

CPS104
Computer Organization
Lecture 4
Data representations, Data Types & Memory
Bitwise operations

September 2 , 2009

Gershon Kedem

Administratrivia

- Homework-1 On the Web: <http://kedem.cs.duke.edu/cps104/Homework.html>
- Due **September 7**.
- Submit your homework *in class*, in writing.
 - ✧ **Written homework MUST be stapled!**
- Programming Puzzle-1 is on the Web, see: kedem.cs.duke.edu/cps104/ppz/PPZ1.html
- Due: **September 9** (11:59 pm).
- Submit the ***.java** file to cps104/PPZ1
- Use the Eclipse system to submit the file.
- Help is on-line.

Review: 2's Complement Negation and Addition

- ✱ To negate a number do:
 - ◆ Step 1. complement the digits
 - ◆ Step 2. add 1

Examples

$$\begin{array}{r} 14_{10} = 001110_2 \\ -14_{10} = 110001_2 \\ \quad \quad + 1 \\ \hline 110010_2 \end{array}$$

$$\begin{array}{r} 010010_2 \\ +110010_2 \\ \hline 000100_2 \end{array}$$

- ✱ To add signed numbers use regular addition but disregard carry out
 - ◆ Example $18_{10} - 14_{10} = 18_{10} + (-14_{10}) = 4_{10}$

Review 2's Complement Precision Extension

- Most computers today support 32-bit (int) or 64-bit integers
 - ◆ 64-bit using `gcc` is `long long`
 - ◆ 64-bit using Digital/Compaq compiler is `long`
- To extend precision do `sign bit extension`
 - ◆ precision is number of bits used to represent a number

Example

$14_{10} = 001110_2$ in 6-bit representation.

$14_{10} = 000000001110_2$ in 12-bit representation

$-14_{10} = 110010_2$ in 6-bit representation

$-14_{10} = 111111110010_2$ in 12-bit representation.

Review: Overflow

Example1:

$$\begin{array}{r} 0110101_2 \quad (= 53_{10}) \\ +0101010_2 \quad (= 42_{10}) \\ \hline 1011111_2 \quad (= -33_{10}) \end{array}$$

Example2:

$$\begin{array}{r} 1010101_2 \quad (= -43_{10}) \\ +1001010_2 \quad (= -54_{10}) \\ \hline 0011111_2 \quad (= 31_{10}) \end{array}$$

Example3:

$$\begin{array}{r} 0110101_2 \quad (= 53_{10}) \\ +1101010_2 \quad (= -22_{10}) \\ \hline 0011111_2 \quad (= 31_{10}) \end{array}$$

Example4:

$$\begin{array}{r} 0010101_2 \quad (= 21_{10}) \\ +0101010_2 \quad (= 42_{10}) \\ \hline 0111111_2 \quad (= 63_{10}) \end{array}$$

Review: Floating Point Representation

Numbers are represented by:

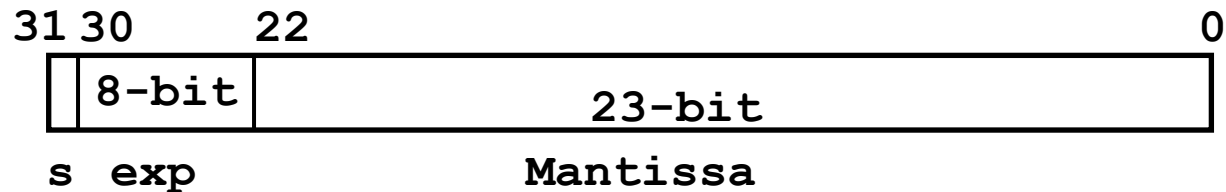
$$Z = (-1)^s \times 2^{E-127} \times 1.M$$

S := 1-bit field ; Sign bit

E := 8-bit field; Exponent: Biased integer, $0 \leq E \leq 255$.

M := 23-bit field; Mantissa: Normalized fraction with hidden 1
(don't actually store it)

Single precision floating point number
uses 32-bits for representation



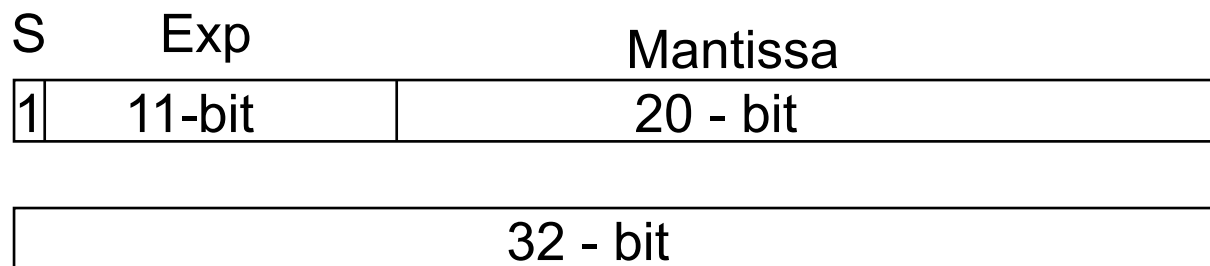
Review: Floating Point Representation

- **Double Precision Floating point:**

64-bit representation: 1-bit **sign**, 11-bit (biased) **exponent**; 52-bit **mantissa** (with hidden 1).

$$X = (-1)^s \times 2^{E-1023} \times 1.M$$

Double precision floating point number



Review: ASCII Character Representation

Hex char

00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL
08 BS	09 HT	0A NL	0B VT	0C NP	0D CR	0E SO	0F SI
10 DLE	11 DC1	12 DC2	13 DC3	14 DC4	15 NAK	16 SYN	17 ETB
18 CAN	19 EM	1A SUB	1B ESC	1C FS	1D GS	1E RS	1F US
20 SP	21 !	22 "	23 #	24 \$	25 %	26 &	27 '
28 (29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [5C \	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C	7D }	7E ~	7F DEL

- Each character is represented by a 7-bit ASCII code.
- It is packed into a byte (8-bits)

Review: Unicode

- **What is Unicode?** *Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.* (unicode.org)
- Unicode is an international character-representation standard for (almost?) all written languages.
- Unicode assigns a unique **16-bit code** to each character.
- Most operating systems support (some) Unicode applications. The transition from ASCII to Unicode is incomplete.
- Java **char** type uses Unicode to represent characters.
- The Unicode representation of English characters extends the ASCII representation by appending a **0x00** byte to a byte holding an ASCII character.
For example: The Unicode representation of **a** is **0x0061**
- To find more information about Unicode see:
<http://www.unicode.org/>

Basic Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length

4 bits is a **nibble**

8 bits is a **byte**

16 bits is a **half-word**

32 bits is a **word**

64 bits is a **double-word**

Character:

ASCII: 7 bit code, **Unicode:** 16-bit code.

Decimal: (BCD code)

digits 0-9 encoded as 0000 thru 1001

two decimal digits packed per 8 bit byte

Integers:

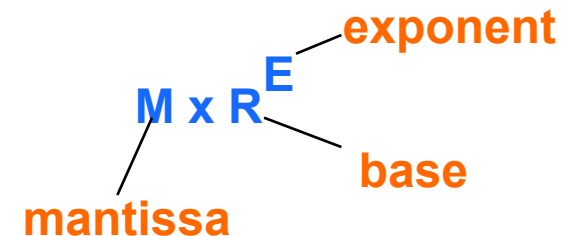
2's Complement (32-bit representation).

Floating Point:

Single Precision (32-bit representation).

Double Precision (64-bit representation).

Extended Precision (128-bit representation).



- How many +/- #'s?
- Where is decimal pt?
- How are +/- exponents represented?

Summary

- **Computers operate on binary numbers (0s and 1s)**
- **Conversion to/from binary, oct, hex**
- **Unsigned Binary numbers.**
- **Signed binary numbers.**
 - ◆ **2's complement.**
 - ◆ **arithmetic, negation.**
- **Floating point representation.**
 - ◆ **hidden 1**
 - ◆ **biased exponent**
 - ◆ **single precision, double precision.**

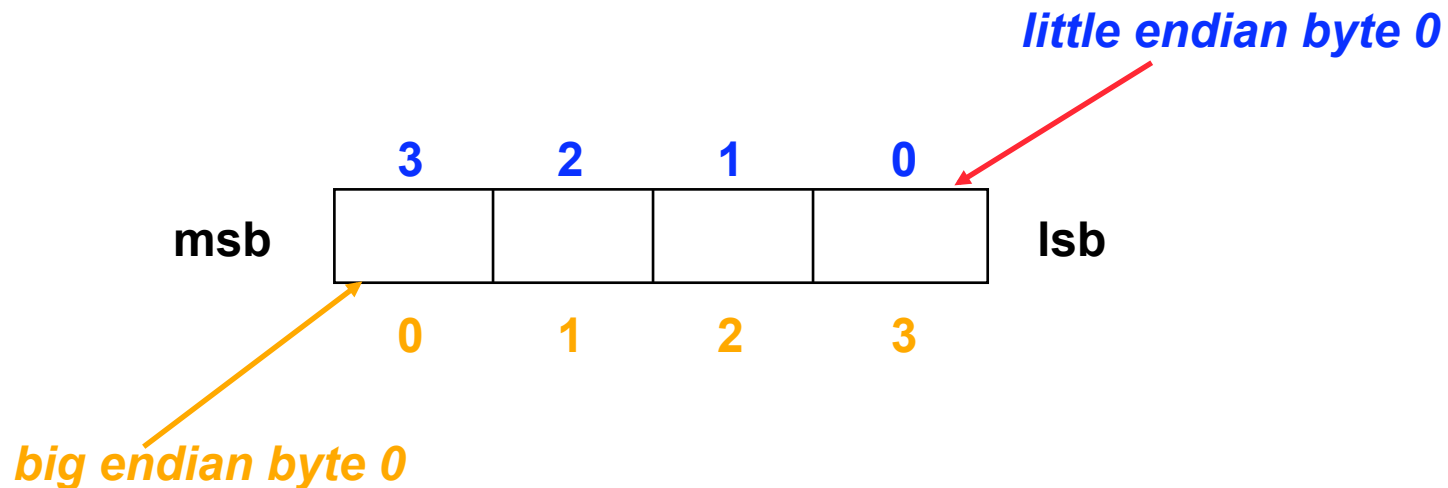
Computer Memory

- What is Computer Memory?
- What does it “look like” to the program?
- How do we find things in computer memory?
- What are **variables** and **where are they stored**?
- What are C style **pointers**, and Java **reference** variables?
- How are more complex data structures (C style structures or Java methods) are represented (stored)?

Buzz Word Definition: Endianness

Byte Order

- **Big Endian:** byte 0 is 8 **most** significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** byte 0 is 8 **least** significant bits Intel X86, DEC Vax, DEC Alpha



Pointers and reference variables

- A pointer is a memory location (variable) that contains the address of another memory location
- C uses “address of” operator `&` and the “dereference” operator `*`
 - ◆ don’t confuse with bitwise **AND** operator (later today)

Given

```
int x; int *p;
```

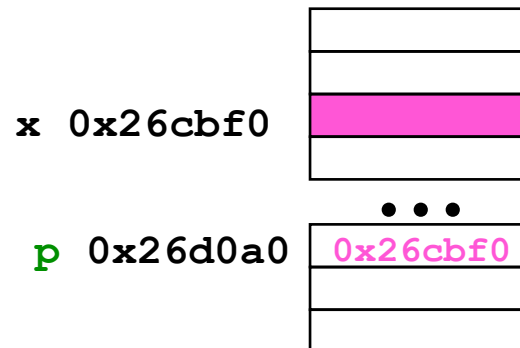
```
p = &x;
```

Then

```
*p = 2; and x = 2; produce the same result
```

On 32-bit machine, p is 32-bits

Please note! Java does NOT support pointers It supports references instead.



Vector Class v.s. Arrays

- **Vector Class**

- ◆ Insulates programmers from underlying data structure.
- ◆ Array bounds checking
- ◆ **Automagically** growing/shrinking when more items are added/deleted

- **How are Vectors implemented?**

- ◆ real understanding comes when more levels of abstraction are understood

- **Programming close to HW**

- ◆ (e.g., operating system, device drivers, etc.)

- **Arrays can be more efficient**

- ◆ but be leery of claims that C-style arrays are required for efficiency

- **Can talk about memory easier in terms of arrays**

- ◆ pointer to a vector?

Arrays

- In C++ and Java allocate arrays using array form of **new**

```
int *a = new int[100];
```

```
double *b = new double[300];
```

- **new []** returns a pointer to a block of memory
 - ◆ how big? where?

- size of chunk can be set at runtime

- **delete [] a;** // storage returned

- In C

```
malloc (nbytes) ;
```

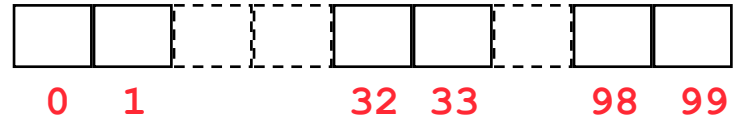
```
free (ptr) ;
```

Address Calculation

- * **x** is a pointer, what is **x+33**?
- * A pointer, but where?
 - ◆ what does calculation depend on?
- * result of adding an int to a pointer depends on size of object pointed to
- * result of subtracting two pointers is an int

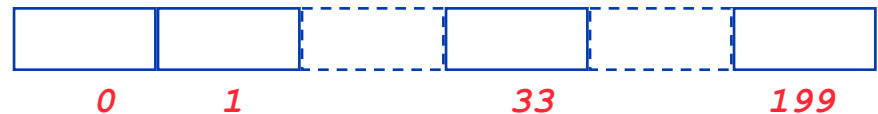
$(d + 3) - d == \underline{\hspace{2cm}}$

```
int * a = new int[100]
```



`a[33]` is the same as `*(a+33)`
if `a` is `0x00a0`, then `a+1` is
`0x00a4`, `a+2` is `0x00a8`
(decimal 160, 164, 168)

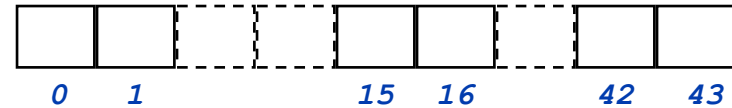
```
double * d = new double[200];
```



`*(d+33)` is the same as `d[33]`
if `d` is `0x00b0`, then `d+1` is
`0x00b8`, `d+2` is `0x00c0`
(decimal 176, 184, 192)

More Pointer Arithmetic

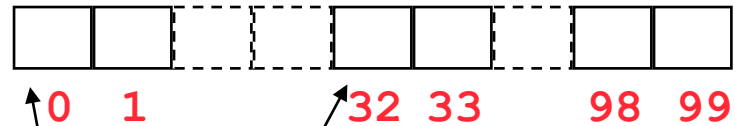
- address one past the end of an array is ok for pointer comparison only
- what's at `* (begin+44)` ?
- what does `begin++` mean?
- how are pointers compared using `<` and using `==` ?
- what is value of `end - begin`?



```
char * a = new char[44];
char * begin = a;
char * end = a + 44;
while (begin < end)
{
    *begin = 'z';
    begin++;
}
```

More Pointers & Arrays

```
int * a = new int[100];
```



a is a pointer

*a is an int

a[0] is an int (same as *a)

a[1] is an int

a+1 is a pointer

a+32 is a pointer

*(a+1) is an int (same as a[1])

*(a+99) is an int

*(a+100) is **trouble**

Array Example (in C++)

```
#include <iostream.h>

main()
{
    int *a = new int[100];
    int *p = a;
    int k;

    for (k = 0; k < 100; k++)
        {
            *p = k;
            p++;
        }

    cout << "entry 3 = " << a[3] <<
endl;

}
```

Array of Classes (Linked List)

```
#include <iostream.h>
class node {
public:
    int me;
    node *next;
};
main()
{
    node *ar = new node[10];
    node *p = ar;
    int k;
    for (k = 0; k < 9; k++)
    {
        p->me = k;
        p->next = &ar[k+1];
        p++;
    }
}
```

```
p->me = 9;
p->next = NULL;
p = &ar[0];
while (p != NULL) {
    cout << p->me << " " <<
    hex << p << " " << p->next <<
    endl;
    p = p->next;
}
}
```

- Given `ar = 0x10000`, what does memory layout look like?

Memory Layout

	Output	
Me	p	p->next
0	0x26ca8	0x26cb0
1	0x26cb0	0x26cb8
2	0x26cb8	0x26cc0
3	0x26cc0	0x26cc8
4	0x26cc8	0x26cd0
5	0x26cd0	0x26cd8
6	0x26cd8	0x26ce0
7	0x26ce0	0x26ce8
8	0x26ce8	0x26cf0
9	0x26cf0	0x0

Memory Address	Memory Contents	Source Symbol
0x26ca8	0	me } ar[0]
	0x26cb0	
0x26cb0	1	next }
	0x26cb8	
0x26cb8	2	me is int (4 bytes)
	0x26cc0	
0x26cc0	3	next is node* (4 bytes)
	0x26cc8	
0x26cc8	4	
	0x26cd0	
0x26cd0	5	
	0x26cd8	
0x26cd8	6	
	0x26ce0	
0x26ce0	7	
	0x26ce8	
0x26ce8	8	
	0x26cf0	
0x26cf0	9	me } ar[9]
	0x0	

Data layout: Example

```
class foo {  
public:  
    int a;  
    char c[3];  
    char *p;  
    foo * next;  
};  
foo x = new foo();
```

How is `x` stored in memory?

If `&X = 0x000f4020` What is `&(x.p)` ?

Strlen()

- **strlen()** returns the # of characters in a string
 - ◆ same as # elements in char array?

```
int strlen(char * s)

// pre: '\0' terminated
// post: returns # chars

{

    int count=0;

    while (*s++)

        count++;

    return count;

}
```

Bit Manipulations

Problem

- **32-bit word contains many values**
 - ◆ e.g., input device, sensors, etc.
 - ◆ current x,y position of mouse and which button (left, mid, right)
- **Assume x, y position is 0-255**
- **How many bits for position?**
- **How many for button?**

Goal

- **Extract position and button from 32-bit word**
- **Need operations on individual bits of binary numbers**

Bitwise AND / OR

- * **&** operator performs bitwise **AND**
- * **|** operator performs bitwise **OR**

* Per bit

$0 \& 0 = 0$	$0 0 = 0$
$0 \& 1 = 0$	$0 1 = 1$
$1 \& 0 = 0$	$1 0 = 1$
$1 \& 1 = 1$	$1 1 = 1$

- * For multiple bits, apply operation to individual bits in same position

AND	OR
011010	011010
<u>101110</u>	<u>101110</u>
001010	111110

Mouse Solution

- **AND** with a **bit mask**
 - ◆ specific values that clear some bits, but pass others through
- To extract x position use mask **0x000ff**

`xpos = 0x1a34c & 0x000ff`

	button		y		x
<code>0x1a34c</code>	<code>= 01</code>	<code>1010</code>	<code>0011</code>	<code>0100</code>	<code>1100</code>
<code>0x000ff</code>	<code>= 00</code>	<code>0000</code>	<code>0000</code>	<code>1111</code>	<code>1111</code>
<code>0x0004c</code>	<code>= 00</code>	<code>0000</code>	<code>0000</code>	<code>0100</code>	<code>1100</code>

More of the Mouse Solution

- To extract y position use mask **0x0ff00**

`ypos = 0x1a34c & 0x0ff00`

- Similarly, button is extracted with mask **0x3ffff**

`button = 0x1a34c & 0x30000`

- Not quite done...why?

		button	y		x
<code>0x1a34c</code>	=	01	1010	0011	0100 1100
<code>0x0ff00</code>	=	00	1111	1111	0000 0000
<code>0x0a300</code>	=	00	1010	0011	0000 0000

The SHIFT operator

- **>>** is shift **right**, **<<** is shift **left**, operands are **int** and number of positions to shift
- **(1 << 3)** is ...000001 -> ...0001000 (it's 2^3)
- **0xff00** is **(0xff << 8)**, and **0xff** is **(0xff00 >> 8)**
- So, true **ypos** value is:

```
ypos = (0x1a34c & 0x0ff00) >> 8
```

```
button = (0x1a34c & 0x30000) >> 16
```

Mask and Shift example: Extracting Parts of Floating Point Number

// x is 32-bit word holding a float

```
#define EXP_BITS 8
#define MANTISSA_BITS 23
#define SIGN_MASK 0x80000000
#define EXP_MASK 0x7f800000
#define MANTISSA_MASK 0x007fffff

class myfloat {
public:
    int sign;
    unsigned int exp;
    unsigned int mantissa;
};

myfloat num;

num->sign = (x & SIGN_MASK) >> (EXP_BITS + MANTISSA_BITS);
num->exp = (x & EXP_MASK) >> MANTISSA_BITS;
num->mantissa = x & MANTISSA_MASK;
```

Summary

- **Computer memory is a linear array of bytes**
- **Pointer is memory location that contains address of another memory location**
- **Bitwise operations**
- **Code examples are linked to course web page**
- **We'll visit these topics again throughout semester.**