Numbering Systems

Ver. 1.4

The Decimal System

- Consider value 234, there are:
  - 2 hundreds
  - 3 tens
  - 4 units

  that is: 2 x 100 + 3 x 10 + 4 x 1

  but 100, 10, and 1 are all powers of 10

  so value 234 can be written as:

  $2 \times \mathbf{10}^2 + 3 \times \mathbf{10}^1 + 4 \times \mathbf{10}^0$

The Decimal System

- That recurring number **10** is called the *base* of the numbering system
- The usual numbering system is then called *base 10* or *decimal*
- The decimal system is a *positional* numbering system, this means that the position of each digit implies the multiplication to a corresponding power of the base (the weight of that position)

The "base-n" System

- The base of a numbering system can be any integer value greater than 1
- Digits can have values from 0 to $n-1$ (e.g. the base-10 system has 10 possible digits, from 0 to 9, for bases >10 see the hexadecimal system)
- When not clear from the context, the base of a value must be indicated with a subscript value: 234_{10}

The “base-n” System

- Base-n systems are positional
- The same value is expressed in different ways depending on the numbering system used
- We want to convert a value from a generic base *n* to base 10 because we are acquainted only with the latter
- Some base-n numbering systems are more suited for special purposes

The Binary System

- Binary means base-2
- There are only 2 digits: 0 and 1
- The digits of a binary value are called ***bits*** (bit comes from “BInary digiT”)
- A value expressed in the binary system is a sequence of zeroes and ones: 100101_2
- The binary system is suited for computers

The Binary System

- What (decimal) value correspond to 100101_2?
- From the definition of positional numbering system: to convert a base-n value to base 10, each digit must be multiplied by the power **of the base** (2 in this case) corresponding to the digit position

The Binary System

write powers right to left on each digit starting from unit

5 4 3 2 1 0

- $100101_2 =$

$$
\begin{array}{lcr}
1 \times 2^5 + & \rightarrow & 32 + \\
0 \times 2^4 + & \rightarrow & 0 + \\
0 \times 2^3 + & \rightarrow & 0 + \\
1 \times 2^2 + & \rightarrow & 4 + \\
0 \times 2^1 + & \rightarrow & 0 + \\
1 \times 2^0 = & \rightarrow & 1 = \\
 & & ----- \\
 & & 37_{10}
\end{array}
$$

The Binary System

- A "bit" is a small quantity (!), so many bits are grouped together to build a more significant entity
- A **byte** is a sequence of 8 bits
  - 10100011
  - 00101001

The Binary System

- Other bit groupings are:
  - A **nibble** is composed of 4 bits
  - A **word** is composed of 2 bytes (16 bits)
  - A **double word** (**dword**) is composed of 2 words (4 bytes, 32 bits)
  - A **quad word** (**qword**) is composed of 4 words (8 bytes, 64)

  The term *word* has a completely different meaning when dealing with computer architecture

The Binary System

- For every bit sequence:
  - **MSB**: *Most Significant Bit*
    The leftmost bit is the most important of the sequence because it multiplies the highest power of the base

    10010010...10101001
  - **LSB**: *Least Significant Bit*
    The rightmost bit is the less important of the sequence because it multiplies the lowest power of the base (0)

    10010010...1010100**1**

The Octal System

- There are 8 digits:
  from 0 to 7 (no 8 nor 9 digits!)
- Example
  - 3642_8
- Conversion to base 10
  - 3 2 1 0
    $3642_8 = 3 \times 8^3 + 6 \times 8^2 + 4 \times 8^1 + 2 \times 8^0 = 1954_{10}$
  - $3\mathbf{8}4_8$ = NOT AN OCTAL NUMBER (8?)

The Hexadecimal System

- 16 digits: 0 to F (subscript 16 or H)
- The first 10 digits are the same as the decimal system, the other 6 digit symbols are taken from the alphabet:

  A_H → 10_{10} (*digit* "A", NOT *letter* "A")
  B_H → 11_{10}
  C_H → 12_{10}
  D_H → 13_{10}
  E_H → 14_{10}
  F_H → 15_{10}

The Hexadecimal System

3 2 1 0

- $C1B8_H = 12 \times 16^3 + 1 \times 16^2 + 11 \times 16^1 + 8 \times 16^0 = 49592_{10}$

Doubling and Dabbling

- Fast conversion to base 10 of small binary values (it could be used with other bases and big values, but memory computation is not such effective):
  - multiply the first (leftmost) bit (MSB) by the base (2) and add the second bit
  - multiply the value just calculated by the base and add the third bit
  - continue until the LSB is added

Doubling and Dabbling

- Example

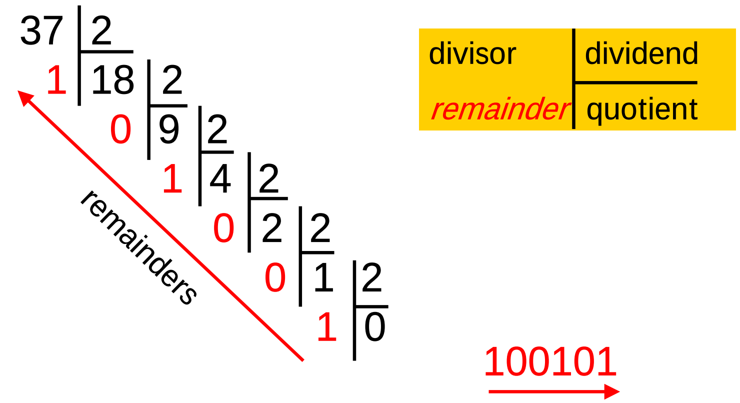
  100101$_2$
  - leftmost bit (MSB) is **1**
  - **1** x 2 = 2 + 0 = 2
  - 2 x 2 = 4 + 0 = 4
  - 4 x 2 = 8 + 1 = 9
  - 9 x 2 = 18 + 0 = 18
  - 18 x 2 = 36 + 1 = 37

Conversion from base 10 to n

- For *integer numbers* only:
  - Divide the value by the target base, computing an integer division, you get a quotient (result) and a remainder
  - Divide the previous quotient by the target base, you get a quotient and a remainder
  - Continue until the quotient is zero
  - Write (from left to right) the remainders from the **last** computed to the **first** one

Conversion from base 10 to n

- Example: convert 37_{10} to base 2



Conversion from Base n to m

- Given a number in base *n*, to convert it to base *m*:
  - convert the number from base *n* to base 10
  - convert the just calculated number from base 10 to base *m*
- It is always possible to pass through the intermediate base 10, but sometimes this is not the easiest and fastest way

Conversion Exercises

- Convert the values as requested
  - $106_{10} \rightarrow ()_2$
  - $35_{10} \rightarrow ()_2$
  - $64_{10} \rightarrow ()_2$
  - $45_6 \rightarrow ()_2$
  - $17_7 \rightarrow ()_2$
  - $23_8 \rightarrow ()_2$
  - $207_{10} \rightarrow ()_2$
  - $B2_{16} \rightarrow ()_{10}$
  - $47_8 \rightarrow ()_{10}$
  - $73_{10} \rightarrow ()_8$
  - $44_7 \rightarrow ()_3$
  - $101001_2 \rightarrow ()_{10}$
  - $11111_2 \rightarrow ()_{10}$
  - $10000_2 \rightarrow ()_{10}$
  - $10000_2 \rightarrow ()_4$
  - $161_{10} \rightarrow ()_{16}$

Conversion Exercises

- Convert the values as requested
  - 106_{10} → 1101010_2
  - 35_{10} → 100011_2
  - 64_{10} → 1000000_2
  - 45_6 → 11101_2
  - 17_7 → IMP.
  - 23_8 → 10011_2
  - 207_{10} → 11001111_2
  - $B2_{16}$ → 178_{10}
  - 47_8 → 39_{10}
  - 73_{10} → 111_8
  - 44_7 → 1012_3
  - 101001_2 → 41_{10}
  - 11111_2 → 31_{10}
  - 10000_2 → 16_{10}
  - 10000_2 → 100_4
  - 161_{10} → $A1_{16}$

Conversion from Bases 2^n

- To convert a number from base *n* to base *m*, when BOTH *n* and *m* are powers of 2 (e.g. 2,4,8,16), it is easier and faster to pass through base 2
- Every digit in a 2^x base requires at least *x* bits (possibly starting with zeroes):
  $5_8 = 101_2$ $2_8 = 010_2$
  $B_{16} = 1011_2$ $2_{16} = 0010_2$

Conversion from Bases 2^n

- A number in base-2^x can be converted to base 2 by simply substituting each of its digits with the corresponding binary value (each composed of *x* bits):
  $52_8 = [101][010]_2 = 101010_2$
  $B2_{16} = [1011][0010]_2 = 10110010_2$

Conversion from Bases 2^n

- A number in base 2 can be converted to base-2^x by grouping its bits from right to left (each group composed of *x* bits) and substituting each group with the corresponding base-2^x digit:
  $101010_2 = [101][010]_2 = 52_8$
  $10110010_2 = [1011][0010]_2 = B2_{16}$

Conversion from Bases 2^n

- If the leftmost group has less than *x* bits, an appropriate number of zeroes are added to the left part:
  $10010_2 = [\underline{\mathit{0}}10][010]_2 = 22_8$
  $110010_2 = [\underline{\mathit{00}}11][0010]_2 = 32_{16}$

Conversion from Bases 2^n

- Example
  Convert number $3CB21F_{16}$ to base 8

  3 C B 2 1 F
  0011 1100 1011 0010 0001 1111
  0011 1100 1011 0010 0001 1111
  1 7 1 3 1 0 3 7

  17131037_8

Conversion from Bases 2^n

- The most important base in Computer Science is 2, but it is difficult and long to write and read long binary values
- Base 16 and 8 are so often used because it is simple and immediate to convert a value between them and base 2 and because of the compact notation:
  it is *much* easier to read and write a 32 bit value like 95DBA6CF instead of
  10010101110110111010011011001111

Exercises on Conversions

- Convert the values as requested
  - $1001010010100101_2 \rightarrow ()_8$
  - $1001010010100101_2 \rightarrow ()_H$
  - $3325_8 \rightarrow ()_2$
  - $3325_8 \rightarrow ()_4$
  - $3325_8 \rightarrow ()_{16}$
  - $1334_8 \rightarrow ()_{16}$
  - $A116_{16} \rightarrow ()_8$
  - $13364_8 \rightarrow ()_{16}$

Exercises on Conversions

- Solutions
  - $\underline{00}1|001|010|010|100|101_2$ → 112245_8
  - $1001|0100|1010|0101_2$ → $94A5_H$
  - 3325_8 → 011011010101_2
  - 3325_8 → 123111_4
  - 3325_8 → $6D5_{16}$
  - 1334_8 → $2DC_{16}$
  - $A116_{16}$ → 120426_8
  - 13364_8 → $16F4_{16}$

Powers of 2

- $2^0=1_{10}$ $=1_2$
- $2^1=2_{10}$ $=10_2$
- $2^2=4_{10}$ $=100_2$
- $2^3=8_{10}$ $=1000_2$
- $2^4=16_{10}$ $=10000_2$
- $2^5=32_{10}$ $=100000_2$
- $2^6=64_{10}$ $=1000000_2$
- $2^7=128_{10}$ $=10000000_2$
- $2^8=256_{10}$ $=100000000_2$
- Note that 2^n in binary is 1 followed by *n* zeroes

$2^9=512$
$2^{10}=1024$
$2^{11}=2048$
$2^{12}=4096$
$2^{13}=8192$
$2^{14}=16384$
$2^{15}=32768$
$2^{16}=65536$

Range

- With *n* bits, just a limited subset of values can be represented
- There are 2^n different combinations of *n* bits, each one is a binary number:

$$\left.\begin{array}{lll} \textit{from} & 000...000 & \rightarrow 0 \\ & ... & ... \\ \textit{to} & 111...111 & \rightarrow 2^n-1 \end{array}\right\} 2^n \text{ numbers}$$

Thus the range of a binary number composed of *n* bits is: 0 ➔ 2^n-1

Number of bits required

- Given a decimal value N, the same value converted to binary requires a minimum of bits equal to:

$$n = \lceil \log_2(N+1) \rceil$$

where the *ceiling* operator $\lceil a \rceil$ returns the minimum integer greater then or equal to a (e.g. 2.1 → 3, 2.9 → 3, 2.0 → 2)

Number of bits required

- A simpler approach uses (wise) trial:
  - count the digits (d) of the decimal num. N
  - each digit requires about 3 bits (good approximation up to 20-30 bit numbers), so the approximate number of bits is: $x = 3 \times d$
  - compare N to powers 2^n with n ranging from (x-1) to (x+1) to find the minimum n so that $2^n \geq N$
  - Note: if needed, consider extending n to (x –2) or to (x +2)

Number of bits required

- Example - How many bits are required for number 400?

  Estimate:
  - 3 decimal digits → 3x3 = 9 bits

  Check values:
  - 9 bits → range: 0 ➔ $2^9 - 1 = 511 \geq 400$ OK

  Verify if a smaller value is also good:
  - 8 bits → range: 0 ➔ $2^8 - 1 = 255 < 400$ NO

  Answer: at least 9 bits

Number of bits required

- Exercises

  How many bits are needed to represent the following values?
  - 47
  - 137
  - 1412
  - 128
  - 884
  - 1
  - 422
  - 15
  - 444
  - 1024
  - 1023
  - 6443

Number of bits required

- Solutions
  - 47 → 6
  - 137 →8
  - 1412 → 11
  - 128 → 8
  - 884 → 10
  - 1 → 1
  - 422 → 9
  - 15 → 4
  - 444 → 9
  - 1024 → 11
  - 1023 → 10
  - 6443 → 13

Fractional Numbers Conversion

- Conversion from any base *n* to base 10 just requires the application of the positional numbering system definition
- Example
  $1011.101_2 \rightarrow ()_{10}$

  3 2 1 0 -1-2-3
  $1011.101 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} =$
  $= 8 + 2 + 1 + 0.5 + 0.125 = 11.625_{10}$

Fractional Numbers Conversion

- Conversion from base 10 to any base *n* requires 2 steps:
  - conversion of the integral part (already seen)
  - conversion of the fractional part (to be seen)

  The two parts are then juxtaposed (added)

Fractional Numbers Conversion

- Conversion of the fractional part (i.e. a value in the form 0.xxxx)
  - multiply the number to the target base (e.g. 2)
  - write down the *integer part* of the result and THEN set it to 0
  - repeat until result is 0 or as otherwise required (more on this later)
  - write left to right “0.” followed by the *integer parts* in the order they have been calculated

Fractional Numbers Conversion

- Example, convert 0.6875_{10} to binary
  0.6875 x 2 = **1**.3750
  0.3750 x 2 = **0**.750
  0.750 x 2 = **1**.50
  0.50 x 2 = **1**.00
  0.00 STOP

  0.1011_2
- Example, convert 12.6875 to binary:
  Result: 1100.1011_2

Fractional Numbers Conversion

- To convert a number with a fractional part from one base to another, it is always possible to pass through the intermediate base 10
- When BOTH bases are powers of 2, it is easier and faster to pass through base 2

Fractional Numbers Conversion

- Regrouping must *start from the point* so that unity remains on the digit at left of the point, if required, zeroes **must** be added *on the right* of the fractional part
- Example
  $C4B2.D6_H \rightarrow ()_8$
  $1100\ 0100\ 1011\ 0010.1101\ 0110_2$
  x 2^0
  $001\ 100\ 010\ 010\ 110\ 010\ .\ 110\ 101\ 100$
  $1\quad 4\quad 2\quad 2\quad 6\quad 2\ .\ 6\quad 5\quad 4_8$

Fractional Numbers Conversion

- A value with a finite number of fractional digits in one base may require an infinite number of fractional digits in another base (often periodic in binary)
- E.g. 2.31 →10.01001100110011001…
- When converting values with an unlimited fractional part, the number of fractional digit to calculate must be known in advance (in some way)

Exercises on Conversions

- Convert the values as requested
  - $56.22_8 \rightarrow ()_{16}$
  - $CC559.9B1_{16} \rightarrow ()_8$
  - $1001.11_2 \rightarrow ()_{10}$
  - $11101.011_2 \rightarrow ()_{10}$
  - $1000.0001_2 \rightarrow ()_{10}$
  - $33.25_{10} \rightarrow ()_2$ (3 fractional digits)
  - $13.34_{10} \rightarrow ()_2$ (5 fractional digits)
  - $256.22_{10} \rightarrow ()_2$ (6 fractional digits)

Exercises on Conversions

- Convert the values as requested
  - 56.22_8 → $2E.48_{16}$
  - $CC559.9B1_{16}$ → 3142531.4661_8
  - 1001.11_2 → 9.75_{10}
  - 11101.011_2 → 29.375_{10}
  - 1000.0001_2 → 8.0625_{10}
  - 33.25_{10} → 100001.010_2
  - 13.34_{10} → 1101.01010_2
  - 256.22_{10} → 100000000.001110_2

Approximation Errors

- When we have to limit the number of fractional digits, the value resulting from conversion is not the same as the original value
- This means that if the resulting value is converted back to the original base, it is slightly different
- An error is introduced

Approximation Errors

- **Absolute precision:** the smallest (positive) quantity that can be written by using a given number of fractional digits

$$\varepsilon = \frac{1}{b^n}$$

where:

b is the numbering base

n is the number of the *fractional* digits

N is the given number

Approximation Errors

- Examples
  - In decimal, with 5 fractional digits the smallest positive quantity is $0.00001_{10} = 1/10^5 = 0.00001_{10}$
  - In binary, with 5 fractional digits the smallest positive quantity is $0.00001_2 = 1/2^5 = 0.03125_{10}$
  - In octal, with 5 fractional digits the smallest positive quantity is $0.00001_8 = 1/8^5 = 0.000030517578125_{10}$

Approximation Errors

- 0.4 has 1 fractional digit, so its absolute precision ε is $1/10^1 = 0.1$
- 10.4 has 1 fractional digit, so its absolute precision ε is $1/10^1 = 0.1$
- Is ε=0.1 a good or a bad precision?
  It depends on the value itself: you have to compare the absolute error to the given value

Approximation Errors

- An absolute precision value may have different significance with respect to the value it is computed for
- **Relative precision:** the absolute precision compared to the given value (usually as a percentage)

$$\eta = \frac{\varepsilon}{|N|} \cdot 100\%$$

Approximation Errors

- Complete

N	ε	η
0.4_{10}	$1/10^1=0.1$	0.1/0.4*100 = 25%
10.4_{10}		
0.1011_2	$1/2^4=0.0625$	
100_{10}		

Approximation Errors

- Solutions

N	ε	η
0.4_{10}	0.1	25%
10.4_{10}	0.1	0.96%
0.1011_2	0.0625	9.09%
100_{10}	1	1%

Approximation Errors

- In a base conversion, a given error margin (also called *precision* or *approximation*) ε_o must not be exceeded
- Errors can be used to establish how many fractional digits to use

Approximation

- Example 1 - Convert value N=0.21 to base 2 with ε_o=1/32 (i.e. 0.03125)
  - Compute how many fractional bits are needed: because the required $\varepsilon_o=1/32$ must be equal to the theoretic $\varepsilon=1/2^n$, then $1/32 = 1/2^n \rightarrow 32=2^n \rightarrow 2^5=2^n \rightarrow n = 5$
  - Calculate the first 5 fractional bits
  - Write the result: 0.00110_2
  - The trailing zero is required: without it the ε would be $1/2^4 = 1/16$ and not 1/32

Approximation

- Note that $0.00110_2 = 0.1875_{10}$ and this is NOT the given value 0.21_{10}
- However the absolute difference between them (the introduced *error*) is less than or equal to the maximum allowed error 1/32 (0.03125): $|0.21 - 0.1875| = 0.0225 \leq 0.03125$

Approximation

- Example 2 - Convert value N=0.21 to base 2 with absolute error ε_o=1/100
  - Compute how many fractional bits are needed: because the required $\varepsilon_o=1/100$ must be equal to the theoretic $\varepsilon=1/2^n$, then $1/100 = 1/2^n \rightarrow 100=2^n \rightarrow n = ?$
    For solving this equation we can use logarithms, but we can use the trial method:
    n=6 bits $\rightarrow 2^n = 64 < 100$ not enough!
    n=7 bits $\rightarrow 2^n = 128 \geq 100$ OK!
  - Calculate the first 7 fractional bits

Approximation

- Note that an error of exactly 1/100 cannot be obtained because 100 is not a power of 2
- Instead of 1/100 we use its nearest (smaller) power of 2, *resulting in an error smaller than the one requested*, so that the requirements are fulfilled
- Having now a power of 2, the exponent can be easily found

Exercises on Conversions

- Convert the values as requested
  - 33.225_{10} → $()_2$ (ε_o =1/64)
  - 13.34_{10} → $()_2$ (ε_o =1/1000)
  - 256.22_{10} → $()_2$ (η_o =0.001%)
  - 12.71_{10} → $()_2$ (preserve the same ε of the decimal value)
  - 12.71_{10} → $()_2$ (preserve the same η)

Exercises on Conversions

- Solutions
  - 33.225_{10}

    $\varepsilon = \frac{1}{64} = \frac{1}{2^n}$ $2^n \geq 64$ → n=6

    100001.001110
  - 13.34_{10}

    $\varepsilon = \frac{1}{1000} = \frac{1}{2^n}$ $2^n \geq 1000$ → n=10

    1101.0101011100

Exercises on Conversions

- Solutions
  - 256.22_{10}

$$\eta_0 = \frac{\varepsilon_0}{N}\cdot 100 \rightarrow 0.001 = \frac{\varepsilon_0}{256.22}\cdot 100 \rightarrow \varepsilon_0 = 0.0025622$$

$$\varepsilon = \frac{1}{2^n} \rightarrow \varepsilon = \frac{1}{2^n} = 0.0025622$$

$$2^n = 390.3$$

$$2^n > 390.3$$

$$n = 9\text{ bit}$$

Exercises on Conversions

- Solutions
  - 12.71_{10}

    $\varepsilon_2 = \varepsilon_{10} = \frac{1}{10^2} = \frac{1}{100} = \frac{1}{2^n}$ → $n=7$

    1100.1011010
  - 12.71_{10}

    $\eta_{10} = \frac{\varepsilon_{10}}{12.71} \cdot 100 \qquad \eta_2 = \frac{\varepsilon_2}{12.71} \cdot 100$

    $\eta_{10} = \eta_2$ ➔ $\varepsilon_{10} = \varepsilon_2$ ➔ same ex. as before

BCD Encoding

- In many cases, the approximation involved with conversion to base 2 is not acceptable
- The most prominent case is currency
- The only way is **to not convert to binary**, but digital computers do need information stored as bits…
- What to do?

BCD Encoding

- **B**inary-**C**oded **D**ecimal is an encoding for decimal numbers in which each digit is represented by its own binary value
- Each binary value is composed of 4 *bits*
- Only 10 groups corresponding to values from 0 to 9 (from 0000 to 1001) are allowed
- *This is NOT an equivalent way to convert to/from base 2!*

BCD Encoding

- Example – Convert value 23.19 to BCD
  Every decimal digit is converted to the corresponding 4-bit binary value:

  2 3 . 1 9

  $0010\ 0011\ .\ 0001\ 1001_{BCD}$

  $0010_2 \times \mathbf{10}^1 + 0011_2 \times \mathbf{10}^0 +$
  $+ 0001_2 \times \mathbf{10}^{-1} + 1001_2 \times \mathbf{10}^{-2}$

  Note the base used is 10, just the decimal digits are expressed in binary

BCD Encoding

- Comparison – Convert value 126.625 to both base 2 and BCD
  - 1111110.101_2
  - $000100100110.011000100101_{BCD}$

  The two sequences of bits are quite different, the only way to transform one into the other is through base 10
- Other types of BCD encoding exist, the one just seen is called *Simple BCD* (SBCD) or *BCD 8421*

BCD Encoding

- BCD values are stored in different ways on different machines:
  - one byte for each digit, the higher nibble can be set to:
    - 0000 or 1111
    - 0011 (in this case, the resulting value is the ASCII code of the value, e.g. BCD digit 0010, if stored preceded by 0011 becomes: 00110010 that is the ASCII value for character '2', i.e. 50)
  - one byte for two digits (*packed BCD*)
  - other compressed ways

BCD Encoding

- BCD operations are slower than binary operations
- BCD circuits are bigger
- Space is wasted (unused bit sequences)
- No approximation errors
- Easy scaling of a factor of 10
- Rounding at a decimal boundary is easy

Exercises on BCD

- Convert the values as requested
  - 123.21_{10} $\rightarrow ()_{BCD}$
  - $82.C_{16}$ $\rightarrow ()_{BCD}$
  - 12.2_{16} $\rightarrow ()_{BCD}$
  - $000100100110.10010001_{BCD}$ $\rightarrow ()_{10}$
  - $000100100110.10010001_{BCD}$ $\rightarrow ()_{2}$
  - 100100110.10010001_2 $\rightarrow ()_{BCD}$

Exercises on BCD

- Solutions
  - 123.21_{10} → $000100100011.00100001_{BCD}$
  - $82.C_{16}$ → 130.75_{10} → → $000100110000.01110101_{BCD}$
  - 12.2_{16} → 18.125_{10} → → $00011000.000100100101_{BCD}$
  - $\underline{000}100100110.10010001_{BCD}$ → 126.91_{10}
  - $\underline{000}100100110.10010001_{BCD}$ → $1111110.11\ldots_2$
  - 100100110.10010001_2 → $294.566\ldots_{10}$ → → $001010010100.010101100110_{BCD}$

Binary Prefixes

- The physical quantities use prefixes as multipliers, their values are powers of 10
- In the binary notation the same prefixes are used, but as powers of 2

Prefix	K	M	G	T	P
Name	kilo	mega	giga	tera	peta
Physics value	10^3	10^6	10^9	10^{12}	10^{15}
Binary value	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}

Binary Prefixes

- An attempt to define separate prefixes for powers of 2 lead to the definition of the (seldom used) following prefixes

Prefix	Ki	Mi	Gi	Ti	Pi
Name	kibi	mebi	gibi	tebi	pebi
Value	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}

Binary Addition

- Usual rules apply:
  - 0+0 = 0
  - 0+1 = 1+0 = 1
  - 1+1 = 0 with carry = 1 to the following power of 2 (that is: 10_2)
  - 1+1+1= 1 with carry = 1 (that is: 11_2)
- It is useful to add column by column, writing carries on top of the next column

Binary Addition

- Example

```
 1111
  10110  +
   1011  =
 ------
 100001
```

non null carries

Exercises on Addition

- Complete:
  - 0+1=
  - 1+1=
  - 10+1=
  - 11+1=
  - 100+1=
  - 101+1=
  - 111+1=
  - 1000+1=
  - 11111+1=

Exercises on Addition

- Solution:
  - 0+1=1
  - 1+1=10
  - 10+1=11
  - 11+1=100
  - 100+1=101
  - 101+1=110
  - 111+1=1000
  - 1000+1=1001
  - 11111+1=100000

Binary Subtraction

- Usual rules apply:
  - 0–0 = 0
  - 1–0 = 1
  - 1–1 = 0
  - 0–1= 1 with borrow = 1 (borrowed from the nearest 1 leftmost)

    Remember that the 1 that gives its value becomes 0 and any intermediate 0 becomes 1 (the highest digit in base 2)

Binary Subtraction

- Examples

```
0 1
1011  –
 110  =
0101
```

The nearest leftmost 1 gives its value and becomes 0
The borrowing 0 becomes 10

```
 0 1 1 1 1
110000 –
    11 =
101101
```

Intermediate 0s becomes 1s
Remember, in base 10 (the highest digit is 9) we have:

```
0 9 9 1
1000 –
   1 =
0999
```


Exercises on Add & Sub

- Complete:
  - 1010 + 10010=
  - 11 + 11=
  - 11011 + 1001=
  - 1101 + 111=
  - 10000 – 10=
  - 11010 – 10101=
  - 10010 – 1111=
  - 10101 – 10101=
  - 10000 – 111=

Exercises on Add & Sub

- Solutions :
  - 1010 + 10010 = 11100
  - 11 + 11 = 110
  - 11011 + 1001 = 100100
  - 1101 + 111 = 10100
  - 10000 – 10 = 1110
  - 11010 – 10101 = 101
  - 10010 – 1111 = 11
  - 10101 – 10101 = 0
  - 10000 – 111 = 1001

Overflow

- The binary numbering system we used until now does not take into account any limitation to the number of bits that can be used
- When binary numbers are stored in a digital computer, the number of bits available is an architectural, fixed, and limiting characteristic

Overflow

- It is not possible to store a number that requires more bits than those provided by the hardware in use (it is out of range)
- When a non-storable number results *from a calculation* (e.g. an addition), it is not a correct value (must be discarded) and there is an **Overflow** error condition

Overflow

- Example
  Consider a computing machine where numbers are stored in 8-bit variables

$$
\begin{array}{r}
\boxed{10011001}\ + \\
\boxed{11001100}\ = \\
\hline
1\boxed{01100101}
\end{array}
$$

  Note that the result requires 9 bits, the machine cannot store it and then signals an Overflow error condition

Shift Operations

- Simple multiplication and division by a power of 2 is achieved by shifting the number bits
- “Shifting” means moving each bit either to the right (right shift) or to the left (left shift)
- Symbols « and » are used to identify the shift operation, they are followed by the number of shifts to perform

Shift Operations

- **Left shift** («): a zero is added to the right

$$1010_2 \rightarrow 10_{10}$$
$$10100_2 \rightarrow 20_{10}$$
$$101000_2 \rightarrow 40_{10}$$

- Each left shift doubles the value, (actually it multiplies the value by the base, in base 10: 12«1 = 120)
- n left shifts ➔ multiplication by 2^n

Shift Operations

- **Right shift** (»): for integer values, the LSB is discarded, for fractional values the LSB goes beyond the radix point

$1010_2 \rightarrow 10_{10}$
$101_2 \rightarrow 5_{10}$
$10_2 \rightarrow 2_{10}$ ($10.1_2 = 2.5_{10}$ for fract. val.)

- Each right shift halves the value, for integer values it is an integer division with truncation of the fractional part
- *n* right shifts ➔ division by 2^n

Exercises on Shifts

- Calculates the following op. by using shifts on the binary integer notation
  - 124 / 8 e.g. $1111\underline{100}$»3 → 1111
  - 22 * 4
  - 128 * 16
  - 131 / 2
  - 28 * 8
  - 47 * 2
  - 12 / 16

Exercises on Shifts

- Solutions
  - 124 / 8 $1111\underline{100}$»3 → 1111
  - 22 * 4 10110«2 → $10110\underline{00}$
  - 128 * 16 10000000«4 → $10000000\underline{0000}$
  - 131 / 2 $1000001\underline{1}$»1 → 1000001
  - 28 * 8 11100«3 → $11100\underline{000}$
  - 47 * 2 101111«1 → $101111\underline{0}$
  - 12 / 16 $\underline{1100}$»4 → 0