

Integers

'20H2

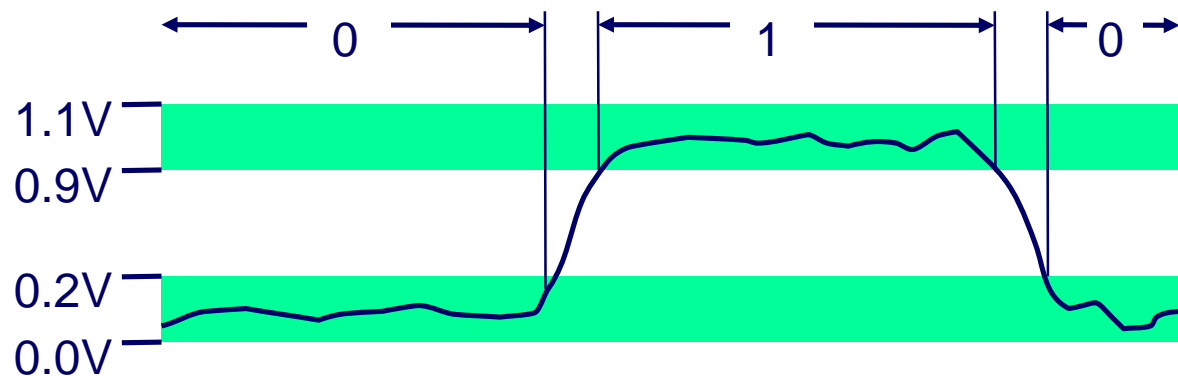
송 인 식

Outline

- Representing information as bits
- Bit-level manipulations
- Integers
- 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 bi-stable elements
 - Reliably transmitted on noisy and inaccurate wires



For example, can count in binary

- Base 2 Number Representation
 - Represent 15213_{10} as 11101101101101_2
 - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
 - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Encoding Byte Values

- Byte = 8 bits
 - Binary 00000000_2 to 11111111_2
 - Decimal: 010 to 255_{10}
 - Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write $FA1D37B_{16}$ in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

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

Example Data Representations

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

Outline

- Representing information as bits
- Bit-level manipulations
- Integers
- Representations in memory, pointers, strings

Boolean Algebra

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

And

- **$A \& B = 1$ when both $A=1$ and $B=1$**

$\&$	0	1
0	0	0
1	0	1

Or

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

$ $	0	1
0	0	1
1	1	1

Not

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

\sim	
0	1
1	0

Exclusive-Or (Xor)

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

\wedge	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

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

- 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)
 - $\sim 0x41 \rightarrow 0xBE$
 - $\sim 01000001_2 \rightarrow 10111110_2$
 - $\sim 0x00 \rightarrow 0xFF$
 - $\sim 00000000_2 \rightarrow 11111111_2$
 - $0x69 \& 0x55 \rightarrow 0x41$
 - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
 - $0x69 | 0x55 \rightarrow 0x7D$
 - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

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

- Logical Operators: `&&`, `||`, `!`

- View 0 as "False"
 - Anything nonzero as "True"
 - Always return 0 or 1
 - **Early termination**

- Examples (char data type)

- `!0x41` → `0x00`
 - `!0x00` → `0x01`
 - `!!0x41` → `0x01`
 - `0x69 && 0x55` → `0x01`
 - `0x69 || 0x55` → `0x01`
 - `p && *p` (avoids null pointer access)

Watch out for `&&` vs. `&` (and `||` vs. `|`)...
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	<u>0</u> 1100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	<u>1</u> 0100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Outline

- Representing information as bits
- Bit-level manipulations
- **Integers**
- Representations in memory, pointers, strings

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 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

	-16	8	4	2	1	
10 =	0	1	0	1	0	$8+2 = 10$

	-16	8	4	2	1	
-10 =	1	0	1	1	0	$-16+4+2 = -10$

Two's 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	
	Integers			

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
- Other Values
 - 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: $abs(TMin)$?
- C Programming
 - `#include <limits.h>`
 - Declares constants, e.g.,
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
 - Values platform specific

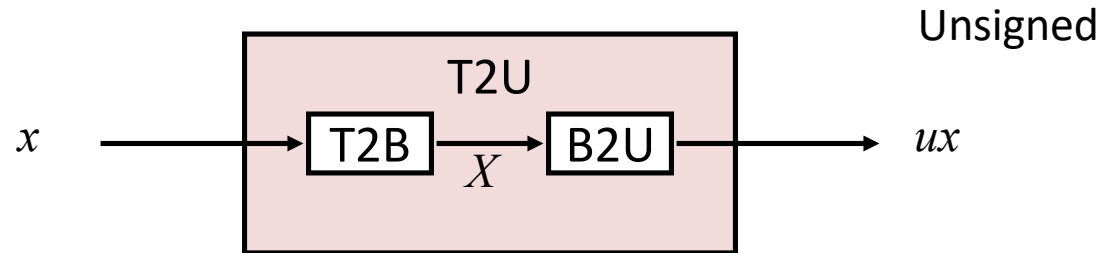
Unsigned & Signed Numeric Values

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
- ➔ 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

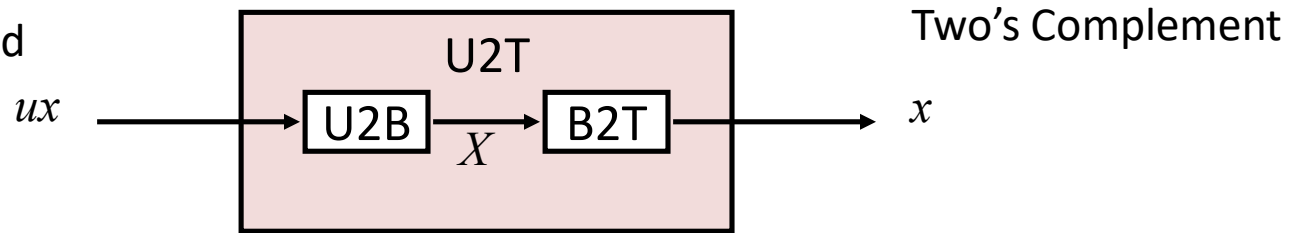
Mapping Between Signed & Unsigned

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

Mapping Signed \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	→ T2U →	5
0110	6	← U2T ←	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

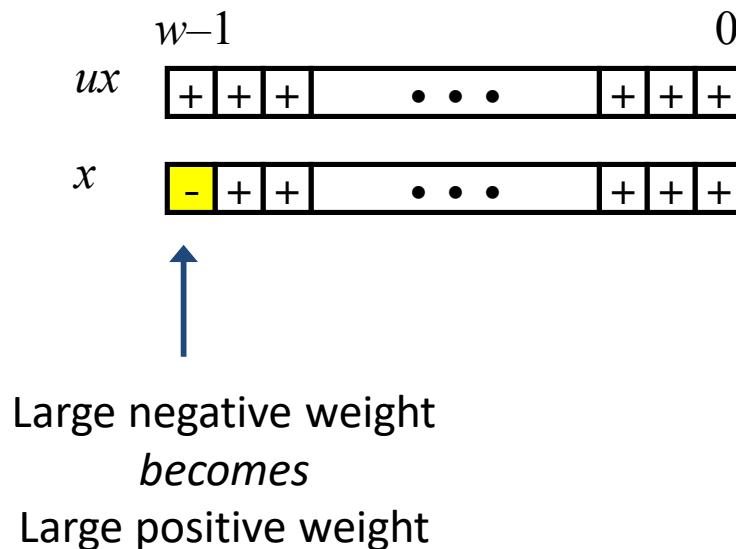
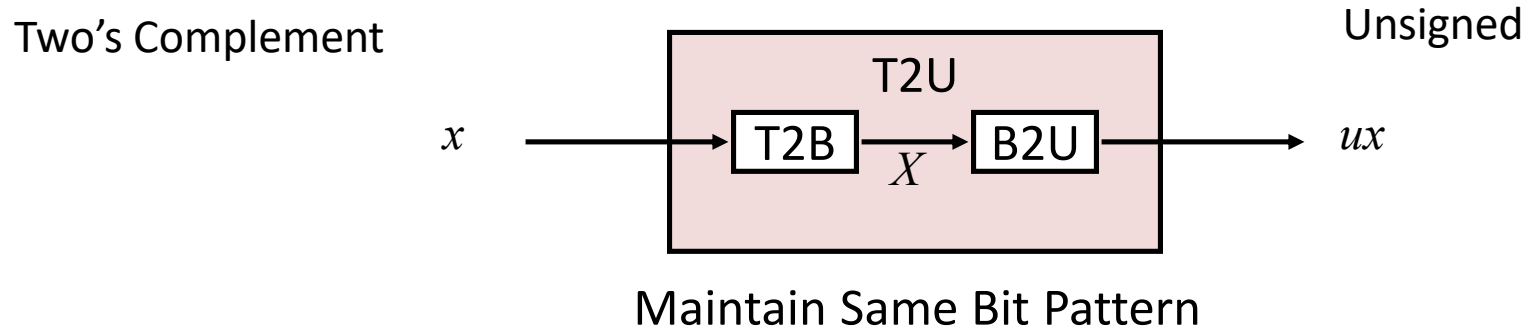
Integers

Mapping Signed \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0	\longleftrightarrow =	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	\longleftrightarrow +/- 16	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

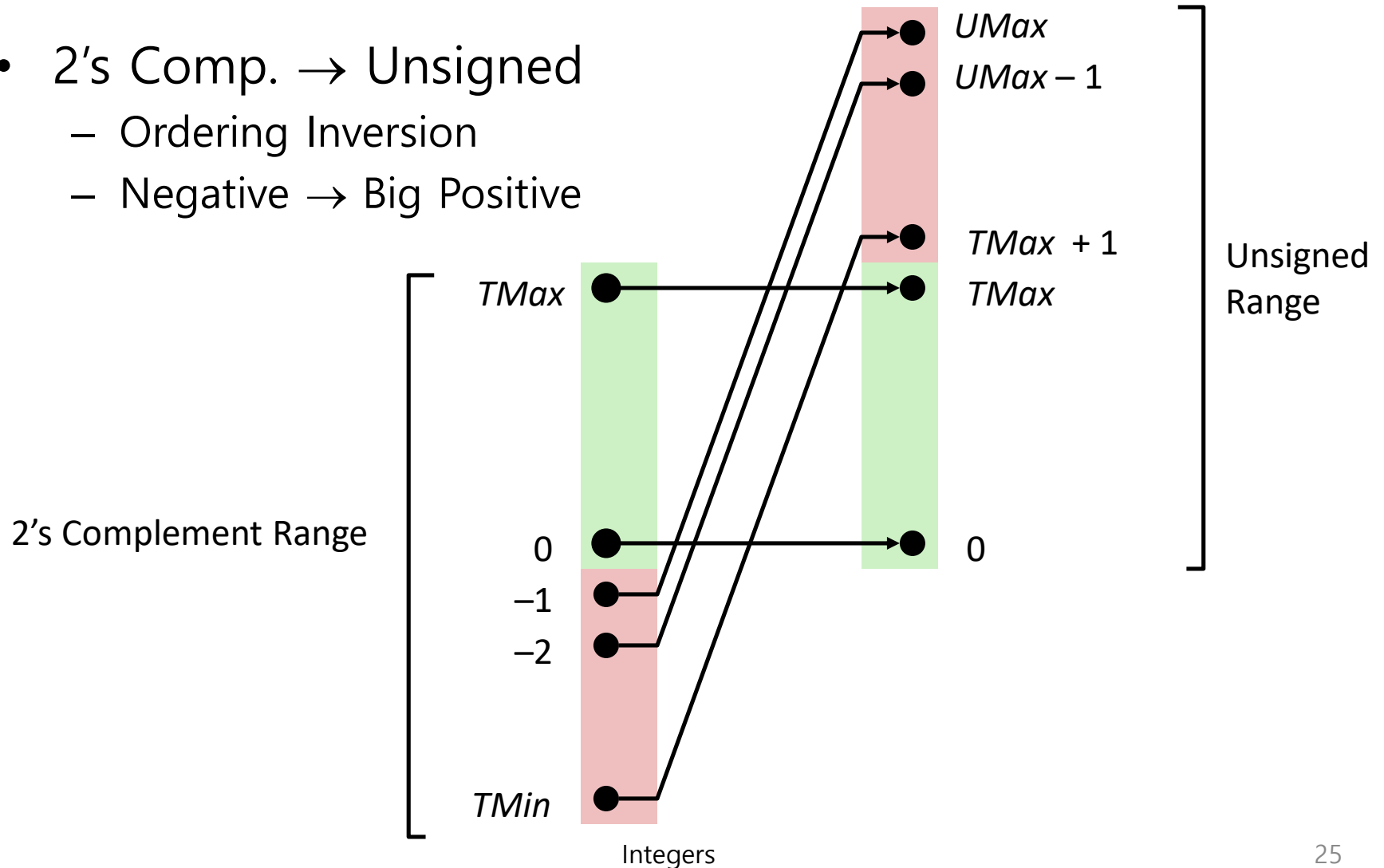
Integers

Relation between Signed & Unsigned



Conversion Visualized

- 2's Comp. \rightarrow Unsigned
 - Ordering Inversion
 - Negative \rightarrow 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;`
 - `uy = ty;`

Casting Surprises

- Expression Evaluation
 - If there is a mix of unsigned and signed in single expression,
signed values implicitly cast to unsigned
 - Including comparison operations $<$, $>$, $==$, $<=$, $>=$
 - Examples for $W = 32$: **$TMIN = -2,147,483,648$, $TMAX = 2,147,483,647$**

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

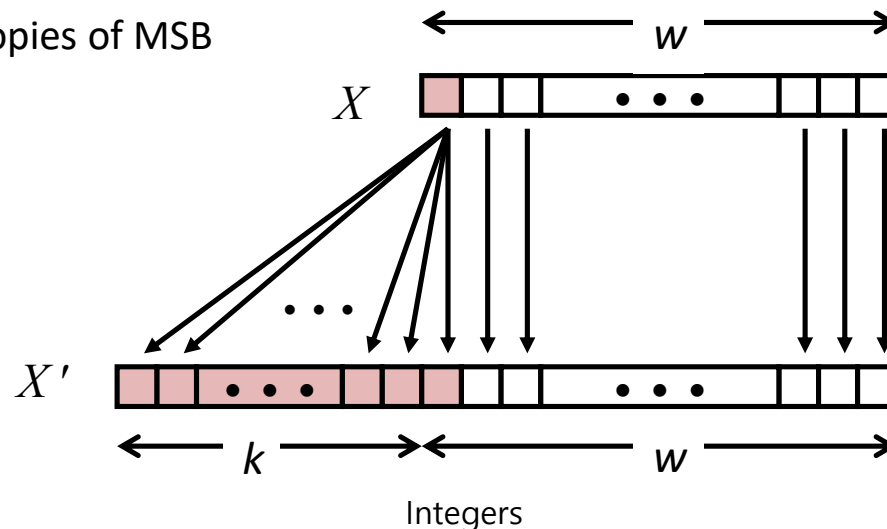
Summary

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!!

Sign Extension

- Task:
 - Given w -bit signed integer x
 - Convert it to $w+k$ -bit integer with same value
- Rule:
 - Make k copies of sign bit:
 - $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \dots, x_0$



Sign Extension: Simple Example

Positive number

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

Sign Extension Example

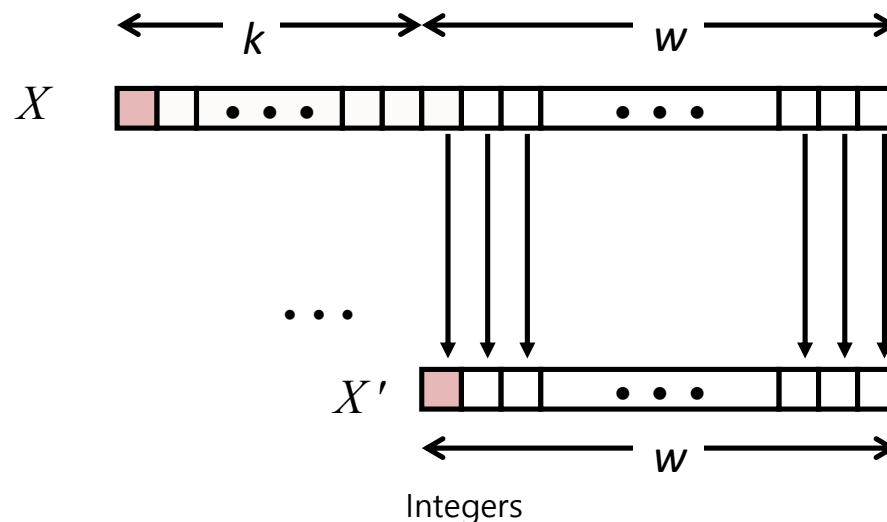
```
short int x = 15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	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

 $2 =$

-16	8	4	2	1
0	0	0	1	0

 $2 =$

-8	4	2	1
0	0	1	0

$$2 \bmod 16 = 2$$

 $-6 =$

-16	8	4	2	1
1	1	0	1	0

 $-6 =$

-8	4	2	1
1	0	1	0

$$-6 \bmod 16 = 26 \bmod 16 = 10 \text{U} = -6$$

Sign change

 $10 =$

-16	8	4	2	1
0	1	0	1	0

 $-6 =$

-8	4	2	1
1	0	1	0

$$10 \bmod 16 = 10 \text{U} \bmod 16 = 10 \text{U} = -6$$

 $-10 =$

-16	8	4	2	1
1	0	1	1	0

 $6 =$

-8	4	2	1
0	1	1	0

$$-10 \bmod 16 = 22 \bmod 16 = 6 \text{U} = 6$$

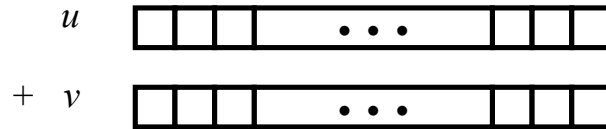
Summary:

Expanding, Truncating: Basic Rules

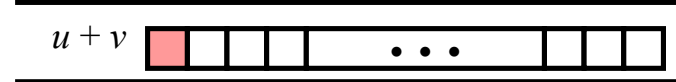
- Expanding (e.g., short int to int)
 - Unsigned: zeros added
 - Signed: sign extension
 - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
 - Unsigned/signed: bits are truncated
 - Result reinterpreted
 - Unsigned: mod operation
 - Signed: similar to mod
 - For small numbers yields expected behavior

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

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

$$\begin{array}{r} \text{E9} \\ +\ \text{D5} \\ \hline 1\text{BE} \\ \hline \text{BE} \end{array}$$

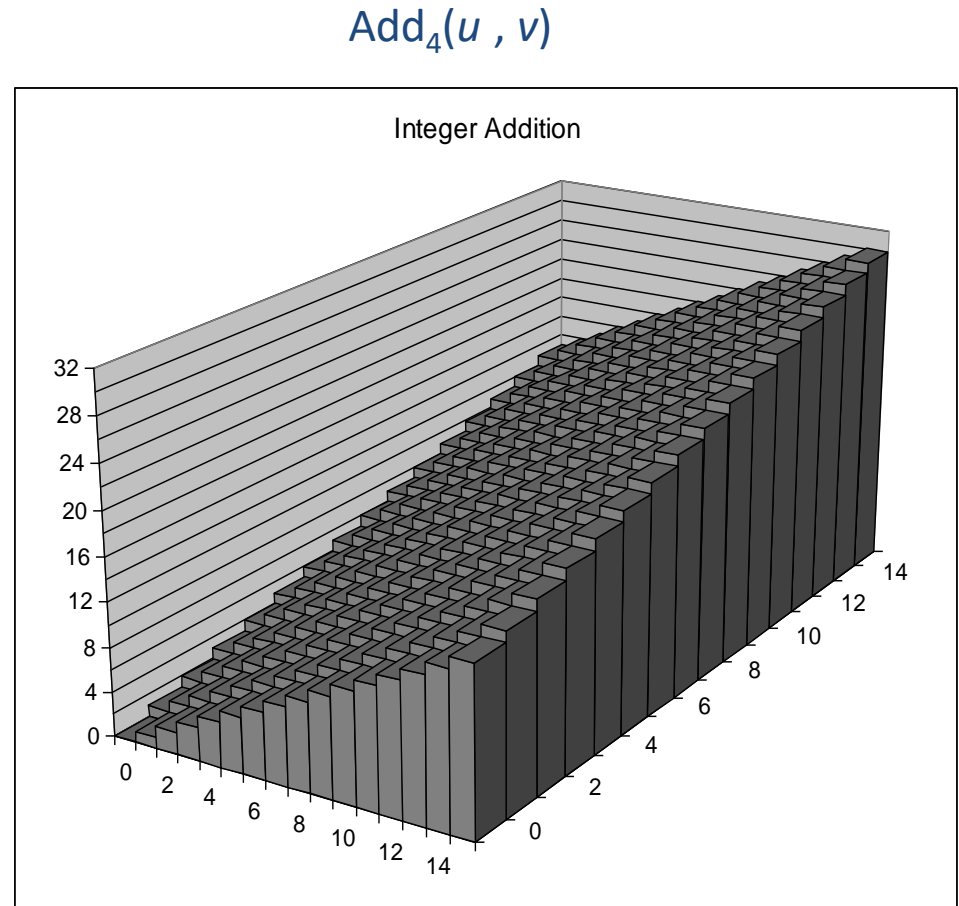
$$\begin{array}{r} 223 \\ +\ 213 \\ \hline 446 \\ \hline 190 \end{array}$$

Integers

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

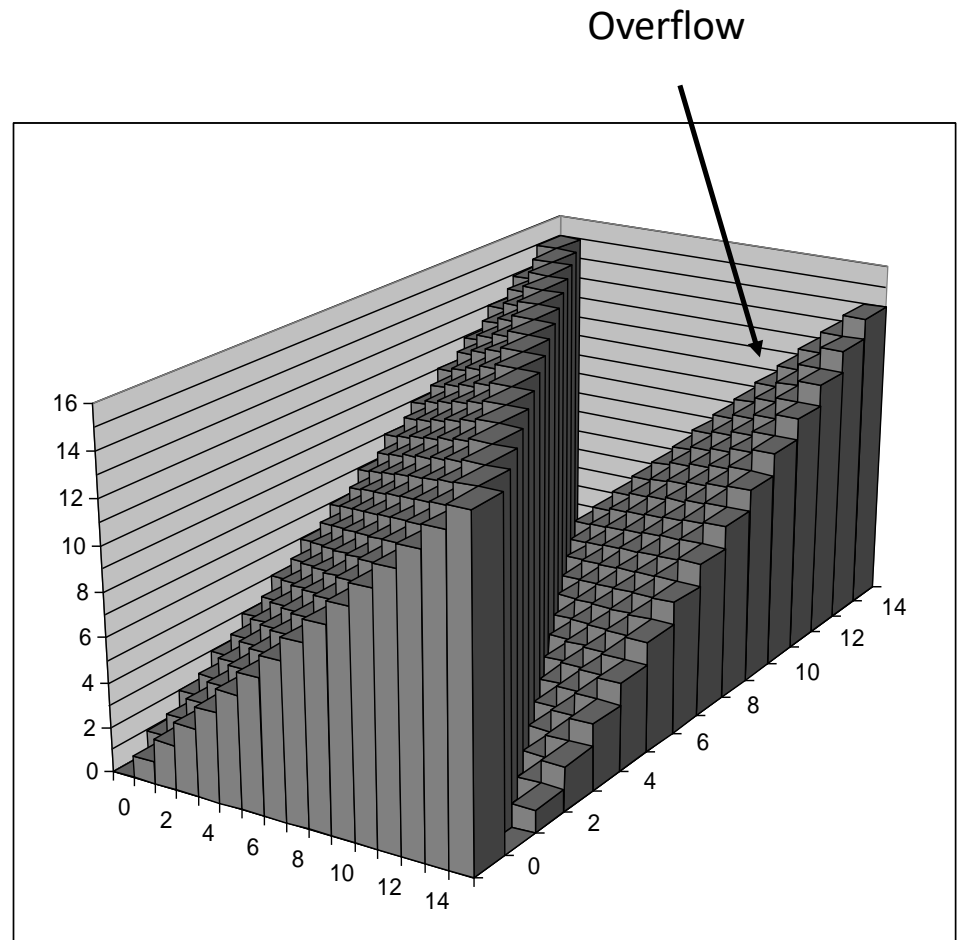
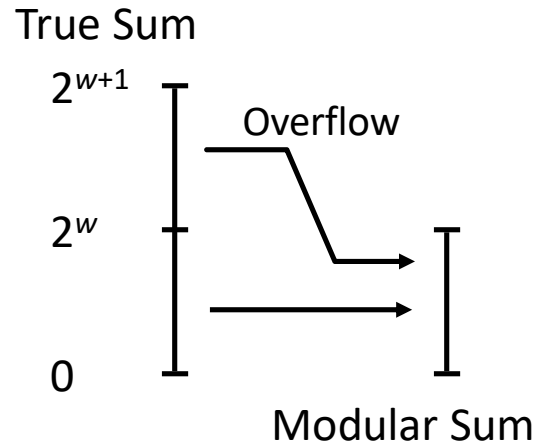
Visualizing (Mathematical) Integer Addition

- 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

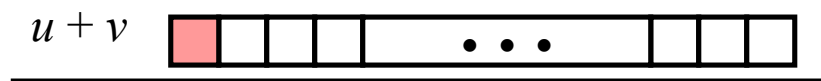


Two's Complement Addition

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);
```

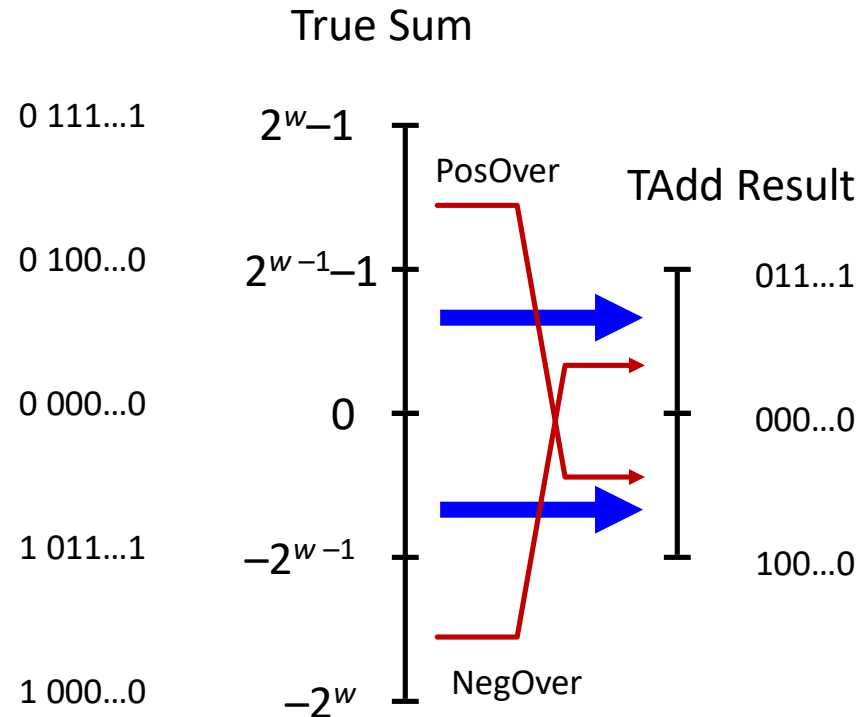
```
t = u + v
```

- Will give `s == t`

	1110 1001	E9	-23
+	1101 0101	+ D5	+ -43
	<u>1 1011 1110</u>	<u>1BE</u>	<u>-66</u>
	1011 1110	BE	-66

TAdd Overflow

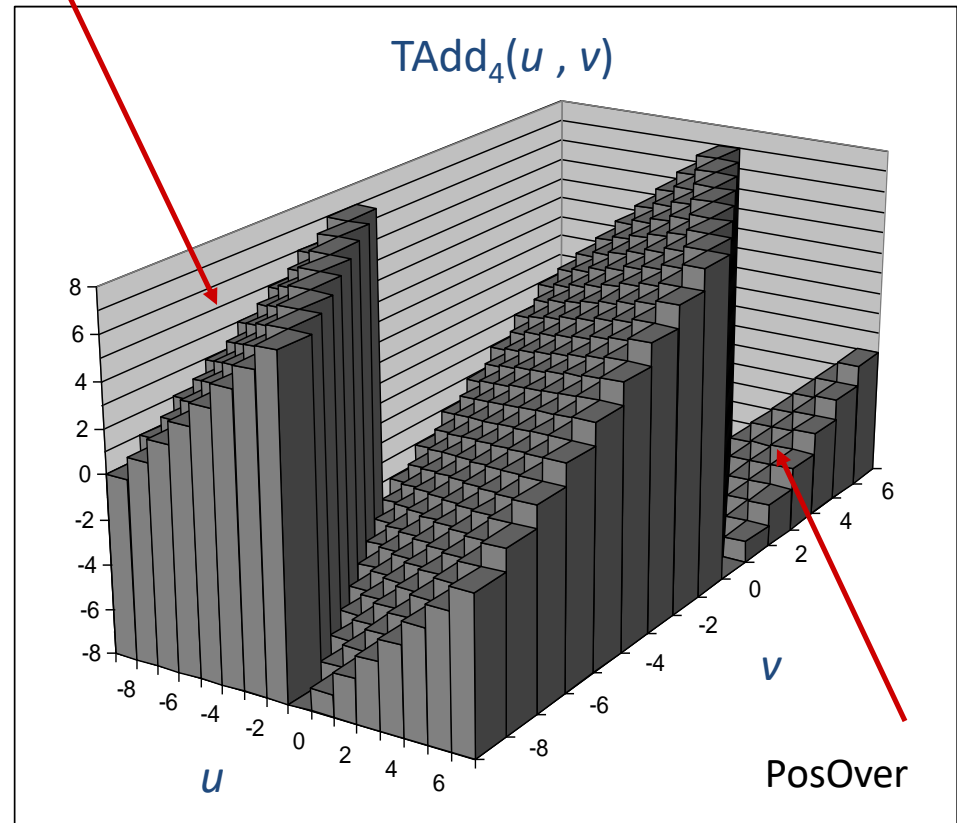
- Functionality
 - True sum requires $w+1$ bits
 - Drop off MSB
 - Treat remaining bits as 2's comp. integer



Visualizing 2's Complement Addition

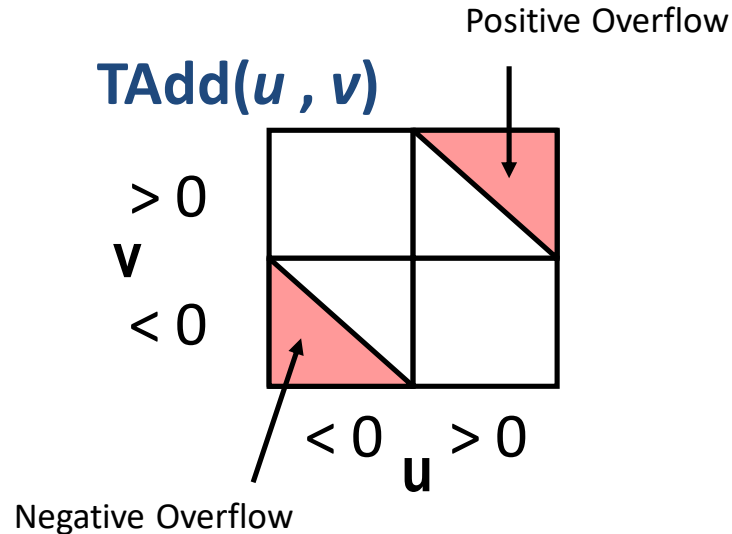
- Values
 - 4-bit two's comp.
 - Range from -8 to +7
- Wraps Around
 - If $\text{sum} \geq 2^{w-1}$
 - Becomes negative
 - At most once
 - If $\text{sum} < -2^{w-1}$
 - Becomes positive
 - At most once

NegOver



Characterizing TAdd

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

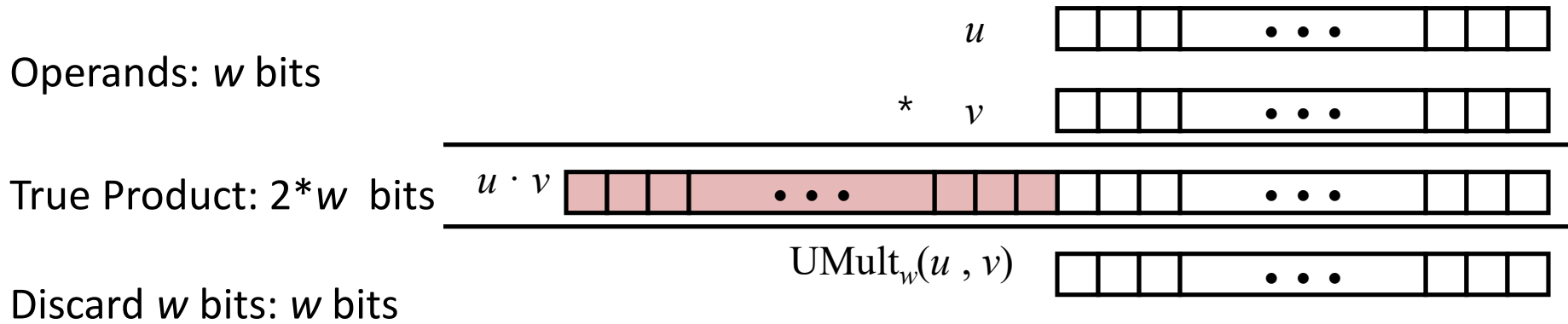


$$TAdd_w(u, v) = \begin{cases} u + v + 2^{w-1} & u + v < TMin_w \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^{w-1} & TMax_w < u + v \end{cases}$$

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



- Standard Multiplication Function
 - Ignores high order w bits
- Implements Modular Arithmetic

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

Signed Multiplication in C

Operands: w bits

u

$*$ v

True Product: $2*w$ bits

$u \cdot v$

Discard w bits: w bits

$\text{TMult}_w(u, v)$

- Standard Multiplication Function

- Ignores high order w bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

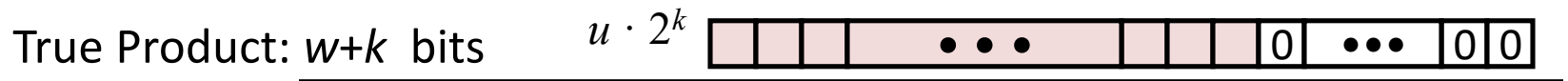
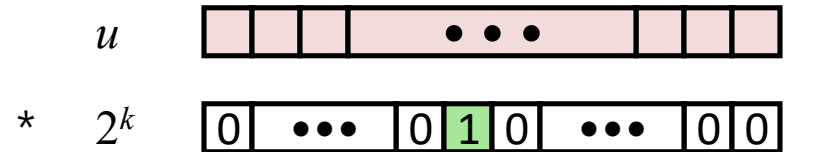
	1110	1001
*	1101	0101
<hr/>		
	0000	0011 1101 1101
<hr/>		
		1101 1101

	E9	-23
*	D5	* -43
<hr/>		
	03DD	989
<hr/>		
	DD	-35

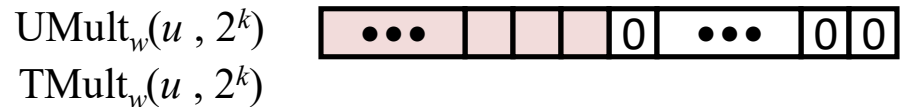
Power-of-2 Multiply with Shift

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

Operands: w bits



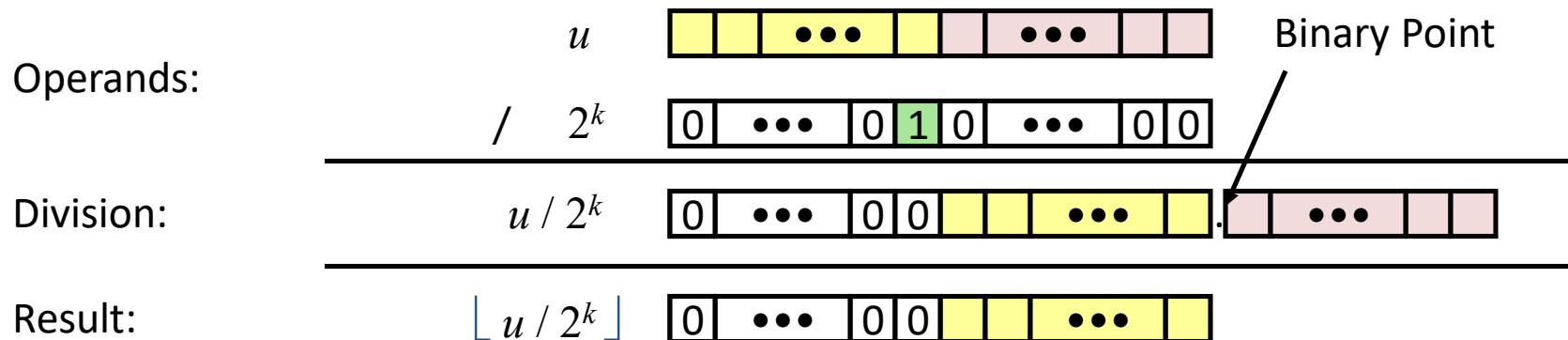
Discard k bits: w bits



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

Unsigned Power-of-2 Divide with Shift

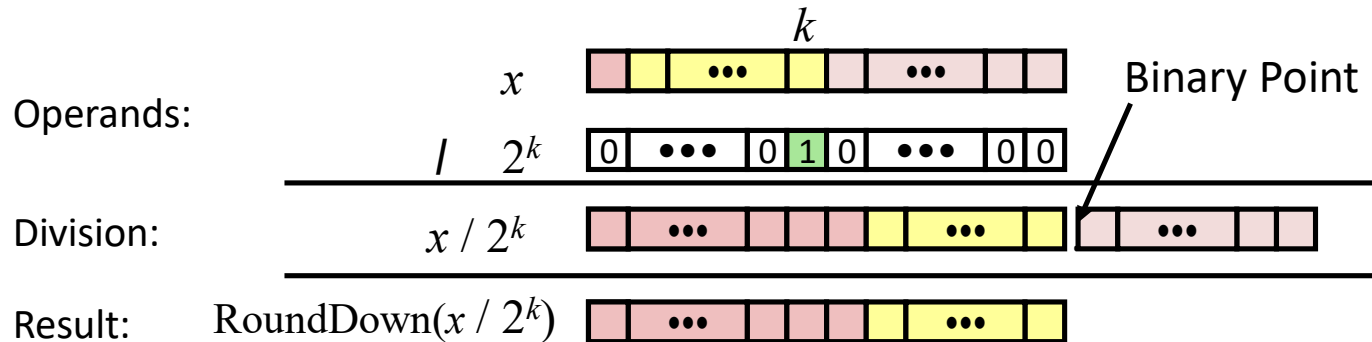
- Quotient of Unsigned by Power of 2
 - $u \gg k$ gives $\lfloor u / 2^k \rfloor$
 - Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
 - $x \gg k$ gives $\lfloor x / 2^k \rfloor$
 - Uses arithmetic shift
 - Rounds wrong direction when $u < 0$

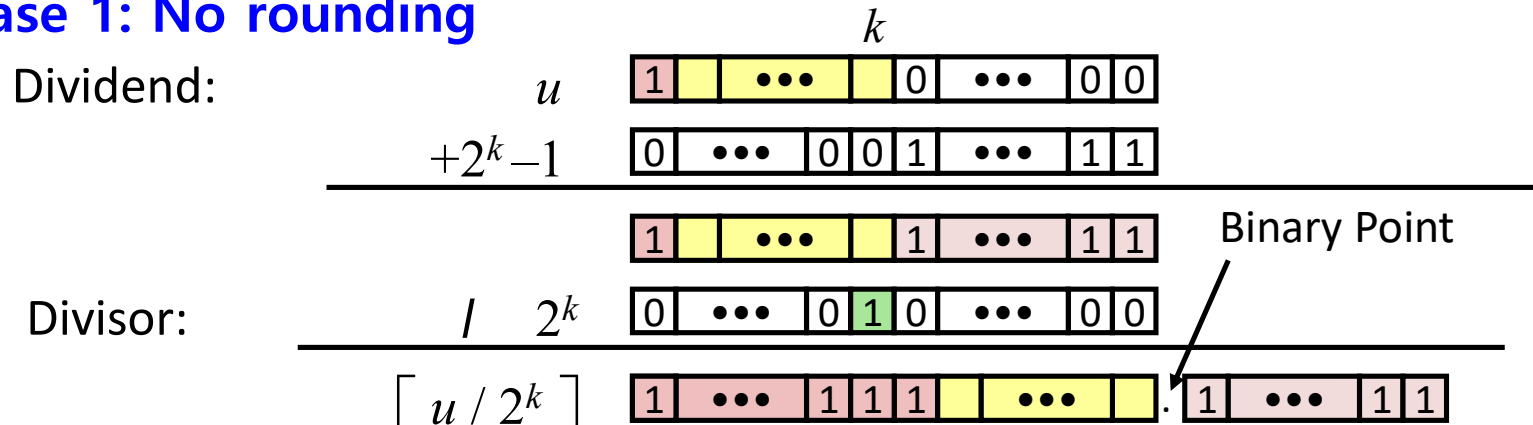


	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
 - Want $\lceil x / 2^k \rceil$ (Round Toward 0)
 - Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0

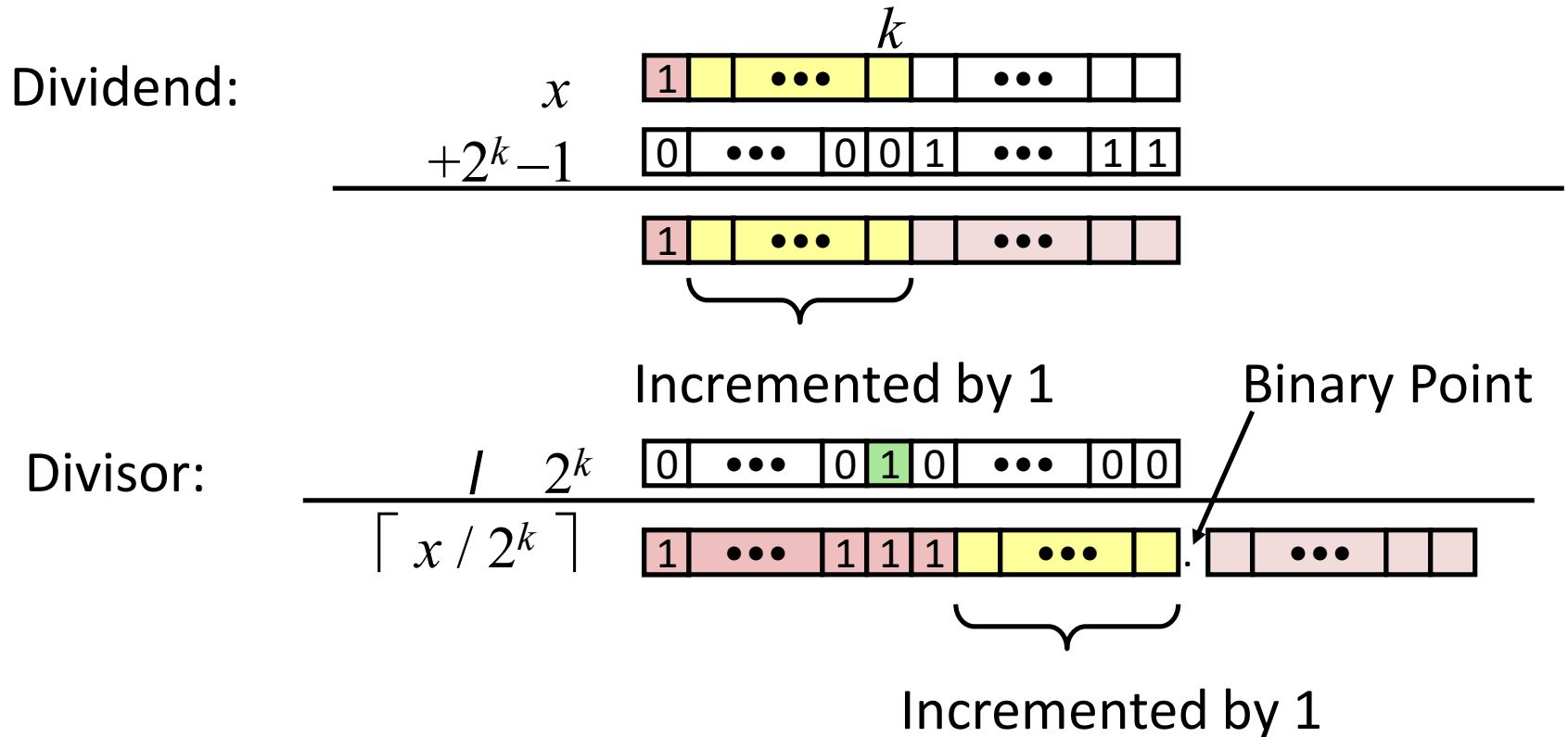
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Negation: Complement & Increment

- Negate through complement and increase

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

- Example

– Observation: $\sim x + x == 1111\dots111 == -1$

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

x = 15213

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

Negation: Complement & Increment

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

x = TMin

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

Canonical counter example

Arithmetic: Basic Rules

- Addition:
 - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
 - Unsigned: addition mod 2^w
 - Mathematical addition + possible subtraction of 2^w
 - Signed: modified addition mod 2^w (result in proper range)
 - Mathematical addition + possible addition or subtraction of 2^w
- Multiplication:
 - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
 - Unsigned: multiplication mod 2^w
 - Signed: modified multiplication mod 2^w (result in proper range)

Why Should I Use Unsigned?

- *Don't* use without understanding implications

- Easy to make mistakes

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

- Can be very subtle

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    . . .
```

Counting Down with Unsigned

- Proper way to use unsigned as loop index

```
unsigned i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- See Robert Seacord, *Secure Coding in C and C++*
 - C Standard guarantees that unsigned addition will behave like modular arithmetic
 - $0 - 1 \rightarrow UMax$
- Even better

```
size_t i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- Data type `size_t` defined as unsigned value with length = word size
- Code will work even if `cnt = UMax`
- What if `cnt` is signed and < 0 ?

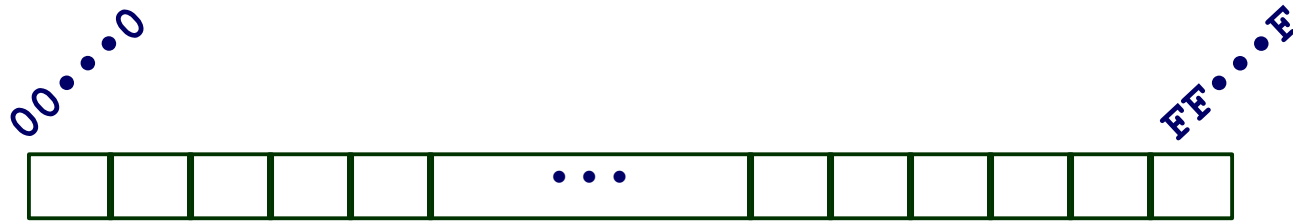
Why Should I Use Unsigned? (cont.)

- *Do Use When Performing Modular Arithmetic*
 - Multiprecision arithmetic
- *Do Use When Using Bits to Represent Sets*
 - Logical right shift, no sign extension
- *Do Use In System Programming*
 - Bit masks, device commands,...

Outline

- Representing information as bits
- Bit-level manipulations
- Integers
- Representations in memory, pointers, strings

Byte-Oriented Memory Organization



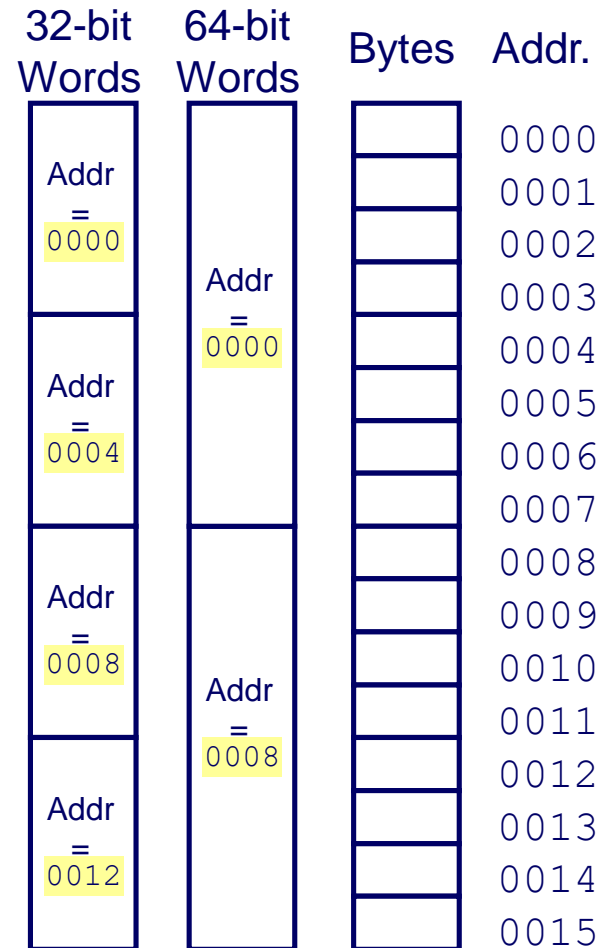
- Programs refer to data by address
 - Conceptually, envision it as a very large array of bytes
 - In reality, it's not, but can think of it that way
 - An address is like an index into that array
 - and, a pointer variable stores an address
- Note: system provides private address spaces to each "process"
 - Think of a process as a program being executed
 - So, a program can clobber its own data, but not that of others

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 18 EB (exabytes) of addressable memory
 - That's 18.4×10^{18}
 - Machines still support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
 - Address of first byte in 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
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	–	–	10/16
pointer	4	8	8

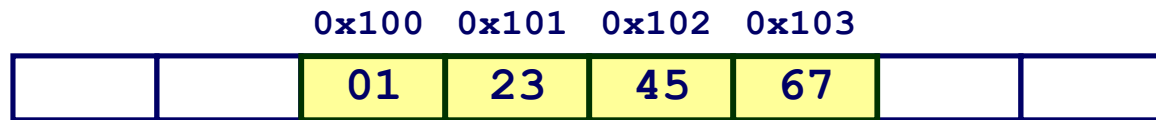
Byte Ordering

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

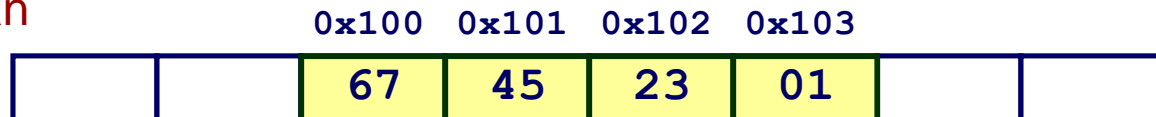
Byte Ordering Example

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

Big Endian



Little Endian



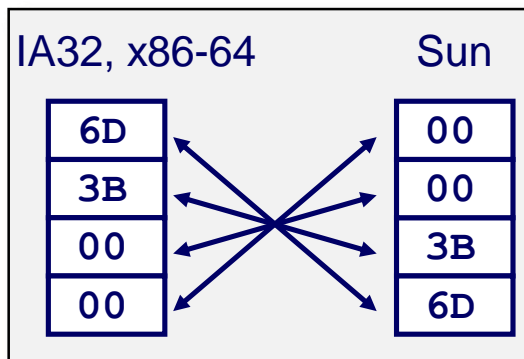
Representing Integers

Decimal: 15213

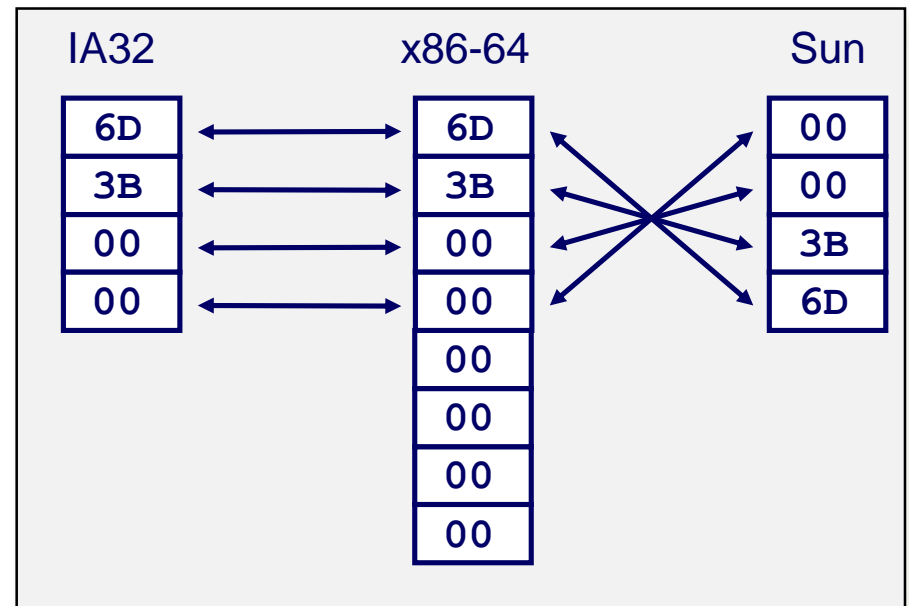
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

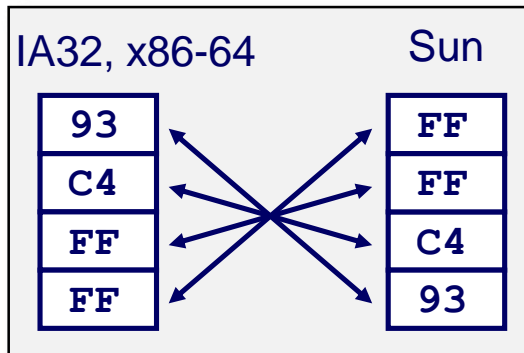
`int A = 15213;`



`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;
```

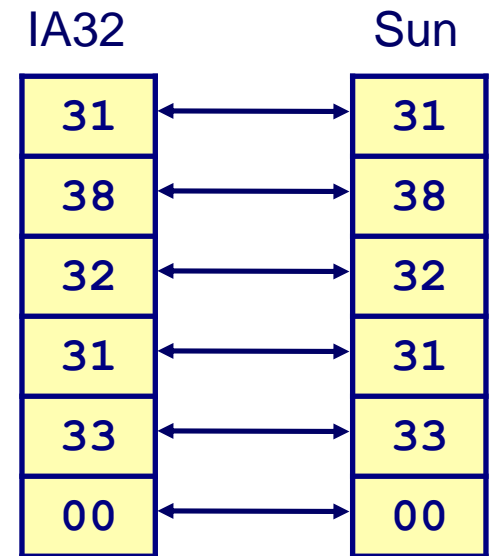
Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

Different compilers & machines assign different locations to objects
Even get different results each time run program

Representing Strings

- 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

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



Reading Byte-Reversed Listings

- Disassembly
 - Text representation of binary machine code
 - Generated by program that reads the machine code
- 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

Integer C Puzzles

Initialization

```
int x = foo();  
int y = bar();  
unsigned ux = x;  
unsigned uy = y;
```

`x < 0` \Rightarrow `((x*2) < 0)`

`ux >= 0`

`x & 7 == 7` \Rightarrow `(x<<30) < 0`

`ux > -1`

`x > y` \Rightarrow `-x < -y`

`x * x >= 0`

`x > 0 && y > 0` \Rightarrow `x + y > 0`

`x >= 0` \Rightarrow `-x <= 0`

`x <= 0` \Rightarrow `-x >= 0`

`(x|-x)>>31 == -1`

`ux >> 3 == ux/8`

`x >> 3 == x/8`

`x & (x-1) != 0`

Example 1

```
#include <stdio.h>
int main ()
{
    unsigned i;

    for (i = 10; i >= 0; i--)
        printf ("%u\n", i);
}
```

Example 2

```
#include <stdio.h>
#define DELTA sizeof(int)
int main ()
{
    int i;

    for (i = 10; i - DELTA >= 0; i -= DELTA)
        printf ("%d\n", i);
}
```

Example 3

```
#include <string.h>
int strlonger (char *s, char *t)
{
    return (strlen(s) - strlen(t)) > 0;
}
```


Example 4

```
int sum_array (int a[], unsigned
len)
{
    int i;
    int result = 0;

    for (i = 0; i <= len - 1; i++)
        result += a[i];

    return result;
}
```

Summary

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

Questions?