Announcement

- **Limited Lab hours this week.**
  - See course webpage for time / location.
- **We will add more lab hours this coming week.**
- **GitHub repositories set up! Will update lab 1 and exercise 1 by tomorrow (will announce on Piazza).**

Bits, Bytes, and Integers (Cont.)

B&O Readings: 2.2-2.3

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Implement a pop_count function, cont

- **How do you implement pop_count, that counts the number of bits set in a 4 byte memory?**
  ex: pop_count(0x000000FF) = 8

```
int pop_count(unsigned int x) {
    int count = 0;
    for(; x != 0; x &= ~(x&(-x))) {
        count++;
    }
    return count;
}
```

The expression `(x&(-x))` computes a mask with a single 1 set at the least-significant position where x has a bit 1 set .

Implement a pop_count function

- **How do you implement pop_count, that counts the number of bits set in a 4 byte memory?**
  ex: pop_count(0x000000FF) = 8

```
#define MASK 0xF;
int count_arr[16] =
        {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4}

int pop_count(unsigned int x) {
    int count = 0;
    while(x != 0) {
        count += count_arr[x&MASK];
        x = x >> 4;
    }
    return count;
}
```

We can check 4 bits at a time!

Recap: What We Learned Thus Far

- The right shift behaves differently depending on whether an expression is signed versus unsigned.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition
    - How to detect overflow
  - Multiplication / Division
    - How to do these operations with shifts

Casting Between 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;
```

Rule of Thumb: Keep bit representations and reinterpret!

  - Implicit casting also occurs via assignments and procedure calls

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

Use -Werror and -Wall compiler flag to catch this!

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 **<, >, ==, <=, >=**

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);
    . . .
}
```

You end up allowing the user to copy lots of data from kernel!

Recap: What We Learned About Casting

- **C allows one to cast from signed to unsigned and vise versa:**
  - Bit pattern is maintained
  - But reinterpreted
  - Can have unexpected effects: adding or subtracting 2^w
- **When an expression contains both signed and unsigned, it's implicitly treated as unsigned.**

- **Understanding these quirks in C allows you to write correct and secure code.**

When Should I Use Unsigned?

- ***Don't*** **Use Just Because the Number are 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 Using Bits to Represent Sets**
  - Logical right shift, no sign extension

Expression Evaluation Puzzles

- **Assuming int type (32 bits)**

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

TMIN = -2147483647-1 (0x80000000)

TMAX = 2147483647 (0x7FFFFFFF)

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition
    - How to detect overflow
  - Multiplication / Division
    - How to do these operations with shifts
- Summary

Extension

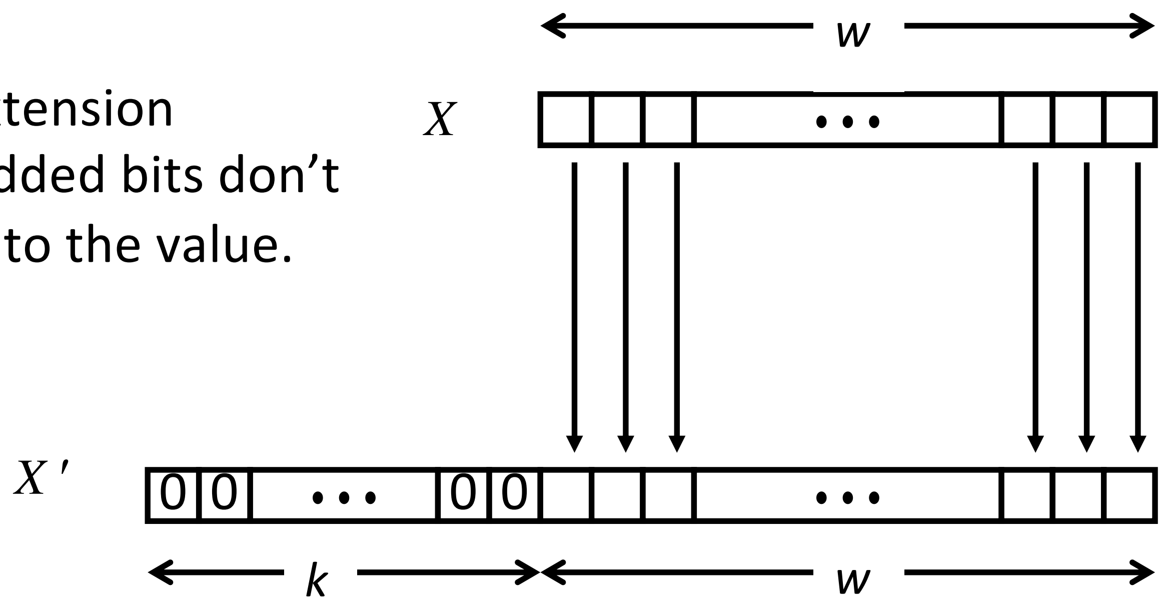
- **When operating with types of different width, C automatically performs extension**
- **Converting from smaller to larger integer data type**
  - Given *w*-bit integer *X*
  - Convert it to *w*+*k*-bit integer *X* ′ **with same value**
- **Two different kinds of extension:**
  - zero extension: used for unsigned data types
    (similar: >> uses logical right shift for unsigned values)
  - sign extension: used for signed data types
    (similar: >> uses arithmetic right shift for signed values)

```
unsigned short sx = 361;
unsigned int x = sx; /* use zero extension */

short sy = -361;
int y = sy; /* use sign extension */
```

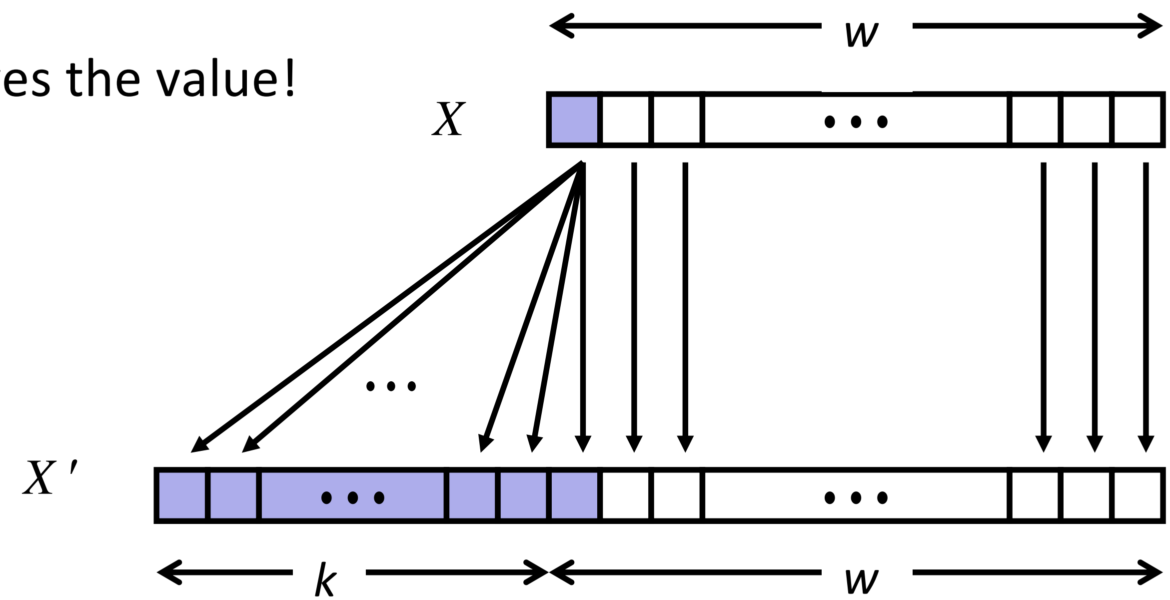
Zero Extension for Unsigned type

- **What It Does:**
  - Given w-bit unsigned integer X
  - Convert it to $w+k$-bit unsigned integer X' with same value
- **Rule:**
  - Prepend k bits of 0:
  - $X' = \underbrace{0, \ldots, 0}_{k \text{ copies}}, x_{w-1}, x_{w-2}, \ldots, x_0$
  - Easy to see that the extension preserves the value: added bits don't contribute any weight to the value.



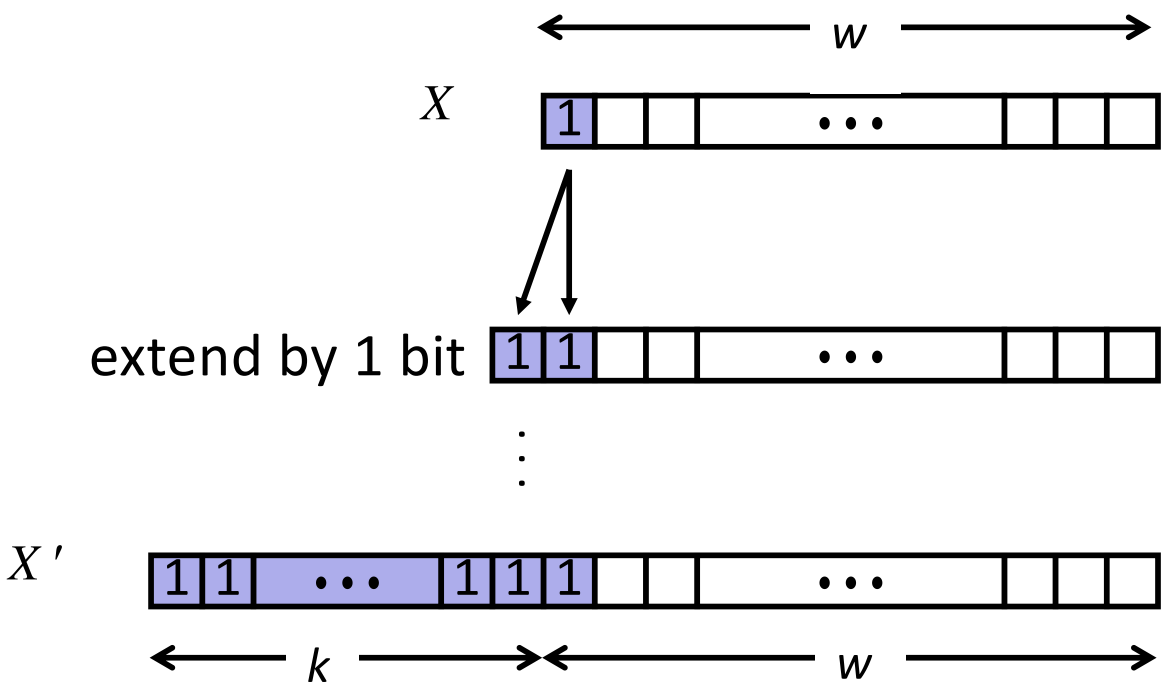
Sign Extension

- **What It Does:**
  - Given w-bit signed integer X
  - Convert it to w+k-bit unsigned integer X' with same value
- **Rule:**
  - Make k copies of the sign bit:
  - $X' = \underbrace{x_{w-1},\ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2},\ldots, x_0$
  - The extension preserves the value!



Sign Extension Preserves the Value

- ***X* is positive:**
  - easy to see: 0 bits don't add weight
- ***X* is negative:**



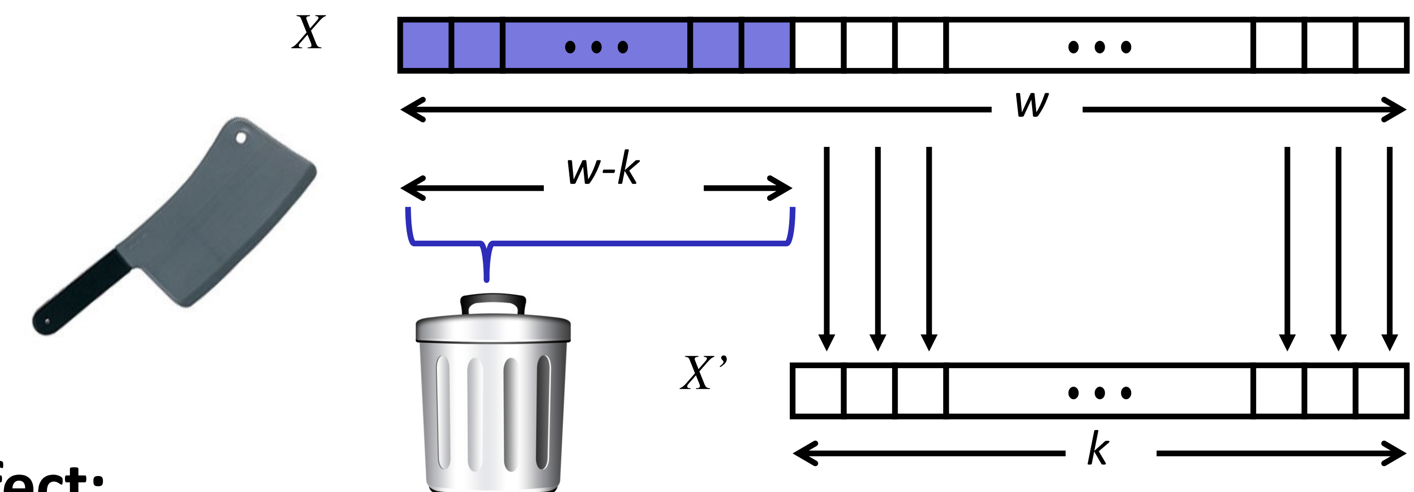
MSB contributed weight -2^{w-1}

The 2^{nd} MSB and MSB contributed weight $2^{w-1} - 2^{w} = -2^{w-1}$

We can show that sign extension does not change the value by inducting on *k*.

Truncation

- **Task:**
  - Given *w*-bit signed integer *X*
  - Convert it to *k*-bit integer *X′* with same value (maybe…)
- **Rule:**
  - Drop high-order *w*-*k* bits



- **Effect:**
  - Can change the value of *X* (overflow)
  - Unsigned: mathematical mod on *X*
  - Signed: reinterpret the bits (add -2^k if the most-significant bit is 1)

Code Puzzle

- **What is the output of the following code?**
  Assume that int is 32 bits, short is 16 bits, and the representation is two's complement.

```
unsigned short y = 0xFFFF;
int x = y;

printf("%x", x); /* print in hexadecimal */
```

A) 0xFFFF B) 0xFFFFFFFF C) None of the above

Answer: A) 0xFFFF

DO NOT write code like this!

Recap: What We Learned Thus Far

- **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
  - Unsigned: keep the last k bits; like a mod operation
  - Signed: keep the last k bits, and reinterpret the bits as signed
  - For small numbers yields expected behavior; for large number, can change the value.

- **Know how an expression with mixed types is treated but avoid writing code like that to avoid unintended behavior!**

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition
    - How to detect overflow
  - Multiplication / Division
    - How to do these operations with shifts
- Summary

Integer Addition

- **Rule of thumb 1: Do the normal binary operations assuming enough bits, and chop off the extra bits that cannot fit.**

- **Rule of thumb 2: The hardware does not care about whether the variables are signed versus unsigned; the operations are the same for both.**

Unsigned Addition

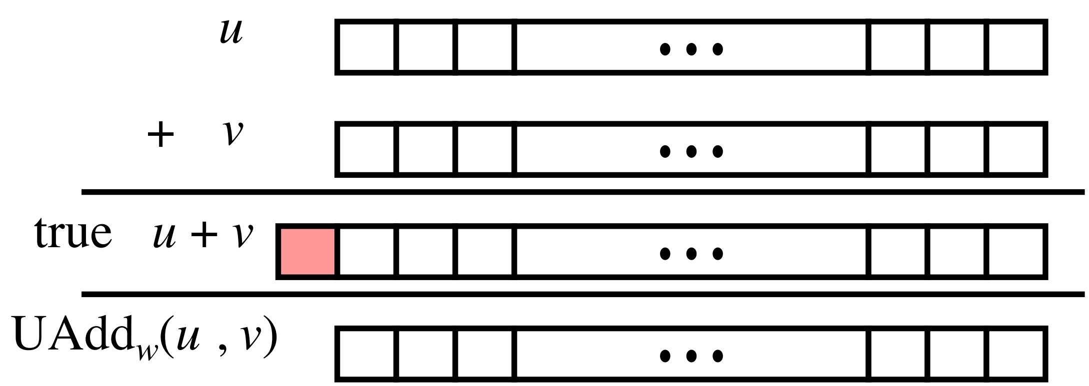
$$0 \le u, v \le 2^w - 1$$

$$0 \le u + v \le 2^{w+1} - 2$$

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **Standard Addition Function**
  - Ignores carry output

How to Detect Overflow in UAdd?

Hint: try performing UAdd with 4-bit values.

- **What's the range of value that a 4-bit variable can represent?**
- **How does one interpret the result with overflow?**

Detecting Overflow in Unsigned Addition

- **When overflow:**
  - Assume *w*-bit operands
  - If overflow, true sum $\geq 2^w$ but can overflow by 1 bit only
  - Uadd(u,v) = true sum mod 2^w

$$= u + v - 2^w$$

$$= u + \underbrace{(v - 2^w)}_{<0} \quad \text{or} \quad v + \underbrace{(u - 2^w)}_{<0}$$

- **To detect overflow in UAdd, check if UAdd(u,v) < u or < v**

Two's Complement Addition

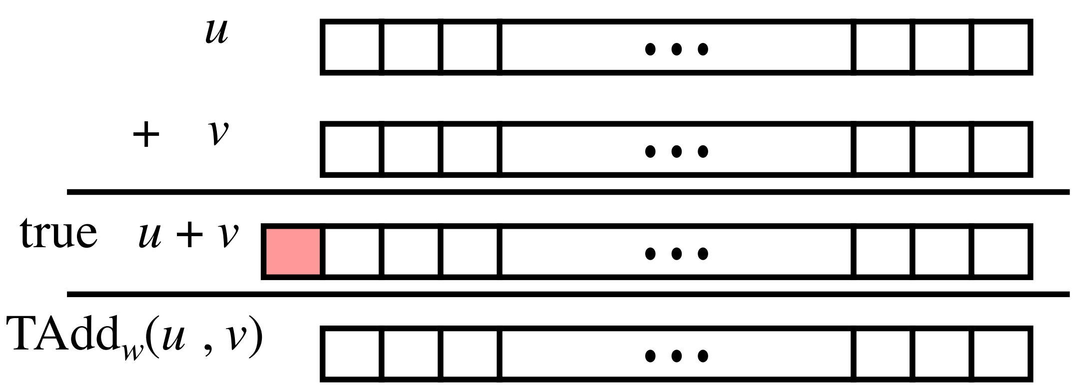
$$-2^{w-1} \le u, v \le 2^{w-1}-1$$

$$-2^{w} \le u + v \le 2^{w} - 2$$

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **TAdd and UAdd have Identical Bit-Level Behavior**

```
int s, t, u, v;
... /* initialize their values */
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
assert(s == t); /* always true! */
```

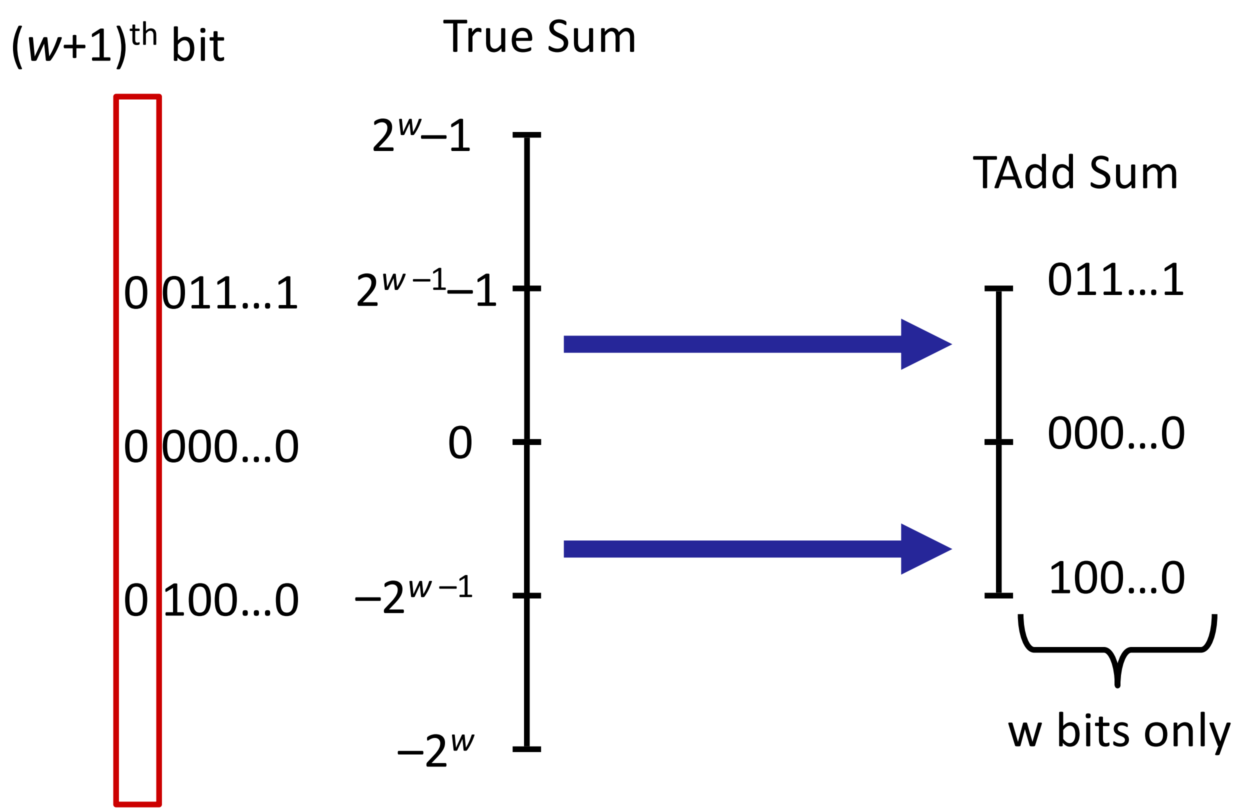
Same bit pattern, different interpretation for sign vs. unsigned.

How to Detect Overflow in TAdd?

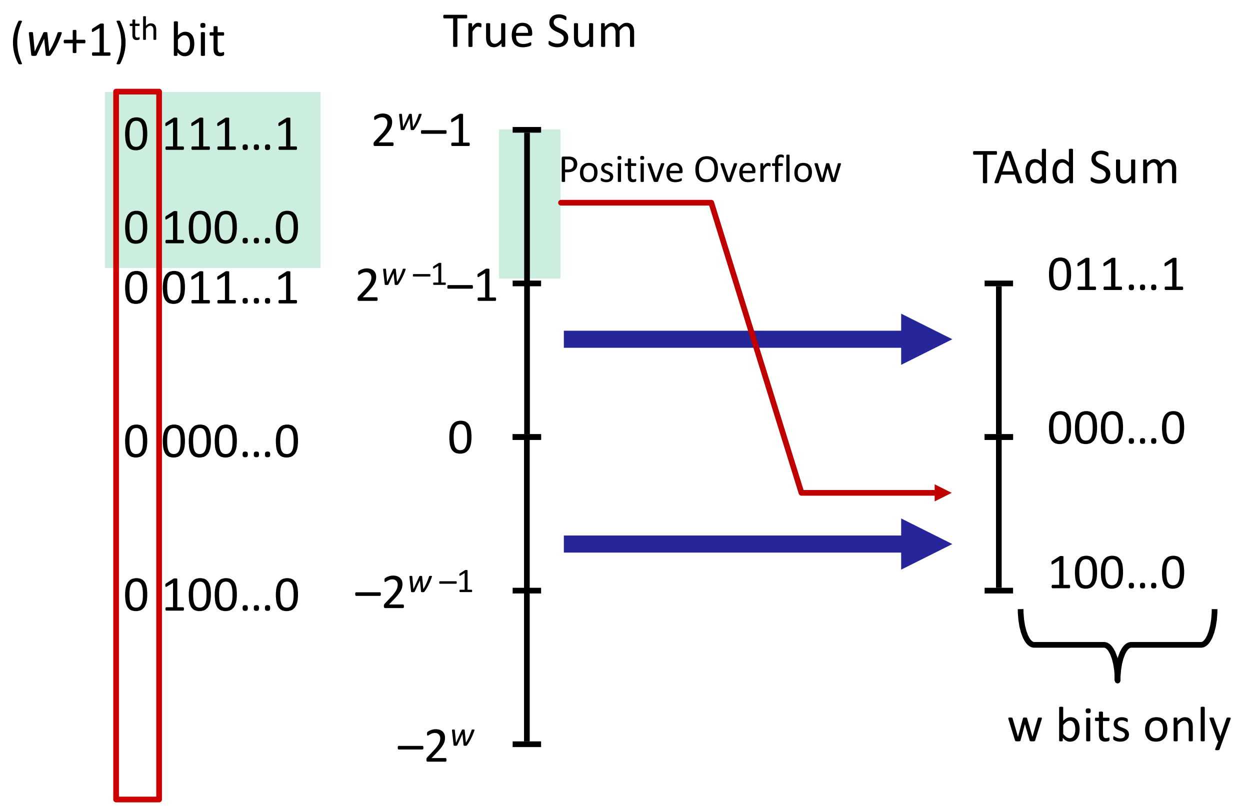
Hint: try performing TAdd with 4-bit signed values.

- **What's the range of value that a 4-bit signed variable can represent?**
- **Try adding two largest values together**
  - 0111 + 0111 = 1110 (-2)
  - Overflow to the MSB
- **Try adding two smallest values together**
  - 1000 + 1000 = 10000 -> 0000 (0)
  - Overflow to a bit that gets truncated
  - The MSB must be 0

TAdd Overflow

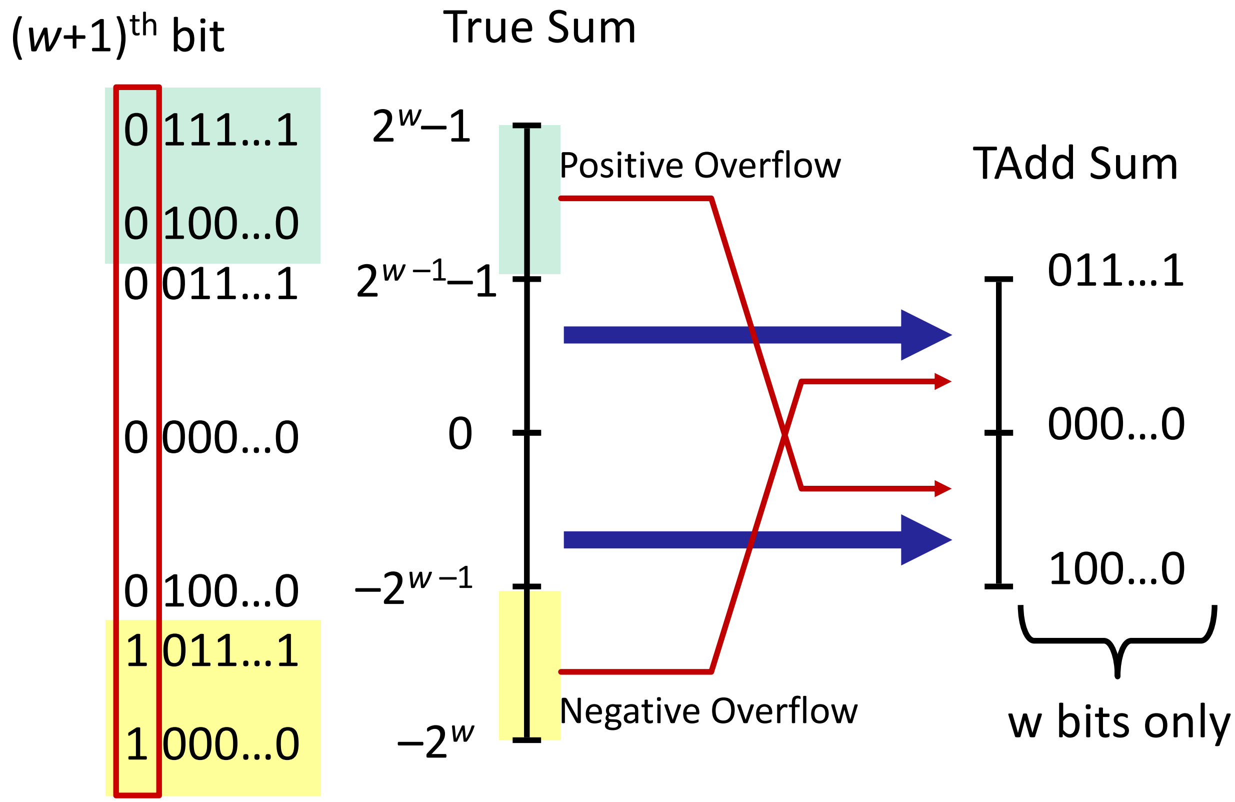


TAdd Overflow



- **Positive overflow:**
  - Adding two positive values, where $(u + v) > 2^{w-1}–1$ (TMax)
  - w^{th} bit contributes to true sum weight of 2^{w-1} but to TAdd sum -2^{w-1}
  - TAdd sum = $\underbrace{\text{true sum}}_{< (2^{w}-1)} - 2^{w}$ (negative)

TAdd Overflow



- **Negative overflow:**
  - Adding two negative values, where (u + v) < -2^{w-1} (TMin)
  - Missing the carry $(w+1)^{th}$ bit (which would have contributed weight -2^w)
  - TAdd sum = $\underbrace{\text{true sum}}_{< (2^w - 1)} + 2^w$ (positive)

Detecting Overflow in Two's Complement Addition

- **Positive overflow:**
  - the carry-bit overflow into the most-significant bit (MSB)
  - true sum $\geq 2^{w-1}$ -1, MSB contributes negative weight instead of positive
  - TAdd(u,v) = (u+v) – 2^w (which results a negative value)

- **Negative overflow:**
  - If true sum $< -2^{w-1}$, the carry-bit overflow into the bit that got truncated
  - The $(w+1)^{th}$ bit would have contributed -2^w weight
  - TAdd(u,v) = (u+v) + 2^w (which results positive value)

- **To detect overflow in Tadd, check if signs of input operands and output differ.**

Recap: What We Learned Thus Far

- **For *w*-bit operands, need *w*+1 bits for true sum**
- **For fixed-width integer addition, do the usual addition and truncate extra bits**

- **For unsigned addition**
  - Check for overflow by checking if the output is smaller than either input
- **For two's complement addition**
  - Can only overflow when both operands have the same sign
  - Check for overflow by checking if the signs of inputs and output differ

- **Knowing when overflow might occur and how to check for them enables you to write correct code.**

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition
    - How to detect overflow
  - Multiplication / Division
    - How to do these operations with shifts
- **Summary**

Integer Multiplication

- Rule of thumb 1: Do the normal binary operations assuming enough bits, and chop off the extra bits that cannot fit.

- Rule of thumb 2: The hardware does not care about whether the variables are signed versus unsigned; the operations are the same for both.

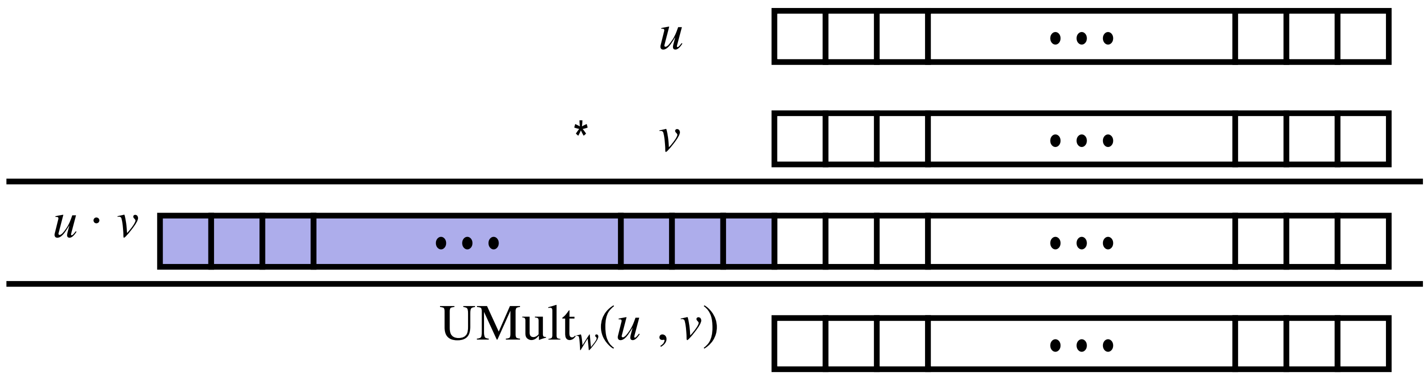
- Same Rules as Integer Addition!

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**

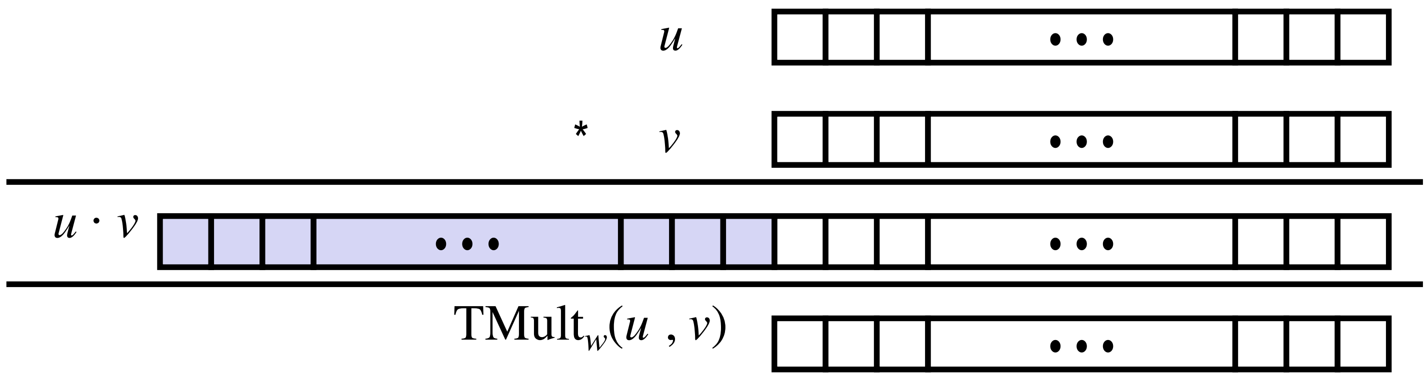
  $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

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
  - Same treatment as unsigned, just reinterpret the bits

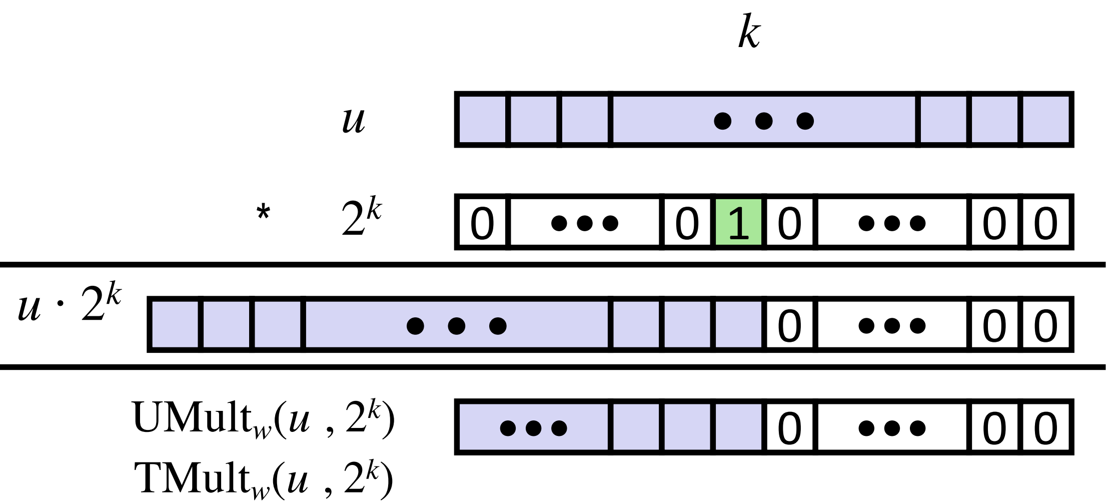
Power-of-2 Multiply with Shift

- **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 * 2`**^{3}
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Power-of-2 Multiply with Shift Example

- Q: How do you computing $X \cdot 6$ by using left shift?

Power-of-2 Multiply with Shift Example

- **Q: How do you computing *X · 6* by using left shift?**

6 = 0…0110 (in binary)

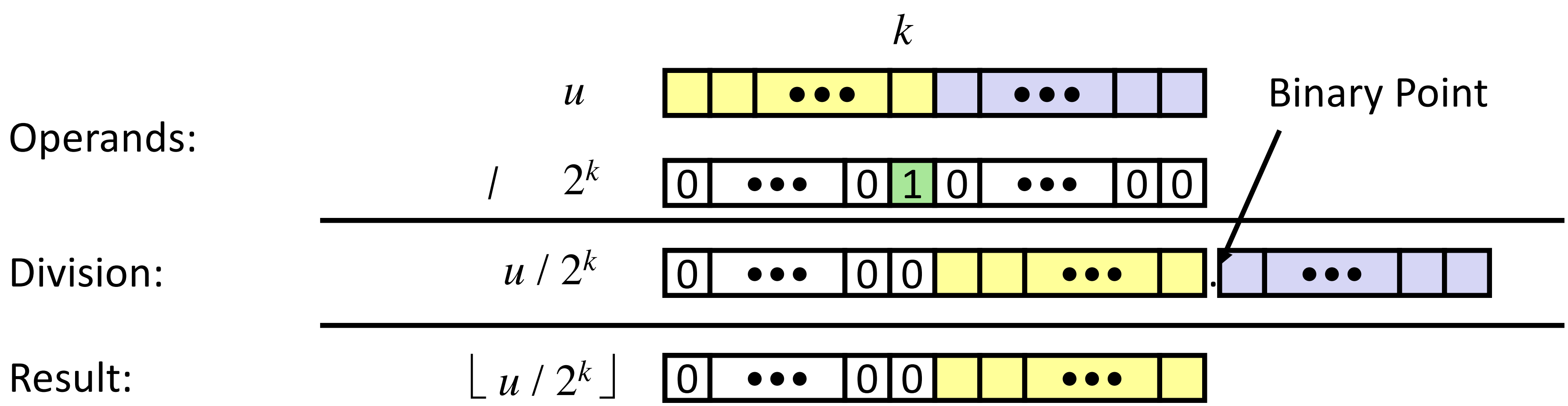
$$
\begin{aligned}
x \cdot 6 &= x \cdot (2^2 + 2^1) \\
&= x \texttt{ << } 2 + x \texttt{ << } 1
\end{aligned}
$$

Or, equivalently (assuming no overflow),

$$
\begin{aligned}
x \cdot 6 &= x \cdot (2^3 - 2^1) \\
&= x \texttt{ << } 3 - x \texttt{ << } 1
\end{aligned}
$$

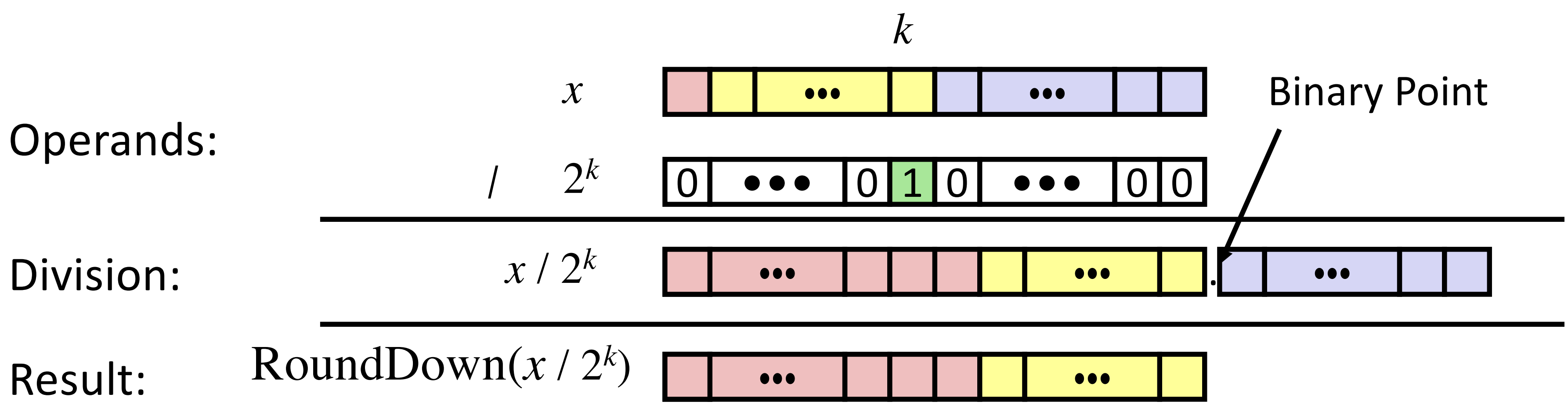
Unsigned Power-of-2 Divide with Shift

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



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`** (normal division rounds towards 0)



Recap: What We Learned Thus Far

- **Integer Multiplication:**
  - For *w*-bit operands, need 2*w* bits for true product
  - Signed vs unsigned values are treated the same way
- **Multiplication by 2^k can be done with left shift**
- **Division by 2^k can be done with right shift**
  - Unsigned: logical shift
  - Signed: arithmetic shift
  - Watch out: for negative numbers, round away from zero!

- **Use 2w-bit integer data type for w-bit multiplications to avoid overflow.**
- **Whenever possible, use shifts for multiplication / division**

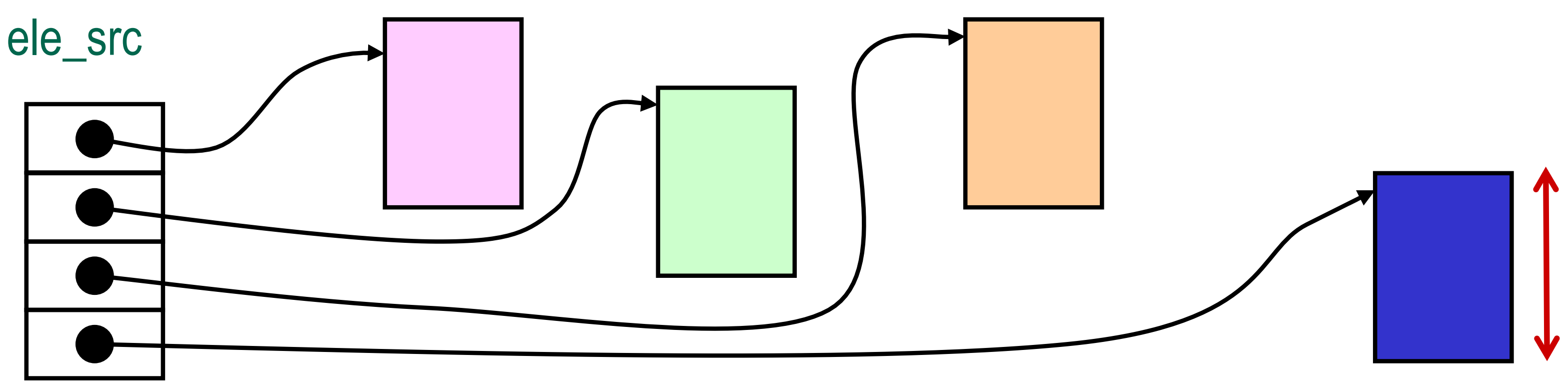
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition
    - How to detect overflow
  - Multiplication / Division
    - How to do these operations with shifts
- **Summary**

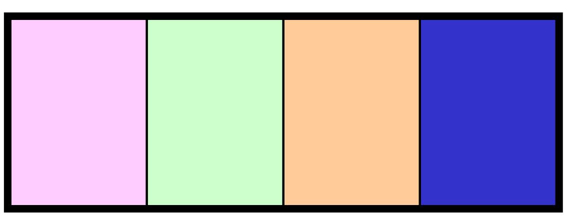
Code Security Example

- **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)



"In this array I've got pointers to 4 chunks of data. I'd like you to allocate a block of memory and store all these chunks in that block."

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 on 32-bit System

```
malloc(ele_cnt * ele_size)
```

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

XDR Vulnerability on 32-bit System

```
malloc(ele_cnt * ele_size)
```

- **What if:**
  - **`ele_cnt`** $= 2^{20} + 1$
  - **`ele_size`** $= 4096 \qquad = 2^{12}$
  - Allocation = $2^{12} (2^{20} + 1) = 2^{32} + 2^{12}$

    = `4096 bytes` (just shy of the 4.3 billion needed)

    You're going to overwrite a lot of data in your program.

Integer C Puzzles

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

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

Integer C Puzzles Answers

1. No (TMin can overflow)
2. No (0x0100000…0 shift becomes Tmin)
3. Yes (same bits reinterpreted)
4. Yes (the 3 LSB are all 1's, after shift, 0x1100…0)
5. No (actually it's never true, since –1 is evaluated as unsigned)
6. No (think of TMin … the range of signed value is asymmetric)
7. No (overflow)

Integer C Puzzles Answers, Cont

8. No (overflow 1. No (TMin can overflow)
9. Yes (the range is asymetric, but for every positive value representable, its negative value is also within the range)
10. No (again, TMin)
11. No (counter example: 0)
12. Yes (since it always rounds towards 0)
13. No (x/8 will round towards 0 if x < 0)
14. No (simple counter example: 0)