C Programming
Bit Masking

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

Manipulating bit patterns

- In 32-bit CPUs, registers are 32 bits wide
- ... so working with individual bytes still means using 32-bit registers

- C provides facilities for low level bit manipulation
- **Bitwise** and **shift** operations
- Hexadecimal representations of bit patterns

Logical operations

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

- x **&** y AND, 1 if both 1
  - Use 0 in y to set to 0
- x **|** y OR, 1 if either 1
  - Use 1 in y to set to 1
- x **^** y XOR (exclusive OR), 1 if different
  - Use 1 in y to flip x from 0 to 1, or 1 to 0

Bitwise operations

- The **&**, **|** and **^** bitwise operators apply in parallel across all bits within an integer
- e.g.

```
int x = 0x55;
int y = 0xaa;
x | y    → 0xff
```

x	…	0	1	0	1	0	1	0	1
y	…	1	0	1	0	1	0	1	0
x|y	…	1	1	1	1	1	1	1	1

- **~** is bitwise NOT

Masks

- A **mask** is a bit pattern with 1s in the bits we're interested in
- e.g. to select bytes from a 32-bit register, using &

```
#define byte0 0x000000ff
   0000 0000 0000 0000 0000 0000 1111 1111
#define byte1 0x0000ff00
   0000 0000 0000 0000 1111 1111 0000 0000
#define byte2 0x00ff0000
   0000 0000 1111 1111 0000 0000 0000 0000
#define byte3 0xff000000
   1111 1111 0000 0000 0000 0000 0000 0000
```

Masks

```
int word = 0x61626364;
   0110 0001 0110 0010 0110 0011 0110 0100

word & byte0
   0000 0000 0000 0000 0000 0000 0110 0100
word & byte1
   0000 0000 0000 0000 0110 0011 0000 0000
word & byte2
   0000 0000 0110 0010 0000 0000 0000 0000
word & byte3
   0110 0001 0000 0000 0000 0000 0000 0000
```

`masking.c`

Masking bytes

Masking bytes: high to low

```
#define BYTE0 0x000000ff
#define BYTE1 0x0000ff00
#define BYTE2 0x00ff0000
#define BYTE3 0xff000000
int main(int argc, char *argv[])
{
    int word = 0x61626364;
    printf("%8x\n", word & BYTE0);
    printf("%8x\n", word & BYTE1);
    printf("%8x\n", word & BYTE2);
    printf("%8x\n", word & BYTE3);
    return 0;
}
```

```
$ ./masking
      64
    6300
  620000
61000000
```

Shifting

- **Shifting** moves bits a specified number of places left/right within a word
  - Bits that fall off the end are lost
  - New bits are filled with 0
    (or with the sign bit, for >> on signed integers)

- x **<<** y – shift x left y places, and add y 0s to right
- x **>>** y – shift x right y places, and add y 0s to left

Selecting bytes: low to high

```
int word = 0x61626364;
  0110 0001 0110 0010 0110 0011 0110 0100
```

- To extract the least significant byte...

```
word & 0xff                          → 0x64
  0000 0000 0000 0000 0000 0000 0110 0100
```

Selecting bytes: low to high

- `int word = 0x61626364;`

```
0110 0001 0110 0010 0110 0011 0110 0100
```

- To extract the second byte...

```
word >> 8                                  → 0x00616263
  0000 0000 0110 0001 0110 0010 0110 0011
(word >> 8) & 0xFF                         → 0x63
  0000 0000 0000 0000 0000 0000 0100 0011
```

Shifting inserts 0s

Selecting bytes: low to high

- `int word = 0x61626364;`

```
0110 0001 0110 0010 0110 0011 0110 0100
```

- To extract the third byte...

```
word >> 16                          → 0x00006162
   0000 0000 0000 0000 0110 0001 0110 0010
(word >> 16) & 0xFF                 → 0x62
   0000 0000 0000 0000 0000 0000 0100 0010
```

Selecting bytes: low to high

- `int word = 0x61626364;`

```
0110 0001 0110 0010 0110 0011 0110 0100
```

- To extract the most significant byte...

```
word >> 24                          → 0x00000061
0000 0000 0000 0000 0000 0000 0110 0001
(word >> 24) & 0xFF                 → 0x61
0000 0000 0000 0000 0000 0000 0100 0001
```

Selecting arbitrary bits

- `int word = 0x61626364;`

```
0110 0001 0110 0010 0110 0011 0110 0100
```

- So generally, to extract N bits that are M bits from the right... (N=5, M=17)

```
word >> M                                   (shift by M bits)
0000 0000 0000 0000 0011 0000 1011 0001
((1 << N) - 1)                    (make a mask of N bits)
0000 0000 0000 0000 0000 0000 0001 1111
(word >> M) & ((1 << N) - 1)                      (and)
0000 0000 0000 0000 0000 0000 0001 0001
```

`unpack1.c`

Unpacking bytes

Selecting bytes: low to high

```
#define BYTE0 0x000000ff

int main(int argc, char *argv[])
{
    int word = 0x61626364;

    for (int i = 0; i < 4; i++) {
        printf("%8x\n", word & BYTE0);
        word = word >> 8;
    }

    return 0;
}
```

```
$ ./unpack1
      64
      63
      62
      61
```

Printing in binary

- To print a 32-bit integer n in binary
- Repeat 32 times:
  - Mask n with 1 to get the 1st bit
  - Shift n right by 1
- If we do this to print 18, it prints the bits in low to high order: 010010...
- (But we probably want high to low: ...010010)

n	binary	1st bit
18	1001**0**	0
9	100**1**	1
4	10**0**	0
2	1**0**	0
1	**1**	1
0	**0**	0
...	...	...

`bits1.c`

Binary print, low to high

Selecting bits: low to high

```
int main(int argc, char *argv[])
{
    unsigned int n;
    printf("enter value> ");
    scanf("%u", &n);

    for (int i = 0; i < 32; i++)
        printf("%3d", i);
    putchar('\n');

    for (int i = 0; i < 32; i++) {
        printf("%3u", n & 1);
        n = n >> 1;
    }
    putchar('\n');

    return 0;
}
```

```
enter value> 247
  0  1  2  3  4  5  6  7...
  1  1  1  0  1  1  1  1...
```

Selecting bits: low to high

```
int main(int argc, char *argv[])
{
    unsigned int n;
    printf("enter value> ");
    scanf("%u", &n);

    for (int i = 0; i < 32; i++)
        printf("%3d", i);
    putchar('\n');

    for (int i = 0; i < 32; i++) {
        printf("%3u", n & 1);
        n = n >> 1;
    }
    putchar('\n');

    return 0;
}
```

```
enter value> 247
  0   1   2   3   4   5   6   7...
  1   1   1   0   1   1   1   1...
```

`unsigned int`
not the default signed int;
we need all 32 bits!

%u format code
for `printf/scanf`

`bits2.c`

Binary print, high to low

Selecting bits: low to high

```
int main(int argc, char *argv[])
{
    unsigned int n;
    printf("enter number> ");
    scanf("%u", &n);

    for (int i = 31; i > 0; i--)
        printf("%3d", i);
    putchar('\n');

    for (int i = 0; i < 31; i++) {
        printf("%3u", n >> 31);
        n = n << 1;
    }
    putchar('\n');

    return 0;
}
```

```
enter number> 247
 ...7  6  5  4  3  2  1  0
 ...1  1  1  1  0  1  1  1
```

This wouldn't work with signed int (it'd print -1)