

Bits, Bytes and Integers

Introduction to Computer Systems

Danfeng Shan

Xi'an Jiaotong University

Announcements

ICSServer已准备好

注意：请不要恶意占用服务器资源！包括不仅限于挖矿、跑大模型！

两个Tutorial

使用SSH登录ICSServer（胡博瑄）

Bash Shell（李雨轩）

Lab 1 (Data Lab) 明天公布，截止时间：03.09（周日）

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed

Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

Everything is bits

Each bit is 0 or 1

By encoding/interpreting sets of bits in various ways

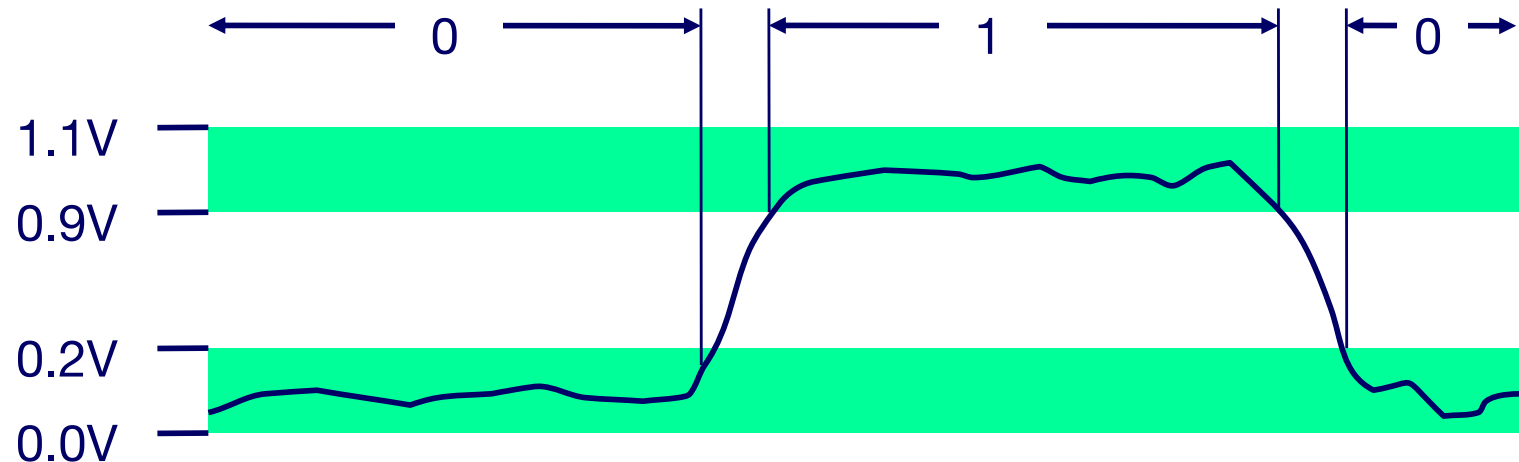
Computers determine what to do (instructions)

... and represent and manipulate numbers, sets, strings, etc...

Why bits? Electronic Implementation

Easy to store with bistable elements (双稳态器件)

Reliably transmitted on noisy and inaccurate wires



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 |

15213: 0011 1011 0110 1101
 3 B 6 D

Example Data Representations

| C Data Type | Typical 32-bit | Typical 64-bit |
|----------------------|----------------|----------------|
| <code>char</code> | 1 | 1 |
| <code>short</code> | 2 | 2 |
| <code>int</code> | 4 | 4 |
| <code>long</code> | 4 | 8 |
| <code>float</code> | 4 | 4 |
| <code>double</code> | 8 | 8 |
| <code>pointer</code> | 4 | 8 |

Example Data Representations

| C Data Type | Typical 32-bit | Typical 64-bit |
|----------------------|----------------|----------------|
| <code>char</code> | 1 | 1 |
| <code>short</code> | 2 | 2 |
| <code>int</code> | 4 | 4 |
| <code>long</code> | 4 | 8 |
| <code>float</code> | 4 | 4 |
| <code>double</code> | 8 | 8 |
| <code>pointer</code> | 4 | 8 |

Example Data Representations

| C Data Type | Typical 32-bit | Typical 64-bit |
|----------------|----------------|----------------|
| char | 1 | 1 |
| short | 2 | 2 |
| int | 4 | 4 |
| long | 4 | 8 |
| float | 4 | 4 |
| double | 8 | 8 |
| pointer | 4 | 8 |

“ILP32”

“LP64”

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed

Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

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 $B=1$ or both

| $ $ | 0 | 1 |
|-----|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Not

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

| \sim | 0 | 1 |
|--------|---|---|
| | 1 | 0 |

Exclusive-Or (Xor)

$A \wedge B = 1$ when $A=1$ 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

| | | | |
|---|--|---|---|
| $\begin{array}{r} 01101001 \\ \& 01010101 \\ \hline 01000001 \end{array}$ | $\begin{array}{r} 01101001 \\ 01010101 \\ \hline 01111101 \end{array}$ | $\begin{array}{r} 01101001 \\ \wedge 01010101 \\ \hline 00111100 \end{array}$ | $\begin{array}{r} \sim 01010101 \\ \hline 10101010 \end{array}$ |
|---|--|---|---|

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

Contrast: Logic Operations in C

Contrast to Bit-Level Operators

Logic Operations: `&&`, `||`, `!`

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)

Watch out for `&&` vs. `&` (and `||` vs. `|`)...
Super common C programming pitfall!

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 left

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 |

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed, negation

Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

Question?

```
int foo = -1;  
unsigned bar = 1;  
  
(foo < bar) == true ?
```


Encoding “Integers”

Unsigned

Given a bit vector x ,
 w bits long...

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

Signed (twos complement)

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

← **Sign Bit**

Examples ($w = 5$)

| ± 16 | 8 | 4 | 2 | 1 |
|----------|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 |

$$0 + 8 + 0 + 2 + 0 = 10$$

| 16 | 8 | 4 | 2 | 1 |
|-----|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| -16 | 8 | 4 | 2 | 1 |

$$16 + 8 + 0 + 2 + 0 = 26$$

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

Negation: Complement & Increment

Negate through complement and increase

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

Why?

$$-x + x == 0 \text{ (by definition)}$$

$$\sim x + x == 1111\dots111 == -1$$

$$\sim x + x + 1 == 0$$

$$(\sim x + 1) + x == 0$$

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

$$\begin{array}{r} x \quad 10011101 \\ + \quad \sim x \quad 01100010 \\ \hline -1 \quad 11111111 \end{array}$$

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 |


Complement & Increment Examples

$$x = 0$$

| | Decimal | Hex | Binary |
|--------------|---------|-------|-------------------|
| 0 | 0 | 00 00 | 00000000 00000000 |
| ~ 0 | -1 | FF FF | 11111111 11111111 |
| $\sim 0 + 1$ | 0 | 00 00 | 00000000 00000000 |

$$x = T_{\min}$$

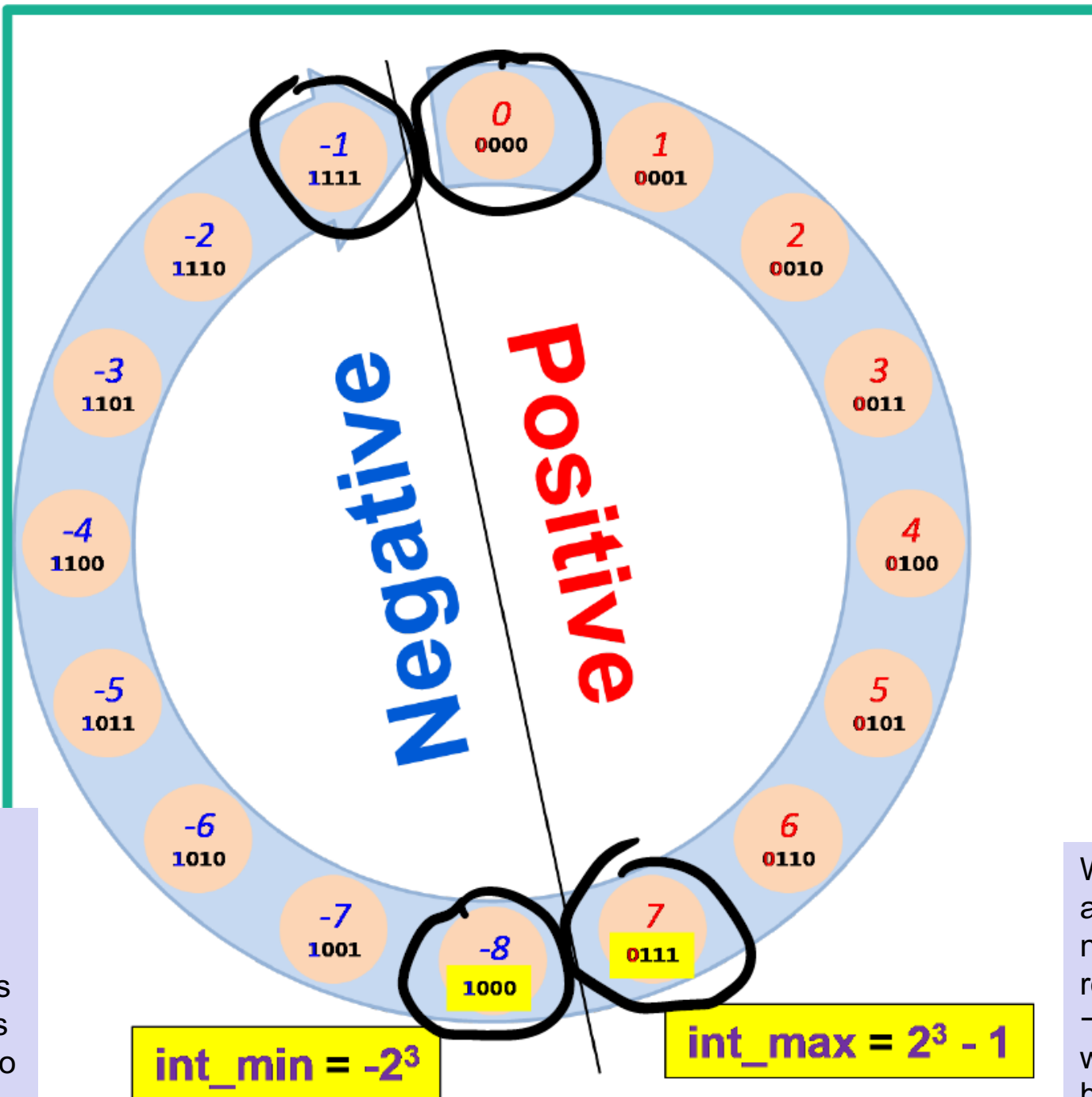
| | Decimal | Hex | Binary |
|--------------|---------|-------|-------------------|
| x | -32768 | 80 00 | 10000000 00000000 |
| $\sim x$ | 32767 | 7F FF | 01111111 11111111 |
| $\sim x + 1$ | -32768 | 80 00 | 10000000 00000000 |



Oops!
It's still
negative!

Eight negative values:
-1, -2, ..., -8

Mathematicians would prefer it if a 4-bit signed number could represent values -8...8, but that's $2^4 + 1$ values, so they won't all fit.



Eight *non*-negative values:
0, 1, ..., 7

What if we made a 4-bit signed number only represent values -7...7? Then we wouldn't be using bit pattern 1000...

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed, negation

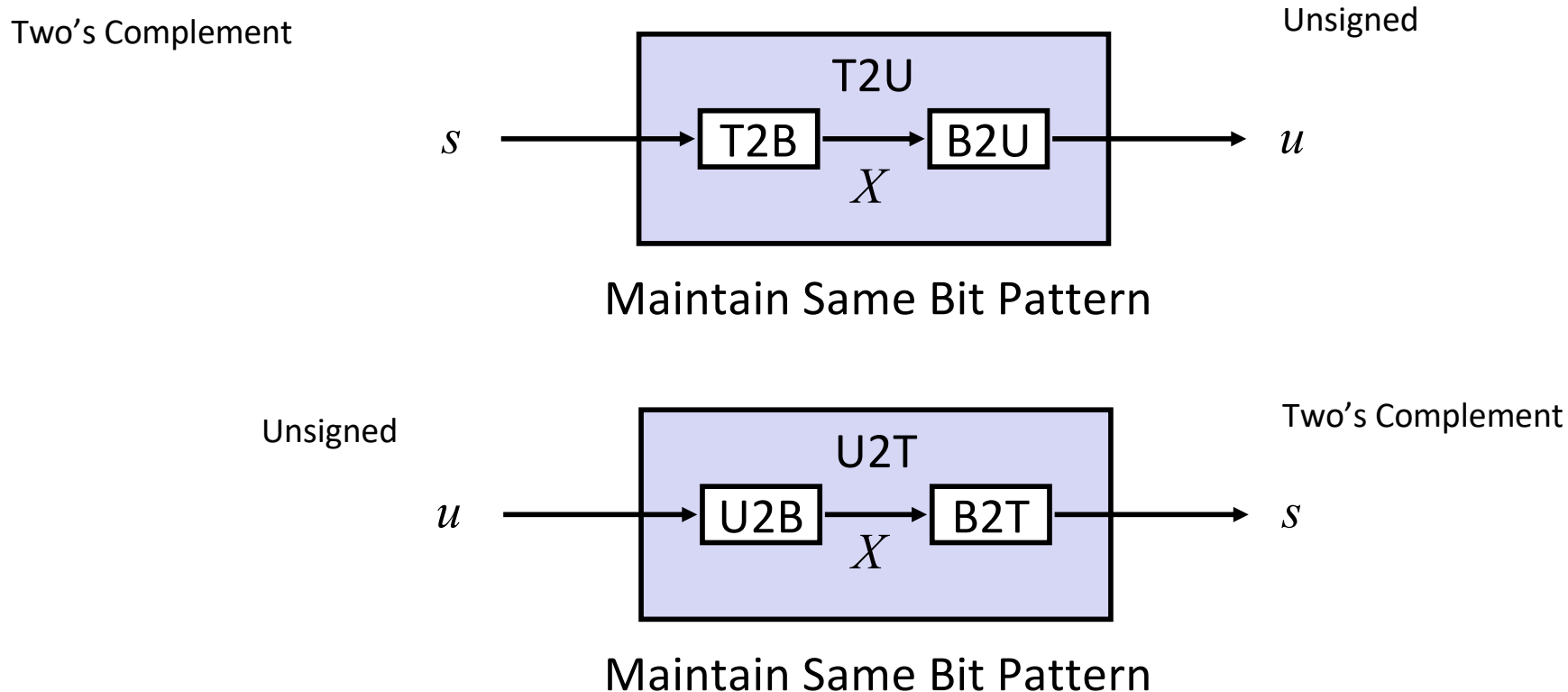
Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

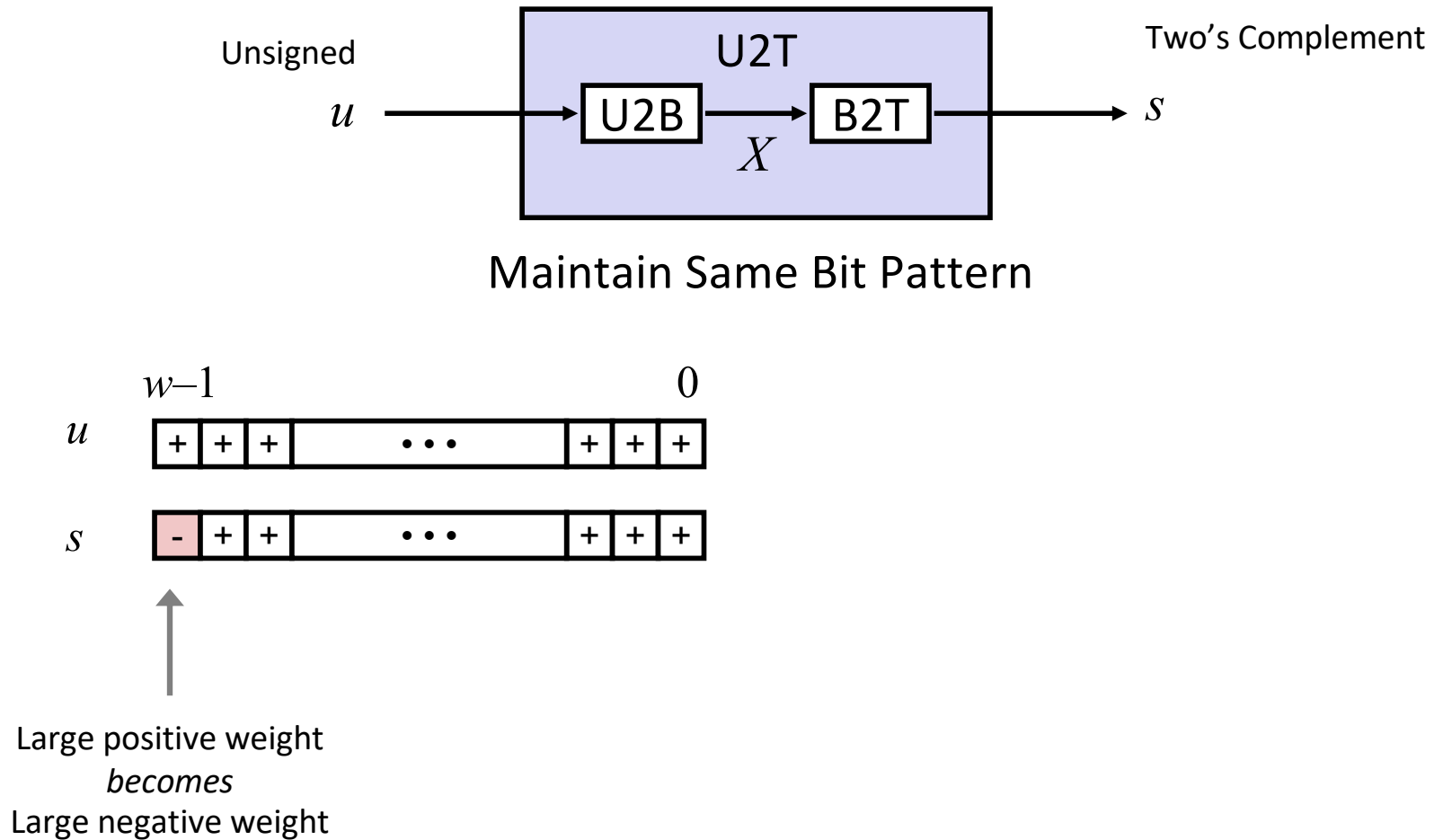
Representations in memory, pointers, strings

Mapping Between Signed & Unsigned



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

Relation between Signed & Unsigned



Mapping Signed ↔ Unsigned

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

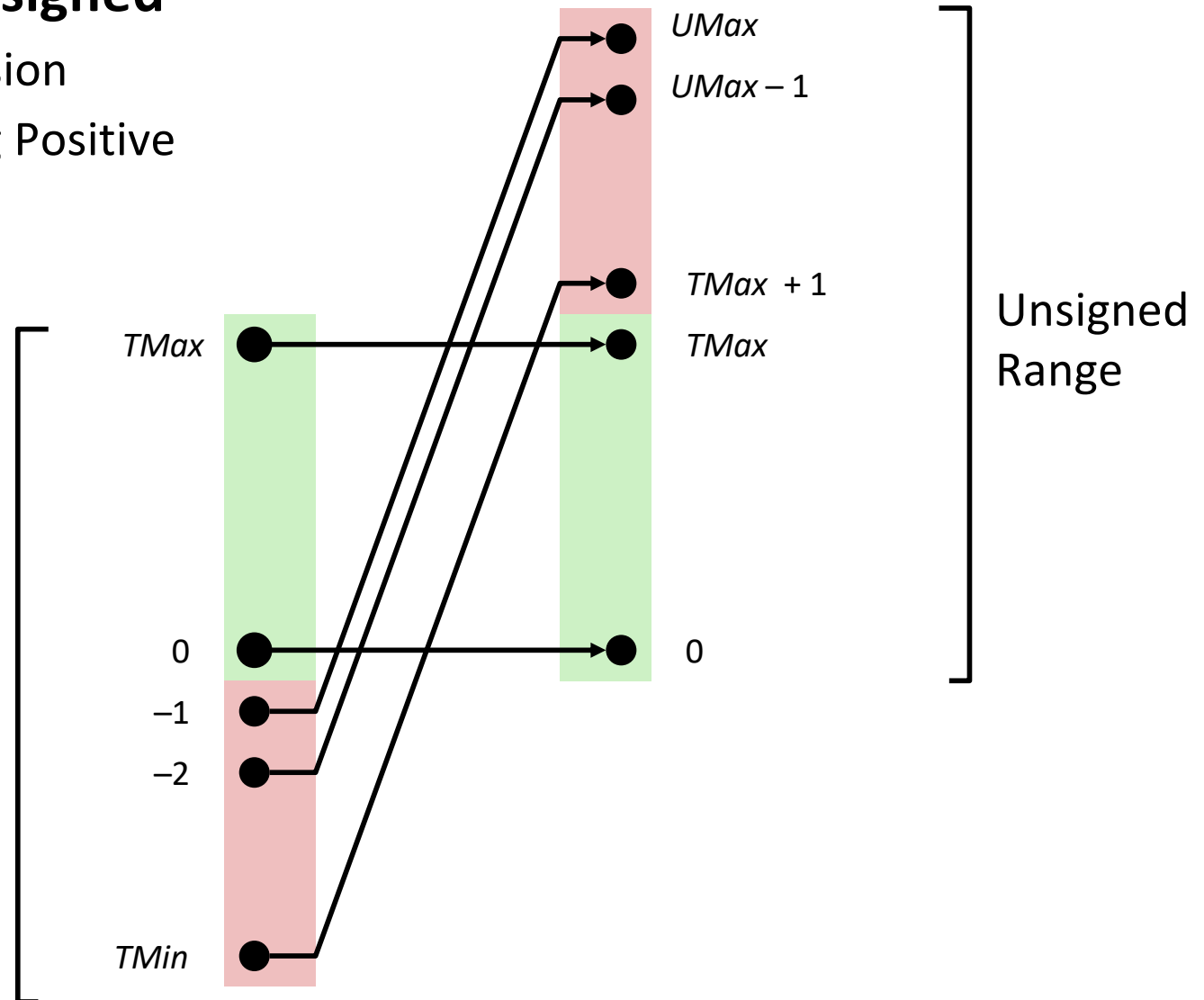
Conversion Visualized

2's Comp. → Unsigned

Ordering Inversion

Negative → Big Positive

2's Complement
Range



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;                                int fun(unsigned u) ;  
uy = ty;                                uy = fun(tx) ;
```

Question?

example02.c

```
int foo = -1;  
unsigned bar = 1;  
  
foo < bar == true ?
```

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:

| Constant 1 | Constant 2 | Relation | Evaluation |
|--------------------|--------------------------------|----------|------------|
| 0 | 0U | == | Unsigned |
| -1 | 0 | < | Signed |
| -1 | 0U | > | Unsigned |
| INT_MAX | INT_MIN | > | Signed |
| (unsigned) INT_MAX | INT_MIN | < | Unsigned |
| -1 | -2 | > | Signed |
| (unsigned) -1 | -2 | > | Unsigned |
| INT_MAX | ((unsigned) INT_MAX) + 1 | < | Unsigned |
| INT_MAX | (int) ((unsigned) INT_MAX) + 1 | > | Signed |

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`!!

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed, negation

Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

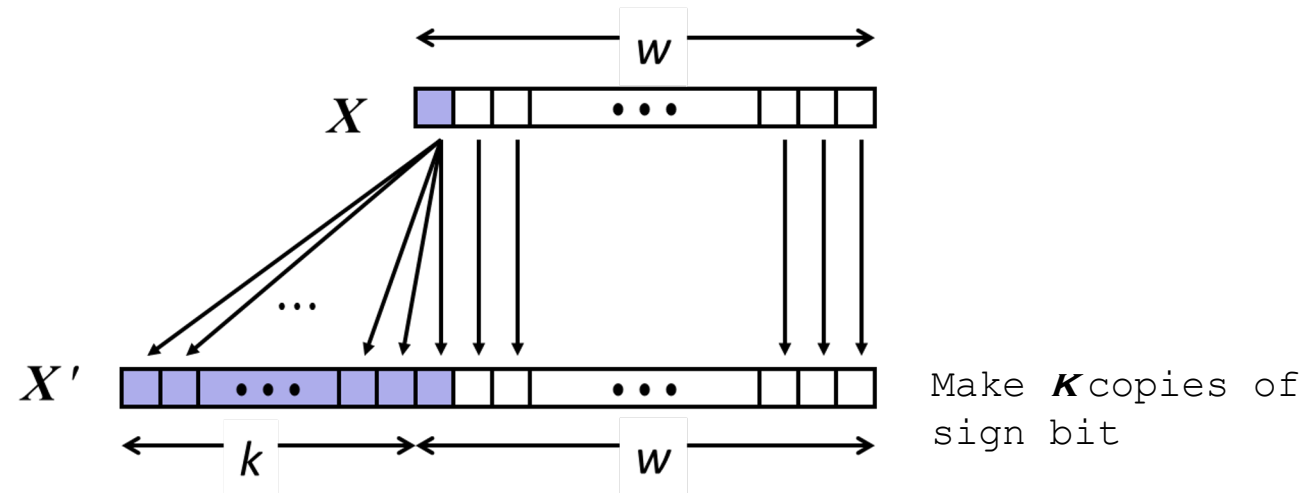
Question?

example03.c

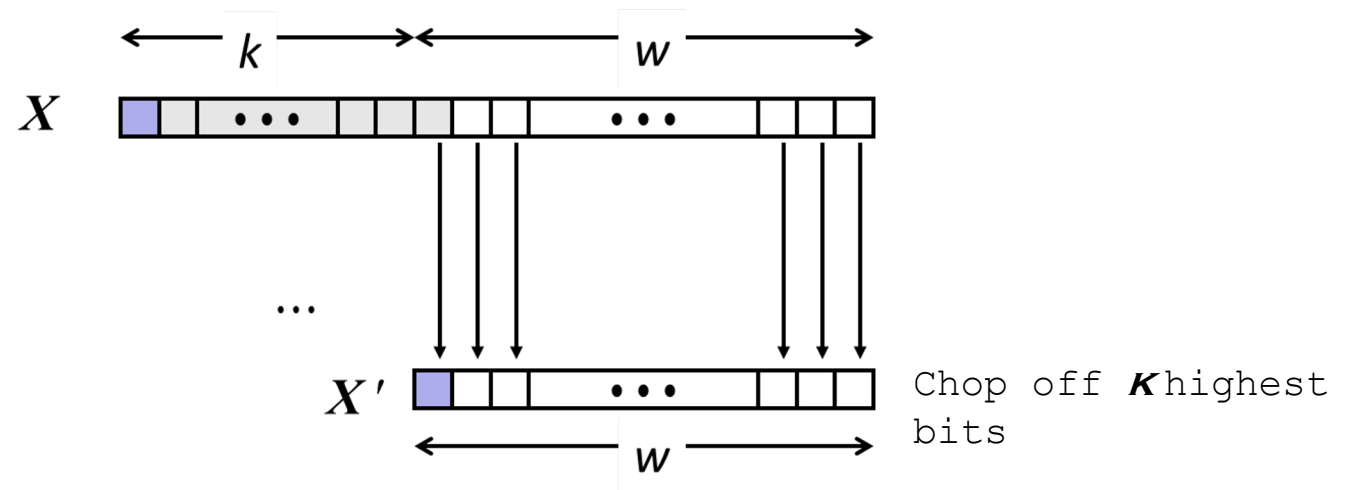
```
int x = 0x8000;  
short sx = (short) x;  
int y = sx;
```

Sign Extension and Truncation

Sign Extension

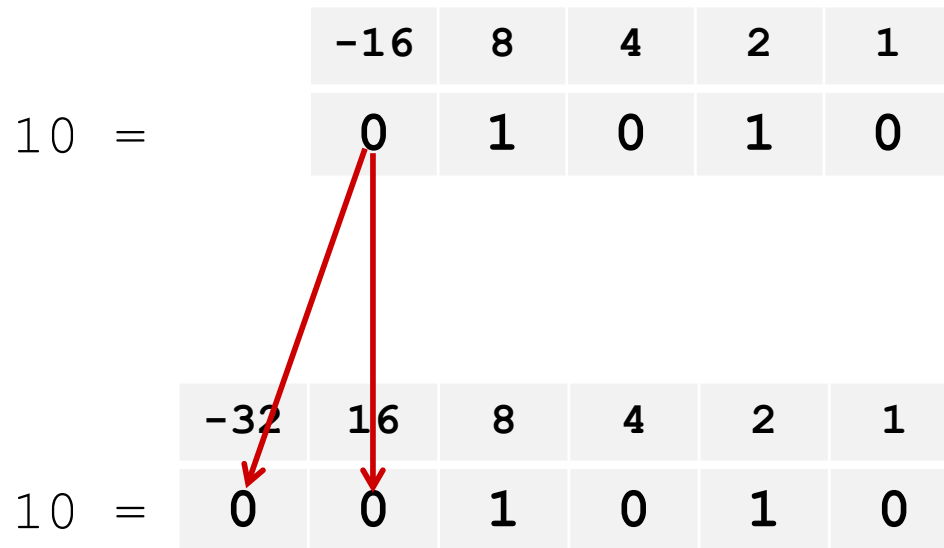


Truncation

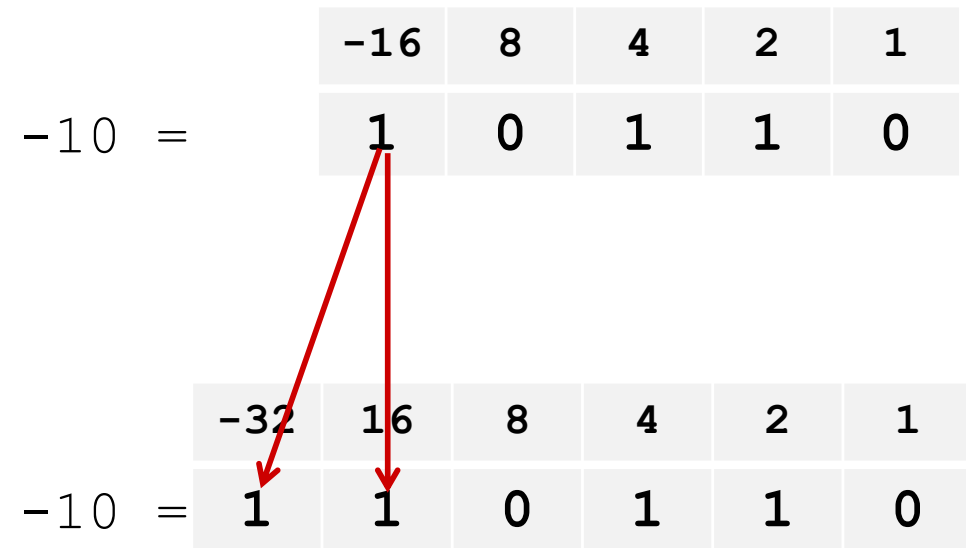


Sign Extension: Simple Example

Positive number



Negative number



Truncation: Simple Example

No sign change

| | | | | | |
|-----|-----|---|---|---|---|
| | -16 | 8 | 4 | 2 | 1 |
| 2 = | 0 | 0 | 0 | 1 | 0 |

| | | | | |
|-----|----|---|---|---|
| | -8 | 4 | 2 | 1 |
| 2 = | 0 | 0 | 1 | 0 |

| | | | | | |
|------|-----|---|---|---|---|
| | -16 | 8 | 4 | 2 | 1 |
| -6 = | 1 | 1 | 0 | 1 | 0 |

| | | | | |
|------|----|---|---|---|
| | -8 | 4 | 2 | 1 |
| -6 = | 1 | 0 | 1 | 0 |

Sign change

| | | | | | |
|------|-----|---|---|---|---|
| | -16 | 8 | 4 | 2 | 1 |
| 10 = | 0 | 1 | 0 | 1 | 0 |

| | | | | |
|------|----|---|---|---|
| | -8 | 4 | 2 | 1 |
| -6 = | 1 | 0 | 1 | 0 |

| | | | | | |
|-------|-----|---|---|---|---|
| | -16 | 8 | 4 | 2 | 1 |
| -10 = | 1 | 0 | 1 | 1 | 0 |

| | | | | |
|-----|----|---|---|---|
| | -8 | 4 | 2 | 1 |
| 6 = | 0 | 1 | 1 | 0 |

Question?

example03.c

```
int x = 0x8000;  
short sx = (short) x;  
int y = sx;
```

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed

Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

Unsigned Addition

Operands: w bits



True Sum: $w+1$ bits



Discard Carry: w bits



Standard Addition Function

Ignores carry output

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 |

unsigned char

| | | |
|-------------|-------|-------|
| 1110 1001 | E9 | 233 |
| + 1101 0101 | + D5 | + 213 |
| <hr/> | <hr/> | <hr/> |
| <hr/> | <hr/> | <hr/> |

Unsigned Addition

Operands: w bits



True Sum: $w+1$ bits



Discard Carry: w bits



Standard Addition Function

Ignores carry output

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 |

unsigned char

```

      1110 1001
    + 1101 0101
    -----
  1 1011 1110
    -----
      1011 1110
  
```

```

      E9
    + D5
    -----
    1BE
    -----
      BE
  
```

```

      233
    + 213
    -----
    446
    -----
      190
  
```

Visualizing (Mathematical) Integer Addition

Integer Addition

4-bit integers u, v

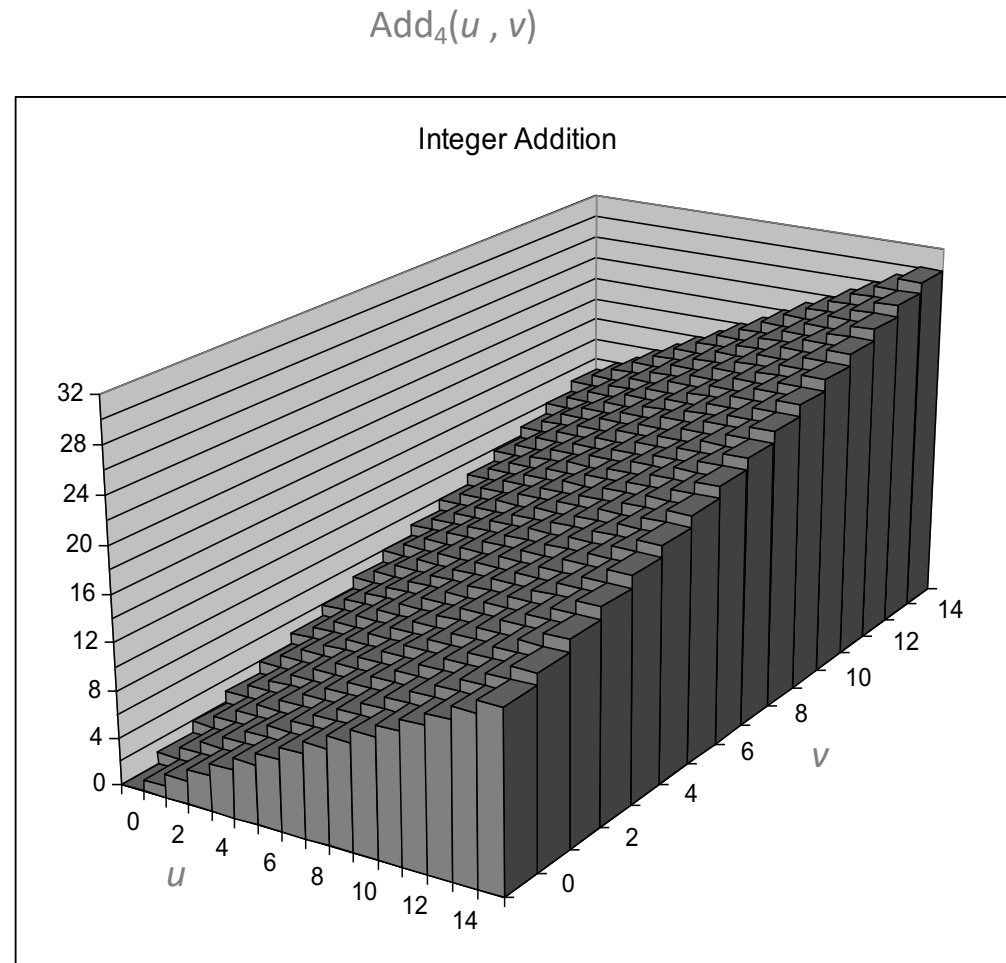
Compute true sum

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

Values increase

linearly with u and v

Forms planar surface



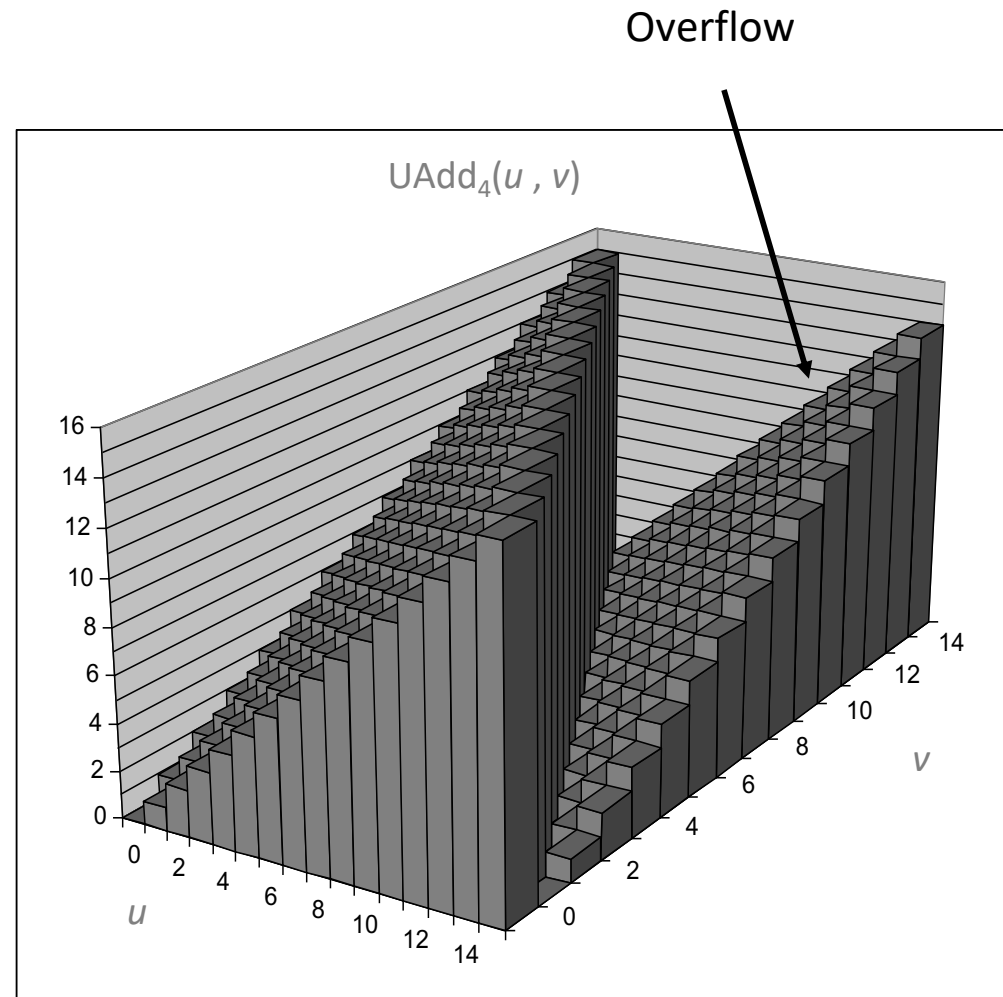
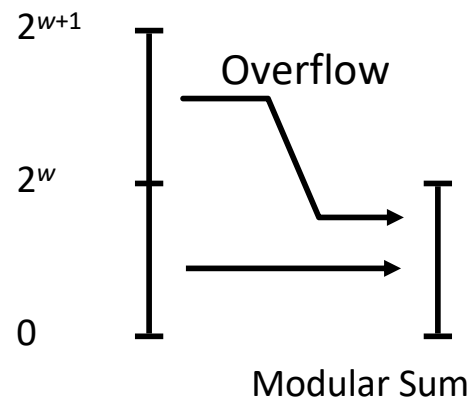
Visualizing Unsigned Addition

Wraps Around

If true sum $\geq 2^w$

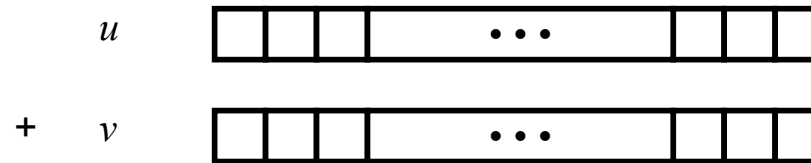
At most once

True Sum

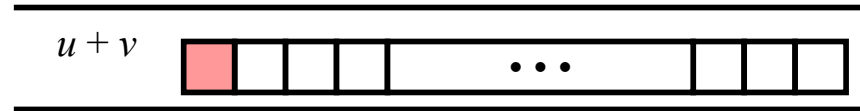


Two's Complement Addition

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 $s == t$

| | | | |
|---|-------------|-------|-------|
| | 1110 1001 | E9 | -23 |
| + | 1101 0101 | + D5 | + -43 |
| | <hr/> | <hr/> | <hr/> |
| | 1 1011 1110 | 1BE | -66 |
| | <hr/> | <hr/> | <hr/> |
| | 1011 1110 | BE | -66 |

Visualizing 2's Complement Addition

Values

4-bit two's comp.

Range from -8 to +7

Wraps Around

If $\text{sum} \geq 2^{w-1}$

Becomes
negative

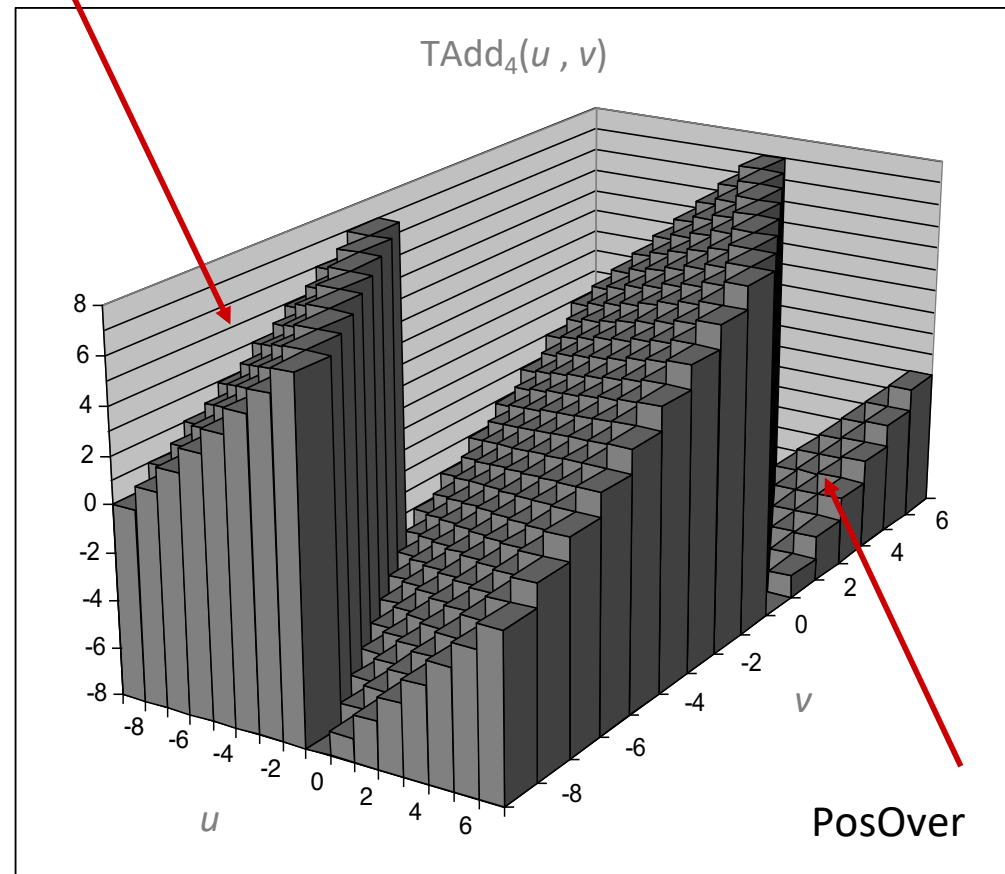
At most once

If $\text{sum} < -2^{w-1}$

Becomes
positive

At most once

NegOver



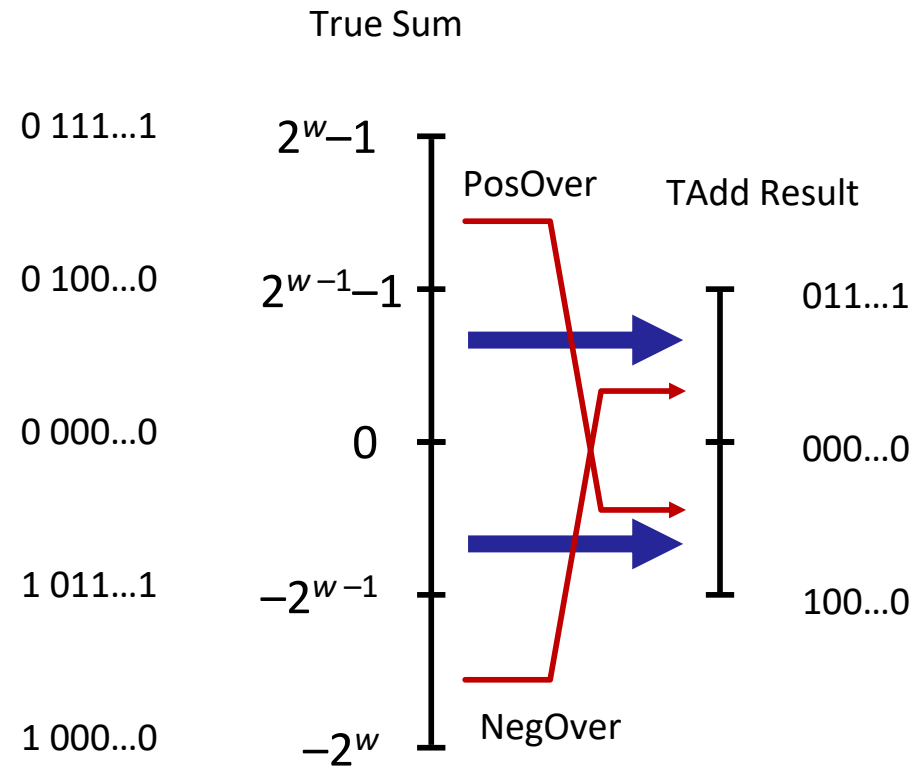
TAdd Overflow

Functionality

True sum requires $w+1$ bits

Drop off MSB

Treat remaining bits as 2's comp. integer



Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed

Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

Shifting

Left Shift: $x \ll y$

Shift bit-vector x left y positions

Throw away extra bits on left

Fill with 0's on right

Equivalent to multiplying by 2^y

Right Shift: $x \gg y$

Shift bit-vector x right y positions

Throw away extra bits on right

Two kinds:

“Logical”: Fill with 0's on left

“Arithmetic”: Replicate most significant bit on left

Almost equivalent to dividing by 2^y

Undefined Behavior (in C)

Shift amount < 0 or \geq word size

| | |
|--------------------|----------|
| Argument x | 01100010 |
| $\ll 3$ | 00010000 |
| Logical $\gg 2$ | 00011000 |
| Arithmetic $\gg 2$ | 00011000 |

| | |
|--------------------|----------|
| Argument x | 10100010 |
| $\ll 3$ | 00010000 |
| Logical $\gg 2$ | 00101000 |
| Arithmetic $\gg 2$ | 11101000 |

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 $2w$ bits

$$\text{Result range: } 0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$$

Two's complement min (negative): Up to $2w-1$ bits

$$\text{Result range: } x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$$

Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$

$$\text{Result range: } x * y \leq (-2^{w-1})^2 = 2^{2w-2}$$

So, maintaining exact results...

would need to keep expanding word size with each product computed
is done in software, if needed

Unsigned Multiplication in C

Operands: w bits

u 

$*$ v 

True Product: $2*w$ bits

$u \cdot v$ 

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

Discard w bits: w bits



Standard Multiplication Function

Ignores high order w bits

```

      1110 1001
    * 1101 0101
    -----
1100 0001 1101 1101
    -----
      1101 1101
  
```

```

      E9
    * D5
    -----
    C1DD
      DD
  
```

```

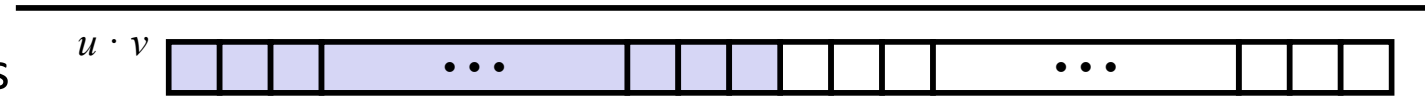
      233
    * 213
    -----
    49629
      221
  
```

Signed Multiplication in C

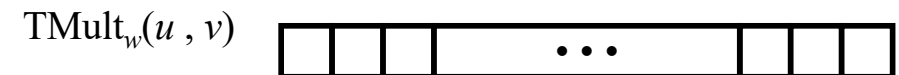
Operands: w bits



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

$$\begin{array}{r}
 1110 \ 1001 \\
 * 1101 \ 0101 \\
 \hline
 0000 \ 0011 \ 1101 \ 1101 \\
 \hline
 1101 \ 1101
 \end{array}$$

$$\begin{array}{r}
 \text{E9} \\
 * \text{D5} \\
 \hline
 03DD \\
 \hline
 \text{DD}
 \end{array}$$

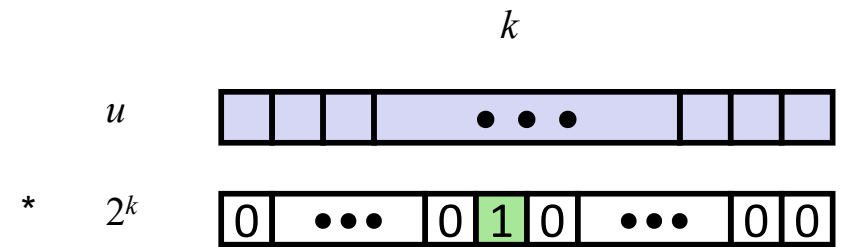
Power-of-2 Multiply with Shift

Operation

$u \ll k$ gives $u * 2^k$

Both signed and unsigned

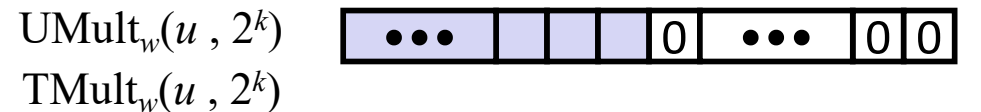
Operands: w bits



True Product: $w+k$ bits



Discard k bits: w bits



Examples

$u \ll 3 \quad == \quad u * 8$

$(u \ll 5) - (u \ll 3) \quad == \quad u * 24$

Most machines shift and add faster than multiply

Compiler generates this code automatically

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

Representation: unsigned and signed

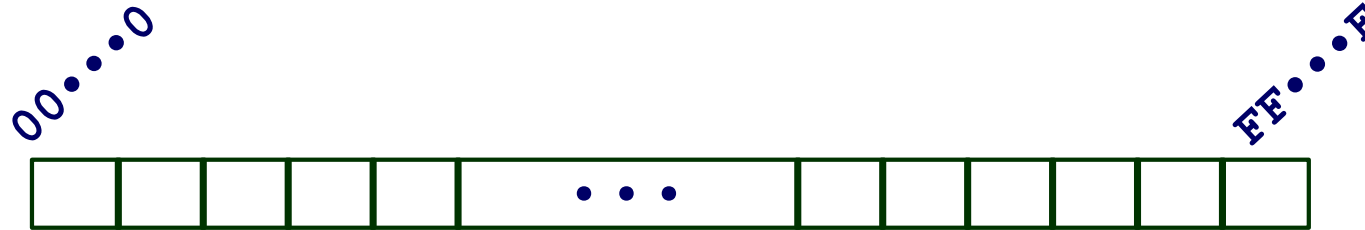
Conversion, casting

Expanding, truncating

Addition, multiplication, shifting

Representations in memory, pointers, strings

Byte-Oriented Memory Organization



Programs refer to data by address

Imagine all of RAM as an enormous array of bytes

An address is an index into that array

A pointer variable stores an address

System provides a private *address space* to each “process”

A process is an instance of a program, being executed

An address space is one of those enormous arrays of bytes

Each program can see only its own code and data within its enormous array

We'll come back to this later (“virtual memory” classes)

Machine Words

Any given computer has a “Word Size”

Nominal size of integer-valued data
and of addresses

Historically, most machines used 32 bits (4 bytes) as word size
Limits addresses to 4GB (2^{32} bytes)

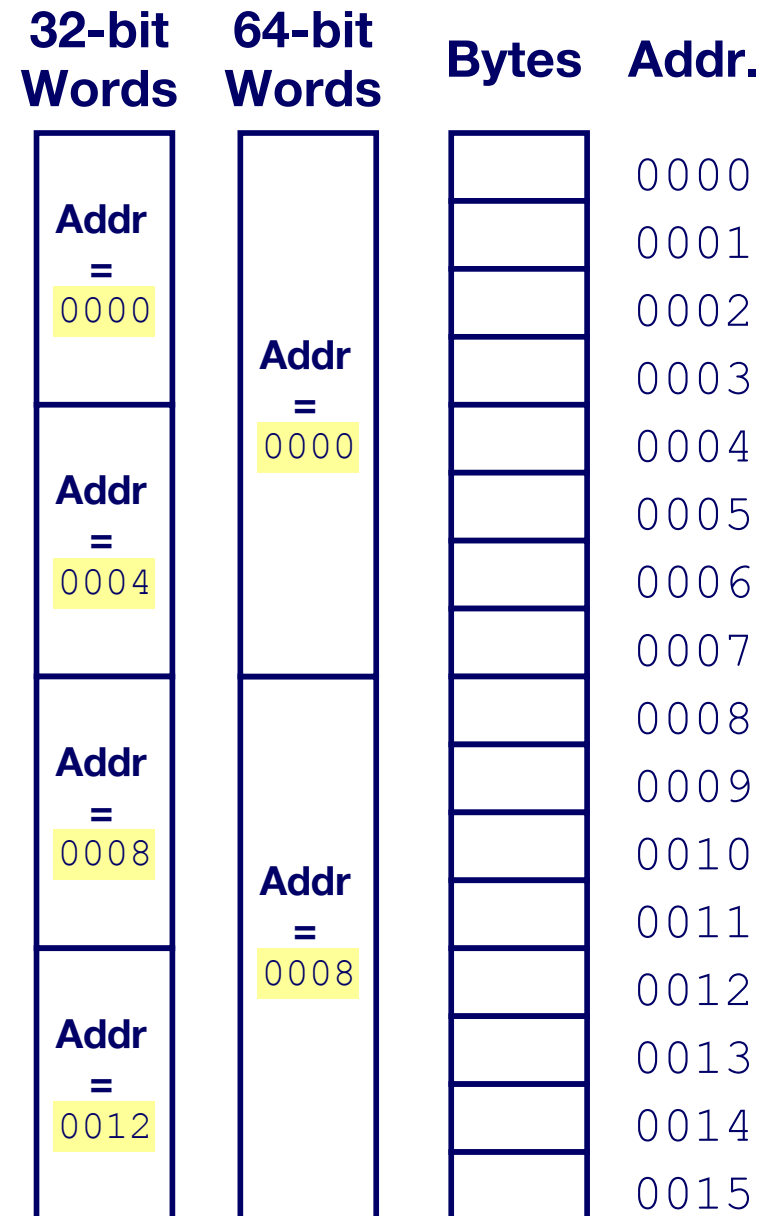
Currently, machines have 64-bit word size
Potentially, could have 16 EB (exabytes) of addressable memory
That's 18.4×10^{18} bytes

Machines still support multiple data formats
Fractions or multiples of word size
Always integral number of bytes

Addresses *Always* Specify Byte Locations

Address of a word is address of the first byte in the 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 |
|----------------------|----------------|----------------|--------|
| <code>char</code> | 1 | 1 | 1 |
| <code>short</code> | 2 | 2 | 2 |
| <code>int</code> | 4 | 4 | 4 |
| <code>long</code> | 4 | 8 | 8 |
| <code>float</code> | 4 | 4 | 4 |
| <code>double</code> | 8 | 8 | 8 |
| <code>pointer</code> | 4 | 8 | 8 |

Question?

example04.c

```
struct foo {  
    char mem1[3]; // 3 bytes  
    int mem2;      // 4 bytes  
    char mem3;     // 1 byte  
};  
sizeof(struct foo) = ?
```

Byte Ordering

So, how are the bytes within a multi-byte word ordered in memory?

Conventions

Big Endian: Sun, PPC Mac, *network packet headers*

Least significant byte has highest address

Little Endian: *x86*, ARM processors running Android, iOS, and Windows

Least significant byte has lowest address

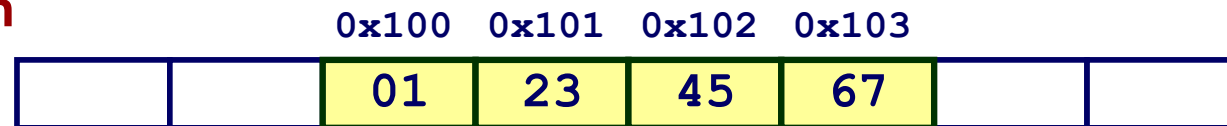
Byte Ordering Example

Example

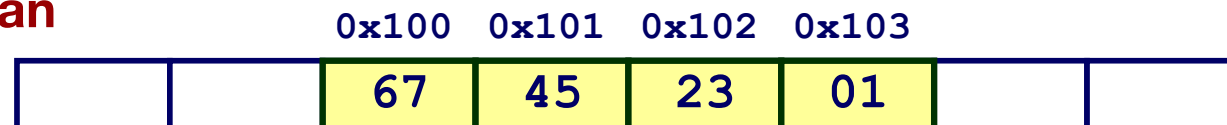
Variable x has 4-byte value of 0x01234567

Address given by &x is 0x100

Big Endian



Little Endian



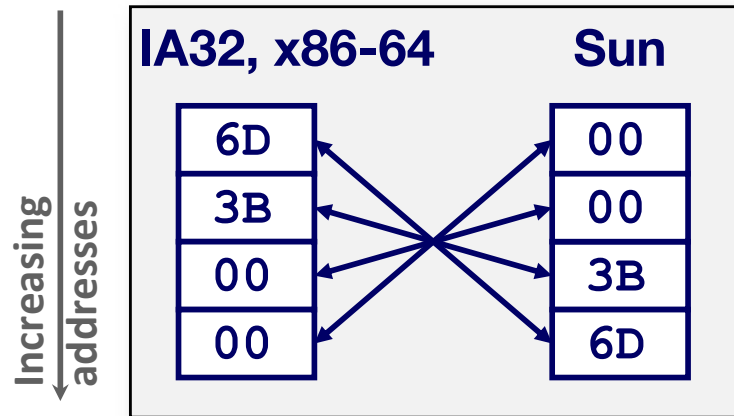
Representing Integers

Decimal: 15213

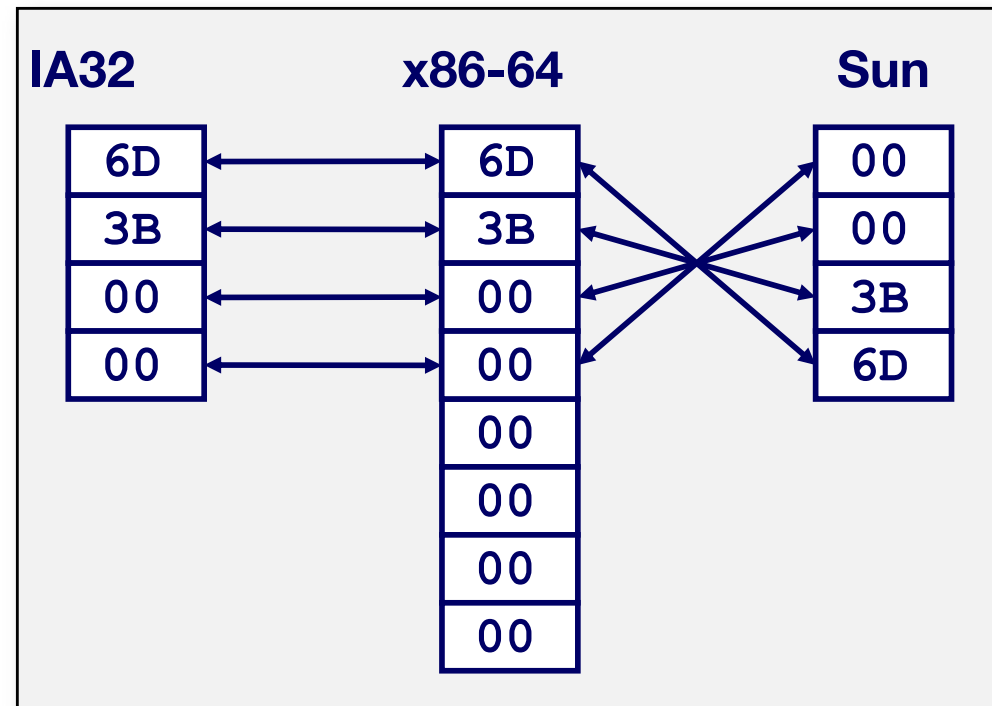
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

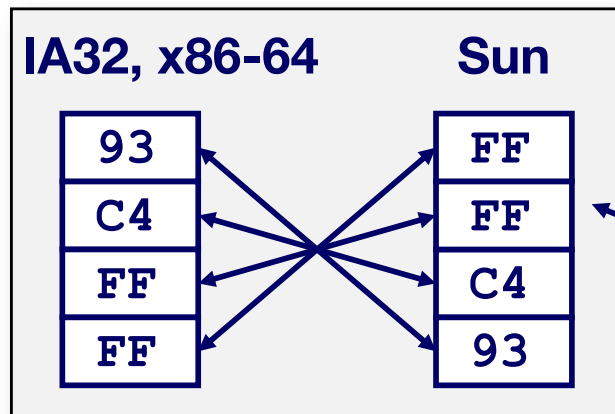
`int A = 15213;`



`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;  
0x7ffffb7f71dbc      6d  
0x7ffffb7f71dbd      3b  
0x7ffffb7f71dbe      00  
0x7ffffb7f71dbf      00
```

Representing Pointers

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

| Sun | IA32 | x86-64 |
|-----|------|--------|
| EF | AC | 3C |
| FF | 28 | 1B |
| FB | F5 | FE |
| 2C | FF | 82 |
| | | FD |
| | | 7F |
| | | 00 |
| | | 00 |

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

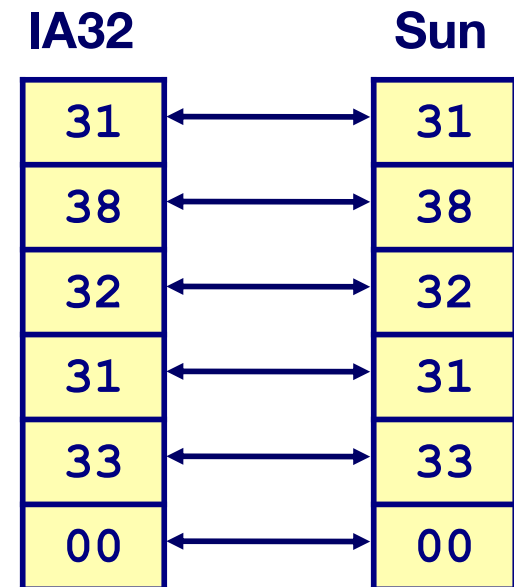
– Digit i has code $0x30+i$

String should be null-terminated

Final character = 0

Compatibility

Byte ordering not an issue



Representing x86 machine code

x86 machine code is a sequence of *bytes*

Grouped into variable-length instructions, which look like strings...

But they contain embedded little-endian numbers...

Example Fragment

| Address | Instruction Code | Assembly Rendition |
|----------|-----------------------------|-----------------------|
| 8048365: | 5b | pop %ebx |
| 8048366: | 81 c3 <u>ab 12 00 00</u> | add \$0x12ab,%ebx |
| 804836c: | 83 bb 28 00 <u>00 00 00</u> | cmpl \$0x0,0x28(%ebx) |

Deciphering Numbers

Value:

Pad to 32 bits:

Split into bytes:

Reverse:

0x12ab

0x000012ab

00 00 12 ab

ab 12 00 00