Boolean Agebra

- Developed by George Boole ∼1852.
- Algebraic representation of Logic.
- AND: $A\&B = 1$ when $A = 1$ and $B = 1$
- OR: $A \mid B = 1$ when $A = 1$ or $B = 1$
- XOR: $A\hat{}B = 1$ when exactly one of A, B is 1.
- NOT: $\sim A = 1$ when $A = 0$
- De Morgans Law: $\sim (A \mid B) =\sim A\& \sim B$
  $: \sim (A\&B) =\sim A \mid\sim B$
- Operations applied bitwise to bit vectors.

01101001	01101001	01101001	
& 01010101	| 01010101	^ 01010101	∼ 01101001
01000001	01111101	00111100	10101010

- All rules of Boolean Algebra apply.

Application: Representing & Manipulating Sets

- **Representation:**
  w-bit vector represents subsets of $\{0, \ldots, w-1\}$
  $a_j = 1$ if $j \in$ Set.
  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 Ops in C v/s Logical Ops:

- For Logic Ops. $0 \rightarrow F$ Any non-zero value $\rightarrow T$
- Logic Ops. **always** return 0 or 1.

C expr	Binary	Result (HEX)
$\sim 0x41$	$\sim$[0100 0001]	
$\sim 0x00$	[1111 1111]	
$0x69 \& 0x55$		
$0x69 \mid 0x55$		
$0x69\ \hat{}\ 0x55$		
$0x69 << 4$		
$0x69 >> 4$		
$0x95 << 4$		
$0x95 >> 4$		
!0x41		
! !0x41		
!0x00		
$0x69 \&\& 0x55$		
$0x69 \mid\mid 0x55$		

Bit-Shift Ops

- Left Shift: `x << y`
  Shift bits of x left by y positions.
  Extra bits on left lost. Fill with 0's on right
  $x = [01100010]$ `x << 3` $\rightarrow$ [00010000]
- Right Shift `x >> y` Shift bits of x right y positions.
  Extra bits on right lost. Fill with **sign** bit on left
  i.e.
  If $y \geq 0$, Fill with 0's on left: **logical** right shift
  $y = [010100010]$; `y >> 3` $\rightarrow$ [00010100]
- Fill with 1's on left: **arithmetic** right shift
- $y = [10100010]$ `y >> 3` $\rightarrow$ [11110100]
- Let w be the *word size*
  Shift < 0 **OR** $> w$. **undefined behaviour**

Bit Masking

- A common use of bit-level operations is to implement **masking**, where a **mask** is a bit-pattern that indicates a selected set of bits within a word.
- The idea is to operate on a variable, changing only the bits indicated by the bit-mask.

Bitmask		Hex	Bits affected
0001	0000	0x10	bit 4
1100	1100	0xCC	bits 2,3,6,7
0000	1111	0x0F	bits 0,1,2,3
1111	1111	0xFF	bits 0 - 7

- The three most common bitmasks are to *(i)* **read** a bit *(ii)* **set** a bit or *(iii)* **clear** a bit

Operation	Logic
read k^{th} bit	x AND 2^k
set k^{th} bit	x OR 2^k
clear k^{th} bit	x AND (NOT 2^k)

Bit-Level Operations: Examples

Operation	C
read k^{th} bit	$x = (x\&(1 << k)) >> k;$
set k^{th} bit	$x = x \mid (1 << k);$
clear k^{th} bit	$x\&(\sim (1 << k));$

- Code fragment to show bit-mask operations in C:

```
char a = 17;                    //a → 0001    0001
int i;
/* Set bit 3 */
a = a | (1 << 3);               //a → 0001    1001 → 25
/*clear bit 4 */
a = a & (~(1 << 4));            //a → 0000    1001 → 9
/* read all bits */
for (i = 7; i >= 0; i--) {
  printf("%d ", (a & (1 << i)) >> i);
printf("\n");
```

Bit Masking: Examples

- Write C expressions, in term of variable x. Your code should work for any word size ≥ 8. The results for $x = 0x87654321$ are shown below
  - The LSByte of x , with the rest of the bytes set to zero.
  - The LSByte of x set to all 1's and all other bytes unchanged.
- Using **only bit-level** ops (without using ^), write a C expression that computes x ^ y.
- Using only bit-level ops, write a C expression that is equivalent to `x == y`. The expression should return 1 if x equals y and 0 otherwise.