Unary/Binary Operations

Unary vs Binary Operation

- An operation takes some number of **operands** and provides a **result**
- The number of operands determines its name
  - An operation with one operand is a unary operation
    - Examples: ++ and -- operators
  - An operation with two operands is a binary operation
    - Examples: +, -, /, *, %

Trivia: Many high-level programming languages have a ternary (three operand) operation

Bitwise Operations

- Similar to logical (boolean) operations from high level languages
- Logical operations work on values as a whole
- Bitwise operations apply the operation to **each** bit in the value
- We will look at the most common bitwise operations
    - There are actually 16 different binary bitwise operations
    - How do we know there are exactly 16 binary bitwise operations?

Bitwise Operation - NOT

NOT is a Unary Operator (i.e. has only one operand)

Operand	Result
0	1
1	0

$$\text{Not}(11011_2) = 00100_2$$

Note:

- The NOT operation is applied to each bit
- Bitwise NOT is the 1’s complement negation operation

Bitwise Operation - AND

Common bitwise operations: **AND (binary operator)**

Operand 1	Operand 2	Result
0	0	0
0	1	0
1	0	0
1	1	1

$$\begin{array}{rr} & 1010_2 \\ \text{AND} & \underline{0110_2} \\ & 0010_2 \end{array}$$

Notice:

A and 0 = 0

A and 1 = A

Bitwise Operation - OR

Common bitwise operations: **OR (binary operator)**

Operand 1	Operand 2	Result
0	0	0
0	1	1
1	0	1
1	1	1

$$\begin{array}{rr} & 1010_2 \\ \text{OR} & \underline{0110_2} \\ & 1110_2 \end{array}$$

Notice:

A or 0 = A

A or 1 = 1

Bitwise Operation - XOR

Common bitwise operations: **XOR (binary operator)**

Operand 1	Operand 2	Result
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{array}{rl} & 1010_2 \\ \text{XOR} & \underline{0110_2} \\ & 1100_2 \end{array}$$

Notice:

A xor 0 = A

A xor 1 = Not A

A Mask

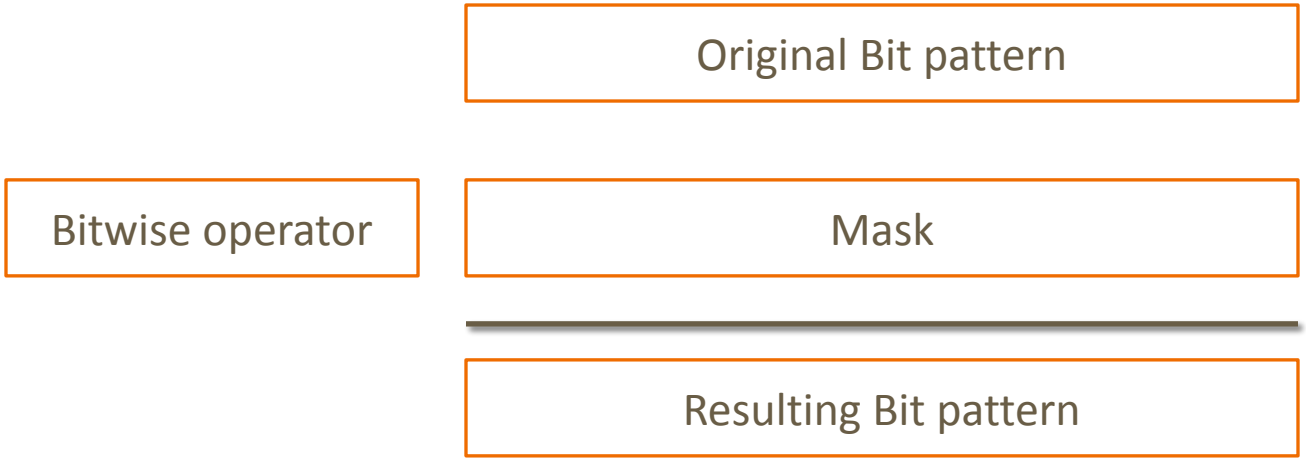
- A mask is a bit pattern designed to affect certain target bits within a binary number.
- The choice of the mask depends on:
  - the operation being completed
    - determines the binary operation to be used
  - the bits to be affected
- The position of a bit starts counting from the left at 0
  - Digits positions in a byte: **76543210**

Uses for a Mask

- Arguments to functions where space is important
- Networking and IP addresses
- Image masking

http://en.wikipedia.org/wiki/Mask_%28computing%29#Inverse_Masks

A Mask


Original Bit pattern
Bitwise operator
Mask
Resulting Bit pattern

Common Bitmask Functions - OR

Masking bits to 1 (called setting)

- Bitwise operator to use: OR
- If you want a bit to stay the same at a specific location
  - Make the bit 0 in the mask.
- If you want to make sure it is set to 1 at a specific location
  - Make the bit 1 in the mask

Common Bitmask Functions - OR

A mask that <u>sets</u> bits 1 and 5 to 1

- Keyword **set** tells us the operation is **OR**
- A value of 1 for bit to be set gives:

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

$00100010_2 = 22_{16}$

Common Bitmask Functions - OR

Original: $10000000_2 = 80_{16}$

OR

MASK: $00100010_2 = 22_{16}$

Result : $10100010_2 = A2_{16}$

Common Bitmask Functions - AND

Masking bits to 0 (called clearing)

- Bitwise operator to use: AND
- If you want a bit to stay the same at a specific location
  - Make the bit 1 in the mask.
- If you want to make sure it is set to 0 at a specific location
  - Make the bit 0 in the mask

Common Bitmask Functions - AND

A mask that clears bits 1 and 5 to 1

- Keyword **clear** tells us the operation is **AND**
- A value of 0 for a bit to be cleared gives:

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

$11011101_2 = DD_{16}$

Common Bitmask Functions - AND

Original:$01111111_2 = 7F_{16}$

AND

MASK: $11011101_2 = DD_{16}$

Result : $01011101_2 = 5D_{16}$

Common Bitmask Functions - XOR

Masking bits to switch their value (called toggling)

- Bitwise operator to use: XOR
- Toggle means change the value to the opposite (i.e. 0->1 and 1->0)
- If you want a bit to stay the same at a specific location
  - Make the bit 0 in the mask.
- If you want to toggle a bit at a specific location
  - Make the bit 1 in the mask

Common bitmask functions - XOR

A mask that <u>toggles</u> bits 1 and 5 to 1

- Keyword **toggle** tells us the operation is **XOR**
- A value of 1 for a bit to be toggled gives:

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

$00100010_2 = 22_{16}$

Common Bitmask Functions

Original: $01111111_2 = 7F_{16}$

XOR

MASK: $00100010_2 = 22_{16}$

Result: $11011101_2 = DD_{16}$

Masks - Example

Given any byte set bit 5 and clear bit 2 (all other bits must be unaffected).

- There is **NO** single bitwise operation that can do both actions
- Need to apply two operations
  - AND (clear) and
  - OR (set)
  - **Note**: the order is irrelevant

Masks - Example

- Clear using **AND** operation
- Use mask where 0 clears original bit, 1 leaves original bit unaffected
- Mask:

```
Position: 76543210
Mask:     11111011
```

Masks - Example

- Set using **OR** operation
- Use mask where 1 sets original bit, 0 leaves original bit unaffected
- Mask:

```
Position: 76543210
Mask:     00100000
```

Masks - Example

Performing these in sequence will generate the desired result

$$
\begin{array}{ll}
 & \texttt{abcdefgh}_2 \\
\texttt{and} & \underline{\texttt{11111011}}_2 \\
 & \texttt{abcde0gh}_2 \\
\texttt{or} & \underline{\texttt{00100000}}_2 \\
 & \texttt{ab1de0gh}_2
\end{array}
$$

You should be able to show that the order in which the operations occur does not impact the final result