

# CS207: File Processing Term I/2018-19

*Lecture 14:*

A Pragmatic View of Computer Systems

## **Partying with Bits**

*Sunsern Cheamanunkul*



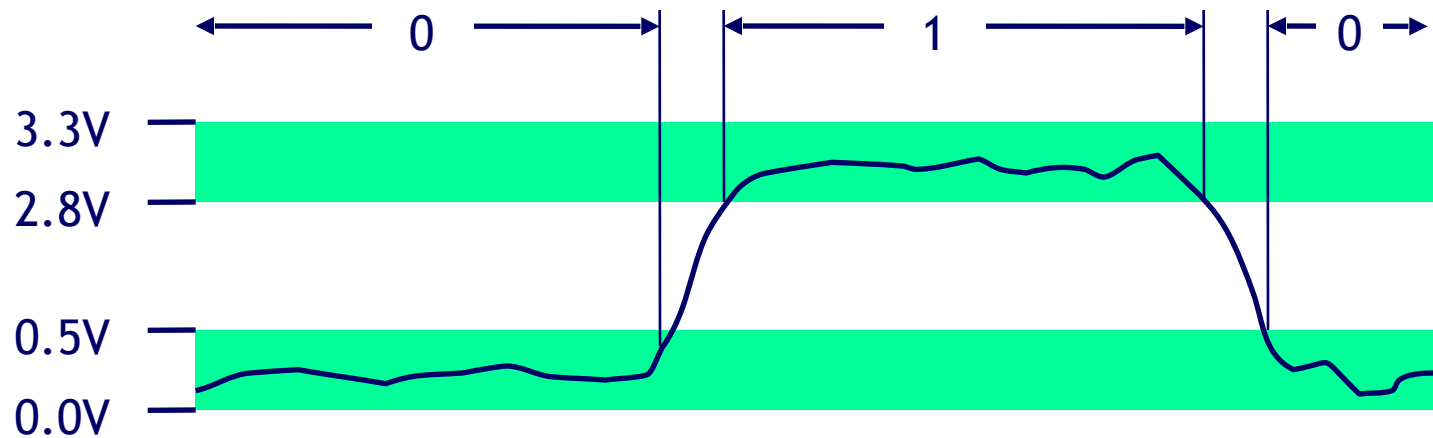
Mahidol University  
International College



# Overview

- **Representing information as bits**
- Bit-level manipulations
- Integer representation
- Addition, negation, multiplication, shifting

# Binary Representations

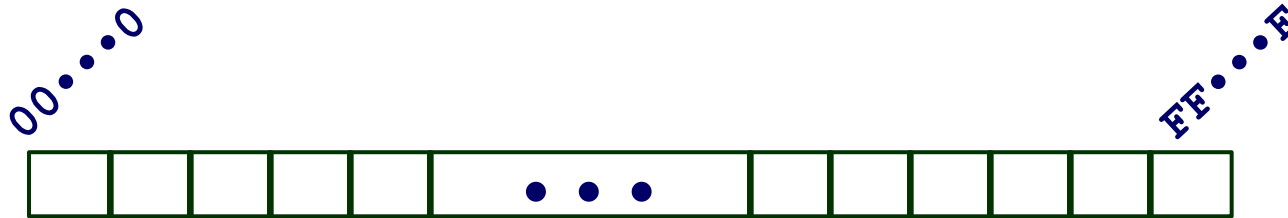


# Encoding Byte Values

## ■ 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`

# Byte-Oriented Memory Organization



## ■ Programs Refer to Virtual Addresses

- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

## ■ Compiler + Run-Time System Control Allocation

- Where different program objects should be stored
- All allocation within single virtual address space

# Machine Words

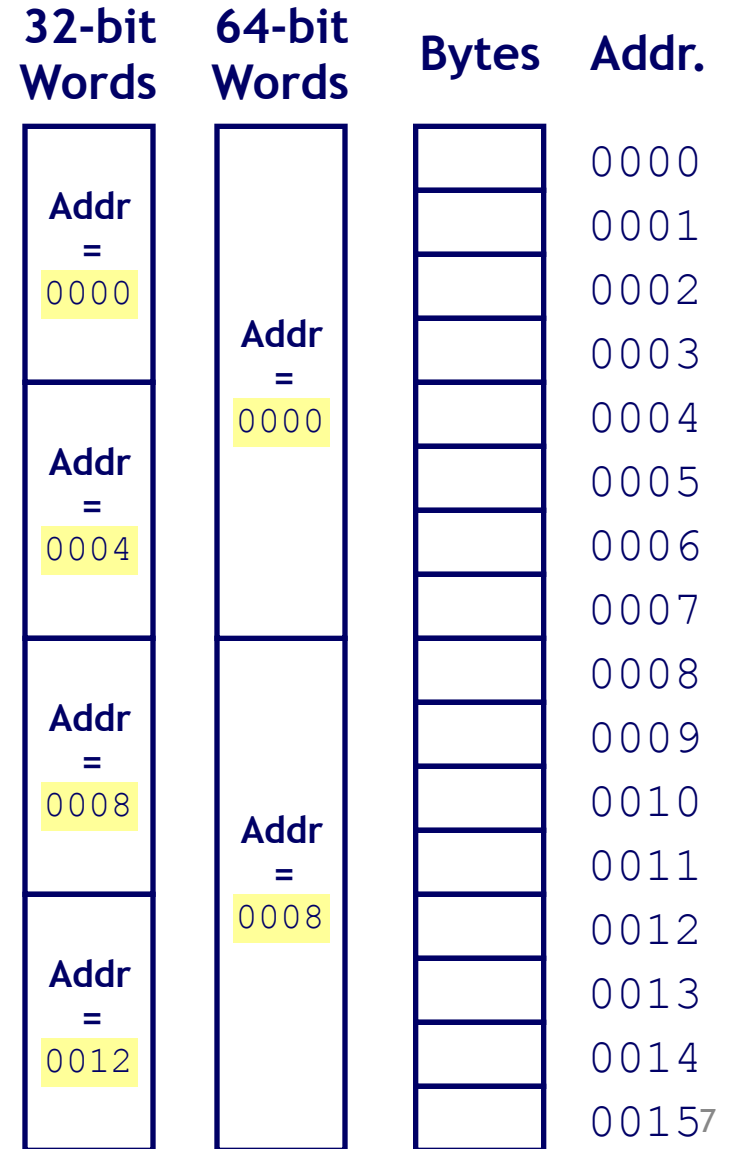
## ■ Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space  $\approx 1.8 \times 10^{19}$  bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines 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)



# Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8



# Byte Ordering

■ How should bytes within a multi-byte word be ordered in memory?

■ Conventions

- Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address

# Byte Ordering Example

## ■ Big Endian

- Least significant byte has highest address

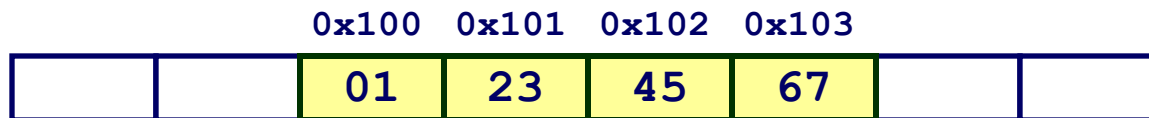
## ■ Little Endian

- Least significant byte has lowest address

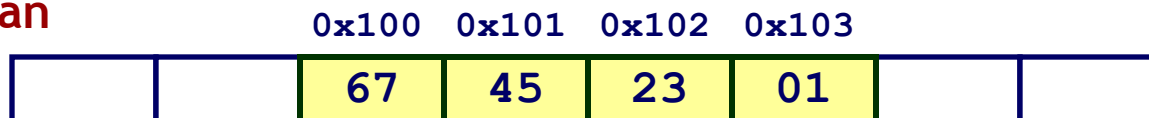
## ■ Example

- Variable x has 4-byte representation 0x01234567
- Address given by &x is 0x100

### Big Endian



### Little Endian



# Examining Data Representations

## ■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char \* creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
    int 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):

```
int a = 15213;  
0x11ffffffcb8 0x6d  
0x11ffffffcb9 0x3b  
0x11ffffffcba 0x00  
0x11ffffffcbb 0x00
```

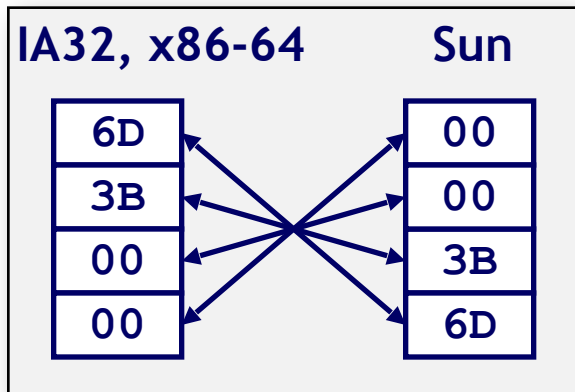
# Representing Integers

Decimal: 15213

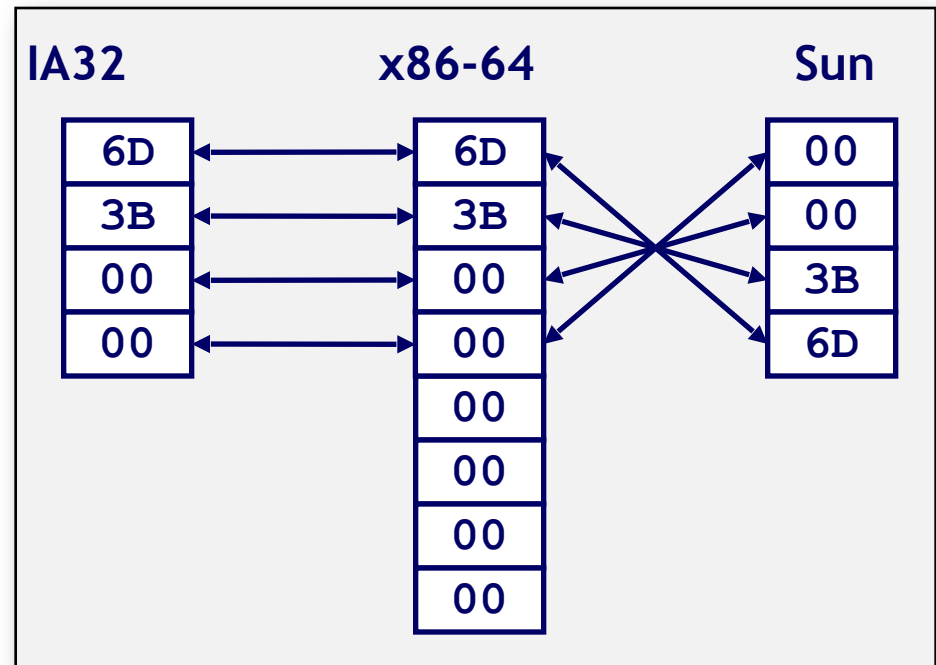
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

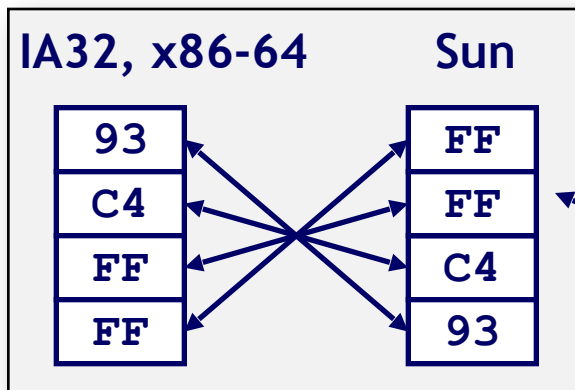
int A = 15213;



long int C = 15213;



int B = -15213;



Two's complement representation  
(Covered later)

# Representing Pointers

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

Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

**Different compilers & machines assign different locations to objects**

# Representing Strings

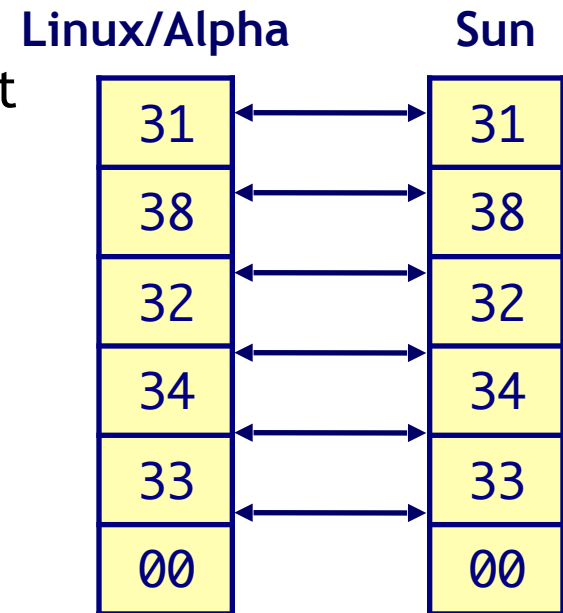
```
char S[6] = "18243";
```

## ■ 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



# Overview

- Representing information as bits
- **Bit-level manipulations**
- Integer representation
- Addition, negation, multiplication, shifting



# 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

### Not

- $\sim A = 1$  when  $A=0$

$\sim$	
0	1
1	0

### Or

- $A | B = 1$  when either  $A=1$  or  $B=1$

$ $	0	1
0	0	1
1	1	1

### Exclusive-Or (Xor)

- $A \wedge B = 1$  when either  $A=1$  or  $B=1$ , but not both

$\wedge$	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
<u>          </u>	<u>          </u>	<u>          </u>	<u>          </u>
01000001	01111101	00111100	10101010

## ■ All of the Properties of Boolean Algebra Apply

# Representing & Manipulating Sets

## ■ Representation

- Width  $w$  bit vector represents subsets of  $\{0, \dots, w-1\}$
- $a_j = 1$  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 $\&$ , $|$ , $\sim$ , $\wedge$ 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)

- $\sim 0x41 \rightarrow 0xBE$ 
  - $\sim 01000001_2 \rightarrow 10111110_2$
- $\sim 0x00 \rightarrow 0xFF$ 
  - $\sim 00000000_2 \rightarrow 11111111_2$
- $0x69 \& 0x55 \rightarrow 0x41$ 
  - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
- $0x69 | 0x55 \rightarrow 0x7D$ 
  - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

# Contrast: Logic Operations in C

## ■ Contrast to Logical Operators

- `&&`, `||`, `!`
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - **Early termination**

## ■ Examples (char data type)

- `!0x41` → `0x00`
- `!0x00` → `0x01`
- `!!0x41` → `0x01`
  
- `0x69 && 0x55` → `0x01`
- `0x69 || 0x55` → `0x01`
- `p && *p` (avoids null pointer access)

# Shift Operations

## ■ Left Shift: $x \ll y$

- Shift bit-vector  $x$  left  $y$  positions
  - Throw away extra bits on left
  - Fill with 0's on right

## ■ Right Shift: $x \gg y$

- Shift bit-vector  $x$  right  $y$  positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on right

## ■ Undefined Behavior

- Shift amount  $< 0$  or  $\geq$  word size

Argument $x$	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument $x$	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

# Overview

- Representing information as bits
- Bit-level manipulations
- **Integer representation**
- Addition, negation, multiplication, shifting

# Encoding Integers

## Unsigned

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

## Two's Complement

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

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

Sign  
Bit



## ■ C short 2 bytes long

	Decimal	Hex	Binary
<b>x</b>	15213	3B 6D	00111011 01101101
<b>y</b>	-15213	C4 93	11000100 10010011

## ■ Sign Bit

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



# Encoding Example (Cont.)

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

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

# Two-complement: Simple Example

	-16	8	4	2	1	
10 =	0	1	0	1	0	$8+2 = 10$

	-16	8	4	2	1	
-10 =	1	0	1	1	0	$-16+4+2 = -10$

# 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
<b>UMax</b>	<b>65535</b>	<b>FF FF</b>	<b>11111111 11111111</b>
<b>TMax</b>	<b>32767</b>	<b>7F FF</b>	<b>01111111 11111111</b>
<b>TMin</b>	<b>-32768</b>	<b>80 00</b>	<b>10000000 00000000</b>
<b>-1</b>	<b>-1</b>	<b>FF FF</b>	<b>11111111 11111111</b>
<b>0</b>	<b>0</b>	<b>00 00</b>	<b>00000000 00000000</b>

# Values for Different Word Sizes

	W			
	8	16	32	64
<b>UMax</b>	255	65,535	4,294,967,295	18,446,744,073,709,551,615
<b>TMax</b>	127	32,767	2,147,483,647	9,223,372,036,854,775,807
<b>TMin</b>	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

## ■ Observations

- $|TMin| = TMax + 1$ 
  - Asymmetric range
- $UMax = 2 * TMax + 1$

## ■ C Programming

- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific

# Unsigned & Signed Numeric Values

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

## ■ Equivalence

- Same encodings for nonnegative values

## ■ Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

## ■ ⇒ 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

# 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 <sub>1</sub>	Constant <sub>2</sub>	Relation Evaluation
0	0U	
-1	0	
-1	0U	
2147483647	-2147483647-1	
2147483647U	-2147483647-1	
-1	-2	
(unsigned)-1	-2	
2147483647	2147483648U	
2147483647	(int) 2147483648U	

# 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;  
}
```

- Copy at most **maxlen** bytes from kernel region to user buffer



# 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);
}
```

# Mali

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

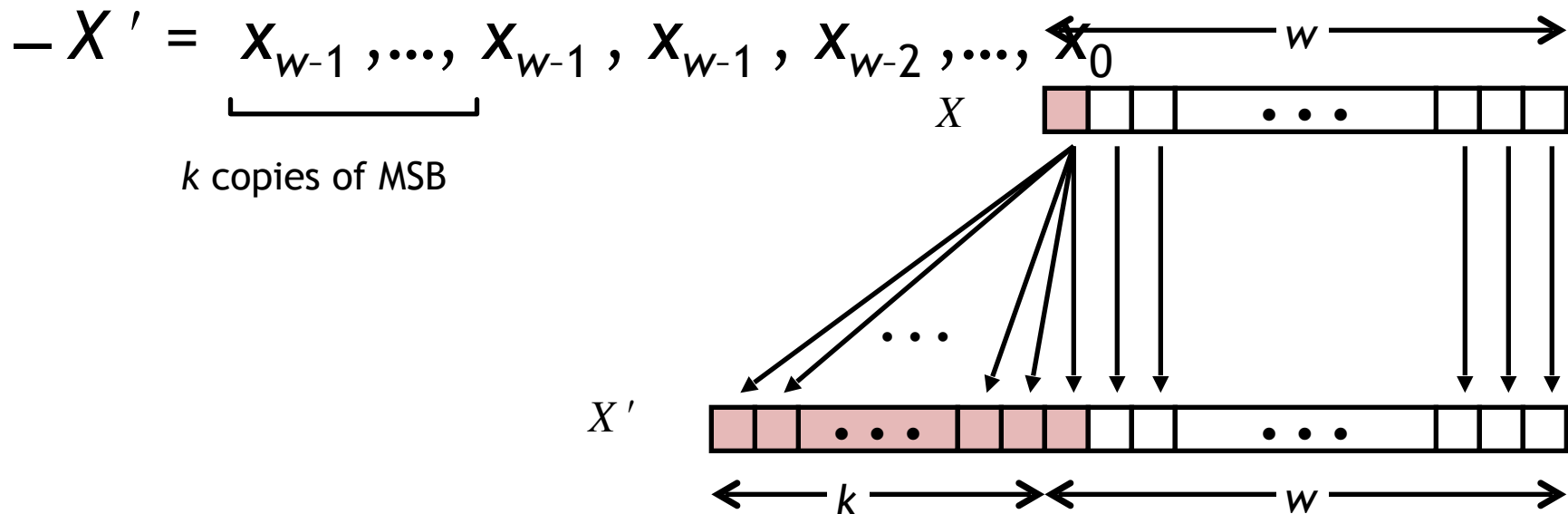
## Summary

### Casting Signed $\leftrightarrow$ 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`!!

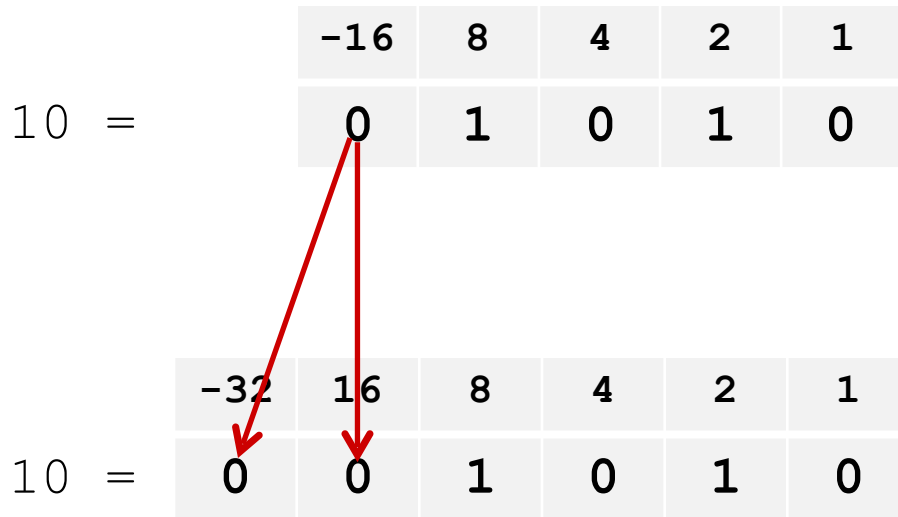
# Sign Extension

- Task:
  - Given  $w$ -bit signed integer  $x$
  - Convert it to  $w+k$ -bit integer with same value
- Rule:
  - Make  $k$  copies of sign bit:

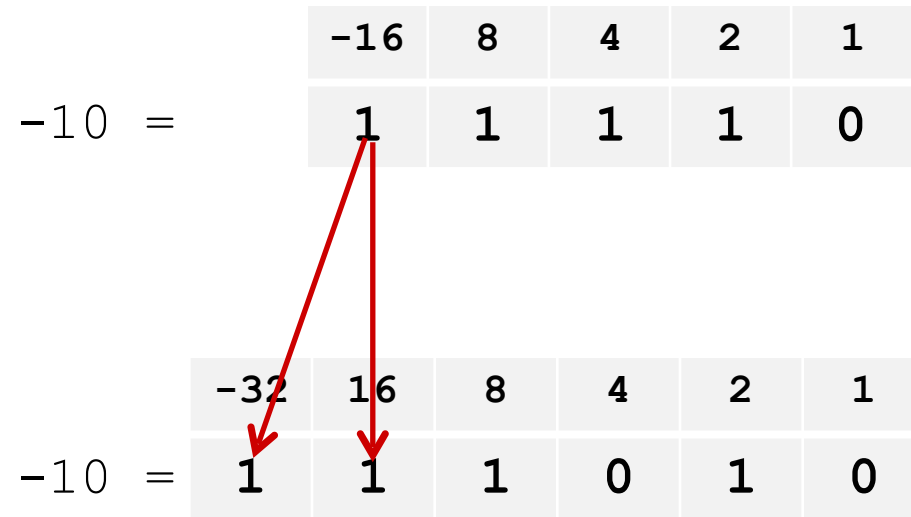


# Sign Extension: Simple Example

Positive number



Negative number



# Sign Extension Example

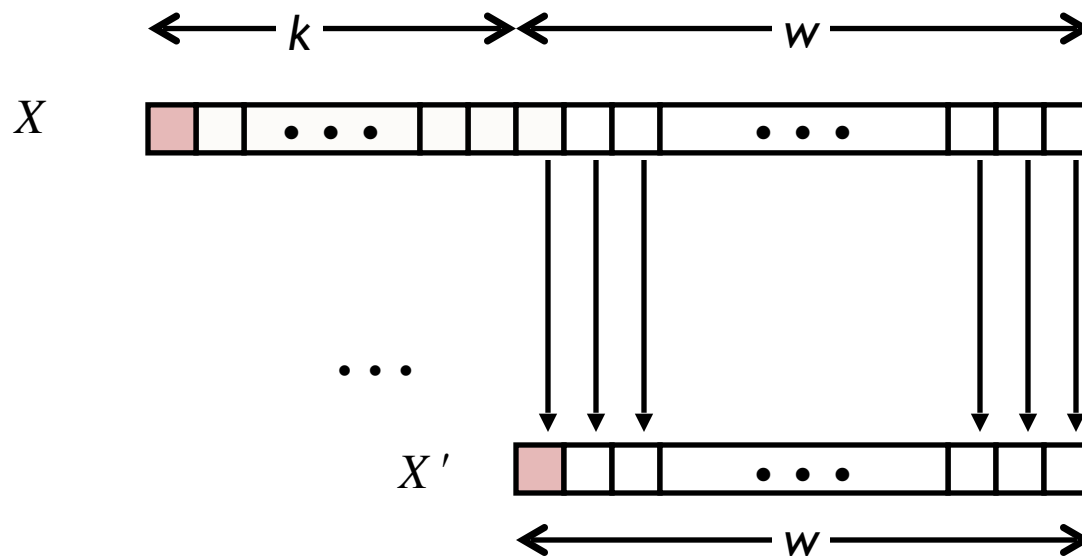
```
short int x = 15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	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

# Truncation

- Task:
  - Given  $k+w$ -bit signed or unsigned integer  $X$
  - Convert it to  $w$ -bit integer  $X'$  with same value for “small enough”  $X$
- Rule:
  - Drop top  $k$  bits:
  - $X' = x_{w-1}, x_{w-2}, \dots, x_0$



# Truncation: Simple Example

No sign change

 $2 =$ 

-16	8	4	2	1
0	0	0	1	0

 $2 =$ 

-8	4	2	1
0	0	1	0

 $-6 =$ 

-16	8	4	2	1
1	1	0	1	0

 $-6 =$ 

-8	4	2	1
1	0	1	0

Sign change

 $10 =$ 

-16	8	4	2	1
0	1	0	1	0

 $-6 =$ 

-8	4	2	1
1	0	1	0

 $-10 =$ 

-16	8	4	2	1
1	0	1	1	0

 $6 =$ 

-8	4	2	1
0	1	1	0



## 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

# Overview

- Representing information as bits
- Bit-level manipulations
- Integer representation
- **Addition, negation, multiplication, shifting**

# Negation: Complement & Increment

- Claim: Following Holds for 2's Complement

$$\sim x + 1 == -x$$

- Complement

– Observation:  $\sim x + x == 1111\dots111 == -1$

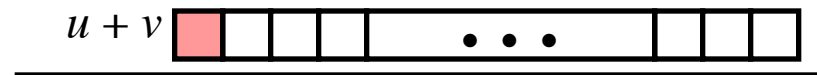
$$\begin{array}{r}
 x \quad \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \\
 + \quad \sim x \quad \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \\
 \hline
 -1 \quad \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1}
 \end{array}$$

# Unsigned Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits



# Two's Complement Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits

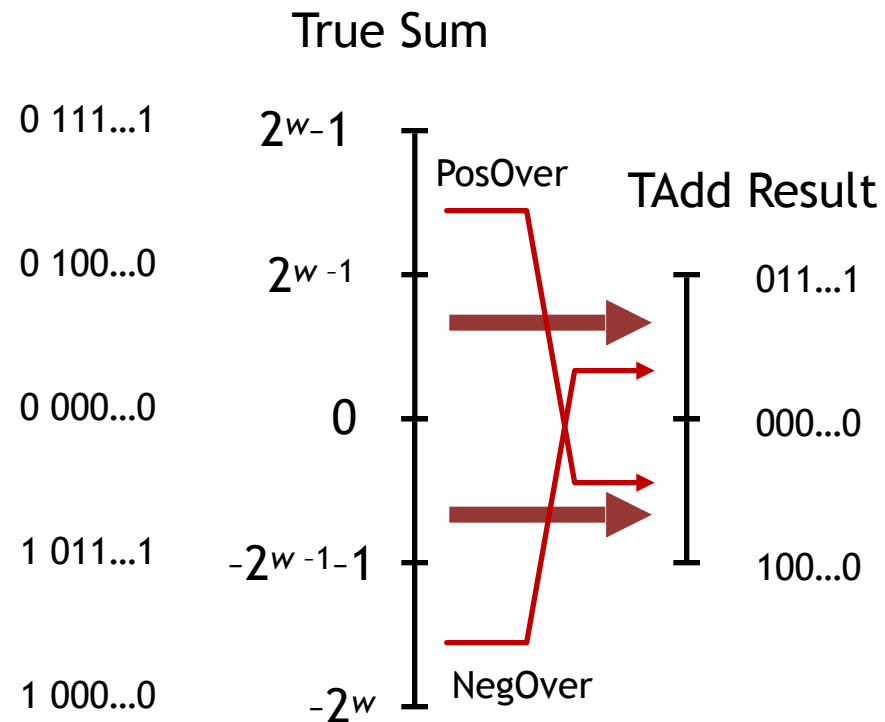


- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
 

```
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

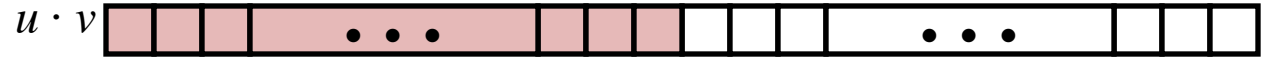


# Unsigned Multiplication in C

Operands:  $w$  bits



True Product:  $2 \cdot w$  bits



Discard  $w$  bits:  $w$  bits



- Standard Multiplication Function
  - Ignores high order  $w$  bits
- Implements Modular Arithmetic
 
$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

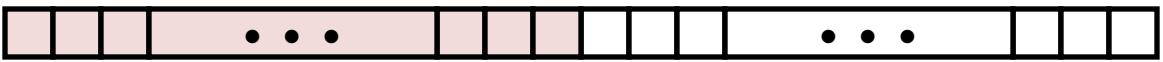
# Signed Multiplication in C

Operands:  $w$  bits

$u$  

$*$   $v$  

True Product:  $2*w$  bits

$u \cdot v$  

Discard  $w$  bits:  $w$  bits

$\text{TMult}_w(u, v)$  

- Standard Multiplication Function
  - Ignores high order  $w$  bits

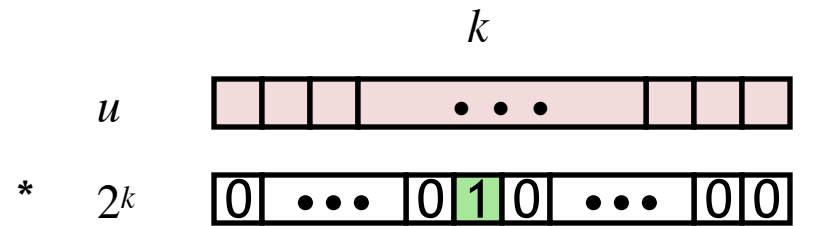


# Power-of-2 Multiply with Shift

- Operation

- $u \ll k$  gives  $u * 2^k$
- Both signed and unsigned

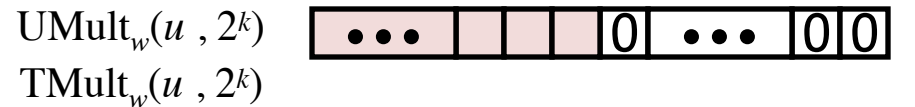
Operands:  $w$  bits



True Product:  $w+k$  bits



Discard  $k$  bits:  $w$  bits

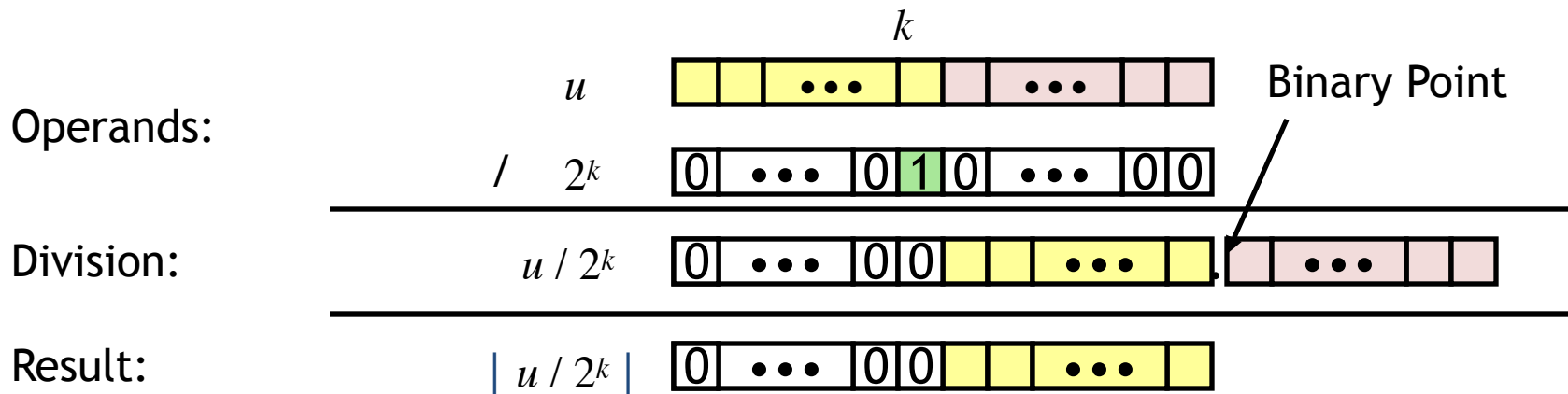


- Examples

- $u \ll 3 == u * 8$
- $u \ll 5 - u \ll 3 == u * 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

# Unsigned Power-of-2 Divide with Shift

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



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

# Study Guide: Bits Manipulations

- Basics

- `>>`, `<<`

- `|` vs. `||`

- `&` vs. `&&`

- `!` vs. `~`

- What is x?

- `int x = (9 | 12) << 1;`

- `x = 26`

# In-class Exercise

- Implement a bit-level storage where each item has the size of exactly 1 bit.

0      1      1      0      1      0      1      0

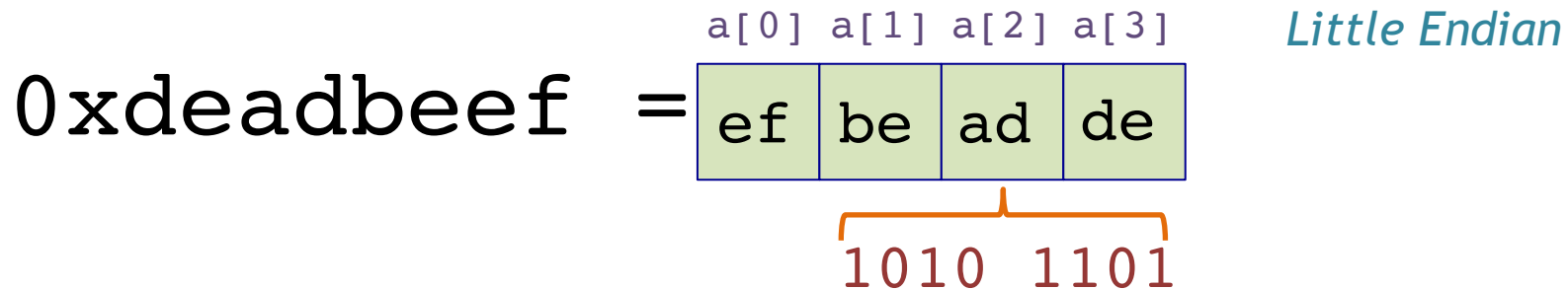
1 byte

- You will write the following functions

```
int get(unsigned char *arr,int idx);  
void set(unsigned char *arr, int idx, int val);
```

- Design your own tests and test your functions.

# How to...




---

```
unsigned char x = 0xad;
```

- How to retrieve any bit from x?
  - What's the i-th bit? E.g., `get_bit(&x, 3)` → 1
- How to set any bit without affecting others?
  - `set_bit(&x, 2, 0)` -> 1010 1001 (== 0xa9)
- How to print all the bits?

# How to set any bit?

Idea: masking

