CS 105
"Tour of the Black Holes of Computing!"

Computer Systems Introduction

Geoff Kuenning
Spring 2022

Topics:
- Class Introduction
- Data Representation

Living on Zoom

Not again! Let's hope it's brief…

I try to keep the sessions as free as possible
- No waiting rooms so you can join early and talk to each other

PowerPoint and PDF versions of slides will be pre-posted
- Use them to take notes if you wish
- See calendar page on class site: https://www.cs.hmc.edu/~geoff/cs105
- Remind me at beginning of class if I forget (sometimes I do)

Please be visible and interactive!
- Sign in with your actual name
- Zoom discourages questions and chatting
  - Please fight that tendency
  - Avoid all those tempting distractions
- Seeing you helps me teach better
  - I know some of you have bandwidth problems, but…

Course Theme

- Abstraction is good, but don't forget reality!

Many CS Courses emphasize abstraction
- Abstract data types
- Asymptotic analysis

These abstractions have limits
- Especially in the presence of bugs
- Need to understand underlying implementations

Useful outcomes
- Become more effective programmers
  - Able to find and eliminate bugs efficiently
  - Able to tune program performance
- Prepare for later "systems" classes in CS
  - Compilers, Operating Systems, File Systems, Computer Architecture, Robotics, etc.

Textbooks

Randal E. Bryant and David R. O'Hallaron,
- "Computer Systems: A Programmer's Perspective", 3rd Edition, Prentice Hall, 2015.
- Note: The "International Edition" is different. Watch out!

Brian Kernighan and Dennis Ritchie,
- "The C Programming Language, Second Edition", Prentice Hall, 1988

Larry Miller and Alex Quilici
- The Joy of C, Wiley, 1997

Syllabus

- Syllabus on Web: https://www.cs.hmc.edu/~geoff/cs105
- Calendar defines due dates
  - Also has links to slides and labs
- Labs: cs105submit for some, others have specific directions

---

Notes:

Work groups

- You must work in pairs on all labs
- Honor-code violation to work without your partner!
- Corollary: showing up late doesn't harm only you

Handins

- Check calendar for due dates
- Electronic submissions only

Grading Characteristics

- Lab scores tend to be high
  - Serious handicap if you don't hand a lab in
- Tests & quizzes typically have a wider range of scores
  - I.e., they're have major effect on your grade
    - » …but not the ONLY one
- Do your share of lab work and reading, or bomb tests
- Do practice problems in book

Facilities

Assignments will use Intel computer systems

- Not all machines are created alike
  - Performance varies (and matters sometimes in 105)
  - Security settings vary and can matter
- Wilkes: x86/Linux specifically set up for this class
- Log in on a lab Mac, then ssh to Wilkes
  - If you want fancy programs, start X11 first
  - Directories are cross-mounted, so you can edit on Knuth or your Mac, and Wilkes will see your files
- …or ssh into Wilkes from wherever you are
- All programs *must* run on Wilkes: we grade there
- Have lecture slides (and textbook) available when working on labs!

CS 105

"Tour of the Black Holes of Computing"

Bits, Bytes, Integers

Topics

- 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

Everything is bits

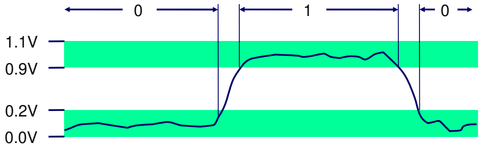
Each bit is 0 or 1

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
- 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

Example Data Sizes

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

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

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Or

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

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

Not

- ~A = 1 when A=0

A	~A
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

Fancier Boolean Algebra

- What is A&~B?

A B	A&~B
0 0	0
0 1	0
1 0	1
1 1	0

- How about ~(~A&~B)?

A B	~A	~B	(~A&~B)	~(~A&~B)	A|B
0 0	1	1	1	0	0
0 1	1	0	0	1	1
1 0	0	1	0	1	1
1 1	0	0	0	1	1

Grouped Boolean Operations

Operate on bit vectors

- Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

All of the properties of Boolean algebra apply

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
- Operations applied bit-wise

Examples (char data type)

- ~0x41 → 0xBE   B E
  - ~01000001_2 → 10111110_2
- ~0x00 → 0xFF
  - ~00000000_2 → 11111111_2
- 0x69 & 0x55 → 0x41
  - 01101001_2 & 01010101_2 → 01000001_2
- 0x69 | 0x55 → 0x7D
  - 01101001_2 | 01010101_2 → 01111101_2

Example: Representing & Manipulating Sets

Representation

- Width w bit vector represents subsets of {0, …, 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 }

Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as “False”
  - Anything nonzero seen 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 != 0 && *p (unreadably avoids null pointer access)

Shift Operations

Left Shift: `x << y`

- Shift bit-vector `x` left `y` positions
  - » Throw away extra bits on left
  - Fill with 0’s on right

Right Shift: `x >> 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 ≥ word size

Argument `x`	`01100010`
`<< 3`	`00010000`
Log. `>> 2`	`00011000`
Arith. `>> 2`	`00011000`

Argument `x`	`10100010`
`<< 3`	`00010000`
Log. `>> 2`	`00101000`
Arith. `>> 2`	`11101000`

C Puzzles

- Taken from old exams
- Assume machine with 32-bit word size, two’s complement integers
- For each of the following C expressions, either:
  - Argue that it is true for all argument values, or
  - Give example where it is not true

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`

Encoding Integers

Unsigned

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

Two’s Complement

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

Sign Bit

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

- C `short` (2 bytes long)

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

↑ 5 ← F

Sign Bit

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

Encoding Integers (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
Sum		**15213**		**-15213**

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
`UMax`	65535	`FF FF`	`11111111 11111111`
`TMax`	32767	`7F FF`	`01111111 11111111`
`TMin`	-32768	`80 00`	`10000000 00000000`
`-1`	-1	`FF FF`	`11111111 11111111`
`0`	0	`00 00`	`00000000 00000000`

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	**255**	**65,535**	**4,294,967,295**	**18,446,744,073,709,551,615**
TMax	**127**	**32,767**	**2,147,483,647**	**9,223,372,036,854,775,807**
TMin	**-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>`
  - K&R Appendix B11
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform-specific

A Critical Detail

No self-identifying data

- Looking at a bunch of bits doesn't tell you what they mean
- Could be signed, unsigned integer
- Could be floating-point number
- Could be part of a string

Only the program (instructions) knows for sure!

- (To be fair, experienced humans can make good guesses—see Lab 2)

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

discontinuity

Equivalence

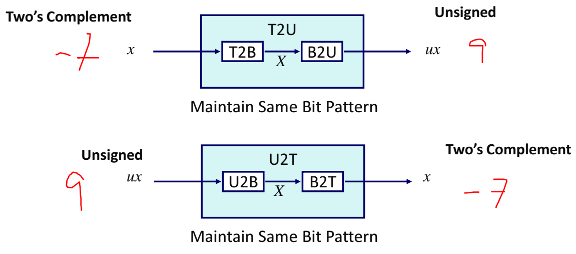
- Same encodings for nonnegative values

Uniqueness

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

Mapping Between Signed & Unsigned

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



Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	–7	9
1010	–6	10
1011	–5	11
1100	–4	12
1101	–3	13
1110	–2	14
1111	–1	15

→ T2U →

← U2T ←

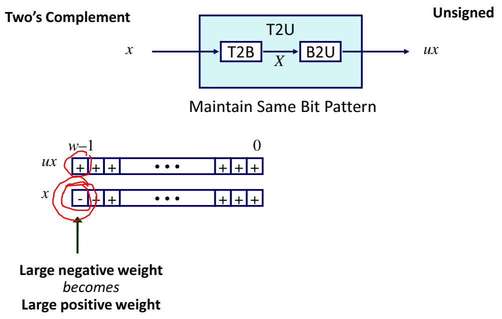
Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	–7	9
1010	–6	10
1011	–5	11
1100	–4	12
1101	–3	13
1110	–2	14
1111	–1	15

= ⟷ (0–7)

+/- 16 ⟷ (8–15)

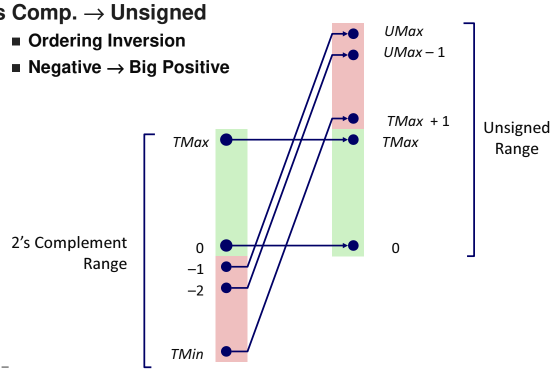
Relation Between Signed & Unsigned



Conversion Visualized

2's Comp. → Unsigned

- **Ordering Inversion**
- **Negative → Big Positive**



Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

```
short ~~int~~          x =  15213;
unsigned short ~~int~~ ux = (unsigned short) x;
short ~~int~~          y = -15213;
unsigned short ~~int~~ uy = (unsigned short) y;
```

Resulting Value

- **No change in bit representation**
- **Nonnegative values unchanged**
  - *ux* = 15213
- **Negative values change into (large) positive values!**
  - *uy* = 50323

Signed vs. Unsigned in C

Integer Constants

- **By default are considered to be signed integers**
  - Exception: unsigned, if too big to be signed but fit in unsigned
- **Unsigned if have "U" as suffix**

`0U, 4294967259u` ⇦ lowercase is better here

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;
```

fro(-3)

Casting Surprises

Expression Evaluation

- If you mix unsigned and signed in single expression, signed values are implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$

Constant$_1$	Constant$_2$	Relation Evaluation
`0`	`0u`	
`-1`	`0`	
`-1`	`0u`	
`2147483647`	`-2147483648`	
`2147483647u`	`-2147483648`	
`-1`	`-2`	
`(unsigned)-1`	`-2`	
`2147483647`	`2147483648u`	
`2147483647`	`(int)2147483648u`	

Summary: Casting Signed ↔ Unsigned: Basic Rules

Bit pattern is maintained—but reinterpreted

Can have unexpected effects: adding or subtracting 2^w

In expression containing signed and unsigned int:

- `int` is cast to `unsigned`!!

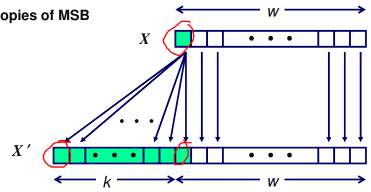
Sign Extension

Task:

- Given *w*-bit signed integer *x*
- Convert it to *w*+*k*-bit integer with same value

Rule:

- Make *k* copies of sign bit:
- $X' = \underbrace{x_{w-1},\ldots,x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2},\ldots, x_0$



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

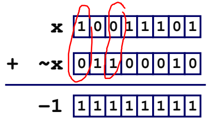
Negating with Complement & Increment

Claim: Following holds for 2's complement

`~x + 1 == -x`

Complement

- **Observation: `~x + x == 1111…11`$_2$ `== -1`**



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

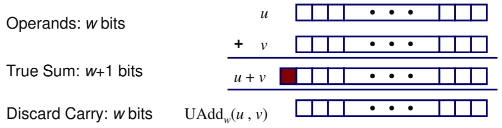
Increment

- ~x + ~~x~~ + (~~-x~~ + 1) == ~~-1~~ + (-x + ~~1~~)
- ~x + 1 == -x

Warning: Be cautious treating `int`'s as integers

- **OK here (associativity and commutativity hold)**

Unsigned Addition



Operands: *w* bits
u
$+\ v$
True Sum: *w*+1 bits
$u + v$
Discard Carry: *w* bits
$\mathrm{UAdd}_w(u, v)$

Standard Addition Function

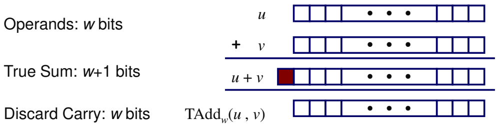
- **Ignores carry output**

Implements Modular Arithmetic

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

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

Two's-Complement Addition



Operands: *w* bits
u
$+\ v$
True Sum: *w*+1 bits
$u + v$
Discard Carry: *w* bits
$\mathrm{TAdd}_w(u, v)$

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
```

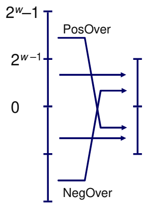
- **Will give `s == t`**

Detecting 2's-Complement Overflow

Task

- **Given** $s = \mathrm{TAdd}_w(u, v)$
- **Determine if** $s = \mathrm{Add}_w(u, v)$
- **Example**

```
int s, u, v;
s = u + v;
```



Claim

- **Overflow iff either:**
  - ***u*, *v* < 0, *s* ≥ 0 (NegOver)**
  - ***u*, *v* ≥ 0, *s* < 0 (PosOver)**

A Fun Fact

Official C standard says overflow is “undefined”

- Intention was to let machine define what happens

Recently ISO C committee and compiler writers have decided “undefined” means “compiler gets to choose”

- Can generate 0, biggest integer, or anything else
- Or if compiler is sure it’ll overflow, it can optimize out completely
- Generates faster—but wrong!—code
- This can introduce some lovely bugs (e.g., it’s tricky to check for overflow)

Fight between compiler community and OS/security community over this issue

---

Multiplication

Computing exact product of w-bit numbers x, y

- Either signed or unsigned

Ranges

- **Unsigned:** $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- **Two’s complement min:** $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w$–1 bits (including 1 for sign)
- **Two’s complement max:** $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - Up to $2w$ bits, but only for $(TMin_w)^2$

Maintaining exact results

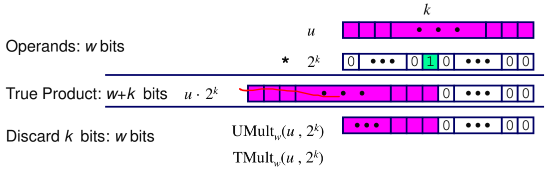
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary-precision” arithmetic packages

---

Power-of-2 Multiply by Shifting

Operation

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



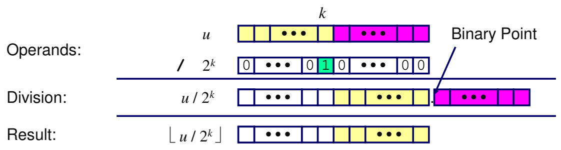
Examples

- `u << 3 == u * 8` 1 clock
- `u << 5 - u << 3 == u * 24` 2+1 clock
- Most machines shift and add much faster than multiply
  - Compiler generates this code automatically 3-4 clocks

Unsigned Power-of-2 Divide by Shifting

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`	**`0`**`0011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	**`0000`**`0011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	**`00000000`** `00111011`

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 2^w
- Signed: modified addition mod 2^w (result in proper range)
  - Mathematical addition + possible addition or subtraction of 2^w

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 range -2^{w-1} to $2^{w-1}-1$)

Why Should I Use Unsigned?

Don't **use without understanding implications**

- 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)
  . . .
```

Counting Down with Unsigned

Proper way to use unsigned as loop index

```
unsigned i;
for (i = cnt - 2; i < cnt; i--)
  a[i] += a[i + 1];
```

See Robert Seacord, *Secure Coding in C and C++*

- C Standard guarantees unsigned addition will behave like modular arithmetic
  - 0 – 1 → *UMax*

Even better

```
#include <sys/types.h>
size_t i;
for (i = cnt - 2; i < cnt; i--)
  a[i] += a[i + 1];
```

- `Data type size_t` is unsigned value with length = word size
- Code will work even if `cnt` = *UMax*
- But what if `cnt` is signed and < 0?

Why Should I Use Unsigned? (cont.)

Do **Use When Performing Modular Arithmetic**

- Multiprecision arithmetic

Do **Use When Using Bits to Represent Sets**

- Logical right shift, no sign extension

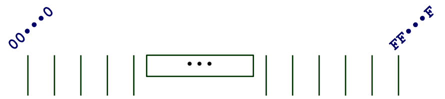
Do **Use for Very Large Arrays**

- Signed index doesn't have range

Do **Use for Bit Fields**

- Need Logical Right Shift

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 space 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

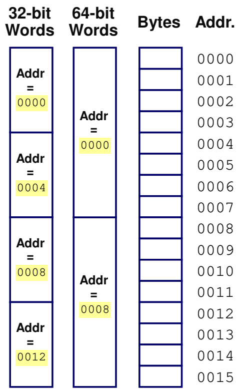
Any given computer has a "word size"

- Nominal size of integer-valued data
  - …and of addresses
- Until about 2010, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB (2^{32} bytes)
- Now most "real" machines (even phones) 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)



Byte Ordering

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

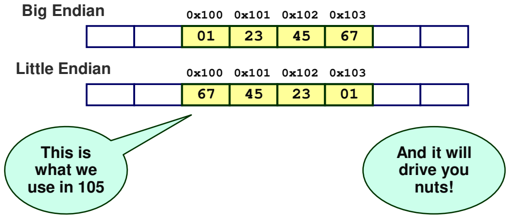
Conventions

- Big Endian: Sun, PPC Mac, Internet
  - Most significant byte has lowest 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



			0x100	0x101	0x102	0x103		
Big Endian			01	23	45	67		
Little Endian			67	45	23	01		

This is what we use in 105

And it will drive you nuts!

Representing Strings

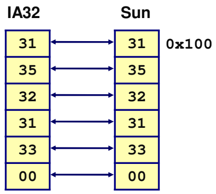
`char s[6] = "15213";`

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 (yay!)



IA32	Sun	
31	31	0x100
35	35	
32	32	
31	31	
33	33	
00	00	