

# Lecture 02

# Integer Representations

CS213 – Intro to Computer Systems  
Branden Gena – Winter 2022

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

# Announcements

- Homework 1 available after class
  - Practice problems on binary-to-hex-to-decimal conversion and integer encodings
  - Due next week Thursday
  - Today's lecture will finish the content you need for it

# Announcements

- Data Lab is available later today (maybe tomorrow)
  - Use bit manipulations to achieve the desired goal (puzzles!)
    - Due in two weeks
    - Everything but the floating point should be do-able after today
  - Should be completed on Moore, one of the EECS remote servers
    - Your CS211 setup will get you into that machine
    - See Campuswire post on Accessing Remote Servers
- Partner survey is on Campuswire for anyone who wants a partner and doesn't already have one

# Today's Goals

- Introduce binary operators and Boolean algebra
- Discuss data representation in memory
- Explore integer data representations
  - Signed and Unsigned numbers
  - Different bit widths
  - Translating between encoding schemes

# Outline

- **Boolean Algebra**
- Data in memory
- Encoding
- Integer Encodings
  - Converting Sign
  - Converting Length

# Boolean algebra

- You've programmed with **and** and **or** in earlier classes
  - Written **&&** and **||** in C and C++
- **Boolean algebra is a generalization of that**
  - A mathematical system to represent (propositional) logic
  - 2 truth values: true = **1**, false = **0**
  - 3 operations: and = **&**, or = **|**, not (or complement) = **~**

# Performing Boolean algebra

- **Follow the rules for each operation to compute results**

- Rules are the like those you know from programming

- OR: |    AND: &    NOT: ~    1: True    0: False

$$(1 | 0) \& 0 \longrightarrow 1 \& 0 \longrightarrow 0$$

$$(1 \& 1) \& \sim(0 | 0) \longrightarrow 1 \& \sim(0) \longrightarrow 1 \& 1 \longrightarrow 1$$

# Truth tables for Boolean algebra

- For each possible value of each input, what is the output
  - Column for each input
  - Column for the output operation

<b><math>\sim A</math></b>	
A	$\sim A$
0	1
1	0

<b>A   B</b>		
A	B	A   B
0	0	0
0	1	1
1	0	1
1	1	1

<b>A &amp; B</b>		
A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1



# Exclusive Or

<b>A ^ B</b>		
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

- Some operations aren't available as C logical operators
  - Xor ^ - either A or B, but not both
- We can build Xor out of &, |, and ~
  - $A \wedge B = (\sim A \& B) | (A \& \sim B)$ 
    - (exactly one of A and B is true)
  - $A \wedge B = (A | B) \& \sim(A \& B)$ 
    - (either is true but not both are true)
- The two definitions are equivalent
  - Produce the same Truth Table

# Practice problem

		<b>(A &amp; B)   B</b>
A	B	(A&B) B
0	0	
0	1	
1	0	
1	1	

# Practice problem

		<b>(A &amp; B)   B</b>
A	B	(A&B) B
0	0	0
0	1	1
1	0	0
1	1	1

# Practice problem

		<b>(A &amp; B)   B</b>
A	B	(A&B) B
0	0	0
0	1	1
1	0	0
1	1	1

This is equivalent to B  
(A has no influence on the solution)

# De Morgan's Law

- We can express Boolean operators in terms of the others
- De Morgan's laws: allow swapping & with |
  - $A \& B = \sim(\sim A \mid \sim B)$        $\rightarrow$        $\sim(A \& B) = \sim A \mid \sim B$ 
    - (neither A nor B is false)
  - $A \mid B = \sim(\sim A \& \sim B)$        $\rightarrow$        $\sim(A \mid B) = \sim A \& \sim B$ 
    - (A and B are not both false)
- Useful for simplifying logical statements

# Generalized Boolean algebra

- Boolean operations can be extended to work on vectors of bits (i.e., bytes)
- Operations are applied one bit at a time: ***bitwise***

01101001	01101001	01101001	
<u>&amp; 01010101</u>	<u>  01010101</u>	<u>^ 01010101</u>	<u>~ 01010101</u>
01000001	01111101	00111100	10101010

- All of the properties of Boolean algebra still apply
  - Relationships between operations, etc.
- Bitwise operations are usable in C: **&, |, ~, ^**
  - Can operate on any integer type (long, int, short, char, signed or unsigned)

# Warning: bitwise operations are NOT logical operations

- Logical operations in C: `||`, `&&`, `!` (logical Or, And, and Not)
  - Only operate on a single bit
    - View 0 as "False"
    - View *anything nonzero* as "True"
    - Always return 0 or 1
  - Short-circuit evaluation: only checks the first operand if that is sufficient

## • Examples

- `!0x41 -> 0x00`                      `!0x00 -> 0x01`
- `0x59 && 0x35 -> 0x01`
- `(p != NULL) && *p` (short circuit avoids null pointer access)

`!!0x41 -> 0x01`



Useful for turning many bits into 1 bit

- Don't confuse the two!! It's a common C mistake

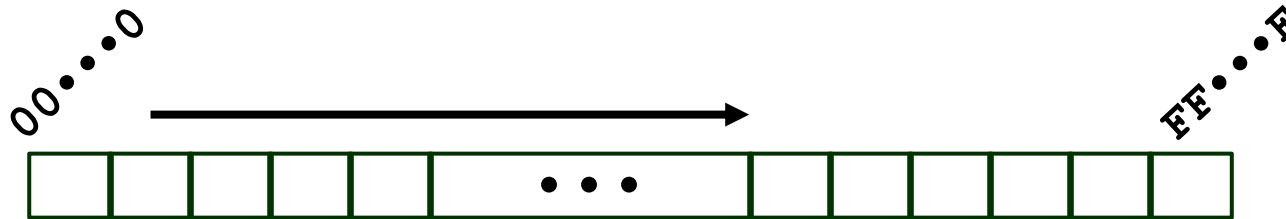
# Outline

- Boolean Algebra
- **Data in memory**
- Encoding
- Integer Encodings
  - Converting Sign
  - Converting Length



# Byte-oriented memory organization

- We've seen how sequences of bits can express numbers
  - And how we usually work with groups of 8 bits (**bytes**) for convenience
- In a computer system, bytes can be stored in memory
  - Conceptually, memory is a very large array of bytes
  - Each byte has its own address ( $\approx$  pointer)



- Compiler + run-time system control allocation
  - Where different program objects should be stored
  - Multiple mechanisms, each with its own region: static, stack, and heap



# Addressing and byte ordering

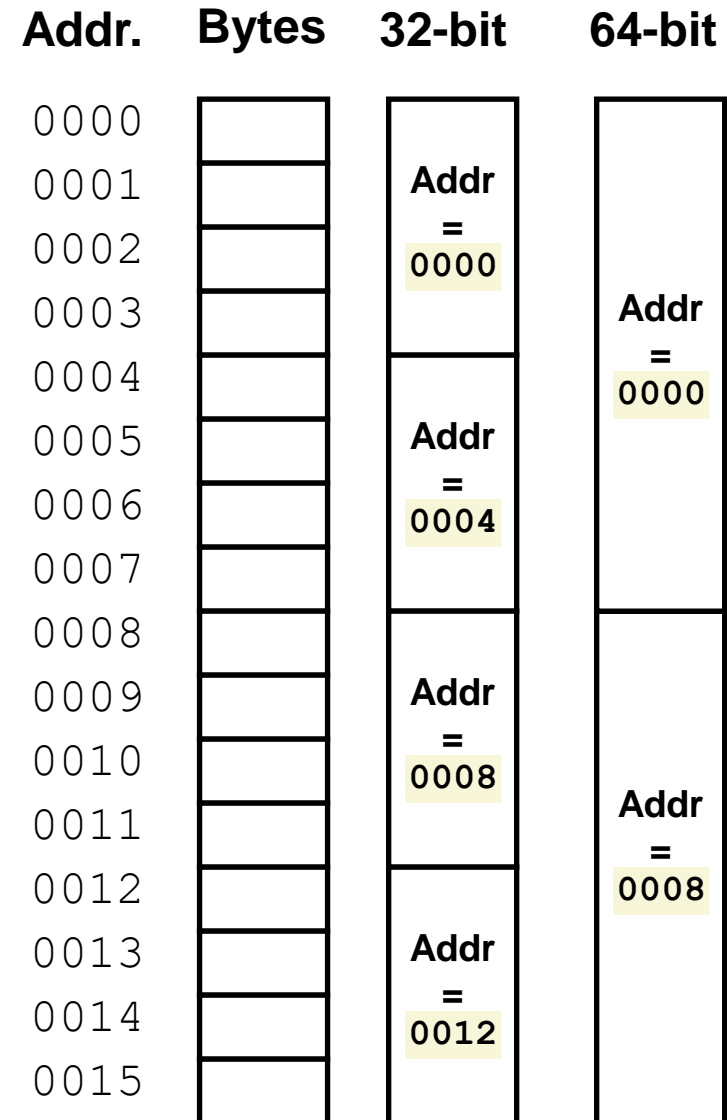
- For data that spans multiple bytes, need to agree on two things
  - **1. What should be the address of the object?** (each byte has its own!)
    - And by extension, given an address, how do we find the relevant bytes (same question!)
  - **2. How should we order the bytes in memory?**
    - Do we put the most or least significant byte at the first address?

# There isn't always one correct answer

- Different systems can pick different answers! (mostly for 2<sup>nd</sup> Q)
  - Very nice illustration of two overarching principles in systems:  
You need to know the specifics of the system you're using!
    - Many questions don't really have right or wrong answers!
    - Instead, they have tradeoffs. What the "right" answer is depends on context!
  - Different answers across systems is perfectly fine
    - But all the parts of a given system must agree with each other!

# 1. Data organization in memory

- Addresses specify byte locations
  - Address of first byte in object is used
  - Addresses of successive objects differ by 4 (32-bit) or 8 (64-bit)
- Systems pretty much universally use the address of the first byte as the address for the whole object
  - I'm not aware of any system that does otherwise
  - But there could be some weirdo systems out there (or historically)

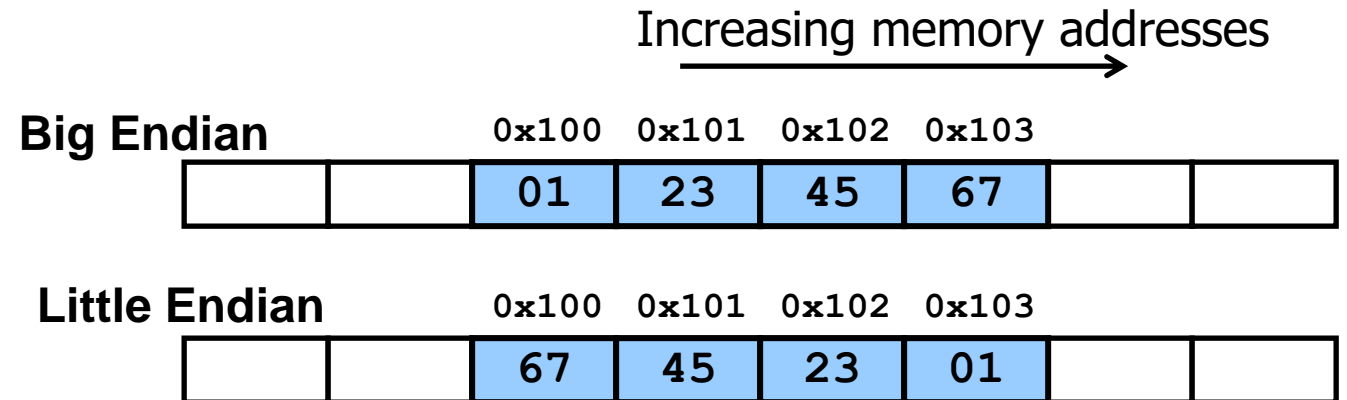


## 2. Byte ordering

- How to order bytes within a multi-byte object in memory
  - Only relevant when working with data larger than a byte!
- Conventions
  - **Big Endian:** Oracle/Sun (SPARC), IBM (Power), computer networks
    - Most significant byte has lowest address (comes first)
  - **Little Endian:** Intel (x86, x86-64)
    - Least significant byte has lowest address (comes first)

- **Example**

- 4-byte piece of data: `0x01234567`
- Address of that data is `0x100`



# Outline

- Boolean Algebra
- Data in memory
- **Encoding**
- Integer Encodings
  - Converting Sign
  - Converting Length

# What do bits and bytes *mean* in a system?

- The answer is: it depends!
- Depending on the context, the bits `11000011` could mean
  - The number 195
  - The number -61
  - The number -19/16
  - The character `'|'`
  - The `ret` x86 instruction
- You have to know the context to make sense of any bits you have!
  - Looking at the same bits in different contexts can lead to interesting results
  - Information = bits + context!
- We'll see *encodings* that give bits meanings



# Encoding characters: ASCII

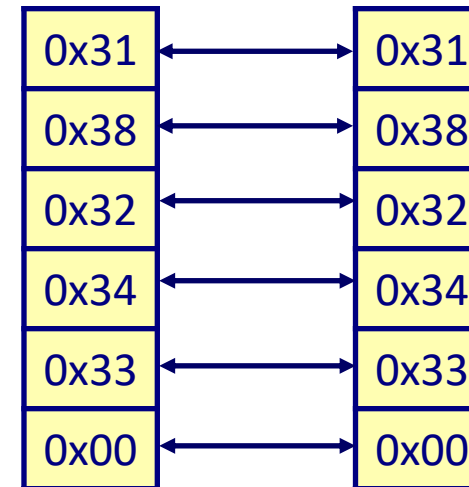
- **ASCII = American Standard Code for Information Interchange**
  - Standard dating from the 60s
- **Maps 8-bit\* bit patterns to characters**
  - We already know how to go from sequences of bits (base 2) to integers
  - Need to take one more step, and interpret these integers as characters
  - (\* the standard is actually 7-bit, leaving the 8<sup>th</sup> bit unused)
- **Examples**
  - $0100\ 0001_2 = 0x41 = 65_{10} = \text{'A'}$
  - $0100\ 0010_2 = 0x42 = 66_{10} = \text{'B'}$
  - $0011\ 0000_2 = 0x30 = 48_{10} = \text{'0'}$
  - $0011\ 0001_2 = 0x31 = 49_{10} = \text{'1'}$
- Reference: <https://www.asciitable.com/>

# Encoding strings (The C way)

- Represented by array of characters
  - Each character encoded in ASCII format
  - NULL character (code 0) to mark the end
- Compatibility
  - Byte ordering not an issue (data all single-byte!)
  - ASCII text files generally platform independent
    - Except for different conventions of line termination character(s)!

```
char S[6] = "18243";
```

Big-Endian      Little-Endian



# Open Question + Break

- **What things might we want to encode?**

# Open Question + Break

- **What things might we want to encode?**
  - Numbers
    - Signed and unsigned integers
    - Real numbers
    - Mathematical symbols:  $\infty$   $\pi$
  - Language
    - Characters in various different languages  $\Omega$ Иش서北
    - Emoji 🤖 😠 😄 😭 🧑 🎮 🎵 🍰 ✨ 🦄 🍦
  - Colors, Playing Cards, User Actions, anything!

# Outline

- Boolean Algebra
- Data in memory
- Encoding
- **Integer Encodings**
  - Converting Sign
  - Converting Length

# Integer types in C

- Integer types in C come in two flavors
  - Signed: `short`, `signed short`, `int`, `long`, ...
  - Unsigned: `unsigned char`, `unsigned short`, `unsigned int`, ...
- And in multiple different sizes
  - 1 byte: `signed char`, `unsigned char`
  - 2 bytes: `short`, `unsigned short`
  - 4 bytes: `int`, `unsigned int`
  - Etc.

# Sizes of C types are system dependent

- Portability
  - Some programmers assume an `int` can be used to store a pointer
  - OK for most 32-bit machines, but fails for 64-bit machines!
- How I program
  - Use fixed width integer types from `<stdint.h>`
  - `int8_t`, `int16_t`, `int32_t`
  - `uint8_t`, `uint16_t`, `uint32_t`

C Data Type	Intel IA32	x86-64	C Standard* (C99)
char	1	1	≥1
short	2	2	≥2
int	4	4	≥2
long	4	8	≥4
long long	8	8	≥8
float	4	4	
double	8	8	
pointer	4	8	Widths for data, code pointers may differ!

# Expressing C types in bits

- Two families of encodings to express those using bits
  - ***Unsigned*** encoding for unsigned integers
  - ***Two's complement*** encoding for signed integers
- Each encoding will use a fixed size (# of bits)
  - For a given machine
  - Size + encoding family determine which C type we're representing
  - Fixed size is because computers are finite!



# Unsigned integer encoding

- Just write out the number in binary
  - Works for 0 and all positive integers
- Example: encode  $104_{10}$  as an **unsigned** 8-bit integer
  - $104_{10} = 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$   
 $\Rightarrow$  **01101000**  
 $\Rightarrow$  **0x68**

$$\begin{array}{l} B2U(X) \\ \text{(Binary To Unsigned)} \end{array} = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

# Bounds of unsigned integers

- For a fixed width  $w$ , a limited range of integers can be expressed
  - Smallest value (we will call ***UMin***):
    - all 0s bit pattern: 000...0, value of 0
  - Largest value (we will call ***UMax***):
    - all 1s bit pattern: 111...1, value of  $2^w - 1$
    - $2^w - 1 = 1 \times 2^{w-1} + 1 \times 2^{w-2} + \dots + 1 \times 2^1 + 1 \times 2^0 = 11111\dots$
- Maximum 8-bit number =  $2^8 - 1 = 256 - 1 = 255$

# Attempting signed encoding

- Goal: encode integers that can be positive or negative
- First attempt: we can use the most significant bit for sign
  - “Sign-and-magnitude” encoding
    - In 8-bits:
      - +4 = 00000100      +127 = 01111111      +0 = 00000000
      - -4 = 10000100      -127 = 11111111      -0 = 10000000
- Big problem: we have two representations of zero!
- Also: hardware to do math with signed and unsigned numbers gets complicated...

# Two's complement encoding

- Bad news: need to make the encoding more complicated
- Good news: it will actually work
- Plan:
  - Start with unsigned encoding, but make the largest power negative
  - Example: for 8 bits, most significant bit is worth  $-2^7$  not  $+2^7$
- To encode a negative integer
  - First, set the most significant bit to 1 to start with a big negative number
  - Then, add positive powers of 2 (the other bits) to "get back" to number we want
- Example: encode -6 as a 4-bit two's complement integer
  - $-6_{10} = 1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \Rightarrow 0b1010 \Rightarrow 0\mathbf{xa}$

# Two's complement examples

- Encode -100 as an 8-bit two's complement number

$$\begin{aligned} \bullet -100_{10} = & 1 \times -2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ & -128 \quad + 0 \quad + 0 \quad + 16 \quad + 8 \quad + 4 \quad + 0 \quad + 0 \end{aligned}$$

Problem becomes:

encode +28 as a 7-bit unsigned number

- $-100_{10} = 0b10011100 = 0x9C$
- **Shortcut:** determine positive version of number, flip it, and add one
  - $100_{10} = 0b01100100$
  - Flipped =  $0b10011011$
  - Plus 1 =  $0b10011100 = 0x9C$  We'll talk about binary addition next lecture

# Interpreting binary signed values

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

↑  
Sign bit
- Note: most significant bit still tells us sign!! 1-> negative
  - Checking if a number is negative is just checking that top bit
- Zero problem is solved too
  - $0b00000000 = 0$                        $0b10000000 = -128$
- -1:  $0b111\dots1 = -1$  (regardless of number of bits!)

# Bounds of two's complement integers

- For a fixed width  **$w$** , a limited range of integers can be expressed
  - Smallest value, most negative (we will call  **$TMin$** ):
    - 1 followed by all 0s bit pattern:  $100\dots0 = -2^{w-1}$
  - Largest value, most positive (we will call  **$TMax$** ):
    - 0 followed by all 1s bit pattern:  $01\dots1$ , value of  $2^{w-1} - 1$
- Beware the asymmetry! Bigger negative number than positive

# Ranges for different bit amounts

	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 are 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 non-negative values

- **Uniqueness**

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

- **⇒ Can Invert Mappings**

- Can go from bits to number and back, and vice versa
- $U2B(x) = B2U^{-1}(x)$ 
  - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$ 
  - Bit pattern for two's complement integer

# Practice + Break

- What range of integers can be represented with 5-bit two's complement?
  - A -31 to +31
  - B -15 to +15
  - C 0 to +31
  - D -16 to +15
  - E -32 to +31

# Practice + Break

- What range of integers can be represented with 5-bit two's complement?

- A -31 to +31

No asymmetry and 6-bits

- B -15 to +15

No asymmetry

- C 0 to +31

Unsigned

- D -16 to +15

Correct

- E -32 to +31

6-bits

# Outline

- Boolean Algebra
- Data in memory
- Encoding
- **Integer Encodings**
  - **Converting Sign**
  - Converting Length

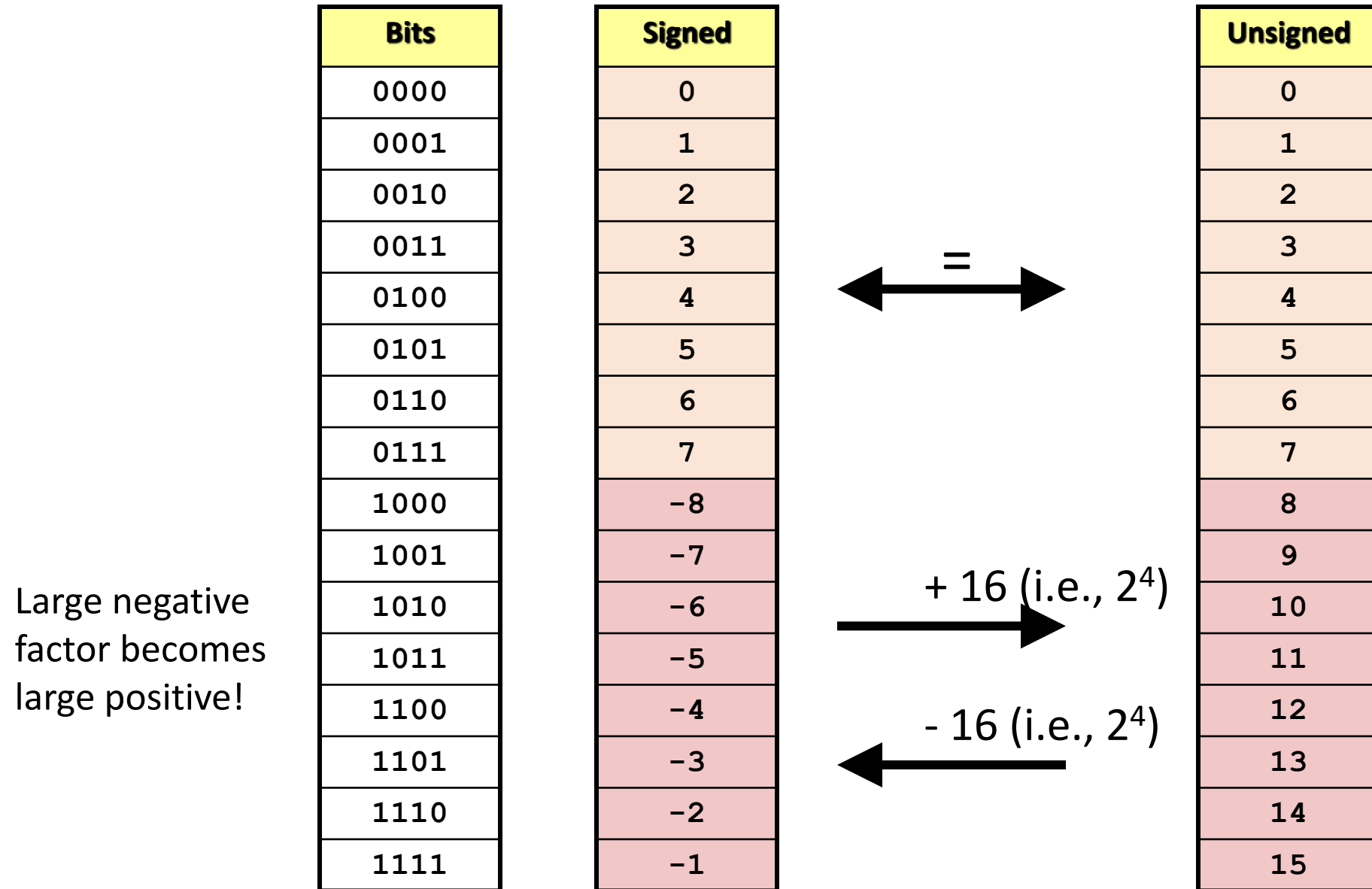
# Casting signed to unsigned

- C allows conversions from signed to unsigned (and vice versa)

```
short int          x = 15213;
unsigned short int ux = (unsigned short) x;
short int          y = -15213;
unsigned short int uy = y; /* implicit cast! */
```

- Resulting value
  - Not based on a numeric perspective: keep the bits and *reinterpret* them!
  - Non-negative values unchanged
    - $ux = 15213$
  - Negative values change into (large) positive values (and vice versa)
    - $uy = 50323$
- Warning: Casts can be implicit in assignments or function calls!
  - More on that in a few slides

# Mapping Signed $\leftrightarrow$ Unsigned (4 bits)



# Signed vs Unsigned in C

- Constants
  - By default are considered to be **signed integers**
  - Unsigned with "U/u" as suffix: `0U`, `4294967295U`
- **Expression evaluation**
  - If there is a mix of unsigned and signed in a single expression, ***signed values are converted to unsigned***
  - Including comparison operations!! `<`, `>`, `==`, `<=`, `>=`
- Can lead to surprising behavior!
  - `-1 < 0U` ⇒ **false!**
  - -1 gets converted to unsigned
  - All 1s bit pattern ⇒ UMax! Definitely not less than 0!

# Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary

2. Convert binary back into unsigned decimal



# Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary

$$-120 = -128 + 8 =$$

2. Convert binary back into unsigned decimal

# Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary

$$-120 = -128 + 8 =$$

$$1x(-128) + 0x64 + 0x32 + 0x16 + 1x8 + 0x4 + 0x2 + 0x1$$

2. Convert binary back into unsigned decimal

# Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary

$$-120 = -128 + 8 =$$

$$1x-128 + 0x64 + 0x32 + 0x16 + 1x8 + 0x4 + 0x2 + 0x1$$

$$0b\ 1000\ 1000$$

2. Convert binary back into unsigned decimal

$$1x128 + 0x64 + 0x32 + 0x16 + 1x8 + 0x4 + 0x2 + 0x1$$

$$128 + 8 = 136$$

# Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD's implementation of **getpeername**
- There are legions of experts trying to find vulnerabilities in programs, not all with good intentions

# Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

# Malicious Usage

```
/* Declaration of library function memcpy */  
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */  
#define KSIZE 1024  
char kbuf[KSIZE];  
  
/* Copy at most maxlen bytes from kernel region to user buffer */  
int copy_from_kernel(void *user_dest, int maxlen) {  
    /* Byte count len is minimum of buffer size and maxlen */  
    int len = KSIZE < maxlen ? KSIZE : maxlen;  
    memcpy(user_dest, kbuf, len);  
    return len;  
}
```

```
#define MSIZE 528  
  
void getstuff() {  
    char mybuf[MSIZE];  
    copy_from_kernel(mybuf, -MSIZE);  
    . . .  
}
```

**size\_t is unsigned!**

# Outline

- Boolean Algebra
- Data in memory
- Encoding
- **Integer Encodings**
  - Converting Sign
  - **Converting Length**

# Truncation

- May want to convert between numeric types of different sizes
- Going from a larger to a smaller number of bits is easy
  - **Truncation**: drop bits from the most significant side until we fit
    - Values that can be represented by both types are preserved!
      - Including negative values!
    - Values that can't be represented by the smaller type are mapped to some that can (modular (= modulo) behavior)
- Example
  - 16 bits  $\rightarrow$  8 bits: ~~10110010~~ 01001000  $\rightarrow$  01001000
  - Unsigned:  $45640_{10} \rightarrow 72_{10}$ 
    - $72_{10} = 45640_{10} \text{ modulo } 2^8$
  - Signed:  $-52664_{10} \rightarrow 72_{10}$ 
    - $72_{10} = -52664_{10} \text{ modulo } 2^8$



# Extension

- Going from smaller to larger: what to do with the “new” bits?
  - These “new” bits go on the most significant side
- **Unsigned:** easy, pad with 0s!
  - Always ok to add 0s on the most significant end:  $15213_{10} = 00015213_{10}$
  - Example: 8 bits  $\rightarrow$  16 bits:  $01001000 \rightarrow 00000000\ 01001000$ 
    - $72_{10} = 72_{10}$
  - Value is preserved!
- **Signed:** a bit more involved (next slides)

# Sign Extension

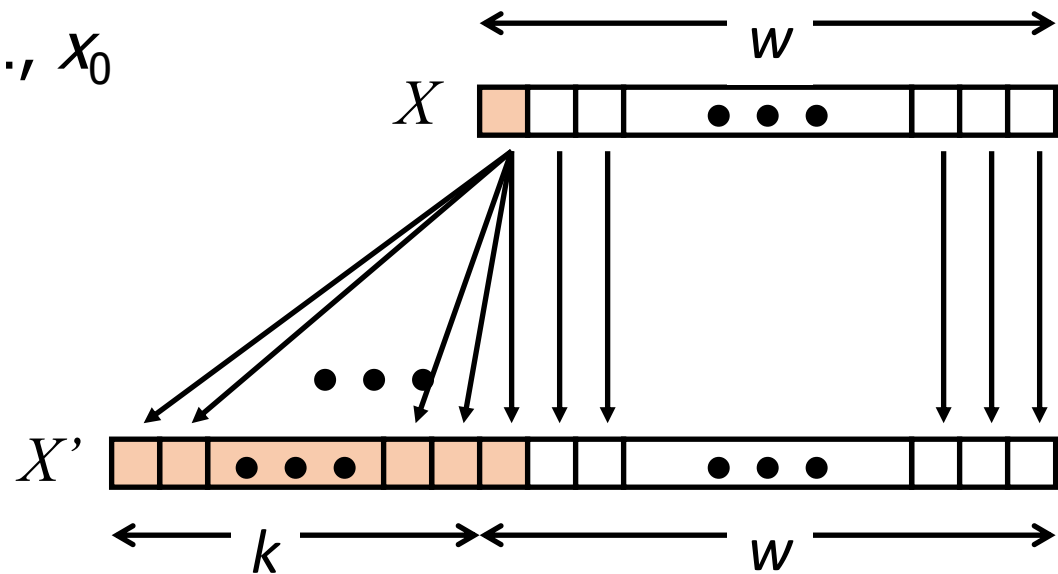
- Task:
  - Given  $w$ -bit **signed** integer  $x$
  - Convert it to  $w+k$ -bit integer ***with same value***

- Rule:

- Make  $k$  copies of sign bit:

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

$k$  copies of MSB  
(MSB = most significant bit)



# Example sign extension

- Extend -128 from an 8-bit to bigger versions
- 8-bit version:
  - $-128 + 0 = 1x(-128) + \text{all zeros} = 0b1000\ 0000$
- 9-bit version:
  - $-256 + 128 = 1x(-256) + 1x128 + \text{all zeros} = 0b1\ 1000\ 0000$
- 10-bit version:
  - $-512 + 256 + 128 = 0b11\ 1000\ 0000$

# Sign Extension Examples

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

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

- Converting from smaller to larger integer data type
- C automatically performs sign extension for signed types
  - If cast changes both sign and size, extends based on **source** signedness
  - But less confusing to write code that makes the types (and casts) explicit

# Outline

- Boolean Algebra
- Data in memory
- Encoding
- Integer Encodings
  - Converting Sign
  - Converting Length