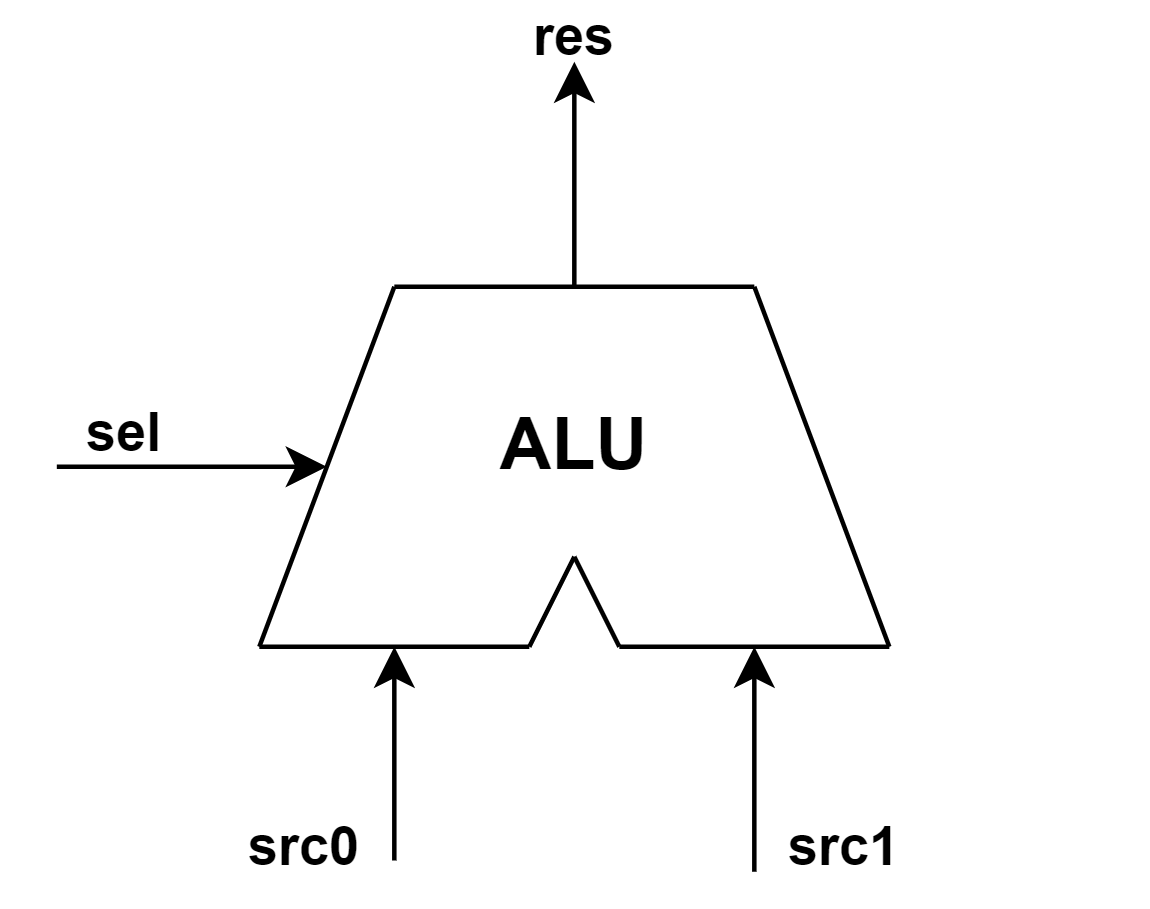
**实验15**

**算术逻辑单元ALU**

# **实验介绍**

ALU 是 CPU 的一种重要组件，它负责实现 CPU 的算术运算和逻辑运算功能。简而言之，CPU 中几乎所有的运算过程，比加减法、比较、位移等，都是由 ALU 来完成的。

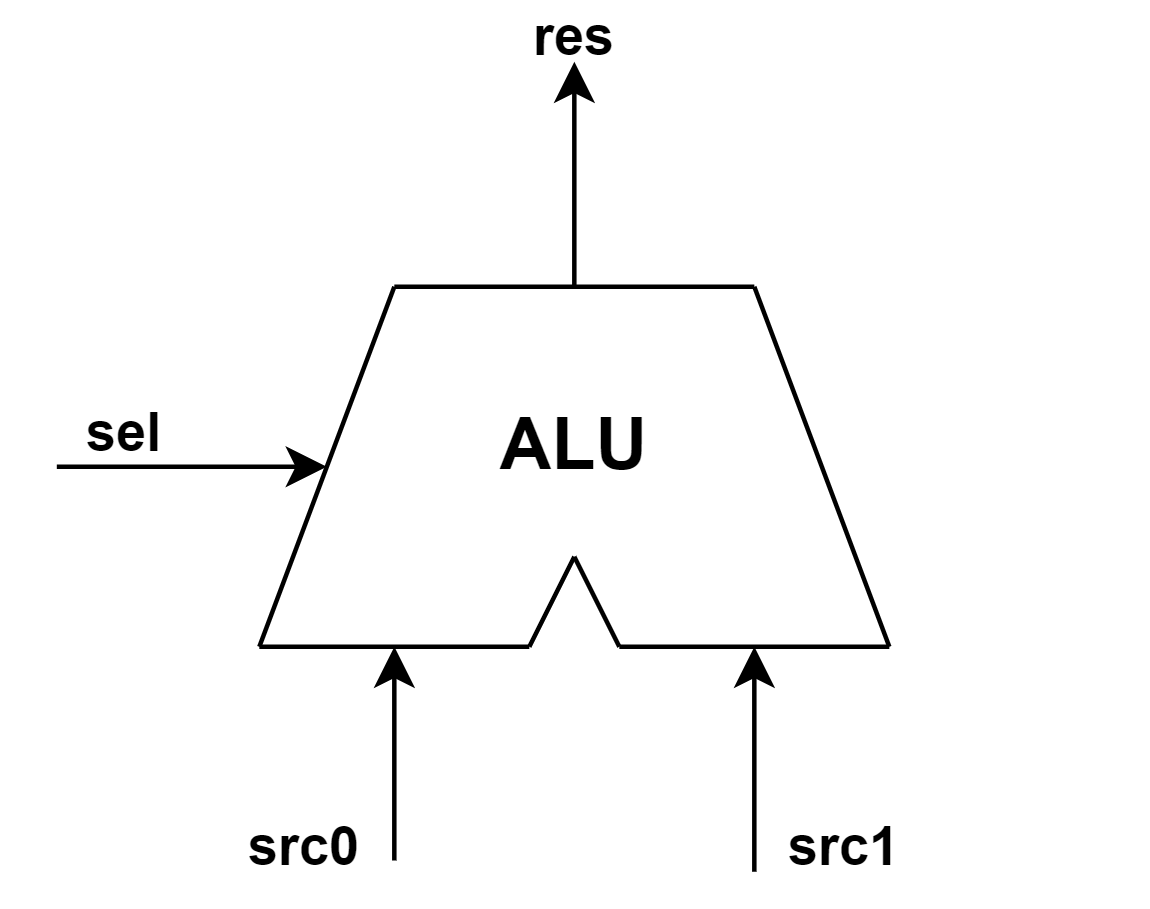
[](https://soc.ustc.edu.cn/Digital/lab6/figs/ALU.png)

那么，如何开发 ALU 呢？我们当然可以选择 Verilog 提供的运算符来实现所有的 ALU 运算功能。但是，这样的实现可能会导致 ALU 的性能较差或资源使用量较大，从而对 CPU 的最终性能产生影响，因此我们将尝试自己实现 ALU 中的一些运算。

# **实验内容**

## 1 原理介绍

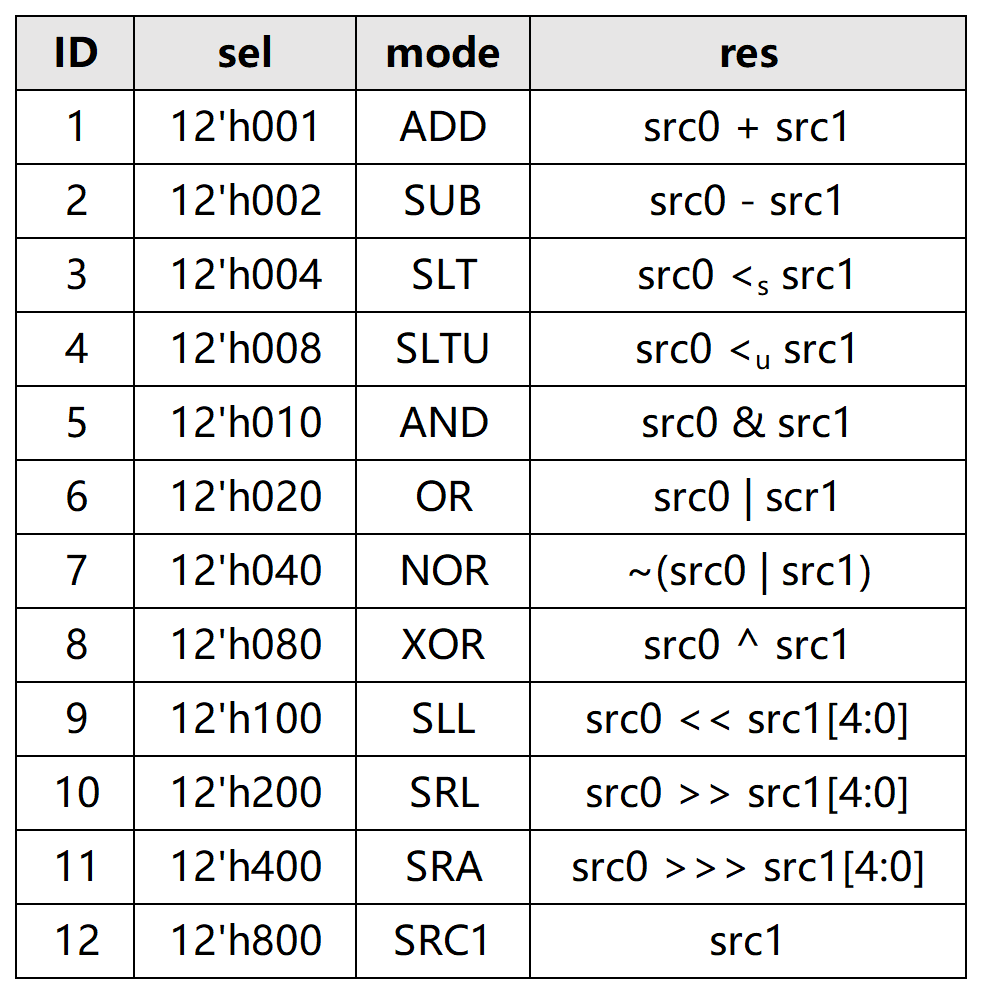
一般而言，为了保证 CPU 的运行效率，我们希望 ALU 能够在一个周期内就能得到计算结果，故 ALU 常被设计为一个组合模块。乘除法等运算由于比较耗时，一般被单独设计为一个模块，放在 CPU 的其他位置而非 ALU 中。在我们的设计中，ALU 接受两个操作数 src0 和 src1，以及一个独热码选择信号 sel，用以选择运算类型。ALU 的输出为运算结果 res。其输入输出模型如下图：

[](https://soc.ustc.edu.cn/Digital/lab6/figs/ALU/ALU.png)

在龙芯架构 32 位精简版指令集中，ALU 需要支持以下运算：

* 简单算术运算：加法（ADD）、减法（SUB）、小于比较（SLT）、无符号小于比较（SLTU）
* 逻辑运算：按位或非（NOR），按位与（AND），按位或（OR），按位异或（XOR），左移（SLL）、右移（SRL）、算术右移（SRA）

各种运算对应的选择信号如下表所示：

[](https://soc.ustc.edu.cn/Digital/lab6/figs/ALU/ALU_table.png)

一些同学可能会想参考 C 语言等高级语言的编程习惯，根据输入信号 sel 选择调用对应的运算模块，将输入 src0、src1 传递给该模块并获取结果。但是这一想法是不符合硬件设计思想的。这是因为 Verilog 中的模块例化与其他语言的函数调用有较大区别，它通常作为一个单独的代码段，不能放在 if-else 等其它代码段中。从硬件的角度看，设计出结构在上板前必须是确定的。我们不可能根据电路的运行结果动态地生成或删除电路结构，只能引导信号经过不同的路径。

因此，合理的思路是，同时将两个源操作数送到所有运算单元，再通过输入信号 sel 来借助多选器选择哪个运算单元的输出作为最终的结果。即，我们选择的不是使用哪个模块，而是选择使用哪个模块的结果。即使没有选择某个运算，它对应的模块也依然在工作，只是我们没有选择它的结果而已。



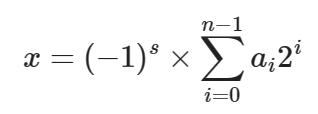
上面是我们给出的 ALU 代码框架以及结果选择的一种实现方式，你也可以选择使用 case 来实现结果的选择。

## 2 减法运算

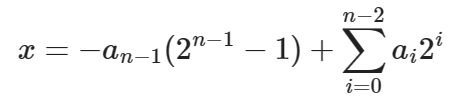
### ****2.1 补码运算****

对于 n位二进制数，我们一般按照  的形式进行表示，其中 ai∈{0,1}。不难看出，n 位二进制数的表示范围是 0∼2^n−1，即所有小于 2^n 的非负正整数。那么，我们应当如何表示负数呢？目前有三种主要的表示方法：

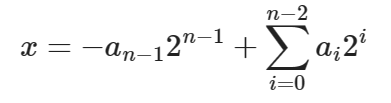
原码表示（符号-数值码）：在数的最前面增加一个符号位 s，并约定若 s=0 则该数为正数，若 s=1 则该数为负数。



反码表示：负数由其相反数的原码取反得到。



补码表示：负数由其相反数取反加一得到

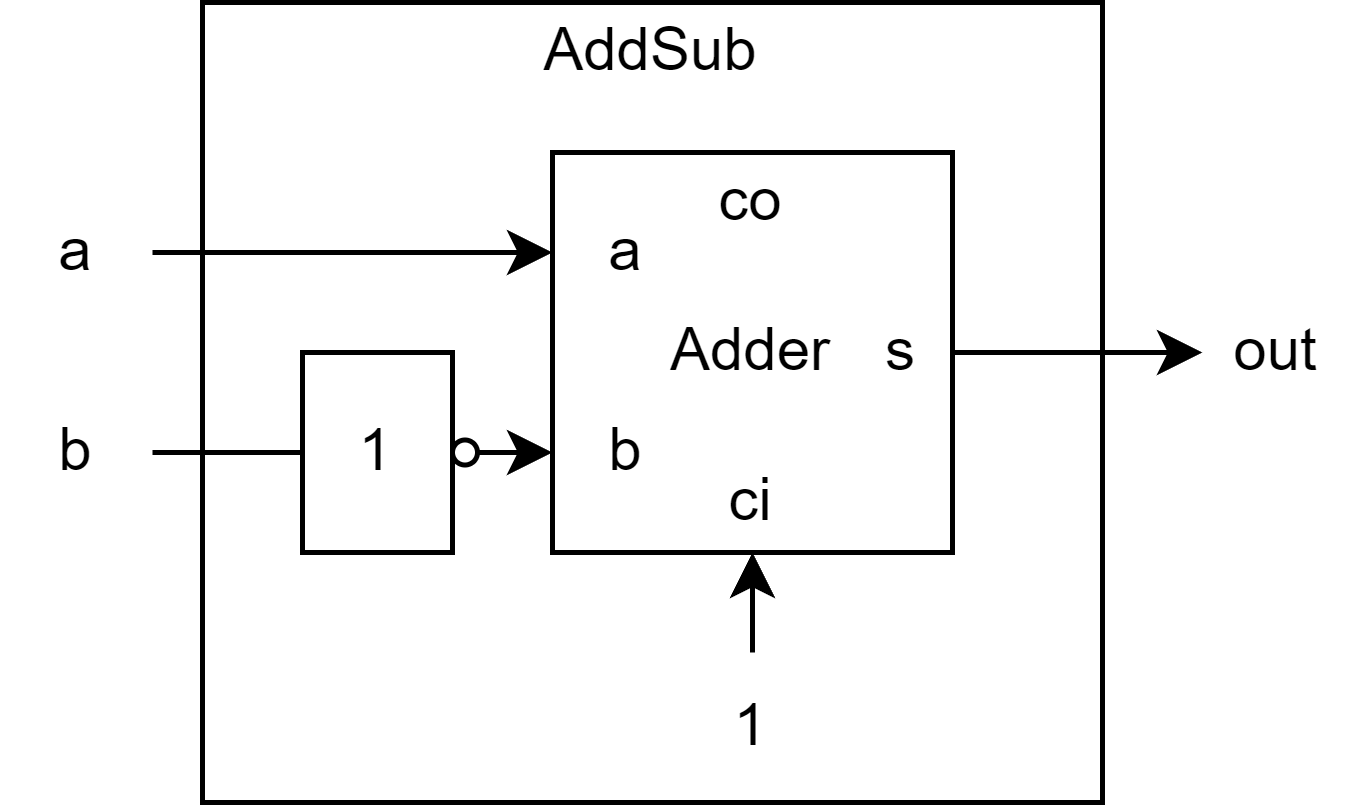


绝大多数数字系统都选择补码进行运算，这是因为它简化了加法和减法的形式。考虑某 n位二进制数x，我们对其按位求反得到 ~x=2^n−1−x，对应的补码即为 x′=2^n−x。由于 n 位二进制数加减法是在模 2^n 的意义下进行的，而 x′≡2^n−x≡−x(mod 2^n)，所以，在补码运算下，有a−b=a+b′。

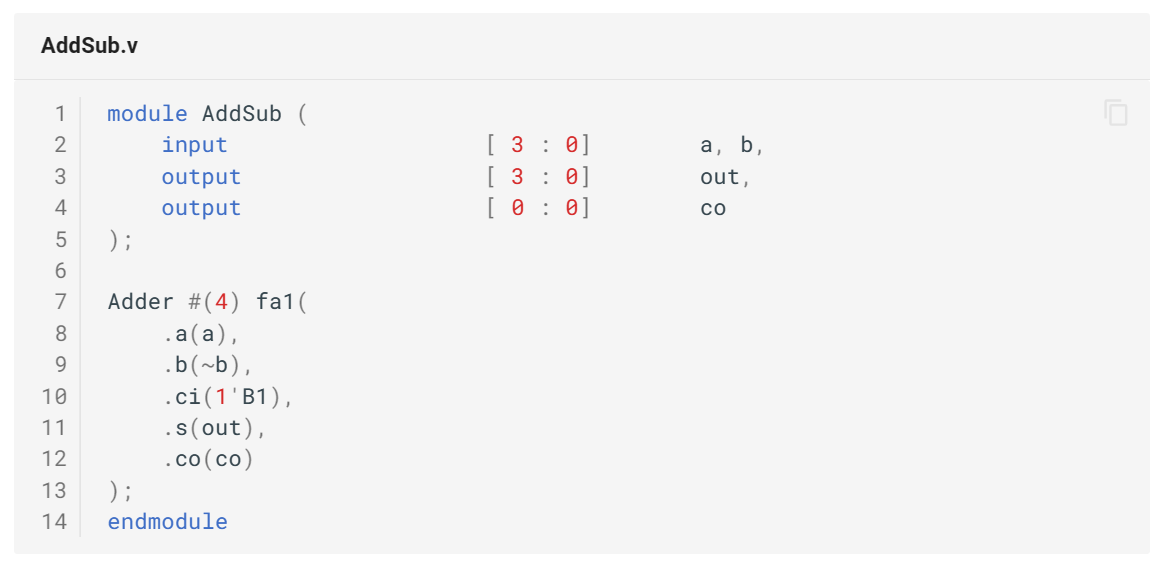
补码最大的意义就是能够将二进制整数的加减法转化为加法与求补码两种运算，让我们可以不再关注符号对运算结果产生的影响（且求补码足够简单）。这一事实启发了我们如何复用加法器来实现减法器。

### ****2.2 用加法器实现减法器****

下面的电路图展示了如何使用加法器进行加减法运算。将加法器的 b 端输入设置为 !b，低位进位端输入设置为 1，输出结果即为 out=a+!b+1=a−b，从而完成了减法运算。

[](https://soc.ustc.edu.cn/Digital/lab6/figs/ALU/sub.png)

基于上面的思路，我们可以用下面的 verilog 代码实现减法器：

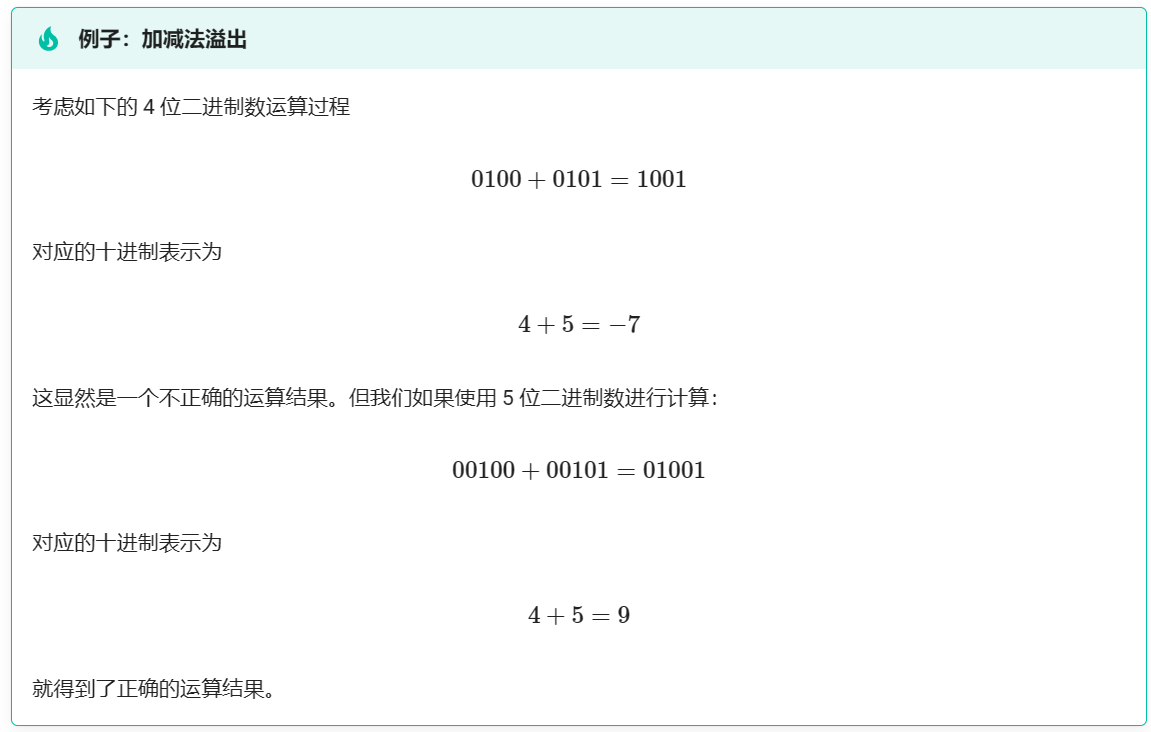


这里将 b 取反后输入，并将 1 作为低位进位。这样，我们就通过复用加法器，完成了减法器的设计。

当然，感兴趣的同学也可以试着对超前进位加法器进行修改，直接得到减法器（就像把全加器改成全减器那样）。不过需要指出的是，这两种实现的性能差距很小，这正是求补码操作足够简单带给我们的好处（只需要通过一个非门的延迟，并把 1 作为低位输入）。

### ****2.3 溢出检测****

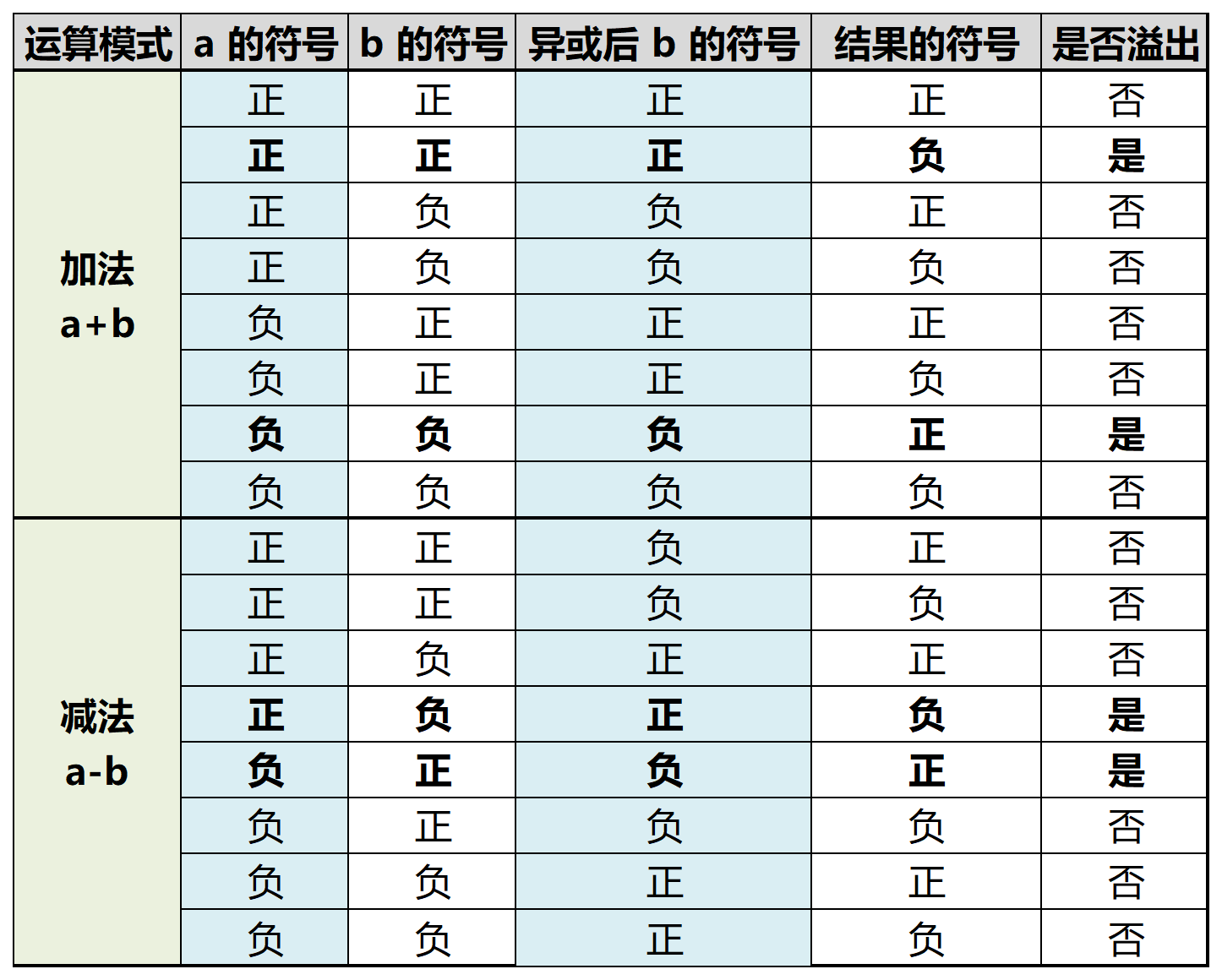
溢出是加减法运算时的一个重要问题。当然，在 ALU 的加减法运算中，我们并不考虑溢出的影响（这一问题的影响由程序自身承担）。但是，当我们将加减法复用到其他模块中，比如比较运算中的时候，溢出可能就会成为影响结果判断的重要因素之一了。考虑如下的例子：



n位补码表示的二进制数表示范围为 −2^(n−1)∼2^(n−1)−1，共计 2^n−1 个数。因此当运算结果超出了这个范围时，我们就称运算结果发生了溢出。对于上面的例子，4 位二进制数的补码表述范围为 −8∼7，因此正确结果 9 就会发生溢出。而 5 位二进制数的补码表示范围更大，为 −16∼15，因此不会发生溢出。

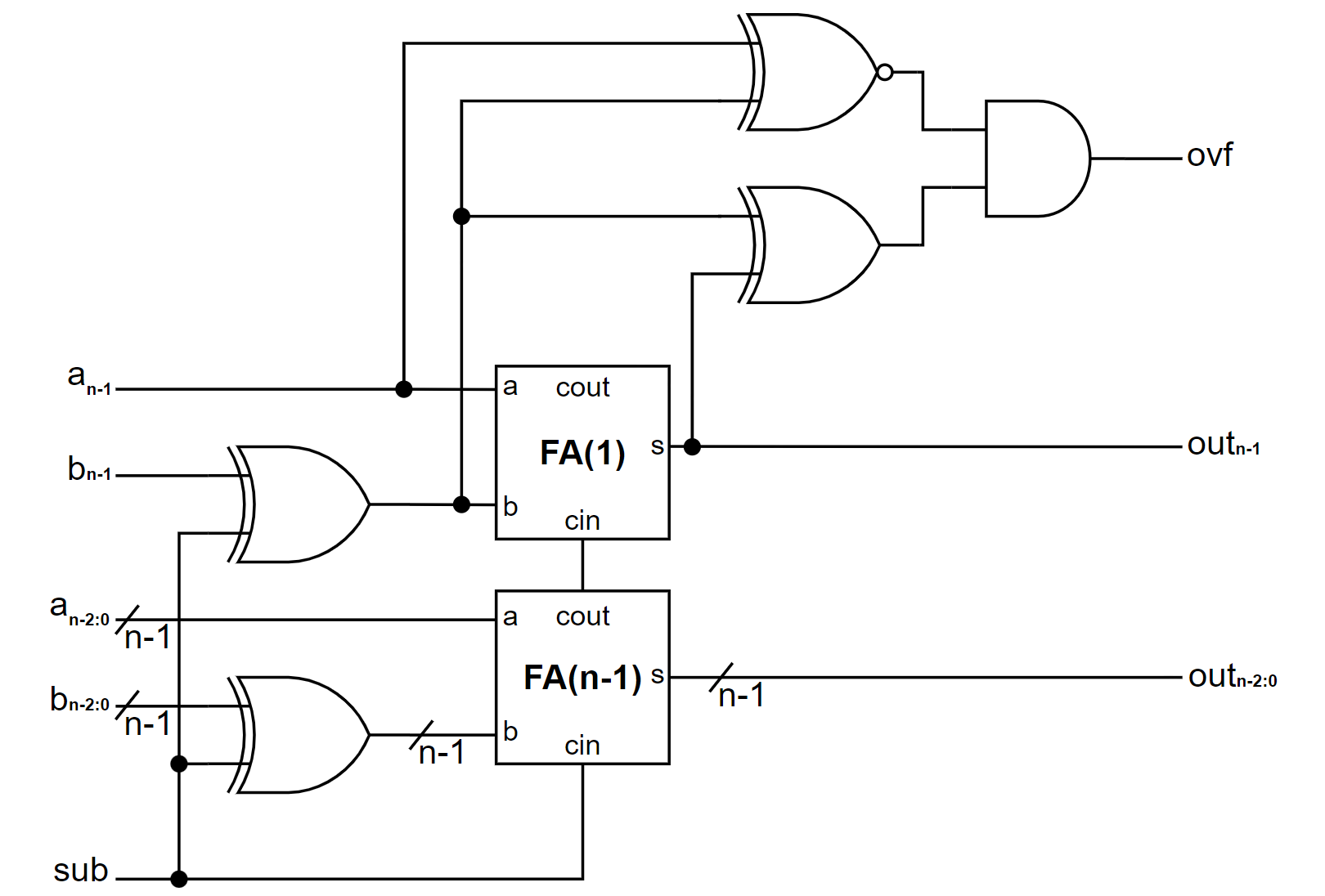
更精细地说，有两种类型的溢出：如果想要的结果比能表示的最大值还要大，那么就叫做正溢出；想要的结果比能表示的最小值还要小，就叫做负溢出。

如何判断结果是否溢出呢？一个很自然也很基本的观点是：正数 + 正数 = 正数，负数 + 负数 = 负数。我们可以检查输入的 a、b 的符号位（也就是最高位）与结果 out 的符号位进行判断。下表列举了所有可能的结果：

[](https://soc.ustc.edu.cn/Digital/lab6/figs/ALU/overflow.png)

不难发现，对于加法而言，当且仅当两个原始输入 a、b 的符号相同，但与结果 out 的符号不同时，才会发生溢出。减法可以视作加法来处理。由此，我们便得到了检测溢出的方法。

溢出检测的参考电路图如下：

[](https://soc.ustc.edu.cn/Digital/lab6/figs/ALU/overflow_circuit.png)

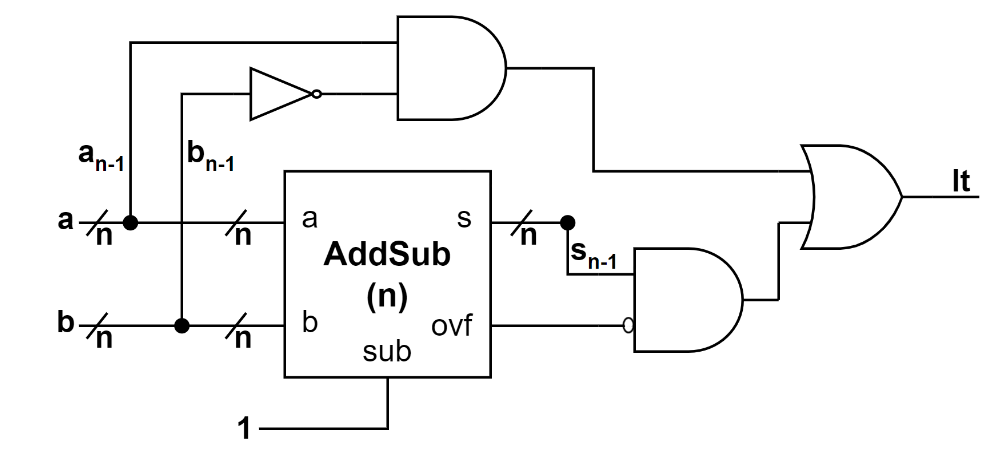
## 3 比较器

在介绍完加减法运算后，让我们来讨论一下比较运算的实现。我们这里实现的是小于比较 (SLT)，即将两个操作数 a、b 均视为补码，进行补码间的小于比较，当 a < b 时结果为 1。无符号小于比较 (SLTU) 将作为作业的一部分。

那么，如何进行基于补码表示的大小比较呢？我们可以借助先前设计的减法电路。不难发现，在不考虑溢出的前提下，若 a−b=0，则 a=b；若 s=a−b 的符号位为 1，则代表 s<0，即 a<b。

然而上面的方法只适用于未发生溢出的情况。经过前面的讨论我们知道，只有当 a、b 异号时，才有可能发生溢出。所以针对 a、b 同号的情况，我们可以放心使用减法器的运算结果。而当 a、b 不同号时，我们可以直接比较它们的符号位：正的那个数必然大于负的那个数。

基于减法器的有符号比较器的电路图如下：



# 思考与练习

1. 请参照实验文档中对 ALU 及其部件的描述，设计并编写一个完整的 32 位 ALU，支持表格中列出的全部十二种运算。其中加法、减法、有符号比较、无符号比较四种运算需要基于超前进位加法器或自己设计的模块实现，而不能直接使用相应的运算符；其他的运算可以使用 Verilog 自带的运算符实现。

2. ALU 中包含三种移位运算，分别是逻辑左移、逻辑右移和算术右移。此前，我们是使用 Verilog 中自带的符号实现的，而在本小题中，我们需要自己去实现移位操作。在这里，我们只实现逻辑右移 SRL 和算术右移 SRA。

可以注意到，常数位移事实上可以通过位拼接来实现，举例来说，对于 src0[31:0]，逻辑左移 3 位，那么 src0<<3 = {src0[28:0],3'b0}；逻辑右移 2 位，那么 src0>>2 = {2'b0,src0[31:2]}；算术右移 2 位，那么 src0>>>2 = {{2{src0[31]}},src0[31:2]}。

对应于前面的超前进位加法器和层次扩展的设计，移位器也有高效但消耗资源、耗时但节省资源的两种实现方式：

* 我们可以枚举所有的右操作数，并通过对应的位拼接来实现；注意到右操作数的范围是 0~31，所以我们需要枚举 32 种情况。这种实现方式只需要经过一次查找表和一个位拼接的延迟，就可以查出对应的结果，但是需要消耗大量的资源，共需要 32 个位拼接单元。
* 我们也可以按照右操作数 src1[4:0] 二进制的各位数字，对 src0 连续地进行 16、8、4、2、1 移位。举例来说，src1[4:0]=5'b10110 时，可以用位拼接对 src0 移 16 位，再移 4 位，最后移 2 位。这种实现方式最多需要经过五次位拼接操作和多选器的延迟，耗时相对较长，但只需要 5 个位拼接单元，资源压力较小。（类似快速幂的思路）

请你尝试实现这两种移位器，并尝试类似加法器的设计过程，对两种方法做一结合，得到在时间和空间上都占优的方法。