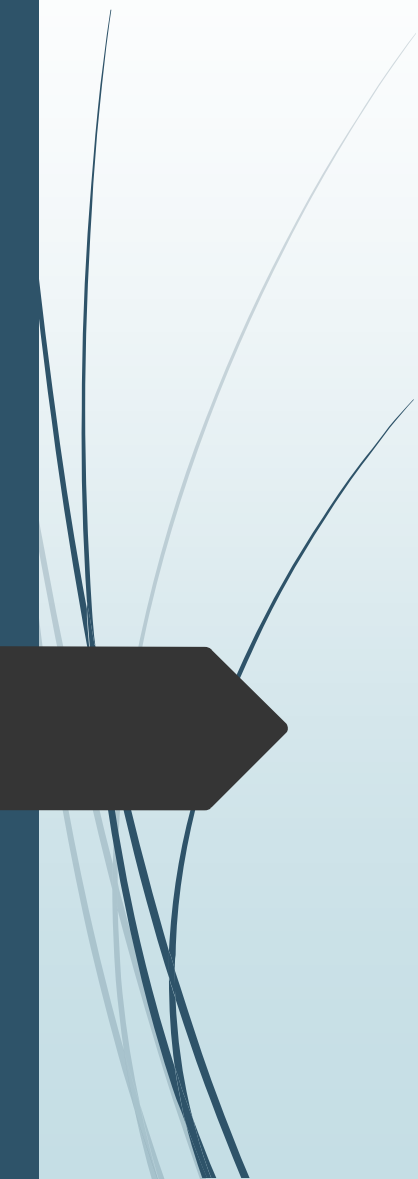


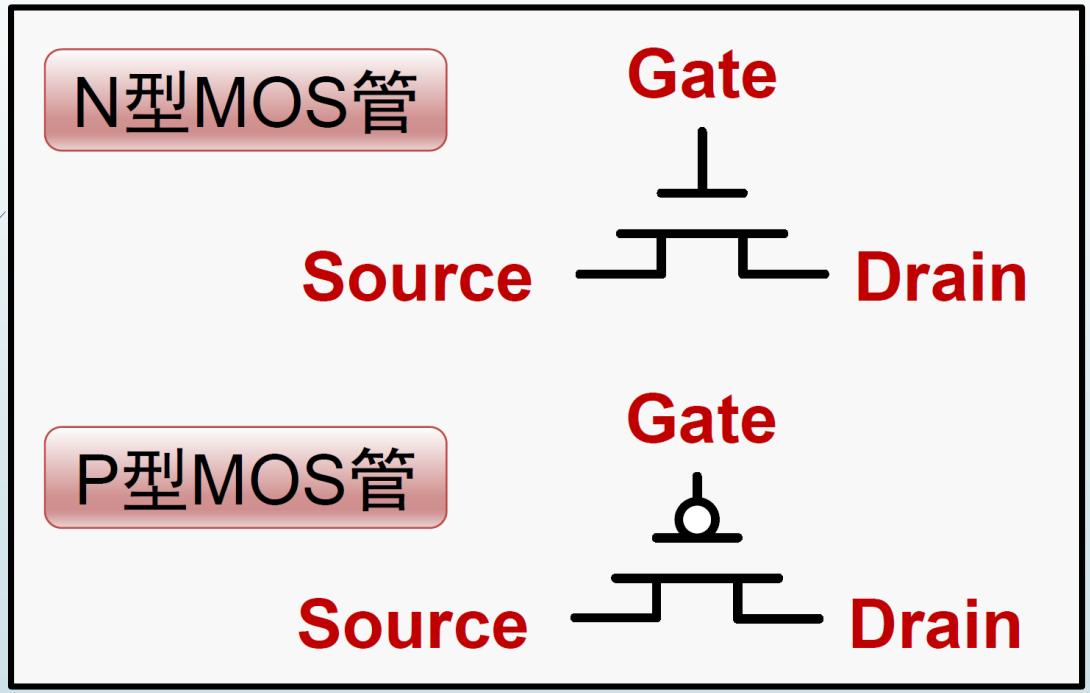
Bit, Bits & Bitset

计算机如何运算？

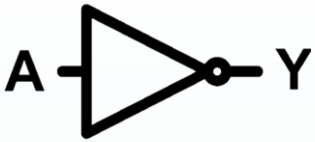
值

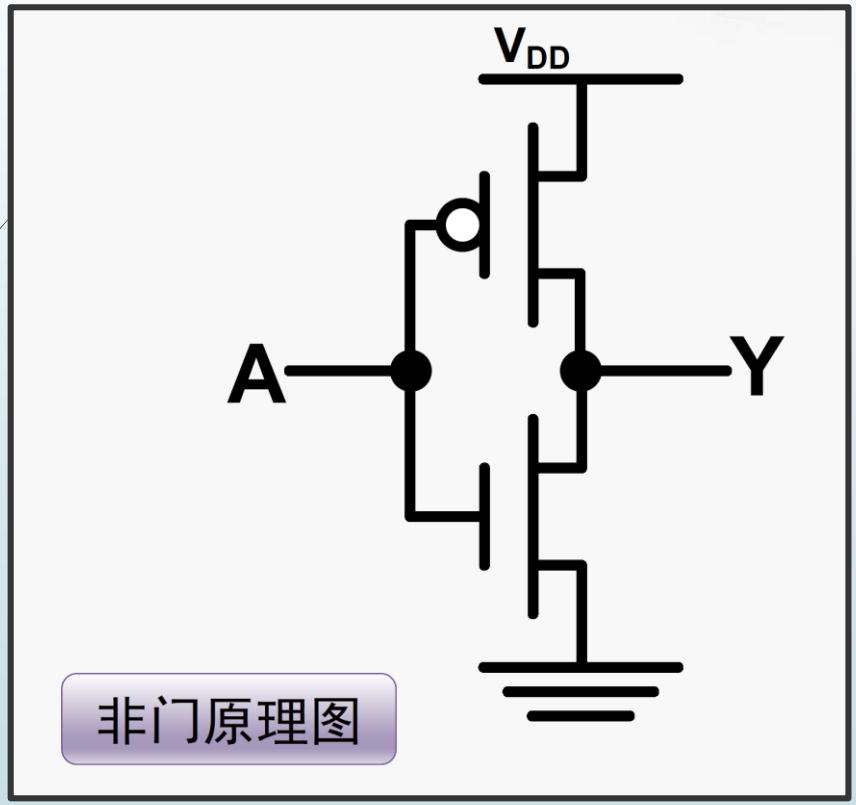
- 高电平
- 低电平

MOS管

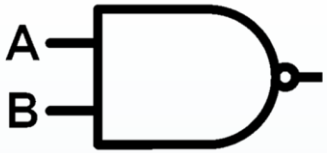


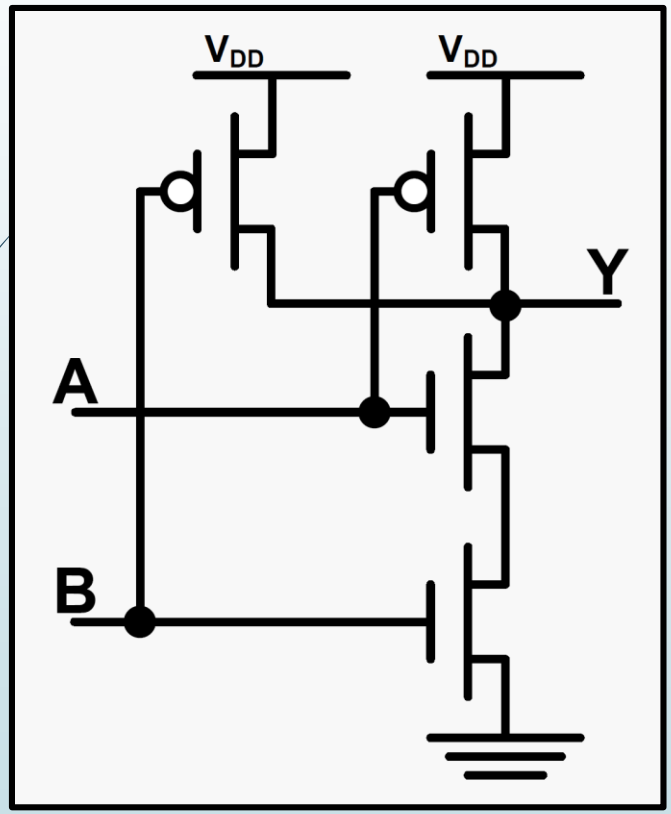
逻辑门 – 非门



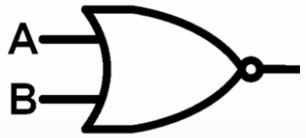


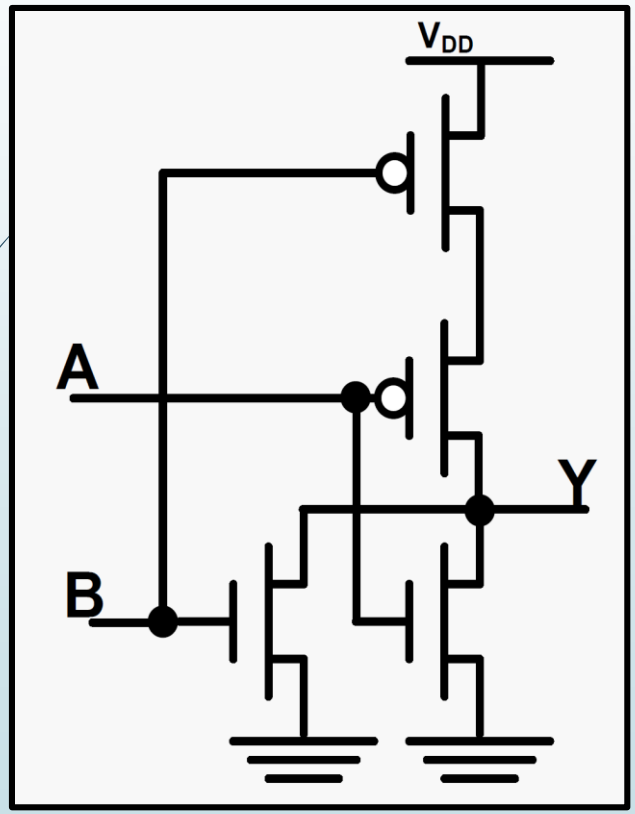
逻辑门 – 与非门



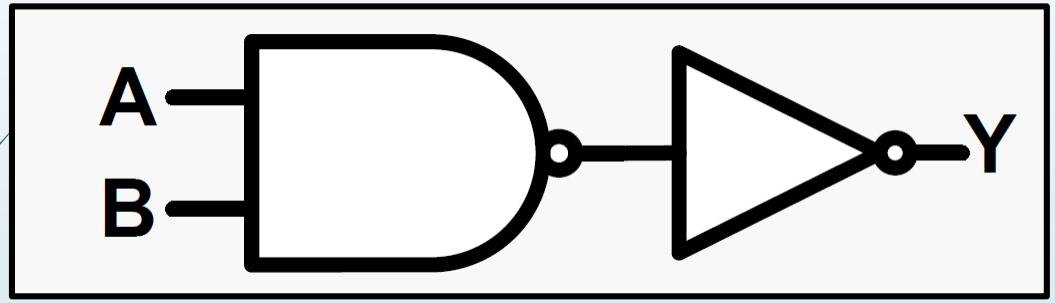


逻辑门 – 或非门

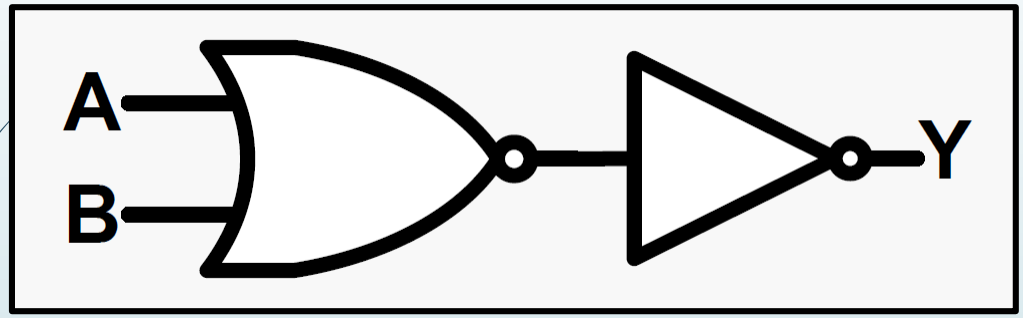




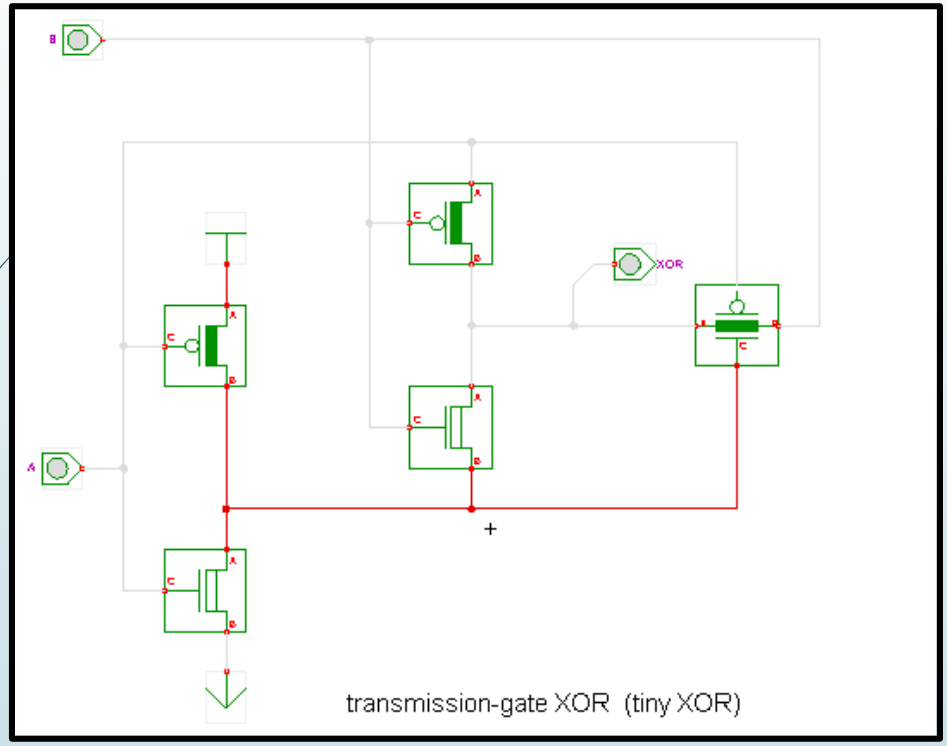
逻辑门 – 与门



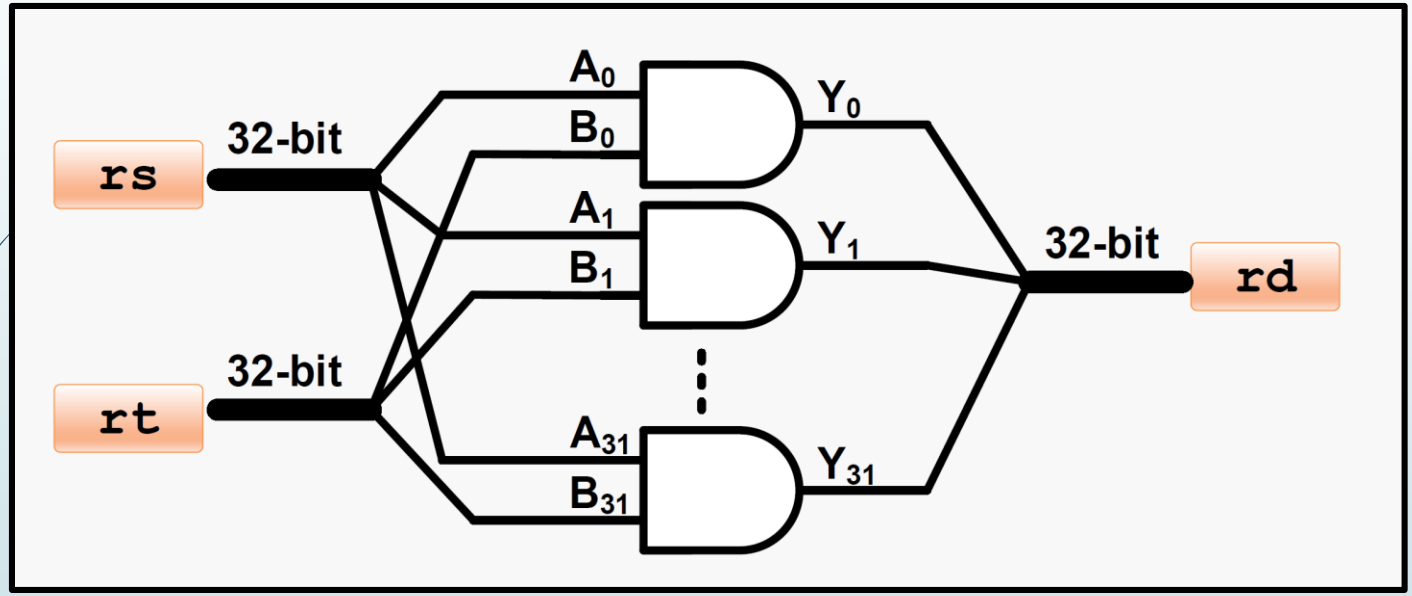
逻辑门 – 或门



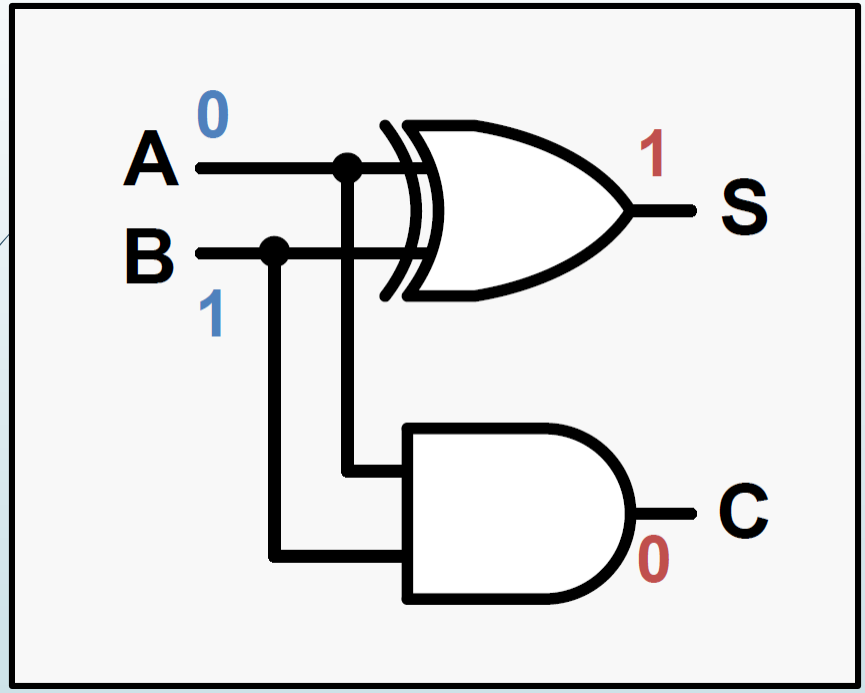
逻辑门 – 异或门



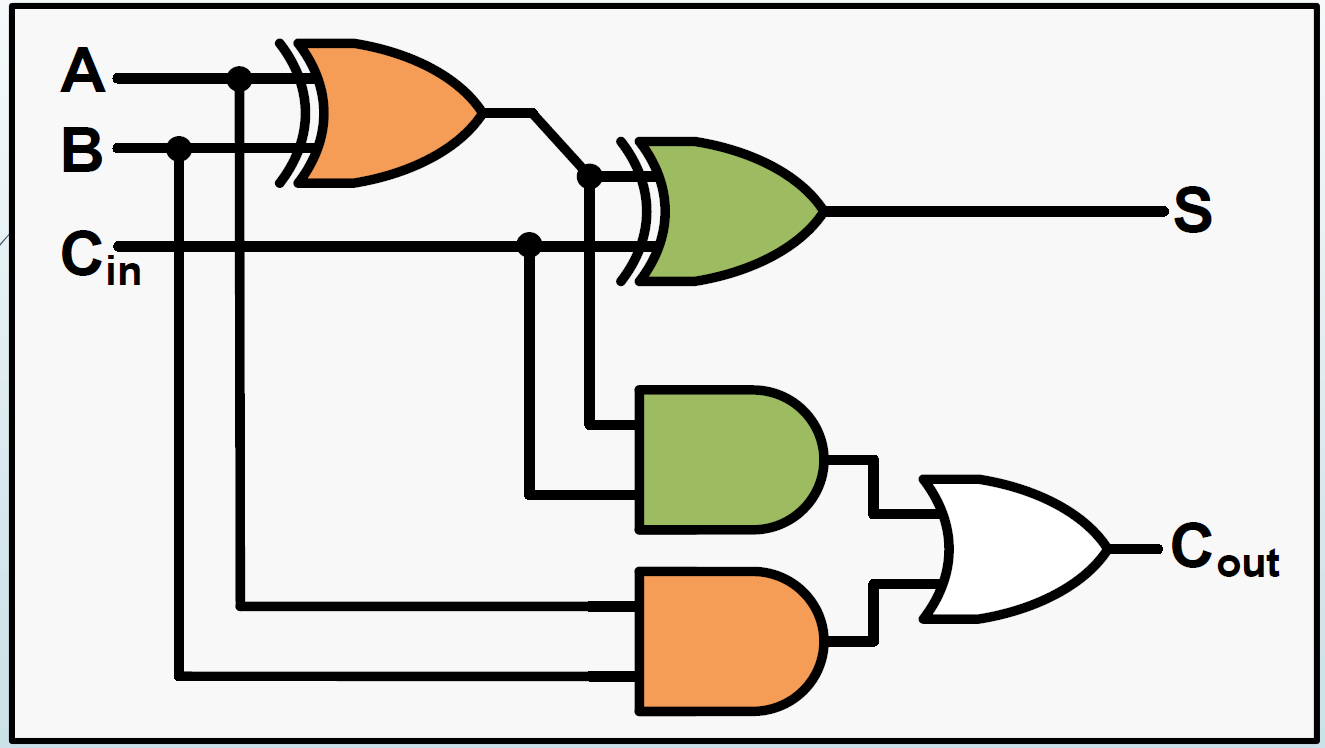
实现位运算



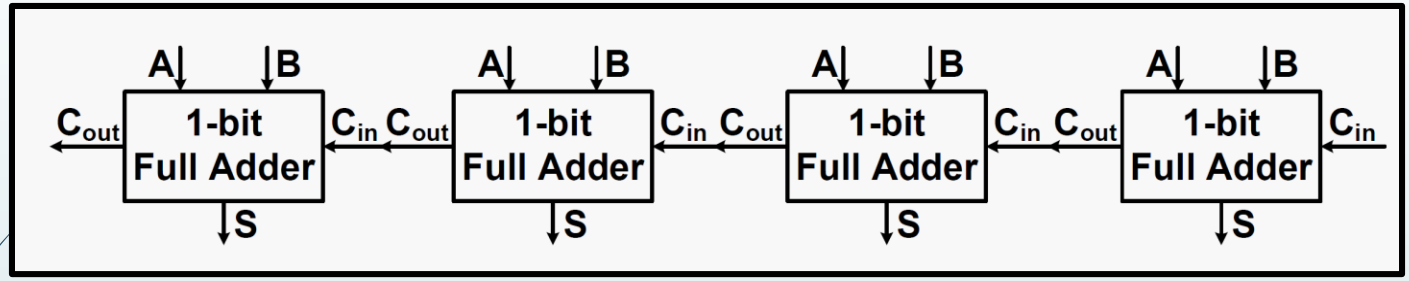
加法 – 半加器



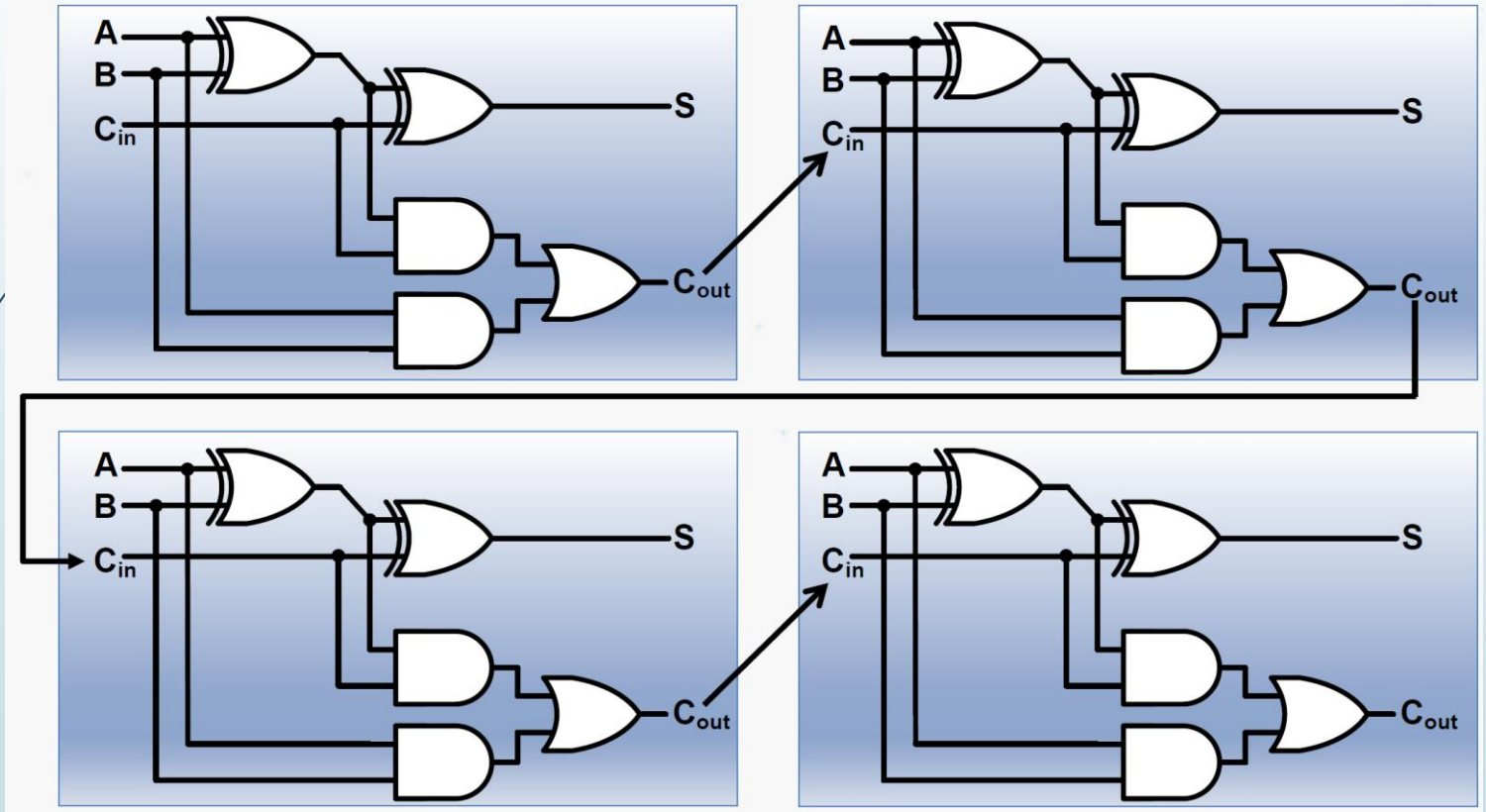
加法 – 全加器



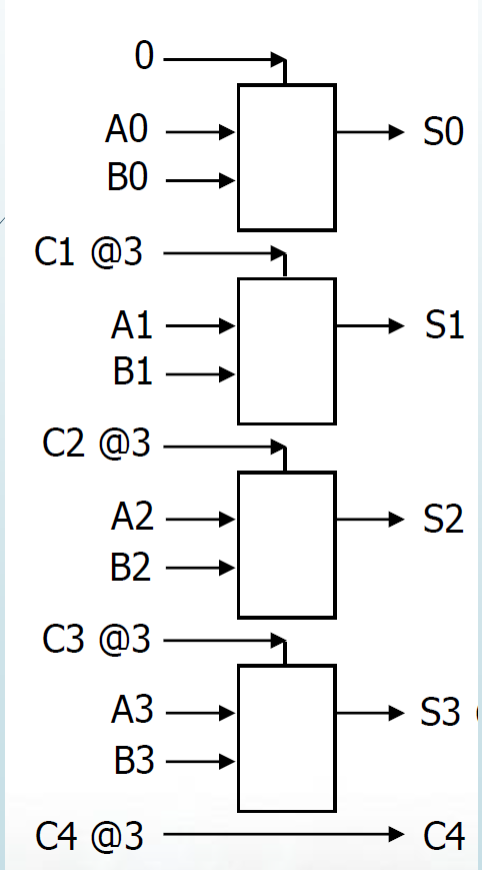
实现多位加法



实现多位加法



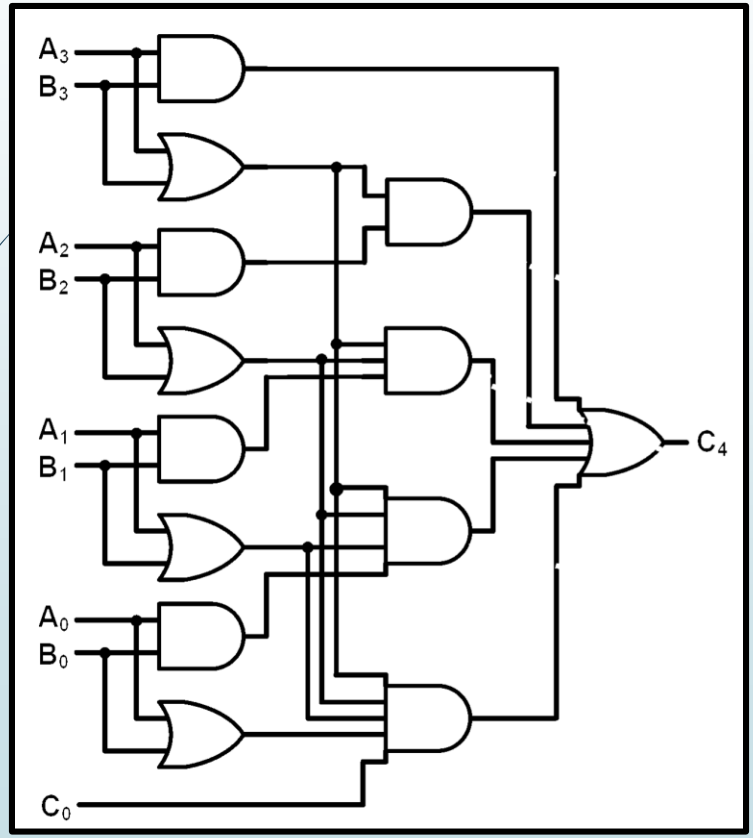
实现多位加法



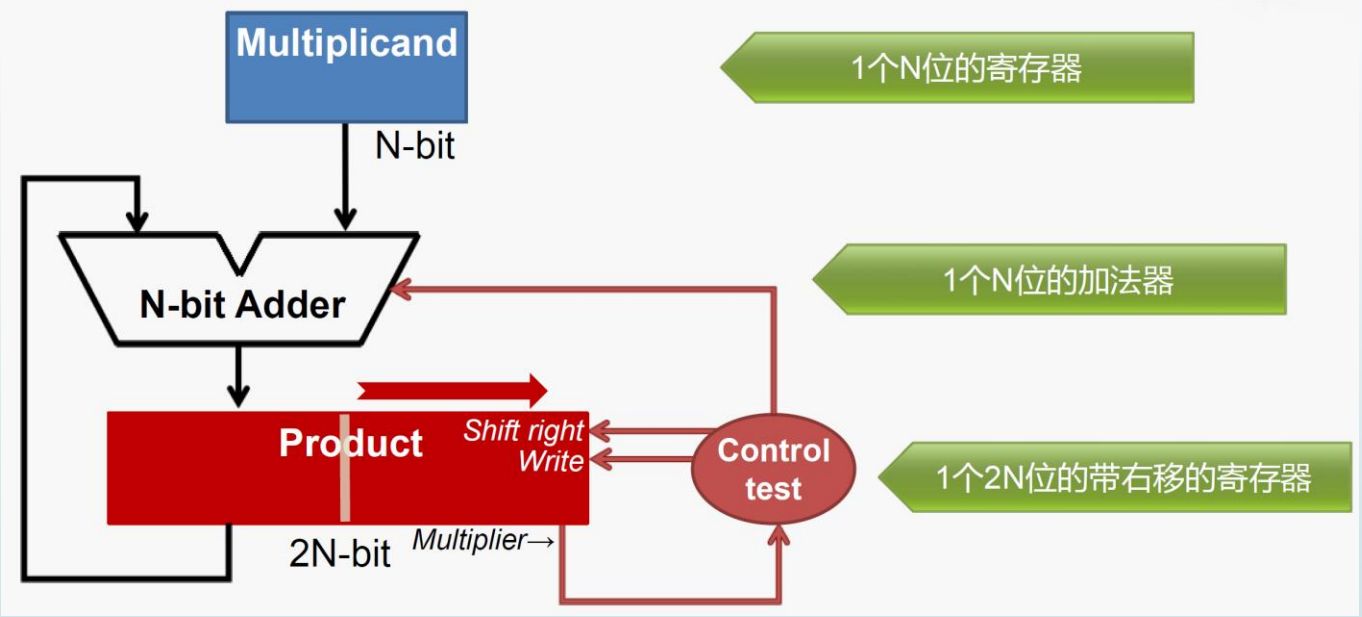
实现多位加法

- $C1 = G0 + P0\ C0$
- $C2 = G1 + P1\ C1 = G1 + P1\ G0 + P1\ P0\ C0$
- $C3 = G2 + P2\ C2 = G2 + P2\ G1 + P2\ P1\ G0 + P2\ P1\ P0\ C0$
- $C4 = G3 + P3\ C3 = G3 + P3\ G2 + P3\ P2\ G1 + P3\ P2\ P1\ G0 + P3\ P2\ P1\ P0\ C0$

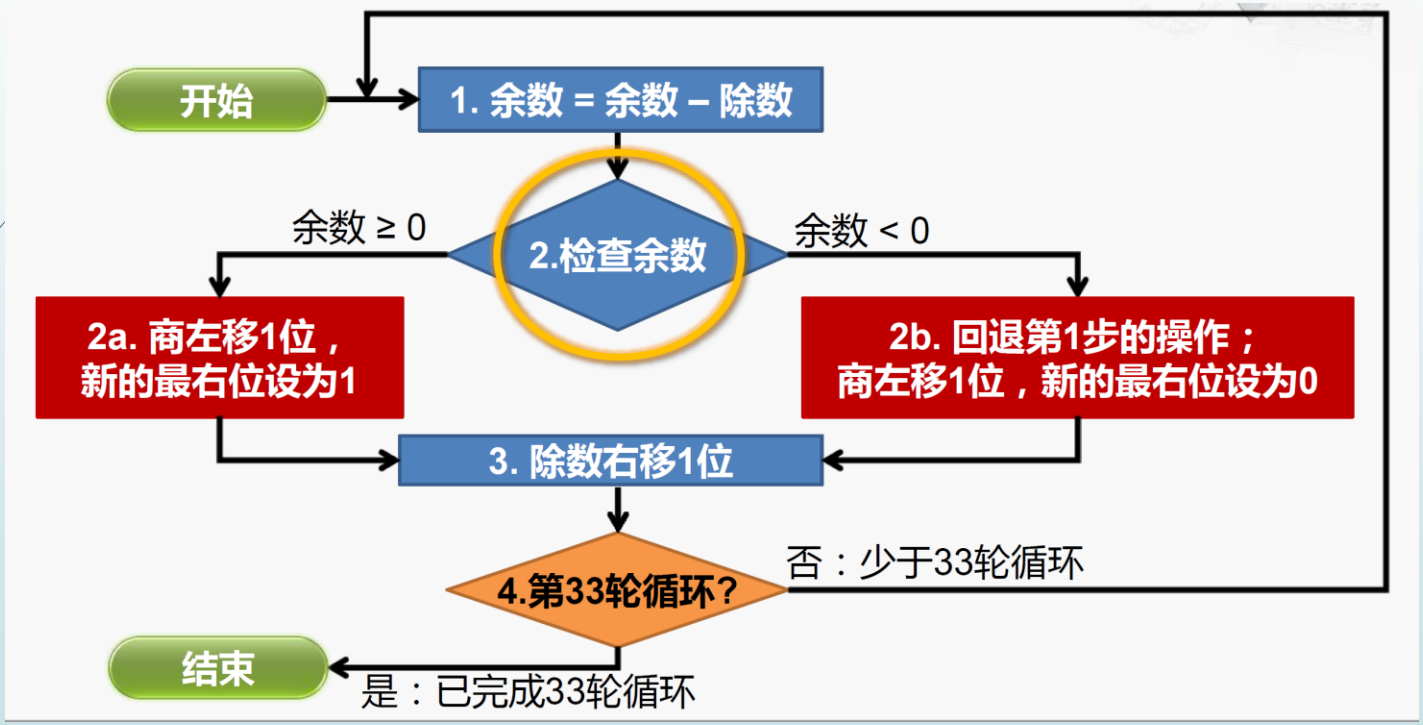
实现多位加法



实现乘法


Multiplicand
N-bit
N-bit Adder
Product
Shift right
Write
Control
test
2N-bit
Multiplier→
1个N位的寄存器
1个N位的加法器
1个2N位的带右移的寄存器

实现除法


开始
1. 余数 = 余数 – 除数
2.检查余数
余数 ≥ 0
余数 < 0
2a. 商左移1位，
新的最右位设为1
2b. 回退第1步的操作；
商左移1位，新的最右位设为0
3. 除数右移1位
4.第33轮循环?
否：少于33轮循环
结束
是：已完成33轮循环

预备知识

- 整数的表示
- 带符号整数的补码表示

- 集合的概念
- C++语言

约定

- 二进制表示最高位(MSB)为最左边，最低位(LSB)为最右边，最低位标号为0
- C++语言
- GCC编译器
- x86指令集

结构

- 位运算
- 位修改和位查询
- 用Bits表示集合
- 位运算的其他应用
- 例题

位运算

与、或、非和异或

逻辑运算

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

p	$\neg p$
0	1
1	0

与、或、非和异或

位运算

- 逻辑运算在整数上的扩展
- 结果的每个二进制位等于运算数的对应二进制位进行相应逻辑运算的结果
- 带符号整数符号位？
- 一视同仁
- 交换律、结合律（与、或、异或）

移位运算

- 逻辑移位
- 算数移位
- 循环移位
- 带进位循环移位

逻辑移位

逻辑左移

- shl x y
- 把x的每个二进制位向左移动y位，移动造成的最右边的空位由0补足，最左边的数溢出。

逻辑右移

- shr x y
- 把x的每个二进制位向右移动y位，移动造成的最左边的空位由0补足，最右边的数溢出。

算术移位

算术左移

- sal x y
- 与逻辑左移完全相同

算术右移

- sar x y
- 与逻辑右移大体相同，唯一的区别在于移动造成的最左边的空位由符号位（最高位）补足而不是由0补足。

循环移位

循环左移

- rol x y
- 把x的每个二进制位向左移动y位，移动造成的最右边的空位由最左边溢出的位补足。

循环右移

- ror x y
- 把x 的每个二进制位向右移动y位，移动造成的最左边的空位由最右边溢出的位补足。

带进位循环移位

- 将CF作为数的一部分看待

注意事项

- x左移/右移y位，y需非负并且小于x的位宽
- 超出范围
- x86：只取y的后几位
- C标准：未定义

数学意义

逻辑移位

- 处理无符号整数
- 左移和右移分别与无符号整数的$x \cdot 2^y$和$x \div 2^y$具有相同的效果（向下取整）

算术移位

- 处理带符号整数
- 左移和右移分别与带符号整数的$x \cdot 2^y$和$x \div 2^y$具有相同的效果（向下取整）

C/C++中的移位运算

- 无符号整数的移位使用逻辑移位，有符号整数的移位使用算术移位。
- 无论是无符号整数还是带符号整数，我们都可以放心的使用左移和右移来代替乘以二的幂或除以二的幂的操作。
- 没有专门的对带符号整数进行逻辑右移的运算符
- 我们可以通过强制类型转换将它转换成无符号整数后再进行运算。

C/C++中的移位运算

- 没有提供循环移位操作符
- 通过其他运算的组合来实现
- 应用于32位整数的两个例子：

  (x << y) | (x >> (31 ^ y))

  (x >> y) | (x << (31 ^ y))

位修改和位查询

在二进制位的层面上对整数进行一些修改和查询

指令、内建函数以及基本位运算的组合

读某些位

- 读取x的第pos个二进制位
- 将x右移pos位，使要读取的位移到最低位，再通过& 1将其取出

```
bool readBit(u32 x, int pos) {
    return (x >> pos) & 1;
}
```

读某些位

- 读x的第pos位开始的cnt位
- 首先将x右移pos位，再通过$\&\ mask$来取出最后cnt位。
- $mask$ 的后cnt个位为1，其余位为0，可以通过$(1\ <<\ cnt)\ -\ 1$得到

```
u32 readBits(u32 x, int pos, int cnt) {
    return (x >> pos) & ((1u << cnt) - 1);
}
```

读某些位

- 与运算
- 通过将原数和一个遮罩进行与运算，可以达到保留指定一些位（将遮罩的对应位设为1），清零其它位（遮罩的对应位设为0）的目的。

改某些位

- “与”、“或”和“异或”运算的应用

将某些位置为1

- 构造遮罩
- 对于要改为1的位，我们将遮罩的对应位设为1，否则将对应位设为0
- 原数与遮罩进行或运算

将某些位置为0

- 构造遮罩
- 对于要修改的位，我们将遮罩的对应位设为0，否则将其设为1
- 原数与遮罩进行与运算

将某些位置取反

- 构造遮罩
- 如果我们要取反某位，则将遮罩的对应位设为1，否则将其设为0
- 原数与遮罩进行异或运算

求1的个数

- 转化为求二进制各位和
- 分治
- 每次将整个数分成两个部分，分别求出每个部分的和，再将它们相加
- 利用位运算，并行完成每一层的工作

求1的个数

```
int bitCount_1(u32 x) {
    x = ((x & 0xAAAAAAAAu) >>  1) + (x & 0x55555555u);
    x = ((x & 0xCCCCCCCCu) >>  2) + (x & 0x33333333u);
    x = ((x & 0xF0F0F0F0u) >>  4) + (x & 0x0F0F0F0Fu);
    x = ((x & 0xFF00FF00u) >>  8) + (x & 0x00FF00FFu);
    x = ((x & 0xFFFF0000u) >> 16) + (x & 0x0000FFFFu);
    return x;
}
```

求1的个数

- 第一步，有32个项需要相加，每一项占1 bit
- 将其中的奇数项和偶数项分别取出来，并将奇数项右移1位和偶数项“对齐”，然后将他们相加
- 作用：16对1 bit的项分别相加，并将结果存放在了原来这两项所在的2 bit空间上。
- 第二步，有16个项需要相加，每一项占2 bit。我们将奇偶项分别相加，形成8个4 bit的项。
- ……

求1的个数

- 继续优化

```
int bitCount_2(u32 x) {
    x-= ((x & 0xAAAAAAAAu) >> 1);
    x = ((x & 0xCCCCCCCCu) >> 2) + (x & 0x33333333u);
    x = ((x >>  4) + x) & 0x0F0F0F0Fu;
    x = ((x >>  8) + x) & 0x00FF00FFu;
    x = ((x >> 16) + x) & 0x0000FFFFu;
    return x;
}
```

求1的个数

- 第一个区别

```
x = ((x & 0xAAAAAAAAu) >>  1) + (x & 0x55555555u);
                          ↓
x-= ((x & 0xAAAAAAAAu) >> 1);
```

- Why ?

求1的个数

- `x = (x & 0xAAAAAAAAu) + (x & 0x55555555u)`
- `ans = ((x & 0xAAAAAAAAu) >> 1) + (x & 0x55555555u)`
- 两者作差
- `x - ans = ((x & 0xAAAAAAAAu) >> 1)`
- `ans = x - ((x & 0xAAAAAAAAu) >> 1)`

求1的个数

- 第二个区别

```
x = ((x & 0xF0F0F0F0u) >>  4) + (x & 0x0F0F0F0Fu);
                          ↓
x = ((x >>  4) + x) & 0x0F0F0F0Fu;
```

- 作用：将4对4 bit整数相加
- 每个整数最大只可能是4，两个数相加的结果也能在4 bit的空间存下
- (x >> 4) + x可以正确地依次求出第0个数加第1个数，第1个数加第2 个数，第2个数加第3个数……
- 从中取出我们需要的结果。

求1的个数

- Keep going !

```
int bitCount_3(u32 x) {
    x-= ((x & 0xAAAAAAAAu) >> 1);
    x = ((x & 0xCCCCCCCCu) >> 2) + (x & 0x33333333u);
    x = ((x >> 4) + x) & 0x0F0F0F0Fu;
    x = (x * 0x01010101u) >> 24;
    return x;
}
```

求1的个数

第四步及以后

↓

```
x = (x * 0x01010101u) >> 24;
```

- 发生了什么？
- 令x = (a<<24)+(b<<16)+(c<<8)+(d<<0)
- 要求a+b+c+d

求1的个数

```
 x * 0x01010101

=(x<<24)+(x<<16)+(x<<8)+(x<<0)

=(a<<48)+(b<<40)+(c<<32)+(d<<24)+
         (a<<40)+(b<<32)+(c<<24)+(d<<16)+
                (a<<32)+(b<<24)+(c<<16)+(d<< 8)+
                       (a<<24)+(b<<16)+(c<< 8)+(d<< 0)

=((a)      << 48) +
 ((a+b)    << 40) +
 ((a+b+c)  << 32) +
 ((a+b+c+d)<< 24) +
 ((b+c+d)  << 16) +
 ((c+d)    << 8 ) +
 ((d)      << 0 )
```

求1的个数

- 前三项：溢出
- 后三项：考虑a,b,c,d的实际意义，后三项之和小于1 << 24，不影响前8位
- 前八位即为a+b+c+d

求1的个数

查表

- 对每个数预处理答案
- 递推
- $f(0) = 0$
- $f(i) = f(i >> 1) + (i \,\&\, 1)$

- 表太大

求1的个数

- 分段
- 预处理所有16位整数的答案，询问时将被询问的整数拆成高16位和低16位分别计算答案

```
int cnt_tbl[65537];
void bitCountPre() {
    cnt_tbl[0] = 0;
    for (int i=1; i<65536; ++i)
        cnt_tbl[i] = cnt_tbl[i >> 1] + (i & 1);
}
int bitCount_4(u32 x) {
    return cnt_tbl[x >> 16] + cnt_tbl[x & 65535u];
}
```

求1的个数

GCC内建函数

- int __builtin_popcount (unsigned int x);
- 对于支持SSE4.2的机器，如果在编译时开启相应开关，则该函数会被翻译成汇编指令popcnt，否则使用查表法计算

求1个数的奇偶性

```
int bitParity(int x) {
  x ^= x >> 16;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return x & 1;
}
```

- 不必非要每次计算相邻两项

求1个数的奇偶性

查表

- 类似于求1的个数

GCC内建函数

- int __builtin_parity (unsigned int x);

翻转位序

- 对于一个32位整数来说，翻转位序是指将它的第0位与第31位交换，第1位与第30位交换，……第i位与第$31-i$位交换，……第15 位与第16位交换
- 32位整数5，对它进行翻转位序将得到2684354560

翻转位序

- 分治
- 将整个数分割成两个部分，分别翻转这两个部分，再将这两个部分对调
- 借由位运算，每一层的工作并行完成

翻转位序

```
u32 bitRev_1(u32 x) {
    x = ((x & 0xAAAAAAAAu) >>  1) | ((x & 0x55555555u) <<  1);
    x = ((x & 0xCCCCCCCCu) >>  2) | ((x & 0x33333333u) <<  2);
    x = ((x & 0xF0F0F0F0u) >>  4) | ((x & 0x0F0F0F0Fu) <<  4);
    x = ((x & 0xFF00FF00u) >>  8) | ((x & 0x00FF00FFu) <<  8);
    x = ((x & 0xFFFF0000u) >> 16) | ((x & 0x0000FFFFu) << 16);
    return x;
}
```

翻转位序

查表

- 预处理所有16位整数的结果
- 递推
- $f(0) = 0$
- $f(i) = (f(i >> 1) >> 1) | ((i \& 1) << 15)$
- 将待翻转的32位整数拆成两个16位整数，通过查表来得到这两个16位整数的答案，再将它们对调

翻转位序

```
u32 rev_tbl[65537];
void bitRevPre() {
    rev_tbl[0] = 0;
    for (int i=1; i<65536; ++i)
        rev_tbl[i] = (rev_tbl[i >> 1] >> 1) | ((i & 1) << 15);
}
u32 bitRev_2(u32 x) {
    return (rev_tbl[x & 65535u] << 16) | rev_tbl[x >> 16];
}
```

求前缀/后缀0的个数

- 以求前缀0为例
- 二分查找

```
inline int countLeadingZeros_1(u32 x) {
    int ans = 0;
    if (x >> 16) x >>= 16; else ans |= 16;
    if (x >>  8) x >>=  8; else ans |=  8;
    if (x >>  4) x >>=  4; else ans |=  4;
    if (x >>  2) x >>=  2; else ans |=  2;
    if (x >>  1) x >>=  1; else ans |=  1;
    ans += !x;
    return ans;
}
```

求前缀/后缀0的个数

- 第一步，判断高16位是否为空
- 若是，则高16位必然均为前缀0，我们给答案加上16，并在第0至15位上继续二分
- 若不是，则前缀0只出现在则高16位上，我们将它们右移到低16位上以便对它继续二分
- 第二步，判断此时的高8位是否均为前缀0，并选择一边继续二分下去。
- ……

求前缀/后缀0的个数

- 同样的思路求解后缀0个数

```
inline int countTrailingZeros_1(u32 x) {
    int ans = 0;
    if (!(x & 65535u)) x >>= 16, ans |= 16;
    if (!(x & 255u  )) x >>=  8, ans |=  8;
    if (!(x & 15u   )) x >>=  4, ans |=  4;
    if (!(x & 3u    )) x >>=  2, ans |=  2;
    if (!(x & 1u    )) x >>=  1, ans |=  1;
    ans += !x;
    return ans;
}
```

求前缀/后缀0的个数

查表

- 依然以前缀0为例
- 先确定第一个1出现在高16位中还是低16位中，再通过查表来得到这个16位整数的答案
- 递推
- $f(0) = 16$
- $f(i) = f(i >> 1) - 1$

求前缀/后缀0的个数

```
int clz_tbl[65537];
void countLeadingZerosPre() {
    clz_tbl[0] = 16;
    for (int i=1; i<65536; ++i)
        clz_tbl[i] = clz_tbl[i >> 1] - 1;
}
inline int countLeadingZeros_2(u32 x) {
    if (x >> 16)
        return clz_tbl[x >> 16]; else
        return clz_tbl[x & 65535] + 16;
}
```

求前缀/后缀0的个数

内建函数

- int __builtin_clz (unsigned int x);
- 求解前缀0
- bsr (Bit Scan Reverse) + xor / lzcnt
- int __builtin_ctz (unsigned int x);
- 求解后缀0
- bsf (Bit Scan Forward)
- 若参数x为0，返回值是未定义

求第k个1的位置

- 转化为求最大的w，使得第0位到第$w-1$中1的个数小于k
- 二分查找
- 利用cnt_tbl

求第k个1的位置

```
int kthBit_1(u32 x, int k) {
    int ans = 0;
    if (cnt_tbl[x & 65535u] < k)
        k -= cnt_tbl[x & 65535u], ans |=16, x >>=16;
    if (cnt_tbl[x & 255u  ] < k)
        k -= cnt_tbl[x & 255u  ], ans |= 8, x >>= 8;
    if (cnt_tbl[x & 15u   ] < k)
        k -= cnt_tbl[x & 15u   ], ans |= 4, x >>= 4;
    if (cnt_tbl[x & 3u    ] < k)
        k -= cnt_tbl[x & 3u    ], ans |= 2, x >>= 2;
    if (cnt_tbl[x & 1u    ] < k)
        k -= cnt_tbl[x & 1u    ], ans |= 1, x >>= 1;
    return ans;
}
```

求第k个1的位置

- 不利用cnt_tbl
- 二分时查询的区间正好是分治时求过的区间
- 利用分治的中间值

求第k个1的位置

```
int kthBit_2(u32 x, int k) {
    int s[5], ans = 0, t;
    s[0] = x;
    s[1] = x - ((x & 0xAAAAAAAAu) >> 1);
    s[2] = ((s[1] & 0xCCCCCCCCu) >> 2) + (s[1] & 0x33333333u);
    s[3] = ((s[2] >>  4) + s[2]) & 0x0F0F0F0Fu;
    s[4] = ((s[3] >>  8) + s[3]) & 0x00FF00FFu;
    t = s[4] & 65535u;
    if (t < k) k -= t, ans |=16, x >>=16;
    t = (s[3] >> ans) & 255u;
    if (t < k) k -= t, ans |= 8, x >>= 8;
    t = (s[2] >> ans) & 15u;
    if (t < k) k -= t, ans |= 4, x >>= 4;
    t = (s[1] >> ans) & 3u;
    if (t < k) k -= t, ans |= 2, x >>= 2;
    t = (s[0] >> ans) & 1u;
    if (t < k) k -= t, ans |= 1, x >>= 1;
    return ans;
}
```

提取末尾连续的1

- 对x加1之后，最右边连续的1会变为0，最右边的0则变为1，而其它位不变。
- $x \ \& \ (x \ \hat{} \ (x \ + \ 1))$

提取lowbit

- 指正整数x在二进制下最右边一个1开始至最低位的那部分，记为$\text{lowbit}(x)$
- $\text{lowbit}(x) = x \,\&\, (x \,\hat{}\, (x - 1))$
- $\text{lowbit}(x) = x \,\hat{}\, (x \,\&\, (x - 1))$
- $x \,\&\, (x - 1)$可用来删除x最右边的1
- $\text{lowbit}(x) = x \,\&\, -x$
- 另类的求后缀0个数方式

遍历所有1

- 不断求lowbit并将它从原数中删去（异或）
- 若需要用到这个1所在的位置
- 转化为求后缀0个数

用Bits表示集合

bitset

用Bits表示集合

- 任意集合
- 映射到整数集合
- 全集为$[0, n)$的整数集合

- 二进制的每个位有0和1两种状态
- 对应集合中某个元素是否存在

用Bits表示集合

- 计算机中单个二进制数的位数是有限的，假设最大位宽为w
- 表示上述n个元素的集合需要$\left\lceil \frac{n}{w} \right\rceil$个二进制数
- 一个二进制数称为一块
- 第i块记录第$w \cdot i$至$w \cdot (i+1) - 1$是否存在

- $w = \theta(\log n)$

用Bits表示集合

- bitset
- 压*w*位的二进制高精度整数
- 可以进行“与”、“或”、非、“异或”、“左移”、“右移”

- `std::bitset`
- `Boost::dynamic_bitset / tr2::dynamic_bitset`

交集

- 与
- 时间复杂度：$O\left(\frac{n}{w}\right)$
- `bitset::operator &`

并集

- 或
- 时间复杂度：$O\left(\frac{n}{w}\right)$
- `bitset::operator |`

补集

- 非
- 时间复杂度：$O\left(\frac{n}{w}\right)$
- `bitset::operator ~`

差集

- x与y的差
- $\{a|a \in x \wedge a \notin y\}$
- 删除x中x与y的交
- $x \text{ ^ } (x \ \& \ y)$
- 时间复杂度：$O\left(\frac{n}{w}\right)$

统计元素个数

- 对每块分别统计，然后相加
- 时间复杂度：$O\left(\frac{n}{w}\right)$

- `bitset::count`

遍历集合元素

- 遍历所有块
- 每块按“遍历所有1”的方法遍历
- 时间复杂度：$O\left(\frac{n}{w}+cnt\right)$

- `bitset::_Find_first \`
  `bitset::_Find_next`

求集合中第k小元素

- 从小到大遍历每个块，找出第k小元素所在块
- 利用“求第k个1的位置”的方法确定具体元素
- 时间复杂度：$O\left(\frac{n}{w}+\log w\right)$

求集合中第k小元素

优化

- 使用辅助线段树维护每个块的元素个数
- 在线段树上二分确定所在块
- 其他的基本集合操作仍能高效进行

求集合中大于x的最小元素

- 从x所在的块开始向后遍历，第一个包含元素的块
- 通过“求后缀0个数”的方法确定具体位置
- 时间复杂度：$O\left(\frac{n}{w}\right)$

- `bitset::_Find_next`

求集合中大于x的最小元素

优化

- 使用辅助线段树
- 使用辅助bitset / w叉线段树
- 使用辅助bitset+额外信息 / vEB Tree

将集合中每个元素加上/减去x

- 左移和右移
- 时间复杂度：$O\left(\frac{n}{w}\right)$

部分元素

- 原集合x，要修改的集合y
- 变化的元素：x与y的交
- 不变的元素：x与y的差
- 修改x与y的交，再并上x与y的差

“卷积”

- x与y的“卷积”
- $r = \{a + b \mid a \in x \wedge b \in y\}$

- 枚举y的每一位
- 若第i位为1则为答案或上$x \ll i$
- 时间复杂度：$O\left(\frac{n}{w}\right)$

“卷积”

优化

- 预处理$x'_0 \dots x'_{2^p-1}$
- x_i表示x与i卷积的结果
- 将y按每p位一段分成$m = \left\lceil \frac{n}{p} \right\rceil$段 $y'_0 \dots y'_m$
- $r = \bigcup_{i=0}^{m-1} x'_{y'_i} \ll ip$
- 取$p = \theta(\log n)$
- 时间复杂度：$O\left(\frac{n}{w \log n}\right)$

“卷积”

寻找“证据”

- $r = \{a + b \mid a \in x \wedge b \in y\}$
- 对于r中的每一个元素，找到至少一组对应的(a, b)

- 算法中每次更新r之前先求差集
- 对于所有在本次更新中新加入r的值，花费$O(p)$的时间寻找并记录其证据

枚举子集

- 对于某个集合的某个子集，求出字典序排在它前一位的集合或后一位的集合
- 集合x和它的一个子集y
- 前一个子集：$(y - 1) \& x$
- 从x开始不断求前一个子集，直到空集

枚举k个元素的子集

- 对于集合x的某个k元素子集y，求下一个
- $l = x \,\&\, -x$
- $y = x + l$
- $ans = y \mid (((x \,\hat{}\, y) / l) >> 2)$

位运算的其他应用

搞笑的

搞笑的

判断奇偶性

- & 1

乘以或除以二的幂

- 左移、右移

对二的幂取模

- $x \bmod 2^y$
- 相当于取x的后y位
- $x \,\&\, ((1 << y) - 1)$

$\log_2 x$的整数部分

- 前缀0个数

交换两个数

- $a \wedge= b$
- $b \wedge= a$
- $a \wedge= b$

绝对值

- $-x = \sim x + 1$
- 32位整数x
- $sign = x >> 31$
- $(x \wedge -sign) + sign$

比较两个数是否相等

- 异或是否为0

比较两个数的大小

- 无符号整数x, y
- 二分求第一个不同的位
- y的这一位为1则$x < y$，否则$x > y$
- 带符号？
- 符号位取反

求两数较小值/较大值

- $\min(x, y)$
- $y \wedge ((x \wedge y) \,\&\, -(x < y))$
- $\max(x, y)$
- $y \wedge ((x \wedge y) \,\&\, -(x > y))$

选择

- $y \wedge ((x \wedge y) \,\&\, -cond)$
- $cond = 1$，结果为x
- $cond = 0$，结果为y

例题

筷子

- $2n+1$个整数
- 某个数出现了奇数次，其他数出现了偶数次
- 求出现奇数次的这个数

筷子

- 将所有数异或即可
- 异或的交换律、结合律
- 从每个二进制位的角度考虑

n皇后问题

- 向走已返大爷致敬！
- $n \times n$的棋盘上放置n个皇后
- 每个皇后能攻击同行、同列、同对角线的格子
- 要求皇后不能互相攻击
- 求方案数

n皇后问题

- 搜索
- 每行只能放一个皇后
- 依次枚举每行放在那里
- 同列、同对角线放过的位置不能放

n皇后问题

```
int ans;
u32 msk;
void dfs(u32 l, u32 m, u32 r) {
    if (m == msk) {
        ++ans;
        return;
    }
    u32 S = msk & ~(l | m | r), p;
    for (; S; S ^= p) {
        p = S & -S;
        dfs((l | p) << 1, m | p, (r | p) >> 1);
    }
}
```

最小斯坦纳树

- n个点m条边的无向图，每条边带权
- 选出一个包含给定的k个点的联通子图
- 最小化子图的边权和

最小斯坦纳树

- 这个子图必定是一棵树
- $f[mask][i]$表示已经包含的点的集合为$mask$，根为i的树的最小权值和
- 考虑转移
- 枚举$mask$的子集sub
- 用$f[sub][i] + f[mask \wedge sub][i]$更新$f[mask][i]$
- 用$f[mask][i]$更新$f[mask][j]$
- 类似Dijkstra，按照$f[mask][i]$从小到大顺序进行更新
- $O(3^k \cdot n + 2^k \cdot n^2)$

抓企鹅

Time Limit: 2 s

- xyz带着他的教徒们乘着科考船一路破冰来到了南极大陆，发现这里有许许多多的企鹅。邪恶的xyz想要抓很多企鹅回去开动物园，当宠物玩
- 有n只企鹅，第i只企鹅在A_i时刻出现在坐标为(B_i, C_i, D_i)的地方
- Q个询问
- 若xyz在T时刻将$(0,0,0)$到(X, Y, Z)这个大长方体里的企鹅都抓走，他抓到了多少企鹅
- $n, Q \leq 30000$

抓企鹅

Time Limit: 2 s

- 四维数点问题
- 分治套分治、树套树……
- 利用bitset
- 每一维分开考虑
- 对于询问i，第j维不超过它的元素构成的集合为$S_{i,j}$
- 企鹅和询问分别排序，离线扫描
- $S_{i,1} \cap S_{i,2} \cap S_{i,3} \cap S_{i,4}$的元素个数即为答案
- $O\left(\frac{nQ}{w}\right)$

Summer Earnings

Time Limit: 9 s

- 平面上n个点
- 选三个点为圆心画三个半径为r的圆
- 三个圆不能相交
- 求r最大值
- $n \leq 3000$

Summer Earnings

Time Limit: 9 s

- 最优解情况下必有两圆相切
- 将所有点对(i,j)按距离（相切时半径）排序
- 从大到小依次在每对点之间连边
- 第一次出现三元环时加入的边对应的半径即为答案
- 对于每个点记录与它相连的点的集合S_i
- 加入边(i,j)时若S_i与S_j有交则表明出现了三元环
- 不是标算
- $O\left(\frac{n^3}{w}\right)$

Quick Tortoise

Time Limit: 3 s

- $n \times m$的网格图上，有一些点是障碍
- q个询问
- 问点(x_1, y_1)是否能只通过向下走和向右走走到(x_2, y_2)
- $q \leq 6 \cdot 10^5$
- $n, m \leq 500$

Quick Tortoise

Time Limit: 3 s

- 考虑所有询问满足$x_1 \le p \le x_2$的情况
- 对于所有在第p行上方的点x, y，求它能走到第p行的哪些点，记这个点集为$f_{x,y}$
- 对于所有在第p行下方的点x, y，求第p行的哪些点能走到它，记这个点集为$g_{x,y}$
- $f_{x,y} = f_{x+1,y} \cup f_{x,y+1}$
- $g_{x,y} = g_{x-1,y} \cup g_{x,y-1}$
- 对于询问(x_1, y_1)，(x_2, y_2)，若$f_{x_1,y_1} \cap g_{x_2,y_2} \neq \emptyset$则表示可以走到

Quick Tortoise

Time Limit: 3 s

- 询问任意的情况
- 分治
- 每次处理过中间的询问
- $O\left(\frac{nm^2}{w}\log n + q\log n\right)$或
- $O\left(\frac{(nm)^{1.5}}{w} + q\log nm\right)$

Robot in Basement

Time Limit: 4 s

- $n \times m$的网格图上，有一些点是障碍，且地图边界均为障碍
- 机器人可以根据程序在网格图上行走
- 程序是一个由UDLR四个指令组成的字符串
- 机器人依次执行每个指令，一个指令会使机器人向指定的方向移动一格，如果对应格子为障碍则不动

Robot in Basement

Time Limit: 4 s

- 网格图中有一个格子称为出口
- 给定一个长度为l的程序
- 求最短的前缀，使得对于一开始在网格图上任意非障碍位置的机器人，在执行完这个前缀之后都停在出口上
- $l \leq 10^5$
- $n, m \leq 150$

Robot in Basement

Time Limit: 4 s

- 模拟
- 在每个不是障碍的格子上都放一个机器人
- 对他们一起执行程序
- bitset保存哪些位置有机器人
- 上下左右走转化为左移、右移
- 碰到障碍的机器人需要删除并放回原处
- $O\left(\frac{nml}{w}\right)$

Bags and Coins

Time Limit: 2.5 s

- 有s个硬币，n个包
- 一个包可以放在其他包里面，可以多层嵌套
- 第i个包里总共有a_i个硬币（如果拿出某个硬币必须要打开第i个包我们就说这个硬币在第i个包里）
- 构造一种满足上述条件的方案
- $1 \leq n, s, a_i \leq 70000$

Bags and Coins

Time Limit: 2.5 s

- 问题转化
- 选一些a_i，使得他们的和等于s，求方案
- 背包问题
- 前$i-1$个物品时可以达到的总和的集合为S，加入第i个物品之后变为$S \mid (S \ll a_i)$
- 如何记录方案？
- 若$x \in S \mid (S \ll a_i)$但$x \notin S$，则记$from[x] = i$
- 根据$from$重构方案
- $O\left(\frac{ns}{w}\right)$

Inna and Binary Logic

Time Limit: 3 s

- n个数$a_0[1] \ldots a_0[n]$
- 进行$n-1$轮操作
- 第i轮操作会产生一个长度为$n-i$的数组a_i，其中$a_i[j] = a_{i-1}[j] \wedge a_{i-1}[j+1]$
- 这个过程中一共会产生$\frac{n(n-1)}{2}$个数
- 这些数的和叫做数组a_0的特征值

Inna and Binary Logic

Time Limit: 3 s

- Q个修改
- 一个修改x, y将$a_0[x]$改为y
- 每次修改之后你需要给出新的a_0的特征值
- $0 \leq n, Q, a_0[i] \leq 10^5$

Inna and Binary Logic

Time Limit: 3 s

- $\frac{n(n-1)}{2}$个数中每一个对应a_0中的一个区间
- 与运算每一位贡献独立
- 对于第i位，一段长度为l的连续的1对答案的贡献为$2^i \cdot \frac{l(l-1)}{2}$
- 利用bitset维护
- $O(Q \log W \log_w n)$，W为权值范围

三分图可达性问题

- 给出一个三分图
- 顶点可分为A, B, C三部 每部包含n个顶点
- 已知A与B之间、 B与C之间的边、 A与C间无边
- 求A, C两部之间顶点对的可达性
- 对于可达的顶点对，需要给出B中哪个顶点使它们可达

三分图可达性问题

- 可达性可由矩阵乘法刻画
- 考虑“卷积”及其证据求法
- 可用类似方法求解矩阵乘法及相应证据

Friends

Time Limit: 6 s

- n个数$a_1 \ldots a_n$
- 从中选出m对数$(a_i, a_j), i \neq j$
- 每一对只能选一次
- 每一对的权值为$a_i \wedge a_j$
- 最大化权值和
- $n \leq 5 \cdot 10^4, m \leq \frac{n(n-1)}{2}$

Friends

Time Limit: 6 s

- 考虑异或的前i位确定，其它位任意的方案数
- 前i位确定，其它位任意的所有方案的权值和
- 枚举一个数的前i位，另一个数的前i位确定
- 两个集合S_1、S_2

方案数

- $size(S_1) \cdot size(S_2)$

权值和

- 逐位计算

Friends

Time Limit: 6 s

- 设计入答案的最小的异或值为lim（权值第m大）
- 逐位确定lim，同时求异或值大于lim的方案权值和
- 异或值等于lim的方案单独考虑
- $O(n \log^2 W)$，W为权值范围

Mutation

Time Limit: 2 s

- 一个字符串S，长度为n，字符集为前k个大写字母
- 对于字母a, b，ab相邻会产生权值$w_{a,b}$
- S的权值定义为所有相邻字母对产生的权值的和
- 现在可以对串进行一些修改
- 一次修改是指，将串中的某个字母全部删除
- 删除字母i会使修改后的串的权值增加k_i
- 问经过任意次数的修改之后，可以得到多少不同、权值不超过T的串（不计空串）

Mutation

Time Limit: 2 s

- $f[mask]$表示删除字母集合$mask$后得到的串的权值
- $f[mask]$ = 删除$mask$所获权值 + 相邻字母贡献权值
- 考虑相邻字母贡献的权值$g[mask]$
- 考虑原串中的某一位置i以及i之后某个字母第一次出现的位置j，设i位置与j位置之间（不包含）出现过的字母集合为set
- 它将对$g[mask]\left(set \subseteq mask, S_i \notin mask, S_j \notin mask\right)$产生$w_{S_i,S_j}$的贡献

Mutation

Time Limit: 2 s

- 将$mask$看作k维数组
- (i,j)的贡献可以看作对某个k维区间整体加上w_{S_i,S_j}
- k维差分
- 区间加转化为：
- $g[set] \mathrel{+}= w_{S_i,S_j}$
- $g[set \cup S_i] \mathrel{-}= w_{S_i,S_j}$
- $g[set \cup S_j] \mathrel{-}= w_{S_i,S_j}$
- $g[set \cup S_i \cup S_j] \mathrel{+}= w_{S_i,S_j}$

Mutation

Time Limit: 2 s

- k维前缀和（子集和）
- 一维一维求
- $O(2^k \cdot k + nk)$

祝大家学习顺利

The End