计算机组织与系统结构

指令系统设计

（第三讲）

程 旭

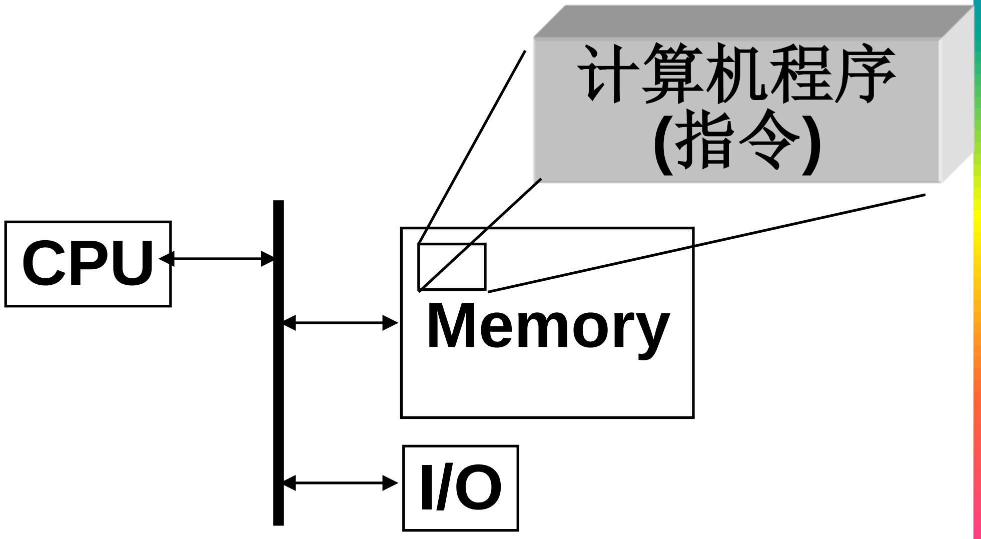
2020.10.29

指令系统体系结构（Instruction Set Architecture）

从程序员来观察

ADD	01010
SUBTRACT	01110
AND	10011
OR	10001
COMPARE	11010
.	.
.	.
.	.

从计算机来观察



Princeton (Von Neumann) 系统结构

--- 数据和指令存放在统一存储器中
(“存储程序计算机”）
(“stored program computer")

--- 程序当作数据

--- 存贮系统的利用（Storage utilization）

--- 单一的存储器接口

Harvard 系统结构

--- 数据 & 指令
存放在不同的存储器中

--- 在某些高性能实现中
具有优势

指令系统设计中的基本问题

--- 应该提供 哪些(以及多少) 操作？
用 **LD/ST/INC/BRN** 已经足够编制任何计算程序
但是并不实用，这主要因为 编出的程序太长!

--- 如何 (以及 多少) 操作数 应该被指明？
大多数操作 是 双值运算(**dyadic**) (例如, **A ⇐ B + C**)
也有一些 是 单值运算 (**monadic**) (例如, **A ⇐ ~B**)

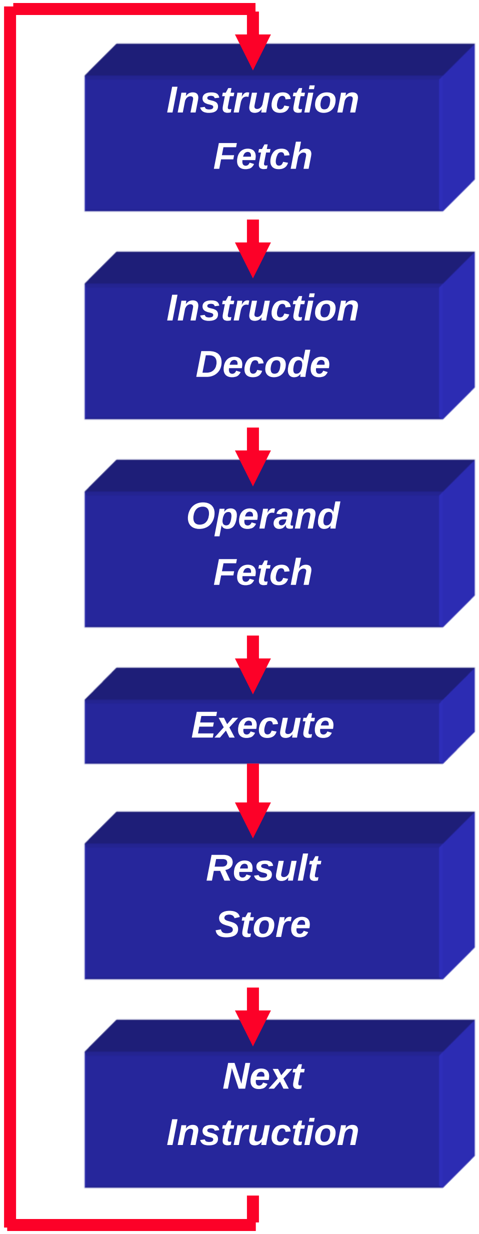
--- 如何将这些指令编码成 一致的指令格式？
指令长度应该为基本数据 / 地址宽度的 倍数！

典型的指令系统:

- ♣ 32位字
- ♣ 基本操作数地址为 32 位长
- ♣ 基本操作数（例如，整数）, 32位长
- ♣ 通常, 指令可以涉及 3 个操作数 (A := B + C)

挑战：用少量的位数，对操作进行编码!

执行周期



从程序存储系统中获得指令

确定所需的动作和指令大小

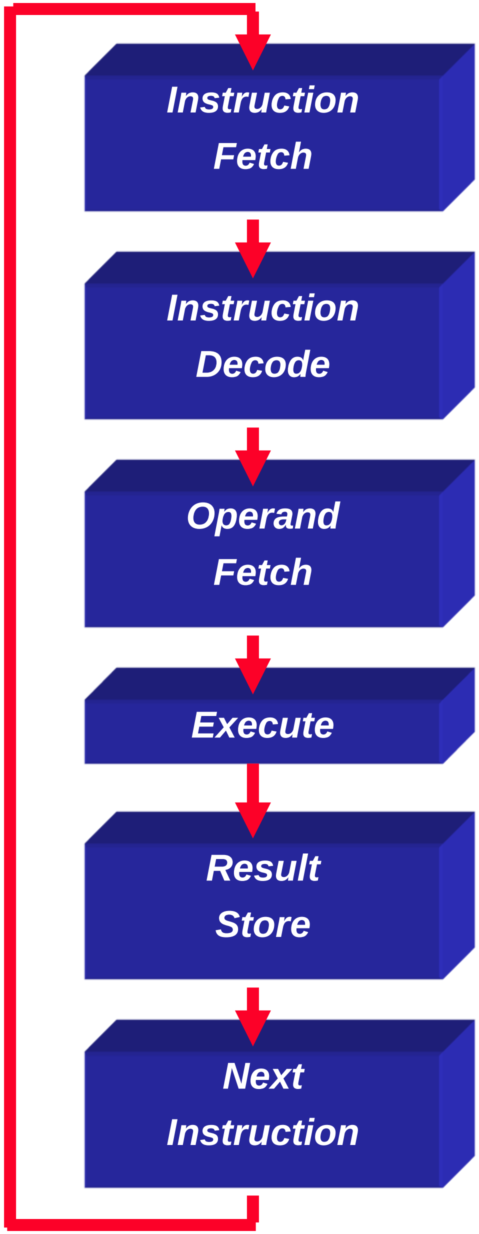
定位并获得操作数数据

计算结果数值或状态

在存储系统中存放结果，以备后用

确定后续指令

必须指定什么？

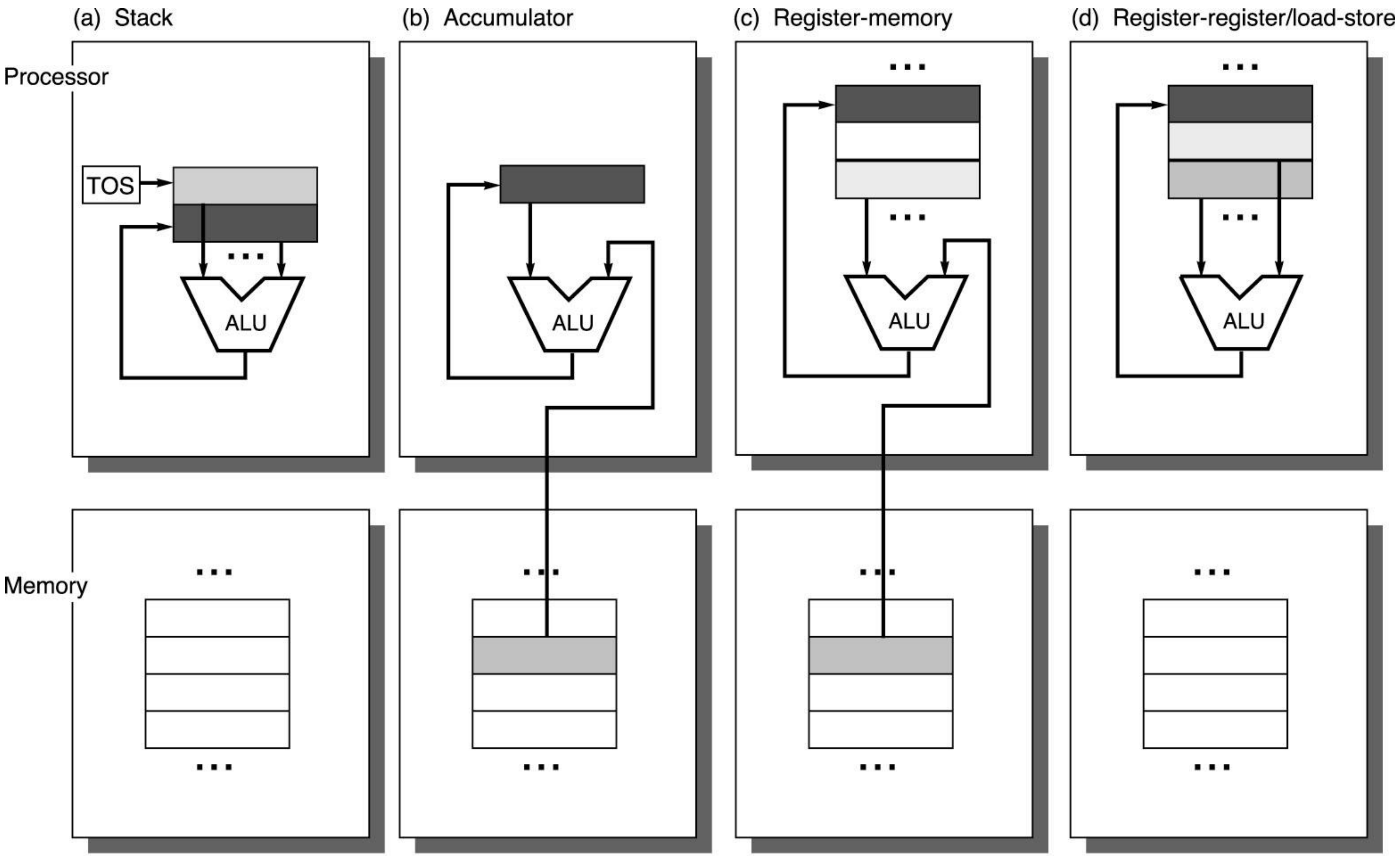


- 指令格式或编码
  - 如何对它译码?
- 对操作数和结果定位
  - 除了存储器之外，还可放在哪里?
  - 有多少个显式操作数?
  - 如何对存储器操作数进行定位?
- 哪些操作数可以或者不可以在存储器中?
- 数据类型和大小
- 操作
  - 支持哪些？
- 后续指令
  - jumps, conditions, branches

- 指令处理必须经过 fetch-decode-execute!

将要揭示的主题

- 对操作数和结果定位
  - 除了存储器之外，还可放在哪里?
  - 有多少个显式操作数?
  - 如何对存储器操作数进行定位?
  - 哪些操作数可以 或者 不可以存放在存储器中?
- 操作类型
- 指令格式或编码
  - 如何对它译码?
- 数据类型和大小
  - 支持哪些?


(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store
Processor
TOS
ALU
ALU
ALU
ALU
Memory

基本的指令系统体系结构种类

累加器（Accumulator）：

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

堆栈（Stack）：

0 address	add	tos ← tos + next

比较：每条指令的字节数？指令数？每条指令的周期数？

基本的指令系统体系结构种类

通用寄存器（General Purpose Register）：

2 address	add A B	Val(A) ← Val(A) + Val(B)
3 address	add A B C	Val(A) ← Val(B) + Val(C)

装入 / 存储（Load/Store）：

3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

比较：每条指令的字节数？指令数？每条指令的周期数？

比较　指令的数量

四类指令系统，完成C = A + B 的代码序列:

堆栈	累加器	寄存器 (寄存器−存储器)	寄存器 (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

通用寄存器结构　占优势

1975年以来，所有的机器都使用通用寄存器

寄存器的优点

- 寄存器比存储器　快
- 寄存器便于编译器使用
  - 例如, (A*B) – (C*D) – (E*F) 可以按任意顺序计算乘法 vs. 堆栈
- 寄存器可以保留变量
  - 可减少存储器通信量, 因而可以加速程序执行 (因为寄存器比存储器要快！)
  - 可改进代码密度 (因为指定寄存器比指定存储器位置所需的位数要少）

寄存器使用的实例

每条典型的ＡＬＵ指令的存储器地址的数目	每条典型的ＡＬＵ指令中操作数的最多数目	*实例*
0	**3**	**SPARC, MIPS, Precision Architecture, Power PC**
1	**2**	**Intel 80x86, Motorola 68000**
2	**2**	**VAX (还有 3操作数格式)**
3	**3**	**VAX (还有 2操作数格式)**

存储器操作数 / 操作数　数目的利弊

- 寄存器–寄存器: 0 存储器操作数/指令, 3 (寄存器)操作数/指令

  + 简单、 固定长度指令编码； 简单的代码生成模式

    指令执行需要相近的时钟数目

  - 与 指令中可以访问存储器单元的结构 相比，需要更高的指令总数

    一些指令较短，编码位数可能会有浪费

存储器操作数 / 操作数　数目的利弊

- 寄存器–存储器 (1,2)

利　数据无需预先装入就可以被访问.
指令格式有易于编码的倾向，可产生较好的指令密度

弊　操作数不等价，这是因为二元操作中的一个源操作数将被破坏.

每条指令都要编码一个寄存器号和一个存储器地址，将可能限制寄存器的数目.

根据不同的操作数位置，每条指令的时钟数可能会多样化.

存储器操作数／操作数　数目的利弊

- 存储器–存储器 (3,3)
  - +最紧凑. 不会因为中间存储而浪费寄存器.
  - –指令大小会有很大变化, 特别是对于 3 操作数指令
    而且, 每条指令的工作也有很大变化.
    存储器访问会产生存储器瓶颈!

指令分类的总结

- 猜测：新的指令系统体系结构需要使用通用寄存器

- 流水 => 猜测：

它将使用通用寄存器指令系统体系结构中的装入／存储形式

存储器寻址

- **1980**年以来，几乎每台计算机的最小寻址单元都为 **8**位（字节）
- 指令系统体系结构设计的**2**个问题**:**
  - 由于可以用四次字节装入方式，从连续的字节地址读入**32**位字，也可以用字装入方式，从单一字节地址，读入**32**位字，那么字节地址如何映射到字?
  - 一个字能否存放与任何字节边界?

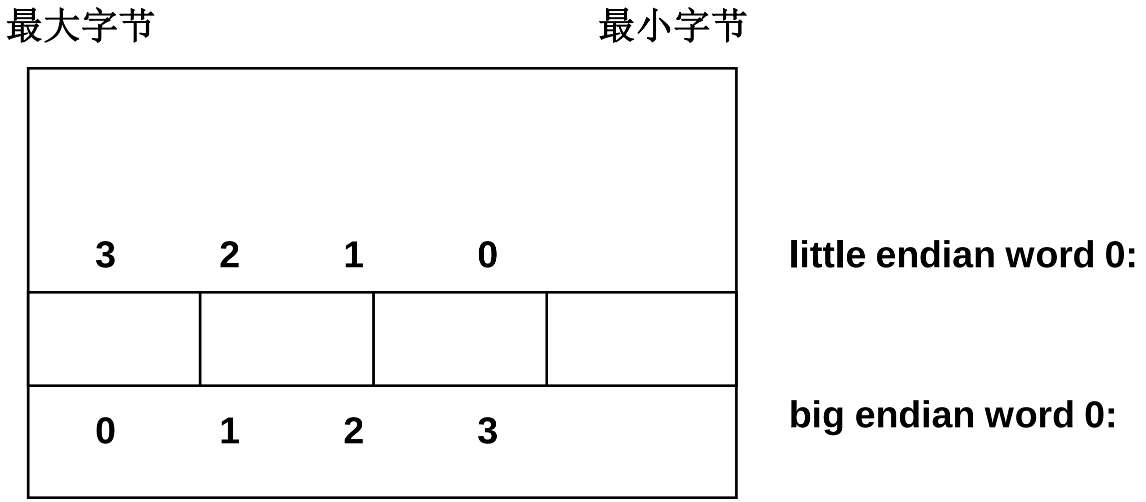
寻址对象

Big Endian:　最大量的地址（address of most significant）

- IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

Little Endian:　最小量的地址（address of least significant）

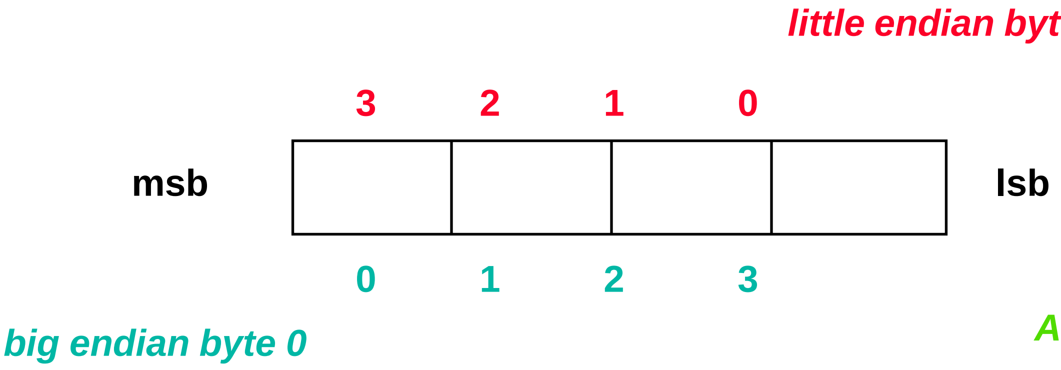
- Intel 80x86, DEC Vax

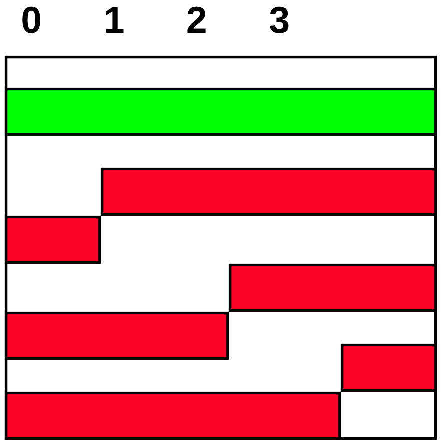


对准（Alignment）：要求对象只能安放于其大小的倍数的地址上！

寻址对象: Endianess and Alignment

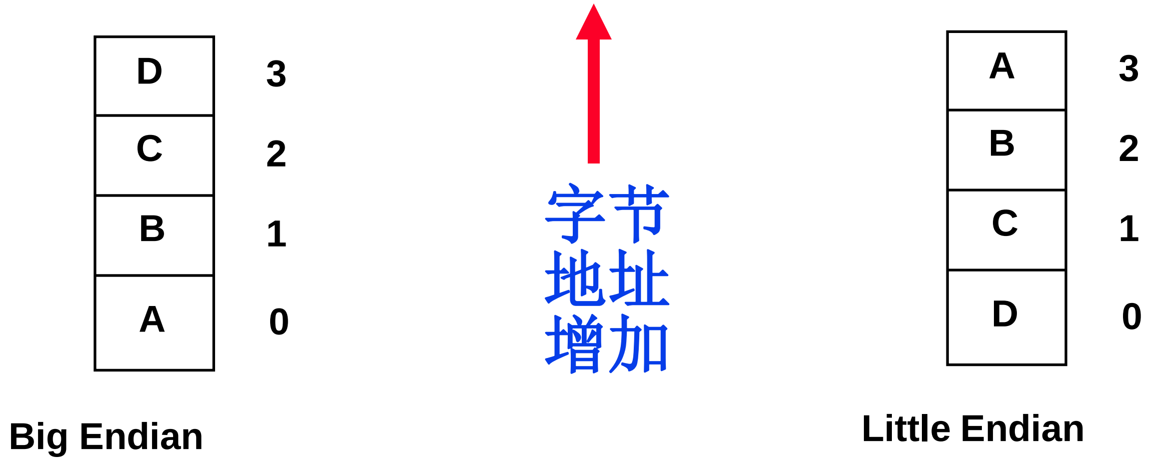
- Big Endian: address of most significant
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)





Alignment: require that objects fall on address that is multiple of their size.

字节交换问题（Byte Swap Problem）



D	3
C	2
B	1
A	0

Big Endian

字节
地址
增加

A	3
B	2
C	1
D	0

Little Endian

ABCD

当在具有不同定义的机器之间，传输存储字时，
必须变换字节的顺序，以保证数据的成功复制

每个系统都是完备的，但是当它们之间需要通信时，就会产生问题！

寻址方式

寻址方式	示例	含义
Register	Add R4,R3	R4←R4+R3
Immediate	Add R4,#3	R4←R4+3
Displacement	Add R4,100(R1)	R4←R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4←R4+Mem[R1]
Indexed	Add R3,(R1+R2)	R3←R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1←R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1←R1+Mem[Mem[R3]]
Post-increment	Add R1,(R2)+	R1←R1+Mem[R2]; R2←R2+d
Pre-decrement	Add R1,–(R2)	R2←R2–d; R1←R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1←R1+Mem[100+R2+R3*d]

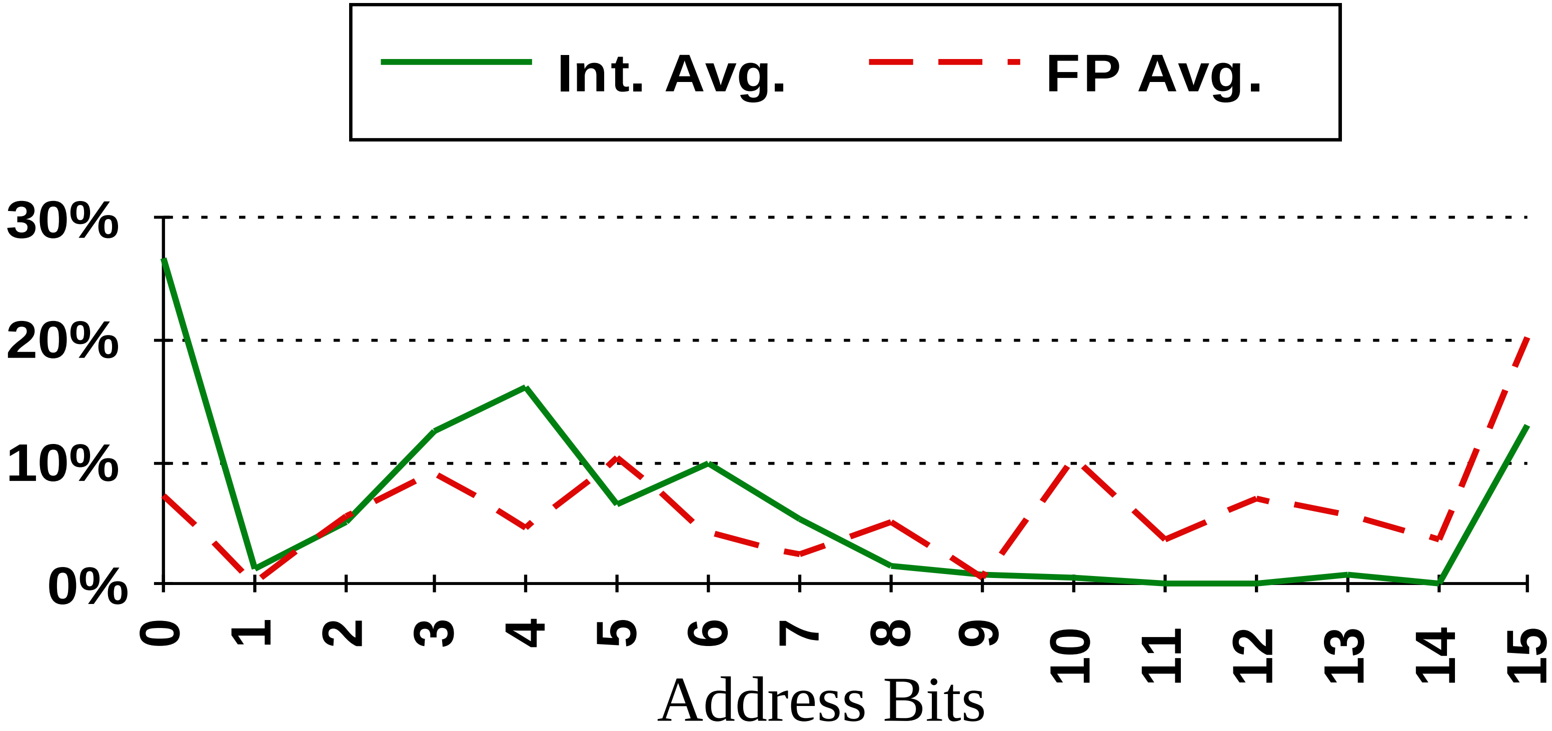
寻址方式的使用情况

3 个程序

--- Displacement	平均 42%, 变化范围：	32% to 55%
--- Immediate	平均 33%,变化范围：	17% to 43%
--- Register deferred (indirect)	平均 13%,变化范围：	3% to 24%
--- Scaled	平均 7%，变化范围：	0% to 16%
--- Memory indirect	平均 3%，变化范围：	1% to 6%
--- 其他	平均 2%，变化范围：	0% to 3%

75% displacement & immediate
85% displacement, immediate & register indirect

Displacement 地址大小



- SPECint92中 5个程序的平均 和 SPECfp92中 5个程序的平均
- X轴的单位是 2的幂次: 4 => 地址 > 2^3 (8) 和 < 2^4 (16)
- 1% 的地址 > 16位

立即数（Immediate）大小

- 50% ~ 60% 少于等于 8 位
- 75% ~ 80% 少于等于 16 位

寻址方式总结

- 非常重要的数据寻址方式:
  Displacement, Immediate,
  Register Indirect

- 位移数（Displacement）的大小应该为 12～16 位

- 立即数（Immediate）的大小应该为 8～16 位

典型操作

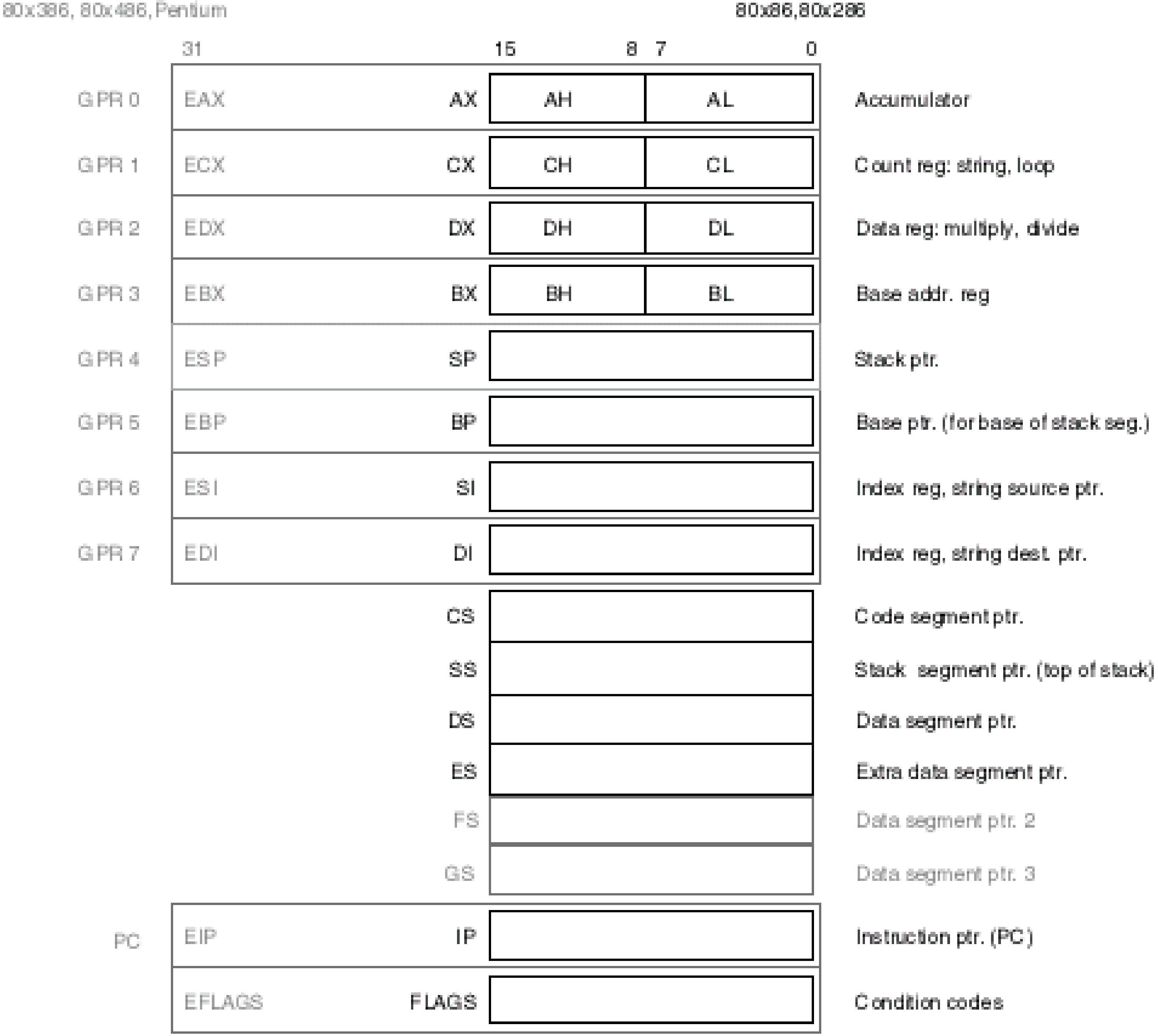
数据移动 Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
算术运算 Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
逻辑运算 Logical	not, and, or, set, clear
移位（Shift）	shift left/right, rotate left/right
控制 (Jump/Branch)	unconditional, conditional
子程序链接（Subroutine Linkage）	call, return
中断（Interrupt）	trap, return
同步（Synchronization）	test & set (atomic r-m-w)
串（String）	search, translate
图形处理Graphics (MMX)	parallel subword ops (4 16bit add)

80x86中使用最多的10条指令

次序	指令	整数平均百分比
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%
	总数	**96%**

简单指令支配指令的频率分布！

80X86中的定点寄存器

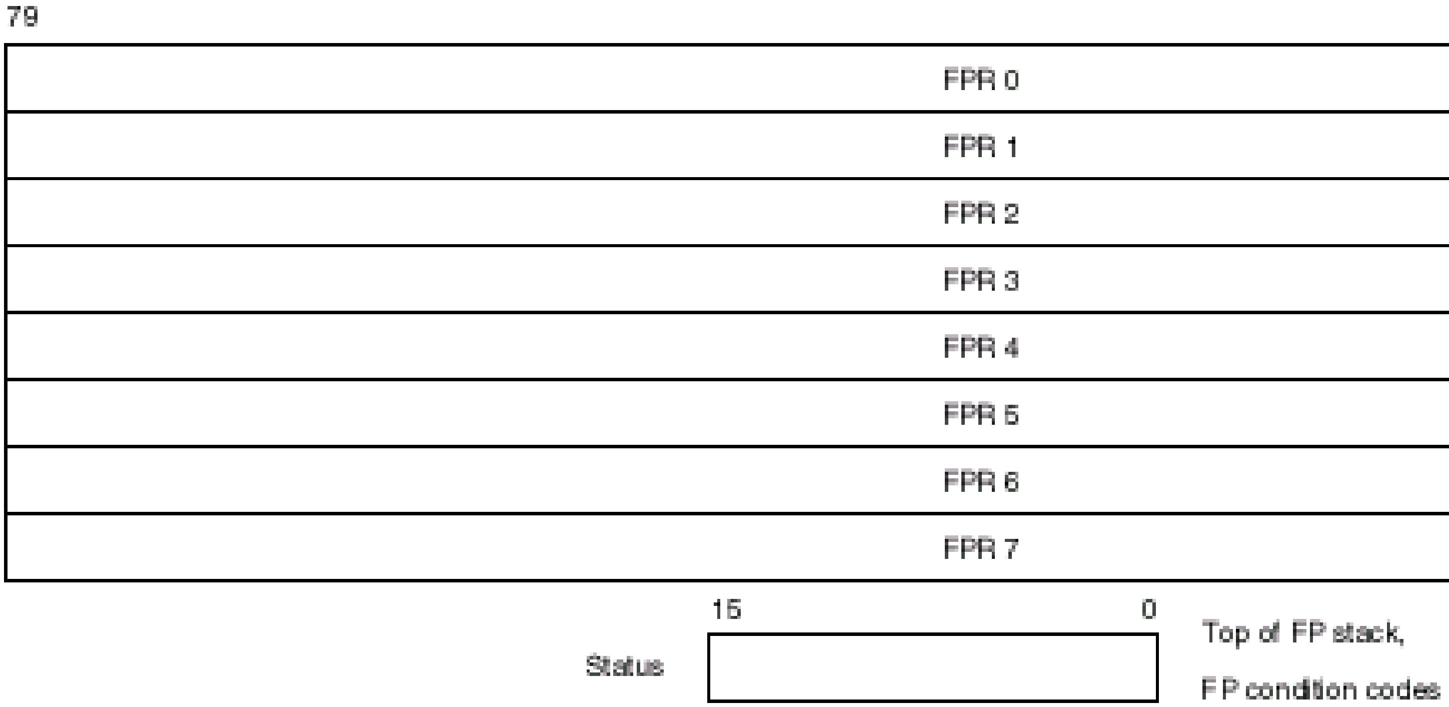


80x386, 80x486, Pentium

80x86,80x286

	31		15 8	7 0	
GPR 0	EAX	AX	AH	AL	Accumulator
GPR 1	ECX	CX	CH	CL	Count reg: string, loop
GPR 2	EDX	DX	DH	DL	Data reg: multiply, divide
GPR 3	EBX	BX	BH	BL	Base addr. reg
GPR 4	ESP	SP			Stack ptr.
GPR 5	EBP	BP			Base ptr. (for base of stack seg.)
GPR 6	ESI	SI			Index reg, string source ptr.
GPR 7	EDI	DI			Index reg, string dest. ptr.
		CS			Code segment ptr.
		SS			Stack segment ptr. (top of stack)
		DS			Data segment ptr.
		ES			Extra data segment ptr.
		FS			Data segment ptr. 2
		GS			Data segment ptr. 3
PC	EIP	IP			Instruction ptr. (PC)
	EFLAGS	FLAGS			Condition codes

８０X86中的浮点寄存器



x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

测试条件码的方法

- ♣ 条件码

  处理器状态位：是在算术指令的执行中隐式设置的 (也可能在数据移动中) 或者 由比较或测试指令显式设置。

  例如:

  ```
  add r1, r2, r3
  bz label
  ```

- ♣ 条件寄存器

  例如:

  ```
  cmp r1, r2, r3
  bgt r1, label
  ```

- ♣ 比较并转移

  例如:

  ```
  bgt r1, r2, label
  ```

条件码

以附带的形式设置条件码可以减少指令总数

```
X:     .
       .
       .
    SUB  r0, #1, r0
    BRP  X
```

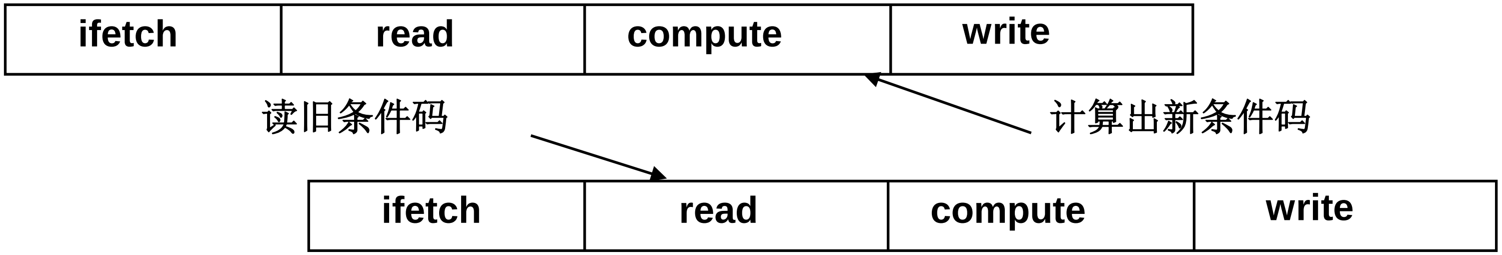
vs.

```
X:     .
       .
       .
    SUB  r0, #1, r0
    CMP  r0, #0
    BRP  X
```

但是，也会产生一些弊端：

--- 并非所有的指令都设置条件码
哪些设置、哪些不设置 常常 很混乱！
例如， 移位指令设置 进位 位

--- 在设置条件码的指令和测试该条件码的指令之间存在依赖关系：
为了重叠它们的执行，可能需要在它们之间插入一条不改变该条件码的指令



ifetch	read	compute	write

读旧条件码 计算出新条件码

ifetch	read	compute	write

转移(Branches)

有条件地控制传输（Conditional control transfers）

4种基本条件:

N -- 负数（negative）

Z -- 零（zero）

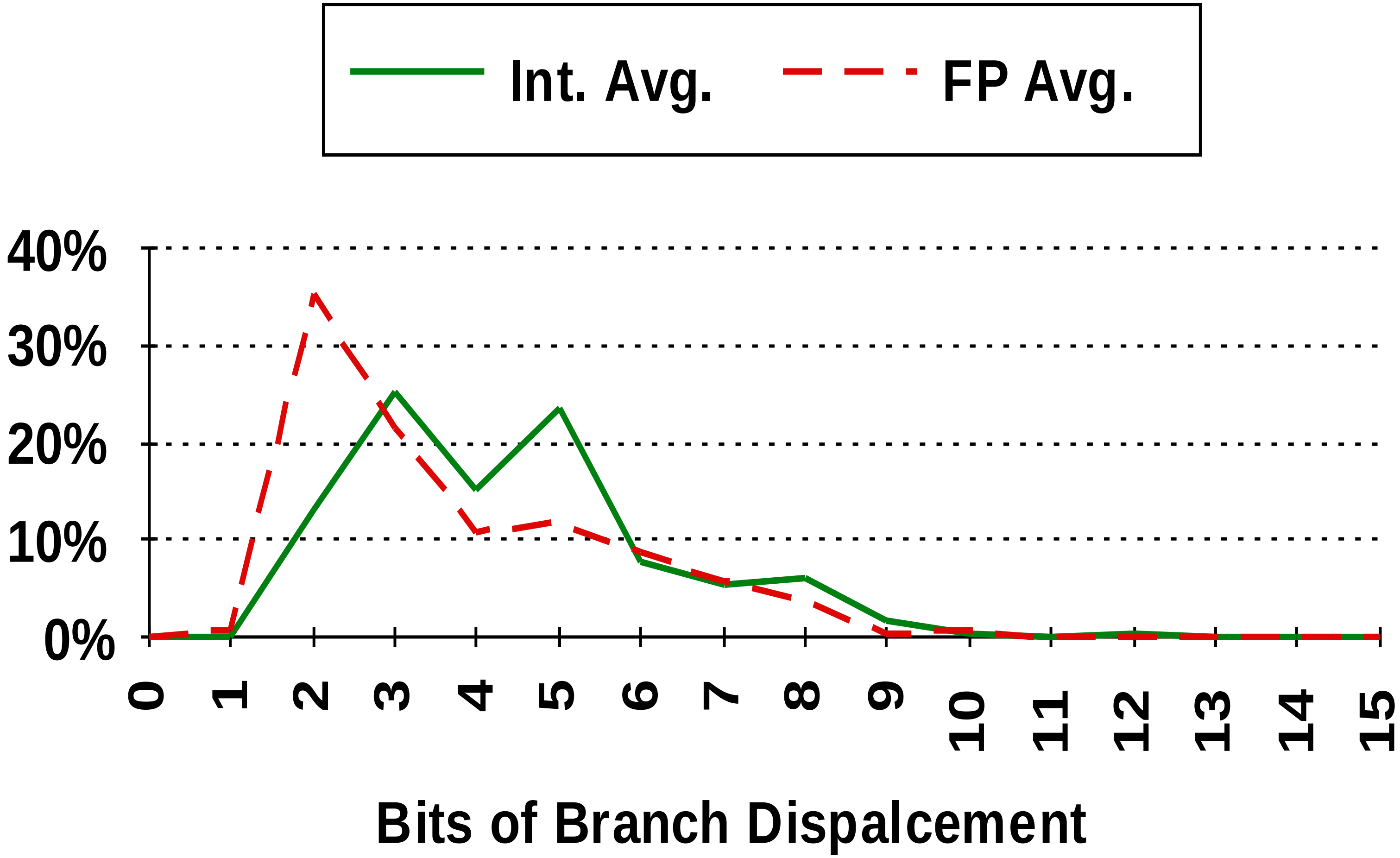
V -- 溢出（overflow）

C -- 进位（carry）

4种基本条件的16种组合:

Always	Unconditional
Never	NOP
Not Equal	~Z
Equal	Z
Greater	~[Z + (N ⊕ V)]
Less or Equal	Z + (N ⊕ V)
Greater or Equal	~(N ⊕ V)
Less	N ⊕ V
Greater Unsigned	~(C ⊕ Z)
Less or Equal Unsigned	C ⊕ Z
Carry Clear	~C
Carry Set	C
Positive	~N
Negative	N
Overflow Clear	~V
Overflow Set	V

条件转移的距离



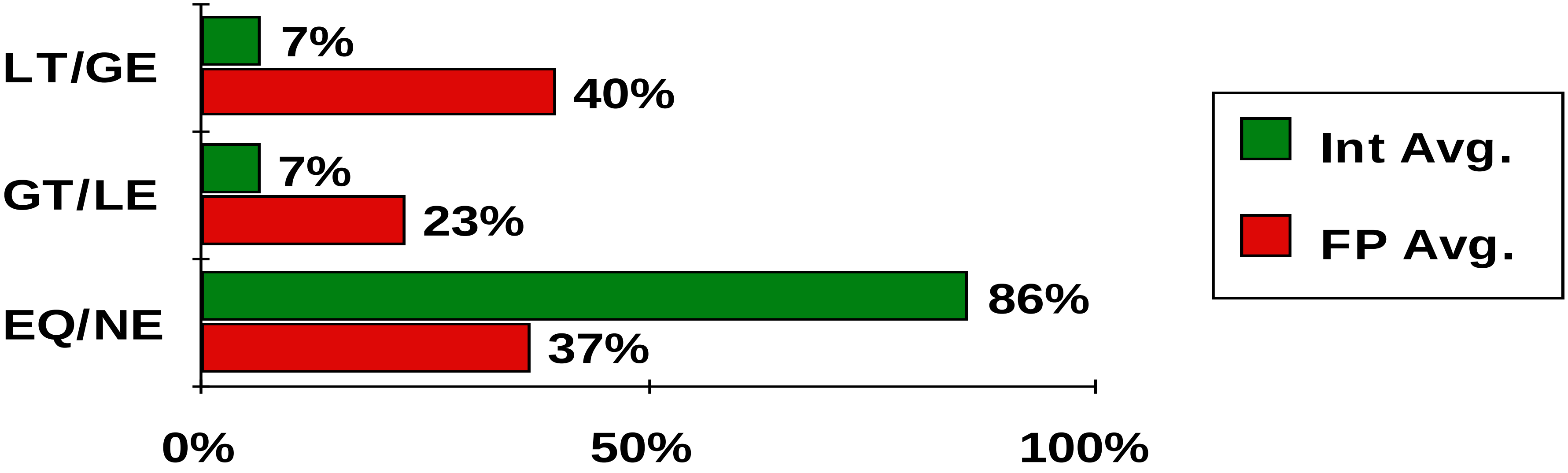
- Distance from branch in instructions   i => <2^{i-1} & > 2^{i-2}
- 25% of integer branches are > 2 & < 4 or -2 to -4

条件转移寻址

♣ PC-相关（PC-relative），因为 大多数条件转移是 相对于当前的PC地址而言的

- 建议：至少 8 位 (即：128条指令)

- 对于整数程序，Compare Equal/Not Equal 是非常重要的！



Frequency of comparison types in branches

操作总结

♣ 支持如下简单指令, **因为这些指令将支配执行的指令数目:**

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
return

数据类型（Data Types）

位: 0, 1

位串: 特定长度的一些位的序列

4 位　是　半字节　（a nibble）
8 位　是　一个字节　（a byte）
16 位　是　半字　（half-word） (VAX: 字)
32 位　是　一个字　(word) (VAX: 长字)

字符:

ASCII　7 位码
EBCDIC　8 位码

十进制:

数字 0-9 编码为 0000b ~ 1001b
每8位字节包括两个十进制数

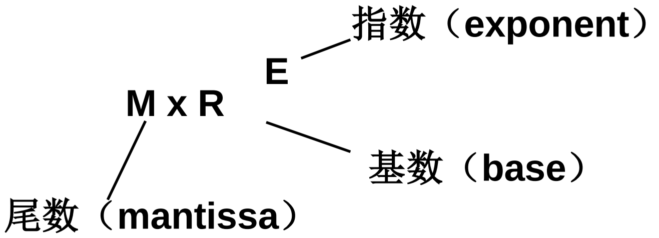
整数:

原码（Sign & Magnitude）: 0X vs. 1X
反码（1's Complement）:　0X vs. 1(~X)
补码（2's Complement）:　0X vs. (1's comp) + 1

正数都相同
前2个都有2个零
通常选择最后一种

浮点:

Single Precision
Double Precision
Extended Precision

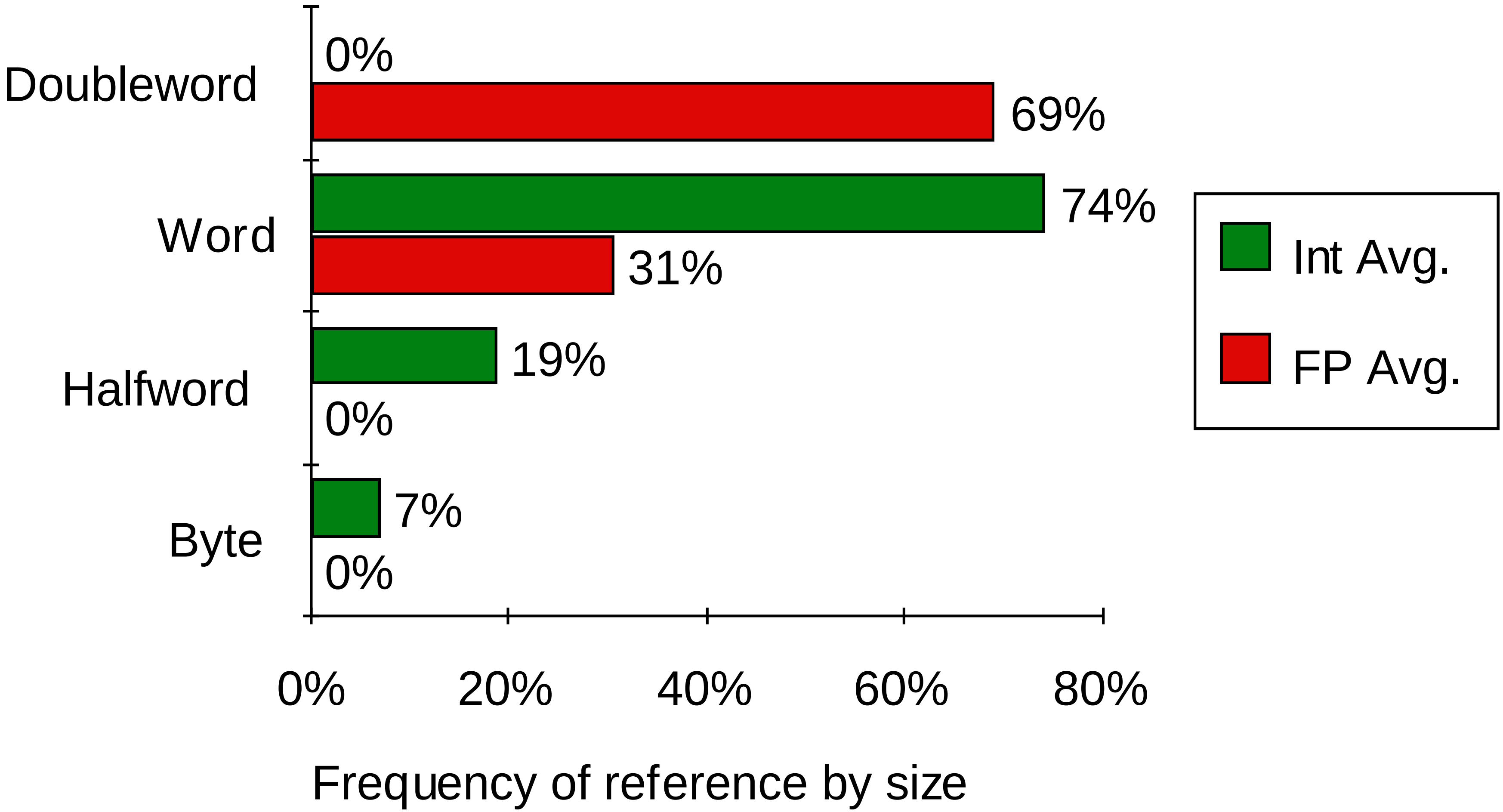


$M \times R^{E}$

指数（exponent）
基数（base）
尾数（mantissa）

具有多少个正负数?
小数点在哪里?
正负指数表示如何?

操作数大小的使用情况



- 支持如下数据大小和类型：

  8位、16位、32位整数 以及

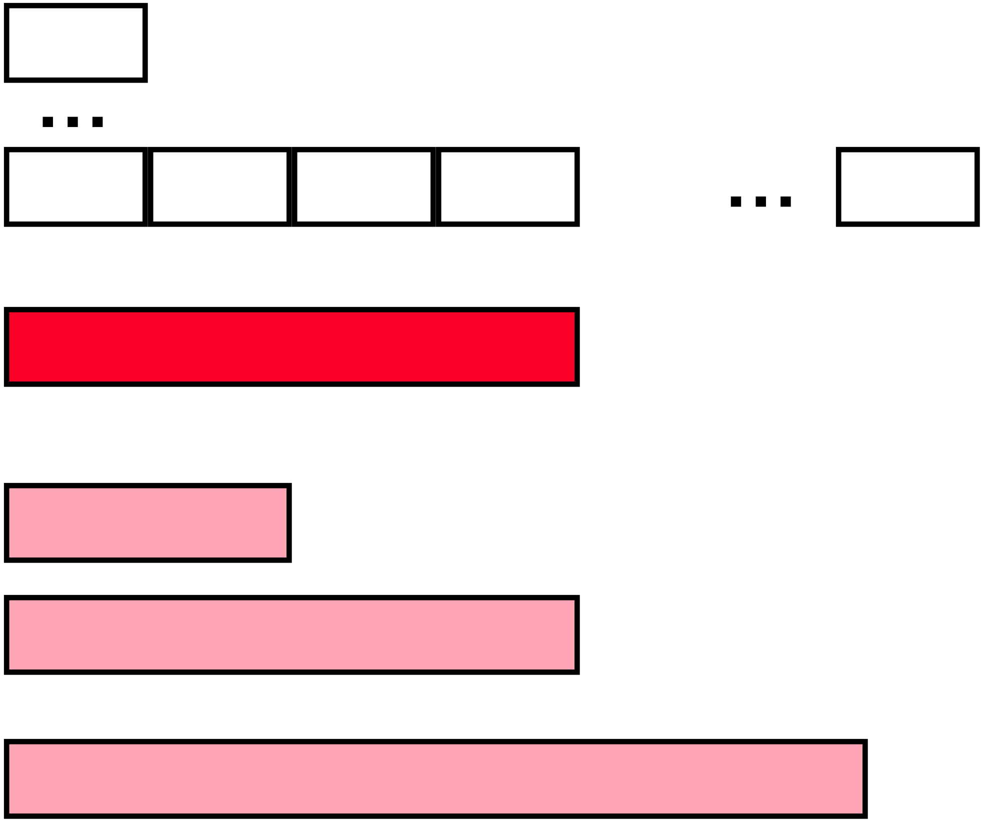
  32位和64位 IEEE 754 浮点数

指令格式

- 如果每条指令都有许多存储器操作数，
  并有多种寻址方式，那么
  每个操作数就需要一个地址描述符
  （**Address Specifier**）

- 如果是**load-store** 机器， 每条指令最多有 **1** 地址，
  并且只有 **1** 或 **2**种寻址方式，那么
  就可以**将寻址方式编码到操作码中**

指令格式的普通例子



可变长度:

固定:

混合:

指令格式总结

- ♣ 如果代码大小至关重要，那么，

使用可变长度指令

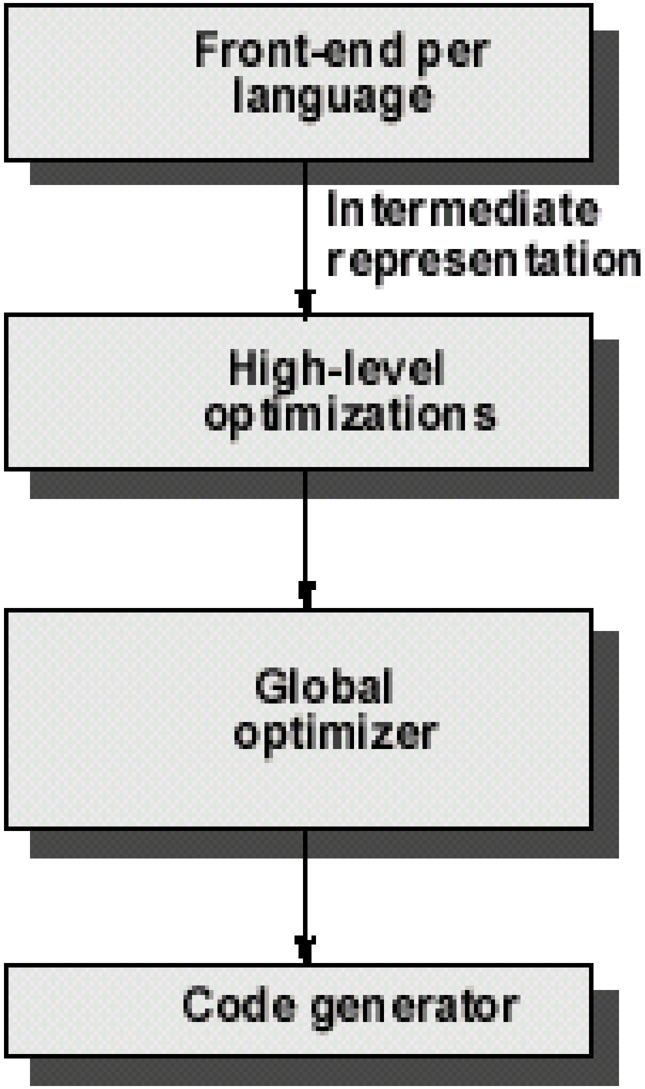
- ♣ 如果性能 至关重要， 那么

使用固定长度指令

- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density

- Some architectures actually exploring on-the-fly decompression for more density.

现代编译技术概貌



Dependencies		Function
Language dependent; machine independent	Front-end per language	Transform language to common intermediate form
	Intermediate representation	
Somewhat language dependent, largely machine independent	High-level optimizations	For example, procedure inlining and loop transformations
Small language dependencies; machine dependencies slight (e.g., register counts/types)	Global optimizer	Including global and local optimizations + register allocation
Highly machine dependent, language independent	Code generator	Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

编译器与指令系统体系结构

- ♣ 易于编译
  - ♥ 正交性（orthogonality）：没有专用寄存器, 特殊情况少, 任何数据类型或指令类型都可使用所有操作数模式
  - ♥ 完全性（completeness）: 支持广泛的操作和目标应用
  - ♥ 规则性（regularity）：没有对指令场位(field)含义的重载(overload)情况
  - ♥ 高效性（streamlined）：易于确定资源需求
- ♣ 寄存器分配也非常重要

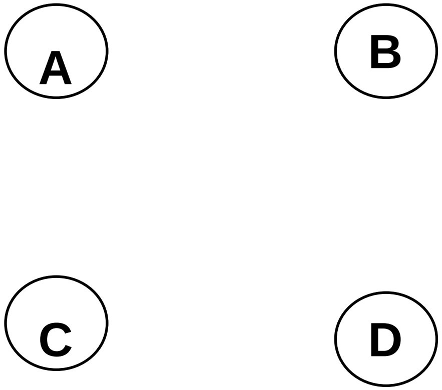
现代寄存器分配

- 将参数(args)和局部变量(local variables)保留在寄存器中

- 利用“图染色(graph coloring)”算法，分配寄存器

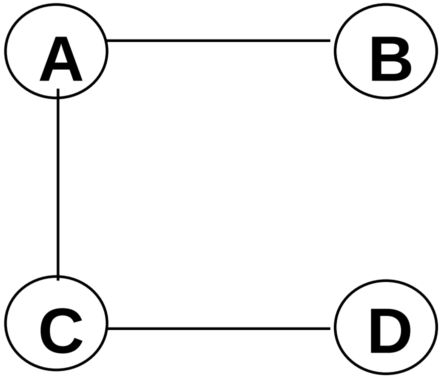
- 如果至少有16个寄存器，工作良好。

```
A=
B=
... B ...
C =
... A ...
D = ...
 ... D ...
 ....C...
```



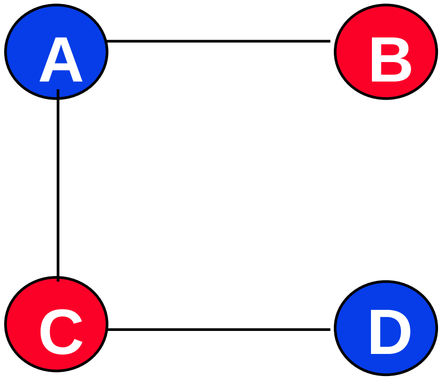
“图染色(graph coloring)”算法

```
A=
B=
... B ...
C =
... A ...
D = ...
 ... D ...
 ....C...
```



Interference graph

The arcs between the nodes show where the ranges of usage for variables (called live ranges) overlap.



```
R1=
R2=
... R2 ...
R2 =
... R1 ...
R1 = ...
 ... R1 ...
 ....R2...
```

对编译器的考虑 总结

- 提供 至少16个 **通用寄存器**，
  附加 单独的 **浮点寄存器**

- 确信 **所有的寻址方式** 都可以适用于
  所有的数据传输指令

- 瞄准
  最低限要求的指令系统
  （minimalist instruction set）

指令系统测度（Instruction Set Metrics）

设计时测度（Design-time metrics）：

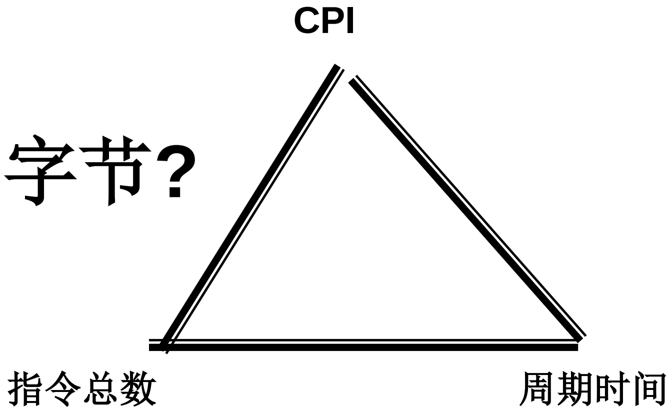
- ♥ 可以 在多长时间内， 以多少成本， 实现它?
- ♥ 它是否可编程? 是否易于编译?

静态测度（Static Metrics）：

- ♥ 程序占用多少字节的存储器?

动态测度（Dynamic Metrics）：

- ♥ 执行了多少条指令?
- ♥ 为执行该程序，处理器取出了多少字节?
- ♥ 每条指令需要多少时钟?



最佳测度：**执行程序的时间！**

注: 这依赖于指令系统、处理器组织，以及编译技术。

小结: ISA

- 使用通用寄存器的load-store 结构；
- 支持如下寻址方式：displacement (with an address offset size of 12 to 16 bits)、 immediate (size 8 to 16 bits), 以及 register deferred;
- 支持如下简单指令（因为它们决定执行的指令总数）：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, 以及return;
- 支持如下数据大小和类型：8位、16位、32位整数； 以及 32位和 64位 IEEE 754 浮点数
- 如果看重性能，就使用 固定指令编码方案
  如果看重代码大小，就使用 可变指令编码方案
- 提供至少16个通用寄存器，以及单独的浮点寄存器;
- 确信所有的寻址方式都可以用于所有的数据传输指令;
- 瞄准最低限要求的指令系统

MIPS Instruction Set Architecture

指令（Instructions）：

- 机器语言的字词
- 比高级语言更加简单、原始
  例如，没有复杂的控制流
- 限制性非常强
  例如：MIPS算术运算指令

- 更课程我们将基于MIPS指令系统体系结构
  - 与二十世纪八十年代后的许多结构都很类似：NEC, Nintendo, Silicon Graphics, Sony

设计目标：更高性能、更低成本、更少设计周期

MIPS 算术指令

- 所有算术指令都有 3 个操作数
- 操作数的次序是固定的（目标操作数领先）

示例：

C代码：　　A = B + C

MIPS代码:　add $s0，$s1，$s2

(编译器完成寄存器与变量的关联)

MIPS arithmetic

- **Design Principle: simplicity favors regularity.** **Why?**
- **Of course this complicates some things...**

C code:

```
A = B + C + D;
E = F - A;
```

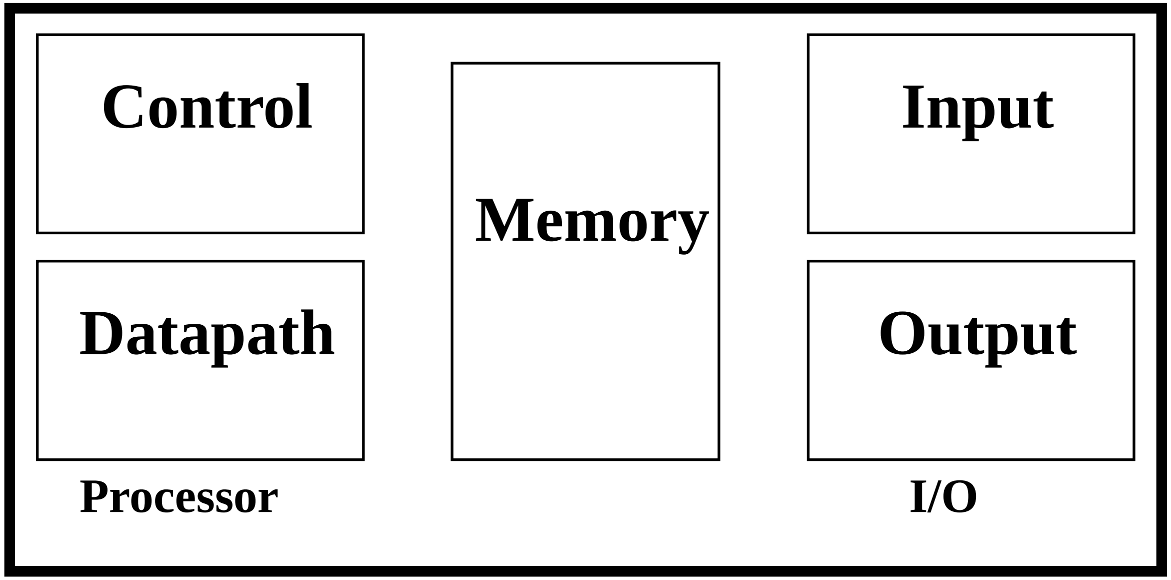
MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- **Operands must be registers, only 32 registers provided**
- **Design Principle: smaller is faster.** **Why?**

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



Memory Organization

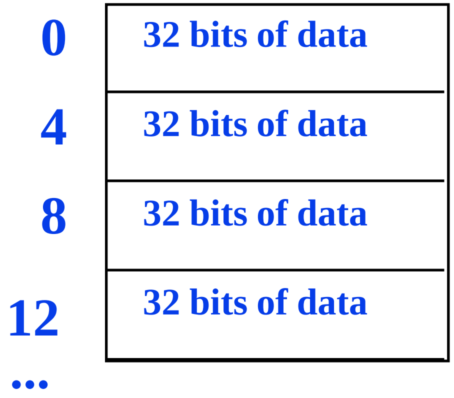
- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "**Byte addressing**" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Instructions

- Load and store instructions
- Example:

```
C code:     A[8] = h + A[8];

MIPS code:  lw $t0, 32($s3)
            add $t0, $s2, $t0
            sw $t0, 32($s3)
```

- Store word has destination last
- Remember arithmetic operands are registers, not memory!

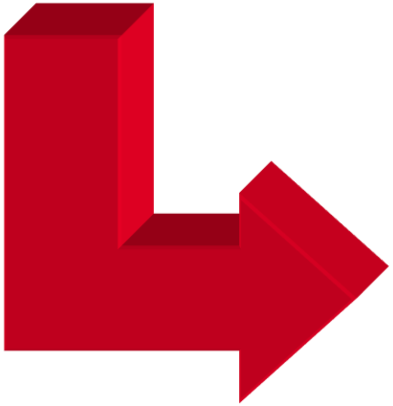
Our First Example

- Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Assume: *k->$5*

v[0]->$4



```
swap:
   muli $2, $5, 4
   add  $2, $4, $2
   lw $15, 0($2)
   lw $16, 4($2)
   sw $16, 0($2)
   sw $15, 4($2)
   jr $31
```

So far we have learned:

- MIPS
    - loading words but addressing bytes
    - arithmetic on registers only

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=9, $s1=17, $s2=18`

- Instruction Format:

000000	10001	10010	01000	00000	100000

op	rs	rt	rd	shamt	funct

- *Can you guess what the field names stand for?*

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

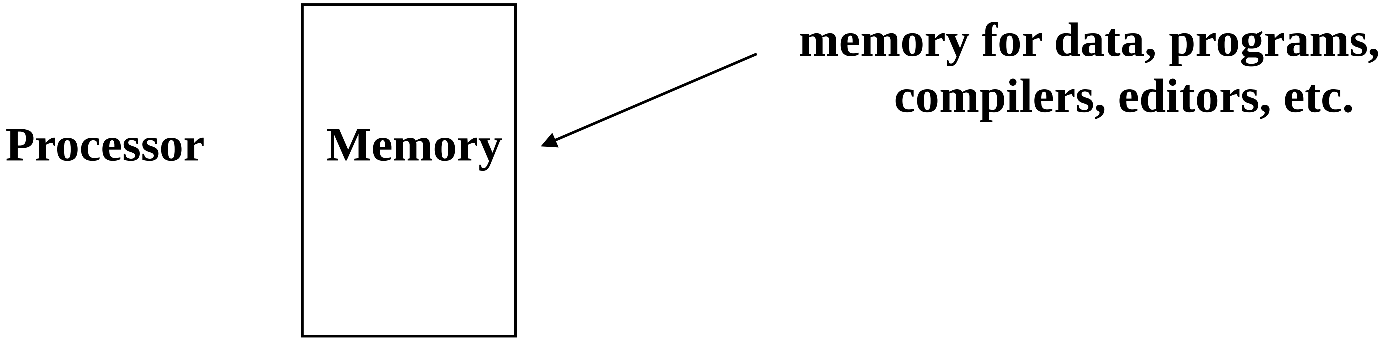
35	18	9	32

op	rs	rt	16 bit number

Where's the compromise?

Stored Program Concept

- **Instructions are bits**
- **Programs are stored in memory to be read or written just like data**



- **Fetch & Execute Cycle**
  - **Instructions are fetched and put into a special register**
  - **Bits in the register "control" the subsequent actions**
  - **Fetch the next instruction and continue**

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
     bne $s0, $s1, Label
     add $s3, $s0, $s1
Label:    ....
```

Control

- **MIPS unconditional branch instructions:**

```
j   label
```

- **Example:**

```
if (i!=j)          beq $s4, $s5, Lab1
     h=i+j;        add $s3, $s4, $s5
else               j Lab2
     h=i-j;        Lab1:sub $s3, $s4, $s5
                   Lab2:...
```

- ***Can you build a sample for loop?***

So far:

Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	Next instr. is at Label if $s4 ≠ $s5
`beq $s4,$s5,L`	Next instr. is at Label if $s4 = $s5
`j Label`	Next instr. is at Label

- Formats:

<table>
<tr><td>R</td><td>op</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
<tr><td>I</td><td>op</td><td>rs</td><td>rt</td><td colspan="3">16 bit address</td></tr>
<tr><td>J</td><td>op</td><td colspan="5">26 bit address</td></tr>
</table>

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

slt : set on less than

- `slt $t0, $s1, $s2`

```
if  $s1 < $s2 then
        $t0 = 1
            else
        $t0 = 0
```

- Can use this instruction to build "`blt $s1, $s2, Label`" can now build general control structures
- Note that the assembler needs a register to do this, there are policy of use conventions for registers

Policy of Use Conventions

$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluati
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:

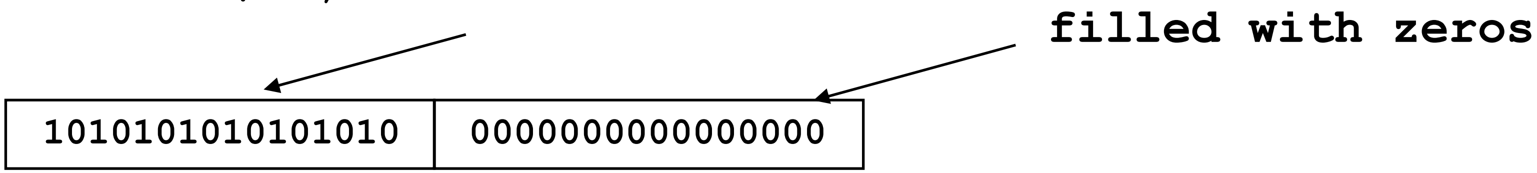
```
addi  $29, $29, 4
slti  $8, $18, 10
andi  $29, $29, 6
ori   $29, $29, 4
```

- How do we make this work?

How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

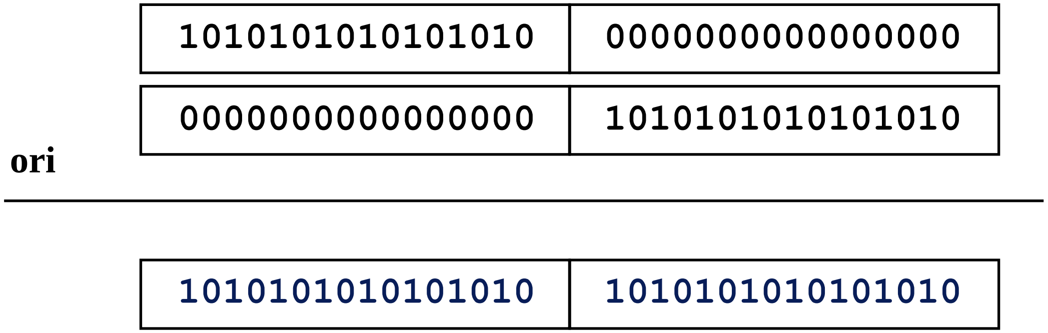


filled with zeros

1010101010101010	0000000000000000

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



	1010101010101010	0000000000000000
ori	0000000000000000	1010101010101010
	1010101010101010	1010101010101010

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudo instructions'
  - e.g., Move $t0, $t1 exists only in Assembly
  - would be implemented using Add $t0,$t1,$zero
- When considering performance you should count real instructions

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` ≠ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4` = `$t5`
`j Label`	Next instruction is at Label

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Addresses are not 32 bits**
  - **How do we handle this with load and store instructions?**

Addresses in Branches

- **Instructions:**

  `bne $t4,$t5,Label` Next instruction is at Label if $t4 ≠ t5
  `beq $t4,$t5,Label` Next instruction is at Label if $t4 =$t5

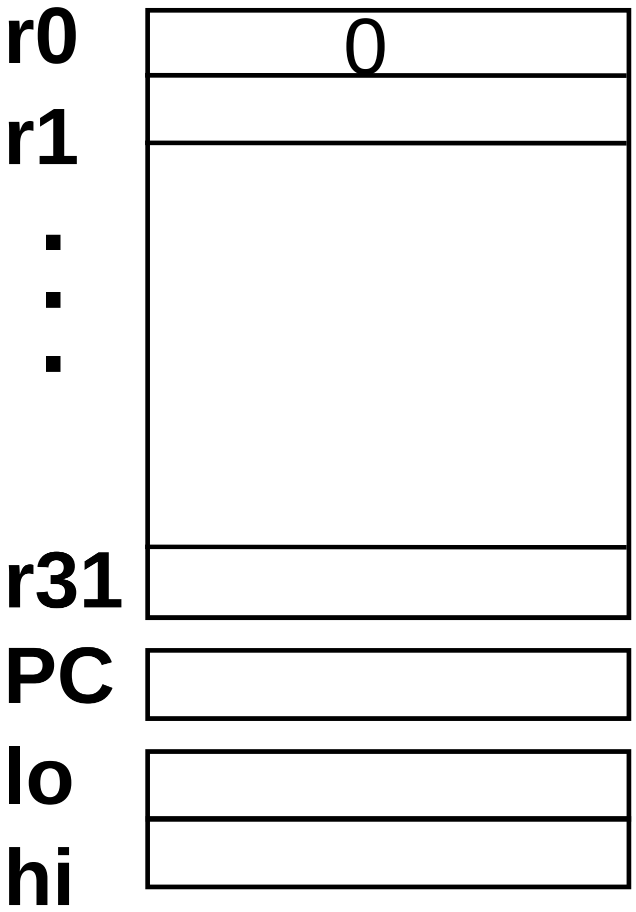
- **Formats:**

I	op	rs	rt	16 bit address

- **Could specify a register (like lw and sw) and add it to address**
  - **use Instruction Address Register (PC = program counter)**
  - **most branches are local (principle of locality)**
- **Jump instructions just use high order bits of PC**
  - **address boundaries of 256 MB**

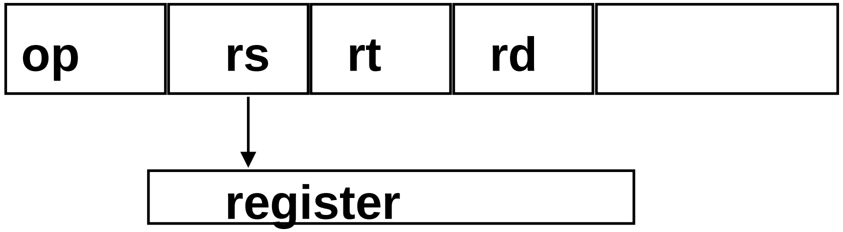
MIPS R2000 / R3000 的寄存器

- 可编程存储
  - 2^{32} x bytes
  - 31 x 32-bit GPRs (R0 ≡ 0)
  - 32 x 32-bit FP regs
(数据处理成对:paired DP)
  - HI, LO, PC

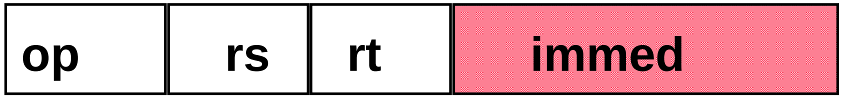


MIPS 的寻址方式/指令格式

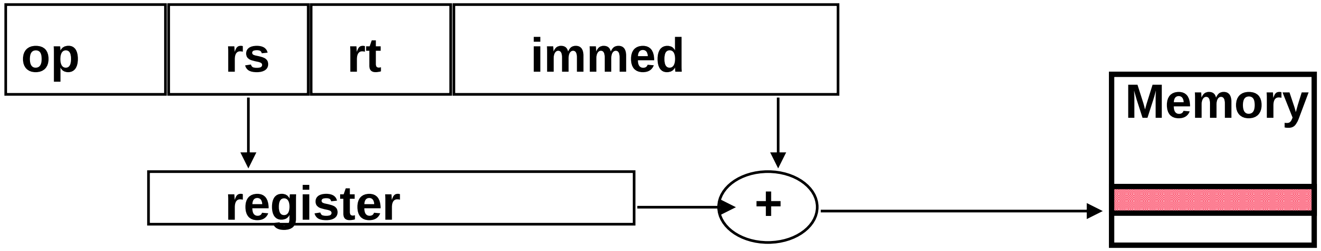
Register (direct)



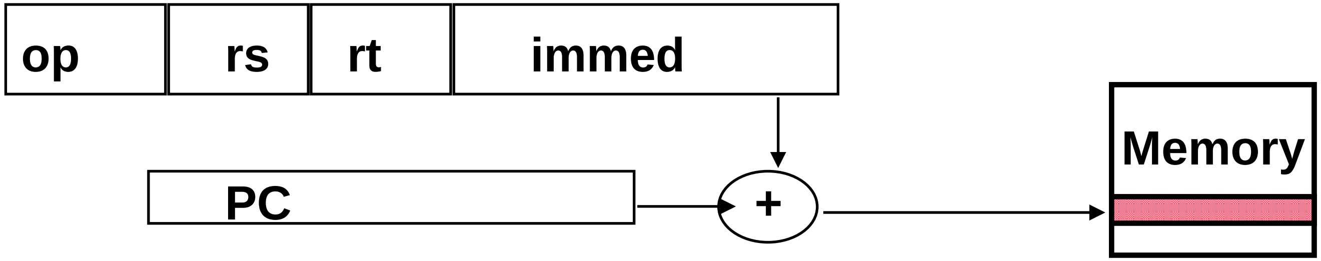
Immediate



Base+index



PC-relative

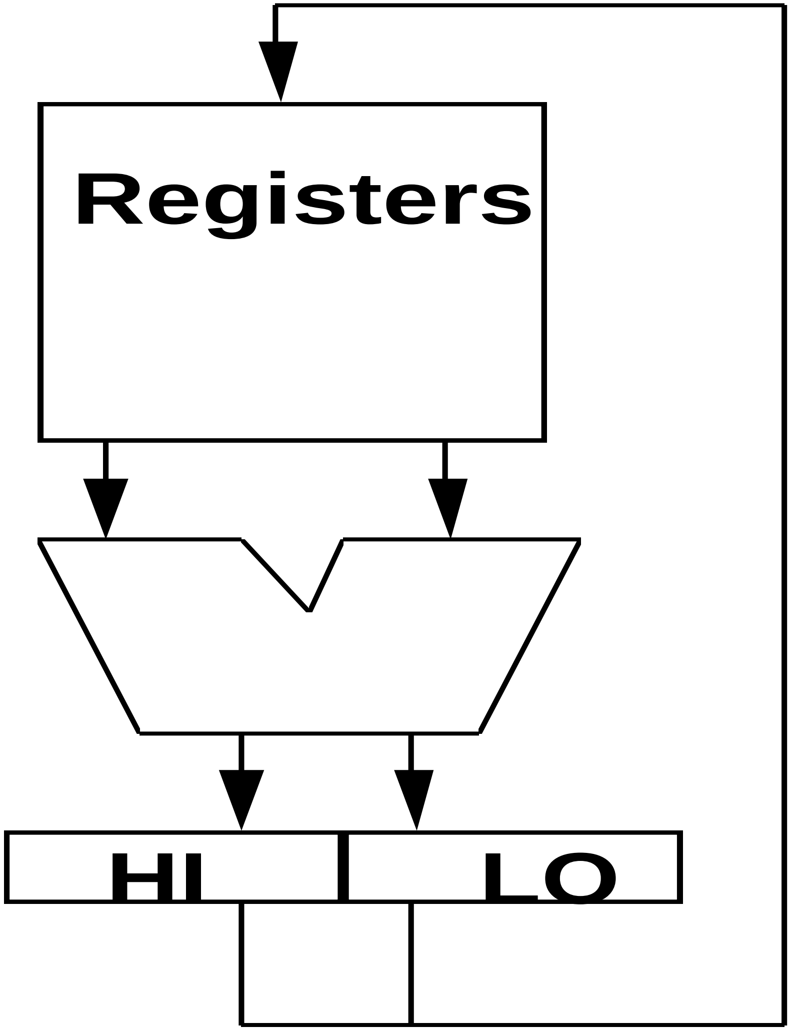


MIPS R2000 / R3000 的操作

- 算术逻辑类(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
- 存储器访问(Memory Access)
  - LB, LBU, LH, LHU, LW, LWL,LWR
  - SB, SH, SW, SWL, SWR

乘法 / 除法

- Start multiply, divide
  - MULT rs, rt
  - MULTU rs, rt
  - DIV rs, rt
  - DIVU rs, rt
- Move from HI or LO
  - MFHI rd
  - MFLO rd
- Move to HI or LO
  - MTHI rd
  - MTLO rd



MIPS 的算术指令

指令	示例	含义	注释
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

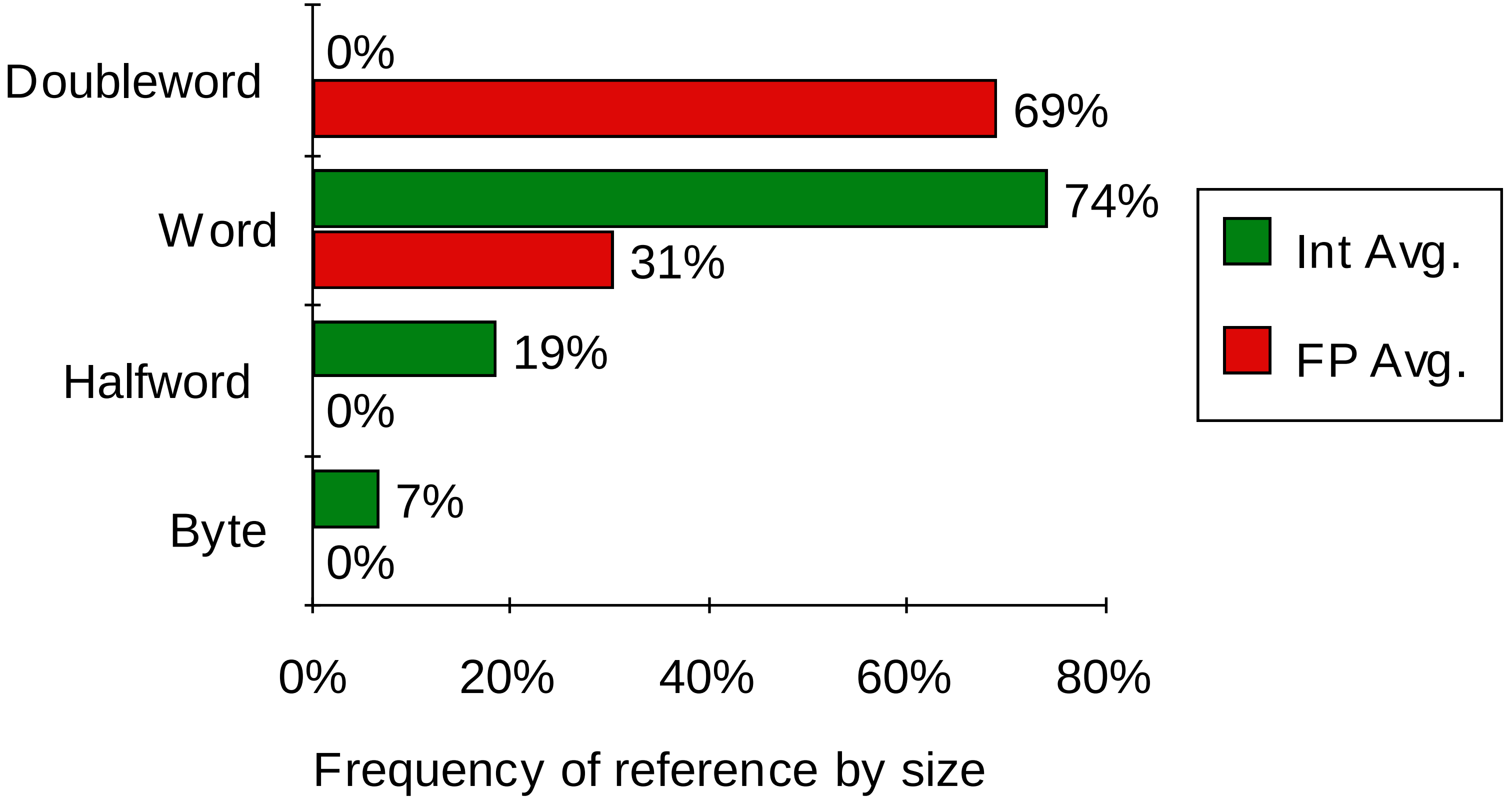
divide	div $2,$3	Lo = $2 ÷$3,	Lo = quotient, Hi = remainder
			Hi = $2 mod $3

divide unsigned	divu $2,$3	Lo = $2 ÷$3,	Unsigned quotient & remainder Hi = $2 mod $3
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

MIPS 的逻辑指令

指令	示例	含义	注释
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

Operand Size Usage



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

MIPS 的数据传输指令

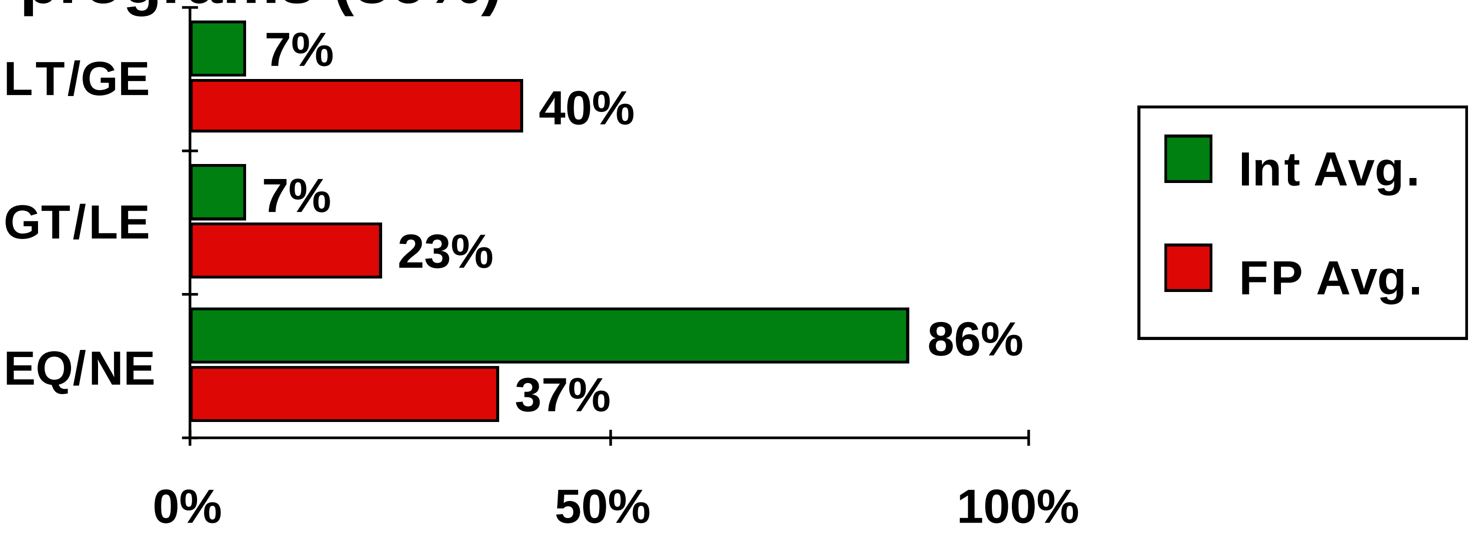
指令	*注释*
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

Conditional Branch Addressing

- PC-relative since most branches to the current PC address

- At least 8 bits suggested (128 instructions)

- Compare Equal/Not Equal most important for integer programs (86%)



Frequency of comparison types in branches

比较并转移

比较并转移

- BEQ rs, rt, offset   if R[rs] == R[rt] then PC-relative branch
- BNE rs, rt, offset   <>

与零比较并转移

- 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   >=

其他的比较并转移 需要两条指令

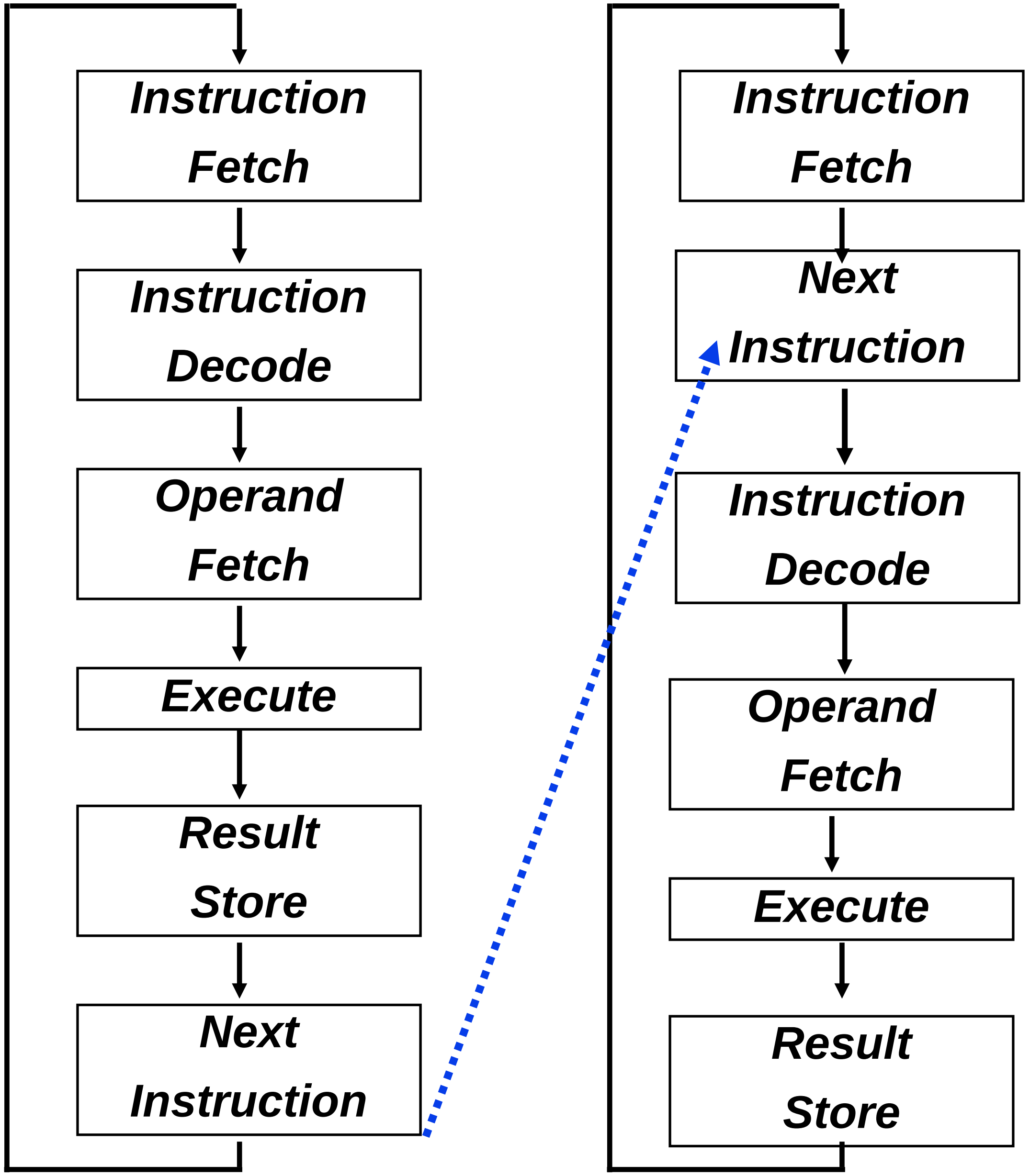
几乎所有的比较 都是与 0 进行比较!

MIPS 的跳转(jump)、转移(branch)、比较(compare)指令

指令	*示例*	*含义*
branch on equal	beq $1,$2,100 *Equal test; PC relative branch*	if ($1 == $2) go to PC+4×100
branch on not eq.	bne $1,$2,100 *Not equal test; PC relative*	if ($1!= $2) go to PC+4 × 100
set on less than	slt $1,$2,$3 *Compare less than; 补码.*	if ($2 < $3) $1=1; else $1=0
set less than imm.	slti $1,$2,100 *Compare < constant; 补码.*	if ($2 < 100) $1=1; else $1=0
set less than uns.	sltu $1,$2,$3 *Compare less than; 自然数*	if ($2 < $3) $1=1; else $1=0
set l. t. imm. uns.	sltiu $1,$2,100 *Compare < constant; 自然数*	if ($2 < 100) $1=1; else $1=0
jump	j 10000 *跳转到目标地址*	go to 10000
jump register	jr $31 *用于switch, procedure return*	go to $31
jump and link	jal 10000 *用于 procedure call*	$31 = PC + 4; go to 10000

执行周期

复习



Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Result Store

Next Instruction

Instruction Fetch

Next Instruction

Instruction Decode

Operand Fetch

Execute

Result Store

从程序存储系统中获得指令

确定后续指令

确定所需的动作和指令大小

定位并获得操作数数据

计算结果数值或状态

在存储系统中存放结果，以备后用

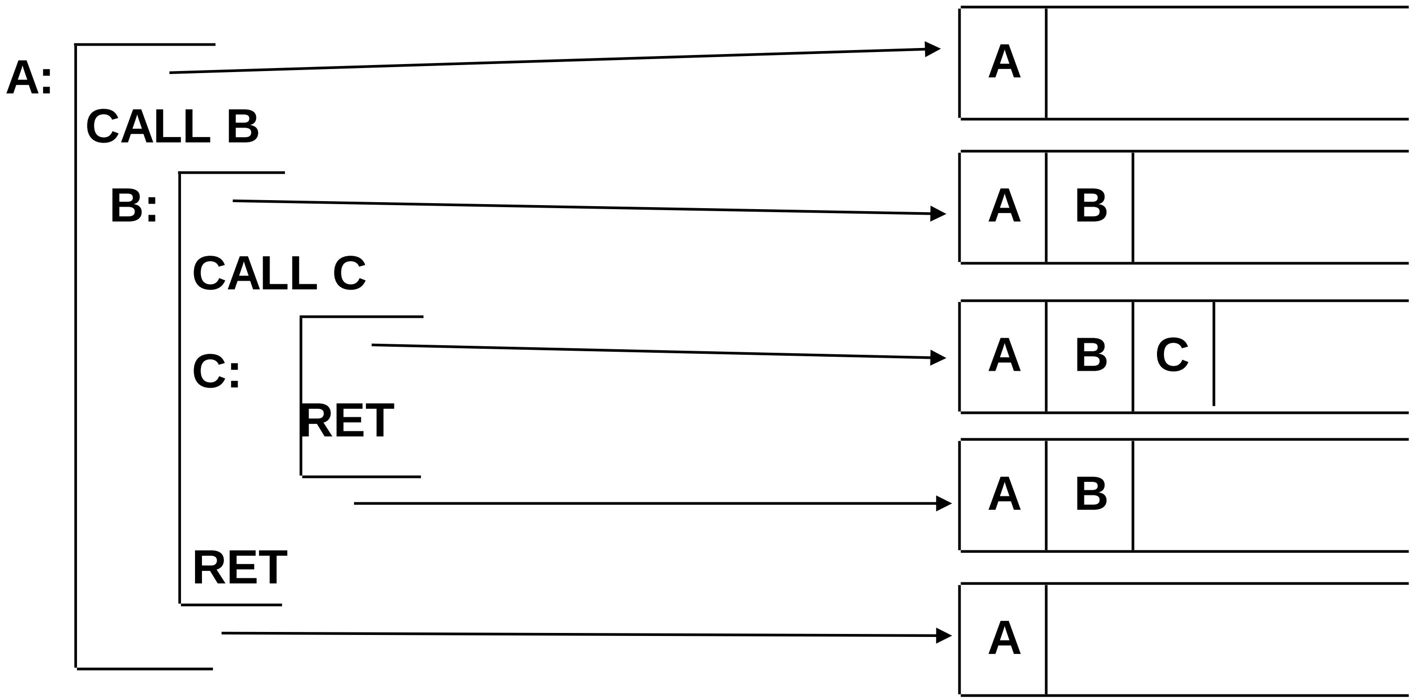
MIPS 的跳转(jump)、转移(branch)、比较(compare)指令

指令	*示例*	*含义*
branch on equal	beq $1,$2,100 *Equal test; PC relative branch*	if ($1 == $2) go to PC+4×100
branch on not eq.	bne $1,$2,100 *Not equal test; PC relative*	if ($1!= $2) go to PC+4 × 100

branch on equal	beq $1,$2,100	if ($1 == $2) goto PC+4×100+4 *Equal test; PC relative branch*
branch on not eq.	bne $1,$2,100	if ($1!= $2) goto PC+4 × 100+4 *Not equal test; PC relative*

为什么堆栈非常重要？

子程序调用&返回时的堆栈操作和环境:



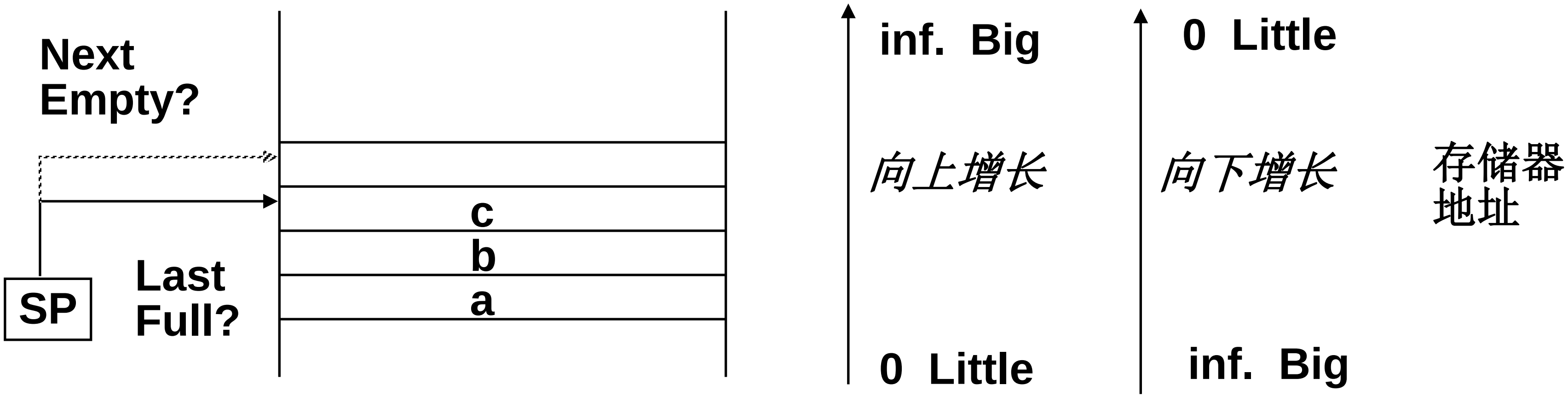
一些机器提供一个存储器堆栈(memory stack), 并将此作为系统结构的一部分 (例如, VAX)

一些机器堆栈是利用软件来实现的 (例如，MIPS)

存储器堆栈(Memory Stacks)

即使操作数堆栈不是系统结构的组成部分, 也有益于 栈式环境(stacked environments),
以及 子程序调用和返回

向上增长的堆栈 **vs.** *向下增长的堆栈:*



堆栈为空表明什么?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

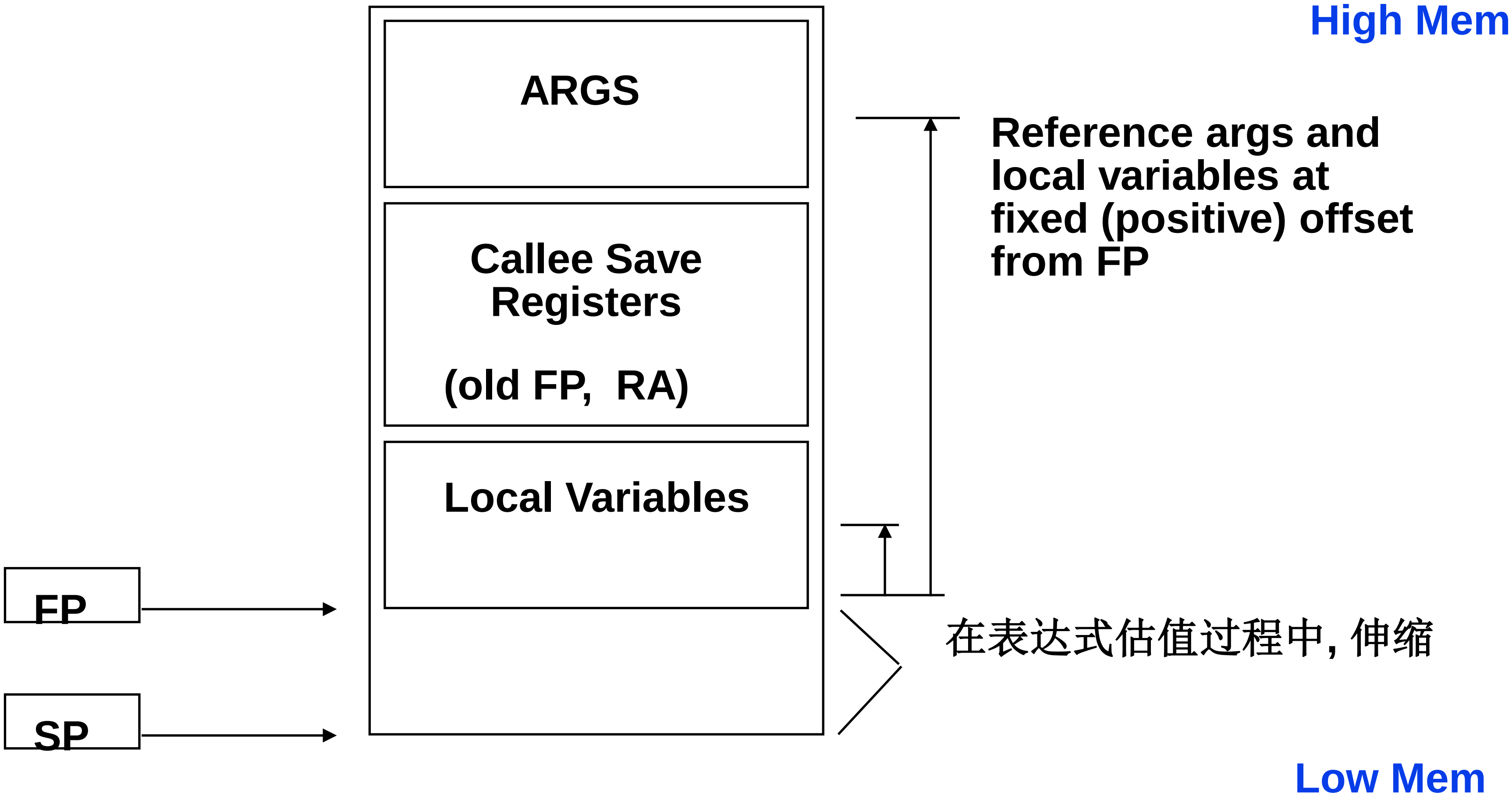
PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP

Call-Return链接：Stack Frames



- 堆栈有许多可能的变种 (up/down, last pushed / next)

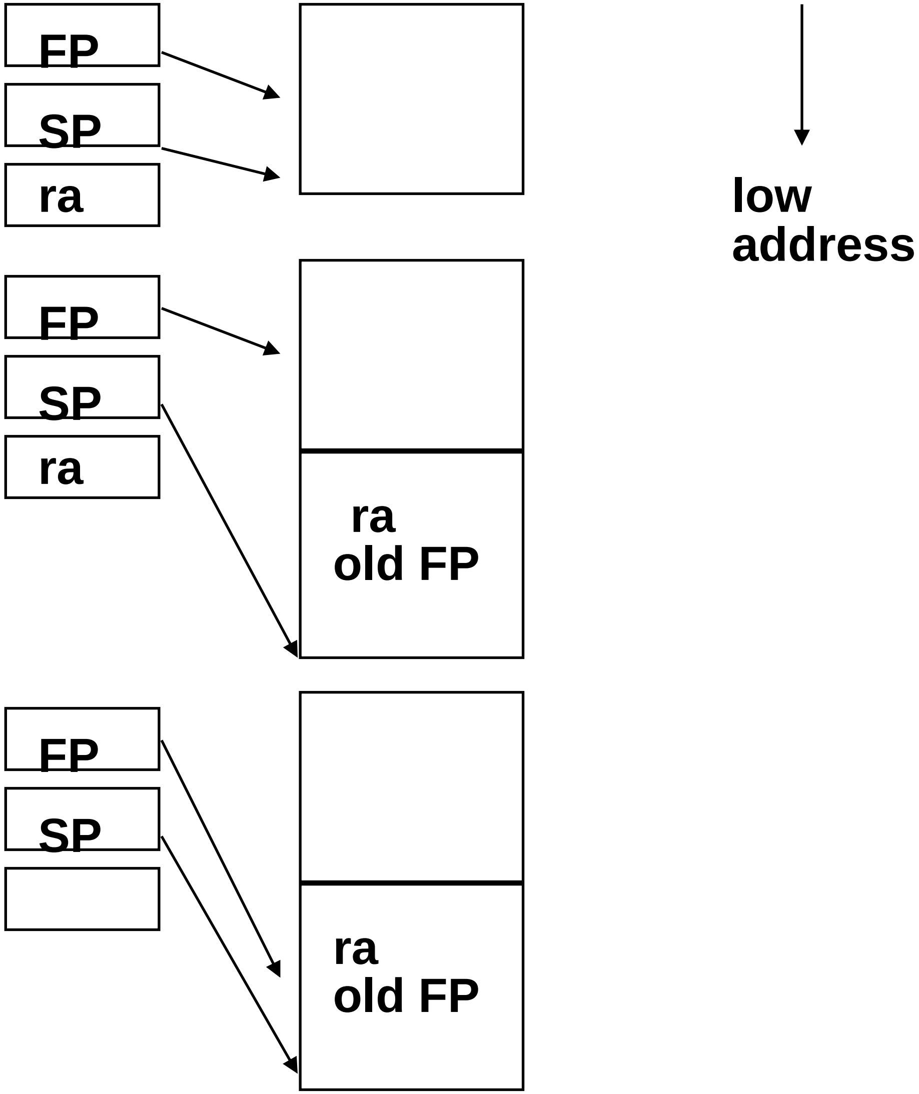
- Block structured languages contain link to lexically enclosing frame.

MIPS寄存器使用情况的软件约定

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (caller saves)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

MIPS / GCC 调用约定

```
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
```



前四个参数通过寄存器传送

C示例: swap

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

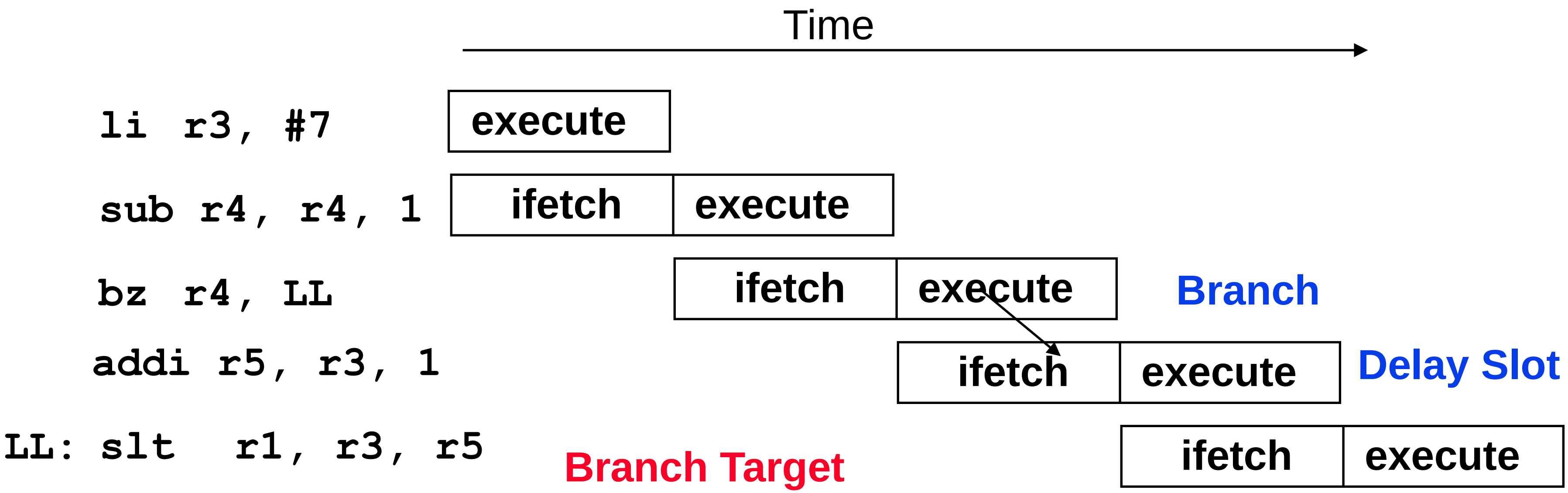
- 假设swap被作为过程进行调用
- 假设temp在寄存器 $15; 参数分别在 $a1, $a2;
  $16 是中间使用的寄存器(scratch reg):

  MIPS代码?

swap: MIPS

```
swap:
   addi      $sp,$sp, -4       ; create space on stack
   sw        $16, 0($sp)       ;callee saved register put onto stack
   sll       $t2, $a2,2        ; mulitply k by 4
   addu      $t2, $a1,$t2      ; address of v[k]
   lw        $15, 0($t2)       ; load v[k]
   lw        $16, 4($t2)       ; load v[k+1]
   sw        $16, 0($t2)       ; store v[k+1] into v[k]
   sw        $15, 4($t2)       ; store old value of v[k] into v[k+1]
   lw        $16, 8($sp)       ; callee saved register restored from stack
   addi      $sp,$sp, 4        ; restore top of stack
   jr        $31               ; return to place that called swap
```

Branch & Pipelines



Time

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

execute

ifetch execute

ifetch execute Branch

ifetch execute Delay Slot

Branch Target ifetch execute

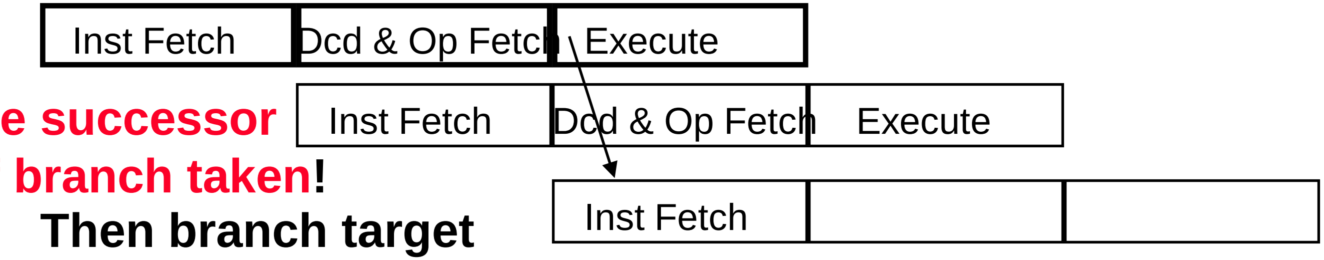
By the end of Branch instruction, the CPU knows whether or not the branch will take place.

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

Branch:



Inst Fetch | Dcd & Op Fetch | Execute

Inst Fetch | Dcd & Op Fetch | Execute

Inst Fetch

execute successor even if branch taken!

Then branch target or continue

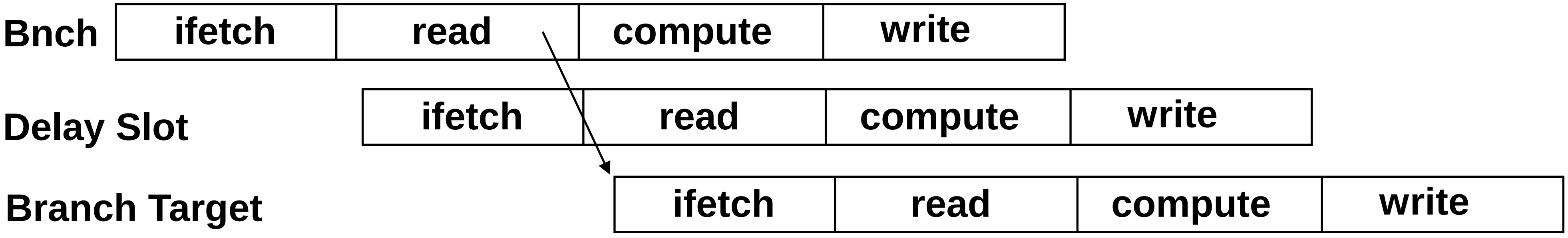
Single delay slot impacts the critical path

- **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?

转移 & 流水线



Bnch | ifetch | read | compute | write

Delay Slot | ifetch | read | compute | write

Branch Target | ifetch | read | compute | write

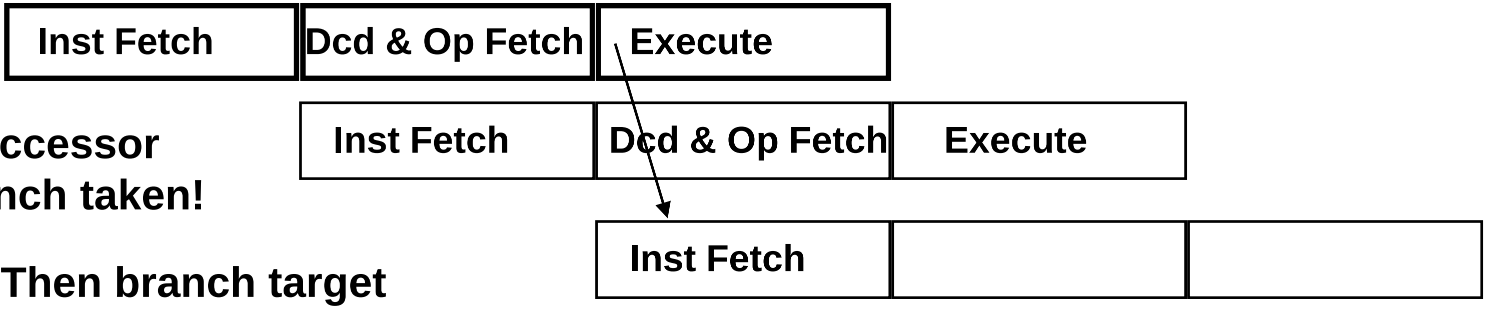
在转移指令的 “READ”段的最后, CPU才知道转移是否真的发生。

然而，在上例中，

无论这个转移是否真的发生，它都将访取下一条指令。

延迟转移（Delayed Branches）：重新定义操作行为

Branch:



execute successor
even if branch taken!

Then branch target
or continue

Single delay slot
impacts the critical path

- 在50%的时间，编译器能够移动一条有效指令填充延迟槽。
  - 试图从跳转之前，移动下来指令
  - 如果安全，可以从目标出移动来指令

```
add r3, r1, r2

sub r4, r4, 1

bz  r4, LL

...


LL: add rd, ...
```

MIPS 的其他指令

■ 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: discussed later
■ TLB instructions	Support for virtual memory: discussed later
■ restore from exception	Restores previous interrupt mask & kernel/user mode bits into status register
■ load word left/right	Supports misaligned word loads
■ store word left/right	Supports misaligned word stores

MIPS 指令系统的细节信息

- #0寄存器总是具有数值0 (即使试图对它写入其他数值，也是如此)
- 转移和跳转指令将返回地址PC+4装入链接寄存器（ link register）
- 所有指令改变目标寄存器的所有 32 位信息 (包括lui, lb, lh)，并读取源寄存器的所有32位信息 (add, sub, and, or, 等等）
- 立即数算术和逻辑指令进行如下扩展:
  - 逻辑立即数（logical immediates）　　零扩展到 32位
  - 算术立即数（arithmetic immediates）符号扩展到32位
- 指令lb 和 lh 装入的数据进行如下扩展:
  - lbu, lhu 是零扩展
  - lb, lh　是符号扩展
- 下面的算术和逻辑指令可能会出现溢出:
  - add, sub, addi
  - 以下指令不会出现溢出： addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as RISC vs. CISC
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy

    instructions from 1 to 54 bytes long!
- We will look at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: `lw $t1,$a0+$s3 #$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example:

    `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register `bc Loop`?

    decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +insts
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few insts (mostly designed for higher performance)
- 1997: MMX is added,
- 1997-2000: Pentium II, Pentium III, Pentium IV

This history illustrates

the impact of the Golden handcuffs of compatibility

An architecture that is difficult to explain and impossible to love?

其他指令系统体系结构

- Intel 8086/88 => 80286 => 80386 => 80486 => Pentium => Pentium Pro =>Itanium?
  - 8086只能使用不多的晶体管，因而它只实现了 16位微处理器
  - 试图与8位8080微处理器比较兼容
  - 在8086的后续微处理器中，不断增加了许多新的功能
  - 许多不同的计算机工程师的成果
  - 1978年宣布

- VAX 简单的编译器 & 小代码空间 =>
  - 高效的指令编码
  - 强大的寻址方式
  - 强大的指令
  - 寄存器较少
  - 一位出色设计师的成果
  - 1977年宣布

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., Base or scaled index with 8 or 32 bit displacement
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

What the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective

机器示例： 地址与寄存器

Intel 8086	2^{20} x 8 bit bytes AX, BX, CX, DX SP, BP, SI, DI CS, SS, DS IP, Flags	acc, index, count, quot stack, string code,stack,data segment
VAX 11	2^{32} x 8 bit bytes 16 x 32 bit GPRs	r15-- program counter r14-- stack pointer r13-- frame pointer r12-- argument ptr
MC 68000	2^{24} x 8 bit bytes 8 x 32 bit GPRs 7 x 32 bit addr reg 1 x 32 bit SP 1 x 32 bit PC	
MIPS	2^{32} x 8 bit bytes 32 x 32 bit GPRs 32 x 32 bit FPRs HI, LO, PC	

VAX 机器的操作

- 一般格式： (操作) (数据类型) (2, 3) 2 或 3个显式操作数

Bits	Date type	DEC’s name
8	Integer	Byte
16	Integer	Word
32	Integer	Long Word
32	Floating Point	F_float
64	Integer	Quad Word
64	Floating Point	D_floating
128	Integer	Octa word
128	Floating Point	H(uge)_Floating

- 例如

add (b, w, l, f, d) (2, 3)

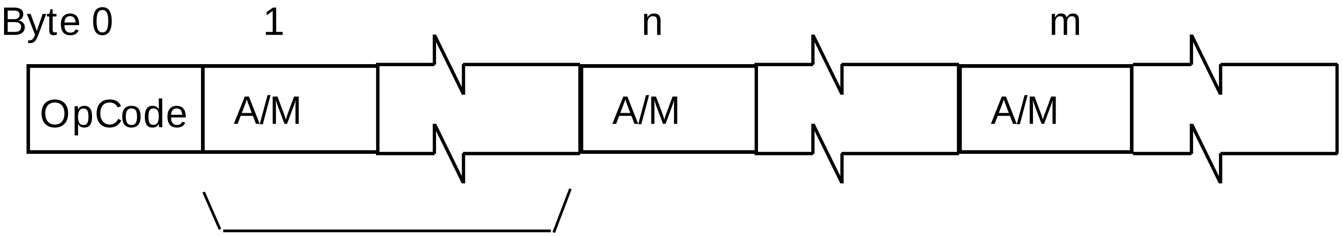
产生

addb2 addw2 addl2 addf2 addd2

addb3 addw3 addl3 addf3 addd3

VAX 的指令格式、寻址方式

一般指令格式



操作数描述符（operand specifier）

register	5	r			
autoinc	8	r			
disp	A	r	byte		
	C	r	half word		
	E	r	word		
index	4	r	m	r	displacement

Mem[Displacement + r+r*m]

swap: MIPS vs. VAX

```
swap:
   addi   $sp,$sp, -4               .word ^m<r0,r1,r2,r3>       ; saves r0 to r3
   sw     $16, 4($sp)
   sll    $t2, $a2,2                movl     r2, 4(ap); move arg v[] to reg
   addu   $t2, $a1,$t2              movl     r1, 8(ap)           ; move arg k to reg
   lw     $15, 0($t2)               movl     r3, (r2)[r1]        ; get v[k]
   lw     $16, 4($t2)               addl3    r0, #1,8(ap)        ;  reg gets k+1
   sw     $16, 0($t2)               movl     (r2)[r1],(r2)[r0]   ; v[k] = v[k+1]
   sw     $15, 4($t2)               movl     (r2)[r0],r3         ; v[k+1] gets old v[k]
   lw     $16, 8($sp)
   addi   $sp,$sp, 4
   jr     $31                       ret      ; return to caller, restore r0 -  r3
```

总结

- 使用通用寄存器的load-store 结构: YES

- 提供至少16个通用寄存器，以及单独的浮点寄存器: 31 GPR & 32 FPR

- 支持如下寻址方式：displacement (with an address offset size of 12 to 16 bits)、

- immediate (size 8 to 16 bits), 以及 register deferred : YES: 16 bits for immediate, displacement (disp=0 => register deferred)

- 所有的寻址方式都可以用于所有的数据传输指令: YES

- 如果看重性能，就使用固定指令编码方案；如果看重代码大小，就使用可变指令编码方案: Fixed

- 支持如下数据大小和类型：8位、16位、32位整数；以及32位和 64位 IEEE 754 浮点数: YES

- 支持如下简单指令（因为它们决定执行的指令总数)：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, 以及 return : YES, 16b

- 瞄准最低限要求的指令系统（ minimalist instruction set）: YES

总结: MIPS R3000的显著特点

- 32位固定格式的指令 (3 种格式)
- 32 个32位 GPR (R0 = 0) 和 32 个 FP 寄存器 (以及 HI LO)
    - 根据软件约定划分
- 3 地址, 寄存器-寄存器算术指令
- 对于load/store指令，单一寻址模式: base+displacement
    - –没有间接寻址（indirection）
    - –16位立即数 加 LUI
- 简单的转移条件
    - 与0比较，或者判断两个寄存器是否相等
    - 没有条件码
- 延迟转移
    - 即使转移发生，也要执行转移（或跳转）之后的指令
    - (在50%的时间，编译器能够在延迟转移的时间中填充有用工作)