Data Representation

Michael Nowak

Texas A&M University

Notes

Overview

Notes

Overview

- Binary number system
  - Motivation for study
  - Binary numbers and modern computers
  - What is a positional number system?
  - The binary number system
  - Binary to decimal conversion
  - Decimal to binary conversion
  - Binary addition
- Data representation in the computer
  - Integral numbers
  - Floating-point numbers
  - Character data
- References
- Appendix
  - Base-b to decimal conversion
  - Decimal to base-b conversion
  - Binary multiplication
  - Binary subtraction
  - Binary division
  - Complements

Notes

Overview

Notes

Motivation for study

- If you've programmed before, you may have had some surprises in the results of your computations; for example,
  - consider that `1500000000+1500000000`
  - evaluates to `-1294967296` on my machine
  - a result that mathematicians would consider wrong
- As an additional example,
  - consider that `.15+.15`
  - evaluates to `0.29999999999999999` on my machine
  - a result that mathematicians would also consider wrong

Notes

Motivation for study

- The results that we've observed are a reflection of the way that numbers are stored in computers
- This can impact how your programs work
- So, it is worth taking a bit of time to discuss the representation of data in the computer

Notes

Overview

Notes

Binary numbers and modern computers

- All information stored on modern computers is represented with numbers
- Binary (base-2) numbers are used to represent these numbers in modern computers
- Electronics in the computer alternate between two states: on and off
  - Sequences of 'these' represent numbers, which represent data

Notes

Overview

- Binary number system
  - Motivation for study
  - Binary numbers and modern computers
  - What is a positional number system?
  - The binary number system
  - Binary to decimal conversion
  - Decimal to binary conversion
  - Binary addition
- Data representation in the computer
  - Integral numbers
  - Floating-point numbers
  - Character data
- References
- Appendix
  - Base-b to decimal conversion
  - Decimal to base-b conversion
  - Binary multiplication
  - Binary subtraction
  - Binary division
  - Complements

Notes

What is a positional number system?

- Any positive integer $b > 1$ can be chosen as a base for a positional number system
  - decimal system ($b = 10$)
  - binary system ($b = 2$)
- Such a system uses b symbols for the integers (know as the digits)
$$0, 1, ..., b - 1$$
- Any integer N is represented by a sequence of base-b digits
$$N = a_n a_{n-1} ... a_1 a_0$$
- Such that b^k is the place value of a_k and
$$N = a_n \times b^n + a_{n-1} \times b^{n-1} + ... + a_1 \times b^1 + a_0 \times b^0$$

Notes

Overview

Notes

The binary number system

- Positional numeration system using base-2
- The two digits used are zero (0) and one (1); are called *bits*
- Binary numbers are a sequence of bits; can have an embedded binary point
- Binary integers are binary numbers without a fractional part
- The integral parts of a binary number are represented as a sequence of bits
$$N_I = a_n a_{n-1} ... a_1 a_0$$
- Such that in N_I 2^k is the place value of a_k and
$$N_I = a_n \times 2^n + a_{n-1} \times 2^{n-1} + ... + a_1 \times 2^1 + a_0 \times 2^0$$
- A similar notion holds for the fractional part; for instance, consider 1001.001_2

$$1001.001_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

Notes

Overview

Notes

Binary to decimal conversion

- A binary number N_2 can be converted to base-10 by writing N_2 in expanded notation and calculating by decimal arithmetic
- We can also use the following algorithm (this should look relatively familiar)

Integral part
- i Double the leftmost digit
- ii Add the result to next digit to the right
- iii Double that sum
- iv Add the result to the next digit
- v Repeat until the last integral digit is added; the final sum is the decimal equivalent

Fractional part
- i Multiply the rightmost fractional digit by $\frac{1}{2}$
- ii Add the next digit to the left to that product
- iii Multiply that sum by $\frac{1}{2}$
- iv Add the next digit to that product
- v Repeat until the leftmost digit (prior to decimal point) is added
- vi Then multiply that sum by $\frac{1}{2}$

Notes

Overview

Notes

Decimal to binary conversion

Notes

- Given decimal number N, with integral part N_I and fractional part N_F, N can be converted to base-2 using the following algorithm:

  Integral part
  - i Subtract the largest possible power of base-2 from N_I
  - ii Subtract the largest possible power of base-2 from the result
  - iii Continue this process until a difference of zero is obtained
  - iv Place a bit value of 1 in the place values of those powers subtracted and a bit value of 0 elsewhere

  Fractional part
  - i Multiply N_F and the *fractional* portion of each succeeding product by 2 until a zero (or repeating) fractional part is observed
  - ii The resultant sequence of integral parts in-order gives the corresponding representation of N_F in base-2

Overview

Notes

Binary addition

Notes

- When adding two binary numbers, you need to know the following facts:
  - $0 + 0 = 0$
  - $1 + 0 = 1$
  - $0 + 1 = 1$
  - $1 + 1 = 0$, with carry 1
  - $1 + 1 + 1 = 1$, with carry 1
- Let's consider the following example,

$$\begin{array}{r} 1111 \\ +\,110 \\ \hline \end{array}$$

Binary addition example

Step 1.) $1 + 0 = 1$

$$\begin{array}{r} 1111 \\ +\,0110 \\ \hline 1 \end{array}$$

Step 2.) 1+1=0, with carry 1

$$\begin{array}{r} {\scriptstyle 1} \\ 1111 \\ +\,0110 \\ \hline 01 \end{array}$$

Step 3.) 1+1+1=1, with carry 1

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 1111 \\ +\,0110 \\ \hline 101 \end{array}$$

Step 4.) 1+1=0, with carry 1

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 1111 \\ +\,0110 \\ \hline 10101 \end{array}$$

Notes

Overview

Notes

Data representation in the computer

- Most modern computers deal with memory as chunks of bits of sizes that are powers of 2
- A byte of memory is usually defined as 8-bits
- Data is represented by a fixed number bytes, with the amount of bytes dependent on the kind of data

Notes

Overview

Notes

Integral numbers

- We only have a limited number of bytes to represent these data in the computer, with the amount dependent on the kind of data
- For the purposes of our discussion, let's say that we only have a half-byte (4-bits) to represent the integral numbers
- The positional system makes it easy to represent positive numbers using straight binary encoding:
  - $\boxed{1}\boxed{0}\boxed{1}\boxed{1}$
  - Which could represent

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$$

- This is great for positive numbers, but *how can we represent negative numbers along side positive numbers?*

Notes

Sign magnitude notation

- One idea is to reserve the most significant bit (that furtherest to the left) as a sign bit.
  - The idea being that if this bit is set (1), we have a negative number
  - Otherwise, we have a positive number
  - We could then encode -3 as $\boxed{1}\boxed{0}\boxed{1}\boxed{1}$ would be represent

$$-1 \times 1 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -3$$

  - Positive 3 would then be encoded as $\boxed{0}\boxed{0}\boxed{1}\boxed{1}$ would be represent

$$-1 \times 0 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3$$

Notes

Sign magnitude notation

- This method proves problematic with basic arithmetic:

$$\begin{array}{r} 0011 \\ +\,1011 \\ \hline 1110 \end{array}$$

- That is,

$$3 + -3 = -6$$

- A result that mathematicians would consider wrong

Notes

Sign magnitude notation

- Another 'problem' with this technique is that we have two values encoding zero:
  - $\boxed{1}\boxed{0}\boxed{0}\boxed{0}$ would represent

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

  - $\boxed{0}\boxed{0}\boxed{0}\boxed{0}$ would represent

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

Notes

A second idea: additive inverses

- Based on the limitations of sign magnitude notation, we'd really like an encoding scheme such that

$$x + (-x) = 0 \mid x \in \mathbb{Z}$$

  - That is, store negative numbers as the additive inverses of their positive counterparts
- How we encode a negative number is answered by figuring out what value we would add to its positive counterpart to get zero:

$$\begin{array}{r} 0011 \\ +\,???? \\ \hline 0000 \end{array}$$

  - There is no value that we can use in place of the question marks to make this work...

Notes

A second idea: additive inverses

- Recall, that our values can be represented by a limited number of bytes depending on the kind of value
- Anything beyond that is lost
- Therefore, we do not need the total to be zero, but need four digits of zero
- What we're really looking for is:

$$\begin{array}{r} 0011 \\ +\,????\\ \hline 10000 \end{array}$$

  - This problem is solvable because the most significant bit will be lost...

Notes

A second idea: additive inverses

- We would therefore encode -3 as 1101 because

$$\begin{array}{r} 0011 \\ +\,1101 \\ \hline 10000 \end{array}$$

- This way of encoding signed numbers is known as **two's complement**
  - To get the two's compliment of any positive integral binary number, flip all the bits and add one
  - Note: *the most significant bit here is not exactly a sign bit, though it can be used to determine whether the integral binary value encoded is positive or negative*

Notes

Two's complement

- The common way to represent integral numbers in memory is using binary form for positive numbers and by its twos complement when negative
- What range of values can we represent using straight binary coding and twos complement using half-byte of memory?
  - Largest (positive) integer:
    $\boxed{0}\boxed{1}\boxed{1}\boxed{1} = 2^{4-1} - 1 = 7$
  - Smallest (most negative) integer:
    $\boxed{1}\boxed{0}\boxed{0}\boxed{0} = -2^{4-1} = -8$

Notes

Overview

Notes

Normalized exponential form

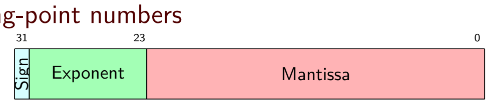
- Every decimal number N_D can be written as a power of ten in *exponential form*
- This form is not unique

$$N_D = 111 = 0.111 \times 10^3 = 1.11 \times 10^2 = 11.1 \times 10^1 = 11100 \times 10^{-2}$$

- However, we can write any non-zero N_D uniquely as a number M multiplied by a power of ten e by ensuring that the decimal point appears directly in front of the first non-zero digit in N_D
  - This is known as *normalized exponential form*
  - The number M is called the *mantissa* of N_D
  - The exponent e is the *exponent* of N_D
  - Note the difference between the definition of *normalized exponential form* and *scientific notation*
- Binary numbers, like decimal numbers, can be written in *normalized exponential form* using powers of 2 opposed to 10

Notes

Floating-point numbers



- Using normalized exponential form of the value to be encoded, we can store a floating-point number at a memory location partitioned into three fields
  - The first bit denotes the sign s of the number
  - The second field, the exponent e of the number
    - Few computers store e in binary form when positive or zero; twos complement when negative
    - Others use a bias, such that the exponent is calculated as $2^{(e-\text{bias})}$ where the *bias*is $2^{n-1} - 1$ such that n is the number of bits of the exponent field
  - The third field is the mantissa m of the number, and contains the fractional part of the number in normalized binary form
- The floating-point value encoded can thus calculated as

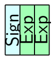
$$(-1)^s * 2^{(e-\text{bias})} * (1 + m)$$

Notes

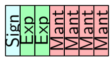
Floating-point numbers

- Let's look at a "toy" floating-point number system, an exponent range of -1 to 1, and four bits for the mantissa
- The left-most bit of this "toy" floating-point system will be the sign bit:

- We then need to be able to represent an exponent range of -1 to 1, a range of three values, meaning we need at two bits for the exponent:



- We then need to provide four bits for the mantissa:



Notes

Floating-point numbers

- Using our "toy" floating-point number system, we could represent 3.5 (with the exponent encoded with a bias and mantissa assuming a leading 1) as follows:

0	1	0	1	1	0	0

- We can then calculate the encoded floating-point value as

$$(-1)^s * 2^{(e-\text{bias})} * (1+m)$$
$$(-1)^0 * 2^{(2-((2^{2-1})-1))} * (1+0.75)$$
$$(-1)^0 * 2^{2-1} * 1.75$$
$$(-1)^0 * 3.5$$
$$3.5$$

Notes

Floating-point numbers

- Special encodings
  - There are positive and negative infinity values, where the exponent is all 1-bits and the mantissa is all 0-bits
  - There are special not a number (or NaN) values, where the exponent is all 1-bits and the mantissa is not all 0-bits
  - There are also a positive and a negative zero values, differing in the sign bit, where all other bits are 0

Notes

Floating-point numbers

- It is also important to note that floating-point numbers are only approximations
0	0	0	0	0	0	1

    $$(-1)^0 * 2^{0-1} * (1 + .0625) = 0.53125$$
0	0	0	0	0	1	0

    $$(-1)^0 * 2^{0-1} * (1 + .125) = 0.5625$$
  - $0.56250 - 0.53125 = 0.03125$

Notes

Floating-point numbers

- Note: the gap between adjacent floating point numbers increases with each power of two:
0	0	1	0	0	0	0

    $$(-1)^0 * 2^{1-1} * 1.00 = 1.000$$
0	0	1	0	0	0	1

    $$(-1)^0 * 2^{1-1} * (1 + .0625) = 1.0625$$
  - $1.0625 - 1.0 = 0.0625$

Notes

Floating-point numbers

- Floating-point numbers fundamentally imprecise because they represent fractional values with a finite number of bits
- Therefore, it makes sense that
  - `.15 + .15`
  - evaluates to `0.29999999999999999` on my machine
  - because `.15` cannot be represented exactly and instead is stored as `.14999999999999999`
- because the digits simply cut off when you get to the end of the mantissa
- so floating-point numbers are not exactly represented in the computer; a a rounding scheme is employed to deal with this
- an implication of this is that you cannot be certain that two floating point numbers are equal if they were calculated using arithmetic

Notes

Floating-point values

- Try calculating the value of 0.6 in binary by hand... would you feel comfortable if your bank used floating-point values to store monetary values?

Notes

Conversion errors

- A terminating decimal fraction could be converted into a nonterminating binary fraction
  - An example is decimal value 0.6, which is $0.\overline{1001}$ in binary
- This would indeed result in an approximation of the true value in the computer
- Such conversion errors are normally small; however, they can be propagated through calculations

Notes

Propagation of errors

- Assume computer truncates all numerical values to four decimal digits
- A floating-point number $A = \frac{2}{3}$ would be stored as 0.6666; the relative error would then be

$$r = \frac{\frac{2}{3} - 0.6666}{\frac{2}{3}} = 0.0001$$

- Notice what happens when A is added to itself six times:
  - $0.6666 + 0.6666 = 1.333$
  - $1.333 + 0.6666 = 1.999$
  - $1.999 + 0.6666 = 2.665$
  - $2.665 + 0.6666 = 3.331$
  - $3.331 + 0.6666 = 3.997$
- In this case, the relative error is propagated across the additions

$$r = \frac{4 - 3.997}{4} = 0.00075$$

Notes

Overview

Notes

Character data

- Another way to represent data in the computer is using binary-coded decimal (BCB) codes
- Initially, 6-bit BCB codes were used

B	A	8	4	2	1

  - The light green bits, labeled B and A, encoded zone bits
  - The light blue bits, labeled 8, 4, 2, and 1, encoded numeric bits
  - This scheme allowed 2^6 64 characters to be encoded (10 digits, 26 letters, and 28 special characters)
- This 6-bit code is usually represented in the computer as 7-bits, with the additional bit is known as a check bit

P	B	A	8	4	2	1

- Frequently, we require more than the 28 special characters provided under any 6-bit BCB code
  - 8-bit codes, comprised of four zone bits and four 8-4-2-1 bits, were therefore developed
  - Extended Binary-Coded Decimal Interchange Code (EBCDIC) by IBM
  - American Standard Code for Information Interchange (ASCII-8)

Notes

Overview

Notes

References

- Lewis, M. C. (2015). *Introduction to the art of programming using Scala*. CRC Press.
- Lipschutz, S. (1987). *Schaum's Outline of Essential Computer Mathematics*. McGraw-Hill.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Pearson Education.

Notes

Overview

Notes

Overview

- Binary number system
  - Motivation for study
  - Binary numbers and modern computers
  - What is a positional number system?
  - The binary number system
  - Binary to decimal conversion
  - Decimal to binary conversion
  - Binary addition
- Data representation in the computer
  - Integral numbers
  - Floating-point numbers
  - Character data
- References
- Appendix
  - Base-b to decimal conversion
  - Decimal to base-b conversion
  - Binary multiplication
  - Binary subtraction
  - Binary division
  - Complements

Notes

Base-b to decimal conversion

- A base-b number N_b can be converted to base-10 by writing N_b in expanded notation and calculating by decimal arithmetic
- We can also perform this conversion through the following algorithm:

  Integral part
  - i Multiply the leftmost digit by base b
  - ii Add the next digit to the right to that product
  - iii Multiply that sum by the base b
  - iv Add the next digit to that product
  - v Repeat until the right-most digit (that before the decimal point) is added

  Fractional part
  - i Multiply the rightmost fractional digit by $\frac{1}{b}$
  - ii Add the next digit to the left to that product
  - iii Multiply that sum by $\frac{1}{b}$
  - iv Add the next digit to that product
  - v Repeat until the leftmost digit (prior to decimal point) is added
  - vi Then multiply that sum by $\frac{1}{b}$

Notes

Overview

Notes

Decimal to base-b conversion

- Given decimal number N, with integral part N_I and fractional part N_F, N can be converted to base-b using the following algorithm:

  Integral part
  - i Divide N_I and each succeeding quotient by b until a zero quotient is obtained
  - ii The sequence of remainders in reverse order is the corresponding representation of N_I in base-b

  Fractional part
  - i Multiply N_F and the *fractional* portion of each succeeding product by b until a zero (or repeating) fractional part is observed
  - ii The resultant sequence of integral parts in-order gives the corresponding representation of N_F in base-b

Notes

Overview

Notes

Binary multiplication

- Can reduce multiplication of binary numbers to multiplication by bits and addition
- Simpler than decimal multiplication; multiplying a number by the multiplicand 1 or 0 yields either the multiplier or zero
- Let's consider the following example:

$$\begin{array}{r} 10011 \\ \times\ 110 \\ \hline \end{array}$$

Notes

Binary Multiplication Example

Step 1.) Multiplication: 10011×0

$$\begin{array}{r} 10011 \\ \times\ 110 \\ \hline 00000 \end{array}$$

Step 2.) Multiplication: 10011×1

$$\begin{array}{r} 10011 \\ \times\ 110 \\ \hline 00000 \\ 10011\_ \end{array}$$

Step 3.) Summation

$$\begin{array}{r} 10011 \\ \times\ 110 \\ \hline 00000 \\ +\ 10011\_ \\ \hline 100110 \end{array}$$

Step 4.) Multiplication: 10011×1

$$\begin{array}{r} 10011 \\ \times\ 110 \\ \hline 00000 \\ +\ 10011\_ \\ \hline 100110 \\ 10011\_\_ \end{array}$$

Step 5.) Summation

$$\begin{array}{r} 10011 \\ \times\ 110 \\ \hline 00000 \\ +\ 10011\_ \\ \hline {\scriptstyle 11} \\ 100110 \\ +\ 10011\_\_ \\ \hline 1110010 \end{array}$$

Notes

Overview

Notes

Binary subtraction

- Binary subtraction can be conducted similarly to the decimal subtraction:
  - If the subtrahend digit is greater than the minuend digit, appropriate from the next column to the left
  - Subtract the lower value from the upper value
    - $0 - 0 = 0$
    - $1 - 0 = 1$
    - $1 - 1 = 0$
    - $0 - 1 = 1$, with a borrow of 1 from the next column
- Let's consider the following example:

$$\begin{array}{r} 10110 \\ -\ 01101 \\ \hline \end{array}$$

Notes

Binary subtraction example

Step 1.) $1(\text{borrowed}) - 1 = 1$

$$\begin{array}{r} {\scriptstyle 0\,1} \\ 101\cancel{1}\cancel{0} \\ -\ 01101 \\ \hline 1 \end{array}$$

Step 2.) $0 - 0 = 0$

$$\begin{array}{r} {\scriptstyle 0\,1} \\ 101\cancel{1}\cancel{0} \\ -\ 01101 \\ \hline 01 \end{array}$$

Step 3.) $1 - 1 = 0$

$$\begin{array}{r} {\scriptstyle 0\,1} \\ 101\cancel{1}\cancel{0} \\ -\ 01101 \\ \hline 001 \end{array}$$

Step 4.) $1(\text{borrowed}) - 1 = 1$

$$\begin{array}{r} {\scriptstyle 0\,1\ \ 0\,1} \\ \cancel{1}\cancel{0}1\cancel{1}\cancel{0} \\ -\ 01101 \\ \hline 1001 \end{array}$$

Step 5.) $0 - 0 = 0$

$$\begin{array}{r} {\scriptstyle 0\,1\ \ 0\,1} \\ \cancel{1}\cancel{0}1\cancel{1}\cancel{0} \\ -\ 01101 \\ \hline 01001 \end{array}$$

Notes

Binary subtraction example 2

$$\begin{array}{r} 100 \\ -\,011 \\ \hline \end{array}$$

Step 1.)

$$\begin{array}{r} {\scriptstyle 011} \\ \cancel{100} \\ -\,011 \\ \hline 1 \end{array}$$

Step 2.)

$$\begin{array}{r} {\scriptstyle 011} \\ \cancel{100} \\ -\,011 \\ \hline 01 \end{array}$$

Step 3.)

$$\begin{array}{r} {\scriptstyle 011} \\ \cancel{100} \\ -\,011 \\ \hline 001 \end{array}$$

Notes

Overview

Notes

Binary division

- Binary division can be reduced to multiplying the divisor by individual digits of the dividend followed by subtraction
- Let's consider the following example:

$$11\overline{)10111}$$

Notes

Binary division example

Step 1.)

$$\begin{array}{rl} 1 & \\ 11\overline{)10111} & \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \end{array}$$

Step 2.)

$$\begin{array}{rl} 11 & \\ 11\overline{)10111} & \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \\ \underline{11} & \text{(Mult.)} \end{array}$$

Step 3.)

$$\begin{array}{rl} 11 & \\ 11\overline{)10111} & \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \end{array}$$

Step 4.)

$$\begin{array}{rl} 111 & \\ 11\overline{)10111} & \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \\ \underline{11} & \text{(Mult.)} \end{array}$$

Step 5.)

$$\begin{array}{rl} 111 & \\ 11\overline{)10111} & \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \\ \underline{11} & \text{(Mult.)} \\ 101 & \text{(Subt.)} \\ \underline{11} & \text{(Mult.)} \\ R = 10 & \text{(Subt.)} \end{array}$$

Notes

Overview

- Binary number system
  - Motivation for study
  - Binary numbers and modern computers
  - What is a positional number system?
  - The binary number system
  - Binary to decimal conversion
  - Decimal to binary conversion
  - Binary addition
- Data representation in the computer
  - Integral numbers
  - Floating-point numbers
  - Character data
- References
- Appendix
  - Base-b to decimal conversion
  - Decimal to base-b conversion
  - Binary multiplication
  - Binary subtraction
  - Binary division
  - Complements

Notes

Decimal complements

- Given the decimal number N, the nines (radix-minus one) complement can be found by subtracting each digit of N from 9
  - Given $N = 1234$, the nines complement is:

$$\begin{array}{r} 9999 \\ -\,1234 \\ \hline 8765 \end{array}$$

- The tens (radix) complement of that number N can then be found by adding 1 its nines complement
  - Given the nines complement of $N = 1234$ (calculated above), the tens complement can be calculated by adding one:

$$\begin{array}{r} 8765 \\ +\,1 \\ \hline 8766 \end{array}$$

Notes

Decimal complements and subtraction

- Given integers X and Y such that $X < Y$ and each integer is composed of four digits, the difference $Z = Y - X$ can be rewritten as
  - $Z = Y - X + (9999 + 1 - 10000)$
  - $Z = Y + (9999 - X + 1) - 10000)$
  - $Z = Y + [(9999 - X) + 1] - 10000$
- We can thus calculate the difference Z
  - By adding the tens complement of X to Y
  - And subsequently subtracting 10000
    - Notice the implications of using a positional system here... subtracting 10000 is as simple as deleting the leading 1 from the sum in this particular instance

Notes

Decimal complements and subtraction example

- Let's consider $Z = Y - X$ when $Y = 1012$ and $X = 0381$ (case: $X < Y$)

- Using normal subtraction,

$$\begin{array}{r} {\scriptstyle 0\,9\,1} \\ \cancel{1}\cancel{0}12 \\ -\ 381 \\ \hline 631 \end{array}$$

- Notice our need to borrow

- Using the tens complement of X, which is its nines complement 9618 plus 1 (9619)

$$\begin{array}{r} {\scriptstyle 1} \\ 1012 \\ +\ 9619 \\ \hline \cancel{1}0631 \end{array}$$

- Note: we added a trailing zero to X to make it the same 'length' as Y; when calculating its nines complement, $9 - 0$ is 9 at that place value

Notes

Decimal complements and subtraction example 2

- Let's now consider the $Z = Y - X$ when $Y = 1012$ and $X = 2017$ (case: $X > Y$)

- Using normal subtraction,

$$\begin{array}{r} 1012 \\ -\ 2017 \\ \hline -1005 \end{array}$$

- Using the tens complement of X, which is its nines complement 7982 plus 1 (7983)

$$\begin{array}{r} 1012 \\ +\ 7983 \\ \hline 8995 \end{array}$$

- As there is no overflow into the fifth-column, we need to take the negative of the tens complement of the sum to 'find' the desired difference *for our interpretation*
  - $$\begin{array}{r} 9999 \\ -\ 8995 \\ \hline 1004 \\ +\ 1 \\ \hline 1005 \end{array}$$
  - Negate $1005 \rightarrow -1005$

Notes

Binary complements

- For a binary number X, the ones complement (i.e., the base-2 radix-minus-one complement) can be obtained by subtracting each bit of X from 1; this is equivalent to inverting the bits
- Twos complement (i.e., the radix complement of the base-2 system) is then obtained by adding 1 to the ones complement of X

Notes

Binary complement and subtraction example

- We can evaluate the difference $Z = Y - X$, where $X = 1011$ and $Y = 1110$ by adding the twos complement of X to Y (case: $X < Y$)

$$\begin{array}{r} {}^{1}\\ 1110 \\ +\,0101 \\ \hline \cancel{1}0011 \end{array}$$

- If we were working on a system that implemented 4-bit registers, the leading one would be lost automatically as the result would exceed the capacity of the register holding the it (Z)
  - When using complements to reduce subtraction to addition, overflow will always occur whenever X is less than Y

Notes

Binary complement and subtraction example 2

- Let's now evaluate the difference $Z = Y - X$, where $X = 1110$ and $Y = 1011$ by adding the twos complement of X to Y (case: $X > Y$)

$$\begin{array}{r} 1011 \\ +\,0010 \\ \hline 1101 \end{array}$$

- Again, assuming 4-bit registers, note in this case ($X > Y$) that there is no leading one that is automatically lost
  - We therefore need to take the negative of the twos complement of the sum to 'find' the desired difference *for our interpretation*
  - $$\begin{array}{r} 1111 \\ -\,1101 \\ \hline 0010 \\ +\,1 \\ \hline 0011 \end{array}$$
  - Negate $0011 \to -0011$

Notes