Bits, Bytes, and Integers (1-2)

15-213/18-213/15-513: Introduction to Computer Systems

2nd Lecture, 24 May 2012

Instructors:

Gregory Kesden

Last Time: Course Overview

- **Course Theme:**

Abstraction Is Good But Don't Forget Reality

- **5 Great Realities**
  - Ints are not Integers, Floats are not Reals
  - You've Got to Know Assembly
  - Memory Matters
  - There's more to performance than asymptotic complexity
  - Computers do more than execute programs
- **Administrative / Logistics details**

Today: Bits, Bytes, and Integers (1-2)

- ■ **Representing information as bits**
- ■ **Bit-level manipulations**
- ■ **Summary**

Binary Representations
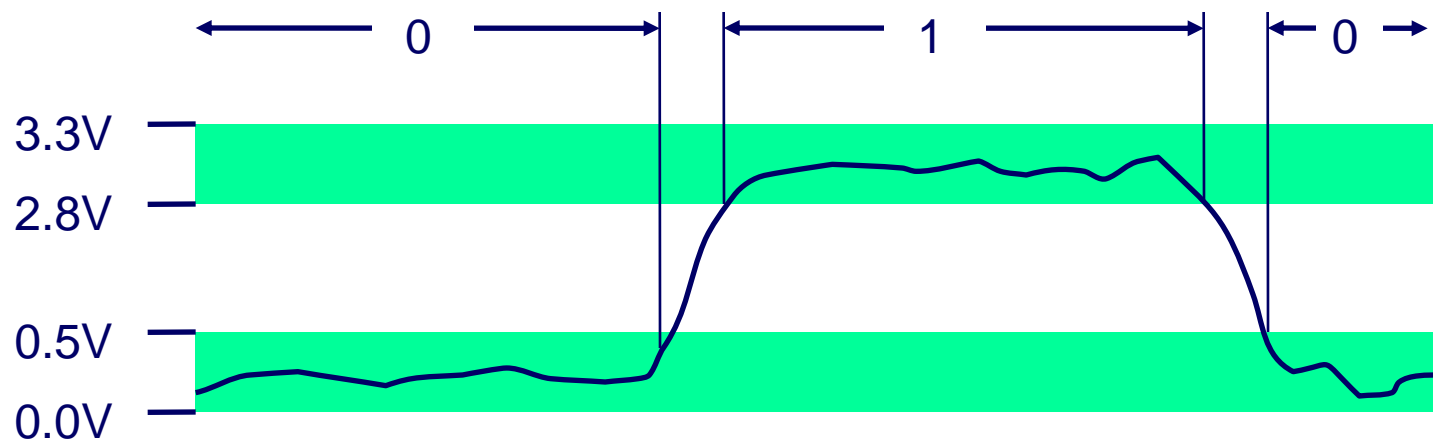
Encoding Byte Values

- **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

Literary Hex

- **Common 8-byte hex fillers:**
  - 0xdeadbeef
  - 0xc0ffeeee
  - Can you think of other 8-byte fillers?

Byte-Oriented Memory Organization
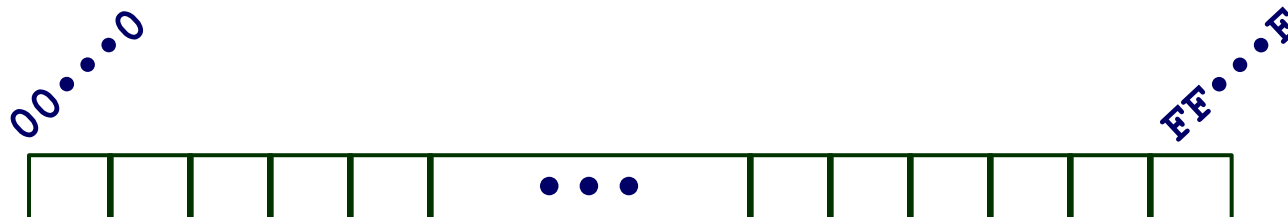
■ **Programs Refer to Virtual Addresses**

- ▪ 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
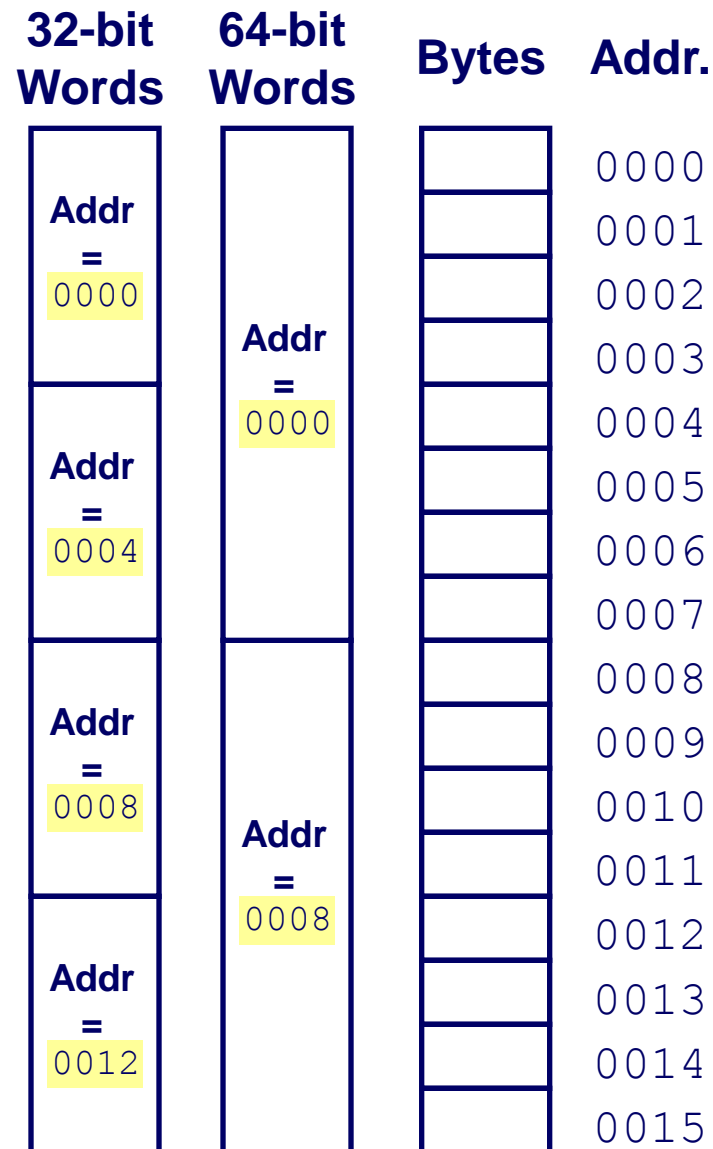
Machine Words

- **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 ≈ 1.8×10^{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)

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

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**
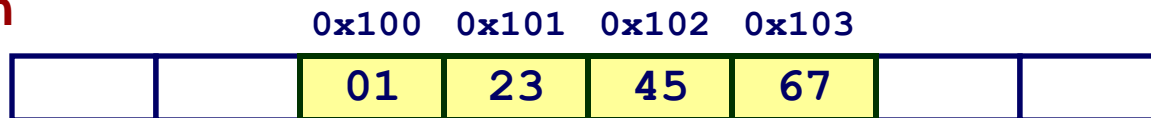 - Least significant byte has highest address
- **Little Endian**
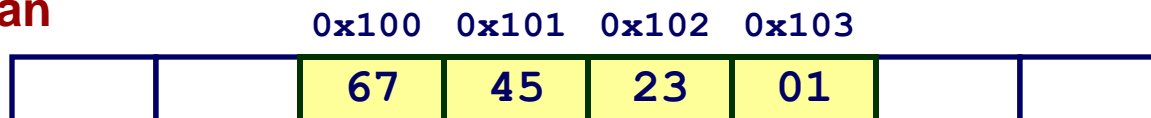 - Least significant byte has lowest address
- **Example**
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

Big Endian

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

Little Endian

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

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
```
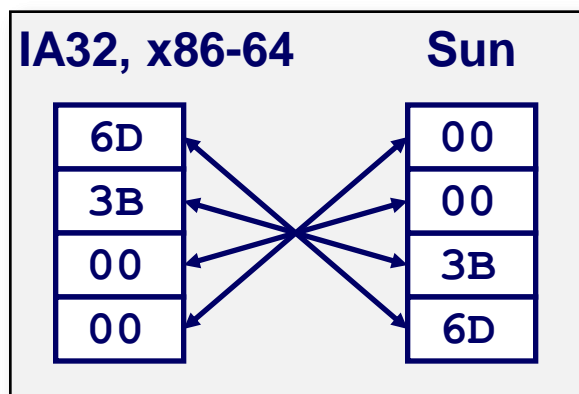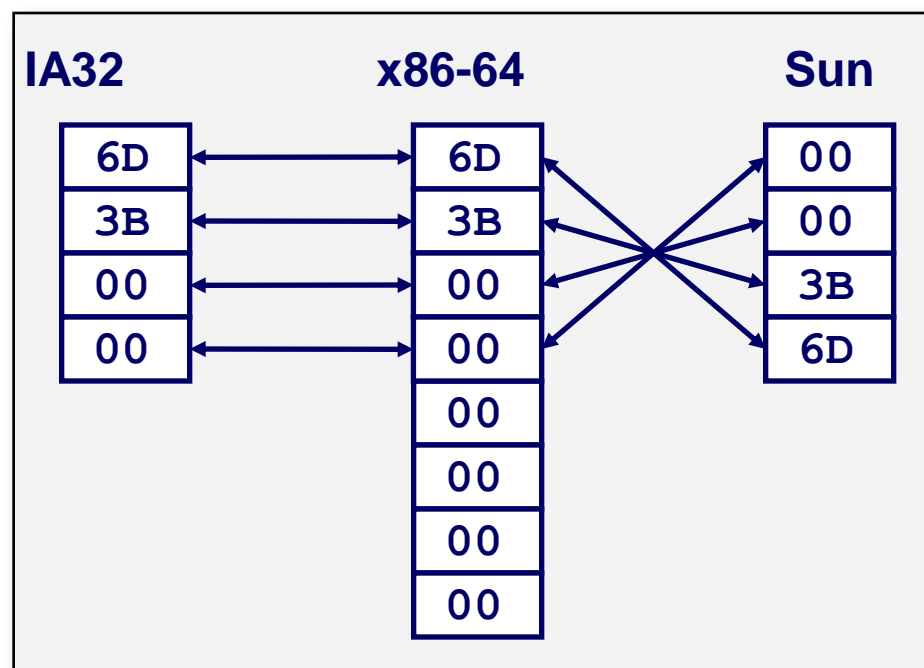
Representing Integers

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

int A = 15213;

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
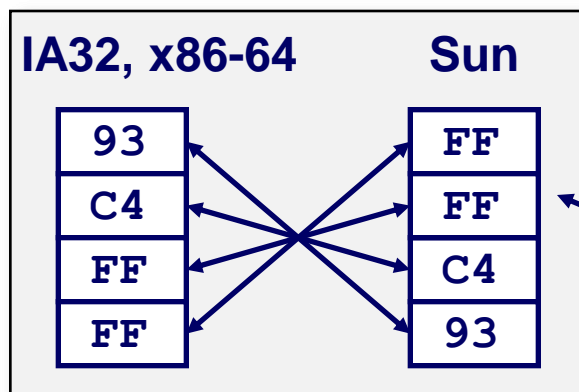
Two's complement representation (Covered later)

Representing Pointers

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

Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

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**

Linux/Alpha		Sun
31	↔	31
38	↔	38
32	↔	32
34	↔	34
33	↔	33
00	↔	00
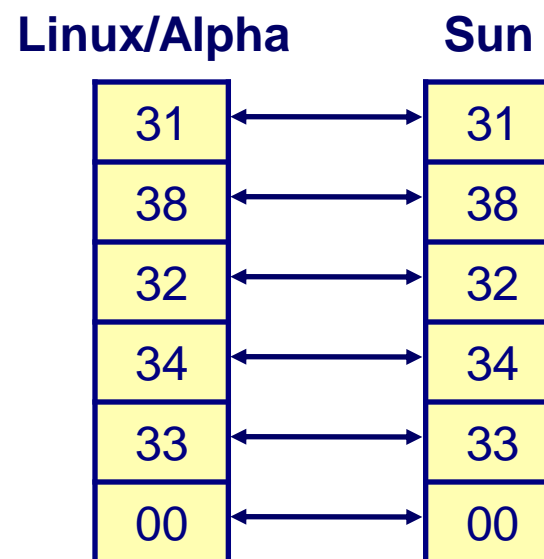
Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- 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

Or

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

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

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

^	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
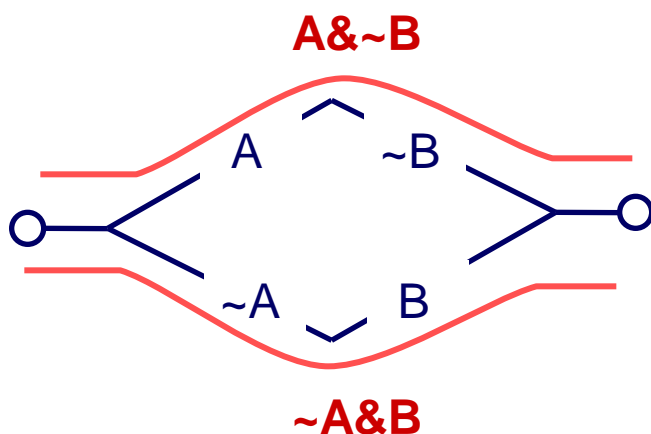
Connection when

A&~B | ~A&B

= A^B

Boolean Algebra ≈ Integer Ring

- *Commutativity*

  A | B = B | A   A + B = B + A

  A & B = B & A   A * B = B * A

- *Associativity*

  (A | B) | C = A | (B | C)   (A + B) + C = A + (B + C)

  (A & B) & C = A & (B & C)   (A * B) * C = A * (B * C)

- *Product distributes over sum*

  A & (B | C) = (A & B) | (A & C)   A * (B + C) = A * B + B * C

- *Sum and product identities*

  A | 0 = A   A + 0 = A

  A & 1 = A   A * 1 = A

- *Zero is product annihilator*

  A & 0 = 0   A * 0 = 0

- *Cancellation of negation*

  ~ (~ A) = A   – (– A) = A

Boolean Algebra   ≠   Integer Ring

- Boolean: *Sum distributes over product*

  A | (B & C) = (A | B) & (A | C) $A + (B * C) \neq (A + B) * (A + C)$
- Boolean: *Idempotency*

  A | A = A $A + A \neq A$
  - "A is true" or "A is true" = "A is true"

  A & A = A $A * A \neq A$
- Boolean: *Absorption*

  A | (A & B) = A $A + (A * B) \neq A$
  - "A is true" or "A is true and B is true" = "A is true"

  A & (A | B) = A $A * (A + B) \neq A$
- Boolean: *Laws of Complements*

  A | ~A = 1 $A + {-}A \neq 1$
  - "A is true" or "A is false"
- Ring: *Every element has additive inverse*

  $A \mid {\sim}A \neq 0$ $A + {-}A = 0$

Relations Between Operations

DeMorgan's Laws

- **Express & in terms of |, and vice-versa**
  - A & B = ~(~A | ~B)
    - » A and B are true if and only if neither A nor B is false
  - A | B = ~(~A & ~B)
    - » A or B are true if and only if A and B are not both false

Exclusive-Or using Inclusive Or

- A ^ B = (~A & B) | (A & ~B)
  - » Exactly one of A and B is true
- A ^ B = (A | B) & ~(A & B)
  - » Either A is true, or B is true, 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**

Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w−1}
  - aj = 1 if j ∈ 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
    - ▪ ~01000001_2 ➙ 10111110_2
  - ■ ~0x00 ➙ 0xFF
    - ▪ ~00000000_2 ➙ 11111111_2
  - ■ 0x69 & 0x55 ➙ 0x41
    - ▪ 01101001_2 & 01010101_2 ➙ 01000001_2
  - ■ 0x69 | 0x55 ➙ 0x7D
    - ▪ 01101001_2 | 01010101_2 ➙ 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)

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 right
- ■ **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

Cool Stuff with Xor

- **Bitwise Xor is form of addition**
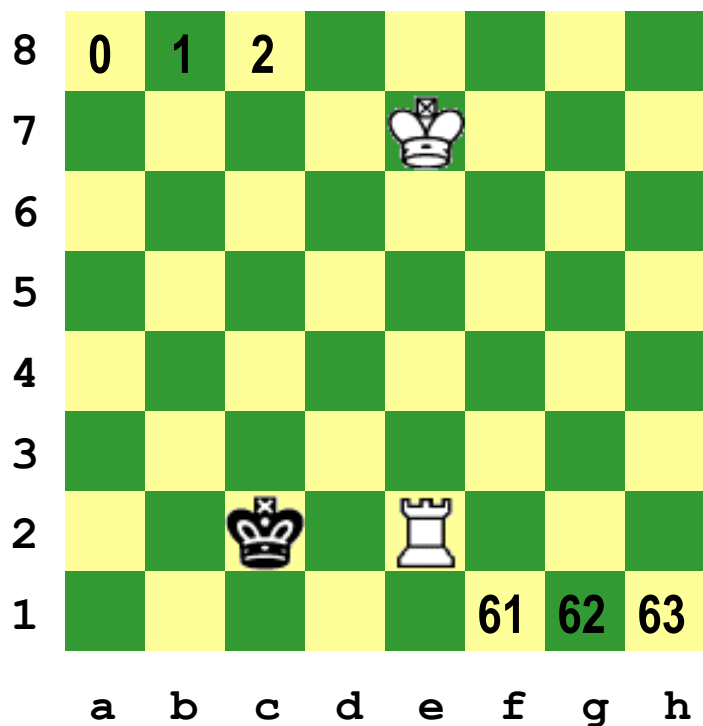- **With extra property that every value is its own additive inverse**
  - **A ^ A = 0**

```
void funny(int *x, int *y)
{
    *x = *x ^ *y;      /* #1 */
    *y = *x ^ *y;      /* #2 */
    *x = *x ^ *y;      /* #3 */
}
```

	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A
3	(A^B)^A = B	A
End	B	A

More Fun with Bitvectors

Bit-board representation of chess position:

```
unsigned long long blk_king, wht_king, wht_rook_mv2,…;
```



```
wht_king     = 0x0000000000001000ull;
blk_king     = 0x0004000000000000ull;
wht_rook_mv2 = 0x10ef101010101010ull;
...
/*
 * Is black king under attach from
 * white rook ?
 */
if (blk_king & wht_rook_mv2)
        printf("Yes\n");
```

More Bitvector Magic

- **Count the number of 1's in a word**
  - MIT Hackmem 169:

```
int bitcount(unsigned int n)
{
    unsigned int tmp;

    tmp = n - ((n >> 1) & 033333333333)
            - ((n >> 2) & 011111111111);
    return ((tmp + (tmp >> 3)) & 030707070707)%63;
}
```

More Bitvector Uses

Representation of small sets

Representation of polynomials:

- **Important for error correcting codes**
- **Arithmetic over finite fields, say GF(2^n)**
- **Example 0x15213 : $x^{16} + x^{14} + x^{12} + x^9 + x^4 + x + 1$**

Representation of graphs

- **A '1' represents the presence of an edge**

Representation of bitmap images, icons, cursors, ...

- **Exclusive-or cursor patent**

Representation of Boolean expressions and logic circuits

Today: Bits, Bytes, and Integers (1-2)

- **Representing information as bits**
- **Bit-level manipulations**
- **Summary**

Summary

It's All About Bits & Bytes

- Numbers
- Programs
- Text

Different Machines Follow Different Conventions for

- Word size
- Byte ordering
- Representations

Boolean Algebra is the Mathematical Basis

- Basic form encodes "false" as 0, "true" as 1
- General form like bit-level operations in C
  - Good for representing & manipulating sets

Bits, Bytes, and Integers (2-2)

15-213/18-243: Introduction to Computer Systems
3rd Lecture, 24 May 2012

Instructors:

Gregory Kesden

Last Time: Bits and Bytes

- **Bits, Bytes, Words**
- **Decimal, binary, hexadecimal representation**
- **Virtual memory space, addressing, byte ordering**
- **Boolean algebra**
- **Bit versus logical operations in C**

Today: Integers

- **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
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

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

Today: Bits, Bytes, and Integers

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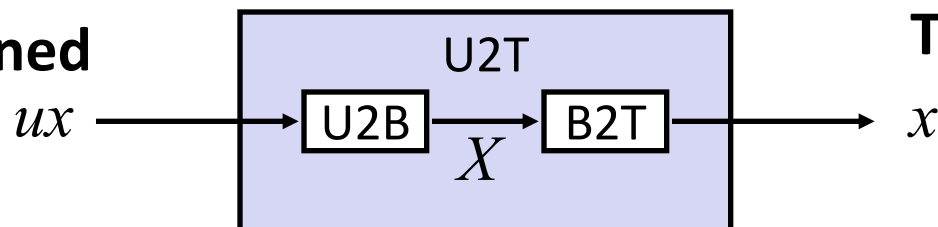- **Summary**

Mapping Between Signed & Unsigned
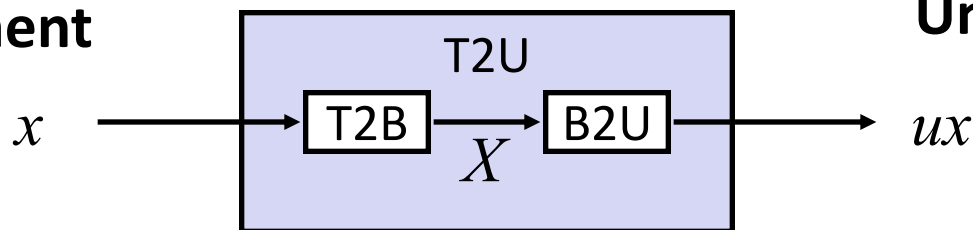
Two's Complement

Unsigned

Maintain Same Bit Pattern

Unsigned

Two's Complement

Maintain Same Bit Pattern

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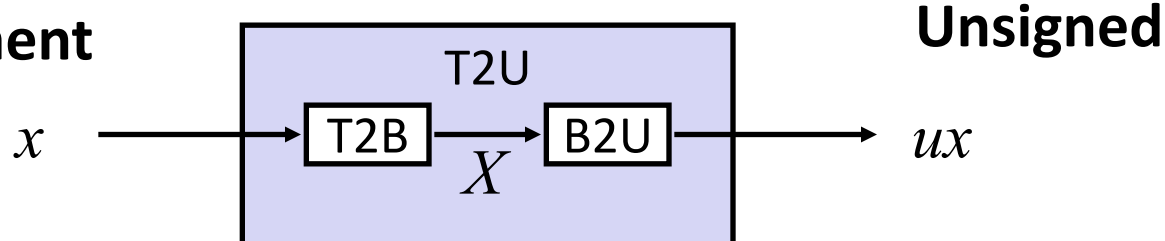**keep bit representations and reinterpret**

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
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

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

⟷ = ⟷

⟷ +/- 16 ⟷

Relation between Signed & Unsigned

Two's Complement   **Unsigned**

x → T2U [T2B → X → B2U] → ux

Maintain Same Bit Pattern

ux [+ | + | + | ··· | + | + | +]   ($w{-}1$ ... 0)

x [- | + | + | ··· | + | + | +]

↑

Large negative weight
becomes
Large positive weight
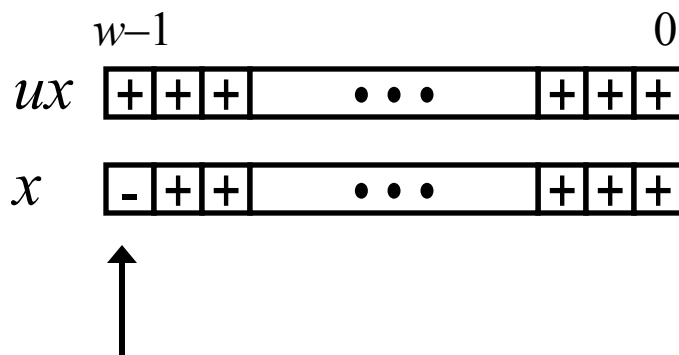
$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

Conversion Visualized

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

2's Complement Range

$TMax$

0

-1

-2

$TMin$

$UMax$
$UMax - 1$

$TMax + 1$
$TMax$

0

Unsigned Range
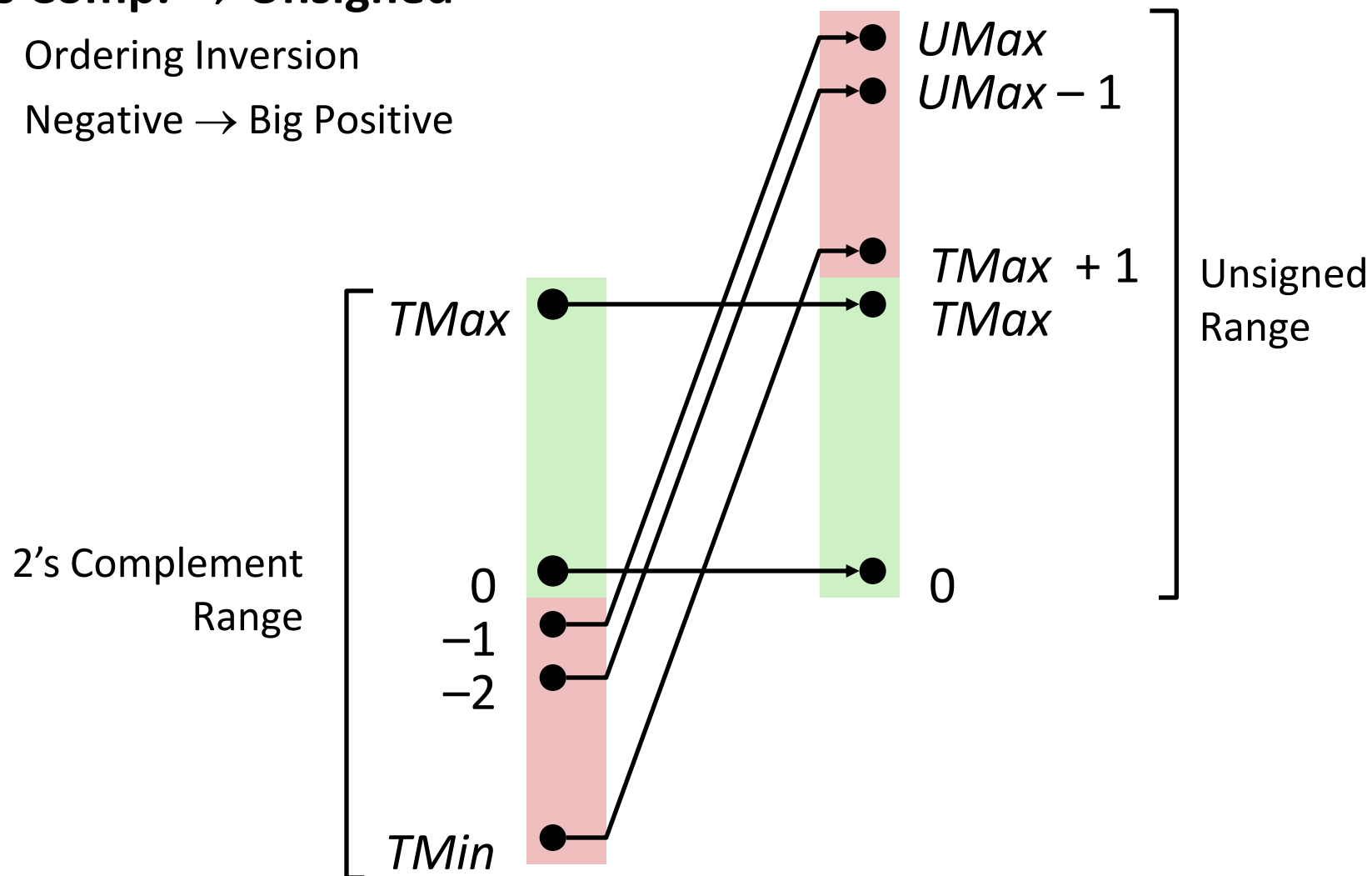
Signed vs. Unsigned in C

- **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**

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);
```

```c
/* 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;
}
```

```c
#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

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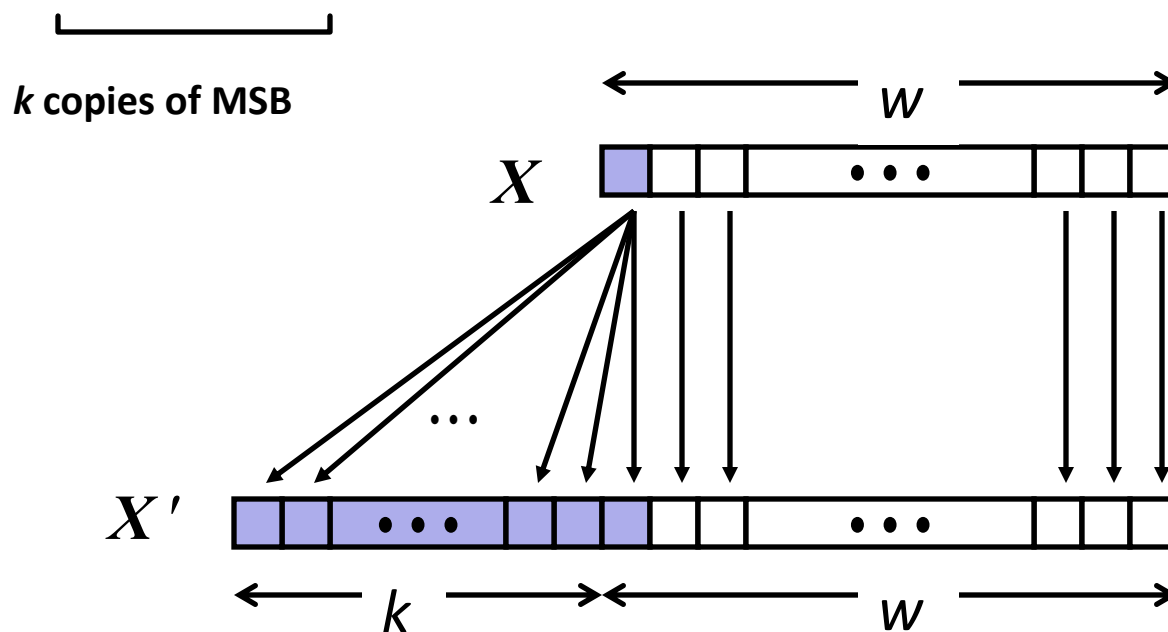 - Addition, negation, multiplication, shifting
- **Summary**

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

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

Today: Bits, Bytes, and Integers

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

Negation: Complement & Increment

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

  `~x + 1 == -x`

- **Complement**

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

$$
\begin{array}{rl}
x & \boxed{1\,0\,0\,1\,1\,1\,0\,1} \\
+ \quad \text{~x} & \boxed{0\,1\,1\,0\,0\,0\,1\,0} \\
\hline
\text{-1} & \boxed{1\,1\,1\,1\,1\,1\,1\,1}
\end{array}
$$

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

Unsigned Addition

Operands: w bits

$$u$$

$$+ \; v$$

True Sum: $w+1$ bits

$$u + v$$

Discard Carry: w bits
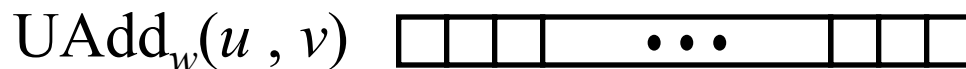
$$\text{UAdd}_w(u \, , \, v)$$

- **Standard Addition Function**

  - Ignores carry output

- **Implements Modular Arithmetic**

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

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

Visualizing (Mathematical) Integer Addition

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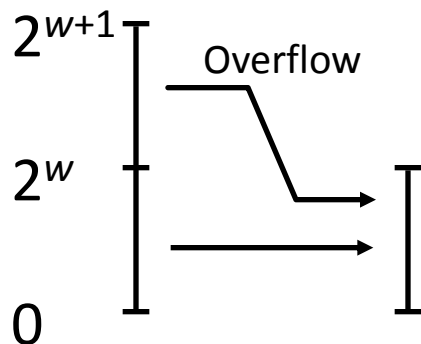 - Values increase linearly with u and v
  - Forms planar surface

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

Integer Addition
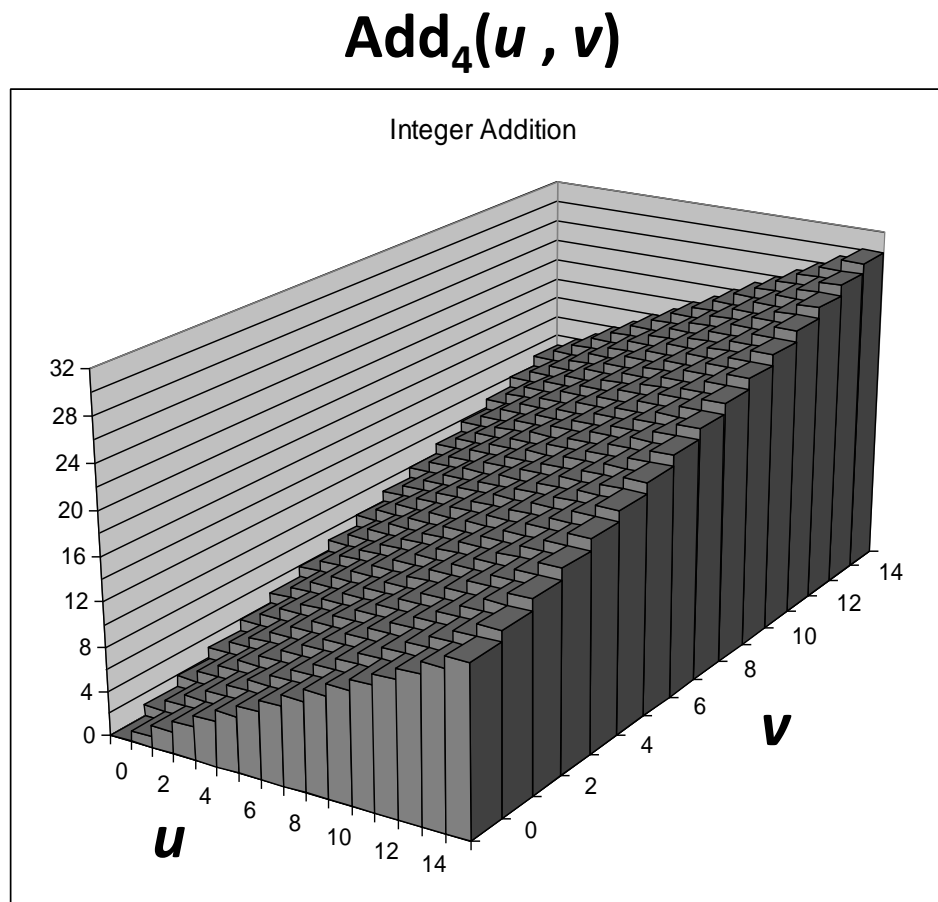
Visualizing Unsigned Addition

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

Overflow

True Sum

2^{w+1} — Overflow

2^w —

0

Modular Sum

UAdd$_4$(u , v)

Overflow

v

u
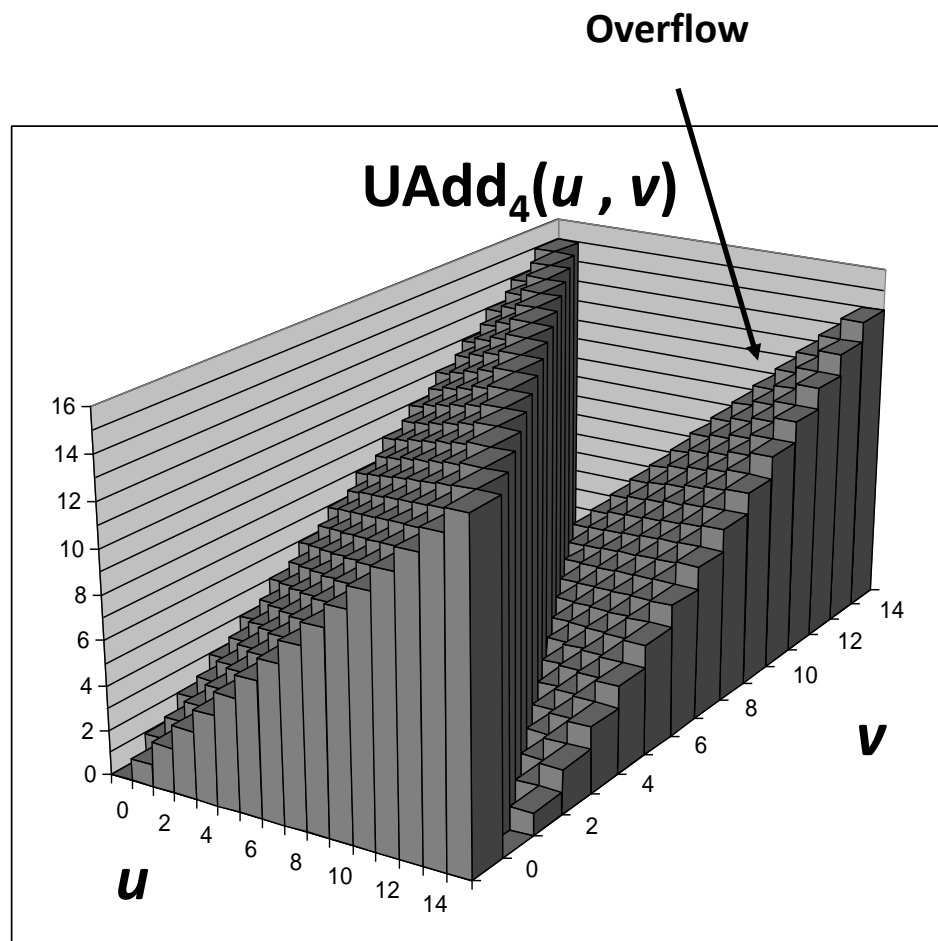
Mathematical Properties

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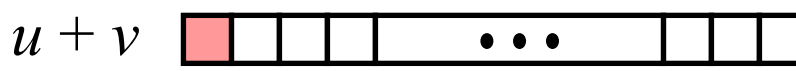 $$\mathrm{UAdd}_w(u\,, 0) = u$$
  - Every element has additive **inverse**
    - Let $\mathrm{UComp}_w(u\,) = 2^w - u$
      $$\mathrm{UAdd}_w(u\,, \mathrm{UComp}_w(u\,)) = 0$$

Two's Complement Addition

Operands: w bits

u

$+\ v$

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
    ```
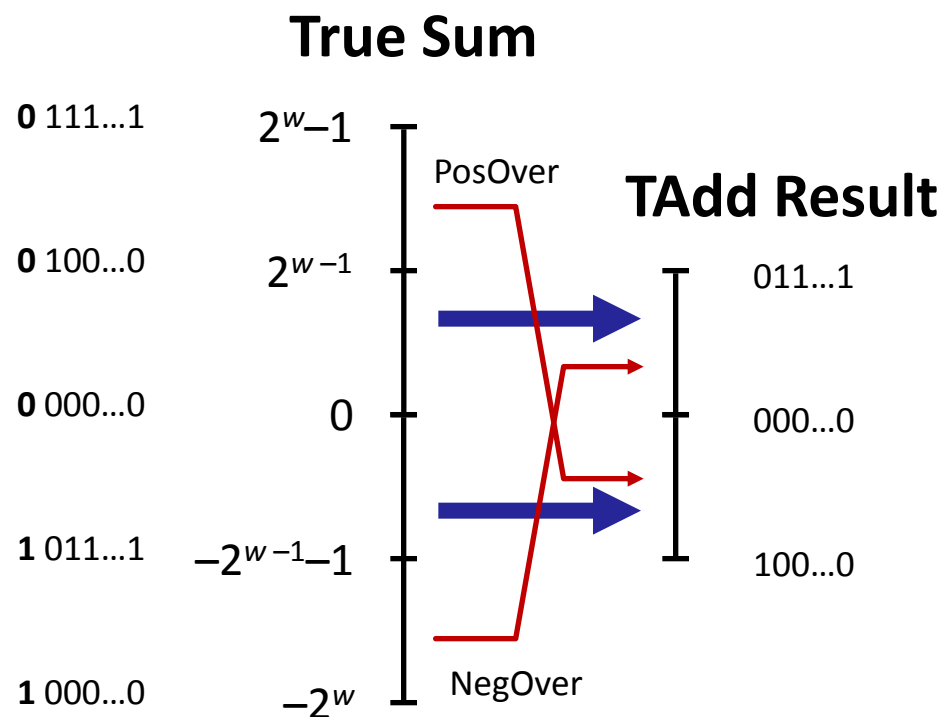 - Will give `s == t`

TAdd Overflow

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

True Sum

$\mathbf{0}\,111...1$ $2^w - 1$

PosOver

TAdd Result

$\mathbf{0}\,100...0$ 2^{w-1} $011...1$

$\mathbf{0}\,000...0$ 0 $000...0$

$\mathbf{1}\,011...1$ $-2^{w-1}-1$ $100...0$

$\mathbf{1}\,000...0$ -2^w

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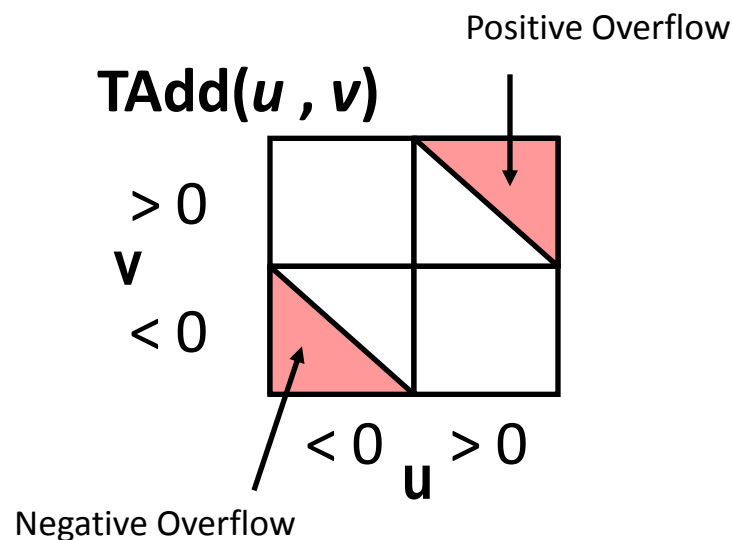 - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once

NegOver

PosOver

$$TAdd_4(u , v)$$

u

v
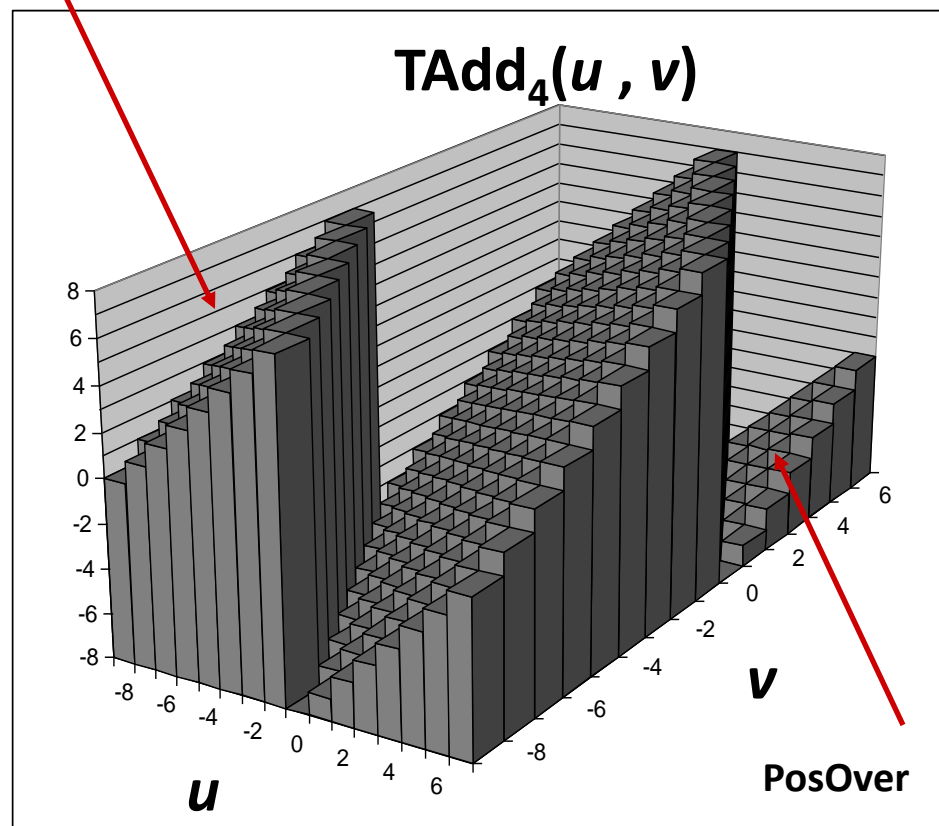
Characterizing TAdd

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

Positive Overflow

TAdd(u , v)

> 0

v

< 0

< 0   > 0
u

Negative Overflow

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

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

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

Multiplication

- **Computing Exact Product of w-bit numbers x, y**
  - Either signed or unsigned

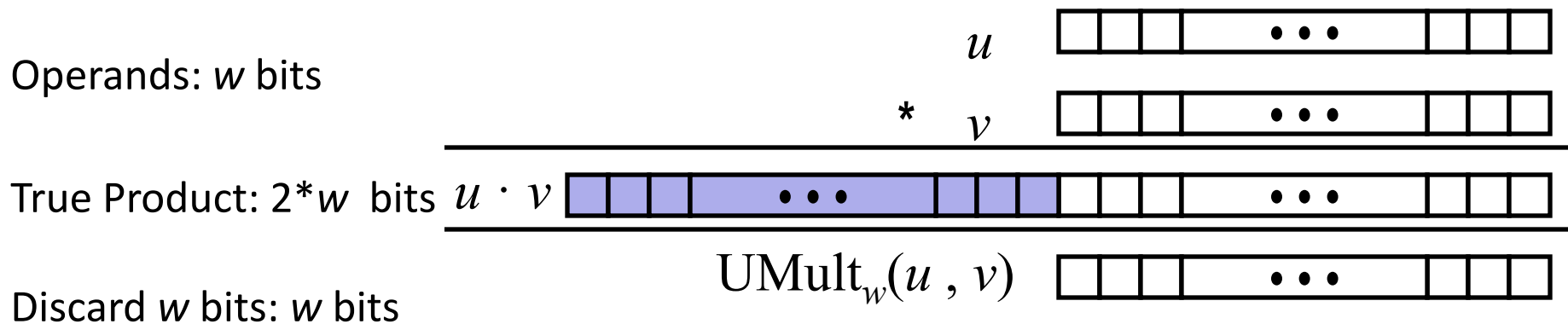- **Ranges**
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w{-}1$ bits
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only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

Operands: w bits

$$u$$

$$* \quad v$$

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

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

Discard w bits: w bits

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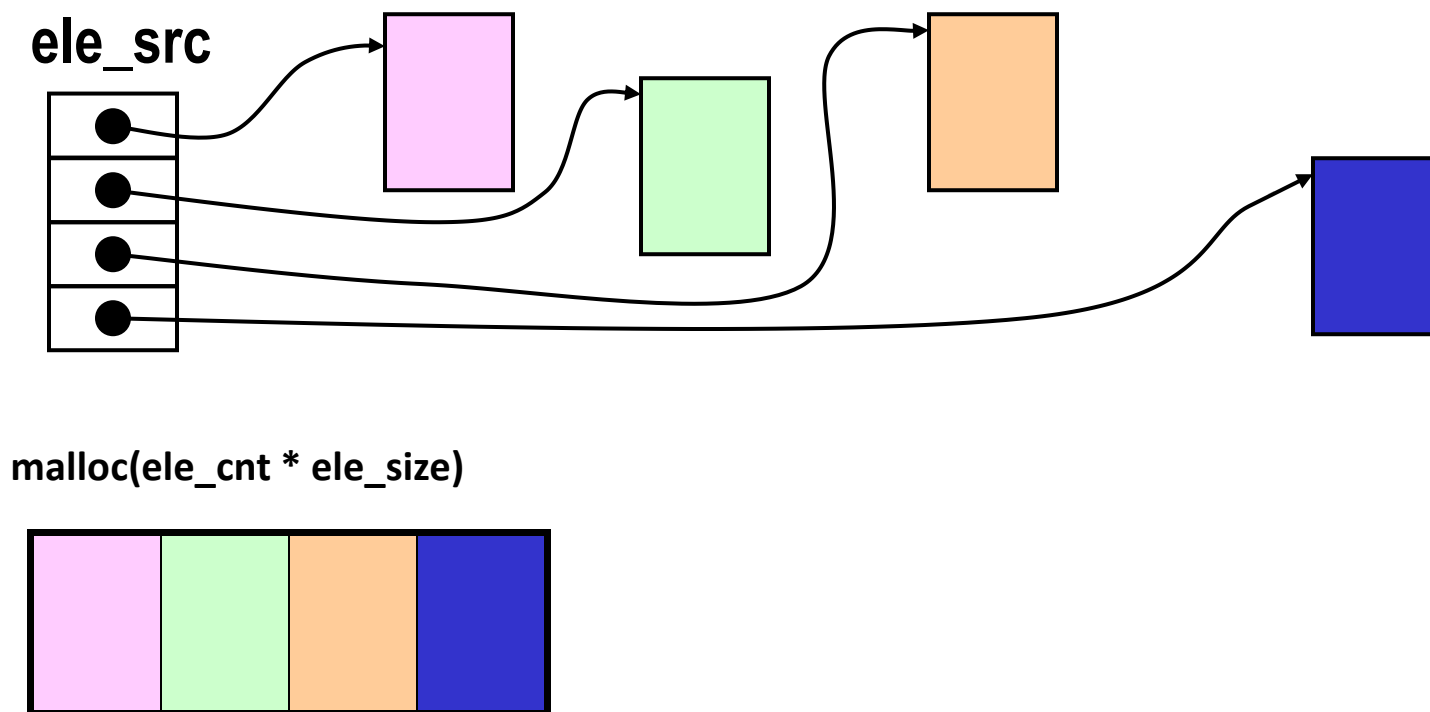
- **Implements Modular Arithmetic**

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

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);
```


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;
}
```
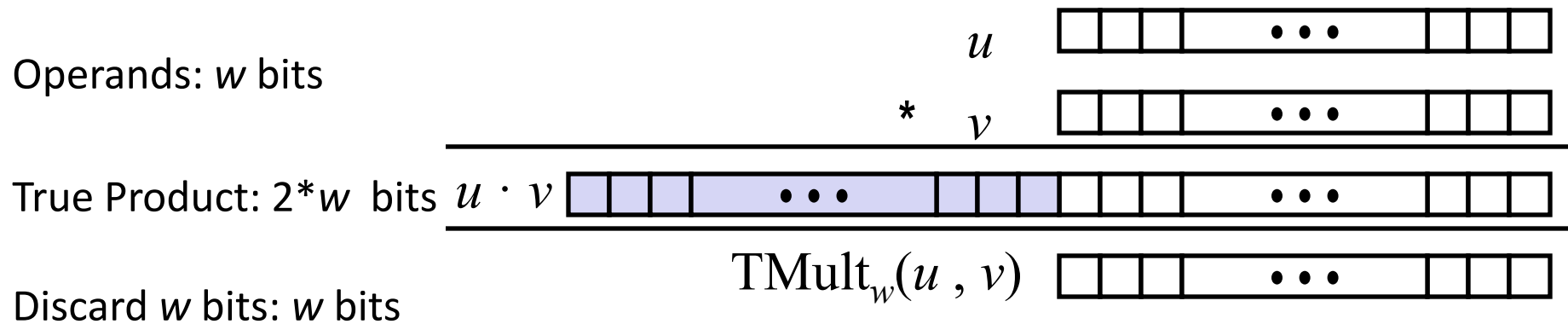
XDR Vulnerability

malloc(ele_cnt * ele_size)

- **What if:**
  - **`ele_cnt`** $= 2^{20} + 1$
  - **`ele_size`** $= 4096$ $\quad = 2^{12}$
  - 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

$$u$$

$$* \quad v$$

$$u \cdot v$$

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

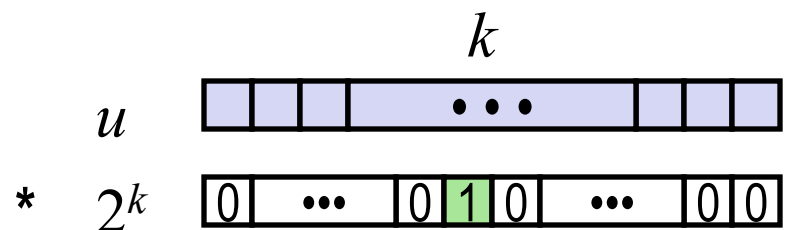- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same
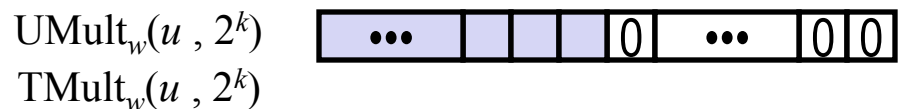
Power-of-2 Multiply with Shift

■ **Operation**

▪ `u << k` gives `u * `2^k

▪ Both signed and unsigned

k

u

Operands: w bits

$* \quad 2^k$

| 0 | ••• | 0 | 1 | 0 | ••• | 0 | 0 |

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

Discard k bits: w bits $\qquad$ $\text{UMult}_w(u, 2^k)$
$\text{TMult}_w(u, 2^k)$

■ **Examples**

▪ `u << 3` $\qquad$ `==` $\quad$ `u * 8`

▪ `u << 5 – u << 3` $\quad$ `==` $\qquad$ `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
```
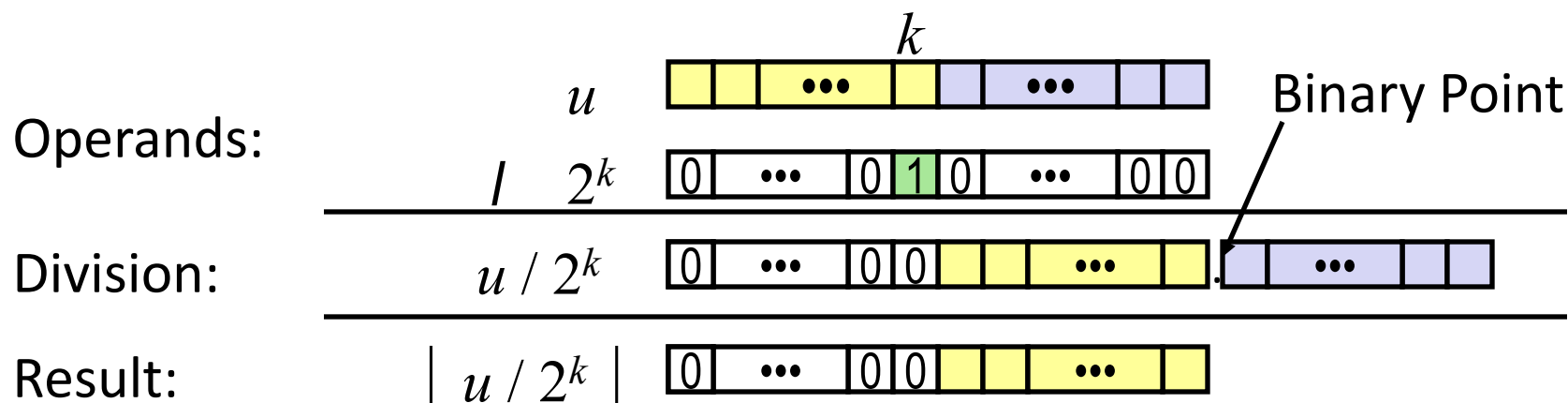
Explanation

```
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 >> 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`

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

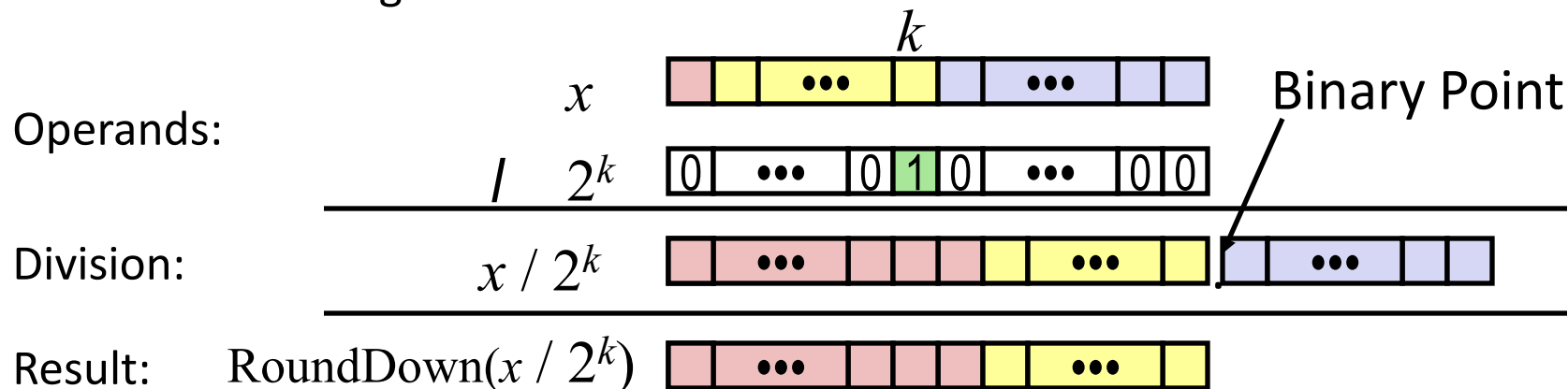```
shrl $3, %eax
```

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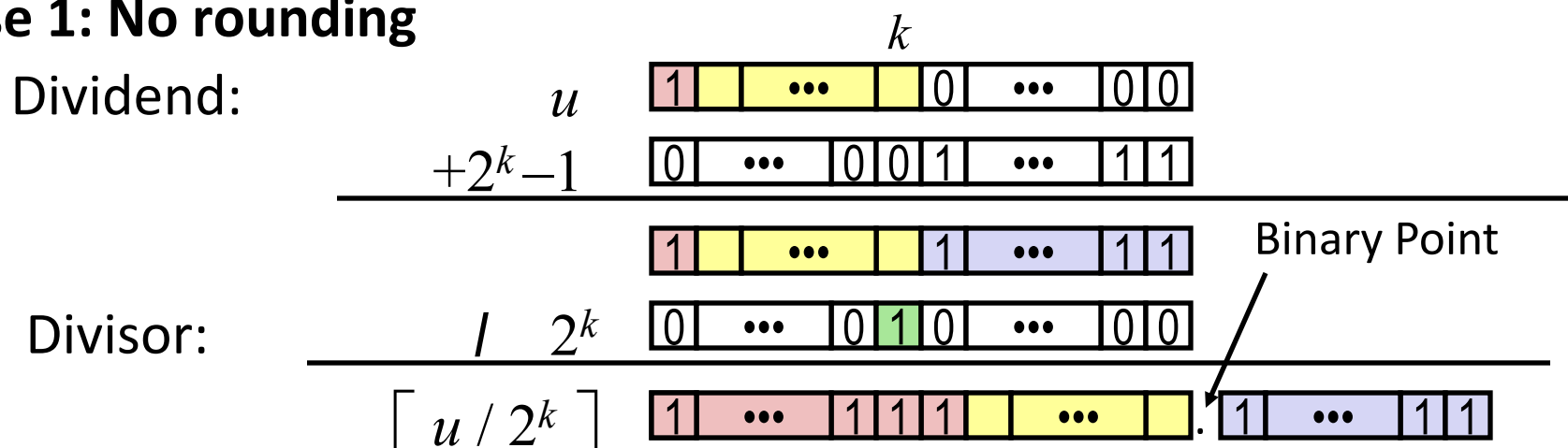 - Uses arithmetic shift
  - Rounds wrong direction when **u < 0**

Operands:

$$x \qquad / \quad 2^k$$

Division: $x / 2^k$

Binary Point

Result: $\text{RoundDown}(x / 2^k)$

	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	111100010 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: u

$+2^k-1$

Divisor: $/\ 2^k$

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

Binary Point

Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend: x

$+2^k-1$

Incremented by 1

Binary Point

Divisor: $/\quad 2^k$

$\lceil x/2^k \rceil$

Incremented by 1
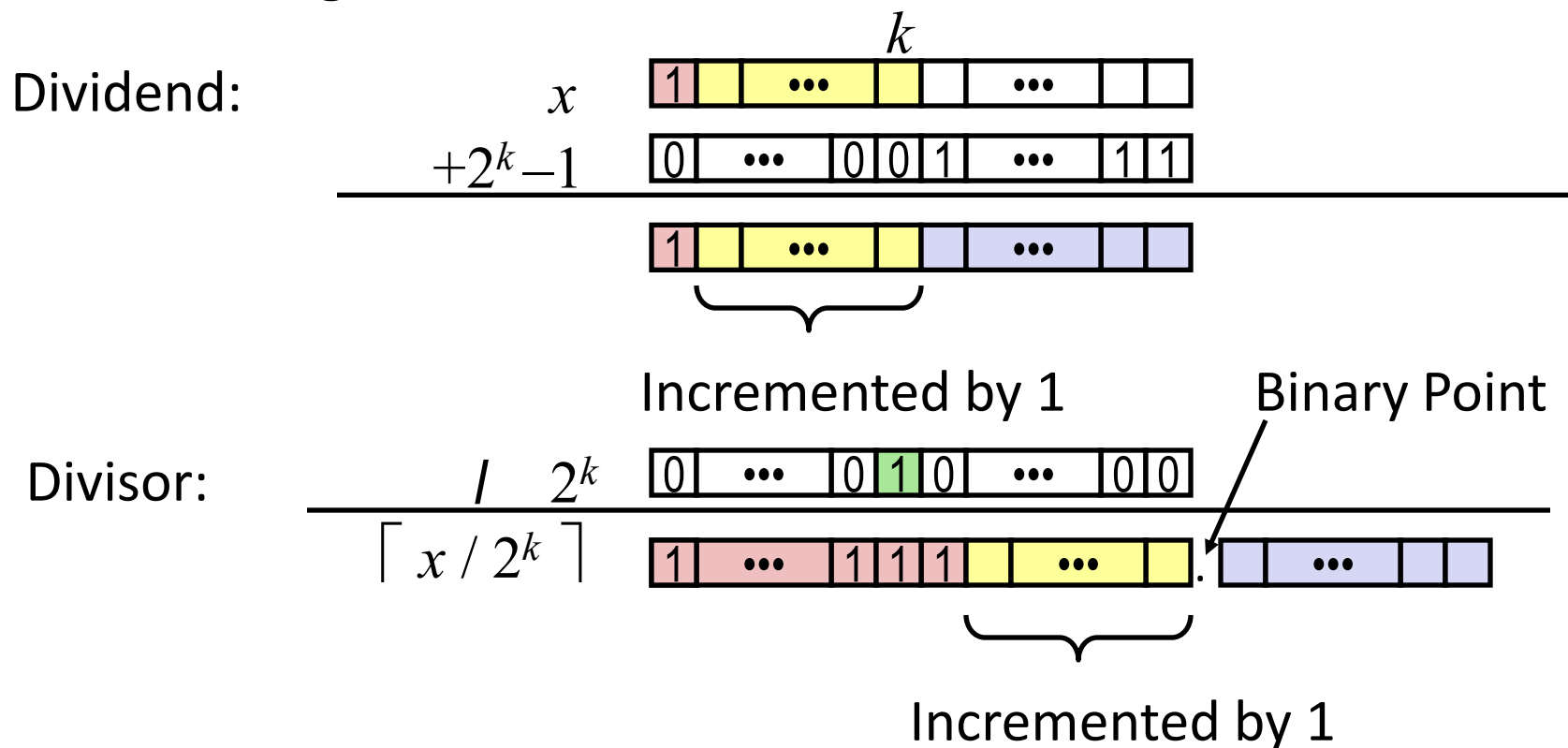
Biasing adds 1 to final result

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

Today: Integers

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

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.,

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

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

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

    $$TMax\; +\; 1\quad ==\quad TMin$$

    $$15213\; *\; 30426\quad ==\quad -10030 \qquad \text{(16-bit words)}$$

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

Integer C Puzzles

Argue that it is always true or provide a counter example.

Assume 32-bit architecture

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

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