Chap.2 Instructions：Language of the Computer

Chap.2 指令系统：计算机的语言

Chap.2： Instructions: Language of the Computer

Chap.2 指令系统：计算机的语言

2.1 Introduction

设计目标

- computer designers have a common goal：

  To find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and power.

- Instruction / Instruction Set：指令 / 指令集

  the words of a computer's language are called instructions, and its vocabulary is called an instruction set.

2.1 Introduction

指令集实例 – MIPS架构

- MIPS： Microprocessor without Interlocked Pipeline Stages
- 是一种采取精简指令集（RISC）的处理器架构，1981年出现，由MIPS科技公司开发并授权，广泛被使用在许多电子产品、网络设备、个人娱乐装置与商业装置上。最早的MIPS架构是32位，最新的版本已经变成64位。
- 1981年，斯坦福大学教授John LeRoy Hennessy领导他的团队，实作出第一个MIPS架构的处理器，通过指令管线化来增加CPU运算的速度。1984年，John L. Hennessy教授离开斯坦福大学，创立MIPS科技公司。1985年设计出R2000芯片，1988年将其改进为R3000芯片。之后有R4000、RM9000x2等多种系列。
- 2002年，中国科学院计算所开始研发龙芯处理器，采用MIPS架构，但未经MIPS公司的授权，遭到侵权的控告。2007年，中国科学院与MIPS公司达成和解，得到正式授权。
- 更多细节可参阅 http://en.wikipedia.org/wiki/MIPS_architecture。

指令集实例 – MIPS架构

MIPS Operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1`=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = 20 * 2^{16}	Loads constant in upper 16 bits

指令集实例 – MIPS架构

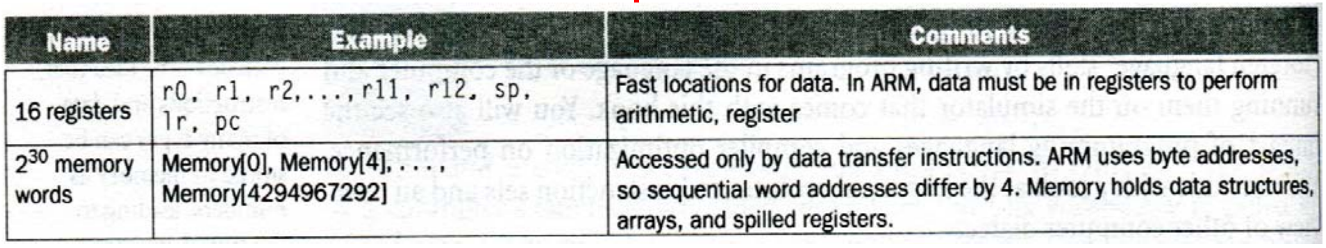
MIPS assembly language

Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

FIGURE 2.1 MIPS assembly language revealed in this chapter.

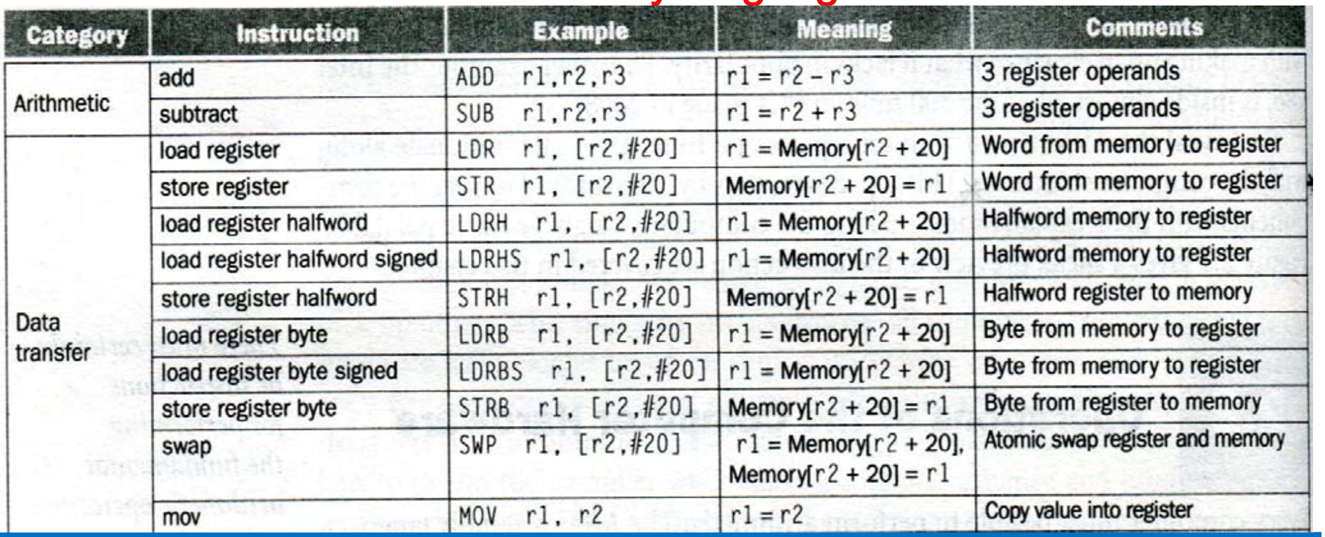
指令集实例－ARM架构

ARM Operands



Name	Example	Comments
16 registers	r0, r1, r2,...,r11, r12, sp, lr, pc	Fast locations for data. In ARM, data must be in registers to perform arithmetic, register
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. ARM uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

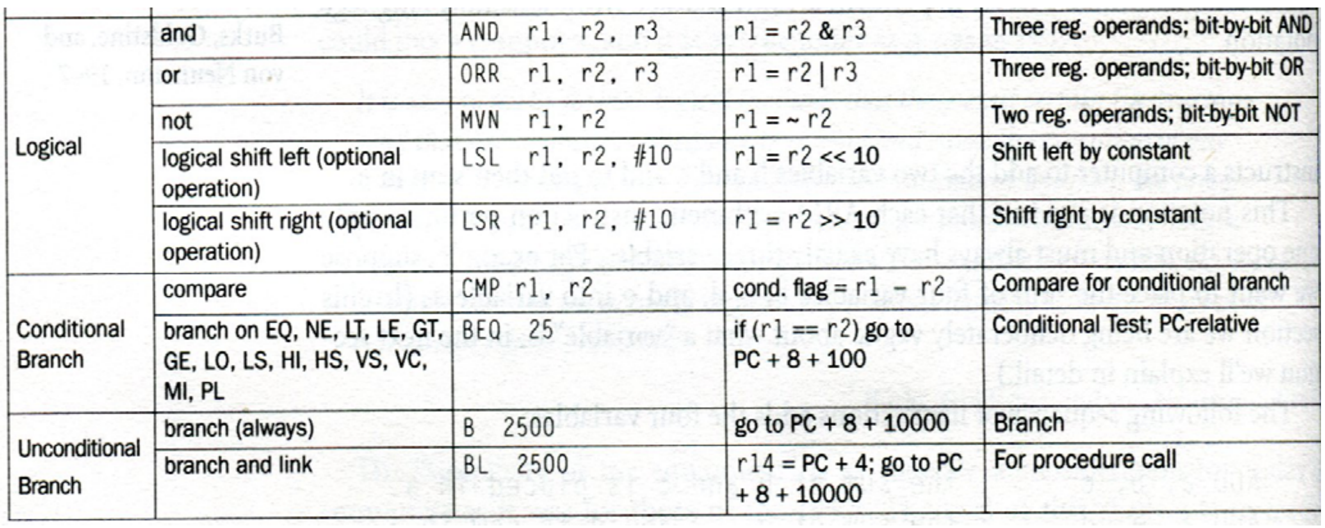
ARM assembly language



Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD r1,r2,r3	r1 = r2 – r3	3 register operands
	subtract	SUB r1,r2,r3	r1 = r2 + r3	3 register operands
Data transfer	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	Word from memory to register
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	Word from memory to register
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	Halfword memory to register
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	Halfword memory to register
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	Halfword register to memory
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	Byte from memory to register
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	Byte from memory to register
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	Byte from register to memory
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	Atomic swap register and memory
	mov	MOV r1, r2	r1 = r2	Copy value into register

指令集实例－ARM架构

ARM assembly language



Logical	and	AND r1, r2, r3	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	ORR r1, r2, r3	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	MVN r1, r2	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	CMP r1, r2	cond. flag = r1 – r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	B 2500	go to PC + 8 + 10000	Branch
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

FIGURE 2.1 ARM assembly language revealed in this chapter.

2.2 Operations of the Computer Hardware

指令硬件操作

■ 自然语言描述的操作：

$$a = b + c + d + e$$

■ MIPS汇编语言描述（类汇编语言（变量））：

```
add  a, b, c     #  The sum of b and c is placed in a.
add  a, a, d     #  The sum of b, c, and d is now in a.
add  a, a, e     #  The sum of b, c, d, and e is now in a.
```

■ ARM汇编语言描述（类汇编语言（变量））：

```
add  a, b, c     ;  The sum of b and c is placed in a.
add  a, a, d     ;  The sum of b, c, and d is now in a.
add  a, a, e     ;  The sum of b, c, d, and e is now in a.
```

2.2 Operations of the Computer Hardware

Design Principle 1: Simplicity favors regularity

设计原理1: 简单即规整

【例】Compiling Two C Assignment Statements into MIPS

■ C语言描述：

```
a = b + c ;
d = a – e ;
```

☐ 经编译后

■ MIPS指令：

```
add a, b, c
sub d, a, e
```

2.2 Operations of the Computer Hardware

Design Principle 1: Simplicity favors regularity

设计原理1: 简单即规整

【例】Compiling a Complex C Assignment Statement into MIPS

■ C语言描述：

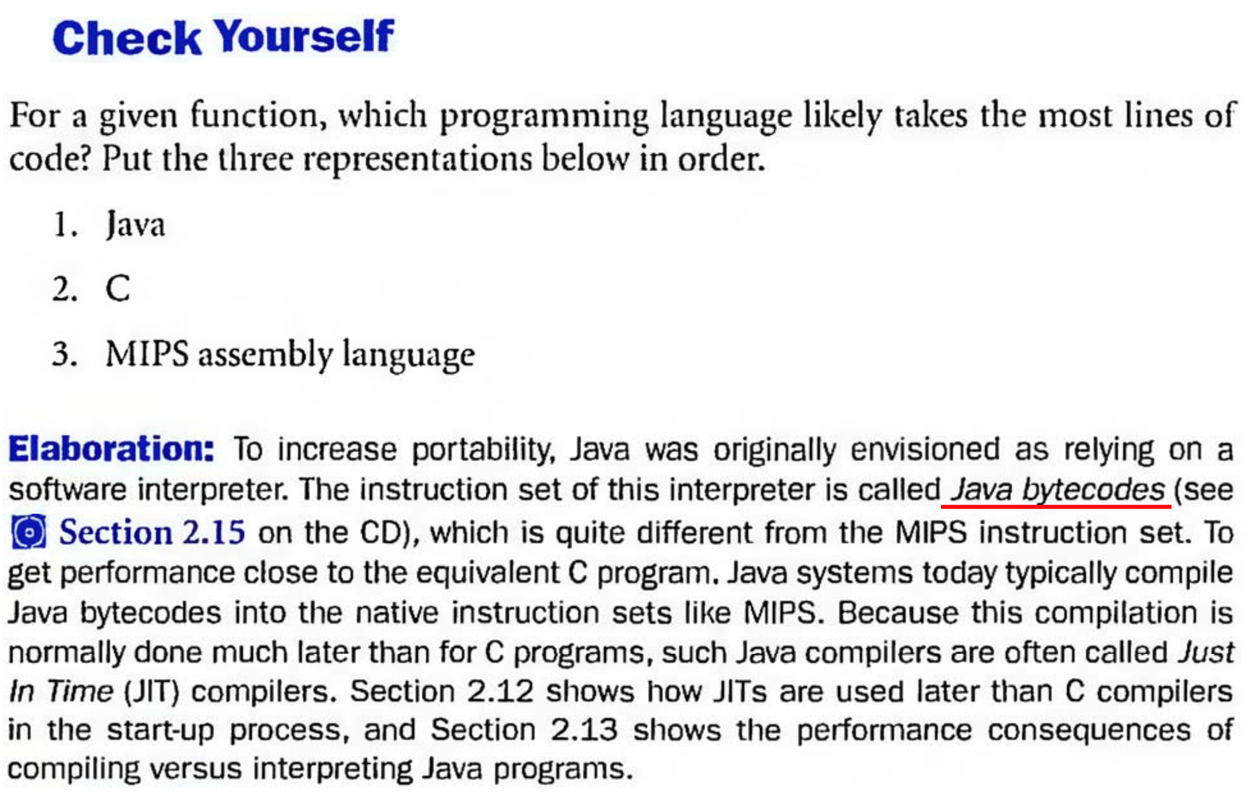
```
f = （ g + h ） - （ i + j ）；
```

□ 经编译后

■ MIPS指令：

```
add t0, g, h     #  temporary variable t0 contains （g + h）
add t1, i,  j    #  temporary variable tl contains （i + j）
sub f,  t0, t1   #  f gets t0 – t1, which is （g + h） - （i + j）
```

2.2 Operations of the Computer Hardware



Check Yourself

For a given function, which programming language likely takes the most lines of code? Put the three representations below in order.

1. Java
2. C
3. MIPS assembly language

Elaboration: To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called *Java bytecodes* (see Section 2.15 on the CD), which is quite different from the MIPS instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like MIPS. Because this compilation is normally done much later than for C programs, such Java compilers are often called *Just In Time* (JIT) compilers. Section 2.12 shows how JITs are used later than C compilers in the start-up process, and Section 2.13 shows the performance consequences of compiling versus interpreting Java programs.

2.2 Operations of the Computer Hardware

Java bytecodes （FIGURE 2.15.8 Java bytecode architecture versus MIPS）

Category	Operation	Java bytecode	Size (bits)	MIPS instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{–1,0,1,2,3,4,5}	8	addi	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3

2.3 Operands of the Computer Hardware

硬件操作数的来源

- 运算操作数可来自于多处，但计算机硬件只选择有限的几种方式。（why？）
- 寄存器（register）：是位于CPU内，距离运算器最近、具有最快访问速度的存储器。
- 寄存器对程序员是可见的。

- MIPS的操作数设计为只能来自于寄存器。
- x86的操作数可直接来自于内存。
- MIPS架构有32个32bit寄存器。
- ARM架构有16个32bit寄存器。

- 似乎寄存器越多越好？？？

■ 似乎寄存器越多越好？？？

Design Principle 2: Smaller is faster.

设计原理 2： 少即快

- A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther. 信号传输快
- The designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast.
- “Energy is a major concern today, so another reason for using fewer registers is to conserve energy”.

- 受限于指令中对寄存器编址字段的位数。

2.3 Operands of the Computer Hardware

【例】Compiling a C Assignment Using Registers

- C语言描述：

  f = （g + h） - （i + j）;

  □ 经编译后

- MIPS指令：

```
add $t0, $s1, $s2    # register $t0 contains （g + h）
add $t1, $s3, $s4    # register $t1 contains （ i + j）
sub $s0, $t0, $t1    # f gets $t0 - $t1, which is （g + h）-（i + j）
```

2.3 Operands of the Computer Hardware

Memory Operands

data transfer instruction A command that moves data between memory and registers.

address A value used to delineate the location of a specific data element within a memory array.

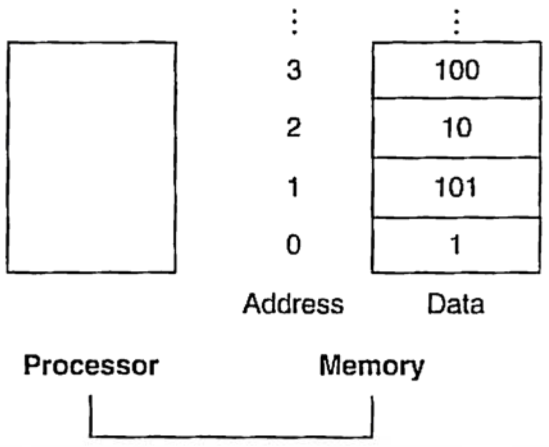


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.

Memory Operands – Load指令（LW）

【例】Compiling an Assignment When an Operand Is in Memory

■ C语言描述：

g = h + A[8]；

□ 经编译后

■ MIPS指令：

```
lw    $t0,  8($s3)        #  Temporary reg $t0 gets A[8]
      # 以 8 +（$s3）、即8加寄存器s3的内容为地址，
      #访问存储器，取到的数放寄存器$t0。
add  $s1, $s2, $t0       #  g = h + A[8]
```

2.3 Operands of the Computer Hardware

Memory Operands -- 字节编址、字访问

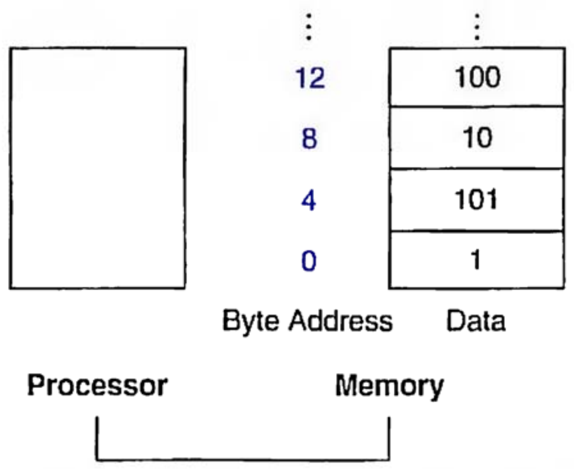


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

Memory Operands -- 字节编址、字访问

Big-Endian or Little-Endian

■如果将一个32位的整数0x12345678 存放到一个整型变量（int ）中，这个整型变量采用大端或者小端模式在内存中的存储由下表所示。为简单起见，本文使用OP3表示一个32位数据的最高字节MSB（Most Significant Byte），使用OP0表示一个32位数据最低字节LSB（Least Significant Byte）。

地址偏移	大端模式	小端模式
0x00	12（OP0）	78（OP3）
0x01	34（OP1）	56（OP2）
0x02	56（OP2）	34（OP1）
0x03	78（OP3）	12（OP0）

Memory Operands -- 字节编址、字访问

Big-Endian or Little-Endian

■【用函数判断系统是Big Endian还是Little Endian】

```
   //如果字节序为big-endian，返回true;
   //反之为  little-endian，返回false
bool  IsBig_Endian()
     {
          unsigned short test = 0x1234 ;
          if (*( (unsigned char *) &test ) ==  0x12 )
               return  TRUE;
          else
               return  FALSE;
     }
```

■ **x86、ARM是小端，MIPS、Power PC是大端。**

Memory Operands – Load指令（LW）

【例】Compiling an Assignment When an Operand Is in Memory

■ C语言描述：

g = h + A[8];

☐ 经编译后

■ MIPS指令：（按32bit字访问）

```
lw    $t0, 32($s3)        # Temporary reg $t0 gets A[8]
                          # 按32bit字访问
add  $s1, $s2, $t0        # g = h + A[8]
```

Memory Operands – Store指令（SW）

【例】Compiling Using Load and Store

■ C语言描述：

A[12] = h + A[8];

□ 经编译后

■ MIPS指令：（按32bit字访问）

```
lw    $t0, 32($s3)      # Temporary reg $t0 gets A[8]
                        # 按32bit字访问
add  $t0, $s2, $t0      # g = h + A[8]

sw   $t0, 48($s3)       # Stores h + A[8] back into A[12]
     # $t0的值，存入48+$s3寄存器值所指的主存单元。
```

2.3 Operands of the Computer Hardware

Constant or Immediate Operands

■ 对于有立即数参与运算的情况，可以采用：

```
lw    $t0, AddrConstant ($s1)   # $t0 = constant 3
add  $s3, $s3, $t0              # $s3 = $s3 + $t0 ($t0 == 3)
```

■ 可以采用在指令中直接包含立即数的指令设计：

```
addi  $s3, $s3, 3         # $s3 = $s3 + 3
```

2.3 Operands of the Computer Hardware

Design Principle 3: Make the common case fast

设计原理3: 加快经常性事件

- Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

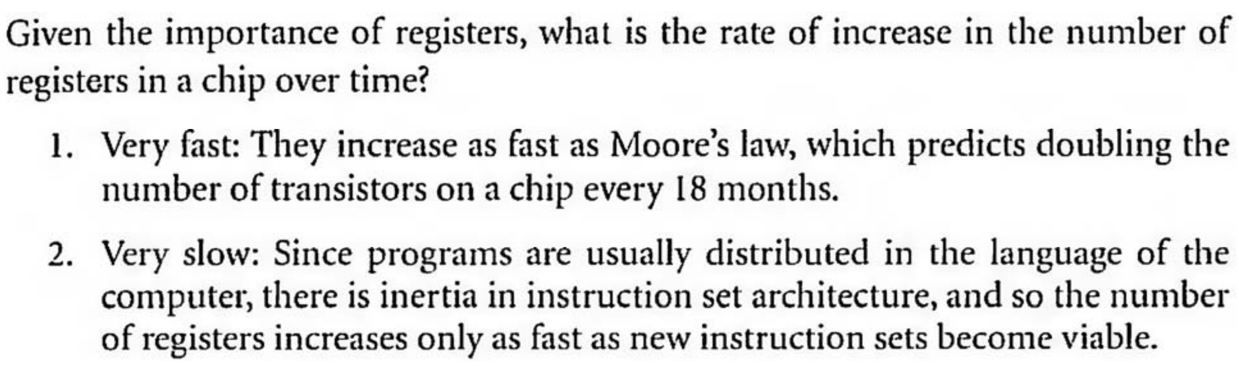
  常数经常被用到。

- The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, the move operation is just an add instruction where one operand is zero. Hence, MIPS dedicates a register $ zero to be hardwired to the value zero. (As you might expect, it is register number 0.)

- MIPS的0号寄存器，通过硬件设置为恒0 。

2.3 Operands of the Computer Hardware

Check Yourself



Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

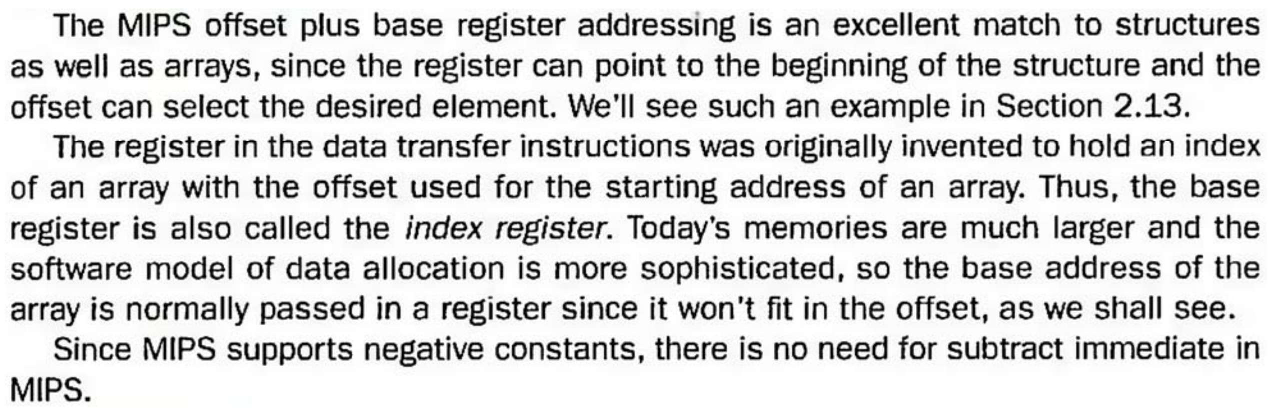
1. Very fast: They increase as fast as Moore's law, which predicts doubling the number of transistors on a chip every 18 months.
2. Very slow: Since programs are usually distributed in the language of the computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable.

2.3 Operands of the Computer Hardware

关于 index register 寻址方式

- 形式：offset[base register/index register]
- 地址 = offset(常数)+ [base/index register]寄存器的数据内容
- 在访问“结构”和“数组”数据结构时，寻址方便。
- 例：8[R2]，(R2)=20080800h,

  则表达的寻址地址为：8+20080800h = 20080808h


The MIPS offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in Section 2.13.

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the *index register*. Today's memories are much larger and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

Since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

2.3 Operands of the Computer Hardware

为什么MIPS的指令操作数不像x86那样，可以直接有内存操作数？

指令设计风格从CISC到RISC的过渡

Intel IA-32指令系统的一些大事记

- 1978年：Intel 8086体系结构是之前已经成功的8位微处理器Intel8080的汇编语言兼容的扩展。8086是16位体系结构，所有内部的寄存器都是16位的，且大都专用。
- 1980年：Intel 8087浮点协处理器问世。该体系结构扩展了8086并多了60条浮点指令。
- 1982年：80286对8086做了扩展，将地址空间增加到24位，并创建了一个详细的内存映射和保护模式，还增加了一些指令去丰富整个指令集以及控制保护模式。
- 1985年：80386扩展80286体系结构的地址空间到32位。它的体系结构除了具有32位的寄存器和32位的地址空间，也增加了一些新的寻址模式和附加的操作。增加的指令使得80386几乎就是通用寄存器型处理器。

设计风格从CISC到RISC的过渡

Intel IA-32指令系统的一些大事记

- 1985~1995：后来1989年的80486，1992的Pentium处理器，以及1995的Pentium Pro都是致力于更高的性能，一共只有4条指令被增加到用户可见的指令集中：其中3个有助于多处理技术和1个条件传送指令。
- 1997：Intel用多媒体扩展MMX（Multi Media Extension）指令集来扩充Pentium和Pentium Pro架构。该新指令集包含57条指令，运用了浮点栈来加速多媒体程序和通信应用程序的运行。MMX通过单指令多数据SIMD（Single Instructions，Multiple Data）方式来一次处理多个短的数据元素。Pentium II没有引入任何新指令。
- 1999：Intel 公司添加了70个指令，将SSE（Streaming SIMD Extension）作为Pentium III的一部分。添加了8个独立的128位寄存器，并增加了一个单精度浮点数据类型。因此，4个32位的浮点操作可以并行进行。为改进内存性能，SSE还包括高速缓存的预取指令，以及可以绕过缓存直接写内存的流存储指令。

设计风格从CISC到RISC的过渡

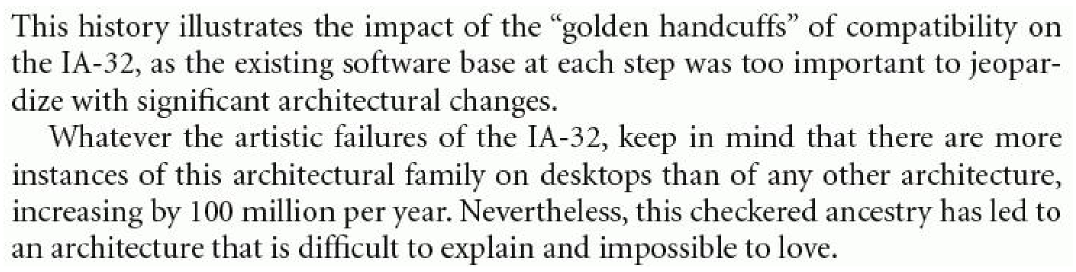
Intel IA-32指令系统的一些大事记

- 2001：Intel公司又增加了另外144条指令，命名为SSE2。新的数据类型是双精度运算，允许并行执行成对的64位浮点型操作。对应于MMX和SSE指令，可并行操作64位数据。它不仅允许更多的多媒体操作，并给编译器提供了更好的浮点寄存器优化支持。从而大大提升了Pentium 4 的浮点型操作性能（Pentium 4是第一个包括SSE2的微处理器）。
- 2003：AMD公司改进了IA-32架构。把地址空间从32位增加到64位，扩展了多种寄存器数目，并增加了指令的长模式（long mode）等。多种增强模式使得AMD64成为一个比HP/Intel的IA-64更加优秀的从32位过渡到64位寻址的处理器。
- 2004：Intel认输，接受了AMD64，重新标记为EM64T（Extended Memory64 Technology），增加了一个128位的比较交换指令，并发布了新一代媒体扩展。SSE3添加了13条指令来支持一些复杂运算，包括在结构数组上进行的图形操作、视频编码、浮点转换以及线程同步。

设计风格从CISC到RISC的过渡

IA-32的这段历史说明了什么？

- 兼容性是一副“金手铐”：已有的软件基础太重要了，以至于在每一步发展的时候，体系结构都不能改变过大而使软件受到危害。
- 无论IA-32有多少不足，这个体系结构家族在桌面计算机上比任何一种其它体系结构都要多，并以每年上亿的速度增长。
- 然而，这个多变的家族带来的是一个难以解释并且不讨人喜欢的体系结构。


This history illustrates the impact of the “golden handcuffs” of compatibility on the IA-32, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the IA-32, keep in mind that there are more instances of this architectural family on desktops than of any other architecture, increasing by 100 million per year. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

- 更多详细内容可参考教材《硬件软件接口》第3版 2.16节等内容。

指令设计风格从CISC到RISC的过渡

在实践中

■ 对于x86系列来说，最常用到的指令是源于80386的32位指令子集，而不是整个的16位和32位指令集。

■ 在IA-32中，有些复杂指令在应用中极少用到。并且有些复杂指令功能的执行，并不比执行一系列独立的简单指令快。

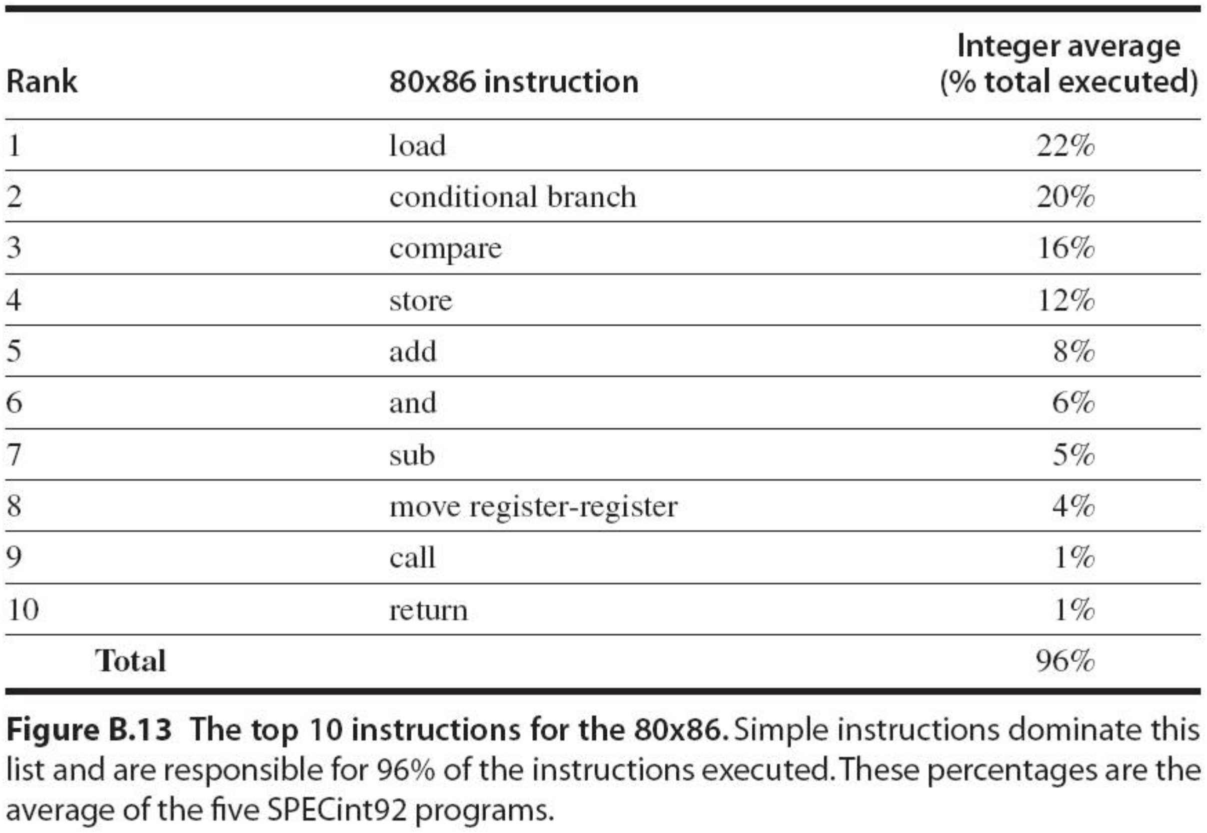
● 如带有重复前缀的串传送指令（REP MOVS），这条指令通常比一个每次移动一个字的循环慢。

● 在近期的Intel IA-32实现中，LOOP指令总是慢于由简单的独立指令组成的代码序列。这样，以提高速度为核心的优化编译器从不生成LOOP指令。造成的结果便是不鼓励在后来的机器中加快LOOP指令的执行，因为它很少被用到。

● 由于历史原因，在设计80286时添加的许多以字节为单位的十进制数运算指令，如今这些指令很少用到，因为使用32位二进制算术运算指令加上与十进制间的转换要快的多。（但从兼容性考虑，在新的处理器中也必须实现这些十进制指令，即使它们很少用到。）

★ 当然，这些变化都是有其发展变化的原因的。

80x86最常用的10条指令



Rank	80x86 instruction	Integer average (% 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%

Figure B.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

设计风格从CISC到RISC的过渡

长期实践表明

- 典型程序中，大约80%的语句仅仅使用处理机中20%的指令。
- 执行频度高的简单指令，因复杂指令的存在，在实现中，难以提高其执行速度。

【问题】：

能否用这20%的简单指令通过组合实现另外的80%的指令功能?

设计风格从CISC到RISC的过渡

RISC（ Reduced Instruction Set Computer ）的主要特征

- 选用使用频度较高的一些简单指令来构成指令集，复杂指令的功能通过由简单指令的组合来实现；
- 指令长度固定、指令格式种类少、寻址方式少；
- 只有LOAD/STORE指令访存；
- CPU中有多个通用寄存器；
- 采用流水技术，一个时钟周期完成成一条指令；
- 采用组合逻辑实现控制器；
- 采用优化的编译程序。

- 值得注意的是，商品化的RISC机通常不会是纯RISC机，故上述这些特点不是所有RISC机全部具备的。

设计风格从CISC到RISC的过渡

一些RISC机器的指令条数

机器名	指令数	机器名	指令数
RISC Ⅱ	39	ACORN	44
MIPS	31	INMOS	111
IBM801	120	IBM RT	118
MIRIS	64	HPPA	140
PYRAMID	128	CLIPPER	101
RIDGE	128	SPARC	89

设计风格从CISC到RISC的过渡

CISC （Complex Instruction Set Computer）的主要特征

- 指令系统复杂庞大，各种指令使用频度相差大；
- 指令长度不固定、指令格式种类多、寻址方式多；
- 访存指令不受限制；
- CPU中多为专用寄存器；
- 大多数指令需要多个时钟周期才能执行完毕；
- 采用微程序控制器实现控制；
- 难以采用优化编译程序生成高效的目标代码。

设计风格从CISC到RISC的过渡

RISC和CISC的比较

- RISC更能充分利用VLSI芯片的面积；
- RISC更能提高计算机运算速度

  指令数少，指令格式少，寻址方式少，通用寄存器多，采用组合逻辑控制，便于实现指令流水
- RISC便于设计，可降低成本，提高可靠性
- RISC有利于编译程序代码优化

- 当然，两种技术是会互相借鉴的。

设计风格从CISC到RISC的过渡

一些CISC与RISC微处理器的特征

特征	CISC			RISC	
	IBM370/168	VAX11/168	Intel 80486	Motorola 88000	MIPS R4000
开发年份	1973	1978	1989	1988	1991
指令数	208	303	235	51	94
指令字长/B	2~6	2~57	1~11	4	32
寻址方式	4	22	11	3	1
通用寄存器数	16	16	8	32	32
控制存储器容量/Kb	420	480	246	0	0
Cache容量/Kb	64	64	8	16	128

设计风格从CISC到RISC的过渡

RISC和CISC的比较

- 多年来，计算机体系结构和组成技术发展的趋势是增加CPU的复杂性，即使用更多的寻址方式及更加专用的寄存器等。RISC的出现象征着与这种趋势的根本决裂，就自然地引起了RISC和CISC的争端。
- 随着技术的不断发展，RISC与CISC还不能说是截然不同的两大体系，很难对它们做出明确的评价。最近几年，RISC与CISC的争端已减少了很多。原因在于这两种技术已逐渐融合。特别是芯片集成度和硬件速度的增大，RISC系统也越来越复杂。
- 与此同时，在努力挖掘最大性能的过程中，CISC的设计已集中到和RISC相关联的主题上，如增加通用寄存器数量以及更加强调指令流水线设计，所以更难去评价它们彼此的优越性了。

设计风格从CISC到RISC的过渡

RISC和CISC的比较

- 80x86是一个与RISC有很大区别且非常成功的系统结构，然而它的成功丝毫没有掩盖掉RISC指令系统的优点。
- 对PC软件二进制兼容的商业重要性，以及在摩尔定律的影响下晶体管容量的不断增加，促使Intel在内部使用RISC指令系统，同时对外支持80x86指令系统。
- 近年来的80x86微处理器，像Pentium 4，其硬件将80x86指令转换成类RISC指令，然后在芯片内执行这些转换后的RISC风格的指令。
- 这样，在程序员和编译器开发者看来，这是80x86的系统结构，而同时基于对性能的追求，计算机设计者可以实现RISC风格的处理器以获取高的性能。

设计风格从CISC到RISC的过渡

部分典型的RISC机器

1975年IBM公司John Cocker提出精简指令系统的设想。1982年加州伯克利大学的研究人员专门研究如何有效利用VLSI的有效空间，并采用RISC思想通过设计更多的寄存器，使其研制的RISC Ⅰ（后来RISC Ⅱ）型微处理的功能超过了当时的VAX-11/780和M68000，其速度比VAX-11/780快了一倍。

同期，美国斯坦福大学RISC研究的课题是MIPS（Micro Processor Without Interlocking Pipeline Stages），即消除流水线各段互锁的微处理器。他们把IBM公司对优化编译程序的研究与伯克利大学对VLSI有效空间利用的思想结合在一起，最终的研究成果转化为MIPS公司RX000的系列产品。

IBM公司又继其IBM801机型、IBM RT/PC后，于1990年推出了著名的IBM RS/6000系列产品。

伯克利大学的研究成果最后发展成为SUN微系统公司的RISC芯片，称为SPARC（Scalable Processor ARChitecture）。

Power PC是IBM、Apple、Motorola三家公司于1991年联合研制的RISC微处理器。

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Geng Wang (SJTU)
  - Yanmin Zhu (SJTU)
  - Yamin Li (Hosei Univ.)