

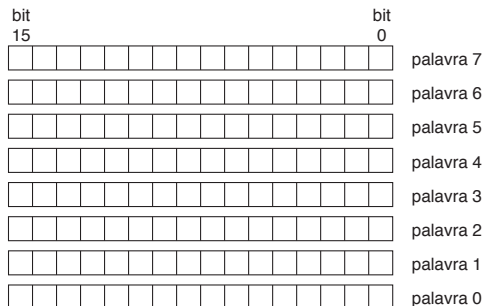
Linguagens de montagem

Capítulo 1 – Organização da memória e representação digital da informação

Ricardo Anido
Instituto de Computação
Unicamp

Organização da memória de computadores

- ▶ A unidade de informação em sistemas digitais é o *bit*, que pode assumir apenas dois valores: zero ou um.
- ▶ Uma *palavra* é um conjunto de tamanho fixo de bits.



Organização da memória de computadores

- ▶ A quantidade de memória de um computador é tradicionalmente medida em bytes, independentemente do tamanho da palavra utilizada.
- ▶ Usualmente são utilizadas as abreviações KB (kilobytes, ou seja, milhares de bytes), MB (megabytes, ou milhões de bytes), GB (gigabytes, ou bilhões de bytes) e TB (terabytes, ou trilhões de bytes).

Organização da memória de computadores

- ▶ Dados e instruções compartilham a mesma memória, na forma de sequências de zeros e uns.
- ▶ O processador não tem meios de determinar se uma dada sequência de bits representa por exemplo uma instrução ou um dado.
- ▶ A interpretação dessas sequências é responsabilidade exclusiva do programador.

Representação de inteiros sem sinal

- ▶ Números no sistema decimal utilizam dez dígitos (de 0 a 9).
- ▶ Para representar valores inteiros no computador é natural utilizar uma representação binária, que utiliza apenas os dígitos 0 e 1.
- ▶ A representação hexadecimal (que utiliza dezesseis dígitos) é uma alternativa mais cômoda do que a binária, por ser mais compacta.

- ▶ Na notação posicional o valor representado por um dígito depende da posição que o dígito ocupa em um número.
- ▶ Por exemplo, o dígito 7 representa setenta tanto em 671 quanto em 6375. Já o dígito 6 representa 600 no primeiro número mas 6000 no segundo número.

Notação Posicional

- ▶ A *base* de um sistema numérico é o número de dígitos que esse sistema utiliza. Por exemplo, no sistema binário a base é 2.
- ▶ O valor com que um dígito contribui para o valor total do número em uma representação posicional é o valor do dígito multiplicado por uma potência da base do sistema numérico usado.
- ▶ Se o número tem, em uma dada base b , a representação

$$d_n d_{n-1} \dots d_2 d_1 d_0$$

em que cada d_i representa um dígito, então d_i contribui para o valor do número com o valor do dígito d_i multiplicado pela base elevada à i -ésima potência.

O valor do número, em decimal, é dado pela Equação 1:

$$(\delta_n \times b^n) + (\delta_{n-1} \times b^{n-1}) + \dots + (\delta_2 \times b^2) + (\delta_1 \times b^1) + \delta_0 \quad (1)$$

em que cada expressão entre parênteses representa o valor (na base 10) de um dígito na base b , e δ_i é o valor do dígito d_i na base 10.

- ▶ Para uma determinada base b , precisamos símbolos que identifiquem b valores de dígitos (de 0 a $b - 1$).
- ▶ Para base 16 a convenção é utilizar os símbolos 0, 1, 2, ..., 8, 9, A, B, C, D, E e F, com os caracteres de A a F representando os dígitos de dez a quinze, respectivamente.

Conversão entre binário e hexadecimal

- ▶ O número de valores distintos que podemos representar usando um número fixo de dígitos k em uma determinada base b é b^k .
- ▶ Se os valores a serem representados são inteiros positivos, podemos representar os números no intervalo $[0, b^k - 1]$.
- ▶ Por exemplo, usando quatro dígitos na base 2 podemos representar todos os números no intervalo $[0, 15]$.
- ▶ Quatro dígitos binários têm o mesmo poder de representação de um dígito hexadecimal (16 valores distintos).

Conversão entre binário e hexadecimal

Decimal	Binário	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Conversão de binário para hexadecimal

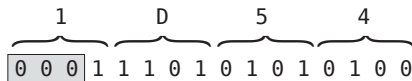
Podemos usar a representação hexadecimal como uma forma abreviada da representação binária, convertendo cada quatro bits para a representação hexadecimal equivalente.

1 1 1 1 0 1 1 1 1 0 0 0
F 7 8

0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 0
3 A 2 E

Conversão de hexadecimal para binário

Usamos o procedimento inverso: cada dígito hexadecimal é substituído pela representação binária do valor do dígito.



Conversão de hexadecimal para decimal

A Equação 1 pode ser utilizada para converter para a base decimal números representados em outras bases.

$$2 \quad A \quad 7 \quad \boxed{F} \longrightarrow 15$$

$$2 \quad A \quad \boxed{7} \quad F \longrightarrow 7 \times 16$$

$$2 \quad \boxed{A} \quad 7 \quad F \longrightarrow 10 \times 256$$

$$\boxed{2} \quad A \quad 7 \quad F \longrightarrow 2 \times 4096$$

$$15 + (7 \times 16) + (10 \times 256) + (2 \times 4096) = 10909$$

Conversão de binário para decimal

A conversão de números binários em decimais se faz de forma similar, mas como o multiplicador é sempre 0 ou 1, basta efetuar a soma das potências de 2 para as posições em que o dígito é 1.

$$\begin{array}{cccccccccl} 1 & 0 & 1 & 1 & 0 & 0 & 0 & \boxed{1} & \longrightarrow & 2^0 \\ 1 & 0 & 1 & \boxed{1} & 0 & 0 & 0 & 1 & \longrightarrow & 2^4 \\ 1 & 0 & \boxed{1} & 1 & 0 & 0 & 0 & 1 & \longrightarrow & 2^5 \\ \boxed{1} & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \longrightarrow & 2^7 \end{array}$$

$$1 + 16 + 32 + 128 = 177$$

Conversão de decimal para hexadecimal e binário

Considere novamente a Equação 1, que calcula o valor de um número N :

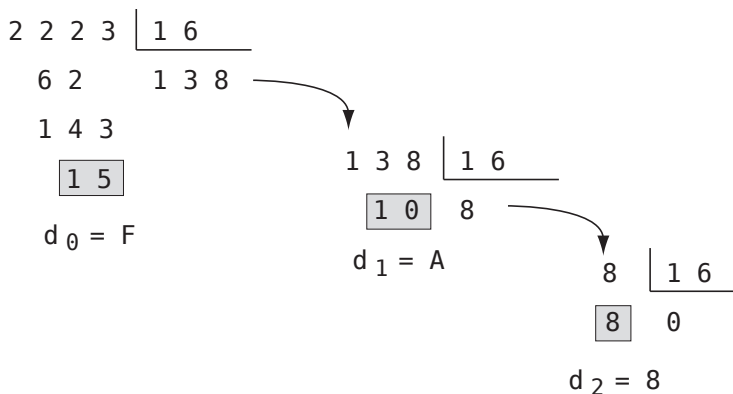
$$(\delta_n \times b^n) + (\delta_{n-1} \times b^{n-1}) + \dots + (\delta_2 \times b^2) + (\delta_1 \times b^1) + \delta_0$$

Se dividirmos N pela base b , o resto da divisão é δ_0 . Assim, partindo de um número decimal N , podemos determinar o dígito mais à direita na representação desse número na base b tomando o resto da divisão de N por b . O quociente dessa divisão é

$$(\delta_n \times b^{n-1}) + (\delta_{n-1} \times b^{n-2}) + \dots + (\delta_3 \times b^2) + (\delta_2 \times b^1) + \delta_1$$

e o processo continua até que o quociente seja zero.

Conversão de decimal para hexadecimal



Aritmética em hexadecimal

1

4	E	D	3
1	A	6	9
			C

+

$3 + 9 = 12 = C_{16}$

2

4	E	D	3
1	A	6	9
			3
			C

+

$(D + 6) = 19 = 16 + 3_{16}$

Escrevemos 3 e vai-um

3

1			
4	E	D	3
1	A	6	9
			9
			3
			C

+

$(1 + E + A)_{16} =$
 $1 + 14 + 10 =$
 $25 = 16 + 9$

Escrevemos 9 e vai-um

4

1			
4	E	D	3
1	A	6	9
			6
			9
			3
			C

+

$1 + 4 + 1 = 6$

Representação de inteiros sem sinal

- ▶ Um número com k dígitos na base b pode representar b^k valores distintos.
- ▶ Se quisermos representar somente números naturais (inteiros positivos), podemos representar os números de 0 a $b^k - 1$.
- ▶ Na representação binária com 16 bits, podemos representar naturais de 0 a $2^{16} - 1$, ou seja, de 0 a 65535.

Representação de inteiros sem sinal

- ▶ Normalmente queremos representar também números negativos.
- ▶ Vamos considerar três métodos de representação, em binário, de números inteiros com sinal: sinal e magnitude, complemento de um e complemento de dois.
- ▶ Os três métodos têm em comum o fato de reservar o bit mais significativo para indicar o sinal do valor.
- ▶ Por convenção, o bit de sinal igual a 1 significa que o valor representado é negativo.

Sinal e magnitude

- ▶ O bit mais à esquerda é usado para indicar o sinal e os demais bits para representar o valor absoluto.
- ▶ A operação de negação na representação sinal e magnitude é simples: basta inverter o bit de sinal. Exemplo:

Convertendo para binário:

$$4100 = 1004_{16} = 0001000000000100_2$$

Para converter em negativo, invertemos o bit de sinal:

0001000000000100
1001000000000100



$$-4100 = 1001000000000100_2 = 9004_{16}$$

Sinal e magnitude

Binário	Hexadecimal	Decimal
0111111111111111	7FFF	32767
0111111111111110	7FFE	32766
0111111111111101	7FFD	32765
...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1000000000000000	8000	-0
1000000000000001	8001	-1
1000000000000010	8002	-2
...
1111111111111101	FFFD	-32765
1111111111111110	FFFE	-32766
1111111111111111	FFFF	-32767

Números inteiros com 16 bits em sinal e magnitude.

Complemento de um


- ▶ O negativo de um número é obtido complementando (invertendo) todos os bits da representação binária.
- ▶ Complementar um bit b é o mesmo que subtrair o valor de b de 1.
- ▶ Exemplo:

Convertendo para binário:

$$4100 = 1004_{16} = 0001000000000100_2$$

Para converter em negativo, invertemos todos os bits:

0001000000000100
1110111111111011



$$-4100 = 1110111111111011_2 = \text{EFFF}_{16}$$

Complemento de um

Binário	Hexadecimal	Decimal
0111111111111111	7FFF	32767
0111111111111110	7FFE	32766
0111111111111101	7FFD	32765
...
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1111111111111111	FFFF	0
1111111111111110	FFFE	-1
1111111111111101	FFFD	-2
...
1000000000000010	8002	-32765
1000000000000001	8001	-32766
1000000000000000	8000	-32767

Números inteiros com 16 bits em complemento de um.

Complemento de dois

- ▶ Todas as representações com o bit de sinal igual a um são usadas para representar um valor negativo (não há duplicidade de zero).
- ▶ Em complemento de dois o valor absoluto do maior número positivo representável é menor que o valor absoluto do menor número negativo, pois a representação complemento de dois é capaz de representar um valor negativo a mais.

Complemento de dois

Binário	Hexadecimal	Decimal
0111111111111111	7FFF	32767
0111111111111110	7FFE	32766
0111111111111101	7FFD	32765
...
0000000000000100	0004	4
0000000000000011	0003	3
0000000000000010	0002	2
0000000000000001	0001	1
0000000000000000	0000	0
1111111111111111	FFFF	-1
1111111111111110	FFFE	-2
1111111111111101	FFFD	-3
1111111111111101	FFFC	-4
...
1000000000000010	8002	-32766
1000000000000001	8001	-32767
1000000000000000	8000	-32768

Números inteiros com 16 bits em complemento de dois.

Complemento de dois

- ▶ O negativo de um número na representação complemento de dois é obtido complementando esse número em relação a 2.
- ▶ Complementar um bit em relação a 2 significa subtrair o bit de 2.
- ▶ Complementar um número N representado em uma sequência de k bits significa subtrair de 2 cada bit da sequência, ou seja, efetuar $2^k - N$ (note que é necessário usar $k + 1$ bits para efetuar essa subtração).
- ▶ Por exemplo, considere o valor decimal 4 representado em 16 bits (0004_{16}). Para encontrar a representação de -4 , calculamos $2^{16} - 4$.
- ▶ Em hexadecimal, essa conta é $10000_{16} - 0004_{16} = \text{FFFC}_{16}$.

Complemento de dois

Em binário:

$$\begin{array}{r} 1\ 0000\ 0000\ 0000\ 0000 \\ \quad 0000\ 0000\ 0000\ 0100 \\ \hline 1111\ 1111\ 1111\ 1100 \end{array}$$

Uma maneira de obter o negativo do valor 4 em complemento de dois:

original = 0000 0000 0000 0100

$\begin{array}{c} \leftarrow \text{mantém até o primeiro 1} \\ 0000\ 0000\ 0000\ 0100 \\ \leftarrow \text{inverte os restantes} \end{array}$

resultado = 1111 1111 1111 1100

Complemento de dois – adição

- ▶ O procedimento para adição de dois números em complemento de dois também é muito simples.
- ▶ A operação é efetuada como se os operandos fossem números sem sinal, ou seja, tratando o bit de sinal como os outros, e desprezando o bit de vai-um, se houver.
- ▶ Como o procedimento de adição não leva em conta os sinais dos operandos, sua implementação no hardware do processador é muito menos custosa.
- ▶ O resultado obtido é correto independentemente dos sinais dos operandos!

Estouro de campo

- ▶ Como os operandos e o resultado de uma operação aritmética são armazenados em palavras de memória de tamanho fixo, o resultado de qualquer operação aritmética realizada pelo processador só é válido se puder ser representado utilizando o número de bits da palavra do processador.
- ▶ Caso contrário o resultado não é correto, ocorrendo uma condição de erro chamada de estouro de campo (em inglês, *overflow*).
- ▶ Considere uma palavra de apenas três bits e a representação complemento de dois.

Estouro de campo

Binário	Decimal (com sinal)	Decimal (sem sinal)
011	3	3
010	2	2
001	1	1
000	0	0
111	-1	7
110	-2	6
101	-3	5
100	-4	4

Números inteiros com e sem sinal representados em uma palavra de três bits.

Considere a operação $011_2 + 100_2 = 111_2$.

- ▶ Inteiros sem sinal: a operação em decimal é $3 + 4 = 7$.
- ▶ Inteiros com sinal: a operação é $3 + (-4) = -1$.
- ▶ Em ambos os casos, o resultado pode ser expresso em três bits, e a operação está correta.

Vamos examinar agora a operação $011_2 + 001_2 = 100_2$.

- ▶ Inteiros sem sinal: a operação em decimal é $3 + 1 = 4$, cujo resultado está correto pois pode ser representado em três bits.
- ▶ Inteiros com sinal: a operação em decimal é $3 + 1 = -4$, o que está obviamente errado.
- ▶ A operação não tem o resultado correto pois este não pode ser expresso em três bits.

Estouro de campo – adição sem sinal

- ▶ Estouro de campo depende não só da operação realizada e tamanho da palavra, mas também da representação utilizada (com ou sem sinal).
- ▶ Para detectar estouro de campo em operações de adição entre números sem sinal, basta verificar a ocorrência de vai-um do bit mais significativo do resultado.
- ▶ Ocorrência de vai-um indica que o resultado necessita de mais bits do que os existentes para ser representado.

Estouro de campo – adição com sinal

- ▶ Note que a adição de dois números de sinal diferentes nunca produz estouro de campo, portanto podemos restringir nossa análise a adições entre números de mesmo sinal.
- ▶ Se ambos os operandos são negativos, ocorre estouro de campo quando o resultado for um número positivo.
- ▶ Similarmente, ocorre estouro de campo quando o resultado da adição entre dois números positivos for um número negativo.

Estouro de campo em linguagem de montagem

- ▶ Como veremos, em linguagem de montagem a ocorrência de estouro de campo pode ser consultada pelo programador após a execução de uma operação aritmética.
- ▶ o programador pode decidir se trata ou não a condição de erro.
- ▶ A linguagem C despreza a informação de estouro de campo!

Representação de números fracionários

A Equação 1 pode ser estendida para considerar números fracionários:

$$(\delta_n \times b^n) + (\delta_{n-1} \times b^{n-1}) + \dots + (\delta_2 \times b^2) + (\delta_1 \times b^1) + \delta_0 + \\ (\delta_{-1} \times b^{-1}) + (\delta_{-2} \times b^{-2}) + \dots + (\delta_{-p} \times b^{-p}) \quad (2)$$

A representação do número fracionário dado pela Equação acima na base b é:

$$d_n d_{n-1} \dots d_2 d_1 d_0, d_{-1} d_{-2} \dots d_{-p} \quad (3)$$

onde d_i representa um dígito na base b . Denominamos p a *precisão* do número representado.

- ▶ Podemos fazer com que o dígito mais à esquerda seja sempre diferente de zero, para economizar dígitos na representação.
- ▶ Podemos também ajustar a posição da vírgula, multiplicando ou dividindo a representação pela base elevada a alguma potência
- ▶ Exemplos:

$$472,8 = 4,728 \times 10^2$$

(a)

$$0,00235 = 2,35 \times 10^{-3}$$

(b)

- ▶ A representação com apenas um dígito antes da vírgula é chamada de *normalizada*.
- ▶ Na forma mais geral, a representação normalizada de um número fracionário na base b é:

$$d_0 , d_{-1}d_{-2} \dots d_{-p+1}d_{-p} \times b^{exp} \quad (4)$$

- ▶ Os dígitos de d_0 a d_{-p} são chamados de *mantissa* ou *coeficiente* da representação.

Conversão de base de números fracionários

- ▶ A conversão de base de números fracionários é realizada em dois passos.
 1. A parte inteira do número é convertida para a nova base, utilizando o método já visto.
 2. A parte fracionária é convertida para a nova base.
- ▶ Considere a parte fracionária da Equação 2:

$$(\delta_{-1} \times b^{-1}) + (\delta_{-2} \times b^{-2}) + \dots + (\delta_{-p} \times b^{-p}) \quad (5)$$

- ▶ A representação do valor da Equação 5 na base b é

$$0, d_{-1}d_{-2} \dots d_{-p+1}d_{-p}$$

Conversão de base de números fracionários

Se multiplicarmos a Equação 5 pela base b obtemos o valor

$$\delta_{-1} + (\delta_{-2} \times b^{-1}) + \dots + (\delta_{-p} \times b^{-p+1})$$

cuja representação na base b é

$$d_{-1} , d_{-2} \dots d_{-p+1} d_{-p}$$

e assim conseguimos determinar o dígito mais significativo da parte fracionária, cujo valor é δ_{-1} e cuja representação é d_{-1} .

O processo é repetido até que a parte descartada seja zero, ou que atinjamos a precisão desejada.

Conversão de número fracionário decimal para binário

1) Parte inteira

$$13 = 1101_2$$

2) Parte fracionária

$$0,625 \times 2 = 1,250 \quad d_{-1} = 1$$

$$0,250 \times 2 = 0,5 \quad d_{-2} = 0$$

$$0,5 \times 2 = 1,0 \quad d_{-3} = 1$$

Resultado:

$$13,625 = 1101,101_2$$

- ▶ O campo *S* armazena o sinal do valor representado.
- ▶ O campo *mantissa* armazena a mantissa normalizada.
- ▶ O campo *expoente* armazena o expoente do número fracionário. É interpretado como um valor inteiro sem sinal, e o valor armazenado deve ser o resultado da expressão $expoente = exp + bias$ onde exp é o expoente da representação normalizada e $bias$ tem o valor 127 para o formato curto e é 1023 para o formato longo.

Decimal	Sinal	Expoente	Mantissa	Hexadecimal
1,0	0	01111111	000000000000...0000	3F800000
-1,0	1	01111111	000000000000...0000	BF800000
-2,5	1	10000000	010000000000...0000	C0200000
-126,75	1	10000101	11111011000...0000	C2FD8000
0,1	0	00111101	10011001100...1100	3DCCCCC

Alguns números no formato IEEE754 curto.

Representação de caracteres

- ▶ Cada caractere é representado por um “código” (valor único predefinido para cada caractere)
- ▶ Diversos padrões: ASCII, UTF-8, LATIN-1,...

Codificação ASCII

Caractere	Decimal	Hexadecimal
...
volta-um-espaco	8	08
tab-horizontal	9	09
nova-linha	10	0A
...
espaco	32	20
!	33	21
“	34	22
...
0	48	30
1	49	31
2	50	32
...
8	56	38
9	57	39
...

Codificação ASCII

Caractere	Decimal	Hexadecimal
...
A	65	41
B	66	42
C	67	43
...
Y	89	59
Z	90	5A
...
a	97	61
b	98	62
c	99	63
...
y	121	79
z	122	7A
...

Representação de caracteres

	1	2		M	a	ç	ã	s		
...	31	32	20	4D	61	E7	E3	73	00	...

codificação ISO-LATIN-1

	1	2		M	a	ç		ã		s		
...	31	32	20	4D	61	C3	A7	C3	A3	73	00	...

codificação UTF-8

Diferentes representações da mesma cadeia de caracteres.

Ordem de bytes na palavra

Memória

byte 5	endereço_byte = 5
byte 4	endereço_byte = 4
byte 3	endereço_byte = 3
byte 2	endereço_byte = 2
byte 1	endereço_byte = 1
byte 0	endereço_byte = 0

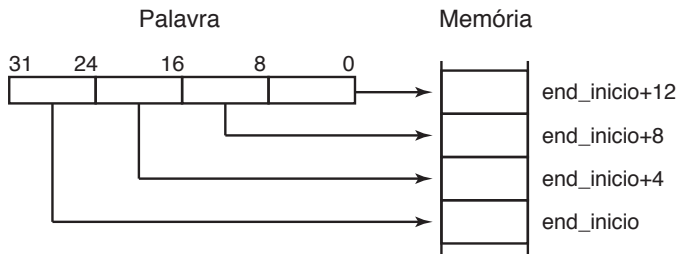
(a) Memória como vetor de bytes

Memória

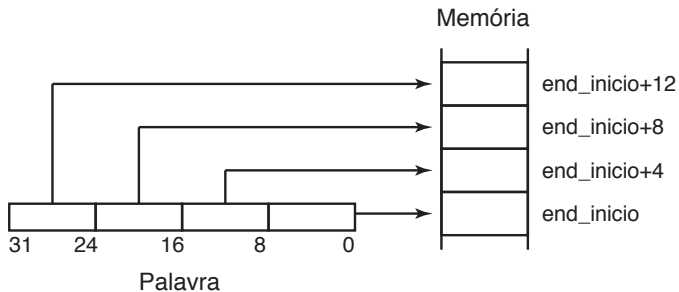
palavra 3	endereço_byte = 12
palavra 2	endereço_byte = 8
palavra 1	endereço_byte = 4
palavra 0	endereço_byte = 0

(b) Memória como vetor de palavras

Ordem *big-endian*



Ordem *little-endian*



Exemplos

Memória (em palavras)

77 66 55 44	end_inic+4
33 22 11 00	end_inic

Memória (em bytes)

77	end_inic+7
66	end_inic+6
55	end_inic+5
44	end_inic+4
33	end_inic+3
22	end_inic+2
11	end_inic+1
00	end_inic

Little-endian

Memória (em bytes)

44	end_inic+7
55	end_inic+6
66	end_inic+5
77	end_inic+4
00	end_inic+3
11	end_inic+2
22	end_inic+1
33	end_inic

Big-endian