

Bits, Bytes and Integers

Computer Systems

Friday, September 27 2024

Homework #2

■ python challenge

1. Refer to the hint and create a python script to decrypt the following ciphertext.

```
g fmnc wms bgblr rpylqjyrc gr zw fylb. rfyrq ufyr amknsrcpq ypc dmp. bmgle  
gr gl zw fylb gq glcddgagclr ylb rfyr'q ufw rfgq rcvr gq qm jmle. sqgle  
qrple.kyicrpylq() gq pcamkkclbcb. lmu ynnjw ml rfc spj.
```

2. Hint :

K -> M
O -> Q
E -> G

Review - Linux

■ Vim

```
:%s/hello/hello4/g
```

```
/99999  
5s  
31337<ESC>  
:9  
yyp  
:w hello4.c  
:make  
:q
```

Review - Python Basic

■ **fstring**

```
Alice = {'name':'Alice', 'age':45, 'id':'cs201'}  
Bob   = {'name':'Bob', 'age':49, 'id':'cs101'}  
Carol  = {'name':'Carol', 'age':19, 'id':'cs401'}  
Dave   = {'name':'Dave', 'age':16 , 'id':'cs301'}  
Ethan  = {'name':'Ethan', 'age':10 , 'id':'c302'}  
  
family = {'Alice':Alice, 'Ethan':Ethan, 'Bob':Bob, 'Dave':Dave, 'Carol':Carol}  
  
for name in family :  
    print (f" {name}'s age is {family[name]['age']}")
```

Review - Python Tools

■ pwntools

```
>>> p32(int('65766144', 16)).decode()
```

```
>>> 0b10000
```

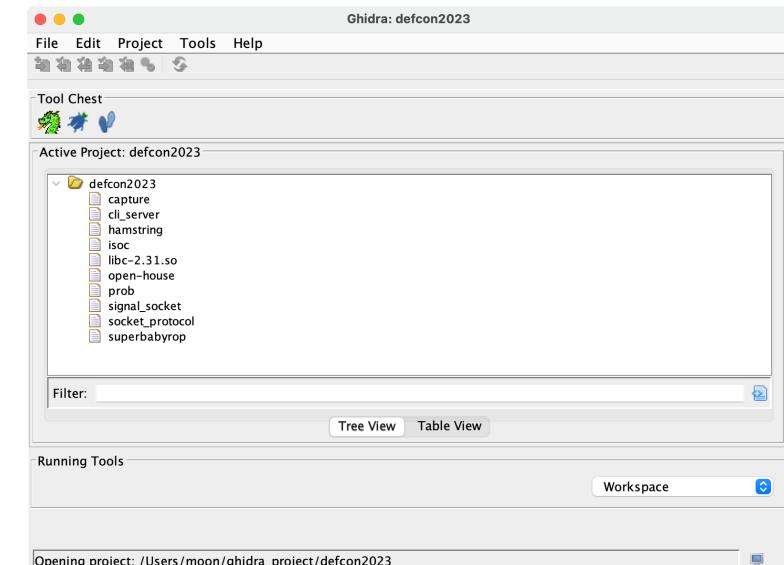
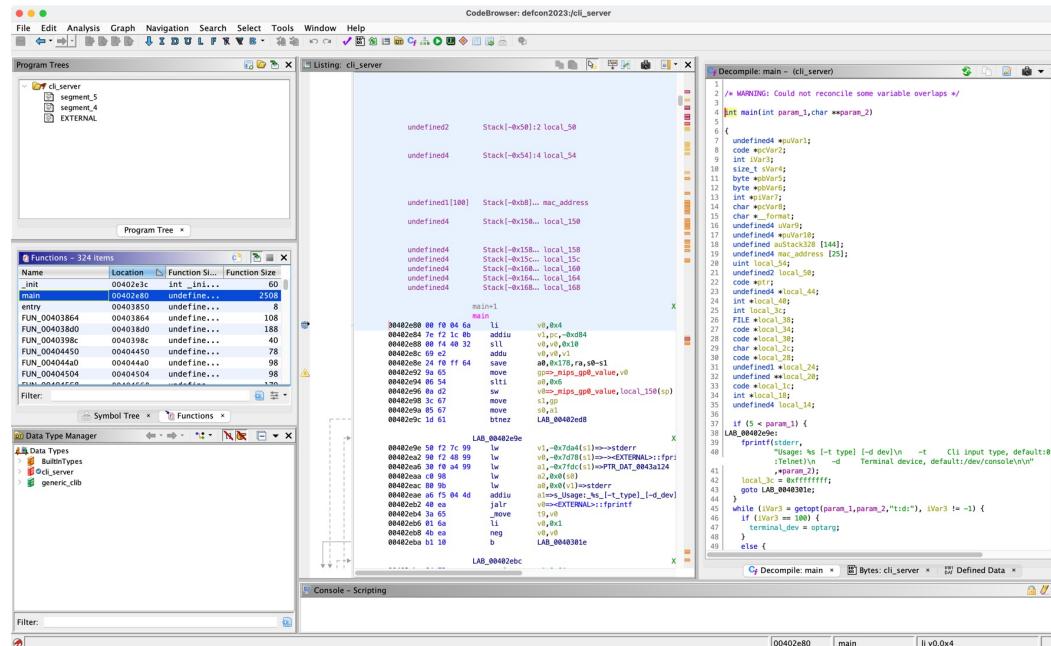
Review - Reversing

Please prepare ghidra for today class!

■ Installation

- <https://github.com/NationalSecurityAgency/ghidra/releases>

■ Decompile executable file



Reversing

And we can use these tools

`objdump -d [executable file]`

<https://godbolt.org/>

The screenshot shows the Godbolt C++ compiler tool interface. On the left, the C++ source code is displayed:

```
--INSERT--  
1 // Type your code here, or load an example.  
2 #include <stdio.h>  
3 int square(int num) {  
4     printf("num=%d\n", num);  
5     return num * num;  
6 }  
7  
8 int caller(int num) {  
9     int result = square(3);  
10    printf("result = %d\n", result);  
11    return 0;  
12 }  
13
```

On the right, the generated x86-64 assembly code is shown:

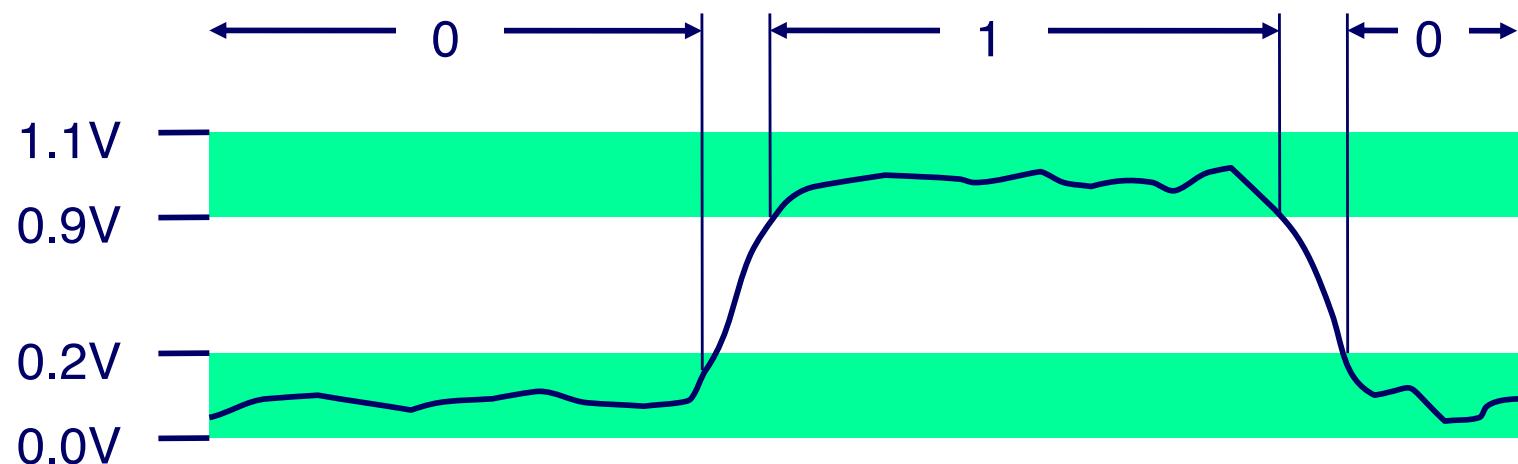
```
x86-64 gcc 14.2 (Editor #1)  
11     mov    eax, 0  
12     call   printf  
13     mov    eax, DWORD PTR [rbp-4]  
14     imul  eax, eax  
15     leave  
16     ret  
.LC1:  
18     .string "result = %d\n"  
caller(int):  
20     push   rbp  
21     mov    rbp, rsp  
22     sub    rsp, 32  
23     mov    DWORD PTR [rbp-20], edi  
24     mov    edi, 3  
25     call   square(int)  
26     mov    DWORD PTR [rbp-4], eax  
27     mov    eax, DWORD PTR [rbp-4]  
28     mov    esi, eax  
29     mov    edi, OFFSET FLAT:.LC1  
30     mov    eax, 0  
31     call   printf  
32     mov    eax, 0  
33     leave  
34     ret
```

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



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
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

Example Data Representations

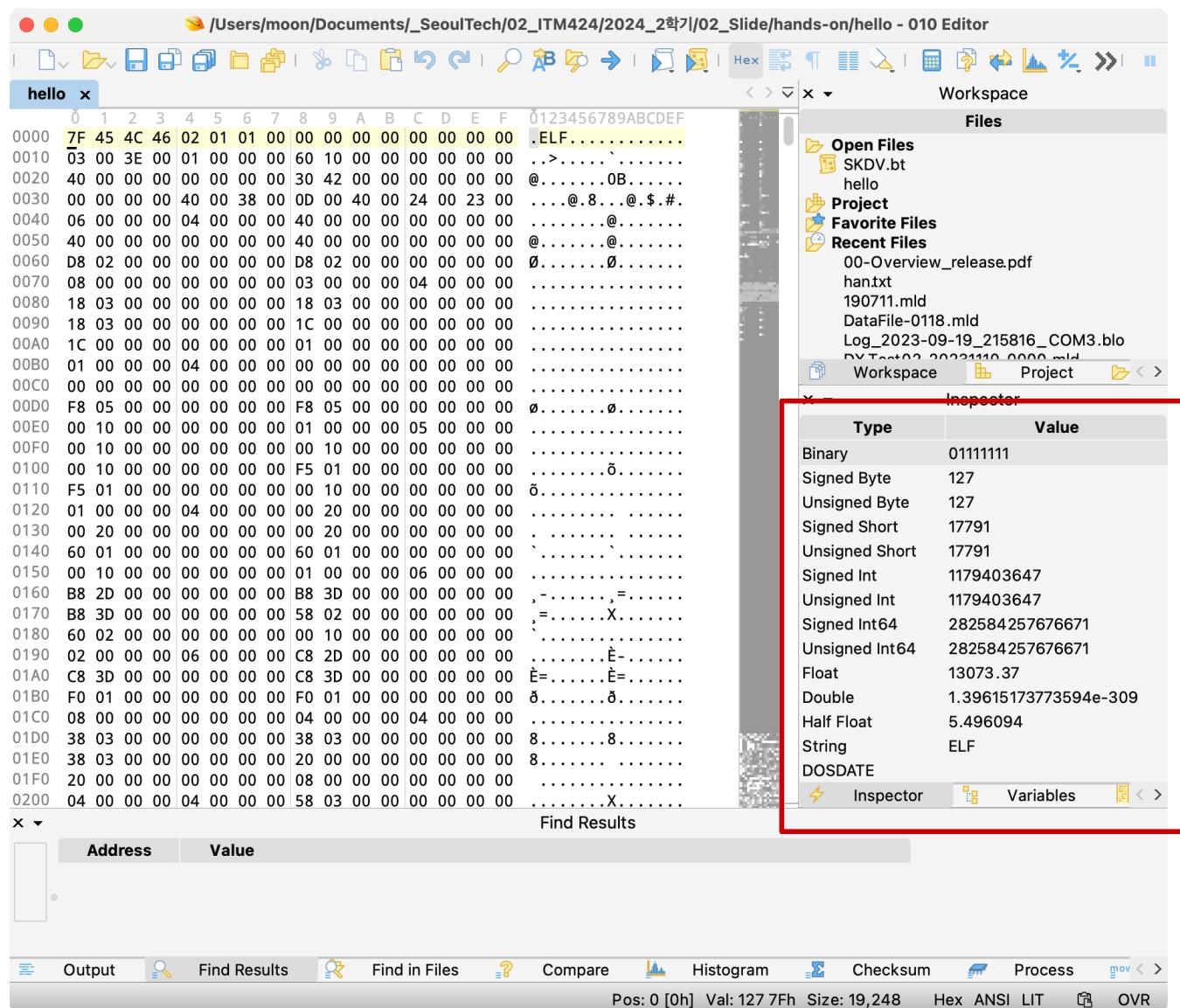
C Data Type	Typical 32-bit	Typical 64-bit
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

“ILP32” “LP64”

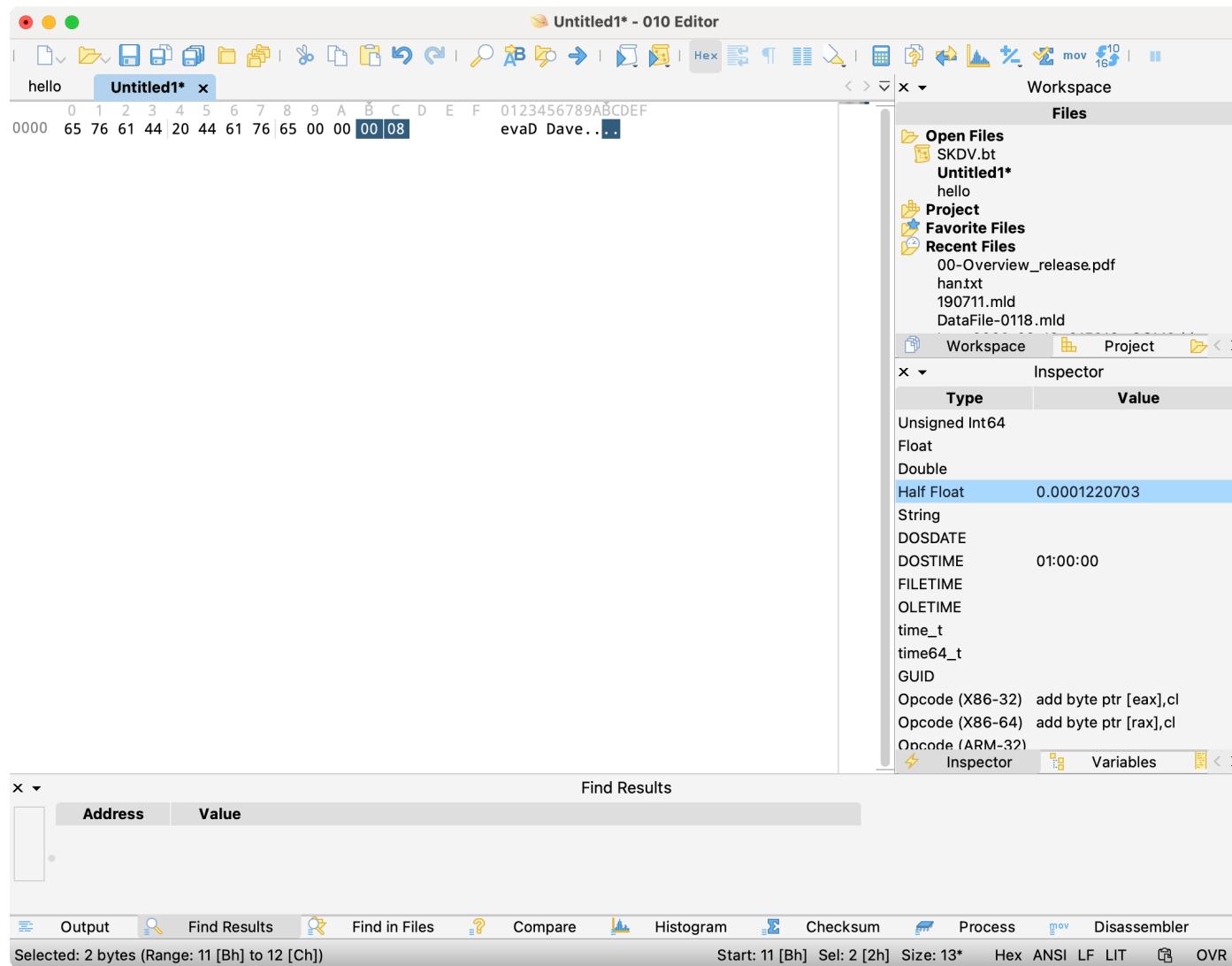
Hex editor

- 010 editor (multi platform)
- iHex (MacOS)
- HxD (Windows)
- xxd (Linux)
- Vim (%!xxd)

Hex editor



Hex editor



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 $B=1$ **or both**

	0	1
0	0	1
1	1	1

Not

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

\sim	0	1
0	1	0
1	0	1

Exclusive-Or (Xor)

$A ^ B = 1$ when $A=1$ or $B=1$, **but not both**

$^$	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

$$\begin{array}{rcl} 01101001 & 01101001 & 01101001 \\ \& 01010101 & | 01010101 & ^ 01010101 & \sim 01010101 \\ \hline 01000001 & 01111101 & 00111100 & 10101010 \end{array}$$

- All of the Properties of Boolean Algebra Apply

Example: Sets of Small Integers

■ Width w bit vector represents subsets of $\{0, 1, \dots, w - 1\}$

- Let a be a bit vector representing set A , then bit $a_j = 1$ if $j \in A$
- Examples:

- 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

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 &, |, ~, ^ 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

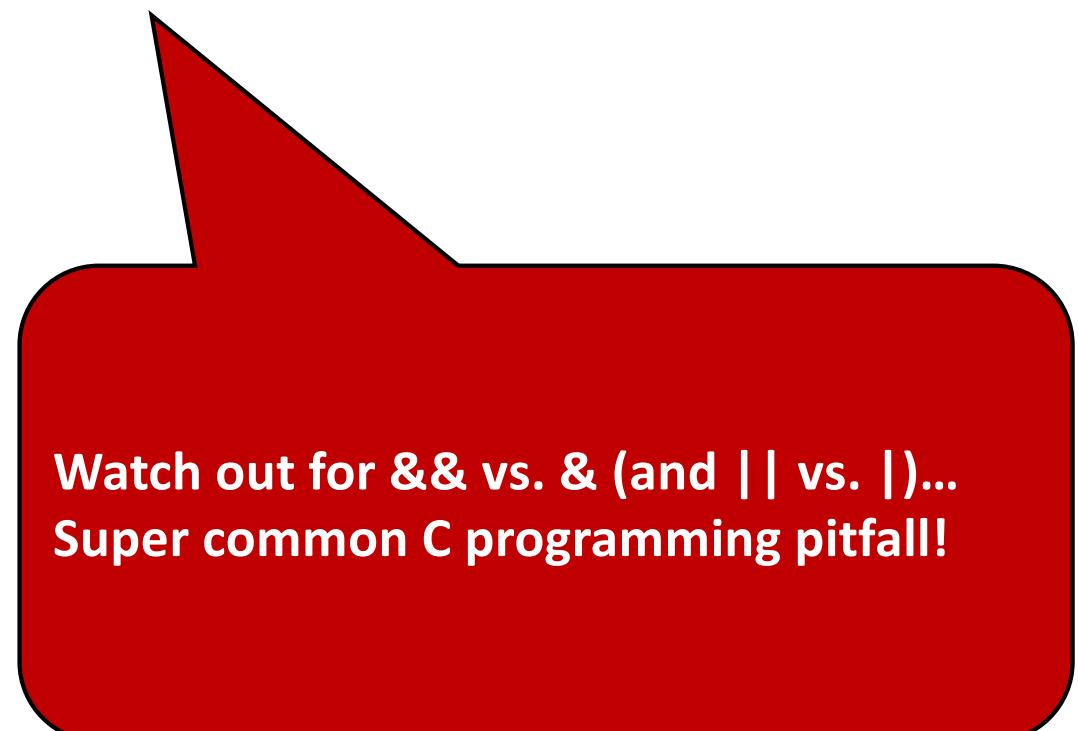
Contrast: Logic Operations in C

■ 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. `|`)...
Super common C programming pitfall!

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

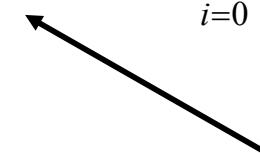
Unsigned

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

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

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

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

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
- $UMax = 2 * TMax + 1$
- Question: $\text{abs}(TMin)?$

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

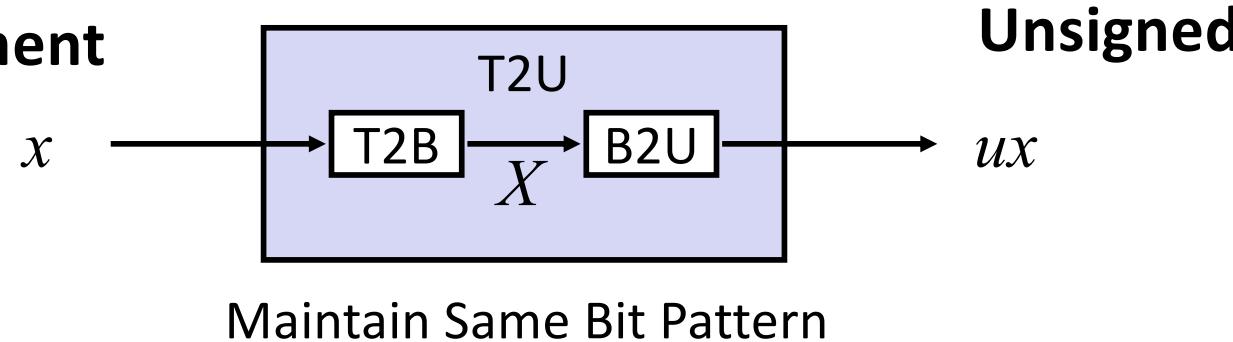
- $U2B(x) = B2U^{-1}(x)$
 - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$
 - Bit pattern for two's comp integer

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

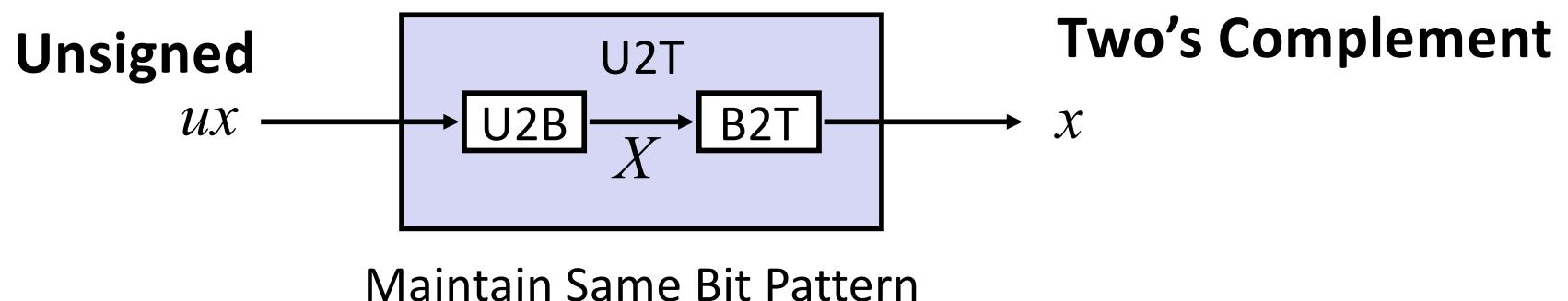
Mapping Between Signed & Unsigned

Two's Complement



Unsigned

Unsigned

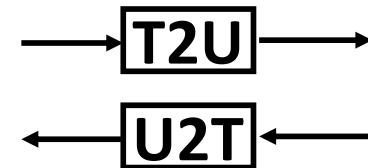


Two's Complement

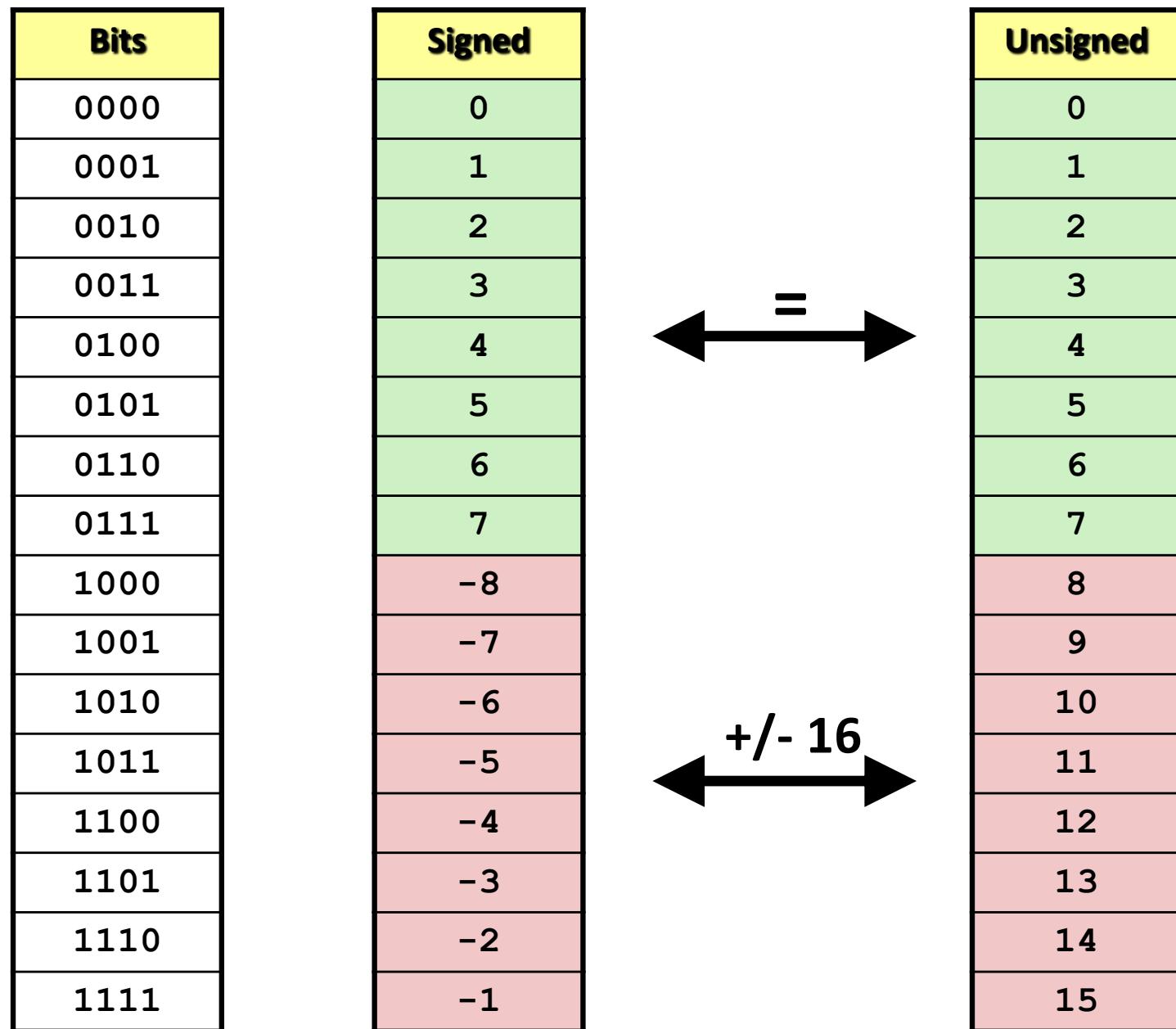
- Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

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

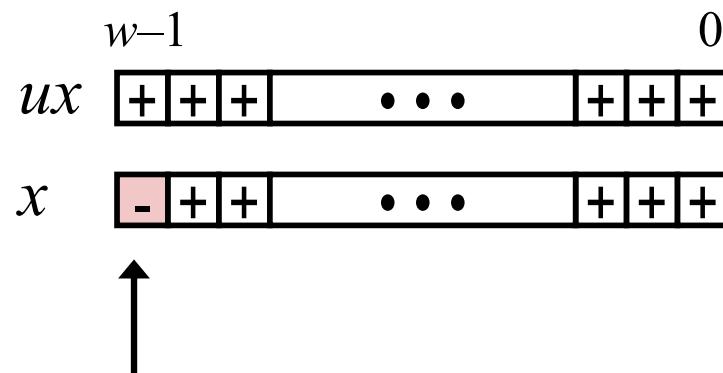
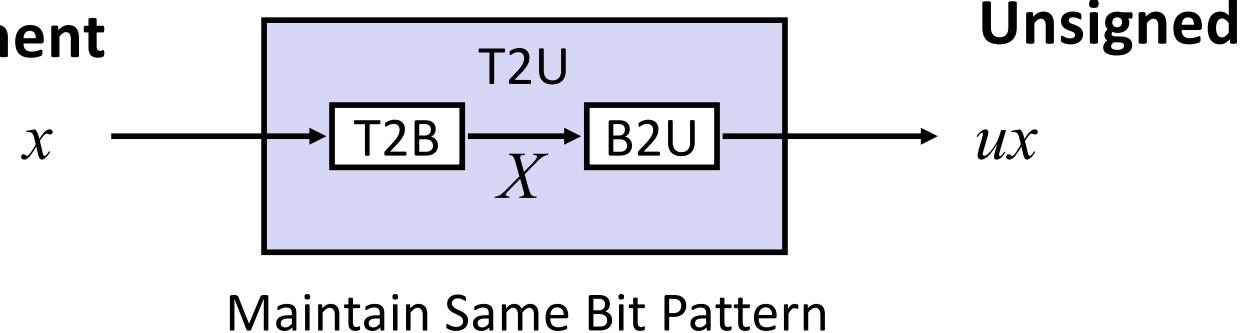


Mapping Signed \leftrightarrow Unsigned



Relation between Signed & Unsigned

Two's Complement

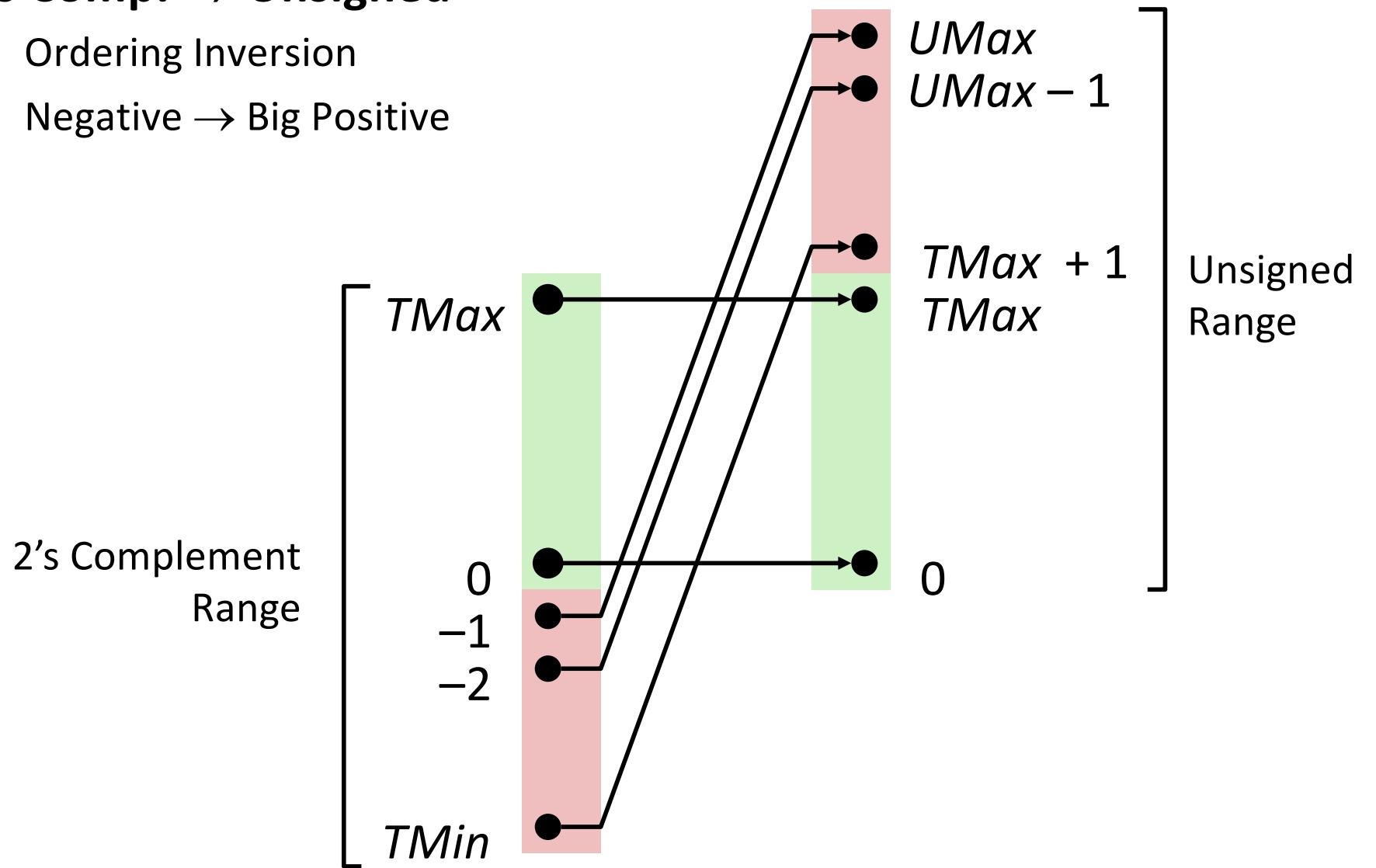


Large negative weight
becomes
Large positive weight

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**

■ Constant ₁	Constant ₂	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

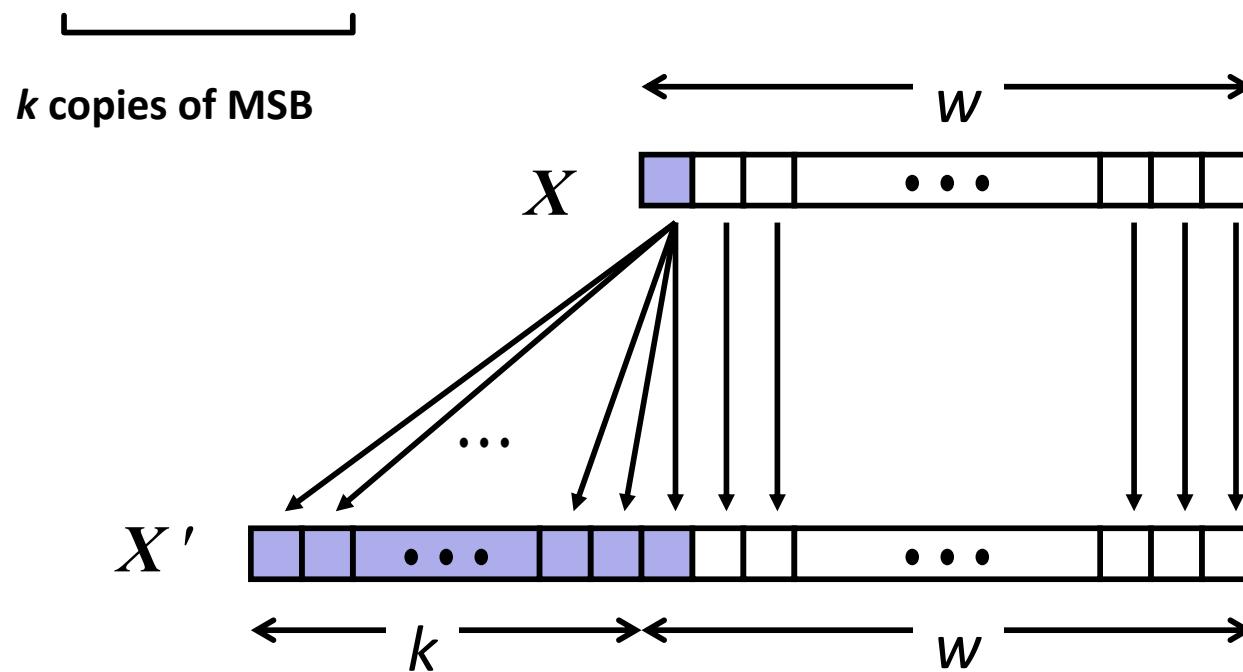
Sign Extension

■ Task:

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

■ Rule:

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



Sign Extension: Simple Example

Positive number

$10 =$

-16	8	4	2	1
0	1	0	1	0

$10 =$

-32	16	8	4	2	1
0	0	1	0	1	0

Negative number

$-10 =$

-16	8	4	2	1
1	0	1	1	0

$-10 =$

-32	16	8	4	2	1
1	1	0	1	1	0

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

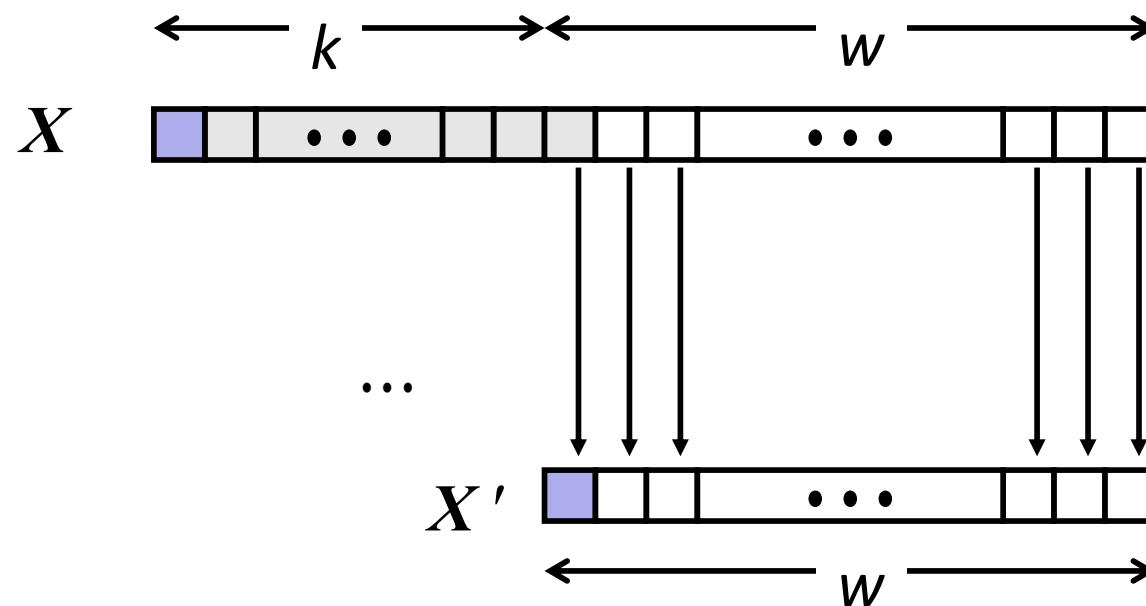
Truncation

■ 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

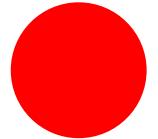
- **Expanding (e.g., short int to int)**
 - Unsigned: zeros added
 - Signed: sign extension
 - Both yield expected result

- **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

불필요한 슬라이드 삭제



Bits, Bytes, and Integers – Part 2

Computer Systems

Friday, September 22 2023

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

Encoding “Integers”

Given a
bit

vector x ,

w bits

long...

Unsigned

$B2U(x)$

$$= \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Examples ($w = 5$)

± 16	8	4	2	1
0	1	0	1	0

Signed (twos complement)

$B2T(x)$

$$= -x_{w-1} \cdot 2^{w-1}$$

$$+ \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

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

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?

$$-x + x == 0 \text{ (by definition)}$$

$$\sim x + x == 1111\dots111 == -1$$

$$\sim x + x + 1 == 0$$

$$(\sim x+1) + x == 0$$

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

	x	10011101
	+	01100010
		-1
		11111111

Example: $x = 15213$

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

Complement & Increment Examples

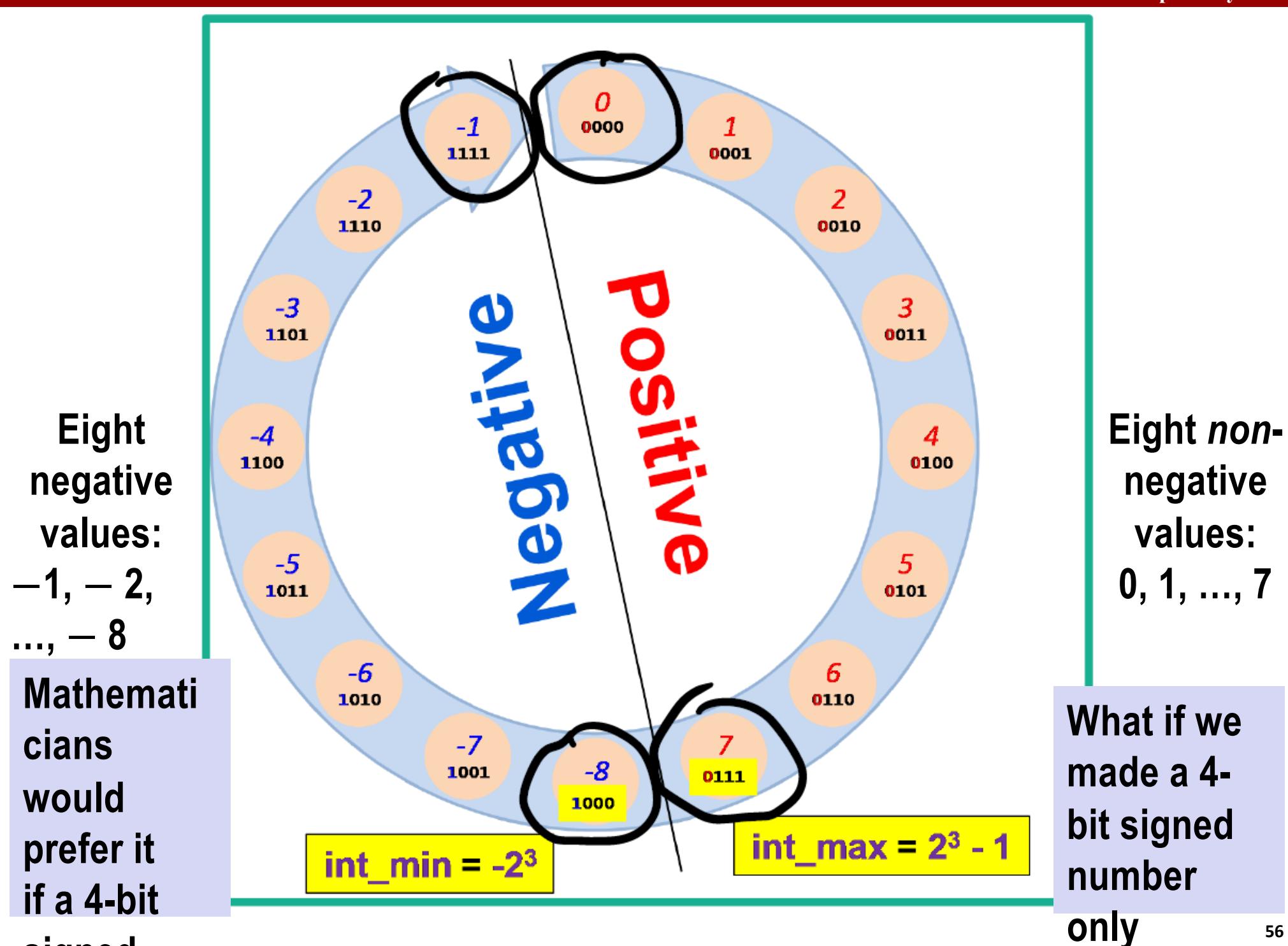
$x = 0$

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~ 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



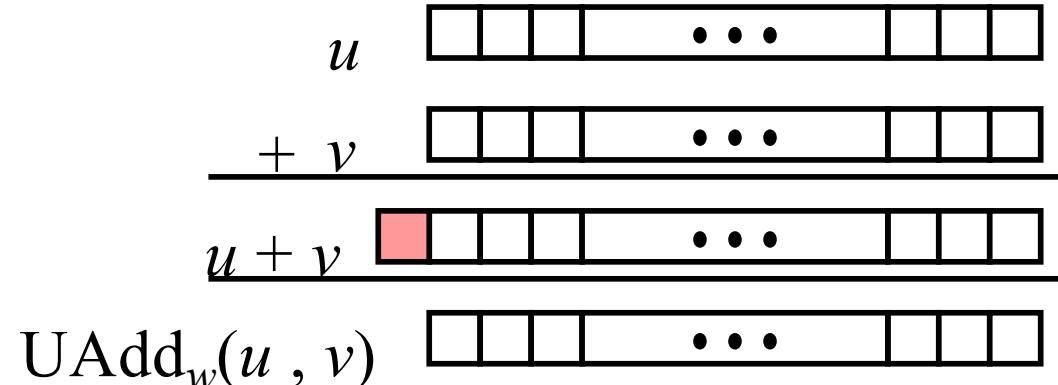


Unsigned Addition

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

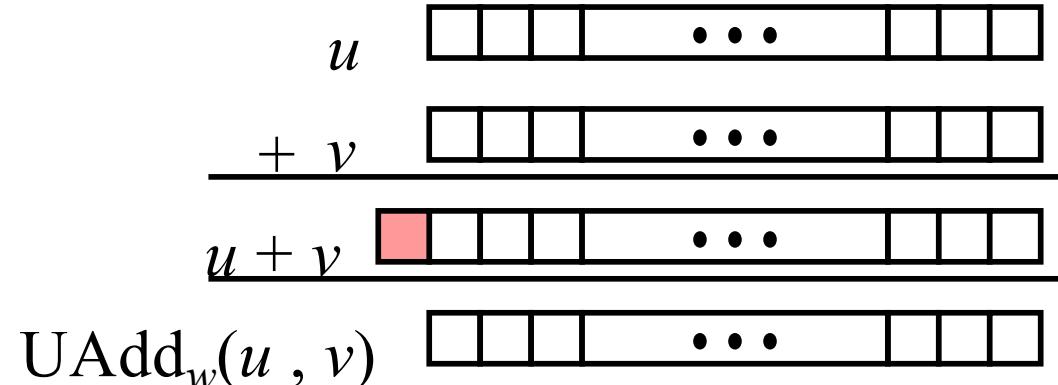
	Hex	Decimal	Binary
0	0	0000	
1	1	0001	
2	2	0010	
3	3	0011	
4	4	0100	
5	5	0101	
6	6	0110	
7	7	0111	
8	8	1000	
9	9	1001	
A	10	1010	
B	11	1011	
C	12	1100	
D	13	1101	
E	14	1110	
F	15	1111	

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`

$$\begin{array}{r}
 1110\ 1001 \\
 +\ 1101\ 0101 \\
 \hline
 1\ 1011\ 1110 \\
 \hline
 1011\ 1110
 \end{array}$$

$$\begin{array}{r}
 \text{E9} \\
 +\ \text{D5} \\
 \hline
 \text{1BE}
 \end{array}
 \quad
 \begin{array}{r}
 233 \\
 +\ 213 \\
 \hline
 446
 \end{array}
 \quad
 \begin{array}{r}
 \text{BE} \\
 +\ \text{BE} \\
 \hline
 190
 \end{array}$$

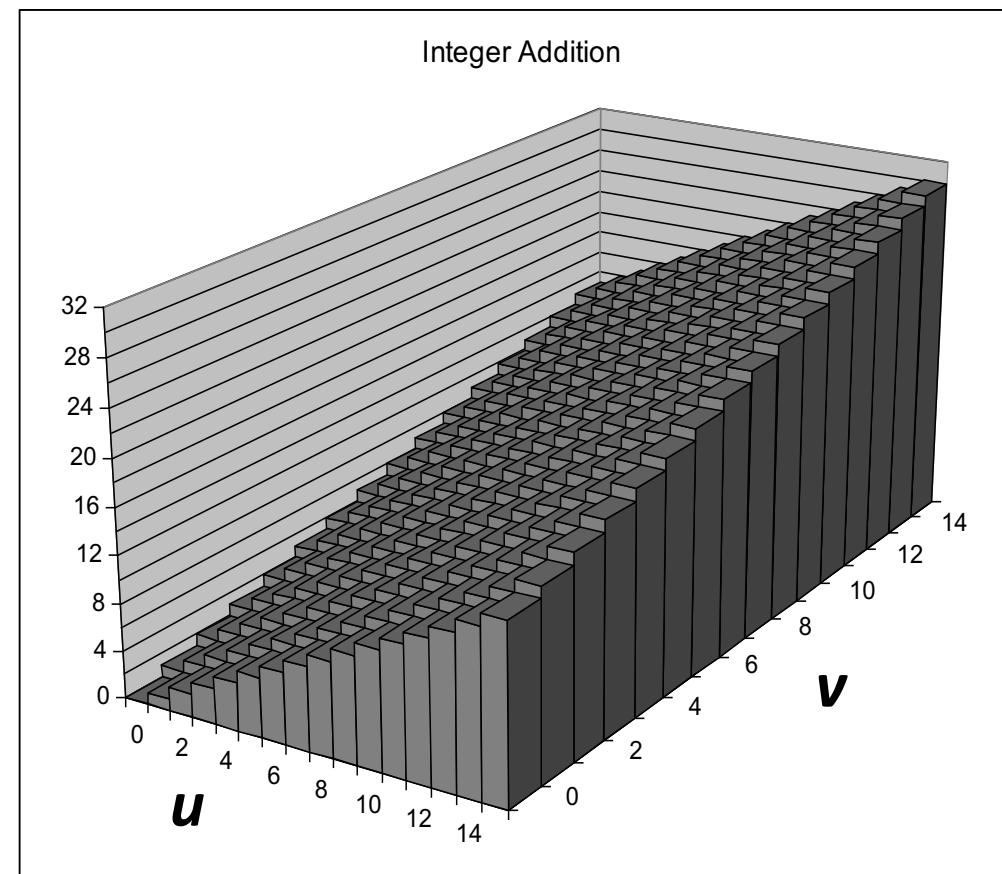
	Hex	Decimal	Binary
0	0	0000	
1	1	0001	
2	2	0010	
3	3	0011	
4	4	0100	
5	5	0101	
6	6	0110	
7	7	0111	
8	8	1000	
9	9	1001	
A	10	1010	
B	11	1011	
C	12	1100	
D	13	1101	
E	14	1110	
F	15	1111	

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

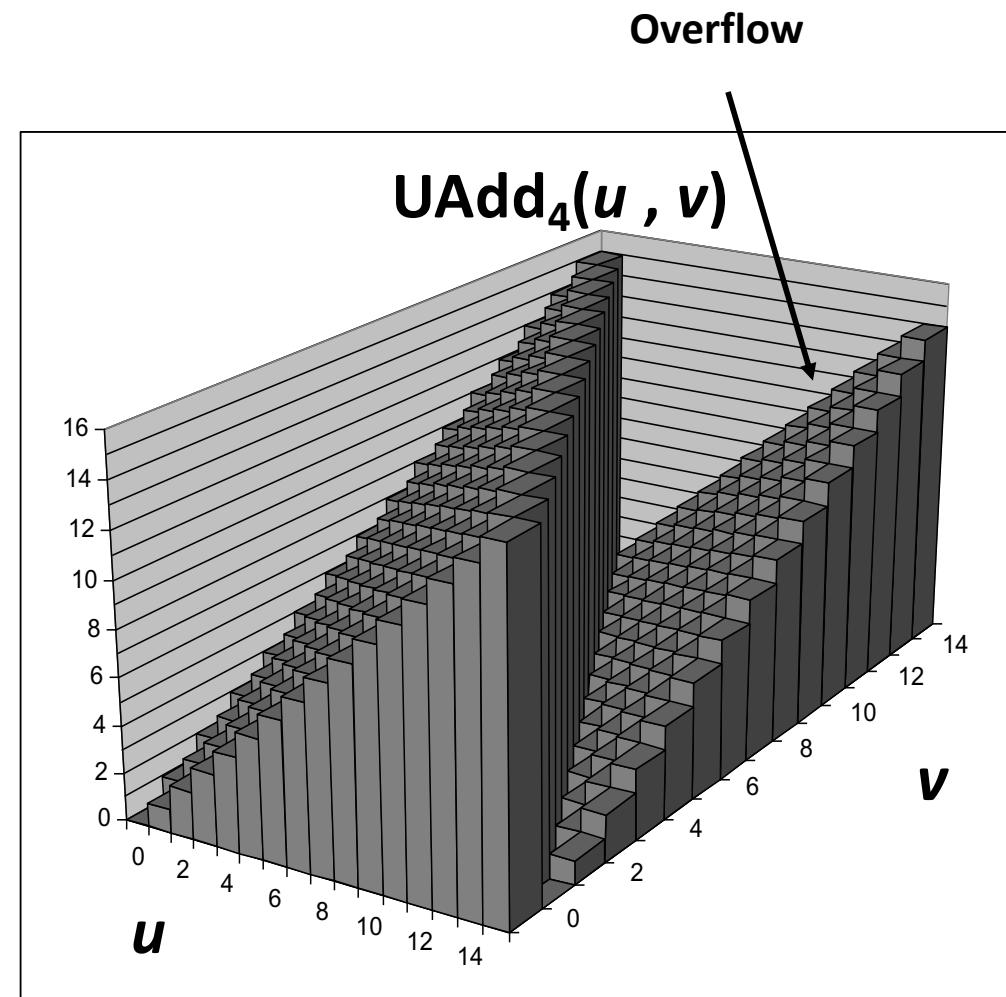
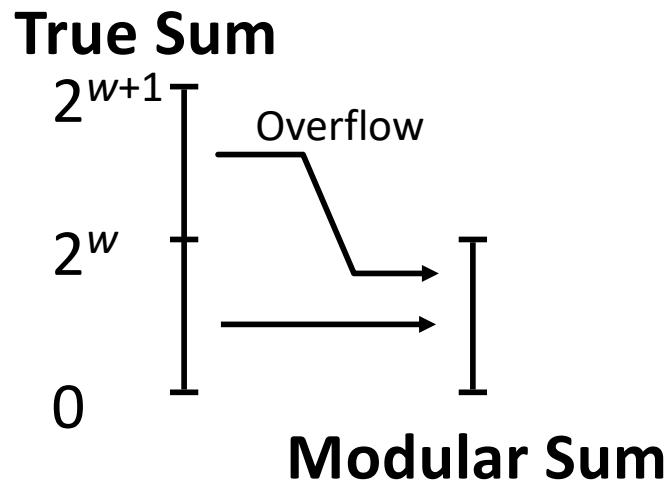
$\text{Add}_4(u, v)$



Visualizing Unsigned Addition

Wraps Around

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

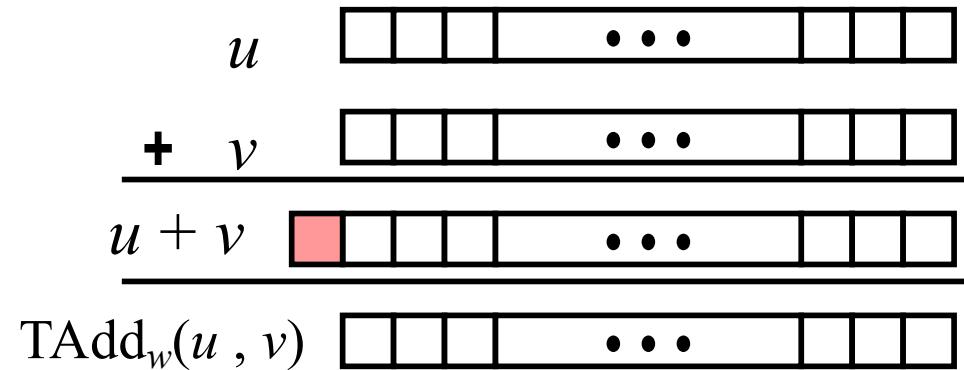


Two's Complement Addition

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 $s == t$

$$\begin{array}{r}
 1110\ 1001 \\
 +\ 1101\ 0101 \\
 \hline
 1\ 1011\ 1110
 \end{array}
 \quad
 \begin{array}{r}
 \text{E9} \\
 +\ D5 \\
 \hline
 1BE
 \end{array}
 \quad
 \begin{array}{r}
 -23 \\
 + -43 \\
 \hline
 -66
 \end{array}$$

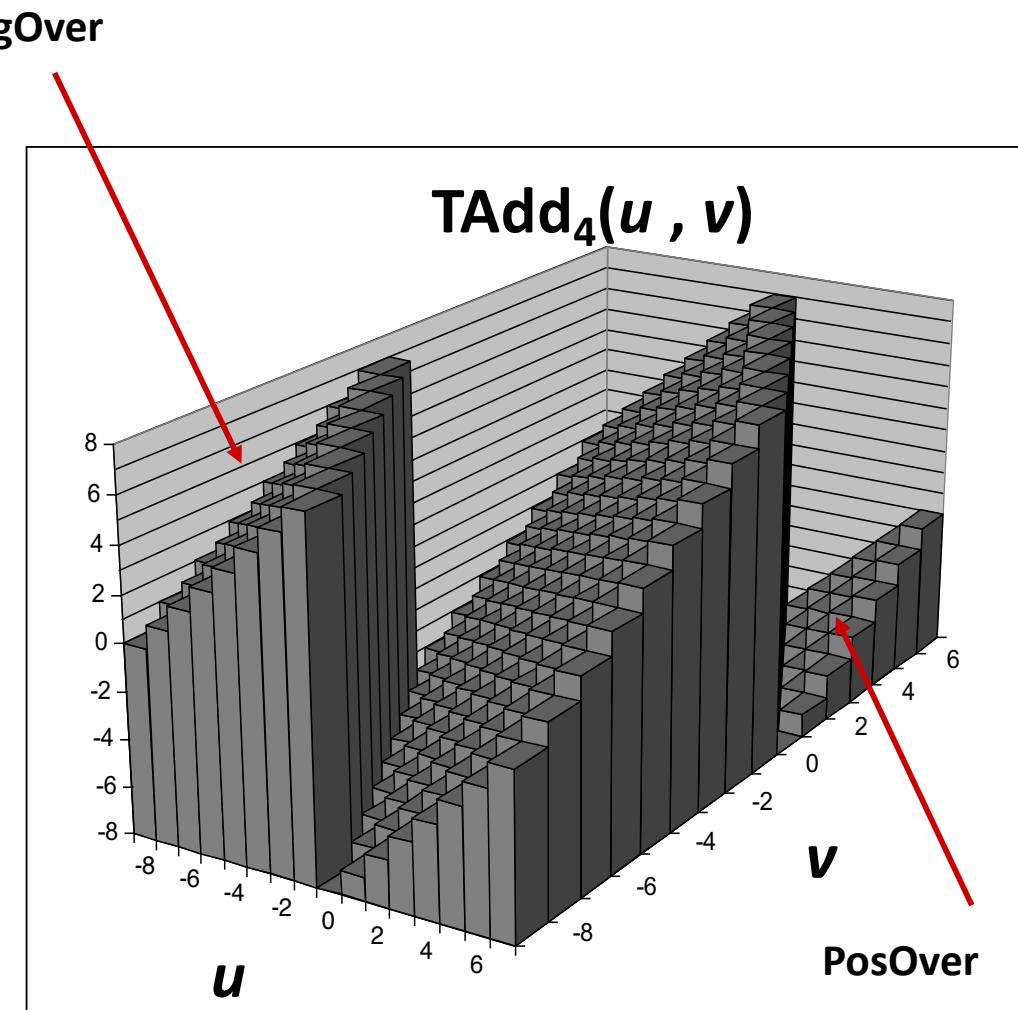
Visualizing 2's Complement Addition

■ 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



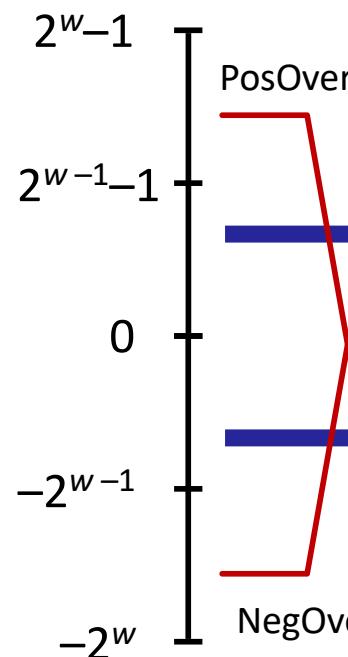
TAdd Overflow

■ Functionality

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

0 111...1
 0 100...0
 0 000...0
 1 011...1
 1 000...0

True Sum



TAdd Result

011...1
 000...0
 100...0

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)

- $A ^ B = 1$ 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

$$\begin{array}{rcl} 01101001 & 01101001 & 01101001 \\ \underline{\& \ 01010101} & \underline{| \ 01010101} & \underline{\wedge \ 01010101} \\ 01000001 & 01111101 & 00111100 \\ & & \\ & & \sim 01010101 \\ & & 10101010 \end{array}$$

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)

$\sim 0x41 \rightarrow$

$\sim 0x00 \rightarrow$

$0x69 \& 0x55 \rightarrow$

$0x69 | 0x55 \rightarrow$

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 &, |, ~, ^ 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 evaluate both sides

Early short-circuiting

Example:

`!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**

Logical versus Bitwise

X	$\text{!}X$	$\text{!!}X$	$\text{!!}X == X$
-1	0	1	No
0	1	0	Yes
1	0	1	Yes
2	0	1	No

X	$\sim X$	$\sim\sim X$	$\sim\sim X == X$
-1	0	-1	Yes
0	-1	0	Yes
1	-2	1	Yes
2	-3	2	Yes

$\text{!!}x \neq x$

$\sim\sim x == x$

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed; negation and addition

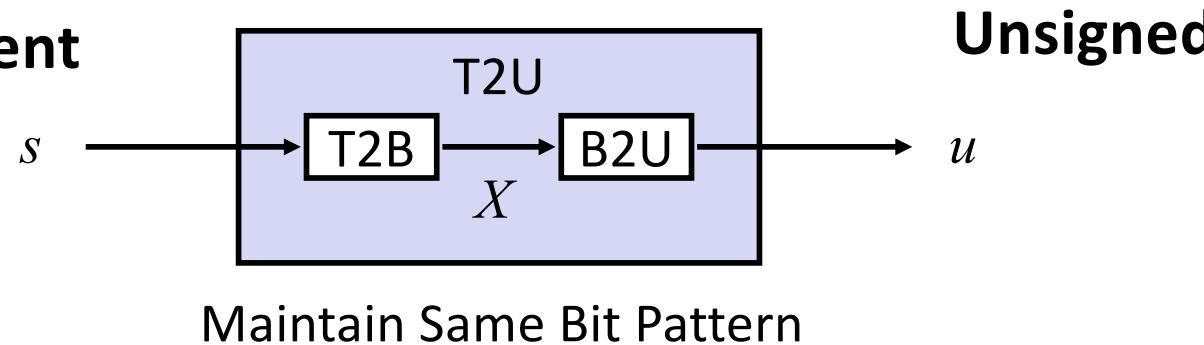
Conversion, casting, extension, truncation

Multiplication, division, shifting

Byte order in memory, pointers, strings

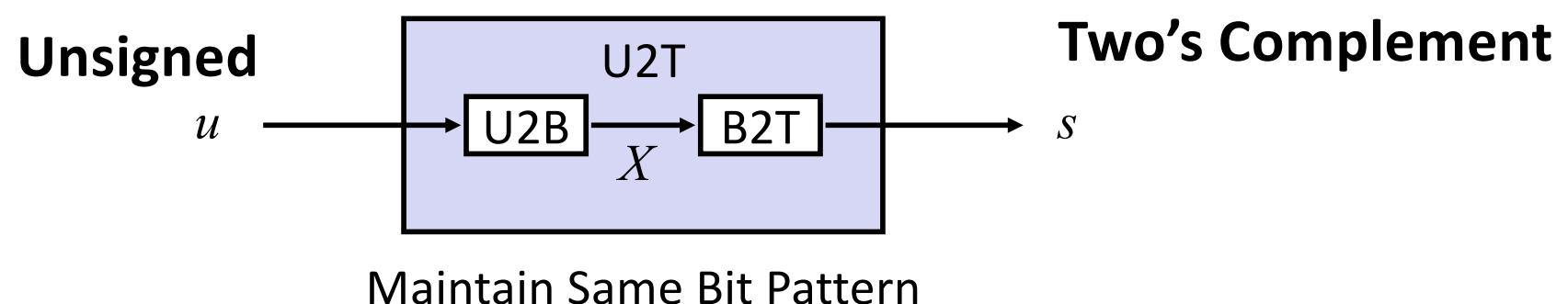
Mapping Between Signed & Unsigned

Two's Complement



Unsigned

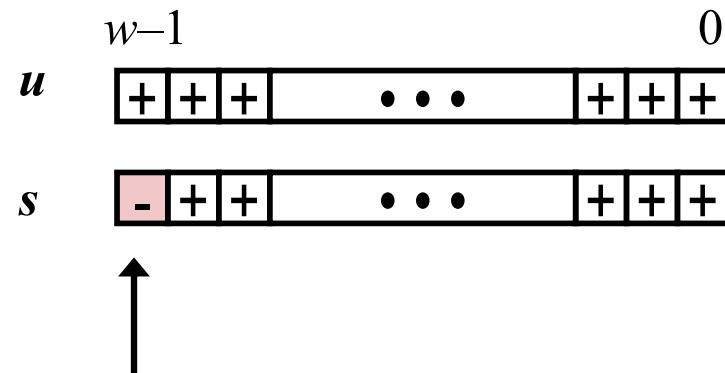
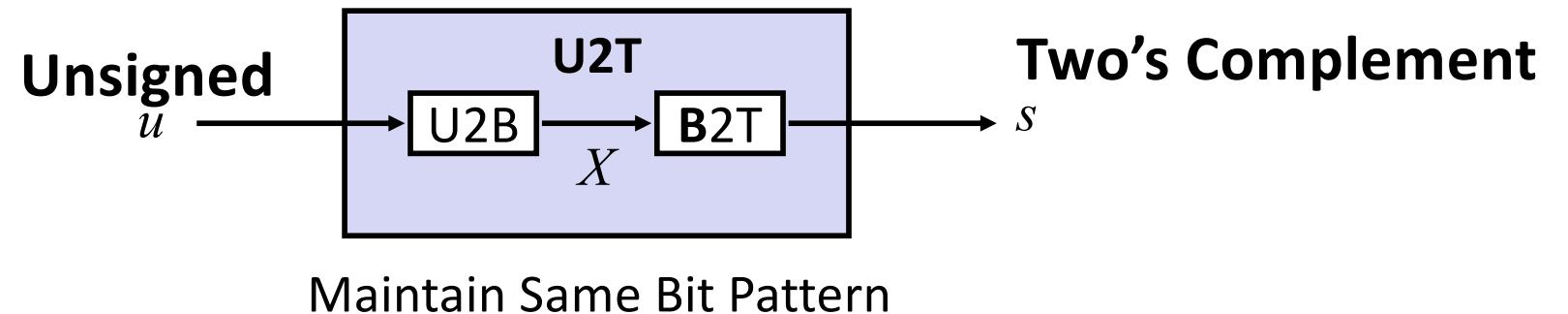
Unsigned



Two's Complement

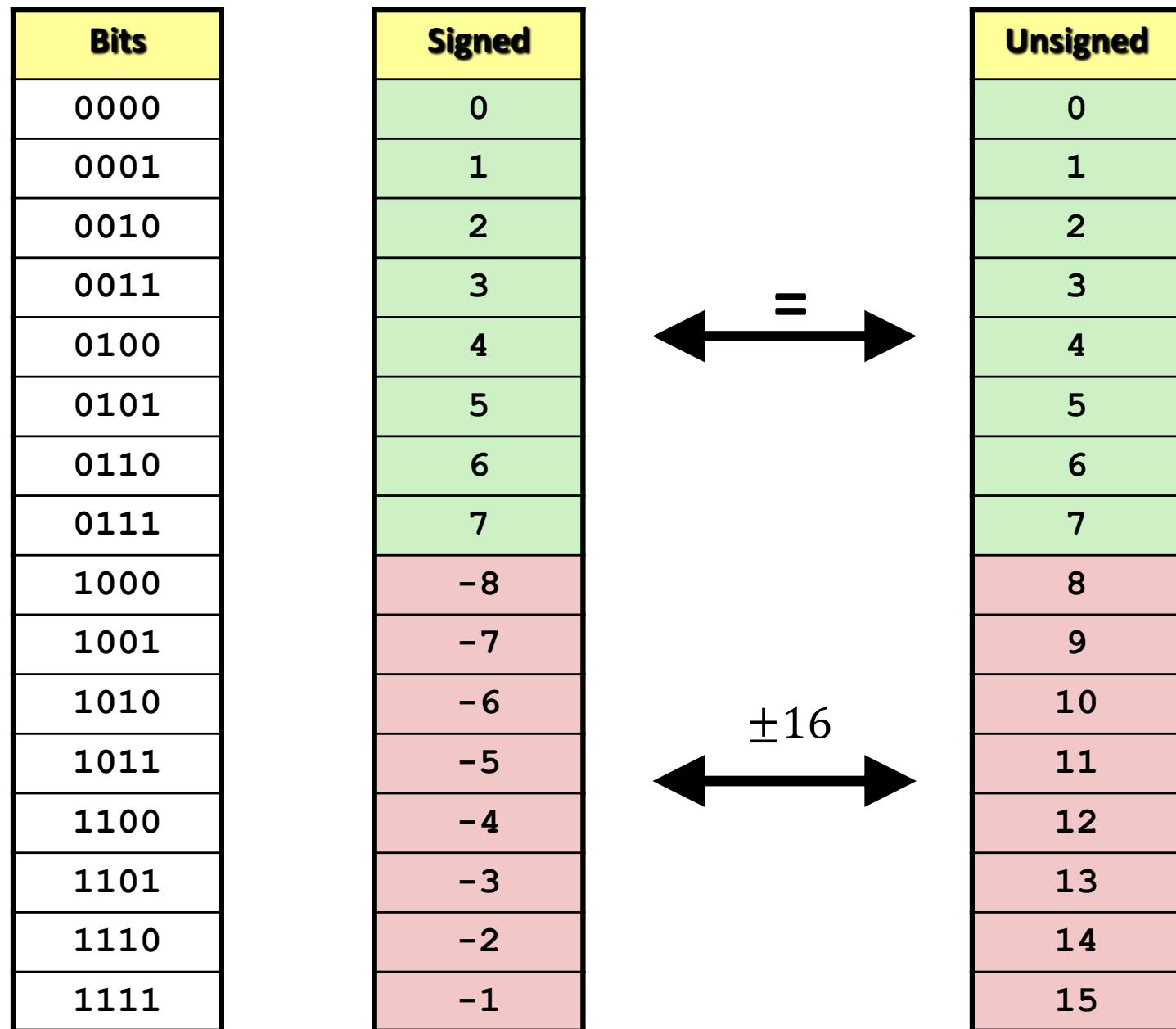
- Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret

Relation between Signed & Unsigned

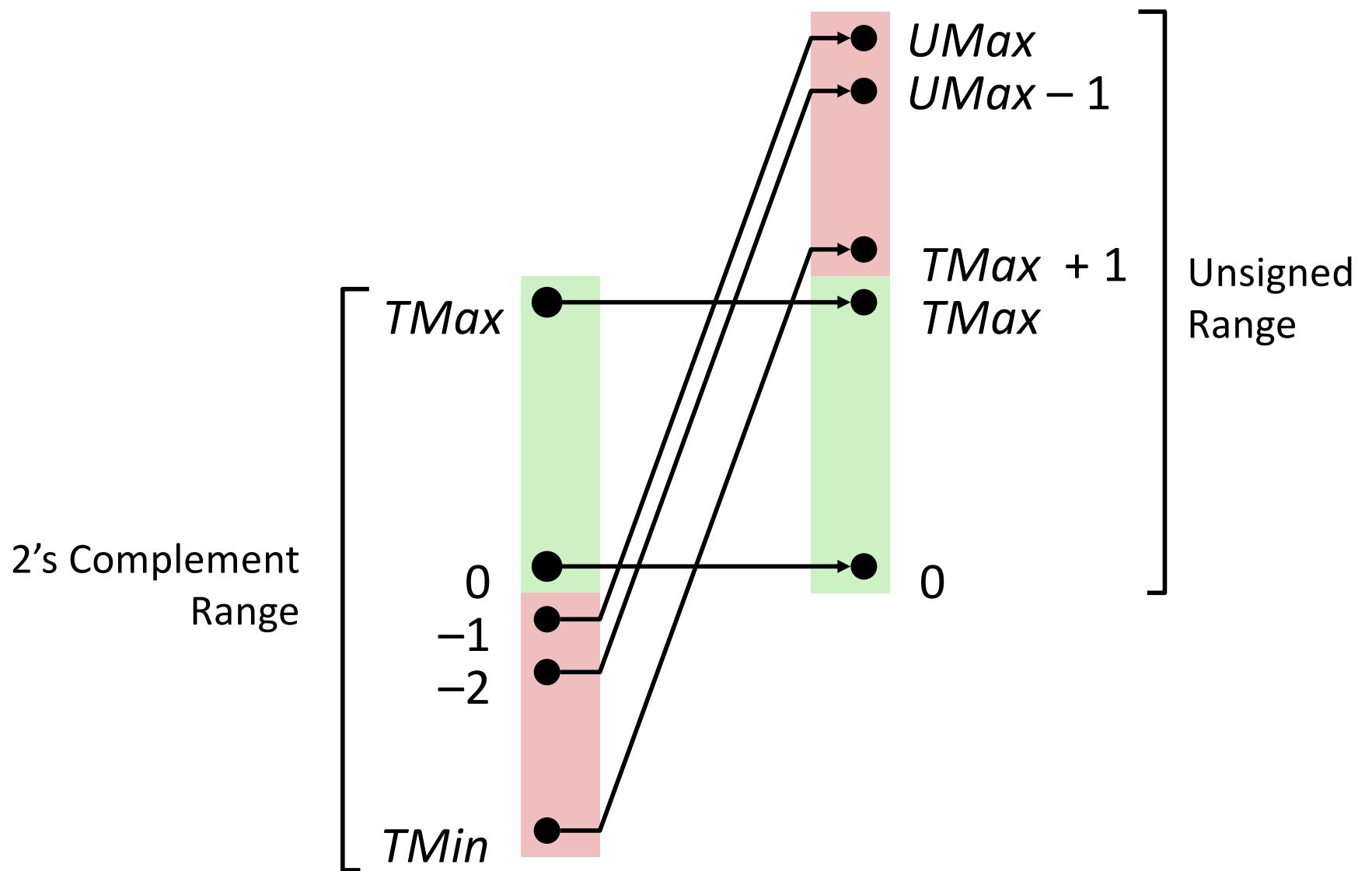


Large positive weight
becomes
Large negative weight

Mapping Signed \leftrightarrow Unsigned



Conversion Visualized



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
-1	0U	>	Unsigned
INT_MAX	INT_MIN	>	Signed
(unsigned) INT_MAX	INT_MIN	<	Unsigned
-1	-2	>	Signed
(unsigned) -1	-2	>	Unsigned
INT_MAX	((unsigned) INT_MAX) + 1	<	Unsigned
INT_MAX	(int) (((unsigned) INT_MAX) + 1)	>	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; negation and addition

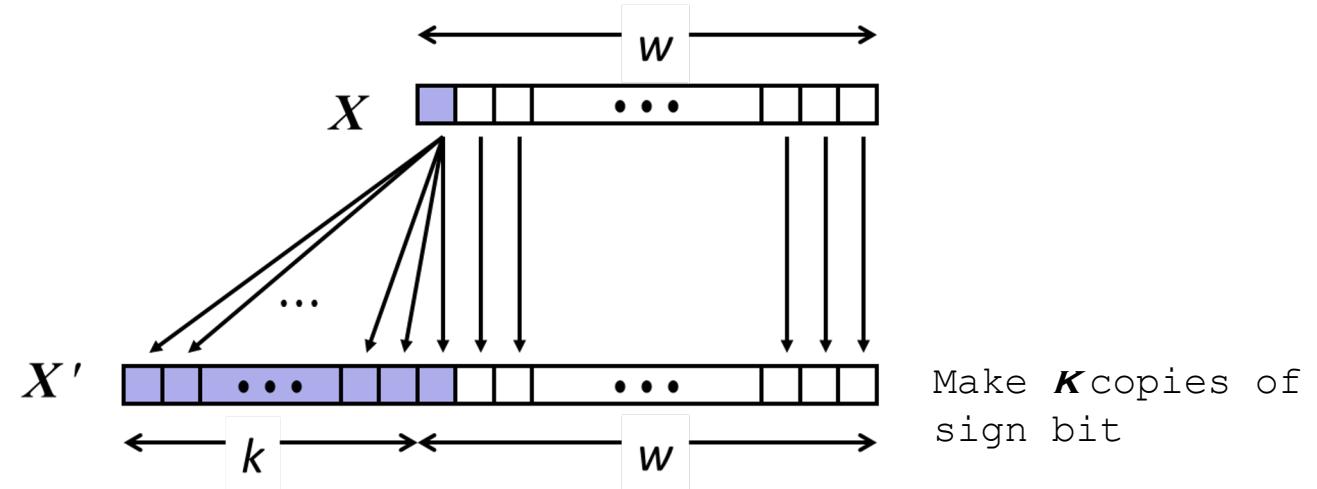
Conversion, casting, extension, truncation

Multiplication, division, shifting

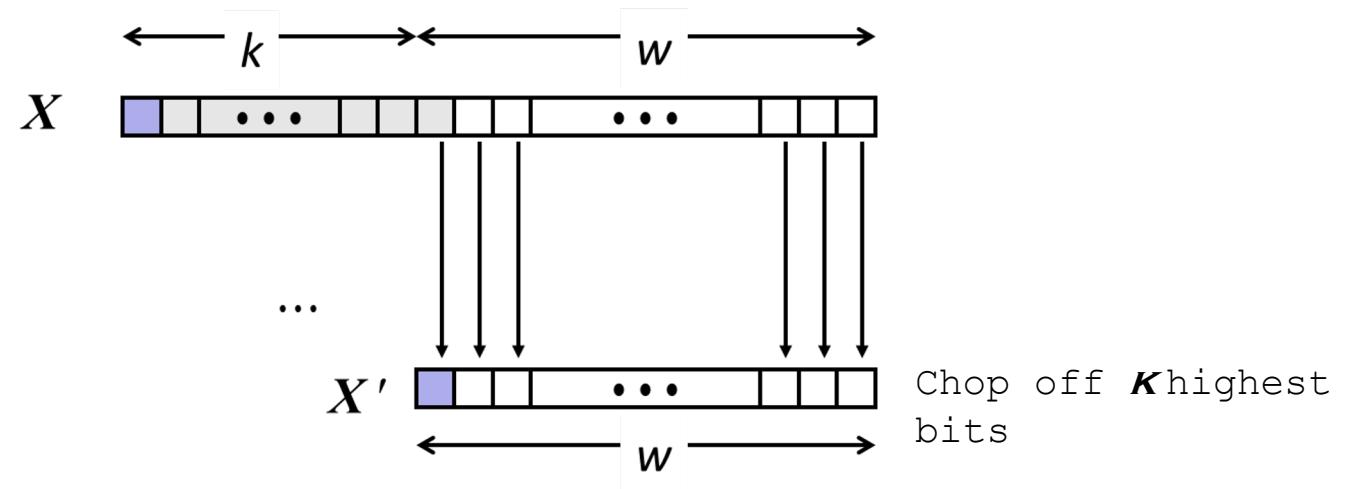
Byte order in memory, pointers, strings

Sign Extension and Truncation

Sign Extension



Truncation



Sign Extension: Simple Example

Positive number

$10 =$

-16	8	4	2	1
0	1	0	1	0

$10 =$

-32	16	8	4	2	1
0	0	1	0	1	0

Negative number

$-10 =$

-16	8	4	2	1
1	0	1	1	0

$-10 =$

-32	16	8	4	2	1
1	1	0	1	1	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

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

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

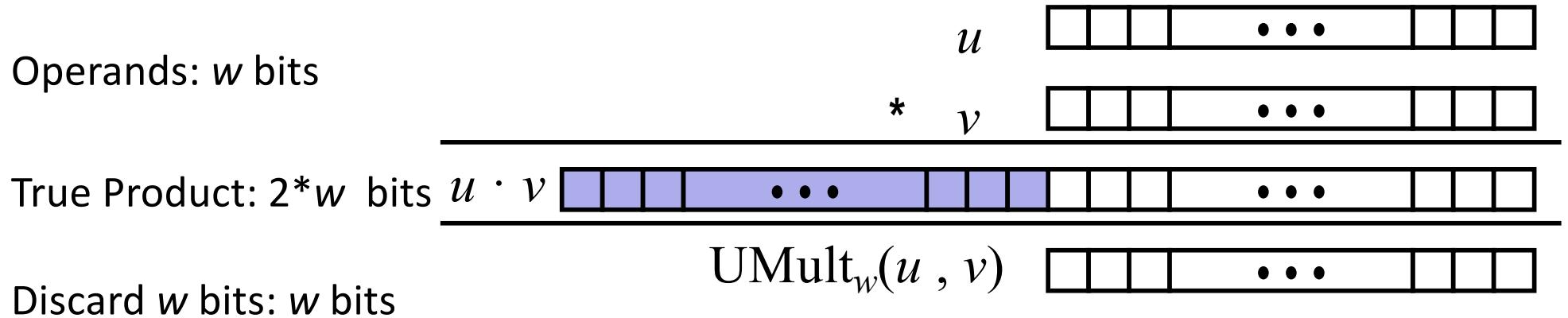
Argument x	
$\ll 3$	
Logical $\gg 2$	
Arithmetic $\gg 2$	
Argument x	
$\ll 3$	
Logical $\gg 2$	
Arithmetic $\gg 2$	

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

Unsigned Multiplication in C

Operands: w bits



■ Standard Multiplication Function

- Ignores high order w bits

■ Implements Modular Arithmetic

$$UMult_w(u, v) = u \cdot v \bmod 2^w$$

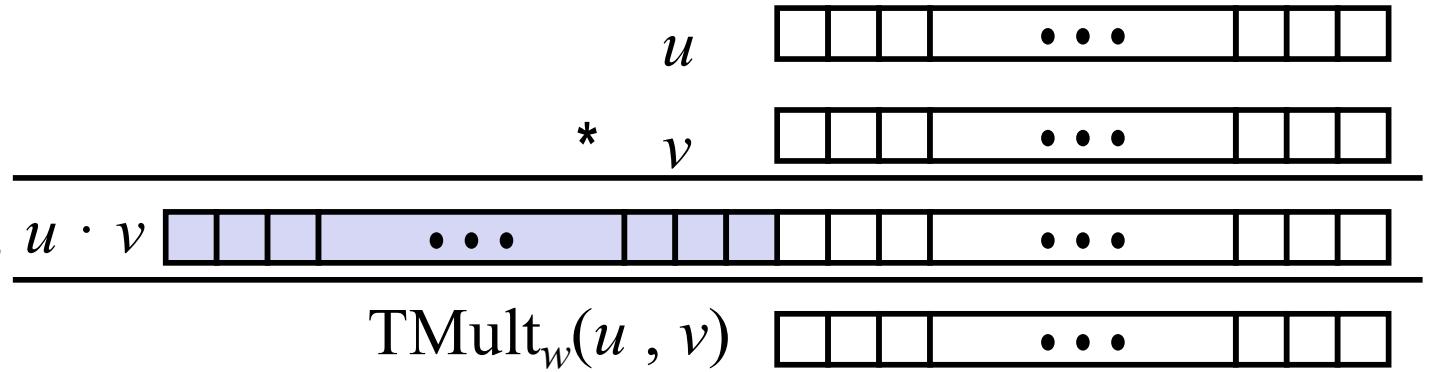
$*$	$1110\ 1001$	$*$	$E9$	$*$	233
	$1101\ 0101$		$D5$		213
	$1100\ 0001$		$C1DD$		49629
	$1101\ 1101$		DD		221

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

$\begin{array}{r} 1110 \ 1001 \\ * \quad 1101 \ 0101 \\ \hline 0000 \ 0011 \ 1101 \ 1101 \end{array}$	$\begin{array}{r} E9 \\ * \quad D5 \\ \hline 03DD \end{array}$	$\begin{array}{r} -23 \\ * \quad -43 \\ \hline 989 \end{array}$
	$1101 \ 1101$	DD
		-35

Power-of-2 Multiply with Shift

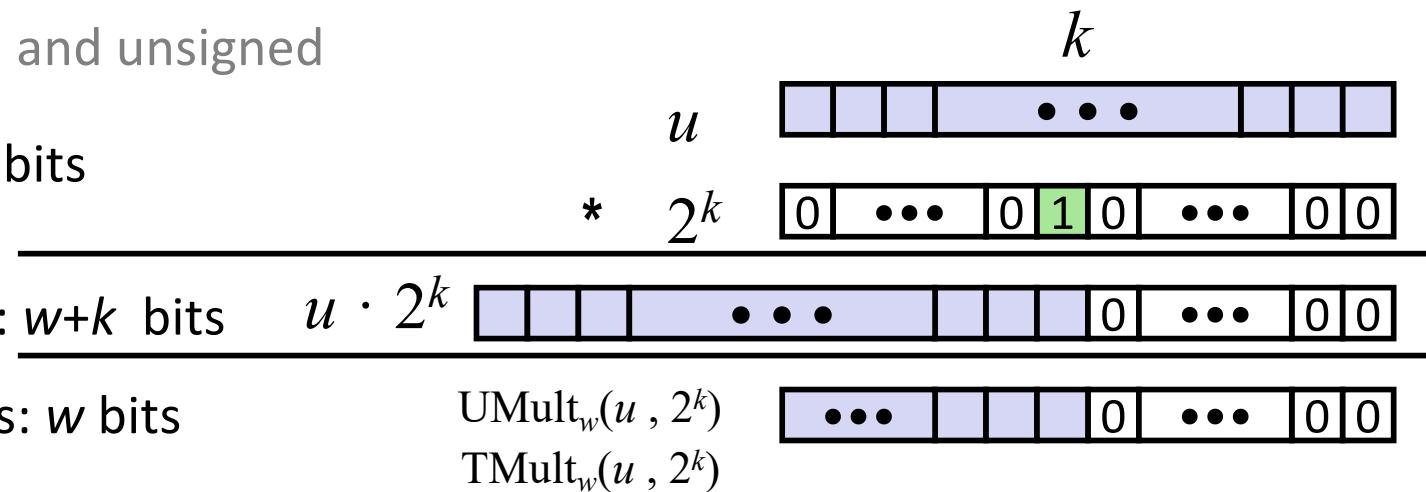
■ Operation

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

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



■ Examples

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- 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

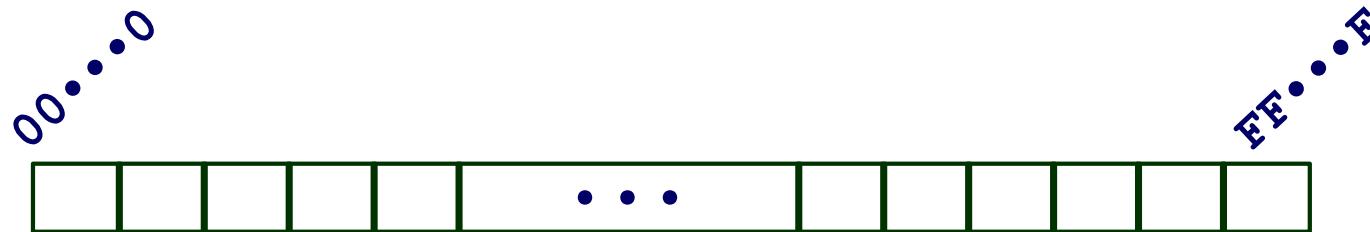
Representation: unsigned and signed; negation and addition

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 18.4×10^{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)

32-bit Words	64-bit Words	Bytes	Addr.
Addr = 0000	Addr = 0000		0000
Addr = 0004			0001
Addr = 0008	Addr = 0008		0002
Addr = 0012			0003
			0004
			0005
			0006
			0007
			0008
			0009
			0010
			0011
			0012
			0013
			0014
			0015

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
pointer	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

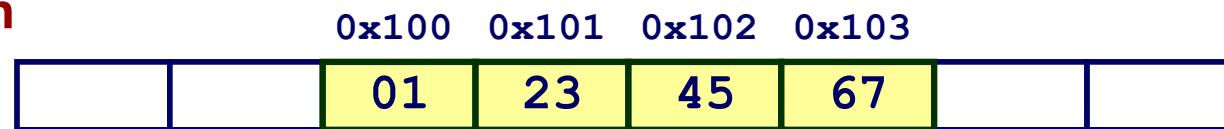
Byte Ordering Example

Example

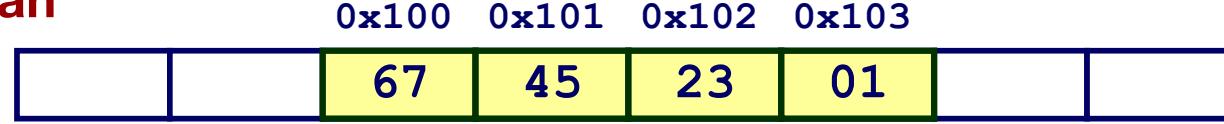
Variable x has 4-byte value of 0x01234567

Address given by &x is 0x100

Big Endian



Little Endian



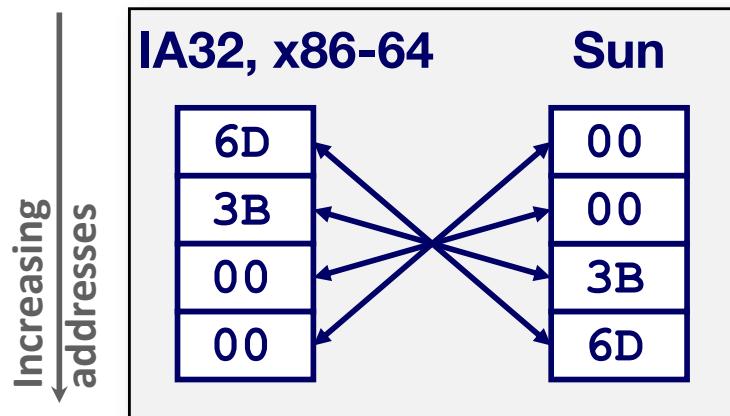
Representing Integers

Decimal: 15213

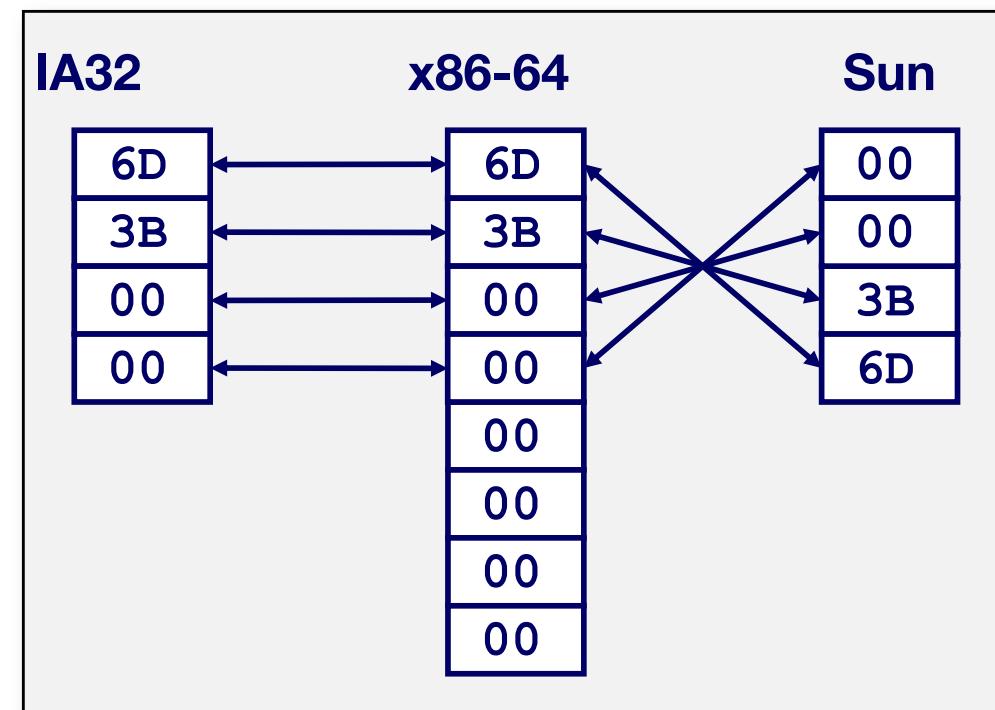
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

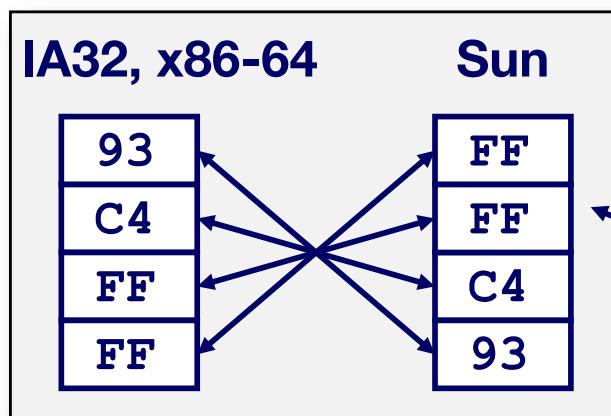
`int A = 15213;`



`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

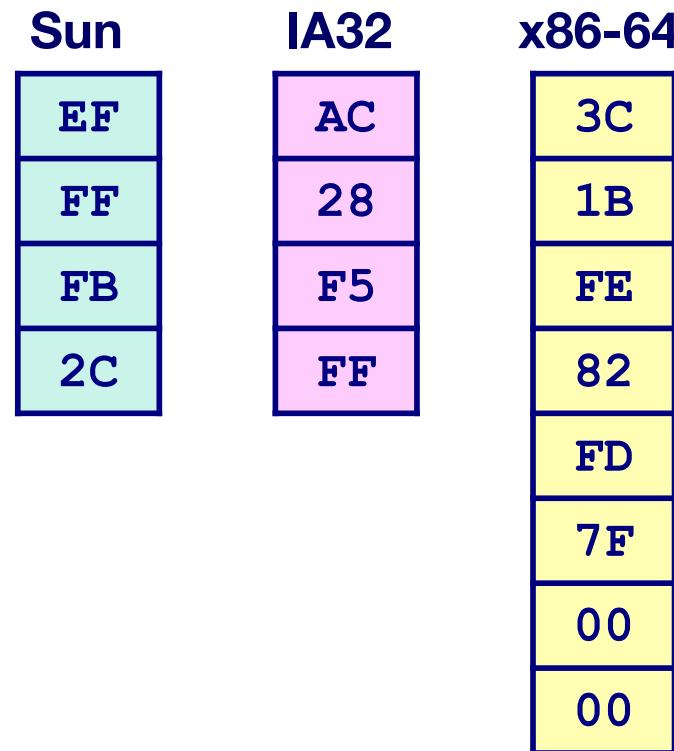
```
int a = 15213;  
printf("int a = 15213;\\n");  
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;  
0x7fffb7f71dbc      6d  
0x7fffb7f71dbd      3b  
0x7fffb7f71dbe      00  
0x7fffb7f71dbf      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

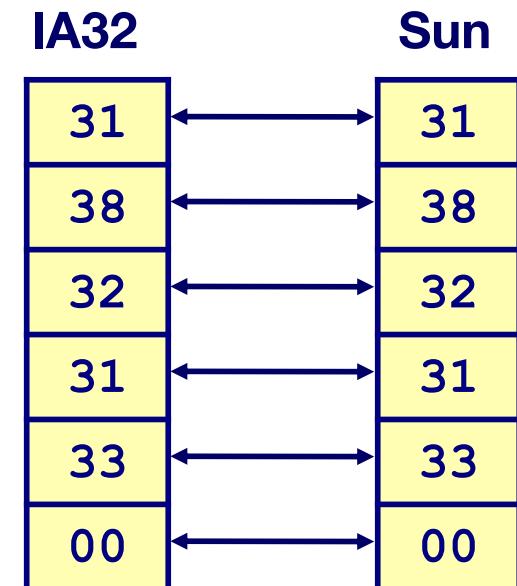
- Digit i has code $0x30+i$

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:
- Pad to 32 bits:
- Split into bytes:
- Reverse:

0x12ab
0x000012ab
00 00 12 ab
ab 12 00 00

2's Complement

■ Exercise

Number	Decimal Representation	Binary Representation
Zero	0	
n/a	-1	
n/a	-2	
n/a	11	
n/a	-13	
n/a	-14	
n/a		0 1100
n/a		1 0100
TMax		
TMin		
TMax+TMin		
TMin+1		
TMax+1		
-TMax		
-TMin		

2's Complement

■ Exercise

Number	Decimal Representation	Binary Representation
Zero	0	0 0000
n/a	-1	1 1111
n/a	-2	1 1110
n/a	11	0 1011
n/a	-13	1 0011
n/a	-14	1 0010
n/a	12	0 1100
n/a	-12	1 0100
TMax	15	0 1111
TMin	-16	1 0000
TMax+TMin	-1	1 1111
TMin+1	-15	1 0001
TMax+1	-16(TMin)	1 0000
-TMax	-15	1 0001
-TMin	-16(TMin)	1 0000

Homework #3

■ datalab-handout

Name	Description	Rating	Max Ops
<code>bitAnd(x, y)</code>	<code>x & y</code> using only <code> </code> and <code>~</code>	1	8
<code>getByte(x, n)</code>	Get byte <code>n</code> from <code>x</code> .	2	6
<code>logicalShift(x, n)</code>	Shift right logical.	3	20
<code>bitCount(x)</code>	Count the number of 1's in <code>x</code> .	4	40
<code>bang(x)</code>	Compute <code>! n</code> without using <code>!</code> operator.	4	12

Table 1: Bit-Level Manipulation Functions.

Question?