Bitwise Operations

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Basic Bitwise AND

Program 1:

```c
#include <stdio.h>
int main() {
  unsigned int a = 12;
  unsigned int b = 10;
  unsigned int result = a & b;
  printf("a = %u (binary: 1100)\n", a);
  printf("b = %u (binary: 1010)\n", b);
  printf("a & b = %u (binary: 1000)\n",
    result);
  printf("Decimal: %u\n", result);
  return 0;
}
```

Output:

```
a = 12 (binary: 1100)
b = 10 (binary: 1010)
a & b = 8 (binary: 1000)
Decimal: 8
```

Note:

```
AND: 1 & 1 = 1, else 0.
Bit by bit comparison.
  1100
& 1010
------
  1000
```

Basic Bitwise OR

Program 2:

```c
#include <stdio.h>
int main() {
  unsigned int a = 12;
  unsigned int b = 10;
  unsigned int result = a | b;
  printf("a = %u (binary: 1100)\n", a);
  printf("b = %u (binary: 1010)\n", b);
  printf("a | b = %u (binary: 1110)\n",
    result);
  printf("Decimal: %u\n", result);
  return 0;
}
```

Output:

```
a = 12 (binary: 1100)
b = 10 (binary: 1010)
a | b = 14 (binary: 1110)
Decimal: 14
```

Note:

```
OR: 0 | 0 = 0, else 1.
Combines bits.
  1100
| 1010
------
  1110
```

Basic Bitwise XOR

Program 3:

```c
#include <stdio.h>
int main() {
  unsigned int a = 12;
  unsigned int b = 10;
  unsigned int result = a ^ b;
  printf("a = %u (binary: 1100)\n", a);
  printf("b = %u (binary: 1010)\n", b);
  printf("a ^ b = %u (binary: 0110)\n",
    result);
  printf("Decimal: %u\n", result);
  return 0;
}
```

Output:

```
a = 12 (binary: 1100)
b = 10 (binary: 1010)
a ^ b = 6 (binary: 0110)
Decimal: 6
```

Note:

```
XOR: Different bits = 1, same = 0.
Toggle bits.
  1100
^ 1010
------
  0110
```

Bitwise NOT

Program 4:

```c
#include <stdio.h>
int main() {
  unsigned char a = 5;
  unsigned char result = ~a;
  printf("a = %u (binary: 00000101)\n", a);
  printf("~a = %u (binary: 11111010)\n",
    result);
  unsigned int b = 5;
  printf("~b (int) = %u\n", ~b);
  return 0;
}
```

Output:

```
a = 5 (binary: 00000101)
~a = 250 (binary: 11111010)
~b (int) = 4294967290
```

Note:

```
NOT: Flips all bits. 0 becomes 1,
1 becomes 0. Result depends on type
size. ~5 in 8-bit = 250, in 32-bit
much larger.
```

Program 5:

```c
#include <stdio.h>
int main() {
  unsigned int a = 5;
  printf("a = %u (binary: 0101)\n", a);
  printf("a << 1 = %u (binary: 1010)\n",
    a << 1);
  printf("a << 2 = %u (binary: 10100)\n",
    a << 2);
  printf("a << 3 = %u (binary: 101000)\n",
    a << 3);
  printf("Multiply by 2: %u * 2 = %u\n",
    a, a << 1);
  printf("Multiply by 4: %u * 4 = %u\n",
    a, a << 2);
  return 0;
}
```

Output:

```
a = 5 (binary: 0101)
a << 1 = 10 (binary: 1010)
a << 2 = 20 (binary: 10100)
a << 3 = 40 (binary: 101000)
Multiply by 2: 5 * 2 = 10
Multiply by 4: 5 * 4 = 20
```

Note:

```
Left shift moves bits left, fills
right with 0s. Equivalent to
multiplying by 2^n. Fast operation.
```

Right Shift

Program 6:

```c
#include <stdio.h>
int main() {
  unsigned int a = 40;
  printf("a = %u (binary: 101000)\n", a);
  printf("a >> 1 = %u (binary: 10100)\n",
    a >> 1);
  printf("a >> 2 = %u (binary: 1010)\n",
    a >> 2);
  printf("a >> 3 = %u (binary: 101)\n",
    a >> 3);
  printf("Divide by 2: %u / 2 = %u\n",
    a, a >> 1);
  printf("Divide by 4: %u / 4 = %u\n",
    a, a >> 2);
  return 0;
}
```

Output:

```
a = 40 (binary: 101000)
a >> 1 = 20 (binary: 10100)
a >> 2 = 10 (binary: 1010)
a >> 3 = 5 (binary: 101)
Divide by 2: 40 / 2 = 20
Divide by 4: 40 / 4 = 10
```

Note:

```
Right shift moves bits right, fills
left with 0s (unsigned). Equivalent
to dividing by 2^n. Fast division.
```

Program 7:

```c
#include <stdio.h>
int main() {
  unsigned char flags = 0;
  int bit_position = 3;
  printf("Initial: %u (binary: 00000000)\n",
    flags);
  flags = flags | (1 << bit_position);
  printf("After setting bit %d: %u\n",
    bit_position, flags);
  printf("Binary: 00001000\n");
  flags = flags | (1 << 5);
  printf("After setting bit 5: %u\n",
    flags);
  printf("Binary: 00101000\n");
  return 0;
}
```

Output:

```
Initial: 0 (binary: 00000000)
After setting bit 3: 8
Binary: 00001000
After setting bit 5: 40
Binary: 00101000
```

Note:

```
Set bit: Use OR with mask.
Mask = 1 << position.
Example: Set bit 3 = OR with 00001000.
Doesn't affect other bits.
```

Clearing a Bit

Program 8:

```c
#include <stdio.h>
int main() {
  unsigned char flags = 255;
  int bit_position = 3;
  printf("Initial: %u (binary: 11111111)\n",
    flags);
  flags = flags & ~(1 << bit_position);
  printf("After clearing bit %d: %u\n",
    bit_position, flags);
  printf("Binary: 11110111\n");
  flags = flags & ~(1 << 5);
  printf("After clearing bit 5: %u\n",
    flags);
  printf("Binary: 11010111\n");
  return 0;
}
```

Output:

```
Initial: 255 (binary: 11111111)
After clearing bit 3: 247
Binary: 11110111
After clearing bit 5: 215
Binary: 11010111
```

Note:

```
Clear bit: Use AND with inverted mask.
Mask = ~(1 << position).
Example: Clear bit 3 = AND with
11110111. Sets only that bit to 0.
```

Toggling a Bit

Program 9:

```c
#include <stdio.h>
int main() {
  unsigned char flags = 5;
  int bit_position = 1;
  printf("Initial: %u (binary: 00000101)\n",
    flags);
  flags = flags ^ (1 << bit_position);
  printf("After toggling bit %d: %u\n",
    bit_position, flags);
  printf("Binary: 00000111\n");
  flags = flags ^ (1 << bit_position);
  printf("After toggling bit %d: %u\n",
    bit_position, flags);
  printf("Binary: 00000101\n");
  return 0;
}
```

Output:

```
Initial: 5 (binary: 00000101)
After toggling bit 1: 7
Binary: 00000111
After toggling bit 1: 5
Binary: 00000101
```

Note:

```
Toggle bit: Use XOR with mask.
Mask = 1 << position.
Flips bit: 0 becomes 1, 1 becomes 0.
Toggle twice returns to original.
```

Checking a Bit

Program 10:

```c
#include <stdio.h>
int main() {
  unsigned char flags = 40;
  int i;
  printf("flags = %u (binary: 00101000)\n",
    flags);
  for (i = 7; i >= 0; i--) {
    int bit = (flags >> i) & 1;
    printf("Bit %d: %d\n", i, bit);
  }
  if (flags & (1 << 3)) {
    printf("Bit 3 is SET\n");
  }
  if (!(flags & (1 << 2))) {
    printf("Bit 2 is CLEAR\n");
  }
  return 0;
}
```

Output:

```
flags = 40 (binary: 00101000)
Bit 7: 0
Bit 6: 0
Bit 5: 1
Bit 4: 0
Bit 3: 1
Bit 2: 0
Bit 1: 0
Bit 0: 0
Bit 3 is SET
Bit 2 is CLEAR
```

Note:

```
Check bit: AND with mask, test if
non-zero. Shift right then AND with 1
extracts specific bit.
```

Counting Set Bits

Program 11:

```c
#include <stdio.h>
int count_bits(unsigned int n) {
  int count = 0;
  while (n) {
    count += n & 1;
    n >>= 1;
  }
  return count;
}
int main() {
  unsigned int nums[] = {7, 15, 255, 256};
  int i;
  for (i = 0; i < 4; i++) {
    printf("%u has %d set bits\n",
      nums[i], count_bits(nums[i]));
  }
  return 0;
}
```

Output:

```
7 has 3 set bits
15 has 4 set bits
255 has 8 set bits
256 has 1 set bits
```

Note:

```
Count set bits (population count).
Check LSB with & 1, shift right.
7 = 0111 (3 bits), 255 = 11111111
(8 bits), 256 = 100000000 (1 bit).
```

Swapping without Temp Variable

Program 12:

```c
#include <stdio.h>
int main() {
  int a = 10, b = 20;
  printf("Before: a=%d, b=%d\n", a, b);
  a = a ^ b;
  b = a ^ b;
  a = a ^ b;
  printf("After: a=%d, b=%d\n", a, b);
  unsigned char x = 5, y = 9;
  printf("Before: x=%u, y=%u\n", x, y);
  x ^= y;
  y ^= x;
  x ^= y;
  printf("After: x=%u, y=%u\n", x, y);
  return 0;
}
```

Output:

```
Before: a=10, b=20
After: a=20, b=10
Before: x=5, y=9
After: x=9, y=5
```

Note:

```
XOR swap trick. No temp variable.
a^b^b = a (XOR with same value twice
returns original). Classic bit hack.
Note: doesn't work if pointers same.
```

Checking Power of Two

Program 13:

```c
#include <stdio.h>
int is_power_of_two(unsigned int n) {
  return n && !(n & (n - 1));
}
int main() {
  unsigned int nums[] = {1, 2, 3, 4, 5,
    8, 15, 16, 32, 33, 64};
  int i;
  for (i = 0; i < 11; i++) {
    printf("%u is %s power of 2\n",
      nums[i],
      is_power_of_two(nums[i]) ?
        "a" : "not a");
  }
  return 0;
}
```

Output:

```
1 is a power of 2
2 is a power of 2
3 is not a power of 2
4 is a power of 2
5 is not a power of 2
8 is a power of 2
15 is not a power of 2
16 is a power of 2
32 is a power of 2
33 is not a power of 2
64 is a power of 2
```

Note:

```
Power of 2 has single bit set.
n & (n-1) clears lowest bit.
If 0, was power of 2.
```

Extracting Bit Fields

Program 14:

```c
#include <stdio.h>
unsigned int extract_bits(unsigned int n,
  int pos, int num_bits) {
  unsigned int mask = (1 << num_bits) - 1;
  return (n >> pos) & mask;
}
int main() {
  unsigned int value = 0xABCD;
  printf("Value: 0x%X\n", value);
  printf("Bits 0-3: 0x%X\n",
    extract_bits(value, 0, 4));
  printf("Bits 4-7: 0x%X\n",
    extract_bits(value, 4, 4));
  printf("Bits 8-11: 0x%X\n",
    extract_bits(value, 8, 4));
  printf("Bits 12-15: 0x%X\n",
    extract_bits(value, 12, 4));
  return 0;
}
```

Output:

```
Value: 0xABCD
Bits 0-3: 0xD
Bits 4-7: 0xC
Bits 8-11: 0xB
Bits 12-15: 0xA
```

Note:

```
Extract bit field: shift right to
position, AND with mask. Mask =
(1 << num_bits) - 1. Isolates
specific bits.
```

Bit Flags Example

Program 15:

```c
#include <stdio.h>
#define READ    (1 << 0)
#define WRITE   (1 << 1)
#define EXECUTE (1 << 2)
#define ADMIN   (1 << 3)
int main() {
  unsigned int permissions = 0;
  permissions |= READ | WRITE;
  printf("Has READ: %s\n",
    (permissions & READ) ? "Yes" : "No");
  printf("Has EXECUTE: %s\n",
    (permissions & EXECUTE) ? "Yes" : "No");
  permissions |= EXECUTE;
  printf("Has EXECUTE: %s\n",
    (permissions & EXECUTE) ? "Yes" : "No");
  permissions &= ~WRITE;
  printf("Has WRITE: %s\n",
    (permissions & WRITE) ? "Yes" : "No");
  return 0;
}
```

Output:

```
Has READ: Yes
Has EXECUTE: No
Has EXECUTE: Yes
Has WRITE: No
```

Note:

```
Bit flags for permissions. Each bit
represents different flag. Combine
with OR, check with AND, clear with
AND NOT. Space efficient.
```

Reversing Bits

Program 16:

```c
#include <stdio.h>
unsigned char reverse_bits(
  unsigned char n) {
  unsigned char result = 0;
  int i;
  for (i = 0; i < 8; i++) {
    result <<= 1;
    result |= (n & 1);
    n >>= 1;
  }
  return result;
}
int main() {
  unsigned char nums[] = {1, 5, 128, 170};
  int i;
  for (i = 0; i < 4; i++) {
    printf("%u reversed = %u\n",
      nums[i], reverse_bits(nums[i]));
  }
  return 0;
}
```

Output:

```
1 reversed = 128
5 reversed = 160
128 reversed = 1
170 reversed = 85
```

Note:

```
Reverse bit order. Extract LSB,
shift result left, add bit, shift
input right. 00000001 becomes
10000000. 10101010 becomes 01010101.
```

Parity Check

Program 17:

```c
#include <stdio.h>
int parity(unsigned int n) {
  int p = 0;
  while (n) {
    p ^= (n & 1);
    n >>= 1;
  }
  return p;
}
int main() {
  unsigned int nums[] = {7, 15, 255, 256};
  int i;
  for (i = 0; i < 4; i++) {
    printf("%u has %s parity\n",
      nums[i],
      parity(nums[i]) ? "odd" : "even");
  }
  return 0;
}
```

Output:

```
7 has odd parity
15 has even parity
255 has even parity
256 has odd parity
```

Note:

```
Parity: XOR all bits. Odd parity if
odd number of 1s. 7 = 0111 (3 ones),
15 = 1111 (4 ones). Used for error
detection.
```

Bit Manipulation for RGB

Program 18:

```c
#include <stdio.h>
unsigned int make_color(unsigned char r,
  unsigned char g, unsigned char b) {
  return (r << 16) | (g << 8) | b;
}
void extract_color(unsigned int color,
  unsigned char *r, unsigned char *g,
  unsigned char *b) {
  *r = (color >> 16) & 0xFF;
  *g = (color >> 8) & 0xFF;
  *b = color & 0xFF;
}
int main() {
  unsigned int color = make_color(
    255, 128, 64);
  unsigned char r, g, b;
  printf("Color: 0x%06X\n", color);
  extract_color(color, &r, &g, &b);
  printf("R:%u G:%u B:%u\n", r, g, b);
  return 0;
}
```

Output:

```
Color: 0xFF8040
R:255 G:128 B:64
```

Note:

```
Pack RGB into single int. R in bits
16-23, G in 8-15, B in 0-7.
Common in graphics. Extract with
shift and mask.
```

Finding Lowest Set Bit

Program 19:

```c
#include <stdio.h>
int lowest_set_bit(unsigned int n) {
  if (n == 0) return -1;
  return n & -n;
}
int position_lowest_bit(unsigned int n) {
  int pos = 0;
  if (n == 0) return -1;
  while (!(n & 1)) {
    n >>= 1;
    pos++;
  }
  return pos;
}
int main() {
  unsigned int nums[] = {12, 16, 20, 255};
  int i;
  for (i = 0; i < 4; i++) {
    printf("%u: lowest bit value=%u pos=%d\n",
      nums[i], lowest_set_bit(nums[i]),
      position_lowest_bit(nums[i]));
  }
  return 0;
}
```

Output:

```
12: lowest bit value=4 pos=2
16: lowest bit value=16 pos=4
20: lowest bit value=4 pos=2
255: lowest bit value=1 pos=0
```

Note:

```
n & -n isolates lowest set bit.
Two's complement magic. 12 = 1100,
-12 = 0100 (in two's complement),
12 & -12 = 0100 = 4.
```

Brian Kernighan's Algorithm

Program 20:

```c
#include <stdio.h>
int count_set_bits_fast(unsigned int n) {
  int count = 0;
  while (n) {
    n &= (n - 1);
    count++;
  }
  return count;
}
int main() {
  unsigned int nums[] = {7, 15, 255,
    1023, 65535};
  int i;
  for (i = 0; i < 5; i++) {
    printf("%u has %d set bits\n",
      nums[i],
      count_set_bits_fast(nums[i]));
  }
  return 0;
}
```

Output:

```
7 has 3 set bits
15 has 4 set bits
255 has 8 set bits
1023 has 10 set bits
65535 has 16 set bits
```

Note:

```
n & (n-1) clears lowest set bit.
Loop runs once per set bit, not once
per total bit. More efficient than
checking each bit. Classic algorithm.
```