

# Bits, Bytes, and Integers

Lectures 1+2, Feb. 10+17, 2011

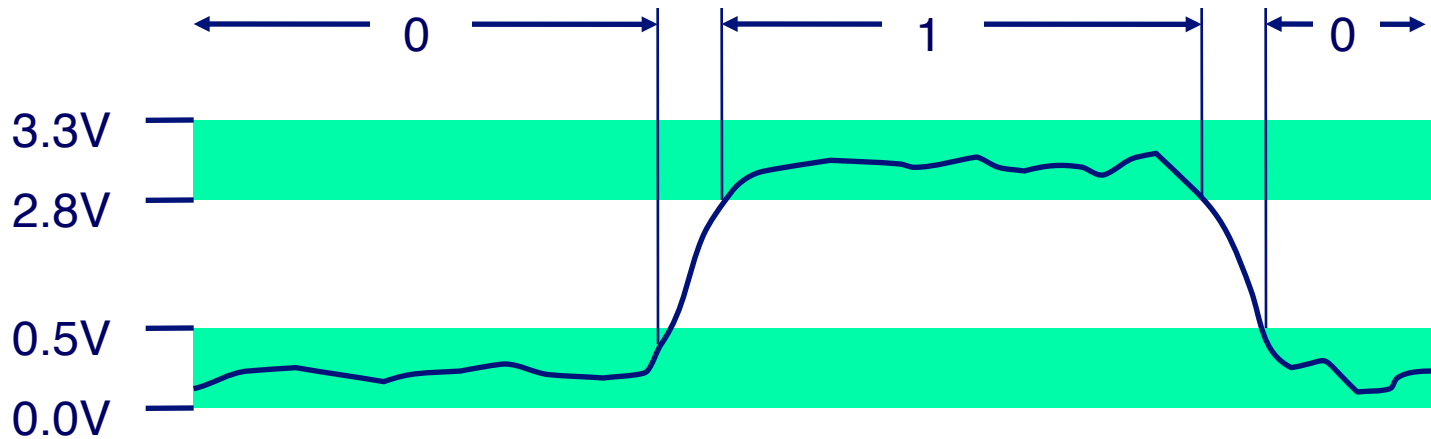
Alexandre David

Credits to Randy Bryant & Dave O'Hallaron  
from Carnegie Mellon

# 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**

# Binary Representations



**Humans: 10 fingers, base 10 natural.**

**Computers: base 2 more natural and reliable.**

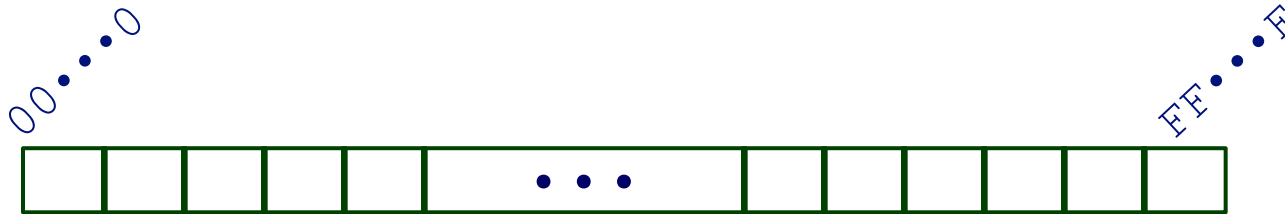
# Encoding Byte Values

## ■ 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

# Byte-Oriented Memory Organization



## ■ Programs Refer to Virtual Addresses

- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

## ■ Compiler + Run-Time System Control Allocation

- Where different program objects should be stored
- All allocation within single virtual address space

# Machine Words

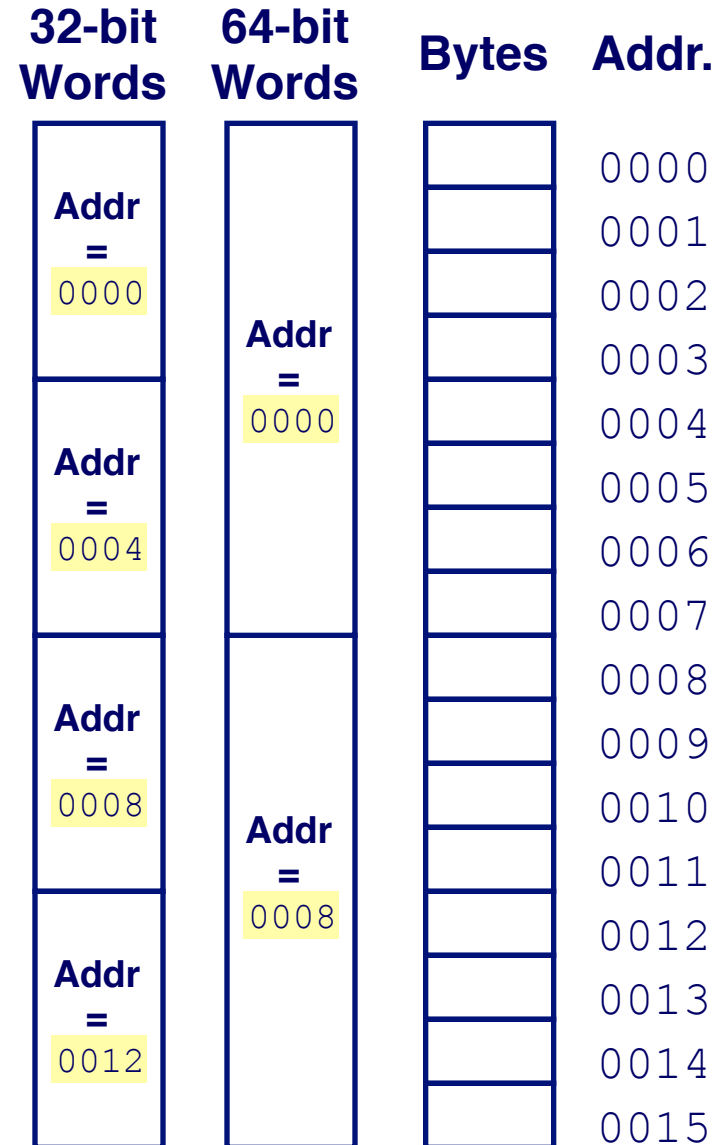
## ■ Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - **Limits addresses to 4GB**
  - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space  $\approx 1.8 \times 10^{19}$  bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines 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)



**Related to pointer arithmetic.**

# Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8



# Byte Ordering

- **How should bytes within a multi-byte word be ordered in memory?**
- **Conventions**
  - **Big Endian:** Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - **Little Endian:** x86
    - Least significant byte has lowest address

# Byte Ordering Example

## ■ Big Endian

- Least significant byte has highest address

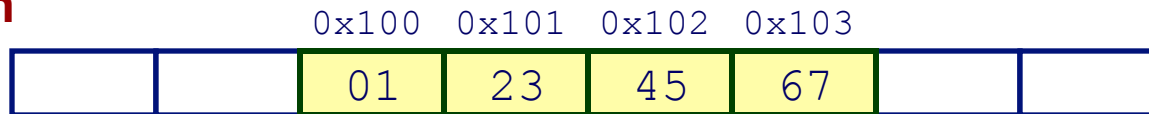
## ■ Little Endian

- Least significant byte has lowest address

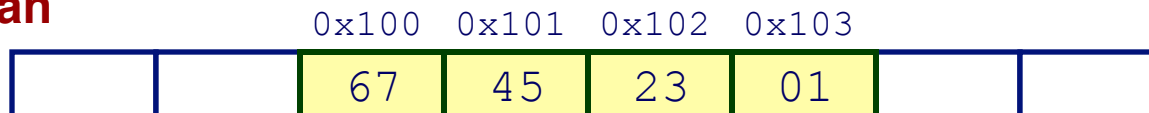
## ■ Example

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

### Big Endian



### Little Endian



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

# Examining Data Representations

## ■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char \* creates byte array

```
typedef unsigned char *pointer;  
  
void show_bytes(pointer start, int len){  
    int 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));
```

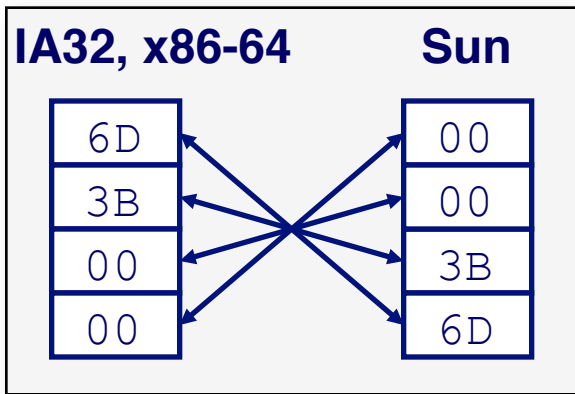
## Result (Linux):

```
int a = 15213;  
0x11ffffcb8 0x6d  
0x11ffffcb9 0x3b  
0x11ffffcba 0x00  
0x11ffffcbb 0x00
```

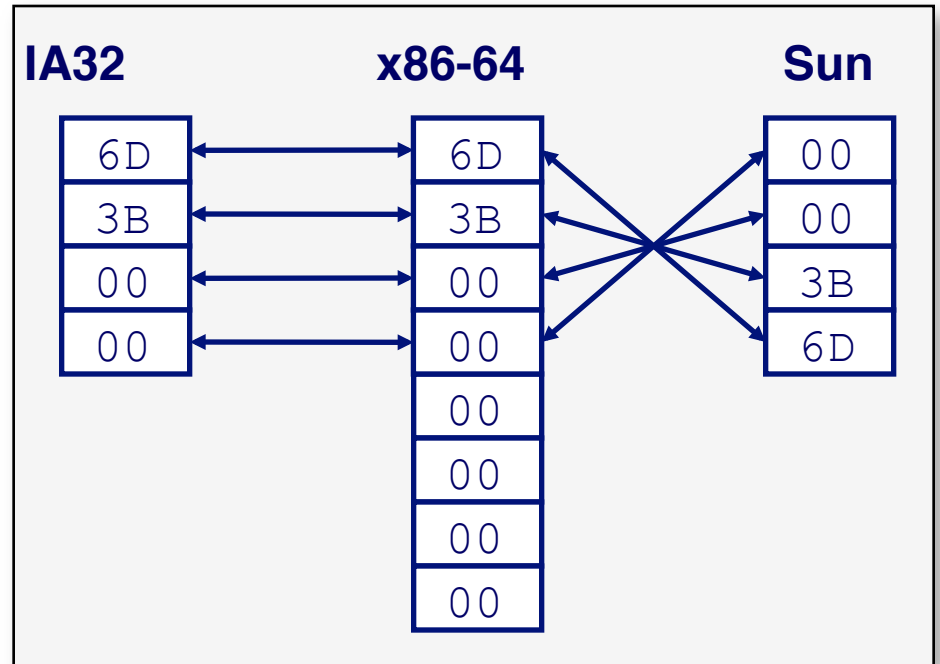
# Representing Integers

**Decimal:** 15213  
**Binary:** 0011 1011 0110 1101  
**Hex:** 3 B 6 D

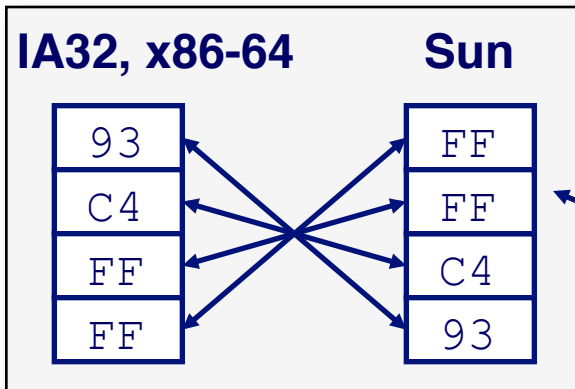
int A = 15213;



long int C = 15213;



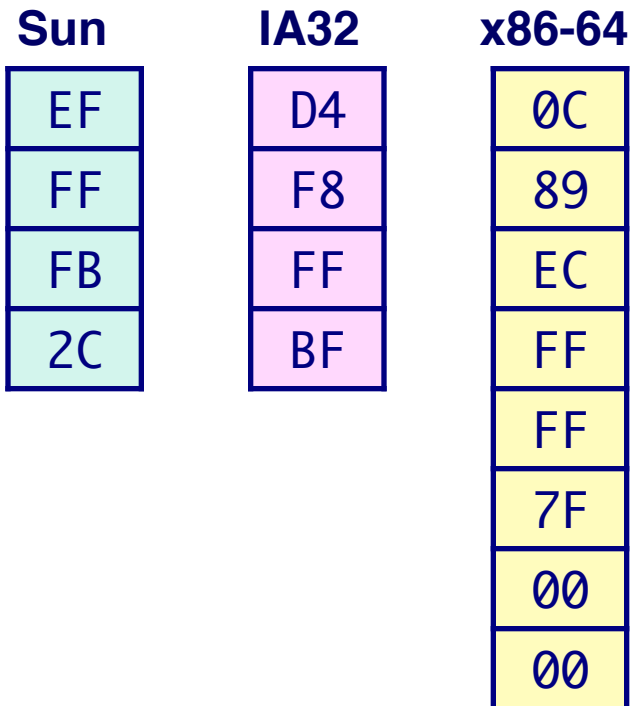
int B = -15213;



Two's complement representation  
(Covered later)

# Representing Pointers

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



Different compilers & machines assign different locations to objects

# (Representing Strings)

```
char S[6] = "18243";
```

## ■ Strings in C

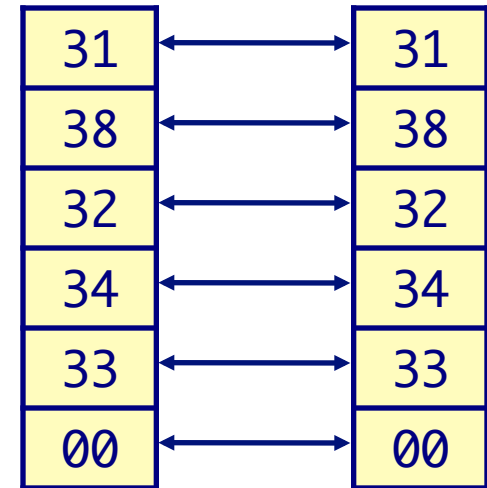
- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character "0" has code 0x30
    - Digit i has code 0x30+i
- String should be null-terminated
  - Final character = 0

## ■ Compatibility

- Byte ordering not an issue

Linux/Alpha

Sun





# 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**

# 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

Not

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

$\sim$	
0	1
1	0

Or

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

	0	1
0	0	1
1	1	1

Exclusive-Or (Xor)

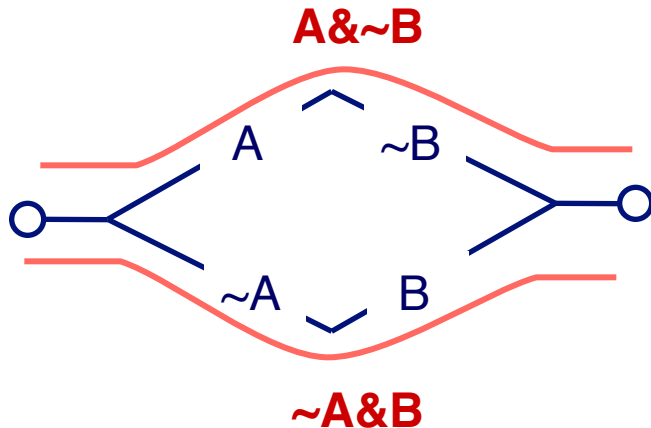
- $A \wedge B = 1$  when either  $A=1$  or  $B=1$ , but not both

$\wedge$	0	1
0	0	1
1	1	0

# Application of Boolean Algebra

## ■ Applied to Digital Systems by Claude Shannon

- 1937 MIT Master's Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0



Connection when

$$A \& \sim B \mid \sim A \& B$$

$$= A \wedge B$$

xor

# General Boolean Algebras

## ■ Operate on Bit Vectors

- Operations applied bitwise

01101001	01101001	01101001	01101001
& 01010101	01010101	^ 01010101	~ 01010101
01000001	01111101	00111100	10101010

## ■ All of the Properties of Boolean Algebra Apply

# 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

Useful, simple, efficient,  
for “small” integers.

## ■ 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)

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

# Contrast: Logic Operations in C

## ■ Contrast to Logical Operators

- `&&`, `||`, `!`
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - **Early termination**

`!a` same as `a == 0`

## ■ Examples (char data type)

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

# 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 right

## ■ Undefined Behavior

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

<b>Argument</b> $x$	01100010
$\ll 3$	00010000
<b>Log.</b> $\gg 2$	00011000
<b>Arith.</b> $\gg 2$	00011000

<b>Argument</b> $x$	10100010
$\ll 3$	00010000
<b>Log.</b> $\gg 2$	00101000
<b>Arith.</b> $\gg 2$	11101000



# Aside: Shifts

- Shifts are undefined for values  $\geq$  word size.
- Why?
- Machine instruction for shifts operates with the CL register and not all bits are used.
  - $x \gg m$  implemented as  $x \gg (m \% \text{word\_size})$
  - $x \ll m$  implemented as  $x \ll (m \% \text{word\_size})$

# Aside: Precedence

- **Beware!**
- **Binary operators have low precedence.**
  - $a \ll 3 + b \ll 2$  same as  $a \ll (3 + b) \ll 2$
  - $a \& 3 \neq 0$  same as  $a \& (3 \neq 0)$
- **Always put parenthesis around these operators.**

# 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**

# 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
<b>x</b>	15213	3B 6D	00111011 01101101
<b>y</b>	-15213	C4 93	11000100 10010011

■ Sign Bit

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

# 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
<b>Sum</b>		<b>15213</b>		<b>-15213</b>

# 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
<b>UMax</b>	<b>65535</b>	<b>FF FF</b>	<b>11111111 11111111</b>
<b>TMax</b>	<b>32767</b>	<b>7F FF</b>	<b>01111111 11111111</b>
<b>TMin</b>	<b>-32768</b>	<b>80 00</b>	<b>10000000 00000000</b>
<b>-1</b>	<b>-1</b>	<b>FF FF</b>	<b>11111111 11111111</b>
<b>0</b>	<b>0</b>	<b>00 00</b>	<b>00000000 00000000</b>

# 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

**Java has its constants too.**

# Aside: Arbitrary Precision

- **Some libraries support arbitrary precision computations.**
  - Limited by RAM.
- **Ruby integers behave like this.**
  - Try it.
- **Not a native type (machine word).**



# 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

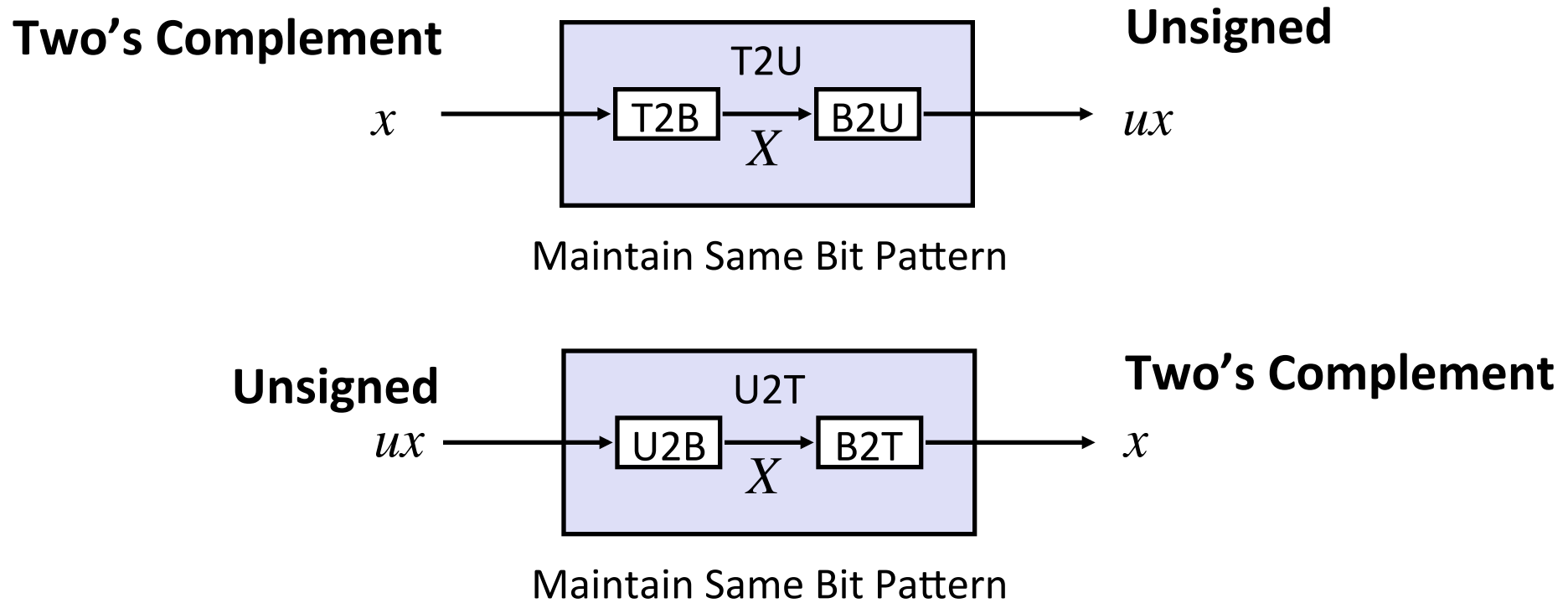
## ■ $\Rightarrow$ Can Invert Mappings

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

# Today: Bits, Bytes, and Integers

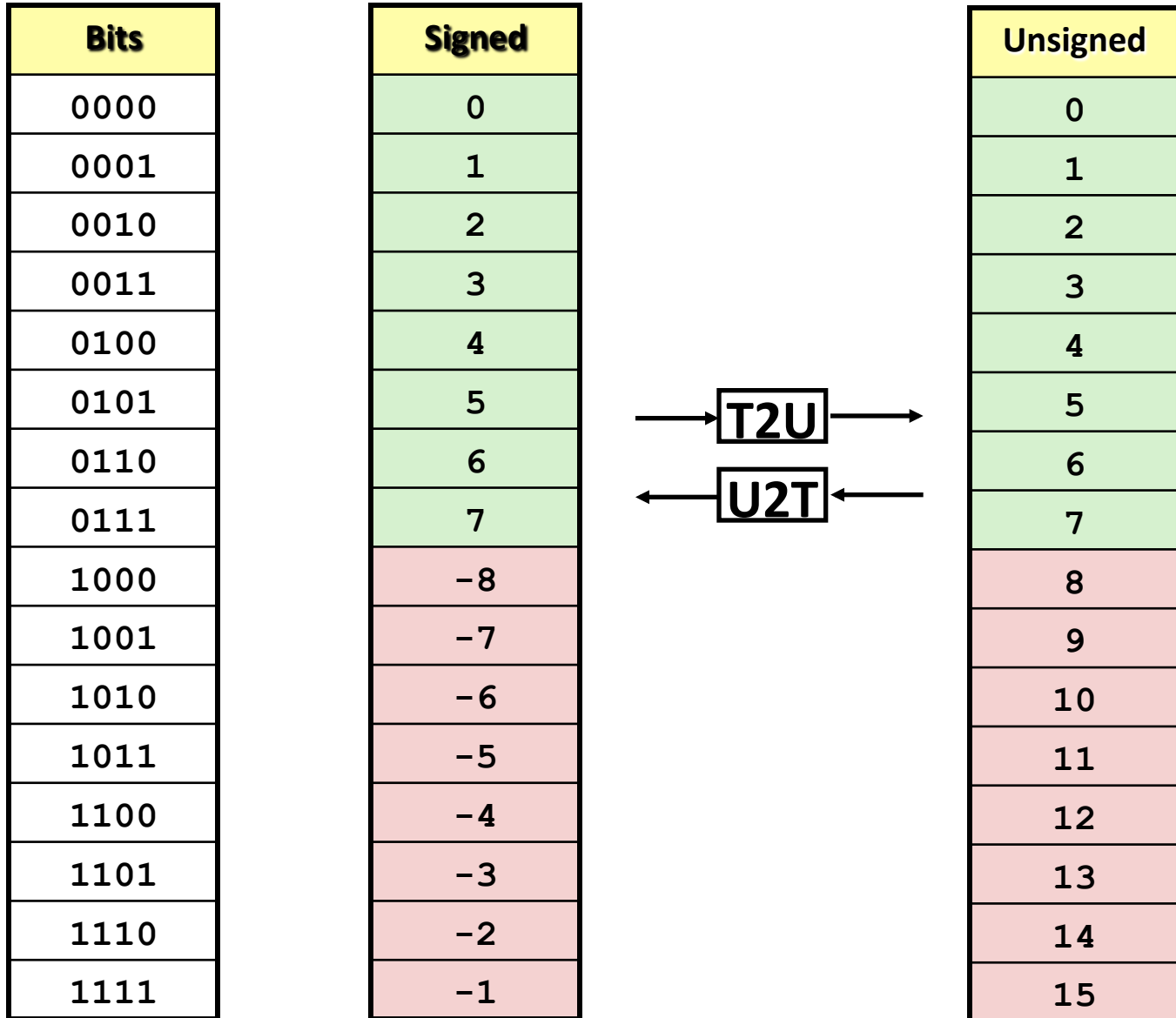
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- **Summary**

# Mapping Between Signed & Unsigned



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

# Mapping Signed ↔ Unsigned



# Mapping Signed ↔ Unsigned

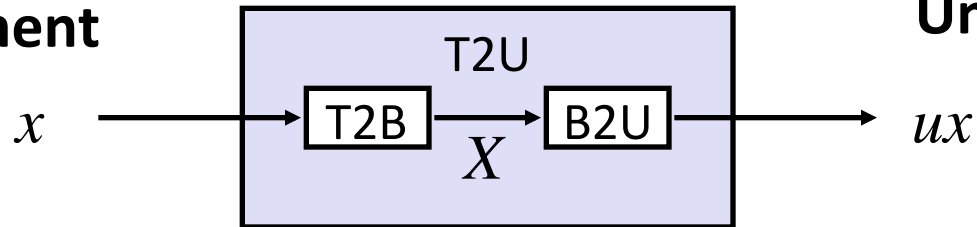
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

↔ = ↔

↔ +/- 16 ↔

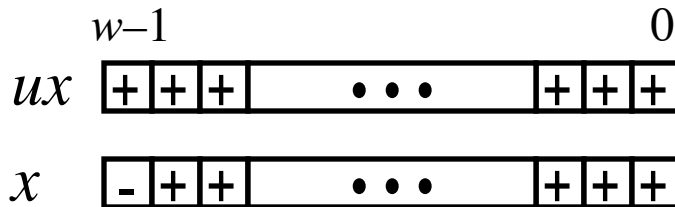
# Relation between Signed & Unsigned

Two's Complement



Unsigned

Maintain Same Bit Pattern



Large negative weight

*becomes*

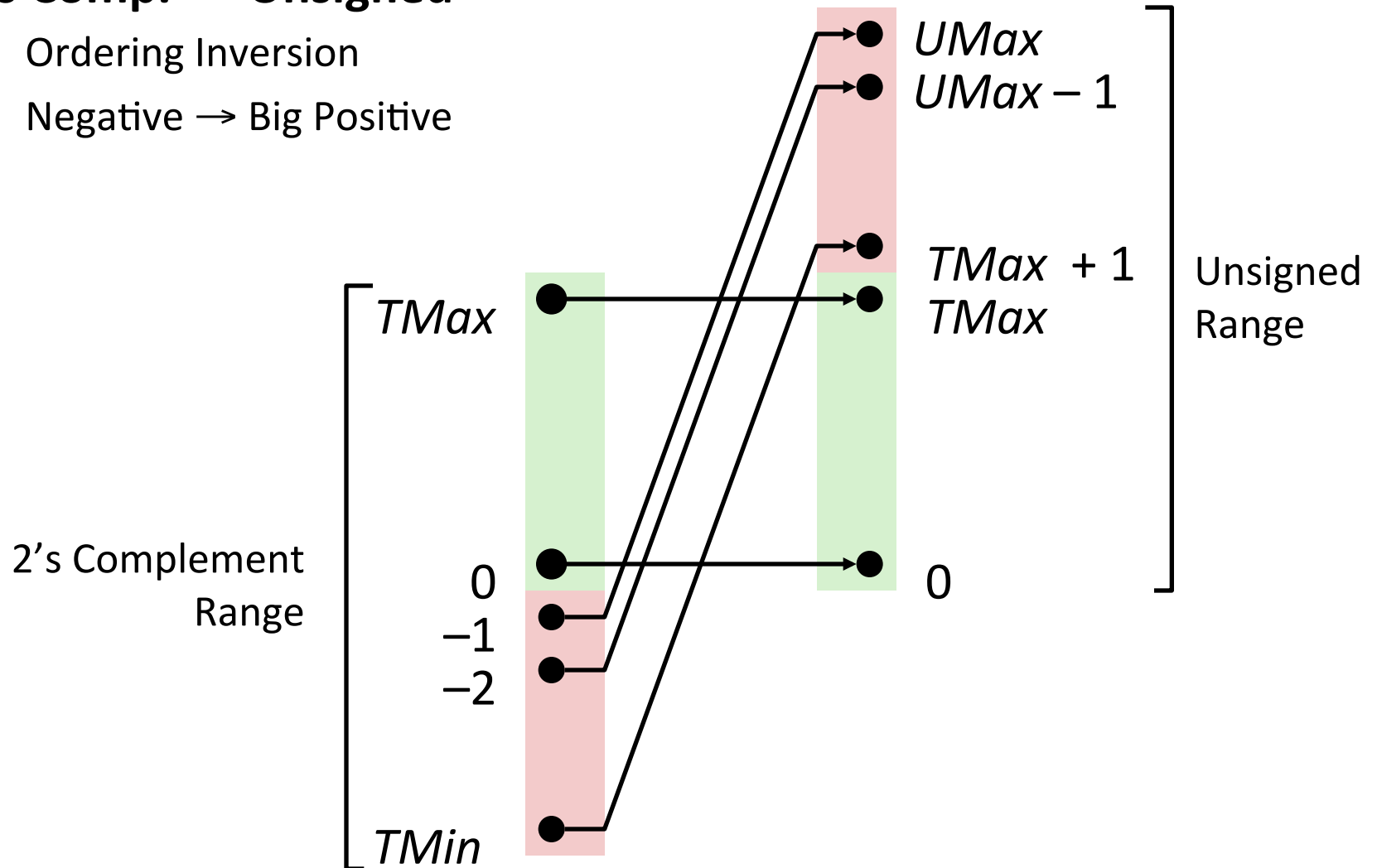
Large positive weight

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

# 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;
uy = ty;
```

Some languages, e.g. Java, have only ints.



# (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 <sub>1</sub>	Constant <sub>2</sub>	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

# Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD's implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs

# Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

# Malicious Usage

```
/* Declaration of library function memcpy */  
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */  
#define KSIZE 1024  
char kbuf[KSIZE];  
  
/* Copy at most maxlen bytes from kernel region to user buffer */  
int copy_from_kernel(void *user_dest, int maxlen) {  
    /* Byte count len is minimum of buffer size and maxlen */  
    int len = KSIZE < maxlen ? KSIZE : maxlen;  
    memcpy(user_dest, kbuf, len);  
    return len;  
}
```

```
#define MSIZE 528  
  
void getstuff() {  
    char mybuf[MSIZE];  
    copy_from_kernel(mybuf, -MSIZE);  
    . . .  
}
```

# 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

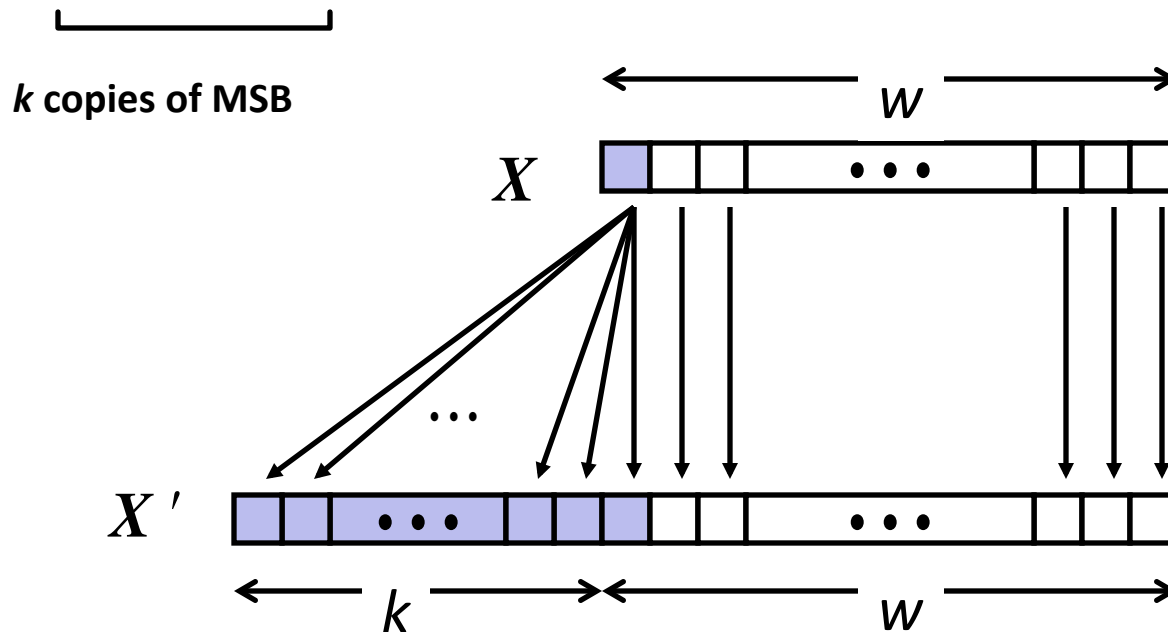
# Sign Extension

## ■ 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
<b>x</b>	15213	3B 6D	00111011 01101101
<b>ix</b>	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
<b>y</b>	-15213	C4 93	11000100 10010011
<b>iy</b>	-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
  
- **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 behaviour

# 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

# Negation: Complement & Increment

## ■ Claim: Following Holds for 2's Complement

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

## ■ Complement

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

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

## ■ Complete Proof?

# Complement & Increment Examples

**x = 15213**

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

**x = 0**

	Decimal	Hex	Binary
<b>0</b>	<b>0</b>	<b>00 00</b>	<b>00000000 00000000</b>
<b>~0</b>	<b>-1</b>	<b>FF FF</b>	<b>11111111 11111111</b>
<b>~0+1</b>	<b>0</b>	<b>00 00</b>	<b>00000000 00000000</b>

# Unsigned Addition

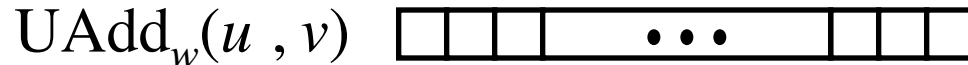
Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits



## ■ Standard Addition Function

- Ignores carry output

## ■ Implements Modular Arithmetic

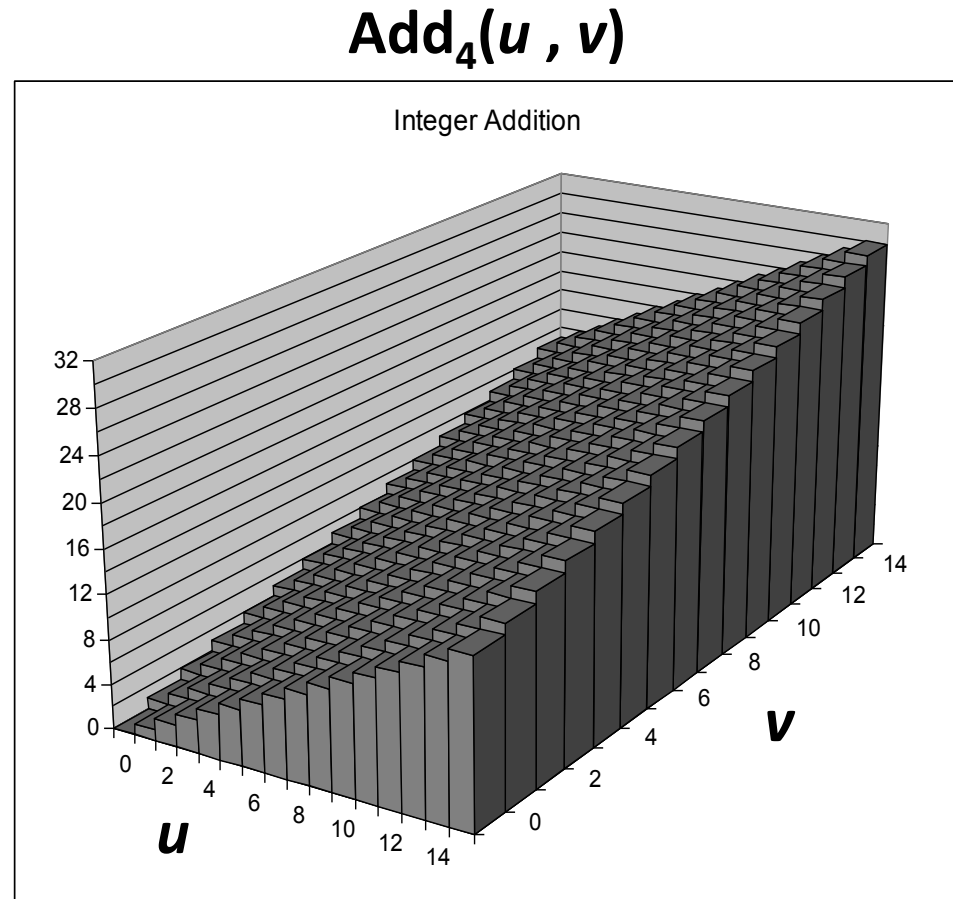
$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

$$\text{UAdd}_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

# 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

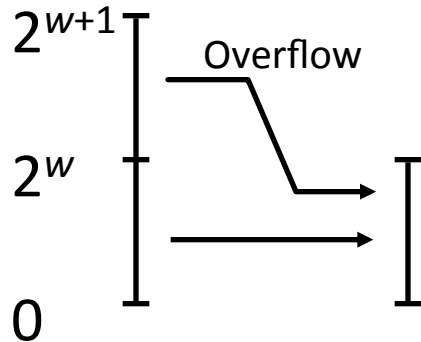


# Visualizing Unsigned Addition

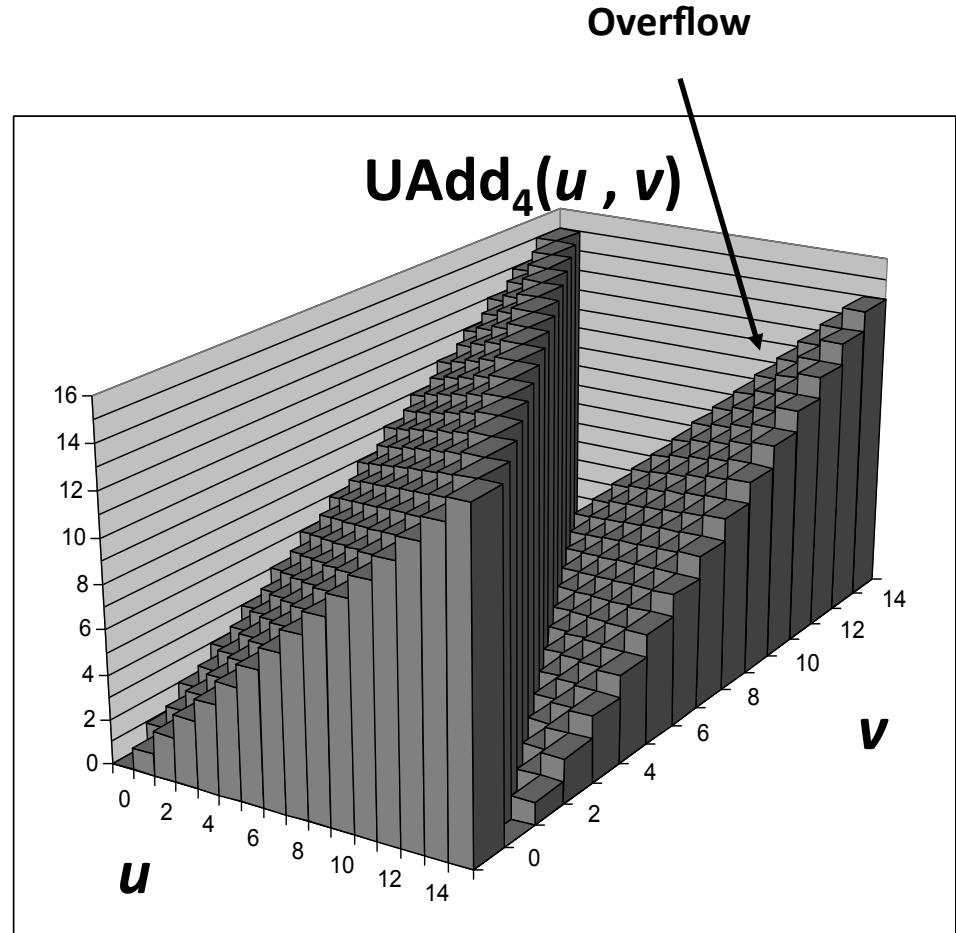
## ■ Wraps Around

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

**True Sum**



**Modular Sum**



# Mathematical Properties

## ■ Modular Addition Forms an *Abelian Group*

- **Closed** under addition

$$0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$$

- **Commutative**

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

- **Associative**

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

- **0** is additive identity

$$\text{UAdd}_w(u, 0) = u$$

- Every element has additive **inverse**

- Let  $\text{UComp}_w(u) = 2^w - u$   
 $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$



# Two's Complement Addition

Operands:  $w$  bits



+  $v$



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);
```

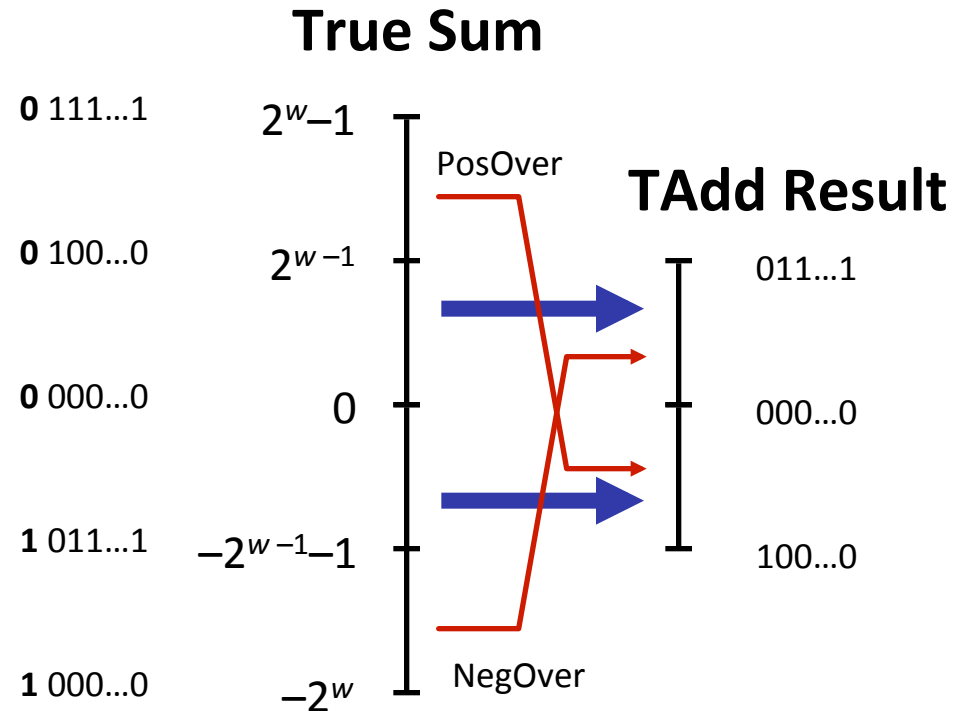
```
t = u + v
```

- 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

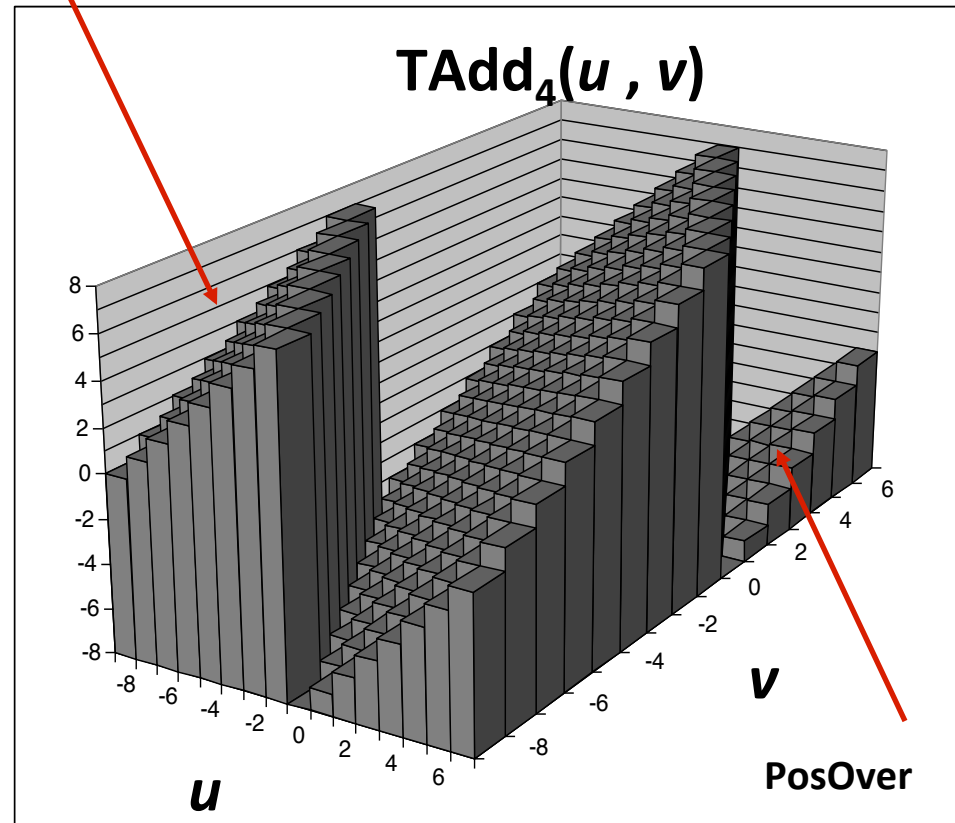
## ■ Values

- 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

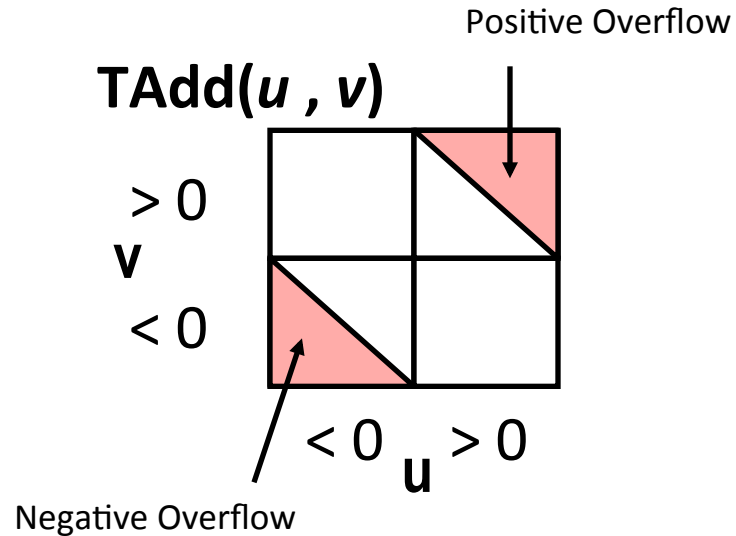
NegOver



# Characterizing TAdd

## ■ Functionality

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



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

# Multiplication

## ■ Computing Exact Product of $w$ -bit numbers $x, y$

- Either signed or unsigned

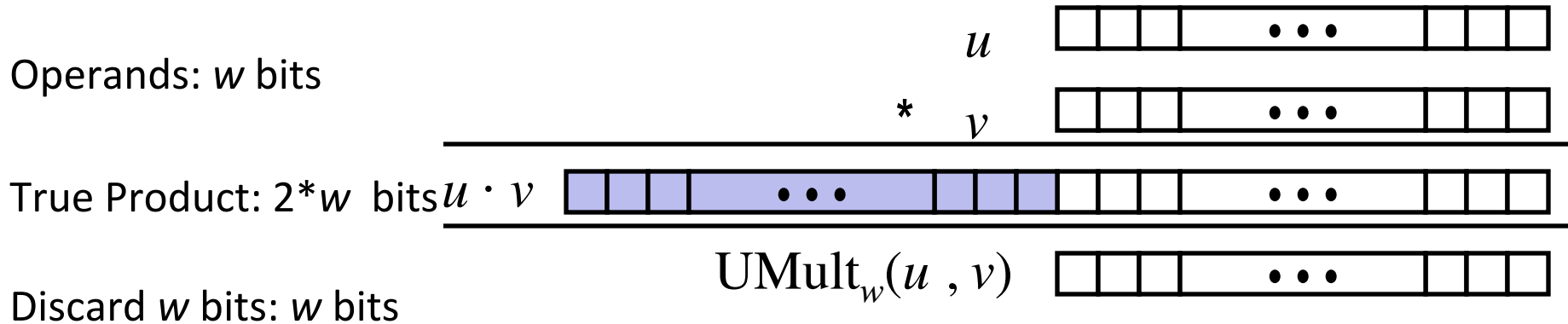
## ■ Ranges

- Unsigned:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$ 
  - Up to  $2w$  bits
- Two's complement min:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ 
  - Up to  $2w-1$  bits
- Two's complement max:  $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$ 
  - Up to  $2w$  bits, but only for  $(TMin_w)^2$

## ■ Maintaining Exact Results

- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages

# Unsigned Multiplication in C



## ■ Standard Multiplication Function

- Ignores high order  $w$  bits

## ■ Implements Modular Arithmetic

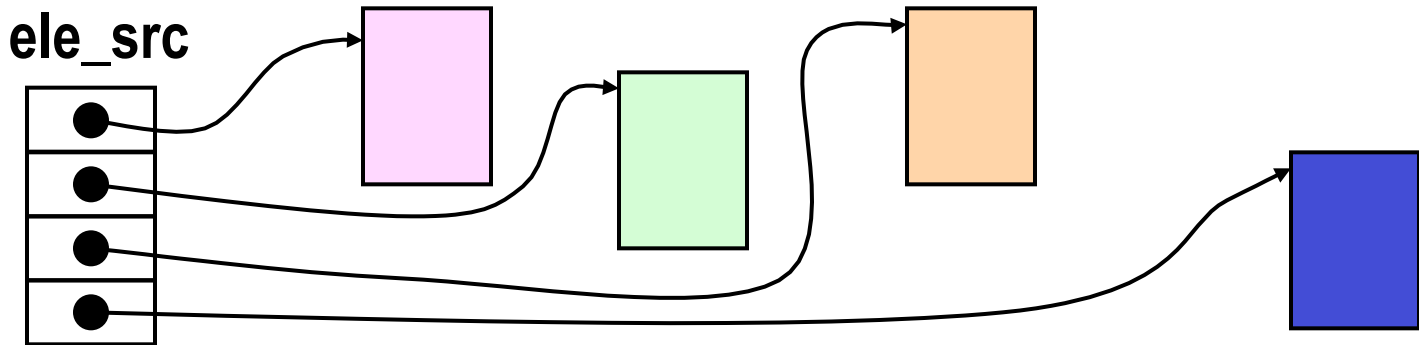
$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

# Code Security Example #2

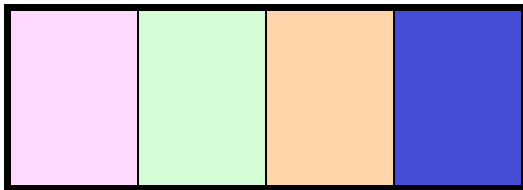
## ■ SUN XDR library

- Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



`malloc(ele_cnt * ele_size)`



# XDR Code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```



# XDR Vulnerability

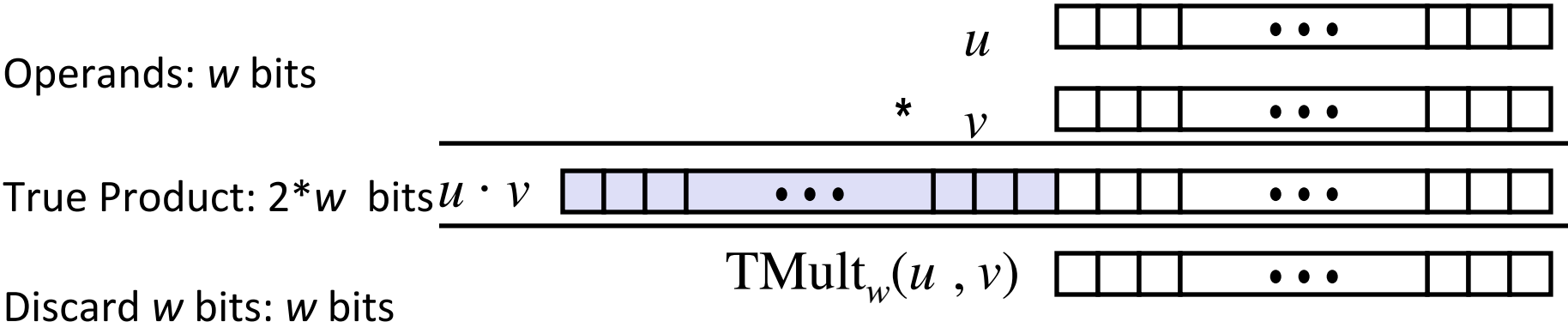
`malloc(ele_cnt * ele_size)`

## ■ What if:

- `ele_cnt` =  $2^{20} + 1$
- `ele_size` = 4096 =  $2^{12}$
- Allocation = ??

## ■ How can I make this function secure?

# Signed Multiplication in C



## ■ Standard Multiplication Function

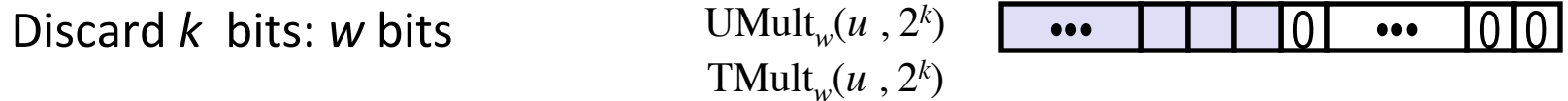
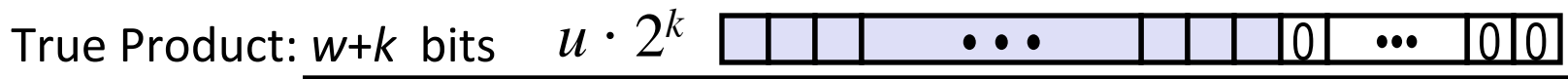
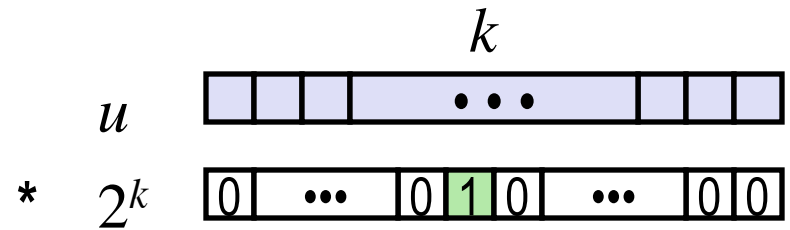
- Ignores high order  $w$  bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

# Power-of-2 Multiply with Shift

## ■ Operation

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

Operands:  $w$  bits



## ■ Examples

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

# Compiled Multiplication Code

## C Function

```
int mul12(int x)
{
    return x*12;
}
```

## Compiled Arithmetic Operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

## Explanation

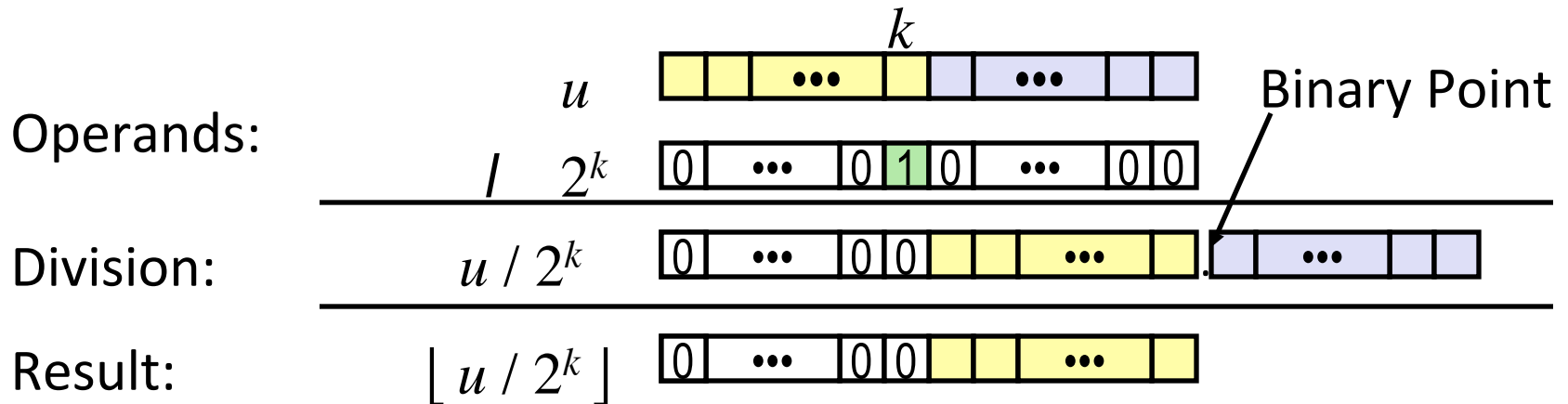
```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

# 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>

# Compiled Unsigned Division Code

## C Function

```
unsigned udiv8(unsigned x)
{
    return x/8;
}
```

## Compiled Arithmetic Operations

```
shrl $3, %eax
```

## Explanation

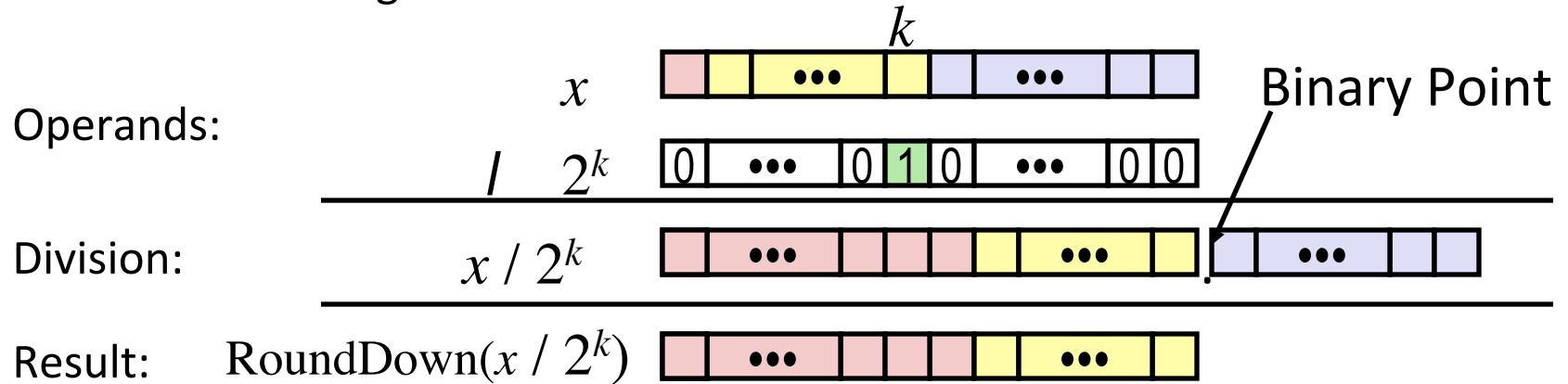
```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>

# 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  $u < 0$



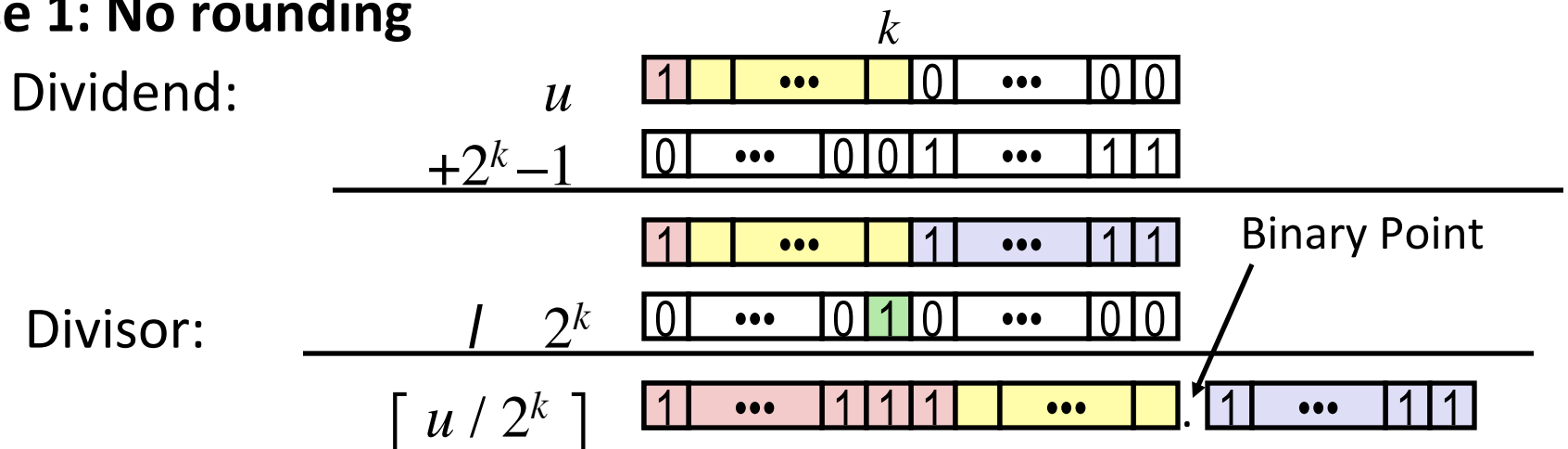
	Division	Computed	Hex	Binary
$y$	-15213	-15213	C4 93	11000100 10010011
$y \gg 1$	-7606.5	-7607	E2 49	11100010 01001001
$y \gg 4$	-950.8125	-951	FC 49	11111100 01001001
$y \gg 8$	-59.4257813	-60	FF C4	11111111 11000100

# Correct Power-of-2 Divide

## ■ Quotient of Negative Number by Power of 2

- Want  $\lceil \mathbf{x} / 2^k \rceil$  (Round Toward 0)
- Compute as  $\lfloor (\mathbf{x} + 2^k - 1) / 2^k \rfloor$ 
  - In C:  $(\mathbf{x} + (1 \ll k) - 1) \gg k$
  - Biases dividend toward 0

### Case 1: No rounding

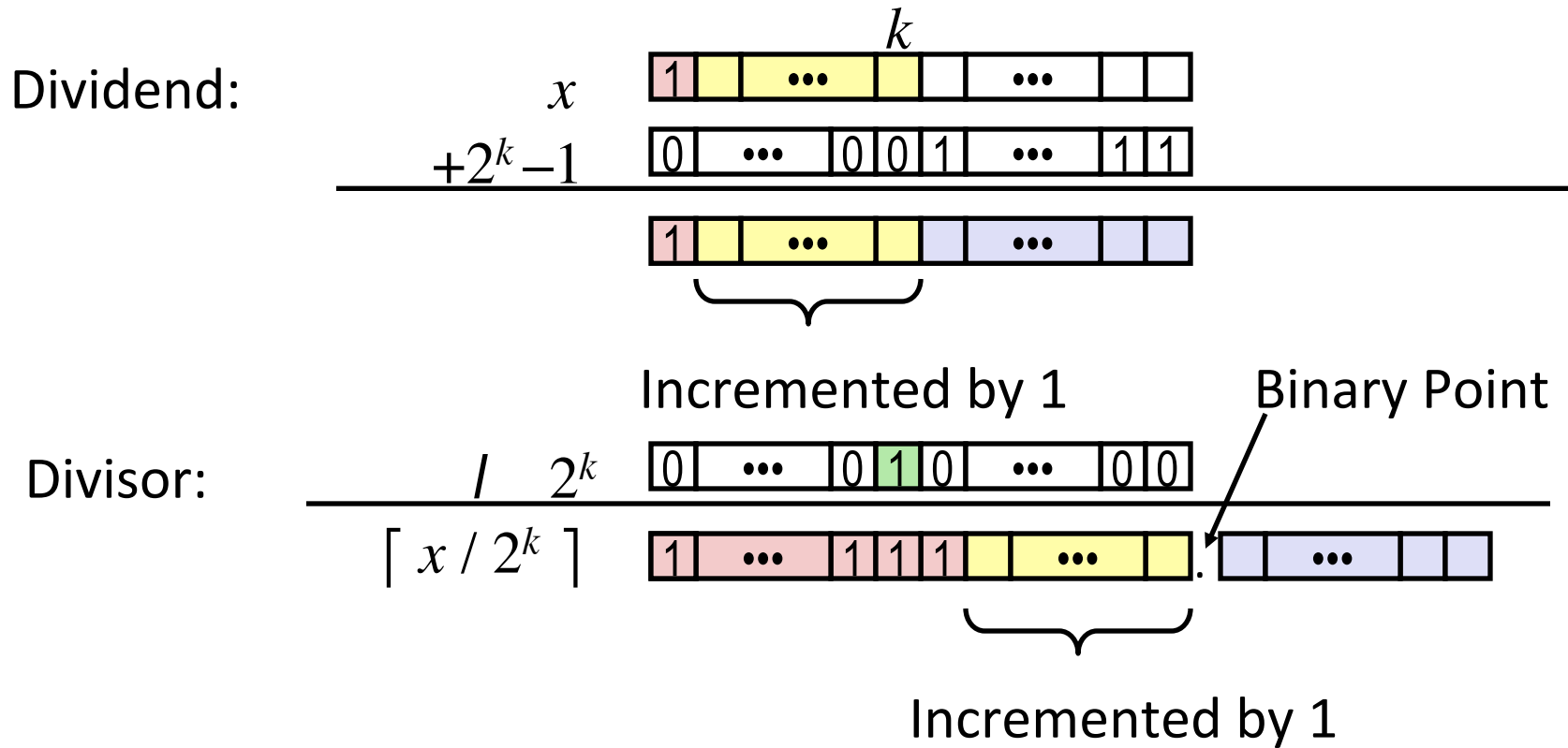


*Biassing has no effect*



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

## Case 2: Rounding



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

# Compiled Signed Division Code

## C Function

```
int idiv8(int x)
{
    return x/8;
}
```

## Compiled Arithmetic Operations

```
    testl %eax, %eax
    js    L4
L3:
    sarl $3, %eax
    ret
L4:
    addl $7, %eax
    jmp  L3
```

## Explanation

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as >>

# 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  $2w$
- Signed: modified addition mod  $2^w$  (result in proper range)
  - Mathematical addition + possible addition or subtraction of  $2w$

## ■ 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)

# Arithmetic: Basic Rules

- **Unsigned ints, 2's complement ints are isomorphic rings: isomorphism = casting**
- **Left shift**
  - Unsigned/signed: multiplication by  $2^k$
  - Always logical shift
- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by  $2^k$
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by  $2^k$
    - Negative numbers: div (division + round away from zero) by  $2^k$   
Use biasing to fix

# Integer C Puzzles

## Initialization

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

- $x < 0 \Rightarrow ((x*2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7 \Rightarrow (x \ll 30) < 0$
- $ux > -1$
- $x > y \Rightarrow -x < -y$
- $x * x \geq 0$
- $x > 0 \&\& y > 0 \Rightarrow x + y > 0$
- $x \geq 0 \Rightarrow -x \leq 0$
- $x \leq 0 \Rightarrow -x \geq 0$
- $(x|-x) \gg 31 == -1$
- $ux \gg 3 == ux/8$
- $x \gg 3 == x/8$
- $x \& (x-1) != 0$