CS330 - Computer Organization and Assembly Language Programming

Lecture 4

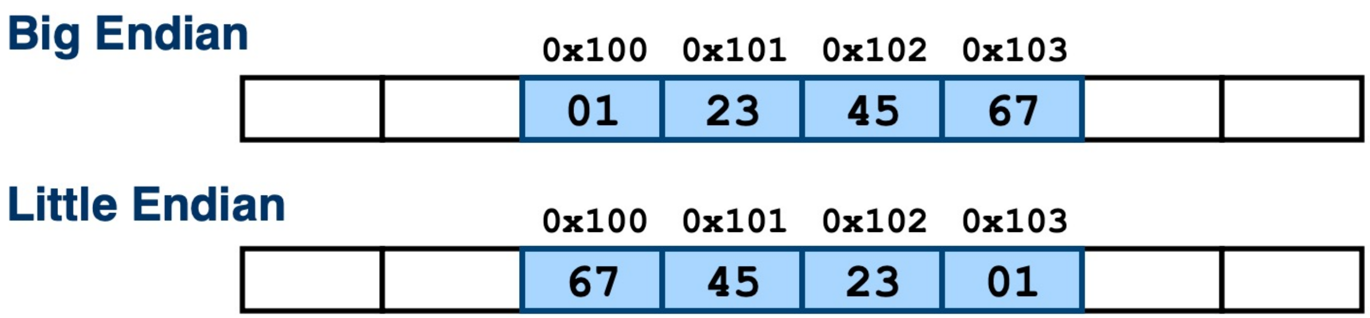
Professor : Mahmut Unan – UAB CS

Agenda

- Byte Ordering
- Boolean Algebra
- Bit-Level Operations in C
- Logical Operations in C
- Shift Operations in C

Byte Ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - (most) Sun's, IBMs are "Big Endian" machines
    - Least significant byte has highest address (comes last)
  - (most) Intel's are "Little Endian" machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100 0x100 0x101



http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

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");
}
}
```

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

`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
```

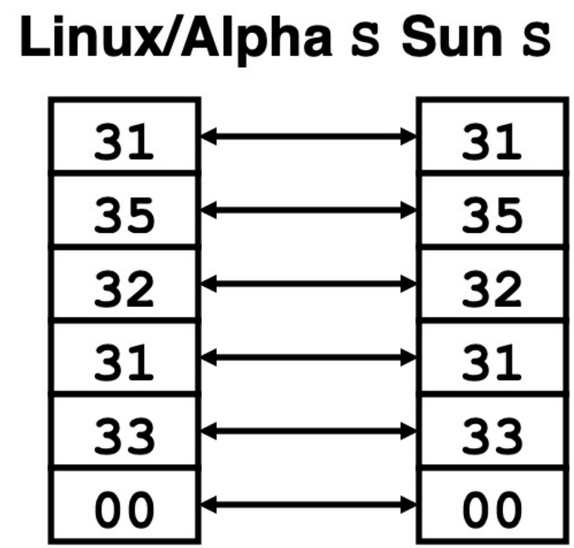
$$\mathbf{0011\ 1011\ 0110\ 1101_2}$$
$$\mathbf{3\quad b\quad 6\quad d_{16}}$$

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Representing Strings

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Other encodings exist, but uncommon
    - 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
    - Data are single byte quantities
  - Text files generally platform independent
    - Except for different conventions of line termination character(s)!

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



http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

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, Mac's use 4 byte instructions
      - » Reduced Instruction Set Computer (RISC)
    - PC's use variable length instructions
      - » Complex Instruction Set Computer (CISC)
  - Different machines → different ISA & encodings
    - Most code not binary compatible
- A fundamental concept:
  - Programs are byte sequences too!

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Representing Instructions

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

- Sun use 2 4-byte instructions
    - Differing numbers in other cases
- PC uses instructions with lengths 1, 2, and 3 bytes
    - Mostly the same for NT and for Linux
    - NT / Linux not fully binary compatible

Linux 32	55	89	E5	8B	45	0C	03	45	08	C9	C3
Windows	55	89	E5	8B	45	0C	03	45	08	5D	C3
Sun	81	C3	E0	08	90	02	00	09			

Different machines use totally different instructions and encodings

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Variables and Operations

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - 〈{0,1}, |, &, ~, 0, 1〉
  - | is “sum” operation, & is “product” operation
  - ~ is “complement” operation (not additive inverse)
  - 0 is identity for sum, 1 is identity for product
- Makes use of variables and operations
  - Are logical
  - A variable may take on the value 1 (TRUE) or 0 (FALSE)
  - Basic logical operations are AND, OR, XOR and NOT

Boolean Variables and Operations / 2

- AND
  - Yields true (binary value 1) if and only if both of its operands are true
  - In the absence of parentheses the AND operation takes precedence over the OR operation
  - When no ambiguity will occur the AND operation is represented by simple concatenation instead of the dot operator
- OR
  - Yields true if either or both of its operands are true
- NOT
  - Inverts the value of its operand

Table: Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \bullet Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \bullet Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0						
0	1						
1	0						
1	1						

Table: Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \bullet Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \bullet Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \bullet B \bullet \ldots$	All of the set {A, B, …} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, …} are 1.
NAND	$\overline{A \bullet B \bullet \ldots}$	Any of the set {A, B, …} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, …} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, …} contains an odd number of ones.

Table: Basic Identities of Boolean Algebra

Basic Postulates		
$A \bullet B = B \bullet A$	$A + B = B + A$	Commutative Laws
$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$	$A + (B \bullet C) = (A + B) \bullet (A + C)$	Distributive Laws
$1 \bullet A = A$	$0 + A = A$	Identity Elements
$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \bullet A = 0$	$1 + A = 1$	
$A \bullet A = A$	$A + A = A$	
$A \bullet (B \bullet C) = (A \bullet B) \bullet C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \bullet B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \bullet \overline{B}$	DeMorgan's Theorem

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

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Exercise 1

Evaluate the following expression when A =0, B =1, and C= 1

F= $B + \overline{C}A + B\overline{A} + A\overline{B}$

Simplify the following function;

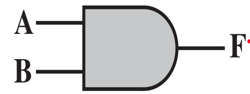
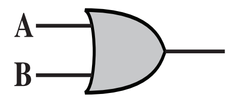
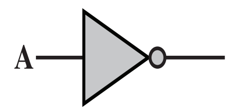
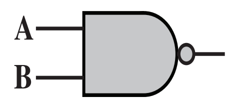
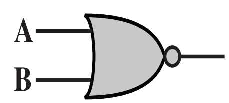
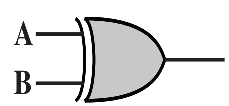
F= $AB + BC + \overline{B}C$

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
- The fundamental building block of all digital logic circuits is the gate.
  - Logical functions are implemented by the interconnection of gates.
  - A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals.

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Basic Logic Gates

Name	Graphical Symbol	Algebraic Function	Truth Table
AND		$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0


Digital Logic & Design
ANDROID
NANDROID
NOTROID
ORROID

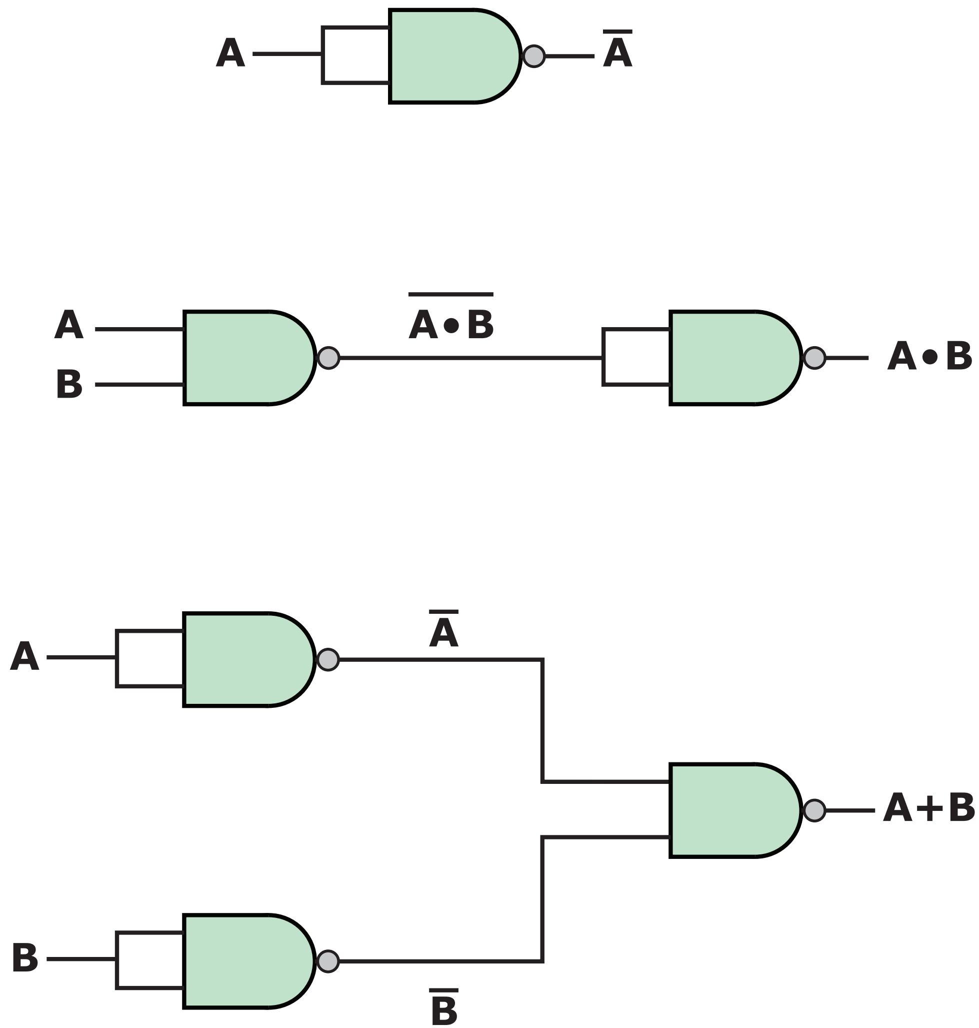


Figure 11.2 Some Uses of NAND Gates

Exercise 2

Draw a truth table for $A\overline{B}\ (A+B)$

Exercise 3

- Draw a logic circuit for A + BC + $\overline{\mathrm{D}}$

General Boolean Algebras

- Boolean operations can be extended to work 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
- Now, Boolean |, & and ~ correspond to set union, intersection and complement

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

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*

0	1	1	0	1	0	0	1
7	6	5	4	3	2	1	0

    - 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 }

Exercise 4

- a =[01101001]
- b=[01010101]
- ~a =
- ~b
- a & b
- a | b
- a ^ b

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
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Exercise 5

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

Step	*x	*y
Initially	*a*	*b*
Step 1		
Step 2		
Step 3		

Exercise 5

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

Step

Initially

Step 1

Step 2

Step 3

Rules to remember

$a \wedge a = 0$

$a \wedge 0 = a$

Exercise 5

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

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a*^*b*
Step 2		
Step 3		

Exercise 5

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

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a^b*
Step 2	*a^(a^b)*	*a^b*
Step 3		

Exercise 5

```
void inplace_swap (int *x, int *y){
*y =*x ^ *y;  /*Step 1*/
*x = *x ^ *y;    /*Step 2*/
*y = *
}
```

a^(a^b) = (a^a)^b

0

= b

Step		
Initially		
Step 1	a	a^b
Step 2	a^(a^b)	a^b
Step 3		

Exercise 5

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

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a*^*b*
Step 2	*a*^(*a*^*b*)	*a*^*b*
Step 3	*b*	*b*^(*a*^*b*)

Exercise 5

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

Step	*x	
Initially	*a*	
Step 1	*a*	
Step 2	*a^(a^b)*	
Step 3	*b*	*b^(a^b)*

b^(a^b) = (b^b)^a
= a

Exercise 5

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

Step	*x	*y
Initially	*a*	*b*
Step 1	*a*	*a^b*
Step 2	*a^(a^b)*	*a^b*
Step 3	***b***	***a***