IESB - Instituto de Educação Superior de Brasília
Engenharia de Computação e Eng. de Produção Elétrica (Telecomunicações)
Sistemas Computacionais: Hardware

1. Sistemas Numéricos

Contamos na base decimal, o que significa que temos 10 algarismos (0 a 9) e, para contar quantidades maiores do que nove, passamos a contar dezenas, centenas, milhares, ou seja, potências de 10. Dizemos, então que os números estão representados na BASE 10.

Exemplo: $4392{,}78_{10} = 4x1000 + 3x100 + 9x10 + 2x1 + 7x0{,}1 + 8x0{,}01$

$= 4x10^3 + 3x10^2 + 9x10^1 + 2x10^0 + 7x10^{-1} + 8x10^{-2}$

Percebe-se que o peso (a base elevada a um expoente) de cada algarismo (no exemplo, 4, 3, 9 ...) depende de sua posição dentro do número. Esse conceito é chamado de notação posicional e esse princípio de formação pode ser aplicado a qualquer outra base numérica. A estrutura geral de um número expresso na base B é:

$$X = A_nB^n + A_{n-1}B^{n-1} + ... + A_2B^2 + A_1B^1 + A_0B^0 + A_{-1}B^{-1} + A_{-2}B^{-2} + ...$$

A base **B** indica a quantidade de algarismos distintos utilizados, de 0 a B-1. Exemplos:

- Base 10: usa 10 algarismos, de 0 a 9 (= 10 - 1)
- Base 5: usa 5 algarismos, de 0 a 4 (= 5 - 1)
- Base 2: usa 2 algarismos, de 0 a 1 (= 2 - 1)
- Base 16: usa 16 algarismos, de 0 a F. As quantidades de 10 a 15 são representadas pelas letras de A a F, respectivamente (um símbolo por algarismo).

1.1. Conversão entre bases

Diversos métodos permitem transformar a representação de um número em uma base de origem para uma base destino:

- Método polinomial
- Método das divisões
- Método das subtrações
- Método da substituição direta
- Método da multiplicação (parte fracionária)

1.1.1. Método polinomial:

Utiliza a estrutura geral de um número em uma base B qualquer e interpreta-se esse número como um polinômio na aritmética da base destino. Por utilizar a forma geral, serve para converter de qualquer base para outra. Entretanto, é muito utilizado para a **conversão de uma base N para a base 10** já que a aritmética que utilizamos é a decimal.

Exemplos de conversão de uma base N (qualquer) para a base 10:

- $1001101_2 = 1x2^6 + 0x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 64 + 8 + 4 + 1 = 77_{10}$
- $423{,}1_5 = 4x5^2 + 2x5^1 + 3x5^0 + 1x5^{-1} = 100 + 10 + 3 + 0{,}25 = 113{,}25_{10}$
- $A5D_{16} = 10x16^2 + 5x16^1 + 13x16^0 = 2560 + 80 + 13 = 2653_{10}$

No entanto, conforme já foi dito, esse método serve para conversões entre quaisquer bases. Para uma conversão da base 3 para a base 6, por exemplo, avalia-se o polinômio na base destino, ou seja, a base 6:

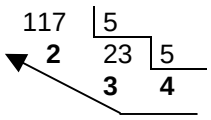
- $1222_3 \rightarrow X_6 \Rightarrow 1 \times 3^3 + 2 \times 3^2 + 2 \times 3^1 + 2 \times 3^0 = 43 + 30 + 10 + 2 = 125_6$

1.1.2. Método das divisões:

Determinam-se os algarismos de menor peso (mais a direita) do polinômio tomando-se o resto da divisão do número X pela base destino B: $X \div B = (A_nB^n + A_{n-1}B^{n-1} + ... + A_1B^1 + A_0B^0) \div B = A_nB^{n-1} + A_{n-1}B^{n-2} + ... + A_1B^0$ e resto A_0 . Repete-se o método sobre o quociente obtido até que este seja menor que a base B.

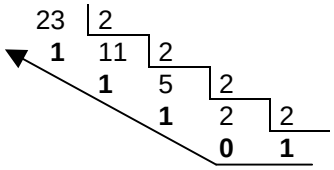
Novamente serve para conversões entre quaisquer bases, desde que as divisões sejam efetuadas utilizando-se a aritmética da base de origem. Assim, é muito utilizado para **conversões de números representados na base 10 para uma base qualquer N**, já que efetuamos as divisões na base 10.

- Exemplo: converter 117_{10} para a base 5



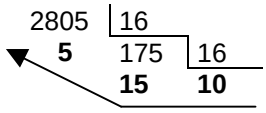
Número convertido é o resto das divisões, do último para o primeiro, além do último quociente menor que a base destino: $\mathbf{432_5}$

- Exemplo: converter 23_{10} para a base 2



Número convertido é o resto das divisões, do último para o primeiro, além do último quociente menor que a base destino: $\mathbf{10111_2}$

- Exemplo: converter 2803_{10} para a base 16



Número convertido: $\mathbf{AF5_{16}}$

Note que o número convertido é "montado" de trás para frente porque o primeiro resto é o coeficiente A_0. A cada divisão, os coeficientes maiores vão sendo obtidos, até que o quociente seja menor que a base de destino. Esse quociente será o coeficiente de maior peso do número convertido.

1.1.3. Método das subtrações:

É uma variação do método anterior. Ele é feito com base na verificação direta de quantas vezes a base "cabe" no número que se deseja representar, o que nada mais é do que divisão.

- Exemplo: converter 23_{10} para a base 2

$16_{10} < 23_{10} < 32_{10}$ ($2^4 < 23_{10} < 2^5$), donde o bit mais significativo do número na base dois está na casa de potência 4 ($1x2^4 = 16$).

A partir daí, podemos ir "preenchendo" as demais casas (potências de 2)

Potências de 2:	2^4	2^3	2^2	2^1	2^0
Expressas na base 10:	16	8	4	2	1
Quantas vezes a potência cabe no número que se deseja representar ?	1	0	1	1	1
Quanto sobra para a próxima casa ?	23 – 16 = 7	7 < 8	7 – 4 = 3	3 – 2 = 1	1 – 1 = 0

1.1.4. Método da substituição direta:

Nas conversões entre bases nas quais uma é potência da outra, a conversão pode ser feita diretamente pela substituição de algarismos da maior base por grupos de algarismos da menor base.

Note que 8 é a terceira potência de 2 ($2^3 = 8$). Assim, para converter um número de binário para octal, basta agrupar cada três bits a partir da casa de potência 0 e identificar o dígito octal equivalente, como mostram os exemplos:

$1\,1\,1\,1\,0\,0\,1\,0\,1\,1\,1\,0_2$

111	100	101	110
7	4	5	6
7456_8			

Da mesma forma, para converter um número da base 8 para a base 2, basta associar 3 dígitos binários a cada dígito octal, como mostra o exemplo a seguir. Note que os zeros à esquerda de cada dígito não podem ser desprezados.

$5\,1\,0\,3\,,\,2_8$

5	1	0	3	2
101	001	000	011	010
$1\,0\,1\,0\,0\,1\,0\,0\,0\,0\,1\,1\,,\,0\,1\,0_2$				

De forma similar à conversão entre as bases 2 e 8, para converter um número de binário para hexadecimal, basta agrupar cada quatro bits ($2^4 = 16$) a partir da casa de potência 0 e identificar o dígito hexadecimal equivalente, como mostra o exemplo:

$1\,0\,1\,1\,1\,1\,1\,0\,0\,1\,0\,1\,1\,1\,0_2$

0101	1111	0010	1110
5	F	2	E
$5F2E_{16}$			

Da mesma forma, para converter um número da base 16 para a base 2, basta associar 4 dígitos binários a cada dígito hexadecimal, como mostra o exemplo a seguir. Note que os zeros à esquerda de cada dígito não podem ser desprezados.

$C\,1\,0\,8\,D_{16}$

C	1	0	8	D
1100	0001	0000	1000	1101
$1\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,1\,1\,0\,1_2$				

1.1.5. Método das multiplicações:

É utilizado quando o número na base de origem é ou possui partes fracionárias. A parte fracionária do número é multiplicada pela base destino. Os algarismos a esquerda da vírgula produzidos por cada multiplicação fornecem a parte fracionária na base destino: $X \bullet B = (A_{-1}B^{-1} + A_{-2}B^{-2} + A_{-3}B^{-3} + ...) \bullet B = A_{-1} + (A_{-2}B^{-1} + A_{-3}B^{-2} + ...)$.

- Exemplo: Decimal para binário: $0{,}8125_{10} \rightarrow X_2$

$0{,}8125 \bullet 2 = 1{,}625 \Rightarrow A_{-1} = 1$
$0{,}625 \bullet 2 = 1{,}25 \Rightarrow A_{-2} = 1$
$0{,}25 \bullet 2 = 0{,}5 \Rightarrow A_{-3} = 0$
$0{,}5 \bullet 2 = 1{,}0 \Rightarrow A_{-4} = 1$

$0{,}8125_{10} = 0{,}1101_2$

1.2. Bases especialmente úteis para sistemas computacionais

Como foi observado, é possível utilizar qualquer base numérica para representar uma dada grandeza. No "mundo" dos sistemas computacionais, há algumas bases que são especialmente úteis:

- A **base binária (base 2)**, pois os computadores utilizam apenas dois níveis lógicos, donde fica evidente a associação do código binário com o processamento binário de informações.

- A **base hexadecimal (base 16)**, pois facilita a visualização de grandezas binárias já que um número binário de 8 bits tem apenas 2 dígitos hexadecimais, donde fica muito mais fácil visualizar a grandeza, sem que seja difícil efetuar a conversão para o binário, em caso de necessidade.

A tabela a seguir facilita bastante as operações de conversões entre essas bases e delas para a decimal (e vice-versa):

Decimal	Binário	Hexa
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Decimal	Binário	Hexa
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

1.2.1. Conversões das bases 2 e 16 para a base 10:

- Se a conversão estiver sendo feita utilizando a **aritmética decimal** (um ser humano) o método mais apropriado é o **método polinomial**;
- Se a conversão estiver sendo feita dentro de um microprocessador, que utiliza a **aritmética hexadecimal**, o método mais apropriado é o **método das divisões**.
- Exemplo: $A73_{16} \rightarrow X_{10} \Rightarrow 10 \times 16^2 + 7 \times 16^1 + 3 \times 16^0 = 2.560 + 112 + 3 = 2.675_{10}$ (método polinomial já que o polinômio foi avaliado na base 10)

1.2.2. Conversões da base10 para as bases 2 e 16:

- Se a conversão estiver sendo feita utilizando a **aritmética decimal** (um ser humano) o método mais apropriado é o **método das divisões**;
- Se a conversão estiver sendo feita dentro de um microprocessador, que utiliza a **aritmética hexadecimal**, o método mais apropriado é o **método polinomial**.
- Exemplo: $2.675_{10} \rightarrow X_{16} \Rightarrow 2675 \div 16 = 167$ e resto 3, $167 \div 16 = 10$ e resto 7. Com o quociente menor que a base, o valor convertido é: $A73_{16}$.

Note que: $16^4 = 65.536$ e $A_3 \times 16^3 + A_2 \times 16^2 + A_1 \times 16^1 + A_0 \times 16^0 = (A_3 \times 16^1 + A_2) \times 16^2 + (A_1 \times 16^1 + A_0) = (A_3 \times 16^1 + A_2) \times 256 + (A_1 \times 16^1 + A_0)$. Com isso, podemos derivar a seguinte regra para converter um número menor que 65.536_{10} da base 10 para a base 16 utilizando o método das divisões (aritmética decimal):

- Se o número for menor que 256, é porque ele só é formado pelos coeficientes A_1 e A_0. Dessa forma, basta dividir o número por 16, o quociente será o coeficiente A_1 e o resto será o coeficiente A_0.
- Se o número for maior que 256, basta dividi-lo por 256. O quociente dessa divisão será formado pelos coeficientes A_3 e A_2 e o resto será formado pelos coeficientes A_1 e A_0. Cada um desses termos será menor que 256 e aplica-se a regra anterior.
- Exemplo: $47.235_{10} \rightarrow X_{16} \Rightarrow 47.235 \div 256 = 184$ e resto 131, $184 \div 16 = 11$ e resto 8 (A_3 =B e $A_2 = 8$), $131 \div 16 = 8$ e resto 3 ($A_1 = 8$ e $A_0 = 3$) $\Rightarrow 47.235_{10} \rightarrow \mathbf{B883_{16}}$

- Obs.: Um forma de indicar em qual base o número está representado é colocando uma letra no final desse número. Assim, 1001B estaria representado na base 2 (binário), 1001D estaria representado na base 10 (decimal), 1001H estaria representado na base 16 (hexadecimal). Essa notação será utilizada no decorrer desse texto e, caso a letra seja omitida, significa que o número está representado na base 10. Em outros casos, quando a base estiver implícita, como é o caso dos itens 3 e 4 (base 2), essa notação também será omitida.

2. Conceitos básicos:

2.1. Sistema binário:

Sistema de informação que utiliza o sistema numérico binário, ou seja, na base 2. Dessa forma, só possui dois símbolos para representação de valores, o 0 e o 1. É o caso dos sistemas computacionais.

2.2. Bit:

Do inglês ***Bi****nary digi****t*** ou dígito binário. É a menor unidade de informação de um sistema binário, representado pelos valores lógicos 0 e 1.

2.3. Byte:

Do inglês ***Bi****nar****y te****rm* ou termo binário. Um sistema computacional ao armazenar dados ou informações, não guarda apenas um bit por vez, mas sim um agrupamento de bits. Assim, o byte é uma unidade de armazenamento de informação em sistemas computacionais. Um byte é composto de **8 bits**, sendo esses bits numerados de 0 a 7:

b7	b6	b5	b4	b3	b2	b1	b0

É importante guardar essa numeração pois ela será utilizada em futuras referências. Com ela é possível identificar a posição de um determinado bit dentro de um byte. O bit 0 é chamado de LSB (do inglês *least significant bit* ou bit menos significativo) e o bit 7 é chamado de MSB (do inglês *most significant bit* ou bit mais significativo).

Outro detalhe a ressaltar é que com 8 bits são possíveis 256 combinações ($2^8 = 256$). Dessa forma, um byte pode conter **valores de 0 a 255** na base decimal ou de **00H a FFH** na base hexadecimal.

2.4. Nibble:

Equivale a **4 bits**. Assim, um byte é composto por dois nibbles. O nibble composto pelos bits de 0 a 3 é chamado de nibble menos significativo. O nibble composto pelos bits de 4 a 7 é chamado de nibble mais significativo. Um nibble equivale a um dígito hexadecimal (de 0H a FH).

2.5. Registradores:

São unidades de memória dentro de um processador e servem para guardar resultados de operações e outros valores. Podem armazenar um ou mais bytes (registradores de 8 bits, registradores de 16 bits, etc.). Se um registrador for de **8 bits** o valor **máximo** que ele pode armazenar é **255** (ou **FFH**).

2.6. Flags:

São bits dentro de um determinado registrador. Servem para **indicar a ocorrência de algum evento** para o processador.

Exemplos:

- **Flag Z** (zero): é colocada em 1 quando o resultado de uma operação é igual a zero.
- **Flag C** (*carry*): é colocada em 1 quando ocorre um "vai um" em uma soma. É afetada também em operações de subtração e por algumas funções lógicas.

3. Principais funções lógicas em sistemas computacionais:

Nas funções a seguir as variáveis A e B são variáveis de entrada e S é o valor de saída da função.

3.1. Função INVERSORA ou NÃO (NOT):

A	**S**
0	1
1	0

A saída dessa função é o inverso (complemento) de sua entrada.

3.2. Função E (AND):

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

A saída só assume o valor lógico 1 se todas as entradas também estiverem no nível lógico 1. Na álgebra booleana, essa função é representada pelo operador: • . Observando a primeira coluna quando B = 0 e quando B = 1, podemos descrever a função E (AND) de uma outra forma:

- **A • 0 = 0** , ou seja, se efetuarmos a operação E (AND) com uma das entradas igual a zero, o resultado (saída) será 0 independente do valor da outra variável.
- **A • 1 = A** , ou seja, se efetuarmos a operação E (AND) com uma das entradas igual a um, o resultado (saída) será igual ao valor da outra variável de entrada. Essa operação é chamada de **neutra** pois não afeta o valor da variável.

A operação E é muito útil em sistemas computacionais quando queremos **zerar um ou mais bits de um byte sem alterarmos o valor (ou estado lógico) dos demais bits**. Por exemplo, suponhamos que queremos zerar os bits 3, 4 e 7 de um determinado byte. Para isso, é criado um valor chamado de **máscara** que irá executar essa função. Conforme foi visto da definição da função E, essa máscara deverá conter o seguinte valor: 01100111. Assim, ao efetuarmos a operação E dessa máscara com o byte em questão, teremos:

Byte:	b7	b6	b5	b4	b3	b2	b1	b0
Máscara:	0	1	1	0	0	1	1	1
Resultado:	0	b6	b5	0	0	b2	b1	b0

Note que, no resultado final da operação, os bits 0, 1, 2, 5 e 6 permaneceram inalterados e o bits 3, 4 e 7 foram zerados.

3.3. Função OU (OR):

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

A saída assume o valor lógico 1 sempre que uma das entradas também estiver no nível lógico 1. Na álgebra booleana, essa função é representada pelo operador: **+** . Observando a primeira coluna quando B = 0 e quando B = 1, podemos descrever a função OU (OR) de uma outra forma:

- **A + 0 = A** , ou seja, se efetuarmos a operação OU (OR) com uma das entradas igual a zero, o resultado (saída) será igual ao valor da outra variável de entrada. Essa operação é chamada de **neutra** pois não afeta o valor da variável.
- **A + 1 = 1** , ou seja, se efetuarmos a operação OU (OR) com uma das entradas igual a um, o resultado (saída) será 1 independente do valor da outra variável.

A operação OU é muito útil em sistemas computacionais quando queremos **"setar" (colocar em nível lógico 1) um ou mais bits de um byte sem alterarmos o valor (ou estado lógico) dos demais bits**. Por exemplo, suponhamos que queremos "setar" os bits 2, 5 e 7 de um determinado byte. Da mesma forma que na operação E, é preciso criar uma máscara que irá executar essa função. Conforme foi visto da definição da função OU, essa máscara deverá conter o seguinte valor: 10100100. Assim, ao efetuarmos a operação OU dessa máscara com o byte em questão, teremos:

Byte:	b7	b6	b5	b4	b3	b2	b1	b0
Máscara:	1	0	1	0	0	1	0	0
Resultado:	1	b6	1	b4	b3	1	b1	b0

Note que, no resultado final da operação, os bits 0, 1, 3, 4 e 6 permaneceram inalterados e o bits 2, 5 e 7 foram setados.

3.4. Função XOR (Ou Exclusivo ou *Exclusive OR*):

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

A saída assume o valor lógico 1 sempre que as entradas possuírem valores distintos (essa regra vale para uma tabela verdade de duas variáveis). Na álgebra booleana, essa função é representada pelo operador: ⊕ . Observando a primeira coluna quando B = 0 e quando B = 1, podemos descrever a função OU (OR) de uma outra forma:

- **A ⊕ 0 = A** , ou seja, se efetuarmos a operação XOR com uma das entradas igual a zero, o resultado (saída) será igual ao valor da outra variável de entrada. Essa operação é chamada de **neutra** pois não afeta o valor da variável.
- **A ⊕ 1 = $\overline{\mathbf{A}}$** , ou seja, se efetuarmos a operação XOR com uma das entradas igual a um, o resultado (saída) será o complemento (inverso) do valor da outra variável.

Além disso:

- **A ⊕ A = 0** , ou seja, se efetuarmos a operação XOR com dois valores de entrada iguais, o resultado (saída) será igual a zero.

A operação XOR é muito útil em sistemas computacionais quando queremos **inverter um ou mais bits de um byte sem alterarmos o valor (ou estado lógico) dos demais bits**. Ela também serve para testarmos se **dois valores são iguais**. Por exemplo, suponhamos que queremos inverter os bits 0, 2 e 6 de um determinado byte. A máscara deverá conter o seguinte valor: 01000101. Assim, ao efetuarmos a operação XOR dessa máscara com o byte em questão, teremos:

Byte:	b7	b6	b5	b4	b3	b2	b1	b0
Máscara:	0	1	0	0	0	1	0	1
Resultado:	b7	$\overline{\text{b6}}$	b5	b4	b3	$\overline{\text{b2}}$	b1	$\overline{\text{b0}}$

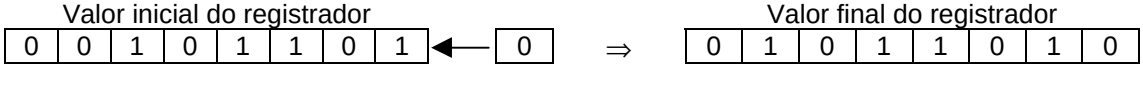
Note que, no resultado final da operação, os bits 0, 1, 3, 4 e 6 permaneceram inalterados e o bits 2, 5 e 7 foram setados.

Supondo agora que um byte B1 contém o valor 74H e o byte B2 contém o valor D3H. Dentro do sistema computacional o valor desses bytes não é conhecido, mas queremos determinar se algum deles é igual a D3H (cuja máscara é 11010011). Com o uso da função XOR efetuamos esse teste:

Byte B1:	0	1	1	1	0	1	0	0
Máscara:	1	1	0	1	0	0	1	1
Resultado:	1	0	1	0	0	1	1	1
Byte B2:	1	1	0	1	0	0	1	1
Máscara:	1	1	0	1	0	0	1	1
Resultado:	0	0	0	0	0	0	0	0

Note que para o byte B2 o resultado final da operação foi igual a zero. Esse resultado é facilmente testado dentro de um sistema computacional como veremos mais à frente.

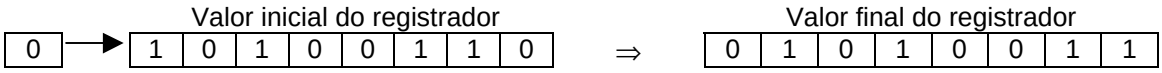
3.5. Deslocamento para a esquerda (*shift left*):



Um zero é inserido no lugar do bit 0, o bit 0 vai para a posição do bit 1, o bit 1 vai para a posição do bit 2 e assim, sucessivamente, até o bit 6 ocupar a posição do bit 7. Nessa operação, o bit 7 é perdido. Dessa forma, essa operação é irreversível, ou seja, nada garante que um deslocamento no sentido inverso irá recuperar o valor original contido no registrador.

Na aritmética decimal, se um número for deslocado à esquerda e um zero inserido na posição das unidades, teremos um valor multiplicado por 10. Assim, da mesma forma que na aritmética decimal, o uso dessa operação, no caso do bit 7 ser igual a 0, resulta em um valor multiplicado por 2. Note que, no exemplo dado, o valor inicial do registrador era 00101101B (45 em decimal) e após a operação seu valor passou para 01011010B (90 em decimal).

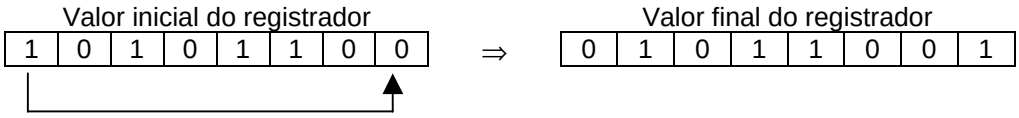
3.6. Deslocamento para a direita (*shift right*):



Um zero é inserido no lugar do bit 7, o bit 7 vai para a posição do bit 6, o bit 6 vai para a posição do bit 5 e assim, sucessivamente, até o bit 1 ocupar a posição do bit 0. Nessa operação, o bit 0 é perdido, ou seja, uma operação irreversível.

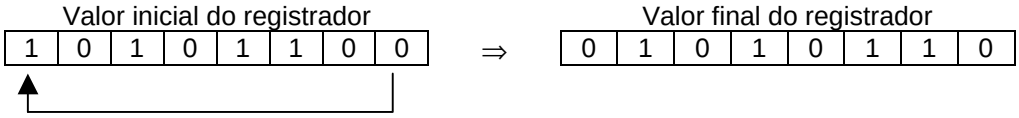
A aplicação dessa operação resulta em um valor dividido por 2. Note que, no exemplo dado, o valor inicial do registrador era 10100110B (166 em decimal) e após a operação seu valor passou para 01010011B (83 em decimal).

3.7. Rotação à esquerda (*rotate left*):



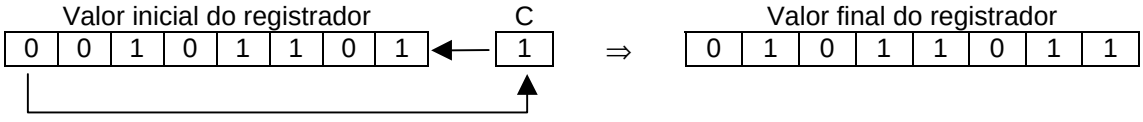
Nessa operação, o bit 0 vai para a posição do bit 1, o bit 1 vai para a posição do bit 2 e assim, sucessivamente, até o bit 6 ocupar a posição do bit 7 e o bit 7 ocupar a posição do bit 0. Nenhum bit é perdido e essa operação é reversível, ou seja, o valor original contido no registrador antes da operação pode ser restaurado com um deslocamento no sentido inverso.

3.8. Rotação à direita (*rotate right*):



Nessa operação, o bit 6 vai para a posição do bit 5, o bit 5 vai para a posição do bit 4 e assim, sucessivamente, até o bit 1 ocupar a posição do bit 0 e o bit 0 ocupar a posição do bit 7. Nenhum bit é perdido e essa operação é reversível.

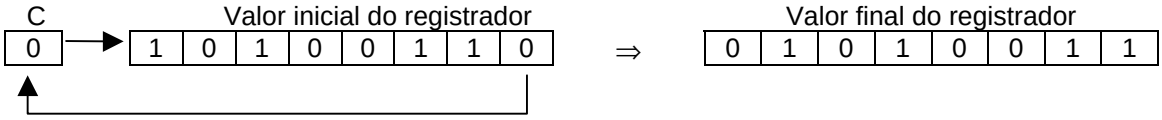
3.9. Rotação à esquerda com a flag *carry* (*rotate left through carry*):



O valor da flag *carry* é inserido no lugar do bit 0, o bit 0 vai para a posição do bit 1, o bit 1 vai para a posição do bit 2 e assim, sucessivamente, até o bit 6 ocupar a posição do bit 7 e o bit 7 ir para a flag *carry*. Nenhum bit é perdido e essa operação é reversível.

No exemplo dado, o valor da flag *carry* após a operação seria 0.

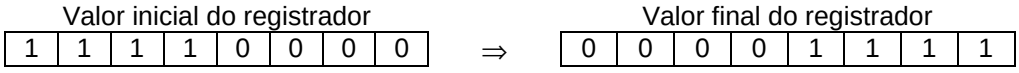
3.10. Deslocamento para a direita (*shift right*):



O valor da flag *carry* é inserido no lugar do bit 7, o bit 7 vai para a posição do bit 6, o bit 6 vai para a posição do bit 5 e assim, sucessivamente, até o bit 1 ocupar a posição do bit 0 e o bit 0 ir para a flag *carry*. Nenhum bit é perdido e essa operação é reversível.

No exemplo dado, o valor da flag *carry* após a operação seria 0.

3.11. Swap:



Os dois nibbles (4 bits) do byte são trocados. O nibble mais significativo vai para o lugar do nibble menos significativo e vice-versa, ou seja, o bit 7 vai para a posição do bit 3, o bit 6 para o bit 2, o 5 para o 1 e o 4 para o 0, da mesma forma, o bit 3 vai para a posição do bit 7, o bit 2 para o bit 6, o 1 para o 5 e o 0 para o 4.

4. Aritmética binária

Apresentamos a seguir as operações básicas da aritmética binária. Note que elas seguem exatamente os mesmos princípios da aritmética decimal. Na verdade, ela é ainda mais simples do que esta, devido ao pequeno número de resultados possíveis. Como só há dois números na base, há menos combinações a considerar em cada operação, como veremos a seguir.

4.1. Soma

Somando dois bits, podemos ter as seguintes combinações:

```
  0      1      0      1
+ 0    + 0    + 1    + 1
---    ---    ---    ---
  0      1      1     10    Soma = 0 e "vai 1"
```

Assim como na aritmética decimal, usamos o "vai 1" (ou *carry*, em inglês) quando o resultado da soma de dois dígitos não cabe mais na base. Na aritmética decimal, o "vai 1" surge quando a soma é maior do que 9. No caso da aritmética binária, o "vai 1" surge quando a soma é maior que 1.

Com o uso do "vai 1", a soma de dois bytes quaisquer demanda o uso da soma de 3 bits. Para uma soma de três bits, temos:

```
  0      1      0      0      1      1      0      1
+ 0    + 0    + 1    + 0    + 1    + 0    + 1    + 1
---    ---    ---    ---    ---    ---    ---    ---
  0      1      1      1     10     10     10     11
```

Com base nessa tabela, a soma de dois bytes, por exemplo 84H e DEH, pode ser efetuada:

```
"Vai um"    1           1  1  1
94H            1  0  0  1  0  1  0  0
DEH            1  1  0  1  1  1  1  0
Resultado   1  0  1  1  1  0  0  1  0
```

Note que o resultado possui 9 bits (101110010B) e, se essa soma tivesse sido efetuada em um processador com registradores de 8 bits, ele não caberia dentro do registrador de operações aritméticas. Nesse caso, no registrador ficaria armazenado o valor 72H (01110010B) e a flag *carry* seria colocada em 1 indicando esse último "vai um".

4.2. Subtração

Subtraindo-se um bit de outro, podemos ter as seguintes combinações:

0	1	1	0	
– 0	– 0	– 1	– 1	
0	1	0	1	Subtração = 1 e "vem 1"

Na última coluna da tabela temos o caso A – B no qual B > A. Assim como na aritmética decimal, usamos o "vem 1" ou "empresta 1" (*borrow* em inglês) da próxima potência quando isso ocorre. Vindo 1 "emprestado", ficamos com o valor 10 que subtraído 1 resulta em 1.

A subtração, por exemplo, dos bytes DEH – A4H ficaria:

"Empresta 1"		-1				-1	-1	
DCH	1	1	10	1	1	1	10	10
ABH	– 1	0	1	0	1	0	1	1
Resultado	0	0	1	1	0	0	0	1

Note que, em A – B, o resultado será negativo quando A < B. A representação de números negativos será apresentada no próximo item. Assim como nos números decimais, é possível obter o resultado da subtração fazendo-se B – A e invertendo-se o sinal do resultado.

4.3. Multiplicação

A multiplicação binária é extremamente simples. Afinal, só existem dois resultados possíveis para a multiplicação A x B com apenas 1 bit:

B	0	1	0	1
A	x 0	x 0	x 1	x 1
	0	0	0	1

- Se A = 0 → A x B = 0
- Se A = 1 → A x B = B

Assim sendo, para o caso dos números A e B possuírem mais bits, a multiplicação é feita exatamente como na álgebra decimal, como mostra o exemplo:

A:			1	0	1			5
B:		x 1	0	0	1			x 9
			1	0	1		1x101	
		0	0	0			0x101	
	0	0	0				0x101	
+ 1	0	1					1x101	
1	0	1	1	0	1			= 45

4.4. Divisão

Assim como a multiplicação, a divisão se torna extremamente simples em função de só termos dois resultados possíveis para A ÷B:

- Se A > B, A ÷B = 1 + resto (onde o resto é A – B)
- Se A < B, A ÷B = 0 + resto (onde o resto é o próprio A)

Observe o exemplo: $A \div B = 10111001_2 \div 101_2 = 100101_2$ (185 ÷ 5 = 37)

```
 1  0  1  1  1  0  0  1        | 1  0  1
-1  0  1                      | 1  0  0  1  0  1
      0  1   (A<B: 1º 0 no quociente)
      0  1  1   (2º 0 no quociente)
      0  1  1  0
        - 1  0  1
      0  0  1  0   (3º 0)
            1  0  1
         - 1  0  1
            0  0  0
```

Note que esta divisão resultou exata. É possível que a divisão resulte em um número não inteiro, o que inclui a possibilidade de obter-se dízimas. A representação de números não inteiros em sistemas digitais necessita do uso de códigos especiais, a notação em ponto flutuante.

5. Números Binários Negativos

Como sabemos, sistemas binários apenas lidam com dígitos binários, que podem assumir os valores 0 ou 1. A representação de números negativos demanda o uso de códigos especiais. Veremos a seguir duas maneiras de representar números negativos.

Como em todos os códigos, é necessário ter em mente que o código funciona por ser uma convenção conhecida por todos os que lidam com o dado. O código é escolhido em função da aplicação e deve ser usado coerentemente.

5.1. Bit de sinal

Adiciona-se à palavra de dados um bit que indica o sinal. A convenção mais usual define o bit mais significativo (MSB ou bit 7) como sendo o bit de sinal, da seguinte forma:

- 0: número positivo
- 1: número negativo

Exemplo:

+25_{10} = **0**0011001_2	**+**100_{10} = **0**1100100_2	**+**32_{10} = **0**0100000_2
- 25_{10} = **1**0011001_2	- 100_{10} = **1**1100100_2	-32_{10} = **1**0100000_2

5.2. Complemento de 2

A codificação por bit de sinal normalmente não é utilizada por computadores e calculadoras, pois possui uma dupla representação para o zero e demanda o uso de circuitos distintos (diferentes) para efetuar operações de soma e subtração.

A representação de sinal por complemento de 2 é mais utilizada, pois permite que a subtração seja feita da mesma forma que a soma, como será mostrado.

As propriedades da representação em complemento de 2 advém do fato de trabalharmos com um número FIXO e LIMITADO de bits. Observe o que ocorre com o hodômetro (4 dígitos) de um carro 0 km:

Andando para frente	0000	0001	0002	0003	0004	0005	0006
Andando para trás	0000	9999	9998	9997	9996	9995	9994

Se considerarmos “andar para frente” como número positivo e “andar para trás” como número negativo, teríamos:

+1=0001	+2=0002	+3=0003	+4=0004
-1=9999	-2=9998	-3=9997	-4=9996

Se tivéssemos um hodômetro com 6 bits, no entanto, teríamos:

+1=000001	+2=000002	+3=000003	+4=000004
-1=999999	-2=999998	-3=999997	-4=999996

Note que é o tamanho do hodômetro que determina qual será o número negativo. No primeiro hodômetro, -1=9998, enquanto no segundo, -1=999998. Assim, se desejamos trabalhar com quantidades negativas representadas de acordo com este princípio, precisamos usar um *número fixo de bits*.

Aplicando o mesmo princípio para números binários, teríamos:

Andando para frente	**0**000	**0**001	**0**010	**0**011	**0**100	**0**101	**0**110
Andando para trás	**0**000	**1**111	**1**110	**1**101	**1**100	**1**011	**1**010
	0	-1	-2	-3	-4	-5	-6

Os números acima estão representados em *complemento de 2*. Note que todos os números positivos começam com **0** e os negativos começam com **1**, assim como quando usamos o bit de sinal. Os bits restantes indicam a magnitude (tamanho) do número.

Para números com 8 bits, como é o caso de vários processadores, a codificação em complemento de 2 gera números positivos de 0 a +127 e negativos de –1 a –128. Note que não existe mais a representação duplicada do número 0.

Para achar a representação em complemento de 2 (valor negativo) de um número binário deve-se executar os seguintes passos:

Procedimento	*Exemplo: – 10*
1. Achar o complemento de 1: *toma-se o número positivo e troca-se cada 1 por 0 e cada 0 por 1*	0 0 0 0 1 0 1 0 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ 1 1 1 1 0 1 0 1
2. Achar o complemento de 2: *soma-se 1 ao complemento de 1 calculado anteriormente*	1 1 1 1 0 1 0 1 + 1 1 1 1 1 0 1 1 0 = -10

Os conceitos de complemento de 1 e complemento de 2 se aplicam a qualquer base numérica. O complemento de 1 é obtido achando-se a diferença entre o algarismo desejado e o maior algarismo da base. O complemento de 2 é obtido somando-se 1 ao resultado (da mesma forma que para os número binários).

5.2.1. Soma e subtração com complemento de 2

A operação de subtração de dois números binários pode ser efetuada como uma soma: A (minuendo) – B (subtraendo) = A + (-B). Logo, para efetuar uma subtração usando complemento de 2 basta somar o minuendo com a representação em complemento de 2 do subtraendo, como mostram os exemplos a seguir:

	Decimais	Binários	
Soma:	18	0 0 0 1 0 0 1 0	
	+ 11	+ 0 0 0 0 1 0 1 1	
	29	0 0 0 1 1 1 0 1	

	Decimais	Binários	
Subtração:	18	0 0 0 1 0 0 1 0	
	– 11	+ 1 1 1 1 0 1 0 1	(complemento de 2 de 11 = – 11)
	7	**1** 0 0 0 0 0 1 1 1	= 7 (desprezado o **"vai um"**)

	Decimais	Binários	
Subtração:	18	0 0 0 1 0 0 1 0	
	– 18	+ 1 1 1 0 1 1 1 0	(complemento de 2 de 18 = – 18)
	0	**1** 0 0 0 0 0 0 0 0	= 0 (desprezado o **"vai um"**)

	Decimais	Binários	
Subtração:	18	0 0 0 1 0 0 1 0	
	– 23	+ 1 1 1 0 1 0 0 1	(complemento de 2 de 23 = – 23)
	– 5	1 1 1 1 1 0 1 1	(complemento de 2 de 5 = – 5)

Note que:

- A soma é efetuada normalmente;
- Na subtração, o subtraendo é transformado em um número negativo (complemento de 2) e depois é efetuada uma soma;
- Se o resultado for positivo, ocorrerá um "vai um". Em um processador isso fará com que a flag *carry* seja setada (colocada em 1);
- Se o resultado for negativo, não ocorrerá um "vai um" e a flag *carry* será zerada. O valor obtido é a representação negativa (complemento de 2) do resultado. No último exemplo de subtração, o resultado obtido foi 11111011 que é a representação negativa de – 5.

Os microcontroladores da família PIC que serão estudados efetuam as subtrações como somas em complemento de 2. É importante entender essas propriedades para se poder interpretar os resultados obtidos dentro do processador.

6. Códigos Binários

6.1. Códigos BCD

Como a base binária é "pequena" (possui apenas os dígitos 0 e 1), a codificação binária de números relativamente pequenos já exige uma grande quantidade de dígitos. Isto torna complexa a visualização dos números e, por vezes, seu processamento.

Para contornar esta dificuldade de conversão e visualização, são usados os códigos BCD – *Binay Coded Decimal* (decimais codificados como binários). Nesta codificação, é feito algo semelhante à conversão hexadecimal ↔ binário. A cada dígito DECIMAL são associados 4 bits. A diferença do uso do BCD para o hexadecimal é que cada 4 bits continuam associados a uma potência de 10, como mostra o exemplo a seguir.

32.947_{10}

3_{10}	2_{10}	9_{10}	4_{10}	7_{10}
0011	0010	1001	0100	0111
$0011\ 0010\ 1001\ 0100\ 0111_{BCD8421}$				

Note que, se convertermos o número acima de binário para decimal, ele será diferente de 32.947. O que fizemos não foi uma conversão puramente matemática – foi a aplicação de um código que associa a cada dígito decimal quatro dígitos binários. Apesar de o número BCD ser maior do que o binário equivalente a 32.947, ele tem a vantagem de ser mais facilmente convertido para decimal.

6.2. Alfanuméricos

Até aqui, foram vistos códigos que permitem a representação de números. Há também os códigos que permitem a representação de letras, símbolos e caracteres de controle (para impressoras, por exemplo). Os principais são apresentados a seguir.

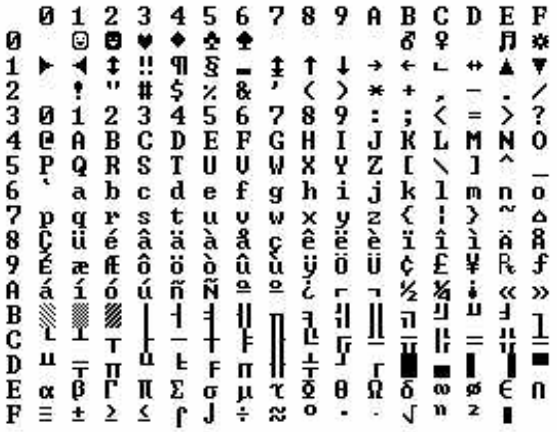
ASCII - *American Standard Code for Information Interchange*: padroniza o uso de 7 bits. O bit mais significativo do byte (bit 7) era originalmente reservado ao uso de bit de segurança (bit de paridade).

Com o aumento da necessidade de símbolos (desenho de tabelas, caracteres acentuados, entre outros), passou-se a usar o bit mais significativo também para gerar códigos válidos. O grande problema desta ampliação foi a falta de padronização (o mesmo código era usado para símbolos diferentes, em diferentes softwares). Ainda hoje ocorrem erros devidos a esta falta de padronização.

A tabela a seguir mostra o código ASCII de 7 bits. As primeiras linha e coluna são o número hexadecimal equivalente a cada código. Por exemplo, a letra A (maiúscula) corresponde ao código 41_{16} (4 na primeira coluna, 1 na primeira linha), que corresponde ao binário $0100\ 0001_2$.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

As palavras na tabela são nomes de códigos de controle (del = apagar, por exemplo). Uma versão do código ASCII para 8 bits é mostrada a seguir:



Outros códigos :

BCDIC - *Binary Coded Decimal Interchanging Code* (7 bits) e

EBCDIC - *Extended Binary Coded Decimal Interchanging Code* (8 bits): Usado em mainframes IBM. O código original tinha 7 bits (128 símbolos) e sua versão estendida tem 8 bits (256 símbolos).

6.3. Códigos de Paridade

Acrescenta-se um bit à palavra de modo que a soma dos "1's" da palavra e do bit de paridade seja par ou ímpar. Tem por objetivo a detecção de erros simples na transmissão de dados.

Código	Paridade Par	Qtde de 1's
000	0	0
001	1	2
010	1	2
011	0	2
100	1	2
101	0	2
110	0	2
111	1	4

Código	Paridade Ímpar	Qtde de 1's
000	1	1
001	0	1
010	0	1
011	1	3
100	0	1
101	1	3
110	1	3
111	0	3