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：

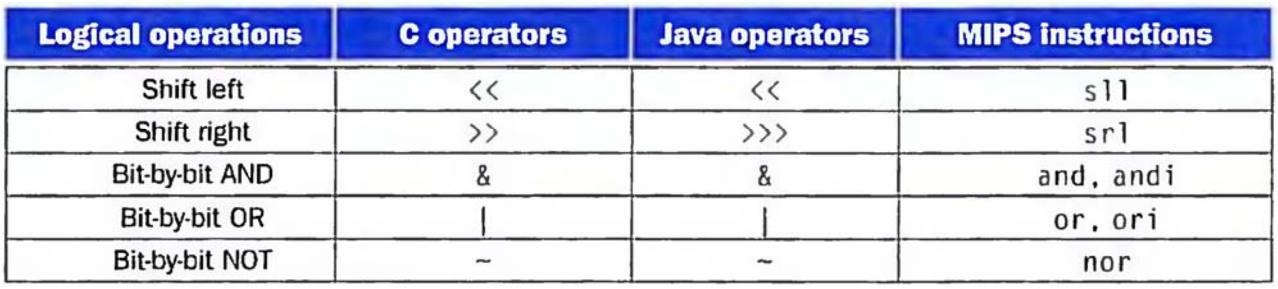
  To find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and power.

- 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.

2.6 Logical Operations

2.6 Logical Operations



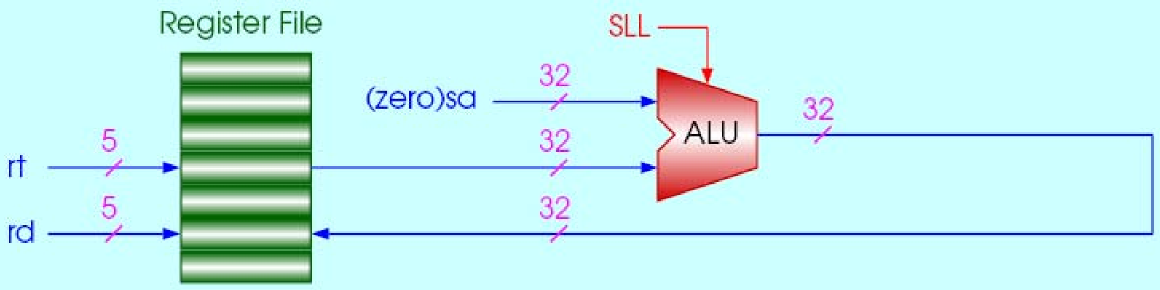
Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

sll 指令 (Shift Left Logical)

```
sll  rd, rt, sa  ; rd <-- rt << sa
```

op	rs	rt	rd	sa	funct
000000	00000	rt	rd	sa	000000
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

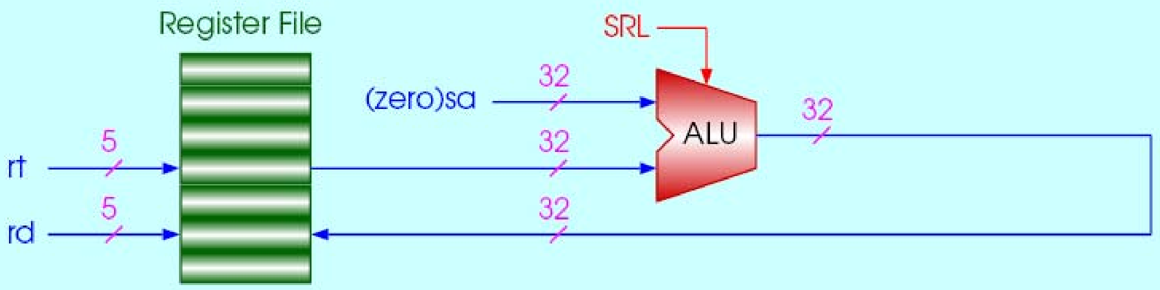


指令的意義: 把寄存器rt中的數據左移sa位, 結果存放在寄存器rd中

srl 指令 (Shift Right Logical)

```
srl  rd, rt, sa  ; rd <-- rt >> sa (logical)
```

op	rs	rt	rd	sa	funct
000000	00000	rt	rd	sa	000010
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

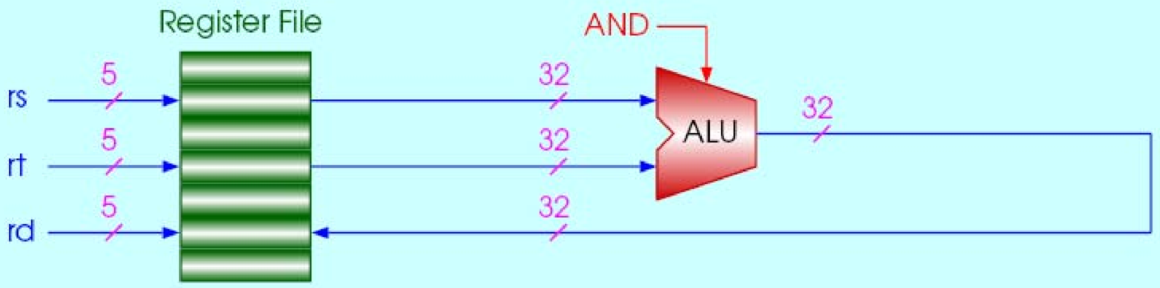


指令的意義: 把寄存器rt中的數據邏輯右移sa位, 結果存放在寄存器rd中

and 指令(And)

```
and   rd, rs, rt   ; rd <-- rs & rt
```

op	rs	rt	rd	sa	funct
000000	rs	rt	rd	00000	100100
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

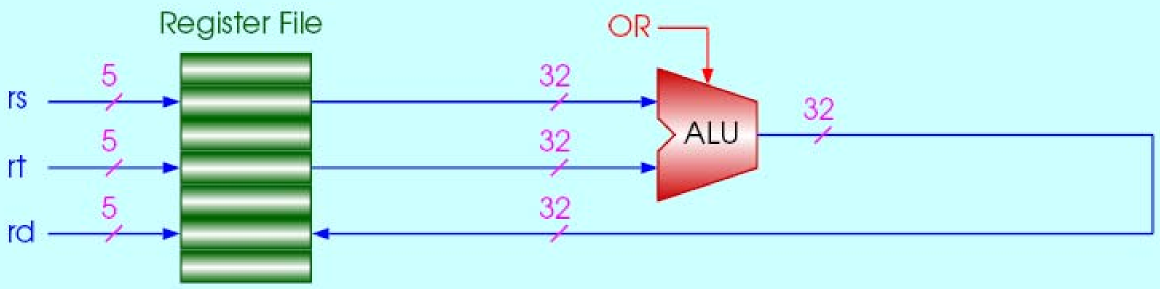


指令的意義: 寄存器rs中的數據和寄存器rt中的數據相與, 結果存放在寄存器rd中

or 指令 (Or)

```
or    rd, rs, rt   ; rd <-- rs | rt
```

op	rs	rt	rd	sa	funct
000000	rs	rt	rd	00000	100101
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

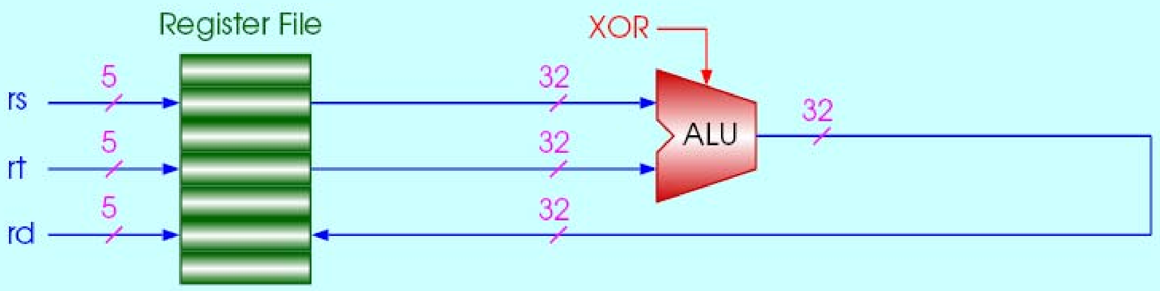


指令的意義: 寄存器rs中的數據和寄存器rt中的數據相或, 結果存放在寄存器rd中

xor 指令 (Exclusive Or)

```
xor  rd, rs, rt  ; rd <-- rs ^ rt
```

op	rs	rt	rd	sa	funct
000000	rs	rt	rd	00000	100110
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit



指令的意義: 寄存器rs中的數據和寄存器rt中的數據相异或, 結果存放在寄存器rd中

算术右移

算术右移/逻辑右移

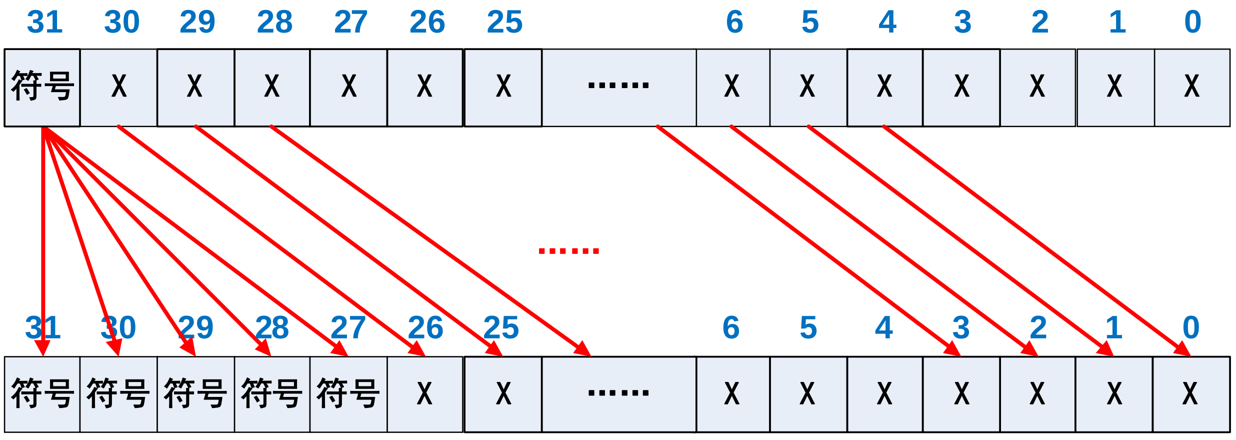
- 算术右移：在右移的过程中，左侧填充高位符号位。
- 算术左移：？右侧应该填充什么？ （如果）
- 目的：算术真值损失最小。

SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0]
SRL	Shift Right Logical	Rd = (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]

详细的MIPS指令系统可参考文档：
《MIPS32 4K™ Processor Core Family Software User's Manual》

算术右移

例：　SRA　$t1, $s1, 4　　# s1中的数值算术右移4位→ t1

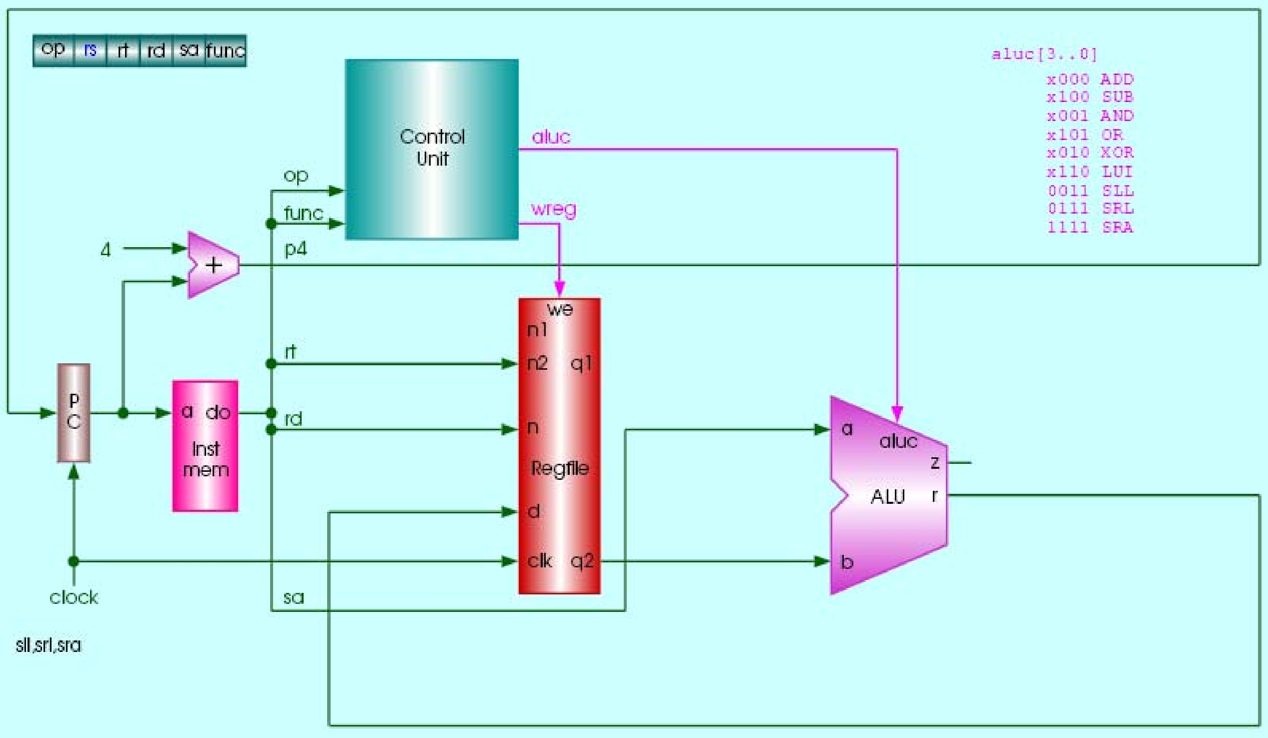


31 30 29 28 27 26 25 6 5 4 3 2 1 0
符号 X X X X X X …… X X X X X X X
……
31 30 29 28 27 26 25 6 5 4 3 2 1 0
符号 符号 符号 符号 符号 X X …… X X X X X X X

目的：算术真值损失最小。

另：移位操作还有一种循环移位。

【例】一种能够执行sll/srl/sra指令的CPU电路/datapath



2.6 Logical Operations

指令操作实例：

■ 已知：

（$t1）= 0000 0000 0000 0000 0011 1100 0000 0000 $_2$

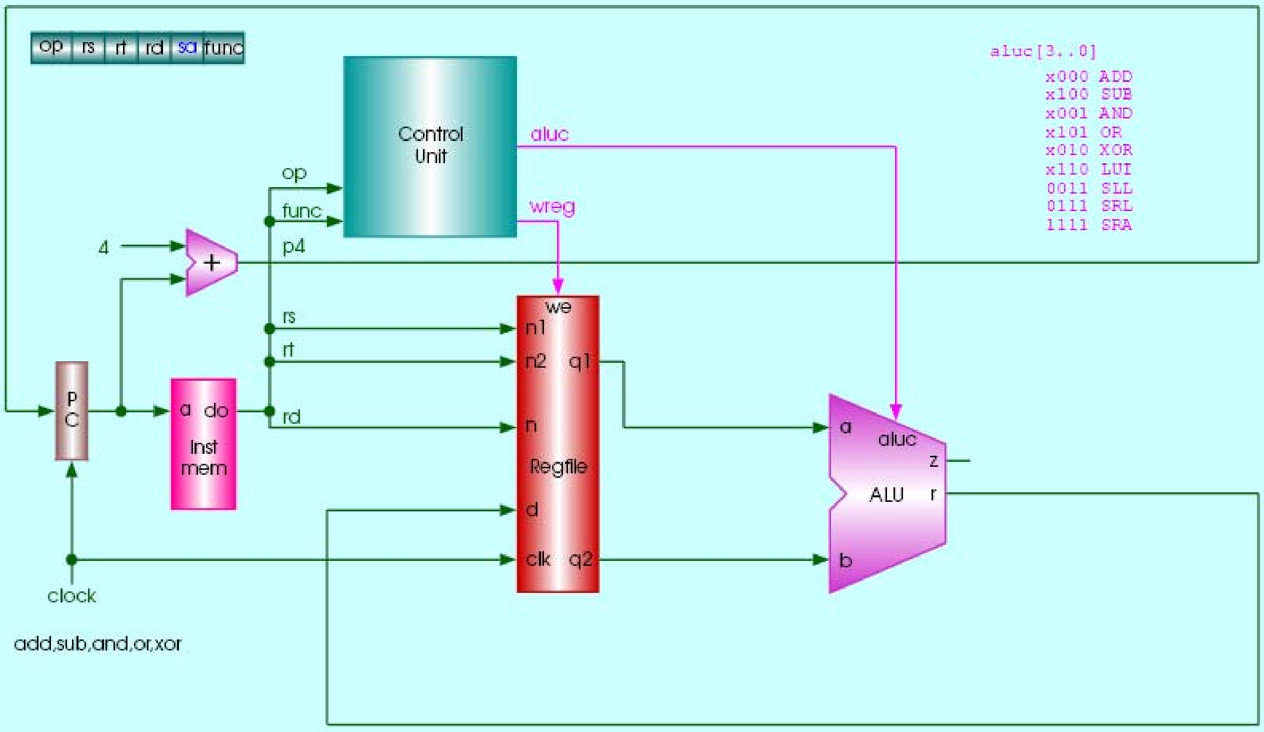
（$t2）= 0000 0000 0000 0000 0000 1101 1100 0000 $_2$

分别执行以下指令后，考察$t0中的值。

```
sll   $t0, $t1, 4      # reg $t0 =  reg $t1 <<  4 bits
and   $t0, $t1, $t2    # reg $t0 =  reg $t1  &   reg $t2
or    $t0, $t1, $t2    # reg $t0 =  reg $t1  |   reg $t2
nor   $t0, $t1, $t2    # reg $t0 =  - (reg $t1 |  reg $t2)

xor  $t1, $t1, $t2     # reg $t1 = reg $t1  ^  reg $t2 （有无疑问？）
xor  $t1, $t1, $t1     # reg $t1 = reg $t1  ^  reg $t1 （有无疑问？）
```

可以实现目的操作数与运算操作数相同（例）



2.6 Logical Operations

Check Yourself

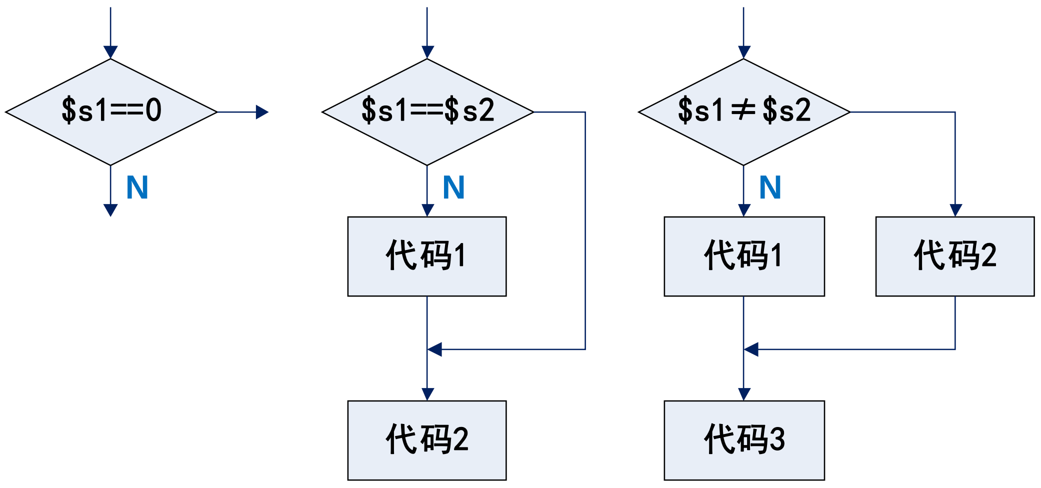
Which operations can isolate a field in a word?

1. AND
2. A shift left followed by a shift right

2.7 Instructions for Making Decisions

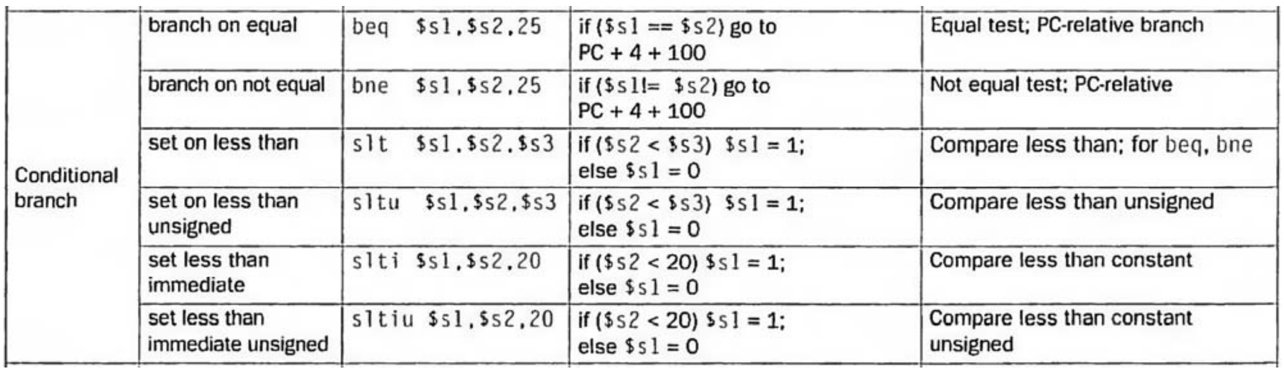
描述和解决问题中需要判断

- 判断（分支）是智能的最基本标志。
- What distinguishes a computer from a simple calculator is its ability to make decisions.



2.7 Instructions for Making Decisions

MIPS32中的部分条件转移（分支）指令



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

MIPS-4000指令说明中关于Branch指令的说明：

10.4.2 Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the16-bit offset (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.（delay slot：后续学习解释）

If a conditional branch likely is not taken, the instruction in the delay slot is nullified.

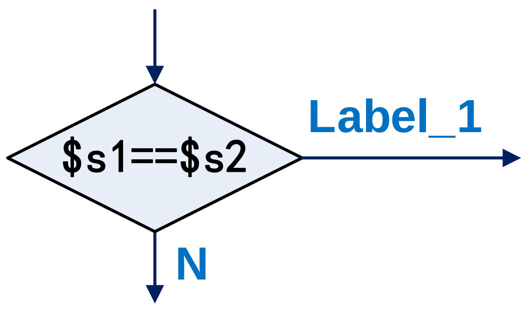
Branches, jumps, ERET, and DERET instructions should not be placed in the delay slot of a branch or jump.

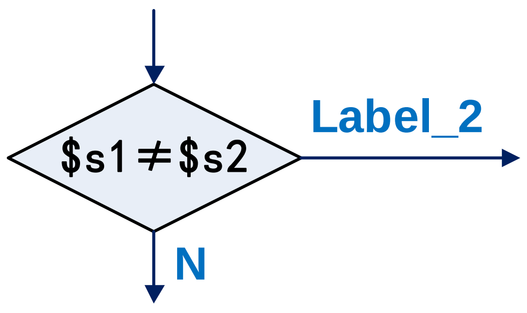
2.7 Instructions for Making Decisions

Conditional branch 条件转移指令

- beq：branch on equal
- bne：branch on not equal

beq $s1,$s2, Label_1 **bne $s1,$s2, Label_1**

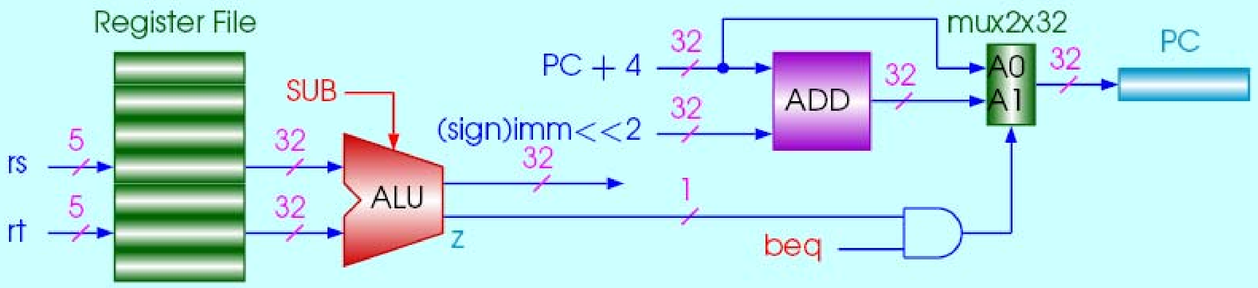




beq 指令 (Branch on Equal)

```
beq rs, rt, imm; if(rs==rt)PC<--PC+4+(sign)imm<<2
```

op	rs	rt	imm
000100	rs	rt	imm
6-bit	5-bit	5-bit	16-bit

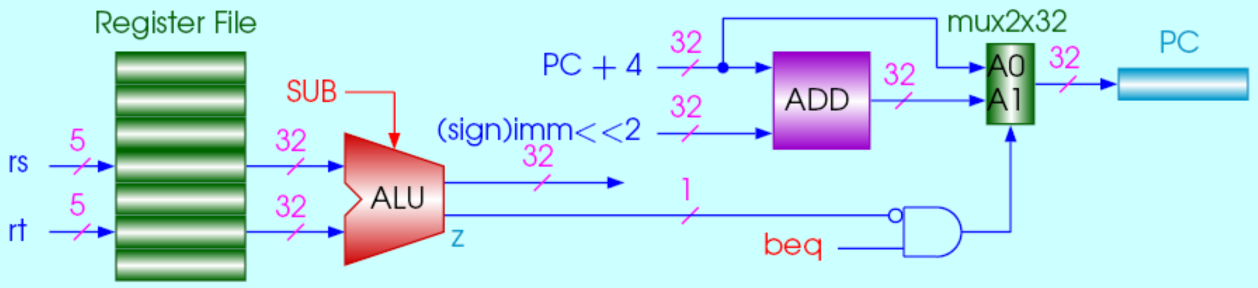


指令的意義: 如果寄存器rs中的數據和寄存器rt中的數據相等, 轉移到地址 PC + 4 + (sign)imm × 4

bne 指令 (Branch on Not Equal)

```
bne rs, rt, imm; if(rs!=rt)PC<--PC+4+(sign)imm<<2
```

op	rs	rt	imm
000101	rs	rt	imm
6-bit	5-bit	5-bit	16-bit



指令的意義: 如果寄存器rs中的數據和寄存器rt中的數據不等, 轉移到地址 PC + 4 + (sign)imm × 4

2.7 Instructions for Making Decisions

【例】Compiling if-then-else into Conditional Branches

■ C语言描述：

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

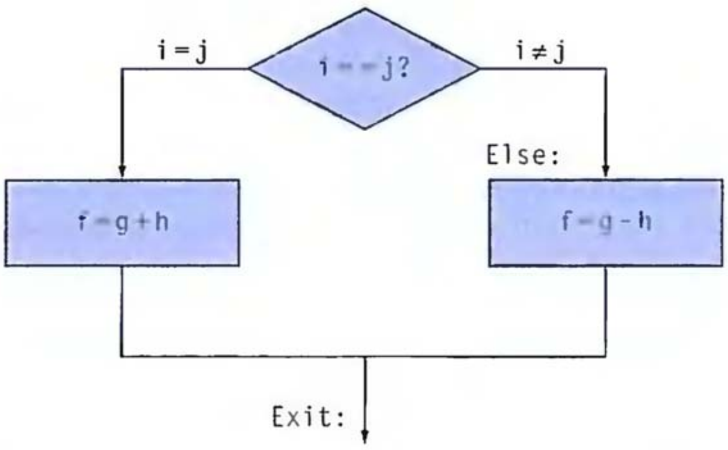


FIGURE 2.9 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

2.7 Instructions for Making Decisions

【例】Compiling if-then-else into Conditional Branches

■ C语言描述：

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

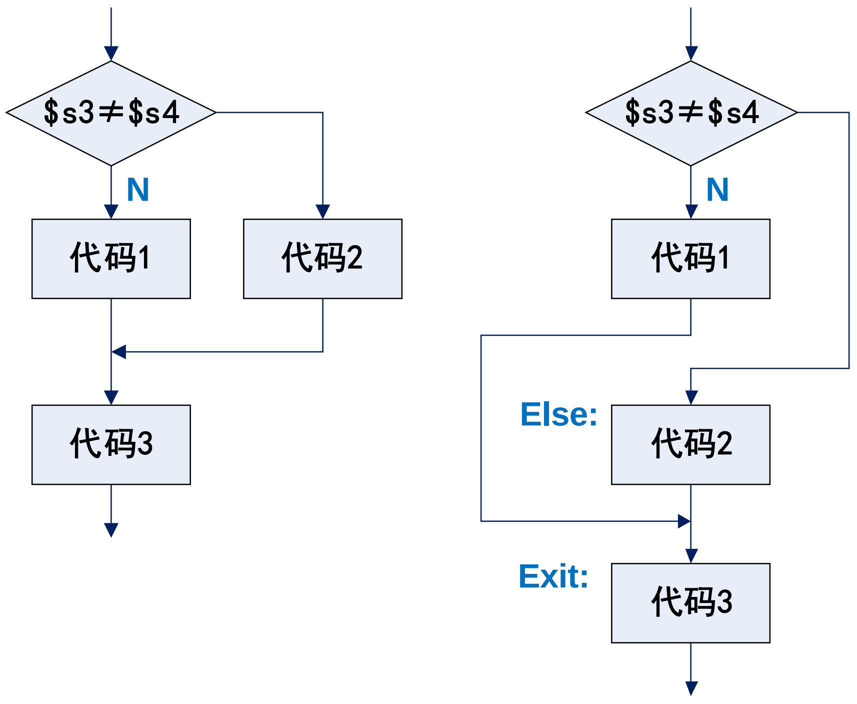
■ 经编译后的MIPS指令：

```
        bne   $s3, $s4, Else     #  go to Else if  i ≠ j
        add   $s0, $s1, $s2      #  f = g + h (skipped  if  i ≠ j)
        j     Exit               #  go to Exit  (无条件跳转指令)
Else:   sub   $s0, $s1, $s2      #  f = g - h (skipped if i == j)
Exit:
```

注：“Else”和“Exit”为用于指示位置的行标号,对应为“模块内的相对地址”。

2.7 Instructions for Making Decisions

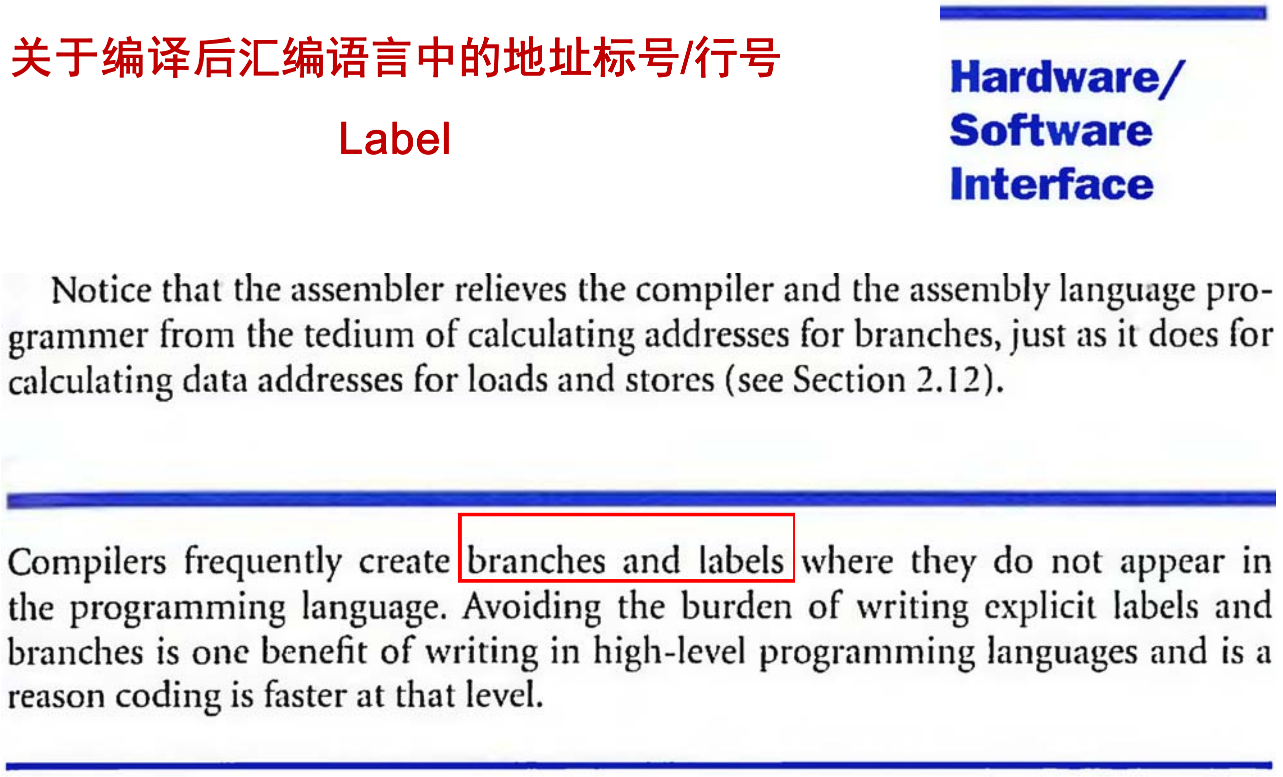
【例】Compiling if-then-else into Conditional Branches



2.7 Instructions for Making Decisions

关于编译后汇编语言中的地址标号/行号

Label



Hardware/ Software Interface

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.12).

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

关于循环操作/Loops

【例】Compiling a while Loop in C

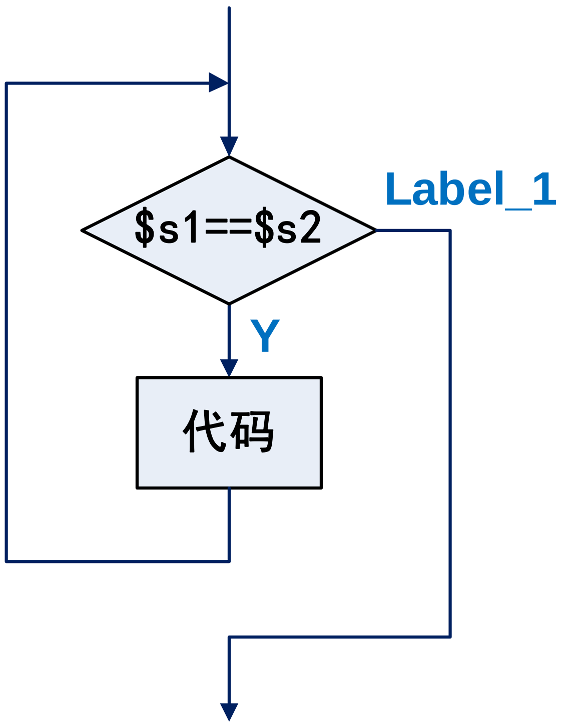
■ C语言描述：

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

Assume that i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6. What is the MIPS assembly code corresponding to this C segment?

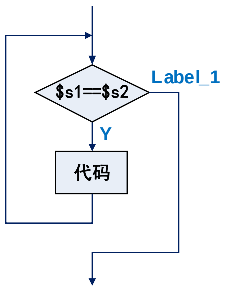
■ 经编译后的MIPS指令：

注：假定
变量 *i* 放在$s3中，
变量 *k* 放在$s5中，
数组save[ ]的基址放在$s6中。



关于循环操作/Loops

【例】Compiling a while Loop in C



■ C语言描述：

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

■ 经编译后的MIPS指令：

注：假定变量 i 放在$s3中，变量k放在$s5中，数组save[ ]的基址放在$s6中。

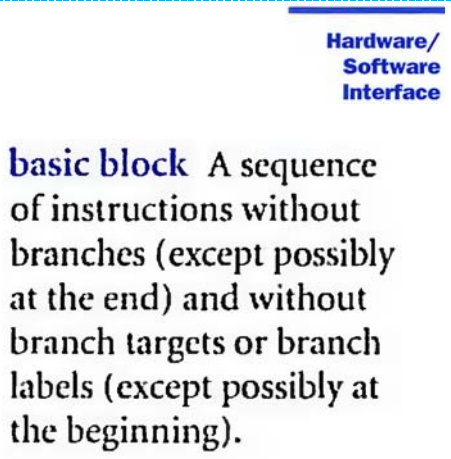
```
Loop:  sll   $t1, $s3, 2        #  Temp reg $t1 = i * 4
       add   $t1, $t1, $s6      #  $t1 = address of save[ i ]
       lw    $t0, 0($t1)        #  Temp reg $t0 = save[ i ]
       bne   $t0, $s5, Exit     #  go to Exit if save[ i ] ≠ k
       addi  $s3, $s3, 1        #  i = i + 1
       j     Loop               #  go to Loop
 Exit:
```

（See the exercises for an optimization of this sequence.）【可否及如何优化？】

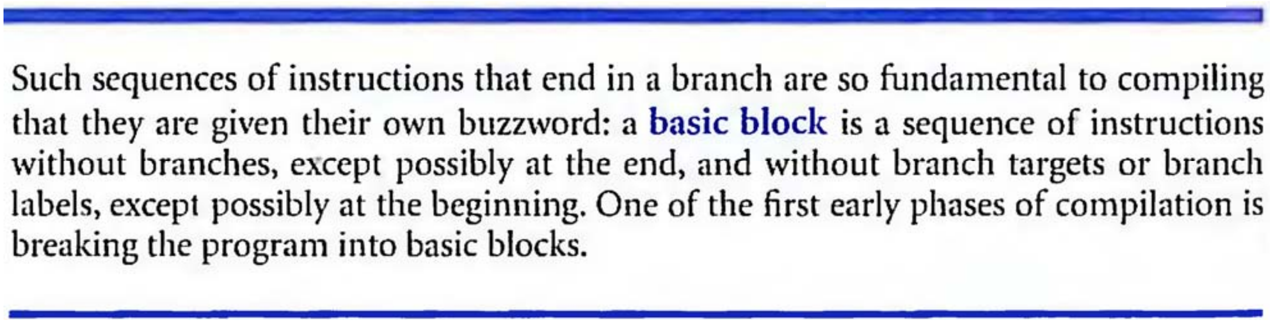
关于分支中的基本代码/语句块（basic block）

关于basic block （基本语句块）

- basic block：
  指一段指令代码，它不包含分支（除非在代码段末尾），也没有其它地方的跳转分支指向本段代码中的指令（除非指向本代码段的第一条指令）。

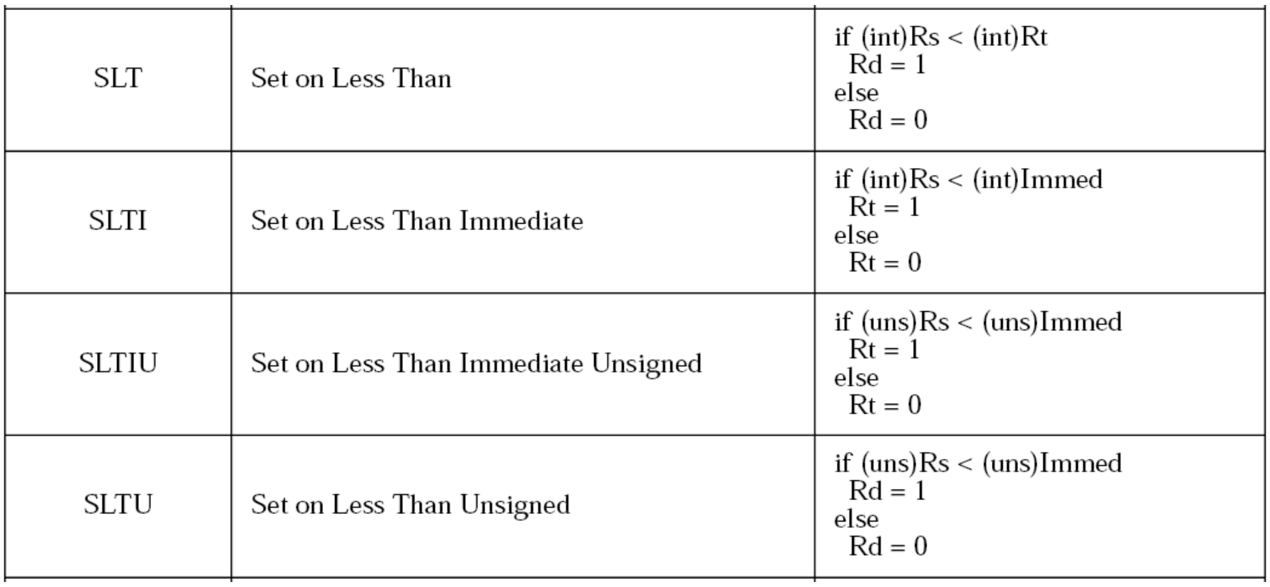

Hardware/ Software Interface

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).


Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

MIPS32的其它判比指令

- slt $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4 判比操作数为补码有符号数
- slti $t0, $s2, 10 # $t0 = 1 if $s2 < 10 判比操作数为补码有符号数
- sltu $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4 判比操作数为无符号数
- sltiu $t0, $s2, 10 # $t0 = 1 if $s2 < 10 判比操作数为无符号数



SLT	Set on Less Than	if (int)Rs < (int)Rt
		Rd = 1
		else
		Rd = 0

SLTI	Set on Less Than Immediate	if (int)Rs < (int)Immed
		Rt = 1
		else
		Rt = 0

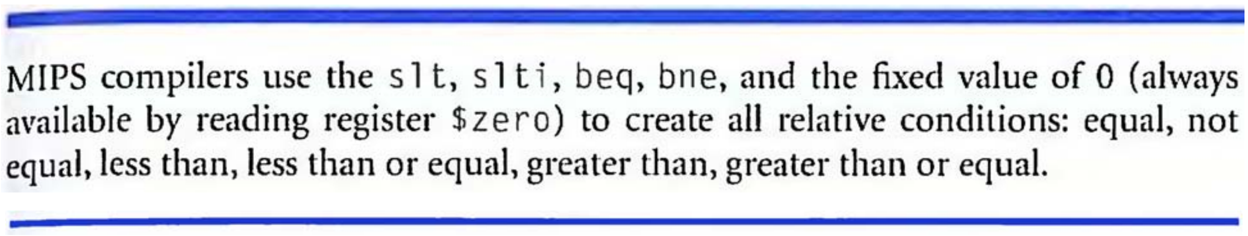
SLTIU	Set on Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed
		Rt = 1
		else
		Rt = 0

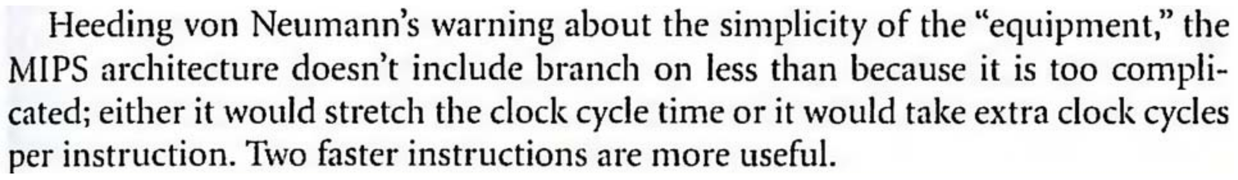
SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed
		Rd = 1
		else
		Rd = 0

2.7 Instructions for Making Decisions

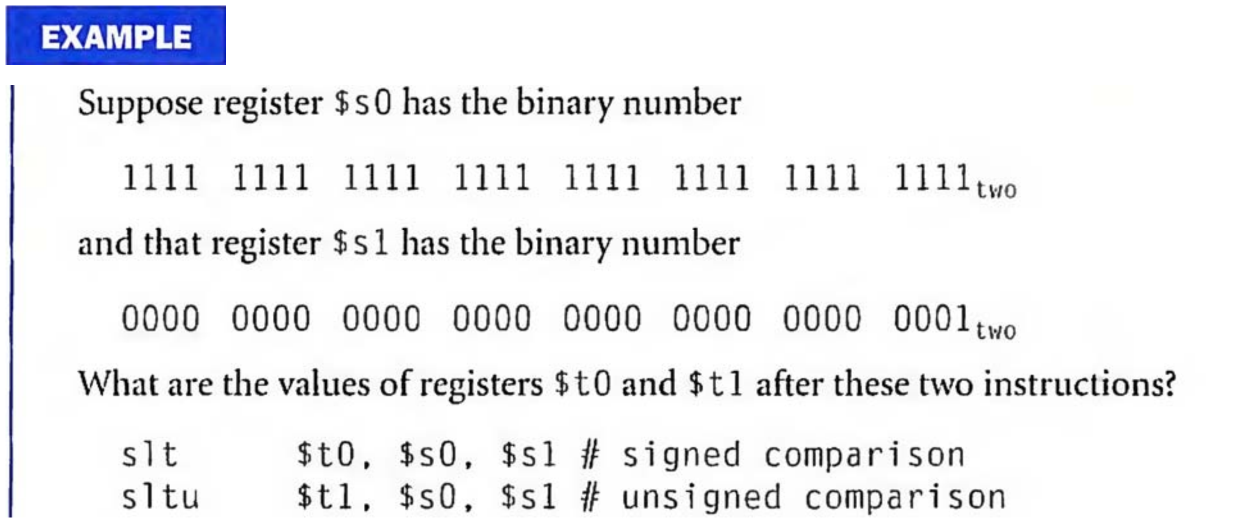
MIPS32所设计采用的条件判断和转移方法

- MIPS编译器利用slt、slti、sltu、slti进行条件判断的结果设置，利用beq、bne指令，以及它们的组合，实现所有条件转移。
  包括：相等转、不等转、小于转、小于等于转、大于转、大于等于转。
- 若直接采用“小于比较后跳转”，会导致该指令复杂、或执行时间偏长、或需额外时钟周期。（思考：在一条指令中实现为什么会比较复杂？）
- 用2条执行较快的指令，组合实现较复杂的判比跳转，总体效率会更高。


MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.


Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

【例】Signed versus Unsigned Comparison


EXAMPLE

Suppose register `$s0` has the binary number

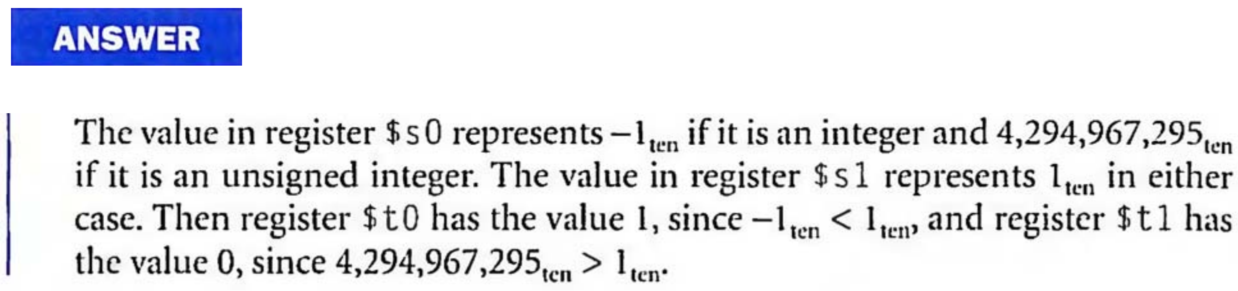
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register `$s1` has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers `$t0` and `$t1` after these two instructions?

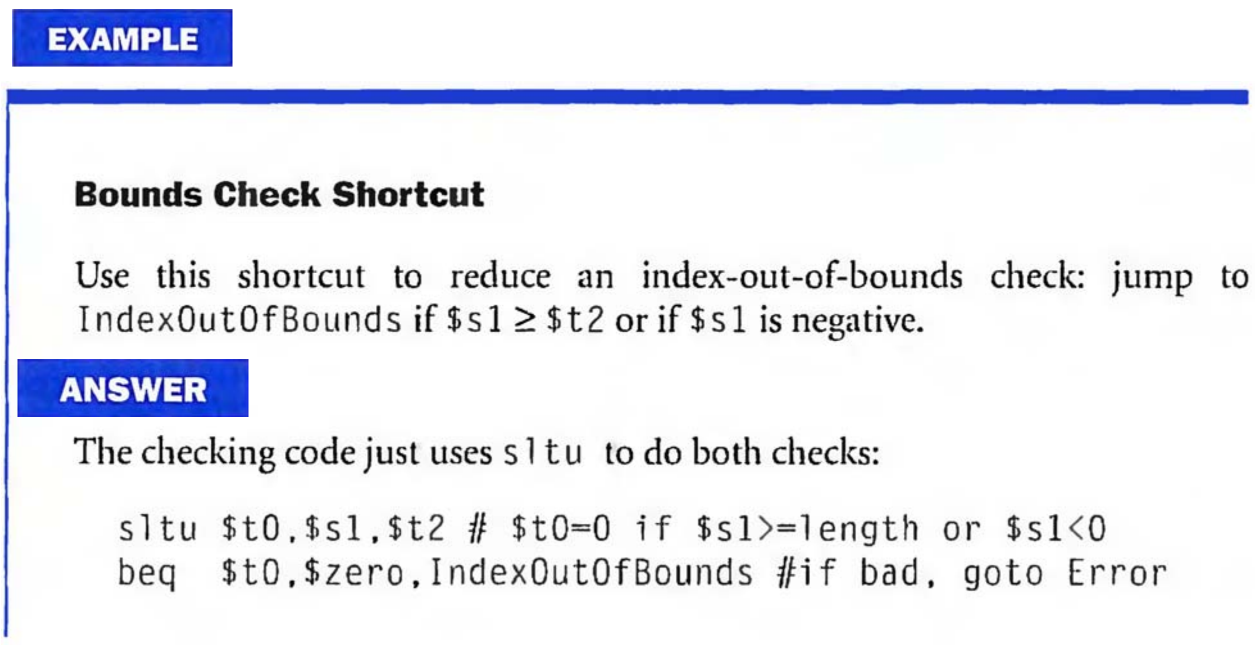
```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```


ANSWER

The value in register `$s0` represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1_{ten} in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

2.7 Instructions for Making Decisions

【例】关于数组下标是否越界的一种简便判法



EXAMPLE

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$s1` ≥ `$t2` or if `$s1` is negative.

ANSWER

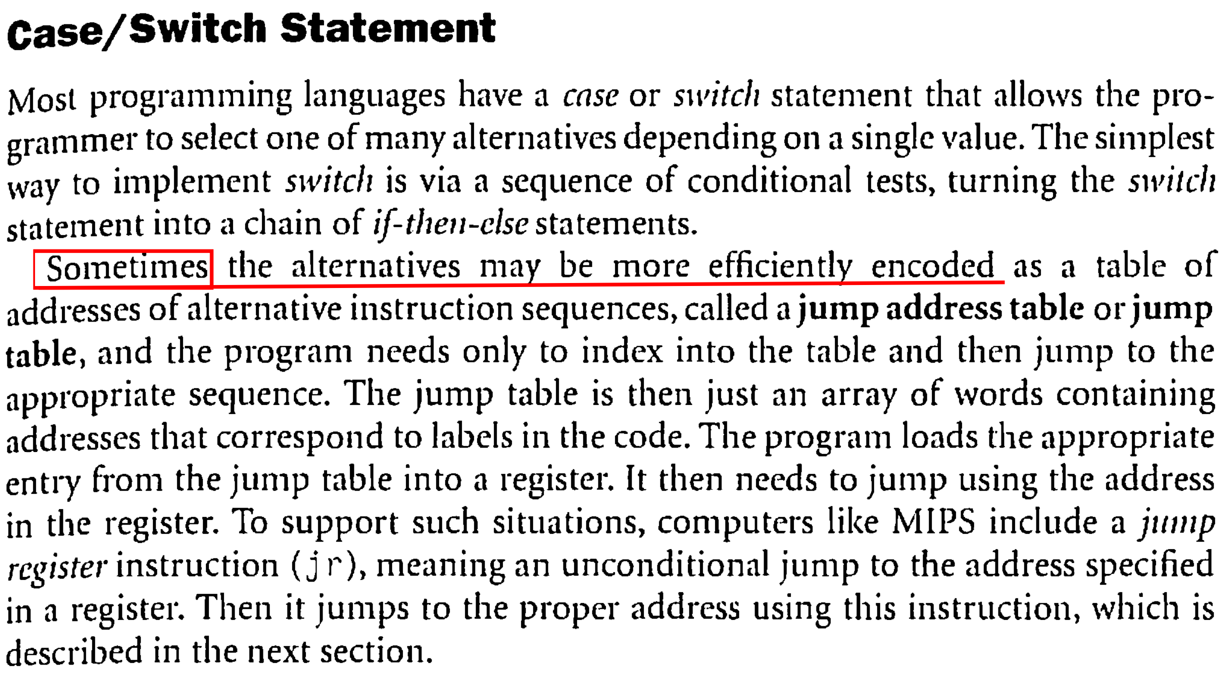
The checking code just uses `sltu` to do both checks:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

2.7 Instructions for Making Decisions

Case/Switch 语句的2种实现方法

- 方法1：利用多个 if-then-else 语句实现。
- 方法2：编译生成“跳转地址表”，直接索引跳至目标代码块。



Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction, which is described in the next section.

2.7 Instructions for Making Decisions

Check Yourself

I. C has many statements for decisions and loops, while MIPS has few. Which of the following do or do not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

2.7 Instructions for Making Decisions

Check Yourself

II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while MIPS doesn't?

1. Logical operations AND and OR implement & and |, while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

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)
  - Li Yamin(Hosei Univ.)