Computer Systems I

Class 2

Why UNIX instead of Windows?

- UNIX is actually simpler
- It lets us get closer to the bits more easily
- But everything we are learning transfers to other environments.

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

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 $\neq$ Integer Ring

- Boolean: *Sum distributes over product*

  A | (B & C) = (A | B) & (A | C) A + (B * C) $\neq$ (A + B) * (B + 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 | ~A $\neq$ 0 A + –A = 0

Properties of & and ^

- **Boolean Ring**
  - ⟨{0,1}, ^, &, *I*, 0, 1⟩
  - Identical to integers mod 2
  - *I* is identity operation: *I*(A) = A

    A ^ A = 0

Property	Boolean Ring
Commutative sum	A ^ B = B ^ A
Commutative product	A & B = B & A
Associative sum	(A ^ B) ^ C = A ^ (B ^ C)
Associative product	(A & B) & C = A & (B & C)
Prod. over sum	A & (B ^ C) = (A & B) ^ (B & C)
0 is sum identity	A ^ 0 = A
1 is prod. identity	A & 1 = A
0 is product annihilator	A & 0 = 0
Additive inverse	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)
      - A 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}
- $a_j = 1$ 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`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41`
  - `~0x00`
  - `0x69 & 0x55`
  - `0x69 | 0x55`

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
    - Useful with two's complement integer representation

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

```
{
   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

Main Points

- **It's All About Bits & Bytes**
  - Numbers
  - Programs
  - Text
- **Different Machines Follow Different Conventions**
  - Word size
  - Byte ordering
  - Representations
- **Boolean Algebra is Mathematical Basis**
  - Basic form encodes "false" as 0, "true" as 1
  - General form like bit-level operations in C
    - Good for representing & manipulating sets

Lab Hints

- **When you want to work with some of the bits of the word:**
  - Use & and mask to turn some bits off (ex. set most significant bit to 0)
  - use | and mask to turn some bits on (ex. set bit in position 2 to 1)
  - Use rotation to get the bits you want *where* you want (often at the least significant position). (ex. what bit is in position 4?)
  - Exercise: Set the least significant byte of x to 0x12.

- **If you are not sure where to begin with a function**
  - Imagine the function works on single bits (Booleans) instead of words
  - Draw the table for the function and pick out the true answers
  - Exercise: majority function on 3 inputs.

Integers (Sections 2.2 and 2.3)

- 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

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

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

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

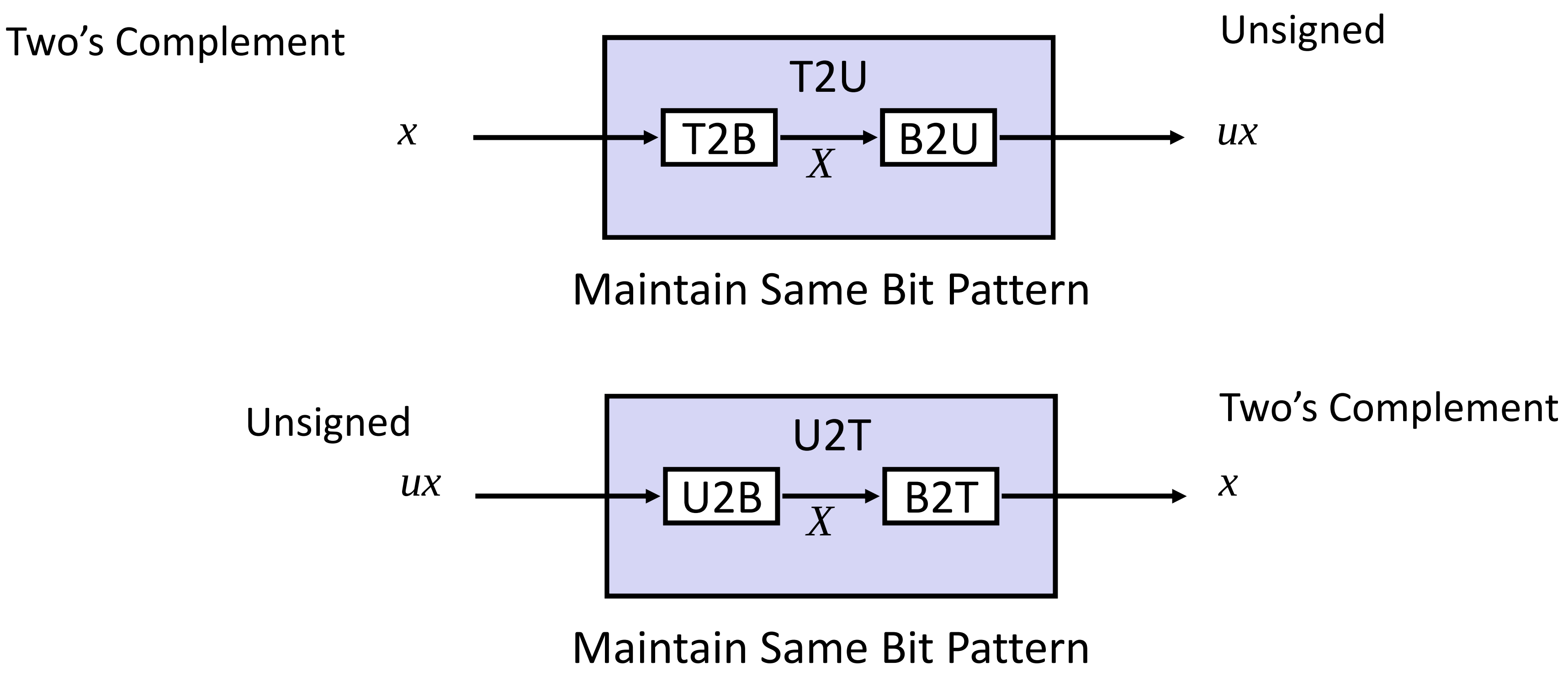
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **$\Rightarrow$ Can Invert Mappings**
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two's comp integer

Today: Integers

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

Mapping Between Signed & Unsigned



- **Mappings been 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 ←

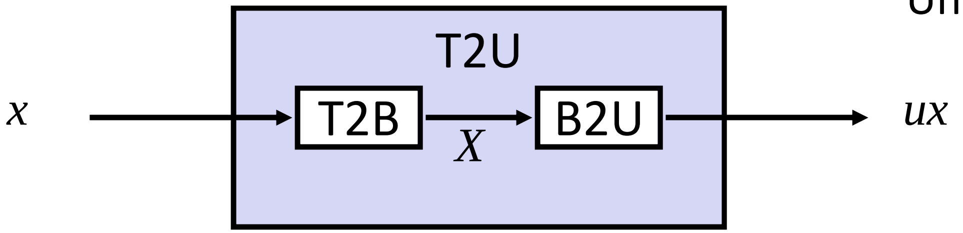
Mapping Signed ↔ Unsigned

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

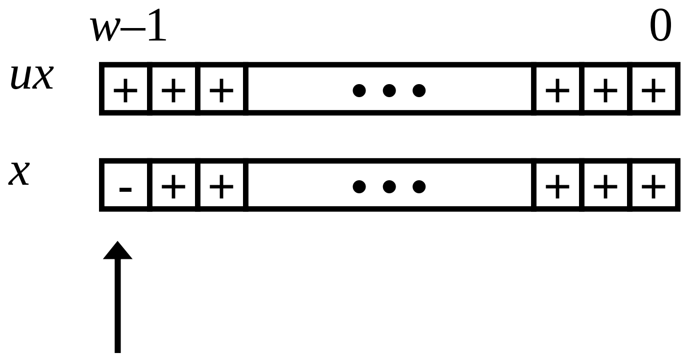
Relation between Signed & Unsigned

Two's Complement

Unsigned



Maintain Same Bit Pattern

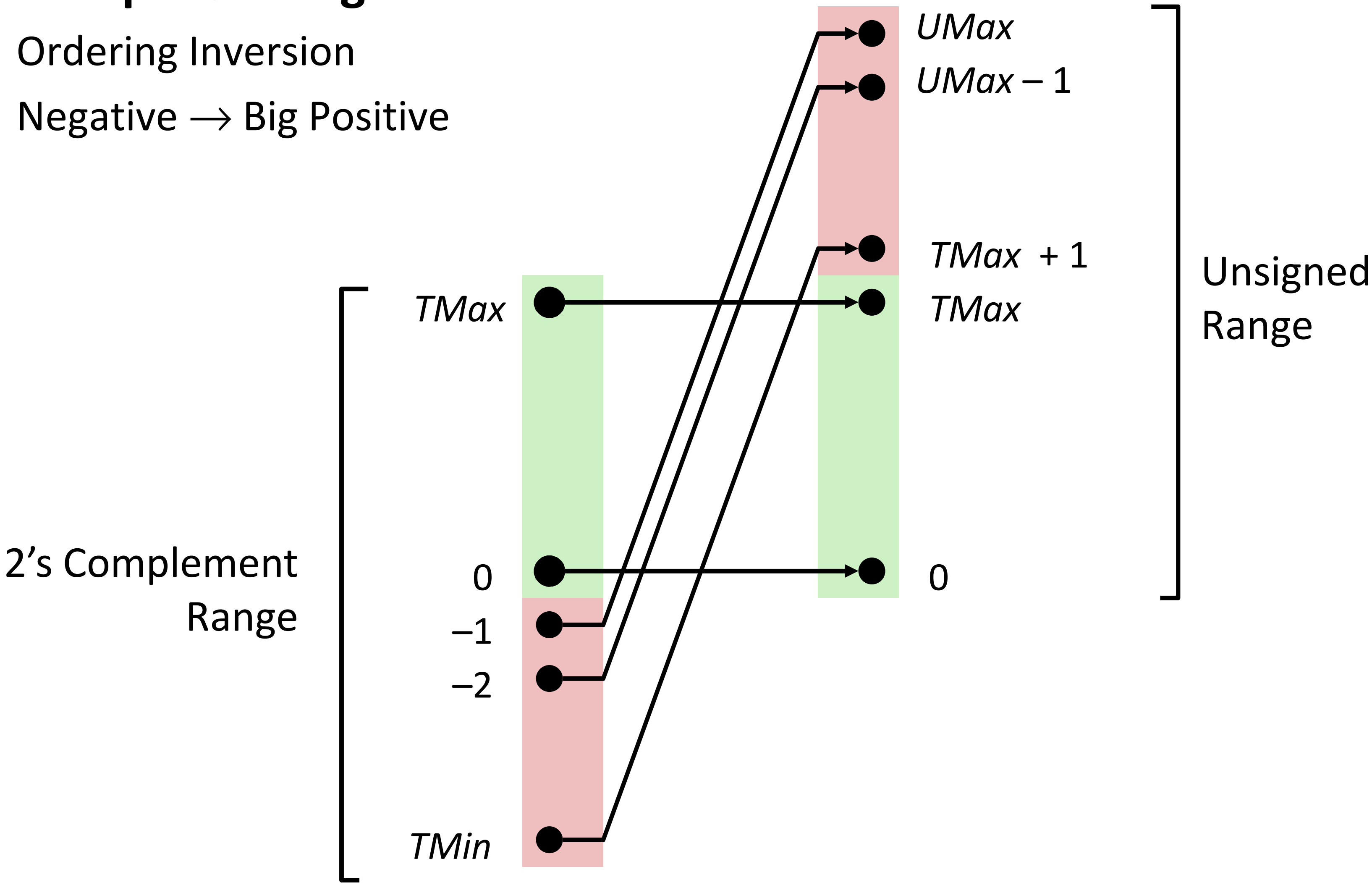


Large negative weight
becomes
Large positive weight

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

Conversion Visualized

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



Signed vs. Unsigned in C

Constants

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

```
0U, 4294967259U
```

Casting

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

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

- Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Casting Surprises

Expression Evaluation

- If mix unsigned and signed in single expression,
  signed values implicitly cast to unsigned
- Including comparison operations **<, >, ==, <=, >=**
- Examples for $W = 32$: **TMIN = -2,147,483,648 ,** **TMAX = 2,147,483,647**

Constant$_1$	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);
```

```
/* 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**

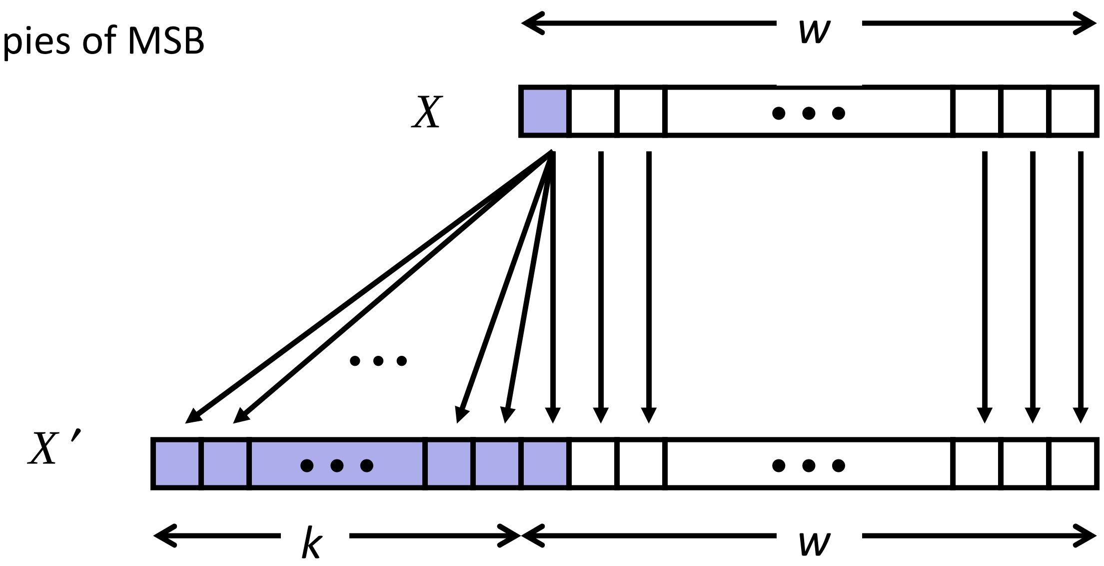
- **Expression containing signed and unsigned int**
  - int is cast to unsigned!!

Integers:

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

Sign Extension

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



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 behaviour

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`**

	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

x = 0

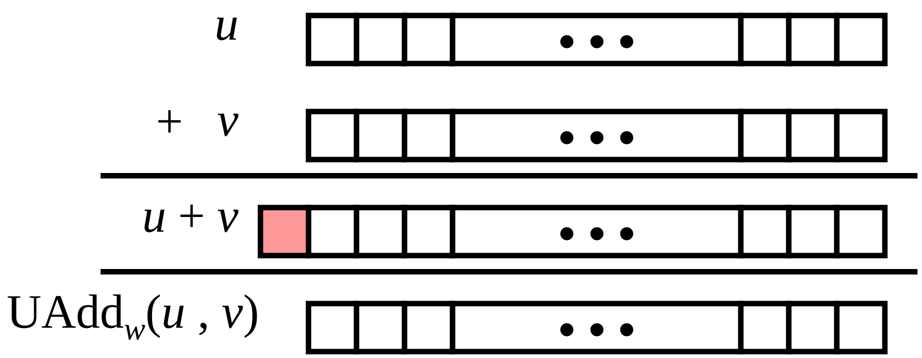
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



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

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

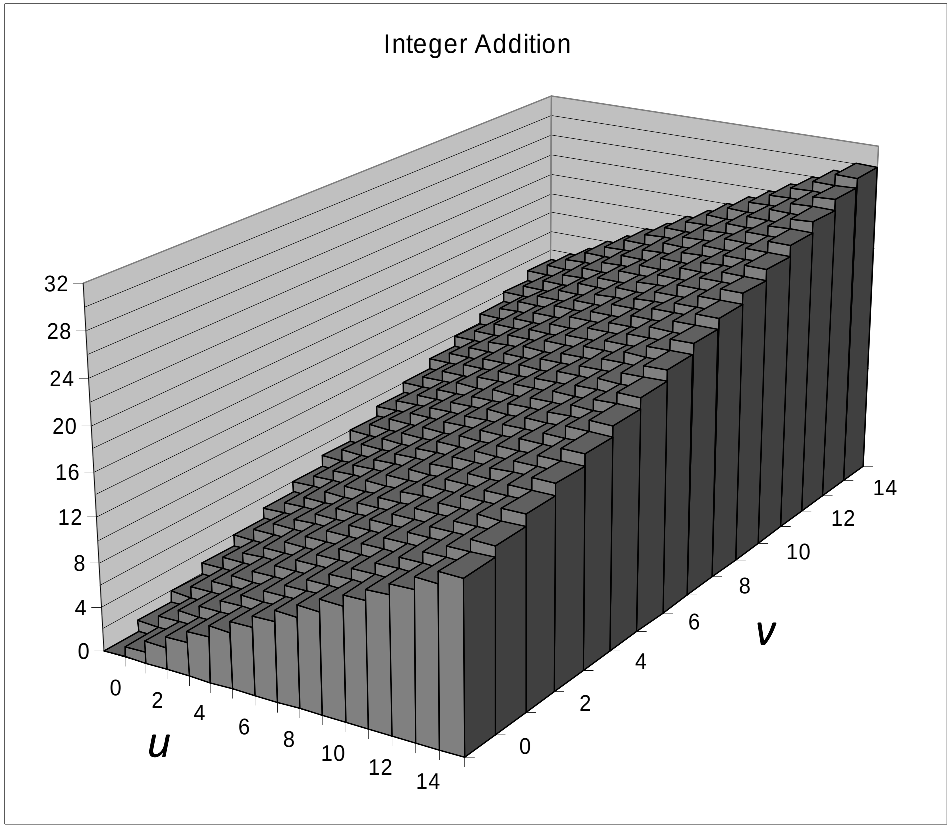
$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Visualizing (Mathematical) Integer Addition

Integer Addition

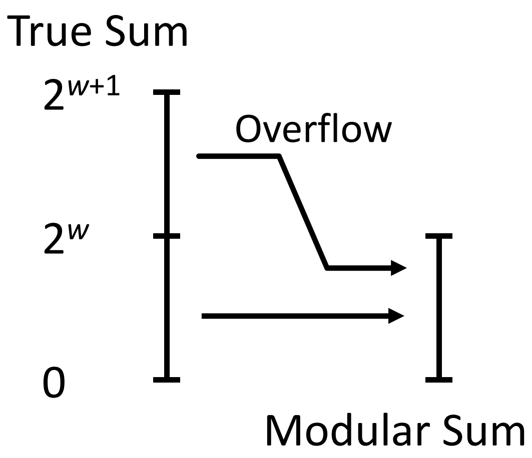
- 4-bit integers *u*, *v*
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with *u* and *v*
- Forms planar surface

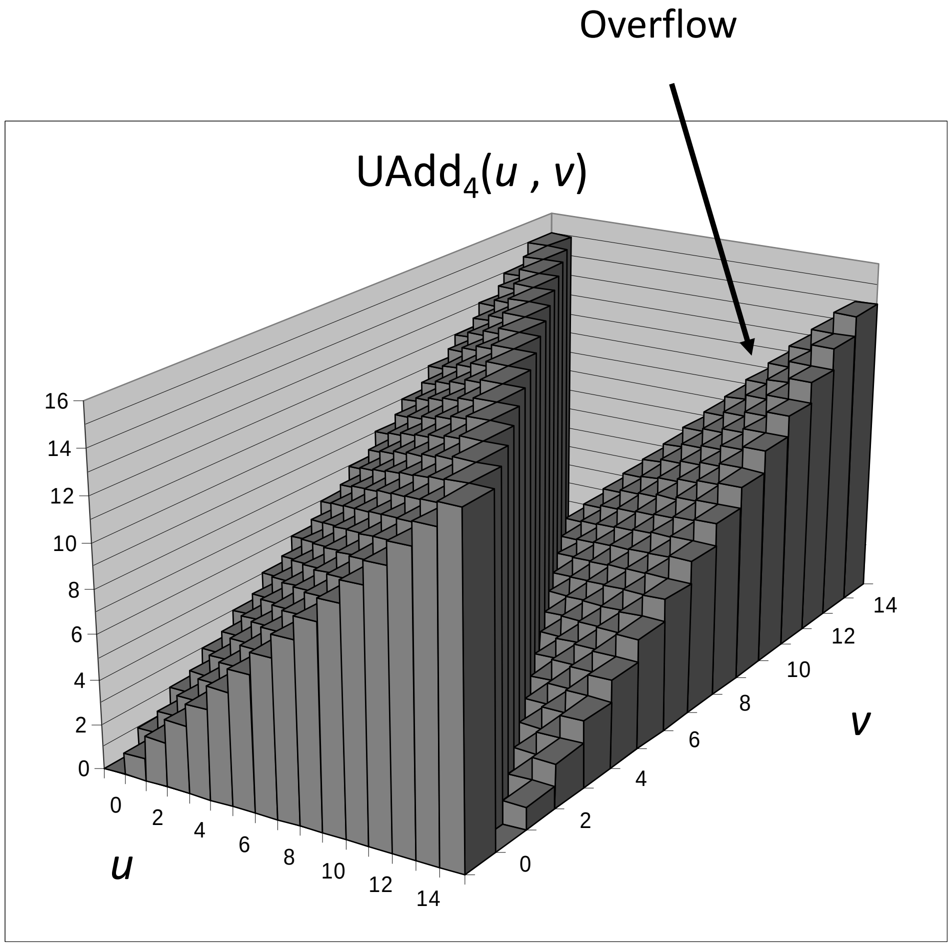
$\text{Add}_4(u, v)$



Visualizing Unsigned Addition

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





Mathematical Properties

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

    $0 \le UAdd_w(u, v) \le 2^w - 1$
  - **Commutative**

    $UAdd_w(u, v) = UAdd_w(v, u)$
  - **Associative**

    $UAdd_w(t, UAdd_w(u, v)) = UAdd_w(UAdd_w(t, u), v)$
  - **0** is additive identity

    $UAdd_w(u, 0) = u$
  - Every element has additive **inverse**
    - Let $UComp_w(u) = 2^w - u$

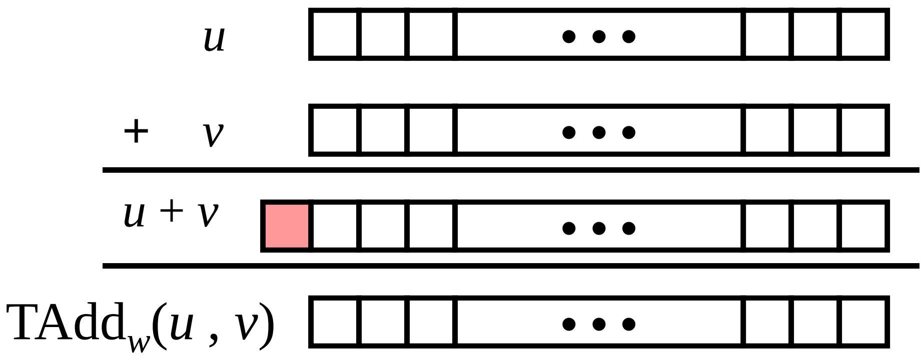
      $UAdd_w(u, UComp_w(u)) = 0$

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



TAdd and UAdd have Identical Bit-Level Behavior

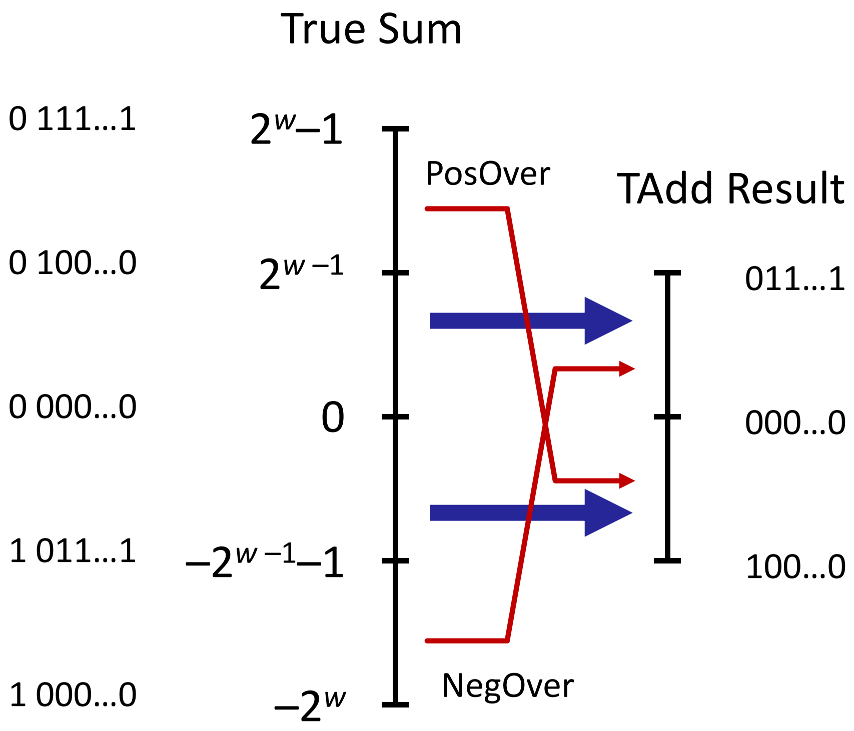
- Signed vs. unsigned addition in C:

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

- Will give `s == t`

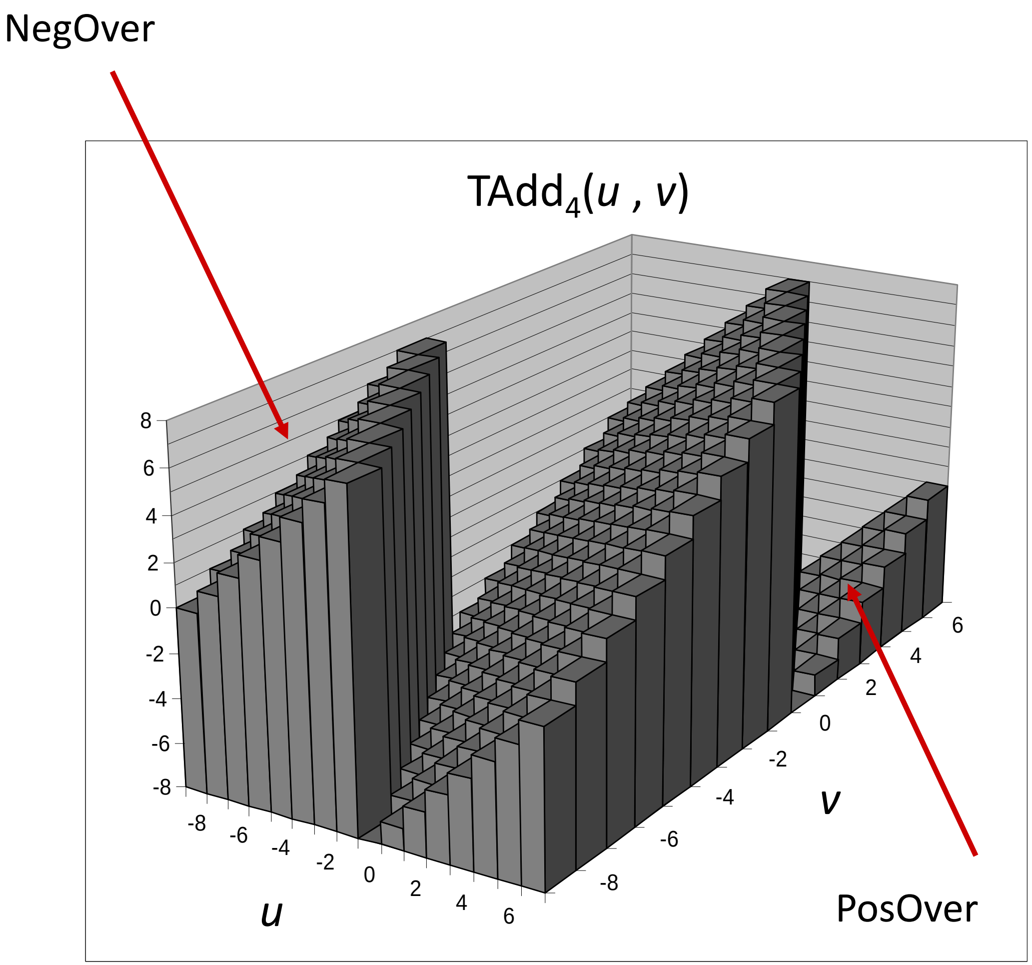
TAdd Overflow

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



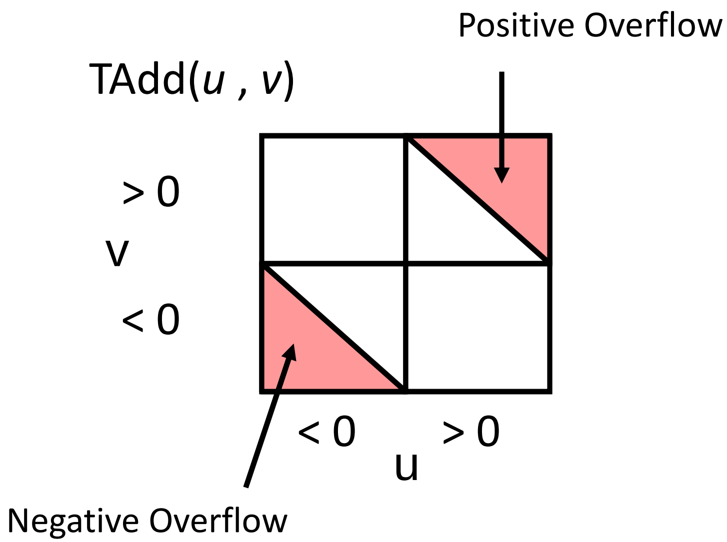
Visualizing 2's Complement Addition

- **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



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

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

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

Multiplication

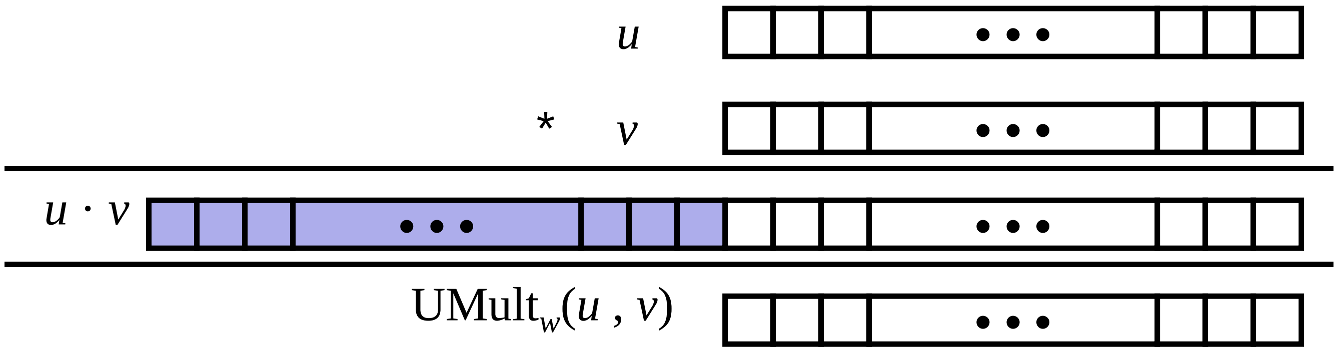
- **Computing Exact Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **Ranges**
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$ (because of sign bit)
- **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

True Product: $2*w$ bits

Discard w bits: w bits



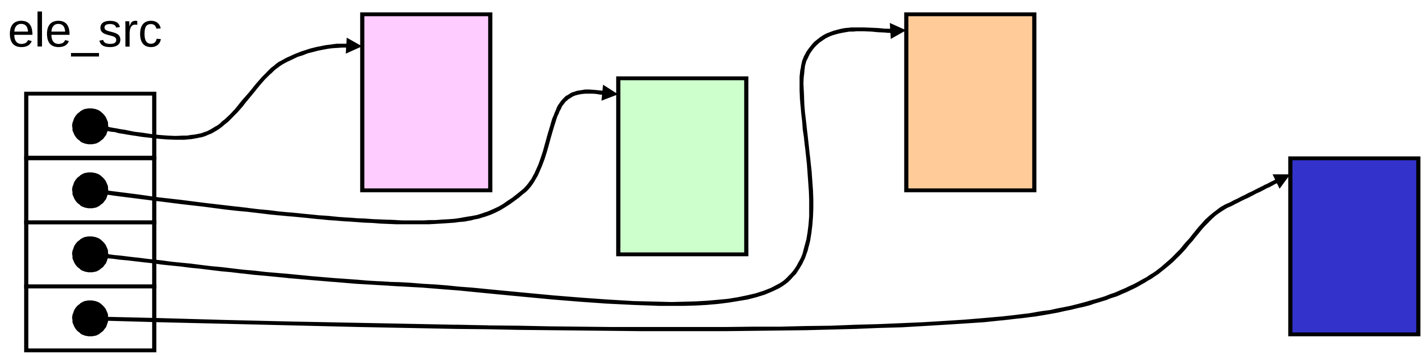
- **Standard Multiplication Function**
  - Ignores high order w bits
- **Implements Modular Arithmetic**

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

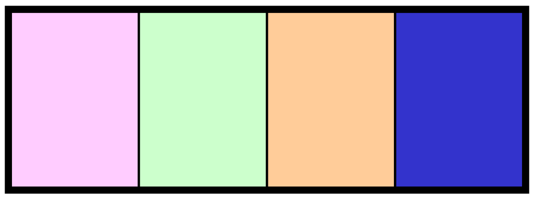
Code Security Example #2

- **SUN XDR library**
  - Widely used library for transferring data between machines

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



malloc(ele_cnt * ele_size)



XDR Code

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

XDR Vulnerability

malloc(ele_cnt * ele_size)

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

$$\boxed{2^{32} + 2^{12} = 4096}$$

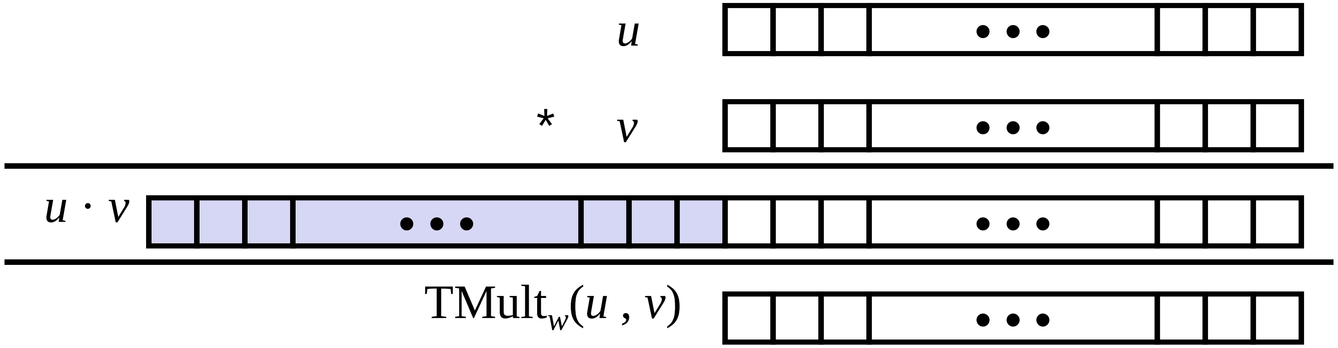
- **How can I make this function secure?**

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

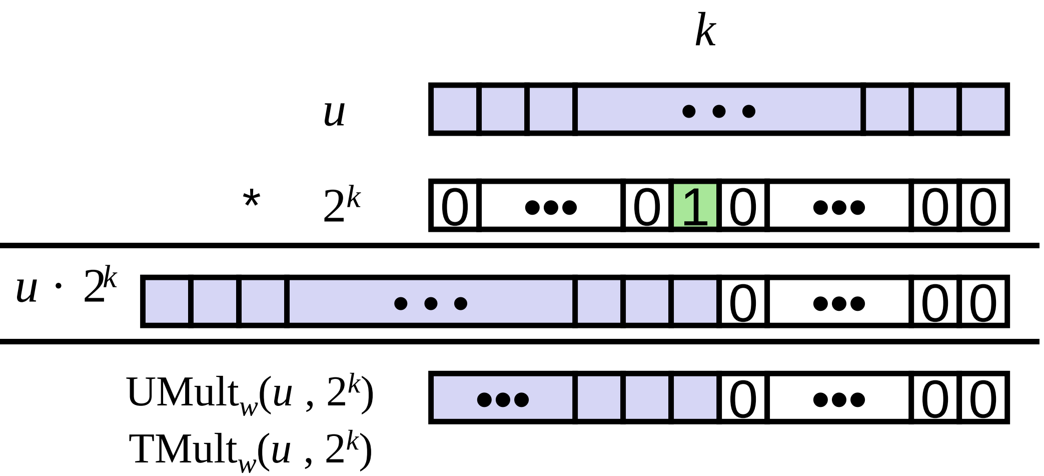
Operation

- **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

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

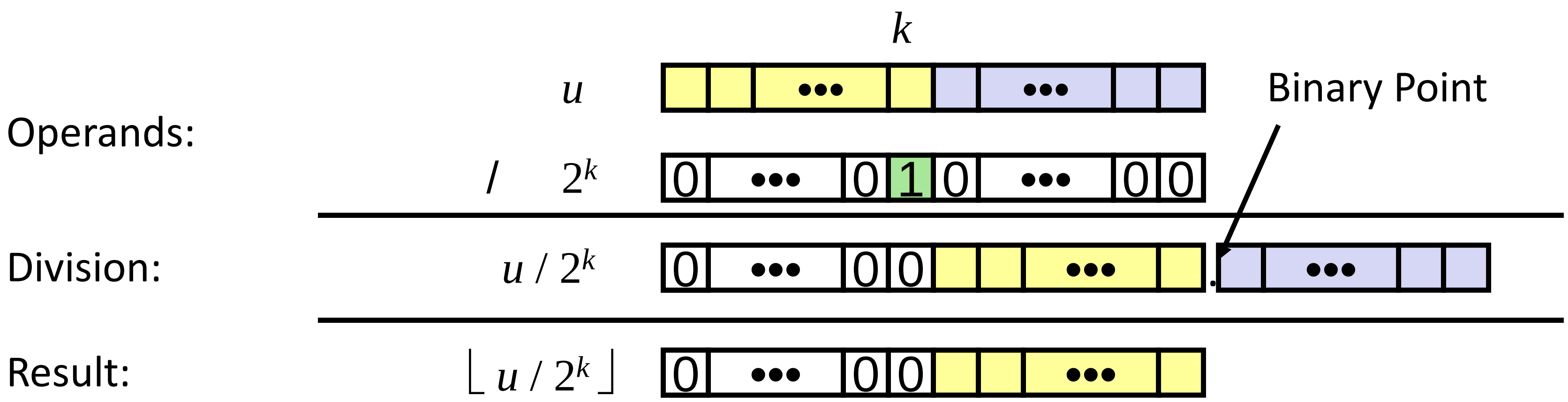
Explanation

```
  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

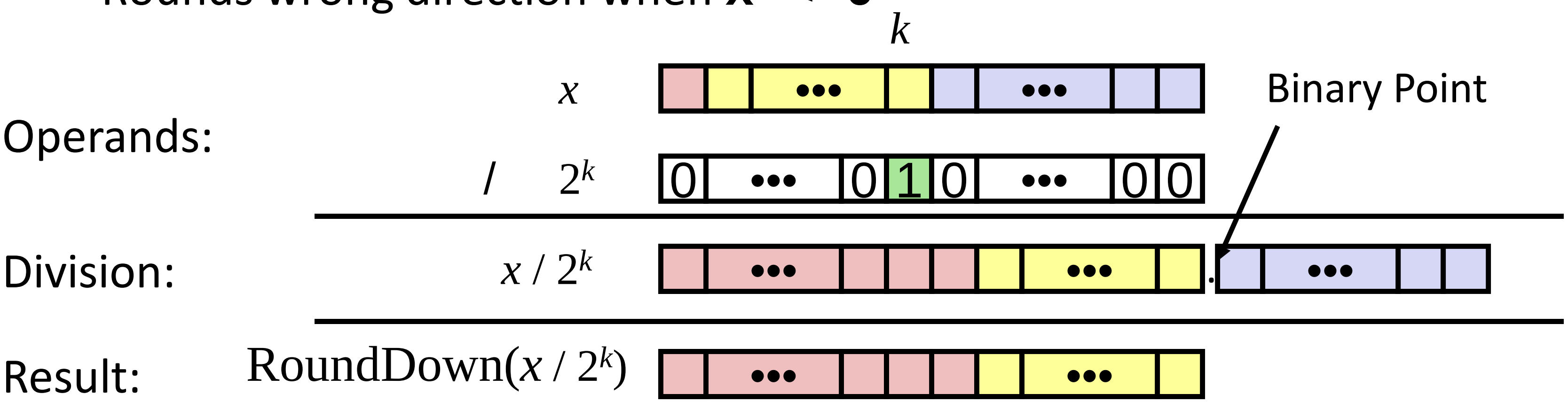
```
  # Logical shift
  return x >> 3;
```

- **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 **x < 0**

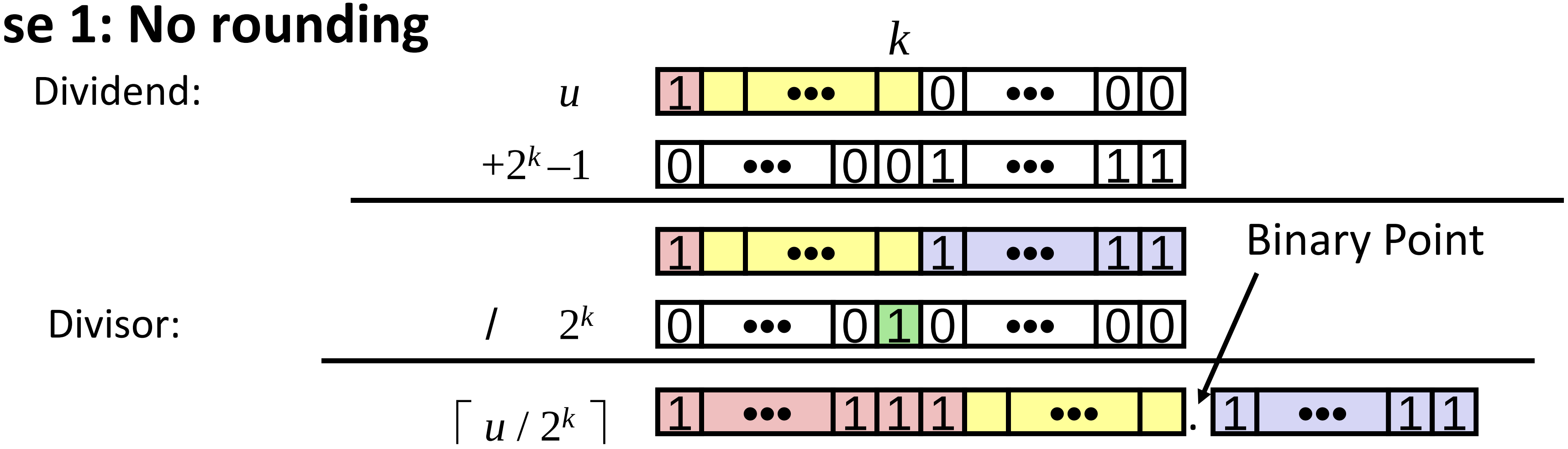


	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

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

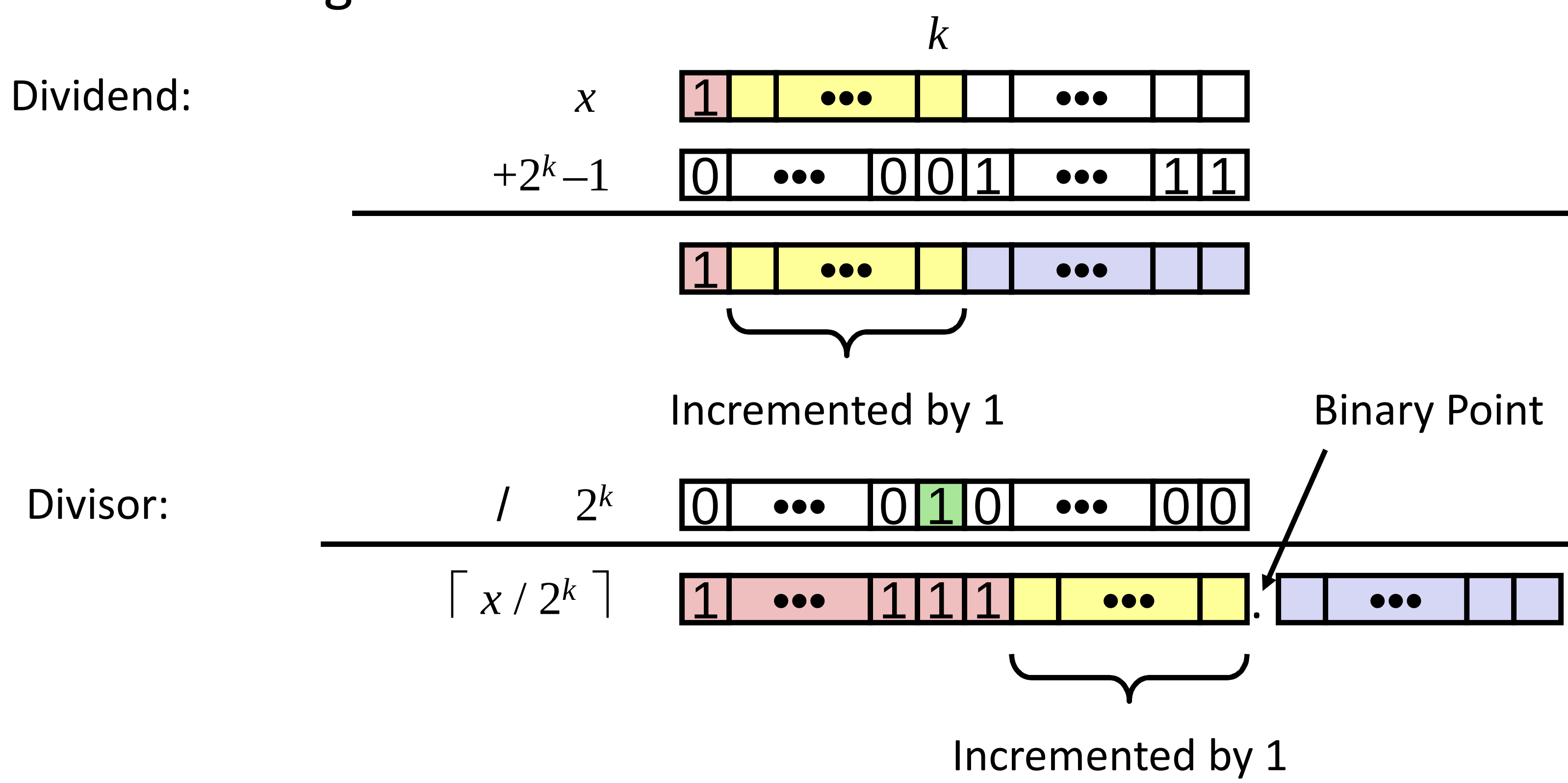
Case 1: No rounding



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

```
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

- **Casting between unsigned and signed ints does not change the bits, only the interpretation.**

- **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

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 and associative
  - Multiplication is commutative and associative
  - Multiplication distributes over addtion

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

Properties of Two's Comp. Arithmetic

Comparison to (Mathematical) Integer Arithmetic

- Addition and Multiplication are commutative and associative for both
- 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)

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 Interested In The Bit Pattern.**

Integer C Puzzles

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	

C Puzzle Answers

- Assume machine with 32 bit word size, two's comp. integers
- *TMin* makes a good counterexample in many cases

`x < 0`	⇒	`((x*2) < 0)`	False:	*TMin*
`ux >= 0`			True:	0 = *UMin*
`x & 7 == 7`	⇒	`(x<<30) < 0`	True:	$x_1 = 1$
`ux > -1`			False:	0
`x > y`	⇒	`-x < -y`	False:	`-1`, *TMin*
`x * x >= 0`			False:	`30426`
`x > 0 && y > 0`	⇒	`x + y > 0`	False:	*TMax*, *TMax*
`x >= 0`	⇒	`-x <= 0`	True:	–*TMax* < 0
`x <= 0`	⇒	`-x >= 0`	False:	*TMin*