Bits, Bytes, and Integers

Lecture 2

Yeongpil Cho

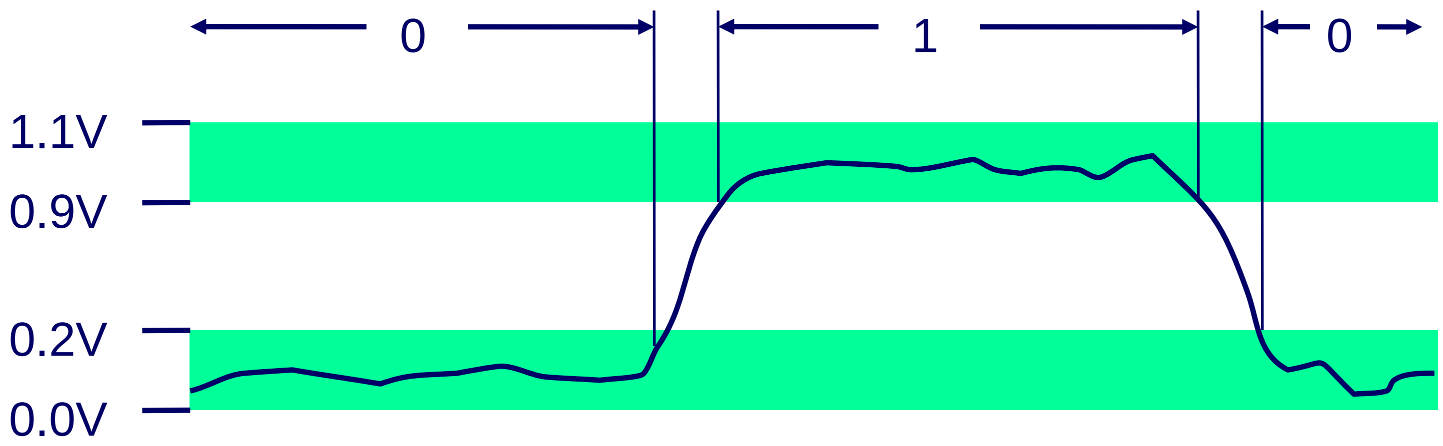
Hanyang University

Topics

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

Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



For example, can count in binary

- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

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

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

Topics

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

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

General Boolean Algebras

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

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

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

Example: Representing & Manipulating Sets

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

    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **Operations**
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }

Bit-Level Operations in C

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

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, ||, !
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
      - There are cases that we can know the result early.
- **Examples (char data type)**
  - !0x41 && 0x00
  - !0x00 || 0x01
  - !!0x41 && 0x01
  - 0x69 && 0x55 || 0x01
  - 0x69 || 0x55 && 0x01
  - p && *p (avoids null pointer access)
    - If p is null, *p is skipped due to the early termination.

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, | |, !
    - View 0 as “Fal
    - Anything nonze
    - Alway
    - Early
      - Th
- **Example**
  - !0x41 &&
  - !0x00 | |
  - !!0x41 &&
  - 0x69 && 0x55 | | 0x01
  - 0x69 | | 0x55 && 0x01
  - p && *p (avoids null pointer access)
    - If p is null, *p is skipped due to the early termination.

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

Shift Operations

- **Left Shift: `x << y`**
  - Shift bit-vector **`x`** left **`y`** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift: `x >> y`**
  - Shift bit-vector **`x`** right **`y`** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

Argument `x`	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

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

Topcis

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

Encoding Integers

Unsigned

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

Two's Complement

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

Sign Bit

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

- **C `short` 2 bytes long**

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

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

Two-complement Encoding Example (Cont.)

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

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

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, UINT_MAX
    - LONG_MAX, INT_MAX
    - LONG_MIN, INT_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

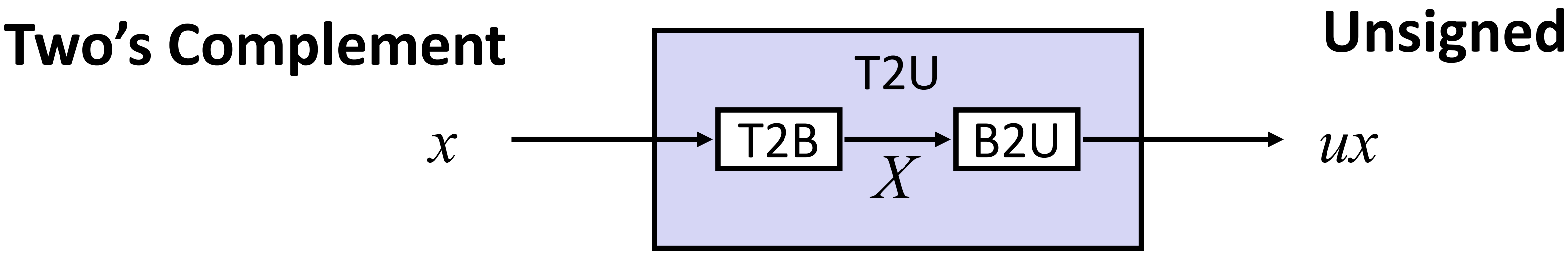
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **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

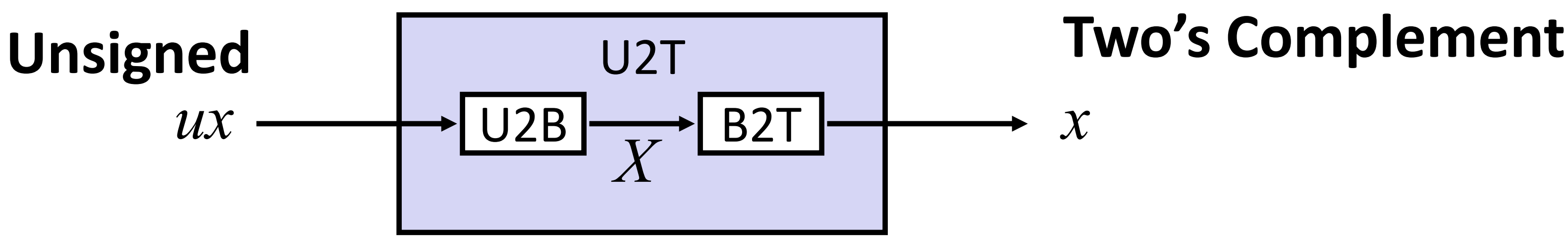
Topics

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

Mapping Between Signed & Unsigned



Maintain Same Bit Pattern



Maintain Same Bit Pattern

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

Mapping Signed ↔ Unsigned

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

→ T2U →

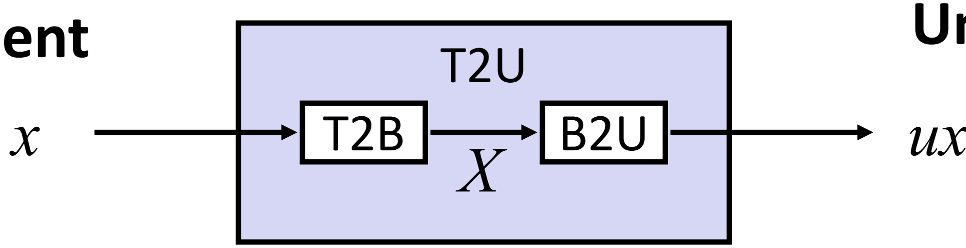
← U2T ←

Mapping Signed ↔ Unsigned

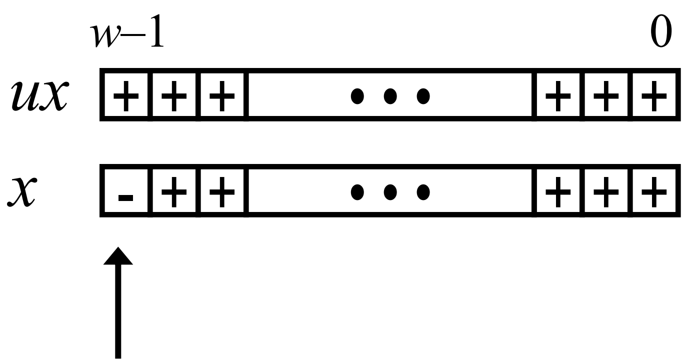
Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4	⟷	4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4	⟷	12
1101	-3		13
1110	-2		14
1111	-1		15

Relation between Signed & Unsigned

Two's Complement **Unsigned**



Maintain Same Bit Pattern



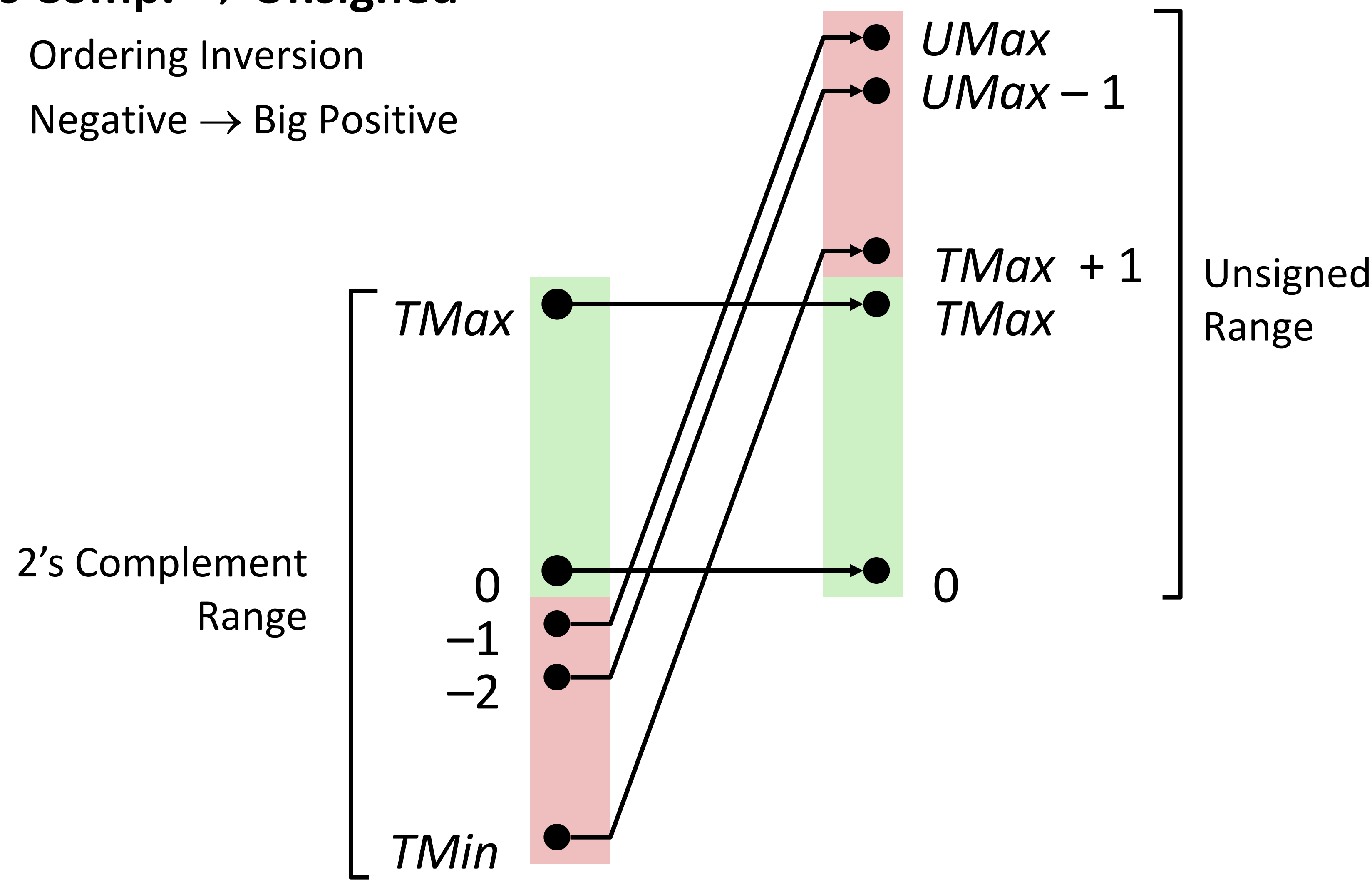
Large negative weight

becomes

Large positive weight

Conversion Visualized

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



Signed vs. Unsigned in C

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

    `0U, 4294967259U`
- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U

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

  - Implicit casting also occurs via assignments and procedure calls

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

Casting Surprises

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

-

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;
}
```

- **What if `maxlen` is negative?**
  - void *memcpy(void *dest, const void *src, **size_t** n);

Summary
Casting Signed ↔ Unsigned: Basic Rules

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

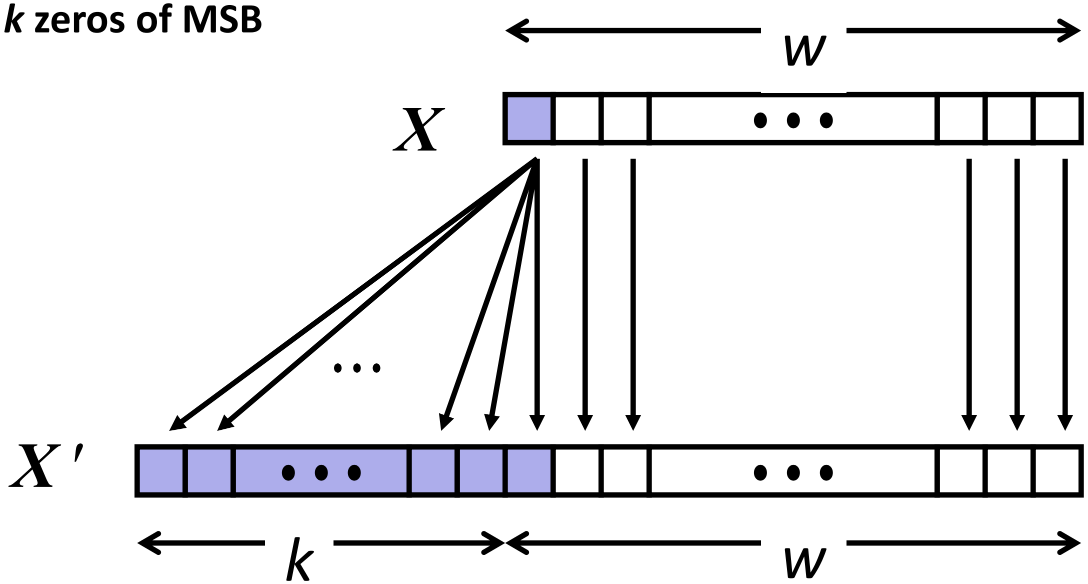
- **Expression containing signed and unsigned int**
  - `int` is cast to `unsigned`!!

Topics

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

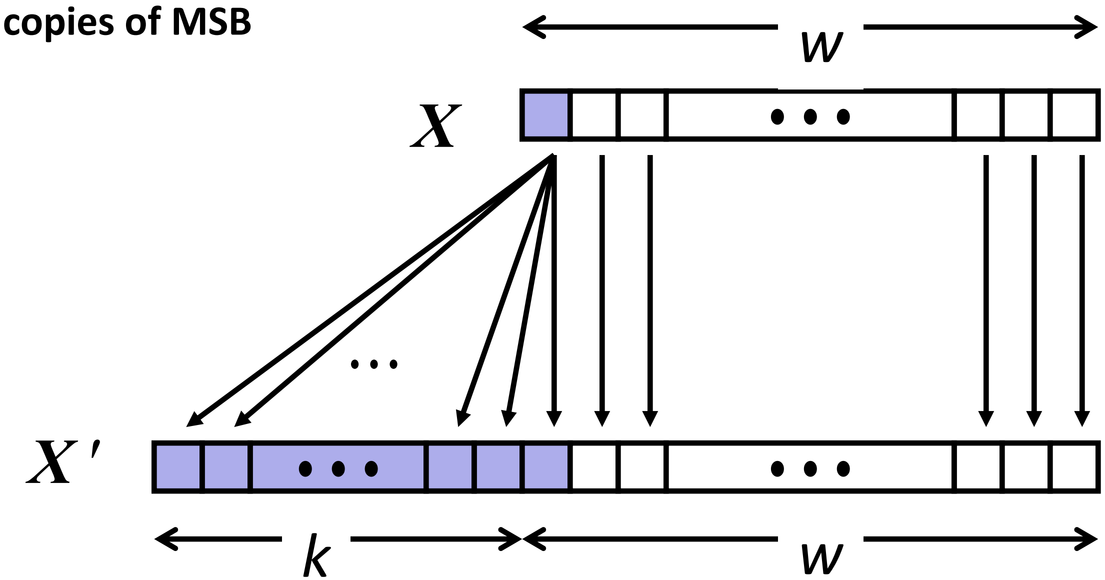
Zero Extension

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



Sign Extension

- **Task:**
  - Given w-bit signed integer x
  - Convert it to w+k-bit signed 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

Proof: Signed extension of Two's complement number

$$B2T_{w+k}([\underbrace{x_{w-1},\ldots,x_{w-1}}_{k\ \textbf{times}},x_{w-1},x_{w-2},\ldots,x_0]) = B2T_w([x_{w-1},x_{w-2},\ldots,x_0])$$

Apply induction to k

When k is 1:

$$\begin{aligned} B2T_{w+1}([x_{w-1},x_{w-1},x_{w-2},\ldots,x_0]) &= -x_{w-1}2^w + \textstyle\sum_{i=0}^{w-1} x_i 2^i \\ &= -x_{w-1}2^w + x_{w-1}2^{w-1} + \textstyle\sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}(2^w - 2^{w-1}) + \textstyle\sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}2^{w-1} + \textstyle\sum_{i=0}^{w-2} x_i 2^i \\ &= B2T_w([x_{w-1},x_{w-2},\ldots,x_0]) \end{aligned}$$

Truncation

```
int x = 53191;
short sx = (short) x;   /* -12345 */
int y = sx;             /* -12345 */
```

- **truncation on unsigned**
  - truncation from w-bit number to k-bit number is to zero the [w:k] bits by applying modulo 2^k
  - $B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k = [\sum_{i=0}^{w-1} x_i 2^i] \bmod 2^k$

    $= [\sum_{i=0}^{k-1} x_i 2^i] \bmod 2^k$

    $= \sum_{i=0}^{k-1} x_i 2^i$

    $= B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0])$
- **truncation on signed**
  - convert to unsigned, truncate, and revert to signed
  - $B2T_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = \mathrm{U2T}_k(B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k)$

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

Topics

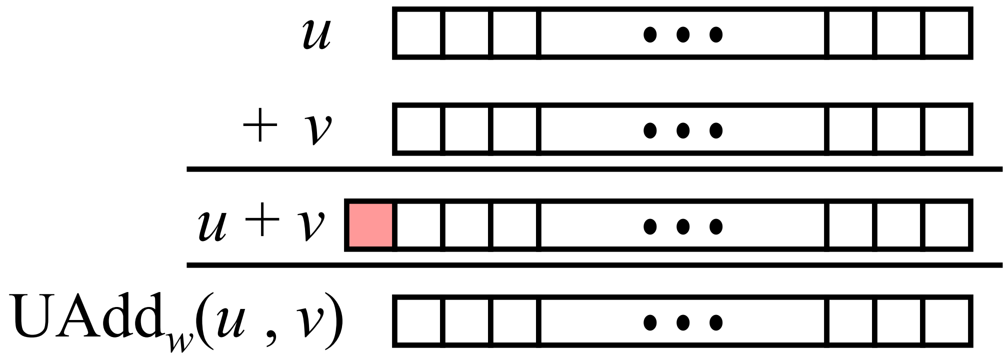
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
  - Use of Unsigned
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



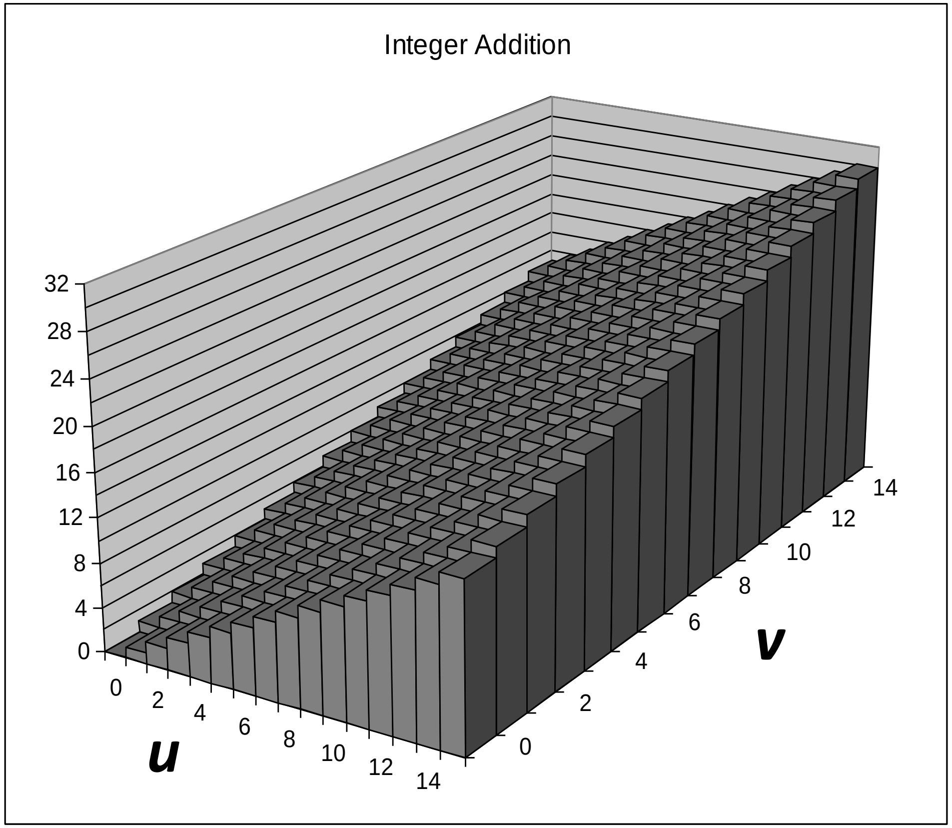
- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

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

Visualizing (Mathematical) Integer Addition

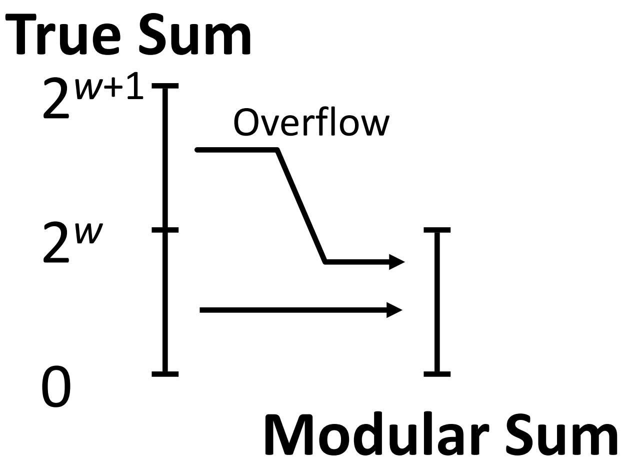
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

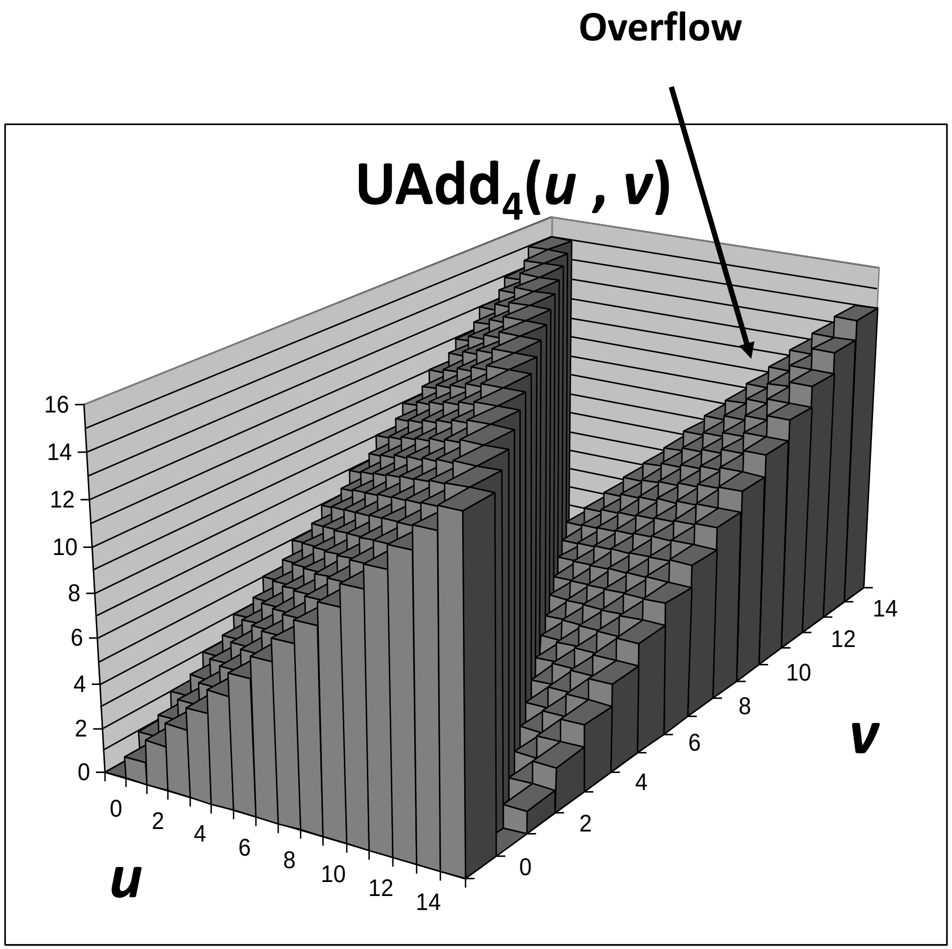
$\text{Add}_4(u , v)$



Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum ≥ 2^w
    - rotate through modulo operations
  - At most once



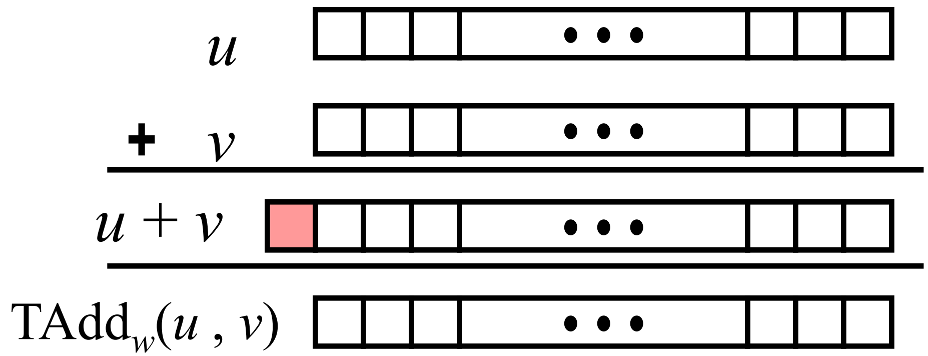


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



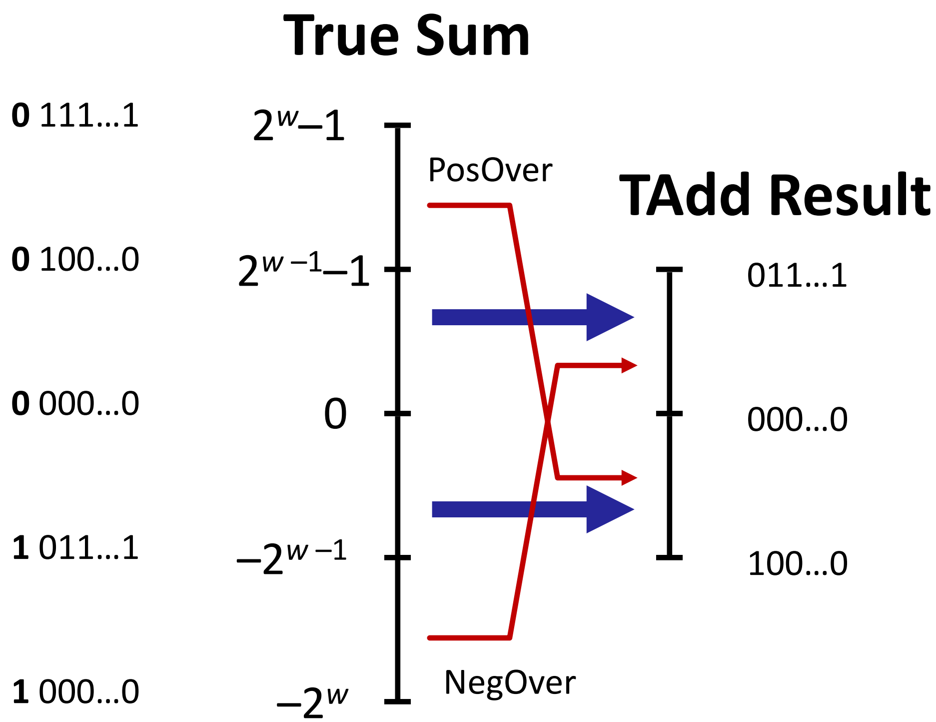
- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

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

  - Will give `s == t`
  - TAdd can be done by (1) converting to unsigned (2) performing UAdd, and (3) applying U2B & B2T

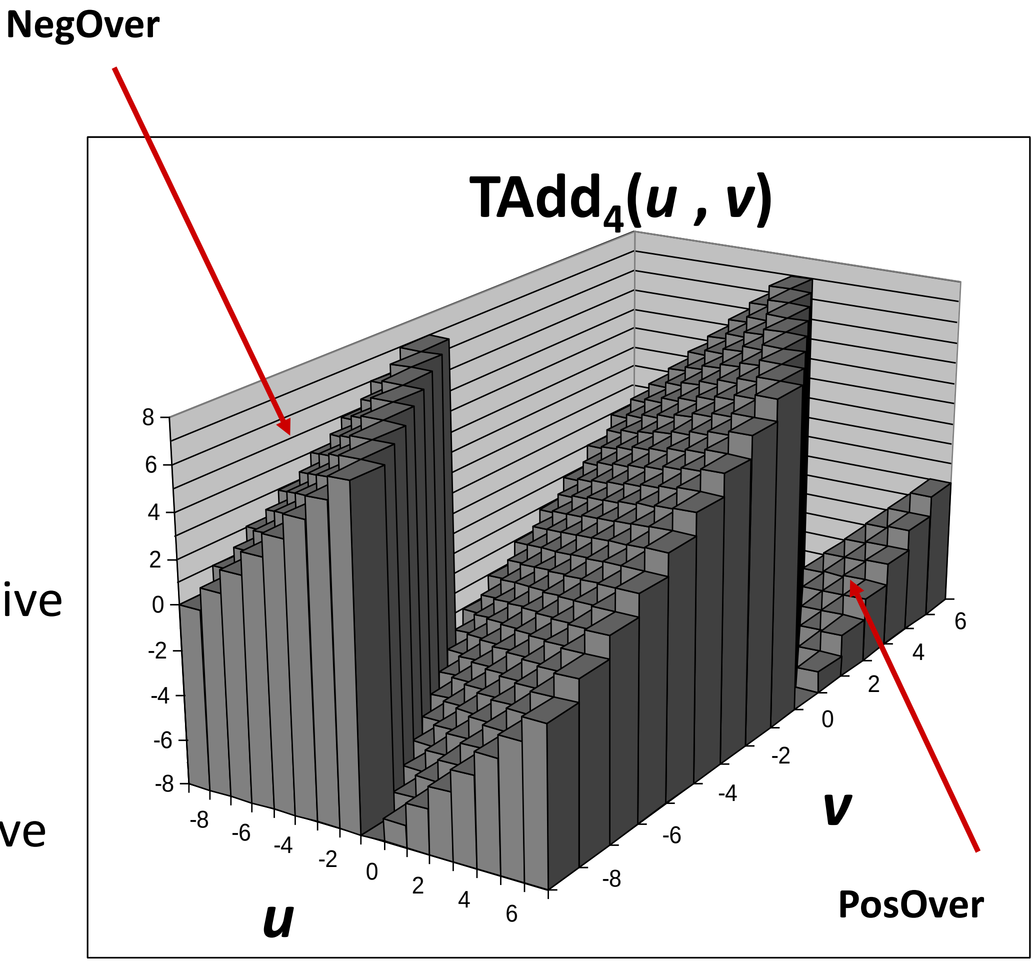
TAdd Overflow

- **Functionality**
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer
- **Positive Overflow**
  - overflows at positive
- **Negative Overflow**
  - overflows at negative



Visualizing 2's Complement Addition

- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Positive overflow
      - Becomes negative
  - If sum $< -2^{w-1}$
    - Negative overflow
      - Becomes positive



Multiplication

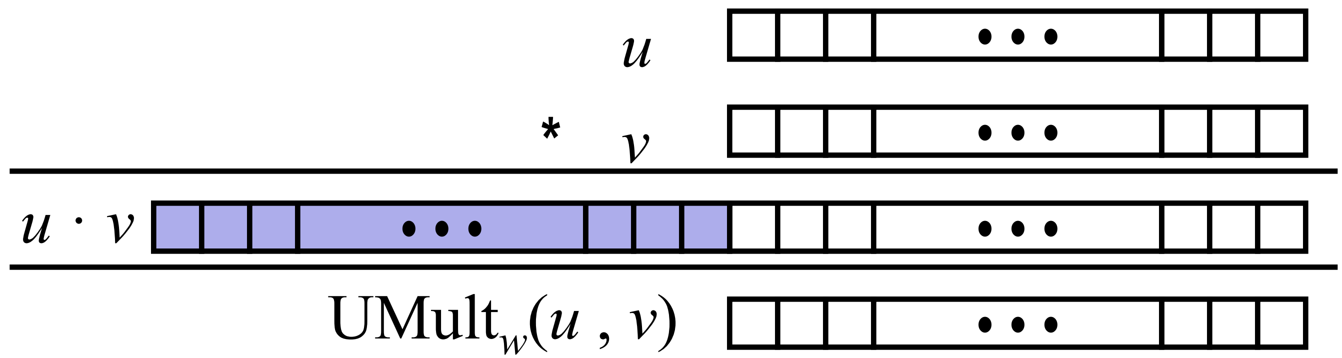
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results…**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" or "big num" arithmetic packages
  - without libraries, the system applies a w-bit truncation

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**

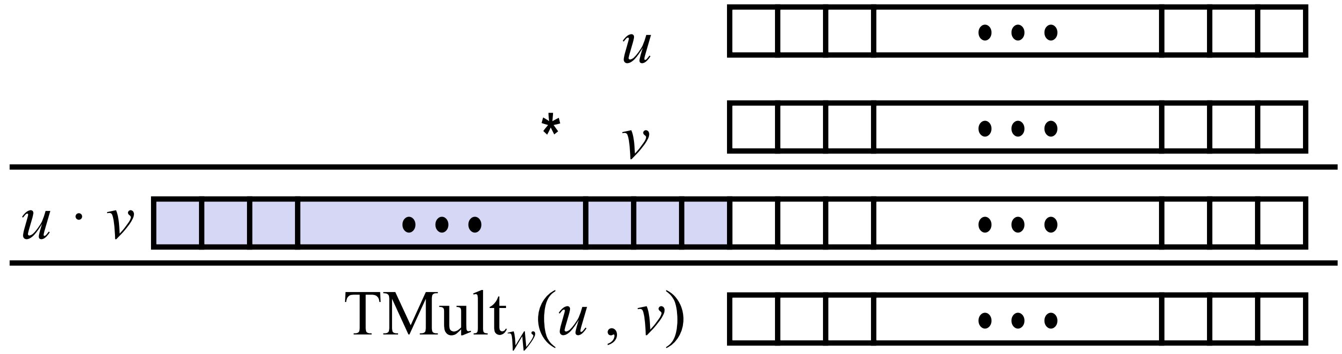
$\mathrm{UMult}_w(u, v) = (u \cdot v) \bmod 2^w$

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits

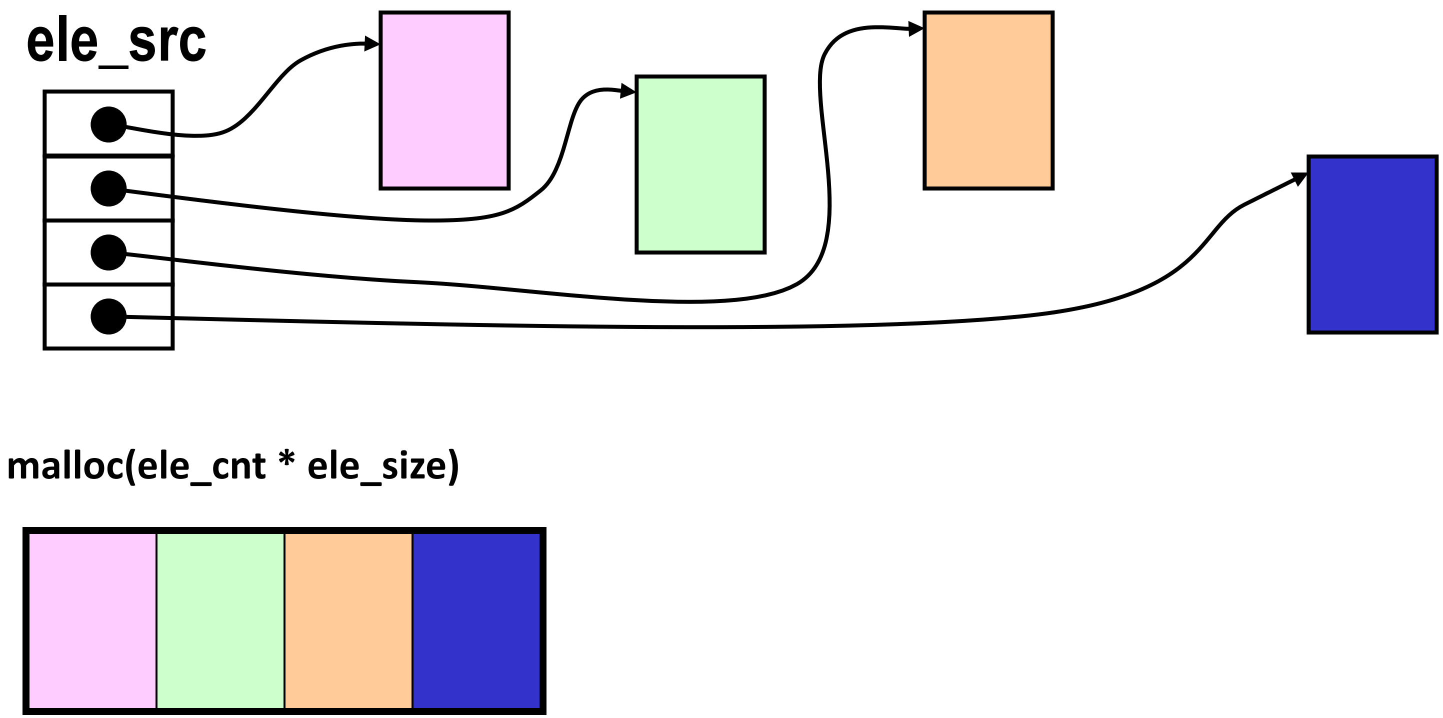


- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same
- **Signed multiplication can be done by (1) converting to unsigned (2) performing unsigned multiplication, and (3) applying U2B & B2T**

Code Security Example

- **SUN XDR library**
  - A library for data transfer between machines

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



Code Security Example

```
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;
}
```

- `ele_cnt` $= 2^{20} + 1$
- `ele_size` $= 4096 = 2^{12}$
- `malloced size (elc_cnt * ele_size) is` 4096

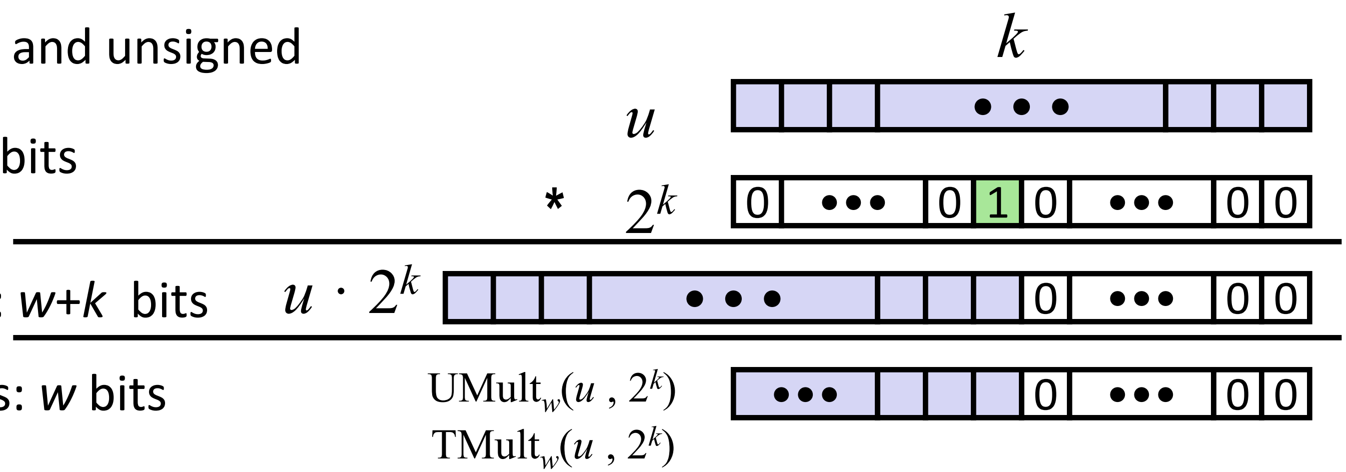
Power-of-2 Multiply with Shift

- **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

- **Compilers try to improve multiplications through addition and shift operations.**

C Function

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

Compiled Arithmetic Operations

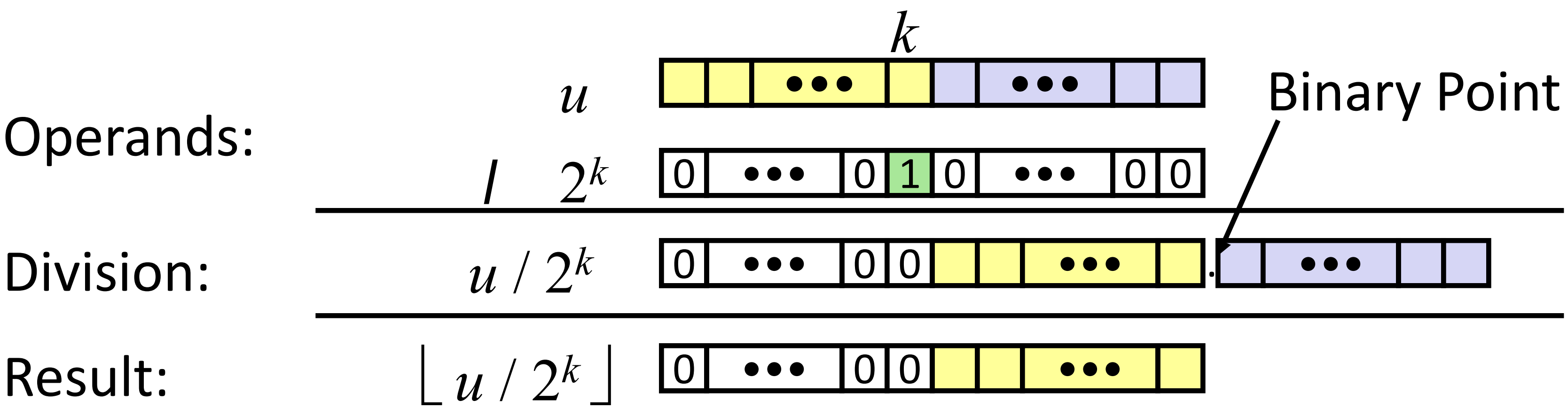
```
  leaq (%rax,%rax,2), %rax
  salq $2, %rax
```

Explanation

```
  t <- x + (x << 1)
  return t << 2;
```

Unsigned Power-of-2 Divide with Shift

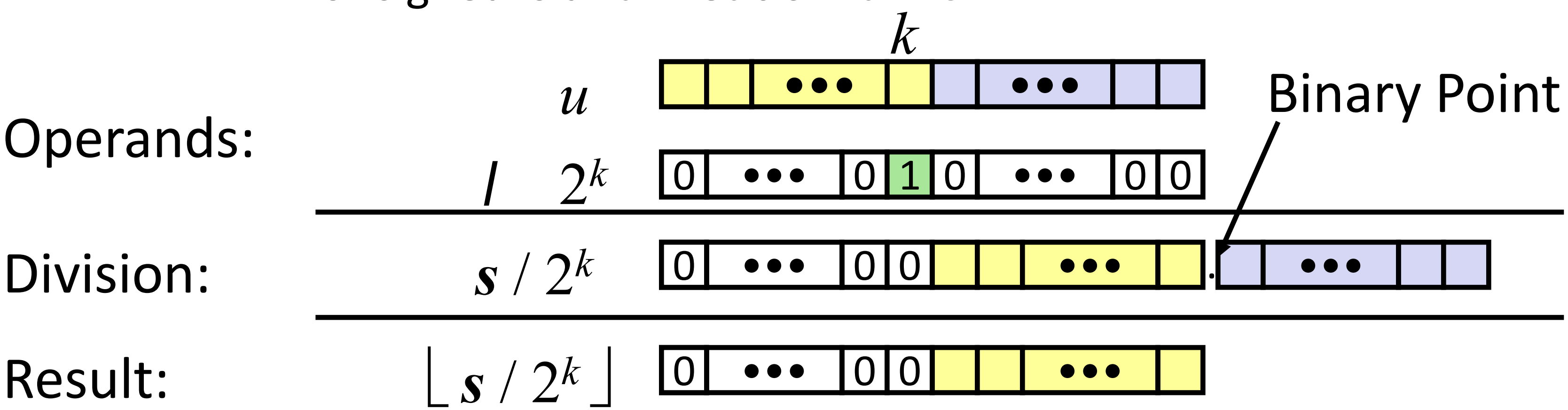
- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - Uses logical shift
    - ">>" for unsigned is logical shift in C



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Signed Power-of-2 Divide with Shift

- **Quotient of Positive Signed by Power of 2**
  - **`s >> k`** gives $\lfloor s / 2^k \rfloor$
  - arithmetic right shift
    - “>>” for signed is arithmetic shift in C



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Signed Power-of-2 Divide with Shift

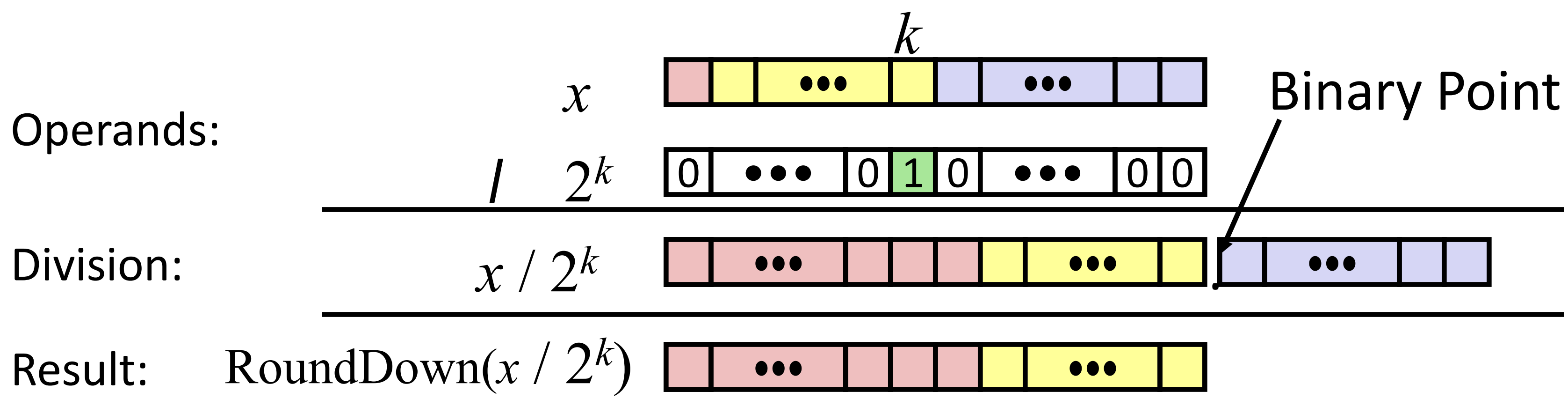
- **Quotient of Positive Signed by Power of 2**
  - Proof
    - $x = [x_{w-1}, x_{w-2}, \ldots, x_0]$ "2's comp of w bits"
      $x' = [x_{w-1}, x_{w-2}, \ldots, x_k]$ "upper w-k bits of x"
      $x'' = [x_{k-1}, x_{k-2}, \ldots, x_0]$ "lower k bits of x"
    - $x = 2^k x' + x''$

      where $0 \le x'' < 2^k$ and $x' = \lfloor x / 2^k \rfloor$
    - Apply arithmetic right shift to x by k
      - $x = [x_{w-1}, x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_k]$
      - which has the same bit pattern as x' sign extended to w-bit
    - As a result, arithmetic shifted x is equal to the quotient by power of 2

Signed Power-of-2 Divide with Shift

- **Quotient of Negative Signed by Power of 2**
  - `x >> k` gives $\lfloor x \;/\; 2^k \rfloor$
    - round down the result
    - ex) When the quotient is $\lfloor -1.5 \rfloor$, the shifted result is -2

      -3 / 2 → 0b1101 / 2 → 0b1110 = -2

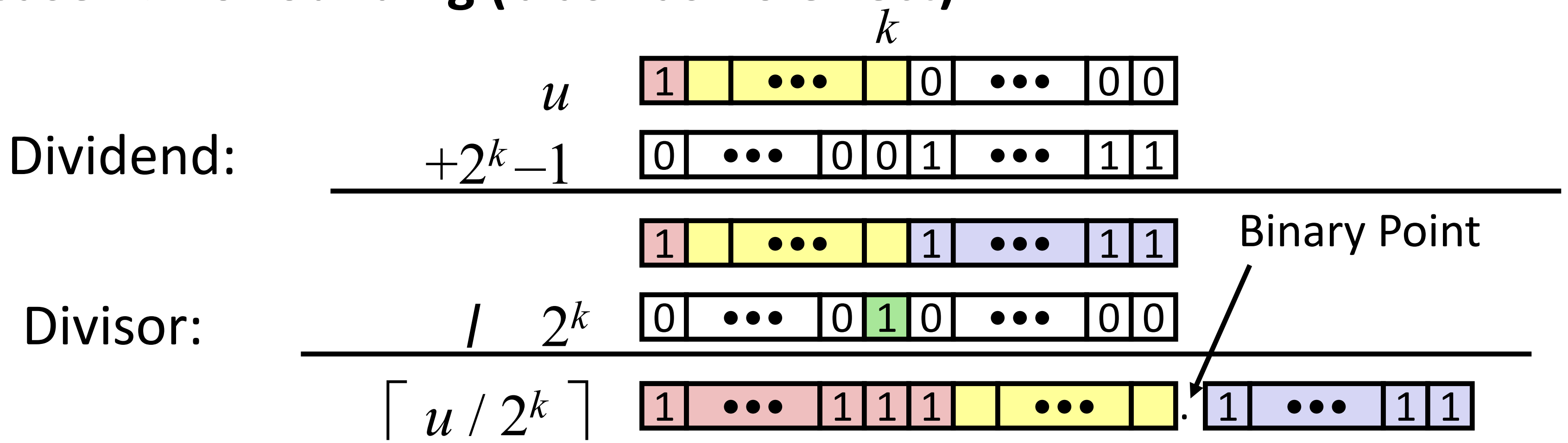


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

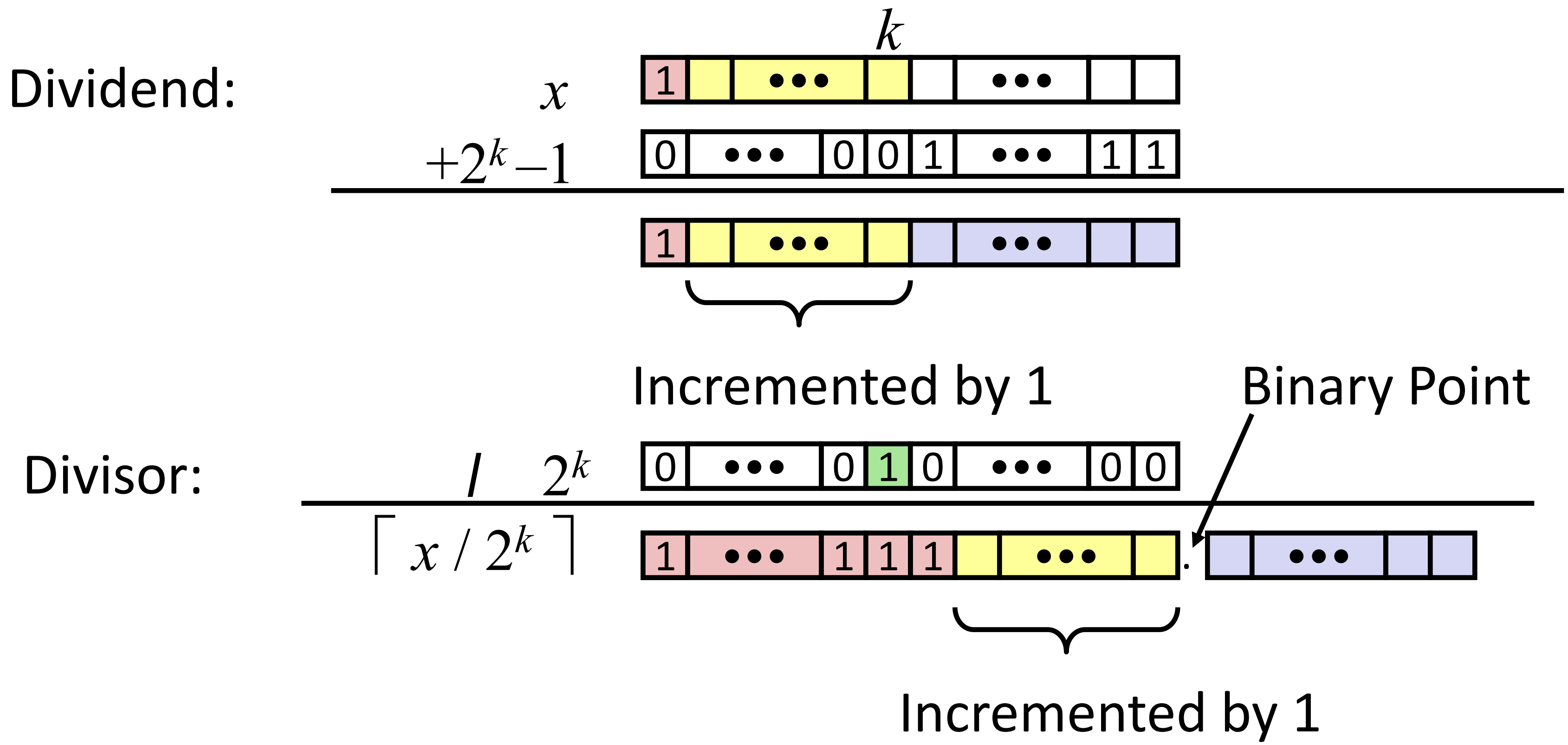
- **Quotient of Negative Signed by Power of 2**
  - Want $\lceil$ **`x / 2`**$^k \rceil$ (Round Toward 0)
  - Solution: add a bias to a negative dividend
  - $\lfloor$ **`(x+2`**^{k}**`-1)/ 2`**$^k \rfloor$
    - bias = **2**^{k}**–1**
    - In C: **`(x + (1<<k)-1) >> k`**

Case 1: No rounding (bias has no effect)



Correct Power-of-2 Divide (Cont.)

Case 2: Rounding (bias adds 1 to the result)



Compiled Unsigned Division Code

- **Logical shift on unsigned**
  - in Java, >>>

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

Compiled Signed Division Code

- **Arithmetic shift on signed**
  - In java, >>

C Function

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

Compiled Arithmetic Operations

```
  testq %rax, %rax  //%rax&%rax
  js    L4           //sign==1?
L3:
  sarq  $3, %rax
  ret
L4:
  addq  $7, %rax
  jmp   L3
```

Explanation

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

Topic

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

Be careful with 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 → *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?