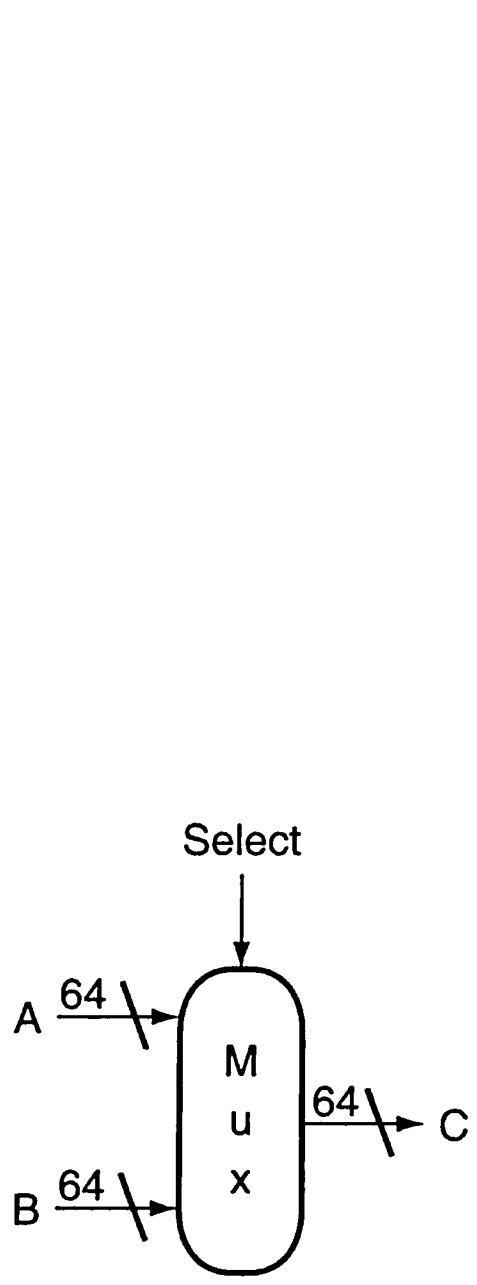
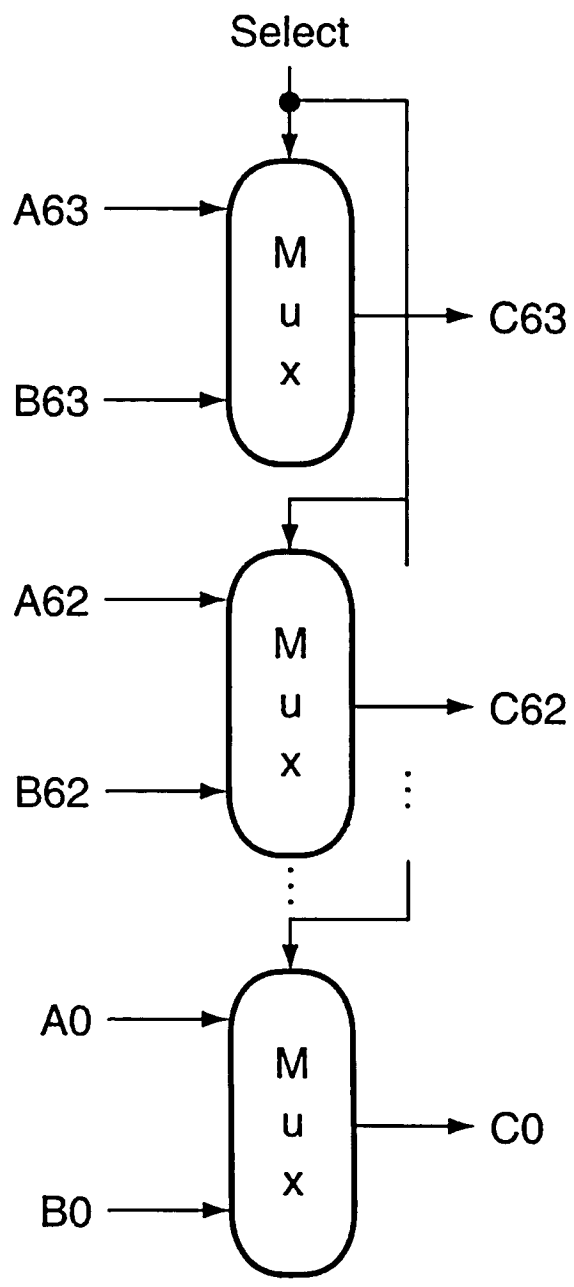
构造原理。在这种情况下，必须明确指出如何产生更宽的阵列，因为阵列中的各个单元不再是独立的了，正如 64 位多选器。



a）64位宽2选1多选器



b）64位宽多选器实际是64个1位多选器组成的陈列

图 A-3-6 将多选器排列 64 次以实现两个 64 位输入的选择。注意，对所有的 64 个 1 位多选器也只有一位的数据选择信号

自我检测 奇偶校验是一种由输入 1 的个数来决定输出的函数。对于偶校验函数，如果输入偶数个 1，则输出为 1。假设用一个 4 位输入的 ROM 来实现偶校验函数。*A*、*B*、*C*、*D* 中的哪一个能表示 ROM 的内容?

地址	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0

A.4　使用硬件描述语言

当前，大多数处理器和相关硬件系统的数字设计都是使用**硬件描述语言**（hardware description language）完成的。使用这个语言有两个目的：首先，它提供了硬件的抽象描述，可用来对设计进行模拟和调试；其次，通过使用逻辑合成和硬件编译工具，可以将其编译成硬件实现。

硬件描述语言：一种描述硬件的编程语言。用于模拟硬件设计，以及作为可生成实际硬件的合成工具的输入。

在本节中，我们将介绍硬件描述语言 Verilog，并展示其如何用于组合逻辑设计。在附录剩余部分扩展了 Verilog 的使用，并用于时序逻辑设计。在第 4 章的线上选读部分中，我们使用 Verilog 来描述处理器的实现。在第 5 章的线上选读部分中，使用 System Verilog 来描述 cache 控制器的实现。相对于 Verilog，System Verilog 增加了一些结构和其他有用的特征。

Verilog 是两种主要的硬件描述语言之一，另一个是 VHDL。相比于基于 Ada 语言的 VHDL，基于 C 语言的 Verilog 在工业界使用更广泛。熟悉 C 语言的读者会发现，我们在附录中使用的 Verilog 的基本内容很容易理解。对 C 语言语法有所了解且熟悉 VHDL 的读者，应该也会发现这些概念很简单。

Verilog：最常用的两种硬件描述语言之一。

VHDL：最常用的两种硬件描述语言之一。

Verilog 可以定义数字系统的行为和结构。**行为规范**（behavioral specification）描述了数字系统的功能操作。**结构规范**（structural specification）描述了数字系统的详细组织（通常使用分层描述）。结构规范可以从基本元件（如门和开关）层次描述硬件系统。因此，可以使用 Verilog 描述上一节中真值表和数据通路的具体内容。

行为规范：从功能上描述数字系统的操作。

结构规范：描述数字系统元件的层次化连接的组织。

随着**硬件综合工具**（hardware synthesis tool）的出现，大多数设计者使用 Verilog 或 VHDL 仅对数据通路进行结构化描述，再通过逻辑综合工具从行为描述生成控制信号。此外，大多数 CAD 系统都提供标准化部件扩展库，例如 ALU、多选器、寄存器堆、存储器、可编程逻辑块以及基本门电路。

硬件综合工具：能根据数字系统的行为描述生成门级设计的计算机辅助设计软件。

想要利用库和逻辑综合获得可接受的结果，需要根据最终的综合结果和期望的输出来编写规范。对于简单设计来说，需要明确的是期望用组合逻辑还是时序逻辑来实现。在本节和附录剩余部分的大多数示例中，编写的 Verilog 都考虑了最终的综合结果。

A.4.1　Verilog 中的数据类型和操作

Verilog 有两种基本数据类型：

1. wire 定义一个组合信号。

2. reg（寄存器）存储一个数值，数值可随时间变化。在实现时，reg 不一定和真实的寄存器相对应，但通常是对应的。

wire：在 Verilog 中，定义一个组合信号。

reg：在 Verilog 中，表示寄存器。

64 位宽的名为 X 的 reg 或 wire 声明为一个数组：`reg [63:0] X` 或 `wire [63:0] X`，将索引设置为 0 以指定寄存器的最低有效位。由于经常会访问寄存器或线的子域，可以用符号 [起始位：结束位] 来引用 reg 或 wire 的一段连续位，起始位和结束位都必须是常量值。

寄存器数组用于寄存器堆或存储器一类的结构。因此，声明

```
reg [63:0] registerfile[0:31]
```

定义了类似于 RISC-V 寄存器堆一个变量寄存器堆，其中寄存器 0 是第一个寄存器。和 C 语言一样，在访问数组时，使用符号 registerfile[regnum] 可以访问单个元素。

Verilog 中 reg 或 wire 的可能取值有：

- 0 或 1，表示逻辑假或真。
- X，表示未知，给定所有寄存器和未连接线的初始值。
- Z，表示三态门的高阻态，不在本附录讨论。

常量值可以指定为十进制、二进制、八进制或十六进制数。由于经常需要确切地说明一个常量字段的位数，可以在数值前加上十进制数来指定其位数。例如：

- 4'b0100 表示一个值为 4 的 4 位二进制常量，也可写作 4'd4。
- -8'h4 表示一个值为 −4（用补码表示）的 8 位常量。

将多个值放到 {} 中，并用逗号分隔，表示多个值的连接。符号 {x {bitfield}} 表示 bitfield 重复 x 次后的连接值。例如：

- {32{2'b01}} 表示创建一个 64 位值，模式是 0101…01。
- {A[31:16],B[15:0]} 表示创建一个 32 位值，由 A 的高 16 位和 B 的低 16 位连接而成。

Verilog 提供类似 C 语言的整套一元和二元操作符，包括：算术运算符（+，-，*，/），逻辑运算符（&，|，～），比较运算符（==，！=，>，<，<=，>=），移位运算符（<<，>>）以及 C 语言的条件运算符（？，使用格式为：condition?expr1:expr2，如果条件为真则返回 expr1，否则返回 expr2）。Verilog 增加了一组一元逻辑运算符（&，|，^），对操作数的所有位进行相应逻辑运算并产生一位结果。例如，&A 返回对 A 的所有位进行与运算得到的一位结果，^A 返回对 A 的所有位进行异或得到的一位结果。

自我检测 以下哪些项的值相同？

1. 8'bimoooo
2. 8'hF0
3. 8'd240
4. {{4{1'b1}},{4{1'b0}}}
5. {4'b1,4'b0)

A.4.2 Verilog 程序的结构

Verilog 程序由一组模块构成，模块可以表示任何内容，小到逻辑门集合，大到完整系统。模块类似于 C++ 中的类，只是没有类那么强大。模块指定其输入 / 输出端口，描述了模块传入和传出的连接。模块还可以声明一些附加的变量。模块的主体包括：

- initial 结构，初始化 reg 变量。
- 连续赋值，仅用于定义组合逻辑。
- always 结构，定义时序逻辑或组合逻辑。
- 其他模块实例，用于定义模块的实现。

A.4.3　Verilog 复杂组合逻辑的表示

关键字 assign 定义的连续赋值类似于组合逻辑函数：输出是连续赋给的值，输入值的变化会立即反映到输出值。wire 只能通过连续赋值进行赋值。使用连续赋值可以定义一个实现半加器的模块，如图 A-4-1 所示。

```
module half_adder (A,B,Sum,Carry);
   input A,B; //two 1-bit inputs
   output Sum, Carry; //two 1-bit outputs
   assign Sum = A ^ B; //sum is A xor B
   assign Carry = A & B; //Carry is A and B
endmodule
```

图 A-4-1　一个 Verilog 块，使用连续赋值定义了一个半加器

编写 Verilog 时，赋值语句的确是生成组合逻辑的一种方法。但是，对于更复杂的结构，使用赋值语句可能很笨拙或冗长。这时可以使用模块的 always 块来描述组合逻辑元件，但在使用时要十分小心。使用 always 块允许包含 Verilog 控制结构，例如 if-then-else、case 语句、for 语句和 repeat 语句。这些语句类似于 C 语句，仅有较小变化。

always 块定义了一个信号可选列表，这些信号对于语句块来说是敏感的（列表从 @ 开始）。如果列表中任何信号值发生改变，always 块都会更新；如果省略列表，则 always 块会不断重新计算。当 always 块定义组合逻辑时，**敏感列表**（sensitively list）应该包括所有输入信号。如果要在 always 块中执行多条 Verilog 语句，要把它们放到关键字 begin 和 end 之间，相当于 C 中的“{”和“}”。因此，always 块应该如下所示：

敏感列表：信号列表，定义 always 块何时应被重新计算。

```
always @(list of signals that cause reevaluation) begin
   Verilog statements including assignments and other
control statements end
```

reg 变量只能在 always 块中用过程赋值语句（区别于前面的连续赋值）进行赋值。但有两种不同类型的过程语句。赋值运算符 = 的执行类似于 C；计算等号右侧，并赋值给左侧。此外，还和 C 语言赋值语句的执行一致：即在下一条语句执行之前完成本条指令的执行。因此，赋值运算符 = 称作**阻塞赋值**（blocking assignment）。这种阻塞对于时序逻辑的生成很有用，稍后再做讲解。另一种赋值形式（**非阻塞**，nonblocking）用 <= 表示。在非阻塞赋值中，always 块所有赋值语句右侧同时进行运算，且赋值也都同时进行。作为使用 always 块实现的第一个组合逻辑示例，图 A-4-2 给出了 4 选 1 多选器的实现，使用了 case 结构使编写更加容易。case 构造类似于 C 语言的 switch 语句。图 A-4-3 给出了 RISC-V ALU 的定义，其中也使用了 case 语句。

阻塞赋值：Verilog 中的赋值语句，在下一条语句执行前完成。

非阻塞赋值：赋值语句，右侧计算持续执行，右侧计算完成后才对左侧赋值。

由于在 always 块中只有 reg 变量可以被赋值，当想用 always 块描述组合逻辑时，必须注意，确保 reg 不会被合成为寄存器。在下面的详细阐述中给出了各种陷阱。

详细阐述　连续赋值语句总能生成组合逻辑，但是其他 Verilog 结构即使在 always 块中，也可能在逻辑合成时产生意想不到的结果。最常见的问题是：使用现成的锁存器或寄存器来建立时序逻辑时，实现会比预期更慢且开销更大。为了确保这种方式能合成需要的组合

逻辑，需要执行以下操作：

1. 将所有组合逻辑置于连续赋值或 always 块中。
2. 确保用作输入的所有信号都出现在 always 块的敏感列表中。
3. 确保通过 always 块的每条通路都赋值给完全相同的位集合。

最后一条是最容易忽视的；仔细阅读图 A-5-15 中的示例，理解为什么要坚持这条准则。

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [63:0] In1, In2, In3, In4; //four 64-bit inputs
   input [1:0] Sel; //selector signal
   output reg [63:0] Out; //64-bit output
   always @(In1, In2, In3, In4, Sel)
   case (Sel) // a 4->1 multiplexor
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

图 A-4-2 Verilog 模块定义，使用 case 语句实现的带有 64 位输入的 4 选 1 多选器。case 语句类似于 C 语言中的 switch 语句，不过只执行 case 选中的 Verilog 代码（好像每个 case 状态后都有 break 一样）且不会执行下一条语句

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

图 A-4-3 RISC-V ALU 的 Verilog 行为定义。可以使用包含基本算术逻辑运算的模块库进行合成

自我检测 假设所有变量都被初始化为 0，在执行完以下 always 块中的 Verilog 代码后，A 和 B 的值是多少？

```
C = 1;
A <= C;
B = C;
```

A.5　构建基本算术逻辑单元

算术逻辑单元（Arithmetic Logical Unit，ALU）是计算机的主要组成部分，执行加 / 减等算术运算或者与 / 或等逻辑运算。本节用四种硬件单元（与门、或门、反相器、多选器）构建 ALU，并说明组合逻辑是如何工作的。下一节将介绍如何通过更好的设计来加速加法器。

算术逻辑单元（Arthritic Logic Unit 或者 Arithmetic Logic Unit，ALU），一种随机数生成器，所有计算机系统的标准部件。
Stan Kelly-Bootle, The Devil's DP Dictionary, 1981

由于 RISC-V 寄存器为 64 位宽，因此需要一个 64 位宽的 ALU。假设用 64 个 1 位 ALU 连接构造所需 ALU。因此，首先构建一个 1 位 ALU。

A.5.1　1 位 ALU

逻辑操作是最简单的，因为它们直接映射到图 A-2-1 中的硬件组件。

图 A-5-1 给出了 1 位与、或逻辑单元。右侧的多选器根据操作取值为 0 或 1，选择 *a* 与 *b* 或者 *a* 或 *b*。多选器的控制线用灰色表示，以区别于数据线。注意：多选器的控制和输出线进行了重命名，通过名称来反映 ALU 的功能。

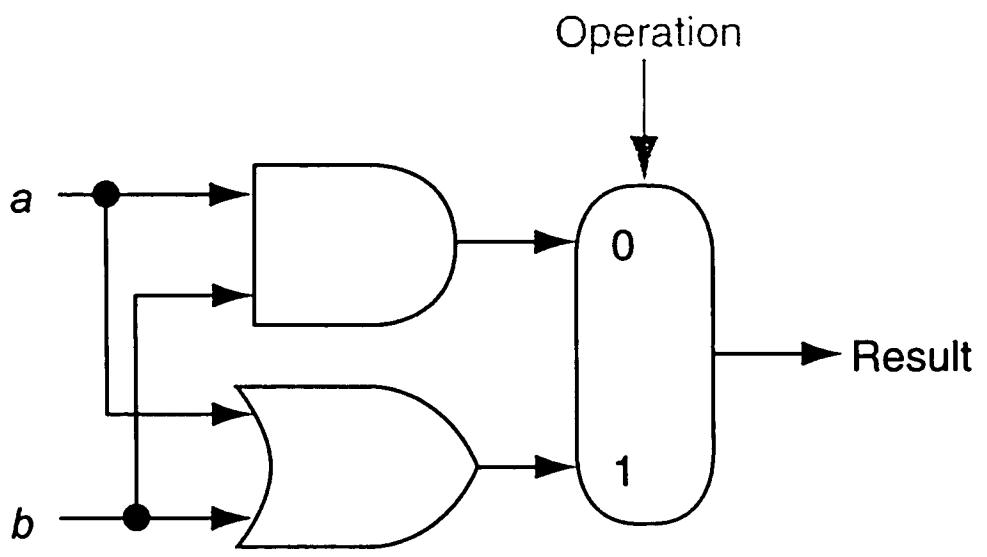


图 A-5-1　与和或的 1 位逻辑单元

需要包含的下一个功能是加法。加法器必须有两个操作数输入以及一位和输出，还必须有第二个输出用来传递进位，称为进位输出（CarryOut）。由于来自相邻加法器的进位输出需要作为输入，因此加法器需要第三个输入，称为进位输入（CarryIn）。图 A-5-2 展示了 1 位加法器的输入和输出。由于已知加法的作用，因此可以根据输入指定这个“黑盒”的输出，如图 A-5-3 所示。

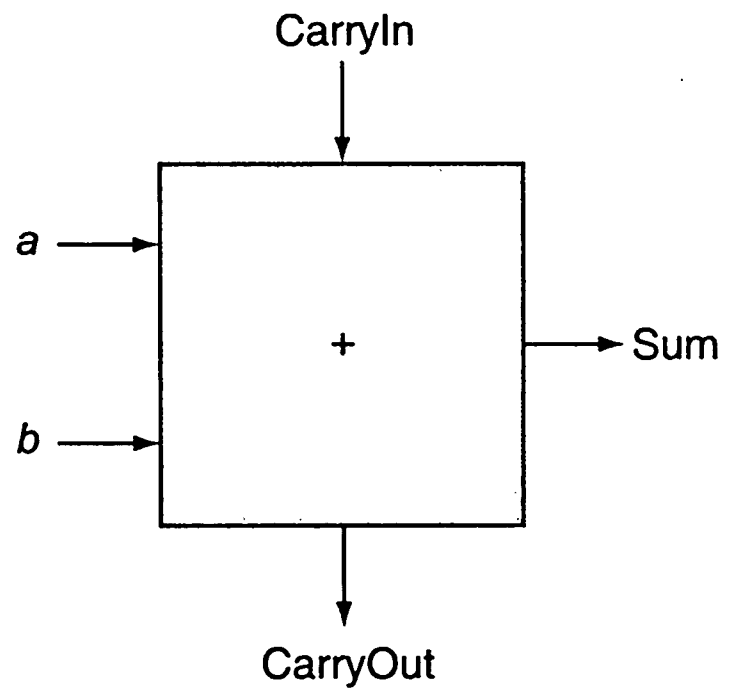


图 A-5-2　1 位加法器。该加法器称为全加器，也称为（3,2）加法器，因为它含有 3 个输入和 2 个输出。只含有 *a* 和 *b* 两个输入的加法器称为（2,2）加法器或半加器

输入			输出		
a	b	CarryIn	CarryOut	Sum	和
0	0	0	0	0	$0+0+0=00_2$
0	0	1	0	1	$0+0+1=01_2$
0	1	0	0	1	$0+1+0=01_2$
0	1	1	1	0	$0+1+1=10_2$
1	0	0	0	1	$1+0+0=01_2$
1	0	1	1	0	$1+0+1=10_2$
1	1	0	1	0	$1+1+0=10_2$
1	1	1	1	1	$1+1+1=11_2$

图 A-5-3　1 位加法器的输入和输出定义

可以用逻辑方程来表示进位输出与和的输出函数，这些方程式又可以用逻辑门实现。以进位输出为例，图 A-5-4 给出了进位输出为 1 时的输入值。

输入		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

图 A-5-4　当进位输出为 1 时输入的值

可以把真值表转化为逻辑方程式：

$$\text{CarrryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

如果 $a \cdot b \cdot \text{CarryIn}$ 为真，那么其余三项也必须为真，可以根据真值表第四行将最后一项省略掉。因此，方程式可以简化为：

$$\text{CarrryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

图 A-5-5 给出了加法器黑盒内进位输出的硬件，由三个与门和一个或门组成。三个与门完全对应上面进位输出公式的三个带括号的项，或门对这三项求和。

当仅有一个输入为 1 或者三个输入都为 1 时，和位被置位。和由复数布尔方程（$\bar{a}$ 表示非 a）产生：

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

加法器黑盒中和位的逻辑绘制留作练习。

图 A-5-6 给出了 1 位 ALU，由加法器与先前的组件组合而成。有时设计人员还希望 ALU 执行一些更简单的操作，例如生成 0。增加操作最简单的方法就是扩展由操作线控制的多选器，对于这个例子，将 0 直接连接到扩展后的多选器的新输入端。

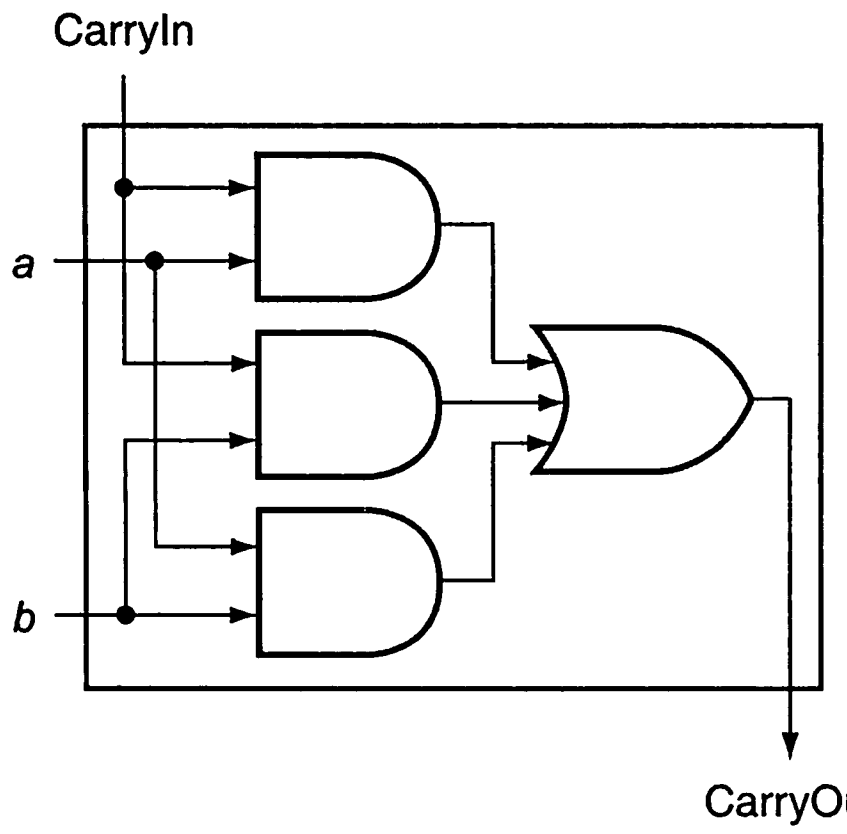


图 A-5-5 进位输出信号的加法器硬件。加法器硬件的剩余部分是下式中和（Sum）的输出逻辑

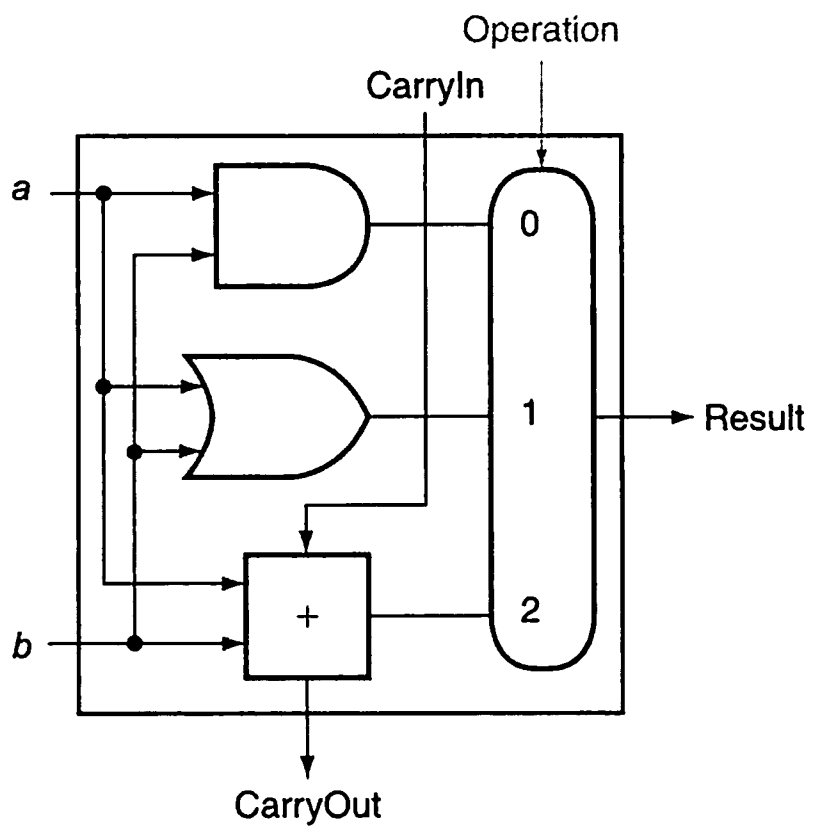


图 A-5-6 实现与、或、加的 1 位 ALU（见图 A-5-5）

A.5.2 64 位 ALU

现在已经完成了 1 位 ALU，通过连接相邻的“黑盒”来创建完整的 64 位 ALU。使用 xi 表示 x 的第 i 位，图 A-5-7 给出了 64 位 ALU。正如一颗石子就能使平静的湖面激起阵阵涟漪，最低有效位（Result0）的一个进位就能导致进位扩展到加法器，致使最高有效位（Result63）产生进位。因此，直接连接 1 位进位加法器构成的加法器称为行波进位加法器。从 A.6 节开始，我们将看到一种更快连接 1 位加法器的方法。

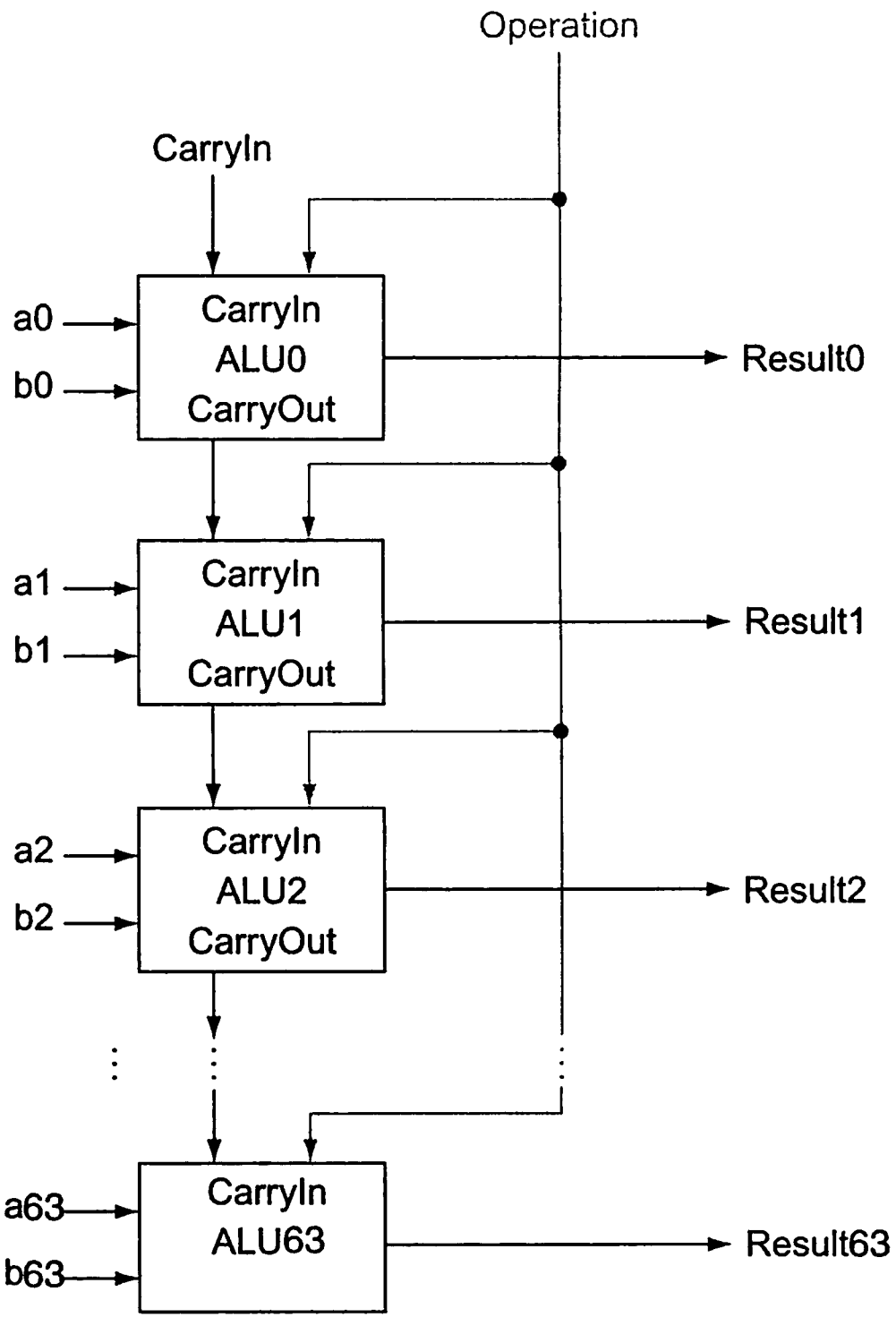


图 A-5-7 64 个 1 位 ALU 构成的 64 位 ALU。最低有效位的进位输出连接到最高有效位的进位输入。这种组织形式叫作行波进位

减法相当于加一个操作数的负数，这正是加法器执行减法的方式。回想一下，二进制补码取负的快捷方式是每位取反（有时称为补码）再加 1。要对每一位取反，只需添加一个 2:1 多选器，在 b 和 $\overline{b}$ 之间选择，如图 A-5-8 所示。

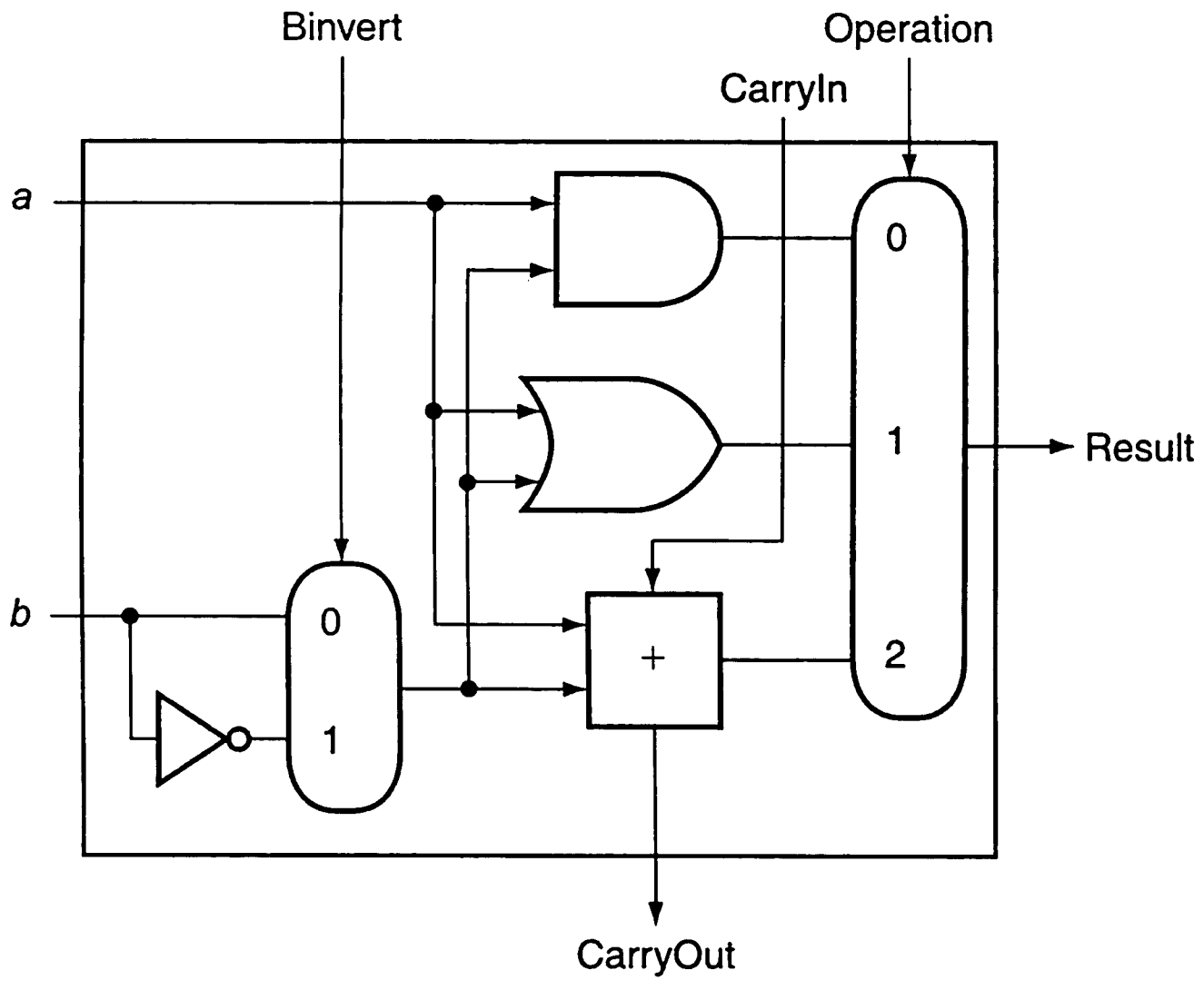


图 A-5-8　一个执行与、或、$a+b$ 或者 $a+\overline{b}$ 加法的 1 位 ALU。通过选择 $\overline{b}$（Binvert=1）以及将最低有效位的进位输入设置为 1，可以得到 $a-b$ 的补码而不是 $a+b$

假设连接 64 个 1 位 ALU，如图 A-5-7 所示。添加的多选器选出 b 或其取反值，具体取决于 Binvert，但这只是求二进制补码的第一步。注意：最低有效位仍然具有进位输入信号，即使对于加法无关紧要。如果将此进位输入信号设置为 1 而不是 0，会发生什么？此时加法器会计算 $a + b + 1$。通过选择 b 的取反，就能得到想要的结果：

$$a+\overline{b}+1=a+(\overline{b}+1)=a+(-b)=a-b$$

二进制补码加法器硬件设计的简单性有助于解释为什么二进制补码表示已成为整数计算机算术的通用标准。

还需要添加或非（NOR）功能。和减法一样，或非也可以通过重用 ALU 中已有的大部分硬件来实现，而不是增加单独的或非门。或非可以转化为：

$$\overline{(a+b)}=\overline{a}\cdot\overline{b}$$

也就是说，（a 或 b）的非等价于非 a 与非 b。这也被称为德摩根定律，在练习中对此进行了更深入的探讨。

由于有与和非 b，只需要在 ALU 中增加非 a 既可。图 A-5-9 给出了变化后的结构。

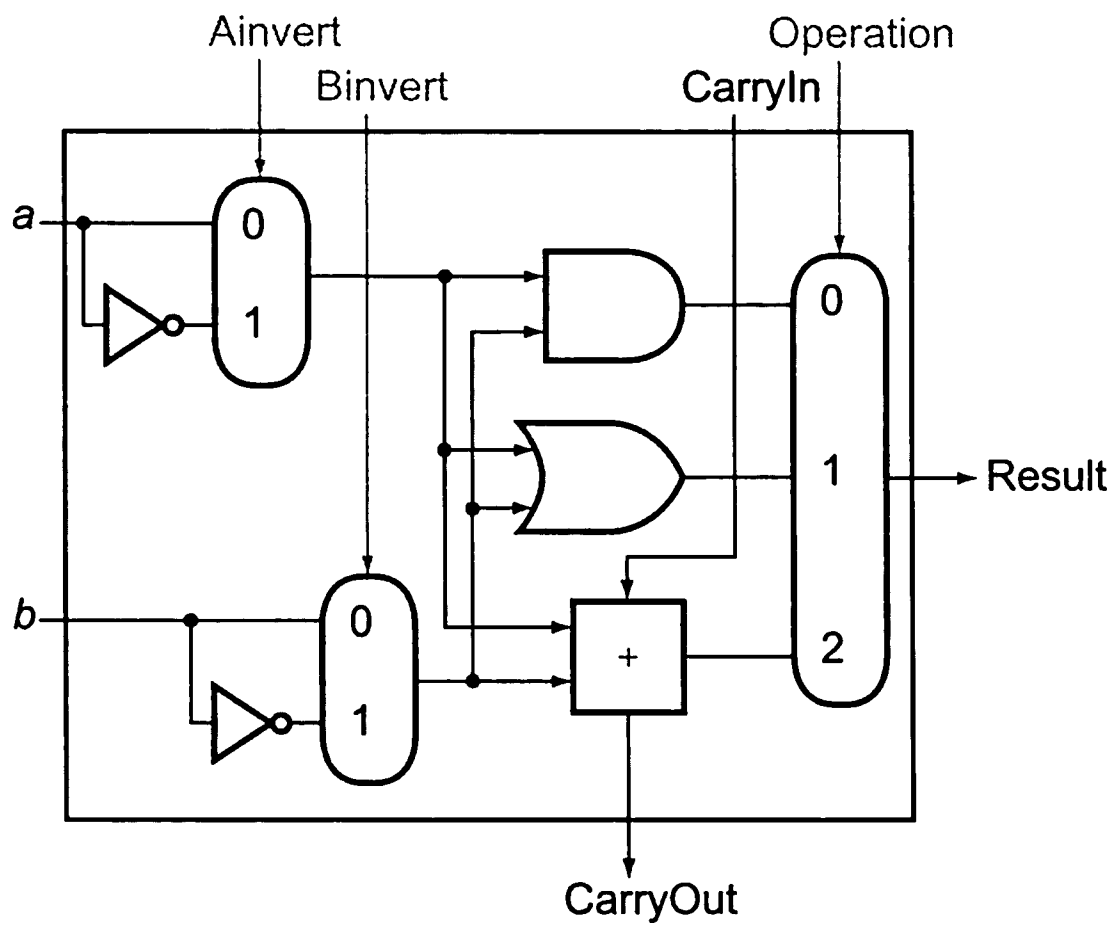


图 A-5-9　一个实现与、或、$a+b$ 或者 $\overline{a}+\overline{b}$ 加法的 1 位 ALU。通过选择 $\overline{a}$（Ainvert=1）和 $\overline{b}$（Binvert=1），可以得到 a 异或 b，而不是 a 与 b

A.5.3　修改 64 位 ALU 以适应 RISC-V

几乎所有计算机的 ALU 都包括加、减、与、或四种操作，且大多数 RISC-V 指令操作都可以用这个 ALU 实现。但是这个 ALU 的设计还不完整。

还需支持的一条指令是小于置位指令（slt）。如果 rs1 < rs2，则操作产生 1，否则返回 0。因此，slt 会将除最低有效位之外的所有位都置为 0，并根据比较结果设置最低有效位。为了让 ALU 执行 slt 指令，首先需要扩展图 A-5-9 中的三输入多选器，为 slt 结果添加一个输入，称这个新输入为 Less，仅用于 slt。

图 A-5-10 的上图给出了带有扩展多选器的新 1 位 ALU。根据以上的 slt 描述，必须将 0 连接到 ALU 高 63 位的 Less 输入，因为这些位总是置 0。还需要考虑的是如何比较和设置 slt 指令的最低有效位。

如果 a−b 会发生什么？如果结果为负，那么 $a < b$，因为

$$(a-b)<0 \Rightarrow ((a-b)+b)<(0+b) \Rightarrow a<b$$

如果 $a < b$，小于置位操作的最低有效位置 1；也就是说，如果 $a-b$ 为负，则最低有效位为 1；如果为正，则为 0。需要的结果与符号位值完全对应：1 表示负，0 表示正。根据这个结果，只需将加法器输出的符号位连接到最低有效位就可以实现小于置位比较。（可惜，这个结果只在减法无溢出时有效；在练习中将讨论其完整实现。）

不幸的是，图 A-5-10 的上图中，slt 操作的来自 ALU 最高有效位的输出结果并不是加法器的输出，slt 操作的 ALU 输出显然是 Less 的输入值。

因此，对于有额外输出位的最高有效位，需要一个新的 1 位 ALU：加法器的输出。图 A-5-10 的下图给出了这个设计，新的加法器输出线称为 Set。最高有效位只需要一个特殊的 ALU，仅增加溢出检测逻辑，因为它与该位是关联的。图 A-5-11 给出了 64 位 ALU。

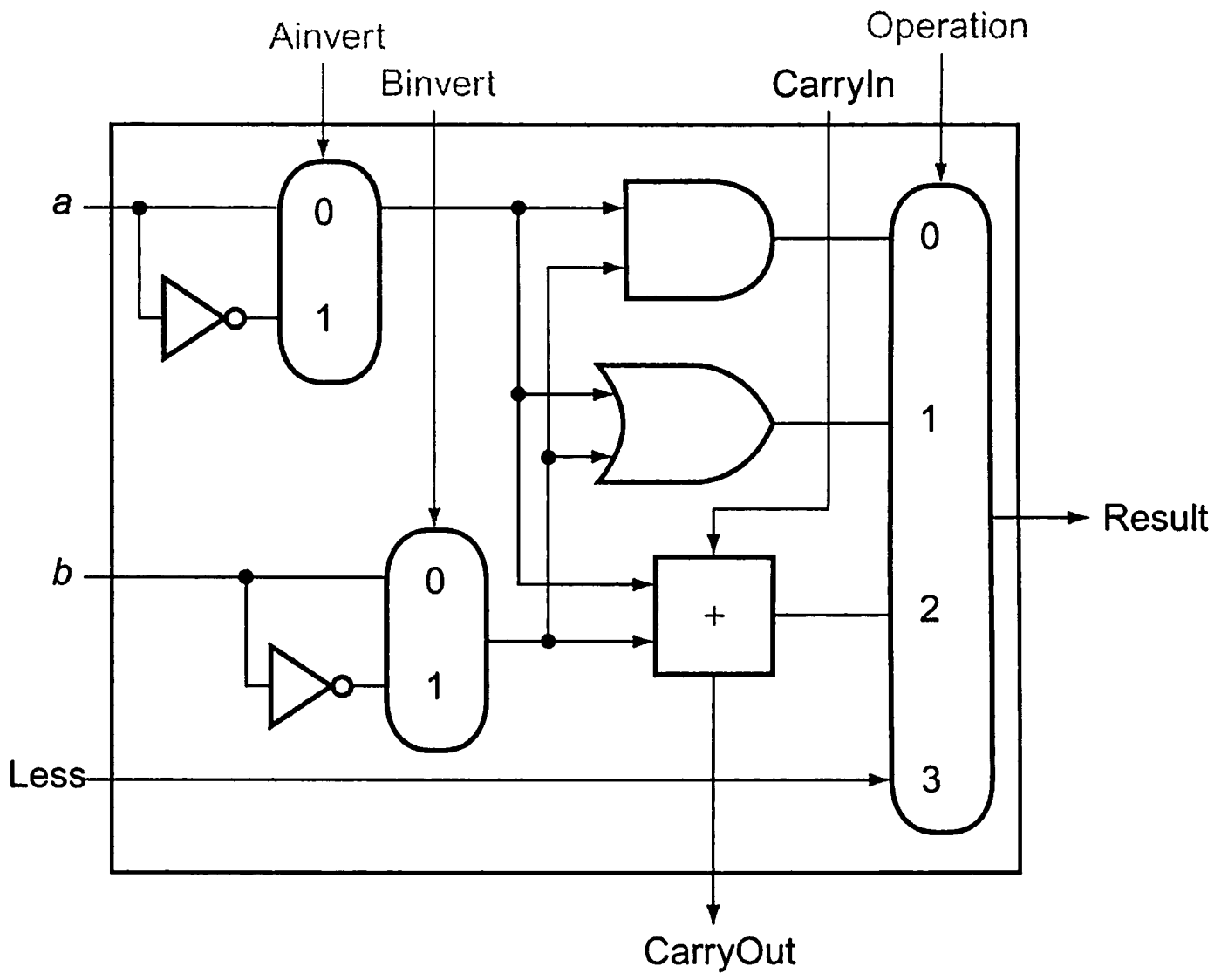


图 A-5-10　（上图）实现与、或、$a+b$ 或者 $a+\overline{b}$ 加法的 1 位 ALU；（下图）带有最高有效位的 1 位 ALU。上图包含一个实现小于比较置位操作的直接输入（见图 A-5-11）；下图有一个实现小于比较的来自加法器的直接输出，称为 set（见附录末练习 A.24，了解如何用更少的输入计算溢出）

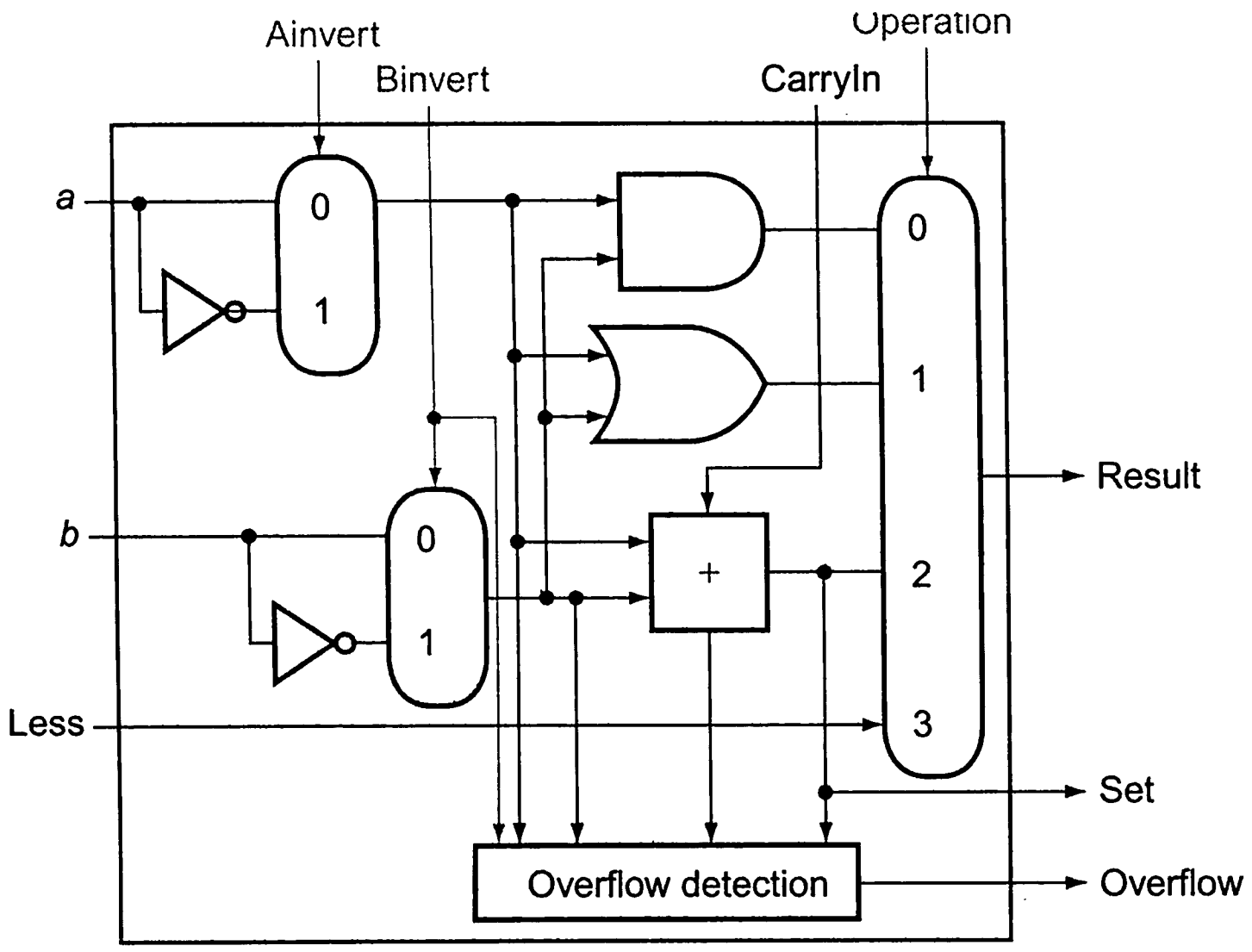


图 A-5-10 （续）

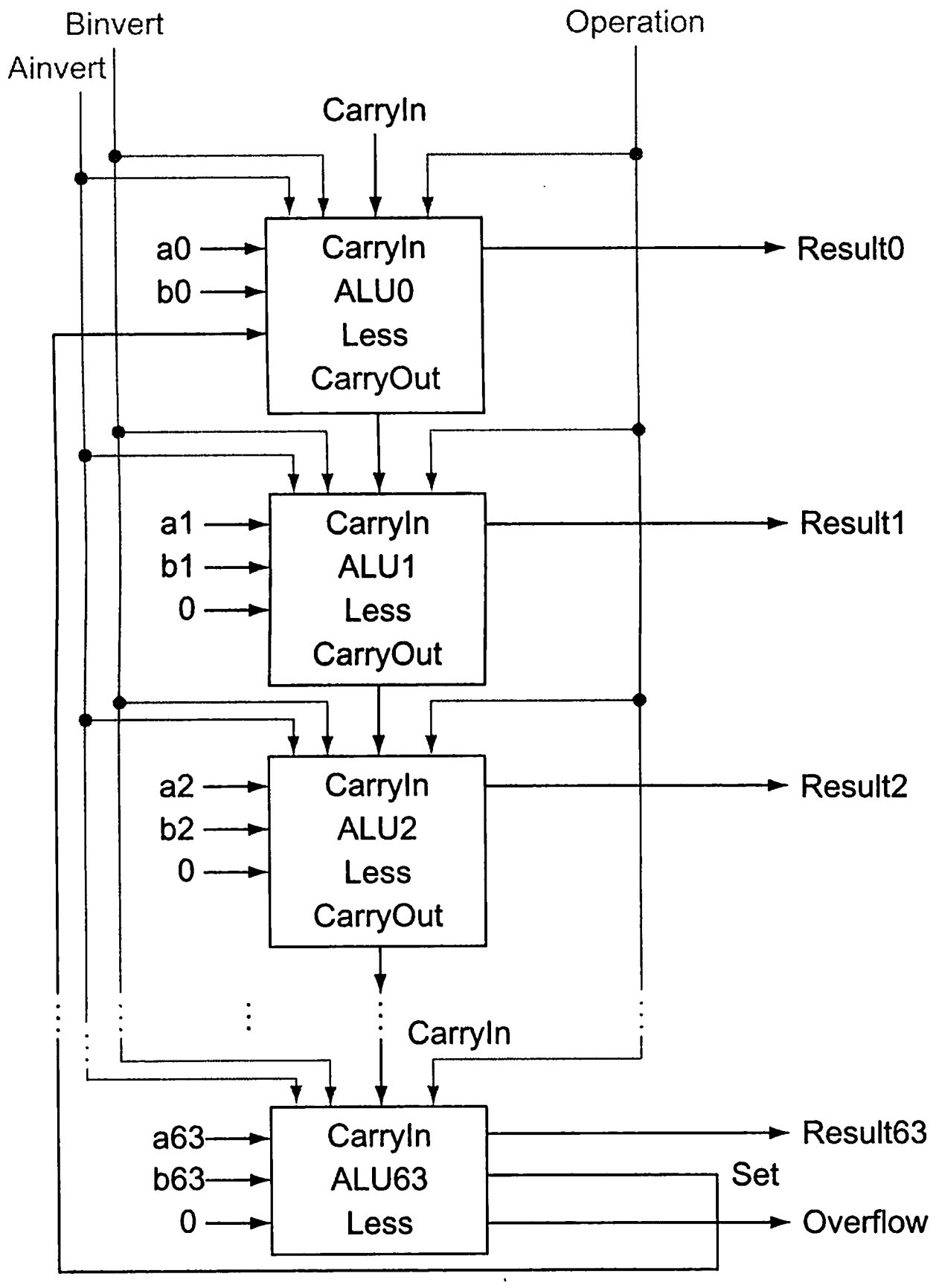


图 A-5-11 63 个图 A-5-10 上部所示的 1 位 ALU 和 1 个下部所示的 1 位 ALU 构成的 64 位 ALU。除最低有效位，Less 输入均连接到 0，最低有效位连接到最高有效位的 Set 输出。如果 ALU 执行 $a-b$ 且图 A-5-10 中的多选器选择输入 3，则如果 $a < b$，Resutl= 0 ... 001，否则 Resutl= 0 ... 000

注意，每次使用 ALU 做减法时，要将 CarryIn 和 Binvert 都设置为 1。对于加法或逻辑运算，两条控制线都置 0。因此，通过将 CarryIn 和 Binvert 合成一条控制线 Bnegate，可以简化 ALU 的控制。

为了进一步定制 ALU 以适应 RISC-V 指令系统，还必须支持条件分支指令，如相等则分支（beq），即如果两个寄存器相等则分支。使用 ALU 进行相等测试的最简单方法是计算 $a-b$，然后查看结果是否为 0，因为

$$(a-b=0)\Rightarrow a=b$$

因此，如果添加硬件以测试结果是否为 0，就可以测试是否相等。最简单的方法是将所有输出组合取或，然后通过反相器发送该信号：

$$\text{Zero}=\overline{(\text{Result63}+\text{Result62}+\cdots+\text{Result2}+\text{Result1}+\text{Result0})}$$

图 A-5-12 给出了修改后的 64 位 ALU。可以将 1 位 Ainvert 线、1 位 Bnegate 线和 2 位操作线组合为 ALU 的 4 位控制线，控制执行加、减、与、或、异或、小于置位。图 A-5-13 给出了 ALU 控制线和相应的 ALU 操作。

最后，既然已经看到了 64 位 ALU 的内部结构，之后将使用通用符号表示完整的 ALU，如图 A-5-14 所示。

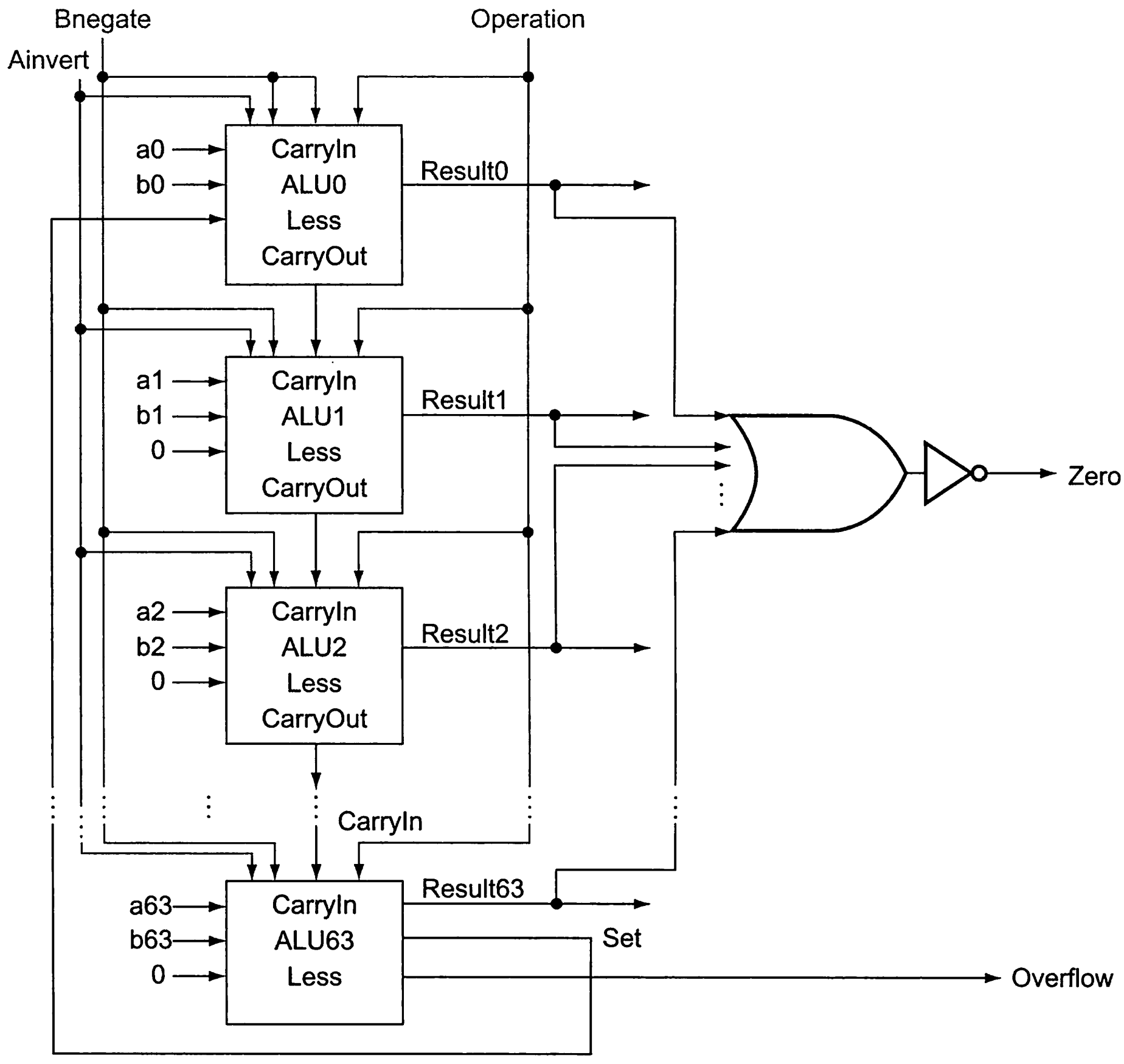


图 A-5-12　最终的 ALU。在图 A-5-11 的基础上增加 0 检测器

ALU控制线	功能
0000	与
0001	或
0010	加
0110	减
0111	小于比较置位
1100	或非

图 A-5-13 三条 ALU 控制线（Ainvert 线、Bnegate 线、Operation 线）的值，以及相应的 ALU 操作

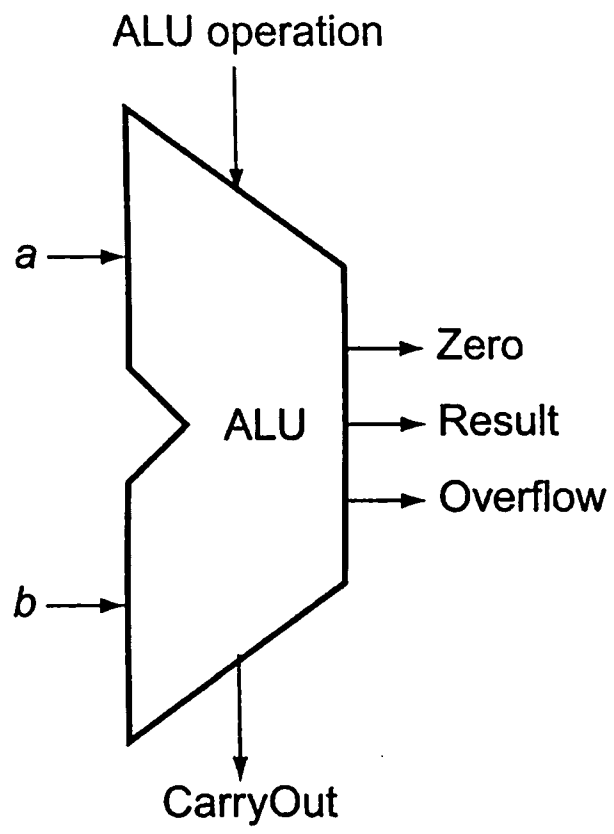


图 A-5-14 通常用来表示图 A-5-12 所示的 ALU 符号。这个符号也用来表示加法器，因此通常使用 ALU 或 Adder 标记

A.5.4 用 Verilog 定义 RISC-V ALU

图 A-5-15 给出了如何使用 Verilog 定义组合 RISC-V ALU。这样的规范可能会使用提供加法器的标准零件库进行编译，可以实例化。为了完整起见，图 A-5-16 给出了 RISC-V ALU 的控制（在第 4 章中使用过），其中构建了 Verilog 版的 RISC-V 数据通路。

```
module RISCVALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
   end
endmodule
```

图 A-5-15 RISC-V ALU 的 Verilog 行为定义

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] reg ALUCtl;
   always case (FuncCode)
   32: ALUOp<=2; // add
   34: ALUOp<=6; // subtract
   36: ALUOP<=0; // and
   37: ALUOp<=1; // or
   39: ALUOp<=12; // nor
   42: ALUOp<=7; // slt
   default: ALUOp<=15; // should not happen
   endcase
endmodule
```

图 A-5-16　RISC-V ALU 控制：一个简单的组合控制逻辑

下一个问题是，这个 ALU 进行两个 64 位操作数的加法运算能有多快？可以确定 a 和 b 的输入，但 CarryIn 输入取决于相邻 1 位加法器的操作。如果跟踪整个依赖关系链，将最高有效位连接到最低有效位，因此和的最高有效位必须等待 64 个 1 位加法器顺序求值（后才能得到）。这种顺序链式反应太慢，无法用于时序要求严格的硬件。下一节将探讨如何加速加法。这个主题对于理解附录的其余部分不是至关重要的，可以跳过。

自我检测　假设想要增加一个 NOT（a AND b）的操作，称为与非（NAND）。如何修改 ALU 以支持该操作？

1. 不做修改。可以使用当前 ALU 快速计算 NAND，因为 $\overline{(a \cdot b)} = \bar{a} + \bar{b}$，且已经有 $\bar{a}$、$\bar{b}$ 和 +。
2. 扩展多选器以增加一个新的输入，然后增加新的逻辑来计算 NAND。

A.6　快速加法：超前进位

加速加法的关键是提高高阶进位的速度。有多种方案可以预测进位，因此最坏的情况是加法器位数的 $\log_2$ 函数。这些预测信号更快，因为它们按顺序通过更少的门，但需要更多的门来预测正确的进位。

理解快速进位的关键是要记住：与软件不同，无论输入何时改变，硬件都会并行执行。

A.6.1　使用“无限”硬件的快速进位

正如前面提到的，任何等式都可以表示成两级逻辑。由于仅有的外部输入是两个操作数，以及加法器最低有效位的进位输入，理论上可以仅用两级逻辑计算加法器所有剩余位的进位输入值。

例如，加法器的第 2 位的进位输入恰好是第 1 位的进位输出，因此公式为

$$\text{CarryIn2} = (\text{b1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{CarryIn1}) + (\text{a1} \cdot \text{b1})$$

类似地，CarryIn1 定义为

$$\text{CarryIn1} = (\text{b0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{CarryIn0}) + (\text{a0} \cdot \text{b0})$$

用 ci 代替 CarryIni，将以上公式重写为

$$\text{c2} = (\text{b1} \cdot \text{c1}) + (\text{a1} \cdot \text{c1}) + (\text{a1} \cdot \text{b1})$$

$$c1 = (b0 \cdot c0) + (a0 \cdot c0) + (a0 \cdot b0)$$

把 c1 代入 c2，可得

$$c2 = (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) + (a1 \cdot b0 \cdot c0) + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (a1 \cdot b1)$$

可以想象一下，加法器得到更高的位时方程会如何扩大——它会随着位数的增高迅速增长。这种复杂性反映在快速进位的硬件开销上，使得这种简单机制对于扩展宽加法器来说过于昂贵。

A.6.2 使用第一级抽象的快速进位：传播和生成

大多数快速进位机制限制了方程式的复杂性以简化硬件，同时获得比行波进位更大的速度提升。其中一种机制就是超前进位加法器（carry-lookahead adder）。在第 1 章中，介绍了计算机系统通过使用抽象层次来应对复杂性。超前进位加法器依赖于其实现中的抽象层次。

将原方程式作为第一步因子：

$$c_{i+1} = (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) = (a_i \cdot b_i) + (a_i \cdot b_i) \cdot c_i$$

如果用这个公示重写 c2 方程式，会看到一些重复的部分：

$$c2 = (a1 \cdot b1) + (a1 \cdot b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0)$$

注意上式中重复出现了（$a_i \cdot b_i$）和（$a_i + b_i$）。这两个重要的因子通常称为（进位）生成（generate（g_i））和（进位）传播（propagate（p_i））：

$$g_i = a_i \cdot b_i$$

$$p_i = a_i + b_i$$

用它们来定义 c_{i+1}，可以得到：

$$c_{i+1} = g_i + p_i \cdot c_i$$

为了理解信号是从哪里来的，假设 g_i 位 1。则

$$c_{i+1} = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$$

也就是说，加法器生成一个进位输出（c_{i+1}）独立于进位输入的值（c_i）。现假设 g_i 为 0 且 p_i 为 1。则

$$c_{i+1} = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$$

也就是说，加法器把进位输入传播给进位输出。将两者合在一起，如果 g_i 为 1 或者 p_i 和 $CarryIn_i$ 均为 1，则 $CarryIn_{i+1}$ 为 1。

作一个类比，想象一排多米诺骨牌。只要两两之间没有间隙，推倒远处的一块多米诺牌可以让最后一块多米诺牌也倒下。类似地，一个远处的（进位）生成也可以使当前的进位输出为真，只要它们之间的所有（进位）传播都为真。

根据传播和生成的定义，将其作为第一级抽象，可以更简洁地表示进位输入信号。4 位进位如下：

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$