

二进制的加法



$$\begin{array}{r}
 1\ 1\ 0\ 1 \\
 +\ 0\ 1\ 1\ 0 \\
 \hline
 1\ 0\ 0\ 1\ 1
 \end{array}$$

被加数 A
加数 B
和 S

两个4-bit二进制数相加

1. 两个1-bit二进制数相加
2. 进位输入参与运算
3. 可以产生进位输出

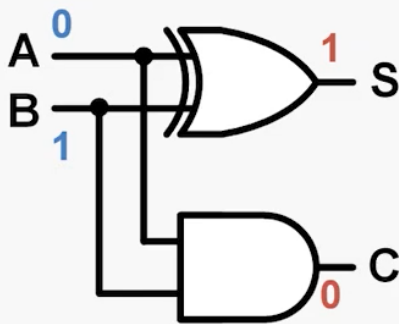
利用逻辑操作实现加法，真是巧妙

半加器 (Half Adder)



半加器的功能是将两个一位二进制数相加

- 输入端口A、B
- 输出端口S (和)、C (进位)



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

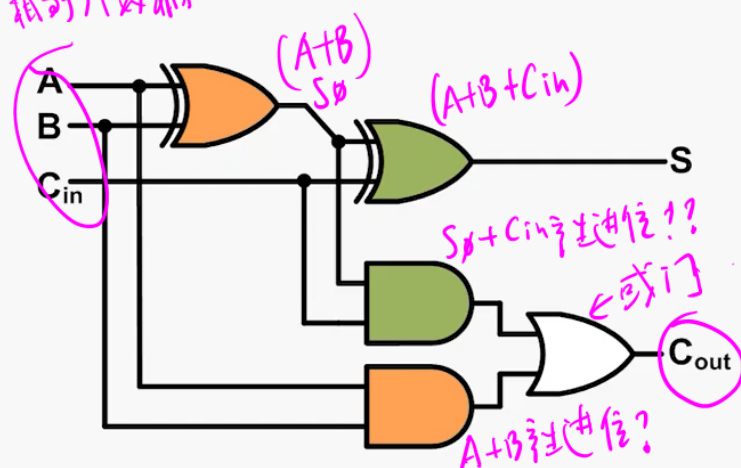
$A+B$ 可以改写成 (1位运算) :

- $S = A \oplus B \Rightarrow$ 加和结果 ;
- $C = A \& B \Rightarrow$ 进位结果 ;

全加器 (Full Adder)



- 全加器由两个半加器构成
 - 输入端口A、B、 C_{in} (进位输入)
 - 输出端口S (和)、 C_{out} (进位输出)



A	B	C _{in}	C _{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

4-bit加法器

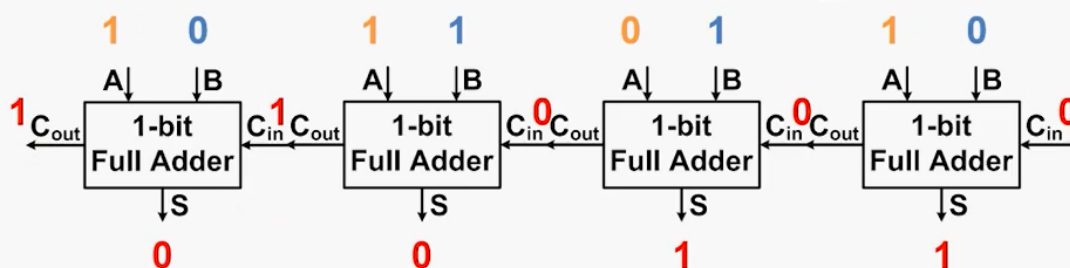


$$\begin{array}{r} 1101 \\ + 0110 \\ \hline 10011 \end{array}$$

被加数 A
加数 B
进位 C
和 S

两个4-bit二进制数相加

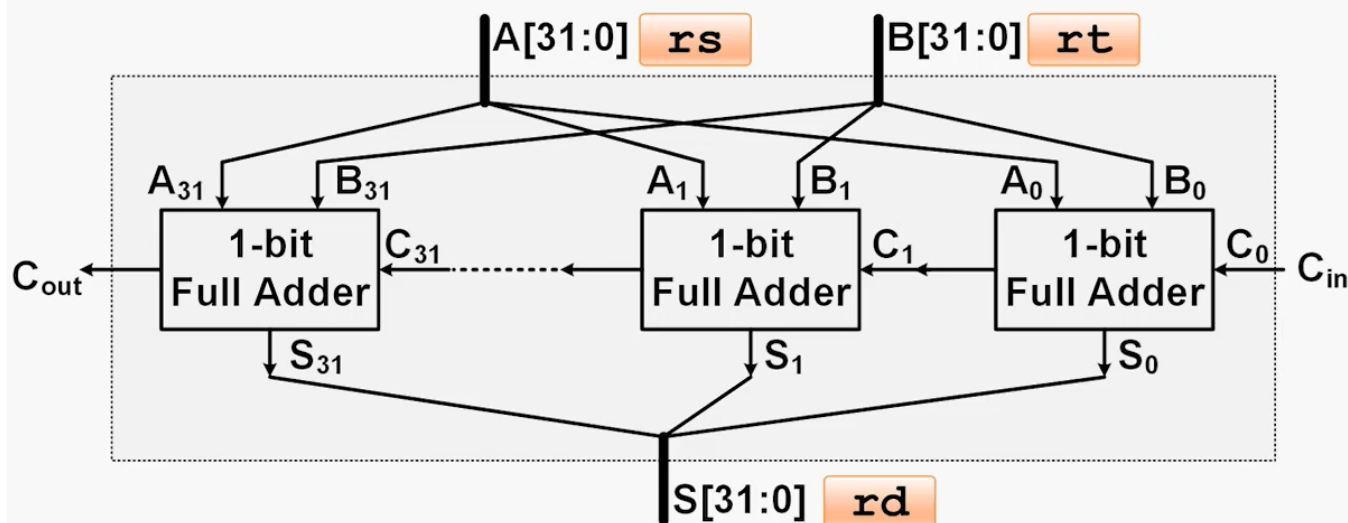
由四个全加器构成的4-bit加法器



加法运算的实现示例

add rd, rs, rt

addu rd, rs, rt



“进位”和“溢出”示例

注意区分“进位”和“溢出”

- 有“溢出”时，不一定有“进位”
- 有“进位”时，不一定有“溢出”

无符号数：3 + 5 = 8

有符号数：3 + 5 = (-8)

```

  0 0 1 1
+ 0 1 0 1
-----
0 1 0 0 0

```

有“溢出”，无“进位”

无符号数：14 + 12 = 10

有符号数：(-2) + (-4) = (-6)

```

  1 1 1 0
+ 1 1 0 0
-----
1 1 0 1 0

```

有“进位”，无“溢出”

溢出是针对有符号数来说的。

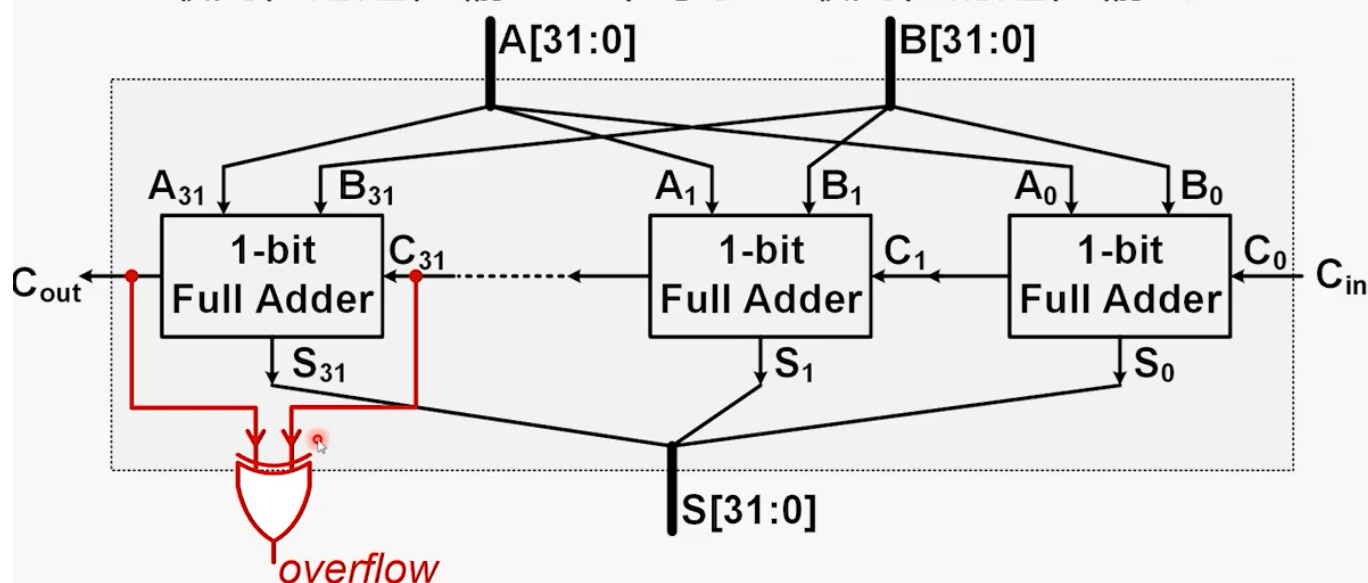
检查加法运算结果是否溢出

- “溢出” (overflow)
 - 运算结果超出了正常的表示范围
- “溢出” 仅针对有符号数运算
 - 两个正数相加，结果为负数
 - 两个负数相加，结果为正数

“溢出” 的检查方法



- “最高位的进位输入” 不等于 “最高位的进位输出”



加法运算不区分有符号数和无符号数，全都是通过上面一套加法器来实现，按位进行相加。

MIPS根据溢出是否要处理，分为两个指令。

对“溢出”的处理方式：MIPS

提供两类不同的指令分别处理

(1) 将操作数看做有符号数，发生“溢出”时产生异常

- `add rd,rs,rt` # $R[rd]=R[rs]+R[rt]$
- `addi rt,rs,imm` # $R[rt]=R[rs]+SignExtImm$

(2) 将操作数看做无符号数，不处理“溢出”

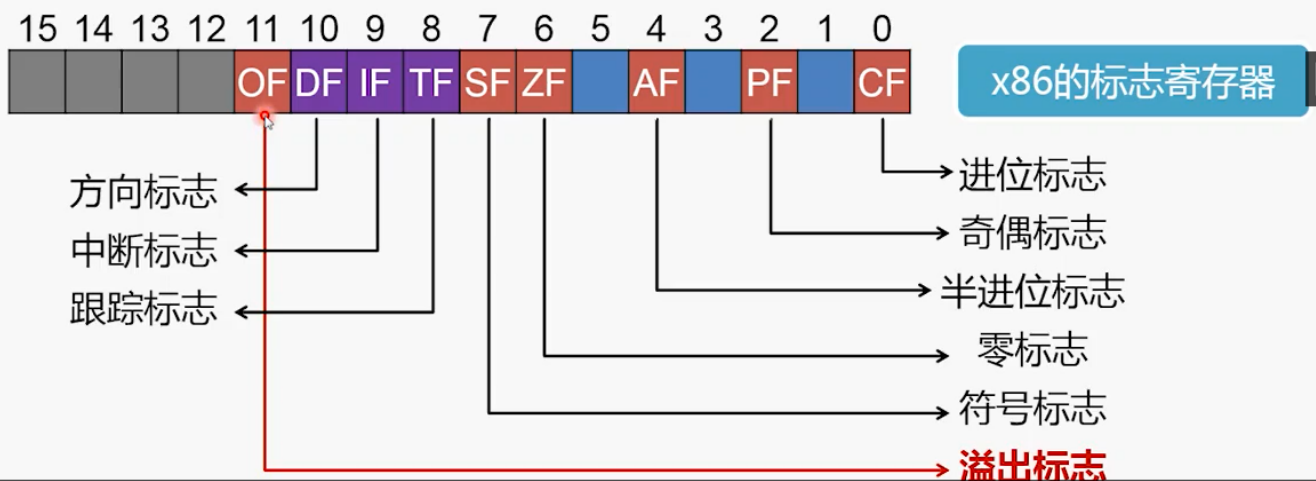
- `addu rd,rs,rt` # $R[rd]=R[rs]+R[rt]$
- `addiu rt,rs,imm` # $R[rt]=R[rs]+SignExtImm$

x86根据溢出标志位来检测是否发生了溢出。

对“溢出”的处理方式：x86

溢出标志OF (Overflow Flag)

- 如果把操作数看做有符号数，运算结果是否发生溢出
- 若发生溢出，则自动设置OF=1；否则，OF=0



减法运算

减法运算均可转换为加法运算

$$\circ A - B = A + (-B)$$

补码表示的二进制数的相反数

◦ 转换规则：按位取反，末位加一

在加法器的基础上实现减法器

$$\circ A + (-B) = A + (\sim B + 1)$$

$$\begin{array}{r} x: 01001011 \\ \sim x: 10110100 \\ \hline -1: 11111111 \\ \downarrow \\ x + (\sim x) = -1 \\ \downarrow \\ (\sim x) + 1 = -x \end{array}$$

联想到 $-B \text{ AND } A$ ， $-B$ 跟 A 进行按位与操作，其实是做**ALIGN DOWN** 如果 B 是一个只有1bit的数值：

```
B:      0001 0000
-B:      1111 0000
~B + 1 : 1110 1111 + 1 => 1111 0000
-B AND A: 等价于将A的低4bits清零
```

利用加法器来实现减法很巧妙：

减法运算的实现示例

$$A - B = A + (\sim B + 1)$$

中国大学MOOC

