CS220 – Logic Design
AS06-Advanced Assembly

• Outline
  - Bit Shifting Instructions
  - Boolean Bitwise Operations
  - Using the Stack
  - Floating Point Numbers
The **shl** and **shr** instructions perform logical left and right shifts respectively. New, incoming bits are always zero. The last bit shifted out is stored in the **CARRY** flag bit.

```assembly
movb $0b10101010, %al  # AL=AAh
shrb $1, %al           # AL=55h (01010101)
shlb $4, %al           # AL=50h (01010000)
movb $2, %cl           # CL
shrb %cl, %al          # AL=14h (00010100)
movl $0xAA00AAA00, xval
shrl $28, xval         # X=00000000Ah
```
The destination operand can be either an 8, 16, or 32 bit register or memory location. The first operand must be either a constant or the CL register. (The shift value is in CL)

Shifting left or right one bit position is equivalent to multiplication or division by two. Shifting is much faster than the equivalent arithmetic operation.

```
shll $3,%eax    # EAX = EAX * 8
```
A logical right shift of a negative number is not equivalent to division by two, because the sign bit is changed from a 1 to a 0.

The `sar` (shift arithmetic right) shifts all bits to the right except the sign bit. New bits that enter on the left are copies of the sign bit.

The `sal` (shift arithmetic left) is an alias for the `shl` instruction.
The **rol** instruction performs a left circular shift left. Bits shifted left from the most significant bit position are shifted into the bit 0 position. The **ror** instruction performs a similar right circular shift.

A copy of the last bit circulated from one end to the other is saved in the CARRY flag.

The **rcl** and **rcr** include the CARRY bit in the circular shift.
AS06-Advanced Assembly Shift Operations

Here is an example that counts the number of ones in the EAX register:

```assembly
movb $0, %bl      # Init one's counter
movl $32, %ecx    # Set loop count

count_loop:
  shll $1, %eax    # shift bit to carry
  jnc skip_inc
  incb %bl         # if (CF == 1) BL++

skip_inc:
  loop count_loop
```
In C++ the shift operators are `<<` (left) and `>>` (right). The shift is **logical/arithmetic** if the variable is unsigned/signed.

```cpp
short s = -256;     // s=FF00h
short unsigned u;
u = 65280;          // u=FF00h
s = s >> 8;         // s=FFFFFFh (-1)
u = u >> 8;         // u=00FFh (+255)
s = s << 3;         // s=FFF8h (-8)
u = u << 3;         // u=07F8h (+2040)
```
The **and** instruction performs a boolean AND operation of its two operands. There are also **or** and **xor** instructions. The **not** instruction inverts all of the bits in its operand.

```
movw $0x3333,xval
andw $0x00FF,xval   # xval = 0033h
movw $0x5555,%ax
orw xval,%ax        # AX = 5577h
notw %ax             # AX = AA88h
```
The **and**, **or** and **xor** instructions are often used to clear, set, and **complement** bits:

- `movw $0xC123, %ax`
- `orw  $8, %ax  # b3=1, ax=C12Bh`
- `andw $0xFFDF, %ax # b5=0, ax=C10Bh`
- `xorw $0x8000, %ax # b31=!b31, ax=410Bh`
- `orw  $0x0F00, %ax # ax=4F0Bh`
- `andw $0xFFFF0, %ax # ax=4F00h`
- `xorw $0xF00F, %ax # ax=BF0Fh`
- `xorw $0xFFFFF, %ax # ax=40F0h`
AS06-Advanced Assembly
Related Operations

- The **test** instruction does a bitwise **AND** of its two operands. It does not store the result but does set bits in the FLAGS register. Typically the **test** is followed by a conditional jump based on the zero flag (ZF).

- The **setCC** instructions can be used to set an 8-bit operand to 0 or 1 based on the status of a flag bit. This can be used to avoid branches in some cases.
The stack is used for storing temporary variables. In fact, all normal (automatic) C/C++ variables are stored on the stack. (They are temporary in that they are only needed while the function is executed.)

The following slide illustrates where variables in C++ are actually stored (in the data or bss sections or on the stack).
int num1, num2=1;  // In .bss, .data
static int num3;  // In .bss segment
int main() {
    double x, a=0.5; // All on the stack
    static int z=0;  // In .data segment
    // main code here
}

int foo() {
    int x=12, y;       // All on the stack
    static float a;   // In .bss segment
    // foo code here
}
In the standard method for allocating space for local variables on the stack the ESP register is copied to the EBP register and then the ESP register is decremented to allocate all of the necessary space.

Local variables are then always a fixed negative offset from the address in the EBP register.
• Assume our pseudo-code uses the following local variables:

```plaintext
short w;    // 2 bytes
int x;      // 4 bytes
int y;      // 4 bytes
```

• A total of **10 bytes** of space is needed on the stack for local variable storage. Intel recommends that we keep the stack 4-byte aligned so we will allocate 12 bytes.
Using the standard method for allocating space on the stack we have:

```
pushl  %ebp
movl   %esp, %ebp
subl   $12, %esp
movl   $100, -8(%ebp)  # x = 100
movl   %eax, -12(%ebp) # y = EAX
movw   %bx, -4(%ebp)   # w = BX
movl   %ebp, %esp
popl   %ebp
```
The variables are mapped to the following locations on the stack:

- # short w: -4(%ebp)
- # int x: -8(%ebp)
- # int y: -12(%ebp)

With this method, we can reference our local variables without having to recalculate offsets even as other items are pushed on the stack.
The `equ` directive can be used to assign a symbol to a particular offset on the stack:

```
.equ w, -4   # short
.equ x, -8   # int
.equ y, -12  # int
```

This allows our assembly to look like this:

```
movl %eax, x(%ebp)
movw $50, w(%ebp)
```
You are strongly encouraged to avoid the use of global variables (variables in the `data` and `bss` sections) in assembly.

Try to write assembly programs that only store constant strings in the `data` section.

It should be mentioned that in addition to the `data`, `bss`, and `stack` areas, data can also be stored in `heap` memory. `Heap` memory is allocated at runtime using `new` (or `malloc`).
A binary point number is similar to a decimal point number except that all weights are powers of two instead of ten:

$$110.011_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 4 + 2 + 0 + 0 + 0.25 + 0.125$$

$$= 6.375_{10}$$

A decimal fraction is converted to a binary fraction by repeatedly multiplying by two and keeping the integer portion of the product.
For example, consider $6.375_{10}$, the integer portion (6) is equal to $110_2$. We can convert the fraction (0.375) as follows:

\[
\begin{align*}
0.375 \times 2 &= 0.75 & \text{first bit} &= 0 \\
0.75 \times 2 &= 1.5 & \text{second bit} &= 1 \\
0.5 \times 2 &= 1.0 & \text{third bit} &= 1
\end{align*}
\]

so $6.375_{10} = 110.011_2$. 
• When $23.85_{10}$ is converted to binary the result is $10111.11011001100110..._2$ which is a repeating binary number. Using a fixed number of bits we can only store an approximation to $23.85_{10}$ as a binary number.

• Computers use a binary form of scientific notation to store floating point numbers:

$$1.sssssssssssssssssssssss x 2^{eeeeeee}$$

where $1.sssssssssssssssssssssssss$ is the significand and $eeeeeee$ is the exponent.
For example, $6.35_{10} (110.011_2)$ would be stored as $1.10011 \times 2^{10}$ (where the 10 in the exponent is in binary, not decimal). Only the significand and exponent are stored.

The IEEE has defined two standard floating point representations. The single precision format uses 32 bits while the double precision format uses 64 bits. Single precision is used by float variables in C++ and double precision by double variables.
In the **single** precision format 1 bit is used for the sign, 8 bits for the exponent, and 23 bits for the fractional portion of the significand. The leading 1 in the significand is not stored.

- **s** sign bit = 0 for positive, 1 for negative
- **e** biased exponent = true exponent + 7F (127) (The values 00 and FF are special.)
- **f** fraction = 23 bits after the 1 in significand
The number $23.85_{10} = 10111.11(0110)_2 = 1.011111(0110) \times 2^{100}$ would be stored as:

```
0100 0001 1011 1110 1100 1100 1100 1101
```

That's 41BECCCD in hex. (The last digit has been rounded up to 1.) The exponent is $10000011_2$ or $4_{10}$ $(100_2) + 127_{10} = 131_{10}$.

If we convert this number back to decimal we get $23.8500003814697265625$
AS06-Advanced Assembly
IEEE Floating Point

- Note that a signed magnitude representation is used for negative numbers, not two's complement.

- When $e$ is 0, the number is either 0 (if $f = 0$) or is a denormalized number. When $e$ is FF, the number represents infinity (if $f = 0$) or is an undefined result (NaN or Not a Number).

- The single precision representation is usually accurate to about 7 significant decimal digits.
Normalized numbers can range in magnitude from $1.0 \times 2^{-126} \approx 1.1755 \times 10^{-35}$ to $1.11111... \times 2^{127} \approx 3.4028 \times 10^{35}$. Denormalized numbers extend the range down to $1.0 \times 2^{-149} \approx 1.40129 \times 10^{-45}$.

IEEE double precision numbers use 11 bits for the exponent (with a bias of $3FF_{16}$ or 1023) and 52 bits for the fraction. They are usually accurate to about 15 decimal digits.
AS06-Advanced Assembly
Floating Point Arithmetic

- Arithmetic operations on numbers in the IEEE floating point format are analogous to operations on decimal numbers in scientific notation.

- Since not all decimal numbers can be exactly represented in a binary format, results are only approximations to the true answers. (Round-off errors in operations can lead to inaccuracies even with numbers that have exact binary representations.)