Bits, Bytes, and Integers (Cont.)

B&O Readings: 2.1-2.3

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Note: these slides were originally created by Markus Püschel at Carnegie Mellon University

Code Puzzle

- **What's the bug in this code?**

```
float sum_elements(float a[], unsigned length) {
    int i;
    float result = 0;

    for (i=0; I <= length-1; i++)
        result += a[i];
    return result;
}
```

Fix: use `i < length` instead

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
- Representations in memory, pointers, strings

Extension

- **Converting from smaller to larger integer data type**
- **C automatically performs extension**

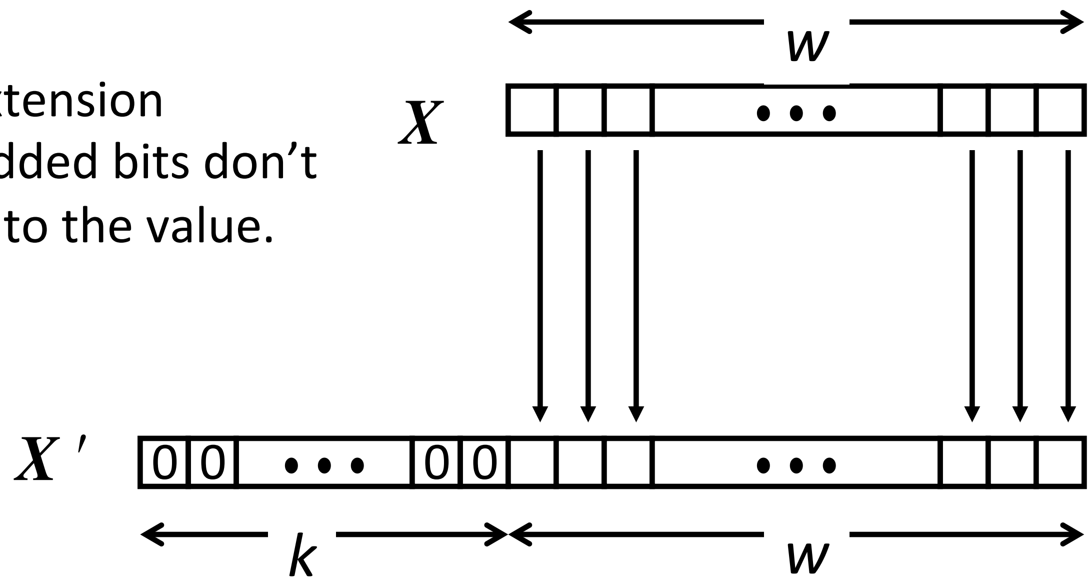
```
unsigned short sx = 15213;
unsigned int x = (int) sx; /* use zero extension */

short sy = -15213;
int y = (int) sy; /* use sign extension */
```

- **Task:**
  - 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)

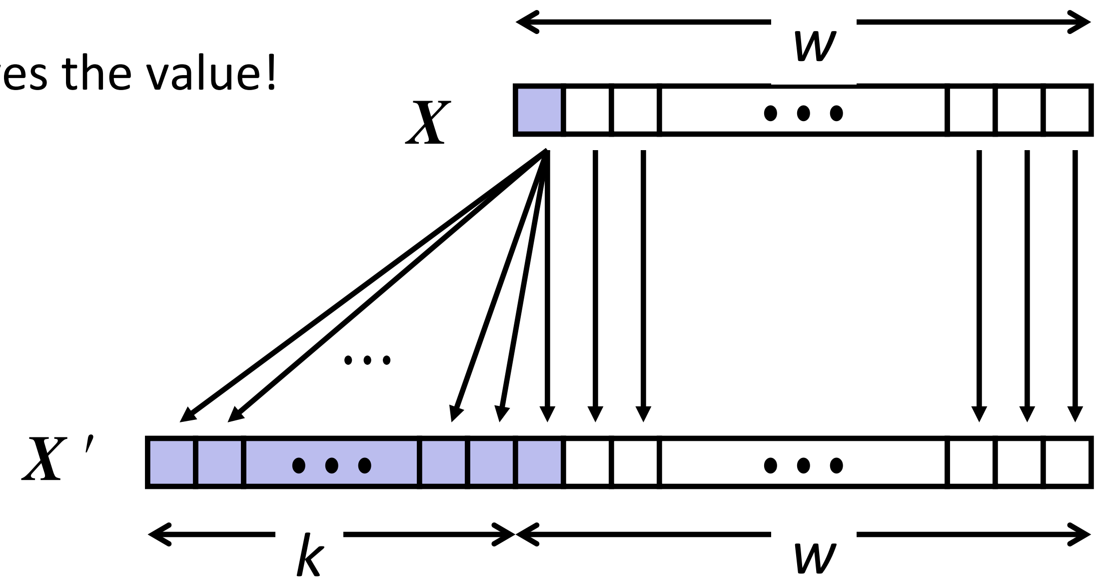
Zero Extension for Unsigned type

- **Task:**
  - 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.



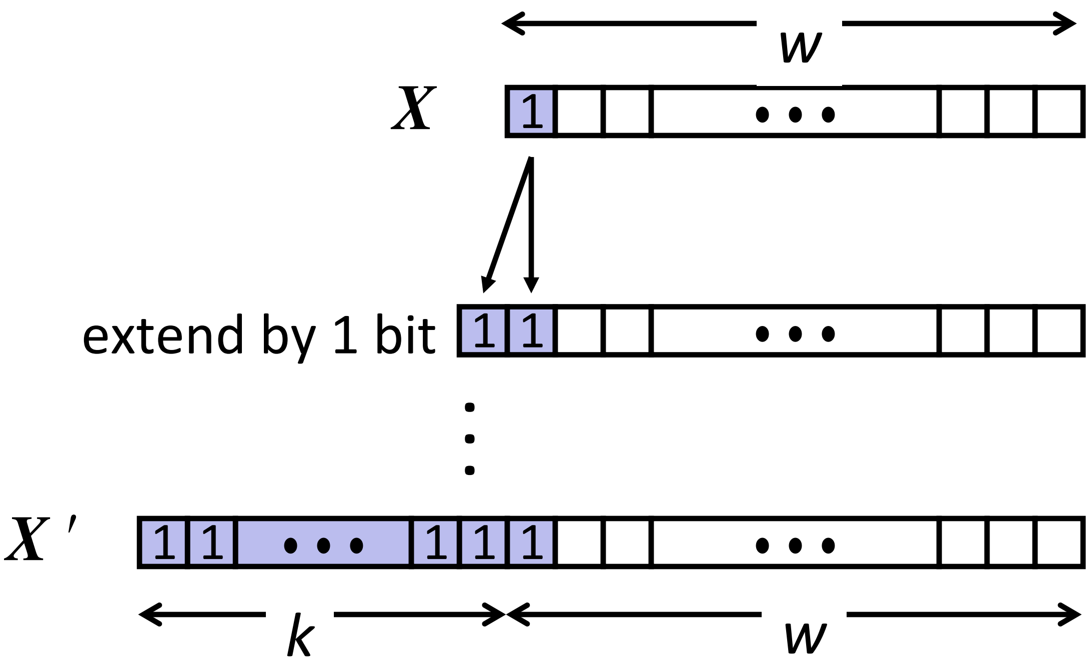
Sign Extension

- **Task:**
  - 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:**



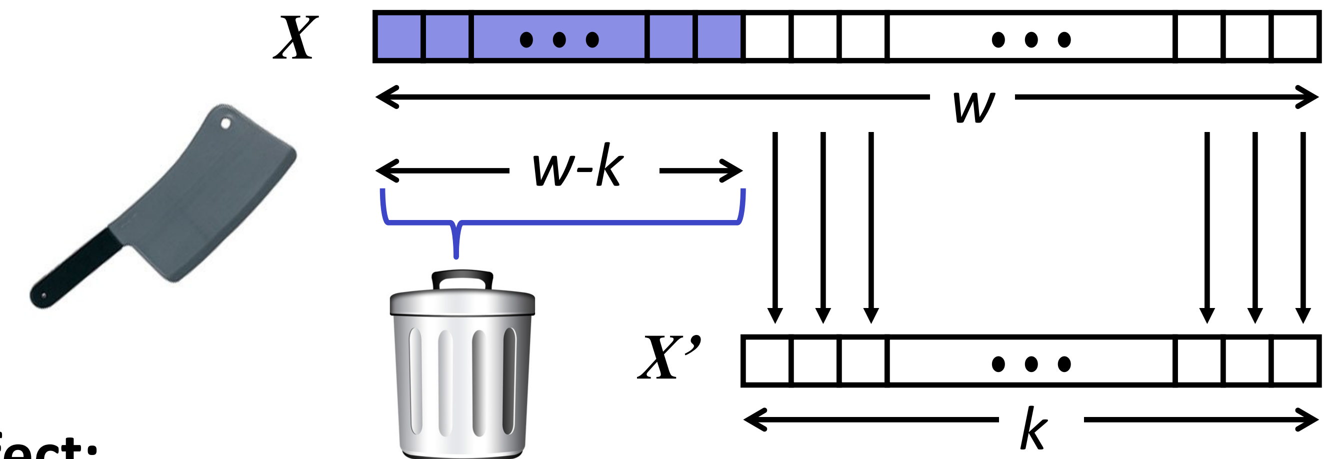
MSB contributed weight -2^{w-1}

The 2nd 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)
  - Mathematical mod on *X* (compute a positive *r* such that $X = q \cdot m + r$)
  - Reinterpret the bits (for signed data: add -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
  - Unsigned: mod operation (keep the last k bits)
  - Signed: mathematical mod, and then reinterpret the bits as signed
  - For small numbers yields expected behavior; for large number, can overflow.

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**
- Representations in memory, pointers, strings
- Summary

Binary Addition

- **4-bit unsigned integer addition: 4 + 5**

```
carry bit:   1
             0 1 0 0
         +   0 1 0 1
      -----------------
             1 0 0 1
```

- **4-bit unsigned integer addition: 12 + 5**

```
carry bit: 1 1
             1 1 0 0
         +   0 1 0 1
      -----------------
           1 0 0 0 1
```

→ **Overflow!**
Not enough bits to store the actual result.

Unsigned Addition

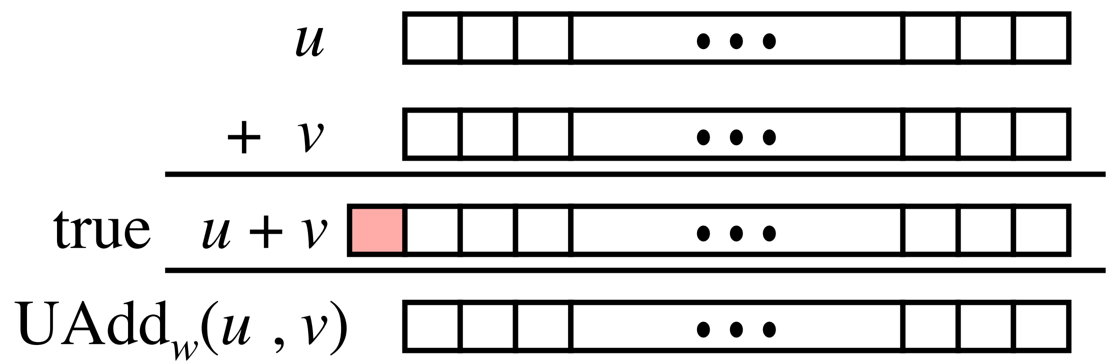
$$0 \leq u, v \leq 2^w - 1$$

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

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

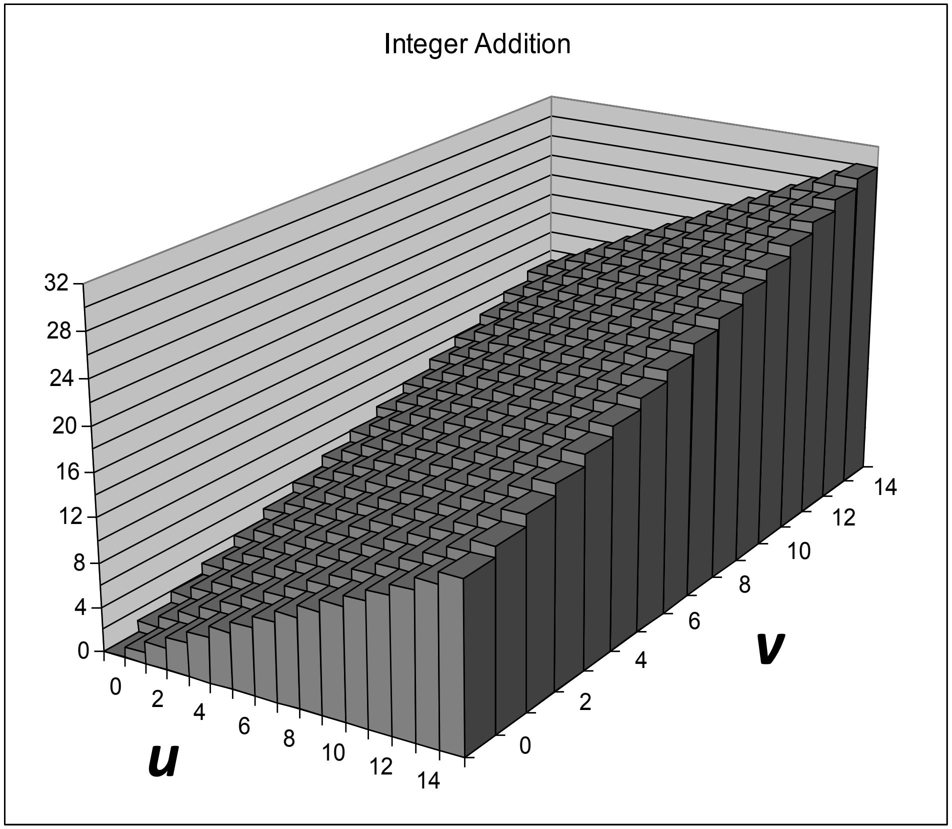


- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**
  - $UAdd_w(u, v) = u + v \mod 2^w$
  - **overflow:** the full result cannot fit within the size limit of the data type

Visualizing (Mathematical) Integer Addition

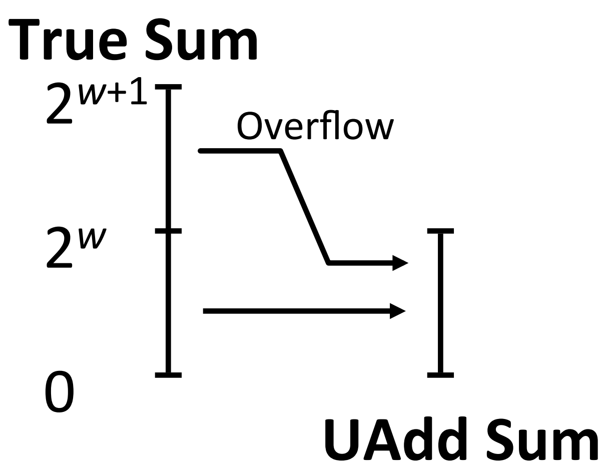
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

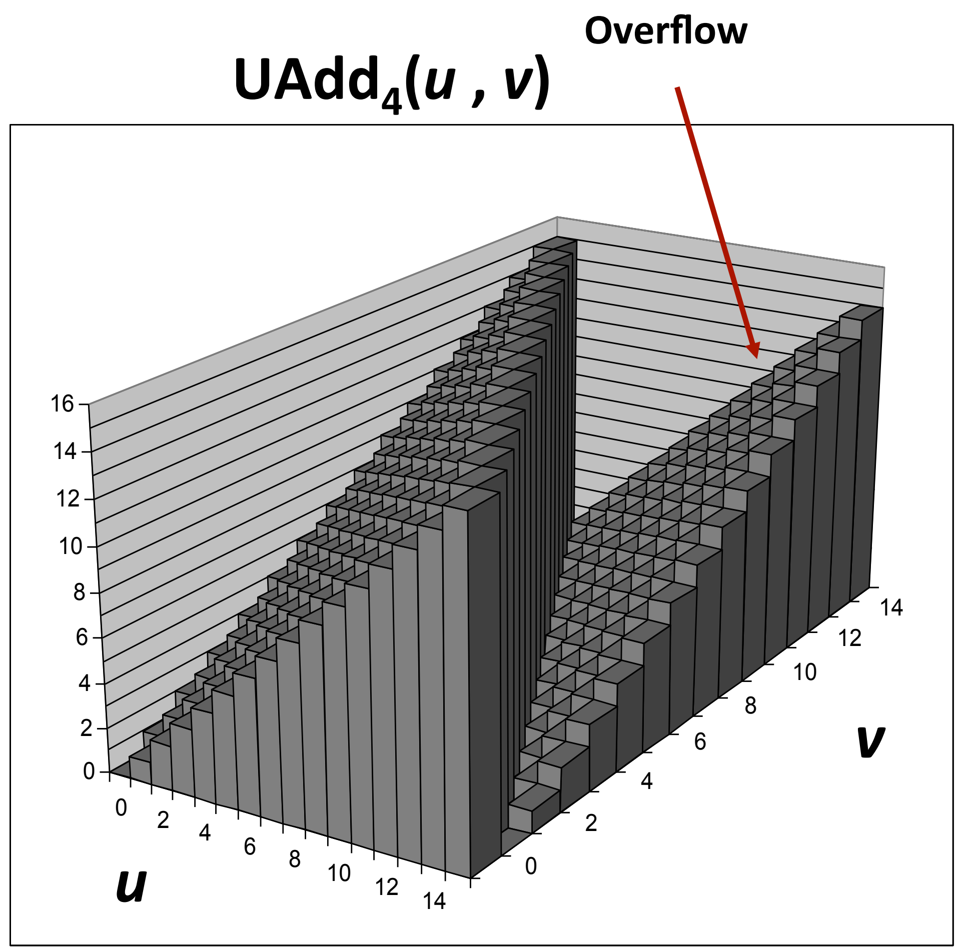
$\text{Add}_4(u, v)$



Visualizing Unsigned Addition

- **When overflow:**
  - If true sum ≥ 2^w
  - wraps around at most once
  - UAdd sum = true sum – 2^w





Q: How to detect overflow in UAdd?

Two's Complement Addition

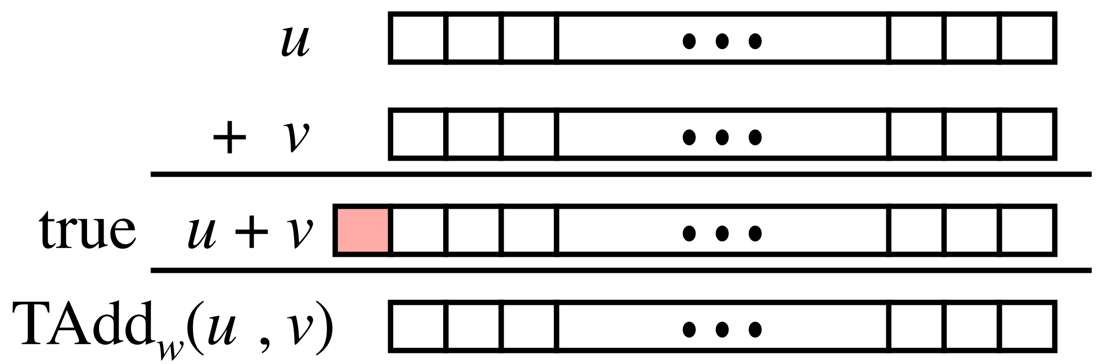
$$-2^{w-1} \le u, v \le 2^{w-1}-1$$

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

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

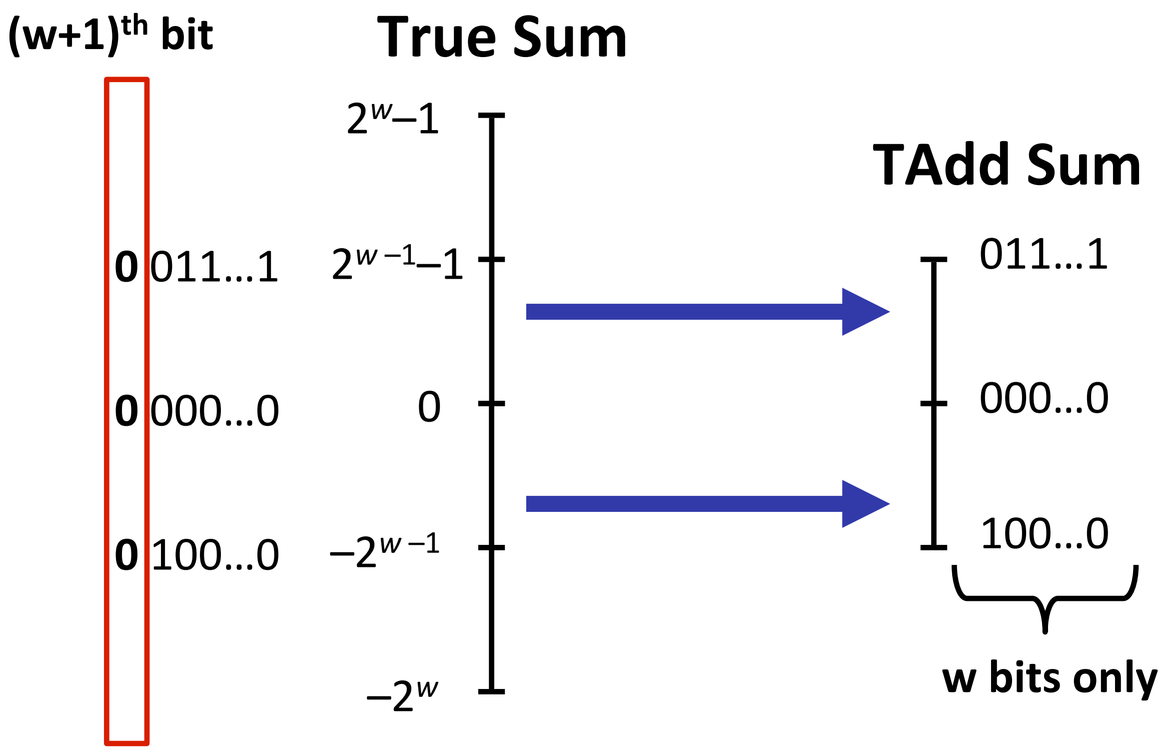


- **TAdd and UAdd have Identical Bit-Level Behavior**

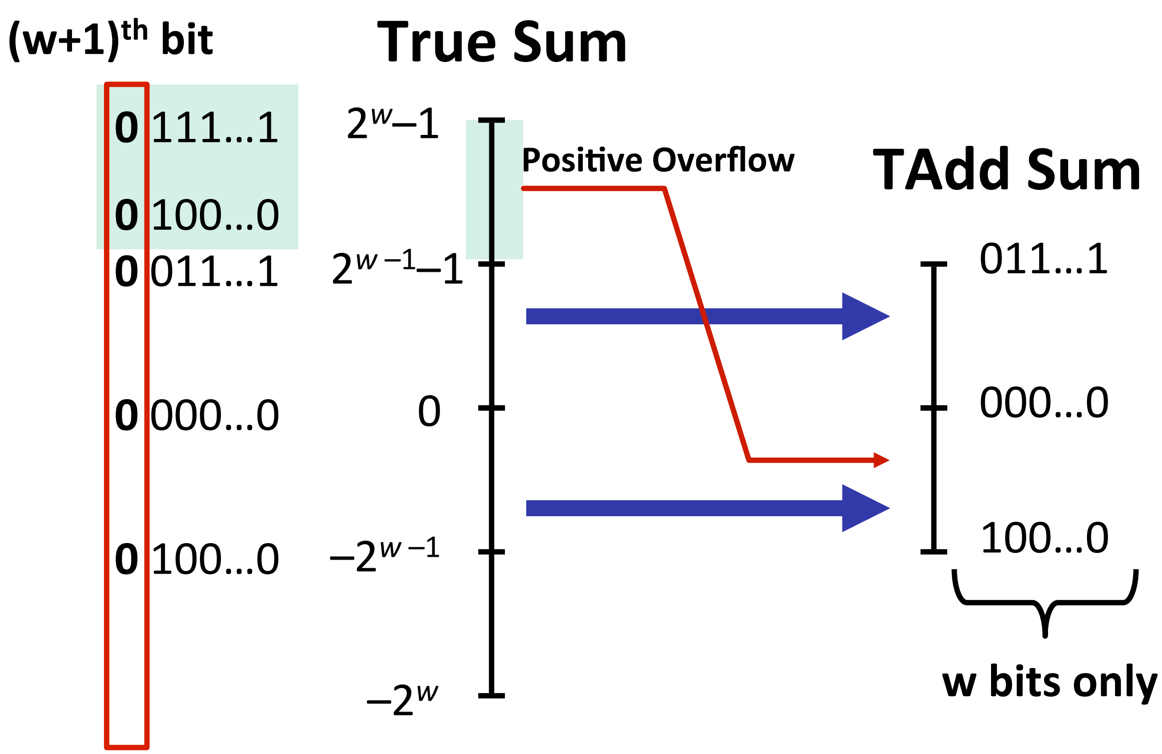
```
int s, t, u, v;
... /* initialize their values */
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
assert(s == t); /* always true! */
```

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

TAdd Overflow

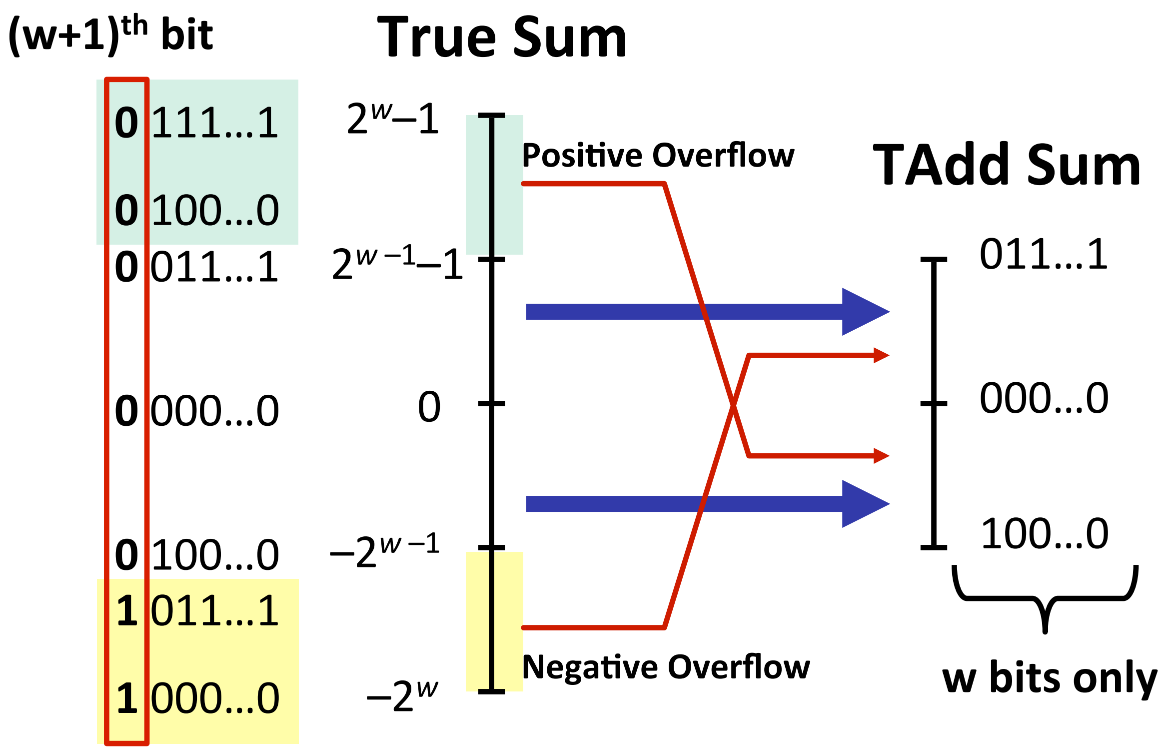


TAdd Overflow



- **Positive overflow:**
  - Adding two positive values, where $u + v > 2^{w-1}-1$
  - w^{th} bit contributes to true sum weight of 2^{w-1} but to TAdd sum -2^{w-1}
  - TAdd sum = true sum – 2^{W}

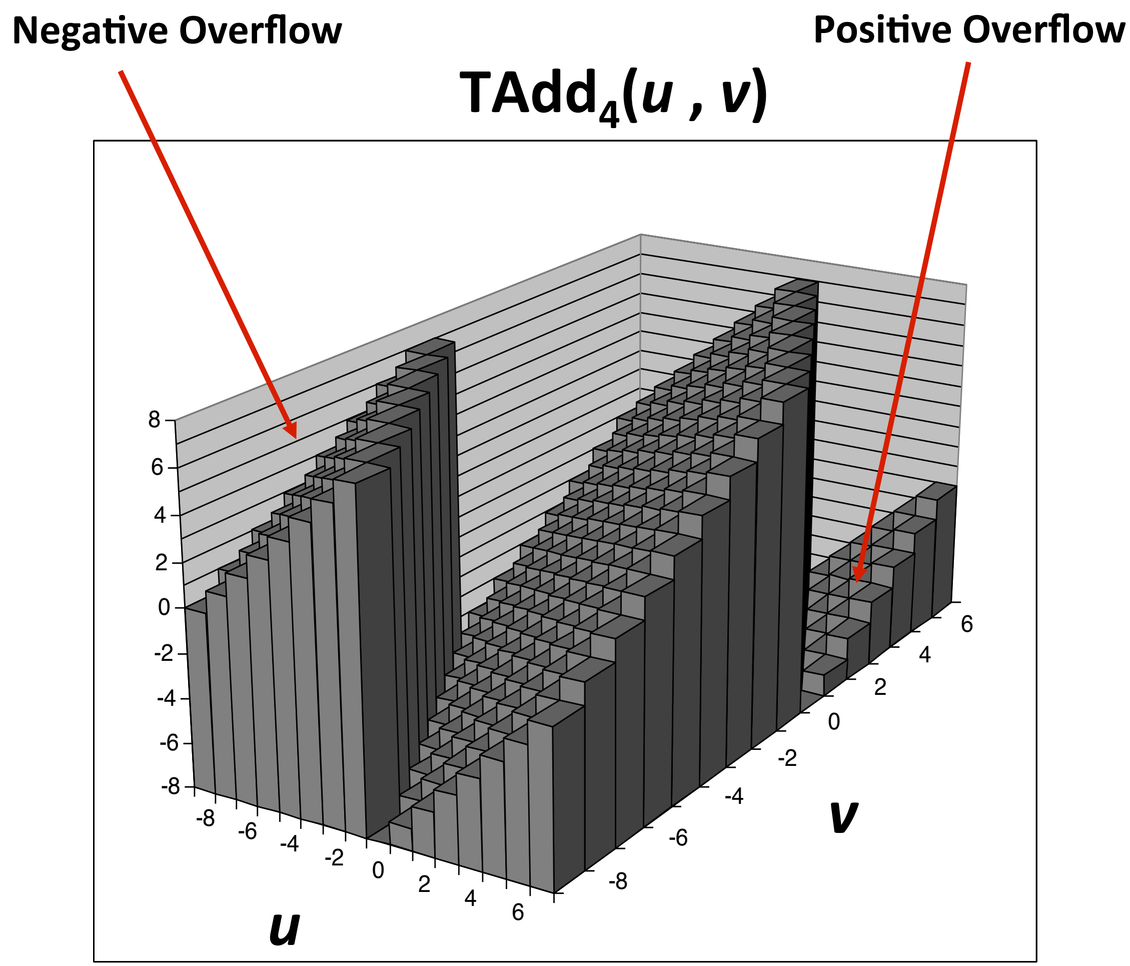
TAdd Overflow



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

Visualizing 2's Complement Addition

- **Positive overflow:**
  - If sum ≥ 2^{w-1}, value becomes negative
- **Negative overflow:**
  - If sum < -2^{w-1}, value becomes positive
- In either case, the value wraps around at most once!
  - (computed sum = true sum + / - 2^w)



Q: How to detect overflow in TAdd?

Negation: Complement & Increment

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

  `-x = ~x + 1`

- **Example:**

x = 15213

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`
`~x`	-15214	`C4 92`	`11000100 10010010`
`~x+1`	-15213	`C4 93`	`11000100 10010011`
`y`	-15213	`C4 93`	`11000100 10010011`

x = 0

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

Negation: Complement & Increment

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

  `-x = ~x + 1`

- **Why is this the case?**

Negation: Complement & Increment

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

  `-x = ~x + 1`

- **Why is this the case?**
- **Observation:**
  - **`~x + x = 0xFF…F = -1; (0xFF…F)`**

	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

  - **`~x = -1 – x;`**
  - **`-x = 0 – x`**
    `= (-1 + 1) – x`
    `= (-1 – x) + 1`
    `= ~x + 1;`

Multiplication

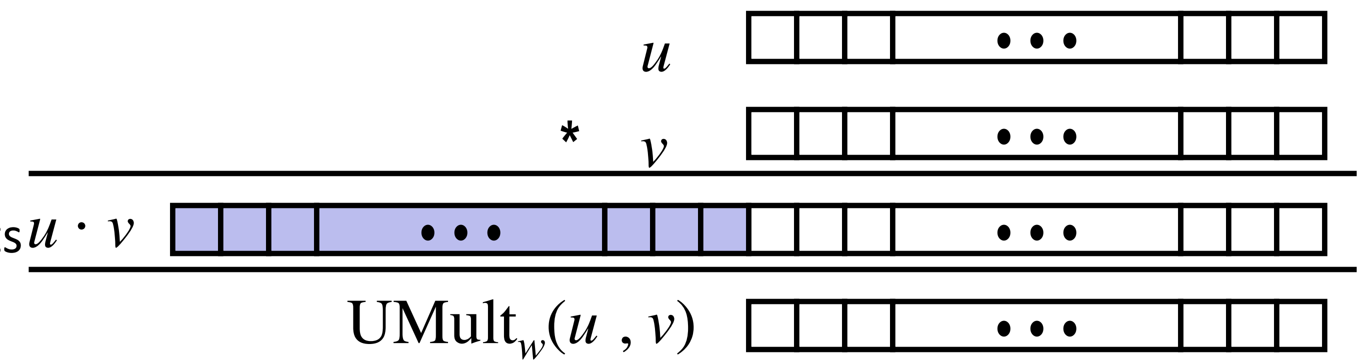
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* bits
    - Result range: $0 \leq x * y \leq (2^{w} - 1)^{2} = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_{w})^{2}$
    - Result range: $x * y \leq (-2^{w-1})^{2} = 2^{2w-2}$
- **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" arithmetic packages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

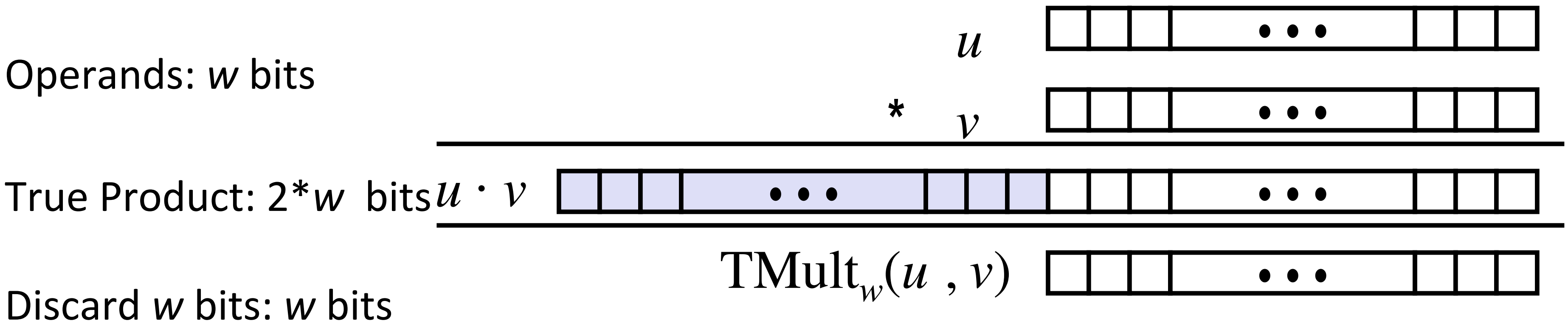
Discard *w* bits: *w* bits



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

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

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

Example of UMult_3 and TMult_3

Mode	*X*	*Y*	*X · Y*	truncated *X · Y*	
UMult	5 [101]	3 [011]	15 [001111]	7	[111]
TMult	-3 [101]	3 [011]	-9 [110111]	-1	[111]
UMult	4 [100]	7 [111]	28 [011100]	4	[100]
TMult	-4 [100]	-1 [111]	4 [000100]	-4	[100]

Although the bit-level representations of the full product may differ, those of the truncated products are identical!

The value difference between sign and unsigned is 0 mod 2^w.

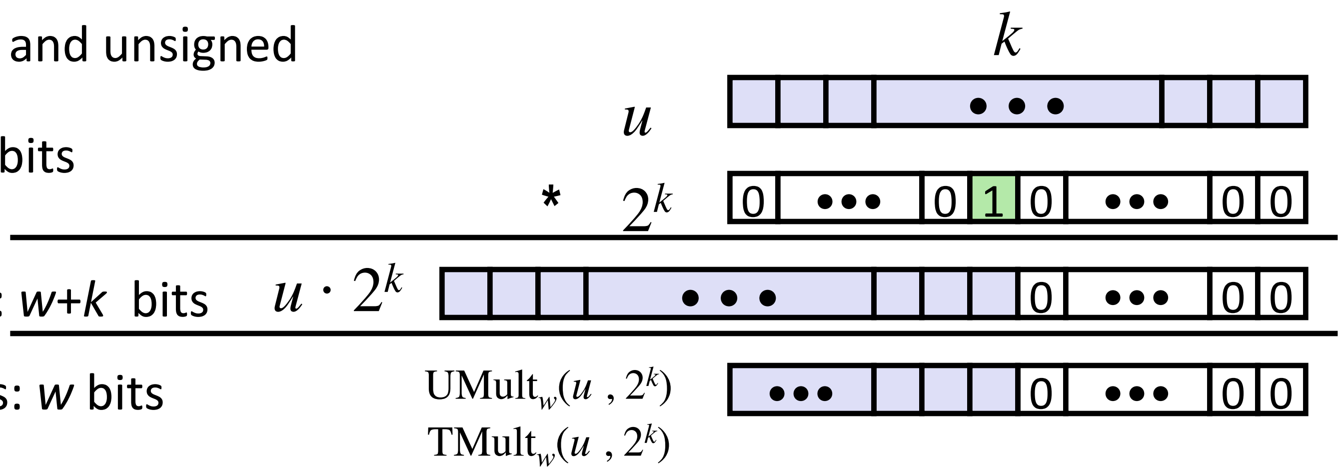
Power-of-2 Multiply with Shift

- **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`
  - 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 · 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)

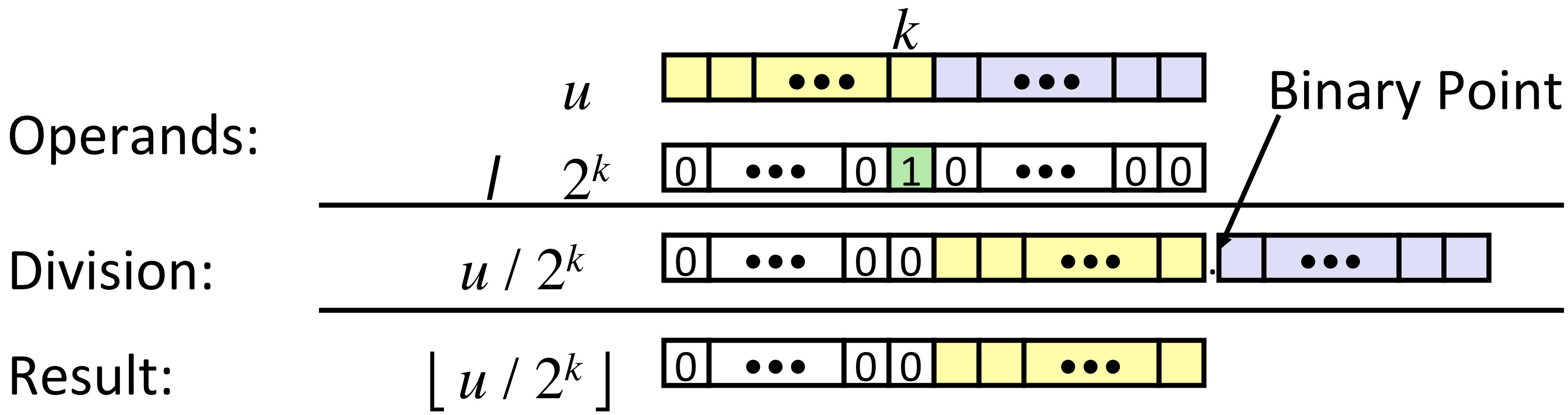
$$x \cdot 6 = x \cdot (2^2 + 2^1)$$
$$= x << 2 + X << 1$$

Or, equivalently,

$$x \cdot 6 = x \cdot (2^3 - 2^1)$$
$$= x << 3 - X << 1$$

Unsigned Power-of-2 Divide with Shift

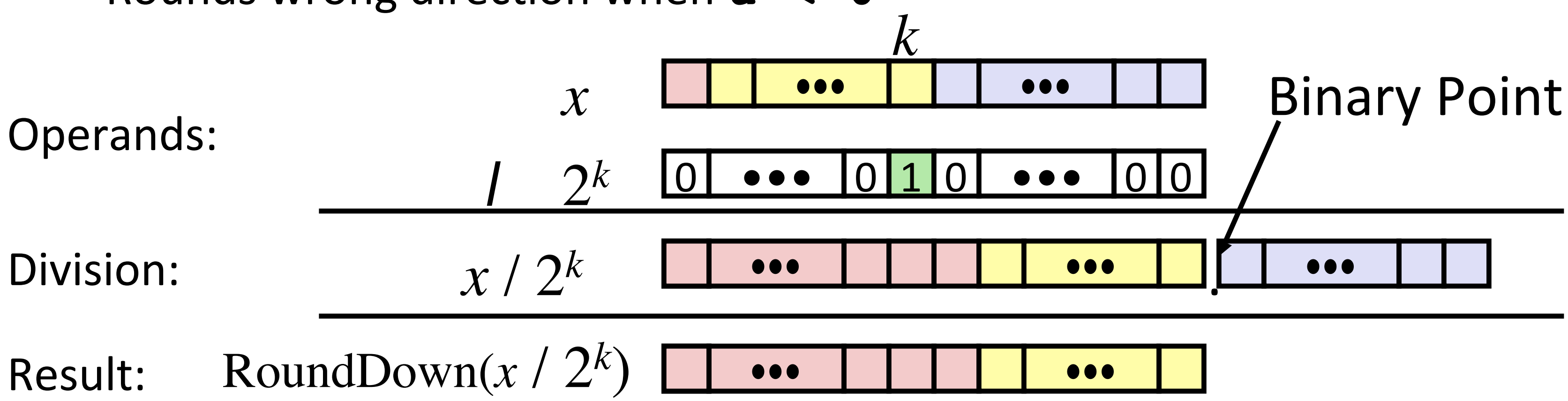
- **Quotient of Unsigned by Power of 2**
  - **`u >> k`** gives $\lfloor$ **`u / `** $2^k \rfloor$
  - Uses logical shift



	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`

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`



	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

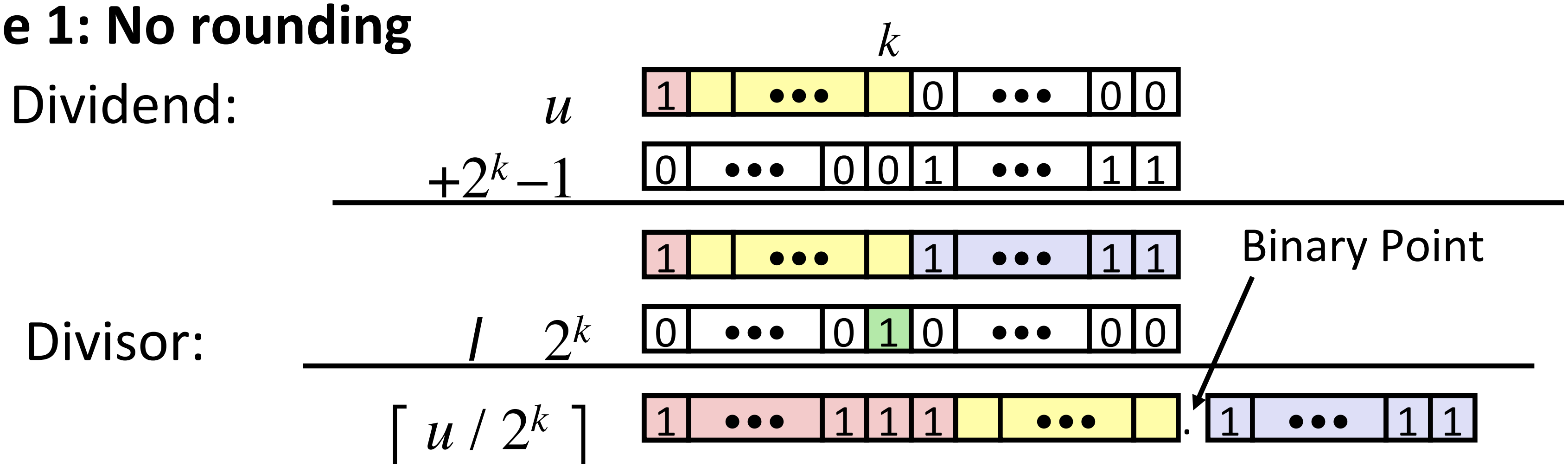
- **Quotient of Negative Number by Power of 2**
  - Want $\lceil$ **x / 2^k** $\rceil$ (Round Toward 0)
  - Compute as $\lfloor$ **(x+2^k-1) / 2^k** $\rfloor$
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0

- **What does adding the Bias do?**

Correct Power-of-2 Divide

- **Quotient of Negative Number by Power of 2**
  - Want ⌈ **`x / 2`**^{k} ⌉ (Round Toward 0)
  - Compute as ⌊ **`(x+2`**^{k}**`-1)/ 2`**^{k} ⌋
    - In C: **`(x + (1<<k)-1) >> k`**
    - Biases dividend toward 0

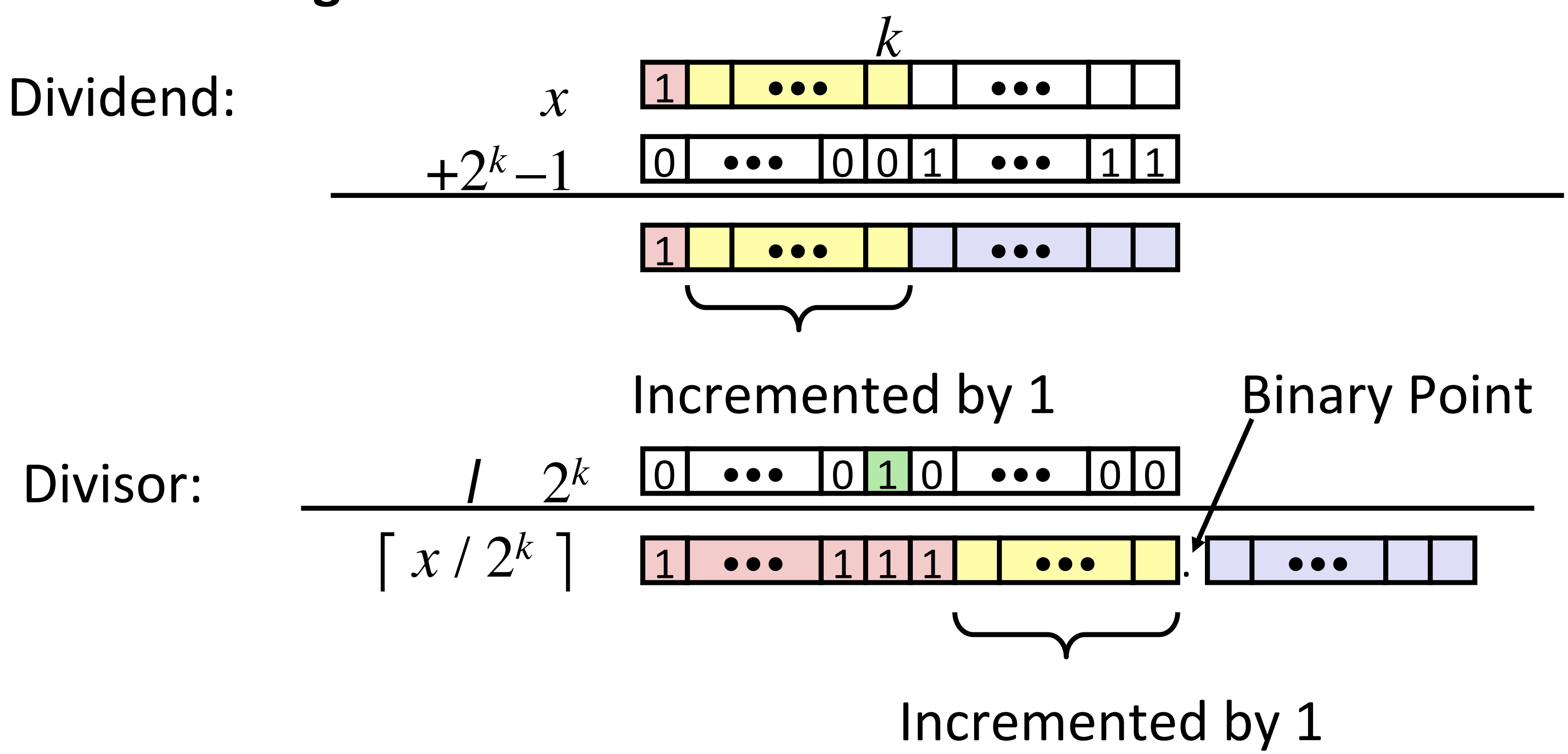
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

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**
- Representations in memory, pointers, strings

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: multiplication mod 2^w and reinterpret the bits as signed

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

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

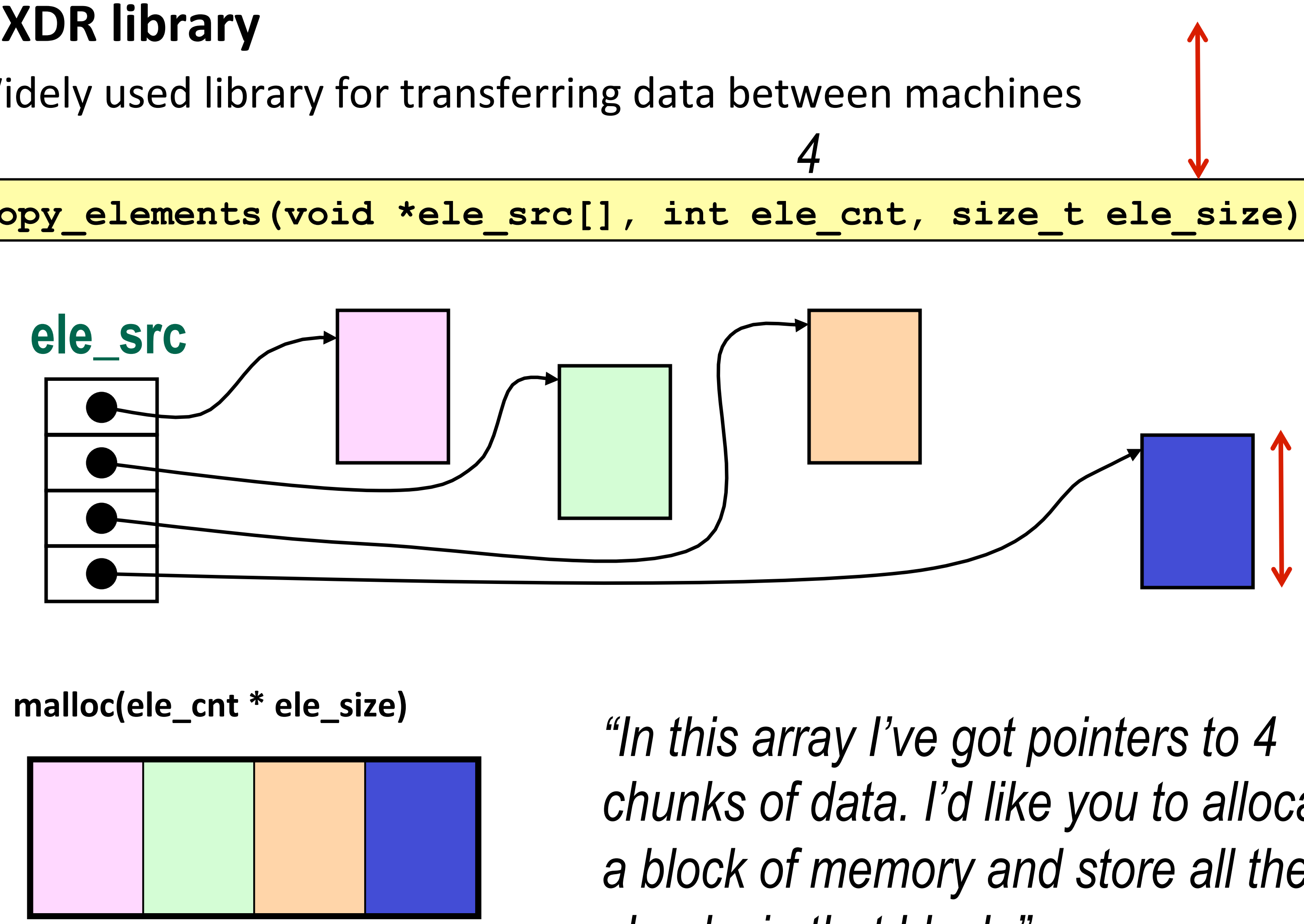
```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

- ***Do*** **Use When Performing Modular Arithmetic**
  - Multi-precision arithmetic
- ***Do*** **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

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);
```



"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

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

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

XDR Vulnerability

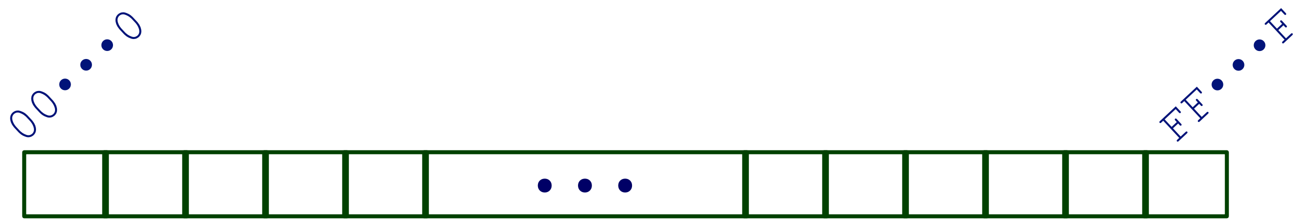
```
malloc(ele_cnt * ele_size)
```

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

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
- **Representations in memory, pointers, strings**

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

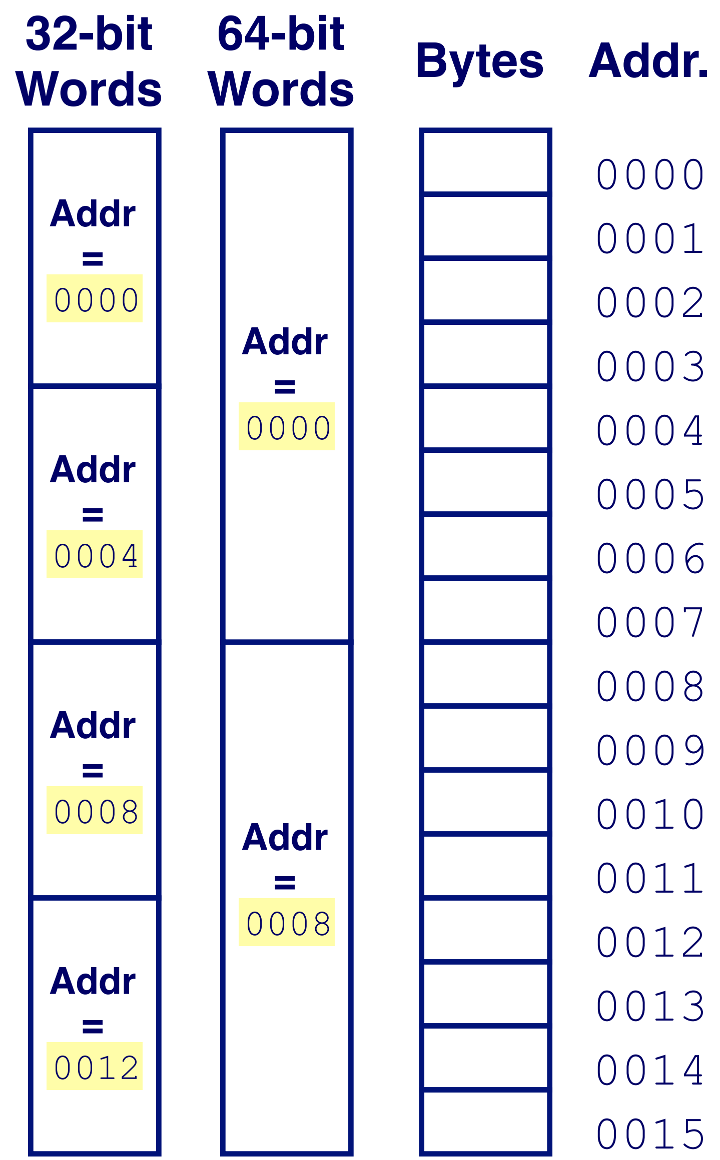
- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 PB (petabytes) of addressable memory
    - That's 18.4 X 10^{15}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Example Data Representations

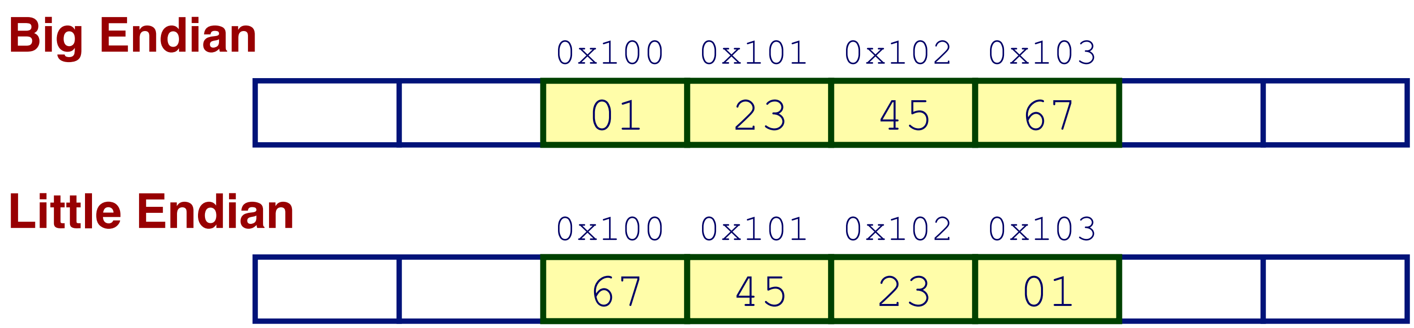
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	−	−	10/16
pointer	4	8	8

Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

Byte Ordering Example

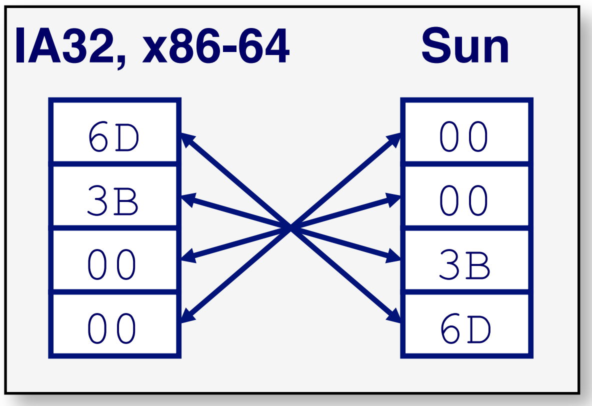
- **Example**
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100



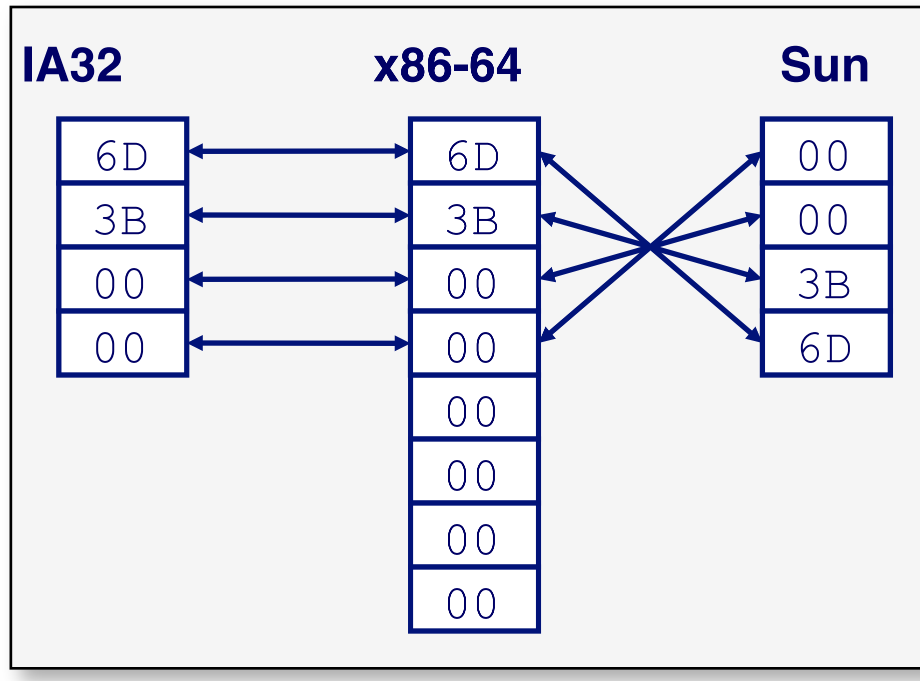
Representing Integers

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

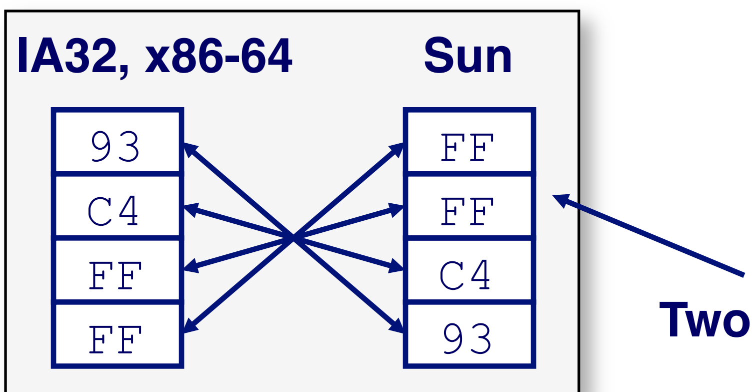
`int A = 15213;`



`long int C = 15213;`



`int B = -15213;`



Two's complement representation

Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf(”%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

`show_bytes` Execution Example

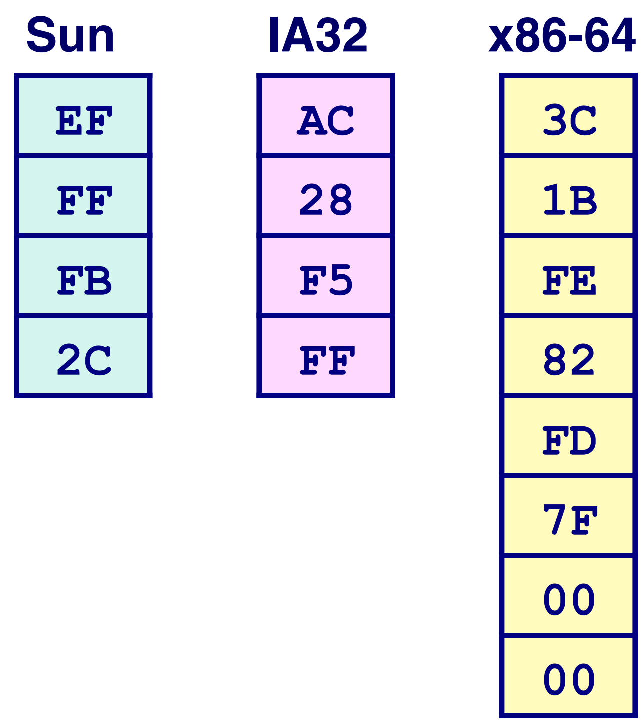
```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

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



Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character "0" has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue