# Bitwise Manipulation

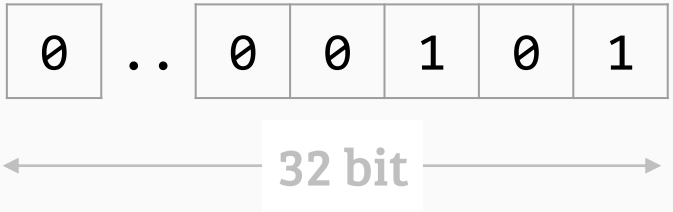
- Low Level Programming
- Set/Reset/Toggle Bits
- Extraction of Bits
- Bit Fields in C

Md. Saidul Hoque Anik
onix.hoque.mist@gmail.com

# Values are stored as bits

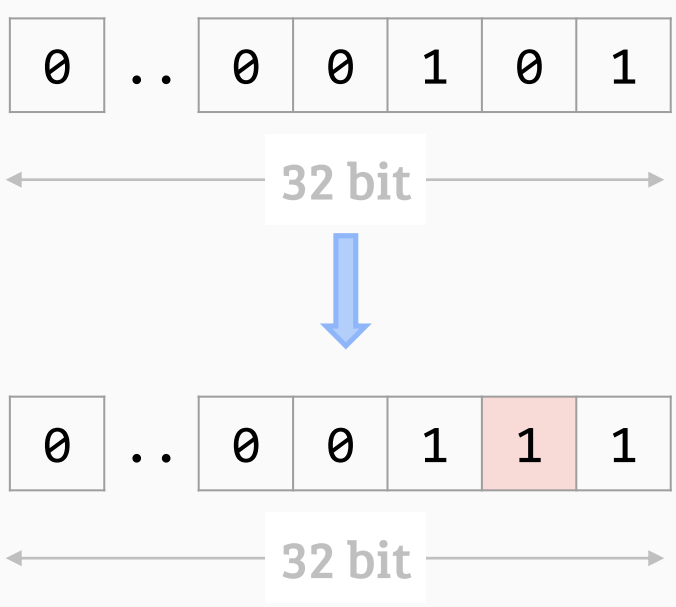
```
int x = 5;
```

**RAM**



# Our goal is to modify the bits

```
int x = 5;
```




```
//value of x is now 7
```

# Bitwise operators (Binary operators)

AND(&):

| A | B | A & B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

1 if **both** are 1

# Bitwise operators (Binary operators)

AND(&):

| A | B | A & B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

1 if **both** are 1

OR(|):

| A | B | A \| B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1 if **any** is 1

# Bitwise operators (Binary operators)

AND(&):

| A | B | A & B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

1 if **both** are 1

OR(|):

| A | B | A \| B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1 if **any** is 1

XOR(^):

| A | B | A ^ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

1 if both are **different**

# Bitwise operators (Binary operators)

Left Shift Operator(<<):

```
int x = 5;    // 0000 0101
x = x << 1;   // 0000 1010
//Bits are shifted 1 bit to the left,
//Filled with zero

x = x << 2;   // 0010 1000
```

# Bitwise operators (Binary operators)

Right Shift Operator(>>):

```
int x = 5;    // 0000 0101
x = x >> 1;   // 0000 0010
//Bits are shifted 1 bit to the right,
//Filled with zero if unsigned or positive
//if negative, then depends on compiler

x = x >> 1;   // 0000 00001
```

# Bitwise operators (Unary Operator)

NOT(~):

| A | ~A |
|---|---|
| 0 | 1 |
| 1 | 0 |

Toggle the bit

# Try out the operators

```
A = 5 = 0000 0101
B = 3 = 0000 0011


   A & B = ?

   A | B = ?

   A ^ B = ?

    ~A   = ?
```

# Try out the operators

```
A = 5 = 0000 0101
B = 3 = 0000 0011


   A & B = 0000 0001

   A | B = ?

   A ^ B = ?

    ~A   = ?
```

# Try out the operators

```
A = 5 = 0000 0101
B = 3 = 0000 0011


   A & B = 0000 0001

   A | B = 0000 0111

   A ^ B = ?

    ~A   = ?
```

# Try out the operators

```
A = 5 = 0000 0101
B = 3 = 0000 0011


   A & B = 0000 0001

   A | B = 0000 0111

   A ^ B = 0000 0110

    ~A   = ?
```

# Try out the operators

```
A = 5 = 0000 0101
B = 3 = 0000 0011
```

```
A & B = 0000 0001
A | B = 0000 0111
A ^ B = 0000 0110
 ~A   = 1111 1010
 ~B   = 1111 1100
```

Don’t confuse
bitwise operator & | ~
with
logical operator && || !

# The decToBin() function

```
void decToBin(unsigned int n)
{
    int i, arr[8];
    for (i = 7; i>=0; i--)
    {
        arr[i] = n % 2;
        n = n/2;
    }
    for (i = 0; i<4; i++)
        printf("%d", arr[i]);
    printf(" ");
    for (i = 4; i<8; i++)
        printf("%d", arr[i]);
    printf("\n");
}
```

# The modifier function – Set()

```
unsigned int set(unsigned int A, int pos)
//set the ‘pos’th bit of A to 1

   unsigned int A = 5; //0000 0101
   unsigned int res = set(A, 4);
   //res = 0001 0101
```

What do we need to do?

# The modifier function – Set()

```
unsigned int set(unsigned int A, int pos)
//set the ‘pos’th bit of A to 1

   unsigned int A = 5; //0000 0101
   unsigned int res = set(A, 4);
   //res = 0001 0101
```

What do we need to do?

```
          0000 0101
          0001 0000      //How do we make it?
   OR     0001 0101
```

# The modifier function – Set()

```
unsigned int set(unsigned int A, int pos)
{
    unsigned int i = 1;
    i = i << pos;
    unsigned int ret = A | i;
    return ret;
}
```

# The modifier function – Reset()

```
unsigned int reset(unsigned int A, int pos)
//reset the ‘pos’th bit of A to 0

   unsigned int A = 5; //0000 0101
   unsigned int res = reset(A, 2);
   //res = 0000 0001
```

What do we need to do?

# The modifier function – Reset()

```
unsigned int reset(unsigned int A, int pos)
//reset the ‘pos’th bit of A to 0

   unsigned int A = 5; //0000 0101
   unsigned int res = reset(A, 2);
   //res = 0000 0001

What do we need to do?

          0000 0101
          1111 1011     //How do we make it?
   AND    0000 0001
```

# The modifier function – Reset()

```
unsigned int reset(unsigned int A, int pos)
{
    unsigned int i = 1;
    i = i << pos;
    i = ~i;
    unsigned int ret = A & i;
    return ret;
}
```

# The modifier function – Toggle()

```
unsigned int toggle(unsigned int A, int pos)
//toggle the ‘pos’th bit of A
//set if 0, reset if 1
```

```
   unsigned int A = 5; //0000 0101
   unsigned int res = toggle(A, 2);
   //res = 0000 0001
   unsigned int res = toggle(res, 2);
   //res = 0000 0101
```

What do we need to do?

# The modifier function – Toggle()

```
unsigned int toggle(unsigned int A, int pos)
//toggle the ‘pos’th bit of A
//set if 0, reset if 1
```

```
unsigned int A = 5; //0000 0101
unsigned int res = toggle(A, 2);
//res = 0000 0001
```

XOR(^):

| A | B | A ^ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

XOR with 0 to keep unchanged

# The modifier function – Toggle()

```
unsigned int toggle(unsigned int A, int pos)
//toggle the ‘pos’th bit of A
//set if 0, reset if 1
```

```
   unsigned int A = 5; //0000 0101
   unsigned int res = toggle(A, 2);
   //res = 0000 0001
```

XOR(^):

| A | B | A ^ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

XOR with 1 to toggle

# The modifier function – Toggle()

```
unsigned int toggle(unsigned int A, int pos)
//toggle the ‘pos’th bit of A
//set if 0, reset if 1
```

```
   unsigned int A = 5; //0000 0101
   unsigned int res = toggle(A, 2);
   //res = 0000 0001
```

XOR(^):

| A | B | A ^ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

```
          0000 0101
          0000 0100
XOR       0000 0001
```

# The modifier function – Toggle()

```
unsigned int toggle(unsigned int A, int pos)
{
    unsigned int i = 1;
    i = i << pos;
    //i = ~i;
    unsigned int ret = A ^ i;
    return ret;
}
```

# The modifier function – check()

```
int check(unsigned int A, int pos)
//1 if ‘pos’th bit of A is 1
//0 otherwise
```

```
   unsigned int A = 5; //0000 0101
   res = check(A, 2);
   //res = 1
   res = check(A, 1);
   //res = 0
```

What do we need to do?

# The modifier function – check()

```
int check(unsigned int A, int pos)
//1 if ‘pos’th bit of A is 1
//0 otherwise
```

```
   unsigned int A = 5; //0000 0101
   res = check(A, 2);
   //res = 1
   res = check(A, 1);
   //res = 0
```

What do we need to do?

We’ll need AND operator

# The modifier function – check()

```
int check(unsigned int A, int pos)
{
    unsigned int i = 1;
    i = i << pos;
    unsigned int ret = A & i;
    ret = ret >> pos;
    return ret;
}
```

Implement the following functions:

```
1) unsigned change(unsigned int A, int pos, int val)
        //set 'pos'th bit of A to val
        //change(0000 0101, 4, 1) = 0001 0101

2) int sum(unsigned int A)
        //return the sum of all 8 bits

3) int parity(unsigned int A)
        //return 1 if even number of bits are set
        //0 otherwise
        //parity(0000 0101) = 1
        //parity(0100 0101) = 0

4) unsigned extract(unsigned int A, int from, int to)
        //extract bits from 'from' to 'to'
        //extract(00000101, 2, 4) = 001
```

# Bit-Fields in C

```c
#include <stdio.h>
struct date
{
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

int main()
{
    printf("Size of date is %d bytes\n",
        sizeof(struct date));
    struct date d1 = {2, 10, 2017};
    printf("Date is %d/%d/%d",
        d1.day, d1.month, d1.year);
}
```

# Notice that,

- Value of day can be at most 31, which requires only 5 bits (2^5 = 32)
- Value of month can be at most 12, which requires only 4 bits (2^4 = 16)

# Bit-Fields in C

```c
struct date
{
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year;
};

//what will be sizeof(struct date)?
```

| Padding (23bit) | day (5bit) | month (4bit) | year (32bit) |
|---|---|---|---|

# Bit-Fields in C

```c
struct date
{
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year;
};

//what will be sizeof(struct date)?
```

# Can we,

- Shrink the total size of `struct date` to 4 byte?
- How about limiting the year up to 4095? How many bits will it require?

# Further study on Bit-Fields:

http://www.geeksforgeeks.org/bit-fields-c/