# Integers

# Decimal, Binary, Hexadecimal

$$1209_{[10]} = 1\times10^{3} + 2\times10^{2} + 0\times10^{1} + 9\times10^{0}$$

$$100101_{[2]} = 1\times2^{5} + 0\times2^{4} + 0\times2^{3} + 1\times2^{2} + 0\times2^{1} + 1\times2^{0}$$

$$B0A_{[16]} = B\times16^{2} + 0\times16^{1} + A\times16^{0}$$

base

Position of digit

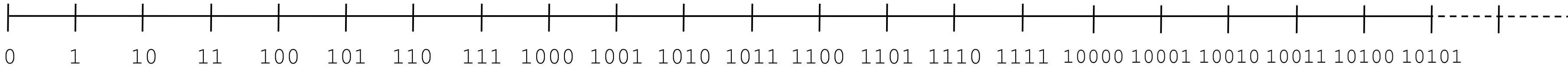
# Hexadecimal!

- $0_{[16]}$ $0000_{[2]}$ $0_{[10]}$
- $1_{[16]}$ $0001_{[2]}$ $1_{[10]}$
- $2_{[16]}$ $0010_{[2]}$ $2_{[10]}$
- $3_{[16]}$ $0011_{[2]}$ $3_{[10]}$
- $4_{[16]}$ $0100_{[2]}$ $4_{[10]}$
- $5_{[16]}$ $0101_{[2]}$ $5_{[10]}$
- $6_{[16]}$ $0110_{[2]}$ $6_{[10]}$
- $7_{[16]}$ $0111_{[2]}$ $7_{[10]}$
- $8_{[16]}$ $1000_{[2]}$ $8_{[10]}$
- $9_{[16]}$ $1001_{[2]}$ $9_{[10]}$
- $A_{[16]}$ $1010_{[2]}$ $10_{[10]}$
- $B_{[16]}$ $1011_{[2]}$ $11_{[10]}$
- $C_{[16]}$ $1100_{[2]}$ $12_{[10]}$
- $D_{[16]}$ $1101_{[2]}$ $13_{[10]}$
- $E_{[16]}$ $1110_{[2]}$ $14_{[10]}$
- $F_{[16]}$ $1111_{[2]}$ $15_{[10]}$

# Binary arithmetic

```
     1
  1010    (10)
+ 1010    (10)
------
 10100    (20)
```

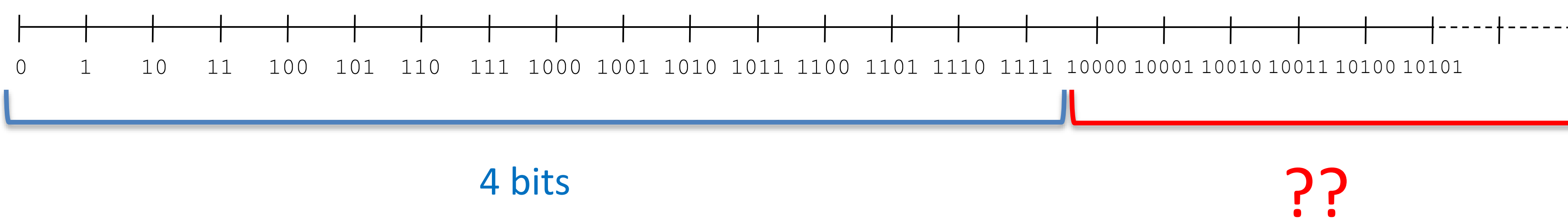
```
     110    (6)
  x 1010    (10)
  ------
       0
     110
      0
+   110
  ------
  111100    (60)
```



# Binary arithmetic … on 4-bit words

```
     1
   1010   (10)
 + 1010   (10)
 -------
  10100   (20)
```

```
     0110   (6)
   x 1010   (10)
 ---------
        0
      110
      0
 +  110
 ---------
   111100   (60)
```



# Overflow

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_E
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(T
end if;
P_M_DERIVE(T_ALG.E_BH) :=
    UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M
```

Ariane 5

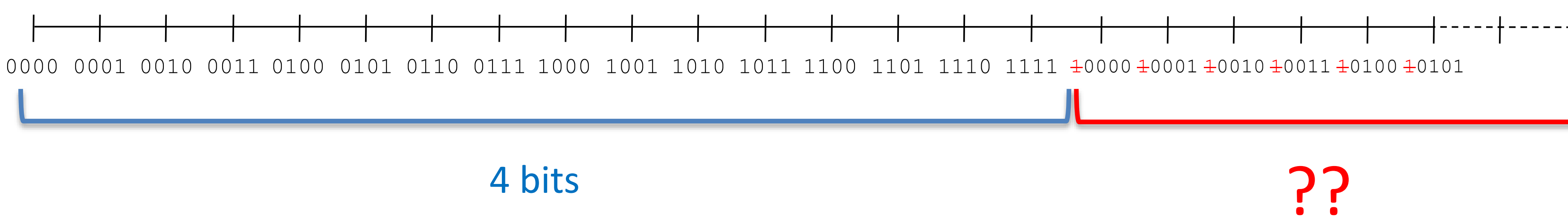
# Modular arithmetic

*1*
**1010** (10)
\+ **1010** (10)
~~1~~**0100** (20)

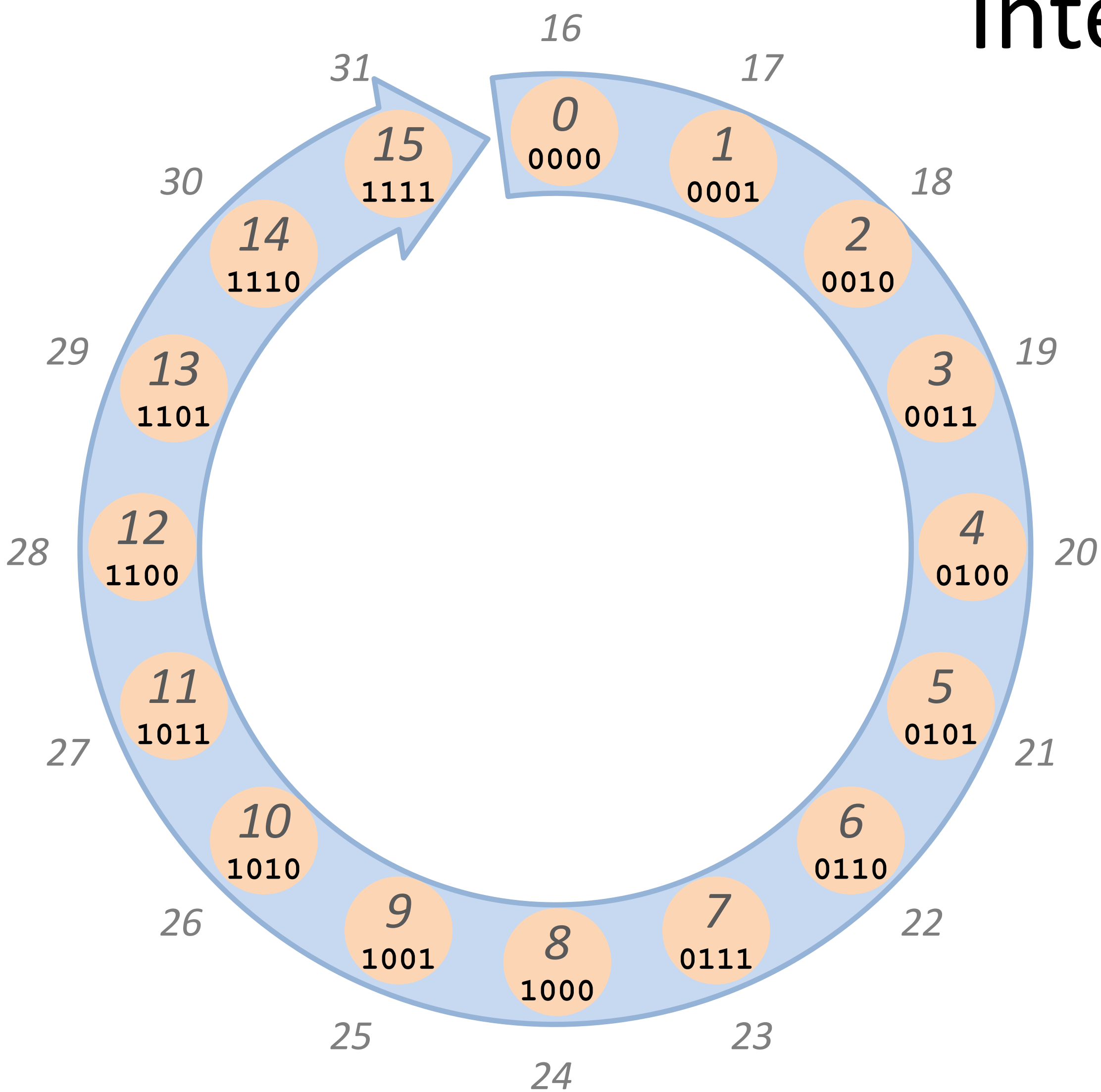
**0100** (4)

**0110** (6)
x **1010** (10)
*0000*
*0110*
*0000*
\+ *0110*
~~11~~**1100** (60)

**1100** (12)



# Integers modulo 16



# Laws of modular arithmetic

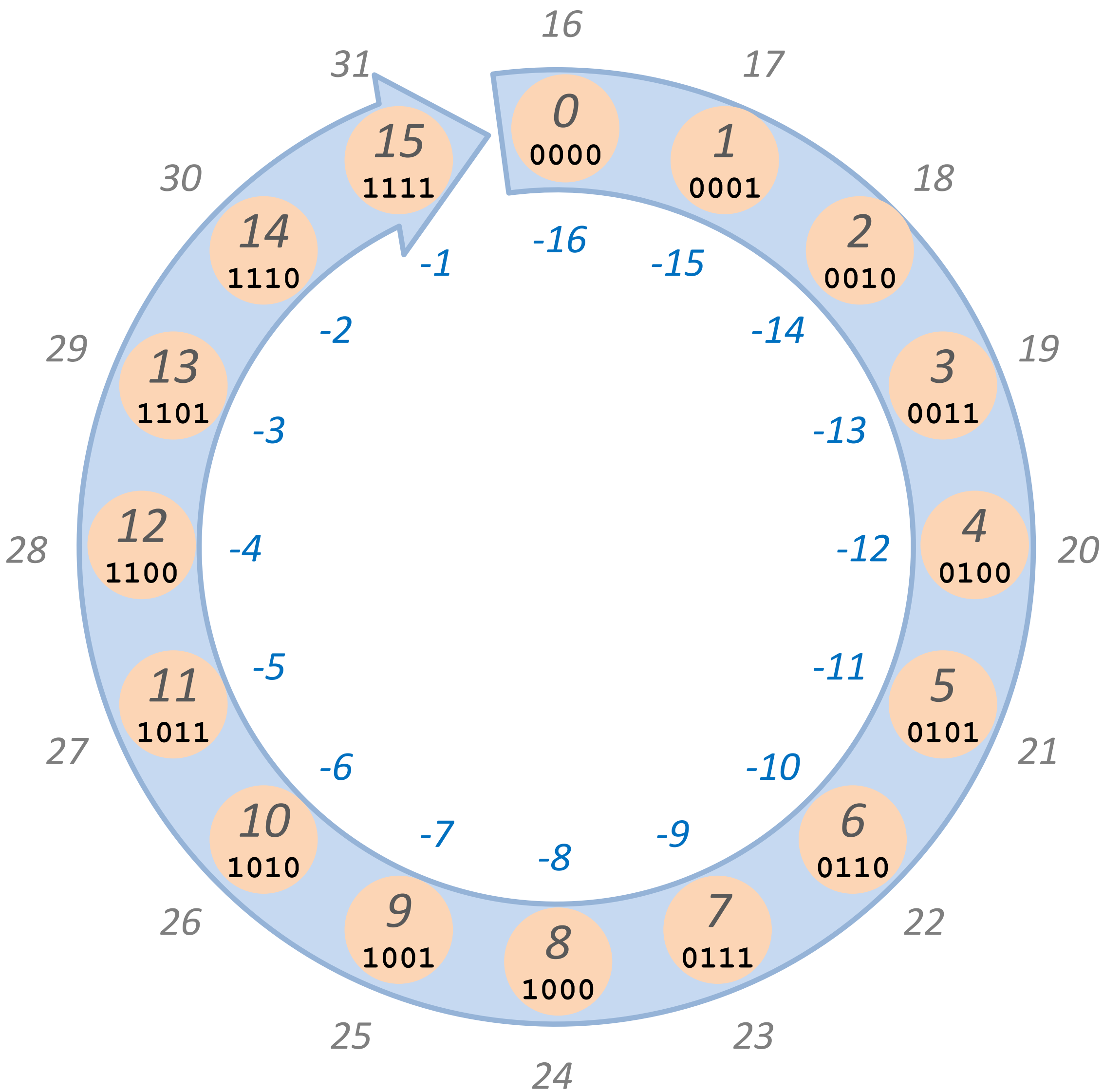
| | |
|---|---|
| $x + y = y+ x$ | Commutativity of addition |
| $(x + y) + z = x + (y + z)$ | Associativity of addition |
| $x + 0 = x$ | Additive unit |
| $x *= y * x$ | Commutativity of multiplication |
| $(x * y) * z = x * (y * z)$ | Associativity of multiplication |
| $x * 1 = x$ | Multiplicative unit |
| $x * (y + z) = x * y + x * z$ | Distributivity |
| $x * 0 = 0$ | Annihilation |

Same laws as traditional arithmetic!

# Reasoning about `int`s

```
string foo(int x) {
  int z = 1+x;
  if (x+1 == z)
    return "Good";
  else
    return "Bad";
}
```

# What about the negatives?



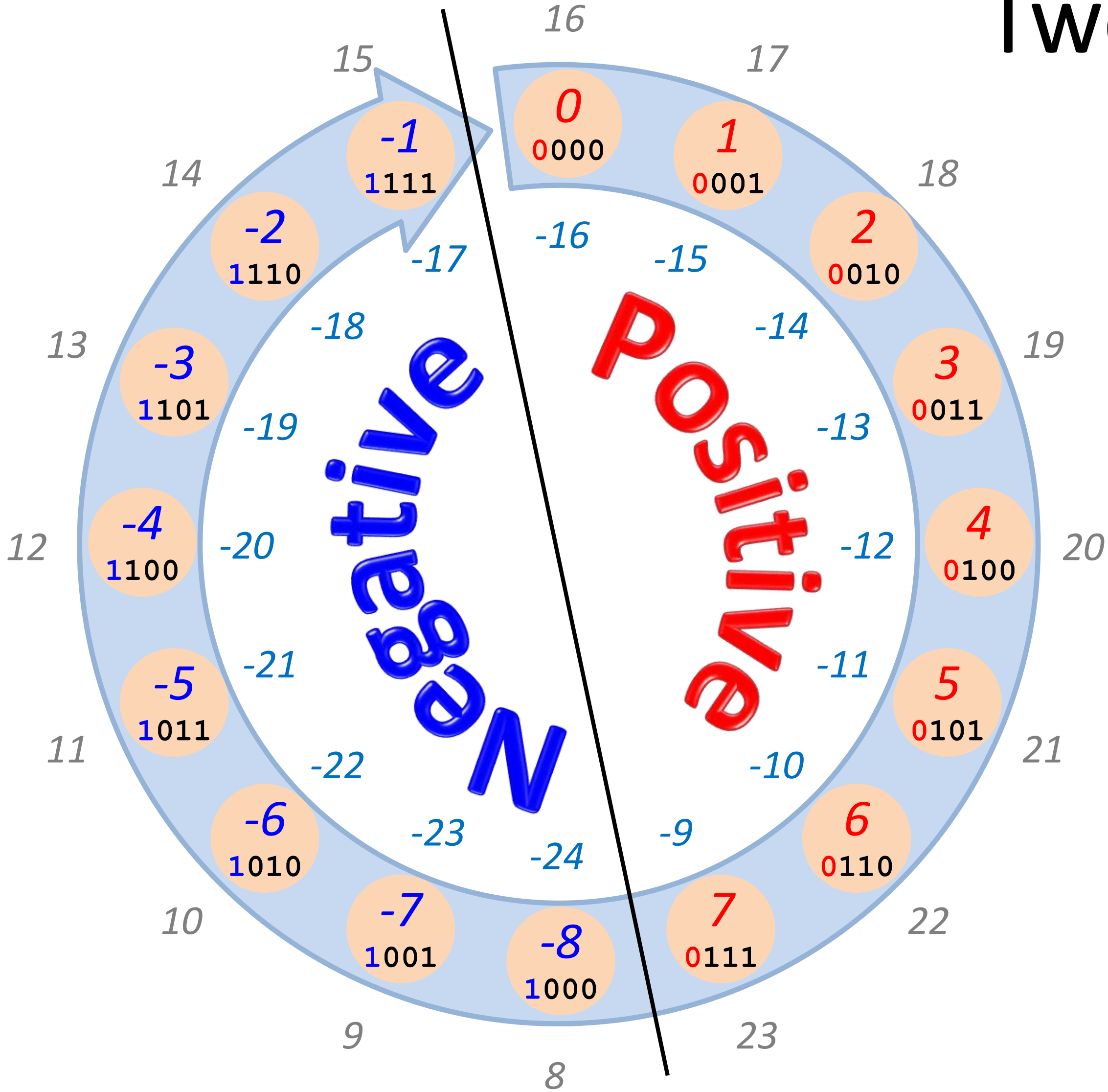
# Subtraction

- $x - y$ is stepping $y$ times counter-clockwise from $x$
- Define $-x = 0 - x$
- Then,

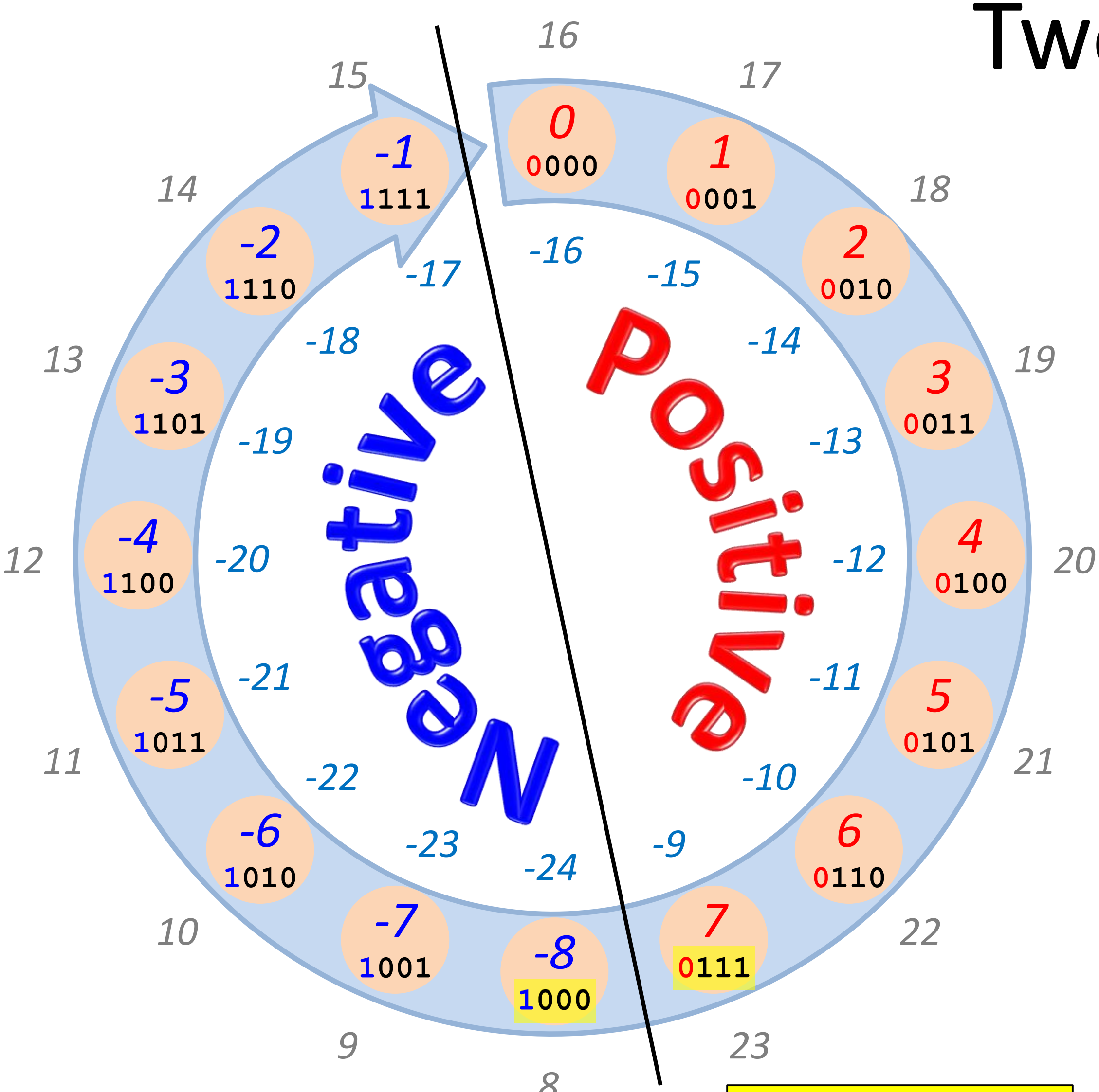
| | |
|---|---|
| $x + (-x) = 0$ | Additive inverse |
| $-(-x) = x$ | Cancelation |

Same laws as traditional arithmetic!

# Two’s complement



# Two’s complement




int_min = $-2^3$

int_max = $2^3 - 1$

# Reasoning about `int`s

```
string bar(int x) {
  if (x+1 > x)
    return "Good";
  else
    return "Strange";
}
```

# Pixels as 32-bit `int`'s (ARGB)