

# Aula 1

## Introdução

# Arquitetura de Computadores II

## Recuando 50 anos...

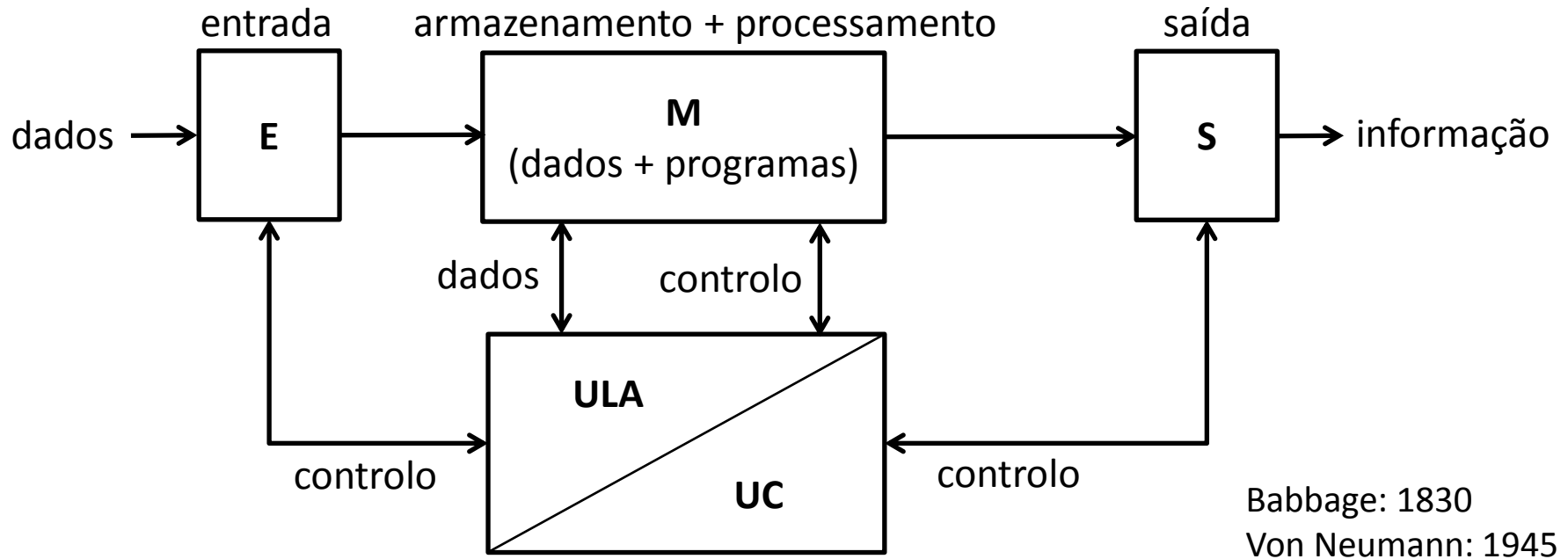
Computar = contar, avaliar, somar → 1º tipo de aplicação – cálculos matemáticos para fins militares

Modelo do computador ? → Homem: resolução de problemas (ex:cálculos matemáticos)

## Resolução de problemas

- 1) aquisição de dados = sentidos → unidade de entrada (E)
- 2) armazenamento (dados/informação) = memória → unidade de memória (M)
- 3) processamento = aplicação de regras e de processos operatórios
  - unidade lógica e aritmética (ULA) : regras
  - unidade de controlo (UC) : sequência das acções
- 4) comunicação de resultados = fala, escrita → unidade de saída (S)

# Arquitetura de Computadores II



Babbage: 1830  
Von Neumann: 1945

E : teclado (interruptores) , rato , scanner, microfone , ...

M: cartões perfurados , sistemas magnéticos , circuitos digitais(Flip-Flops) , condensadores , ...

ULA(Unidade Lógica e Aritmética): rodas dentadas , circuitos digitais (somadores/mux-demux), ...

UC(Unidade de Controlo): rodas dentadas , cartões perfurados , circuitos digitais (tabelas de verdade), ...

S: cartões perfurados , impressoras , ecrans vídeo , ...

# Arquitetura de Computadores II

na prática de que é feito tudo isto?

**lixo + areia da praia + materiais duvidosos...**

**...além de muita água!**



# Arquitetura de Computadores II

Em grande parte os computadores são feitos de materiais plásticos...

- os plásticos são originados a partir de resinas derivadas do petróleo...
- o petróleo resulta da decomposição de restos orgânicos de animais e de vegetais... → **lixo!**

Principal matéria prima dos processadores... **areia da praia!** Quem o diz é a Intel!

From Sand to Silicon: the Making of a Chip

<https://www.youtube.com/watch?v=Q5paWn7bFg4>

**Materiais duvidosos:** borracha, ouro, prata, paládio, cobre, estanho, gálio, índio, boro, rutênio, alumínio, titânio, silício, germânio... podem usar até cerca de 40 elementos da tabela periódica  
→ a maior parte deles venenosos e perigosos para o ambiente!

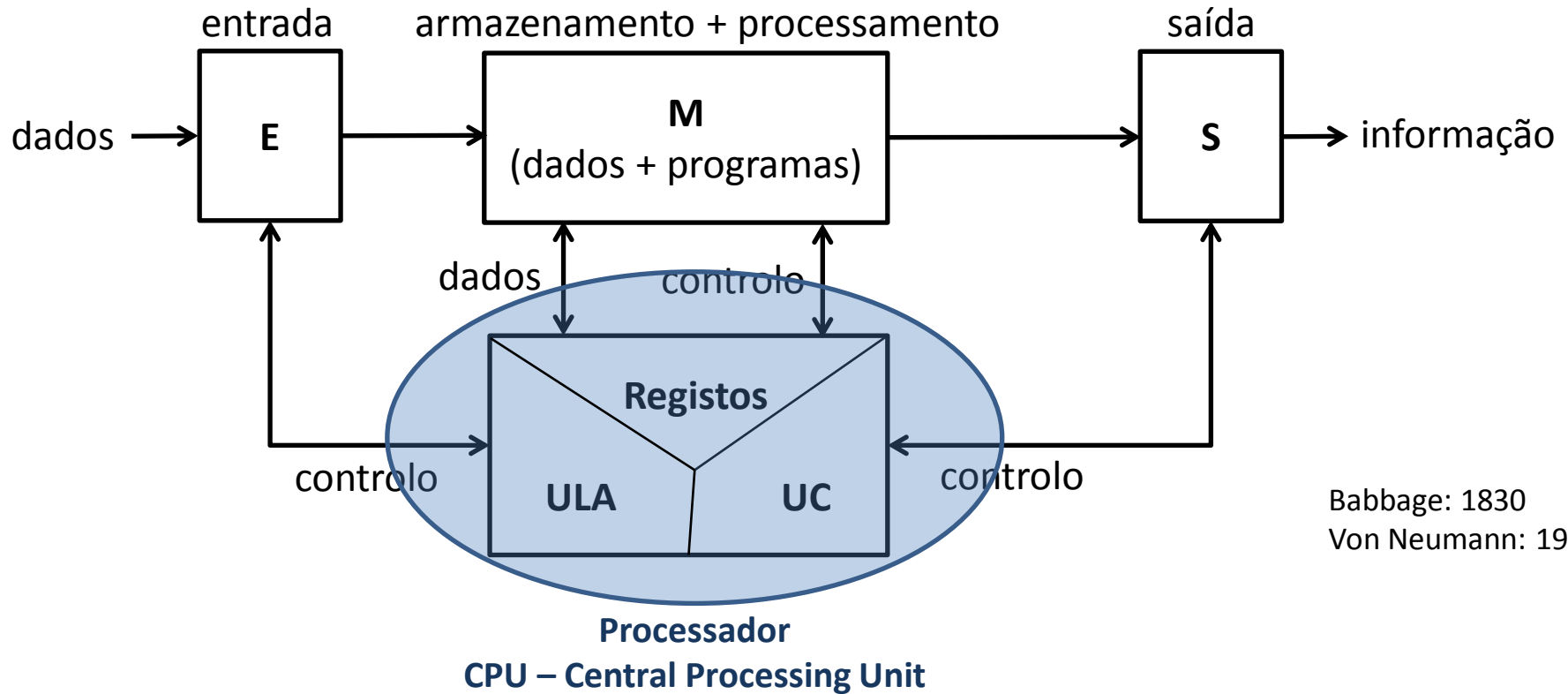
E **água**...muita água! Cerca de 1500Kg por PC...

História dos microprocessadores: [http://www.ritacris.com/ac12/ac12\\_mod4A.html](http://www.ritacris.com/ac12/ac12_mod4A.html)

## Aula 2

CPU – Central Processing Unit

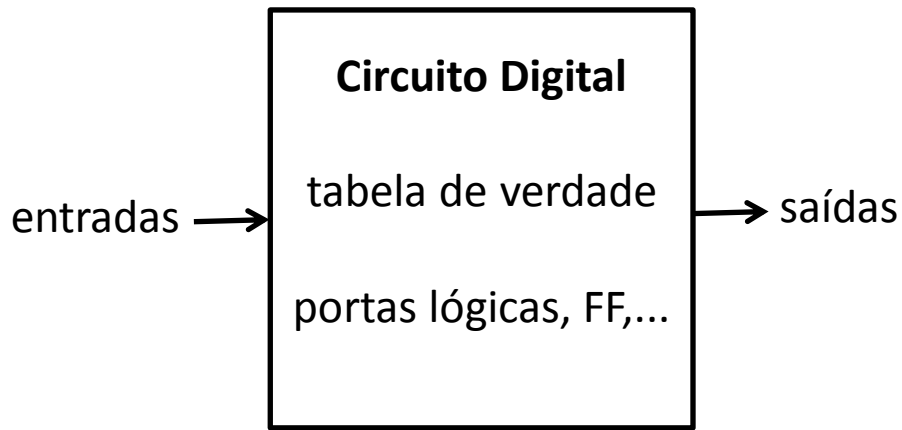
# Modelo do computador digital



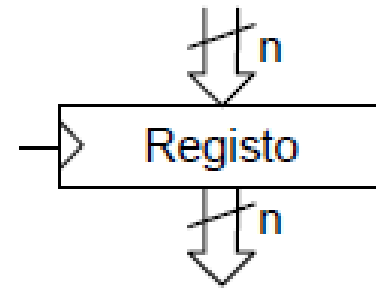
**Processador** : responsável pela execução das instruções e pelo controlo dos restantes dispositivos

- ULA(Unidade Lógica e Aritmética): responsável pelas instruções lógicas (“A>=B ?” , “X=0 ?”) e pelas instruções aritméticas (“+” , “-” , “\*” , “/”)
- UC(Unidade de Controlo): responsável pelo controlo do próprio processador e de outros dispositivos
- Registos – zona de memória para dados, resultados de operações e códigos de funções

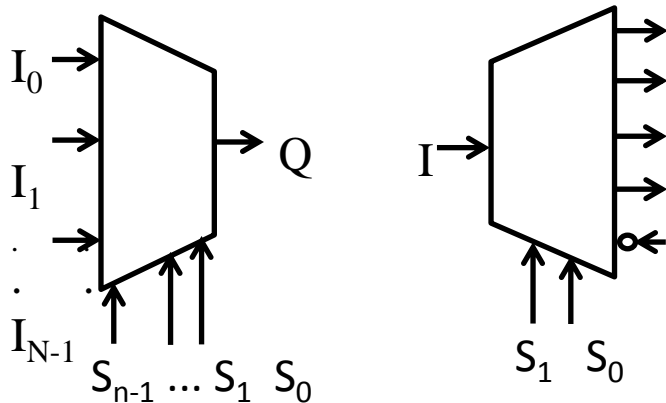
# Processador : componentes básicos



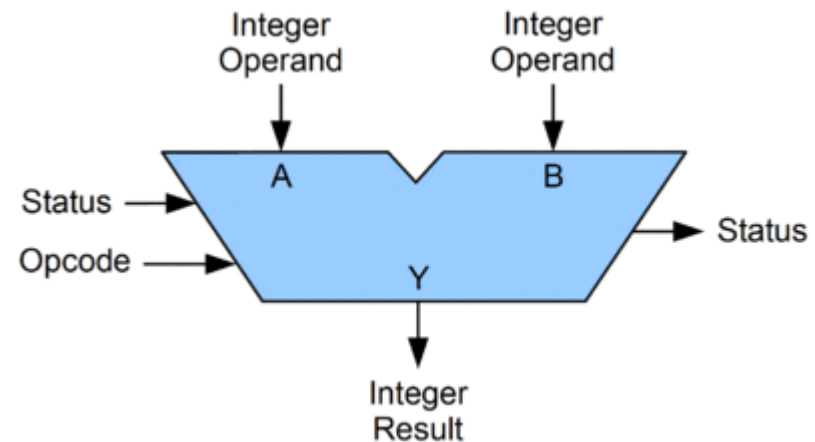
realizam funções lógicas (ex: controlador do LPB)



armazenam conjuntos de bits (ex: 8bit=1byte)



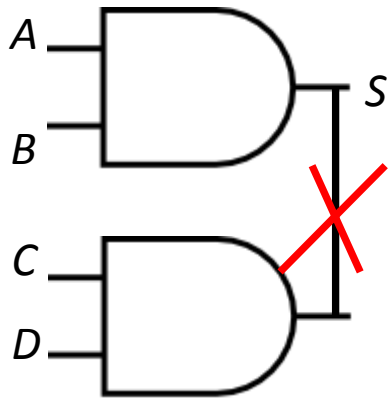
multiplex/demultiplex: encaminham bits (dados)



ALU: realizam operações lógicas/aritméticas



# Z : o terceiro estado digital

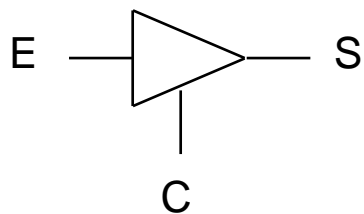


Qual o valor de S para diferentes situações?

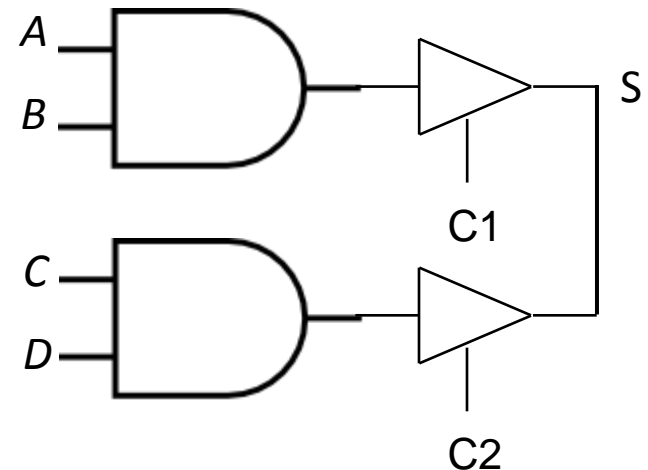
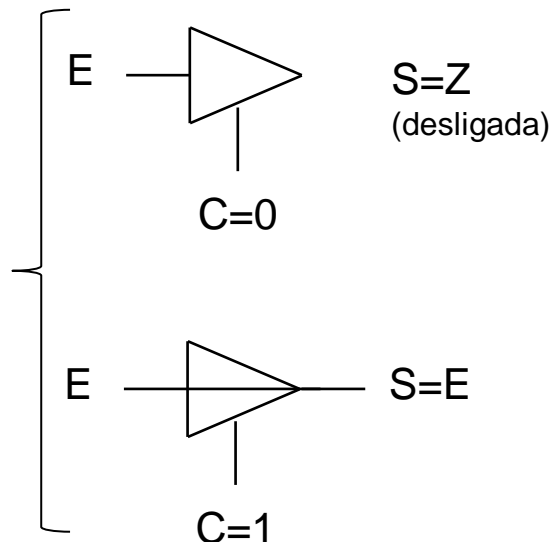
A	B	C	D	S
0	0	0	0	0
1	0	0	1	0
1	1	1	1	1
0	1	1	1	? (conflito)
1	1	0	0	? (conflito)

**Porta tri-state:** cria um terceiro estado, designado por Z (alta-impedância), que não é 0 nem 1 (corresponde ao nada, é como se a saída ficasse desligada)

ex: buffer tri-state

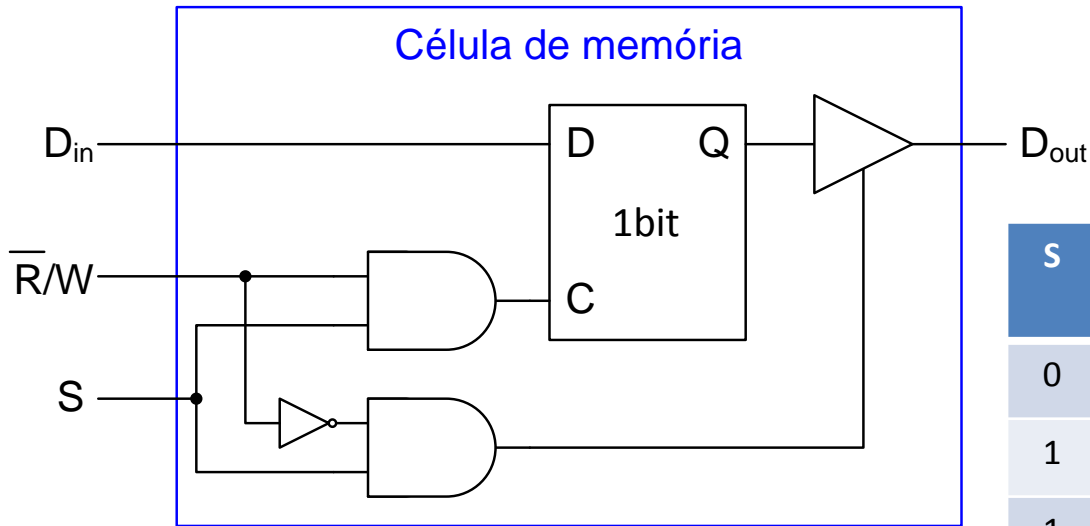


C	E	S
1	0	0
1	1	1
0	-	Z



Desde que C1 e C2 não estejam ambos activos (1) não haverá conflito

# Célula básica de memória de 1 bit



S	$\sim R/W$	Din	Dout	Estado
0	-	-	Z	Inativo
1	0	-	Conteúdo do FF	Read
1	1	0 / 1	Z	Write FF captura Din

$D_{in}$  – bit de entrada

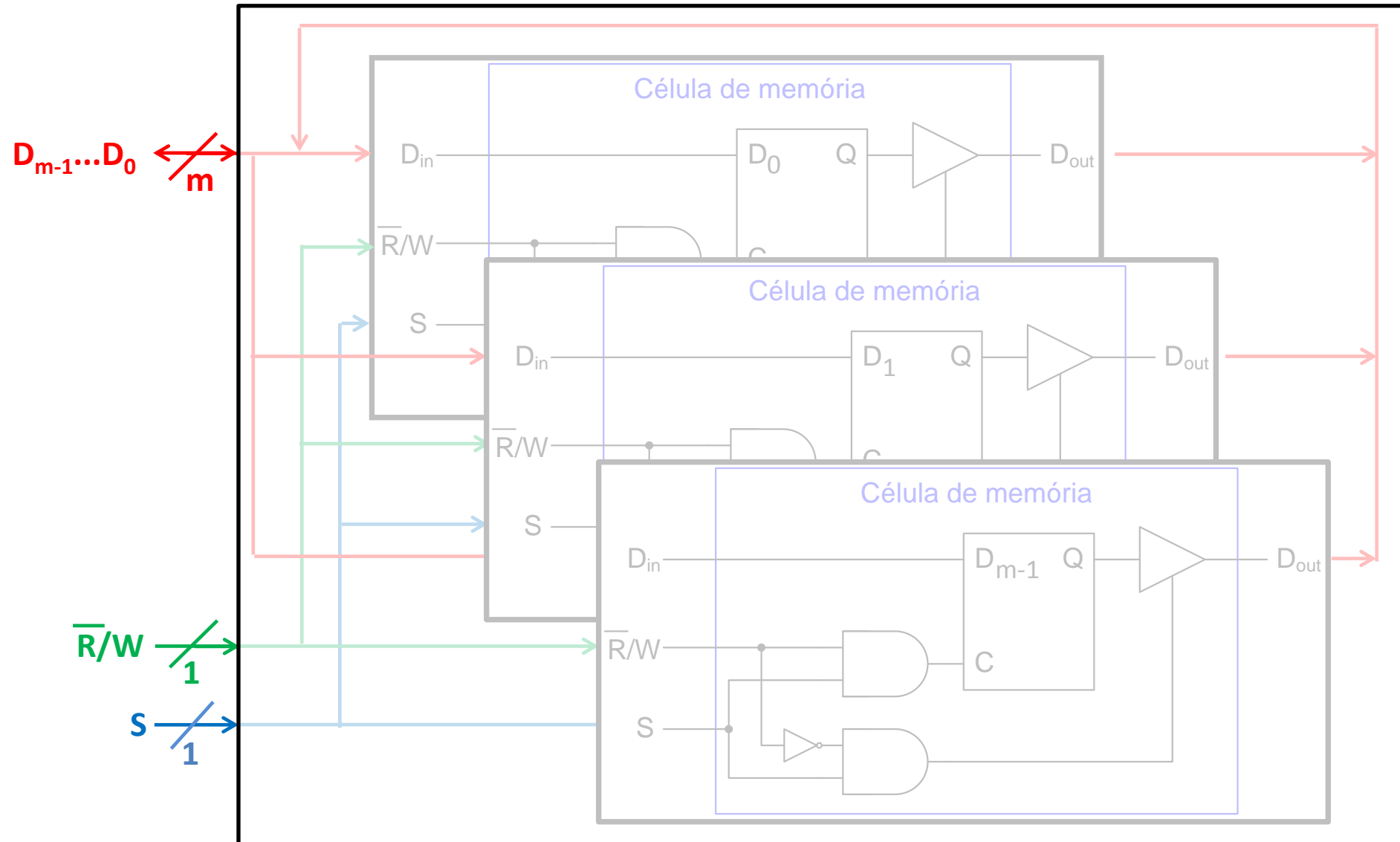
$\overline{R/W}$  – sinal de Read/Write (0=Read , 1=Write)

S – Select(Strobe) : habilita/desabilita a acção de read/write (0=desabilita , 1=habilita)

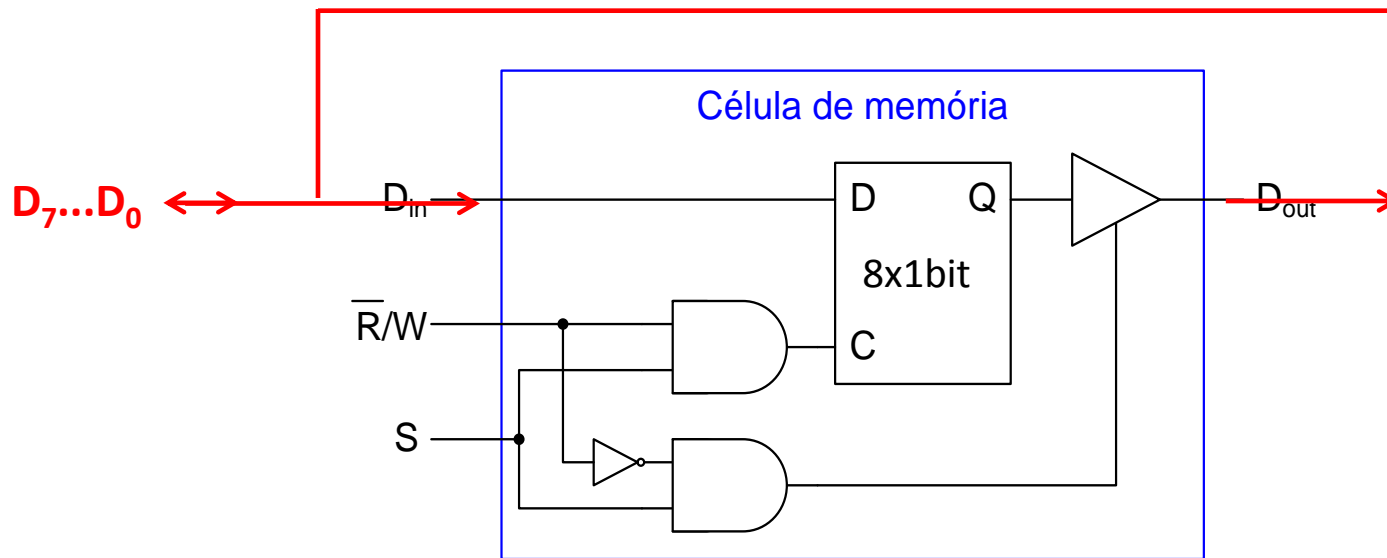
$D_{out}$  – bit de saída

# Célula básica de memória : registo de $m$ bit

Exemplo:  $m = 8$  bit (1 byte)  $\rightarrow$  registo constituído por 8 células de memória de 1 bit



# Célula básica de memória de 8 bit (1 byte)



$D_{7...0}$  – bits de entrada/saída [ $D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0$ ] ( $m=8$ )

$\overline{R/W}$  – sinal de Read/Write (0=Read , 1=Write) – só uma acção é possível em cada instante

$S$  – Select(Strobe) : habilita/desabilita a acção de read/write (0=desabilita , 1=habilita)

# Memória : p registos de m bits

## **dados (Data Bus)**

dados a serem lidos  
ou escritos

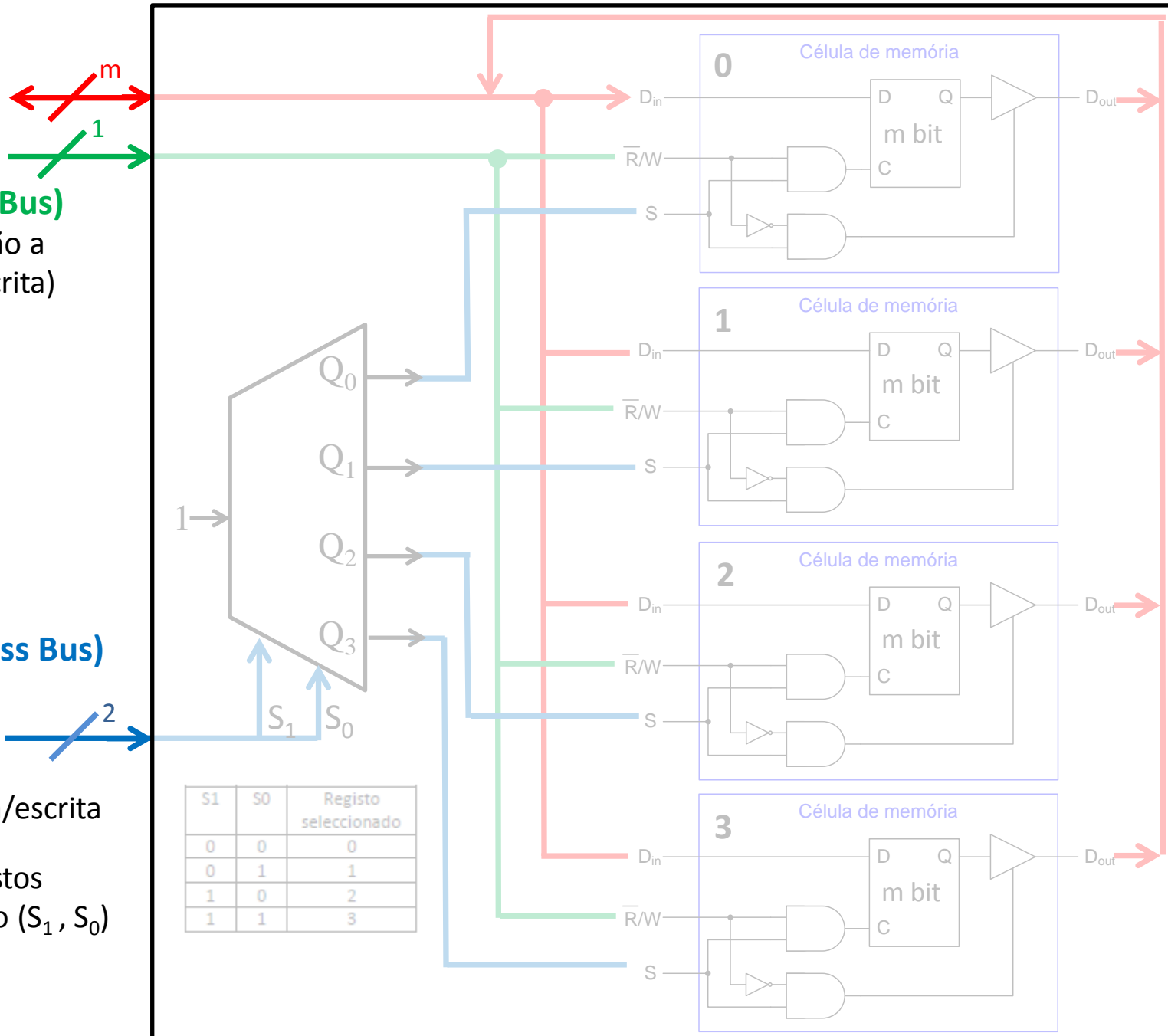
## **controlo (Control Bus)**

determina a operação a  
executar (leitura/escrita)

## **endereços (Address Bus)**

permite seleccionar  
o registo sobre o  
qual são feitas as  
operações de leitura/escrita

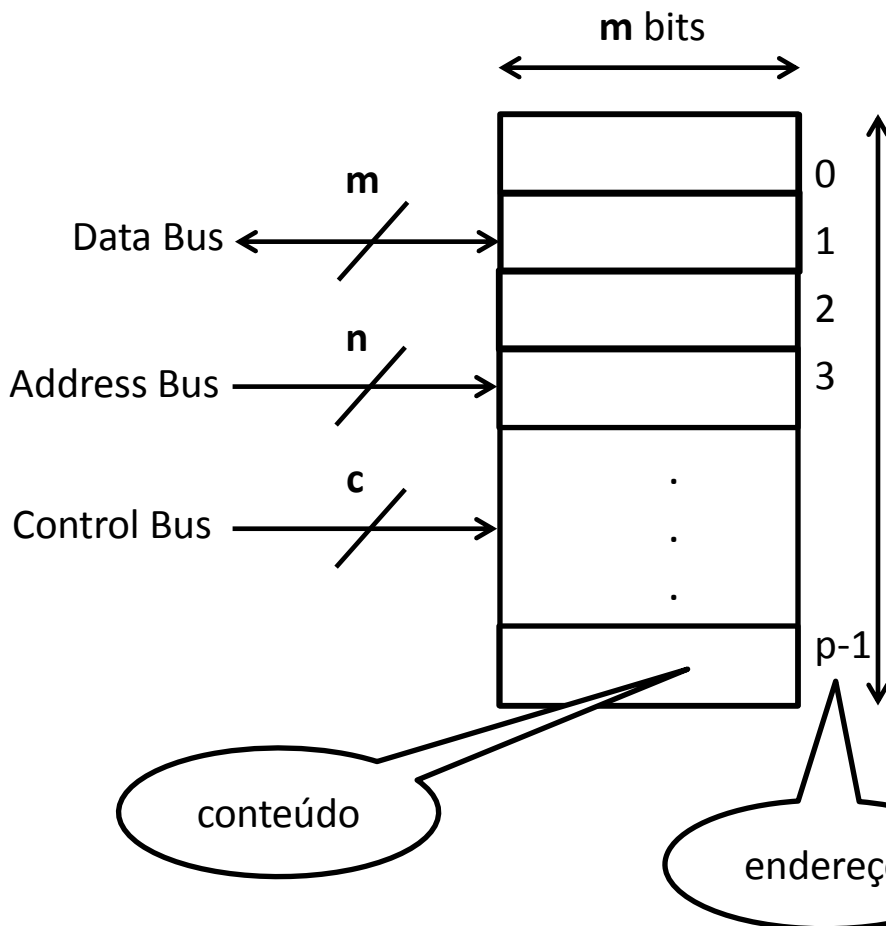
Exemplo:  $p = 4$  registos  
dois bits de selecção ( $S_1, S_0$ )



# Memória : p registos de m bits

## Organização e capacidade da memória (RAM – Random Access Memory)

- palavra  $\rightarrow m$  bits (ex.  $m=8$ , 1 byte)
- capacidade  $\rightarrow p = 2^n$  registos, endereços ou palavras  $\rightarrow n$  bits de selecção
- operações  $\rightarrow$  read(leitura) , write(escrita) , chip select(selecção)



$p$  registos :  $n$  bits de endereço  $\rightarrow p=2^n$

ex:  $n=3 \rightarrow p=2^3 = 8$  byte

$n=4 \rightarrow p=2^4 = 16$  byte

$n=8 \rightarrow p=2^8 = 256$  byte

$n=10 \rightarrow p=2^{10} = 1024=1\text{Kbyte}$

$n=20 \rightarrow p=2^{20} = 1024*1024=1\text{Mbyte}$

ex:  $n=3 \rightarrow p=2^3 = 8$  byte

	000
	001
	010
	011
	100
	101
	110
	111

# Estrutura dos Processadores (CPU)

**Desafio:** construir um dispositivo (processador) capaz de executar uma tarefa descrita por um conjunto de instruções (programa)

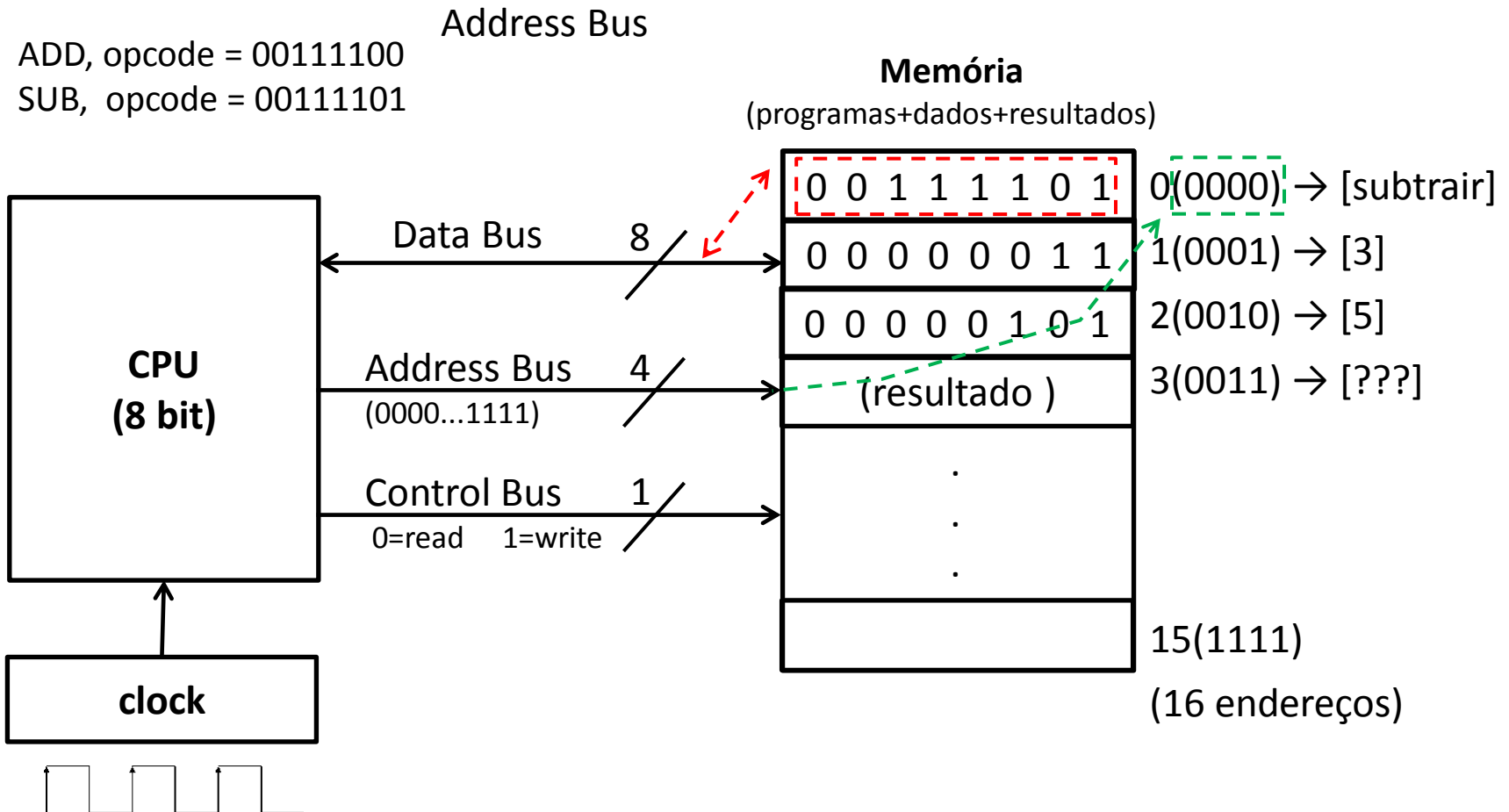
- 1) As tarefas a realizar limitam-se a operações aritméticas sobre valores inteiros, representados em código de complemento para 2 (valores positivos e negativos);
- 2) As instruções, os dados do problema e respetivos resultados são constituídos por conjuntos de n bits (ex:  $n = 8$  bits – dizemos que o processador é de 8 bits; o primeiro processador da Intel, o 4004, era de 4 bits... actualmente vamos nos 64 bits!)
- 2.1) codificação das instruções: opcode(operation code)  
ex: ADD (soma) → opcode = 00111100  
SUB(subtração) → opcode = 00111101
- 3) As instruções e os dados do problema estão armazenados na memória e os resultados da operação também deverão ficar armazenados na memória;
- 4) As instruções deverão ser executadas sequencialmente pela ordem em que constam na memória, ao ritmo constante de um sinal de controlo (clock);

[https://pt.wikipedia.org/wiki/Lista\\_de\\_microprocessadores\\_da\\_Intel](https://pt.wikipedia.org/wiki/Lista_de_microprocessadores_da_Intel)



# Estrutura dos Processadores (CPU)

Exemplo de tarefa: subtrair o valor 5 do valor 3 e colocar o resultado na memória  
 $\text{subtrair}(3,5) = 3 - 5 = -2$

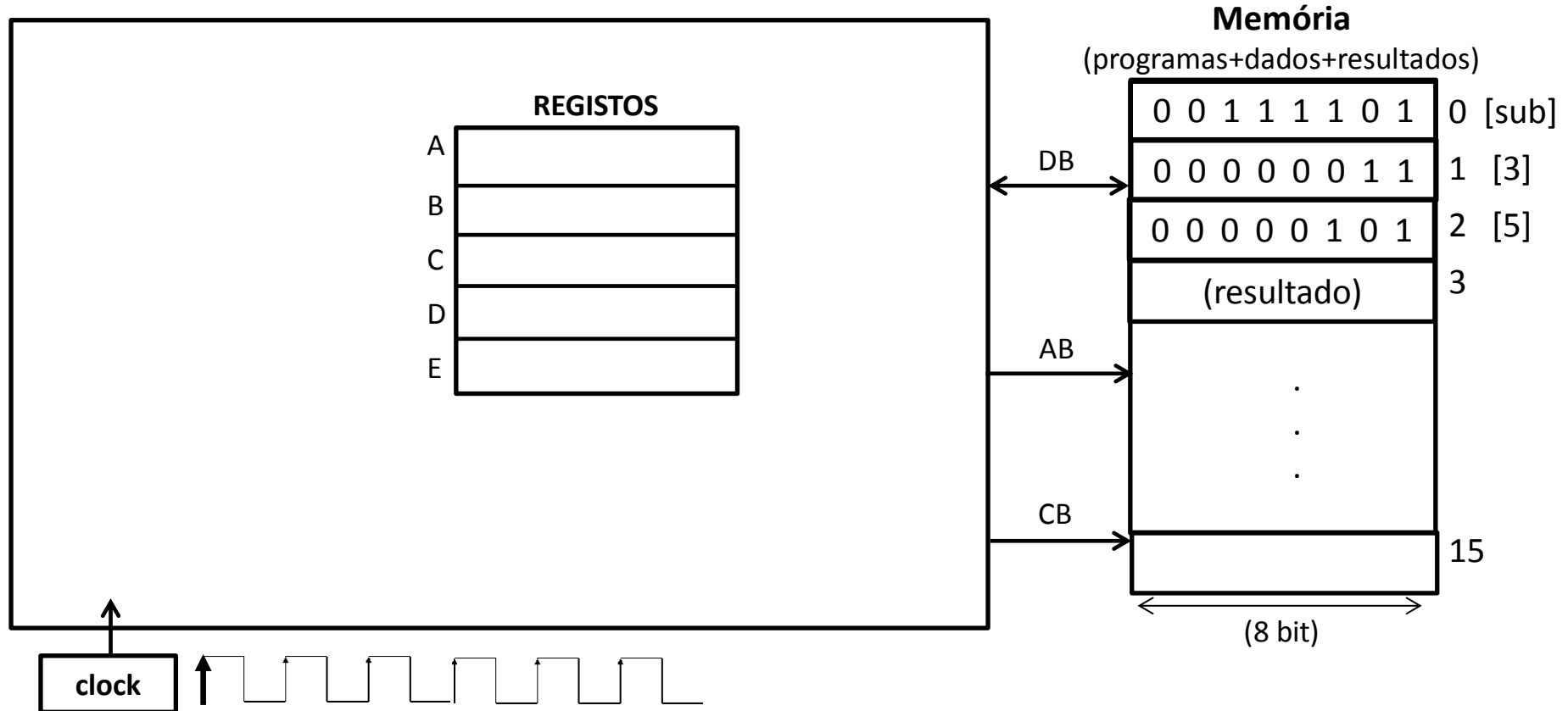




# Aula 3

## Execução das operações

# Estrutura dos Processadores (CPU)

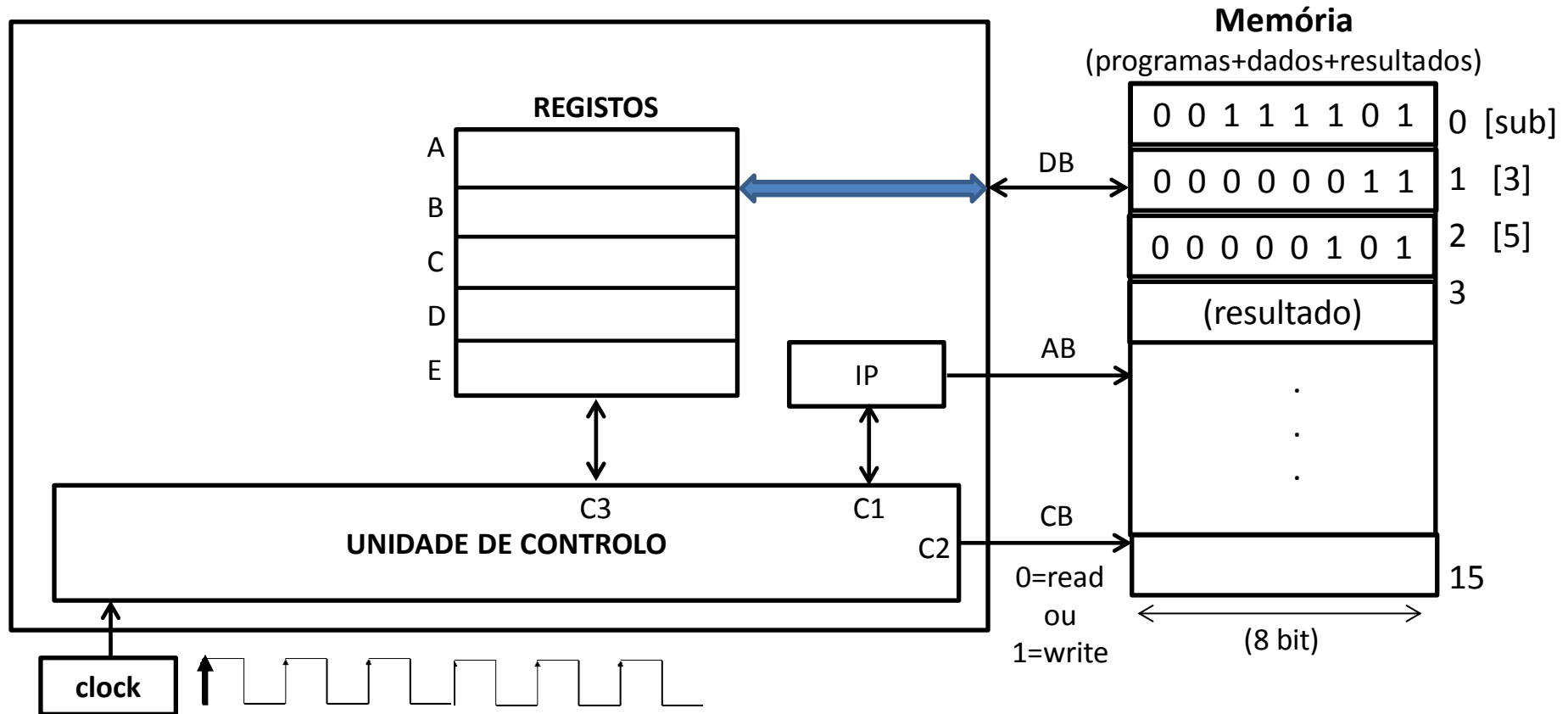


1º) Como indicar ao CPU qual a operação a executar e os correspondentes dados?

→ colocando os respectivos códigos em locais apropriados dentro do CPU

→ o conteúdo desses locais muda frequentemente, logo deverão ser registos; os registos têm nomes e destinam-se a guardar dados e/ou resultados bem como códigos de operações

# Estrutura dos Processadores (CPU)

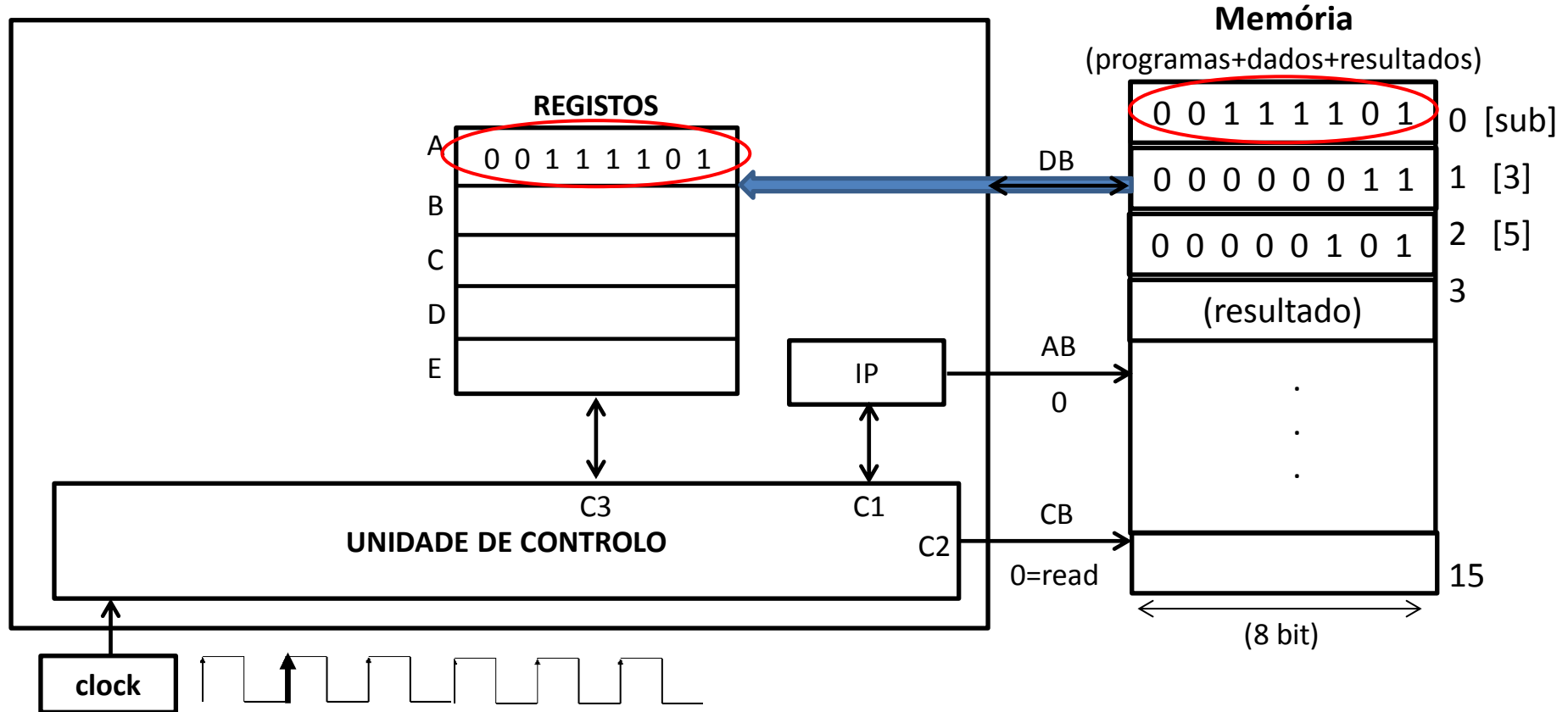


2º) Como pode o CPU aceder às várias células de memória?

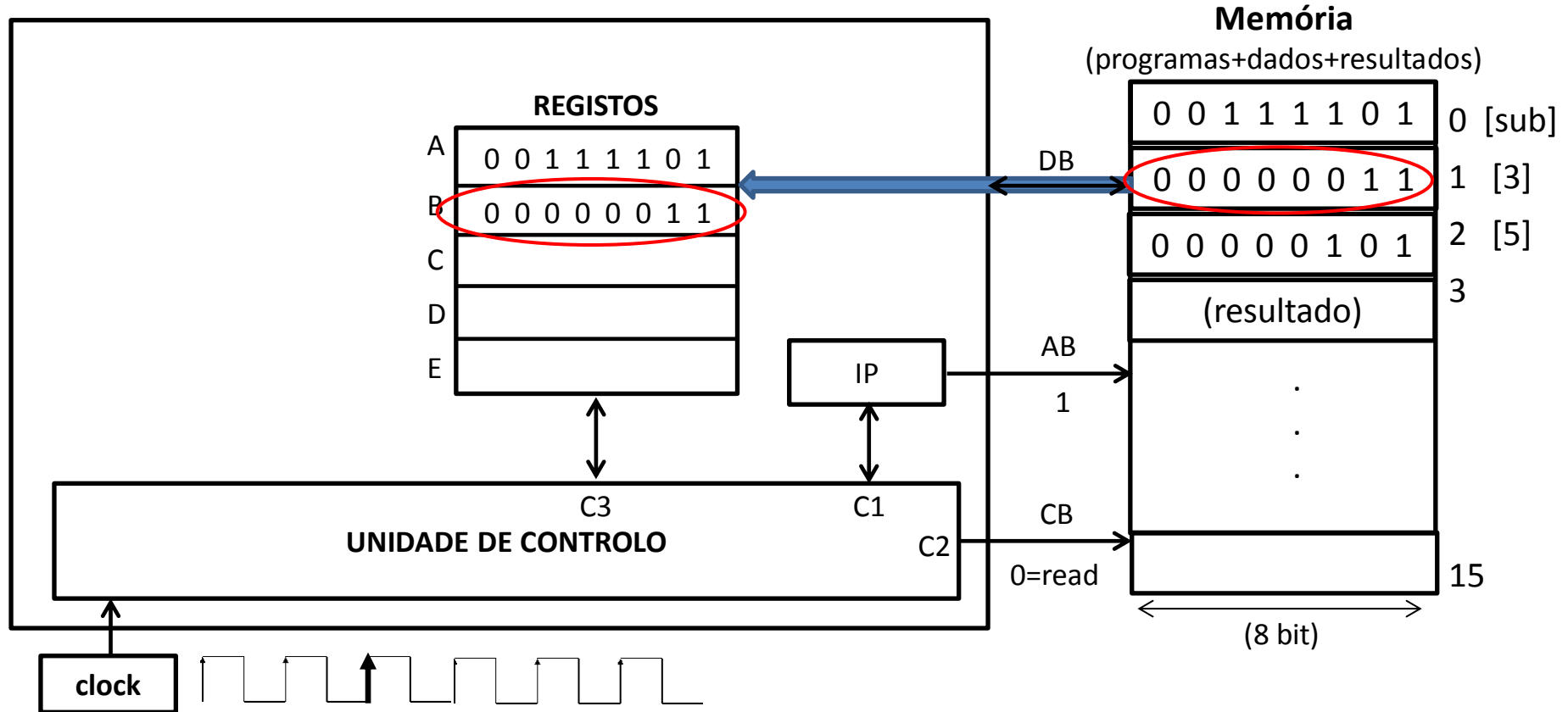
- CPU indica qual a posição ou endereço a que pretende aceder → através do AB(Address Bus)  
existe um elemento (IP-Instruction Pointer) que aponta para os endereços e que pode avançar ou recuar consoante o programa vai evoluindo → poderá ser formado por um contador
- CPU indica qual a operação que pretende executar, leitura ou escrita → através do CB(Control Bus)
- CPU recebe ou envia o conteúdo das diversas posições de memória ↔ através do DB(Data Bus)

Esta sequência é controlada pela UC(Unidade de Controlo)

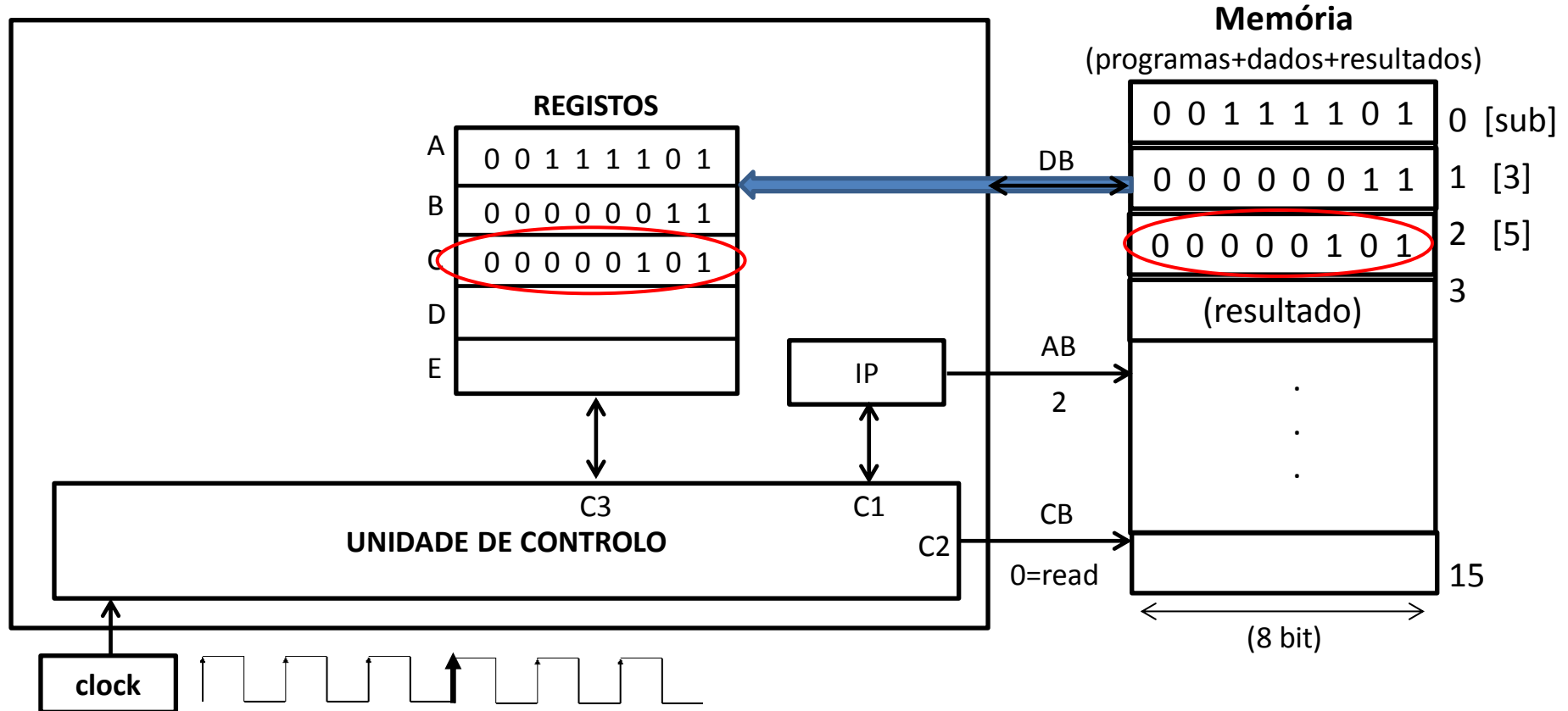
# Estrutura dos Processadores (CPU)



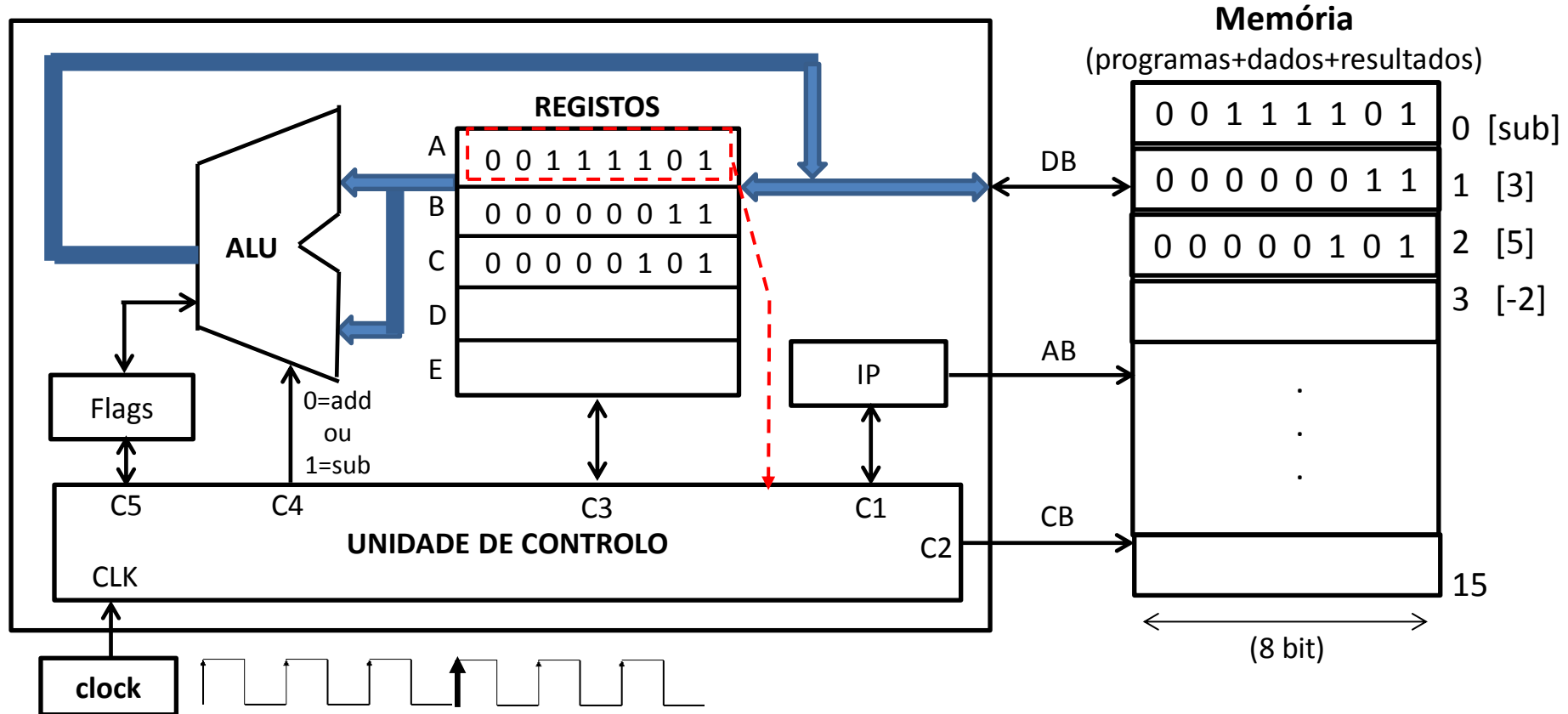
# Estrutura dos Processadores (CPU)



# Estrutura dos Processadores (CPU)



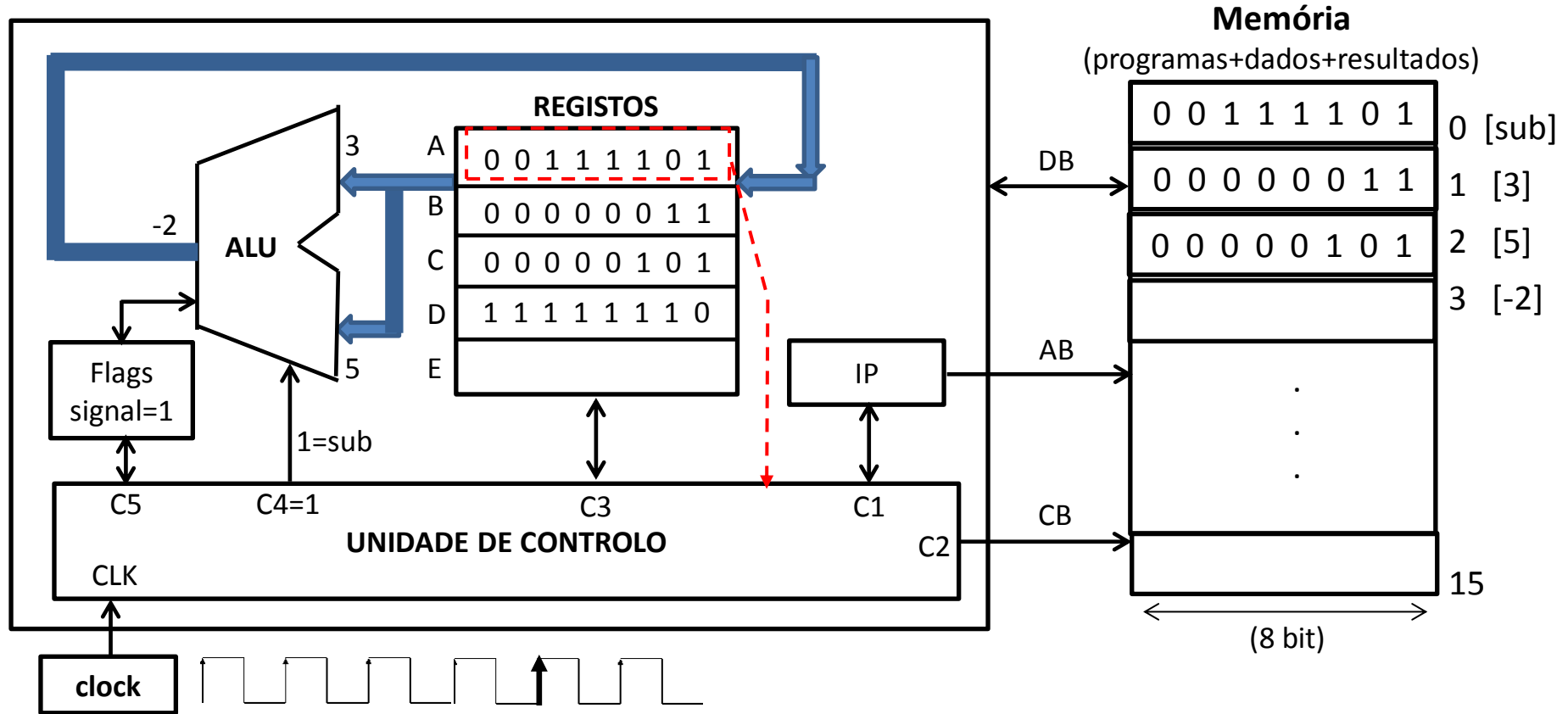
# Estrutura dos Processadores (CPU)



3º) Como pode o CPU executar a operação?

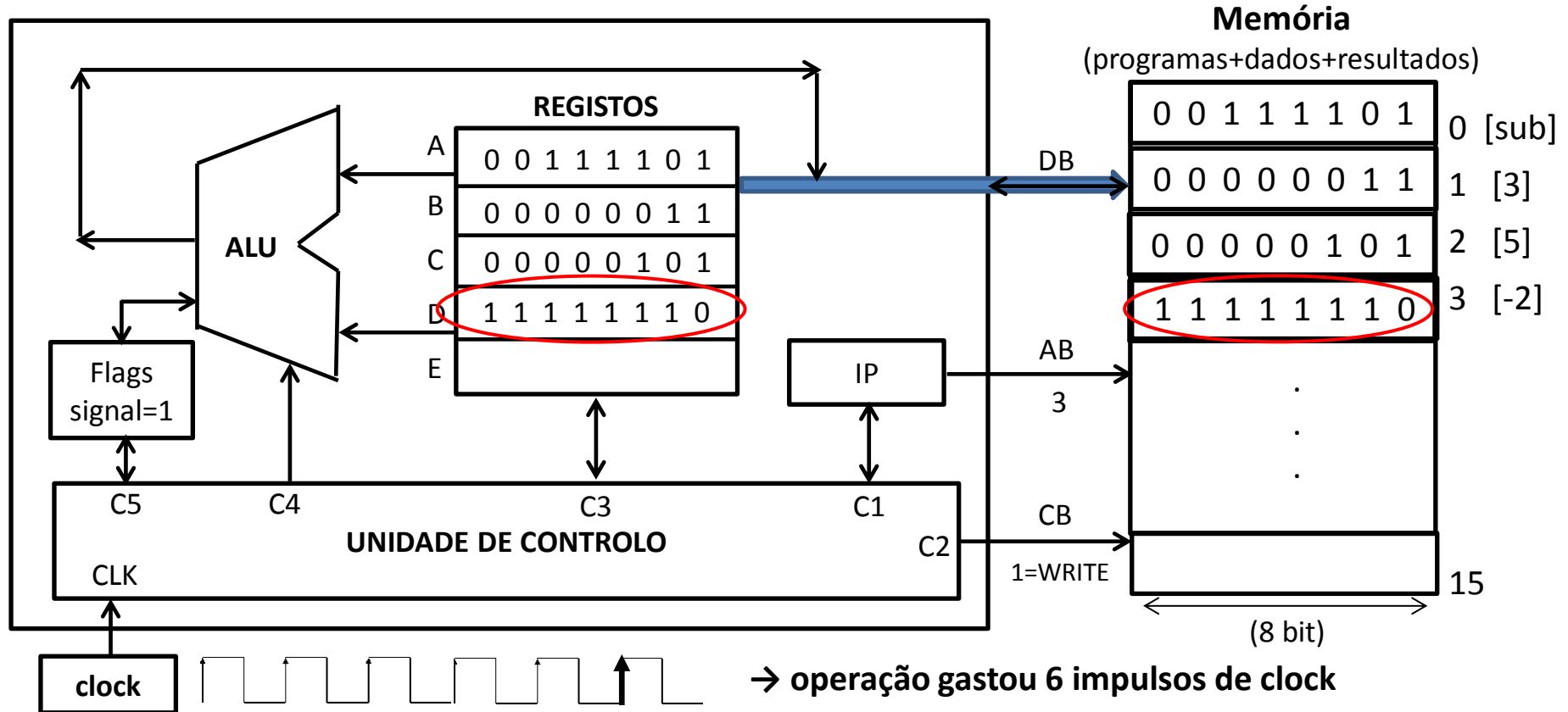
- dispondo de uma ALU a qual realiza operações aritméticas e lógicas – esta é controlada pela UC em função do código da instrução a executar o “opcode” (o qual neste exemplo está contido no registo A)
- a UC decodifica o *opcode* e em resultado disso gera os sinais de controlo apropriados
- certas situações devem ser registadas para controlarem as próximas acções do CPU - por exemplo se houver uma divisão por zero o processamento poderá parar e ser indicado um erro, ou, como neste exemplo, assinalar que o resultado é negativo → registo de Flags(bandeiras)

# Estrutura dos Processadores (CPU)

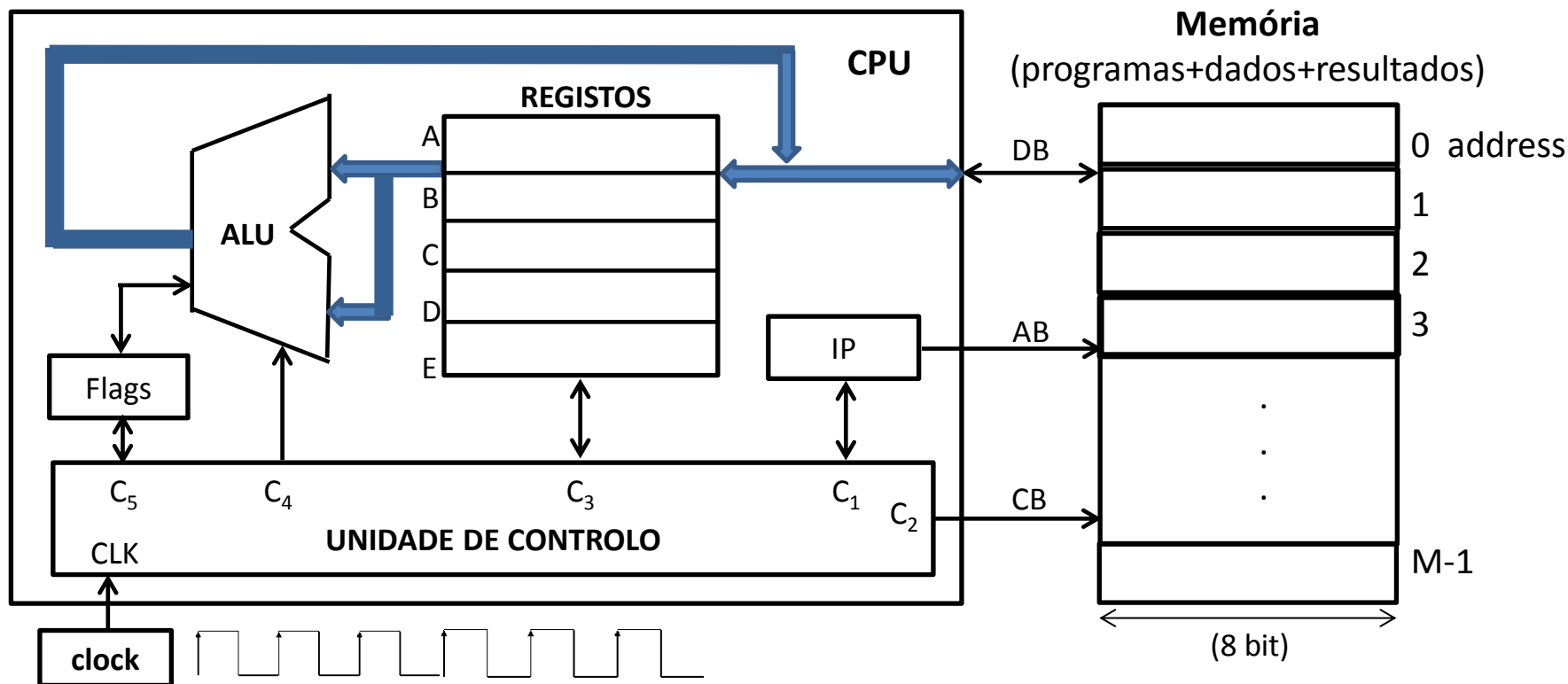




# Estrutura dos Processadores (CPU)



# Estrutura dos Processadores (CPU)



**Memória:** contém os dados dos problemas e as instruções para operar sobre esses dados (ie, os programas)

- constituída por M registros de n bits (ex: n=8), cada um com o seu endereço (address)

**CPU:** - executa as instruções a partir da memória

- REGISTOS: armazenam os dados/resultados das operações, bem como os códigos das operações a executar

- IP (Instruction Pointer): aponta na memória qual o endereço da próxima instrução a executar

- Flags: sinalizam determinadas ocorrências (ex: operação com resultado negativo)

- ALU: executa as operações aritméticas e lógicas

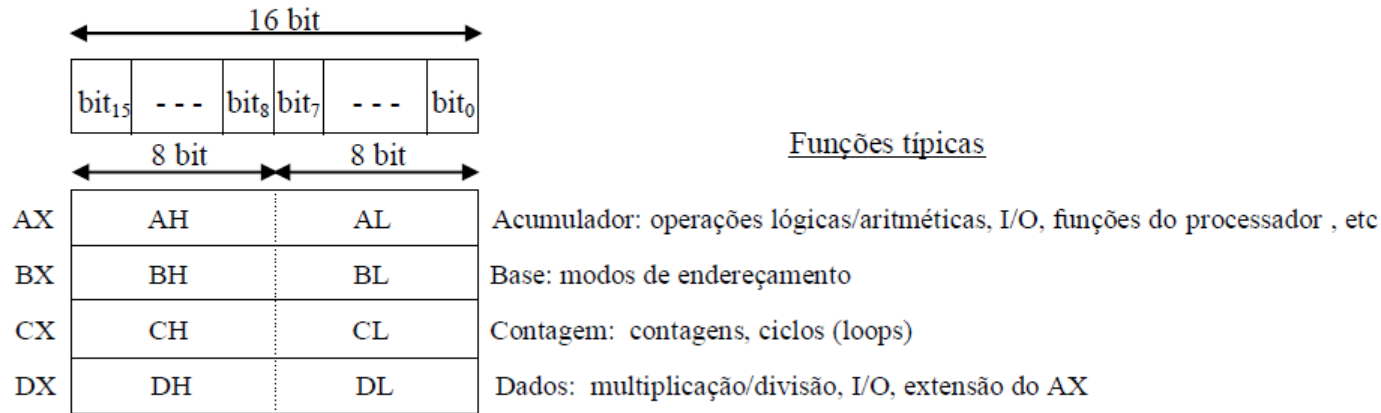
- UC (Unidade de Controle): coordena o funcionamento dos restantes blocos, gerando sinais de controle  $C_n$

- Clock: determina a cadência a que as instruções são executadas

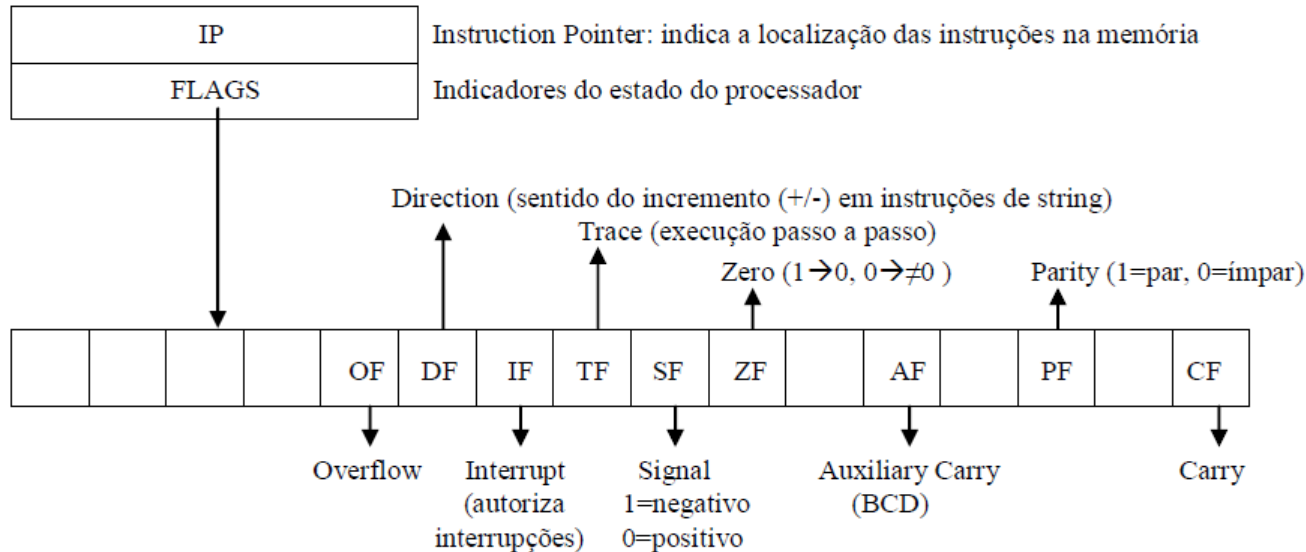
# Registos do processador 8086 (x86-16 bits)

Agrupam-se em: 1)dados 2)endereços 3)segmento 4)controlo

1)DADOS (uso geral e funções do processador)      X=16bit      H=High(8bit)      L=Low(8bit)



## 4)CONTROLO



# Conjunto de registros x86 de 64-bit

Register encoding	Not modified for 8-bit operands			Low 8-bit	16-bit	32-bit	64-bit
	Not modified for 16-bit operands						
	Zero-extended for 32-bit operands						
0			AH†	AL	AX	EAX	RAX
3			BH†	BL	BX	EBX	RBX
1			CH†	CL	CX	ECX	RCX
2			DH†	DL	DX	EDX	RDX
6				SIL‡	SI	ESI	RSI
7				DIL‡	DI	EDI	RDI
5				BPL‡	BP	EBP	RBP
4				SPL‡	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15
	63	32 31	16 15	8 7 0			
	† Not legal with REX prefix			‡ Requires REX prefix			

# Processo de programação

**Máquina** : apenas trabalha com bits (0 e 1) → tudo tem de ser introduzido na máquina nesta forma, incluindo dados e programas(conjuntos de instruções)

➤ Programa em código binário → programa em linguagem máquina ou em baixo nível

**Homem**: é difícil trabalhar com extensas listas de bits, é mais fácil trabalhar com linguagens naturais (ex:Inglês)

➤ Programa em linguagem natural (ex: C, Java, HTML) → programa em alto nível

**Processo de tradução**: passagem de um programa de uma linguagem de alto nível para uma de baixo nível

ex. `printf("Hello World")` → 10010001001001000100001111100101010...

**Compilador**: programa que traduz o código fonte de um programa escrito numa linguagem de alto nível para o código correspondente numa linguagem de programação de baixo nível (por exemplo, Assembly ou código máquina) – todas as instruções do programa são primeiro convertidas e depois o programa é executado como um todo;

**Interpretador**: a diferença para o compilador é que a tradução do programa fonte é feita instrução a instrução, sendo cada uma delas executada de imediato;

# Assembly

**Assembly (linguagem de montagem):** havendo necessidade de programar directamente em código máquina, isso obrigaria a utilizar códigos de 0 e 1's, difíceis de manipular e memorizar.

O *Assembly* simplifica este processo ao atribuir nomes(mnemónicas) a esses conjuntos de bits, nomes esses que permitem usar as instruções nativas do processador sem usar o respectivo código binário.

Em seguida é usado um programa de montagem, designado por Assembler, que irá converter cada mnemónica para o correspondente código binário.

instrução em Assembly = <opcode> , <operand>

Ex: opcode(mnemónica)	operand	binário	ação
mov al	5	10110000 00000101	mover para AL o valor 5
add ax	539Fh	00000101 0101 0011 1001 1111	somar o registo AX o valor 539Fh

**NOTA:** a designação da linguagem é *Assembly* e não Assembler (é errado dizer “programar em Assembler”). Assembler é um programa que efectua a tradução das mnemónicas do *Assembly* para código máquina e não uma linguagem.

# Assembly

## Razões para utilizar Assembly

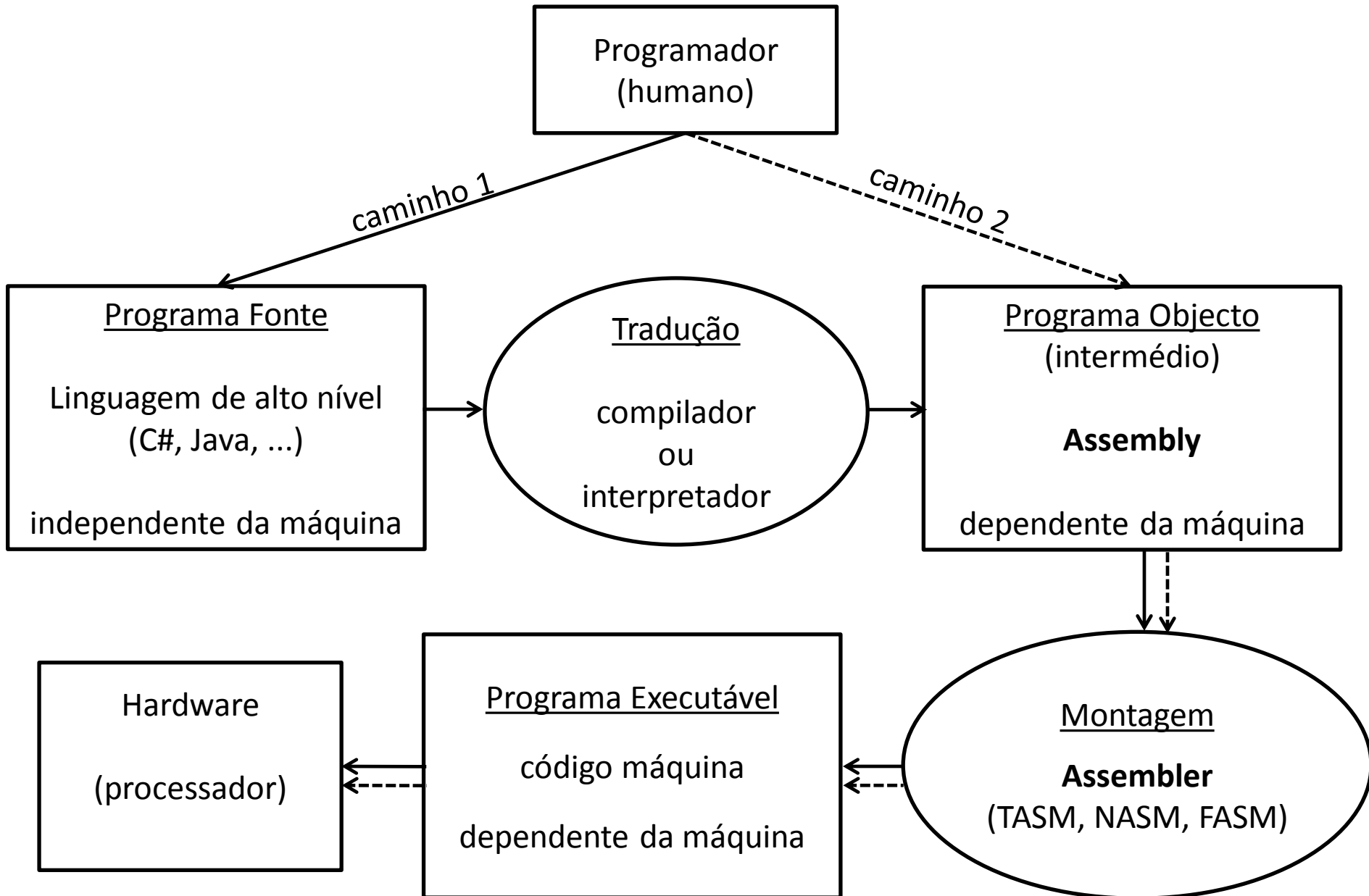
**Rapidez:** o código em Assembly pode ser otimizado, resultando em programas mais pequenos, logo mais rápidos. Tomando como exemplo a programação de jogos, estes têm de responder muito rapidamente às ações do utilizador.

**Memória:** os tradutores automáticos (compiladores, tradutores) geram por vezes código supérfluo o qual pode ocupar memória desnecessária. Programando directamente em Assembly pode reduzir-se a ocupação da memória.

**Eficiência:** um programa em Assembly faz uso directo das características do processador a que se destina, o que obriga a um conhecimento aprofundado dessas características para tirar partido da linguagem. Ao usar o Assembly, o programador adquire conhecimentos que lhe permitem inclusivamente escrever código eficiente mesmo ao usar linguagens de alto-nível.

**Controlo:** a maioria dos compiladores/interpretadores bloqueia ou dificulta o acesso a certas componentes do hardware (por motivos de segurança) as quais podem ser ultrapassadas pelo uso do Assembly.

# Processo de programação



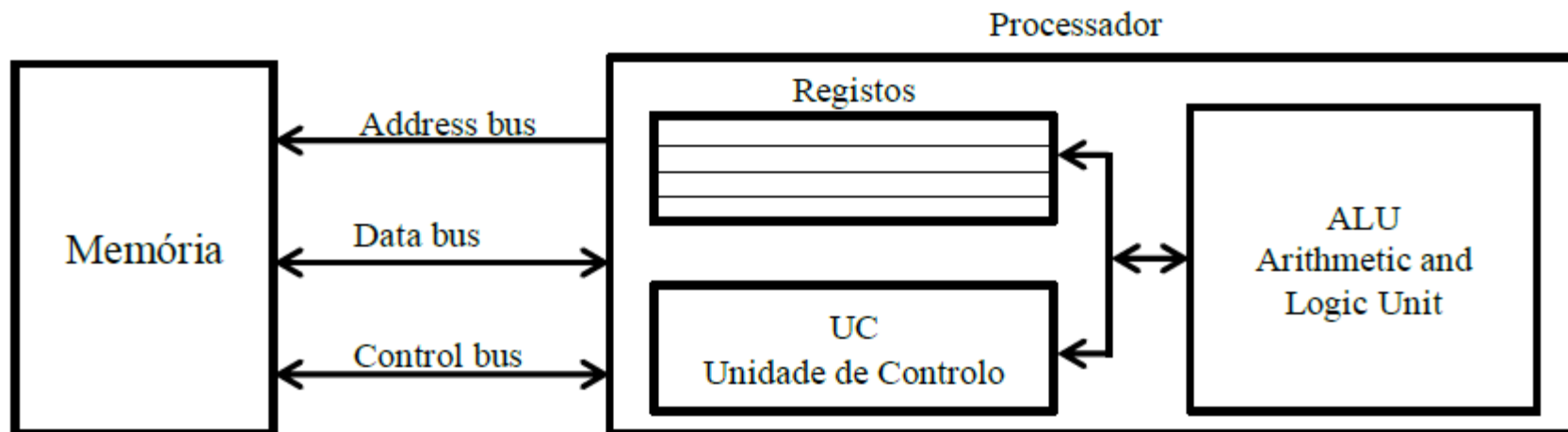


# Tabela ASCII - American Standard Code for Information Interchange

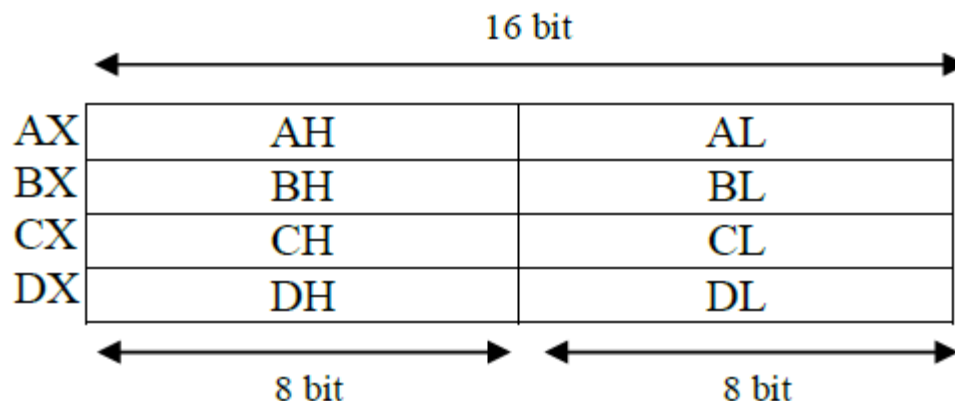
Relaciona os caracteres com a sua representação numérica (decimal, hexadecimal, binária)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	<b>Space</b>	64	40	100	&#64;	<b>@</b>	96	60	140	&#96;	<b>`</b>
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	<b>!</b>	65	41	101	&#65;	<b>A</b>	97	61	141	&#97;	<b>a</b>
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	<b>"</b>	66	42	102	&#66;	<b>B</b>	98	62	142	&#98;	<b>b</b>
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	<b>#</b>	67	43	103	&#67;	<b>C</b>	99	63	143	&#99;	<b>c</b>
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	<b>\$</b>	68	44	104	&#68;	<b>D</b>	100	64	144	&#100;	<b>d</b>
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	<b>%</b>	69	45	105	&#69;	<b>E</b>	101	65	145	&#101;	<b>e</b>
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	<b>&amp;</b>	70	46	106	&#70;	<b>F</b>	102	66	146	&#102;	<b>f</b>
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	<b>'</b>	71	47	107	&#71;	<b>G</b>	103	67	147	&#103;	<b>g</b>
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	<b>(</b>	72	48	110	&#72;	<b>H</b>	104	68	150	&#104;	<b>h</b>
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	<b>)</b>	73	49	111	&#73;	<b>I</b>	105	69	151	&#105;	<b>i</b>
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	<b>*</b>	74	4A	112	&#74;	<b>J</b>	106	6A	152	&#106;	<b>j</b>
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	<b>+</b>	75	4B	113	&#75;	<b>K</b>	107	6B	153	&#107;	<b>k</b>
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	<b>,</b>	76	4C	114	&#76;	<b>L</b>	108	6C	154	&#108;	<b>l</b>
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	<b>-</b>	77	4D	115	&#77;	<b>M</b>	109	6D	155	&#109;	<b>m</b>
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	<b>.</b>	78	4E	116	&#78;	<b>N</b>	110	6E	156	&#110;	<b>n</b>
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	<b>/</b>	79	4F	117	&#79;	<b>O</b>	111	6F	157	&#111;	<b>o</b>
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	<b>0</b>	80	50	120	&#80;	<b>P</b>	112	70	160	&#112;	<b>p</b>
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	<b>1</b>	81	51	121	&#81;	<b>Q</b>	113	71	161	&#113;	<b>q</b>
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	<b>2</b>	82	52	122	&#82;	<b>R</b>	114	72	162	&#114;	<b>r</b>
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	<b>3</b>	83	53	123	&#83;	<b>S</b>	115	73	163	&#115;	<b>s</b>
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	<b>4</b>	84	54	124	&#84;	<b>T</b>	116	74	164	&#116;	<b>t</b>
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	<b>5</b>	85	55	125	&#85;	<b>U</b>	117	75	165	&#117;	<b>u</b>
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	<b>6</b>	86	56	126	&#86;	<b>V</b>	118	76	166	&#118;	<b>v</b>
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	<b>7</b>	87	57	127	&#87;	<b>W</b>	119	77	167	&#119;	<b>w</b>
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	<b>8</b>	88	58	130	&#88;	<b>X</b>	120	78	170	&#120;	<b>x</b>
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	<b>9</b>	89	59	131	&#89;	<b>Y</b>	121	79	171	&#121;	<b>y</b>
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	<b>:</b>	90	5A	132	&#90;	<b>Z</b>	122	7A	172	&#122;	<b>z</b>
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	<b>;</b>	91	5B	133	&#91;	<b>[</b>	123	7B	173	&#123;	<b>{</b>
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<b>&lt;</b>	92	5C	134	&#92;	<b>\</b>	124	7C	174	&#124;	<b> </b>
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	<b>=</b>	93	5D	135	&#93;	<b>]</b>	125	7D	175	&#125;	<b>}</b>
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	<b>&gt;</b>	94	5E	136	&#94;	<b>^</b>	126	7E	176	&#126;	<b>~</b>
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	<b>?</b>	95	5F	137	&#95;	<b>_</b>	127	7F	177	&#127;	<b>DEL</b>

## Processador básico



Registos principais do CPU (família Intel x86):  $x = 16$



# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

**mov ah, 40h ;escrita**

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

**mov bx, 1 ;ecran**

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	CH	CL
DX	DH	DL

# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

**mov cx, 7 ;nº caracteres**

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	DH	DL

# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

**mov dx, msg ;string**

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg (endereço da string a escrever)	

# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

**int 21h ;executa**

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

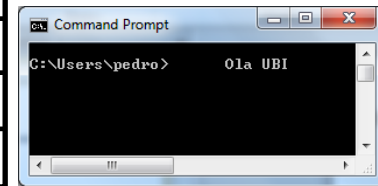
→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg (endereço da string a escrever)	

**escreve no ecran**





# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

**mov ah, 3Fh ;leitura**

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

msg db 'Ola UBI'

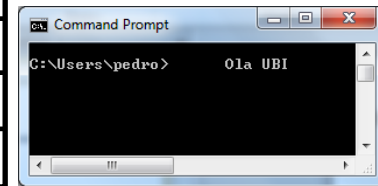
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg (endereço da string a escrever)	



AX	3Fh = 0011 1111b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg (endereço da string a escrever)	



# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

**int 21h ;executa**

;terminar o programa

mov ah, 4Ch ;terminar

int 21h ;executa

msg db 'Ola UBI'

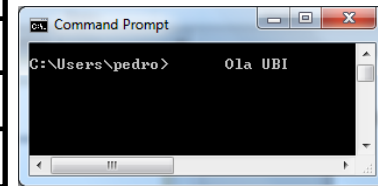
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg (endereço da string a escrever)	



AX	3Fh = 0011 1111b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg (endereço da string a escrever)	



**aguarda por tecla**



# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

**mov ah, 4Ch ;terminar**

int 21h ;executa

msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



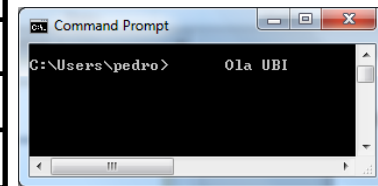
AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg	



AX	3Fh = 0011 1111b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg	



AX	4Ch = 0100 1100b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg	



# Execução dos programas

1º exemplo: escrever no ecrã a mensagem “Ola UBI” → printf(“Ola UBI” )

## Programa Assembly

org 100h ;inicio do programa

;escrever no ecran

mov ah, 40h ;escrita

mov bx, 1 ;ecran

mov cx, 7 ;nº caracteres

mov dx, msg ;string

int 21h ;executa

;aguarda tecla

mov ah, 3Fh ;leitura

int 21h ;executa

;terminar o programa

mov ah, 4Ch ;terminar

**int 21h ;executa**

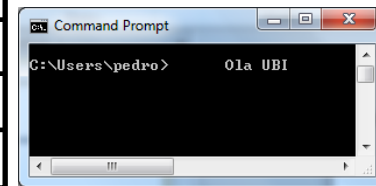
→ msg db 'Ola UBI'

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

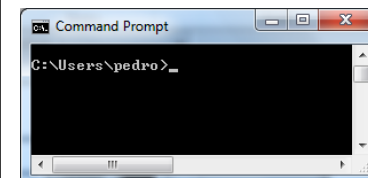
AX	40h = 0100 0000b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg	

AX	3Fh = 0011 1111b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg	

AX	4Ch = 0100 1100b	AL
BX	1 → 0000 0000 0000 0001	
CX	7 → 0000 0000 0000 0111	
DX	msg	



**termina**



# Execução dos programas

org 100h ;início do programa

;escrever no ecran

mov ah, 40h ↔ B440 ;escrita

mov bx, 1 ↔ BB0100 ;ecran

mov cx, 7 ↔ B90700 ;nº char

mov dx, msg ↔ BA???? ;string

int 21h ↔ CD21 ;executa

;aguarda tecla

mov ah, 3Fh ↔ B43F ;leitura

int 21h ↔ CD21 ;executa

;terminar o programa

mov ah, 4Ch ↔ B44C ;terminar

int 21h ↔ CD21 ;executa

msg db 'Ola UBI' ;[4F,6C,61,20,55,42,49]  
códigos ASCII

programa

dados

## Memória

address

1011 0100	0100	B4h
0100 0000	0101	40h
1011 1011	0102	BBh
0000 0001	0103	01h
0000 0000	0104	00h
1011 1001	0105	B9h
0000 0111	0106	07h
0000 0000	0107	00h
1011 1010	0108	BAh
?	0109	?
?	010A	?
1100 1101	010B	CDh
0010 0001	010C	21h
1011 0100	010D	B4h
0011 1111	010E	3Fh
1100 1101	010F	CDh
0010 0001	0110	21h
1011 0100	0111	B4h
0100 1100	0112	4Ch
1100 1101	0113	CDh
0010 0001	0114	21h
0100 1111	0115	'O' ← msg
0110 1100	0116	'l'
0110 0001	0117	'a'
0010 0000	0118	
0101 0101	0119	'U'
0100 0010	011A	'B'
0100 1001	011B	'i'
.....	.....	
	M	

little-endian

little-endian

little-endian

Big-endian: os bytes são armazenados da esquerda(maior byte)  
para a direita (menor byte)

Little-endian: os bytes são armazenados da direita(menor byte)  
para a esquerda(maior byte) – INTEL x86

# Execução dos programas

org 100h ;início do programa

;escrever no ecran

mov ah, 40h ↔ B440 ;escrita

mov bx, 1 ↔ BB0100 ;ecran

mov cx, 7 ↔ B90700 ;nº char

mov dx, msg ↔ BA1501 ;string

int 21h ↔ CD21 ;executa

;aguarda tecla

mov ah, 3Fh ↔ B43F ;leitura

int 21h ↔ CD21 ;executa

;terminar o programa

mov ah, 4Ch ↔ B44C ;terminar

int 21h ↔ CD21 ;executa

msg db 'Ola UBI' ;[4F,6C,61,20,55,42,49]

códigos ASCII

programa

dados

## Memória

address

1011 0100	0100	B4h
0100 0000	0101	40h
1011 1011	0102	BBh
0000 0001	0103	01h
0000 0000	0104	00h
1011 1001	0105	B9h
0000 0111	0106	07h
0000 0000	0107	00h
1011 1010	0108	BAh
0001 1001	0109	15h
0000 0001	010A	01h
1100 1101	010B	CDh
0010 0001	010C	21h
1011 0100	010D	B4h
0011 1111	010E	3Fh
1100 1101	010F	CDh
0010 0001	0110	21h
1011 0100	0111	B4h
0100 1100	0112	4Ch
1100 1101	0113	CDh
0010 0001	0114	21h
0100 1111	0115	'O' ← msg
0110 1100	0116	'l'
0110 0001	0117	'a'
0010 0000	0118	
0101 0101	0119	'U'
0100 0010	011A	'B'
0100 1001	011B	'i'
.....	.....	M

Big-endian: os bytes são armazenados da esquerda(maior byte)  
para a direita (menor byte)

Little-endian: os bytes são armazenados da direita(menor byte)  
para a esquerda(maior byte) – INTEL x86

# Execução dos programas

org 100h ;início do programa

;aguarda tecla

mov ah, 3Fh ↔ B43F ;leitura

int 21h ↔ CD21 ;executa

;terminar o programa

mov ah, 4Ch ↔ B44C ;terminar

int 21h ↔ CD21 ;executa

msg db 'Ola UBI' [4F,6C,61,20,55,42,49]

;escrever no ecran

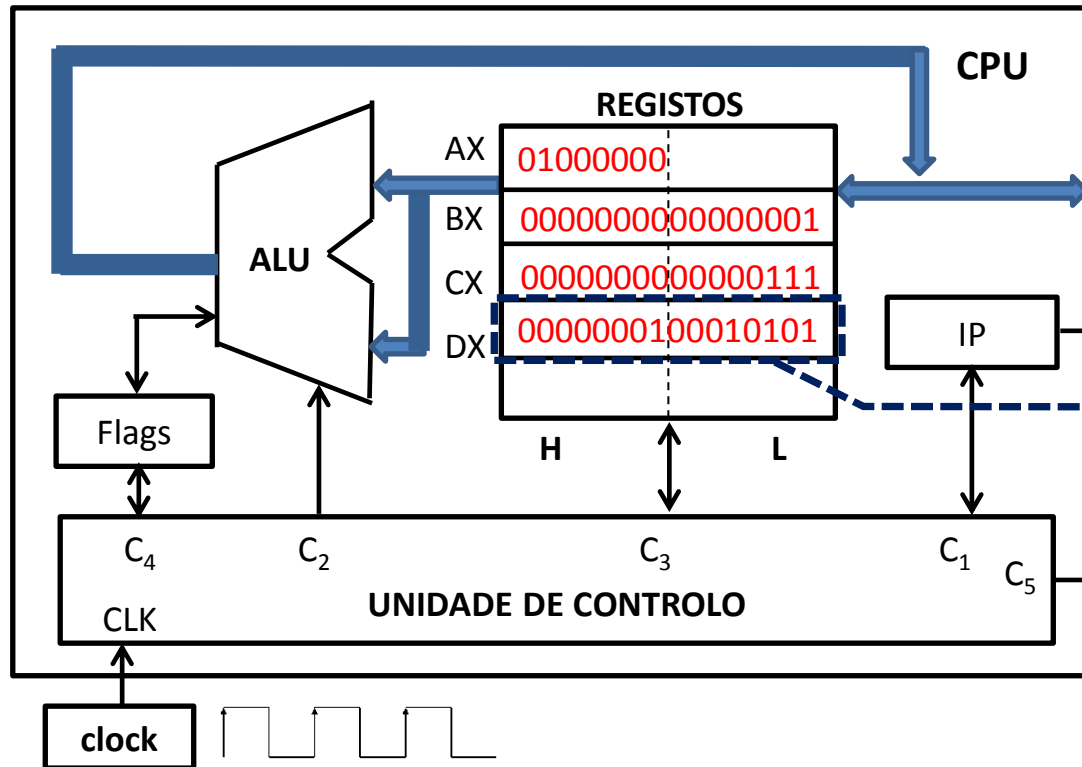
mov ah, 40h ↔ B440 ;escrita

mov bx, 1 ↔ BB0100 ;ecran

mov cx, 7 ↔ B90700 ;nº char

mov dx, msg ↔ BA1501 ;string

int 21h ↔ CD21 ;executa



## Memória

address

1011 0100	0100	B4h
0100 0000	0101	40h
1011 1011	0102	BBh
0000 0001	0103	01h
0000 0000	0104	00h
1011 1001	0105	B9h
0000 0111	0106	07h
0000 0000	0107	00h
1011 1010	0108	BAh
0001 0101	0109	15h
0000 0001	010A	01h
1100 1101	010B	CDh
0010 0001	010C	21h
1011 0100	010D	B4h
0011 1111	010E	3Fh
1100 1101	010F	CDh
0010 0001	0110	21h
1011 0100	0111	B4h
0100 1100	0112	4Ch
1100 1101	0113	CDh
0010 0001	0114	21h
0100 1111	0115	'O' ← msg
0110 1100	0116	'l'
0110 0001	0117	'a'
0010 0000	0118	
0101 0101	0119	'U'
0100 0010	011A	'B'
0100 1001	011B	'I'
.....	.....	.....
	M	

# Execução dos programas

org 100h ;inicio do programa

;aguarda tecla

**mov ah, 3Fh** ↔ B43F ;leitura

**int 21h** ↔ CD21 ;executa

;escrever no ecran

mov ah, 40h ↔ B440 ;escrita

mov bx, 1 ↔ BB0100 ;ecran

mov cx, 7 ↔ B90700 ;nº char

mov dx, msg ↔ BA1501 ;string

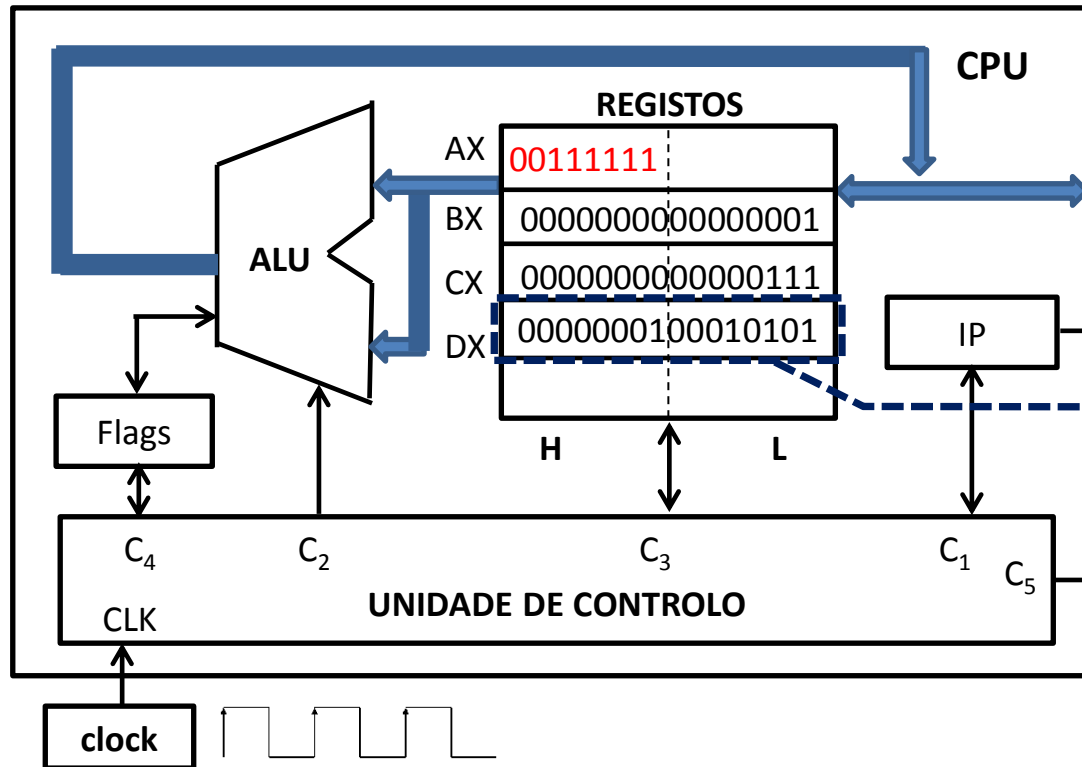
int 21h ↔ CD21 ;executa

;terminar o programa

mov ah, 4Ch ↔ B44C ;terminar

int 21h ↔ CD21 ;executa

msg db 'Ola UBI' [4F,6C,61,20,55,42,49]



## Memória

address

1011 0100	0100	B4h
0100 0000	0101	40h
1011 1011	0102	BBh
0000 0001	0103	01h
0000 0000	0104	00h
1011 1001	0105	B9h
0000 0111	0106	07h
0000 0000	0107	00h
1011 1010	0108	BAh
0001 0101	0109	15h
0000 0001	010A	01h
1100 1101	010B	CDh
0010 0001	010C	21h
1011 0100	010D	B4h
0011 1111	010E	3Fh
1100 1101	010F	CDh
0010 0001	0110	21h
1011 0100	0111	B4h
0100 1100	0112	4Ch
1100 1101	0113	CDh
0010 0001	0114	21h
0100 1111	0115	'O' ← msg
0110 1100	0116	'l'
0110 0001	0117	'a'
0010 0000	0118	
0101 0101	0119	'U'
0100 0010	011A	'B'
0100 1001	011B	'I'
.....	.....	.....
	M	



# Execução dos programas

org 100h ;inicio do programa

;aguarda tecla

mov ah, 3Fh ↔ B43F

;leitura(default=teclado)

int 21h ↔ CD21 ;executa

;terminar o programa

**mov ah, 4Ch ↔ B44C ;terminar**

**int 21h ↔ CD21 ;executa**

msg db 'Ola UBI' [4F,6C,61,20,55,42,49]

;escrever no ecran

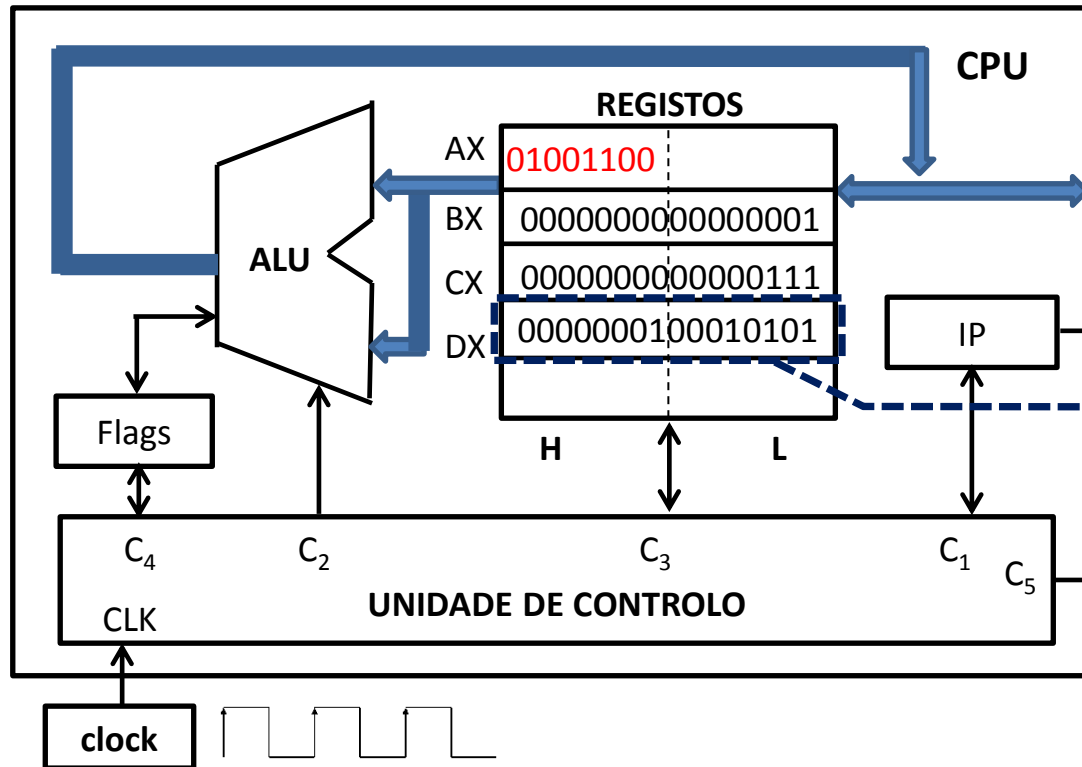
mov ah, 40h ↔ B440 ;escrita

mov bx, 1 ↔ BB0100 ;ecran

mov cx, 7 ↔ B90700 ;nº char

mov dx, msg ↔ BA1501 ;string

int 21h ↔ CD21 ;executa



## Memória

address

1011 0100	0100	B4h
0100 0000	0101	40h
1011 1011	0102	BBh
0000 0001	0103	01h
0000 0000	0104	00h
1011 1001	0105	B9h
0000 0111	0106	07h
0000 0000	0107	00h
1011 1010	0108	BAh
0001 0101	0109	15h
0000 0001	010A	01h
1100 1101	010B	CDh
0010 0001	010C	21h
1011 0100	010D	B4h
0011 1111	010E	3Fh
1100 1101	010F	CDh
0010 0001	0110	21h
1011 0100	0111	B4h
0100 1100	0112	4Ch
1100 1101	0113	CDh
0010 0001	0114	21h
0100 1111	0115	'O' ← msg
0110 1100	0116	'l'
0110 0001	0117	'a'
0010 0000	0118	
0101 0101	0119	'U'
0100 0010	011A	'B'
0100 1001	011B	'I'
.....	.....	.....
.....	.....	M

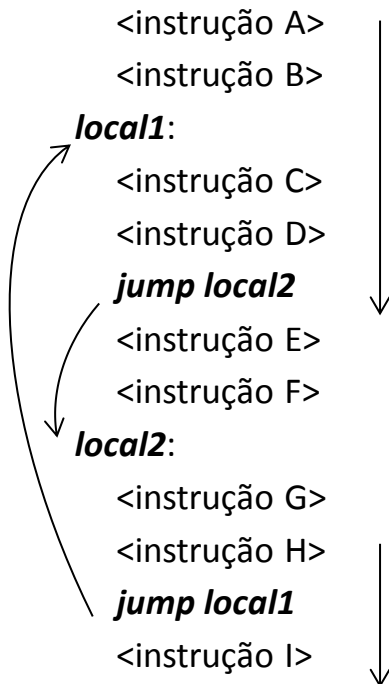
# Execução dos programas

**Instruções de salto (jump) :** alteram a sequência de execução das instruções

- permitem programar ciclos e tomada de decisões em função de determinadas condições

jump incondicional – não depende de nenhuma condição para saltar

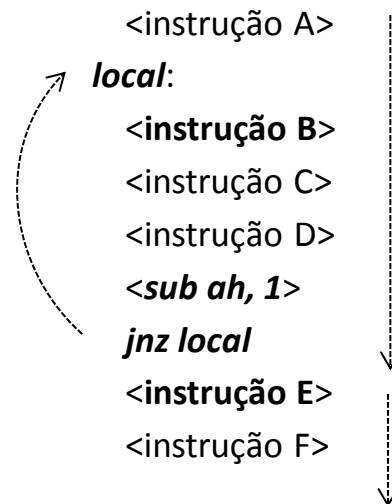
ex: *jump local* → ao encontrar esta instrução o programa continua a partir de “local”



jump condicional – depende de uma condição para saltar

ex: *sub ah, 1* ;subtrai 1 ao registo AH

*jnz local* → (jump if not zero) : se AH≠0 o programa continua a partir de “local”(instrução B), senão continua na instrução seguinte (instrução E)



# Instruções de salto (jump)

exemplo: repetir 20 vezes a escrita de “UBI”

```
org 100h
```

```
mov [cont], 20 ;20=14h
```

➤ ciclo:

```
mov ah, 40h ;escrita
```

```
mov bx, 1 ;ecran
```

```
mov cx, 4 ;nº caracteres
```

```
mov dx, msg ;string
```

```
int 21h ;executa
```

```
dec [cont] ;cont=cont-1 → é o resultado da operação que está imediatamente
```

- - jnz ciclo ;cont=0 ? antes do *jump* que controla o salto: neste caso haverá salto enquanto a variável “cont” não for zero

```
mov ah, 07h ;leitura
```

```
int 21h ;executa
```

```
mov ah, 4ch ;terminar
```

```
int 21h ;executa
```

```
msg db "UBI", 10 ;10=0Ah=new line
```

```
cont rb 1
```

‘U’=55h ‘B’=42h ‘I’=49h

# Sub-rotinas

Os ciclos permitem repetir um bloco de instruções num certo local do programa

Por vezes esse bloco precisa de ser usado várias vezes e em diferentes locais do programa:

- podem usar-se *subrotinas* que funcionam como pequenos programas dentro do programa principal;
- as subrotinas são chamadas (*call*) a partir de qualquer local do programa principal, executam a sua função e retornam (*ret*) devolvendo o controlo ao programa principal;
- torna-se necessário armazenar o local de onde a subrotina é chamada para que quando esta terminar o programa retome a execução a partir daí;

ex: na sequência abaixo, ao executar a primeira instrução *call nome\_subrotina*, o programa armazena o endereço de <instrução C> que representa o endereço de retorno da subrotina e em seguida executa o corpo desta; a instrução *ret* faz o programa retomar a execução a partir do endereço que foi previamente armazenado.

Na segunda chamada (a vermelho) o endereço de retorno é o da <instrução F> .

## programa principal

<instrução A>

<instrução B>

***call nome\_subrotina***

<instrução C>

<instrução D>

<instrução E>

***call nome\_subrotina***

<instrução F>

## subrotina

***nome\_subrotina:***

<instrução X>

<instrução Y>

<instrução Z>

***ret***

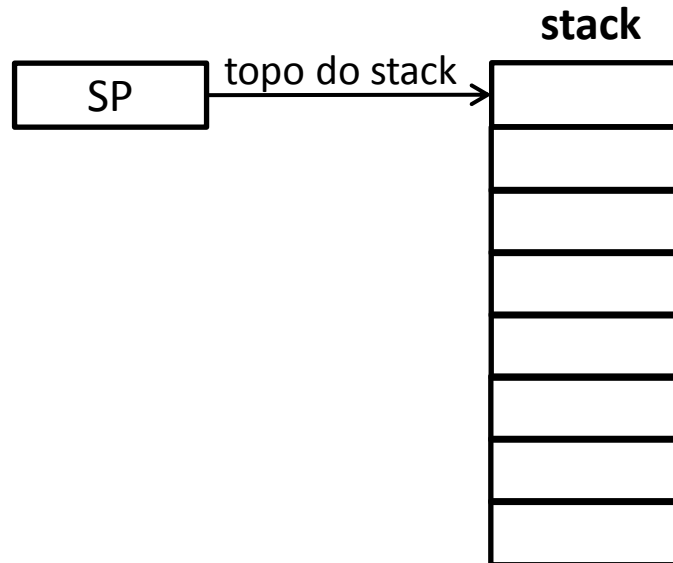
armazena endereço de retorno

retorna ao endereço armazenado

# Uso do stack (pilha)

O stack é uma estrutura de dados que serve para armazenar valores de forma temporária.

É constituído por uma zona reservada de memória, cujos endereços são referenciados através de um registo especial designado *SP-stack pointer*.



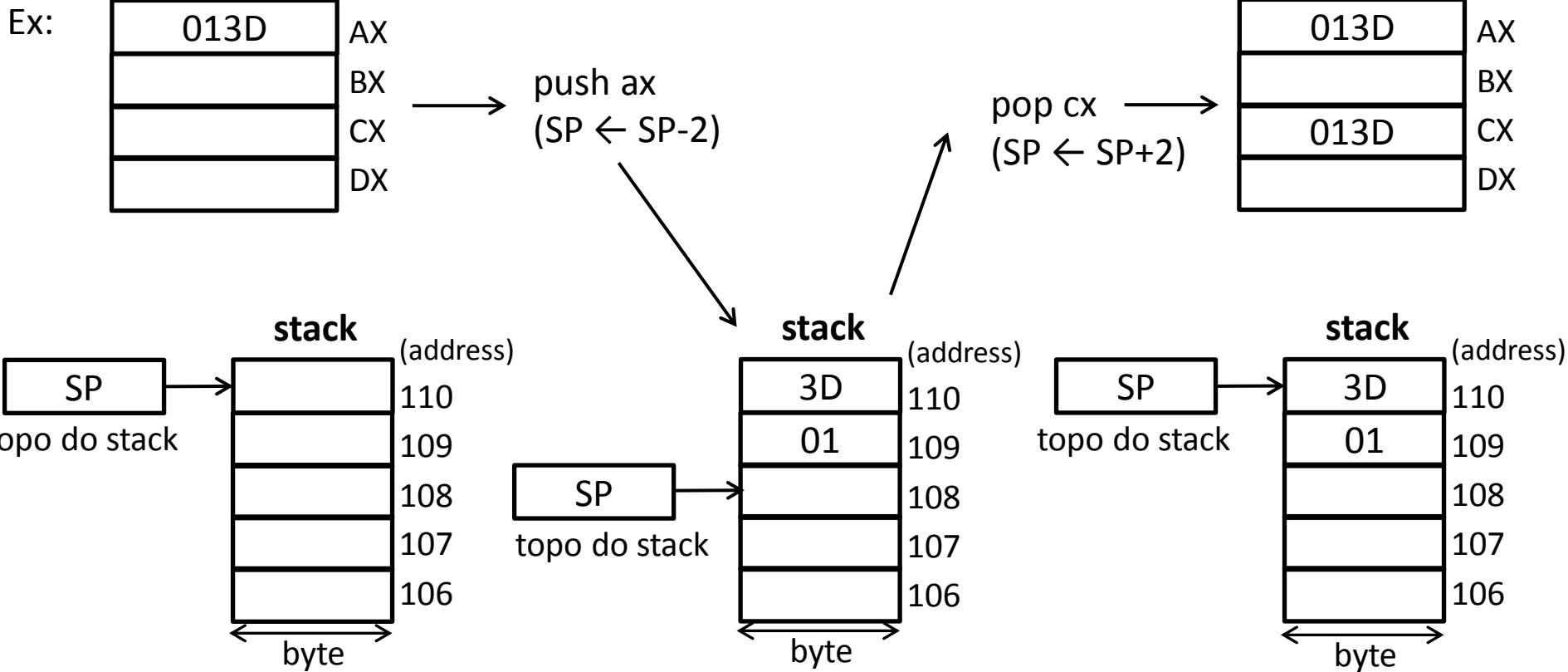
O registo SP aponta para o próximo endereço livre, designado por *topo do stack*, o qual habitualmente vai diminuindo (descendo) à medida que a pilha vai contendo mais dados.

Além de permitir armazenar variáveis o stack também é usado para armazenar o endereço de retorno das sub-rotinas.

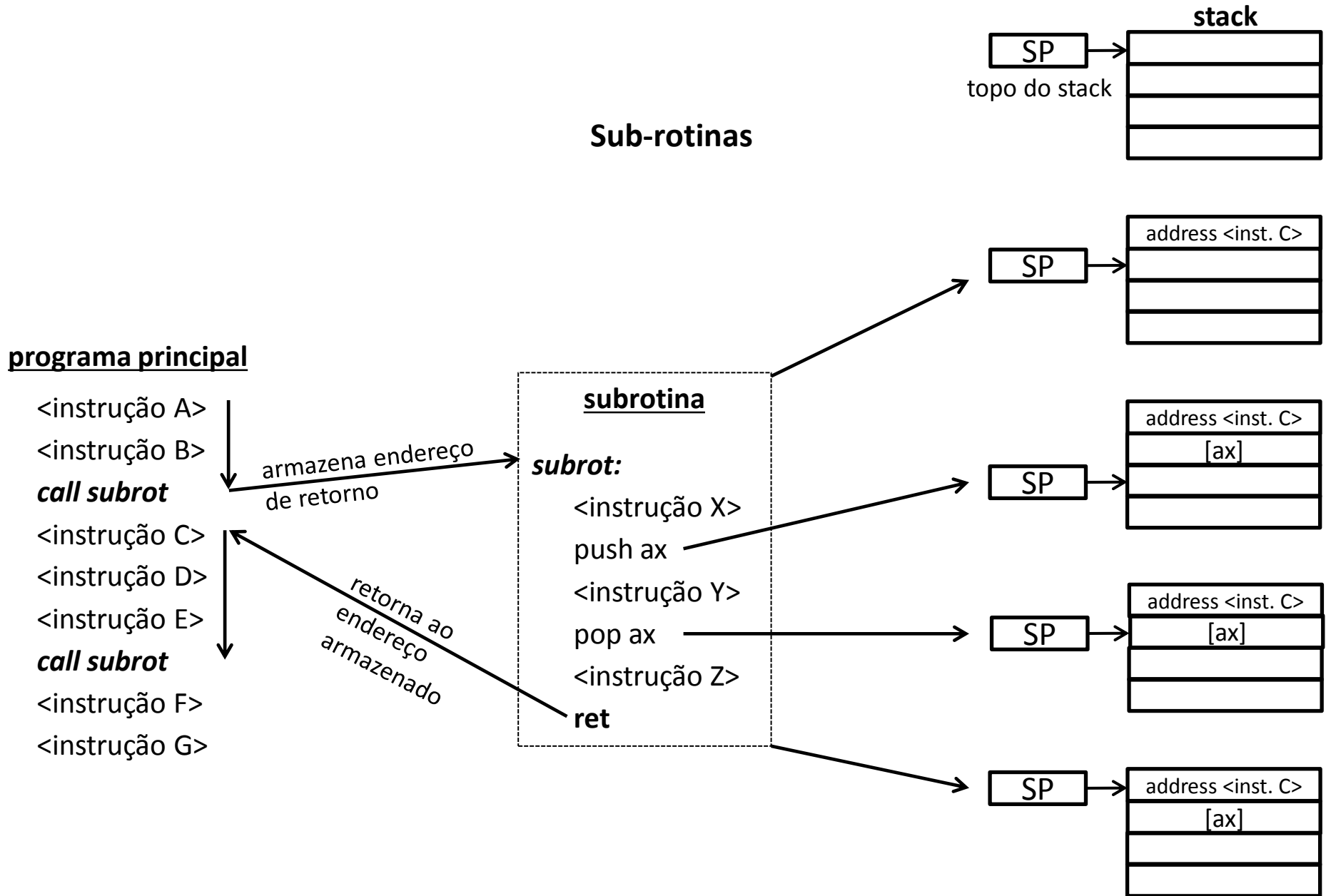
# Uso do stack

Instruções de manipulação do stack (16 bit):

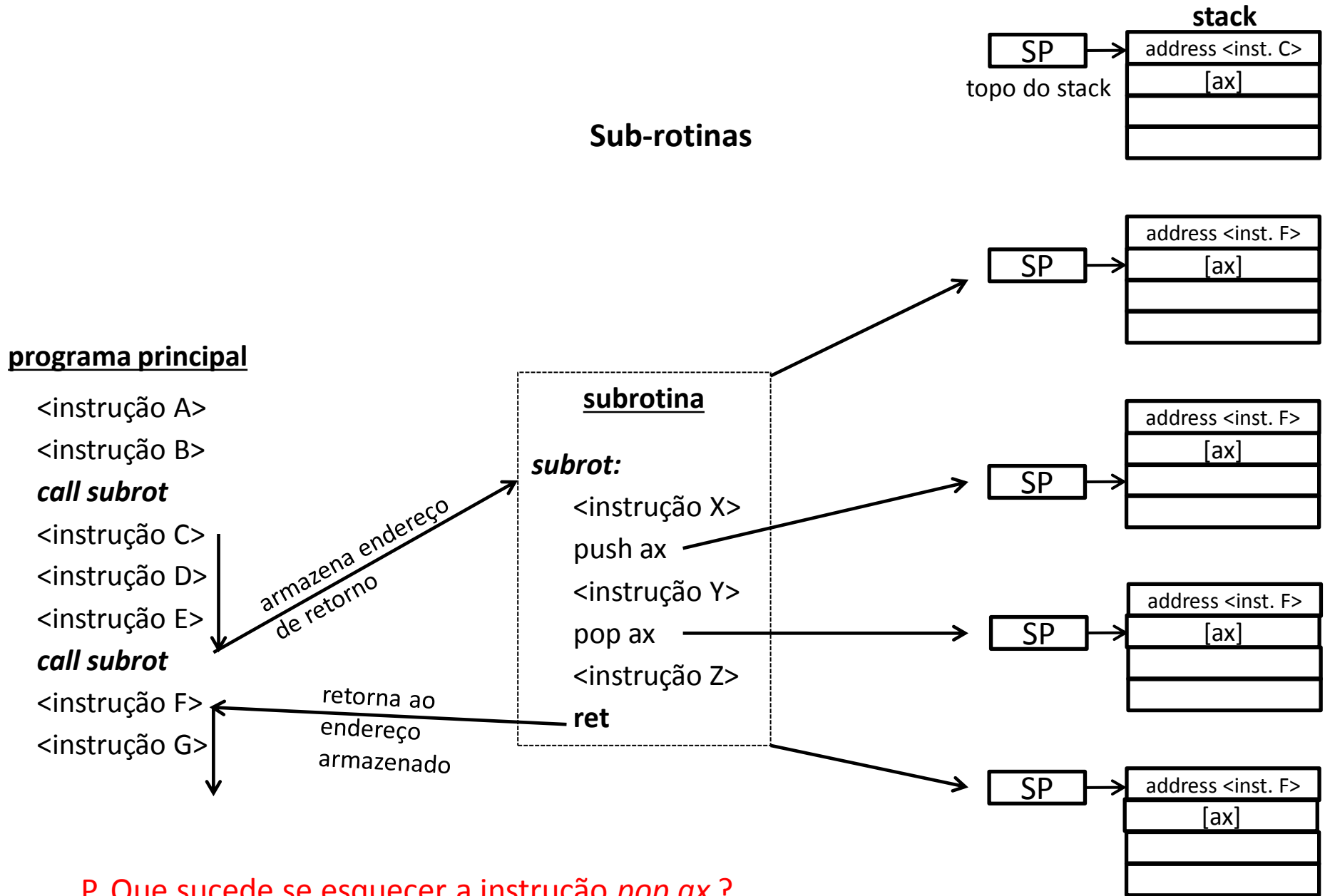
- *push* <val> : insere um valor no topo do stack  
<val> pode ser um valor imediato ou o conteúdo de um registro;
- *pop* <local> : retira o valor que está no topo do stack e coloca-o em <local> , este pode ser uma variável ou um registro;



# Uso do stack



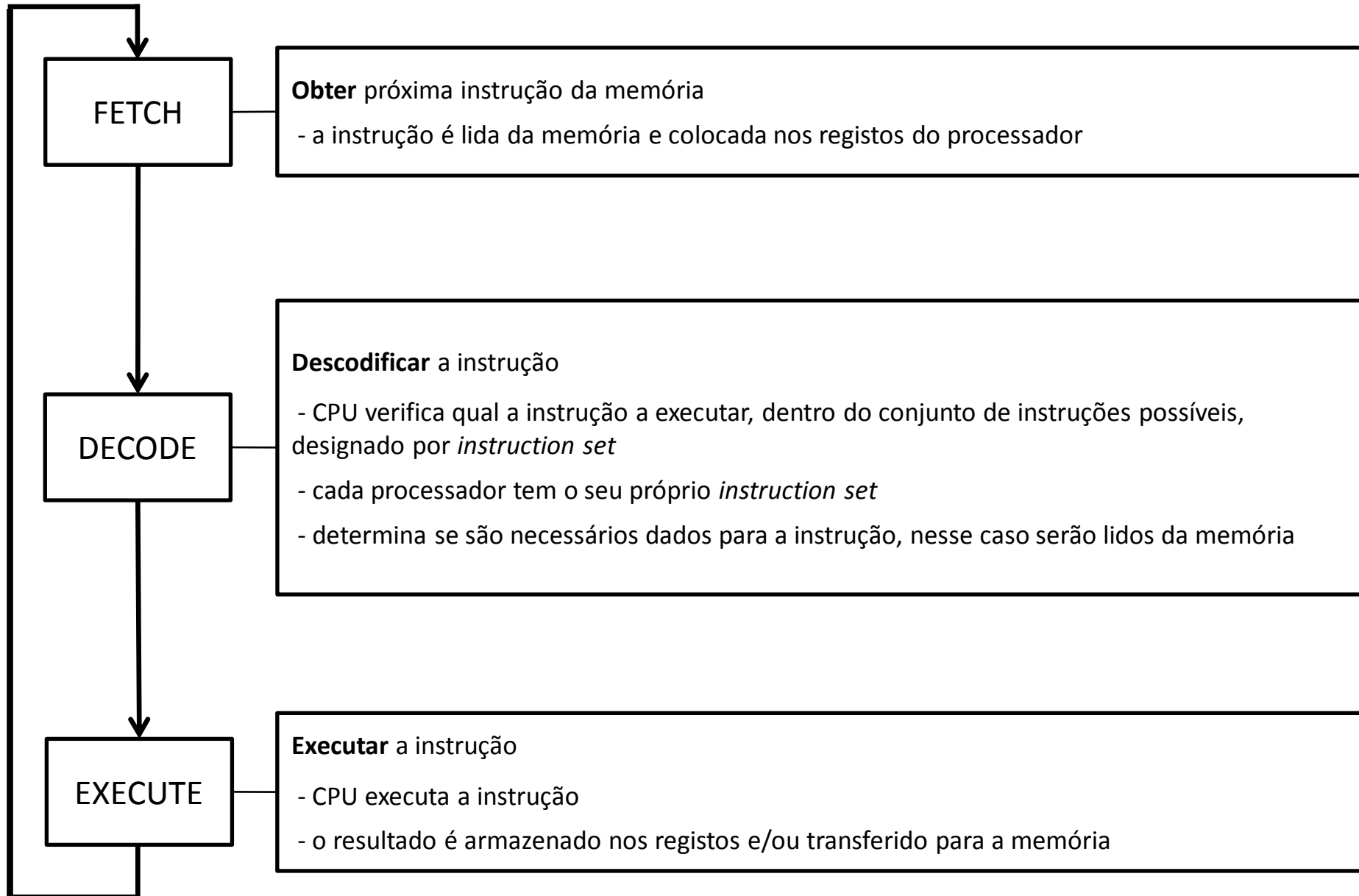
# Uso do stack



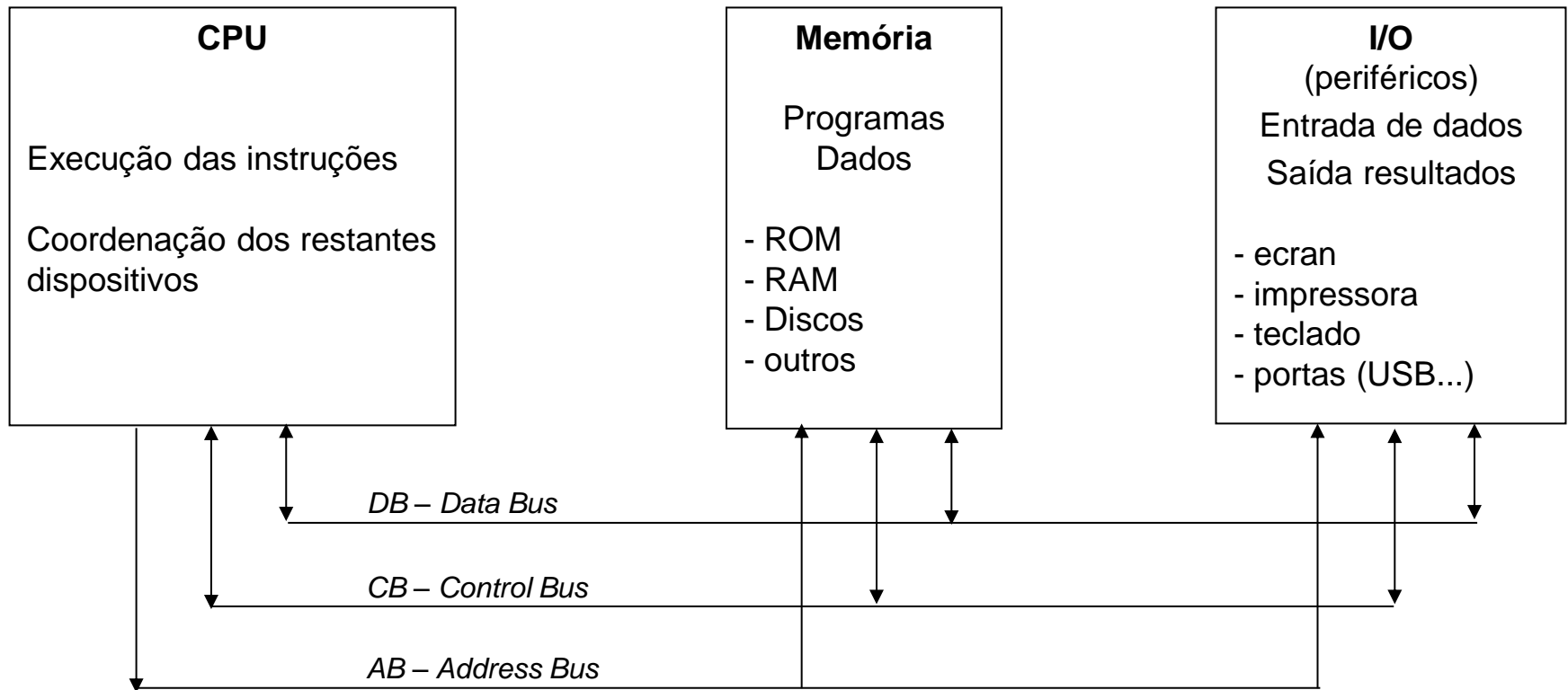
P. Que sucede se esquecer a instrução *pop ax* ?



# Ciclo: FETCH – DECODE - EXECUTE



# CPU – MEMÓRIA – PERIFÉRICOS(Input/Output)



**Bus (barramento) :** conjunto de linhas de comunicação que interligam os vários componentes de um sistema de computação. Principais características: largura(nº de bits), velocidade de transmissão(bps-bits por segundo)

DB(Data Bus) – caminho dos dados, bidireccional. (P IV: 64/128 bits externos, 32/64 internos; 3.2GB/s)

CB(Control Bus) – bidireccional, sinais de controlo. (ex: Read, Write, Reset)

AB(Address Bus) – unidireccional, sinais de endereço (P IV: 32 bits/4GB MEM ; 36bits/64GB)

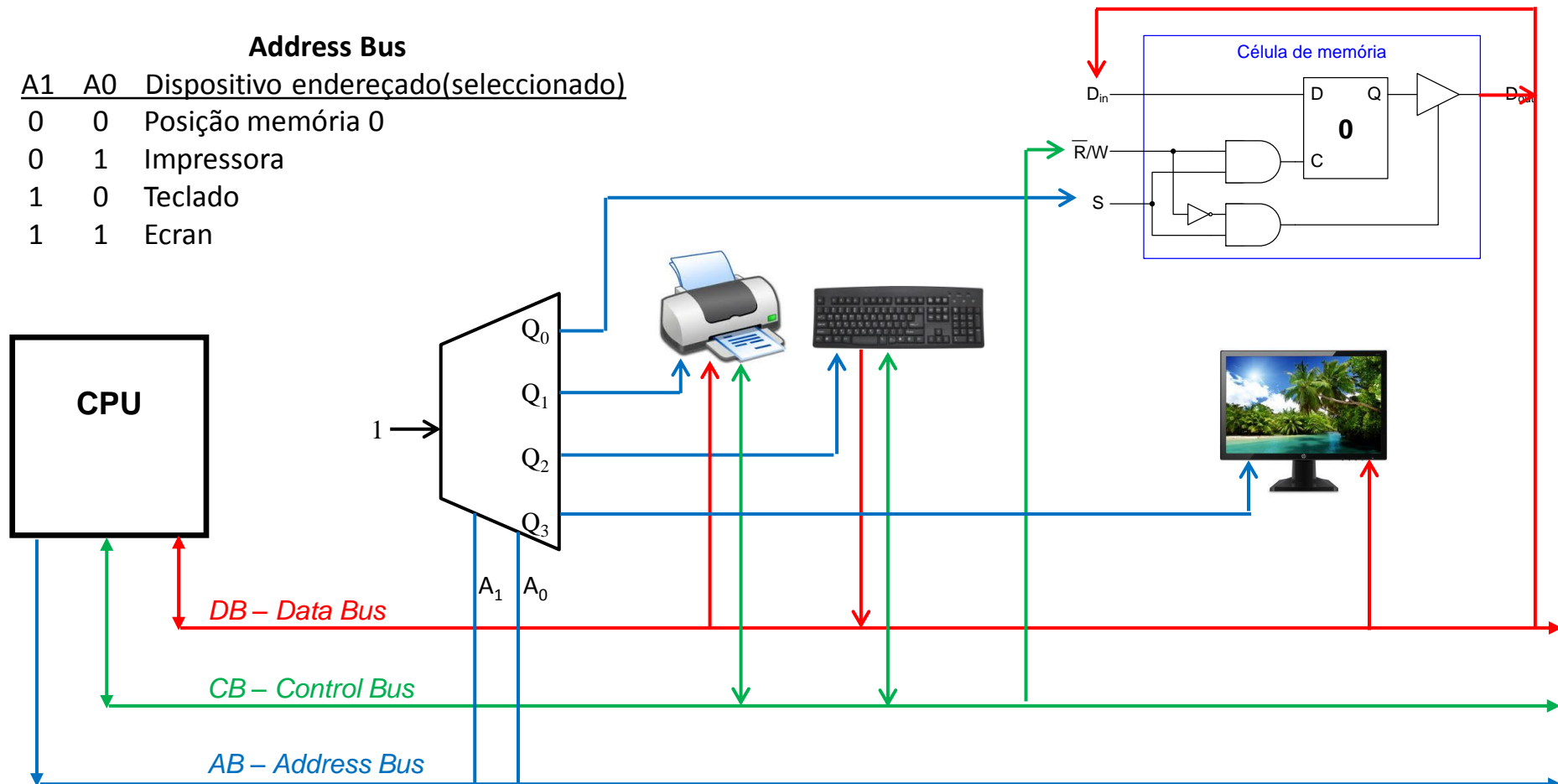
# Endereçamento

Como seleccionar um de entre vários dispositivos (ex: ler da memória ou de um periférico) ?

Através do mecanismo de endereçamento → só o periférico endereçado fica activo

O AB (Address Bus) é usado para enviar o endereço do dispositivo a seleccionar

Ex: imaginemos um CPU capaz de aceder apenas a 4 dispositivos (registo, teclado, impressora, ecran)



# Métrica binária

## Agrupamentos de n bits

n	gama= $2^n$	designação	nº de bytes
1	2 : (0 , 1)	bit	-
4	16 : (0 .. 15)	nibble	-
8	256 : (0 .. 255)	byte (octeto)	1
10	1024 : (0 .. 1023)	Kbit	-
16	65536 : (0 .. 65535)	Palavra 16 bits (word)	2
32	$2^{32}$ : (0 .. $2^{32}-1$ )	Palavra 32 bits	4
64	$2^{64}$ : (0 .. $2^{64}-1$ )	Palavra 64 bits	8

bit = binary digit

byte = binary term

$$K=2^{10}=1024$$

NOTA: a definição de “word” varia consoante a máquina usada, máquinas com registos de 16 bit → word=16 bit; máquinas com registos de 32 bit → word=32 bit

## Agrupamentos de bytes

binário	gama	unidade	decimal aproximado
$2^{10}$	1024 byte	K (Kilo) - Kbyte	$10^3$
$2^{20}$	1024 * K = 1 048 576 byte	M (Mega) - Mbyte	$10^6$
$2^{30}$	1024 * M = 1 073 741 824 byte	G (Giga) - Gbyte	$10^9$
$2^{40}$	1024 * G = 1 099 511 627 776 byte	T (Tera) - Tbyte	$10^{12}$

$$K=2^{10}=1024$$

# Métrica binária

Correspondência de números em diversas bases

Decimal (pesos) $10^1$ $10^0$	Binário (pesos) $2^3$ $2^2$ $2^1$ $2^0$ 8 4 2 1	Hexadecimal (pesos) $16^0$
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	3
4	0 1 0 0	4
5	0 1 0 1	5
6	0 1 1 0	6
7	0 1 1 1	7
8	1 0 0 0	8
9	1 0 0 1	9
10	1 0 1 0	A
11	1 0 1 1	B
12	1 1 0 0	C
13	1 1 0 1	D
14	1 1 1 0	E
15	1 1 1 1	F

← nibble →

1 byte = 2 nibble = 2 dígitos hexadecimais      ex: E4h = 1110 0100b

# Métrica Binária

A norma IEC 80000-13: *Quantities and units – Part 13: Information science and technology* , publicada em 2008 define os seguintes prefixos binários:

Nome	Símbolo	Potência = valor
kibi	Ki	$2^{10} = 1024$
mebi	Mi	$2^{20} = 1\,048\,576$
gibi	Gi	$2^{30} = 1\,073\,741\,824$
tebi	Ti	$2^{40} = 1\,099\,511\,627\,776$
pebi	Pi	$2^{50} = 1\,125\,899\,906\,842\,624$
exbi	Ei	$2^{60} = 1\,152\,921\,504\,606\,846\,976$
zebi	Zi	$2^{70} = 1\,180\,591\,620\,717\,411\,303\,424$
yobi	Yi	$2^{80} = 1\,208\,925\,819\,614\,629\,174\,706\,176$

*Prefixos binários segundo a norma IEC 80000-13 (2008).*

Consultar: Prefixes for binary multiples - <http://physics.nist.gov/cuu/Units/binary.html>

Exemplo: 1 KByte =  $10^3$  Byte = 1000 Byte (Kilo Byte, decimal)  
1 KiByte =  $2^{10}$  Byte = 1024 Byte (Kibi Byte, binário)

# Tabela ASCII - American Standard Code for Information Interchange

Relaciona os caracteres com a sua representação numérica (decimal, hexadecimal, binária)

Tabela ASCII de 8 bit → 1 carácter = 1 byte      ex: letra 'A' → 'A' = 65 = 41h = 0100 0001b

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	<b>Space</b>	64	40	100	&#64;	<b>@</b>	96	60	140	&#96;	<b>`</b>
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	<b>!</b>	65	41	101	&#65;	<b>A</b>	97	61	141	&#97;	<b>a</b>
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	<b>"</b>	66	42	102	&#66;	<b>B</b>	98	62	142	&#98;	<b>b</b>
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	<b>#</b>	67	43	103	&#67;	<b>C</b>	99	63	143	&#99;	<b>c</b>
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	<b>\$</b>	68	44	104	&#68;	<b>D</b>	100	64	144	&#100;	<b>d</b>
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	<b>%</b>	69	45	105	&#69;	<b>E</b>	101	65	145	&#101;	<b>e</b>
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	<b>&amp;</b>	70	46	106	&#70;	<b>F</b>	102	66	146	&#102;	<b>f</b>
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	<b>'</b>	71	47	107	&#71;	<b>G</b>	103	67	147	&#103;	<b>g</b>
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	<b>(</b>	72	48	110	&#72;	<b>H</b>	104	68	150	&#104;	<b>h</b>
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	<b>)</b>	73	49	111	&#73;	<b>I</b>	105	69	151	&#105;	<b>i</b>
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	<b>*</b>	74	4A	112	&#74;	<b>J</b>	106	6A	152	&#106;	<b>j</b>
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	<b>+</b>	75	4B	113	&#75;	<b>K</b>	107	6B	153	&#107;	<b>k</b>
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	<b>,</b>	76	4C	114	&#76;	<b>L</b>	108	6C	154	&#108;	<b>l</b>
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	<b>-</b>	77	4D	115	&#77;	<b>M</b>	109	6D	155	&#109;	<b>m</b>
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	<b>.</b>	78	4E	116	&#78;	<b>N</b>	110	6E	156	&#110;	<b>n</b>
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	<b>/</b>	79	4F	117	&#79;	<b>O</b>	111	6F	157	&#111;	<b>o</b>
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	<b>0</b>	80	50	120	&#80;	<b>P</b>	112	70	160	&#112;	<b>p</b>
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	<b>1</b>	81	51	121	&#81;	<b>Q</b>	113	71	161	&#113;	<b>q</b>
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	<b>2</b>	82	52	122	&#82;	<b>R</b>	114	72	162	&#114;	<b>r</b>
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	<b>3</b>	83	53	123	&#83;	<b>S</b>	115	73	163	&#115;	<b>s</b>
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	<b>4</b>	84	54	124	&#84;	<b>T</b>	116	74	164	&#116;	<b>t</b>
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	<b>5</b>	85	55	125	&#85;	<b>U</b>	117	75	165	&#117;	<b>u</b>
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	<b>6</b>	86	56	126	&#86;	<b>V</b>	118	76	166	&#118;	<b>v</b>
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	<b>7</b>	87	57	127	&#87;	<b>W</b>	119	77	167	&#119;	<b>w</b>
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	<b>8</b>	88	58	130	&#88;	<b>X</b>	120	78	170	&#120;	<b>x</b>
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	<b>9</b>	89	59	131	&#89;	<b>Y</b>	121	79	171	&#121;	<b>y</b>
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	<b>:</b>	90	5A	132	&#90;	<b>Z</b>	122	7A	172	&#122;	<b>z</b>
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	<b>;</b>	91	5B	133	&#91;	<b>[</b>	123	7B	173	&#123;	<b>{</b>
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<b>&lt;</b>	92	5C	134	&#92;	<b>\</b>	124	7C	174	&#124;	<b> </b>
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	<b>=</b>	93	5D	135	&#93;	<b>]</b>	125	7D	175	&#125;	<b>}</b>
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	<b>&gt;</b>	94	5E	136	&#94;	<b>^</b>	126	7E	176	&#126;	<b>~</b>
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	<b>?</b>	95	5F	137	&#95;	<b>_</b>	127	7F	177	&#127;	<b>DEL</b>

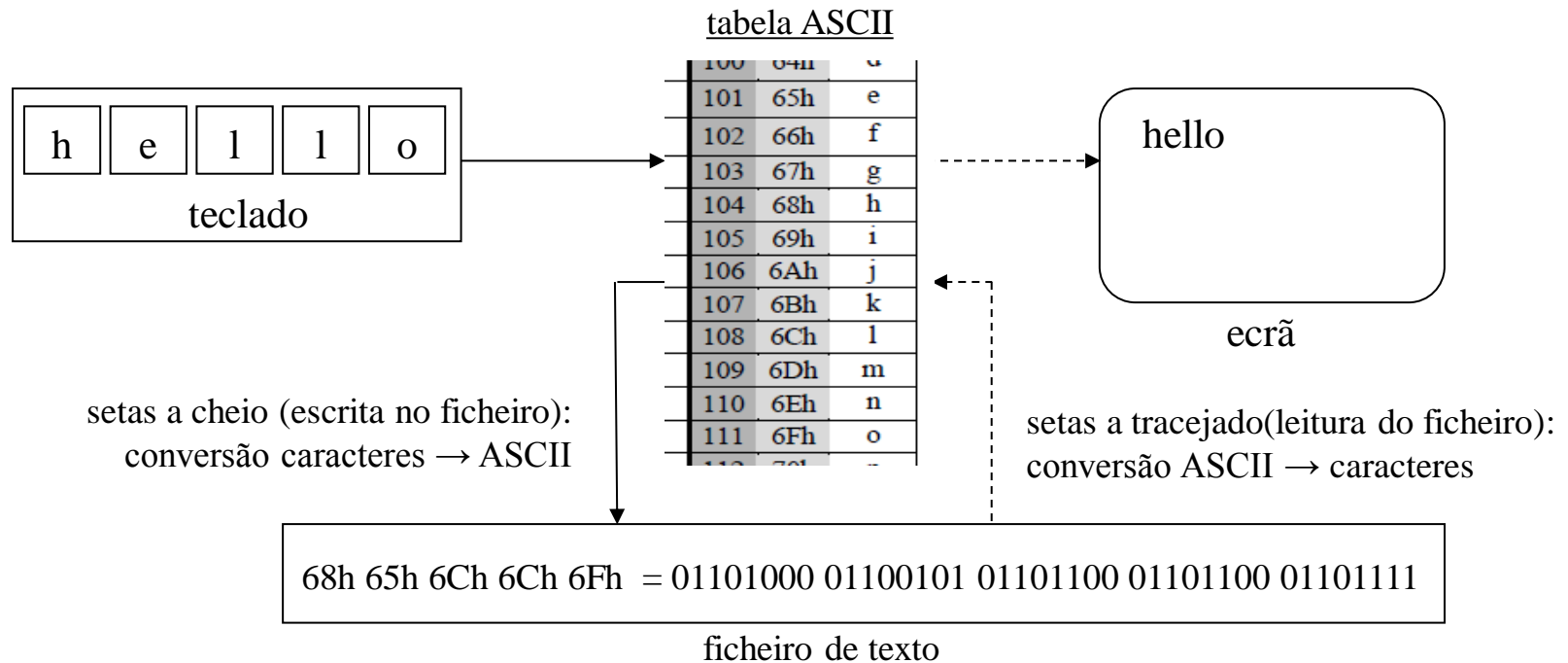
# Armazenamento da Informação

(texto , números)

## Texto

**Escrita** → para cada símbolo alfanumérico, sinal de pontuação, etc, é obtido o respectivo código da tabela ASCII, o qual é armazenado em memória (ficheiro);

**Leitura** → para cada código ASCII lido da memória (ficheiro) , é obtido o respectivo símbolo alfanumérico o qual é apresentado no ecrã;

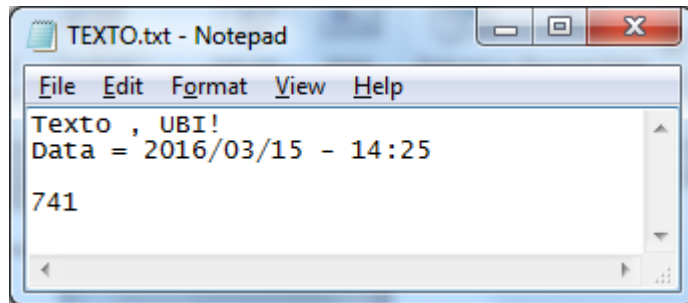




# Armazenamento da Informação

## Texto (letras, algarismos, sinais de pontuação)

editor de texto Notepad → ficheiro TEXTO.txt



editor hexadecimal “MiTeC” → ficheiro TEXTO.txt

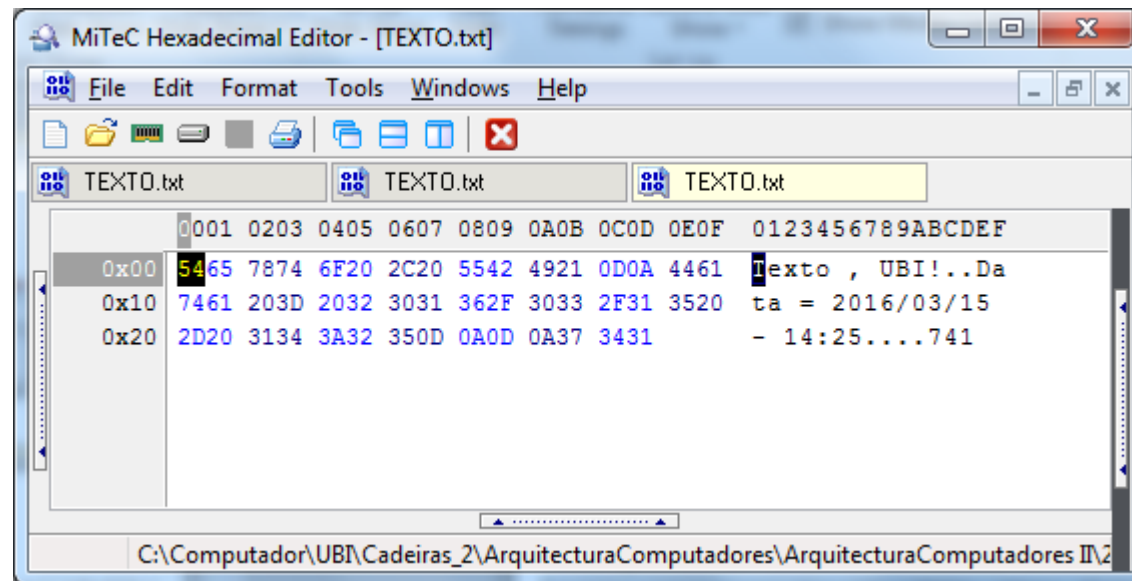


Tabela ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20h		64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	\$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_	127	7Fh	△

# Armazenamento da Informação

(texto , números)

## Valores numéricos (inteiros)

Como é armazenado o valor decimal 741?

conversão para binário (hexadecimal) →

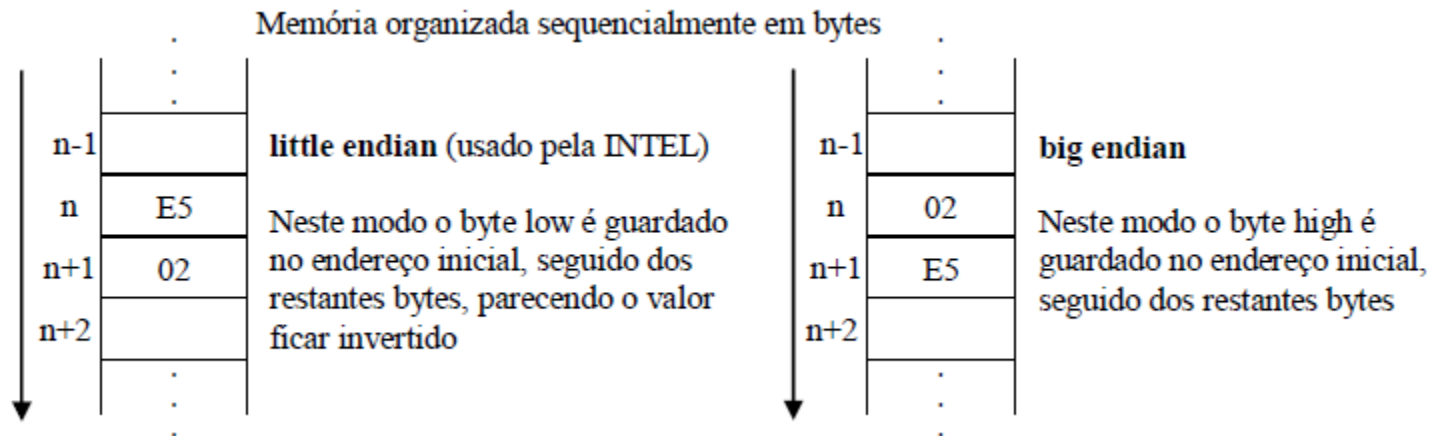
741		16	
736		46	16
	5	14	2   16
			2 0
	↓	↓	↓
	5	E	2
	(LSB)	←	(MSB)

→ 02E5h (2 byte)

memória → conjunto de bytes organizados sequencialmente

armazenamento de conjuntos de n bytes → dois modos possíveis:

- *little endian* : byte de menor peso primeiro (usado pela Intel, linguagem C, C#, etc)
- *big endian* : byte de maior peso primeiro (usado pelo Java)

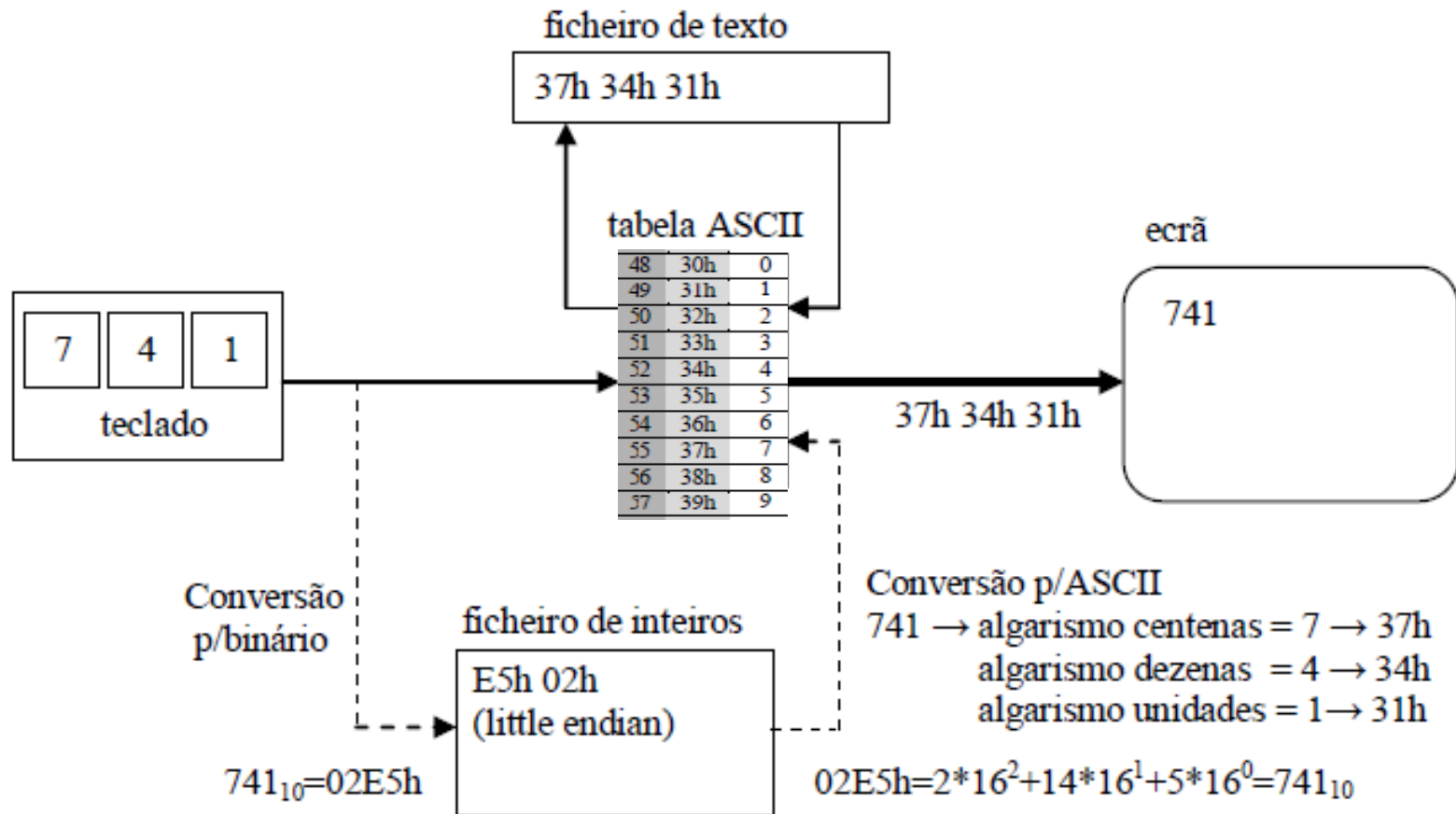


O termo *endian* tem origem no livro “As viagens de Gulliver” e refere-se à questão de qual dos lados os ovos cozidos devem ser quebrados.

# Armazenamento da Informação

(texto , números)

Diferença entre armazenamento em modo texto (ASCII) ou em modo binário (hexadecimal)



setas a cheio: tratamento de texto (cadeias de caracteres ASCII) → os programas de processamento de texto (NotePad, Word,...) interpretam dados em código ASCII;

setas a tracejado: tratamento de valores numéricos em binário (inteiros);

# Armazenamento da Informação

1) Valores numéricos: ficheiro de inteiros com o valor 741

C# → `using (BinaryWriter b = new BinaryWriter(File.Open("file.bin", FileMode.Create))) { b.Write(741); }`

int → 32 bits(4 bytes) com sinal , -2.147.483.648 ↔ 2.147.483.647

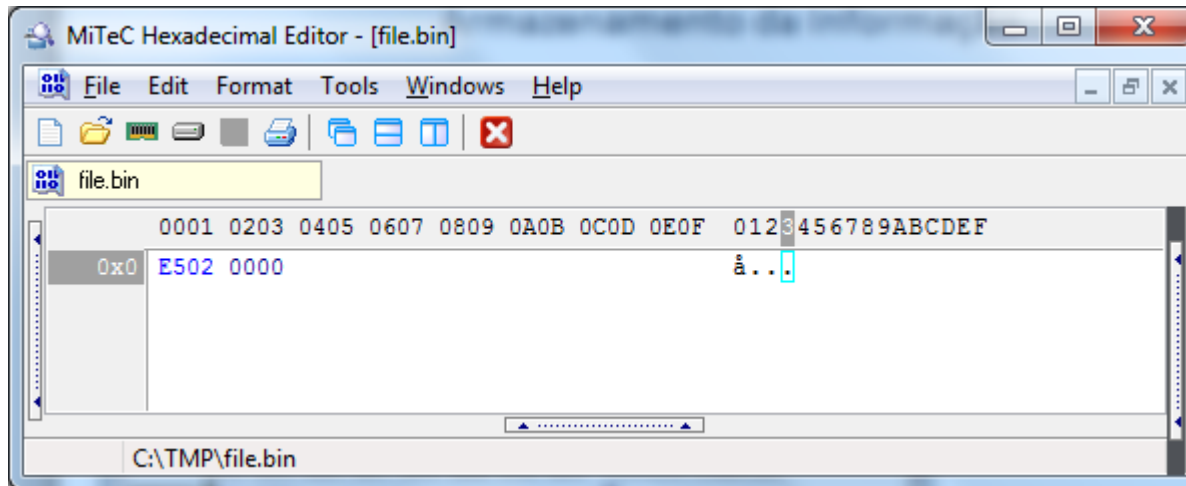
$741_{10} = 00\ 00\ 02\ E5h$  (4 byte)

MSB

LSB

MSB-Most Significant Byte

LSB-Least Significant Byte



C# → little-endian  
primeiro é guardado o byte  
de menor peso (LSB)

# Armazenamento da Informação

## 2) Valores numéricos: ficheiro de inteiros com o valor 741

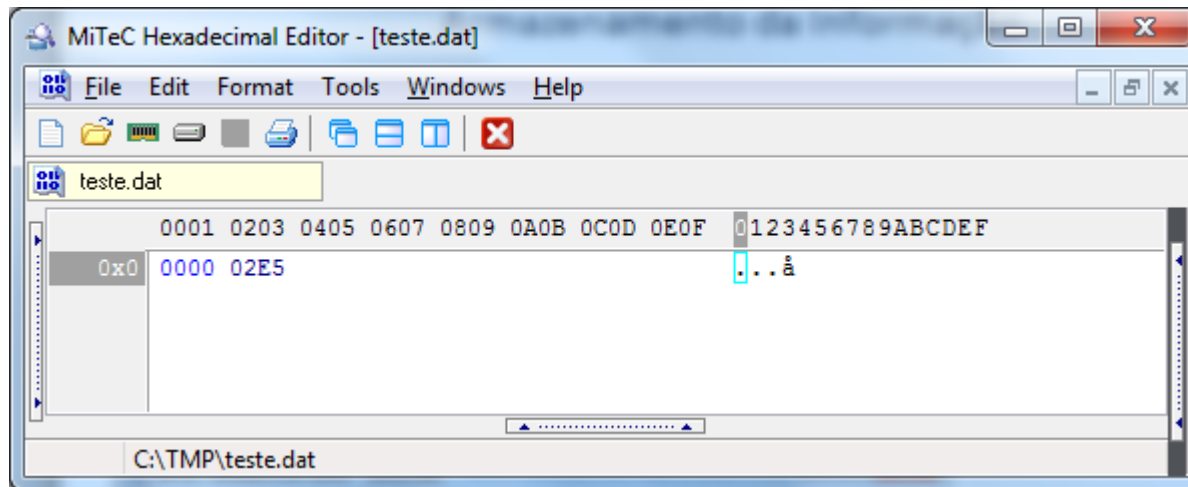
```
Java → FileOutputStream os = new FileOutputStream (new File ("teste.dat"), true);  
      DataOutputStream dos = new DataOutputStream (os);  
      dos.writeInt(741);
```

**int** → 32 bits(4 bytes) com sinal , -2.147.483.648 ↔ 2.147.483.647

$741_{10} = 00\ 00\ 02\ E5h$  (4 byte)

MSB                  LSB

MSB-Most Significant Byte      LSB-Least Significant Byte



Java → big-endian  
primeiro é guardado o byte  
de maior peso (MSB)

<http://mindprod.com/jgloss/endian.html>

<http://howtodoinjava.com/core-java/basics/little-endian-and-big-endian-in-java/>

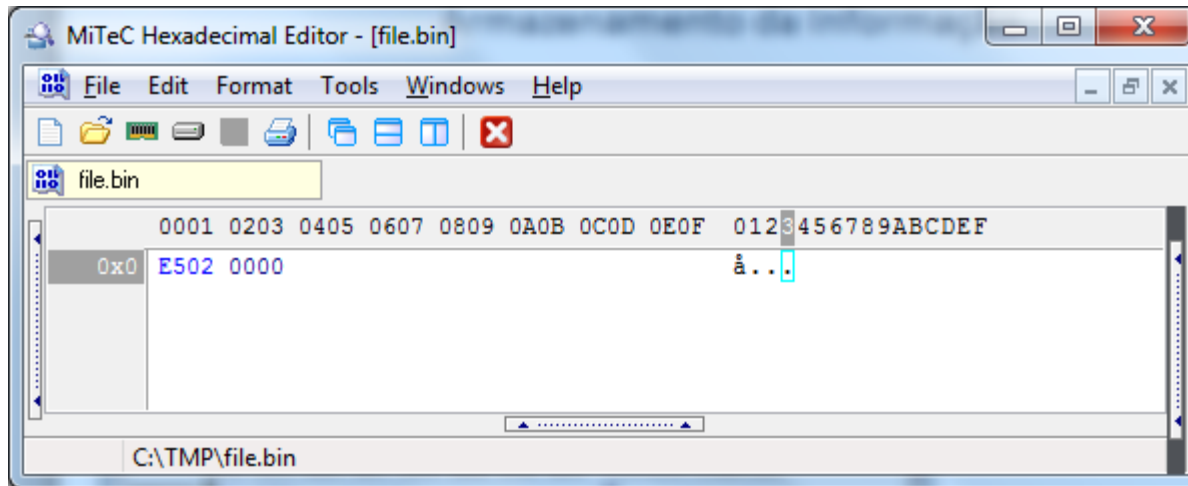
## Valores numéricos: ficheiro de inteiros com o valor 741

$741_{10} = 00\ 00\ 02\ E5h$  (4 byte)

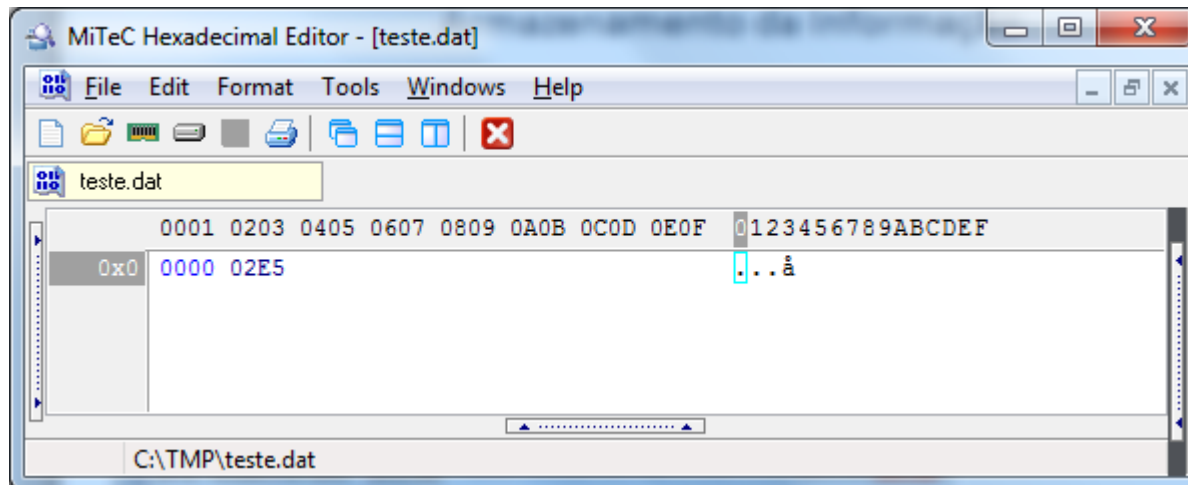
MSB      LSB

MSB-Most Significant Byte

LSB-Least Significant Byte



C# → little-endian  
primeiro é guardado o byte  
de menor peso (LSB)

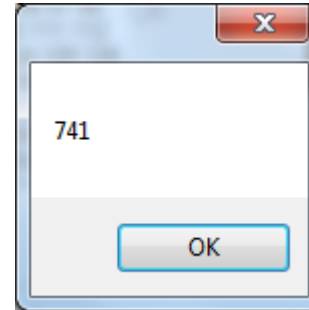


Java → big-endian  
primeiro é guardado o byte  
de maior peso (MSB)

# Armazenamento da Informação

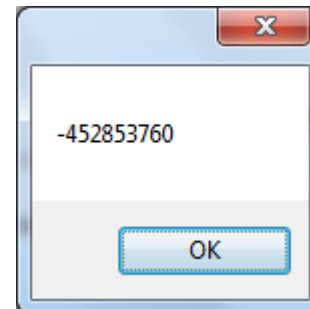
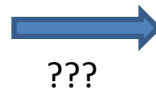
1) Ler com C# o ficheiro escrito em C# contendo o valor inteiro 741 ?

```
C# → using (BinaryReader b = new BinaryReader(File.Open("file.bin", FileMode.Open)))  
int v = b.ReadInt32();  
MessageBox.Show(v.ToString());
```



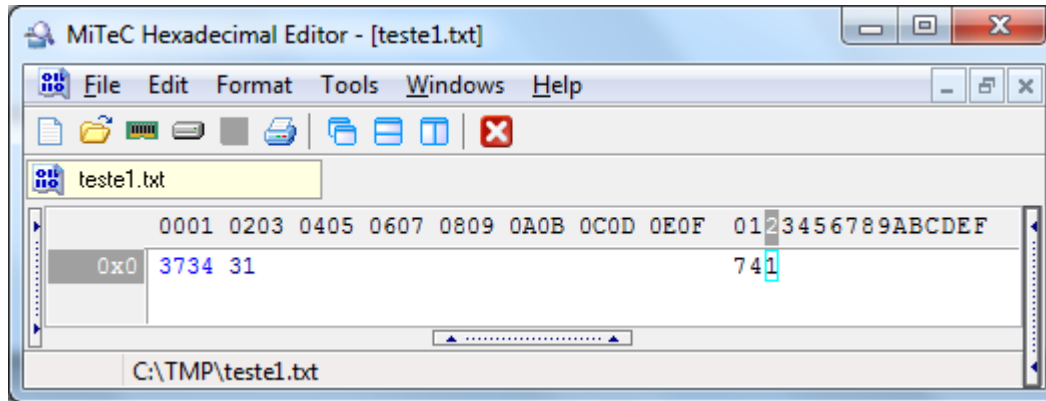
2) Ler com C# o ficheiro escrito em Java contendo o valor inteiro 741 ?

```
C# → using (BinaryReader b = new BinaryReader(File.Open("teste.dat", FileMode.Open)))  
int v = b.ReadInt32();  
MessageBox.Show(v.ToString());
```

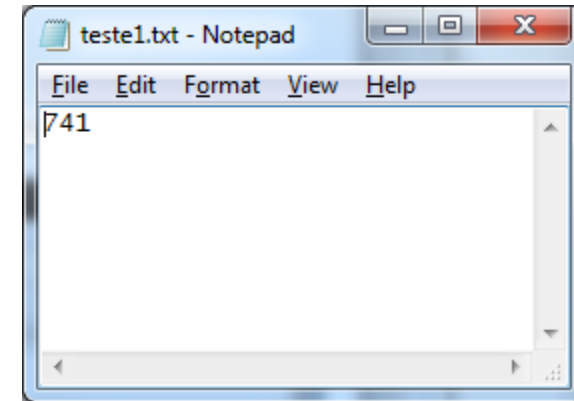


# Armazenamento da Informação

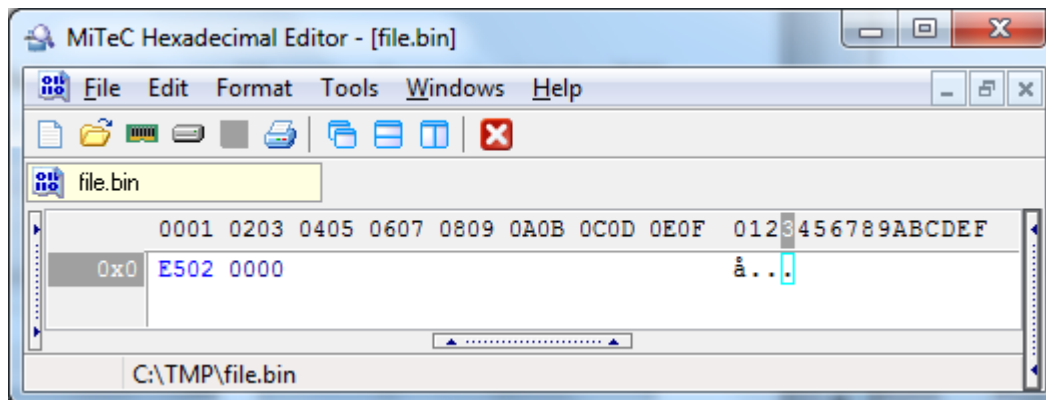
1) Ler com um processador de texto o ficheiro escrito em C# contendo a string (conjunto de caracteres) “741” ?



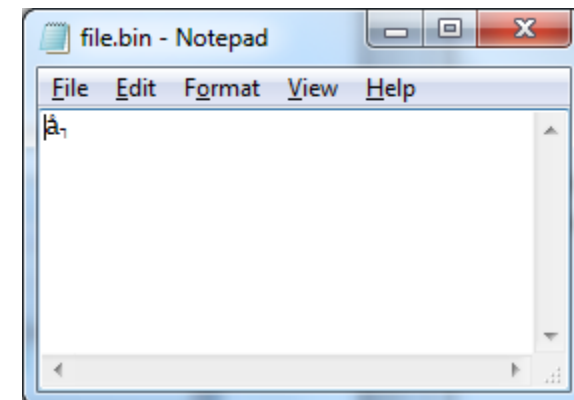
ASCII  
→  
OK!



2) Ler com um processador de texto o ficheiro escrito em C# contendo o inteiro ( em binário) 741 ?

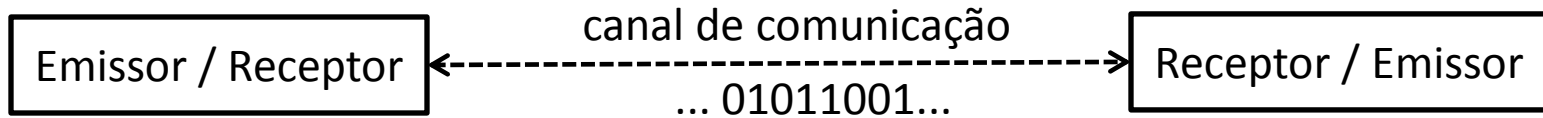


ASCII  
→  
???





# Tratamento de erros



A transmissão de informação ao longo de um canal de comunicação pode sofrer erros:

- transferência de dados entre registos dentro do CPU;
- entre o CPU e a memória;
- nas entradas/saídas de dispositivos periféricos (teclado, impressora);
- em comunicações em rede (cabos Ethernet, fibra óptica);

A existência de ruído pode provocar com que um bit mude de estado:  $1 \rightarrow 0$  ou  $0 \rightarrow 1$

ex: 01011001  $\rightarrow$  01001001 ou 01011001  $\rightarrow$  01011101 (num erro em rajada pode mudar mais de um bit)

O ruído (interferência) pode ter várias causas: interferência electromagnética, falha de energia, deficiência nos circuitos,...

## Abordagens

Deteção de erros: detectar que ocorreu um erro – em seguida a transmissão pode ser repetida;

Correcção de erros: detectar ocorrência do erro e efectuar a sua correcção (sem retransmissão);

Todos os métodos se baseiam na utilização de bits adicionais à mensagem original, designados por bits de paridade ou de verificação.

# Detecção de erros

Princípio: adicionar bits extra a cada bloco de dados a transmitir de modo que quando o bloco é recebido os bits extra são verificados para constatar se ocorreu ou não um erro.

**Bit de paridade** → conta-se o nº de bits “1” do bloco a transmitir e acrescenta-se um bit adicional, tal que:

- paridade par (EVEN) → nº total de bits “1” seja par
- paridade ímpar (ODD) → nº total de bits “1” seja ímpar

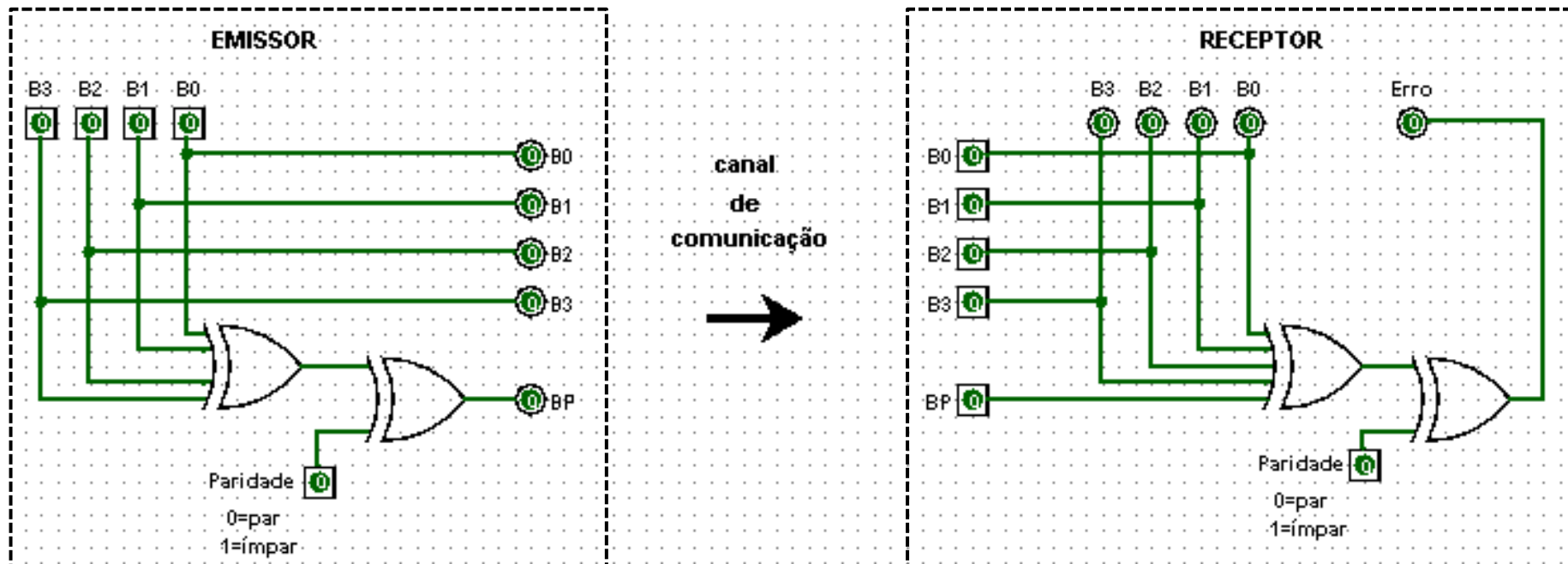
Dados binário	Paridade	
	Par	Ímpar
0000	0000 <b>0</b>	0000 <b>1</b>
0001	0001 <b>1</b>	0001 <b>0</b>
0010	0010 <b>1</b>	0010 <b>0</b>
0011	0011 <b>0</b>	0011 <b>1</b>
0100	0100 <b>1</b>	0100 <b>0</b>
0101	0101 <b>0</b>	0101 <b>1</b>
0110	0110 <b>0</b>	0110 <b>1</b>
0111	0111 <b>1</b>	0111 <b>0</b>
1000	1000 <b>1</b>	1000 <b>0</b>
1001	1001 <b>0</b>	1001 <b>1</b>

XOR = 1 → quando há um nº ímpar de 1's nas entradas  
= 0 → quando há um nº par de 1's nas entradas

paridade	bit de paridade
par	XOR entre os bits de dados
ímpar	$\overline{\text{XOR}}$ entre os bits de dados

# Detecção de erros

Circuito gerador/detector do bit de paridade para palavras de 4 bits



- Se durante a transmissão um número ímpar de bits for alterado (incluindo o próprio bit de paridade), a paridade altera-se e o erro é detectado.
- Se o número de bits alterados for par, a paridade não sofre alteração e o erro não é detectado.
- Este circuito detecta erro num bit mas não indica qual o bit errado → quando há erro os dados devem ser descartados e retransmitidos novamente.

# Detecção de erros

O bit de paridade encontra-se muito associado às comunicações série, tal como nas portas COM

COM → bits por segundo (baud rate) + Data bits + Parity

Bps: 4800, 9600, 19200...

Data Bits: 4 , 5 , 6 , 7 , 8

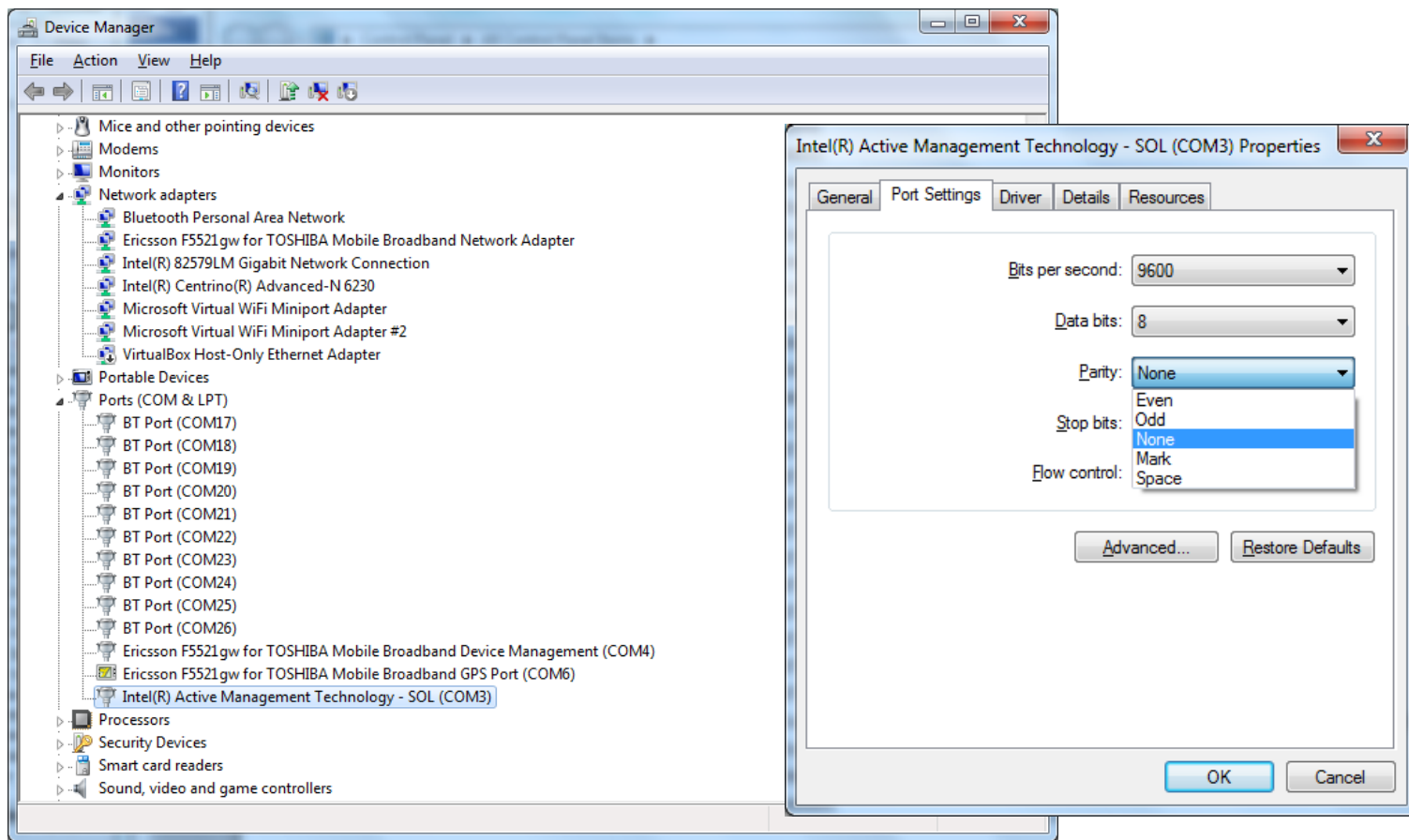
Parity: EVEN, ODD

NONE – nenhuma

MARK – sempre 1

SPACE – sempre 0

Windows > Control Panel > Device Manager



# Detecção e Correção de erros

- bit de paridade → permite detectar erros num certo grupo de bits;  
→ não permite corrigir o erro pois não se sabe qual o bit que o originou;
- Para detectar, identificar e corrigir o erro (em qual bit ou bits) terá de adicionar-se mais informação, ou seja, bits de paridade/verificação adicionais;
- Códigos de correção possibilitam recuperar o dado original a partir do código com erros;
- Consistem na inclusão de informação adicional que detecta situações inválidas mas que mantém a identidade do dado original;

# Código de Hamming

- faz uso do conceito de bit de paridade, mas além da detecção de um erro indica qual foi o bit no qual ele ocorreu, permitindo assim corrigir esse erro.

O código de paridade do código de Hamming é obtido a partir da palavra de dados, inserindo pontos de controlo, denominados bits de paridade.

Em cada palavra de dados, de comprimento  $n$  bits, são inseridos um número fixo  $k$ , de bits de paridade, ficando a palavra de código com um comprimento  $N = n + k$ , sendo:

$$N = 2^k - 1, n = N - k, n = 2^k - 1 - k$$

Ex:  $k = 3$  bits de paridade  $\rightarrow N = 7$  bits no total,  $n = 2^3 - 1 - 3 = 4$  bits de dados

$n$ bits de dados	$k$ bits de paridade	$N = n + k$ total de bits
1	2	3
4	3	7
11	4	15
26	5	31
57	6	63
120	7	127
247	8	255

Nota: determinar  $k$  a partir de  $n \rightarrow 2^k \geq k + n + 1$

# Código de Hamming

## Construção do código de Hamming

- 1) Numerar os bits a partir da esquerda → posição do bit : 1, 2, 3, 4, 5, 6, 7, 8, ...
- 2) Todas as posições que correspondem a potências de dois são bits de paridade ( $p_n$ )
- 3) Todas as outras posições são os bits de dados ( $d_n$ )

Regra para construção do código

		2 <sup>0</sup>	2 <sup>1</sup>		2 <sup>2</sup>				2 <sup>3</sup>								2 <sup>4</sup>					
Posição do bit		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
bits codificados		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
bits de paridade	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

## cada bit de paridade

→ é calculado a partir da função XOR entre os bits marcados na tabela

→ na verificação, os bits de paridade são recalculados e codificam em binário a posição do bit errado

- se,  $p_1 = p_2 = p_4 = p_8 = \dots = 0$  → todos os bits de paridade são zero → nenhum bit errado
- se,  $p_n \neq 0$  → algum bit de paridade não é zero → a posição do bit errado é dada pelo código  $p_8 p_4 p_2 p_1$   
(ex:  $p_8 p_4 p_2 p_1 = 0011$  → bit3[d1] → errado)

# Código de Hamming

## Emissor: geração do código de Hamming

Exemplo : n = 4 bits de dados (d1,d2,d3,d4) , k = 3 bits de paridade (p1,p2,p4) , N = 7 bits total  
palavra a codificar: **1011**

	2 <sup>0</sup>	2 <sup>1</sup>		2 <sup>2</sup>			
Posição do bit	1	2	3	4	5	6	7
bits codificados	p1	p2	d1	p4	d2	d3	d4
bits de paridade	p1	X		X		X	X
	p2		X	X		X	X
	p4			X	X	X	X
	p8						
	p16						

bits de dados

d1 d2 d3 d4 = **1011**



	2 <sup>0</sup>	2 <sup>1</sup>		2 <sup>2</sup>			
Posição do bit	1	2	3	4	5	6	7
bits codificados	p1	p2	<b>1</b>	p4	<b>0</b>	<b>1</b>	<b>1</b>
bits de paridade	p1	X		X		X	X
	p2		X	X		X	X
	p4			X	X	X	X
	p8						
	p16						

bits de paridade

$$p1 \rightarrow \text{bit3} \oplus \text{bit5} \oplus \text{bit7} = 1 \oplus 0 \oplus 1 = \mathbf{0}$$

$$p2 \rightarrow \text{bit3} \oplus \text{bit6} \oplus \text{bit7} = 1 \oplus 1 \oplus 1 = \mathbf{1}$$

$$p4 \rightarrow \text{bit5} \oplus \text{bit6} \oplus \text{bit7} = 0 \oplus 1 \oplus 1 = \mathbf{0}$$



	2 <sup>0</sup>	2 <sup>1</sup>		2 <sup>2</sup>			
Posição do bit	1	2	3	4	5	6	7
bits codificados	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>
bits de paridade	p1	X		X		X	X
	p2		X	X		X	X
	p4			X	X	X	X
	p8						
	p16						

palavra original : **1011** → palavra codificada : **0110011**



# Código de Hamming

## Recetor: verificação do código de Hamming

O recetor vai recalculer os bits de paridade - para tal usa os próprios bits de paridade que foram recebidos, além de bits de dados:

- Se todos os bits de paridade recalculados forem 0 (zero) então não houve erros na comunicação
- Se algum bit de paridade for 1 (um) então houve algum erro – nesse caso os bits de paridade codificam em binário a posição do bit errado

Exemplo1 : palavra recebida pelo canal = 0110011, conterá erros?



	2 <sup>0</sup>	2 <sup>1</sup>		2 <sup>2</sup>			
Posição do bit	1	2	3	4	5	6	7
bits codificados	p1	p2	d1	p4	d2	d3	d4
	0	1	1	0	0	1	1
bits de paridade	p1	X		X		X	X
	p2		X	X		X	X
	p4				X	X	X
	p8						
	p1						
	6						

bits de dados

d1 d2 d3 d4 = **1011**

bits de paridade

$$\begin{aligned} p1 &\rightarrow \text{bit1} \oplus \text{bit3} \oplus \text{bit5} \oplus \text{bit7} = 0 \oplus 1 \oplus 0 \oplus 1 \\ p2 &\rightarrow \text{bit2} \oplus \text{bit3} \oplus \text{bit6} \oplus \text{bit7} = 1 \oplus 1 \oplus 1 \oplus 1 \\ p4 &\rightarrow \text{bit4} \oplus \text{bit5} \oplus \text{bit6} \oplus \text{bit7} = 0 \oplus 0 \oplus 1 \oplus 1 \end{aligned}$$

0 0 0

bits de paridade todos zero → nenhum bit errado → dados são aceites !

# Código de Hamming

## Recetor: verificação do código de Hamming

Exemplo2 : palavra recebida pelo canal = 0110001 , conterá erros?



	2 <sup>0</sup>	2 <sup>1</sup>		2 <sup>2</sup>			
Posição do bit	1	2	3	4	5	6	7
bits codificados	p1	p2	d1	p4	d2	d3	d4
	0	1	1	0	0	0	1
bits de paridade	p1	X		X		X	
	p2		X	X		X	X
	p4				X	X	X
	p8						
	p16						

bits de dados

d1 d2 d3 d4 = **1001**

↓ inverter  
**1011**

bits de paridade

p1 → bit1 ⊕ bit3 ⊕ bit5 ⊕ bit7 = 0 ⊕ 1 ⊕ 0 ⊕ 1  
 p2 → bit2 ⊕ bit3 ⊕ bit6 ⊕ bit7 = 1 ⊕ 1 ⊕ 0 ⊕ 1  
 p4 → bit4 ⊕ bit5 ⊕ bit6 ⊕ bit7 = 0 ⊕ 0 ⊕ 0 ⊕ 1

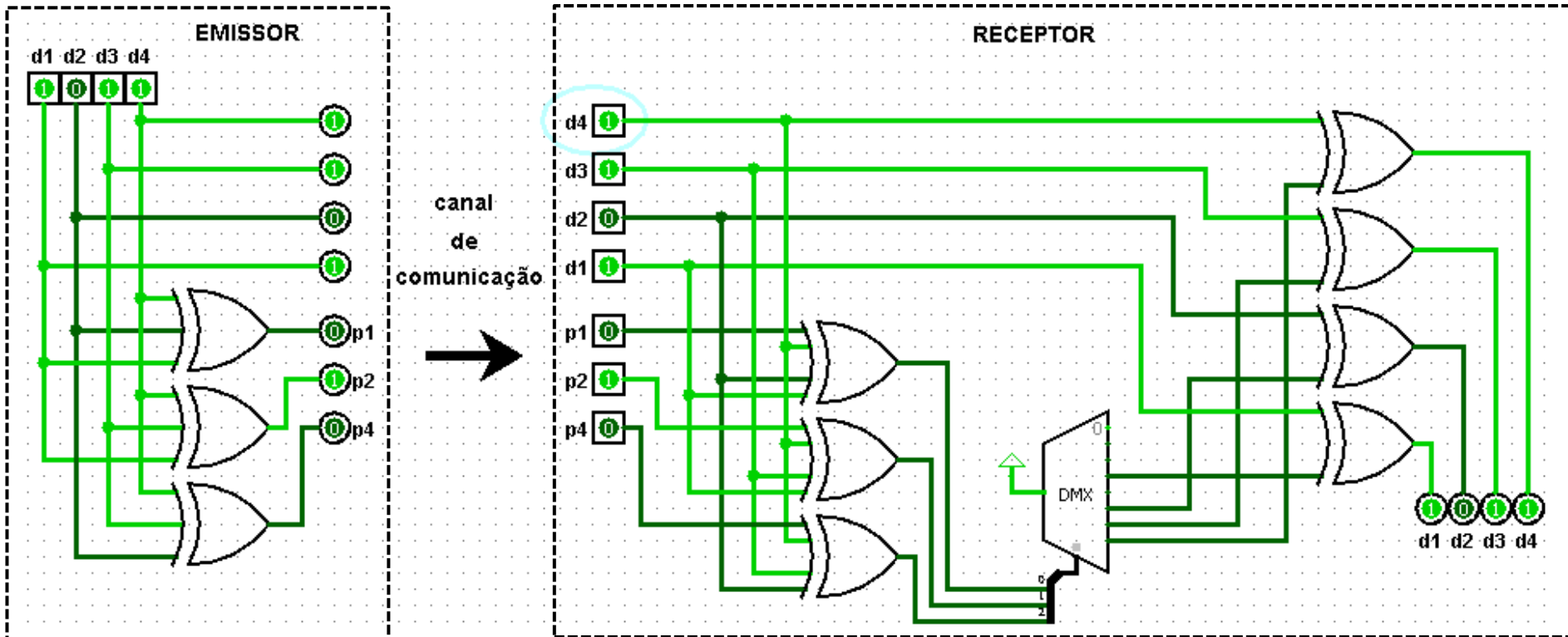
**1 1 0** (=6) → posição 6 = bit d3 → errado!

sabendo-se que um bit está errado (trocado), então é só invertê-lo

# Código de Hamming

Circuito gerador/detector do código de Hamming para palavras de 4 bits:

detecta e corrige um erro em um bit → assim não obriga a retransmissão



bits de paridade

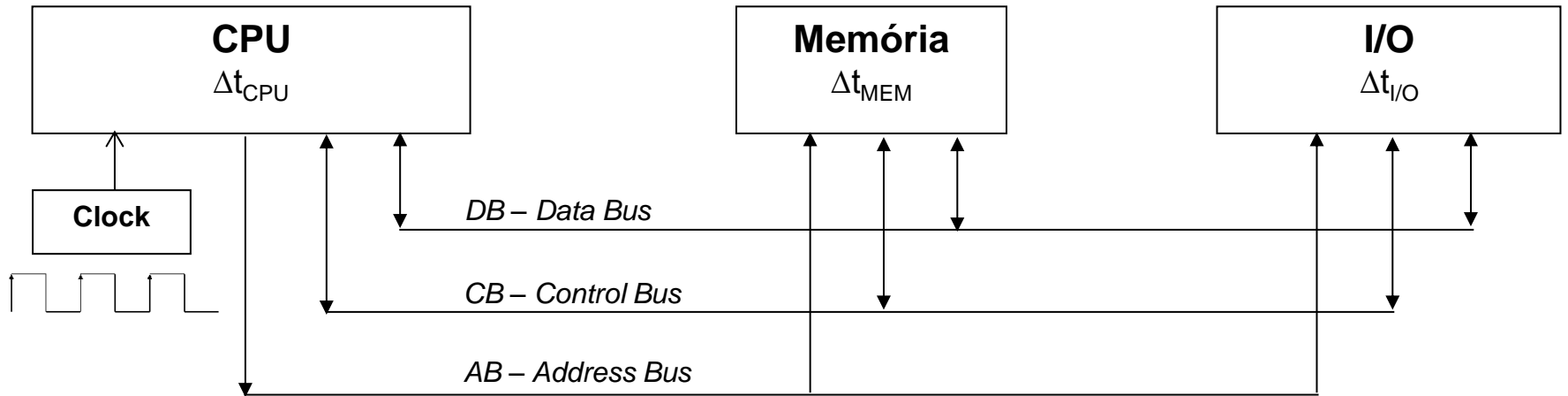
$$\begin{aligned} p1 &\rightarrow d1 \oplus d2 \oplus d3 \oplus d4 \\ p2 &\rightarrow d1 \oplus d3 \oplus d4 \\ p4 &\rightarrow d2 \oplus d3 \oplus d4 \end{aligned}$$

bits de paridade

$$\begin{aligned} p1 &\rightarrow p1 \oplus d1 \oplus d2 \oplus d4 \\ p2 &\rightarrow p2 \oplus d1 \oplus d3 \oplus d4 \\ p4 &\rightarrow p4 \oplus d2 \oplus d3 \oplus d4 \end{aligned}$$

bits paridade			bit errado
p4	p2	p1	
0	0	0	-
0	0	1	p1
0	1	0	p2
0	1	1	d1
1	0	0	p4
1	0	1	d2
1	1	0	d3
1	1	1	d4

# Avaliação do Desempenho

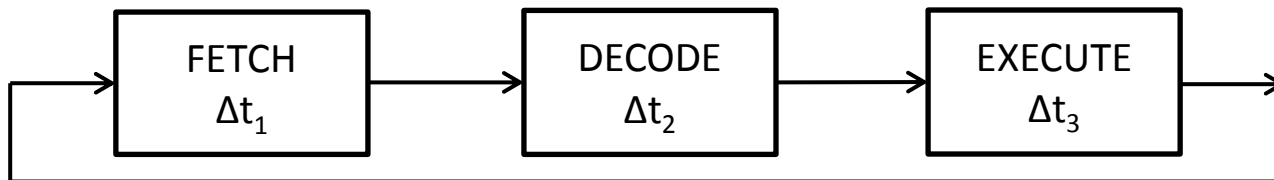


$\Delta t_{\text{CPU}}$  : tempo de CPU

$\Delta t_{\text{MEM}}$  : tempo de acesso memória

$\Delta t_{\text{I/O}}$  : tempo de acesso periféricos

CPU → ciclo : Fetch-Decode-Execute



Tempo gasto pelo CPU

$\Delta t_1$  – busca da instrução

$\Delta t_2$  – decodificação

$\Delta t_3$  – execução

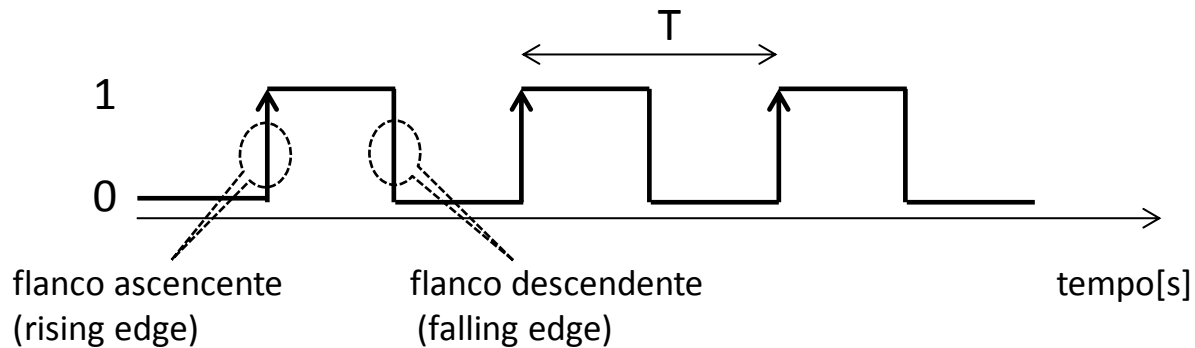
$$\Delta t_{\text{CPU}} = \Delta t_1 + \Delta t_2 + \Delta t_3$$

Tempo total para executar uma instrução :  $\Delta T = \Delta t_{\text{CPU}} + \Delta t_{\text{MEM}} + \Delta t_{\text{I/O}}$

Como avaliar (e depois diminuir)  $\Delta T$  ?

# Avaliação do Desempenho

Sinal de relógio (clock) : controla os tempos em que as instruções são executadas



$T$  = período(clock cycle) [s] : tempo que decorre entre dois acontecimentos iguais

$F$  = frequência(clock rate) [Hz] : nº de repetições de períodos por unidade de tempo

$F = 1 / T$  (frequência é o inverso do período) ,  $T = 1 / F$  (período é o inversa da frequência)

Unidade de Frequência : Hz (Hertz)

F - frequência		T - tempo	
[Hz]	Designação	[s]	Designação
1 Hz	Hertz = 1 período por segundo	1 s	segundo
1KHz = $10^3$ Hz	Kilo Hertz = $10^3$ periodos por segundo	$10^{-3}$ s = 0.001s = 1ms	mili segundo
1MHz = $10^6$ Hz	Mega Hertz = $10^6$ periodos por segundo	$10^{-6}$ s = 0.000001s = 1μs	micro segundo
1GHz = $10^9$ Hz	Giga Hertz = $10^9$ periodos por segundo	$10^{-9}$ s = 0.000000001s = 1ns	nano segundo

(ex: rede eléctrica doméstica →  $F=50\text{Hz}$ ,  $T=20\text{ms}$ )

# Avaliação do Desempenho

## Sinal de relógio

Em cada período T podem ser executadas um certo número (n) de instruções elementares

Se for possível executar essas n instruções num período T menor, então o desempenho melhora

$$\Delta t_{\text{CPU}} = f(T_{\text{clock}}) : \downarrow T \rightarrow \downarrow \Delta t_{\text{CPU}} \text{ (melhora o desempenho)}$$

ou

$$\Delta t_{\text{CPU}} = f(1/F_{\text{clock}}) : \uparrow F \rightarrow \downarrow \Delta t_{\text{CPU}} \text{ ( melhora o desempenho)}$$

$\downarrow T$  (ou  $\uparrow F$ ) :

- diminui o tempo para executar as instruções, podendo não ser possível de as executar;
- pode obrigar a mais períodos para executar as mesmas instruções;
- obriga a tornar as instruções mais eficientes (demorarem menos tempo) ;
- aumentar a frequência está condicionado por questões técnicas, como tempos de comutação das portas lógicas (transições  $0 \rightarrow 1$  e  $1 \rightarrow 0$ ), aumento da temperatura, etc;

Os fabricantes têm vindo a conseguir aumentar sucessivamente o valor de F:

1981 : IBM PC original..... 4.77 MHz

1995 : Pentium..... 100 MHz

2000 : AMD ..... 1 GHz

2002 : Pentium 4..... 3 GHz

(a partir daqui a evolução tem sido mais lenta)

# Avaliação do Desempenho

## Métricas (fracas) para avaliação do desempenho

**1) F - frequência** : maior F não implica necessariamente melhor desempenho, pois um CPU com menor  $F_{\text{clock}}$  pode ser mais eficiente que um com  $F_{\text{clock}}$  mais elevada, podendo realizar mais operações em menos tempo.

ex: CPU<sub>A</sub>: F=1GHz , 2 ciclos por instrução (média)

CPU<sub>B</sub>: F=1.5GHz , 3 ciclos por instrução (média)

Qual CPU tem melhor desempenho?

$$\begin{aligned}\text{CPU}_A \rightarrow T &= 1/F = 1/10^9 = 10^{-9}\text{s} = 1\text{ns} \\ t(\text{instrução}) &= 2 * T = 2 * 1 = 2\text{ns}\end{aligned}$$

$$\begin{aligned}\text{CPU}_B \rightarrow T &= 1/F = 1/(1.5*10^9) = 0.67*10^{-9}\text{s} = 0.67\text{ns} \\ t(\text{instrução}) &= 3 * T = 3 * 0.67 = 2.01\text{ns}\end{aligned}$$

Conclusão: CPU<sub>A</sub> é mais rápido que CPU<sub>B</sub> , apesar de uma menor  $F_{\text{clock}}$

**2) MIPS - Millions of Instructions Per Second** : normalmente é o tempo de execução de certas instruções (ex: NOP). No entanto os programas são constituídos por diversas outras instruções, pelo que devem ser usadas médias ponderadas.

**3) MFLOPS-Millions of Floating-Point Operations Per Second** : número máximo de instruções de vírgula flutuante (não inteiros) por segundo. No entanto, a maioria dos programas não faz uso intensivo destas instruções.

# Avaliação do Desempenho

## Tempo de execução

- única medida completa e de confiança para avaliar o desempenho
- a máquina que executa o mesmo trabalho em menos tempo é a mais rápida

Para um certo programa executando na máquina X

$$\text{DesempenhoX} = \frac{1}{\text{Tempo}_{\text{execuçãoX}}} \quad ( \downarrow \text{Tempo}_{\text{execuçãoX}} \rightarrow \uparrow \text{DesempenhoX} )$$

Para duas máquinas X e Y

$$\text{DesempenhoX} > \text{DesempenhoY} \rightarrow \frac{1}{\text{Tempo}_{\text{execuçãoX}}} > \frac{1}{\text{Tempo}_{\text{execuçãoY}}} \rightarrow \text{Tempo}_{\text{execuçãoX}} < \text{Tempo}_{\text{execuçãoY}}$$

Se a máquina X é N vezes mais rápida que a máquina Y

$$\text{DesempenhoX} = N * \text{DesempenhoY} \rightarrow \frac{\text{DesempenhoX}}{\text{DesempenhoY}} = \frac{\text{Tempo}_{\text{execuçãoY}}}{\text{Tempo}_{\text{execuçãoX}}} = N \rightarrow \text{Tempo}_{\text{execuçãoX}} = \frac{\text{Tempo}_{\text{execuçãoY}}}{N}$$

Ex: máquina X  $\rightarrow$   $\text{Tempo}_{\text{execuçãoX}}$  , para uma certa tarefa = 10s

máquina Y  $\rightarrow$   $\text{Tempo}_{\text{execuçãoY}}$  , para a mesma tarefa = 15s

$$N = \frac{\text{DesempenhoX}}{\text{DesempenhoY}} = \frac{\text{Tempo}_{\text{execuçãoY}}}{\text{Tempo}_{\text{execuçãoX}}} = \frac{15}{10} = 1.5$$

$$\text{DesempenhoX} = 1.5 * \text{DesempenhoY} \quad , \quad \text{Tempo}_{\text{execuçãoX}} = \frac{\text{Tempo}_{\text{execuçãoY}}}{1.5}$$

$\rightarrow$  a máquina X é 1.5 vezes mais rápida que máquina Y



# Avaliação do Desempenho

Tempo total para executar uma tarefa:  $\Delta T = \Delta t_{\text{CPU}} + \Delta t_{\text{MEM}} + \Delta t_{\text{I/O}}$

vamos considerar apenas a fracção referente ao CPU ( $\Delta t_{\text{CPU}}$ )

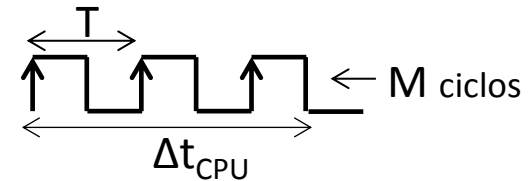
$\Delta t_{\text{CPU}} = \text{Tempo}_{\text{execução}} = \text{tempo que o CPU leva para executar um programa}$

$$= M(\text{ciclos clock}) * T_{\text{clock}}(\text{clock cycle})$$

$$= M(\text{ciclos clock}) / F_{\text{clock}}(\text{clock rate})$$

$F_{\text{clock}} \rightarrow$  conhecida

$M = ?$  (quantos ciclos de clock correspondem a um certo programa ?)



## Programa

- N instruções no total
- cada instrução precisa de um certo valor médio de periodos de clock para ser executada:  
CPI (Clocks Per Instruction) : nº médio ponderado de clocks para cada instrução

$$M = N * CPI \rightarrow \boxed{\text{Tempo}_{\text{execução}} = N * CPI / F_{\text{clock}}} \rightarrow 3 \text{ factores condicionantes}$$

- 1) N : depende do programador e do compilador, quanto melhor programador/compilador menor N
- 2) CPI : depende da arquitectura do processador, necessita de mais ou menos clocks por instrução
- 3) F : depende do hardware , a electrónica permite uma frequência de relógio maior ou menor

# Avaliação do Desempenho

## Exemplo

Disponemos de duas máquinas diferentes mas da mesma arquitectura (x86):

CPU<sub>A</sub> : F<sub>A</sub>=500MHz , CPI<sub>A</sub>=1.2      CPU<sub>B</sub> : F<sub>B</sub>=800MHz , CPI<sub>B</sub>=2

clock da máquina A (F<sub>A</sub>) < clock da máquina B (F<sub>B</sub>)

CPI<sub>A</sub> < CPI<sub>B</sub> → a máquina A é mais eficiente que a máquina B (aproveita melhor o clock)

Qual a máquina mais rápida a executar um determinado programa?

$$\text{Tempo}_{\text{execução}} = N * \text{CPI} / F_{\text{clock}}$$

Programa → N instruções

$$\text{Tempo}_{\text{execuçãoA}} = N * \text{CPI}_A / F_A = N * 1.2 / 500 * 10^6$$

$$\text{Tempo}_{\text{execuçãoB}} = N * \text{CPI}_B / F_B = N * 2 / 800 * 10^6$$

$$\text{DesempenhoA} / \text{DesempenhoB} = \text{Tempo}_{\text{execuçãoB}} / \text{Tempo}_{\text{execuçãoA}} = n$$

$$\text{Tempo}_{\text{execuçãoB}} / \text{Tempo}_{\text{execuçãoA}} = \frac{N * 2 / 800 * 10^6}{N * 1.2 / 500 * 10^6} \rightarrow n = 1.04 \rightarrow \text{máquina A é mais rápida que B}$$

(apesar do menor clock)

# Avaliação do Desempenho

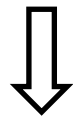
De que modo aperfeiçoamentos introduzidos na arquitectura de um CPU melhoram o seu desempenho?

- Cada aperfeiçoamento introduzido vai ser responsável por melhorar apenas um aspecto do funcionamento do CPU  
ex: uma unidade de multiplicação por hardware acelera as multiplicações (e talvez as divisões) mas em nada afeta os acessos à memória;
- As preocupações com as melhorias devem centrar-se nos casos mais comuns (mais frequentes) tornando mais rápida (mais optimizada) a sua execução;

## Lei de Amdahl

*O ganho de desempenho que pode ser obtido melhorando uma determinada parte do sistema é limitado pela fração de tempo em que essa parte é utilizada durante a operação.*

Tempo<sub>antigo</sub> : tempo anterior à introdução de um aperfeiçoamento (ou melhoramento)



introdução de um aperfeiçoamento

Tempo<sub>novo</sub> = Tempo<sub>melhorado</sub> + Tempo<sub>inalterado</sub> : tempo após a introdução do aperfeiçoamento

Tempo<sub>melhorado</sub> : tempo afectado pela introdução do aperfeiçoamento

Tempo<sub>inalterado</sub> : tempo não afectado pela introdução do aperfeiçoamento

# Avaliação do Desempenho

## Lei de Amdahl

$$\text{Aceleração}_{\text{global}} (\text{speedup}) = \text{Desempenho}_{\text{novo}} / \text{Desempenho}_{\text{antigo}} = \text{Tempo}_{\text{antigo}} / \text{Tempo}_{\text{novo}}$$

$$= \frac{1}{(1 - \text{Fracção}_{\text{melhorada}}) + \frac{\text{Fracção}_{\text{melhorada}}}{\text{Aceleração}_{\text{melhorada}}}}$$

$$\text{Tempo}_{\text{novo}} = \text{Tempo}_{\text{antigo}} * \left( (1 - \text{Fracção}_{\text{melhorada}}) + \frac{\text{Fracção}_{\text{melhorada}}}{\text{Aceleração}_{\text{melhorada}}} \right)$$

$\text{Aceleração}_{\text{global}}$  (speedup) : aceleração final do sistema

$\text{Desempenho}_{\text{antigo}}$  : desempenho anterior à introdução do aperfeiçoamento

$\text{Desempenho}_{\text{novo}}$  : desempenho após a introdução do aperfeiçoamento

$\text{Fracção}_{\text{melhorada}}$  : fracção de tempo em que o aperfeiçoamento é usado

$\text{Aceleração}_{\text{melhorada}}$  : aceleração obtida quando o aperfeiçoamento é usado (seria a aceleração global se esse aperfeiçoamento fosse usado o tempo todo)

# Avaliação do Desempenho

## Exemplo

Uma arquitectura não tem suporte hardware para multiplicações fazendo-as por software usando adições sucessivas (ex:  $3*2 = 2+2+2$ )

Admitindo que:

- uma multiplicação por software consome  $M=200$  ciclos de relógio
- uma multiplicação por hardware consome  $M=4$  ciclos de relógio

Calcule a aceleração produzida pela introdução de uma unidade de multiplicação por hardware, nos casos:

- a) se um programa gasta 10% do seu tempo em multiplicações
- b) se um programa gasta 40% do seu tempo em multiplicações

## Resolução

a)  $Aceleração_{melhorada} = \text{Tempo}_{antigo} / \text{Tempo}_{novo} = 200 / 4 = 50$  (aceleração apenas da parte melhorada)

$$\text{Fracção}_{melhorada} = 10\% = 0.1$$

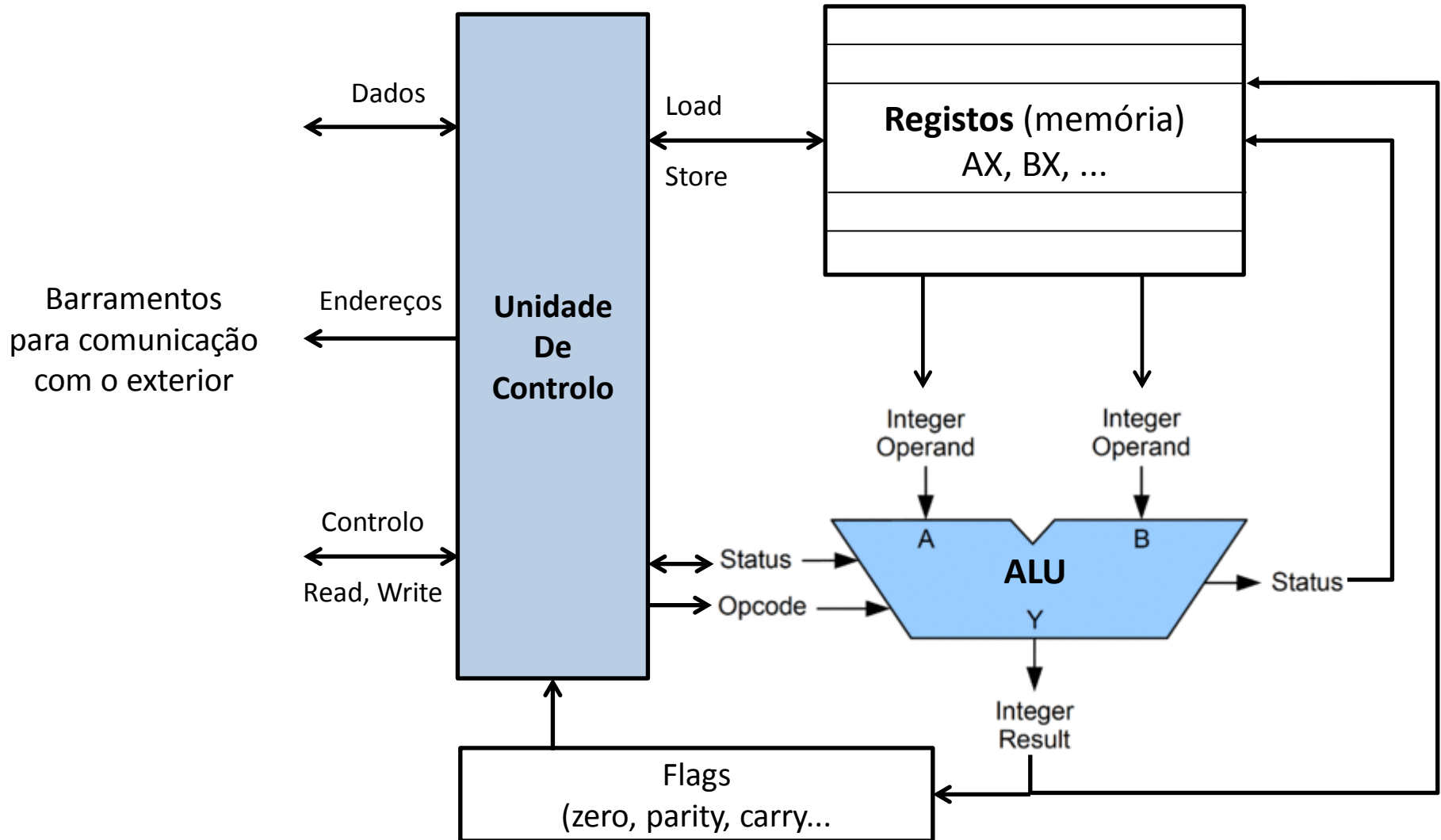
$$Aceleração_{global} = \frac{1}{(1 - \text{Fracção}_{melhorada}) + \frac{\text{Fracção}_{melhorada}}{Aceleração_{melhorada}}} = \frac{1}{(1-0.1) + 0.1/50} = 1.11 \rightarrow 11\%$$

b)  $Aceleração_{melhorada} = \text{Tempo}_{antigo} / \text{Tempo}_{novo} = 200 / 4 = 50$  (aceleração apenas da parte melhorada)

$$\text{Fracção}_{melhorada} = 40\% = 0.4$$

$$Aceleração_{global} = \frac{1}{(1 - \text{Fracção}_{melhorada}) + \frac{\text{Fracção}_{melhorada}}{Aceleração_{melhorada}}} = \frac{1}{(1-0.4) + 0.4/50} = 1.64 \rightarrow 64\% (>> 11\%)$$

# Estrutura dos Processadores (CPU)



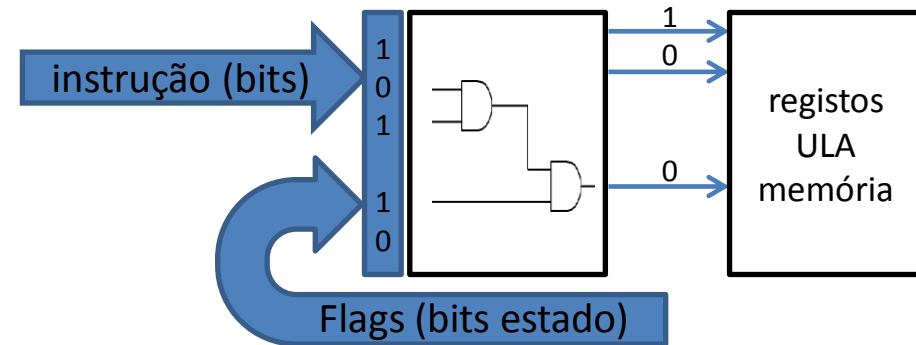
# Técnicas de melhoria do desempenho : unidade de controlo

## Unidade de controlo :

- responsável por obter e decodificar as instruções
- gerar os sinais que indicam aos restantes blocos as acções a executar, em função das instruções do programa e do estado do processador (flags)

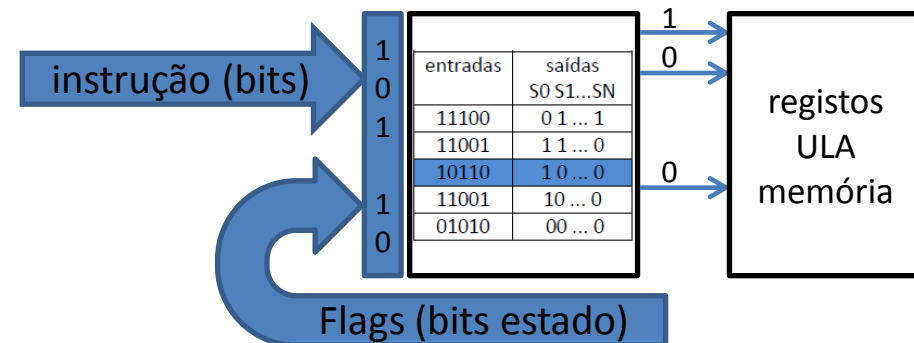
**Hardwired** (por hardware): os diversos sinais de controlo são gerados por circuitos lógicos (portas lógicas AND, OR, NOT, etc) segundo uma certa tabela de verdade e em função da instrução a executar:

- rápidas
- custo elevado
- alterações na arquitetura implicam alterações no hardware
- usada em máquinas RISC



**Microcoded** (microprogramado): existe uma tabela em memória (micromemória) interna ao processador, que contém os sinais a gerar em função da instrução a executar (palavra de controlo)

- mais lentas, pois dependem da rapidez da micromemória
- alterações na arquitetura apenas alteram o conteúdo da micromemória
- usada em máquinas CISC



# Técnicas de melhoria do desempenho : CISC-RISC

Inicialmente:

- computadores eram programados quase exclusivamente em linguagem *Assembly* (linguagem máquina);
- não havia linguagens de programação nem compiladores evoluídos;
- memórias eram lentas e caras;
- processadores incluíam o maior número de funcionalidades na expectativa de que os compiladores as iriam utilizar;

John Cocke (IBM, 1974) & D. Patterson, descobriram que:

- a maior parte dos programas envolve poucas instruções, relativamente ao total de instruções disponíveis no processador ;  
Regra dos 80/20 : 20% das instruções fazem 80% do trabalho(um conjunto pequeno de instruções realiza a maior parte do trabalho);
- Instruções mais simples são as mais frequentes (ex: mov a,b);
- As instruções mais complexas pouco ou nunca eram geradas pelos compiladores;
- As instruções complexas complicavam toda a arquitetura, obrigando a diminuir a frequência do relógio (levando a que as instruções mais simples ficassem mais lentas);

Frequência das instruções

- mov: 33%
- salto: 20%
- aritméticas/logicas: 16%
- outras: 0.1% - 10%

Objectivo: minimizar o tempo de execução do conjunto de instruções mais frequentes (20%) e mesmo substituir as restantes( 80%) por combinações mais rápidas daquelas, reduzindo e simplificando o total de instruções do CPU  
→ isto conduziu à filosofia RISC.



# Técnicas de melhoria do desempenho : CISC-RISC

## Principais arquiteturas de processadores

### **CISC (*Complex Instruction Set Computer*)**

- Grande número de instruções (~ 200 a 300) capazes de efectuar acções complexas, embora relativamente lentas, exigindo múltiplos ciclos de clock para serem executadas;
- Muitos modos de acesso à memória (directo, indirecto, imediato,...);
- Instruções de tamanho variável de acordo com o modo de endereçamento;
- Unidade de controlo baseada em “Microcoded” (lento);
- Exs: *Intel 80x86 , Motorola 68K*;

### **RISC (*Reduced Instruction Set Computer*)**

- Princípios : hardware mais simples e optimização do caso mais frequente;
- Unidade de controlo baseada em “*Hardwired*” (rápido);
- Conjunto reduzido de instruções (~50) simples e rápidas de executar;
- Permitem uma maior frequência de relógio (clock);
- Poucos modos de acesso à memória (baseado em LOAD/STORE);
- Cada instrução CISC dá origem a várias instruções RISC;
- Exs: MIPS(*Microprocessor without Interlocking Pipe Stages* ), microcontroladores PIC (Microhip), Motorola PowerPC (*Performance Optimization With Enhanced RISC – Performance Computing*);

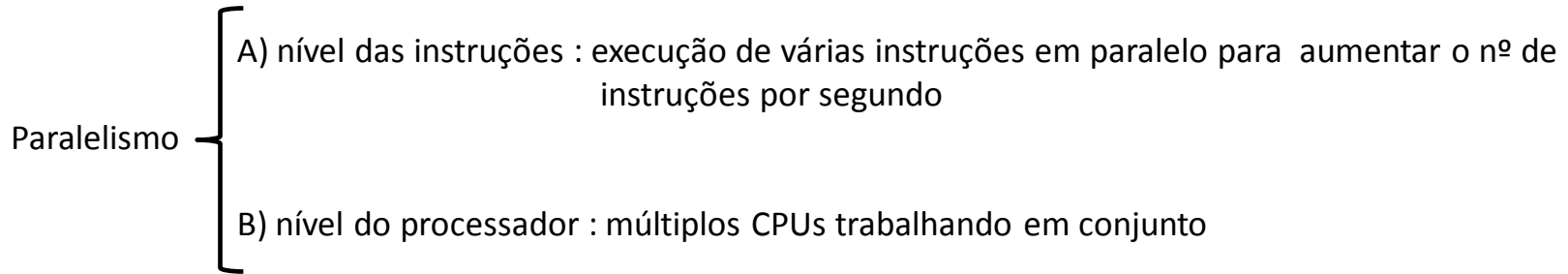
Actualmente muitos processadores adoptam uma estrutura híbrida CISC/RISC (ex.Pentium):

instruções frequentes -----> RISC : *hardwired*

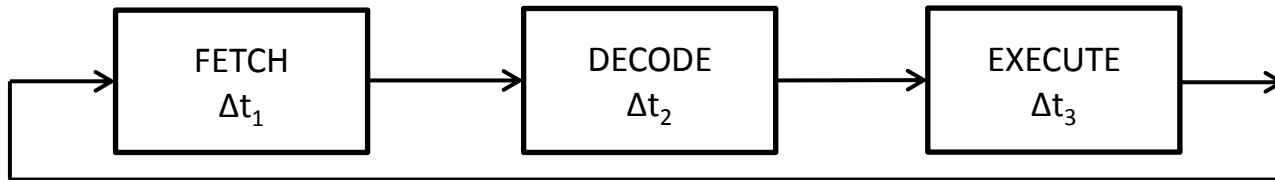
instruções menos frequentes ---> CISC : *microcoded*

Permitem um compromisso em termos de desempenho mantendo a compatibilidade com sistemas antigos

# Técnicas de melhoria do desempenho : paralelismo



## A) Paralelismo ao nível das instruções

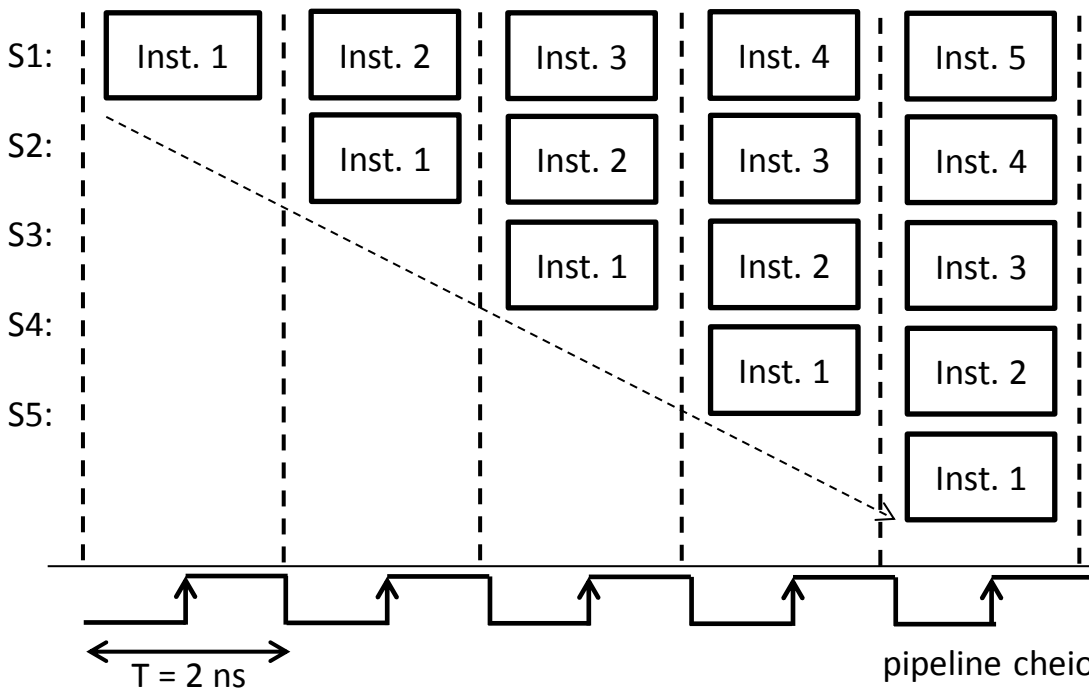
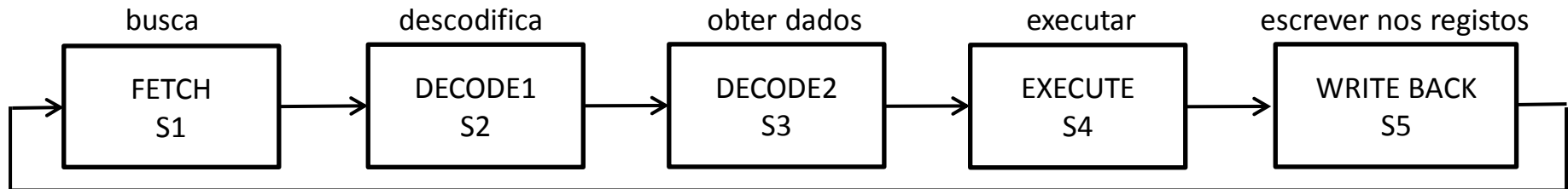


Executando apenas uma instrução de cada vez, só trabalha um bloco em cada instante  
→ mau aproveitamento!

Interessa fazer como numa linha de fabrico: cada operário executa uma operação passando o trabalho ao operário seguinte e iniciando de imediato uma nova operação.

# Técnicas de melhoria do desempenho : pipeline

**Pipeline:** permite o processamento simultâneo de múltiplas instruções, cada uma em estágios de processamento diferentes  
exs: Pentium: pipeline de 5 estágios    P IV: pipeline de 20 estágios



## Sem pipeline

1 instrução  $\rightarrow 5 \cdot T = 5 \cdot 2\text{ns} = 10\text{ns} = 10 \cdot 10^{-9} = 10^{-8}\text{s}$

$F = 1/10^{-8} = 10^8 = 100 \cdot 10^6 = 100 \text{ MIPS}$

CPI(Clocks Per Instruction) = 5

## Com pipeline (cheio)

1 instrução a cada  $T = 2\text{ns} = 2 \cdot 10^{-9} = 0.2 \cdot 10^{-8}\text{s}$

$F = 1/(0.2 \cdot 10^{-8}) = 500 \text{ MIPS}$  (5 vezes mais)

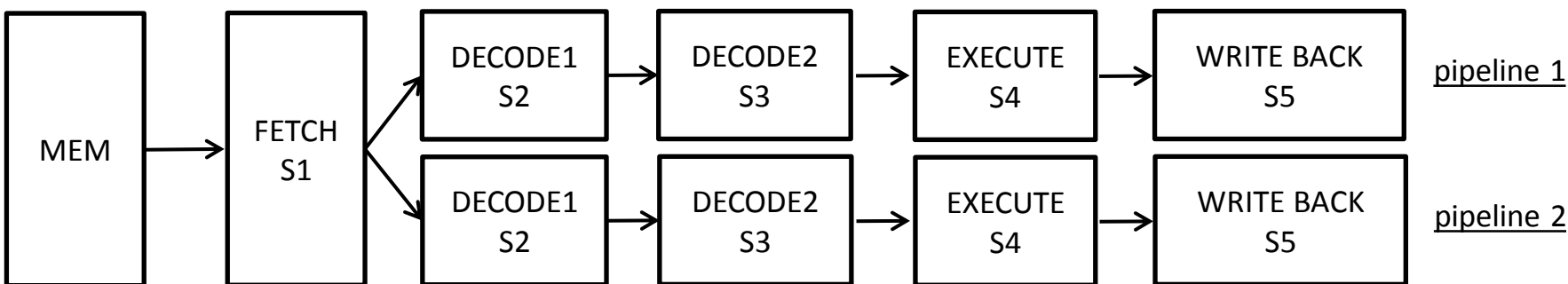
CPI(Clocks Per Instruction) = 1

(cada instrução demora à mesma  $5 \cdot T$ )

# Técnicas de melhoria do desempenho : estruturas superescalares

Superescalar: o hardware permite extrair paralelismo ao nível das instruções em programas sequenciais

Pipeline duplo(se um é bom dois é melhor...): duas instruções executadas de uma cada vez



## Problemas (hazards) do pipeline superescalar

1) Dependência de dados : quando uma instrução precisa de dados de outra instrução tem de parar à espera dessa outra:

pipeline 1 → mul bl ; ax = ah \* bl

pipeline 2 → add [var], ax ; [var] = [var] + ax : tem de esperar que o mul termine antes de poder continuar

2) Dependência de recursos : duas instruções acessam à mesma posição de memória ou precisam de usar o mesmo registo:

pipeline 1 → mul bl ; ax = ah \* bl

pipeline 2 → mov ah, 2 ; ah = 2 : mais rápida , se terminar primeiro afecta a instrução mul

3) Instruções de salto(não sequencialidade) - uma instrução de salto pode invalidar outras instruções que estão no pipeline:

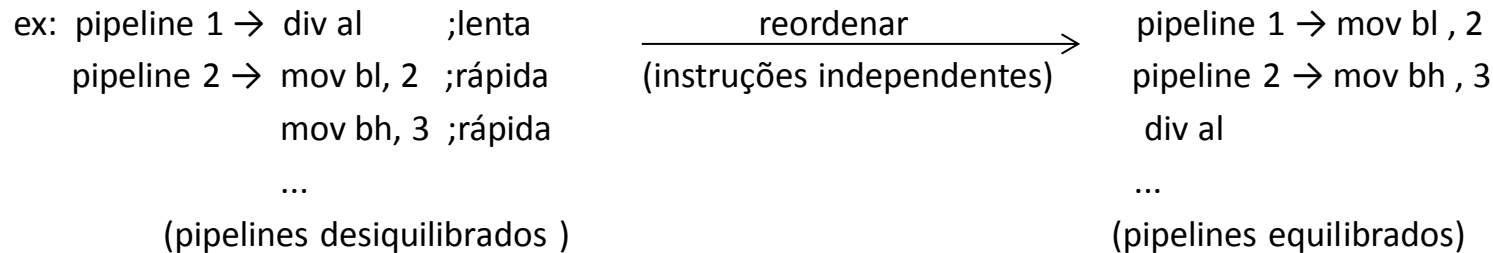
pipeline 1 → jz label { se ocorrer salto para label, a instrução sub (que está a ser executada no pipeline 2)

pipeline 2 → sub ax, 2 { terá de ser descartada e substituída pelo add

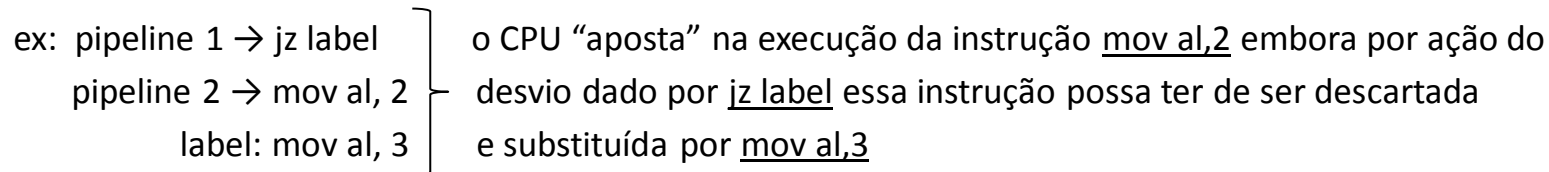
label: add ax, 2

# Técnicas de melhoria do desempenho

**Execução fora de ordem (out-of-order)** : as instruções são executadas por ordem diferente daquela em que aparecem no programa, desde que o resultado não seja alterado → exige um hardware complexo.



**Execução especulativa** : as instruções são executadas em sequência mesmo que não venham a ser usadas por ação de um desvio.



(os CPUs implementam mecanismos de previsão se o desvio vai acontecer ou não)

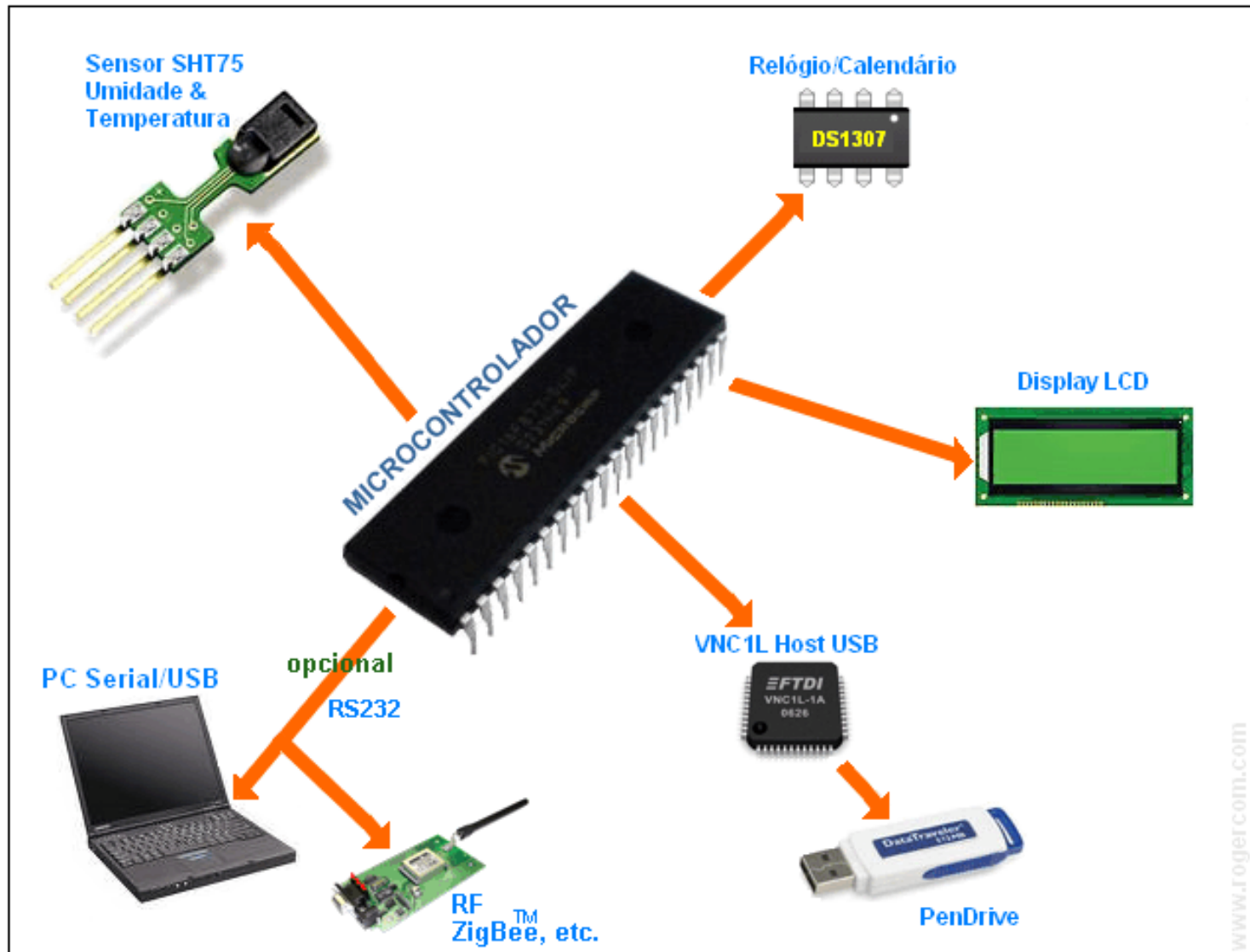
**Técnicas de compilação** : os compiladores também são responsáveis por explorar as melhores características do paralelismo do hardware.

ex: desfazer ciclos → aumentar o nº de operações independentes dentro de uma iteração

for (i=0; i<100; i++) → ciclo executa 100 vezes  
a[i] = b[i] + c[i]; → pipeline 1 : só um pipeline é usado

for (i=0; i<100; i+=2) → ciclo executa 50 vezes  
a[i] = b[i] + c[i]; → pipeline 1  
a[i+1] = b[i+1] + c[i+1]; → pipeline 2

# Microprocessadores vs Microcontroladores

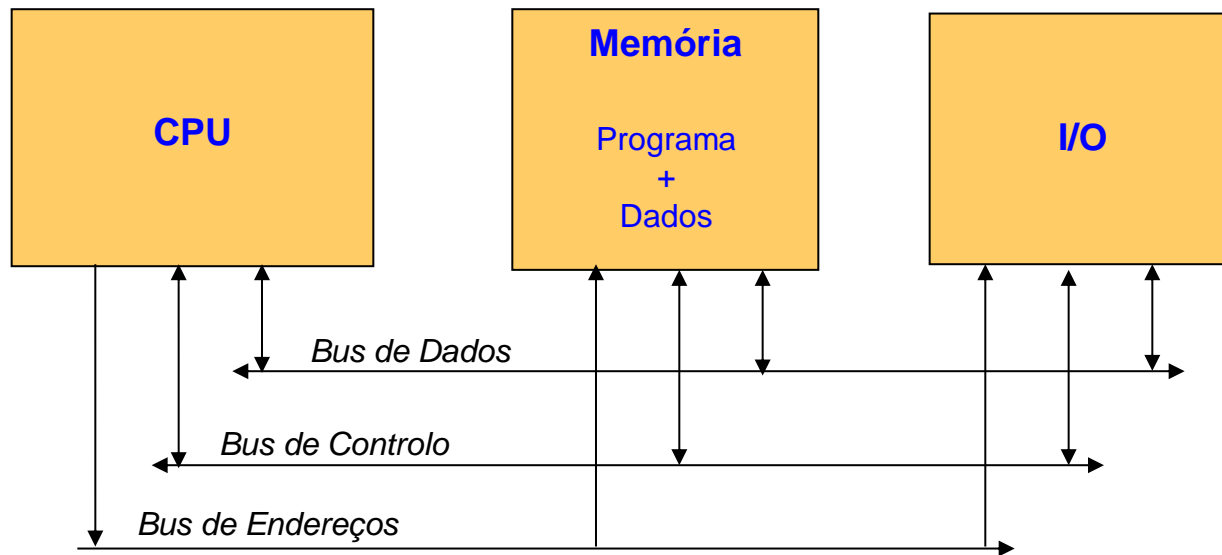


# Microprocessadores - Arquitetura de Von Neuman



John von Neumann

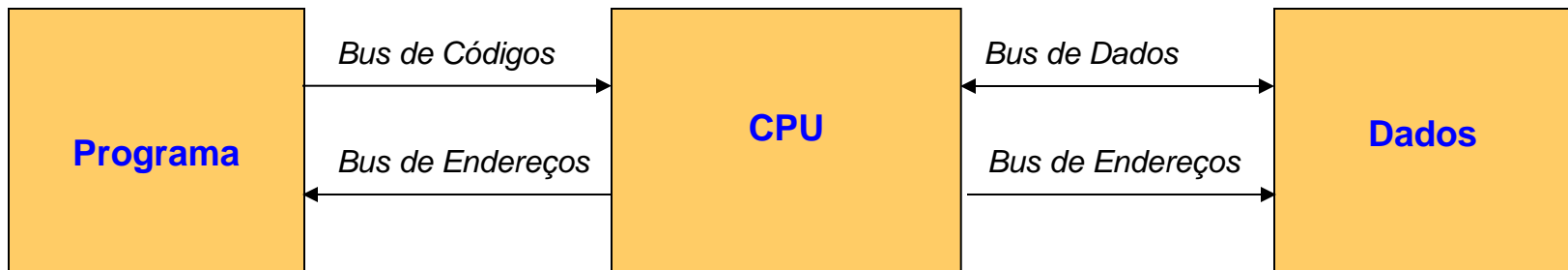
- instruções e dados compartilham a mesma unidade física de memória
- CISC – Complex Instruction Set Computer (mas também RISC)
- A vantagem é a simplicidade de acesso à memória - possui um barramento único para aceder à memória (endereços, dados e controlo)
- O grande inconveniente é o facto da memória do programa e dos dados ser comum, pois impede que se possa aceder ao programa e aos dados simultaneamente e muitas vezes o tamanho dos dados é diferente do tamanho das instruções



Utilização genérica

# Microcontroladores - Arquitetura de Harvard

- instruções e dados são armazenados em memórias diferentes
- RISC – Reduced Instruction Set Computer
- vantagens: instruções mais rápidas → instruções e dados podem ser acedidos simultaneamente → aumento do desempenho!



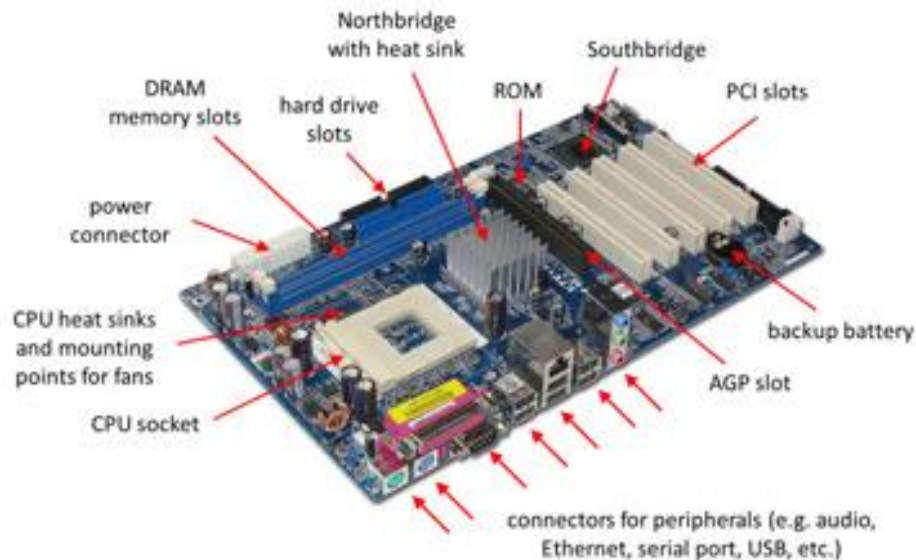
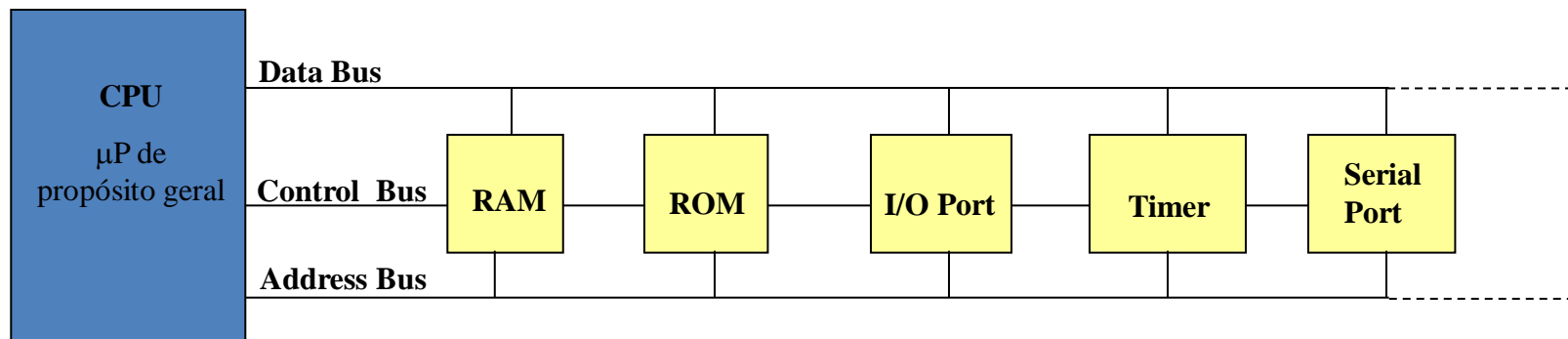
Utilização específica



# Microprocessadores

## Sistema microprocessador de propósito geral...

- CPU para computadores de propósito geral
- RAM-ROM, I/O, Portas, Timers, A/D & D/A... são exteriores ao CPU
- exemplo : Intel x86(Pentium), Motorola 680x0



Diversos chips na motherboard

# Microcontroladores

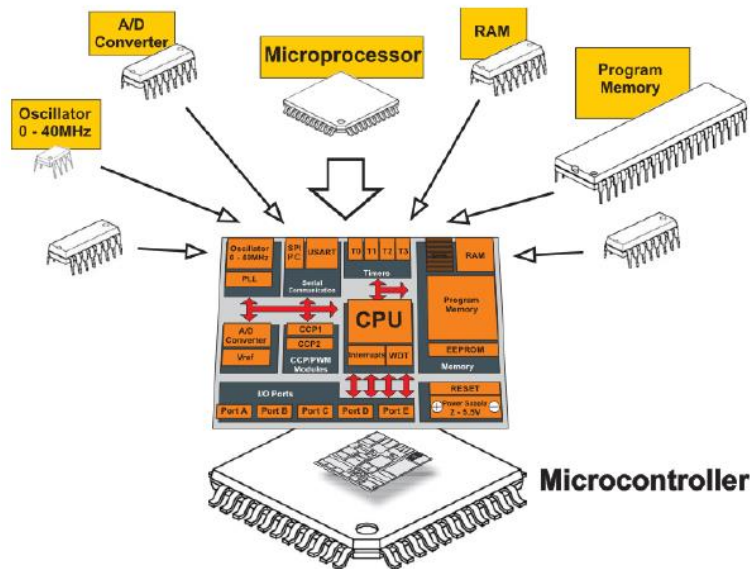
## Sistema microcontrolador de uso específico...

- um computador em um único circuito integrado (chip)!
- RAM-ROM, I/O ports, A/D & D/A...etc. embutidos no chip
- exemplos : Motorola 6811, Intel 8051, Zilog Z8, PICs, AVRs, ...

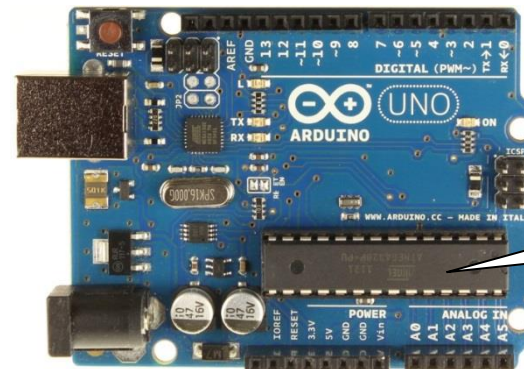
chip único

O microcontrolador integra num único componente os três elementos principais da arquitectura de um computador: CPU, memória e I/O

CPU	RAM	ROM
I/O Port	Timer	Serial Port



Um só chip na motherboard



Microcontrolador  
ATmega328

# Microcontroladores

- podem ser vistos como dispositivos de propósito(objectivo) específico
- usados em tarefas “simples” sem grandes requisitos de processamento, a nível de rapidez e de tipo de instruções
- integram num único circuito integrado (CI - chip):
  - processador;
  - memória;
  - portas de I/O;
  - contadores (contam impulsos);
  - timers (temporizadores, contam tempo);
  - conversores A/D (analógico→digital) e D/A(digital→analógico)
  - ...
- tornam-se assim mais baratos e compactos que os circuitos com microprocessador e outros integrados associados (memória, controladores, etc)

# Microprocessadores vs Microcontroladores

## Microprocessador

- CPU => *stand-alone*  
RAM, ROM, I/O, timers... separados
- projectista pode decidir a quantidade de ROM, RAM e *ports* de I/O;
- expansível
- versatilidade
- uso geral

## Microcontrolador

- CPU, RAM, ROM, I/O, timer... estão integrados em um só *chip*
- quantidade fixa de elementos *on-chip* (ROM, RAM, I/O *ports*)
- para aplicações onde custo, potência e espaço são factores críticos;
- uso específico

# Microcontroladores - exemplos

## 8051 (INTEL)

- 8 bits
- um dos mais utilizados na prática
- conjunto reduzido de instruções (RISC)
- usado numa grande diversidade de equipamentos (ex:máquinas de costura)



## PIC (Microchip Technology - <https://www.microchip.com/>)

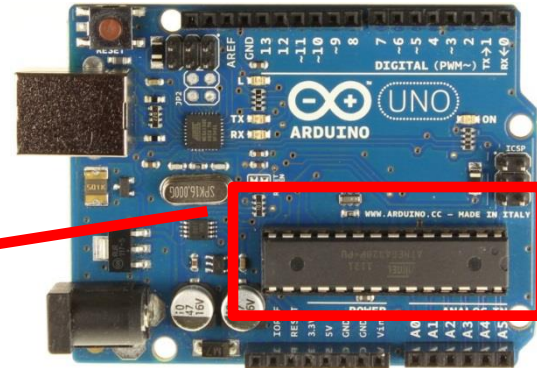
- melhor desempenho
- possui um conjunto de instruções e funções mais elaborados
- baratos (há versões que custam menos de 1€)
- a Microchip fabrica uma família de processadores de 8, 16, 24 e 32 bits
- fáceis de utilizar:
  - a nível de programação
  - a nível de integração com outros componentes electrónicos



# Microcontroladores

## ARDUINO

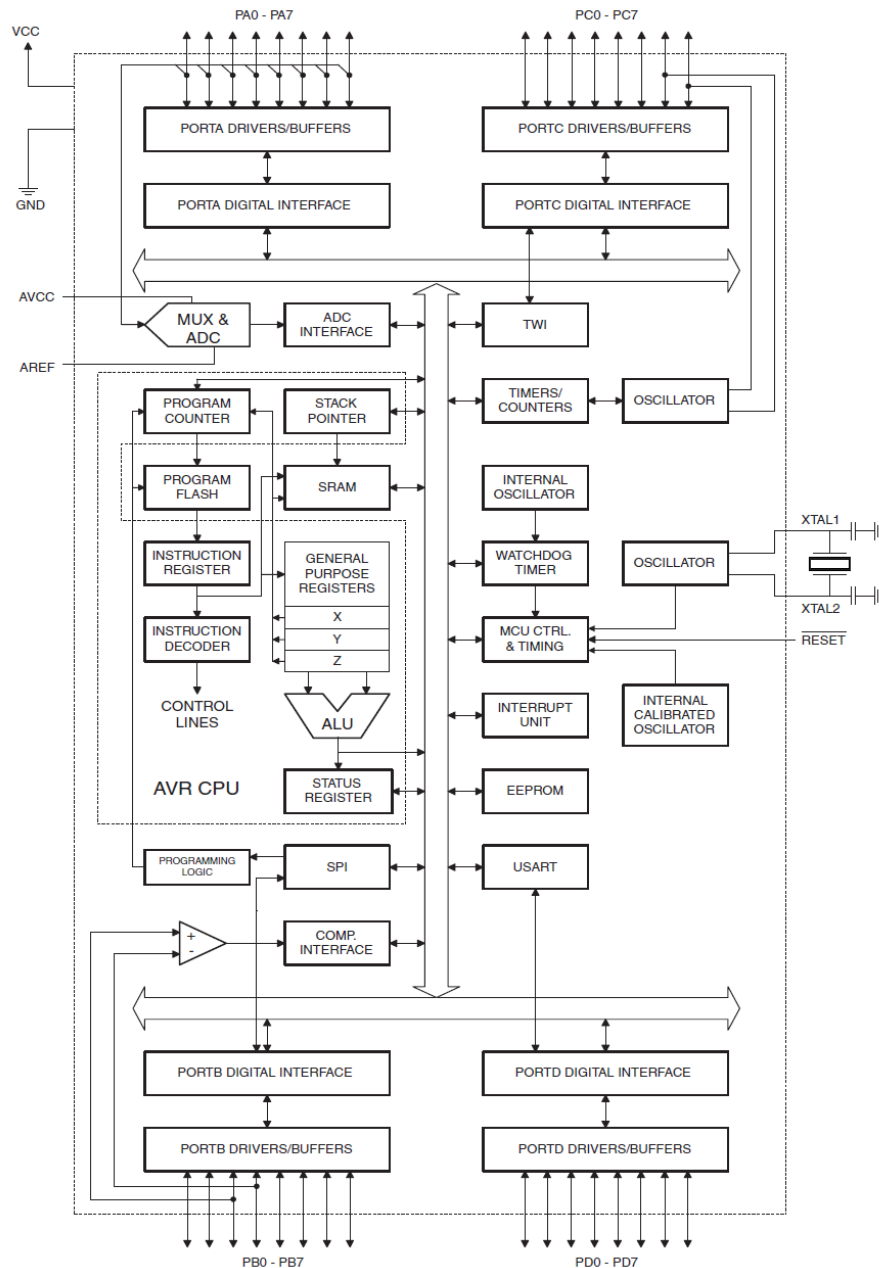
- microcontrolador ATMEL ATMEGA328 : família AVR(Microchip), 8 bits, arquitetura RISC
- 32 KB de Flash , 2 KB de RAM e 1 KB de EEPROM
- Portas I<sup>2</sup>C, série, I/O digital e analógico, A/D e D/A
- Clock de 16 MHz (máx=20MHz)



ATMEL ATMEGA328

# Arduino UNO : Microcontrolador ATMEL ATMEGA328P (Microchip)

[http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P\\_Datasheet.pdf](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf)



ATmega328/P is a low-power CMOS 8-bit microcontroller based on the AVR® enhanced RISC architecture

## Advanced RISC Architecture

- 131 Powerful Instructions
- Most Single Clock Cycle Execution
- 32 x 8 General Purpose Working Registers
- 32KBytes Flash program Memory
- 1KBytes EEPROM
- 2KBytes Internal SRAM
- Data Retention: 20 years at 85°C/100 years at 25°C(1)
- 23 Programmable I/O Lines
- One Programmable Serial USART
- Two 8-bit Timer/Counters + One 16-bit Timer/Counter
- 6-channel 10-bit ADC



# Processadores Digitais de Sinais (DSPs)

## Características:

- diferem dos microprocessadores na arquitetura de hardware, software e no conjunto de instruções, o qual é otimizado para o tratamento digital de sinais
- são empregues em aplicações que exigem processamento de sinais em tempo real

## Usos:

- telecomunicações (filtros, compressão, multiplexação e cancelamento de eco);
- processamento de áudio (gravação em estúdio, sintetizadores, mixers, filtros e reconhecimento de voz);
- processamento de imagem (principalmente na área médica);
- instrumentação e controlo (precisão das medidas e controlo industrial).



Processamento de áudio digital



Texas Instruments TMS320



Consola Nintendo

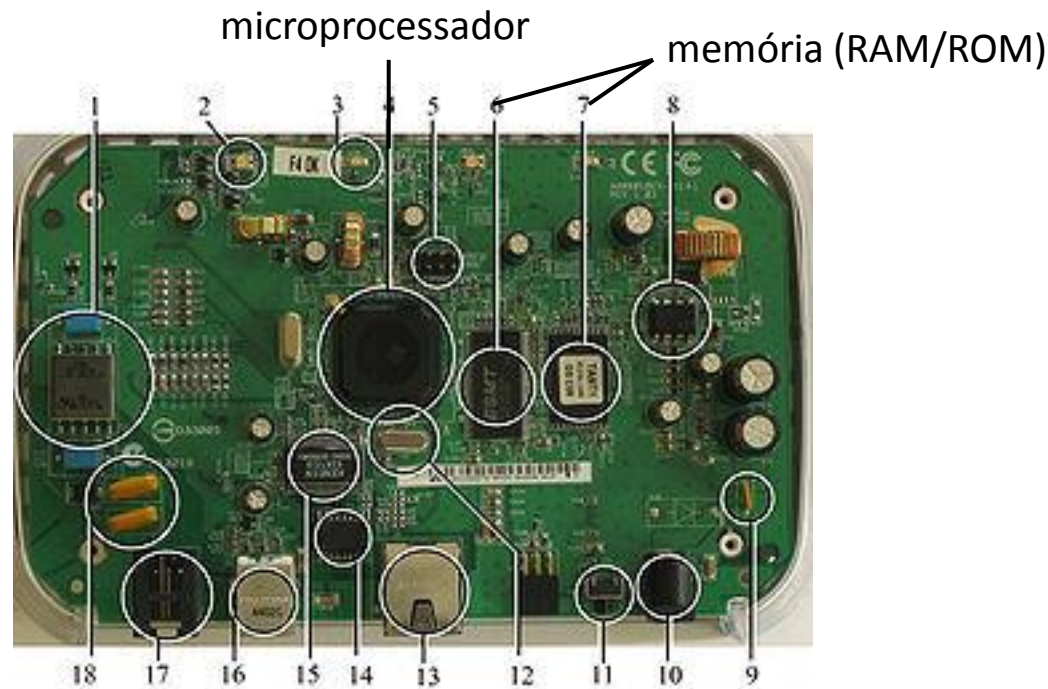


# Sistemas embebidos/embutidos (embedded systems)

## Características:

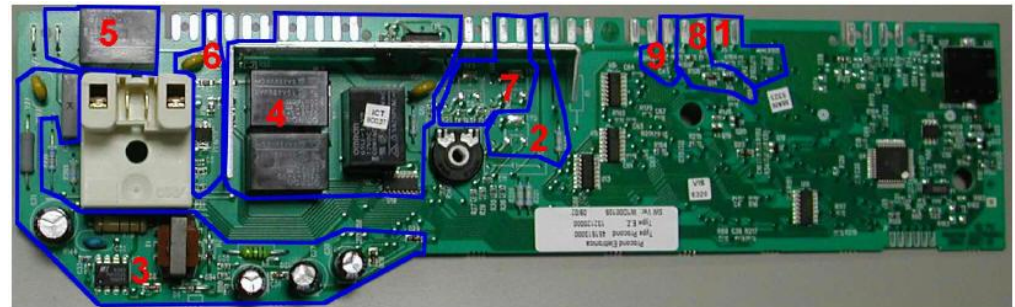
- sistema embebido significa que o processador está embutido na aplicação;
- um produto embebido utiliza um microprocessador/microcontrolador (ou DSP) para fazer uma ou poucas tarefas dedicadas;
- existe somente uma aplicação de software que normalmente está gravada em ROM (firmware);
- normalmente existe a interação com o meio ambiente ou com o operador;
- exemplos: impressora, teclado, consola jogos, telemóvel, modem...

sistema embebido  
modem ADSL



# Sistemas embebidos/embutidos - exemplos

## Controlo de uma máquina de lavar roupa



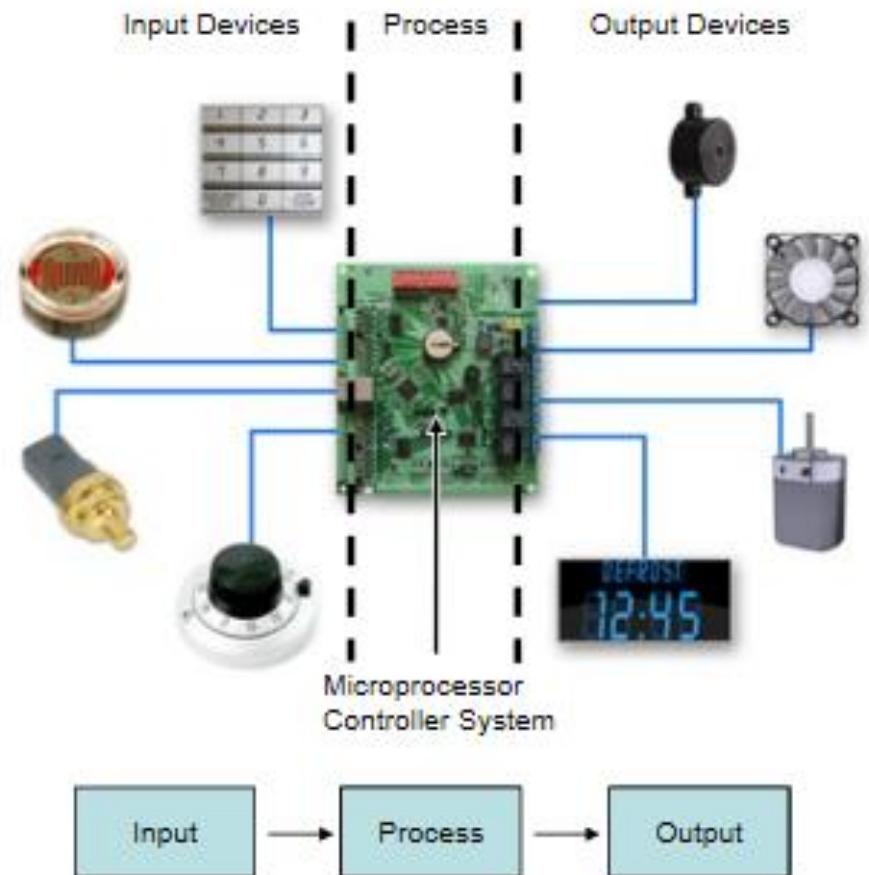
DSP56F800 - 16-bit Digital Signal Controllers

1. NTC washing temperature sensor
2. Drain pump
3. Power supply
4. Motor

5. Heating element
6. Door safety interlock
7. Water fill solenoids
8. Tachymetric generator (motor)
9. Drum positioning system (top-loaders)

# Sistemas embebidos/embutidos - exemplos

## Controlo de um forno microondas





# Sistemas embebidos/embutidos - exemplos

## Aplicações em automóveis

### EMBEDDED SYSTEM IN A CAR



### ***ELECTRONIC CONTROL UNIT***

- *The ECU controls the fuel injection system, ignition timing, and the idle speed control system.*
- *The ECU consists of an 8-bit microprocessor, random access memory (RAM), read only memory (ROM), and an input/output interface.*



ECU / centralina

# Arduino : carta controladora programável

**Arduino** → plataforma de prototipagem de código aberto (hardware e software) criada em 2005 pelo italiano Massimo Banzi.

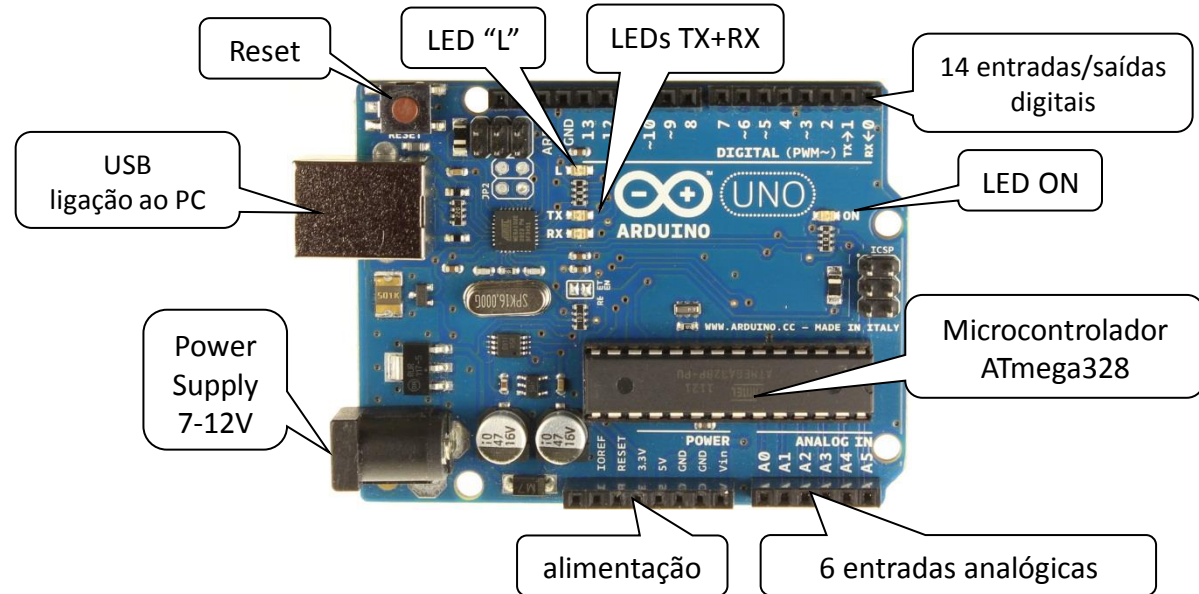
É destinado a artistas, designers, hobbistas, professores e qualquer pessoa interessada em criar objetos ou ambientes interativos.

Objetivo principal : criar uma plataforma hardware de baixo custo apoiada por um sistema de programação de código aberto, para possibilitar o desenvolvimento de protótipos de baixo custo.

Através de sinais provenientes de sensores, o Arduino pode detectar o estado do ambiente que o rodeia e após processamento desses sinais é capaz de controlar indicadores luminosos ou motores e uma diversidade de outros atuadores.

Existe uma enorme comunidade de utilizadores, com inúmeras propostas de ideias e projetos.

# Arduino : modelo UNO

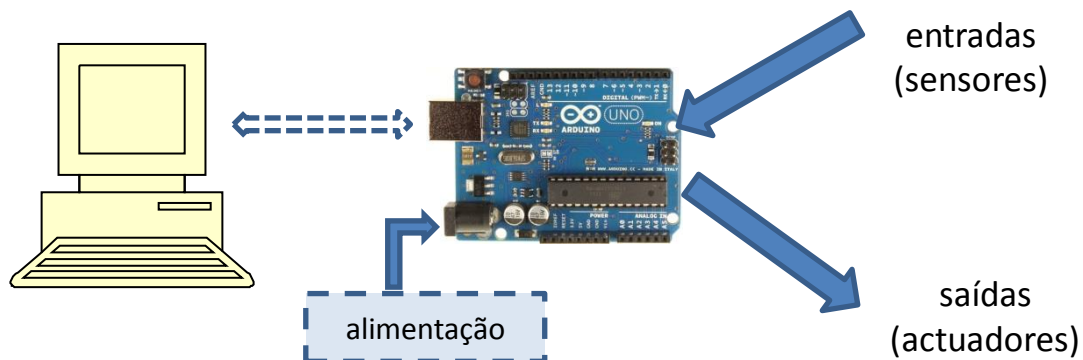


## Características Principais

- Baixo custo (≈25€)
- Microcontrolador: ATmega328, 16MHz
- Tensão de alimentação: USB ou fonte externa
- Memória: Flash(código)=32KB, SRAM(variáveis)=2KB, EEPROM=1KB
- Entradas/saídas digitais: 14, entradas analógicas: 6
- Ligação ao PC: USB
- Comunicações: UART & I<sup>2</sup>C
- Conector de expansão
- Diversos módulos externos (*Shields*): controlo motores, comunicações s/fio, ...
- Existem diversas variantes: Due, Uno, Duemilenove, Mega, ADK, Lillypad, Nano,...
- Existem "clones" com funções melhoradas (ex: chipKIT)
- Fornecedores nacionais: PT Robotics(<http://www.ptrobotics.com/>), SAR (<http://www.botnroll.com/>), ElectroFun(<https://www.electrofun.pt/arduino>)

# Arduino : carta controladora programável

Depois de programado pode funcionar autonomamente ou ligado a um sistema externo (PC)



```
Arduino IDE 1.6.8 - Blink
File Edit Sketch Tools Help
Blink
/*
 * Faz piscar o Led interno do Arduino
 */

const int LedPin = 13; //Led interno está ligado ao pino 13

const int tempoON = 100; //tempo em que o Led está aceso
const int tempoOFF = 500; //tempo em que o Led está apagado

// the setup function runs once when you press reset or power the
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(LedPin, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {

}

Arduino/Genuino Uno on COM17
```

Ambiente de desenvolvimento(Java): ling. tipo C/C++, gratuito, muitas bibliotecas existentes: Ethernet, LCD, DateTime, ...

KIT disponível no LTC: Arduino Physical Computing Kit

IDE: <http://arduino.cc/en/Main/Software>

# Arduino - Variantes



Uno



Chipkit



Due

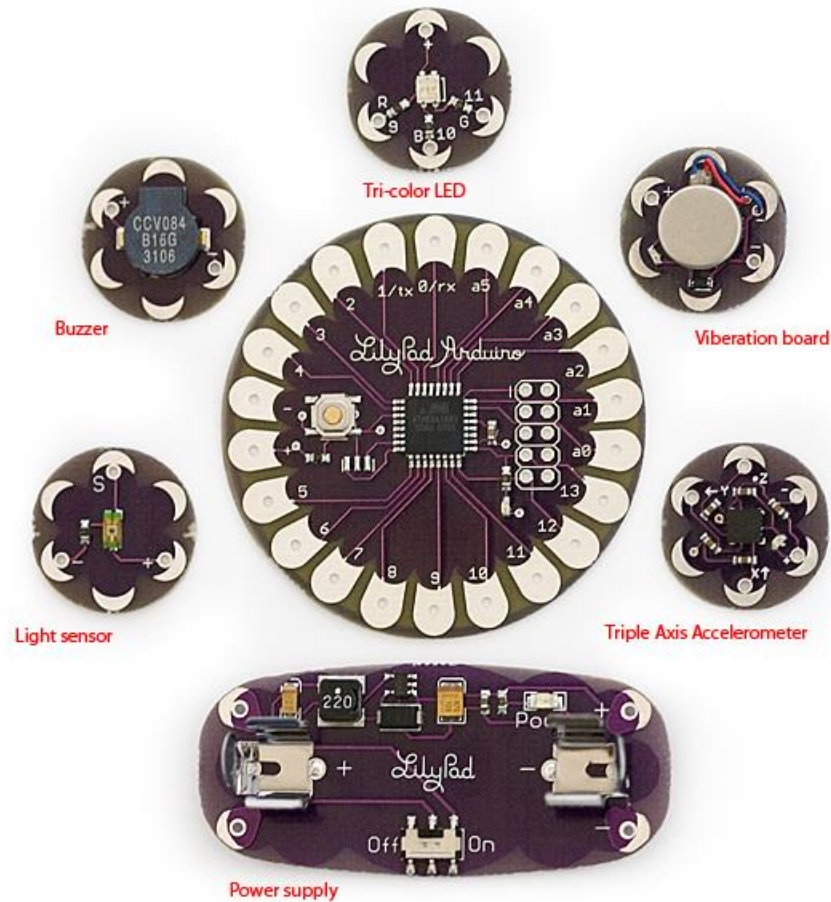


Mini



# Arduino - Variantes

Arduino **Lilypad**: versão para aplicação no vestuário



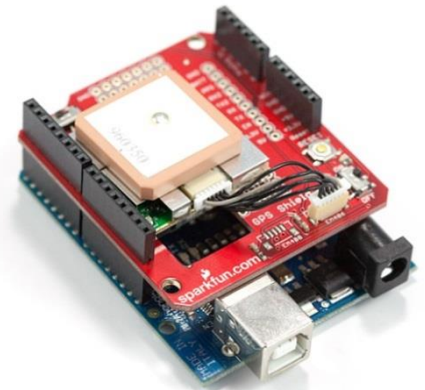
# Arduino – Shields (ampliam as funções da placa base)



Motor



GSM



GPS



WiFi



LCD+keyboard



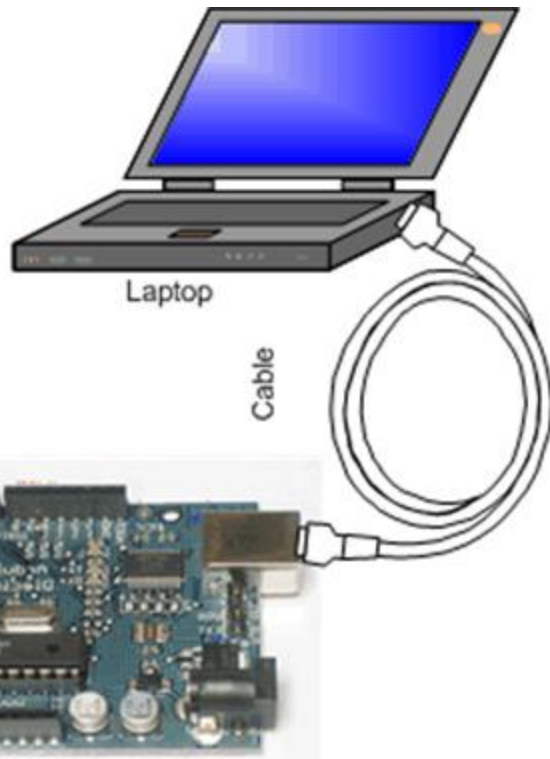
Ethernet

# Arduino - Variantes

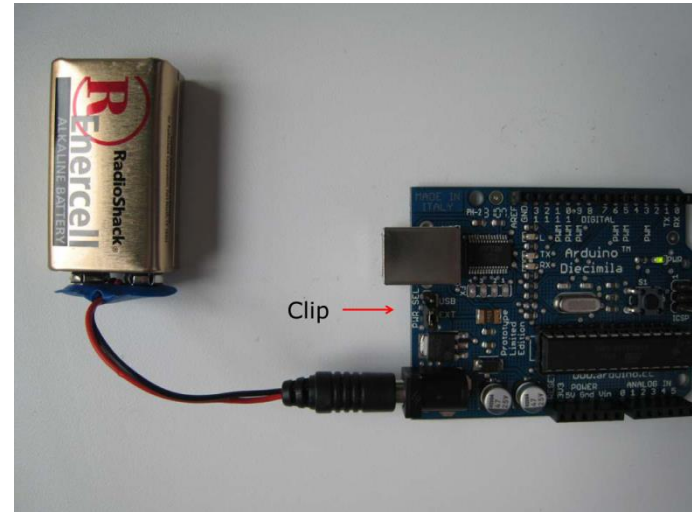
## Plataforma de e-Health: Kit para Arduino (e Raspberry Pi)



# Arduino – alimentação eléctrica



cabo USB



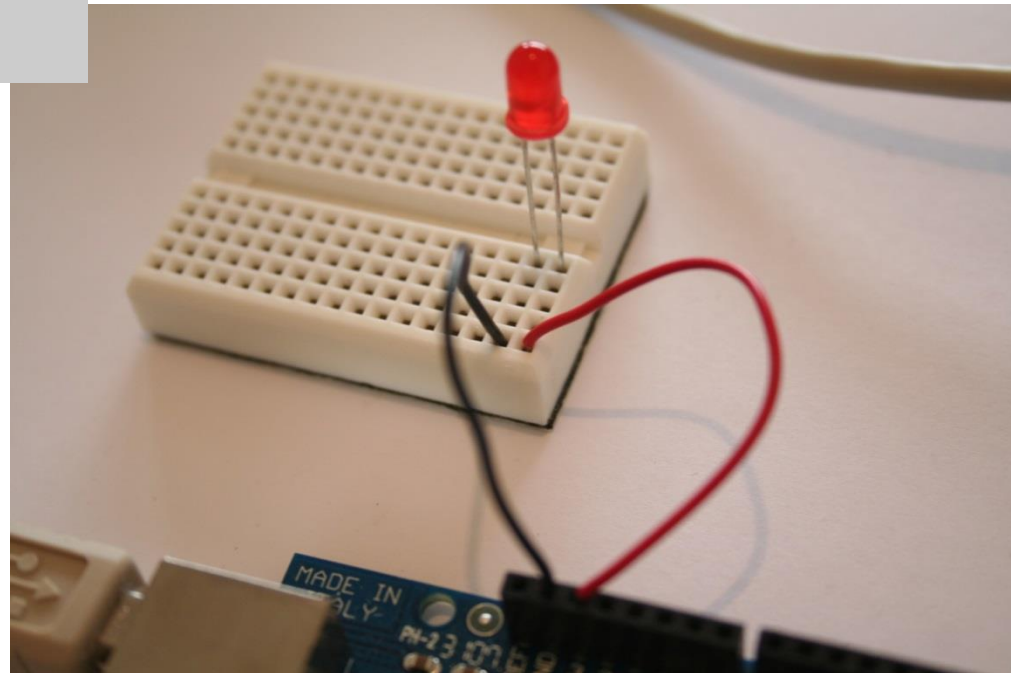
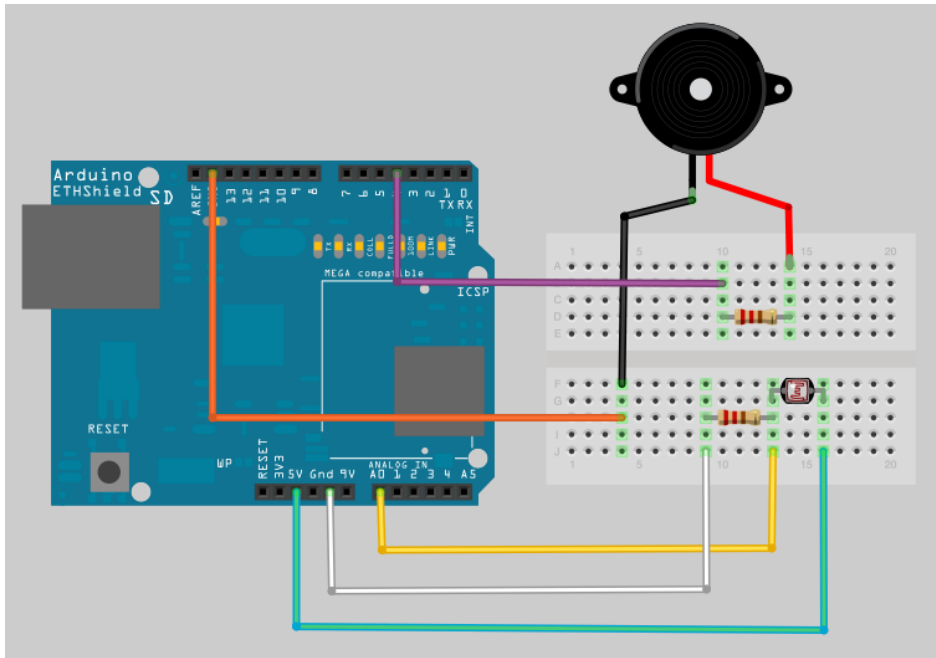
pilha / bateria



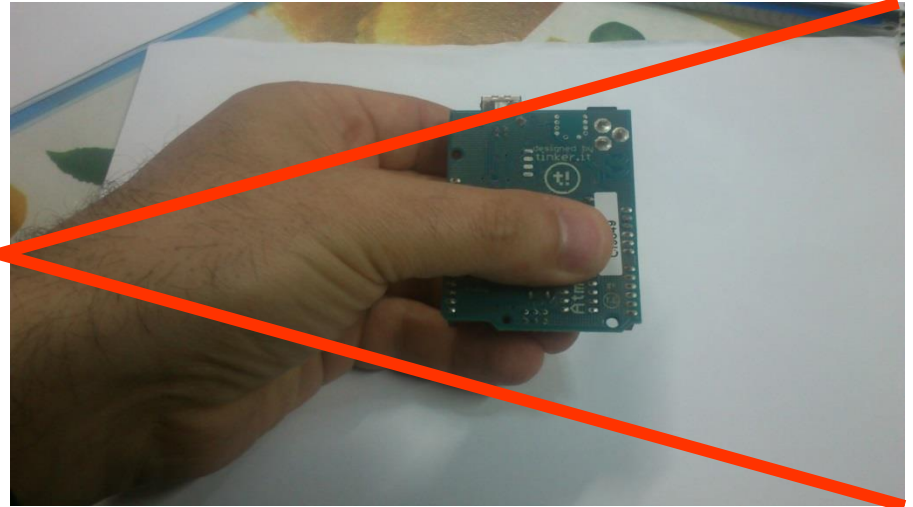
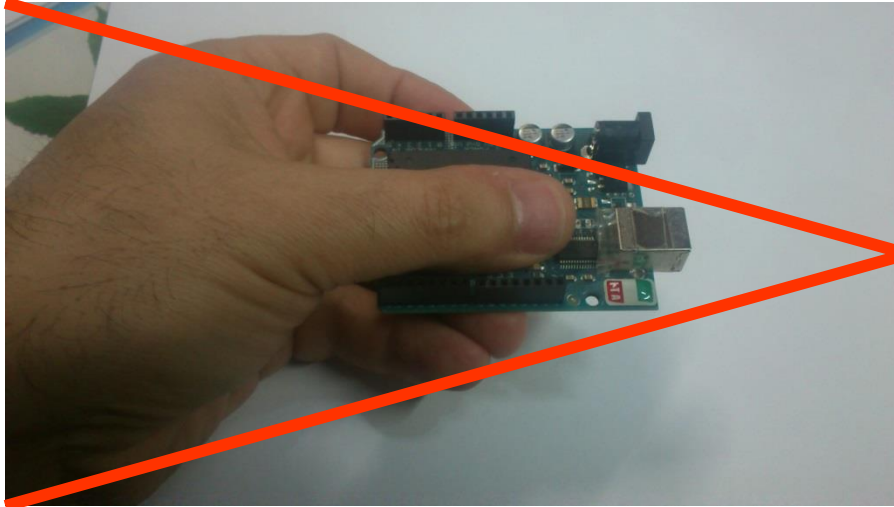
fonte de alimentação (power supply)



# Arduino - ligações



# Arduino - manipulação



# Arduino - programação

Programa para Arduino = “SKETCH”

## Estrutura de um sketch

<declarações> : declaração de constantes, variáveis, tipos, etc    (OPCIONAL)

***void setup ( )***

```
{  
  código executado uma só vez; serve principalmente para efectuar inicializações  
}
```

***void loop ( )***

```
{  
  código executado de modo contínuo (em ciclo) até que a alimentação seja desligada (ou reset).  
}
```

## Comentários

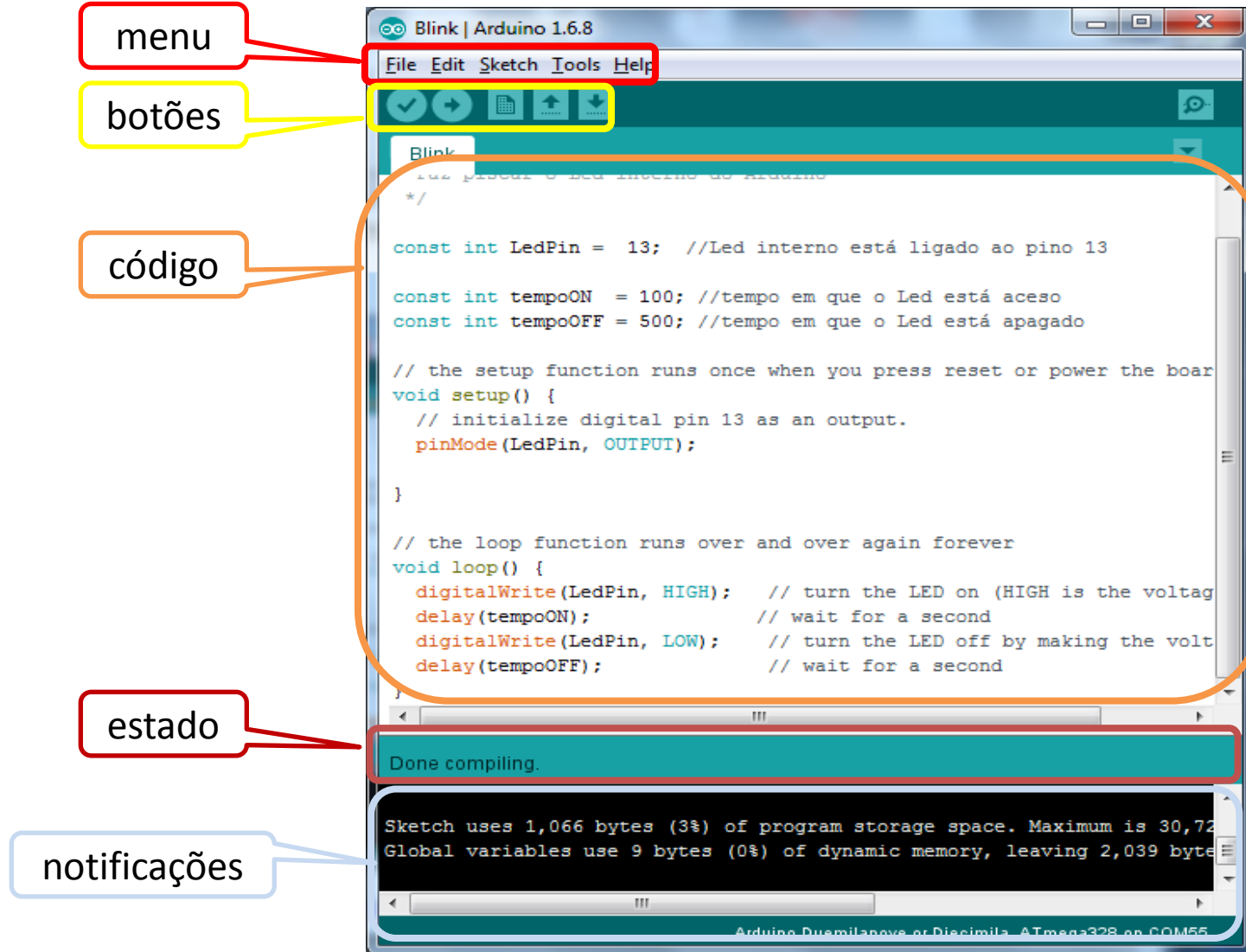
//linha de comentário

/\*

  texto de comentário

\*/

# Arduino IDE(Integrated Development Environment )

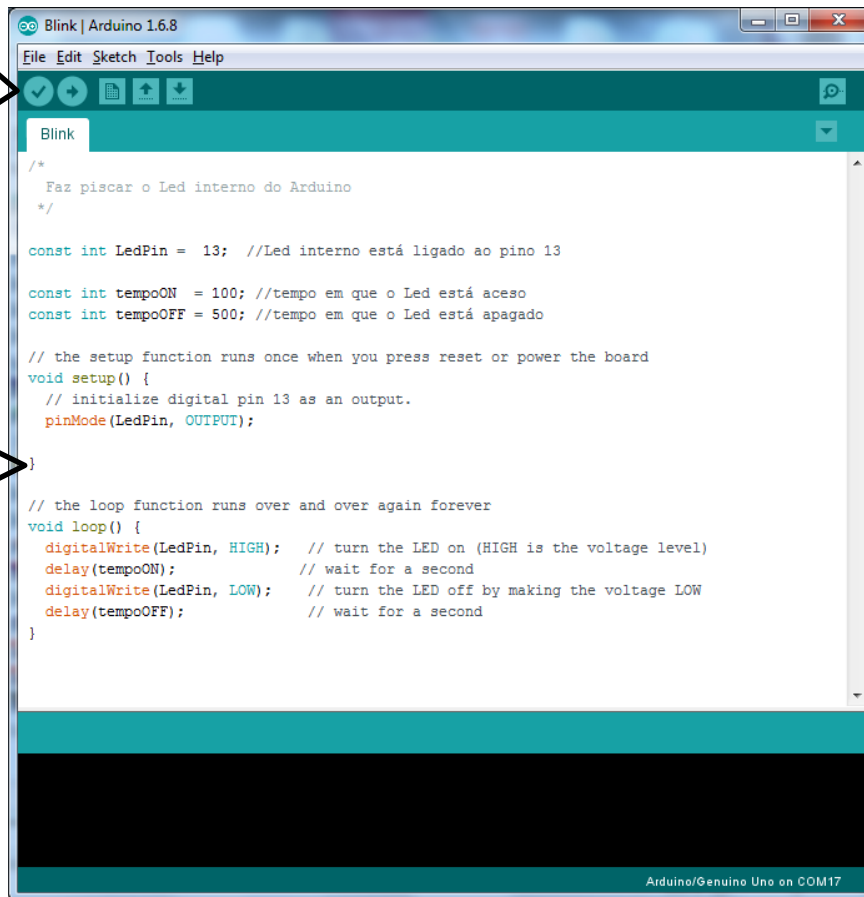




# Arduino – utilização

(2)  
carregar  
código  
(verificar)

(1)  
escrever  
código



```
/*
  Faz piscar o Led interno do Arduino
  */

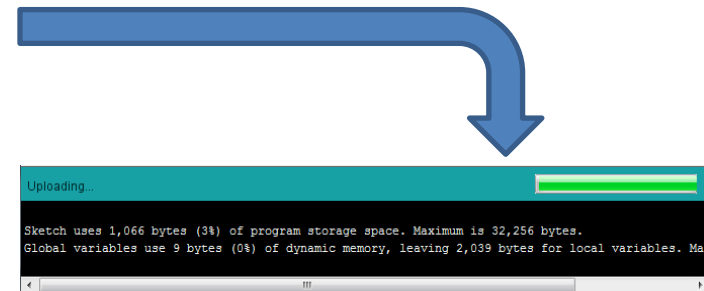
const int LedPin = 13; //Led interno está ligado ao pino 13

const int tempoON = 100; //tempo em que o Led está aceso
const int tempoOFF = 500; //tempo em que o Led está apagado

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(LedPin, OUTPUT);
}

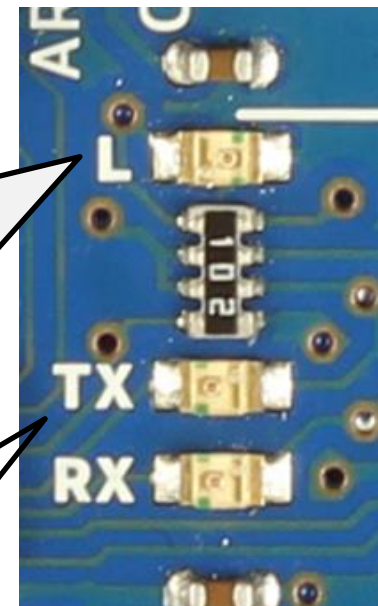
// the loop function runs over and over again forever
void loop() {
  digitalWrite(LedPin, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(tempoON); // wait for a second
  digitalWrite(LedPin, LOW); // turn the LED off by making the voltage LOW
  delay(tempoOFF); // wait for a second
}
```

Sketch = “Blink” (Led L a piscar)



(4)  
Led L  
pisca  
segundo  
o código  
do  
sketch

(3)  
TX/RX  
cintilam

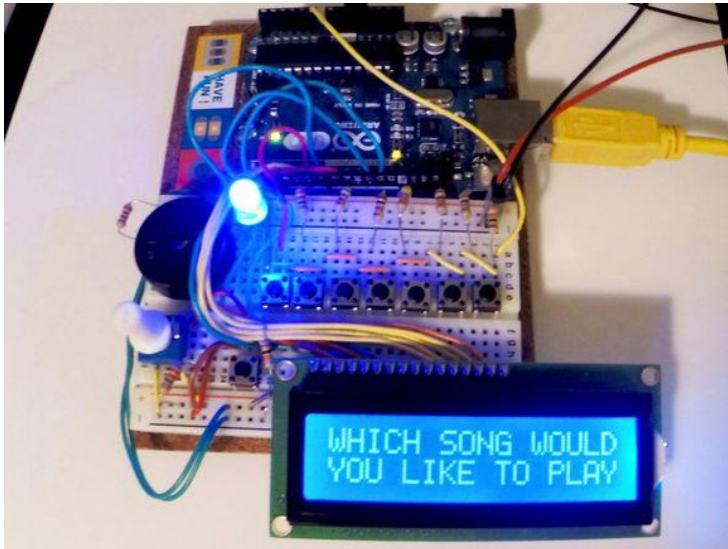


# Arduino : proyectos diversos

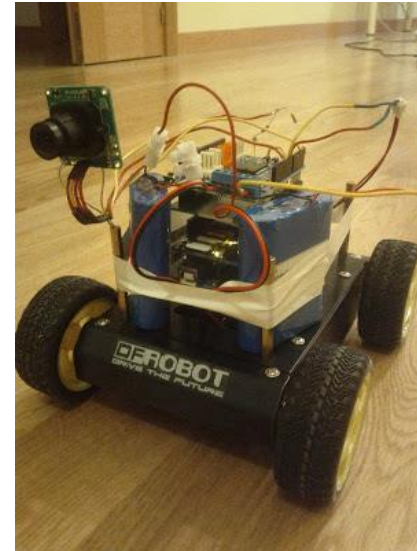
<http://playground.arduino.cc/Projects/Ideas>

<http://runtimeprojects.com/>

Electronic Piano



Internet controlled Arduino car



# Arduino : proyectos diversos

## LilyPad Example: LED Biking Jacket



<http://www.trendhunter.com/trends/lilypad-arduino-diy-programmable-fashion>

# Arduino : proyectos diversos

## T-shirt modded to let you know when you have new emails



<http://www.engadget.com/2010/03/30/t-shirt-modded-to-let-you-know-when-you-have-new-emails/>

# Arduino : resumen de comandos (cheat-sheet)



## ARDUINO CHEAT SHEET

JEROEN DOGGEN, AP UNIVERSITY COLLEGE ANTWERP



### Structure

```
void setup()
void loop()
```

### Control Structures

```
if (x < 5) {}
for (int i = 0; i < 255; i++) {}
while (x < 6) {}
```

### Further Syntax

```
// Single line comment
/* Multi line comment */
#define ANSWER 42
#include <myLib.h>
```

### General Operators

```
= assignment
+, - addition, subtraction
*, / multiplication, division
% modulo
== equal to
!= not equal to
< less than
<= less than or equal to
```

### Pointer Access

```
& reference operator
* dereference operator
```

### Bitwise Operators

```
& bitwise AND
| bitwise OR
^ bitwise XOR
~ bitwise NOT
```

### Compound Operators

```
++ Increment
-- Decrement
+= Compound addition
&= Compound bitwise AND
```

### Constants

HIGH, LOW  
INPUT, OUTPUT  
true, false  
53: Decimal  
B11010101: Binary  
0x5BA4: Hexadecimal

### Data Types

void  
boolean 0, 1, false, true  
char e.g. 'a' -128 → 127  
unsigned char 0 → 255  
int -32768 → 32767  
unsigned int 0 → 65535  
long -2147483648 → 2147483647  
float -3,4028235E+38 → 3.402835E+38  
sizeof(myint) returns 2 bytes

### Arrays

```
int myInts[6];
int myPins[] = {2, 4, 8, 5, 6};
int myVals[6] = {-2, -4, 9, 3, 5};
```

### Strings

```
char S1[15];
char S2[8] = 'A','r','d','u','i','n','o';
char S3[8] = 'A','r','d','u','i','n','o','\0';
char S4[] = "Arduino";
char S5[8] = "Arduino";
char S6[15] = "Arduino";
```

### Conversion

```
char() int() long()
byte() word() float()
```

### Qualifiers

static Persist between calls  
volatile Use RAM (nice for ISR)  
const Mark read-only  
PROGMEM Use flash memory

### Interrupts

```
attachInterrupt(interrupt, function, type)
detachInterrupt(interrupt)
boolean(interrupt)
interrupts()
noInterrupts()
```

### Advanced I/O

```
tone(pin, freqHz)
tone(pin, freqHz, duration_ms)
noTone(pin)
shiftOut(dataPin, clockPin, how, value)
unsigned long pulseIn(pin, [HIGH, LOW])
```

### Time

```
unsigned long millis() 50 days overflow
unsigned long micros() 70 min overflow
delay(ms)
delayMicroseconds(us)
```

### Math

```
min(x,y) max(x,y) abs(x)
sin(rad) cos(rad) tan(rad)
pow(base, exponent)
map(val, fromL, fromH, toL, toH)
constrain(val, fromL, toH)
```

### Pseudo Random Numbers

```
randomSeed(seed)
long random(max)
long random(min, max)
```

### ATmega328 Pinout

ATmega328 Arduino									
RESET	1	5V	5V	5V	5V	5V	5V	5V	5V
D0	2	3	4	5	6	7	8	9	10
D1	11	12	13	14	15	16	17	18	19
D2	20	21	22	23	24	25	26	27	28
D3	29	30	31	32	33	34	35	36	37
D4	38	39	40	41	42	43	44	45	46
D5	47	48	49	50	51	52	53	54	55
D6	56	57	58	59	60	61	62	63	64
D7	65	66	67	68	69	70	71	72	73
D8	74	75	76	77	78	79	80	81	82
D9	83	84	85	86	87	88	89	90	91
D10	92	93	94	95	96	97	98	99	100
D11	101	102	103	104	105	106	107	108	109
D12	110	111	112	113	114	115	116	117	118
D13	119	120	121	122	123	124	125	126	127
D14	128	129	130	131	132	133	134	135	136
D15	137	138	139	140	141	142	143	144	145
D16	146	147	148	149	150	151	152	153	154
D17	155	156	157	158	159	160	161	162	163
D18	164	165	166	167	168	169	170	171	172
D19	173	174	175	176	177	178	179	180	181
D20	182	183	184	185	186	187	188	189	190
D21	191	192	193	194	195	196	197	198	199
D22	200	201	202	203	204	205	206	207	208
D23	209	210	211	212	213	214	215	216	217
D24	218	219	220	221	222	223	224	225	226
D25	227	228	229	230	231	232	233	234	235
D26	236	237	238	239	240	241	242	243	244
D27	245	246	247	248	249	250	251	252	253
D28	254	255	256	257	258	259	260	261	262
D29	263	264	265	266	267	268	269	270	271
D30	272	273	274	275	276	277	278	279	280
D31	281	282	283	284	285	286	287	288	289
D32	290	291	292	293	294	295	296	297	298
D33	299	300	301	302	303	304	305	306	307
D34	308	309	310	311	312	313	314	315	316
D35	317	318	319	320	321	322	323	324	325
D36	326	327	328	329	330	331	332	333	334
D37	335	336	337	338	339	340	341	342	343
D38	344	345	346	347	348	349	350	351	352
D39	353	354	355	356	357	358	359	360	361
D40	362	363	364	365	366	367	368	369	370
D41	371	372	373	374	375	376	377	378	379
D42	380	381	382	383	384	385	386	387	388
D43	389	390	391	392	393	394	395	396	397
D44	398	399	400	401	402	403	404	405	406
D45	407	408	409	410	411	412	413	414	415
D46	416	417	418	419	420	421	422	423	424
D47	425	426	427	428	429	430	431	432	433
D48	434	435	436	437	438	439	440	441	442
D49	443	444	445	446	447	448	449	450	451
D50	452	453	454	455	456	457	458	459	460
D51	461	462	463	464	465	466	467	468	469
D52	470	471	472	473	474	475	476	477	478
D53	479	480	481	482	483	484	485	486	487
D54	488	489	490	491	492	493	494	495	496
D55	497	498	499	500	501	502	503	504	505
D56	506	507	508	509	510	511	512	513	514
D57	515	516	517	518	519	520	521	522	523
D58	524	525	526	527	528	529	530	531	532
D59	533	534	535	536	537	538	539	540	541
D60	542	543	544	545	546	547	548	549	550
D61	551	552	553	554	555	556	557	558	559
D62	560	561	562	563	564	565	566	567	568
D63	569	570	571	572	573	574	575	576	577
D64	578	579	580	581	582	583	584	585	586
D65	587	588	589	590	591	592	593	594	595
D66	596	597	598	599	600	601	602	603	604
D67	605	606	607	608	609	610	611	612	613
D68	614	615	616	617	618	619	620	621	622
D69	623	624	625	626	627	628	629	630	631
D70	632	633	634	635	636	637	638	639	640
D71	641	642	643	644	645	646	647	648	649
D72	650	651	652	653	654	655	656	657	658
D73	659	660	661	662	663	664	665	666	667
D74	668	669	670	671	672	673	674	675	676
D75	677	678	679	680	681	682	683	684	685
D76	686	687	688	689	690	691	692	693	694
D77	695	696	697	698	699	700	701	702	703
D78	704	705	706	707	708	709	710	711	712
D79	713	714	715	716	717	718	719	720	721
D80	722	723	724	725	726	727	728	729	730
D81	731	732	733	734	735	736	737	738	739
D82	740	741	742	743	744	745	746	747	748
D83	749	750	751	752	753	754	755	756	757
D84	758	759	760	761	762	763	764	765	766
D85	767	768	769	770	771	772	773	774	775
D86	776	777	778	779	780	781	782	783	784
D87	785	786	787	788	789	790	791	792	793
D88	794	795	796	797	798	799	800	801	802
D89	803	804	805	806	807	808	809	810	811
D90	812	813	814	815	816	817	818	819	820
D91	821	822	823	824	825	826	827	828	829
D92	830	831	832	833	834	835	836	837	838
D93	839	840	841	842	843	844	845	846	847
D94	848	849	850	851	852	853	854	855	856
D95	857	858	859	860	861	862	863	864	865
D96	866	867	868	869	870	871	872	873	874
D97	875	876	877	878	879	880	881	882	883
D98	884	885	886	887	888	889	890	891	892
D99	893	894	895	896	897	898	899	900	901
D100	902	903	904	905	906	907	908	909	910
D101	911	912	913	914	915	916	917	918	919
D102	920	921	922	923	924	925	926	927	928
D103	929	930	931	932	933	934	935	936	937
D104	938	939	940	941	942	943	944	945	946
D105	947	948	949	950	951	952	953	954	955
D106	956	957	958	959	960	961	962	963	964
D107	965	966	967	968	969	970	971	972	973
D108	974	975	976	977	978	979	980	981	982
D109	983	984	985	986	987	988	989	990	991
D110	992	993	994	995	996	997	998	999	1000

### I/O Pins

	Uno	Mega
# of IO	14 + 6	54 + 11
Serial Pins	0 - RX, 1 - TX	RX1 → RX4
Interrupts	2, 3	2, 3, 18, 19, 20, 21
PWM Pins	5, 6 - 9, 10 - 3, 11	0 → 13
SPI (SS, MISO, MOSI, SCK)	10 → 13	50 → 53
I2C (SDA, SCL)	A4, A5	20, 21

### Analog I/O

```
analogReference(EXTERNAL, INTERNAL)
analogRead(pin)
analogWrite(pin, value)
```

### Digital I/O

```
pinMode(pin, [INPUT, OUTPUT])
digitalRead(pin)
digitalWrite(pin, value)
```

### Serial Communication

```
Serial.begin(speed)
Serial.print("text")
Serial.println("text")
```

### Websites

forum.arduino.cc  
playground.arduino.cc  
arduino.cc/en/Reference

### Arduino Uno Board





# Arduino : links úteis

- **Site oficial Arduino** : <http://www.arduino.cc>

Comandos da linguagem Arduino: <http://www.arduino.cc/en/Reference/HomePage>

- **Simuladores**

123D Circuits.io : <http://123d.circuits.io>

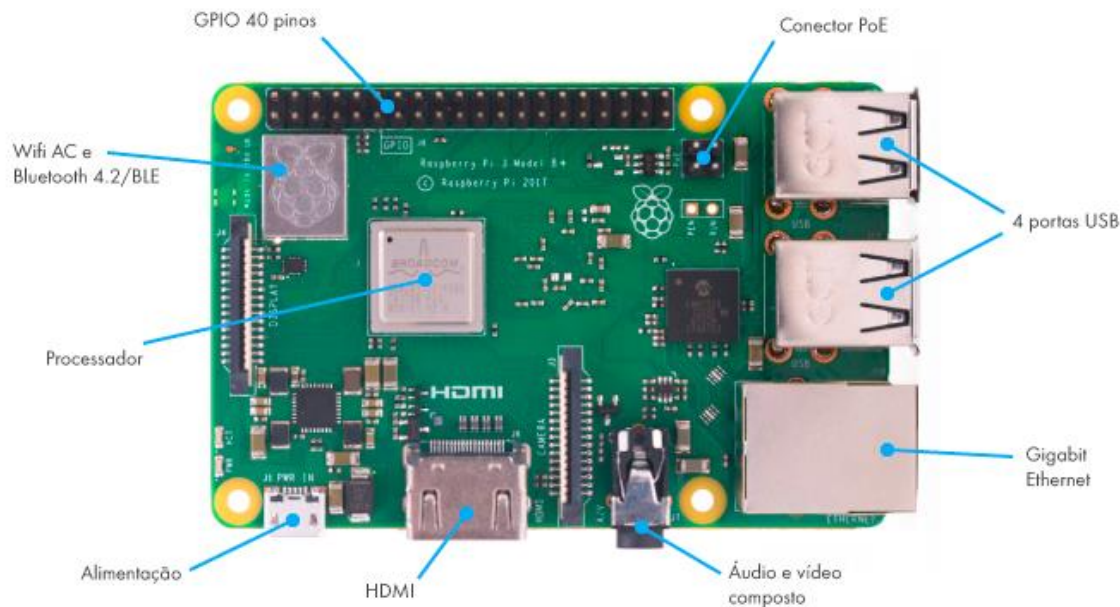
Virtual Breadboard

<http://www.virtualbreadboard.com/Main.aspx?TAB=Home>

<http://www.virtualbreadboard.com/Main.aspx?TAB=Downloads>

- **Fritzing** - para desenhar esquemas elétricos: [www.fritzing.org](http://www.fritzing.org)
- **Processing** - linguagem de programação usada para escrever programas com interface gráfica:  
<https://playground.arduino.cc/Interfacing/Processing>

# Raspberry Pi : [https://pt.wikipedia.org/wiki/Raspberry\\_Pi](https://pt.wikipedia.org/wiki/Raspberry_Pi)



**Raspberry Pi** é um computador do tamanho de um cartão de crédito, que se conecta a um monitor de computador ou TV, e usa um teclado e um mouse padrão; foi desenvolvido no Reino Unido pela *Fundação Raspberry Pi*.

Todo o hardware é integrado numa única placa.

O Raspberry Pi 3 B+ é baseado em um SoC(system on a chip) Broadcom BCM2837(64 bit, quad core) a 1.4GHz, 1 GB de RAM, Bluetooth 4.2.

O projeto não inclui uma memória não-volátil - como um disco rígido - mas possui uma entrada de cartão SD para armazenamento de dados.

Pode correr o Linux (Snappy Ubuntu Core) ou o Microsoft Windows 10 IoT edition.

[www.raspberrypi.org](http://www.raspberrypi.org)

# **Sistemas e Tecnologias de Informação**



# Tecnologia ?

Tecnologia  $\Rightarrow$  do Grego: Techné + Logia

Techné – saber fazer

Logia – conhecimento organizado

Tecnologia:

- conhecimento aplicado à prática
- saber fazer

# Tecnologia ?

- **Uma tecnologia é criada para resolver um problema...**
  - ✓ desenvolvimento de um componente – ex: transistor
  - ✓ um produto completo – ex: CI's, placa principal (motherboard)
  - ✓ uma transformação no interior de um processo complexo –  
ex: tecnologia do LCD's, plasmas, ...
  - ✓ frequentemente faz uso de outras tecnologias – ex: laser, vidro, plásticos,...
- implícito:
  - saber fazer coisas
  - processo de satisfação das necessidades e desejos da sociedade

# Tecnologia : uma definição

**Tecnologia** = “conjunto complexo de conhecimentos, de meios e de know-how, organizado com vista a uma produção”

conhecimentos: pertencem a uma disciplina científica, mas não constituem uma tecnologia (ex: cálculos matemáticos);

meios: concretizam a tecnologia, mas não garantem a sua utilização (ex: equipamentos não têm utilidade sem pessoal qualificado);

know-how : meio de produção de resultados, mas que sem suporte cai rapidamente em desuso (ex: têxteis, especialização não aplicada);

**Obriga aos três componentes em simultâneo** : um ou mesmo dois não bastam!

Exemplo: se o conhecimento incompleto dos fenómenos impede a utilização plena de uma tecnologia, a inexistência de meios não a deixa sequer sair do papel.

# Tecnologia e Conceitos Associados

Relacionamento entre:

- **Tecnologia ↔ Ciência** : tecnologia ≠ ciência
  - ciência visa a aquisição ou reforço do conhecimento (certezas provisórias)
  - tecnologia visa a aplicação útil desse conhecimento (produção em condições industriais, não muito difíceis nem esotéricas, mas definidas com precisão)
- **Tecnologia ↔ Inovação** : tecnologia é o suporte da inovação;
  - as empresas inovam para competir e sobreviver
  - a inovação deve constar da estratégia da empresa
- **Tecnologia ↔ Investimento** : é necessário investir para a obter
  - seja desenvolvendo, seja adquirindo

# Características da tecnologia

- **É negociável e transferível**
  - uma empresa pode comprar tecnologia desenvolvida por outra
  - podem conceder-se licenças de exploração (ex: patentes), com base em contratos que estabelecem o *know-how* que é transferido, incluindo cláusulas de garantia (de qualidade, de cedência e de resultados)
- **Apela a várias disciplinas científicas**
  - ex: Laser  $\Rightarrow$  óptica, electrónica, mecânica dos fluidos, termodinâmica, ...

# Características da tecnologia

- **Oportunidade (de negócio)**

- esperança: para quem se lança numa nova tecnologia (ex:RFID)
- ameaça: para quem faz investimentos industriais e comerciais sem certeza de os ver amortizados (ex: BluRay[Sony] / HD-DVD[Toshiba] ou VHS[JVC] e Betamax[Sony] década de 1980)

- **Avaliação da actualidade (lançamento de uma nova tecnologia)**

- há muita actividade de I&D nesta área?
- que empresas estão interessadas nela, quer para a desenvolver, quer para a aplicar?
- que inovações origina: novos produtos e novos processos?
- como está protegida? existem patentes registadas?
- principal suporte da sua difusão? (como é difundida?)
- regista insucessos?

# Apresentação da Tecnologia

Uma tecnologia pode apresentar-se sob a forma de:

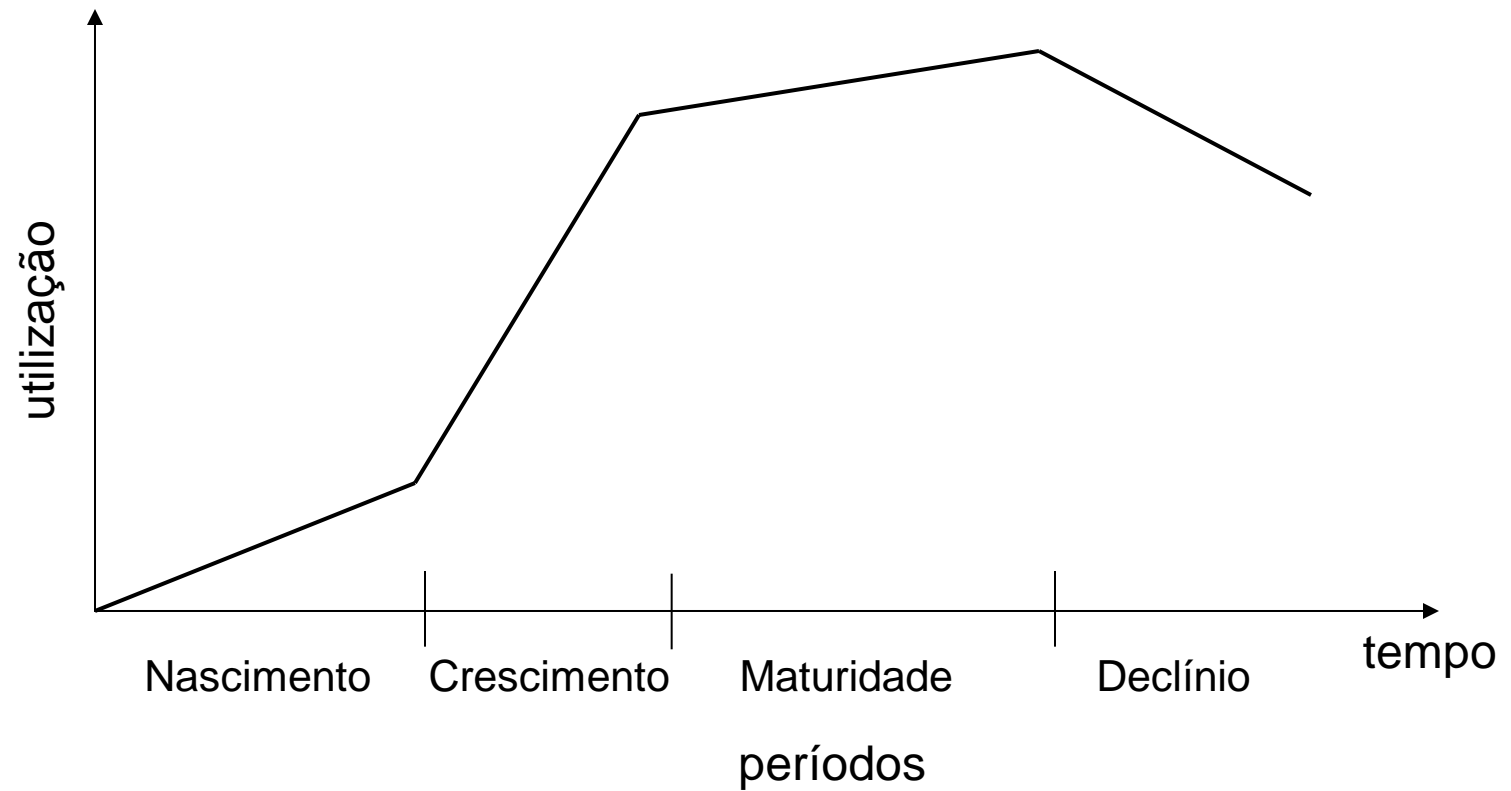
- um produto tecnológico (tangível) [ex: transístor]
- um processo tecnológico (método intangível) [ex: tecnologia dos semicondutores, processo de trabalhar os materiais semicondutores]
- um tipo incorporado no outro (tangível e intangível) [ex: saber como produzir um transístor]
- um conhecimento ou um modelo conceptual pronto para ser produzido (conhecimento explicitado em patentes, relatórios de investigação aplicada, manuais etc.).

Trabalhar com a tecnologia tangível é mais simples do que tratar do intangível que está incorporado num produto ou processo

[ex: é fácil calcular o valor de um produto com base no custo dos materiais que o compõem, mas é difícil avaliar a contribuição do *know-how*]

# Tecnologia: ciclo de vida

- Nasce, vive e morre





# Sistemas

Em geral: sistemas físicos: sistema solar;  
sistema biológicos: o corpo humano;  
sistema sócio-económicos: empresas, sociedades

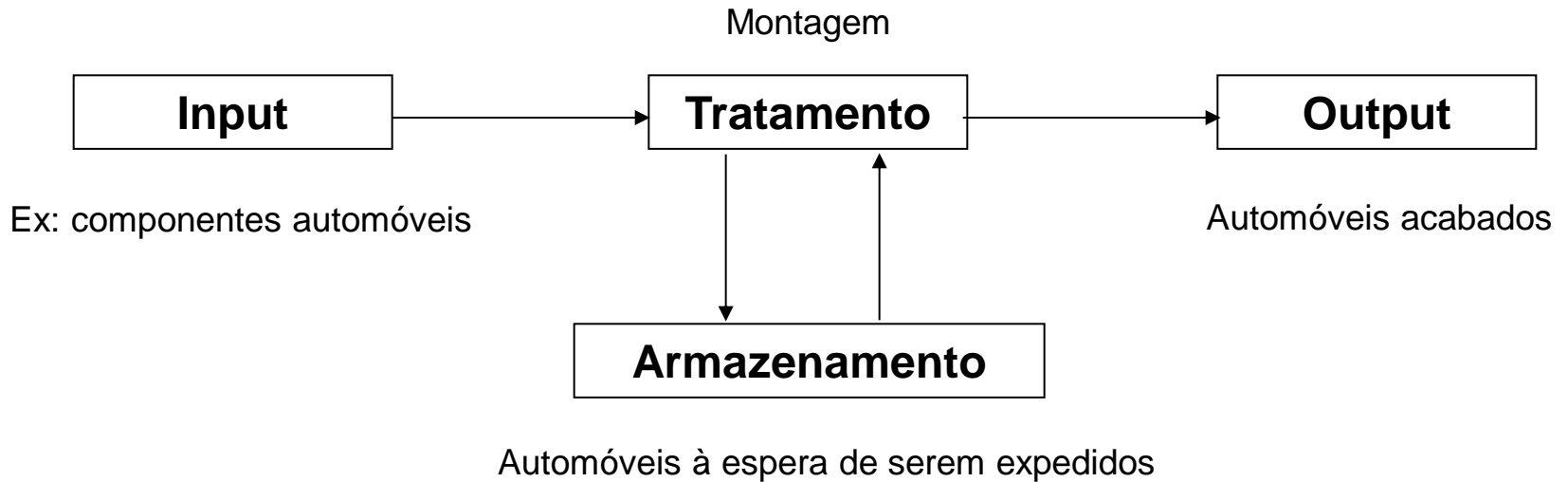
Definição: conjunto de componentes inter-relacionados e inter-dependentes que formam um todo e que trabalham em conjunto para atingirem objectivos comuns.

Um sistema é composto por quatro funções básicas:

- 1) Input: recolha/aquisição dos elementos que entram no sistema para serem processados. Ex: matérias primas, energia, esforço humano
- 2) Tratamento: processo de transformação dos dados em produto acabado
- 3) Armazenamento: armazenamento temporário dos produtos
- 4) Output: produto acabado, resultante do processo de transformação

# Sistema

Ex: linha de montagem de automóveis



# Dados → Informação

## **Dados**

Factos e/ou eventos isolados (palavras, números, sons, imagens), não estando agrupados em nenhuma forma particular que os torne úteis para serem utilizados.

Ex: Covilhã, 9°C, 1/1/2009, 10h30m

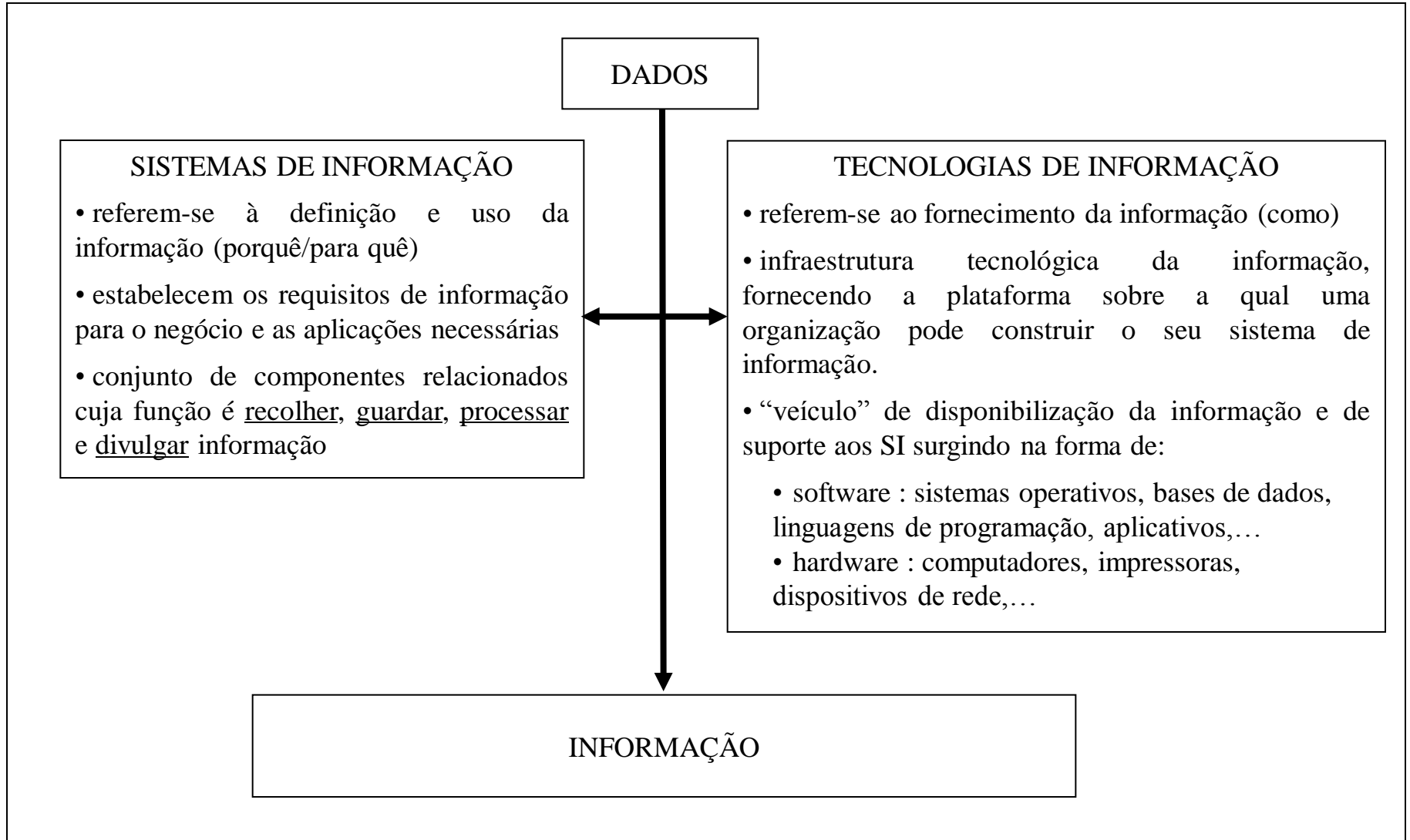
## **Informação**

Resultado do tratamento dos dados, atribuindo-lhes significado e um formato que possibilita compreender esses dados (e usá-los para tomar decisões).

Ex: temperatura na Covilhã às 10h30m do dia 1/1/2009 foi de 9°C

*“Informação é aquele conjunto de dados que, quando fornecido de forma e a tempo adequado, melhora o conhecimento da pessoa que o recebe, ficando ela mais habilitada a desenvolver determinada actividade ou a tomar determinada decisão”*

# TSI - Tecnologias e Sistemas de Informação



Finalidade das TSI: *“obter as informações certas, para as pessoas certas, no momento certo, na quantidade e formato certos”*

# Computador ?

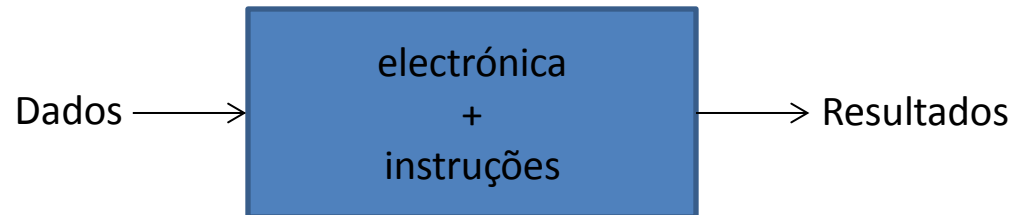
- "Computador é uma máquina que executa operações matemáticas e/ou lógicas com símbolos numéricos, alfanuméricos ou outras formas de informação e produz resultados compreensíveis pelo homem ou por outras máquinas." (*in Collier's Encyclopedia*).

Esta definição não caracteriza o computador, pois abrange por exemplo uma máquina de lavar roupa: calcula tempos de lavagem, quantidade de água, executa funções com base em sequências pré-determinadas e produz um resultado (roupa lavada)

- A definição seguinte apresenta outra característica: as instruções.

"Computador é um conjunto de dispositivos electrónicos capazes de aceitar dados e instruções, executar essas instruções para processar os dados, e apresentar os resultados. (*in Academic Press Dictionary of Science Technology*)

Assim, computador pode ser esquematicamente entendido como:



No entanto, esta definição/esquema não distingue, por exemplo, uma calculadora de um computador.

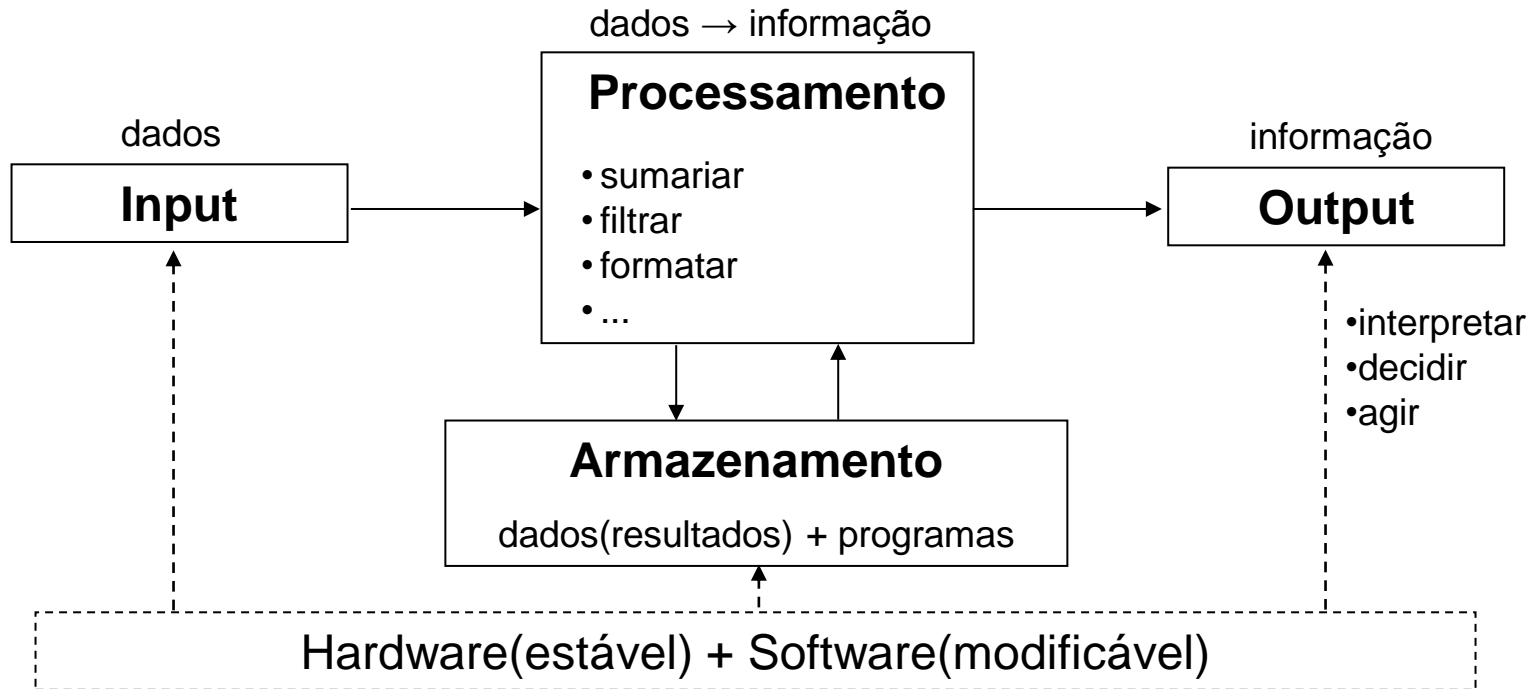
# Computador ?

- Definição um pouco mais abrangente:

“É um equipamento eletrónico capaz de armazenar, recuperar e processar dados. O processamento inclui ordenar, calcular, testar, pesquisar e editar dados e informações de acordo com instruções estabelecidas, segundo uma representação binária e de acordo com um conjunto de regras aritméticas e lógicas.”

- São constituídos por uma parte física, o *hardware* e pelas instruções de controlo, o *software*. O *hardware* é relativamente imutável, enquanto o *software* pode alterar-se facilmente.
- O termo *computador*, está hoje em dia associado a máquinas que operam de modo diferente consoante as necessidades e indicações recebidas do exterior, pelo que computador é um sistema, cujas tarefas diferem ao longo do tempo, em função das necessidades e requisitos de utilização, alterando-se apenas o componente de *software*.

# Computador como um sistema tecnológico



# Memória : Tipos

- Internas: construídas em torno de circuitos integrados
- Externas: construídas em torno de sistemas magnéticos ou ópticos (discos, DVDs)



# Memória Interna

Memórias internas → dois grandes grupos : RAMs e ROMs

## **RAM (Random-Access Memory)**

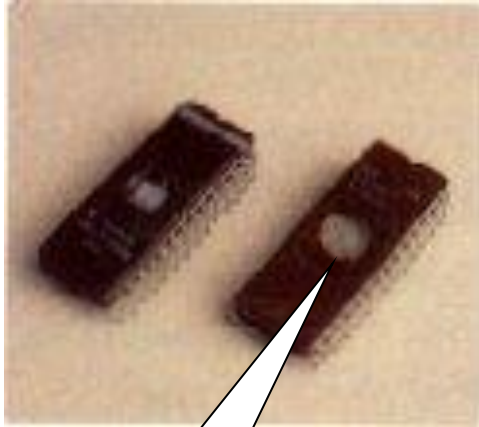
- O nome é pouco correto → significa que o tempo de acesso é igual para cada posição de memória
- Podem ser lidas e escritas um elevado número de vezes
- Voláteis – a informação perde-se quando se deixa de fornecer energia eléctrica

## **ROM(Read Only Memory)**

- Também é RAM pois o tempo de acesso é igual para cada posição de memória
- Podem ser programadas (uma ou mais vezes) mas normalmente são usadas apenas para leitura
- Existem variantes que podem ser usadas para leitura/escrita
- Não-voláteis – a informação continua armazenada quando se deixa de fornecer energia eléctrica
- Usadas na BIOS e para guardar configurações do utilizador

# ROM & RAM

ROM



Janela para  
desprogramar

RAM



Contactos  
eléctricos

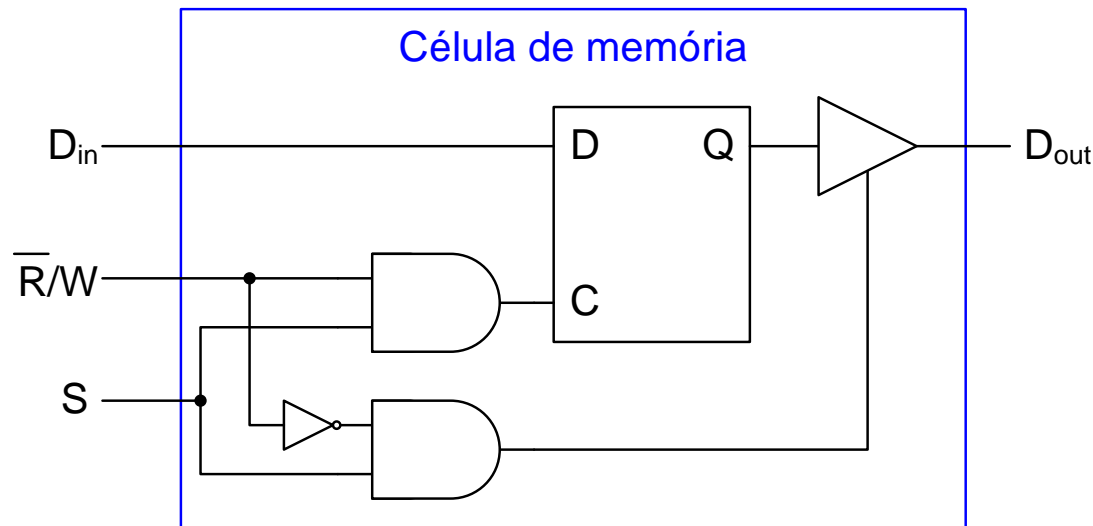


**SO-DIMM** – usadas em portáteis  
(*small outline dual in-line memory module*)

# Memória RAM : SRAM

## SRAM (*Static RAM*)

- Baseada em células de memória do tipo flip-flop
- Rápidas – tempos de acesso baixos para leitura e para escrita
- Caras (usam cerca de 6 transistores por célula, 1 bit)
- Utilizadas tipicamente como memórias *cache* (associadas ao processador)



$D_{in}$  – bit de entrada

$\overline{R/W}$  – sinal de Read/Write (0=read , 1=write)

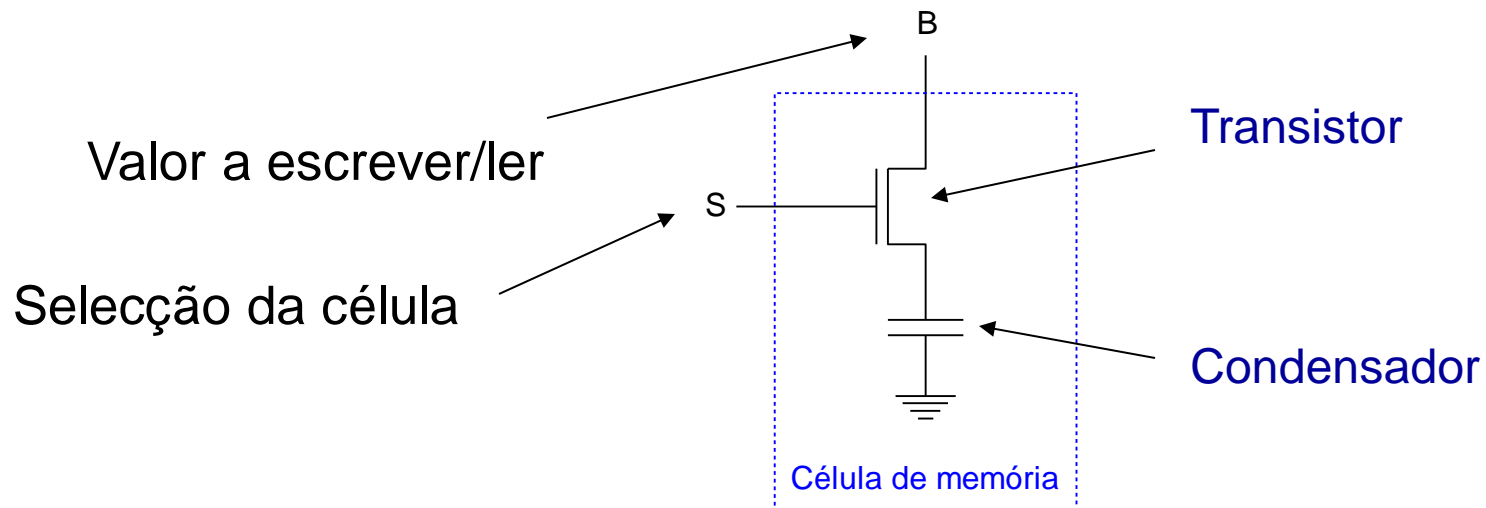
S – Strobe (habilita/desabilita a acção de read/write)

$D_{out}$  – bit de saída

# Memória RAM : DRAM

## Dinâmicas – DRAM (*Dynamic* RAM)

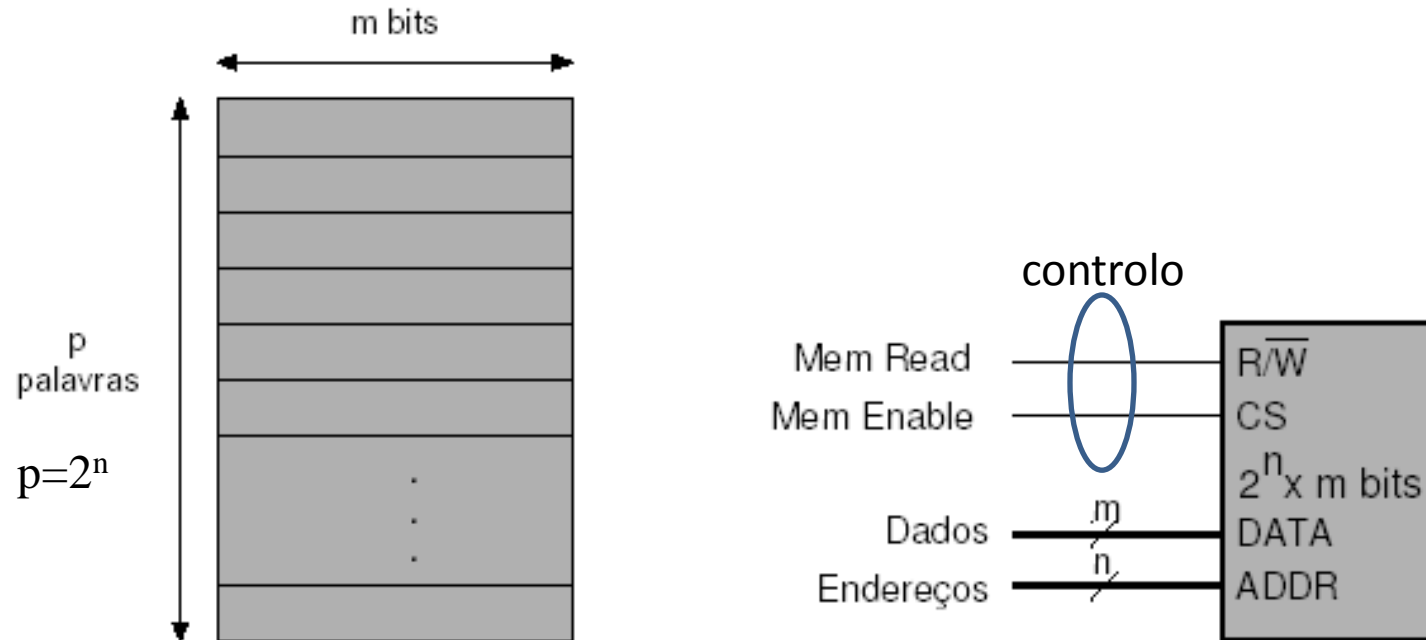
- Células de memória:
  - Pares transistor-condensador, que conseguem manter o nível lógico armazenado durante curtos espaços de tempo
  - Necessitam de ciclos de refrescamento periódicos para reposição dos níveis lógicos nos condensadores  $\Rightarrow$  mais lentas que as SRAMs
- Maior capacidade de armazenamento a menor custo
- Utilizadas como memória principal de um computador



# RAM : arquitectura

## Organização e capacidade de uma RAM (SRAM/DRAM)

- palavra =  $m$  bits
- $n$  linhas de endereço
- Capacidade (p palavras) =  $2^n$  endereços ou palavras de  $m$  bits ( $2^n * m$ )



RAM – *Random Access Memory*

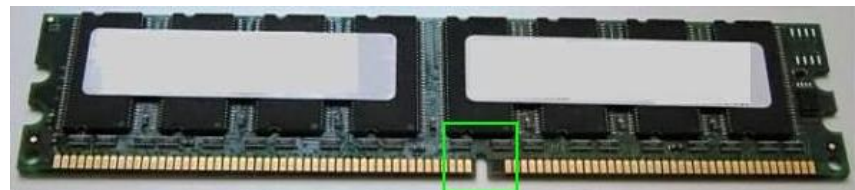
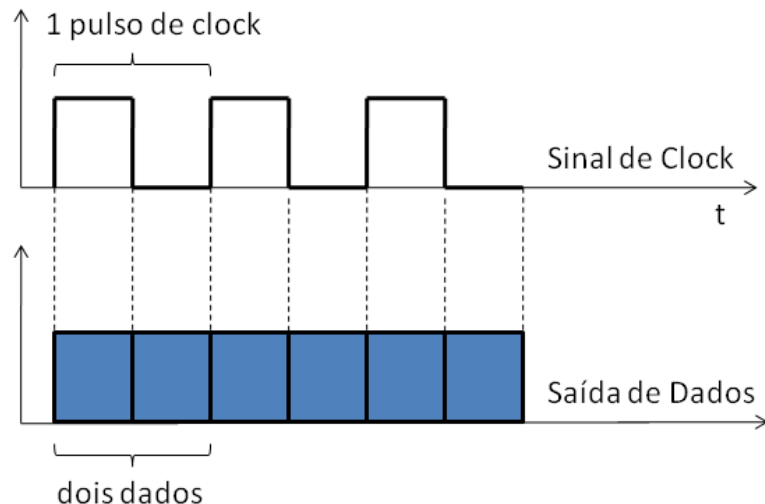
# DDR-SDRAM(*Double Data Rate SDRAM*) & DDR-SDRAM(*Double Data Rate SDRAM*)

**SDR SDRAM** (*Single Data Rate - Synchronous DRAM*) → trabalham de forma sincronizada com o processador, evitando problemas de atraso.



Módulo **DIMM** (*Double In-Line Memory Module*)

**DDR SDRAM** : transferem dois dados por impulso de clock → conseguem obter o dobro do desempenho das SDR-SDRAM para o mesmo clock (Ex: DDR a 100 MHz equivale a SDR a trabalhar a 200 MHz)



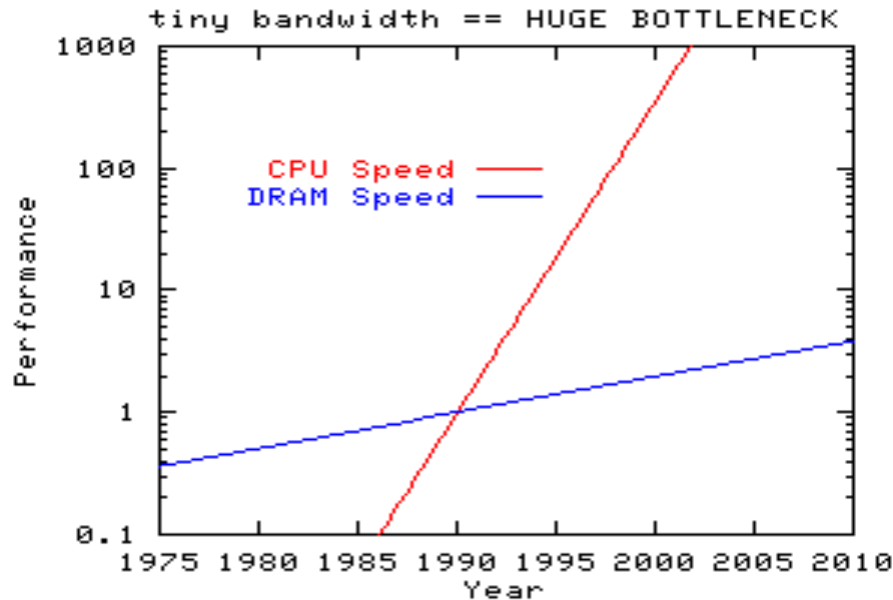
Módulo **DIMM** (*Double In-Line Memory Module*)

# Hiato Processador-Memória

- O desempenho dos micro-processadores tem vindo a aumentar a uma taxa de cerca de 60% / ano (desde 1986).
- O desempenho das memórias tem vindo a aumentar a uma taxa de perto de 10% / ano (diminuição do tempo de acesso)

DRAM: tempo de acesso ronda os 5 .. 60 ns

CPU:  $f_{\text{CPU}} = 1.0 \dots 3.0 \text{ GHz} \Rightarrow T_{\text{cc}} = 1.0 \dots 0.33 \text{ ns}$



The STREAM benchmark

<http://www.cs.virginia.edu/stream/ref.html>

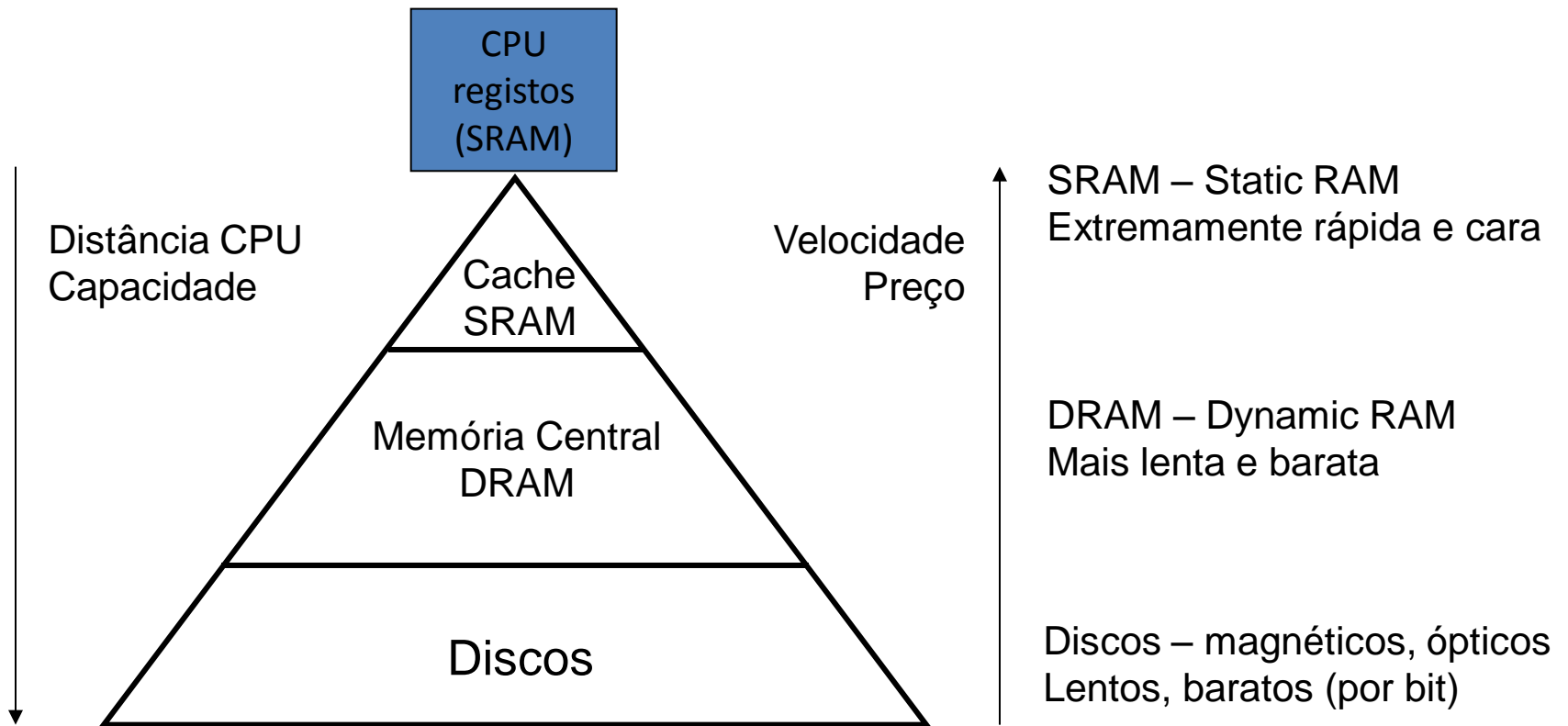
**"The Processor-Memory bottleneck: Problems and Solutions."**; Nihar R. Mahapatra and Balakrishna Venkatrao, ACM  
(<http://www.acm.org/crossroads/xrds5-3/pmqgap.html>)

**"The Memory Gap and the Future of High Performance Memories"**; Maurice V. Wilkes, ACM  
(<http://www.cl.cam.ac.uk/research/dtg/attarchive/pub/docs/URL/tr.2001.4.pdf>)

# Hierarquia de Memória

Como manter o processador alimentado com dados e instruções?

Técnica: dotar a máquina de vários níveis de memória, com diferentes propriedades, cada nível contém uma cópia do código e dados mais usados em cada instante.



Os dados contidos num nível mais próximo do processador são sempre um sub-conjunto dos dados contidos no nível inferior - o nível mais baixo contém a totalidade dos dados.



# Cache : princípio da localidade

**Localidade Temporal** – um elemento de memória acedido pelo CPU será provavelmente acedido em breve;  
**ex:** dentro de ciclos, instruções e variáveis usadas como contadores de ciclos, são acedidas repetidamente em curtos intervalos de tempo.

## **Acessos:**

- 1ª vez que um elemento de memória é acedido deve ser lido do nível mais baixo (p. ex., da memória central).
- 2ª vez que é acedido existem grandes hipóteses que se encontre na *cache*, evitando-se o tempo de leitura da memória central.

**Ideia:** manter na cache os últimos endereços acedidos.

**Localidade Espacial** – se um elemento de memória é acedido pelo CPU, então elementos com endereços próximos serão, com grande probabilidade, acedidos num futuro próximo.

**ex:** as instruções são acedidas em sequência, assim como, na maior parte dos programas os elementos dos *arrays*.

## **Acessos:**

- 1ª vez que um elemento de memória é acedido, deve ser lido do nível mais baixo (por exemplo, memória central) não apenas esse elemento, mas sim um **bloco** de elementos com endereços na sua vizinhança.
- Se o processador, nos próximos ciclos, aceder a um endereço próximo (ex.: próxima instrução ou próximo elemento de um *array*) aumenta a probabilidade de este estar na *cache*.

**Ideia:** carregar para a cache um conjunto de posições contíguas ao último endereço acedido.

# Cache

**Linha** – a cache está dividida em linhas. Cada linha tem o seu endereço (índice) e tem a capacidade de um bloco

**Bloco** – Quantidade de informação que é transferida de cada vez da memória central para a cache.

**Hit** – Diz-se que ocorreu um *hit* quando o elemento de memória acedido pelo CPU se encontra na cache.

**Hit rate** – Percentagem de *hits* ocorridos relativamente ao total de acessos à memória

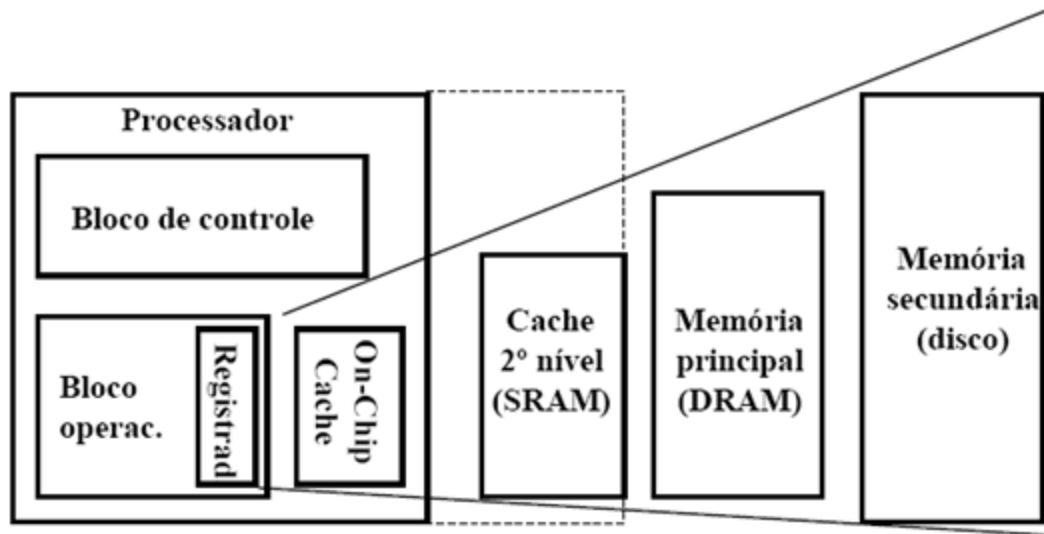
**Miss** – Diz-se que ocorreu um *miss* quando o elemento de memória acedido pelo CPU não se encontra na cache, sendo necessário lê-lo da memória central.

**Miss rate** – Percentagem de *misses* ocorridos relativamente ao total de acessos à memória.  $Miss\ rate = (1 - hit\ rate)$

Cache

	000
	001
	010
	011
	100
	101
	110
	111

# Cache: níveis



**L1** – dentro do CPU : pequenas dimensões

**L2** – fora do CPU : maiores dimensões