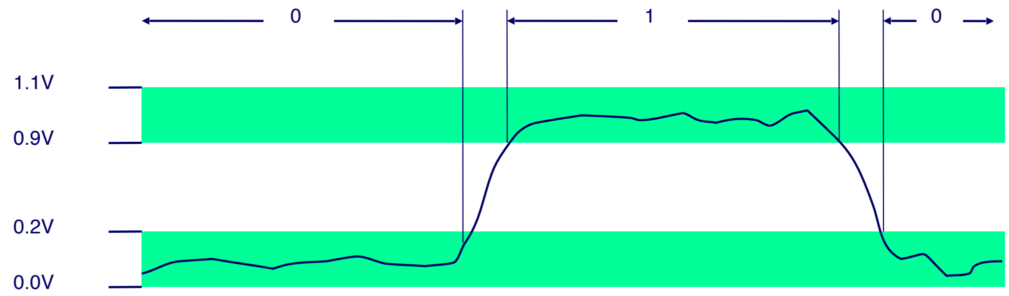
Data representation and C types

CS210 - Fall 2023

Vasiliki Kalavri
vkalavri@bu.edu

Everything is bits

- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Bit interpretation

2+4

5

6 5 1+4

Hex

Dec

01100101

→ a number (binary)

→ a character, 'e'

→ an address

→ an instruction

65

A bit pattern can represent multiple objects depending on the interpretation we assign to it

That's fundamentally how computer systems work!

- we assign meaning to bit patterns to encode elements of a finite set

Integer representations: unsigned

How can we use bits to encode non-negative integers? no negative, $0 \leq$

Vector $\vec{x}$ of length $\underline{w}$ bits

$\vec{x} = [b_{w-1}, b_{w-2}, \ldots, b_1, b_0]$, $x_i = \{0,1\}$

most significant — least significant

$$B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Binary to unsigned length w

$\boxed{w=4}$

$$\overset{3\,2\,1\,0}{0101} = \underline{0} \cdot \overset{2^3}{8} + \underline{1} \cdot \overset{2^2}{4} + \underline{0} \cdot \overset{2^1}{2} + \underline{1} \cdot \overset{2^0}{1}$$

Range of unsigned representation

How many values can we represent with **w** bits? $2^w - 1$

16 values

$$\underline{U_{min_w}}$$

$$\underbrace{0000 \ldots 0}_{w} = 0$$

$$\underline{U_{max_w}}$$

$$\underbrace{1111 \ldots 1}_{w} = 2^w - 1$$

eg $\boxed{w = 4}$ $\quad U_{max_4} = 15$ ie. 1111

Integer representations: signed

How can we represent both negative and non-negative integers?

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

With $w=4$ bits, we can represent 16 (2^4) different patterns.

* We will use the most significant bit (MSB) to represent the sign.

-1×2^{MSB}

2’s complement

$-2^3 \quad 2^2 \quad 2^1 \quad 2^0$

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

T_{Max} → 7

$-2^3 \quad 2^2 \quad 2^1 \quad 2^0$

1×-8

-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

T_{Min} = -8

We assign a <u>negative</u> weight to the MSB.

2's complement definition

$$B2T_w(\vec{x}) = \underbrace{-x_{w-1} \cdot 2^{w-1}}_{\text{negative weight}} + \underbrace{\sum_{i=0}^{w-2} x_i \cdot 2^i}_{\text{positive sum}}$$

Range

$Tmin_w = -2^{w-1}$ (positive sum is 0), ie 100....0

$Tmax_w = 2^{w-1} - 1$ (negative weight is 0), ie 011....1

2's complement example

```
short int x =  15213;
short int y = -15213;
```

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Not just flip

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

* To compute negation

1) Flip all bits
2) Add 1

eg. 7: 0111

~7: 1000

(~7+1): 1001 → B2T(1001) = -7

Important numbers

unsigned

bits

	8 2bytes	16	32	64
UMax_w	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
Tmin_w	10000000 0x80 (-2^8]	0x8000	0x80000000	0x8000000000000000
	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808
TMax_w	01111111 0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0 just 0	0	0x00	0x0000	0x0000000000000000

4bytes 8bytes 16bytes

1 2 3 4 5 6 7
2x 4 8 16 32 64 128

Summary of integer representations

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

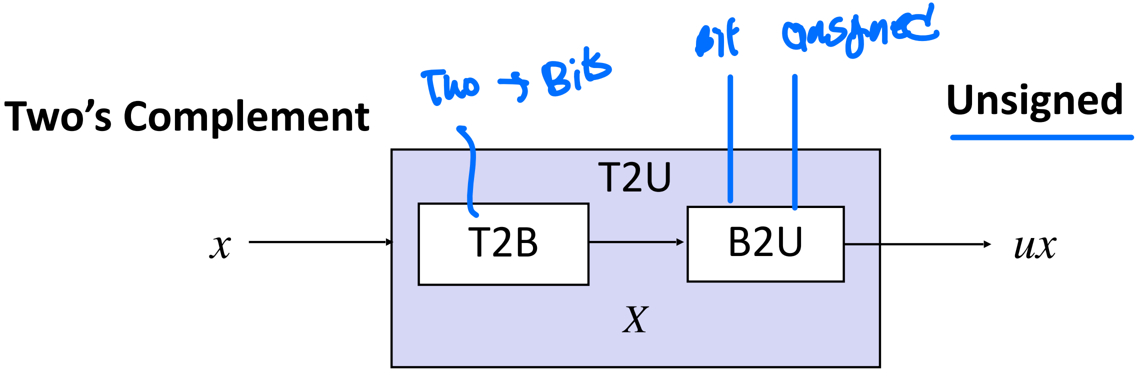
1) Equivalence

2) Uniqueness.

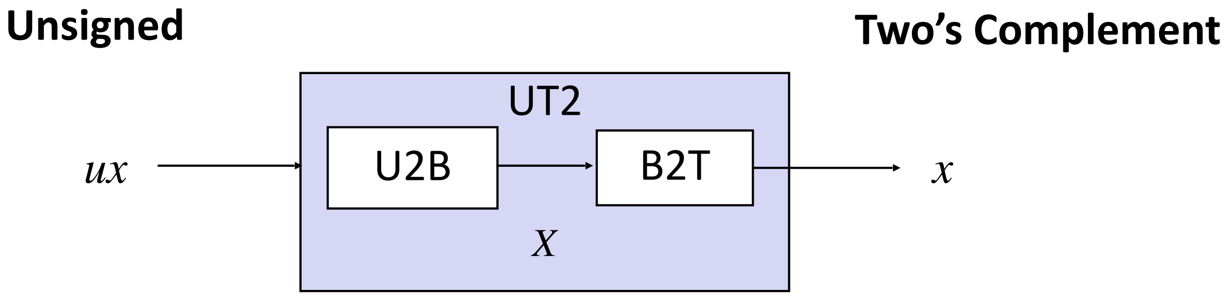
- Equivalence
  - Same encodings for nonnegative values → 똑같음
- Uniqueness
  - Every bit pattern represents a unique integer value
  - Each representable integer has a unique bit encoding

mapping

Mapping between signed and unsigned



Maintain Same Bit Pattern



Maintain Same Bit Pattern

Mapping Signed ↔ Unsigned

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

$\xrightarrow{\text{T2U}}$

$\xleftarrow{\text{U2T}}$

Mapping Signed ↔ Unsigned

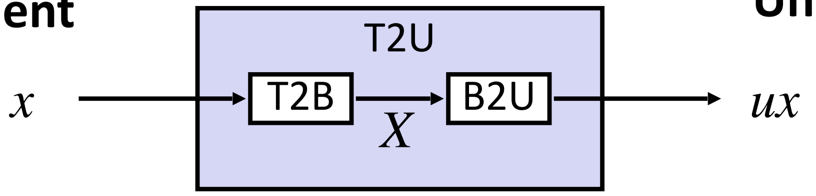
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

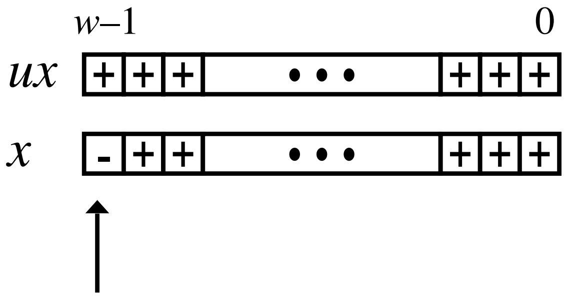
+/- 16

Relation between Signed & Unsigned

Two's Complement **Unsigned**



Maintain Same Bit Pattern



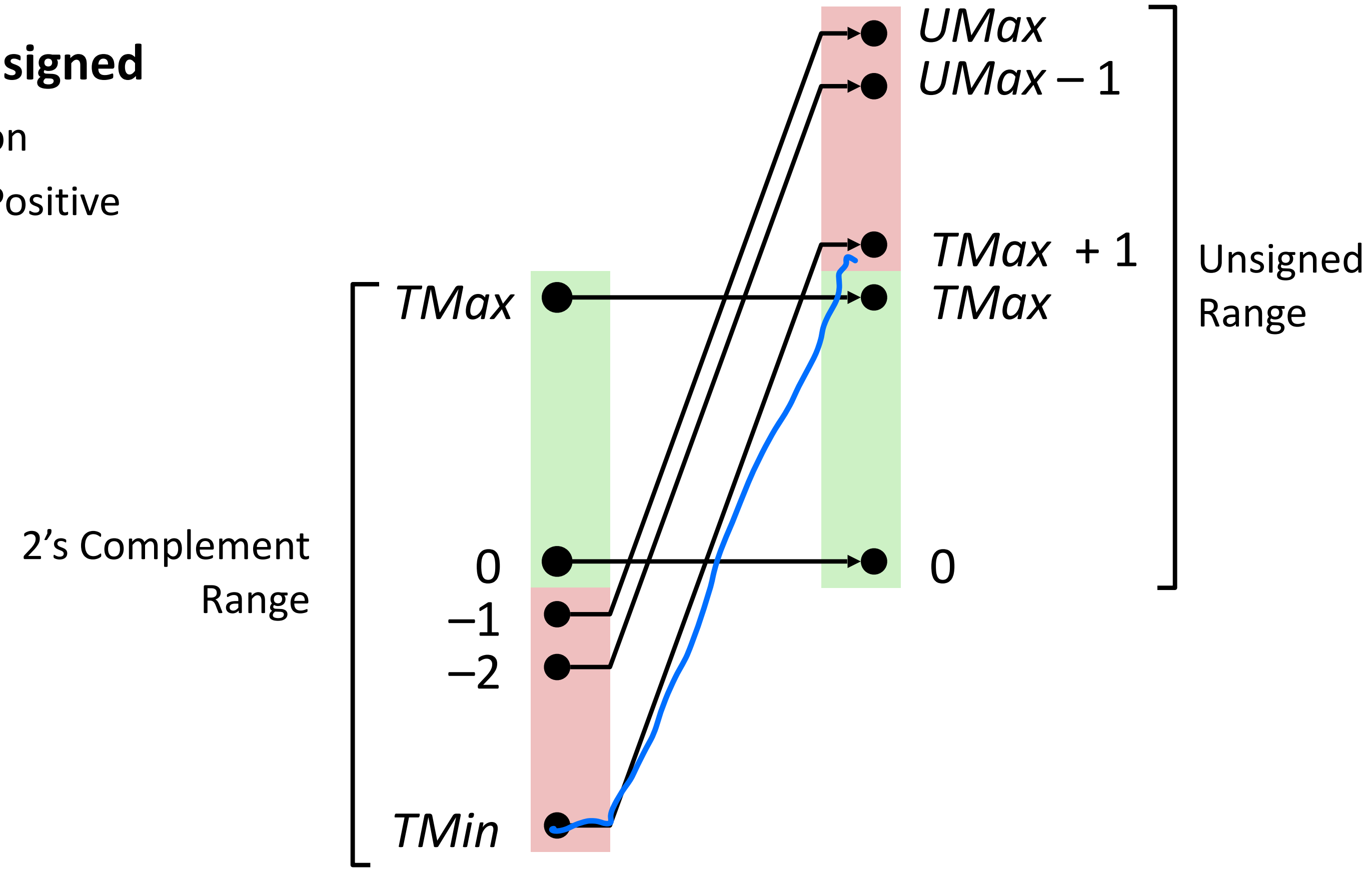
Large negative weight

becomes

Large positive weight

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



Types

Basic C Types

4) float
int
char
Boolean

- C's ***basic*** (built-in) ***types:*** four
  - Integer types, including long integers, short integers, and unsigned integers
  - Floating types (`float`, `double`, and `long double`) 3.
  - `char` one
  - `_Bool` (C99) one

Integer Types

- C supports two fundamentally different kinds of numeric types: integer types and floating types.
- Values of an ***integer type*** are whole numbers.
- Values of a floating type can have a fractional part as well.
- The integer types, in turn, are divided into two categories: signed and unsigned.

+ 2's complement

Signed and Unsigned Integers

sign

- By default, integer variables are signed in C.
- To tell the compiler that a variable has no sign bit, declare it to be $\boxed{\texttt{unsigned}.}$
- Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications.

C : Basic Types and Sizes

long은 일단 8.

C basic data types — standardizes sizes in bytes

C Declaration	32-bit	64-bit
char	1	1
short int	2	2
int	4	4
8 long int	4	8
long long int	8	8
float	4	4 4
double	8	8

C Declaration	32-bit	64-bit
unsigned char	1	1
unsigned short int	2	2
unsigned int	4	4
unsigned long int	4	8
unsigned long long int	8	8

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix

    `0U, 4294967259U`

- 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;
    ```

Casting Surprises

sign + unsigned → unsigned + unsigned

- Expression Evaluation
  - If there is a mix of unsigned and signed in single expression, ***signed values are implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**

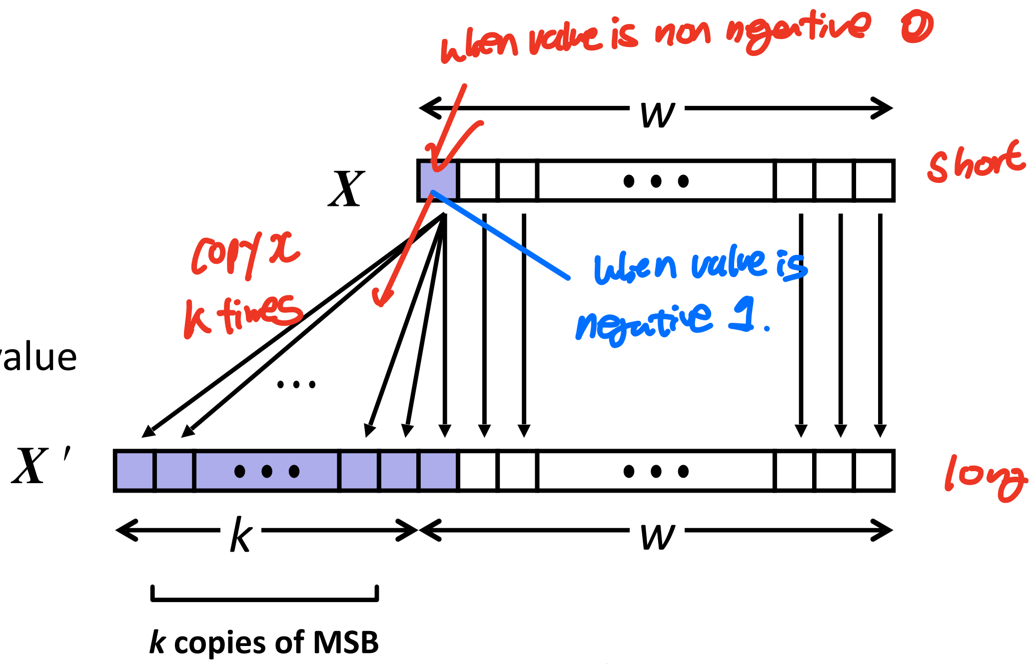
Casting Signed ↔ Unsigned: Basic Rules

- 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`!!

Sign extension

- Task:
  - Given w-bit signed integer x
  - Convert it to $w+k$-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



$w = 4$
$w' = 5$

$-5 = 1011 = -8 + 2 + 1 = -5$

$\downarrow$

$11011 = -16 + 8 + 2 + 1 = -5$

111011 = -32+16+8+2+1 = -5

Sign extension example

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Smaller → larger type Automatically

Truncating numbers

- 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

Floating Types

- C provides three ***floating types,*** corresponding to different floating-point formats:
  - `float` Single-precision floating-point
  - `double` Double-precision floating-point
  - `long double` Extended-precision floating-point

Floating Types

- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.
- The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since that depends on how numbers are stored.
- Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559).

Character Types

- The only remaining basic type is `char`, the character type.
- The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.

Character sets

- A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';   /* lower-case a */
ch = 'A';   /* upper-case A */
ch = '0';   /* zero         */
ch = ' ';   /* space        */
```

- Notice that character constants are enclosed in single quotes, not double quotes.

Operations on characters

- Working with characters in C is simple, because of one fact: *C treats characters as small integers.* (as one byte int)

- In ASCII, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127.

- The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32.

- Character constants actually have `int` type rather than `char` type.

Operations on characters

- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';          /* i is now 97    */
ch = 65;          /* ch is now 'A' */
ch = ch + 1;      /* ch is now 'B' */
ch++;             /* ch is now 'C' */
```

Operations on characters

- Characters can be compared, just as numbers can.
- An `if` statement that converts a lower-case letter to upper case:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Comparisons such as `'a' <= ch` are done using the integer values of the characters involved.
- These values depend on the character set in use, so programs that use `<`, `<=`, `>`, and `>=` to compare characters may not be portable.

Attendance

Implications of data representation

Remember our simple calculator?

```
// adds two integers and returns the result
int add(int i, int j) {
  return i + j;
}
```

Integer Arithmetic

signed

```
int x = foo();
int y = bar();
```

W: 4bits

-8 ~ 7

$2^3 = 8 \times 1$

$2^3 - 1 = 7$

positive positive

- If `x > 0` and `y > 0`, does `x + y > 0` always hold? — overflow

$5 + 6 = 11$ overflow

Think about the extreme cases

-8 7

- Does the expression `x < y` yield the same result as `x - y < 0`? overflow

$x - y = -15$

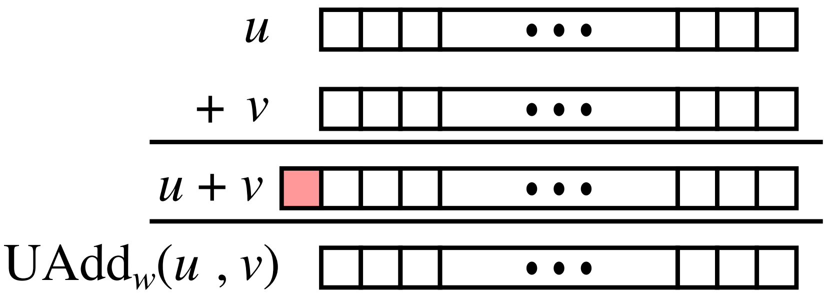
$\underline{x - y < 0}$

Unsigned addition

Operands: w bits

True Sum: w+1 bits incase of overflow

Discard Carry: w bits



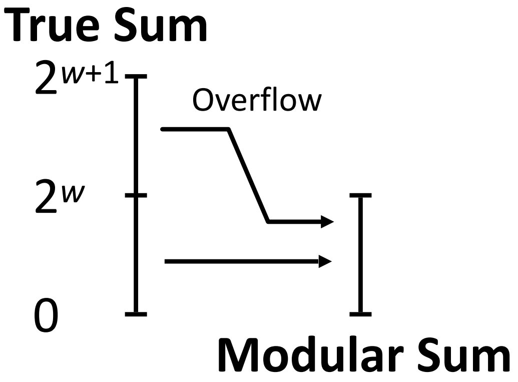
- Standard Addition Function
  - Ignores carry output

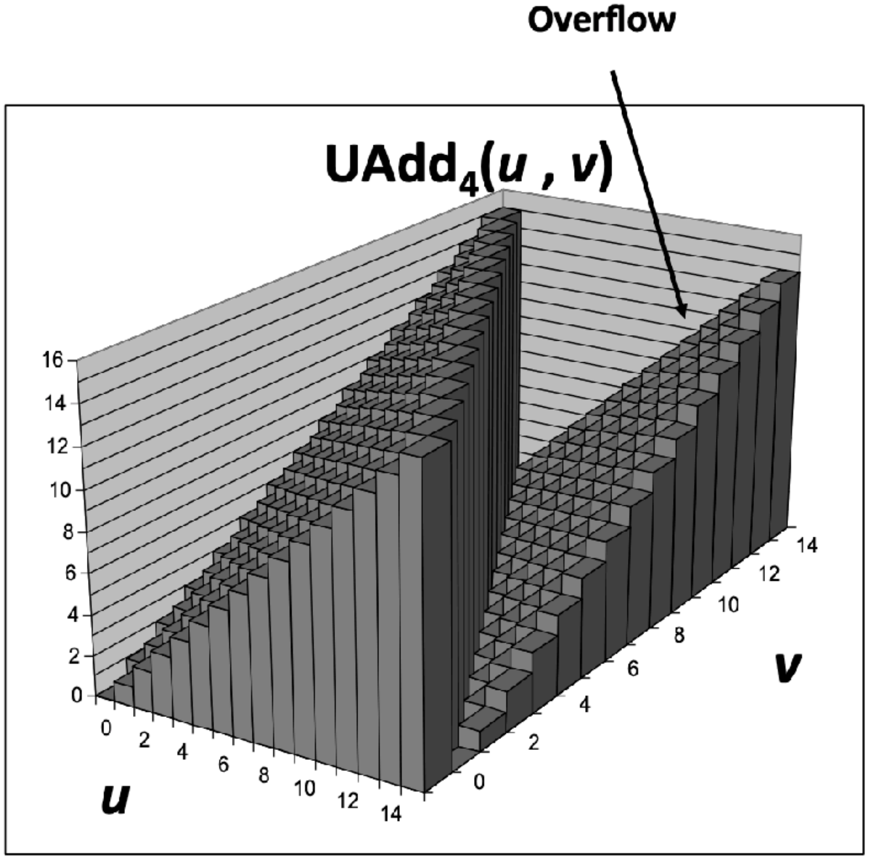
- Implements Modular Arithmetic

$s\ = \mathrm{UAdd}_w(u\ ,\ v) = \underline{u + v \mod 2^w}$

Visualizing unsigned addition

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \le x + y < 2^{w+1} \end{cases}$$





How can we check for overflow?

UA

```
// adds two integers and returns the result
unsigned u_add(unsigned i, unsigned j) {
  // TODO: check overflow
  // ???
  return i + j;
}
```

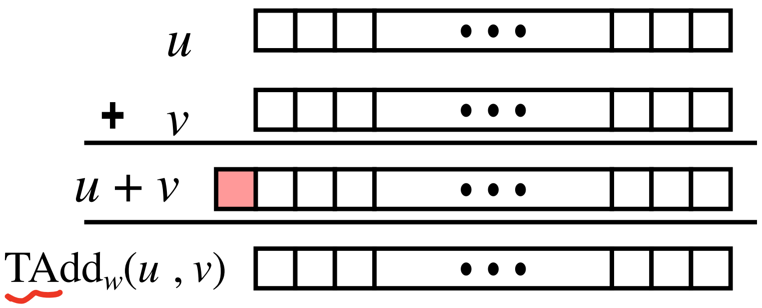
D check for overflow

Two's complement addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- TAdd and UAdd have identical bit-level behavior
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give **`s == t`**

TAdd Overflow

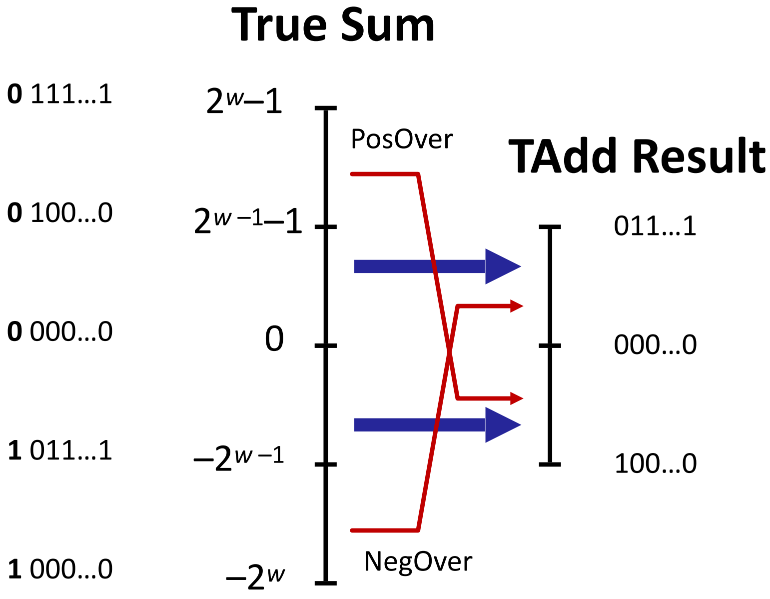
Two's compliment

- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer

positive overflow

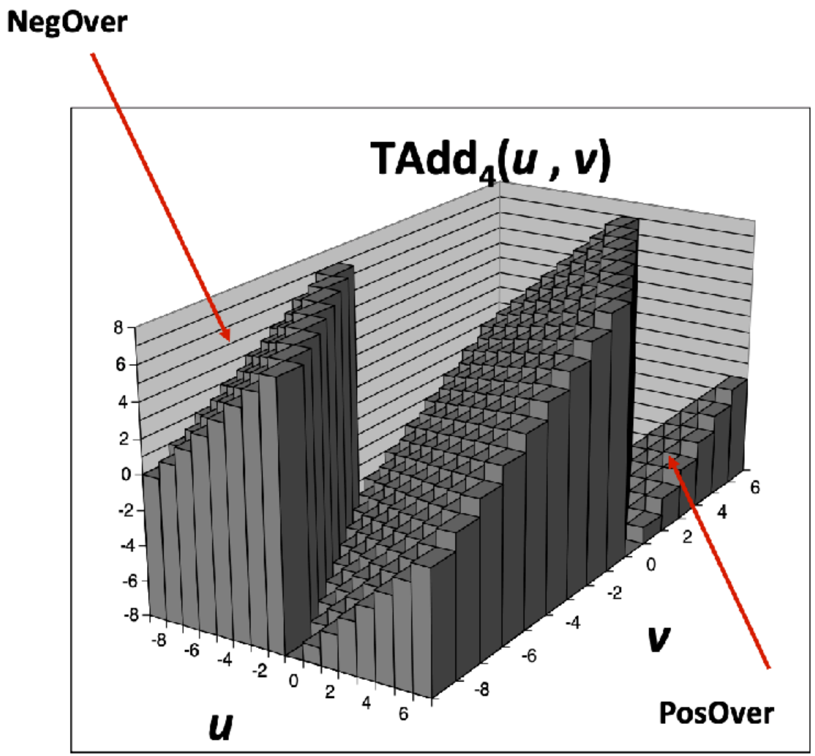
$$x +_w^t y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \end{cases}$$

negative overflow



Visualizing 2's complement addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps around
  - If sum ≥ 2^{w-1}
    - Becomes negative
    - At most once
  - If sum < -2^{w-1}
    - Becomes positive
    - At most once



Checking for overflow (2's complement)

$w=4$

$s = x +$

-8

7

Let $TMin_w \leq x, y \leq TMax_w$ and $s = x +^t_w y$.

The computation of s:

- has positive overflow iff $x > 0$ and $y > 0$ but $s \leq 0$
- has negative overflow iff $x < 0$ and $y < 0$ but $s \geq 0$

Multiplication

- Goal: Computing the product of w-bit numbers x, y
  - Either signed or unsigned

- But, exact results can be bigger than w bits
  - Unsigned: up to $2w$ bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to $2w$-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$

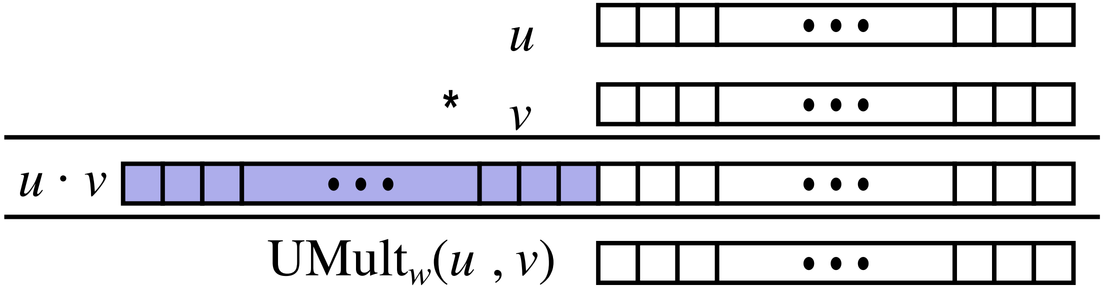
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned multiplication in C

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



- Standard multiplication function
  - Ignores high order w bits
- Implements modular arithmetic

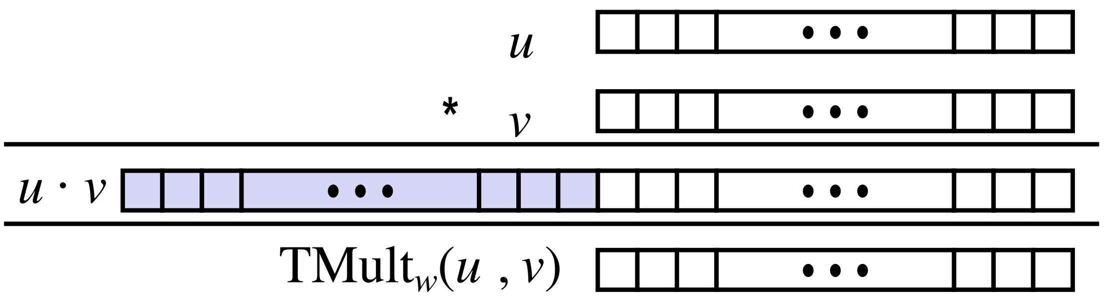
$\mathrm{UMult}_w(u, v) = u \cdot v \mod 2^w$

Signed multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- Standard multiplication function
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

3-bit multiplication examples

w=3

Mode	x	y	x*y	Truncated result
Unsigned	5 [101]	3 [011]	15 [001111]	7 [~~001~~111]
2's complement	-3 [101]	3 [011]	-9 [110111]	-1 [~~110~~111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [~~011~~100]
2's complement	-4 [100]	-1 [111]	4 [000100]	-4 [~~000~~100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [~~001~~001]
2's complement	3 [011]	3 [011]	9 [001001]	1 [~~001~~001]

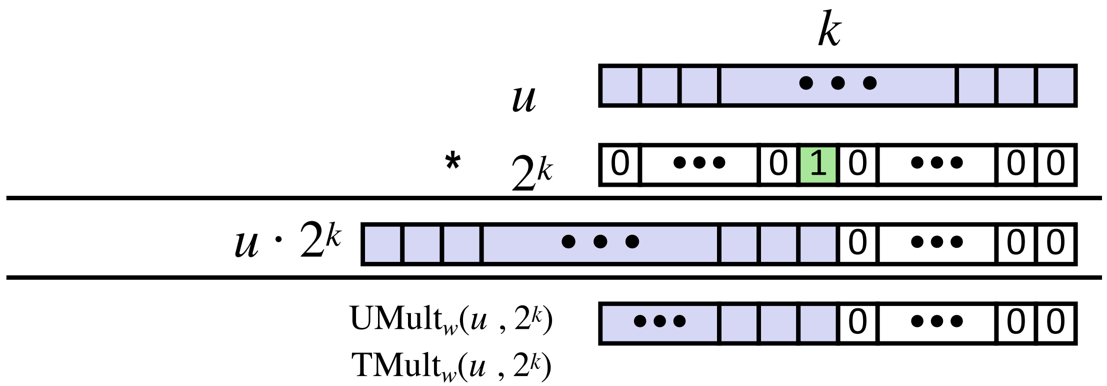
101
011 001111
101
101

Power-of-2 multiply with shift

Operands: *w* bits

True Product: *w+k* bits

Discard *k* bits: *w* bits

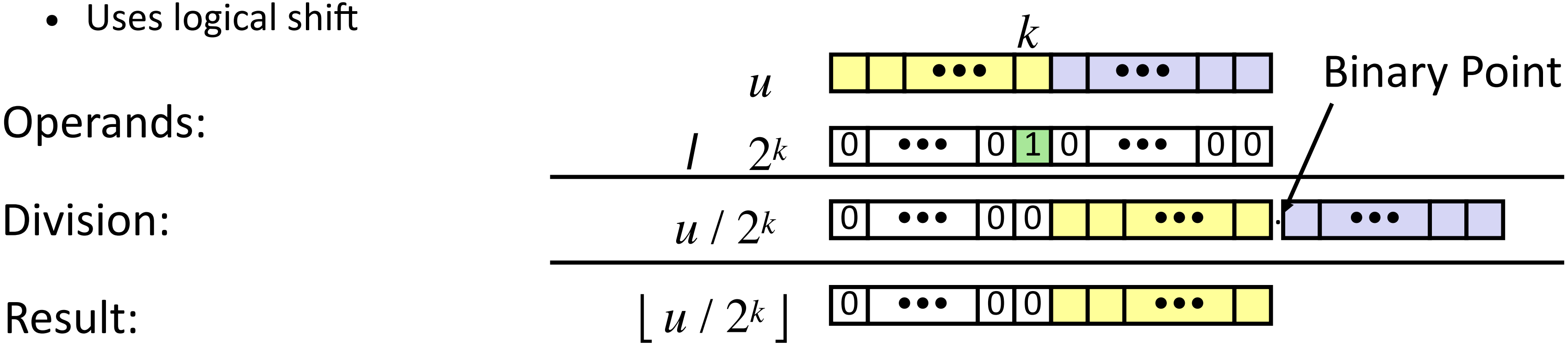


- Operation shift bits to the left
  - `u << k` gives `u * 2`^{k}
  - Both signed and unsigned

- Examples
  - `u << 3 == u * 8`
  - `(u << 5) – (u << 3) == u * 24` $2^5 - 2^3 = 32 - 8 = 24$
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Unsigned power-of-2 divide with shift

- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor \texttt{u} / 2^k \rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Integer arithmetic: Basic rules

- Addition:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Integer C puzzles

$x \le 0$

$x \geq 1$

W: 4

Not

$1 > x > 0$

$x \le 0$

$x < 1$

$x < 1$

x has to be negative

or

Initialization

Signed
Signed

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- A. `(x > 0) || (x-1 < 0)` when $x < 1$ then false
- B. `(x&7) != 7 || (x<<29 < 0)` shift 2^{29}
- C. `(x*x) >=0`
- D. `x < 0 || -x <= 0`
- E. `x > 0 || -x >=0`
- F. `x+y == ux+uy`
- G. `x*~y + uy*ux == -x`

false when $x=7$

b_2 has to be 1

0000111

A: $w=4$

$-8 \rightarrow 1000$ $-8-1=-9 \rightarrow \cancel{1}0111 \rightarrow 0111$

$(x-1<0)$ –

7

false

$$
\begin{array}{lccccccccc}
B: & b_{31} & b_{30} & b_{29} & \cdots & & b_2 & b_1 & b_0 \\
\& & 0 & 0 & 0 & & & 1 & 1 & 1 \\
\hline
 & 0 & 0 & 0 & \cdots & & b_2 & b_1 & b_0
\end{array}
$$

$$x \ll 29 = b_2 b_1 b_0\, 0\, 0\, 0 \cdots \underset{1}{\downarrow}\, 0\, 0\, 0$$