2. 数据的机器级表示与处理

2.1. 整数和字符的表示

2.2. 数据的转换和运算

2.3. 浮点数表示和运算

无符号整数和有符号整数的转换

- C语言显式的强制类型转换

```
short v = -1;
unsigned short uv = (unsigned short) v;
printf("v=%d, uv=%u\n", v , uv);
```

请问上述代码的输出结果是什么?

v=-1, uv=65535

无符号整数和有符号整数的转换

- C语言显式的强制类型转换

```
short v = -1;
unsigned short uv = (unsigned short) v;
printf("v=%d, uv=%u\n", v , uv);
```

输出：v=-1, uv=65535

- (unsigned) short长16位
- 有符号整数-1的16位补码为

  1111 1111 1111 1111
- 作为无符号整数解释时

  1111 1111 1111 1111

  $=2^{16}-1=65535$

无符号整数和有符号整数的转换

- C语言显式的强制类型转换

```
unsigned ui = 4294967295;
int i = (int) ui;
printf("ui=%u, i=%d\n" , ui, i);
```

写出上述代码的输出，已知
4294967295=0xffffffff

ui=4294967295, i=-1

- (unsigned) int长32位
- 无符号整数（4294967295）的32位机器数为

  FFFF FFFF（32位1）
- 作为有符号整数解释时

  FFFF FFFF [X]补码

  0000 0001 |X|=2^n-[X]补码

  -1 X

无符号整数和有符号整数的转换

- C语言隐式类型转换

```
int i;
unsigned ui = 0xffffffff;
i = ui;  i = (int) ui;
printf("i=%d , ui=%u\n", i, ui);
```

写出上述代码的输出

i=-1, ui=4294967295

- (unsigned) int长32位
- 无符号整数的32位机器数为

  FFFF FFFF（32位1）
- 作为有符号整数解释时

  FFFF FFFF [X]补码

  0000 0001 |X|=2^n-[X]补码

  -1 X

无符号整数和有符号整数的转换

- C语言隐式类型转换（格式化输出）

```
int i = -1;
unsigned ui = 2147483648;
printf("i=%u=%d\n", i, i);
printf("ui=%u=%d\n", ui, ui);
```

写出上述代码的输出，

已知2147483648=0x80000000

```
i=4294967295=-1
ui=2147483648=-2147483648
```

- (unsigned) int长32位
- 有符号整数-1的32位补码为

  FFFF FFFF（32位1）

  解释为无符号整数=2^{32}-1
- 无符号整数2147483648的32位机器数为

  8000 0000（10…0，2^{31}）

  解释为有符号整数时

  0x8000 0000 [X]补码

  0x8000 0000 |X|=2^n-[X]补码

  -2^{31} X

无符号整数和有符号整数的转换

- 处理同样字长的有符号整数和无符号整数之间相互转换的一般规则
  - 保持内存中存储的位值不变
  - 改变解释这些位的方式
  - 数值可能会改变

练习

1. 字长为8，有符号数X的真值为-1，求将其转换为无符号数时的真值。
2. 字长为8，有符号数X的真值为-128，求将其转换为无符号数时的真值。

解1:

X = -1

|X| = 0000 0001

[X]补 = 2^8-|X|

= 1 0000 0000 – 0000 0001

= 1111 1111

无符号数 1111 1111 = 2^8 – 1 = 255

解2:

X = -128

|X| = 1000 0000

[X]补 = 2^8-|X|

= 1 0000 0000 – 1000 0000

= 1000 0000

无符号数 1000 0000 = 2^7= 128

练习

3. 字长为8，无符号数X的真值为255，求将其转换为有符号数时的真值。

解：

无符号数 X = 255 = 1111 1111

有符号数X的内存表示 [X]补 = 1111 1111

|X| = 2^8 - [X]补

= 1 0000 0000 – 1111 1111

= 0000 0001 = 1

X = -1

无符号整数和有符号整数的转换

- 有符号数真值T 转换为 无符号数真值U
  - w=8
    - -1 → 255，-128 → 128

对满足 $TMin_w \leqslant x \leqslant TMax_w$ 的 x 有：

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geqslant 0 \end{cases}$$

- $T2U_8(-1) = -1 + 2^8 = 255$
- $T2U_8(-128) = -1 + 2^8 = 128$

有符号整数的真值范围：

$TMin_w = -2^{w-1}$

$TMax_w = 2^{w-1} - 1$

无符号整数和有符号整数的转换

- 无符号数真值U 转换为 有符号数真值T
  - w=8
    - 255 → -1

对满足 $0 \leqslant u \leqslant UMax_w$ 的 u 有：

$$U2T_w(u) = \begin{cases} u, & u \leqslant TMax_w \\ u - 2^w, & u > TMax_w \end{cases}$$

  - $U2T_8(255)=255-2^8=-1$

有符号整数的真值范围：

$TMin_w=-2^{w-1}$

$TMax_w=2^{w-1}-1$

无符号整数的真值范围：

$UMax_w=2^w-1$

编译器处理常量时的默认类型

- C语言中大部分数字常量默认是有符号数
  - -1 < 0
- 数字常量加上u或U后缀表示无符号数
  - -1 < 0u
- 类型根据常量的值判断

范围	类型
-2^{31}~$2^{31}-1$	int
-2^{63}~$-2^{31}-1$，2^{31}~$2^{63}-1$	long
2^{63}~$2^{64}-1$	unsigned long

编译器处理常量时的默认类型

- 如果运算中同时有无符号整数和有符号整数，则按无符号整数运算

关系表达式	运算类型	结果	说明
0 == 0U			
-1 < 0			
-1 < 0U			
2147483647 > -2147483647-1			
2147483647U > -2147483647-1			
2147483647 > (int) 2147483648U			
-1 > -2			
(unsigned) -1 > -2			

编译器处理常量时的默认类型

- 如果运算中同时有无符号整数和有符号整数，则按无符号整数运算

关系表达式	运算类型	结果	说明
0 == 0U	无符号	1	0…0 (0) == 0…0 (0)
-1 < 0	有符号	1	1…1 (-1) < 0…0 (0)
-1 < 0U	无	0	1…1 (2^{32}-1) > 0…0 (0)
2147483647 > -2147483648	有	1	01…1 (2^{31}-1) > 10…0 (-2^{31})
2147483647U > -2147483648	无	0	01…1 (2^{31}-1) < 10…0 (2^{31})
2147483647 < (int) 2147483648U			
-1 > -2			
(unsigned) -1 > -2			

编译器处理常量时的默认类型

- 如果运算中同时有无符号整数和有符号整数，则按无符号整数运算

关系表达式	运算类型	结果	说明
0 == 0U	无符号	1	0…0 (0) == 0…0 (0)
-1 < 0	有符号	1	1…1 (-1) < 0…0 (0)
-1 < 0U	无	0	1…1 (2^{32}-1) > 0…0 (0)
2147483647 > -2147483648	有	1	01…1 (2^{31}-1) > 10…0 (-2^{31})
2147483647U > -2147483648	无	0	01…1 (2^{31}-1) < 10…0 (2^{31})
2147483647 < (int) 2147483648U	有	0	01…1 (2^{31}-1) > 10…0 (-2^{31})
-1 > -2			
(unsigned) -1 > -2			

```
unsigned int ui = 2147483648;
int i = (int) ui;
```

编译器处理常量时的默认类型

- 如果运算中同时有无符号整数和有符号整数，则按无符号整数运算

关系表达式	运算类型	结果	说明
0 == 0U	无符号	1	0…0 (0) == 0…0 (0)
-1 < 0	有符号	1	1…1 (-1) < 0…0 (0)
-1 < 0U	无	0	1…1 (2^{32}-1) > 0…0 (0)
2147483647 > -2147483648	有	1	01…1 (2^{31}-1) > 10…0 (-2^{31})
2147483647U > -2147483648	无	0	01…1 (2^{31}-1) < 10…0 (2^{31})
2147483647 < (int) 2147483648U	有	0	01…1 (2^{31}-1) > 10…0 (-2^{31})
-1 > -2	有	1	11…1 (-1) > 11…10 (-2)
(unsigned) -1 > -2	无	1	11…1 (2^{32}-1) > 11…10 (2^{32}-2)

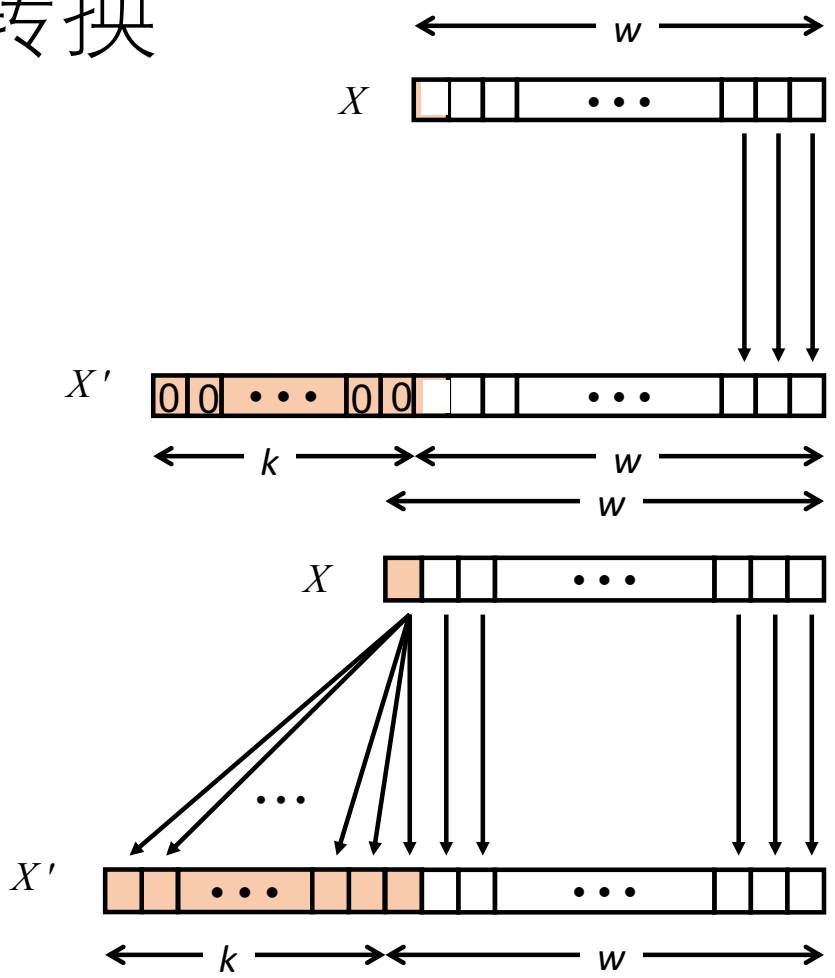
编译器处理常量时的默认类型

- 如果运算中同时有无符号整数和有符号整数，则按无符号整数运算

关系表达式	运算类型	结果	说明
0 == 0U	无符号	1	0…0 (0) == 0…0 (0)
-1 < 0	有符号	1	1…1 (-1) < 0…0 (0)
-1 < 0U	无	0	1…1 (2^{32}-1) > 0…0 (0)
2147483647 > -2147483648	有	1	01…1 (2^{31}-1) > 10…0 (-2^{31})
2147483647U > -2147483648	无	0	01…1 (2^{31}-1) < 10…0 (2^{31})
2147483647 < (int) 2147483648U	有	0	01…1 (2^{31}-1) > 10…0 (-2^{31})
-1 > -2	有	1	11…1 (-1) > 11…10 (-2)
(unsigned) -1 > -2	无	1	11…1 (2^{32}-1) > 11…10 (2^{32}-2)

不同字长的整数之间的转换

- 位扩展（短转长）
  - 零扩展（无符号数）
    - 扩展的高位补0
  - 符号扩展（有符号数）
    - 扩展的高位补原本的符号位



不同字长的整数之间的转换

- 位截断（长转短）
  - 直接将高位丢弃
    - 长n位无符号数X → 短w位无符号数
      - 长n位无符号数 真值表示范围：$0 \sim 2^{n-1}$
      - 短w位无符号数 真值表示范围：$0 \sim 2^{w-1}$
      - 如果$X > 2^{w-1}$，溢出
    - 将一个无符号数截断为w位等价于原值模2^w

不同字长的整数之间的转换

- 位扩展
  1. 写出以下程序中si、usi、i、ui的十进制和十六进制值。

```
short si = -32768;
unsigned short usi = si;
int i = si;
unsigned ui = usi;
```

解:
$32768=2^{15}$
[si] short长16位，有符号，补码 1000 0000 0000 0000
si=-32768，0x8000
[usi] 16位，无符号，10…0解释为无符号整数
usi=32768，0x8000
[i] 32位，有符号，符号扩展，1…1 1000…
i=-32768，0xffff8000
[ui] 32位，无符号，零扩展，0…0 1000…
ui=32768，0x00008000

不同字长的整数之间的转换

- 位扩展

```
short x = 15213;
int ix = (int) x;
short y = -15213;
int iy = (int) y;
```

	十进制真值	十六进制机器数	二进制机器数
x	15213	**填空1（1分）**	0011 1011 0110 1101
ix	**填空4（1分）**	**填空3（1分）**	**填空2（2分）**
y	-15213	**填空6（1分）**	**填空5（2分）**
iy	**填空9（1分）**	**填空8（1分）**	**填空7（2分）**

不同字长的整数之间的转换

```
short x = 15213;
int ix = (int) x;
short y = -15213;
int iy = (int) y;
```

- 位扩展

	十进制真值	十六进制机器数	二进制机器数
x	15213	**填空1（1分）** 3B6D	0011 1011 0110 1101
ix	**填空4（1分）** 15213	**填空3（1分）** 0000 3B6D	**填空2（2分）** 0000 0000 0000 0000 0011 1011 0110 1101
y	-15213	**填空6（1分）** C493	**填空5（2分）** 1100 0100 1001 0011
iy	**填空9（1分）** -15213	**填空8（1分）** FFFF C493	**填空7（2分）** 1111 1111 1111 1111 1100 0100 1001 0011

零扩展和符号扩展不会改变真值。

不同字长的整数之间的转换

- 位扩展+有符号无符号转换

```
short sx = -1;
unsigned int uy = sx;
printf("uy=%u\t%x\n", uy, uy);
```

输出：uy=4294967295　ffffffff

解：

sx，16位，有符号，补码：1111 1111 1111 1111

uy，32位，无符号

先位扩展，后有符号无符号转换

- 位扩展：1…1（32位1）=ffff ffff
- 有符号无符号转换：解释为无符号数=2^{32}-1=4,294,967,295

不同字长的整数之间的转换

- 位截断

```
int i = 32768;
short si = (short) i;
int j = si;
```

上面代码中的i和j是否相等?

位截断时发生了溢出

16位有符号数的表示范围:

-2^{15}~2^{15}-1 (32767<32768)

解:

32768=2^{15}

- i，32位，有符号，补码:

  0000 0000 0000 0000 1000 0000 0000 0000

- si，16位，有符号

位截断，补码：1000 0000 0000 0000 -32768

- j，32位，有符号

符号扩展，补码:

1111 1111 1111 1111 1000 0000 0000 0000

对应的真值=-32768

C语言标准：未定义行为（Undefined Behavior, UB），编译器可以不报错

2. 数据的机器级表示与处理

2.1. 整数和字符的表示

2.2. 数据的转换和运算

2.3. 浮点数表示和运算

数据的运算

- 高级语言程序中涉及的运算（以C语言为例）
  - （按位）布尔运算
  - 逻辑运算
  - 移位运算
  - 整数的加减运算
  - 整数的乘除运算
  - 变量与常数之间的乘除运算

C语言中的运算
↓
表达式
运算指令
运算电路

计算机如何实现高级语言程序中的运算

- C赋值语句y=(x>>2)+k，如何在计算机中实现？

1. 表达式编译为指令序列

```
sarw $2, %ax          ; x>>2
addw %bx, %ax         ; (x>>2) + k
```

2. 执行指令完成运算

控制器对指令进行译码，产生控制信号送运算电路

3. 运算类指令由相应的运算电路完成

```
sarw $2, %ax          将操作数“2”和“R[ax]”送移位器运算
addw %bx, %ax         将R[ax]和R[bx]送整数加减器中运算
```

4. 运算电路由基本的逻辑门电路实现

数据的运算

- 高级语言中的运算表达式（以C语言为例）
  - 整数算术运算、浮点数算术运算
  - 按位、逻辑、移位、位扩展和位截断等运算
- 指令集中的运算指令
  - 定点数运算
    - 算术运算
      - 有符号整数：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
      - 无符号整数：零扩展 / 加 / 减 / 乘 / 除 / 逻辑移位
    - 按位运算：按位与 / 按位或 / 按位非 / …
    - 逻辑运算：与 / 或 / 非 / …
  - 浮点数运算：加、减、乘、除
- 计算机组成中的运算电路
  - 基本运算部件ALU、通用寄存器组、…

布尔运算

- 二进制值1和0表示逻辑值TRUE和FALSE
  - 与，And，A&B

&	0	1
0	0	0
1	0	1

  - 非，Not，~A

~	
0	1
1	0

  - 或，Or，A|B

|	0	1
0	0	1
1	1	1

  - 异或，Xor，A^B

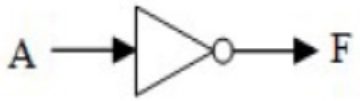
^	0	1
0	0	1
1	1	0

一位逻辑门电路

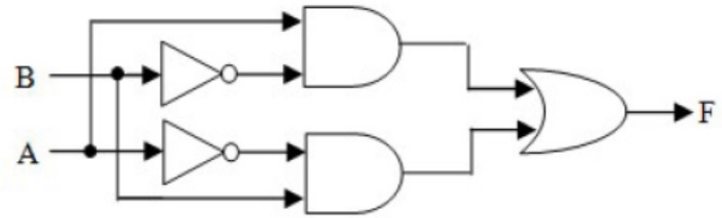
- 通过逻辑门电路实现逻辑运算
  - 三种基本门电路：与门、或门、非门
  - 其他门电路（如异或门）可以通过三种基本门电路组合形成

F=A&B

F=A|B



F=~A



F=A^B=(~A&B) | (A&~B)

按位布尔运算

- 长度为n、由0和1组成的位向量之间的布尔运算
  - 若$A=A_{n-1}A_{n-2}\cdots A_1A_0$，$B=B_{n-1}B_{n-2}\cdots B_1B_0$，则布尔运算F=A op B实际上是按位op，即$F_i=A_i$ op B_i（$0\leqslant i\leqslant n-1$）

```
  01101001          01101001          01101001
& 01010101        | 01010101        ^ 01010101        ~ 01010101
----------        ----------        ----------        ----------
  01000001          01111101          00111100          10101010
```

n位逻辑门电路

- n位逻辑运算
  - 重复使用n个相同的门电路

F=A&B

F=A|B

F=~A

F=A^B

C语言中的布尔运算

- （按位）布尔运算
  - 按位或 |、按位与 &、按位取反 ~、按位异或 ^
  - 用途：对位串实现“掩码”（mask）等操作
    - 多媒体数据处理
    - 状态/控制信息处理

C表达式	二进制表达式	二进制结果	十六进制结果
~0x41	~[0100 0001]	[1011 1110]	0xBE
~0x00	~[0000 0000]	[1111 1111]	0xFF
0x69 & 0x55	[0110 1001] & [0101 0101]	[0100 0001]	0x41
0x69 | 0x55	[0110 1001] | [0101 0101]	[0111 1101]	0x7D

C语言中的布尔运算

- short y = 0x2C0B;

1. 按位取反

~y=~0x2C0B=~[0010 1100 0000 1011]=1101 0011 1111 0100=0xD3F4

2. 使高字节不变、低字节变为0

y&0xFF00=0x2C00

3. 使高字节不变、低字节变为FF

y|0x00FF=0x2CFF

4. 使高字节不变、低字节按位取反

y^0x00FF=[0010 1100 0000 1011]^[0000 0000 1111 1111]

=0010 1100 1111 0100=0x2CF4

C语言中的布尔运算

- short y = 0x2C0B;

1. 按位取反

~y=~0x2C0B=~[0010 1100 0000 1011]=1101 0011 1111 0100=0xD3F4

2. 使高字节不变、低字节变为0

y&0xFF00=0x2C00

3. 使高字节不变、低字节变为FF

y|0x00FF=0x2CFF

4. 使高字节不变、低字节按位取反

y^0x00FF=[0010 1100 0000 1011]^[0000 0000 1111 1111]

=0010 1100 1111 0100=0x2CF4

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

练习

已知异或运算满足交换率、结合律、自反性。解释以下函数的功能。

```
void f(int *x, int *y)
{
  *y = *x ^ *y;
  *x = *x ^ *y;
  *y = *x ^ *y;
}
```

交换律

A^B=B^A

结合律

A^(B^C)=(A^B)^C

自反性

A^A=0

练习

已知异或运算满足交换率、结合律、自反性。解释以下函数的功能。

```
void f(int *x, int *y)
{
   *y = *x ^ *y;
   *x = *x ^ *y;
   *y = *x ^ *y;
}
```

f：不使用临时变量交换两个值

设*x=A，*y=B

1. `*y = *x ^ *y;`

*x=A，*y=A^B

2. `*x = *x ^ *y;`

*x=A^(A^B)=B，*y=A^B

3. `*y = *x ^ *y;`

*x=B，*y=B^(A^B)=A

C语言中的逻辑运算

- 操作
  - 或，||
  - 与，&&
  - 非，!
- 逻辑运算 VS 布尔运算
  - 运算过程：整体 VS 按位
  - 运算结果：逻辑值 VS 位串
- 用途
  - 关系表达式
    - `if ((x>y) && (i<100)) then …`

C语言中的逻辑运算

- 运算过程：整体
  - 所有非零的参数都表示TRUE，而参数0表示FALSE
- 运算结果：逻辑值
  - 返回1或者0，分别表示结果为TRUE或者为FALSE

表达式	结果
!0x41	0x00
!0x00	0x01
!!0x41	0x01
0x69&&0x55	0x01
0x69||0x55	0x01

移位运算

- 左移位，x<<y
  - 高位移出，低位补0
- 右移位，x>>y
  - 逻辑右移：低位移出，高位补0
  - 算术右移：低位移出，高位补符号位

x	01100010
<<3	~~011~~00010000
Log>>2	00011000~~10~~
Arith>>2	00011000~~10~~

x	10100010
<<3	~~101~~00010000
Log>>2	00101000~~10~~
Arith>>2	11101000~~10~~

C语言中的移位运算

- 用途
  - 提取部分信息
  - 数值扩大（左移）或缩小（右移）2、4、8…倍
    - 右移可能损失精度
- 操作
  - 左移 <<，右移 >>
  - C语言不区分逻辑移位和算术移位，编译器根据操作数的类型确定
    - 无符号数：左移、逻辑右移
    - 有符号数：左移、算术右移
  - 溢出的情况
    - 无符号数左移时移出了1
    - 有符号数左移后符号改变

练习

=149D

- 无符号数10010101右移一位的结果。

解:

无符号数，逻辑右移

010010101

=74D

练习

=-107D

- 有符号数10010101右移一位的结果。

解:

有符号数，算术右移

1 10010101

=-54D

练习

对于一个n（n≥8）位的变量x，请根据C语言中按位运算的定义，写出满足下列要求的C语言表达式。

1. x的最低字节不变，其余各位全变为0。
2. x的最低字节全变1，其余各位不变。
3. x的最高字节不变，其余各位全变为0。
4. x的最低字节全变为0，其余各位取反。

解：$x=x_{n-1}\cdots x_{n-8}\cdots x_7\cdots x_0$

1. x&0xFF
2. x | 0xFF
3. (x >> (n-8)) << (n-8)
4. ((x ^ ~0xFF) >> 8) << 8　　x ^ ~0xFF = x ^ 1…100000000　低字节不变，其余位取反

C语言程序中加减运算

```
int x = 9, y = -6, z1, z2;
z1 = x + y;
z2 = x - y;
```

- 上述程序段中，x和y的机器数是什么？z1和z2的机器数是什么？

x：[x]补，y：[y]补，z1：[x+y]补，z2：[x-y]补

需要一个运算电路，实现已知[x]补 、[y]补，计算[x+y]补、[x-y]补。

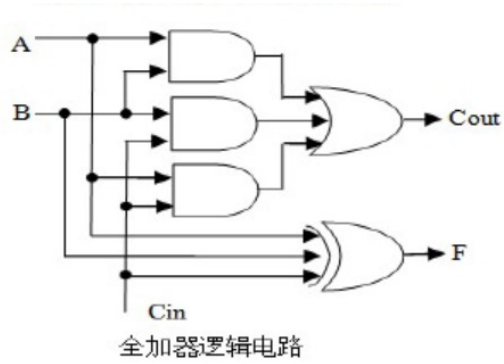
一位加法器

- 全加器：一位带进位的加法器
  - 两个加数为A和B
  - 低位进位为Cin
  - 和为F
  - 向高位的进位为Cout

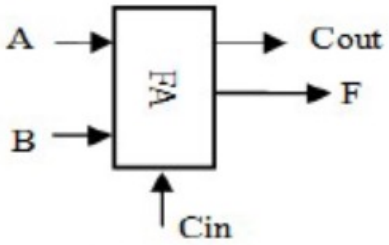
  - F=A^B^Cin
  - Cout=A&B | A&Cin | B&Cin

A	B	Cin	F	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器真值表



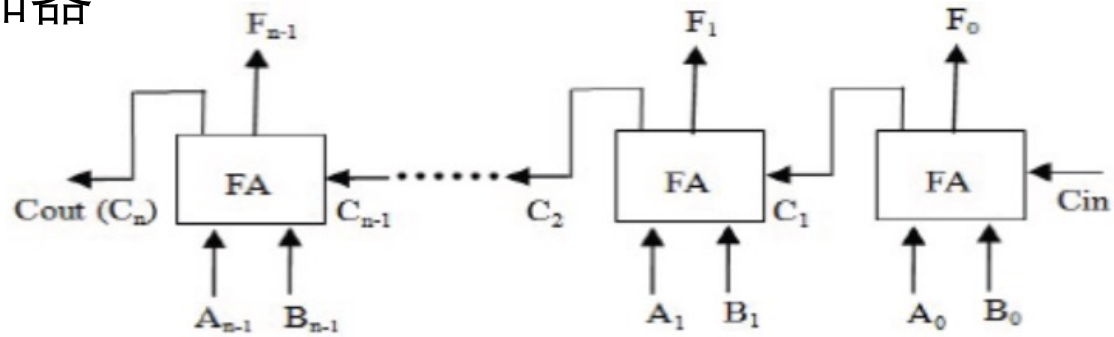
全加器逻辑电路



全加器符号

n位加法器

- n个全加器



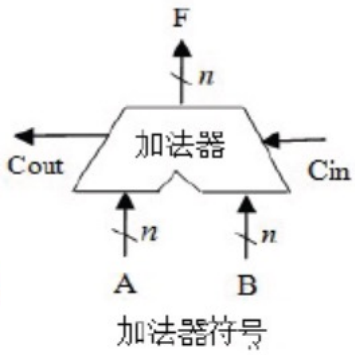
A	B	Cin	F	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器真值表

例：n=4，A=1001，B=1100

F=?，Cout=?

F=0101，Cout=1

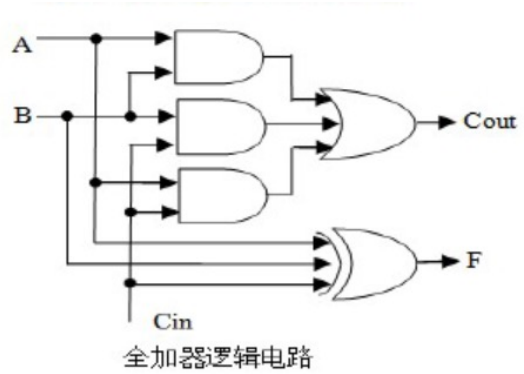


加法器符号

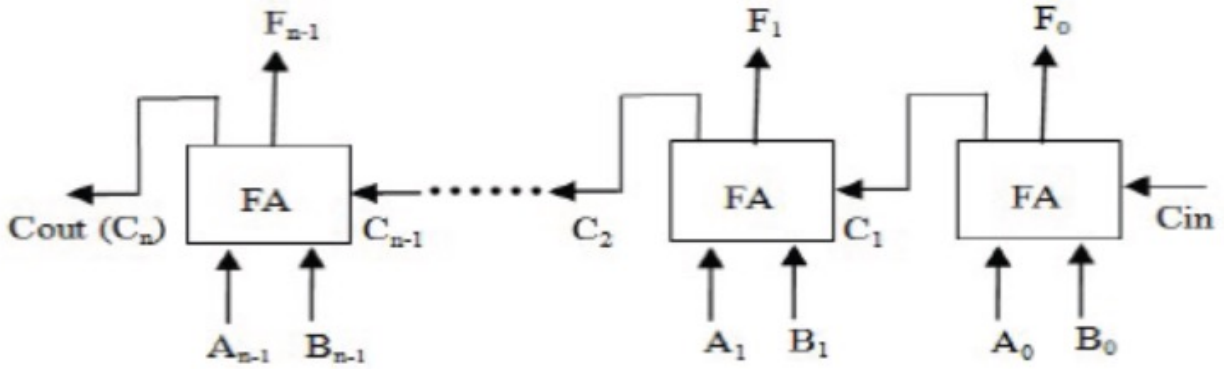
n位加法器

- F=A^B^Cin
- Cout=A&B | A&Cin | B&Cin

- 加法由逻辑部件实现



全加器逻辑电路

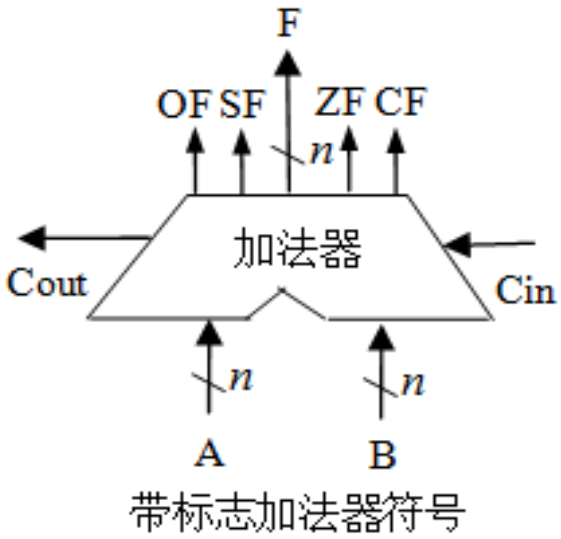


n位带标志加法器

- 不同标志的作用
  - 在分支指令中判断转移执行

  `if (…) {…}`
  - 在加减运算中判断溢出

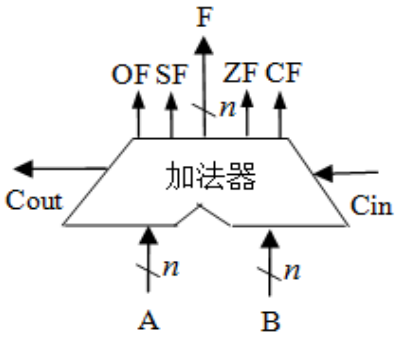
  short x = a + b;



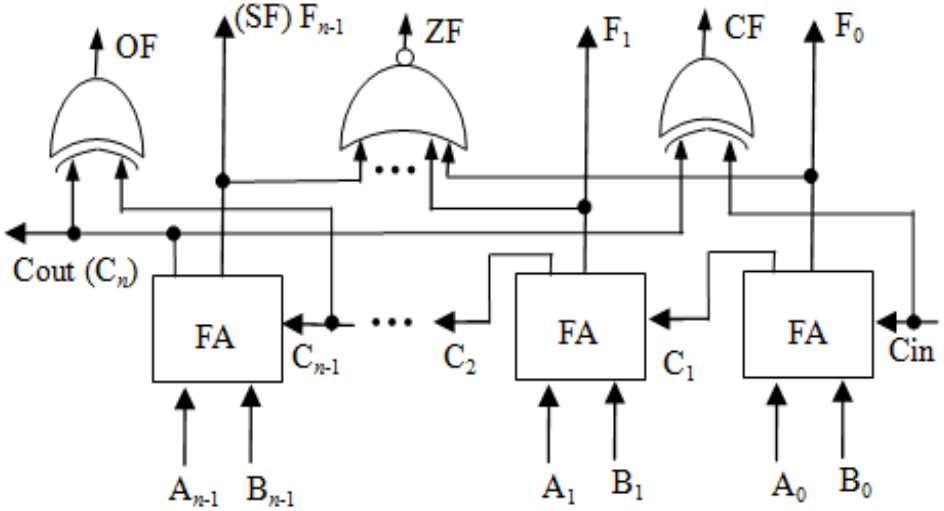
带标志加法器符号

n位带标志加法器

- 输入：加数A、B，低n位进位Cin
- 输出
  - n位和F=Fn-1…F0
    - Fi=Ai^Bi^Ci (C0=Cin)
  - 向高n位进位Cout
    - Ci+1=Ai&Bi | Ai&Ci | Bi&Ci
    - Cout=Cn
  - OF（溢出标志）
    - OF=Cn^Cn-1
  - SF（符号标志）
    - SF=Fn-1
  - ZF（零标志）
    - ZF= ~ (Fn-1 | … | F0)
  - CF（进位标志）
    - CF=Cout^Cin



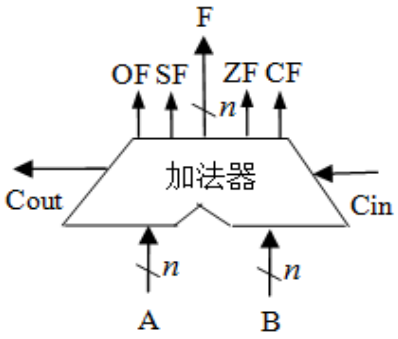
带标志加法器符号



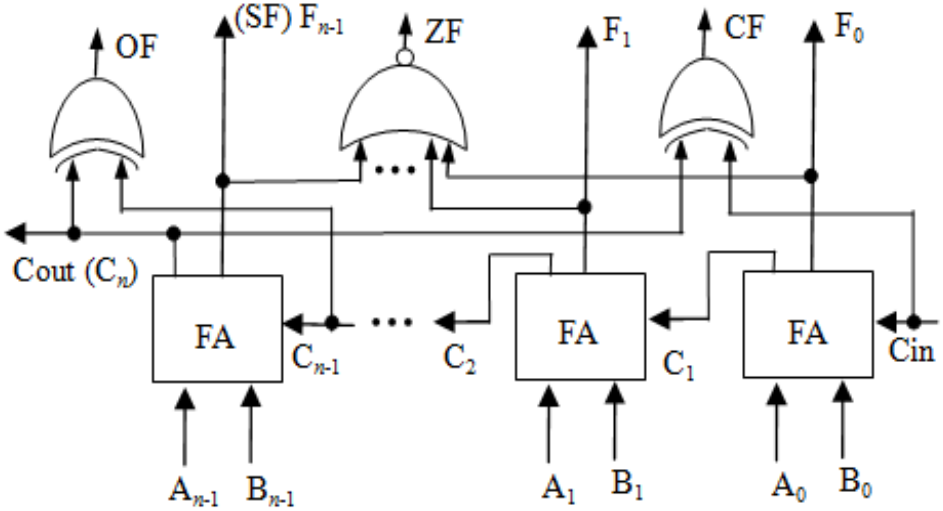
带标志加法器的逻辑电路

n位带标志加法器

- n=4, A=1101, B=1011, Cin=0
- F=1000
  - F0=1^1^0=0, …
- Cout=1
  - C1=1, …
- OF=0
  - A=-3, B=-5
- SF=1
- ZF=0
- CF=1



带标志加法器符号



带标志加法器的逻辑电路

条件标志

- 条件标志（Flag）
  - 溢出标志OF
  - 符号标志SF
  - 零标志ZF
  - 进位标志CF

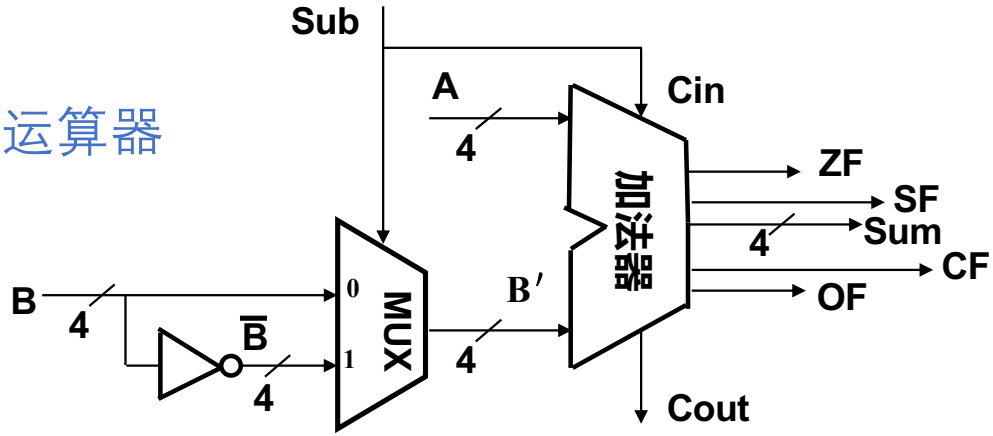
  - 由运算电路产生，记录到专用的寄存器中
    - 程序/状态字寄存器、标志寄存器
      - 每个标志对应寄存器中的一个标志位
        - x86-64：FLAGS寄存器

n位整数加/减运算器

- 补码加减运算公式
  - [A+B]补 = [A]补 + [B]补 (mod 2^n)
  - [A–B]补 = [A]补 + [–B]补 (mod 2^n)
    - [–B]补=$\overline{[B]补}$+1（Cin）

- 带标志加法器 + MUX → 整数加减运算器
  - 无符号整数加、无符号整数减
  - 有符号整数加、有符号整数减

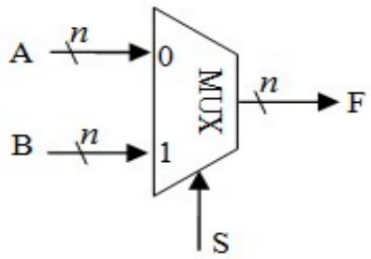
  - Cin=Sub=1，减法
  - Cin=Sub=0，加法



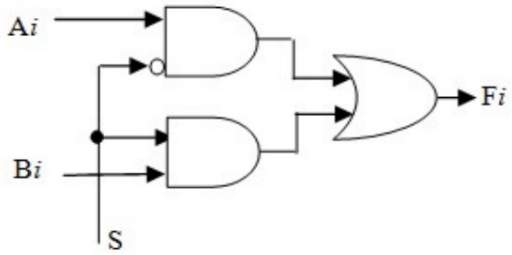
整数加/减运算部件

多路选择器（MUX）

- 二路选择器
  - 两个输入端：A、B
  - 控制端（一位）：S
  - 一个输出端：F
  - 当S=0时，F=A；当S=1时，F=B。
- k路选择器
  - k个输入端
  - 控制端S，位数：$\lceil \log_2 k \rceil$
    - k=3、4，位数=2
    - k=5-8，位数=3



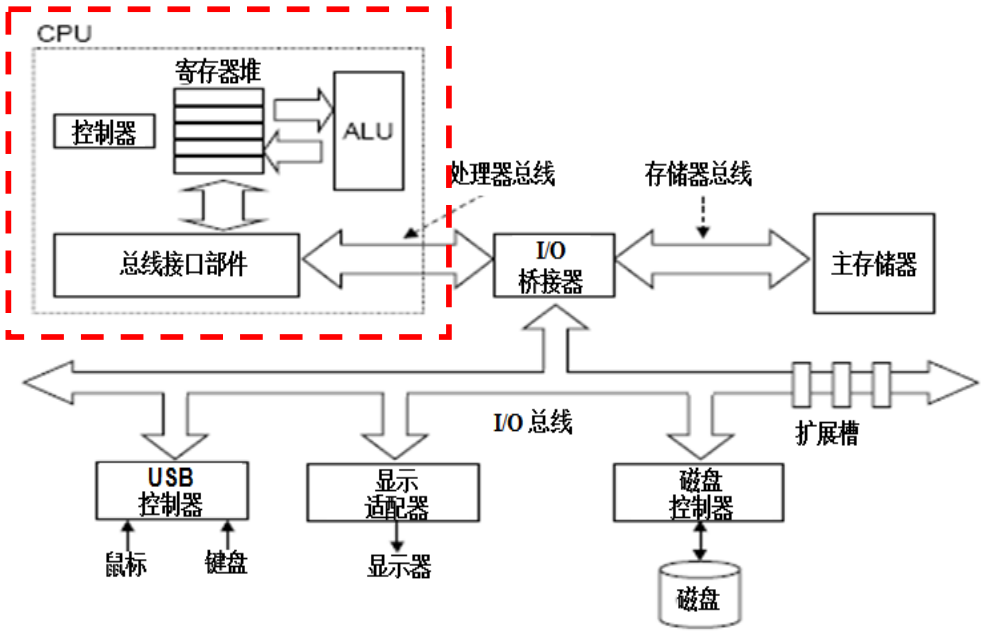
二路选择器符号



一位二路选择器逻辑电路

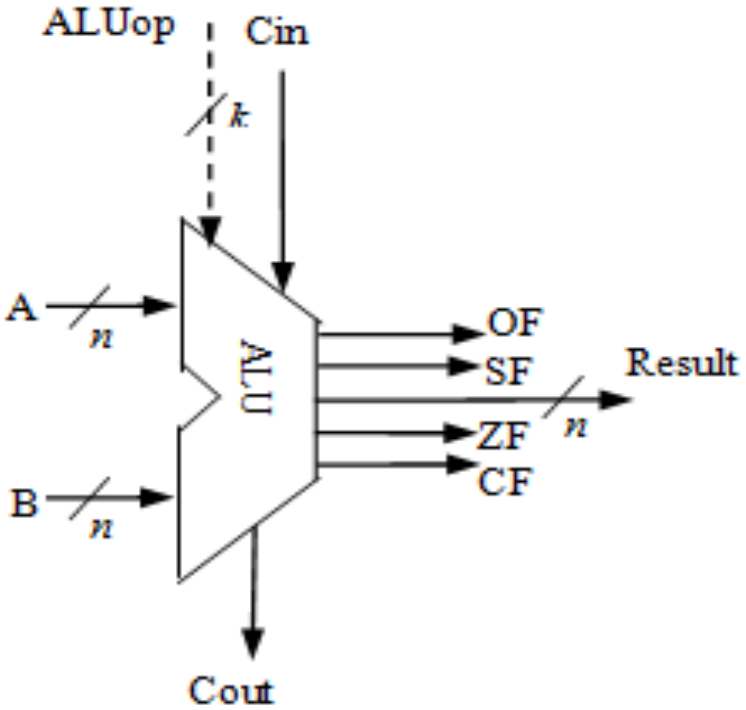
现代计算机结构模型

- 逻辑部件 → 加法器、MUX
- 带标志加法器 + MUX → 整数加减运算器
- 整数加减运算器 + 寄存器 + 控制逻辑
  → ALU、乘除运算器、浮点运算电路



算术逻辑单元（ALU）

- 基本算术运算与逻辑运算
  - 与、或、非、异或等逻辑运算
  - 无符号整数加减
  - 有符号整数加减
- 核心电路：整数加减运算部件
- 输出：和（差），标志位
- 操作控制端（ALUop）
  - 控制执行何种操作
  - 位数k，操作种类2^k



ALU 符号

操作控制端（ALUop）

- 位数k=3

ALUop	op
000	A+B
001	A-B
010	A&B
011	A|B
100	~A
101	A^B
110	A
111	未用

整数加减运算

- C语言程序中的整数
  - 有符号整数：char、short、int、long、…
  - 无符号整数
    - unsigned char、…
    - 指针、地址等
- 有/无符号整数加/减运算电路一样，整数加减运算部件，基于带标志加法器实现

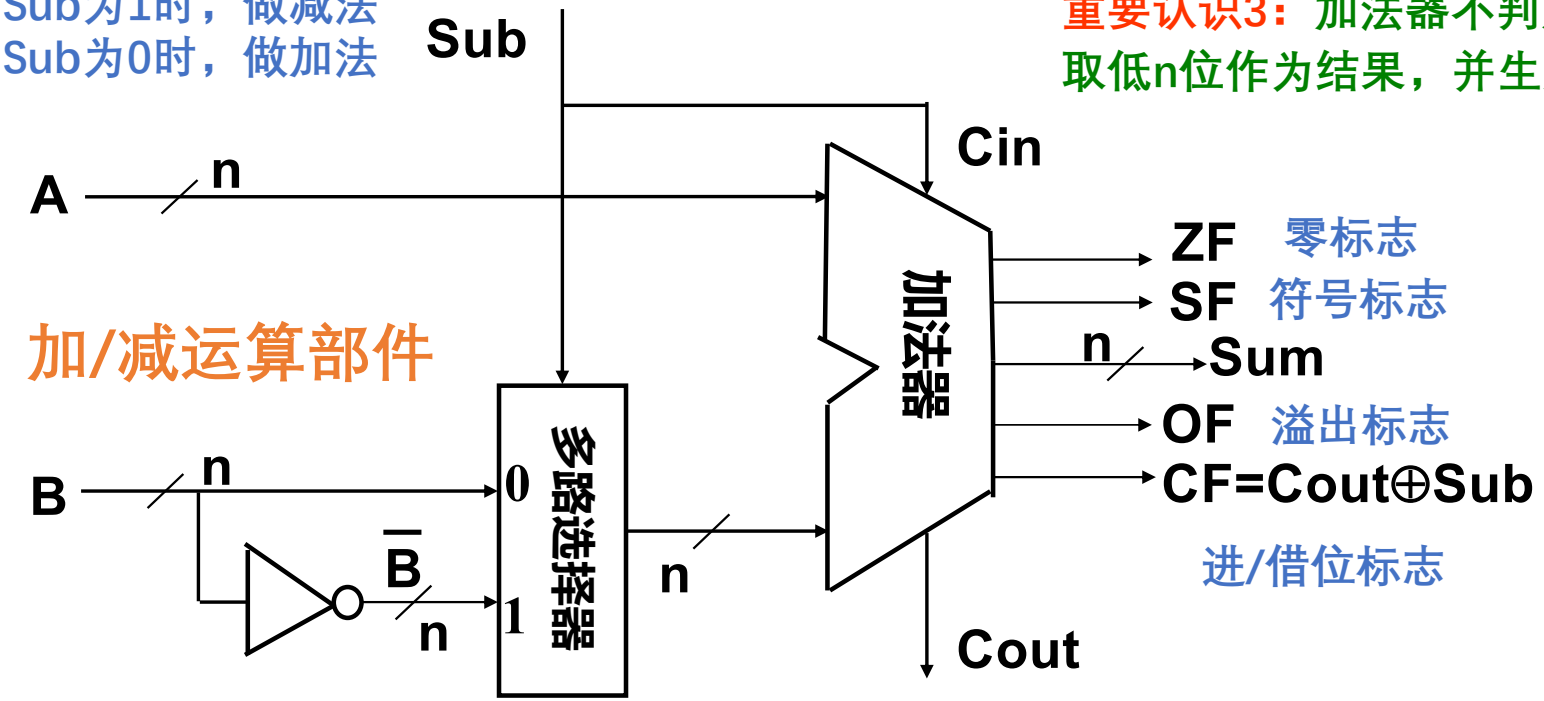
各种运算电路的核心

重要认识1：计算机中所有算术运算都基于加法器实现！

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。

当Sub为1时，做减法
当Sub为0时，做加法



整数加减运算部件：判溢出

- 无符号加减溢出条件：CF=1
  - 最高位进位/借位
- 有符号加减溢出条件：OF=1
  - 正+正=负
  - 负+负=正

- 例：n=4, A=1101, B=1010

```
A    1 1 0 1
B    1 0 1 0
C 1 0 0 0 0
F    0 1 1 1
OF=1, SF=0, ZF=0, CF=1
```

- 无符号
  - A(13)+B(10)=23>15
  - CF=1
  - 溢出
- 有符号
  - A(-3)+B(-6)=-9<-8
  - OF=1
  - 溢出

整数加减运算部件：判溢出

- 无符号加减溢出条件：CF=1
  - 最高位进位/借位
- 有符号加减溢出条件：OF=1
  - 正+正=负
  - 负+负=正

- 例：n=4, A=1010, B=0111

```
A    1 0 1 0
B    0 1 1 1
C 1  1 1 0 0
F    0 0 0 1
OF=0, SF=0, ZF=0, CF=1
```

- 无符号
  - A(10)+B(7)=17>15
  - CF=1
  - 溢出
- 有符号
  - A(-6)+B(7)=1>-8
  - OF=0
  - 未溢出

整数加减运算部件：判大小

- if (A>B) {…}
  - A-B > 0
  - A=9, B=6
- 无符号大于条件：CF=0
  - 最高位无借位（被减数大）
- 有符号大于条件：OF=SF

- 无符号
  - [A]补=1001，[B]补=0110
  - [A-B]补=[A]补+[-B]补
    =[A]补+$[\overline{B}]$补+1
  - $[\overline{B}]$补=1001

```
  1 0 0 1
  1 0 0 1
C 1 0 0 1 1
F   0 0 1 1
OF=1,SF=0,ZF=0,CF=0
A>B
```

整数加减运算部件：判大小

- if (A>B) {…}
  - A-B > 0
  - A=-3, B=5
- 无符号大于条件：CF=0
  - 最高位无借位（被减数大）
- 有符号大于条件：OF=SF
  - OF=0不溢出，SF=0被减数大

- 有符号
  - [A]补=1101，[B]补=0101
  - [A-B]补=[A]补+[-B]补
    =[A]补+$[\overline{B}]$补+1
  - $[\overline{B}]$补=1010

```
  1 1 0 1
  1 0 1 0
C 1 1 1 1 1
F   1 0 0 0
OF=0,SF=1,ZF=0,CF=0
A<B
```

整数加减法实例

```
int x = 1;
int y = 2147483647; // 2^31-1
unsigned int ux = x;
unsigned int uy = y;
int z1 = x + y;
int z2 = x - y;
unsigned int uz1 = ux + uy;
unsigned int uz2 = ux - uy;
```

求z1、z2、uz1、uz2的机器数和真值。

x和ux的机器数：[x]补=[ux]补=0…01
y和uy的机器数：[y]补=[uy]补=01…1

[z1]补=[x+y]补=[x]补+[y]补

```
   0 ..  01
   01 ..  1
C 011 .. 10
F  10 ..  0
```

OF=1(有符号溢出),CF=0,SF=1,ZF=0
[z1]补=10…0=0x80000000
$z1=-2^{31}=-2147483648$

整数加减法实例

```
int x = 1;
int y = 2147483647; // 2^31-1
unsigned int ux = x;
unsigned int uy = y;
int z1 = x + y;
int z2 = x - y;
unsigned int uz1 = ux + uy;
unsigned int uz2 = ux - uy;
```

求z1、z2、uz1、uz2的机器数和真值。

x和ux的机器数：[x]补=[ux]补=0…01
y和uy的机器数：[y]补=[uy]补=01…1

[z2]补=[x-y]补=[x]补+$\overline{[y]补}$+1

```
   0 ..  01
   10 ..  0
C 000 .. 11
F  10 .. 10          OF≠SF(有符号小于)
```

OF=0(有符号未溢出),CF=1,SF=1,ZF=0
[z2]补=10…010=0x80000002
z2=-2^{31}+2=-2147483646

整数加减法实例

```
int x = 1;
int y = 2147483647; // 2^31-1
unsigned int ux = x;
unsigned int uy = y;
int z1 = x + y;
int z2 = x - y;
unsigned int uz1 = ux + uy;
unsigned int uz2 = ux - uy;
```

求z1、z2、uz1、uz2的机器数和真值。

x和ux的机器数：[x]补=[ux]补=0…01
y和uy的机器数：[y]补=[uy]补=01…1

[uz1]补=[ux+uy]补=[ux]补+[uy]补

```
   0 ..  01
   01 ..  1
C 011 .. 10
F  10 ..  0
```

OF=1,CF=0(无符号未溢出),SF=1,ZF=0
[uz1]补=10…0=0x80000000
$uz1=2^{31}=2147483648$

整数加减法实例

```
int x = 1;
int y = 2147483647; // 2^31-1
unsigned int ux = x;
unsigned int uy = y;
int z1 = x + y;
int z2 = x - y;
unsigned int uz1 = ux + uy;
unsigned int uz2 = ux - uy;
```

求z1、z2、uz1、uz2的机器数和真值。

x和ux的机器数：[x]补=[ux]补=0…01
y和uy的机器数：[y]补=[uy]补=01…1

[uz2]补=[ux-uy]补=[ux]补+$\overline{[uy]补}$+1

```
   0 ..  01
   10 ..  0
C 000 .. 11
F  10 .. 10
```

OF=0,CF=1(无符号溢出，小于),SF=1,ZF=0
[uz2]补=10…010=0x80000002
uz2=2^{31}+2=2147483650

无符号整数加法溢出判断

- 无符号加减溢出条件：CF=1
  - 无符号加，Cin=0，仅当Cout=1时，CF=1
    - Cout=1说明result=x+y超出F表示范围

$$\text{result}=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n \leqslant x+y<2^{n+1}) \end{cases}$$

- 溢出时，一定满足result<x且result<y

有符号整数加法溢出判断

- 有符号加减溢出条件：OF=1
  - OF=Cn^Cn-1
    - Cn=0, Cn-1=1
    - Cn=1, Cn-1=0

$$\text{result}=\begin{cases} x+y-2^{n} & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^{n} & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

整数乘法运算

- 两个n位整数相乘
  - 数学中：积最少n位、最多2n位
    - 0*0=0, 3*3=9, 4*4=16, 9*9=81
    - 10*10=100, 99*99=9801
  - 3*3=3+3+3，加法器是最基本的算术运算单元
  - 计算机：积一定是低n位
    - C语言：参与运算的操作数的类型和结果的类型必须一致，否则先转换再运算

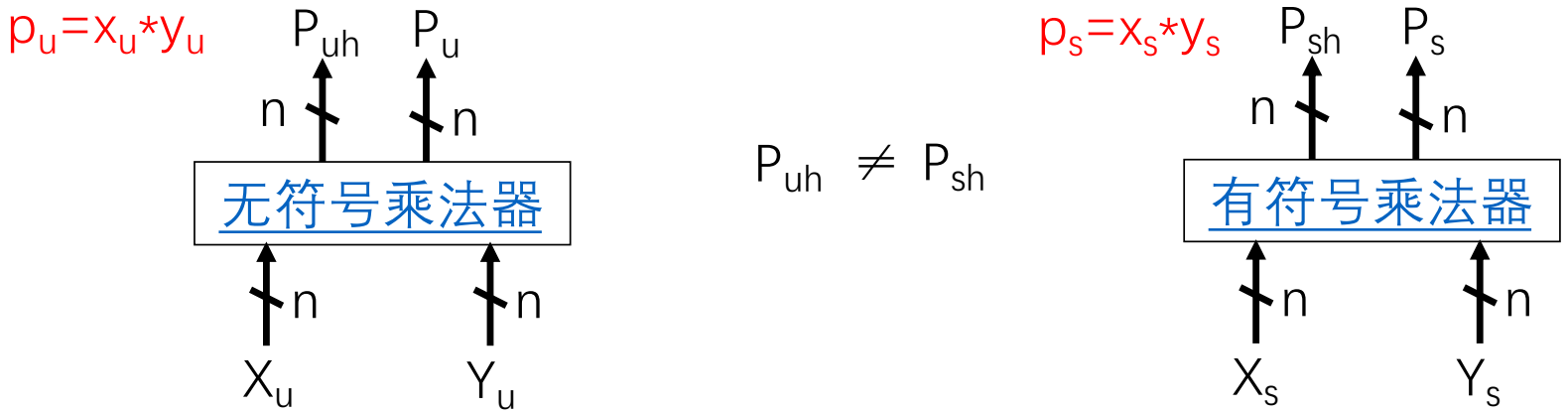
```
int mul(int x, int y)
{
  int z=x*y;
  return z;
}
```

int x, int y（32位）
乘法表达式 x*y，乘法指令，乘法运算电路（64位）
乘积64位
int z（32位）

整数乘法运算

- 两个n位无符号整数x_u和y_u，对应的机器数为X_u和Y_u
- $p_u=x_u \times y_u$，p_u为n位无符号整数，对应的机器数为P_u
- 无符号整数乘法指令（MUL）

- 两个n位有符号整数x_s和y_s，对应的机器数为X_s和Y_s
- $p_s=x_s \times y_s$，p_s为n位无符号整数，对应的机器数为P_s
- 有符号整数乘法指令（IMUL）



整数乘法运算

- X*Y的高n位可以用来判断溢出
  - 无符号：若高n位全0，则不溢出，否则溢出
  - 有符号：若高n位全0或全1且等于低n位的最高位，则不溢出，否则溢出

运算	x	X	y	Y	x×y	X×Y	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	0000 0110	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	1111 1000	-8	1000	不溢出

整数乘法溢出漏洞

- 2002年，Sun公司RPC XDR库，整数溢出漏洞，攻击者获取root权限

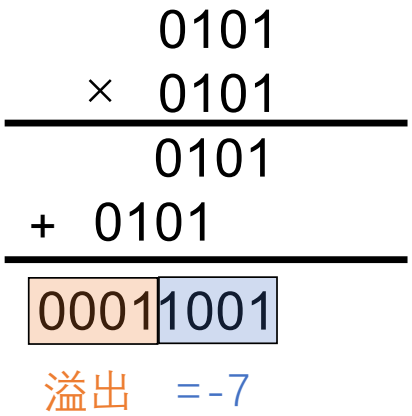
```
/* 复制数组到堆中，count为数组元素个数 */
int copy_array(int *array, int count) {
  int i;
  int *myarray = (int *) malloc(count*sizeof(int));
  if (myarray == NULL) return -1;
  for (i = 0; i < count; i++)
    myarray[i] = array[i];
  return count;
}
```

count很大时，
count*sizeof(int)溢出。
如count=2^{30}+1时，
count*sizeof(int)=4。

攻击者构造array用预设信息覆盖已分配的堆空间

整数乘法运算

- 与数学运算的区别
- $x^2 \geq 0$ 一定成立吗?
  - 若x是有符号整数，不一定!
  - 当n=4时，5^2=-7<0



变量与常量之间的乘法运算

- 整数乘法运算比移位和加法运算更加耗时
  - 乘法运算：一般需要多个时钟周期
  - 移位、加减运算：一般只要一个或更少时钟周期
- 编译器：用移位和加减运算的组合来代替乘法运算
  - x*20 转换为 (x<<4)+(x<<2)
  - 一次乘法 转换为 两次移位和一次加法

  - 不管有/无符号，是否溢出，结果始终是一样的

整数除法运算

- 除法器的基本工作原理
  - A / B 被分解为
    - A 重复减去 B，直到 A 小于 B
    - 减去的次数是商，剩下的 A 是余数
  - 7 / 2 = 3 … 1
    - 7 – 2 = 5，5 – 2 = 3，3 – 2 = 1
  - 加法器是最基本的算术运算单元
  - 一次除法运算需要几十个或更多时钟周期

整数除法运算

- 舍入
  - 整数除法，商也是整数
  - 不能整除时舍入，向0舍入
    - 正数商向下取整（floor）
      - 7/2=3
    - 负数商向上取整（ceiling）
      - -7/2=-3

整数除法运算

- 溢出
  - 对于有符号，n位整数除以n位整数
    - 商的绝对值不可能大于被除数的绝对值
    - 只有$-2^{n-1} / -1 = 2^{n-1}$会发生溢出
- 除0
  - 整数除以0的结果不能用一个机器数表示
  - 触发操作系统异常（SIGFPE）

整数除法运算

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

运行结果：-2147483648（-2^{31}）

反汇编：objdump –S a.out

- 优化为 neg 指令（取负）
- $-(-2^{31}) = 2^{31} > 2^{31}-1$
- 解释为%d（32位有符号）：-2^{31}

```
int a = 0x80000000;
int c = -1
int b = a / c;
printf("%d\n", b);
```

运行结果：Floating point exception (core dumped)

0x8000000 和 -1 送入除法器，
硬件触发算术异常

变量与常量之间的除法运算

- 一次除法运算需要几十个或更多时钟周期
- 编译器：一个变量除以一个2的幂次形式的整数时，采用右移运算
  - 结果取整
  - 能整除时，直接右移
    - 12 / 4 = 3，逻辑右移 0000 1100 >> 2 = 0000 0011 = 3
    - -12 / 4 = -3，算术右移 1111 0100 >> 2 = 1111 1101 = -3
  - 不能整除时，即右移移出的位中存在非0，向0舍入
    - 无符号数、有符号正整数：向下取整，移出的低位直接舍弃
      - 14 / 4 = 3，0000 1110 >> 2 = 0000 0011
    - 有符号负整数
      - -14 / 4 = -3，~~1111 0010 >> 2 = 1111 1100 = -4~~
      1. 先纠偏（加偏移量 2^k-1）：1111 0010 + 0000 0011（2^2-1）= 1111 0101
      2. 再算术右移：1111 0101 >> 2 = 1111 1101=-3

为什么需要纠偏？
整数除法是“向零取整”，算术右移是“向下取整”。负数时两者结果不同，所以需要“纠偏”来模拟向零取整。

例题：变量与常量之间的除法运算

- 设x为int类型变量，请给出一个用来计算x/32的值的函数div32。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

解：

- int，有符号整数，32位。除以32，即右移5位。
- 若x为正数，x直接算术右移5位；若x为负数，x先加偏移量$2^5-1=31$，再直接算术右移5位。

$$(x >= 0 ? x : (x+31)) >> 5$$

- 但是不能使用比较语句、条件语句
- 正数视作先偏移0，即用按位运算得到正数变0，负数变31。

```
int div32(int x)
{
  int b = (x>>31) & 0x1F;
  return (x+b)>>5;
}
```