Number Systems

Md. Rokanujjaman
Associate Professor
Dept. of Computer Science and Engineering
University of Rajshahi
Rokon_cstru@yahoo.com, 01712043386

Number Systems & Codes

Part I: Number Systems

- Information Representations
- Positional Notations
- Decimal (base 10) Number System
- Other Number Systems & Base-R to Decimal Conversion
- Decimal-to-Binary Conversion
  - Sum-of-Weights Method
  - Repeated Division-by-2 Method (for whole numbers)
  - Repeated Multiplication-by-2 Method (for fractions)

Lecture 2: Number Systems & Codes

- Conversion between Decimal and other Bases
- Conversion between Bases
- Binary-Octal/Hexadecimal Conversion
- Binary Arithmetic Operations
- Negative Numbers Representation
  - Sign-and-magnitude
  - 1s Complement
  - 2s Complement
- Comparison of Sign-and-Magnitude and Complements

Lecture 2: Number Systems & Codes

- Complements
  - Diminished-Radix Complements
  - Radix Complements
- 2s Complement Addition and Subtraction
- 1s Complement Addition and Subtraction
- Overflow
- Fixed-Point Numbers
- Floating-Point Numbers
- Excess Representation
- Arithmetics with Floating-Point Numbers

Information Representation (1/4)

- Numbers are important to computers
  - represent information precisely
  - can be processed
- For example:
  - to represent *yes* or *no*: use 0 for *no* and 1 for *yes*
  - to represent 4 seasons: 0 (autumn), 1 (winter), 2(spring) and 3 (summer)
  - matriculation number (8 alphanumeric) to represent individual students

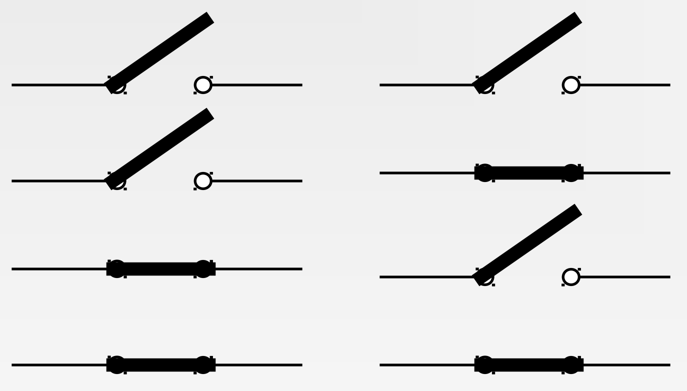
Information Representation (2/4)

- Elementary storage units inside computer are *electronic switches*. Each switch holds one of two states: *on* (1) or *off* (0).

- We use a *bit* (*bi*nary digi*t*), 0 or 1, to represent the state.

Information Representation (3/4)

- Storage units can be grouped together to cater for larger range of numbers. Example: 2 switches to represent 4 values.



0 (00)

1 (01)

2 (10)

3 (11)

Information Representation (4/4)

- In general, N bits can represent 2^N different values.
- For M values, $\lceil \log_2 M \rceil$ bits are needed.

1 bit $\rightarrow$ represents up to 2 values (0 or 1)

2 bits $\rightarrow$ rep. up to 4 values (00, 01, 10 or 11)

3 bits $\rightarrow$ rep. up to 8 values (000, 001, 010. …, 110, 111)

4 bits $\rightarrow$ rep. up to 16 values (0000, 0001, 0010, …, 1111)

32 values $\rightarrow$ requires 5 bits

64 values $\rightarrow$ requires 6 bits

1024 values $\rightarrow$ requires 10 bits

40 values $\rightarrow$ requires 6 bits

100 values $\rightarrow$ requires 7 bits

Positional Notations (1/3)

- Position-independent notation
  - each symbol denotes a value independent of its position: Egyptian number system
- Relative-position notation
  - Roman numerals symbols with different values: I (1), V (5), X (10), C (50), M (100)
  - Examples: I, II, III, IV, VI, VI, VII, VIII, IX
  - Relative position important: IV = 4 but VI = 6
- Computations are difficult with the above two notations

Positional Notations (2/3)

- Weighted-positional notation
  - Decimal number system, symbols = { 0, 1, 2, 3, …, 9 }
  - Position is important
  - Example: $(7594)_{10} = (7\text{x}10^3) + (5\text{x}10^2) + (9\text{x}10^1) + (4\text{x}10^0)$
  - The value of each symbol is dependent on its type and its position in the number
  - In general,

    $(a_n a_{n-1} \ldots a_0)_{10} = (a_n \text{ x } 10^n) + (a_{n-1} \text{ x } 10^{n-1}) + \ldots + (a_0 \text{ x } 10^0)$

Positional Notations (3/3)

- Fractions are written in decimal numbers after the *decimal point*.
  - $2\frac{3}{4} = (2.75)_{10} = (2 \times 10^{0}) + (7 \times 10^{-1}) + (5 \times 10^{-2})$
  - In general,

    $(a_n a_{n-1} \ldots a_0 \,.\, f_1 f_2 \ldots f_m)_{10} =$

    $(a_n \times 10^{n}) + (a_{n-1} \times 10^{n-1}) + \ldots + (a_0 \times 10^{0}) +$

    $(f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \ldots + (f_m \times 10^{-m})$
- The radix (or base) of the number system is the total number of digits allowed in the system.

Decimal (base 10) Number System

- Weighting factors (or weights) are in powers-of-10:

  $\ldots\ 10^{3}\ 10^{2}\ 10^{1}\ 10^{0}.10^{-1}\ 10^{-2}\ 10^{-3}\ 10^{-4}\ \ldots$

- To evaluate the decimal number 593.68, the digit in each position is multiplied by the corresponding weight:

  $5\times10^{2} + 9\times10^{1} + 3\times10^{0} + 6\times10^{-1} + 8\times10^{-2}$

  $= (593.68)_{10}$

Other Number Systems & Base-R to Decimal Conversion (1/3)

- Binary (base 2): weights in powers-of-2.
  - Binary digits (bits): ***0,1.***
- Octal (base 8): weights in powers-of-8.
  - Octal digits: ***0,1,2,3,4,5,6,7.***
- Hexadecimal (base 16): weights in powers-of-16.
  - Hexadecimal digits: ***0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.***
- Base *R*: weights in powers-of-R.

Other Number Systems & Base-R to Decimal Conversion (2/3)

- $(1101.101)_2 = 1\times2^3 + 1\times2^2 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-3}$
  $= 8 + 4 + 1 + 0.5 + 0.125 = (13.625)_{10}$
- $(572.6)_8 = 5\times8^2 + 7\times8^1 + 2\times8^0 + 6\times8^{-1}$
  $= 320 + 56 + 2 + 0.75 = (378.75)_{10}$
- $(2A.8)_{16} = 2\times16^1 + 10\times16^0 + 8\times16^{-1}$
  $= 32 + 10 + 0.5 = (42.5)_{10}$
- $(341.24)_5 = 3\times5^2 + 4\times5^1 + 1\times5^0 + 2\times5^{-1} + 4\times5^{-2}$
  $= 75 + 20 + 1 + 0.4 + 0.16 = (96.56)_{10}$

Other Number Systems & Base-R to Decimal Conversion (3/3)

- Counting in Binary
- Assuming non-negative values, *n bits* $\rightarrow$ largest value $2^n - 1$.

  Examples: 4 bits $\rightarrow$ 0 to 15;

  6 bits $\rightarrow$ 0 to 63.
- range of *m* values $\rightarrow$ $\log_2 m$ bits

Decimal Number	Binary Number
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
10	1 0 1 0
11	1 0 1 1
12	1 1 0 0
13	1 1 0 1
14	1 1 1 0
15	1 1 1 1

Decimal-to-Binary Conversion

- Method 1: *Sum-of-Weights Method*
- Method 2:
  - *Repeated Division-by-2 Method* (for whole numbers)
  - *Repeated Multiplication-by-2 Method* (for fractions)

Sum-of-Weights Method

- Determine the set of binary weights whose sum is equal to the decimal number.

$(9)_{10} = 8 + 1 = 2^3 + 2^0 = (1001)_2$

$(18)_{10} = 16 + 2 = 2^4 + 2^1 = (10010)_2$

$(58)_{10} = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1 = (111010)_2$

$(0.625)_{10} = 0.5 + 0.125 = 2^{-1} + 2^{-3} = (0.101)_2$

Repeated Division-by-2 Method

- To convert a whole number to binary, use successive division by 2 until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$(43)_{10} = (101011)_2$

2	43		
2	21	rem 1	← LSB
2	10	rem 1	
2	5	rem 0	
2	2	rem 1	
2	1	rem 0	
	0	rem 1	← MSB

Repeated Multiplication-by-2 Method

- To convert decimal fractions to binary, repeated multiplication by 2 is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$(0.3125)_{10} = (.0101)_2$

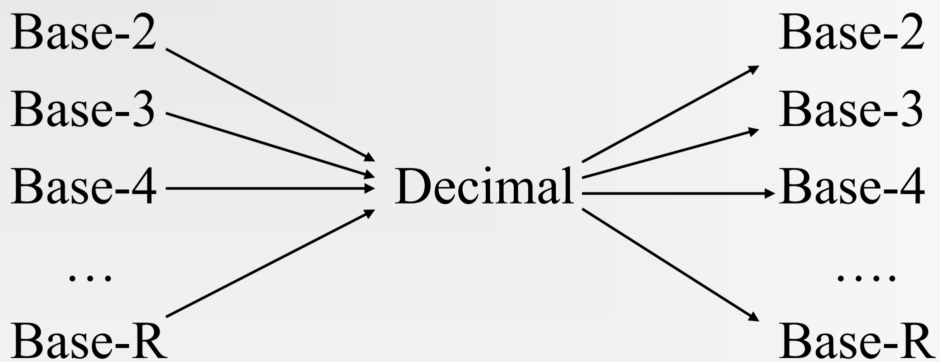
	Carry	
$0.3125 \times 2 = 0.625$	0	←MSB
$0.625 \times 2 = 1.25$	1	
$0.25 \times 2 = 0.50$	0	
$0.5 \times 2 = 1.00$	1	←LSB

Conversion between Decimal and other Bases

- Base-R to decimal: multiply digits with their corresponding weights.
- Decimal to binary (base 2)
  - whole numbers: repeated division-by-2
  - fractions: repeated multiplication-by-2
- Decimal to base-R
  - whole numbers: repeated division-by-R
  - fractions: repeated multiplication-by-R

Conversion between Bases

- In general, conversion between bases can be done via decimal:



- Shortcuts for conversion between bases 2, 8, 16.

Binary-Octal/Hexadecimal Conversion

- Binary $\rightarrow$ Octal: Partition in groups of 3

  $(10\ 111\ 011\ 001\ .\ 101\ 110)_2 = (2731.56)_8$

- Octal $\rightarrow$ Binary: reverse

  $(2731.56)_8 = (10\ 111\ 011\ 001\ .\ 101\ 110)_2$

- Binary $\rightarrow$ Hexadecimal: Partition in groups of 4

  $(101\ 1101\ 1001\ .\ 1011\ 1000)_2 = (5D9.B8)_{16}$

- Hexadecimal $\rightarrow$ Binary: reverse

  $(5D9.B8)_{16} = (101\ 1101\ 1001\ .\ 1011\ 1000)_2$

Binary Arithmetic Operations (1/6)

- ADDITION
- Like decimal numbers, two numbers can be added by adding each pair of digits together with carry propagation.

$$
\begin{array}{r}
(11011)_2 \\
+\ (10011)_2 \\
\hline
(101110)_2 \\
\hline
\end{array}
\qquad
\begin{array}{r}
(647)_{10} \\
+\ (537)_{10} \\
\hline
(1184)_{10} \\
\hline
\end{array}
$$

Binary Arithmetic Operations (2/6)

- Digit addition table:

BINARY	DECIMAL
0 + 0 + 0 = 0 0	0 + 0 + 0 = 0 0
0 + 1 + 0 = 0 1	0 + 1 + 0 = 0 1
1 + 0 + 0 = 0 1	0 + 2 + 0 = 0 2
1 + 1 + 0 = 1 0	…
0 + 0 + 1 = 0 1	1 + 8 + 0 = 0 9
0 + 1 + 1 = 1 0	1 + 9 + 0 = 1 0
1 + 0 + 1 = 1 0	…
1 + 1 + 1 = 1 1	9 + 9 + 1 = 1 9

Carry in

Carry out

$$
\begin{array}{r}
(11011)_2 \\
+\ (10011)_2 \\
\hline
(101110)_2 \\
\hline
\end{array}
$$

1	0	0	1	1	0
0	1	1	0	1	1
0	1	0	0	1	1
1	0	1	1	1	0
0	1	0	0	1	1

Binary Arithmetic Operations (3/6)

- SUBTRACTION
- Two numbers can be subtracted by subtracting each pair of digits together with borrowing, where needed.

$$\begin{array}{r} (11001)_2 \\ -\ (10011)_2 \\ \hline (00110)_2 \\ \hline \end{array} \qquad \begin{array}{r} (627)_{10} \\ -\ (537)_{10} \\ \hline (090)_{10} \\ \hline \end{array}$$

Binary Arithmetic Operations (4/6)

- Digit subtraction table:

BINARY	DECIMAL
0 - 0 - 0 = 0 0	0 - 0 - 0 = 0 0
0 - 1 - 0 = 1 1	0 - 1 - 0 = 1 9
1 - 0 - 0 = 0 1	0 - 2 - 0 = 1 8
1 - 1 - 0 = 0 0	…
0 - 0 - 1 = 1 1	0 - 9 - 1 = 1 0
0 - 1 - 1 = 1 0	1 - 0 - 1 = 0 0
1 - 0 - 1 = 0 0	…
1 - 1 - 1 = 1 1	9 - 9 - 1 = 1 9

Borrow

$$
\begin{array}{r}
(11001)_2 \\
-\ (10011)_2 \\
\hline
(00110)_2 \\
\hline
\end{array}
$$

0	0	1	1	0	0
0	1	1	0	0	1
0	1	0	0	1	1
0	0	0	1	1	0
0	0	0	1	1	0

Binary Arithmetic Operations (5/6)

- MULTIPLICATION
- To multiply two numbers, take each digit of the multiplier and multiply it with the multiplicand. This produces a number of partial products which are then added.

$(11001)_2$	$(214)_{10}$	Multiplicand
x $(10101)_2$	x $(152)_{10}$	Multiplier
$(11001)_2$	$(428)_{10}$	Partial products
$(11001)_2$	$(1070)_{10}$	
$+(11001)_2$	$+(214)_{10}$	
$(1000001101)_2$	$(32528)_{10}$	Result

Binary Arithmetic Operations (6/6)

- Digit multiplication table:

BINARY	DECIMAL
0 X 0 = 0	0 X 0= 0
0 X 1= 0	0 X 1= 0
1 X 0 = 0	…
1 X 1= 1	1 X 8 = 8
	1 X 9= 9
	…
	9 X 8 = 72
	9 X 9 = 81

Negative Numbers Representation

- Unsigned numbers: only non-negative values.
- Signed numbers: include all values (positive and negative).
- Till now, we have only considered how unsigned (non-negative) numbers can be represented. There are three common ways of representing signed numbers (positive and negative numbers) for binary numbers:
  - Sign-and-Magnitude
  - 1s Complement
  - 2s Complement

Negative Numbers: Sign-and-Magnitude (1/4)

- Negative numbers are usually written by writing a minus sign in front.
  - Example:

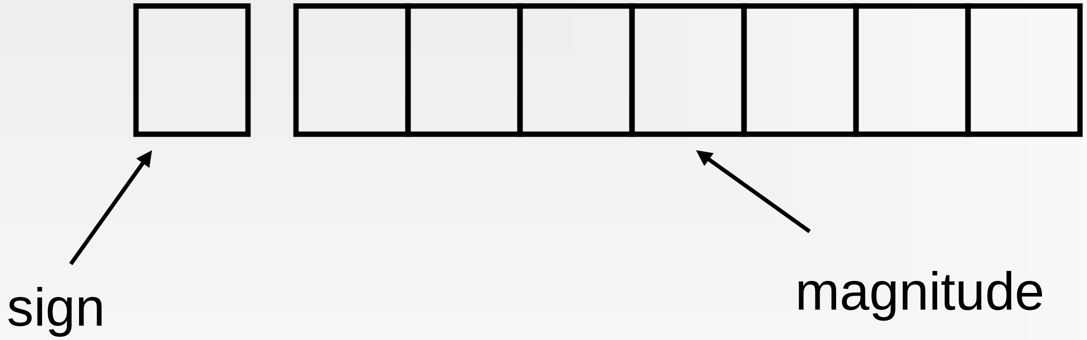
    $- (12)_{10}$, $- (1100)_{2}$
- In sign-and-magnitude representation, this sign is usually represented by a bit:

    0 for +

    1 for -

Negative Numbers: Sign-and-Magnitude (2/4)

- Example: an 8-bit number can have 1-bit sign and 7-bit magnitude.



Negative Numbers: Sign-and-Magnitude (3/4)

- Largest Positive Number: 0 1111111 $+(127)_{10}$
- Largest Negative Number: 1 1111111 $-(127)_{10}$
- Zeroes: 0 0000000 $+(0)_{10}$
  1 0000000 $-(0)_{10}$
- Range: $-(127)_{10}$ to $+(127)_{10}$
- Question: For an *n*-bit sign-and-magnitude representation, what is the range of values that can be represented?

Negative Numbers: Sign-and-Magnitude (4/4)

- To negate a number, just invert the sign bit.
- Examples:
  - $-(0\ 0100001)_{sm} = (1\ 0100001)_{sm}$
  - $-(1\ 0000101)_{sm} = (0\ 0000101)_{sm}$

1s and 2s Complement

- Two other ways of representing signed numbers for binary numbers are:
  - 1s-complement
  - 2s-complement
- They are preferred over the simple sign-and-magnitude representation.

1s Complement (1/3)

- Given a number *x* which can be expressed as an *n*-bit binary number, its negative value can be obtained in 1s-complement representation using:

  $$- x = 2^n - x - 1$$

  Example: With an 8-bit number 00001100, its negative value, expressed in 1s complement, is obtained as follows:

  $$\begin{aligned} -(00001100)_2 &= -(12)_{10} \\ &= (2^8 - 12 - 1)_{10} \\ &= (243)_{10} \\ &= \mathbf{(11110011)_{1s}} \end{aligned}$$

1s Complement (2/3)

- Essential technique: invert all the bits.

  Examples: 1s complement of $(00000001)_{1s} = (11111110)_{1s}$

  1s complement of $(01111111)_{1s} = (10000000)_{1s}$

- Largest Positive Number: 0 1111111 $+(127)_{10}$
- Largest Negative Number: 1 0000000 $-(127)_{10}$
- Zeroes: 0 0000000

  1 1111111

- Range: $-(127)_{10}$ to $+(127)_{10}$
- The most significant bit still represents the sign:

  0 = +ve; 1 = -ve.

1s Complement (3/3)

- Examples (assuming 8-bit binary numbers):

  $(14)_{10} = (00001110)_2 = (00001110)_{1s}$

  $-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

  $-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{1s}$

2s Complement (1/4)

- Given a number *x* which can be expressed as an *n*-bit binary number, its negative number can be obtained in 2s-complement representation using:

$$-x = 2^n - x$$

Example: With an 8-bit number 00001100, its negative value in 2s complement is thus:

$$\begin{aligned} -(00001100)_2 &= -(12)_{10} \\ &= (2^8 - 12)_{10} \\ &= (244)_{10} \\ &= \mathbf{(11110100)_{2s}} \end{aligned}$$

2s Complement (2/4)

- Essential technique: invert all the bits and add 1.

Examples:

2s complement of

$(00000001)_{2s}$ $= (11111110)_{1s}$ (invert)

$= (11111111)_{2s}$ (add 1)

2s complement of

$(01111110)_{2s}$ $= (10000001)_{1s}$ (invert)

$= (10000010)_{2s}$ (add 1)

2s Complement (3/4)

- Largest Positive Number: 0 1111111 $+(127)_{10}$
- Largest Negative Number: 1 0000000 $-(128)_{10}$
- Zero: 0 0000000
- Range: $-(128)_{10}$ to $+(127)_{10}$
- The most significant bit still represents the sign:

  0 = +ve; 1 = -ve.

2s Complement (4/4)

- Examples (assuming 8-bit binary numbers):

  $(14)_{10} = (00001110)_2 = (00001110)_{2s}$

  $-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$

  $-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{2s}$

Comparisons of Sign-and-Magnitude and Complements (1/2)

- Example: 4-bit signed number (*positive values*)

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Important slide!
Mark this!

Comparisons of Sign-and-Magnitude and Complements (2/2)

- Example: 4-bit signed number (*negative values*)

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

Important slide!
Mark this!

2s Complement Addition/Subtraction (1/3)

- Algorithm for addition, A + B:

1. Perform binary addition on the two numbers.
2. Ignore the carry out of the MSB (most significant bit).
3. Check for overflow: Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.

- Algorithm for subtraction, A – B:

  A – B = A + (–B)

1. Take 2s complement of B by inverting all the bits and adding 1.
2. Add the 2s complement of B to A.

2s Complement Addition/Subtraction (2/3)

- Examples: 4-bit binary system

```
  +3         0011
+ +4       + 0100
----       -------
  +7         0111
----       -------
```

```
  -2         1110
+ -6       + 1010
----       -------
  -8        11000
----       -------
```

```
  +6        0110
+ -3      + 1101
----       -------
  +3      10011
----       -------
```

```
  +4         0100
+ -7       + 1001
----       -------
  -3         1101
----       -------
```

- Which of the above is/are overflow(s)?

2s Complement Addition/Subtraction (3/3)

- More examples: 4-bit binary system

```
   -3          1101
 + -6        + 1010
 ----        -------
   -9         10111
 ----        -------
```

```
   +5          0101
 + +6        + 0110
 ----        -------
  +11          1011
 ----        -------
```

- Which of the above is/are overflow(s)?

1s Complement Addition/Subtraction (1/2)

- Algorithm for addition, A + B:

1. Perform binary addition on the two numbers.
2. If there is a carry out of the MSB, add 1 to the result.
3. Check for overflow: Overflow occurs if result is opposite sign of A and B.

- Algorithm for subtraction, A – B:

  A – B = A + (–B)

1. Take 1s complement of B by inverting all the bits.
2. Add the 1s complement of B to A.

1s Complement Addition/Subtraction (2/2)

- Examples: 4-bit binary system

```
  +3        0011
+ +4      + 0100
----      -------
  +7        0111
----      -------
```

```
  +5        0101
+ -5      + 1010
----      -------
  -0        1111
----      -------
```

```
  -2       1101
+ -5   +   1010
----      ------
  -7      10111
----     +    1
          ------
           1000
```

```
  -3        1100
+ -7      + 1000
----      -------
 -10       10100
----      +    1
          -------
            0101
```

Overflow (1/2)

- Signed binary numbers are of a fixed range.
- If the result of addition/subtraction goes beyond this range, overflow occurs.
- Two conditions under which overflow can occur are:

  (i) *positive add positive* gives negative

  (ii) *negative add negative* gives positive

Overflow and Carry Conditions

- *Carry flag*: set when the result of an addition or subtraction exceeds fixed number of bits allocated
- *Overflow*: result of addition or subtraction overflows into the sign bit

Overflow/Carry Examples

- Example 1:
  - Correct result
  - No overflow, no carry

0100	=	(+ 4)
0010	=	+ (+ 2)
0110	=	(+ 6)

- Example 2:
  - Incorrect result
  - Overflow, no carry

0100	=	(+ 4)
0110	=	+ (+ 6)
1010	=	(– 6)

Overflow/Carry Examples

- Example 3:
  - Result correct ignoring the carry
  - Carry but no overflow
- Example 4:
  - Incorrect result
  - Overflow, carry ignored

$$\begin{array}{rcr} 1100 & = & (-4) \\ 1110 & = & +(-2) \\ \hline 11010 & = & (-6) \end{array}$$

$$\begin{array}{rcr} 1100 & = & (-4) \\ 1010 & = & +(-6) \\ \hline 10110 & = & (+3) \end{array}$$

Overflow (2/2)

- Examples: 4-bit numbers (in 2s complement)
- Range : $(1000)_{2s}$ to $(0111)_{2s}$ or (-8_{10} to 7_{10})

(i) $(0101)_{2s} + (0110)_{2s} = (1011)_{2s}$

$(5)_{10} + (6)_{10} = -(5)_{10}$?! (overflow!)

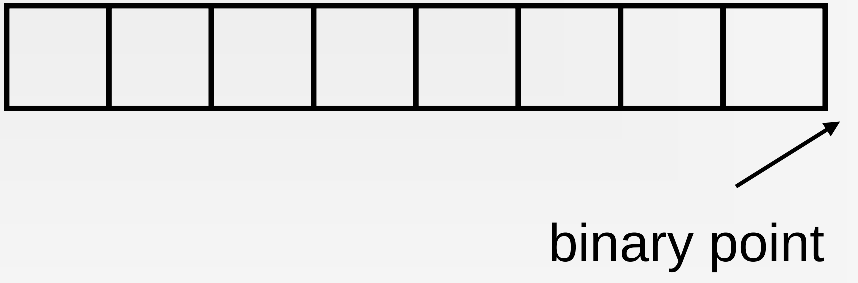
(ii) $(1001)_{2s} + (1101)_{2s} = (\underline{1}0110)_{2s}$ discard end-carry

$= (0110)_{2s}$

$(-7)_{10} + (-3)_{10} = (6)_{10}$?! (overflow!)

Fixed Point Numbers (1/2)

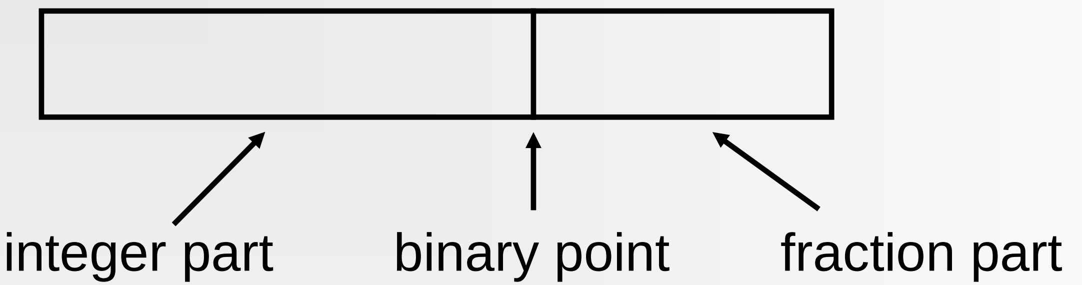
- The signed and unsigned numbers representation given are fixed point numbers.
- The binary point is assumed to be at a fixed location, say, at the end of the number:



- Can represent all integers between –128 to 127 (for 8 bits).

Fixed Point Numbers (2/2)

- In general, other locations for binary points possible.



- Examples: If two fractional bits are used, we can represent:

  $(001010.11)_{2s} = (10.75)_{10}$

  $(111110.11)_{2s} = -(000001.01)_2$

  $= -(1.25)_{10}$

Floating Point Numbers (1/5)

- Fixed point numbers have limited range.
- To represent very large or very small numbers, we use floating point numbers (cf. scientific numbers). Examples:

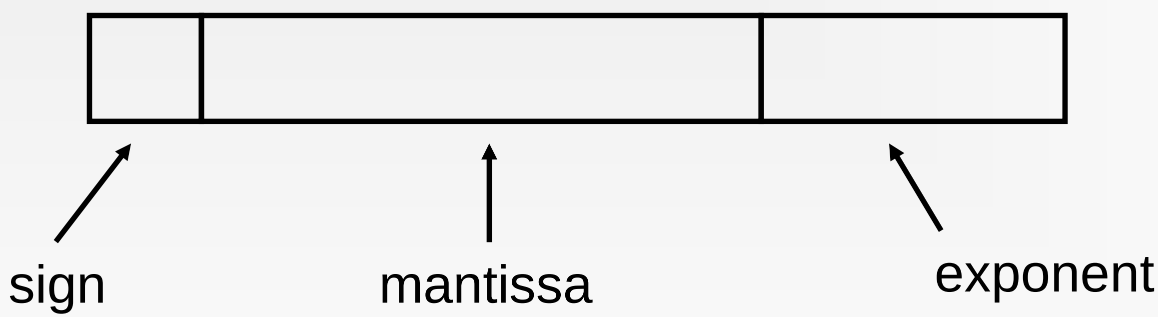
  0.23×10^{23} (very large positive number)

  0.5×10^{-32} (very small positive number)

  -0.1239×10^{-18} (very small negative number)

Floating Point Numbers (2/5)

- Floating point numbers have three parts:
  sign, mantissa, and exponent
- The base (radix) is assumed (usually base 2).
- The sign is a single bit (0 for positive number, 1 for negative).



Floating Point Numbers (3/5)

- Mantissa is usually in normalised form:

  (base 10) 23 x 10^{21} normalised to 0.$\boxed{23}$ x 10^{23}

  (base 10) -0.0017 x 10^{21} normalised to -0.$\boxed{17}$ x 10^{19}

  (base 2) 0.01101 x 2^{3} normalised to 0.$\boxed{1101}$ x 2^{2}

- Normalised form: The fraction portion cannot begin with zero.
- More bits in exponent gives larger range.
- More bits for mantissa gives better precision.

Floating Point Numbers (4/5)

- Exponent is usually expressed in complement or excess form (excess form to be discussed later).
- Example: Express $-(6.5)_{10}$ in base-2 normalised form

  $-(6.5)_{10} = -(110.1)_2 = -0.1101 \times 2^3$

- Assuming that the floating-point representation contains 1-bit sign, 5-bit normalised mantissa, and 4-bit exponent.
- The above example will be represented as

1	11010	0011

Floating Point Numbers (5/5)

- Example: Express $(0.1875)_{10}$ in base-2 normalised form

  $(0.1875)_{10} = (0.0011)_2 = 0.11 \times 2^{-2}$

- Assuming that the floating-pt rep. contains 1-bit sign, 5-bit normalised mantissa, and 4-bit exponent.
- The above example will be represented as

0	11000	1101	If exponent is in 1's complement.
0	11000	1110	If exponent is in 2's complement.
0	11000	0110	If exponent is in excess-8.

Excess Representation (1/2)

- The excess representation allows the range of values to be distributed <u>evenly</u> among the positive and negative value, by a simple translation (addition/subtraction).
- Example: For a 3-bit representation, we may use excess-4.

Excess-4 Representation	*Value*
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

Excess Representation (2/2)

- Example: For a 4-bit representation, we may use excess-8.

Excess-8 Representation	*Value*
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

Excess-8 Representation	*Value*
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

Arithmetics with Floating Point Numbers (1/2)

- Arithmetic is more difficult for floating point numbers.
- MULTIPLICATION

  Steps: (i) multiply the mantissa

  (ii) add-up the exponents

  (iii) normalise
- Example:

  $(0.12 \times 10^{2})_{10} \times (0.2 \times 10^{30})_{10}$

  $= (0.12 \times 0.2)_{10} \times 10^{2+30}$

  $= (0.024)_{10} \times 10^{32}$ (normalise)

  $= (0.24 \times 10^{31})_{10}$

Arithmetics with Floating Point Numbers (2/2)

- ADDITION

  Steps: (i) equalise the exponents

  (ii) add-up the mantissa

  (iii) normalise

- Example:

  $(0.12 \times 10^3)_{10} + (0.2 \times 10^2)_{10}$

  $= (0.12 \times 10^3)_{10} + (0.02 \times 10^3)_{10}$ (equalise exponents)

  $= (0.12 + 0.02)_{10} \times 10^3$ (add mantissa)

  $= (0.14 \times 10^3)_{10}$

- Can you figure out how to do perform SUBTRACTION and DIVISION for (binary/decimal) floating-point numbers?

Number Systems & Codes

Part II: Codes

- Binary Coded Decimal (BCD)
- Gray Code
  - Binary-to-Gray Conversion
  - Gray-to-Binary Conversion
- Other Decimal Codes
- Self-Complementing Codes
- Alphanumeric Codes
- Error Detection Codes

Binary Coded Decimal (BCD) (1/3)

- Decimal numbers are more natural to humans. Binary numbers are natural to computers. Quite expensive to convert between the two.
- If little calculation is involved, we can use some *coding schemes* for decimal numbers.
- Represent each decimal digit as a 4-bit binary code.

Binary Coded Decimal (BCD) (2/3)

Decimal digit	0	1	2	3	4
BCD	0000	0001	0010	0011	0100
Decimal digit	5	6	7	8	9
BCD	0101	0110	0111	1000	1001

- Some codes are unused, eg: $(1010)_{BCD}$, $(1011)_{BCD}$, …, $(1111)_{BCD}$. These codes are considered as errors.
- Easy to convert, but arithmetic operations are more complicated.
- Suitable for interfaces such as keypad inputs and digital readouts.

Binary Coded Decimal (BCD) (3/3)

Decimal digit **BCD**	0 **0000**	1 **0001**	2 **0010**	3 **0011**	4 **0100**
Decimal digit **BCD**	5 **0101**	6 **0110**	7 **0111**	8 **1000**	9 **1001**

- Examples:

  $(234)_{10} = (0010\ 0011\ 0100)_{BCD}$

  $(7093)_{10} = (0111\ 0000\ 1001\ 0011)_{BCD}$

  $(1000\ 0110)_{BCD} = (86)_{10}$

  $(1001\ 0100\ 0111\ 0010)_{BCD} = (9472)_{10}$

Notes: BCD is not equivalent to binary.

Example: $(234)_{10} = (11101010)_2$

The Gray Code (1/3)

- Unweighted (not an arithmetic code).
- Only a *single bit change* from one code number to the next.
- Good for error detection.

Decimal	Binary	Gray Code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Q. How to generate 5-bit standard Gray code?

Q. How to generate *n*-bit standard Gray code?

The Gray Code (2/3)

```
0000          1100
0001          1101
0011          1111
0010          1110
0110          1010
0111          1011
0101          1001
0100          1000
```

Generating <u>4-bit standard</u> Gray code.

Binary-to-Gray Code Conversion

- Retain most significant bit.
- From left to right, add each adjacent pair of binary code bits to get the next Gray code bit, discarding carries.
- Example: Convert binary number 10110 to Gray code.

1	0	1	1	0	Binary
↓					
1					Gray

1	+	0	1	1	0	Binary
		↓				
1		1				Gray

1	0	+	1	1	0	Binary
			↓			
1	1		1			Gray

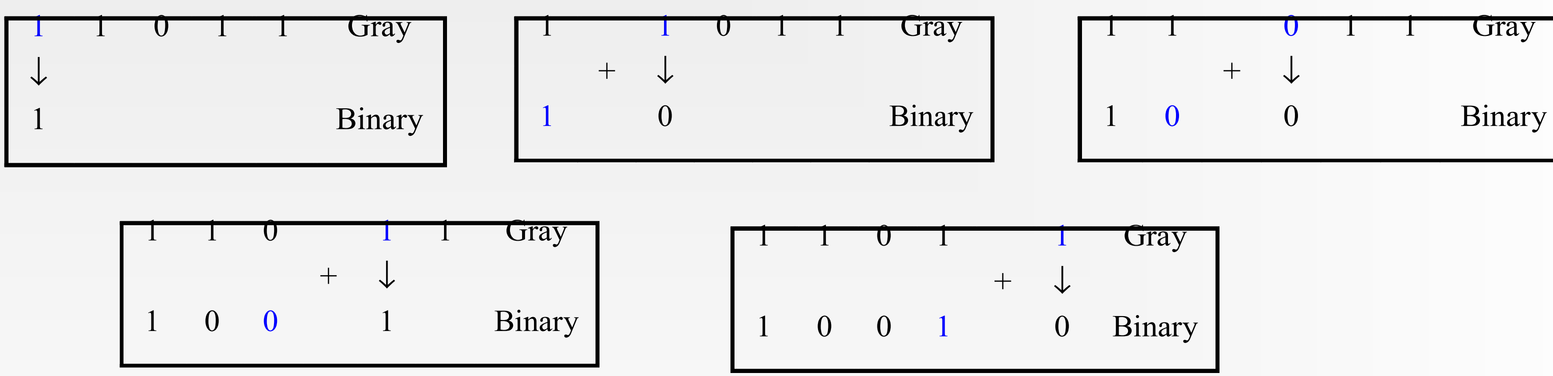
1	0	1	+	1	0	Binary
				↓		
1	1	1		0		Gray

1	0	1	1	+	0	Binary
					↓	
1	1	1	0		1	Gray

$$(10110)_2 = (11101)_{Gray}$$

Gray-to-Binary Conversion

- Retain most significant bit.
- From left to right, add each binary code bit generated to the Gray code bit in the next position, discarding carries.
- Example: Convert Gray code 11011 to binary.



$$(11011)_{Gray} = (10010)_2$$

Alphanumeric Codes (1/3)

- Apart from numbers, computers also handle textual data.
- Character set frequently used includes:

alphabets:	‘A’ .. ‘Z’, and ‘a’ .. ‘z’
digits:	‘0’ .. ‘9’
special symbols:	‘$’, ‘.’, ‘,’, ‘@’, ‘*’, …
non-printable:	SOH, NULL, BELL, …

- Usually, these characters can be represented using 7 or 8 bits.

Alphanumeric Codes (2/3)

- ASCII: 7-bit, plus a *parity bit* for error detection (odd/even parity).

Character	ASCII Code
0	0110000
1	0110001
. . .	. . .
9	0111001
:	0111010
A	1000001
B	1000010
. . .	. . .
Z	1011010
[	1011011
\	1011100

Alphanumeric Codes (3/3)

- ASCII table:

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC$_1$	!	1	A	Q	a	q
0010	STX	DC$_2$	“	2	B	R	b	r
0011	ETX	DC$_3$	#	3	C	S	c	s
0100	EOT	DC$_4$	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Error Detection Codes (1/4)

- Errors can occur data transmission. They should be detected, so that re-transmission can be requested.
- With binary numbers, usually single-bit errors occur.

  Example: 0010 erroneously transmitted as 0011, or 0000, or 0110, or 1010.

- Biquinary code uses 3 additional bits for error-detection. For single-error detection, one additional bit is needed.

Error Detection Codes (2/4)

- Parity bit.
  - Even parity: additional bit supplied to make total number of '1's even.
  - Odd parity: additional bit supplied to make total number of '1's odd.
- Example: Odd parity.

Character	ASCII Code
0	0110000 1
1	0110001 0
. . .	. . .
9	0111001 1
:	0111010 1
A	1000001 1
B	1000010 1
. . .	. . .
Z	1011010 1
[	1011011 0
\	1011100 1

Parity bits

Error Detection Codes (3/4)

- Parity bit can detect odd number of errors but not even number of errors.

  Example: For odd parity numbers,

  10011 $\rightarrow$ 10001 (detected)

  10011 $\rightarrow$ 10101 (non detected)

- Parity bits can also be applied to a block of data:

0110 1
0001 0
1011 0
1111 1
1001 1
0101 0

Column-wise parity

Row-wise parity

Error Detection Codes (4/4)

- Sometimes, it is not enough to do error detection. We may want to do error correction.
- Error correction is expensive. In practice, we may use only single-bit error correction.
- Popular technique: Hamming Code (not covered).

Thank You