



# Bits, Bytes, and Integers – Part 2

15-213: Introduction to Computer Systems  
3<sup>rd</sup> Lecture, Sept. 5, 2017

**Today's Instructor:**  
Randy Bryant

# Assignment Announcements

## ■ Lab 0 available via course web page and Autolab.

- Due Thurs. Sept. 7, 11:59pm
- No grace days
- No late submissions
- Just do it!

## ■ Lab 1 available via TPZ and Autolab

- Due Thurs, Sept. 14, 11:55pm
- Read instructions carefully: writeup, bits.c, tests.c
  - Quirky software infrastructure
- Based on lectures 2, 3, and 4 (CS:APP Chapter 2)
- After today's lecture you will know everything for the integer problems
- Floating point covered Thursday Sept. 7

# Summary From Last Lecture

- **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**

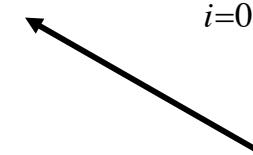
# Encoding Integers

## Unsigned

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

## Two's Complement

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



Sign Bit

## Two's Complement Examples (w = 5)

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

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

# 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

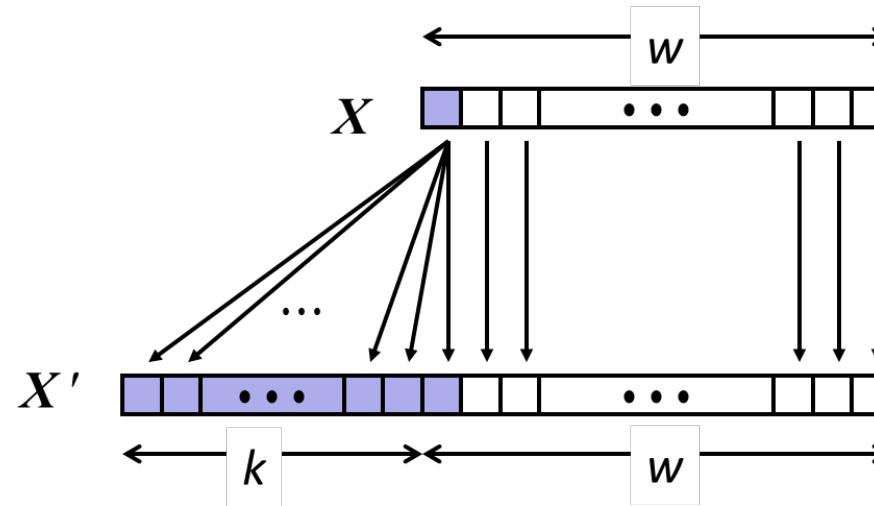
- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

## ■ Expression containing signed and unsigned int:

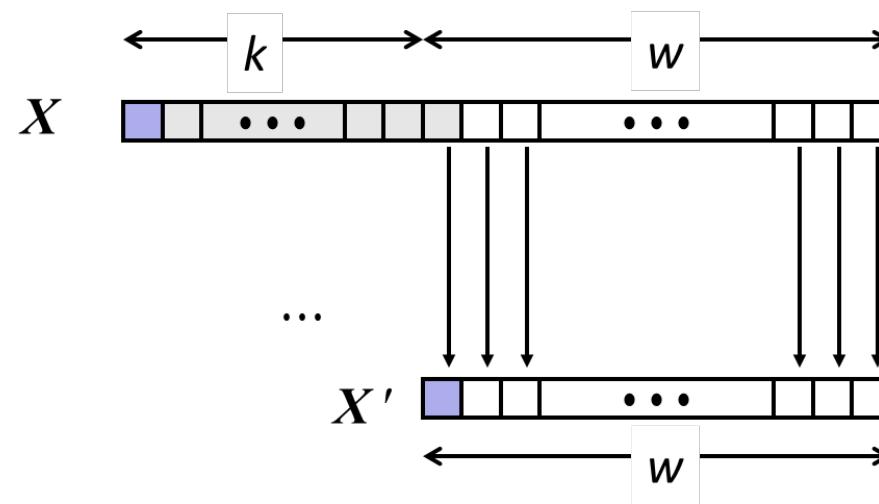
int is cast to unsigned

# Sign Extension and Truncation

## ■ Sign Extension



## ■ Truncation



# Today: Bits, Bytes, and Integers

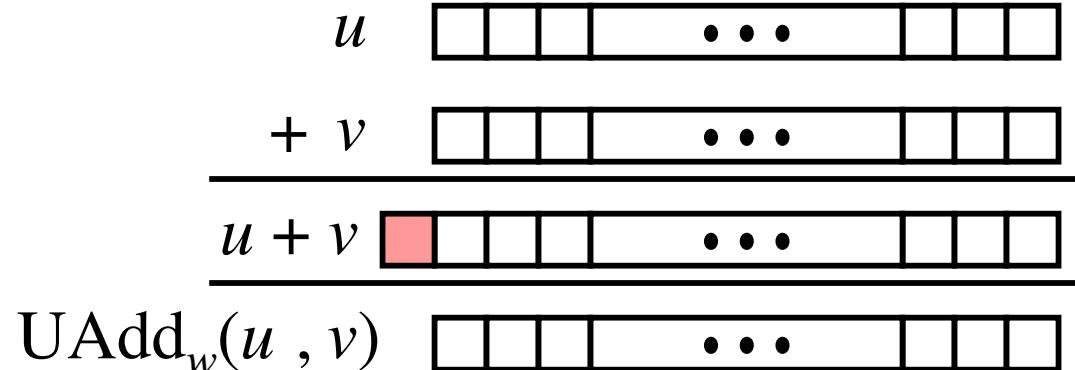
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

# 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	223
	+ 1101 0101	+ D5	+ 213
	<hr/>	<hr/>	<hr/>
	1 1011 1110	1BE	446
	<hr/>	<hr/>	<hr/>
	1011 1110	BE	190

Hex    Decimal    Binary

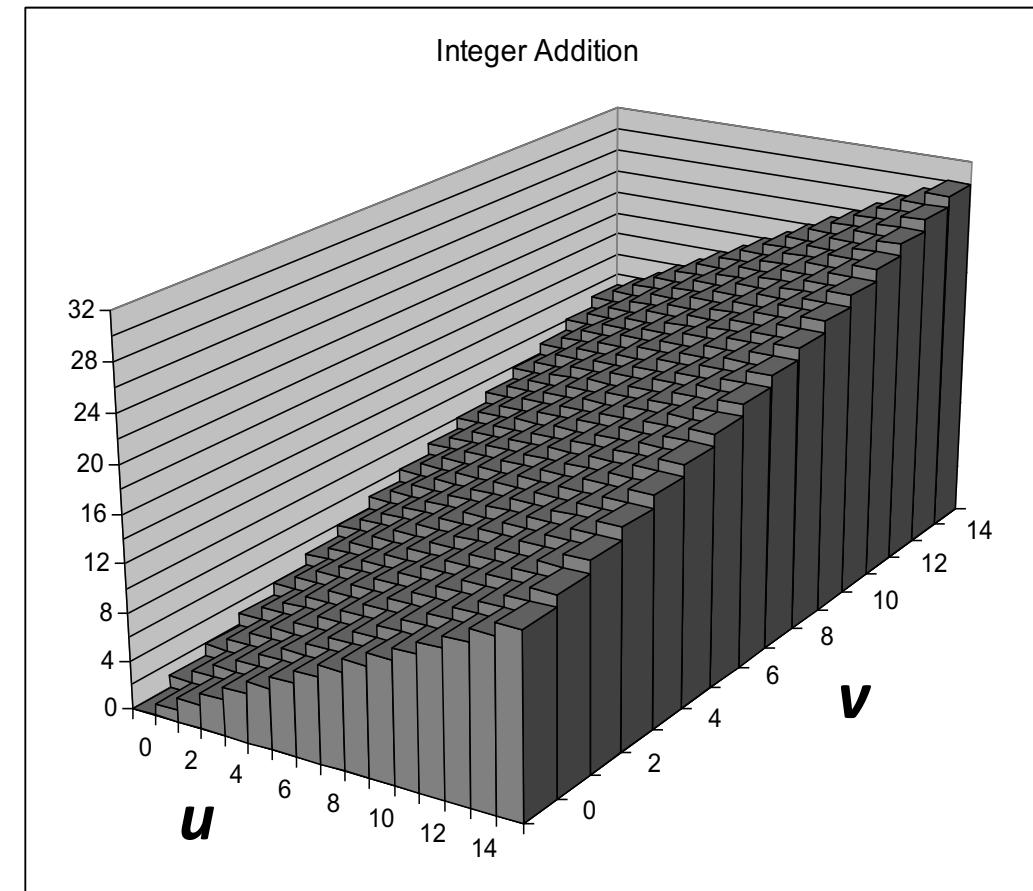
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# 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

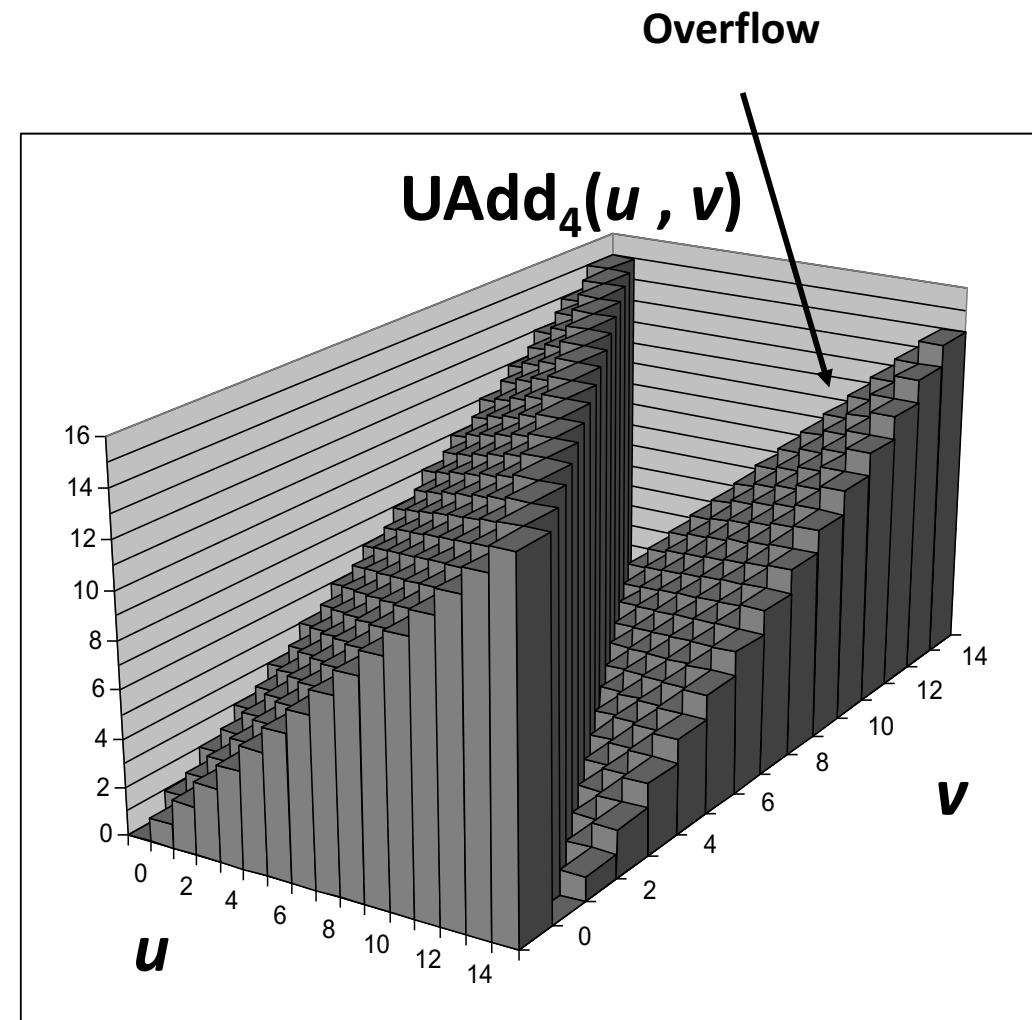
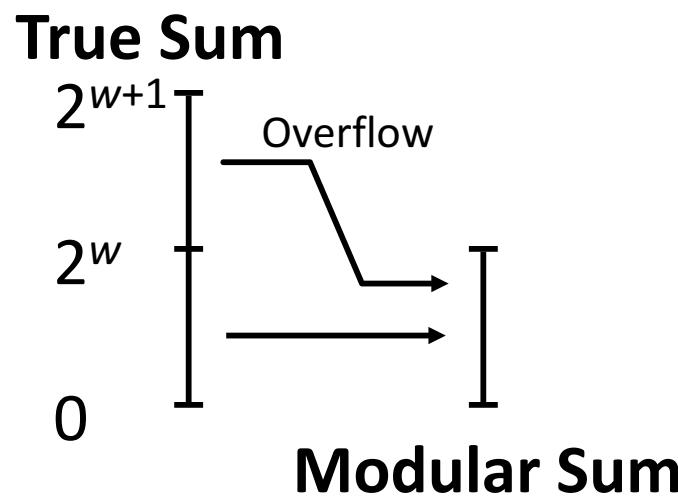
$\text{Add}_4(u, v)$



# Visualizing Unsigned Addition

## Wraps Around

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

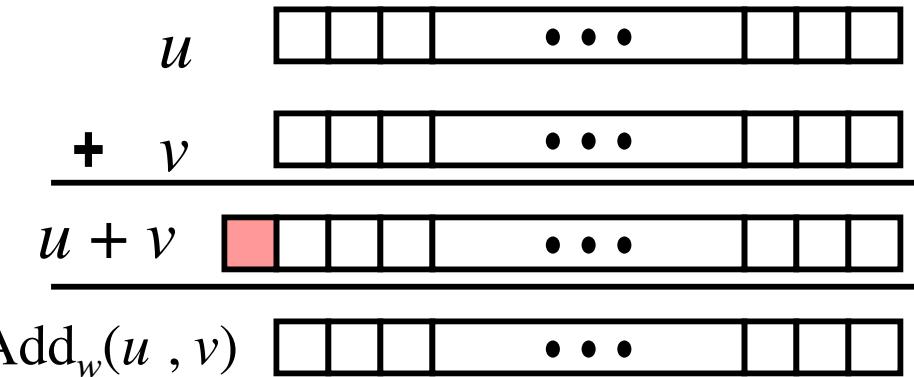


# Two's Complement Addition

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  $s == t$

$$\begin{array}{rccccc}
 & 1110 & 1001 & & \text{E9} & & -23 \\
 & + & 1101 & 0101 & + & \text{D5} & + & -43 \\
 \hline
 & \textcolor{red}{1} & 1011 & 1110 & \textcolor{red}{1BE} & & \textcolor{red}{446} \\
 \hline
 & 1011 & 1110 & & \textcolor{red}{BE} & & -66
 \end{array}$$

# 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

$2^{w-1}$

PosOver

$2^{w-1}-1$

0

$-2^{w-1}$

$-2^w$

## TAdd Result

011...1

000...0

100...0

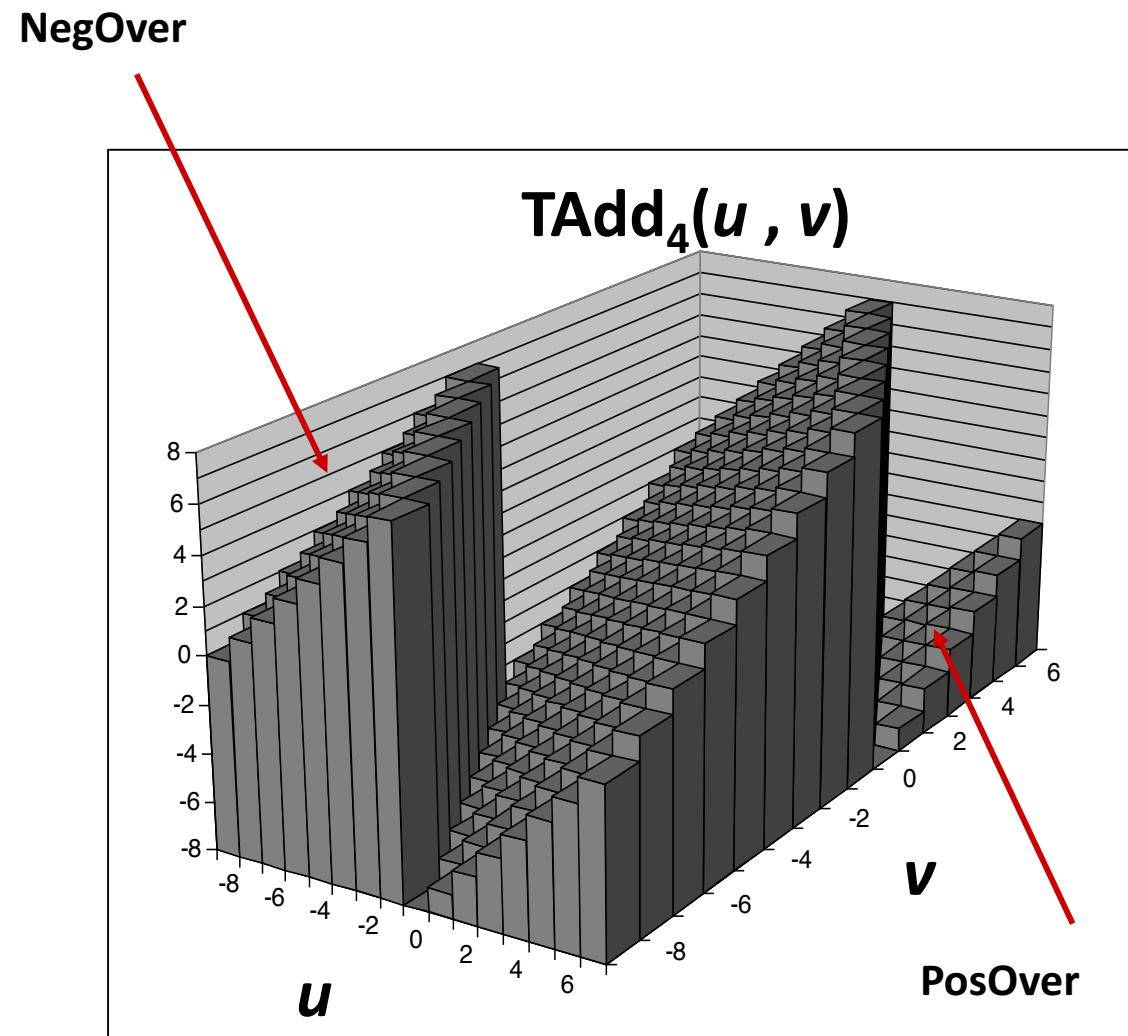
# Visualizing 2's Complement Addition

## ■ Values

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

## ■ Wraps Around

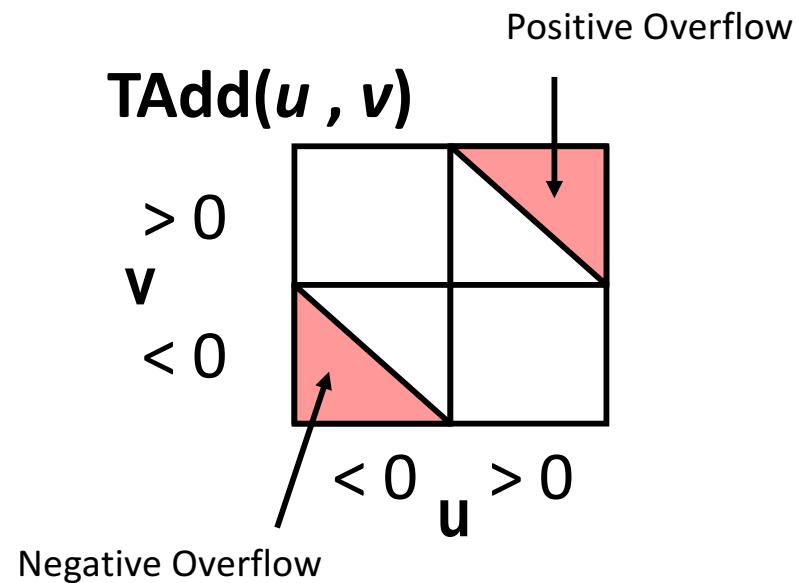
- If  $\text{sum} \geq 2^{w-1}$ 
  - Becomes negative
  - At most once
- If  $\text{sum} < -2^{w-1}$ 
  - Becomes positive
  - At most once



# Characterizing TAdd

## ■ Functionality

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



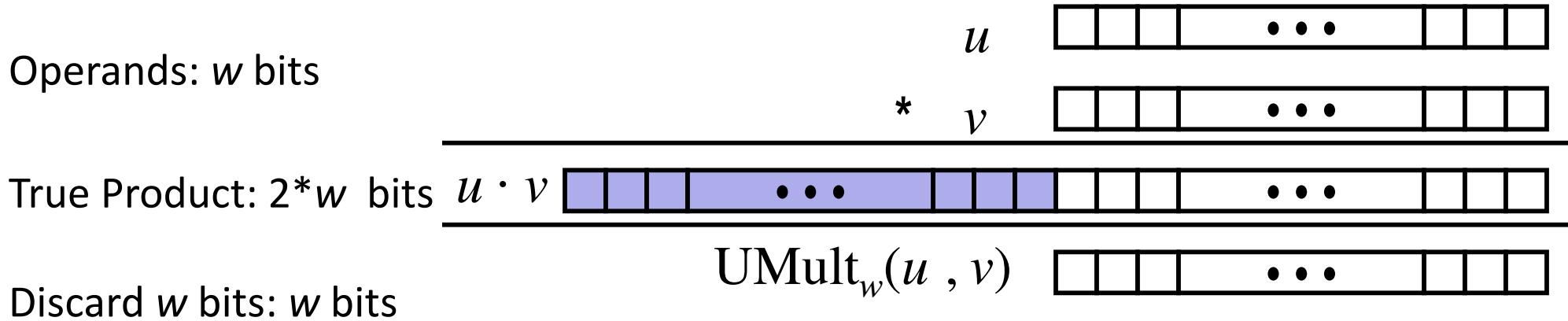
$$TAdd_w(u, v) = \begin{cases} u + v + 2^w & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

# 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

Operands:  $w$  bits



## ■ Standard Multiplication Function

- Ignores high order  $w$  bits

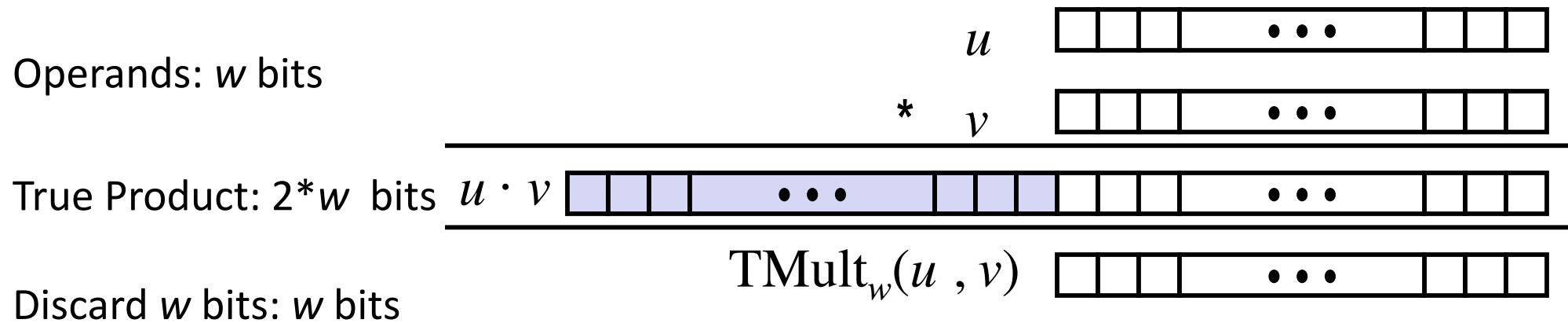
## ■ Implements Modular Arithmetic

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

$$\begin{array}{r}
 & 1110 & 1001 & & \text{E9} & & 223 \\
 * & 1101 & 0101 & & * & \text{D5} & * & 213 \\
 \hline
 & \text{1100} & 0001 & 1101 & 1101 & \text{C1DD} & & 47499 \\
 \hline
 & & & & 1101 & 1101 & & \\
 & & & & & & \text{DD} & \\
 \hline
 & & & & & & & 221
 \end{array}$$

# Signed Multiplication in C

Operands:  $w$  bits



## ■ Standard Multiplication Function

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

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

# Power-of-2 Multiply with Shift

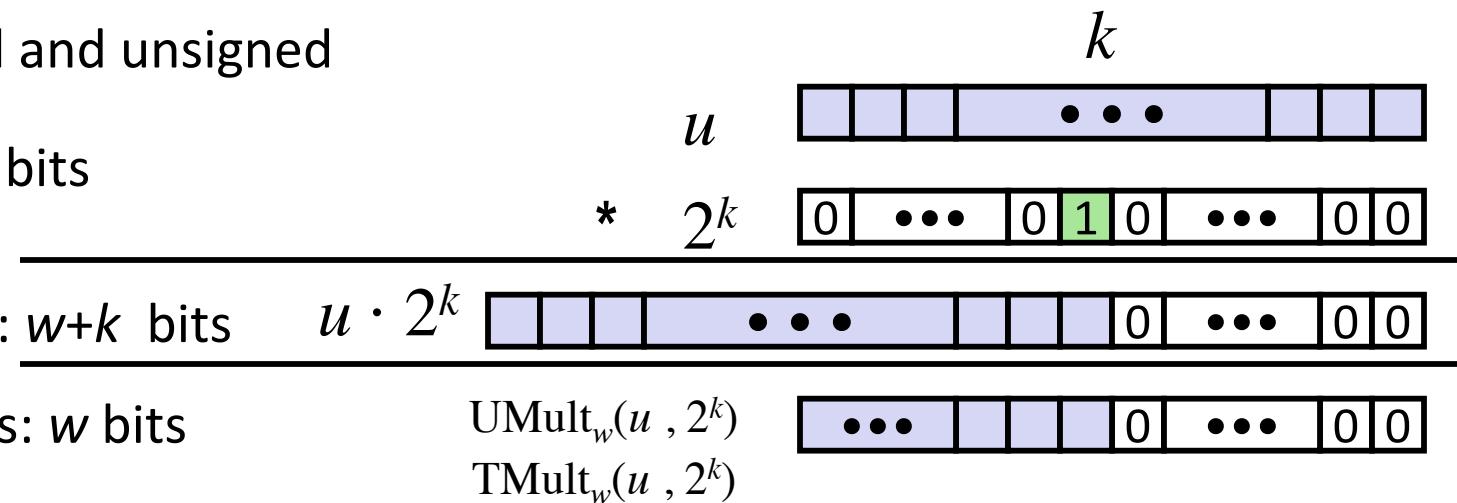
## ■ Operation

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

Operands:  $w$  bits

True Product:  $w+k$  bits

Discard  $k$  bits:  $w$  bits



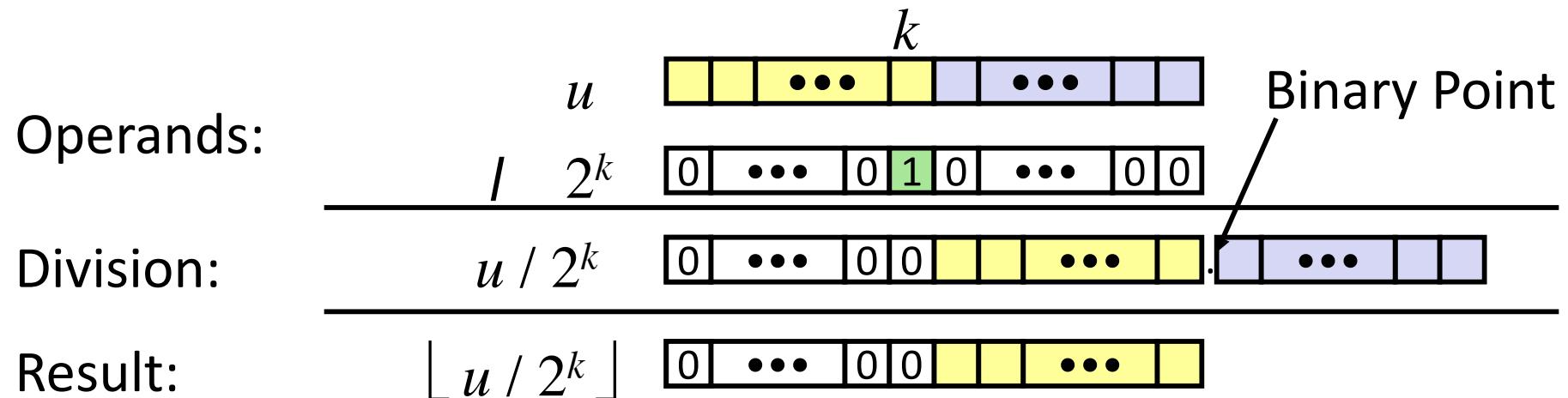
## ■ Examples

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

# 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

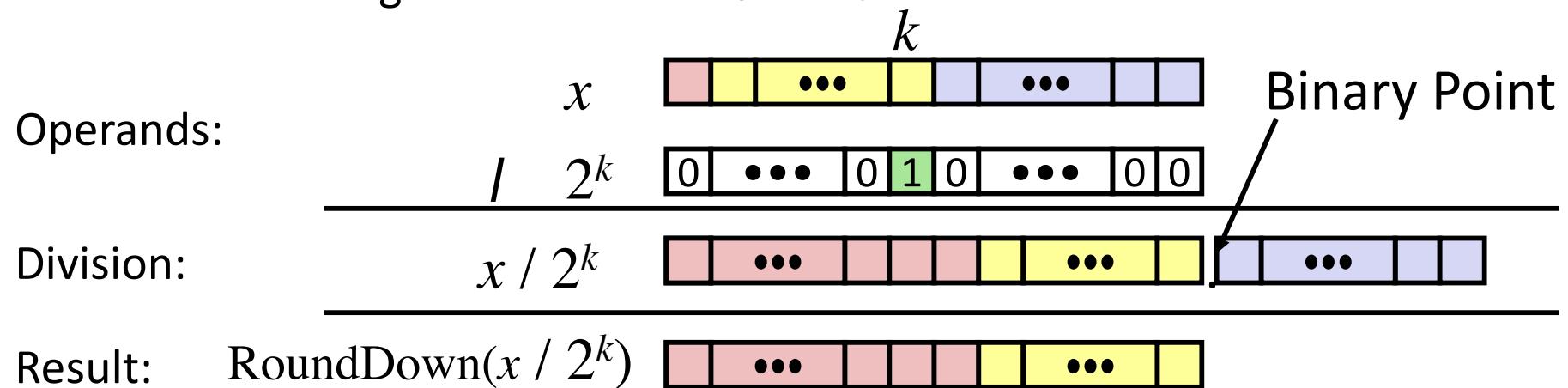


	Division	Computed	Hex	Binary
<b>x</b>	<b>15213</b>	<b>15213</b>	<b>3B 6D</b>	<b>00111011 01101101</b>
<b>x &gt;&gt; 1</b>	<b>7606.5</b>	<b>7606</b>	<b>1D B6</b>	<b>00011101 10110110</b>
<b>x &gt;&gt; 4</b>	<b>950.8125</b>	<b>950</b>	<b>03 B6</b>	<b>00000011 10110110</b>
<b>x &gt;&gt; 8</b>	<b>59.4257813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

# 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

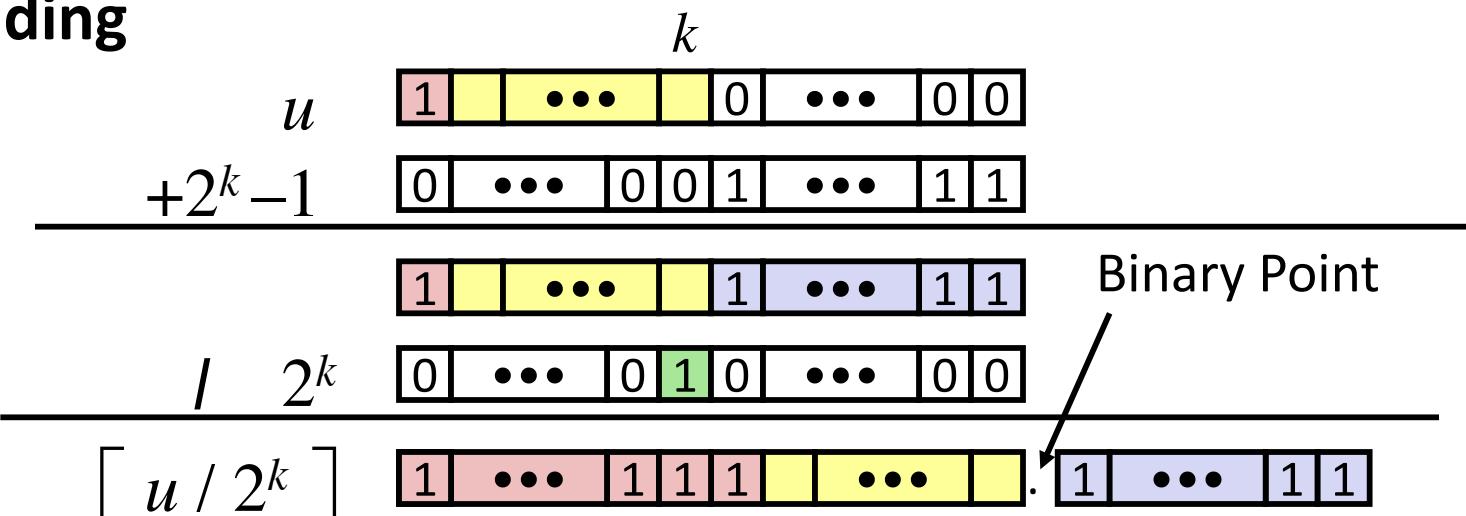
# Correct Power-of-2 Divide

## ■ 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<<k)-1) >> k`
  - Biases dividend toward 0

## Case 1: No rounding

Dividend:



***Biasing has no effect***

# Correct Power-of-2 Divide (Cont.)

## Case 2: Rounding

Dividend:

$$\begin{array}{r}
 x \\
 +2^k - 1 \\
 \hline
 \end{array}$$

$k$

Divisor:

$$\begin{array}{r}
 / 2^k \\
 \hline
 \lceil x / 2^k \rceil
 \end{array}$$

Incremented by 1

Binary Point

Incremented by 1

***Biasing adds 1 to final result***

# Negation: Complement & Increment

- Negate through complement and increase

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

- Example

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

$$\begin{array}{r}
 x \quad \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \\
 + \quad \sim x \quad \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \\
 \hline
 -1 \quad \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1}
 \end{array}$$

$$x = 15213$$

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

# Complement & Increment Examples

**x = 0**

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

**x = TMin**

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

## Canonical counter example

# 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?

## ■ *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  $< 0$ ?

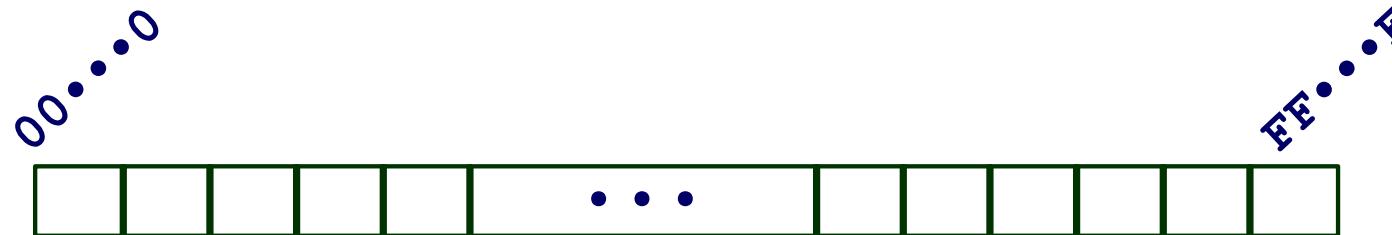
# Why Should I Use Unsigned? (cont.)

- ***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,...

# 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

# 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

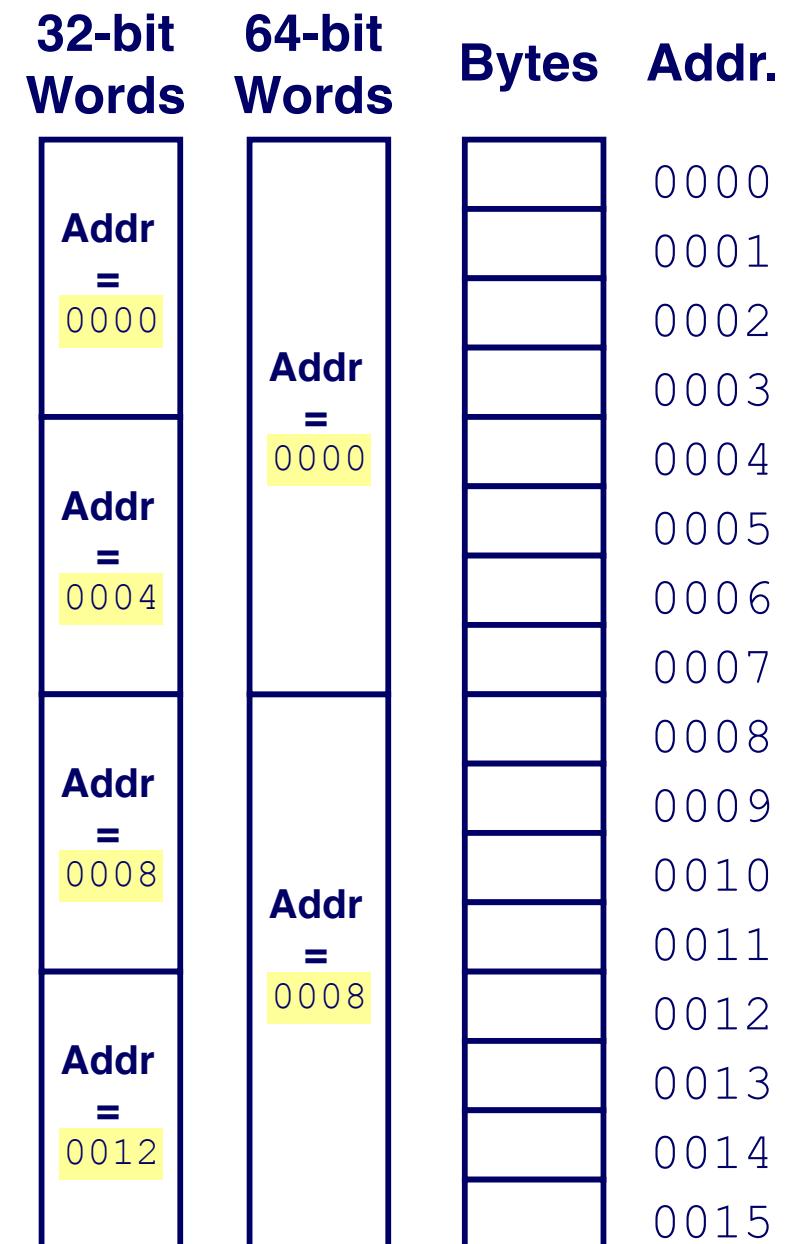
# Machine Words

- 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  $18.4 \times 10^{18}$
  - 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
<b>char</b>	1	1	1
<b>short</b>	2	2	2
<b>int</b>	4	4	4
<b>long</b>	4	8	8
<b>float</b>	4	4	4
<b>double</b>	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 (Oracle SPARC), PPC Mac, *Internet*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address

# Byte Ordering Example

## ■ Example

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

### BigEndian

0x100 0x101 0x102 0x103



### LittleEndian

0x100 0x101 0x102 0x103



# Representing Integers

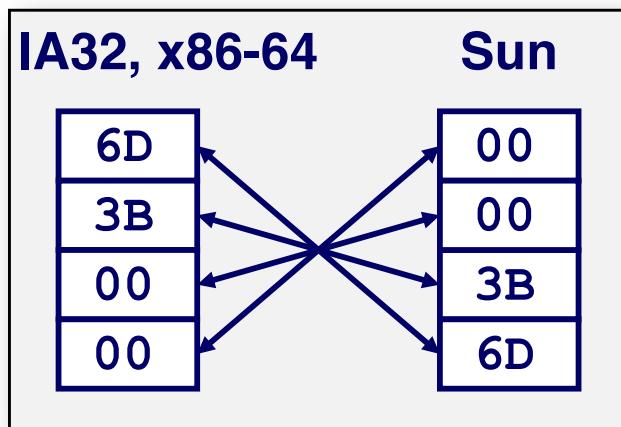
Decimal: 15213

Binary: 0011 1011 0110 1101

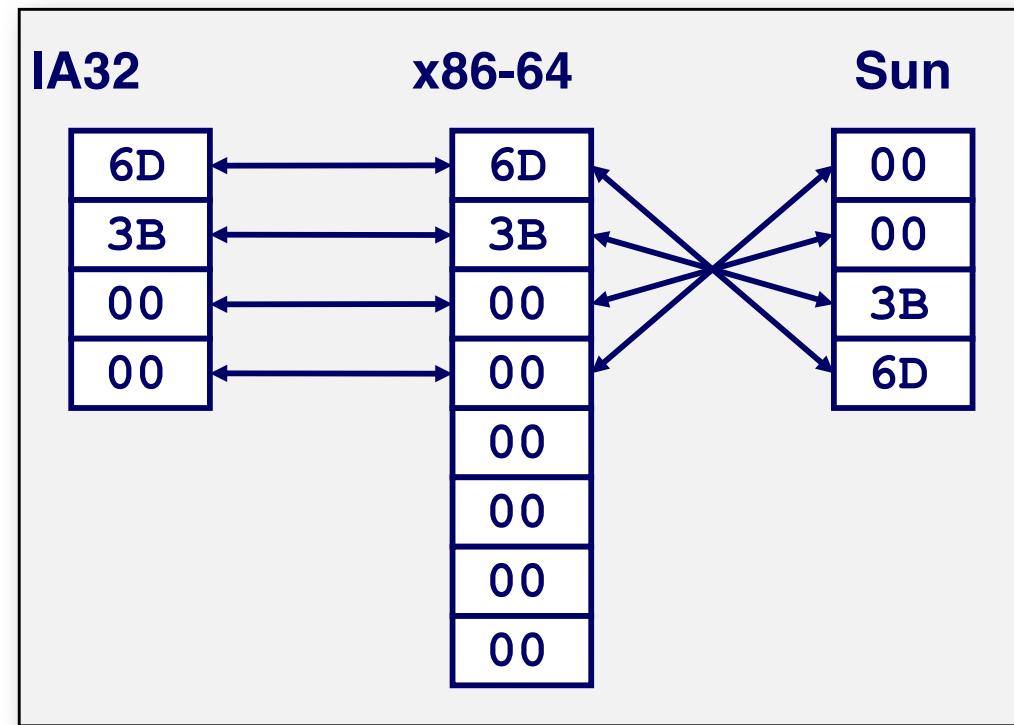
Hex: 3 B 6 D

`int A = 15213;`

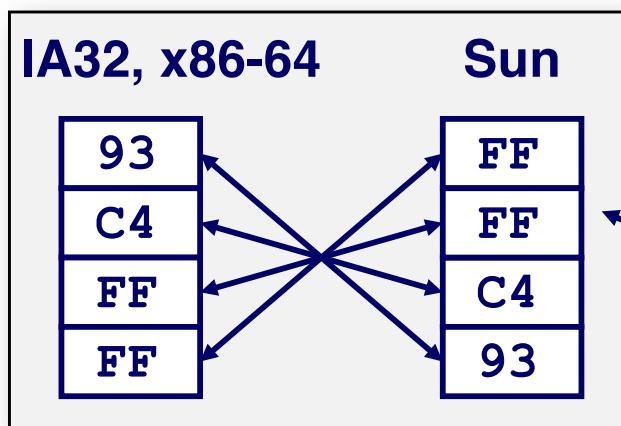
Increasing addresses ↓



`long int C = 15213;`



`int B = -15213;`



Two's complement representation

# Examining Data Representations

## ■ Code to Print Byte Representation of Data

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

```
typedef unsigned char *pointer;

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

### Printf directives:

%p: Print pointer  
%x: Print Hexadecimal

# show\_bytes Execution Example

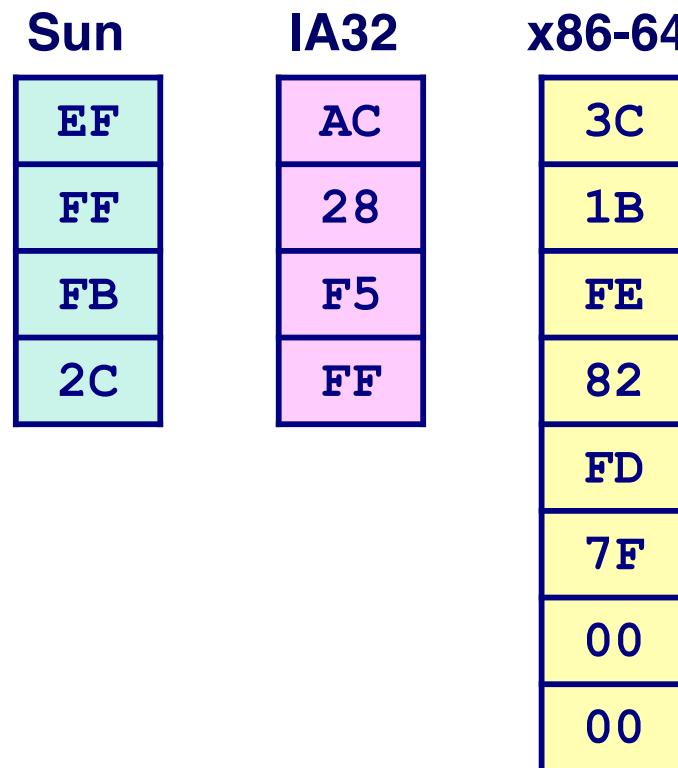
```
int a = 15213;  
printf("int a = 15213; \n");  
show_bytes((pointer) &a, sizeof(int));
```

## Result (Linux x86-64):

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

# Representing Pointers

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



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

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

# Representing Strings

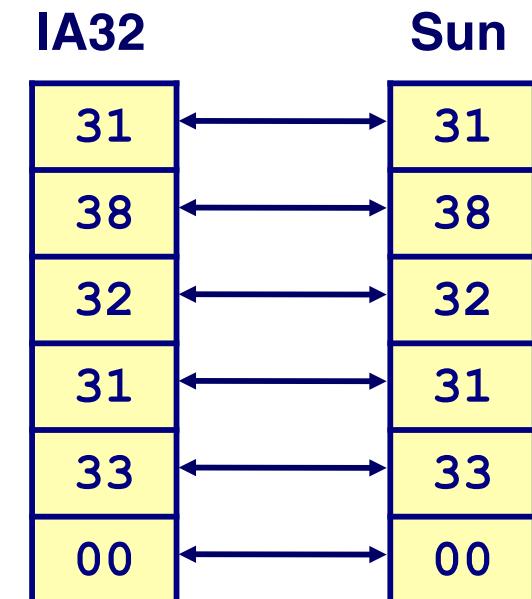
```
char S[6] = "18213";
```

## ■ Strings in C

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

## ■ Compatibility

- Byte ordering not an issue



# 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;
```

$x < 0$	$\Rightarrow (x^2) < 0$	<span style="color: red;">X</span>
$ux \geq 0$		<span style="color: green;">✓</span>
$x \& 7 == 7$	$\Rightarrow (x << 30) < 0$	<span style="color: green;">✓</span>
$ux > -1$		<span style="color: red;">X</span>
$x > y$	$\Rightarrow -x < -y$	<span style="color: red;">X</span>
$x * x \geq 0$		<span style="color: red;">X</span>
$x > 0 \&& y > 0$	$\Rightarrow x + y > 0$	<span style="color: red;">X</span>
$x \geq 0$	$\Rightarrow -x \leq 0$	<span style="color: green;">✓</span>
$x \leq 0$	$\Rightarrow -x \geq 0$	<span style="color: red;">X</span>
$(x   -x) >> 31 == -1$		<span style="color: red;">X</span>
$ux >> 3 == ux/8$		<span style="color: green;">✓</span>
$x >> 3 == x/8$		<span style="color: red;">X</span>
$x \& (x-1) != 0$		<span style="color: red;">X</span>

# 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, strings**
- **Summary**