

numbers

base-10 numbers

$$12345 = 1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$$

$$987.65 = 9 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

base-2 numbers

$$\begin{aligned}20_{\text{TEN}} \text{ (or } 20_{10}) &= 11101_{\text{TWO}} \text{ (or } 11101_2) \\&= 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\4_{\text{TEN}} &= 100_{\text{TWO}} \\&= 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\1.25_{\text{TEN}} &= 1.01_{\text{TWO}} \\&= 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}\end{aligned}$$

base-16 numbers

0 1 2 3 4 5 6 7 8 9 A B C D E F

$$15_{\text{TEN}} =$$

$$F_{\text{SIXTEEN}} =$$

$$15 \cdot 16^0$$

$$100_{\text{TEN}} =$$

$$64_{\text{SIXTEEN}} =$$

$$6 \cdot 16^1 + 4 \cdot 16^0$$

$$0.5_{\text{TEN}} =$$

$$0.8_{\text{SIXTEEN}} =$$

$$8 \cdot 16^{-1}$$

integers in C++

15 _{TEN}		15
17 _{EIGHT}		017
F _{SIXTEEN}		0xF

99 _{TEN}		99
143 _{EIGHT}		0143
63 _{SIXTEEN}		0x63

16 _{TEN}		16
20 _{EIGHT}		020
10 _{SIXTEEN}		0x10

terminology

base- X number — X is the **radix**

I will call components of base X number 'digits'

but not a great term — digit sometimes implies base-10
sometimes "radit"

base-2 digit = bit

base-16 digit = nibble (sometimes)

base-10 = decimal

base-2 = binary

base-8 = octal

base-16 = hexadecimal

convert to decimal

$$42_{\text{FIVE}} =$$

$$121_{\text{THREE}} =$$

convert to decimal

$$42_{\text{FIVE}} = 4 \cdot 5^1 + 2 \cdot 5^0$$
$$=$$

$$121_{\text{THREE}} =$$

convert to decimal

$$\begin{aligned} 42_{\text{FIVE}} &= 4 \cdot 5^1 + 2 \cdot 5^0 \\ &= 20_{\text{TEN}} + 2 = 22_{\text{TEN}} \end{aligned}$$

$$121_{\text{THREE}} =$$

convert to decimal

$$\begin{aligned} 42_{\text{FIVE}} &= 4 \cdot 5^1 + 2 \cdot 5^0 \\ &= 20_{\text{TEN}} + 2 = 22_{\text{TEN}} \end{aligned}$$

$$\begin{aligned} 121_{\text{THREE}} &= 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 \\ &= \end{aligned}$$

convert to decimal

$$\begin{aligned} 42_{\text{FIVE}} &= 4 \cdot 5^1 + 2 \cdot 5^0 \\ &= 20_{\text{TEN}} + 2 = 22_{\text{TEN}} \end{aligned}$$

$$\begin{aligned} 121_{\text{THREE}} &= 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 \\ &= 9 + 6 + 1 = 16_{\text{TEN}} \end{aligned}$$

convert to something (1)

$$42_{\text{TEN}} \text{ as radix } 5 =$$

convert to something (1)

$$42_{\text{TEN}} \text{ as radix } 5 = \underline{\quad} 2$$

$$42 \div 5 = 8 + \dots$$

$$42 \bmod 5 = 2$$

$$42 = 8 \cdot 5 + 2$$

convert to something (1)

$$42_{\text{TEN}} \text{ as radix } 5 = \underline{\textcolor{red}{3}}2$$

$$42 \div 5 = 8 + \dots$$

$$42 \bmod 5 = 2$$

$$42 = 8 \cdot 5 + 2$$

$$8 = 1 \cdot 5 + \textcolor{red}{3}$$

convert to something (1)

$$42_{\text{TEN}} \text{ as radix } 5 = \textcolor{red}{1}32_{\text{FIVE}}$$

$$42 \div 5 = 8 + \dots$$

$$42 \bmod 5 = 2$$

$$42 = 8 \cdot 5 + 2$$

$$8 = 1 \cdot 5 + 3$$

$\textcolor{red}{1}$

convert to something (2)

$$121_{\text{TEN}} \text{ as radix } 11 =$$

convert to something (2)

$$121_{\text{TEN}} \text{ as radix } 11 = \underline{\quad} 0_{\text{ELEVEN}}$$

$$121 \div 11 = 11$$

$$121 \bmod 11 = 0$$

$$121 = 11 \cdot 11 + 0$$

convert to something (2)

$$121_{\text{TEN as radix 11}} = \underline{}\textcolor{red}{0}0_{\text{ELEVEN}}$$

$$121 \div 11 = 11$$

$$121 \bmod 11 = 0$$

$$121 = 11 \cdot 11 + 0$$

$$11 = 1 \cdot 11 + \textcolor{red}{0}$$

convert to something (2)

$$121_{\text{TEN as radix 11}} = \textcolor{red}{1}00_{\text{ELEVEN}}$$

$$121 \div 11 = 11$$

$$121 \bmod 11 = 0$$

$$121 = 11 \cdot 11 + 0$$

$$11 = 1 \cdot 11 + 0$$

$\textcolor{red}{1}$

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

$$\begin{aligned}uz_{\text{SIXTEEN}} &= u \cdot 16^1 + z \cdot 16^0 \\&= (u_3 \cdot 2^3 + u_2 \cdot 2^2 + u_1 \cdot 2^1 + u_0 \cdot 2^0)2^4 + z_3 \cdot 2^3 + \dots \\&= u_3 \cdot 2^7 + u_2 \cdot 2^6 + u_1 \cdot 2^5 + u_0 \cdot 2^4 + z_3 \cdot 2^3 + \dots \\&= (u_3u_2u_1u_0z_3z_2z_1z_0)_{\text{TWO}}\end{aligned}$$

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

1		2		3		4 _{SIXTEEN}
---	--	---	--	---	--	----------------------

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

1	2	3	4 _{SIXTEEN}
0001	0010	0011	0100 _{TWO}

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

1	2	3	4 _{SIXTEEN}
0001	0010	0011	0100 _{TWO}

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

1	2	3	4 _{SIXTEEN}
0001	0010	0011	0100 _{TWO}

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

1	2	3	4 _{SIXTEEN}
0001	0010	0011	0100 _{TWO}

1101	1110	0011	0000 _{TWO}
------	------	------	---------------------

special case: base-16 to base-2

each “nibble” (hexadecimal digit) = 4 binary bits

1	2	3	4 _{SIXTEEN}
0001	0010	0011	0100 _{TWO}

1101	1110	0011	0000 _{TWO}
C	D	3	0 _{SIXTEEN}

a note on bytes

one byte = one “octet” =
two nibbles (hexadecimal digits) =
eight bits

this class — byte is always eight bits
(some very old machines called different sizes “bytes”)

a note on bytes

one byte = one “octet” =
two nibbles (hexadecimal digits) =
eight bits

this class — byte is always eight bits
(some very old machines called different sizes “bytes”)

a note on bytes

one byte = one “octet” =
two nibbles (hexadecimal digits) =
eight bits

this class — byte is always eight bits
(some very old machines called different sizes “bytes”)

exercise

$$17_{\text{NINE}} = ?_{\text{SEVEN}}$$

exercise

$$17_{\text{NINE}} = ?_{\text{SEVEN}}$$

$$17_{\text{NINE}} = 7 + 9 = 2 \cdot 7 + 2$$

$$17_{\text{NINE}} = 22_{\text{SEVEN}}$$

on math in other bases

you can do math in other bases

usually makes most sense for base 2...

$$\begin{array}{rcccccc} & & 1 & 1 & 1 & & \\ & 1 & 2 & 3 & 4 & 4_{\text{SIXTEEN}} & \\ \times & & & & 1 & 5_{\text{SIXTEEN}} & \\ \hline 5 & B & 0 & 5 & 4 & & \\ 1 & 2 & 3 & 4 & 4 & & \\ \hline 1 & 7 & E & 4 & 9 & 4_{\text{SIXTEEN}} & \end{array}$$

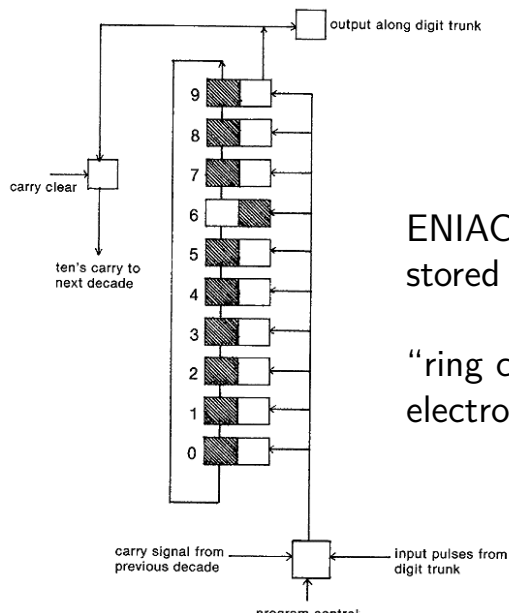
```
$ python3 -c 'print("{:x}".format(0x12344*0x15))'
17e494
```


integer representation

modern machine represent integers as series of **bits** (base-2)

why not base-10?

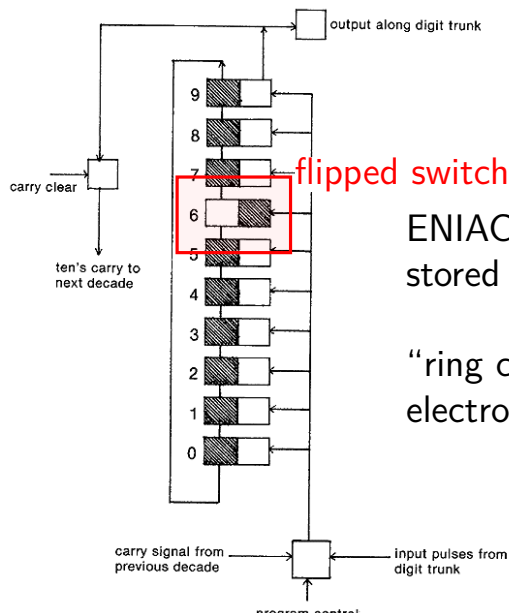
ENIAC: base-10 representation



ENIAC: 1946 computer
stored base-10 digits

“ring counter” of ten
electronic switches per digit

ENIAC: base-10 representation

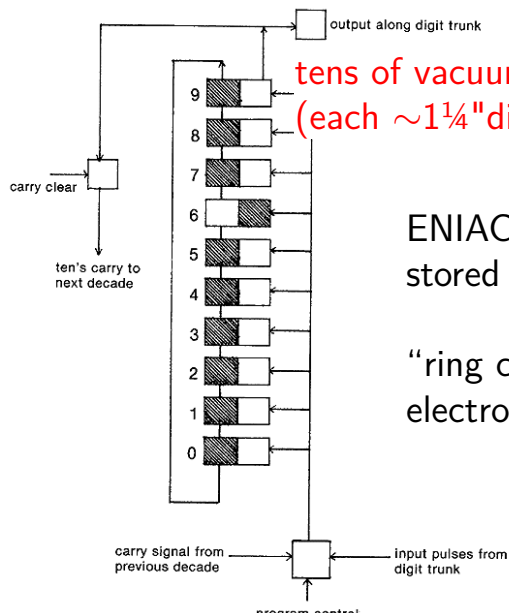


flipped switch indicates digit stored

ENIAC: 1946 computer
stored base-10 digits

“ring counter” of ten
electronic switches per digit

ENIAC: base-10 representation



tens of vacuum tubes total
(each $\sim 1\frac{1}{4}$ " diameter by $2\frac{3}{4}$ " height)

ENIAC: 1946 computer
stored base-10 digits

"ring counter" of ten
electronic switches per digit

base-2 representation

base 2 — each switch represents one “digit”

much more efficient use of switches

used in some pre-ENIAC electronic computers

Atanasoff-Berry computer (1937, Ohio State)

Z3 (1941, German Laboratory for Aviation)

base-2 representation

base 2 — each switch represents one “digit”

much more efficient use of switches

used in some pre-ENIAC electronic computers

Atanasoff-Berry computer (1937, Ohio State)

Z3 (1941, German Laboratory for Aviation)

why not used in ENIAC?

Eckert (ENIAC designer), 1953: “Although [binary-based digit counters] were known at the time of the construction of the ENIAC, it was not used because it required stable resistors, which were then much more expensive than they are now.”

also, important to input/output decimal digits directly

base-2 bit addition

+	0	1
0	00	01
1	01	10

base-2 bit addition

+	0	1
0	00	01
1	01	10

exactly one set to 1 — result (w/o carry) is 1; otherwise 0

base-2 bit addition

+	0	1
0	00	01
1	01	10

exactly one set to 1 — result (w/o carry) is 1; otherwise 0

both set to 1 — carry is 1; otherwise 0

base-2 capacity

$$\begin{aligned} n\text{-bit number:} \quad & b_{n-1}b_{n-2}b_{n-3} \dots b_2b_1b_0 \\ &= \sum_{i=0}^{n-1} b_i \cdot 2^i \\ &\leq \sum_{i=0}^{n-1} 1 \cdot 2^i = 2^n - 1 \end{aligned}$$

base-2 capacity

$$\begin{aligned} n\text{-bit number: } & b_{n-1}b_{n-2}b_{n-3} \dots b_2b_1b_0 \\ = & \sum_{i=0}^{n-1} b_i \cdot 2^i \\ \leq & \sum_{i=0}^{n-1} 1 \cdot 2^i = 2^n - 1 \end{aligned}$$

base-2 capacity

$$\begin{aligned} n\text{-bit number: } & b_{n-1}b_{n-2}b_{n-3} \dots b_2b_1b_0 \\ &= \sum_{i=0}^{n-1} b_i \cdot 2^i \\ &\leq \sum_{i=0}^{n-1} 1 \cdot 2^i = 2^n - 1 \end{aligned}$$

missing pieces:

negative numbers?

non-whole numbers?

what is n ?

base-2 capacity

$$\begin{aligned} n\text{-bit number: } & b_{n-1}b_{n-2}b_{n-3} \dots b_2b_1b_0 \\ &= \sum_{i=0}^{n-1} b_i \cdot 2^i \\ &\leq \sum_{i=0}^{n-1} 1 \cdot 2^i = 2^n - 1 \end{aligned}$$

missing pieces:

negative numbers?

non-whole numbers?

what is n ?

integer size in C++

varies between machines

compiler uses what makes most sense on each machine?

type	size in bits	
	minimum	on lab machines
unsigned char	8	8
unsigned short	16	16
unsigned int	16	32
unsigned long	32	64

integer size in C++

varies between machines

compiler uses what makes most sense on each machine?

type		size in bits	
		minimum	on lab machines
unsigned	char	8	8
unsigned	short	16	16
unsigned	int	16	32
unsigned	long	32	64

“unsigned” — can't be negative (no \pm sign)

integer size in C++

varies between machines

compiler uses what makes most sense on each machine?

type	size in bits	
	minimum	on lab machines
unsigned char	8	8
unsigned short	16	16
unsigned int	16	32
unsigned long	32	64

minimum size required by standard for all C++ compilers
all allowed to be bigger

querying sizes in C++

```
#include <climits>  // C: <limits.h>
...
ULONG_MAX or UINT_MAX or USHRT_MAX or UCHAR_MAX
// e.g. USHRT_MAX == 65535 on lab machines
```

```
#include <limits>
...
std::numeric_limits<unsigned long>::max()
    // == ULONG_MAX
...
```

```
sizeof(unsigned long) // number of *bytes*
    // == 8 on lab machines
...
```

numbering bits

option 1: n -bit number:

$$b_{n-1}b_{n-2}b_{n-3}\dots b_2b_1b_0$$
$$= \sum_{i=0}^{n-1} b_i \cdot 2^i$$

option 2: n -bit number:

$$b_0b_1b_2\dots b_{n-3}b_{n-2}b_{n-1}$$
$$= \sum_{i=0}^{n-1} b_i \cdot 2^{n-i-1}$$

numbering bits

option 1: n -bit number:

$$b_{n-1}b_{n-2}b_{n-3}\dots b_2b_1b_0$$
$$= \sum_{i=0}^{n-1} b_i \cdot 2^i$$

option 2: n -bit number:

$$b_0b_1b_2\dots b_{n-3}b_{n-2}b_{n-1}$$
$$= \sum_{i=0}^{n-1} b_i \cdot 2^{n-i-1}$$

two viable ways to number bits

numbering bits

option 1: n -bit number:

$$b_{n-1}b_{n-2}b_{n-3}\dots b_2b_1b_0$$
$$= \sum_{i=0}^{n-1} b_i \cdot 2^i$$

option 2: n -bit number:

$$b_0b_1b_2\dots b_{n-3}b_{n-2}b_{n-1}$$
$$= \sum_{i=0}^{n-1} b_i \cdot 2^{n-i-1}$$

two viable ways to number bits

does it matter which I use?

do I have a way to ask for bit i ?

numbering bytes

option 1: 4-byte number: $B_3B_2B_1B_0$

$$= \sum_{i=0}^3 B_i \cdot 256^i$$

option 2: 4-byte number: $B_0B_1B_2B_3$

$$= \sum_{i=0}^3 b_i \cdot 256^{3-i}$$

numbering bytes

option 1: 4-byte number: $B_3B_2B_1B_0$

$$= \sum_{i=0}^3 B_i \cdot 256^i$$

option 2: 4-byte number: $B_0B_1B_2B_3$

$$= \sum_{i=0}^3 b_i \cdot 256^{3-i}$$

two viable ways to number bytes

numbering bytes

option 1: 4-byte number: $B_3B_2B_1B_0$

$$= \sum_{i=0}^3 B_i \cdot 256^i$$

option 2: 4-byte number: $B_0B_1B_2B_3$

$$= \sum_{i=0}^3 b_i \cdot 256^{3-i}$$

two viable ways to number bytes

does it matter which I use?

in memory, yes — each byte needs an address (number)

memory

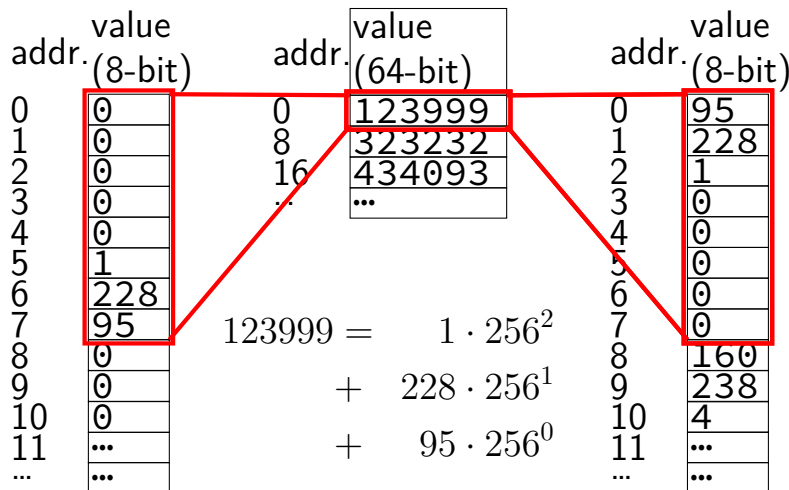
memory
(as 64-bit values)

addr.	value (64-bit)
0	123999
8	323232
16	434093
...	...

$$\begin{aligned} 123999 &= 1 \cdot 256^2 \\ &+ 228 \cdot 256^1 \\ &+ 95 \cdot 256^0 \end{aligned}$$

memory

if **big endian** memory if **little endian**
(as 8-bit values) (as 64-bit values) (as 8-bit values)



finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

little endian (e.g. lab machine):

123456789abcdef
ef cd ab 89 67 45 23 1

big endian:

123456789abcdef
1 23 45 67 89 ab cd ef

finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

get pointer to **byte** with
lowest address in value

little endian (e.g. lab m
123456789abcdef
ef cd ab 89 67 45 23 1

big endian:

123456789abcdef
1 23 45 67 89 ab cd ef

finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

unless you do something like this
won't see endianness

little endian (e.g. ia

123456789abcdef

ef cd ab 89 67 45 23 1

big endian:

123456789abcdef

1 23 45 67 89 ab cd ef

finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

use pointer to get *i*th byte of value
(cast to int to output as number, not character)

little endian
123456789abcdef
ef cd ab 89 67 45 23 1

big endian:

123456789abcdef
1 23 45 67 89 ab cd ef

finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

little endian: byte 0 is **least significant**
(affects overall value the least)

little endian (e.g.

123456789abcdef

ef cd ab 89 67 45 23 1

big endian:

123456789abcdef

1 23 45 67 89 ab cd ef

finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

big endian: byte 0 is **most significant**
(affects overall value the most)

little endian (e.g.

123456789abcdef
ef cd ab 89 67 45 23 1

big endian:

123456789abcdef
1 23 45 67 89 ab cd ef

finding endianness in C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
int main() {
    unsigned long value = 0x0123456789ABCDEF;
    cout << hex << value << endl;
    unsigned char *ptr = (unsigned char*) &value;
    for (int i = 0; i < sizeof(unsigned long); ++i) {
        cout << (int) ptr[i] << "_";
    }
    ...
}
```

but we don't write numbers in a different order
based on which end we call "part 0"

little endian

123456789abcdef

ef cd ab 89 67 45 23 1

big endian:

123456789abcdef

1 23 45 67 89 ab cd ef

little versus big endian

little endian — least significant part has lowest address

i.e. index 0 is the one's place

big endian — most significant part has the lowest address

i.e. index $n - 1$ is the one's place

endianness in the real world

today and this course: little endian is dominant

e.g. x86, *typically* ARM

historically: big endian was dominant

e.g. *typically* SPARC, POWER, Alpha, MIPS, ...
still commonly used for networking because of this

many architectures have switchable endianness

e.g. ARM, SPARC, POWER, MIPS
usually, OS chooses one endianness

middle endian

sometimes not just big/little endian

e.g. number bytes most to least significant as
5, 6, 7, 8, 1, 2, 3, 4

e.g. doubles on little-endian ARM

generally some sort of historical accident

e.g. ARM floating point designed for big endian?

endianness is about addresses

endianness is about numbering,
not (necessairily) placement on the page

but, probably assume English order (left to right, etc.) if not
otherwise specified

addr.	value		addr.	value
0	95	=
1	228		11	...
2	1		10	4
3	0		9	238
4	0		8	160
5	0		7	0
6	0		6	0
7	0		5	0
8	160		4	0
9	238		3	0
10	4		2	1
11	...		1	228

endianness and bit-order

we won't talk about **bit order**

because bits don't have addresses

if I say “bit 0”, question: “numbering from least significant or most significant”?

nothing about how pointers, etc. work suggests either answer is correct

endianness and writing out bytes

0x0102 in binary: 000000001000000010

English's order — most significant first

bytes of 0x0102 in big endian:

(byte 0) 000000001 (byte 1) 00000010

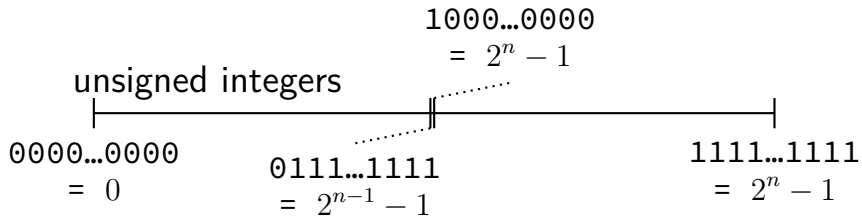
bytes of 0x0102 in little endian:

(byte 0) 00000010 (byte 1) 000000001

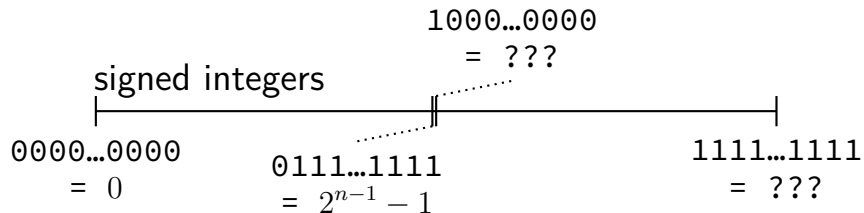
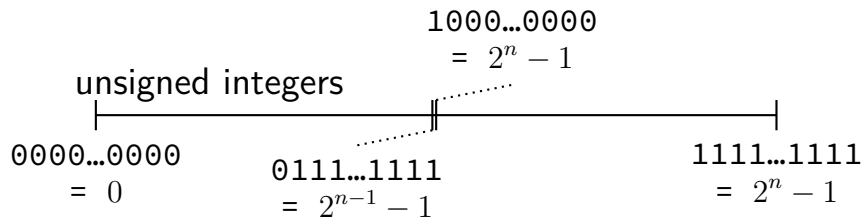
usually, we don't change the order we write bits

if writing out bytes, first in reading order is usually lowest address
(we'll specify if not)

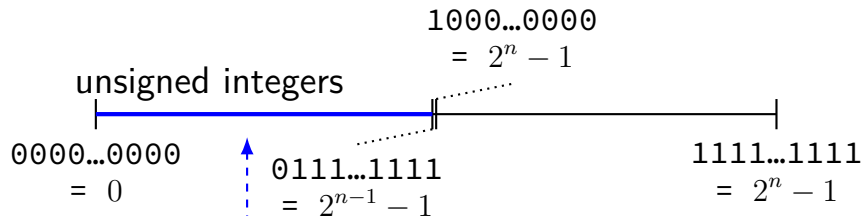
representing negative numbers



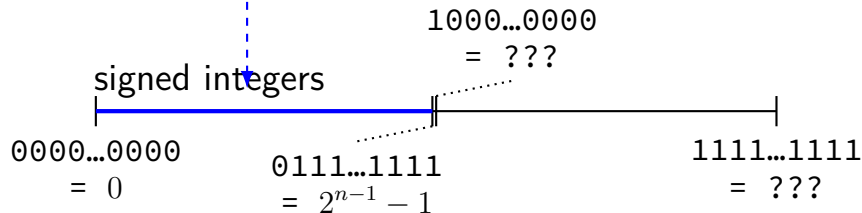
representing negative numbers



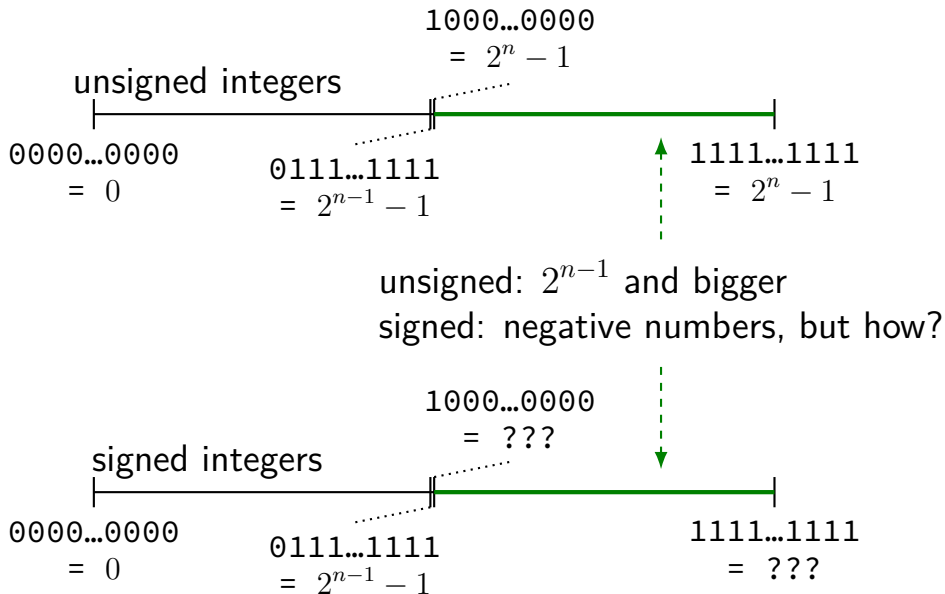
representing negative numbers



positive numbers up to $2^n - 1$
goal: same bits, signed or not

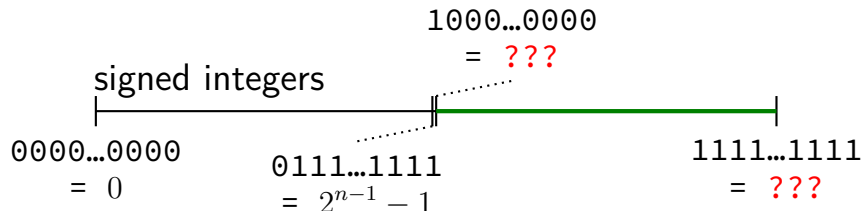


representing negative numbers



representing negative numbers

	sign & magnitude	1's complement	2's complement
000...000	0	0	0
011...111	$2^{n-1} - 1$	$2^{n-1} - 1$	$2^{n-1} - 1$
100...000	0	$-2^{n-1} + 1$	-2^{n-1}
111...111	$-2^{n-1} + 1$	0	-1

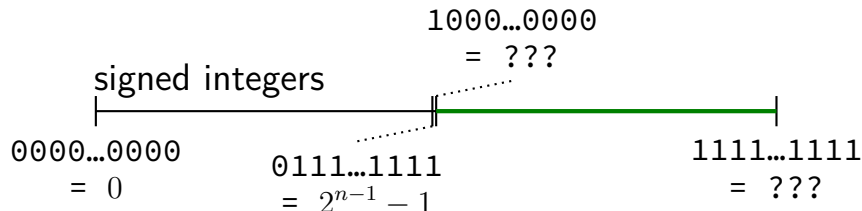


representing negative numbers

	sign & magnitude	1's complement	2's complement
000...000	0	0	0
011...111	$2^{n-1} - 1$	$2^{n-1} - 1$	$2^{n-1} - 1$
100...000	0	$-2^{n-1} + 1$	-2^{n-1}
111...111	$-2^{n-1} + 1$	0	-1

two representations of zero?

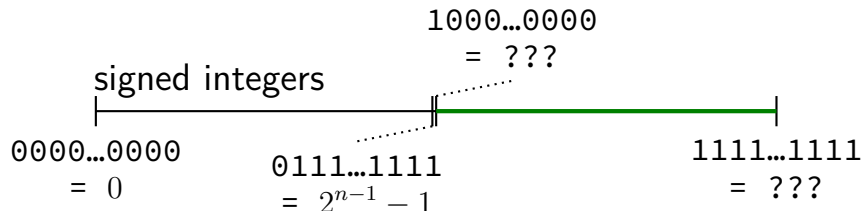
$x == y$ needs to do something special



representing negative numbers

	sign & magnitude	1's complement	2's complement
000...000	0	0	0
011...111	$2^{n-1} - 1$	$2^{n-1} - 1$	$2^{n-1} - 1$
100...000	0	$-2^{n-1} + 1$	-2^{n-1}
111...111	$-2^{n-1} + 1$	0	-1

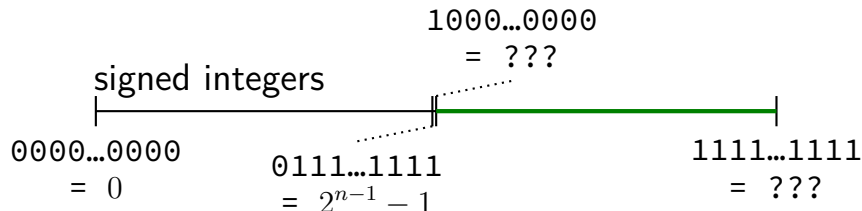
more negative values than positive values?



representing negative numbers

	sign & magnitude	1's complement	2's complement
000...000	0	0	0
011...111	$2^{n-1} - 1$	$2^{n-1} - 1$	$2^{n-1} - 1$
100...000	0	$-2^{n-1} + 1$	-2^{n-1}
111...111	$-2^{n-1} + 1$	0	-1

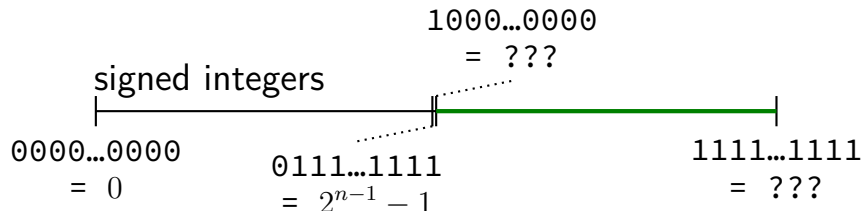
all 1's — least negative?



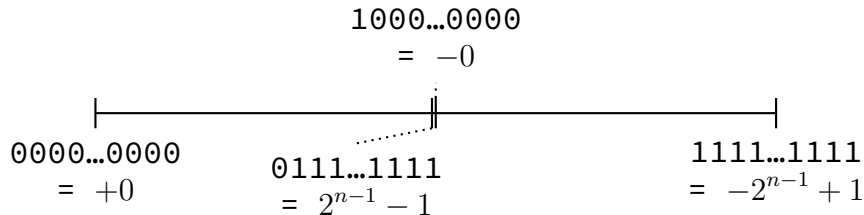
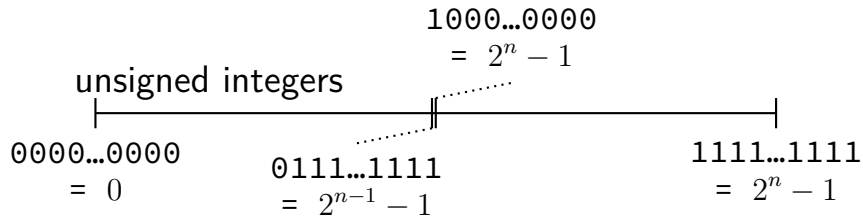
representing negative numbers

	sign & magnitude	1's complement	2's complement
000...000	0	0	0
011...111	$2^{n-1} - 1$	$2^{n-1} - 1$	$2^{n-1} - 1$
100...000	0	$-2^{n-1} + 1$	-2^{n-1}
111...111	$-2^{n-1} + 1$	0	-1

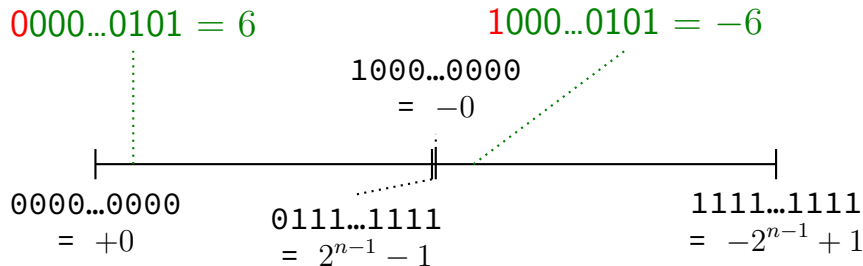
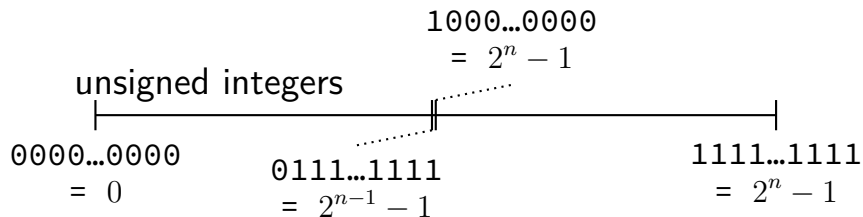
all 1's — most negative?



sign and magnitude

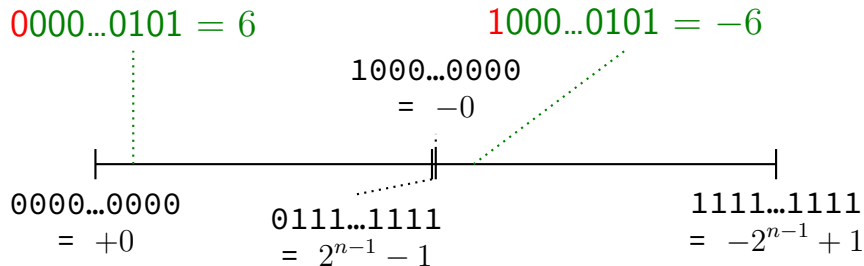


sign and magnitude



sign and magnitude

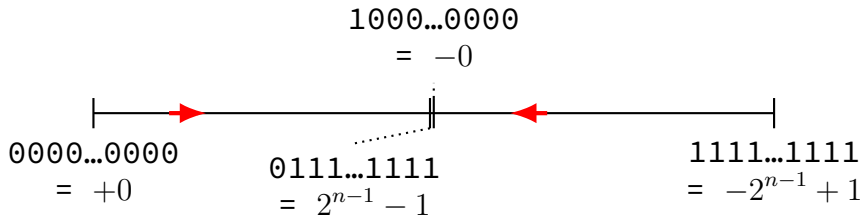
first bit is “sign bit” — 0 = positive, 1 = negative
flip sign bit to negate number



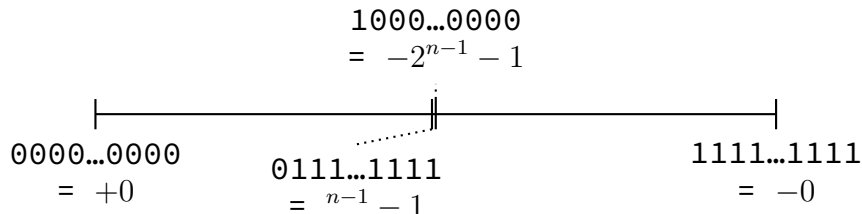
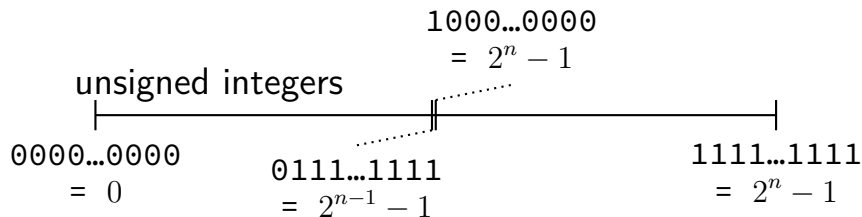
sign and magnitude

adding 1

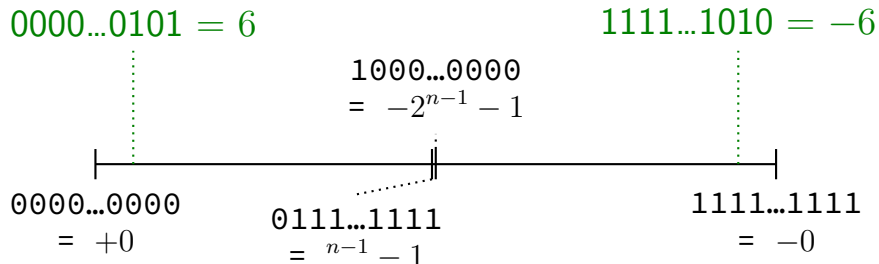
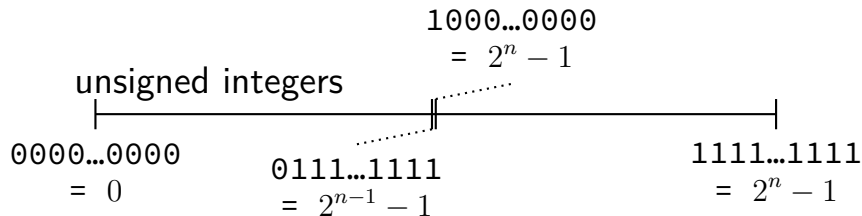
different direction if negative



1's complement



1's complement



1's complement

flip all bits to negate number

$$0000\dots0101 = 6$$

$$1111\dots1010 = -6$$

$$1000\dots0000 \\ = -2^{n-1} - 1$$

$$0000\dots0000 \\ = +0$$

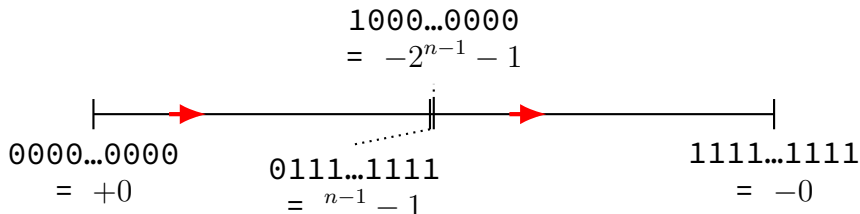
$$0111\dots1111 \\ = 2^{n-1} - 1$$

$$1111\dots1111 \\ = -0$$

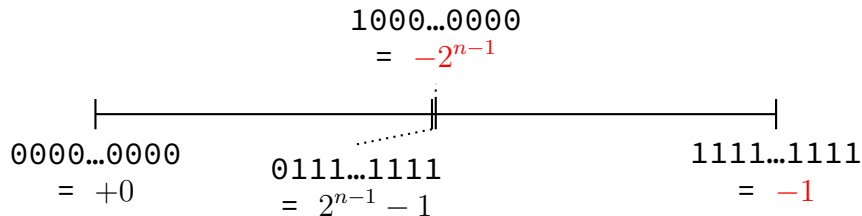
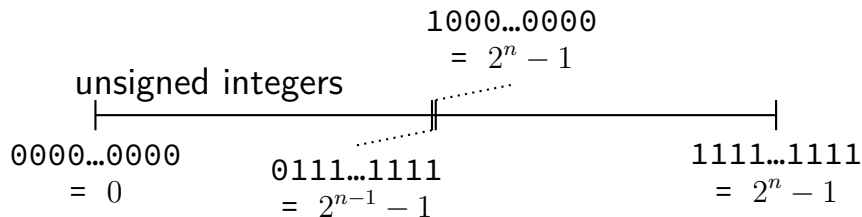
1's complement

adding 1

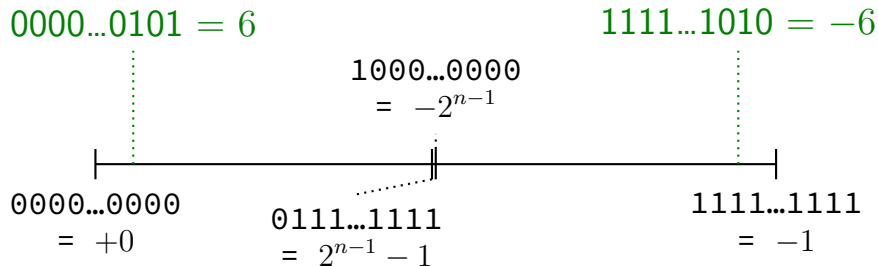
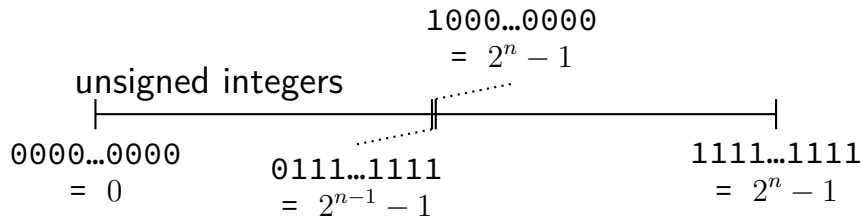
same direction, no matter original sign



two's complement



two's complement



two's complement

flip all bits and add 1 to negate number

$$0000\dots0101 = 6$$

$$1111\dots1010 = -6$$

$$1000\dots0000 = -2^{n-1}$$

$$0000\dots0000 = +0$$

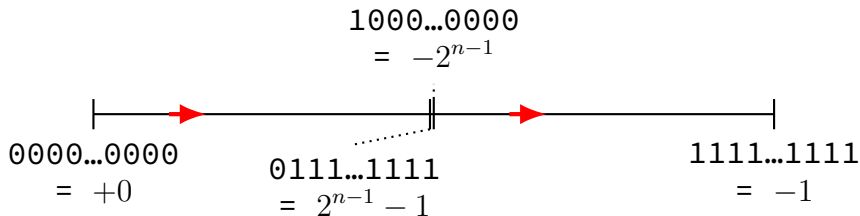
$$0111\dots1111 = 2^{n-1} - 1$$

$$1111\dots1111 = -1$$

two's complement

adding 1

same direction, no matter original sign



2's complement (alt. perspective)

2's complement (5 bit)

$$+10 = \quad 0 \quad 1 \quad 0 \quad 1 \quad 0$$

$$0 \cdot (-2^4) + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

$$0 + 2^3 + 0 + 2^1 + 0 = 10$$

$$-10 = \quad 1 \quad 0 \quad 1 \quad 1 \quad 0$$

$$1 \cdot (-2^4) + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

$$-2^4 + 0 + 2^2 + 2^1 + 0 = -10$$

2's complement (alt. perspective)

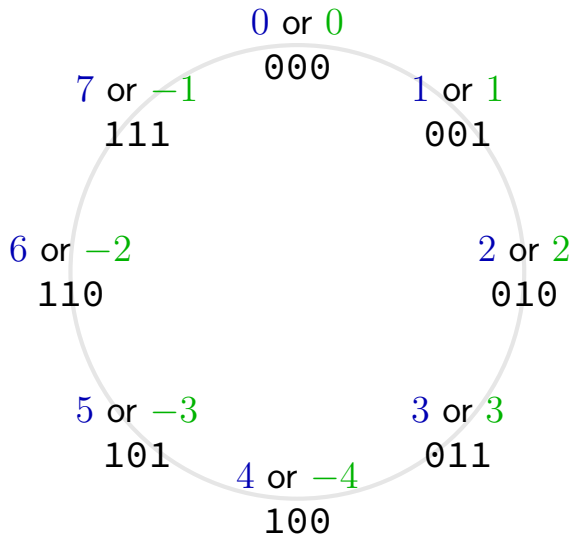
2's complement (5 bit)

$$\begin{array}{rccccccccc} +10 = & \boxed{0} & 1 & 0 & 1 & 0 & & & \\ & 0 \cdot (-2^4) & + & 1 \cdot 2^3 & + & 0 \cdot 2^2 & + & 1 \cdot 2^1 & + & 0 \cdot 2^0 \\ & 0 & + & 2^3 & + & 0 & + & 2^1 & + & 0 = 10 \end{array}$$

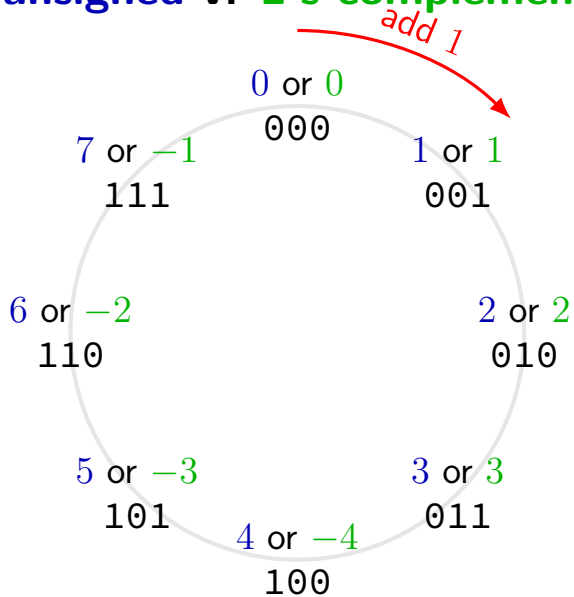
$$\begin{array}{rccccccccc} -10 = & 1 & 0 & 1 & 1 & 0 & & & \\ & 1 \cdot (-2^4) & + & 0 \cdot 2^3 & + & 1 \cdot 2^2 & + & 1 \cdot 2^1 & + & 0 \cdot 2^0 \\ & \boxed{-2^4} & + & 0 & + & 2^2 & + & 2^1 & + & 0 = -10 \end{array}$$

" -2^4 's place"

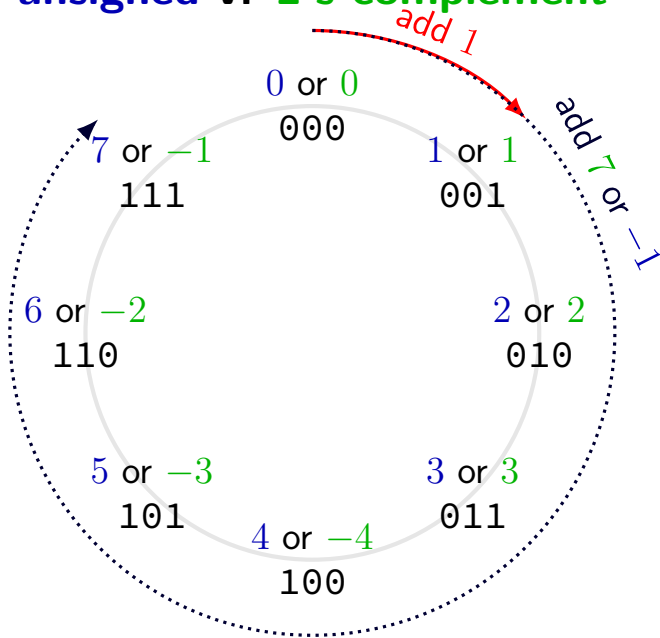
unsigned v. 2's complement



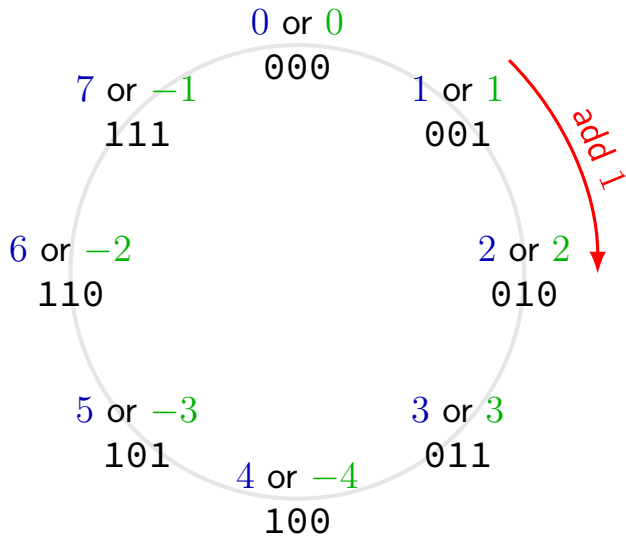
unsigned v. 2's complement



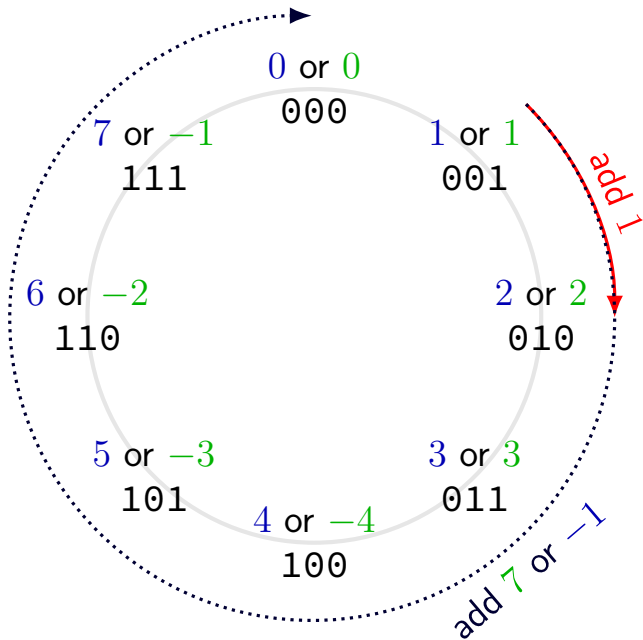
unsigned v. 2's complement



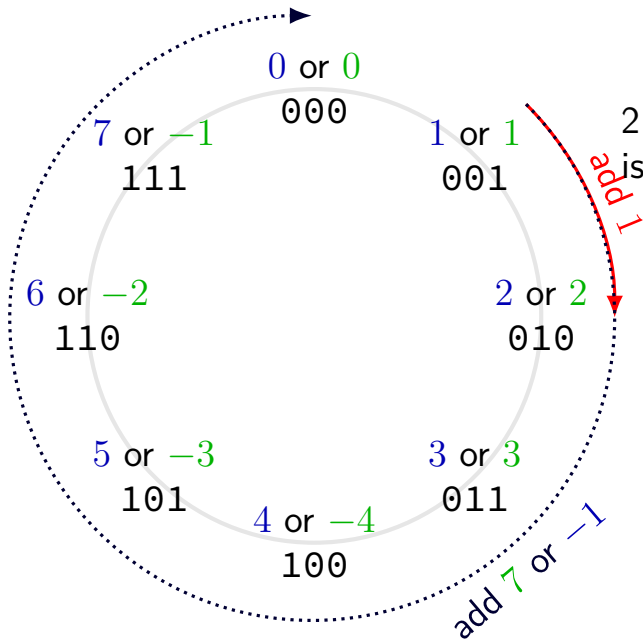
unsigned v. 2's complement



unsigned v. 2's complement



unsigned v. 2's complement



2's complement addition
is same as unsigned addition

other 2's complement arithmetic

subtraction also the same as unsigned

multiplication — repeated addition — mostly the same
(but need some extra precision)

converting to 2's complement (version 1)

take absolute value, convert to bits

if negative, flip all the bits and add one

$$-14 \rightarrow -00001110 \rightarrow 11110001 + 1 \rightarrow 11110010$$

$$-127 \rightarrow -01111111 \rightarrow 10000000 + 1 \rightarrow 10000001$$

$$-128 \rightarrow -10000000 \rightarrow 01111111 + 1 \rightarrow 10000000$$

converting to 2's complement (version 2)

if negative, take absolute value, subtract from 2^n , encode that

$$-14 \rightarrow 2^8 - 14 = 242 \rightarrow 11110010$$

$$-127 \rightarrow 2^8 - 127 = 129 \rightarrow 10000001$$

$$-128 \rightarrow 2^8 - 127 = 129 \rightarrow 10000000$$

sign extension

have 8-bit 2's complement number 1101 0111

what is this as a 16-bit 2's complement number?

sign extension

have 8-bit 2's complement number 1101 0111

what is this as a 16-bit 2's complement number?

general rule: extend by copying the *sign bit*:

1101 0111 → 1111 1111 1101 0111

0010 1111 → 0000 0000 0010 1111

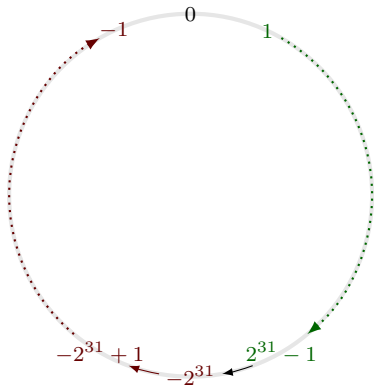
“sign extension”

two's complement summary

$$\begin{array}{ccccccc} & -2^{31} & +2^{30} & +2^{29} & & +2^2 & +2^1 & +2^0 \\ -1 = & 1 & 1 & 1 & \dots & 1 & 1 & 1 \end{array}$$

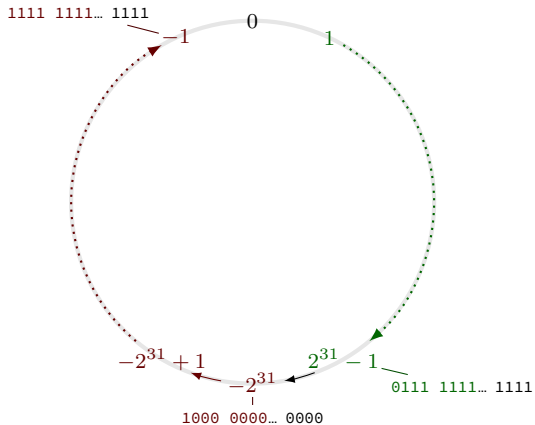
two's complement summary

$$-1 = \begin{matrix} & -2^{31} & +2^{30} & +2^{29} & & +2^2 & +2^1 & +2^0 \\ 1 & 1 & 1 & \dots & 1 & 1 & 1 \end{matrix}$$



two's complement summary

$$-1 = \begin{matrix} & -2^{31} & +2^{30} & +2^{29} & & +2^2 & +2^1 & +2^0 \\ 1 & 1 & 1 & \dots & 1 & 1 & 1 \end{matrix}$$



integer overflow

“wrap around”

8-bit signed: $127 + 1 \rightarrow -128$

8-bit unsigned: $255 + 1 \rightarrow 0$

16-bit signed: $32\,767 + 1 \rightarrow -32\,768$

16-bit unsigned: $65\,536 + 1 \rightarrow 0$

32-bit signed: around 2 billion

64-bit signed: around 9×10^{18}

...

on integer overflow in C++ (1)

```
unsigned int x; // lab machines: 32-bit unsigned  
x = 4294967295; // (2 to the 32) minus 1  
x += 10;  
cout << x << endl; // OUTPUT: 9
```

on integer overflow in C++ (1)

```
int x; // lab machines: 32-bit signed  
x = 2147483647; // maximum integer  
x += 10; // UNDEFINED!  
cout << x << endl; // EXPECT big negative number,  
                   // but not guaranteed
```

in practice: usually get wraparound behavior...

but compiler is not required to do this for signed numbers
and takes advantage of this to optimize, sometimes

some real numbers

1

3

$-\frac{100}{7}$

π

0.1

$\sqrt{2}$

...

want to represent these: accurately? compactly? efficiently?

fixed point

$$\frac{1}{3} = 0.101010101 \dots_{\text{TWO}}$$

$$\approx +0000.1010_{\text{TWO}} \text{--- represent as } 00000 \ 1010$$

$$\frac{100}{7} = 1110.001001001 \dots_{\text{TWO}}$$

$$\approx -1110.0010_{\text{TWO}} \text{--- represent as } 01110 \ 0010$$

fixed point

$$\frac{1}{3} = 0.101010101 \dots_{\text{TWO}}$$

$$\approx +0000.1010_{\text{TWO}} \text{ — represent as } 00000 \ 1010$$

$$\frac{100}{7} = 1110.001001001 \dots_{\text{TWO}}$$

$$\approx -1110.0010_{\text{TWO}} \text{ — represent as } 01110 \ 0010$$

$x \approx y/2^K$ — represent with fixed-sized signed integer y

this case: $y/2^4$ and y is 9 bits.

why fixed-point?

$x \approx y/2^K$ (y fixed-sized signed integer)

math similar to integer math:

- addition/subtraction — same

- multiplication — same except divide by 2^K

- division — same except multiply by 2^K

easy to understand what values are represented well

why not fixed-point?

pretty small range of numbers for space used

hard to choose a 2^K that works for lots of applications

recall (?): scientific notation

$$+\frac{1}{3} = +0.33333333 \dots$$

$$\approx +3.33 \cdot 10^{-1}$$

$$-\frac{100}{7} = -14.285714 \dots$$

$$\approx -1.42 \cdot 10^{+1}$$

recall (?): scientific notation

$$+\frac{1}{3} = +0.33333333 \dots$$

$$\approx +3.33 \cdot 10^{-1}$$

$$-\frac{100}{7} = -14.285714 \dots$$

$$\approx -1.42 \cdot 10^{+1}$$

\pm mantissa \cdot base^{exponent}

recall (?): scientific notation

$$+\frac{1}{3} = +0.33333333 \dots$$

$$\approx +3.33 \cdot 10^{-1}$$

$$-\frac{100}{7} = -14.285714 \dots$$

$$\approx -1.42 \cdot 10^{+1}$$

\pm mantissa \cdot base^{exponent}

recall (?): scientific notation

$$+\frac{1}{3} = +0.33333333 \dots$$

$$\approx +\textcolor{red}{3.33} \cdot 10^{-1}$$

$$-\frac{100}{7} = -14.285714 \dots$$

$$\approx -\textcolor{red}{1.42} \cdot 10^{+1}$$

$\pm \textcolor{red}{\text{mantissa}} \cdot \text{base}^{\text{exponent}}$

recall (?): scientific notation

$$+\frac{1}{3} = +0.33333333 \dots$$

$$\approx +3.33 \cdot 10^{-1}$$

$$-\frac{100}{7} = -14.285714 \dots$$

$$\approx -1.42 \cdot 10^{+1}$$

\pm mantissa \cdot base^{exponent}

recall (?): scientific notation

$$+\frac{1}{3} = +0.33333333 \dots$$

$$\approx +3.33 \cdot 10^{-1}$$

$$-\frac{100}{7} = -14.285714 \dots$$

$$\approx -1.42 \cdot 10^{+1}$$

\pm mantissa \cdot base^{exponent}

base-2 scientific notation

$$\frac{1}{3} = 0.101010101 \dots_{\text{TWO}}$$

$$\approx 0.1010101010_{\text{TWO}} = +1.0101010101_{\text{TWO}} \cdot 2^{-1}$$

$$-\frac{125}{4} = -111111.01 \dots_{\text{TWO}}$$

$$= -1.1111101_{\text{TWO}} \cdot 2^2$$

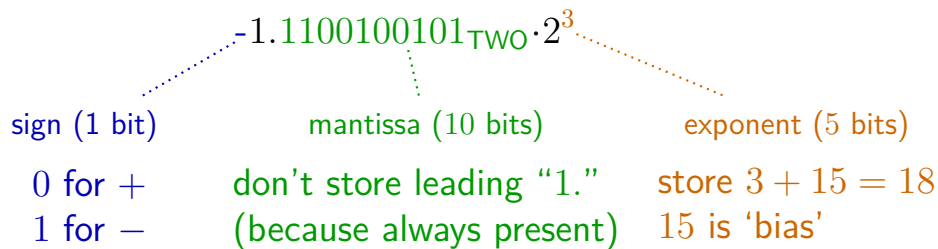
$$-\frac{100}{7} = -1110.01001001 \dots_{\text{TWO}}$$

$$\approx -1110.010010_{\text{TWO}} = -1.1100100101_{\text{TWO}} \cdot 2^3$$

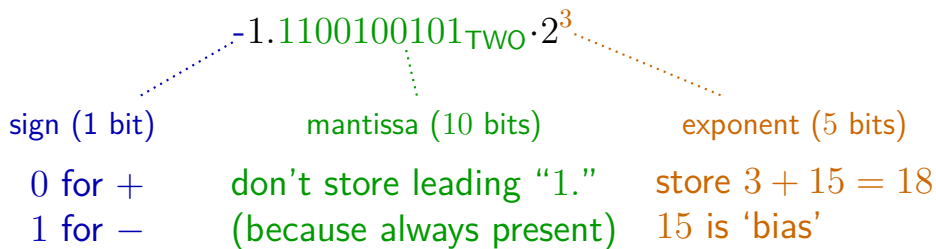
IEEE half-precision floating point

$$-1.1100100101_{\text{TWO}} \cdot 2^3$$

IEEE half-precision floating point

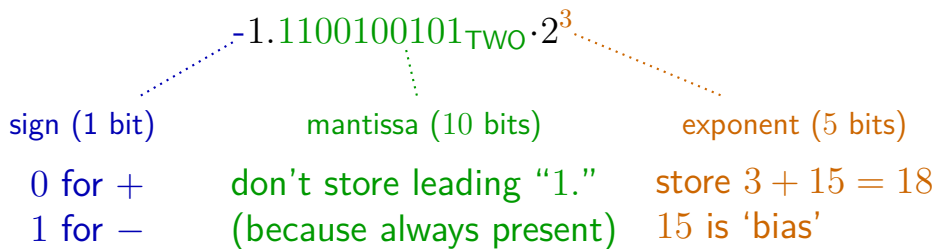


IEEE half-precision floating point



1 10010 1100100101

IEEE half-precision floating point



1 10010 1100100101

on typical little endian system:

byte 0: 00100101

byte 1: 11001011

IEEE half precision float

1 sign bit (1 for negative)

5 exponent bits

bias of 15 — if bits as unsigned are e , exponent is $E = e - 15$

10 mantissa bits

leading “1.” not stored

$$\text{value} = (1 - 2 \cdot \text{sign}) \cdot (1.\text{mantissa}_{\text{TWO}}) \cdot 2^{\text{exponent}-15}$$

approximation

example: represented $\frac{100}{7} \approx 14.285$ as $\frac{1829}{128} \approx 14.289$

too large by $\frac{3}{896}$

10 bits mantissa + implicit “1” — about $\log_{10}(2^{11}) \approx 3.3$ decimal digits

other IEEE precisions

	half	single	double	quad
C++*/Java type	—	float	double	—
sign bits	1	1	1	1
exponent bits	5	8	11	15
exponent bias	15 ($2^5 - 1$)	127 ($2^7 - 1$)	1023 ($2^{10} - 1$)	16383 ($2^{14} - 1$)
mantissa bits	10	23	52	112
total bits	16	32	64	128

(* = typical C++ type; might vary in some implementations)

float example: manually (1)

$$25.25 = \frac{101}{4} = \frac{101}{2^2}$$

largest power of two < 25.25 ? $16 = 2^4$

$$\begin{aligned}\frac{101}{4} \cdot \frac{2^4}{2^4} &= \frac{101 \cdot 2^4}{2^6} \\ &= \frac{101}{2^6} \times 2^4 \\ &= \frac{1100101_{\text{TWO}}}{2^6} \times 2^4 \\ &= 1.000101_{\text{TWO}} \times 2^4\end{aligned}$$

float example: manually (2)

$$25.25 = \frac{101}{4} = 11001.01_{\text{TWO}} = \\ +1.1001\ 0100\ 0000\ 0000\ 0000\ 0000_{\text{TWO}} \cdot 2^4$$

float example: manually (2)

$$25.25 = \frac{101}{4} = 11001.01_{\text{TWO}} =$$

$$+1.1001\ 0100\ 0000\ 0000\ 0000\ 0000_{\text{TWO}} \cdot 2^4$$

sign (1 bit)

0 for +

mantissa (23 bits)

(leading "1." not stored)

exponent (5 bits)

store "4 + 127 =

1000 0011_{TWO}"

127 is bias for float

float example: manually (2)

$$25.25 = \frac{101}{4} = 11001.01_{\text{TWO}} =$$

$$+1.1001\ 0100\ 0000\ 0000\ 0000\ 0000_{\text{TWO}} \cdot 2^4$$

sign (1 bit)

0 for +

mantissa (23 bits)

(leading "1." not stored)

exponent (5 bits)

store "4 + 127 =
1000 0011_{TWO}"
127 is bias for float

0 1000 0011 1001 0100 0000 0000 0000 0000

diversion: 25.25 to binary

$$\begin{aligned} 25.25 &= \frac{101}{4} \\ &= \frac{1100101_{\text{TWO}}}{2^2} \\ &= 11001.01_{\text{TWO}} \end{aligned}$$

diversion: 25.25 to binary

$$\begin{aligned} 25.25 &= 2^4 + 2^3 + (9.25 - 2^3) = 2^4 + 2^3 + 1.25 \\ &\quad (1.25 < 2^2) \\ &\quad (1.25 < 2^1) \\ &= 2^4 + 2^3 + (1.25 - 2^0) = 2^4 + 2^3 + 2^0 + 0.25 \\ &\quad (0.25 < 2^{-1}) \\ &= 2^4 + 2^3 + 2^0 + 2^{-2} + (0.25 - 2^{-2}) = 2^4 + 2^3 + 2^0 + 2^{-2} \end{aligned}$$

float example: from C++

```
#include <iostream>
using std::cout; using std::hex; using std::endl;
// union: all elements use the *same memory*
union floatOrInt {
    float f;
    unsigned int u;
};
int main() {
    union floatOrInt x;
    x.f = 25.25;
    cout << hex << x.u << endl;
    // OUTPUT: 41ca0000
}
```

4	1	c	a	0	0	0	0
0100	0001	1100	1010	0000	0000	0000	0000

float example 2: manually

$$\begin{aligned} 0.1_{\text{TEN}} &= \frac{1}{16} + 0.0375 = \frac{1}{16} + \frac{1}{32} + 0.00625 = \\ &\dots = 0.00011001100110011\dots_{\text{TWO}} \approx \\ &+ 1.1001\ 1001\ 1001\ 1001\ 1001\ 101_{\text{TWO}} \cdot 2^{-4} \end{aligned}$$

float example 2: manually

$$0.1_{\text{TEN}} = \frac{1}{16} + 0.0375 = \frac{1}{16} + \frac{1}{32} + 0.00625 = \dots = 0.00011001100110011\dots_{\text{TWO}} \approx$$

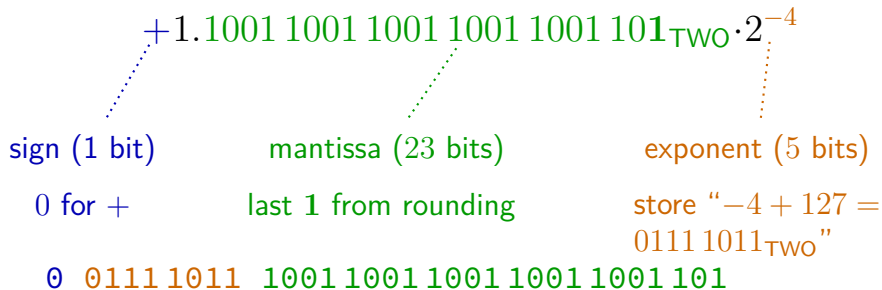
$+1.1001\ 1001\ 1001\ 1001\ 1001\ 101_{\text{TWO}} \cdot 2^{-4}$

sign (1 bit) mantissa (23 bits) exponent (5 bits)

0 for + last 1 from rounding store “ $-4 + 127 = 0111\ 1011_{\text{TWO}}$ ”

float example 2: manually

$$0.1_{\text{TEN}} = \frac{1}{16} + 0.0375 = \frac{1}{16} + \frac{1}{32} + 0.00625 = \dots = 0.00011001100110011\dots_{\text{TWO}} \approx$$



float example 2: manually

$$0.1_{\text{TEN}} = \frac{1}{16} + 0.0375 = \frac{1}{16} + \frac{1}{32} + 0.00625 = \dots = 0.00011001100110011\dots_{\text{TWO}} \approx$$

$+1.1001\ 1001\ 1001\ 1001\ 1001\ 101_{\text{TWO}} \cdot 2^{-4}$

sign (1 bit) mantissa (23 bits) exponent (5 bits)

0 for + last 1 from rounding store “ $-4 + 127 = 0111\ 1011_{\text{TWO}}$ ”

0 0111 1011 1001 1001 1001 1001 1001 101

closest float to 0.1 between 0.1 and 0.1000001

float example 2: inaccurate (1)

```
#include <iostream>
using std::cout; using std::endl;

int main(void) {
    int count;
    float base = 0.1f;
    for (count = 0; base * count < 100000000; ++count) {}
    cout << count << endl;
    // OUTPUT: 99999996
    return 0;
}
```

float example 2: inaccurate (2)

```
#include <iostream>
using std::cout; using std::endl;

int main(void) {
    int count = 0;
    for (float f = 0; f < 2000.0; f += 0.1) {
        ++count;
    }
    cout << count << endl;
    // OUTPUT: 20004
    return 0;
}
```

float example 2: inaccurate (3)

```
#include <iostream>
using std::cout; using std::endl;
int main(void) {
    cout.precision(30);
    for (float f = 0; f < 2000.0; f += 0.1) {
        cout << f << endl;
    }
    return 0;
}
```

0

0.100000001490116119384765625

0.20000000298023223876953125

...

2.2000000476837158203125

2.2999999523162841796875

...

floating point is not uniform

in half-precision, next number after:

$$1 = 1.000\,000\,000\,0_{\text{TWO}} \cdot 2^0 \text{ is } 1.000\,000\,000\,1_{\text{TWO}} \cdot 2^0 \approx 1.0010_{\text{TEN}} \\ \sim +.001$$

$$100 = 1.100\,100\,000\,0_{\text{TWO}} \cdot 2^6 \text{ is } 1.100\,100\,000\,1_{\text{TWO}} \cdot 2^6 \approx 100.06_{\text{TEN}} \\ \sim +.06$$

possible numbers are **unevenly spaced**

same as with 'normal' scientific notation:

$$1 = 1.00 \cdot 10^0 \rightarrow 1.01 \cdot 10^0 = 1.01 \text{ versus } 1.00 \cdot 10^2 \rightarrow 1.01 \cdot 10^2 = 101$$

don't compare with `==`/`!=`

```
double x = 0.3;
double y = 0.1;
double y3 = y * 3;
if (x != y3) {
    cout << "not_equal" << endl;
}
cout.setprecision(30);
cout << x << endl;
cout << y3 << endl;
```

not equal

0.2999999999999999988897769753748

0.30000000000000000044408920985006

on comparing floats

```
#include <cmath>
using std::fabs;
// or #include <math.h> and use fabs
// without a using statement
...
// chose based on expected accuracy
const float EPSILON = 1e-6;
float x, y;
...
if (fabs(x - y) < EPSILON) {
    ...
}
```

floating point accuracy

`float` — about 7 decimal places

`double` — about 15 decimal places

rounding errors (1)

$$2^{100} + 1$$

$2^{100} + 1$ cannot be represented exactly
would need 99 mantissa bits
rounds to 2^{100}

(but 2^{100} and 1 can)

rounding errors (2)

$$\begin{aligned} & (2^{100} + 1) - 2^{100} \\ & \quad 2^{100} - 2^{100} \\ & \quad \quad 0 \end{aligned}$$

$$\begin{aligned} & (2^{100} - 2^{100}) + 1 \\ & \quad 0 + 1 \\ & \quad \quad 1 \end{aligned}$$

dealing with rounding errors

avoid: adding and subtracting values of very different magnitudes

- tend to have big errors

- tend to have errors in one direction (compound over a calculation)

...by reordering and rearranging calculations

the problem of 0

0 is a very important number

can't be represented with implicit "1."

solution: special cases

IEEE float special cases

exponent bits	mantissa bits	meaning
00000000	000...000	± 0
00000000	non-zero	<i>denormal</i> number
11111111	000...000	$\pm \infty$
11111111	non-zero	not a number (NaN)

$(+1/1000000000) \div \text{huge positive number} = +0$

$(-1/1000000000) \div \text{huge positive number} = -0$

$(+1000000000) \times \text{huge positive number} = +\infty$

$(-1000000000) \times \text{huge positive number} = -\infty$

$1 \div 0 = +\infty$

$0 \div 0 = \text{NaN}$

$\sqrt{-1} = \text{NaN}$

float min magnitude value

exponent of 0000 0001 (not 0 since that's special)

mantissa of 000...000

$$1.000000 \dots_{\text{TWO}} \cdot 2^{1-\text{bias}} = 2^{-126}$$

float max magnitude value

exponent of 1111 1110 (not all 1s since that's special)

mantissa of 111...111

$$1.111111 \dots 11_{\text{TWO}} \cdot 2^{254 - \text{bias}} = 1.111111 \dots 1_{\text{TWO}} \cdot 2^{127} = 2^{128} - 2^{104}$$

on denormals

denormals — minimum exponent bits, non-zero mantissa

smaller in magnitude than “normal” minimum value

ignore the “implicit 1.” rule

notorious for being superslow on some systems

some CPUs take 100s of times longer to compute on them

we won't ask you about them

decimal floating point

if storing 0.001 exactly is important?

floating point formats base of 10 instead of 2

$$1.000 \times 10^{-3}$$

example: IEEE decimal floating point

32, 64, 128-bit formats

still store exponent+mantissa

no leading “1.” trick (doesn’t work with 10^x)

binary-coded decimal

if integer conversion to/from base-10 is important?

but want to use binary hardware

one option: every 4 bits is a decimal digit

not all possible bit patterns used

e.g. represent 147_{TEN} as 0001 0100 0111

part of family on decimal-in-binary encodings

some more compact than this (e.g. store 2 digits at a time)

backup slides

optimizing with overflow example

```
void foo(int x) {  
    while (--x < 0) {  
        bar();  
    }  
}
```

in latest version of clang++ or g++

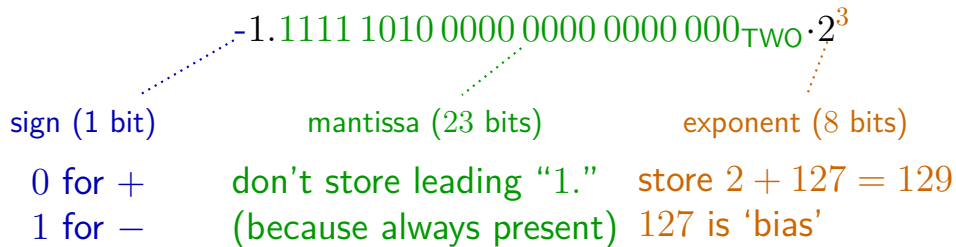
compiles into an infinite loop if x is initially negative

if maximum optimizations are enabled (`-O3` command-line option, not default)

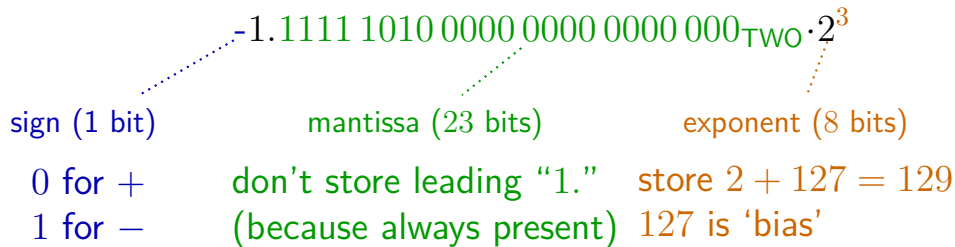
IEEE half-precision floating point

$$-1.1111\ 1010\ 0000\ 0000\ 0000\ 0000_{\text{TWO}} \cdot 2^3$$

IEEE half-precision floating point



IEEE half-precision floating point



1 100 0000 1 111 1101 0000 0000 0000 0000

IEEE single precision float

1 sign bit (1 for negative)

10 exponent bits

bias of 127 — if bits as unsigned are e , exponent is $E = e - 127$

23 mantissa bits

leading “1.” not stored

$$\text{value} = (1 - 2 \cdot \text{sign}) \cdot (1.\text{mantissa}_{\text{TWO}}) \cdot 2^{\text{exponent}-127}$$