Bits, Bytes, and Integers

B&O Readings: 2.1-2.3

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

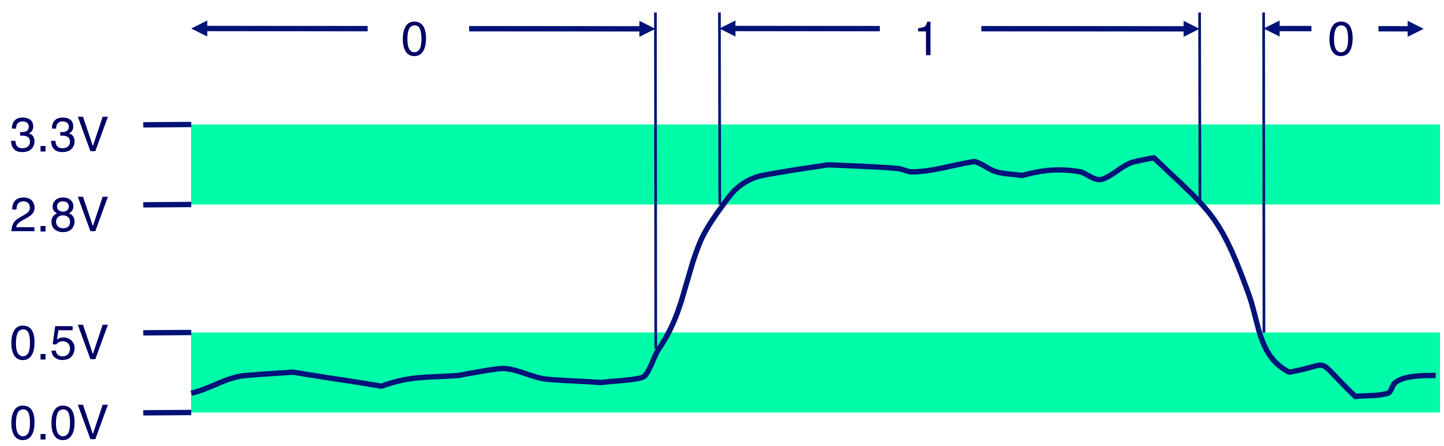
Note: these slides were originally created by Markus Püschel at Carnegie Mellon University

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- **Summary**

Binary Representations

- Everything is a collection of bits (a bit: 0 or 1)
- 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



For example, can count in binary

- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - 15213_{10}

    $3 \cdot 10^0$

    $1 \cdot 10^1$

    $2 \cdot 10^2$

    $5 \cdot 10^3$

    $1 \cdot 10^4$
  - $011101101101101_2 \quad = \quad 1{\cdot}2^0 + 0{\cdot}2^1 + 1{\cdot}2^2 + 1{\cdot}2^3 + 0{\cdot}2^4 + 1{\cdot}2^5 + 1{\cdot}2^6 + 0{\cdot}2^7 + 1{\cdot}2^8 + 1{\cdot}2^9 + 0{\cdot}2^{10} + 1{\cdot}2^{11} + 1{\cdot}2^{12} + 1{\cdot}2^{13} + 0{\cdot}2^{14}$

Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

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

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	−	−	10/16
pointer	4	8	8

Typically treated as a signed value, but no guarantee!

Watch out for portability issues!
(Use ISO C99 data types when necessary.)

Terminology:

- Binary representation for `short x = 15213`

16 bits

- 011101101101101_2 = $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 +$
$0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 +$
$1 \cdot 2^8 + 1 \cdot 2^9 + 0 \cdot 2^{10} + 1 \cdot 2^{11} +$
$1 \cdot 2^{12} + 1 \cdot 2^{13} + 0 \cdot 2^{14} + 0 \cdot 2^{15}$

most-significant bit (MSB)

least-significant bit (LSB)

Converting Between Different Bases

- **Find the hexadecimal representation for the following numbers:**
  - 912559 12648430 2989

```
0xDECAF     0xC0FFEE     0xBAD
```

- **How do you convert from decimal to hex?**
  - Take the value, mod it by 16 to find the quotient and remainder
  - Take the reminder as the next digit (from least-significant to most)
  - Repeat with the quotient as the new value it reaches 0

- **What about their binary representation?**

Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- **All of the Properties of Boolean Algebra Apply**

Neat Tricks with XOR

- What does `foo` do?

```
void foo(int *x, int *y) {
    *y = *x ^ *y;    /* step 1 */
    *x = *x ^ *y;    /* step 2 */
    *y = *x ^ *y;    /* step 3 */
}
```

Step	`*x`	`*y`
initial	`a`	`b`
1	`a`	`a^b`
2	`a^(a^b)`	`a^b`
3	`b`	`(a^b)^b = a`

`foo` swaps the two numbers!

XOR:

^	0	1
0	0	1
1	1	0

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **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**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41 → 0xBE`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00 → 0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 → 0x41`
    - ____________ & ____________ → ____________
  - `0x69 | 0x55 → 0x7D`
    - ____________ & ____________ → ____________

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any “integral” data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41` → `0xBE`
    - ~01000001_2 → 10111110_2
  - `~0x00` → `0xFF`
    - ~00000000_2 → 11111111_2
  - `0x69 & 0x55` → `0x41`
    - 01101001_2 & 01010101_2 → 01000001_2
  - `0x69 | 0x55` → `0x7D`
    - 01101001_2 | 01010101_2 → 01111101_2

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as “False”
    - Anything nonzero as “True”
    - **Always return 0 or 1**
    - Early termination
- **Examples (char data type)**
  - `!0x41` ➙
  - `!0x00` ➙
  - `!!0x41` ➙
  - `0x69 && 0x55` ➙
  - `0x69 || 0x55` ➙
  - `p && *p`

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any “integral” data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - ~0x41 → 0xBE
    - ~01000001_2 → 10111110_2
  - ~0x00 → 0xFF
    - ~00000000_2 → 11111111_2
  - 0x69 & 0x55 → 0x41
    - 01101001_2 & 01010101_2 → 01000001_2
  - 0x69 | 0x55 → 0x7D
    - 01101001_2 | 01010101_2 → 01111101_2

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`
  - `0x69 && 0x55` → `0x01`
  - `0x69 || 0x55` → `0x01`
  - `p && *p` (avoids null pointer access)

Watch out for && vs. & (and || vs. |)... one of the more common oopsies in C programming

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

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

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

For unsigned data >> performs logical shift. For signed data, the standard does not define it, but likely arithmetic.

Another Puzzle for Your Amusement …

- What does `bar` do?

```
void bar(int x, int y) {
    return (x & ~y) | (~x & y);
}
```

`(x & ~y)` : if a bit in **y** is set, clear that same bit in **x**

`(~x & y)` : if a bit in **x** is set, clear that same bit in **y**

`(x&~y)|(~x&y)` :

x \ **y**	**ret**	0	1
	0	0	1
	1	1	0

`bar` computes XOR!

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - **Representation: unsigned and signed**
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
- Summary

Encoding Integers

Unsigned

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

Two's Complement

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

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

Two-complement Encoding Example (Cont.)

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

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**

Numeric Ranges

- **How many of the following statements are FALSE?**
  - The maximum value an unsigned short (16 bits) can represent is 0xFFFF.
  - The minimum value of a signed short can represent is ~~0xFFFF.~~ **0x8000**
  - The maximum value of a signed short can represent is 0x7FFF.

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0

    0x000…0
  - *UMax* = $2^w - 1$

    0xFFF…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}

    0x800…0
  - *TMax* = $2^{w-1} - 1$

    0x7FF…F
- **Other Values**
  - Minus 1

    0xFF…F

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - | *TMin* | = *TMax* + 1
    - Asymmetric range
  - *UMax* = 2 * *TMax* + 1

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

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

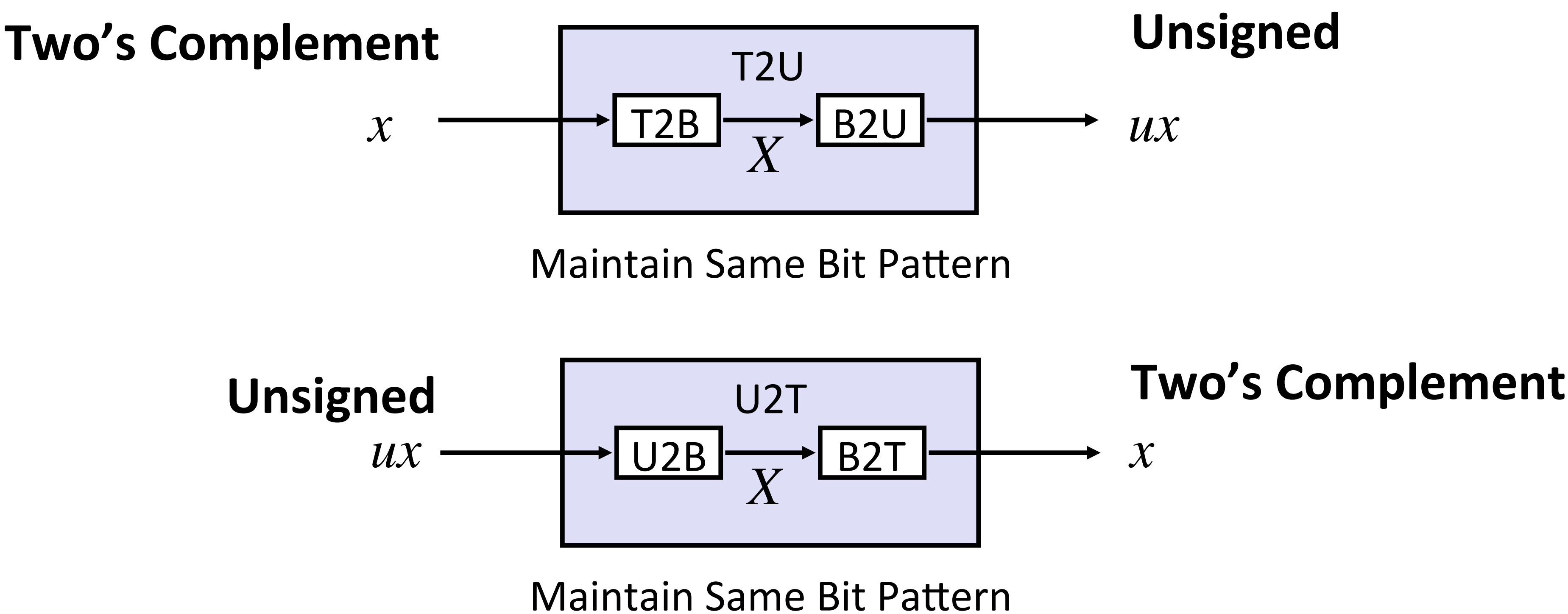
- **Equivalence**
  - Same encodings for nonnegative values

- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Mapping Between Signed & Unsigned



- **Mappings between unsigned and two's complement numbers:**
  Keep bit representations and reinterpret

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

→ T2U →

← 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

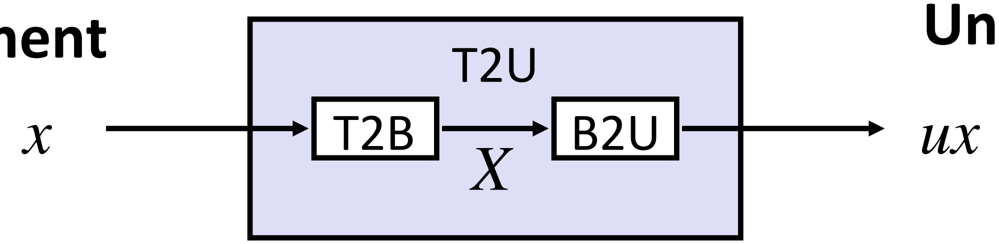
=

+/- 16

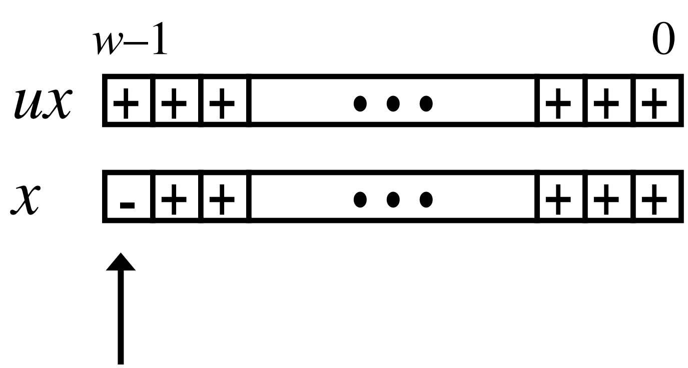
Relation between Signed & Unsigned

Two's Complement

Unsigned



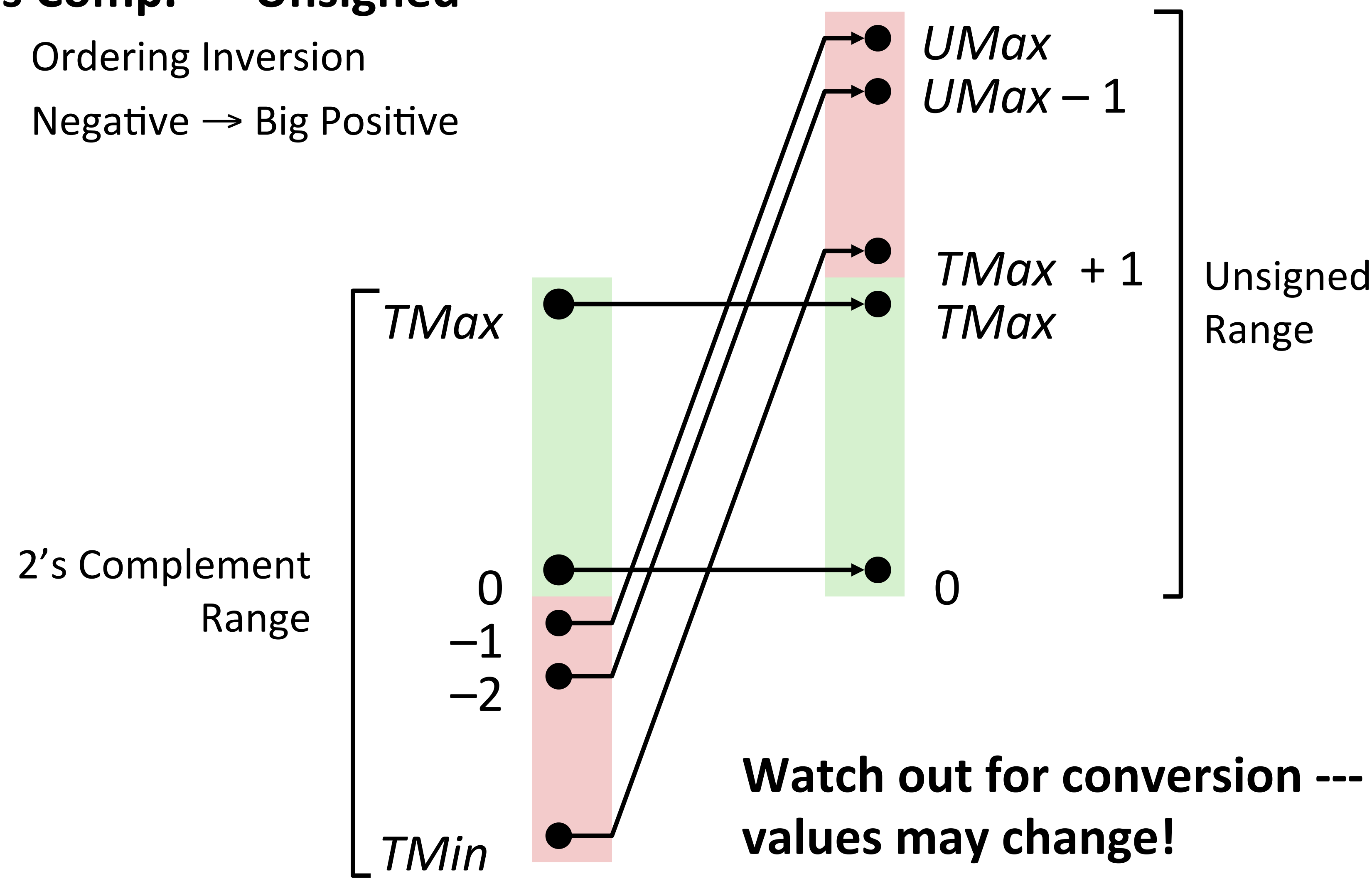
Maintain Same Bit Pattern



Large negative weight
becomes
Large positive weight

Conversion Visualized

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



Watch out for conversion --- values may change!

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

- **Expression Evaluation**
  - 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 = -2,147,483,648 ,** **TMAX = 2,147,483,647**

Expression Evaluation Puzzles

- **Assuming int type (32 bits)**

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

TMIN = -2147483648 (0x8000000)
TMAX = 2147483647 (0x7FFFFFFF)

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`!!