CS 101:
Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur
(cs101@cse.iitb.ac.in)

Lecture 3: Number Representations

Representing Numbers

- Digital Circuits can store and manipulate 0’s and 1’s.
- How can we represent numbers using this capability?
- Key idea : Binary number system
- Represent all numbers using only 1's and 0's

Number Systems

- **Roman system**
  – new symbols for larger numbers
  – Cludgy and a real pain to add and do other arithmetic.

Roman Numeral Table

1	I	14	XIV	27	XXVII	150	CL
2	II	15	XV	28	XXVIII	200	CC
3	III	16	XVI	29	XXIX	300	CCC
4	IV	17	XVII	30	XXX	400	CD
5	V	18	XVIII	31	XXXI	500	D
6	VI	19	XIX	40	XL	600	DC
7	VII	20	XX	50	L	700	DCC
8	VIII	21	XXI	60	LX	800	DCCC
9	IX	22	XXII	70	LXX	900	CM
10	X	23	XXIII	80	LXXX	1000	M
11	XI	24	XXIV	90	XC	1600	MDC
12	XII	25	XXV	100	C	1700	MDCC
13	XIII	26	XXVI	101	CI	1900	MCM

MathATube.com

- Radix based number systems (e.g. Decimal)
- Revolutionary concept in number representation!

Radix-Based Number Systems

- Key idea: position of a symbol determines it's value!
  PLACE VALUE

- Number systems with radix ***r*** should have ***r*** symbols
  - The value of a symbol is multiplied by ***r*** for each left shift.
  - Multiply from right to left by: 1, r, r^2, r^3, ... and then add

Decimal Number System

- **RADIX is 10**. Symbols go from 0-9.
- Place-Values: 1, 10,100,1000...
- In the decimal system: 346

  Value of "6" = 6×10^0

  Value of "4" = 4×10^1

  Value of "3" = 3×10^2

  and, $\mathbf{346 = 6 \times 10^0 + 4 \times 10^1 + 3 \times 10^2}$

Octal Number Systems

- **RADIX is 8**. Symbol set: 0-7
- Place Value: 1, 8, 64, 512,....
- **23 in octal = ? In Decimal?**
  - Value of 3 = 3
  - Value of 2 = 2 x 8
  - ***Value of 23 in octal = 19 in decimal***
- **45171 in octal =**
  - 1 + 8*7+ 8*8*1+ 8*8*8*5+ 8*8*8*8*4

    = 19065 in decimal

Binary System

- **Radix= 2. Symbol set** : 0 and 1
- Place-value: powers of two:

128	64	32	16	8	4	2	1

- **11 in binary:**
  - Value of rightmost 1 = 1
  - Value of next 1 = 1 x2
  - 11 in binary = **3 in decimal**

- **110011**

128	64	32	16	8	4	2	1
		1	1	0	0	1	1

= 1x1 + 1 x 2 + 0 x 2*2+ 0 x 2*2*2 + 1 x 2^4 + 1 x 2^5

= 1 + 2 + 16 + 32= **51 (in decimal)**

Binary System: Representing Numbers

- Decimal to binary conversion
  - Express it as a sum of powers of two
- Example: the number 154 in binary:
  - $154 = 128 + 16 + 8 + 2$
  - $154 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	0	1	1	0	1	0

  - Thus 154 in binary is 10011010

Hexadecimal

- Long bit strings are unwieldy for humans to read and type in.
  - 00010010101010000111110000011000
- Decimal notations don't fit the "power of 2" concept well.
- Compromise is ***<u>Hex</u>***
  - Base 16 => much shorter strings
  - Conversion between binary and Hex is easy.

Hex Digits and Conversions

Hexadecimal Digits

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

Each Hex digit encodes 4 bits

0001 0010 1010 1000 0111 1100 0001 1000

= **0x12A87C18**

Fractions In Binary

- ***Powers on the right side of the point are negative:***

8	4	2	1	1/2	1/4	1/8	1/16

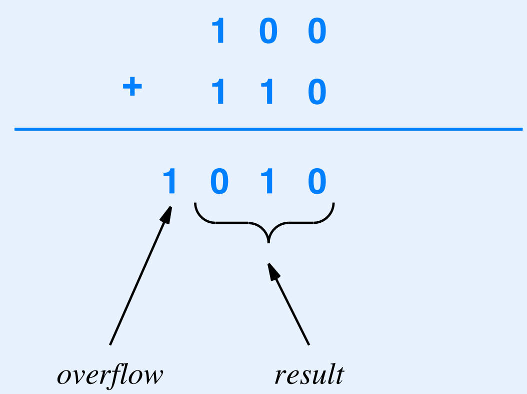
- Binary 0.1 = $0 + 1 \times 2^{-1}$ = 0.5 in decimal
- In Binary 0.11 = $0 \times 1 + 1 \times 2^{-1} + 1 \times 2^{-2}$

= 0.5 + 0.25 = 0.75 in decimal

Representing Non-Negative Numbers (unsigned int)

- The number of bits used cannot be chosen arbitrarily
- Choices allowed: 8, 16, 32, 64
- Example: To store 25 using 32 bits:
  - 25 Decimal = 00000000000000000000000000011001
  - So store the following charge pattern (H=High, L=Low)
  - LLLLLLLLLLLLLLLLLLLLLLLLLLLHHLLH
- Range stored: 0 to $2^{32} - 1$.

Illustration Of Overflow



- Hardware records overflow in separate carry indicator
  - Software must test after arithmetic operation if programmer wishes the program to proceed differently in the case of overflow
  - Can be used to raise an *exception* which starts the hardware on a pre-defined sequence of actions to respond; not steps from the executing program

Signed Values

- Signed arithmetic is needed by most programs
- Some bit patterns are used for negative values (typically half)
- Tradeoff: unsigned representation cannot store negative values, but can store integers that are twice as large as a signed representation
- Several Representations:
  - Sign magnitude
  - One's complement
  - Two's complement
- Each has interesting quirks and has been used in at least one computer.

Sign Magnitude Form

- Familiar to humans
- MS bit represents sign (0 for +ve and 1 for –ve)
- Successive bits represent absolute value of integer
- Interesting quirk: can create negative zero. For 8-bit representation of numbers, this quirk looks like this

  00000000
  10000000

  where both of these strings represent the value zero
- For 32 bits, range stored: $-(2^{31} - 1)$ to $2^{31} - 1$

Two's complement

- Positive number uses positional representation
- Negative number formed by inverting all bits of the value of the number and adding 1 to the result.
- Example of 4-bit two's complement

  0 0 1 0 represents 2

  1 1 1 0 represents –2
- High-order bit is set if number is negative
- Interesting quirk: for a given number of bits, one more negative value than positive value in the range.

Values In Four Bit Unsigned And Two's Complement Representations

Binary Value	Unsigned Interpretation	Two's Complement Interpretation
1 1 1 1	15	– 1
1 1 1 0	14	– 2
1 1 0 1	13	– 3
1 1 0 0	12	– 4
1 0 1 1	11	– 5
1 0 1 0	10	– 6
1 0 0 1	9	– 7
1 0 0 0	8	– 8
0 1 1 1	7	7
0 1 1 0	6	6
0 1 0 1	5	5
0 1 0 0	4	4
0 0 1 1	3	3
0 0 1 0	2	2
0 0 0 1	1	1
0 0 0 0	0	0

(Again, written in decimal notation)

Addition in Two's complement

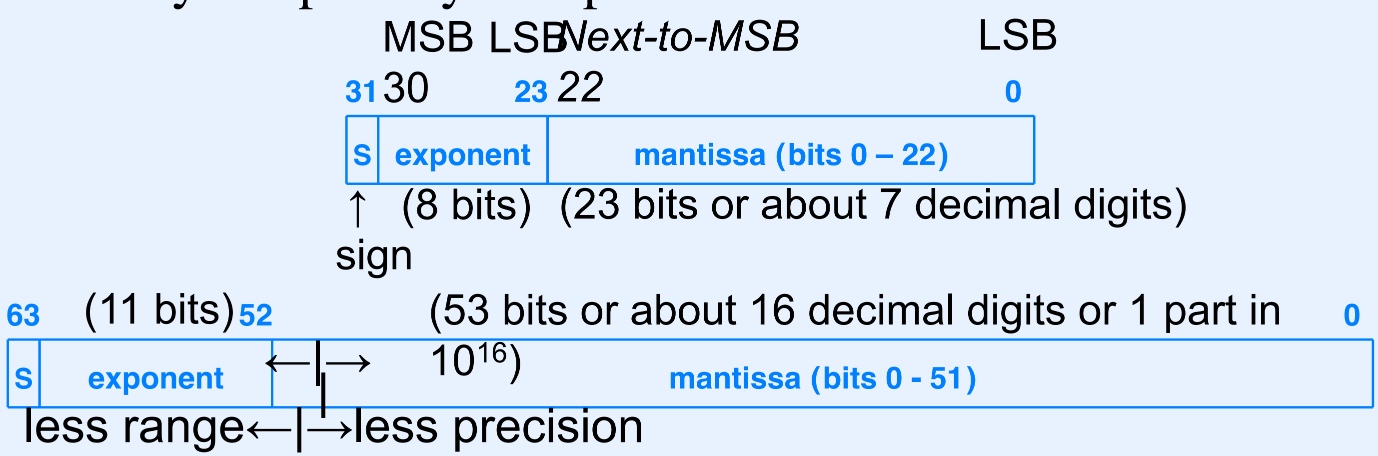
Both +ve and SUM < $2^{(n-1)}$	Both +ve nos and SUM >= $2^{(n-1)}$	One +ve and one –ve with neg > pos	One +ve and one –ve with pos > neg	Both –ve and |SUM| <= $2^{(n-1)}$	Both –ve and |SUM| > $2^{(n-1)}$
+3 0011 +4 0100 ------------- +7 0111 **Correct**	+5 0101 +6 0110 -------------- 1011 **Incorrect because of overflow**	+5 0101 -6 1010 -------------- -1 = 1111 **Correct**	+6 0110 -5 1011 -------------------- 1 = 1 0001 **Answer is correct if the overflow bit is ignored.**	-3 1101 -4 1100 ----------------- 1 1001 **Answer is correct (-7) if carry or overflow bit is ignored.**	-5 1011 -6 1010 -------------- 1 0101 **Overflow implies result in incorrect**

Floating Point

- So far, we have only seen signed integer representations.
- For FP, you have:
  - 1 sign bit (MSB)
  - Bit string for Mantissa – decides precision
  - Bit String for Exponent (how many bits to shift the decimal point) – decides range
- Some optimizations in IEEE FP standard:
  - Eliminate leading zeros from the Mantissa
    - $0.003 \times 10^4 = 3 \times 10^1$
  - Since normalization in binary always leaves the mantissa with a 1 in the MSB we can simply omit it and have it be assumed by the hardware.

Example Floating Point Representation: IEEE Standard 754

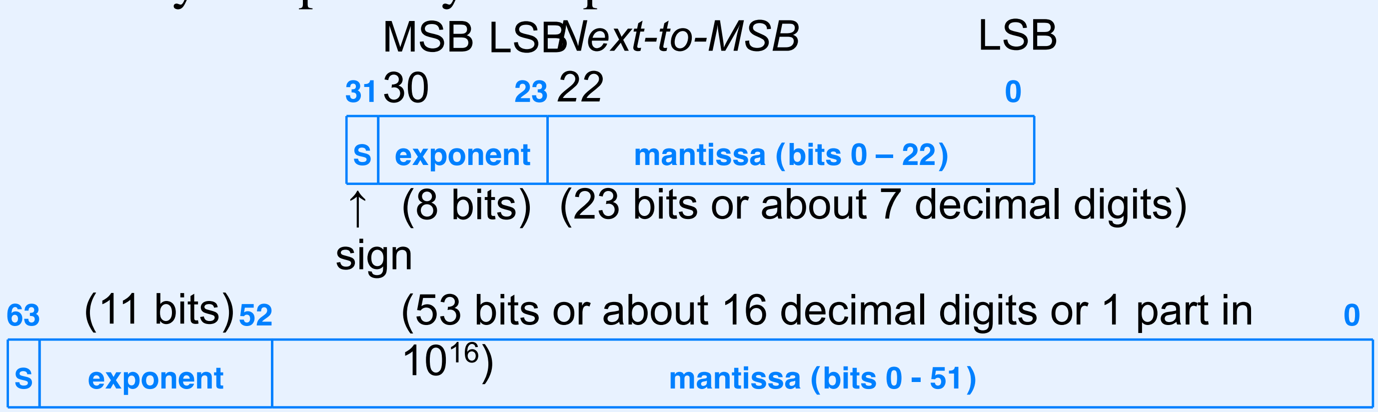
- Specifies single-precision and double-precision representations
- Widely adopted by computer architects



Radix, r, is implicit in the hardware circuit, not stored explicitly. *Precision* is determined by the number of digits in the mantissa; *Range* by number of digits in the exponent. Assume a fixed word length. Then moving the line between the exponent and mantissa fields left decreases range and increases precision. If moved right, then the opposite is true. This is the range/precision tradeoff for FP.

Example Floating Point Representation: IEEE Standard 754

- Specifies single-precision and double-precision representations
- Widely adopted by computer architects



STORAGE REPRESENTATION of Floating Point is depicted above.

Sign *is a 1-bit field identical in definition to sign-magnitude format.*

Exponent *is an integer, but not a representation we have seen already.*

Mantissa *is stored in memory without including its MSB!*

ALU hardware takes advantage of this deeply thought out

IEEE FP implementation details

- Format:
  Sign | Biased exponent | Mantissa *with hidden 1 MSB*

  or s|E|M

- Sign field definition same as ever: 0 = + 1 = –

IEEE FP implementation details

- Exponent uses a new representation: Biased
- Bias makes comparing two FP numbers much easier
  - Range of 8-bit exponent e is –127 to +127 in sign magnitude
  - Represent e as unsigned integer $E = e + Bias$ where Bias = 127
  - Thus, $0 \le E \le 255$
  - Consider FP numbers $x_1 = s_1E_1M_1$ and $x_2 = s_2E_2M_2$
  - Let • denote concatenation of bit fields and interpret the bit strings $s_1 \bullet E_1 \bullet M_1$ and $s_2 \bullet E_2 \bullet M_2$ as sign magnitude numbers
  - FP numbers $x_1 > x_2$ if and only if $s_1 \bullet E_1 \bullet M_1 > s_2 \bullet E_2 \bullet M_2$
  - Lets you compare FP numbers using *integer* hardware

IEEE FP implementation details

- Normalized mantissa has one significant digit to the left of the radix point
- In binary, this digit must be 1, no other choice
- Mantissa form is $1.m_{22}m_{21}\ldots m_1m_0$ so
  - *Neither FP registers nor memory store the 1*
  - ALU inserts this 1 bit into each incoming operand, uses it within ALU circuits, then strips it from each result on the way to storage
  - Gives one more bit of precision *for free*
  - Called the *hidden* bit or *implicit* bit

Range Of Values In IEEE Floating Point

- Single precision range is

$$2^{-126} \text{ to } 2^{127}$$

- Decimal equivalent is approximately

$$10^{-38} \text{ to } 10^{38}$$

- Double precision range is approximately
  (64-bit format)

$$10^{-308} \text{ to } 10^{308}$$

Example interpretation of FP

- Interpret the following bit string in IEEE FP, then write its value in decimal scientific notation

  1100 0001 1111 0000 0000 0000 0000 0000
- 1 10000011 11100000000000000000000 (sEM)
- Recalling hidden 1, IEEE FP fields literally say

  – 131 1.11100000000000000000000

  which is $-1.111 \times 2^{131-127}$ = -1.111×2^4
- To write in decimal scientific, normalize to obtain exponent=0 so radix change is trivial ($2^0 = Y^0$ for any Y a positive integer)
- $-1.111 \times 2^4 = -11110.0 \times 2^0 = -30.0 \times 10^0 =$ -3.0×10^1

Example decimal scientific to FP

- What bit string represents -9.175×10^1 in IEEE FP?
- Step 1: determine sign bit Sign = –
- Step 2: express as rational number: 91.75
- Step 3: convert decimal rational number to binary
  1011011.11
- Step 4: normalize binary rational number
  1.01101111×2^6
- Step 5: compute E=e+Bias=6+127=133= 10000101
- Step 6: strip mantissa of MSB 1. and assemble fields
  1 10000101 01101111000000000000000
  or 0xC2B78000

Steps in floating point + and –

- Align mantissas of operands
- Add or subtract mantissas as usual
- Post-normalize result mantissa
- Round result mantissa
- Check for overflow or underflow

Concluding Remarks

- Key idea 1: use numerical codes to represent non numerical entities
  - letters and other symbols: ASCII code
  - operations to perform on the computer: Operation codes
- Key idea 2: Current/charge/voltage values in the computer circuits represent bits (0 or 1).
- Key idea 3: Larger numbers can be represented using sequence of bits.
  - In a fixed number of bits you can represent numbers in a fixed range.
  - If you dedicate a bit to representing the sign, the range of representable numbers changes.
  - Real numbers are represented approximately. If you want more precision or greater range, you need to use larger number of bits.

Human perception

- We naturally live in a base 10 environment
- Computer exist in a base 2 environment
- So give the computer/digital system the task of doing the conversions for us.