

Representação de Dados (inteiros não negativos)

Representação da informação

- Computadores armazenam/processam informações representadas por “sinais” de dois valores/estados (bits)
- Ao invés de acessar bits individuais, computadores usam “blocos” de 8 bits (bytes) como a menor unidade endereçável
 - memória pode ser vista como um “array” de bytes
- Agrupando conjuntos de bits podemos representar valores numéricos
 - Ex. sequência de bits → representação em notação posicional, base 2
 - representação de valores numéricos podem utilizar diferentes tamanhos (número de bytes)

Notação posicional

- A base determina o número de dígitos utilizados para representar números
 - no sistema decimal, a base é 10, e utiliza 10 dígitos, de 0 a 9
 - no sistema binário, a base é 2 e utiliza 2 dígitos, 0 e 1
 - no sistema hexadecimal, a base é 16 e utiliza os dígitos de 0 a 9 e as letras de A a F
- O “valor” de cada dígito é a sua multiplicação pela base elevada à posição que ele ocupa. A soma dessas multiplicações é o valor numérico representado:

$$1011_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11_{10}$$

Conversão entre representações

- Uma tarefa comum ao trabalhar com programas “assembly” é converter entre representações decimais, binárias e hexadecimais
 - desconforto em trabalhar com base muito pequena
 - tamanho da base ~ tamanho da representação) → $15213_{10} = 11101101101101_2$
 - notação decimal é inadequada para representar padrões de bits e endereços de memória
- O uso da notação hexadecimal (base 16) é muito comum em software básico
 - Mapeamento direto para a base 2

Notação hexadecimal

- Cada 2 dígitos hexadecimais representam um byte

– $0xAA = 10101010_2$

- Para converter de hexa para binário basta expandir cada dígito hexadecimal

hexa	3	A	4	C
binário	0011	1010	0100	1100

- Para converter de binário para hexa basta converter cada grupo de 4 bits

binário	11	1100	1010	1101
hexa	3	C	A	D

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

Conversão de base b para base 10

- Uso direto da definição da notação posicional

– Hexadecimal para decimal

$$0x2ABC = 2 \cdot 16^3 + 10 \cdot 16^2 + 11 \cdot 16^1 + 12 \cdot 16^0 = 10940_{10}$$

– Binário para decimal

$$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$$

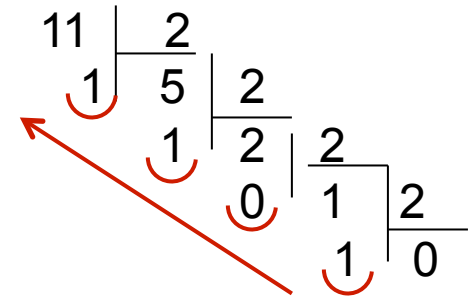
Conversão de base 10 para base b

- Divisões sucessivas por b (relação com notação posicional)

$$11_{10} = 5 * 2^1 + 1 * 2^0$$

$$5 = 2 * 2^1 + 1 * 2^0 \rightarrow 11 = (2 * 2^1 + 1 * 2^0) * 2^1 + 1 * 2^0 = 2 * 2^2 + 1 * 2^1 + 1 * 2^0$$

$$2 = 1 * 2^1 + 0 * 2^0 \rightarrow 11 = (1 * 2^1 + 0 * 2^0) * 2^2 + 1 * 2^1 + 1 * 2^0 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$

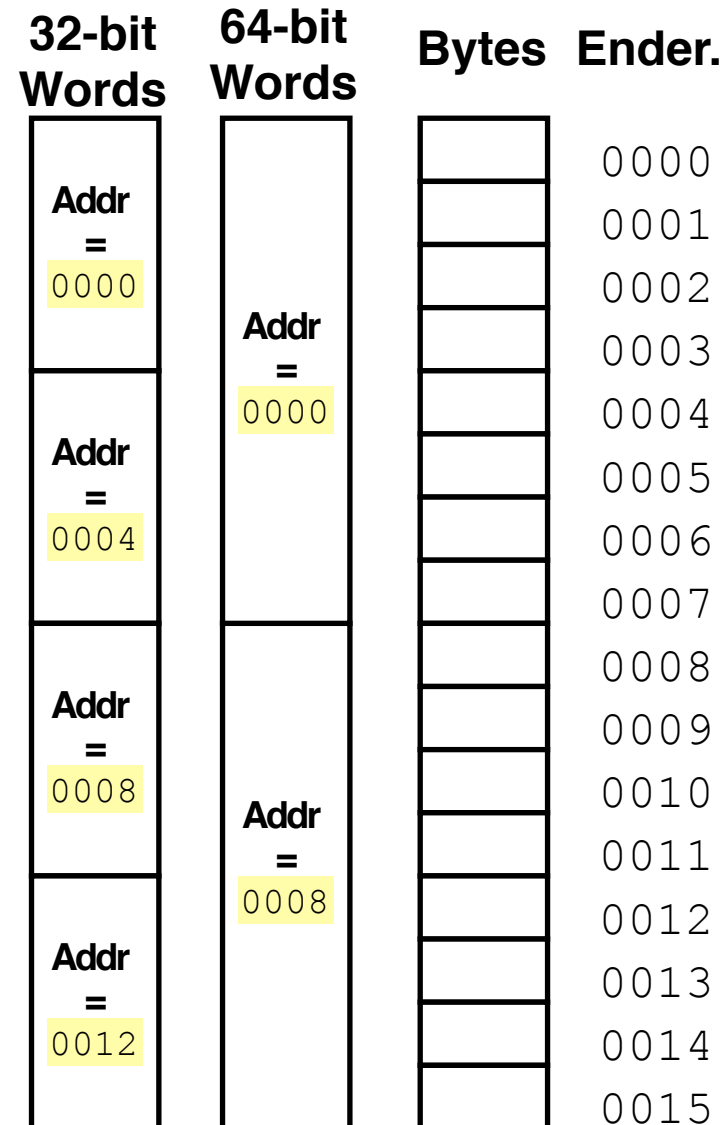


Palavras (words)

- Cada máquina (arquitetura) tem seu tamanho de palavra:
 - número de bits transferidos em um *chunk* entre memória e CPU
 - número de bits de endereços (pointer)
 - número de bits dos registradores !
- A maioria das máquinas hoje (ainda?) tem palavra de 32 bits (4 bytes)
 - máquinas de 64 bits estão se tornando mais comuns ...
- A representação de diferentes tipos de dados pode ocupar uma fração ou um múltiplo do tamanho da palavra, mas sempre **um número inteiro de bytes**
 - os processadores tem instruções para manipular valores inteiros representados por 2, 4, 8 bytes e números “floating point”

Memória orientada a palavras

- Memória é um vetor de bytes (cada um com um endereço), mas acesso ocorre palavra a palavra
- Endereço corresponde ao primeiro byte da palavra
- Endereços de palavras subsequentes diferem de 4 em máquina de 32 bits



Limitações de Representação

- Representação de valores numéricos podem utilizar diferentes tamanhos (número de bytes)
- Representação de inteiros não negativos
 - com 1 byte podemos representar inteiros de 0 a 255 (2^8-1)
 - com 2 bytes, inteiros de 0 a 65535 ($2^{16}-1$)
 - com 4 bytes, inteiros de 0 a 4294967295 ($2^{32}-1$)
- Essa limitação vale também para endereços
 - com 2 bytes só podemos endereçar 64KB de memória
 - com 4 bytes podemos endereçar 4GB de memória

Tamanhos de Tipos em C

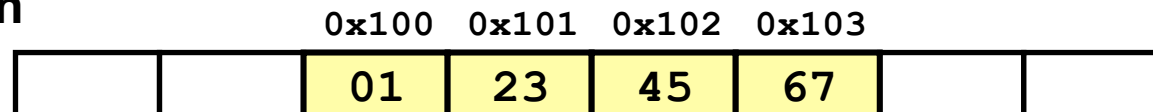
- A linguagem C define uma variedade de tipos que representam inteiros em diferentes intervalos
 - para inteiros não negativos: prefixar declaração com **unsigned**
 - o número de bytes alocados pode variar dependendo do tamanho da palavra da máquina e do compilador
 - **sizeof(T)** retorna o número de bytes usado pelo tipo T

Tipo C	32-bit	64-bit
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int (*)	8	8
char *	4	8

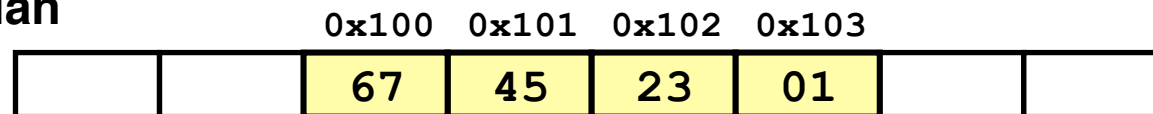
Ordenação de Bytes

- Big Endian (computadores Sun, Mac – não Intel)
 - byte **mais significativo** com endereço mais baixo
- Little Endian (Intel)
 - byte **mais significativo** com endereço mais alto
- Exemplo
 - variável y do tipo inteiro tem valor $0x01234567$ (hexa)
 - Endereço $\&y$ é $0x100$

Big Endian



Little Endian



Ordenação de Bytes

- Diz respeito ao **armazenamento** na memória
- É em geral transparente para a maioria dos programadores de aplicações
 - programas compilados para qualquer das arquiteturas tem resultado semelhante...

- Relevante quando analisamos o código de máquina
 - Exemplo (litte endian):

```
01 05 64 94 04 08 add $0x8049464 %eax
```

- Importante também na transmissão de dados entre máquinas big e little endian
 - conversão entre a representação interna e o “network standard” e vice-versa

Verificar se a máquina é big ou little endian

- Do “ponto de vista” do programa C, para enxergar se a máquina é big/little endian é necessário **quebrar** o sistema de tipos
 - inteiro → word → sequência de bytes

```
#include <stdio.h>
typedef unsigned char *byte_ptr;

void mostra (byte_ptr inicio, int tam) {
    int i;
    for (i=0;i<tam;i++)
        printf("%.2x", inicio[i]);
    printf("\n");
}

void mostra_int (int num) {
    mostra((byte_ptr) &num, sizeof(int));
}
```

Tipo char

- Em C, o tipo `char` define um valor inteiro representado em um byte
- Podemos utilizar esse valor para armazenar caracteres ou manipulá-lo como um valor inteiro (`int`)

```
char c;  
int val;  
...  
if (c > '0') ...  
...  
val = c - '0';  
...  
val = c - 'a' + 10;  
...
```

Operadores bit a bit em C

- Manipulação de dados bastante comum em sw básico
- Podem ser aplicados a qualquer dos tipos inteiros
 - char, short, int, long
- Baseados na álgebra booleana (bit a bit)

- And

- $A \& B = 1$ quando $A=1$ e $B=1$

&	0	1
0	0	0
1	0	1

- Or

- $A | B = 1$ quando $A=1$ ou $B=1$

	0	1
0	0	1
1	1	1

- Not

- $\sim A = 1$ quando $A=0$

\sim	0	1
0	1	
1	0	

- Or exclusivo (Xor)

- $A \wedge B = 1$ quando $A=1$ ou $B=1$, mas não ambos

\wedge	0	1
0	0	1
1	1	0

Exemplos

```
int a, b, c;
a = 0xFF00;          /* a = 1111 1111 0000 0000 */
b = 0x00FF;          /* b = 0000 0000 1111 1111 */

c = a | b;           /* c = 0xFFFF */
c = a & b;           /* c = 0x0000 */
c = ~a;              /* c = 0x00FF */
c = a^b;             /* c = 0xFFFF */
```

- Swap sem terceira variável:

```
void troca (int *x, int *y) /* 0x00000001 0x01000000 */
{
    *y = *x ^ *y;          /* 0x00000001 0x01000001 */
    *x = *x ^ *y;          /* 0x01000000 0x01000001 */
    *y = *x ^ *y;          /* 0x01000000 0x00000001 */
}
```

Operações lógicas em C

- Operadores lógicos: or (||), and (&&) e not (!)
 - tratam qualquer argumento diferente de zero como **true** e igual a zero como **false**
 - retornam 1 ou 0 (**true, false**)
- Exemplos:

Expressão	Resultado
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>0xaa && 0x55</code>	<code>0x01</code>
<code>0xaa 0x55</code>	<code>0x01</code>

Deslocamento de bits

- Deslocamento de padrões de bits para a direita ou para a esquerda
- **x << n**: shift p/ esquerda (equivale a $* 2^n$)
 - bits menos significativos são preenchidos com 0
- **x >> n**: shift p/ direita (equivale a $/ 2^n$)
 - shift lógico: bits mais significativos preenchidos com 0
 - shift aritmético: bits mais significativos preenchidos com o bit mais significativo original
 - quando o operando à esquerda é sem sinal, o shift é lógico
 - quando o operando é com sinal, tipo de shift é escolha do implementador
 - a aplicação do operador >> a operandos com sinal **não é portátil**

Deslocamento de bits: exemplos

Operação	x = [01100011]	x = [10010101]
x << 4	0 0 1 1 0 0 0 0	0 1 0 1 0 0 0 0
x >> 4 (lógico)	0 0 0 0 0 1 1 0	0 0 0 0 1 0 0 1
x >> 4 (aritmético)	0 0 0 0 0 1 1 0	1 1 1 1 1 0 0 1