

# Bits, Bytes, and Integers – Part 2

15-213/14-513/15-513: Introduction to Computer Systems  
3<sup>rd</sup> Lecture, September 6, 2022

## Instructors:

Dave Andersen (15-213)

Zack Weinberg (15-213)

Brian Railing (15-513)

David Varodayan (14-513)

# Waitlist questions

- **15-213: Mary Widom (marwidom@cs.cmu.edu)**
- **15-513: Mary Widom (marwidom@cs.cmu.edu)**
- **14-513: INI Enrollment (ini-academic@andrew.cmu.edu)**
  
- **Please don't contact the instructors with waitlist questions.**

# Reminder about labs 0 and 1

- **Lab 0 is due today September 6 at 11:59pm**
  - No late days, no grace days
  - Email instructors if you need an extension
  - It's supposed to be easy—if it takes you more than a couple hours' effort, you may not be prepared for this course
  
- **Lab 1 (data lab) is out**
  - Due Thursday September 15 at 11:59pm ET
  - Start early!
  - Lab 2 (bomb lab) comes out on Thursday September 8

# Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - **Representation: unsigned and signed; negation and addition**
  - Conversion, casting, extension, truncation
  - Multiplication, division, shifting
- Byte order in memory, pointers, strings

Activity: <https://canvas.cmu.edu/courses/30386/assignments/525231>

Use the “People” section of Canvas to assign yourself to a group.

Do models 0 and 1, then stop.

# Encoding “Integers”

## Unsigned

Given a bit  
vector  $x$ ,  
 $w$  bits long...

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

## Signed (twos complement)

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

← Sign Bit

## Examples ( $w = 5$ )

±16	8	4	2	1
0	1	0	1	0

$$0 + 8 + 0 + 2 + 0 = 10$$

16	8	4	2	1
1	0	1	1	0
-16	8	4	2	1

$$16 + 8 + 0 + 2 + 0 = 26$$

$$-16 + 8 + 0 + 2 + 0 = -10$$

# Negation: Complement & Increment

## ■ Negate through complement and increase

$$\sim x + 1 == -x$$

## ■ Why?

- $-x + x == 0$  (by definition)
- $\sim x + x == 1111\dots111 == -1$
- $\sim x + x + 1 == 0$
- $(\sim x + 1) + x == 0$
- $\sim x + 1 == -x$

$$\begin{array}{r}
 x \quad 10011101 \\
 + \quad \sim x \quad 01100010 \\
 \hline
 -1 \quad 11111111
 \end{array}$$

## Example: $x = 15213$

	Decimal	Hex	Binary
$x$	<b>15213</b>	3B 6D	00111011 01101101
$\sim x$	<b>-15214</b>	C4 92	11000100 10010010
$\sim x + 1$	<b>-15213</b>	C4 93	11000100 10010011
$y$	<b>-15213</b>	C4 93	11000100 10010011


# Complement & Increment Examples

$$x = 0$$

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
$\sim 0$	-1	FF FF	11111111 11111111
$\sim 0 + 1$	0	00 00	00000000 00000000

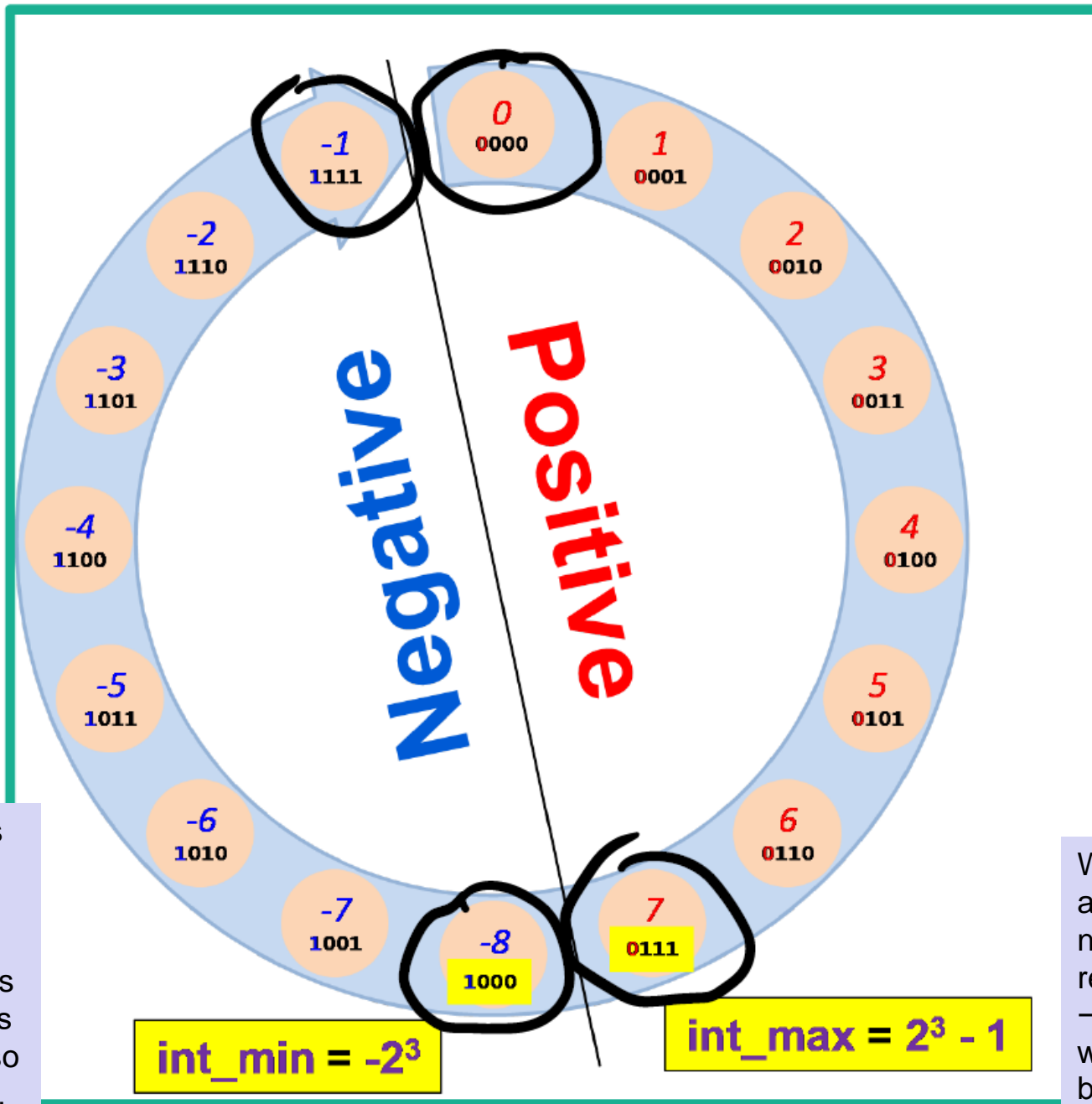
$$x = T_{\min}$$

	Decimal	Hex	Binary
$x$	-32768	80 00	10000000 00000000
$\sim x$	32767	7F FF	01111111 11111111
$\sim x + 1$	-32768	80 00	10000000 00000000



Oops!  
It's still  
negative!





Eight negative values:  
 $-1, -2, \dots, -8$

Eight *non*-negative values:  
 $0, 1, \dots, 7$

Mathematicians would prefer it if a 4-bit signed number could represent values  $-8 \dots 8$ , but that's  $2^4 + 1$  values, so they won't all fit.

What if we made a 4-bit signed number only represent values  $-7 \dots 7$ ? Then we wouldn't be using bit pattern 1000...

# Unsigned Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits



## ■ Standard Addition Function

- Ignores carry output

## ■ Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

```

unsigned char      1110 1001      E9      233
                   + 1101 0101      + D5      + 213
                   -----
                   -----
  
```

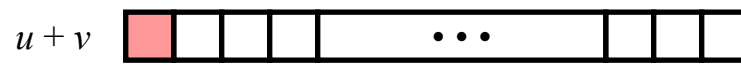
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# Unsigned Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits



## Standard Addition Function

- Ignores carry output

## Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

unsigned char

1110 1001	<b>E9</b>	233
+ 1101 0101	+ D5	+ 213
<b>1 1011 1110</b>	<b>1BE</b>	<b>446</b>
<b>1011 1110</b>	<b>BE</b>	<b>190</b>

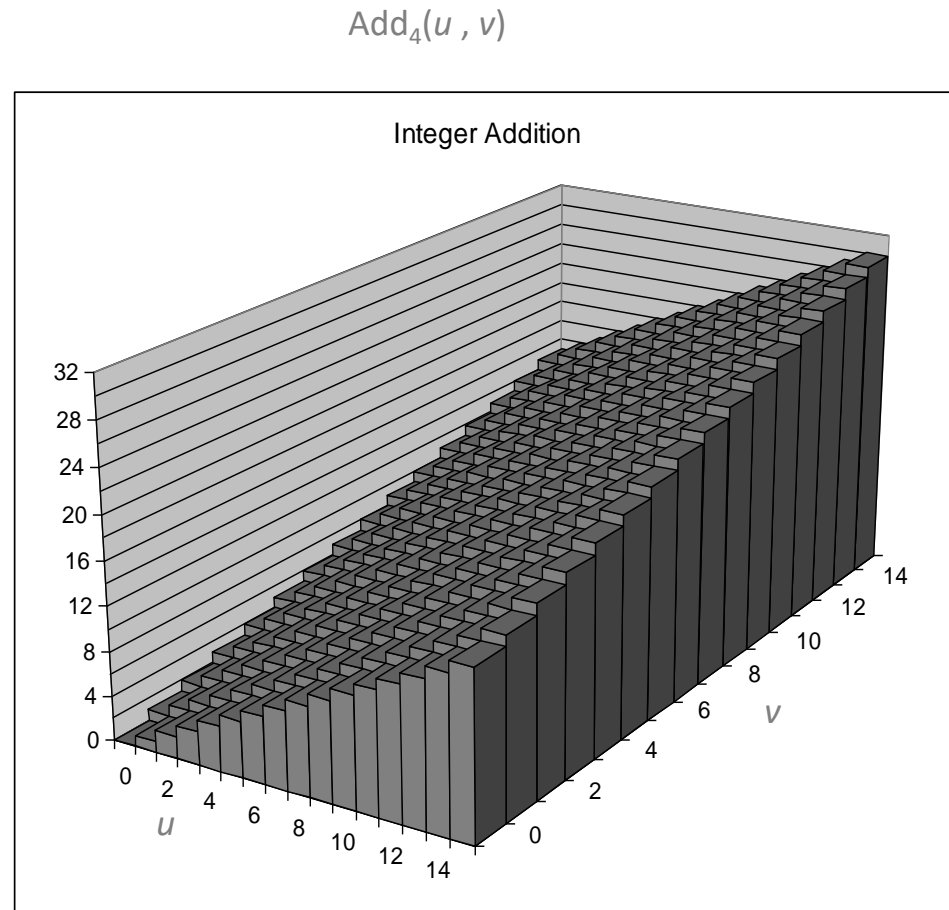
Hex  
Decimal  
Binary

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

# Visualizing (Mathematical) Integer Addition

## Integer Addition

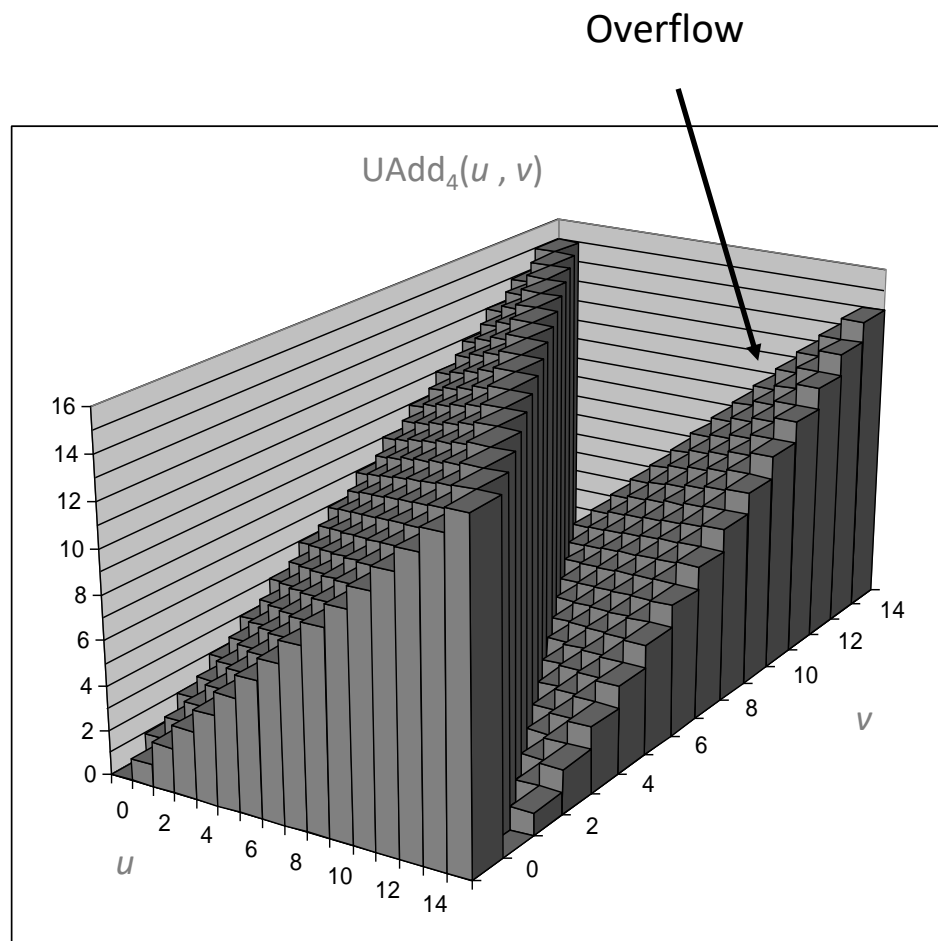
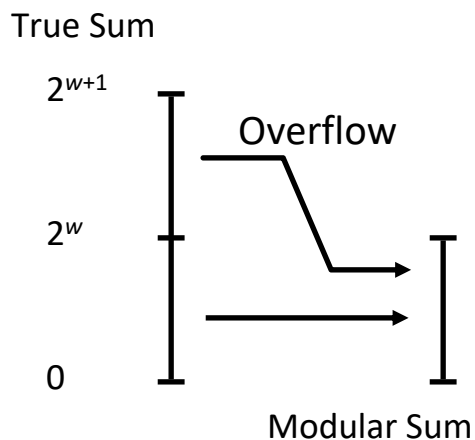
- 4-bit integers  $u, v$
- Compute true sum  $\text{Add}_4(u, v)$
- Values increase linearly with  $u$  and  $v$
- Forms planar surface



# Visualizing Unsigned Addition

## ■ Wraps Around

- If true sum  $\geq 2^w$
- At most once

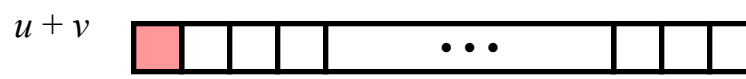


# Two's Complement Addition

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`

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
1 1011 1110	1BE	-66
1011 1110	BE	-66

# Visualizing 2's Complement Addition

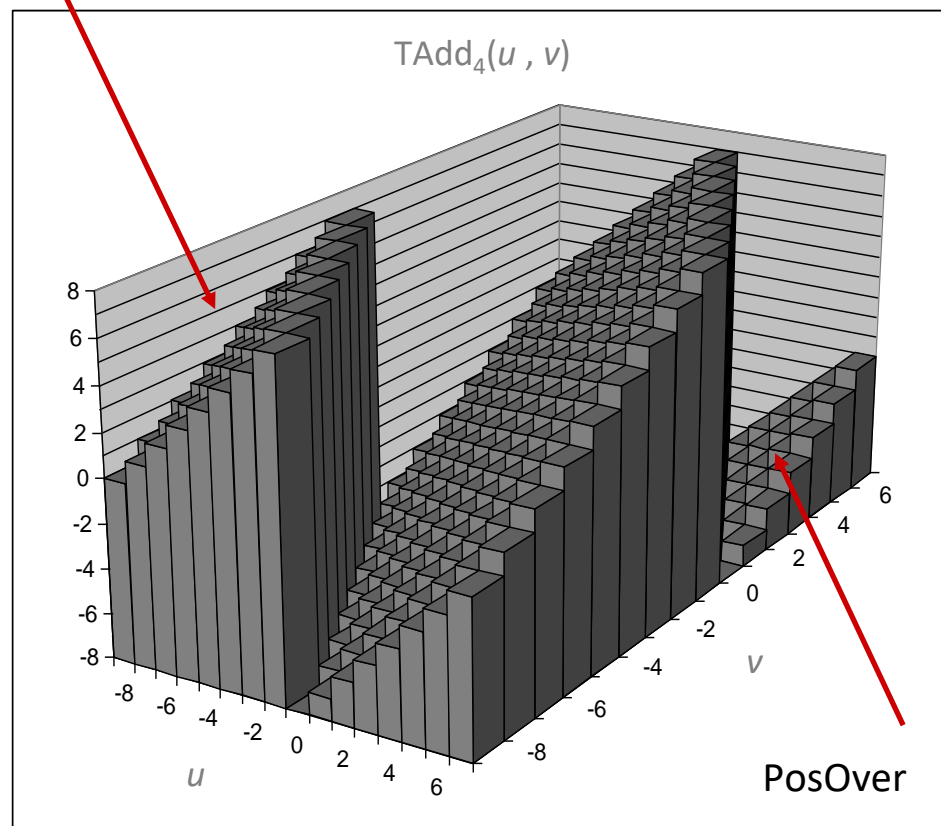
## ■ Values

- 4-bit two's comp.
- Range from -8 to +7

## ■ Wraps Around

- If  $\text{sum} \geq 2^{w-1}$ 
  - Becomes negative
  - At most once
- If  $\text{sum} < -2^{w-1}$ 
  - Becomes positive
  - At most once

NegOver

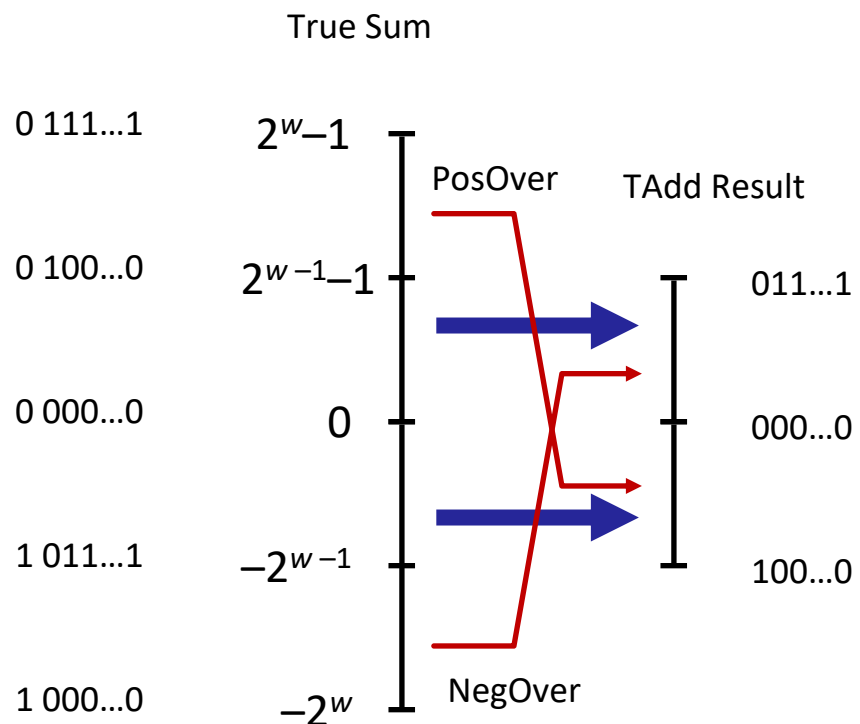


PosOver

# TAdd Overflow

## ■ Functionality

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





# Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - Representation: unsigned and signed; negation and addition
  - Conversion, casting, extension, truncation
  - Multiplication, division, shifting
- Byte order 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

Activity: <https://canvas.cmu.edu/courses/30386/assignments/525231>

Do model 2, then stop.

# General Boolean Algebras

## ■ Operate on Bit Vectors

- Operations applied bitwise

01101001	01101001	01101001	01101001
& 01010101	01010101	^ 01010101	~ 01010101
<u>01010101</u>	<u>01010101</u>	<u>01010101</u>	<u>01010101</u>
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 $\&$ , $|$ , $\sim$ , $\wedge$ 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)

- $\sim 0x41 \rightarrow$
- $\sim 0x00 \rightarrow$
- $0x69 \& 0x55 \rightarrow$
- $0x69 | 0x55 \rightarrow$

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 is “True”
  - Always returns 0 or 1
  - Early exit

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

## ■ Example

- `!0x41` → 0
- `!0x00` → 1
- `!!0x41` → 1
- `0x69 && 0x55` → 0
- `0x69 || 0x55` → 1
- `p && *p` (avoids null pointer access)



Activity: <https://canvas.cmu.edu/courses/30386/assignments/525231>

Do model 3, then stop.

# Logical versus Bitwise

<b>X</b>	<b>!X</b>	<b>!!X</b>	<b>!!X == X</b>
-1	0	1	No
0	1	0	Yes
1	0	1	Yes
2	0	1	No

<b>X</b>	<b>~X</b>	<b>~~X</b>	<b>~~X == X</b>
-1	0	-1	Yes
0	-1	0	Yes
1	-2	1	Yes
2	-3	2	Yes

**!!x != x**

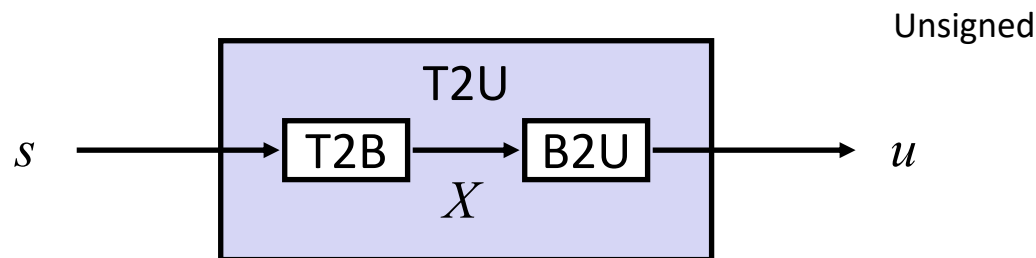
**~~x == x**

# Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed; negation and addition
  - **Conversion, casting, extension, truncation**
  - Multiplication, division, shifting
- Byte order in memory, pointers, strings

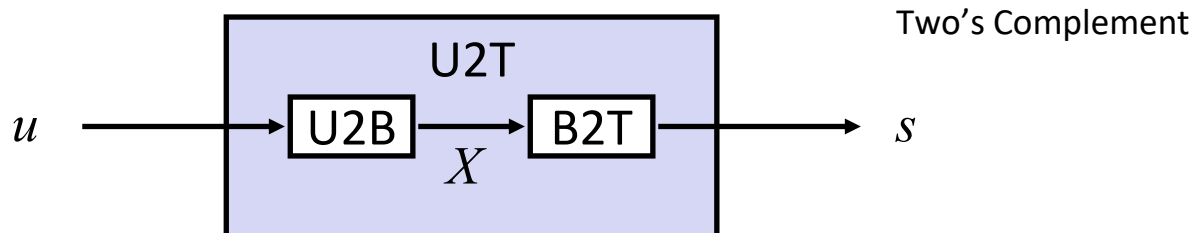
# Mapping Between Signed & Unsigned

Two's Complement



Maintain Same Bit Pattern

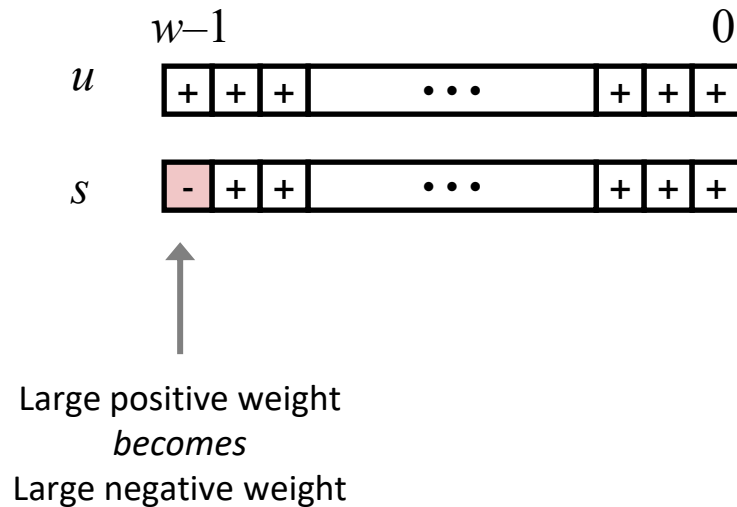
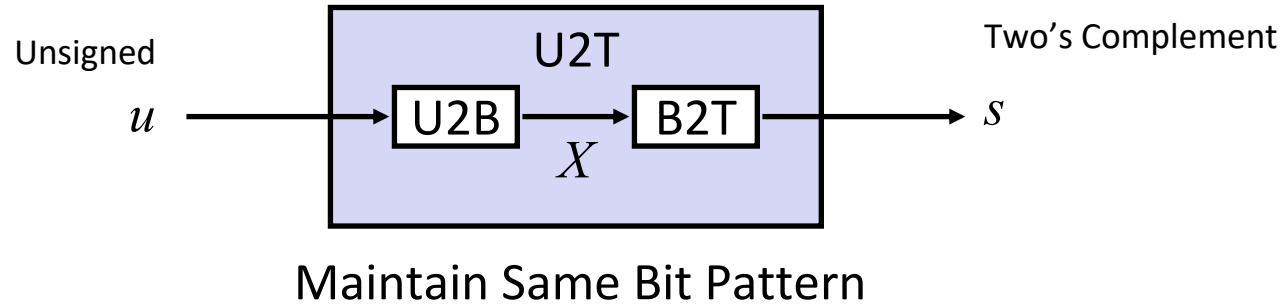
Unsigned



Maintain Same Bit Pattern

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

# Relation between Signed & Unsigned



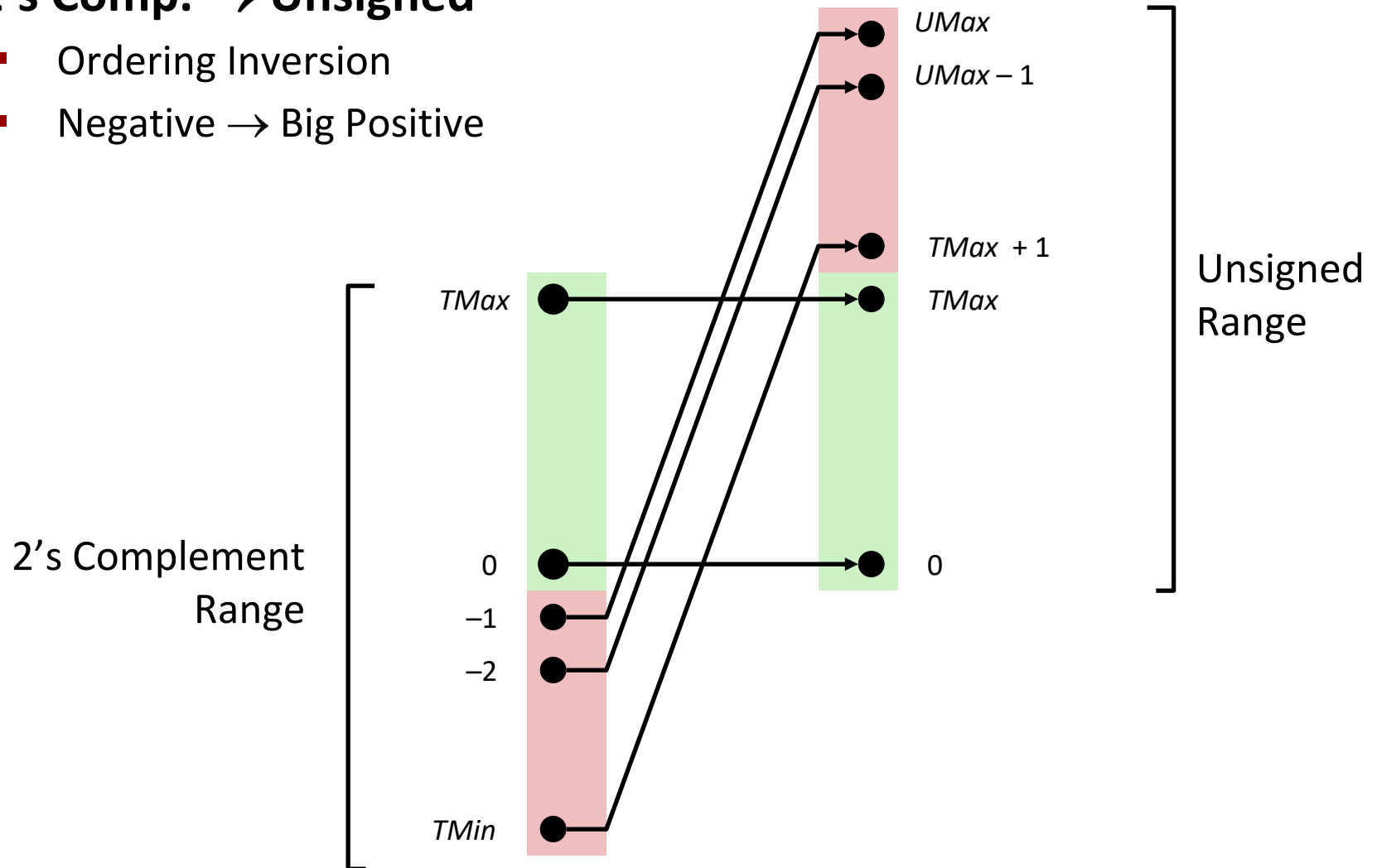
# Mapping Signed $\leftrightarrow$ Unsigned

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

# 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:

Constant 1	Constant 2	Relation	Evaluation
0	0U	==	Unsigned
-1	0	<	Signed
<b>-1</b>	<b>0U</b>	<b>&gt;</b>	<b>Unsigned</b>
INT_MAX	INT_MIN	>	Signed
<b>(unsigned) INT_MAX</b>	<b>INT_MIN</b>	<b>&lt;</b>	<b>Unsigned</b>
-1	-2	>	Signed
<b>(unsigned) -1</b>	-2	<b>&gt;</b>	<b>Unsigned</b>
INT_MAX	((unsigned) INT_MAX) + 1	<	Unsigned
<b>INT_MAX</b>	<b>(int) ((unsigned) INT_MAX) + 1</b>	<b>&gt;</b>	<b>Signed</b>

# Summary

## Casting Signed $\leftrightarrow$ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting  $2^w$

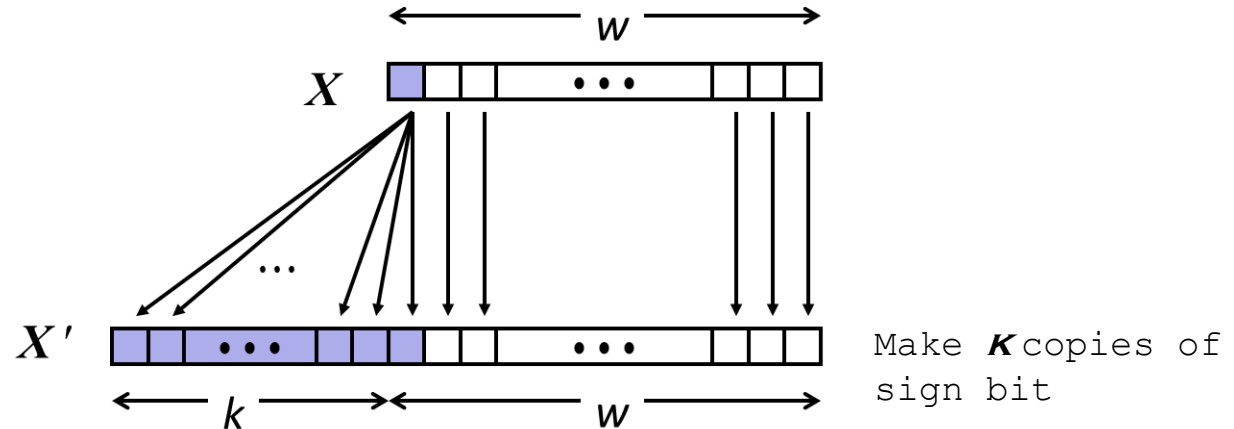
- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

# Today: Bits, Bytes, and Integers

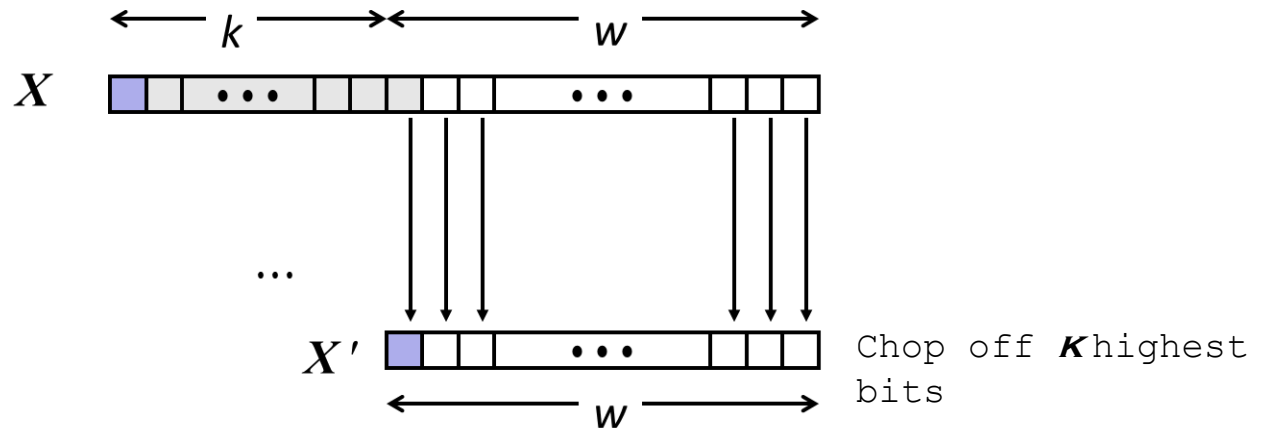
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed; negation and addition
  - Conversion, casting, extension, truncation
  - **Multiplication, division, shifting**
- Byte order in memory, pointers, strings

# Sign Extension and Truncation

## ■ Sign Extension

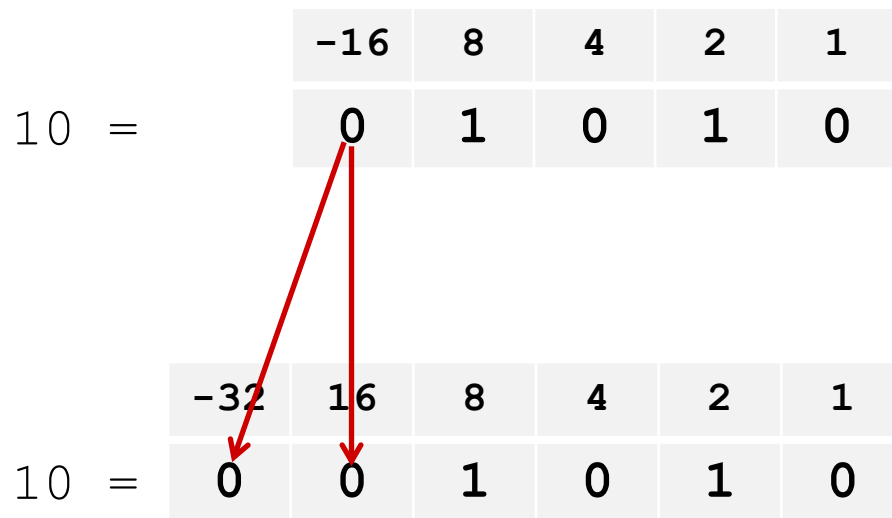


## ■ Truncation

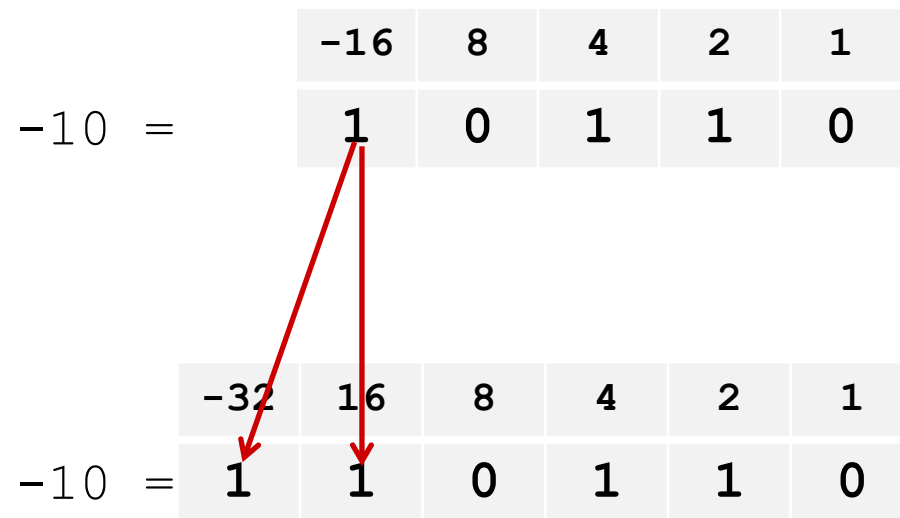


# Sign Extension: Simple Example

Positive number



Negative number



# Truncation: Simple Example

No sign change

$$2 = \begin{array}{|c|c|c|c|c|} \hline -16 & 8 & 4 & 2 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

$$2 = \begin{array}{|c|c|c|c|} \hline -8 & 4 & 2 & 1 \\ \hline 0 & 0 & 1 & 0 \\ \hline \end{array}$$

$$2 \bmod 16 = 2$$

$$-6 = \begin{array}{|c|c|c|c|c|} \hline -16 & 8 & 4 & 2 & 1 \\ \hline 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

$$-6 = \begin{array}{|c|c|c|c|} \hline -8 & 4 & 2 & 1 \\ \hline 1 & 0 & 1 & 0 \\ \hline \end{array}$$

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

Sign change

$$10 = \begin{array}{|c|c|c|c|c|} \hline -16 & 8 & 4 & 2 & 1 \\ \hline 0 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

$$-6 = \begin{array}{|c|c|c|c|} \hline -8 & 4 & 2 & 1 \\ \hline 1 & 0 & 1 & 0 \\ \hline \end{array}$$

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

$$-10 = \begin{array}{|c|c|c|c|c|} \hline -16 & 8 & 4 & 2 & 1 \\ \hline 1 & 0 & 1 & 1 & 0 \\ \hline \end{array}$$

$$6 = \begin{array}{|c|c|c|c|} \hline -8 & 4 & 2 & 1 \\ \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$$

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

# Today: Bits, Bytes, and Integers

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

# Shifting

## Left Shift: $x \ll y$

- Shift bit-vector  $x$  left  $y$  positions
- Throw away extra bits on left
- Fill with 0's on right
- Equivalent to multiplying by  $2^y$

## Right Shift: $x \gg y$

- Shift bit-vector  $x$  right  $y$  positions
- Throw away extra bits on right
- Two kinds:
  - "Logical": Fill with 0's on left
  - "Arithmetic": Replicate most significant bit on left
- Almost* equivalent to dividing by  $2^y$

## Undefined Behavior (in C)

- Shift amount  $< 0$  or  $\geq$  word size

Argument $x$	01100010
$\ll 3$	00010000
Logical $\gg 2$	00011000
Arithmetic $\gg 2$	00011000

Argument $x$	10100010
$\ll 3$	00010000
Logical $\gg 2$	00101000
Arithmetic $\gg 2$	11101000



# Multiplication

## ■ Goal: Computing 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:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
- Two's complement min (negative): Up to  $2w-1$  bits
  - Result range:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
- Two's complement max (positive): Up to  $2w$  bits, but only for  $(TMin_w)^2$ 
  - Result range:  $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$

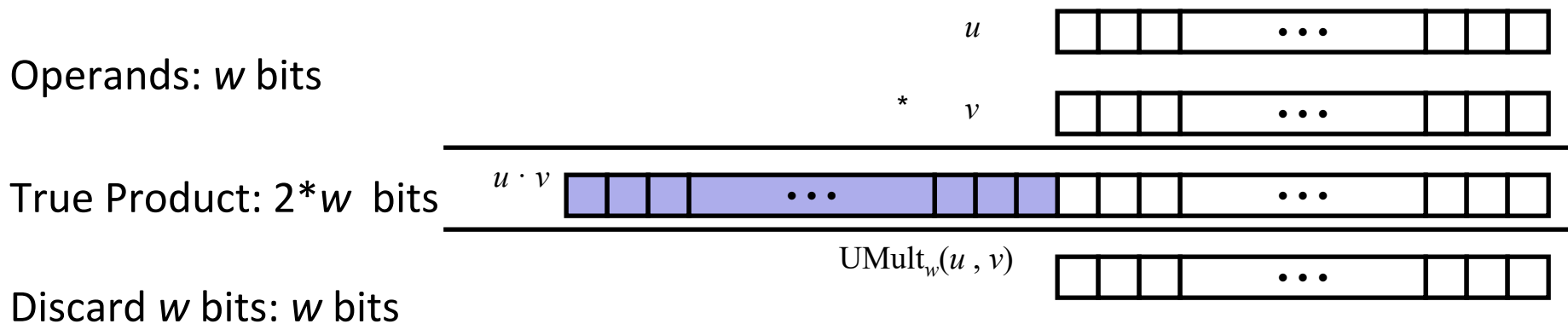
## ■ So, maintaining exact results...

- would need to keep expanding word size with each product computed
- is done in software, if needed
  - e.g., by “arbitrary precision” arithmetic packages

Activity: <https://canvas.cmu.edu/courses/30386/assignments/525231>

Do model 4, then stop.

# Unsigned Multiplication in C



## ■ Standard Multiplication Function

- Ignores high order  $w$  bits

## ■ Implements Modular Arithmetic

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

$$\begin{array}{r}
 1110\ 1001 \\
 * 1101\ 0101 \\
 \hline
 1100\ 0001\ 1101\ 1101 \\
 \hline
 1101\ 1101
 \end{array}$$

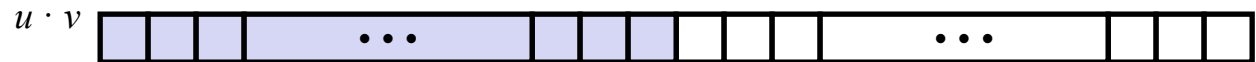
$$\begin{array}{r}
 \text{E9} \quad 233 \\
 * \text{D5} \quad * 213 \\
 \hline
 \text{C1DD} \quad 49629 \\
 \hline
 \text{DD} \quad 221
 \end{array}$$

# 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

	1110 1001	E9	-23
*	1101 0101	* D5	* -43
	0000 0011 1101 1101	03DD	989
	1101 1101	DD	-35

# Power-of-2 Multiply with Shift

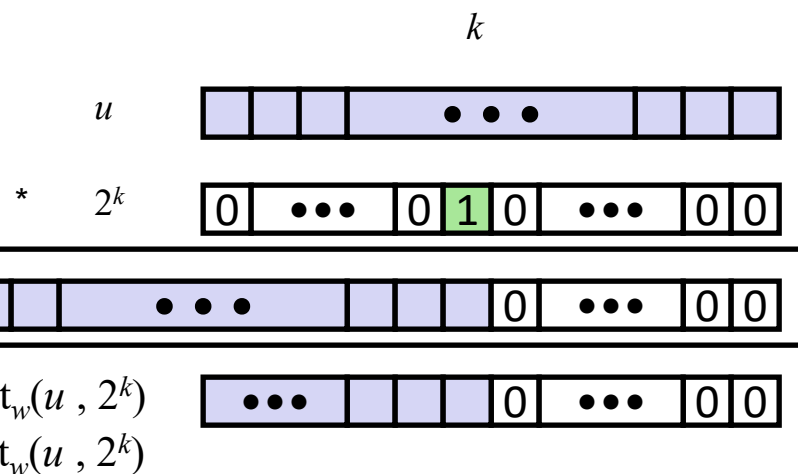
## Operation

- $u \ll k$  gives  $u * 2^k$
- Both signed and unsigned

Operands:  $w$  bits

True Product:  $w+k$  bits

Discard  $k$  bits:  $w$  bits



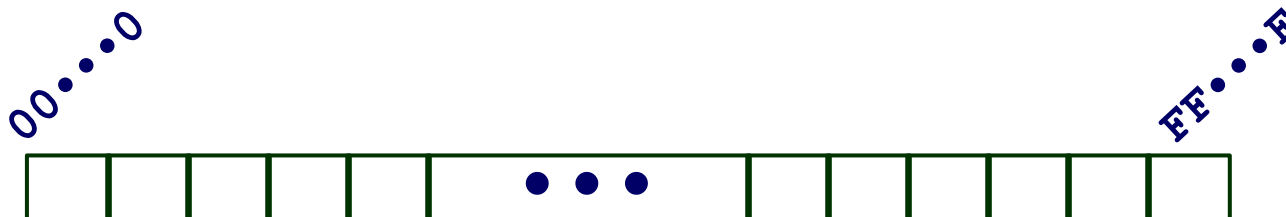
## Examples

- $u \ll 3 \quad == \quad u * 8$
- $(u \ll 5) - (u \ll 3) \quad == \quad u * 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

# Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed; negation and addition
  - Conversion, casting, extension, truncation
  - Multiplication, division, shifting
- **Byte order in memory, pointers, strings**

# Byte-Oriented Memory Organization



## ■ Programs refer to data by address

- Imagine all of RAM as an enormous array of bytes
- An address is an index into that array
  - A pointer variable stores an address

## ■ System provides a private *address space* to each “process”

- A process is an instance of a program, being executed
- An address space is one of those enormous arrays of bytes
- Each program can see only its own code and data within its enormous array
- We’ll come back to this later (“virtual memory” classes)

# Machine Words

## ■ Any given computer has a “Word Size”

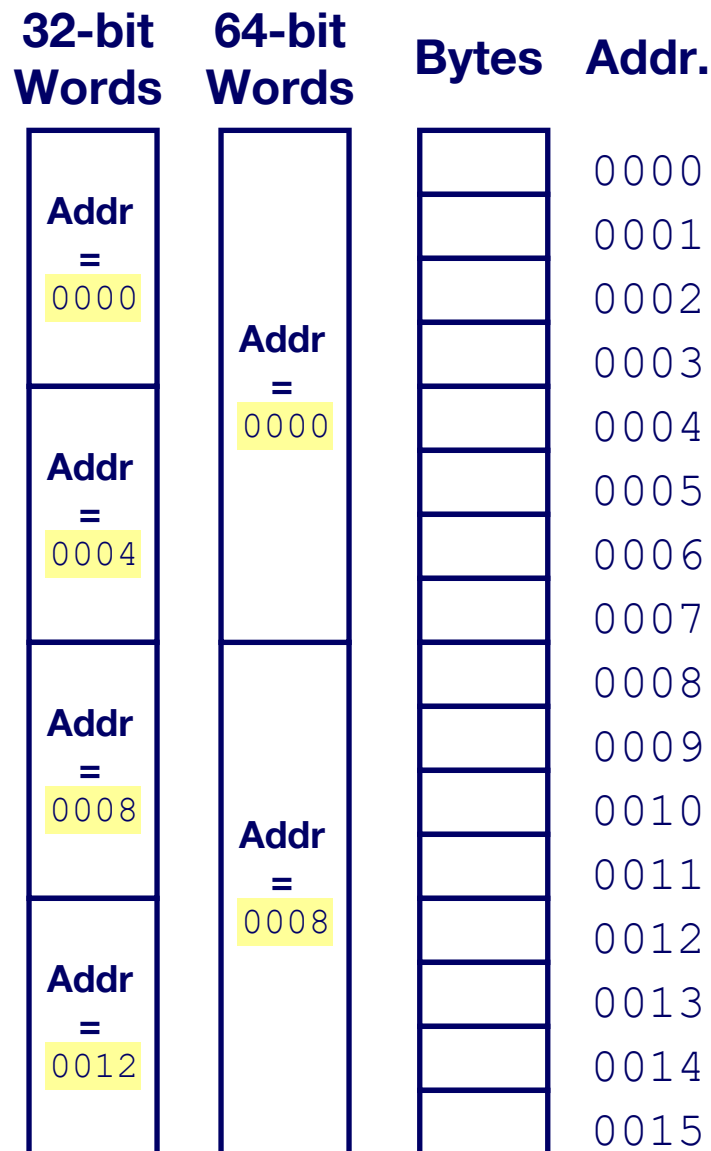
- Nominal size of integer-valued data
  - and of addresses
- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB ( $2^{32}$  bytes)
- Increasingly, machines have 64-bit word size
  - Potentially, could have 16 EB (exabytes) of addressable memory
  - That's  $18.4 \times 10^{18}$  bytes
- Machines still support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

Yes, both of these numbers are correct. This discrepancy is known as the Great Storage Industry Marketing Lie. Ask me about it after class if you really want to know.



# Addresses *Always* Specify Byte Locations

- Address of a word is address of the first byte in the word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



# 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

# Byte Ordering

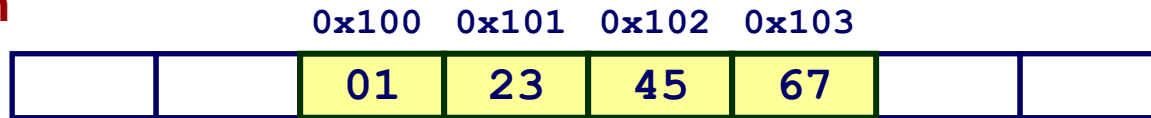
- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, *network packet headers*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

# Byte Ordering Example

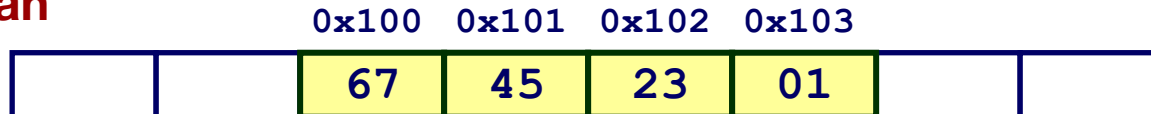
## ■ Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

### Big Endian



### Little Endian



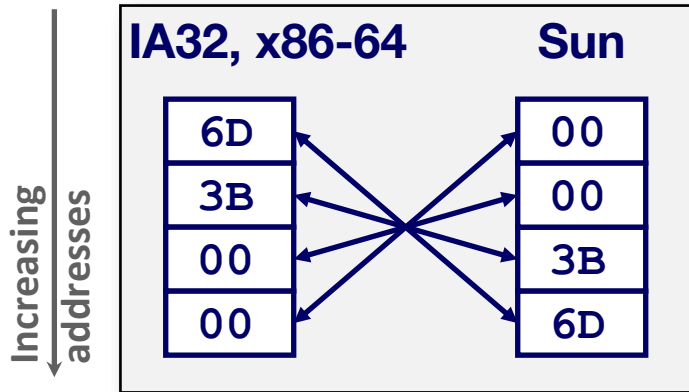
# Representing Integers

Decimal: 15213

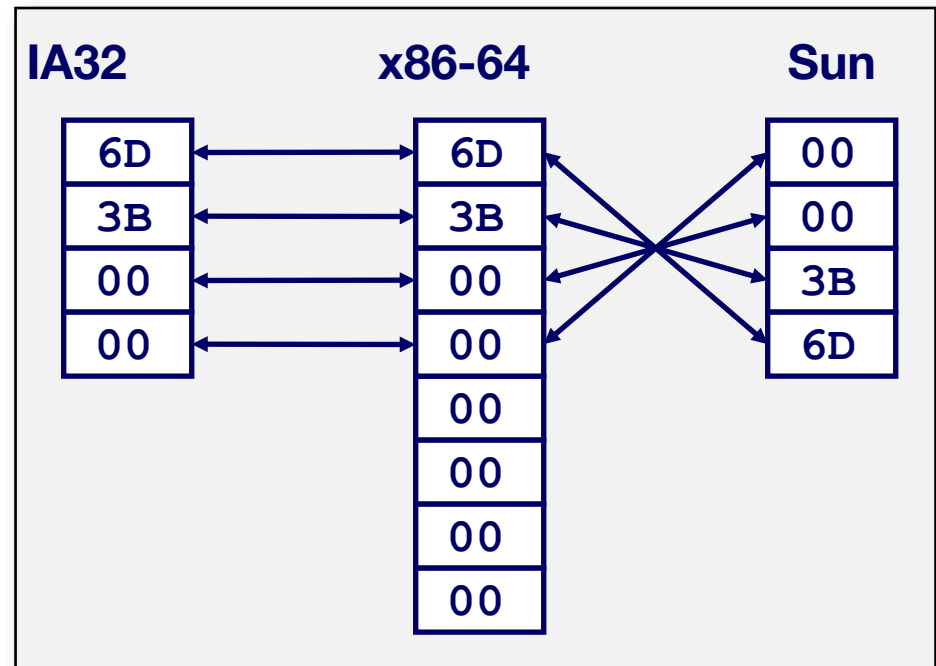
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

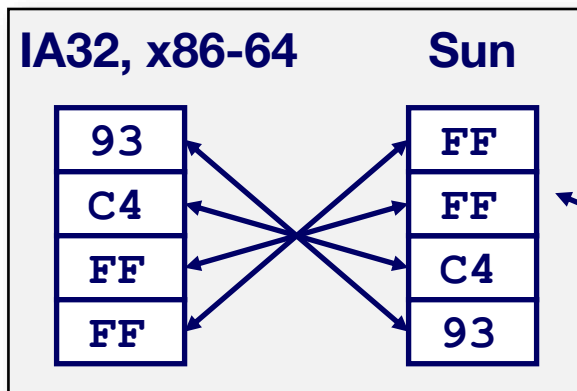
`int A = 15213;`



`long int C = 15213;`



`int B = -15213;`



Two's complement representation

# Examining Data Representations

## ■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char \* allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

### Printf directives:

%p:     Print pointer  
%x:     Print Hexadecimal

# show\_bytes Execution Example

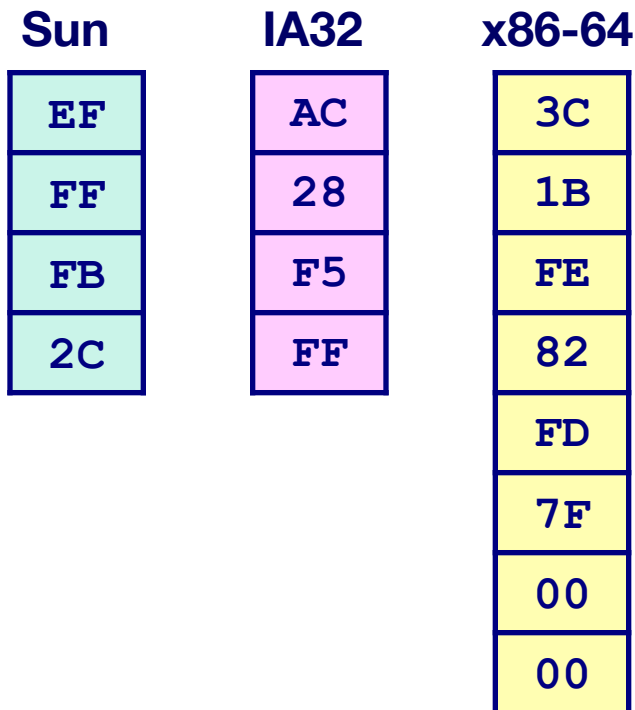
```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

## Result (Linux x86-64):

```
int a = 15213;
0x7ffffb7f71dbc    6d
0x7ffffb7f71dbd    3b
0x7ffffb7f71dbe    00
0x7ffffb7f71dbf    00
```

# Representing Pointers

```
int B = -15213;  
int *P = &B;
```



**Different compilers & machines assign different locations to objects**

**Even get different results each time run program**



# Representing Strings

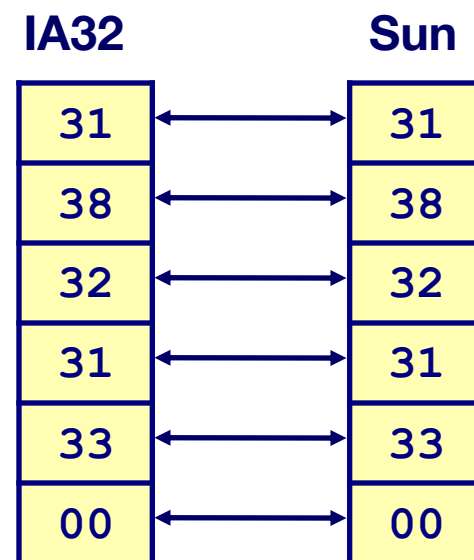
```
char S[6] = "18213";
```

## ■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character "0" has code 0x30
    - Digit  $i$  has code  $0x30+i$
- String should be null-terminated
  - Final character = 0

## ■ Compatibility

- Byte ordering not an issue



# Representing x86 machine code

## ■ x86 machine code is a sequence of *bytes*

- Grouped into variable-length instructions, which look like strings...
- But they contain embedded little-endian numbers...

## ■ Example Fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab, %ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0, 0x28(%ebx)

## ■ Deciphering Numbers

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00