Integer Representation

Representation of integers: unsigned and signed
Modular arithmetic and overflow
Sign extension
Shifting and arithmetic
Multiplication
Casting

Fixed-width integer encodings

Unsigned $\subset \mathbb{N}$ non-negative integers only

Signed $\subset \mathbb{Z}$ both negative and non-negative integers

n bits offer only 2^n distinct values.

Terminology:

“Most-significant” bit(s) or “high-order” bit(s)

MSB

0110010110101001

“Least-significant” bit(s) or “low-order” bit(s)

LSB

(4-bit) unsigned integer representation

1	0	1	1
8	4	2	1
2^3	2^2	2^1	2^0
3	2	1	0

$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

weight

position

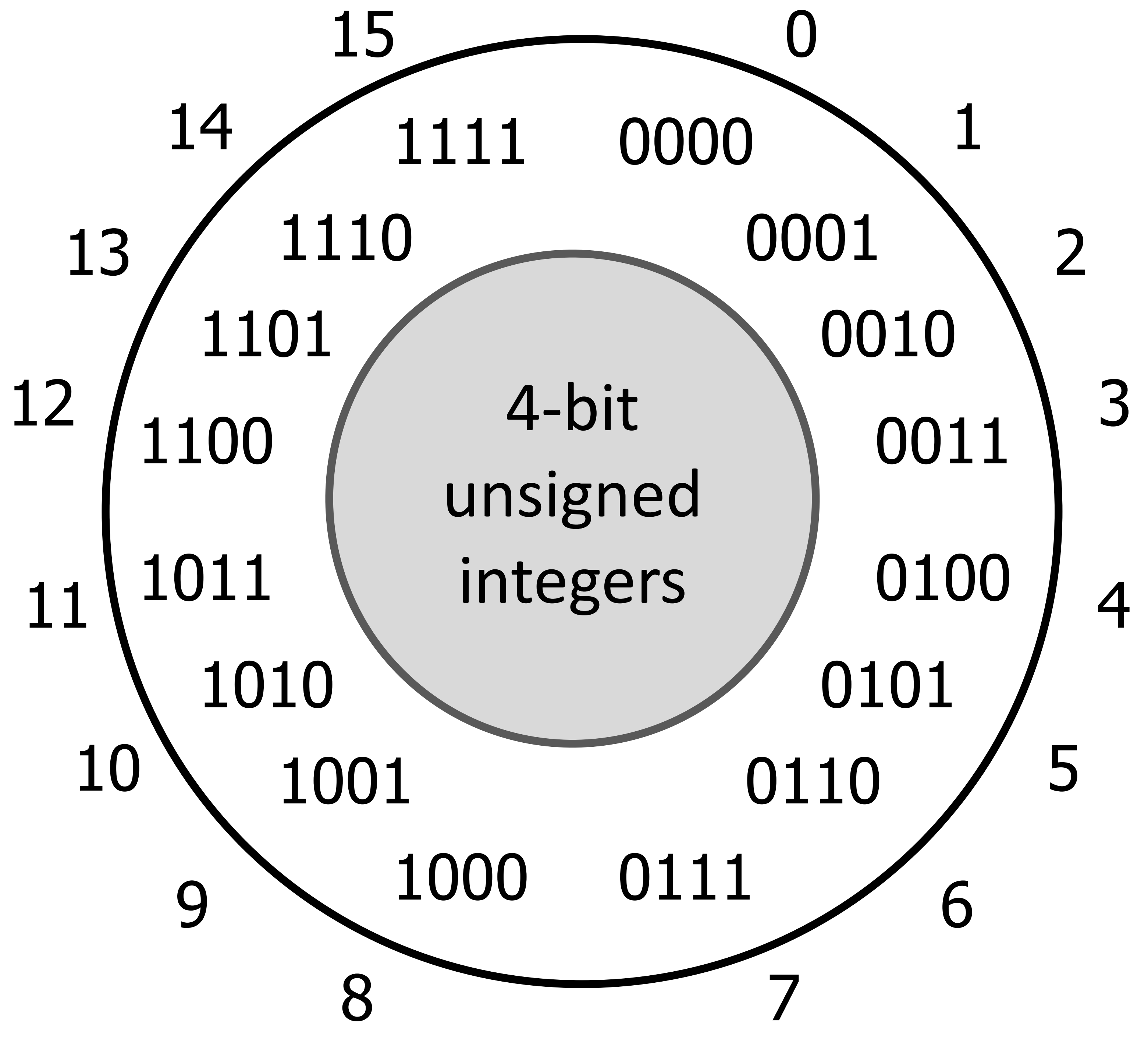
n-bit unsigned integers:

unsigned minimum = 0

unsigned maximum = $2^n - 1$

modular arithmetic, unsigned overflow

```
       1
 11   1011
+ 2 + 0010
 13   1101
```




```
      1 1  1
 13    1101
+ 5  + 0101
  2    0010
```

`x+y` in *n*-bit unsigned arithmetic is ***(x + y) mod 2^N*** in math

unsigned overflow *= "wrong" answer = wrap-around = carry 1 out of MSB = math answer too big to fit*

Unsigned addition *overflows* if and only if **a carry bit is dropped.**

(4-bit) **two's complement**
signed integer representation

compare to unsigned

1	0	1	1
$-(2^3)$	2^2	2^1	2^0

$= 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

still only 2^n distinct values, half negative.

4-bit two's complement integers:

signed minimum = $-\left(2^{(n-1)}\right)$

signed maximum = $2^{(n-1)} - 1$

4-bit min: **1000**

4-bit max: **0111**

alternate signed attempt: **sign-magnitude**

!!!

Most-significant bit (MSB) is *sign bit*

0 means non-negative 1 means negative

Remaining bits are an unsigned magnitude

Note: this is *not two's complement*

8-bit sign-magnitude:

00000000 represents _____

01111111 represents _____

10000101 represents _____

10000000 represents _____

Anything weird here?

ex

Arithmetic?

Example:

4 - 3 != 4 + (-3)

↓

```
 00000100
+10000011
```

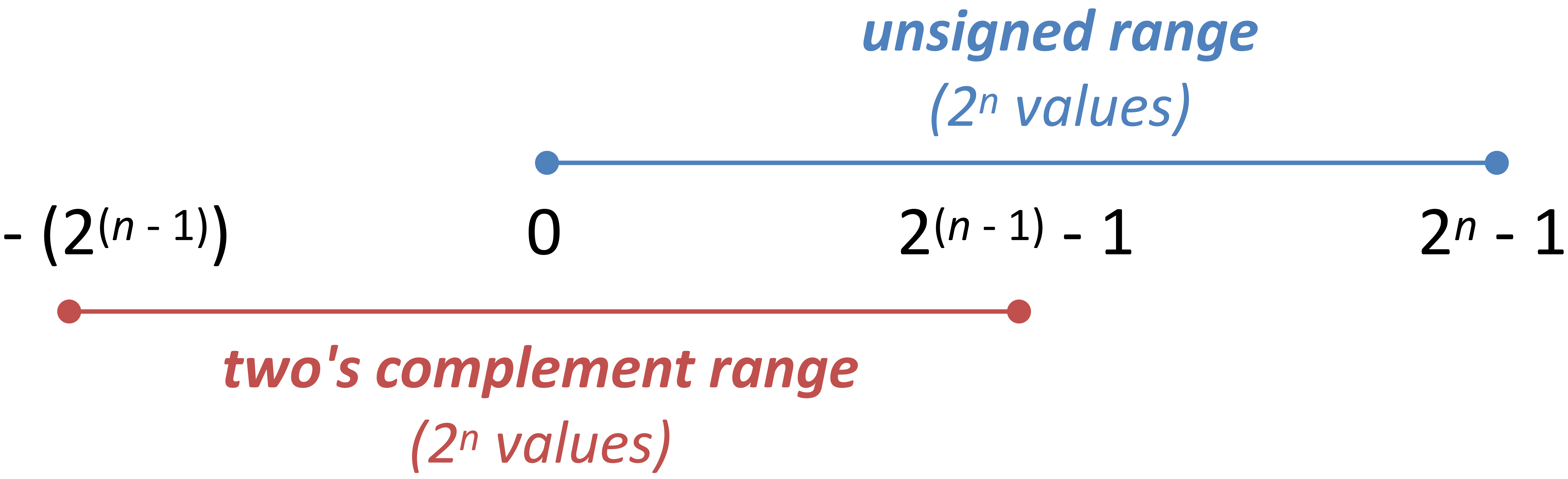
Zero?

two’s complement vs. unsigned

_	_	…	_	_	_
2^{n-1}	2^{n-2}	…	2^2	2^1	2^0
$-(2^{n-1})$	2^{n-2}	…	2^2	2^1	2^0

unsigned places (row 2)

two's complement places (row 3)



4-bit **unsigned** vs. 4-bit **two's complement**

1 0 1 1

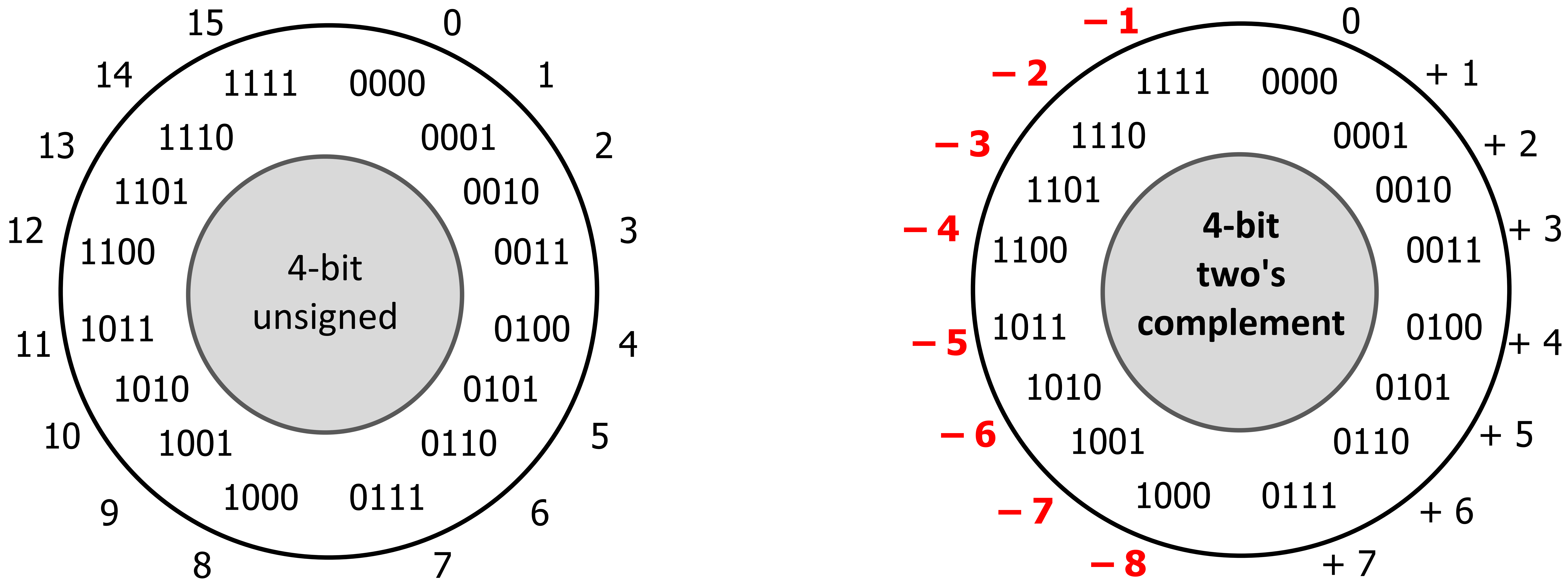
$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

11

difference = ___ = 2__

-5



8-bit representations

ex

0 0 0 0 1 0 0 1

1 0 0 0 0 0 0 1

1 1 1 1 1 1 1 1

0 0 1 0 0 1 1 1

n-bit two's complement numbers:

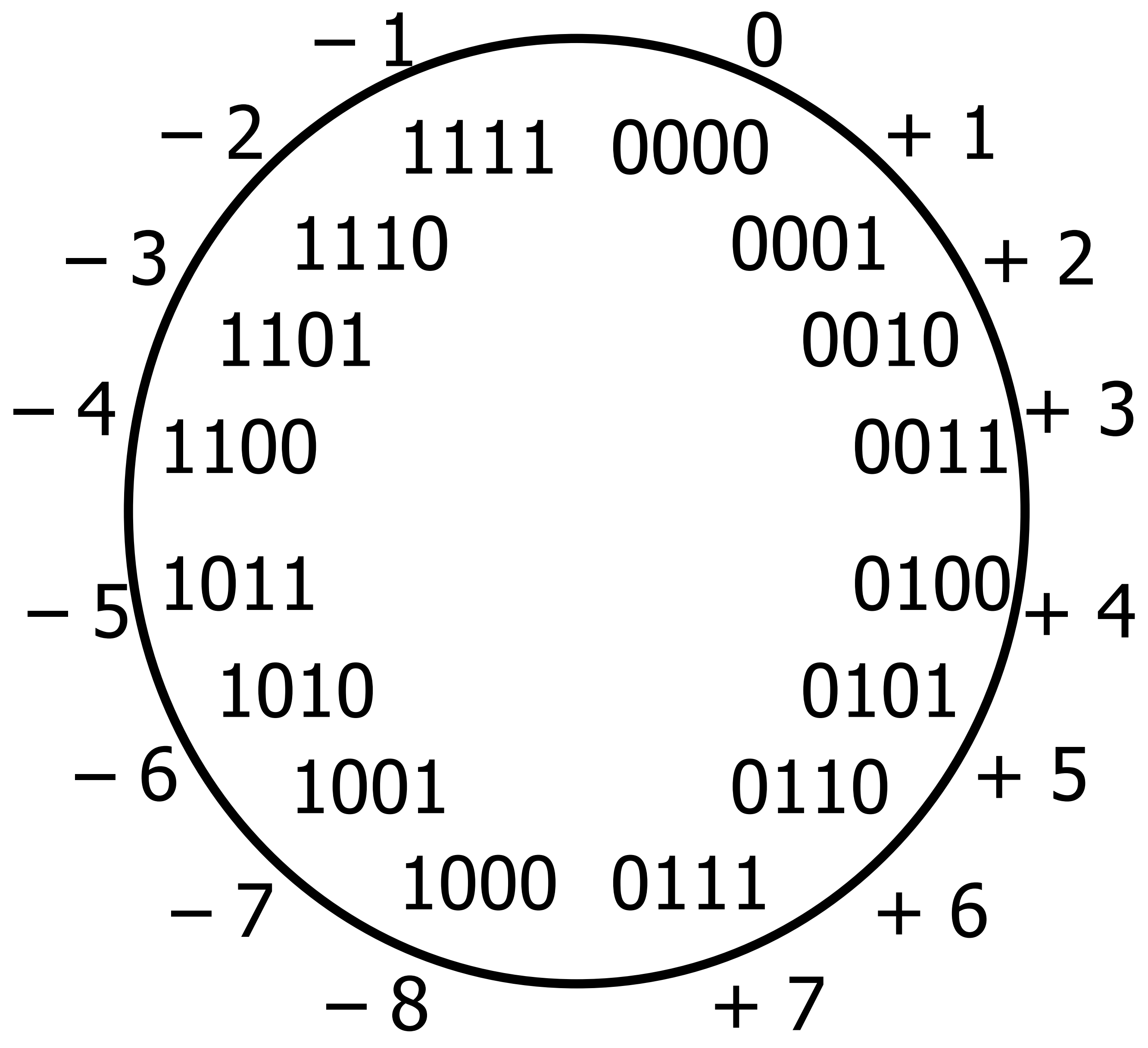
minimum =

maximum =

two's complement (signed) **addition**

```
          1              1 1
   2    0010      -2     1110
 + 3  + 0011    + -3   + 1101
   5    0101      -5     1011
```

```
       1 11
  -2    1110       2     0010
 + 3  + 0011    + -3   + 1101
   1    0001      -1     1111
```



Modular Arithmetic

two's complement (signed) ***overflow***

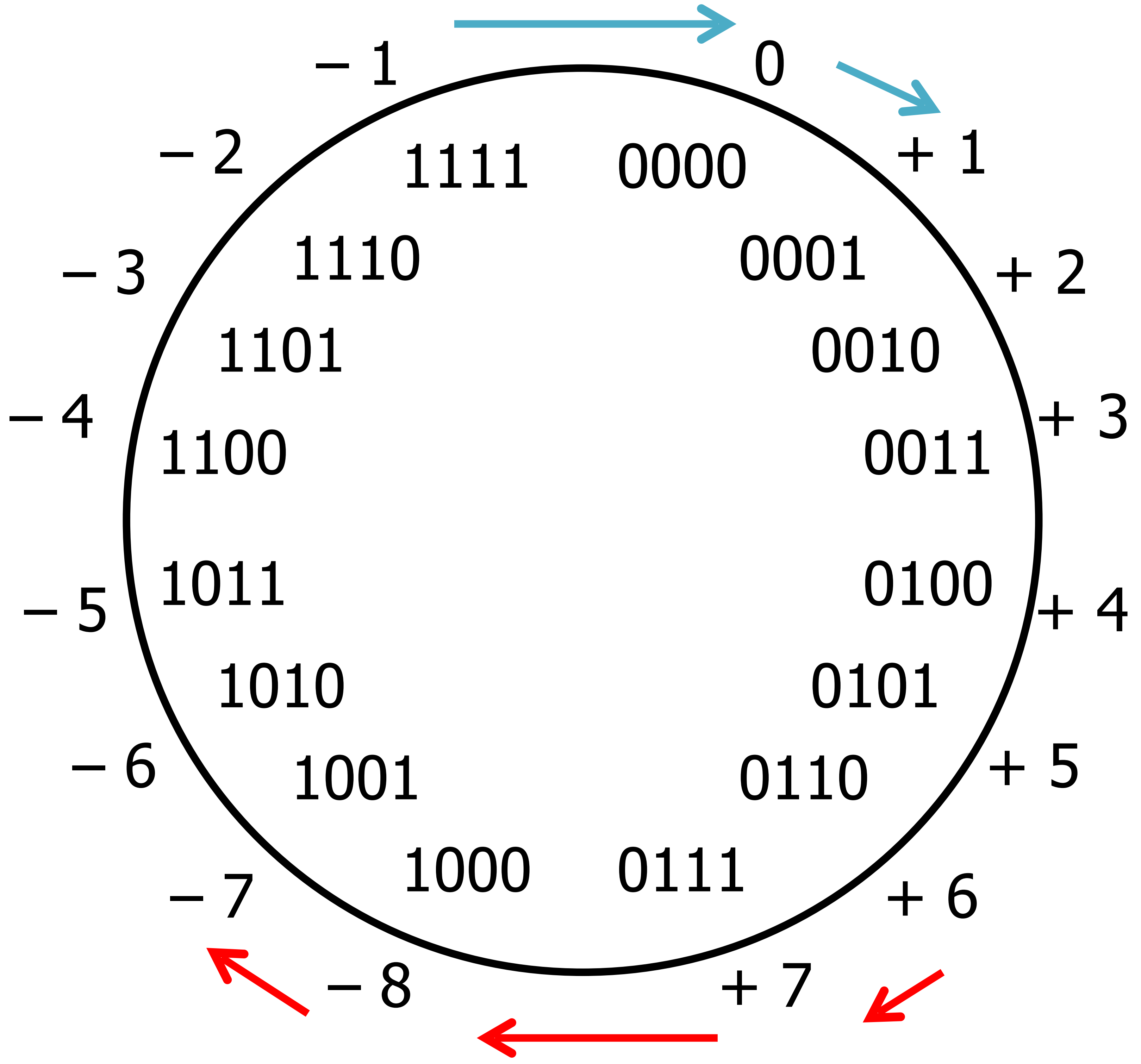
Addition *overflows*

if and only if the **arguments** have the **same sign** but the **result does not.**

if and only if the **carry in** and **carry out** of the **sign bit differ.**

```
       1 11
 -1     1111
+ 2   + 0010
        0001

       0 11
  6     0110
+ 3   + 0011
        1001
```



Modular Arithmetic

Some CPUs/languages raise exceptions on overflow.
C and Java cruise along silently... Feature? Oops?

Recall: software correctness

Ariane 5 Rocket, 1996

Exploded due to **cast** of 64-bit floating-point number to 16-bit signed number. **Overflow.**

Boeing 787, 2015

"... a **Model 787 airplane** … can lose all alternating current (AC) electrical power … caused by a **software counter** internal to the GCUs that will **overflow** after **248 days** of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in **loss of control of the airplane**."
--FAA, April 2015

A few reasons two's complement is awesome

Arithmetic hardware

The carry algorithm works for everything!

Sign

The MSB can be interpreted as a sign bit.

Negative one

-1_{10} is encoded as all ones: 0b11…1

Complement rules

-x == ~x + 1

5 is 0b0101

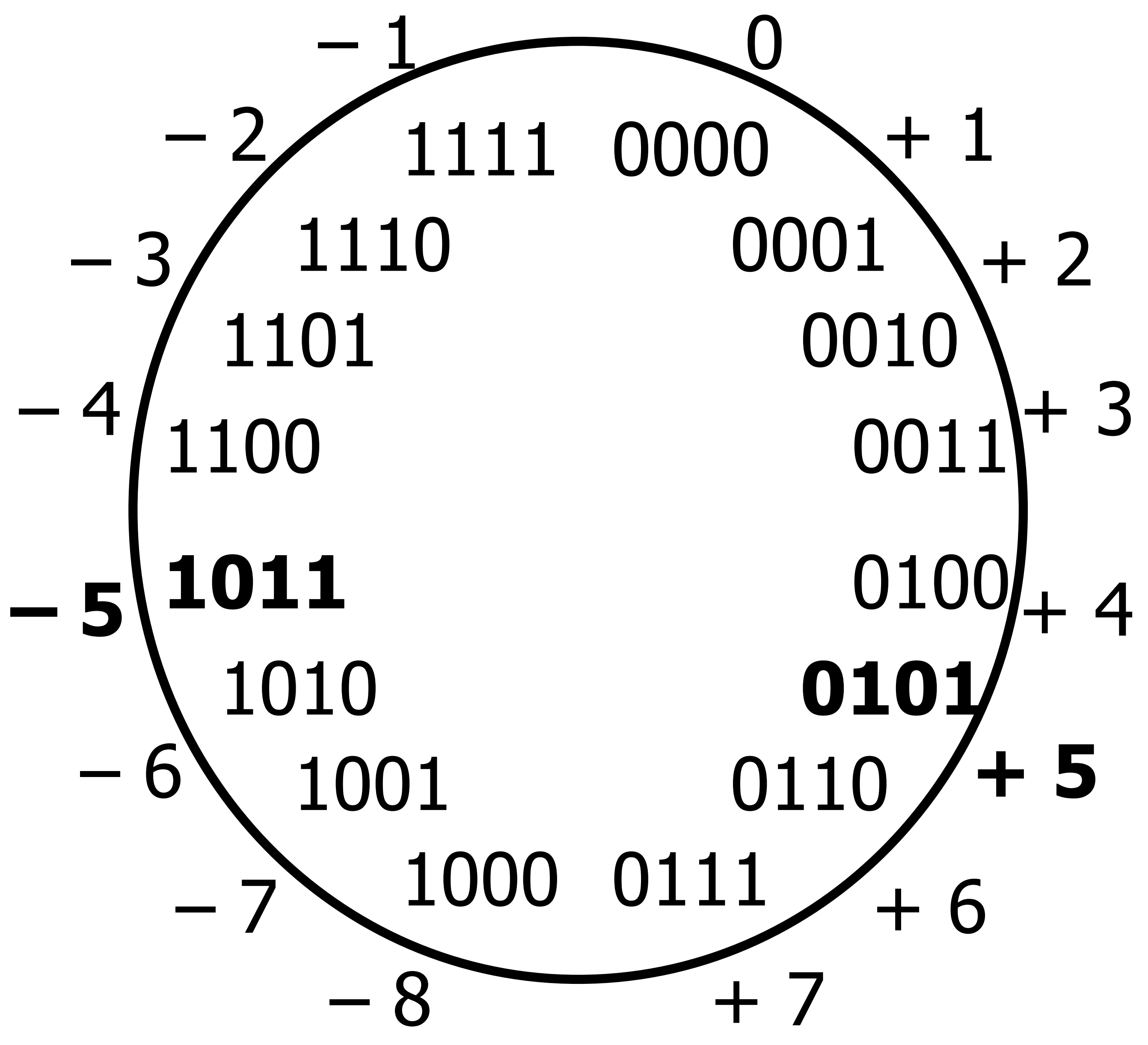
~0b0101 is 0b1010

\+ 1

0b1011 is -5

Even subtraction!

x - y == x + -y == x + ~y + 1



Another derivation

ex

How should we represent 8-bit negatives?

- For all positive integers ***x***,
  we want the representations of ***x*** and ***–x*** to sum to zero.
- We want to use the standard addition algorithm.

```
 11111111      1111111       11111111
  00000001      00000010      00000011
+11111111     +11111110     +11111101
----------    ----------    ----------
  00000000      00000000      00000000
```

- Find a rule to represent –x where that works…

Convert/cast signed number to larger type.

0 0 0 0 0 0 1 0 8-bit 2

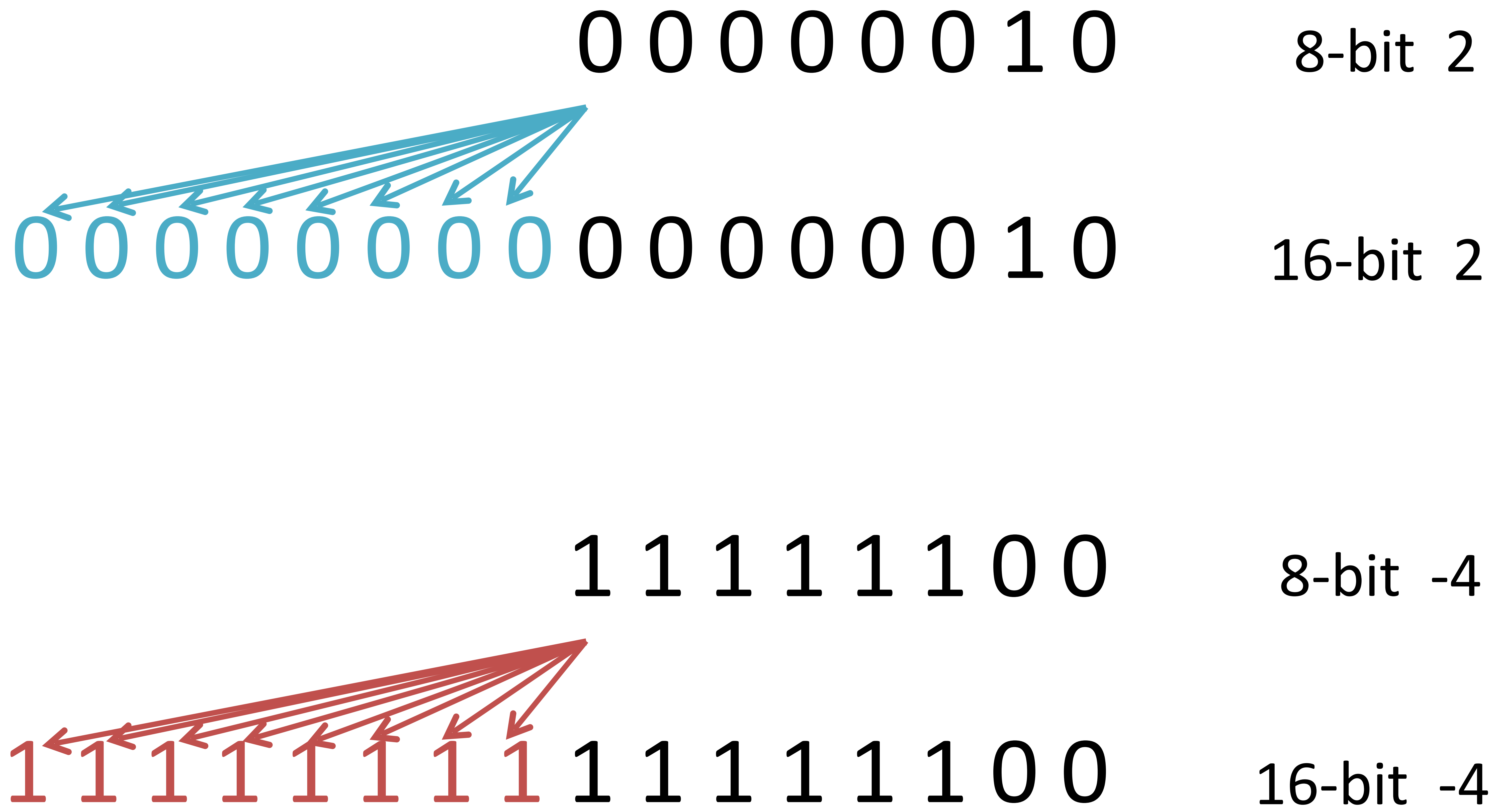
_ _ _ _ _ _ _ _ 0 0 0 0 0 0 1 0 16-bit 2

1 1 1 1 1 1 0 0 8-bit -4

_ _ _ _ _ _ _ _ 1 1 1 1 1 1 0 0 16-bit -4

Rule/name?

Sign extension for two's complement



Casting from smaller to larger signed type does sign extension.

unsigned **shifting** and **arithmetic**

unsigned
x = 27;

y = x << 2;

y == 108

0 0 0 1 1 0 1 1

0 0 0 1 1 0 1 1 0 0

logical shift left

n = shift distance in bits, w = width of encoding in bits

logical shift right

1 1 1 0 1 1 0 1

0 0 1 1 1 0 1 1 0 1

unsigned
x = 237;

y = x >> 2;

y == 59

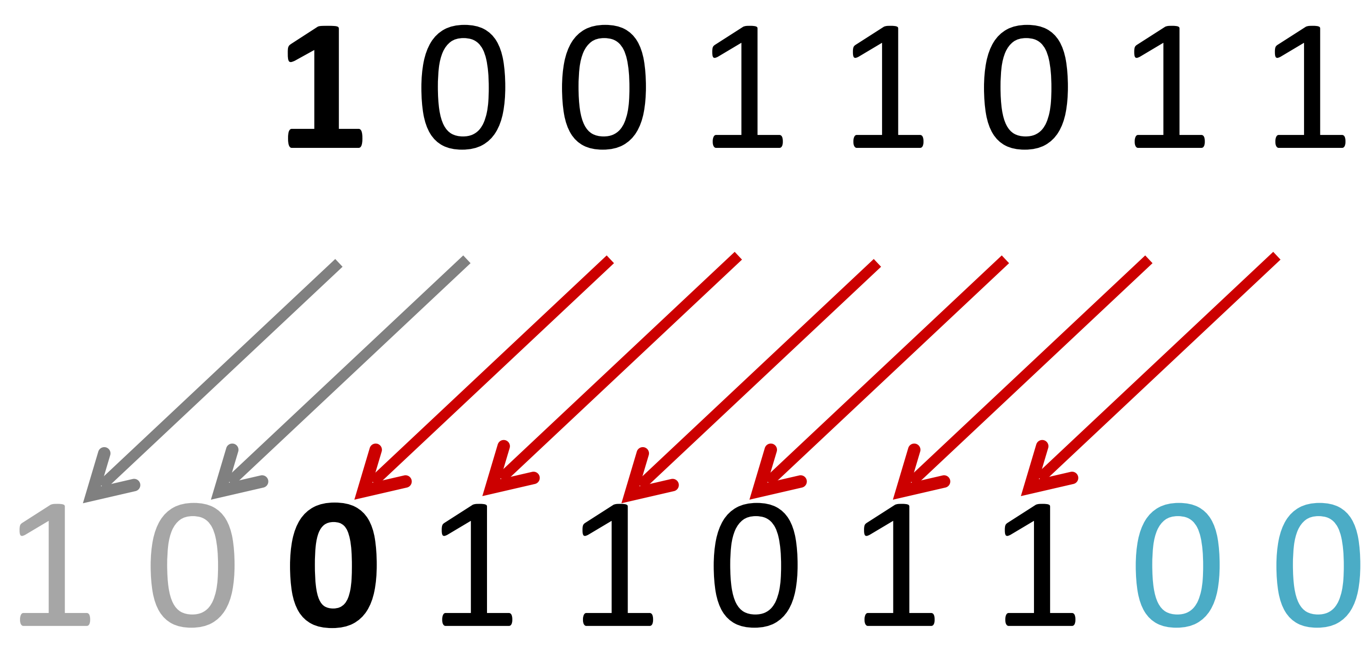
two's complement **shifting** and **arithmetic**

signed

x = -101;

y = x << 2;

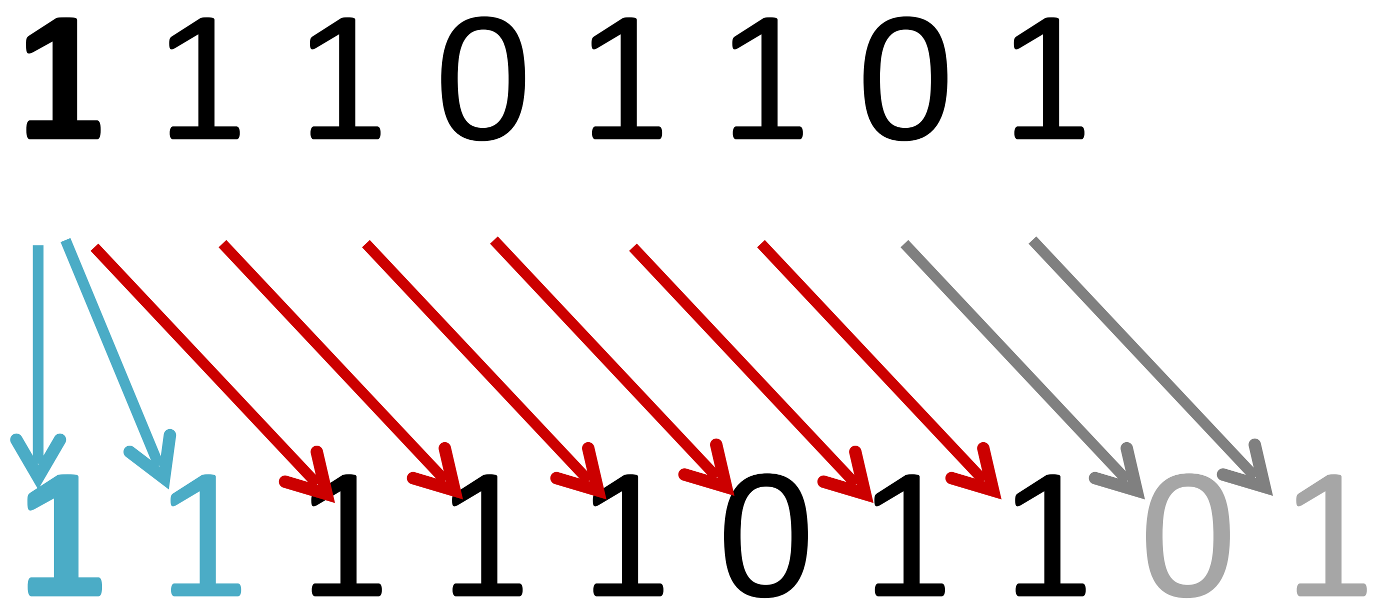
y == 108



logical shift left

n = shift distance in bits, w = width of encoding in bits

arithmetic shift right



signed

x = -19;

y = x >> 2;

y == -5

shift-*and*-***add***

ex

Available operations

`x << k` implements `x *` 2^k

`x + y`

Implement `y = x * 24` using only <<, +, and integer literals

```
y = x * (16 + 8);

y = (x * 16) + (x * 8);

y = (x << 4) + (x << 3)
```

Parenthesize shifts to be clear about precedence, which may not always be what you expect.

What does this function compute?

ex

```
unsigned puzzle(unsigned x, unsigned y) {
  unsigned result = 0;
  for (unsigned i = 0; i < 32; i++){
    if (y & (1 << i)) {
      result = result + (x << i);
    }
  }
  return result;
}
```

What does this function compute?

ex

Downsize to fake unsigned nybble type (4 bits) to make this easier to write…

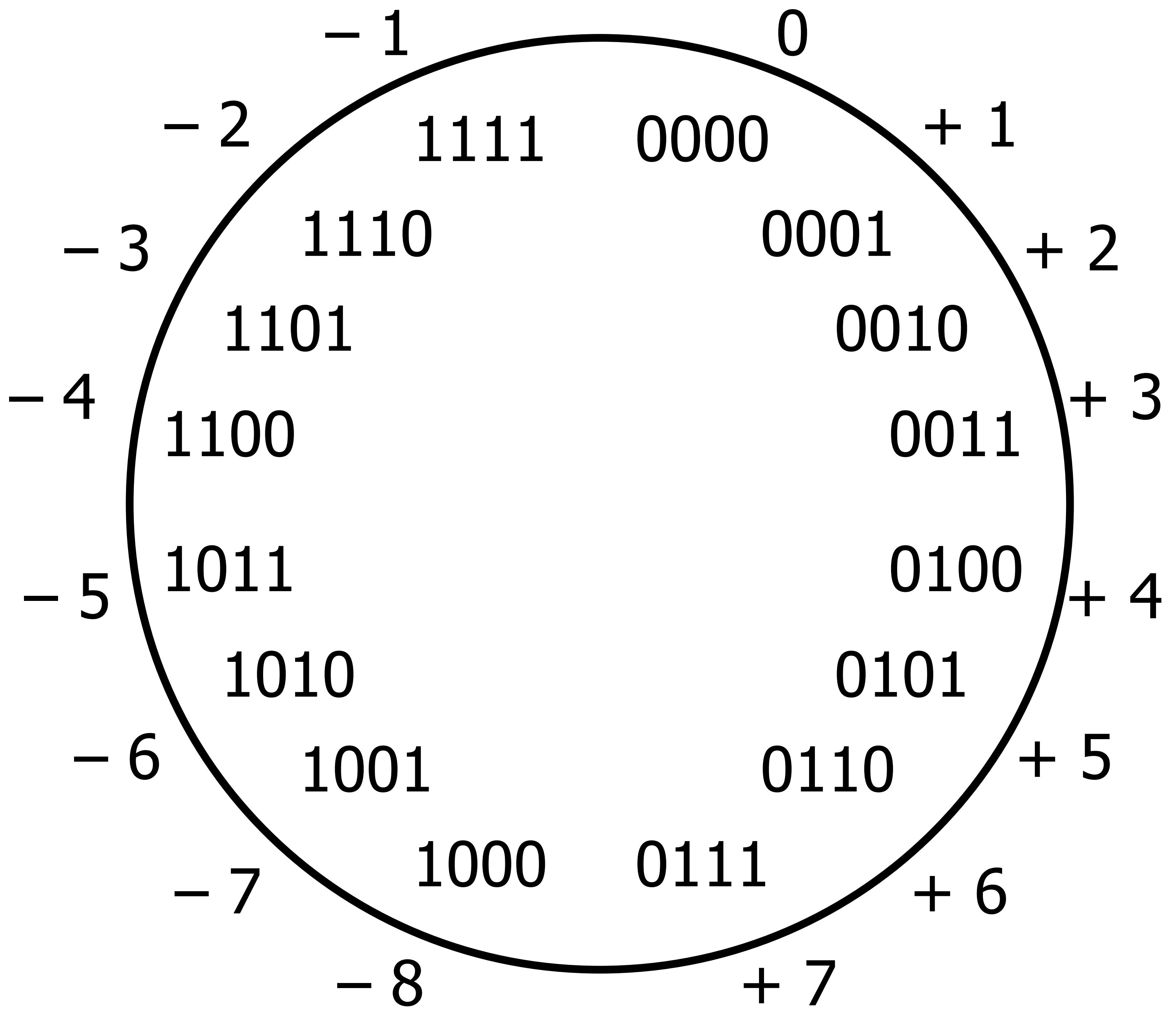
```
nybble puzzle(nybble x, nybble y) {
  nybble result = 0;
  for (nybble i = 0; i < 4; i++){
    if (y & (1 << i)) {
      result = result + (x << i);
    }
  }
  return result;
}
```

y_2	x_2

i_{10}	$y\&(1<<i)_2$	$result_2$
0		0 0 0 0
1		
2		
3		
4		

multiplication

2	0010
x 3	x 0011
6	00000110
-2	1110
x 2	x 0010
-4	11111100

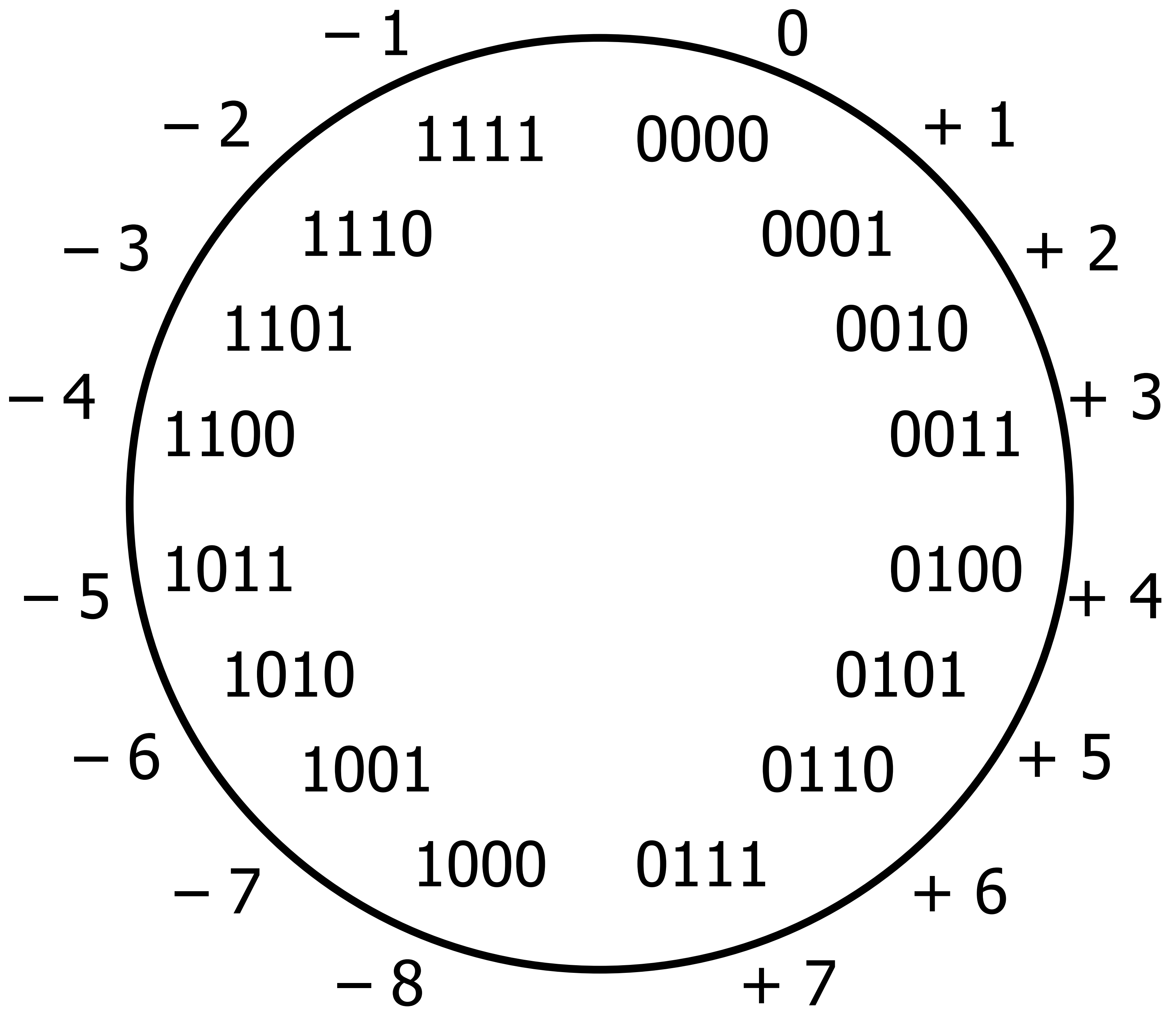


Modular Arithmetic

multiplication

5	0101
x 4	x 0100
~~20~~	00010100
4	

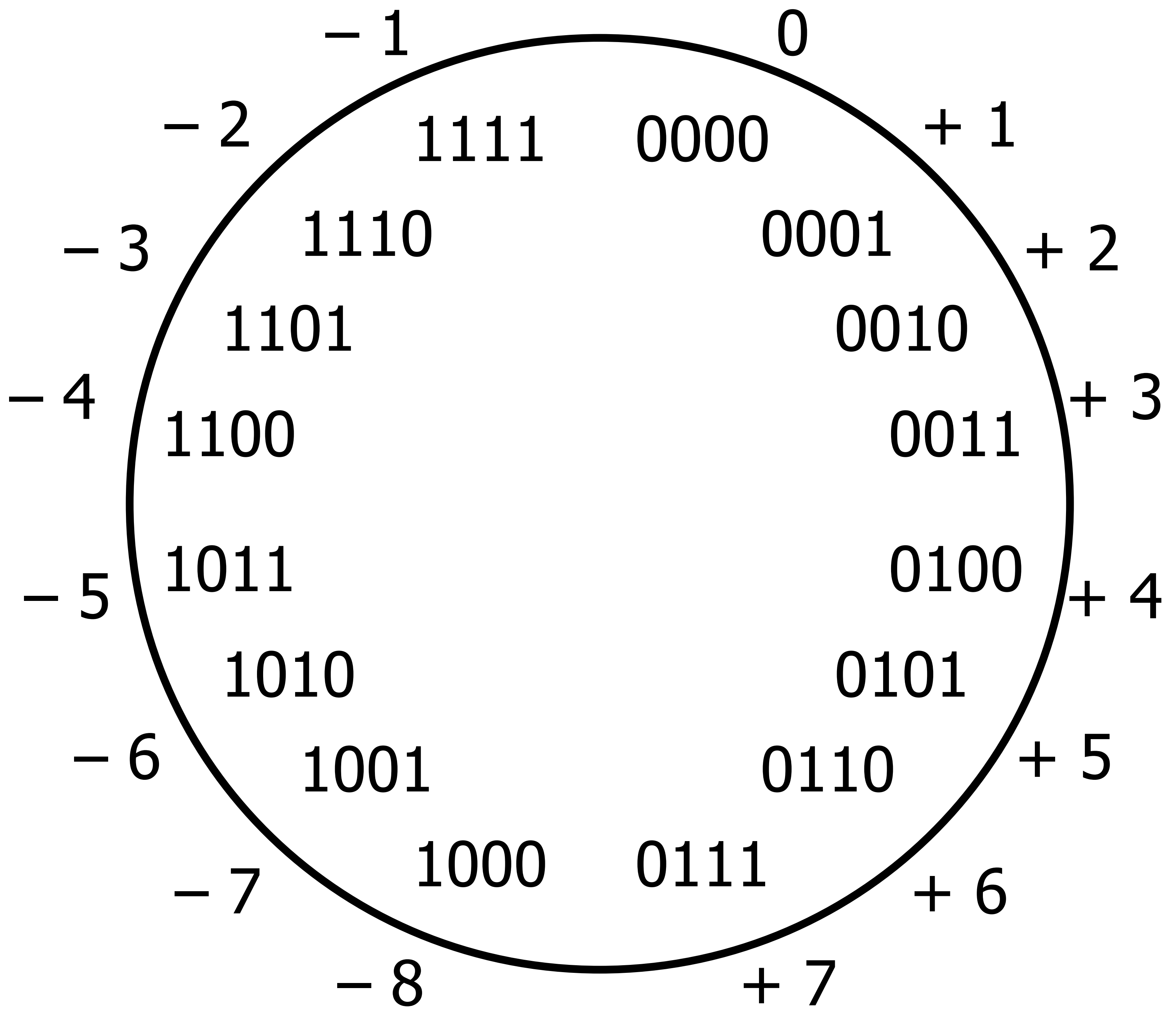
-3	1101
x 7	x 0111
~~-21~~	11101011
-5	



Modular Arithmetic

multiplication

5	0101
x 5	x 0101
~~25~~	00011001
-7	
-2	1110
x 6	x 0110
~~-12~~	11110100
4	



Modular Arithmetic

Casting Integers in C

!!!

Number literals: `37` is signed, `37U` is unsigned

Integer Casting: *bits unchanged, just reinterpreted.*

Explicit casting:

```
int tx = (int) 73U;          // still 73
unsigned uy = (unsigned) -4;  // big positive #
```

Implicit casting: **Actually does**

```
tx = ux;      // tx = (int)ux;
uy = ty;      // uy = (unsigned)ty;
void foo(int z) { ... }
foo(ux);      // foo((int)ux);
if (tx < ux) ...   // if ((unsigned)tx < ux) ...
```

More Implicit Casting in C

!!!

If you mix unsigned and signed in a single expression, then ***signed** values are **implicitly cast to** unsigned*.

How are the argument bits interpreted?

Argument$_1$	Op	Argument$_2$	Type	Result
`0`	`==`	`0U`	unsigned	1
`-1`	`<`	`0`	signed	1
`-1`	`<`	`0U`	unsigned	0
`2147483647`	`<`	`-2147483647-1`		
`2147483647U`	`<`	`-2147483647-1`		
`-1`	`<`	`-2`		
`(unsigned)-1`	`<`	`-2`		
`2147483647`	`<`	`2147483648U`		
`2147483647`	`<`	`(int)2147483648U`		

Note: T_{min} = -2,147,483,648 T_{max} = 2,147,483,647

T_{min} must be written as `-2147483647-1` (see pg. 77 of CSAPP for details)

Aside: real-world connection to Alexa’s research

Guest-controlled out-of-bounds read/write on x86_64

Critical **alexcrichton** published **GHSA-ff4p-7xrq-q5r8** on Mar 8

Package	Affected versions	Patched versions
cranelift-codegen (Rust)	<= 0.93.0, >= 0.84.0	0.93.1, 0.92.1, 0.91.1
wasmtime (Rust)	<= 6.0.0, >= 0.37.0	6.0.1, 5.0.1, 4.0.1

Description

Impact

Wasmtime's code generator, Cranelift, has a bug on x86_64 targets where address-mode computation mistakenly would calculate a 35-bit effective address instead of WebAssembly's defined 33-bit effective address. This bug means that, with default codegen settings, a wasm-controlled load/store operation could read/write addresses up to 35 bits away from the base of linear memory. Wasmtime's default sandbox settings provide up to 6G of protection from the base of linear memory to guarantee that any memory access in that range will be semantically correct. Due to this bug, however, addresses up to `0xffffffff * 8 + 0x7ffffffc = 36507222004 = ~34G` bytes away from the base of linear memory are possible from guest code. This means that the virtual memory 6G away from the base of linear memory up to ~34G away can be read/written by a malicious module.

Severity

Critical 9.9 / 10

CVSS base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	Low
User interaction	None
Scope	Changed
Confidentiality	High
Integrity	High
Availability	High

CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H

CVE ID

CVE-2023-26489

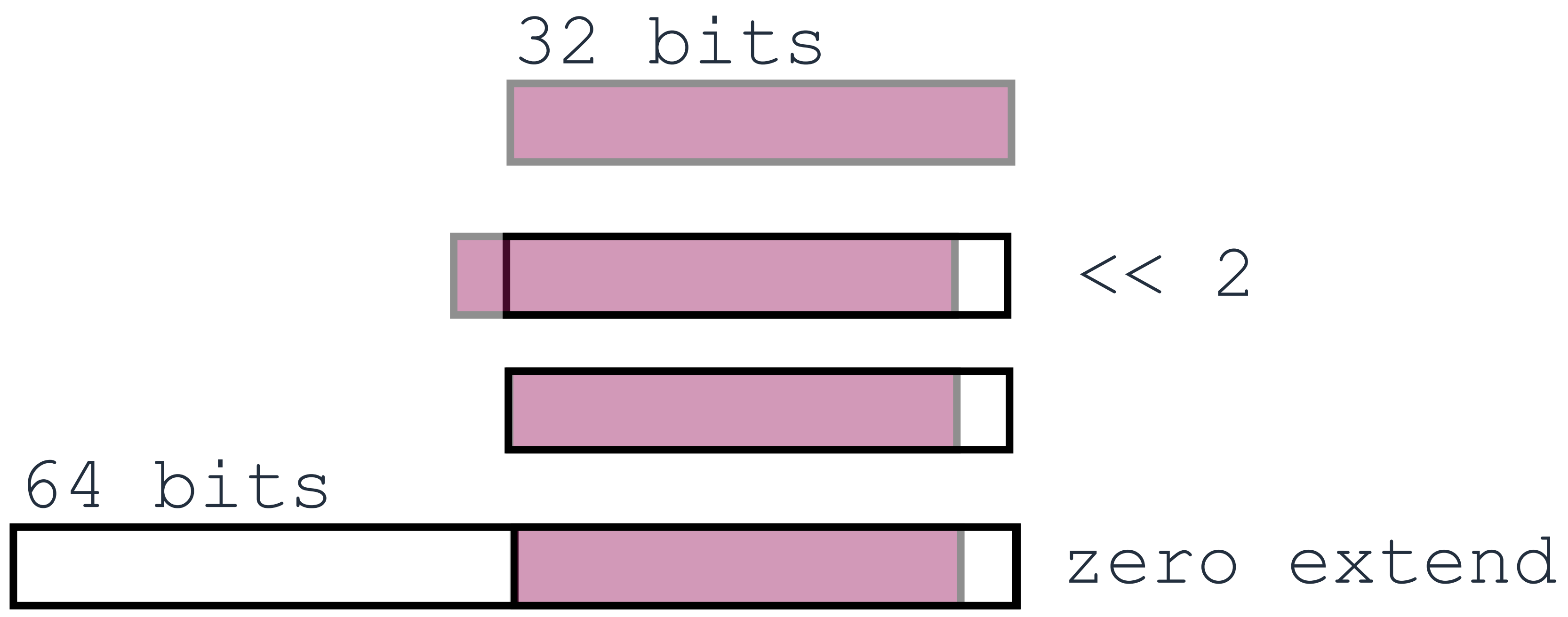
Security-critical bug in shift-and-extend code

Guest-controlled out-of-bounds read/write on x86_64

GHSA-ff4p-7xrq-q5r8 published on Mar 8 by alexcrichton

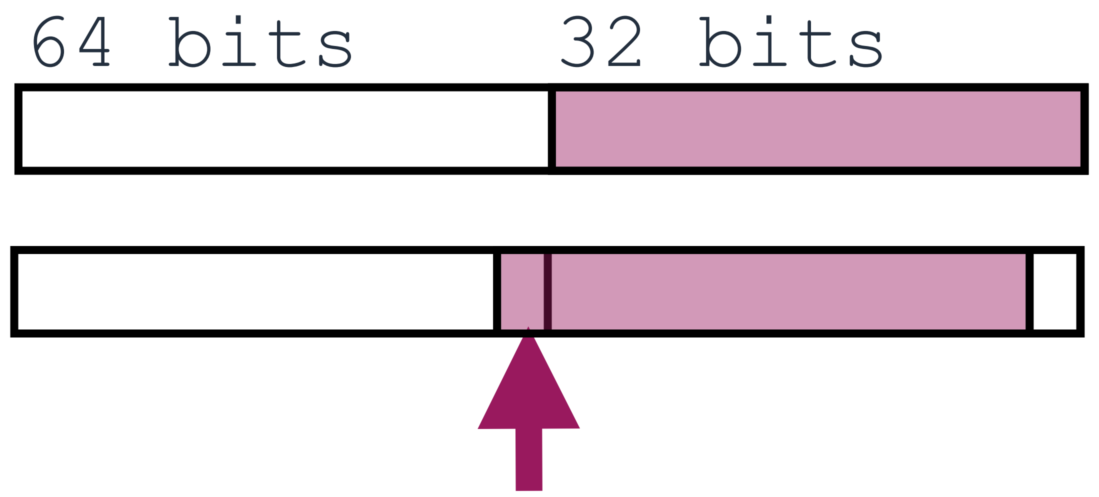
Conceptually, the compiler tried to convert this with a 32-bit **x**:

```
address + zero_extend_64(x << 2)
```



To this:

```
address + (zero_extend_64(x) << 2)
```



Incorrect address calculated!