得多。而且，一旦编码，接下来的危险就是它将变为一个流行程序。这种程序的寿命总是比预期的长，意味着程序员不得不几年就更新一次代码以使其适用于新版本的操作系统和最新的计算机。使用更高级语言而不是汇编语言编写不仅允许未来的编译器为将来的机器定制代码，还使软件更易于维护，并允许程序在更多品牌的计算机上运行。

谬误：商用计算机二进制兼容的重要性意味着成功的指令系统无须改变。

虽然向后的二进制兼容性是神圣不可侵犯的，但图 2-39 显示了 x86 指令系统的快速发展。在其 35 年的生命周期中，平均每月至少新增一条指令！

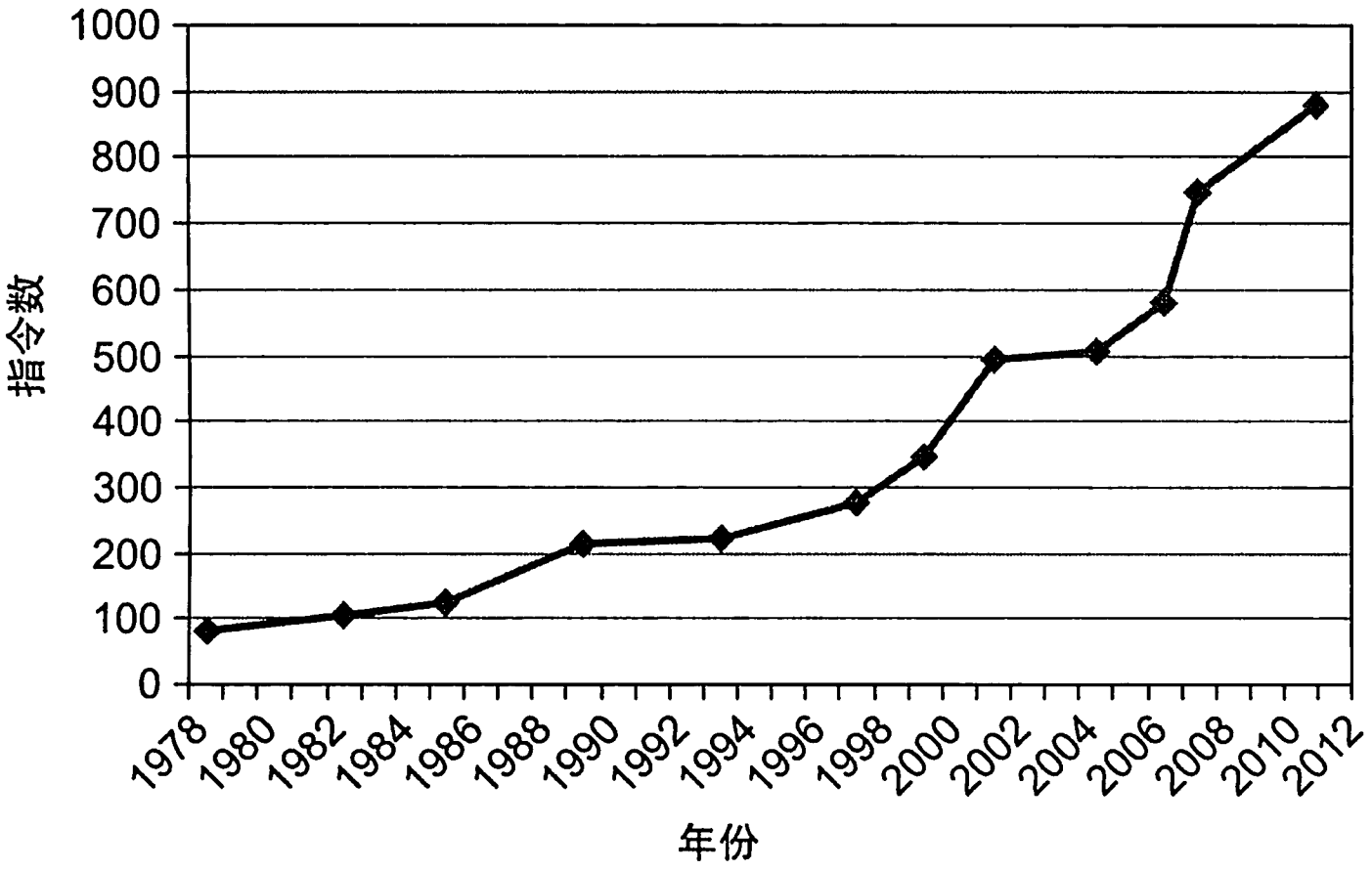


图 2-39 x86 指令系统随时间推移的增长。虽然其中一些扩展具有明显的技术价值，但这种快速变化也增加了其他公司试图构建兼容处理器的难度

陷阱：忘记在字节寻址的机器中，连续的字或双字地址相差不为 1。

许多汇编语言程序员假设将寄存器中的地址递增 1 而不是字或双字大小（以字节为单位）来找到下一个字或双字的地址，他们因此而烦恼。提前预备以避后患！

陷阱：在变量的定义过程外，使用一个指针指向该变量。

处理指针的一个常见错误是从一个过程传递结果，而这个过程包含指向其局部数组的指针。遵循图 2-12 中的栈规则，一旦过程返回，包含局部数组的存储器将被重用。指向变量的指针可能导致混乱。

2.20 本章小结

> 少即多。
> *Robert Browning,Andrea del Sarto, 1855*

存储程序（stored-program）计算机的两个原理是，对于程序来说，指令的使用与数据没有区别，以及都使用可变的存储器。这些原则允许单个机器在不同的专业领域辅助癌症研究人员、经济顾问和小说家。选择机器能够理解的指令系统时，需要全面平衡执行程序所需的指令数目、指令所需的时钟周期数以及时钟速度。正如本章所述，三条设计原则指导指令系统设计者如何权衡：

1. 简单源于规整。规整性使 RISC-V 指令系统具有很多特点：所有指令都保持单一长度、算术指令中总是使用寄存器作为操作数、所有指令格式中寄存器字段都保持在相同位置。

2. 更少则更快。对速度的要求导致 RISC-V 有 32 个寄存器而不是更多。

3. 优秀的设计需要适当的折中。一个 RISC-V 的例子是，在指令中提供更大的地址和常数，与保持所有的指令具有相同的长度之间的折中。

在第1章中已经看到如何将加速经常性事件的伟大思想应用于指令系统以及计算机体系结构。加速RISC-V经常性事件的例子，包括条件分支指令的PC相对寻址和较大常量操作数的立即数寻址。

在机器之上的是人类可读的汇编语言。汇编器将其翻译为机器可以理解的二进制数，甚至通过创建硬件中没有的符号指令来“扩展”指令系统。例如，太大的常量或地址被分成大小合适的块，常见的指令变体都有它们自己的名字，等等。图2-40列出了到目前为止所涉及的RISC-V指令，包括实际指令和伪指令。隐藏更高层次的细节是抽象概念的另一个例子。

RISC-V指令	名称	格式	RISC-V伪指令	名称	真实指令
Add	add	R	Move	mv	addi
Subtract	sub	R	Load immediate	li	addi
Add immediate	addi	I	Jump	j	jal
Load doubleword	ld	I	Load address	la	lui+addi
Store doubleword	sd	S			
Load word	lw	I			
Load word, unsigned	lwu	I			
Store word	sw	S			
Load halfword	lh	I			
Load halfword, unsigned	lhu	I			
Store halfword	sh	S			
Load byte	lb	I			
Load byte, unsigned	lbu	I			
Store byte	sb	S			
Load reserved	lr.d	R			
Store conditional	sc.d	R			
Load upper immediate	lui	U			
And	and	R			
Inclusive or	or	R			
Exclusive or	xor	R			
And immediate	andi	I			
Inclusive or immediate	ori	I			
Exclusive or immediate	xori	I			
Shift left logical	sll	R			
Shift right logical	srl	R			
Shift right arithmetic	sra	R			
Shift left logical immediate	slli	I			
Shift right logical immediate	srli	I			
Shift right arithmetic immediate	srai	I			
Branch if equal	beq	SB			
Branch if not equal	bne	SB			
Branch if less than	blt	SB			
Branch if greater or equal	bge	SB			
Branch if less, unsigned	bltu	SB			
Branch if greatr/eq, unsigned	bgeu	SB			
Jump and link	jal	UJ			
Jump and link register	jalr	I			

图2-40　到目前为止涉及的RISC-V指令系统，左侧是真实的RISC-V指令，右侧是伪指令。图2-1展示了本章介绍的RISC-V体系结构的更多细节。这里给出的信息可以在本书RISC-V参考数据卡的第1列和第2列中找到

每种类型的 RISC-V 指令都与编程语言中出现的结构相关：

- 算术指令对应于赋值语句中的操作。
- 传输指令最有可能发生在处理数组或结构体等数据结构时。
- 条件分支用于 if 语句和循环中。
- 无条件分支用于过程调用和返回，以及 case/switch 语句。

这些指令出现频率不相等，少数常用指令在大多数指令中占主导地位。例如，图 2-41 展示了 SPEC CPU2006 的每类指令的出现频率。指令的不同出现频率在数据路径、控制和流水线的章节中起着重要作用。

指令类别	RISC-V示例	对应的高级语言	频率	
			整数	浮点
算术	add, sub, addi	赋值语句中的操作	16%	48%
数据传输	ld, sd, lw, sw, lh, sh, lb, sb, lui	对存储器中数据结构的引用	35%	36%
逻辑	and, or, xor, sll, srl, sra	赋值语句中的操作	12%	4%
分支	beq, bne, blt, bge, bltu, bgeu	if语句；循环	34%	8%
跳转	jal, jalr	过程调用&返回；switch语句	2%	0%

图 2-41 RISC-V 指令类别、示例、相应的高级程序语言结构，以及 RISC-V 指令按平均整数和浮点数在 SPEC CPU2006 基准测试上分类执行的百分比。图 3-24 显示了执行的各个 RISC-V 指令的平均百分比

在第 3 章解释计算机运算之后，将继续揭示 RISC-V 指令系统体系结构。

2.21 历史视角和扩展阅读

本节将逐步介绍指令系统体系结构（Instruction Set Architecture，ISA）的历史，并简要介绍编程语言和编译器。ISA 包括累加器体系结构、通用寄存器体系结构、栈体系结构以及 x86 和 ARM 的 32 位体系结构 ARMv7 的简短历史。也回顾了高级语言计算机体系结构的争议问题和精简指令系统计算机体系结构。编程语言的历史包括 Fortran、Lisp、Algol、C、Cobol、Pascal、Simula、Smalltalk、C++ 和 Java，编译器的历史包含其重要里程碑和实现它们的先驱。2.21 节的剩余部分可在线查找。

2.22 练习

2.1 [5] <2.2> 对于以下 C 语句，请编写相应的 RISC-V 汇编代码。假设 C 变量 f、g 和 h 已经分别放于寄存器 x5、x6 和 x7 中。使用最少数量的 RISC-V 汇编指令。

```
f = g + (h - 5);
```

2.2 [5] <2.2> 编写一条 C 语句，对应以下两条 RISC-V 汇编指令。

```
add f, g, h
add f, i, f
```

2.3 [5] <2.2, 2.3> 对于以下 C 语句，请编写相应的 RISC-V 汇编代码。假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
B[8] = A[i-j];
```

2.4 [10] <2.2, 2.3> 对于以下 RISC-V 汇编指令，相应的 C 语句是什么？假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
slli x30, x5, 3      // x30 = f*8
add  x30, x10, x30   // x30 = &A[f]
slli x31, x6, 3      // x31 = g*8
add  x31, x11, x31   // x31 = &B[g]
ld   x5, 0(x30)      // f = A[f]

addi x12, x30, 8
ld   x30, 0(x12)
add  x30, x30, x5
sd   x30, 0(x31)
```

2.5 [5] <2.3> 分别给出值 0xabcdef12 在小端对齐和大端对齐机器的存储器中的排列。假设数据从地址 0 开始存储，字长为 4 个字节。

2.6 [5] <2.4> 将 0xabcdef12 转换成十进制。

2.7 [5] <2.2, 2.3> 将以下 C 代码转换成 RISC-V 指令。假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。假设数组 A 和 B 的元素是一个字为 8 字节：

```
B[8] = A[i] + A[j];
```

2.8 [10] <2.2, 2.3> 将以下 RISC-V 指令转换成 C 代码。假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
addi x30, x10, 8
addi x31, x10, 0
sd   x31, 0(x30)
ld   x30, 0(x30)
add  x5,  x30, x31
```

2.9 [20] <2.2, 2.5> 对于练习 2.8 中的每条 RISC-V 指令，写出操作码（op）、源寄存器（rs1）和目标寄存器（rd）字段的值。对于 I 型指令，写出立即数字段的值，对于 R 型指令，写出第二个源寄存器（rs2）的值。对于非 U 型和 UJ 型指令，写出 funct3 字段，对于 R 型和 S 型指令，写出 funct7 字段。

2.10 假设寄存器 x5 和 x6 分别保存值 0x8000000000000000 和 0xD000000000000000。

2.10.1 [5] <2.4> 以下汇编代码中 x30 的值是多少？

```
add x30, x5, x6
```

2.10.2 [5] <2.4> x30 中的结果是否为预期结果，或者是否溢出？

2.10.3 [5] <2.4> 对于上面指定的寄存器 x5 和 x6 的内容，以下汇编代码中 x30 的值是多少？

```
sub x30, x5, x6
```

2.10.4 [5] <2.4> x30 中的结果是否为预期结果，或者是否溢出？

2.10.5 [5] <2.4> 对于上面指定的寄存器 x5 和 x6 的内容，以下汇编代码中 x30 的值是多少？

```
add x30, x5, x6
add x30, x30, x5
```

2.10.6 [5] <2.4> x30 中的结果是否为预期结果，或者是否溢出？

2.11 假设寄存器 x5 保存值 128_{10}。

2.11.1 [5] <2.4> 对于指令 add x30, x5, x6，求出导致结果溢出的 x6 值的范围？

2.11.2 [5] <2.4> 对于指令 sub x30, x5, x6，求出导致结果溢出的 x6 值的范围？

2.11.3 [5] <2.4> 对于指令 sub x30, x6, x5，求出导致结果溢出的 x6 值的范围？

2.12 [5] <2.2, 2.5> 写出以下二进制值对应的指令类型和汇编语言指令：

0000 0000 0001 0000 1000 0000 1011 0011_2

提示：图 2-20 可能有所帮助。

2.13 [5] <2.2, 2.5> 写出以下指令的指令类型和十六进制表示：

```
sd x5, 32(x30)
```

2.14 [5] <2.5> 写出以下 RISC-V 字段描述的指令的指令类、汇编语言指令以及二进制表示：

opcode=0x33, funct3=0x0, funct7=0x20, rs2=5, rs1=7, rd=6

2.15 [5] <2.5> 写出以下 RISC-V 字段描述的指令的指令类型、汇编语言指令以及二进制表示：

opcode=0x3, funct3=0x3, rs1=27, rd=3, imm=0x4

2.16 假设希望将 RISC-V 寄存器堆扩展为 128 个寄存器，并将指令系统扩展为原来的四倍。

2.16.1 [5] <2.5> 这将如何影响 R 型指令中每个字段的大小？

2.16.2 [5] <2.5> 这将如何影响 I 型指令中每个字段的大小？

2.16.3 [5] <2.5, 2.8, 2.10> 提出的两种变化中，每种变化如何减小 RISC-V 汇编程序的大小？另一方面，变化如何增加 RISC-V 汇编程序的大小？

2.17 假设有如下寄存器内容：

```
x5 = 0x00000000AAAAAAAA, x6 = 0x1234567812345678
```

2.17.1 [5] <2.6> 对于上面显示的寄存器值，以下指令序列中 x7 的值是多少？

```
slli x7, x5, 4
or   x7, x7, x6
```

2.17.2 [5] <2.6> 对于上面显示的寄存器值，以下指令序列中 x7 的值是多少？

```
slli x7, x6, 4
```

2.17.3 [5] <2.6> 对于上面显示的寄存器值，以下指令序列中 x7 的值是多少？

```
srli  x7, x5, 3
andi  x7, x7, 0xFEF
```

2.18 [10] <2.6> 找到完成以下功能的最短 RISC-V 指令序列：从寄存器 x5 中提取 16 位到 11 位，并使用该字段的值来替换寄存器 x6 中的 31 位到 26 位而不改变其他位。（务必使用 x5 = 0 和 x6 = 0xffffffffffffffff 来测试代码。）

2.19 [5] <2.6> 写出可用于实现以下伪指令的 RISC-V 指令系统的最小子集：

```
not x5, x6      // bit-wise invert
```

2.20 [5] <2.6> 对于以下 C 语句，编写执行相同操作的最小 RISC-V 汇编指令序列。假设 x6 = A，且 x17 是 C 的基地址。

```
A = C[0] << 4;
```

2.21 [5] <2.7> 假设 x5 保存值 0x0000000001010000。以下指令完成后 x6 的值是多少？

```
        bge x5, x0, ELSE
        jal x0, DONE
ELSE:   ori x6, x0, 2
DONE:
```

2.22 假设程序计数器（PC）置为 0x20000000。

2.22.1 [5] <2.10> 使用 RISC-V 跳转 – 链接（jal）指令可以到达的地址范围是什么？（换句话说，跳转指令执行后 PC 的可能值是多少？）

2.22.2 [5] <2.10> 使用 RISC-V 的相等则分支（beq）指令可以到达的地址范围是什么？（换句话说，分支指令执行后 PC 的可能值是多少？）

2.23 考虑一条名为 rpt 的新指令。该指令将循环的条件检查和计数器递减组合成单条指令。例如 rpt x29, loop 将执行以下操作：

```
if (x29 > 0) {
        x29 = x29 -1;
        goto loop
    }
```

2.23.1 [5] <2.7, 2.10> 如果要将该指令添加到 RISC-V 指令系统，那么最合适的指令格式是什么？

2.23.2 [5] <2.7> 执行相同操作的 RISC-V 指令的最短序列是什么？

2.24 考虑以下 RISC-V 循环：

```
LOOP:  beq  x6, x0, DONE
       addi x6, x6, -1
       addi x5, x5, 2
       jal  x0, LOOP
DONE:
```

2.24.1 [5] <2.7> 假设寄存器 x6 初始化为 10。寄存器 x5 的最终值是多少（假设 x5 初始值为零）？

2.24.2 [5] <2.7> 对于上面的循环，编写等效的 C 代码。假设寄存器 x5 和 x6 分别是整型 acc 和 i。

2.24.3 [5] <2.7> 对于上面用 RISC-V 汇编语言编写的循环，假设寄存器 x6 初始化为 N。总共执行了多少条 RISC-V 指令？

2.24.4 [5] <2.7> 对于上面用 RISC-V 汇编语言编写的循环，将指令“beq x6, x0, DONE”替换为“blt x6, x0, DONE”指令并写出等效的 C 代码。

2.25 [10] <2.7> 将以下 C 代码转换成 RISC-V 汇编代码。使用最少数量的指令。假设 a、b、i 和 j 的值分别在寄存器 x5、x6、x7 和 x29 中。另外，假设寄存器 x10 保存数组 D 的基址。

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

2.26 [5] <2.7> 实现练习 2.25 中的 C 代码需要多少条 RISC-V 指令？如果变量 a 和 b 初始化为 10 和 1 且数组 D 的所有元素初始值都为 0，那么为完成循环执行的 RISC-V 指令总数是多少？

2.27 [5] <2.7> 将以下循环转换为 C 代码。假设 C 语言级的整数 i 保存在寄存器 x5 中，x6 保存名为 result 的 C 语言级的整数，x10 保存整数 MemArray 的基址。

```
        addi x6, x0, 0
        addi x29, x0, 100
LOOP:   ld   x7, 0(x10)
        add  x5, x5, x7
        addi x10, x10, 8
        addi x6, x6, 1
        blt  x6, x29, LOOP
```

2.28 [10] <2.7> 重写练习 2.27 中的循环以减少执行的 RISC-V 指令数。提示：注意变量 i 仅用于循环控制。

2.29 [30] <2.8> 用 RISC-V 汇编语言实现以下 C 代码。提示：记住栈指针必须保持 16 位对齐。

```
int fib(int n){
    if (n==0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

2.30 [20] <2.8> 对于练习 2.29 中的每个函数调用，写出函数调用后栈的内容。假设栈指针最初位于地址 0x7ffffffc，且遵循图 2-11 中指定的寄存器约定。

2.31 [20] <2.8> 将函数 f 转换为 RISC-V 汇编语言。假设 g 的函数声明是 int g(int a,int b)。函数 f 的代码如下：

```
int f(int a, int b, int c, int d){
  return g(g(a,b), c+d);
}
```

2.32 [5] <2.8> 可以在这个函数中使用尾调用优化吗？如果不能，请解释原因。如果能，在有和没有优化的情况下，f 执行的指令数有何不同？

2.33 [5] <2.8> 在练习 2.31 的函数 f 返回之前，寄存器 x10 ~ x14、x8、x1 和 sp 的内容是什么？记住，函数 f 的整体是已知的，但只知道函数 g 的声明。

2.34 [30] <2.9> 用 RISC-V 汇编语言编写一个程序，将包含十进制正整数或负整数字符串的 ASCII 字符串转换为整数。要求程序用寄存器 x10 保存一个以空字符结尾的字符串的地址，该字符串包含一个可选的“+”或“-”，后跟一些 0 到 9 的数字组合。要求程序计算与这个数字字符串相等的整数值，然后将数字放入寄存器 x10 中。如果字符串中的任何位置出现非数字字符，则程序停止并将值 -1 存入寄存器 x10 中。例如，如果寄存器 x10 指向三个字节 50_{10}、52_{10}、0_{10}（以空字符结尾的字符串“24”）的序列，则当程序停止时，寄存器 x10 应包含值 24_{10}。RISC-V 的 mul 指令将两个寄存器作为输入。没有“muli”指令。因此，只需将常数 10 存储在寄存器中。

2.35 考虑以下代码：

```
lb x6, 0(x7)
sd x6, 8(x7)
```

假设寄存器 x7 包含地址 0x10000000，且地址中的数据是 0x1122334455667788。

2.35.1 [5] <2.3, 2.9> 在大端对齐的机器上 0×10000008 中存储的是什么值？

2.35.2 [5] <2.3, 2.9> 在小端对齐的机器上 0×10000008 中存储的是什么值？

2.36 [5] <2.10> 写出产生 64 位常量 0x1122334455667882 的 RISC-V 汇编代码，并将该值存储到寄存器 x10 中。

2.37 [10] <2.11> 编写 RISC-V 汇编代码，使用 lr.d/sc.d 指令将以下 C 代码实现为原子“set max”操作。这里，参数 shvar 包含共享变量的地址，如果 x 大于它指向的值，则应该用 x 替换：

```
void setmax(int* shvar, int x) {
  // Begin critical section
```

```
    if (x > *shvar)
        *shvar = x;
    // End critical section}
}
```

2.38 [5] <2.11> 以练习 2.37 中编写的代码为例，说明当两个处理器同时开始执行该关键部分时会发生什么，假设每个处理器每周期只执行一条指令。

2.39 假设给定处理器算术指令的 CPI 为 1，加载 / 存储指令的 CPI 为 10，分支指令的 CPI 为 3。假设程序有 5 亿条算术指令、3 亿条加载 / 存储指令和 1 亿条分支指令。

2.39.1 [5] <1.6, 2.13> 假设将新的、更强大的算术指令添加到指令系统中。平均而言，通过使用这些更强大的算术指令，可以将执行程序所需的算术指令数减少 25%，同时时钟周期时间仅增加 10%。这是一个好的设计选择吗？为什么？

2.39.2 [5] <1.6, 2.13> 假设有一种让算术指令性能加倍的方法。机器的整体加速是多少？如果有一种方法可以将算术指令的性能提高 10 倍，机器性能的整体加速又是多少？

2.40 假设对于一个给定程序，70% 的执行指令是算术指令，10% 是加载 / 存储指令，20% 是分支指令。

2.40.1 [5] <1.6, 2.13> 假设算术指令需要 2 周期，加载 / 存储指令需要 6 周期，而一条分支指令需要 3 周期，求平均 CPI。

2.40.2 [5] <1.6, 2.13> 对于性能提高 25%，如果加载 / 存储和分支指令都没有改进，一条算术指令平均需要多少周期？

2.40.3 [5] <1.6, 2.13> 对于性能提高 50%，如果加载 / 存储和分支指令都没有改进，一条算术指令平均需要多少周期？

2.41 [10] <2.19> 假设 RISC-V ISA 包含比例偏移寻址模式，类似于 2.17 节（图 2-35）中描述的 x86。描述如何使用比例偏移寻址模式加载来进一步减少执行练习 2.4 中给出的函数所需的汇编指令数。

2.42 [10] <2.19> 假设 RISC-V ISA 包含比例偏移寻址模式，类似于 2.17 节（图 2-35）中描述的 x86。描述如何使用比例偏移寻址模式加载来进一步减少实现练习 2.7 中给出的 C 代码所需的汇编指令数。

自我检测答案

2.2 节　RISC-V、C、Java

2.3 节　2. 非常慢。

2.4 节　2. -8_{10}

2.5 节　3. `sub x11, x10, x9`

2.6 节　都可以。1 掩码模式的“逻辑与”会导致想要的字段之外其他全为 0。正确的左移操作将左端字段的位都移走。合适的右移将字段放到双字最右端，剩余字段均为 0。注意：“逻辑与”操作会保留字段原有的值，移位操作将字段移到双字的最右边。

2.7 节　I. 全对。II. 1。

2.8 节　两个都对。

2.9 节　I. 1 和 2；II. 3。

2.10 节　I. 4. ±4K；II. 4. ±1M。

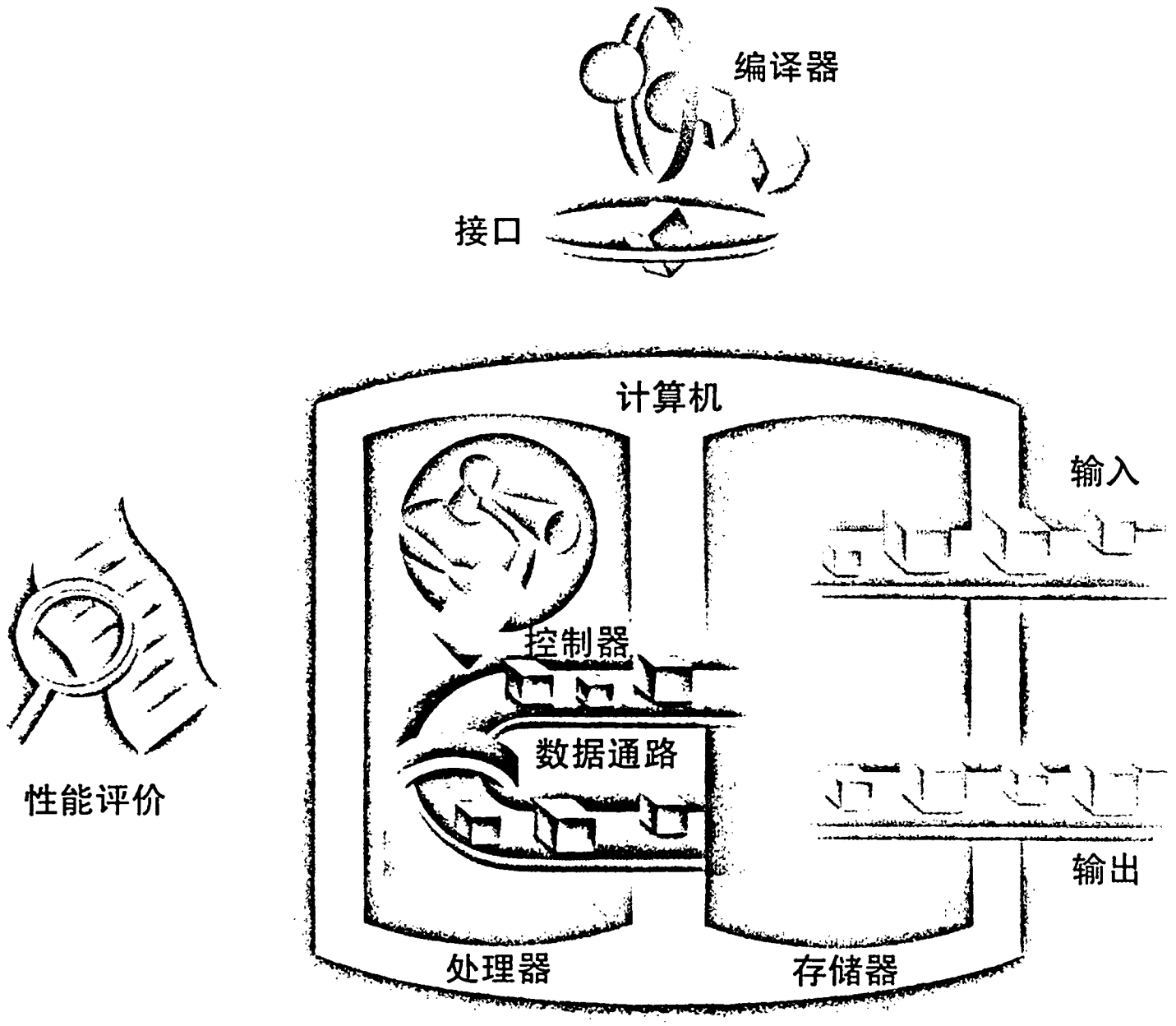
2.11 节　两个都对。

2.12 节　4. 机器无关。

第 3 章

Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition

计算机的算术运算



计算机的五个经典部件

3.1 引言

计算机的字由位组成，因此，字可以被表示为二进制数字。第 2 章说明了整数可以用十进制或二进制形式表示，但其他常用数字又该如何表示呢？例如：

数值精度才是科学的灵魂。
Sir D'arcy Wentworth Thompson, On Growth and Form, 1917

- 如何表示小数和其他实数？
- 如果运算产生了一个大到无法表示的数该如何处理？
- 这些问题中隐藏着一个谜：硬件如何真正实现乘法或除法？

本章的目标就是揭开这些奥秘，包括实数的表示、算术的算法、实现这些算法的硬件，以及所有这些在指令系统中的含义。这些知识也许就能解释你已经遇到过的计算机疑难了。另外，我们还将介绍如何使用这些知识来加速计算密集型程序的运行。

3.2 加法和减法

加法是计算机中的必备操作。数字从右到左逐位相加，并将进位传送到左侧的下一位数字，就和手动运算一样。减法也使用加法实现：相应的操作数被简单取反后再进行加法操作。

减法：加法机智的朋友。
No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., Book of Top Ten Lists, 1990

例题 | 二进制加法和减法

用二进制表示形式计算 6_{10} 加 7_{10}，再计算 7_{10} 减 6_{10}。

$$
\begin{array}{ll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_2 = 7_{10} \\
+ & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_2 = 6_{10} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101_2 = 13_{10}
\end{array}
$$

右边的 4 位发生了变化。图 3-1 展示了求和与进位，箭头表示进位是如何传递的。

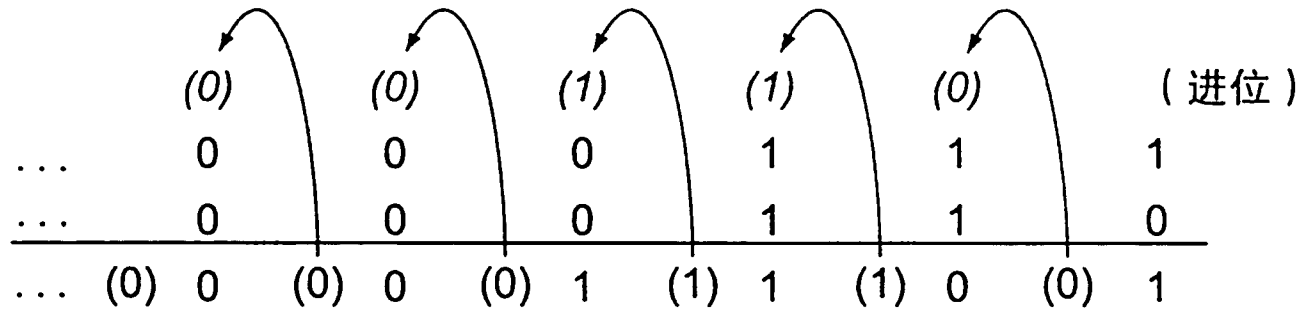


图 3-1　二进制加法，显示从右到左的进位。最右边一位将 0 和 1 相加，该位的结果为 1 且进位为 0。因此，右边第二位的操作为 0 + 1 + 1。求和产生结果 0 与进位 1。第三位是 1 + 1 + 1 求和，进位为 1 且和为 1。第四位是 1 + 0 + 0，和为 1 且无进位

答案 | 7_{10} 减去 6_{10} 可以直接算得：

$$
\begin{array}{ll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_2 = 7_{10} \\
- & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_2 = 6_{10} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_2 = 1_{10}
\end{array}
$$

或者通过使用 −6 的二进制补码表示来进行加法运算：

$$
\begin{array}{ll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_2 = 7_{10} \\
+ & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111010_2 = -6_{10} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_2 = 1_{10}
\end{array}
$$

回想一下，硬件规模总是有一定限制的，比如字宽为 64 位，当运算结果超过这个限制时，就会发生溢出。在加法中何时会发生溢出？当不同符号的操作数相加时，不会发生溢出，因为总和一定不会大于其中任意一个操作数。例如，−10+4=−6。由于操作数可以表示成 64 位且其总和不大于任一操作数，所以总和也一定能表示成 64 位。因此，当正负操作数相加时不会发生溢出。

在减法中也有类似的不会发生溢出的情况，但原理相反：当操作数的符号相同时，不会发生溢出。为了说明这一点，需要记住 c−a=c+（−a），这是因为我们通过将第二个操作数取反然后相加来实现减法。因此，当相同符号的操作数相减时，最终会变成相反符号的操作数相加。从上一段落可知，在这种情况下不会发生溢出。

知道加法和减法运算在什么时候不会发生溢出固然很好，但如何检测它何时发生呢？显然，加或减两个 64 位的数字可能产生一个需要 65 位才能表示的结果。

缺少第 65 位意味着当溢出发生时，符号位被结果的值占用而非结果的正确符号。由于溢出结果只可能多一位，所以只有符号位可能是错误的。因此，当两个正数相加但和为负数时，说明发生了溢出，反之亦然。这个假的和值意味着产生了向符号位的进位。

当正数减负数并得到负数结果，或者当负数减正数且结果为正数时，减法运算发生溢出。这种荒谬的结果意味着产生了从符号位的借位。图 3-2 展示了溢出发生时的运算、操作数和结果的各种组合情况。

操作	操作数A	操作数B	表明溢出的结果
A+B	≥0	≥0	<0
A+B	<0	<0	≥0
A−B	≥0	<0	<0
A−B	<0	≥0	≥0

图 3-2　加法和减法运算的溢出条件

刚刚看到了如何在计算机中检测二进制补码操作的溢出。那么无符号整数的溢出情况是怎样的呢？无符号整数通常用于表示忽略溢出的内存地址。

幸运的是，编译器可以轻松检查出使用分支指令的无符号溢出。如果总和小于加数中的任何一个，则加法溢出，而如果差大于被减数，则减法溢出。

算术逻辑单元：执行加法、减法和常见逻辑操作（如 AND 和 OR）的硬件。

附录 A 描述了执行加法和减法运算的硬件实现，被称为**算术逻辑单元**或 ALU。

硬件 / 软件接口　计算机设计者必须考虑如何处理算术溢出。尽管一些类似 C 和 Java 这样的语言会忽略整数溢出，但像 Ada 和 Fortran 这样的语言则需要告知程序溢出。当溢出发生时，程序员或编程环境必须决定该如何处理。

小结

本节主要指出，无论数的表示形式如何，计算机的有限字长意味着算术运算可能会产生过量而无法用这种固定字长表示的运算结果，即发生溢出。虽然无符号数的溢出容易检测，但无符号数通常使用自然数做地址运算，而程序通常不需要检测地址计算的溢出，所以这些溢出总被忽略。二进制补码（有符号数）对检测溢出提出了更大的挑战，但仍有一些软件系统需要识别溢出，因此今天所有的计算机都支持溢出检测。

详细阐述　在通用微处理器中不常见的一个特性是饱和操作。饱和（saturation）意味着当计算溢出时，结果设置为最大正数或最小负数，而不是像二进制补码运算那样采用取模计算。饱和操作可能更适用于多媒体操作。例如，收音机上的音量旋钮可能会令人懊恼，当你转动它时，音量会持续变大一段时间，然后又立即变得非常柔和。无论旋钮开到多大，具有饱和度的旋钮都会停在最高音量处。标准指令系统的多媒体扩展通常提供饱和计算。

详细阐述　加法的速度取决于向高位进位的计算速度。有多种方案可预测进位，最坏情况下进位时间是加法器位长的 $\log_2$ 的函数。这些预期信号速度更快，因为它们依次通过的门数更少，但需要更多的门来预测正确的进位。最常见的是超前进位加法器（carry lookahead），见附录 A 中的 A.6 节。

自我检测　一些编程语言允许对字节和半字的二进制补码进行整数运算，而 RISC-V 仅对整字进行整数算术运算。正如第 2 章所述，RISC-V 确实有字节和半字的数据传输操作。那么应该为字节和半字算术运算生成哪些 RISC-V 指令呢？

1. 用 lb、lh 进行取数；用 add、sub、mul、div 进行算术运算，在每次操作后用 and 将结果对齐到 8 位或 16 位；然后使用 sb、sh 进行存储。
2. 用 lb、lh 进行取数；用 add、sub、mul、div 进行算术运算；然后使用 sb、sh 进行存储。

3.3 乘法

现在我们已经完成了对加法和减法的解释，准备构建更复杂的乘法运算。

> 乘法令人烦恼，除法同样糟糕；比例法使我迷茫，而演算让我疯狂。
> *佚名，Elizabethan manuscript,1570*

首先让我们回顾一下在手工计算十进制数乘法时的步骤和操作数名称。先看一个例子，$1000_{10} \times 1001_{10}$，我们限制这个例子中只使用数字 0 和 1，稍后会解释限制原因。

```
被乘数          1000₁₀
  乘数    x     1001₁₀
              ------
              1000
             0000
            0000
           1000
          ------
  积      1001000₁₀
```

第一个操作数称为被乘数，第二个操作数称为乘数，最后的结果称作积。你也许还记得小学时学到的乘法法则，即每次从右到左选择乘数中的一位，用这一位乘上被乘数，然后将所得到的中间结果相对于前一位的中间结果左移一位。

可以观察到，积的位数比被乘数和乘数都大得多。事实上，如果我们忽略符号位，n 位被乘数和 m 位乘数的积是 $n+m$ 位的数。也就是说，需要 $n+m$ 位来表示所有可能的结果。因此，像加法一样，乘法也需要处理溢出，因为我们常常想用一个 64 位的乘积来表示两个 64 位数相乘的结果。

在这个例子中，我们把十进制数限制为 0 和 1。因为只有两个选择，所以每一步的乘法都很简单：

1. 如果乘数位为 1，只需将被乘数（1 × 被乘数）复制到适当的位置。
2. 如果乘数位为 0，则将 0（0 × 被乘数）置于适当的位置。

虽然上面十进制的例子恰巧只使用了 0 和 1，但是二进制乘法必须只使用 0 和 1，因此也只提供了两个选择。

现在我们已经回顾了乘法的基础知识，按照惯例下一步是要介绍高度优化的乘法硬件。我们打破了这一惯例，相信你将通过观察多代乘法硬件和算法的演变而获得更深入的理解。现在，让我们假设被乘数和乘数都是正数。

3.3.1 串行版的乘法算法及其硬件实现

该设计模仿了我们在小学学到的算法，图 3-3 展示了该设计的硬件结构。我们绘制出了硬件结构，使得数据从上到下流动，更接近于使用纸笔计算的方法。

假设乘数位于 64 位乘法器寄存器中，并且将 128 位乘积寄存器初始化为 0。从上面的纸笔法示例中可以清楚地看到，我们需要在每一步计算中将被乘数左移一位，因为它可能会和之前的中间结果相加。在 64 步计算之后，64 位被乘数会向左移动 64 位。因此，我们需要一个 128 位的被乘数寄存器，将其初始化为右半部分的 64 位被乘数和左半部分的零。然后

该寄存器每执行一步便左移 1 位，将被乘数与 128 位的乘积寄存器中的中间结果对齐并累加到中间结果。

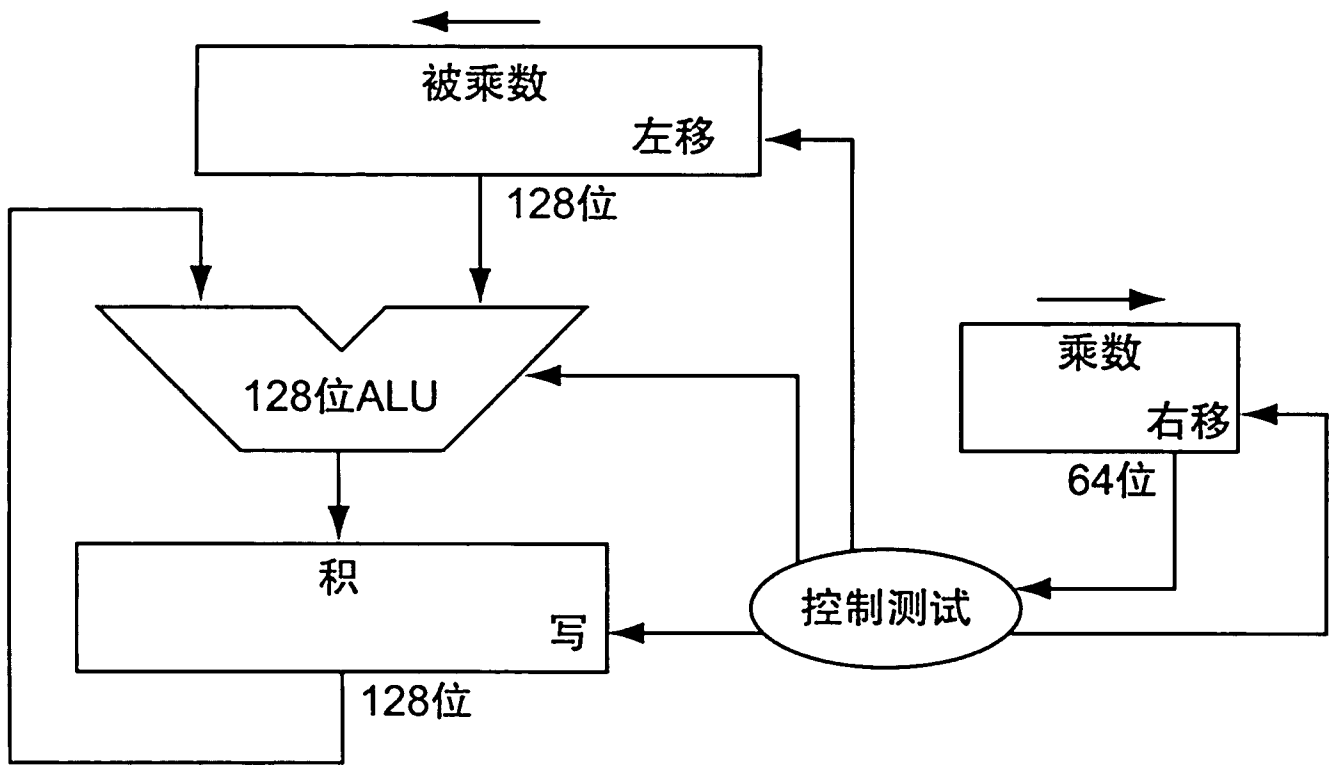


图 3-3　第一版乘法器硬件。被乘数寄存器、ALU 和乘积寄存器都是 128 位长，只有乘数寄存器是 64 位。（附录 A 描述了 ALU。）64 位被乘数最初存放在被乘数寄存器的右半部分，每一步左移 1 位。乘数在每步以相反方向移位。算法开始前，积初始化为 0。控制部件决定何时对被乘数寄存器和乘数寄存器进行移位，以及何时将新值写入积寄存器

图 3-4 显示了对于操作数的每一位都需要做的三个基本步骤。第一步中的乘数最低位（乘数第 0 位）决定了是否要把被乘数加到积寄存器当中。第二步中的左移起着将中间操作数左移的作用，就像手工纸笔做乘法一样。第三步中的右移给出了下次迭代要检测的乘数的下一位。这三个步骤重复 64 次就会得到最后的积。如果每个步骤花费一个时钟周期，那么该算法计算两个 64 位数相乘差不多要花费 200 个时钟周期。像乘法这样的算术运算的重要性随程序的不同而变化，但一般加法和减法出现的次数会是乘法的 5 到 100 倍。因此，在许多应用中，乘法花费若干时钟周期并不会显著影响性能。但是，Amdahl 定律（参见 1.10 节）提醒我们，一个慢速操作如果占据了一定的比例，也会限制程序性能。

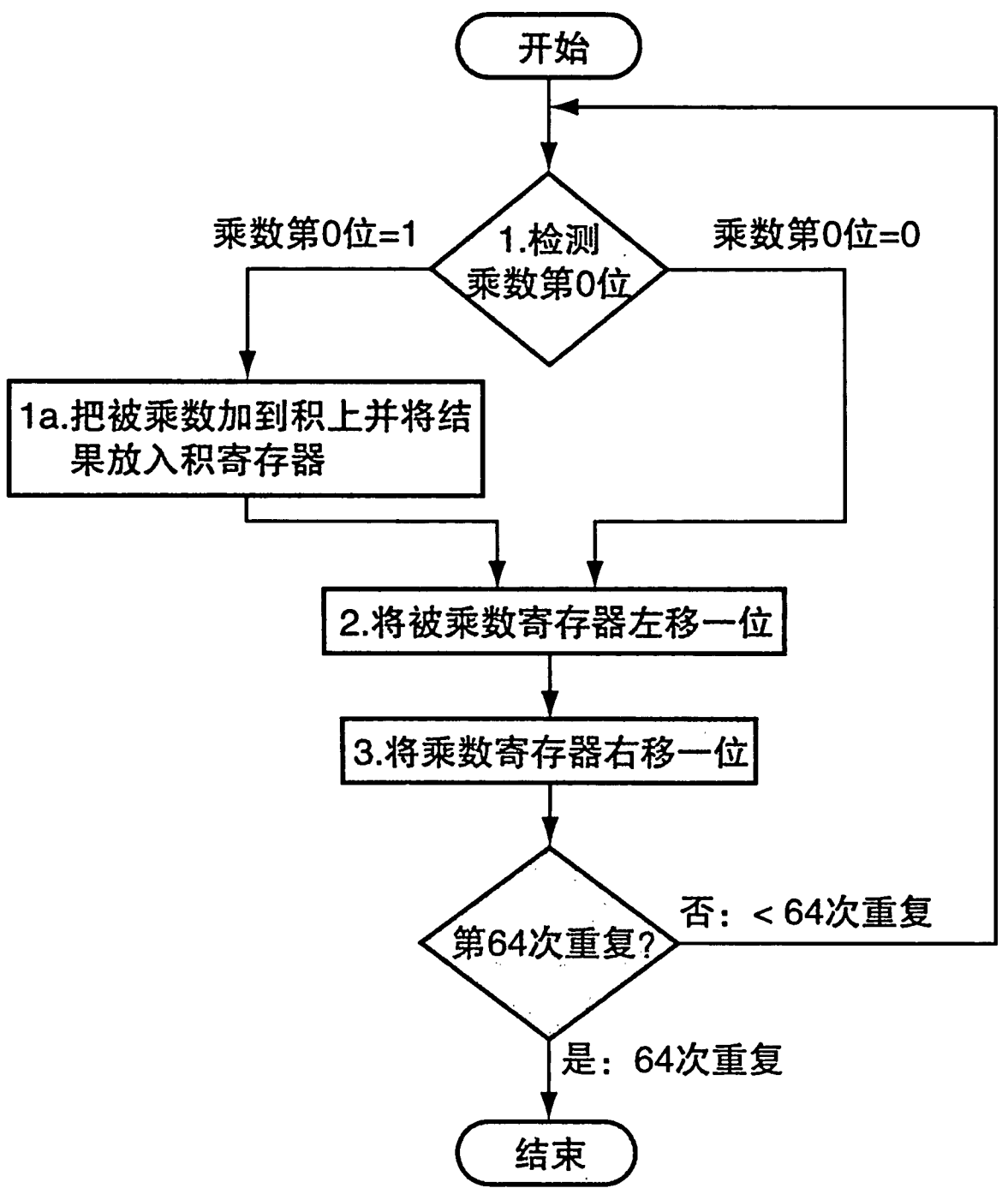


图 3-4　第一种乘法算法，采用了图 3-3 所示的硬件。如果乘数的最低有效位为 1，则将乘数加到积上。如果不是，则执行下一步。在下两步中将被乘数左移和乘数右移。将这三个步骤重复 64 次

这种算法和硬件很容易改进到每步只花费一个时钟周期。加速来源于操作的并行执行：如果乘数位是 1，那么

对被乘数和乘数进行移位，与此同时，把被乘数加到积上。硬件只需要保证它检测的是乘数的最右位，而且得到的是被乘数移位前的值。注意到寄存器和加法器有未使用的部分后，通常会将加法器和寄存器的位长减半以进一步优化硬件结构。图 3-5 展示了修正后的硬件。

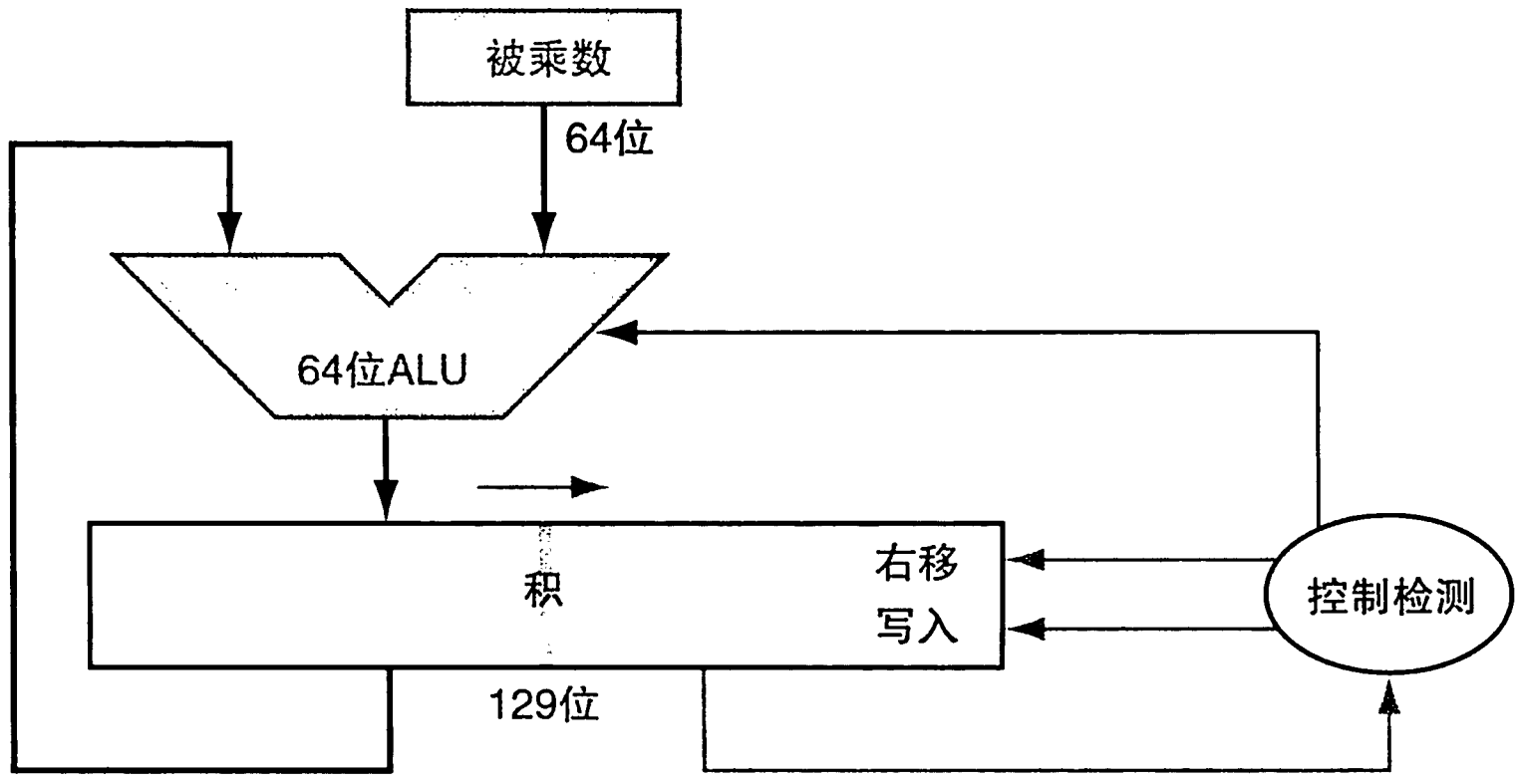


图 3-5　改良版乘法器硬件。和图 3-3 的第一版相比：被乘数寄存器和 ALU 都被缩减为 64 位，现在是积寄存器来进行右移，独立的乘数寄存器也消失了；乘数被放到了积寄存器的右半部分，该寄存器也从 128 位增加一位到 129 位以保存加法器的进位。这些改变的地方用灰色显示

硬件/软件接口　当乘数是常数的时候，也可以用移位运算来代替算术运算。有些编译器会将短常数的乘法运算替换为一系列的移位和加法运算。因为左移一位相当于把这个数变为之前的 2 倍，因此左移相当于把它和 2 的幂相乘。正如第 2 章提到的，几乎所有的编译器都会用移位运算来代替同 2 的幂相乘，进行强度缩减优化。

例题｜乘法算法

使用 4 位长的数以节省空间，计算 $2_{10} \times 3_{10}$ 或者 $0010_2 \times 0011_2$ 的乘积。

答案｜ 图 3-6 给出了按图 3-4 标出的每一步各个寄存器中的值，最后得到的结果是 00000110_2 或者 6_{10}。灰色的数表示寄存器值在该步的变化，被圈起来的位是用来决定下一步操作的待检测位。

迭代次数	步骤	乘数	被乘数	积
0	初始值	001①	0000 0010	0000 0000
1	1a：1=> 积 = 积 + 被乘数	0011	0000 0010	0000 0010
	2 ：被乘数左移	0011	0000 0100	0000 0010
	3 ：乘数右移	000①	0000 0100	0000 0010
2	1a：1=> 积 = 积 + 被乘数	0001	0000 0100	0000 0110
	2 ：被乘数左移	0001	0000 1000	0000 0110
	3 ：乘数右移	000⓪	0000 1000	0000 0110
3	1 ：0=> 无操作	0000	0000 1000	0000 0110
	2 ：被乘数左移	0000	0001 0000	0000 0110
	3 ：乘数右移	000⓪	0001 0000	0000 0110
4	1 ：0=>无操作	0000	0001 0000	0000 0110
	2 ：被乘数左移	0000	0010 0000	0000 0110
	3 ：乘数右移	0000	0010 0000	0000 0110

图 3-6　采用图 3-4 算法的乘法举例。圆圈圈起来的是决定下一步执行的待检测位

3.3.2 带符号乘法

到目前为止，我们只处理了正数乘法的情况。对于如何处理带符号乘法，最简单的方式是先把被乘数和乘数转换为正数，然后记住它们的初始符号。这样，将之前的算法迭代执行31次，符号位不参与计算。正如我们小学学到的那样，只有在乘数和被乘数符号相反时，对积取反。

事实证明，如果记住我们正在处理具有无限位长的数，并且只用64位来表示它们，则上面的最后一种算法适用于带符号数。因此，在移位时需要对带符号数的积进行符号扩展。当算法结束时，低位的双字就是64位积。

3.3.3 快速乘法

摩尔定律提供了非常充足的资源，从而使硬件设计人员可以实现更快的乘法硬件。通过在乘法运算开始的时候检查64个乘数位，就可以判定是否要将被乘数加上。快速乘法可以通过为每个乘数位提供一个64位加法器来实现：一个输入是被乘数和一个乘数位相与的结果，另一个输入是上一个加法器的输出。

一种简单的方法是将右侧加法器的输出端连接到左侧加法器的输入端，形成一个高64位的加法器栈。另一种方式是将这64个加法器组织成如图3-7所示的并行树。这样我们就只需要等待 $\log_2(64)$，即6次64位长加法的时间，而不是64次。

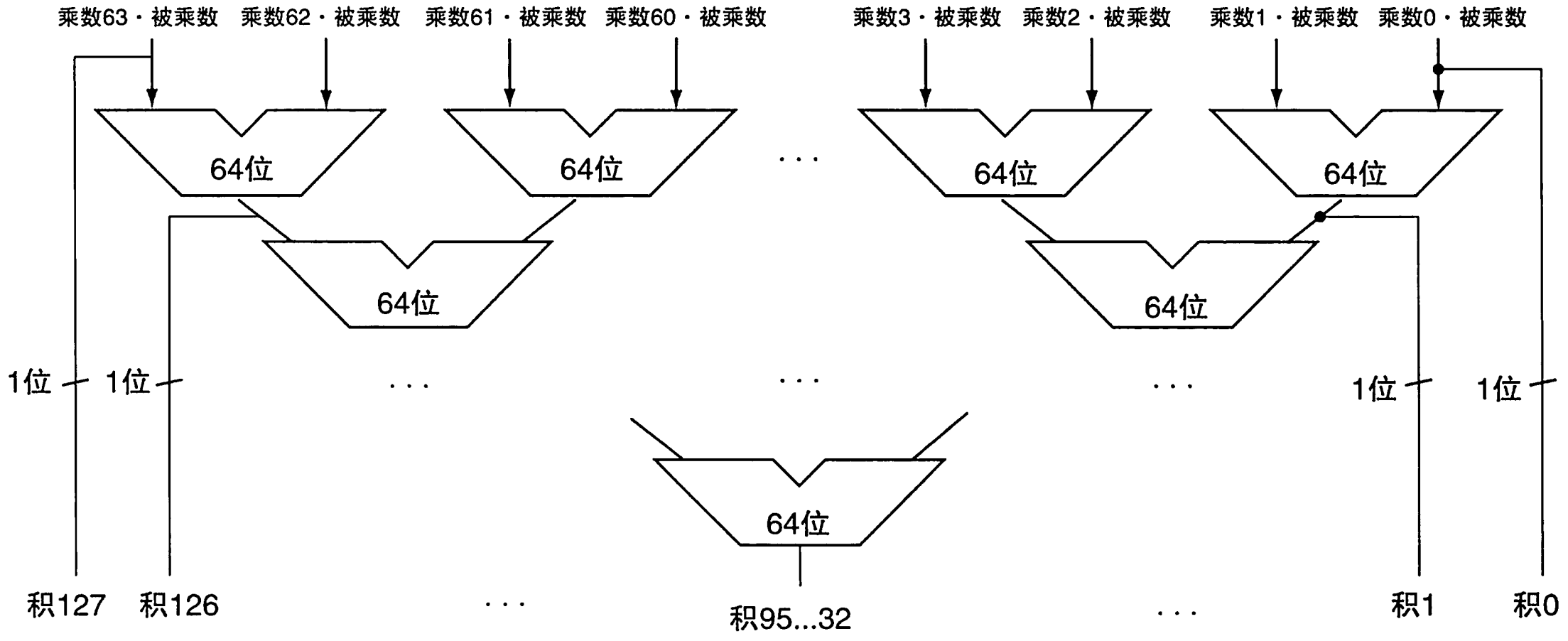


图3-7 快速乘法器硬件。这个硬件“展开循环”来使用63个加法器，并协调它们实现最小化时延，而不是使用单个64位的加法器63次

事实上，由于使用进位保留加法器（参见附录A中的A.6节），乘法的速度甚至比6次加法还要快，并且因为容易将上述的设计流水化，它能够同时支持多个乘法。

3.3.4 RISC-V中的乘法

为了产生正确带符号或无符号的128位积，RISC-V有四条指令：乘（mul），乘法取高位（mulh），无符号乘法取高位（mulhu）和有符号/无符号乘法取高位（mulhsu）。要获得整数64位积，程序员应使用用mul指令。要想得到128位积的高64位，如果两个操作数都是有符号的，程序员应使用mulh指令；如果两个操作数都是无符号的，则使用mulhu