CS429: Computer Organization and Architecture

Bits and Bytes

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: February 3, 2020 at 14:57

Topics of this Slideset

There are 10 kinds of people in the world: those who understand binary, and those who don't!

- Why bits?
- Representing information as bits
  - Binary and hexadecimal
  - Byte representations : numbers, characters, strings, instructions, etc.
- Bit level manipulations
  - Boolean algebra
  - C constructs

It's Bits All the Way Down

Great Reality 7: Whatever you plan to store on a computer ultimately has to be represented as a finite collection of bits.

That's true whether it's integers, reals, characters, strings, data structures, instructions, programs, pictures, videos, etc.

That really means that only *discrete* quantities can be represented exactly. Non-discrete (continuous) quantities have to be approximated.

Why Binary? Why Not Decimal?

Base 10 Number Representation.

- Fingers *are* called as "digits" for a reason.
- Natural representation for financial transactions. Floating point number cannot exactly represent \$1.20.

- Even carries through in scientific notation: 1.5213×10^4

If we lived in Homer Simpson's world, we'd all use octal!

Why Not Base 10?

Implementing Electronically

- 10 different values are hard to store. ENIAC (First electronic computer) used 10 vacuum tubes / digits
- They're hard to transmit. Need high precision to encode 10 signal levels on single wire.
- Messy to implement digital logic functions: addition, multiplication, etc.

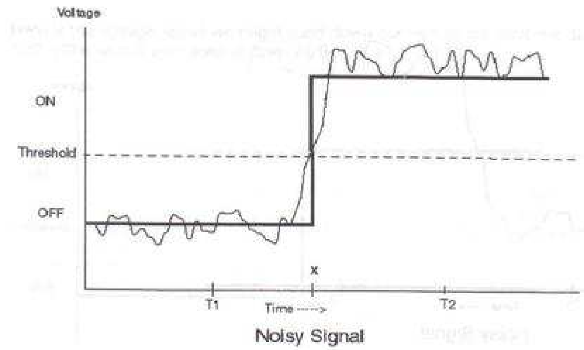
Even Bits are an Abstraction!

Base 2 Number Representation

- Represent 15213_{10} as 11101101101101_2
- Represent 1.20_{10} as $1.0011001100110011[0011]_2$
- Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Electronic Implementation

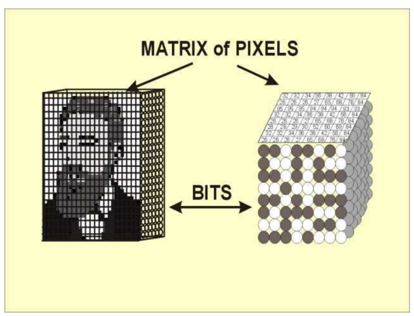
- Easy to store bits with bistable elements.
- Reliably transmitted on noisy and inaccurate wires.



Representing Data

To store data of type X, someone had to invent a mapping from items of type X to bit strings. That's the *representation mapping*.

In a sense the representation is *arbitrary*. The representation is just a *mapping from the domain onto a finite set of bit strings*.



The mapping should be one-one, but not necessarily onto. But some representations are better than others. Why would that be? Hint: what operations do you want to support?

Some Representations

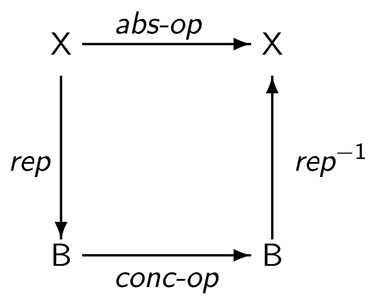
Suppose you want to represent the finite set of natural numbers $[0 \ldots 7]$ as 3-bit strings. Would 2-bit strings work?

Dec.	Rep1	Rep2
0	101	000
1	011	001
2	111	010
3	000	011
4	110	100
5	010	101
6	001	110
7	100	111

Why is one of these representations is "better" than the other? Hint: How would you do addition using Rep1?

Representing Data

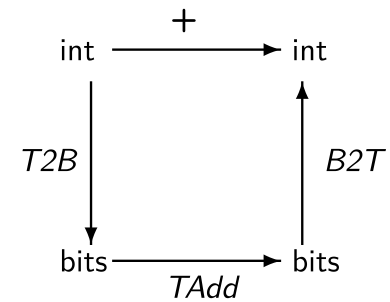
A "good" mapping will map X data onto bit strings (B) in a way that makes it easy to compute common operations on that data. I.e., the following diagram should *commute*, for a reasonable choice of *conc-op*.



Representing Data: Integer Addition

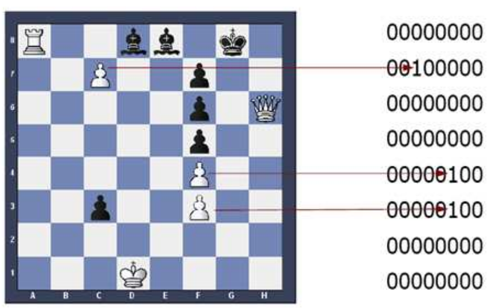
```
int x;
int y;
...
t = x + y;
```

To carry out any operation at the C level means converting the data into bit strings, and implementing an operation on the bit strings that has the "intended effect" under the mapping.



Representing Data

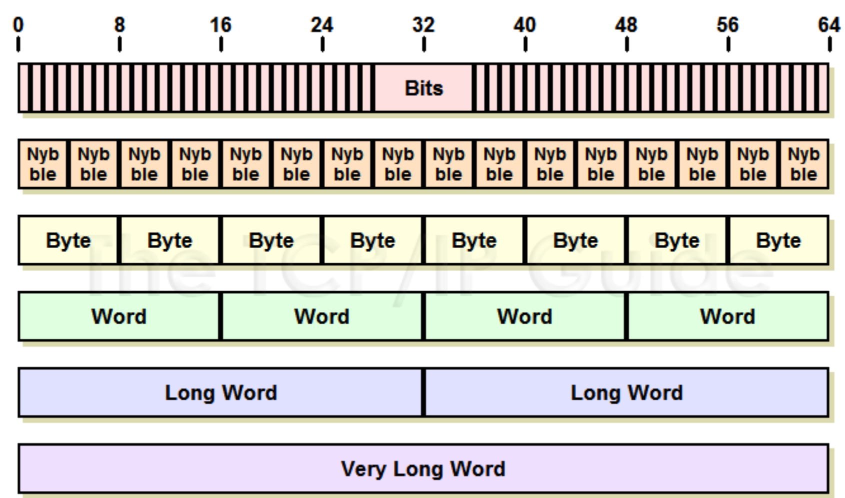
Important Fact 1: If you are going to represent any type in k bits, you can only represent 2^k different values.



Important Fact 2: The same bit string can represent an integer (signed or unsigned), float, character string, list of instructions, address, etc. depending on the context. How do you represent the context in C?

Bits Aren't So Convenient

Since it's tedious always to think in terms of bits, we group them together into larger units. *Sizes of these units depends on the architecture / language.*



Bytes

Byte = 8 bits
Which can be represented in various forms:

- 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` or `0xfa1d37b`

BTW: one hexadecimal digit represents 4 bits *(one nybble)*.

Hex	Dec	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

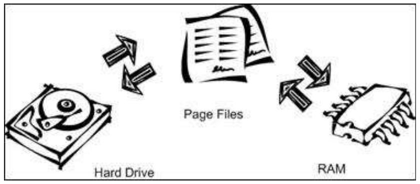
Memory as Array

Computer Address	Computer Content	Programmers Name	Programmers Type	Programmers Value
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

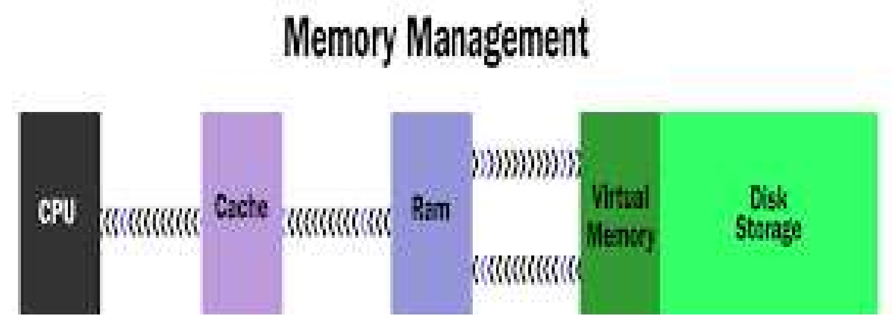
Note: this picture is appropriate for a 32-bit, big endian machine.
How did I know that?

Byte-Oriented Memory Organization



- Conceptually, memory is a very large array of bytes.
- Actually, it's implemented with hierarchy of different memory types.
  - SRAM, DRAM, disk.
  - The OS only allocates storage for regions actually used by program.
- In Unix and Windows, address space private to particular "process."
  - Encapsulates the program being executed.
  - Program can clobber its own data, but not that of others.

Byte-Oriented Memory Organization



Compiler and Run-Time System Control Allocation

- Where different program objects should be stored.
- Multiple storage mechanisms: static, stack, and heap.
- In any case, all allocation within single virtual address space.

Machine Words

Machines generally have a specific "word size."

- It's the nominal size of addresses on the machine.
- Most current machines run 64-bit software (8 bytes).
  - 32-bit software limits addresses to 4GB.
  - Becoming too small for memory-intensive applications.
- All x86 current hardware systems are 64 bits (8 bytes). Potentially address around $1.8X10^{19}$ bytes.
- Machines support multiple data formats.
  - Fractions or multiples of word size.
  - Always integral number of bytes.
- X86-hardware systems operate in 16, 32, and 64-bit modes.
  - Initially starts in 286 mode, which is 16-bit.
  - Under programmer control, 32- and 64-bit modes are enabled.

Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Which is the address of the *first* byte in word.
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit).
- Addresses of multi-byte data items are typically *aligned* according to the size of the data.

32-bit words	64-bit words	bytes	addr.
Addr: 0000	Addr: 0000		0000
			0001
			0002
			0003
Addr: 0004			0004
			0005
			0006
			0007
Addr: 0008	Addr: 0008		0008
			0009
			0010
			0011
Addr: 0012			0012
			0013
			0014
			0015

Data Representations

Sizes of C Objects (in Bytes)

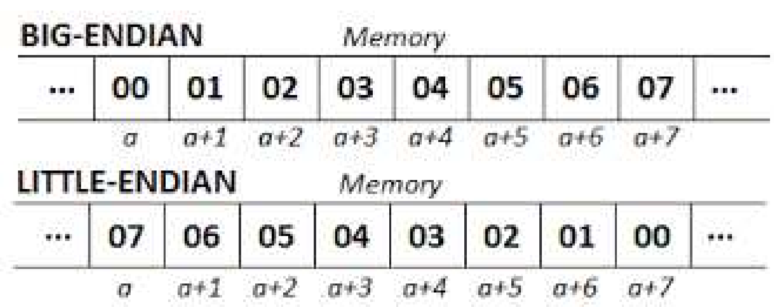
C Data Type	Alpha	Intel x86	AMD 64
int	4	4	4
long int	8	8	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	8	10/12
char *	8	8	8
other pointer	8	8	8

The *integer data types* (`int`, `long int`, `short`, `char`) can all be either signed or unsigned.

Byte Ordering

How should bytes within a multi-byte data item be ordered in memory?

Given 64-bit hex value 0x0001020304050607, it is common to store this in memory in one of two formats: big endian or little endian.



Note that "endian-ness" only applies to multi-byte *primitive* data items, not to strings, arrays, or structs.

Byte Ordering Examples

Big Endian: Most significant byte has lowest (first) address.

Little Endian: Least significant byte has lowest address.

Example:

- Int variable x has 4-byte representation **0x01234567**.
- Address given by `&x` is `0x100`

Big Endian:

Address:			0x100	0x101	0x102	0x103		
Value:			01	23	45	67		

Little Endian:

Address:			0x100	0x101	0x102	0x103		
Value:			67	45	23	01		

Byte Ordering

Conventions

- Sun, PowerPC MacIntosh computers are "big endian" machines: most significant byte has lowest (first) address.
- Alpha, Intel MacIntosh, x86s are "little endian" machines: least significant byte has lowest address.
- ARM processor offers support for big endian, but mainly they are used in their default, little endian configuration.
- There are many (hundreds) of microcontrollers, so check before you start programming!

Reading Little Endian Listings

Disassembly

- Yields textual representation of binary machine code.
- Generated by program that reads the machine code.

Example Fragment (IA32):

```
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        cmpl $0x0,0x28(%ebx)
```

Deciphering Numbers: Consider the value `0x12ab` in the second line of code:

- Pad to 4 bytes: `0x000012ab`
- Split into bytes: `00 00 12 ab`
- Make little endian: `ab 12 00 00`

Examining Data Representations

Code to Print Byte Representations of Data

Casting a pointer to unsigned `char *` creates a 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;
0x7fff90c56c7c 0x6d
0x7fff90c56c7d 0x3b
0x7fff90c56c7e 0x00
0x7fff90c56c7f 0x00
```


Representing Integers

```
int A = 15213;
int B = −15213;
long int C = 15213;
```

$$15213_{10} = 0011101101101101_2 = 3B6D_{16}$$

	Linux (little endian)	Alpha (little endian)	Sun (big endian)
A	6D 3B 00 00	6D 3B 00 00	00 00 3B 6D
B	93 C4 FF FF	93 C4 FF FF	FF FF C4 93
C	6D 3B 00 00 00 00 00 00	6D 3B 00 00 00 00 00 00	00 00 00 00 00 00 3B 6D

We'll cover the representation of negatives later.

Representing Pointers

```
int B = −15213;
int *P = &B;
```

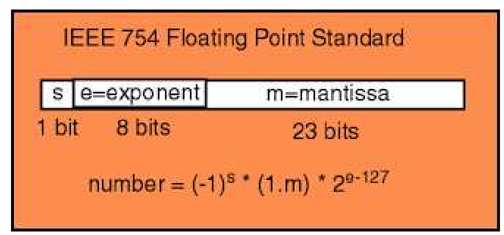
Linux Address:
Hex: BFFFF8D4AFBB4CD0
In memory: D0 4C BB AF D4 F8 FF BF

Sun Address:
Hex: EFFFFFB2CAA2C15C0
In Memory: EF FF FB 2C AA 2C 15 C0

Pointer values generally are not predictable. Different compilers and machines assign different locations.

Representing Floats

All modern machines implement the IEEE Floating Point standard. This means that it is consistent across all machines.




```
float F = 15213.0;
```

Binary: 01000110011011011011010000000000
Hex: 466DB400
In Memory (Linux/Alpha): 00 B4 6D 46
In Memory (Sun): 46 6D B4 00

Note that it's not the same as the `int` representation, but you can see that the int is in there, if you know where to look.

Representing Strings

0	1	2	3	4	5
H	e	l	l	o	\0

- Strings are represented by a sequence of characters.
- Each character is encoded in ASCII format.
  - Standard 7-bit encoding of character set.
  - Other encodings exist, but are less common.
- Strings should be null-terminated. That is, the final character has ASCII code 0. I.e., a string of k chars requires $k+1$ bytes.

Compatibility

- *Byte ordering (endian-ness) is not an issue* since the data are single byte quantities.
- Text files are generally platform independent, except for different conventions of line break character(s).

Machine Level Code Representation

Encode Program as Sequence of Instructions

- Each simple operation
  - Arithmetic operation
  - Read or write memory
  - Conditional branch
- Instructions are encoded as sequences of bytes.
  - Alpha, Sun, PowerPC Mac use 4 byte instructions (Reduced Instruction Set Computer" (RISC)).
  - PC's and Intel Mac's use variable length instructions (Complex Instruction Set Computer (CISC)).
- Different instruction types and encodings for different machines.
- Most code is not binary compatible.

Remember: Programs are byte sequences too!

Representing Instructions

```
int sum( int x, int y ) {
   return x + y;
}
```

For this example, Alpha and Sun use two 4-byte instructions. They use differing numbers of instructions in other cases.

PC uses 7 instructions with lengths 1, 2, and 3 bytes. Windows and Linux are not fully compatible.

Different machines typically use different instuctions and encodings.

Instruction sequence for sum program:

Alpha: 00 00 30 42 01 80 FA 68
Sun: 81 C3 E0 08 90 02 00 09
PC: 55 89 E5 8B 45 OC 03 45 08 89 EC 5D C3

Assembly vs. Machine Code

Machine code bytes	Assembly language statements
	`foo:`
B8 22 11 00 FF	`movl $0xFF001122, %eax`
01 CA	`addl %ecx, %edx`
31 F6	`xorl %esi, %esi`
53	`pushl %ebx`
8B 5C 24 04	`movl 4(%esp), %ebx`
8D 34 48	`leal (%eax, %ecx, 2), %esi`
39 C3	`cmpl %eax, %ebx`
72 EB	`jnae foo`
C3	`retl`

Instruction stream
B8 22 11 00 FF 01 CA 31 F6 53 8B 5C 24 04 8D 34 48 39 C3 72 EB C3

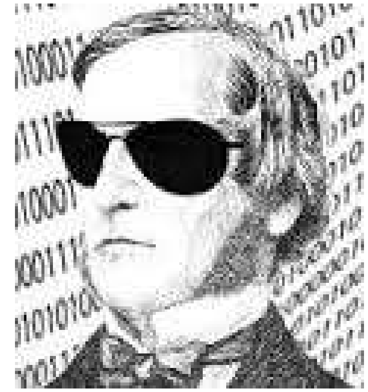
And So On

Recall **Great Reality 7:** Whatever you plan to store on a computer ultimately has to be represented as a finite collection of bits.

That's true whether it's integers, reals, characters, strings, data structures, instructions, programs, pictures, videos, audio files, etc. Anything!

Boolean Algebra

Developed by George Boole in the 19th century, Boolean algebra is the algebraic representation of logic. We encode "True" as 1 and "False" as 0.



Boolean Algebra

And: `A & B` = 1 when both A = 1 and B = 1.

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	|
0	0	0
0	1	1
1	0	1
1	1	1

Not: `~A` = 1 when A = 0.

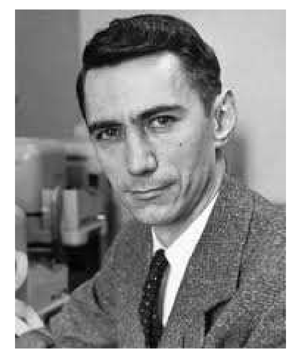
A	~
0	1
1	0

Xor: `A ^ B` = 1 when either A = 1 or B = 1, but not both.

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

Application of Boolean Algebra

In a 1937 MIT Master's Thesis, Claude Shannon showed that Boolean algebra would be a great way to model digital networks.



At that time, the networks were relay switches. But today, all combinational circuits can be described in terms of Boolean "gates."

Boolean Algebra

- $\langle \{0,1\}, |, \&, \sim, 0, 1\rangle$ forms a *Boolean algebra*.
- Or is the sum operation.
- And is the product operation.
- $\sim$ is the "complement" operation (not additive inverse).
- 0 is the identity for sum.
- 1 is the identity for product.

Boolean Algebra Properties

Some boolean algebra properties are similar to integer arithmetic, some are not.

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:

$A \,\&\, (B \| C) = (A \,\&\, B) \| (A \,\&\, C)$	$A * (B + C) = (A * B) + (A * C)$

Sum and Product Identities:

$A \| 0 = A$	$A + 0 = A$
$A \,\&\, 1 = A$	$A * 1 = A$

Boolean Algebra Properties

Zero is product annihilator:

$A \,\&\, 0 = 0$	$A * 0 = 0$

Cancellation of negation:

$\sim(\sim A)) = A$	$-(-A)) = A$

The following boolean algebra rules don't have analogs in integer arithmetic.

Boolean: Sum distributes over product

$A \| (B \,\&\, C) = (A \| B) \,\&\, (A \| C)$	$A + (B * C) \neq (A + B) * (A + C)$

Boolean: Idempotency

$A \| A = A$	$A + A \neq A$
$A \,\&\, A = A$	$A * A \neq A$

Boolean Algebra Properties

Boolean: Absorption

$A \| (A \,\&\, B) = A$	$A + (A * B) \neq A$
$A \,\&\, (A \| B) = A$	$A * (A + B) \neq A$

Boolean: Laws of Complements

$A \| \sim A = 1$	$A + -A \neq 1$

Ring: Every element has additive inverse

$A \| \sim A \neq 0$	$A + -A = 0$

Properties of & and ^

Commutative sum:	$A \hat{} B = B \hat{} A$
Commutative product:	$A \,\&\, B = B \,\&\, A$
Associative sum:	$(A \hat{} B) \hat{} C = A \hat{} (B \hat{} C)$
Associative product:	$(A \,\&\, B) \,\&\, C = A \,\&\, (B \,\&\, C)$
Prod. over sum:	$A \,\&\, (B \hat{} C) = (A \,\&\, B) \hat{} (A \,\&\, C)$
0 is sum identity:	$A \hat{} 0 = A$
1 is prod. identity:	$A \,\&\, 1 = A$
0 is product annihilator:	$A \,\&\, 0 = 0$
Additive inverse:	$A \hat{} A = 0$

Relations Between Operations

DeMorgan's Laws
Express & in terms of |, and vice-versa:

$$A \,\&\, B = \sim(\sim A \mid \sim B)$$

$$A \mid B = \sim(\sim A \,\&\, \sim B)$$

Exclusive-Or using Inclusive Or:

$$A \hat{} B = (\sim A \,\&\, B) \mid (A \,\&\, \sim B)$$

$$A \hat{} B = (A \mid B) \,\&\, \sim(A \,\&\, B)$$

Generalized Boolean Algebra

We can also operate on bit vectors (bitwise). All of the properties of Boolean algebra apply:

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


Bit Level Operations in C

The operations &, |, ~, ^ are all available in C.

- Apply to any *integral* data type: long, int, short, char.
- View the arguments as bit vectors.
- Operations are applied bit-wise to the argument(s).

Examples: (char data type)

$\sim 0x41$	$\rightarrow 0xBE$
$\sim 01000001_2$	$\rightarrow 10111110_2$
$\sim 0x00$	$\rightarrow 0xFF$
$\sim 00000000_2$	$\rightarrow 11111111_2$
$0x69 \,\&\, 0x55$	$\rightarrow 0x41$
$01101001_2 \,\&\, 01010101_2$	$\rightarrow 01000001_2$
$0x69 \mid 0x55$	$\rightarrow 0x7D$
$01101001_2 \mid 01010101_2$	$\rightarrow 01111101_2$
$01101001_2 \hat{} 01010101_2$	$\rightarrow 00111100_2$

Logical Operators in C

There is another set of operators in C, called the *logical operators*, (`&&`, `||`, `!`). These treat inputs as booleans, not as strings of booleans.

- View 0 as "False."
- View anything nonzero as "True."
- Always return 0 or 1.
- Always do short-circuit evaluation (early termination)
- There isn't a "logical" xor, but `!=` works if you know the inputs are boolean.

Examples:

`!0x41`	$\rightarrow$ 0x00
`!0x00`	$\rightarrow$ 0x01
`!!0x41`	$\rightarrow$ 0x01
`!!0x69 && 0x55`	$\rightarrow$ 0x01
`!!0x69 || 0x55`	$\rightarrow$ 0x01

A Puzzle

Given 8 light switches on each of floors A and B, how could you store the following information efficiently?

1. Which lights are on on floor A?
2. Which lights are on on floor B?
3. Which corresponding lights are on both floors?
4. Which lights are on on either floor?
5. Which lights are on on floor A but not floor B?

Representing Sets with Masks

Representation
A bit vector a may represent a subset S of some "reference set" (actually list) L: $a_j = 1$ iff $L[j] \in S$

Bit vector A:

01101001	represents $\{B, C, E, H\}$
ABCDEFGH	

Bit vector B:

01010101	represents $\{B, D, F, H\}$
ABCDEFGH	

What bit operations on these set representations correspond to: intersection, union, complement?

Representing Sets

Bit vector A: 01101001 $= \{$B, C, E, H$\}$
Bit vector B: 01010101 $= \{$B, D, F, H$\}$

Operations:
Given the two sets above, perform these bitwise ops to obtain:

Set operation	Bool op	Result	Set
Intersection	`A & B`	01000001	$\{B, H\}$
Union	`A | B`	01111101	$\{B, C, D, E, F, H\}$
Symmetric difference	`A ^ B`	00111100	$\{C, D, E, F\}$
Complement	`~A`	10010110	$\{A, D, F, G\}$

How would you know if lights D and E were on? How about if *only* lights D and E? How about without using ==?

Shift Operations

Left Shift: `x << y`
Shift bit vector x left by y positions

- Throw away extra bits on the left.
- Fill with 0's on the right.

Right Shift: `x >> y`
Shift bit vector x right by y positions.

- Throw away extra bits on the right.
- **Logical shift:** Fill with 0's on the left.
- **Arithmetic shift:** Replicate with most significant bit on the left.

Unlike Java, C uses the same operator for logical and arithmetic right shift; the compiler "guesses" which one you meant according to the type of the operand (logical for unsigned and arithmetic for signed).

Shift Examples

Argument x	01100010
x << 3	00010000
x >> 2 (logical)	00011000
x >> 2 (arithmetic)	00011000

Argument x	10100010
x << 3	00010000
x >> 2 (logical)	00101000
x >> 2 (arithmetic)	11101000

For right shift, the compiler will choose arithmetic shift if the argument is signed, and logical shift if unsigned.

Cool Stuff with XOR

Bitwise XOR is a form of addition, with the extra property that each value is its own additive inverse: `A ^ A = 0`.

```
void funny_swap(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

Is there ever a case where this code fails?

Main Points

It's all about bits and bytes.

- Numbers
- Programs
- Text

Different machines follow different conventions.

- Word size
- Byte ordering
- Representations

Boolean algebra is the mathematical basis.

- Basic form encodes "False" as 0 and "True" as 1.
- General form is like bit-level operations in C; good for representing and manipulating sets.