Integers

Number Representation

Representing Numbers

- We, people, have many ways to represent numbers

7

सात

sept

VII

seven

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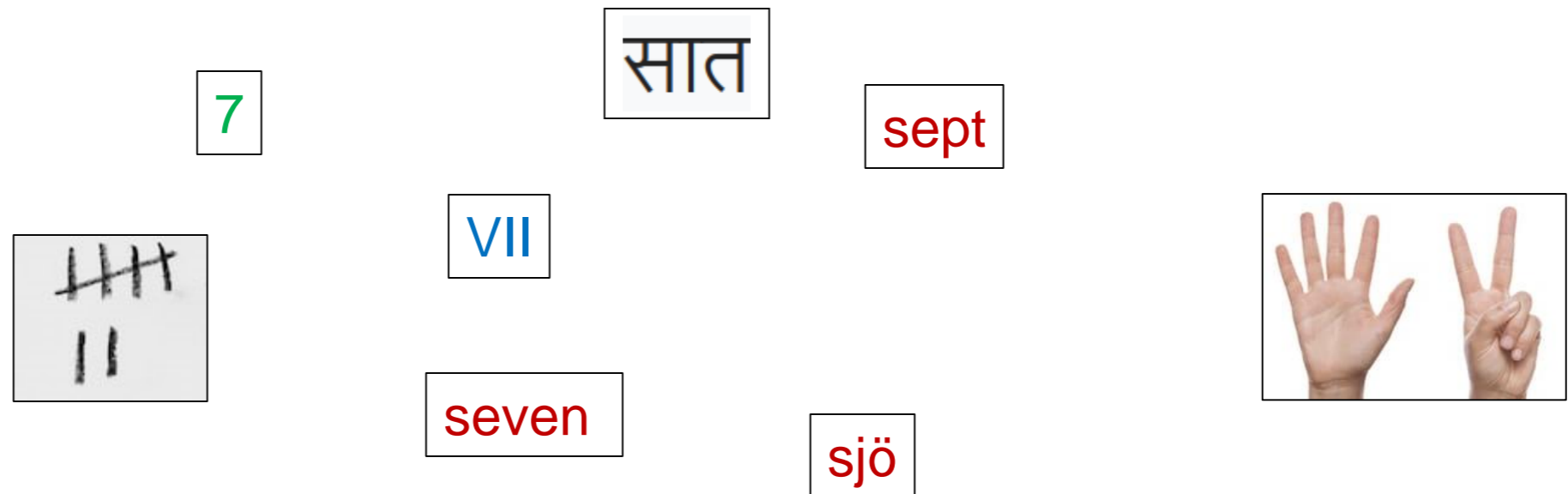
- They all express the same concept
  - ○ that some collection consists of *seven* things

2

Decimal Numbers

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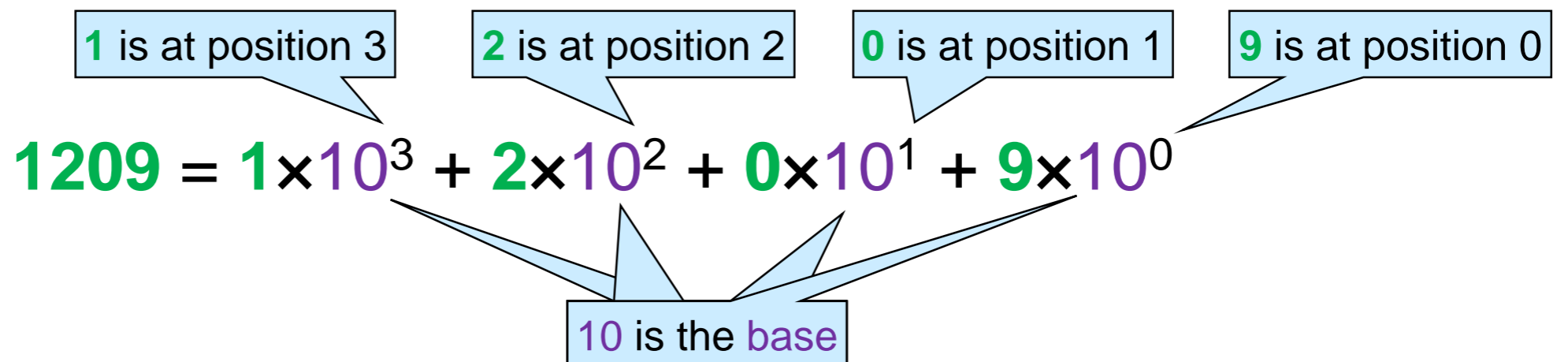   - they are the **digits**

    This comes from us having 10 fingers

  - "ten" is the **base**

- 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

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

10 is the base

Decimal Numbers

- *It uses ten symbols:*

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

  - *each represents a number between 0 and 9*

- Different languages use other symbols

	0	1	2	3	4	5	6	7	8	9
Arabic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩
Bengali	০	১	২	৩	৪	৫	৬	৭	৮	৯
Chinese (simple)	〇	一	二	三	四	五	六	七	八	九
Chinese (complex)	零	壹	貳	參	肆	伍	陸	柒	捌	玖
Chinese 花碼 (huā mǎ)	〇	〡	〢	〣	〤	〥	〦	〧	〨	〩
Devanagari	०	१	२	३	४	५	६	७	८	९
Ethiopic		፩	፪	፫	፬	፭	፮	፯	፰	፱
Gujarati	૦	૧	૨	૩	૪	૫	૬	૭	૮	૯
Gurmukhi	੦	੧	੨	੩	੪	੫	੬	੭	੮	੯
Kannada	೦	೧	೨	೩	೪	೫	೬	೭	೮	೯
Khmer	០	១	២	៣	៤	៥	៦	៧	៨	៩
Lao	໐	໑	໒	໓	໔	໕	໖	໗	໘	໙
Limbu	᥆	᥇	᥈	᥉	᥊	᥋	᥌	᥍	᥎	᥏
Malayalam	൦	൧	൨	൩	൪	൫	൬	൭	൮	൯
Mongolian	᠐	᠑	᠒	᠓	᠔	᠕	᠖	᠗	᠘	᠙
Myanmar	၀	၁	၂	၃	၄	၅	၆	၇	၈	၉
Oriya	୦	୧	୨	୩	୪	୫	୬	୭	୮	୯
Tamil	௦	௧	௨	௩	௪	௫	௬	௭	௮	௯
Telugu	౦	౧	౨	౩	౪	౫	౬	౭	౮	౯
Thai	๐	๑	๒	๓	๔	๕	๖	๗	๘	๙
Tibetan	༠	༡	༢	༣	༤	༥	༦	༧	༨	༩
Urdu	٠	١	٢	٣	۴	۵	۶	٧	٨	٩

4

Decimal Numbers

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

1   1   ⬅ carry

```
      1 2 0 9
  +   9 5 1 7
 _____
    1 0 7 2 6
```

> We used our 10 fingers for that

> 9+7 = 6 with a carry of 1

  - **multiplication** is done as iterated additions

```
        1 2 0 9
  ×       4 0 2
 _____
        2 4 1 8
            0
  +     4 8 3 6
 _____
      4 8 6 0 1 8
```

Binary Numbers

● Computers have *one* way to represent information: **binary**

○ they use two symbols, 0 and 1

- 1 = on
- 0 = off

● 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

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

2 is the base

Binary Numbers

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

$$1111 \quad \leftarrow \text{carry}$$

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

This works **exactly** as with decimal numbers

```
   1111    ← carry
   11011
 +  1110
 ‾‾‾‾‾‾‾‾
  101001
```

  - **multiplication** is done as iterated additions

```
      1010
  ×    101
 ‾‾‾‾‾‾‾‾‾‾
      1010
         0
 +    1010
 ‾‾‾‾‾‾‾‾‾‾
    110010
```

Converting Binary Numbers to Decimal

- Simply use the positional formula and carry out the calculation in decimal

$$100101_{[2]} = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 32 \quad + 0 \quad + 0 \quad + 4 \quad + 0 \quad + 1$$
$$= 37_{[10]}$$

Base

- Alternatively, use *Horner's rule*:

$$100101_{[2]} = (((((1 \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1$$
$$= \qquad\qquad (((2 \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1$$
$$= \qquad\qquad\qquad ((4 \times 2 + 1) \times 2 + 0) \times 2 + 1$$
$$= \qquad\qquad\qquad\qquad (9 \times 2 + 0) \times 2 + 1$$
$$= \qquad\qquad\qquad\qquad\qquad 18 \times 2 + 1$$
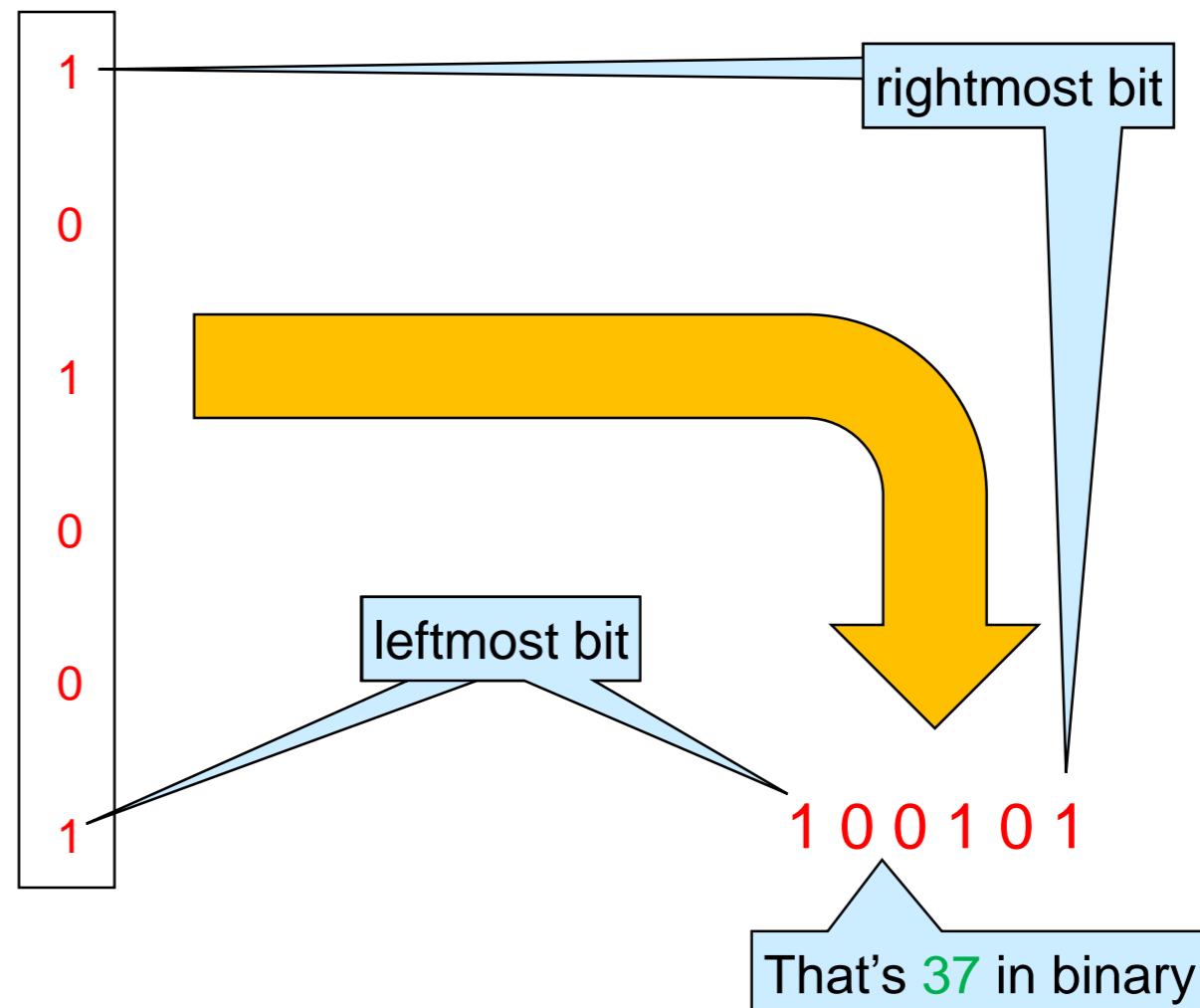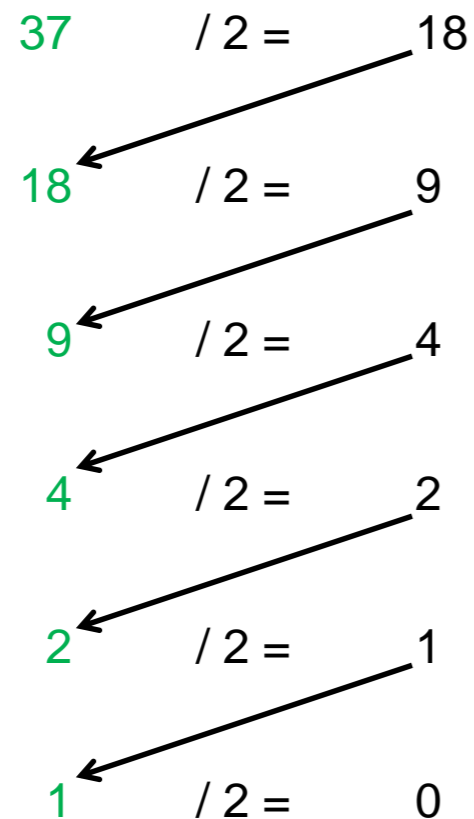$$= 37_{[10]}$$

That's because
$$1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (((((1 \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1$$

Converting Decimal Numbers to Binary

- Repeatedly divide the number by 2, harvesting the remainder, until we reach 0
  - the remainder is either 0 or 1
  - the binary representation comes out from right to left

… divided by 2   is …   with remainder …

37	/ 2 =	18		1
18	/ 2 =	9		0
9	/ 2 =	4		1
4	/ 2 =	2		0
2	/ 2 =	1		0
1	/ 2 =	0		1

rightmost bit

leftmost bit

1 0 0 1 0 1

That's 37 in binary

9

Hexadecimal Numbers

- Binary is fine for computers, but unwieldy for people

  11000000111111111101110

  ➢ hard to remember

  ➢ hard to communicate

- The **hexadecimal representation** makes things simpler

  ○ it uses 16 symbols: the numbers 0 to 9 and the letters A to F

  ➢ each represents a number between 0 and 15

  ➢ they are the **hex digits**

The
decimal to binary to hexadecimal
conversion table
(0 to 15)

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

Hexadecimal Numbers

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

- **1 hex digit** corresponds to **4 bits**
  - and vice versa
- **This makes converting between hex and binary very simple**
  - **hex** to **binary**: replace each **hex digit** with the corresponding **4 bits**
  - **binary** to **hex**: replace each group of **4 bits** with the corresponding **hex digit**

1100	0000	1111	1111	1110	1110
C	0	F	F	E	E

  - People find it a lot simpler to remember and communicate binary information in hexadecimal
    - and not just numbers

Not all hex words are this cute, though!

Hexadecimal Numbers

- Any number has a positional representation in hex as a sequence of hex digits
  - the **position** *i* of a hex digit *h* indicates its importance
    - it contributes *h×16^i* to the value of the number
  - the value of the number is the sum of the contribution of each position

$$\textbf{C0FFEE} = \textbf{C}×16^5 + \textbf{0}×16^4 + \textbf{F}×16^3 + \textbf{F}×16^2 + \textbf{E}×16^1 + \textbf{E}×16^0$$

After plugging in 12 for C, etc, that's 12648430 in decimal

- We can also do arithmetic in hex
  - but hex is primarily used to represent two types of non-numerical data
    - memory addresses ——— next lecture
    - bit patterns ——— later in this lecture

Numbers in C0

- All numbers in C0 have type int

- We can enter numbers in C0
  - in decimal
  - in hexadecimal
    - by prefixing them with **0x**

- Internally, it stores them in binary
  - but there is no way to enter numbers in binary

- C0 always prints numbers back to us in decimal

When we enter C0FFEE in hex ..

… coin responds it's 12648430 in decimal

C0FFEE and 12648430 are two different ways of entering the **same number**

Linux Terminal

coin
C0 interpreter (coin) …
…
--> 0xC0FFEE;
12648430 (int)
--> 0xC0FFEE == 12648430;
true (bool)

Numbers in C0

- *C0 always prints numbers back in decimal*

- Use the function int2hex in the <util> library to display a number in hexadecimal
  - as a string, not an int

Loads the <util> library when starting coin
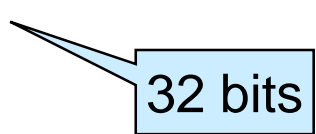
Linux Terminal

coin -l util
C0 interpreter (coin) …
…
--> int2hex(0xC0FFEE);
"00C0FFEE" (string)
--> int2hex(12648430);
"00C0FFEE" (string)

There is no int2bin

You can write your own!

14

Fixed-size Number Representation

Machine Words

- Computers store and manipulate binary data
  - *everything is a bit in a computer*

- Computer hardware processes batches of k bits in parallel
  - a batch of k bits is called a **machine word**
  - nowadays, a typical value of k is 32

- Computation is very **efficient** on whole words
  - but less so on parts of words

- Most programming languages use a word to represent an int
  - **in C0, an int is always 32 bits long**
    - internally, 37 is not represented as 100101
      but as 00000000000000000000000000100101

      32 bits

Fixed-size Numbers

- A k-bit computer uses **exactly k bits** to represent an int

  That's a computer whose words are k bits long

- In our discussion, we will assume that k = 4

  This will simplify our examples

  ➢ but in C0, an int is always 32 bits long

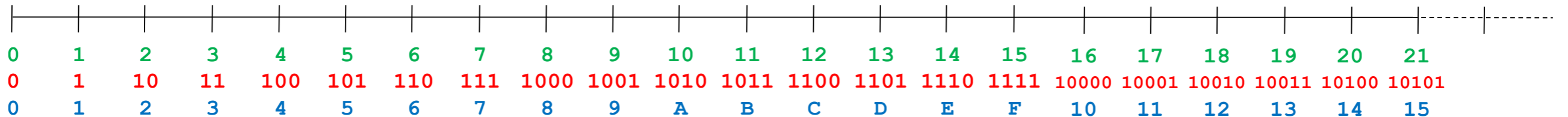- In a 4-bit computer, 6 is not represented as 110 but as 0110
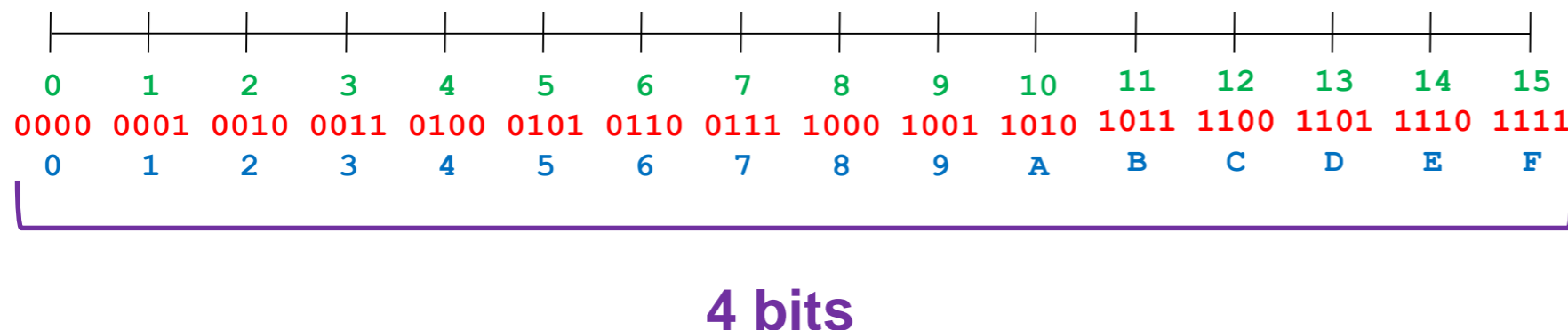
  ➢ Numbers have a *fixed-size* in a computer

  4 bits

Numbers in Math vs. in a Computer

- **In math, there are infinitely many numbers**
  - we visualize them as an infinite **number line**

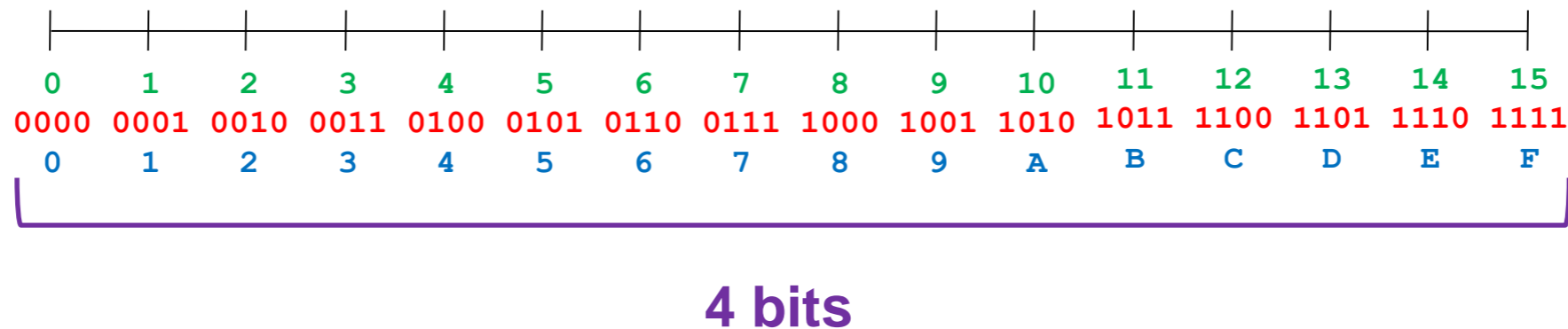> The beginning of the number line with numbers in decimal and binary and hex

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000	10001	10010	10011	10100	10101
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15

- **In a 4-bit computer, there are finitely many numbers**
  - exactly $16 = 2^4$
  - the line is *finite*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

4 bits

- On a k-bit computer, we can represent only 2^k distinct numbers
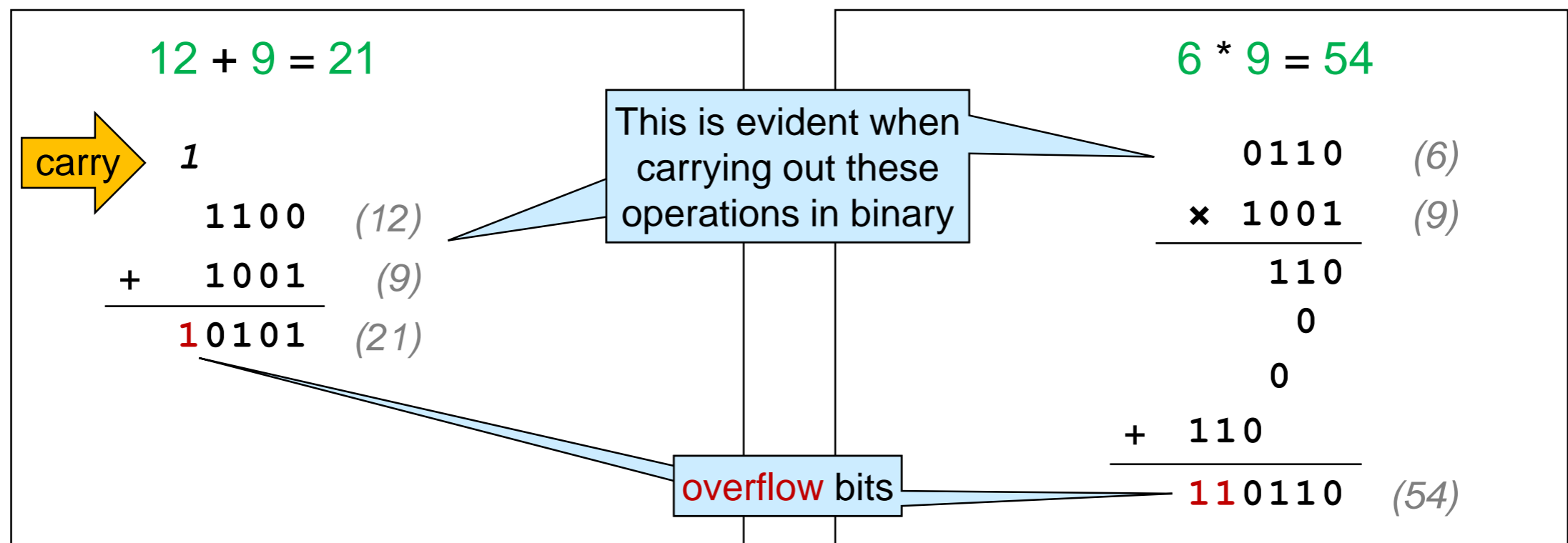 - C0 can represent only 2^{32} distinct numbers

18

Numbers in a Computer

- *In a 4-bit computer, we can represent only 2^4 distinct numbers*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

4 bits

- We cannot represent numbers larger than what fits in 4 bits
  - e.g., 21
    - in binary it's 10101, but that requires 5 bits

- Even if we avoid writing larger numbers in a program, they may emerge during computation
  - intermediate results need to be stored in a word in memory!

Overflow

- The result of adding two int's may not fit into a k bits word
  - it may be a k+1 bit number!
  - the result may be even longer when multiplying two int's

12 + 9 = 21

6 * 9 = 54

```
carry    1

      1100    (12)
   +  1001    (9)
      10101   (21)
```

This is evident when carrying out these operations in binary

```
        0110    (6)
     ×  1001    (9)
         110
        0
       0
    +  110
      110110    (54)
```

overflow bits

4-bit examples

- We have an **overflow** when the result of an operation doesn't fit in a machine word
  - k bit operands, but the result has more than k bits

How to Deal with Overflow?

- The result of an operation does not fit into a k-bit word

4 bits

These numbers
are **not representable**
with 4 bits
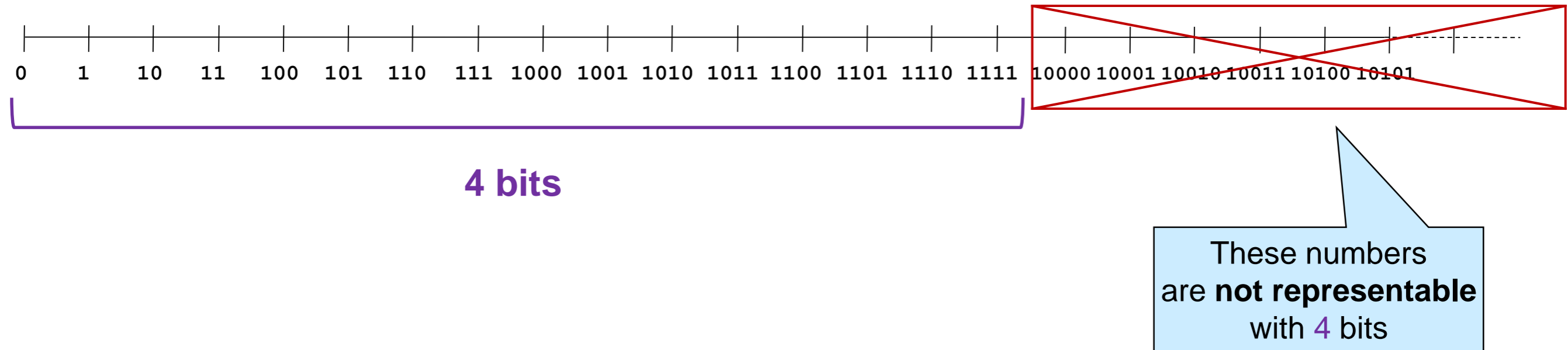
- Two common approaches to handling overflow
  1. Raise an error or an exception
     - an *error* aborts the program
     - an *exception* is an error that can be handled to continue computation
  2. Continue execution in some meaningful way

Handling Overflow as Error

- Signaling an error is not always the right thing to do
  - The **Ariane 5** rocket exploded on its first launch because an unexpected overflow raised an unhandled exception

```
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(TDB
end if;
P_M_DERIVE(T_ALG.E_BH) :=
    UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LS
```

Handling Overflow as Error

- Treating overflows as errors makes it **hard to write correct code** involving ints
  - hard to debug
  - hard to reason about

- Example
  - *n + (n - n)* and *(n + n) - n* are equal **in math**
  - but with fixed size numbers, they may yield different outcomes
    - ➢ *n + (n - n)* is **always** equal to *n*
    - ➢ *(n + n) - n* may overflow

  Writing one or the other is the same

  Writing one or the other is **not** the same; although it feels like it is

- People instinctively use math when writing code
  - we want the laws of arithmetic to hold
    - ➢ whenever possible

Modular Arithmetic

Continuing Computation on Overflow

- Instead of aborting execution, just **ignore the overflow bits**

$12 + 9 = \ldots$

carry → 1

```
  1100     (12)
+ 1001     (9)
 10101    (21  5)
```

overflow bits

$6 * 9 = \ldots$

```
    0110     (6)
  × 1001     (9)
    110
    0
    0
+ 110
 110110    (54  6)
```

4-bit examples
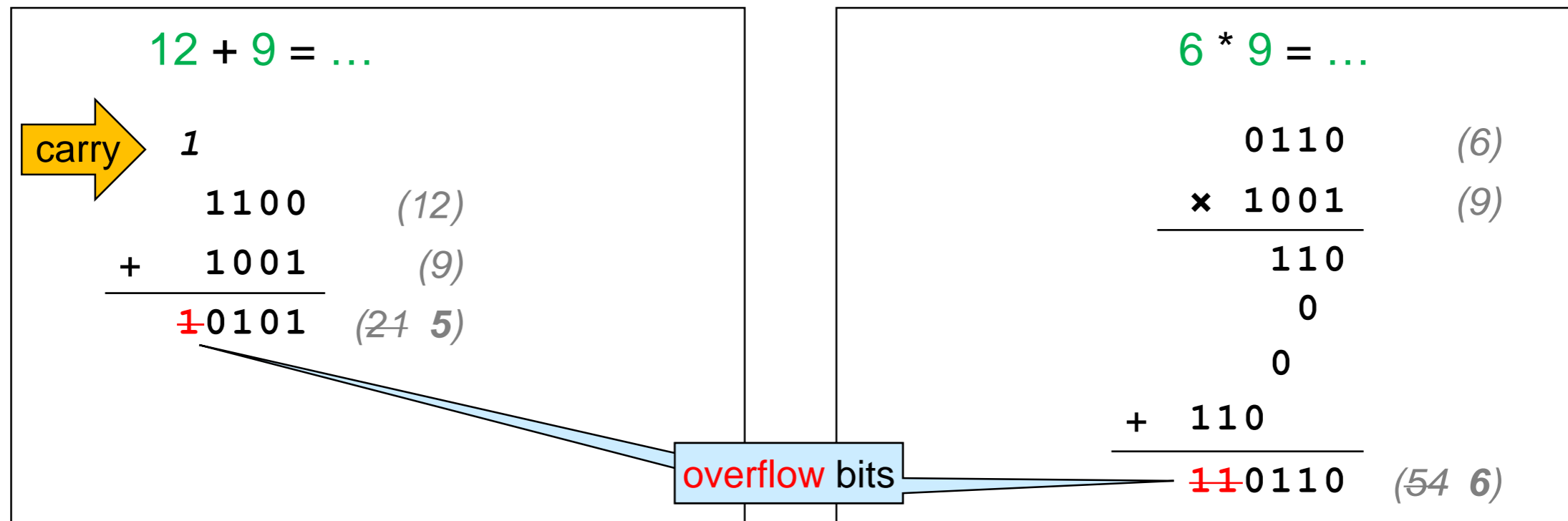
- The result of the operation is what fits in the word

$\ldots = 5$

$\ldots = 6$

- This is **not** the correct mathematical value
  - *but does it relate to it in any way?*

Ignoring the Overflow Bits

```
        1
      1100      (12)
 +    1001      (9)
    ̶10101     (2̶1̶  5)
```

○ Throwing out the overflow bit amounts to subtracting **10000** from the result

➢ that's 16 in decimal

10101 − 10000 = 0101

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 ̶10000 ̶10001 ̶10010 ̶10011 ̶10100 ̶10101

4 bits
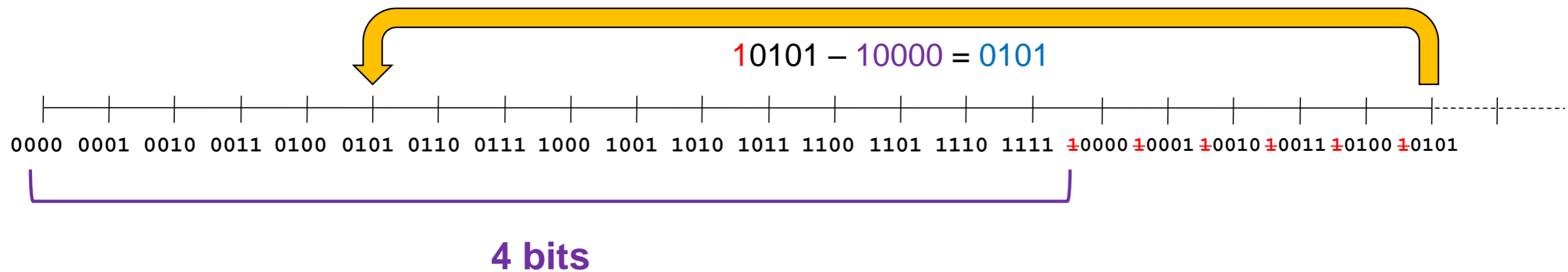
○ Note that 16 is 2^4

➢ 4 is how many bits our words have

Ignoring the Overflow Bits

```
      0110      (6)
  ×   1001      (9)
  ─────────
      110
       0
      0
+   110
  ─────────
   110110      (54 6)
```

○ Throwing out the overflow bits amounts to subtracting a *multiple* of **10000** from the result

   ➢ that's 16 in decimal

110110 − 11 * 10000 = 0110

```
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 10000 10001 10010 10011 10100 10101
```
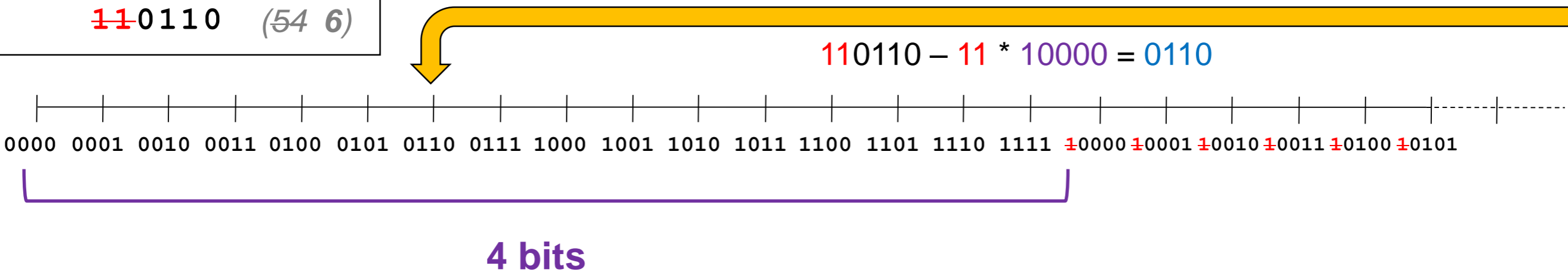
4 bits

● In general, we subtract as many multiples of 16 (= 2^4) as necessary so that the result fits in 4 bits

● Ignoring the overflow bits computes the result **modulo 16**

Computing Modulo *n*

- Evaluate an expression normally but return the remainder of dividing it by *n*
  - ➤ a number between 0 and *n-1*
  - ○ $12 + 9 =_{\text{mod } 16} 5$
  - ○ $9 * 6 =_{\text{mod } 16} 6$

- This is called **modular arithmetic**

- Modular arithmetic works just like traditional arithmetic

Modular Arithmetic

- Modular arithmetic obeys the **same laws** as traditional arithmetic
  - ➢ *for expressions involving + and * so far*

$x + y =_{\text{mod } n} y + x$	Commutativity of addition
$(x + y) + z =_{\text{mod } n} x + (y + z)$	Associativity of addition
$x + 0 =_{\text{mod } n} x$	Additive unit
$x * y =_{\text{mod } n} y * x$	Commutativity of multiplication
$(x * y) * z =_{\text{mod } n} x * (y * z)$	Associativity of multiplication
$x * 1 =_{\text{mod } n} x$	Multiplicative unit
$x * (y + z) =_{\text{mod } n} x * y + x * z$	Distributivity
$x * 0 =_{\text{mod } n} 0$	Annihilation

- We use these laws *implicitly* every time we do arithmetic
  - ○ in particular when writing programs

Handling Overflow in C0

- C0 discards overflow bits
  - **C0 handles overflow using modular arithmetic**
  - numerical expressions are computed modulo 2^{32}
    - because C0 assumes 32-bit words

- This makes it easy to reason about programs
  - modular arithmetic works like traditional arithmetic
    - we apply it innately
  - there is no need to consider special cases for overflow
    - for expressions using + and * so far

Overflow does <u>not</u> abort computation in C0

Reasoning about int Code

- This function always returns "Good"

```
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

(modulo 2^{32})

- We don't need to worry about 1+x or x+1 overflowing
  - they may, but that doesn't matter
    - overflow doesn't abort computation
    - the laws of (modular) arithmetic tell us they always evaluate to the same value

What does Computing Modulo *n* Mean?

- *Rather than viewing the numbers as lying on an infinite line,* we think of them as **wrapping around a circle** with *n* positions
  - ○ values that are equal modulo *n* share the same position

Example for $n = 16$

This position corresponds to 10, 26, 42, 58, 74, 90, 106, …

16

31   17

15   0   1
1111 0000 0001   18

30

14   2
1110   0010

29   19

13   3
1101   0011

28   4   20
12   0100
1100

11   5
1011   0101

27   21

10   6
1010   0110

26   9   8   7   22
1001 1000 0111

25   23

24
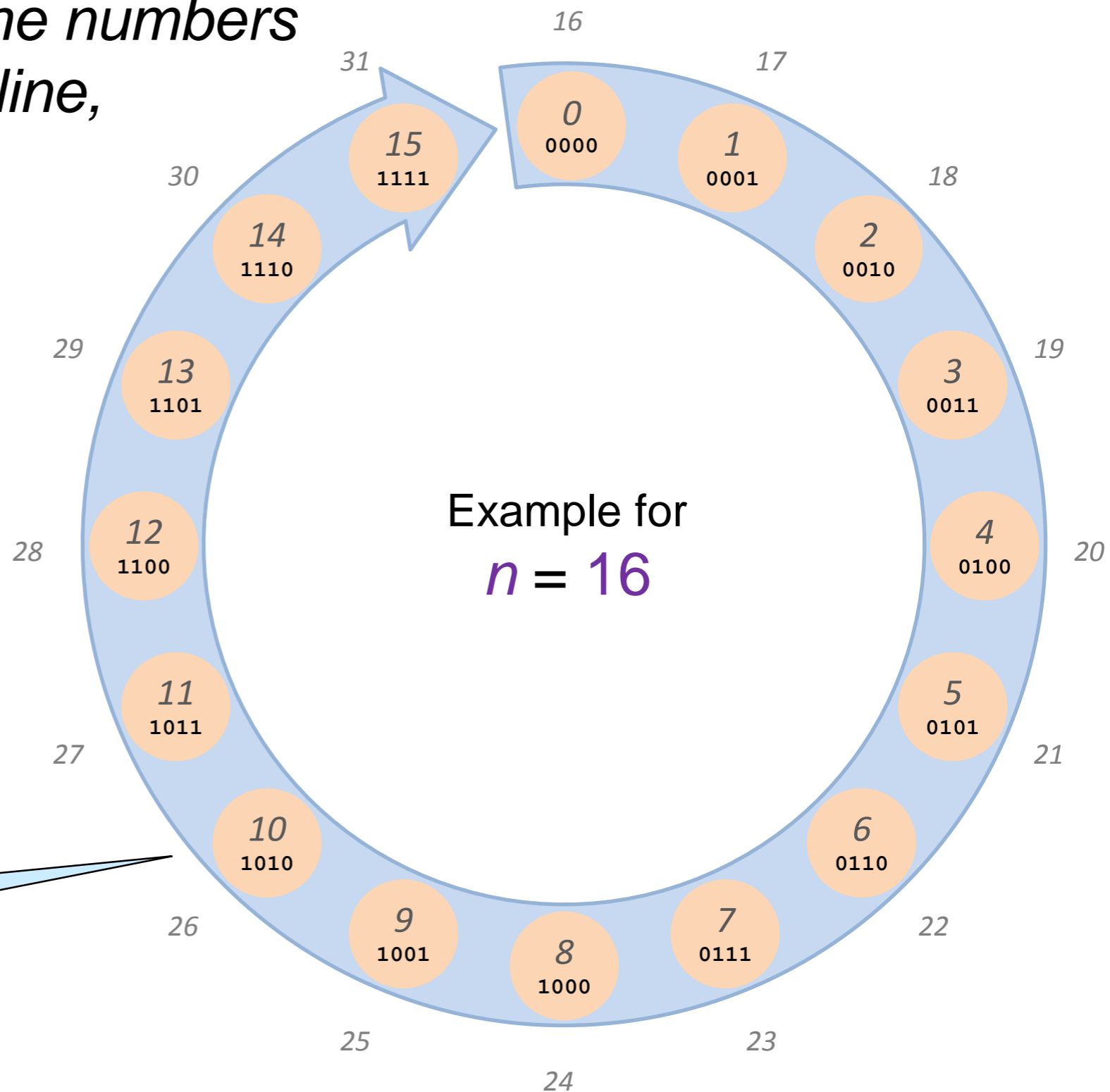
What does Computing Modulo *n* Mean?

- We carry out computations normally but return the position of the result on the circle

  ○ 12 + 9 =$_{\text{mod 16}}$ 5

  ○ 9 * 6 =$_{\text{mod 16}}$ 6

- Then, addition corresponds to *moving clockwise* around the circle

  ○ to compute 12 + 9 start from 12 and step 9 times clockwise

12 + 9 lands **here** because 21 is 5 mod 16

and 9 * 6 lands **here** because 54 is 6 mod 16
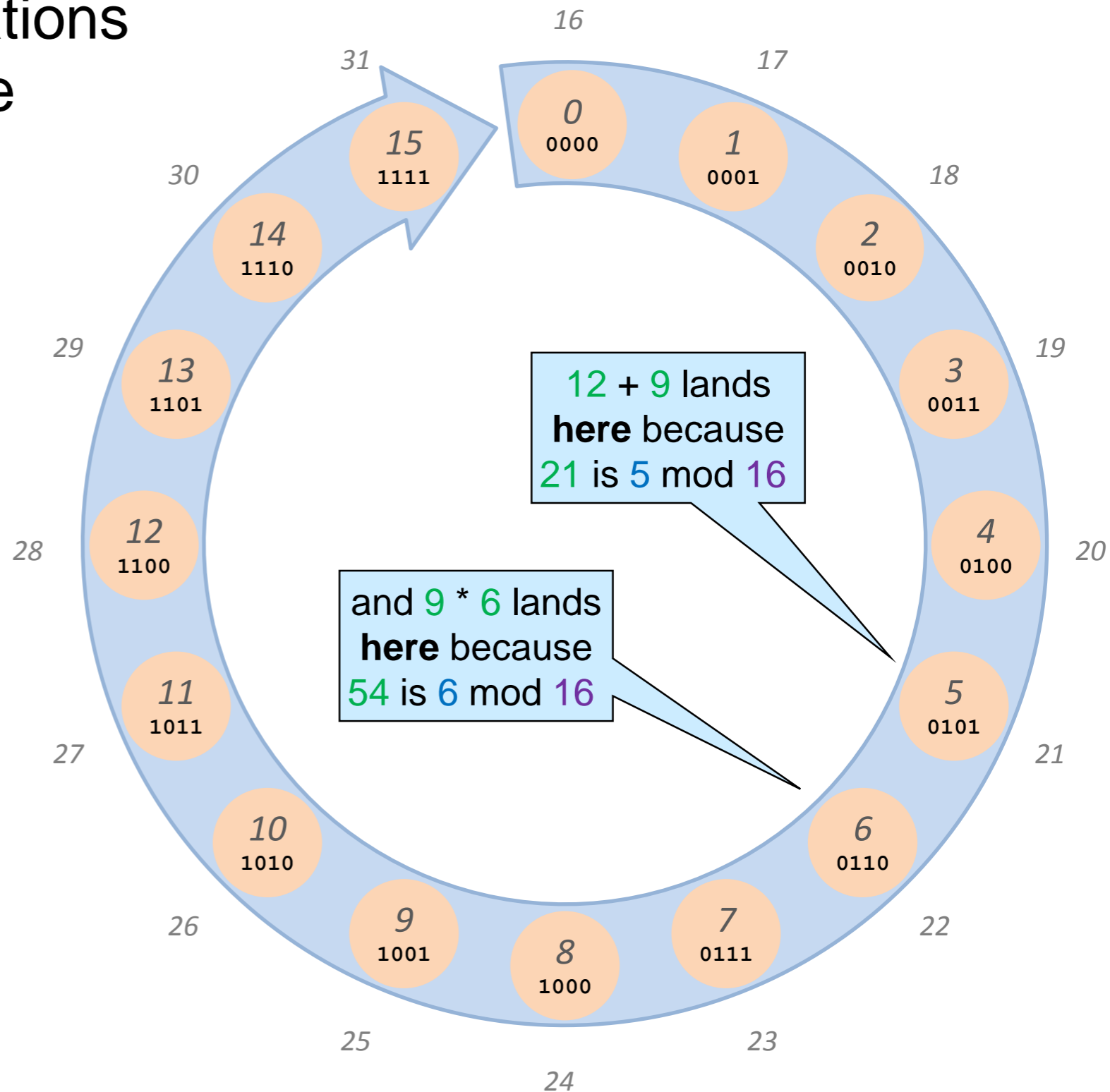
What about the Negatives?

- The negative numbers too wrap around the circle
  - -1 =$_{mod\ 16}$ 15
  - -6 =$_{mod\ 16}$ 10

Example for
$n = 16$
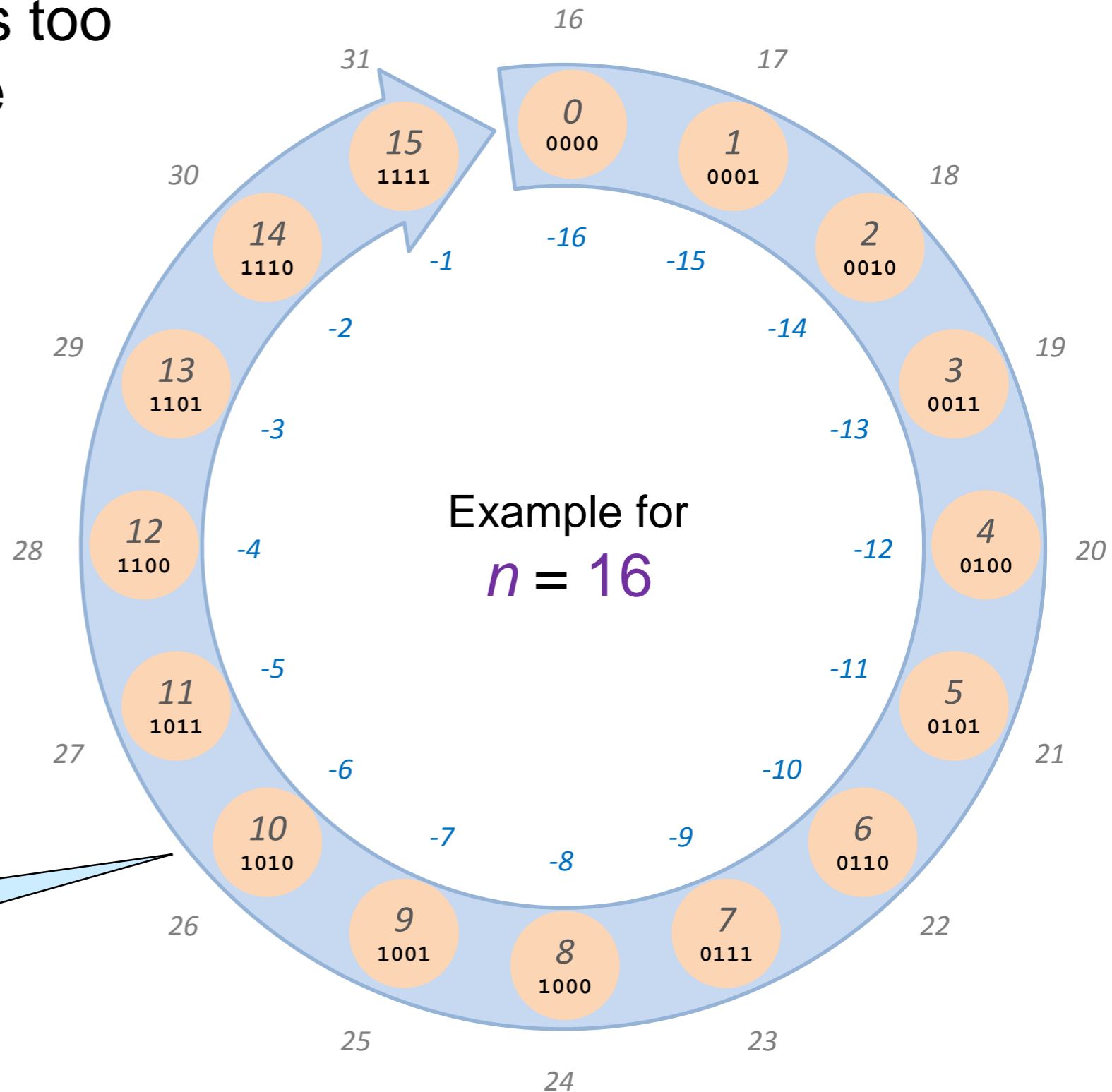
This position corresponds to …, -86, -70, -54, -38, -22, -6, 10, 26, 42, 58, 74, 90, 106, …

Subtraction modulo *n*

- We can then do subtraction modulo *n*
  - $5 - 7 =_{mod\ 16} 14$
    - We evaluate it normally but return the remainder of dividing it by *n*
    - Equivalently, return the position of the result on the circle

- x - y is stepping y times *counter-clockwise* from x
  - to compute 5 - 7 start from 5 and step 7 times counter-clockwise

5 - 7 lands **here** because -2 is 14 mod 16

Subtraction modulo *n*

- With subtraction, we can define the *additive inverse -x* of any number x

  ➢ the number that added to x yields 0

  $$-x =_{\text{mod } n} 0 - x$$

- Then, more laws of traditional arithmetic are valid in modular arithmetic

$x + (-x) =_{\text{mod } n} 0$	Additive inverse
$-(-x) =_{\text{mod } n} x$	Cancelation

  o More programs behave as if we were using normal arithmetic

  ➢ even in the presence of overflows

Reasoning about int Code

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

- This function always returns "Good"
  - ○ x + x - x = x in normal arithmetic
  - ○ so x + x - x == x in C0

- If the compiler understands x + x - x
  - ○ as x + (x - x), then
    - ➢ x + (x - x) = x + 0   by additive inverse
      -            = x   by additive unit
  - ○ as (x + x) - x, then
    - ➢ (x + x) - x = x + (x - x)   by associativity of +
      -             = x   as above

x + x may overflow
but it doesn't matter

37

Two's Complement

Printing Numbers

- Modular arithmetic tells us that many numbers correspond to the same bit sequence

  1110 could stand for
  …, -82, -66, -50, -34, -18, -2,
  14, 30, 46, 62, 78, 94, 110, …

  int x

- But what number should the computer print **1110** as?

  ○ 14?

  ○ -2?

  ○ 78?

  ○ …

  Say our program reaches
  printint(x);
  where x contains 1110
  (on a hypothetical 4-bit computer)
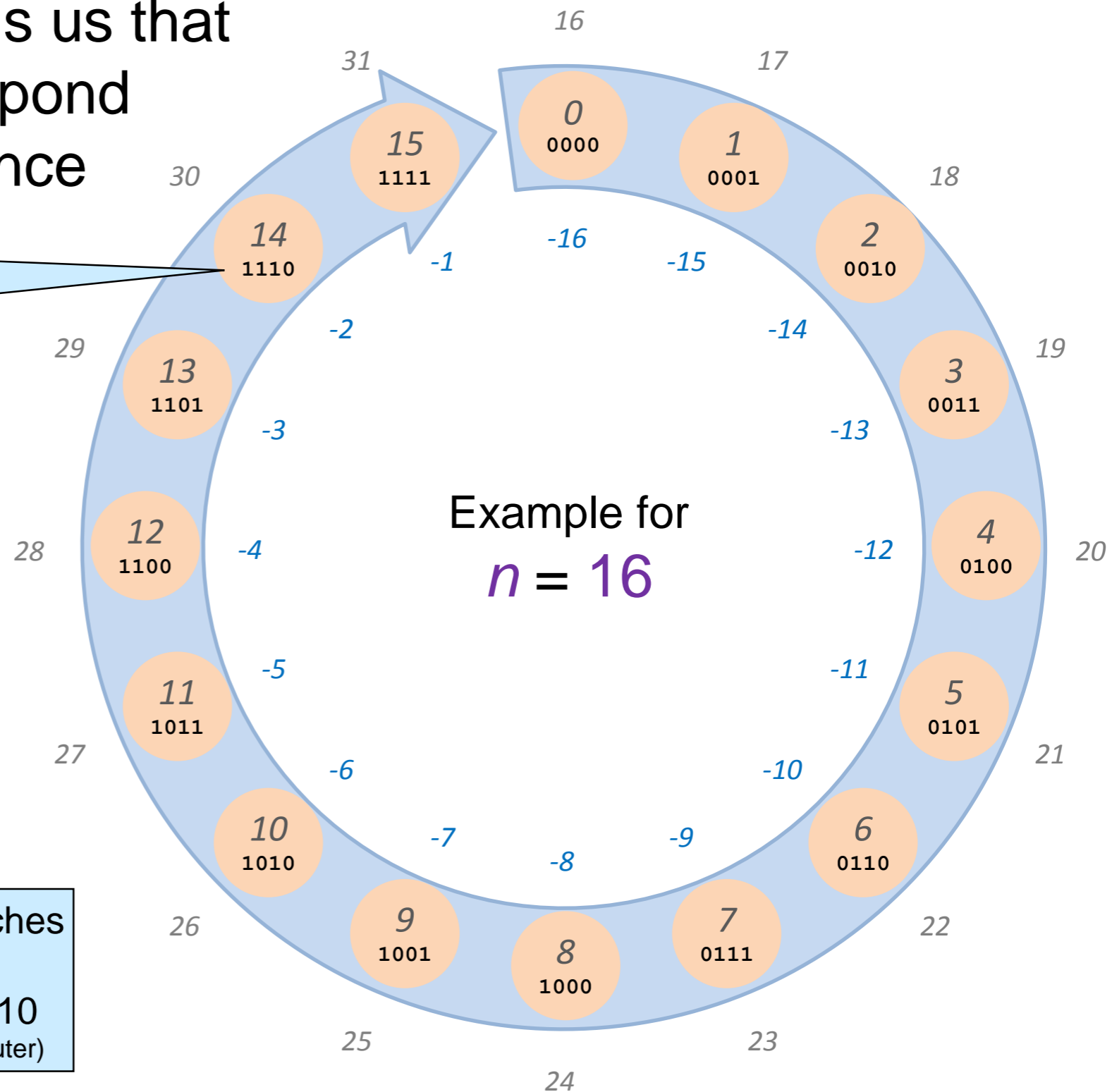
Example for
n = 16

Comparing Numbers

- Modular arithmetic tells us that many numbers correspond to the same bit sequence

  1110 could stand for
  …, -82, -66, -50, -34, -18, -2, 14, 30, 46, 62, 78, 94, 110, …

  int x

  0011 could stand for
  …, -77, -61, -45, -29, -13, 3, 19, 35, 51, 67, 83, 99, 115, …

  int y
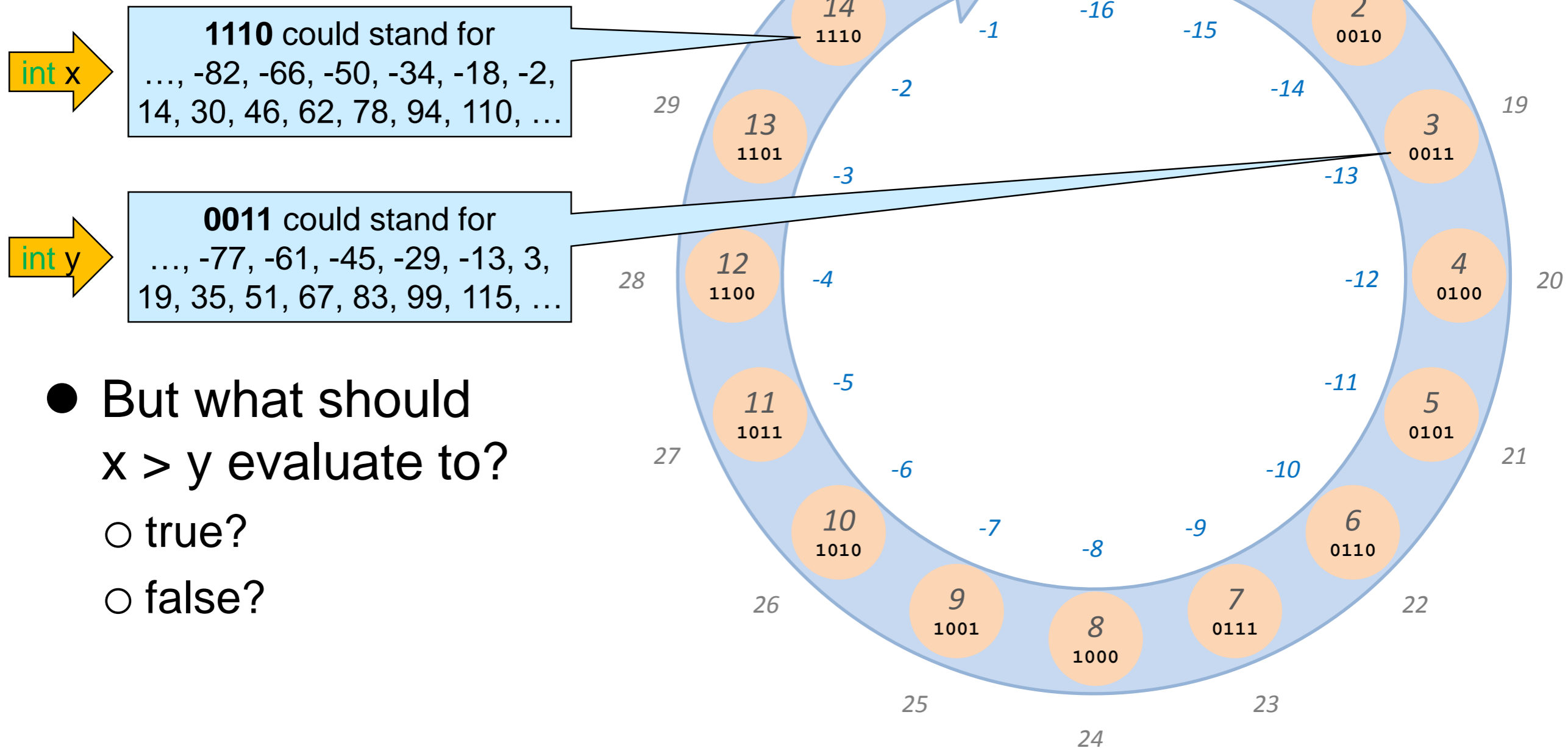
- But what should x > y evaluate to?
  - ○ true?
  - ○ false?

16
31
17
30
18
29
19
28
20
27
21
26
22
25
23
24

0
0000
15
1111
1
0001
14
1110
2
0010
-16
-15
-1
-2
-14
13
1101
3
0011
-3
-13
12
1100
4
0100
-4
-12
11
1011
5
0101
-5
-11
10
1010
6
0110
-6
-10
9
1001
8
1000
7
0111
-7
-8
-9

40

The Range of int's

- In both case, the computer needs to decide what number each k-bit word corresponds to

> This is the opposite of the earlier problem: *what k-bit word does each number correspond do*

- Common requirements
  - successive bit values should correspond to successive numbers
    - ❑ 16, 1, -14, … won't do
  - 0 should be one of them
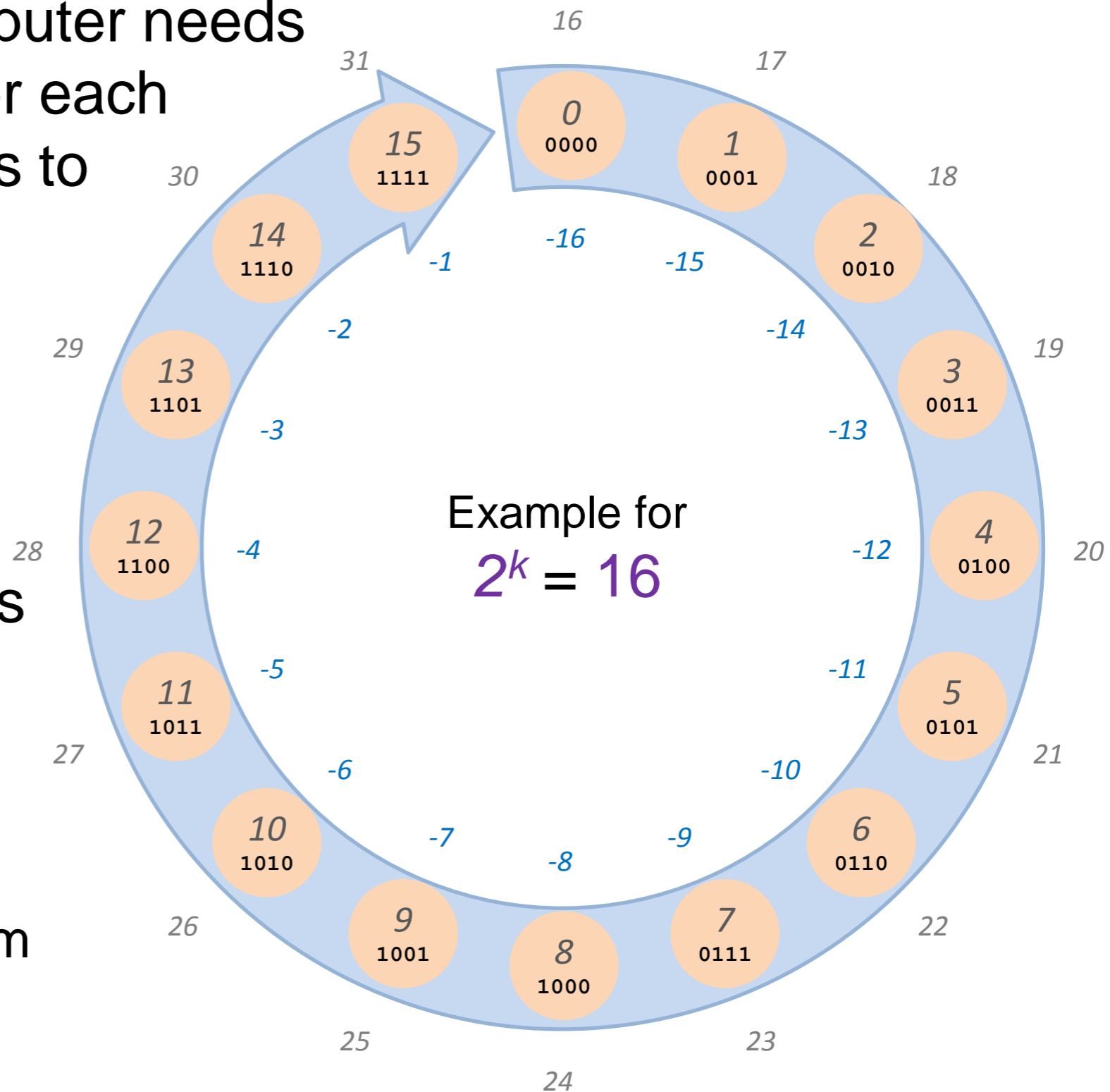
Example for
$2^k = 16$

The Range of int's

- *What number does each k-bit word correspond to?*
  - *successive bit values should correspond to successive numbers*
  - *0 should be one of them*

- Pick the first 2^k integers starting at 0
  - here 0, 1, … 15
    - **1110** is printed as 14
    - **1110** > **0011** returns true

- int's that behave this way are called **unsigned**
  - This is **not** how C0's int's work
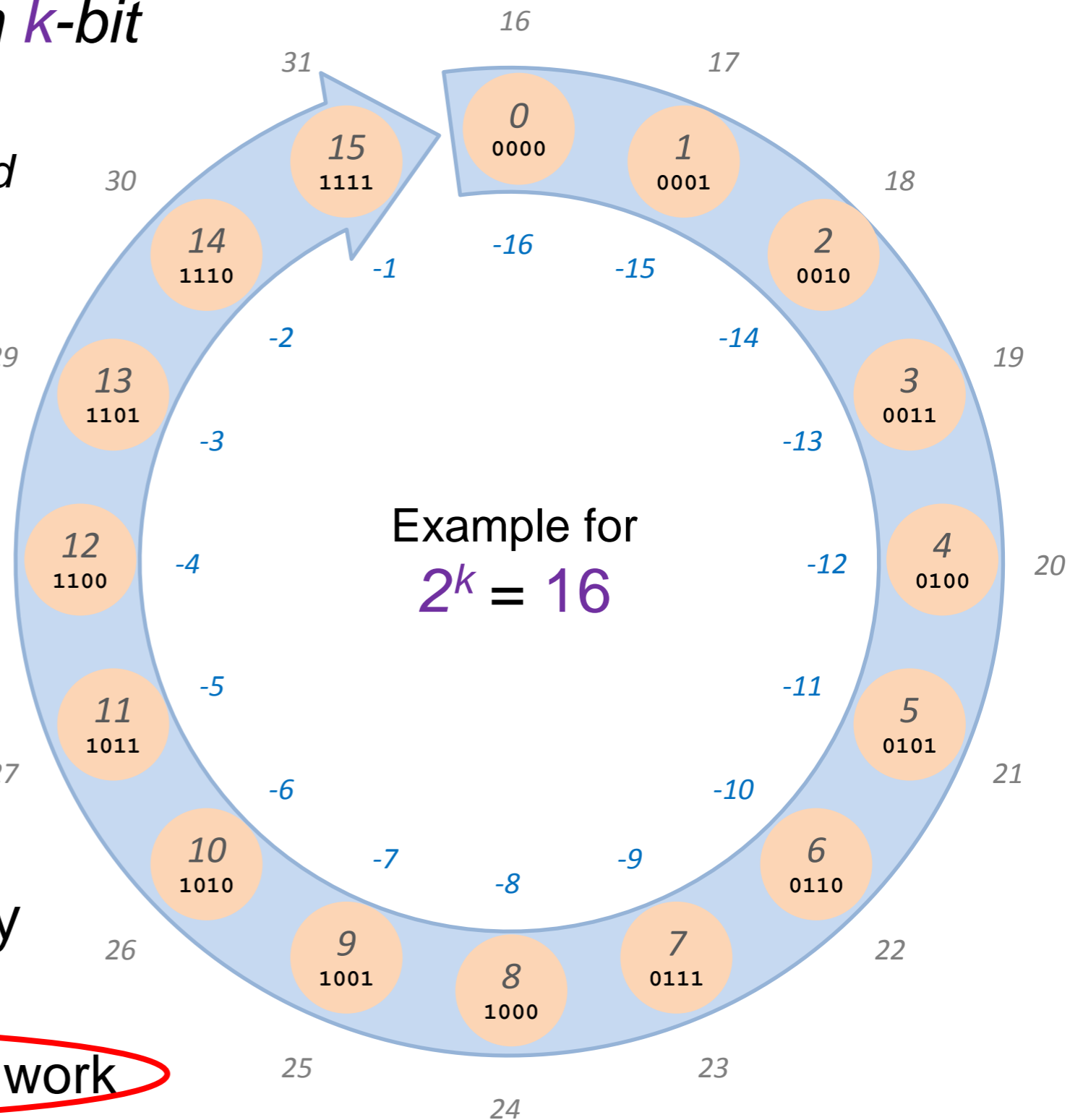
Example for
$2^k = 16$

16
31   17
15 0   1
1111 0000 0001   18
30
-16 -15
14   2
1110 -1   0010
-2   -14
29
13   3   19
1101 -3   -13 0011

12   4
1100 -4   -12 0100   20
28

-5   -11 5
11   0101   21
1011

-6   -10
10   6   22
1010 -7   -8 -9 0110
27

9   8   7
1001 1000 0111
26   22

25   23
24

42

The Range of int's

- *What number does each k-bit word correspond to?*
  - *successive bit values should correspond to successive numbers*
  - *0 should be one of them*

- We also want some negative numbers
  - about half

- One common option
  - Pick the range -2^{k-1} to $2^{k-1} - 1$
  - This choice is called **two's complement**
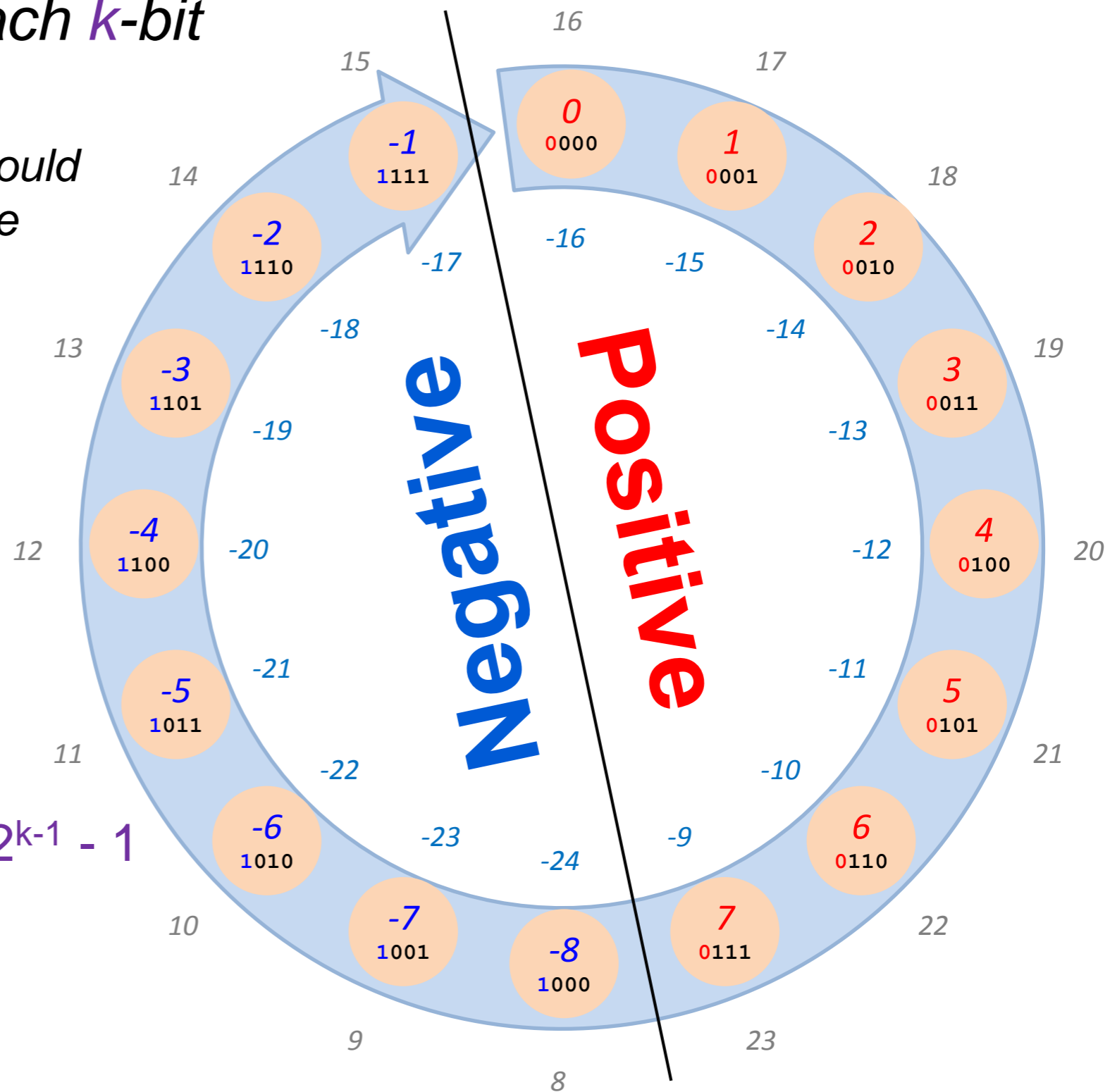
Two's Complement

- Each k-bit word corresponds to a number between -2^{k-1} and $2^{k-1} - 1$
  - the negative numbers go from -1 to -2^{k-1}
  - the positive numbers go from 1 to $2^{k-1} - 1$
  - and there is 0

- The leftmost bit tells the sign
  - 1 for negative numbers
  - 0 for positive numbers and 0

  It is called the **sign bit**
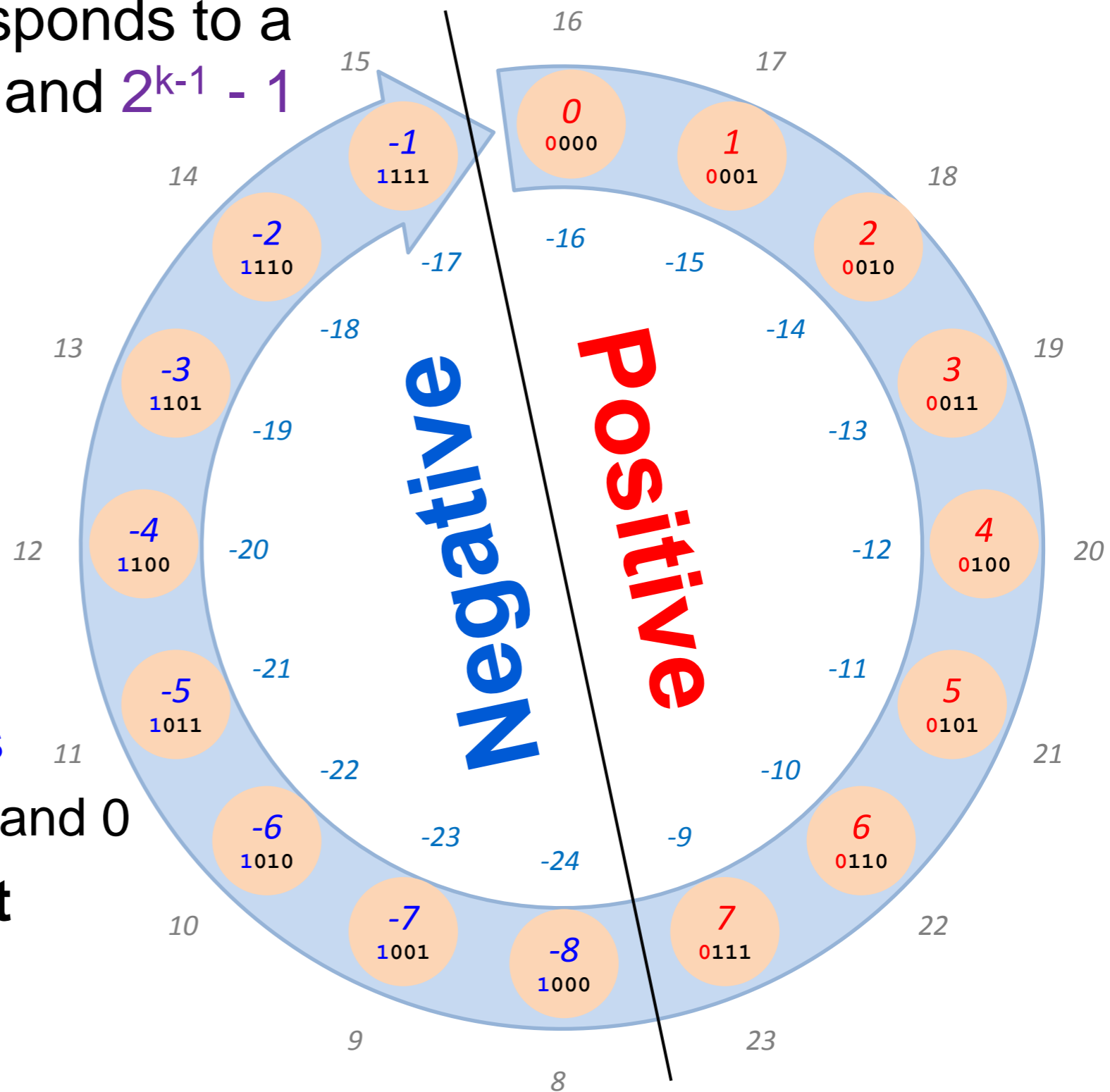
Efficient way to determine the sign of a number

Two's Complement

- Each k-bit word corresponds to a number in the range -2^{k-1} to $2^{k-1} - 1$

  - The *smallest number* is called **int_min**
    - -2^{k-1}
    - $1$00…000 in binary

  - The *largest number* is called **int_max**
    - $2^{k-1} - 1$
    - $0$11…111 in binary

  - Other notable numbers:
    - 0 is $0$00…000
    - -1 is $1$11…111

int_min = -2^3

int_max = $2^3 - 1$
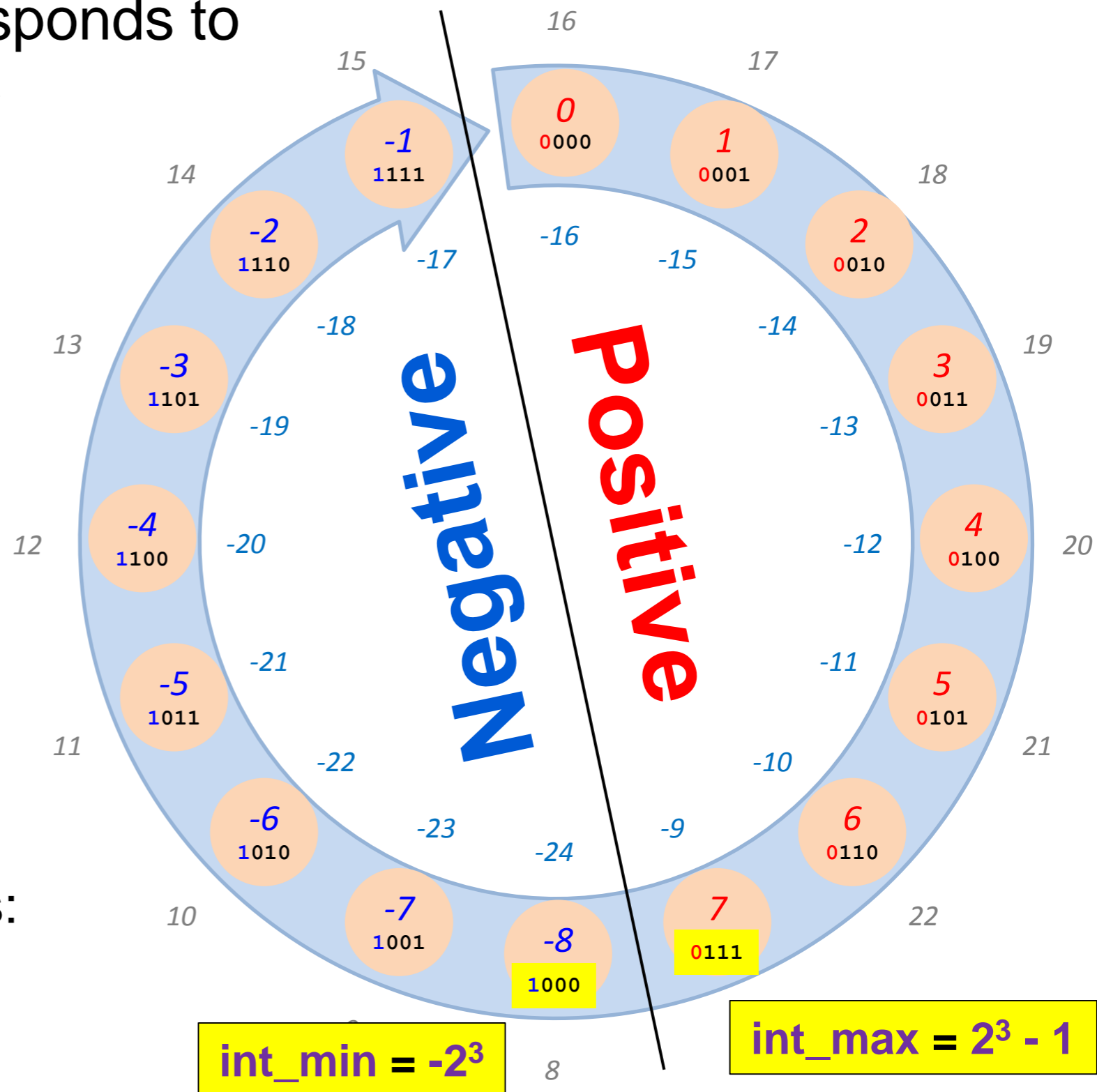
Two's Complement Overflow

- An operation **overflows** it its mathematical result is outside the range -2^{k-1} to $2^{k-1} - 1$

  If it is $< -2^{k-1}$, this is sometimes called **underflow**

- E.g.,
  - int_max + 1
  - int_min - 3
  - 2 * int_max
  - 17 * int_min
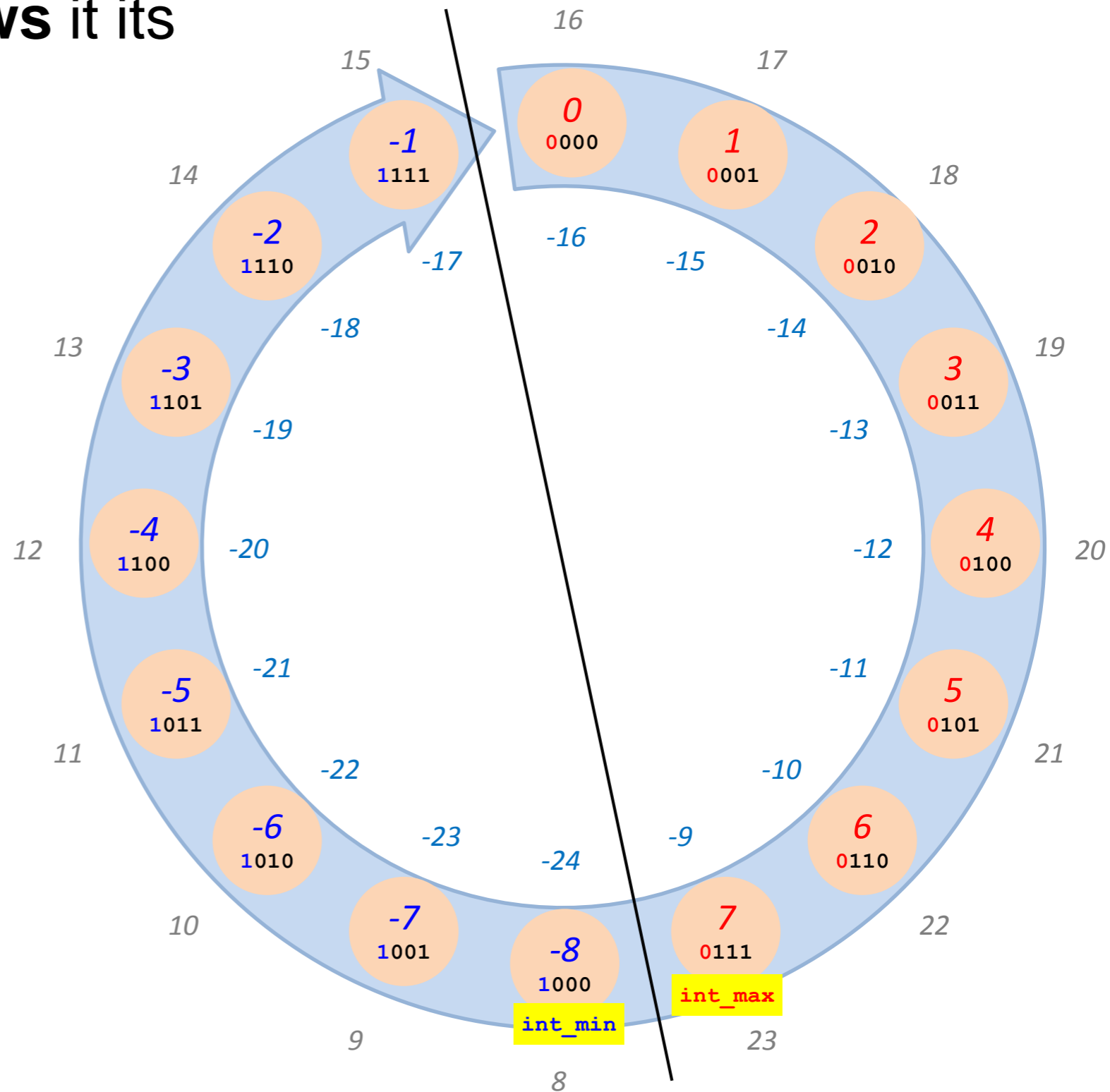
int's in C0

- C0 represents integers as 32-bit words
- It handles overflow using modular arithmetic
- The range of int's is based on two's complement
  - **int_max** = 2^{31} - 1 = 2147483647
  - **int_min** = -2^{31} = -2147483648

  - Their values are defined as the functions int_max() and int_min() in the <util> system library

```
Linux Terminal

# coin -l util
C0 interpreter (coin) …
…
--> int_max();
2147483647 (int)
--> int_min();
-2147483648 (int)
-->
```

Reasoning about int Code

- Comparing int values in C0 does **not** work like comparing numbers in normal arithmetic

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

- This function does not always return "Good"
  - if x is **int_max**, it returns "Strange"!
    - but in math x+1 > x for **any** x!

- When reasoning about code that uses >, >=, < and <=, we often need to account for overflow
  - by considering special cases

  Also operators dealing with sign

  - Code that only uses +, * and - doesn't need a special treatment

Division and Modulus

Operations on int's

- So far, we learned how C0 handles
  - +, -, *:   using modular arithmetic
  - >, >=, <, <=:   using two's complement
  - *Division is missing!*

  == and != too

- We are used to division on *real numbers*:
  - *x/y* is the number *z* such that *z*y = x*
    - if y ≠ 0

- But this definition doesn't work with integers
  - there is no ***integer z*** such that 2**z* = 3

Integer Division

- With ***integers***, there is not always z such that $z * y = x$
  - z is x/y in calculus

- We introduce a new operation, the **modulus**, to pick up the slack
  - We want to define the operations x/y and $x\%y$ so that

  <div style="text-align:center">integer division of x by y   modulus of x by y</div>

  $$(x/y) * y + (x\%y) = x$$

- That's not enough!
  - defining x/y to always return 0 and $x\%y$ to return x would work
    - we don't want that!

51

Integer Division and Modulus

$$(x/y) * y + (x\%y) = x$$

- We also want the modulus to be between 0 and *y-1*
  - Also require

  > We take the absolute value in case y is negative

  $$0 \leq |x \% y| < |y|$$

- This is still not enough!
  - defining 9/4 to be 3 and 9%4 to be -3 would work
    - ❑ (9/4) * 4 + (9%4) = 3*4 - 3 = 9 and 0 ≤ |-3| < 4
    - ➢ We don't want that!

- We want division to "round down"
  - in a calculator, 9/4 = 2.25
  - so with integer division, we want 9/4 = 2
    - ➢ and therefore 9%4 = 1

Integer Division and Modulus

$$(x/y) * y + (x\%y) = x$$

$$0 \le |x \% y| < |y|$$

Division should "round down"

- But what does "rounding down" mean for negative numbers?
  - does -2.25 rounds down to -2? — "down" towards 0
  - or does -2.25 round down to -3? — "down" towards -∞

- In C0, integer division rounds toward 0
  - so -9/4 == -2 in C0
  - In other languages, it rounds towards -∞ — Python, for example

Division by Zero

- In math, division by zero is **undefined**

- In a program, division by zero is an **error**
  - C0 will abort execution

- Any time we have x/y in a program, we must have a reason to believe that y != 0
  - 0 is not a *valid* value for the denominator of a division

- In C0, we flag invalid values using **preconditions**
  - *some primitive operations come with preconditions*
    - not just user-defined functions

Linux Terminal

```
# coin
C0 interpreter (coin) …
--> 5/0;
Error: division by zero.
Last position: <stdio>:1.1-1.4
-->
```

Safety Requirements

- Integer division, x/y, has the precondition

//@requires y != 0;

- There is another *invalid* input:
int_min()/-1 also **aborts** the program
  - ➤ this is because computer chips raise errors on these values
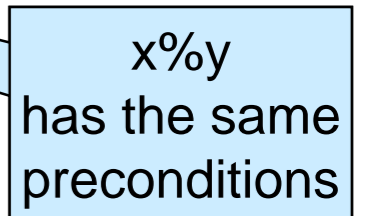
- Integer division has a second precondition:

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

x%y
has the same
preconditions

- Code that uses / or % must be **safe**
  - ○ We must prove that these preconditions are satisfied

Operations on int's – Summary

- **+, -, *:**   handled using **modular arithmetic**

  == and != too

- **>, >=, <, <=:**   handled using **two's complement**

- x/y rounds towards 0 – always

- x/y and x%y have preconditions

  //@requires y != 0;

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

Bit Patterns

Using int Beyond Numbers

- So far, we used the type int to represent integers
  - numbers!

- But in C0, an int is always 32 bits

- We can use an int to represent any data we can fit in 32 bits
  - pixels, network packets, …

  Then, an int does not represent a number but a **bit pattern**

- C0 has a special set of operations to manipulate bit patterns
  - they are the **bitwise operations** and the **shifts**
  - +, -, *, / and % are called the **arithmetic operations**

  We *could* use the arithmetic operations to manipulate bit patterns
  but that's inefficient and error prone

Pixels as 32-bit int's

- A **pixel** is a dot of color in an image
  - The color of a pixel can be described by specifying
    - how much **red**, **green** and **blue** it contains
    - how opaque it is – this part is called the **alpha** component

> This is called the ARGB representation

- Pixels are efficiently represented as bit patterns

alpha	red	green	blue

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
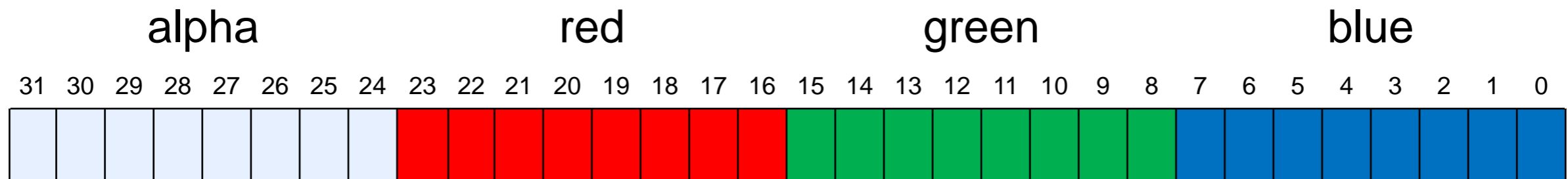
- bits 0-7 give the intensity of blue

> • A value of 0 means there is no blue
> • A value of 255 means maximally blue

- bits 8-15 give the intensity of green

> *Similar*

- bits 16-23 give the intensity of red

> *Similar*

- bits 24-31 specify the opacity

> • 0 means fully transparent
> • 255 means fully opaque

Pixels as Bit Patterns

- To describe a pixel, we need to give all its 32 bits
  - ➢ E.g., 10110011011100110101101011111001
  - ○ We are better off using hexadecimal
    - ➢ 0xB3735AF9

This is mind numbing!

We always use hex with bit patterns

alpha	red	green	blue

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
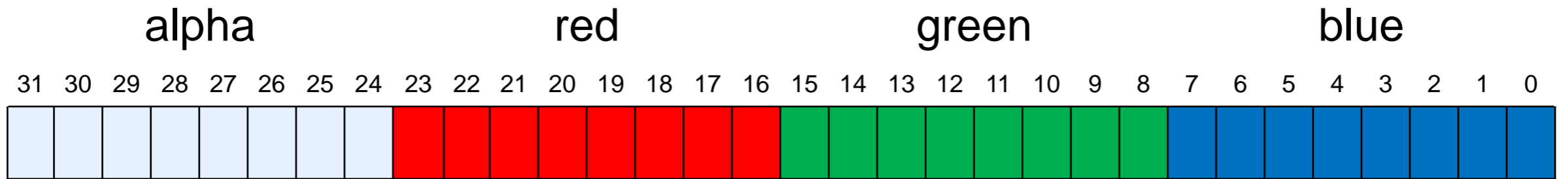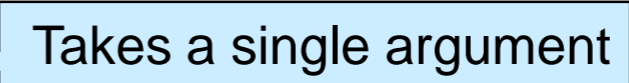
1011 0011 0111 0011 0101 1010 1111 1001

B   3   7   3   5   A   F   9

Background
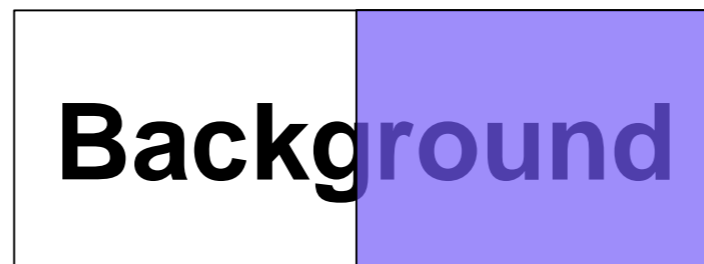
Here's the color of this pixel

Bitwise Operations

Bitwise Operations

- The **bitwise operations** manipulate the bits of a bit pattern independently of the other bits nearby

- They are

  ~ – pronounced "not"   Takes a single argument

  & – pronounced "and"

  | – pronounced "or"   Take two arguments

  ^ – pronounced "xor"

- Let's see how they work on an individual bit

Bitwise Operations on One Bit

- Here are the tables that give the output for each input

This says that:
- 0 & 0 is 0
- 0 & 1 is 0
- 1 & 0 is 0
- 1 & 1 is 1

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

- C0's bitwise operations take int's as input and return an int
  - ○ there is no type for individual bits in C0

- They apply the tables on each bit of their inputs, **position by position**
  - ○ so, if int's were 6 bits,

But we know they are 32 bit

```
   000111          000111          000111
 & 010101        | 010101        ^ 010101        ~ 010101
 _____        _____        _____        _____
   000101          010111          010010          101010
```

6-bit examples

- **&** and **|** are related to **&&** and **||** but
  - ○ **&** and **|** take two int's and return an int
  - ○ **&&** and **||** take two bool's and return a bool

64

Bitwise And – &

Let's see how to use the bitwise operations to manipulate bit patterns

- If we "and" any bit **b** with
  - **0**, we always get **0**
    - ❑ b & 0 = 0
  - **1**, we always get **b** back
    - ❑ b & 1 = b

&	0	1
0	0	0
1	0	1

b

mask

b & 0 = 0   b & 1 = b
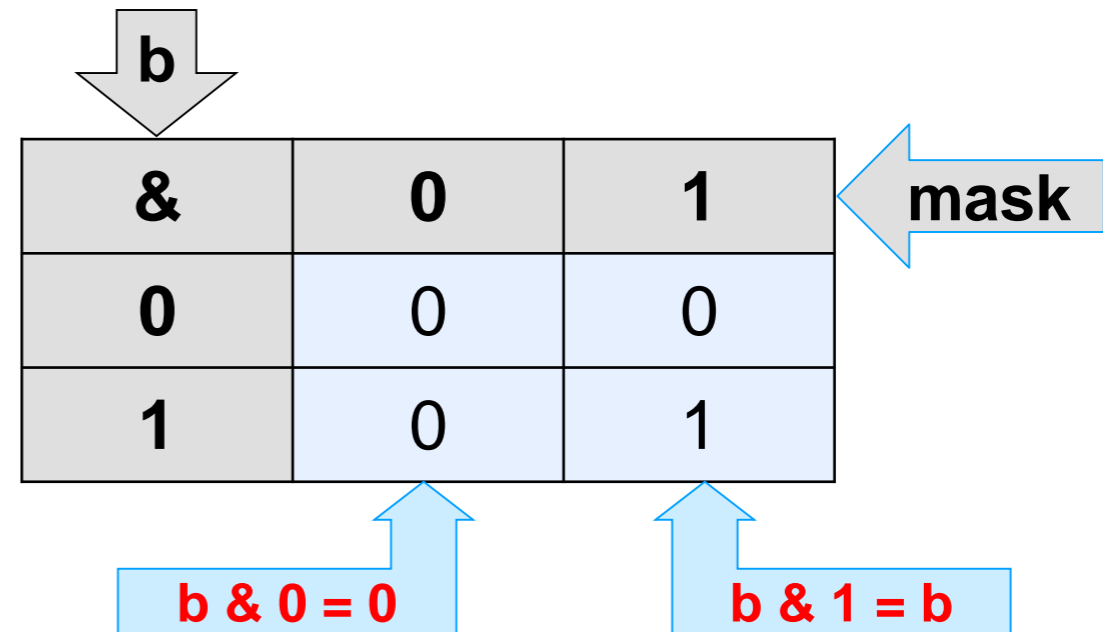
- If the int x is a bit pattern, then **x & m** is an int that
  - has the same bits as x where m is 1
  - and has a zero where m is 0

- The int m is called a **mask**
  - it allows us to retain specific bits of interest in x

&: Clearing Bits

&	0	1	mask
0	0	0	
1	0	1	

b

b & 0 = 0   b & 1 = b

alpha | red | green | blue

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
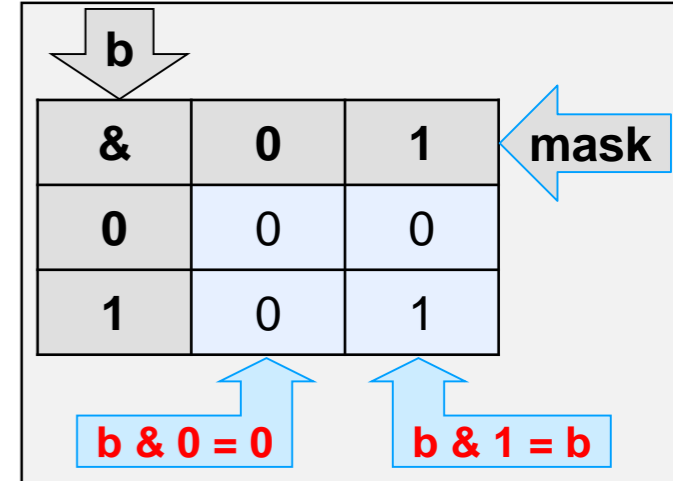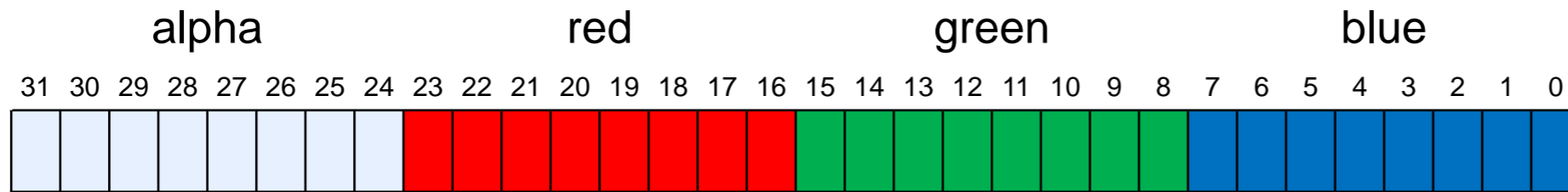
- We want to write a function that returns a pixel identical to p but with no red in it
  - zero out red component of p – bits 16-23
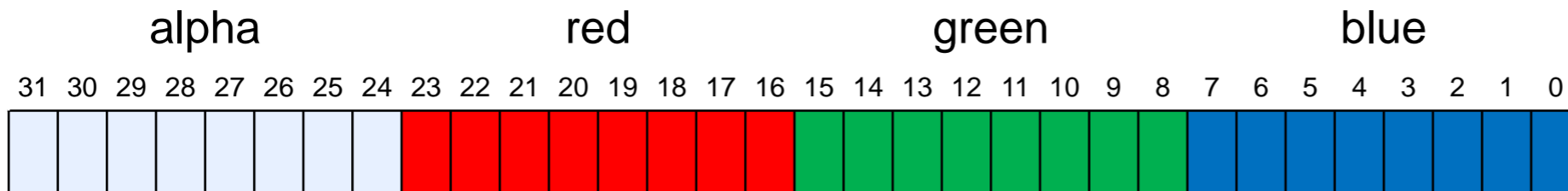 - preserve the all other bits

- We can use the **mask** 0xFF00FFFF
  - bits 16-23 are 0
  - all other bits are 1

Mask

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

- Here's how it looks on our example

This is 0xB3735AF9

Background

clear_red

This is 0xB3005AF9

Background

&: Isolating Red

&	0	1	**mask**
0	0	0	
1	0	1	

b & 0 = 0   **b & 1 = b**

alpha   red   green   blue

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
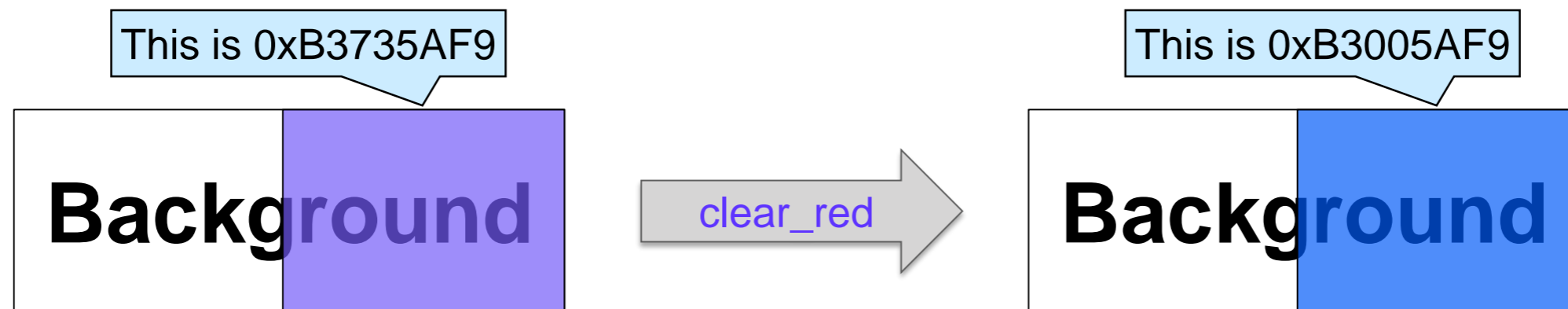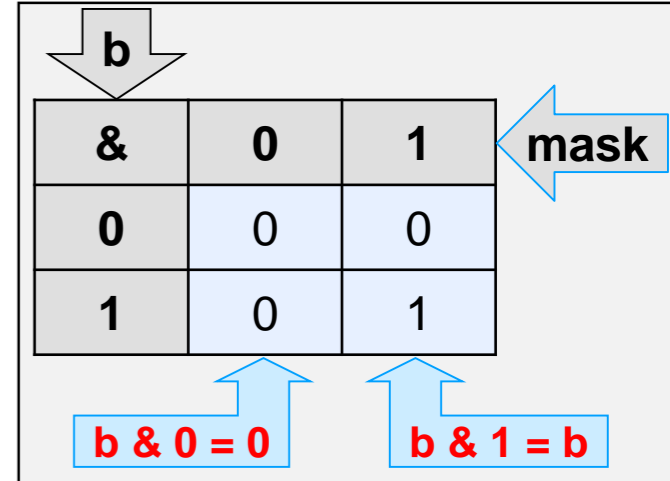
- We want to return a pixel with just the red component of p
  - preserve the red component of p – bits 16-23
  - zero out all other bits
  - "and" p with the mask 0x00FF0000
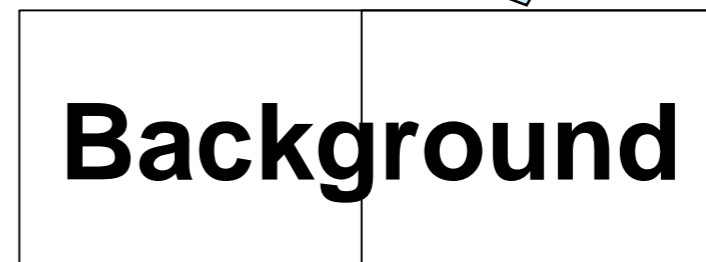
```
int make_red(int p) {
  int red = p & 0x00FF0000;
  return red;
}
```

Mask
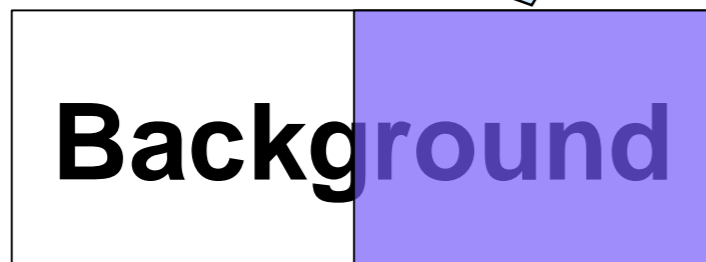
Where's the red? The alpha channel is 00 so it is totally transparent

This is 0xB3735AF9

This is 0x00730000

Background

make_red

Background

Bitwise Or – |

- If we "or" any bit **b** with
  - **0**, we always get **b** back
    - ❑ b | 0 = b
  - **1**, we always get **1**
    - ❑ b | 1 = 1

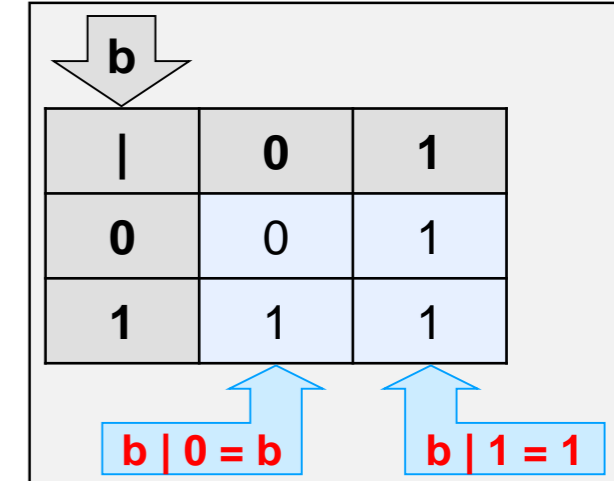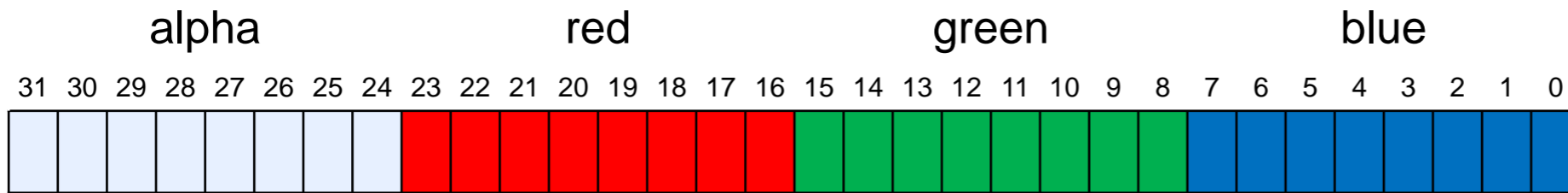| **|**	**0**	**1**
0	0	1
1	1	1

b | 0 = b   **b | 1 = 1**

- Common uses of | are
  - setting bits to 1
  - constructing a bit pattern from parts
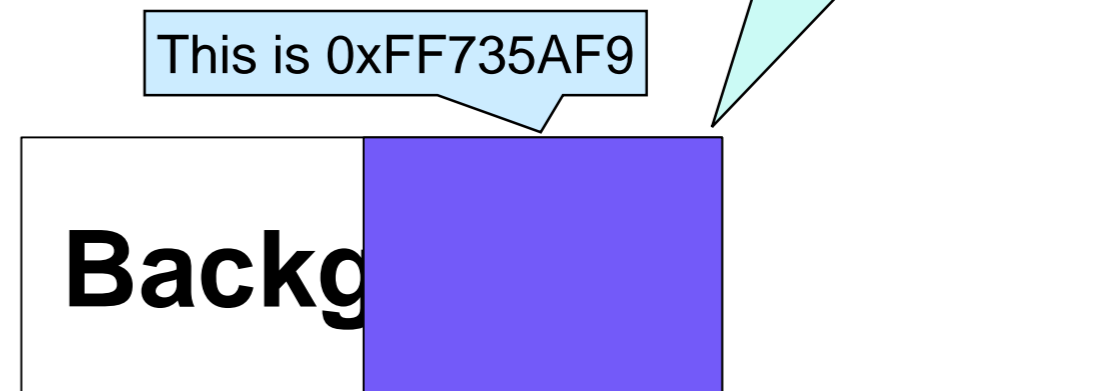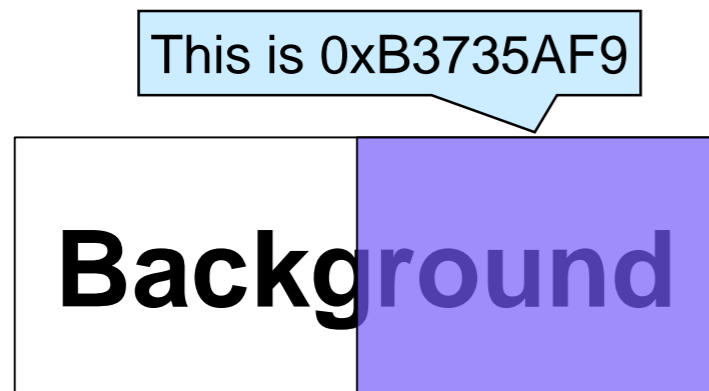
This is similar to clearing bits with &

|: Opacify

		b	
|		**0**	**1**
0		0	1
1		1	1

b | 0 = b   b | 1 = 1

	alpha								red								green								blue							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **We want to make a pixel fully opaque**
  - set the alpha bits to 1 – bits 24-31
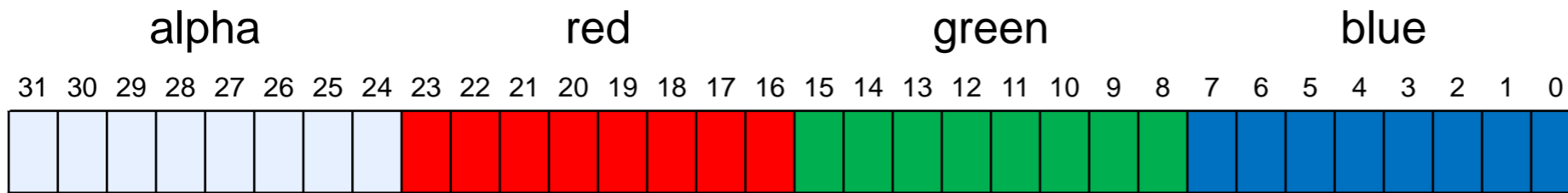 - preserve the other component of p

- **We can "or" p with 0xFF000000**
  - bits 24-31 become 1
  - all other bits stay as in p

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

This is 0xB3735AF9

Background

opacify

This is 0xFF735AF9

Same color but fully opaque

Backg

|: Constructing Pixels from Parts

	alpha	red	green	blue

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

b		
|	**0**	**1**
0	0	1
1	1	1

b | 0 = b   b | 1 = 1
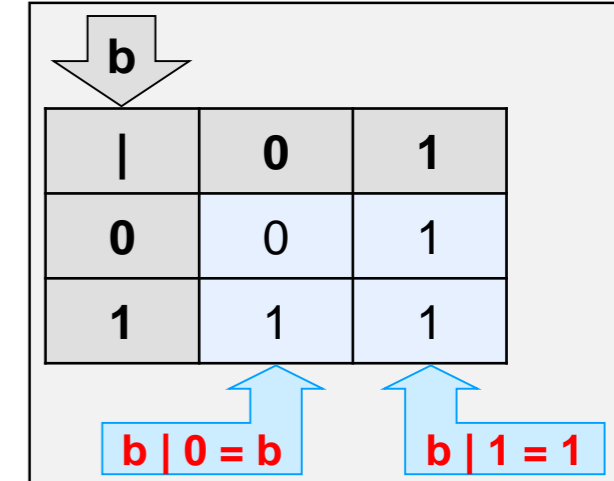
- Return a pixel with the same green component as p and the same alpha, red and blue components as q

  ○ isolate the green component of p using the mask 0x0000FF00

  ○ isolate the other components of q using the mask 0xFFFF00FF

  ○ combine them with "or"

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

if p is 0xB3735AF9, then p_green is 0x00005A00

if q is 0xCDA1E805, then q_others is 0xCDA10005

0x00005A00
| 0xCDA10005
= 0xCDA15A05
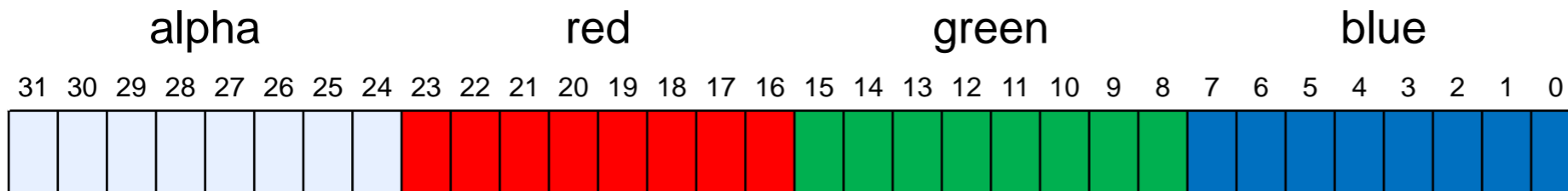
This is 0xB3735AF9

This is 0xCDA1E805

This is 0xCDA15A05

Background

Background

franken_pixel

Background
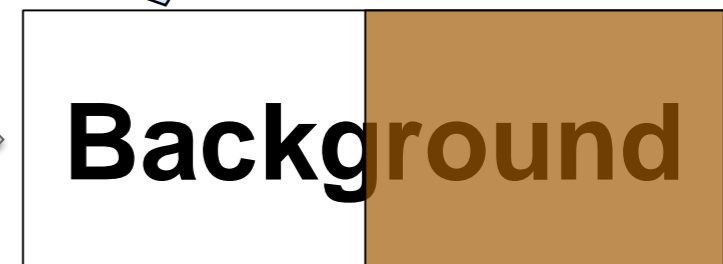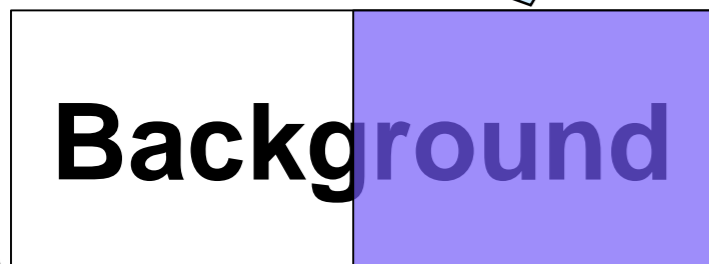
Bitwise Not – ~

● Bitwise negation flips bits

~	0	1
	1	0

~: Flipping bits

~	0	1
	1	0

alpha	red	green	blue

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

- **Return the pixel with the same opacity but inverted colors**
  - preserve the alpha channel
  - change the value of all other channels to 255 minus their original value
    - ❑ that's the same as flipping the bits of all channels

```
int invert(int p) {
    return (p & 0xFF000000) | (~p & 0x00FFFFFF);
}
```

This is 0xB3735AF9

Background
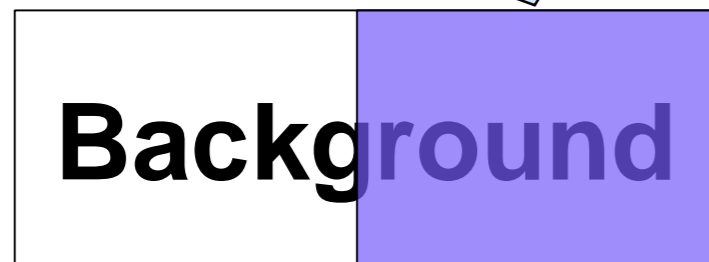
opacify

This is 0xB38CA506

Background

Bitwise Xor – ^

- If we "xor" any bit **b** with
  - **0**, we always get **b** back
    - ❑ b ^ 0 = b
  - **b** itself, we always get **0**
    - ❑ b ^ b = 0
  - furthermore, "xor" is associative and commutative

^	0	1
0	0	1
1	1	0

b

b ^ 0 = b

b ^ b = 0

- One consequence is that **(m ^ k) ^ k = m**
  - if **m** is a *message* and **k** is a *key*
    then **x** = **(m ^ k)** is the *encryption* of **m** with **k**
  - to *decrypt* **x**, we do **x ^ k**, and **m** pops out

- "xor" is commonly used in cryptography

Shifts

Moving Bits Around

- The bitwise operations manipulate each position independently from all other positions in a bit pattern
  - We can't use them to move bits to new positions

- The **shift operations** enable us to move bits around
  - **left shift**:   x << k moves the bits of x left by k positions
  - **right shift**: x >> k moves the bits of x right by k positions

The int x is understood as a **bit pattern**

The int k is understood as a **number**
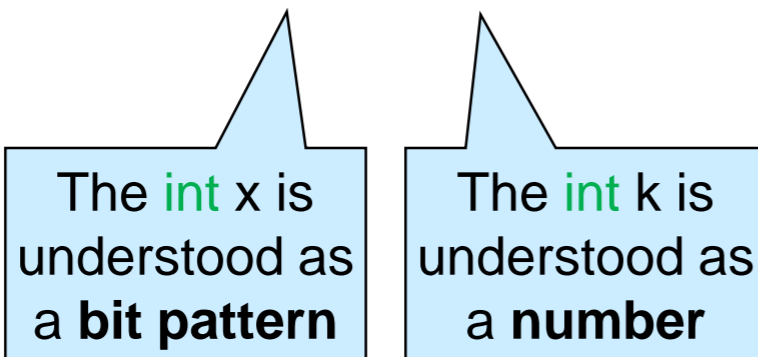
- Since an int has 32 bits, k must be between 0 and 31

//@requires 0 <= k && k < 32;

Unsafe otherwise

Left Shift

- **x << k** shifts the bits of x left by k positions
  - the leftmost k bits of x are dropped
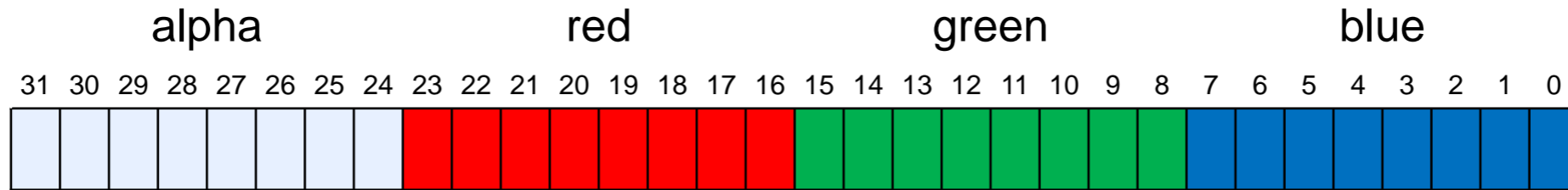 - the rightmost k bits of the result are set to 0

- So
  - 0101 << 1 evaluates to 1010:   0101

    1010

 - 0101 << 3 evaluates to 1000:   0101

    1000

4-bit examples

76

Blue Everywhere

alpha   red   green   blue

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

- Return a pixel whose red and green components have the same intensity as p's blue component
  - isolate the blue component of p
  - put it in the red, green and blue positions

```
int blue_everywhere(int p) {
    int alpha = p & 0xFF000000;
    int blue  = p & 0x000000FF;
    return alpha | (blue << 16) | (blue << 8) | blue;
}
```

Leave alpha unchanged

Why is it gray? Gray is when all colors are the same

This is 0xB3735AF9

This is 0xB3F9F9F9

Background   blue_everywhere   **Background**
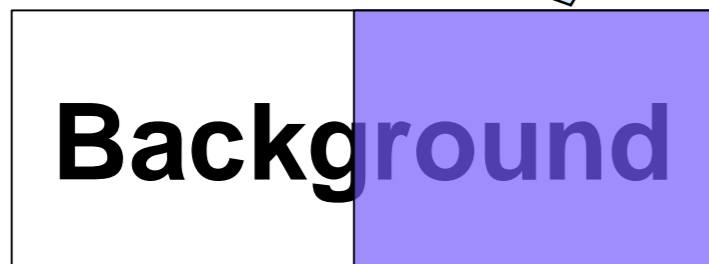
Right Shift

- **x >> k** shifts the bits of x right by k positions
  - ○ the rightmost k bits of x are dropped
  - ○ the leftmost k bits of the result are a copy of the leftmost bit of x
    - ➢ This is called **sign extension**

    That's because in two's complement, the leftmost bit is the sign bit

- So
  - ○ 0101 >> 1 == 0010
  - ○ 0101 >> 3 == 0000

    The sign bit is 0, so we add 0's

  - ○ 1010 >> 1 == 1101
  - ○ 1010 >> 3 == 1111

    The sign bit is 1, so we add 1's

    4-bit examples

    Sign bit

78

Swapping the Alpha and Red Channels

alpha	red	green	blue

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
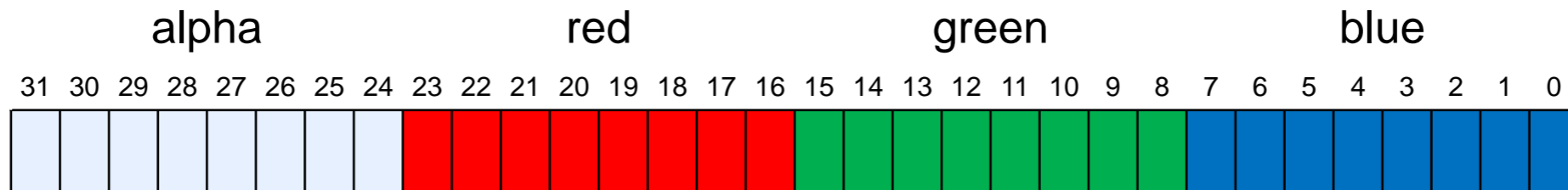
- Return a pixel identical to p, but where the red and alpha channel are swapped

If **p** is 0xB3735AF9

Background

swap_alpha_red

we want 0x73B34AF9

Background
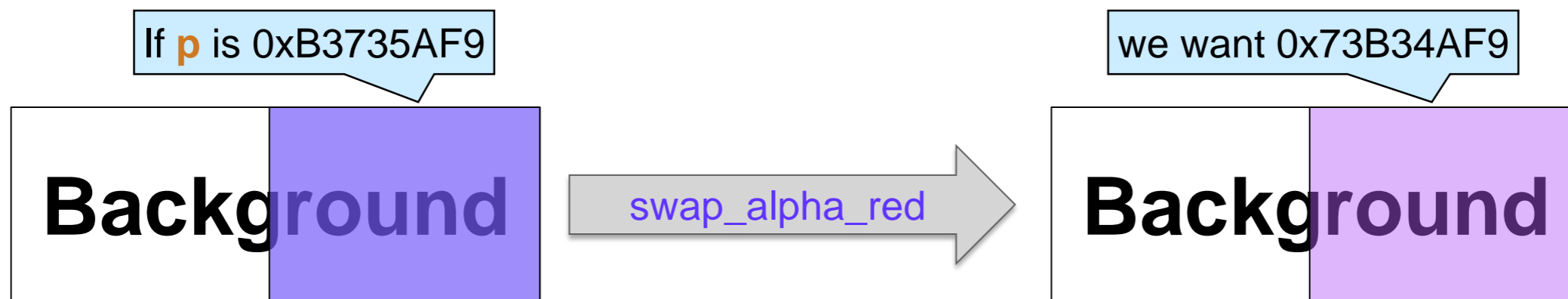
- ○ isolate the channels of p
- ○ shift alpha right by 8 bits — so that its bits are in the red position
- ○ shift red left by 8 bits — so that its bits are in the alpha position
- ○ combine the parts and return

Swapping the Alpha and Red Channels
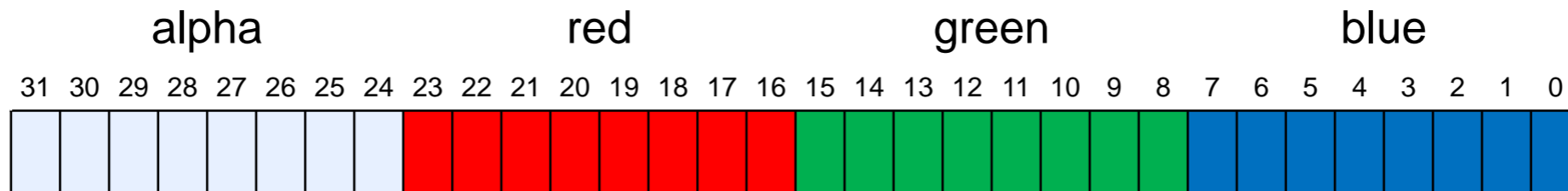
alpha	red	green	blue

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
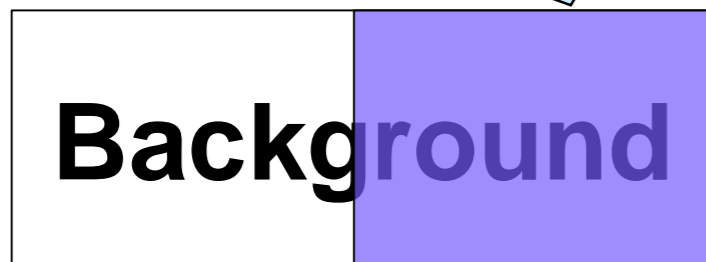
```
int swap_alpha_red(int p) {
   int new_alpha = (p & 0x00FF0000) << 8;
   int new_red   = (p & 0xFF000000) >> 8;
   int old_green = p & 0x0000FF00;
   int old_blue  = p & 0x000000FF;
 return new_alpha | new_red | old_green | old_blue;
}
```
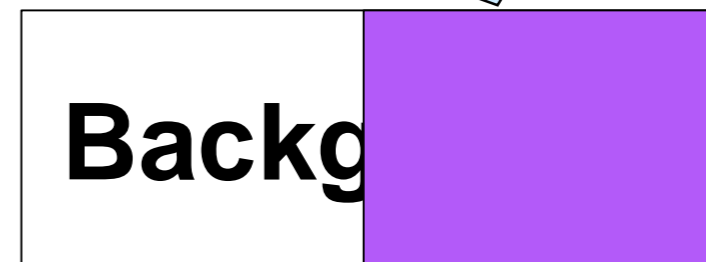
○ isolate the channels of p

○ shift alpha right by 8 bits

○ shift red left by 8 bits

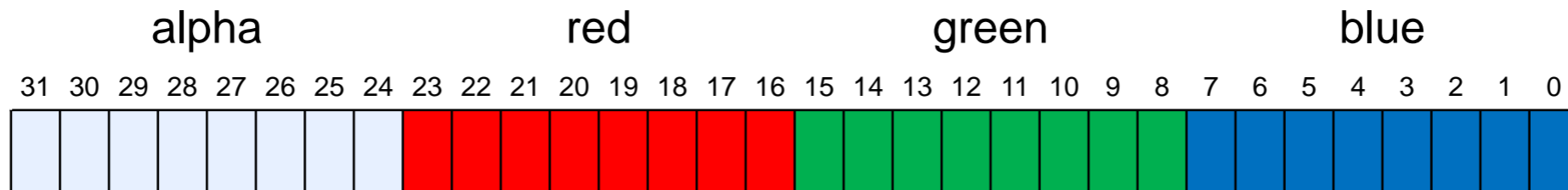○ combine the parts and return

● Let's test it

This is 0xB3735AF9

Background

swap_alpha_red

This is wrong!

This is 0xFFB35AF9

Backg

Swapping the Alpha and Red Channels

alpha	red	green	blue

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

- We have a bug!

If p is 0xB3735AF9,

```
int 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;
}
```

this is 0x73000000 ✓
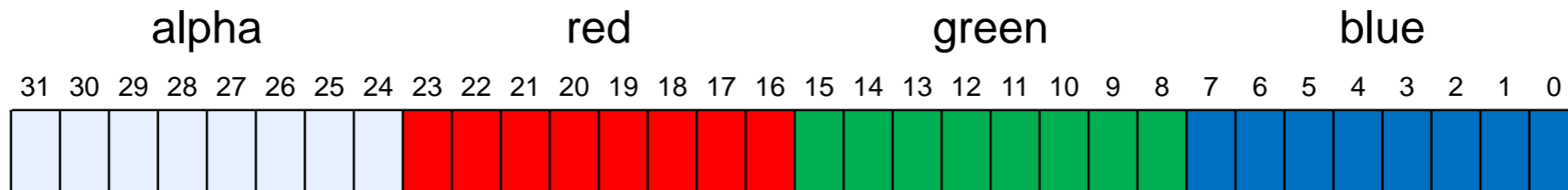
this is 0xFFB30000 ✗

this is 0x00005A00 ✓

this is 0x000000F9 ✓

- (p & 0xFF000000) >> 8 extends p's sign bit over the 8 leftmost bits
  - Beware of sign extension!

81

Swapping the Alpha and Red Channels

|	alpha								red								green								blue							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- To fix the bug, get rid of the sign-extended bits
  - **mask after shifting**

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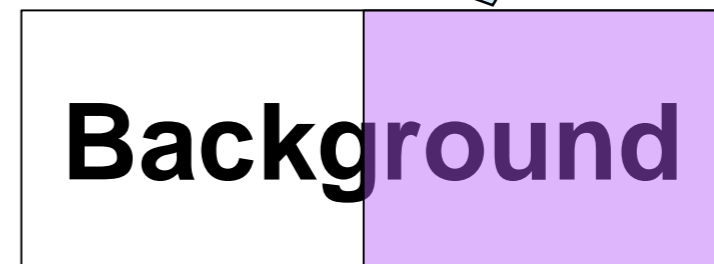
This solves the issue

This is equivalent to what we had, but better be consistent

This is 0xB3735AF9

Background

swap_alpha_red
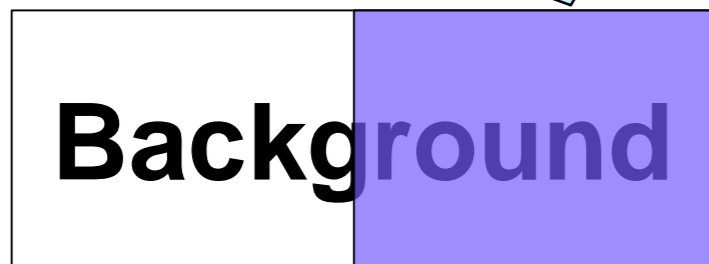
This is 0x73B35AF9

Background

int Summary

The type int is used to

- **represent integers**
  - it uses modular arithmetic and two's complement
  - it manipulates them using the **arithmetic operations**
    - +, -, &, /, %, >, >=, <, <=

- **encode bit patterns**
  - it manipulates them using the **bitwise operations** and the **shifts**
    - &, |, ~, ^
    - <<, >>

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

Arithmetic vs. Bitwise Operations

NEVER mix and match arithmetic and bitwise operations

● Exceptions

  ○ $-x = \sim x + 1$

  > Inside a processor chip,
  > - this is an efficient way to compute -x
  > - it avoids the need for circuitry for subtraction

  ○ $x << k = x * 2^k$

  > $x << k$ is a very efficient way to computer $x * 2^k$.
  > You are very likely to use it

  ➢ in particular, $1 << k = 2^k$

  ○ $x >> k = x$ divided by 2^k (*Python* division, not C0's)

  > $x >> k$ is a very efficient too, but you are unlikely to use it: it's the "wrong" division