Number Systems

decimal number:
7397 = 7\times10^3 + 3\times10^2 + 9\times10^1 + 7\times10^0

\[ a_5a_4a_3a_2a_1a_0 \]

By repeated division for integers and repeated multiplication for fractions.

Ex1. Convert (153.513)_{10} to octal number.

Integer:

<table>
<thead>
<tr>
<th>153</th>
<th>(231)_{8}</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Fractional:

| 0.513 \times 8 = 4.104 |
| 0.104 \times 8 = 0.832 |
| 0.832 \times 8 = 6.656 |

\[ (153.513)_{10} = (231.4065\ldots)_{8} \]

Ex2. Convert (12A)_{16} to octal.

1. Convert (12A)_{16} to (298)_{10}
2. Convert (298)_{10} to octal as in Ex1.

Conversions between binary, octal, and hexadecimal can be done in a simpler way:

Ex1. Convert (10111010011)_{2} to octal and hexadecimal.

\[ \begin{array}{c|c|c|c|c|c}
\text{Binary} & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
\text{Octal} & 2 & 7 & 2 & 3 \\
\text{Hexadecimal} & (2723)_{8} & (5D3)_{16} \\
\end{array} \]

Ex2. Convert (12A7F)_{16} to binary and octal.

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
\text{Binary} & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\text{Octal} & 2 & 6 & 7 & 7 \\
\end{array} \]

Complement: for simplifying subtraction.

Two types of complements for each base-r system:

- \((r-1)’s\) complement
- \(r’\)s complement

The \((r-1)’s\) complement of an n-digit number \(N\) is \((r^n - 1) - N\)

Ex. In decimal system, the \(9’\)s complement of 546700 is

\(999999 - 546700 = 453299\)

The 9’s complement of 012398 is 999999 – 012398 = 987601

The 1’s complement of 0111000 is

\((2^6 - 1) – 0111000 = 01011111 = 01011100 = 10100011\)

The 1’s complement of 0101101 is 1010010.
The $r$'s complement of an $n$-digital number $N$:
\[ r^r - N \quad N \neq 0 \]
\[ 0 \quad N = 0 \]

Ex. The 10's complement of 012398 is 987602.
The 10's complement of 2467000 is 7533000.
The 2's complement of 110100 is 0010100.
The 2's complement of 0110111 is 1001001.

Comments:
1. To compute the complement of a number having radix point, first, remove the radix point, compute the complement of the new number, and restore the radix point.

Ex. The 1's complement of 01101.101 is 10010.010.
The 2's complement of 01101.101 is 10010.011.

Signed Binary Numbers

Addition:
\[ + \quad 9 \quad 00001001 \]
\[ + \quad 13 \quad + \quad 00001101 \]
\[ + \quad 22 \quad + \quad 00010110 \]

Subtraction:
\[ - \quad 9 \quad 10001001 \]
\[ - \quad 13 \quad - \quad 1011 \]
\[ - \quad 4 \quad - \quad 10110 \]

How do we represent signed numbers in digital systems?
representation of $+9$ with 8 bits: 00001001
representation of $-9$:
signed magnitude: 10001001
signed -1's-complement: 11110110
signed -2's-complement: 11110111

Numbers that can be represented in $n$ bits:
signed magnitude: $(2^n - 1 - (2^n - 1))$
signed-1's-complement: $(2^n - 1 - (2^n - 1))$
signed-2's-complement: $(2^n - 1 - (2^n - 1))$

Addition and Subtraction on signed-magnitude numbers:
\[ + \quad 9 \quad 00001001 \]
\[ + \quad 13 \quad + \quad 00001101 \]
\[ + \quad 4 \quad + \quad 00000100 \]
\[ - \quad 9 \quad 10001001 \]
\[ - \quad 13 \quad - \quad 1011 \]
\[ - \quad 4 \quad - \quad 10110 \]

Summary:
To perform arithmetic operations on signed magnitude numbers, we need to compare the signs and the magnitudes of the two numbers, and then perform either addition or subtraction.

Addition and subtraction on signed-2's-complement numbers:
Addition:
\[ + \quad 9 \quad 00001001 \]
\[ + \quad 13 \quad + \quad 11110011 \]
\[ + \quad 4 \quad + \quad 11111100 \]
\[ + \quad 9 \quad 10001001 \]
\[ + \quad 13 \quad + \quad 1011 \]
\[ + \quad 22 \quad + \quad 10010110 \]

Discussion:
\[ (\neg N) \times (M) \Rightarrow (2^n - N) \times M \]
\[ = 2^n \times (N - M) \]

Case $N > M$:
there will be no carry out, and the result is the 2's complement of $(N - M)$.

Case $N < M$:
there will be a carry out, and the result after discarding the carry out is $(M - N)$.
overflow:
+ 64 01000000 −64 11000000
+ 96 + 01100000 −96 +01010000
\( \pm \text{positive!} \)
\( \pm \text{negative!} \)

using 8 bits, we can represent only \(-128 \sim +127\).

When add two \(n\)-bits (including the sign bit) numbers \(N\) and \(M\), overflow occurs when:
1. \(N, M \geq 0, N+M > 2^n-1\)
2. \(N, M < 0, N+M < -2^n\)

Note that overflow cannot occur if one of \(N\) and \(M\) is positive and the other is negative.

Addition and Subtraction on Signed-1's-complement numbers:
Assume that \(M, N \geq 0\), and they have \(n\) bits (including sign)

Case \(M + N\): normal binary addition.
+ 9 00001001 
+ 13 00001101 
+ 22 00010110

Sub-case: \(N - M \geq 0\):
\((1\text{'s complement of } N) + M = \left(2^n-1 - N\right) + M \rightarrow \text{there will be no carry.}\)

Sub-case: \(N - M < 0\):
\((2^n-1 - N) + M = 2^n + (M - N) - 1 \rightarrow \text{there will be a carry. To obtain } (M-N), \text{we discard the carry and add } 1 \text{ to the result.}\)

Ex. \(-9\) 11110110 
+ 13 00001101 
+ 4 00000100 
\( \pm \text{end-around carry} \)

\( \frac{11101001}{11101001} \) \( \pm \text{1's complement of } 00010110 \) \( \text{22} \)

Comparison of the three signed numbers systems:
- signed-magnitude: useful in ordinary arithmetic, awkward in computer arithmetic
- signed-1’s-complement: used in old computers, but is now seldom used.
- signed-2’s-complement: used in most computers.
Binary Codes

Binary codes can be established for any set of discrete elements. Using \( n \) bits, we can represent at most \( 2^n \) distinct elements. So, to represent \( m \) distinct objects, we need at least \( \lceil \log_2 m \rceil \) bits. For example, we need \( \lceil \log_2 10 \rceil = 4 \) bits to represent \{ 0,1,…,9 \}.

Alphanumeric Codes

--- ASCII (American Standard Code for Information Interchange)
- originally use 7 bits to code 128 characters
- since most digital systems handle 8-bit (byte) more efficiently, an 8-bit version ASCII has also been developed.

--- EBCDIC (Extended Binary-Coded Decimal Interchange Code)
- used in most IBM systems
- use 8 bits to code the same 128 character symbols as ASCII

Gray Code

each binary code differs from either its successor or predecessor by only one bit.

Gray Code is often used in situations where other binary code might produce erroneous or ambiguous results during transitions from one code to another.

If we use BCD code to transmit 0111 (7) and then 1000 (8), then, an erroneous intermediate code 1001 may be generated if the rightmost bit takes more time to change. The Gray code eliminates this problem since only one bit changes between two consecutive numbers.

Generation of Gray Codes:
an \( n \)-bits code is generated by reflecting the \((n-1)\)-bit code.

Some possible binary codes are illustrated below.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>BCD ((8421))</th>
<th>Excess-3 ((5043210))</th>
<th>84-2-1 ((5043210))</th>
<th>Bi-quinary ((5043210))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>00000001</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0011</td>
<td>0001</td>
<td>00000100</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0101</td>
<td>0010</td>
<td>00010010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0110</td>
<td>0011</td>
<td>00010100</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0111</td>
<td>0100</td>
<td>00011000</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1000</td>
<td>0101</td>
<td>00011001</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1001</td>
<td>0110</td>
<td>01000001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1010</td>
<td>0111</td>
<td>01000100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1011</td>
<td>1000</td>
<td>10000100</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1100</td>
<td>1001</td>
<td>10001000</td>
</tr>
</tbody>
</table>

4-bit Gray Code

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Gray Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
</tr>
<tr>
<td>4</td>
<td>0111</td>
</tr>
<tr>
<td>5</td>
<td>0110</td>
</tr>
<tr>
<td>6</td>
<td>0101</td>
</tr>
<tr>
<td>7</td>
<td>0100</td>
</tr>
<tr>
<td>8</td>
<td>1100</td>
</tr>
<tr>
<td>9</td>
<td>1111</td>
</tr>
<tr>
<td>10</td>
<td>1110</td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
</tr>
<tr>
<td>12</td>
<td>1011</td>
</tr>
<tr>
<td>13</td>
<td>1001</td>
</tr>
<tr>
<td>14</td>
<td>1000</td>
</tr>
<tr>
<td>15</td>
<td>1010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generation of Gray Codes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>an ( n )-bits code is generated by reflecting the ((n-1))-bit code.</td>
</tr>
<tr>
<td>0 0 00</td>
</tr>
<tr>
<td>1 1 01</td>
</tr>
<tr>
<td>1 0 11</td>
</tr>
<tr>
<td>0 1 11</td>
</tr>
<tr>
<td>0 1 10</td>
</tr>
<tr>
<td>1 0 10</td>
</tr>
<tr>
<td>0 0 10</td>
</tr>
<tr>
<td>0 1 00</td>
</tr>
<tr>
<td>1 0 00</td>
</tr>
<tr>
<td>0 0 00</td>
</tr>
<tr>
<td>0 1 01</td>
</tr>
<tr>
<td>1 0 11</td>
</tr>
<tr>
<td>1 1 10</td>
</tr>
<tr>
<td>1 1 01</td>
</tr>
<tr>
<td>1 0 10</td>
</tr>
<tr>
<td>0 0 10</td>
</tr>
<tr>
<td>0 1 10</td>
</tr>
<tr>
<td>0 1 01</td>
</tr>
<tr>
<td>0 0 11</td>
</tr>
<tr>
<td>0 0 01</td>
</tr>
<tr>
<td>0 0 00</td>
</tr>
</tbody>
</table>

A/D Converter

Receiver

Transmitter

If we use BCD code to transmit 0111 (7) and then 1000 (8), then, an erroneous intermediate code 1001 may be generated if the rightmost bit takes more time to change. The Gray code eliminates this problem since only one bit changes between two consecutive numbers.
Error-Detecting Code

Binary code that can detect errors during data transmission. The most common ways to achieve error detection is by means of a parity bit. A parity bit is an extra bit included in a binary code to make the total number of 1’s transmitted either odd (odd parity) or even (even parity).

<table>
<thead>
<tr>
<th>message</th>
<th>parity bit</th>
<th>message</th>
<th>parity bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>1</td>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>1</td>
<td>1010</td>
<td>0</td>
</tr>
</tbody>
</table>

Odd parity

```
message   parity bit
0010       0
0110       1
1110       0
1010       1
```

Even parity

```
message   parity bit
0010       1
0110       0
1110       1
1010       0
```

**Transmitter**

```
source
Parity generator
```

**Parity detector**

```
Parity detector
destination
```

Digital Circuits

<table>
<thead>
<tr>
<th>Gate</th>
<th>Name</th>
<th>Algebraic Function</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>AND</td>
<td>F = xy</td>
<td><img src="image" alt="Table" /></td>
</tr>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>OR</td>
<td>F = x + y</td>
<td><img src="image" alt="Table" /></td>
</tr>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>Inverter</td>
<td>F = x'</td>
<td><img src="image" alt="Table" /></td>
</tr>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>Buffer</td>
<td>F = x</td>
<td><img src="image" alt="Table" /></td>
</tr>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>NAND</td>
<td>F = (xy)'</td>
<td><img src="image" alt="Table" /></td>
</tr>
</tbody>
</table>

Extension to multiple inputs

```
x \oplus y \oplus z = (x \oplus y) \oplus z
```

In real implementation, the three-input Exclusive-OR is usually implemented by 2-input Exclusive-OR:

```
x \oplus y = x'y + xy'
```

Even a two input Exclusive-OR is usually constructed with other types of gates.
This is how a gate adds a binary 1 and 0 in the “ones” place. If you feed a 1 and a 0 to the gate, it puts out a 1, which is the correct result of adding a binary 1 and 0.

Addition with a carryover is a little more difficult, for instance, adding 1 plus 1. If you feed the gate a 1 and 1, it will put a 0 in the “ones” place and put a carryover of 1 in the “twos” place. This produces the correct result for adding 1 and 1 in binary.

Various Implementations of a half-adder

Binary Adder and Subtractor
\[ c = f(x,y,z) = \sum(3,5,6,7) \]
\[ s = \sum(1,2,4,7) \]
\[ c = xy + yz + xz \]
\[ s = x'y' + x'yz' + xy'z' + xyz \]

4-bit adder-subtractor