（上图中第 0 位），**最高有效位**（most significant bit）指的是最左边的位（上图中第 63 位）。

最高有效位：以 RISC-V 为例，指双字中最左边的位。

RISC-V 的双字为 64 位宽，因此可以表示 2^{64} 种不同的组合模式。这些组合自然就可以表示从 0 到 $2^{64}-1$ 之间（$18\,446\,774\,073\,709\,551\,615_{10}$）的数：

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_2 = 0_{10}$$
$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_2 = 1_{10}$$
$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_2 = 2_{10}$$
$$\ldots \qquad\qquad \ldots$$
$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_2 = 18\,446\,774\,073\,709\,551\,613_{10}$$
$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_2 = 18\,446\,744\,073\,709\,551\,614_{10}$$
$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_2 = 18\,446\,744\,073\,709\,551\,615_{10}$$

也就是说，64 位二进制数可以用每位的数值乘上 2 的幂之和（这里 x_i 表示 x 的第 i 位）来表示：

$$(x_{63}\times 2^{63})+(x_{62}\times 2^{62})+(x_{61}\times 2^{61})+\cdots+(x_1\times 2^1)+(x_0\times 2^0)$$

稍后将会看到，由于一些原因，这些正数称为无符号数。

硬件/软件接口 以 2 为基数不符合人类常识。我们有 10 个手指，自然会用 10 作基数。为什么计算机不使用十进制？事实上，第一台商用计算机确实提供了十进制算术。问题在于计算机仍然使用开关信号，需要用若干个二进制数位来表示一个十进制数，因此十进制被证明是非常低效的。所以后来的计算机全部转向了二进制，只将相对不频繁的输入/输出事件转为十进制。

注意，上面的二进制位模式只是数的简单表示。实际上数有无限多位，除了最右边的少数几位以外，其余大部分都是 0。因此，通常不显示前导 0。

可以为加、减、乘、除操作的二进制位模式设计硬件。如果这些操作结果正确但不能用最右端的硬件位来表示，则称*溢出*发生。如何处理溢出取决于编程语言、操作系统和程序。

计算机程序可以计算正数和负数，因此需要一种表示方法来区分正数和负数。最显然的解决方法是增加一个单独的符号位来表示，这种表示方法称为*原码*（sign and magnitude）表示。

然而，这种表示有若干缺点。首先，在哪里放置符号位并不明确，放在右边还是左边？早期的计算机都尝试过。其次，由于无法预先知道符号是什么，这种加法器可能需要额外的步骤来设置符号。最后，一个单独的符号位意味着这种表示既有正零又有负零，这会给疏忽大意的程序员带来问题。这些缺点导致符号和幅值表示方法很快就被放弃了。

在寻找一个更具吸引力的替代方法时出现了这样一个问题，如果试图从一个小数中减去一个大数，无符号数表示方法的结果会是什么？答案是它会尝试从一串前导 0 中借位，所以结果会有一串前导 1。

鉴于没有明显的更优替代方法，最终解决方案是选择简化硬件的表示方法：前导 0 表示正数，前导 1 表示负数。这种表示有符号二进制数的约定称为*二进制补码*（two's complement）表示法：

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_2 = 0_{10}$$
$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_2 = 1_{10}$$
$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_2 = 2_{10}$$

```
...                                                                ...
01111111 11111111 11111111 11111111 11111111 11111111 11111111 11111101₂ = 9 223 372 036 854 775 805₁₀
01111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110₂ = 9 223 372 036 854 775 806₁₀
01111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111₂ = 9 223 372 036 854 775 807₁₀
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000₂ = - 9 223 372 036 854 775 808₁₀
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001₂ = - 9 223 372 036 854 775 807₁₀
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010₂ = - 9 223 372 036 854 775 806₁₀
...                                                                ...
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111101₂ = - 3₁₀
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110₂ = - 2₁₀
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111₂ = - 1₁₀
```

从 0 到 $9\ 223\ 372\ 036\ 854\ 775\ 807_{10}$（$2^{63}-1$）的正数与之前的表示相同。位模式 $1000\cdots0000_2$ 表示最小负数 $-9\ 223\ 372\ 036\ 854\ 775\ 808_{10}$（$-2^{63}$）。而后是一组递增的负数：从 $-9\ 223\ 372\ 036\ 854\ 775\ 807_{10}$（$1000\ \ldots\ 0001_2$）到 -1_{10}（$1111\ \ldots\ 1111_2$）。

二进制补码表示确实有一个负数没有相应的正数：$-9\ 223\ 372\ 036\ 854\ 775\ 808_{10}$。这种不平衡也会令疏忽大意的程序员发愁，但是符号和幅值表示却会给程序员与硬件设计人员都带来问题。因此，现在所有计算机都使用二进制补码来表示有符号数。

二进制补码表示的优点是，所有负数的最高有效位都为 1。因此，硬件只需要检测这一位就可以查看是正数还是负数（数字 0 被认为是正数）。这个位通常被称为符号位（sign bit）。理解了符号位的作用，就可以用每位数值乘以 2 的幂之和来表示正负数的 64 位数：

$$(x_{63} \times -2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

符号位乘以 -2^{63}，然后其余位分别乘以它们各自基值的正值。

例题 | 二进制转换为十进制

下面这个 64 位二进制补码的十进制数是多少？

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_2$$

答案 |

$$\begin{aligned}
&(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^1) + (0 \times 2^1) + (0 \times 2^0) \\
&= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^2 + 0 + 0 \\
&= -9223372036854775808_{10} + 9223372036854775804_{10} \\
&= -4_{10}
\end{aligned}$$

稍后将给出一个简化负数转正数计算的捷径。

正如对无符号数的操作结果可能超出硬件容量而产生溢出一样，对二进制补码的操作也是如此。当二进制位模式下最左边的保留位与左边的无限数位不相同时（即符号位不正确），溢出发生：当数为负数时最左侧为 0，或当数为正数时最左侧为 1。

硬件 / 软件接口 有符号相比无符号更适用于载入和算术运算。有符号载入的功能是复制符号位填充寄存器的剩余部分（称为符号扩展），其目的是在该寄存器中正确表示该数。由于位模式表示的是无符号数，因此无符号载入只需在数据的左侧填充 0。

当把一个 64 位双字载入一个 64 位寄存器中时，上述讨论是无意义的，此时有符号数和无符号数的载入是相同的。RISC-V 确实提供了两种字节载入方式：无符号字节载入（lbu）将字节视为无符号数，因此用零扩展填充寄存器的最左位，而字节载入（lb）使用带符号整数。由于 C 程序几乎总是使用字节来表示字符，而不是将字节视为有符号短整数，所以 lbu 实际上专门用于字节载入。

硬件/软件接口 与上面讨论的有符号数不同，内存地址自然地从 0 开始并延续到最大的地址。换句话说，负地址是没有意义的。因此，程序有时需要处理可正可负的数，而有时需要处理只能为正的数。一些编程语言反映了这种区别。例如，C 语言将以上两种情况中的前者称作整数（在程序中声明为 long long int）并将后者称作无符号整数（unsigned long long int）。一些 C 语言风格的指导书甚至建议将前者声明为 signed long long int，加以清晰区别。

看一下使用二进制补码时的两种有用的快捷方式。第一种是对二进制补码求相反数的快速方法。简单地把每个 0 都转为 1 以及每个 1 都转为 0，然后对结果加 1。这个捷径是基于以下观察：一个数与其取反表达式的和一定是 $111\ldots111_2$，它表示 −1。由于 $x + \bar{x} = -1$，因此 $x + \bar{x} + 1 = 0$ 或 $\bar{x} + 1 = -x$。（用符号 $\bar{x}$ 表示 x 按位取反。）

例题 | 求相反数的捷径

对 2_{10} 求相反数，然后通过对 -2_{10} 求相反数来验证结果。

答案 |

$2_{10} = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_2$

通过按位取反再加 1 对该数求反，

$$\begin{array}{rl} & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_2 \\ + & 1_2 \\ \hline = & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_2 \\ = & -2_{10} \end{array}$$

另一方面，对

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_2$

先取反再加 1：

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_2 \\ + & 1_2 \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_2 \\ = & 2_{10} \end{array}$$

第二种方式是将一个用 n 位表示的二进制数转换为一个用多于 n 位表示的数。先取位数更少的数的最高位（符号位），并将其复制来填充位数更多的数的新位。原来的非符号位被复制到新双字的右侧部分。这个方式通常被称为符号扩展（sign extension）。

例题 | 符号扩展的快捷方式

将 16 位二进制数 2_{10} 和 -2_{10} 转换成 64 位二进制数。

答案 | 数字 2 的 16 位二进制表示为：

$$00000000\ 00000010_2 = 2_{10}$$

通过把最高有效位（0）复制 48 份放到双字的左侧，将其转换为 64 位数。把原来的值放到右侧：

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_2 = 2_{10}$$

使用前面的快捷方式对 16 位二进制数 2 求反。因此，

$$0000\ 0000\ 0000\ 0010_2$$

变成

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_2 \\ +\qquad\qquad\qquad\quad 1_2 \\ \hline = 1111\ 1111\ 1111\ 1110_2 \end{array}$$

将负数转换为 64 位意味着要将符号位复制 48 次并放到左侧：

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_2 = -2_{10}$$

该方式之所以有效是因为在正数二进制补码的左边实际上是无限个 0，而负数的二进制补码在左边是无限个 1。二进制位模式隐藏了前面的位以适应硬件的宽度，符号扩展只是恢复其中的一些。

总结

本节的重点是要在计算机中表示正整数和负整数，尽管所有表示方法都有优缺点，但自 1965 年以来的一致选择都是二进制补码表示方法。

详细阐述 对于有符号十进制数，由于其没有大小限制，因此使用“−”来表示负数。对于给定数据大小的二进制和十六进制（见图 2-4）位串，可以对符号进行编码，因此通常不使用带“+”或“−”的二进制或十六进制符号。

详细阐述 二进制补码的命名源于下述规则：一个 n 位数与其 n 位相反数的无符号和为 2^n，因此，一个数 x 的相反数或“二进制补码”为 $2^n - x$。

“二进制补码”和“原码”以外的第三种替代表示方法称为反码。反码的相反数通过反转每一位，0 变成 1，1 变成 0 或 $\bar{x}$ 来求得。因为 x 的补数是 $2^n - x - 1$，这正好可以解释补码的命名。它试图成为一种比“原码”更好的解决方案，并且一些早期的科学计算机确实使用了这种表示方法。它与二进制补码相似，但它有两个 0：$00...00_2$ 为正 0，而 $11...11_2$ 为负 0。最小负数 $10...000_2$ 表示 $-2\ 147\ 483\ 647_{10}$，所以正数和负数的个数是平衡的。反码加法器中需要一个额外的步骤来减去一个数字，因此，现在的计算机中二进制补码占主导地位。

反码：一种数的表示方法。用 $10...000_2$ 表示最小负数，$01...11_2$ 表示最大正数，负数和正数的个数相等，且有两个零，一个正零（$00...00_2$）和一个负零（$11...11_2$）。该术语也用于表示按位取反：0 变 1，1 变 0。

第 3 章中讨论浮点时，将会看到最后一种表示方法。它用 $00...000_2$ 表示最小负数，$11...11_2$ 表示最大正数，而 0 通常表示为 $10...00_2$。这种表示方法通过给数加上偏移量得到一个非负表示形式，因此称为偏移表示法（biased notation）。

偏移表示法：一种数的表示方法。用 $00...000_2$ 表示最小负数，$11...11_2$ 表示最大正数，$10...000_2$ 表示零，通过给数加上偏移量得到一个非负表示形式。

自我检测 以下 64 位二进制补码数的十进制数是多少？

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111000_2

1. -4_{10}
2. -8_{10}
3. -16_{10}
4. $18\,446\,744\,073\,709\,551\,609_{10}$

2.5 计算机中的指令表示

人们使用计算机指令的方式和计算机识别指令的方式是不同的，现在我们开始解释这种差异。

指令以一系列高低电平信号的形式保存在计算机中，并且以数字的形式表示。实际上，每条指令的各个部分都可以被视为一个单独的数，把这些数字并排拼到一起便形成了指令。RISC-V 的 32 个寄存器也只是用 0 到 31 这些数来表示。

例题 | 将一条 RISC-V 汇编指令翻译为一条机器指令

下面以 RISC-V 为例，对于符号表示为

```
add x9, x20, x21
```

的 RISC-V 指令，首先以十进制表示，然后用二进制表示。

答案 | 十进制表示为：

0	21	20	0	9	51

一条指令的每一段称为一个字段。第一、第四和第六个字段（0、0 和 51）组合起来告诉 RISC-V 计算机该指令执行加法操作。第二个字段给出了作为加法运算的第二个源操作数的寄存器编号（21 表示 x21），第三个字段给出了加法运算的另一个源操作数（20 代表 x20）。第五个字段存放要接收总和的寄存器编号（9 代表 x9）。因此，该指令将寄存器 x20 和寄存器 x21 相加并将和存放在寄存器 x9 中。

该指令也可表示为二进制的形式：

0000000	10101	10100	000	01001	0110011
7 位	5 位	5 位	3 位	5 位	7 位

这种指令的设计被称为**指令格式**。从位数可以看出，这个 RISC-V 指令只需要 32 位，刚好是一个字或一个双字的一半。按照“简单源于规整”的设计原则，RISC-V 指令都是 32 位长。

指令格式：由二进制数字字段组成的指令表示形式。

为了把它和汇编语言区分开来，我们把指令的数字表示称作**机器语言**，把这样的指令序列称作机器码。

机器语言：用于计算机系统内通信的二进制表示。

也许你现在正在阅读和编写冗长的二进制字串。为了避免这种乏味，我们可以通过使用比二进制基数更大且更容易转换为二进制的进制表示。由于几乎所有的计算机数据大小都是 4 的倍数，所以**十六进制**（基数为 16）数很流行。由于基数 16 是 2 的幂，所以我们

十六进制数：以 16 为基数的数字表示。

可以通过将二进制数的每4位替换为一位16进制数来完成二进制到十六进制的转换，反之亦然。图2-4显示了在十六进制和二进制之间的转换。

十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制
0_{16}	0000_2	4_{16}	0100_2	8_{16}	1000_2	c_{16}	1100_2
1_{16}	0001_2	5_{16}	0101_2	9_{16}	1001_2	d_{16}	1101_2
2_{16}	0010_2	6_{16}	0110_2	a_{16}	1010_2	e_{16}	1110_2
3_{16}	0011_2	7_{16}	0111_2	b_{16}	1011_2	f_{16}	1111_2

图2-4　十六进制－二进制转换表。只需将一位十六进制数替换为相应的4位二进制数，反之亦然。如果二进制数的长度不是4的倍数，则从右向左进行转换

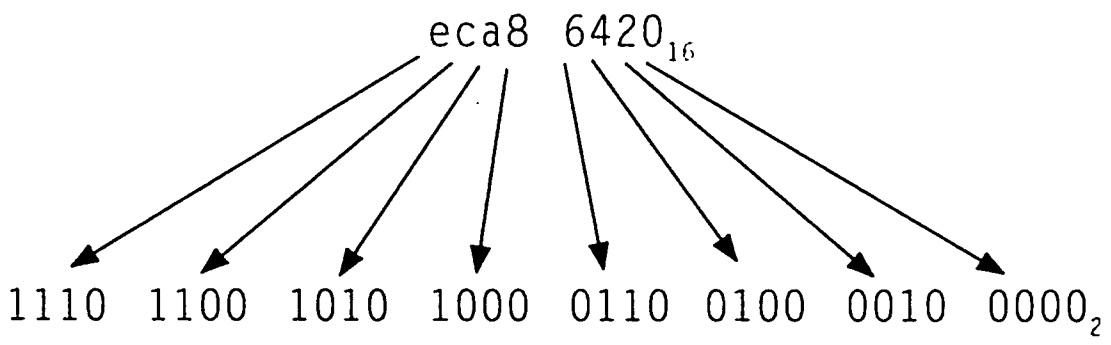
因为经常要处理不同的进制，为了避免混淆，我们给十进制数加下标10，给二进制数加下标2，给十六进制数加下标16。（如果没有下标，则默认为十进制。）此外，C和Java使用符号0x*nnnn*来表示十六进制数。

例题｜二进制和十六进制间的转换

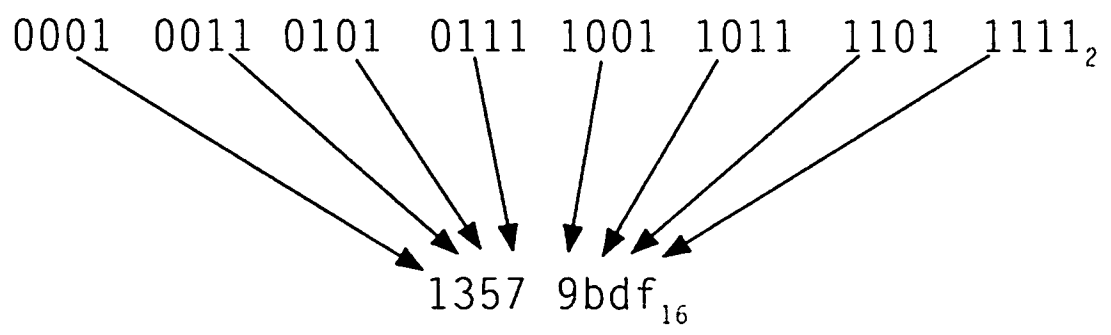
将下面的8位十六进制数转换为二进制，将32位二进制数转换为十六进制：

$$\texttt{eca8}\ \ 6420_{16}$$
$$0001\ \ 0011\ \ 0101\ \ 0111\ \ 1001\ \ 1011\ \ 1101\ \ 1111_2$$

答案｜利用图2-4，答案只需查表一次即可得到：



按反方向查表一次：



RISC-V字段

给RISC-V字段命名使其更易于讨论：

funct7	rs2	rs1	funct3	rd	opcode
7位	5位	5位	3位	5位	7位

以下是RISC-V指令中每个字段名称的含义：

- opcode（**操作码**）：指令的基本操作，这个缩写是它的惯用名称。
- rd：目的操作数寄存器，用来存放操作结果。
- funct3：一个另外的操作码字段。
- rs1：第一个源操作数寄存器。

操作码：用于表示指令操作和指令格式的字段。

- rs2：第二个源操作数寄存器。
- funct7：一个另外的操作码字段。

当指令需要比上面显示的更长的字段时就会出现问题。例如，加载寄存器指令必须指定两个寄存器和一个常数。如果地址使用上述格式中 5 位字段中的一个，则加载寄存器指令内的最大常数将被限制为 2^5-1 或 31。该常数用于从数组或数据结构中取数，并且它通常需要比 31 大得多的数。所以 5 位字段太小，用处不大。

因此，在所有指令保持相同长度的需求和保持单一的指令格式的需求之间产生了矛盾，这个矛盾也将我们引向最终的设计原则。

设计原则 3：优秀的设计需要适当的折中。

RISC-V 设计人员选择的折中方案是保持所有指令长度相同，对于不同的指令使用不同的指令格式。例如，上面的格式称为 R 型（用于寄存器）。另一种指令格式的类型是 I 型，用于带一个常数的算术指令（例如 addi）以及加载指令。I 型的字段如下所示：

immediate	rs1	funct3	rd	opcode
12 位	5 位	3 位	5 位	7 位

12 位 immediate 字段为补码值，所以它可以表示从 -2^{11} 到 $2^{11}-1$ 之间的整数。当 I 型格式用于加载指令时，immediate 字段表示一个字节偏移量，所以加载双字指令可以取相对于基址寄存器 rd 中基地址偏移 ±（2^{11} 或 2048）字节（±（2^8 或 256）个双字）的任何双字。我们发现，超过 32 个寄存器在这种格式下使用起来会很困难，因为 rd 和 rs1 字段都需要增添额外的一位，这导致一个字是不够的。

让我们分析一下 2.3.1 节提到的加载寄存器指令：

```
ld x9, 64(x22) // Temporary reg x9 gets A[8]
```

这里，22（x22）存放在 rs1 字段中，64 存放在 immediate 字段中，9（x9）存放在 rd 字段中。我们还需要一个存储双字指令 sd 的指令格式，它需要两个源寄存器（用于基址和存储数据）和一个用于地址偏移量的 immediate 字段。S 型的字段如下所示：

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 位	5 位	5 位	3 位	5 位	7 位

S 型格式的 12 位 immediate 字段分成了两个字段，低 5 位和高 7 位。RISC-V 体系结构设计师选择这种设计是因为它能够在所有指令格式中保持 rs1 和 rs2 字段在相同的位置。保持尽可能相似的指令格式降低了硬件的复杂性。同样，opcode 和 funct3 字段也总是保持同样的大小并在同一个位置。

指令格式通过操作码字段中的值来区分：每个格式在第一个字段（opcode）中被分配了一组不同的操作码值，以便硬件知道如何处理指令的其余部分。图 2-5 显示了迄今为止涉及的所有 RISC-V 指令在每个字段中的值。

指令	格式	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

图 2-5 RISC-V 指令编码。这里“reg”表示 0 至 31 之间的寄存器编号，“address”表示 12 位地址或常量。funct3 和 funct7 字段充当附加的操作码字段

指令	格式	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011
指令	**格式**	**immediate**	**rs2**	**rs1**	**funct3**	**opcode**
sd (store doubleword)	S	address	reg	reg	011	0100011

图 2-5 （续）

例题 | 将 RISC-V 汇编语言翻译为机器语言

现在我们可以举一个例子来描述从程序员编写程序到计算机执行指令的整个过程。假设数组 A 的基址存放于 x10，h 存放于 x21，则赋值语句：

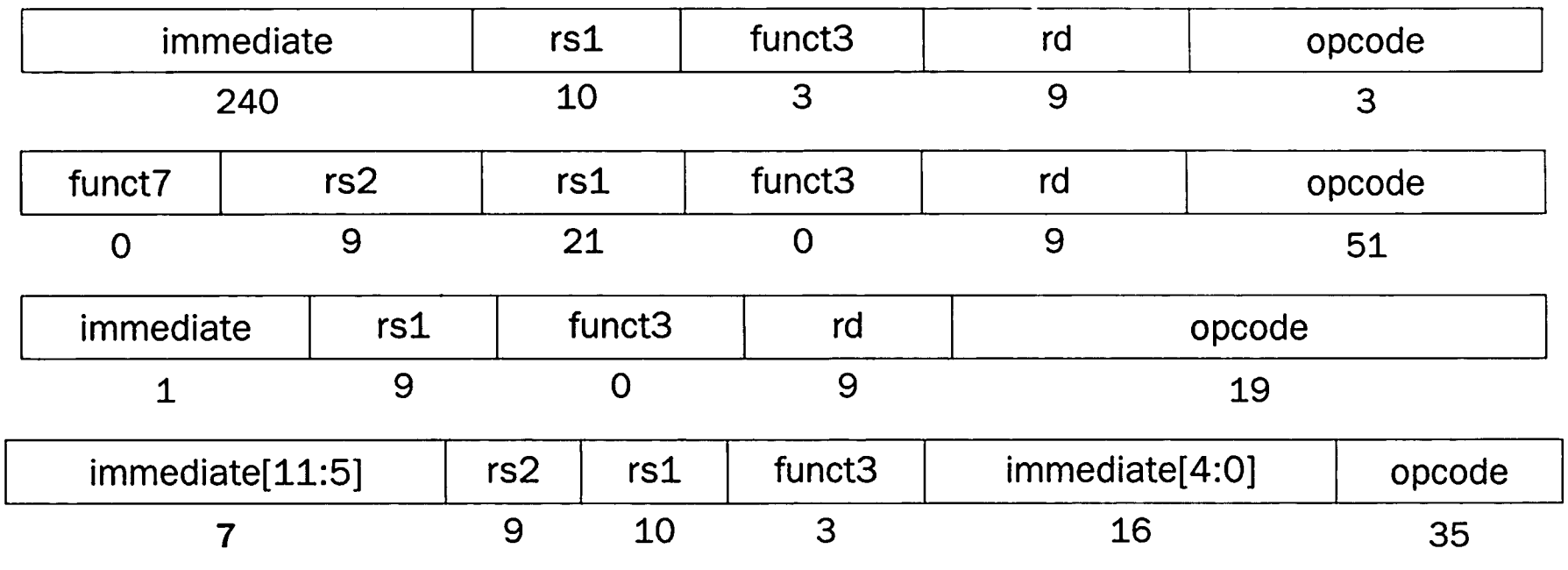
```
A[30] = h + A[30] + 1;
```

被编译成：

```
ld   x9, 240(x10)    // Temporary reg x9 gets A[30]
add  x9, x21, x9     // Temporary reg x9 gets h+A[30]
addi x9, x9, 1       // Temporary reg x9 gets h+A[30]+1
sd   x9, 240(x10)    // Stores h+A[30]+1 back into A[30]
```

这三条指令的 RISC-V 机器语言代码是什么？

答案 | 为了方便起见，我们首先使用十进制数来表示机器语言指令。从图 2-5 中，我们可以确定三条机器语言指令：



ld 指令由操作码字段中的值 3（见图 2-5）和 funct3 字段中的值 3 共同标识。基址寄存器 10 在 rs1 字段中指定，目标寄存器 9 在 rd 字段中指定。用于选定 A[30]（240 = 30 × 8）的偏移量存放在 immediate 字段中。

接下来的 add 指令由操作码字段中的值 51、funct3 字段中的值 0 和 funct7 字段中的值 0 共同指定。三个寄存器操作数（9、21 和 9）存放于 rd、rs1 和 rs2 字段中。

随后的 addi 指令由操作码字段中的值 19 和 funct3 字段中的值 0 共同决定。寄存器操作数（9 和 9）存放在 rd 和 rs1 字段中，常量加数 1 存放于 immediate 字段中。

sd 指令由操作码字段中的值 35 和 funct3 字段中的值 3 共同标识。寄存器操作数（9 和 10）分别存放于 rs2 和 rs1 字段中。地址偏移量 240 分开存放于两个 immediate 字段之中。由于高位的 immediate 字段已经包含了低 5 位的值，所以可以通过除以 2^5 来分解偏移量 240。则高位的 immediate 字段存储除以 2^5 的商 7，低位的 immediate 字段存储余数 16。

因为 $240_{10} = 0000\ 1111\ 0000_2$，所以与上述十进制对应的二进制机器指令如下所示：

immediate	rs1	funct3	rd	opcode
000011110000	01010	011	01001	0000011

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000111	01001	01010	011	10000	0100011

详细阐述 在处理常量时，RISC-V 汇编语言程序员不会被强迫使用 addi。程序员只需简单地写上一个 add，根据操作数是否都取自寄存器（R 型）或有一个操作数是常量（I 型），汇编器就能生成正确的操作码和指令格式。我们在 RISC-V 中对于不同的操作码和指令格式使用显式名称，这样使得在引入汇编语言和机器语言时不会混淆。

详细阐述 虽然 RISC-V 同时具有 add 和 sub 指令，但它并没有与 addi 相对应的 subi 指令。因为 immediate 字段表示的是二进制补码整数，所以 addi 可以用来做常数减法。

硬件 / 软件接口 保持所有指令长度相同的需求与设置尽可能多的寄存器的需求相矛盾。任何增加的寄存器数量都会让指令格式的每个寄存器字段至少增加一位。鉴于这些约束和更少则更快的设计原则，如今的大多数指令系统体系结构都是设置 16 或 32 个通用寄存器。

图 2-6 总结了本节中描述的 RISC-V 机器语言的各个部分。正如我们将在第 4 章中所看到的，相关指令的二进制表示的相似性简化了硬件设计。这些相似之处是 RISC-V 体系结构中规整性的另一个实例。

R型指令	funct7	rs2	rs1	funct3	rd	opcode	示例
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I型指令	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**示例**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S型指令	**immediate**	**rs2**	**rs1**	**funct3**	**immediate**	**opcode**	**示例**
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

图 2-6 2.5 节介绍的 RISC-V 体系结构。目前为止介绍的三种 RISC-V 指令格式是 R 型、I 型和 S 型。R 型格式有两个源寄存器操作数和一个目标寄存器操作数。I 型格式用一个 12 位的 immediate 字段替换了一个源寄存器操作数。S 型格式有两个源操作数和一个 12 位的 immediate 字段，但没有目标寄存器操作数。S 型的 immediate 字段分为两部分，最左边的字段是 11 ～ 5 位，最右边的字段是 4 ～ 0 位

重点 当前的计算机构建基于两个关键原则：

1. 指令由数字形式表示。
2. 程序和数据一样保存在存储器中来进行读写。

这些原则引出了“存储程序”的概念，这一发明让计算机突破了瓶颈。图 2-7 展示了这个概念的力量。具体而言，存储器可以存储一个编辑器程序的源代码、相应的编译后的机器

代码、编译程序正在使用的文本，甚至是生成机器代码的编译器。

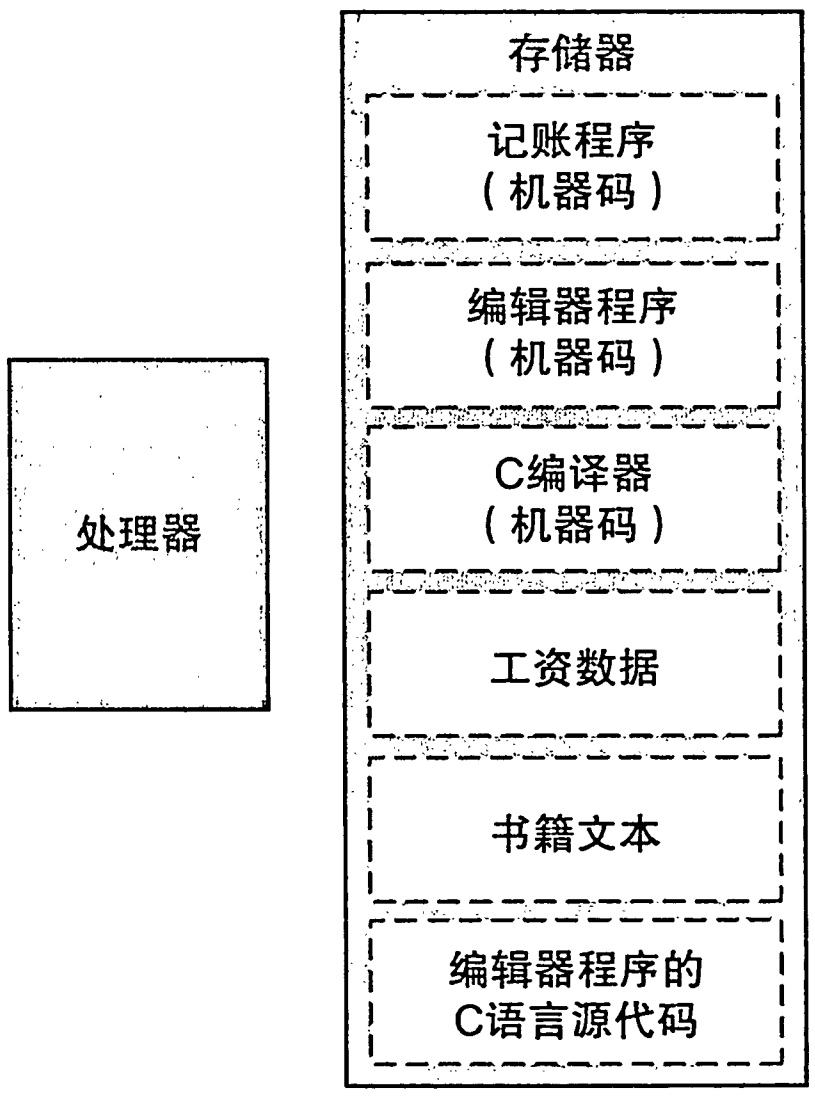


图 2-7　存储程序概念。存储程序可以让一台执行记账的计算机在眨眼之间变成帮助作者写作的计算机。只需将程序和数据加载到存储器，然后告诉计算机在存储器中的给定位置开始执行，就可以实现这种功能切换。采用与处理数据相同的方式处理指令极大地简化了存储器硬件和计算机系统软件。具体来说，针对数据的存储器技术也可以用于程序，例如编译器等程序可以将方便人类使用的符号编码翻译成计算机可以理解的代码

将指令作为数据的一个结果就是程序经常以二进制数据文件的形式来发布。商业上的意义是计算机可以继承（已有的）与指令系统体系结构兼容的软件。这种“二进制兼容性”通常会导致行业围绕少数几个指令系统体系结构形成联盟。

自我检测　下图代表的是哪条 RISC-V 指令？请从四个选项中选出最正确的一项。

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	000	11	51

1. `sub x9, x10, x11`
2. `add x11, x9, x10`
3. `sub x11, x10, x9`
4. `sub x11, x9, x10`

2.6　逻辑操作

尽管最初计算机只对整字进行操作，但人们很快发现，在一个字内对几个位构成的字段甚至是对单个位进行操作都是十分有用的。检查一个字中每个由 8 位组成的字符就是一个例子（见 2.9 节）。随之而来的是，人们在编程语言和指令系统体系结构中添加了一些操作，用于简化打包或者拆包。这些指令被称为逻辑操作。图 2-8 显示了 C、Java 和 RISC-V 中的逻辑运算。

> “正相反，”叮当弟接着说，“如果那是真的，那就是真的；如果那曾经是真的，它就是曾经为真过；但是既然现在它不是真的，那么现在它就是假的。这就是逻辑。”
> *Lewis Carroll，爱丽丝漫游仙境，1865*

逻辑操作	C操作符	Java操作符	RISC-V指令
左移	<<	<<	sll, slli
右移	>>	>>>	srl, srli
算术右移	>>	>>	sra, srai
按位与	&	&	and, andi
按位或	|	|	or, ori
按位异或	^	^	xor, xori
按位取反	~	~	xori

图 2-8　C 和 Java 中的逻辑操作符及其对应的 RISC-V 指令。实现 NOT 的一种方法是使用 XOR，其中一个操作数为全 1（FFFF FFFF FFFF $FFFF_{16}$）

第一类操作称为移位。一个双字中的所有位都向左或向右移，用 0 填充空出来的位。例如，如果寄存器 x19 中的值为：

00000000 00000000 00000000 00000000 00000000 00000000 00000000 $00001001_2 = 9_{10}$

并且指令让它左移 4 位，那么新的值为：

00000000 00000000 00000000 00000000 00000000 00000000 00000000 $10010000_2 = 144_{10}$

对应于左移的是右移。这两条 RISC-V 移位指令的实际名称是*左移逻辑立即数*（slli）和*右移逻辑立即数*（srli）。假设初始值位于寄存器 x19 中且结果应存入寄存器 x11，则以下指令执行上述操作：

```
slli x11, x19, 4 // reg x11 = reg x19 << 4 bits
```

这些移位指令使用 I 型格式。因为它不适用于对一个 64 位寄存器移动大于 63 位的操作，只有 I 型格式中 12 位的 immediate 字段中的低 6 位被实际使用。其余的 6 位被重新用作额外的操作码字段，即 funct6。

funct6	immediate	rs1	funct3	rd	opcode
0	4	19	1	11	19

slli 的编码在 opcode 字段为 19，rd 字段为 11，funct3 字段为 1，rs1 字段为 19，immediate 字段为 4，funct6 字段为 0。

逻辑左移提供了另外一个好处。左移 i 位相当于乘以 2^i，就像是十进制数左移 i 位相当于乘以 10^i 一样。例如，上述 slli 左移 4 位，相当于乘以 16（2^4）。上面的第一个二进制[⊖]表示 9，并且 $9 \times 16 = 144$，刚好是第二个二进制的值。RISC-V 提供了第三种类型的移位指令——*算术右移*（srai）。这个变体与 srli 很相似，但它不是用零填充空出的左边的位，而是用原来的符号位来填充。它还提供了三个移位操作的变体：sll、srl 和 sra。它们从寄存器中取出移位的位数，而不是从立即数中。

另外一个有用的操作是**与**（AND）。（我们把这个词大写来避免操作和英文连接词之间的混淆。）AND 是按位操作的，只有当两个操作数的位都是 1 时，结果才是 1。例如，如果寄存器 x11 中的值为：

00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000_2

寄存器 x10 的值为：

⊖ 寄存器 x19 中的值。——译者注

00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000$_2$

那么，在执行 RISC-V 指令之后：

```
and x9, x10, x11      // reg x9 = reg x10 & reg x11
```

寄存器 x9 的值为：

00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000$_2$

如你所见，AND 可以在源操作数的某些位为 0 时，将结果数的对应位设为 0。这种和 AND 联合使用的源操作数习惯上被称为*掩码*，因为掩码隐藏了某些位。

为了将一个值放入到一大堆 0 当中，存在一个与 AND 相对应的操作，称为**或**（OR）。这是一个按位操作，如果任一操作数的位为 1，则结果的对应位为 1。为了详细说明，如果寄存器 x10 和 x11 中的值与上例一样，则 RISC-V 指令的结果：

AND：两个操作数的逻辑按位操作，只有在两个操作数中的对应位都是 1 时，结果的对应位才是 1。

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

OR：两个操作数的逻辑按位操作，如果两个操作数中的对应位有一个为 1，则结果的对应位为 1。

是寄存器 x9 中的这个值：

00000000 00000000 00000000 00000000 00000000 00000000
00111101 11000000$_2$

最后一个逻辑操作是**按位取反**（NOT）。NOT 只有一个操作数，如果操作数中的某位为 0，那么它将结果的对应位设为 0，反之亦然。使用我们以前的符号，它计算 $\bar{x}$。

NOT：一个操作数的逻辑按位取反操作，也即是说，把每个 1 替换为 0，把每个 0 替换为 1。

为了保持三操作数格式，RISC-V 的设计者决定引入指令 XOR（**异或**）来取代 NOT。因为异或是在两个操作数对应位相同时设 0，不同时设 1，所以 NOT 等价于异或 111...111。

XOR：两个操作数的逻辑按位操作，用于计算两个操作数的异或。也就是说，只有两个操作数的对应位不同时，它才会计算为 1。

如果寄存器 x10 的值和上例一样，寄存器 x12 的值为 0，则 RISC-V 指令的结果：

```
xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12
```

是寄存器 x9 中的这个值：

00000000 00000000 00000000 00000000 00000000 00000000 00110001 11000000$_2$

上面的图 2-8 显示了 C 和 Java 操作符与 RISC-V 指令之间的关系。常量在逻辑运算和算术运算中都很有用，所以 RISC-V 还提供了*立即数与*（andi）、*立即数或*（ori）和*立即数异或*（xori）。

详细阐述 C 允许在双字中定义“位字段”或“字段”，它们都允许将对象打包在双字中，并匹配外部强制接口（如 I/O 设备）。所有字段必须适用于单个双字。字段是无符号整数并可以短至 1 位。C 编译器使用 RISC-V 中的逻辑指令来插入和提取字段：andi、ori、slli 和 srli。

自我检测　下面哪个操作可以从一个双字中分离出一个字段？

1. AND
2. 左移之后进行右移

2.7 用于决策的指令

计算机与简单计算器的区别在于它的决策能力。根据输入数据和计算中产生的值执行不同的指令。在编程语言中，通常使用 if 语句来表示决策，有时 if 语句也会和 go to 语句以及标签结合使用。RISC-V 汇编语言包含两个决策类指令，类似于带 go to 的 if 语句。第一条指令是：

```
beq rs1, rs2, L1
```

该指令表示如果寄存器 rs1 中的值等于寄存器 rs2 中的值，则转到标签为 L1 的语句执行。助记符 beq 代表相等则分支。第二条指令是：

```
bne rs1, rs2, L1
```

该指令表示如果寄存器 rs1 中的值不等于寄存器 rs2 中的值，则转到标签为 L1 的语句执行。助记符 bne 代表不等则分支。这两条指令通常称作**条件分支**指令。

自动计算机的实用性在于重复使用给定的指令序列的可能性，其重复次数取决于计算结果。这种选择取决于数的符号（0 被机器认为是正数）。因此，我们引入了一条“指令”（条件传输“指令”），该指令将根据给定数的符号来从两条路径中选择正确的一个执行。

Burks、Goldstine 和 von Neumann，1947

条件分支指令：一条指令，先检测一个值，然后根据检测结果允许后续控制流转移到程序中的一个新地址。

例题 | 将 if-then-else 语句编译为条件分支指令

在下面的代码段中，f、g、h、i 和 j 是变量。如果五个变量 f 到 j 对应于 x19 到 x23 这 5 个寄存器，这个 C 语言的 if 语句编译后的 RISC-V 代码是什么？

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

答案 | 图 2-9 显示了 RISC-V 代码应该执行的操作的流程图。第一个表达式比较寄存器中两个变量是否相等。如果 i 和 j 相等则跳转，那么需要一条 beq 指令。一般来说，如果我们测试相反的条件来进行跳转，代码将更有效率，那么需要一条 bne 指令。代码如下：

```
bne x22, x23, Else   // go to Else if i ≠ j
```

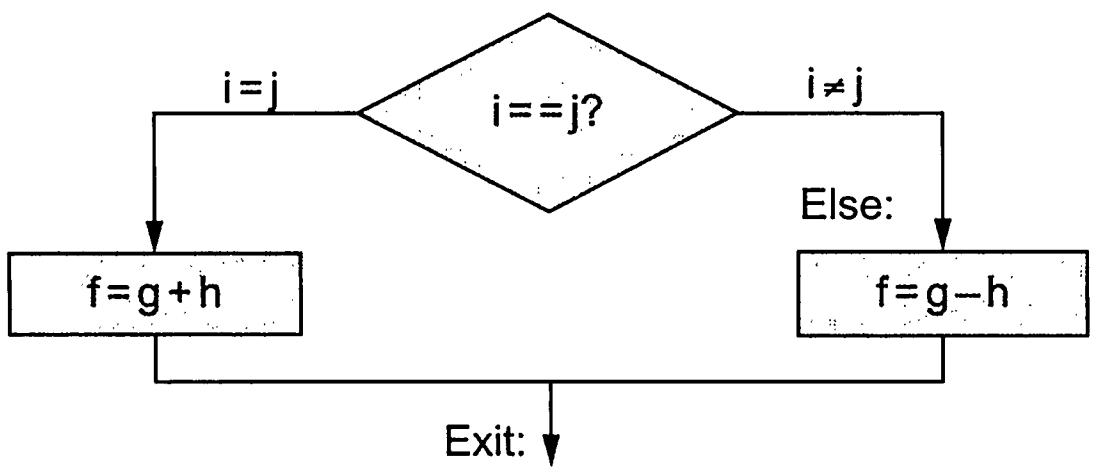


图 2-9 上述 if 语句的操作流程图。左边方框对应 if 语句的 then 部分，右边方框对应 else 部分

下一个赋值语句执行单个操作，并且如果所有操作数都分配给了寄存器，则它只是一条指令：

```
add x19, x20, x21        // f = g + h (skipped if i ≠ j)
```

现在看看 if 语句的结尾部分。本例引入了另一种分支，通常称为无条件分支。该指令表示在遇到该指令时，程序必须分支。在 RISC-V 中表达无条件分支的一种方法是使用条件始终为真的条件分支：

```
beq x0, x0, Exit         // if 0 == 0, go to Exit
```

if 语句的 else 部分中的赋值语句可以再次编译成单一指令。我们只需要在该指令中附加标签

Else。另外，在该指令之后还需 Exit 标签，表示 if-then-else 编译后的代码结束：

```
Else:sub x19, x20, x21    // f = g - h (skipped if i = j)
Exit:
```

注意，就像汇编器计算 load 和 store 的数据地址一样，它也会计算分支目标地址，这缓解了编译器和汇编语言程序员计算分支地址的枯燥工作（见 2.12 节）。

硬件/软件接口 编译器经常产生分支和标签，但它们在编程语言中并不出现。使用高级编程语言编程的好处之一是可以避免编写显式标签和分支，这也是其编码速度更快的原因。

2.7.1 循环

对于二选一的 if 语句和循环中的迭代计算，决策都是十分重要的。而且在这两种情况下，汇编语言指令都是相同的。

例题｜编译一个 C 语言的 while 循环

下面是一个 C 语言的常见循环：

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

假设 i 和 k 对应于寄存器 x22 和 x24，数组的基址保存在 x25 中。与此 C 语言代码相对应的 RISC-V 汇编代码是什么？

答案｜ 第一步是将 save[i] 加载到临时寄存器中。在将 save[i] 加载到临时寄存器之前，需要得到它的地址。在将 i 加到数组 save 的基址以形成地址之前，由于字节寻址问题，必须将索引 i 乘以 8。幸运的是，我们可以使用左移，因为左移 3 位相当于乘以 2^3 或 8（参见上一节）。我们需要给指令添加标签 Loop，以便我们可以在循环结尾跳转回该指令：

```
Loop: slli x10, x22, 3     // Temp reg x10 = i * 8
```

要获得 save[i] 的地址，我们需要将 x10 和保存在 x25 中的基址相加：

```
add x10, x10, x25       // x10 = address of save[i]
```

现在我们可以利用这个地址将 save[i] 加载到临时寄存器了：

```
ld x9, 0(x10)       // Temp reg x9 = save[i]
```

下一条指令将进行循环判断，如果 save[i] ≠ k 则退出：

```
bne x9, x24, Exit     // go to Exit if save[i] ≠ k
```

接下来的指令将 i 加 1：

```
addi x22, x22, 1       // i = i + 1
```

循环的结尾回到循环顶部的 while 判断。我们只需在它后面添加 Exit 标签：

```
beq x0, x0, Loop       // go to Loop
Exit:
```

（见自我检测中对该指令序列的优化。）

硬件/软件接口 这种以分支结尾的指令序列对于编译来说非常基础，它们有专门的术语：基本块。基本块指的是指令序列：除了在指令序列的结尾，序列中没有分支；以及除了在序列起始处，序列中没有分支目标和分支标签。编译的基础工作之一就是将程序划分为基本块。

基本块：一个没有分支的指令序列（除了可能在结尾处），同时没有分支目标或分支标签（除了可能在起始处）。

对相等或不相等的判断可能是最常见的判断，但也有很多其他两个数之间的关系。例如，for 循环可能需要判断下标变量是否小于 0。完整的相互关系有小于（<）、小于等于（≤）、大于（>）、大于等于（≥）、相等（=）、不等于（≠）。

位模式的比较还必须处理有符号和无符号数之间的差别。有时候，最高有效位是 1 代表一个负数，当然，它小于任何正数（最高有效位是 0）。另一方面，对于无符号整数，最高有效位是 1 表示大于任何最高有效位是 0 的数。（我们很快将利用最高有效位的这种双重含义来降低数组边界检查的成本。）RISC- V 提供了指令来处理这两种情况。这些指令与 beq 和 bne 具有相同的形式，但是执行不同的比较。小于则分支指令（blt）比较寄存器 rs1 和 rs2 中的值（采用二进制补码表示），如果 rs1 中的值较小则跳转。大于等于分支（bge）指令是相反情况，也就是说，如果 rs1 中的值至少不小于 rs2 中的值则跳转。无符号的小于则分支指令（bltu）意味着，如果二者是无符号数，那么 rs1 中的值小于 rs2 中的值则跳转。最后，无符号数的大于等于则分支指令（bgeu）在相反的情况下跳转。

另一种提供这些额外分支指令的方法是根据比较结果设置寄存器，然后使用 beq 或 bne 指令根据该临时寄存器中的值来进行分支判断。这种由 MIPS 指令系统使用的方法可以使处理器数据通路稍微简单一些，但它需要更多指令来表达程序。

ARM 指令系统使用的另一种方法是，保留额外的位来记录指令执行期间发生的情况。这些额外的位称为条件代码或标志位，用于表明例如算术运算的结果是否为负数或零，或溢出。

条件分支利用这些条件代码的组合来执行期望的判断。

条件代码的一个缺点是，如果许多指令总是设置它们，则会生成让流水线执行困难的依赖关系（参见第 4 章）。

2.7.2 边界检查的简便方法

将有符号数当作无符号数处理，给我们提供了一种低成本的方式检查是否 $0 \leqslant x < y$，常用于检测数组下标是否越界。关键在于二进制补码表示中的负整数看起来像无符号表示中很大的数；因为最高有效位在有符号数中表示符号位，但在无符号数中表示数的很大一部分。因此，无符号比较 $x<y$ 在检测 x 是否小于 y 的同时，也检测了 x 是否为负数。

例题

利用该简便方法可以降低下标越界检查的开销：如果 x20 ≥ x11 或 x20 是负数则跳转到 IndexOutOfBounds。

答案 检查代码仅使用无符号数的大于或等于来进行两项检查：

```
bgeu x20, x11, IndexOutOfBounds // if x20 >= x11 or
    x20 < 0, goto IndexOutOfBounds
```