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:

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

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

<table>
<thead>
<tr>
<th>Number System</th>
<th>Symbols</th>
<th>Base or Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decimal Numbers:</strong></td>
<td>10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$136.25 = 1 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$</td>
<td></td>
</tr>
<tr>
<td><strong>Binary Numbers:</strong></td>
<td>2 Symbols {0,1}, Base or Radix is 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$</td>
<td></td>
</tr>
<tr>
<td><strong>Octal Numbers:</strong></td>
<td>8 Symbols {0,1,2,3,4,5,6,7}, Base or Radix is 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$621.03 = 6 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 + 0 \times 8^{-1} + 3 \times 8^{-2}$</td>
<td></td>
</tr>
</tbody>
</table>
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:
  \[ B = b_{n-1} b_{n-2} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-m} \]

Corresponding value in decimal:

\[ D = \sum_{i=-m}^{n-1} b_i 2^i \]
Examples

101011 \rightarrow 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 \rightarrow 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}
= .3125

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

101.11 \rightarrow 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.

<table>
<thead>
<tr>
<th>Base</th>
<th>Numb</th>
<th>Rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\((89)_{10} = (1011001)_{2}\)
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.

<table>
<thead>
<tr>
<th>Base</th>
<th>Numb</th>
<th>Rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[(89)_{10} = (1011001)_{2}\]

<table>
<thead>
<tr>
<th>Base</th>
<th>Numb</th>
<th>Rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[(66)_{10} = (1000010)_{2}\]
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.

\[
\begin{array}{|c|c|c|}
\hline
\text{Base} & \text{Numb} & \text{Rem} \\
\hline
2 & 89 & 1 \\
2 & 44 & 1 \\
2 & 22 & 0 \\
2 & 11 & 0 \\
2 & 5 & 1 \\
2 & 2 & 1 \\
2 & 1 & 0 \\
0 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
2 & 66 & 0 \\
2 & 33 & 0 \\
2 & 16 & 1 \\
2 & 8 & 0 \\
2 & 4 & 0 \\
2 & 2 & 0 \\
2 & 1 & 0 \\
0 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
2 & 239 & 1 \\
2 & 119 & 1 \\
2 & 59 & 1 \\
2 & 29 & 1 \\
2 & 14 & 1 \\
2 & 7 & 0 \\
2 & 3 & 1 \\
2 & 1 & 1 \\
0 & 1 \\
\hline
\end{array}
\]

\[
(89)_{10} = (1011001)_{2}
\]

\[
(66)_{10} = (1000010)_{2}
\]

\[
(239)_{10} = (11101111)_{2}
\]
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

\[
\begin{align*}
.634 \times 2 &= 1.268 \\
.268 \times 2 &= 0.536 \\
.536 \times 2 &= 1.072 \\
.072 \times 2 &= 0.144 \\
.144 \times 2 &= 0.288 \\
\vdots & & \vdots \\
(.634)_{10} &= (.10100\ldots)_2
\end{align*}
\]
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**

- \(0.634 \times 2 = 1.268\)
- \(0.268 \times 2 = 0.536\)
- \(0.536 \times 2 = 1.072\)
- \(0.072 \times 2 = 0.144\)
- \(0.144 \times 2 = 0.288\)
- \(\vdots\)
- \((0.634)_{10} = (0.10100\ldots)_{2}\)

**Example: 0.0625**

- \(0.0625 \times 2 = 0.125\)
- \(0.1250 \times 2 = 0.250\)
- \(0.2500 \times 2 = 0.500\)
- \(0.5000 \times 2 = 1.000\)
- \((0.0625)_{10} = (0.0001)_{2}\)
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**

\[
\begin{align*}
.634 \times 2 &= 1.268 \\
.268 \times 2 &= 0.536 \\
.536 \times 2 &= 1.072 \\
.072 \times 2 &= 0.144 \\
.144 \times 2 &= 0.288 \\
\vdots
\end{align*}
\]

\( (.634)_{10} = (.10100......)_{2} \)

**Example: 0.0625**

\[
\begin{align*}
.0625 \times 2 &= 0.125 \\
.1250 \times 2 &= 0.250 \\
.2500 \times 2 &= 0.500 \\
.5000 \times 2 &= 1.000 \\
(.0625)_{10} &= (.0001)_{2}
\end{align*}
\]

\[
\begin{align*}
(37)_{10} &= (100101)_{2} \\
(.0625)_{10} &= (.0001)_{2} \\
(37.0625)_{10} &= (100101.0001)_{2}
\end{align*}
\]
Hexadecimal Number System

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary 4-Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>
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. \((1011 \ 0100 \ 0011)_2 = (B43)_{16}\)
2. \((10 \ 1010 \ 0001)_2 = (2A1)_{16}\)
3. \((.1000 \ 010)_2 = (.84)_{16}\)
4. \((101 \ . \ 0101 \ 111)_2 = (5.5E)_{16}\)
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
  - \(n=8\) \(\Rightarrow\) 0 to \(2^8-1\) (255)
  - \(n=16\) \(\Rightarrow\) 0 to \(2^{16}-1\) (65535)
  - \(n=32\) \(\Rightarrow\) 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 $\rightarrow$ 0 000....0
  -0 $\rightarrow$ 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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
</tr>
<tr>
<td>1111</td>
<td>-0</td>
</tr>
</tbody>
</table>

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.
    - $+0 \rightarrow 0 000\ldots0$
    - $-0 \rightarrow 1 111\ldots1$

- 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: \( n=4 \)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Value</th>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
<td>1111</td>
<td>-1</td>
</tr>
</tbody>
</table>

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

\[ +4 = 0100 \]
\[ -4 = 2\text{'s complement of } 0100 = 1011+1 = 1100 \]

Rule: Value = \(- \text{msb} \times 2^{(n-1)} + \text{[unsigned value of rest]}\)

Example: 0110 = 0 + 6 = 6  
1110 = -8 + 6 = -2
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
In C

- **short int**
  - 16 bits \( \Rightarrow + (2^{15}-1) \) to \(-2^{15}\)

- **int or long int**
  - 32 bits \( \Rightarrow + (2^{31}-1) \) to \(-2^{31}\)

- **long long int**
  - 64 bits \( \Rightarrow + (2^{63}-1) \) to \(-2^{63}\)
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 $A - B$?

1. Compute the 1’s complement of $B$ (say, $B_1$).
2. Compute $R = A + B_1$
3. 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
4. Else
   - The result is negative, and is in 1’s complement form
Example 1 :: 6 – 2

1’s complement of 2 = 1101

\[
\begin{array}{c}
6 & : & 0110 \\
-2 & : & 1101 \\
\hline
1 & 0011 \\
\end{array}
\]

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

The result is positive

End-around carry
Example 2 :: 3 – 5

1’s complement of 5 = 1010

\[
\begin{array}{c|c}
3 & 0011 \\
-5 & 1010 \\
\hline
1101 & R
\end{array}
\]

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
-2 :: 1110

A
B₂
R

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

Ignore carry

+4
Example 2 :: 3 – 5

2’s complement of 5 = 1010 + 1 = 1011

3 :: 0011
-5 :: 1011
1110

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

\[
\begin{array}{r}
00000111 \\
+ \quad 11110111 \\
\hline
11111110
\end{array}
\]

11111110 (with a carry of 0 which is ignored)
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.

\[
\begin{array}{c}
(64) & 01000000 \\
(4) & 00000100 \\
\hline
(68) & 01000100 \\
\end{array}
\]

carry (out)(in)
\[
0 \quad 0
\]
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.

<table>
<thead>
<tr>
<th>(64) 01000000</th>
<th>(64) 01000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 4) 00000100</td>
<td>(96) 01100000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(68) 01000100</td>
<td>(-96) 10100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>carry (out)(in)</th>
<th>carry out in</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 1</td>
</tr>
</tbody>
</table>

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 $<M,E>$:
  \[ F = M \times B^E \]
  - $B \rightarrow$ exponent base (usually 2)
  - $M \rightarrow$ mantissa
  - $E \rightarrow$ exponent

- $M$ is usually represented in 2’s complement form, with an implied binary point before it

- For example,
  - In decimal, $0.235 \times 10^6$
  - In binary, $0.101011 \times 2^{0110}$
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)

<table>
<thead>
<tr>
<th>S (31)</th>
<th>E (Exponent) (30 ... 23)</th>
<th>M (Mantissa) (22 ... 0)</th>
</tr>
</thead>
</table>

- **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

<table>
<thead>
<tr>
<th>S (31)</th>
<th>E (Exponent)</th>
<th>M (Mantissa)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(30 … 23)</td>
<td>(22 … 0)</td>
</tr>
</tbody>
</table>

| 1 | 10001100 | 1101100000000000000000000 |

Value of a Normal Number:

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

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

\[ = -1.1101100 \times 2^{1101} = -111011000000000 \]

\[ = -15104.0 \text{ (in decimal)} \]
Representing 0.3

<table>
<thead>
<tr>
<th>S (31)</th>
<th>E (Exponent) (30 … 23)</th>
<th>M (Mantissa) (22 … 0)</th>
</tr>
</thead>
</table>

0.3 (decimal)

= 0.0100100100100100100100100…

= 1.00100100100100100100100… × 2⁻²

= 1.00100100100100100100100… × 2¹²⁵⁻¹²⁷

= (-1)S × (1.0 + 0.M) × 2^(E - 127)

| 0 | 01111101 | 00100100100100100100100 |

What are the largest and smallest numbers that can be represented in this scheme?
### Representing 0

<table>
<thead>
<tr>
<th>S (31)</th>
<th>E (Exponent)</th>
<th>M (Mantissa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>00000000000000000000000</td>
</tr>
</tbody>
</table>

### Representing Inf (\(\infty\))

<table>
<thead>
<tr>
<th>S (31)</th>
<th>E (Exponent)</th>
<th>M (Mantissa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111111</td>
<td>00000000000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
<td>00000000000000000000000</td>
</tr>
</tbody>
</table>

### Representing NaN (Not a Number)

<table>
<thead>
<tr>
<th>S (31)</th>
<th>E (Exponent)</th>
<th>M (Mantissa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111111</td>
<td>Non zero</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
<td>Non zero</td>
</tr>
</tbody>
</table>
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 (size qualifier)
  - long (size qualifier)
  - signed (sign qualifier)
  - unsigned (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)

<table>
<thead>
<tr>
<th>Integer data type</th>
<th>Bit size</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>(-2^7=-128)</td>
<td>(2^7-1=127)</td>
</tr>
<tr>
<td>short int</td>
<td>16</td>
<td>(-2^{15}=-32768)</td>
<td>(2^{15}-1=32767)</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>(-2^{31}=-2147483648)</td>
<td>(2^{31}-1=2147483647)</td>
</tr>
<tr>
<td>long int</td>
<td>32</td>
<td>(-2^{31}=-2147483648)</td>
<td>(2^{31}-1=2147483647)</td>
</tr>
<tr>
<td>long long int</td>
<td>64</td>
<td>(-2^{63}=-9223372036854775808)</td>
<td>(2^{63}-1=9223372036854775807)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>(2^8-1=255)</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>16</td>
<td>0</td>
<td>(2^{16}-1=65535)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>0</td>
<td>(2^{32}-1=4294967295)</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>32</td>
<td>0</td>
<td>(2^{32}-1=4294967295)</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>64</td>
<td>0</td>
<td>(2^{64}-1=18446744073709551615)</td>
</tr>
</tbody>
</table>
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**

```c
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
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
<th>Character</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td>00100000</td>
<td>SPACE</td>
<td>80</td>
<td>50</td>
<td>01010000</td>
<td>P</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>00100001</td>
<td>!</td>
<td>81</td>
<td>51</td>
<td>01010001</td>
<td>Q</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>00100010</td>
<td>&quot;</td>
<td>82</td>
<td>52</td>
<td>01010010</td>
<td>R</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>00100011</td>
<td>#</td>
<td>83</td>
<td>53</td>
<td>01010011</td>
<td>S</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>00100100</td>
<td>$</td>
<td>84</td>
<td>54</td>
<td>01010100</td>
<td>T</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>00100101</td>
<td>%</td>
<td>85</td>
<td>55</td>
<td>01010101</td>
<td>U</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>00100110</td>
<td>&amp;</td>
<td>86</td>
<td>56</td>
<td>01010110</td>
<td>V</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>00100111</td>
<td>'</td>
<td>87</td>
<td>57</td>
<td>01010111</td>
<td>W</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>00101000</td>
<td>(</td>
<td>88</td>
<td>58</td>
<td>01011000</td>
<td>X</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>00101001</td>
<td>)</td>
<td>89</td>
<td>59</td>
<td>01011001</td>
<td>Y</td>
</tr>
<tr>
<td>42</td>
<td>2a</td>
<td>00101010</td>
<td>*</td>
<td>90</td>
<td>5a</td>
<td>01011010</td>
<td>Z</td>
</tr>
<tr>
<td>43</td>
<td>2b</td>
<td>00101011</td>
<td>+</td>
<td>91</td>
<td>5b</td>
<td>01011111</td>
<td>[</td>
</tr>
<tr>
<td>44</td>
<td>2c</td>
<td>00101100</td>
<td>,</td>
<td>92</td>
<td>5c</td>
<td>01011100</td>
<td>\</td>
</tr>
<tr>
<td>45</td>
<td>2d</td>
<td>00101101</td>
<td>-</td>
<td>93</td>
<td>5d</td>
<td>01011110</td>
<td>]</td>
</tr>
<tr>
<td>46</td>
<td>2e</td>
<td>00101110</td>
<td>.</td>
<td>94</td>
<td>5e</td>
<td>01011111</td>
<td>^</td>
</tr>
<tr>
<td>47</td>
<td>2f</td>
<td>00101111</td>
<td>/</td>
<td>95</td>
<td>5f</td>
<td>01011111</td>
<td>_</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>00110000</td>
<td>0</td>
<td>96</td>
<td>60</td>
<td>01100000</td>
<td>\</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>00110001</td>
<td>1</td>
<td>97</td>
<td>61</td>
<td>01100001</td>
<td>a</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>00110010</td>
<td>2</td>
<td>98</td>
<td>62</td>
<td>01100010</td>
<td>b</td>
</tr>
<tr>
<td>Hex</td>
<td>Dec</td>
<td>ASCII</td>
<td>Value</td>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>c</td>
<td>0110011</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>d</td>
<td>0110010</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>e</td>
<td>0110011</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>f</td>
<td>0110010</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>g</td>
<td>0110011</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>h</td>
<td>0110100</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>i</td>
<td>0110101</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>3a</td>
<td>j</td>
<td>0110110</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>3b</td>
<td>k</td>
<td>0110111</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>3c</td>
<td>l</td>
<td>0110110</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>3d</td>
<td>m</td>
<td>0110111</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>3e</td>
<td>n</td>
<td>0110110</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>3f</td>
<td>o</td>
<td>0110111</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>p</td>
<td>0111000</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>q</td>
<td>0111001</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>r</td>
<td>0111010</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>s</td>
<td>0111011</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>t</td>
<td>0111100</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>u</td>
<td>0111101</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>v</td>
<td>0111110</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>01000111</td>
<td>G</td>
<td>119</td>
<td>77</td>
<td>01110111</td>
<td>w</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----------</td>
<td>----</td>
<td>-----</td>
<td>----</td>
<td>----------</td>
<td>----</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>01001000</td>
<td>H</td>
<td>120</td>
<td>78</td>
<td>01111000</td>
<td>x</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>01001001</td>
<td>I</td>
<td>121</td>
<td>79</td>
<td>01111001</td>
<td>y</td>
</tr>
<tr>
<td>74</td>
<td>4a</td>
<td>01001010</td>
<td>J</td>
<td>122</td>
<td>7a</td>
<td>01111010</td>
<td>z</td>
</tr>
<tr>
<td>75</td>
<td>4b</td>
<td>01001011</td>
<td>K</td>
<td>123</td>
<td>7b</td>
<td>01111011</td>
<td>{</td>
</tr>
<tr>
<td>76</td>
<td>4c</td>
<td>01001100</td>
<td>L</td>
<td>124</td>
<td>7c</td>
<td>01111100</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>4d</td>
<td>01001101</td>
<td>M</td>
<td>125</td>
<td>7d</td>
<td>01111101</td>
<td>}</td>
</tr>
<tr>
<td>78</td>
<td>4e</td>
<td>01001110</td>
<td>N</td>
<td>126</td>
<td>7e</td>
<td>01111110</td>
<td>~</td>
</tr>
<tr>
<td>79</td>
<td>4f</td>
<td>01001111</td>
<td>O</td>
<td>127</td>
<td>7f</td>
<td>01111111</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
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

```c
int a;
a = 'c' * 3 + 5;
printf("%d", a);
```

Will print 296 (97*3 + 5)
(ASCII code of ‘c’ = 97)

```c
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

```c
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

```c
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.
void main () {
    int operand1, operand2;
    int result = 0;
    char operation ;

    /* Get the input values */
    printf ("Enter operand1 :");
    scanf("%d", &operand1) ;
    printf ("Enter operation :");
    scanf ("%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 ");
        return;
    }

    printf("The answer is %d ", result);
}

Evaluating expressions
Character Strings

- Two ways of representing a sequence of characters in memory

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

- 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” ➔ Hello

- A null string “” occupies one byte in memory.
  - Only the ‘\0’ character