Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets
- 存储程序思想

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

32个寄存器

- 一条指令只能对存放在寄存器中的数据执行算术操作

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	Reserve for assmbler
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	Reserve for Operating
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

指令

算术运算	add	add $s1, $s2, $s3	$s1=$s2+$s3
	subtract	sub $s1, $s2, $s3	$s1=$s2-$s3
	add immediate	addi $s1, $s2, 20	$s1=$s2+20
	……		
数据传送	load word	lw $s1, 20($s2)	$s1=Mem[$s2+20]
	store word	sw $s1, 20($s2)	Mem[$s2+20]=$s1
	load half	lh $s1, 20($s2)	
	load half unsigned	lhu $s1, 20($s2)	
	store half	sh $s1, 20($s2)	
	load byte	lb $s1, 20($s2)	
	load byte unsigned	lbu $s1, 20($s2)	
	store byte	sb $s1, 20($s2)	
	……		

<table>
<tr><td rowspan="6">逻辑运算</td><td>and</td><td>and $s1, $s2, $s3</td><td>$s1=$s2 & $s3</td></tr>
<tr><td>or</td><td>or $s1, $s2, $s3</td><td>$s1=$s2 | $s3</td></tr>
<tr><td>nor</td><td>nor $s1, $s2, $s3</td><td>$s1=~($s2 | $s3)</td></tr>
<tr><td>shift left logical</td><td>sll $s1, $s2, 10</td><td>$s1=$s2<<10</td></tr>
<tr><td>shift right logical</td><td>srl $s1, $s2, 10</td><td>$s1=$s2>>10</td></tr>
<tr><td>……</td><td></td><td></td></tr>
<tr><td rowspan="4">条件分支</td><td>branch on equal</td><td>beq $s1, $s2, 25</td><td>if ($s1==$s2)
go to PC+4+100</td></tr>
<tr><td>branch on not equal</td><td>bne $s1, $s2, 25</td><td>if ($s1!=$s2)
go to PC+4+100</td></tr>
<tr><td>set on less than</td><td>slt $s1, $s2, $s3</td><td>if ($s2<$s3) $s1=1;
else $s1=0</td></tr>
<tr><td>……</td><td></td><td></td></tr>
<tr><td rowspan="4">无条件跳转</td><td>jump</td><td>j 2500</td><td>go to 10000</td></tr>
<tr><td>jump register</td><td>jr $ra</td><td>go to $ra</td></tr>
<tr><td>jump and link</td><td>jal 2500</td><td>$ra=PC+4; go to 10000</td></tr>
<tr><td>……</td><td></td><td></td></tr>
</table>

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

R-type Instruction

- This group contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand
- includes arithmetic and logic with all operands in registers, shift instructions, and register jump instruction (jr)
- All R-type instructions use opcode **000000**.

add $t0, $s1, $s2

1	30	9	8	7	6	5	4	3	2	1	20	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	0
OP:6						Rs:5					Rt:5					Rd:5					Shamt:5					Func:6					
0	0	0	0	0	0																					1	0	0	0	0	0

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

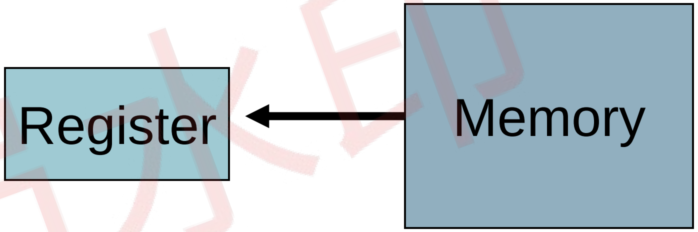
Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4

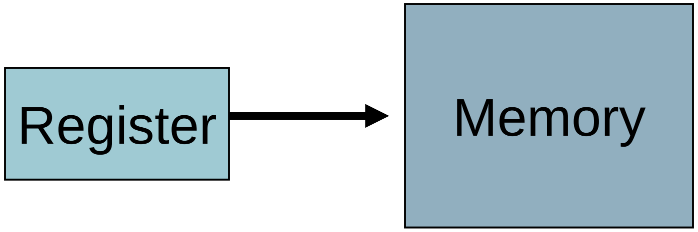
Memory Operands

- Values must be fetched from memory before (e.g. add and sub) instructions can operate on them

Load word
lw $t0, memory-address



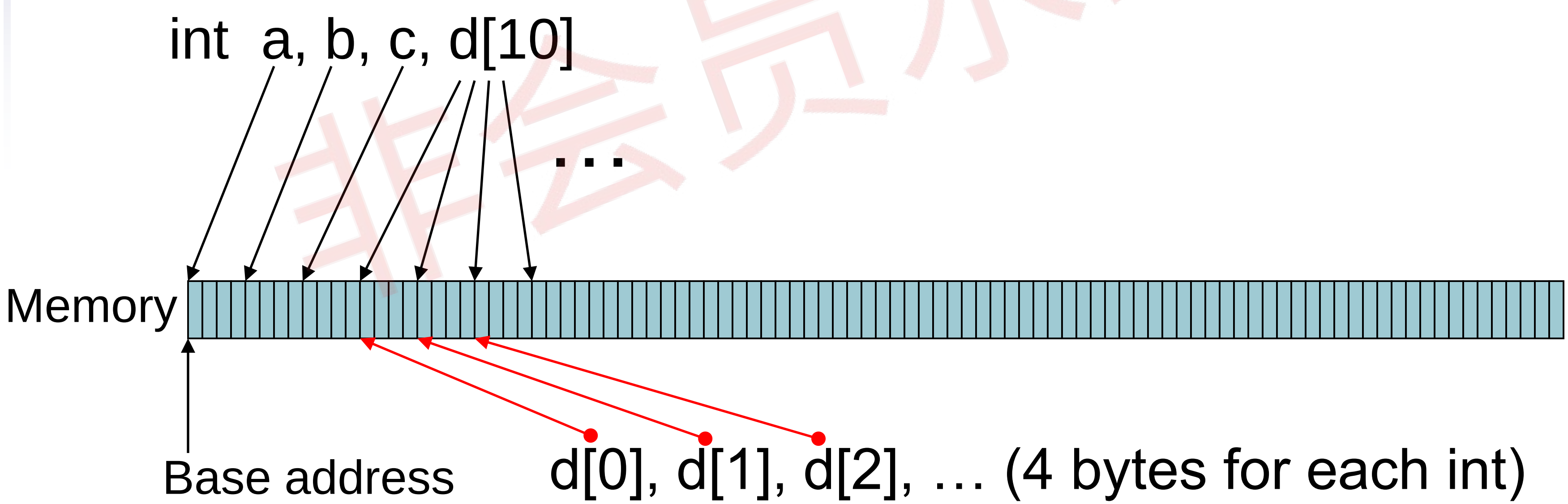
Store word
sw $t0, memory-address



How is memory-address determined?

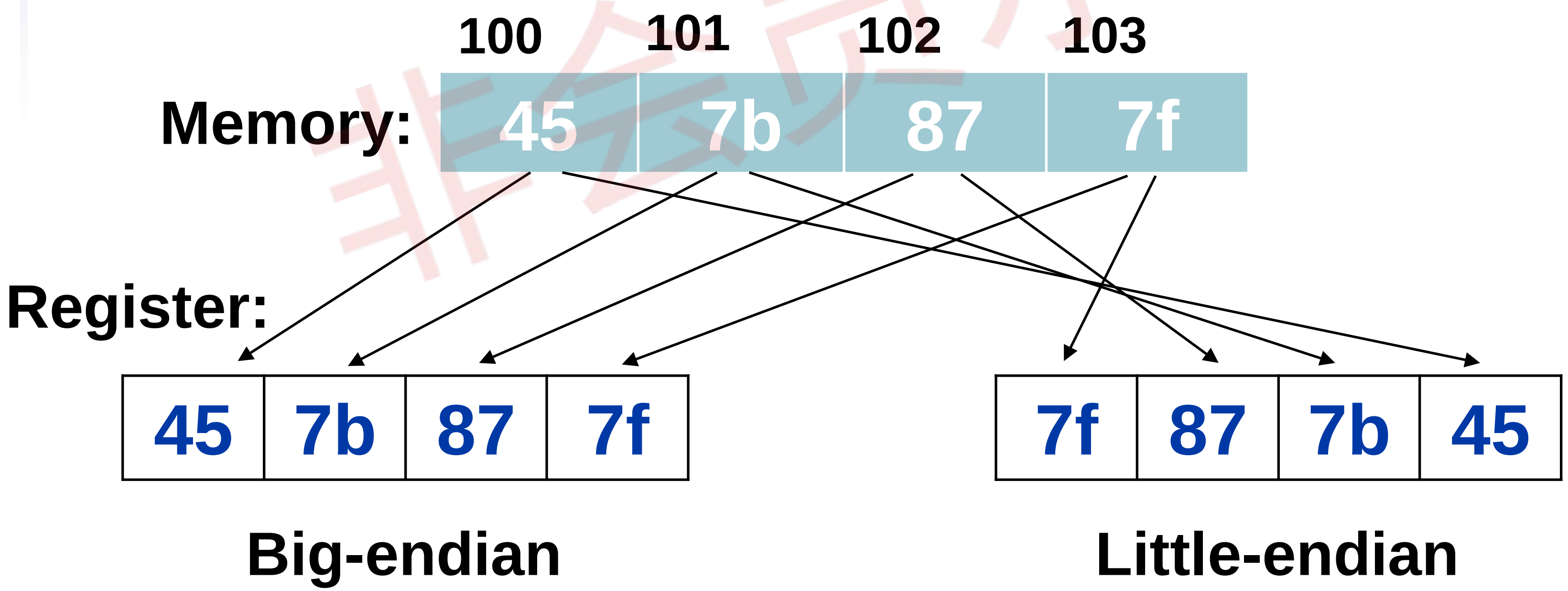
Memory Address

- The compiler organizes data in memory. It knows the location of every variable (saved in a table) and can fill in the appropriate mem-address for load-store instructions(L/S)



Endian-ness

- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset 32 (4 bytes per word)

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```

offset

base register

Instructions

- R-type
  - add, sub, or, and, …
- I-type
  - lw, sw, beq, bne, addi, …
- J-type
  - J, …

I-type Instruction

- The format of a load instruction

destination register　　　　source address

lw　　$t0, 32($s3)

<table>
<tr><td>1</td><td>30</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>20</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="6">OP:6</td><td colspan="5">Rs:5</td><td colspan="5">Rt:5</td><td colspan="16">Offset</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word A[8]
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word A[12]
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- An instruction may require a constant as input
- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

Numeric Representations

- Decimal $35_{10} = 3 \text{ x } 101 + 5 \text{ x } 10^0$
- Binary $00100011_2 = 1 \text{ x } 2^5 + 1 \text{ x } 2^1 + 1 \text{ x } 2^0$
- Hexadecimal (compact representation)

  0x 23 or $23_{16} = 2 \text{ x } 16^1 + 3 \text{ x } 16^0$

- 0-15 (decimal) → 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

Conversions of numbers

- 八进制数转换成二进制数
  $(13.724)_8=(001\ 011.111\ 010\ 100)_2=(1011.1110101)_2$
- 十六进制数转换成二进制数
  $(\ 2B.5E)_{16} = (0010\ 1011.\ 0101\ 1110)_2 = (101011.0101111)_2$
- 二进制数转换成八进制数
  $(0.10101)_2 = (000.101\ 010)_2 = (0.52)_8$
- 二进制数转换成十六进制数
  $(11001.11)_2 = (0001\ 1001.1100)_2 = (19.C)_{16}$

Conversions of numbers

- R进制数 => 十进制数，按“权”展开 (a power of R)

  例: $(10101.01)_2 = 1x2^4+1x\ 2^2+1x2^0+1x2^{-2} = (21.25)_{10}$

  例: $(307.6)_8 = 3x8^2+7x8^0+6x8^{-1} = (199.75)_{10}$

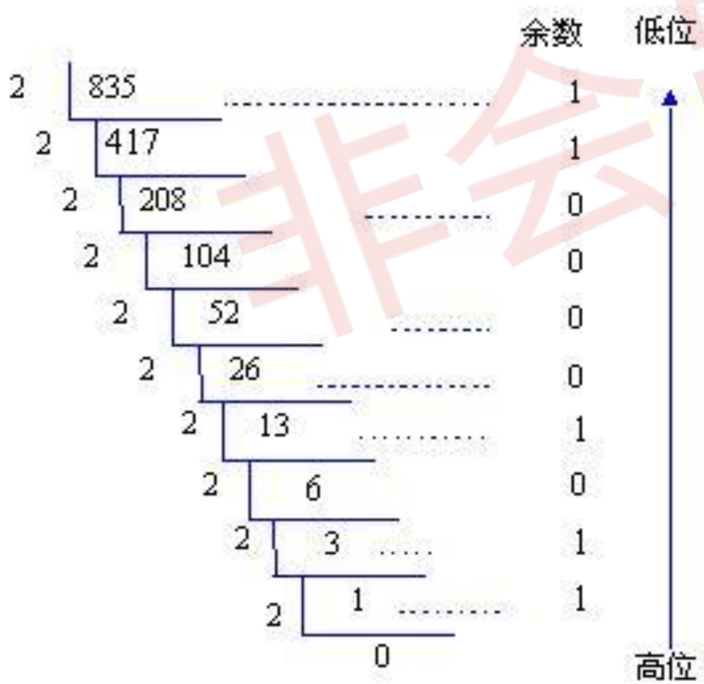
  例: $(3A.1)_{16} = 3x16^1+10x16^0+1x16^{-1} = (58.0625)_{10}$

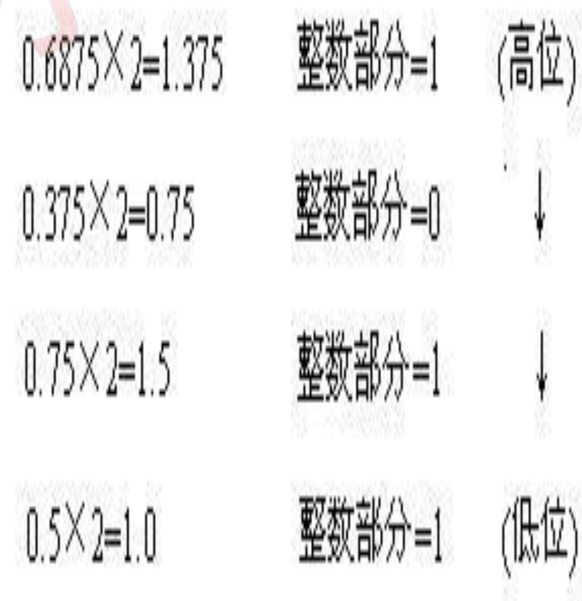
Decimal to Binary Conversions

- 整数部分和小数部分分别转换
  - 整数：“除基取余，上右下左”
  - 小数：“乘基取整，上左下右

有可能乘积的小数部分总得不到0，此时得到一个近似值。

例: $(835.6785)_{10}=(1101000011.1011)_2$





Decimal to Binary Conversions

- 实际按简便方法先转换为二进制数，再按需转换为8/16进制数
  - 整数：2、4、8、16、…、512、1024、2048、4096、…、65536
  - 小数：0.5、0.25、0.125、0.0625、0.03125、……

例：4123.25 = 4096+16+8+2+1+0.25 = $1\ 0000\ 0001\ 1011.01_2$ =$(101B.4)_{16}$

4023=(4096-1)-64-8=$1111\ 1111\ 1111_2$-$100\ 0000_2$-1000_2 =$1111\ 1011\ 0111_2$ = $(FB7)_{16}$

补码特性 - 模运算（modular运算）

在一个模运算系统中，一个数与它除以“模”后的余数等价，如：13 mod 12 等于1，即13点钟等于1点钟

时钟是一种模12系统

假定钟表时针指向10点，要将它拨向6点，

有两种拨法：

① 倒拨4格：10- 4 = 6

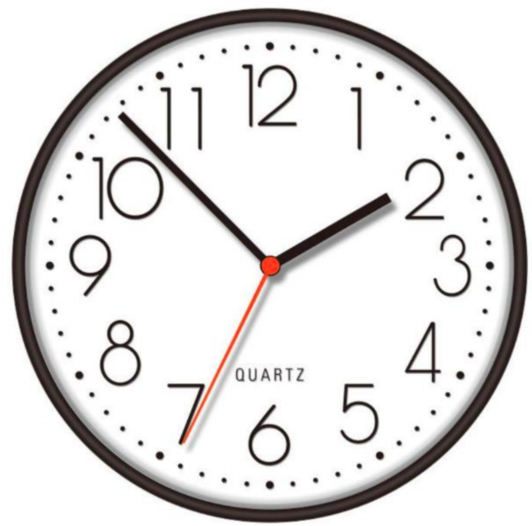
② 顺拨8格：10+8 = 18 ≡ 6 (mod 12)

模12系统中：10- 4 ≡ 10+8 (mod 12)

- 4 ≡ 8 (mod 12)

- 4的模12补码等于8。

同样有 -3 ≡ 9 (mod 12)；-5 ≡ 7 (mod 12) 等



补码特性 - 模运算（modular运算）

结论1：一个负数的补码等于模减该负数的绝对值。

结论2：对于某一确定的模，数x减去小于模的数y，总可以用数x加上−y的补码来代替。

补码（modular运算）：实现 + 和 – 的统一

现实世界的模运算系统举例

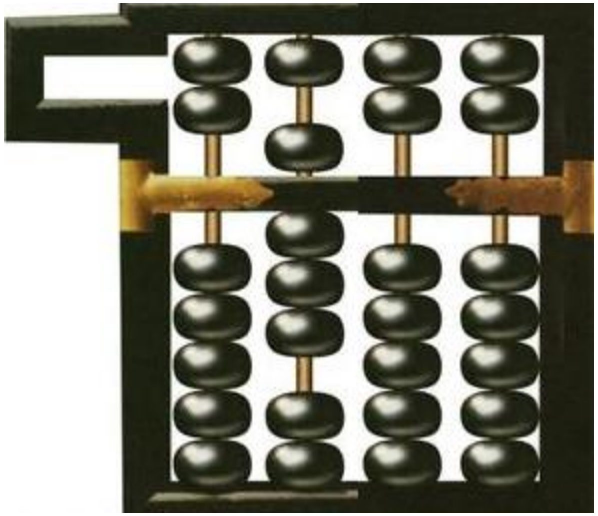
例1：“钟表”模运算系统

假定时针只能顺拨，从10点倒拨4格后是几点？

$10-4 = 10+(12-4) = 10+8 = 6$ （mod 12）

例2：“4位十进制数”模运算系统

假定算盘只有四档，且只能做加法，则在算盘上计算9828-1928等于多少？



$9828-1928 = 9828+(10^4-1928)$

$=9828+8072$

$=\boxed{1}7900$

$=7900$ （mod 10^4）

取模即只留余数，高位“1”被丢弃！相当于只有低4位留在算盘上。

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- 数x的相反数-x的二进制补码是2^n-x
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Sign and Magnitude（原码）

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	Binary
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

Sign and Magnitude（原码）

- 采用符号和幅值表示，容易理解
- 缺点
  - 0 的表示不唯一，故不利于程序员编程
  - 加、减运算方式不统一
  - 需额外对符号位进行处理，故不利于硬件设计
  - 特别当 a<b时，实现 a-b比较困难
- 现在计算机整数都采用补码来表示，但浮点数的尾数用原码定点小数表示

反码和移码

- 反码：一个数的相反数就是将这个数的每一位按位取反，0变成1,1变成0, x的相反数是2^n-x-1。使用$10...000_2$表示最小负数，$01...11_2$表示最大正数。正数和负数数量相同，但保留两个0，一个正零（$00...00_2$），一个负零（$11...11_2$）。当采用反码时，加法器需要一个额外的步骤，即减去一个数来修正结果。
- 移码：通过将数加一个偏移量使其具有非负的表示形式。最小的负数用$00...000_2$表示，最大的正数用$11...11_2$表示，0一般用$10...00_2$表示

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

add	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

I型指令

- `lw  $t0, 32($s3)     # load word A[8]`

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits
35	19	8	32

I-Type

- This group includes instructions with an immediate operand
  - branch instructions
  - load and store instructions
- All opcodes except 000000, 00001x, and 0100xx are used for I-type instructions.

指令编码

指令	类型	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{10}	
sub	R	0	reg	reg	reg	0	34_{10}	
addi	I	8_{10}	reg	reg				常数
lw (load word)	I	35_{10}	reg	reg				地址
sw (store word)	I	43_{10}	reg	reg				地址

Machine Language Example

- C code: `A[12] = h + A[8];`
  - h in $s2, base address of A in $s3
- Compiled MIPS code:

```
lw  $t0, 32($s3)    # load word A[8]
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word A[12]
```

op	rs	rt	rd	address/ shamt	funct
35	19	8	32		
0	18	8	8	0	32
43	19	8	48		

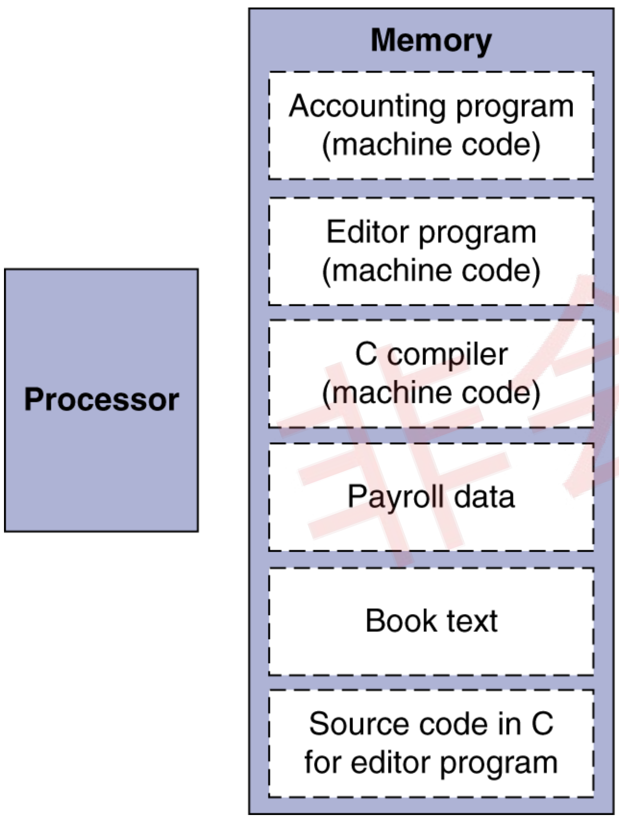
对应的二进制机器指令

op	rs	rt	rd	address/ shamt	funct
100011	10011	01000	0000000000100000		
000000	10010	01000	01000	00000	100000
101011	10011	01000	0000000000110000		

机器语言

名称	类型							注释
add	R	0	18	19	17	0	32	add $1, $2, $3
sub	R	0	18	19	17	0	34	sub $1, $2, $3
addi	I	8	18	17	100			addi $1, $2, 100
lw	I	35	18	17	100			lw $s1, 100($2)
sw	I	43	18	17	100			sw $s1, 100($2)
位数		6	5	5	5	5	5	
R型		op	rs	rt	rd	shamt	funct	
I型		op	rs	rt	address			

Stored Program Computers



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOR	~, |	~, |	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

立即数的扩展

- 在与立即数进行逻辑操作时，立即数的高16位补0后形成32位常数进行计算
- 而与立即数做加法运算时，将立即数进行符号扩展

Making Decision

- Based on the input data and the value created during computation, different instructions execute.
- Conditional branches
  - BEQ, BNE
  - SLT
  - …
- Unconditional branch
  - J
  - JR, JAL

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
  - PC=PC+4+(Label<<2)
  - PC relative addressing
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;

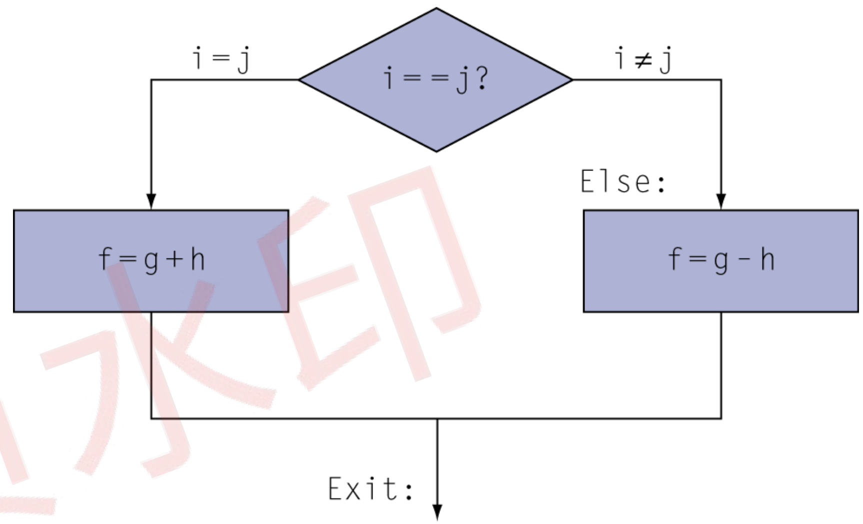
Unconditional Operations

- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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



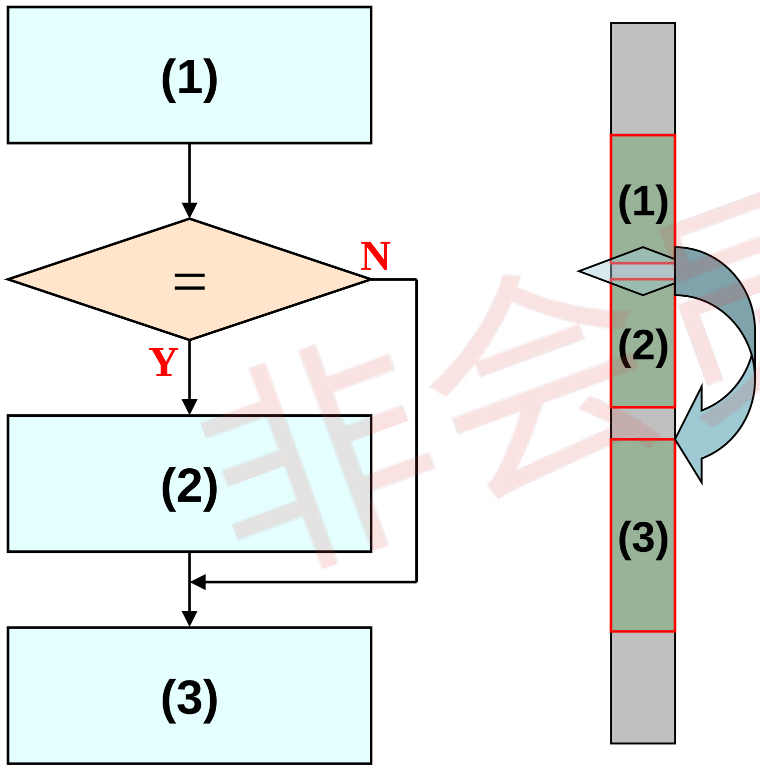
  - f, g, … in $s0, $s1, …

- Compiled MIPS code:

Assembler calculates addresses

```
      bne  $s3, $s4, Else
      add  $s0, $s1, $s2
      j    Exit
Else: sub  $s0, $s1, $s2
Exit: …
```

Conditional branch



(1) bne $s3, $s4, Exit
(2) add $s0, $s1, $s2
Exit: (3)

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Jump: J-type Instruction

- J label

1	30	9	8	7	6	5	4	3	2	1	20	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	0
OP:6						Addr:26																									

- Execute:
  - PC = label
    - Direct addressing. but ***impossible***, why?
  - PC = ((PC+4) & 0xF000_0000) | (label << 2)
    - Pseudodirect addressing
  - PC: Program Count
    - The register that always holds the address of the current instruction being executed.

J-type

- J-Type - This group consists of the two direct jump instructions, j and jal (Jump and Link). These instructions require a memory address to specify their operand.
- J-type instructions use opcodes 00001x.
- Jal L1
  - 1. $ra = PC+4;
  - 2. go to L1;

SLT

- SLT \$rd, \$r1, \$r2
  - If (\$r1 < \$r2) \$rd = 1; else \$rd = 0;

slt   \$t0, \$s1, \$s2

1	30	9	8	7	6	5	4	3	2	1	20	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	0
OP:6						Rs:5					Rt:5					Rd:5					Shamt:5					Func:6					
0	0	0	0	0	0																					1	0	1	0	1	0

<、>、<=、>=

- If ($s0 < $s1) goto L1

```
Slt  $t0, $s0, $s1
Bne  $t0, $zero, L1
```

- if ($s0 > $s1) goto L1

```
Slt  $t0, $s1, $s0
Bne  $t0, $zero, L1
```

- if ($s0 >= $s1) goto L1

```
Slt  $t0, $s0, $s1
Beq  $t0, $zero, L1
```

- If ($s0 <= $s1) goto L1

```
Slt  $t0, $s1, $s0
Beq  $t0, $zero, L1
```

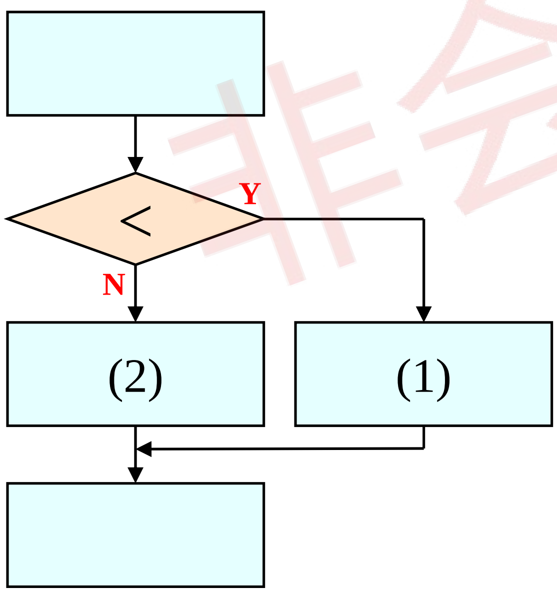
Control Flow

- if ($s1 < $s2) then

  …(1)

  else

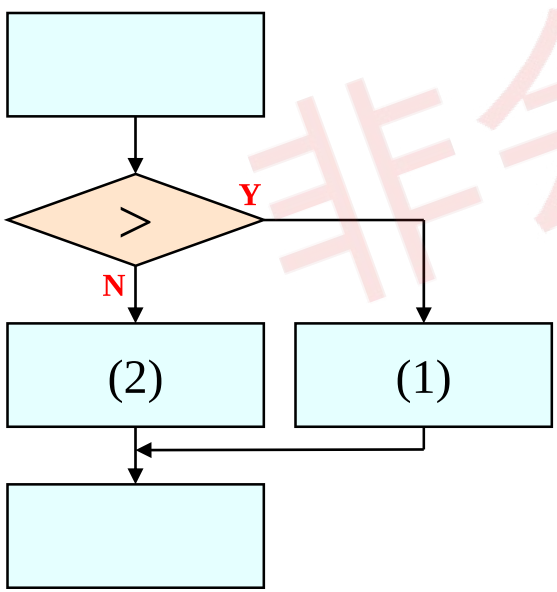
  …(2)




```
        Slt   $t0, $s1, $s2
        bne   $t0, $zero, (1)
              …(2)
        j     exit
(1)           …(1)
Exit:
        ……
```

Control Flow

- if ($s1 > $s2) then
  …(1)
  else
  …(2)




```
          Slt    $t0, $s2, $s1
          bne    $t0, $zero, (1)
                 …(2)
          j      exit
(1)              …(1)
Exit:
          ……
```

SLT

Pseudo instruction

- SLT $rd, $r1, $r2
  - if($rs<$rt)$rd=1; else $rd=0;
- if($r1 < $r2)goto lable
  - Blt $r1, $r2, label

SLT $at, $r1, $r2
Bne $at, $zero, label

- if($r1 > $r2)goto lable
  - Bgt $r1, $r2, label

SLT $at, $r2, $r1
Bne $at, $zero, label

- if($r1<=$r2)goto lable
  - Ble $r1, $r2, label

SLT $at, $r2, $r1
Beq $at, $zero, label

- if($r1>=$r2)goto lable
  - Bge $r1, $r2, label

SLT $at, $r1, $r2
Beq $at, $zero, label

Pseudo instruction

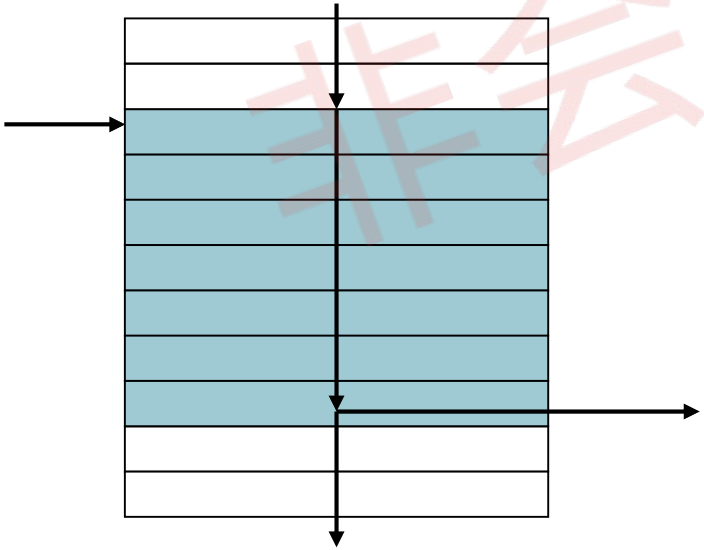
- These instructions need not be implemented in hardware; however, their appearance in assembly language simplifies translation and programming.
  - When considering performance you should count real instructions.
- e.g.
  - Move $s1, $s2   # $s1=$s2

    Add $s1, $s2, $zero
  - Beqz $s1, L1

    Beq $r, $zero, L1

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - \$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - \$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - $-1 < +1 \Rightarrow$ \$t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow$ \$t0 = 0

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

边界检查的简便方法

- 将有符号数作为无符号数来处理，是检验$0 \le x < y$的一种低开销方法，常用于检查数组的下标是否越界
- 使用无符号比较$x < y$，在检查x是否小于y的同时，也检查了x是不是一个负数

- Example:
  Sltu \$t0, \$s1, \$t2　　　　#\$t0=0 if \$s1>= \$t2 or \$s1<0

寄存器跳转

- JR $s1
  - PC = $s1

Exercise

- Assemble language
  ```
                ADD   $S2, $T8, $T0
                LW    $S0, $S1(-123)
                SW    $RA, $SP(123)
  For:          BEQ   $T0, $T1, For
  ```
- Machine language

Exercise

- Assemble language
  - `            ADD   $S2, $T8, $T0`
  - `            LW    $S0, $S1(-123)`
  - `            SW    $RA, $SP(123)`
  - `For:        BEQ   $T0, $T1, For`
- Machine Language
  - `02488824`
  - `8E30FF85`
  - `AFBF007B`
  - `1109FFFF`

Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
  - Since jal may overwrite thevalue in $ra, it must be saved somewhere before invoking the jal instruction

- Procedure return: jump register

  `jr $ra`

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Register Usage

- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$t0 – \$t9: temporaries
  - Can be overwritten by callee
- \$s0 – \$s7: saved
  - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure Example

- MIPS code:

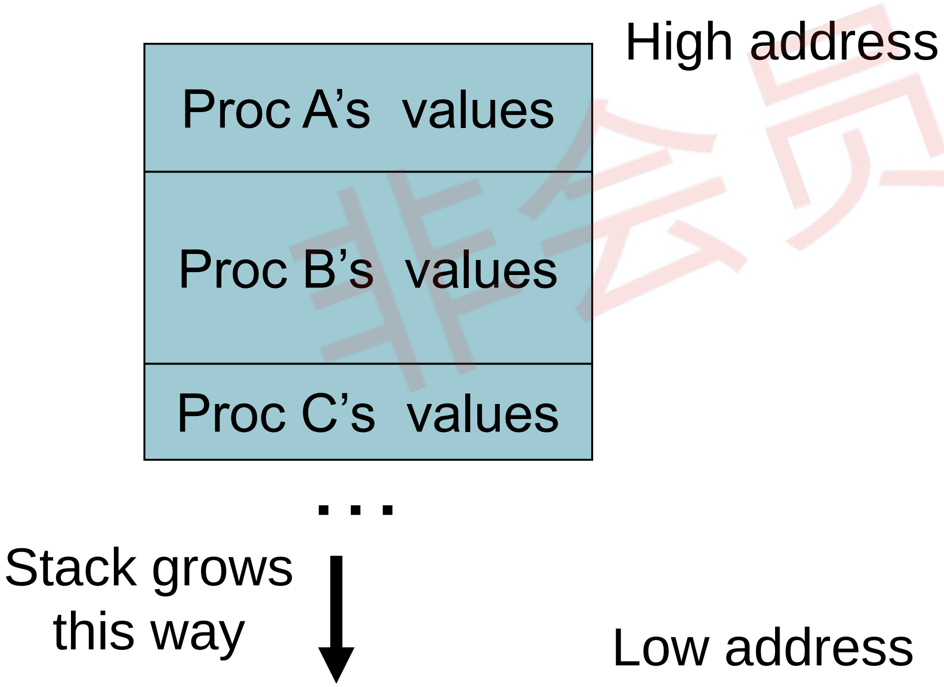
MIPS code	
`leaf_example:`	
`addi $sp, $sp, -4` `sw   $s0, 0($sp)`	Save $s0 on stack
`add  $t0, $a0, $a1` `add  $t1, $a2, $a3` `sub  $s0, $t0, $t1`	Procedure body
`add  $v0, $s0, $zero`	Result
`lw   $s0, 0($sp)` `addi $sp, $sp, 4`	Restore $s0
`jr   $ra`	Return

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

The Stack

- The register for a procedure seems volatile – it seems to disappear every time we switch procedures
- a procedure's values are therefore backed up in memory on a stack




```
Proc A

        call Proc B
            …
            call Proc C
              …
            return
        return
return
```

Storage Management on a Call/Return

- A new procedure must create space for all its variables on the stack
- Before executing the jal, the caller must save relevant values in \$t0-\$t9, \$a0-\$a3, \$ra into its own stack space
- Arguments are copied into \$a0-\$a3; the jal is executed
- After the callee creates stack space, it updates the value of \$sp
- Once the callee finishes, it copies the return value into \$v0 and \$v0, frees up stack space, and \$sp is incremented
- On return, the caller may bring in its stack values into registers

Saves on Stack

- Caller saved
  - $a0-a3 -- old arguments must be saved before setting new arguments for the callee
  - $ra -- must be saved before the jal instruction over-writes this value
  - $t0-t9 -- if you plan to use your temps after the return, save them. Note that callees are free to use temps as they please
  - You need not save $s0-s7 as the callee will take care of them

Saves on Stack

- Callee saved
  - \$s0-s7 -- before the callee uses such a register, it must save the old contents since the caller will usually need it on return
  - local variables -- space is also created on the stack for variables local to that procedure

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

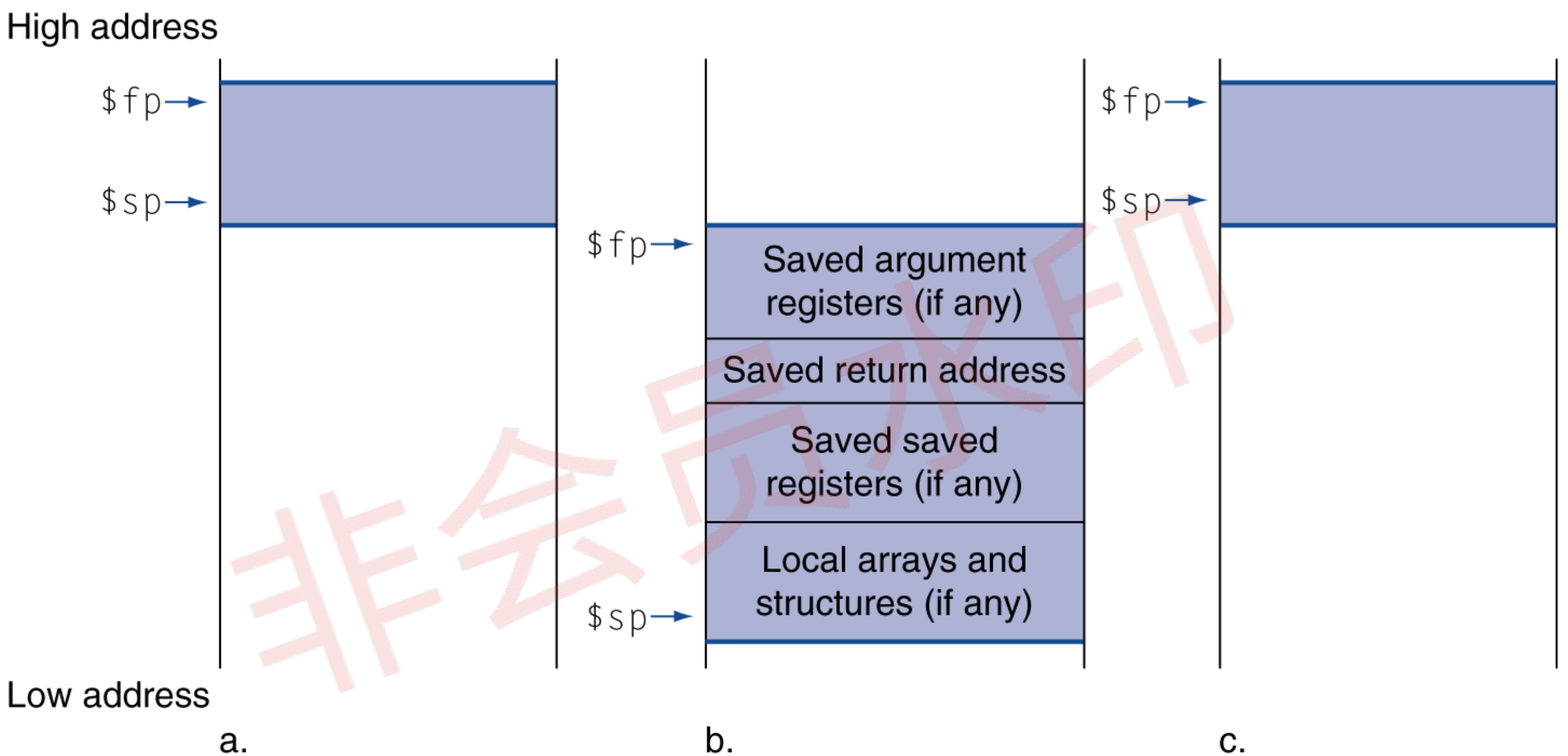
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code:

```
fact:
     addi $sp, $sp, -8      # adjust stack for 2 items
     sw   $ra, 4($sp)       # save return address
     sw   $a0, 0($sp)       # save argument
     slti $t0, $a0, 1       # test for n < 1
     beq  $t0, $zero, L1
     addi $v0, $zero, 1     # if so, result is 1
     addi $sp, $sp, 8       #   pop 2 items from stack
     jr   $ra               #   and return
L1:  addi $a0, $a0, -1      # else decrement n
     jal  fact              # recursive call
     lw   $a0, 0($sp)       # restore original n
     lw   $ra, 4($sp)       #   and return address
     addi $sp, $sp, 8       # pop 2 items from stack
     mul  $v0, $a0, $v0     # multiply to get result
     jr   $ra               # and return
```

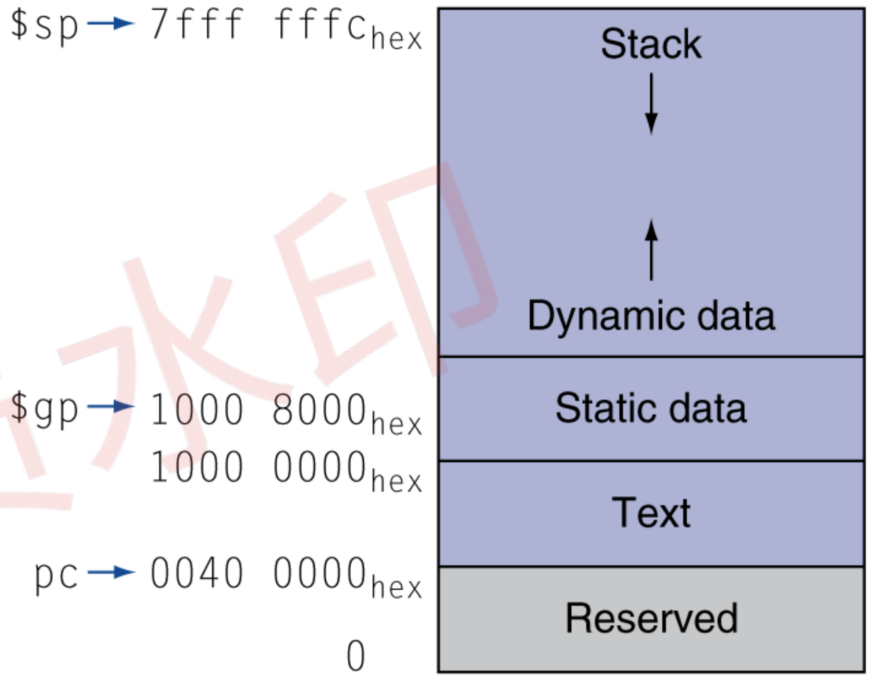
Local Data on the Stack



- Local data allocated by callee, e.g., C variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack:



Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure)
- frame pointer points to the start of the record and stack pointer points to the end
- variable addresses are specified relative to \$fp as \$sp may change during the execution of the procedure
- \$gp points to area that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-16, UTF-32
  - UTF-8: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)     lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)    lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)     sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

```
strcpy:
     addi $sp, $sp, -4      # adjust stack for 1 item
     sw   $s0, 0($sp)       # save $s0
     add  $s0, $zero, $zero # i = 0
L1:  add  $t1, $s0, $a1     # addr of y[i] in $t1
     lbu  $t2, 0($t1)       # $t2 = y[i]
     add  $t3, $s0, $a0     # addr of x[i] in $t3
     sb   $t2, 0($t3)       # x[i] = y[i]
     beq  $t2, $zero, L2    # exit loop if y[i] == 0
     addi $s0, $s0, 1       # i = i + 1
     j    L1                # next iteration of loop
L2:  lw   $s0, 0($sp)       # restore saved $s0
     addi $sp, $sp, 4       # pop 1 item from stack
     jr   $ra               # and return
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

Large Constants

- Immediate instructions can only specify 16-bit constants
- The lui instruction is used to store a 16-bit constant into the upper 16 bits of a register. thus, two immediate instructions are used to specify a 32-bit constant
- The destination address in a conditional branch is specified as a 16-bit constant, relative to the current PC
- A jump (j) instruction can specify a 26-bit constant; if more bits are required, the jump-register (jr) instruction is used

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo) Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

`Loop: sll $t1, $s3, 2`		80000	0	0	19	9	4	0
`add $t1, $t1, $s6`		80004	0	9	22	9	0	32
`lw $t0, 0($t1)`		80008	35	9	8	0		
`bne $t0, $s5, Exit`		80012	5	8	21	2		
`addi $s3, $s3, 1`		80016	8	19	19	1		
`j Loop`		80020	2	20000				
`Exit: …`		80024						

Branching Far Away

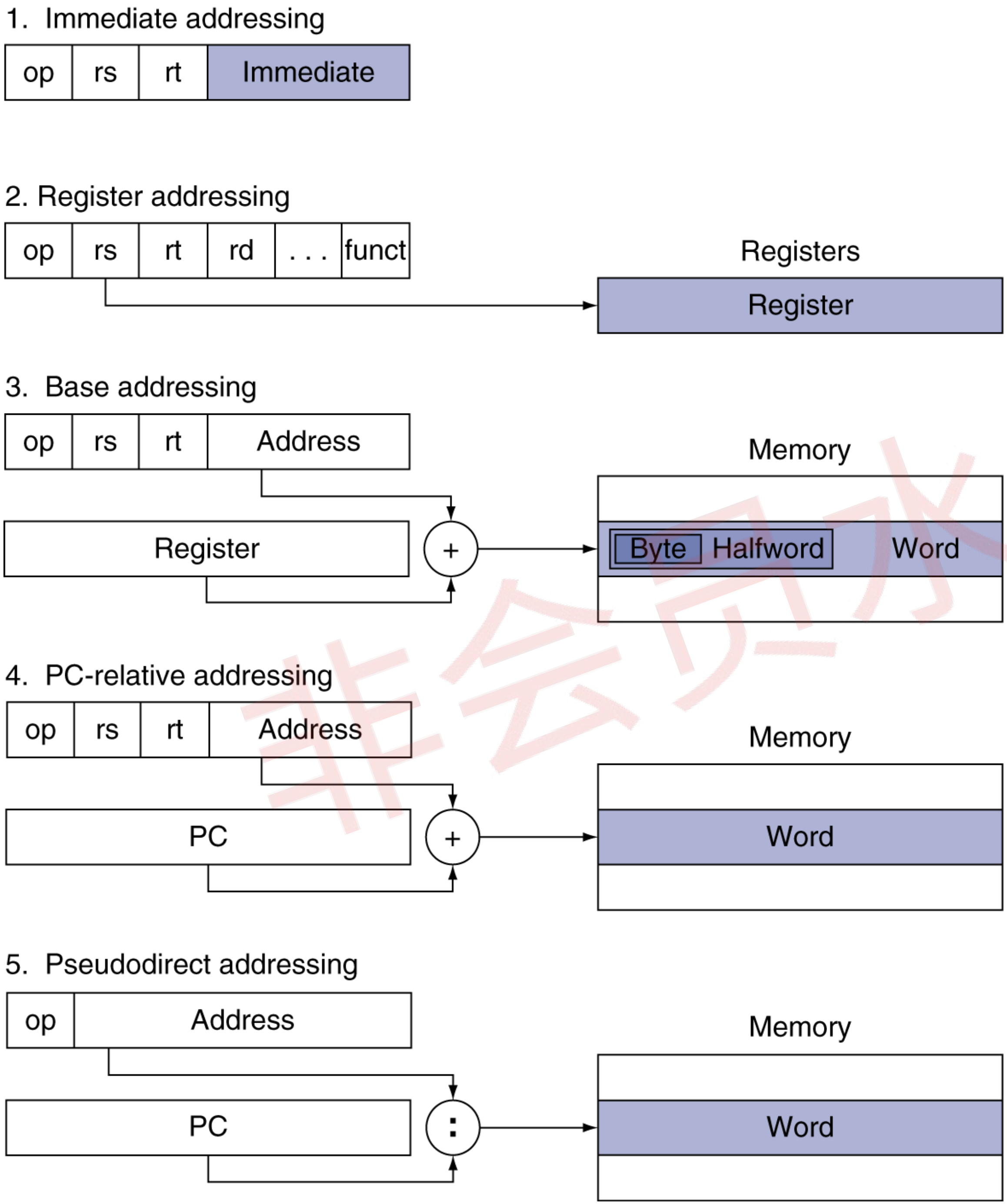
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1

              ↓

        bne $s0,$s1, L2
        j L1
    L2: …
```

Addressing Mode Summary



机器语言解码 Example

- 下面这条机器指令对应的汇编语句是什么
  00af8020hex
- 先转换为二进制
- 0000 0000 1010 1111 1000 0000 0010 0000

	op	rs	rt	rd	address/ shamt	funct
R类型	000000	00101	01111	10000	00000	100000
I类型						
J类型						

- add $s0, $a1, $t7

Synchronization

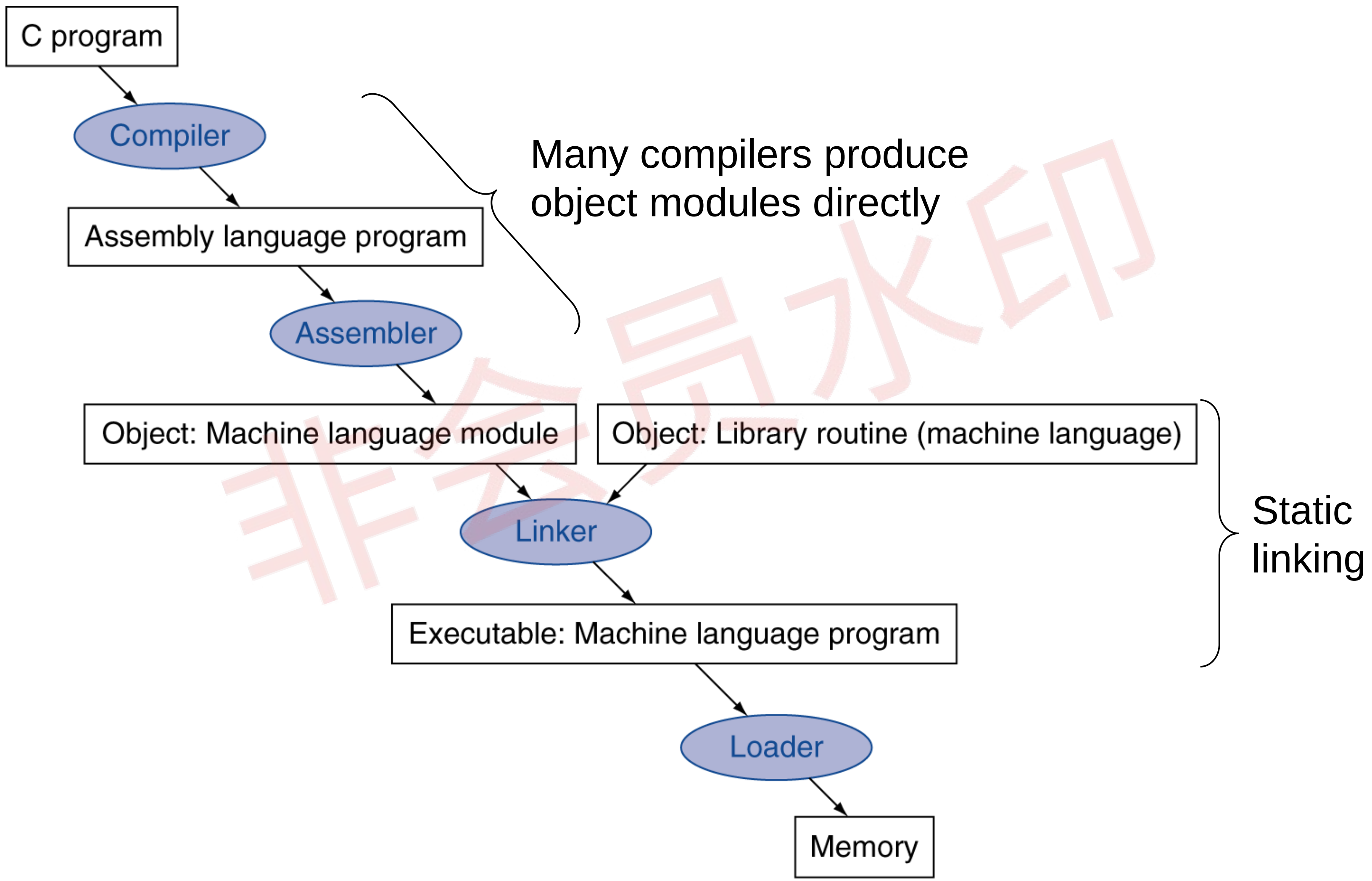
- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

Translation and Startup



Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1     →  add $t0, $zero, $t1
blt $t0, $t1, L   →  slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

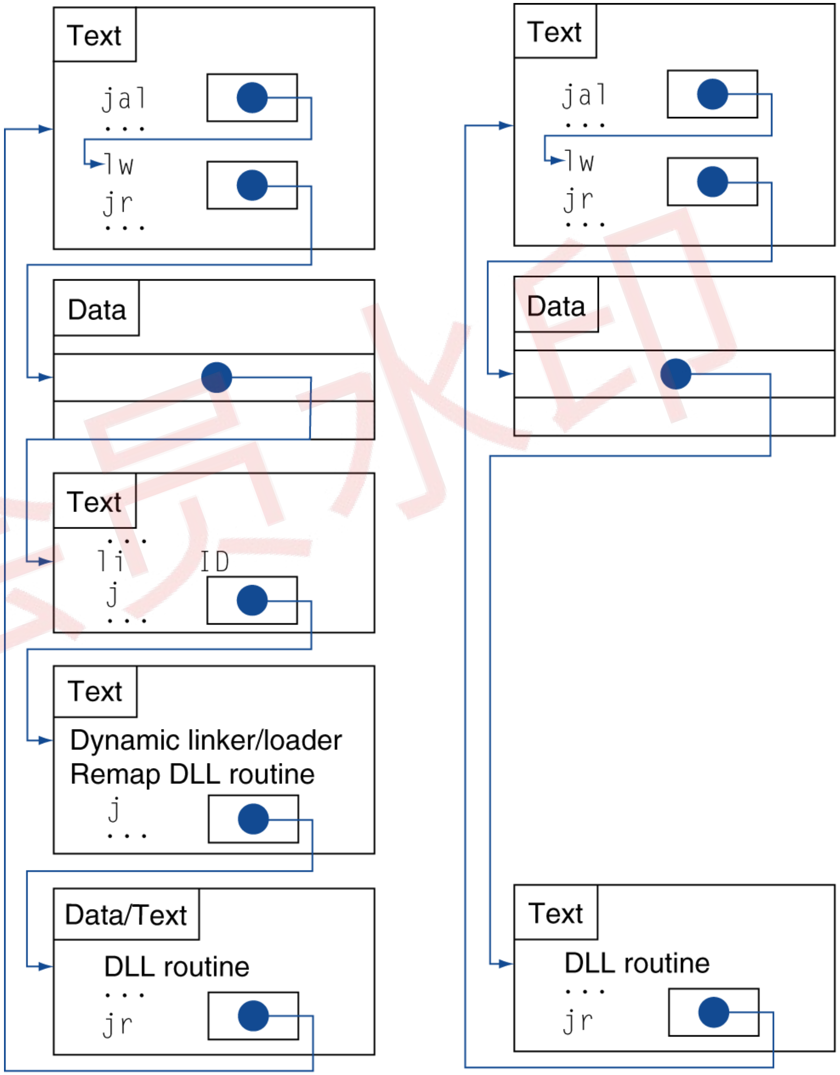
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

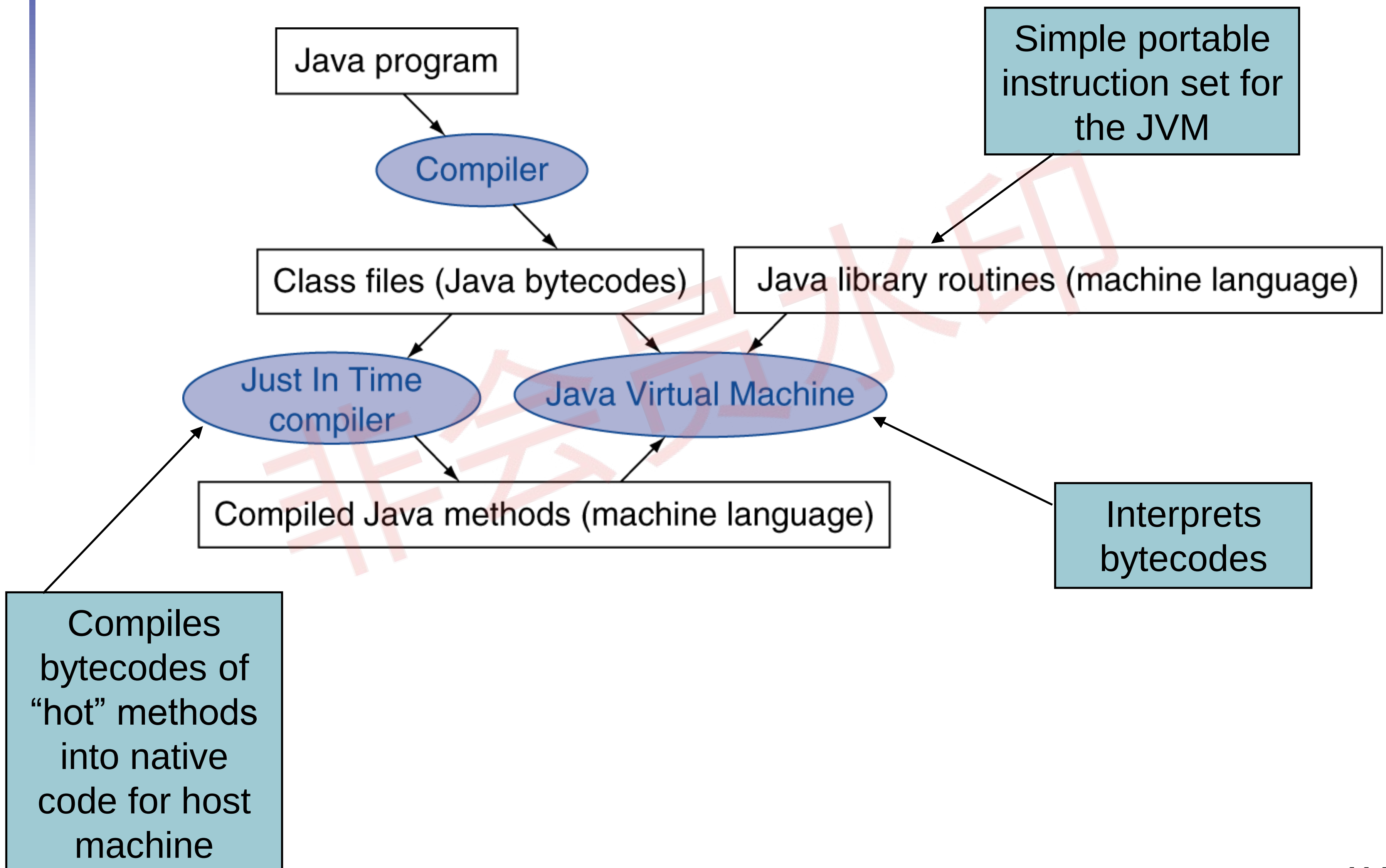
Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting Java Applications



C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
         move $s2, $a0            # save $a0 into $s2
         move $s3, $a1            # save $a1 into $s3
```
Move params

```
         move $s0, $zero          # i = 0
for1tst: slt  $t0, $s0, $s3       # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
```
Outer loop

```
         beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1        # j = i – 1
for2tst: slti $t0, $s1, 0         # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2   # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2         # $t1 = j * 4
         add  $t2, $s2, $t1       # $t2 = v + (j * 4)
         lw   $t3, 0($t2)         # $t3 = v[j]
         lw   $t4, 4($t2)         # $t4 = v[j + 1]
         slt  $t0, $t4, $t3       # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
```
Inner loop

```
         move $a0, $s2            # 1st param of swap is v (old $a0)
         move $a1, $s1            # 2nd param of swap is j
         jal  swap                # call swap procedure
```
Pass params & call

```
         addi $s1, $s1, –1        # j –= 1
         j    for2tst             # jump to test of inner loop
```
Inner loop

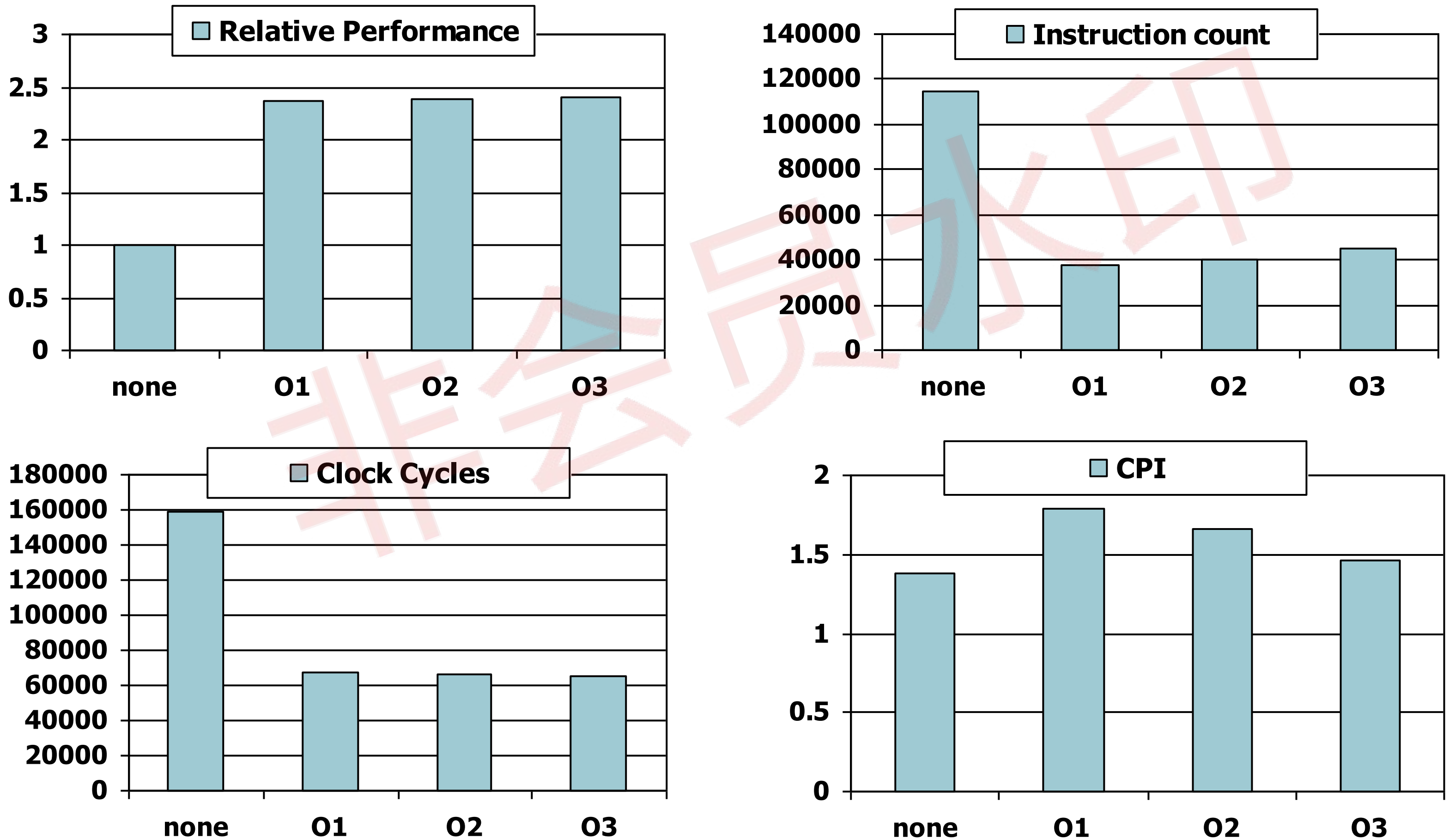
```
exit2:   addi $s0, $s0, 1         # i += 1
         j    for1tst             # jump to test of outer loop
```
Outer loop

The Full Procedure

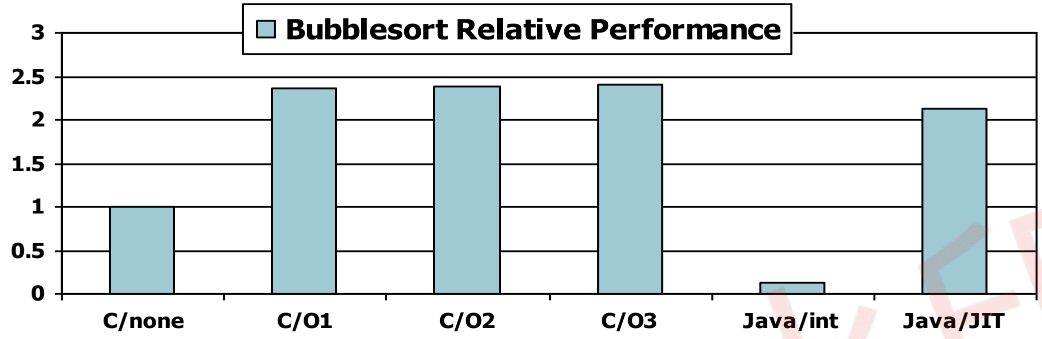
```
sort:   addi $sp,$sp, -20        # make room on stack for 5 registers
        sw $ra, 16($sp)          # save $ra on stack
        sw $s3,12($sp)           # save $s3 on stack
        sw $s2, 8($sp)           # save $s2 on stack
        sw $s1, 4($sp)           # save $s1 on stack
        sw $s0, 0($sp)           # save $s0 on stack
        …                        # procedure body
        …
        exit1: lw $s0, 0($sp)    # restore $s0 from stack
        lw $s1, 4($sp)           # restore $s1 from stack
        lw $s2, 8($sp)           # restore $s2 from stack
        lw $s3,12($sp)           # restore $s3 from stack
        lw $ra,16($sp)           # restore $ra from stack
        addi $sp,$sp, 20         # restore stack pointer
        jr $ra                   # return to calling routine
```

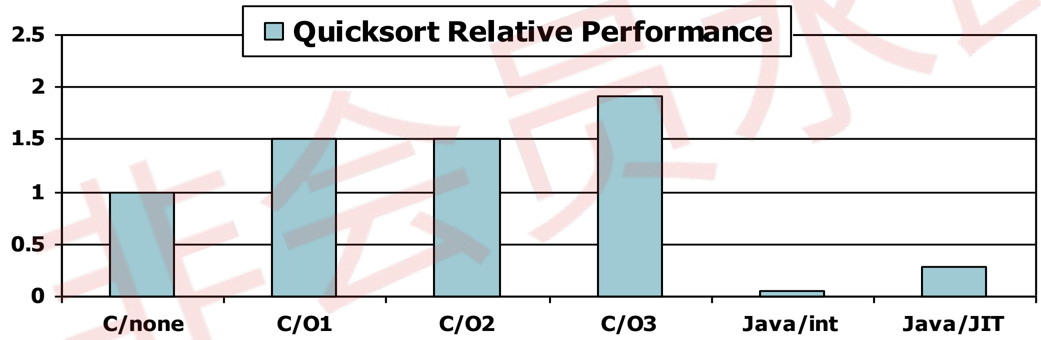
Effect of Compiler Optimization

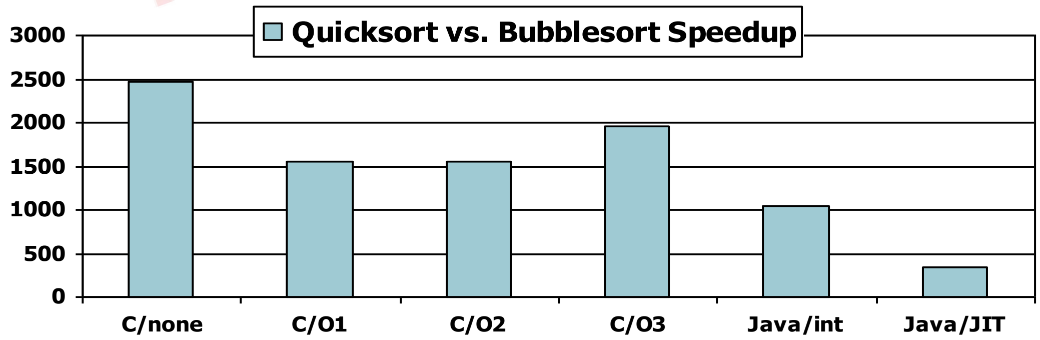
Compiled with gcc for Pentium 4 under Linux



Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

```
       move $t0,$a0      # p = & array[0]
       sll $t1,$a1,2     # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                       #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4    # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                       #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

ARM & MIPS Similarities

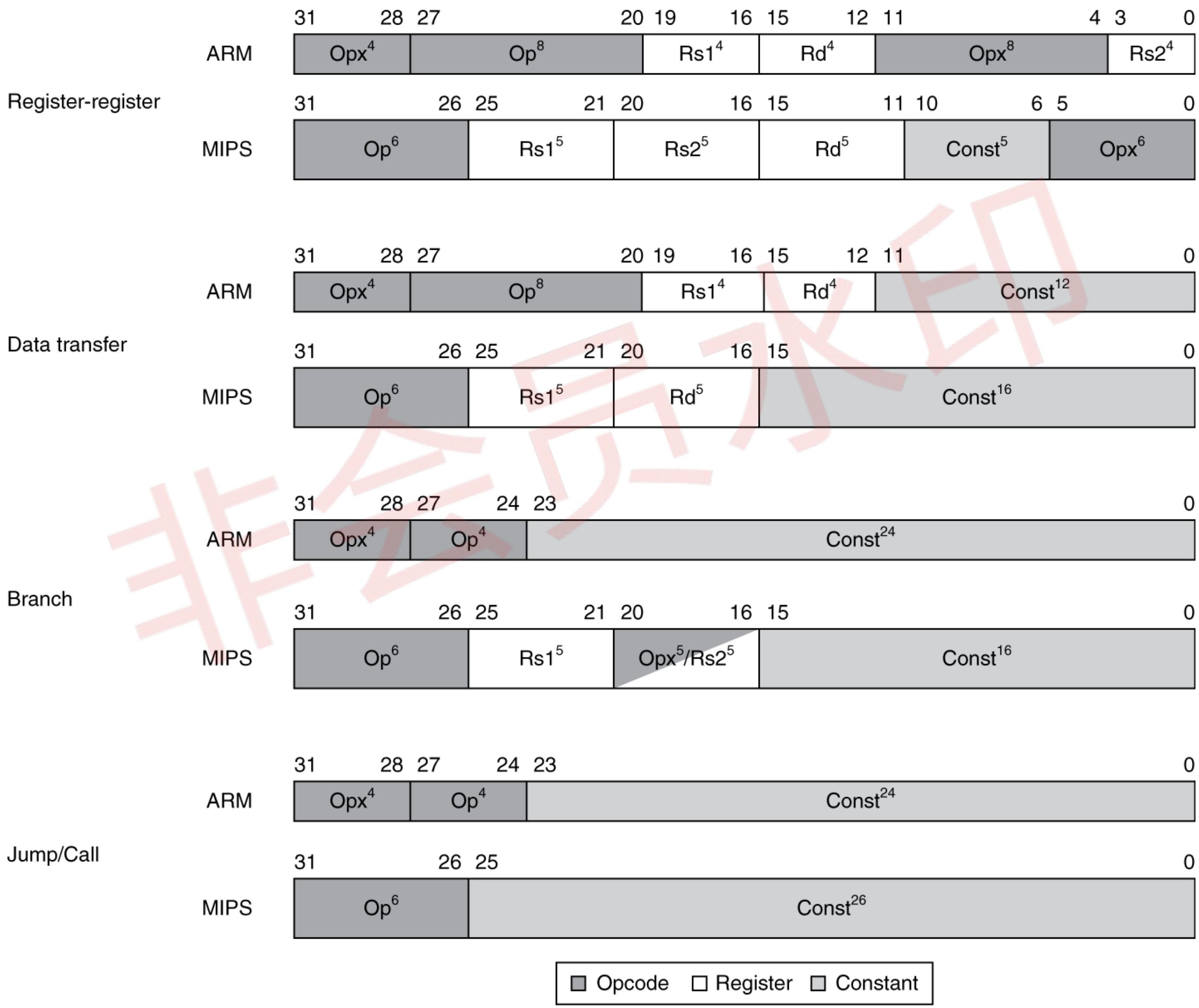
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Encoding



ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers

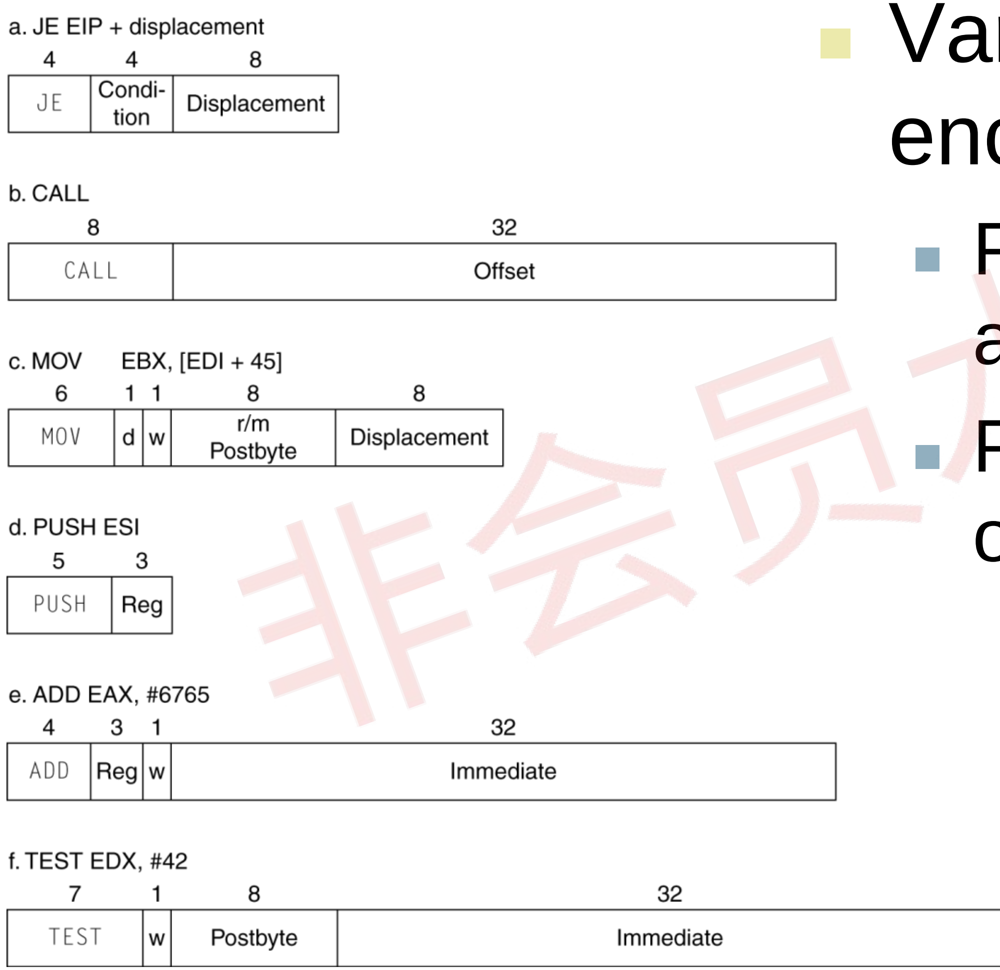
Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

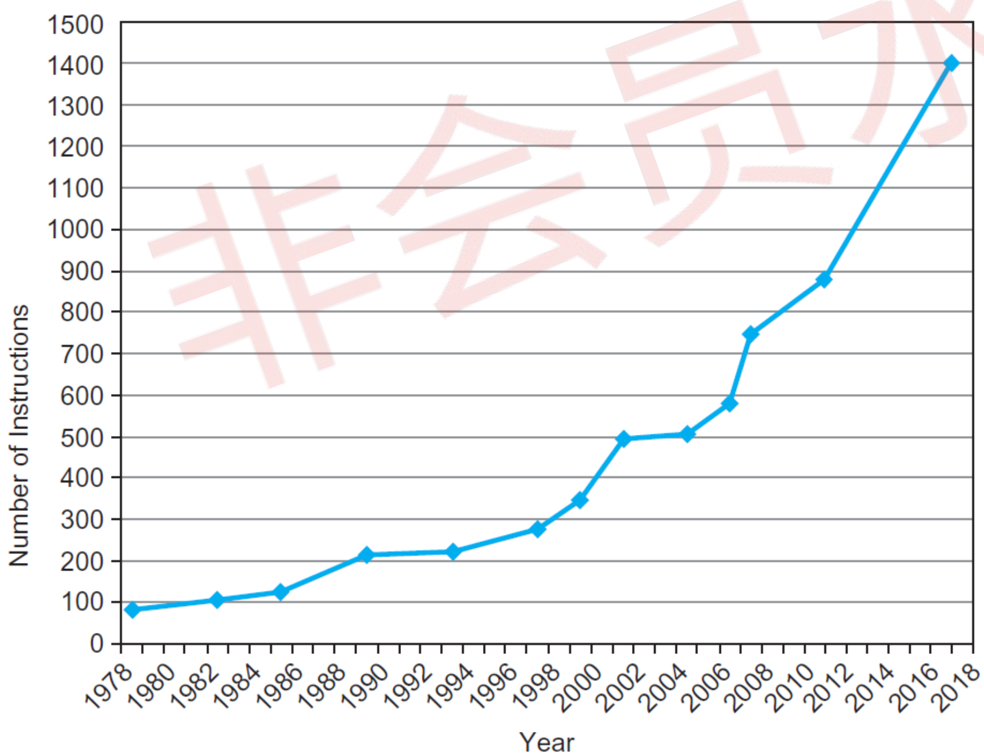
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%