2. 指令：计算机的语言

计算机语言中的基本词汇（word）称为指令，一台计算机的全部指令称为该计算机的指令集（可见低级语言依赖于机器）。

一条指令首先要指明执行什么操作，一般用 0-1 串的前几位表示，称为**操作码（opcode）**；然后需要指出操作的数据来自哪里以及操作后的数据放到哪里，通常用 0-1 串的后几位表示，称为**操作数（operand）**或**地址码（address code）**。

注意，指令和数字对计算机来说没有区别，指令本身就是操作数。

2.1. MIPS 概述

此部分除了简单概述 MIPS 之外，还有汇总 MIPS 的一些常见规则（比如寄存器和指令等）来供快速查阅，需要结合本章其他内容理解。

MIPS（Microprocessor without Interlocked Piped Stages，无内部互锁流水级的微处理器）作为一种 RISC 指令集，设计力求保证硬件设备的简单性。在 MIPS 汇编语言中，所有指令都是 32 位（bit）（4 字节）长，体现着 1.3.4 硬件设计的基本原则：简单源于规整。

> **Note**
>
> MIPS 的一个指令长称为 1 个字（word），即：1 字=4 字节=32 位。

2.1.1. MIPS 操作数

本章 MIPS 操作数来自 32 个通用寄存器和内存，如 Figure 2-1 所示。

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.

Figure 2-1 MIPS Operands

MIPS 只能对寄存器里的数据执行操作！

2.1.2. MIPS 寄存器

MIPS 中的操作数必须来自寄存器或者指令本身。大量的寄存器可能使时钟周期变长，因为电信号传输需要时间。所以，这体现了 1.3.4 硬件设计的基本原则：越小越快（并非绝对）。

MIPS 的寄存器按照功能分为通用寄存器、协处理器 0、浮点寄存器、乘法部件寄存器和流水线寄存器。

本章着重介绍的是 32 个通用寄存器（General-Purpose Registers, GPR），均可以被程序员所使用。

名称	寄存器号	用法	叫法
$zero	0	存储常数 0	零寄存器
$at	1	为汇编器保留	at 寄存器
$v0 - $v1	2-3	过程/函数返回值（value）	（返回）值寄存器
$a0 - $a3	4-7	过程/函数参数（argument）	参数寄存器
$t0 - $t7	8-15	临时变量（temporary）	临时寄存器
$s0 - $s7	16-23	保存数值（saved）	保存寄存器
$t8 - $t9	24-25	临时变量	临时寄存器
$k0 -$k1	26-27	为操作系统保留，异常处理	内核（kernel）寄存器
$gp	28	全局指针（global pointer）	全局指针寄存器
$sp	29	栈指针（stack pointer）	栈指针寄存器
$fp	30	帧指针（frame pointer）	帧指针寄存器
$ra	31	返回地址 （return address）	返回地址寄存器

Figure 2-2 32 GPRs of MIPS

寄存器即可以用名称表示，也可以用 “$+寄存器号”表示。

2.1.3. 本章 MIPS 指令集

分类	指令	格式	写法	含义
乘除运算	add	R	`add rd, rs, rt`	rd=rs+rt
	subtract	R	`sub rd, rs, rt`	rd=rs-rt
	add immediate	I	`addi rt, rs, im`	rt=rs+im
数据传送	load word	I	`lw/ld rt, ad(rs)`	rt=Memory[rs+ad]
	store word	I	`sw/sd rt, ad(rs)`	Memory[rs+ad]=rt
	load upper immediate	I	`lui rt, im`	rt=im*2^{16}
	and	R	`and rd, rs, rt`	rd=rs&rt

逻辑运算	or	R	or rd, rs, rt	rd=rs|rt
	nor	R	nor rd, rs, rt	rd=~(rs|rt)
	xor	R	xor rd, rs, rt	rd=rs⊕rt
	and immediate	I	andi rt, rs, im	rt=rs&im
	or immediate	I	ori rt, rs, im	rt=rs|im
	xor immediate	I	xori rt, rs, im	rt=rs⊕im
	shift left logical	R	sll rd, rs, shamt	rd=rs<<shamt
	shift right logical	R	srl rd, rs, shamt	rd=rs>>shamt
条件分支	branch on equal	I	beq rs, rt, ad	if (rs==rt) go to PC+4+4*ad
	branch on not equal	I	bne rs, rt, ad	if (rs!=rt) go to PC+4+4*ad
	set on less than	R	slt rd, rs, rt	if (rs<rt) rd=1; else rd=0
	set on less than unsigned	I	sltu rt, rs, im	if (rs<rt(unsign)) rd=1; else rd=0
	set on less than immediate	I	slti rt, rs, im	if (rs<im) rt=1; else rt=0
	set on less than immediate unsigned	I	sltiu, rt, rs, im	if (rs<im(unsign)) rt=1; else rd=0
无条件跳转	jump	J	j address	go to 4*address
	jump register	R	jr rs	go to address in rs
	jump and link	J	jal address	$ra=PC+4; go to 4*address

Figure 2-3 MIPS Instructions in Chapter 2

伪指令	写法	翻译
move	move rd, rs, rt	add rd, rs, $zero
load immediate	li rt, immediate	addi rt, $zero, immediate
branch less than	blt rt, rs, Else	slt rd, rs, rt bnq rd, $zero, Else
branch less than or equal	ble rt, rs, Else	slt rd, rt, rs beq rd, $zero, Else
branch greater than	bgt rt, rs, Else	slt rd, rt, rs bnq rd, $zero, Else
branch greater than or equal	bge rt, rs, Else	slt rd, rs, rt beq rd, $zero, Else

Figure 2-4 Pseudo Instruction in Chapter 2

2.2. 基本 MIPS 汇编语言

本节用到的寄存器有$zero（零寄存器），$t0~$t7（临时寄存器），$s0~$s7（保留寄存器）和$t8~$t9（临时寄存器）共 19 个通用寄存器。

一般地（R 型和 I 型指令），MIPS 指令含有一个操作码和三个操作数，用汇编语言通法表示为“opcode des, src1, src2”（des 是 destination 的缩写，代表目标寄存器/目标操作数；src 是 source 的缩写，代表源寄存器/源操作数），也体现着“简单源于规整”的设计原则。

下面的指令中，如果 rs 和 rt 二者地位相同，即作为源操作寄存器无差异，将用 src1 和 src2 分别指代 rs 和 rt，借此来表示其先后顺序。

2.2.1. 算术运算（Arithmetic）

2.2.1.1. 加、减（add; subtract）

```
add/sub rd, src1, src2
```

2.2.1.2. 加立即数（add immediate）

```
addi rt, rs, immediate (常数)
```

Note

没有 subi（减立即数指令），因为可以 addi 一个负的立即数。

常数都是十进制数，机器识别运行时会转换对应的二进制码

Example 2.1

假定$s0 中的值为 128_{10}。则对于指令：

(1) add $t0, $s0, $s1; (2) sub $t0, $s0, $s1;

分别求出使结果产生溢出的$s1 的值的范围。

MIPS 寄存器均为 32 位二进制串，用补码表示数值

最大正整数为0x7FFF FFFF $= 2^{31} - 1$

最小负整数为0x8000 0000 $= -2^{31}$

(1)

当$s0+$s1$> 2^{31} - 1$或$s0+$s1$< -2^{31}$时溢出

此时，$s1$\in (-\infty, -2^{31} - 128) \cup (2^{31} - 129, +\infty)$

(2)

当$s0-$s1$> 2^{31} - 1$或$s0-$s1$< -2^{31}$时溢出

此时，$s1$\in (-\infty, -2^{31} + 129) \cup (2^{31} + 128, +\infty)$

Note

add（addi）/sub 指令会把立即数或者寄存器里的数看作有符号数。

计算机里的二进制串用补码表示，关于补码的知识，可以参考《数字电路与逻辑设计》。重要的是二进制补码与十进制的转换，设 n 位二进制补码 $x_{n-1}x_{n-2}\cdots x_1x_0$，则其十进制数为：

$$(-x_{n-1}2^{n-1}+x_{n-2}2^{n-2}+\cdots+x_12^1+x_02^0)_{10}=(x_{n-1}x_{n-2}\cdots x_1x_0)_2 \quad 2.1$$

另外可以快速求二进制补码数的相反数：将该数按位取反后再加 1。

Example 2.2

假定寄存器$s0 和$s1 分别存放数值 0x8000 0000 和 0xD000 0000，则执行代码：

(1) add $t0, $s0, $s1;

(2) sub $t0, $s0, $s1;

后，$t0 的值是多少，是否溢出？

(1)

0x8000 0000+0xD000 0000=0x1 5000 0000

但 1 溢出，故结果为 0x5000 0000

(2)

0x8000 0000-0xD000 0000

=0x$(1000-1101)_2$ 000 0000=0x 1011_2 000 0000=0xB000 0000 没有溢出

Note

加减运算等价，只有“正数+正数”，“正数+负数”和“负数+负数”三种情况，其中“正数+负数”永远不会溢出 。注意补码的最高位是负权重。

2.2.2. 逻辑运算（Logical）

2.2.2.1. 与、或（and; or）

逐位对应与、或。

```
and/or rd, src1, src2
```

Note

与、或也可以有立即数操作（and immediate; or immediate），即：

```
andi/ori rt, rs, immediate
```

and 可以将一种源操作数的某些位置 0，前提是另一个操作数中的对应位为 0，后者常被称为掩码（mask）。

2.2.2.2. 非、或非（not; nor）

任何数据与 0 进行或非操作，都会变为原来的非，为了保持 3 操作数格式，用或非指令取代非：

```
nor rd, src1, $zero # not
nor rd, src1, src2 # nor
```

2.2.2.3. 逻辑左移、右移（shift left/right logical）

```
sll/srl rd, src2, shamt
```

Note

逻辑左移可以实现乘法，左移x =shamt 位，相当于 rs 乘以2^x；同理逻辑右移可以实现除法，但用的不多。

逻辑移位属于 R 型指令！只是将 rs 置为 0。

2.2.3. 数据传送（Data Transfer）

MIPS 指令的操作数只能来自寄存器和指令本身，一些复杂的数据结构如数组、结构体等存储在内存中，需要使用数据传送指令进行通信。为了访问存储器中的一个字，指令必须给出存储器的地址（在存储器数组/阵列（array）中指明特定元素位置的值）。

2.2.3.1. 寄存器-存储器数据传送

将数据从存储器复制到寄存器的指令叫取数（load word）

```
lw rt, constant (常数) (rs)
```

将数据从寄存器复制到存储器的指令叫存数（store word）

```
sw rt, constant (常数) (rs)
```

rs 存放着存储器数组的基址（base address），即首元素的地址，被称为基址寄存器（base reg）；constant 称为偏移量（offset），基址加上偏移量组成访问元素的地址，注意偏移量是以字节为单位而非字，所以偏移量是 4 的倍数。

Note

存储器是按字节顺序编址的。由于 MIPS 的指令和数据都是 32 位（4 字节）长，32 位长的串无法用 1 个字节表示，所以一个字内部 4 个字节就需要有一定的字节序。

计算机按字节编址方式可分为“大端”（Big Endian）和“小端”（Little Endian）两种字节序。大端使用字最左侧字节的地址作为该字的地址，符合人的习惯；小

端使用字最右侧字节的地址作为该字的地址，符合字节地址的编址顺序。以数值 0x12345678 的存放为例：

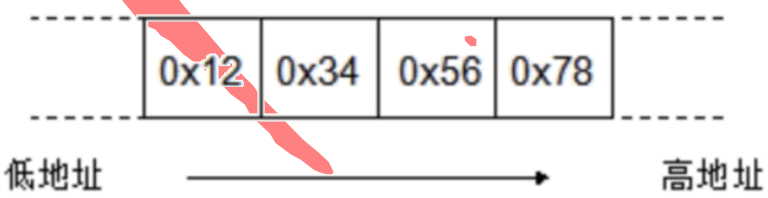

Figure 2-5 The Byte Order of Big Endian

高位字节数据存放在内存低地址处，低位字节数据存放在内存高地址处

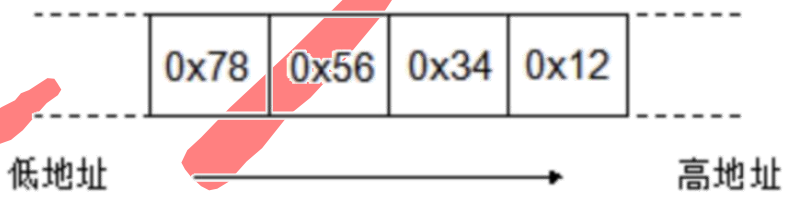

Figure 2-6 The Byte Order of Little Endian

高位字节数据存放在内存高地址处，低位数据存放在内存低地址处

因此字的起始地址必须是 4 的倍数：

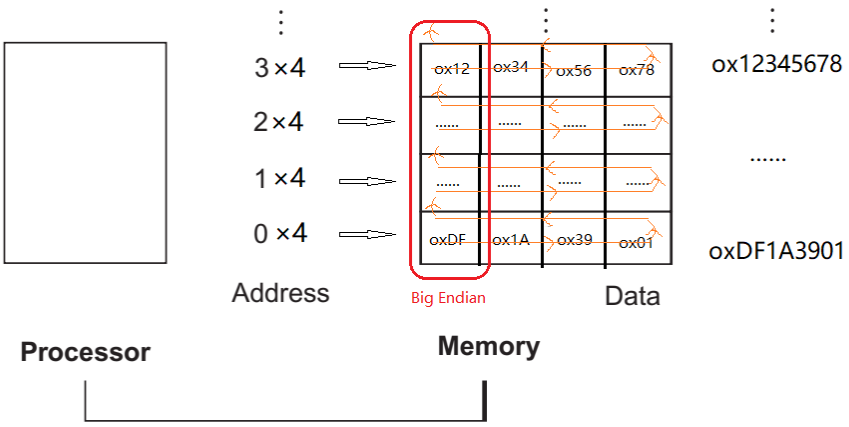

Figure 2-7 Memory Addresses and Contents of Memory at those Locations

2.2.3.2. 寄存器-寄存器数据传送

MIPS 中没有专门的寄存器间数据传送的指令，但是可以通过把源寄存器中的数据加上 0 后在保存到目标寄存器中，实现同样的功能：

```
addi rt, rs, 0
add rd, src1, $zero
```

这个功能可以用 move 伪指令（翻译为 add）来代替：

```
move rd, src1
```

Note

MIPS 伪指令（Pseudo MIPS）机器无法识别，需要翻译成 MIPS 中机器可以识别的指令再交给机器执行。在翻译伪指令时，汇编器会使用$at 寄存器将伪指令展开为多条机器可以识别的 MIPS 指令。

同样地，若要将一个常数装入寄存器，可以使用 addi 指令：

```
addi rt, $zero, immediate
```

用伪指令“取立即数（load immediate）”来代替为：

```
li rt, immediate
```

装载 32 位立即数

如何装载一个 32 位立即数到寄存器（立即数指令是 I 型，立即数最多只能占用指令的 16 位）呢？

可先把 32 位数的高 16 位放在寄存器的高 16 位，然后把寄存器的低 16 位置零，最后把待装载的 32 位立即数的低 16 位与寄存器的低 16 位进行或运算。

前两步可以使用“取高位立即数（load upper immdiate）”指令，同时实现取高位和置零低位；最后一步使用 ori 指令。以装载 10A2 7FFF$_{16}$ 为例（须将十六进制转换为十进制：10A2$_{16}$=4258$_{10}$，7FFF$_{16}$=32767$_{10}$），具体指令如下：

```
lui rt, 4258
ori rt, 32767
```

Note

不能使用 addi 代替 ori 指令，因为 addi 进行的是算术运算，如果低 16 位的最高位是 1，会把它理解为负数从而进行减法。

lui 指令是 I 型指令，rs 被置 0。

Example 2.3

将下列 C 语句编译为 MIPS 语句：

```
a[i]=a[1]+100000
```

假设数组 a 的基地址位于$s0，变量 i 位于$s1。

100000=0x1 86A0, 0x86A0=34464

```
# 装载 32 位立即数
lui $t0, 1
ori $t0, 34464
# 取 a[1]
lw $t1, 4($s0)
add $t2, $t0, $t1 # a[1]+100000
```

```
# 得&a[i]
sll $t3, $s1, 2
add $t4, $s0, $t3
sw $t2, 0($t4)
```

Note

偏移量是以字节为单位而非字，所以偏移量是 4 的倍数

2.2.4. 决策指令（Making Decisions）

计算机与简单计算器的区别在于决策能力，即计算机可以根据输入数据和计算过程中产生的值执行不同的指令。

2.2.4.1. 条件分支（Conditional Branch）

相等则分支（不等则分支），在两个源操作数寄存器中的值相同（不同）时则跳转到分支标签的位置：

```
beq src1, src2, Label # branch if equal
bne src1, src2, Label # branch if not equal
```

如果不发生分支，则继续执行内存中相邻的下一条指令。

Note

相等/不等则分支是 I 型指令。

高级语言中判断==，MIPS 使用 bne；判断!=，使用 beq。因为满足 if 的条件，则顺序执行下一条指令；不满足才跳转到对应分支。

2.2.4.2. 无条件分支（Unconditional Branch）/跳转（Jump）

当遇到无条件分支时，程序必须跳转到标签的位置，通常用于离开 if-else 语句块。MIPS 中的无条件分支指令一般称为“跳转”：

```
j Label
```

Example 2.4

In the following code segment, f, g, h, i, and j are variables. What is the compiled MIPS code for this C *if* statement?

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

Suppose that the five variables f through j correspond to the five registers $s0 through $s4.

```
    bne $s3, $s4, Else # 不等则分支
    add $s0, $s1, $s2
    j Exit
Else:
```

```
    sub $s0, $s1, $s2 # 执行完 else 语句后顺序离开 if-else 语句块
Exit:
```

Note

MIPS 不区分标签的大小写。

条件分支是 I 型指令，最多只能跳转到偏移量在 16 位以内的标签位置。在实现远距离跳转时，汇编器会自动在原条件分支语句后插入一条无条件转移指令。

例如，当 L1 的偏移量超过 16 位时，编译器把

```
  beq $s0, $s1, L1
```

变为

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

2.2.4.3. 小于则置位

MIPS 没有直接“小于/大于则分支”的指令，需要借助小于则置位这一基本指令实现：

```
slt rd, src1, src2
```

若 src1 存储的值小于 src2，则将 des 寄存器置位为 1（前 31 位为 0，第 32 位为 1）；若不小于（大于等于），则 des 复位为 0。

Note

置位：1/0 到 1；复位：1/0 到 0。小于则置位也有立即数操作，即：

```
slti rt, rs, immediate
```

除此之外，小于则置位还有一条“小于无符号数则置位（set on less than unsigned）”的指令，相应的也有其立即数指令：

```
sltu rd, src1, src2
sltiu rt, rs, immediate
```

该指令将源操作数均看作无符号数（此时最高位不再起符号位作用）。

Example 2.5

把有符号数看成无符号数时，有一种快速方法检查条件 $0 \leqslant x < y$ 是否满足（对于数组下标越界检查非常方便）。

假设 x, y 的数值分别存储在寄存器$s0 和$s1 中。

```
sltu $t0, $s0, $s1
# $t0=1 则满足，$t0=0 则不满足
```

Note

数组边界检查默认 y 是非负整数。

有符号数的最大正整数是 $7FFFFFFF_{16} = 2^{31}-1$（最高位为 0），负整数的最高位为 1。若 x 是非负整数，则 sltu 可以判断其是否在[0, y)区间：在则置位，不在则复位；若 x 是负整数，则将其看做无符号数时，因其最高位是 1，必然大于有符号的正整数 y，所以 sltu 将$t0 复位。

综上：sltu 可以判断其是否在[0, y)区间：在则置位，不在则复位。

2.2.4.4. 决策指令综合

所有比较条件的实现

借助 slt, beq, bne 可以组合实现全部六种条件（相等，不等，大于，大于等于，小于，小于等于），从而对应高级语言进行分支。

六种比较条件分支（比较 src1 和 src2 里的值）总结在 Figure 2-8 中：

比较条件分支	实现	伪指令
相等则分支	`beq src1, src2, Label`	无
不等则分支	`bne src1, src2, Label`	无
小于则分支	`slt rd, src1, src2` `bnq rd, $zero, Else`	`blt src1, src2, Label`
小于等于则分支	`slt rd, src2, src1` `beq rd, $zero, Else`	`ble src1, src2, Label`
大于则分支	`slt rd, src2, src1` `bnq rd, $zero, Else`	`bgt src1, src2, Label`
大于等于则分支	`slt rd, src1, src2` `beq rd, $zero, Else`	`bge src1, src2, Label`

Figure 2-8 Six Conditions of Branch Instructions

Note

与相等（不等）则分支一样，高级语言中 if (i ? j)编译成 MIPS 时，需要使用与?意义相反的 “?则分支”。例如，if (i < j)编译为 MIPS 时需要使用 “大于等于则分支（bge）”。

循环

Example 2.6: “while” Loop

将下方 C 语言的 while 循环编译成 MIPS 代码：

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

假设 i, k 分别存放在$s3 和$s5 中，a 的基址存放在$s6 中。

```
Loop:
```

```
  # 从内存中取得 a[i]给$t2
  sll $t0, $s3, 2
  add $t1, $s6, $t0 # $t1 是 a[i]地址
  lw $t2, 0($t1)
  # 判断循环条件
  bne $t2, $s5, Exit # 不等则离开出循环
  # 执行循环体
  addi $s3, $s3, 1
  j Loop
Exit:
```

Example 2.7: "for" Loop

Try to translate the following C code to MIPS assembly code with a minimum number of instructions.

```
for (i=0; i<a; ++i)
    for (j=0; j<b; ++j)
        D[4*j]=i+j;
```

Assume that the values of a, b, i, and j are in registers $s0, $s1, $t0, and $t1, respectively. Also, assume that register $s2 holds the base address of the array D.

```
  addi $t0, $zero, 0 # 初始化 i=0
  beq $zero, $zero, Judge1 # 判断条件复杂，跳转分支执行
Loop1:
  addi $t1, $zero, 0 # 初始化 j=0
  beq $zero, $zero, Judge2
Loop2:
  add $t3, $t1, $t0
  # 从内存中取得 D[4*j]地址给$t2
  sll $t2, $t1, 4 # 2^4=16=4*4
  add $t2, $t2, $s2
  # 将 i+j 存入 D[4*j]
  sw $t3, 0($t2)
  addi $t1, $t1, 1 # ++j
Judge2:
  slt $t4, $t1, $s1
  bne $t4, $zero, Loop2
  addi $t0, $t0, 1 # ++i
Judge1:
  slt $t5, $t0, $s0
  bne $t5, $zero, Loop1
```

Note

当判断条件复杂时（嵌套、不允许使用伪指令等），需要用恒成立跳转至一个专门实现判断的标签。

循环嵌套一般从循环体到判断条件、从内到外书写。这样写的好处是：当跳出循环时（分支不发生），可以直接执行接下来的指令，而无需额外的跳转。

Example 2.8: 数组和指针

下面是 C 程序编写的 clear 函数，分别用数组下标访问和指针操作实现：

```
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;
}
```

试写出两种实现方式的 MIPS 汇编程序并比较二者的性能。假设参数 array 和 size 分别存储在寄存器$a0 和$a1 中，i 保存在$t0 中。

用数组实现 clear：

```
  addi $t0, $zero, 0 # move $t0, $zero
  slt $t3, $t0, $a1
  beq $t3, $zero, Exit
Loop:
  sll $t1, $t0, 4
  add $t2, $t1, $a0
  sw $zero, 0($t2)
  addi $t0, $t0, 1
  slt $t3, $t0, $a1
  bne $t3, $zero, Loop
Exit:
```

用指针实现 clear：

```
  move $t0,$a0 # p = address of array[0]
  sll $t1,$a1,2
  add $t2,$a0,$t1 # $t2 = address of array[size]
  beq $zero, $zero, Judge
Loop:
  sw $zero,0($t0)
  addi $t0,$t0,4
Judge:
  slt $t3,$t0,$t2 # $t3 = (p<&array[size])
  bne $t3,$zero,Loop
```

显然，用指针实现消除了数组地址的计算，循环内的指令数更少，性能更高。

Note

传参时，array[]和*array 没有任何区别，都被理解为数组的基址。

在上面的两段汇编程序中，分别用两种方式实现了进入循环前对 size 是否等于 0 的检查，显然后者使用的指令更少（复用了循环内的判断条件而不是独立判断）。

尽管在 C 中使用指针比数组的效率更高，但指针会使代码难于理解，不如数组直观。现在大部分程序员为了让代码便于理解，更喜欢让编译器去做更繁重的工作，现代的优化编译器也可以使数组产生同样好的代码。

以分支指令结束的指令序列称为基本块（basic block），基本块没有分支（或者分支在末尾）并且没有分支标签（可以在开始），例如循环的循环体。