MC102 – Introdução à Programação de Computadores

Aula 2: Representação de Números

Felipe P.G. Bergo

1 Memória

A memória dos computadores pode ser vista como uma longa sequência de *chaves*, *botões liga-desliga* ou *interruptores*. Cada chave tem duas posições possíveis: **desligada** ou **ligada**; Cada unidade de informação básica, com dois valores possíveis, é denominada **bit**. A memória dos computadores pode ser vista como um conjunto de bits, que podem ser ligados ou desligados pelos programas (Figura 1). Vamos chamar o estado desligado de 0 (zero) e o estado ligado de 1 (um). Nos dias de hoje, a memória RAM dos computadores é construída com capacitores: cada bit de memória é um capacitor. O capacitor carregado representa um 1, e o capacitor descarregado representa um 0. Memórias mais rápidas podem ser implementadas com transistores (semicondutores), mas o custo por bit é maior. A memória "cache" e os elementos de memória presentes dentro da CPU são feitos com transistores.

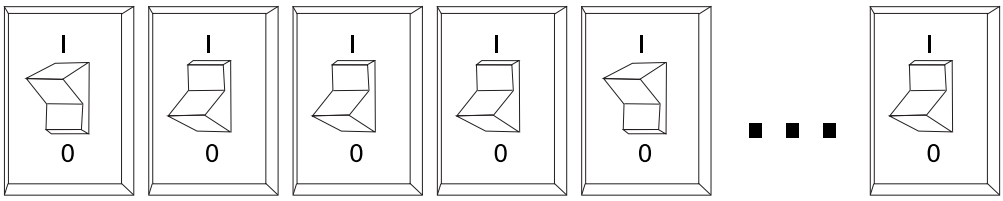

Figura 1: Bits podem ser vistos como interruptores.

Para dizer qual bit da memória queremos ligar ou desligar, poderíamos considerar que eles estão posicionados lado a lado e enumerados de forma crescente da esquerda para a direita. Assim, poderíamos dar uma instrução na forma "Ligue o Bit número 4", ou "Troque a posição do Bit número 6". Entretanto, um bit sozinho representa muito pouca informação, então agrupamos os bits em grupos de 8 bits. Um grupo de 8 bits é chamado de **byte**. A memória dos computadores é *endereçada* por bytes. Os comandos de baixo nível são da forma "Ponha os bits 01110010 no byte número 80 da memória". Este número "80" é chamado de "endereço de memória". Toda a informação de nossos programas (conteúdo de variáveis, texto, números, imagens) precisa ser representada na forma de zeros e uns. Cada variável declarada no C ocupa um ou mais bytes da memória do computador.

Podemos imaginar a memória do computador como uma sequência de escaninhos de correspondência, cada um com um número (endereço) e a capacidade de guardar 8 bits de informação (Figura 2).

Nas próximas seções vamos estudar como representar números como sequências *binárias* e como

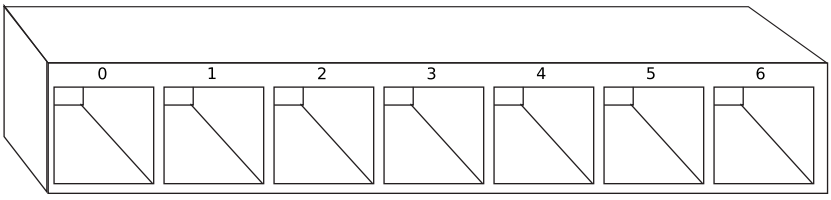

Figura 2: A memória como escaninhos de bytes.

realizar aritmética elementar sem ter que converter entre esta representação e a nossa representação decimal.

2 Base 10

Nossa representação numérica usual é realizada em **base 10**, ou **decimal**. Em geral aprendemos a realizar operações aritméticas sem perceber a importância da base, decorando tabuadas. Na notação decimal, cada dígito multiplica uma diferente potência de 10. A casa das unidades multiplica 10^0. Cada casa à esquerda multiplica uma potência de dez a mais, e cada casa à direita multiplica uma potência de 10 a menos. As casas fracionárias multiplicam potências de expoentes negativos:

$$459.75 = 4 \times 10^2 + 5 \times 10^1 + 9 \times 10^0 + 7 \times 10^{-1} + 5 \times 10^{-2}$$

3 Sistema Binário

A forma mais simples de representar números inteiros positivos em bits é utilizar diretamente o sistema binário (base 2). Desta forma conseguimos representar os valores $0 \ldots 255$ (0 a $2^8 - 1$) com um byte. Assim como cada posição decimal está associada a uma potência de 10, cada bit está associado a uma potência de 2. Em um byte, temos bits representando de $2^0 = 1$ a $2^7 = 128$ (Figura 3).

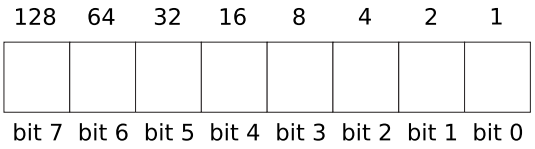

Figura 3: "Valores" dos bits.

O bit que representa a maior potência de 2 é chamado MSB (Most Significant Bit – Bit Mais Significativo). O bit que representa a menor potência de 2 é chamado LSB (Least Significant Bit – Bit Menos Significativo). Os bits são enumerados do LSB para o MSB. O LSB é o bit 0.

3.1 Conversão de Binário para Decimal

A conversão de um número representado em binário para o sistema decimal pode ser realizada com a soma das multiplicações pelas potências de 2, como no exemplo do 459.75 acima. No caso particular do sistema binário, cada potência de 2 é somada (bit 1) ou ignorada (bit 0), portanto a conversão é uma soma simples:

$$01101100_{(2)} = 64_{(10)} + 32_{(10)} + 8_{(10)} + 4_{(10)} = 108_{(10)}$$

3.2 Conversão de Decimal para Binário

A conversão inversa pode ser feita de duas maneiras diferentes: da esquerda para a direita ou da direita para a esquerda.

Esquerda para a direita. Identifique a maior potência de 2 que "cabe" dentro do número a ser convertido. O bit associado àquela potência será 1. Subtraia esta potência de 2 do valor a ser convertido. Continue o processo até a subtração dar zero.

Exemplo: Converter 179 para binário. A maior potência de 2 que "cabe" dentro de 179 é 128 (2^7). Temos $10000000_{(2)}$ e $179 - 128 = 51$. A maior potência de 2 que cabe em 51 é 32 (2^5). Temos $10100000_{(2)}$ e $51 - 32 = 19$. A maior potência de 2 que cabe em 19 é 16. Temos $10110000_{(2)}$ e $19 - 16 = 3$. A maior potência de 2 que cabe em 3 é 2. Temos $10110010_{(2)}$ e $3 - 2 = 1$. A maior potência de 2 que cabe em 1 é 1. Temos $10110011_{(2)}$. Como $1 - 1 = 0$, a conversão está completa.

De forma mais formal, a cada passo estamos identificando $b = \lfloor \log_2 x \rfloor$ e setando[1] o bit associado a 2^b, e continuamos o processo até que $x - \lfloor \log_2 x \rfloor = 0$.

Direita para a Esquerda. Este método usa duas propriedades simples: números ímpares têm o bit 1 setado, e para dividir um número binário por 2, basta deslocar seus bits uma posição para a direita. Assim, faremos divisões sucessivas para trazer os bits mais altos para a posição do bit 0, e usamos a propriedade par/ímpar para setar/resetar os bits. Algoritmo: Se x é par, adicione um bit 0 à esquerda do número binário, senão escreva um bit 1 à esquerda do número binário. Faça $x \leftarrow \lfloor x \div 2 \rfloor$ (jogue a parte fracionária fora, se houver – não arredonde). Se $x \neq 0$, repita o processo.

Exemplo: Converter 179 para binário. 179 é ímpar, então escrevemos $1_{(2)}$. $\lfloor 179 \div 2 \rfloor = 89$, que é ímpar. Escrevemos $11_{(2)}$. $\lfloor 89 \div 2 \rfloor = 44$, que é par. Escrevemos $011_{(2)}$. $\lfloor 44 \div 2 \rfloor = 22$, que é par. Escrevemos $0011_{(2)}$. $\lfloor 22 \div 2 \rfloor = 11$, que é ímpar. Escrevemos $10011_{(2)}$. $\lfloor 11 \div 2 \rfloor = 5$, que é ímpar. Escrevemos $110011_{(2)}$. $\lfloor 5 \div 2 \rfloor = 2$, que é par. Escrevemos $0110011_{(2)}$. $\lfloor 2 \div 2 \rfloor = 1$, que é ímpar. Escrevemos $10110011_{(2)}$. $\lfloor 1 \div 2 \rfloor = 0$, então paramos e a conversão terminou.

[1] Setar significa ligar, acender, atribuir 1. Resetar significa desligar, apagar, atribuir 0.

3.3 Soma

A soma binária é realizada de forma semelhante à soma decimal. Precisamos lembrar uma tabuada básica: $0_{(2)} + 0_{(2)} = 0_{(2)}$, $0_{(2)} + 1_{(2)} = 1_{(2)}$, $1_{(2)} + 1_{(2)} = 10_{(2)}$ e $1_{(2)} + 1_{(2)} + 1_{(2)} = 11_{(2)}$. O último caso será usado nos casos de vai-um.

3.4 Números Negativos

O sistema mostrado até agora representa apenas números positivos. Um forma trivial de representar números negativos é usar um bit a mais para representar o sinal do número (0 para positivo e 1 para negativo, por exemplo). Entretanto, esta forma traz dois grandes inconvenientes: A soma aritmética precisa verificar o sinal para tratar somas e subtrações separadamente; e o número 0 passaria a ter duas representações válidas (+0 e −0).

Os computadores modernos usam uma representação de *complemento de dois*, que resolve ambos os problemas. Somas de valores de qualquer sinal podem ser realizadas com o mesmo algoritmo de soma binária sem sinal, e o zero tem apenas uma representação.

Para obter o complemento de 2 de um número (em binário), invertemos seus bits e somamos 1. Exemplo: $5_{(10)} = 00000101_{(2)}$. Invertendo os bits temos $11111010_{(2)}$. Somando 1, temos $-5_{(10)} = 11111011_{(2)}$. Note que o byte 11111011 pode representar tanto um valor positivo como um valor negativo. Sempre que tratamos um dado, precisamos saber se estamos usando notação de complemento de 2 ou valores sem sinal.

Usando complemento de 2 ($\overline{2}$) com n bits podemos representar valores entre -2^{n-1} e $2^{n-1} - 1$. Se usarmos os mesmos n bits para representar inteiros sem sinal, podemos representar valores entre 0 e $2^n - 1$. A tabela abaixo mostra os limites de representação com valores comuns de n e os tipos usados em C em PCs de 32 bits para representá-los:

Bits	Bytes	Limites	Representação	Tipo C
8	1	-128 a 127	$\overline{2}$	char
8	1	0 a 255	Sem sinal	unsigned char
16	2	-32 768 a 32 767	$\overline{2}$	short int
16	2	0 a 65 535	Sem sinal	unsigned short int
32	4	-2 147 483 648 a 2 147 483 647	$\overline{2}$	int
32	4	0 a 4 294 967 296	Sem sinal	unsigned int
64	8	-9 223 372 036 854 775 808 a 9 223 372 036 854 775 807	$\overline{2}$	long long
64	8	0 a 18 446 744 073 709 551 616	Sem sinal	unsigned long long

4 Números de Ponto Flutuante

Não é possível representar um número real qualquer com um número finito de bits. A representação de números racionais através de seu numerador e denominador inteiros pode exigir muitos bits

para aplicações práticas. Em particular, gostaríamos de poder representar tanto números muito grandes como muito pequenos. A solução "manual" para este problema é a notação científica (Ex: 3.5×10^{37}), em que representamos um número através de uma *mantissa* (3.5), uma *base* e um *expoente* (37).

Números reais são representados por aproximações de ponto flutuante (o expoente desloca o ponto sem modificar o número de dígitos significativos da mantissa). As representações de ponto flutuante de quase todos os computadores atuais seguem o padrão IEEE 754, definido em 1985. Este padrão define quatro tamanhos de números de ponto flutuante com 32, 43 (raro), 64 e 80 bits. Todos os processadores modernos usam o padrão IEEE 754 internamente, por isso o uso do padrão evita conversões na linguagem de máquina.

O padrão IEEE754 divide os bits em 3 campos. Os tamanhos variam entre os 4 tipos, vamos dar exemplos com o formato de 32 bits (Figura 4).

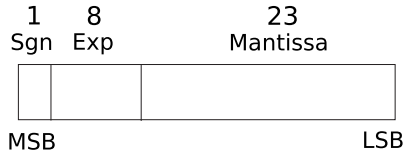

Figura 4: Tipo IEEE 754 de 32 bits.

O campo de sinal (bit mais significativo) armazena o sinal do número (0 para números positivos, 1 para números negativos).

O valor do número armazenado é dado por $\pm 1.Mantissa \times 2^{(Exp-127)}$. O sinal é dado pelo bit de sinal.

O campo de expoente é armazenado como um inteiro positivo. Para obter o verdadeiro expoente, subtrai-se 127 do valor armazenado. O tipo de 32 bits pode armazenar expoentes entre -126 e $+127$ (1 a 254 no campo de expoente. Os valores 0 e 255 são usados para representar valores especiais.

O campo de mantissa é representado de forma *normalizada*. Isto evita que um mesmo número tenha mais que uma representação válida. Assim como em notação científica costumamos deixar apenas um dígito antes do ponto (3.5×10^7 em vez de 35×10^6), representamos a mantissa de forma que haja exatamente um dígito 1 antes do ponto, e armazenamos apenas a parte fracionária no campo de mantissa. O único número que não tem nenhum 1 em sua representação binária é o zero, que tem uma representação especial.

O padrão IEEE 754 permite a representação de 3 valores especiais: Zero (preenchendo os campos Exp e Mantissa com 0), Infinito (preenchendo o campo Exp com 1 e a Mantissa com 0) e Not-a-Number (NaN, preenchendo o campo Exp com 1 e a Mantissa com qualquer valor diferente de 0). Infinito e NaN admitem sinal.

O tipo de 64 bits usa 1 bit de sinal, 11 bits de expoente (subtrai-se 1023 do valor armazenado para obter o expoente real) e 52 bits de mantissa.

Em C, o tipo *float* representa o IEEE754 de 32 bits, e *double* representa o IEEE754 de 64 bits. O

tipo *long double* pode variar entre computadores diferentes, mas no PC representa o IEEE754 de 80 bits.

O número de casas decimais representadas pode ser encontrado por $\frac{p}{\log_2 10}$, onde p é o número de bits da mantissa (incluindo o 1 implícito – $p = 24$ no tipo de 32 bits). Na prática, temos 7 dígitos decimais significativos com o tipo float e 15 dígitos decimais significativos com o tipo double.

Exemplo de representação: Para representar o número -118.625 em IEEE754 de 32 bits: O bit de sinal é 1 (número negativo). Convertendo 118.625 para binário, temos $1110110.101_{(2)}$. Movendo o ponto decimal para a esquerda até manter apenas um 1 na parte inteira, temos $1.110110101_{(2)} \times 2^6$. A mantissa será a parte fracionária, completada com zeros à direita até preencher 23 bits: 11011010100000000000000. O expoente é 6. Somamos 127 e obtemos 133, que é $10000101_{(2)}$. A representação de -118.625 em ponto flutuante será 1 10000101 11011010100000000000000 ou, sem a divisão de campos, 11000010111011010100000000000000.

5 Outras Bases

Como pode ser notado no exemplo de conversão entre decimal e ponto flutuante, a representação binária de números pode se tornar muito longa. Bases que são potências de dois permitem um mapeamento direto entre grupos de bits na representação binária e dígitos na nova base. As duas bases mais utilizadas são 8 (octal) e 16 (hexadecimal). No sistema octal usamos os dígitos de 0 a 7, e na base 16 usamos os dígitos 0 a 9 e as letras A a F (representando 11 a 15). Em octal, cada dígito corresponde a 3 bits. Em hexadecimal, cada dígito corresponde a 4 bits.

6 Representação de Texto

Em geral usamos um byte para representar cada caractere de texto, armazenado no tipo char. O padrão mais usado para mapeamento entre caracteres e valores é o ASCII (American Standard Code for Information Interchange), definido em 1963. Sendo um padrão americano, não permite a representação de caracteres acentuados, cirílico, kanji, letras gregas. ASCII define os caracteres de 0 a 127 (portanto usa apenas 7 bits, os valores positivos de uma representação complemento de 2 de 8 bits). Destes, os caracteres 32 a 126 são imprimíveis. Os caracteres 0–31 e 127 representam códigos de controle (quebra de linha, tabulação, entre outros). A Figura 5 mostra os caracteres ASCII imprimíveis, numerados em hexadecimal e decimal.

Como a maioria dos computadores usa um byte de 8 bits para armazenar caracteres, e há várias tentativas de extensão do padrão ASCII para permitir a exibição de caracteres extras (acentos, letras gregas) usando os valores 128–255. A extensão padrão é o ISO-8859-1, também chamado de Latin-1, que inclui caracteres para as línguas latinas (Figura 6).

Existem 15 tabelas ISO-8859-X para tratar de famílias específicas. Estas tabelas não conseguem resolver o problema de programas que precisam escrever em várias línguas em um mesmo documento. Para acabar com estes problemas, em 1990 foi proposto o padrão Unicode. É um padrão complexo, que não usaremos em nossos programas (embora Linux suporte seu uso).

	2	3	4	5	6	7
0:		0	@	P	‘	p
1:	!	1	A	Q	a	q
2:	"	2	B	R	b	r
3:	#	3	C	S	c	s
4:	$	4	D	T	d	t
5:	%	5	E	U	e	u
6:	&	6	F	V	f	v
7:	’	7	G	W	g	w
8:	(	8	H	X	h	x
9:	)	9	I	Y	i	y
A:	*	:	J	Z	j	z
B:	+	;	K	[	k	{
C:	,	<	L	\	l	|
D:	-	=	M	]	m	}
E:	.	>	N	^	n	~
F:	/	?	O	_	o	

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:		*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V	‘	j	t	~
7:	%	/	9	C	M	W	a	k	u	
8:	&	0	:	D	N	X	b	l	v	
9:	’	1	;	E	O	Y	c	m	w	

Figura 5: Tabela de caracteres ASCII imprimíveis.

ISO/IEC 8859-1															
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*unused*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*unused*															
9x																
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figura 6: Tabela de caracteres do ISO-8859-1.

7 Múltiplos Decimais e Binários de Bytes

O sistema internacional de unidades define os prefixos kilo, mega, giga, tera, peta, exa, zetta e yotta como potências de 10. Na computação, entretanto, é comum representar grupos de bytes em termos de potências de 2. Ao longo dos anos, usou-se estes prefixos para representar potências de 2. Em 1999 foi aceito pela IEC um padrão de prefixos para desambiguar as unidades, embora ainda seja comum ver o uso incorreto dos prefixos do SI. As tabelas abaixo mostram as quantidades representadas pelos dois sistemas.

Prefixo	Quantidade	Unidade	Abreviação
kilo	10^3	kilobyte	kB
mega	10^6	megabyte	MB
giga	10^9	gigabyte	GB
tera	10^{12}	terabyte	TB
peta	10^{15}	petabyte	PB
exa	10^{18}	exabyte	EB
zetta	10^{21}	zettabyte	ZB
yotta	10^{24}	yottabyte	YB

Figura 7: Prefixos Decimais.

Prefixo	Quantidade	Unidade	Abreviação
kibi	2^{10} = 1 024	kibibyte	KiB
mebi	2^{20} = 1 048 576	mebibyte	MiB
gibi	2^{30} = 1 073 741 824	gibibyte	GiB
tebi	2^{40} = 1 099 511 627 776	tebibyte	TiB
pebi	2^{50} = 1 125 899 906 842 624	pebibyte	PiB
exbi	2^{60} = 1 152 921 504 606 846 976	exbibyte	EiB

Figura 8: Prefixos Binários.

8 Leituras Adicionais

Para maiores detalhes sobre o sistema IEEE754, consulte `http://en.wikipedia.org/wiki/IEEE_754` (e as referências na seção de links externos). Padrões ISO 8859: `http://en.wikipedia.org/wiki/ISO_8859`.