COM 410 – Slides 4
Representing and Manipulating Information

Some Vocabulary

- Bits
- Bytes
- Overflow
- Memory and virtual memory
- Memory address and virtual address space
- Program objects
- Information type
- Word size
- Big endian vs. little endian
- MSB, LSB

Three types of number representation:

- Unsigned notation
  (binary and hexadecimal notations)

- Two’s Complement
  (positive and negative numbers)

- Floating Point

Digit, Hex, and Binary notations

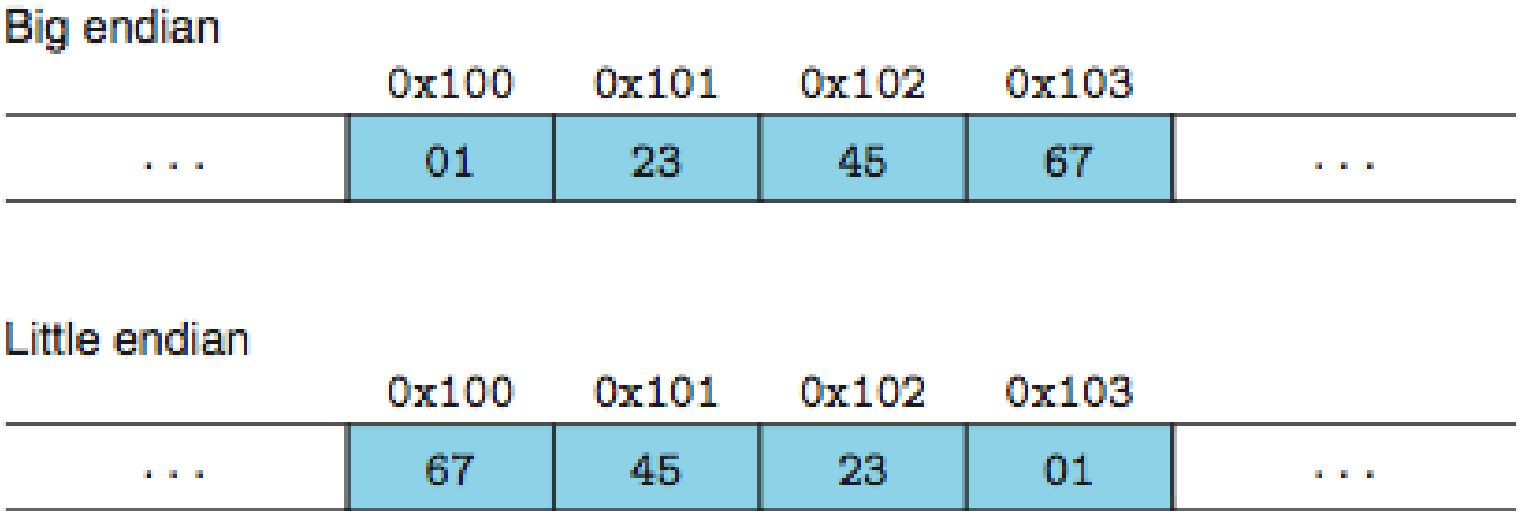
Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Type sizes are not guaranteed!

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

```
typedef long                SLong;
typedef unsigned long       ULong;
typedef short               SWord;
typedef unsigned short      UWord;
typedef char                SByte;
typedef unsigned char       UByte;
//typedef unsigned char      Boolean;
typedef unsigned short      Boolean;
//typedef long double        GFloat;
```

Ox01234567 stored as:



Representations on different machines:

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	`39 30 00 00`
Windows	12,345	`int`	`39 30 00 00`
Sun	12,345	`int`	`00 00 30 39`
Linux 64	12,345	`int`	`39 30 00 00`
Linux 32	12,345.0	`float`	`00 e4 40 46`
Windows	12,345.0	`float`	`00 e4 40 46`
Sun	12,345.0	`float`	`46 40 e4 00`
Linux 64	12,345.0	`float`	`00 e4 40 46`
Linux 32	`&ival`	`int *`	`e4 f9 ff bf`
Windows	`&ival`	`int *`	`b4 cc 22 00`
Sun	`&ival`	`int *`	`ef ff fa 0c`
Linux 64	`&ival`	`int *`	`b8 11 e5 ff ff 7f 00 00`

Boolean Algebra Overview

$$\begin{array}{r} 0110 \\ \& \ 1100 \\ \hline 0100 \end{array} \qquad \begin{array}{r} 0110 \\ | \ 1100 \\ \hline 1110 \end{array} \qquad \begin{array}{r} 0110 \\ \hat{} \ 1100 \\ \hline 1010 \end{array} \qquad \begin{array}{r} \\ \sim \ 1100 \\ \hline 0011 \end{array}$$

Shifting Operations

Operation	Value 1	Value 2
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

Masking Operation

e.g. 0x25AF3255 & 0x00FF0000 = ?

Shifting Operation

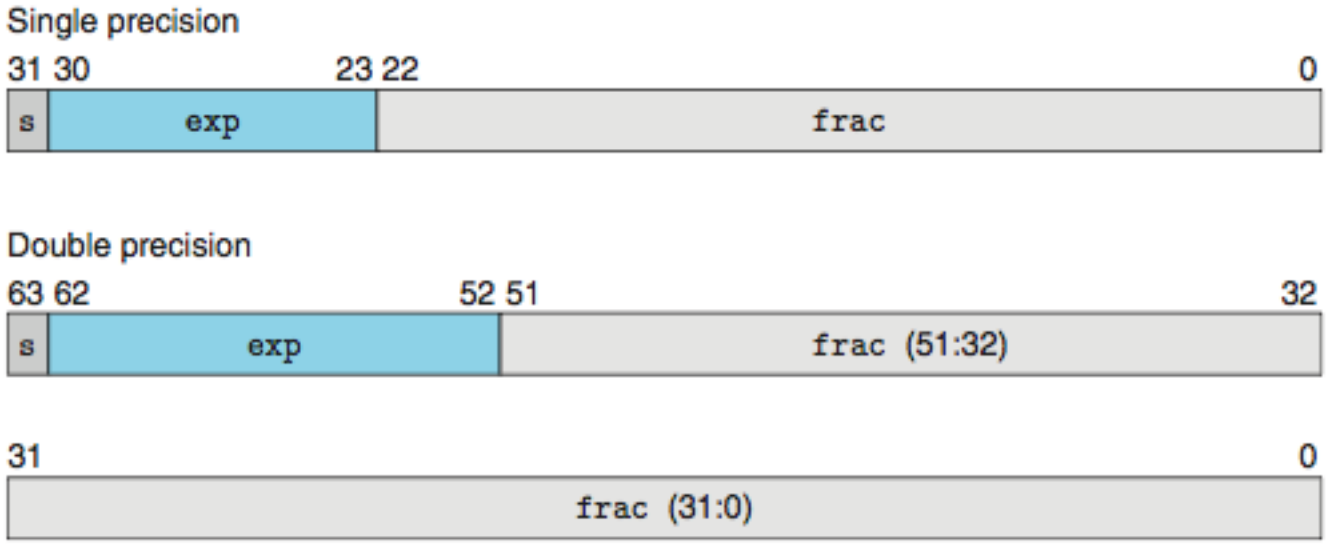
e.g. 0x25AF3255 << 8 = ?

C data type	Minimum	Maximum
`[signed] char`	−127	127
`unsigned char`	0	255
`short`	−32,767	32,767
`unsigned short`	0	65,535
`int`	−32,767	32,767
`unsigned`	0	65,535
`long`	−2,147,483,647	2,147,483,647
`unsigned long`	0	4,294,967,295
`int32_t`	−2,147,483,648	2,147,483,647
`uint32_t`	0	4,294,967,295
`int64_t`	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`uint64_t`	0	18,446,744,073,709,551,615

Two’s compliment
(signed numbers in binary)

Many beginning programmers are surprised to find that adding two positive numbers can yield a negative result, and that the comparison x < y can yield a different result than the comparison x-y < 0. These properties are artifacts of the finite nature of computer arithmetic. Understanding the nuances of computer arithmetic can help programmers write more reliable code.

Floating point



One useful exercise for understanding floating-point representations is to convert sample integer values into floating-point form. For example: 12,345 has binary representation [11000000111001]. We create a normalized representation of this by shifting 13 positions to the right of a binary point, giving 12,345 = 1.10000001110012 × 2^13. To encode this in IEEE single-precision format, we construct the fraction field by dropping the leading 1 and adding 10 zeros to the end, giving binary representation [10000001110010000000000]. To construct the exponent field, we add bias (127 for single precision floats) to 13, giving 140, which has binary representation [10001100]. We combine this with a sign bit of 0 to get the floating-point representation in binary of [01000110010000001110010000000000].

```
     0     0     0     0     3     0     3     9
00000000000000000011000000111001
                          ***********

                  4     6     4     0     E     4     0     0
                01000110010000001110010000000000
```