3.10　本章小结

数十年来，计算机算术已经在很大程度上被标准化，这极大地提高了程序的可移植性。在当今出售的每台计算机中，都有二进制补码整数算法，若其包含对浮点的支持，则提供IEEE 754 二进制浮点算法。

计算机算术与传统算术的不同在于其受到有限精度的约束。该限制可能因为计算大于或小于预定限制的数而导致无效操作。这种例外现象称为“上溢”或“下溢”，可能导致例外或中断，以及类似于意外子程序调用的突发事件。第4章和第5章更详细地讨论了例外。

浮点算术作为实数的近似增加了挑战性，并且需要注意确保选择的计算机数能最接近实际数字。不精确性和有限浮点表达带来的挑战是数值分析领域的部分灵感来源。最近转向并行性的趋势使得数值分析再次获得关注，虽然在顺序计算机上长期被认为是安全的解决方案，但是在并行计算机中必须重新考虑，在寻找最快算法的同时也要获取正确结果。

数据级并行，特别是子字平行，为整数或浮点数据的算术密集型程序提供了一条提高性能的简单途径。我们展示了使用可一次执行4个浮点操作的指令来将矩阵乘法速度提高近4倍。

本章在解释计算机算术的同时也包含了更多RISC-V指令系统的描述。

图3-24对SPEC CPU2006整数和浮点基准的20个最常用RISC-V指令的使用频率进行了排序。可以看出，少数的指令主导着这些排名。在第4章中可以看到，这一观察结果对处理器的设计有重大影响。

RISC-V指令	名称	频率	累计
立即数加法	addi	14.36%	14.36%
取双字	ld	8.27%	22.63%
取双精度浮点	fld	6.83%	29.46%
寄存器加法	add	6.23%	35.69%
取字	lw	4.38%	40.07%
存双字	sd	4.29%	44.36%
不等则分支	bne	4.14%	48.50%
左移立即数	slli	3.65%	52.15%
乘加混合	fmadd.d	3.49%	55.64%
相等则分支	beq	3.27%	58.91%
立即数字加法	addiw	2.86%	61.77%
存双精度浮点	fsd	2.24%	64.00%
双精度浮点乘法	fmul.d	2.02%	66.02%
取立即数高位	lui	1.56%	67.59%
存字	sw	1.52%	69.10%
跳转并链接	jal	1.38%	70.49%
小于则分支	blt	1.37%	71.86%
字加法	addw	1.34%	73.19%
双精度浮点减法	fsub.d	1.28%	74.47%
大于或等于则分支	bge	1.27%	75.75%

图3-24　SPEC CPU2006基准测试中RISC-V指令的使用频率。表中包含20条最常使用的指令，这些指令占所有执行指令的76%。伪指令在执行前转换为RISC-V，因此在这里不会出现，这在一定程度上说明addi使用频繁

无论指令系统是什么或其规模如何（RISC-V、MIPS、x86），永远不要忘记数位没有内在含义。相同的数位可以表示有符号整数、无符号整数、浮点数、字符串、指令等。在存储程序计算机中，对数位的操作决定其含义。

3.11 历史视角和拓展阅读

本节将回溯到冯·诺依曼时代以纵览浮点历史，包括有争议的IEEE标准令人惊讶的成就，以及x86中80位浮点堆栈体系结构的基本原理。详见配套网站上的3.11节。

Gresham法则（“劣币驱逐良币”）对计算机而言是：“快的淘汰慢的，即使快的是错误的。”
W. Kahan, 1992

3.12 练习

从不屈服，决不屈服，从不、决不、绝不屈服于任何事情，无论伟大或渺小，庞大或细微——决不屈服。
1941年 Winston Churchill 在 Harrow School 的演讲

3.1 [5] <3.2> 5ED4-07A4用无符号16位十六进制数表示是什么？结果用十六进制表示。

3.2 [5] <3.2> 5ED4-07A4用带符号16位十六进制数且以符号–数值形式存储时如何表示？结果用十六进制表示。

3.3 [10] <3.2> 将5ED4转换为二进制数。是什么让十六进制表示计算机中的数值充满魅力？

3.4 [5] <3.2> 4365-3412用无符号12位八进制数表示是什么？结果用八进制表示。

3.5 [5] <3.2> 4365-3412用带符号12位八进制数且以符号–数值形式存储时如何表示？结果用八进制表示。

3.6 [5] <3.2> 假设185和122是无符号的8位十进制整数。计算185−122。是否上溢或下溢，或都没有？

3.7 [5] <3.2> 假设带符号的8位十进制整数185和122以符号–数值形式存储。计算185 + 122。是否上溢或下溢，或都没有？

3.8 [5] <3.2> 假定带符号的8位十进制整数185和122以符号–数值形式存储。计算185–122。是否上溢或下溢，或都没有？

3.9 [10] <3.2> 假设带符号的8位十进制整数151和214以二进制补码形式存储。使用饱和算法计算151 + 214。结果用十进制表示。

3.10 [10] <3.2> 假设带符号的8位十进制整数151和214以二进制补码形式存储。使用饱和算法计算151−214。结果用十进制表示。

3.11 [10] <3.2> 假设151和214是无符号8位整数。使用饱和算法计算151+ 214。结果用十进制表示。

3.12 [20] <3.3> 使用类似于图3-6所示的表格，使用图3-3中的硬件描述计算八进制无符号6位整数62和12的乘积。须在每步中写出各个寄存器的内容。

3.13 [20] <3.3> 使用类似于图3-6所示的表格，使用图3-5中的硬件描述计算十六进制无符号8位整数62和12的乘积。须在每步中写出各个寄存器的内容。

3.14 [10] <3.3> 如果一个整数是8位宽且每步操作需要4个时间单位，使用图3-3和图3-4中给出的方法计算执行一次乘法所需的时间。假设在步骤1a中总是执行加法，无论是加上被乘数还是零。另外假设寄存器已经被初始化（你只需要计算执行乘法循环本身需要多长时间）。如果是在硬件中执行，则可以同时完成被乘数和乘数的移位。如果这是在软件中执行，则必须依次完成。给出每种情况的解答。

3.15 [10] <3.3> 如果一个整数是 8 位宽且一次加法需要 4 个时间单位，计算使用文中描述的方法（31 个垂直加法堆栈）执行乘法所需的时间。

3.16 [20] <3.3> 如果一个整数是 8 位宽且一次加法需要 4 个时间单位，计算使用图 3-7 给出的方法执行乘法所需的时间。

3.17 [20] <3.3> 正如文中讨论的，增强性能的一种可能是做移位和加法来替代实际的乘法。因为例如 9×6 可以写成（2×2×2 + 1）×6，所以可以通过将 6 向左移动 3 次再加上 6 来计算 9×6。给出使用移位和加 / 减法计算 0×33×0×55 的最佳方法。假设两个输入都是 8 位无符号整数。

3.18 [20] <3.4> 使用类似于图 3-10 所示的表格，使用图 3-8 中的硬件描述计算 74 除以 21。须在每步中给出各个寄存器的内容。假设两个输入都是无符号的 6 位整数。

3.19 [30] <3.4> 使用类似于图 3-10 所示的表格，使用图 3-11 中的硬件描述计算 74 除以 21。须在每步中给出各个寄存器的内容。假设 A 和 B 是无符号的 6 位整数。此算法使用的方法与图 3-9 中所示的稍有不同。你需要认真思考这个问题，做一个或两个实验，或者去网上寻找方法以使其正确工作。（提示：一种可能的解决方案涉及图 3-11 暗示的余数寄存器可以向任一方向移位的事实。）

3.20 [5] <3.5> 如果是整数的二进制补码，位模式 0x0C000000 表示的十进制数是什么？如果是无符号整数呢？

3.21 [10] <3.5> 如果位模式 0x0C000000 被放入指令寄存器，将执行什么 MIPS 指令？

3.22 [10] <3.5> 如果是浮点数，位模式 0x0C000000 表示的十进制数是什么？使用 IEEE 754 标准。

3.23 [10] <3.5> 假定采用 IEEE 754 单精度格式，写出十进制数 63.25 的二进制表示。

3.24 [10] <3.5> 假定采用 IEEE 754 双精度格式，写出十进制数 63.25 的二进制表示。

3.25 [10] <3.5> 假定使用单精度 IBM 格式（基数为 16，而不是 2，指数为 7 位）存储，写出十进制数 63.25 的二进制表示形式。

3.26 [20] <3.5> 假定采用一种类似 DEC PDP-8 的格式（最左边 12 位是指数，以二进制补码形式存储，最右边 24 位是小数，同样以二进制补码形式存储），写出 -1.5625×10^{-1} 的二进制位模式。没有隐含 1。与 IEEE 754 标准的单精度和双精度进行比较，评估这个 36 位模式的范围和精确度。

3.27 [20] <3.5 >IEEE 754-2008 包含一种只有 16 位宽的“半精度”格式。最左边仍是符号位，指数 5 位宽且以余 −16（excess -16）的形式存储，偏移量是 15，尾数是 10 位宽。假设隐含 1。写出该格式表示 -1.5625×10^{-1} 的位模式。与 IEEE 754 标准的单精度进行比较，评估该 16 位浮点格式的范围和精确度。

3.28 [20] <3.5 >Hewlett-Packard 2114、2115 和 2116 使用的格式为：最左边的 16 位是以二进制补码形式存储的小数，接在其后的是另一个 16 位字段，其最左边的 8 位作为小数扩展（让小数长为 24 位）且最右边的 8 位表示指数。然而，作为一个有趣的交叉，指数以符号 – 数值形式存储且符号位在最右边！写出该格式下 -1.5625×10^{-1} 的位模式。没有隐含 1。与 IEEE 754 标准的单精度进行比较，评估这个 32 位模式的范围和精确度。

3.29 [20] <3.5> 手动计算 2.6125×10^{1} 与 $4.150390625 \times 10^{-1}$ 的和，假设 A 和 B 以练习 3.27 所述的 16 位半精度格式存储。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤。

3.30 [30] <3.5> 手动计算 -8.0546875×10^{0} 与 $-1.79931640625 \times 10^{-1}$ 的乘积，假设 A 和 B 以练习 3.27 所述的 16 位半精度格式存储。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到

最近的偶数。给出所有步骤。然而，正如文中示例所示，你可以使用人为可读的格式执行乘法，而不是使用练习 3.12 ～ 3.14 中描述的技术。指出是否上溢或下溢。分别使用练习 3.27 中描述的 16 位浮点格式和十进制数来作答。你的结果精确度如何？如果在计算器上进行乘法运算，它与你得到的数字相比如何？

3.31 [30] <3.5> 手动计算 8.625×10^{1} 除以 -4.875×10^{0}。给出得到答案所需的所有必要步骤。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并在必要时使用它们。用练习 3.27 中描述的 16 位浮点格式和十进制数字写出最终答案，并将十进制结果与计算器算得结果进行比较。

3.32 [20] <3.10> 手动计算 $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3}$，假设每个值都以练习 3.27 中描述的 16 位半精度格式存储（在文中也有描述）。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤，并用 16 位浮点格式和十进制写出答案。

3.33 [20] <3.10> 手动计算 $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$，假设每个值都以练习 3.27 中描述的 16 位半精度格式存储（在文中也有描述）。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤，并用 16 位浮点格式和十进制写出答案。

3.34 [10] <3.10> 根据练习 3.32 和 3.33 的答案，计算 $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3} = 3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$ 是否成立？

3.35 [10] <3.10> 手动计算 $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^{2}$，假设每个值都以练习 3.27 中描述的 16 位半精度格式存储（在文中也有描述）。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤，并用 16 位浮点格式和十进制写出答案。

3.36 [30] <3.10> 手动计算 $3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$，假设每个值都以练习 3.27 中描述的 16 位半精度格式存储（在文中也有描述）。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤，并用 16 位浮点格式和十进制写出答案。

3.37 [30] <3.10> 根据练习 3.35 和 3.36 的答案，计算 $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^{2} = 3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$ 是否成立？

3.38 [30] <3.10> 手动计算 $1.666015625 \times 10^{0} \times (1.9760 \times 10^{4} + (-1.9744) \times 10^{4})$，假设每个值都以练习 3.27 中描述的 16 位半精度格式存储（在文中也有描述）。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤，并用 16 位浮点格式和十进制写出答案。

3.39 [30] <3.10> 手动计算 $(1.666015625 \times 10^{0} \times 1.9760 \times 10^{4}) + (1.666015625 \times 10^{0} \times (-1.9744) \times 10^{4})$，假设每个值都以练习 3.27 中描述的 16 位半精度格式存储（在文中也有描述）。假设有 1 个保护位、1 个舍入位和 1 个粘滞位，并舍入到最近的偶数。给出所有步骤，并用 16 位浮点格式和十进制写出答案。

3.40 [10] <3.10> 根据练习 3.38 和 3.39 的答案，计算 $(1.666015625 \times 10^{0} \times 1.9760 \times 10^{4}) + (1.666015625 \times 10^{0} \times (-1.9744) \times 10^{4}) = 1.666015625 \times 10^{0} \times (1.9760 \times 10^{4} + (-1.9744) \times 10^{4})$ 是否成立？

3.41 [10] <3.5> 使用 IEEE 754 浮点格式，写出表示 −1/4 的位模式。你能准确表示 −1/4 吗？

3.42 [10] <3.5> 如果将 −1/4 自加四次会得到什么？$-1/4 \times 4$ 呢？它们一样吗？它们应该是多少？

3.43 [10] <3.5> 假设分数使用二进制数浮点格式，写出值 1/3 的分数位模式。假设有 24 位，且不需要规格化。这种表示准确吗？

3.44 [10] <3.5> 假设分数使用二进制编码的十进制（底数为 10）数字而非底数为 2 的浮点格式，

写出值 1/3 的分数位模式。假设有 24 位，且不需要规格化。这种表示准确吗?

3.45 [10] <3.5> 假设在值 1/3 的分数中使用底数为 15 的数字而非底数为 2，写出其位模式。(底数为 16 的数字使用符号 0 ～ 9 和 A ～ F。底数为 15 的数字将使用 0 ～ 9 和 A ～ E。) 假设有 24 位，且不需要规格化。这种表示准确吗?

3.46 [20] <3.5> 假设在值 1/3 的分数中使用底数为 30 的数字而非底数为 2，写出其位模式。(底数为 16 的数字使用符号 0 ～ 9 和 A ～ F。底数为 30 的数字将使用 0 ～ 9 和 A ～ T。) 假设有 20 位，且不需要规格化。这种表示准确吗?

3.47 [45] <3.6, 3.7> 以下 C 代码在输入数组 sig_in 上实现了一个四阶 FIR 滤波器。假设所有数组都是 16 位定点值。

```
for (i = 3;i< 128;i+ +)
sig_out[i] = sig_in[i - 3] * f[0] + sig_in[i - 2] * f[1]
  + sig_in[i - 1] * f[2] + sig_in[i] * f[3];
```

假设你要在具有 SIMD 指令和 128 位寄存器的处理器上，用汇编语言编写此代码的优化实现。在不了解指令系统细节的情况下，简要描述你将如何实现此代码，最大限度地利用子字操作并最大限度地减少寄存器和内存之间的数据传输量。指明你对所使用指令的所有假设。

自我检测答案

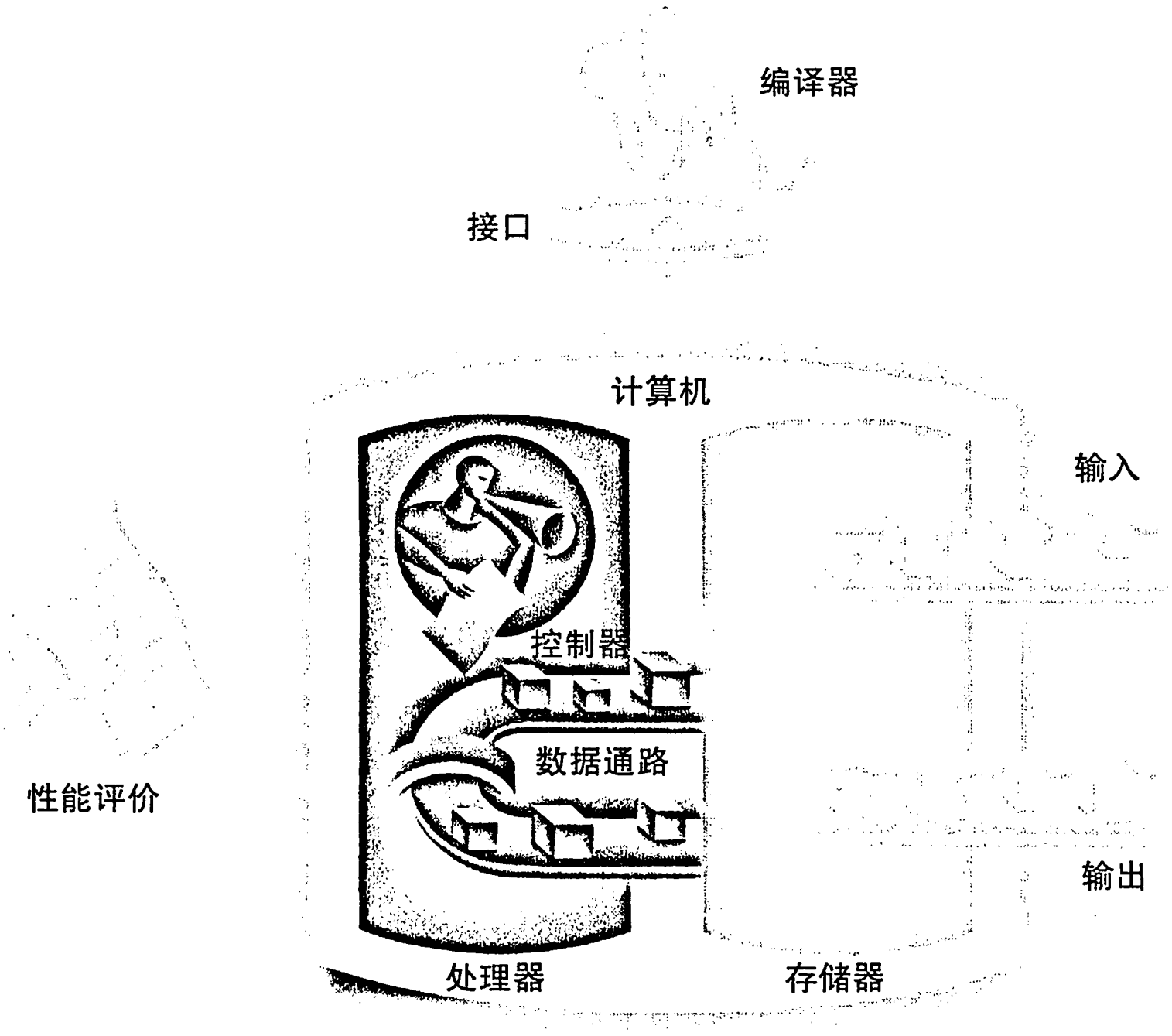
3.2 节　2。

3.5 节　3。

第4章

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

处 理 器



计算机的五个经典部件

4.1 引言

第1章阐述了计算机的性能取决于三个因素：指令数、时钟周期长度和每条指令的时钟周期数（CPI）。第2章说明了给定程序需要的指令数由编译器和指令系统体系结构共同决定。而处理器的实现方式则决定了时钟周期长度和CPI。在本章中，我们为RISC-V指令系统的两种不同实现方式分别设计数据通路并加入控制单元。

在关键问题上，没有什么细节是小事。
法国谚语

本节将概括地介绍实现处理器所要用到的原理和技术。首先从一个高度抽象和简化的概述开始，之后以此为基础为RISC-V指令系统构建数据通路，并设计一种简单的、能够实现指令系统的处理器。然而，更接近实际情况的是流水线RISC-V，所以本章的大部分篇幅将介绍这种实现方式。最后一节将介绍实现更复杂的指令系统（如x86指令集）时所需要了解的概念。

本节和4.5节介绍流水线的基本概念，如果想要理解指令的高层解释及其对程序性能的影响，可以仔细阅读这些部分。4.10节介绍目前处理器实现的发展趋势，4.11节讲述了最新处理器Intel Core i7和ARM Cortex-A53的架构。4.12节介绍如何通过指令级并行来提高矩

阵乘法的性能（见 3.9 节）。这几节为在高层次理解流水线概念提供了必要的背景知识。

如果读者想要更深入地理解处理器结构及其对性能的影响，那么 4.3、4.4、4.6 节的内容将有助于你的学习。如果想要学习如何实现处理器，那么你应该阅读 4.2、4.7 ～ 4.9 节。如果读者有兴趣学习硬件设计，4.13 节介绍了实现硬件时使用的硬件设计语言与 CAD 工具，以及如何使用硬件设计语言来描述一个流水化的实现。4.13 节对于理解流水化硬件执行的细节也有很大帮助。

4.1.1　一种基本的 RISC-V 实现

我们将实现 RISC-V 的一个核心子集：

- 存储器访问指令 load doubleword（ld）和 store doubleword（sd）。
- 算术逻辑指令 add、sub、and 和 or。
- 条件分支指令 branch if equal（beq）。

这个子集没有包含所有的定点指令（例如 shift、multiply 和 divide 指令均不在集合中），也没有包含任何浮点指令。但是，这个子集说明了建立数据通路和设计控制的关键原理。其余指令的实现与这个子集类似。

在完成指令系统实现的过程中，我们将认识到指令系统体系结构如何从多方面影响指令系统的实现，以及各种实现策略的选择会怎样影响计算机的时钟频率和 CPI。此外，实现过程也证明了许多第 2 章介绍的关键设计原则，例如简单源于规整。而且，本章在实现 RISC-V 子集时所涉及的大多数概念，与实现各种计算机所涉及的基本概念是相同的，如高性能服务器、通用微处理器、嵌入式处理器等各种计算机。

4.1.2　实现概述

在第 2 章，我们着眼于 RISC-V 核心指令，包括定点算术逻辑指令、存储器访问指令和分支指令。执行这些指令所要做的大部分工作是相同的，与确切的指令类别无关。具体地，实现每条指令的前两个步骤是相同的：

1. 程序计数器（PC）发送到指令所在的存储单元，并从中取出指令。

2. 根据指令的某些字段选择要读取的一个或两个寄存器。对于 ld 指令，只需要读取一个寄存器，但大多数其他指令需要读取两个寄存器。

在这两个步骤后，完成指令所需的剩余操作则取决于指令类别。幸运的是，对于三类指令（存储器访问指令、算术逻辑指令和分支指令）中的每一种，剩余操作基本上是相同的，与具体的指令无关。RISC-V 指令系统的简单性和规整性使不同类的指令具有类似的执行过程，从而简化了实现。

例如，所有类型的指令在读取寄存器后都使用算术逻辑单元（ALU）。存储器访问指令用 ALU 进行地址计算，算术逻辑指令用 ALU 来执行运算，而条件分支指令用 ALU 进行比较。但是经过 ALU 后，完成各类指令所需的操作就不同了。存储器访问指令需要访问存储器以读取数据或存储数据。算术逻辑指令或载入指令需要将来自 ALU 或存储器的数据写回寄存器。而条件分支指令需要根据比较结果更改下一条指令的地址；否则，下一条指令的地址会通过 PC 加 4 来获得。

图 4-1 是 RISC-V 实现的抽象视图，图中主要描述了各功能单元及它们之间的互连。尽管该图展示了经过处理器的大部分数据流动，但忽略了指令执行的两个重要方面。

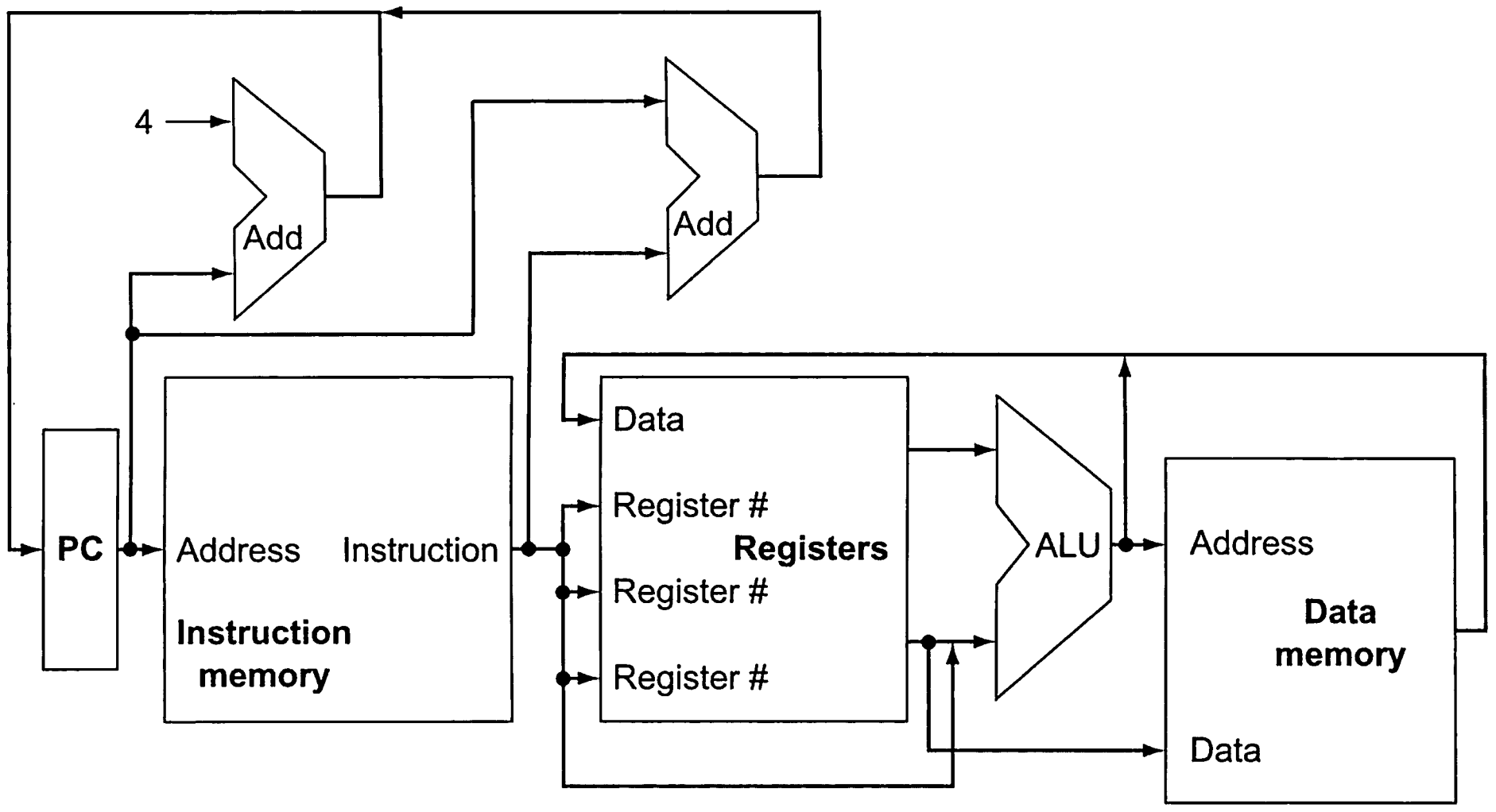


图 4-1 RISC-V 子集实现的抽象图，描述主要功能单元及它们之间的主要连接。所有的指令都始于用程序计数器获得指令在指令存储器中的地址。取到指令后，指令的对应字段指定要使用的寄存器操作数。寄存器操作数一被取出，即可用于计算存储器地址（load 指令或 store 指令）、计算算术运算结果（定点算术逻辑指令）或进行相等检查（分支指令）。如果是算术逻辑指令，ALU 的结果必须写回寄存器。如果是存取操作，ALU 的结果将作为存储器地址以存储来自寄存器的值，或将存储器数据加载到寄存器中。ALU 结果或访问存储器的结果将写回到寄存器堆。分支指令需要根据 ALU 输出来确定下一条指令的地址，这个地址可能来自计算 PC 和分支偏移量相加的加法器，也可能来自计算当前 PC 加 4 的加法器。图中连接功能单元的粗线表示总线，它由多个信号组成。箭头表示信息如何流动。由于信号线可能会交叉，所以在交叉信号线确实相连时用一个黑点来表示

首先，图 4-1 中有许多这样的位置：两个来自不同源的数据流向同一个单元。例如，写入 PC 的值可能来自两个加法器中的一个，写入寄存器堆的数据可能来自 ALU 或数据存储器，而 ALU 的第二个输入可能来自寄存器或指令的立即数字段。实际上，这些数据线不能简单地连接在一起。我们必须添加一种逻辑单元以从多个数据源中选择一个送给目标单元。这种选择通常由称为多选器（multiplexor）的设备来完成，虽然它可能更合适被称为数据选择器。附录 A 详细描述了多选器如何根据控制信号从多个输入中进行选择。控制信号通常由当前执行指令中包含的信息决定。

图 4-1 忽略的第二个内容是，一些功能单元的控制依赖于当前执行的指令类型。例如，数据存储器必须在指令是 load 时被读，在指令是 store 时被写。寄存器堆只能在指令是 load 或算术逻辑指令时被写。但是，ALU 的控制不依赖指令类型，它一定会做某种运算。（附录 A 介绍了 ALU 的详细设计。）与多选器类似，ALU 的控制线也根据指令的某些字段来设置，进而控制 ALU 做哪种运算。

图 4-2 在图 4-1 的基础上添加了三个必要的多选器，以及主要功能单元的控制线。图中还增加了一个控制单元，它以指令作为输入，为功能单元及两个多选器设置控制信号。图中最上面的多选器确定写入 PC 的是 PC + 4 还是分支目标地址，在执行 beq 指令时，该多选器的控制信号由 ALU 进行比较时设置的 Zero 输出来设置。RISC-V 指令系统的规整性和简单性使得只需简单的译码过程即可确定如何设置控制线。

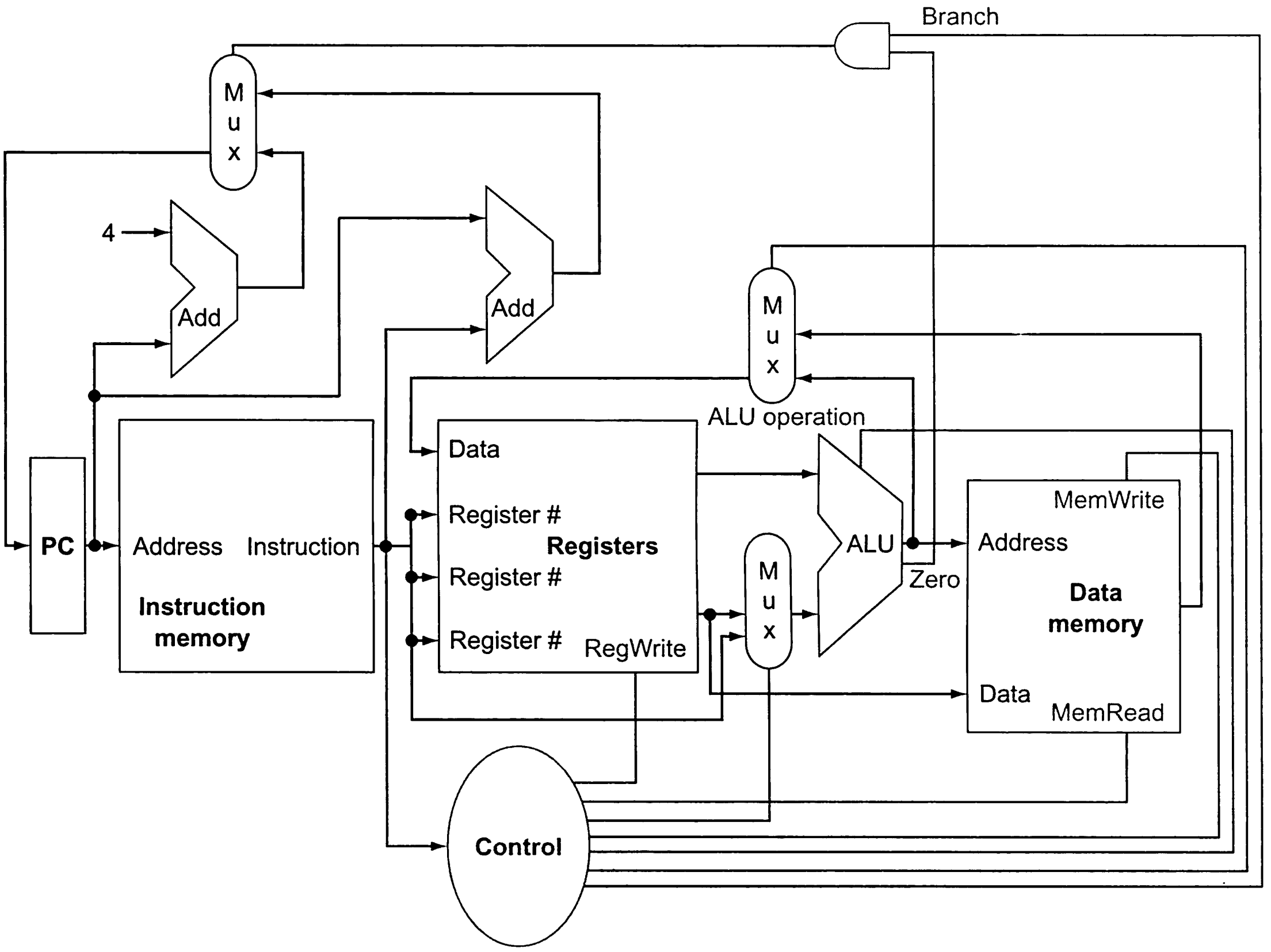


图 4-2 包含必要多选器和控制线的 RISC-V 子集的基本实现。最上面的多选器（“Mux”）控制写入 PC 的值（PC + 4 或分支目标地址）；该多选器由逻辑门控制，该控制将 ALU 的零输出和表示指令为分支指令的控制信号进行“与”。中间的输出到寄存器堆的多选器用于确定将 ALU 的输出（算术逻辑指令）或数据存储器的输出（load 指令）写入寄存器堆。最后，最下面的多选器用于确定 ALU 的第二个输入是来自寄存器（算术逻辑指令或分支指令）还是指令的偏移量字段（load 指令或 store 指令）。图中新增的控制线确定 ALU 做哪种运算、数据存储器的读写以及寄存器堆的写入等。控制线在图中用灰色标识

在本章剩余部分中，我们将逐步改进图 4-2 并补充细节，添加更多的功能单元以及单元之间的连接，并且改进控制单元以控制不同类指令完成执行。4.3 节和 4.4 节描述了每条指令使用一个时钟周期的简单实现方式，它遵循图 4-1 和图 4-2 的一般形式。在这种设计中，每条指令从一个时钟边沿开始执行，并在下一个时钟边沿完成执行。

这种方式虽然很容易理解，但并不实用，因为时钟周期必须设置为足够容纳执行时间最长的指令。在实现了这种简单的计算机控制之后，我们将介绍流水线实现方式及其复杂性和例外。

自我检测 图 4-1 和图 4-2 包含了计算机的五个主要组成部分（见本章开始处的图片）中的哪几个部分？

4.2 逻辑设计的一般方法

在考虑计算机的设计时，需要确定实现计算机的硬件逻辑以及这些逻辑如何定时。本节

将回顾一些在本章中会广泛用到的数字逻辑的关键概念。如果你的数字逻辑知识很少，那么在阅读本节之前先阅读附录 A 将有所帮助。

RISC-V 实现中的数据通路包含两种不同类型的逻辑单元：处理数据值的单元和存储状态的单元。处理数据值的单元是**组合逻辑**，它们的输出仅依赖于当前输入。给定相同的输入，组合逻辑单元总是产生相同的输出。例如，图 4-1 和附录 A 中讨论的 ALU 就是一个组合逻辑单元。由于组合逻辑单元没有内部存储功能，当给定一组输入时，它总是产生相同的输出。

组合逻辑单元：一个操作单元，如 AND 门或 ALU。

状态单元：一个存储单元，如寄存器或存储器。

设计中的其他单元不是组合逻辑，而是包含状态的。如果一个单元有内部存储功能，它就包含状态，称其为**状态单元**。这是因为关机后重启计算机，通过恢复状态单元的原值，计算机可继续运行，就像没有发生过断电一样。进一步地，这些状态单元可以完整地表征计算机。例如，图 4-1 中的指令存储器、数据存储器以及寄存器都是状态单元。

一个状态单元至少有两个输入和一个输出。必需的输入是要写入状态单元的数据值和决定何时写入数据值的时钟信号。状态单元的输出提供了在前一个时钟周期写入单元的数据值。例如，逻辑上最简单的一种状态单元是 D 触发器（见附录 A），它有两个输入（一个数据值和一个时钟）和一个输出。除了触发器，RISC-V 的实现中还用到了另外两种状态单元：存储器和寄存器。这两种状态单元均在图 4-1 中出现过。状态单元何时被写入由时钟确定，但是它随时可以被读。

包含状态的逻辑部件也被称为时序的，因为其输出取决于输入和内部状态。例如，表示寄存器的功能单元的输出取决于所提供的寄存器号和之前写入寄存器的内容。附录 A 更详细地讨论了组合逻辑单元和时序逻辑单元的相关操作及其结构。

时钟同步方法

时钟同步方法（clocking methodology）规定了信号可以读出和写入的时间。规定信号的读写时间非常重要，因为如果在读信号的同时写信号，那么读到的值可能是该信号的旧值，也可能是新写入的值，甚至可能是二者的混合。计算机设计无法容忍这种不可预测性。时钟同步方法就是为避免这种情况而提出的。

时钟同步方法：用来确定数据相对于时钟何时稳定和有效的方法。

边沿触发的时钟：所有状态的改变发生于时钟边沿的机制。

为简单起见，假定我们采用**边沿触发的时钟**（edge-triggered clocking），即存储在时序逻辑单元中的所有值仅在时钟边沿更新，这是从低电平快速跳变到高电平（反之亦然）的过程（见图 4-3）。因为只有状态单元能存储数据值，所有组合逻辑单元都必须从状态单元集合接收输入，并将输出写入状态单元集合。其输入是之前某时钟周期写入的值，输出的值可以在后续时钟周期使用。

图 4-3 描述了一个组合逻辑单元及与其相连的两个状态单元。组合逻辑单元的操作在一个时钟周期内完成：所有信号在一个时钟周期内从状态单元 1 经组合逻辑单元到达状态单元 2。信号到达状态单元 2 所需的时间决定了时钟周期的长度。

为简单起见，如果状态单元在每个有效时钟边沿都进行写入，则可忽略写**控制信号**（control signal）。相反，如果状态单元不是在每个时钟边沿都更新，那么它需要一个写控制信号。时钟信号和写控制信号都是输入。仅当时钟边沿到来并且写控制信号有效时，状

控制信号：用来决定多选器选择或指示功能单元操作的信号；它与数据信号相对应，数据信号包含功能单元所操作的信息。

态单元才改变状态。

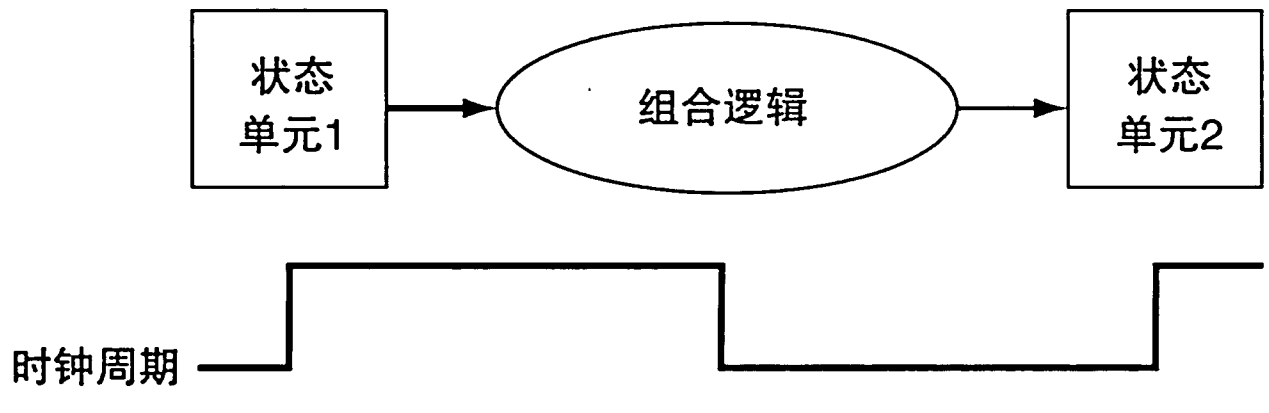


图 4-3　组合逻辑、状态单元和时钟周期的关系。在同步数字系统中，时钟信号决定了数据值何时写入状态单元。在有效的时钟边沿导致状态变化之前，状态单元的所有输入必须达到稳定（也就是说，状态单元会保持这个值不变，直到时钟边沿到来）。本章所有状态单元（包括存储器）都假定为上升沿触发，即这些信号在时钟的上升沿发生变化

我们将用术语**有效**（asserted）表示信号为逻辑高，用使有效表示信号应为逻辑高，用**无效**或使无效表示信号为逻辑低。我们使用术语有效和无效，是因为在进行硬件实现时，数字 1 有时表示逻辑高，有时表示逻辑低。

有效：信号为逻辑高或真。

无效：信号为逻辑低或假。

在边沿触发的时钟同步方法中，需在一个时钟周期内读出寄存器的值，并使之经过组合逻辑单元，将新值写入该寄存器。图 4-4 给出了一个通用的例子。选择在时钟的上升沿（从低到高）还是下降沿（从高到低）进行写操作无关紧要，因为组合逻辑的输入只有在所规定的时钟边沿才可能发生变化。在本书中，我们选择在时钟上升沿写入。边沿触发的时钟同步在单个时钟周期内不会出现反馈，图 4-4 中的逻辑可以正确工作。在附录 A 中，我们简要讨论了其他的时序约束（例如建立和保持时间）和时钟同步方法。

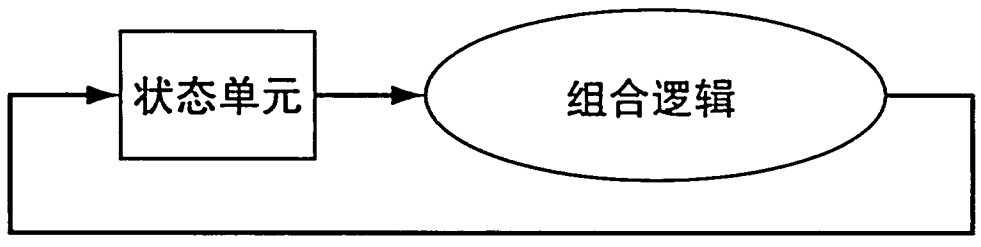


图 4-4　边沿触发的时钟同步方法，支持状态单元在同一个时钟周期内读和写，这样不会因为竞争而出现中间数据。当然，时钟周期必须足够长，以便在有效时钟边沿到来时输入值已经稳定。由于状态单元在时钟边沿更新，所以在一个时钟周期内不可能出现反馈。如果有反馈，这种设计就不能正常工作。我们在本章和下一章的设计都采用边沿触发的时钟同步方法以及如图所示的结构

对于 64 位 RISC-V 指令系统体系结构，几乎所有的状态单元和逻辑单元的输入和输出都是 64 位，因为处理器处理的大部分数据的宽度是 64 位。如果某个单元的输入或输出不是 64 位宽，我们会特别指出。图中用粗线表示总线，即宽度为 1 位以上的信号。有时要把几根总线合起来构成一根更宽的总线，例如，将两根 32 位总线合成一根 64 位总线。在这种情况下，总线标注将做出相应说明。箭头用以指明单元间数据流动的方向。最后，灰色线表示的控制信号将其与数据信号区分开来，两者的差别将随本章的进展越来越明显。

自我检测　判断题：由于寄存器堆在同一个时钟周期内既要写入又要读出，所以任何边沿触发式写入的 RISC-V 数据通路中都必须包含多个寄存器堆的备份。

详细阐述 还有一种 32 位版本的 RISC-V 指令系统，其实现中的大多数通路都是 32 位宽。

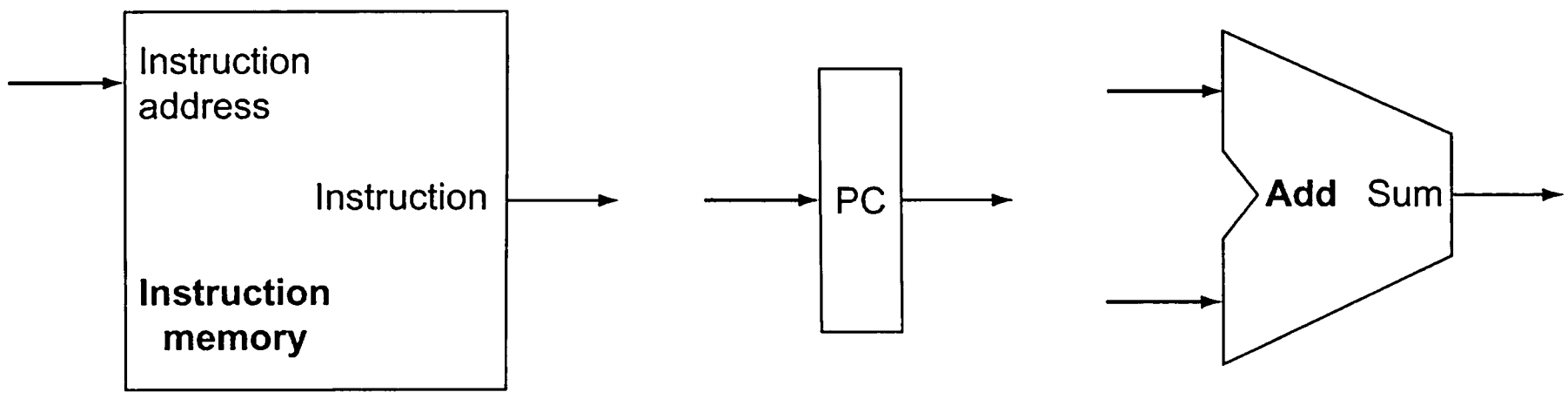
4.3 建立数据通路

设计数据通路的合理方法是，先分析每类 RISC-V 指令需要哪些主要执行单元。本节首先讨论每条指令需要哪些**数据通路单元**（datapath element），然后逐渐降低抽象的层次。在设计数据通路单元的同时，也会设计它们的控制信号。我们将自底向上地使用抽象的思想对此进行说明。

数据通路单元：一个用来操作或保存处理器中数据的单元。在 RISC-V 实现中，数据通路单元包括指令存储器、数据存储器、寄存器堆、ALU 和加法器。

图 4-5a 所示为需要的第一个单元——存储单元，用于存储程序的指令，并根据给定地址提供指令。图 4-5b 所示为**程序计数器**（PC），正如第 2 章所述，它用于保存当前指令的地址。最后还需要一个加法器来增加 PC 的值以获得下一条指令的地址。这个加法器是一个组合逻辑电路，可由附录 A 中描述的 ALU 实现，只需将其中的控制信号设为总是进行加法运算即可。如图 4-5c 所示，给这样的 ALU 加上“Add”标记，以表明它是加法器并且不能执行其他 ALU 操作。

程序计数器：包含当前程序正在执行的指令地址的寄存器。



a）指令存储器　　b）程序计数器　　c）加法器

图 4-5　存取指令需要的两个状态单元，以及计算下一条指令的地址所需的加法器。两个状态单元分别是指令存储器和程序计数器。因为数据通路不会写入指令，所以指令存储器只提供读访问，因此将其视为组合逻辑：任何时刻的输出都反映了输入地址内容的变化，而不需要读控制信号。（在加载程序时需要写入指令存储器；这并不难实现，所以为简单起见我们将这个步骤忽略。）程序计数器是一个 64 位寄存器，它在每个时钟周期结束时会被写入，所以不需要写控制信号。加法器是一个 ALU，它计算两个 64 位输入的加法，并输出结果

要执行任意一条指令，首先要从存储器中取出指令。为准备执行下一条指令，必须增加程序计数器的值，使其指向下一条指令，即向后移动 4 个字节。图 4-6 所示为数据通路，它将图 4-5 中的三个单元组合起来，可以取出指令并增加 PC 以获得下一条指令的地址。

现在考虑 R 型指令（见图 2-19）。这类指令读两个寄存器，对它们的内容执行 ALU 操作，再将结果写回寄存器。这些指令被称为 R 型指令或算术逻辑指令（因为它们执行算术或逻辑运算）。这类指令包括第 2 章介绍的 add、sub、and 和 or 指令。回想一下，此类指令的典型实例是 `add x1, x2, x3`，它读取寄存器 x2 和 x3 并将总和写入 x1 寄存器。

处理器的 32 个通用寄存器位于被称为**寄存器堆**的结构中。寄存器堆是寄存器的集合，其中的寄存器可以通过指定相应的寄存器号来进行读写。寄存器堆包含了计算机的寄存器状态。另外，还需要一个 ALU 对从寄存器读出的值进行运算。

寄存器堆：包含一系列寄存器的状态单元，可通过所提供的寄存器号进行读写。

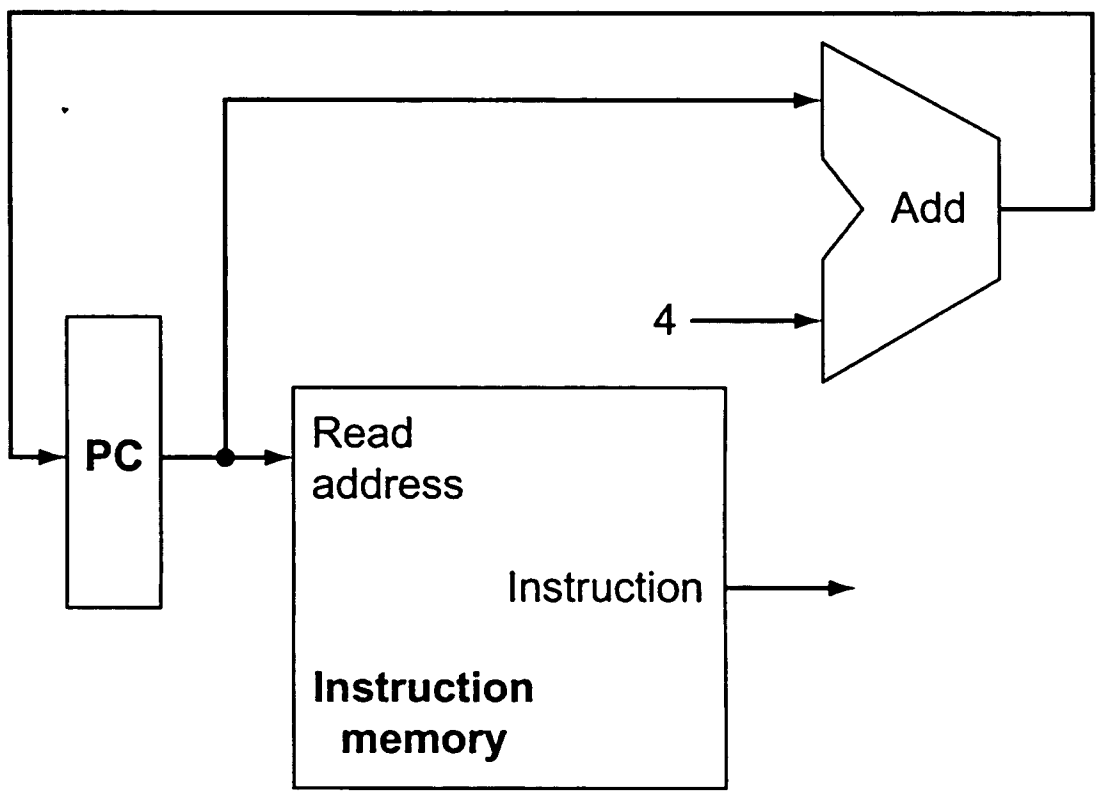


图 4-6　用于取出指令并增加程序计数器的部分数据通路。取出的指令供数据通路的其他部分使用

由于 R 型指令有三个寄存器操作数，每条指令需要从寄存器堆中读出两个数据字，再写入一个数据字。为读出一个数据字，需要一个输入指定要读的寄存器号，以及一个从寄存器堆读出的输出。为写入一个数据字，寄存器堆需要两个输入：一个输入指定要写的寄存器号，另一个提供要写入寄存器的数据。寄存器堆根据输入的寄存器号输出相应寄存器的内容。而写操作由写控制信号控制，在写操作发生的时钟边沿，写控制信号必须是有效的。如图 4-7 所示，我们总共需要四个输入（三个寄存器编号和一个数据）和两个输出（两个数据）。输入的寄存器号为 5 位宽，用于指定 32 个寄存器（$32 = 2^5$）中的一个，而数据输入总线和两个数据输出总线均为 64 位宽。

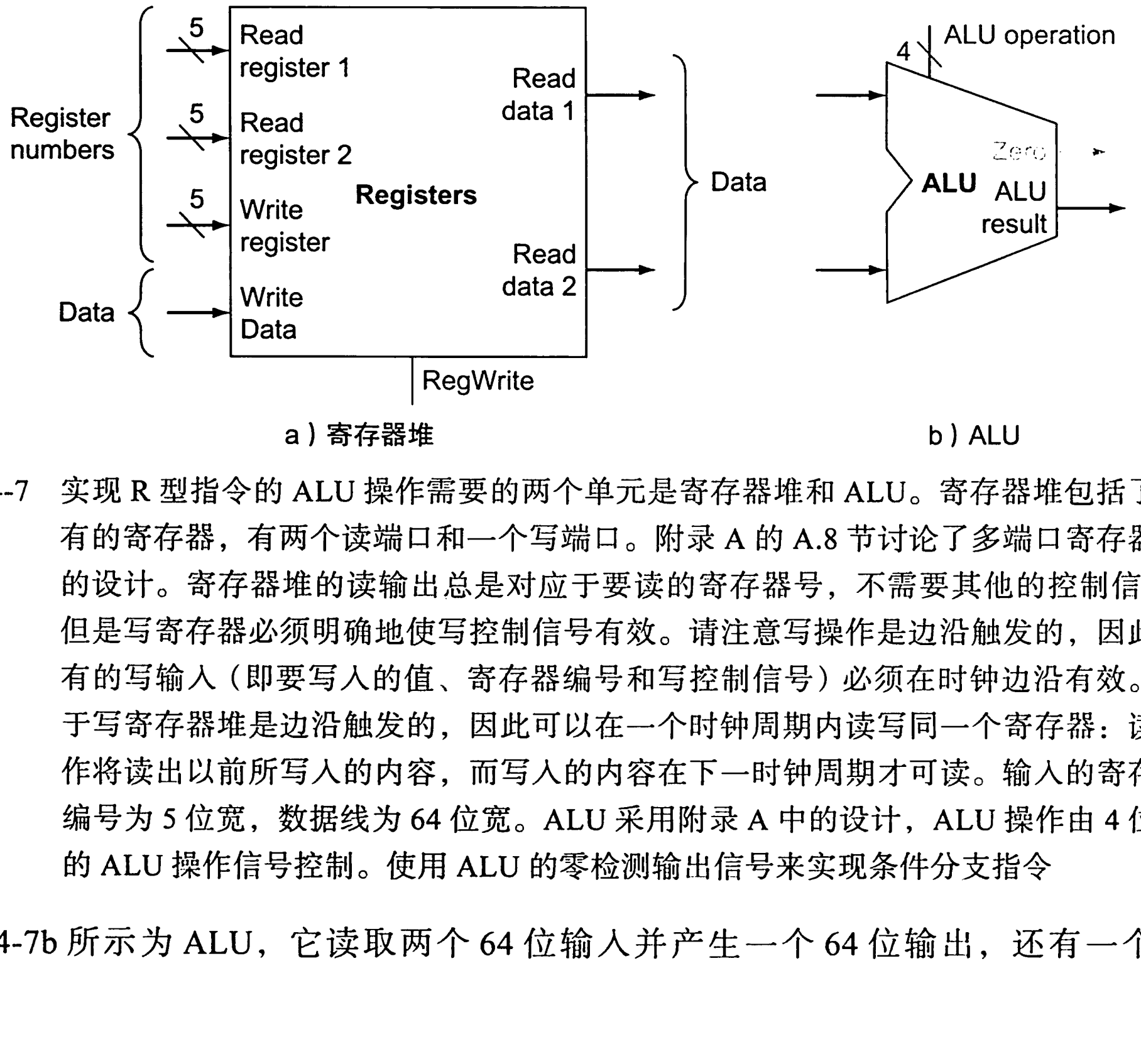


图 4-7　实现 R 型指令的 ALU 操作需要的两个单元是寄存器堆和 ALU。寄存器堆包括了所有的寄存器，有两个读端口和一个写端口。附录 A 的 A.8 节讨论了多端口寄存器堆的设计。寄存器堆的读输出总是对应于要读的寄存器号，不需要其他的控制信号。但是写寄存器必须明确地使写控制信号有效。请注意写操作是边沿触发的，因此所有的写输入（即要写入的值、寄存器编号和写控制信号）必须在时钟边沿有效。由于写寄存器堆是边沿触发的，因此可以在一个时钟周期内读写同一个寄存器：读操作将读出以前所写入的内容，而写入的内容在下一时钟周期才可读。输入的寄存器编号为 5 位宽，数据线为 64 位宽。ALU 采用附录 A 中的设计，ALU 操作由 4 位宽的 ALU 操作信号控制。使用 ALU 的零检测输出信号来实现条件分支指令

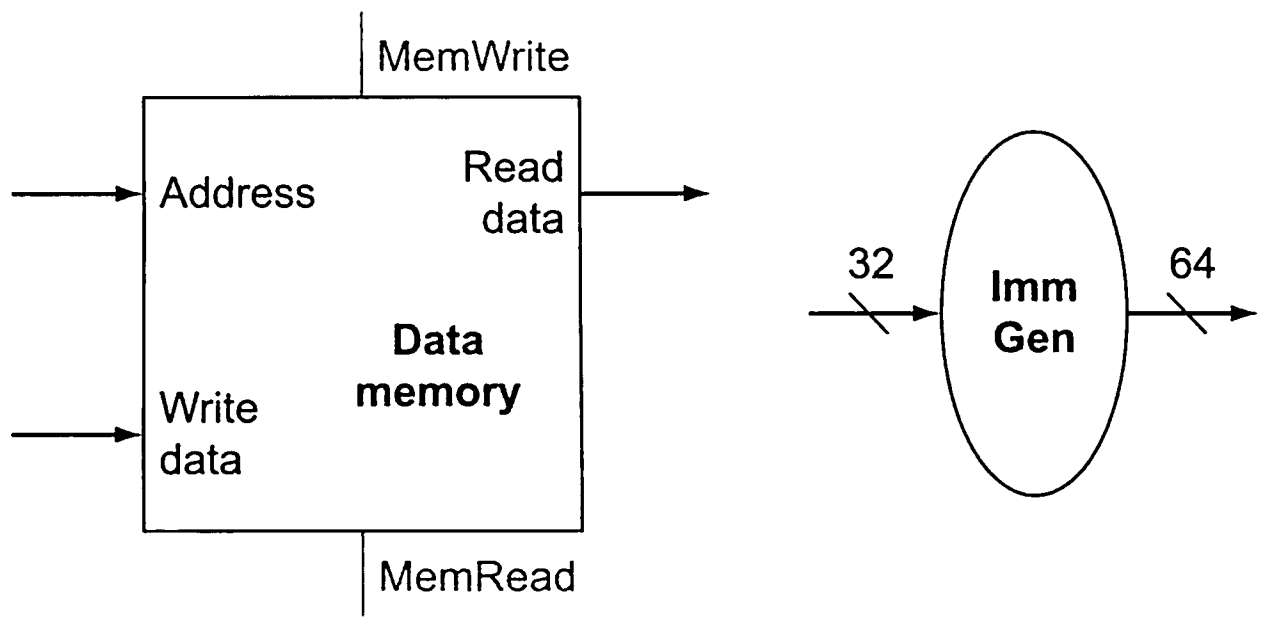
图 4-7b 所示为 ALU，它读取两个 64 位输入并产生一个 64 位输出，还有一个 1 位输

出指示其结果是否为0。附录A中详细描述了ALU的4位控制信号；在需要了解如何设置ALU控制信号时，将进行简要的回顾。

下面考虑RISC-V的存取指令，其一般形式为ld x1, offset(x2)或sd x1, offset(x2)。这类指令通过将基址寄存器x2与指令中包含的12位有符号偏移量相加，得到存储器地址。对于存储指令，从寄存器x1中读出要存储的数据。如果是载入指令，那么从存储器中读出的数据要写入指定的寄存器x1中。因此，图4-7中的寄存器堆和ALU都会被用到。

此外，还需要一个单元将指令中的12位偏移量**符号扩展**（sign-extend）为64位有符号数，以及一个执行读写操作的数据存储单元。数据存储单元在存储指令时被写入，所以它有读写控制信号、地址输入和写入存储器的数据输入。图4-8给出了这两个单元。

符号扩展：为增加数据的长度，将原数据的最高位复制到新数据多出来的高位。



a）数据存储单元　　b）立即数生成单元

图4-8　除图4-7的寄存器堆和ALU外，实现存储指令和载入指令还需要两个单元：数据存储单元和立即数生成单元。数据存储单元是一个状态单元，它有地址输入和写数据输入，以及读取结果的单个输出。读、写控制信号相互独立，但仅有一个可以在任意给定的时钟上有效。数据存储单元需要一个读信号，因为它与寄存器堆不同，读取无效地址处的值可能会导致问题，第5章中将看到这种情况。立即数生成单元（ImmGen）有一个32位指令的输入，如果是载入、存储和分支条件成立时的分支指令，它将指令中的一个12位字段符号扩展为64位结果输出（见第2章）。假定数据存储单元的写是边沿触发的。标准的存储芯片实际上有一个用于写操作的写使能信号。尽管标准存储器芯片的写使能信号不是边沿触发的，但我们的边沿触发设计很容易适用于真正的存储器芯片。关于实际存储器芯片工作细节的更多讨论见附录A的A.8节

beq指令有三个操作数，其中两个寄存器用于比较是否相等，另一个是12位偏移量，用于计算相对于分支指令所在地址的**分支目标地址**（branch target address）。它的指令格式是beq x1, x2, offset。为实现beq指令，需将PC值与符号扩展后的指令偏移量相加以得到分支目标地址。分支指令的定义（见第2章）中有两个必须注意的细节：

分支目标地址：分支指令中指定的地址，如果分支发生，该地址成为新的程序计数器的值。在RISC-V体系结构中，分支目标地址为该指令的偏移量字段与分支指令所在地址的和。

- 指令系统体系结构规定了计算分支目标地址的基址是分支指令所在地址。
- 指令系统体系结构还说明了计算分支目标地址时，将偏移量左移1位以表示半字为单位的偏移量，这样偏移量的有效范围就扩大到2倍。

为了处理这种复杂情况，需要将偏移量左移 1 位。

在计算分支目标地址的同时，必须确定是顺序执行下一条指令，还是执行分支目标地址处的指令。当分支条件为真（例如，两个操作数相等）时，分支目标地址成为新的 PC，我们就说**分支发生**。如果条件不成立，自增后的 PC 成为新的 PC（就像其他普通指令一样）；这时就说**分支未发生**。

分支发生：一种分支指令，其分支条件满足，程序计数器变为分支目标地址。所有无条件分支指令都是发生的分支。

分支未发生：一种分支指令，其分支条件不成立，程序计数器变为分支指令的下一条指令的地址。

因此，分支指令的数据通路需要执行两个操作：计算分支目标地址和检测分支条件。（很快将讲到，分支指令也会影响数据通路的取指部分。）图 4-9 为分支指令的数据通路。为计算分支目标地址，分支指令数据通路包含一个图 4-8 所示的立即数生成单元和一个加法器。为执行比较，需要图 4-7a 所示的寄存器堆提供两个寄存器操作数（不需要写入寄存器堆）。此外，使用附录 A 中的 ALU 完成相等性比较。由于该 ALU 提供一个表示结果是否为 0 的输出信号，故可将两个寄存器操作数发给 ALU，并将控制设置为减法。如果 ALU 输出的零信号有效，可知两个寄存器值相等。尽管零输出信号总是指示结果是否为 0，但我们仅用它来实现条件分支指令的相等测试。稍后将详细地介绍如何为数据通路中的 ALU 连接控制信号。

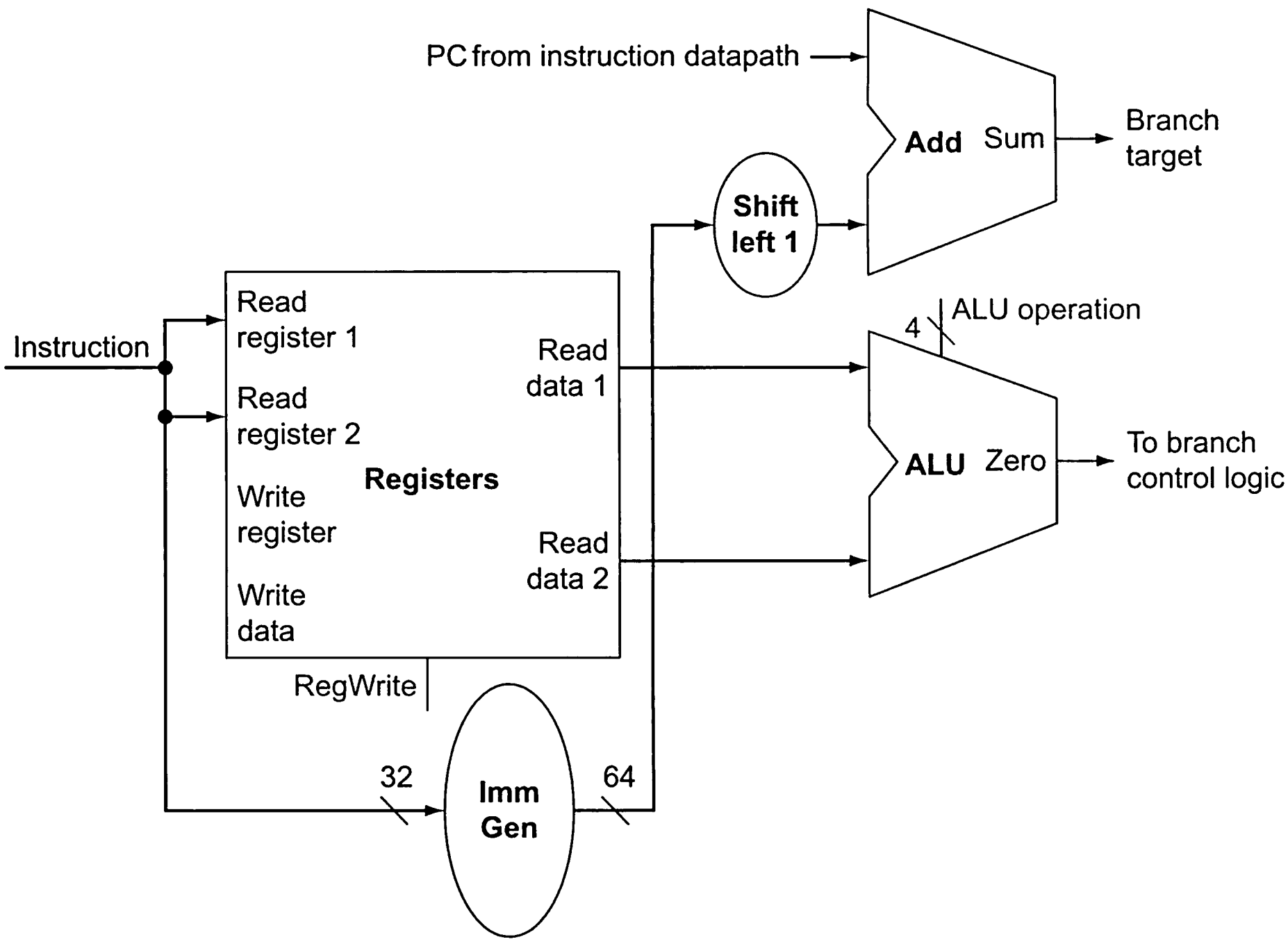


图 4-9　分支指令数据通路使用 ALU 检测分支条件是否成立，并使用单独的加法器将 PC 和符号扩展后左移一位的指令中的 12 位（分支偏移量）相加，以得到分支目标地址。标记 Shift left 1 的单元只是输入到输出之间一条简单的数据通路，它给符号扩展后的偏移量的低位加上一个 0_2；因为“移动”的距离是固定的，所以并不需要真正的移位电路。由于我们知道偏移量是从 12 位扩展而来的，所以移位只丢弃“符号位”。控制逻辑根据 ALU 的零输出信号来决定用自增的 PC 还是分支目标地址取代当前 PC

分支指令将指令中的 12 位偏移量左移一位与 PC 相加。如第 2 章所述，移位通过简单地给偏移量后面加上一个 0 实现。