Chapter 2: Bits and Bytes II

Topics

- Byte Ordering: Little vs Big Endian
  - Ints
  - Pointers
  - Characters
  - Strings
- Bit-Level Operations in C
  - &, |, ~, ^
  - Bit Masking
  - Logical Expressions

Announcements

- Data Lab is due Friday Sept 12 by 11:55 pm
  - TAs may offer extra office hours later in the week
- Recitation Exercise #1 is available on moodle and is due Monday Sept 8
  - Print and hand in a hard copy at the beginning of recitation
  - These problems are useful in helping to study for the midterm
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 2 (2.1-2.3 this week)
  - Next week: begin Chapter 3

Recap…

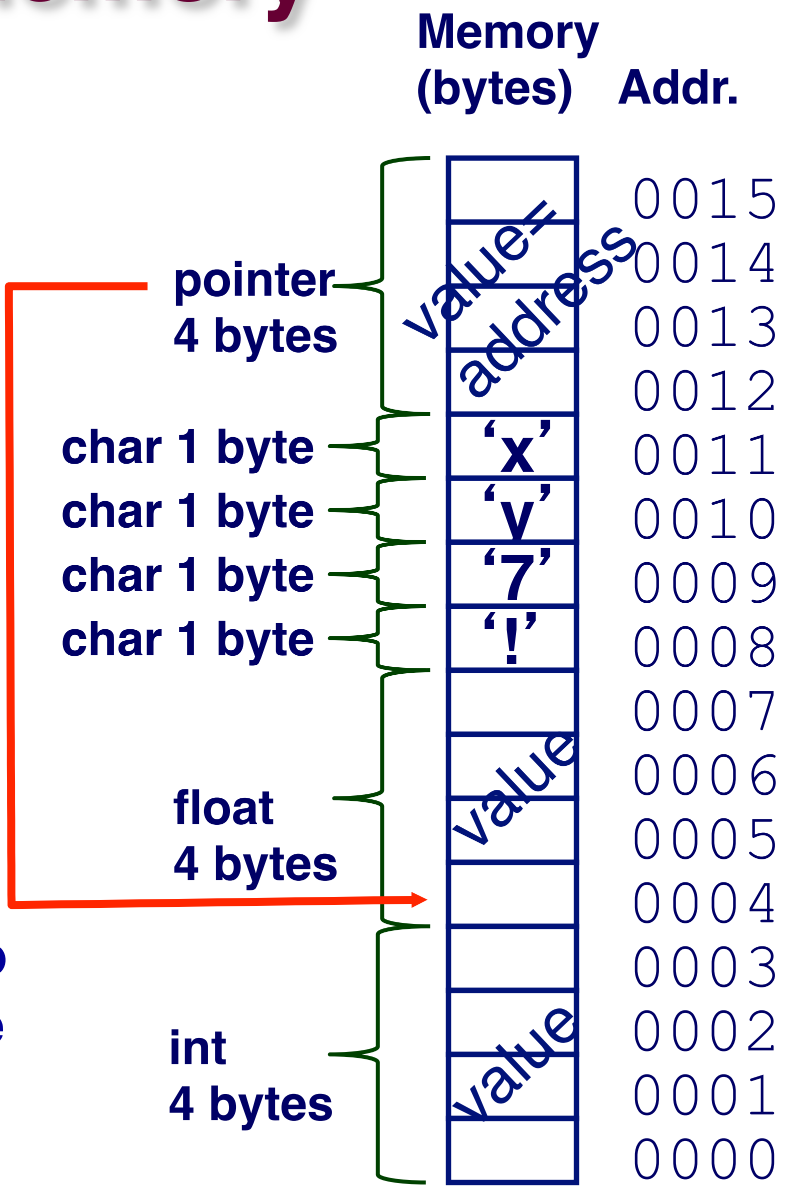
- Binary representations – base 2
- Binary digital logic
- Hexadecimal representations – base 16
- Byte-addressed memory
- Representing data in C
  - Ints, shorts, floats, doubles, chars, etc.
- Pointers in C

Recap: Byte-based Memory

IA32 Example:

- Address of int is 0x00000000
- Address of float is 0x00000004
- Address of character = ‘7’ is 0x00000009
- Address of pointer is 0x0000000c
  - Note: the pointer points to another memory location, i.e. stores a memory location *address*

    e.g. if pointer = 0x00000004 it means the pointer is pointing to the float! (actually the first byte of the float)



Byte Ordering

In what order should bytes within a multi-byte word be stored in memory?

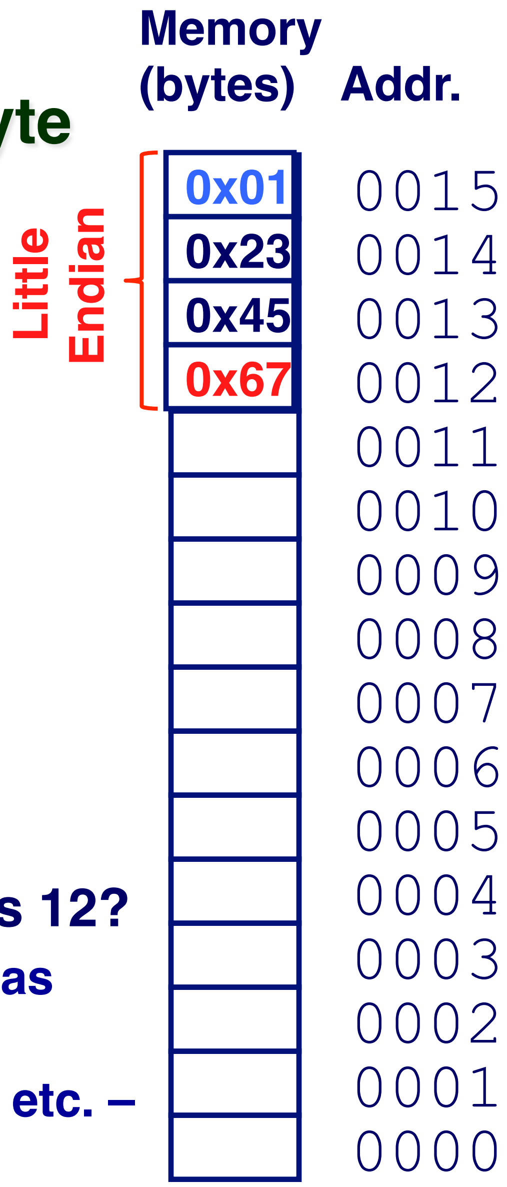
- Consider a 32-bit integer or int (4 bytes)

  $0000\ 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111_2$

  = 0x 0 1 2 3 4 5 6 7

  = 0x $\underbrace{01}_{\text{Most Significant Byte (MSB)}}$ 23 45 $\underbrace{67}_{\text{Least Significant Byte (LSB)}}$

- In what order do we store these four bytes 0x01234567 in memory, say at starting address 12?
  - **Little Endian approach**: Least significant byte has lowest address (12), followed by the next least significant byte in the next lowest address (13), etc. – i.e. "little end first"



Byte Ordering Example

Big Endian

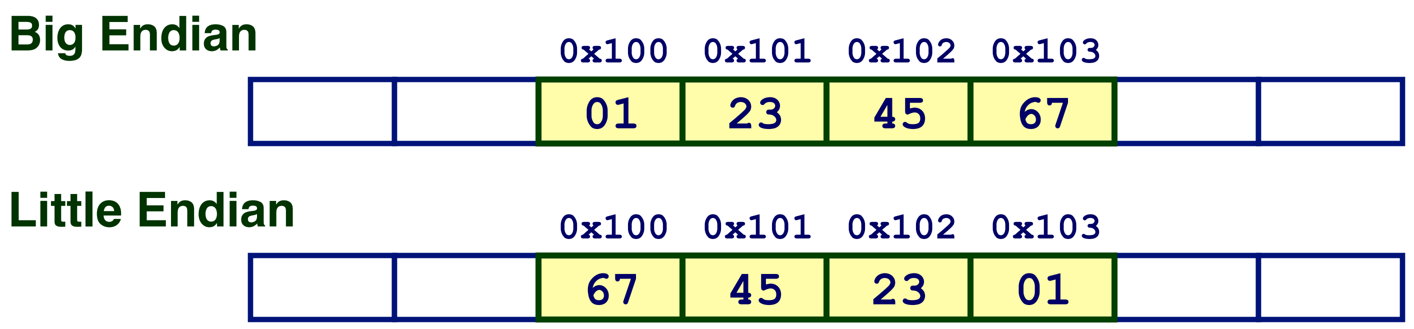
- Most significant byte has lowest address, i.e. “big end first”

Little Endian

- Least significant byte has lowest address

Example

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



Byte Ordering in Hardware

Aside: Endianness came from Gulliver’s Travels

CPU Conventions

- Sun SPARC, Motorola 68K, Power PC (PPC’s) are “Big Endian” machines
  - Most significant byte has lowest address
- ISA32 are “Little Endian” machines
  - Least significant byte has lowest address
- Some are “bi-endian” (hardware supports both types of Endianness, which can improve performance)
  - MIPS, Alpha, ARM

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 4 bytes: `0x000012ab`
- Split into bytes: `00 00 12 ab`
- Reverse: `ab 12 00 00`

So this is 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("0x%p\t0x%.2x\n",
           start+i, start[i]);
  printf("\n");
}
```

equivalent to *(start+i)

Printf directives:

`%p`: Print pointer

`%x`: Print Hexadecimal

Use show_bytes() to find if your machine is Little Endian or Big Endian

show_bytes Execution Example

```
int a = 15213;                   // = 0x3B6D
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

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

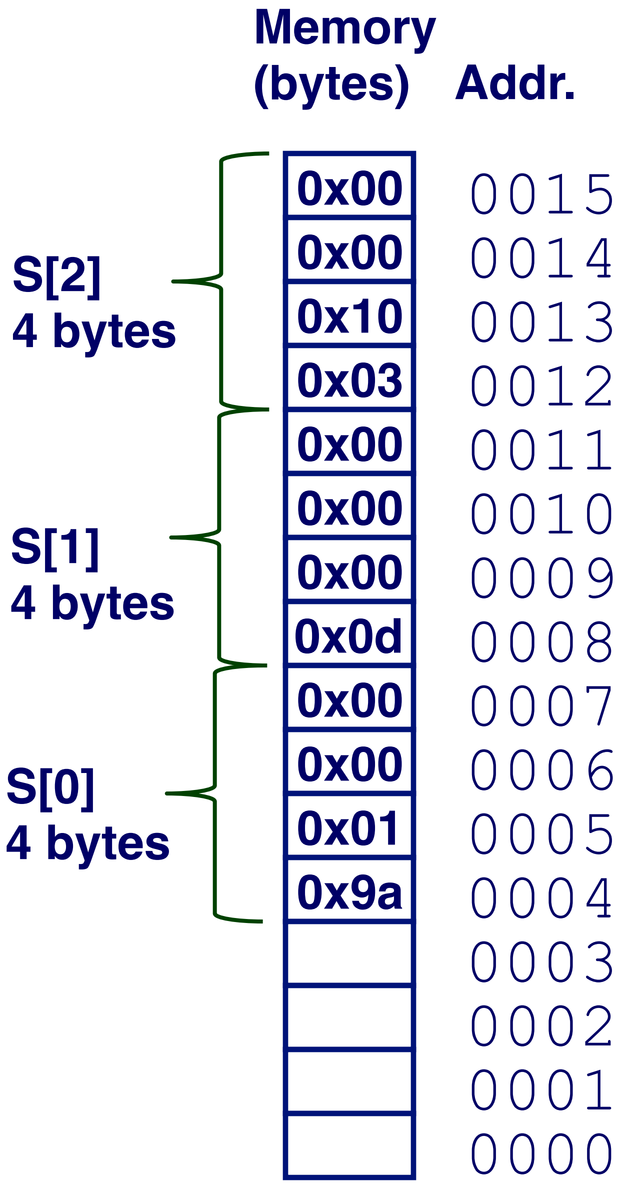
Little Endian with zero padding in most significant bytes

Endianness and Arrays

Integer Array Example:

```
int S[3];
S[0] = 410;         /* = 0x0000019a  */
S[1] = 14;          /* = 0x0000000d  */
S[2] = 4099;        /* = 0x00001003  */
```

- Let the int array S be stored starting at memory address 4
- Lowest element of array S[0] is stored starting at the lowest address of the array (byte 4)
  - Next lowest element of array S[1] is stored at the next lowest memory address of int array (byte 8). And so on…
- Note how each int in the array is stored according to byte ordering rules (little endian)



Endianness and Pointers

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

Sun Address (32-bit)

Hex:	E	F	F	F	F	B	2	C
Binary:	1110	1111	1111	1111	1111	1011	0010	1100

Big Endian Sun `P`

2C	High mem
FB	
FF	
EF	Low mem

Little EndianLinux `P`

BF
FF
F8
D4

Linux Address (32-bit)

Hex:	B	F	F	F	F	8	D	4
Binary:	1011	1111	1111	1111	1111	1000	1101	0100

Alpha `P`

00
00
00
01
FF
FF
FC
A0

X64 Address (64-bit, higher order hex not shown)

Hex:	1	F	F	F	F	F	C	A	0
Binary:	0001	1111	1111	1111	1111	1111	1100	1010	0000

Different compilers & machines assign different locations to objects, i.e. not only are the addresses stored in the pointer of different widths, but they are also of different values (locations)

Endianness and Floats

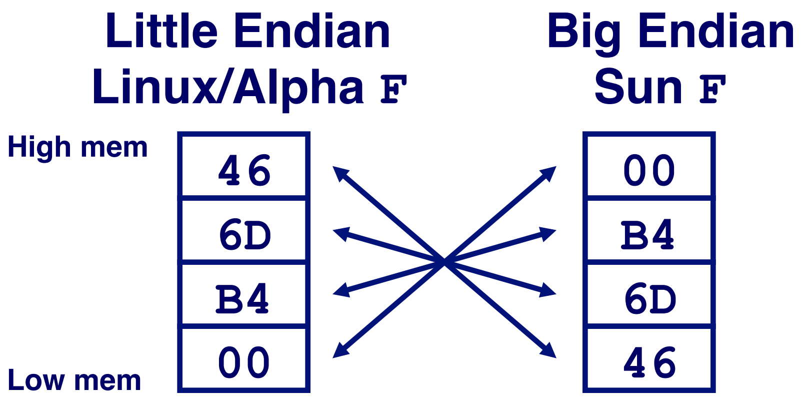
```
Float F = 15213.0;
```

IEEE Single Precision Floating Point Representation

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
15213:			1110	1101	1011	01		

Not same as integer representation, but consistent across machines

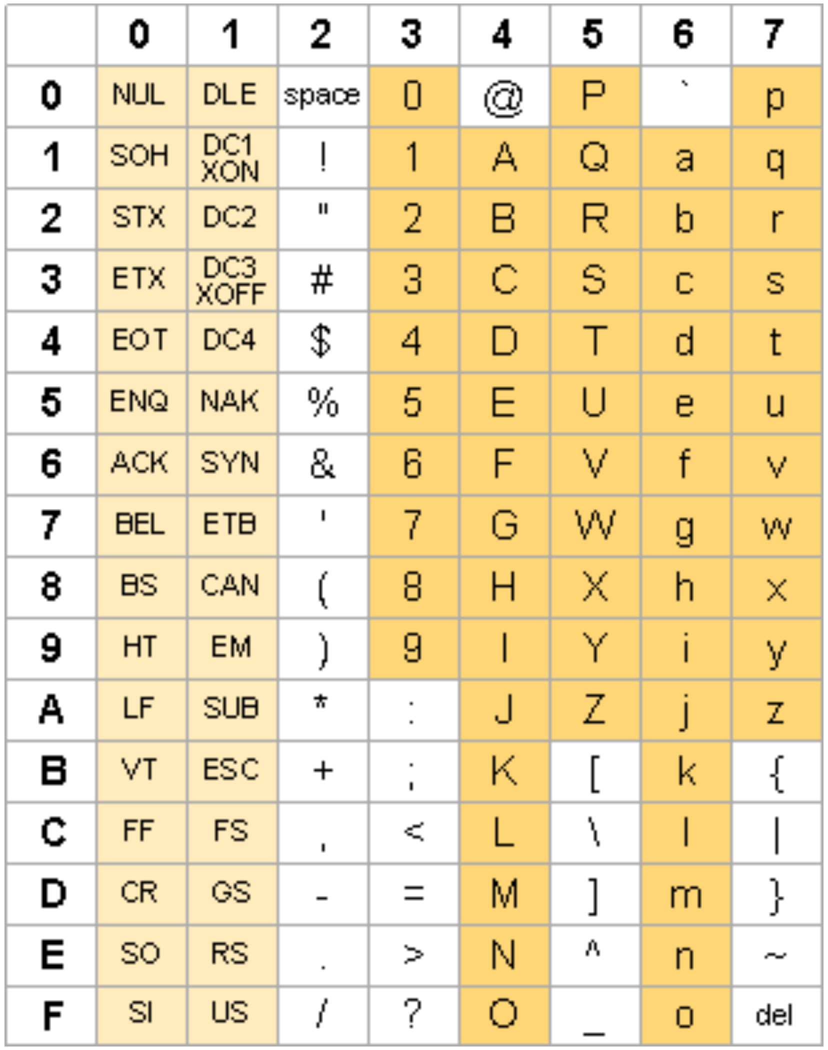
Can see some relation to integer representation, but not obvious



Representing Characters

- Actually unsigned characters, via ASCII (American Standard Code for Information Exchange)
- The alphabet, numbers, punctuation, and symbols are encoded via an 8-bit ASCII table:
- Example: numbers start with '0' = 0x30, capital letters start at 'A' = 0x41 = 65, lower case letter 'a' = 0x61
- Endianness doesn't affect character representations

(hex)



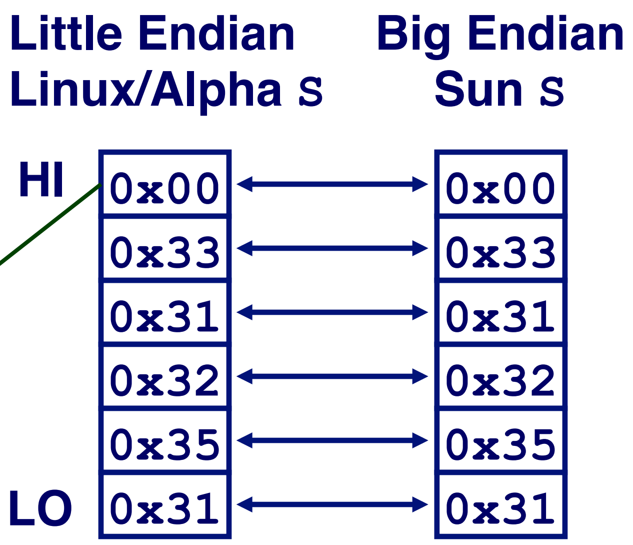
	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Representing Strings

Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - *'1' = 0x31*
  - *'5' = 0x35*
  - *'2' = 0x32*
  - *'1' = 0x31*
  - *'3' = 0x33*
- String should be null-terminated
  - Final character = 0

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



Compatibility

- Byte ordering not an issue
  - Data are single byte quantities
- Text files generally platform independent
  - Except for different conventions of line termination character(s)!

Machine-Level Code Representation

Encode Program as Sequence of Instructions

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

Programs are Byte Sequences Too!

Representing Instructions

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

- For this example, Alpha & Sun use two 4-byte instructions
  - Use differing numbers of instructions in other cases
- PC uses 7 instructions with lengths 1, 2, and 3 bytes
  - Same for NT and for Linux
  - NT / Linux not fully binary compatible

Alpha `sum`	Sun `sum`	PC `sum`
00	81	55
00	C3	89
30	E0	E5
42	08	8B
01	90	45
80	02	0C
FA	00	03
6B	09	45
		08
		89
		EC
		5D
		C3

Different machines use totally different instructions and encodings

Relations Between Logic 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
  - » This is Shannon's circuit.
- 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

Using Boolean Operators for 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`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - `~0x41 --> 0xBE`

    `~`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

Bit Masking

- Example: Mask out the 4 least significant bits of a byte 0x69
  - `Let mask = 0xF0`

    `0x69 & 0xF0 -->  0x60`

    01101001_2 & 11110000_2 --> 01100000_2

- Example: Mask out all but the most significant bit of a byte 0x69
  - `Let mask = 0x80`

    `0x69 & 0x80 -->  0x00`

    01101001_2 & 10000000_2 --> 00000000_2

Logical vs Bitwise Operations in C

- **Logical Operators**
  - `&& (AND), || (OR), ! (NOT or "bang")`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- **Example code:**

  -
    ```
    int x,y,z;
    ….
    if ( !((x==0) && (x>y))  || (z<256) ) {
      …
      z = ~(x & y) | z;
    }
    ```

Each logical expression is either TRUE (1) or FALSE (0):
(x==0)
(x>y)
((x==0) && (x>y))
!((x==0) && (x>y))
(z<256)
!((x==0) && (x>y)) || (z<256)

Compare to the bit-wise logical operations, where input, intermediate, and final values don't have to be 0 or 1

By the way, parentheses are your friend, and good programming practice