

Bits, Bytes and Integers – Part 1

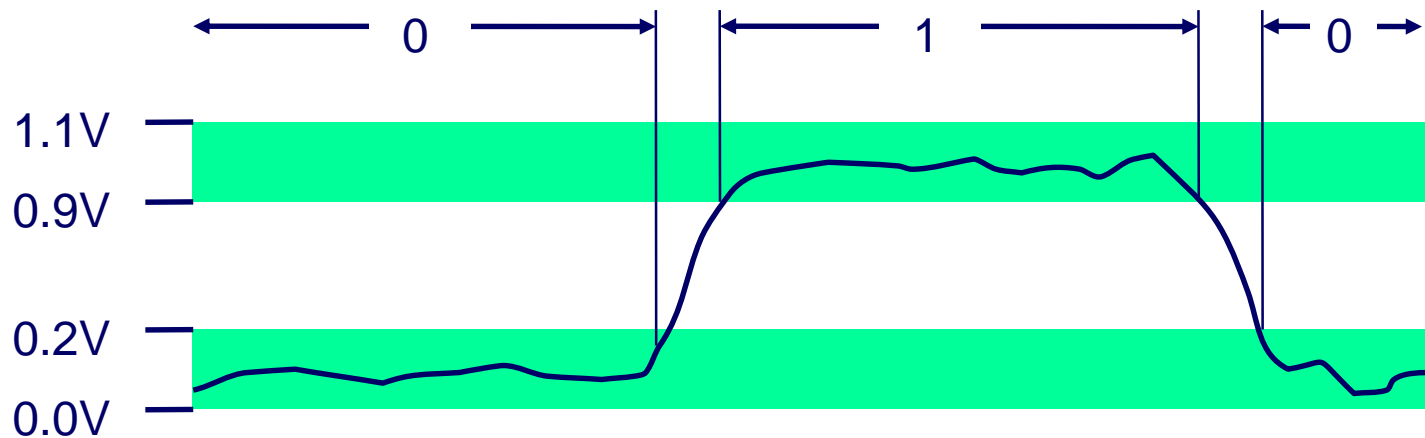
15-213/18-213/14-513/15-513: Introduction to Computer Systems
2nd Lecture, Jan. 17, 2019

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**

Everything is bits

- Each bit is 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
- Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
- Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

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 |

15213: 0011 1011 0110 1101
 3 B 6 D

Example Data Representations

| C Data Type | Typical 32-bit | Typical 64-bit | x86-64 |
|----------------------|----------------|----------------|--------|
| <code>char</code> | 1 | 1 | 1 |
| <code>short</code> | 2 | 2 | 2 |
| <code>int</code> | 4 | 4 | 4 |
| <code>long</code> | 4 | 8 | 8 |
| <code>float</code> | 4 | 4 | 4 |
| <code>double</code> | 8 | 8 | 8 |
| <code>pointer</code> | 4 | 8 | 8 |

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

- $\sim A = 1$ when $A=0$

| | |
|--------|---|
| \sim | |
| 0 | 1 |
| 1 | 0 |

Exclusive-Or (Xor)

- $A \wedge B = 1$ when either $A=1$ or $B=1$, but not both

| | | |
|----------|---|---|
| \wedge | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

General Boolean Algebras

■ Operate on Bit Vectors

- Operations applied bitwise

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

■ All of the Properties of Boolean Algebra Apply

Example: Representing & Manipulating Sets

■ Representation

- Width w bit vector represents subsets of $\{0, \dots, w-1\}$
- $a_j = 1$ 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` →
- `~0x00` →
- `0x69 & 0x55` →
- `0x69 | 0x55` →

| 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 |

Bit-Level Operations in C

■ Operations $\&$, $|$, \sim , \wedge 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)

- $\sim 0x41 \rightarrow 0xBE$
 - $\sim 0100\ 0001_2 \rightarrow 1011\ 1110_2$
- $\sim 0x00 \rightarrow 0xFF$
 - $\sim 0000\ 0000_2 \rightarrow 1111\ 1111_2$
- $0x69 \& 0x55 \rightarrow 0x41$
 - $0110\ 1001_2 \& 0101\ 0101_2 \rightarrow 0100\ 0001_2$
- $0x69 | 0x55 \rightarrow 0x7D$
 - $0110\ 1001_2 | 0101\ 0101_2 \rightarrow 0111\ 1101_2$

| 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 |

Contrast: Logic Operations in C

■ Contrast to Bit-Level Operators

- **Logic Operations: &&, ||, !**
 - 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: $x \ll y$**
 - Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right
- **Right Shift: $x \gg y$**
 - 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 \geq word size

| | |
|----------------|----------|
| Argument x | 01100010 |
| $\ll 3$ | 00010000 |
| Log. $\gg 2$ | 00011000 |
| Arith. $\gg 2$ | 00011000 |

| | |
|----------------|----------|
| Argument x | 10100010 |
| $\ll 3$ | 00010000 |
| Log. $\gg 2$ | 00101000 |
| Arith. $\gg 2$ | 11101000 |

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

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

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

Sign Bit

- **C does not mandate using two's complement**
 - But, most machines do, and we will assume so
- **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: Simple Example

$$\begin{array}{rcccccc}
 & & -16 & 8 & 4 & 2 & 1 \\
 10 = & 0 & 1 & 0 & 1 & 0 & \\
 & & & & & & 8+2 = 10
 \end{array}$$

$$\begin{array}{rcccccc}
 & & -16 & 8 & 4 & 2 & 1 \\
 -10 = & 1 & 0 & 1 & 1 & 0 & \\
 & & & & & & -16+4+2 = -10
 \end{array}$$

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

■ Unsigned Values

- $UMin = 0$
000...0
- $UMax = 2^w - 1$
111...1

■ Two's Complement Values

- $TMin = -2^{w-1}$
100...0
- $TMax = 2^{w-1} - 1$
011...1
- Minus 1
111...1

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

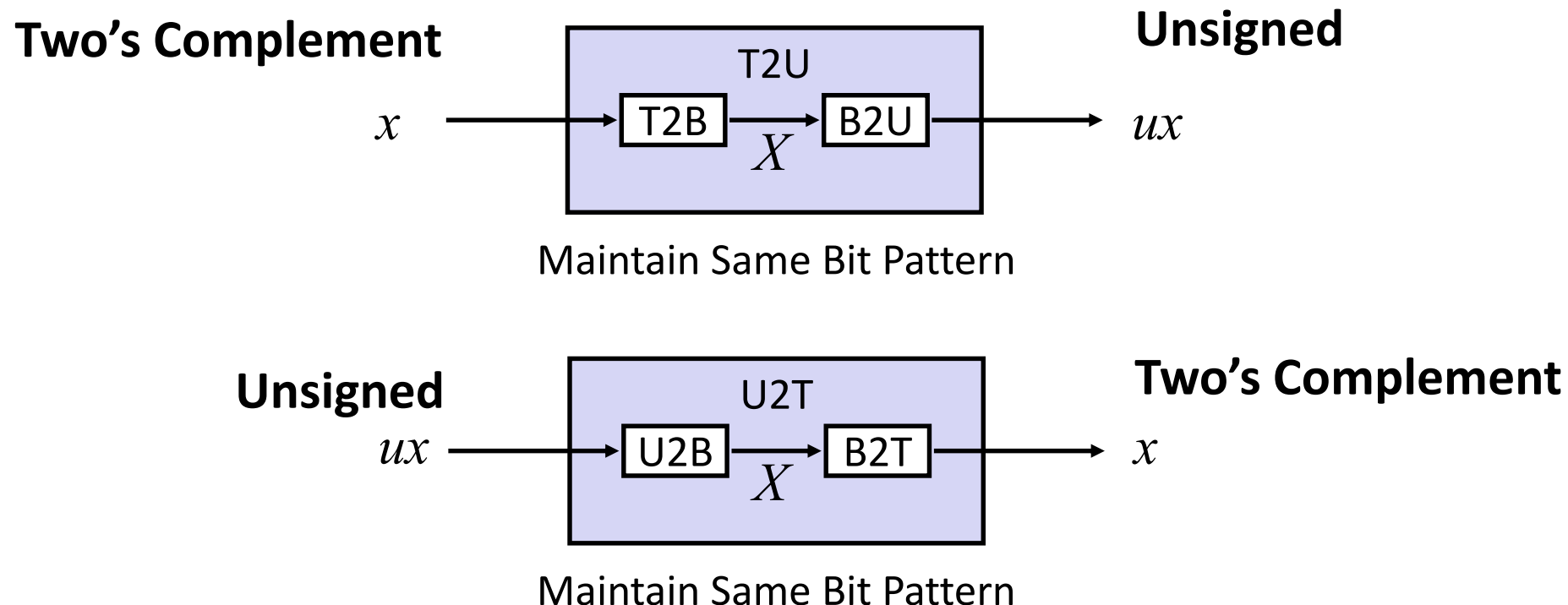
■ \Rightarrow Can Invert Mappings

- $U2B(x) = B2U^{-1}(x)$
 - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$
 - Bit pattern for two's comp integer

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



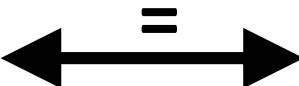
- Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

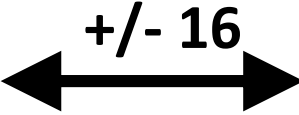
Mapping Signed \leftrightarrow Unsigned

| 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 |

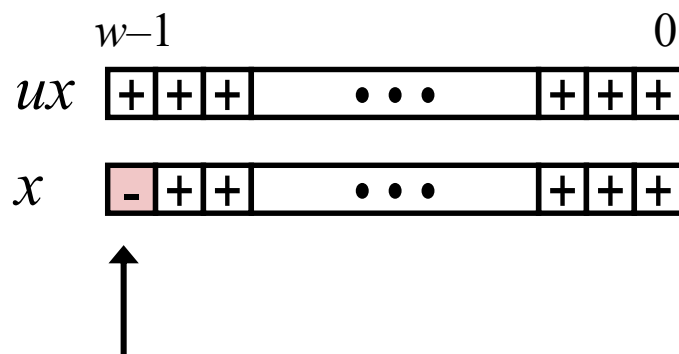
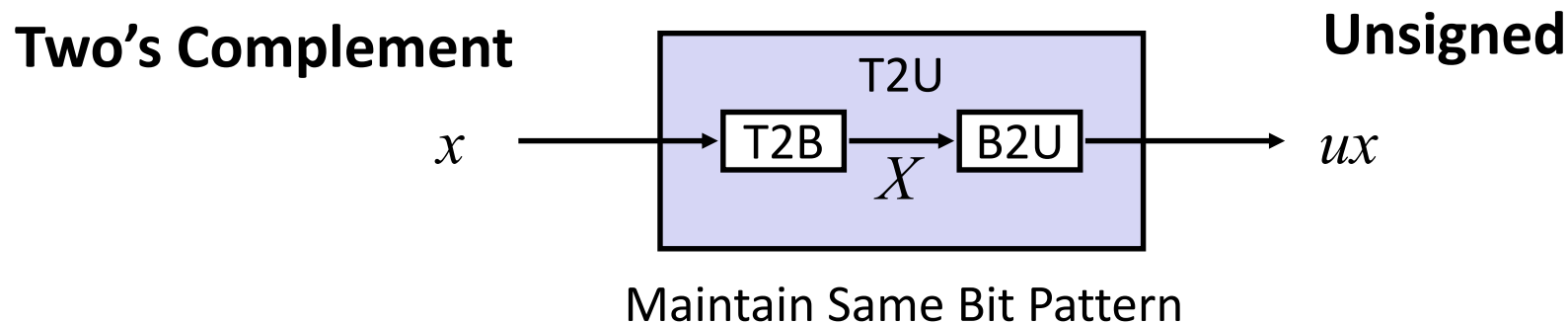
Mapping Signed \leftrightarrow Unsigned

| 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 |





Relation between Signed & Unsigned

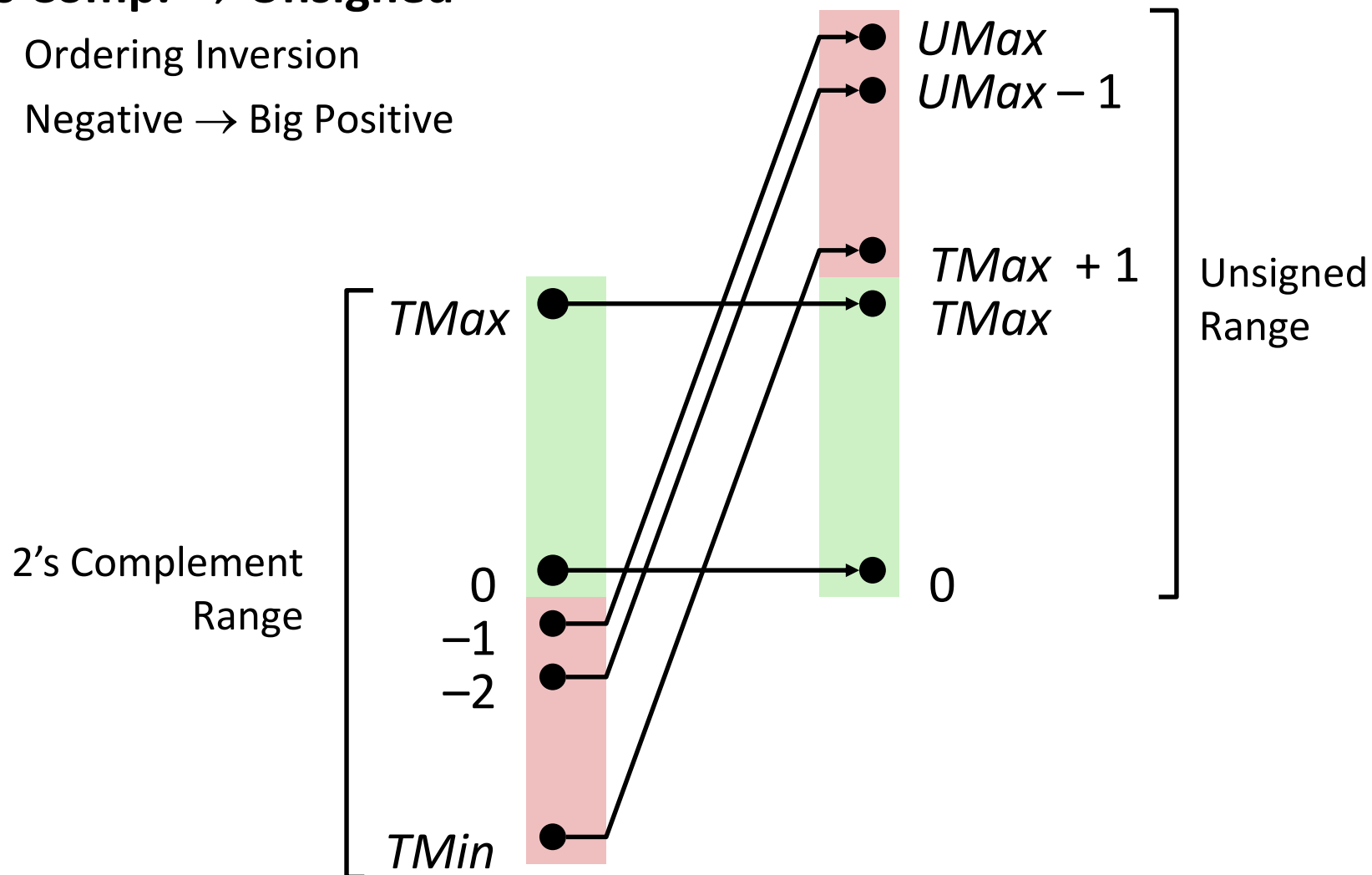


Large negative weight
becomes
Large positive weight

Conversion Visualized

■ 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



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;                int fun(unsigned u);
uy = ty;                uy = fun(tx);
```

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`**

| ■ Constant₁ | Constant₂ | Relation | Evaluation |
|-------------------------------|-----------------------------|-------------------|-------------------|
| 0 | 0U | <code>==</code> | unsigned |
| -1 | 0 | <code><</code> | signed |
| -1 | 0U | <code>></code> | unsigned |
| 2147483647 | -2147483647-1 | <code>></code> | signed |
| 2147483647U | -2147483647-1 | <code><</code> | unsigned |
| -1 | -2 | <code>></code> | signed |
| (unsigned)-1 | -2 | <code>></code> | unsigned |
| 2147483647 | 2147483648U | <code><</code> | unsigned |
| 2147483647 | (int) 2147483648U | <code>></code> | signed |

Summary

Casting Signed \leftrightarrow 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`!!

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

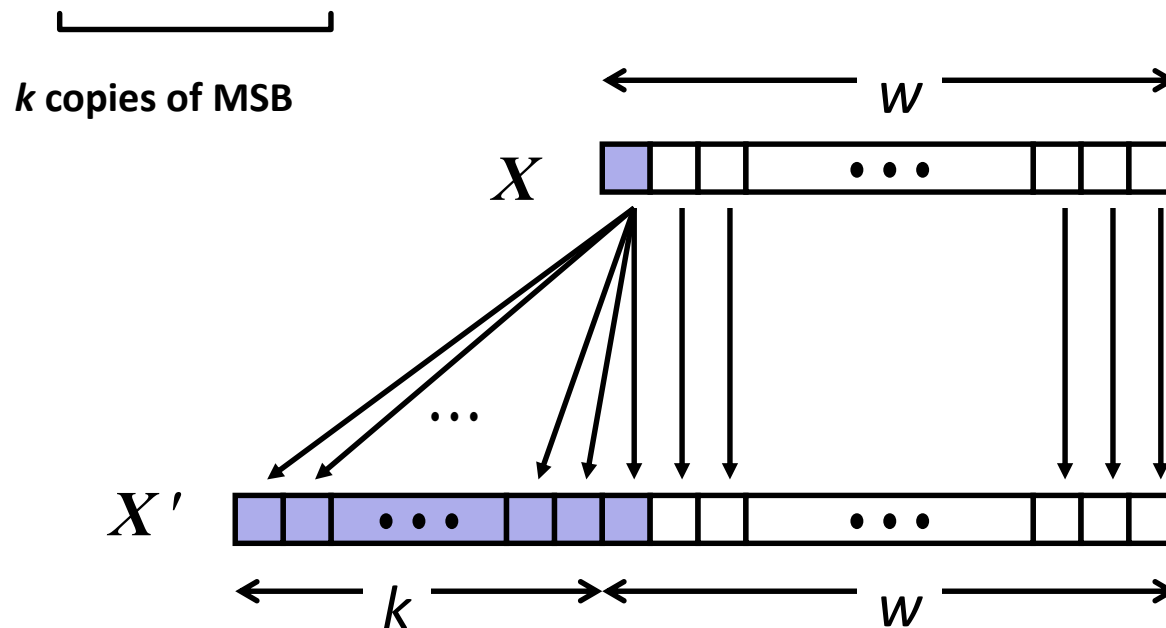
Sign Extension

■ Task:

- Given w -bit signed integer x
- Convert it to $w+k$ -bit integer with same value

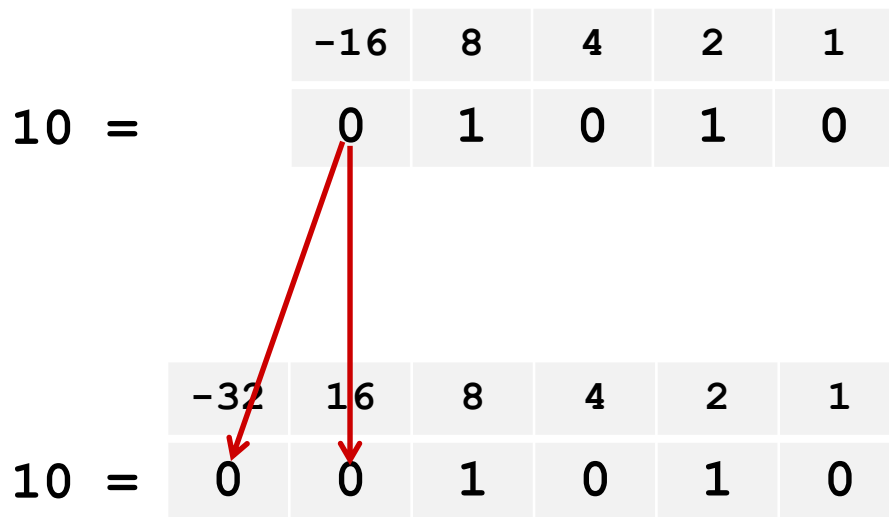
■ Rule:

- Make k copies of sign bit:
- $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \dots, x_0$

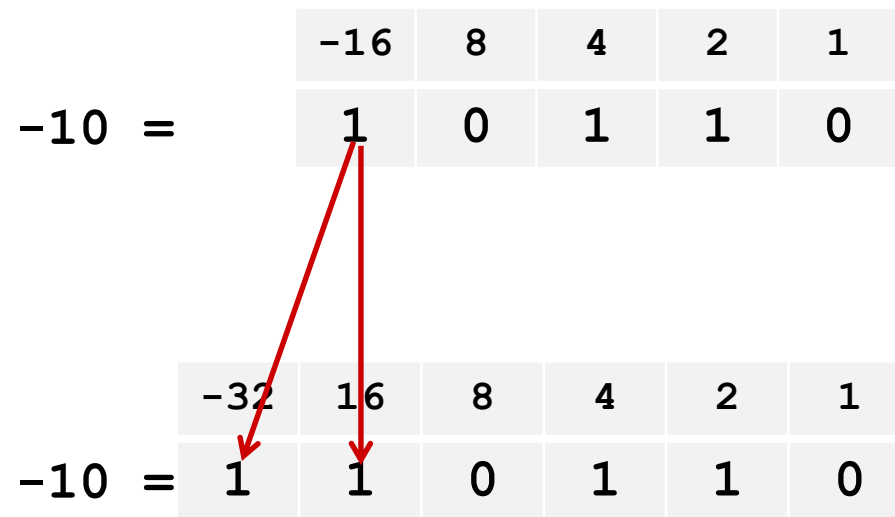


Sign Extension: Simple Example

Positive number



Negative number



Larger 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

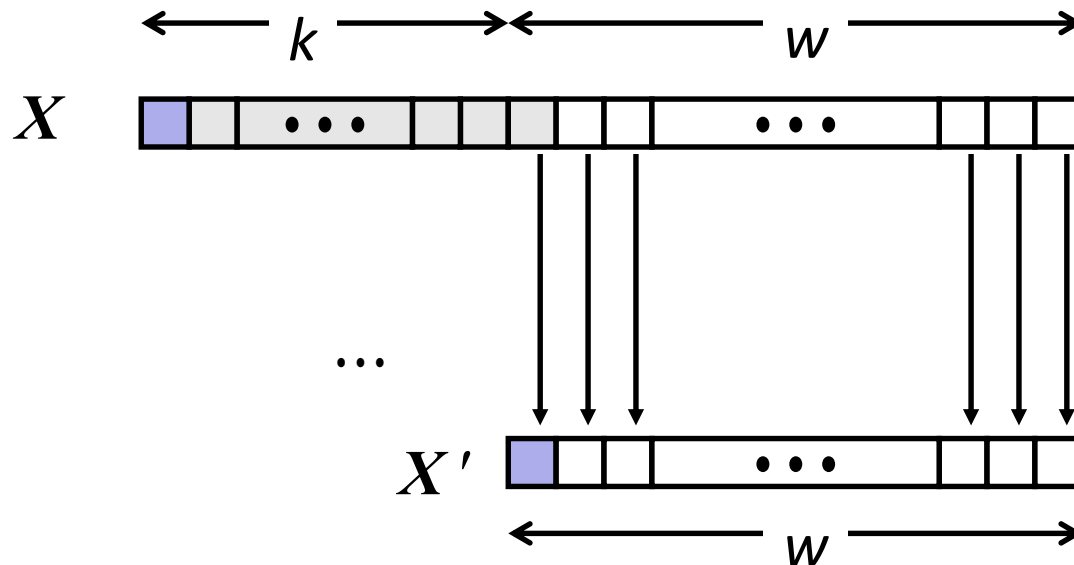
Truncation

■ Task:

- Given $k+w$ -bit signed or unsigned integer X
- Convert it to w -bit integer X'
 - (with same value for “small enough” X)

■ Rule:

- Drop top k bits:
- $X' = x_{w-1}, x_{w-2}, \dots, x_0$



Truncation: Simple Example

No sign change

| | -16 | 8 | 4 | 2 | 1 |
|-----|-----|---|---|---|---|
| 2 = | 0 | 0 | 0 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|-----|----|---|---|---|
| 2 = | 0 | 0 | 1 | 0 |

$$2 \bmod 16 = 2$$

| | -16 | 8 | 4 | 2 | 1 |
|------|-----|---|---|---|---|
| -6 = | 1 | 1 | 0 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|------|----|---|---|---|
| -6 = | 1 | 0 | 1 | 0 |

$$-6 \bmod 16 = 26U \bmod 16 = 10U = -6$$

Sign change

| | -16 | 8 | 4 | 2 | 1 |
|------|-----|---|---|---|---|
| 10 = | 0 | 1 | 0 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|------|----|---|---|---|
| -6 = | 1 | 0 | 1 | 0 |

$$10 \bmod 16 = 10U \bmod 16 = 10U = -6$$

| | -16 | 8 | 4 | 2 | 1 |
|-------|-----|---|---|---|---|
| -10 = | 1 | 0 | 1 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|-----|----|---|---|---|
| 6 = | 0 | 1 | 1 | 0 |

$$-10 \bmod 16 = 22U \bmod 16 = 6U = 6$$

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 (in magnitude) numbers yields expected behavior

Summary of 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
- Representations in memory, pointers, strings
- Summary