Number Representation

Number System :: The Basics

- We are accustomed to using the so-called decimal number system
  - Ten digits :: 0,1,2,3,4,5,6,7,8,9
  - Every digit position has a weight which is a power of 10
  - Base or radix is 10

Example:

$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

$250.67 = 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$

Binary Number System

- Two digits:
  - 0 and 1
  - Every digit position has a weight which is a power of 2
  - Base or radix is 2
- Example:

$110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Positional Number Systems (General)

Decimal Numbers:

- **10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10**
- **$136.25 = 1 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 2 \times 10^{-1} + 3 \times 10^{-2}$**

Positional Number Systems (General)

Decimal Numbers:

- **10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10**
- **$136.25 = 1 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2}$**

Binary Numbers:

- **2 Symbols {0,1}, Base or Radix is 2**
- **$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$**

Positional Number Systems (General)

Decimal Numbers:

- **10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10**
- **$136.25 = 1 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 2 \times 10^{-1} + 5 \times 10^{-2}$**

Binary Numbers:

- **2 Symbols {0,1}, Base or Radix is 2**
- **$101.01 = 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} + 0 \times 2^{-1} + 1 \times 2^{-2}$**

Octal Numbers:

- **8 Symbols {0,1,2,3,4,5,6,7}, Base or Radix is 8**
- **$621.03 = 6 \times 8^{2} + 2 \times 8^{1} + 1 \times 8^{0} + 0 \times 8^{-1} + 3 \times 8^{-2}$**

Positional Number Systems (General)

Decimal Numbers:

- **10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10**
- **$136.25 = 1 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 2 \times 10^{-1} + 3 \times 10^{-2}$**

Binary Numbers:

- **2 Symbols {0,1}, Base or Radix is 2**
- **$101.01 = 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} + 0 \times 2^{-1} + 1 \times 2^{-2}$**

Octal Numbers:

- **8 Symbols {0,1,2,3,4,5,6,7}, Base or Radix is 8**
- **$621.03 = 6 \times 8^{2} + 2 \times 8^{1} + 1 \times 8^{0} + 0 \times 8^{-1} + 3 \times 8^{-2}$**

Hexadecimal Numbers:

- **16 Symbols {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}, Base is 16**
- **$6AF.3C = 6 \times 16^{2} + 10 \times 16^{1} + 15 \times 16^{0} + 3 \times 16^{-1} + 12 \times 16^{-2}$**

Binary-to-Decimal Conversion

- Each digit position of a binary number has a weight
  - Some power of 2
- A binary number:

$$B = b_{n-1}\ b_{n-2}\ \ldots\ldots b_1\ b_0\ .\ b_{-1}\ b_{-2}\ \ldots\ldots\ b_{-m}$$

Corresponding value in decimal:

$$D = \sum_{i=-m}^{n-1} b_i\, 2^i$$

Examples

101011 ➔ $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$= 43$

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

.0101 ➔ $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$

$= .3125$

$(.0101)_2 = (.3125)_{10}$

101.11 ➔ $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$

$= 5.75$

$(101.11)_2 = (5.75)_{10}$

Decimal to Binary: Integer Part

- Consider the integer and fractional parts separately.
- For the integer part:
  - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
  - Arrange the remainders in reverse order.

Base	Numb	Rem
2	89	
2	44	1
2	22	0
2	11	0
2	5	1
2	2	1
2	1	0
	0	1

$(89)_{10} = (1011001)_2$

Decimal to Binary: Integer Part

- Consider the integer and fractional parts separately.
- For the integer part:
  - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
  - Arrange the remainders in reverse order.

Base	Numb	Rem
2	89	
2	44	1
2	22	0
2	11	0
2	5	1
2	2	1
2	1	0
	0	1

$(89)_{10} = (1011001)_2$

2	66	
2	33	0
2	16	1
2	8	0
2	4	0
2	2	0
2	1	0
	0	1

$(66)_{10} = (1000010)_2$

Decimal to Binary: Integer Part

- Consider the integer and fractional parts separately.
- For the integer part:
  - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
  - Arrange the remainders in reverse order.

Base	Numb	Rem
2	89	
2	44	1
2	22	0
2	11	0
2	5	1
2	2	1
2	1	0
	0	1

$(89)_{10} = (1011001)_2$

2	66	
2	33	0
2	16	1
2	8	0
2	4	0
2	2	0
2	1	0
	0	1

$(66)_{10} = (1000010)_2$

2	239	
2	119	1
2	59	1
2	29	1
2	14	1
2	7	0
2	3	1
2	1	1
	0	1

$(239)_{10} = (11101111)_2$

Decimal to Binary: Fraction Part

- Repeatedly multiply the given fraction by 2.
  - Accumulate the integer part (0 or 1).
  - If the integer part is 1, chop it off.
- Arrange the integer parts in the order they are obtained.

Example: 0.634

$$.634 \times 2 = 1.268$$
$$.268 \times 2 = 0.536$$
$$.536 \times 2 = 1.072$$
$$.072 \times 2 = 0.144$$
$$.144 \times 2 = 0.288$$

:

:

$$(.634)_{10} = (.10100\ldots\ldots)_2$$

Decimal to Binary: Fraction Part

- Repeatedly multiply the given fraction by 2.
  - Accumulate the integer part (0 or 1).
  - If the integer part is 1, chop it off.
- Arrange the integer parts in the order they are obtained.

Example: 0.634

$.634 \times 2 = 1.268$

$.268 \times 2 = 0.536$

$.536 \times 2 = 1.072$

$.072 \times 2 = 0.144$

$.144 \times 2 = 0.288$

:

:

$(.634)_{10} = (.10100......)_2$

Example: 0.0625

$.0625 \times 2 = 0.125$

$.1250 \times 2 = 0.250$

$.2500 \times 2 = 0.500$

$.5000 \times 2 = 1.000$

$(.0625)_{10} = (.0001)_2$

Decimal to Binary: Fraction Part

- Repeatedly multiply the given fraction by 2.
  - Accumulate the integer part (0 or 1).
  - If the integer part is 1, chop it off.
- Arrange the integer parts in the order they are obtained.

Example: 0.634

$.634 \times 2 = 1.268$

$.268 \times 2 = 0.536$

$.536 \times 2 = 1.072$

$.072 \times 2 = 0.144$

$.144 \times 2 = 0.288$

:

:

$(.634)_{10} = (.10100\ldots\ldots)_2$

Example: 0.0625

$.0625 \times 2 = 0.125$

$.1250 \times 2 = 0.250$

$.2500 \times 2 = 0.500$

$.5000 \times 2 = 1.000$

$(.0625)_{10} = (.0001)_2$

$(37)_{10} = (100101)_2$

$(.0625)_{10} = (.0001)_2$

$(37.0625)_{10} = (100101.0001)_2$

Hexadecimal Number System

- A compact way of representing binary numbers
- 16 different symbols (radix = 16)

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

Binary-to-Hexadecimal Conversion

- For the integer part,
  - Scan the binary number from right to left
  - Translate each group of four bits into the corresponding hexadecimal digit
    - Add leading zeros if necessary

- For the fractional part,
  - Scan the binary number from left to right
  - Translate each group of four bits into the corresponding hexadecimal digit
    - Add trailing zeros if necessary

Example

1. $(\underline{1011}\ \underline{0100}\ \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10}\ \underline{1010}\ \underline{0001})_2 = (2A1)_{16}$
3. $(.\underline{1000}\ \underline{010})_2 = (.84)_{16}$
4. $(\underline{101}\ .\ \underline{0101}\ \underline{111})_2 = (5.5E)_{16}$

Hexadecimal-to-Binary Conversion

- Translate every hexadecimal digit into its 4-bit binary equivalent

- Examples:

  $(3A5)_{16} = (0011\ 1010\ 0101)_2$

  $(12.3D)_{16} = (0001\ 0010\ .\ 0011\ 1101)_2$

  $(1.8)_{16} = (0001\ .\ 1000)_2$

Unsigned Binary Numbers

- An n-bit binary number

  $B = b_{n-1}b_{n-2} \ldots. b_2b_1b_0$

  - 2^n distinct combinations are possible, 0 to 2^n-1.
- For example, for n = 3, there are 8 distinct combinations
  - 000, 001, 010, 011, 100, 101, 110, 111
- Range of numbers that can be represented

  n=8 ➔ 0 to 2^8-1 (255)

  n=16 ➔ 0 to $2^{16}-1$ (65535)

  n=32 ➔ 0 to $2^{32}-1$ (4294967295)

Signed Integer Representation

- Many of the numerical data items that are used in a program are signed (positive or negative)
  - Question:: How to represent sign?

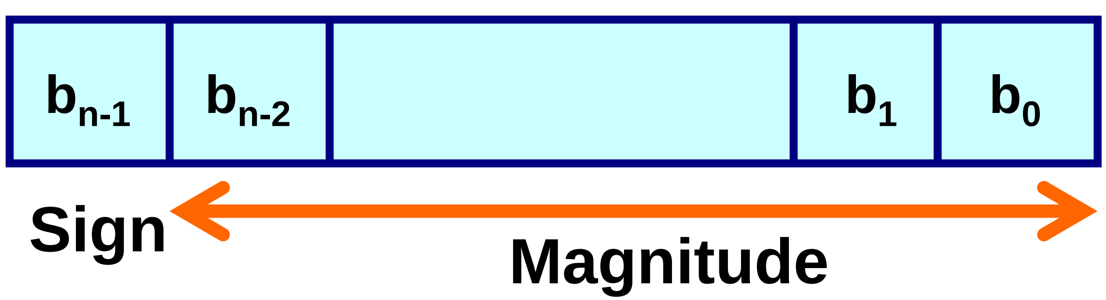
- Three possible approaches:
  - Sign-magnitude representation
  - One’s complement representation
  - Two’s complement representation

Sign-magnitude Representation

- For an n-bit number representation
  - The most significant bit (MSB) indicates sign

    0 → positive

    1 → negative
  - The remaining n-1 bits represent magnitude



Contd.

- Range of numbers that can be represented:

  Maximum :: $+ (2^{n-1} - 1)$

  Minimum :: $- (2^{n-1} - 1)$

- A problem:

  Two different representations of zero

  +0 → 0 000….0

  -0 → 1 000….0

One's Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form
  - Negative numbers are represented in 1's complement form
- How to compute the 1's complement of a number?
  - Complement every bit of the number (1→0 and 0→1)
  - MSB will indicate the sign of the number

    0 → positive

    1 → negative

Example :: n=4

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

To find the representation of, say, -4, first note that

+4 = 0100

-4 = 1's complement of 0100 = 1011

Contd.

- Range of numbers that can be represented:

  Maximum :: $+(2^{n-1} - 1)$

  Minimum :: $-(2^{n-1} - 1)$

- A problem:

  Two different representations of zero.

  +0 → 0 000….0

  -0 → 1 111….1

- Advantage of 1's complement representation
  - Subtraction can be done using addition
  - Leads to substantial saving in circuitry

Two’s Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form
  - Negative numbers are represented in 2’s complement form
- How to compute the 2’s complement of a number?
  - Complement every bit of the number (1→0 and 0→1), and then add one to the resulting number
  - MSB will indicate the sign of the number

    0 → positive

    1 → negative

Example : n=4

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

To find the representation of, say, -4, first note that

+4 = 0100

-4 = 2's complement of 0100 = 1011+1 = 1100

Rule : Value = – msb*2$^{(n–1)}$ + [unsigned value of rest]

Example: 0110 = 0 + 6 = 6 1110 = – 8 + 6 = – 2

Contd.

- Range of numbers that can be represented:

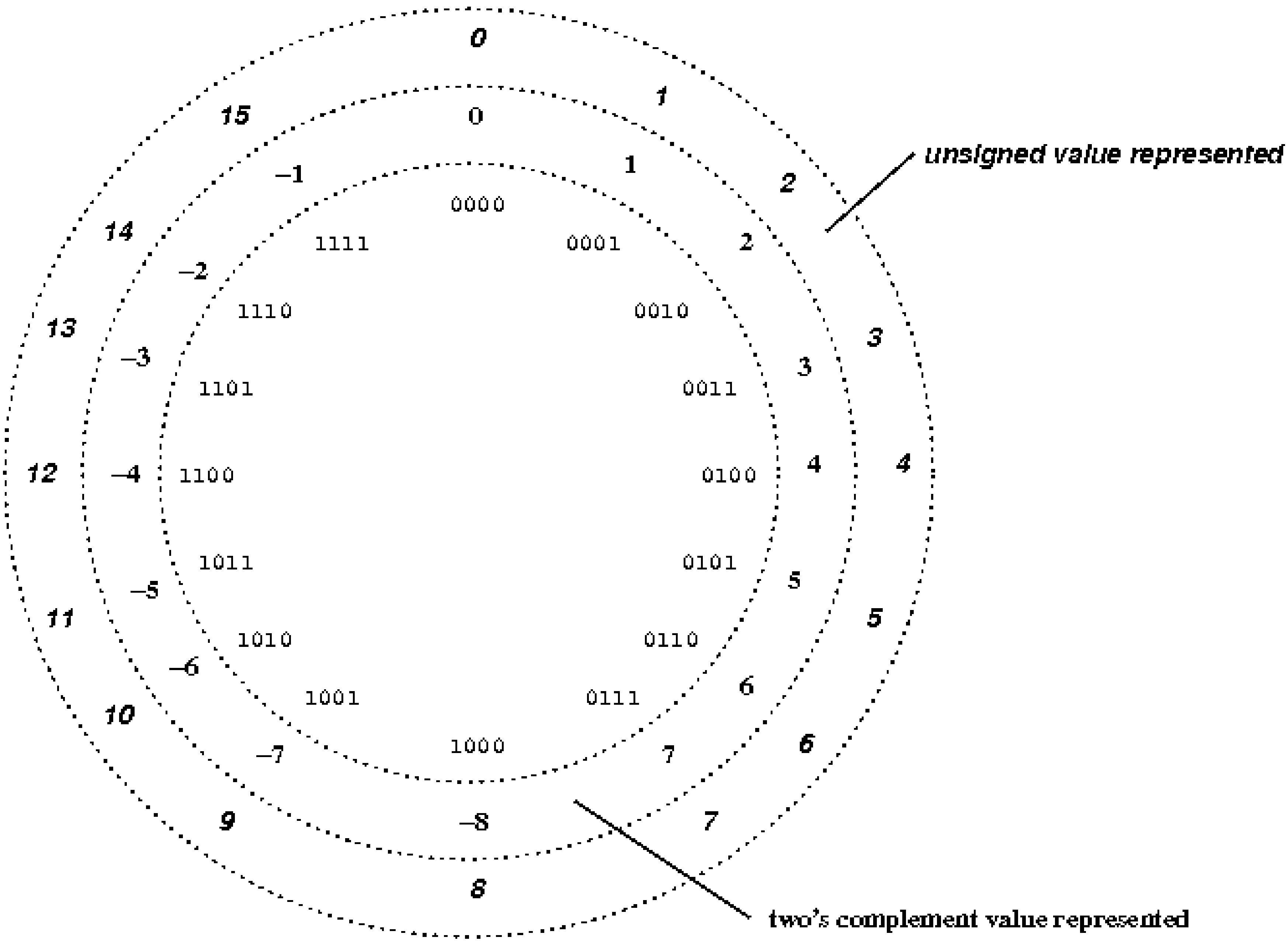
  Maximum :: $+ (2^{n-1} - 1)$

  Minimum :: $- 2^{n-1}$

- Advantage:
  - Unique representation of zero
  - Subtraction can be done using addition
  - Leads to substantial saving in circuitry
- Almost all computers today use the 2's complement representation for storing negative numbers

Contd.

- In C
  - short int
    - 16 bits → $+(2^{15}-1)$ to -2^{15}
  - int or long int
    - 32 bits → $+(2^{31}-1)$ to -2^{31}
  - long long int
    - 64 bits → $+(2^{63}-1)$ to -2^{63}



unsigned value represented
two's complement value represented
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
2
3
4
5
6
7
–8
–7
–6
–5
–4
–3
–2
–1
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Adding Binary Numbers

- Basic Rules:
  - 0+0=0
  - 0+1=1
  - 1+0=1
  - 1+1=0 (carry 1)

- Example:

  01101001

  00110100

  10011101

Subtraction Using Addition :: 1's Complement

- How to compute A – B ?
  - Compute the 1's complement of B (say, B_1).
  - Compute $R = A + B_1$
  - If the carry obtained after addition is '1'
    - Add the carry back to R (called *end-around carry*)
    - That is, R = R + 1
    - The result is a positive number

    Else
    - The result is negative, and is in 1's complement form

Example 1 :: 6 – 2

1's complement of 2 = 1101

6	::	0110	**A**
-2	::	1101	**B_1**
		1 0011	**R**
		1	
		0100	➔ +4

End-around carry

Assume 4-bit representations

Since there is a carry, it is added back to the result

The result is positive

Example 2 :: 3 – 5

1's complement of 5 = 1010

3	::	0011	**A**
-5	::	1010	$\mathbf{B_1}$
		1101	**R**

↓

-2

Assume 4-bit representations

Since there is no carry, the result is negative

1101 is the 1's complement of 0010, that is, it represents –2

Subtraction Using Addition :: 2's Complement

- How to compute A – B ?
  - Compute the 2's complement of B (say, B_2)
  - Compute $R = A + B_2$
  - If the carry obtained after addition is '1'
    - Ignore the carry
    - The result is a positive number

    Else
    - The result is negative, and is in 2's complement form

Example 1 :: 6 – 2

2's complement of 2 = 1101 + 1 = 1110

6	::	0110	**A**
-2	::	1110	**B_2**
		1 0100	**R**

Ignore carry

+4

Assume 4-bit representations

Presence of carry indicates that the result is positive

No need to add the end-around carry like in 1's complement

Example 2 :: 3 – 5

2's complement of 5 = 1010 + 1 = 1011

3	::	0011	A
-5	::	1011	B_2
		1110	R

⬇

-2

Assume 4-bit representations

Since there is no carry, the result is negative

1110 is the 2's complement of 0010, that is, it represents –2

2's complement arithmetic: More Examples

- Example 1: 18-11 = ?
- 18 is represented as 00010010
- 11 is represented as 00001011
  - 1's complement of 11 is 11110100
  - 2's complement of 11 is 11110101
- Add 18 to 2's complement of 11

```
   00010010
+  11110101
---------------
   00000111 (with a carry of 1
                which is ignored)
```

00000111 is 7

- Example 2: 7 - 9 = ?
- 7 is represented as 00000111
- 9 is represented as 00001001
  - 1's complement of 9 is 11110110
  - 2's complement of 9 is 11110111
  - Add 7 to 2's complement of 9

```
   00000111
+  11110111
---------------
   11111110 (with a carry of 0
                which is ignored)
```

11111110 is -2

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a –ve (+ve) number. If it does, overflow (underflow) occurs

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a –ve (+ve) number. If it does, overflow (underflow) occurs

Another equivalent condition : carry in and carry out from Most Significant Bit (MSB) differ.

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a –ve (+ve) number. If it does, overflow (underflow) occurs

Another equivalent condition : carry in and carry out from Most Significant Bit (MSB) differ.

```
(64)  01000000
( 4)  00000100
      --------------
 (68) 01000100
```

carry (out)(in)
0 0

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a –ve (+ve) number. If it does, overflow (underflow) occurs

Another equivalent condition : carry in and carry out from Most Significant Bit (MSB) differ.

```
(64)  01000000
( 4)  00000100
      --------------
 (68) 01000100
```

carry (out)(in)
0 0

```
(64)  01000000
(96)  01100000
      --------------
(-96) 10100000
```

carry out in
0 1

differ:

overflow

Floating-point Numbers

- The representations discussed so far applies only to integers
  - Cannot represent numbers with fractional parts
- We can assume a decimal point before a signed number
  - In that case, pure fractions (without integer parts) can be represented
- We can also assume the decimal point somewhere in between
  - This lacks flexibility
  - Very large and very small numbers cannot be represented

Representation of Floating-Point Numbers

- A floating-point number F is represented by a doublet <M,E> :

  $F = M \times B^E$

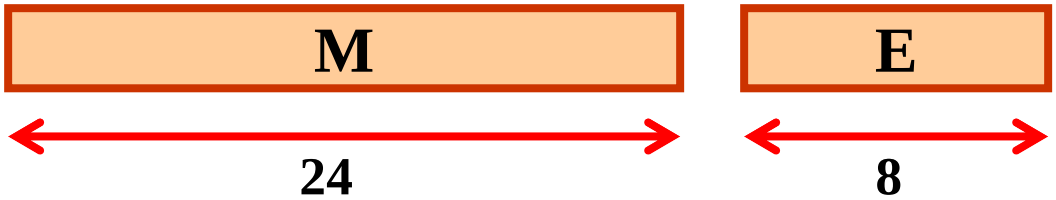
  - B → exponent base (usually 2)
  - M → mantissa
  - E → exponent

  - M is usually represented in 2's complement form, with an implied binary point before it
- For example,

  In decimal, 0.235×10^6

  In binary, 0.101011×2^{0110}

Example :: 32-bit representation



- M represents a 2's complement fraction

  1 > M > -1

- E represents the exponent (in 2's complement form)

  127 > E > -128

- Points to note:
  - The number of significant digits depends on the number of bits in M
    - 6 significant digits for 24-bit mantissa
  - The range of the number depends on the number of bits in E
    - 10^{38} to 10^{-38} for 8-bit exponent.

A Warning

- The representation for floating-point numbers as shown is just for illustration
- The actual representation is a little more complex
- Example: IEEE 754 Floating Point format

IEEE 754 Floating-Point Format (Single Precision)

S (31)	E (Exponent) (30 … 23)	M (Mantissa) (22 … 0)

S: Sign (0 is +ve, 1 is –ve)

E: Exponent (More bits gives a higher range)

M: Mantissa (More bits means higher precision)

[8 bytes are used for double precision]

Value of a Normal Number:

$$(-1)^S \times (1.0 + 0.M) \times 2^{(E - 127)}$$

An example

S (31)	E (Exponent) (30 … 23)	M (Mantissa) (22 … 0)
1	10001100	11011000000000000000000

Value of a Normal Number:

$= (-1)^S \times (1.0 + 0.M) \times 2^{(E-127)}$

$= (-1)^1 \times (1.0 + 0.1101100) \times 2^{(10001100 - 1111111)}$

$= -1.1101100 \times 2^{1101} = -11101100000000$

$= -15104.0$ (in decimal)

Representing 0.3

S	E (Exponent)	M (Mantissa)
(31)	(30 … 23)	(22 … 0)

0.3 (decimal)

= 0.0100100100100100100100100…

= 1.00100100100100100100100100… $\times 2^{-2}$

= 1.00100100100100100100100100… $\times 2^{125-127}$

= $(-1)^S \times (1.0 + 0.M) \times 2^{(E-127)}$

0	01111101	00100100100100100100100

What are the largest and smallest numbers that can be represented in this scheme?

Representing 0

S (31)	E (Exponent) (30 … 23)	M (Mantissa) (22 … 0)
0	00000000	00000000000000000000000
1	00000000	00000000000000000000000

Representing Inf (∝)

0	11111111	00000000000000000000000
1	11111111	00000000000000000000000

Representing NaN (Not a Number)

0	11111111	Non zero
1	11111111	Non zero

More on Data Types

More Data Types in C

- Some of the basic data types can be augmented by using certain data type qualifiers:
  - short
  - long
  - signed
  - unsigned

size qualifier

sign qualifier

- Typical examples:
  - short int (usually 2 bytes)
  - long int (usually 4 bytes)
  - unsigned int (usually 4 bytes, but no way to store + or -)

Some typical sizes (some of these can vary depending on type of machine)

Integer data type	Bit size	Minimum value	Maximum value
char	8	$-2^{7}=-128$	$2^{7}-1=127$
short int	16	$-2^{15}=-32768$	$2^{15}-1=32767$
int	32	$-2^{31}=-2147483648$	$2^{31}-1=2147483647$
long int	32	$-2^{31}=-2147483648$	$2^{31}-1=2147483647$
long long int	64	$-2^{63}=-9223372036854775808$	$2^{63}-1=9223372036854775807$
unsigned char	8	0	$2^{8}-1=255$
unsigned short int	16	0	$2^{16}-1=65535$
unsigned int	32	0	$2^{32}-1=4294967295$
unsigned long int	32	0	$2^{32}-1=4294967295$
unsigned long long int	64	0	$2^{64}-1=18446744073709551615$

The bool type

- Used to store boolean variables, like flags to check if a condition is true or false
- Can take only two values, true and false

```
bool negative = false;
int n;
scanf(%d", &n);
if (n < 0) negative = true;
```

- Internally, false is represented by 0, true is usually represented by 1 but can be different (print a bool variable with %d to see what you get)
- More compact storage internally

Representation of Characters

- Many applications have to deal with non-numerical data.
  - Characters and strings
  - There must be a standard mechanism to represent alphanumeric and other characters in memory
- Three standards in use:
  - Extended Binary Coded Decimal Interchange Code (EBCDIC)
    - Used in older IBM machines
  - American Standard Code for Information Interchange (ASCII)
    - Most widely used today
  - UNICODE
    - Used to represent all international characters.
    - Used by Java

ASCII Code

- Each character is assigned a unique integer value (code) between 32 and 127
- The code of a character is represented by an 8-bit unit. Since an 8-bit unit can hold a total of 2^8=256 values and the computer character set is much smaller than that, some values of this 8-bit unit do not correspond to visible characters

ASCII Code

- Each individual character is numerically encoded into a unique 7-bit binary code
  - A total of 2^7 or 128 different characters
  - A character is normally encoded in a byte (8 bits), with the MSB not been used.
- The binary encoding of the characters follow a regular ordering
  - Digits are ordered consecutively in their proper numerical sequence (0 to 9)
  - Letters (uppercase and lowercase) are arranged consecutively in their proper alphabetic order

Decimal	Hex	Binary	Character	Decimal	Hex	Binary	Character
32	20	00100000	SPACE	80	50	01010000	P
33	21	00100001	!	81	51	01010001	Q
34	22	00100010	"	82	52	01010010	R
35	23	00100011	#	83	53	01010011	S
36	24	00100100	$	84	54	01010100	T
37	25	00100101	%	85	55	01010101	U
38	26	00100110	&	86	56	01010110	V
39	27	00100111	'	87	57	01010111	W
40	28	00101000	(	88	58	01011000	X
41	29	00101001	)	89	59	01011001	Y
42	2a	00101010	*	90	5a	01011010	Z
43	2b	00101011	+	91	5b	01011011	[
44	2c	00101100	,	92	5c	01011100	\
45	2d	00101101	-	93	5d	01011101	]
46	2e	00101110	.	94	5e	01011110	^
47	2f	00101111	/	95	5f	01011111	_
48	30	00110000	0	96	60	01100000	`
49	31	00110001	1	97	61	01100001	a
50	32	00110010	2	98	62	01100010	b

51	33	00110011	3	99	63	01100011	c
52	34	00110100	4	100	64	01100100	d
53	35	00110101	5	101	65	01100101	e
54	36	00110110	6	102	66	01100110	f
55	37	00110111	7	103	67	01100111	g
56	38	00111000	8	104	68	01101000	h
57	39	00111001	9	105	69	01101001	i
58	3a	00111010	:	106	6a	01101010	j
59	3b	00111011	;	107	6b	01101011	k
60	3c	00111100	<	108	6c	01101100	l
61	3d	00111101	=	109	6d	01101101	m
62	3e	00111110	>	110	6e	01101110	n
63	3f	00111111	?	111	6f	01101111	o
64	40	01000000	@	112	70	01110000	p
65	41	01000001	A	113	71	01110001	q
66	42	01000010	B	114	72	01110010	r
67	43	01000011	C	115	73	01110011	s
68	44	01000100	D	116	74	01110100	t
69	45	01000101	E	117	75	01110101	u
70	46	01000110	F	118	76	01110110	v

71	47	01000111	G		119	77	01110111	w
72	48	01001000	H		120	78	01111000	x
73	49	01001001	I		121	79	01111001	y
74	4a	01001010	J		122	7a	01111010	z
75	4b	01001011	K		123	7b	01111011	{
76	4c	01001100	L		124	7c	01111100	|
77	4d	01001101	M		125	7d	01111101	}
78	4e	01001110	N		126	7e	01111110	~
79	4f	01001111	O		127	7f	01111111	DELETE

More on the char type

- Is actually an integer type internally
- Each character has an integer code associated with it (ASCII code value)
- Internally, storing a character means storing its integer code
- All operators that are allowed on int are allowed on char
  - 32 + ‘a’ will evaluate to 32 + 97 (the integer ascii code of the character ‘a’) = 129
  - Same for other operators
- Can switch on chars constants in switch, as they are integer constants

Another example

```
int a;
a = ‘c’ * 3 + 5;
printf(“%d”, a);
```

Will print 296 (97*3 + 5)

(ASCII code of ‘c’ = 97

```
char c = ‘A’;
printf(“%c = %d”, c, c);
```

Will print A = 65

(ASCII code of ‘A’ = 65)

Assigning char to int is fine. But other way round is dangerous, as size of int is larger

Switching with char type

```
char letter;
scanf(“%c”, &letter);
switch ( letter ) {
    case 'A':
            printf ("First letter \n");
            break;
    case 'Z':
            printf ("Last letter \n");
            break;
    default :
            printf ("Middle letter \n");
}
```

Switching with char type

```
char letter;
scanf(“%c”, &letter);
switch ( letter ) {
    case 'A':
        printf ("First letter \n");
        break;
    case 'Z':
        printf ("Last letter \n");
        break;
    default :
        printf ("Middle letter \n");
}
```

Will print this statement for all letters other than A or Z

Another Example

```
switch (choice = getchar()) {
    case ‘r’ :
    case ‘R’: printf(“Red”);
              break;
    case ‘b’ :
    case ‘B’ : printf(“Blue”);
               break;
    case ‘g’ :
    case ‘G’: printf(“Green”);
               break;
    default: printf(“Black”);
}
```

Another Example

```
switch (choice = getchar()) {
    case ‘r’ :
    case ‘R’: printf(“Red”);
              break;
    case ‘b’ :
    case ‘B’ : printf(“Blue”);
               break;
    case ‘g’ :
    case ‘G’: printf(“Green”);
               break;
    default: printf(“Black”);
}
```

Since there isn’t a break statement here, the control passes to the next statement (printf) without checking the next condition.

Evaluating expressions

```
void main () {
    int operand1, operand2;
    int result = 0;
    char operation ;
    /* Get the input values */
    printf ("Enter operand1 :");
    scanf("%d",&operand1) ;
    printf ("Enter operation :");
    scanf ("\n%c",&operation);
    printf ("Enter operand 2 :");
    scanf ("%d", &operand2);
    switch (operation)   {
    case '+' :
        result=operand1+operand2;
         break;
    case '-' :
        result=operand1-operand2;
        break;
    case '*' :
        result=operand1*operand2;
        break;
    case '/' :
        if (operand2 !=0)
            result=operand1/operand2;
        else
            printf("Divide by 0 error");
        break;
    default:
        printf("Invalid operation\n");
       return;
    }
    printf ("The answer is %d\n",result);
}
```

Character Strings

- Two ways of representing a sequence of characters in memory

5	**H**	**e**	**l**	**l**	**o**

  - The first location contains the number of characters in the string, followed by the actual characters

H	**e**	**l**	**l**	**o**	$\perp$

  - The characters follow one another, and is terminated by a special delimiter

String Representation in C

- In C, the second approach is used
  - The '\0' character is used as the string delimiter

- Example:

  "Hello" ➔

H	**e**	**l**	**l**	**o**	**'\0'**

- A null string "" occupies one byte in memory.
  - Only the '\0' character