

Integers

CENG331 - Computer Organization

Instructor:

Murat Manguoglu (Section 1)

Unless otherwise noted adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

Today: Integers

■ Integers

- **Representation: unsigned and signed**
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

■ Representations in memory, pointers, strings

$X = x_{w-1}, x_{w-2} \dots \dots x_1, x_0$ (w -bits)

What does X represent?

Unsigned binary

$$(x)_{10} = \sum_{i=0}^{w-1} x_i * 2^i$$

smallest $(x)_{10} = 0$

largest $(x)_{10} = 2^w - 1$

$$(x)_{10} \in [0, 2^w - 1]$$

Signed binary (2's complement)

$$(x)_{10} = -x_{w-1} * 2^{w-1} + \sum_{i=0}^{w-2} x_i * 2^i$$

$$10 \dots 00 = 2^w$$

smallest $(x)_{10} = -2^{w-1}$

largest $(x)_{10} = 2^{w-1} - 1$

$$(x)_{10} \in [-2^{w-1}, 2^{w-1} - 1]$$

Sign-magnitude representation

$\begin{Bmatrix} 1 \\ 0 \end{Bmatrix} \times \times \dots \times$

$$(X)_{10} = \pm \left(\sum_{i=0}^{w-2} x_i * 2^i \right)$$

$$(X)_{10} \in [-(2^{w-1}-1), +(2^{w-1}-1)]$$

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	<u>-15213</u>	C4 93	<u>1</u> .000100 10010011

■ Sign Bit

- For 2's complement, most significant bit indicates sign
 - 0 for nonnegative
 - 1 for negative

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

■ Other Values

- Minus 1
111...1

Values for $W = 16$

	Decimal	Hex	Binary
UMax	<u>65535</u>	FF FF	11111111 11111111
→ TMax	32767	7F FF	01111111 11111111
→ TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808



■ Observations

- $|TMin| = Tmax + 1$
 - Asymmetric range
- $UMax = 2 * Tmax + 1$

■ C Programming

- `#include <limits.h>`
- Declares constants, e.g.,
 - ULONG_MAX
 - LONG_MAX
 - LONG_MIN
- Values platform specific

Unsigned & Signed Numeric Values

X	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

■ Equivalence

- Same encodings for nonnegative values

■ Uniqueness

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

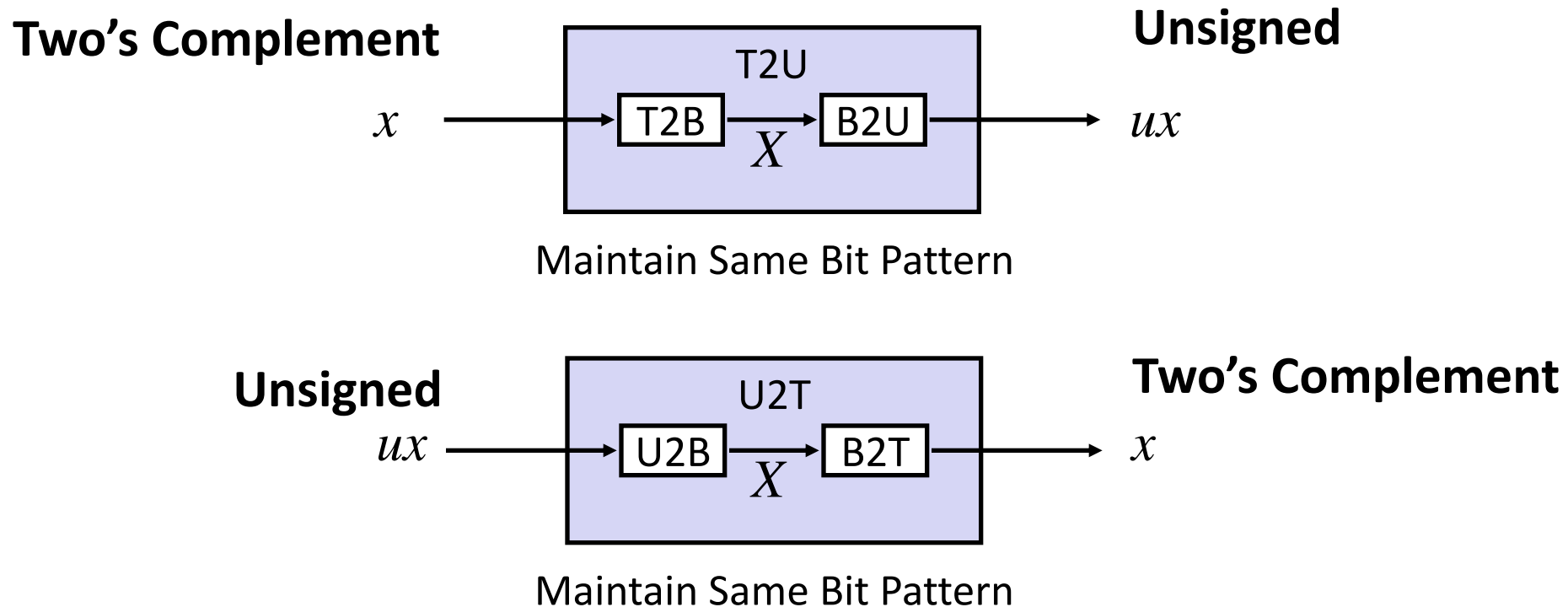
■ \Rightarrow Can Invert Mappings

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

Today: Bits, Bytes, and Integers

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

Mapping Between Signed & Unsigned



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

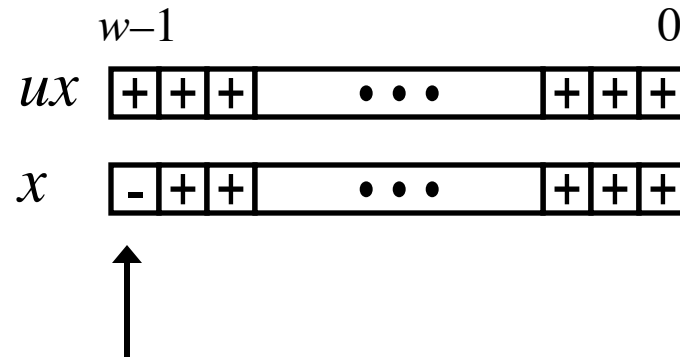
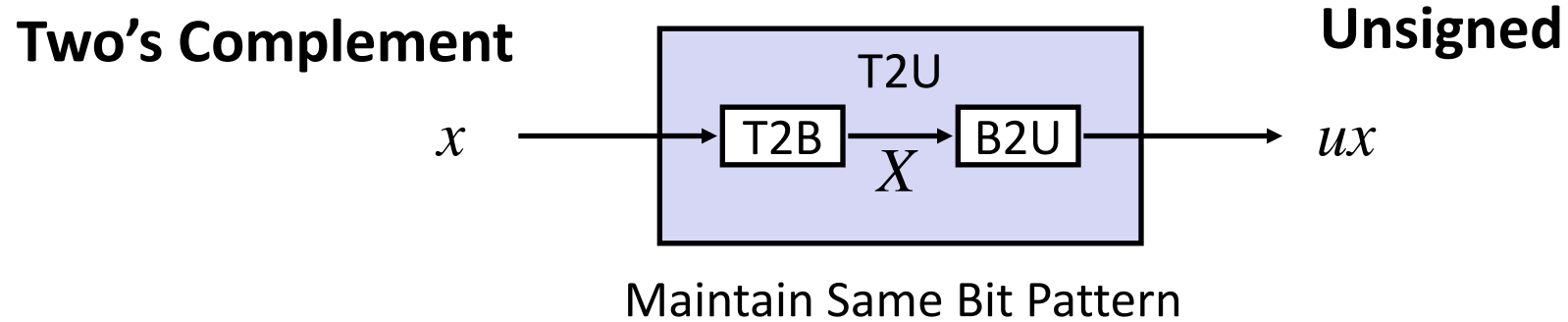
Mapping Signed \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0	<div><div>T2U</div><div>U2T</div></div>	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Mapping Signed \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0	$=$	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
<u>1</u> 000	-8	$+/- 16$	8
<u>1</u> 001	-7		9
<u>1</u> 010	-6		10
<u>1</u> 011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Relation between Signed & Unsigned

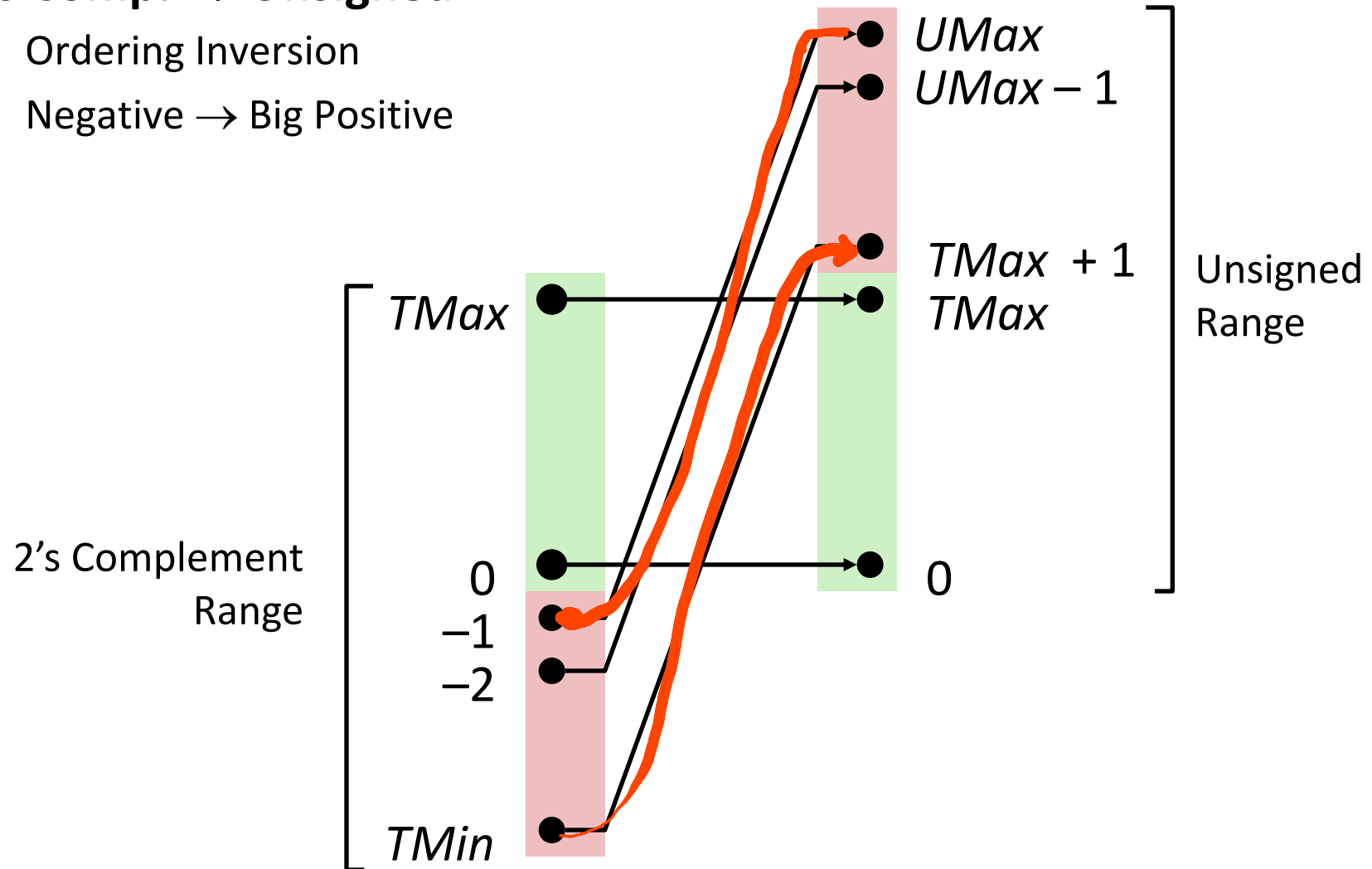


Large negative weight
becomes
Large positive weight

Conversion Visualized

■ 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



|||||. . . |

Signed vs. Unsigned in C

1 2 3 4 1 U

■ 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
<u>0</u>	<u>0U</u>	$==$	unsigned
-1	<u>0</u>	$<$	signed
<u>-1</u>	<u>0U</u>	$>$	unsigned
<u>2147483647</u>	<u>-2147483647-1</u>	$>$	signed
<u>2147483647U</u>	<u>-2147483647-1</u>	$<$	unsigned
-1	-2	$>$	signed
(unsigned)-1	-2	$>$	unsigned
2147483647	2147483648U	$<$	unsigned
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`!!

Today: Integers

■ Integers

- Representation: unsigned and signed
- Conversion, casting
- **Expanding, truncating**
- Addition, negation, multiplication, shifting
- Summary

■ Representations in memory, pointers, strings

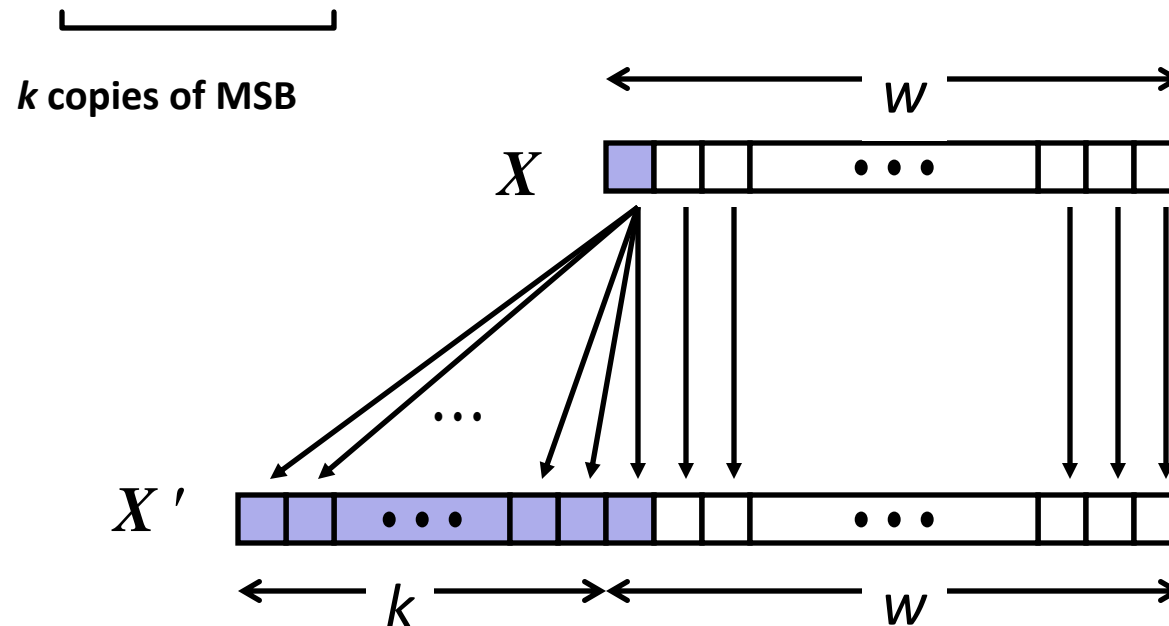
Sign Extension

■ Task:

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

■ Rule:

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



Sign Extension 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

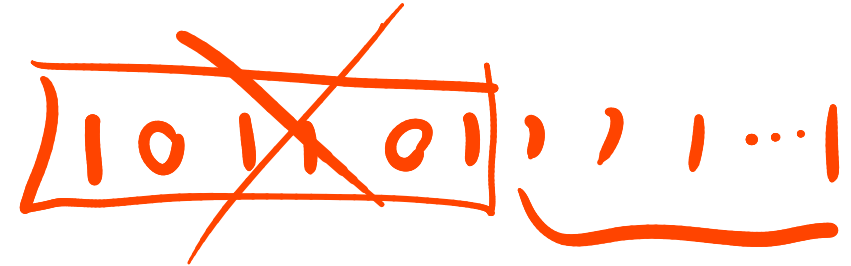
- Converting from smaller to larger integer data type
- C automatically performs sign extension

Summary:

Expanding, Truncating: Basic Rules

■ Expanding (e.g., short int to int)

- Unsigned: zeros added
- Signed: sign extension
- Both yield expected result



64 → 8

■ 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
 - For others?

Today: Integers

■ Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- **Addition, negation, multiplication, shifting**

■ Representations in memory, pointers, strings




Unsigned Addition

Operands: w bits

u 

+ v 

True Sum: $w+1$ bits

$u + v$ 

Discard Carry: w bits

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

■ Standard Addition Function

- Ignores carry output

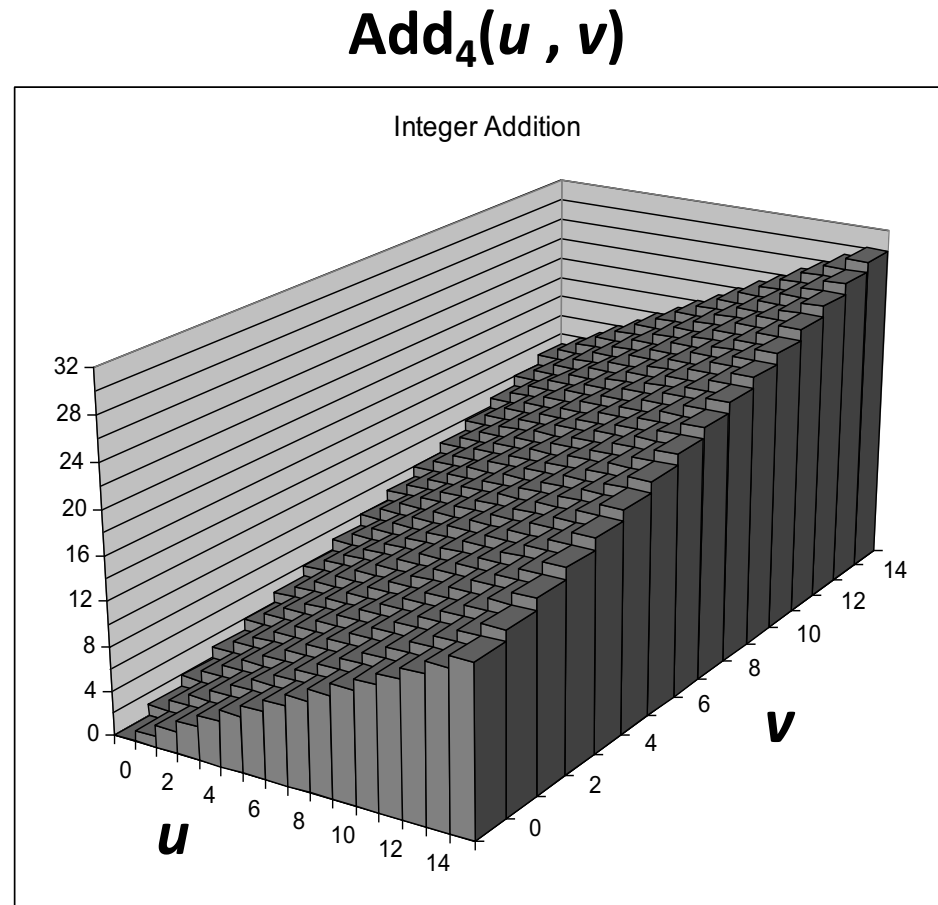
■ Implements Modular Arithmetic

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

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

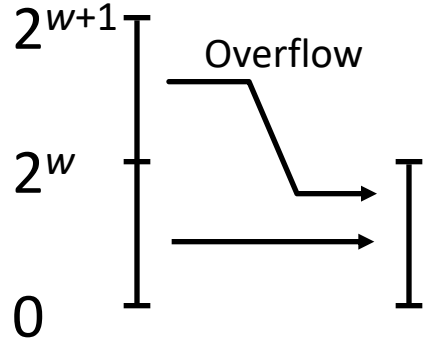


Visualizing Unsigned Addition

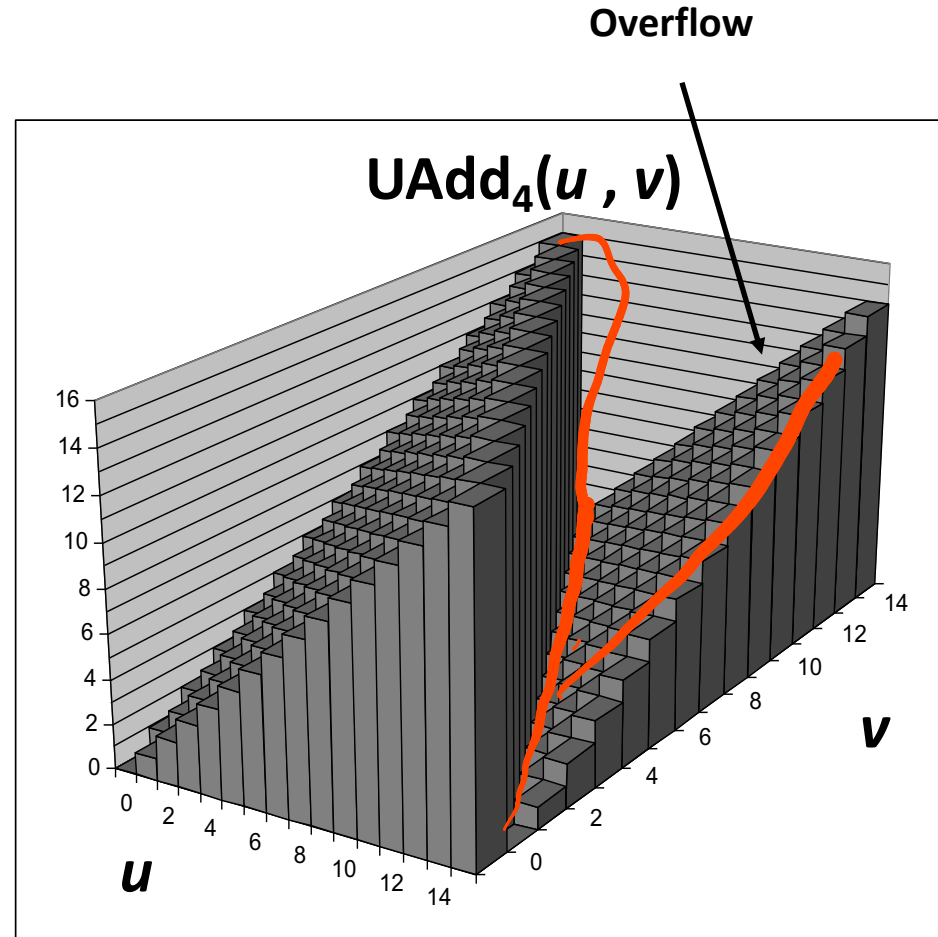
■ Wraps Around

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

True Sum



Modular Sum

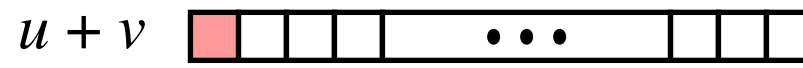


Two's Complement Addition

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);
```

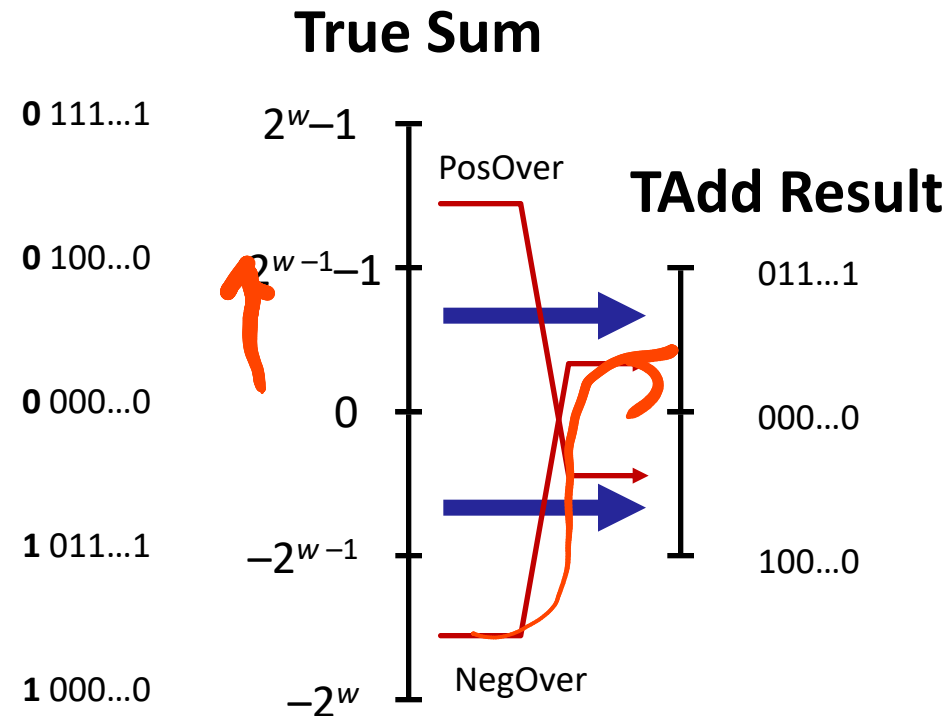
```
t = u + v
```

- Will give `s == t`

TAdd Overflow

■ Functionality

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



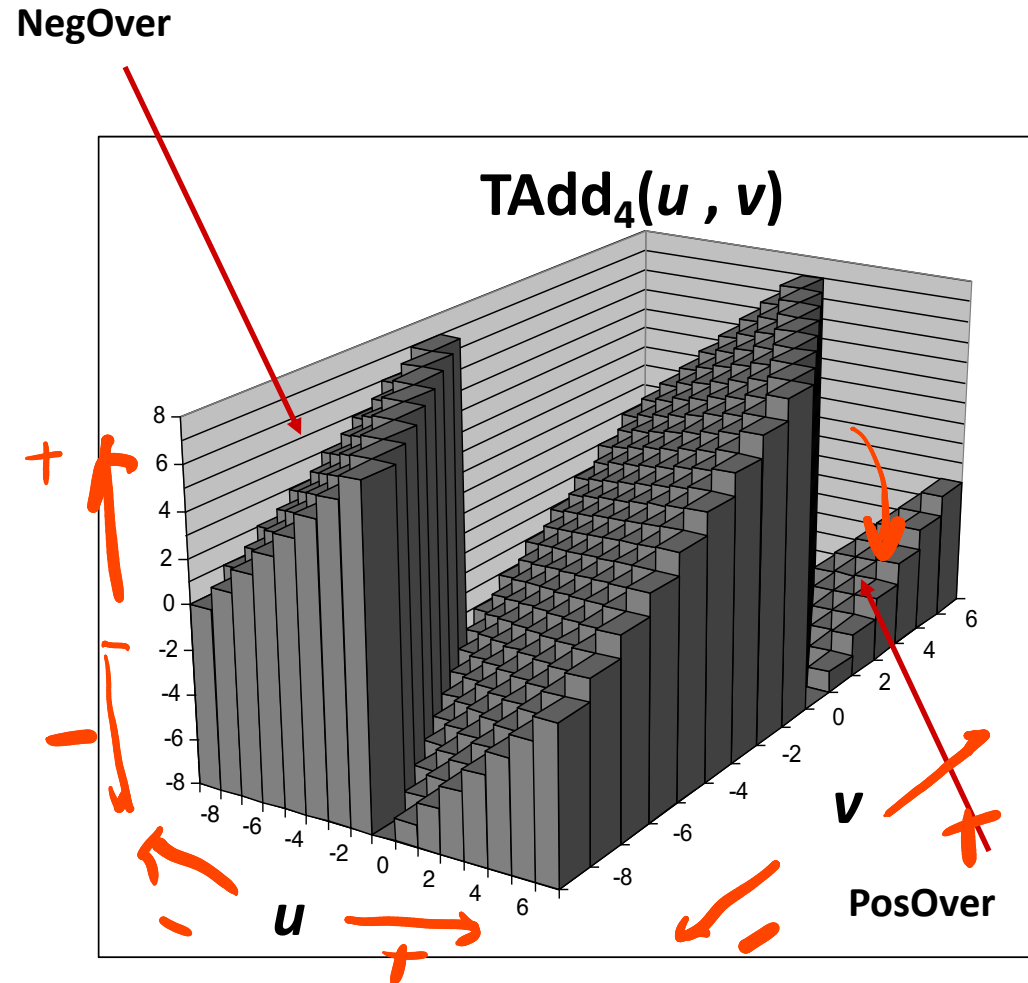
Visualizing 2's Complement Addition

■ Values

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

■ Wraps Around

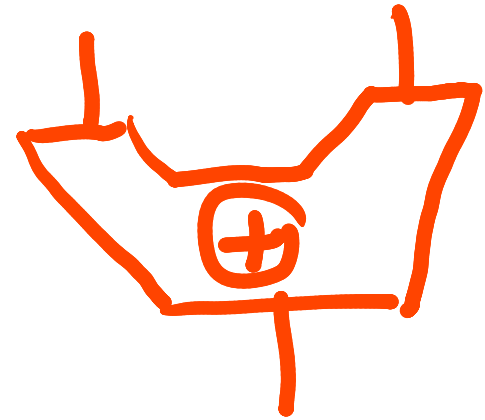
- If $\text{sum} \geq 2^{w-1}$
 - Becomes negative
 - At most once
- If $\text{sum} < -2^{w-1}$
 - Becomes positive
 - At most once



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

$1001 \leftarrow \text{multiplicand}$
 $\times 1101 \leftarrow \text{multiplier}$
 $\quad \quad \quad (n\text{-digits})$



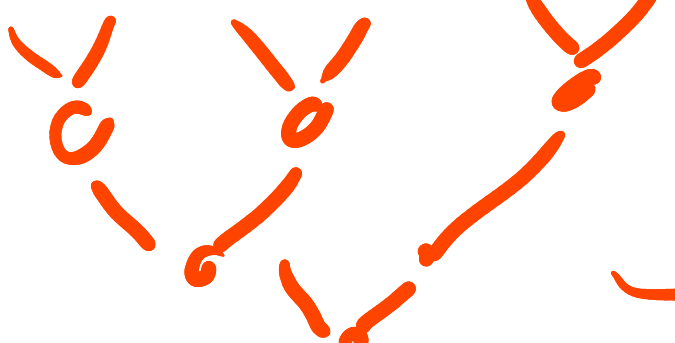
$1^{\text{st}} \rightarrow 1001 - n_1$
 $2^{\text{nd}} \rightarrow 0000 - n_2$
 $3^{\text{rd}} \rightarrow 1001 - n_3$
 $4^{\text{th}} \rightarrow 1001 - n_4$

$O(n)$ additions

~~1110101~~

$n_1 + n_2 + n_3 + n_4$

0000...0



$O(n)$

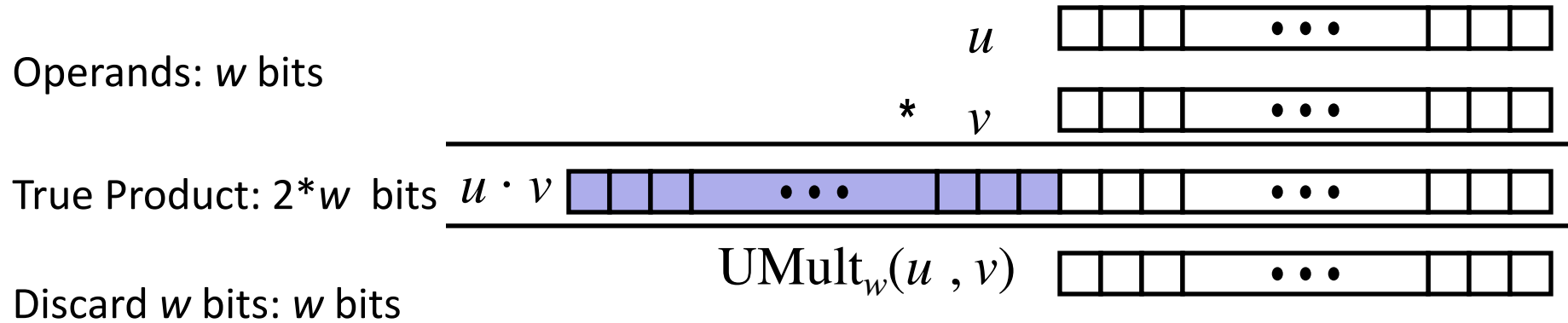
$+$
 n'

$+$
 n''

$+$
 N

$t_{+/-} < t_* < t_+$

Unsigned Multiplication in C



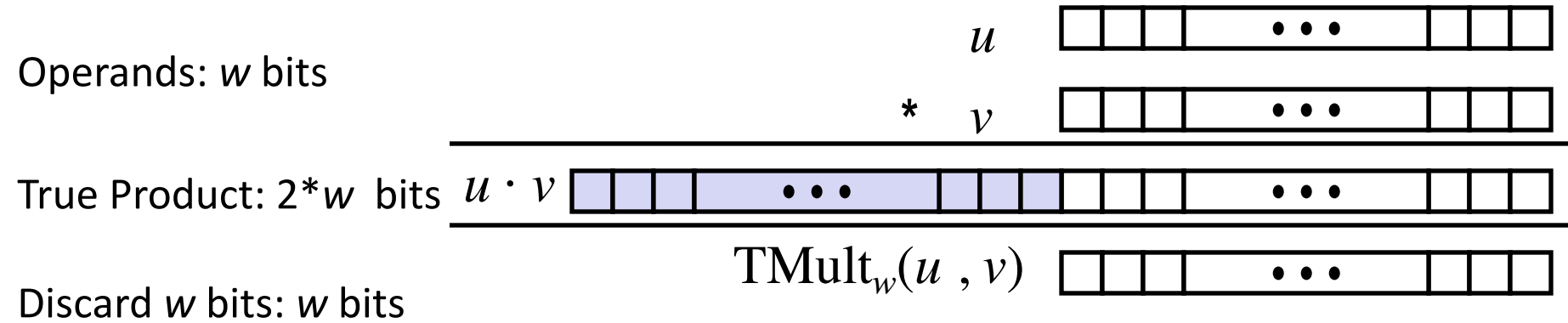
- **Standard Multiplication Function**

- Ignores high order w bits

- **Implements Modular Arithmetic**

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

Signed Multiplication in C



■ Standard Multiplication Function

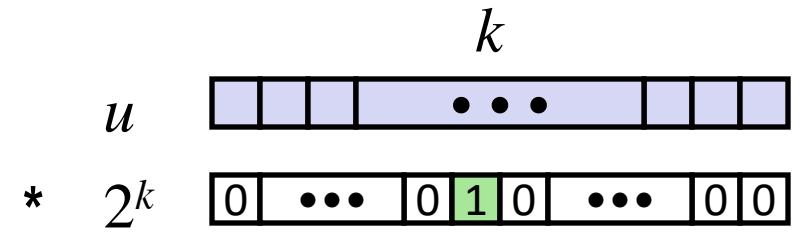
- Ignores high order w bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

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 $u \cdot 2^k$

Discard k bits: w bits

UMult _{w} (u , 2^k)
TMult _{w} (u , 2^k)

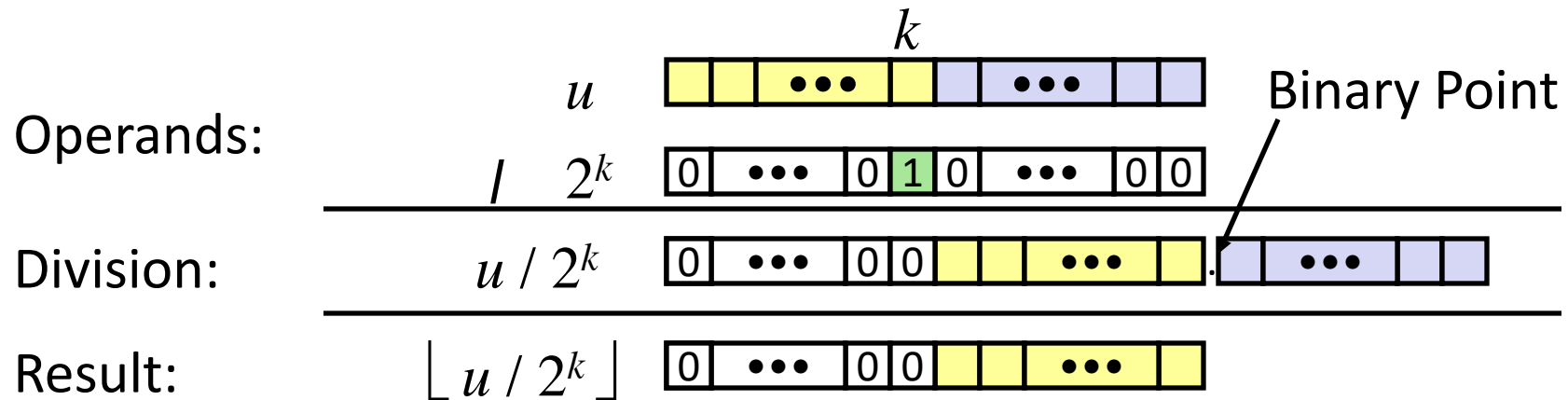
■ Examples

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

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

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

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? (cont.)

- ***Do Use When Performing Modular Arithmetic***
 - Multiprecision arithmetic
- ***Do Use When Using Bits to Represent Sets***
 - Logical right shift, no sign extension

Thank you!