Number Systems

- General form of a number in ***base b*** is

$$x = x_n b^n + x_{n-1} b^{n-1} + \dots + x_1 b^1 + x_0 b^0$$
$$+ x_{-1} b^{-1} + \dots + x_{-m} b^{-m}$$

where x_i are the ***positional coefficients***

- Modern computers use binary arithmetic, i.e., base 2

$$140_{10} = 1 \times 10^2 + 4 \times 10^1 + 0 \times 10^0$$
$$= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$
$$= 10001100_2$$
$$= 2 \times 8^2 + 1 \times 8^1 + 4 \times 8^0 = 214_8$$
$$= 8 \times 16^1 + \mathrm{C} \times 16^0 = 8\mathrm{C}_{16}$$

Conversions

- To convert from decimal to binary, divide by 2 repeatedly, read remainders up.

```
2|140
 2|70   0
 2|35   0
 2|17   1
  2|8   1
  2|4   0
  2|2   0
  2|1   0
    0   1
```

```
8|140
 8|17   4
  8|2   1
    0   2
```

- Easier to convert to octal, then to binary

$$140 = \underbrace{10}_{2}\underbrace{001}_{1}\underbrace{100}_{4}, \quad \overbrace{1000}^{8}\overbrace{1100}^{\mathrm{C}}$$

```
          8   C      hex
140 =   10001100     binary
         2  1  4     octal
```


Addition

- Addition in base b

$$x_n b^n + x_{n-1} b^{n-1} + x_{n-2} b^{n-2} + \dots + x_1 b^1 + x_0 b^0$$
$$+ y_n b^n + y_{n-1} b^{n-1} + y_{n-2} b^{n-2} + \dots + y_1 b^1 + y_0 b^0$$
$$z_{n+1} b^{n+1} + z_n b^n + z_{n-1} b^{n-1} + z_{n-2} b^{n-2} + \dots + z_1 b^1 + z_0 b^0$$

where $S_i = x_i + y_i + C$, $C = S_{i-1}/b$, and $z_i = S_i \bmod b$ where $S_{-1} = 0$

- Addition in base 2:

```
  00101101
+ 10011001
  --------
  11000110
```

- the sum might have ***one*** more digit than the largest operand

Multiplication

- Multiplication in base 2: `00101101 * 10111001`

```
1 00101101
0  00000000
1   00101101
1    00101101
1     00101101
0      00000000
0       00000000
1        00101101
  ---------------
  010000010000101
```

- The product has about as many digits as the two operands combined, i.e.

$$\log(a \times b) = \log(a) + \log(b)$$

Machine Arithmetic

- Computers usually have a fixed number of binary digits ("bits"), e.g., 32 bits
- For example, using 6 bits, numbered 0 to 5 from the right

  largest number $111111_2 = 63_{10} = 2^6 - 1$

  smallest number $000000_2 = 0$
- What is 50 + 20?

```
  110010
+ 010100
  ------
 1000110
```

- The highest bit does not fit, so we get $000110_2 = 6_{10}$
- Spilling over the lefthand side is called ***overflow***

Sign Magnitude and One's Complement

- ***Sign-magnitude*** notation:

  bit $n-1$ is the sign; 0 for **+**, 1 for **-**

  bits $n-2$ through 0 hold an unsigned number

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $111111_2 = -31_{10} = -\left(2^{6-1} - 1\right)$
- Addition and subtraction are complicated when signs differ
- Sign-magnitude is rarely used
- ***One's-complement*** notation:

  bit $n-1$ is the sign; bits $n-2$ through 0 hold an unsigned number

  bits $n-2$ through 0 hold ***complement*** of negative numbers

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $100000_2 = -31_{10} = -\left(2^{6-1} - 1\right)$
- Addition and subtraction are easy, but there are ***2*** representations for 0

Two's Complement

- ***Two's-complement*** notation:

  bit $n-1$ is the sign; bits $n-2$ through 0 hold an unsigned number

  bits $n-2$ through 0 hold the ***complement*** of a negative number ***plus 1***

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $100000_2 = -32_{10} = -2^{6-1}$; note ***asymmetry***
- To negate a 2's compl. number: first complement all the bits, then add 1

	start with	complement	increment	
+6	000110	111001	111010	-6
-6	111010	000101	000110	+6
+0	000000	111111	000000	-0
+1	000001	111110	111111	-1
+31	011111	100000	100001	-31
-31	100001	011110	011111	+31
-32	100000	011111	100000	-32

Two's Complement, Cont'd

- Adding 2's-complement numbers: ignore signs, add unsigned bit strings

```
  +20     010100        -20     101100
+ - 7   + 111001      + + 7   + 000111
  ---     ------        ---     ------
  +13     001101        -13     110011

  +20     010100        -20     101100
+ + 7   + 000111      + - 7   + 111001
  ---     ------        ---     ------
  +27     011011        -27     100101
```

- Signed overflow occurs if

  the result does not fit, ***and***

  the carry ***into*** the sign bit differs from the carry ***out*** of the sign bit

```
  +20     010100        -20     101100
+ +17   + 010001      + -17   + 101111
  ---     ------        ---     ------
  -27     100101        +27     011011
```

- Same hardware for ***both*** unsigned and signed, but flags ***two*** conditions

  overflow signed overflow

  carry unsigned overflow

Sign Extension

- To convert from a small signed integer to a larger one, copy the sign bit

	+5	−5
4 bits	0101	1011
8 bits	00000101	11111011

- To convert a large signed integer to a smaller one: check trunced bits

	+5	−5	
8 bits	**00000101**	**11111011**	
4 bits	**0101**	**1011**	**OK!**

	+20	−20	
8 bits	00010100	11101100	
4 bits	0100	1100	Bad!

- Hardware does extension, but ***may not*** check for truncation; nor does C

```
short small = -50; long big = small;
printf("%d %d\n, small, big);          -50 -50

long big = 40000; short small = big;
printf("%d %d\n", small, big);         -25536 40000

char c = 255;
printf("%d\n", c);                     -1
```


Floating Point Numbers

- Floating point numbers are like scientific notation

1.386×10^{6}

-3.0083×10^{-14}

4.32×10^{-8}

general form is

$\pm m \times 10^{\pm p}$ **exponent** **significand**

- Significand restricted to range, e.g., $0 \le m < 1$, and fixed number of digits

- Floating point is approx. representation for infinitely many real numbers

$m \times \beta^{k}$ $\quad m$ is an n-bit ***significand*** or ***fraction***

β is the ***base*** (usually 2)

k is the ***exponent***

e.g. for base 2

$$0.100011 \times 2^{6} = \left(1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6}\right) \times 2^{6}$$

Floating Point Numbers, cont'd

- ***Normalized*** floating point numbers make the representation unique

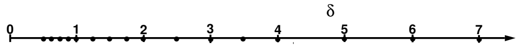
most significant digit is nonzero, e.g., $0.00486 \times 10^{1} \Rightarrow 0.486 \times 10^{-1}$

for floating point numbers, $\beta^{n-1} \le m < \beta^{n}$ or $1/\beta \le |m| < 1$

i.e., when $\beta = 2$, most significant bit of m is 1

- Example: $n = 3, \beta = 2, -1 \le k \le 2$

m \ k	-1	0	1	2
1.00	.5	1.	2.	4.
1.01	.625	1.25	2.5	5.
1.10	.75	1.5	3.	6.
1.11	.875	1.75	3.5	7.
δ	.125	.25	.5	1.



- What about 0.0? Use reserved values of k, e.g.,

$1.00_2 \times 2^{-2}$ for 0.0, $1.11_2 \times 2^{5}$ for ∞

IEEE Floating Point

- IEEE format uses a ***hidden bit*** to increase precision by 1 bit

all ***normalized*** floating point numbers have the form $1.f \times 2^{e}$,

so ***assume*** the leading 1 and omit it

- Single precision (`float`) format

31	30–23	22–0
s	e	f
sign	exponent	fraction

$-126 \le e \le 127$, $\textit{bias} = 127$, $0 \le f < 2^{23}$

- Values `1.1754943508222875e-38` to `3.40282346638528860000e+38`

$e - 127$	f	f. p. number
$-126 \le e \le 127$	$0 \le f < 2^{23}$	$\pm 1.f \times 2^{e}$
128	0	$\pm\infty$
128	$\ne 0$	NaN (signaling/quiet)
−127	0	± 0.0
−127	$\ne 0$	$\pm 0.f \times 2^{-126}$ **(denormalized)**

IEEE Floating Point, cont'd

- Double precision (`double`) format

63	62–52	51–0
s	e	f
sign	exponent	fraction

$-1022 \le e \le 1023$, ***bias*** $= 1023$, $0 \le f < 2^{52}$

- Values: `2.2250738585072014e-308` to `1.7976931348623157e+308`

$e - 1023$	f	**f. p. number**
$-1022 \le e \le 1023$	$0 \le f < 2^{52}$	$\pm 1.f \times 2^{e}$
1024	0	$\pm\infty$
1024	$\ne 0$	**NaN (signaling/quiet)**
-1023	0	± 0.0
-1023	$\ne 0$	$\pm 0.f \times 2^{-1022}$ **(denormalized)**

- Biased exponents in the most-significant bits are useful because

  integer compare instructions can be used to compare floating point values

  a bit string of 0's represents the value 0.0