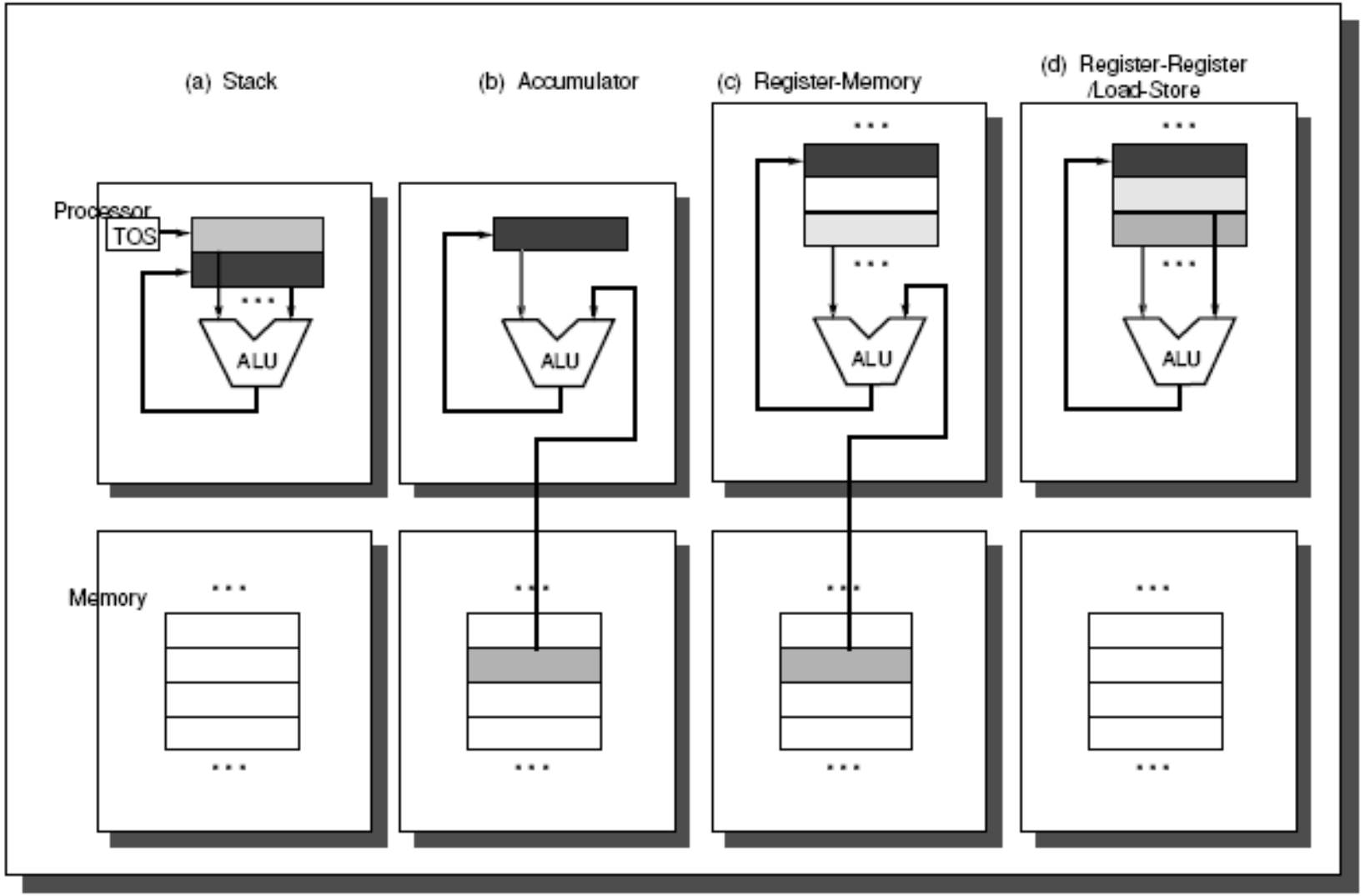


- 指令格式或编码方式。即如何编码?
- 操作数和操作结果的存放位置
  - 存放位置?
  - 多少个显式操作数?
  - 存储器操作数如何定位?
  - 哪些操作数可以或不可以放到存储器中?
- 数据类型和大小
- 寻址方式
- 支持哪些操作
- 下一条指令地址
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*

有关ISA的若干问题

- Class of ISA
- Memory addressing
- Types and sizes of operands
- Operations
- Control flow instructions
- Encoding an ISA

ISA 的基本分类


(a) Stack
(b) Accumulator
(c) Register-Memory
(d) Register-Register
/Load-Store
Processor
TOS
ALU
ALU
ALU
ALU
Memory

ISA 的基本分类

- 累加器型(Accumulator) (1 register):

  1 address `add A` acc ← acc + mem[A]

  1+x address `addx A` acc ← acc + mem[A + x]

- 堆栈型（Stack）:

  0 address `add` tos ← tos + next

- 通用寄存器型(General Purpose Register):
  - **Register-memory**

  2 address `add A B` EA[A] ← EA[A] + EA[B]

  3 address `add A B C` EA[A] ← EA[B] + EA[C]

  - **Load/Store:**

  3 address `add Ra Rb Rc` Ra ← Rb + Rc

  `load Ra Rb` Ra ← mem[Rb]

  `store Ra Rb` mem[Rb] ← Ra

- 存储器-存储器型 （目前已经没有）

比较指令条数

计算 C = A+B

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

通用寄存器型占主导地位

- **1980**以后至今几乎所有的机器都用通用寄存器结构
- 原因
  - 寄存器比存储器快
  - 对编译器而言寄存器更容易使用

    (A*B)-(B*C)-(A*D)
    - » 寄存器可以存放变量
    - » 代码紧凑

通用寄存器的分类

- 分类原则：
  - ALU指令到底是两地址指令还是三地址指令
  - ALU指令中有多少个操作数可以用存储器寻址，即有多少个存储器操作数

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

常见的通用寄存器型指令集结构的优缺点

Type	Advantages	Disadvantages
Register–register (0,3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute.	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register–memory (1,2)	Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. CPI varies by operand location.
Memory –memory (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

寻址技术

如何解释存储器地址？如何说明寻址方式？

- **1980**年以来几乎所有机器的存储器都是按字节编址的
- **ISA**设计要关注两个问题：

  8bits－字节， **16bits-**半字，**32bits** －字， **64bits** －双字

  如何读**32**位字，两种方案

  – 每次一个字节，四次完成

  – 每次一个字，一次完成

  问题：

  （**1**）如何将字节地址映射到字地址 （尾端问题）

  （**2**）一个字是否可以存放在任何字节边界上 （对齐问题）

  即尾端（**Endian)**和对齐问题

尾端问题

- **little endian, big endian, 在一个字内部的字节顺序问题，如地址xxx00指定了一个字（int），存储器中从xxx00处连续存放ffff0000, 则有两种方式：**
  - **Little endian 方式下xxx00位置是字的最低字节，整数值为0000ffff, Intel 80x86, DEC Vax, DEC Alpha (Windows NT)**
  - **Big endian 方式下xxx00位置是字的最高字节，整数值为ffff0000, IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA**

对齐问题

- 对一个s字节的对象访问，地址为A，如果A mod s =0 那么它就是边界对齐的。
- 边界对齐的原因是存储器本身读写的要求，存储器本身读写通常就是边界对齐的，对于不是边界对齐的对象的访问可能要导致存储器的两次访问，然后再拼接出所需要的数。（或发生异常）

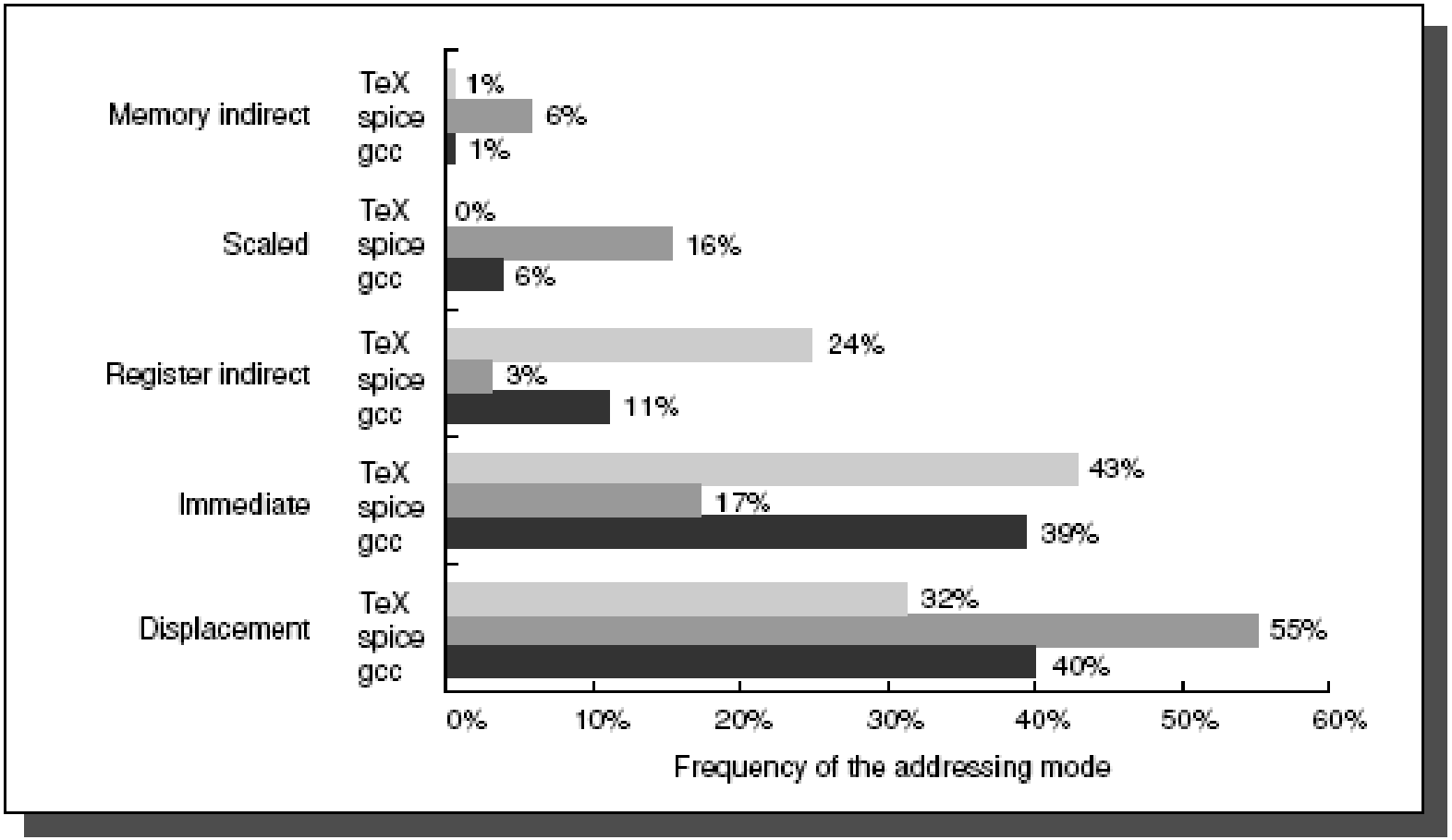
寻址方式

- 寻址方式：如何说明要访问的对象地址
- 有效地址：由寻址方式说明的某一存储单元的实际存储器地址。
- 有效地址 **vs.** 物理地址

寻址方式

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4] ← Regs[R4] + Regs[R3]	When a value is in a register.
Immediate	Add R4,#3	Regs[R4] ← Regs[R4] + 3	For constants.
Displacement	Add R4,100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1 + R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	Regs[R1] ← Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer p, then mode yields $*p$.
Autoincrement	Add R1,(R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + d	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.
Autodecrement	Add R1,−(R2)	Regs[R2] ← Regs[R2] − d Regs[R1] ← Regs[R1] + Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]	Used to index arrays. May be applied to any indexed addressing mode in some computers.

各种寻址方式的使用情况？（忽略寄存器直接寻址）

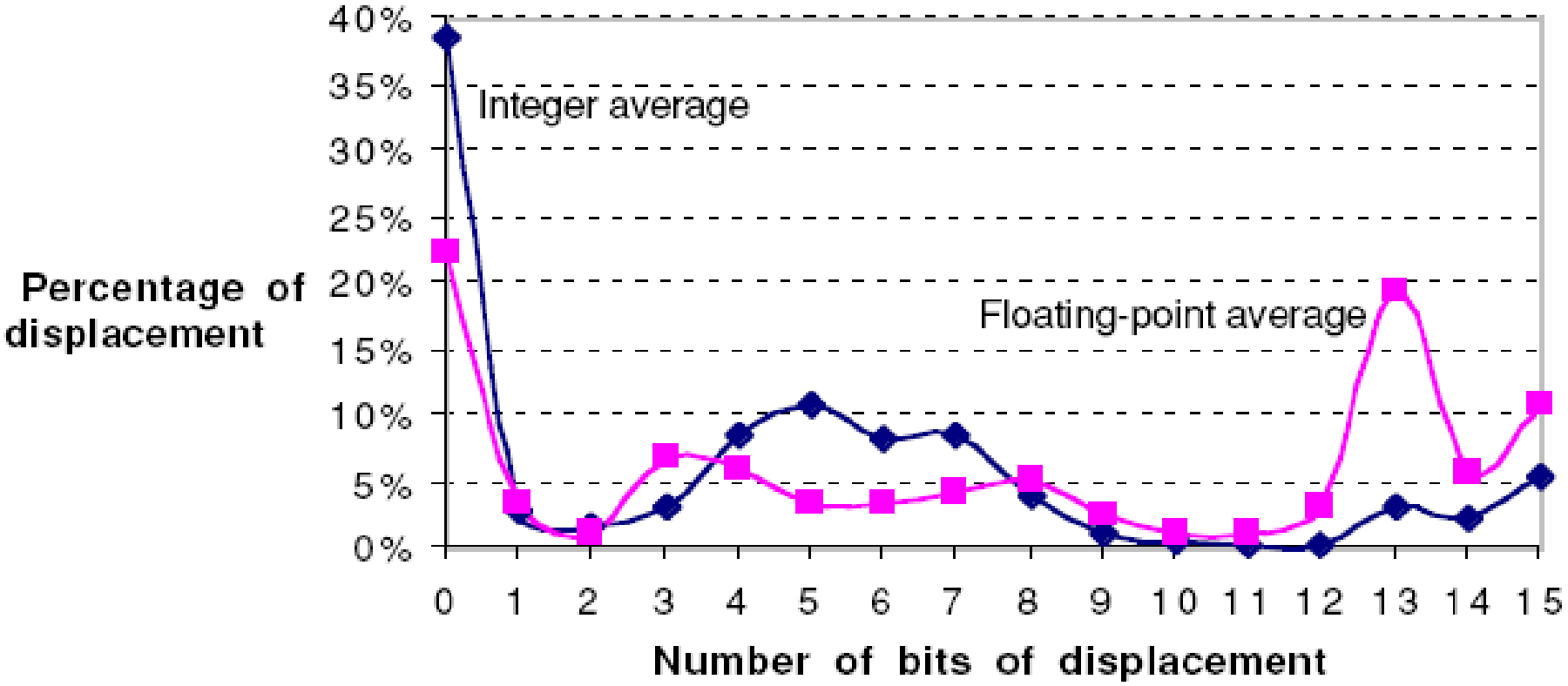


三个**SPEC89**程序在**VAX**结构上的测试结果：

立即寻址，偏移寻址使用较多

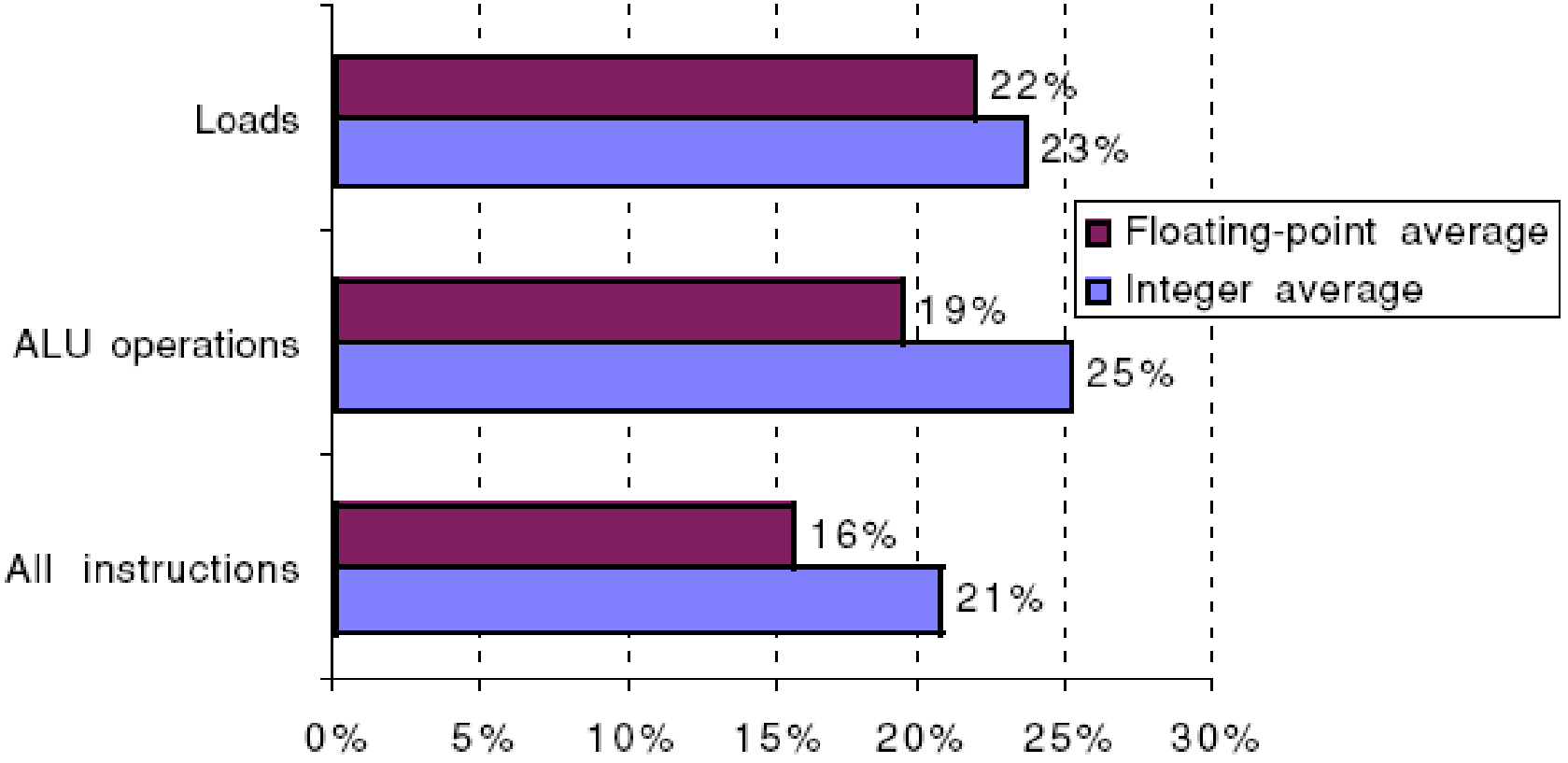
偏移寻址

- 主要问题：偏移的范围（偏移量的大小）



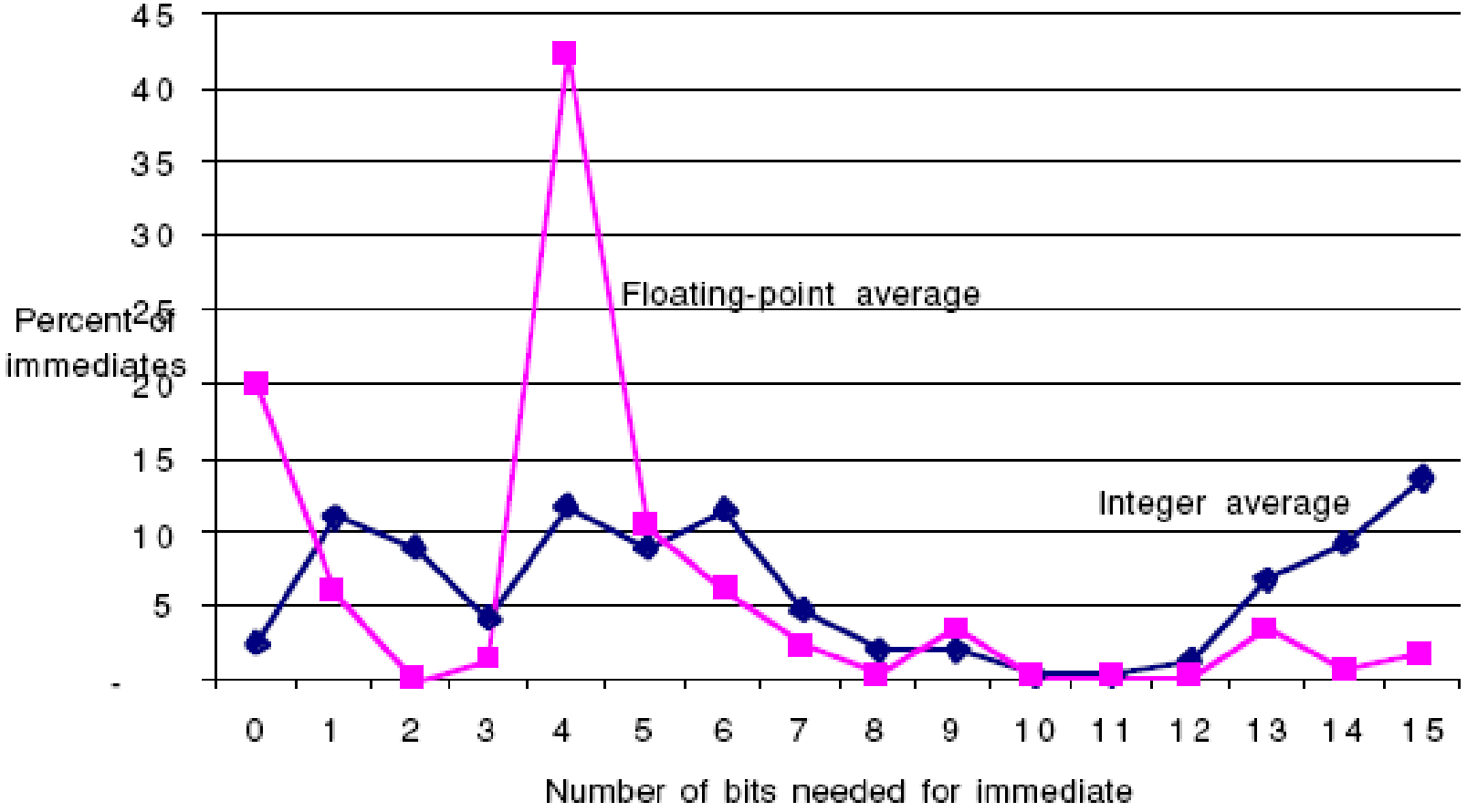
Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

立即数寻址



Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

立即数的大小



The distribution of immediate values. 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 spec cpu2000 programs. 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

- 重要的寻址方式:
  偏移寻址方式, 立即数寻址方式, 寄存器间址方式
  SPEC测试表明，使用频度达到 75%--99%
- 偏移字段的大小应该在 12 - 16 bits
  可满足75%-99%的需求

- 立即数字段的大小应该在 8 -16 bits
  可满足50%-80%的需求

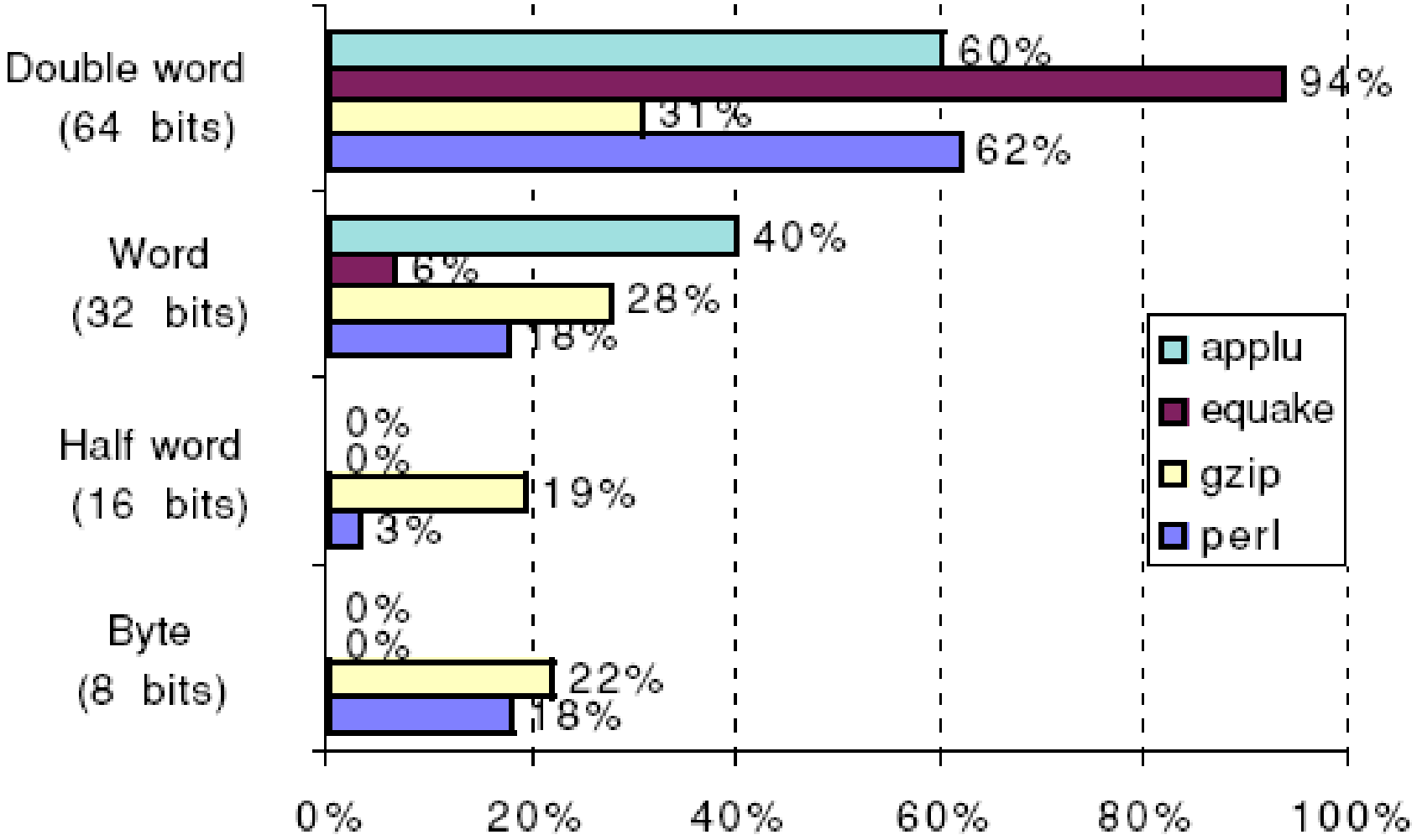
Review

- **ISA需考虑的问题**
  - **Class of ISA**
  - **Memory addressing**
  - **Types and sizes of operands**
  - **Operations**
  - **Control flow instructions**
  - **Encoding an ISA**
- **ISA的类型**
  - 通用寄存器型占主导地位
- 寻址方式
  - 重要的寻址方式: 偏移寻址方式, 立即数寻址方式, 寄存器间址方式
    - » **SPEC测试表明，使用频度达到 75%--99%**
  - 偏移字段的大小应该在 **12 - 16 bits,** 可满足**75%-99%**的需求
  - 立即数字段的大小应该在 **8 -16 bits,** 可满足**50%-80%**的需求

操作数的类型、表示和大小

- 操作数类型和操作数表示也是软硬件的主要界面之一。
- 操作数类型：是面向应用、面向软件系统所处理的各种数据类型。
  - 整型、浮点型、字符、字符串、向量类型等
  - 类型由操作码确定或数据附加硬件解释的标记，一般采用由操作码确定
  - 数据附加硬件解释的标记，现在已经不采用
- 操作数的表示：操作数在机器中的表示，硬件结构能够识别，指令系统可以直接使用的表示格式
  - 整型：原码、反码、补码
  - 浮点：IEEE 754标准
  - 十进制：BCD码，二进制十进制表示

操作数的大小



基准测试的结论：（1）对单字、双字的数据访问具有较高的频率
（2）定义操作数字段长度为64位，更具有一般性

ISA的操作

- **CISC计算机指令集结构的功能设计**
- **RISC计算机指令结构的功能设计**

典型操作类型

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

- 一般计算机都支持前三类所有的操作；
- 不同计算机系统 对系统支持程度不同，但都支持基本的系统功能。
- 对最后四类操作的支持程度差别也很大，有些机器不支持，有些机器还在此基础上做一些扩展，这些指令有时作为可选的指令。

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

- Simple instructions dominate instruction frequency

ISA对操作类型的选择

- 需考虑的因素：速度、价格和灵活性
- 基本要求：指令系统的完整性、规整性、高效率和兼容性
  - 完整性设计：具备基本指令种类
  - 兼容性：系列机
  - 高效率：指令执行速度快、使用频度高
  - 规整性
    - » 让所有运算部件都能对称、均匀的在所有数据存储单元之间进行操作。
    - » 对所有数据存储单元都能同等对待，无论是操作数或运算结果都可以无约束地存放到任意数据存储单元中
  - 正交性
    - » 数据类型独立于寻址方式
    - » 寻址方式独立于所要完成的操作
- 当前对这一问题的处理有两种截然不同的方向
  - **CISC和RISC**

review

- **ISA 研究的问题**
  - **ISA的分类**
  - 操作数部分
    - » 理解存储器地址（尾端和对齐问题）
    - » 寻址方式
    - » 操作数的类型、表示和大小问题
  - 操作码部分
    - » 支持哪些类型的操作
  - 指令格式
- 研究方法：基于统计的方法
- 研究的一些结论
  - 常用寻址方式：立即数、偏移寻址、寄存器间址
  - 偏移字段的大小应该在 **12 - 16 bits**，可满足**75%-99%**的需求
  - 立即数字段的大小应该在 **8 -16 bits**，可满足**50%-80%**的需求
  - 操作数大小：单字、双字的数据访问具有较高的频率，定义操作数字段长度为**64**位，更具有一般性

- **ISA的功能设计**
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
  - 两种类型：**CISC**和**RISC**
- 控制类指令
  - 条件分支最常用
  - 寻址方式：**PC-relative** 和偏移地址至少**8**位，说明动态的转移地址方式
- **CISC**
  - 强化指令功能，减少程序中指令条数，以提高系统性能

CISC计算机ISA的功能设计

- 目标：强化指令功能，减少指令的指令条数，以提高系统性能
- 基本优化方法

1. 面向目标程序的优化

面向目标程序的优化是提高计算机系统性能最直接的方法

- 其指标主要
  - » 缩短程序的长度
  - » 缩短程序的执行时间
- 优化方法
  - » 对大量的目标程序机器执行情况进行统计分析，找出使用频度高，执行时间长的指令或指令串
  - » 对于那些使用频度高的指令，用硬件加快其执行，对于那些使用频度高的指令串，用一条新的指令来代替它

优化目标程序的主要途径（1/2）

1）增强运算型指令的功能

如sin(x), Cos(x), SQRT(X)，甚至多项式计算

如用一条三地址指令完成

$P(X) = C(0)+C(1)X+C(2)X^2+C(3)X^3+\ldots..$

2) 增强数据传送类指令的功能

主要是指数据块传送指令

R-R, R-M, M-M之间的数据块传送可有效的支持向量和矩阵运算，如IBM370

R-Stack之间设置数据块传送指令，能够在程序调用和程序中断时，快速保存和恢复程序现场，如 VAX-11

优化目标程序的主要途径（2/2)

3) 增强程序控制指令的功能

在**CISC**中，一般均设置了多种程序控制指令，正常仅需要转移指令和子程序控制指令

2. 面向高级语言和编译程序改进指令系统

主要是缩小**HL-ML**之间的差距

1）增强面向**HL**和**Compiler**支持的指令功能

在用高级语言编写的源程序中，对各种语句的使用频度和执行时间进行统计分析，对使用频度高、执行时间长的语句，增强有关指令的功能，或增加相关的专门指令，从而达到缩短目标程序长度，减少目标程序执行时间的目的，同时也缩短了编译时间

例如FORTRAN语言和COBOL语言中各种主要语句的使用频度

语言	一元赋值	其他赋值	IF	GOTO	I/O	DO	CALL	其他
FORTRAN	31.0	15.0	11.5	10.5	6.5	4.5	6.0	15.0
COBOL	42.1	7.5	19.1	19.1	8.46	0.17	0.17	3.4

观察结果：

（1）一元赋值在其中比例最大，增强数据传送类指令功能，缩短这类指令的执行时间是对高级语言非常有力的支持，

（2）其他赋值语句中，增1操作比例较大，许多机器都有专门的增1指令

（3）条件转移和无条件转移占22％，38.2%,因此增强转移指令的功能，增加转移指令的种类是必要的

2）高级语言计算机系统

缩小HL和ML的差别时，走到极端，就是把HL和ML合二为一，即所谓的高级语言计算机。在这种机器中，高级语言不需要经过编译，直接由机器硬件来执行。如LISP机，PROLOG机

3）支持操作系统的优化实现一些特权指令

任何一种计算机系统必须有操作系统的支撑才能工作，而OS又必须用指令系统来实现，指令系统对OS的支持主要有

- 处理器工作状态和访问方式的转换
- 进程的管理和切换
- 存储管理和信息保护
- 进程同步和互斥，信号量的管理等

RISC计算机指令集结构的功能设计

- 采用**RISC**体系结构的微处理器
  - **SUN Microsystem: SPARC, SuperSPARC, Ulta SPARC**
  - **SGI: MIPS R4000, R5000, R10000,**
  - **IBM: Power PC**
  - **Intel: 80860, 80960**
  - **DEC: Alpha**
  - **Motorola 88100**
  - **HP HP300/930系列，950系列**

从CISC到RISC

- **1975 IBM公司率先组织力量，研究指令系统的合理性问题，John Coche (1987年图灵奖获得者）1979年 IBM 801, 1986年推出IBM RT PC**
- **1979 David Patterson 研究了CISC指令系统主要存在以下几方面的问题：**
  - **指令使用频度；CISC指令系统复杂-〉增加研制时间和成本，容易出错；**
  - **VLSI设计困难，不利于单片集成；**
  - **许多复杂指令操作复杂，运行速度慢；**
  - **各条指令不规整，不利于先进计算机体系结构技术来提高系统的性能。**
- **1981: Patterson等人研制成功了32位RISC I 微处理器。31种指令，三种数据类型，只有变址和相对寻址两种寻址方式，字长 32位**
- **1983：RISC II**

RISC的定义和特点

- RISC是一种计算机体系结构的设计思想，它不是一种产品。RISC是近代计算机体系结构发展史中的一个里程碑，直到现在，RISC还没有一个确切的定义
- CMU的一篇论文选择论述RISC的特点
  - 大多数指令在单周期内完成
  - 采用Load/Store结构
  - 硬布线控制逻辑
  - 减少指令和寻址方式的种类
  - 固定的指令格式
  - 注重代码的优化
- 从目前的发展看，RISC体系结构还应具有如下特点：
  - 面向寄存器结构
  - 十分重视流水线的执行效率－尽量减少断流
  - 重视优化编译技术

90年代，IEEE的Michael Slater对RISC的定义做了如下描述：

- **RISC为使流水线高效执行，应具有如下特征：**
  - 简单而统一格式的指令译码
  - 大部分指令可以单周期执行完成
  - 只有**Load/Store**指令访存
  - 简单寻址方式
  - 采用**Load**延迟技术
- **RISC为使优化编译便于产生优化代码，应具有如下特征：**
  - 三地址指令格式
  - 较多的寄存器
  - 对称的指令格式
- **减少指令平均执行周期数是RISC思想的精华**

问题

- **RISC的指令系统精简了，CISC中的一条指令可能由一串指令才能完成，那么为什么RISC执行程序的速度比CISC还要快？**

  ExecuteTime = CPI X IC X T

	IC	CPI	T
CISC	1	2~15	33ns~5ns
RISC	1.3~1.4	1.1~1.4	10ns~2ns

IC：实际统计结果，RISC的IC只比CISC 长30％~40%

CPI: CISC CPI一般在4~6之间，RISC 一般CPI =1 , Load/Store 为2

T: RISC采用硬布线逻辑，指令要完成的功能比较简单，

RISC为什么会减少CPI

- 硬件方面：硬布线控制逻辑，减少指令和寻址方式的种类，使用固定格式，采用**Load/Store**，指令执行过程中设置多级流水线。
- 软件方面：十分强调优化编译的作用

RISC的关键技术

- 延时转移技术
- 指令取消技术
- 重叠寄存器窗口技术
  - 处理器中设置较多的寄存器堆，并划分很多窗口
  - 每个过程使用其中相邻的三个窗口和一个公共的窗口
  - 一个与前一个过程共用，一个与下一个过程共用，一个作为局部寄存器
- 指令流调整技术
- 硬件为主固件为辅

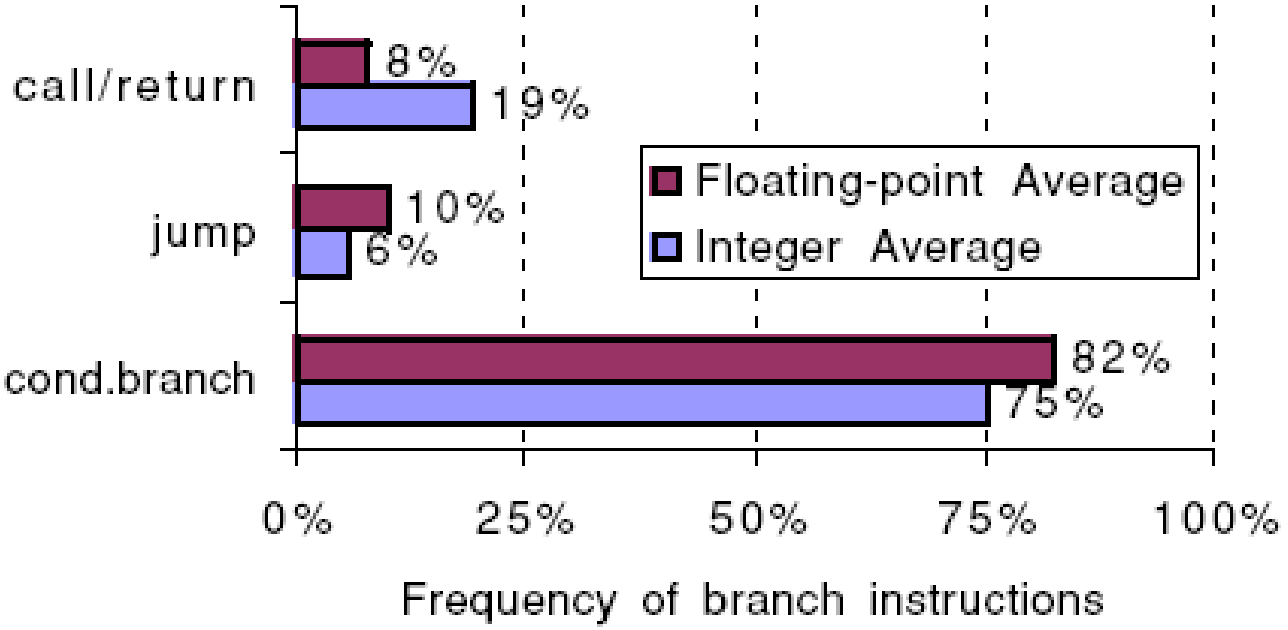
操作类型小结

以下指令类型使用频度最高，指令系统应该支持这些类型的指令

load,
store,
add,
subtract,
move register-register,
and,
shift,
compare equal, compare not equal,
branch,
jump,
call,
return;

控制类指令

- 四种类型的控制流改变：条件分支(Conditional branch) 、跳转(Jump)、过程调用(Procedure calls)、过程返回(Procedure returns)

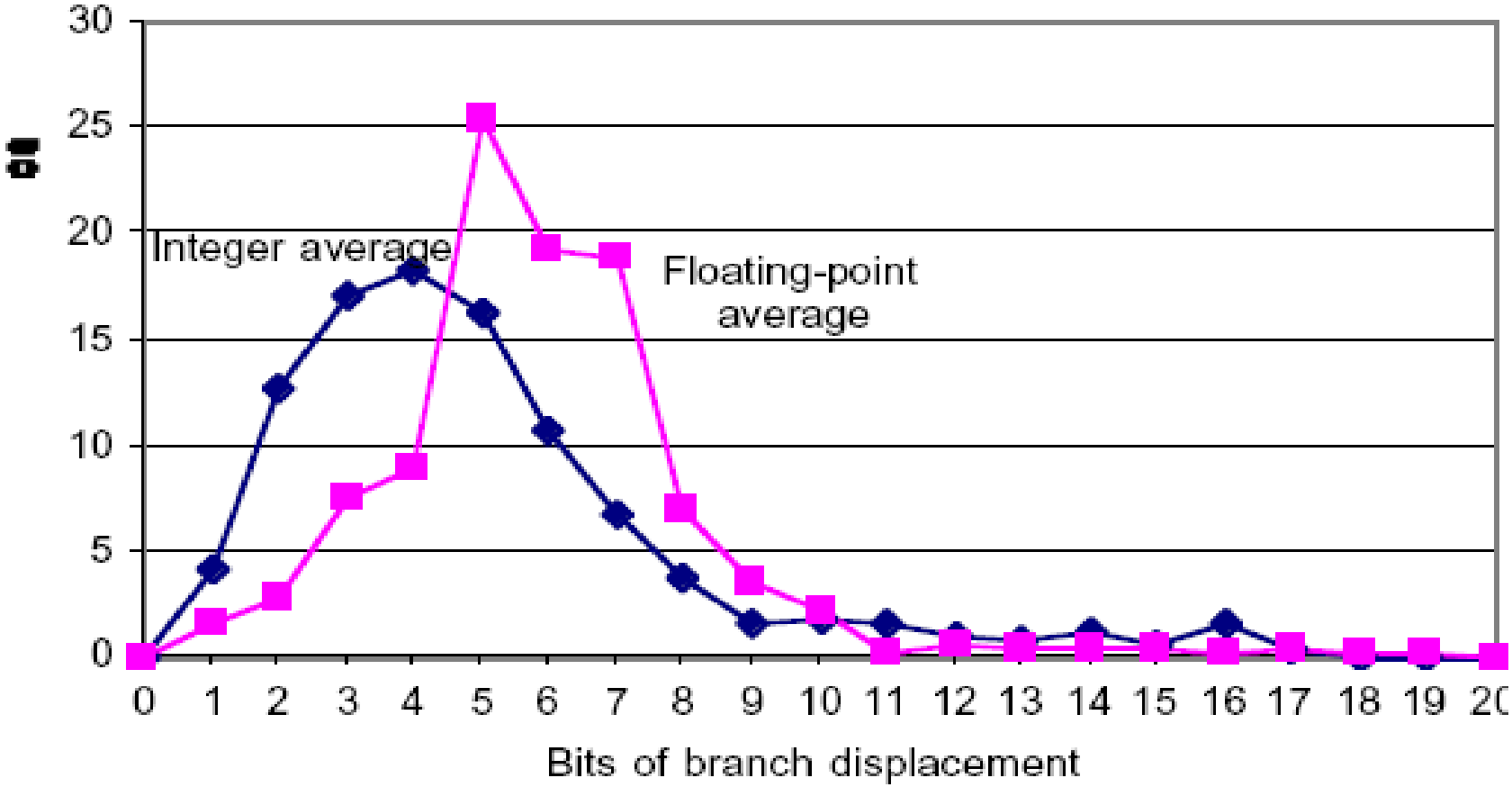


Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

控制流类指令中的寻址方式

- **PC-relative** 方式 （相对寻址）
- 说明动态的转移地址方式:
  - 编译时不知道转移地址，程序执行时动态确定
  - 转移地址放到某一寄存器中
  - …….

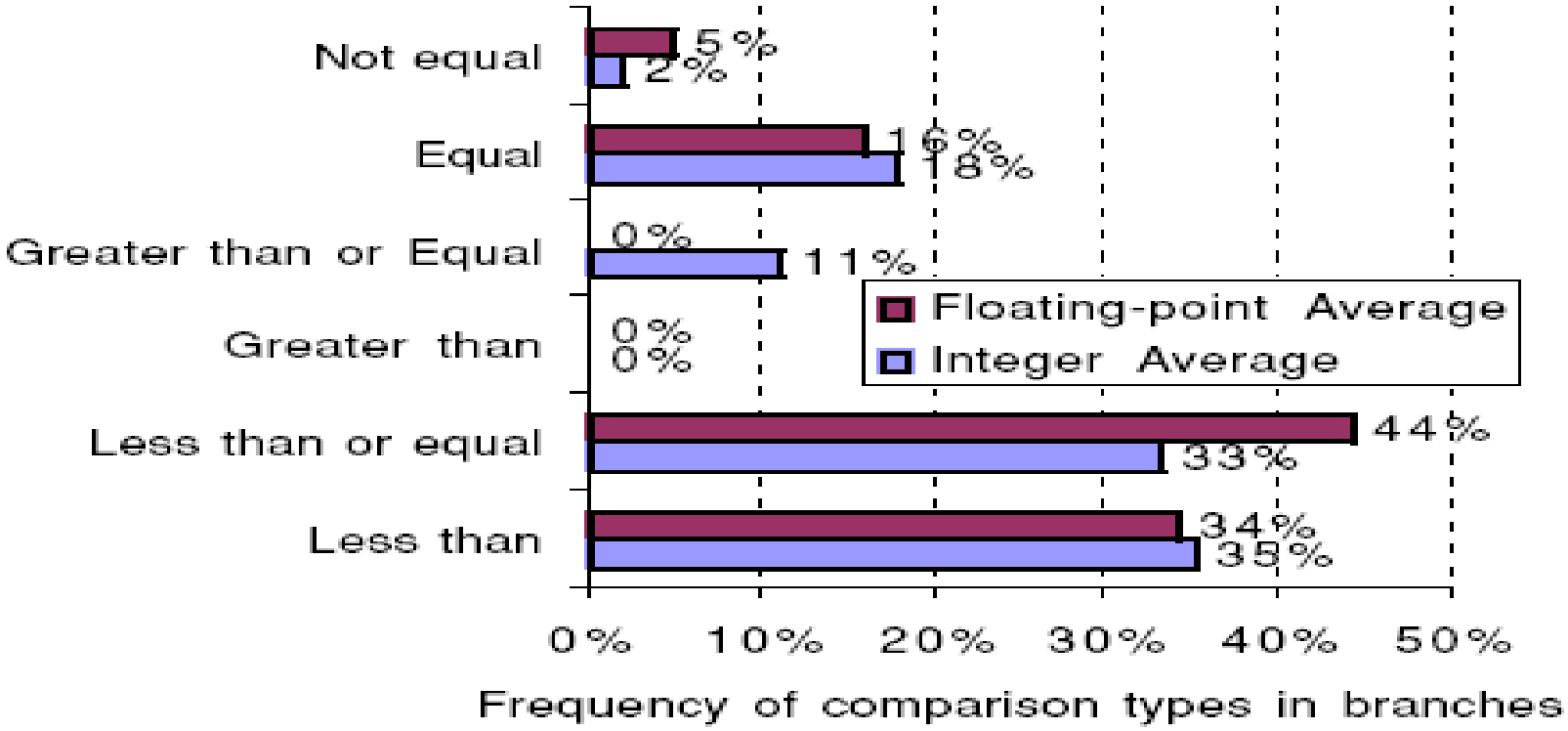
转移目标地址与当前指令的距离



Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

建议：PC-relative 寻址，偏移地址至少8位

分支比较类型比较

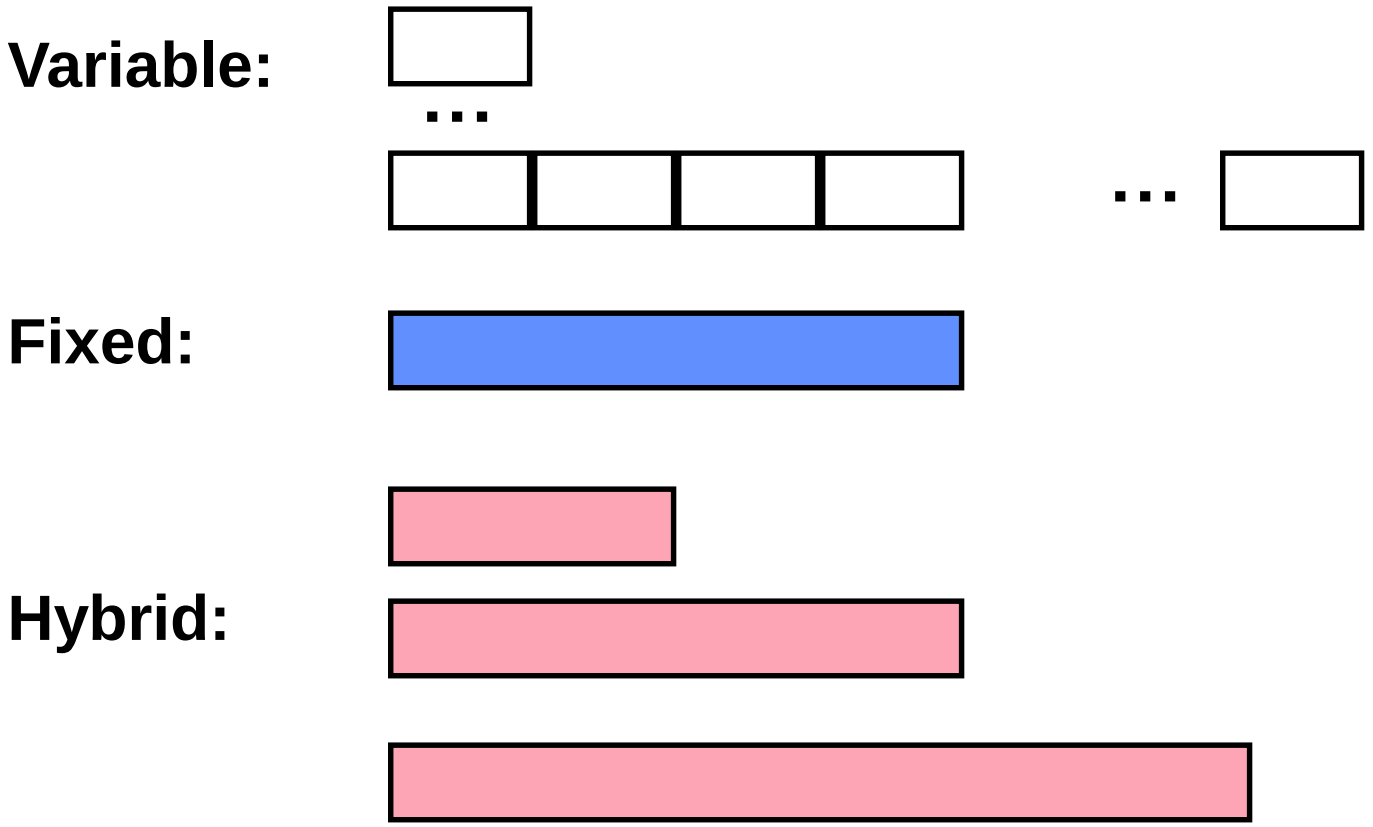


Alpha Architecture with full optimization for Spec CPU2000, showing the average of integer programs(CINT2000) and the average of floating-point programs (CFP2000)

review

- ISA功能设计－确定提供哪些操作类型。
- CISC
  - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
  - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
- RISC
  - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
  - 主要手段：充分发挥流水线的效率，提高CPI

指令编码



指令格式选择策略

- 如果代码长度最重要，那么使用变长指令格式
- 如果性能至关重要，使用固定长度指令
- 有些嵌入式CPU附加可选模式，
  由每一应用自己选择性能还是代码量
- 有些机器使用边执行边解压的方式

指令格式

•

•如果每条指令存在多个存储器操作数，或有多种寻址方式

=>每一操作数都要说明其寻址方式

• 对于**Load-store**型机器，每条指令只有一个存储器地址，并且只有较少的寻址方式

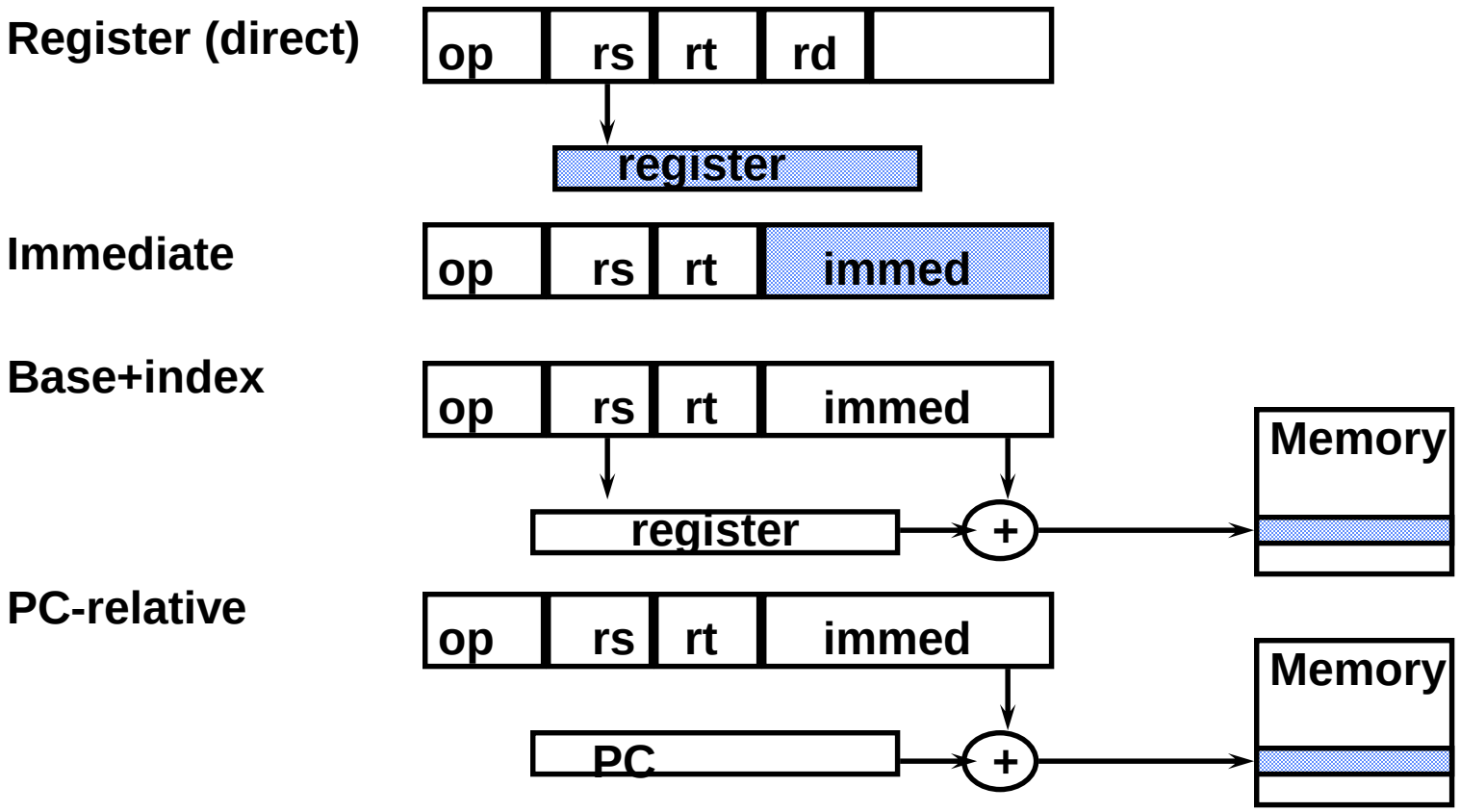
=> 可以将寻址方式反映在操作码中

review

- **ISA的分类**
  - 累加器型、堆栈型和寄存器型
- 寻址方式
  - 立即寻址、偏移寻址和寄存器寻址
  - 偏移量字段：**12 ～ 16 bits**
  - 立即数字段： **8 to 16 bits**
- 指令格式
  - 变长指令格式、定长指令格式、
  - 以上两种格式的折中

MIPS 寻址方式/指令格式

- 所有指令都是32位宽



- **Register Indirect?**

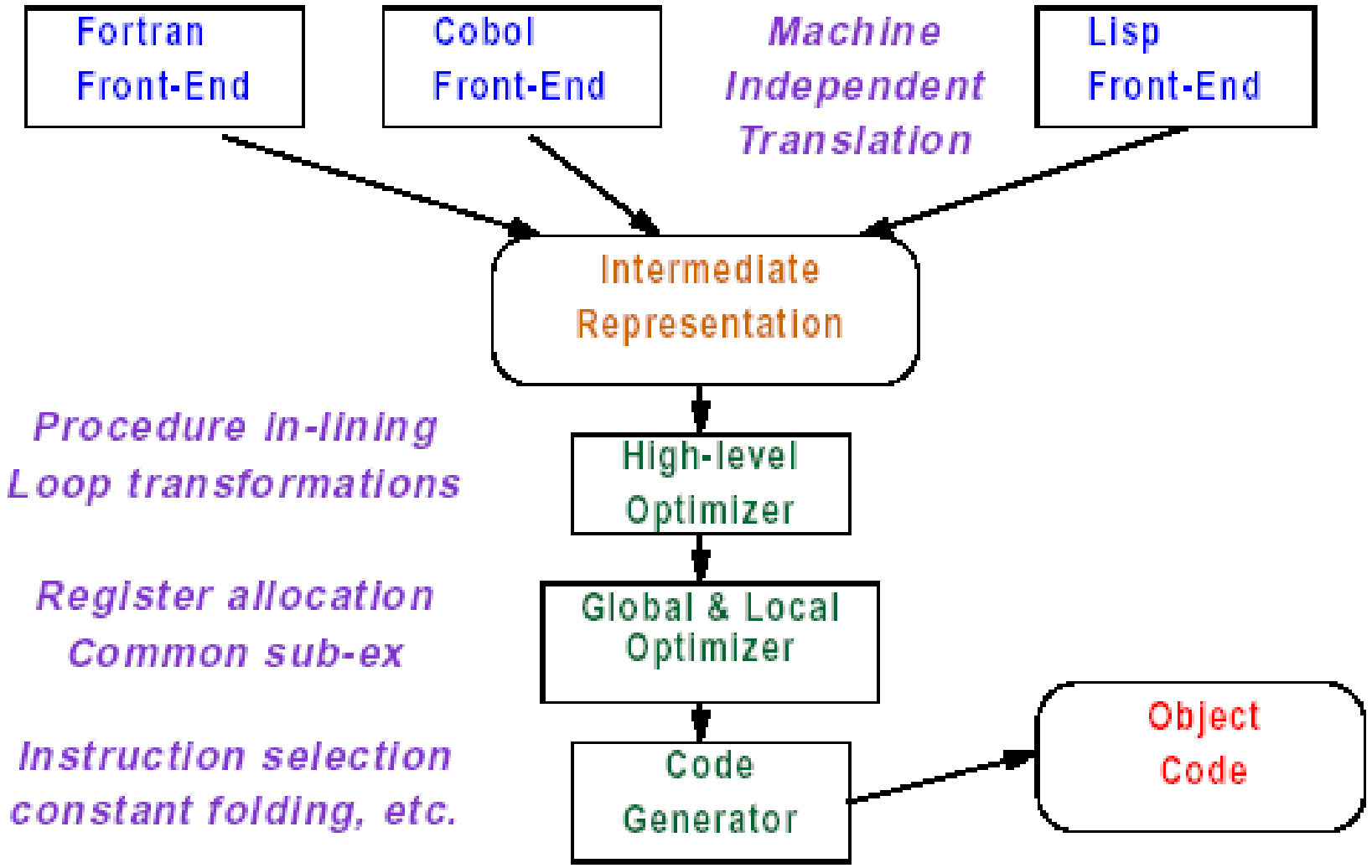
review

- ISA功能设计－确定提供哪些操作类型。
- CISC
  - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
  - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
- RISC
  - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
  - 主要手段：充分发挥流水线的效率，提高CPI
- 指令格式
  - 变长、固定长度、若干种长度

编译技术与计算机体系结构设计

- 使用汇编语言编程已不是主流
  - 现在使用汇编的人很少
  - 因此编译器更需要ISA
- 编译器的设计目标
  - 正确性
  - 编译后代码的速度
  - 其他目标
    - 编译的速度
    - 调试的支持
    - 语言间的互操作性等

现代编译器的典型结构


Fortran
Front-End
Cobol
Front-End
Machine
Independent
Translation
Lisp
Front-End
Intermediate
Representation
Procedure In-lining
Loop transformations
High-level
Optimizer
Register allocation
Common sub-ex
Global & Local
Optimizer
Instruction selection
constant folding, etc.
Code
Generator
Object
Code

优化类型（1/2）

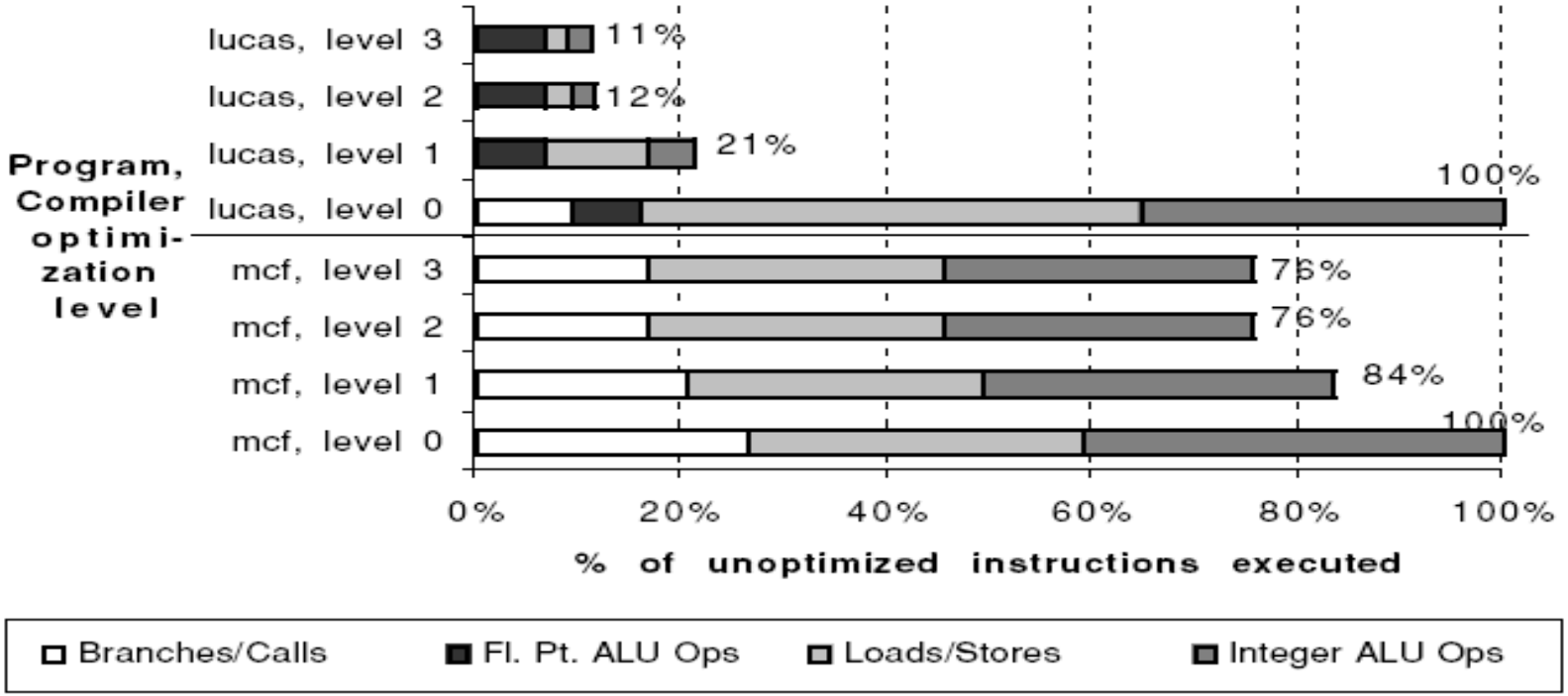
- 高级优化－在源代码级
  - 例如：仅一次调用的过程，采用in-line方式，避免CALL
- 局部优化
  - 消去公共子表达式：用临时变量保存第一次计算的公共子表达式的值
  - 常数传递，将所有被赋值为常数的变量，用该常数值代替
  - 降低堆栈的深度，对表达式重新组织，尽量减少表达式求值所用的资源
- 全局优化－对循环和分支进行优化转换
  - 代码移动－将在每次循环中计算相同值的代码移到循环外面
  - 简化或消去数组下标的计算

优化类型（2/2）

- 寄存器分配
- 与机器相关的优化
  - 降低计算负载
    - » 如乘法用移位和加法来完成
  - 指令调度：重新组织指令序列，尽可能减少断流现象
  - 分支偏移的优化
    - » 重新安排代码，使分支偏移尽可能小

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	**At or near the source level; processor-independent**	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	**Within straight-line code**	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	**Across a branch**	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array-addressing calculations within loops	2%
Processor-dependent	**Depends on processor knowledge**	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (*software pipelining*), and global register allocation. Level 3 adds procedure integration. These experiments were performed on the Alpha compilers.



Change in instruction count for the programs lucas and mcf from the SPEC2000 as compiler optimization levels vary.

Stanford Ucode Compiler 优化的效果

Based on 12 small Fortran and Pascal Programs

Optimization	Percent Faster
Procedure integration only	10%
Local optimizations only	5%
Local optimizations + register allocation	26%
Global and Local optimizations	14%
Local + Global optimizations + register allocation	63%
Everything	81%

Review

- 功能设计
  - **RISC vs. CISC**
- 指令格式
  - 变长指令格式、定长指令格式、
  - 以上两种格式的折中
- 编译技术与计算机系统结构
  - 现代编译器的基本结构
- 编译优化-4个层次
  - 高层优化：一般在源码上进行，同时把输出传递给以后的优化扫描步骤
  - 局部优化：仅在一系列代码片断之内（基本块）将代码优化
  - 全局优化：将局部优化扩展为跨越分支，并且引入一组针对优化循环的转换
  - 与机器相关的优化：充分利用特定的系统结构

ISA设计需了解的有关Compiler的问题

- 如何分配变量
- 如何寻址变量
- 需要多少寄存器
- 优化技术对指令使用频度有何影响
- 使用哪些控制结构
- 这些控制结构使用频度

高级语言分配数据的区域

- 栈（Stack）
  - 用来分配局部变量
  - 用于存储活动记录（过程调用和返回），通过堆栈指针访问其中的内容。
- 全局数据区（global data area）
  - 用来分配被静态说明的对象，如常量和全局变量。其中数组和其他聚集类型的数据结构占很大比例
- 堆（heap）
  - 用于分配动态对象，用指针访问，通常不是标量。全局变量和堆变量因为存在别名问题而无法分配到寄存器

寄存器分配问题

- 栈中对象的寄存器分配相对简单
- 全局变量的寄存器分配相对比较困难
- 堆对象的寄存器分配几乎是不可能的
  - 原因：一般用指针来访问，使得几乎无法分配
- 有些全局变量和栈对象由于存在别名问题也无法分配
  - 原因：存在多条路径来访问变量的地址。
- 寄存器分配是优化的主要手段，因此在这方面的努力是非常重要的

关于编译优化的一些研究的结论

- 减少分支语句非常困难
- 大量的优化使得存储器访问操作减少
- 可以减少一些ALU操作

结果是

- 控制类指令在统计上占较大比例
- 控制类指令很难加速

编译技术与计算机体系结构设计小结

- 有利于编译器的ISA
  - 规整性和正交性：没有特殊的寄存器，例外情况尽可能少，所有操作数模式可用于任何数据类型和指令类型,要求操作、数据类型和寻址方式必须正交，有利于简化代码生成过程。
  - 完整性：支持基本的操作和满足目标的应用系统需求
  - 帮助编译器设计者了解各种代码序列的执行效率和代价，有助于编译器的优化
  - 对于在编译时就已经可确定的量，提供能够将其变为常数的指令
- 寄存器分配是关键问题
  - 寄存器数目多有利于编译器的设计与实现

- 提供至少**16**个通用寄存器和独立的浮点寄存器
- 保证所有的寻址方式可用于各种数据传送指令
- 最小指令集

一些统计比较结果

Memory-Memory vs. Load-Store ISA's

% of total dynamic instruction count

- Branches

Program	m-m ISA	l-s ISA
TeX	31%	14%
Spice	22%	7%
GCC	32%	18%

- Moves

Program	m-m ISA	l-s ISA
TeX	33%	36%
Spice	17%	28%
GCC	23%	36%

M-M vs L-S (continued)

- ALU operations

Program	m-m ISA	l-s ISA
TeX	36%	50%
Spice	61%	65%
GCC	45%	46%

- 一些结论
  - **Load-Store 型ISA，move类指令比例大于M-M型ISA**
  - **Load-Store型ISA，Branch类指令比例小于M-M型ISA**
  - **Load-Store型ISA，平均CPI和T较小**

MIPS指令集结构

- MIPS64的一个子集，简称为MIPS
- MIPS的寄存器
  - 32个64位通用寄存器（GPRs）
    - » R0，R1，...，R31
    - » 也被称为整数寄存器
    - » R0的值永远是0
  - 32个64位浮点数寄存器（FPRs）
    - » F0，F1，...，F31
    - » 用来存放32个单精度浮点数（32位），也可以用来存放32个双精度浮点数（64位）。
    - » 存储单精度浮点数（32位）时，只用到FPR的一半，其另一半没用
  - 一些特殊寄存器
    - » 它们可以与通用寄存器交换数据。
    - » 例如，浮点状态寄存器用来保存有关浮点操作结果的信息。

MIPS的数据表示

- 整数
  - 字节（8位）　半字（16位）字（32位）　双字（64位）
- 浮点数
  - 单精度浮点数（32位）　双精度浮点数（64位）

- **字节、半字或者字在装入64位寄存器时，用零扩展或者用符号位扩展来填充该寄存器的剩余部分。装入以后，对它们将按照64位整数的方式进行运算。**

MIPS的寻址方式

- 立即数寻址与偏移量寻址
  - 立即数字段和偏移量字段都是16位的。
- 寄存器间接寻址是通过把0作为偏移量来实现的
- 16位绝对寻址是通过把R0（其值永远为0）作为基址寄存器来完成的
- MIPS的存储器是按字节寻址的，地址为64位
- 所有存储器访问都必须是边界对齐的

存储器中的数据存放（存储字长为 32 位）

边界对准

地址（十进制）

字（地址 0）				0
字（地址 4）				4
字节（地址11）	字节（地址10）	字节（地址 9）	字节（地址 8）	8
字节（地址15）	字节（地址14）	字节（地址13）	字节（地址12）	12
半字（地址18）		半字（地址16）		16
半字（地址22）		半字（地址20）		20
双字（地址24）				24
双字				28
双字（地址32）				32
双字				36

边界未对准

地址（十进制）

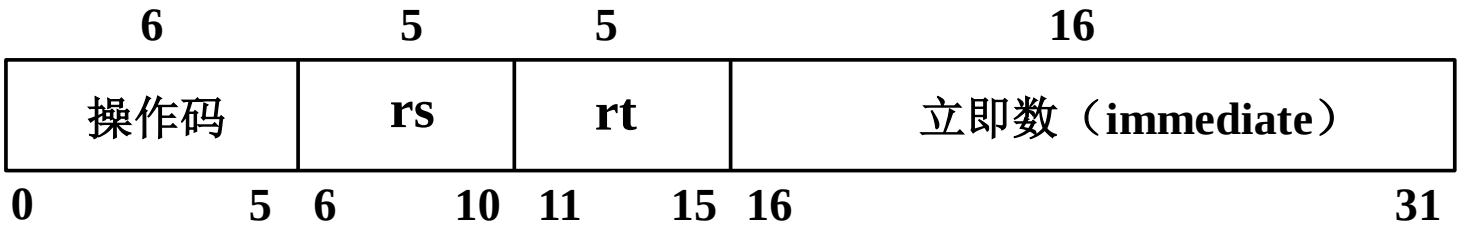
字(地址2)	半字(地址0)	0
字节(地址7)	字节(地址6)	字(地址4)	4
半字(地址10)	半字(地址8)	8

MIPS的指令格式

- 寻址方式编码到操作码中
- 所有的指令都是32位的
- 操作码占6位
- 3种指令格式
  - I类
  - R类
  - J类

I类指令

- 包括所有的load和store指令、立即数指令、，分支指令、寄存器跳转指令、寄存器链接跳转指令。
- 立即数字段为16位，用于提供立即数或偏移量。



具体的I类指令

- load指令

  访存有效地址：Regs[rs]＋immediate

  从存储器取来的数据放入寄存器rt

- store指令

  访存有效地址：Regs[rs]＋immediate

  要存入存储器的数据放在寄存器rt中

- 立即数指令

  Regs[rt] ← Regs[rs] op immediate

- 分支指令

  转移目标地址：Regs[rs]＋immediate，rt无用

- 寄存器跳转、寄存器跳转并链接

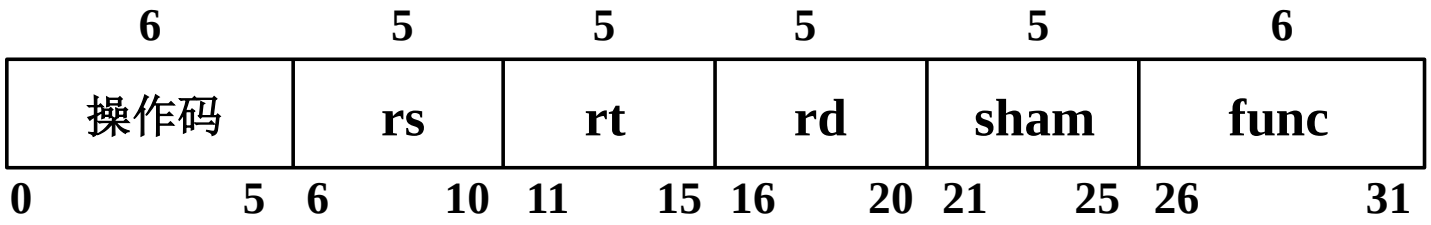
  转移目标地址为Regs[rs]

R类指令

- 包括ALU指令、专用寄存器读/写指令、move指令等
- ALU指令

  Regs[rd]← Regs[rs] funct Regs[rt]

  func为具体的运算操作编码



J类指令

- 包括跳转指令、跳转并链接指令、自陷指令、异常返回指令
- 在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址

MIPS的操作

- MIPS指令可以分为四大类
  - load和store、ALU操作、分支与跳转、浮点操作
- 符号的意义
  - $x \leftarrow_n y$：从y传送n位到x
  - x，y←z：把z传送到x和y
  - 下标：表示字段中具体的位；
    - » 对于指令和数据，按从最高位到最低位（即从左到右）的顺序依次编号，最高位为第0位，次高位为第1位，依此类推。
    - » 下标可以是一个数字，也可以是一个范围。例如：

      $Regs[R4]_0$：R4的符号位；$Regs[R4]_{56..63}$：R4的最低字节

- **Mem：表示主存；按字节寻址，可以传输任意个字节。**
- **上标：用于表示对字段进行复制的次数。**

例如：0^{32}：一个32位长的全0字段

- 符号##：用于两个字段的拼接，并且可以出现在数据传送的任何一边。举例：R8、R10：64位的寄存器，则

$$Regs[R8]_{32..63} \leftarrow_{32} (Mem\ [Regs[R6]]_0)^{24}\ \#\#\ Mem\ [Regs[R6]]$$

表示的意义是：

以R6的内容作为地址访问内存，得到的字节按符号位扩展为32位后存入R8的低32位，R8的高32位（即$Regs[R8]_{0..31}$）不变。

- load和store指令

指令举例	指令名称	含 义
LD R2，20(R3)	装入双字	Regs[R2] $\leftarrow_{64}$ Mem[20+Regs[R3]]
LW R2，40(R3)	装入字	Regs[R2] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{32} ## Mem[40+Regs[R3]]
LB R2，30(R3)	装入字节	Regs[R2] $\leftarrow_{64}$ (Mem[30+Regs[R3]]$_0$)^{56} ## Mem[30+Regs[R3]]
LBU R2，40(R3)	装入无符号字节	Regs[R2] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R2，30(R3)	装入半字	Regs[R2] $\leftarrow_{64}$ (Mem[30+Regs[R3]]$_0$)^{48} ## Mem[30+Regs[R3]]## Mem[31+Regs[R3]]
L.S F2，60(R4)	装入半字	Regs[F2] $\leftarrow_{64}$ Mem[60+Regs[R4]] ## 0^{32}
L.D F2，40(R3)	装入双精度浮点数	Regs[F2] $\leftarrow_{64}$ Mem[40+Regs[R3]]
SD R4，300(R5)	保存双字	Mem[300+Regs[R5]] $\leftarrow_{64}$ Regs[R4]
SW R4，300(R5)	保存字	Mem[300+Regs[R5]] $\leftarrow_{32}$ Regs[R4]
S.S F2，40(R2)	保存单精度浮点数	Mem[40+Regs[R2]] $\leftarrow_{32}$ Regs[F2] $_{0\cdot 31}$
SH R5，502(R4)	保存半字	Mem[502+Regs[R4]] $\leftarrow_{16}$ Regs[R5] $_{48\cdot\cdot 63}$

- **ALU指令**

寄存器-寄存器型（RR型）指令或立即数型

算术和逻辑操作：加、减、与、或、异或和移位等

指令举例	指令名称	含义
DADDU R1，R2，R3	无符号加	Regs[R1]← Regs[R2]+ Regs[R3]
DADDIU R4，R5，#6	加无符号立即数	Regs[R4]← Regs[R5]+6
LUI R1，#4	把立即数装入到一个字的高16位	Regs[R1]← 0^{32} ## 4 ## 0^{16}
DSLL R1，R2，#5	逻辑左移	Regs[R1]← Regs[R2]<<5
DSLT R1，R2，R3	置小于	If(Regs[R2]< Regs[R3]) Regs[R1]← 1 else Regs[R1]←0

R0的值永远是0，它可以用来合成一些常用的操作。

例如：

```
DADDIU R1，R0，#100
                    //给寄存器R1装入常数100
DADD R1，R0，R2
                    //把寄存器R2中的数据传送到寄存器R1
```

MIPS控制类指令

指令举例	指令名称	含义
J name	跳转	PC $_{36\cdot63}$← name<<2
JAL name	跳转并链接	Regs[R31]←PC+4；PC $_{36\cdot63}$←name<<2； ((PC+4)－2^{27})≤name＜((PC+4)+2^{27})
JALR R3	寄存器跳转并链接	Regs[R31]←PC+4；PC← Regs[R3]
JR R5	寄存器跳转	PC← Regs[R5]
BEQZ R4，name	等于零时分支	if(Regs[R4]== 0) PC←name ； ((PC+4)－2^{17})≤name＜((PC+4)+2^{17})
BNE R3，R4，name	不相等时分支	if(Regs[R3]!= Regs[R4]) PC←name ((PC+4)－2^{17})≤name＜((PC+4)+2^{17})
MOVZ R1，R2，R3	等于零时移动	if(Regs[R3]==0) Regs[R1]← Regs[R2]

跳转指令

- 根据跳转指令确定目标地址的方式不同以及跳转时是否链接，可以把跳转指令分成4种。
- 确定目标地址的方式
  - 把指令中的26位偏移量左移2位（因为指令字长都是4个字节）后，替换程序计数器的低28位。
  - 间接跳转：由指令中指定的一个寄存器来给出转移目标地址。
- 跳转的两种类型
  - 简单跳转：把目标地址送入程序计数器。
  - 跳转并链接：把目标地址送入程序计数器，把返回地址（即顺序下一条指令的地址）放入寄存器R31。

分支指令（条件转移）

- 分支条件由指令确定。

  例如：测试某个寄存器的值是否为零

- 提供一组比较指令，用于比较两个寄存器的值。

  例如：“置小于”指令

- 有的分支指令可以直接判断寄存器内容是否为负，或者比较两个寄存器是否相等。

- 分支的目标地址。

  由16位带符号偏移量左移两位后和PC相加的结果来决定

- 一条浮点条件分支指令：通过测试浮点状态寄存器来决定是否进行分支。

MIPS的浮点操作

- 由操作码指出操作数是单精度（SP）或双精度（DP）
  - 后缀S：表示操作数是单精度浮点数
  - 后缀D：表示是双精度浮点数
- 浮点操作

  包括加、减、乘、除，分别有单精度和双精度指令。
- 浮点数比较指令
  - 根据比较结果设置浮点状态寄存器中的某一位，以便于后面的分支指令BC1T（若真则分支）或BC1F（若假则分支）测试该位，以决定是否进行分支。

本章小结

- **90年代ISA的主要变化：**
  - **Address size doubles**
  - **Optimization of conditional branches via conditional execution**
  - **Optimization of cache performance via prefetch**
  - **Support for multimedia**
  - **Faster floating-point operations**
- **ISA设计的发展趋势**
  - **Long instruction words**
  - **Increased conditional execution**
  - **Blending of general-purpose and DSP architectures**
  - **80x86 emulation**

附件

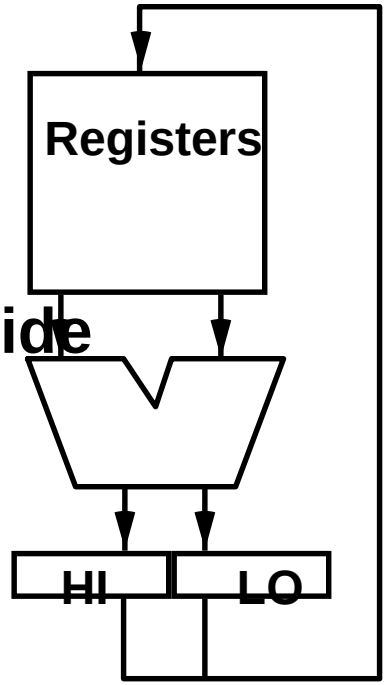
- MIPS I Operation Overview

MIPS I Operation Overview

- **Arithmetic Logical:**
  - **Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU**
  - **AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI**
  - **SLL, SRL, SRA, SLLV, SRLV, SRAV**
- **Memory Access:**
  - **LB, LBU, LH, LHU, LW, LWL,LWR**
  - **SB, SH, SW, SWL, SWR**

Multiply / Divide

- **Start multiply, divide**
  - **MULT rs, rt**
  - **MULTU rs, rt**
  - **DIV rs, rt**
  - **DIVU rs, rt**
- **Move result from multiply, divide**
  - **MFHI rd**
  - **MFLO rd**
- **Move to HI or LO**
  - **MTHI rd**
  - **MTLO rd**
- **Why not Third field for destination? (Hint: how many clock cycles for multiply or divide vs. add?)**



Data Types

Bit: 0, 1

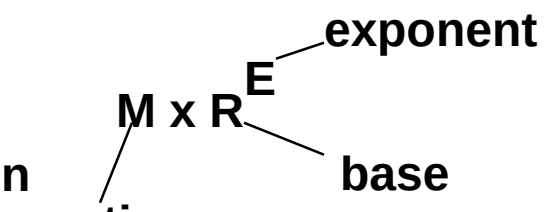
Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word

Character:
- ASCII 7 bit code
- UNICODE 16 bit code

Decimal:
- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

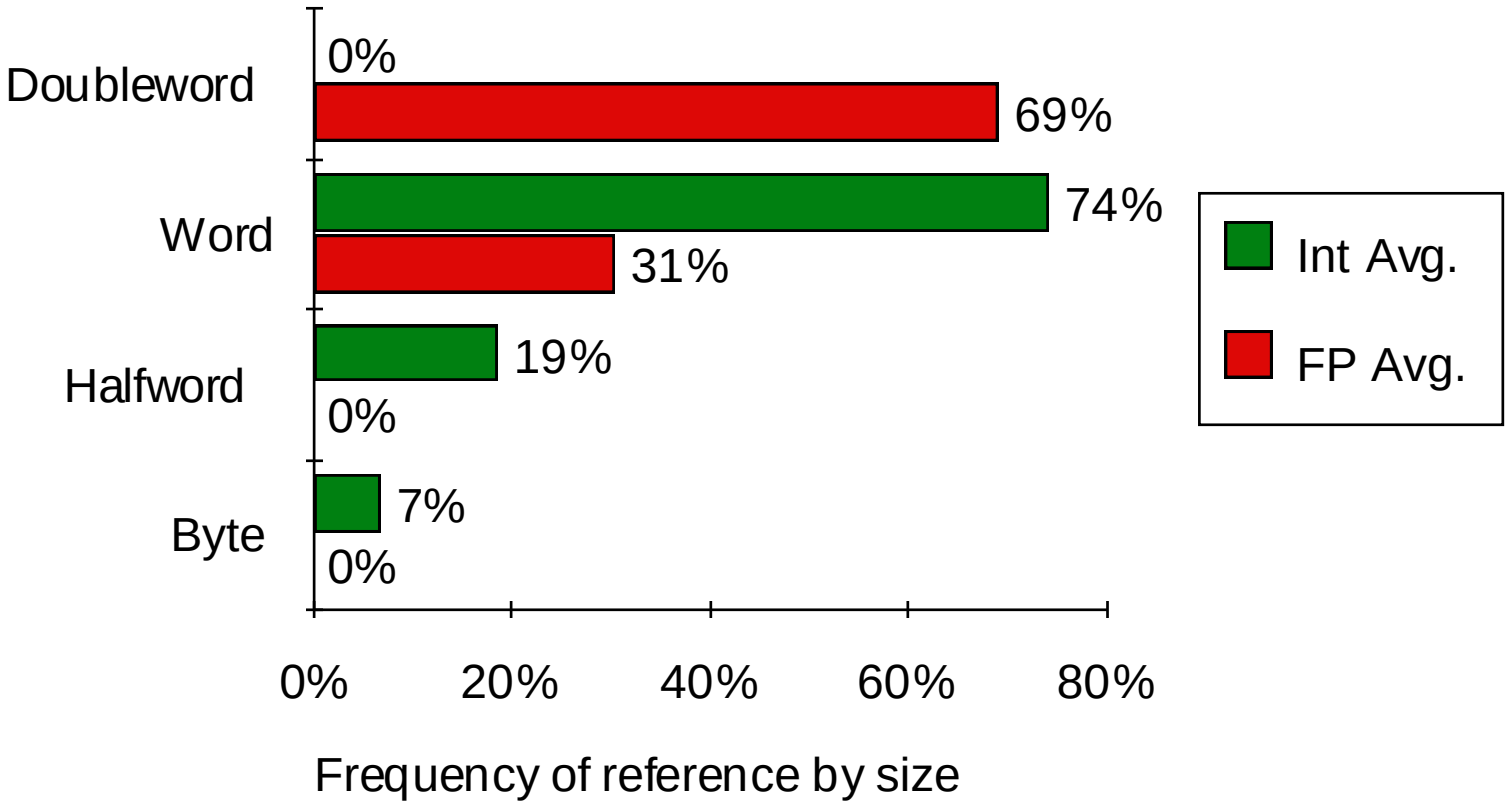
Integers:
- 2's Complement

Floating Point:
- Single Precision
- Double Precision
- Extended Precision



How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?

Operand Size Usage



- **Support for these data sizes and types:**
  8-bit, 16-bit, 32-bit integers and
  32-bit and 64-bit IEEE 754 floating point numbers

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

Which add for address arithmetic? Which add for integers?

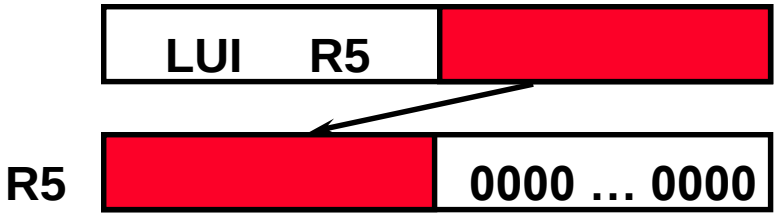
MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

MIPS data transfer instructions

Instruction	*Comment*
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

Why need LUI?



When does MIPS sign extend?

- **When value is sign extended, copy upper bit to full value:**

  Examples of sign extending 8 bits to 16 bits:

  00001010 ⇒ 00000000 00001010
  10001100 ⇒ 11111111 10001100

- **When is an immediate value sign extended?**
  - **Arithmetic instructions (add, sub, etc.) sign extend immediates *even for the unsigned versions of the instructions!***
  - **Logical instructions *do not sign extend***

- **Load/Store half or byte *do sign extend*, but unsigned versions do not.**

Methods of Testing Condition

- **Condition Codes**

  Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

  ex: add r1, r2, r3

  bz label

- **Condition Register**

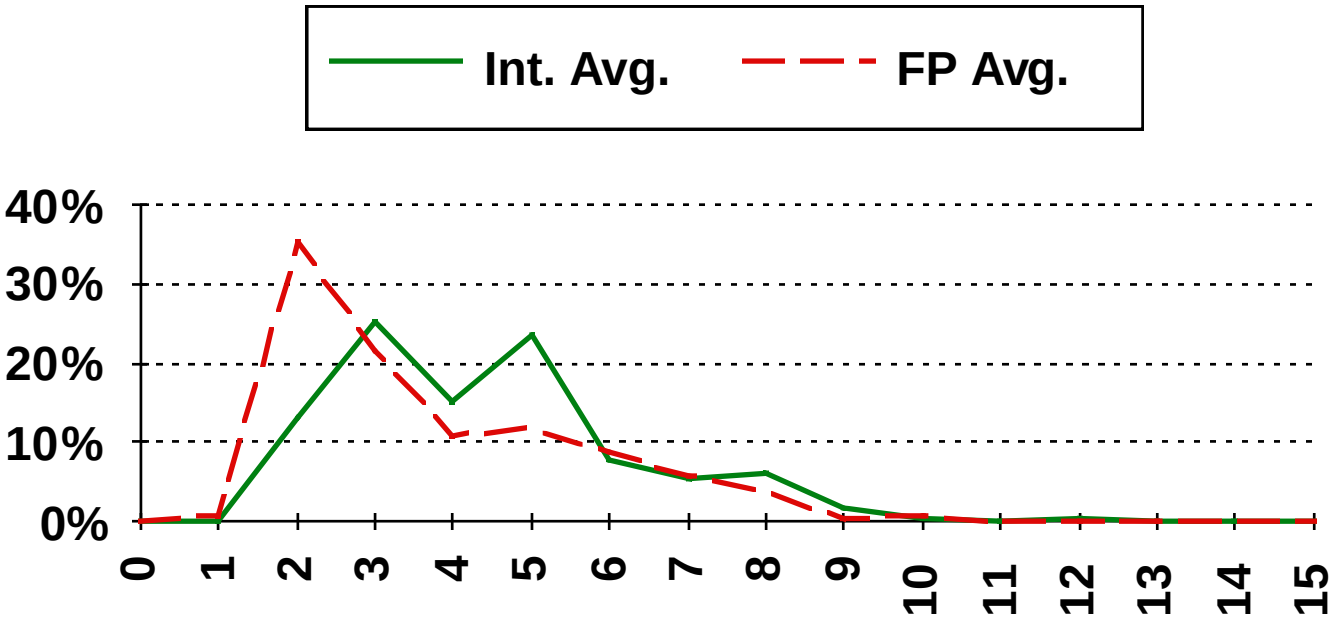
  Ex: cmp r1, r2, r3

  bgt r1, label

- **Compare and Branch**

  Ex: bgt r1, r2, label

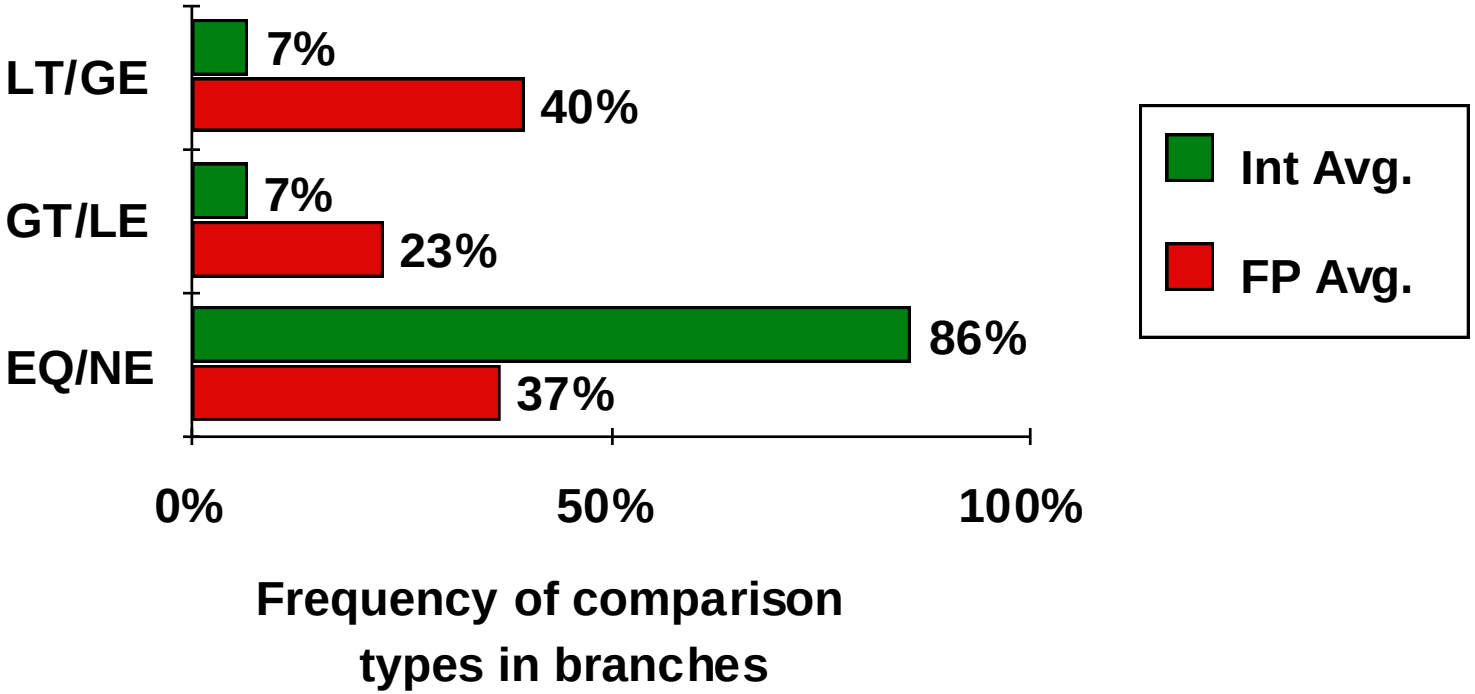
Conditional Branch Distance



- 25% of integer branches are 2 to 4 instructions

Conditional Branch Addressing

- **PC-relative since most branches are relatively close to the current PC**
- **At least 8 bits suggested (±128 instructions)**
- **Compare Equal/Not Equal most important for integer programs (86%)**



MIPS Compare and Branch

- **Compare and Branch**
  - **BEQ rs, rt, offset** **if R[rs] == R[rt] then PC-relative branch**
  - **BNE rs, rt, offset** **<>**
- **Compare to zero and Branch**
  - **BLEZ rs, offset** **if R[rs] <= 0 then PC-relative branch**
  - **BGTZ rs, offset** **>**
  - **BLT** **<**
  - **BGEZ** **>=**
  - **BLTZAL rs, offset** **if R[rs] < 0 then branch and link (into R 31)**
  - **BGEZAL** **>=!**
- **Remaining set of compare and branch ops take two instructions**
- **Almost all comparisons are against zero!**

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
	Equal test; PC relative branch	
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100
	Not equal test; PC relative	
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; 2's comp.	
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; 2's comp.	
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; natural numbers	
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; natural numbers	
jump	j 10000	go to 10000
	Jump to target address	
jump register	jr $31	go to $31
	For switch, procedure return	
jump and link	jal 10000	$31 = PC + 4; go to 10000
	For procedure call	

Signed vs. Unsigned Comparison

R1= `0…00 0000 0000 0000 0001` **two**

R2= `0…00 0000 0000 0000 0010` **two**

R3= `1…11 1111 1111 1111 1111` **two**

- **After executing these instructions:**

```
slt  r4,r2,r1 ;
```
if (r2 < r1) r4=1; else r4=0

```
slt  r5,r3,r1 ;
```
if (r3 < r1) r5=1; else r5=0

```
sltu r6,r2,r1 ;
```
if (r2 < r1) r6=1; else r6=0

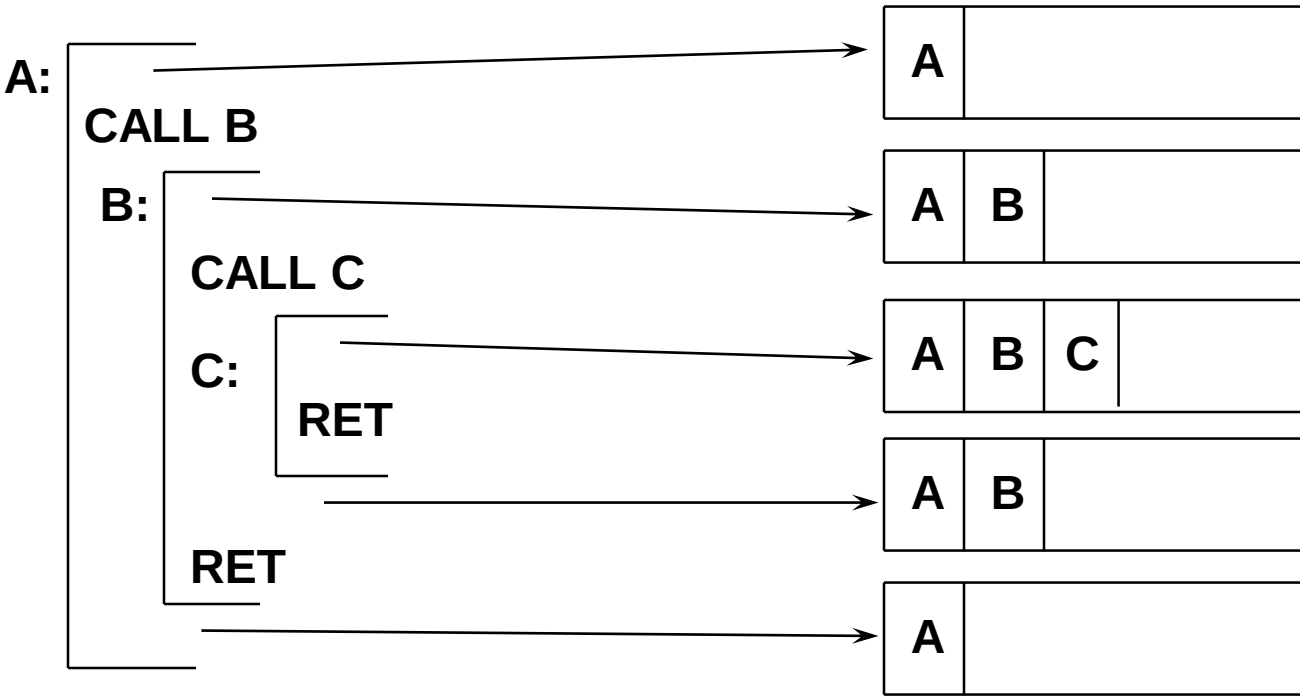
```
sltu r7,r3,r1 ;
```
if (r3 < r1) r7=1; else r7=0

- **What are values of registers r4 - r7? Why?**

r4 = ; r5 = ; r6 = ; r7 = ;

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



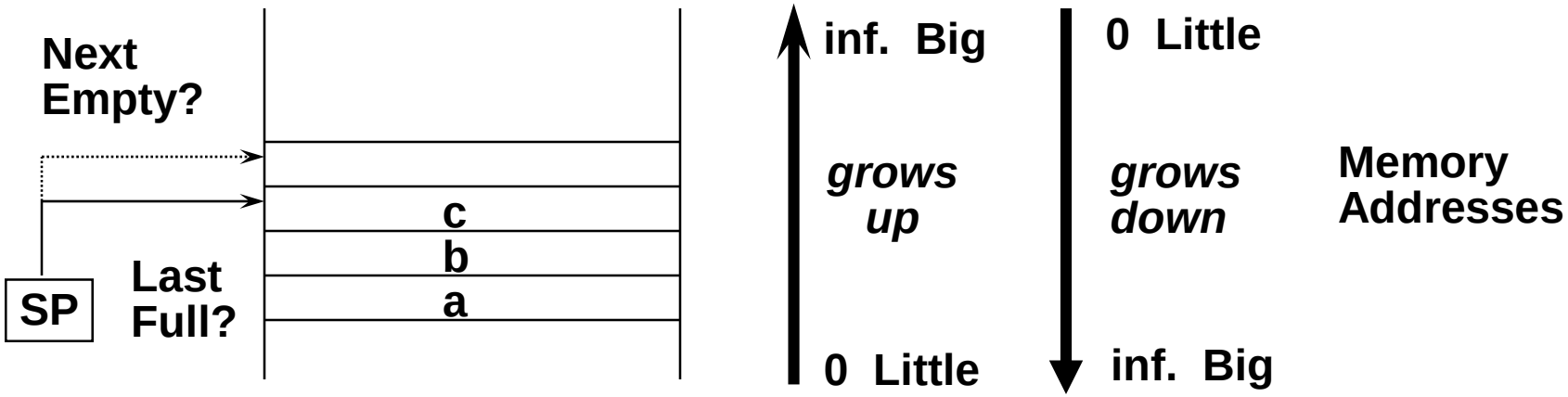
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

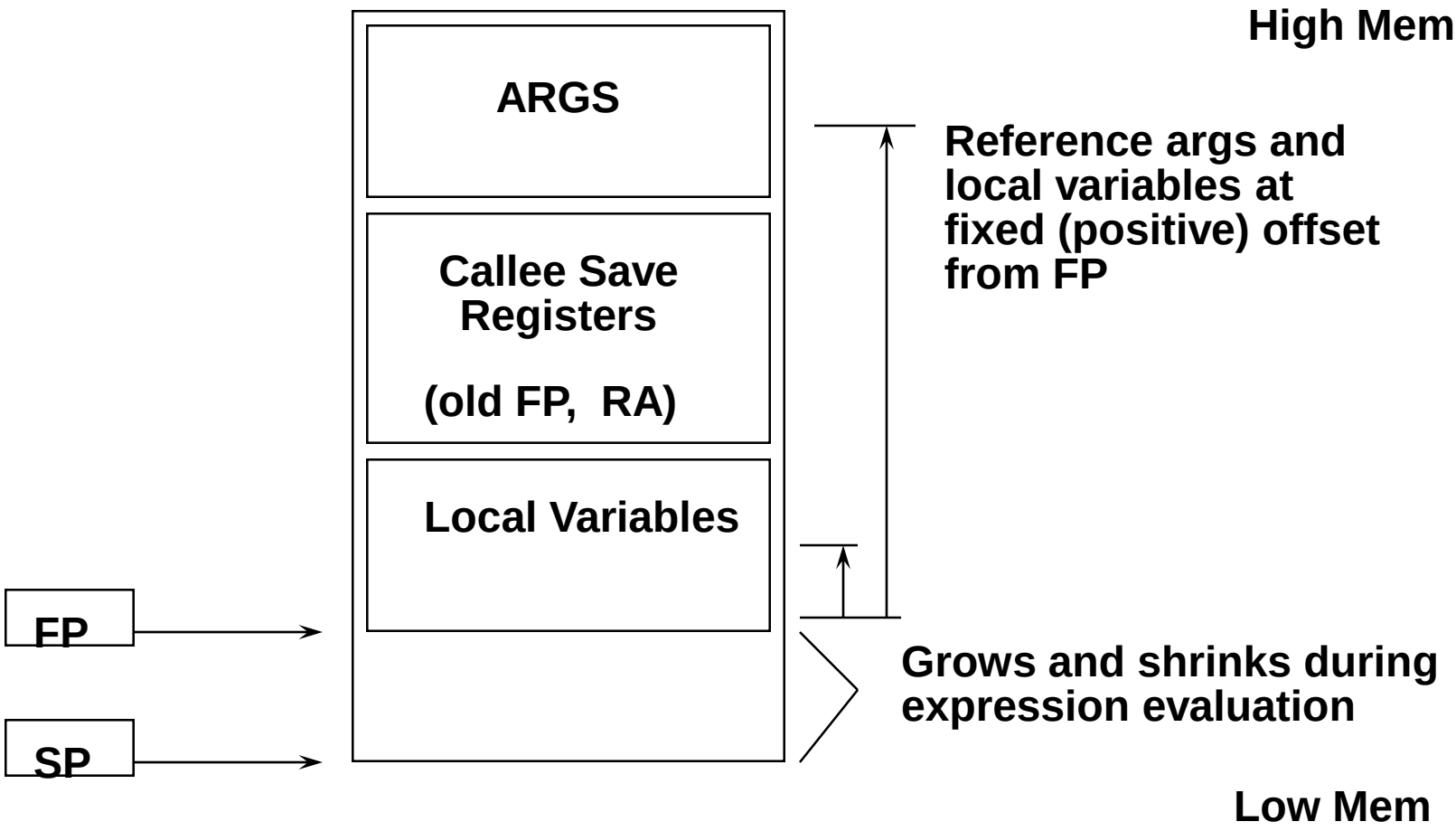
PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

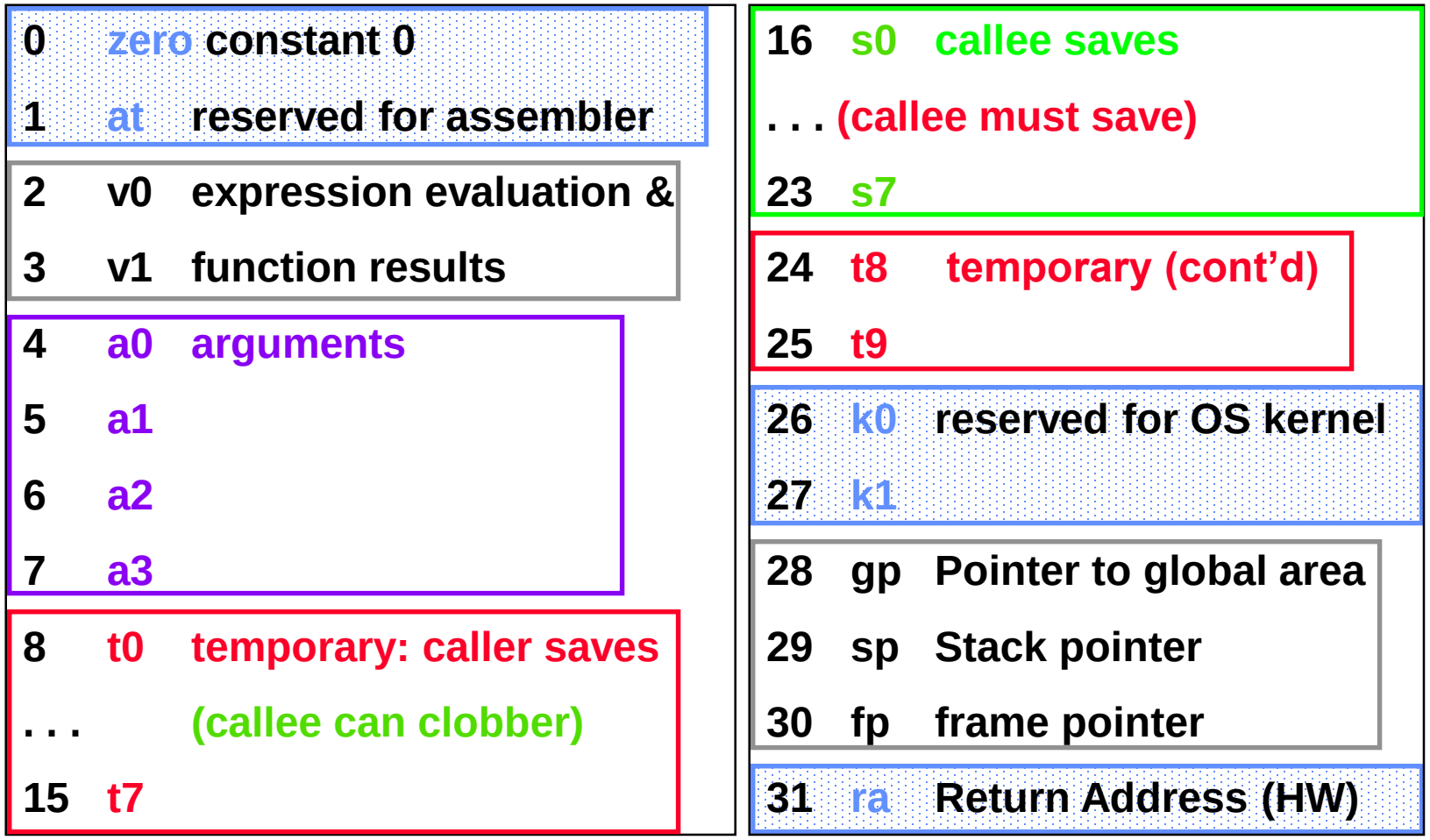
PUSH: Write to Mem(SP)
Increment SP

Call-Return Linkage: Stack Frames



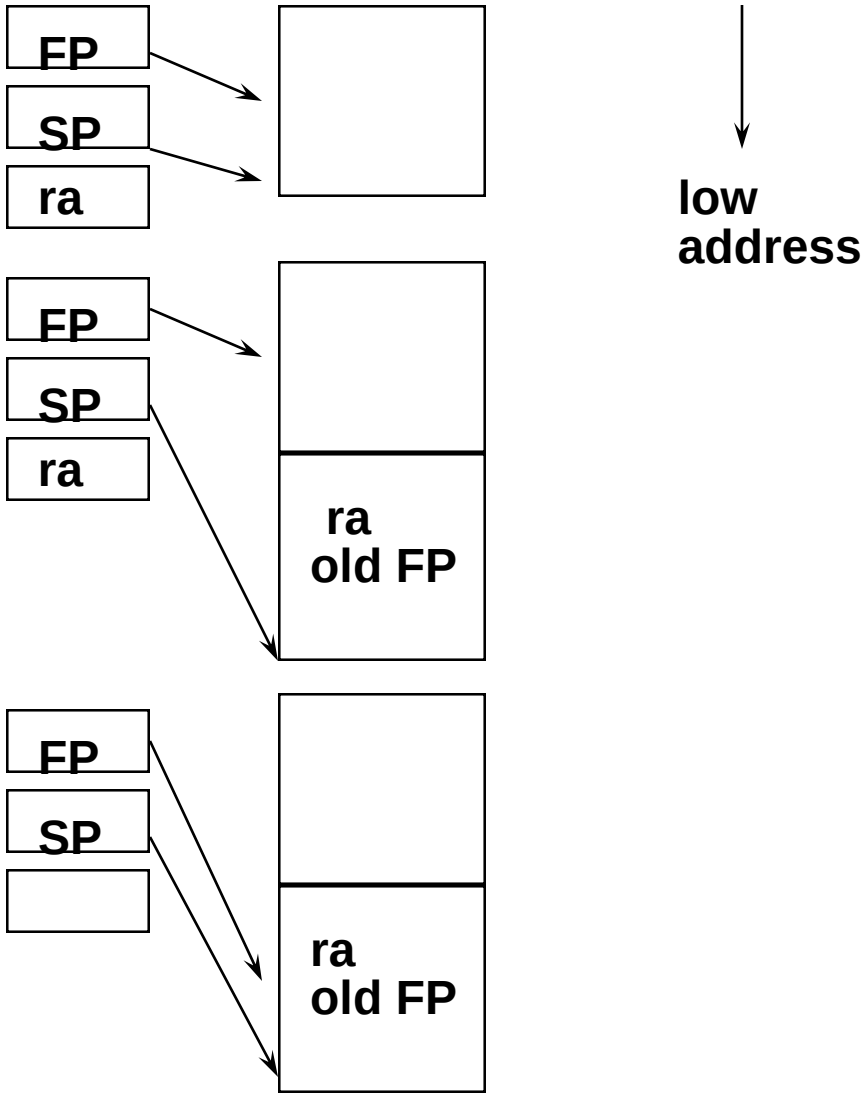
- **Many variations on stacks possible (up/down, last pushed / next)**
- **Compilers normally keep scalar variables in registers, not memory!**

MIPS: Software conventions for Registers



MIPS / GCC Calling Conventions

```
fact:
   addiu          $sp, $sp, -32
   sw     $ra, 20($sp)
   sw     $fp, 16($sp)
   addiu  $fp, $sp, 32
. . .
   sw     $a0, 0($fp)
...
   lw     $31, 20($sp)
   lw     $fp, 16($sp)
   addiu  $sp, $sp, 32
   jr     $31
```



First four arguments passed in registers.

Details of the MIPS instruction set

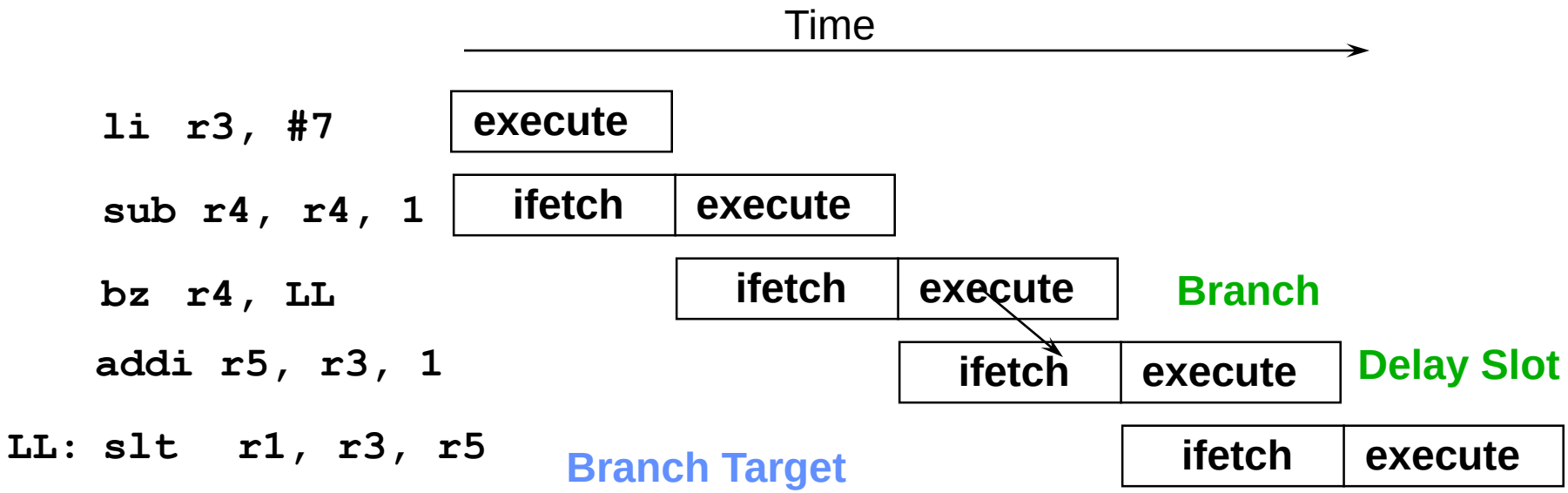
- **Register zero always has the value zero (even if you try to write it)**
- **Branch/jump and link put the return addr. PC+4 or 8 into the link register (R31) (depends on logical vs physical architecture)**
- **All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)**
- **Immediate arithmetic and logical instructions are extended as follows:**
  - **logical immediates ops are zero extended to 32 bits**
  - **arithmetic immediates ops are sign extended to 32 bits (including addu)**
- **The data loaded by the instructions lb and lh are extended as follows:**
  - **lbu, lhu are zero extended**
  - **lb, lh are sign extended**
- **Overflow can occur in these arithmetic and logical instructions:**
  - **add, sub, addi**
  - **it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu**

Delayed Branches

```
        li   r3, #7
        sub  r4, r4, 1
        bz   r4, LL
        addi r5, r3, 1
        subi r6, r6, 2
LL:     slt  r1, r3, r5
```

- **In the "Raw" MIPS, the instruction after the branch is executed even when the branch is taken?**
  - **This is hidden by the assembler for the MIPS "virtual machine"**
  - **allows the compiler to better utilize the instruction pipeline (???)**

Branch & Pipelines

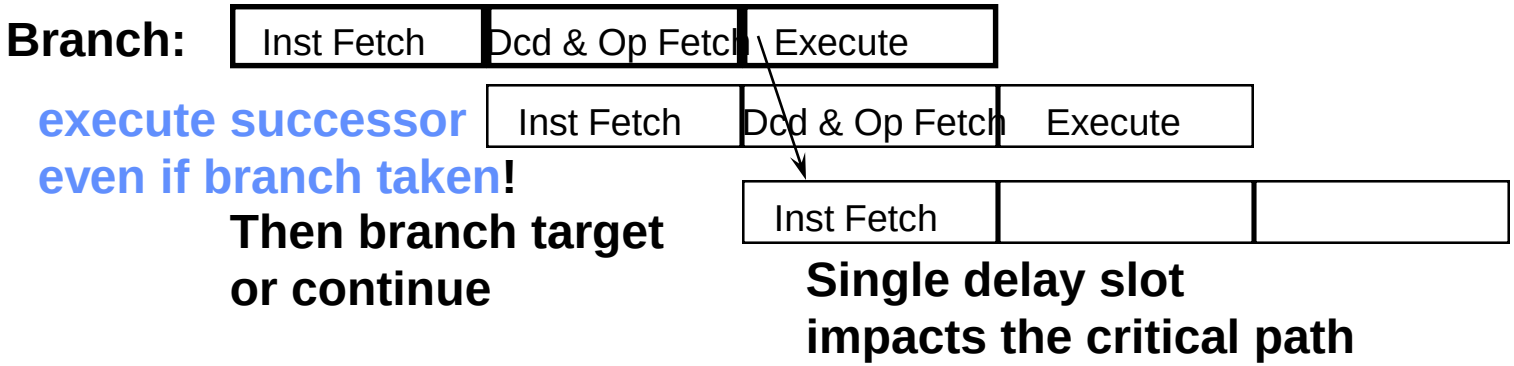


By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?

Filling Delayed Branches



- Compiler can fill a single delay slot with a useful instruction 50% of the time.
  - try to move down from above jump
  - move up from target, if safe

```
        add r3, r1, r2
        sub r4, r4, 1
        bz  r4, LL
        NOP
        ...
LL:     add rd, ...
```

Is this violating the ISA abstraction?

Miscellaneous MIPS I instructions

- **break** — **A breakpoint trap occurs, transfers control to exception handler**
- **syscall** — **A system trap occurs, transfers control to exception handler**
- **coprocessor instrs.** — **Support for floating point**
- **TLB instructions** — **Support for virtual memory: discussed later**
- **restore from exception mask & kernel/user status register** — **Restores previous interrupt mode bits into**
- **load word left/right loads** — **Supports misaligned word**
- **store word left/right stores** — **Supports misaligned word**

Summary: Salient features of MIPS I

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection, scaled
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =,≠
  - no integer condition codes
- **Delayed branch**
  - execute instruction after a branch (or jump) even if the branch is taken

  (Compiler can fill a delayed branch with useful work about 50% of the time)

Summary: Instruction set design (MIPS)

- **Use general purpose registers with a load-store architecture: YES**
- **Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR**
- **Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : YES: 16 bits for immediate, displacement (disp=0 => register deferred)**
- **All addressing modes apply to all data transfer instructions : YES**
- **Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed**
- **Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES**
- **Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16b**
- **Aim for a minimalist instruction set: YES**