# Bits, Bytes, and Integers

15-213: Introduction to Computer Systems
2nd and 3rd Lectures, Aug 29 and Sep 3, 2013

**Instructors:**

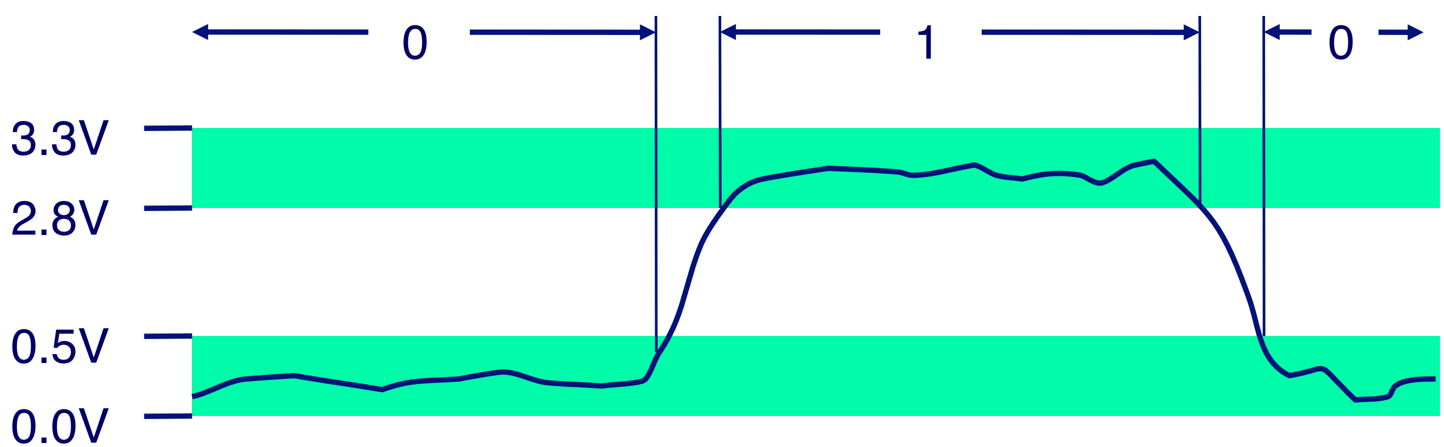
Randy Bryant, Dave O'Hallaron, and Greg Kesden

# 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

# Binary Representations

- **Base 2 Number Representation**
  - Represent $15213_{10}$ as $11101101101101_2$
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
  - Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$
- **Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



# Encoding Byte Values

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

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

# 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 |

# Today: Bits, Bytes, and Integers

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

# Boolean Algebra

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

And

- A&B = 1 when both A=1 and B=1

Or

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

Not

- ~A = 1 when A=0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

# General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

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

- **All of the Properties of Boolean Algebra Apply**

# Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **Operations**

| | | | |
|---|---|---|---|
| & | Intersection | 01000001 | { 0, 6 } |
| \| | Union | 01111101 | { 0, 2, 3, 4, 5, 6 } |
| ^ | Symmetric difference | 00111100 | { 2, 3, 4, 5 } |
| ~ | Complement | 10101010 | { 1, 3, 5, 7 } |

# Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any “integral” data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41 → 0xBE`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00 → 0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 → 0x41`
    - $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55 → 0x7D`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

# Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`

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

# Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as “Fal
    - Anything nonze
    - Alway
    - Early
- **Example**
  - `!0x41`
  - `!0x00`
  - `!!0x41`

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

**Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming**

# Shift Operations

- **Left Shift: `x << y`**
  - Shift bit-vector `x` left `y` positions
    - – Throw away extra bits on left
    - Fill with `0`'s on right
- **Right Shift: `x >> 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 ≥ word size

| | |
|---|---|
| **Argument** `x` | `01100010` |
| `<< 3` | `00010000` |
| **Log.** `>> 2` | `00011000` |
| **Arith.** `>> 2` | `00011000` |

| | |
|---|---|
| **Argument** `x` | `10100010` |
| `<< 3` | `00010000` |
| **Log.** `>> 2` | `00101000` |
| **Arith.** `>> 2` | `11101000` |

# Today: Bits, Bytes, and Integers

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

# Encoding Integers

**Unsigned**

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

**Two's Complement**

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

**Sign Bit**

```
short int x =  15213;
short int y = -15213;
```

- **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 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** |

# Binary Number Property

**Claim**

$$1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} = 2^w$$

$$1 + \sum_{i=0}^{w-1} 2^i \quad = \quad 2^w$$

- **w = 0:**
  - $1 = 2^0$
- **Assume true for w-1:**
  - $\underbrace{1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1}}_{= \; 2^w} + 2^w \;=\; 2^w + 2^w \;=\; 2^{w+1}$

# 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

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

# Unsigned & Signed Numeric Values

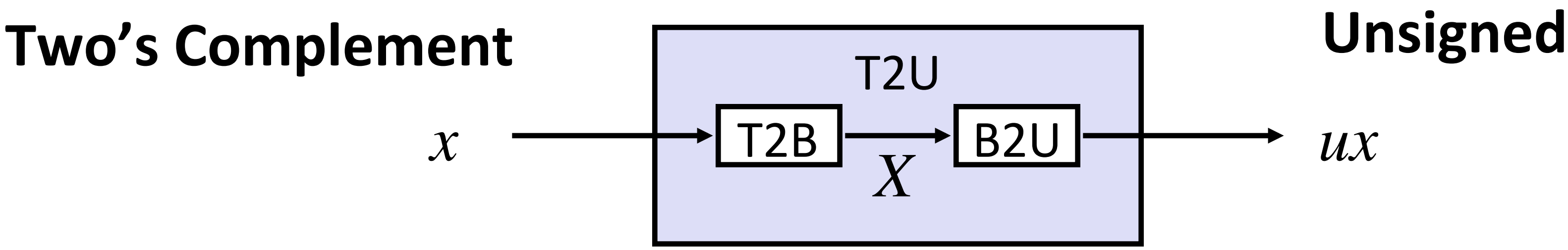
| *X* | B2U(*X*) | B2T(*X*) |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | –8 |
| 1001 | 9 | –7 |
| 1010 | 10 | –6 |
| 1011 | 11 | –5 |
| 1100 | 12 | –4 |
| 1101 | 13 | –3 |
| 1110 |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

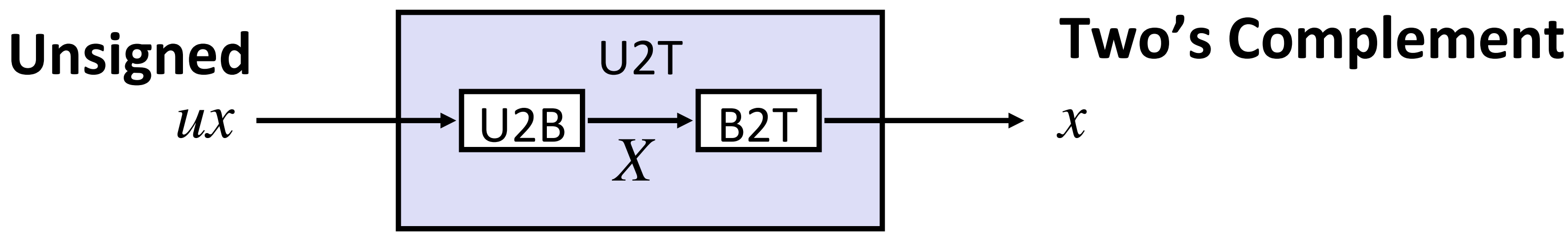
# Today: Bits, Bytes, and Integers

- 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




Maintain Same Bit Pattern




Maintain Same Bit Pattern

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

# Mapping Signed ↔ Unsigned

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

→ T2U →

← U2T ←

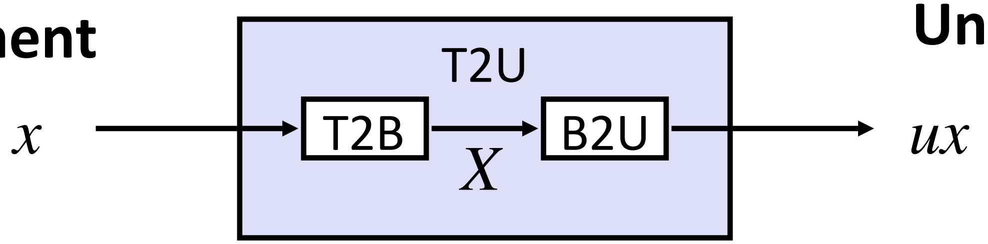
# Mapping Signed ↔ Unsigned

| 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 | +/- 16 | 11 |
| 1100 | -4 | | 12 |
| 1101 | -3 | | 13 |
| 1110 | -2 | | 14 |
| 1111 | -1 | | 15 |

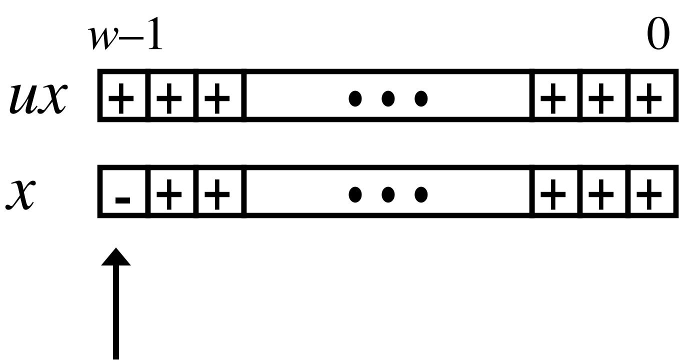
# Relation between Signed & Unsigned

**Two's Complement**

**Unsigned**




Maintain Same Bit Pattern



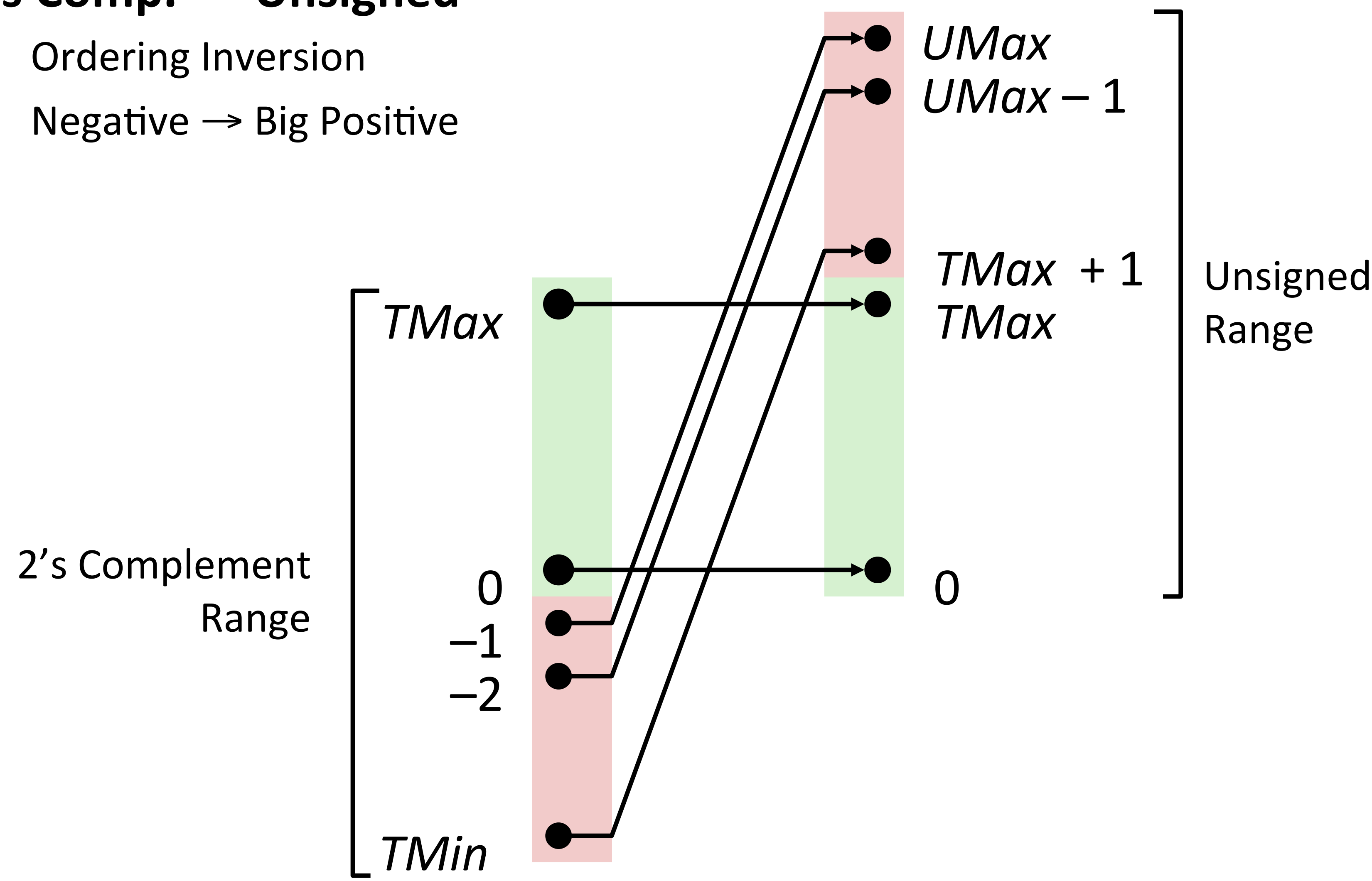

**Large negative weight**

*becomes*

**Large positive weight**

# Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



# Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
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**

| **$\text{Constant}_1$** | **$\text{Constant}_2$** | **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 ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$

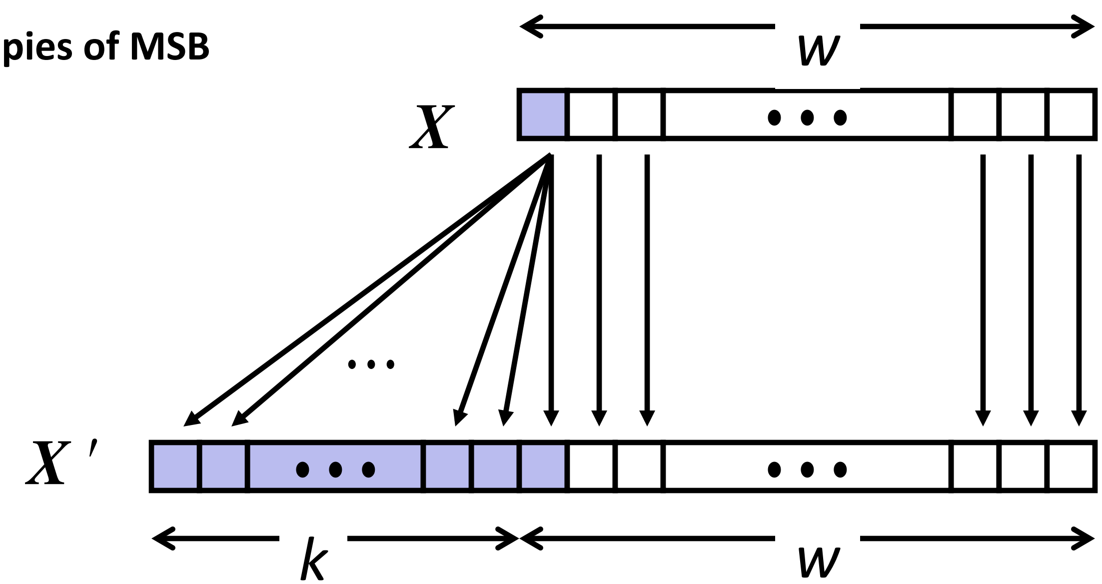
- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

# Today: Bits, Bytes, and Integers

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

# Sign Extension

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



# 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

- **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 behavour

# Today: Bits, Bytes, and Integers

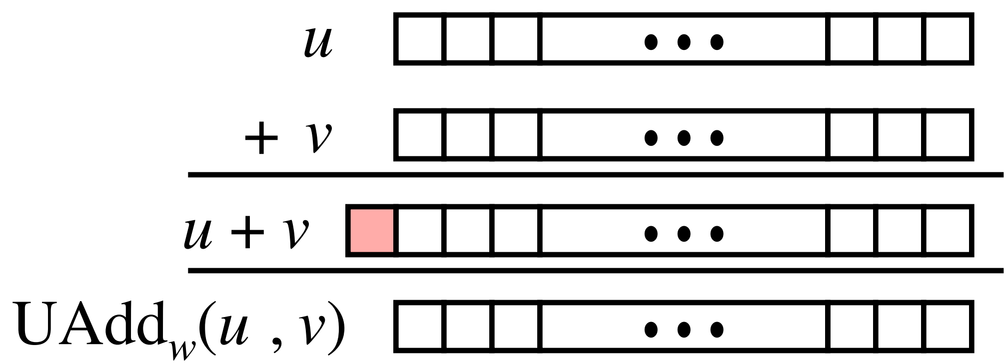
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

# Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



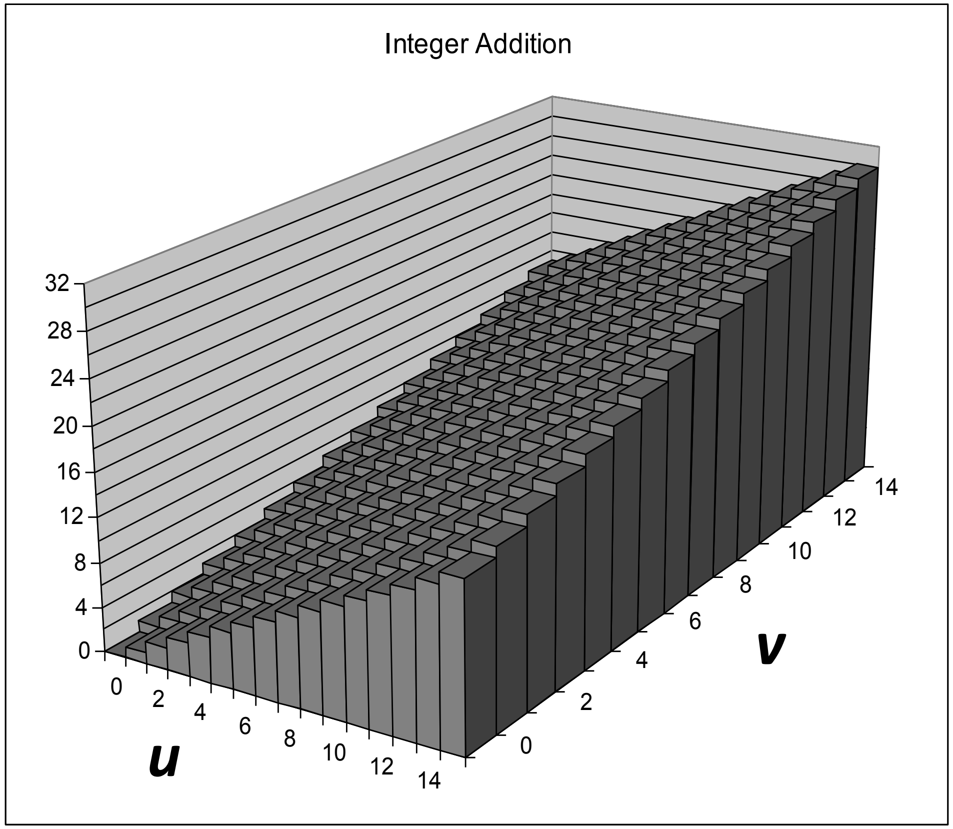

- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

$$s \quad = \quad \mathrm{UAdd}_w(u\ ,\ v) \quad = \quad u + v \mod 2^w$$

# Visualizing (Mathematical) Integer Addition

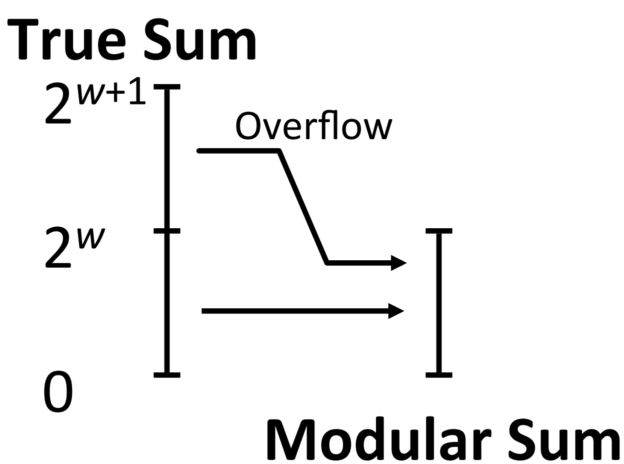
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

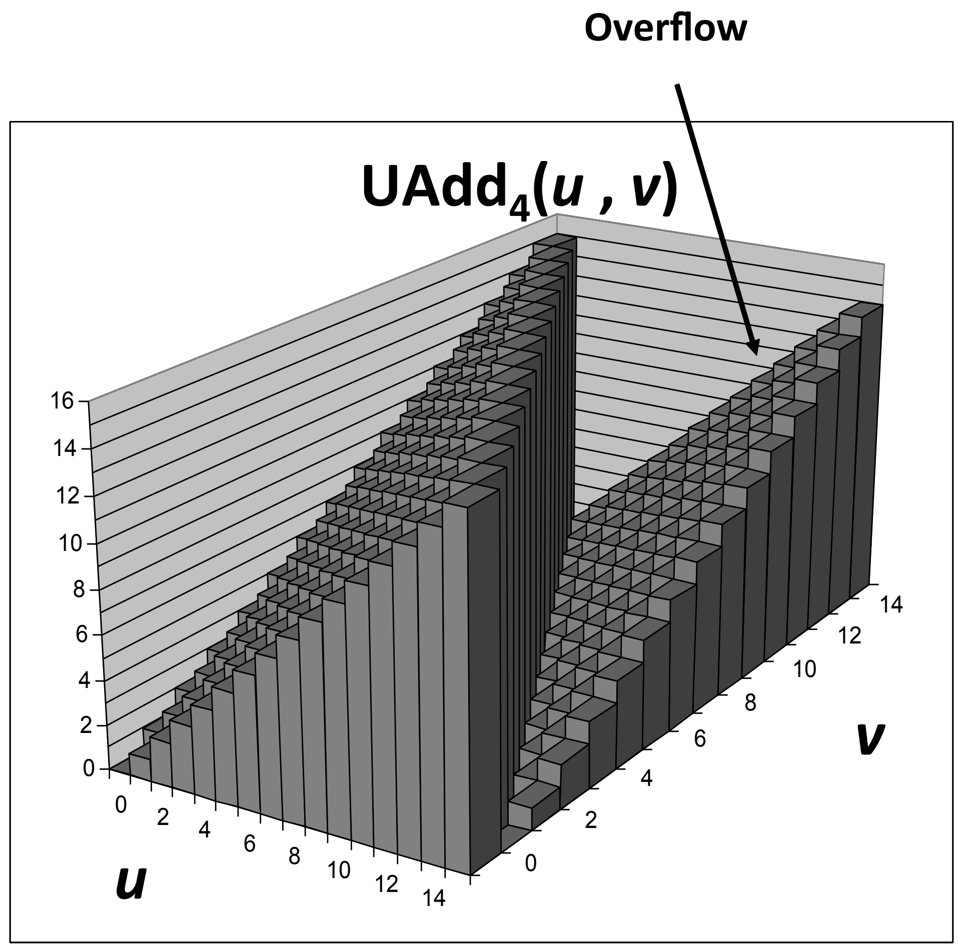
**$\text{Add}_4(u , v)$**



# Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum ≥ $2^w$
  - At most once



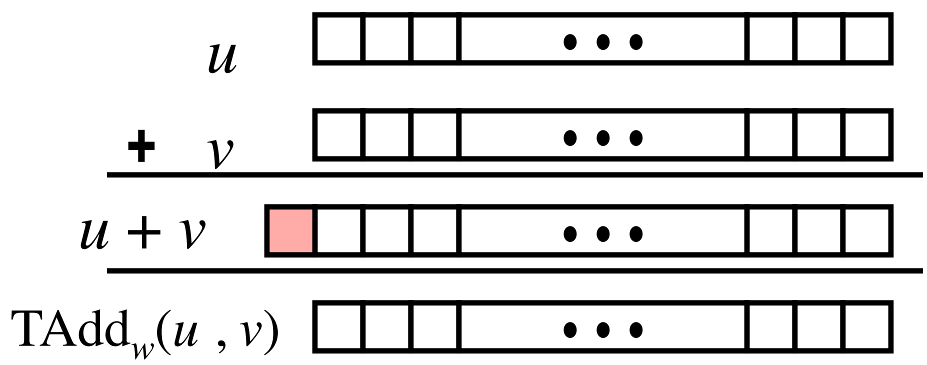



# Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



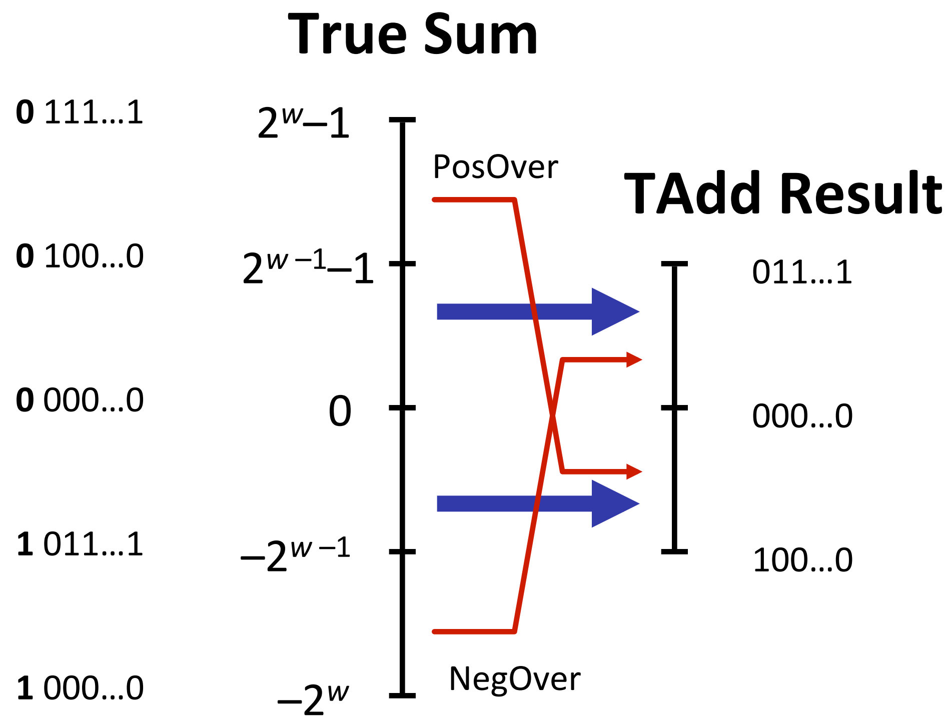

- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

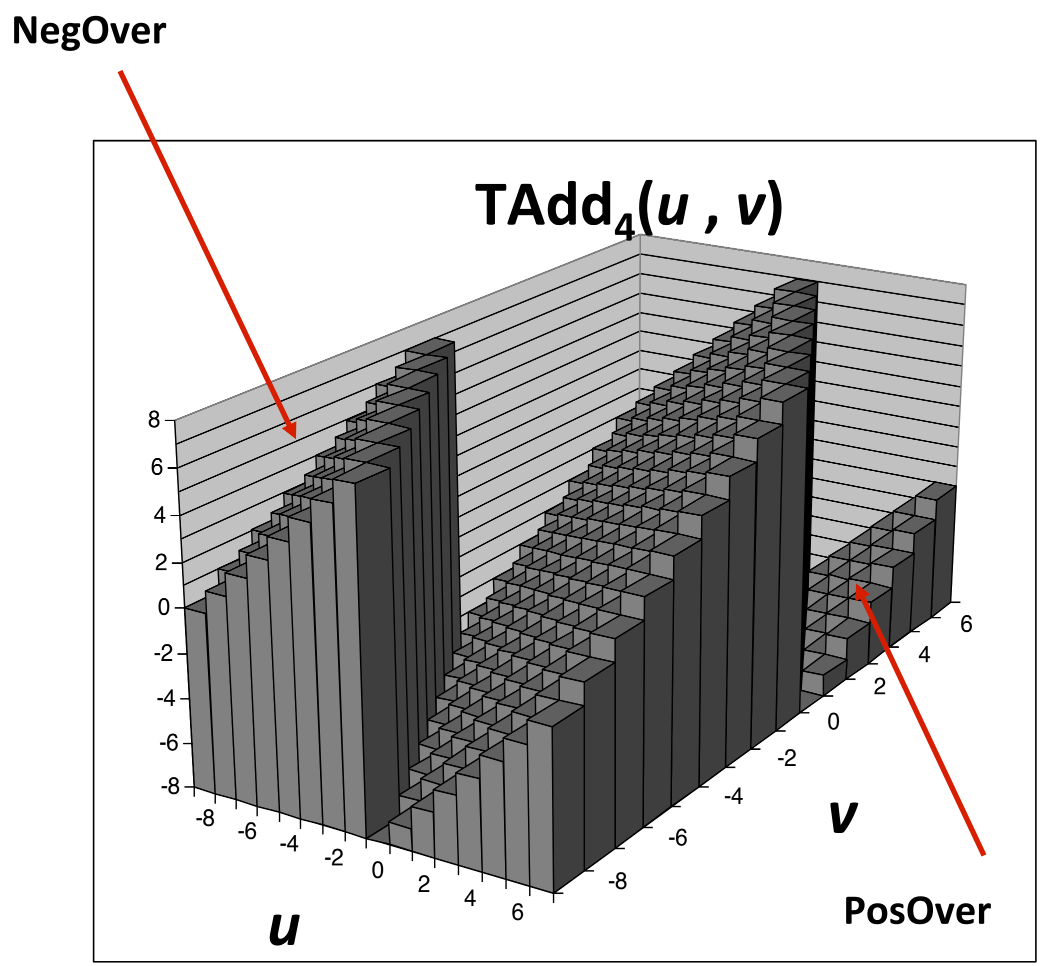
# TAdd Overflow

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



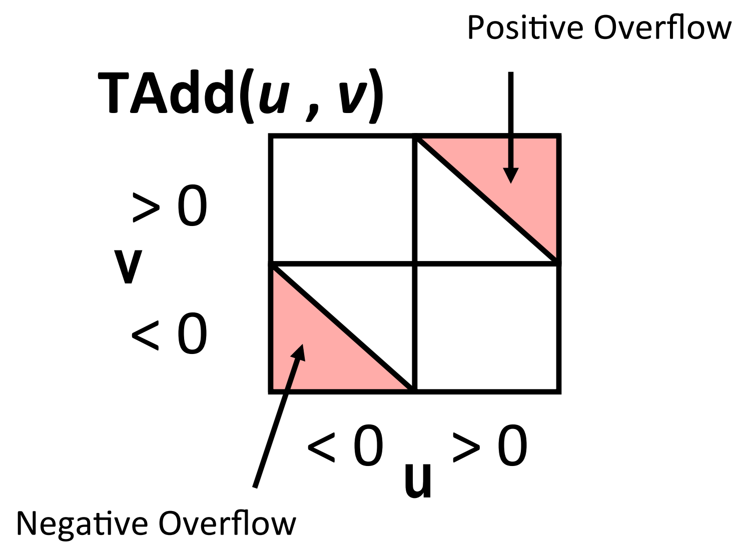
# Visualizing 2's Complement Addition

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




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

# Negation: Complement & Increment

- **Claim: Following Holds for 2's Complement**

  `~x + 1 == -x`

- **Complement**
  - Observation: `~x + x == 1111…111 == -1`

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | x | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| + | ~x | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| | -1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

- **Complete Proof?**

# Complement & Increment Examples

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 |

x = 0

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

# Multiplication

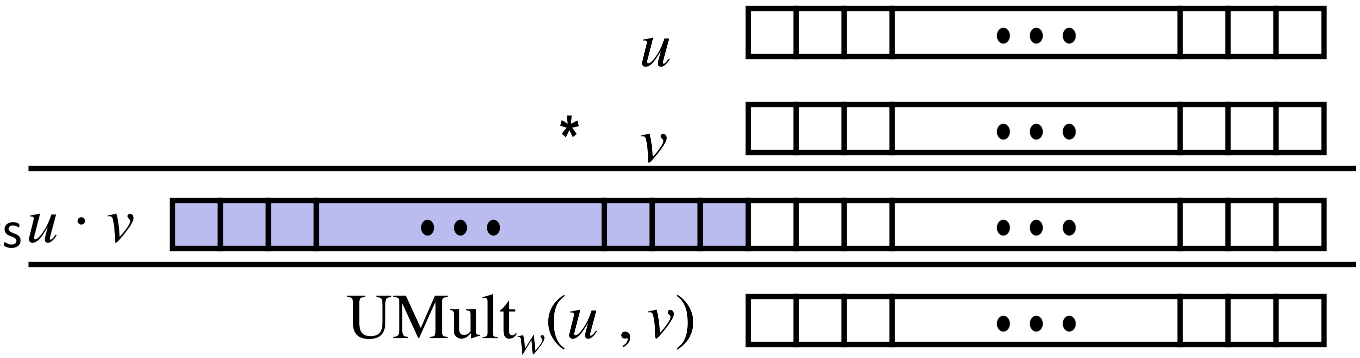
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^{w} - 1)^{2} = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_{w})^{2}$
    - Result range: $x * y \le (-2^{w-1})^{2} = 2^{2w-2}$
- **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

True Product: $2*w$ bits

Discard $w$ bits: $w$ bits



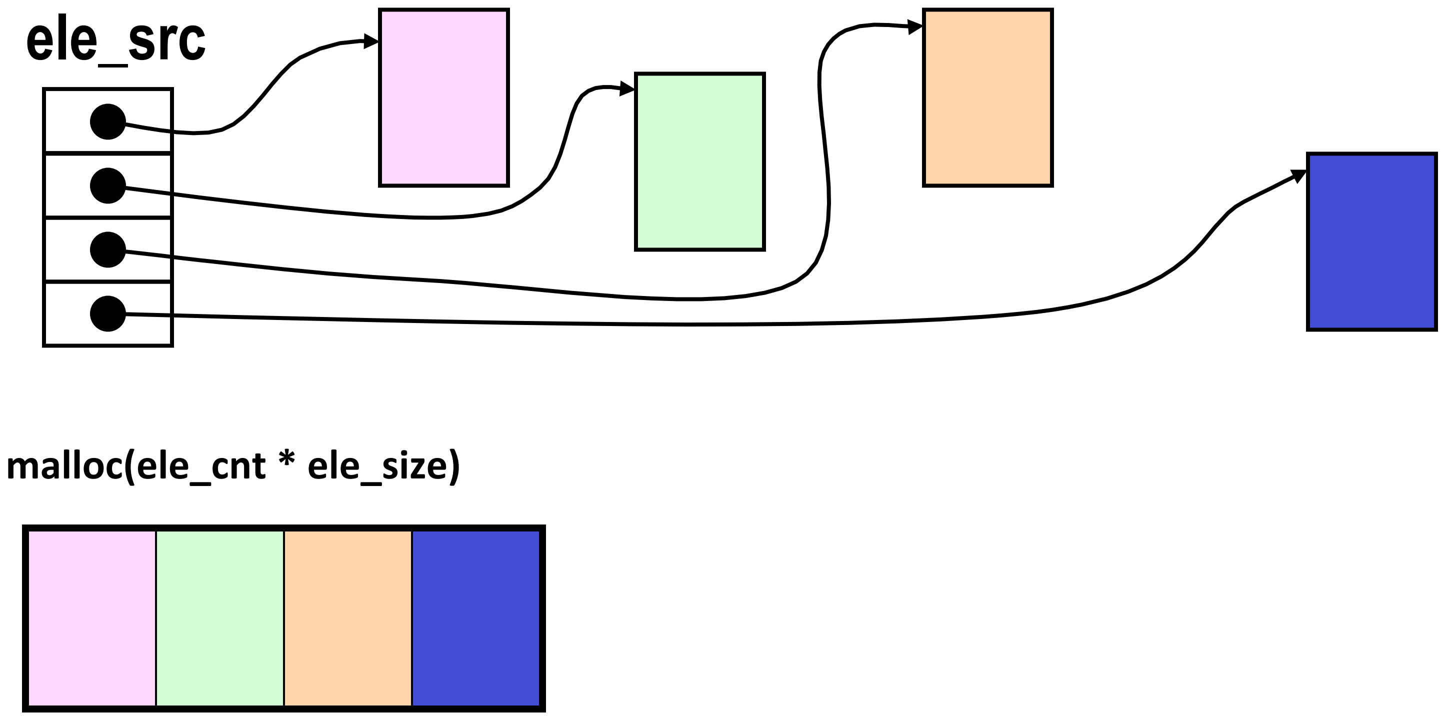

- **Standard Multiplication Function**
  - Ignores high order $w$ bits
- **Implements Modular Arithmetic**

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

# 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);
```



# 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 = ??

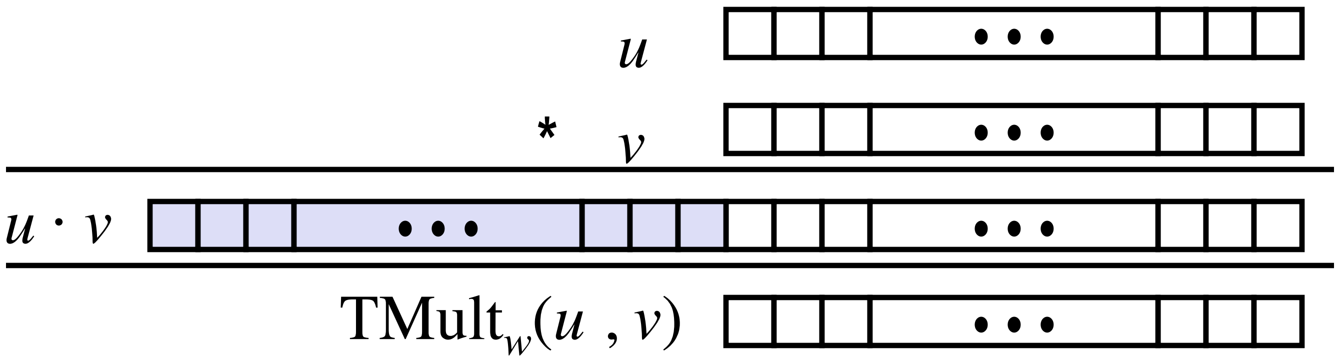
- **How can I make this function secure?**

# Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits




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

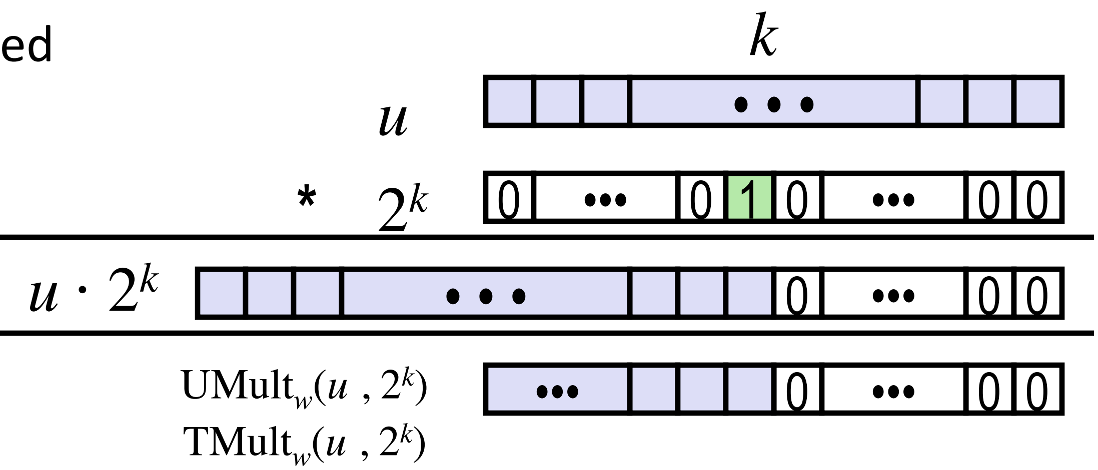
# Power-of-2 Multiply with Shift

- **Operation**
  - `u << k` gives `u * 2`$^k$
  - Both signed and unsigned

Operands: *w* bits

True Product: *w+k* bits

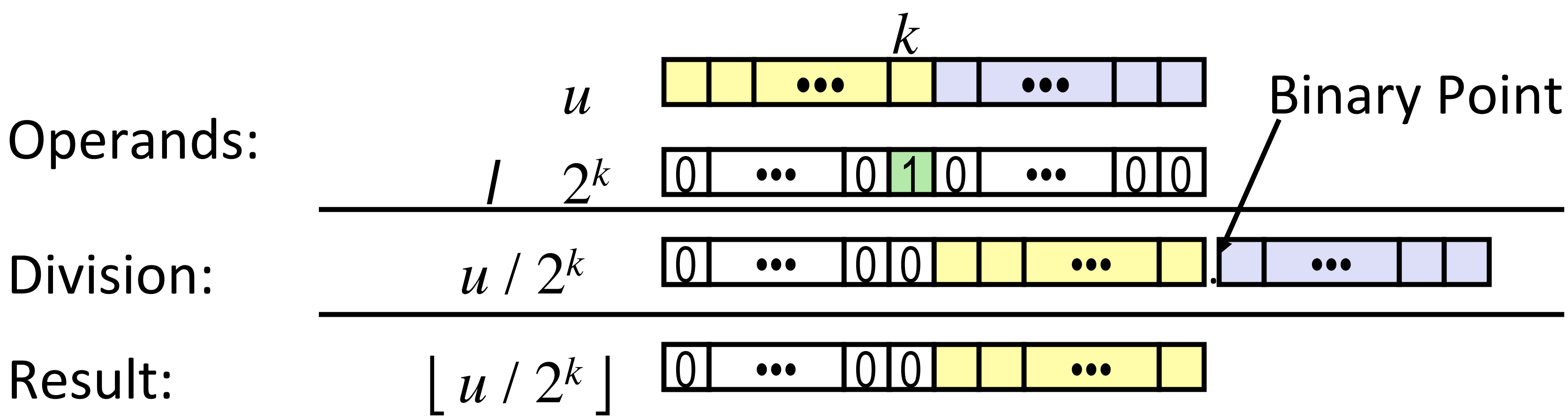
Discard *k* bits: *w* bits




- **Examples**
  - `u << 3          ==   u * 8`
  - `u << 5 - u << 3    ==     u * 24`
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

# Unsigned Power-of-2 Divide with Shift

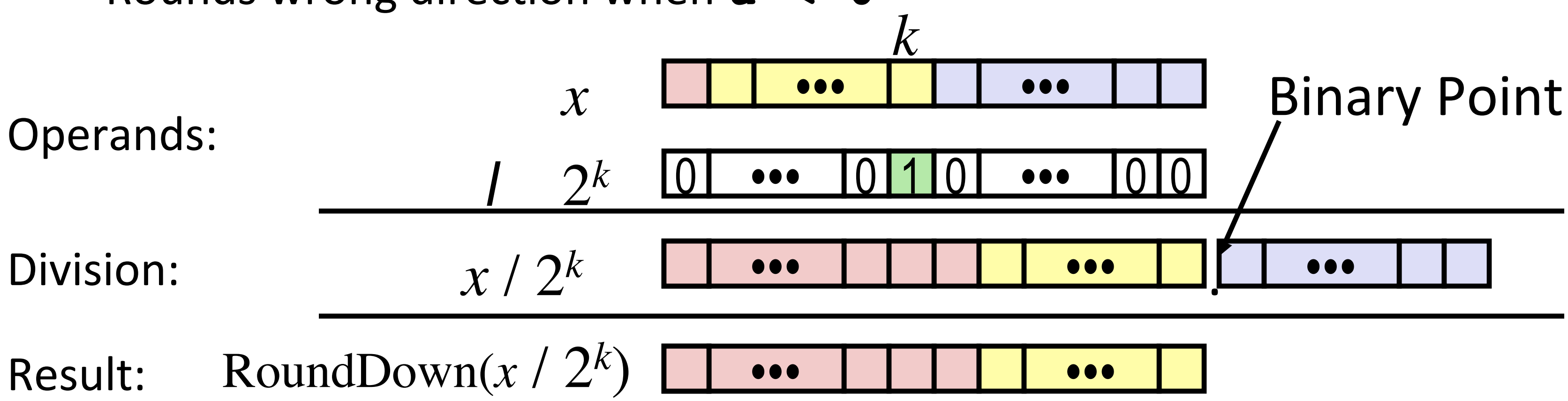
- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`$^k$ $\rfloor$
  - Uses logical shift




| | 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 >> k` gives $\lfloor \mathtt{x} / 2^k \rfloor$
  - 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` |

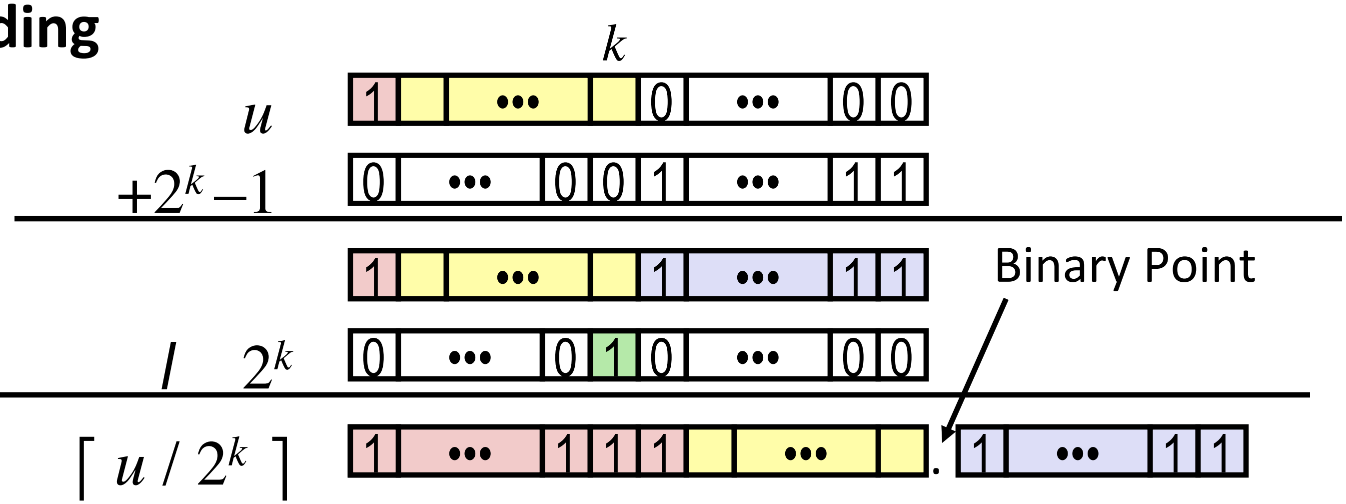
# Correct Power-of-2 Divide

- **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:

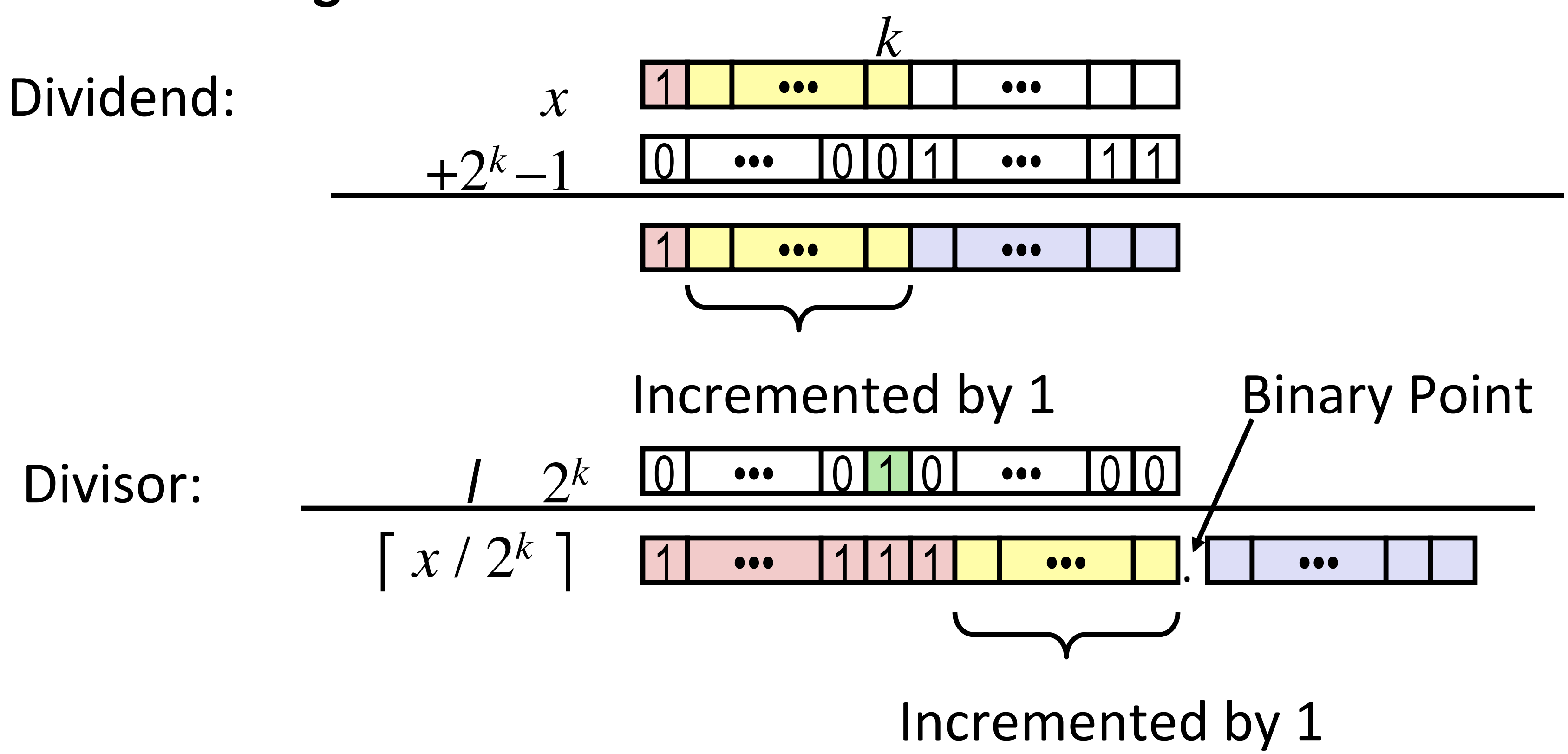
Divisor:




***Biasing has no effect***

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

## Case 2: Rounding




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

# 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 Just Because Number Nonnegative**
  - Easy to make mistakes

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

  - Can be very subtle

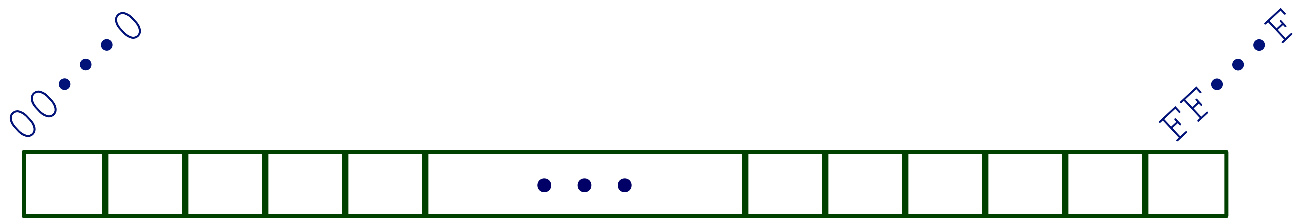
```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

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

# 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

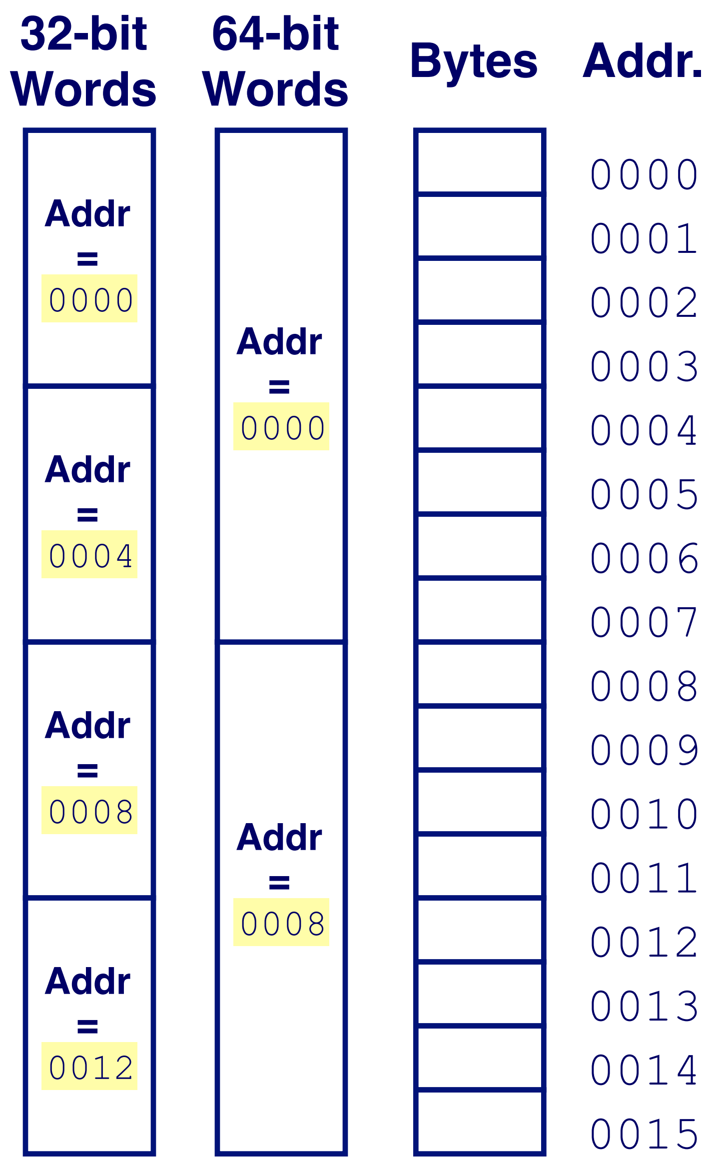
- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

# 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 PB (petabytes) of addressable memory
    - That's 18.4 X $10^{15}$
  - 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)



# For other data representations too …

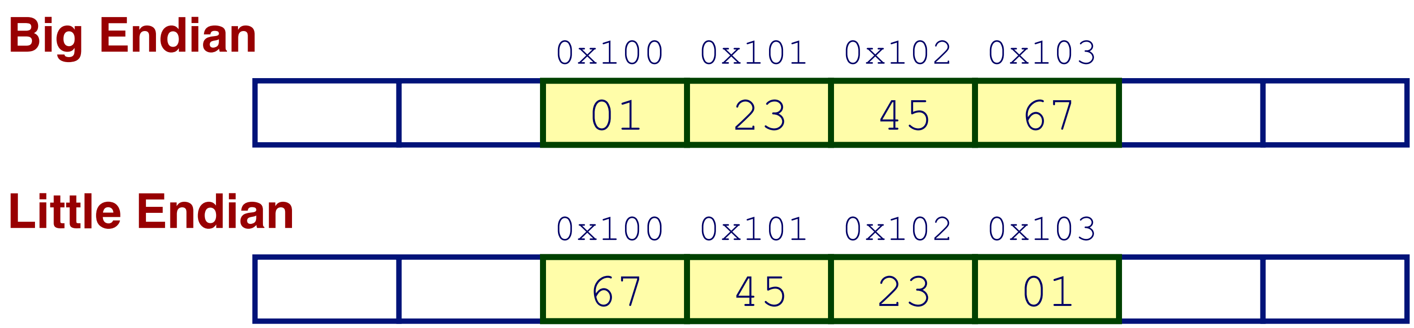
| C Data Type | Typical 32-bit | Intel IA32 | x86-64 |
|---|---|---|---|
| char | 1 | 1 | 1 |
| short | 2 | 2 | 2 |
| int | 4 | 4 | 4 |
| long | 4 | 4 | 8 |
| long long | 8 | 8 | 8 |
| float | 4 | 4 | 4 |
| double | 8 | 8 | 8 |
| long double | 8 | 10/12 | 10/16 |
| pointer | 4 | 4 | 8 |

# Byte Ordering

- **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

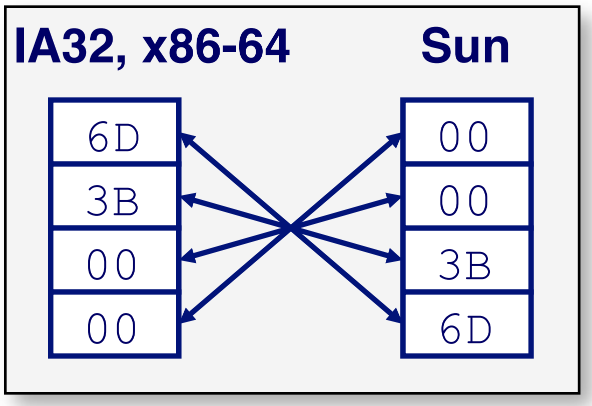
- **Example**
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100



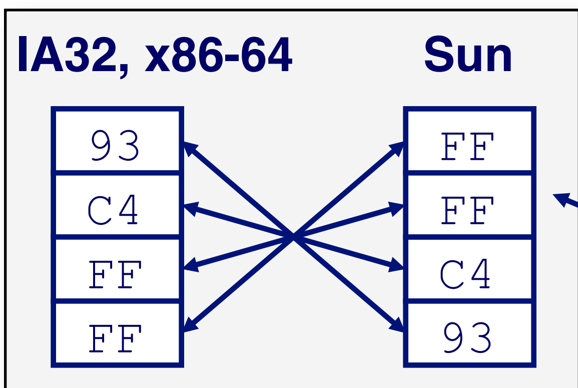
# Representing Integers

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

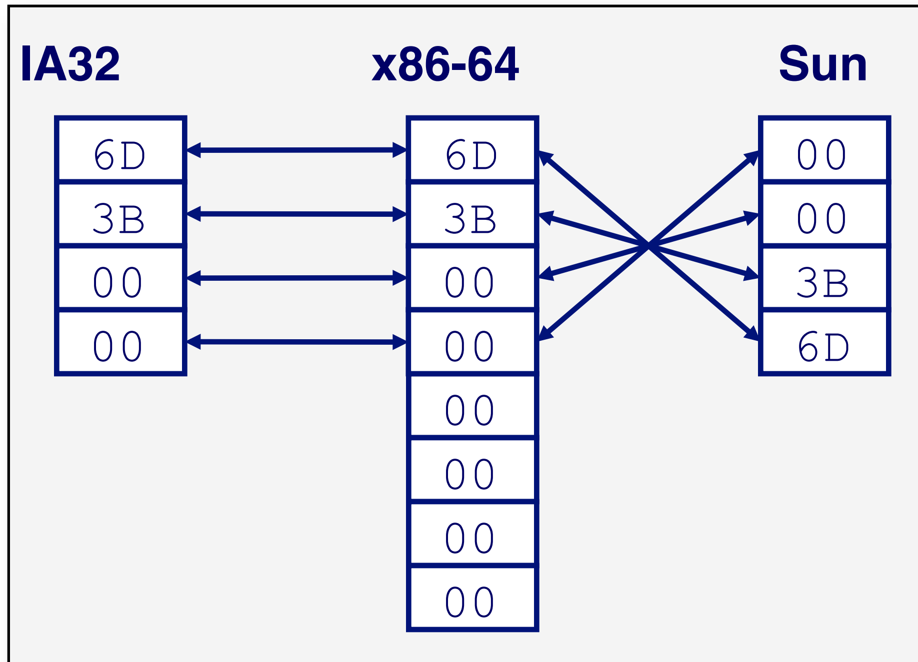
`int A = 15213;`




`int B = -15213;`




`long int C = 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, 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

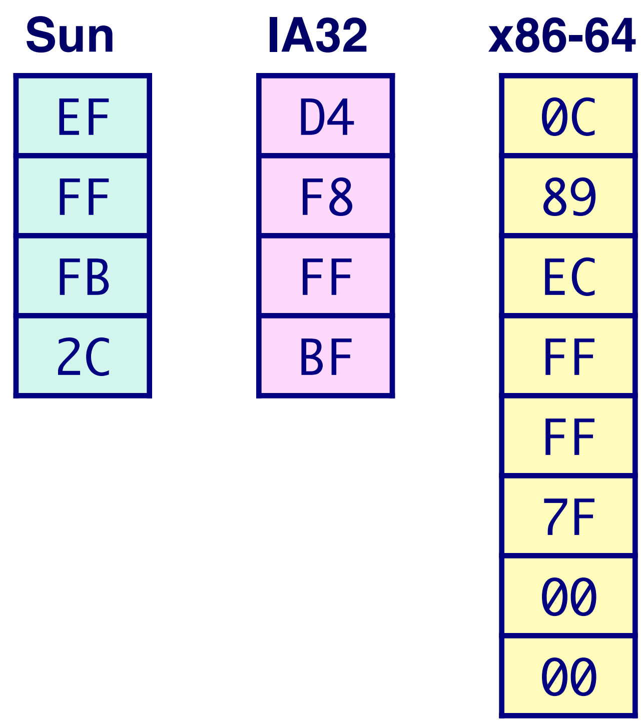
```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

**Result (Linux):**

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

# Representing Pointers

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

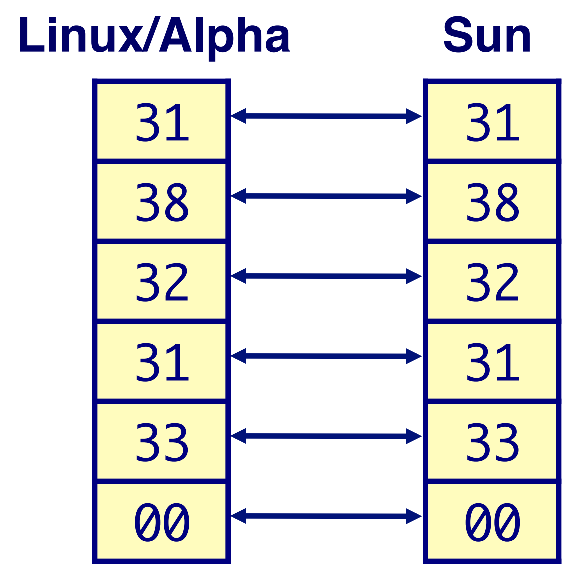



Different compilers & machines assign different locations to objects

# 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



# Integer C Puzzles

- x < 0 ⇒ ((x*2) < 0)
- ux >= 0
- x & 7 == 7 ⇒ (x<<30) < 0
- ux > -1
- x > y ⇒ -x < -y
- x * x >= 0
- x > 0 && y > 0 ⇒ x + y > 0
- x >= 0 ⇒ -x <= 0
- x <= 0 ⇒ -x >= 0
- (x|-x)>>31 == -1
- ux >> 3 == ux/8
- x >> 3 == x/8
- x & (x-1) != 0

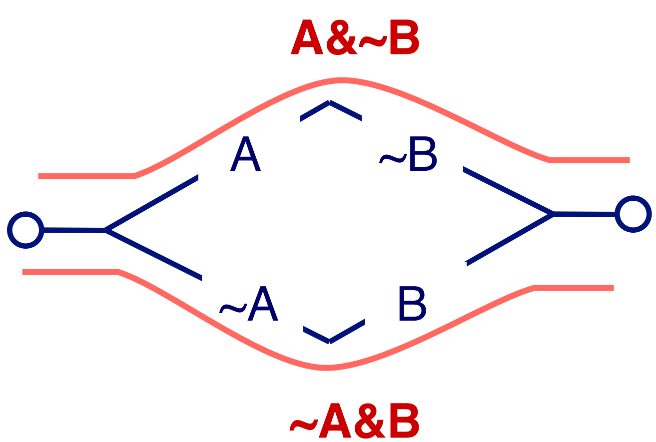
## Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

# Bonus extras

# 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&~B | ~A&B**

**= A^B**

# Mathematical Properties

- **Modular Addition Forms an *Abelian Group***
  - **Closed** under addition

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

    $\mathrm{UAdd}_w(u, v) = \mathrm{UAdd}_w(v, u)$
  - **Associative**

    $\mathrm{UAdd}_w(t, \mathrm{UAdd}_w(u, v)) = \mathrm{UAdd}_w(\mathrm{UAdd}_w(t, u), v)$
  - **0** is additive identity

    $\mathrm{UAdd}_w(u, 0) = u$
  - Every element has additive **inverse**
    - Let $\mathrm{UComp}_w(u) = 2^w - u$
      $\mathrm{UAdd}_w(u, \mathrm{UComp}_w(u)) = 0$

# Mathematical Properties of TAdd

- **Isomorphic Group to unsigneds with UAdd**
  - $TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$
    - Since both have identical bit patterns

- **Two's Complement Under TAdd Forms a Group**
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

# 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

# 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 >>>

# 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

- **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

# Properties of Unsigned Arithmetic

- **Unsigned Multiplication with Addition Forms Commutative Ring**
  - Addition is commutative group
  - Closed under multiplication

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

    $\text{UMult}_w(u\,,\,v) = \text{UMult}_w(v\,,\,u)$
  - Multiplication is Associative

    $\text{UMult}_w(t, \text{UMult}_w(u\,,\,v)) = \text{UMult}_w(\text{UMult}_w(t, u\,), v)$
  - 1 is multiplicative identity

    $\text{UMult}_w(u\,,\,1) = u$
  - Multiplication distributes over addtion

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

# Properties of Two's Comp. Arithmetic

- **Isomorphic Algebras**
  - Unsigned multiplication and addition
    - Truncating to *w* bits
  - Two's complement multiplication and addition
    - Truncating to *w* bits
- **Both Form Rings**
  - Isomorphic to ring of integers mod $2^w$
- **Comparison to (Mathematical) Integer Arithmetic**
  - Both are rings
  - Integers obey ordering properties, e.g.,

$$u > 0 \quad \Rightarrow \quad u + v > v$$

$$u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$$

  - These properties are not obeyed by two's comp. arithmetic

*TMax* `+ 1` `==` *TMin*

`15213 * 30426 == -10030` (16-bit words)

# 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