Bits, Bytes, and Integers (1-2)

15-213/18-243: Introduction to Computer Systems
2nd Lecture, 13 January 2011

Instructors:

Gregory Kesden and Anthony Rowe

Last Time: Course Overview

- **Course Theme:**

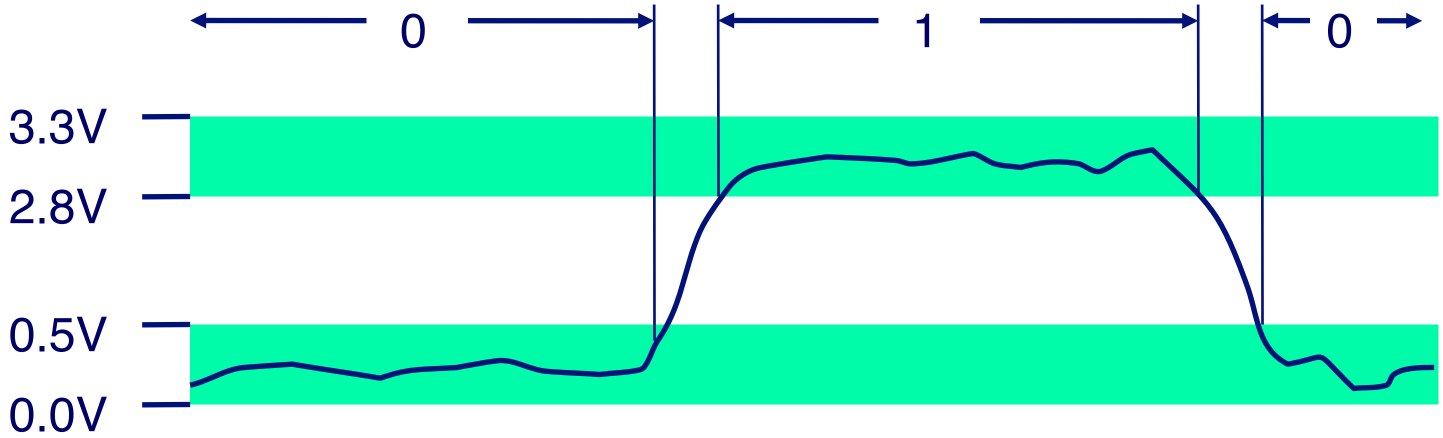
Abstraction Is Good But Don't Forget Reality

- **5 Great Realities**
  - Ints are not Integers, Floats are not Reals
  - You've Got to Know Assembly
  - Memory Matters
  - There's more to performance than asymptotic complexity
  - Computers do more than execute programs
- **Administrative / Logistics details**

Today: Bits, Bytes, and Integers (1-2)

- **Representing information as bits**
- Bit-level manipulations
- Summary

Binary Representations



Encoding Byte Values

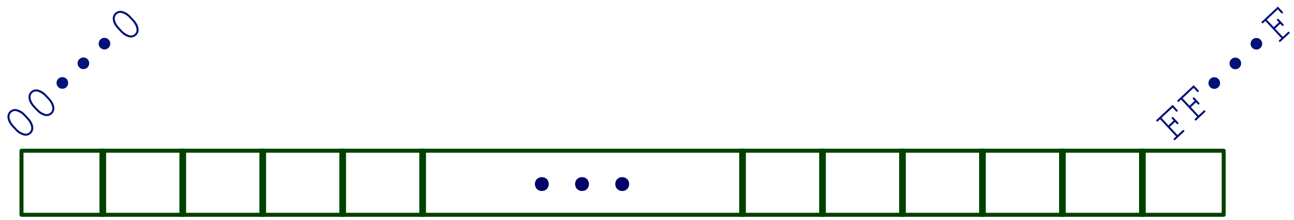
- **Byte = 8 bits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

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

Literary Hex

- **Common 8-byte hex fillers:**
  - 0xdeadbeef
  - 0xc0ffeeee
  - Can you think of other 8-byte fillers?

Byte-Oriented Memory Organization



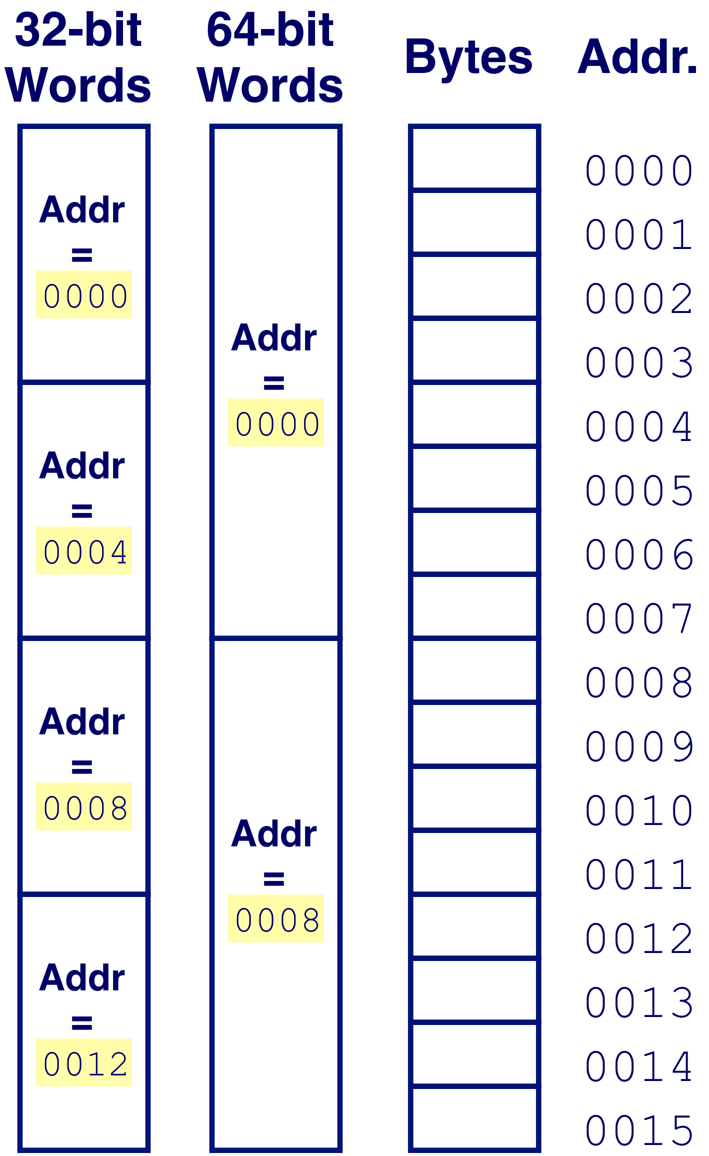
- **Programs Refer to Virtual Addresses**
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of 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 ≈ 1.8×10^{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

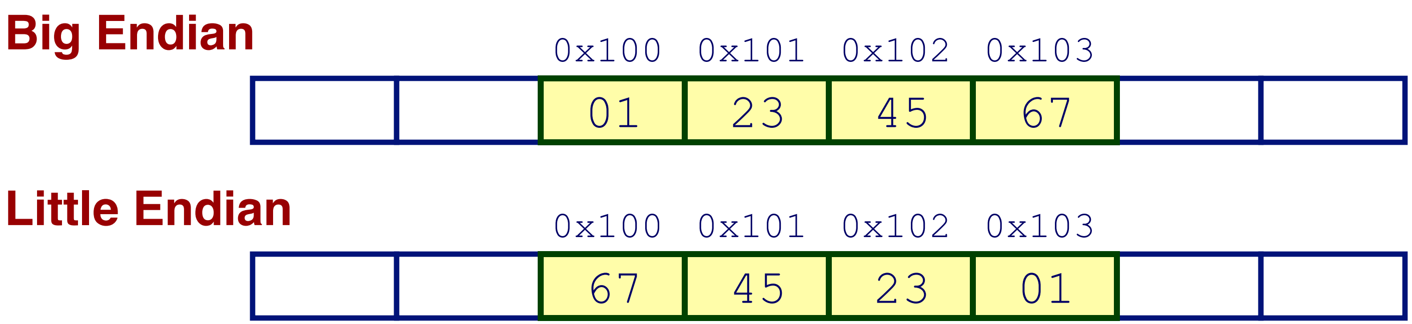
C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

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



Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example Fragment**

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

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00

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));
```

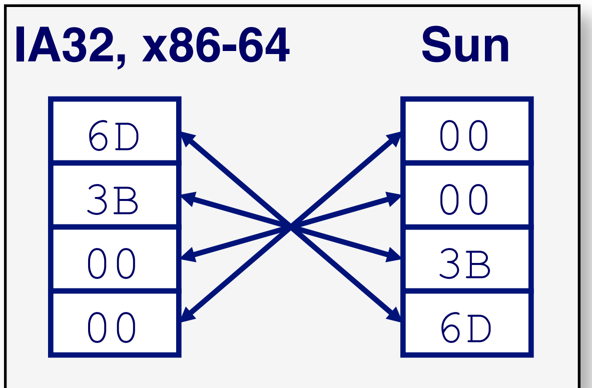
Result (Linux):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

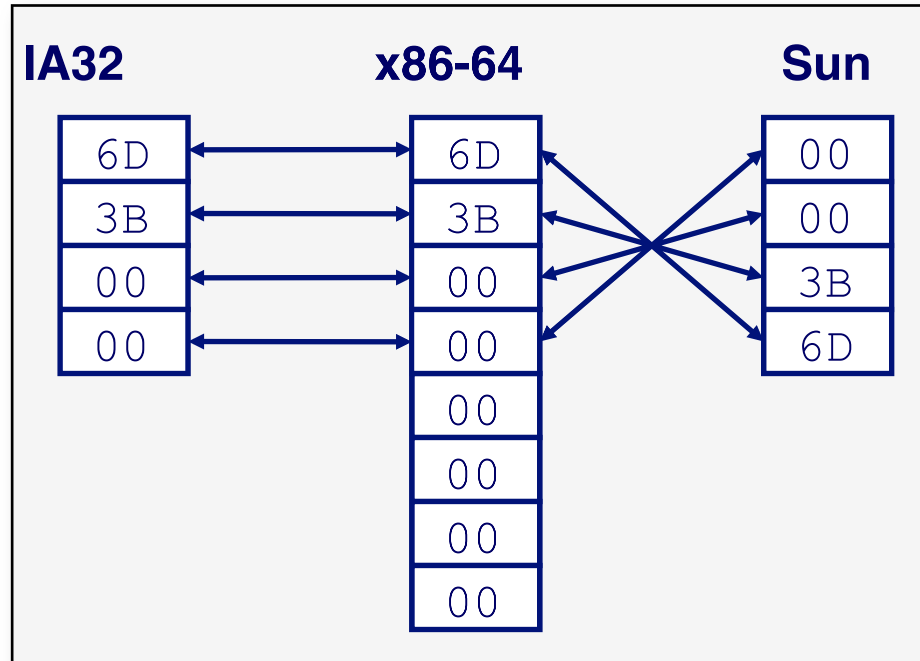
Representing Integers

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

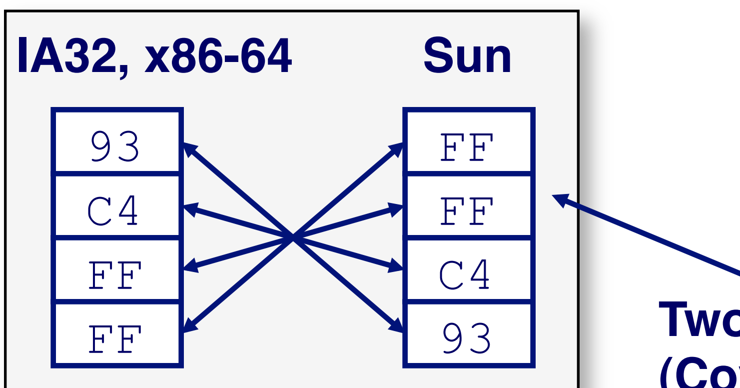
```
int A = 15213;
```




```
long int C = 15213;
```




```
int B = -15213;
```



Two's complement representation (Covered later)

Representing Pointers

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

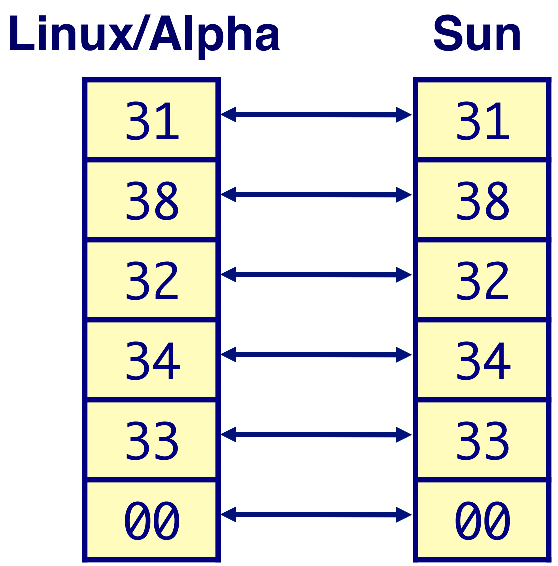
Sun	IA32	x86-64
EF	D4	0C
FF	F8	89
FB	FF	EC
2C	BF	FF
		FF
		7F
		00
		00

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



Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Summary

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

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

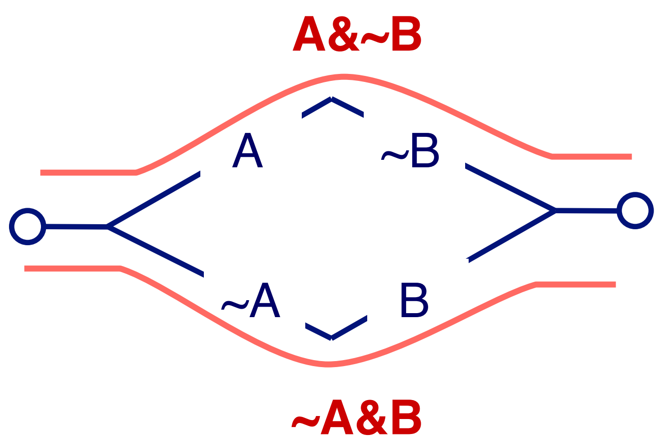
Exclusive-Or (Xor)

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

^	0	1
0	0	1
1	1	0

Application of Boolean Algebra

- **Applied to Digital Systems by Claude Shannon**
  - 1937 MIT Master's Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

Boolean Algebra ≈ Integer Ring

- *Commutativity*

A | B = B | A	A + B = B + A
A & B = B & A	A * B = B * A

- *Associativity*

(A | B) | C = A | (B | C)	(A + B) + C = A + (B + C)
(A & B) & C = A & (B & C)	(A * B) * C = A * (B * C)

- *Product distributes over sum*

A & (B | C) = (A & B) | (A & C)	A * (B + C) = A * B + B * C

- *Sum and product identities*

A | 0 = A	A + 0 = A
A & 1 = A	A * 1 = A

- *Zero is product annihilator*

A & 0 = 0	A * 0 = 0

- *Cancellation of negation*

~ (~ A) = A	– (– A) = A

Boolean Algebra ≠ Integer Ring

- Boolean: *Sum distributes over product*

  A | (B & C) = (A | B) & (A | C)   A + (B * C) ≠ (A + B) * (A + C)

- Boolean: *Idempotency*

  A | A = A   A + A ≠ A

  - "A is true" or "A is true" = "A is true"

  A & A = A   A * A ≠ A

- Boolean: *Absorption*

  A | (A & B) = A   A + (A * B) ≠ A

  - "A is true" or "A is true and B is true" = "A is true"

  A & (A | B) = A   A * (A + B) ≠ A

- Boolean: *Laws of Complements*

  A | ~A = 1   A + –A ≠ 1

  - "A is true" or "A is false"

- Ring: *Every element has additive inverse*

  A | ~A ≠ 0   A + –A = 0

Relations Between Operations

DeMorgan's Laws

- Express & in terms of |, and vice-versa
  - A & B = ~(~A | ~B)
    - » A and B are true if and only if neither A nor B is false
  - A | B = ~(~A & ~B)
    - » A or B are true if and only if A and B are not both false

Exclusive-Or using Inclusive Or

- A ^ B = (~A & B) | (A & ~B)
  - » Exactly one of A and B is true
- A ^ B = (A | B) & ~(A & B)
  - » Either A is true, or B is true, but not both

General Boolean Algebras

- **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**

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 }

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
- **Examples (Char data type)**
  - `~0x41` → `0xBE`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00` → `0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55` → `0x41`
    - $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55` → `0x7D`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

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 << 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 right
- **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`

Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse

  A ^ A = 0

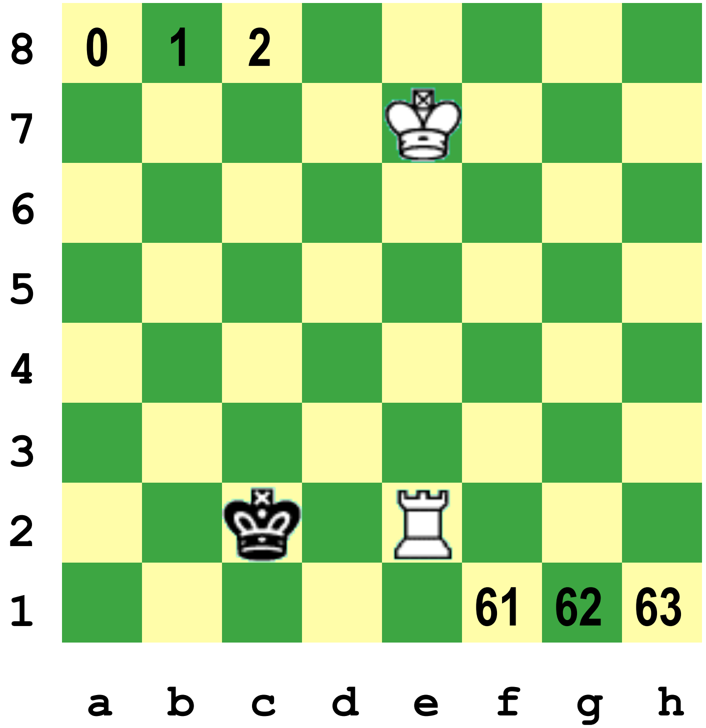
```
void funny(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```

	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A
3	(A^B)^A = B	A
End	B	A

More Fun with Bitvectors

Bit-board representation of chess position:

```
unsigned long long blk_king, wht_king, wht_rook_mv2,…;
```




```
wht_king      = 0x0000000000001000ull;
blk_king      = 0x0004000000000000ull;
wht_rook_mv2 = 0x10ef101010101010ull;
...
/*
 * Is black king under attach from
 * white rook ?
 */
if (blk_king & wht_rook_mv2)
       printf("Yes\n");
```

More Bitvector Magic

- **Count the number of 1's in a word**
  - Naïve Approach

```
int bitcount(unsigned int n)
{
   int count=0;
   while(n |=0 )
   {
       count += n & 1;
       n >>=1;

   }
   return count;
}
```

More Bitvector Magic

- **Count the number of 1's in a word**
  - Divide-and-conquer Approach

```
int bitcount(unsigned int n)
{
   n = (n & 0x55555555) + ((n >> 1) & 0x55555555);
   n = (n & 0x33333333) + ((n >> 2) & 0x33333333);
   n = (n & 0x0f0f0f0f) + ((n >> 4) & 0x0f0f0f0f);
   n = (n & 0x00ff00ff) + ((n >> 8) & 0x00ff00ff);
   n = (n & 0x0000ffff) + ((n >> 16) & 0x0000ffff);
   return (n & 0x0000003f);
}
```

More Bitvector Magic

- **Count the number of 1's in a word**
  - MIT Hackmem 169:

```
int bitcount(unsigned int n)
{
    unsigned int tmp;

    tmp = n - ((n >> 1) & 033333333333)
            - ((n >> 2) & 011111111111);
    return ((tmp + (tmp >> 3)) & 030707070707)%63;
}
```

More Bitvector Uses

Representation of small sets

Representation of polynomials:

- Important for error correcting codes
- Arithmetic over finite fields, say GF(2^n)
- Example 0x15213 : $x^{16} + x^{14} + x^{12} + x^9 + x^4 + x + 1$

Representation of graphs

- A '1' represents the presence of an edge

Representation of bitmap images, icons, cursors, …

- Exclusive-or cursor patent

Representation of Boolean expressions and logic circuits

Today: Bits, Bytes, and Integers (1-2)

- Representing information as bits
- Bit-level manipulations
- Summary

Summary

It's All About Bits & Bytes

- Numbers
- Programs
- Text

Different Machines Follow Different Conventions for

- Word size
- Byte ordering
- Representations

Boolean Algebra is the Mathematical Basis

- Basic form encodes "false" as 0, "true" as 1
- General form like bit-level operations in C
  - Good for representing & manipulating sets