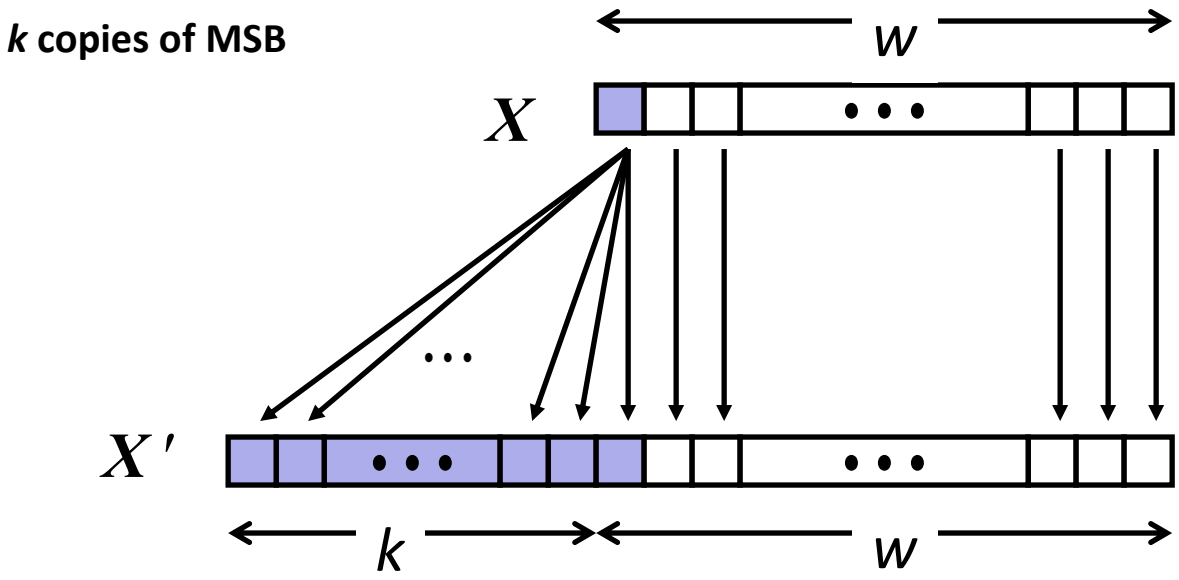
Bits, Bytes, and Integers

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **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:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

```
1   short sx = -12345;        /* -12345 */
2   unsigned short usx = sx; /*  53191 */
3   int   x = sx;             /* -12345 */
4   unsigned  ux = usx;       /*  53191 */
5
6   printf("sx  = %d:\t", sx);
7   show_bytes((byte_pointer) &sx, sizeof(short));
8   printf("usx = %u:\t", usx);
9   show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10  printf("x   = %d:\t", x);
11  show_bytes((byte_pointer) &x, sizeof(int));
12  printf("ux  = %u:\t", ux);
13  show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run on a 32-bit big-endian machine using a two's-complement representation, this code prints the output

```
sx  = -12345:   cf c7
usx = 53191:    cf c7
x   = -12345:   ff ff cf c7
ux  = 53191:    00 00 cf c7
```

Sign Extension

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program.

```
1    short sx = -12345;        /* -12345   */
2    unsigned uy = sx;         /* Mystery! */
3
4    printf("uy  = %u:\t", uy);
5    show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

```
uy = 4294954951:  ff ff cf c7
```

This shows that, when converting from short to unsigned, the program first changes the size and then the type. That is, (unsigned) sx is equivalent to (unsigned) (int) sx, evaluating to 4,294,954,951, not (unsigned) (unsigned short) sx, which evaluates to 53,191.

Truncating

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

53191 = 00000000 00000000 11001111 11000111

-12345 = 11001111 11000111

- When truncating a *w*-bit number $x = [x_{w-1}, x_{w-2}, \ldots, x_0]$ to a *k*-bit number, we drop the high-order $w - k$ bits
- Truncating a number can alter its value—a form of overflow.
- For an unsigned number *x*, the result of truncating it to *k* bits is equivalent to computing $x \bmod 2^k$

$$
\begin{aligned}
B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\
&= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\
&= \sum_{i=0}^{k-1} x_i 2^i \\
&= B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0])
\end{aligned}
$$

In this derivation, we make use of the property:

$$2^i \bmod 2^k = 0 \text{ for any } i \geq k$$

The same is applicable for signed numbers

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

Practice

Suppose we truncate a 4-bit value (represented by hex digits 0 through F) to a 3-bit value (represented as hex digits 0 through 7). Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's-complement interpretations of those bit patterns.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	_____	0	_____
2	2	2	_____	2	_____
9	1	9	_____	−7	_____
B	3	11	_____	−5	_____
F	7	15	_____	−1	_____

Today: Bits, Bytes, and Integers

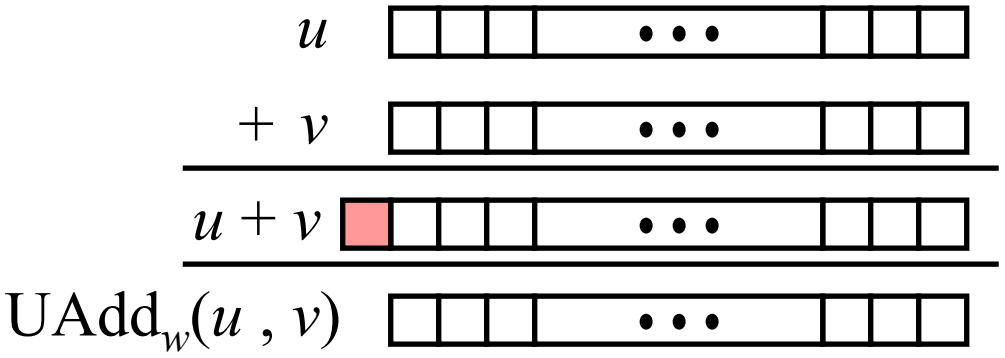
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

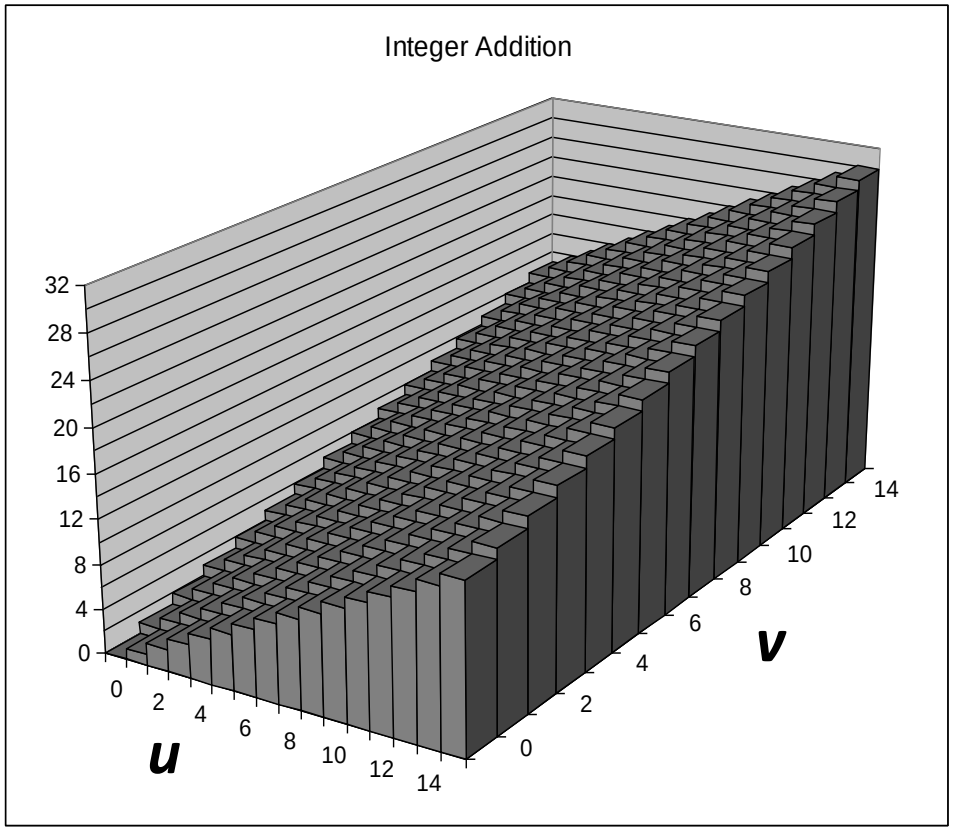
$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

For example, consider a 4-bit number representation with $x = 9$ and $y = 12$, having bit representations [1001] and [1100], respectively. Their sum is 21, having a 5-bit representation [10101]. But if we discard the high-order bit, we get [0101], that is, decimal value 5. This matches the value 21 mod 16 = 5

Visualizing (Mathematical) Integer Addition

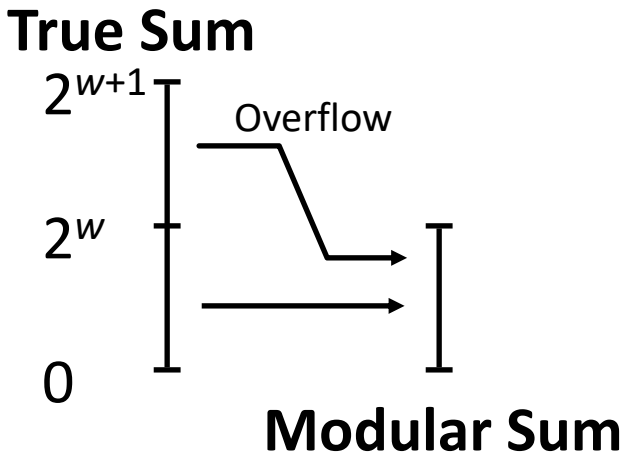
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u , v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

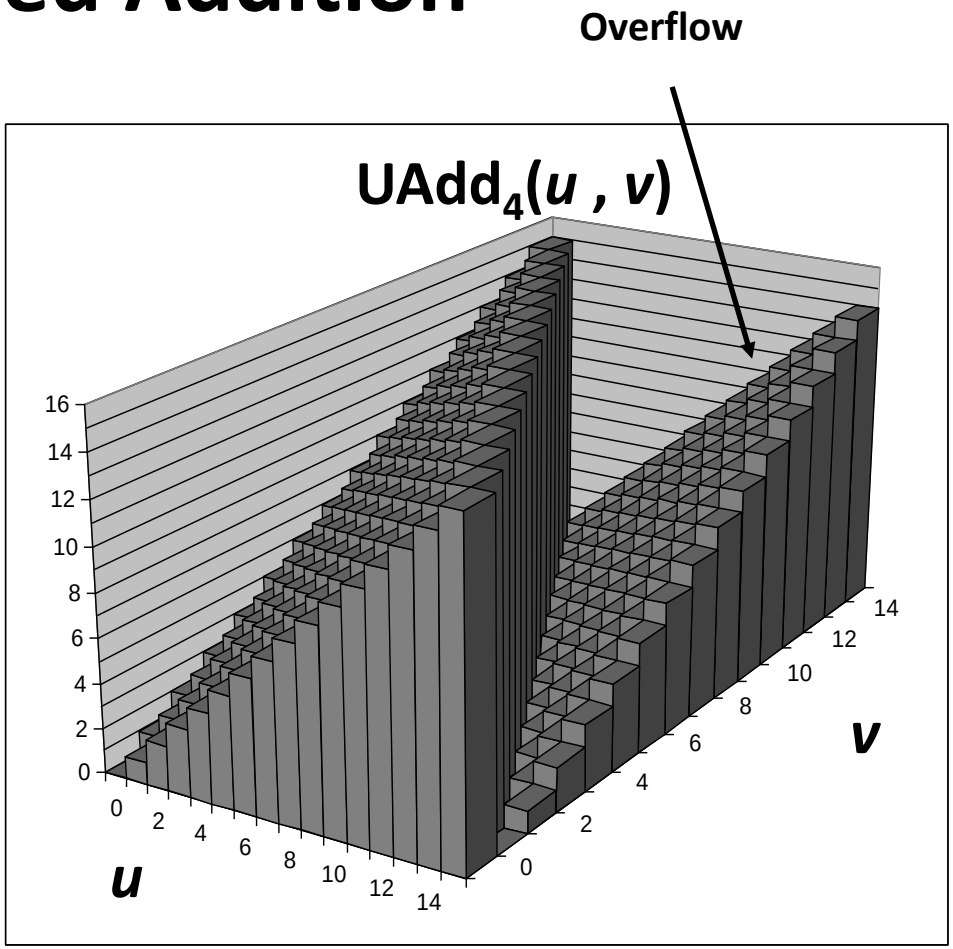
$\text{Add}_4(u , v)$



Visualizing Unsigned Addition

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





When executing C programs, overflows are not signaled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute *s* .=*x* +*y*, and we wish to determine whether *s* equals *x* + *y*. We claim that overflow has occurred if and only if *s* < *x* (or equivalently, *s* < *y*)

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 $\mathrm{UComp}_w(u) = 2^w - u$

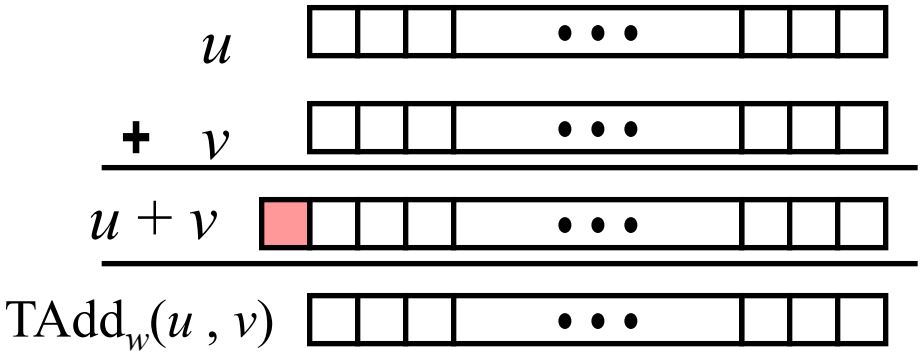
      $\mathrm{UAdd}_w(u, \mathrm{UComp}_w(u)) = 0$

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



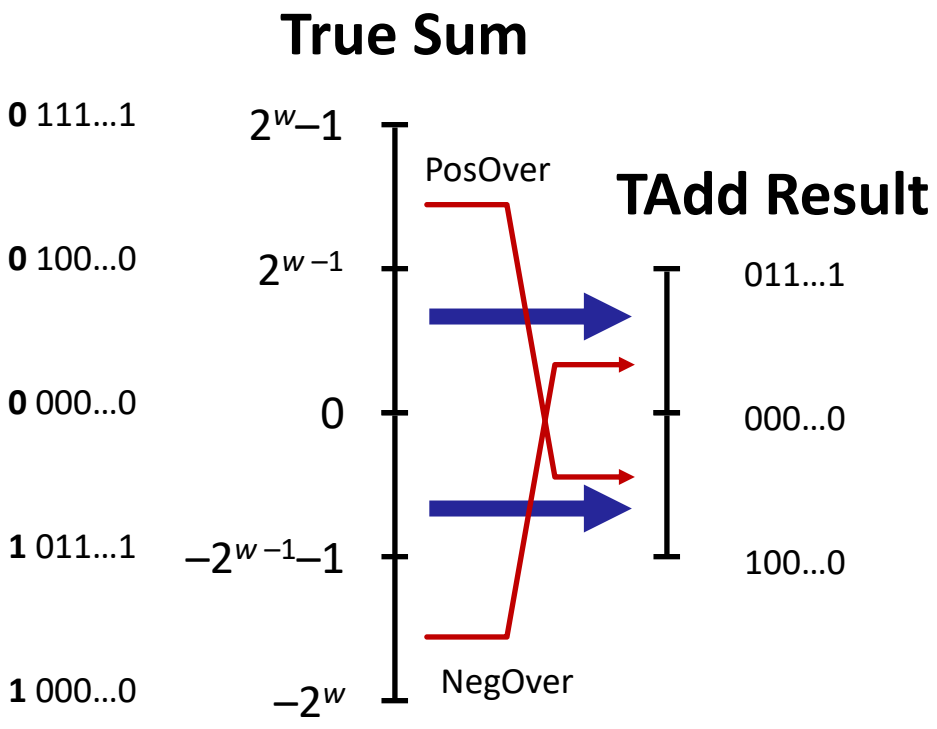
- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

```
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

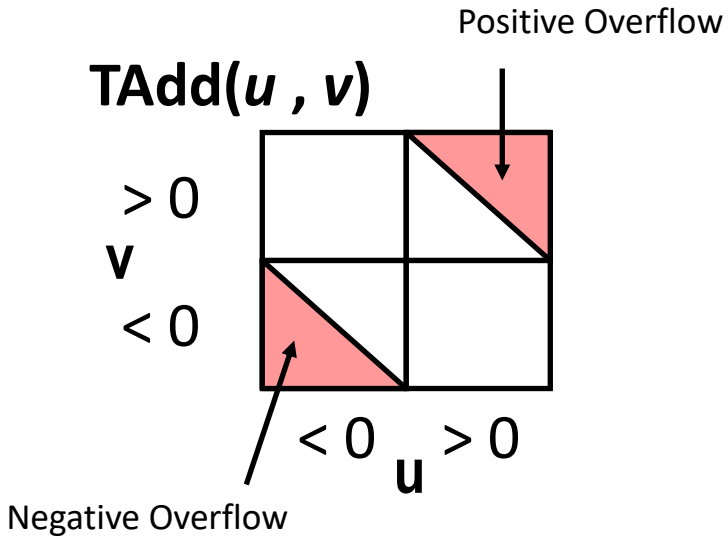
- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once

x	y	$x + y$	$x +_4^t y$	Case
−8	−5	−13	3	1
[1000]	[1011]	[10011]	[0011]	
−8	−8	−16	0	1
[1000]	[1000]	[10000]	[0000]	
−8	5	−3	−3	2
[1000]	[0101]	[11101]	[1101]	
2	5	7	7	3
[0010]	[0101]	[00111]	[0111]	
5	5	10	−6	4
[0101]	[0101]	[01010]	[1010]	

Figure 2.24 **Two's-complement addition examples.** The bit-level representation of the 4-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

Characterizing TAdd

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



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

Mathematical Properties of TAdd

- **Isomorphic Group to unsigneds with UAdd**
  - $\text{TAdd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$
    - 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

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

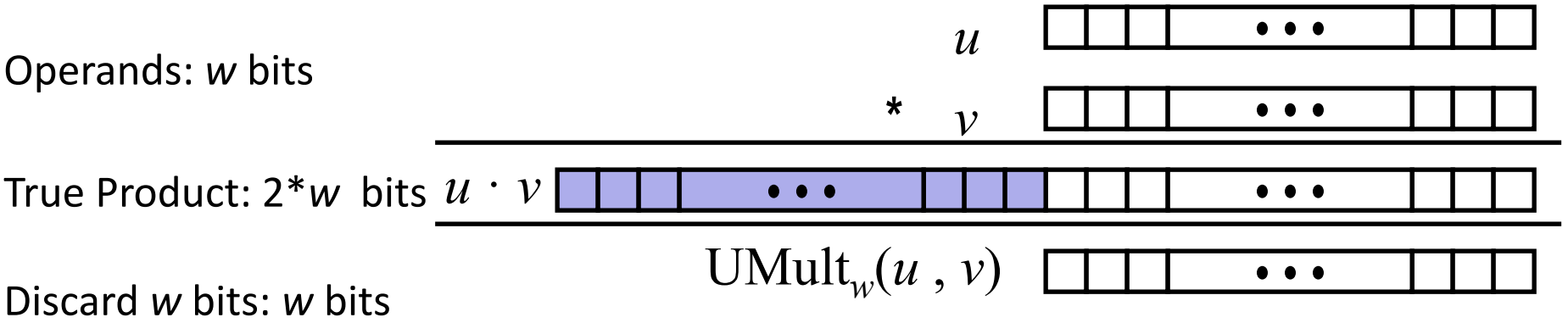
Practice

x	y	$x + y$	$x +_5^t y$	Case
[10100]	[10001]			
[11000]	[11000]			
[10111]	[01000]			
[00010]	[00101]			
[01100]	[00100]			

Multiplication

- **Computing Exact Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **Ranges**
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 0$ and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - **Up to 2*w* bits**
- **Maintaining Exact Results**
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

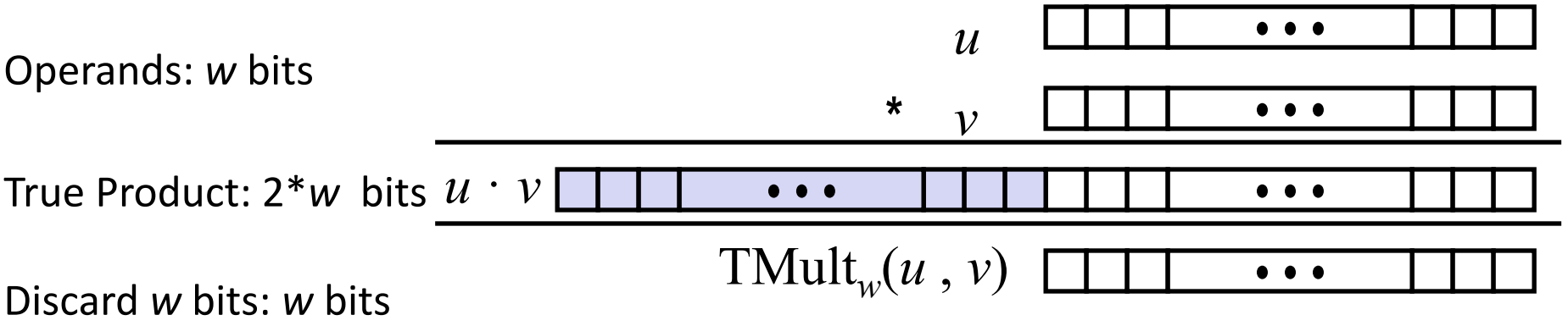
Unsigned Multiplication in C



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

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

Signed Multiplication in C



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

Truncating a two's-complement number to *w* bits is equivalent to first computing its value modulo 2_w and then converting from unsigned to two's complement.

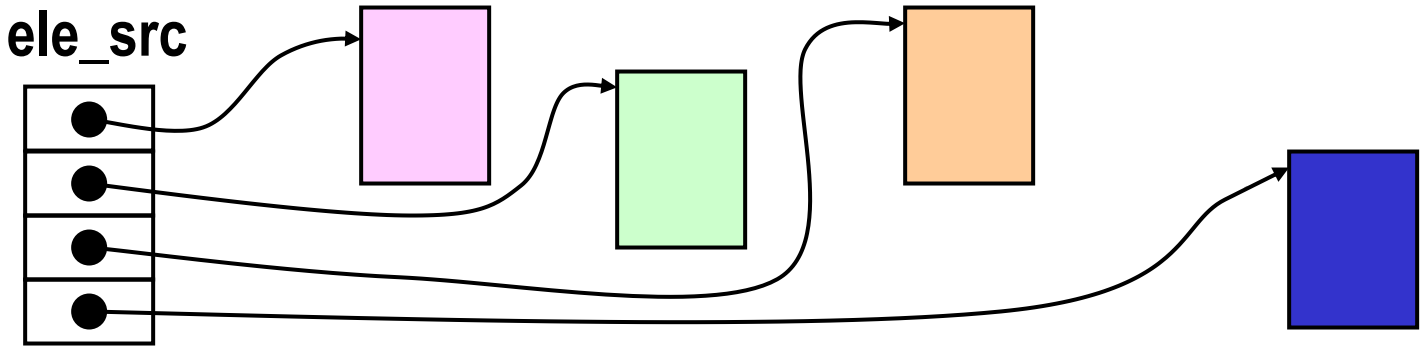
Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's comp.	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's comp.	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's comp.	3	[011]	3	[011]	9	[001001]	1	[001]

Figure 2.26 **Three-bit unsigned and two's-complement multiplication examples.** Although the bit-level representations of the full products may differ, those of the truncated products are identical.

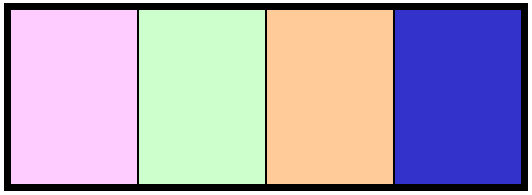
Code Security Example #2

- **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;
}
```

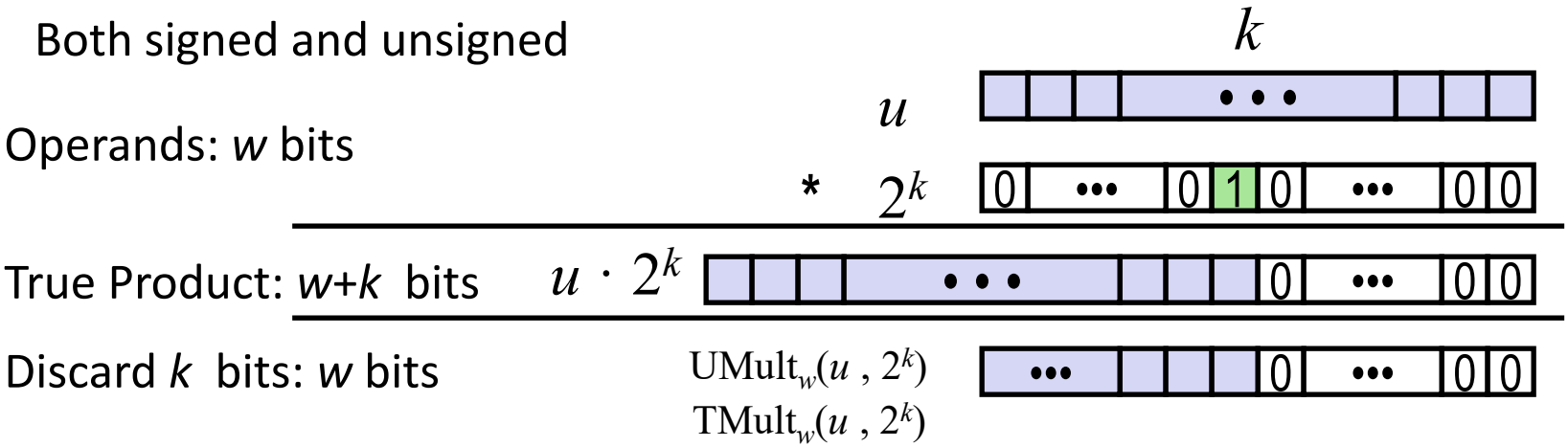
XDR Vulnerability

malloc(ele_cnt * ele_size)

- **What if:**
  - `ele_cnt` = $2^{20} + 1$
  - `ele_size` = 4096 = 2^{12}
  - Allocation = ??
  - Allocation needed: 4,294,971,392
  - Actual allocation: 4096
- **How can I make this function secure?**

Power-of-2 Multiply with Shift

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



- **Examples**
  - `u << 3 == u * 8`
  - `u << 5 - u << 3== u * 24 (24 = 2^5 - 2^3)`
  - 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
```

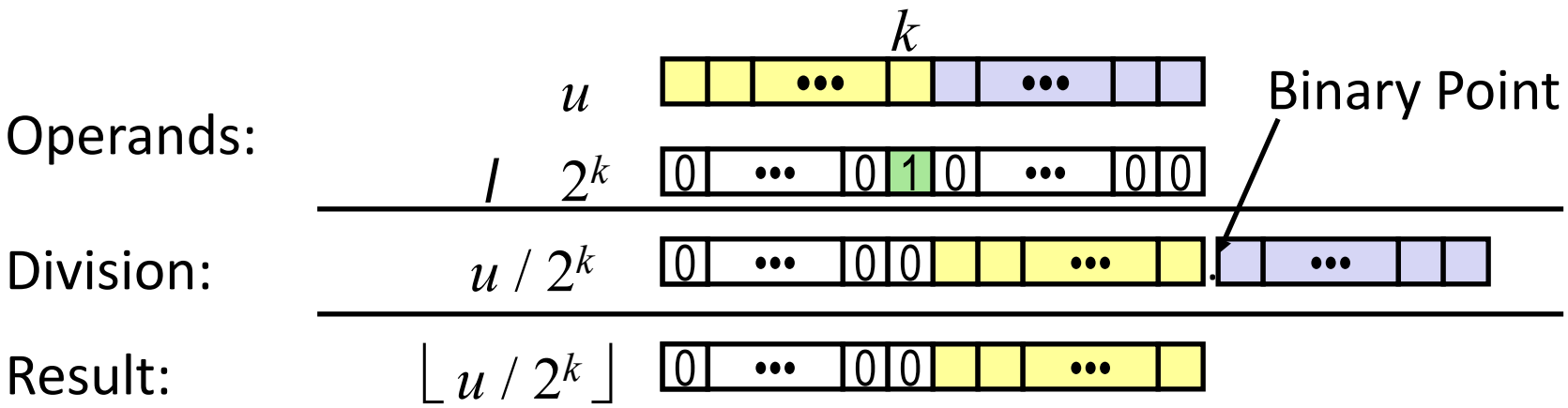
Explanation

```
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



k	>> k (Binary)	Decimal	$12340/2^k$
0	0011000000110100	12340	12340.0
1	0001100000011010	6170	6170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

Figure 2.27 **Dividing unsigned numbers by powers of 2.** The examples illustrate how performing a logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

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

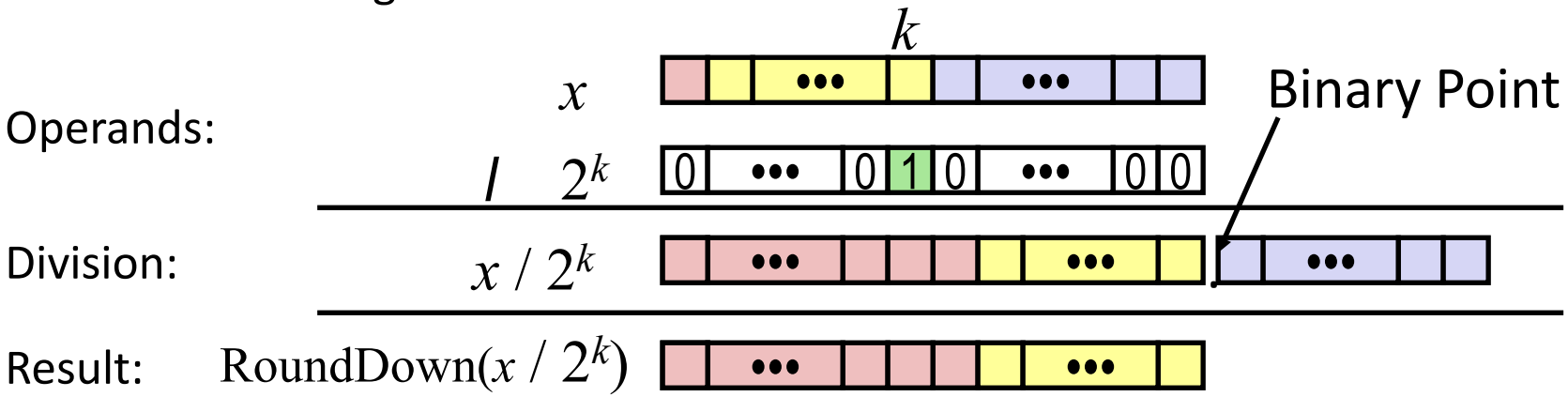
Explanation

```
  # 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`**



k	>> k (Binary)	Decimal	$-12340/2^k$
0	1100111111001100	−12340	−12340.0
1	1110011111100110	−6170	−6170.0
4	1111110011111100	−772	−771.25
8	1111111111001111	−49	−48.203125

Figure 2.28 **Applying arithmetic right shift.** The examples illustrate that arithmetic right shift is similar to division by a power of 2, except that it rounds down rather than toward zero.

Correct division

- **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

This technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. As examples, when $x = -30$ and $y = 4$, we have $x + y - 1 = -27$, and $\lceil -30/4 \rceil = -7 = \lfloor -27/4 \rfloor$.

k	Bias	−12,340 + Bias (Binary)	>> k (Binary)	Decimal	$-12340/2^k$
0	0	1100111111001100	1100111111001100	−12340	−12340.0
1	1	1100111111001101	1110011111100110	−6170	−6170.0
4	15	1100111111011011	1111110011111101	−771	−771.25
8	255	1101000011001011	1111111111010000	−48	−48.203125

Figure 2.29 **Dividing two's-complement numbers by powers of 2.** By adding a bias before the right shift, the result is rounded toward zero.

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

- **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**

Integer C Puzzles

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

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