Ch5: Instruction Set
指令系统

第1讲：指令系统的设计

第2讲：程序的机器级表示

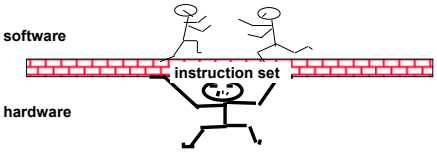
第一讲 指令系统设计

主 要 内 容

- 一条指令必须指定的信息
- 指令中的地址码个数
- 指令系统设计的基本原则
- 指令类型
- 数据类型
  - 寄存器组织
  - 存储器组织
- 操作数的寻址方式
  - 立即 / 寄存器 / 寄存器间接 / 直接 / 间接 / 堆栈 / 偏移
- 操作码的编码
  - 定长编码法
  - 变长扩展编码法
- 条件码和标志寄存器
- 指令设计风格
- 指令系统举例

Instruction Set Design

- 指令系统处在软件和硬件交界面上，能同时被硬件设计者和系统程序员看到
- 硬件设计者角度：IS为CPU提供功能需求
  - IS设计目标为：易于硬件设计
- 系统程序员角度：通过IS来使用硬件
  - IS设计目标为：易于编写编译器
- IS设计的好坏还决定了：计算机的性能和成本



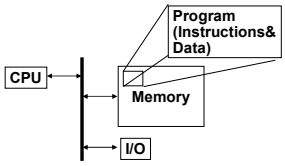
回顾：冯.偌依曼结构机器对指令规定：
- 用二进制表示，和数据一起存放在主存中
- 由两部分组成：操作码和操作数（或其地址码）
  - Operation Code: defines the operation type
  - Operands: indicate operation source and destination

Instruction Set Architecture

Programmer's View

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

Computer's View



Princeton (Von Neumann) Architecture

--- 数据和指令存放在同一个存储器中

-- 存储空间利用率高

-- 统一的访问接口

Harvard Architecture

--- 数据和指令存在不同的存储器

-- 有利于流水线执行

一条指令须包含的信息

一条指令必须明显或隐含地包含以下信息：

操作码：指定操作类型

（操作码长度：固定 / 可变）

源操作数参照：一个或多个源操作数所在的地址

（操作数来源：主(虚)存/寄存器/I/O端口/指令本身）

结果值参照：产生的结果存放何处

（结果地址：主(虚)存/寄存器/I/O端口）

下一条指令地址：下条指令存放何处

（下条指令地址 ：主(虚)存）

（正常情况隐含在PC中，改变顺序时由指令给出）

地址码字段的个数

据上述分析知，一条指令包含 1 个操作码和多个地址码

零地址指令

(1) 无需操作数 如：空操作 / 停机等

(2) 所需操作数为默认的 如：堆栈 / 累加器等

形式：

OP

一地址指令

其地址既是操作数的地址，也是结果的地址

(1) 单目运算：如：取反 / 取负等

(2) 双目运算：另一操作数为默认的 如：累加器等

形式：

OP	A1

二地址指令（最常用）

分别存放双目运算中两个操作数，并将其中一个地址作为结果的地址。

形式：

OP	A1	A2

三地址指令（RISC风格）

分别作为双目运算中两个源操作数的地址和一个结果的地址。

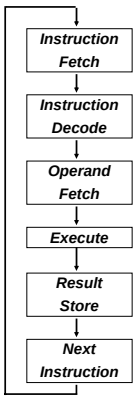
形式：

OP	A1	A2	A3

多地址指令

大中型机中用于成批数据处理的指令,如:向量 / 矩阵等

从指令执行周期看指令设计涉及的问题



Obtain instruction from program storage
指令地址、指令长度（定长/变长）

Determine required actions
指令格式、操作码编码、操作数类型

Locate and obtain operand data
地址码格式、寻址方式、操作数格式和存放方式

Compute result value or status
操作类型、标志或条件码

Deposit results in storage for later use
结果数据位置

Determine successor instruction
下条指令地址（顺序 / 转移）

指令格式的设计

指令格式的选择应遵循的几条基本原则：

- 应尽量短
- 要有足够的操作码位数
- 指令编码必须有唯一的解释，否则是不合法的指令
- 指令字长应是字节的整数倍
- 合理地选择地址字段的个数
- 指令尽量规整

与指令集设计相关的重要方面

- 操作码的全部组成：操作码个数/种类/复杂度
  LD/ST/INC/BRN 四种指令已足够编制任何可计算程序，但程序会很长
- 数据类型：对哪几种数据类型完成操作
- 指令格式：指令长度/地址码个数/各字段长度
- 通用寄存器：个数/功能/长度
- 寻址方式：操作数地址的指定方式
- 下条指令的地址如何确定：顺序，PC+1；条件转移；无条件转移；……

一般通过对操作码的不同编码定义不同的含义，操作码相同时，再由功能码定义不同的含义!

Typical Operations(典型的操作)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move push, pop (to/from stack)
Input/Output	input (from I/O device) output (to I/O device)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide Adc(带进位加)，Sbb (带借位减)
Logical	not, and, or, set, clear
Shift	(arithmatic,logic,rotate)left/right shift
String	search, translate
Exec-Seq control	Jump, branch
CPU control	stop, sti(开中断), break
Subroutine Linkage	call, return
Interrupt	trap, interrupt return
Synchronization	test & set (atomic r-m-w)

操作数类型和存储方式

操作数是指令处理的对象，其基本类型有：

地址

被看成无符号整数，用来参加运算以确定主(虚)存地址

数值数据

定点数(整数)：一般用二进制补码表示

浮点数(实数)：大多数机器采用IEEE754标准

十进制数：一般用NBCD码表示，压缩/非压缩

位、位串、字符和字符串

用来表示文本、声音和图象等

- 4 bits is a nibble（一个十六进制数字）
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word

逻辑(布尔)数据

按位操作（0-假 / 1-真）

Pentium & MIPS Data Type

- Pentium
  - 基本类型：
    - » 字节、字(16位)、双字(32位)、四字(64位)
  - 整数：
    - » 16位、32位、64位三种2-补码表示的整数
    - » 18位压缩BCD码表示的十进制整数
  - 无符号整数（8、16或32位）
  - 近指针：32位段内偏移（有效地址）
  - 浮点数：IEEE754（80位扩展精度浮点数）
- MIPS
  - 基本类型：
    - » 字节、半字(16位)、字(32位)、四字(64位)
  - 整数： 16位、32位、64位三种2-补码表示的整数
  - 无符号整数：（16、32位）
  - 浮点数：IEEE754（32位/64位浮点数）

Addressing Modes（寻址方式）

- 什么是“寻址方式”?
  操作数指定方式。即：用来指定操作数或操作数所在位置的方法
- 地址码编码由操作数的寻址方式决定
- 地址码编码原则：
  为什么？
  指令地址码尽量短 → 目标代码短，省空间
  操作数存放位置灵活，空间应尽量大 → 有利于编译器优化产生高效代码
  有效地址计算过程尽量简单 → 指令执行快
- 指令的寻址----简单
- 正常：PC增值
- 跳转 (jump / branch / call / return)：同操作数的寻址
- 操作数的寻址----复杂
- 操作数来源：寄存器 / 外设端口 / 主(虚)存 / 栈顶
- 操作数结构：位 / 字节 / 半字 / 字 / 双字 / 一维表 / 二维表 /…
- 通常寻址方式特指“操作数的寻址”

Addressing Modes

- 寻址方式的确定
  - （1）在操作码中给定寻址方式
    - 如：MIPS指令，指令中仅有一个主(虚)存地址的，且指令中仅有一二种寻址方式。Load/store型机器指令属于这种情况。
  - （2）专门的寻址方式位
    - 如：X86指令，指令中有多个操作数，且寻址方式各不相同，需要各自说明寻址方式。
- 有效地址的含义
  - 通过指令计算得到的操作数地址
- 基本寻址方式
  - 立即 / 直接 / 间接 / 寄存器 / 寄存器间接 / 偏移 / 堆栈
- 基本寻址方式的算法及优缺点
  - （见下页）

ISA.13

基本寻址方式的算法和优缺点

指令：OP A，R，.....

假设：A=地址字段值，R=寄存器编号，
EA=有效地址，(X)=地址X中的内容

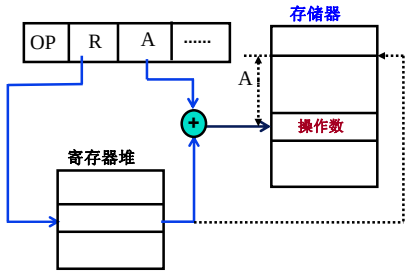
方式	算法	主要优点	主要缺点
立即	操作数=A	指令执行速度快	操作数幅值有限
直接	EA=A	有效地址计算简单	地址范围有限
间接	EA=(A)	有效地址范围大	多次存储器访问
寄存器	操作数=(R)	指令执行快，指令短	地址范围有限
寄间接	EA=(R)	地址范围大	额外存储器访问
偏移	EA=A+(R)	灵活	复杂
堆栈	EA=栈顶	指令短	应用有限

偏移方式：将直接方式和寄存器间接方式结合起来。
有：相对 / 基址 / 变址三种 （见后面几页！）

问题：以上各种寻址方式下，操作数在寄存器中还是在存储器中？有没有可能在磁盘中？什么情况下，所取数据在磁盘中？

ISA.14

偏移寻址方式



偏移寻址：EA=A+(R) R可以明显给出，也可以隐含给出
R可以为PC、基址寄存器B、变址寄存器I

- 相对： EA=A+(PC) 相对于当前指令处位移量为A的单元
- 基址： EA=A+(B) 相对于基址(B)处位移量为A的单元
- 变址： EA=A+(I) 相对于起址A处位移量为(I)的单元

ISA.15

偏移寻址方式

③ 相对寻址

指令地址码给出一个偏移量(带符号数)，基准地址隐含由PC给出。
即：EA=(PC)+A （ex. MIPS's instruction: Beq）
可用来实现程序(公共子程序)的浮动，或指定转移目标地址
注意：当前PC的值可能是正在执行指令的地址或下条指令的地址

③ 基址寻址

指令地址码给出一个偏移量，基准地址明显或隐含由基址寄存器B给出。即：EA=(B)+A （ex. MIPS's instructions: lw / sw）
可用来实现多道程序重定位，或过程调用中参数的访问

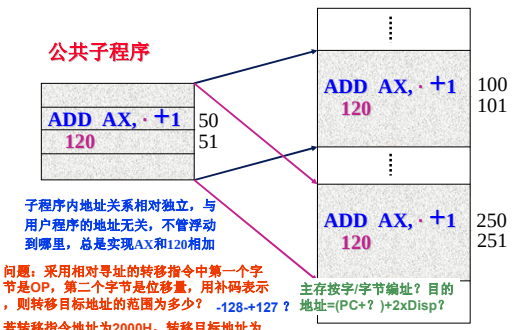
③ 变址寻址

指令地址码给出一个基准地址，而偏移量(无符号数)明显或隐含由变址寄存器I给出。即：EA=(I)+A
可为循环重复操作提供一种高效机制，如实现对线形表的方便操作

SKIP

ISA.16

相对寻址实现公共子程序的浮动和相对转移



子程序内地址关系相对独立，与用户程序的地址无关，不管浮动到哪里，总是实现AX和120相加

问题：采用相对寻址的转移指令中第一个字节是OP，第二个字节是位移量，用补码表示，则转移目标地址的范围为多少？ -128-+127？

若转移指令地址为2000H，转移目标地址为1FF0H，总是在取指令同时对PC增量，则转移指令第二字节位移量为多少？
1FF0H – 2002H = EEH（-18）？

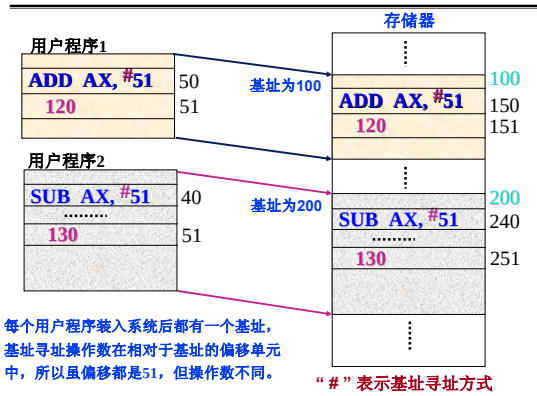
主存按字/字节编址？目的地址=(PC+？)+2xDisp？

“ . ”表示相对寻址方式

BACK

ISA.17

基址寻址实现程序重定位



每个用户程序装入系统后都有一个基址，基址寻址操作数在相对于基址的偏移单元中，所以虽偏移都是51，但操作数不同。

“ # ”表示基址寻址方式

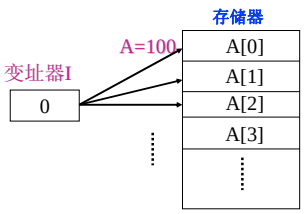
BACK

ISA.18

变址寻址实现线性表元素的存取

- 自动变址
  指令中的地址码A给定数组基址，变址器I每次自动加/减数组元素的长度X
  EA=(I)+A
  I=(I) ± x
- 在元素地址从低→高地址增长时，“+”；
- 在元素地址从高→低地址增长时，“-”
- 在没有硬堆栈的情况下，
- 用它来建立软堆栈
- 提供对线性表的方便访问

假定一维数组A从内存100号单元开始



若每个元素为一个字节，则I=(I) ± 1
若每个元素为4个字节，则I=(I) ± 4

BACK

寻址方式Addressing Modes

位、字节和块的寻址

- 位寻址
  当需要对寄存器或内(虚)存中单独一位进行操作(如：置位/复位/测试等)时，需要进行位寻址。
  指令中必须隐含或明显地给出位指针。
- 字节寻址
  当操作数为一个字节时，指令必须对字节进行定位。
  字节编址时，指令须给出访问的是字节 / 半字 / 字 / 双字...
  字编址时，指令须给出是否为字节访问，并指出是哪个字节
  （但目前基本都采用字节编址）
- 块寻址
  当需对一个信息块进行操作时，指令必须对块定位。(如：VAX11/780)
  指令须给出：首址+长度 / 首址+末址 / 首址+末端标志

Addressing Modes（寻址方式的汇编表示）

Addressing mode	Example	Meaning
Immediate	Add R4,3	R4 ← R4+3
Register	Add R4,R3	R4 ← R4+R3
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+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]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2– d; R1← R1+Mem[R2]
Scaled(乘比例因子)	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

上述形式是一种示意性表示，不同系列处理器的汇编表示形式不同！

Instruction Format(指令格式)

- 操作码的编码有两种方式
  - Fixed Length Opcodes (定长操作码法)
  - Expanding Opcodes (扩展操作码编法)
- instructions size
  - 代码长度更重要时：采用变长指令字、变长操作码
  - 性能更重要时：采用定长指令字、定长操作码

为什么？

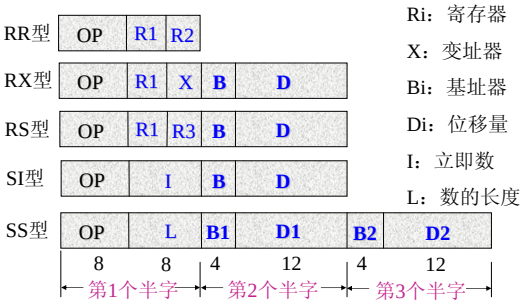
问题：是否可以有定长指令字、变长操作码？定长操作码、变长指令字呢？

实际上，指令长度是否可变与操作码长度是否可变没有绝对关系，但通常是“定长操作码、不一定是定长指令字”、“变长操作码、一般是变长指令字”。

定长编码Fixed Length Opcodes

- 基本思想
  指令的操作码部分采用固定长度的编码
  如：假设操作码固定为6位，则系统最多可表示64种指令
- 特点
  译码方便，但有信息冗余
- 举例
  IBM360/370采用:
  8 位定长操作码，最多可有256条指令
  只提供了183条指令，有73种编码为冗余信息
  机器字长32位，按字节编址
  有16个32位通用寄存器，基址器B和变址器X可用其中任意一个

IBM370指令格式



Ri：寄存器
X：变址器
Bi：基址器
Di：位移量
I：立即数
L：数的长度

RR：寄存器 - 寄存器
RX：寄存器 - 变址存储器
RS：寄存器 – 基址存储器
SS：基址存储器 - 基址存储器
SI：基址存储器 - 立即数

扩展编码Expanding Opcodes

- 基本思想

  将操作码的编码长度分成几种固定长的格式。被大多数指令集采用。PDP-11是典型的变长操作码机器。

- 种类

  等长扩展法：4-8-12；3-6-9；…… / 不等长扩展法

- 举例说明如何扩展

  设某指令系统指令字是16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条？

  解:操作码按短到长进行扩展编码

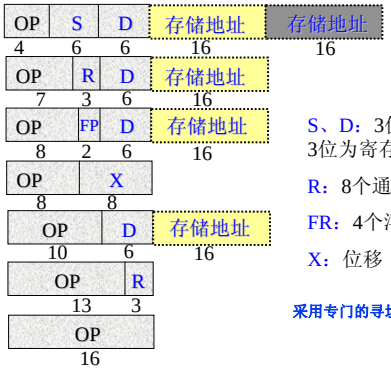
  二地址指令: (0000 – 1110)

  一地址指令: 11110 (00000 – 11111); 11111 (00000 – 00001)

  零地址指令: 11111 (00010 – 11111) (000000 – 11111)

  故零地址指令最多有$30x2^6=15x2^7$ 种

PDP-11中典型指令格式



S、D：3位指定寻址方式，3位为寄存器编号

R：8个通用寄存器之一

FR：4个浮点寄存器之一

X：位移

采用专门的寻址方式字段

Methods of Testing Condition (条件测试方式)

- 条件转移指令通常根据Condition Codes (条件码 / 状态位 / 标志位)进行转移

  通过执行算术指令或显式地由比较和测试指令来设置

  ex: sub r1, r2, r3; r2和r3相减, 结果存储在r1中，并生成标志位ZF、CF等

  bz label; 标志位ZF=1时，转移到label处执行

- 常用的标志有四种：

  NF(SF) -- negative VF(OF) -- overflow CF -- carry ZF -- zero

- 标志位可存放在标志(Flag)寄存器（条件码CC寄存器 / 状态Status寄存器 / 标志寄存器 / 程序状态字PSW寄存器）中

  也可由指定的通用寄存器来存放状态位

  Ex: cmp r1, r2, r3; 比较r2和r3, 标志位存储在r1中

  bgt r1, label; 判断r1是否大于0，是则转移到label处

- 可以将两条指令合成一条指令，即：计算并转移

  Ex: bgt r1, r2, label; 根据r1和r2比较结果，决定是否转移

  不同处理器，对标志位的处理不同！

指令设计风格 -- 按操作数位置指定风格来分

Accumulator: (earliest machines) 累加器型

特点：其中一个操作数总在累加器中

1 address	add A	acc <– acc + mem[A]
1(+x) address	add x A	acc <– acc + mem[A + x]

Stack: (e.g. HP calculator, Java virtual machines) 堆栈型

特点：总是将栈顶两个操作数进行运算，指令无需指定操作数地址

0 address	add	tos <– tos + next

General Purpose Register: (e.g. IA-32, Motorola 68xxx) 通用寄存器型

特点：操作数可以是寄存器或存储器数据

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: (e.g. SPARC, MIPS, PowerPC) 装入/存储型

特点：操作数只能是寄存器数据，只有load/store能访问存储器

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

Comparing Instructions

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

- Code sequence for C = A + B for four classes of instruction sets:

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

指令条数较少

复杂表达式时，累加器型风格指令条数变多，因为所有运算都要用累加器，使得程序中多出许多移入 / 移出累加器的指令！

75年开始，寄存器型占主导地位 (Java Virtual Machine 采用Stack型)

- 寄存器速度快，使用大量通用寄存器可减少访存操作
- 表达式编译时与顺序无关（相对于Stack）

Examples of Register Usage

每条典型ALU指令中的存储器地址个数

每条典型ALU指令中的最多操作数个数

		Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

In VAX(CISC): ADDL (R9), (R10), (R11)

mem[R9] <-- mem[R10] + mem[R11] 一条指令！

In MIPS(RISC):

```
lw   R1, (R10);     load a word
lw   R2, (R11)
add  R3, R1, R2;    R3 <-- R1+R2
sw   R3, (R9);      store a word
```

四条指令！

哪一种风格更好呢？学了第5、6章后会有更深的体会！

指令设计风格– 按指令格式的复杂度来分

按指令格式的复杂度来分，有两种类型计算机：

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

精简指令集计算机RISC (Reduce Instruction Set Computer)

早期CISC设计风格的主要特点

(1) 指令系统复杂

指令多/寻址方式多/指令格式多

(2) 指令周期长

绝大多数指令需要多个时钟周期才能完成

(3) 各种指令都能访问存储器

除了专门的存储器读写指令外，运算指令也能访问存储器。

(4) 采用微程序控制

(5) 有专用寄存器

(6) 难以进行编译优化生成高效目标代码

例如，VAX-11/780小型机

16种寻址方式；9种数据格式；303条指令；

一条指令包括1～2个字节的操作码和下续N个操作数说明符。一个说明符的长度达1 ～10个字节。

复杂指令集计算机CISC

- CISC的缺陷
  - 日趋庞大的指令系统不但使计算机的研制周期变长，而且难以保证设计的正确性，难以调试和维护，并且因指令操作复杂而增加机器周期，从而降低了系统性能。
- 1975年IBM公司开始研究指令系统的合理性问题，John Cocks提出精简指令系统计算机 RISC (Reduce Instruction Set Computer)。
- 对CISC进行测试，发现一个事实：
  - 在程序中各种指令出现的频率悬殊很大，最常使用的是一些简单指令，这些指令占程序的80%，但只占指令系统的20%。而且在微程序控制的计算机中，占指令总数20%的复杂指令占用了控制存储器容量的80%
- 1982年美国加州伯克利大学的RISC Ⅰ，斯坦福大学的MIPS，IBM公司的IBM801相继宣告完成，这些机器被称为第一代RISC机。

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

(简单指令占主要部分，使用频率高！)

back

RISC设计风格的主要特点

(1) 简化的指令系统

指令少/寻址方式少/指令格式少/指令长度一致

(2) 以RR方式工作

除Load/Store指令可访问存储器外，其余指令都只访问寄存器。

(3) 指令周期短

以流水线方式工作， 因而除Load/Store指令外，其他简单指令都只需一个或一个不到的时钟周期就可完成。

(4) 采用大量通用寄存器，以减少访存次数

(5) 采用组合逻辑电路控制，不用或少用微程序控制

(6) 采用优化的编译系统，力求有效地支持高级语言程序

MIPS是典型的RISC处理器，82年以来新的指令集大多采用RISC体系结构

Intel x86因为“兼容”的需要，保留了CISC的风格，同时也借鉴了RISC思想

指令系统举例: Address & Registers

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, stack frame, 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 pointer
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	HI和LO是MIPS内部的乘商寄存器

问题：谁记得GPR是什么？Flags是什么？

指令系统举例：Pentium指令格式

前缀：包括指令、段、操作数长度、地址长度四种类型

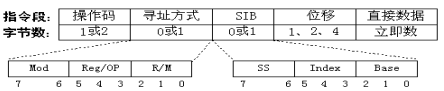
前缀类型：	指令前缀	段前缀	操作数长度	地址长度
字节数：	0或1	0或1	0或1	0或1

指令：指出操作类型和操作数（或地址），含操作码、寻址方式、SIB、位移量和直接数据五部分

位移量和立即数都可是1/2/4B。SIB中基址B和变址I都可是8个GRS中任一个。SS给出比例因子

操作码：opcode; w：与机器模式（16 / 32位）一起确定寄存器位数（AL / AX / EAX）; d：操作方向

寻址方式： mod、r/m、 reg/op三个字段与w字段和机器模式一起确定操作数所在的寄存器编号或有效地址计算方式



变长指令字：1B~17B

变长操作码：4b / 5b / 6b / 7b / 8b /……

变长操作数：Byte / Word / DW / QW

变长寄存器：8位 / 16位 /32位

调用指令自动把返回地址压栈

专门的push/pop指令，自动修改栈指针

ALU指令在Flags中隐含生成条件码

ALU指令中的一个操作数可来自存储器

提供基址加比例索引寻址

问题:是累加器型、通用计算器型、ld/st型？是CISC型、RISC型？

（自学）Pentium处理器的寻址方式

操作数的来源：

- 立即数(立即寻址)：直接来自指令
- 寄存器(寄存器寻址)：来自32位 / 16位 / 8位通用寄存器
- 存储单元(其他寻址)：需进行地址转换

虚拟地址 => 线性地址LA (=> 内存地址)

分段　　分页

指令中的信息：

(1) 段寄存器SR（隐含或显式给出）

(2) 8/16/32位偏移量A （显式给出）

(2) 基址寄存器B （明显给出，任意通用寄存器皆可）

(3) 变址寄存器I （明显给出，除ESP外的任意通用寄存器皆可。）

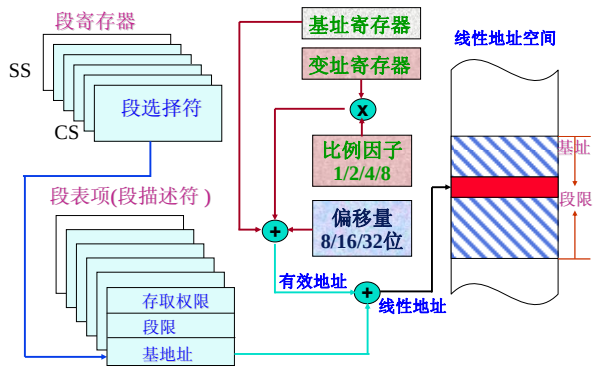
★有比例变址和非比例变址

★比例变址时要乘以比例因子S (1:8位 / 2:16位 / 4:32位 / 8:64位)

（自学） Pentium处理器寻址方式

寻址方式	算法
立即(地址码A本身为操作数)	操作数=A
寄存器(通用寄存器的内容为操作数)	操作数= (R)
偏移量(地址码A给出8/16/32位偏移量)	LA=(SR)+A
基址(地址码B给出基址器编号)	LA=(SR)+(B)
基址带偏移量(一维表访问)	LA=(SR)+(B)+A
比例变址带偏移量(一维表访问)	LA=(SR)+ (I)xS+A
基址带变址和偏移量(二维表访问)	LA=(SR)+(B)+(I) +A
基址带比例变址和偏移量(二维表访问)	LA=(SR)+(B)+(I)xS+A
相对(给出下一指令的地址，转移控制)	转移地址=(PC)+A

（自学）Pentium处理器的存储器寻址



指令系统举例: PowerPC

- RISC型 （类似于MIPS，32位定长操作码、定长指令字），主要不同在于:
  - 提供了特殊的两种变址寻址方式，可减少指令数
    - » 两个寄存器相加变址（基址寄存器和索引寄存器：间接变址寻址）

例：add $t0,$a0,$s3

lw &t1,0($t0) → lw $t1,$a0+$s3

- » 自动变址（变址器自动+1）

例： lw &t0, 4($s3) → lwu $t0, 4($s3)

addi $s3,$s3,4

  - 引入特殊的数据块指令，可减少指令数
    - » 单条指令可传送多达32个字，并可进行存储区数据传送
    - » 提供一个特殊计数寄存器ctr，自动减1，用于循环处理

例：for (i=n; i!=0; i=i-1) { };

Loop: ……

addi $t0,$t0,-1

bne &t0, $zero, loop

→

Loop: ……

bc loop, ctr!=0

SKIP

MMX(Microprocessor Media Extension)指令技术

- 图形/像、音/视频多媒体信息处理特点
  - 多个短整数并行操作(如8位图形像素和16位音频信号)
  - 频繁的乘-累加(如FIR滤波，矩阵运算)
  - ……
- MMX的出发点：
  - 使用专门指令对大量数据进行并行、复杂处理
  - 处理的数据基本单位是8b、16b、32b、64b等
- MMX指令集由Intel提出，1997年首次用于P54C Pentium处理器
  - 引入新的数据类型和通用寄存器
    - » 四种64位紧缩定点整数类型（8x1B、4x1W、2x2W、1x4W）
    - » 8个64位通用寄存器MX0-MX7（借用8个80位浮点寄存器）
  - 采用SIMD(Single Instruction Multi Data)技术
    - » 单条指令同时并行处理多个数据元素

例如，一条指令完成图像中8个像素的并行操作

  - 引入饱和(Situration)运算
    - » 非饱和（环绕）运算：上溢时，高位数据被截去
    - » 饱和运算：上溢时，结果取最大值

例如，图像中像素点的插值运算：a点亮度值F3H，b点亮度值1DH，对a和b线性插值的结果为：

环绕运算：(F3H+1DH)/2=10H/2=08H　插值点的亮度比1DH还低，不合理！

饱和运算：(F3H+1DH)/2=FFH/2=7FH

- 在Intel以后的处理器中又增加了SSE、SSE2、SSE3等指令集

SSE（Streaming SIMD extensions）

SIMD（Single Instrution Multi Data）：单指令多数据技术

第一讲小结

- 指令由“操作码”和“地址码”两部分组成。
- 操作类型
  - 传送 / 算术 / 逻辑 / 移位 / 字符串 / 转移控制 / 调用 / 中断 / 信号同步
- 操作数类型
  - 整数（带符号、无符号、十进制）、浮点数、位、位串
- 地址码的编码要考虑：
  - 操作数的个数
  - 寻址方式：立即 / 寄存器 / 寄间 / 直接 / 间接 / 相对 / 基址 / 变址 / 堆栈
- 操作码的编码要考虑：
  - 定长操作码 / 扩展操作码
- 条件码的生成
  - 四种基本标志：NF / VF / CF / ZF
- 指令设计风格：
  - 按操作数地址指定方式来分：
    - » 累加器型 、堆栈型 、通用寄存器型、load/store型
  - 按指令格式的复杂度来分
    - » 复杂指令集计算机CISC、精简指令集计算机RISC
- 典型指令系统举例
  - Pentium / PowerPC / MMX

以下通过MIPS指令系统，介绍如何在机器语言级表示程序

第二讲 程序的机器级表示

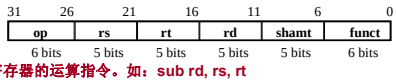
主要内容

- MIPS指令格式
  - R-类型 / I-类型 / J-类型
- MIPS寄存器
  - 长度 / 个数 / 功能分配
- MIPS操作数
  - 寄存器操作数 / 存储器操作数 / 立即数 / 文本 / 位
- MIPS指令寻址方式
  - 立即数寻址 / 寄存器寻址 / 相对寻址 / 伪直接寻址 / 偏移寻址
- MIPS指令类型
  - 算术 / 逻辑 / 数据传送 / 条件分支 / 无条件转移
- MIPS汇编语言形式
  - 操作码的表示 / 寄存器的表示 / 存储器数据表示
- 机器语言的解码（反汇编）
- 高级语言、汇编语言、机器语言之间的转换
- 过程调用与堆栈

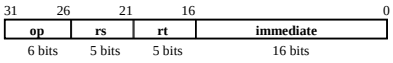
MIPS指令格式

- 所有指令都是32位宽，须按字地址对齐
- 有三种指令格式
  - R-Type
    两个操作数都是寄存器的运算指令。如：sub rd, rs, rt
  - I-Type
    - 一个寄存器，一个立即数的运算指令。如：ori rt, rs, imm16
    - LOAD和STORE指令。如：lw rt, rs, imm16
    - 条件分支指令。如：beq rs, rt, imm16
  - J-Type
    无条件跳转指令。如：j target

R-Type指令



I-Type指令



J-Type指令

MIPS指令字段含义

OP：操作码

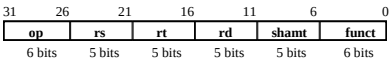
rs：第一个源操作数寄存器

rt：第二个源操作数寄存器

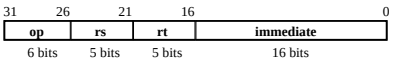
rd：结果寄存器

shamt：移位指令的位移量

R-Type指令



I-Type指令



J-Type指令

func：R-Type指令的OP字段是特定的“000000”，具体操作由func字段给定。如：func=“100000”时，表示“加法”运算。

immediate：立即数或load/store指令和分支指令的偏移地址

target address：无条件转移地址的低26位。将PC高4位拼上26位直接地址，最后添2个“0”就是32位目标地址。为何最后两位要添“0”？

操作码的不同编码定义不同的含义，操作码相同时，再由功能码定义不同的含义！

OP字段的含义（MIPS指令的操作码编码/解码表）

op(31:26)　op=0:R型；op=2/3：J型；其余：I型

28-26 / 31-29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

Back to Load/Store　BACK to Assemble

R-Type指令的解码（op=0时，func字段的编码/解码表）

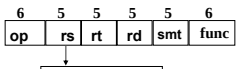
op(31:26)=000000 (R-format), funct(5:0)

2-0 / 5-3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

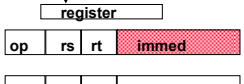
BACK to Assemble

MIPS Addressing Modes（寻址方式）

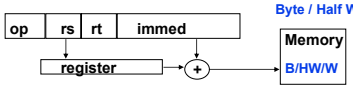
R-format:
Register



I-format:
Immediate

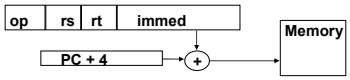


Base或index

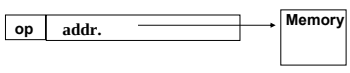


Byte / Half Word / Word

PC-relative



J-format:
Pseudodirect



Example：汇编形式与指令的对应

- 若从存储器取来一条指令为00AF8020H，则对应的汇编形式是什么？

32位指令代码：0000 0000 1010 1111 1000 0000 0100 0000

指令的前6位为000000，根据指令解码表知，是一条R-Type指令，按照R-Type指令的格式

31 6 bits 26	5 bits 21	5 bits 16	5 bits 11	5 bits 6	6 bits 0
op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

得到： rs=00101, rt=01111, rd=10000, shamt=00000, funct=100000

1. 根据R-Type指令解码表，知是 “add”操作（非移位操作)
2. rs、rt、rd的十进制值分别为5、15、16，从MIPS寄存器功能表知:
   rs、rt、rd分别为：$a1、$t7、$s0

故对应的汇编形式为：

add $s0，$a1，$t7

功能：$a1+$t7 → $s0

这个过程称为“反汇编”，可用来破解他人的二进制代码（可执行程序）

ISA.49

Example：汇编形式与指令的对应

- 若MIPS Assembly Instruction: Add $t0,$s1,$s2 → 汇编器 → ?
  则对应的指令机器代码是什么？

从助记符表中查到Add是R型指令，即：

6	5	5	5	5	6
op	rs	rt	rd	smt	func

问题：如何知道是R型指令？

Decimal representaton:

6	5	5	5	5	6
0	17	18	8	0	32
R-Type	$s1	$s2	$t0	No shift	Add

根据汇编指令中的操作码助记符查表能知道是什么格式！

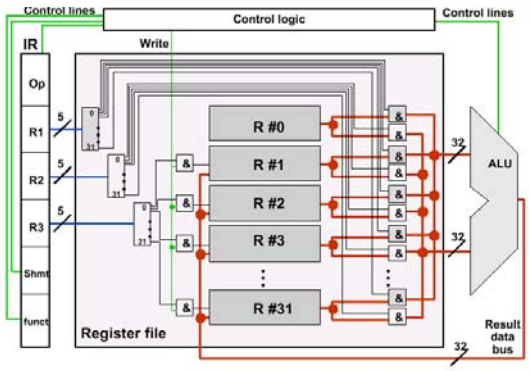
Binary representaton:

6	5	5	5	5	6
000000	10001	10010	01000	00000	100000

这个过程称为“汇编”，所有汇编源程序都必须汇编成二进制机器代码

ISA.50

MIPS Circuits for R-Type Instructions



ISA.51

MIPS R-type指令实现电路的执行过程

Phase1: Preparation (1：准备阶段)

- 装入指令寄存器
- 相应字段送控制逻辑
  - op field (OP字段)
  - funct field (funct字段)
  - shmt field (shmt字段)
- 相应字段送寄存器
  - 第一操作数寄存器
  - 第二操作数寄存器
  - 存放结果的目标寄存器

这个过程描述仅是示意性的，实际上整个过程需要时钟信号的控制，并还有其他部件参与。将在下一章详细介绍。

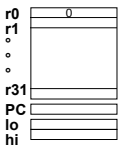
Phase2: Execution(2：执行阶段)

- 寄存器号被送选择器
- 对应选择器输出被激活
- 被选寄存器的输出送到数据线
- 控制逻辑提供：
  - ALU操作码
  - 写信号
- 结果被写回目标寄存器

ISA.52

MIPS指令中寄存器数据和存储器数据的指定

- 寄存器数据指定：
  - 31 x 32-bit GPRs (r0 = 0)
  - 寄存器编号占5 bit
  - 32 x 32-bit FP regs (f0 - f31, paired DP)
  - HI, LO, PC: 特殊寄存器
  - 寄存器功能和2种汇编表示方式



- 存储器数据指定
  - 32-bit machine --> 可访问空间: 2^{32}bytes
  - Big Endian(大端方式)
  - 只能通过Load/Store指令访问存储器数据
  - 数据地址通过一个32位寄存器内容加16位偏移量得到
  - 16位偏移量是带符号整数
  - 数据要求按边界对齐

SKIP

ISA.53

MIPS寄存器的功能定义和两种汇编表示

Name	number	Usage	Reserved on call?
zero	0	constant value =0(恒为0)	n.a.
at	1	reserved for assembler(为汇编程序保留)	n.a.
v0 – v1	2 – 3	values for results(过程调用返回值)	no
a0 – a3	4 – 7	Arguments(过程调用参数)	yes
t0 – t7	8 – 15	Temporaries(临时变量)	no
s0 – s7	16 – 23	Saved(保存)	yes
t8 – t9	24 – 25	more temporaries(其他临时变量)	no
k0 – k1	26 – 27	reserved for kernel(为OS保留)	n.a.
gp	28	global pointer(全局指针)	yes
sp	29	stack pointer (栈指针)	yes
fp	30	frame pointer (帧指针)	yes
ra	31	return address (过程调用返回地址)	yes

Registers are referenced either by number—$0…$31, or by name —$t0,$s1…$ra.

zero	at	v0-v1	a0 - a3	t0 - t7	s0 - s7	t8 - t9	k0 - k1	gp	sp	fp	ra
0	1	2 - 3	4 - 7	8 --- 15	16 --- 23	24 - 25	26 - 27	28	29	30	31

BACK to Assemble BACK to Procedure BACK to last

ISA.54

MIPS arithmetic and logic 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
multiply	mult $2,$3	Hi, Lo = $2×$3	64-bit signed product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
Move from Hi	mfhi $1	$1=Hi	get a copy of Hi
Move from Lo	mflo $1	$1=lo	

Instruction	Example	Meaning	Comment
and	and $1,$2,$3	$1 = $2 & $3	Logical AND
or	or $1,$2,$3	$1 = $2 | $3	Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	Logical NOR

这里没有全部列出，还有其他指令，如addu(不带溢出处理)， addui等

问题：Intel没有分add还是addu，会不会有问题？

ISA.55

Example：算术运算

E.g. f= (g+h) - (i+j),
assuming f, g, h, i, j be assigned to $1, $2, $3, $4, $5

```
add $7, $2, $3
add $8, $4, $5
sub $1, $7, $8
```

寄存器资源由编译器分配！
通常将简单变量尽量分配在寄存器中，为什么？
程序中的常数如何处理呢？

E.g. f= (g+100) - (i+50) →

```
addi $7, $2, 100
addi  $8, $4, 50
sub $1, $7, $8
```

```
addi $7, $2, 65000
addi  $8, $4, 50
sub $1, $7, $8
```

← E.g. f= (g+65000) - (i+50)

问题：以下程序如何处理呢？

指令设计时必须考虑这种情况！MIPS有一条专门指令，后面介绍。

ISA.56

MIPS data transfer instructions

Instruction	Comment	Meaning
SW $3, 500($4)	Store word	$3 →($4+ 500)
SH $3, 502($2)	Store half	Low Half of $3 →($2+ 502)
SB $2, 41($3)	Store byte	LQ of $2 →($3+ 41)
LW $1, -30($2)	Load word	($2-30) → $1
LH $1, 40($3)	Load half	($3+ 40) → LH of $1
LB $1, 40($3)	Load byte	($3+ 40) → LQ of $1

操作数长度的不同由不同的操作码指定。
问题：为什么需要不同长度的操作数？

高级语言中的数据类型有char，short，int，long,……等，故需要存取不同长度的操作数；操作数长度和指令长度没有关系

ISA.57

Example (Base register)

Assume A is an array of 100 words, and compiler has associated the variables g and h with the register $1 and $2.
Assume the base address of the array is in $3. Translate

g = h + A[8]

```
lw $4, 8($3);      $4 <-- A[8]
add $1, $2, $4;
```

offset or displacement（偏移量）

```
lw $4, 32($3);
add $1, $2, $4
sw $1, 48($3)
```

A[12] = h+A[8]

base register（基址寄存器）

问题：如果在一个循环体内执行：g = h + A[i]，则能否用基址寻址方式？

不行，因为循环体内指令不能变，故首地址A不变，只能把下标i放在变址寄存器中，每循环一次下标加1，所以，不能用基址方式而应该用变址方式。

ISA.58

Example (Index Register)

Assume A is an array of 100 words, and compiler has associated the variables g and i with the register $1, $5.
Assume the base address of the array is in $3. Translate

g = g + A[i]

```
addi  $6, $0, 4;     $6 = 4
mult  $5, $6;        Hi,Lo = i*4
mflo  $7;            $6 = i*4, assuming i is small

add $4, $3, $7;      $4 <--address of  A[i]
lw  $4, 0($4);
add $1, $2, $4
addi $5, $5, 1
```

Index Register（变址寄存器）

Why should index i multiply 4 ?
How do speedup i multiply 4 ?
Index mode suitable for Array!

问题：若循环执行 g=g+A[i]，怎样使上述循环体内的指令条数减少？

用$5做变址器，每次&5加4 或 用移位指令　　若增设专门的"变址自增（即自动变址）"指令则可使循环更短

ISA.59

MIPS的call/return/ jump/branch和compare指令

Instruction	Example	Meaning
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	
jump	j 10000	go to 10000
	Jump to target address	

call / return

Pseudoinstruction blt, ble, bgt, bge
not implemented by hardware, but synthesized by assembler

set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0

按补码比较大小

问题：指令中立即数是多少？ 100=0064H

branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100

问题：指令中立即数是多少？ 25=0019H

汇编中给出的是立即数符号扩展后乘以4得到的值

BACK to Procedure

ISA.60

Example：if-then-else语句和“=”判断

```
if (i = = j)
     f = g+h ;
else
     f = g-h ;
```

Assuming variables i, j, f, g, h, ~ $1, $2, $3, $4, $5

```
        bne $1, $2, else        ; i!=j, jump to else
        add $3, $4, $5
        j   exit                ; jump to exit
else:   sub $3, $4, $5
exit:
```


Example：“less than”判断

if (a < b) f = g+h ; else f = g-h ;
Assuming variables a, b, f, g, h, ~ $1, $2, $3, $4, $5

```
        slt   $6, $1, $2        ; if a<b, $6=1, else $6=0
        bne $6, $zero, else     ; $6!=0, jump to else
        add $3, $4, $5
        j   exit                ; jump to exit
else:   sub $3, $4, $5
exit:
```

```
        slt   $6, $1, $2        ; if a<b, $6=1, else $6=0
        beq $6, $zero, else     ; $6=0, jump to else
        add $3, $4, $5
        j   exit                ; jump to exit
else:   sub $3, $4, $5
exit:
```


Example：Loop循环

```
Loop:  g = g +A[i];
       i = i+ j;
       if (i != h) go to Loop:
```

Assuming variables g, h, i, j ~ $1, $2, $3, $4 and base address of array is in $5

```
Loop:  add $7, $3, $3      ; i*2
       add $7, $7, $7      ; i*4
       add $7, $7, $5
       lw $6, 0($7)        ; $6=A[i]
       add $1, $1, $6      ; g= g+A[i]
       add $3, $3, $4
       bne $3, $2, Loop
```

加法比乘法快！

也可用移位来实现乘法！

$3中是i，
$7中是i*4

汇编程序使得编译器和汇编语言程序员不必计算分支指令的地址，而只要用标号即可！汇编器完成地址计算

Example：过程调用

```
int i;
void set_array(int num)
{
      int array[10];
      for (i = 0; i < 10; i ++) {
          arrar[i] = compare (num, i);
      }
}

int compare (int a, int b)
{
      if ( sub (a, b) >= 0)
          return 1;
      else
          return 0;
}

int sub (int a, int b)
{
   return a-b;
}
```

i是全局静态变量

array数组是局部变量

set_array是调用过程
compare是被调用过程

compare是调用过程
sub是被调用过程

问题1：过程调用对应的机器代码如何表示？

需要解决：

1. 如何从调用程序把参数传递到被调用程序？
2. 如何从调用程序的执行转移到被调用程序执行？
3. 如何从被调用程序返回到调用程序执行？
4. 如何保证调用程序中寄存器内容不被破坏？

Procedure Call and Stack(过程调用和栈)

- 过程调用的执行步骤（假定过程P调用过程Q）：
  - 将参数放到Q能访问到的地方
  - 将P中的返回地址存到特定的地方，将控制转移到过程Q

  （在调用过程P中完成）
  - 为Q的局部变量分配空间（局部变量临时保存在栈中）
  - 执行过程Q
  - 将Q执行的返回结果放到P能访问到的地方
  - 取出返回地址，将控制转移到P，即返回到P中执行

  （在被调用过程Q中完成）
- MIPS中用于过程调用的指令（见MIPS过程调用指令）
- MIPS规定少量过程调用信息用寄存器传递（见MIPS寄存器功能定义）
- 如果过程中用到的参数超过4个，返回值超过2个，怎么办？
  - 更多的参数和返回值要保存到存储器的特殊区域中
  - 这个特殊区域为：栈(Stack)

  一般用“栈”来传递参数、保存返回地址、临时存放过程的局部变量等。为什么？

  便于递归调用！

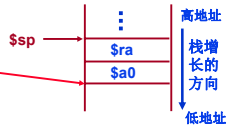
栈(Stack)的概念

- 栈的基本概念
  - 是一个“先进后出”队列
  - 需一个栈指针指向栈顶元素
  - 每个元素长度一致
  - 用“入栈”（push）和“出栈”（pop）操作访问栈元素
- MIPS中栈的实现
  - 用栈指针寄存器$sp来指示栈顶元素
  - 每个元素的长度为32位，即：一个字(4个字节)
  - “入栈”和“出栈”操作用 sw / lw 指令来实现，需用add / sub指令调整$sp的值，不能像x86那样自动进行栈指针的调整

    （有些处理器有专门的push/pop指令，能自动调整栈指针。如x86系列处理器）
  - 栈生长方向：从高→低地址“增长”，而取数/存数的方向是低→高地址（大端方式）
    - » 每入栈1字，$sp- 4→$sp ； 每出栈1字，$sp+4 →$sp

例：若将返回地址$ra和参数$a0保存到栈，则指令序列为：

```
sub  $sp, $sp, 8
sw   $ra, 4($sp)
sw   $a0, 0($sp)
```

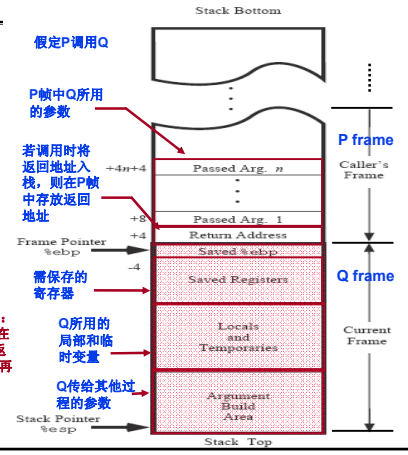


栈帧的概念

- 每个过程都有自己的栈区，称为栈帧（Stack frame），即：过程的帧（procedure frame）
- 栈由若干栈帧组成
- 用专门的帧指针寄存器指定起始位置
- 当前栈帧范围由帧指针和栈指针之间
- 程序执行时，栈指针可移动，帧指针不变所以，过程内对栈信息的访问大多通过帧指针进行

MIPS返回地址处理有所不同：调用指令jal把返回地址保存在&ra中，Q再把&ra入栈，Q返回前将&ra出栈，返回指令jr再根据&ra返回到调用过程P

MIPS中帧指针寄存器为&fp



MIPS中的过程调用（假定P调用Q）

- 程序可访问的寄存器组是所有过程共享的资源，给定时刻只能被一个过程使用，因此，一个过程中使用的寄存器的值不能被另一个过程覆盖！
- MIPS的寄存器使用约定：
  - 保存寄存器$s0 ~$s7 的值在从被调用过程返回后还要被用，被调用者需要保留
  - 临时寄存器$t0 ~$t9的值在从被调用过程返回后不需要被用（需要的话，由调用者保存），被调用者可以随意使用
  - 参数寄存器$a0~$a3在从被调用过程返回后不需要被用（需要的话，由调用者保存在栈帧或其他寄存器中），被调用者可以随意使用
  - 全局指针寄存器$gp的值不变
  - 在过程调用时帧指针寄存器$fp用栈指针寄存器$sp- 4来初始化
- 需在被调用过程Q中入栈保存的寄存器（称为被调用者保存）
  - 返回地址$ra（如果Q又调用R，则$ra内容会被破坏，故需保存）
  - 保存寄存器$s0 ~$s7 (从Q返回后P可能还会用到，Q中用的话就被破坏，故需保存)
- 除了上述寄存器以外，所有局部数组和结构类型变量也要入栈保存
- 如果局部变量和临时变量发生寄存器溢出（寄存器不够分配），则也要入栈
- 每个处理器对栈帧规定的“调用者保存”和“被调用者保存”的寄存器可能不同。例：
  - x86处理器中返回地址保存在调用过程栈帧中；而MIPS则在被调用过程中保存
  - x86处理器中调用参数都保存在调用过程栈帧中；而MIPS则在被调用过程中保存额外参数
  - X86处理器中调用过程的帧指针保存在被调用过程的栈帧中；MIPS也一样。

Example in C: swap

假定swap作为一个过程被调用，temp对应$t0, 变量v 和 k分别对应$s0和$s1

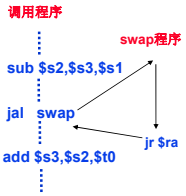
写出对应的MIPS汇编代码。 问题：上述假设有何问题？ 参数v和k应该在$a0和$a1

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

```
sll   $s2, $a1, 2     ; mulitply k by 4
addu  $s2 $s2, $a0    ; address of v[k]
lw    $t0, 0($s2)     ; load v[k]
lw    $s3, 4($s2)     ; load v[k+1]
sw    $s3, 0($s2)     ; store v[k+1] into v[k]
sw    $t0, 4($s2)     ; store old v[k] into v[k+1]
```

在调用过程中用指令“jal swap” 进行swap调用

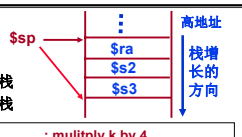
jal --- jump and link (跳转并链接)
$31 = PC+4 ; $31=$ra
goto swap



问题1：如果在swap中不保存$s2，则caller会发生什么情况？
caller中$s2的值被swap破坏！ 须在swap的栈帧中保存$s2
问题2：如果在swap中不保存$t0，则caller会发生什么情况？
$t0约定由caller保存，故无须在swap的栈帧中保存$t0

swap: MIPS中的一个过程示例

```
swap:
addi  $sp,$sp, –12    ; 栈增长3个
sw    $31, 8($sp)     ; 返回地址入栈
sw    $s2, 4($sp)     ; 保留寄存器$s2入栈
sw    $s3, 0($sp)     ; 保留寄存器$s3入栈
```




```
....
sll   $s2, $a1, 2     ; mulitply k by 4
addu  $s2, $s2, $a0   ; address of v[k]
lw    $t0, 0($s2)     ; load v[k]
lw    $s3, 4($s2)     ; load v[k+1]
sw    $s3, 0($s2)     ; store v[k+1] into v[k]
sw    $t0, 4($s2)     ; store old v[k] into v[k+1]
lw    $s3, 0($sp)     ; 恢复$s3
lw    $s2, 4($sp)     ; 恢复$s2
lw    $31, 8($sp)     ; 恢复$31（$ra）
addi  $sp,$sp, 12     ; 退栈
jr    $31             ; 从swap返回到调用过程
```

问题：是否一定要将返回地址（$31）保存到栈帧中？
如果swap是叶子过程，则无需保存返回地址到栈中，为什么？ $ra的内容不会被破坏！
如果将所有内部寄存器都用临时寄存器(如$t1等)，则叶子过程swap的栈帧为空，即上述黑色指令都可去掉

过程调用举例

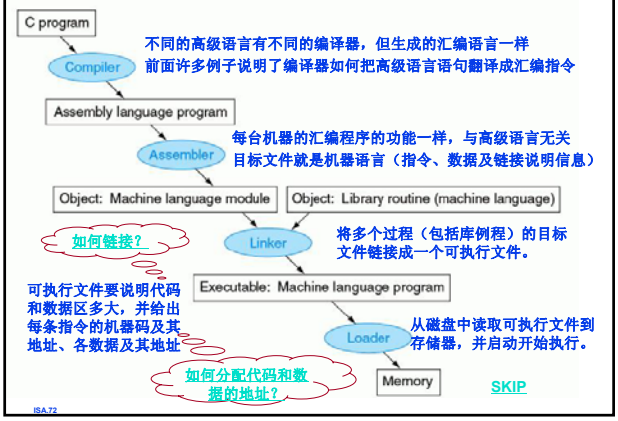
```
int i;                                  i是全局静态变量
void set_array(int num)
{
    int array[10];                      array数组是局部变量
    for (i = 0; i  < 10; i ++) {
        arrar[i] = compare (num, i);    set_array是调用过程
    }                                   compare是被调用过程
}

int compare (int a, int b)
{
    if ( sub (a, b) >= 0)               compare是调用过程
        return 1;                       sub是被调用过程
    else
        return 0;
}

int sub (int a, int b)
{
   return a-b;
}
```

问题1：编译器如何为全局变量和局部变量分配空间？
问题2：执行set_array的结果是什么？

程序的翻译、链接和加载（自学）



MIPS程序和数据的存储器分配（自学）

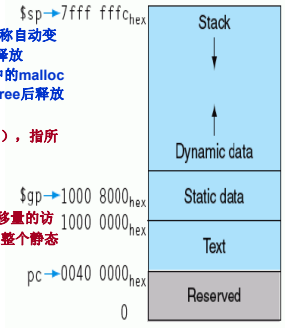
- 每个MIPS程序都按如下规定进行存储器分配
- 每个可执行文件都按如下规定给出代码和数据的地址

栈区位于堆栈高端，堆区位于堆栈低端
- 栈(Stack)区存放每个过程的局部数据（也称自动变量），从高往低长，从被调用过程返回后释放
- 堆(heap)区存放程序的动态数据（如：C中的malloc申请区域、链表等），从低往高长，执行free后释放

静态数据区存放的是全局变量（也称静态变量），指所有过程之外声明的变量和用Static声明的变量
从固定的0x1000 0000处 开始存放

全局指针$gp固定为0x1000 8000，其16位偏移量的访问范围为0x1000 0000 到0x1000 ffff，可遍及整个静态数据区的访问

程序代码从固定的0x0040 0000处开始存放
故PC的初始值为0x0040 0000



BACK

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

Object file header	过程A的目标文件		
	Name	Procedure A	
	Text size	100_{hex}	代码的长度为0x100
	Data size	20_{hex}	数据的长度为0x20
Text segment	Address	Instruction	
链接前地址总是从0开始	0	lw $a0, 0($gp)	0是由x待定的地址
实际是指令机器码	4	jal 0	0是由B待定的地址
	…	…	
Data segment	0	(X)	
链接前地址总是从0开始	…	…	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	X的地址待定
	B	—	B的地址待定

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

Object file header	过程B的目标文件		
	Name	Procedure B	
	Text size	200_{hex}	代码的长度为0x200
	Data size	30_{hex}	数据的长度为0x30
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	0是由Y待定的地址
	4	jal 0	0是由A待定的地址
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

目标文件的链接（自学）

过程A和过程B分别编译、汇编成目标文件，链接后生成一个可执行文件

Executable file header	生成的可执行文件	
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
代码地址总是从0040 0000开始	0040 0000 hex	lw $a0, 8000 hex ($gp)
	0040 0004 hex	jal 40 0100 hex
	…	…
过程B从A后的0x100开始	0040 0100 hex	sw $a1, 8020 hex ($gp)
	0040 0104 hex	jal 40 0000 hex
	…	…
Data segment	Address	0x1000 0000=? + 0x1000 8000
静态数据地址从0x1000 0000开始	1000 0000 hex	(X)
	…	…
过程B从A后的0x20开始	1000 0020 hex	(Y)
	…	…

0x1000 8000+0xFFFF 8000=0x1000 0000 → ?= 8000 (符号扩展后为FFFF 8000)

BACK

MIPS指令中位的指定和逻辑运算

逻辑数据表示
- 用一位表示 真：1 -True / 假：0-False
- N位二进制数可表示N个逻辑数据

逻辑运算
- 按位进行， 如: And / Or / Shift Left / Shift Right等

位的指定
- 设置某位的值：
  - 清0: 与掩码（1…101…1）相“与”
  - 置1：与位串（0…010…0）相“或”
- 判断某位的值：
  - 是否为0: 与位串（0…010…0）相“与”后，是否为0
  - 是否为1：与位串（ 0…010…0 ）相“与”后，是否不为0

MIPS中的移位指令（sll / srl）

例 : srl $t2,$s0,8
$s0右移8位后送$t2

op	rs	rt	rd	shamt	funct
000000	00000	10000	01010	01000	000010

逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，机器本身不能识别，需靠指令的类型来识别。包括后面所讲的字符数据等都一样。

MIPS指令中常数的指定

- 程序中经常需要使用常数，例如：
  - C编译器gcc中52%的算术指令用到常数
  - 电路模拟程序Spice中69%的算术指令用到常数
- 指令中如何取得常数
  - 若程序装入时，常数已在内存中，则需用load指令读入寄存器
  - 在指令中用一个“立即数”来使用常数

例1：i=i+4; Assuming variable i ~ $1
则： addi $1, $1, 4

例2：if (i<20) ….; Assuming variable i ~ $1
则： slti $3, $1, 20 ; if (i<20) $3=1 else $3=0

如果常数的值用16位无法表示，怎么办？

用lui指令把高16位送到寄存器的高16位，再把低16位加到该寄存器中。

例3：将“0000 0000 0011 1101 0000 0000 0000 1000”送$3中
则： lui $3, 61
  addi $3, $3, 8

MIPS指令中如何表示文本字符串

- 有些情况下，程序需要处理文本。例如：
  - 西文文本由ASCII码字符构成字符串
  - Java等语言使用Unicode编码构成字符串
  - 汉字文本使用的汉字编码字符构成字符串
- 字符串的表示
  - 由一个个字符组成，长度不定。有三种表示方式：
    - » 字符串的首字节记录长度
    - » 用其他变量来记录长度（即：用“struc”类型来描述）
    - » 字符串末尾用一个特殊字符表示。
      如：C语言用字符（NULL）来标记字符串结束
- 如何在指令中表示字符
  - ASCII字符串，每个字符由8位组成，用 “lb/sb”指令存/取一个字节
  - Unicode和汉字字符串，每个字符有16位，用 “lh/sh”指令存/取两个字节

例1：x[i] = y[j]; Assuming variables i,j ~ $1,$2, base address x,y ~ $3,$4
则：

```
add  $5, $3, $1        ; $5=the address of x[i]
add $6, $4, $2         ; $6=the address of y[j]
lb   $7, 0($6)         ; $7=y[j]
sb   $7, 0($5)         ; x[i]=$7
```

SKIP

数据类型和MIPS指令的对应

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
char	—	lb, sb, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

C语言中的“char”为8位， Java语言中的“char”为16位(Unicode)

本讲小结

- MIPS指令格式
  - R-类型 / I-类型 / J-类型
- MIPS寄存器
  - 长度 / 个数 / 功能分配
- MIPS操作数
  - 寄存器操作数 / 存储器操作数 / 立即数 / 文本
- MIPS指令寻址方式
  - 立即数寻址 / 寄存器寻址 / 相对寻址 / 伪直接寻址 / 偏移寻址
- MIPS指令类型
  - 算术 / 逻辑 / 数据传送 / 条件分支 / 无条件转移
- MIPS汇编语言形式
  - 操作码的表示 / 寄存器的表示 / 存储器数据表示
- 机器语言的解码（反汇编）
- 高级语言、汇编语言、机器语言之间的转换
  - 运算表达式 / If语句 / 循环 / 数组访问 / 过程 / 堆栈 / 栈帧
- 其他指令系统：PowerPC、80x86
- CISC vs. RISC

本章总结1

- 指令格式
  - 定长指令字：所有指令长度一致
  - 变长指令字：指令长度有长有短
- 操作类型
  - 数据传送：数据在寄存器、主存单元、栈顶等处进行传送
  - 操作运算：各种算术运算、逻辑运算
  - 字符串处理：字符串查找、扫描、转换等
  - I/O操作： 与外设接口进行数据/状态/命令信息的交换
  - 程序流控制：条件转移、无条件转移、转子、返回等
  - 系统控制：启动、停止、自愿访管、空操作等
- 操作数类型（以Pentium处理器数据类型为例）
  - 序数或指针：8位、16位、32位无符号整数表示
  - 整数：16位、32位、64位三种补码表示的整数
  - 实数：IEEE754浮点数格式
  - 十进制数：18位十进制数，用80个二进位表示
  - 字符串：字节为单位的字符序列，一般用ASCII码表示
- 操作数宽度：有多种，如：字节、16位、32位、64位等

本章总结3

- 寻址方式
  - 立即：地址码直接给出操作数本身
  - 直接：地址码给出操作数所在的内存单元地址
  - 间接：地址码给出操作数所在的内存单元地址所在的内存单元地址
  - 寄存器：地址码给出操作数所在的寄存器编号
  - 寄存器间接：地址码给出操作数所在单元的地址所在的寄存器编号
  - 堆栈：操作数约定在堆栈中，总是从栈顶取数或存数
  - 偏移寻址：用基地址+形式地址得到操作数所在的内存单元地址，包括三种：
    - » 变址寻址：地址码给出一个形式地址，并且隐含或明显地指定一个寄存器作为变址寄存器，变址寄存器的内容（变址值）和形式地址相加，得到操作数的有效地址，
    - » 相对寻址：指令中的形式地址给出一个位移量D，而基准地址由程序计数器PC提供。即：有效地址EA=（PC）+ D
    - » 基址寻址：地址码给出一个形式地址，作为位移量，并且隐含或明显地指定一个寄存器作为基址寄存器，基址寄存器的内容和形式地址相加，得到操作数的有效地址

本章总结4

- 指令系统风格：决定了处理器的设计
  - 按地址码指定风格来分
    - 累加器型：一个操作数和结果都隐含在累加器中
    - 堆栈型：操作数和结果都隐含在堆栈中
    - 通用寄存器型：操作数明显地指定在哪个通用寄存器中
    - 装入/存储型：运算类指令的操作数只能在寄存器中，只有装入(Load)指令和存储(Store)指令才能访问内存
  - 按指令系统的复杂度来分
    - CISC：复杂指令系统计算机
    - RISC：精简指令系统计算机

谬误和陷阱

- 谬误1：功能更强的指令意味着更高的性能。
  - 反例：块拷贝或块比较指令
    - » 将循环展开，重复多次执行简单指令，大约快1.5倍
    - » 使用更长的浮点寄存器重复执行拷贝或比较，大约快2倍
- 谬误2：使用汇编语言编程能获得最高的性能。
  - 直接用汇编语言编程或让程序员提供编译指示反而没有编译器自动生成的代码更优化。
  - 汇编语言程序比高级语言程序更长，所以编码调试时间更长、可移植性差、难于维护。
- 陷阱1：现在基本上所有机器都采用字节编址，计算数据的地址时，应该考虑其长度占几个字节。
- 陷阱2：不同的机器在进行数据存放时，采用的顺序可能是小端或大端方式，小端方式以反序形式显示数据的值。
- 陷阱3：在自动变量的定义过程外使用指向该变量的指针，会发生混乱。
  - 过程体内的局部变量（也被称为自动(automatic)变量）会随着过程的调用在栈帧中生成，并随着过程的返回在栈帧中消退。
  - 局部数组的指针在过程体外引用时，会发生错误。

结论

重要结论

- 简单来自于规整
- 指令在程序中出现的频率是不同的
- 尽量加快常用操作的速度
- 好的设计需要在各种因素中进行权衡
- 高级语言中的不同的结构对应不同类别的指令
  - 算术运算指令对应于赋值语句
  - 出现数组或结构等的数据操作时，需要访存指令
  - 条件分支指令用于if语句和循环结构
  - 无条件转移用于Case/Switch语句结构
  - 调用指令、入/出栈指令用于过程调用
  - .....

本章作业

- 2.（7）（8），3，4，7，8，11，12，13，14

下星期二（5月12号）交作业