Lecture 19: Boolean Algebra

- Basic Boolean Algebra
- Boolean Functions in Assembly

Boolean Algebra

- Two valued algebra
- Used to analyze the basic elements that digital computers are built from.
- A way of manipulating true/false values.

Boolean Operations

- Basic operations:
  - AND – true iff both operands are true
  - OR – true if either or both operands are true
  - NOT – true when its operand is false (inverts the operand)
- Other common operations:
  - XOR – true if inputs are different
  - NAND – inversion of AND
  - NOR – inversion of OR
- 1 = true, 0 = false

AND

- Truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A AND B can also be represented as:
- A • B
- AB
- A ^ B
**OR**

- Truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A OR B can also be represented as:
- A + B
- A v B

**NOT**

- Truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

NOT A can also be represented as:
- A'
- ¬A
- •A

**XOR (Exclusive OR)**

- Truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A XOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A XOR B can also be represented as:

**16 Possible Boolean Functions of Two Variables**

- table from AoA, Chapter 2
**Identities of Boolean Algebra**

**AND form** | **OR form**
---|---
Identity law | 1A = A | 0 + A = A
Null law | 0A = 0 | 1 + A = 1
Idempotent law | AA = A | A + A = A
Inverse Law | A̅A = 0 | A + A̅ = 1
Commutative Law | AB = BA | A + B = B + A
Associative Law | (AB)C = ABC | (A + B) + C = A + (B + C)
Distributive Law | A + BC = (A + B)(A + C) | A(B + C) = AB + AC
Absorption Law | A(A + B) = A | A + AB = A
De Morgan’s Law

- proof of AND form of distributive law using Truth Tables (on the board)

- proof of OR-form of DeMorgan’s Law using truth tables (on board)

**Generating a Logic Function from a Truth Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Find all the combinations that result in a one.
- Put them into an expression:
Boolean Functions in Assembly

• Boolean functions fall into the category of bit-operations.
• What other bit operations have we seen?

• For boolean functions, the operations take place between the individual bits of the two operands.

AND

• AND performs a bitwise AND operation between each bit of the two operands and places the result in the first operand.
• Formats:
  AND  reg, reg
  AND  reg, mem
  AND  reg, immed
  AND  mem, reg
  AND  mem, immed

AND, cont.

• AND can clear selected bits in an operand while preserving (masking) the remaining bits.

```
  mov  al, 00111011b
  and  al, 00001111b ; al = 00001011b
```

The 00001111b is called a bit mask, it clears the upper four bits while preserving the lower four bits.

AND Example

• Converting from lower case to upper case.
• Upper case letters have bit 5 set

```
.data
char   db  ? ; put uppercase letter here
mask   db  0DFh : 11011111b
.code
mov   ah, 1
int   21h ; get the char into AL
and   al, mask ; mask out bit 5
mov   char, al ; store uppercase char
```
OR

- Performs a bitwise OR operation between each bit of the two operands and places the result in the first operand.
- Same formats as AND.
- OR is useful for setting certain bits to one while leaving the other bits unchanged.
  ```
  mov al, 00111011b ;3Bh
  or al, 00001111b ;AL = 3Fh
  the lower four bits of the result are set, the others remain unchanged.
  ```

OR Example

- Converting from upper case to lower.
- When we converted the other way, we cleared bit 5. Now we need to set it:
  ```
  .data
  char db ?;put lowercase letter here
  setb db 20h ; 00100000b
  .code
  mov ah, 1
  int 21h ;get the char into AL
  or al, setb;set bit 5
  mov char, al ;store lowercase char
  ```

Another OR Example

;converting a decimal digit to ASCII

```
DIGIT DW 7
ASCBias DW 30h
....
  MOV AX, DIGIT
  OR AX, ASCBias
```
**NOT**

- NOT reverses all the bits in an operand (takes the 1’s complement).
- Formats:
  - NOT reg
  - NOT mem

  \[
  \begin{align*}
  \text{mov} & \text{ al, 11110000b} \\
  \text{not} & \text{ al} \quad ;\text{al} = 00001111b
  \end{align*}
  \]

**NEG**

- NEG reverses the sign of a number by converting it to its two’s complement.
- Formats:
  - NEG reg
  - NEG mem

  \[
  \begin{align*}
  \text{mov} & \text{ al, +127} \ ; \text{AL} = 01111111b \\
  \text{neg} & \text{ al} \quad ;\text{AL} = 10000011b
  \end{align*}
  \]

**Overflow with NEG**

- You can get overflow:

  \[
  \begin{align*}
  \text{mov} & \text{ al, -128} \ ;\text{AL} = 10000000b \\
  \text{neg} & \text{ al} \quad ;\text{AL} = 10000000b, \\
  & \text{OF} = 1
  \end{align*}
  \]

**XOR**

- Performs a bit-by-bit exclusive OR, puts the result in the first operand.

  \[
  \begin{align*}
  \text{mov} & \text{ al, 10110100b} \\
  \text{xor} & \text{ al, 10000110b} \ ; \text{al} = 00110010b
  \end{align*}
  \]

- Commonly used to set a register to zero:

  XOR AX, AX ;same effect as mov ax, 0