

CPS 104
Computer Organization and Programming
Lecture-2 : Data representations,

Sep. 1, 1999

Dietolf Ramm

<http://www.cs.duke.edu/~dr/cps104.html>

Data Representation

- **Computers store variables (data)**
- **Typically Numbers and Characters or composition of these**
- **We reason about numbers many different ways**
- **The key is to use a representation that is “efficient”**

Number Systems

- A number is a mathematical concept
 - * 10
- Many ways to represent a number
 - * 10, ten, 2x5, X, 100/10, |||| |
- Symbols are used to create a representation
- Which representation is best for addition and subtraction?
- Which representation is best for multiplication and division?

More Number Systems

- **Humans use decimal (base 10)**
 - * **digits 0-9 are composed to make larger numbers**
$$11 = 1*10^1 + 1*10^0$$
 - * **weighted positional notation**
- **Addition and Subtraction are straightforward**
 - * **carry and borrow (today called regrouping)**
- **Multiplication and Division less so**
 - * **can use logarithms and then do adds and subtracts**

Changing Base (Radix)

- **Given 4 positions, what is largest number you can represent?**

Number Systems for Computers

- Today's computers are built from transistors
- Transistor is either **off** or **on**
- Need to represent numbers using only **off** and **on**
 - * two symbols
- **off** and **on** can represent the digits **0** and **1**
 - * **A bit can have a value of 0 or 1**
- **Binary representation**
 - * weighted positional notation using base 2

$$11_{10} = 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1011_2$$

$$11_{10} = 8 + 2 + 1$$

What is largest number, given 4 bits?

Conversion from Decimal to Binary

- **N** is a positive Integer (**in decimal representation**)
- b_i $i=0,\dots,k$ are the bits (binary digits) for the binary representation of **N**
- $N = b_k * 2^k + \dots + b_2 * 2^2 + b_1 * 2 + b_0$
- binary representation: $b_k \dots b_3 b_2 b_1 b_0$
- How do I compute b_0 ?

Compute binary representation of 11?

Conversion from Decimal

```
i=0;
while ( N > 0 )
{
    bi = N % 2; // bi = remainder; N mod 2
    N = N / 2; // N becomes quotient of division
    i++;
}
```

- Replace 2 by **A** and you have an algorithm that computes the **base A** representation for N

Powers of 2

<u>N</u>	<u>2ⁿ</u>	<u>Binary</u>
0	1	0000000001
1	2	0000000010
2	4	0000000100
3	8	0000001000
4	16	0000010000
5	32	0000100000
6	64	00001000000
7	128	00010000000
8	256	00100000000
9	512	01000000000
10	1024 (1K)	10000000000

Binary, Octal and Hexidecimal numbers

- Computers can input and output decimal numbers but they convert them to internal binary representation.
- Binary is good for computers, hard for us to read
 - * Use numbers easily computed from binary
- Binary numbers use only two different digits: {0,1}
 - * Example: $1200_{10} = 0000010010110000_2$
- Octal numbers use 8 digits: {0 - 7}
 - * Example: $1200_{10} = 04260_8$
- Hexidecimal numbers use 16 digits: {0-9, A-F}
 - * Example: $1200_{10} = 04B0_{16} = 0x04B0$
 - * does not distinguish between upper and lower case

Binary and Octal

- Easy to convert Binary numbers To/From Octal.
- Group the binary digits in groups of three bits and convert each group to an Octal digit.
- $2^3 = 8$

Bin.	Oct.
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Example:

11 000 010 011 001 110 100 111 101 010 101₂
3 0 2 3 1 6 4 7 5 2 5₈

Binary and Hex

- To convert to and from hex: group binary digits in groups of four and convert according to table
- $2^4 = 16$

Hex	Bin	Hex	Bin
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Example:

1100 0010 0110 0111 0100 1111 1101 0101₂
C 2 6 7 4 F D 5₁₆

Issues for Binary Representation

- **Complexity of arithmetic operations**
- **Negative numbers**
- **Maximum representable number**

Binary Integers

- **Unsigned Integers:**

- * $i = 100101_2; i = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

- 4 bits => max number is 15

- What about representing negative numbers?

Sign-Magnitude Representation for Integers

- Add a sign bit

 - * **Example:** $010110_2 = 22_{10}$; $110110_2 = -22_{10}$

- Advantages:

 - * Simple extension of unsigned numbers.
 - * Same number of positive and negative numbers.

- Disadvantages:

 - * Two representations for 0: $0=000000$; $-0=100000$.
 - * Algorithm (circuit) for addition depends on the arguments' signs.

2's Complement Representation for Integers

- Key idea is to use largest positive binary numbers to represent negative numbers
- Obtain negative number by subtracting large constant
- $i = -a_{n-1} * 2^{n-1} + a_{n-2} * 2^{n-2} + \dots + a_0 * 2^0$

6-bit examples:

$$010110_2 = 22_{10}; 101010_2 = -22_{10}$$

$$0_{10} = 000000_2; 1_{10} = 000001_2; -1_{10} = 111111_2$$

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Advantages:

- * Only one representation for 0: $0 = 000000$
- * Addition algorithm independent of sign bits.

- Disadvantage:

- * One more negative number than positive :

Example: 6-bit 2's complement number.

$100000_2 = -32_{10}$; but 32_{10} could not be represented

2's Complement Negation and Addition

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

Examples

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

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

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

2's Complement (cont.)

- Example: $A = 0x0ABC$; $B = 0x0FEB$.
- Compute: $A + B$ and $A - B$ in 16-bit 2's complement arithmetic.

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.

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
 - * speed of light $\approx 3 \times 10^8$
 - * $\pi = 3.1415\dots$
- Fixed number of bits limits range of integers
 - * Can't represent some important numbers
- Humans use Scientific Notation
 - * 1.3×10^4

Floating Point Representation

Numbers are represented by:

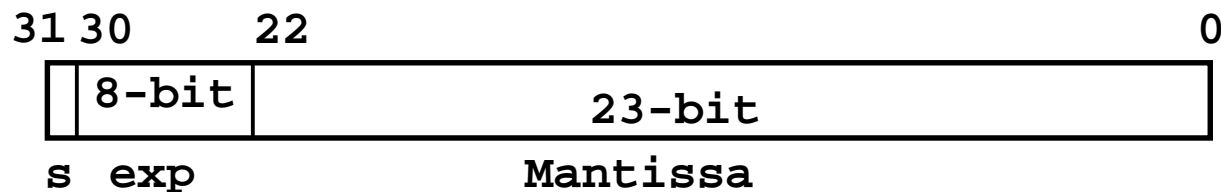
$$X = (-1)^s \times 2^{E-127} \times 1.M$$

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



Floating Point Representation

- The mantissa represents a fraction using binary notation:

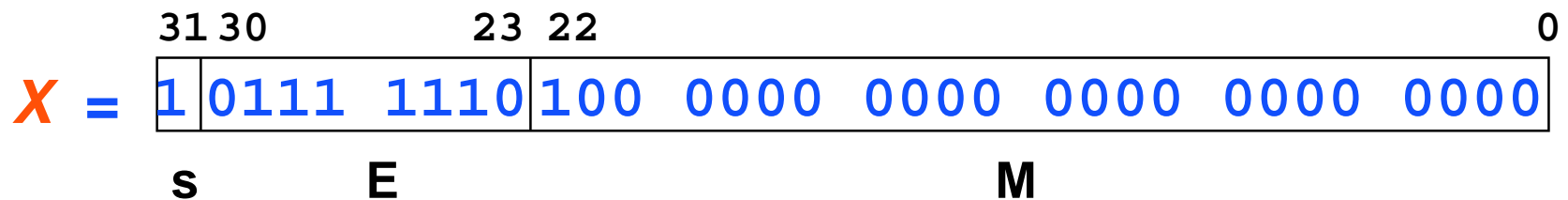
$$M = . s_1, s_2, s_3 \dots = 1.0 + s_1 * 2^{-1} + s_2 * 2^{-2} + s_3 * 2^{-3} + \dots$$

- Example: $X = -0.75_{10}$ in single precision $(-(1/2 + 1/4))$

$$-0.75_{10} = -0.11_2 = (-1) \times 1.1_2 \times 2^{-1} = (-1) \times 1.1_2 \times 2^{126-127}$$

$$S = 1 ; \text{Exp} = 126_{10} = 0111\ 1110_2 ;$$

$$M = 100\ 0000\ 0000\ 0000\ 0000\ 0000_2$$



Floating Point Representation

Example:

What floating-point number is:

0xC1580000?

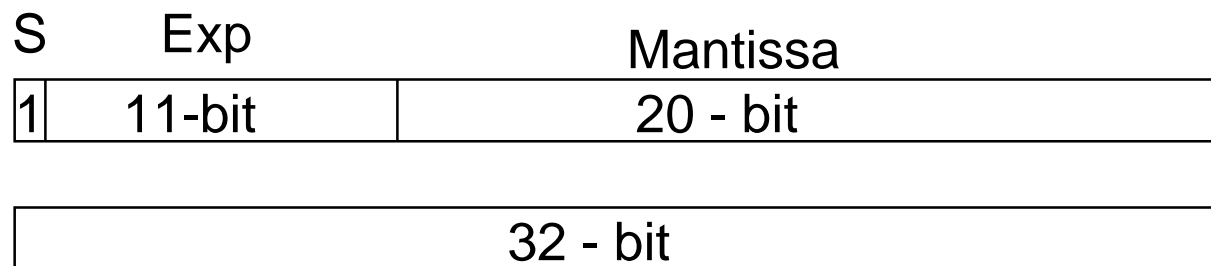
Floating Point Representation

- 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



ASCII Character Representation

Oct. Chr.

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	\$	045	%	046	&	047	'
050	(051)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[134	\	135]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174		175	}	176	~	177	del

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

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

Decimal: (BCD code)

digits 0-9 encoded as 0000 thru 1001

two decimal digits packed per 8 bit byte

Integers:

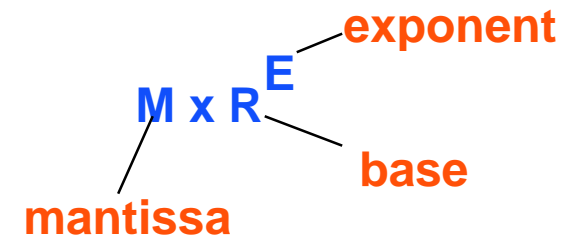
2's Complement (32-bit representation).

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**
- **Signed binary numbers**
 - * **2's complement**
 - * **arithmetic, negation**
- **Floating point representation**
 - * **hidden 1**
 - * **biased exponent**
 - * **single precision, double precision**
 - * **sign magnitude**

Next Time

Memory

- Pointers
- Arrays
- Strings

Bitwise operations

Reading

- Start Chapter 3