Apostila de Organização de Computadores

Índice

1 NÚMEROS DE PRECISÃO FINITA

Ao se executarem operações aritméticas, geralmente se dá pouca importância à questão de quantos dígitos decimais são gastos para representar um número. Os físicos podem calcular que existem 10^{78} elétrons no universo sem se preocuparem com o fato de que são necessários 79 dígitos decimais para escrever o número completo. Ao se calcular o valor de uma função utilizando lápis e papel, necessitando de uma resposta com seis dígitos significativos, basta manter resultados intermediários com sete, oito dígitos ou quantos forem necessários. Nunca acontece de o papel não ser suficientemente grande para números de sete dígitos.

Com os computadores é bastante diferente. Na maioria dos computadores, a quantidade de memória disponível para armazenar um número é determinada no instante em que o computador é projetado. Com um certo esforço, o programador pode representar números com duas, três ou mesmo muitas vezes esta quantidade fixa, mas isto não muda a natureza deste problema. A natureza finita do computador nos força a lidar apenas com números que podem ser representados com um número fixo de dígitos. Chamamos tais números de números de precisão finita.

Para estudar as propriedades dos números de precisão finita, vamos examinar o conjunto dos inteiros positivos representáveis por três dígitos decimais, sem ponto decimal e sem sinal. Este conjunto tem exatamente 1000 elementos: 000, 001, 003, ..., 999. Com esta restrição, é impossível expressar vários conjuntos importantes de números, tais com:

1. Números maiores que 999.
2. Números negativos.
3. Números irracionais e fracionários.
4. Números complexos.

Podemos concluir que embora os computadores sejam dispositivos de uso geral, sua natureza finita os torna inadequados para operações aritméticas. Esta conclusão, naturalmente, não é verdadeira porque ela serve para ilustrar a importância de entender como os computadores trabalham e quais são as suas limitações.

2 SISTEMA DE NUMERAÇÃO

Os sistemas de numeração tem por objetivo prover símbolos e convenções para representar quantidades, de forma a registrar a informação quantitativa e poder processá-la. A representação de quantidades se faz com os **números**. Na antigüidade, duas formas de representar quantidades foram inventadas. Inicialmente, os egípcios, criaram um sistema em que cada dezena era representada por um símbolo diferente. Usando por exemplo os símbolos # para representar uma centena, & para representar uma dezena e @ representando uma unidade (símbolos escolhidos ao acaso), teríamos que ###&&@ representaria 321.

Relembremos ainda um outro sistema, o sistema de numeração romano. Eram usados símbolos (letras) que representavam as quantidades, como por exemplo: I (valendo 1), V (valendo 5), X (valendo 10), C (valendo 100), etc. A regra de posicionamento determinava que as letras que representavam quantidades menores e precediam as que representavam quantidades maiores, seriam somadas; se o inverso ocorresse, o menor valor era subtraído do maior (e não somado). Assim, a quantidade 128 era representada por CXXVIII = 100 + 10 + 10 + 5 + 1 + 1 + 1 = 128. Por outro lado, a quantidade 94 era representada por XCIV = (-10 + 100) + (-1 + 5) = 94.

Nesses sistemas, os símbolos tinham um valor intrínseco, independente da posição que ocupavam na representação (sistema numérico não-posicional). Um grande problema desse sistema é a dificuldade de realizar operações com essa representação. Experimente multiplicar CXXVIII por XCIV! Assim, posteriormente foram criados sistemas em que a posição dos algarismos no número passou a alterar seu valor (sistemas de numeração posicionais).

2.1 Base de um Sistema de Numeração

2.1.1 Sistemas de Numeração Posicionais

Nos sistemas de numeração posicionais, o valor representado pelo algarismo no número depende da posição em que ele aparece na representação. O primeiro sistema desse tipo foi inventado pelos chineses. Eram usados palitos, sendo 1 a 5 pali-

tos dispostos na vertical para representar os números 1 a 5; de 6 a 9 eram representados por 1 a 4 palitos na vertical, mais um palito na horizontal (valendo 5) sobre os demais. Cada número era então representado por uma pilha de palitos, sendo uma pilha de palitos para as unidades, outra para as dezenas, outra para as centenas, etc. Esse sistema, com as pilhas de palitos dispostas em um tabuleiro, permitia a realização das quatro operações aritméticas. Não existia representação para o zero (o espaço relativo ficava vazio). O tabuleiro aritmético (chamado *swan-pan*), além das quatro operações, era usado na álgebra e na solução de equações. Essa técnica era chamada de Método do Elemento Celestial.

O Alfabeto e o Ábaco

No Oriente Médio, por esses tempos, criou-se uma das mais importantes invenções da humanidade: o **alfabeto**. Na antigüidade, usava-se um símbolo para representar cada conceito ou palavra. Assim, eram necessários milhares de símbolos para representar todos os objetos, ações, sentimentos, etc - como são ainda hoje algumas linguagens. Como decorar todos? O grande achado foi decompor a linguagem em alguns poucos símbolos e regras básicas. Uma conseqüência de fundamental importância para nossos estudos de informática foi possibilitar a **ordenação alfabética** (essa é uma tarefa típica dos computadores). Nessa época, foi também criado o **ábaco** - uma calculadora decimal manual.

Os Algarismos e o Zero

Por volta do ano de 650, os hindus inventaram um método de produzir papel (que antes já fora inventado pelos chineses) e seus matemáticos criaram uma representação para os números em que existiam diferentes símbolos para os as unidades, incluindo um símbolo para representar o zero. Essa simples criação permitiu que se processasse a aritmética decimal e se fizesse contas - no papel! Bom, depois de milhares de anos em que todos os cálculos eram feitos com calculadoras (ábacos, swan-pan, etc) finalmente era possível calcular sem auxílio mecânico, usando um instrumento de escrita e papel. A matemática criada pelos hindus foi aprendida pelos árabes (que depois foram copiados pelos europeus). Por volta de 830, um matemático persa (chamado Al-khwarismi, que inspirou o nome algarismo) escreveu um livro

(Al-gebr we'l Mukabala, ou álgebra) em que apresentava os algarismos hindus. E esse livro, levado para a Europa e traduzido, foi a base da matemática do Renascimento.

Desde quando se começou a registrar informações sobre quantidades, foram criados diversos métodos de representar as quantidades.

O método ao qual estamos acostumados usa um sistema de numeração posicional. Isso significa que a posição ocupada por cada algarismo em um número altera seu valor de uma potência de 10 (na base 10) para cada casa à esquerda.

Por exemplo, no sistema decimal (base 10), no número 125 o algarismo 1 representa 100 (uma centena ou 10^2) , o 2 representa 20 (duas dezenas ou $1x10^1$) e o 5 representa 5 mesmo (5 unidades ou $5x10^0$). Assim, em nossa notação,
$125 = 1x10^2 + 2x10^1 + 5x10^0$

2.1.2 Base de um Sistema de Numeração

A **base** de um sistema é a quantidade de algarismos disponível na representação. A base 10 é hoje a mais usualmente empregada, embora não seja a única utilizada. No comércio pedimos uma dúzia de rosas ou uma grosa de parafusos (base 12) e também marcamos o tempo em minutos e segundos (base 60).

Os computadores utilizam a base 2 (sistema binário) e os programadores, por facilidade, usam em geral uma base que seja uma potência de 2, tal como 2^4 (base 16 ou sistema hexadecimal) ou eventualmente ainda 2^3 (base 8 ou sistema octal).

Na base 10, dispomos de 10 algarismos para a representação do número: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9. Na base 2, seriam apenas 2 algarismos: 0 e 1. Na base 16, seriam 16: os 10 algarismos aos quais estamos acostumados, mais os símbolos A, B, C, D, E e F, representando respectivamente 10, 11, 12, 13, 14 e 15 unidades.
A representação $125,38_{10}$ (base 10) significa $1x10^2+ 2x10^1+ 5x10^0 + 3x10^{-1}+ 8x10^{-2}$.
Generalizando, representamos uma quantidade **N** qualquer, numa dada base **b**, com um número tal como segue:

$N_b = a_n.b^n + ... + a_2.b^2 + a_1.b^1 + a_0.b^0 + a_{-1}.b^{-1} + a_{-2}.b^{-2} + ... + a_{-n}.b^{-n}$ sendo que
$a_n.b^n + ... + a_2.b^2 + a_1.b^1 + a_0.b^0$ é a parte inteira e
$a_{-1}.b^{-1} + a_{-2}.b^{-2} + ... + a_{-n}.b^{-n}$ é a parte fracionária.

Intuitivamente, sabemos que o maior número que podemos representar, com **n** algarismos, na base **b**, será o número composto **n** vezes pelo maior algarismo disponível naquela base (ou seja, b-1). Por exemplo, o maior número que pode ser representado na base 10 usando 3 algarismos será 999 (ou seja, $10^3 - 1 = 999$).

Podemos ver que o maior número inteiro **N** que pode ser representado, em uma dada base **b**, com **n** algarismos (**n** "casas"), será $N = b^n - 1$. Assim, o maior número de 2 algarismos na base 16 será FF_{16} que, na base 10, eqüivale a $255_{10} = 16^2 - 1$.

2.1.3 Representação Binária

Os computadores modernos utilizam apenas o sistema binário, isto é, todas as informações armazenadas ou processadas no computador usam apenas DUAS grandezas, representadas pelos algarismos 0 e 1. Essa decisão de projeto deve-se à maior facilidade de representação interna no computador, que é obtida através de dois diferentes níveis de tensão (ver em Bits & Bytes). Havendo apenas dois algarismos, portanto dígitos binários, o elemento mínimo de informação nos computadores foi apelidado de **bit** (uma contração do inglês ***b**inary dig**it***).

Bits & Bytes

Devido à simplicidade de projeto e construção, acarretando na redução de seu custo e maior confiabilidade, os circuitos eletrônicos que formam os computadores digitais atuais são capazes de distinguir apenas dois níveis de tensão - computadores digitais binários. Estes sinais elétricos são tensões que assumem dois diferentes valores: um valor positivo (hoje, nos PC's, cerca de +3 V - três volts positivos) para representar o valor binário 1 e um valor aproximado a 0 V (zero volt) para representar o valor binário 0. Na realidade, estes valores não são absolutos, e sim faixas de valores, com uma margem de tolerância (entre +2.5 e +3.5 V, representando o valor binário 1, e entre 0 e + 0,5 V representando o valor binário 0).

A lógica que permite aos computadores operar baseados nestes dois valores é chamada Álgebra de Boole, em homenagem ao matemático inglês George Boole (1815-1864).

Obs.: os primeiros computadores eram decimais (por exemplo, o ENIAC) e hoje existem também computadores analógicos (para determinadas aplicações específicas).

BIT é uma contração de BInary DigiT e representa um dos valores possíveis em binário, 0 ou 1.

BYTE é um grupo de 8 bits (é bom lembrar que $2^3 = 8$). Em um byte, há $2^8 = 256$ combinações, portanto pode-se representar 256 diferentes valores, desde 00000000 até 11111111. O termo "byte" foi inventado pela IBM.

Em informática, a expressão kilo (abreviada por k) eqüivale a 2^{10}, ou seja 1024. Desta forma, 1 kb eqüivale a 2^{10} bits, ou seja 1024 bits e kilobyte (1 kB) eqüivale a 2^{10} bytes, ou seja 1024 bytes ou ainda 8.192 bits.

Da mesma forma, a expressão mega eqüivale a 2^{20}, ou seja $2^{10} \times 2^{10} = 1.048.576$. Desta forma, 1 megabit (1 Mb) eqüivale a 2^{20} bits, ou seja 1024 kb ou 1.048.576 bits e 1 megabyte eqüivale a 2^{20} bytes, ou seja 1.048.576 bytes.

Seguem-se 1 giga, equivalente a 2^{30} ou 1024 megas, 1 tera, equivalente a 2^{40} ou 1.024 gigas, 1 peta, equivalente a 2^{50} ou 1.024 teras. É bom decorar estes termos, e seus valores, como quem decora uma tabuada. Vamos usar muito isso daqui por diante.

Na base 2, o número "10" vale dois. Mas se $10_2 = 2_{10}$, então dez é igual a dois?
Não, dez não é e nunca será igual a dois!

Na realidade, "10" não significa necessariamente "dez". Nós estamos acostumados a associar "10" a "dez" porque estamos acostumados a usar o sistema de numeração decimal.
O número 10_2 seria lido "um-zero" na base 2 e vale 2_{10} (convertido para "dois" na

base dez), 10_5 seria lido "um-zero" na base 5 e vale 5_{10} (convertido para "cinco" na base dez), 10_{10} pode ser lido como "um-zero" na base 10 ou então como "dez" na base dez, 10_{16} seria lido "um-zero" na base 16 e vale 16_{10} (convertido para "dezesseis" na base dez), etc.

Portanto, 10 só será igual a dez se - e somente se - o número estiver representado na base dez!

Uma curiosidade: o número "10_b" vale sempre igual à base, porque em uma dada base **b** os algarismos possíveis vão sempre de 0 a (b - 1)! Como o maior algarismo possível em uma dada base **b** é igual a (b-1), o próximo número será (b - 1 + 1 = b) e portanto será sempre 10 e assim, numa dada base qualquer, o valor da base será sempre representado por "10"!

Obs.: Toda vez que um número for apresentado sem que seja indicado em qual sistema de numeração ele está representado, estenderemos que a base é dez. Sempre que outra base for utilizada, a base será obrigatoriamente indicada.
Um dia pode ser que os computadores se tornem obrigatórios e sejamos todos forçados por lei a estudar a aritmética em binário! Mas, mesmo antes disso, quem programa computadores precisa conhecer a representação em binário!
Vamos começar *entendendo* as potências de dois:

Repr.Binária	Potência	Repr.Decimal
1	2^0	1
10	2^1	2
100	2^2	4
1000	2^3	8
10000	2^4	16
100000	2^5	32
1000000	2^6	64
10000000	2^7	128

100000000	2^8	256
1000000000	2^9	512
10000000000	2^{10}	1.024

Depois (e só depois) de compreender bem a tabela acima, fazendo a devida correlação com a representação decimal, é conveniente decorar os valores da tabela. As conversões entre base 2 e base 10 e as potências de 2 são utilizadas a todo momento e seria perda de tempo estar toda hora convertendo. Da mesma forma que, uma vez entendido o mecanismo da multiplicação, decoramos a tabuada, é muito mais efetivo saber de cor a tabela acima que fazer as contas de conversão toda vez que for necessário.

A representação binária é perfeitamente adequada para utilização pelos computadores. No entanto, um número representado em binário apresenta muitos bits, ficando longo e passível de erros quando manipulado por seres humanos normais, como por exemplo, os programadores, analistas e engenheiros de sistemas. Para facilitar a visualização e manipulação por programadores de grandezas processadas em computadores, são usualmente adotadas as representações octal (base 8) e principalmente hexadecimal (base 16). Ressaltamos mais uma vez que o computador opera apenas na base 2 e as representações octal e hexadecimal não são usadas no computador, elas se destinam apenas à manipulação de grandezas pelos programadores.

2.1.4 Representação em Octal e em Hexadecimal

Em projetos de informática (trabalhos realizados pelos programadores, analistas e engenheiros de sistemas), é usual representar quantidades usando sistemas em potências do binário (octal e principalmente hexadecimal), para reduzir o número de algarismos da representação e consequentemente facilitar a compreensão da grandeza e evitar erros. No sistema octal (base 8), cada três bits são representados por apenas um algarismo octal (de 0 a 7). No sistema hexadecimal (base 16), cada quatro bits são representados por apenas um algarismo hexadecimal (de 0 a F).
A seguir, será apresentada uma tabela com os números em decimal e sua represen-

tação correspondente em binário, octal e hexadecimal:

Base 10	Base 2	Base 8	Base 16
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Nota: a base 16 ou sistema hexadecimal pode ser indicada também por um "H" ou "h" após o número; por exemplo: FFH significa que o número FF (ou 255 em decimal) está em hexadecimal. Não confundir o "H" ou "h" com mais um dígito, mesmo porque em hexadecimal só temos algarismos até "F" e portanto não existe um algarismo "H".

Exemplo: Como seria a representação do número 16_{10} em binário, octal e hexadecimal?

Solução: Seria respectivamente 10000_2, 20_8 e 10_{16}.

2.2 Conversões entre Bases

Vamos analisar agora as regras gerais para converter números entre duas bases quaisquer.

2.2.1 Conversões entre as bases 2, 8 e 16

As conversões mais simples são as que envolvem bases que são potências entre si. Vamos exemplificar com a conversão entre a base 2 e a base 8. Como $2^3 = 8$, separando os bits de um número binário em grupos de três bits (começando sempre da direita para a esquerda!) e convertendo cada grupo de três bits para seu equivalente em octal, teremos a representação do número em octal. Por exemplo:
$10101001_2 = 10.101.001_2$ (separando em grupos de 3, sempre começando da direita para a esquerda). Sabemos que $010_2 = 2_8$; $101_2 = 5_8$; $001_2 = 1_8$ portanto $10101001_2 = 251_8$

Se você ainda não sabe de cor, faça a conversão utilizando a regra geral. Vamos agora exemplificar com uma conversão entre as bases 2 e 16. Como $2^4 = 16$, basta separarmos em grupos de 4 bits (começando sempre da direita para a esquerda!) e converter. Por exemplo:

$11010101101_2 = 110.1010.1101_2$ (separando em grupos de 4 bits, sempre começando da direita para a esquerda). Sabemos que $110_2 = 6_{16}$; $1010_2 = A_{16}$; $1101_2 = D_{16}$; portanto $11010101101_2 = 6AD_{16}$

Vamos agora exercitar a conversão inversa. Quanto seria 3F5H (lembrar que o H está designando "hexadecimal") em octal? O método mais prático seria converter para binário e em seguida para octal.

$3F5H=11.1111.0101_2$ (convertendo cada dígito hexadecimal em 4 dígitos binários) =
$= 1.111.110.101_2$ (agrupando de três em três bits) =
$= 1765_8$ (convertendo cada grupo de três bits para seu valor equivalente em octal).

2.2.2 Conversão de Números em uma base b qualquer para a base 10

Vamos lembrar a expressão geral já apresentada:

$N_b = a_n.b^n + ... + a_2.b^2 + a_1.b^1 + a_0.b^0 + a_{-1}.b^{-1} + a_{-2}.b^{-2} + ... + a_{-n}.b^{-n}$

A melhor forma de fazer a conversão é usando essa expressão. Tomando como exemplo o número 101101_2, vamos calcular seu valor representado na base dez. Usando a expressão acima, fazemos:

$101101_2 = 1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$

Podemos fazer a conversão de números em qualquer base para a base 10 usando o algoritmo acima.

Exemplos:

a) Converter 4F5H para a base 10 .

Solução: Lembramos que o H significa que a representação é hexadecimal (base 16). Sabemos ainda que $F_{16}=15_{10}$. Então: $4x16^2 + 15x16^1 + 5x16^0 = 4x256 + 15x16 + 5 = 1024 + 240 + 5 = 1269_{10}$

b) Converter 3485_9 para a base 10.

Solução: $3x9^3 + 4x9^2 + 8x9^1 + 5x9^0 = 3x729 + 4x81 + 8x9 + 5 = 2187 + 324 + 72 + 5 = 2588_{10}$.

c) Converter $7G_{16}$ para a base 10.

Solução: Uma base **b** dispõe dos algarismos entre 0 e (b-1). Assim, a base 16 dispõe dos algarismos 0 a F e portanto o símbolo G não pertence à representação hexadecimal.

d) Converter $1001,01_2$ para a base 10.

Solução: $1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 + 0x2^{-1} + 1x2^{-2} = 8 + 0 + 0 + 1 + 0 + 0,25 = 9,25_{10}$

e) Converter $34,3_5$ para a base 10.

Solução: $3x5^1 + 4x5^0 + 3x5^{-1} = 15 + 4 + 0,6 = 19,6_{10}$

f) Converter $38,3_8$ para a base 10.

Solução: Uma base **b** dispõe dos algarismos entre 0 e (b-1). Assim, a base 8 dispõe

dos algarismos 0 a 7 e portanto o algarismo 8 não existe nessa base. A representação 38,3 não existe na base 8.

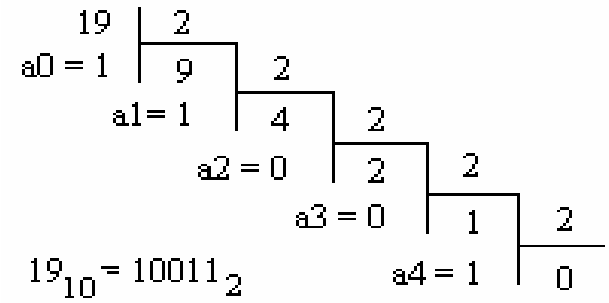
2.2.3 Conversão de Números da Base 10 para uma Base b qualquer

A conversão de números da base dez para uma base qualquer emprega algoritmos que serão o inverso dos acima apresentados. Os algoritmos serão melhor entendidos pelo exemplo que por uma descrição formal. Vamos a seguir apresentar os algoritmos para a parte inteira e para a parte fracionária:

Parte Inteira

O número decimal será dividido sucessivas vezes pela base; o resto de cada divisão ocupará sucessivamente as posições de ordem 0, 1, 2 e assim por diante até que o resto da última divisão (que resulta em quociente zero) ocupe a posição de mais alta ordem. Veja o exemplo da conversão do número 19_{10} para a base 2:

Conversão do número 19 para a base 2

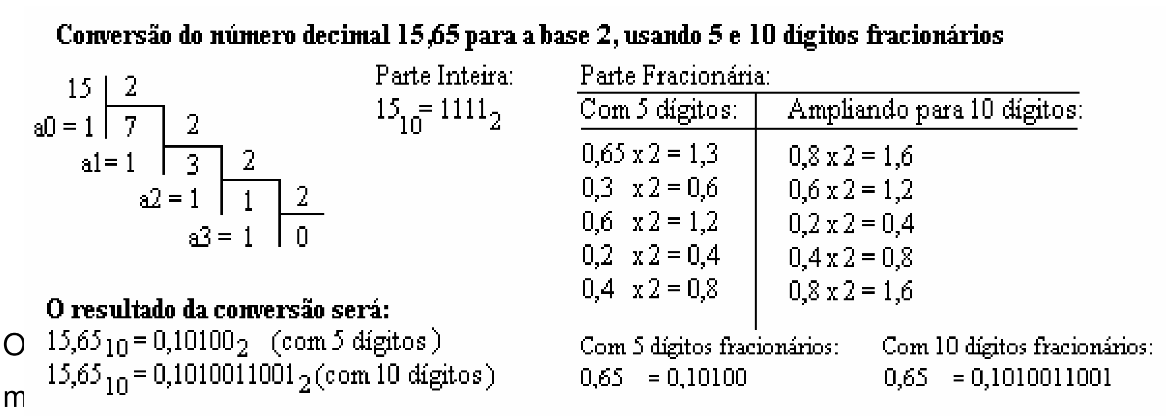


Experimente fazer a conversão contrária (retornar para a base 10) e ver se o resultado está correto.

Parte Fracionária

Se o número for fracionário, a conversão se fará em duas etapas distintas: primeiro a parte inteira e depois a parte fracionária. Os algoritmos de conversão são diferentes. O algoritmo para a parte fracionária consiste de uma série de multiplicações sucessivas do número fracionário a ser convertido pela base; a parte inteira do resultado da primeira multiplicação será o valor da primeira casa fracionária e a parte fracionária será de novo multiplicada pela base; e assim por diante, até o resultado dar zero ou até encontrarmos o número de casas decimais desejado. Por exemplo, vamos

converter $15{,}65_{10}$ para a base 2, com 5 e com 10 algarismos fracionários:



O

m

contramos resultado 0 em nenhuma das multiplicações, poderíamos continuar efetuando multiplicações indefinidamente até encontrar (se encontrarmos) resultado zero. No caso de interrupção por chegarmos ao número de dígitos especificado sem encontramos resultado zero, o resultado encontrado é aproximado e essa aproximação será função do número de algarismos que calcularmos. Fazendo a conversão inversa, encontraremos:

Com 5 algarismos fracionários:

Parte inteira: $1111_2 = 15_{10}$

Parte fracionária: $0{,}10100_2 = 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 0x2^{-5} = 0{,}5 + 0{,}125 = 0{,}625_{10}$

Com 10 algarismos fracionários:

Parte inteira: $1111_2 = 15_{10}$

Parte fracionária: $0{,}1010011001_2 = 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 0x2^{-5} + 1x2^{-6} + 1x2^{-7} + 0x2^{-8} + 0x2^{-9} + 1x2^{-10} = 1/2 + 1/8 + 1/64 + 1/128 + 1/1024 = 0{,}5 + 0{,}125 + 0{,}015625 + 0{,}0078125 + 0{,}0009765625 = 0{,}6494140625_{10}$

Ou seja, podemos verificar (sem nenhuma surpresa) que, quanto maior número de algarismos forem considerados, melhor será a aproximação.

2.2.4 <u>Conversão de Números entre duas Bases quaisquer</u>

Para converter números de uma base **b** para uma outra base **b'** quaisquer (isso é, que não sejam os casos particulares anteriormente estudados), o processo prático utilizado é converter da base **b** dada para a base 10 e depois da base 10 para a base **b'** pedida.

Exemplo: Converter 43_5 para $(\ )_9$.
$43_5 = (4 \times 5 + 3)_{10} = 23_{10}$ ==> 23/9 = 2 (resto 5) logo $43_5 = 23_{10} = 25_9$

2.3 Aritmética em Binário

2.3.1 Soma

A tabuada da soma aritmética em binário é muito simples. São poucas regras:
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (e "vai 1" para o dígito de ordem superior)
1 + 1 + 1 = 1 (e "vai 1" para o dígito de ordem superior)

Exemplo:
Efetuar 011100 + 011010
Obs.: 1) Lembre-se: soma-se as colunas da direita para a esquerda, tal como uma soma em decimal.

Obs.: 2) No exemplo, são usadas, em seqüência, da direita para a esquerda, todas as regrinhas acima.

Obs.: 3) Na primeira linha, em azul, é indicado o "vai um".

Obs.: 4) Por simplicidade, no exemplo estamos considerando os dois números positivos.

Solução:

11-----> "vai um"

011100

011010+

110110

2.3.2 Subtração

Vamos ver agora a tabuada da subtração:

0 - 0 = 0

0 - 1 = 1 ("vem um do próximo")

1 - 0 = 1

1 - 1 = 0

Obs.: Como é impossível tirar 1 de zero, o artifício é "pedir emprestado" 1 da casa de ordem superior. Ou seja, na realidade o que se faz é subtrair 1 de 10 e encontramos 1 como resultado, devendo então subtrair 1 do dígito de ordem superior (aquele 1 que se "pediu emprestado").

Exemplo:

Efetuar 111100 + 011010

Obs.: 1) Lembre-se: subtrai-se as colunas da direita para a esquerda, tal como uma subtração em decimal.

Obs.: 2) No exemplo, são usadas, em seqüência, da direita para a esquerda, todas as regrinhas acima.

Obs.: 3) Na primeira linha, em vermelho, é indicado o "vem um".

Obs.: 4) Por simplicidade, no exemplo estamos considerando os dois números positivos.

Solução:

---02-> "vem um"

11100

01010-

10010

2.3.3 Complemento a Base

A implementação do algoritmo da subtração em computadores é complexa, requerendo vários testes. Assim, em computadores a subtração em binário é feita por um artifício. O método utilizado é o *"Método do Complemento a Base"* que consiste em encontrar o complemento do número em relação à base e depois somar os números.

Os computadores funcionam sempre na base 2, portanto o complemento à base será complemento a dois. Computadores encontram o complemento a dois de um número através de um algoritmo que pode ser assim descrito:

se o número é positivo, mantenha o número (o complemento de um número positivo é o próprio número)

- se o número é negativo:
---inverta o número negativo ou o subtraendo na subtração (todo 1 vira zero, todo zero vira um)
--- some 1 ao número em complemento
--- some as parcelas (na subtração, some o minuendo ao subtraendo)
--- se a soma em complemento acarretar "vai-um" ao resultado, ignore o transporte final)

Como exemplo, vamos usar o algoritmo acima na subtração abaixo:

1101

1100-

0001

mantém o minuendo	--->	1101
inverte o subtraendo	--->	0011
soma minuendo e subtraendo	--->	10000
soma 1	--->	10001
ignora o "vai-um"	--->	0001

2.3.4 Multiplicação e Divisão

Vamos ver agora a tabuada da multiplicação:

0 x 0 = 0

0 x 1 = 0

1 x 0 = 0

1 x 1 = 1

No entanto, também a multiplicação em computadores é feita por um artifício: para multiplicar um número A por n, basta somar A com A, n vezes. Por exemplo, 4 x 3 = 4 + 4 + 4. E a divisão também pode ser feita por subtrações sucessivas! O que concluímos? Que qualquer operação aritmética pode ser realizada em computadores apenas através de somas (diretas ou em complemento

Uns exercícios um pouco diferente, para desenvolver o raciocínio:

a) Durante uma exploração, a arqueóloga Lar Acroft encontrou numa escavação uma pedra gravada com os seguintes caracteres:

%@#%

&

%&#&%

Concluindo brilhantemente (e com uma boa dose de adivinhação) que os símbolos correspondiam a uma operação de adição entre dois números positivos e que todos os algarismos usados pela antiga civilização estão presentes na gravação, determine a base de numeração utilizada, o algarismo arábico correspondente a cada símbolo e a representação das parcelas e do resultado da adição, convertidas para a base 10.

b) O Sr. M. recebeu certo dia um e-mail de seu agente Jaime Bonde, que estava em missão. O e-mail continha apenas o seguinte texto:

SEND
MORE
MONEY

Concluindo (também) brilhantemente (e também com uma boa dose de adivinhação) que os símbolos correspondiam a uma operação de adição entre dois números positivos representados em decimal (Jaime NÃO era forte em informática!), o Sr. M. raciocinou e então enviou ao agente uma determinada quantia. Quanto o Sr. M. enviou para seu agente J. Bonde?

2.3.5 Representação de Números em Complemento

Complemento é a diferença entre cada algarismo do número e o maior algarismo possível na base. Uma vantagem da utilização da representação em complemento é que a subtração entre dois números pode ser substituída pela sua soma em complemento.

2.3.5.1 Representação de Números Positivos em Complemento

A representação de números positivos em complemento não tem qualquer alteração, isto é, é idêntica à representação em sinal e magnitude.

2.3.5.2 Representação de Números Negativos em Complemento A (Base - 1)

A representação dos números inteiros negativos é obtida efetuando-se: (base - 1) menos cada algarismo do número. Fica mais fácil entender através de exemplos:

Ex.1: Calcular o complemento a (base - 1) do número 297_{10}.
Se a base é 10, então 10-1 = 9 e o complemento a (base -1) será igual a complemento a 9

Ex.2: Calcular o complemento a (base - 1) do número 3A7EH.
Se a base é 16, então 10H-1 = F e o complemento a (base -1) será igual a complemento a F.

Portanto:

	Ex.1	**Ex.2**
(base -1) --->	999	FFFF
-	297	3A7E
Complemento --->	702	C581

Caso Particular: Números na Base 2 (Complemento a 1)
Para se obter o complemento a 1 de um número binário, devemos subtrair cada algarismo de 1. Uma particularidade dos números binários é que, para efetuar esta operação, basta *inverter todos os bits*.

Como exemplo, vamos calcular o complemento a 1 (C1) de um número binário 0011 com 4 dígitos.

```
 1111
-0011
 1100 (C1)
```

Portanto, bastaria inverter todos os bits!

Vamos analisar como ficaria a representação em C1 dos números binários de 4 dígitos:

Decimal (positivo)	Binário (se o número é positivo, não há alteração)	Decimal (negativo)	Binário (em C1)
0	0000	0	1111
1	0001	-1	1110
2	0010	-2	1101
3	0011	-3	1100
4	0100	-4	1011
5	0101	-5	1010
6	0110	-6	1001
7	0111	-7	1000

3 CIRCUITOS LÓGICOS

3.1.1 Conceitos de Lógica Digital

Todos as complexas operações de um computador digital acabam sendo combinações de simples operações aritméticas e lógicas básicas: somar bits, complementar bits (para fazer subtrações), comparar bits, mover bits. Estas operações são fisicamente realizadas por circuitos eletrônicos, chamados circuitos lógicos (ou *gates* - "portas" lógicas).

Computadores digitais (binários) são construídos com circuitos eletrônicos digitais - as portas lógicas (circuitos lógicos).

Os sistemas lógicos são estudados pela álgebra de chaveamentos, um ramo da álgebra moderna ou álgebra de Boole, conceituada pelo matemático inglês George Boole (1815 - 1864). Boole construiu sua lógica a partir de símbolos, representando as expressões por letras e ligando-as através de conectivos - símbolos algébricos.

A álgebra de Boole trabalha com apenas duas grandezas: falso ou verdadeiro. As duas grandezas são representadas por 0 (falso) e 1 (verdadeiro).

Nota: nos circuitos lógicos do computador, os sinais binários são representados por níveis de tensão.

3.1.2 Operadores Lógicos

Os conectivos ou OPERADORES LÓGICOS ou FUNÇÕES LÓGICAS são:

3.1.2.1 E (ou AND)

Uma sentença é verdadeira SE - e somente se - todos os termos forem verdadeiros.

3.1.2.2 OU (ou OR)

Uma sentença resulta verdadeira se QUALQUER UM dos termos for verdadeiro.

3.1.2.3 NÃO (ou NOT)

Este operador INVERTE um termo.
Os operadores lógicos são representados por:

____

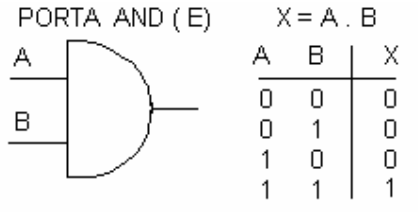
NOT --> (uma barra horizontal sobre o termo a ser invertido ou negado).
E ------> **.** (um ponto, como se fosse uma multiplicação)
OU ----> **+** (o sinal de soma)

3.1.3 Tabela Verdade

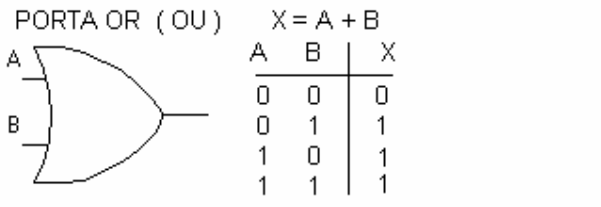
São tabelas que representam todas as possíveis combinações das variáveis de entrada de uma função, e os seus respectivos valores de saída.

A seguir, apresentamos as funções básicas, e suas representações em tabelas-verdade.

3.1.3.1 AND - Função E

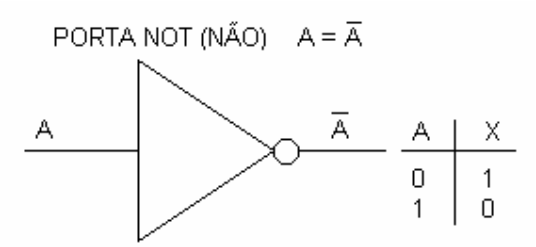


3.1.3.2 OR - Função OU



Nota: A menos da estranha expressão 1 + 1 = 1, as demais expressões "parecem" a aritmética comum a que estamos acostumados, onde E substitui "vezes" e OU substitui "mais".

3.1.3.3 NOT - Função NÃO



Obs.: a inversão em binário funciona como se fizéssemos 1 - A = X. Ou seja, 1 - 0 = 1 e 1 - 1 = 0.

3.1.4 Aplicação da Álgebra de Boole aos Computadores Digitais

Boole desenvolveu sua álgebra a partir desses conceitos básicos e utilizando apenas os algarismos 0 e 1.

Os primeiros computadores fabricados, como o ENIAC, trabalhavam em DECIMAL. No entanto, a utilização de circuitos eletrônicos que operassem com 10 diferentes níveis de tensão (para possibilitar detectar as 10 diferentes grandezas representadas no sistema decimal) acarretavam uma grande complexidade ao projeto e construção dos computadores, tendo por conseqüência um custo muito elevado. Surgiu então a idéia de aplicar a álgebra de Boole, simplificando extremamente o projeto e construção dos computadores.

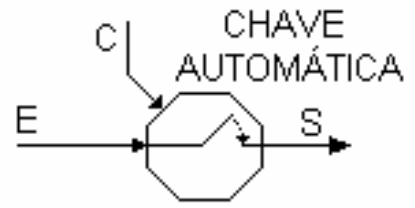
Mas como os conceitos da álgebra de chaveamentos (um ramo da álgebra do Boole) são aplicados ao projeto dos computadores digitais?

A chave de tudo é um circuito eletrônico chamado CHAVE AUTOMÁTICA.

Como funciona uma chave automática?

Vamos imaginar um circuito chaveador com as seguintes entradas:

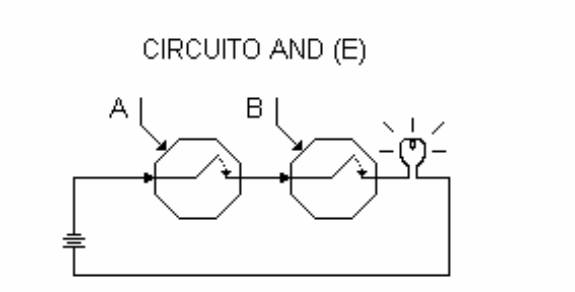
- uma fonte de alimentação (fornece energia para o circuito)
- um fio de controle (comanda a operação do circuito)
- um fio de saída (conduz o resultado)



No desenho, a chave permanece aberta enquanto o sinal C no fio de controle for 0 (ou Falso). Enquanto não houver um sinal (sinal 1 ou Verdadeiro) no fio de controle, que mude a posição da chave, o sinal no fio de saída S será 0 (ou Falso). Quando for aplicado um sinal (sinal 1 ou Verdadeiro) ao fio de controle, a chave muda de posição, tendo como resultado que o sinal na saída será então 1 (ou Verdadeiro). A posição da chave se manterá enquanto não ocorrer um novo sinal na entrada.

A chave automática foi inicialmente implementada com relês eletromecânicos e depois com válvulas eletrônicas. A partir da metade da década de 50, passaram a ser utilizados dispositivos em estado sólido - os TRANSISTORES, inventados em Stanford em 1947. Os modernos Circuitos Integrados - CI's e os microprocessadores são implementados com milhões de transistores "impressos" em minúsculas pastilhas.

Vamos agora analisar o que ocorreria se nós ligássemos em SÉRIE duas chaves automáticas como as acima, e ligássemos uma lâmpada ao circuito. O circuito resultante poderia ser representado assim:



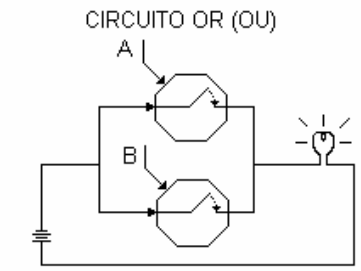
A lâmpada acenderia se - e somente se - as DUAS chaves estivessem na posição LIGADO (ou verdadeiro), o que seria conseguido com as duas entradas A e B em estado 1 (Verdadeiro). Substituindo CORRENTE (ou chave ligada) por 1 e AUSÊNCIA DE CORRENTE (ou chave desligada) por 0, como ficaria nossa tabela verdade para LÂMPADA LIGADA = 1 e LÂMPADA DESLIGADA = 0?

A	B	L
0	0	0
0	1	0
1	0	0
1	1	1

Dá para reconhecer a nossa já familiar FUNÇÃO E?

O circuito acima que implementa a função E é chamado de PORTA E (*AND GATE*).

Vamos agora analisar o que ocorreria se nós ligássemos em PARALELO duas chaves automáticas como as acima, e ligássemos uma lâmpada ao circuito. O circuito resultante poderia ser representado assim:



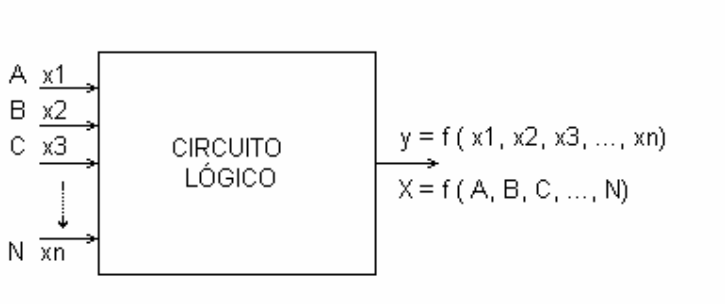
A lâmpada acenderia SE QUALQUER UMA DAS-CHAVES estivesse na posição LIGADO (ou verdadeiro), o que seria conseguido com uma das duas entradas A ou B em estado 1 (Verdadeiro). Substituindo CORRENTE (ou chave ligada) por 1 e AUSÊNCIA DE CORRENTE (ou chave desligada) por 0, como ficaria nossa tabela verdade para LÂMPADA LIGADA = 1 e LÂMPADA DESLIGADA = 0?

A	B	L
0	0	0
0	1	1
1	0	1
1	1	1

E agora, dá para reconhecer a nossa já familiar FUNÇÃO OU?
O circuito acima, que implementa a função OU, é chamado de PORTA OU (*OR GATE*).

3.1.5 Porta Lógica ou Gate

São dispositivos ou circuitos lógicos que operam um ou mais sinais lógicos de entrada para produzir uma (e somente uma) saída, a qual é dependente da função implementada no circuito.



Um computador é constituído de uma infinidade de circuitos lógicos, que executam as seguintes funções básicas:

a) realizam operações matemáticas
b) controlam o fluxo dos sinais
c) armazenam dados

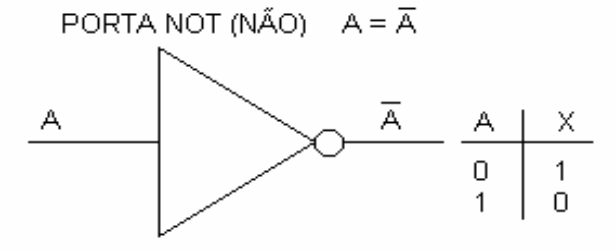
Existem dois tipos de circuitos lógicos:

a) **COMBINACIONAL** - a saída é função dos valores de entrada correntes; esses circuitos não tem capacidade de armazenamento [casos a) e b) acima].
b) **SEQUENCIAL** - a saída é função dos valores de entrada correntes e dos valores de entrada no instante anterior; é usada para a construção de circuitos de memória (chamados "flip-flops").

3.2 Portas Lógicas

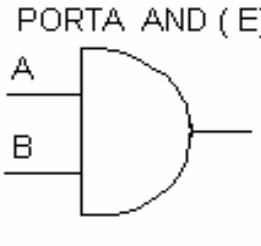
3.2.1 Porta NOT (NÃO)

A porta NOT inverte o sinal de entrada (executa a NEGAÇÃO do sinal de entrada), ou seja, se o sinal de entrada for 0 ela produz uma saída 1, se a entrada for 1 ela produz uma saída 0.



3.2.2 Porta AND (E)

A porta AND combina dois ou mais sinais de entrada de forma equivalente a um circuito em série, para produzir um único sinal de saída, ou seja, ela produz uma saída 1, se todos os sinais de entrada forem ; caso qualquer um dos sinais de entrada for 0, a porta AND produzirá um sinal de saída igual a zero.

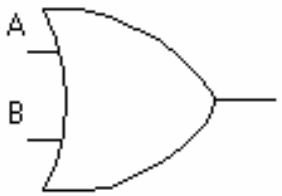


PORTA AND (E) X = A . B

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

3.2.3 Porta OR (OU)

A porta OR combina dois ou mais sinais de entrada de forma equivalente a um circuito em paralelo, para produzir um único sinal de saída, ou seja, ela produz uma saída 1, se qualquer um dos sinais de entrada for igual a 1; a porta OR produzirá um sinal de saída igual a zero apenas se todos os sinais de entrada forem 0.

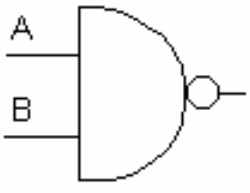


PORTA OR (OU) X = A + B

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

3.2.4 Porta NAND (NÃO E)

A porta NAND eqüivale a uma porta AND seguida por uma porta NOT, isto é, ela produz uma saída que é o inverso da saída produzida pela porta AND.

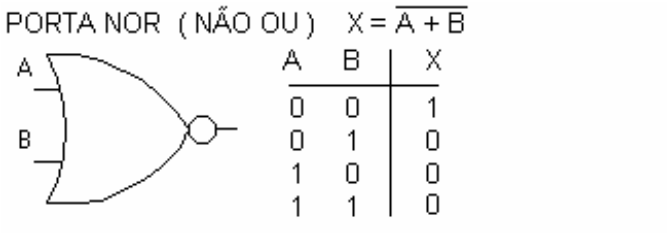


PORTA NAND (NÃO E) $X = \overline{A \cdot B}$

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

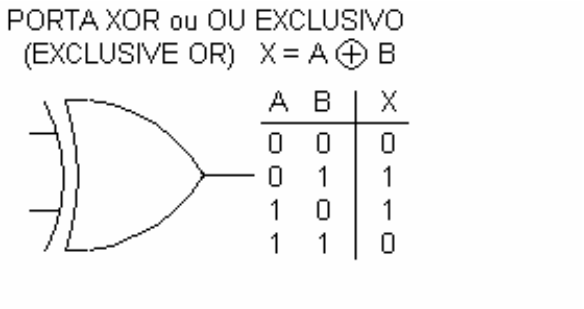
3.2.5 Porta NOR (NÃO OU)

A porta NOR eqüivale a uma porta OR seguida por uma porta NOT, isto é, ela produz uma saída que é o inverso da saída produzida pela porta OR.



3.2.6 Porta XOR (OU EXCLUSIVO)

A porta XOR compara os bits; ela produz saída 0 quando todos os bits de entrada são iguais e saída 1 quando pelo menos um dos bits de entrada é diferente dos demais.



Exemplo de circuitos utilizando portas lógicas:

A) Uma campainha que toca (saída) se o motorista der a partida no motor do carro (entrada) sem estar com o cinto de segurança afivelado (entrada). Se a ignição for ACIONADA (1) e o cinto estiver DESAFIVELADO (1), a campainha é ACIONADA (1). Caso contrário, a campainha não toca

Tabela Verdade:

Ignição	Cinto	Campainha
0	0	0
0	1	0
1	0	0
1	1	1

Basta incluir uma porta AND.

B) Detector de incêndio com vários sensores (entradas) e uma campainha para alarme (saída).

Se QUALQUER UM dos sensores for acionado (significando que um dos sensores detectou sinal de incêndio), a campainha é ACIONADA.

Tabela verdade:

Sensor 1	Sensor 2	Campainha
0	0	0
0	1	1
1	0	1
1	1	1

Basta incluir uma porta OR.

3.3 Algebra de Boole

As operações básicas da Álgebra de Boole são:

1. NOT ---> $X = \overline{A}$
2. AND ---> $X = A \cdot B$
3. OR ---> $X = A + B$
4. NAND ---> $X = \overline{A \cdot B}$
5. NOR ---> $X = \overline{A + B}$
6. XOR ---> $A \oplus B$

3.3.1 Avaliação de uma Expressão Booleana

Uma expressão booleana é uma expressão formada por sinais de entrada (chamados variáveis de entrada) ligados por conectivos lógicos, produzindo como resultado um único sinal de saída.

Na avaliação de uma expressão Booleana, deverá ser seguida uma ordem de precedência conforme a seguir definido:

1º - avalie NOT

2º - avalie AND

3º - avalie OR

Obs.: respeitando-se sempre os parênteses!

Ex.: Avalie a expressão:

$X = A \cdot B + \overline{C}$

A	B	C	$\overline{C}$	A.B	$X=A.B+\overline{C}$
0	0	0	1	0	1
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1

3.3.2 Equivalência de Funções Lógicas

Duas funções Booleanas são equivalentes se - e somente se - para a mesma entrada, produzirem iguais valores de saída .

PORTANTO, DUAS FUNÇÕES LÓGICAS EQUIVALENTES TEM A MESMA TABELA VERDADE.

Ex.: Verifique se as funções lógicas a seguir representam funções equivalentes:

a) $\overline{\overline{(\overline{X+Y})+(\overline{X \cdot Z})}}$ e $X \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$

b) $\overline{\overline{(\overline{X+Y})+(\overline{X \cdot Y})}}$ e $X \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$

c) $X \cdot Y + Z$ e $X \cdot (Y + Z)$

3.3.3 Propriedade da Álgebra de Boole

PROPRIEDADE	VERSÃO OR	VERSÃO AND
1 - IDENTIDADE	$X+0=X$	$X.1=X$
2 - ELEMENTO NULO	$X+1=1$	$X.0=0$
3 - EQUIVALÊNCIA	$X+\overline{X}=X$	$X.\overline{X}=X$
4 - COMPLEMENTO	$X+\overline{X}=1$	$X.\overline{X}=0$
5 - INVOLUÇÃO	$\overline{\overline{X}}=X$	$\overline{\overline{X}}=X$
6 - COMUTATIVA	$X+Y=Y+X$	$X.Y=Y.X$
7 - ASSOCIATIVA	$(X+Y)+Z=X+(Y+Z)$	$(X.Y).Z=X.(Y.Z)$
8 - DISTRIBUTIVA	$X+Y.Z=(X+Y).(X+Z)$	$X.(Y+Z)=X.Y+X.Z$
9 - ABSORÇÃO 1	$X+X.Y=X$	$X.(X+Y)=X$
10 - ABSORÇÃO 2	$X+\overline{X}.Y=X+Y$	$X.(\overline{X}+Y)=X.Y$
11 - CONSENSUS	$X.Y+\overline{X}.Z+Y.Z=X.Y+\overline{X}.Z$	$(X+Y).(\overline{X}+Z).(Y+Z)=(X+Y).(\overline{X}+Z)$
12 - DE MORGAN	$\overline{X+Y}=\overline{X}.\overline{Y}$	$\overline{X.Y}=\overline{X}+\overline{Y}$

Exercício:

Simplifique a seguinte expressão:

$$\overline{\overline{B+D}.(\overline{B.C}).(\overline{A}.B)}$$

SOLUÇÃO:

$$\overline{\overline{B+D}.(\overline{B.C}).(\overline{A}.B)} = \overline{\overline{B+D}} + \overline{\overline{B.C}} + \overline{\overline{A}.B} =$$

$$= B + D + B.C + \overline{\overline{A}} + \overline{B} = D + B.C + A + 1 = 1$$

3.3.4 Propriedades da Função XOR (EXCLUSIVE OR)

A	B	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Soma dos minitermos:

$A \oplus B = \overline{A}.B + A.\overline{B}$

$\overline{A \oplus B} = A.B + \overline{A}.\overline{B}$

Outras propriedades:

$A \oplus A = 0$

$A \oplus \overline{A} = 1$

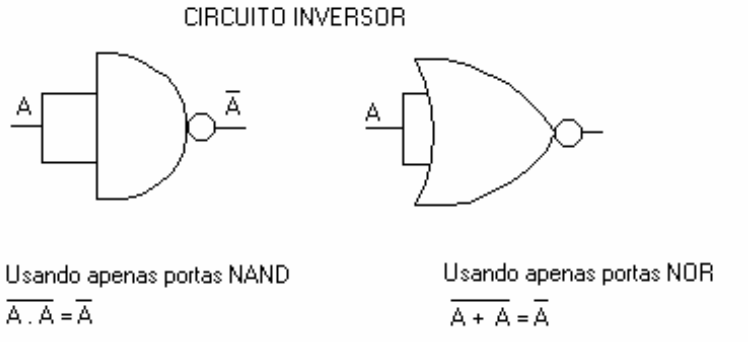
3.4 Representação de Circuitos com as Funções NAND e NOR

Usando as propriedades apresentadas, todo e qualquer circuito pode ser representado usando exclusivamente as função NAND ou as função NOR.

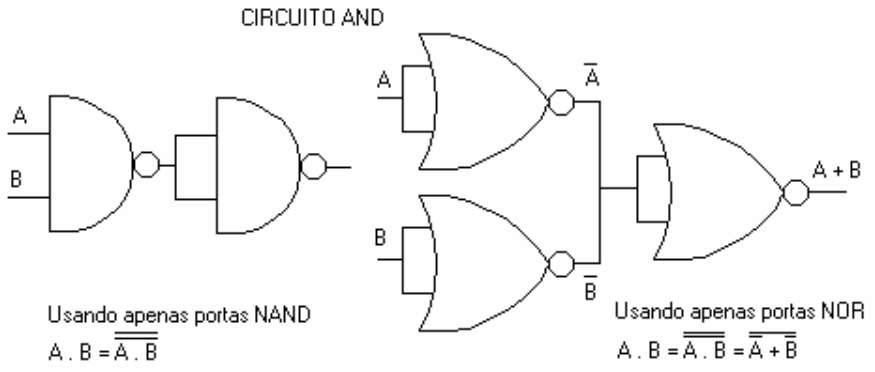
Para que serviria tal artimanha, além da dor de cabeça aos estudantes? Há neste

caso uma razão maior que a comodidade ou a aparente dificuldade: a razão econômica. Por diversas razões construtivas, fica mais barato construir TODOS os circuitos de um computador usando APENAS UM ÚNICO TIPO DE CIRCUITO. Aceitando essa afirmação, vamos enfrentar a tarefa de representar os nossos circuitos já conhecidos usando apenas funções NAND ou os NOR.

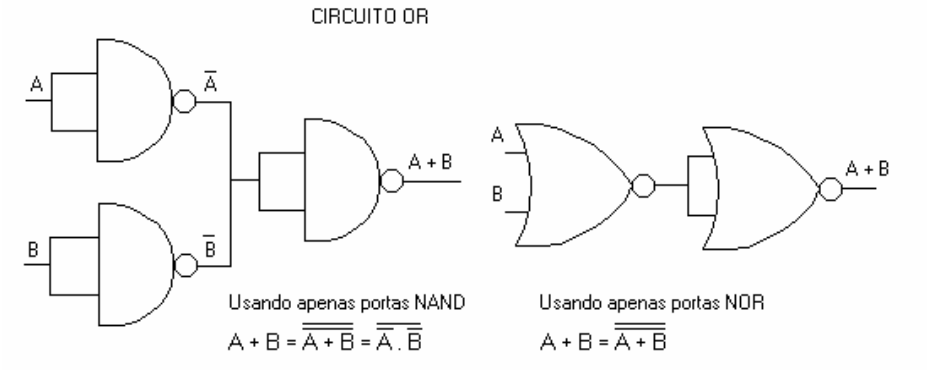
3.4.1 Circuito Inversor



3.4.2 Circuito AND

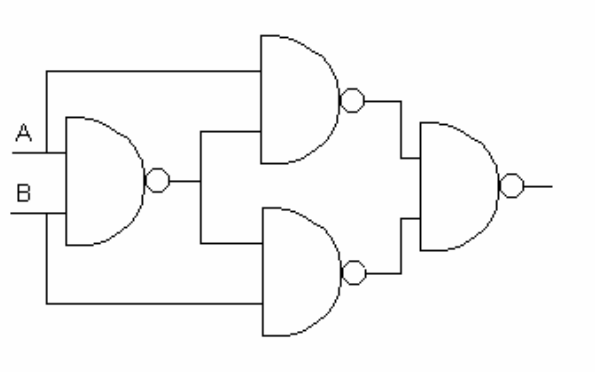


3.4.3 Circuito OR



Exercício:

Escreva a expressão do circuito abaixo e simplifique.



Solução:

$$C = \overline{\overline{A \cdot \overline{AB}} \cdot \overline{B \cdot \overline{AB}}} = \overline{\overline{A \cdot \overline{AB}}} + \overline{\overline{B \cdot \overline{AB}}} = A \cdot \overline{AB} + B \cdot \overline{AB} =$$
$$= \overline{AB} \cdot (A + B) = (\overline{A} + \overline{B}) \cdot (A + B) = \cancel{\overline{A} \cdot A}^{0} + \overline{A} \cdot B + \overline{B} \cdot A + \cancel{\overline{B} \cdot B}^{0} =$$
$$= \overline{A} \cdot B + \overline{B} \cdot A = A \oplus B$$

Este circuito implementa a função XOR, usando apenas portas NAND!

3.5 Formas Canônicas

3.5.1 Representação de um Circuito através de uma Tabela Verdade

Os circuitos de um computador realizam funções de grande complexidade, cuja representação geralmente não é óbvia. O processo para realização de uma função através de um circuito começa na descrição verbal do circuito (descrição do comportamento de suas possíveis saídas, em função das diversas combinações possíveis de seus sinais de entrada), a partir do que é possível montar sua tabela verdade.

Exemplos:

a) Considere um circuito elétrico composto de uma fonte de energia comercial (a alimentação da empresa de distribuição de energia, p.ex., a Light) e um interruptor (nossas entradas) e uma lâmpada (nossa saída). A lâmpada acenderá se - e somente se - a) houver energia disponível (se não estiver "faltando luz") e b) o interruptor estiver ligado. Elabore a tabela verdade que representa esse circuito lógico.

b) Considere um sistema composto de duas caixas d'água (uma superior e uma cis-

terna). A cisterna é alimentada pela entrada de água da "rua", via empresa distribuidora (ex.: CEDAE). A caixa superior serve para distribuir a água, por gravidade, em todo o prédio: bicas, chuveiros, descargas sanitárias, circuitos anti-incêndio, etc, com a água sendo impulsionada por uma bomba hidráulica através de uma tubulação que liga a cisterna à caixa superior. Considerando que a bomba queimará se for acionada sem haver água no circuito hidráulico, projete um circuito lógico para acionar a bomba sempre que a caixa superior estiver vazia, desde que tenha água na cisterna.

c) Considere um circuito elétrico composto de uma fonte de energia comercial (a alimentação da empresa de distribuição de energia, p.ex., a Light), uma alimentação auxiliar (um gerador e um no-break, com bateria de acumulação) e um interruptor (nossas entradas) e um sistema de computadores (nossa saída). O computador poderá operar se: a) houver energia disponível (se não estiver "faltando luz") em um dos circuitos de alimentação e b) o interruptor estiver ligado. Elabore a tabela verdade que representa esse circuito lógico.

A partir da tabela verdade produzida conforme item anterior, é possível chegar à expressão que representa o comportamento do circuito, e em seguida construir o circuito, usando as portas lógicas já estudadas. O processo de elaboração da expressão usa as chamadas formas canônicas, que consistem em regras para representar as condições de entrada que:

a) produzirão saída 1 (e portanto as demais condições produzirão saída 0) ou alternativamente,
b) produzirão saída 0 (e portanto as demais condições produzirão saída 1).

São portanto duas as formas canônicas: uma representa as condições que produzem saída 1 (soma dos minitermos) , a outra representa as condições que produzirão saída 0 (produto dos maxitermos). Essas formas são alternativas, isto é, a expressão poderá ser encontrada aplicando-se alternativamente uma ou outra das formas.

MINITERMO - são termos somente com AND (termos **PRODUTO**)

MAXITERMO - são termos somente com OR (termos **SOMA**)

3.5.2 Soma dos Minitermos

É produzida construindo:
um termo (uma sub-expressão) para cada linha da tabela verdade (que representa uma combinação de valores de entrada) em que a saída é 1,
cada um desses termos é formado pelo PRODUTO (FUNÇÃO AND) das variáveis de entrada, sendo que:

- quando a variável for 1, mantenha;
- quando a variável for 0, complemente-a (função NOT).

a função booleana será obtida unindo-se os termos PRODUTO (ou minitermos) por uma porta OR (ou seja, "forçando-se" a saída 1 caso qualquer minitermo resulte no valor 1).

Dessa forma, ligando os termos-produto (também chamados minitermos) pela porta OR, caso qualquer um dos minitermos seja 1 (portanto, caso qualquer uma das condições de valores de entrada que produz saída 1se verifique), a saída pela porta OR será também 1. Ou seja, basta que se verifique qualquer uma das alternativas de valores de entrada expressos em um dos minitermos, e a saída será também 1, forçada pelo OR. Caso nenhuma dessas alternativas se verifique, produz-se a saída 0.

Exemplo:

A	B	C	f	MINITERMOS
0	0	0	0	
0	0	1	1	$\overline{A}.\overline{B}.C +$
0	1	0	1	$\overline{A}.B.\overline{C} +$
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	$A.B.\overline{C}$
1	1	1	0	

$$f = \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.B.\overline{C}$$

3.5.3 Produto dos Maxitermos

É produzida construindo:
um termo (uma sub-expressão) para cada linha da tabela verdade (que representa uma combinação de valores de entrada) em que a saída é 0,
cada um desses termos é formado pela SOMA (FUNÇÃO OR) das variáveis de entrada, sendo que:

- quando a variável for 0, mantenha;
- quando a variável for 1, complemente-a (função NOT).

A função booleana será obtida unindo-se os termos SOMA (ou maxitermos) por uma porta AND (ou seja, "forçando-se" a saída 0 caso qualquer minitermo resulte no valor 0).

Dessa forma, ligando os termos-soma (também chamados maxitermos) pela porta AND, caso qualquer um dos minitermos seja 0 (portanto, caso qualquer uma das condições de valores de entrada que produz saída 0 se verifique), a saída pela porta AND será também 0. Ou seja, basta que se verifique qualquer uma das alternativas de valores de entrada 0 expressos em um dos maxitermos, e a saída será também 0, forçada pelo AND. Caso nenhuma dessas alternativas se verifique, produz-se a saída 1.
Exemplo:

A	B	C	f	MAXITERMOS
0	0	0	1	
0	0	1	0	$A+B+\overline{C}$.
0	1	0	0	$A+\overline{B}+C$.
0	1	1	1	
1	0	0	0	$\overline{A}+B+C$.
1	0	1	1	
1	1	0	1	
1	1	1	1	

$f = A+B+\overline{C} \,.\, A+\overline{B}+C \,.\, \overline{A}+B+C$

Obs:. O mesmo comportamento (a mesma tabela verdade) pode ser igualmente representada por qualquer das formas canônicas.
Exemplo:

A	B	C	f	MINITERMOS	MAXITERMOS
0	0	0	1	$\overline{A}.\overline{B}.\overline{C}$ +	
0	0	1	0		$A+B+\overline{C}$.
0	1	0	0		$A+\overline{B}+C$.
0	1	1	1	$\overline{A}.B.C$ +	
1	0	0	0		$\overline{A}+B+C$.
1	0	1	1	$A.\overline{B}.C$ +	
1	1	0	0		$\overline{A}+\overline{B}+C$.

Soma dos minitermos
$f = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.C + A.\overline{B}.C + A.B.C$

Produto dos maxitermos
$f = (A+B+\overline{C}).(A+\overline{B}+C).(\overline{A}+B+C).(\overline{A}+\overline{B}+C)$

Se ambas as formas canônicas produzem expressões equivalentes, como escolher qual a representação a utilizar? Escolha a que resultar em menor número de termos, produzindo uma expressão mais simples.

Por esse método, pode-se encontrar a expressão que represente qualquer tabela verdade.
Após se encontrar uma expressão que represente o comportamento esperado, é possível que não seja uma expressão simples que possa ser construída com poucas portas lógicas. Antes de projetar o circuito, é útil SIMPLIFICAR a expressão, de forma a possibilitar construir um circuito mais simples e portanto mais barato.

Portanto, o fluxo de nosso procedimento será:

DESCRIÇÃO VERBAL ---> TABELA VERDADE ---> FORMA CANÔNICA ---> --->FUNÇÃO SIMPLIFICADA ---> CIRCUITO

3.6 Construção de Circuitos Reais de Computador

3.6.1 Circuitos Aritméticos

Vamos lembrar a aritmética de ponto fixo, para a soma de dois bits.

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 e vai um

Se houver vai um na soma anterior, teremos:
vem 1 + 1 + 1 = 1 e vai um

Lembram-se do algoritmo da soma? Para somar dois números (em qualquer base, binários ou não), soma-se os algarismos dos números, coluna a coluna, transportan-

do o "vai um" para a próxima coluna, quando for o caso.

Como fazemos para somar dois números de n algarismos?
Na prática, representamos os dois números a serem somados, um sobre o outro, e somamos coluna a coluna; quando o valor da soma de dois algarismos supera a base, somamos um à próxima coluna (isto é, fazemos resultado + base, sendo base igual a 10 para qualquer base). O "VAI UM" de uma coluna (uma ordem) é o "VEM UM" da próxima coluna. Fácil? Não? Se a explicação complicou, basta vermos um exemplo prático para concluir o quanto é simples:

NA BASE 10	NA BASE 2	
110	11010	"vem 1" (carry in)
085	01101	parcela
+16	01101	parcela
101	11010	soma
011	01101	"vai 1" (carry out)

Bom, difícil não pode ser, já que vimos fazemos somas desse jeito desde o curso primário.

E para fazer uma subtração? Lembrando também o estudado anteriormente.

UMA SUBTRAÇÃO É UMA SOMA EM COMPLEMENTO! Ou seja,

A - B = A + (- B)

No computador, fazemos a subtração através de uma soma em complemento.

E a multiplicação? A multiplicação é obtida de duas formas: por somas sucessivas (por exemplo, A + A = 2A) e pela movimentação de bits. Lembremo-nos que quando queremos multiplicar um número decimal por 10, basta acrescentar um zero à direita do número. Como nesse caso a base é 10, isso eqüivale a **MULTIPLICAR PELA**

BASE. Generalizando, para multiplicar um número pela base, basta acrescentar um zero à direita do número, ou seja, movemos todos os algarismo de um número para a esquerda de uma posição (uma ordem de grandeza), preenchendo a última ordem à direita (que ficaria vaga) com um zero. Isso, portanto, eqüivale a multiplicar esse número pela base.

Por exemplo, na base 10, tomando 50 e fazendo 500 (movendo 50 para a esquerda uma posição e colocando um zero para preencher a posição mais à direita) eqüivale a multiplicar 50 por 10 (a base)!

Idem na base 2:
100 x 10 = 1000

Atenção: devemos ler esse número como: um zero zero vezes um zero = um zero zero zero. Para ficar ainda mais claro, vamos lembrar que, passando para decimal, $100_{(2)}$ é $4_{(10)}$, $10_{(2)}$ é $2_{(10)}$ e portanto teríamos em decimal: 4 x 2 = 8). Lembrem-se: 10 só é dez em decimal! Agora, sempre que estivermos usando uma base diferente de 10, vamos sempre ler 10 como um-zero! E como base decimal não é usada em computadores digitais, DEZ, a partir de agora, fica banido de nosso vocabulário!

As mesmas propriedades, aplicadas no sentido contrário, valem para a divisão!

Desta forma, podemos ver que

O COMPUTADOR PODE REALIZAR TODAS AS OPERAÇÕES ARITMÉTICAS USANDO APENAS SOMAS!

Com esta introdução, podemos agora estudar alguns exemplos dos circuitos aritméticos usados em computadores.

3.6.2 Circuito Meio-Somador

O circuito meio-somador SOMA DOIS BITS (sem levar em conta bit de carry).

Entrada - os dois bits a serem somados - A e B

Saída - a soma dos bits e o bit de carry out ("vai um") - S e Co

Como descrevemos anteriormente, uma função lógica produz uma e apenas uma saída. Portanto, sendo duas as saídas, serão necessárias duas funções diferentes, ou um circuito composto, podendo haver interseção de portas lógicas.

a. Construir a tabela
b. Forma canônica
c. c. Simplificação (não há o que simplificar)

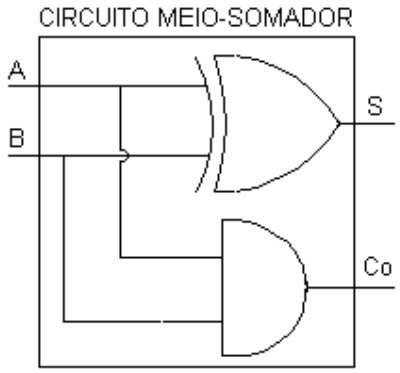
A	B	S	Co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Soma dos minitermos

$S = \bar{A}.B + A.\bar{B}$

$Co = A.B$

d. Circuito



3.6.3 Circuito Somador Completo

O circuito somador completo SOMA DOIS BITS (considerando na soma o bit de carry in que veio da soma anterior).

Entrada - os dois bits a serem somados e o bit de carry in - A, B e Ci

Saída - a soma dos bits e o bit de carry out ("vai um") - S e Co

a. Construir a tabela

b. Forma canônica

c. Simplificação

A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Função S (soma) - Soma dos minitermos

$S = \bar{A}.\bar{B}.Ci + \bar{A}.B.\overline{Ci} + A.\bar{B}.\overline{Ci} + A.B.Ci$

Simplificação

$S = Ci\,(A.\overline{B} + \overline{A}.B) + \overline{Ci}\,.\,(\bar{A}.B + A.\bar{B}) =$

$S = Ci\,.\,(A \oplus B) + \overline{Ci}\,.\,(A \oplus B)$

$S = Ci \oplus (A \oplus B)$

Função Co ("vai um" ou "carry out") - Soma dos minitermos

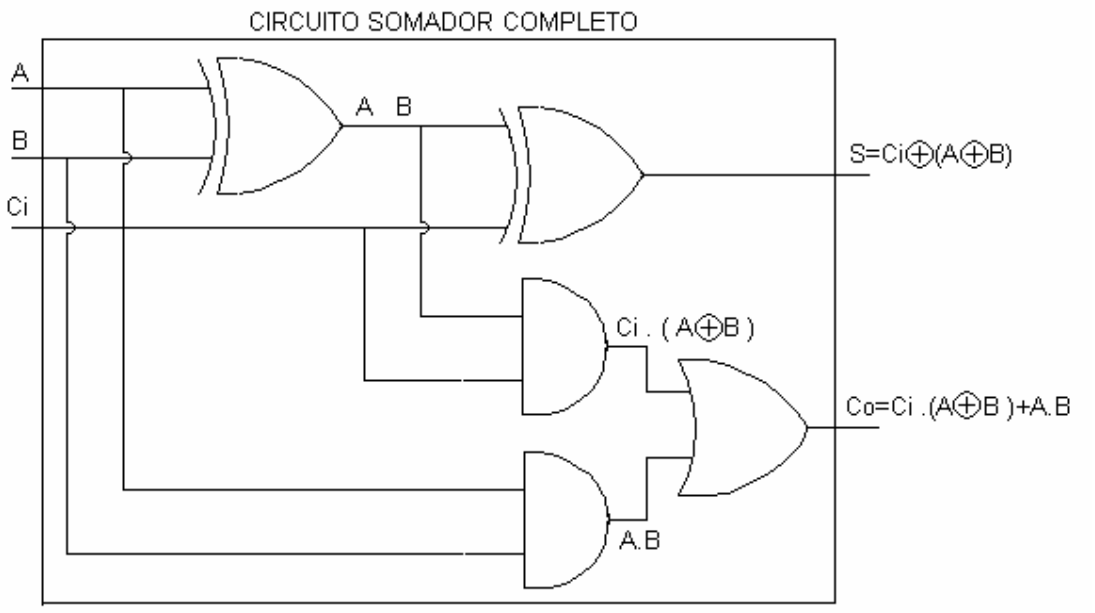
$Co = (\bar{A}.B.Ci) + (A.\bar{B}.Ci) + (A.B.\overline{Ci}) + (A.B.Ci)$

Simplificação

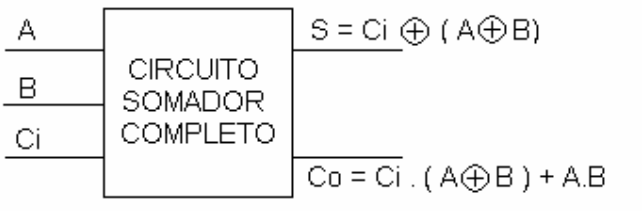
$Co = Ci\,.\,(\bar{A}.B + A.\bar{B}) + A.B\,.\,\underbrace{(\overline{Ci} + Ci)}_{1}$

$Co = Ci\,.\,(A \oplus B) + A.B$

d. Circuito



e. Representação esquemática



4 A ARQUITETURA DOS COMPUTADORES

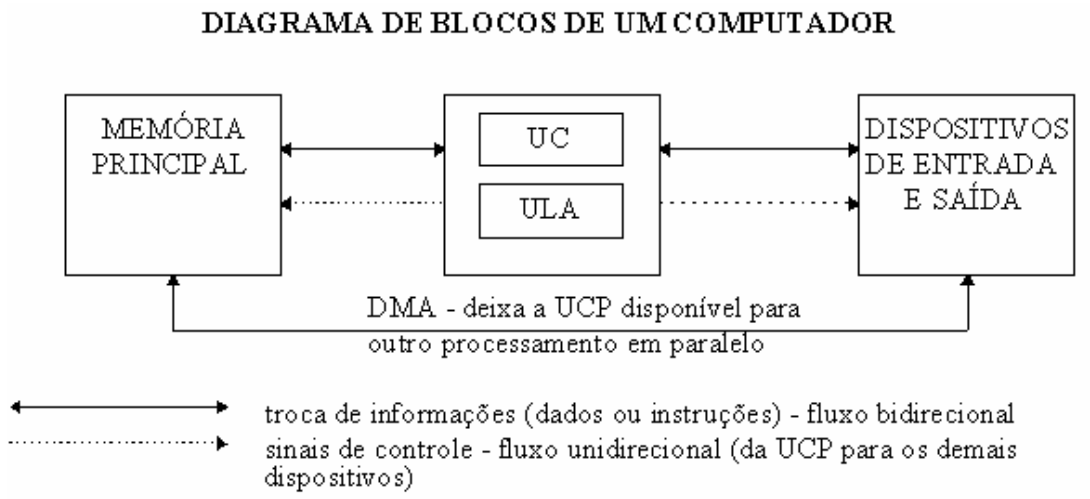
A arquitetura básica de um computador moderno segue ainda de forma geral os conceitos estabelecidos pelo Professor da Universidade de Princeton, John Von Neumann (1903-1957). Von Neumann propôs construir computadores que:

1. Codificassem instruções que pudessem ser armazenadas na memória e sugeriu que usassem cadeias de uns e zeros (binário) para codificá-los ;

2. Armazenassem na memória as instruções e todas as informações que fossem necessárias para a execução da tarefa desejada;
3. Ao processarem o programa, as instruções fossem buscadas na diretamente na memória.

Este é o conceito de **PROGRAMA ARMAZENADO**.

4.1 Diagrama de Blocos dos Computadores



Toda a lógica dos computadores é construída a partir de chaves liga / desliga. Inicialmente foram usados chaves mecânicas, depois relês eletro-mecânicos - o Z-1 construído por Konrad Zuse em 1941 e o MARK 1 de Howard Aiken em 1944 (capazes de executar até 5 chaveamentos por segundo). Posteriormente, foram substituí-

dos pelas válvulas no ENIAC em 1946 (capazes de 100.000 de chaveamentos por segundo), e finalmente pelos transistores (semicondutores) inventados em Stanford em 1947. Os circuitos integrados (ou CI's) são encapsulamentos compactos (LSI - Large Scale Integration e VLSI - Very Large Scale Integration) de circuitos constituídos de minúsculos transistores.

4.2 Unidade Central de Processamento

A Unidade Central de Processamento é a responsável pelo processamento e execução de programas armazenados na MP.

Funções:
Executar instruções - realizar aquilo que a instrução determina.
Realizar o controle das operações no computador.

a) Unidade Lógica e Aritmética (ULA) - responsável pela realização das operações lógicas (E, OU, etc) e aritméticas (somar, etc).

b) Unidade de Controle (UC) - envia sinais de controle para toda a máquina, de forma que todos os circuitos e dispositivos funcionem adequada e sincronizadamente.

4.3 Memória Principal (MP)

A Memória Principal tem por finalidade armazenar toda a informação que é manipulada pelo computador - programas e dados. Para que um programa possa ser manipulado pela máquina, ele primeiro precisa estar armazenado na memória principal.

OBS.: os circuitos da Memória Principal não são combinatórias, eles tem capacidade de armazenar bits. Os circuitos usados são do tipo "flip-flop", que serão vistos em sistemas operacionais.

Tem por finalidade permitir a comunicação entre o usuário e o computador. OBS.:

Para executar um programa, bastaria UCP e MP; no entanto, sem os dispositivos de E/S não haveria a comunicação entre o usuário e o computador.

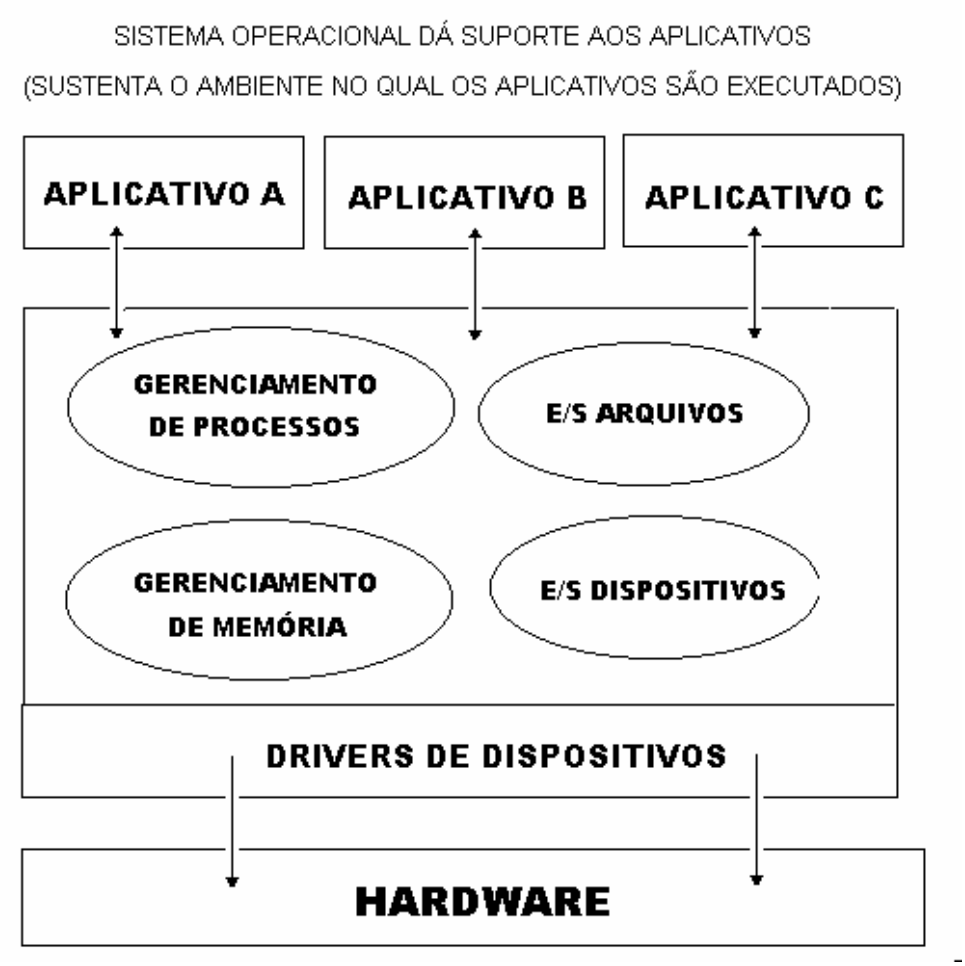
Processamento automático de dados

O programas são armazenados na MP e a UCP é capaz de executar um processamento inteiro sem a intervenção do usuário, mesmo que haja vários desvios no programa.

Passos:

- armazenar o programa na MP;
- - indicar à UCP onde o programa está armazenado.

Estas operações são realizadas pelo **SISTEMA OPERACIONAL**:



Sincronização de Operação do Sistema

Imagine um barco a remo em uma competição, em que a plena velocidade e direção somente é atingida porque todos os remadores fazem seus movimentos de forma coordenada, regidos por um "patrão" - geralmente o timoneiro que indica o ritmo das remadas.

Imagine uma outra analogia: o final do intervalo de um espetáculo é sinalizado por uma campainha, indicando que os espectadores devem dirigir-se aos seus lugares para o próximo ato. Alguns instantes depois, tocará uma outra campainha, indicando que terminou o intervalo e que o espetáculo vai recomeçar, entendendo-se que os espectadores já devem estar todos sentados em seus lugares, prontos para o próximo ato.

Nestes dois exemplos, aparece a necessidade de um elemento externo que fica responsável pela coordenação dos tempos entre diferentes componentes de um sistema, que se comportam de acordo com suas respectivas leis próprias e com tempos próprios, permitindo que suas atividades interrelacionadas sejam **SINCRONIZADAS** de forma a poder realizar um trabalho em conjunto.

As diversas partes de um computador comportam-se aproximadamente desta forma: instruções e dados, após sofrerem algum processamento em um determinado componente, devem trafegar para o próximo estágio de processamento (através de condutores - um barramento ou um cabo), de forma a estarem lá a tempo de serem processados. O computador envia a todos os seus componentes um sinal elétrico regular - o pulso de "*clock*" - que fornece uma referência de tempo para todas as atividades e permite o sincronismo das operações internas. O pulso de *clock* indica que um ciclo (um "estado") terminou, significando que o processamento deste ciclo está terminado e um outro ciclo se inicia, determinando a alguns circuitos que iniciem a transferência dos dados nele contidos (abrindo a porta lógica para os próximos estágios) e a outros que recebam os dados e executem seu processamento.

O ***clock*** é um pulso alternado de sinais de tensão alta ("high") e baixa ("low"), gerado pelos circuitos de relógio (composto de um cristal oscilador e circuitos auxiliares).

4.3.1 Tecnologias das memórias

As primeiras tecnologias utilizadas em memórias foram as memórias de núcleos magnéticos, hoje apenas uma curiosidade. As memórias modernas são compostas por circuitos semicondutores, com novas tecnologias sendo criadas a cada ano permitindo que grandes quantidades de células de memória sejam encapsuladas em pequenas pastilhas.

4.3.2 Hierárquia da memória

A MP não é o único dispositivo de armazenamento de um computador. Em função de características como tempo de acesso, capacidade de armazenamento, custo, etc., podemos estabelecer uma hierarquia de dispositivos de armazenamento em computadores.

Tipo	**Capacidade**	**Velocidade**	**Custo**	**Localização**	**Volatilidade**
Registrador	Bytes	muito alta	muito alto	UCP	Volátil
Memória Cache	Kbytes	alta	alto	UCP/placa	Volátil
Memória Principal	Mbytes	média	médio	Placa	Volátil
Memória Auxiliar	Gbytes	baixa	baixo	Externa	Não Volátil

A UCP vê nesta ordem e acessa primeiro a que está mais próxima. Subindo na hierarquia, quanto mais próximo da UCP, maior velocidade, maior custo, porém menor capacidade de armazenamento.

4.3.3 Controle de Memória

A Memória Principal é a parte do computador onde programas e dados são armazenados para processamento. A informação permanece na memória principal apenas enquanto for necessário para seu emprego pela UCP, sendo então a área de MP ocupada pela informação pode ser liberada para ser posteriormente sobregravada por outra informação. Quem controla a utilização da memória principal é o Sistema Operacional.

4.3.4 Registradores

Registradores são dispositivos de armazenamento temporário, localizados na UCP, extremamente rápidos, com capacidade para apenas um dado (uma palavra). Devido a sua tecnologia de construção e por estar localizado como parte da própria pastilha ("*chip*") da UCP, é muito caro. O conceito de registrador surgiu da necessidade da UCP de armazenar temporariamente dados intermediários durante um processamento. Por exemplo, quando um dado resultado de operação precisa ser armazenado até que o resultado de uma busca da memória esteja disponível para com ele realizar uma nova operação.

Máquinas RISC são geralmente construídas com um grande conjunto de registradores, de forma a trazer os dados para o mais próximo possível da UCP, de forma a que o programa opere sempre sobre dados que estão em registradores.

Registradores são VOLÁTEIS, isto é, dependem de estar energizados para manter armazenado seu conteúdo.

4.3.5 Memória Cache

Com o desenvolvimento da tecnologia de construção da UCP, as velocidades foram ficando muito mais altas que as das memórias, que não tiveram a mesma evolução de velocidade (o aperfeiçoamento das memórias se deu mais no fator capacidade). Desta forma, os tempos de acesso às memórias foram ficando insatisfatórios e a UCP ao buscar um dado na memória precisa ficar esperando muitos ciclos até que a memória retorne o dado buscado ("*wait states*"), configurando um gargalo (*"bottle-neck"*) ao desempenho do sistema.

Por esse motivo, desenvolveram-se outras arquiteturas de memória privilegiando a velocidade de acesso. A arquitetura da memória *cache* é muito diferente da arquitetura da memória principal e o acesso a ela é muitas vezes mais rápido (p.ex: 5 ns contra 70 ns).

No entanto, o custo de fabricação da memória *cache* é muito maior que o da MP. Desta forma, não é econômico construir um computador somente com tecnologia de memória *cache*. Criou-se então um artifício, incorporando-se ao computador uma pequena porção de memória *cache*, localizada entre a UCP e a MP, e que funciona como um espelho de parte da MP.

Desenvolveram-se ainda algoritmos que fazem com que, a cada momento, a memória *cache* armazene a porção de código ou dados (por exemplo, uma sub-rotina) que estão sendo usados pelas UCP. Esta transferência (MP <--> *Cache*) é feita pelo *hardware*: ela independe do *software*, que ignora se existe ou não memória cache, portanto ignora essa transferência; nem o programador nem o sistema operacional têm que se preocupar com ela.
A memória *cache* opera em função de um princípio estatístico comprovado: em geral, os programas tendem a referenciar várias vezes pequenos trechos de programas, como *loops*, sub-rotinas, funções e só tem sentido porque programas executados linearmente, seqüencialmente, são raros. Desta forma, algoritmos (chamados algoritmos de cache) podem controlar qual parte do código ficará copiado na *cache*, a cada momento.

Quando a MP busca um determinado trecho de código e o encontra na cache, dá-se um *"cache hit"* , enquanto se o dado não estiver presente na cache será necessário requisitar o mesmo à MP, acarretando atraso no processamento e dá-se um "cache miss" ou "cache fault". O índice de *cache hit* ou taxa de acerto da *cache* é geralmente acima de 90%.

Memórias *cache* também são VOLÁTEIS, isto é, dependem de estar energizadas para manter gravado seu conteúdo.

4.3.6 Memórias Auxiliares

Memórias auxiliares resolvem problemas de armazenamento de grandes quantidades de informações. A capacidade da MP é limitada pelo seu relativamente alto cus-

to, enquanto as memórias auxiliares tem maior capacidade e menor custo; portanto, o custo por bit armazenado é muito menor.

Outra vantagem importante é que as memórias auxiliares não são VOLÁTEIS, isto é, não dependem de estar energizadas para manter gravado seu conteúdo.

Os principais dispositivos de memória auxiliar são: discos rígidos (ou HD), drives de disquete, unidades de fita, CD-ROM, DVD, unidades ótico-magnéticas, etc.

OBS.: *Cache* de disco não é a mesma tecnologia da memória *cache*. Trata-se do emprego do mesmo **conceito** da memória *cache*, para acelerar a transferência de dados entre disco, MP e UCP, usando um programa (um *software*, por ex.: *Smart-Drive*) para manter um espelho do conteúdo de parte do disco (a mais provável de ser requisitada a seguir pela UCP) gravado em uma parte da Memória Principal. Recentemente, as unidades de disco passaram a incorporar em sua interface *chips* de memória - tipicamente 32 a 64 Kbytes - para acelerar a transferência de dados, utilizando um algoritmo de *cache*.

4.3.7 Estrutura da Memória Principal - Células e Endereços

A memória precisa ter uma organização que permita ao computador guardar e recuperar informações quando necessário. Não teria nenhum sentido armazenar informações que não fosse possível recuperar depois. Portanto, não basta transferir informações para a memória. É preciso ter como encontrar essa informação mais tarde, quando ela for necessária, e para isso é preciso haver um mecanismo que registre exatamente onde a informação foi armazenada (lembrando nossa analogia com o computador hipotético, imagine encontrar uma informação guardada ao acaso, se nosso escaninho tivesse 1 milhão de compartimentos).

Célula é a unidade de armazenamento do computador. A memória principal é organizada em células. Célula é a menor unidade da memória que pode ser endereçada (não é possível buscar uma "parte" da célula) e tem um tamanho fixo (para cada máquina). As memórias são compostas de um determinado número de células ou

posições. Cada célula é composta de um determinado número de bits. Todas as células de um dado computador tem o mesmo tamanho, isto é, todas as células daquele computador terão o mesmo número de bits.

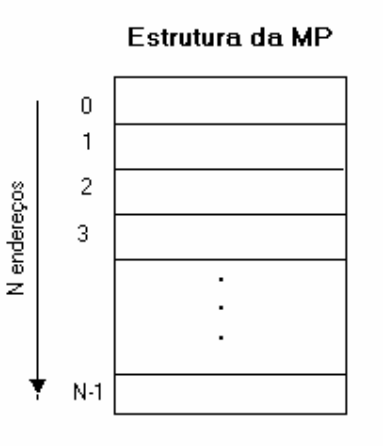
Cada célula é identificada por um **endereço** único, pela qual é referenciada pelo sistema e pelos programas. As células são numeradas seqüencialmente, uma a uma, de 0 a (N-1), chamado o **endereço da célula**. **Endereço** é o localizador da célula, que permite identificar univocamente uma célula. Assim, **cada célula pode ser identificada pelo seu endereço**.

Unidade de transferência é a quantidade de bits que é transferida da memória em uma única operação de leitura ou transferida para a memória em uma única operação de escrita. O tamanho da célula poderia ser igual ao da **palavra**, e também à unidade de transferência, porém por razões técnicas e de custo, são freqüentemente diferentes.

OBS.: Uma célula não significa o mesmo que uma palavra; uma célula não necessariamente contém uma palavra. Palavra é a unidade de processamento da UCP. Uma palavra deve representar um dado ou uma instrução, que poderia ser processada, armazenada ou transferida em uma única operação. No entanto, em geral não é assim que acontece e os computadores comerciais não seguem um padrão único para a organização da UCP e MP. Computadores comerciais (tais como por exemplo os baseados nos processadores Intel 486) podem ter o tamanho da palavra definido como de 32 bits, porém sua estrutura de memória tem células de 16 bits.

A estrutura da memória principal é um problema do projeto de hardware:
- mais endereços com células menores ou
- menos endereços com células maiores?

O tamanho mais comum de célula era 8 bits (1 byte); hoje já são comuns células contendo vários bytes.



Número de bits para representar um endereço

Expressão geral: MP com endereços de 0 a (N-1)

$N = 2^x$ logo:

$x = \log_2 N$

sendo x = nº de bits para representar um endereço e N o número de endereços.

4.3.8 Capacidade da Memória Principal

A capacidade da MP em bits é igual ao produto do nº de células pelo total de bits por célula.

T = N x M

T = capacidade da memória em bits

N = nº de endereços (como vimos anteriormente, $N=2^x$ sendo x = nº de bits do endereço)

M = nº de bits de cada célula

Para encontrar a capacidade em bytes, bastaria encontrar a capacidade em bits e depois multiplicar por 8 (cada byte contém 8 bits) ou então converter o tamanho da célula para bytes e depois multiplicar pelo número de células.

O último endereço na memória é o endereço N-1 (os endereços começam em zero e vão até N-1).

4.3.9 Terminologia

- **CÉLULA DE MEMÓRIA** - (Flip-flop, armazenam um único bit)
- **PALAVRA DE MEMÓRIA** - (Um grupo de células, normalmente 4 a 64 bits)
- **CAPACIDADE** - 1K = 1024 , 1M = 1.048.576 - (2K x 8 = 2048 x 8 = 16384 bits)
- **ENDEREÇO** - Identifica a posição de uma palavra na memória.
- **OPERAÇÃO DE LEITURA** - Também chamada de “busca” na memória.
- **OPERAÇÃO DE ESCRITA** - Também chamada de “armazenamento”.
- **TEMPO DE ACESSO** - Quantidade de tempo necessária à busca ou armazenamento.
- **MEMÓRIA VOLÁTIL** - Necessitam de energia elétrica para reter a informação armazenada.
- **MEMÓRIA DE ACESSO RANDÔMICO (RAM)** - O tempo de acesso é constante para qualquer endereço da memória.
- **MEMÓRIA DE ACESSO SEQUENCIAL (SAM)** - O tempo de acesso não é constante, mas depende do endereço. Ex: fitas magnéticas.
- **MEMÓRIA DE LEITURA/ESCRITA (RWM)** - Qualquer memória que possa ser lida ou escrita com igual facilidade.
- **MEMÓRIA DE LEITURA (ROM)** - Uma classe de memórias a semicondutor projetadas para aplicações onde a taxa de operações de leitura é infinitamente mais alta do que as de escrita. São não-voláteis.
- **DISPOSITIVOS DE MEMÓRIA ESTÁTICA** - Enquanto houver energia elétrica aplicada, não há necessidade de rescrever a informação.
- **DISPOSITIVOS DE MEMÓRIA DINÂMICA** - Necessitam de recarga (refresh)
- **MEMÓRIA PRINCIPAL (INTERNA)** - É a mais rápida do sistema. (Instruções e dados).
- **MEMÓRIA DE MASSA** - É mais lenta que a principal. Grande capacidade de armazenamento

4.4 Unidade Central de Processamento

A Unidade Central de Processamento - UCP (em inglês, Central Processing Unity - CPU) é a responsável pelo processamento e execução dos programas armazenados na MP. As funções da UCP são: executar as instruções e controlar as operações no

computador.

A UCP é composta de duas partes: Unidade Aritmética e Lógica e Unidade de Controle.

4.4.1 Unidade Aritmética e Lógica

Tem por função a efetiva execução das instruções.

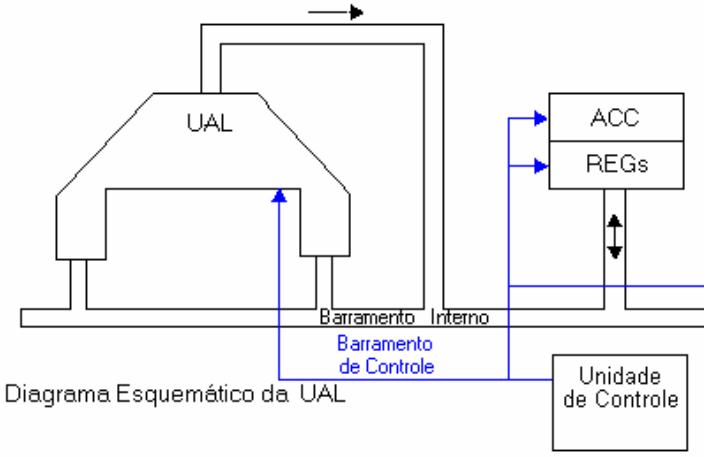
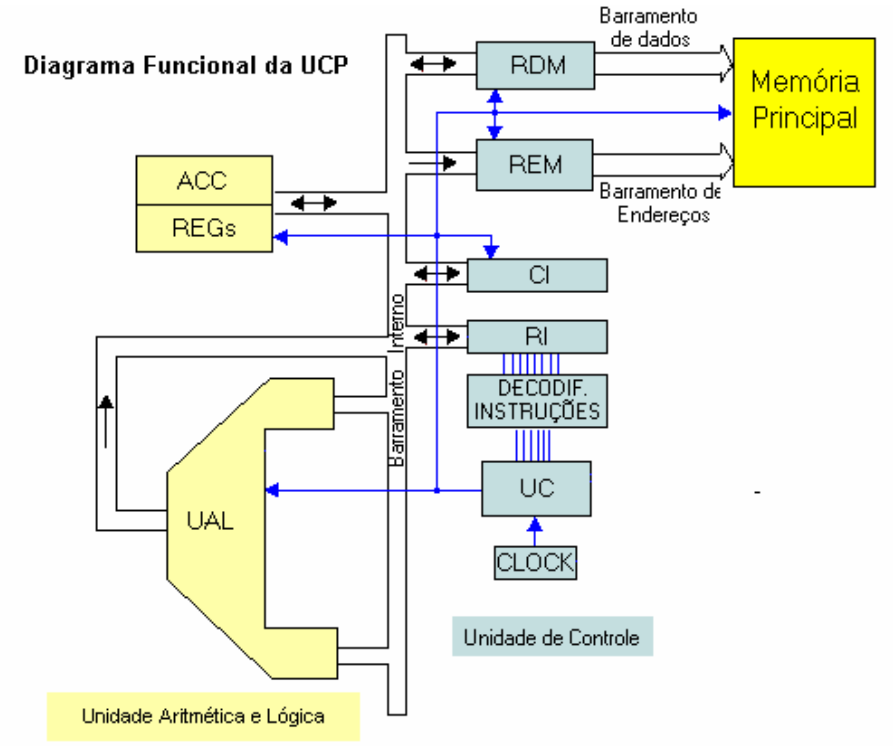


Diagrama Esquemático da UAL

4.4.2 Unidade de Controle

Tem por funções a busca, interpretação e controle de execução das instruções, e o controle dos demais componentes do computador. A seguir é apresentado o diagrama esquemático de uma UCP.



4.4.3 Registradores Importantes na UCP

- Na **UC - CI Contador de Instruções** (em inglês: PC - Program Counter) - armazena o endereço da próxima instrução a ser executada - tem **sempre o mesmo tamanho do REM.**
- Na **UC - RI Registrador de Instrução** (em inglês: IR - Instruction Register) - armazena a instrução a ser executada.
- Na **UAL - ACC Acumulador** (em inglês: ACC - Accumulator) - armazena os dados (de entrada e resultados) para as operações na UAL; o acumulador é um dos principais elementos que definem o tamanho da palavra do computador - o **tamanho da palavra é igual ao tamanho do acumulador**.

4.4.4 Instruções

Para que um programa possa ser executado por um computador, ele precisa ser constituído de uma série de instruções de máquina e estar armazenado em células sucessivas na memória principal. A UCP é responsável pela execução das instruções que estão na memória.

Quem executa um programa é o hardware e o que ele espera encontrar é um programa em linguagem de máquina (uma seqüência de instruções de máquina em código binário). A linguagem de máquina é composta de códigos binários, representando instruções, endereços e dados e está totalmente vinculada ao conjunto ("*set*") de instruções da máquina.

Um ser humano usa seu conhecimento e inteligência para traduzir uma tarefa complexa (tal como, por exemplo, a tarefa de buscar uma pasta num arquivo) numa série de passos elementares (identificar o móvel e gaveta onde está a pasta, andar até o móvel, abrir a gaveta, encontrar a pasta, retirar a pasta e fechar a gaveta). Para o computador, uma instrução precisa ser detalhada, dividida em pequenas etapas de operações, que são dependentes do conjunto de instruções do computador e individualmente executáveis.

Fazendo um paralelo com linguagens de alto nível, o programa elaborado pelo programador (o código-fonte, composto de instruções complexas) precisa ser "traduzido" em pequenas operações elementares (primitivas) executáveis pelo hardware. Cada uma das instruções tem um código binário associado, que é o código da operação.

4.4.4.1 Formato Geral de uma Instrução

Código de operação (OPCODE)	**Operando (s) (OP)**

- **Código de Operação ou OPCODE** - identifica a operação a ser realizada pelo processador. É o campo da instrução cuja valor binário identifica (é o código binário) da operação a ser realizada. Este código é a entrada no decodificador de instruções na unidade de controle. Cada instrução deverá ter um código único que a identifique.

- **Operando(s)** - é ou são o(s) campo(s) da instrução cujo valor binário sinaliza a localização do dado (ou é o próprio dado) que será manipulado (processado) pela instrução durante a operação. Em geral, um operando identifica o endereço de memória onde está contido o dado que será manipulado, ou pode conter o endereço onde o resultado da operação será armazenado. Finalmente, um operando pode também indicar um Registrador (que conterá o dado propriamente dito ou um endereço de memória onde está armazenado o dado). Os operandos fornecem os dados da instrução.

Obs: Existem instruções que não tem operando. Ex.: Instrução HALT (PARE).

4.4.5 Conjunto de Instruções

Quando se projeta um *hardware*, define-se o seu conjunto (*"set"*) de instruções - o conjunto de instruções elementares que o *hardware* é capaz de executar. O projeto de um processador é centrado no seu **conjunto (*"set"*) de instruções**. Essa é uma das mais básicas decisões a ser tomada pelo Engenheiro de projeto. Quanto menor e mais simples for este conjunto de instruções, mais rápido pode ser o ciclo de tempo do processador.

Funcionalmente, um processador precisa possuir instruções para:

- operações matemáticas

1. aritméticas: +, - , × , ÷ ...
2. lógicas: and, or, xor, ...
3. de complemento
4. de deslocamento

- operações de movimentação de dados (memória <--> UCP, reg <--> reg)
- operações de entrada e saída (leitura e escrita em dispositivos de E/S)
- operações de controle (desvio de seqüência de execução, parada)

As estratégias de implementação de processadores são:

- *CISC - Complex Instruction Set Computer* - exemplo: PC, Macintosh; um conjunto de instruções maior e mais complexo, implicando num processador mais complexo, com ciclo de processamento mais lento; ou
- *RISC - Reduced Instruction Set Computer* - exemplo: Power PC, Alpha, Sparc; um conjunto de instruções menor e mais simples, implicando num processador mais simples, com ciclo de processamento rápido.

Obs.: adotaremos o termo **instrução** para as instruções de máquina ou em linguagem Assembly e **comando** para linguagens de alto nível.

Há hoje uma crescente tendência a se utilizar um conjunto de instruções reduzido, de vez que os compiladores tendem a usar em geral apenas uma pequena quantidade de instruções. Há também vantagens na implementação do *hardware* - maior simplicidade, menor tempo de ciclo de instrução.

O projeto de um processador poderia ser resumido em:
a) Definir o conjunto de instruções (todas as possíveis instruções que o processador poderá executar)
· definir formato e tamanho das instruções
· definir as operações elementares

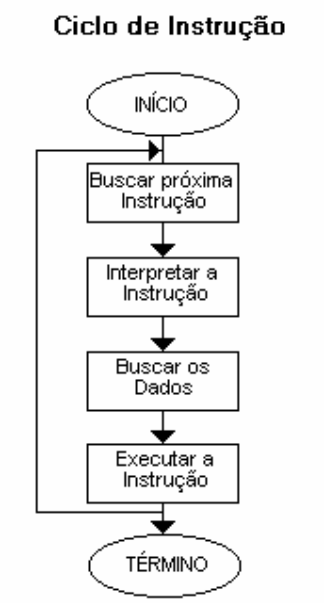
a) Projetar os componentes do processador (UAL, UC, registradores, barramentos)

Duas estratégias são possíveis na construção do decodificador de instruções da UC:
wired logic (as instruções são todas implementadas em circuito).
microcódigo (apenas um grupo básico de instruções são implementadas em circuitos; as demais são "montadas" através de microprogramas que usam as instruções básicas.

4.4.6 Ciclo de Instrução

As instruções são executadas seqüencialmente (a não ser pela ocorrência de um desvio), uma a uma.

O CI indica a seqüência de execução, isto é, o CI controla o fluxo de execução das instruções. A seguir é ilustrado o ciclo de processamento de uma instrução.



Descrição do processamento de uma instrução na UCP:

- a UC lê o endereço da próxima instrução no CI;
- a UC transfere o endereço da próxima instrução, através do barramento interno, para o REM.

4.5 Comunicação entre Memória Principal e a Unidade Central de Processamento

4.5.1 Barramentos

Os diversos componentes dos computadores se comunicam através de **barramentos**. Barramento é um conjunto de condutores elétricos que interligam os diversos componentes do computador e de circuitos eletrônicos que controlam o fluxo dos bits. Para um dado ser transportado de um componente a outro, é preciso emitir os sinais de controle necessários para o componente-origem colocar o dado no barramento e para o componente-destino ler o dado do barramento. Como um dado é composto por bits (geralmente um ou mais bytes) o barramento deverá ter tantas linhas condutoras quanto forem os bits a serem transportados de cada vez.

Obs.: Em alguns computadores (usando uma abordagem que visa a redução de custos), os dados podem ser transportados usando mais de um ciclo do barramento.

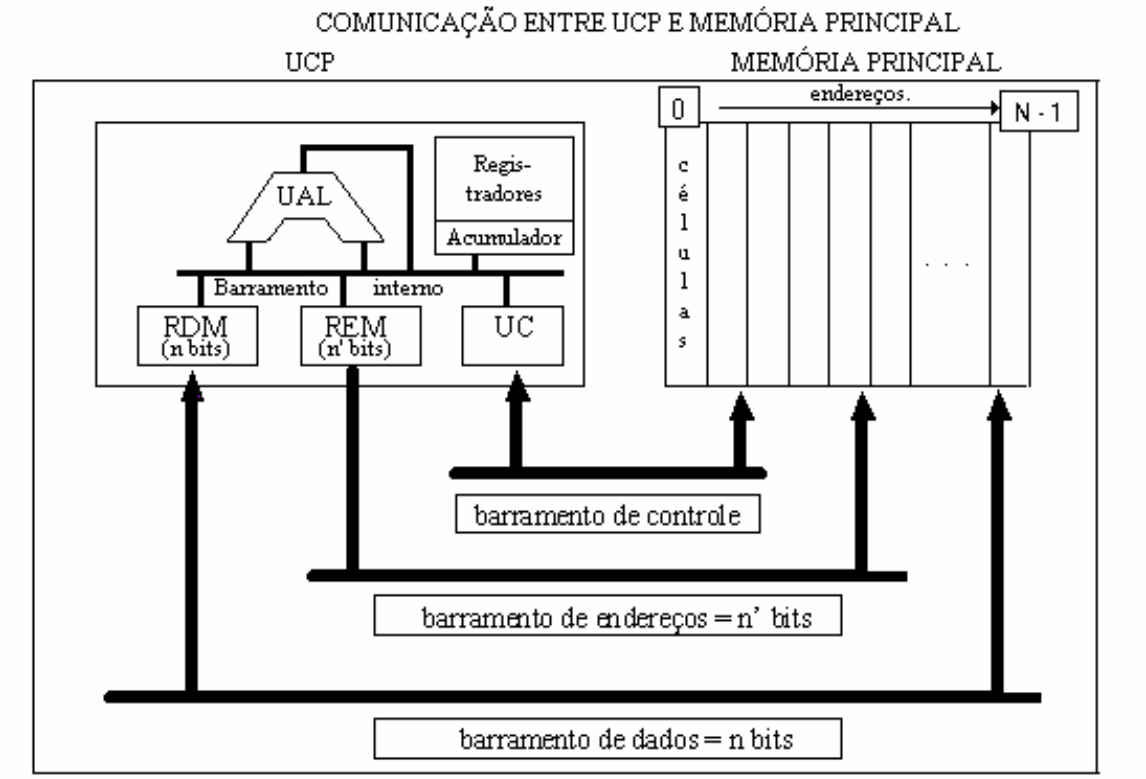
Assim, se quisermos transferir um byte - por exemplo, 01001010 - da UCP para a Memória Principal, os circuitos de controle se encarregarão de colocar o byte 01001010 no barramento, ou seja, colocariam sinais de tensão "*high*" nas 2ª, 4ª e 7ª linhas do barramento (por convenção, os bits são sempre ordenados da direita para a esquerda) e de informar à memória para ler o dado no barramento. Os dados são representados no barramento na forma de sinais de tensão, sendo que um sinal de tensão de uns poucos volts (*"high"*) representa o bit "1" e um sinal próximo de zero volts (*"low"*) representa o bit "0".

4.5.2 Registradores Utilizados

A comunicação entre MP e UCP usa dois registradores da UCP chamados de Registrador de Endereços de Memória - **REM** ou, em inglês, Memory Address Register (MAR), bem o como Registrador de Dados da Memória - **RDM** ou, em inglês, Memory Buffer Register (MBR).

x = nº de bits do barramento de endereços; em geral (mas não obrigatoriamente) é igual ao nº de bits do Registrador de Endereços de Memória - REM.
M = nº de bits contidos em uma célula; M em geral (mas não obrigatoriamente) é igual ao nº de bits do Registrador de Dados da Memória - RDM.

4.6 Esquema de Funcionamento da Comunicação entre MP / UCP



UCP / MP

Barramento de endereços – unidirecional (só a UCP envia dados - *write* - ou lê dados - *read* - da MP)

Barramento de dados – bidirecional

Barramento de controle – bidirecional

UCP ---> MP (controles ... - r/w)

MP -----> UCP (wait ...)

ELEMENTOS ENVOLVIDOS

- Barramentos

de dados – bidirecional

de endereços –unidirecional

de controle – bidirecional

- Registradores

REM - Registrador de Endereços de Memória (MAR - Memory Address Register)

RDM - Registrador de Dados de Memória (MBR - Memory Buffer Register)

4.7 Palavra (Unidade de Informação)

Palavra é a unidade de informação do sistema UCP / MP. A conceituação mais usada (IBM, Digital) define palavra como sendo a capacidade de manipulação de bits do núcleo do computador (UCP e MP). Pressupõe-se aqui que todos os elementos do núcleo do computador (o que incluí o tamanho da UAL, do acumulador e registradores gerais da UCP e o barramento de dados) tenham a mesma largura (processem simultaneamente o mesmo número de bits), o que nem sempre acontece. Muitas vezes encontram-se computadores em que o tamanho da UAL e do acumulador (e registradores gerais) não é o mesmo tamanho dos barramentos. Desta forma, encontram-se especificações de "computadores de 64 bits" mesmo quando seu barramento de dados é de 32 bits, nesse caso referindo-se exclusivamente à capacidade de manipulação da UCP de 64 bits (isto é, sua UAL e acumulador tem 64 bits). Esta conceituação é imprecisa (às vezes, enganosa) e pode levar a erros de avaliação da capacidade de processamento de um computador.

Como exemplos, citamos os microprocessadores Intel 8086 (16 bits, sendo todos seus elementos de 16 bits) e seu "irmão" mais novo 8088, usado nos primeiros IBM/PC e XT (idêntico sob quase todos os aspectos ao 8086 e também dito de 16 bits, sendo que UAL e registradores são de 16 bits mas o barramento de dados é de apenas 8 bits, por economia e razões de compatibilidade com toda uma geração de placas de 8 bits). Destaque-se que nesse caso as transferências de dados através do barramento de dados se fazem em duas etapas, um byte de cada vez, e em conseqüência no 8088 elas consomem o dobro dos ciclos de barramento que o 8086), o que torna suas operações de transferência de dados mais lentas que as de seu "irmão" 8086.

Concluindo, deve-se analisar caso a caso, porque a simples menção ao tamanho da palavra não é uma terminologia que permita definir de forma conclusiva sobre a ar-

quitetura do computador.

Em geral, o termo "célula" é usada para definir a unidade de armazenamento (o tamanho de células de memória) e o termo "palavra" para definir a unidade de transferência e processamento, significando na prática quantos bits o computador movimenta e processa em cada operação.

Não confundir: célula não é sinônimo de palavra, embora em algumas máquinas a palavra seja igual à célula. A palavra de um computador pode ter 1 byte (p.ex, 8080), 2 bytes (p.ex. 80286), 4 bytes (p.ex. 486, o Pentium, e muitos mainframes IBM) e mesmo 8 bytes (p.ex. o Alpha da DEC).

Células de memória muitas vezes tem o tamanho de 1 ou 2 bytes - de 8 a 16 bits.

4.8 Tempo de Acesso

Tempo de acesso (ou tempo de acesso para leitura) é o tempo decorrido entre uma requisição de leitura de uma posição de memória e o instante em que a informação requerida está disponível para utilização pela UCP. Ou seja, o tempo que a memória consome para colocar o conteúdo de uma célula no barramento de dados. O tempo de acesso de uma memória depende da tecnologia da memória. As memórias DRAM (Dynamic RAM - as mais comuns hoje) tem tempo de acesso na faixa de 60 ns.

Tempo de ciclo (ou ciclo de memória é conceituado como o tempo decorrido entre dois ciclos sucessivos de acesso à memória. As memórias dinâmicas perdem seu conteúdo em alguns instantes e dependem de ser periodicamente atualizadas (ciclo de *"refresh"*). No caso das SRAM (*Static RAM* ou memórias estáticas), que não dependem de *"refresh"*, o tempo de ciclo é igual ao tempo de acesso. As memórias dinâmicas, no entanto, requerem ciclos periódicos de *"refresh"*, o que faz com que a memória fique indisponível para novas transferências, a intervalos regulares necessários para os ciclos de *"refresh"*. Assim, as memórias DRAM tem ciclo de memória maior que o tempo de acesso.

O tempo de acesso de qualquer memória tipo RAM (*Random Access Memory* ou memória de acesso aleatório) é independente do endereço a ser acessado (a posição de memória a ser escrita ou lida), isso é, o tempo de acesso é o mesmo qualquer que seja o endereço acessado.

4.9 Acesso à Memória Principal

O acesso à MP é ALEATÓRIO, portanto qualquer que seja o endereço (a posição) de memória que se queira acessar, o tempo de acesso é o mesmo (constante).

Obs.: Embora a MP seja endereçada por célula, a UCP em geral acessa a MP por palavra.

O endereçamento por célula dá maior flexibilidade de armazenamento, em compensação o número de acessos é em geral maior.

4.9.1 Acesso Tipo Ler ou Escrever

4.9.1.1 Leitura: Ler da Memória

Significa requisitar à MP o conteúdo de uma determinada célula (recuperar uma informação). Esta operação de recuperação da informação armazenada na MP consiste na transferência de um conjunto de bits (cópia) da MP para a UCP e é **não destrutiva**, isto é, o conteúdo da célula não é alterado.

SENTIDO: da MP para a UCP

Passos Executados pelo Hardware:

a.1) a UCP armazena no REM o endereço onde a informação requerida está armazenada;

a.2) a UCP comanda uma leitura;

a.3) o conteúdo da posição identificada pelo endereço contido no REM é transferido para o RDM e fica disponível para a UCP.

4.9.1.2 Escrita: Escrever na Memória

Significa escrever uma informação em uma célula da MP (armazenar uma informação). Esta operação de armazenamento da informação na MP consiste na transferência de um conjunto de bits da UCP para a MP e é **destrutiva** (isto significa que qualquer informação que estiver gravada naquela célula será sobregravada).

SENTIDO: da UCP para a MP

PASSOS EXECUTADOS PELO HARDWARE:

b.1) a UCP armazena no REM o endereço de memória da informação a ser gravada e no RDM a própria informação;

b.2) a UCP comanda uma operação de escrita;

b.3) a informação armazenada no RDM é transferida para a posição de memória cujo endereço está contido no REM.

4.10 Classificação das Memórias

Quanto à leitura e escrita, as memórias podem ser classificadas como:

4.10.1 R/W – Read and Write (memória de leitura e escrita) - RAM

Comumente (e impropriamente) chamada de RAM (*Random Access Memory* ou memória de acesso aleatório), embora não seja a única RAM.

Esta memória permite operações de escrita e leitura pelo usuário e pelos programas. Seu tempo de acesso é da ordem de 70ns e independe do endereço acessado. É construída com tecnologia de semicondutores (bipolar, CCD), pode ser estática (SRAM) ou dinâmica (DRAM) e é volátil. A MP é construída com memória R/W.

4.10.2 ROM – Read Only Memory (memória apenas de leitura)

Esta memória permite apenas a leitura e uma vez gravada não pode mais ser alterada. Também é de acesso aleatório (isto é, é também uma RAM), mas não é volátil. É utilizada geralmente por fabricantes para gravar programas que não se deseja permitir que o usuário possa alterar ou apagar acidentalmente (tal como por ex. a BIOS

- *Basic Input Output System* e microprogramas de memórias de controle). Quando se liga uma máquina, é da ROM que vem os programas que são carregados e processados no "*boot*" (na inicialização o *hardware* aponta automaticamente para o primeiro endereço da ROM). Desta forma, parte do espaço de endereçamento da MP é ocupado por ROM. A ROM é mais lenta que a R/W e é barata, porém o processo produtivo depende de ser programada por máscara ("*mask programmed*") em fábrica e devido ao alto custo da máscara somente se torna econômica em grandes quantidades.

Obs.: *Boot* (ou *bootstrap loader*) é o processo de inicialização e carga dos programas básicos de um computador, automática, sem intervenção externa. Este termo vem de uma analogia com um processo (impossível) que seria uma pessoa se levantar puxando-se pelos cordões de suas próprias botas.

4.10.3 PROM – Programmable Read Only Memory (Memória programável de leitura)

Esta memória é uma ROM programável (em condições e com máquinas adequadas, chamadas queimadores de PROM) e geralmente é comprada "virgem" (sem nada gravado), sendo muito utilizada no processo de testar programas no lugar da ROM, ou sempre que se queira produzir ROM em quantidades pequenas. Uma vez programada (em fábrica ou não), não pode mais ser alterada.

4.10.4 EPROM - Erasable Programmable Read Only Memory (Memória programável apagável de leitura)

Esta memória é uma PROM apagável. Tem utilização semelhante à da PROM, para testar programas no lugar da ROM, ou sempre que se queira produzir ROM em quantidades pequenas, com a vantagem de poder ser apagada e reutilizada.

4.10.5 EEPROM ou E2PROM – Eletrically Erasable Programmable Read Only Memory (Memória programável apagável eletronicamente)

Também chamada EAROM (*Electrically Alterable* ROM). Esta memória é uma EPROM apagável por processo eletrônico, sob controle da UCP, com equipamento e

programas adequados. É mais cara e é geralmente utilizada em dispositivos aos quais se deseja permitir a alteração, via modem, possibilitando a carga de novas versões de programas à distância ou então para possibilitar a reprogramação dinâmica de funções específicas de um determinado programa, geralmente relativas ao *hardware* (p.ex., a reconfiguração de teclado ou de modem, programação de um terminal, etc).

4.11 Lógica Temporizada

Conforme vimos ao analisar a comunicação entre UCP e memória, as instruções, os dados e os endereços "trafegam" no computador através dos barramentos (de dados, de endereços e de controle), sob a forma de bits representados por sinais elétricos: uma tensão positiva alta ("*high*" - geralmente no entorno de 3 volts) significando "1" e uma tensão baixa ("*low*" - próxima de zero) significando "0". Mas os dados no computador não ficam estáticos; pelo contrário, a cada ciclo (cada "estado") dos circuitos, os sinais variam, de forma a representar novas instruções, dados e endereços. Ou seja, **os sinais ficam estáticos apenas por um curto espaço de tempo**, necessário e suficiente para os circuitos poderem detectar os sinais presentes no barramento naquele instante e reagir de forma apropriada. Assim, periodicamente, uma nova configuração de bits é colocada nos circuitos, e tudo isso só faz sentido se pudermos de alguma forma organizar e sincronizar essas variações, de forma a que, num dado instante, os diversos circuitos do computador possam "congelar" uma configuração de bits e processá-las. Para isso, é preciso que exista um outro elemento, que fornece uma base de tempo para que os circuitos e os sinais se sincronizem. Este circuito é chamado ***clock*** - o **relógio** interno do computador. Cada um dos estados diferentes que os circuitos assumem, limitados pelo sinal do *clock*, é chamado um **ciclo de operação**.

4.11.1 Clock

A Unidade de Controle da UCP envia a todos os componentes do computador um sinal elétrico regular - o pulso de "*clock*" - que fornece uma referência de tempo para todas as atividades e permite o sincronismo das operações internas. O *clock* é um

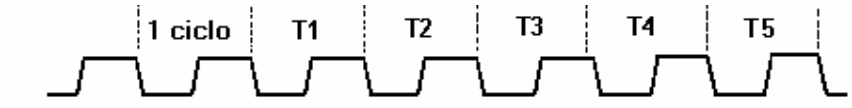
pulso alternado de sinais de tensão, gerado pelos circuitos de relógio (composto de um cristal oscilador e circuitos auxiliares).

4.11.2 Ciclo de Operação

Cada um destes intervalos regulares de tempo é delimitado pelo início da descida do sinal, eqüivalendo um **ciclo** à excursão do sinal por um "*low*" e um "*high*" do pulso.

O tempo do ciclo eqüivale ao período da oscilação. A física diz que período é o inverso da freqüência. Ou seja,
P = 1 / f.
A freqüência f do *clock* é medida em hertz. Inversamente, a duração de cada ciclo é chamada de período, definido por P=1/f (o período é o inverso da freqüência).

Por exemplo, se f = 10 hz logo P = 1/10 = 0,1 s.



1 Mhz (1 megahertz) eqüivale a um milhão de ciclos por segundo. Sendo a freqüência de um processador medida em megahertz, o período será então medido em nanosegundos, como vemos no exemplo abaixo:

f = 10 Mhz = 10 x 10^6 hz
P = 10 / 10^6 = 100 ns (1 nanosegundo).

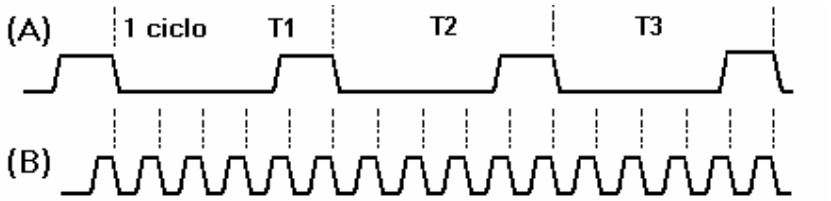
Sempre que se fala sobre máquinas velozes, citamos números em megahertz. Para um melhor entendimento sobre o que ocorre na máquina, em vez de falar sobre a freqüência do *clock* seria mais ilustrativo discutirmos uma outra grandeza: o período (isto é, o tempo de duração de cada ciclo ou simplesmente tempo de ciclo).

Quando se diz que um processador é de 200 Mhz, está-se definindo a freqüência de operação de seu processador (seu *clock*), significando que o processador pode alternar seus estados internos 166 milhões de vezes por segundo. Isto acarreta que

cada ciclo (equivalente a um estado lógico) de operação dura

1 / 200.000.000 s = 5 x 10^{-9} s ou seja, 5 nanosegundos.

Como podemos ver pelo exemplo a seguir, o processador com o *clock* ilustrado em (B) teria um tempo de ciclo cinco vezes menor que o (A) e portanto teria (teoricamente) condições de fazer cinco vezes mais operações no mesmo tempo.



Quando analisamos os números de *clock* de um processador ou barramento, pode ficar uma impressão que esses números não fazem sentido: 133 MHz, 166 MHz ... Vejamos como ficam seus períodos, e como esses números apresentam um padrão regular:

Freqüência (MHz)	Período (ns)
25	40
33	30
40	25
50	20
66	15
100	10
133	7.5
166	6
200	5
266	3.75

Os primeiros computadores tinham um único sinal de *clock* geral, válido para UCP, memória, barramentos de E/S (entrada / saída), etc. À medida que a tecnologia foi se aperfeiçoando, a freqüência de *clock* de operação dos processadores (e, em menor escala, também a das memórias) aumentou em uma escala muito maior que a

dos demais componentes. Desta forma, foi necessário criar diferentes pulsos de *clock* para acomodar as freqüências de operação dos diferentes componentes. A placa-mãe de um PC utiliza uma freqüência-mestra (hoje em geral de 66 Mhz, equivalente a um período de 15 ns, estando em prancheta placas para 100 MHz) para seu barramento (ciclo de barramento), a qual é multiplicada ou dividida para ser utilizada pelos demais componentes:

· o processador tem essa freqüência multiplicada por 2 (133 Mhz) a 4 (266 MHz),
· o barramento PCI usa freqüências reduzidas pela metade (33 Mhz),
· as memórias (ciclos da ordem de 60 ns) usam freqüências reduzidas a um quarto e
· as cache secundárias (ciclos entre 10 e 20 ns) usam a própria freqüência da placa-mãe.

As memórias cache primárias são hoje construídas como parte do processador e usam o mesmo *clock* do processador.

O efeito prático (econômico$$$$) do aumento da freqüência de operação é que a precisão de fabricação dos circuitos tem que ser também maior. O tamanho de cada junção de transistor fica menor (hoje são construídos *chips* com 0,35 microns e uma nova geração com 0,25 microns está em gestação). Uma junção menor requer menor potência para sua operação, menos elétrons para operar uma transição de estados, menor tempo de propagação do sinal, menor potência dissipada.

Em conseqüência da grande precisão exigida, apenas uma pequena parcela dos processadores fabricados (cerca de 5%) consegue operar na máxima freqüência para a qual foram projetados, e a maioria é certificada para operar a freqüências mais baixas. Isto acarreta que, embora todos os processadores de um tipo sejam fabricados pelos mesmos processos e nas mesmas máquinas, apenas alguns serão certificados para a máxima freqüência prevista, o que obriga que o preço dos processadores de *clock* máximo seja muito mais caro que o dos muitos outros que não obtiveram certificação para aquele elevado *clock* e serão vendidos com "*tags*" de 166 ou 133 Mhz, a preços reduzidos.

4.11.3 Instruções por Ciclo

Qual a real importância e significado da freqüência do processador?
Quando se faz uma soma em uma calculadora, basta teclar cada um dos algarismos do 1º número, teclar o sinal de mais, depois teclamos os algarismos do segundo número e teclamos o sinal de igual.

É comum pensar que nos computadores as coisas se passam mais ou menos do mesmo jeito. No entanto, a soma propriamente dita é a menor das tarefas que um computador executa quando soma dois números. Neste exemplo, o computador precisaria localizar a instrução de soma na memória e movê-la para a UCP. Esta instrução estaria codificada na forma de dígitos binários e teria que ser decodificada para determinar a operação a ser realizada (no caso, ADD), o tamanho dos dados (quantas células eles ocupam), determinar a localização e buscar cada um dos números na memória, e só então, finalmente, fazer a soma. Para o computador, a soma é realmente a parte mais simples e mais rápida de toda a operação, já que decodificar a instrução e principalmente obter os dados tomam a maior parte do tempo.

Cada nova geração de processadores tem sido capaz de executar as operações relativas ao processamento de uma instrução em menor número de ciclos do clock. Por exemplo, na família Intel x86:

- 386 - mínimo de 6 ciclos por instrução de soma de 2 números
- 486 - em geral, 2 ciclos por instrução de soma de 2 números
- Pentium - 1 ciclo por instrução de soma de 2 números
- Pentium Pro - 1 ciclo por instrução de soma de 3 números; na soma de mais números, quando um dos números está em memória de baixa velocidade, o Pentium Pro é capaz de "pular" este número e, enquanto busca o número que falta, seguir adiante, buscando e somando os demais números para finalmente incluir o número que faltava na soma.

Usando uma analogia com um automóvel, para andar mais rápido geralmente é mais eficaz trocar de marcha do que acelerar. Comparativamente, um processador de 66

Mhz hoje eqüivaleria a uma pequena pressão no acelerador e um de 300 Mhz ao acelerador pressionado até o fundo (pé na tábua!). Mas, se um 8088 fosse a 1ª marcha, um Pentium II seria equivalente à 5ª marcha e seria certamente muito mais rápido, mesmo que fosse possível "acelerar" o 8088 ao mesmo *clock* do Pentium. Se compararmos um 486 DX4-100 (100 Mhz) com um Pentium também de 100 Mhz, veremos que o Pentium 100 será substancialmente mais rápido, o que se deve à sua arquitetura e não ao *clock*.

É portanto um engano comparar apenas a freqüência do *clock*: o desempenho do processador deve ser avaliado por um conjunto de características da arquitetura, do qual a freqüência do *clock* é apenas um deles - e não o mais importante .

5 CONCEITOS DE INTERRUPÇÃO E TRAP

Pode-se dizer que interrupções e *traps* são as forças que movimentam e dirigem os sistemas operacionais, pois um sistema operacional só recebe o controle da execução quando ocorre alguma interrupção ao *trap*.

Uma **interrupção** é um sinal de *hardware* que faz com que o processador sinalizado interrompa a execução do programa que vinha executando (guardando informações para poder continuar, mais tarde, a execução desse programa) e passe a executar uma rotina específica que trata da interrupção.

Um ***trap*** é uma instrução especial que, quando executada pelo processador, origina as mesmas ações ocasionadas por uma interrupção (salvamento de informações para poder continuar, mais tarde, a execução do programa e desvio para uma rotina específica que trata do *trap*). Pode-se dizer que um *trap* é uma interrupção ocasionada por *software*.

Interrupções podem ser originadas pelos vários dispositivos periféricos (terminais, discos, impressoras, etc.), pelo operador (através das teclas do console de operação) ou pelo relógio do sistema. O **relógio** (*timer*) é um dispositivo de *hardware* que decrementa automaticamente o conteúdo de um registrador ou posição de memória, com uma freqüência constante, e interrompe a UCP quando o valor decrementado

atinge zero. O sistema operacional garante que ocorrerá pelo menos uma interrupção (e ele voltará a trabalhar) dentro de um intervalo de tempo **t**, colocando no relógio um valor que demore **t** unidades de tempo para ser decrementado até zero. Esta atribuição de valor ao relógio é feita imediatamente antes do sistema operacional entregar a UCP para um programa de usuário.

Uma interrupção não afeta a instrução que está sendo executada pela UCP no momento em que ela ocorre: a UCP detecta interrupções apenas após o término da execução de uma instrução (e antes do início da execução da instrução seguinte).
Os computadores possuem instruções para **mascarar** (desabilitar, inibir) o sistema de interrupções. Enquanto as interrupções estão mascaradas elas podem ocorrer, mas não são sentidas pelo processador. Neste caso, as interrupções ficam pendentes (enfileiradas) e só serão sentidas quando uma instrução que desmascara as mesmas é executada.

Conforme já foi dito, os *traps* são instruções especiais que, quando executadas, originam ações idênticas às que ocorrem por ocasião de uma interrupção. Pode-se dizer que um *trap* é uma interrupção prevista, programada no sistema pelo próprio programador. Uma interrupção, por outro lado, é completamente imprevisível, ocorrendo em pontos que não podem ser pré-determinados.

Os *traps* têm a finalidade de permitir aos programas dos usuários a passagem do controle da execução para o sistema operacional. Por esse motivo também são denominados “chamadas do supervisor” ou “chamadas do sistema” (*supervisor call* ou *system call*). Os *traps* são necessários principalmente nos computadores que possuem **instruções protegidas** (privilegiadas). Nesses computadores o registrador (palavra) de estado do processador possui um *bit* para indicar se a UCP está em **estado privilegiado** (estado de sistema, estado de supervisor, estado mestre) ou não privilegiado (estado de usuário, estado de programa, estado escravo). Sempre que ocorre uma interrupção ou *trap*, o novo valor carregado no registrador do estado do processador, indica estado privilegiado de execução. No estado de supervisor qualquer instrução pode ser executada e no estado de usuário apenas as instruções não

protegidas podem ser executadas. Exemplos de instruções protegidas são instruções para desabilitar e habilitar interrupções e instruções para realizar operações de E/S. Operações que envolvam o uso de instruções protegidas só podem ser executadas pelo sistema operacional, portanto. Quando um programa de usuário necessita executar alguma dessas operações, o mesmo deve executar um *trap*, passando como argumento o número que identifica a operação que está sendo requerida.

6 Dispositivos de Entradas e Saídas

Conforme vimos no capítulo relativo a componentes, o usuário se comunica com o núcleo do computador (composto por UCP e memória principal) através de dispositivos de entrada e saída (dispositivos de E/S ou *I/O devices*). Os tópicos a seguir vão analisar como funcionam os dispositivos de entrada e saída e como se faz a comunicação entre eles e o núcleo do computador.

Os dispositivos de entrada e saída tem como funções básicas:

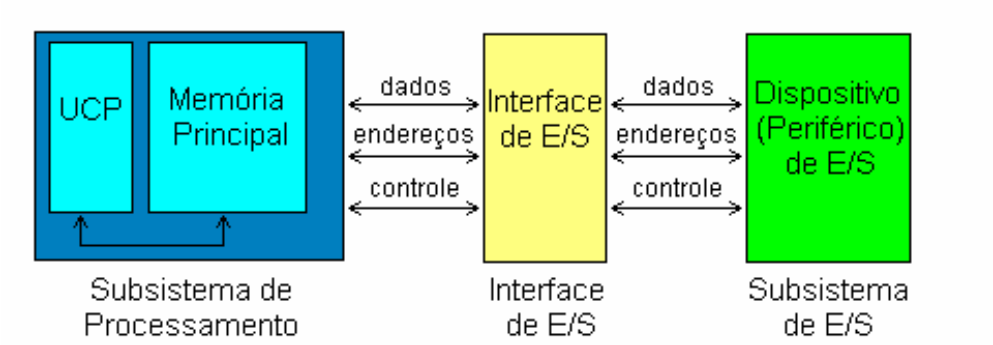
- a comunicação do usuário com o computador
- a comunicação do computador com o meio ambiente (dispositivos externos a serem monitorados ou controlados)
- armazenamento (gravação) de dados.

As características que regem a comunicação de cada um dos dispositivos de E/S (entrada e saída) com o núcleo do computador (composto de UCP e memória principal) são muito diferentes entre si. Cada dispositivo de E/S se comunica com o núcleo de forma diversa do outro. Entre outras diferenças, os dispositivos de entrada e saída são muito mais lentos que o computador, característica essa que impõe restrições à comunicação, de vez que o computador precisaria esperar muito tempo pela resposta do dispositivo. Outra diferença fundamental diz respeito às características das ligações dos sinais dos dispositivos.

Os primeiros computadores, especialmente os de pequeno porte, eram muito lentos e os problemas de diferença de velocidade eram resolvidos sem dificuldade e não representavam problema importante. Dessa forma, a ligação dos dispositivos de E/S era feita através de circuitos simples (as *interfaces*) que apenas resolviam os aspec-

tos de compatibilização de sinais elétricos entre os dispositivos de E/S e a UCP. Os aspectos relativos a diferenças de velocidade (especialmente tempo de acesso e *throughput*) eram resolvidas por programa (isto é, por *software*).

Entre esses componentes, trafegam informações relativas a dados, endereços e controle.



6.1 Tipos de Dispositivos

Os dispositivos de ENTRADA são: teclado, *mouses*, *scanners*, leitoras óticas, leitoras de cartões magnéticos, câmeras de vídeo, microfones, sensores, transdutores.

As funções desses dispositivos são coletar informações e introduzir as informações na máquina, converter informações do homem para a máquina e vice-versa, e recuperar informações dos dispositivos de armazenamento.

Os dispositivos de SAÍDA são: impressoras, monitores de vídeo, *plotters*, atuadores, chaves, etc.

As funções desses dispositivos são exibir ou imprimir os resultados do processamento, ou ainda controlar dispositivos externos.

A UCP não se comunica diretamente com cada dispositivo de E/S e sim com "*interfaces*", de forma a compatibilizar as diferentes características. O processo de comunicação ("protocolo") é feito através de transferência de informações de controle, endereços e dados propriamente ditos. Inicialmente, a UCP interroga o dispositivo, enviando o endereço do dispositivo e um sinal dizendo se quer mandar ou receber dados através da *interface*. O periférico, reconhecendo seu endereço, responde quando está pronto para receber (ou enviar) os dados. A UCP então transfere (ou recebe)

os dados através da interface, e o dispositivo responde confirmando que recebeu (ou transferiu) os dados (*acknowledge* ou *ACK*) ou que não recebeu os dados, neste caso solicitando retransmissão (*not-acknowledge* ou *NAK*).
As interfaces de entrada e saída são conhecidas por diversos nomes, dependendo do fabricante:

Interface de E/S = Adaptador de Periférico, Controladora de E/S, Processador de Periférico, Canal de E/S

Por exemplo, os computadores de grande porte da IBM chamam de *"I/O channel*". Na CDC, o nome é *Peripheral Processor Unit* ou PPU.

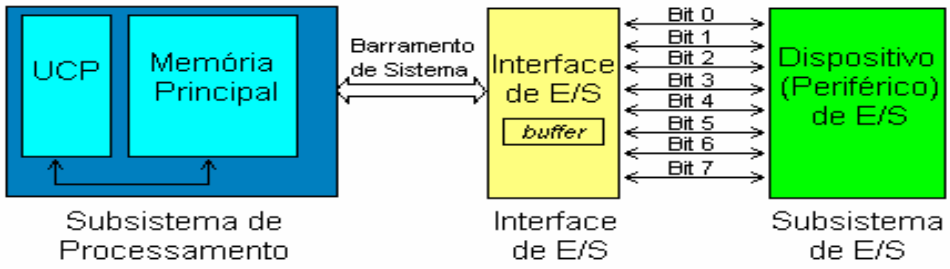
A compatibilização de velocidades é feita geralmente por programa, usando memórias temporárias na *interface* chamadas *"buffers"* que armazenam as informações conforme vão chegando da UCP e as libera para o dispositivo à medida que este as pode receber.

6.2 Formas de Comunicação

De uma forma geral, a comunicação entre o núcleo do computador e os dispositivos de E/S poderia ser classificada em dois grupos: comunicação paralela ou serial.

6.2.1 Comunicação em Paralelo

Na comunicação em paralelo, grupos de bits são transferidos simultaneamente (em geral, byte a byte) através de diversas linhas condutoras dos sinais. Desta forma, como vários bits são transmitidos simultaneamente a cada ciclo, a taxa de transferência de dados ("*throughput*") é alta.



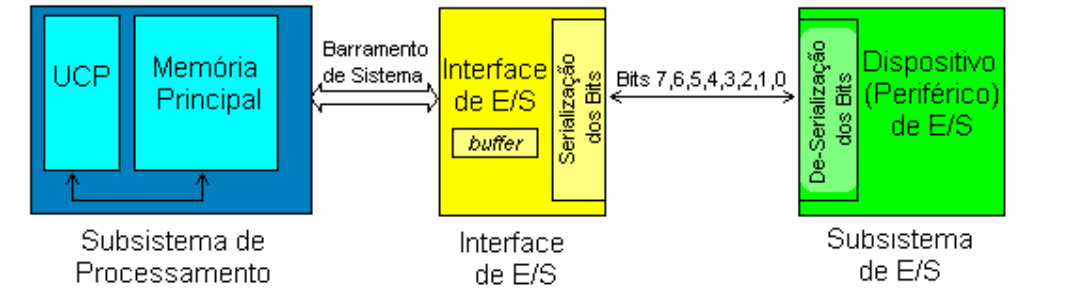
No entanto, o processo de transferência em paralelo envolve um controle sofisticado e é razoavelmente complexo, o que o torna mais caro. Um dos problemas importantes diz respeito à propagação dos sinais no meio físico, isto é, no cabo de conexão entre o dispositivo e a *interface*. Essa propagação deve se fazer de modo que os sinais (os bits) correspondentes a cada *byte* cheguem simultaneamente à extremidade oposta do cabo, onde então serão re-agrupados em *bytes*. Como os condutores que compõem o cabo usualmente terão pequenas diferenças físicas, a velocidade de propagação dos sinais digitais nos condutores poderá ser ligeiramente diferente nos diversos fios. Dependendo do comprimento do cabo, pode ocorrer que um determinado fio conduza sinais mais rápido (ou mais lento) que os demais fios e que desta forma um determinado bit *x* em cada *byte* se propague mais rápido e chegue à extremidade do cabo antes que os outros *n-1* bits do *byte*. Este fenômeno é chamado *skew,* e as conseqüências são catastróficas: os bits *x* chegariam fora de ordem (os *bytes* chegariam embaralhados) e a informação ficaria irrecuperável. Em decorrência desse problema, há limites para o comprimento do cabo que interliga um dispositivo ao computador, quando se usa o modo paralelo.

As restrições citadas contribuem para que a utilização da comunicação em paralelo se limite a aplicações que demandem altas taxas de transferência, normalmente associadas a dispositivos mais velozes tais como unidades de disco, ou que demandem altas taxas de transferência, como CD-ROM, DVD, ou mesmo impressoras, e que se situem muito próximo do núcleo do computador. Em geral, o comprimento dos cabos paralelos é limitado a até um máximo de 1,5 metro.

6.2.2 Comunicação Serial

Na comunicação serial, os bits são transferidos um a um, através de um único par condutor. Os *bytes* a serem transmitidos são serializados, isto é, são "desmontados" bit a bit, e são individualmente transmitidos, um a um. Na outra extremidade do condutor, os bits são contados e quando formam 8 bits, são remontados, reconstituindo os *bytes* originais. Nesse modo, o controle é comparativamente muito mais simples que no modo paralelo e é de implementação mais barata. Como todos os bits são

transferidos pelo mesmo meio físico (mesmo par de fios), as eventuais irregularidades afetam todos os bits igualmente. Portanto, a transmissão serial não é afetada por irregularidades do meio de transmissão e não há *skew*. No entanto, a transmissão serial é intrinsecamente mais lenta (de vez que apenas um bit é transmitido de cada vez).



Como os bits são transmitidos seqüencialmente um a um, sua utilização é normalmente indicada apenas para periféricos mais lentos, como por exemplo teclado, *mouse*, etc. ou quando o problema da distância for mandatário, como nas comunicações a distâncias médias (tal como em redes locais) ou longas (comunicações via linha telefônica usando *modems*).

Obs.: Comparativamente, a transmissão serial tem recebido aperfeiçoamentos importantes (seja de protocolo, de *interface* e de meio de transmissão) que vem permitindo o aumento da velocidade de transmissão por um único par de fios, cabo coaxial ou de fibra ótica. Como o aumento da velocidade de transmissão em interfaces paralelas ocasiona mais *skew*, a tendência tem sido no sentido do aperfeiçoamento das interfaces seriais que hoje permitem taxas de transferência muito altas com relativamente poucas restrições de distância. Em microcomputadores, a *interface USB - Universal Serial Bus* permite hoje ligar até 128 dispositivos a taxas muito altas (centenas de kbps).

6.2.3 Tabela Comparativa

Característica	**Paralelo**	**Serial**
Custo	maior	menor
Distância	curta	sem limite

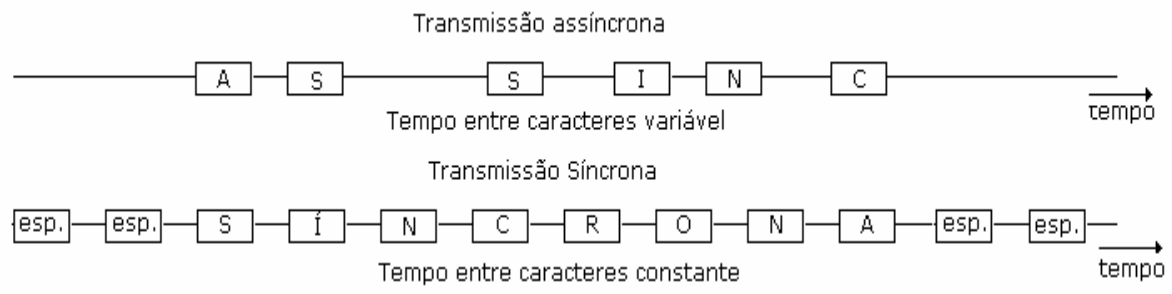
Throughput	alto	baixo

6.2.4 Transmissão Síncrona e Assíncrona

A transmissão de caracteres através de uma linha de comunicação pode ser feita por dois diferentes métodos: transmissão síncrona e assíncrona.

6.2.4.1 Transmissão Síncrona

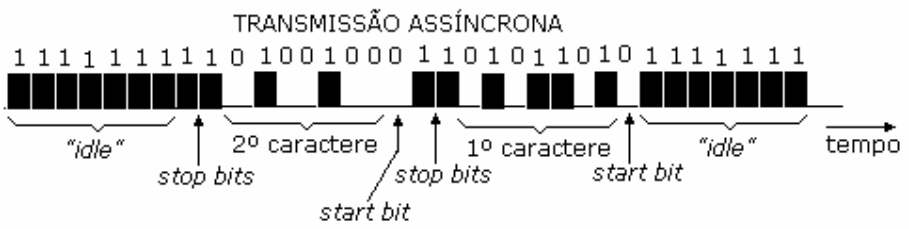
Na **transmissão síncrona**, o intervalo de tempo entre dois caracteres subseqüentes é fixo. Nesse método, os dois dispositivos - transmissor e receptor - são sincronizados, pois existe uma relação direta entre tempo e os caracteres transferidos. Quando não há caracteres a serem transferidos, o transmissor continua enviando caracteres especiais de forma que o intervalo de tempo entre caracteres se mantém constante e o receptor mantém-se sincronizado. No início de uma transmissão síncrona, os relógios dos dispositivos transmissor e receptor são sincronizados através de um *string* de sincronização e então mantém-se sincronizados por longos períodos de tempo (dependendo da estabilidade dos relógios), podendo transmitir dezenas de milhares de bits antes de terem necessidade de re-sincronizar.



6.2.4.2 Transmissão Assíncrona

Já na **transmissão assíncrona**, o intervalo de tempo entre os caracteres não é fixo. Podemos exemplificar com um digitador operando um terminal, não havendo um fluxo homogêneo de caracteres a serem transmitidos. Como o fluxo de caracteres não é homogêneo, não haveria como distinguir a ausência de bits sendo transmitidos de um eventual fluxo de bits zero e o receptor nunca saberia quando virá o próximo caractere, e portanto não teria como identificar o que seria o primeiro bit do caractere. Para resolver esses problemas de transmissão assíncrona, foi padronizado que na

ausência de caracteres a serem transmitidos o transmissor mantém a linha sempre no estado 1 (isto é, transmite ininterruptamente bits 1, o que distingue também de linha interrompida). Quando for transmitir um caractere, para permitir que o receptor reconheça o início do caractere, o transmissor insere um bit de partida (*start bit*) antes de cada caractere. Convenciona-se que esse *start bit* será um bit zero, interrompendo assim a seqüência de bits 1 que caracteriza a linha livre (*idle*). Para maior segurança, ao final de cada caractere o transmissor insere um (ou dois, dependendo do padrão adotado) bits de parada (*stop bits*), convencionando-se serem bits 1 para distingui-los dos bits de partida. Os bits de informação são transmitidos em intervalos de tempo uniformes entre o *start bit* e o(s) *stop bit(s)*. Portanto, transmissor e receptor somente estarão sincronizados durante o intervalo de tempo entre os bits de *start* e *stop*. A transmissão assíncrona também é conhecida como *"start-stop"*.

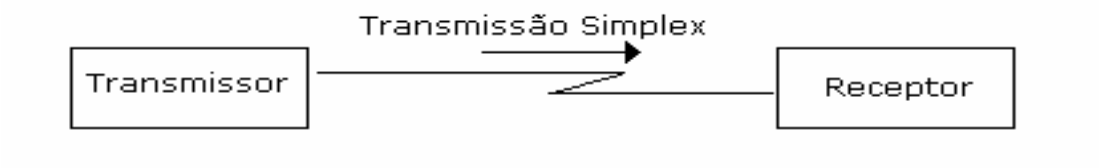


A taxa de eficiência de uma transmissão de dados é medida como a relação de número de bits úteis dividido pelo total de bits transmitidos. No método assíncrono, a eficiência é menor que a no método síncrono, uma vez que há necessidade de inserir os bits de partida e parada, de forma que a cada caractere são inseridos de 2 a 3 bits que não contém informação.

6.2.5 Transmissão Simplex, Half-Duplex e Full-Duplex

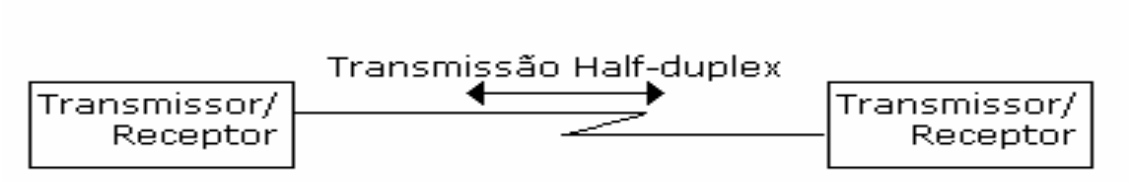
6.2.5.1 Transmissão Simplex

Uma comunicação é dita *simplex* quando permite comunicação apenas em um único sentido, tendo em uma extremidade um dispositivo apenas transmissor (*transmitter*) e do outro um dispositivo apenas receptor (*receiver*). Não há possibilidade do dispositivo receptor enviar dados ou mesmo sinalizar se os dados foram recebidos corretamente. Transmissões de rádio e televisão são exemplos de transmissão *simplex*.



6.2.5.2 Transmissão Half-Duplex

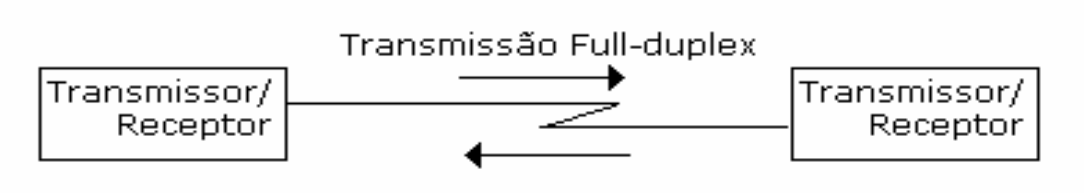
Uma comunicação é dita *half-duplex* (também chamada *semi-duplex*) quando existem em ambas as extremidades dispositivos que podem transmitir e receber dados, porém não simultaneamente. Durante uma transmissão *half-duplex*, em determinado instante um dispositivo A será transmissor e o outro B será receptor, em outro instante os papéis podem se inverter. Por exemplo, o dispositivo A poderia transmitir dados que B receberia; em seguida, o sentido da transmissão seria invertido e B transmitiria para a informação se os dados foram corretamente recebidos ou se foram detectados erros de transmissão. A operação de troca de sentido de transmissão entre os dispositivos é chamada de *turn-around* e o tempo necessário para os dispositivos chavearem entre as funções de transmissor e receptor é chamado de *turn-around time*.



6.2.5.3 Transmissão Full-Duplex

Uma transmissão é dita *full-duplex* (também chamada apenas *duplex*) quando dados podem ser transmitidos e recebidos simultaneamente em ambos os sentidos. Poderíamos entender uma linha *full-duplex* como funcionalmente equivalente a duas linhas *simplex*, uma em cada direção. Como as transmissões podem ser simultâneas em ambos os sentidos e não existe perda de tempo com *turn-around*, uma linha *full-*

duplex pode transmitir mais informações por unidade de tempo (maior *throughput*) que uma linha *half-duplex,* considerando-se a mesma taxa de transmissão de dados.



6.3 Dispositivos de Entrada e Saída

6.3.1 Teclado

O teclado é um dispositivo de entrada de dados composto de um conjunto de teclas, associadas aos caracteres utilizados para escrita e para controle (letras, algarismos, sinais de pontuação, teclas de movimentação de cursor, teclas de função, etc).

A parte visível do teclado é o conjunto de teclas. Por baixo das teclas, existe uma matriz de condutores que, quando uma tecla é pressionada, fecha contato entre dois de seu condutores, de forma que um processador (processador de teclado) possa identificar qual tecla foi pressionada. Uma vez identificada a tecla, esta informação é codificada e enviada para o processador principal do computador.

São utilizados mais usualmente dois códigos: **ASCII** (American Standard Code for Information Interchange), o mais utilizado, inclusive em microcomputadores, ou **EBCDIC** (Extended Binary Coded Decimal Interchange Code), usado pela IBM em máquinas de grande porte.

A codificação é feita em duas fases:

- 1ª fase: identificação da tecla e interpretação pelo software de controle do teclado (parte da BIOS).
- 2ª fase: conversão do código identificador da tecla para ASCII ou EBCDIC.

6.3.2 Monitor de Vídeo

O monitor de vídeo é um dispositivo de saída que utiliza uma tela semelhante à de TV como meio de visualização das informações processadas pelo computador. Também são utilizados monitores com tela de cristal líquido em microcomputadores portáteis (*laptops, notebooks, hand-helds*, etc). A informação relativa à imagem que deve ser exibida é gerada no computador e transmitida (em formato digital, isto é, bits) para a *interface* de vídeo, onde os sinais analógicos de vídeo que vão formar a imagem propriamente dita são produzidos.

Os monitores em geral tem suas telas de imagem construídas a partir de um CRT - Tubo de Raios Catódicos (nos microcomputadores portáteis são geralmente usadas telas de cristal líquido).Cada ponto da imagem precisa ser "impresso" na tela. Isso é conseguido iluminando individualmente todos os pontos, um de cada vez, ponto por ponto, linha por linha, do início ao fim da tela, então de volta ao início e assim sucessivamente, ininterruptamente, sem parar. Como os pontos iluminados esmaecem após alguns instantes, o computador (o processador) precisa ficar constantemente re-enviando a mesma imagem (ou imagens modificadas) para a *interface* que por sua vez renova a informação de imagem ("refresca" a tela).

6.3.2.1 Tipos de Monitor e Modo de Exibição

Os monitores eram inicialmente utilizados para exibir apenas caracteres (modo caractere ou modo alfanumérico) em uma única cor (geralmente um fósforo verde, algumas vezes branco ou ainda laranja). Dessa forma, o que trafegava na *interface* entre computador e monitor eram apenas códigos em bits (geralmente ASCII) que representavam os caracteres que seriam exibidos. Na *interface* esses códigos digitais eram decodificados e transformados em sinais analógicos (sinais de vídeo) com os pontos que formariam cada caractere. Cada caractere possuía poucos atributos, podendo apenas destacar brilho, exibir piscante (*"blink"*) e reverso. Cada caractere requer apenas 7 bits no código ASCII (ou 8 bits, no ASCII estendido) mais um bit para cada atributo (brilho normal x realçado, normal ou piscante, normal ou reverso).

Posteriormente, foram desenvolvidos monitores gráficos (*pixel oriented*) em cores.

Nesses monitores, a imagem passou a ser constituída, não mais por caracteres de uma só cor que podiam ser tratados como códigos ASCII, mas agora por pontos individualmente produzidos e transmitidos para a tela e que vistos em conjunto formam a imagem gráfica. Cada um desses pontos (chamados *pixels - picture elements*) passou a ter diversos atributos, entre eles a cor. Cada cor exibida precisa ser identificada por um código, bem como pelos bits de atributo (um bit por atributo para cada ponto). Considerando apenas o atributo de cor, se tivermos 16 cores, serão necessários $16 = 2^4$ códigos e portanto serão necessários 4 bits para identificá-las individualmente. Sendo 256 cores, serão $256 = 2^8$ portanto 8 bits e assim por diante, até a chamada *"true color"* com 64 milhões = 2^{32} cores exigindo 32 bits.

Também em termos de resolução (número de pontos de imagem por tela) as exigências cresceram muito. Quanto mais *pixels* maior resolução, mas também maior número de bits a serem transmitidos em cada tela. A quantidade de informações que passou a trafegar entre computador e monitor aumentou de forma extraordinária, exigindo novas soluções de projeto para evitar que a exibição de informações na tela se transformasse em um "gargalo" (*bottleneck*) para o desempenho do sistema. A solução para esse problema veio com o desenvolvimento de *interfaces* mais elaboradas, possibilitando maior taxa de transmissão de informações (*throughput*), bem como pela utilização de verdadeiros *processadores de imagem* (*interfaces* dotadas de memória local e de processadores especializados para processamento gráfico). Dessa forma, o computador passou a transmitir *primitivas gráficas* (informações codificadas que eram transformadas em imagem gráfica definida em *pixels* apenas no processador gráfico da *interface*). O processo de *refresh* também passou a ser atribuição somente do processador de vídeo, não havendo necessidade do processador principal (o processador do computador) re-enviar uma imagem que não sofresse alterações. Mais ainda: o processo de envio das modificações de uma imagem passou a ser feito por diferença, isto é, o processador principal transmite apenas *o que mudou* e o processador de vídeo se encarrega de alterar a imagem de acordo.

De uma forma bastante simplificada, podemos calcular aproximadamente quantos *bytes* devem ser transferidos entre computador e *interface* para carregar uma determinada tela, pela seguinte expressão:

Modo caractere: nº de colunas x nº de linhas x nº de *bytes* por caractere
Modo gráfico: nº de colunas x nº de linhas x nº de *bytes* por *pixel*

No cálculo a seguir apresentado como exemplo, no número de bits por caractere ou por *pixel* foi considerado (por simplicidade) apenas o atributo *cor.* O padrão VGA possui diversos outros atributos, entre eles diversos "modos" que definem número de cores, modo alfanumérico ou gráfico, etc, que não serão considerados nessa discussão.

Informações transferidas entre computador e *interface*

Monitores Modo Caractere	**Nº colunas**	**Nº linhas**	**Nº caracter**	**Nº cores**	**Bytes transferidos**
monocromático 80 colunas	80	25	2.000	$2 = 2^1$	2 kbytes
monocromático 132 colunas	132	25	3.300	$2 = 2^1$	3,2 kbytes
Monitores Gráficos	**Nº colunas**	**Nº linhas**	**Nº *pixels***	**Nº cores**	**Bytes transferidos**
CGA	320	200	64 k	$4 = 2^2$	32 kbytes
EGA	640	350	218,75 k	$16 = 2^4$	110 kbytes
VGA	640	480	300 k	$16 = 2^4$	150 kbytes
SVGA	800	600	468,75 k	$256 = 2^8$	468,75 kbytes
SVGA	1024	768	768 k	$64\ k = 2^{16}$	1,5 Mbytes
SVGA	1280	1024	1,25 M	$64\ M = 2^{32}$	5 Mbytes

6.3.3 Impressoras

Impressoras são dispositivos de saída que tem por finalidade imprimir em papel ou filme plástico os resultados do processamento. Da mesma forma que os monitores, a imagem impressa é resultado de muitos pontos impressos individualmente que no conjunto formam o texto ou a imagem desejados. Também de forma semelhante aos monitores, as impressoras evoluíram a partir de dispositivos que imprimiam apenas caracteres em uma única cor para as modernas impressoras capazes de reproduzir imagens sofisticadas, de alta resolução gráfica, em milhares de cores.

6.3.3.1 Impressoras Alfanuméricas

Esses equipamentos recebem do computador códigos que representam caracteres alfanuméricos e portanto tem capacidade de imprimir apenas esses caracteres. Geralmente é possível usar apenas uma fonte gráfica, característica do equipamento. Algumas impressoras permitem trocar o dispositivo de impressão, viabilizando a utilização de um pequeno número de fontes gráficas.

6.3.3.2 Impressoras Gráficas

Esses equipamentos recebem do computador a informação sobre os pontos a serem impressos. Dessa forma, podem imprimir gráficos. Na impressão de textos, os caracteres são impressos como pontos, que em determinada configuração formam a imagem gráfica do caractere a ser impresso. Quando se utiliza uma impressora gráfica para imprimir texto, existe a possibilidade de utilizar um grande número de diferentes fontes gráficas, definidas por *software*.

6.3.3.3 Impressora de Esfera e Impressora Margarida ("daisy wheel") - Caracter

- baixa velocidade
- velocidade de impressão de 20 cps a 45 cps
- utilizam tecnologia derivada das máquinas de escrever
- tinham preço relativamente acessível, mas hoje estão obsoletas
- usadas em sistemas de microcomputadores

6.3.3.4 Impressoras de Tambor - Linha

- maior velocidade de impressão
- imprime de 80 a 132 caracteres simultaneamente
- unidade de medida de velocidade: 1pm (linhas por minuto)
- usadas em ambientes de grande porte

6.3.3.5 Impressoras Matriciais - Impacto

- com 9 ou 24 agulhas (80 a 400 cps)
- baixa definição gráfica (até 300 dpi)

- baixa velocidade
- permitem uso de papel carbonato, viabilizando múltiplas cópias
- estão obsoletas, reduzidas hoje às aplicações que requerem múltiplas cópias

6.3.3.6 Impressoras de Jato de Tinta

- média resolução gráfica (até cerca de 1200 dpi)
- baixa velocidade
- permite cartuchos de tinta de várias cores, viabilizando a utilização de cor
- baixo custo

6.3.3.7 Impressoras Laser

- 4 a 7 ppm - impressoras de microcomputadores
- 20.000 ppm - impressoras de computadores de grande porte
- alta definição gráfica (de 600 até 4800 dpi)
- hoje já estão disponíveis modelos com recurso de cor.

6.3.4 Fita Magnética

Unidades de fita magnética são dispositivos de armazenamento de massa (isto é, usados para armazenar grandes volumes de informação). As unidades de fita são constituídas basicamente de um dispositivo de transporte (para a movimentação da fita) e das cabeças magnéticas (que executam a gravação e leitura das informações na fita), além da eletrônica de controle. A fita propriamente dita é uma fina superfície contínua feita em material plástico flexível, revestido de material magnetizável.

Unidades de Fita são dispositivos de acesso seqüencial. Essa é uma das principais razões para que as unidades de fita sejam muito lentas. As fitas magnéticas são usadas principalmente como meio de armazenamento *off-line* (para aplicação fora do processamento). Atualmente, utiliza-se discos magnéticos durante o processamento e a fita para armazenamento posterior de dados, geralmente para gerar cópias de segurança (cópias de *back-up*). Desta forma, elas não interagem com o processador durante a execução do programa, evitando o desperdício de tempo inerente à sua lentidão.

As maiores vantagens das fitas são o baixo custo e a portabilidade da mídia, proporcionando um baixo custo por *byte* armazenado. Em compensação, suas maiores desvantagens são a lentidão, a baixa confiabilidade da mídia e a pouca duração da gravação. O baixo custo por *byte* armazenado ainda mantém um mercado para utilização da fita hoje, embora venha sendo aceleradamente substituída por meios de armazenamento mais modernos, especialmente meios ótico-magnéticos.

Os **comprimentos de fita** mais utilizados são: 300, 600, 1200 e 2.400 pés.
Densidade pode ser definido como "quantos caracteres podem ser armazenados por unidade de comprimento da fita" e é medida em bpi (*bytes* por polegada). Por ex: 800, 1.600 ou 6.250 bpi.

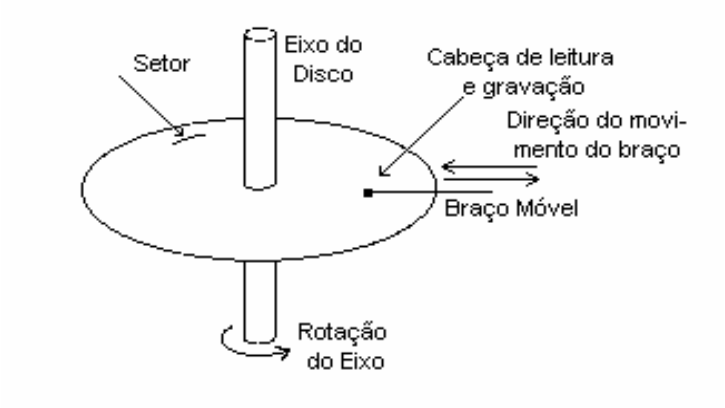
6.3.4.1 Tipos de Fitas

> *Streamer* - pequena, parecida com uma fita cassete
> DAT (*Digital Audio Tape*) - grande capacidade, menor que uma fita cassete
> Fitas Cartucho - grande densidade: 30.000 bpi
> Rolo ou carretel

6.3.5 Discos Magnéticos

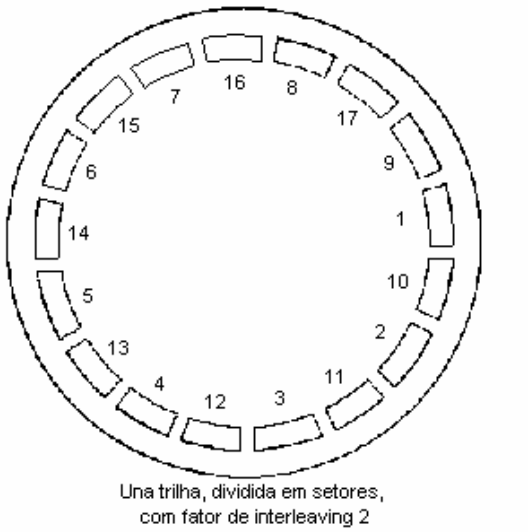
Discos magnéticos são dispositivos para armazenamento de dados (que independem de alimentação de energia e portanto permanecem gravados após ser desligado o computador, mas que podem, a critério do usuário, ser apagados ou alterados). Os discos magnéticos englobam os discos flexíveis ou *disquetes* ("*floppy disks*") e os discos rígidos.

Um disco magnético incorpora eletrônica de controle, motor para girar o disco, cabeças de leitura / gravação e o mecanismo para o posicionamento das cabeças, que são móveis. Os discos propriamente ditos são superfícies de formato circular, compostos de finos discos de alumínio ou vidro, revestidas de material magnetizável em ambas as faces.
O desenho a seguir ilustra um disco:



6.3.5.1 Organização Física da Informação nos Discos

As informações são gravadas nos discos em "setores", distribuídos ao longo de "trilhas" concêntricas marcadas magneticamente como setores circulares no disco, conforme ilustração a seguir.



Una trilha, dividida em setores, com fator de interleaving 2

O processo de marcação magnética das trilhas e setores em um disco faz parte da "*formatação*" do disco. Esta formatação é dependente do sistema operacional que usará o disco. O sistema operacional DOS define que cada setor armazena 512 *bytes*. Todas as trilhas armazenam o mesmo número de *bytes*; desta forma, os dados na trilha mais interna estarão gravados com maior densidade, pois o espaço físico é menor.

6.3.5.2 Tempo de Acesso

O tempo de acesso aos dados de um disco é definido como o período decorrido entre a ordem de acesso e o final da transferência dos dados. O tempo de acesso não é constante, variando em função da posição relativa entre o braço atuador (que posiciona as cabeças de leitura e gravação) e o setor que será lido e portanto só tem sentido falar em **tempo médio de acesso**. Os tempos médios de acesso dos discos atuais são da ordem de 10 ms e são resultado das seguintes operações:
TEMPO DE ACESSO = TEMPO DE *(SEEK* + LATÊNCIA + TRANSFERÊNCIA)

6.3.5.3 Tempo de Seek

Seek ou busca é o tempo gasto na interpretação da localização do dado no disco (endereço do dado no disco) pela unidade de controle e no movimento mecânico do braço que sustenta a cabeça magnética, até alcançar a trilha desejada. Este tempo é variável de acesso para acesso. os tempos típicos de discos rígidos atuais podem variar de aproximadamente 0 ms (tempo de *seek* mínimo, referente ao acesso a um setor localizado na mesma trilha onde no momento está a cabeça de leitura), 3 ms (para acesso a setores em trilhas adjacentes) a até 20 ms (tempo de busca máximo, referente ao acesso entre trilhas localizadas nas extremidades do disco). Este tempo é diretamente dependente da qualidade dos mecanismos eletromecânicos que comandam os braços atuadores. Discos de menores dimensões também tendem a ser mais rápidos, pois o percurso linear dos braços atuadores é menor.

6.3.5.4 Tempo de Latência

É o tempo de latência (também chamada latência rotacional) é o tempo gasto entre a chegada da cabeça de leitura / gravação sobre a trilha e a passagem do setor desejado na posição da cabeça. Como o disco permanece constantemente girando, a cabeça magnética só pode ler ou gravar um dado quando o setor endereçado está imediatamente abaixo dela. Portanto, há que aguardar que o disco gire até que o setor endereçado fique posicionado abaixo da cabeça. Esse tempo depende diretamente da velocidade com que o disco gira. Discos rígidos atuais em geral giram a 5.400 rpm (rotações por minuto), o que significa que um giro completo se realiza em

1 ÷ (5400rpm ÷ 60s) = 11ms. Portanto, o tempo de latência poderá variar entre quase 0 ms (se o setor estiver exatamente na posição certa para ser lido) até 11 ms (se for necessário aguardar uma volta completa em um disco).

6.3.5.5 Tempo de Transferência

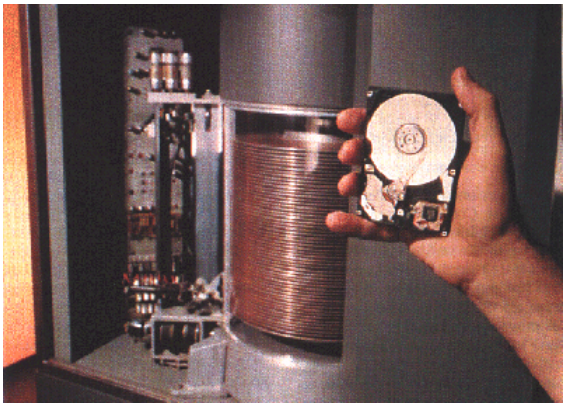
É o tempo consumido na transmissão dos bits entre computador e disco e vice-versa. Este tempo depende da *interface* e do disco, que definem o *throughput* (taxa de transferência) do disco. Atualmente, dependendo da *interface*, o *throughput* seria da ordem de até 33 Mbytes/s. Como um setor tem 512 bytes, em 1 ms se poderia transferir cerca de 33 setores e o tempo de transferência de um setor seria da ordem de 15 ns.

Obs.: Os tempos relativos a unidades (*drives*) de *disquetes* são muito maiores que os acima indicados para discos rígidos. *Drives* de *disquete* giram a aproximadamente 300 rpm e o *throughput* é da ordem de 500 kbytes/s; os tempos de acesso médios são da ordem de 60 a 100 ms.

6.3.6 Discos Rígidos

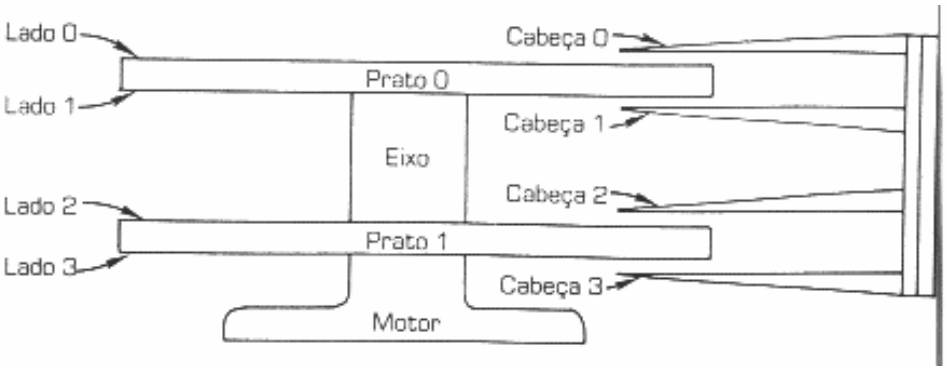
Os primeiros discos rígidos selados para microcomputadores foram projetados e construídos na fábrica da IBM localizada em Winchester. Alguns autores atribuem a isso o motivo deles terem sido apelidados de *Winchester drives* (unidades de disco *Winchester)*, denominação que perdurou por muito tempo, até que a tecnologia de construção dos discos mudou. Hoje estes discos são conhecidos na literatura apenas por "*hard disks*" ou HDs (em inglês, traduzindo-se literalmente por "discos rígidos").

Obs.: Há também autores que atribuem o nome a uma analogia com os famosos rifles 30/30 Winchester. A seguir é mostrado o primeiro disco rígido (RAMAC) de 5 Mb distribuídos em 50 pratos com diâmetro de 24", desenvolvido pela IBM em 1956, nos laboratórios de Almaden (CA), ao lado de um moderno HD IBM Seascape de 5 Gb com 2.5".



A construção em forma de unidade selada, com ar filtrado eliminando as partículas de pó, permitiu que as cabeças de leitura / gravação fossem posicionadas a uma distância ínfima em relação às superfícies magnetizadas do disco, de vez que a possibilidade de impurezas que se interpusessem entre a superfície e a cabeça e pudessem riscar o disco ou danificar a cabeça foi eliminada. A proximidade entre cabeça e superfície, bem como a grande uniformidade de superfície conseguida, possibilitaram obter uma grande densidade de gravação dos dados nos discos rígidos selados. A utilização de atuadores eletromecânicos de alta precisão também permitiu reduzir o espaço entre trilhas. Os discos rígidos são pequenos e compactos, tem custo de armazenagem por Mbyte muito baixo e alto desempenho (alta taxa de transferência e pequeno tempo de acesso), oferecendo ainda segurança de armazenagem dos dados. Tudo isso permitiu a construção de discos rápidos e com alta capacidade.

Os discos rígidos atuais são construídos com muitas superfícies de gravação, montadas em torno de um eixo comum. Os braços atuadores responsáveis pelo posicionamento das cabeças de leitura / gravação são montados em uma única estrutura, de forma que os braços se movem solidariamente.



Devido a esta conformação física, podemos verificar que, conforme uma determinada cabeça é posicionada sobre uma trilha, as demais cabeças estarão também posicionadas sobre as trilhas das outras superfícies que ficam localizadas à mesma distância do eixo central e portanto todas as trilhas localizadas a uma mesma distância do eixo central do disco poderão ser acessadas simultaneamente por todas as cabeças. Um conjunto de trilhas localizadas a uma mesma distância do eixo central pode ser visto espacialmente como se fora um cilindro, e é assim que são chamados. Dessa forma, num disco de quatro cabeças (quatro faces) o cilindro 10 seria composto de todos os setores localizados na trilha 10 da face 1, na trilha 10 da face 2, na trilha 10 da face 3 e na trilha 10 da face 4. Os dados gravados no mesmo cilindro (na mesma trilha porém em superfícies diferentes) podem ser acessados (lidos ou gravados) sem que o braço atuador das cabeças de leitura / gravação tenha que ser movido (como vimos na discussão sobre tempo de acesso, esta operação é relativamente lenta!).

6.3.7 Discos Flexíveis

As unidades de discos flexíveis (*floppy disks* ou FDs) surgiram como uma solução para armazenamento de dados a baixo custo em microcomputadores e substituíram com grandes vantagens o armazenamento em fitas *cassete* que equipava os primeiros microcomputadores. Os discos flexíveis são feitos de *poliester* flexível e sua capacidade atual de armazenamento é geralmente de 1,44 Mbytes.

Obs.: Os discos flexíveis tem um orifício de índice que indica fisicamente o início das trilhas.

6.3.7.1 Cálculo do Espaço de Armazenamento em um Disco

O espaço de armazenamento num disco pode ser calculado como segue:
nº total de setores = nº de superfícies (= nº de cabeças) x nº de setores por trilha x nº de trilhas (= nº de cilindros)

Como no DOS e Windows cada setor tem 512 bytes, para obter o resultado em kbytes basta multiplicar o número de setores por 0,5 (512 bytes = 0,5 kbyte) e para obter em Mbytes, dividir então por 1.024.

Por exemplo, um *disquete* de 5 ¼" com 2 cabeças, 15 setores por trilha e 80 trilhas, teria 2 x 15 x 80 = 2.400 setores de 512 bytes, portanto a capacidade seria de 1.200 kbytes ou aproximadamente 1, 2 Mbytes. Um *disquete* de 3 ½" com 2 cabeças, 18 setores por trilha e 80 trilhas, teria 2 x 18 x 80 = 2.880 setores de 512 bytes, portanto a capacidade seria de 1.440 kbytes ou aproximadamente 1, 44 Mbytes.

7 EXECUÇÃO DE PROGRAMAS

7.1 Programa em Linguagem de Máquina

Para executar uma tarefa qualquer, um computador precisa receber instruções precisas sobre o que fazer. Uma seqüência adequada de instruções de computador, para a realização de uma determinada tarefa, se constitui num PROGRAMA de computador. Uma linguagem de programação é um conjunto de ferramentas, regras de sintaxe e símbolos ou códigos que nos permitem escrever programas de computador, destinados a instruir o computador para a realização de suas tarefas.

A primeira e mais primitiva linguagem de computador é a própria linguagem de máquina, aquela que o computador entende diretamente e pode ser diretamente executada pelos circuitos do processador (pelo *hardware*). No início da era da computação, os programas eram escritos em linguagem de máquina, isto é, as instruções eram escritas diretamente na linguagem do computador (formada apenas com 1's e 0's). Um programa em linguagem de máquina é uma longa série de 0's e 1's, ordenados de forma que alguns representam códigos de instruções e outros representam os dados que serão processados (ou indicam onde esses dados estão armazenados). Em um programa escrito em linguagem de máquina, cada instrução escrita pelo programador será individualmente executada, isto é, a cada instrução do programa corresponderá uma ação do computador. A relação é portanto 1 para 1 - uma instrução do programa corresponde a uma operação do computador.

Imagine então um programa extenso escrito apenas usando 1's e 0's; imagine que para cada diferente marca ou modelo de computador, as regras para entender esses códigos serão totalmente diferentes; e finalmente imagine que voce teria que escrever uma a uma as instruções e os dados adequadamente codificados e ordenados, perfurar todos o programa em cartões e submeter toda a massa de cartões ao computador, para finalmente receber algumas horas depois o seu programa de volta com uma mensagem de erro tipo "erro no cartão X" ... e mais nada! Um programa escrito nessa linguagem era difícil de ser escrito sem que se cometessem muitos erros, processo esse longo, difícil, entediante e principalmente caro. Um programa em linguagem de máquina era também extremamente difícil de ser entendido por outros programadores que futuramente viessem a trabalhar na manutenção do programa. Essa complexidade levou à necessidade de se desenvolverem técnicas e ferramentas para tornar a escrita e manutenção de programas mais fácil, mais rápida e principalmente mais barata.

Cada família de computadores possui sua própria linguagem de máquina. Um programa em linguagem de máquina é dependente do computador ou seja, tendo sido escrito para um determinado computador, somente poderá ser executado em computadores da mesma família, que lhe sejam 100% compatíveis.

Obs.: Um programa em linguagem de máquina pode ser apresentado em binário (0's e 1's) ou em hexadecimal (usando de 0 a F, ou seja, transformando cada 4 bits em um dígito hexadecimal). A apresentação em hexadecimal torna mais enxuta a representação (mas não mais simples...), mas serve somente para visualização, pois um programa somente pode ser submetido ao computador em binário.

7.2 Linguagem de Montagem

A primeira tentativa bem-sucedida para resolver o problema acima descrito foi a criação de uma linguagem em que os códigos numéricos foram substituidos por mnemônicos (palavras ou símbolos) como por exemplo:

LOAD = carregar e ADD = somar, que se aproximam de palavras comuns da lingua inglesa).

As localizações dos dados foram substituídas por referências simbólicas. Foram também definidas regras de sintaxe de fácil memorização, de forma a tornar a escrita de programas e sua posterior manutenção uma técnica de complexidade relativamente menor.

Essa linguagem simbólica recebeu o nome de *Assembly Language* (Linguagem de Montagem). Assim, o programador não mais precisava decorar os códigos numéricos que representavam as diferentes instruções e os endereços reais de armazenamento, bastando decorar mnemônicos para as instruções e definir nomes para as referências dos endereços (por exemplo, NOME para o local onde seriam armazenados os nomes e SALARIO para o local onde seriam armazenados os salários, etc), o que sem dúvida facilita enormemente o trabalho.

É importante lembrar que um computador é sempre monoglota, isto é, ele entende única e exclusivamente a sua própria linguagem de máquina. Portanto, para escrever um programa em outra linguagem e ele ser entendido e processado no computador, é preciso haver algum outro programa que leia o programa escrito nessa linguagem alternativa e o traduza para a linguagem nativa do computador (isto é, a linguagem de máquina entendida pelo computador).

O processo de tradução da linguagem de montagem para a linguagem de máquina (chamado de montagem) é realizado por um programa chamado *Assembler* (Montador). O programa *Assembler* lê cada instrução escrita em linguagem *Assembly* e a converte em uma instrução equivalente em linguagem de máquina; e também converte cada uma das referências simbólicas de memória em endereços reais (resolve as referências de memória).

A criação de programas Montadores facilitou muito o trabalho dos programadores. Uma outra vantagem menos óbvia foi possibilitar o desenvolvimento de programas

de crítica de sintaxe (os *debuggers*), facilitando o processo de depuração de erros de programação.

No entanto, o processo continuava lento e complexo, exigindo do programador uma grande compreensão do processo e profundo conhecimento da máquina que ele estava programando. Um programa de computador ainda era difícil de ser escrito, caro, e dependente do computador para o qual foi escrito, já que um programa escrito em linguagem de máquina para um determinado computador só poderá ser processado em computadores 100% compatíveis com ele.

7.3 Linguagem de Programação

Esses problemas levaram a uma busca por linguagens que fossem mais simples de programar e entender, mais rápidas e eficientes (levando a programas mais enxutos, com menos instruções), menos dependente do computador-alvo, mas que processassem com boa eficiência (não acarretando processamento lento no computador).

Foram desenvolvidas diversas linguagens de programação, buscando afastar-se do modelo centrado no computador. Essas linguagens foram estruturadas buscando refletir melhor os processos humanos de solução de problemas. Essas linguagens orientadas a problema são também chamadas linguagens de alto nível, por serem afastadas do nível de máquina.

As primeiras linguagens foram FORTRAN (1957), usada basicamente para manipulação de fórmulas; ALGOL (1958), para manipulação de algoritmos; COBOL (1959), para processamento comercial e ainda hoje bastante usada, especialmente em computadores de grande porte (*mainframes*) em bancos.

Nas décadas de 60 e 70, podemos citar Pascal, a primeira linguagem de alto nível estruturada; BASIC, linguagem criada para facilitar a programação por não-profissionais; e ADA, linguagem para processamento em tempo real criada sob encomenda do *DoD* (*Department of Defense* norte-americano) e ainda hoje a única linguagem aceita para programas escritos sob encomenda do *DoD*.

Na década de 80, surgiu o C e depois o C++ (com suporte a objetos), que estão entre as linguagens mais utilizadas hoje.

Cada nova linguagem criada visa atingir níveis de abstração mais altos, isto é, afastam cada vez mais o programador do nível de máquina. Se por um lado essas novas linguagens facilitam muito o trabalho do programadores (e reduzem sua necessidade de conhecer o *hardware* da máquina), elas cobram um alto preço em termos de desempenho (isto é, são cada vez mais lentas, ao consumir cada vez mais ciclos de máquina e espaço em memória). Esse aumento de exigência ao poder de processamento dos computadores é compensado (e desta forma se faz menos notado) pelo aumento acelerado do poder de processamento dos novos *chips* (exemplificado pela chamada Lei de *Moore*, que afirma que o poder de processamento dos *chips* dobra a cada 18 meses) e pelos avanços na arquitetura dos computadores.

Dentre as importantes tendências atuais, citamos as linguagens de manipulação de bancos de dados (como *dBase*, *Clipper*, *FoxPro*, *Paradox*, *Access*, etc) e as linguagens visuais, como o *Visual Basic*, *Visual C* e *Delphi*.

Tal como na linguagem humana, as linguagens de computadores proliferam e sempre há problemas que ainda persistem, continuando a busca por uma linguagem ideal - a solução "definitiva". A linguagem *Java* é a mais importante tendência atual e mais recente avanço na busca pela linguagem universal - o "esperanto" dos computadores. *Java* eleva a abstração ainda mais um nível e se propõe a ser independente da máquina onde será executado. Porém, na realidade, quando não está sendo processado em um processador *Java* nativo, o código *Java* é interpretado por uma camada de *software* chamada máquina virtual *Java* (*JVM - Java Virtual Machine*), ou seja, um emulador.

7.4 Tradução

Um programa escrito por um programador (chamado código fonte) em uma linguagem de alto nível é um conjunto de instruções que é clara para programadores, mas

não para computadores. Ou seja, os computadores entendem única e exclusivamente suas linguagens nativas, as linguagens de máquina. Programas em linguagem de alto nível, a exemplo dos programas escritos em linguagem de Montagem, também precisam ser traduzidos para linguagem de máquina para poderem ser submetidos ao computador e processados.

O processo de tradução do programa escrito em uma linguagem simbólica pelo programador, chamado código fonte (*source code*) para a linguagem de máquina do computador chamada código objeto (*object code*), é chamado compilação e é realizado por um programa chamado Compilador (*Compiler*).

7.5 Montagem

Citamos anteriormente uma forma de tradução rápida e simples: a executada pelo programa Montador. O processo de montagem traduz um programa escrito em linguagem *Assembly* em um programa equivalente em linguagem de máquina, possível de ser executado pelo computador. A seguir, é apresentado o fluxo que representa o processo de montagem.

No processo de montagem, o código fonte (programa em linguagem simbólica escrito pelo programador) é examinado, instrução por instrução e é feita a tradução, gerando o código que será executado (código objeto). Os passos executados pelo programa Montador são:

a) Verificar a correção do código de instrução (se o mnemônico corresponde a uma instrução válida para o computador, se os campos definidos na estrutura da linguagem e a sintaxe estão corretos) e substituir os mnemônicos pelos códigos numéricos binários equivalentes. Qualquer erro no código acarreta a interrupção do processo e a emissão de mensagem de erro.
b) Resolver as referências de memória: os nomes simbólicos adotados pelo programador são convertidos para endereços reais de memória (valores numéricos binários de endereços).
c) Reservar espaço em memória para o armazenamento das instruções e dados.

d) Converter valores de constantes em binário.

7.6 Compilação

Compilação é o processo de tradução de um programa escrito em linguagem de alto nível para código em linguagem de máquina. Compilação é um processo análogo ao da montagem (verificação / análise do código fonte, resolução das referências de memória, reserva de espaço em memória e conversão para código de máquina binário). O que diferencia a compilação do processo de montagem é sua maior complexidade. No processo de montagem, há uma relação de 1:1, ou seja, cada instrução do código fonte resulta em uma instrução de máquina, enquanto na compilação a relação é múltipla, cada instrução do código fonte gerando várias instruções de máquina.

Durante a compilação, o código fonte é analisado (análise léxica, sintática e semântica), é gerado um código intermediário e são construídas tabelas de símbolos, alocam-se as áreas de memória para variáveis e atribui-se os registradores a serem utilizados, e é finalmente gerado o código objeto em linguagem binária de máquina. Em alguns compiladores, é gerado um código intermediário em Assembly (que pode ser visualizado pelo programador) e que em seguida passa pelo montador para gerar finalmente o código objeto em linguagem de máquina.

O código objeto pode ser absoluto (os endereços constantes são endereços reais de memória) ou relocável (os endereços são relativos, tendo como referência o início do programa, e os endereços reais de memória são definidos apenas em tempo de execução).

7.7 Bibliotecas

O desenvolvimento de um programa certamente utilizará diversas operações que são comuns a muitos outros programas. Por exemplo, a execução de uma instrução de entrada e saída, a classificação dos dados de um arquivo, o cálculo de funções matemáticas, etc. Uma linguagem de alto nível geralmente incorpora diversas rotinas

prontas (que fazem parte da linguagem) e que compõem bibliotecas (*librarys*) de funções pré-programadas que poderão ser utilizadas pelo programador, poupando tempo, aumentando a eficiência e evitando erros. Dessa forma, um programa em alto nível possivelmente conterá diversas chamadas de biblioteca (*library calls*). Essas funções não devem ser confundidas com as instruções da linguagem - na realidade, são pequenos programas externos que são chamados através de instruções especiais de chamada de biblioteca. Para serem executadas, essas rotinas precisam ser incorporadas ao código do programador, isto é, a chamada de biblioteca precisa ser substituída pelo código propriamente dito, incluindo os parâmetros necessários.

7.8 Ligação

Assim, o código objeto preparado pelo compilador em geral não é imediatamente executável, pois ainda existe código (as rotinas de biblioteca) a ser incorporado ao programa. A cada chamada de biblioteca encontrada no código fonte, o compilador precisará incluir uma chamada para a rotina e o endereço dos dados que devam ser passados para a rotina.

A tarefa de examinar o código objeto, procurar as referências a rotinas de biblioteca (que constituem referências externas não resolvidas), buscar a rotina da biblioteca, substituir a chamada pelo código ("resolver as referências externas") e obter os parâmetros para incluí-los no código objeto é executada por um programa chamado Ligador (*LinkEditor*). O resultado da execução do Ligador é o código final pronto para ser executado pelo computador, chamado módulo de carga ou código executável.

O módulo de carga, após testado e depurado (isto é, depois de resolvidos todos os erros, também chamados *"bugs"*) é armazenado em memória de massa para ser executado quando necessário. O processo de compilação e ligação é executado apenas pelo programador na fase de desenvolvimento e não mais precisará ser executado pelo usuário, quando da execução do programa.

7.9 Interpretação

Com o processo de execução de um programa em fases distintas (compilação / ligação / execução) apresentado, um programa para ser executado precisa primeiro ter sido convertido para código objeto pelo compilador e depois ter passado pelo ligador. Esse processo é o mais largamente utilizado, porém não é o único.

O método alternativo chama-se de interpretação e, a partir do programa fonte, realiza as três fases (compilação, ligação e execução), comando por comando, em tempo de execução. Não existem fases distintas nem se produzem códigos intermediários. Todo o processo de conversão é efetuado em tempo de execução e imediatamente executado. Ou seja, cada comando é lido, verificado, convertido em código executável e imediatamente executado, antes que o comando seguinte seja sequer lido.

Linguagens como C, Pascal, COBOL, etc, são linguagens tipicamente compiladas, enquanto o BASIC foi desenvolvido como linguagem interpretada (hoje também existem linguagens BASIC compiladas e o programador pode optar). As linguagens de programação tipicamente de usuário, tais como das planilhas Excel, o Word Basic (linguagem de construção de Macros do Word), o Access, etc, são todas linguagens interpretadas.

7.10 Comparação entre Compilação e Interpretação

Sempre que houver duas opções, haverá vantagens e desvantagens para cada uma delas (pois se assim não fosse, a que apresentasse sempre desvantagem seria abandonada). Vamos comparar os métodos:

7.10.1 Tempo de Execução

No método de interpretação, cada vez que o programa for executado, haverá compilação, ligação e execução de cada um dos comandos. No método de Compilação, o tempo de execução do programa é reduzido, porque todos os passos preliminares (compilação e ligação) foram previamente cumpridos.

7.10.2 Consumo de Memória

No método de interpretação, o interpretador é um programa geralmente grande e que precisa permanecer na memória durante todo o tempo que durar a execução do programa, pois um programa necessita do interpretador para ter traduzidos cada um dos seus comandos, um a um, até o término de sua execução (o interpretador somente é descarregado depois do término da execução do programa).

No método de compilação, o compilador é carregado e fica na memória apenas durante o tempo de compilação, depois é descarregado; o ligador é carregado e fica na memória apenas durante o tempo de ligação, depois é descarregado. Essas são funções realizadas pelo programador e executadas apenas durante o desenvolvimento do programa. Quando o usuário for executar o programa, apenas o módulo de carga (código executável) é carregado e fica na memória durante a execução.
Desta forma, vemos que o método de interpretação acarreta um consumo de memória muito mais elevado durante a execução do programa.

7.10.3 Repetição de Interpretação

No método de compilação, um programa é compilado e ligado apenas uma vez, e na hora da execução é carregado apenas o módulo de carga, que é diretamente executável. No método de interpretação, cada programa terá que ser interpretado toda vez que for ser executado.

Outro aspecto é que, em programas contendo *loops*, no método de interpretação as partes de código pertencentes ao *loop* serão várias vezes repetidas e terão que ser interpretadas tantas vezes quantas o *loop* tiver que ser percorrido. No método de compilação, a tradução do código do *loop* se faz uma única vez, em tempo de compilação e ligação.

Estas características levam a um maior consumo de tempo no método de interpretação, que é portanto mais lento.

7.10.4 Desenvolvimento de Programas e Depuração de Erros

No método de compilação, a identificação de erros durante a fase de execução fica sempre difícil, pois não há mais relação entre comandos do código fonte e instruções do executável.
No método de interpretação, cada comando é interpretado e executado individualmente, a relação entre código fonte e executável é mais direta e o efeito da execução (certa ou errada) é direta e imediatamente sentido. Quando a execução de um comando acarreta erro, quase sempre o erro pode ser encontrado no comando que acabou de ser executado. Assim, o interpretador pode informar o erro, indicando o comando ou variável causador do problema.

Essa característica, que é a rigor a maior vantagem do método de interpretação sob o ponto de vista do usuário, faz com que esse método seja escolhido sempre que se pretende adotar uma linguagem que vá ser usada por não-profissionais ou para a programação mais expedita. Por exemplo, o método de interpretação é usado pela maioria dos *Basic* (uma linguagem projetada para ser usada por iniciantes), e por todas as linguagens típicas de usuário como *dBase*, *Access*, *Excel*, etc.

7.11 Emuladores e Máquinas Virtuais

Uma aplicação interessante dos interpretadores é a geração de código universal e máquinas virtuais. Sabemos que um computador somente é capaz de executar programas que tenham sido desenvolvidos para ele. Assim, um programa desenvolvido para rodar em PC's rodando *Windows* não funciona em PC's com UNIX ou em *Mcintosh*. Se voce concorda com essa afirmação, imagine então uma página na *Internet*, com textos, imagens e programas que podem ser visualizados e processados por quase qualquer computador. Como isso pode ser feito? Como computadores de marcas e modelos diferentes estarão lendo, interpretando e executando corretamente comandos que podem ter sido desenvolvidos usando um outro computador?

O segredo é a utilização de linguagens padronizadas (tais como HTML - para a es-

crita das páginas - e PERL, CGI, *Java*, *Java Script*, etc, para programas) que são suportadas por diversas plataformas. Assim, cada uma das plataformas (através dos programas visualizadores de páginas Internet, conhecidos como *browsers* ou mesmo através de seus respectivos sistemas operacionais) pode interpretar corretamente qualquer página feita e hospedada em qualquer computador.

Uma situação semelhante ocorre quando alguém desenvolve um programa que "interpreta" o código executável produzido para um determinado computador e o converte para ser executado em outro computador incompatível com o primeiro. Imagine pegar um jogo desenvolvido para os consoles originais Atari (por exemplo, o *Space Invaders*, *Asteroids* ou *PacMan*) e poder executá-lo num moderno PC. Ou pegar um programa que voce tenha escrito (por exemplo, uma planilha) para um dos *Apple II* originais e rodá-lo num PC. Esse processo, em que um computador opera como se fosse o outro, é conhecido como emulação.

Levando esse conceito um pouco mais adiante, imagine desenvolver um programa conversor que pegasse qualquer programa escrito para uma determinada máquina e interpretasse seu código executável traduzindo-o em tempo de execução para instruções de um outro computador. Esse programa criaria uma camada de emulação em que uma máquina se comportaria como uma outra máquina. Poderíamos ter um PC "virtual" emulado em um *Mcintosh*, que estaria assim apto a rodar qualquer programa escrito para PC. Esse programa emulador criaria um ambiente que chamamos de máquina virtual, isto é, uma máquina que se comporta como uma outra máquina diferente, não compatível.

Algo parecido foi desenvolvido pela *Sun Microsystems* na linguagem *Java*. *Java* é uma linguagem que em princípio permite que programas escritos nela rodem em qualquer máquina. Na realidade, a *Sun* desenvolveu uma *plataforma Java* (*JVM - Java Virtual Machine*), com características de ambiente necessárias para que os programas escritos em *Java* rodem adequadamente. A *JVM* suporta uma representação em *software* de uma UCP completa, com sua arquitetura perfeitamente definida incluindo seu próprio conjunto de instruções. Os programadores *Java* escrevem

código usando o conjunto de instruções definido pela linguagem *Java*. Esse fonte será então compilado gerando código de máquina virtual *Java*. Como o código *Java* é universal, os códigos fonte *Java* e os códigos objeto gerados são independentes da máquina em que o *software* será depois processado. Assim, os programadores *Java* não precisam se preocupar em qual computador ou sistema operacional o programa vai ser executado: desenvolver para *Java* é independente da máquina!

Mas como é possível que isso funcione, sem desmentir tudo o que antes dissemos sobre o código de máquina ser dependente do ambiente? Tudo bem que o código *Java* rode bem em um processador *Java*. Mas, e em outras plataformas? Bom, em outras plataformas, programas escritos em *Java* rodariam em máquinas virtuais que emulariam o ambiente *Java*. Isso significa que as empresas desenvolvedoras de *software* contruiram uma camada de *software* que, em tempo de execução, lêem o código *Java* e o interpretam, traduzindo-o para o código nativo daquele ambiente. Esses programas ***interpretadores*** permitem que um programa *Java* seja traduzido, em tempo de execução, para o código nativo e seja executado no computador do usuário, com maior ou menor eficiência, porém sempre mais lentos que na máquina *Java* ideal. Assim, a linguagem *Java*, que foi inicialmente recebida com grande interesse, apresenta resultados algumas vezes decepcionantes (especialmente quanto ao desempenho) e sua aplicabilidade atualmente tem sido restrita a sistemas com requisitos de desempenho não-críticos.

Comentário: A *Sun* tem investido em melhorar o desempenho do código *Java*. Além de uma série de interessantes otimizações de *software*, foi criado um *chip Java*, que poderia ser adicionado aos computadores e criaria um ambiente *Java* ideal. Ora, se a vantagem do *Java* é exatamente poder ser executado em qualquer computador, é um contra-senso ser necessário dispor de um *chip Java* para poder rodar *Java* eficientemente. Outro aspecto preocupante é o esforço da *Microsoft* para criar extensões ao Java (ou seja, criar um *Java Microsoft*, sem respeitar o padrão da *Sun*). Essa extensões visam fazer o código *Java* rodar com mais eficiência no ambiente PC / *Windows*, porém podem destruir a padronização. O sonho (mais das empresas de software que dos programadores, é verdade) de uma linguagem universal, em que se

desenvolveria para um e se rodaria em todos, parece que ainda fica para um tempo mais à frente.

8 BIBLIOGRAFIA E LEITURAS AUXILIARES

Livro Recomendado

Introdução à Organização de Computadores (2ª ou 3ª edições)

Mario Monteiro - Ed. LTC

Livro de Apoio

Organização Estruturada de Computadores (3ª edição)

Andrew Tanembaum - Ed. Prentice-Hall do Brasil

Literatura Auxiliar

Computer Organization and Architecture

William Stallings - Mcmillam Publishing Company

Revistas

BYTE Brasil

PCMagazine Brasil

Windows Computing (Brasil)

Info EXAME (Brasil)

PCWorld (Brasil)

ZDNet (Brasil)

BYTE (USA)

PCMagazine (USA)

ZDNet (USA)

Todas as revistas e cadernos de jornais publicados no Brasil dão ênfase aos micro-computadores PC. A BYTE (tanto a edição brasileira quanto a americana) tem conteúdo mais técnico. A PC Magazine apresenta muitos testes comparativos de produtos. Windows Computing apresenta muitos artigos sobre software relativo ao Windows. A PC World é mais voltada para o usuário final (usuário não técnico). A Info

EXAME tem maior ênfase em gerenciamento (o público típico da EXAME). Existem ainda revistas com cobertura específica sobre Internet, que podem ser recomendadas aos interessados. No Brasil, Internet World e .Net são as principais; nos EUA, leiam a Wired.

O Globo - Informática etc. - publicado às 2ª feiras

Jornal do Brasil - Caderno Informática - publicado às 2ª feiras

Dicionário

FOLDOC - The Free On-Line Dictionary Of Computing

Enciclopédia sobre computadores

PC Webopaedia

Principais Fabricantes de Processadores e Sistemas Operacionais

IBM

Intel

AMD

AMD Brasil

Cyrix

Sun

HP

Microsoft

Microsoft Brasil

Links Independentes sobre Hardware

Tom's Hardware Guide - um dos melhores *sites* sobre *hardware* do PC na Web: testes, artigos, etc

Painet - *site* com artigos coletados na *Web* sobre *hardware*

The PC Guide

The Hardware Group - informações sobre *hardware*: *benchmarking*, placas-mãe, etc.

CPU Central

System Optimization

Intel Secrets - esse *site* se anuncia como "tudo aquilo que a Intel não gostaria que você viesse a saber... "

B. Piropo *site* com artigos publicados na PCWorld, no Globo, etc.

Laércio Vasconcellos - *site* com artigos, *links*, dicas, etc.