Batos Radio & TV

Your one stop Electronic Shop

Bato`s House
93 3rd St
Springs 1560

Tel : **(011) 362-2085** Fax : (011) 815-3637
batos.co.za Email : bato@batos.co.za

Skype Contact : **batojak**
You Tube: **batojak**
P.O. Box 1518
Springs 1560
Gauteng R.S.A.

With Compliments

Importers & Distributors of Electronic Components

Fully but I mean **Fully Equiped Workshop** + small scale manufacturing

We offer the most Comprehensive range of Electronic Components *..*Fully equiped and most sophisticated Workshop on the East Rand *..*Speaker & Speaker Box Test Fascility ..*We custom make SHAKER speaker Crosovers* etc.etc

Get the Distance with Batos Antennas and 2 way Radios ! ! !

Bato de ZS6BAT

Batos Radio & TV

Your one stop Electronic Shop

Bato`s House
93 3rd St
Springs 1560
Gauteng
South Africa

Tel : (011) 362-2085 Fax : (011) 815-3637
batos.co.za Email : bato@batos.co.za

Skype Contact : batojak
You Tube: batojak
P.O. Box 1518
Springs 1560
Gauteng
South Africa

Assembly

An **assembly language** is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Each assembly language is specific to a particular computer architecture, in contrast to most high-level programming languages, which are generally portable across multiple systems.
Assembly language is converted into executable machine code by a utility program referred to as an *assembler*; the conversion process is referred to as *assembly*, or *assembling* the code.
Assembly language uses a mnemonic to represent each low-level machine operation or opcode. Some opcodes require one or more operands as part of the instruction, and most assemblers can take labels and symbols as operands to represent addresses and constants, instead of hard coding them into the program. **Macro assemblers** include a macroinstruction facility so that assembly language text can be pre-assigned to a name, and that name can be used to insert the text into other code. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging.
A program written in assembly language consists of a series of (mnemonic) processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters.These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

Assembler

An **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities.The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution—e.g., to generate common short sequences of instructions as inline, instead of *called* subroutines.
Assemblers have been available since the 1950s and are far simpler to write than compilers for high-level languages as each mnemonic instruction / address mode combination translates directly into a single machine language opcode. Modern assemblers, especially for RISC architectures, such as SPARC or Power Architecture, as well as x86 and x86-64, optimize Instruction scheduling to exploit the CPU pipeline efficiently.

Decimal

Decimal counting uses the ten symbols **0** through **9**. Counting primarily involves incremental manipulation of the "low-order" digit, or the rightmost digit, often called the "first digit". When the available symbols for the low-order digit are exhausted, the next-higher-order digit (located one position to the left) is incremented, and counting in the low-order digit starts over at 0. In decimal, counting proceeds like so:

000, 001, 002, ... 007, 008, 009, (rightmost digit starts over, and next digit is incremented)
0**1**0, 011, 012, ...
090, 091, 092, ... 097, 098, 099, (rightmost two digits start over, and next digit is incremented)
100, 101, 102, ...

After a digit reaches 9, an increment resets it to 0 but also causes an increment of the next digit to the left.

Binary

In binary, counting follows similar procedure, except that only the two symbols **0** and **1** are used. Thus, after a digit reaches 1 in binary, an increment resets it to 0 but also causes an increment of the next digit to the left:

0000,
000**1**, (rightmost digit starts over, and next digit is incremented)
00**1**0, 0011, (rightmost two digits start over, and next digit is incremented)
0**1**00, 0101, 0110, 0111, (rightmost three digits start over, and the next digit is incremented)
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 ...

Since binary is a base-2 system, each digit represents an increasing power of 2, with the rightmost digit representing 2^0, the next representing 2^1, then 2^2, and so on. To determine the decimal representation of a binary number simply take the sum of the products of the binary digits and the powers of 2 which they represent. For example, the binary number 100101 is converted to decimal form as follows:

$100101_2 = [\ (\mathbf{1}) \times 2^5\ ] + [\ (\mathbf{0}) \times 2^4\ ] + [\ (\mathbf{0}) \times 2^3\ ] + [\ (\mathbf{1}) \times 2^2\ ] + [\ (\mathbf{0}) \times 2^1\ ] + [\ (\mathbf{1}) \times 2^0\ ]$
$100101_2 = [\ \mathbf{1} \times 32\ ] + [\ \mathbf{0} \times 16\ ] + [\ \mathbf{0} \times 8\ ] + [\ \mathbf{1} \times 4\ ] + [\ \mathbf{0} \times 2\ ] + [\ \mathbf{1} \times 1\ ]$
$\mathbf{100101_2 = 37_{10}}$

To create higher numbers, additional digits are simply added to the left side of the binary representation

Hexadecimal

Binary may be converted to and from hexadecimal somewhat more easily. This is because the radix of the hexadecimal system (16) is a power of the radix of the binary system (2). More specifically, $16 = 2^4$, so it takes four digits of binary to represent one digit of hexadecimal, as shown in the table to the right.

To convert a hexadecimal number into its binary equivalent, simply substitute the corresponding binary digits:

$3A_{16} = 0011\ 1010_2$

$E7_{16} = 1110\ 0111_2$

To convert a binary number into its hexadecimal equivalent, divide it into groups of four bits. If the number of bits isn't a multiple of four, simply insert extra **0** bits at the left (called padding). For example:

$1010010_2 = 0101\ 0010$ grouped with padding $= 52_{16}$

$11011101_2 = 1101\ 1101$ grouped $= DD_{16}$

To convert a hexadecimal number into its decimal equivalent, multiply the decimal equivalent of each hexadecimal digit by the corresponding power of 16 and add the resulting values:

$C0E7_{16} = (12 \times 16^3) + (0 \times 16^2) + (14 \times 16^1) + (7 \times 16^0) = (12 \times 4096) + (0 \times 256) + (14 \times 16) + (7 \times 1) = 49{,}383_{10}$

0_{hex}	=	0_{dec}	=	0_{oct}	0	0	0	0
1_{hex}	=	1_{dec}	=	1_{oct}	0	0	0	1
2_{hex}	=	2_{dec}	=	2_{oct}	0	0	1	0
3_{hex}	=	3_{dec}	=	3_{oct}	0	0	1	1
4_{hex}	=	4_{dec}	=	4_{oct}	0	1	0	0
5_{hex}	=	5_{dec}	=	5_{oct}	0	1	0	1
6_{hex}	=	6_{dec}	=	6_{oct}	0	1	1	0
7_{hex}	=	7_{dec}	=	7_{oct}	0	1	1	1
8_{hex}	=	8_{dec}	=	10_{oct}	1	0	0	0
9_{hex}	=	9_{dec}	=	11_{oct}	1	0	0	1
A_{hex}	=	10_{dec}	=	12_{oct}	1	0	1	0
B_{hex}	=	11_{dec}	=	13_{oct}	1	0	1	1
C_{hex}	=	12_{dec}	=	14_{oct}	1	1	0	0
D_{hex}	=	13_{dec}	=	15_{oct}	1	1	0	1
E_{hex}	=	14_{dec}	=	16_{oct}	1	1	1	0
F_{hex}	=	15_{dec}	=	17_{oct}	1	1	1	1

When a Pic is described as being 8 or 16 bit , this referes to the amount of data that can processed at once : the width of the data memory (registers in Microchip terminology) and ALU (arithmetic and logic Unit).

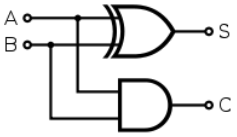
Binary Arithmetic

Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction, multiplication, and division can be performed on binary numerals.

Addition

The circuit diagram for a binary half adder, which adds two bits together, producing sum and carry bits.

The simplest arithmetic operation in binary is addition. Adding two single-digit binary numbers is relatively simple, using a form of carrying:



0 + 0 → 0

0 + 1 → 1

1 + 0 → 1

1 + 1 → 0, carry 1 (since 1 + 1 = 0 + 1 × binary 10)

Adding two "1" digits produces a digit "0", while 1 will have to be added to the next column. This is similar to what happens in decimal when certain single-digit numbers are added together; if the result equals or exceeds the value of the radix (10), the digit to the left is incremented:

5 + 5 → 0, carry 1 (since 5 + 5 = 10 carry 1)

7 + 9 → 6, carry 1 (since 7 + 9 = 16 carry 1)

This is known as *carrying*. When the result of an addition exceeds the value of a digit, the procedure is to "carry" the excess amount divided by the radix (that is, 10/10) to the left, adding it to the next positional value.

This is correct since the next position has a weight that is higher by a factor equal to the radix. Carrying works the same way in binary

```
  1 1 1 1 1     (carried digits)
    0 1 1 0 1
+   1 0 1 1 1
-------------
```

= 1 0 0 1 0 0 = 36

In this example, two numerals are being added together: 01101_2 (13_{10}) and 10111_2 (23_{10}). The top row shows the carry bits used. Starting in the rightmost column, $1 + 1 = 10_2$. The 1 is carried to the left, and the 0 is written at the bottom of the rightmost column. The second column from the right is added: $1 + 0 + 1 = 10_2$ again; the 1 is carried, and 0 is written at the bottom. The third column: $1 + 1 + 1 = 11_2$. This time, a 1 is carried, and a 1 is written in the bottom row. Proceeding like this gives the final answer 100100_2 (36 decimal).

Subtraction

0 − 0 → 0
0 − 1 → 1, borrow 1
1 − 0 → 1
1 − 1 → 0

Subtracting a "1" digit from a "0" digit produces the digit "1", while 1 will have to be subtracted from the next column. This is known as *borrowing*. The principle is the same as for carrying. When the result of a subtraction is less than 0, the least possible value of a digit, the procedure is to "borrow" the deficit divided by the radix (that is, 10/10) from the left, subtracting it from the next positional value.

```
   *   * * *   (starred columns are borrowed from)
  1 1 0 1 1 1 0
-     1 0 1 1 1
---------------
```

= 1 0 1 0 1 1 1

Subtracting a positive number is equivalent to *adding* a negative number of equal absolute value; computers typically use two's complement notation to represent negative values. This notation eliminates the need for a separate "subtract" operation. Using two's complement notation subtraction can be summarized by the following formula:
A − B = A + not B + 1

Multiplication

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Multiplication in binary is similar to its decimal counterpart. Two numbers *A* and *B* can be multiplied by partial products: for each digit in *B*, the product of that digit in *A* is calculated and written on a new line, shifted leftward so that its rightmost digit lines up with the digit in *B* that was used. The sum of all these partial products gives the final result. Since there are only two digits in binary, there are only two possible outcomes of each partial multiplication:

- If the digit in *B* is 0, the partial product is also 0
- If the digit in *B* is 1, the partial product is equal to *A*

For example, the binary numbers 1011 and 1010 are multiplied as follows:

```
1 0 1 1    (A)
       × 1 0 1 0    (B)
       ---------
         0 0 0 0   ← Corresponds to the rightmost 'zero' in B
   +     1 0 1 1   ← Corresponds to the next 'one' in B
   +   0 0 0 0
   + 1 0 1 1
   ---------------
   = 1 1 0 1 1 1 0
```

Division

0 / 0 = x
0 / 1 = 0
1 / 0 = x
1 / 1 = 1

Binary division is again similar to its decimal counterpart:
Here, the divisor is 101_2, or 5 decimal, while the dividend is 11011_2, or 27 decimal. The procedure is the same as that of decimal long division; here, the divisor 101_2 goes into the first three digits 110_2 of the dividend one time, so a "1" is written on the top line. This result is multiplied by the divisor, and subtracted from the first three digits of the dividend; the next digit (a "1") is included to obtain a new three-digit sequence

```
              1
          ___________
1 0 1    ) 1 1 0 1 1
         - 1 0 1
           -----
             0 1 1
```

The procedure is then repeated with the new sequence, continuing until the digits in the dividend have been exhausted:

```
              1 0 1
          ___________
1 0 1    ) 1 1 0 1 1
         - 1 0 1
           -----
             0 1 1
           - 0 0 0
             -----
               1 1 1
             - 1 0 1
               -----
                 1 0
```