Bitwise Operations

- Many situation, need to operate on the bits of a data word –
  - Register inputs or outputs
  - Controlling attached devices
  - Obtaining status
- Corresponding bits of both operands are combined by the usual *logic operations*.
- Apply to all kinds of *integer* types
  - Signed and unsigned
  - char, short, int, long, long long

Bitwise Operations (cont)

- **&** – AND
  - Result is **1** if both operand bits are **1**
- **|** – OR
  - Result is **1** if either operand bit is **1**
- **^** – Exclusive OR
  - Result is **1** if operand bits are different
- **~** – Complement
  - Each bit is reversed
- **<<** – Shift left
  - Multiply by 2
- **>>** – Shift right
  - Divide by 2

Examples

a	1	1	1	1	0	0	0	0
b	1	0	1	0	1	0	1	0

NOTE: when signed → all the same
FYI: integers are really 32 bits so what is the "real" value?
~a has preceding 1's and a<<2 is 0x 3c0

```
unsigned int c, a, b;
   c = a & b;                // 1010 0000
   c = a | b;                // 1111 1010
   c = a ^ b;                // 0101 1010
   c = ~a                    // 0000 1111
   c = a << 2;               // 1100 0000
   c = a >> 3;               // 0001 1110
```

Bitwise AND/OR

char x = 'A';
tolower(x) returns 'a'… HOW?

char y = 'a';
toupper(y) returns 'A'… HOW?

'A' = 0x41 = 0100 0001
'a' = 0x61 = 0110 0001

"mask" = 0010 0000
Use OR

'A' = 0100 0001
<u>mask = 0010 0000</u> |
'a' 0110 0001

"mask" = 1101 1111
Use AND

'a' = 0110 0001
<u>mask = 1101 1111</u> &
'A' 0100 0001

Notice the masks are complements of each other
TRY: char digit to a numeric digit

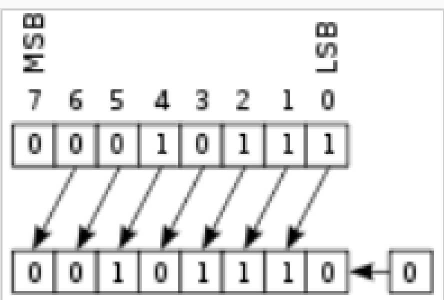
Bitwise XOR

- The bitwise XOR may be used to invert selected bits in a register (toggle)
- XOR as a short-cut to setting the value of a register to zero

0100 0010
<u>0000 1010</u> XOR (toggle)
0100 1000

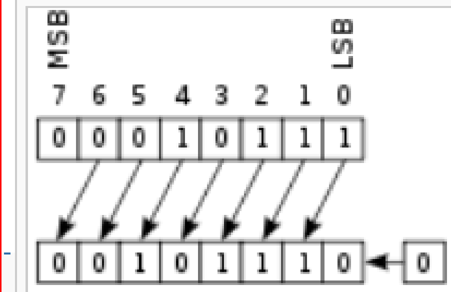
Bitwise left/right shifts

- Possible overflow issues
- Exact behavior is implementation dependent

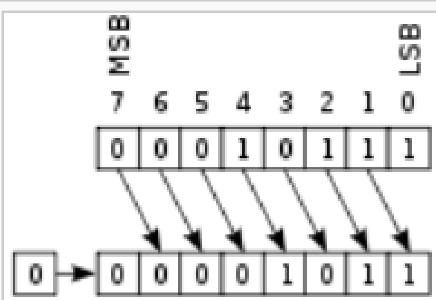


Left logical shift

When you shift left by **k** bits == **multiplying by 2^K**



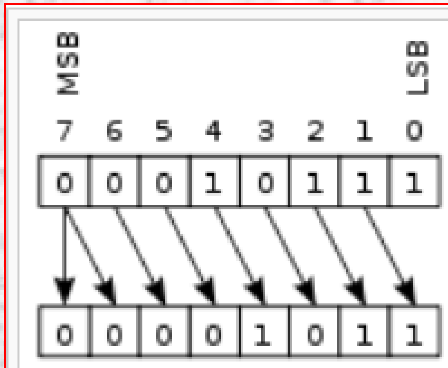
Left arithmetic shift



Right logical shift

When you shift right by **k** bits == **dividing by 2^K**

***** If it's signed, then it's*** implementation dependent.**



Right arithmetic shift

Bitwise right shifts

a	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0

b	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

```
unsigned int c, a;
   c = a >> 3;
```

c	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0		0	0	0

```
signed int c, a, b;
   c = b >> 3;
   c = a >> 3;
```

c	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0		1	0	1

c	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0		0	0	0

EXAMPLE: 8-bit instruction format
101 01000 // ADD 8 → ALU adds ACC reg to value at address 8
To get just the instruction i.e. 101… shift right by 5
To get just the address i.e. 01001… shift left by 3, then right by 3

C example…

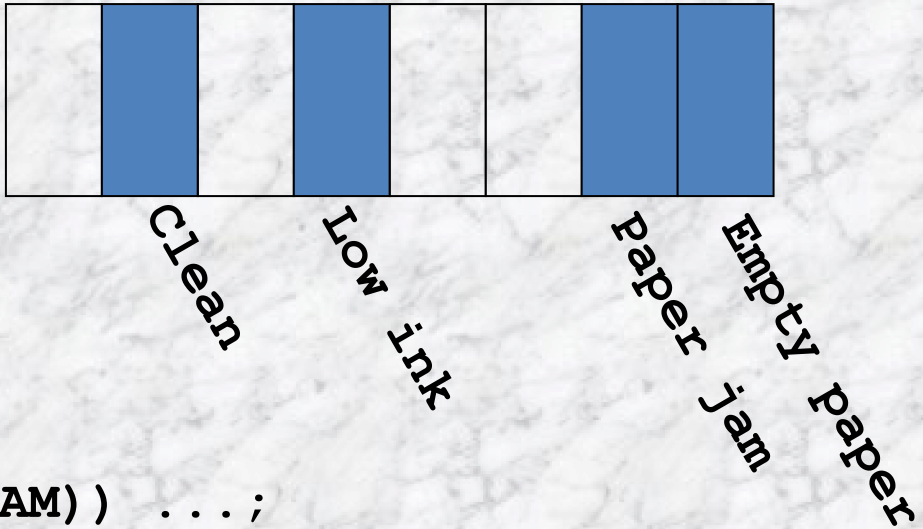
```
#include <stdio.h>
void main()
{
  signed int c, d, a, b, e, f;
  a = 0xF0F0;
  b = 0x5555;
  e = 0b01000001;
  f = 'A';

  c = b >> 3;
  d = a >> 3;

  printf("b >> 3 is %x\n",c);
  printf("a >> 3 is %x\n",d);
  printf("binary = %x\n",e);
  printf("char a = %c",f);
}
```

Output is:

```
b >> 3 is aaa
a >> 3 is 1e1e
binary = 41
char a = A
```

Traditional Bit Definition

8-bit Printer Status Register



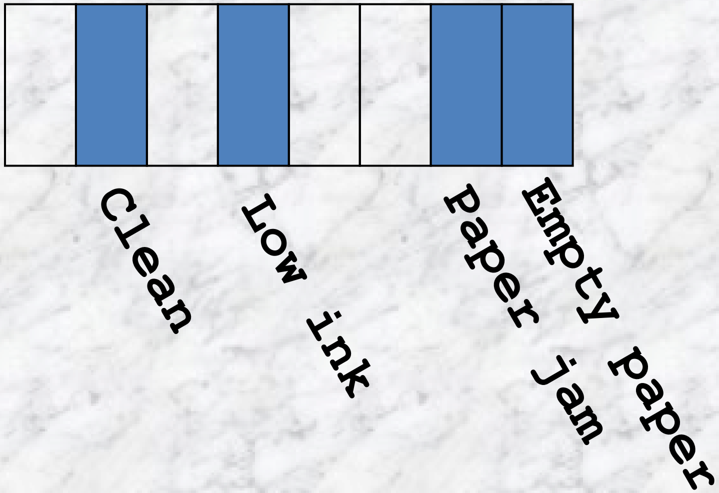

```
#define EMPTY   01
#define JAM     02
#define LOW_INK 16
#define CLEAN   64

char status;
if (status == (EMPTY | JAM)) ...;
if (status == EMPTY || status == JAM) ...;
while (! status & LOW_INK) ...;

int flags |= CLEAN        /* turns on CLEAN bit */
int flags &= ~JAM         /* turns off JAM bit */
```

Traditional Bit Definitions

- Used very widely in *C*
  - Including a *lot* of existing code
- No checking
  - You are on your own to be sure the right bits are set
- Machine dependent
  - Need to know *bit order* in bytes, *byte order* in words
- Integer fields within a register
  - Need to `AND` and shift to extract
  - Need to shift and `OR` to insert

Modern Bit-field Definitions

```
struct statusReg {
   unsigned int empty          :1;
   unsigned int jam            :1;
                               :2;  //???
   unsigned int lowInk         :1;
                               :1;  //???
   unsigned int needsCleaning  :1;
                               :1;  //???
};
struct statusReg s;

if (s.empty && s.jam) ...;
while(! s.lowInk) ...;

s.needsCleaning = true;
s.Jam = false;
```



Conditional Operator

- Consists of two symbols
  - Question mark
  - Colon
- Syntax: exp1 ? exp2 : exp3
- Evaluation:
  - If exp1 is true, then exp2 is the resulting value
  - If exp1 is false, then exp3 is the resulting value
- Example: if a = 10 and b = 15
  - x = (a > b) ? a : b
  - b is the resulting value and assigned to x
  - Parentheses not necessary
  - Similar, but shorter than, if/else statement

Conditional Operator (cont)

- *expr1* ? *expr2* : *expr3*
- In the expression *expr1* ? *expr2* : *Expr3*, the operand *expr1* must be of scalar type. The operands *expr2* and *Expr3* must obey one of the following sets of rules:
- Both of arithmetic type. In this case, both *expr2* and *Expr3* are subject to the usual arithmetic conversions, and the type of the result is the common type resulting from these conversions.
- Both of compatible structure or union types. In this case, the type of the result is the structure or union type of *expr2* and *expr3*.
- Both of void type. In this case, the result is of type void.
- Both of type pointer to qualified or unqualified versions of compatible types. In this case, the type of the result is pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
- One operand of pointer type, the other a null pointer constant In this case, the type of the result is pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
- One operand of type pointer to an object, the other of type pointer to a qualified or unqualified version of void. In this case, the type of the result is that of the non-pointer-to-void operand.
- In all cases, *expr1* is evaluated first. If its value is nonzero (true), then *expr2* is evaluated and *expr3* is ignored (not evaluated at all). If *expr1* evaluates to zero (false), then *expr3* is evaluated and *expr2* is ignored. The result of *expr1* ? *expr2* : *expr3* will be the value of whichever of *expr2* and *expr3* is evaluated.

The Comma Operator

- Used to link related expressions together
- Evaluated from left to right
- The value of the right most expression is the value of the combined expression
- Example:
  - Value = (x = 10, y = 5, x + y);
- Comma operator has lowest precedence
  - Parentheses are necessary!
- For loop:
  - for (n=1, m=10; n<=m; n++, m--)
- While:
  - while (c=getchar(), c!= ‘10’)
- Exchanging values:
  - t=x, x=y, y=t;