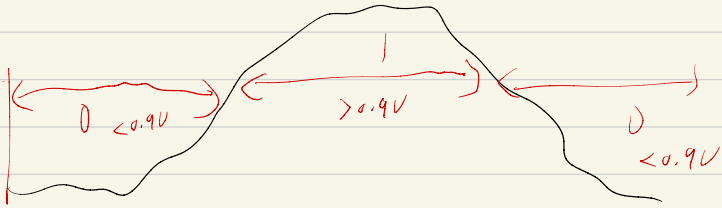
Everything is bits

Each bit is 0 or 1.

By Encoding / interpreting sets of bits in various ways,
computer determine what to do (instructions)
... and represent and manipulate numbers, sets, strings, etc...

Why bits? Electronic Implementation.
1. Easy to store with bistate elements
2. Reliably transmitted on noisy and inaccurate wires



For example can count in binary:

Base 2 Number Representation:
1. Represent 15213_{10} as 11101101110101_2
2. Represent 120_{10} as $1.0011100100111001\,[0011]\ldots_2$
3. Represent 1.5213×10^4 as $1.1011011011101_2 \times 2^{13}$

Byte = 8 bits.

Hex	Decimal	Binary
16	10	2

adders. 64位 8 bits
32位 4 bits

And Or
Not Xor (Exclusive-or)

Example: Representing & Manipulating Sets.

1. Representation:
   (i) Width w bit vector represents subsets of $\{0, \dots, w-1\}$
   $a_j = 1$ if $j \in A$
   01101001 $\{0,3,5,6\}$
   01010101 $\{0,2,4,6\}$

Operations:

&	Intersection	01000001	$\{0,6\}$
|	Union.	01111101	$\{0,2,3,4,5,6\}$
^	(symmetric difference)	00111100	$\{2,3,4,5\}$
~	complement	10101010	$\{1,3,5,7\}$

Bit-Level Operations in C.

Operations:

& | ^ ~ Available in C:

1. Apply to any "integral" data type.
   long, int, short, char, unsigned.

2. View arguments as bit vectors.

3. Arguments applied bit-wise.

Examples (char data type).

$\sim 0x41 \rightarrow 0xBE$

$\sim 01000001_2 \rightarrow 10111110_2$

Contrast: Logic Operations in C

1. Contrast to Logical Operators.

&& || !

View 0 as "false"

Anything nonzero as "true"

always return 0 or 1

Early termination.

!0x41 → 0x00

!0x00 → 0x01

!!0x41 → 0x01

0x69 && 0x55 → 0x01

p && *p. (avoid null pointer access)

(shortcut logistics)

Shift Operations:

Left shift: x<<y:
Shift bit-vector x left y positions.
— Throw away extra bits on left.
Fill with 0's on right.

Right Shift: x>>y:
Shift bit-vector x right y positions.
Throw away extra bits on right.

Logical Shift:
Fill with 0's on left.

Arithmetic Shift:
Replicate most significant bit on left.

Undefined behavior:
shift amount < 0 or ≥ word size

e.g. 1.

Argument x	01100010
<<3	00010000
Log. >>2	00011000
Arith. >>2	00011000

e.g. 2.

Argument x	10100010
<<3	00010000
Log. >>2	00101000
Arith. >>2	11101000

Encoding Integers:

Unsigned:

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

(w bits)

Two's complement:

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

(w bits)

sign bit

short int x = 15213

short int y = -15213.

C short 2 bytes long.

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

Sign Bit:

For 2's complement, most significant bit indicates sign:

0 for nonnegative

1 for negative.

Numeric Ranges.

Unsigned Values:

$U_{min} = 0$

000…0

$U_{max} = 2^w - 1$

111…1

Two's complement Values.

$T_{Min} = -2^{w-1}$

100…0

$T_{Max} = 2^{w-1} - 1$

011…1

Other Values

Minus 1

111…1

All bits are 1 $\iff$ value is minus one.

$|T_{Min}| = T_{Max} + 1$

- Asymmetric range

$U_{Max} = 2 * T_{Max} + 1.$

C programming

#include <limits.h>

Declares constants, e.g.

ULONG_MAX

LONG_MAX

LONG_MIN.

value platform specific

Unsigned & Signed Numeric Values

1. Equivalence
   - same encodings for nonnegative values

2. Uniqueness
   (1) Every bit pattern represents integer value
   (2) Each representable integer has unique bit encoding.

3. Can invert Mappings.

$$U2B(x) = B2U^{-1}(x)$$

Bit pattern for unsigned integer.

$$T2B(x) = B2T^{-1}(x)$$

Bit pattern for two's complement numbers.

Mapping between unsigned and two's complement numbers

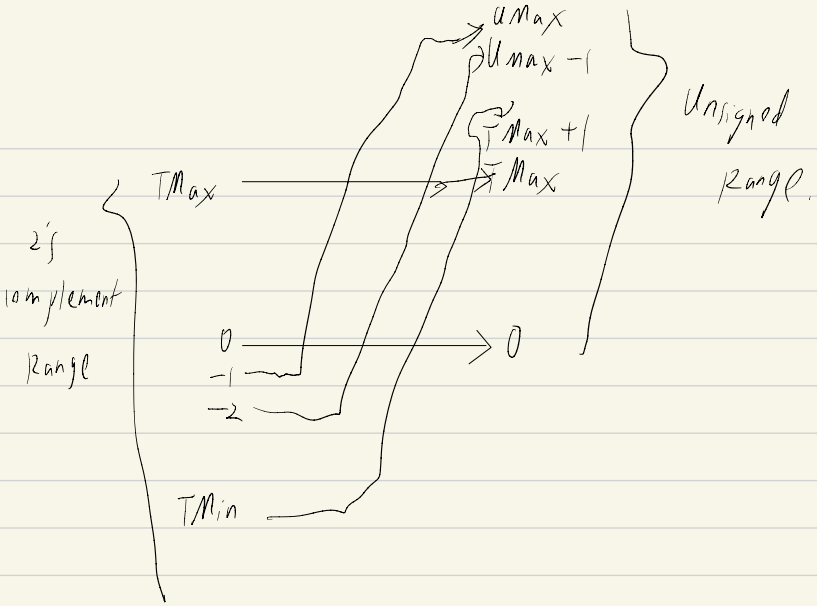
Keep bit representations and reinterpret.

Conversion Visualized.

2's comp. → Unsigned

Ordering Inversion

Negative → Big positive.


UMax
UMax -1
TMax +1
TMax
TMax
2's
complement
Range
0
-1
-2
0
TMin
Unsigned
Range

Signed vs. Unsigned in C.

⊖ Constants.

(i) By default are considered to be signed integers. Unsigned if have "U" as suffix

0U, 4294967259U

(ii) Casting

Explicit casting between signed & unsigned same as U2T and T2U.

```
int      tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

Implicit casting also occurs via assignments and procedure calls.

```
tx = ux;
uy = ty;
```

(python Java ☺)

BUT.. (C few language ← unsigned is a explicit data type)

casting surprises

Expression Evaluations

If there is a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned.

Including comparison operations <, >, ==, <=, >=.

Examples for W = 32: TMIN = -2147483648 ☆

TMAX = 2147483647 ☆

e.g. (-1 > 0U)

2147483647 > (int) 2147483648U

Summary

Casting signed ⟷ unsigned: Basic Rules

Bit pattern is maintained

But reinterpreted.

Can have unexpected effects: adding or subtracting 2^w.

Expression containing signed and unsigned int.

int is cast to unsigned.

e.g.1. sizeof(...) return an unsigned type value.

Casting Surprises:

Expression Evaluation:

If there is a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned.

Including comparison operation <, >, ==, <=, >=

Example for W=32: TMIN = -2147483648

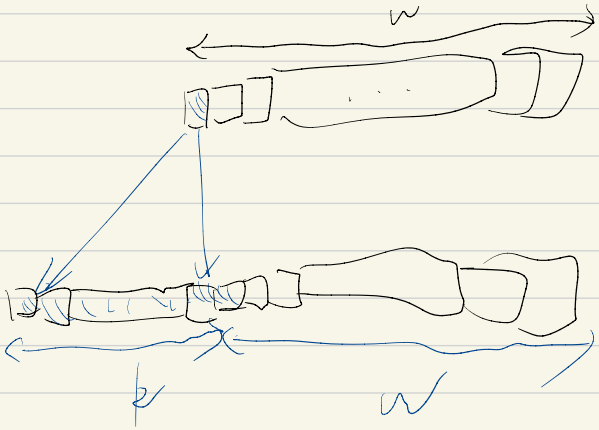
TMAX = 2147483647

Sign Extension:

1. Task:
   (i) Given w-bit signed integer X.
   (ii) Convert it to (w+k)bit integer with same value.

2. Rule.
   (i) Make k copies of sign bit.
   (ii) $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

($x_{w-1}, \ldots, x_{w-1}$: k copies of MSB)



Summary:

Expanding, Truncating: Basic Rules

Expanding (e.g. short int to int):
Unsigned: zeros added
signed: sign extension
Both yield expected result.

Truncating (e.g. unsigned to unsigned short):
unsigned/signed: bits are truncated.
Result reinterpreted.
Unsigned: mod operation.
Signed: similar to mod
For small numbers yields expected behavior.

(signed truncating:)
1101 -5
↓
1011 -5

10011 -30
↓
0011 +3