Integers

Number Representation

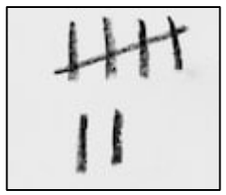
Representing Numbers

- We, people, have many ways to represent numbers

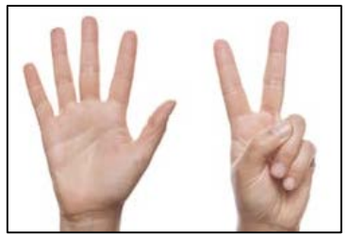
7

सात

sept



VII

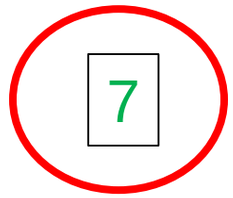


seven

sjö

- They all express the same concept
  - that some collection consists of *seven* things

Decimal Numbers



- The **decimal representation** is succinct and systematic
  - We have ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    - each represents a number between 0 and 9
    - they are the **digits**
  - "ten" is the **base**

This comes from us having 10 fingers

- Any number is represented as a sequence of digits
  - the **position** *i* of a digit *d* indicates its importance
    - it contributes $d \times 10^i$ to the value of the number
  - the value of the number is the sum of the contribution of each position

1 is at position 3 | **2** is at position 2 | **0** is at position 1 | **9** is at position 0

$$\mathbf{1209} = \mathbf{1}\times 10^3 + \mathbf{2}\times 10^2 + \mathbf{0}\times 10^1 + \mathbf{9}\times 10^0$$

10 is the base

Decimal Numbers

- Positional systems make it easy to do calculations
  - **addition** is done position by position

```
   1   1
    1209
+   9517
-------
   10726
```

carry

We used our
10 fingers for that

- **multiplication** is done as iterated additions

```
    1209
×    402
-------
    2418
     0
+ 4836
-------
  486018
```

Binary Numbers

- Computers have *one* way to represent information: **binary**
  - they use two symbols, 0 and 1

> • 0 = no currant
> • 1 = currant
> *(in reality, it's more complicated)*

- In particular, they represent numbers in positional notation using base 2
  - that's the **binary representation**

That's what we call the binary digits 0 and 1

- Any number is represented as a sequence of **bits**
  - the **position** *i* of a bit *b* indicates its importance
    - ➢ it contributes $b \times 2^i$ to the value of the number
  - the value of the number is the sum of the contribution of each position

1 is at position 5 … **0** is at position 3 … **1** is at position 0

$$\mathbf{100101} = \mathbf{1}\times 2^5 + \mathbf{0}\times 2^4 + \mathbf{0}\times 2^3 + \mathbf{1}\times 2^2 + \mathbf{0}\times 2^1 + \mathbf{1}\times 2^0$$

2 is the base

Binary Numbers

- Positional systems make it easy to do calculations
  - **addition** is done position by position

```
    111
   11011
+   1110
------
  101001
```

carry

Here, 1+1 = 0
with a carry of 1

This works **exactly** as with decimal numbers

- **multiplication** is done as iterated additions

```
    1010
  ×  101
------
    1010
     0
+ 1010
------
  110010
```

Converting Binary Numbers to Decimal

Converting Decimal Numbers to Binary

Decimal, Binary, Hexadecimal

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

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

$$\mathbf{B0A}_{[16]} = \mathbf{B}\times16^2 + \mathbf{0}\times16^1 + \mathbf{A}\times16^0$$

base

position of digit

Hexadecimal

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

Hexadecimal

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

- What is this number in hex?

1100 0000 1111 1111 1110 1110

Fixed-size Number Representation

Binary Arithmetic

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

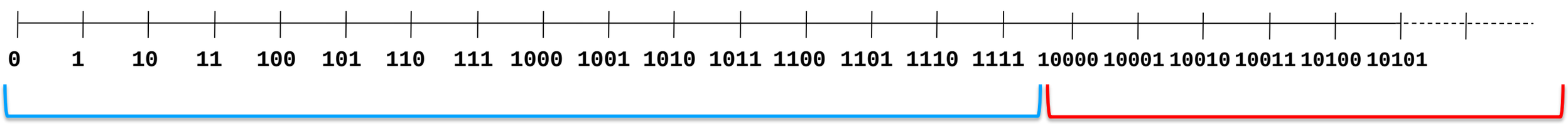
```
    110    (6)
 × 1010    (10)
-------
      0
    110
   0
+ 110
-------
 111100    (60)
```

- *What if we have only 4 binary digits to represent integers?*

Binary Arithmetic … on 4-bit Words

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

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



Fixed-size Representation

32 bits in C0

- Allows efficient operations in hardware
- We have to handle **overflow**
  - Raise error/exception
  - Something else …

Handling Overflow as Error

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV
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

Handling Overflow as Error

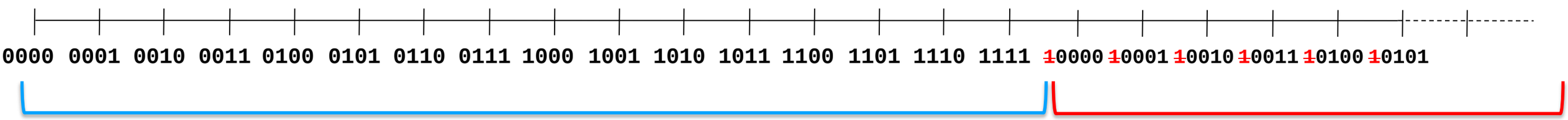
- Hard to reason about code
  - *n + (n - n)* and *(n + n) - n* are equal in math …
- … but with fixed size numbers,
  - *n + (n - n)* always equal to *n*
  - *(n + n) - n* may overflow

We want to be able to use the laws of arithmetic

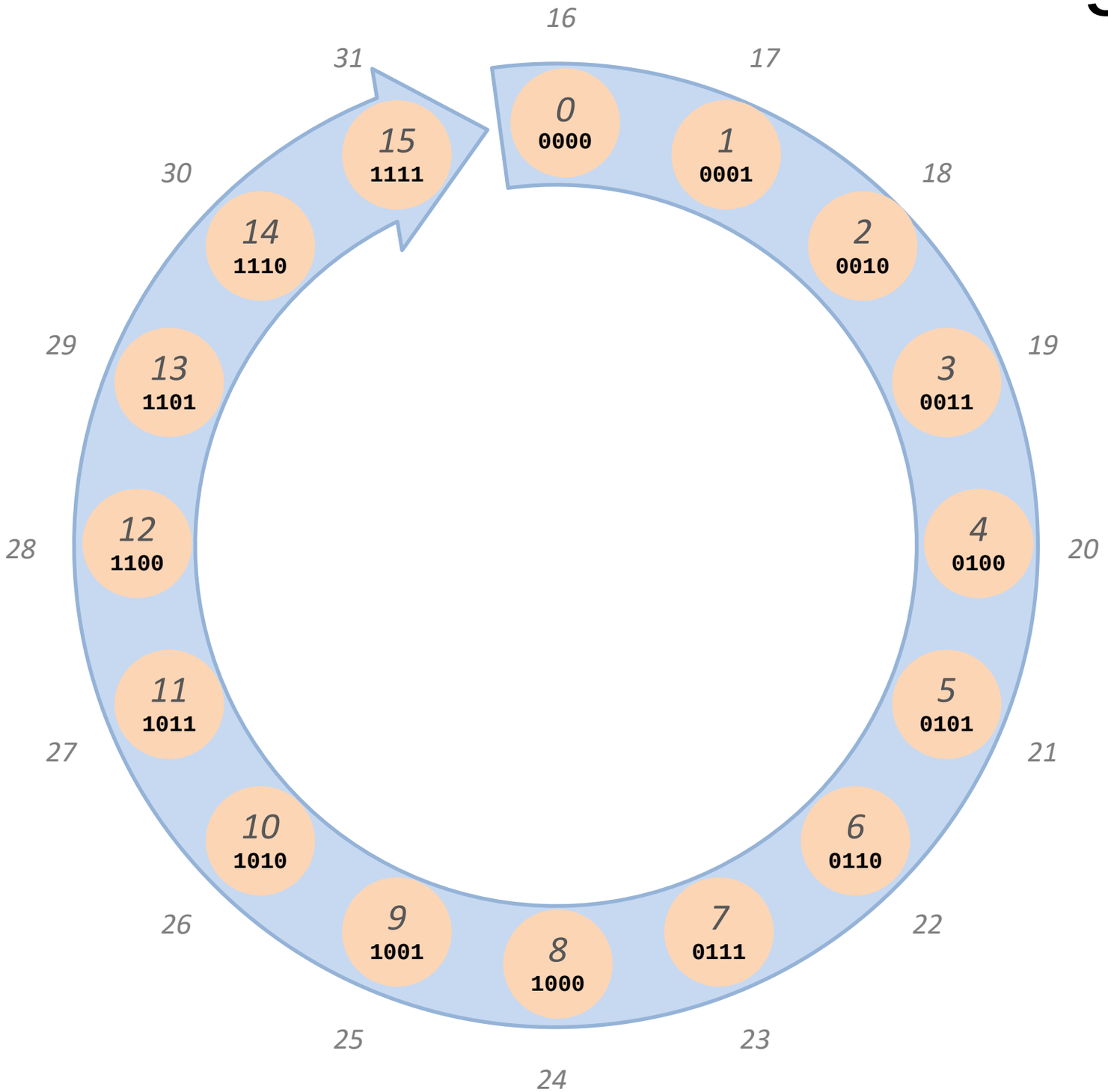
Modular Arithmetic

```
      1
   1010   (10)
+  1010   (10)
-------
  ~~1~~0100  (20)
```

```
    110    (6)
×  1010   (10)
-------
      0
    110
   0
+ 110
-------
  ~~11~~1100  (60)
```



Integers Modulo 16



- From number line to a number circle
- Addition is moving clockwise
- Arithmetic mod 16 (= 2^4), corresponds to a fictional machine with word size 4

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
$x * y = y * x$	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

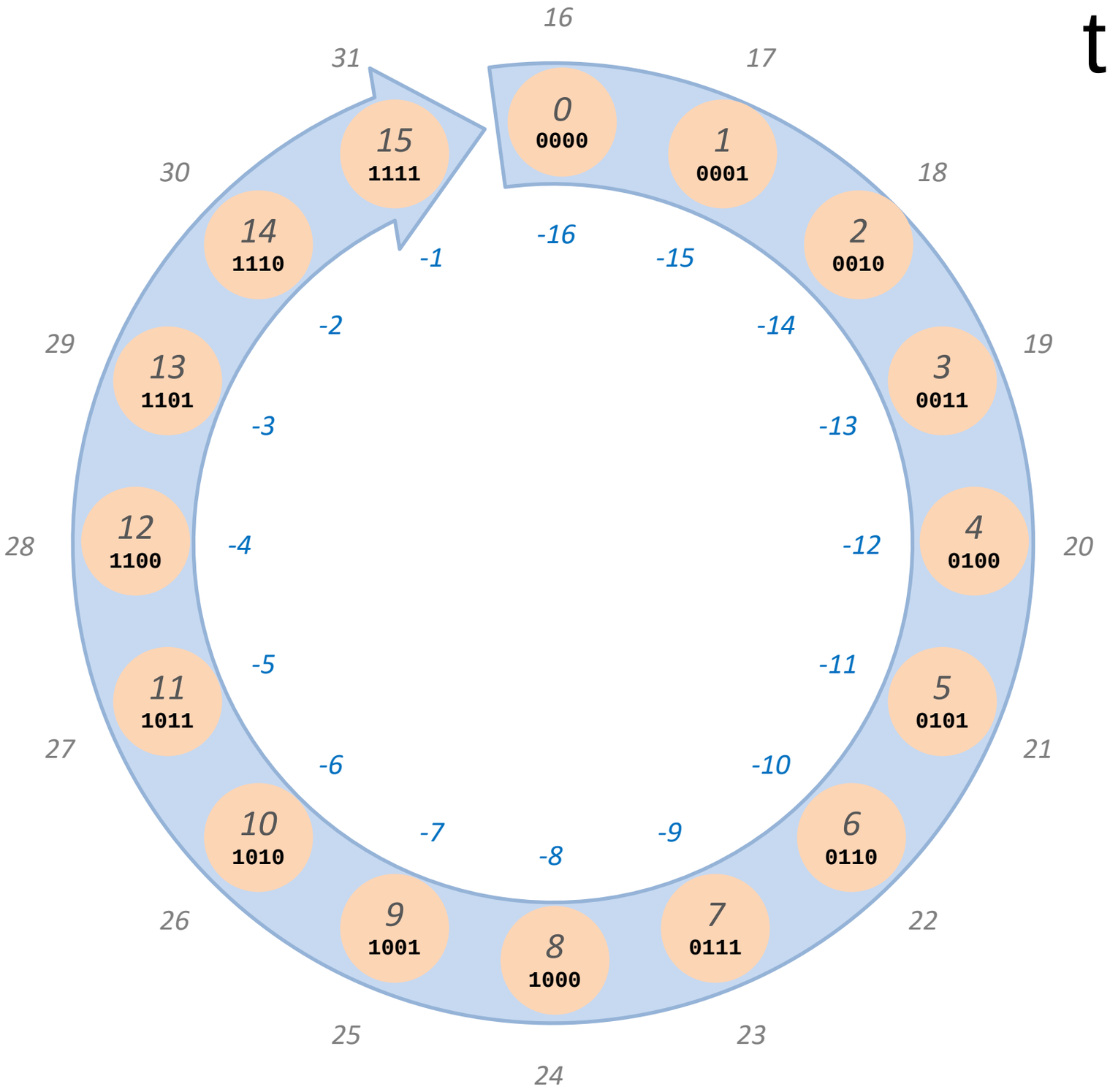
```
string foo(int x) {
  int z = 1+x;
  if (x+1 == z)
     return "Good";
  else
     return "Bad";
}
```

This is equivalent to
x+1 == 1+x
by substitution

x+1 == 1+x
is always **true**
by commutativity of addition

… so foo always returns "Good"

What about the Negatives?



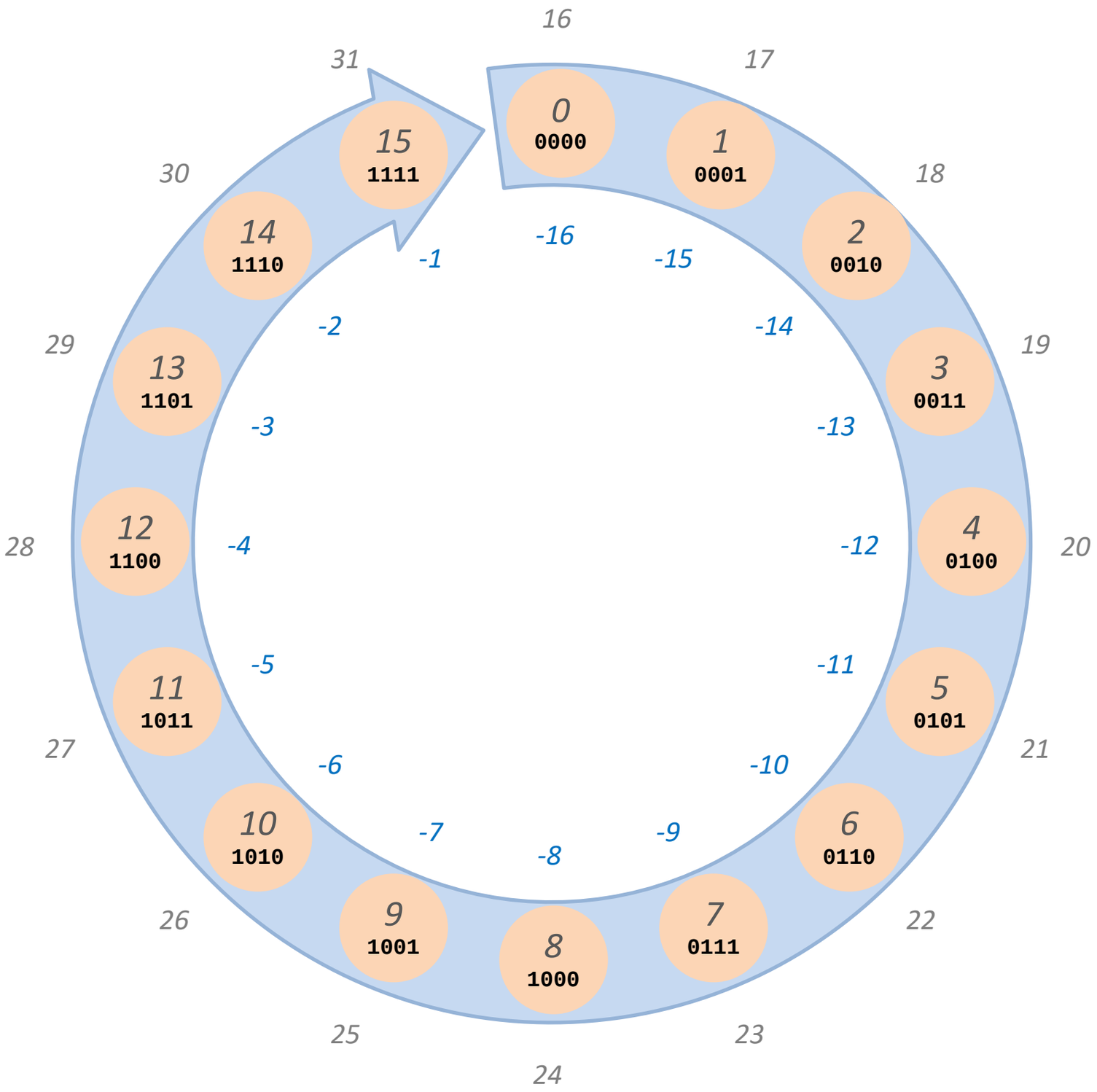
Subtraction

- $x - y$ 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!

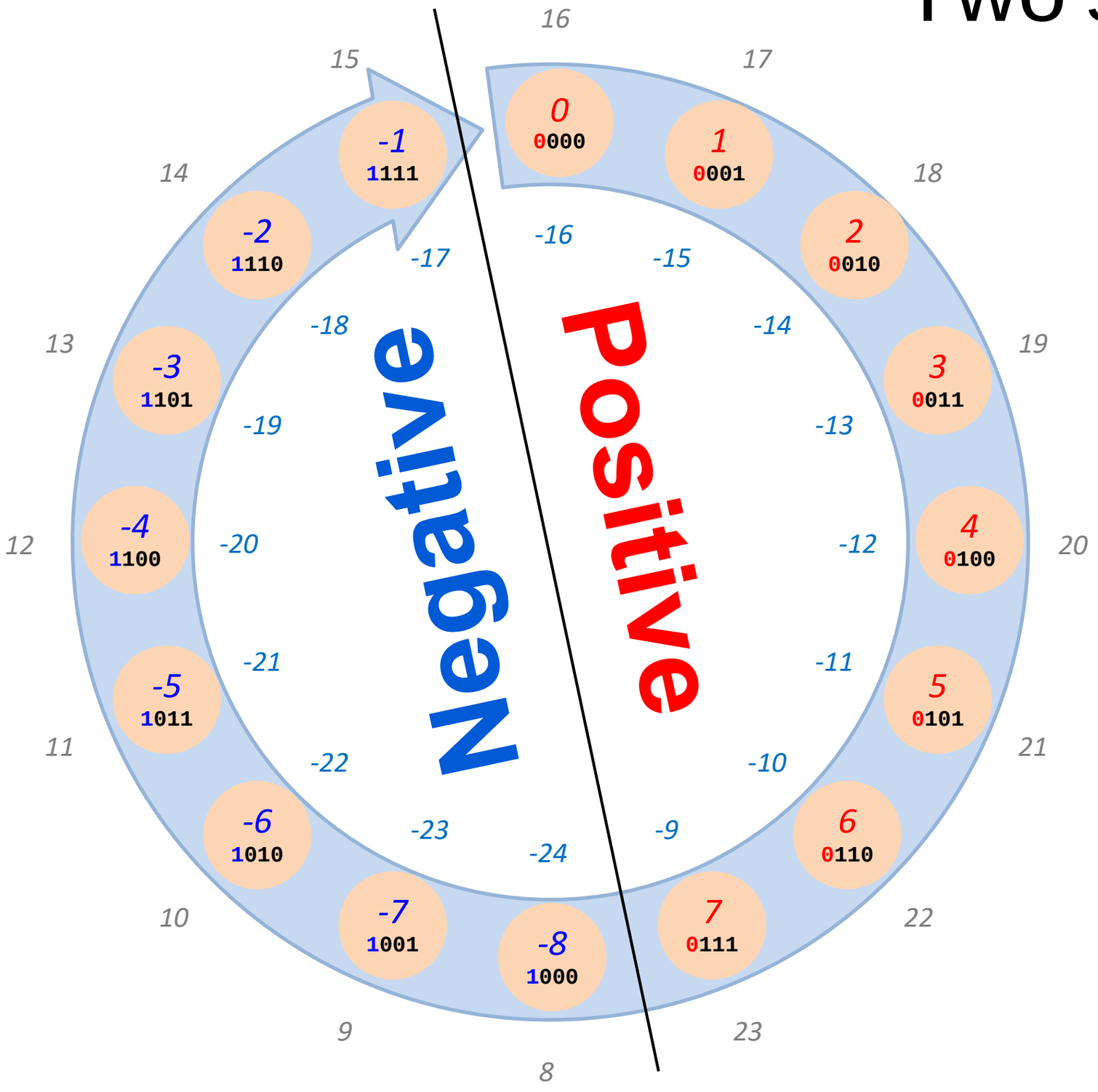
Two’s Complement



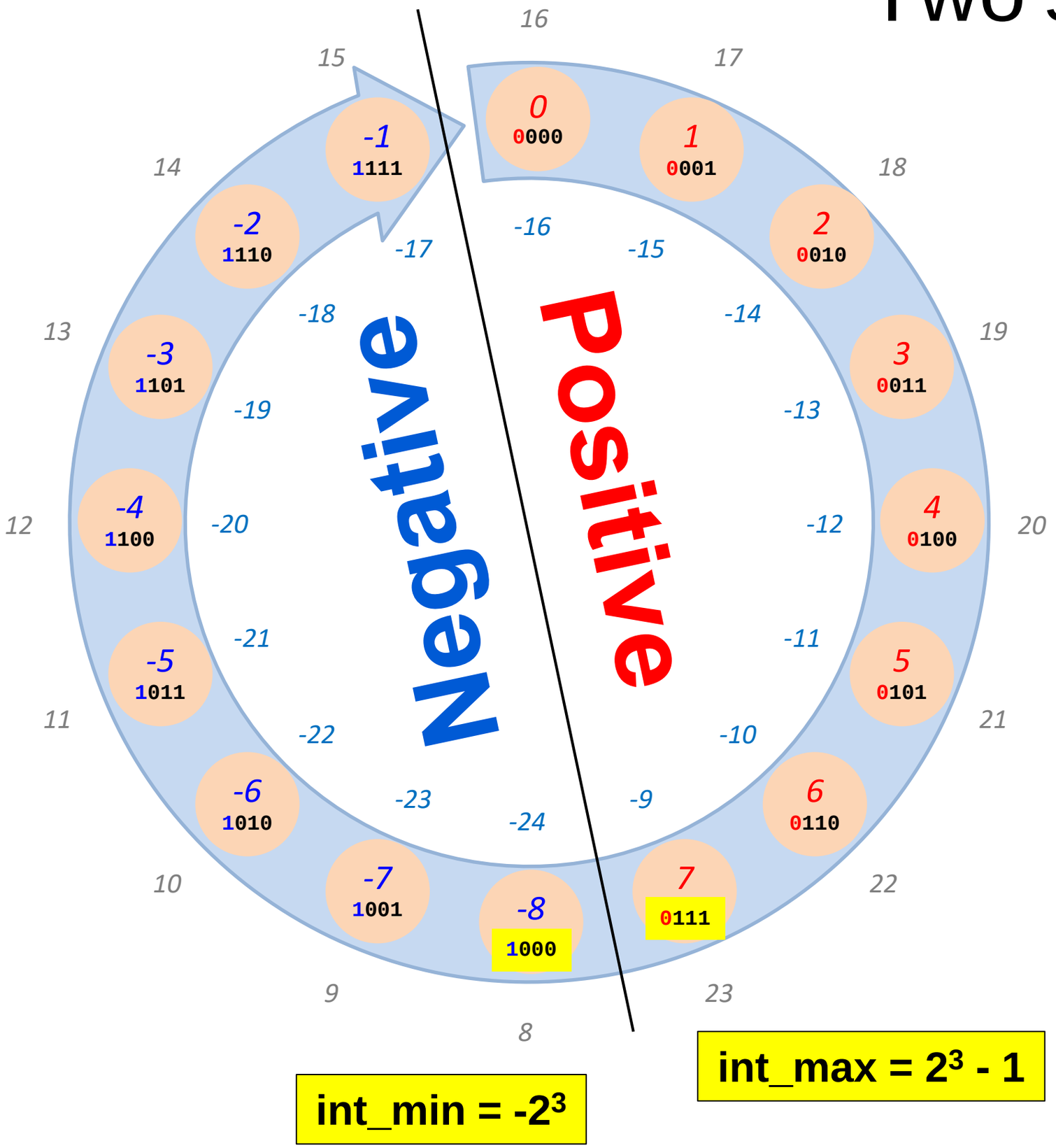
Rendering

- How should the computer print back to us **0100**?
  - 4? 20? -12?
- What about **1101**?
  - 13? 29? -3?

Two's Complement



Two's Complement



- With k bits
  - int_max = 2k - 1
  - int_min = -2k
- Off by one because of 0
- We can now talk about **ordering**
  - x < y, x ≤ y, …

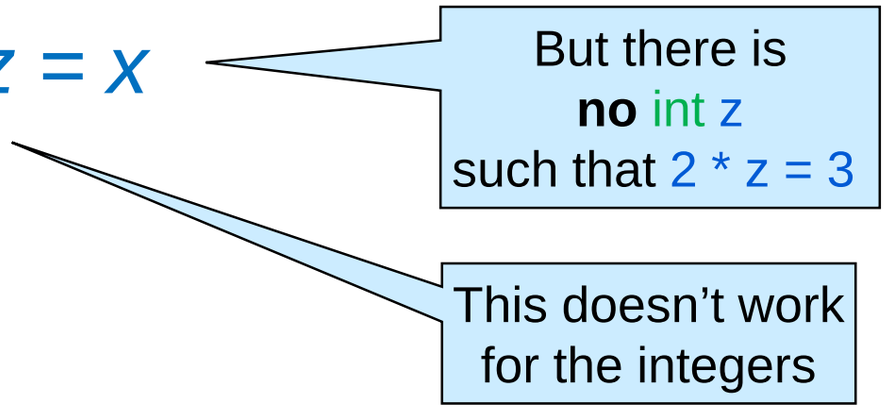
Reasoning about int`s

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

When is x+1 **not** larger than x in C0?

Division and Modulus

- In calculus, *(x/y)* is *z* such that *y * z = x*



- Introduce a new operation to pick up the slack: **modulus**
  - ➢ *(x/y) * y + (x%y) = x*
  - ➢ *0 <= |x % y| < |y|*

- *x/y* rounds down for positive *x* and *y*
- What should *(x/y)* round down to for negative numbers?
  - C0 rounds "down" to 0
  - Python rounds towards -∞

Safety Requirements

- Division by 0 is undefined (same for modulus)
  - Any time we have x/y in a program, we must have a reason to believe that y != 0
  - This is a safety requirement
  - x/y and x%y have *preconditions*

```
//@requires y != 0;
//@requires !(x == int_min() && y == -1);
```

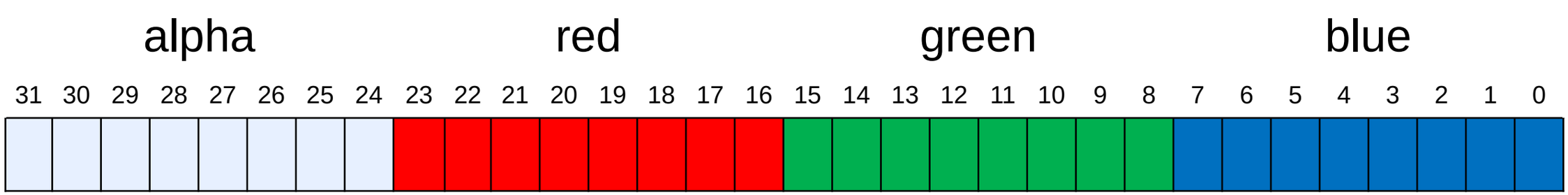
  - because chips raise errors on these inputs

Bit Patterns

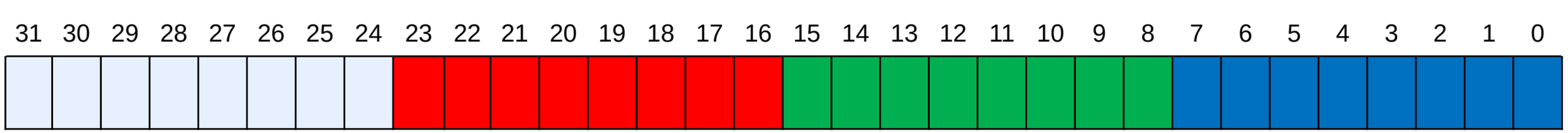
Bit Patterns

- use int to represent data other than numbers
  - pixels
  - network packets
  - …
- New set of operations to manipulate them
  - bitwise operators
  - shifts

Pixels as 32-bit int's (ARGB)

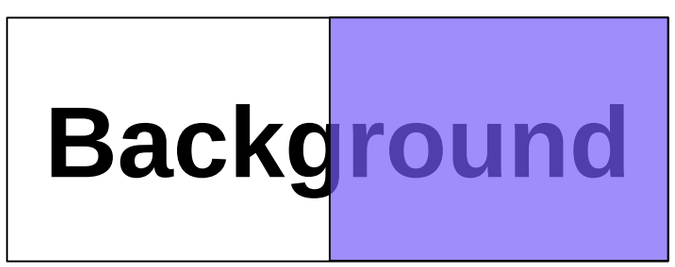


Example: Pixel



1011 0011 0111 0011 0101 1010 1111 1001

B 3 7 3 5 A F 9



Bitwise Operations

and

&	0	1
0	0	0
1	0	1

or

|	0	1
0	0	1
1	1	1

xor

^	0	1
0	0	1
1	1	0

not

~	0	1
	1	0

Bitwise Operations

- Apply to `int`'s, position by position
  - examples with just 4 bits

```
  1010          1010
& 1001        | 1001
------        ------
  1000          1011
```

- Related to && and || but not interchangeable
  - take int's as input, not bool's

Bitwise Operations

b

&	0	1
0	0	0
1	0	1

mask

b & 0 = 0

b & 1 = b

b

|	0	1
0	0	1
1	1	1

b | 0 = b

b | 1 = 1

~	0	1
	1	0

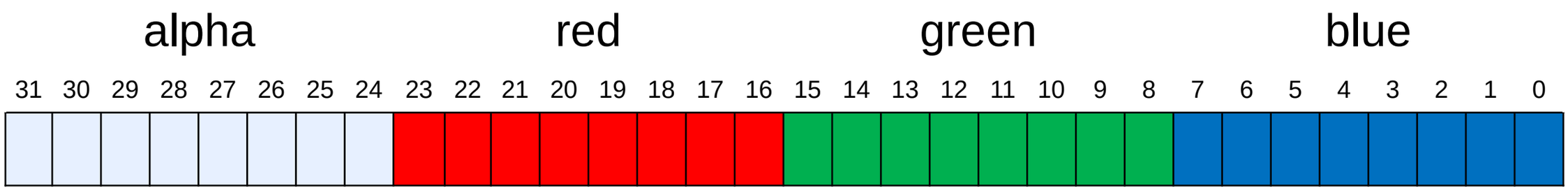
b

^	0	1
0	0	1
1	1	0

b ^ 0 = b

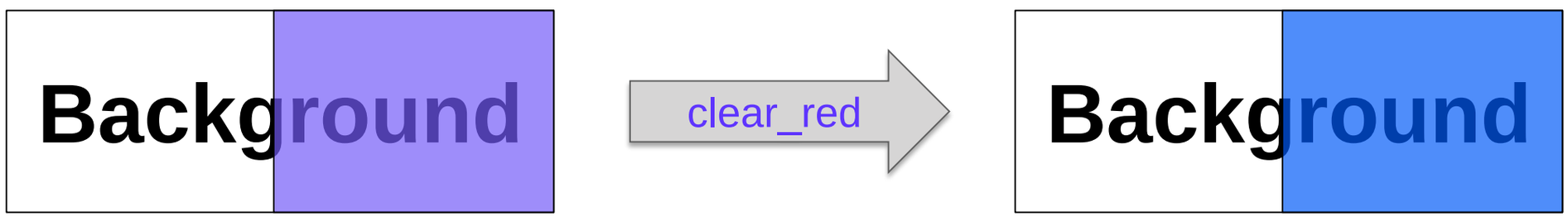
b ^ 1 = ~b

Clearing Bits

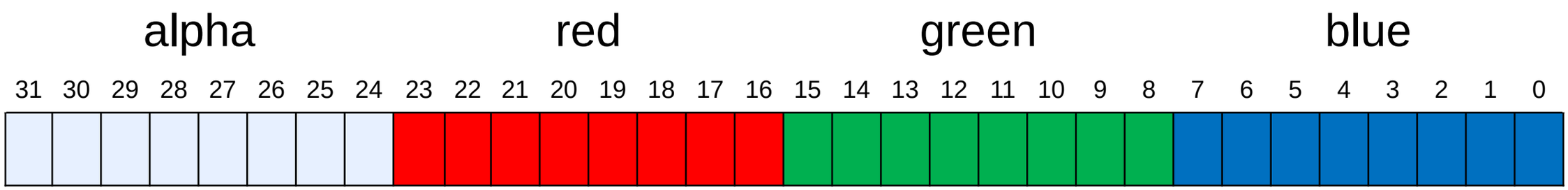



```
int clear_red(int p) {
  return p & 0xFF00FFFF;
}
```

Mask

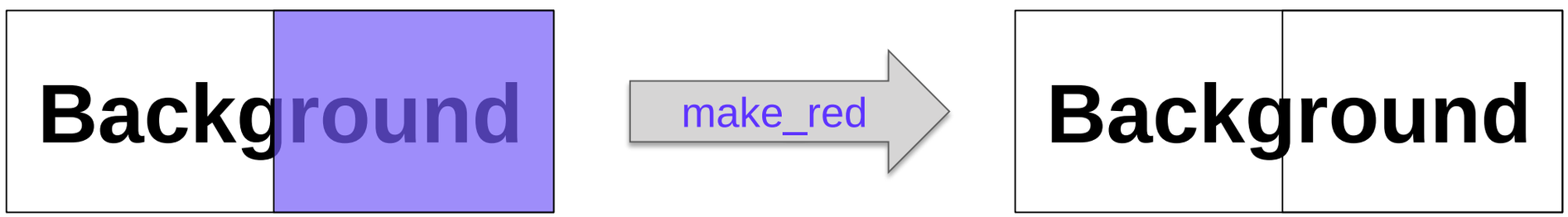


Isolating Red

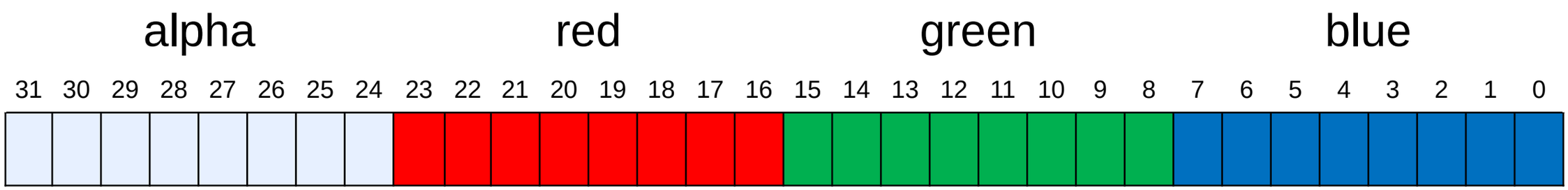



```
int make_red(int p) {
   int red = p & 0x00FF0000;
  return red;
}
```

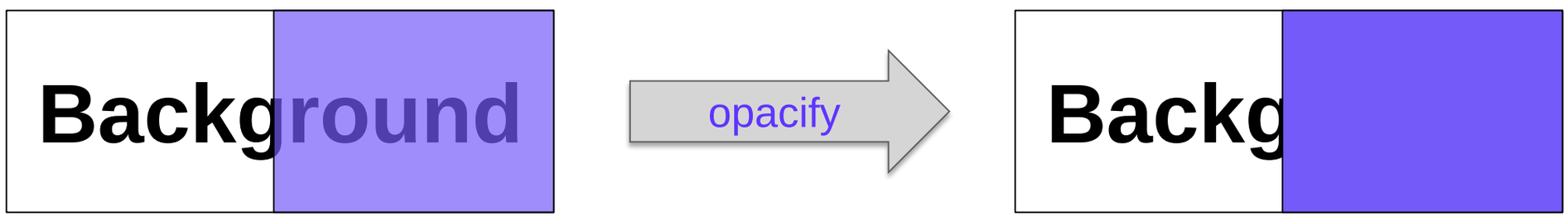
Mask



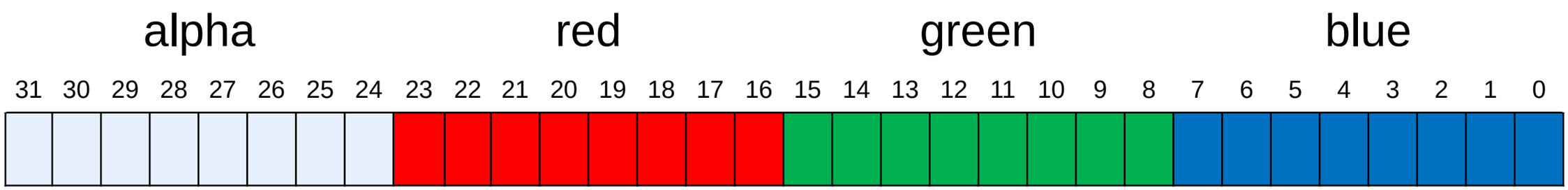
Example: Opacify




```
int opacify(int p) {
return p | 0xFF000000;
}
```



What does this Function do?




```
int franken_pixel(int p, int q) {
    int p_green = p & 0x0000FF00;
    int q_others = q & 0xFFFF00FF;
   return p_green | q_others;
}
```

Shifts: Moving Bits Around

Left shift: x << k

- shifts x by k bits to the left
  - k leftmost bits are dropped
  - k rightmost bits are 0

- 0101 << 1 = 1010
- 0101 << 3 = 1000

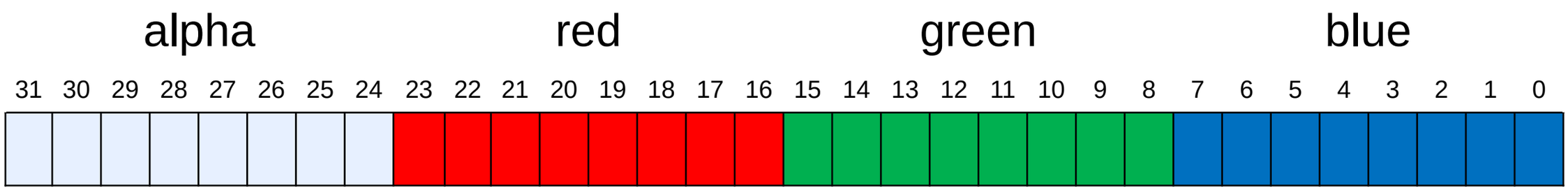
Right shift: x >> k

- shifts x by k bits to the right
  - k rightmost bits are dropped
  - k leftmost bits are a ***copy*** of the leftmost bit
    - **sign extension**

- 0101 >> 1 = 0010
- 0101 >> 3 = 0000
- 1010 >> 1 = 1101
- 1010 >> 3 = 1111

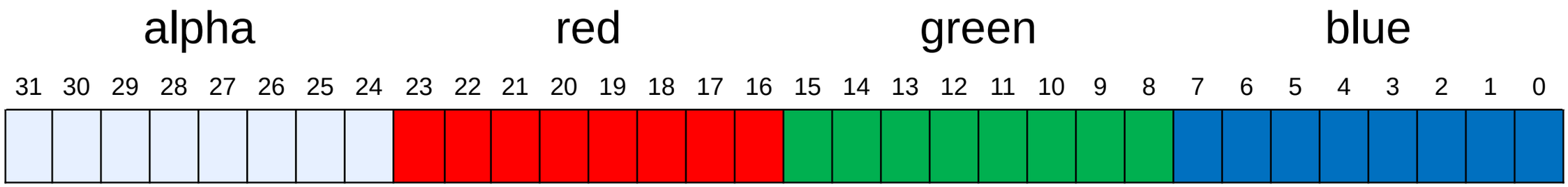
Preconditions: `//@requires 0 <= k && k < 32;`

Red Everywhere




```
int red_everywhere(int p) {
    int alpha = p & 0xFF000000;
    int red = p & 0x00FF0000;
   return alpha | red | (red >> 8) | (red >> 16);
}
```

Swapping the Alpha and Red Channels

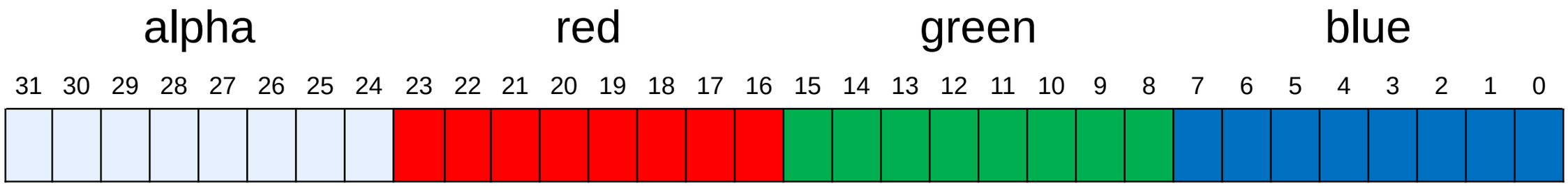



```
int BAD_swap_alpha_red(int p) {
  int new_alpha   = (p & 0x00FF0000) << 8;
  int new_red     = (p & 0xFF000000) >> 8;
  int old_green   = p & 0x0000FF00;
  int old_blue    = p & 0x000000FF;
 return new_alpha | new_red | old_green | old_blue;
}
```

What if the first bit is **1**?

Why is this function bad?

Swapping the Alpha and Red Channels




```
int swap_alpha_red(int p) {
   int new_alpha   = (p << 8) & 0xFF000000;
   int new_red     = (p >> 8) & 0x00FF0000; // fixed
   int old_green   = p & 0x0000FF00;
   int old_blue    = p & 0x000000FF;
 return new_alpha | new_red | old_green | old_blue;
}
```

Arithmetic vs. Bitwise Operations

- **NEVER** mix and match them
  - it does not make sense to multiply pixels
  - nor to & two numbers

- Few allowed exceptions
  - -x = ~x + 1
    - efficient way to compute the additive inverse
  - x << k = x * 2^k
  - x >> k = x divided by 2^k (*Python* division)