## 建立一个简单的数据通路

我们已经分别讨论了几类指令需要的数据通路单元，现在可将它们组合成一个完整的数据通路并添加控制信号以完成实现。这个最简单的数据通路在每个时钟周期执行一条指令。这意味着每条指令在执行过程中的任何数据通路单元都只能使用一次，如果需要多次使用某数据通路单元，则要将其复制多份。因此，需要一个指令存储器和一个与之分开的数据存储器。尽管还有一些功能单元需要多份，但很多功能单元可以在不同的指令流动中被共享。

为在两个不同类指令之间共享数据通路单元，需要允许一个单元有多个输入，我们用多路选择器和控制信号在多个输入中进行选择。

**例题 | 建立数据通路**

算术逻辑（或 R 型）指令和存储类指令的数据通路非常相似。它们的主要区别如下：

- 算术逻辑指令使用 ALU 时，输入 ALU 的数据来自两个寄存器。存储类指令也使用 ALU 进行地址计算，但是第二个输入是对指令中 12 位偏移量进行符号扩展后的值。
- 存入目标寄存器的值来自 ALU（R 型指令）或存储器（载入指令）。

为存储类指令和算术逻辑指令的操作部分建立数据通路，只能使用一个寄存器堆和一个 ALU，并添加必要的多路选择器。

**答案** 为建立只有一个寄存器堆和一个 ALU 的数据通路，需支持 ALU 的第二个输入和要存入寄存器堆的数据都有两个不同的来源。因此，在 ALU 的输入端和寄存器堆的数据输入端分别添加一个多路选择器。图 4-10 给出了组合后的数据通路。

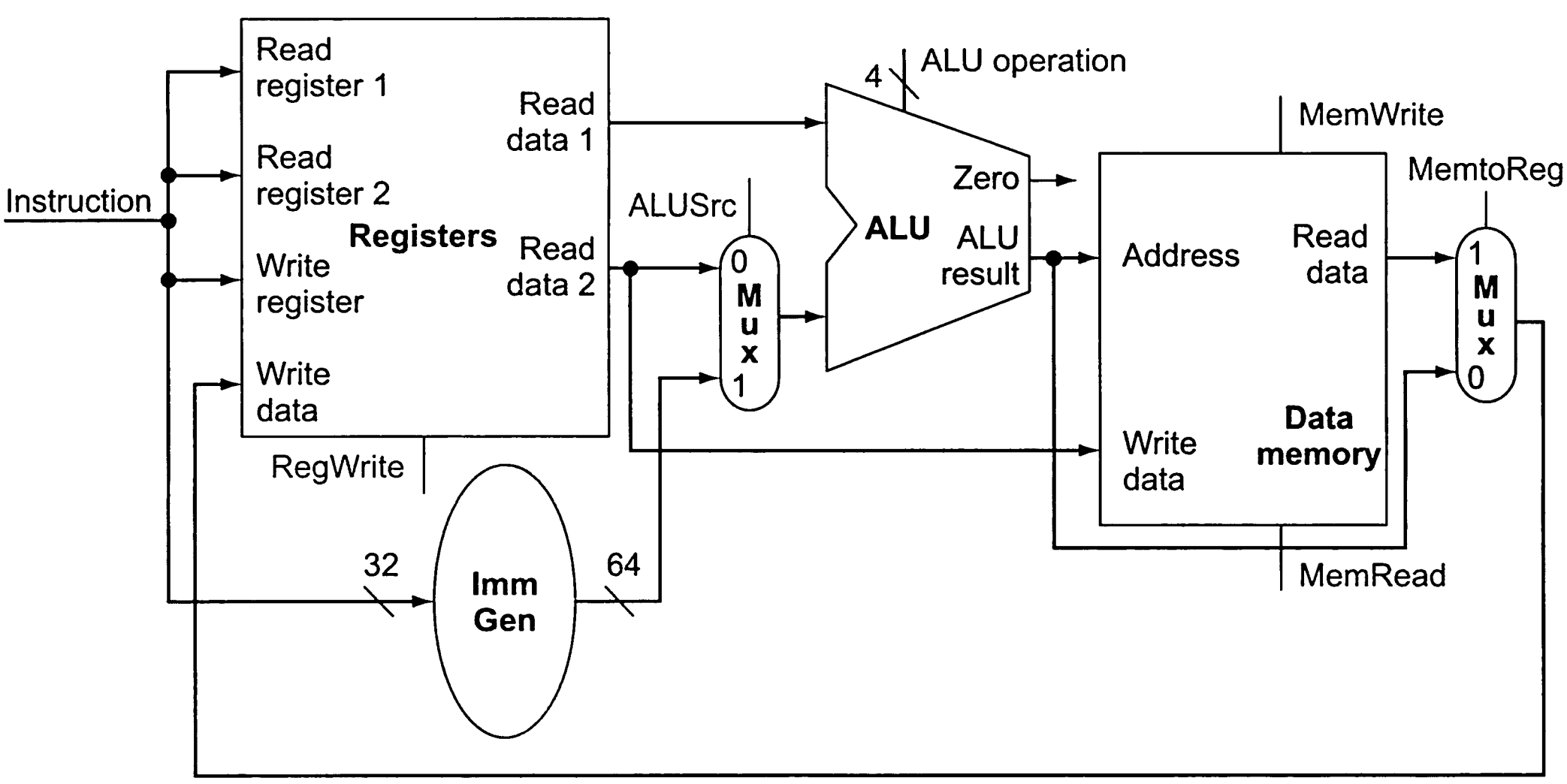


图 4-10 存储类指令和 R 型指令的数据通路。这个例子展示了如何通过添加多路选择器把图 4-7 和图 4-8 中的部分组合为一个数据通路。如上所述，增加了两个多路选择器

现在，把取指令数据通路（图 4-6）、R 型指令和存储类指令数据通路（图 4-10）、分支指令数据通路（图 4-9）合并，得到 RISC-V 指令系统核心集的一个简单数据通路，如图 4-11 所示。由于分支指令使用主 ALU 来比较两个寄存器操作数是否相等，所以要保留图 4-9 中的计算分支目标地址的加法器。增加一个多路选择器，用于选择是将顺序的指令地

址（PC+4）还是分支目标地址写入 PC。

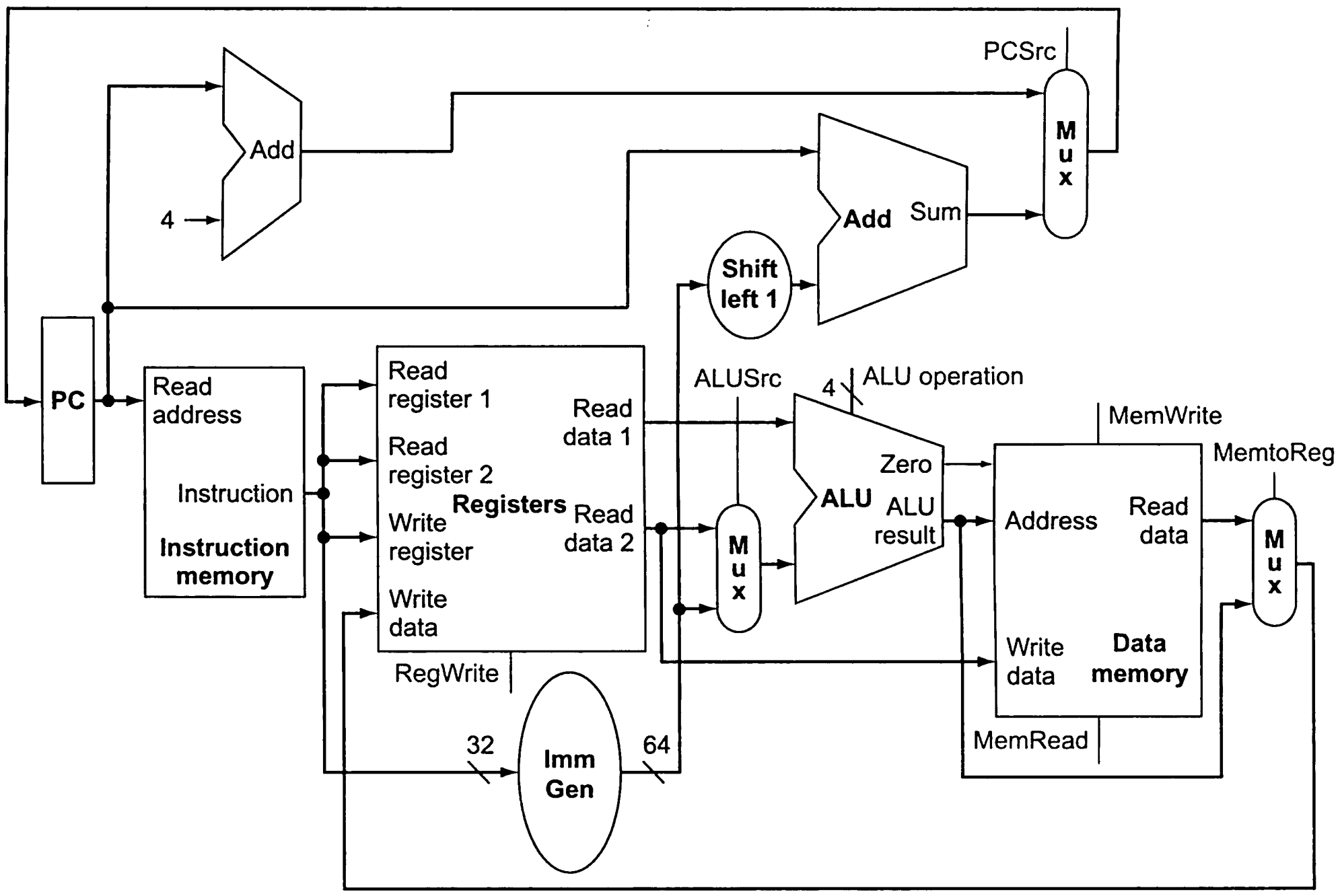


图 4-11　组合不同类指令所需的功能单元形成的 RISC-V 指令系统核心集的简单数据通路。图中的单元来自图 4-6、图 4-9 和图 4-10。该数据通路可以在一个时钟周期内执行基本指令（载入－存储寄存器、ALU 操作和分支）。为支持分支指令还增加了一个额外的多路选择器

现在已经完成了这个简单的数据通路，我们可以添加控制单元。控制单元必须能够接受输入并生成每个状态单元的写信号、每个多选器的选择信号和 ALU 的控制信号。由于 ALU 控制信号与其他控制很不一样，因此在设计控制单元的其他部分之前先设计 ALU 控制信号。

**详细阐述**　立即数生成逻辑选择指令中要进行符号扩展的 12 位字段：载入指令的 31:20 位、存储指令的 31:25 和 11:7 位、分支指令的 31、7、30:25 和 11:8 位。由于输入是完整的 32 位指令，因此可以使用指令的操作码位选择合适的字段。RISC-V 操作码的第 6 位在数据传输指令中为 0，且在分支指令中为 1；RISC-V 操作码的第 5 位在载入指令中为 0，且在存储指令中为 1。这样，操作码的第 5 和 6 位可以控制立即数生成逻辑内的一个 3:1 多路选择器的输出，为载入、存储和条件分支指令选择合适的 12 位字段。

**自我检测**

Ⅰ. 对载入指令来说，以下哪项是正确的？参考图 4-10。

a. MemtoReg 信号线应该被设置为将存储器中的数据发送至寄存器堆。

b. MemtoReg 信号线应该被设置为将正确的目标寄存器的数据发送至寄存器堆。

c. 对载入指令而言，MemtoReg 信号线的设置无关紧要。

Ⅱ. 本节描述的单周期数据通路必须有独立的指令存储器和数据存储器，因为：

a. RISC-V 中指令与数据的格式是不同的，所以需要不同的存储器。

b. 使用独立的存储器会比较便宜。

c. 处理器在一个周期内只能操作每个部件一次，而在一个周期内不可能对一个（单端口）存储器进行两次存取。

## 4.4 一个简单的实现方案

在本节中，我们学习 RISC-V 子集的一种简单实现。这个简单实现使用上一节中的数据通路并增加一个简单的控制单元来完成。它实现了指令 ld、sd、beq 以及算术逻辑指令 add、sub、and 和 or。

### 4.4.1 ALU 控制

附录 A 中的 RISC-V ALU 定义了四根输入控制线的以下四种组合：

| ALU控制线 | 功能 |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |

根据不同的指令类型，ALU 需执行以上四种功能中的一种。对于 load 和 store 指令，ALU 做加法计算存储器地址。对于 R 型指令，根据指令的 7 位 funct7 字段（位 31:25）和 3 位 funct3 字段（位 14:12）(参见第 2 章)，ALU 需执行四种操作（与、或、加、减）中的一种。对于条件分支指令，ALU 将两个操作数做减法并检测结果是否为 0。

4 位 ALU 的输入控制信号可由一个小型控制单元产生，其输入是指令的 funct7 和 funct3 字段以及 2 位的 ALUOp 字段。ALUOp 指明要执行的操作是 load 和 store 指令要做的加法（$00_2$），还是 beq 指令要做的减法并检测是否为 0（$01_2$），或是由 funct7 和 funct3 字段决定（$10_2$）。该控制单元输出一个 4 位信号，即前面介绍的 4 位组合之一来直接控制 ALU。

图 4-12 说明如何根据指令中的 2 位 ALUOp 控制字段、funct7 和 funct3 字段设置 ALU 的输入控制信号。在本章的后面将看到主控制单元如何生成 ALUOp。

| 指令操作码 | ALUOp | 操作 | funct7字段 | funct3字段 | ALU期望行为 | ALU控制输入 |
|---|---|---|---|---|---|---|
| ld | 00 | load doubleword | XXXXXXX | XXX | add | 0010 |
| sd | 00 | store doubleword | XXXXXXX | XXX | add | 0010 |
| beq | 01 | branch if equal | XXXXXXX | XXX | subtract | 0110 |
| R-type | 10 | add | 0000000 | 000 | add | 0010 |
| R-type | 10 | sub | 0100000 | 000 | subtract | 0110 |
| R-type | 10 | and | 0000000 | 111 | AND | 0000 |
| R-type | 10 | or | 0000000 | 110 | OR | 0001 |

图 4-12 根据 ALUOp 控制位和 R 型指令的操作码设置 ALU 的控制信号。第一列是指令，它决定了 ALUOp 位。所有的编码都以二进制给出。注意，当 ALUOp 为 $00_2$ 或 $01_2$ 时，ALU 操作不依赖于 funct7 或 funct3 字段；在这种情况下，“不关心”操作码的值，所以将其记为一串 X。当 ALUOp 为 $10_2$ 时，根据 funct7 和 funct3 字段来设置 ALU 的输入控制信号。见附录 A

这种多级译码的方式——主控制单元生成 ALUOp 位用作 ALU 的输入控制信号，再生成实际信号来控制 ALU——是一种常见的实现方式。多级控制可以减小主控制单元的规模。多个小的控制单元可能潜在地减小控制单元的延迟。这样的优化很重要，因为控制单元的延

迟是决定时钟周期的关键因素。

有几种不同的方法把 2 位 ALUOp 字段和 funct 字段映射到四位 ALU 输入控制信号。由于只有少数 funct 字段有意义，并且仅在 ALUOp 位等于 $10_2$ 时才使用 funct 字段，因此可以使用一个小逻辑单元来识别可能的取值并生成恰当的 ALU 控制信号。

为设计这个逻辑单元，有必要为 funct 字段和 ALUOp 信号的有意义组合生成一张**真值表**（truth table），如图 4-13 所示，该表给出了如何根据这些输入字段设置 4 位 ALU 输入控制信号。由于完整真值表非常大，我们并不关心所有的输入组合，所以只列出了使 ALU 控制信号有值的部分表项。在本章中，我们将一直采用这种方式列出真值表。（这样做的缺点在附录 C 的 C.2 节讨论。）

**真值表**：逻辑操作的一种表示方法，即列出输入的所有情况和每种情况下的输出。

| ALUOp | | funct7字段 | | | | | | | funct3字段 | | | 操作 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | [31] | [30] | [29] | [28] | [27] | [26] | [25] | [14] | [13] | [12] | |
| 0 | 0 | X | X | X | X | X | X | X | X | X | X | 0010 |
| X | 1 | X | X | X | X | X | X | X | X | X | X | 0110 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0010 |
| 1 | X | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0110 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0000 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0001 |

图 4-13　4 位 ALU 控制信号（称为操作）的真值表。输入是 ALUOp 和 funct 字段。仅示出使得 ALU 控制信号有效的条目，也包括一些无关项。例如，ALUOp 不使用 $11_2$ 编码，因此真值表包含条目 $1X_2$ 和 $X1_2$ 项，而不是 $10_2$ 和 $01_2$ 项。尽管 funct 字段的 10 位全部示出，但注意，对于四类 R 型指令而言，取值不同的只有 30、14、13 和 12 位。因此，只需要将这四位作为 ALU 控制的输入，而不是 funct 字段的全部 10 位

因为在很多情况下不关心某些输入的取值，为了简化真值表，我们也列出**无关项**。真值表中的无关项（在输入列中用 X 表示）表明输出不依赖于与该列对应的输入。例如，当 ALUOp 位为 $00_2$ 时（如图 4-13 的第一行），ALU 控制信号总被设置为 $0010_2$，而与 funct 字段无关。在这种情况下，真值表中此行的 funct 即为无关项。稍后将看到另一个无关项的例子。如果不熟悉无关项的概念，参见附录 A 以了解更多信息。

**无关项**：逻辑函数的一个元素，输出与所有输入的取值无关。无关项可以用不同的方式指定。

真值表构建好后，可对其进行优化并转化为门电路。这个过程是完全机械的。所以，将在附录 C 的 C.2 节中描述此过程和结果。

### 4.4.2　设计主控制单元

我们已经描述了如何使用操作码和 2 位信号作为输入进行 ALU 控制单元的设计，现在考虑控制的其他部分。在开始之前，首先看一条指令的各个字段和图 4-11 的数据通路所需的控制信号。为了理解如何将指令的各个字段与数据通路相连，需回顾四类指令的格式：算术、载入、存储和条件分支指令。见图 4-14。

RISC-V 的指令格式遵循以下规则：

- 正如第 2 章所述，**操作码**字段总是 0 ～ 6 位（opcode[6:0]）。根据操作码，funct3 字段（opcode[14:12]）和 funct7 字段

**操作码**：表示指令操作和格式的字段。

（opcode [31:25]）作为扩展的操作码字段。

- 对于 R 型指令和分支指令，第一个寄存器操作数始终在 15 ～ 19 位（opcode [19:15] rs1）。该字段也可用来定义载入和存储指令的基址寄存器。
- 对于 R 型指令和分支指令，第二个寄存器操作数始终在 20 ～ 24 位（opcode [24:20] rs2）。该字段也可用来定义存储指令中的寄存器，该寄存器保存了写入存储器的操作数。
- 对于分支指令、载入指令和存储指令，另一个操作数可以是 12 位立即数。
- 对于 R 型指令和载入指令，目标寄存器始终在 7 ～ 11 位（opcode [11:7] rd）。

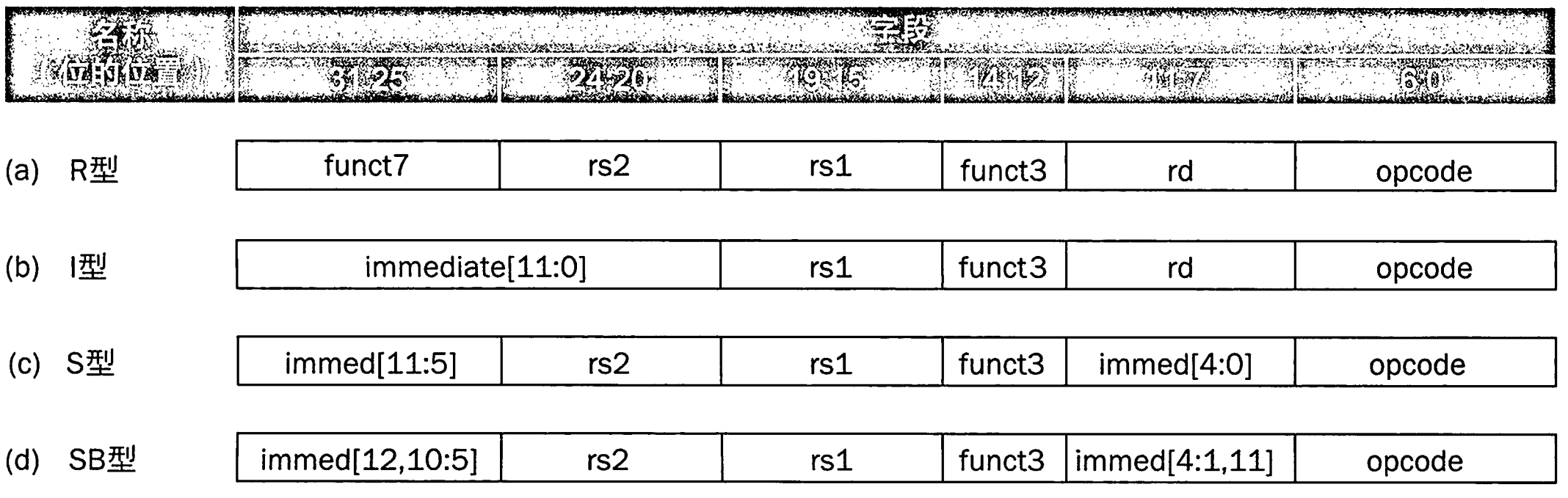


图 4-14 四类指令（算术、载入、存储和条件分支）使用的四类不同的指令格式。（a）R 型算术指令（操作码 = $51_{10}$）。寄存器操作数有三个：rs1、rs2 和 rd。字段 rs1 和 rs2 是源寄存器，rd 是目标寄存器。funct3 和 funct7 字段指示 ALU 功能，并由之前设计的 ALU 控制单元进行译码。我们需要实现的 R 型指令有 add、sub、and 和 or。（b）I 型载入指令（操作码 = $3_{10}$）。寄存器 rs1 是基址寄存器，它与 12 位立即数字段相加得到存储器地址。字段 rd 是目标寄存器，它存放从存储器中读出的值。（c）S 型存储指令（操作码 = $35_{10}$）。寄存器 rs1 是基址寄存器，与 12 位立即数字段相加得到存储器地址（立即数字段在指令编码中被分成 7 位和 5 位两部分）。字段 rs2 是源寄存器，它的值应存入存储器中。（d）SB 型条件分支指令（操作码 = $99_{10}$）。寄存器 rs1 和 rs2 进行比较。将 12 位立即数地址字段进行符号扩展，再左移 1 位，与 PC 相加得到分支目标地址

从第 2 章得到的第一个设计原则——简单源于规整——在这里就得到了印证。

根据这些信息，可为简单的数据通路添加指令标记，图 4-15 给出了这些增加的单元和 ALU 控制模块、状态单元的写信号、数据存储器的读信号以及多路选择器的控制信号。由于所有多路选择器都有两个输入，每个多路选择器都需要一条单独的控制线。

图 4-15 给出了 6 根 1 位控制线和 2 位 ALUOp 控制信号。我们已经定义了 ALUOp 控制信号如何工作，在确定指令执行过程中如何设置这些控制信号之前，应先非正式地定义其他六个控制信号如何工作。图 4-16 描述了这 6 根控制线的功能。

我们已经了解各个控制信号的功能，再来看看它们如何设置。除 PCSrc 控制信号外，所有的控制信号可由控制单元仅根据指令的操作码和 funct 字段设置。PCSrc 控制线是例外。若指令是 branch if equal（由控制单元确定）并且做相等检测的 ALU 的零输出有效，那么 PCSrc 控制信号有效。为生成 PCSrc 信号，需要将来自控制单元（称为“Branch”）的信号与来自 ALU 的零输出信号相“与”。

这 8 个控制信号（图 4-16 中的 6 个和 ALUOp 中的 2 个）可根据控制单元的输入信号

（即操作码的6：0位）进行设置。图4-17给出了包含控制单元和控制信号的数据通路。

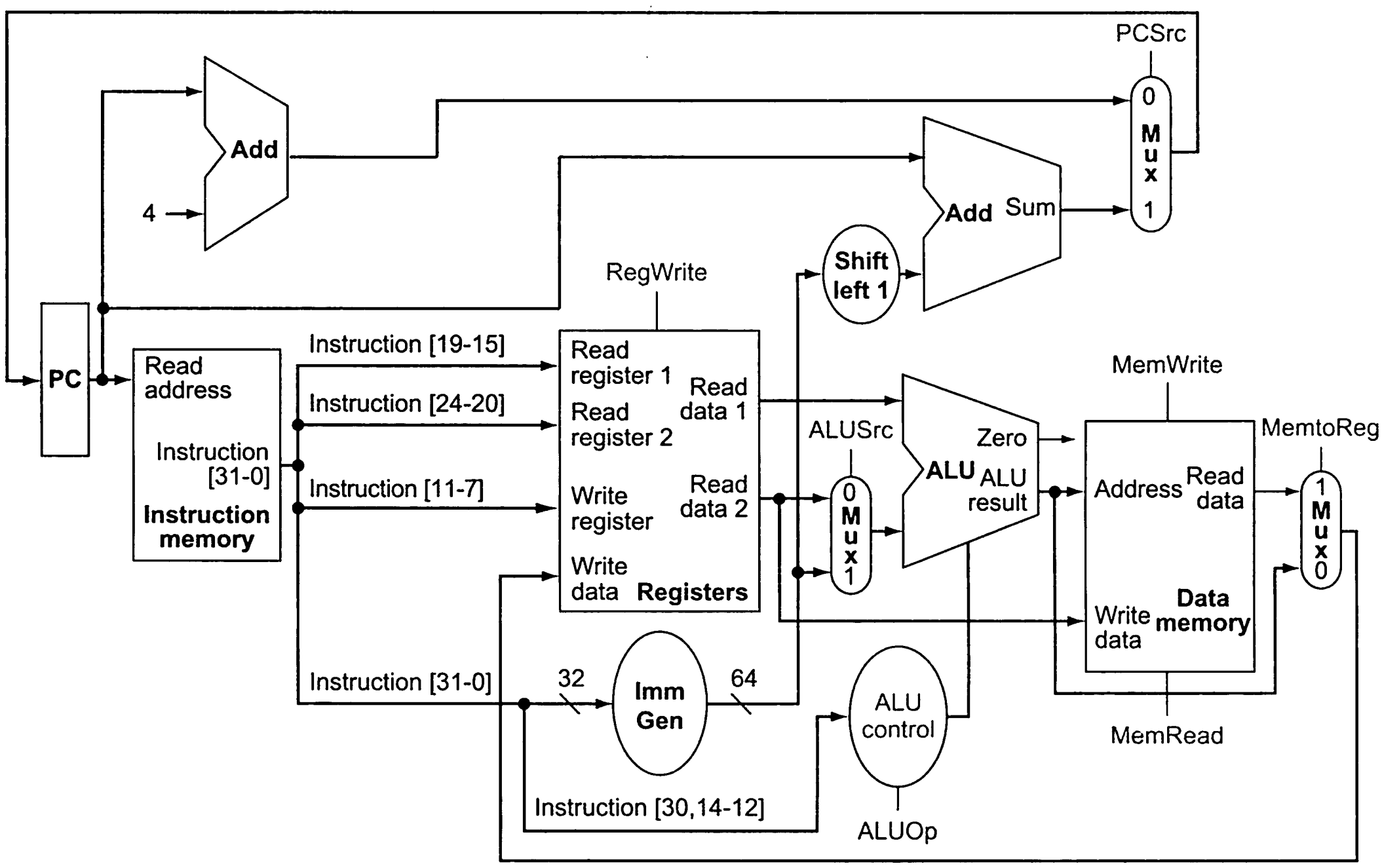


图4-15　为图4-11数据通路图添加了所有必要的多路选择器，并标识了所有控制线。控制线以灰色示出。也添加了ALU控制块，它取决于funct3字段和部分funct7字段。PC不需要写控制信号，因为它在每个时钟周期结束时被写入一次；分支控制逻辑确定PC是自增的PC还是分支目标地址

| 信号名 | 无效时的效果 | 有效时的效果 |
|---|---|---|
| RegWrite | 无 | 被写的寄存器号来自Write register信号的输入，数据来自Write data信号的输入 |
| ALUSrc | 第二个ALU操作数来自第二个寄存器堆的输出（即Read data 2信号的输出） | 第二个ALU操作数是指令的低12位符号扩展 |
| PCSrc | PC值被adder的输出所替换，即PC+4的值 | PC值被adder的输出所替换，即分支目标 |
| MemRead | 无 | 读地址由Address信号的输入指定，输出到Read data信号的输出中 |
| MemWrite | 无 | 写地址由Address信号的输入指定，写入内容是Write data信号的输入中的值 |
| MemtoReg | 寄存器写数据的输入值来自ALU | 寄存器写数据的输入值来自数据存储器 |

图4-16　6个控制信号的功能。当两路多选器的1位控制信号有效时，多选器选择对应于1的输入。否则选择对应于0的输入。请记住，所有状态单元都将时钟信号作为隐式输入，而且时钟控制写操作。时钟信号从来不在状态单元之外通过任何门电路，因为这样可能导致时序问题（附录A对此问题有进一步讨论）

在设计控制单元的计算公式或真值表之前，应非正式地定义控制功能。由于控制信号的设置仅取决于操作码，我们需要定义每个控制信号在每个操作码的取值下是0、1或无关（X）。根据图4-12、图4-16和图4-17，图4-18定义了对应于每种操作码的控制信号。

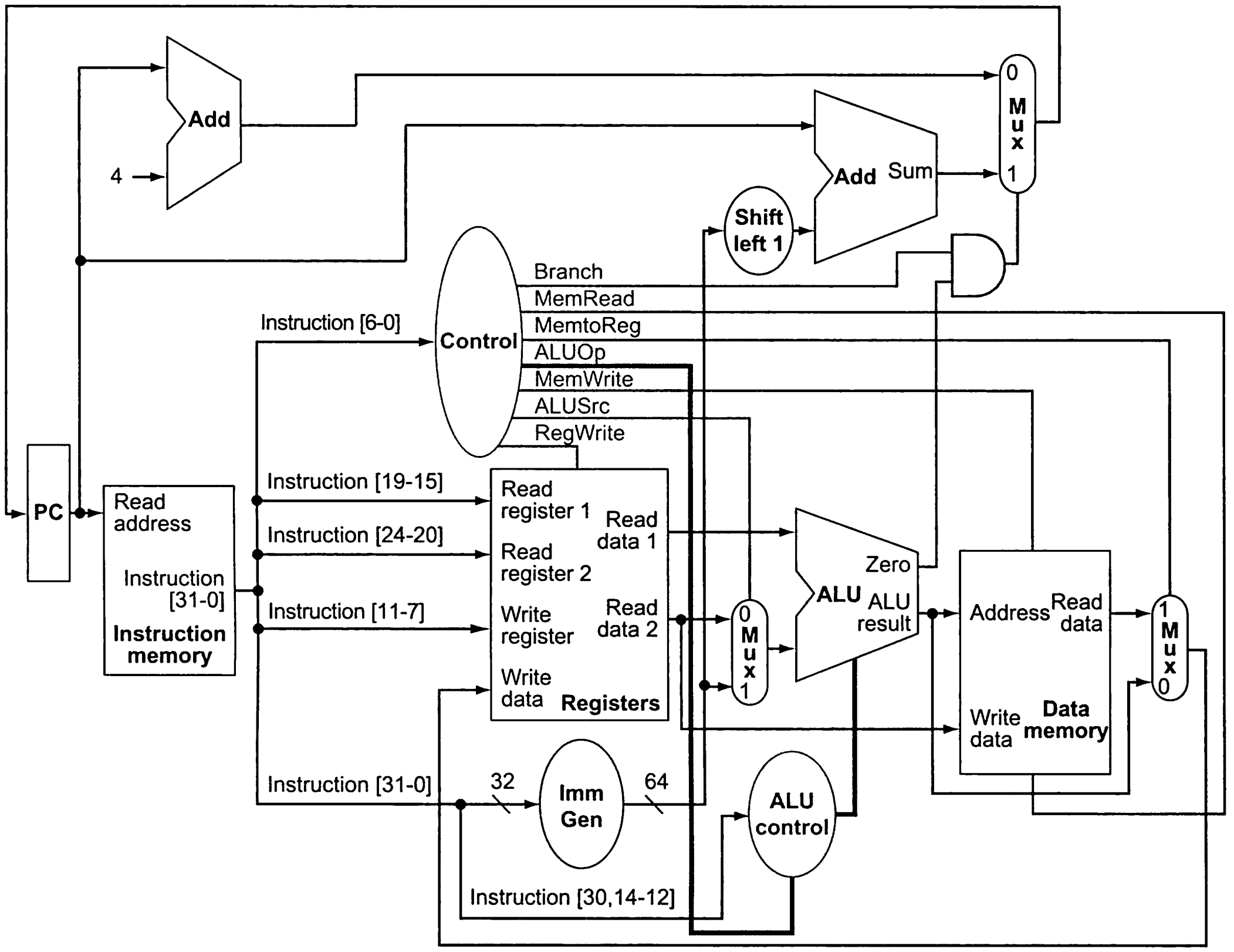


图 4-17 带有控制单元的简单数据通路。控制单元的输入是指令的 7 位操作码字段，输出包含两个控制多路选择器的 1 位信号（ALUSrc 和 MemtoReg）、三个控制寄存器堆和数据存储器读写的信号（RegWrite、MemRead 和 MemWrite）、一个确定是否分支的 1 位信号（Branch）和一个 ALU 的 2 位控制信号（ALUOp）。分支控制信号与 ALU 的零输出信号一起送入一个与门，其输出控制下一个 PC 的选择。注意 PCSrc 是衍生信号，不是直接来自控制单元。因此在图中我们没有标出这个信号名称

| 指令 | ALUSrc | MemtoReg | RegWrite | MemRead | MemWrite | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|
| R型 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| ld | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sd | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

图 4-18 控制线的设置完全取决于指令的操作码字段。表格中第一行对应 R 型指令（add、sub、and 和 or 指令）。源寄存器字段为 rs1 和 rs2，目标寄存器字段为 rd；这决定了信号 ALUSrc 如何设置。此外，R 型指令写入寄存器（RegWrite = 1），但不读写数据存储器。当 Branch 控制信号为 0 时，PC 无条件地由 PC + 4 取代；否则，如果 ALU 的零输出也为高，则 PC 由分支目标地址取代。R 型指令的 ALUOp 字段为 10，表示 ALU 控制信号应由 funct 字段生成。该表的第二行和第三行给出了 ld 和 sd 的控制信号。ALUSrc 和 ALUOp 字段被设置为执行地址计算。MemRead 和 MemWrite 被设置为执行存储器访问。最后，为 load 指令设置 RegWrite，将结果存入 rd 寄存器中。分支指令的 ALUOp 字段设置为减法（ALU 控制信号 = 01），用于测试相等性。请注意，当 RegWrite 信号为 0 时，MemtoReg 字段无关紧要：因为寄存器没有被写入，寄存器写端口的数据值不被使用，所以最后两行中的 MemtoReg 值由于不被关心而被 X 取代。这种无关项设计必须由设计者加入，因为其依赖于对数据通路工作原理的了解

### 4.4.3　数据通路操作

根据图 4-16 和图 4-18 包含的信息，可以设计控制单元的逻辑，但在这之前先了解每条指令是如何使用数据通路的。接下来的几张图说明了三类不同指令在数据通路中的流动。有效的控制信号和数据通路单元已标出。请注意，多选器在控制信号为 0 时也有相应的动作，即使其控制信号没有着重标出。对于多位信号，只要其中任何信号有效，就着重标出。

图 4-19 所示为 R 型指令的数据通路操作，例如 add x1,x2,x3。虽然所有操作都发生在一个时钟周期内，但我们认为执行该指令共分为四个步骤，这些步骤按照信息的流动排序：

1. 取出指令，PC 自增。
2. 从寄存器堆读出两个寄存器 x2 和 x3，同时主控制单元在此步骤计算控制信号。
3. 根据部分操作码确定 ALU 的功能，对从寄存器堆读出的数据进行操作。
4. 将 ALU 的结果写入寄存器堆中的目标寄存器（x1）。

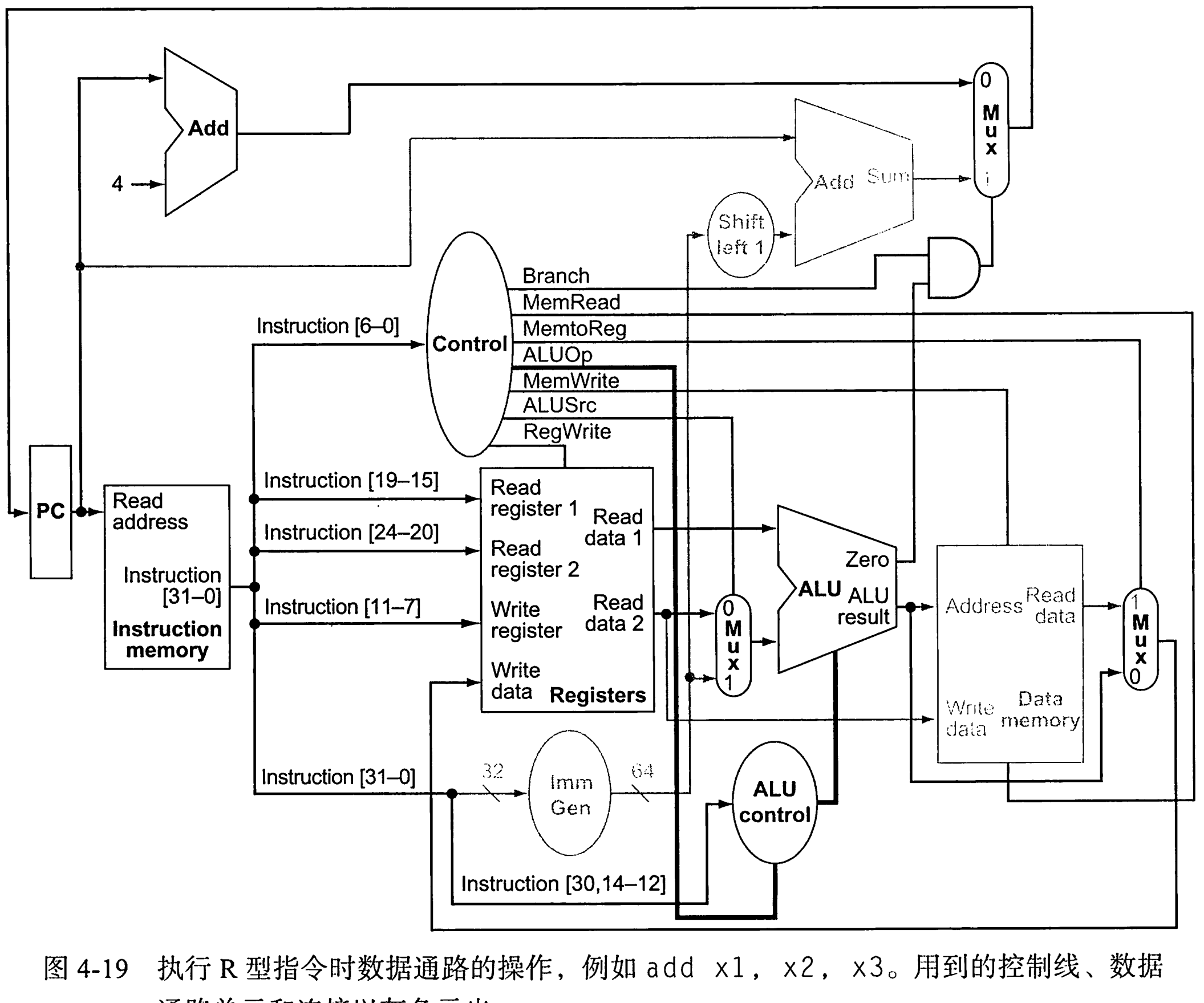


图 4-19　执行 R 型指令时数据通路的操作，例如 add x1，x2，x3。用到的控制线、数据通路单元和连接以灰色示出

用类似图 4-19 的方式，可以说明 load 指令的执行，例如 ld x1，offset(x2)。图 4-20 所示为在 load 指令执行过程中有效的功能单元和控制线。可将 load 指令的执行分为五个步骤（与 R 型指令分四步执行类似）：

1. 从指令存储器中取出指令，PC 自增。
2. 从寄存器堆读出寄存器（x2）的值。
3. ALU 将从寄存器堆中读出的值和符号扩展后的指令中的 12 位（偏移量）相加。
4. 将 ALU 的结果用作数据存储器的地址。
5. 将从存储器读出的数据写入寄存器堆（x1）。

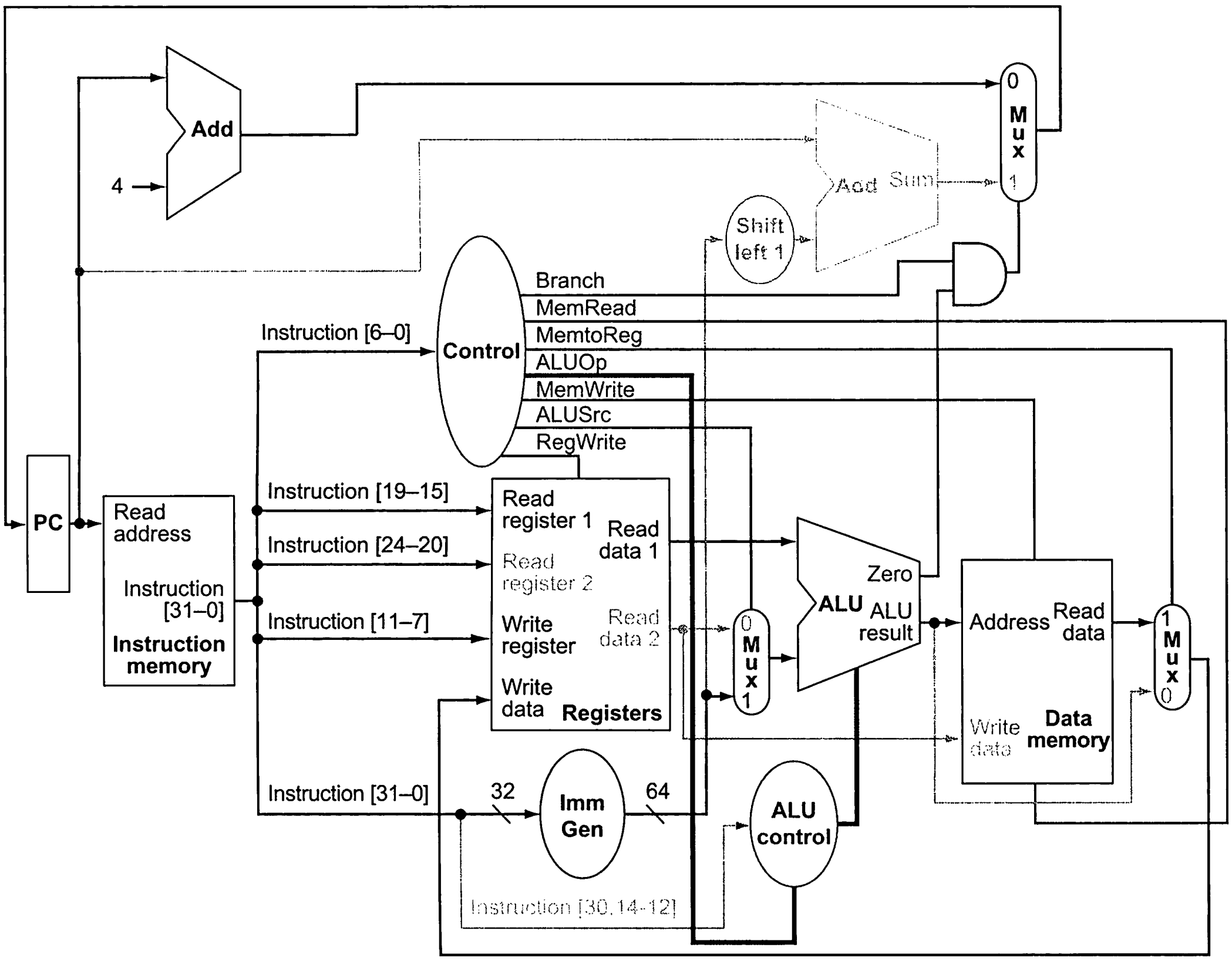


图 4-20　执行 load 指令时数据通路的操作。用到的控制线、数据通路单元和连接着重标记。store 指令的操作与之相似。主要区别在于存储器控制将指明操作是写而不是读，读出的第二个寄存器的值将作为要存储的数据，并且不存在将数据存储器的内容写入寄存器堆的操作

最后，用相同的方式说明 branch-if-equal 指令的操作，例如 `beq x1, x2, offset`。它的操作与 R 型指令非常相似，但 ALU 的输出被用来确定 PC 由 PC+4 还是分支目标地址写入。图 4-21 所示为执行的四个步骤：

1. 从指令存储器中取出指令，PC 自增。
2. 从寄存器堆中读出两个寄存器 x1 和 x2。
3. ALU 将从寄存器堆读出的两数相减。PC 与左移一位、符号扩展的指令中的 12 位（偏移）相加，结果是分支目标地址。
4. ALU 的零输出决定将哪个加法器的结果写入 PC。

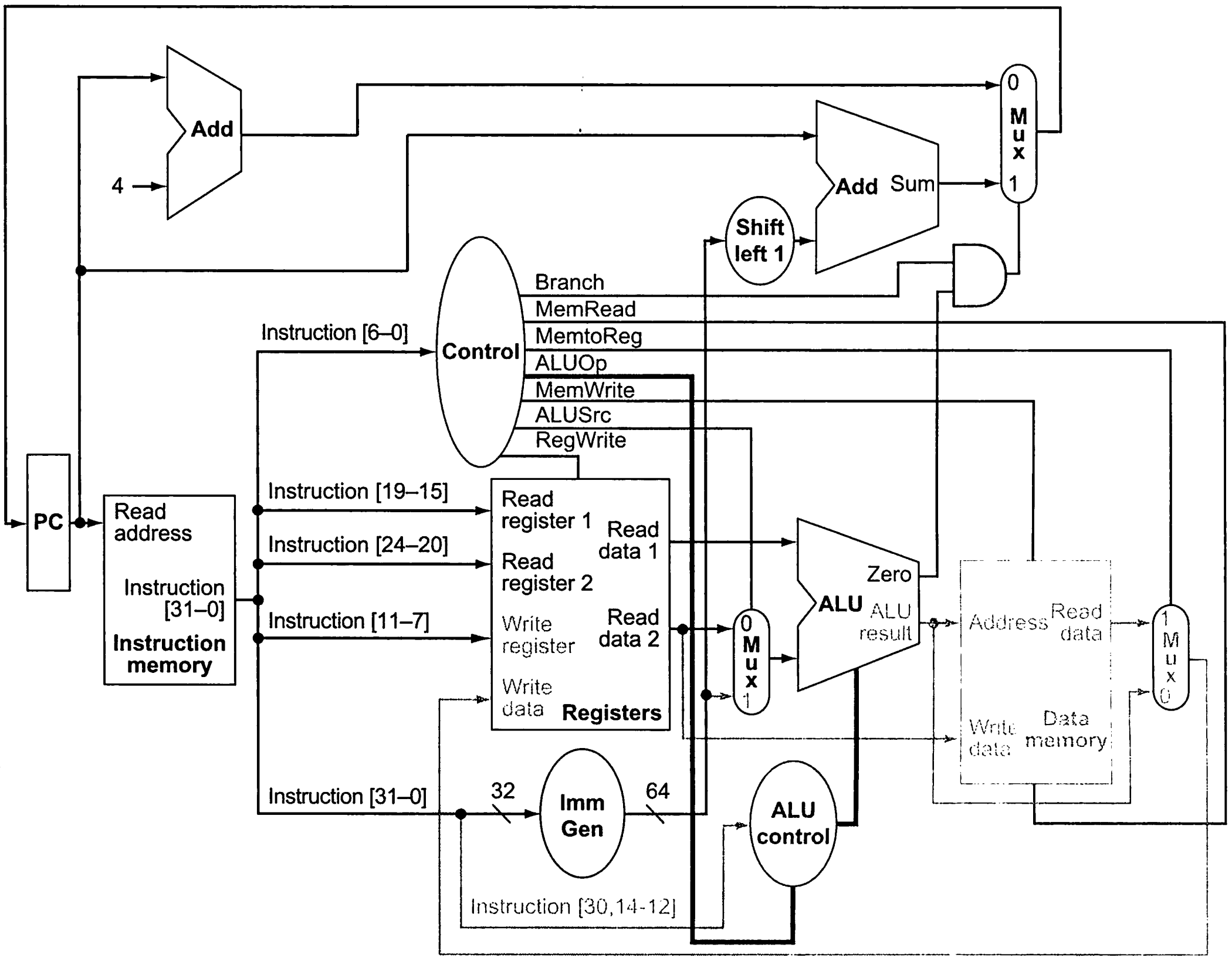


图 4-21 执行 branch-if-equal 指令时数据通路的操作。用到的控制线、数据通路单元和连接着重标出。在寄存器堆和 ALU 执行比较操作之后，零输出从两个可能的 PC 之间选择下一个 PC

### 4.4.4 控制的结束

我们已经了解了指令如何按步骤操作，现在继续讨论控制单元的实现。控制单元的功能可根据图 4-18 的内容进行精确定义，其输出是控制线，输入是几位操作码。因此，可以根据操作码的二进制编码为每个输出建立一个真值表。

图 4-22 将控制单元的逻辑定义为一个大的真值表，它将所有输出与输入组合在一起，输入为操作码，并且完整地描述了控制单元的功能，可以自动地转换为门电路实现，附录 C 中的 C.2 节描述了这个步骤。

### 4.4.5 为什么现在不使用单周期实现

尽管单周期设计可以正确工作，但是在现代设计中不采取这种方式，因为它的效率太低。究其原因，是在单周期设计中时钟周期对于每条指令必须等长。这样，处理器中的最长路径决定了时钟周期。这条路径很可能是一条 load 指令，它连续地使用 5 个功能单元：指令存储器、寄存器堆、ALU、数据存储器和寄存器堆。虽然 CPI 为 1（见第 1 章），但由于时钟周期太长，单周期实现的整体性能可能很差。

使用单周期设计的代价是显著的，但对于这个小指令集而言，或许是可以接受的。历史

上，早期具有简单指令集的计算机确实采用这种实现方式。但是，如果要实现浮点单元或更复杂的指令集，单周期设计根本无法正常工作。

| 输入或输出 | 信号名称 | R型 | lw | sw | beq |
|---|---|---|---|---|---|
| 输入 | I[6] | 0 | 0 | 0 | 1 |
| | I[5] | 1 | 0 | 1 | 1 |
| | I[4] | 1 | 0 | 0 | 0 |
| | I[3] | 0 | 0 | 0 | 0 |
| | I[2] | 0 | 0 | 0 | 0 |
| | I[1] | 1 | 1 | 1 | 1 |
| | I[0] | 1 | 1 | 1 | 1 |
| 输出 | ALUSrc | 0 | 1 | 1 | 0 |
| | MemtoReg | 0 | 1 | X | X |
| | RegWrite | 1 | 1 | 0 | 0 |
| | MemRead | 0 | 1 | 0 | 0 |
| | MemWrite | 0 | 0 | 1 | 0 |
| | Branch | 0 | 0 | 0 | 1 |
| | ALUOp1 | 1 | 0 | 0 | 0 |
| | ALUOp0 | 0 | 0 | 0 | 1 |

图 4-22 真值表完整地描述了简单的单周期实现的控制功能。上半部分给出了与四类指令（每列一类）相对应的输入信号的组合，决定了输出如何设置。下半部分给出了对应四种操作码的输出。例如，在两种不同的输入情况下，输出信号 RegWrite 有效。如果只考虑表中的四种操作码，那么可以使用输入部分的无关项来简化真值表。例如，可以用表达式 Op4 · Op5 检测 R 型指令，因为这足以将 ld、sd 和 beq 指令与 R 型指令区分开。由于在 RISC-V 的完整实现中会用到操作码的其余部分，所以我们没有使用这种简化

由于时钟周期必须满足所有指令中最坏的情况，所以不能使用那些缩短常用指令执行时间而不改变最坏情况的实现技术。因此，单周期实现违反了第 1 章中加速经常性事件这一设计原则。

在下一节，我们将看到一种称为流水线的实现技术，它使用与单周期相似的数据通路，但吞吐量更高，效率更高。流水线技术通过同时执行多条指令来提高效率。

**自我检测** 回顾图 4-22 中的控制信号。你能将它们结合在一起吗？图中的控制信号可以被其他控制信号取反来代替吗？（提示：考虑无关项。）如果有，不加反相器是否可以直接用一个控制信号替代另一个呢？

## 4.5 流水线概述

*决不浪费时间。*
*美国谚语*

**流水线**是一种能使多条指令重叠执行的实现技术。目前，流水线技术广泛应用。

**流水线**：一种实现多条指令重叠执行的技术，与生产流水线类似。

本节用一个比喻概述流水线的概念及相关问题。如果只想了解流水线的主要内容，应详细阅读本节后跳到 4.10 节和 4.11 节学习最

新处理器（如 Intel Core i7 和 ARM Cortex-A53）中使用的高级流水线技术。如果想深入了解流水线计算机，4.6 ～ 4.9 节做了详细介绍。

任何做洗衣工作的人都不自觉地使用流水线技术。非流水线的洗衣过程包含如下步骤：

1. 将一批脏衣服放入洗衣机。
2. 洗衣机洗完后，将湿衣服取出并放入烘干机。
3. 烘干机完成后，将干衣服取出，放在桌上并叠起来。
4. 叠好后，请你的室友帮忙把衣服收好。

你的室友把衣服收好后，再开始洗下一批脏衣服。

流水线方法花费的时间少得多，如图 4-23 所示。当第一批衣服从洗衣机中取出并放入烘干机后，就可以把第二批脏衣服放入洗衣机。当第一批衣服烘干完成后，就可以把它们放在桌上叠起来，同时把洗衣机中洗好的衣服放入烘干机，再将下一批脏衣服放入洗衣机。接着让你的室友把第一批衣服从桌上收好，你开始叠第二批衣服，烘干机开始烘干第三批衣服，同时可以把第四批衣服放入洗衣机。此时，所有的洗衣步骤（称为流水线阶段）在同时工作。只要每个阶段使用不同的资源，我们就可以用流水线的方法完成任务。

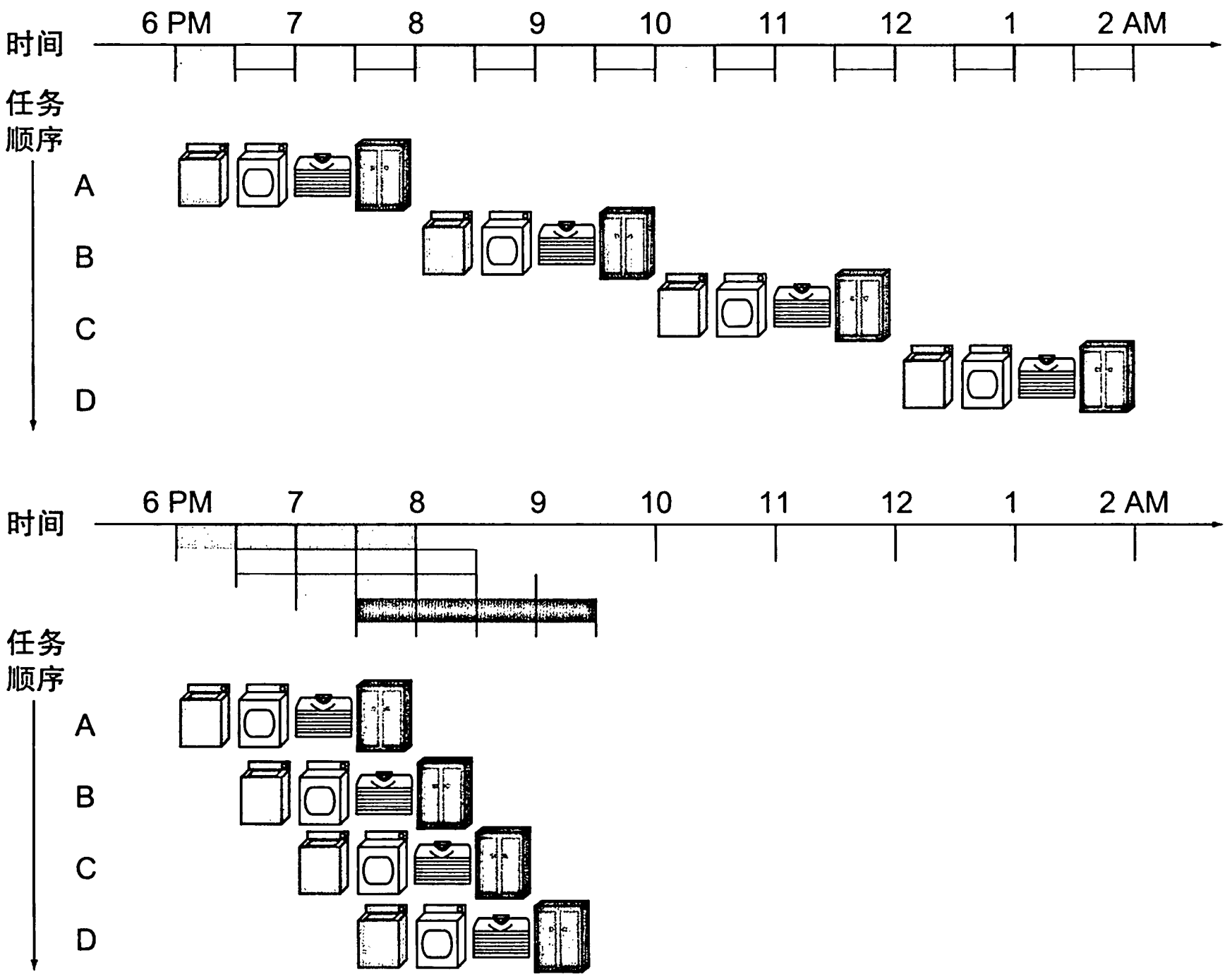


图 4-23　以洗衣服为例类比流水线。安、布莱恩、凯茜和唐各自都有脏衣服要清洗、烘干、折叠和收起。洗衣机、烘干机、“折叠机”和“收衣机”每台机器需要 30 分钟完成各自任务，顺序的洗衣方法需要 8 小时洗完 4 批衣服，而流水线洗衣方法只需 3.5 小时。图中在二维时间轴中通过资源的 4 次复制表明不同工作负载的流水线阶段，但实际上每种资源只有一份

流水线的矛盾在于，对于一双脏袜子，从把它放入洗衣机到被烘干、叠好和收起的时间在流水线中并没有缩短；然而对于许多负载来说，流水线更快的原因是所有工作都在并行地执行。所以单位时间能够完成更多工作，流水线提高了洗衣系统的吞吐率（throughput）。因

此，流水线不会缩短洗一次衣服的时间，但是当有很多衣物需要洗时，吞吐率的提高减少了完成整个任务的时间。

如果每个步骤需要的时间相同，并且要完成的工作足够多，那么由流水线产生的加速比等于流水线中步骤的数目，在这个例子中是 4 倍：洗涤、烘干、折叠和收起。因此，流水线方式洗衣是非流水线方式洗衣速度的 4 倍：流水线中 20 次洗衣需要的时间是一次洗衣的 5 倍，而 20 次非流水线洗衣的时间是一次洗衣的 20 倍。图 4-23 中流水线方式的速度仅为非流水线方式的 2.3 倍，因为图中只包括 4 次洗衣过程。注意到图 4-23 流水线中的工作负载在开始和结束时，流水线并未完全充满；当任务数量与流水线的步骤数量相比不是很大时，流水线的启动和结束会影响它的性能。在本例中，如果负载的数量远远大于 4，那么流水线步骤在大部分时间是充满的，吞吐率的增加接近 4 倍。

同样的原则也可用于处理器，即采用流水线方式执行指令。RISC-V 指令执行通常包含五个步骤：

1. 从存储器中取出指令。
2. 读寄存器并译码指令。
3. 执行操作或计算地址。
4. 访问数据存储器中的操作数（如有必要）。
5. 将结果写入寄存器（如有必要）。

因此，本章探讨的 RISC-V 流水线有五个阶段。正如流水线加速洗衣过程一样，下面的例子将说明流水线如何加速指令执行。

**例题　单周期实现与流水线性能**

为了使讨论具体化，我们先建立一个流水线。在本例和本章的其余部分，我们只考虑这七条指令：双字载入（ld）、双字存储（sd）、加（add）、减（sub）、与（and）、或（or）和相等就跳转（beq）指令。

本例将单周期指令执行（每条指令执行需要一个时钟周期）与流水线指令执行的平均执行时间进行对比。假设在本例中主要功能单元的操作时间为：指令或数据存储器访问为 200ps，ALU 操作为 200ps，寄存器堆的读或写为 100ps。在单周期模型中，每条指令的执行需要一个时钟周期，所以时钟周期必须满足最慢的指令。

**答案**　图 4-24 所示为七条指令中每条指令所需的执行时间。单周期设计必须满足最慢的指令——图 4-24 中是 ld 指令，所以每条指令所需的执行时间是 800ps。类似图 4-23，图 4-25 比较了三条载入寄存器指令的非流水线方式和流水线方式的执行过程。因此，在非流水线设计中，第一条和第四条指令之间的时间为 3 × 800ps=2400ps。

| 指令类型 | 取指令 | 寄存器 | ALU 操作 | 数据存取 | 寄存器 | 总时间 |
|---|---|---|---|---|---|---|
| Load doubleword (ld) | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| Store doubleword (sd) | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-format (add, sub,and, or) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| Branch (beq) | 200 ps | 100 ps | 200 ps | | | 500 ps |

图 4-24　根据各功能单元所需时间计算出的每条指令执行总时间。假定多路选择器、控制单元、PC 访问和符号扩展单元没有延迟

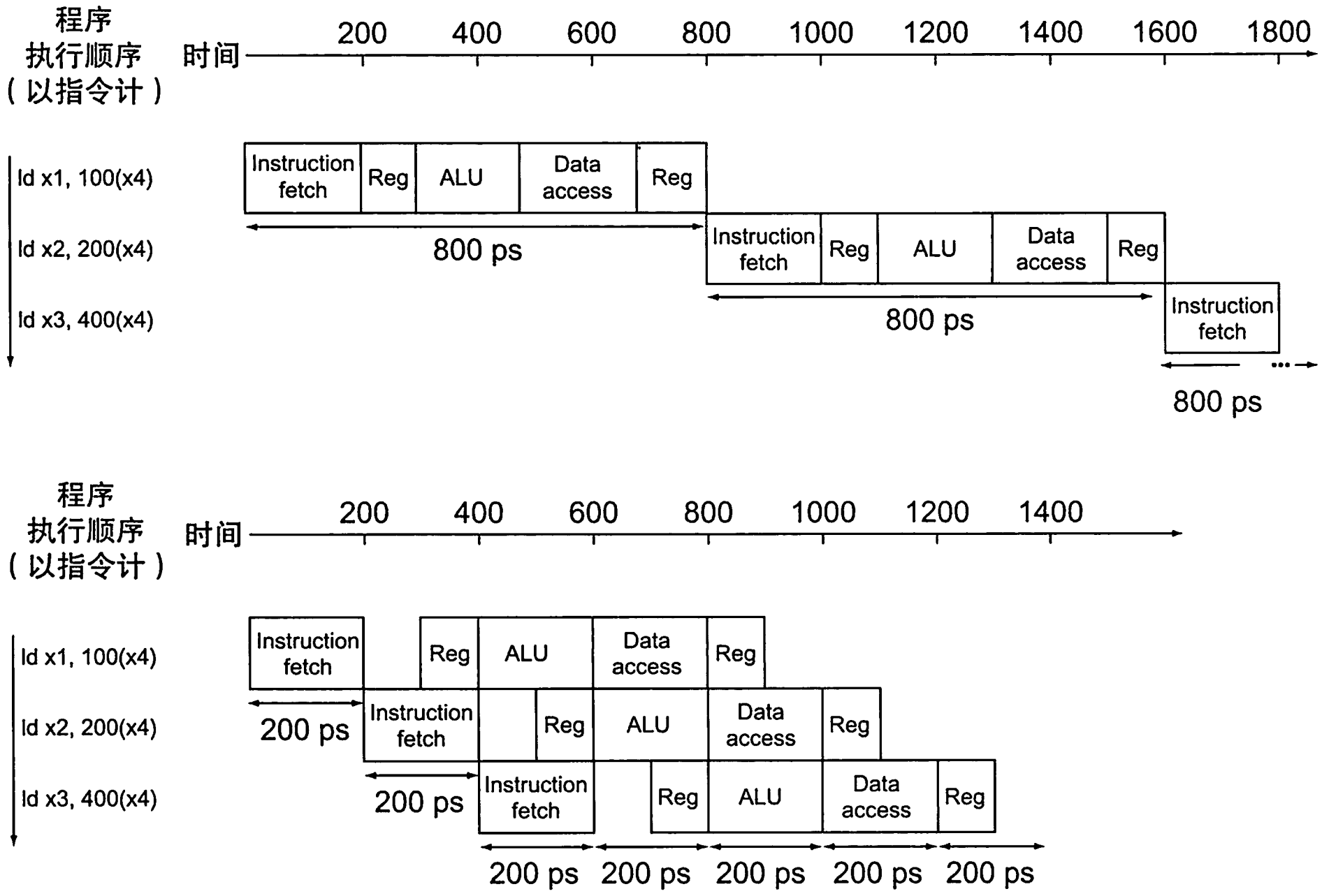


图 4-25 单周期、非流水线的指令执行（上）与流水线的指令执行（下）。两者采用相同的功能单元，各功能单元执行时间如图 4-24 所示。在这种情况下，我们看到指令的平均执行时间从 800ps 降低到 200ps，速度提高了 4 倍。比较此图与图 4-23。对于洗衣的例子，假设所有阶段所需时间相等。如果烘干是最慢的阶段，那么烘干阶段所需时间定为每个阶段的时间。计算机流水线阶段时间也受限于最慢的阶段，要么是 ALU 操作，要么是存储器访问。同时我们假设写寄存器堆操作发生在时钟周期的前半段，读寄存器堆操作发生在时钟周期的后半段。本章后面将一直遵循这个假设

所有的流水线阶段都需要一个时钟周期，所以流水线的时钟周期必须足够长以满足最慢的操作。就像单周期设计中，即使某些指令的执行可能只需要 500ps，但时钟周期要满足最坏情况 800ps。流水线的时钟周期也必须满足最坏情况 200ps，尽管有些阶段只需要 100ps。流水线仍然提供了 4 倍的性能改进：第一条和第四条指令之间的时间是 3 × 200ps=600ps。——■

我们可以将上面讨论的流水线带来的性能加速比归纳为一个公式。如果流水线各阶段操作平衡，那么流水线处理器上的指令执行时间（假设理想条件下）等于

$$\text{指令执行时间}_{\text{流水线}} = \frac{\text{指令执行时间}_{\text{非流水线}}}{\text{流水线级数}}$$

在理想的条件下和有大量指令的情况下，流水线带来的加速比约等于流水线级数；五级流水线带来的加速比接近 5。

该公式表明，一个五级流水线在 800ps 非流水线执行时间的情况下，能带来接近 5 倍的性能提高，即相当于时钟周期为 160ps。然而，在前面的例子中，各阶段不完全平衡。此外，流水线引入了一些开销，开销的来源稍后会更加清晰。因此，流水线处理器中每条指令的执行时间将超过最小值，所以加速比将小于流水线的级数。

此外，尽管在前面的分析中断言将有 4 倍的性能提升，但在本例的三条指令的总执行

时间中却并未反映出来：实际加速比是 2400ps/1400ps。当然，这是因为指令的数量不够多。如果增加指令的数量会发生什么？我们将前面图中的指令数增加到 1 000 003 条，也就是说在流水线中增加 1 000 000 条指令，每条指令使总执行时间增加 200ps。这样，总执行时间为 1 000 000 × 200ps + 1400ps=200 001 400ps。在非流水线方式中，增加 1 000 000 条指令，每条指令需要 800ps，因此总执行时间为 1 000 000 × 800ps + 2400ps=800 002 400ps。在这些条件下，在非流水线处理器与流水线处理器上，真实程序执行时间的比值接近于指令执行时间的比值：

$$\frac{800\ 002\ 400\text{ps}}{200\ 001\ 400\text{ps}} \simeq \frac{800\text{ps}}{200\text{ps}} \simeq 4.00$$

流水线技术通过提高指令吞吐率来提高性能，而不是减少单个指令的执行时间。由于真实程序会执行数十亿条指令，所以指令吞吐率是一个重要指标。

### 4.5.1 面向流水线的指令系统设计

尽管上面的例子只是对流水线的简单介绍，但我们也能够通过它了解面向流水线设计的 RISC-V 指令系统。

第一，所有 RISC-V 指令长度相同。这个限制简化了流水线第一阶段取指令和第二阶段指令译码。在像 x86 这样的指令系统中，指令长度从 1 个字节到 15 个字节不等，流水线设计更具挑战性。现代 x86 架构在实现时，将 x86 指令转换为类似 RISC-V 指令的简单操作，然后流水化这些简单操作，而不是流水化原始的 x86 指令（见 4.10 节）。

第二，RISC-V 只有几种指令格式，源寄存器和目标寄存器字段的位置相同。

第三，存储器操作数只出现在 RISC-V 的 load 或 store 指令中。这个限制意味着可以利用执行阶段来计算存储器地址，然后在下一阶段访问存储器。如果可以操作内存中的操作数，就像在 x86 中一样，那么第三阶段和第四阶段将扩展为地址计算阶段、存储器访问阶段和执行阶段。很快就会看到较长流水线的缺点。

### 4.5.2 流水线冒险

流水线中有一种情况，在下一个时钟周期中下一条指令无法执行。这种情况被称为冒险（hazard），我们将介绍三种冒险。

**结构冒险**

第一种冒险叫作**结构冒险**（structural hazard）。即硬件不支持多条指令在同一时钟周期执行。在洗衣例子中，如果用洗衣烘干一体机而不是分开的洗衣机和烘干机，或者如果你的室友正在做其他事情而不能收好衣服，都会发生结构冒险。这时，我们精心设计的流水线就会受到破坏。

**结构冒险**：因缺乏硬件支持而导致指令不能在预定的时钟周期内执行的情况。

如上所述，RISC-V 指令系统是面向流水线设计的，这使得设计人员在设计流水线时很容易避免结构冒险。然而，假设图 4-25 的流水线结构只有一个而不是两个存储器，那么如果有第四条指令，则会发生第一条指令从存储器取数据的同时第四条指令从同一存储器取指令，流水线会发生结构冒险。