Number Systems

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

  where are the ***positional coefficients***

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

Conversions

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

- Easier to convert to octal, then to binary

Addition

- Addition in base b

  where , , and where

- 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
-----------------
  0100000100001O1
```

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

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
  - smallest number
- What is 50 + 20?

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

- The highest bit doesn’t fit, so we get
- Spilling over the lefthand side is ***overflow***

Sign Magnitude and One’s Complement

- ***Sign-magnitude*** notation:
  - bit is the sign; 0 for +, 1 for -
  - bits through 0 hold an unsigned number
  - largest number
  - smallest number
- Addition and subtraction are complicated when signs differ
- Sign-magnitude is rarely used
- ***One’s-complement*** notation: $-k = (2^n-1) - k = 11111...(n\ bits) - k$
  - bit is the sign; bits through 0 hold an unsigned number
  - bits through 0 hold ***complement*** of negative numbers
  - largest number
  - smallest number
- Addition and subtraction are easy, but there are ***2*** representations for 0

Two’s Complement

- ***Two’s-complement*** notation: $-k = 2^n - k = (2^n-1) - k + 1$
  - bit is the sign; bits through 0 hold an unsigned number
  - bits through 0 hold the ***complement*** of a negative number ***plus 1***
  - largest number
  - smallest number ; 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 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

Floating Point Numbers

- Floating point numbers are like scientific notation

- Significand restricted to range, e.g., , and fixed number of digits

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

  m is an n-bit ***significand*** or ***fraction***

  is the ***base*** (usually 2)

  k is the ***exponent***

  e.g. for base 2

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
```


IEEE Floating Point

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

  all ***normalized*** floating point numbers have the form ,

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

- Single precision (`float`) format

, ,

- Values `1.1754943508222875e-38` to `3.4028234663852886e+38`

k =	f	f. p. number
		NaN (signaling/quiet)
		(denormalized)

Floating Point Numbers, cont'd

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

  most significant digit is nonzero, e.g.,

  for floating point numbers, or

  i.e., when , most significant bit of m is 1

- Example:

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.
	.125	.25	.5	1.

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

  for 0.0, for

IEEE Floating Point, cont'd

- Double precision (`double`) format

, ,

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

k =	f	f. p. number
		NaN (signaling/quiet)
		(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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh