CS11001/CS11002

Programming and Data Structures (PDS)

(Theory: 3-1-0)

Numbers in Computers

Think of a number between 1 and 15

8	9	10	11
12	13	14	15

4	5	6	7
12	13	14	15

2	~~3~~	~~6~~	~~7~~
10	11	14	15

~~1~~	~~3~~	~~5~~	~~7~~
9	11	13	15

Binary Numbers

Yes = 1

No = 0

- Number 7 appears on the four cards in the pattern 'No, Yes, Yes, Yes'
- The number 7 in binary code is <u>0111</u>
- This is the **Computers Language!**

Why binary?

- Information is stored in computer via voltage levels.
- Using decimal would require 10 distinct and reliable levels for each digit.
- This is not feasible with reasonable reliability and financial constraints.
- Everything in computer is stored using binary: numbers, text, programs, pictures, sounds, videos, ...

Bit, Byte, and Word

| 0 | A bit is a size that can store 1 digit of a binary number, 0 or 1.

| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | A byte is 8 bits, which can store eight 0's or 1's.

cont'd

| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | →

→ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | A word is either 32 or 64 bits, depending on computers. Regular PC's are 32-bit word in size, higher-end workstations are 64-bit. Word size is the size of the registers.

What do these bits mean is a *matter of interpretation*! All information in a computer are represented in a uniform format of bit patterns.

Binary Nonnegative Integers

Given a 32-bit pattern, e.g.,

0000 0000 ... 0000 1101 1100,

a_{31} highest bit a_0 lowest bit

it can represent the integer (if you interpret it that way)

$$n = a_0 + a_1 \times 2 + a_2 \times 2^2 + a_3 \times 2^3 + \cdots a_k 2^k + \cdots, \qquad a_k = 0,1$$

Note that if we use 32-bit word, the smallest number is 0, and the largest number is 11111111 11111111 11111111 11111111, which is $2^{32}-1=4294967295$. Numbers bigger than this cannot be represented. If such things happen in a calculation, we say it overflowed. Such interpretation is called *unsigned int* in the programming language C.

In General (binary)

No. of bits	Binary	
	Min	Max
n	0	$2^n - 1$

Negative Numbers

- Popular schemes:
  - Signed Magnitude
  - One’s Complement
  - Two’s Complement

Sign-Magnitude

- Extra bit on left to represent sign
  - 0 = positive value
  - 1 = negative value
- E.g., 6-bit sign-magnitude representation of +5 and –5:

+5: 0 0 0 1 0 1   -5: 1 0 0 1 0 1

+ve   5   -ve   5

Ranges (revisited)

No. of bits	Binary			
	Unsigned		Sign-magnitude	
	Min	Max	Min	Max
1	0	1		
2	0	3	-1	1
3	0	7	-3	3
4	0	15	-7	7
5	0	31	-15	15
6	0	63	-31	31
Etc.				

In General (revisited)

No. of bits	Binary			
	Unsigned		Sign-magnitude	
	Min	Max	Min	Max
n	0	$2^n - 1$	$-(2^{n-1} - 1)$	$2^{n-1} - 1$

Difficulties with Sign-Magnitude

- Two representations of zero
  - Using 6-bit sign-magnitude…
    - `0: 000000`
    - `0: 100000`
- Arithmetic is awkward!

pp. 95-96

Complementary Representations

- 9's complement
- 10's complement
- 1's complement
- 2's complement

Exercises – Complementary Notations

- What is the 3-digit 10’s complement of 247?
  - Answer:_______
- What is the 3-digit 10’s complement of 17?
  - Answer:_______
- 777 is a 10’s complement representation of what decimal value?
  - Answer:_______

Skip answer Answer

Exercises – Complementary Notations

Answer

- What is the 3-digit 10’s complement of 247?
  - Answer:__753___
- What is the 3-digit 10’s complement of 17?
  - Answer:__983___
- 777 is a 10’s complement representation of what decimal value?
  - Answer:__223___

Ones' Complement

- Bitwise Not (simple)
- Used in UNIVAC
- Two representation for 0

Ones' Complement

- **binary decimal**

  11111110 -1
  + 00000010 +2
  ....
  1 00000000 0 <-- not the correct answer
  1 +1 <-- add carry
  ....
  00000001 1 <-- correct answer

Two's Complement

- Most common scheme of representing negative numbers in computers
- Affords natural arithmetic (no special rules!)
- To represent a negative number in 2's complement notation…
  1. Decide upon the number of bits (*n*)
  2. Find the binary representation of the +ve value in *n*-bits
  3. Flip all the bits (change 1's to 0's and vice versa)
  4. Add 1

Two's Complement Example

- Represent -5 in binary using 2's complement notation
  1. Decide on the number of bits

     6 (for example)
  2. Find the binary representation of the +ve value in 6 bits

     000101 (+5)
  3. Flip all the bits

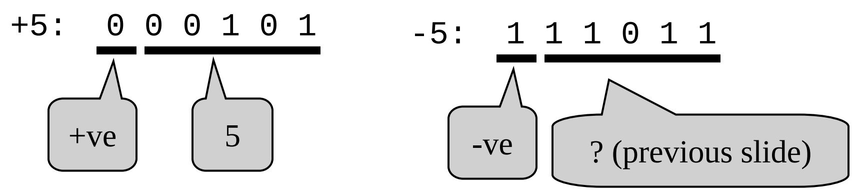
     111010
  4. Add 1

```
  111010
+      1
  ------
  111011
```

-5

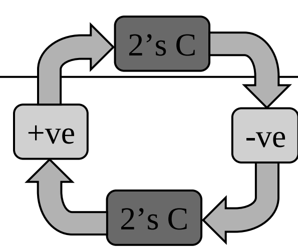
Sign Bit

- In 2’s complement notation, the MSB is the sign bit (as with sign-magnitude notation)
  - 0 = positive value
  - 1 = negative value

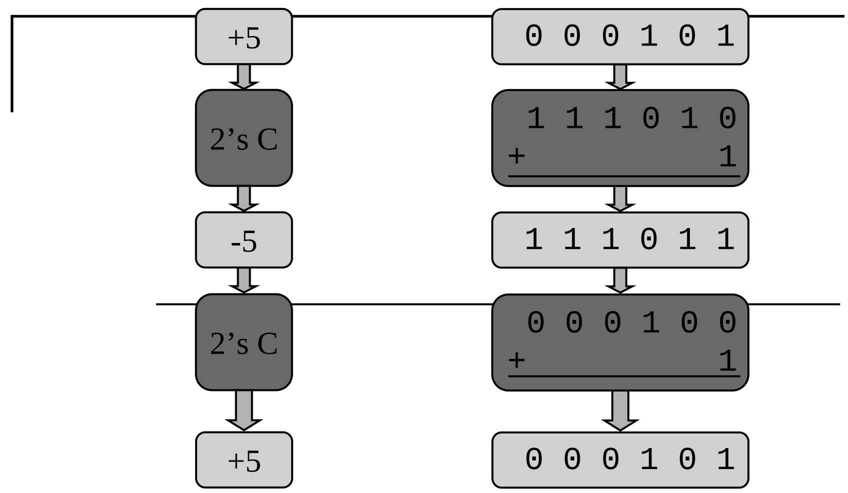


“Complementary” Notation

- Conversions between positive and negative numbers are easy
- For binary (base 2)…



Example



In General (revisited)

No. of bits	Binary					
	Unsigned		Sign-magnitude		2's complement	
	Min	Max	Min	Max	Min	Max
n	0	$2^n - 1$	$-(2^{n-1} - 1)$	$2^{n-1}-1$	-2^{n-1}	$2^{n-1} - 1$

Negative Integers

- To represent negative numbers, we'll agree that the highest bit a_{31} representing sign, rather than magnitude, 0 for positive, 1 for negative numbers.
- More precisely, all modern computers use 2's complement representations.
- The rule to get a negative number representation is: first write out the bit pattern of the corresponding positive number, complement all bits, then add 1.

Property of 2's Complement Numbers

- Two's complement *m* of a number *n* is such that adding it together you get zero ($m + n = 0$, modulo word size)
- Thus *m* is interpreted as negative of *n*.
- The key point is that computer has a finite word length, the last carry is thrown away.

2's Complement, example

- 1 in bits is 00000000 00000000 00000000 00000001

  complementing all bits, we get

  11111111 11111111 11111111 11111110

  Add 1, we get representation for –1, as

  11111111 11111111 11111111 11111111

	Decimal
00000000 00000000 00000000 00000001	1
+ 11111111 11111111 11111111 11111111	–1
1 00000000 00000000 00000000 00000000	0

Overflow bit not kept by computer

2's Complement, another e.g.

- 125 in bits is 00000000 00000000 00000000 01111101

  complement all bits, we get

  11111111 11111111 11111111 10000010

  Add 1, we get representation for –125, as

  11111111 11111111 11111111 10000011

	Decimal
00000000 00000000 00000000 01111111	127
+ 11111111 11111111 11111111 10000011	–125
1 00000000 00000000 00000000 00000010	+2

Overflow bit not kept by computer

What is -5 plus +5?

- Zero, of course, but let's see

Sign-magnitude	Twos-complement

```
Sign-magnitude                 Twos-complement

                                      11111111
-5:    10000101              -5:    11111011
+5:   +00000101              +5:   +00000101
       10001010                     00000000
```

Signed and Unsigned Int

- If we interpret the number as unsigned, an unsigned integer takes the range 0 to 2^{32}-1 (that is 000…000 to 1111….1111)
- If we interpret the number as signed value, the range is -2^{31} to 2^{31}-1 (that is 1000….000 to 1111…111, to 0, to 0111….1111).
- Who decide what interpretation to take? What if we need numbers bigger than 2^{32}?

Addition, Multiplication, and Division in Binary

```
   0 0 0 1 0 0 1 1     19+37=56
 + 0 0 1 0 0 1 0 1
 -----------------
   0 0 1 1 1 0 0 0
```

```
      0 1 1 1      7×5=35
    × 0 1 0 1
    ---------
      0 1 1 1
  + 0 1 1 1
  -----------
  1 0 0 0 1 1
```

```
              1 0 0 1
        _______________
1 0 0 0 | 1 0 0 1 0 1 0
        – 1 0 0 0
        ---------------
                1 0 1 0
              – 1 0 0 0
              ---------
                    1 0
```

74 ÷ 8 = 9 remainder 2

Overflow - Explanation

- 2147483645 + 2147483645 = -6
- Why?
  - 2^{31} -1 = 2147483647 and has 32 bit binary representation 0111…111. This is largest 2's complement 32 bit number.
  - 2147483645 would have representation
    011111…101.
  - When we add this to itself, we get
    X = 1111…1010 (overflow)
  - So, -X would be 000…0101 + 1 = 00…0110 = 6
  - So, X must be -6.

Overflows in signed magnitude system

- In signed-magnitude representation, a carry out means an overflows:

```
Ex:        0     1 0 1 1    (11)
           0     0 1 1 0     (6)
        -----------------------------
           0   1 0 0 0 1     (1)
```

The carry-out implies an overflow.

Overflows in complement system

- In both complement systems, however, a carry out DOES NOT mean an overflows:

Ex: 13 – 8 = 5

```
      0 1 1 0 1 (13)
      1 1 0 0 0  (-8)
  -----------------------------
    1 0 0 1 0 1  (5)
```

There is a carry-out, but there is no overflow.

Rule for overflow

- If X and Y (the two operands) are of different signs there is no overflow, regardless of a carry out.
- If X and Y are of the same sign and the sign of the result is different from the signs of the two operands, then an overflow occurs.

Examples

- 1 1 0 0 1 (-7)
  1 0 1 1 0 (-10)

----

1 0 1 1 1 1 (15) [Carry-out and overflow]

- 0 0 1 1 1 (7)
  0 1 0 1 0 (10)

----

1 0 0 0 1 (-15) [No-carry out but overflow]

Floating Point Numbers (reals)

- To represent numbers like 0.5, 3.1415926, etc, we need to do something else. First, we need to represent them in binary, as

$$n = \cdots + a_m 2^m \cdots + a_2 2^2 + a_1 2 + a_0 + a_{-1} \times \frac{1}{2} + a_{-2} \times 2^{-2} + a_{-3} \times 2^{-3} + \cdots a_{-k} 2^{-k} + \cdot$$

E.g. ~~11.00110 for 2+1+1/8+1/16=3.1875~~

- Next, we need to rewrite in scientific notation, as 1.100110 $\times 2^1$. That is, the number will be written in the form:

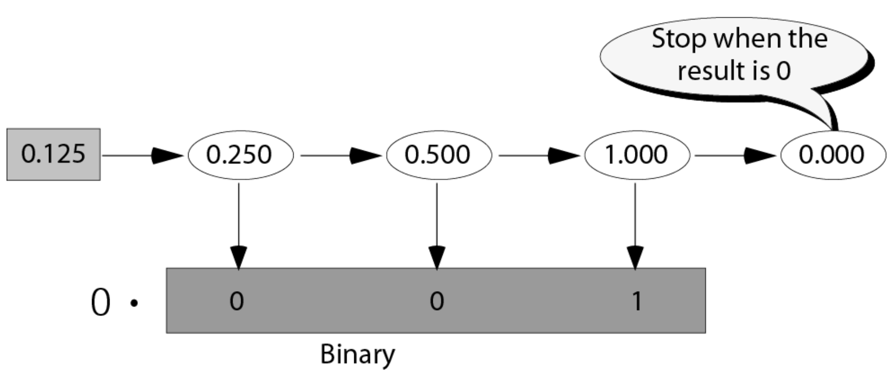
1.xxxxxx… $\times$ 2^e

x = 0 or 1

Figure 3-7

Changing fractions to binary

- Multiply the fraction by 2,...



Example 17

Transform the fraction 0.875 to binary

Solution

Write the fraction at the left corner. Multiply the number continuously by 2 and extract the integer part as the binary digit. Stop when the number is 0.0.

0.875 ➔	1.750 ➔	1.5 ➔	1.0 ➔	0.0
0 .	1	1	1	

Example 18

Transform the fraction 0.4 to a binary of 6 bits.

Solution

Write the fraction at the left cornet. Multiply the number continuously by 2 and extract the integer part as the binary digit. You can never get the exact binary representation. Stop when you have 6 bits.

0.4 ➔	0.8 ➔	1.6 ➔	1.2 ➔	0.4 ➔	0.8 ➔	1.6
0 .	0	1	1	0	0	1

Normalization

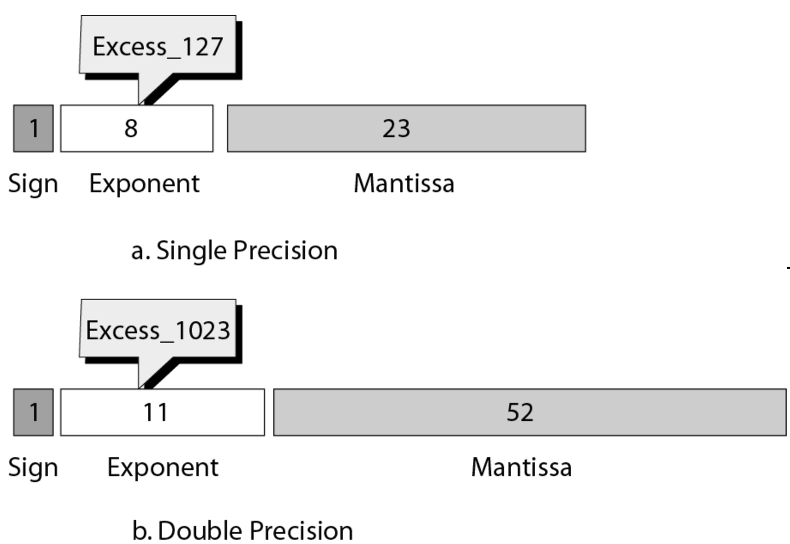
Example of normalization

Original Number	*Move*	*Normalized*
$+1010001.1101$	$\leftarrow$ 6	$+2^{6}$ x 1.01000111001
-111.000011	$\leftarrow$ 2	-2^{2} x 1.11000011
$+0.00000111001$	6 $\rightarrow$	$+2^{-6}$ x 1.11001
-0.001110011	3 $\rightarrow$	-2^{-3} x 1.110011

- Sign, exponent, and mantissa

Figure 3-8

IEEE standards for floating-point representation



Example 19

Show the representation of the normalized number $+ 2^6 \times 1.01000111001$

Solution

The sign is positive. The Excess_127 representation of the exponent is 133. You add extra 0s on the right to make it 23 bits. The number in memory is stored as:

0 10000101 01000111001000000000000

Number	Sign	Exponent	Mantissa
$-2^2 \times 1.11000011$	1	10000001	11000011000000000000000
$+2^{-6} \times 1.11001$	0	01111001	11001000000000000000000
$-2^{-3} \times 1.110011$	1	01111100	11001100000000000000000

Example of floating-point representation

Example 20

Interpret the following 32-bit floating-point number

1 01111100 11001100000000000000000

Solution

The sign is negative. The exponent is –3 (124 – 127). The number after normalization is

$$-2^{-3} \times 1.110011$$

Limitations in 32-bit Integer and Floating Point Numbers

- Limited range of values (e.g. integers only from -2^{31} to $2^{31}-1$)
- Limited resolution for real numbers. E.g., if x is a machine representable value, the next value is $x + \varepsilon$ (for some small ε). There is no value in between. This causes "floating point errors" in calculation. The accuracy of a single precision floating point number is about 6 decimal places.

Limitations of Single Precision Numbers

- Given the representation of the single precision floating point number format, what is the largest magnitude possible? What is the smallest number possible?
- With floating point number, it can happen that $1 + \varepsilon = 1$. What is that largest ε?

Floating Point Rounding Error

- Consider 4-bit mantissa floating point addition:
- $1.010 \times 2^{2} + 1.101 \times 2^{-1}$

$$
\begin{array}{rll}
 & 1.010000 \times 2^{2} & \\
+ & 0.001101 \times 2^{2} & \text{Shift exponent to that of the large number} \\
\hline
 & 1.011101 \times 2^{2} & \\
 & 1.100 \quad \times 2^{2} & \text{Round to 4 bits}
\end{array}
$$

In decimal, it means $5.0 + 0.8125 \approx 6.0$

Representation of Characters, the ASCII code

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

How to read the table: the top line specifies the last digit in hexadecimal, the leftmost column specifies the higher value digit. E.g., at location 41_{16} (=$0100\ 0001_2$=65_{10}) is the letter ‘A’.

Base-16 Number, or Hexadecimal

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

- Instead of writing out strings of 0’s and 1’s, it is easier to read if we group them in group of 4 bits. A four bit numbers can vary from 0 to 15, we denote them by 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

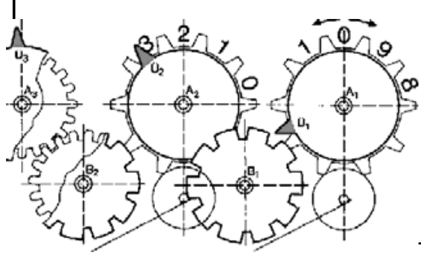
Program as Numbers

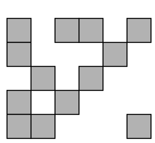
- High level programming language
  C = A + B;
- Assembly language
  add $5, $10, $3
- Machine code (MIPS computer)

bit 31 … bit 0

	$10	$3	$5		add
0 0 0 0 0 0	0 1 0 1 0	0 0 0 1 1	0 0 1 0 1	0 0 0 0 0	1 0 0 0 0 0
Not used	source 1	source 2	destination	not used	operation

Graphics as Numbers





A picture like this is also represented on computer by bits. If you magnify the picture greatly, you'll see square "pixels" which is either black or white. These can be represented as binary 1's and 0's.

Color can also be presented with numbers, if we allow more bits per pixel.

Music as Numbers – MP3 format

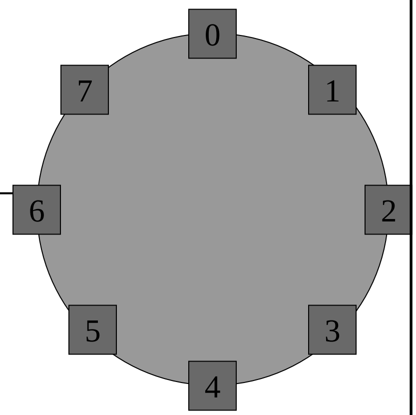
- CD music is sampled 44,100 times per second (44.1 kHertz), each sample is 2 bytes long
- The digital music signals are compressed, at a rate of 10 to 1 or more, into MP3 format
- The compact MP3 file can be played back on computer or MP3 players

Some other points

- Computer Science starts counting from 0 (why?)
- We have to perform operations in a finite space, unlike what we have done when we counted with real numbers, which were infinite…
  - imagine a world in which we are born, grow older by one year, become 1, 2, 3,…,62, 63 then again 0. Say, we decide we will not grow older beyond 64… strange…but computer does similar things!
  - A computer counting our age will count like 0, 1, 2, 3, …, 62, 63, 0,…! This is called modular arithmetic and gives lot of interesting results. Can you tell me from the above count values, some information of our computer…?

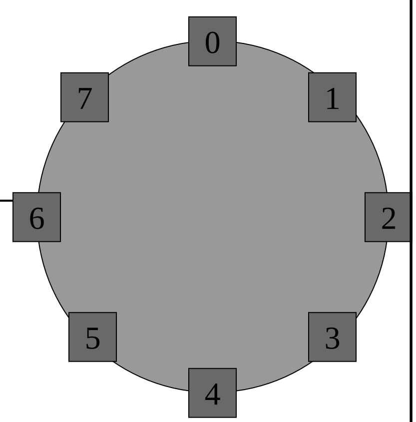
Modulo Arithmetic

- Consider the set of number {0, … ,7}
- Suppose all arithmetic operations were finished by taking the result modulus 8
- 3 + 6 = 9, 9 mod 8 = 1
  - 3 + 6 = 1
- 3*5 = 15, 15 mod 8 = 7
  - 3*5 = 7



Modulo Arithmetic: Computing in a finite world

- What is the additive inverse of 7 in modulo arithmetics?
- 7 + x = 0
- 7 + 1 = 0
- 0 and 4 are their own additive inverses
- Does each number also have a multiplicative inverse?
  - 7 x 7 = 1
  - Does each number has a multiplicative inverse?
  - What if m=11? Now does each number have a multiplicative inverse?



Summary

- All information in computer are represented by bits. These bits encode information. It's meaning has to be interpreted in a specific way.
- We've learnt how to represent unsigned integer, negative integer, floating pointer number, as well as ASCII characters.
- Computers have to compute in a finite world.