CSC 411

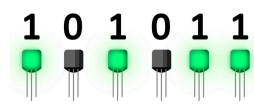
Computer Organization (Spring 2025)
Lecture 3: Bitwise Operations

Christian Esteves, University of Rhode Island

Original slides by Prof. Marco Alvarez, University of Rhode Island

Bits

- Computers use the binary number system to represent and process data
- A **bit** (binary digit) is the smallest unit of data in computing
  - can have a value of 0 or 1
  - easy to implement in digital circuits
  - forms the foundation for all digital information
- Bit Representation
  - bits are typically represented by electrical voltages in computer hardware
    - high voltage corresponds to 1 and low voltage to 0



Bytes

- A **byte** is a group of 8 bits
  - commonly used to represent characters, numbers, and other data
  - smallest addressable unit of memory in most computer architectures

1	0	0	0	1	0	0	1
MSB							LSB
most significant bit							least significant bit

- Important calculations
  - how many different values can be stored in 1 byte?
  - how many different values can be stored in n bits?

Basic data types in C

The C language does not explicitly define data sizes. The actual sizes can vary depending on the compiler and the system architecture.

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Boolean algebra

- Developed by George Boole in the 19th century
  - branch of mathematics dealing with binary variables and logic operations
  - fundamental to digital circuit design and computer science
- Three basic logic operations
  - **AND**: output is 1 only if both inputs are 1 — **conjunction**
  - **OR**: output is 1 if at least one input is 1 — **disjunction**
  - **NOT** output is the opposite of the input — **negation**
- Boolean expressions
  - formed by combining variables and logic operations

Bit vectors

- Sequences of bits that can represent various types of data
- Boolean algebra can be extended to operate on bit vectors
- Applications in Computer Science
  - efficient set representation
  - implementation of data structures
  - low-level programming and bitwise manipulation

```
Understanding boolean algebra with bit vectors is essential for
working with binary data in computer science and digital design
```


Bitwise operators in C

- Operate on “integer” data types
  - long, int, short, char, unsigned variants
- Treat arguments as bit vectors
- Corresponding logic operators are applied bitwise to operands
- Commonly used to manipulate sets and masks

`~`	bitwise NOT	`~a`	the bitwise NOT of **a**
`&`	bitwise AND	`a & b`	the bitwise AND of **a** and **b**
`|`	bitwise OR	`a | b`	the bitwise OR of **a** and **b**
`^`	bitwise XOR	`a ^ b`	the bitwise XOR of **a** and **b**
`<<`	bitwise left shift	`a << b`	**a** left shifted by **b**
`>>`	bitwise right shift	`a >> b`	**a** right shifted by **b**

Bitwise operators in C

bit a	bit b	a & b (a AND b)
0	0	0
0	1	0
1	0	0
1	1	1

bit a	bit b	a I b (a OR b)
0	0	0
0	1	1
1	0	1
1	1	1

bit a	bit b	a ^ b (a XOR b)
0	0	0
0	1	1
1	0	1
1	1	0

~a (NOT a) is trivial

Examples

	1	0	1	1	1	0	0	1
&	0	1	1	1	0	1	1	0
	0	0	1	1	0	0	0	0

	1	0	1	1	1	0	0	1
|	0	1	1	1	0	1	1	0
	1	1	1	1	1	1	1	1

	1	0	1	1	1	0	0	1
^	0	1	1	1	0	1	1	0
	1	1	0	0	1	1	1	1

~	0	1	1	1	0	1	1	0
	1	0	0	0	1	0	0	1

Practice

```
~0x102
```

```
0xABC & 0x411
```


Practice

```
0xABC | 0x411
```

```
0x102030 & 0x00FF00
```


Shift operations

- Left shift (x << y)
  - shifts each bit in **x** to the left by **y** positions
    - discards **y** bits on the left
  - fills **y** blank spaces on the right with zeros

1	0	1	1	1	0	0	1	<< 2
1	1	1	0	0	1	0	0	

Shift operations

- Right shift (x >> y)
  - shifts each bit in **x** to the right by **y** positions
    - discards **y** bits on the right

Logical shift:
fill blank spaces on left with zeroes

Arithmetic shift:
fill blank spaces by replicating original MSB (most compilers implement it — preserves sign bit)

1	0	1	1	1	0	0	1	>> 2
0	0	1	0	1	1	1	0	

logical

MSB

1	0	1	1	1	0	0	1	>> 2
1	1	1	0	1	1	1	0	

arithmetic

Practice

```
0xF3 << 2

0x9A >> 3    (logical)

0x9A >> 3    (arithmetic)
```


Example: bit masking

- Assume an unsigned integer j that stores the value 0x1A35B127
  - define a mask to extract the most significant byte
  - write C code to store the extracted value in another variable (unsigned int)

Example: bit masking

- Assume an integer j that stores the value 0x1A35B127
  - write C code to set the least significant byte of j to all ones leaving all other bytes unchanged

Practice

- Consider a genomic database
  - four DNA bases: Adenine (A), Cytosine (C), Thymine (T), and Guanine (G)
  - estimate the size in bytes of a text file storing a database of 100,000,000 DNA bases, assuming each base is represented as a single character (`char`)
- Determine the minimum number of bits required to uniquely represent each base
  - write a possible encoding, mapping each base to a specific bit pattern
- Assume DNA sequences are stored as integers (4 bytes)
  - calculate the maximum number of DNA bases that can be represented within a single integer
  - given the integer value 0x10012001, assuming your encoding, decode the corresponding DNA sequence
  - estimate the size in bytes of a binary file storing a database of 100,000,000 DNA bases

Encoding sets

- Arrays can be inefficient for storing sets, especially when many elements are absent
  - use bits to represent membership, each bit corresponding to a unique object
- Example:
  - consider a set of 8 objects, a char variable, can represent all possible subsets

1	0	0	0	1	0	0	1
Mercury	Venus	Earth	Mars	Jupiter	Saturn	Uranus	Neptune

- Questions
  - how to add, remove, or flip individual objects from the set?
  - how to check whether an object is in the set?
  - how to perform intersection, union, symmetric difference, and complement?

Show me the code

```c
enum Planet { MERCURY, VENUS, EARTH, MARS, JUPITER,
              SATURN, URANUS, NEPTUNE, NUM_PLANETS };

int main() {
    char planets = 0;

    planets = add_planet(planets, EARTH);
    planets = add_planet(planets, MARS);
    planets = add_planet(planets, JUPITER);
    print_set(planets);

    planets = remove_planet(planets, MARS);
    print_set(planets);

    planets = flip_planet(planets, SATURN);
    print_set(planets);

    return 0;
}
```


Show me the code

```c
char add_planet(char set, enum Planet planet) {
    return set | (1 << planet);
}

char remove_planet(char set, enum Planet planet) {
    return set & ~(1 << planet);
}

char flip_planet(char set, enum Planet planet) {
    return set ^ (1 << planet);
}

bool is_in_set(char set, enum Planet planet) {
    return (set & (1 << planet)) != 0;
}

void print_set(char set) {
    const char* planet_names[] = {"Mercury", "Venus", "Earth",
        "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"};
    printf("Set: ");
    for (enum Planet p = 0 ; p < NUM_PLANETS ; p++) {
        if (is_in_set(set, p)) {
            printf(" %s", planet_names[p]);
        }
    }
    printf("\n");
}
```

Bitwise vs logical operators in C

- Bitwise operators
  - operate on individual bits of integer values
  - operators: `&, |, ^, ~, <<, >>`
- Logical operators
  - operate on boolean values (true or false)
  - return a boolean value (true or false)
  - operators: `!, &&, ||`

any non-zero value is considered true, zero is false

Practice

```
!0xF3

!0x00

!!0xF3

~0xF3

0xF3 && 0xF1

0xF3 || 0xF1

0xF3 & 0xF1
```