Integers

CS-392-A Systems Programming

Instructor:

Georgios Portokalidis – *Stevens Institute of Technology*

Integers

- **One-minute review of last week**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- **Representations in memory, pointers, strings**
- **Summary**

Bits and Bytes

- **Everything is bits**
  - Computers interpret them as numbers, instructions, characters, etc. based on conventions

- **General Boolean algebra**
  - Apply Boolean operators OR, AND, XOR, NOT as
    - Bitwise operators OR |, & AND, ^ XOR, complement ~
    - Values are treated as bit strings (not character strings!)
  - Separate from Boolean operators && AND, || OR, ! NOT
    - Where zero values → false, non-zero → true

- **Make sure to study the book chapters corresponding to each week's topics**
  - Last week it was chapters 1, 2.1

Integers

- **One-minute review of last week**

- **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

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

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

Sign Bit

- **C short 2 bytes long**

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

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

Two-complement Encoding Example (Cont.)

```
x =        15213: 00111011 01101101
y =       −15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	**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

Integers

- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- **Representations in memory, pointers, strings**

Mapping Between Signed & Unsigned

Two's Complement x → [T2B] X [B2U] → ux **Unsigned** (T2U)

Maintain Same Bit Pattern

Unsigned ux → [U2B] X [B2T] → x **Two's Complement** (U2T)

Maintain Same Bit Pattern
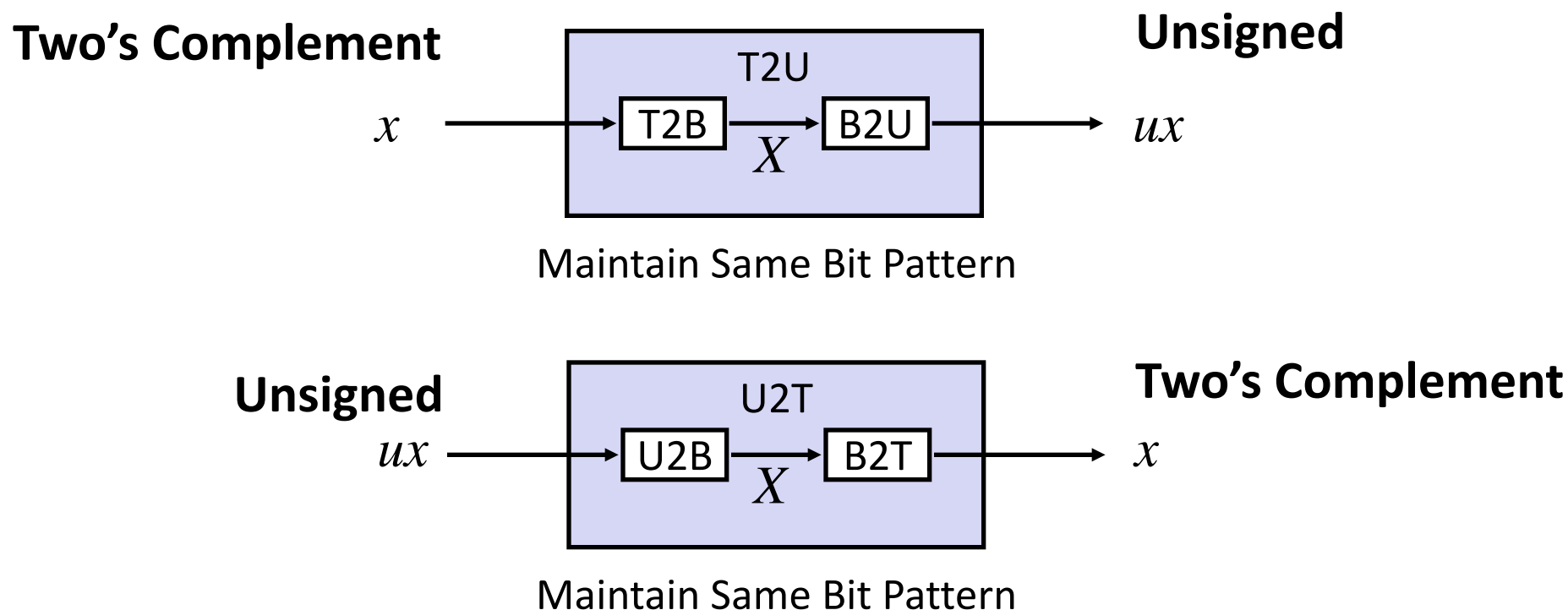
- **Mappings between unsigned and two's complement numbers:
Keep bit representations and reinterpret**

Mapping Signed ↔ Unsigned

Bits
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Signed
0
1
2
3
4
5
6
7
–8
–7
–6
–5
–4
–3
–2
–1

T2U

U2T

Unsigned
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

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**

T2U

x → [T2B] X → [B2U] → ux

Maintain Same Bit Pattern

$w{-}1$ 0

ux | + | + | + | • • • | + | + | + |

x | - | + | + | • • • | + | + | + |

↑
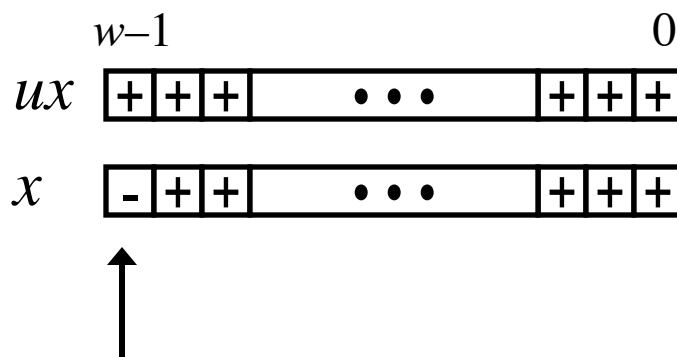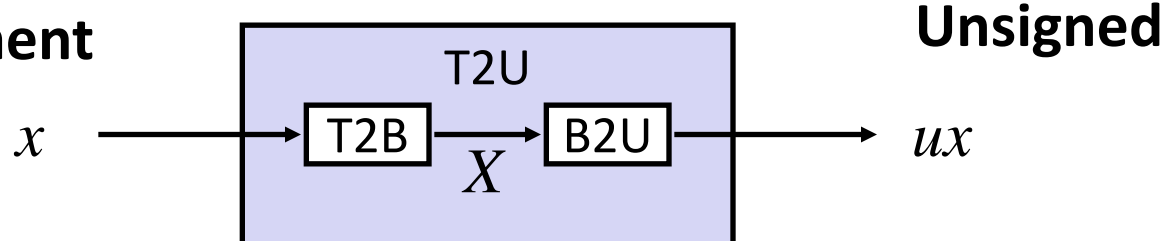
Large negative weight
becomes
Large positive weight

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive
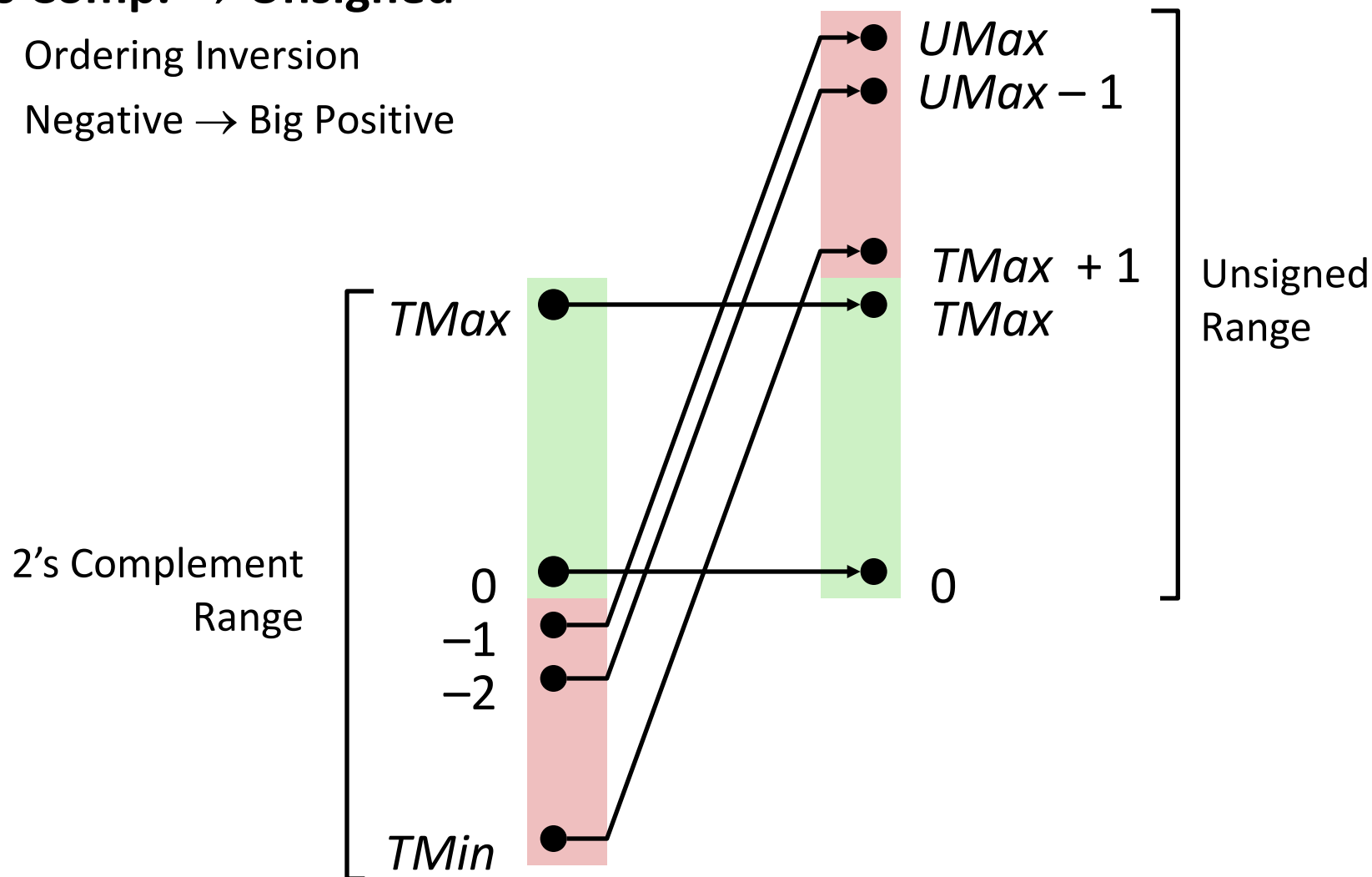
Signed vs. Unsigned in C

- **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**

- **Constant₁**

Constant$_1$	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

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`!!

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:
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$

k copies of MSB
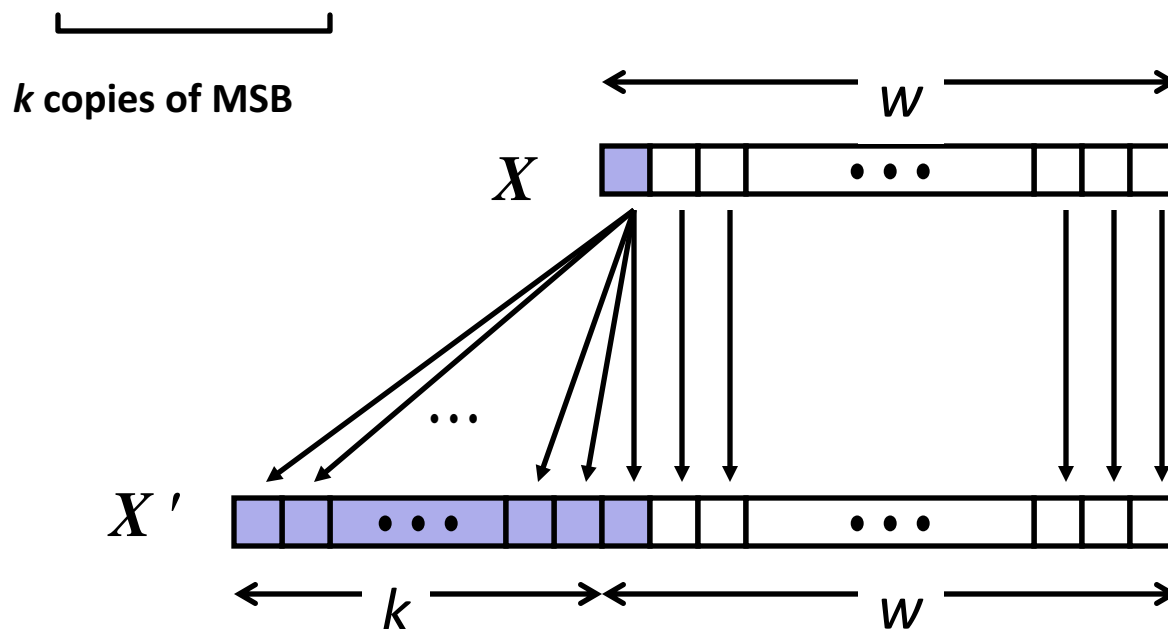
Sign Extension Example

```
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 behavior

Integers

- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- **Representations in memory, pointers, strings**
- **Summary**

Unsigned Addition

Operands: w bits

u

$+\ v$

True Sum: $w+1$ bits

$u + v$

Discard Carry: w bits

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

- **Standard Addition Function**

  - Ignores carry output

- **Implements Modular Arithmetic**

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

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

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

Integer Addition
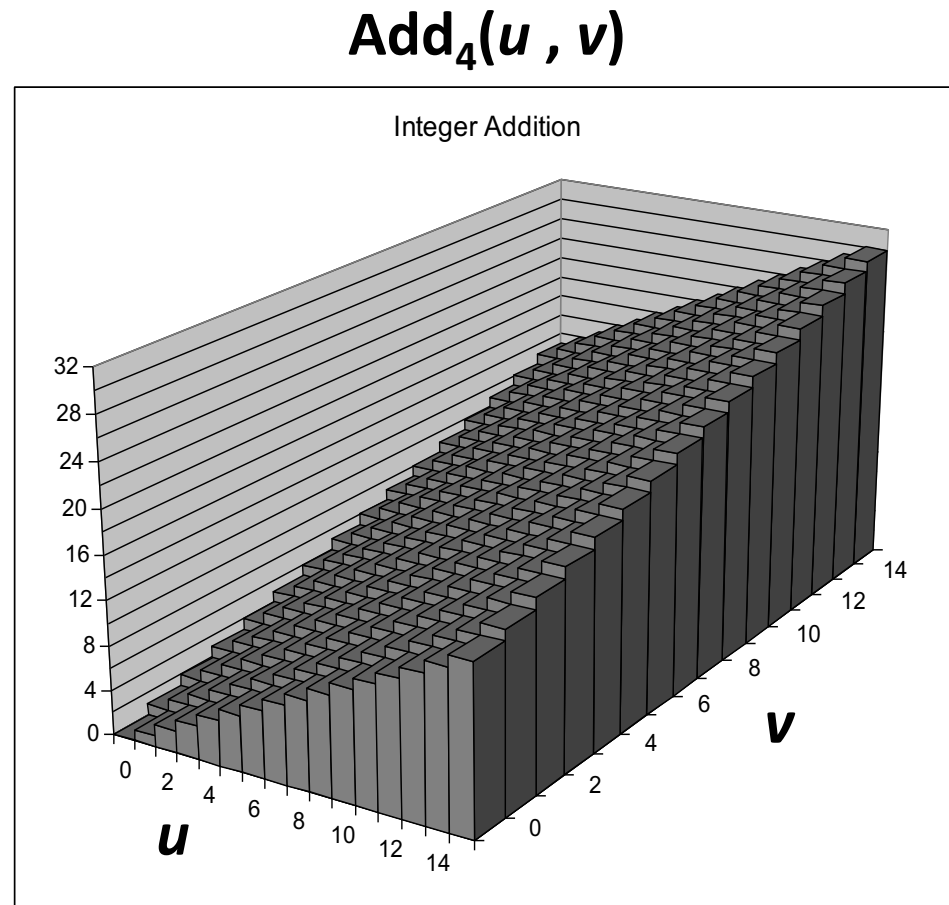
Visualizing Unsigned Addition

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

True Sum

2^{w+1}

Overflow

2^w

0

Modular Sum

Overflow

UAdd$_4$(u , v)

v

u
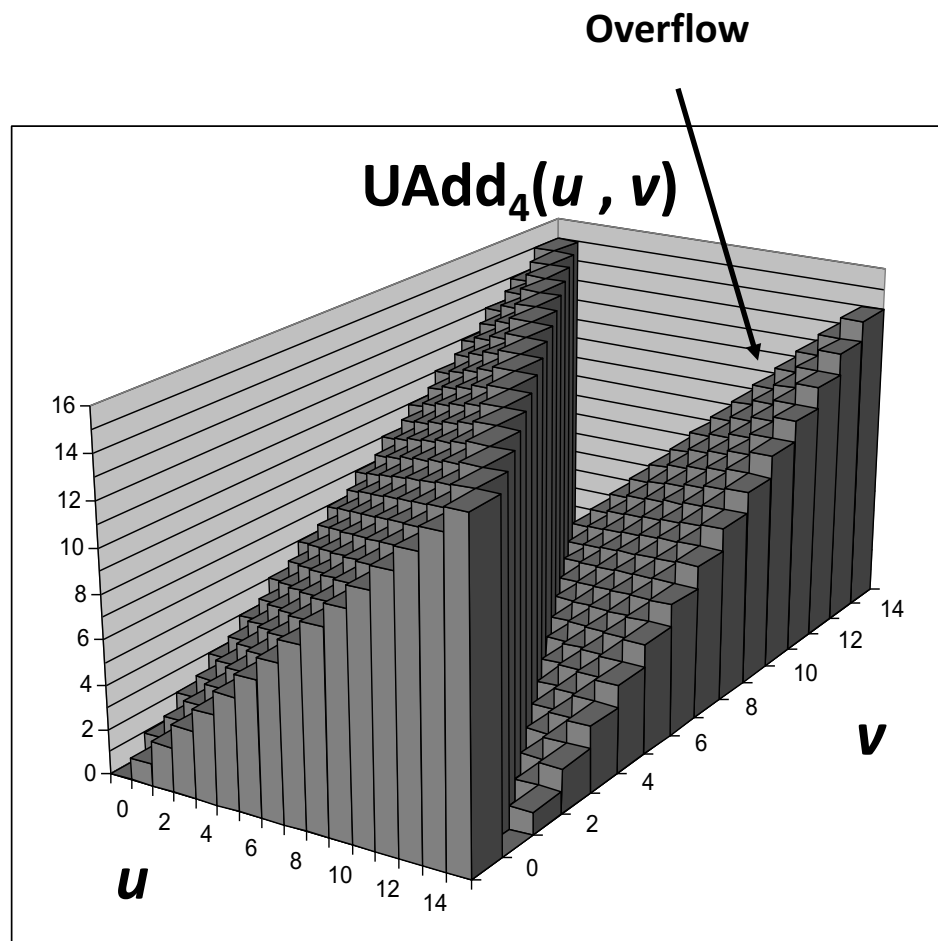
Two's Complement Addition

Operands: w bits

u

$+ \quad v$

True Sum: $w+1$ bits

$u + v$
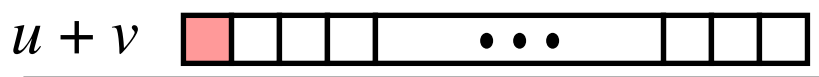
Discard Carry: w bits

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

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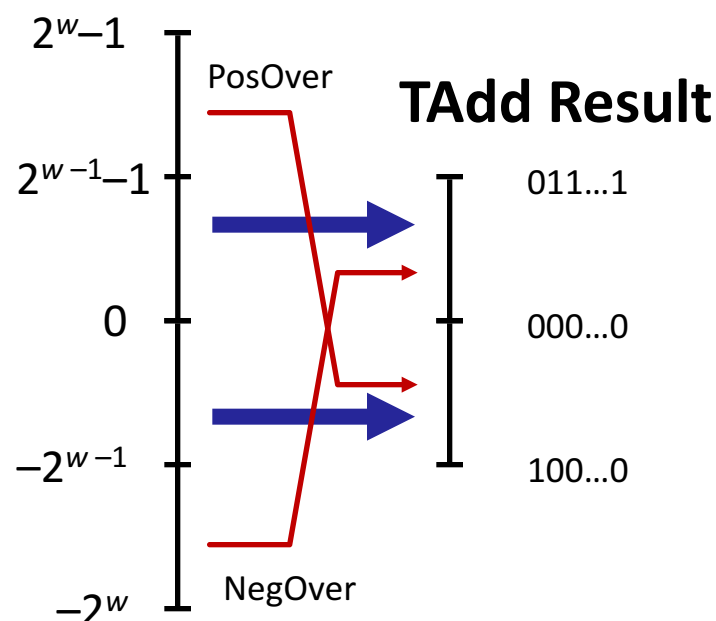 - True sum requires $w+1$ bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer

True Sum

0 111…1 2^w-1

0 100…0 $2^{w-1}-1$

0 000…0 0

1 011…1 -2^{w-1}

1 000…0 -2^w

PosOver

TAdd Result

011…1

000…0

100…0

NegOver

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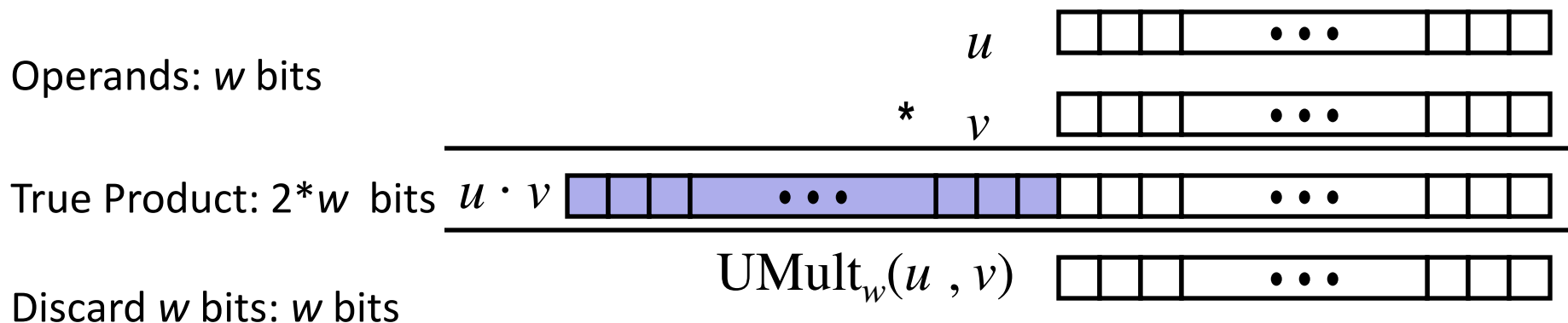 - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once

Negative Overflow

TAdd$_4$(u , v)

u

v

Positive Overflow
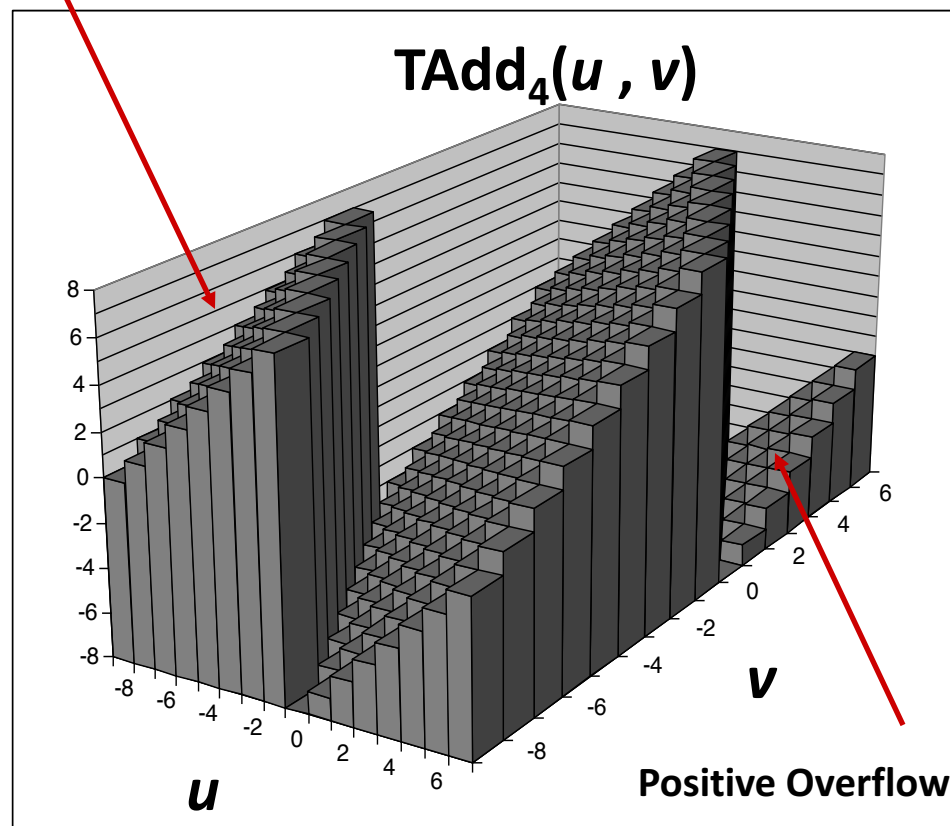
Multiplication

- **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
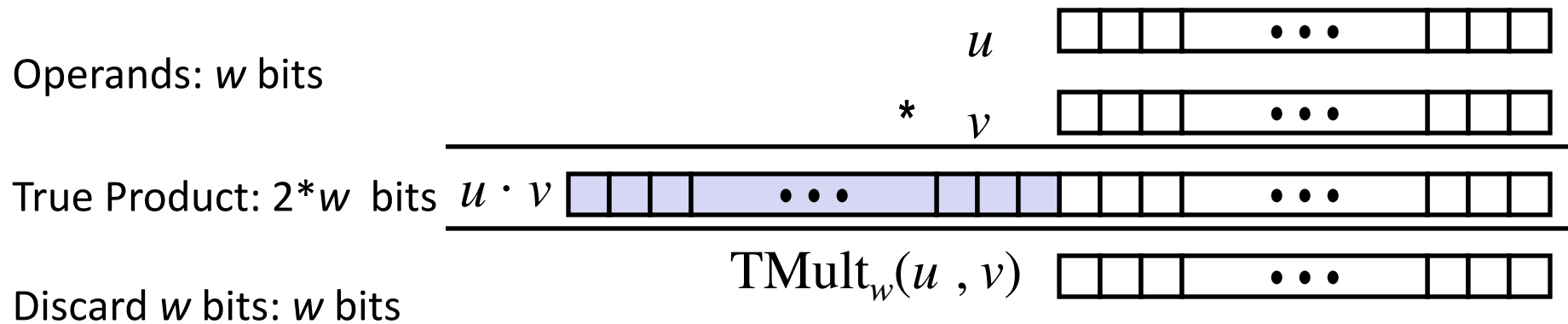
Unsigned Multiplication in C

Operands: *w* bits

u

$*$ v

True Product: 2*w bits $u \cdot v$

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

Discard *w* bits: *w* bits

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

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

Signed Multiplication in C

Operands: w bits

u

$*$ v

True Product: $2*w$ bits $u \cdot v$

Discard w bits: w bits

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

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

Two's-Complement Negation

- **Every element u has additive inverse x, where x + u = 0**
  - It's not always its opposite

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

Negation: Complement & Increment

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

    `~x + 1 == -x`

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

		1	0	0	1	1	1	0	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`

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

Power-of-2 Multiply with Shift

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

k

Operands: w bits

u

$* \quad 2^k$
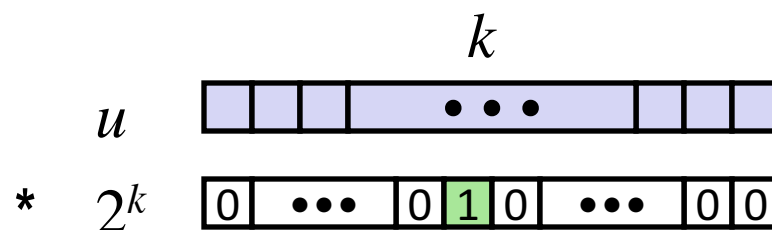
True Product: $w+k$ bits $\quad u \cdot 2^k$

Discard k bits: w bits $\quad$ $\text{UMult}_w(u, 2^k)$
$\text{TMult}_w(u, 2^k)$
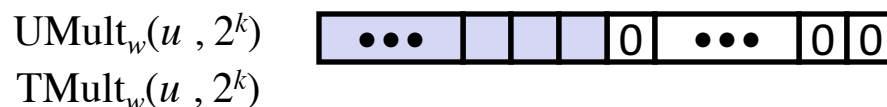
- **Examples**
  - ▪ `u << 3         ==    u * 8`
  - ▪ `(u << 5) – (u << 3) ==     u * 24`
  - ▪ Most machines shift and add faster than multiply
    - ▪ Compiler generates this code automatically

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

k

u

Operands:

$/\ \ 2^k$

Binary Point

Division: $u / 2^k$

Result: $\lfloor u / 2^k \rfloor$
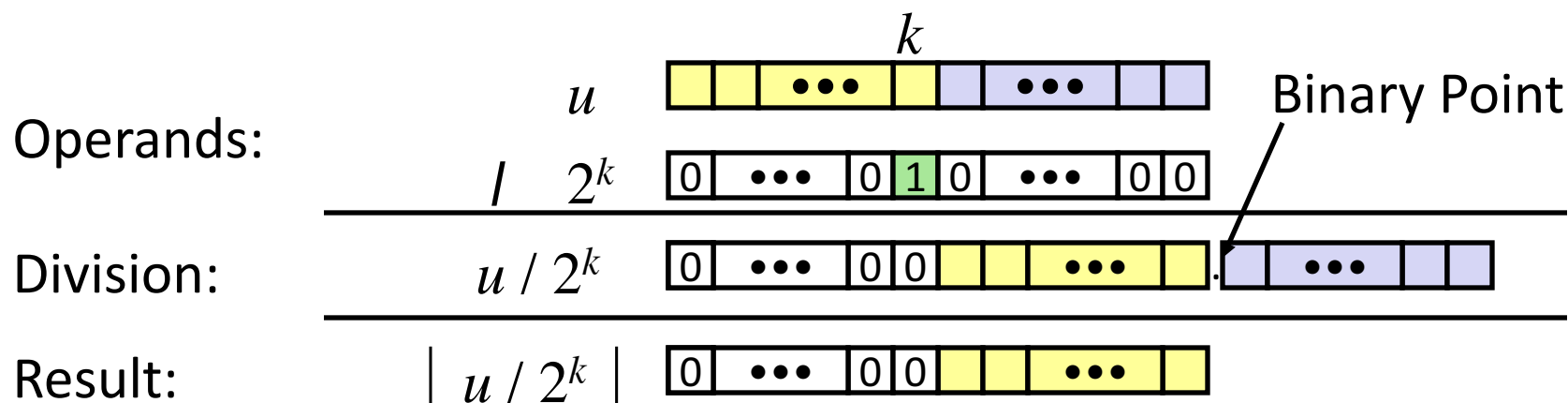
	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

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	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Integers

- **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.)

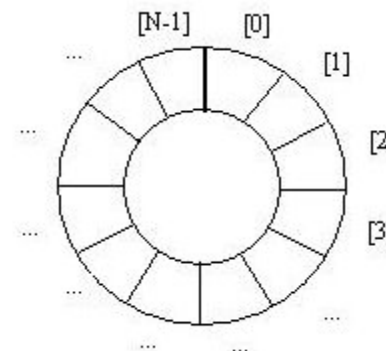- *Do* **Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic

- *Do* **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

- **Example: Ring buffers or circular buffers**
  - N is UINT_MAX, USHORT_MAX, or UCHAR_MAX
  - Or N is a power of 2 and access index is masked with (N-1)
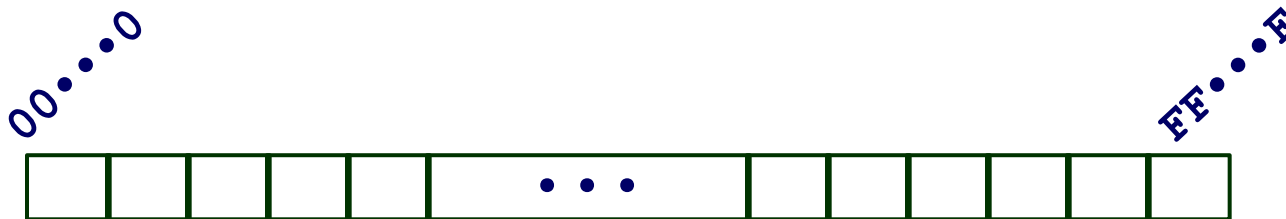   - N = 512, index & 0x1FF (1 1111 1111)

Integers

- **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
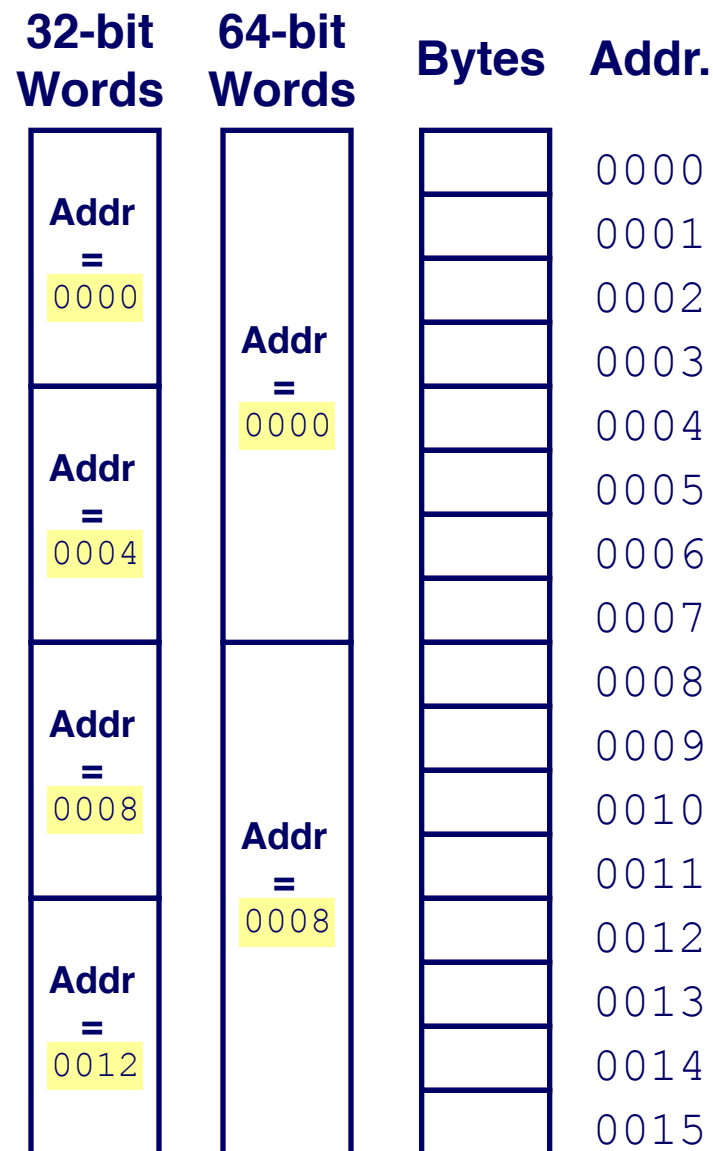
Machine Words

- **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 X 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)

32-bit Words	64-bit Words	Bytes	Addr.
Addr = 0000	**Addr =** 0000		0000
			0001
			0002
			0003
Addr = 0004			0004
			0005
			0006
			0007
Addr = 0008	**Addr =** 0008		0008
			0009
			0010
			0011
Addr = 0012			0012
			0013
			0014
			0015

Example Data Representations

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

Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**

- **Conventions**
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
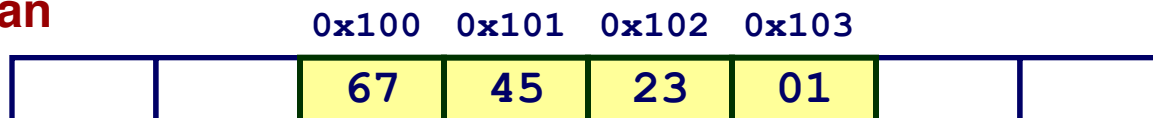   - Least significant byte has lowest address

Byte Ordering Example

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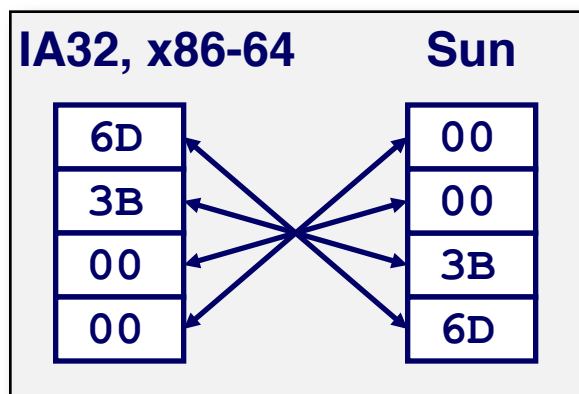 - Address given by &x is 0x100

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		
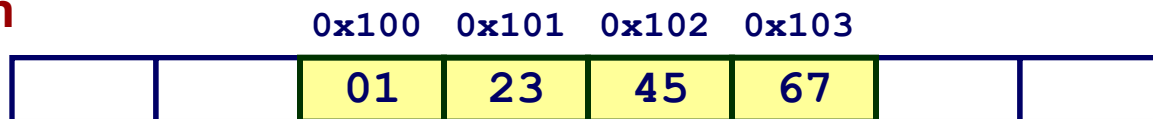
Representing Integers

Decimal: `15213`

Binary: `0011 1011 0110 1101`

Hex: `    3    B    6    D`

int A = 15213;

IA32, x86-64		Sun
6D		00
3B		00
00		3B
00		6D

long int C = 15213;

IA32	x86-64	Sun
6D	6D	00
3B	3B	00
00	00	3B
00	00	6D
	00	
	00	
	00	
	00	

int B = -15213;
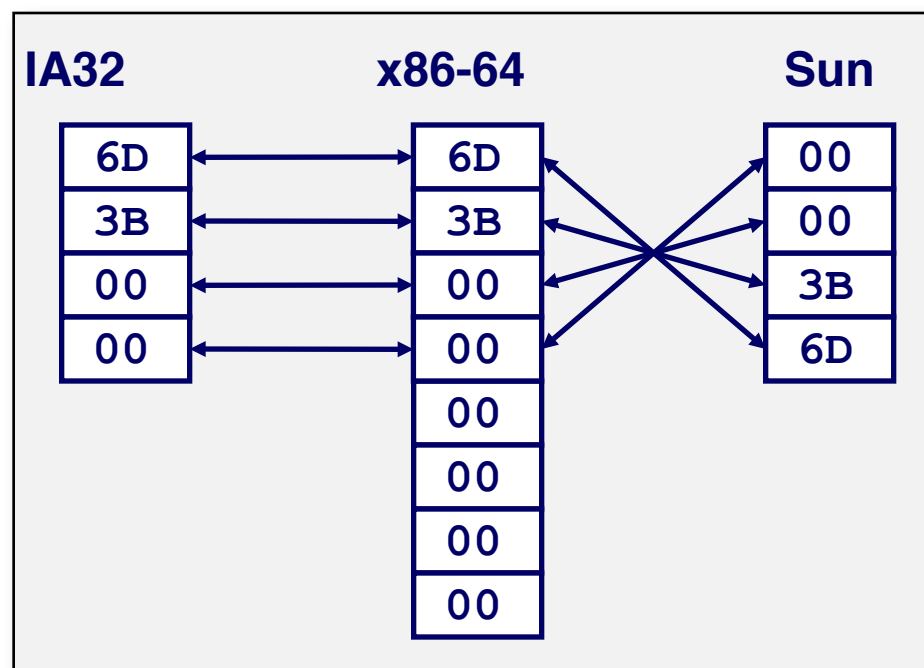
IA32, x86-64		Sun
93		FF
C4		FF
FF		C4
FF		93
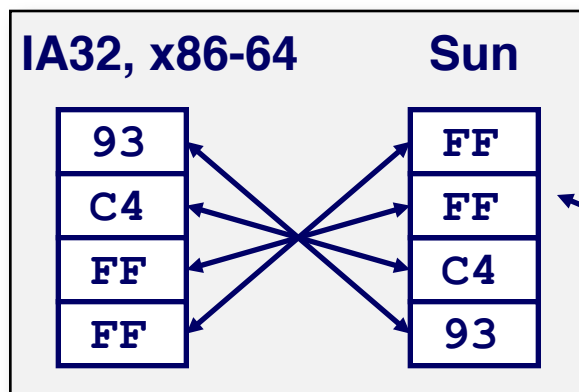
Two's complement representation

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

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_t;

void show_bytes(pointer_t 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;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

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

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

- **Different compilers & machines assign different locations to objects**
- **In modern systems you even get different results each time you run program**

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
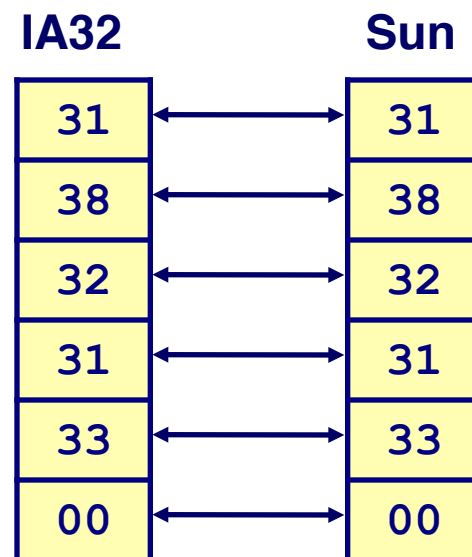
Representing Strings

```
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

IA32		Sun
31	⟷	31
38	⟷	38
32	⟷	32
31	⟷	31
33	⟷	33
00	⟷	00

Integer C Puzzles

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

Integer C Puzzles

1. `x < 0` $\Rightarrow$ `((x*2) < 0)`
2. `ux >= 0`
3. `x & 7 == 7` $\Rightarrow$ `(x<<30) < 0`
4. `ux > -1`
5. `x > y` $\Rightarrow$ `-x < -y`
6. `x * x >= 0`
7. `x > 0 && y > 0` $\Rightarrow$ `x + y > 0`
8. `x >= 0` $\Rightarrow$ `-x <= 0`
9. `x <= 0` $\Rightarrow$ `-x >= 0`
10. `(x|-x)>>31 == -1`
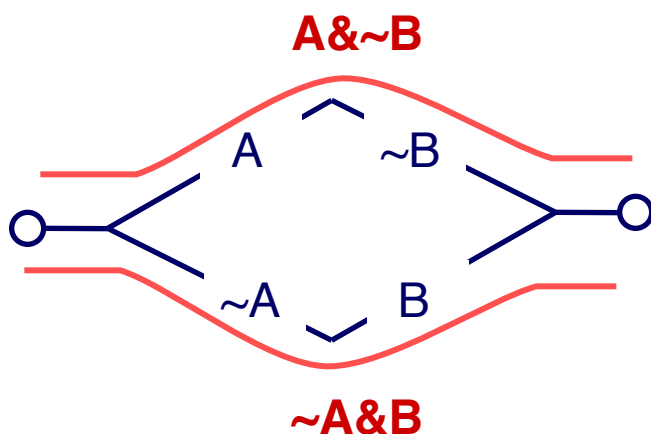11. `ux >> 3 == ux/8`
12. `x >> 3 == x/8`
13. `x & (x-1) != 0`

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**
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

- **Exclusive-OR Function (XOR)**

A&~B

~A&B

Connection when

A&~B | ~A&B

= A^B

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;
}
```
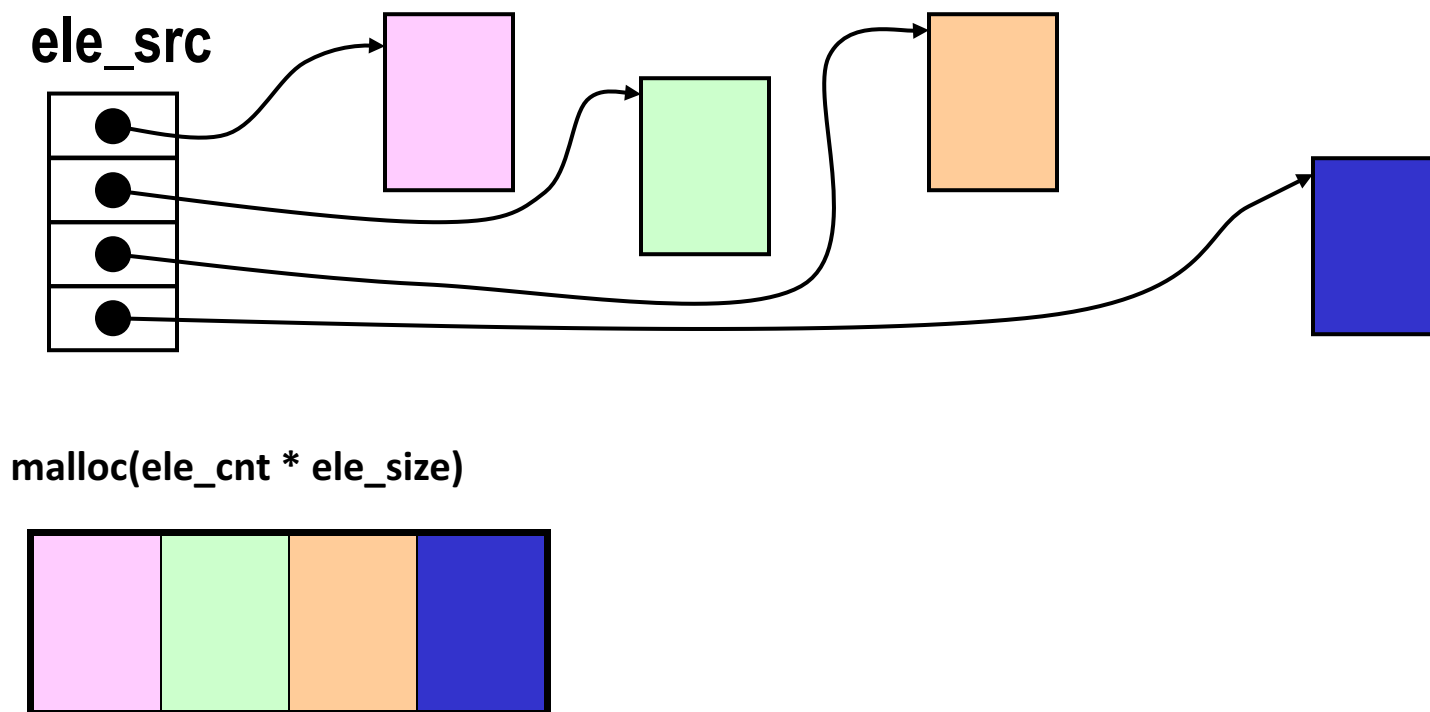
```
#define MSIZE 528

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

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);
```


ele_src

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 Code

What if:

$$\text{ele\_cnt} \quad = 2^{20} + 1$$
$$\text{ele\_size} \quad = 4096 \quad\quad = 2^{12}$$
$$\text{Allocation} = \text{??}$$

```
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;
}
```

How Can I Make This Function Secure?

- **https://locklessinc.com/articles/calloc/**

- **Use a larger data type (if there is one)**

```
__uint128_t sz = (__uint128_t)ele_cnt * ele_size;
/* Overflow? */
if (sz >> 64) return -1;
```

- **Check with division (what I would do)**

```
size_t sz = ele_cnt * ele_size;
/* Overflow? */
If (sz / ele_cnt != ele_size) return -1;
```

How Can I Make This Function Secure?

- **https://locklessinc.com/articles/calloc/**

- **Use a larger data type (if there is one)**

```
__uint128_t sz = (__uint128_t)ele_cnt * ele_size;
/* Overflow? */
if (sz >> 64) return -1;
```

- **Check with division (what I would do)**

```
size_t sz = ele_cnt * ele_size;
/* Overflow? */
If (ele_cnt != 0 && sz / ele_cnt != ele_size) return -1;
```

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

```
leaq (%rax,%rax,2), %rax
salq $2, %rax
```

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

Compiled Arithmetic Operations

```
  shrq $3, %rax
```
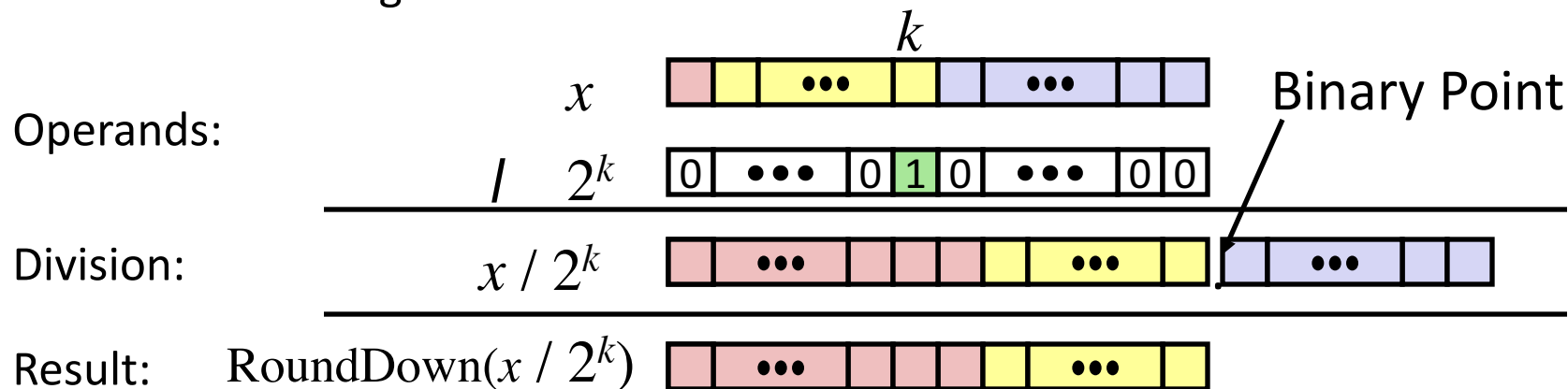
Explanation

```
# 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**
  - $x \text{ >> } k$ gives $\lfloor x \text{ / } 2^k \rfloor$
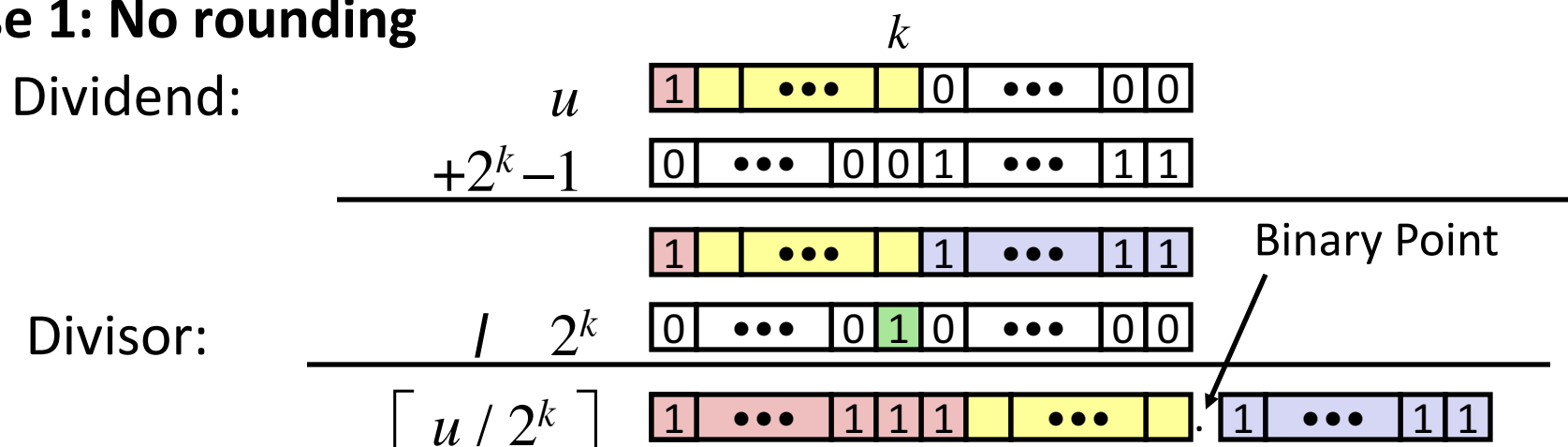 - Uses arithmetic shift
  - Rounds wrong direction when $u \text{ < } 0$

	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`1``1100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`1111``1100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- **Quotient of Negative Number by Power of 2**
  - Want $\lceil \texttt{x / 2}^k \rceil$ (Round Toward 0)
  - Compute as $\lfloor \texttt{(x+2}^k\texttt{–1)/ 2}^k \rfloor$
    - In C: `(x + (1<<k)–1) >> k`
    - Biases dividend toward 0

Case 1: No rounding

k

Dividend: u

| 1 | | ••• | | 0 | ••• | | 0 | 0 |

$+2^k-1$

| 0 | ••• | | 0 | 0 | 1 | ••• | | 1 | 1 |

| 1 | | ••• | | 1 | ••• | | 1 | 1 |

Binary Point

Divisor: $/\ 2^k$

| 0 | ••• | | 0 | 1 | 0 | ••• | | 0 | 0 |

$\lceil u\ /\ 2^k \rceil$

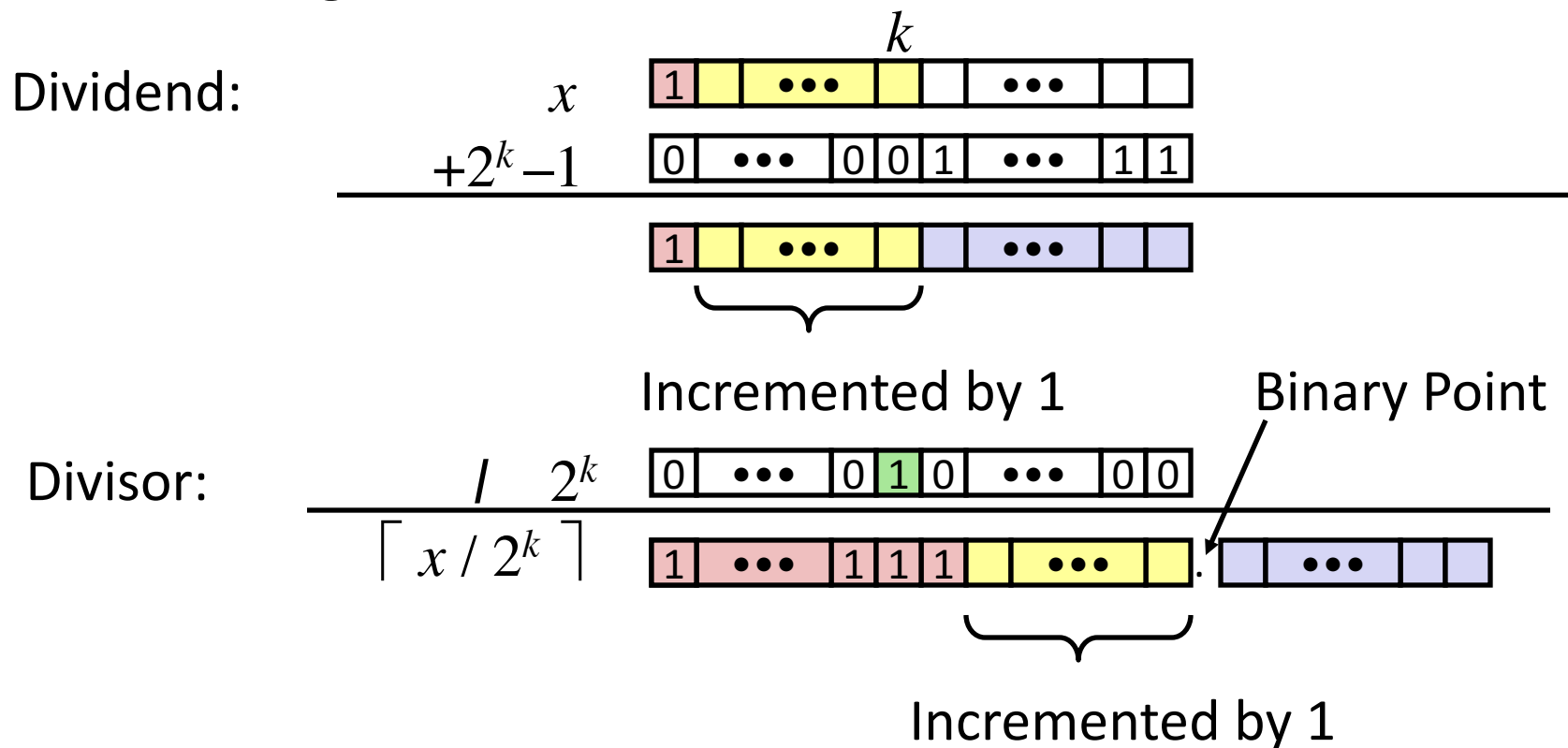| 1 | ••• | | 1 | 1 | 1 | | ••• | | . | 1 | ••• | | 1 | 1 |

Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Biasing adds 1 to final result

Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

```
   testq %rax, %rax
   js    L4
L3:
   sarq $3, %rax
   ret
L4:
   addq $7, %rax
   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 >>

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

Reading

- **Sections 2.2-2.3 from the Computer Systems book**