Arquitetura de Computadores

MIEI – 2021/22
DI-FCT/UNL

O Funcionamento Geral do CPU

Sumário

- Arquitetura de Von Neumman (revisitada)
- Funcionamento do CPU

Bibliografia:

Bryant, O'Hallaron Computer Systems: A Programmer's Perspective, 2nd ou 3rd Ed, secções 1.4.1 and 3.2

João M. Fernandes, Essentials of computing systems, Coleção Educação | Ciências, Engenharia e Tecnologia, Cap 2. secção 1.2 (https://doi.org/10.21814/uminho.ed.33)

Computador

- Equipamento que é **capaz de computar** (efetuar sequências de operações aritméticas, lógicas, ...), controlado por **programas internos**, **sem intervenção humana**.
- Deve ser **genérico** (ou universal), podendo-se mudar o programa para efetuar qualquer processamento

Programas

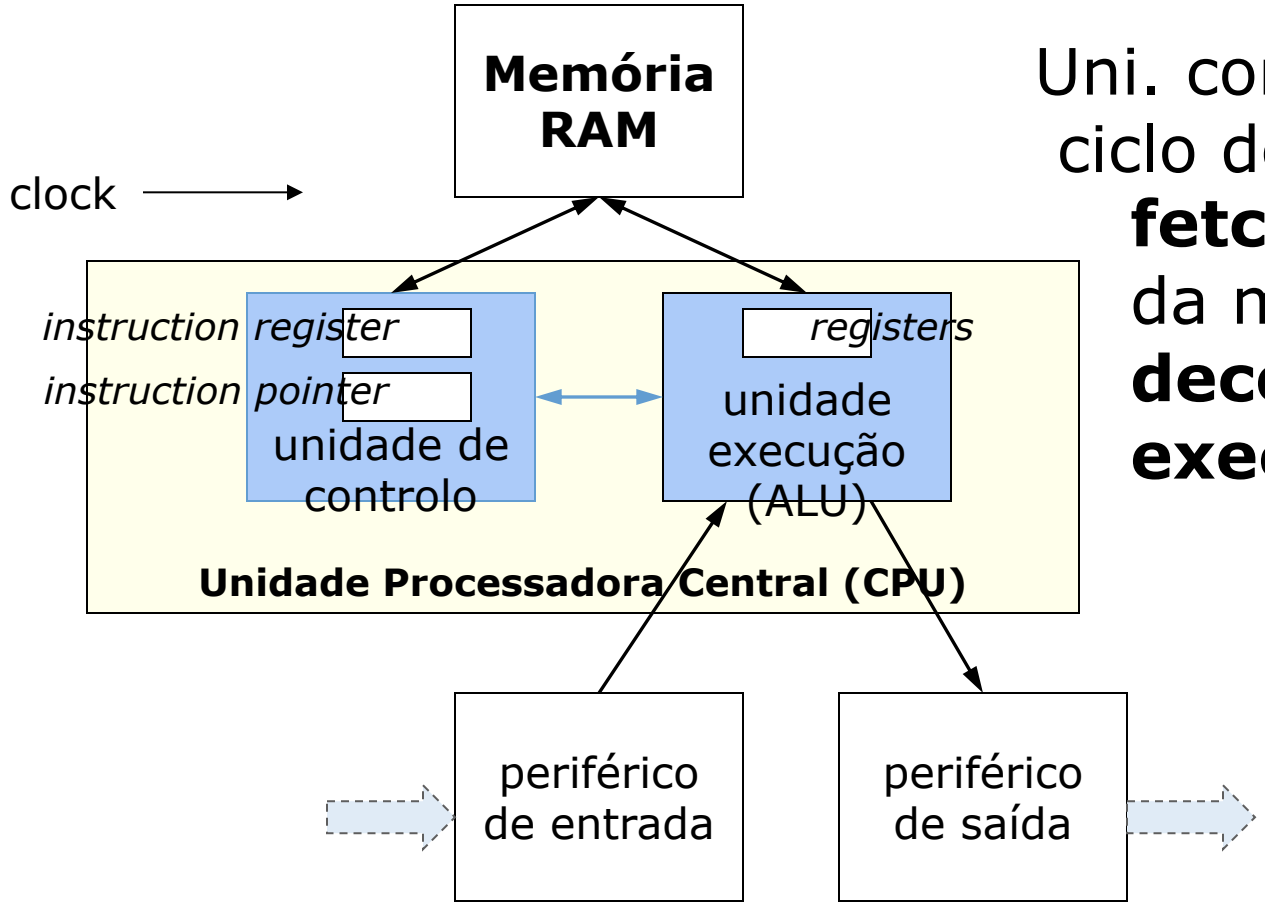
- Conjunto de instruções escritas numa linguagem que alguma "máquina" (real ou virtual) é capaz de reconhecer e interpretar
- Nível das instruções:
  - Mais próximas do domínio de aplicação: mais complexas e específicas
  - Mais próximas da arquitetura do computador: mais simples e genéricas
  - Existem linguagens a múltiplos níveis de abstração

Computadores com memória interna

- Von Neumann (e outros) propõe que o programa seja codificado numa memória tal como os dados
  - *Stored-program computer*

- A unidade central de processamento (CPU) necessita de novos componentes:
  - Para aceder a essa memória
  - Para interpretar esses programas

Arquitectura de Von Neumann (década 1940)



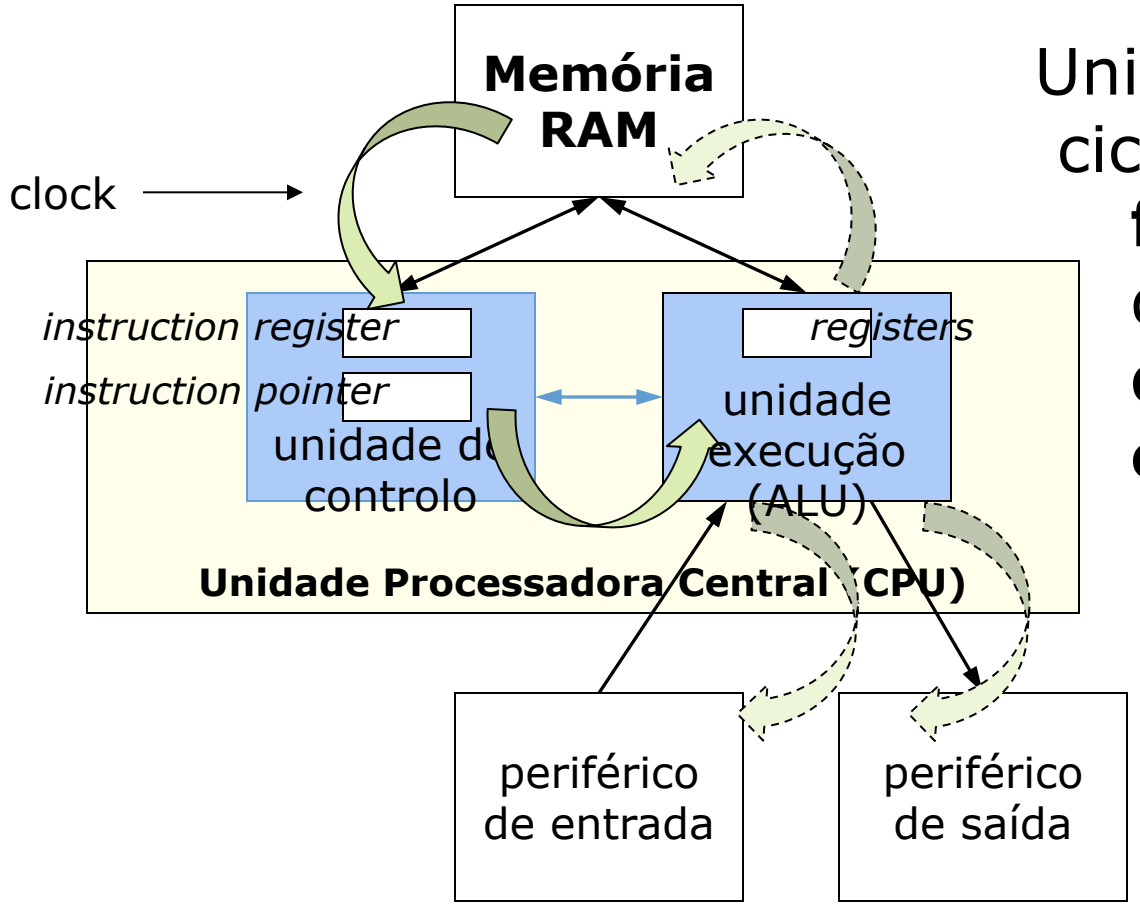
Uni. controlo executa ciclo de funcionamento:

fetch (obter instrução da memória)

decode (descodifica)

execute (executa)

Arquitectura de Von Neumann (década 1940)



Uni. controlo executa ciclo de funcionamento:

fetch (obter instrução da memória)

decode (descodifica)

execute (executa)

Processador - CPU

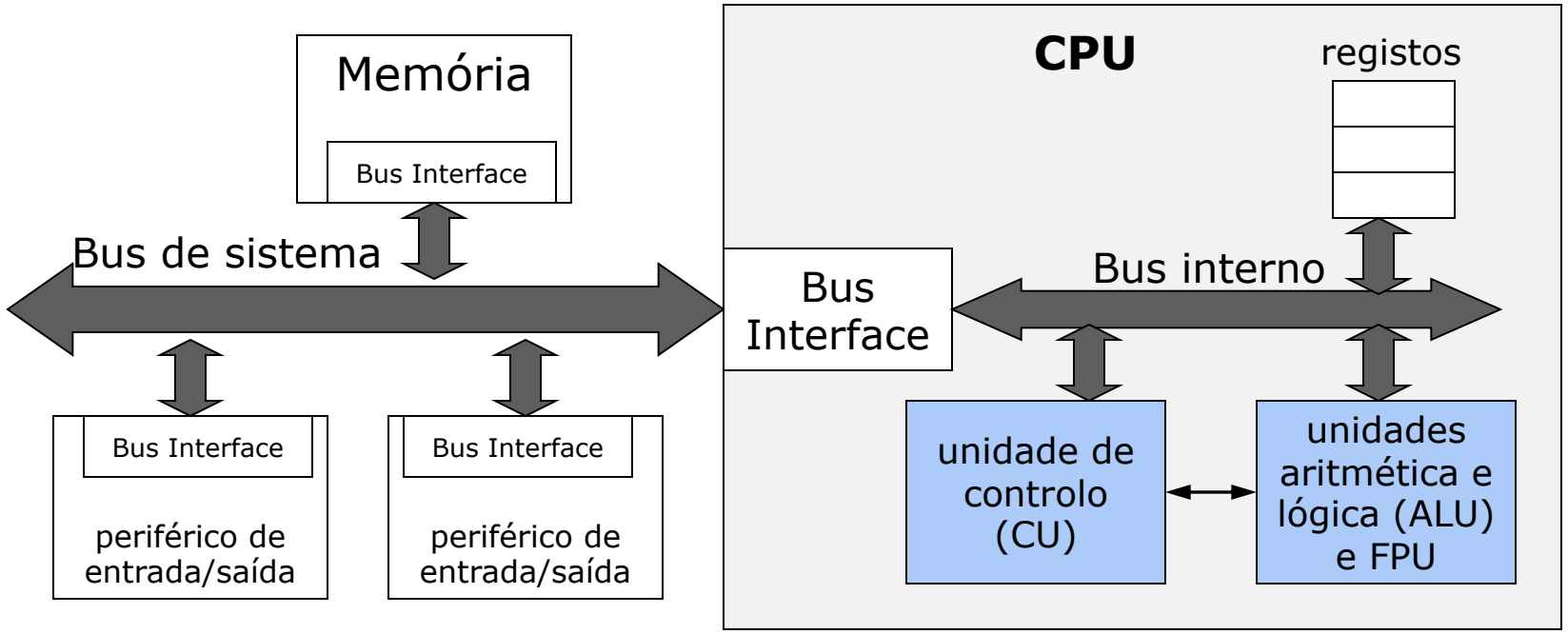
- Controlo global das operações do computador e responsável pela interpretação das instruções
- Contém:
  - ***Unidade de controlo***: obtém, descodifica e interpreta as instruções (uma de cada vez)
  - ***Unidade aritmética e lógica***: **ALU** – Arithmetic and Logic Unit
  - Possivelmente uma **FPU** – Floating Point Unit
  - Conjunto de **registos** (ou registadores): células de memória locais ao CPU, a usar pelas instruções e para manter valores intermédios

Ligações entre componentes

- Como os componentes comunicam?
  - componentes internos do CPU; componentes externos e CPU
- Quantas ligações?
  - Tantas quantos os bits a transportar de cada vez
- Cada componente ligado a todos os outros?
    - Complexo e dispendioso!
- Os componentes partilham um meio único: BUS
    - Seguem um protocolo para partilhar o BUS

Arquitectura de Von Neumann

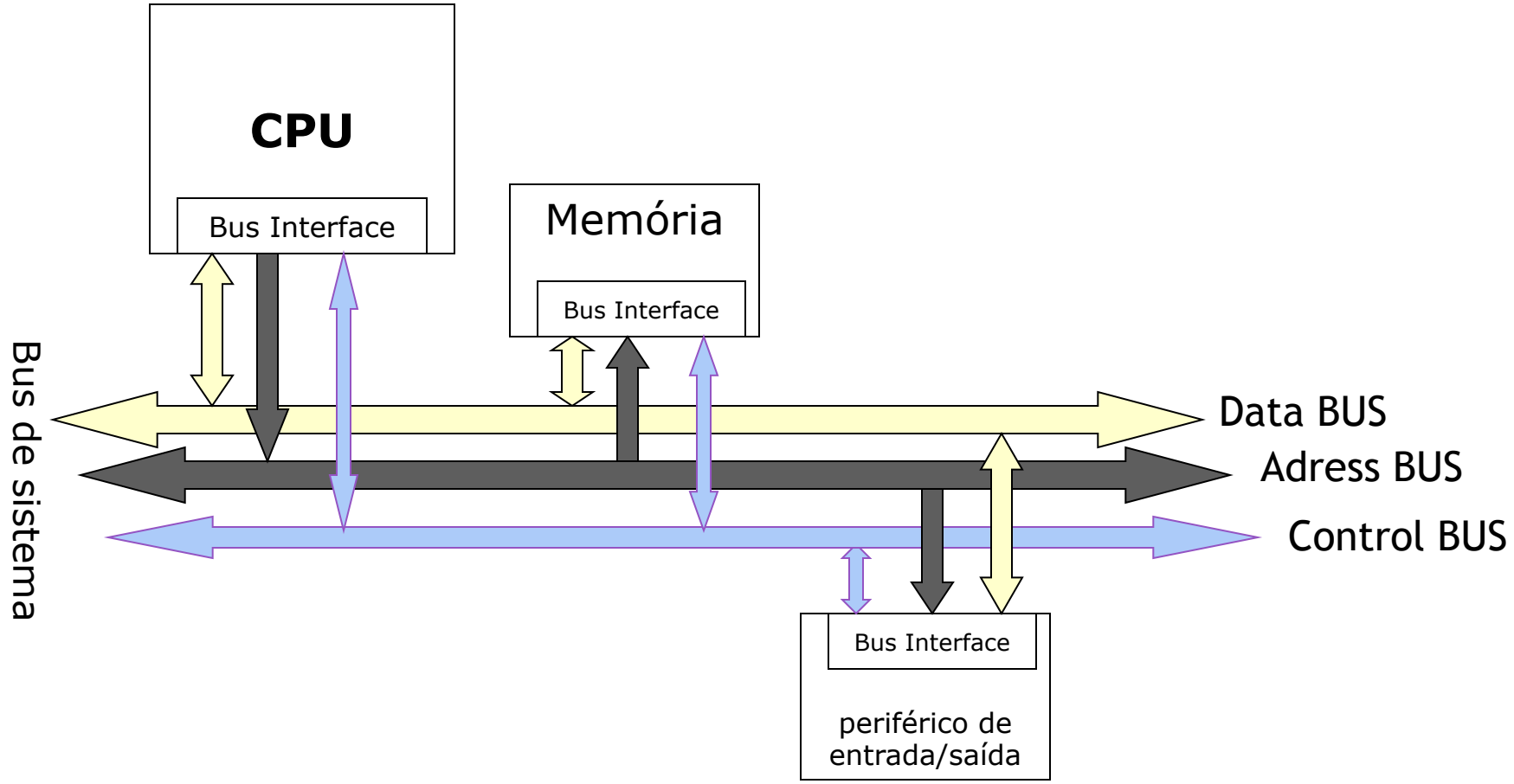
Actual (com BUS de sistema)



Os componentes podem ter diferentes ritmos de funcionamento

Diferentes relógios (*clocks*)

BUS de Sistema



Bus de Sistema

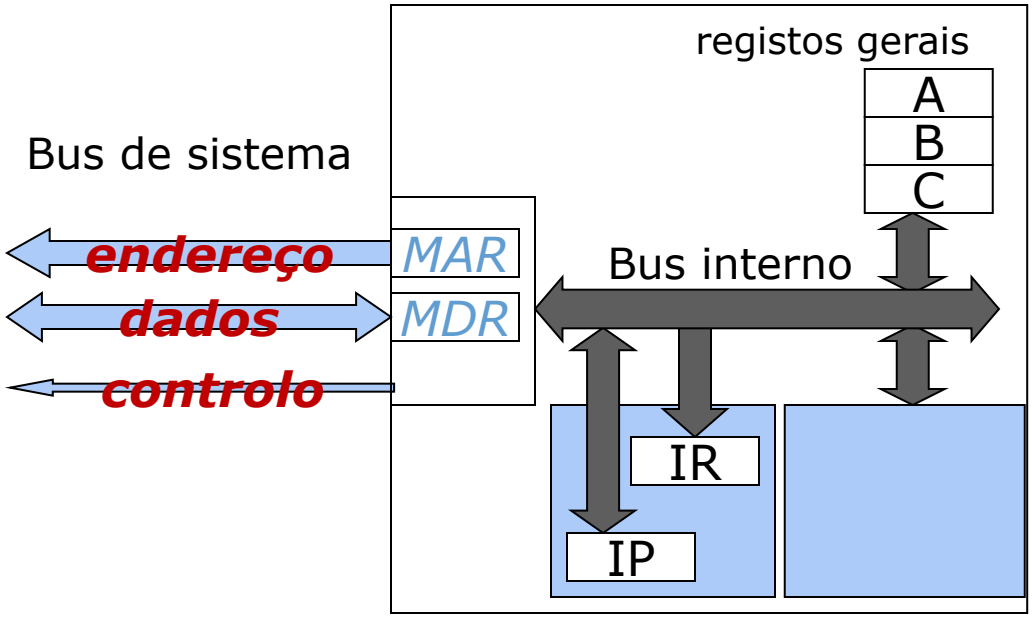
- Conjunto de linhas paralelas, cada uma codificando um bit
  - Número de linhas define a **largura** do Bus
  - 1 linha para dados → ligação **série**
- Bus de endereços:
  - conjunto de linhas que codificam a célula de memória ou a unidade periférica
- Bus de dados:
  - codificam os dados a transferir
- Bus de controlo:
  - para coordenar as transferências entre unidades,
  - sinais de comandos (por exemplo: ler/escrever, memória/periférico)

Código executado pelo CPU?

- Código máquina (instruções do ISA – Instruction Set Architecture)
  - ISA – conjunto de instruções que o CPU providencia para a sua programação (via software)

- Cada código: padrão de bits que define uma operação elementar, capaz de ser executada pela máquina (interpretada pelo CPU)
- Que instruções fazem parte do ISA?
  - Que operações são implementadas no *hardware*?
- Que informação faz parte do código de cada instrução?
  - Como descrevemos a operação e os seus operandos?

ISA vs Ações internas



- O CPU tem mais do que é visível pelo código máquina:
  - Mais registos e Micro Ações internas
  - Exemplo:

    ***Store Register to memory*:**

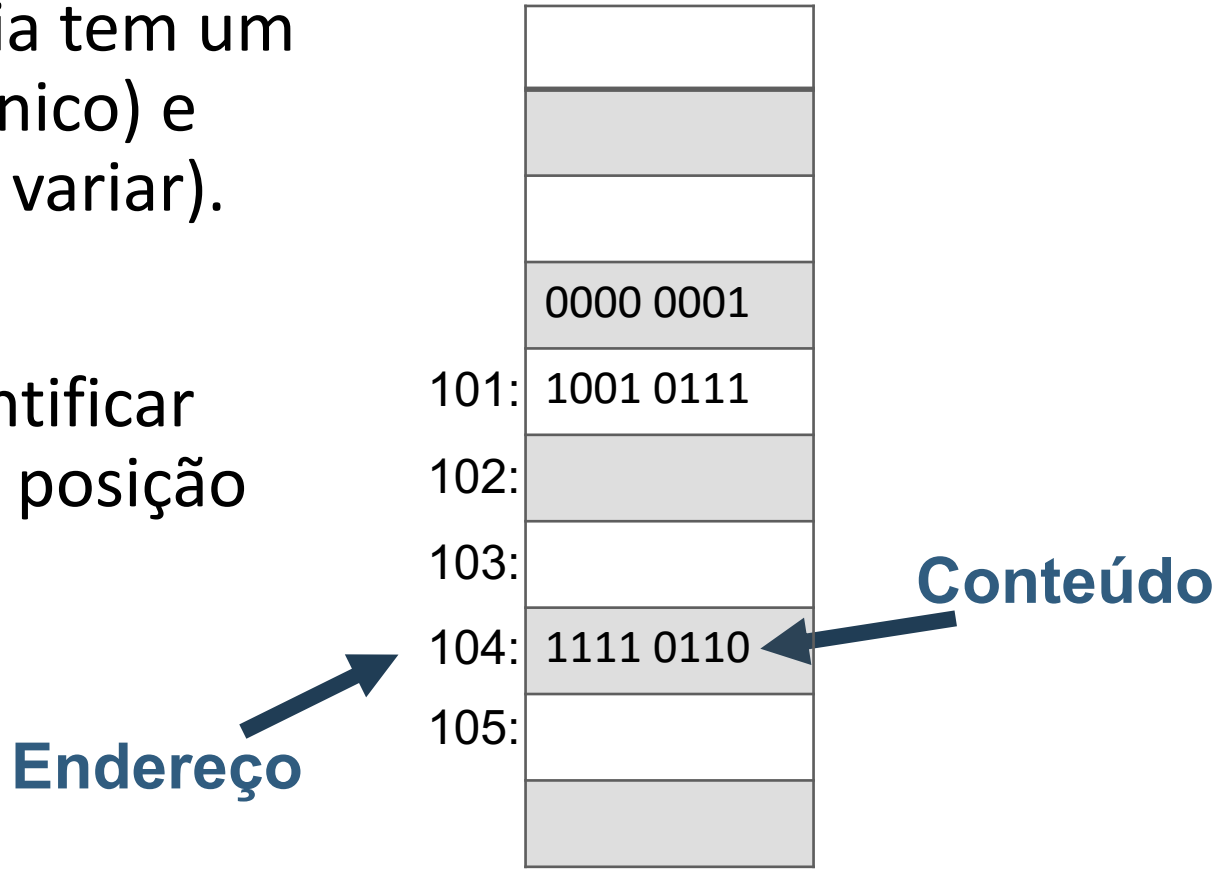
    MAR ← *endereço mem.*
    MDR ← conteúdo do reg.
    controlo ← escr. memória

  - RTN/RTL Register Transfer Notation/Language

Memória Central (RAM)

Cada posição de memória tem um **endereço** (que é fixo e único) e um **conteúdo** (que pode variar).

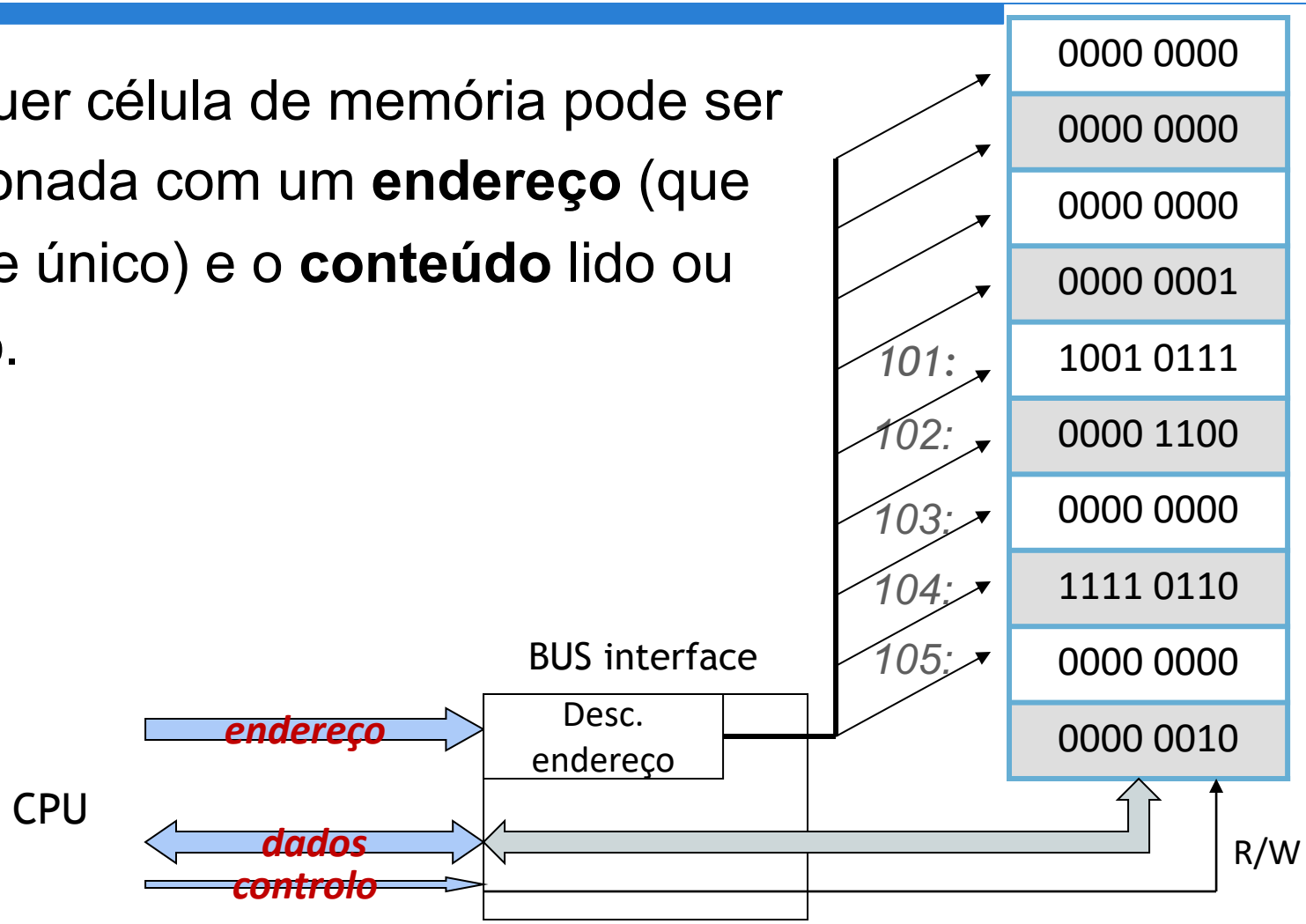
O **endereço** permite identificar (sem ambiguidade) cada posição da memória.



O conteúdo da posição de memória com o endereço **104** é **1111 0110**

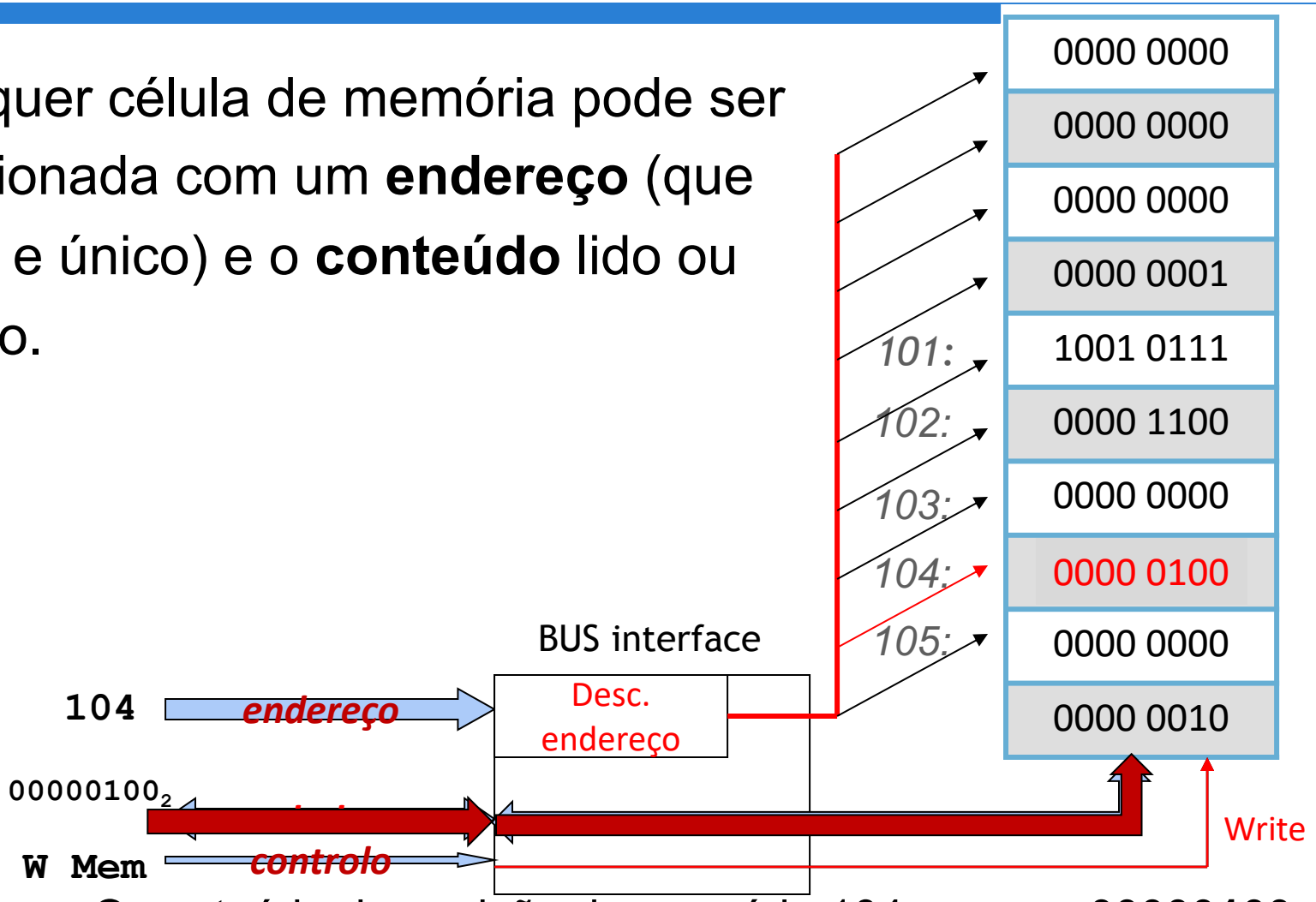
Memória Central (RAM)

Qualquer célula de memória pode ser selecionada com um **endereço** (que é fixo e único) e o **conteúdo** lido ou escrito.



Memória Central (RAM)

Qualquer célula de memória pode ser selecionada com um **endereço** (que é fixo e único) e o **conteúdo** lido ou escrito.



O conteúdo da posição de memória 104 passa a **00000100_2**

Memória Central (RAM)

- Guarda grupos de bits que representam instruções ou dados
  - Cada célula é normalmente um byte (8 bits) ou uma “palavra”

- É acedida como um vetor:
  - Mem[0]…Mem[n-1] (capacidade = n células)
  - O endereço da célula corresponde ao índice i em Mem[i]
  - O endereço é representado em binário, como um número inteiro sem sinal
  - Pode ser lido/escrito em grupos de células
  - O acesso é **direto**: dá-se i para aceder a partir de Mem[i]
    - RAM: *Random Access Memory*

Variáveis na memória

- Variáveis são designações simbólicas para posições de memória:

```
int A = 5;
int B = -2;   // 0xFFFFFFFE
```

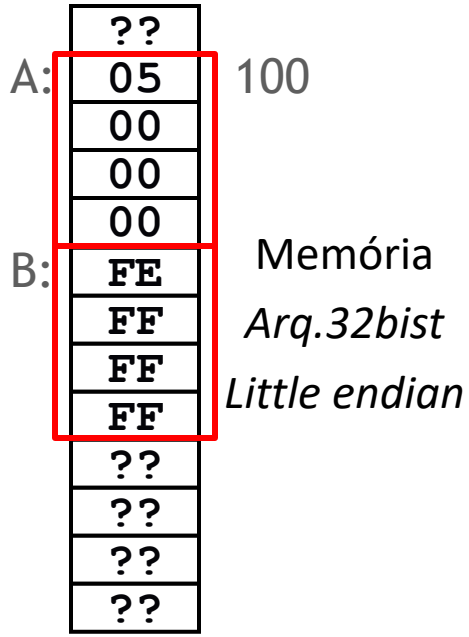


Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes

Variáveis na memória

- Variáveis são designações simbólicas para posições de memória:

```
int A = 15213;    // 0x00003B6D
int B = -15213;
long C = 15213;
long long D = 15213;
```

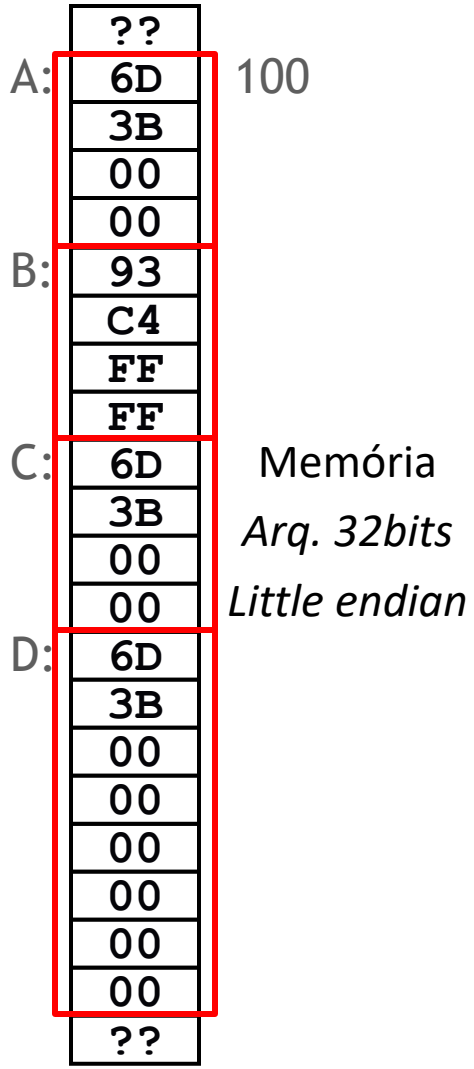
Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes

C: 108 : 4 bytes

D: 112 : 8 bytes



Variáveis na memória

- Qual o código produzido pelo compilador se:

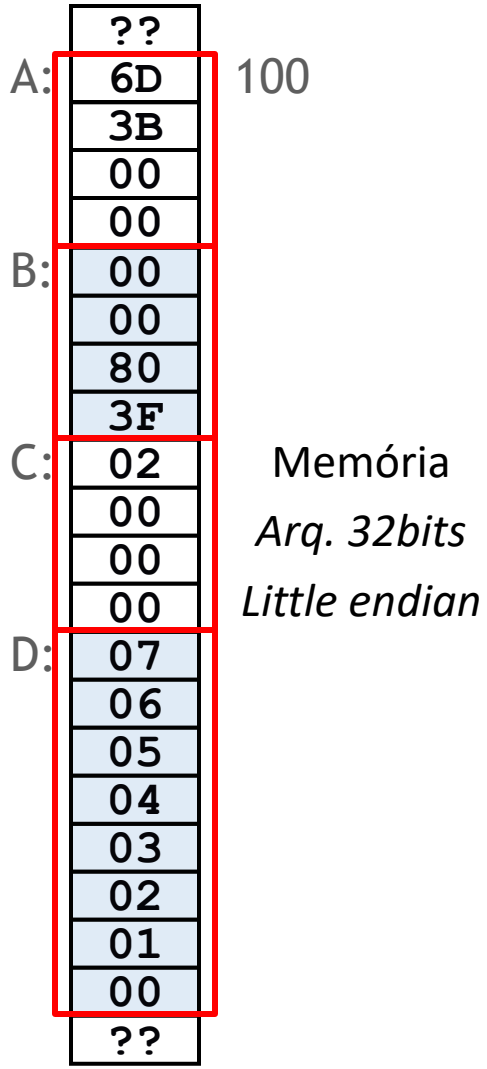
```
int A = 15213;    // 0x00003B6D
float B = 1.0;    // 0x3F800000
int C = 2;
long long D = 0x01020304050607;
A = D; //Não pode copiar os 8 bytes para A!
A = B;  //Não é só copiar!
```

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes

D: 112 : 8 bytes



Variáveis na memória

- Qual o código produzido pelo compilador se:

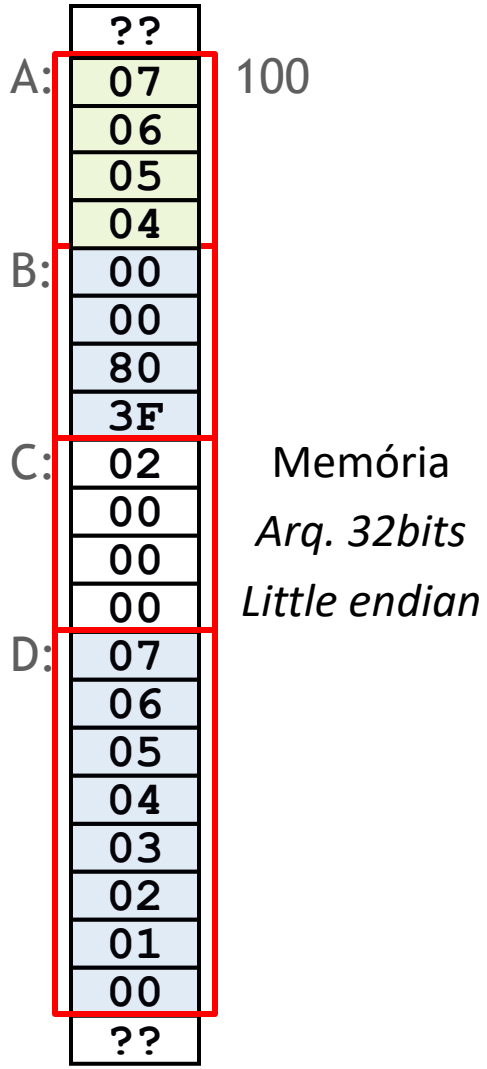
```
int A = 15213;    // 0x00003B6D
float B = 1.0;    // 0x3F800000
int C = 2;
long long D = 0x01020304050607;
A = D; //Não pode copiar os 8 bytes para A!
A = B;  //Não é só copiar!
```

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes

D: 112 : 8 bytes



Variáveis na memória

- Qual o código produzido pelo compilador se:

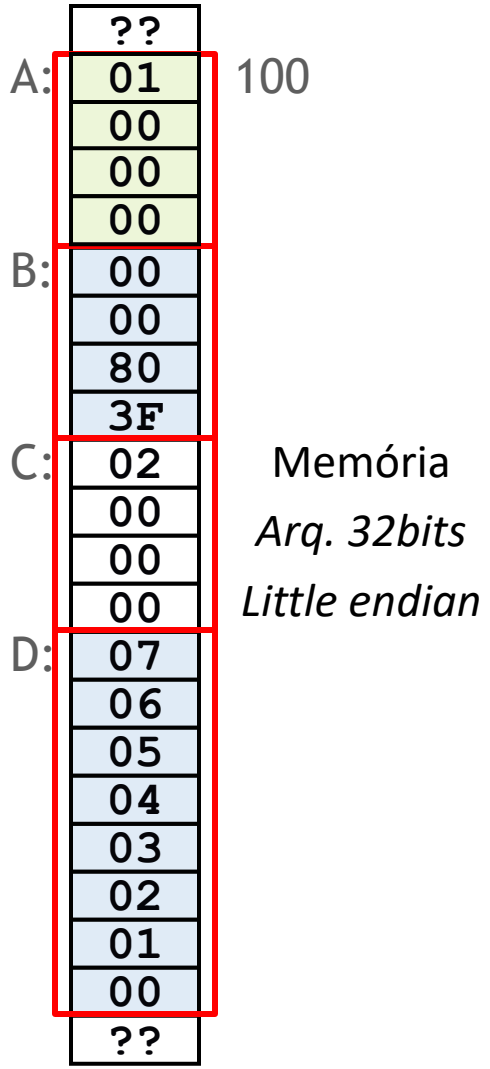
```
int A = 15213;    // 0x00003B6D
float B = 1.0;    // 0x3F800000
int C = 2;
long long D = 0x01020304050607;
A = D; //Não pode copiar os 8 bytes para A!
A = B;  //Não é só copiar!
```

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes

D: 112 : 8 bytes



Referências para memória - C

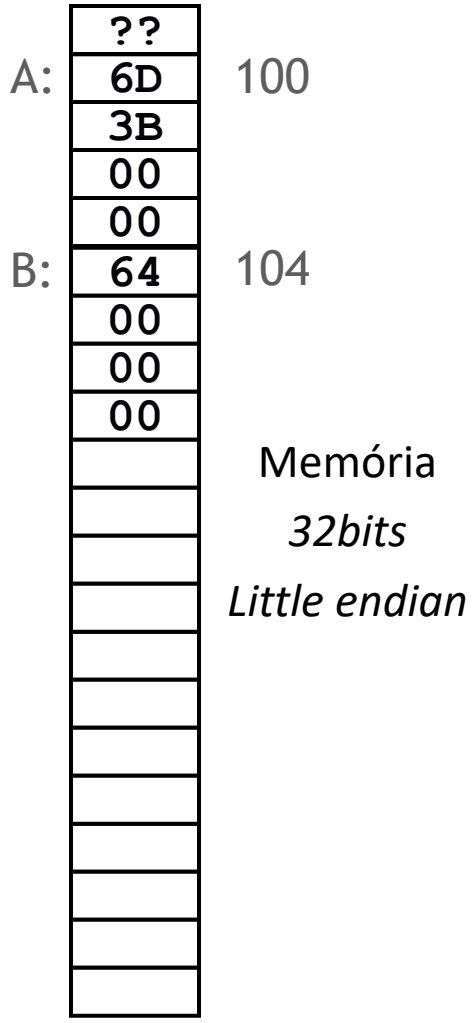
- Variável B refere A, guarda endereço de A, ou aponta para A:

```
int A = 15213;     // 0x00003B6D
int *B = &A;       // 100 = 0x00000064
```

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes ← *arq. end. 32 bits (8 bytes se 64 bits)*



Referências para memória - C

- Variável B refere A, guarda endereço de A, ou aponta para A:

```
int A = 15213;     // 0x00003B6D
int *B = &A;       // 100 = 0x00000064
```

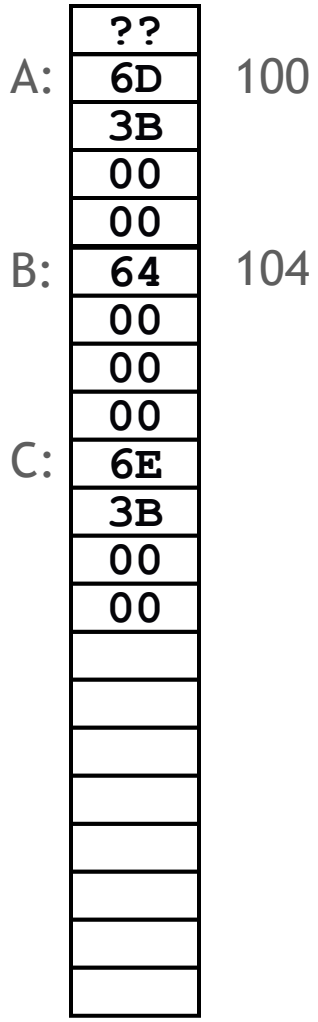
```
int C = *B + 1;
```

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes ← *arq. end. 32 bits (8 bytes se 64 bits)*

C: 108 : 4 bytes



Referências para memória - C

- Variável B refere A, guarda endereço de A, ou aponta para A:

```
int A = 15213;     // 0x00003B6D
int *B = &A;       // 100 = 0x00000064
```

int C = *B + 1;

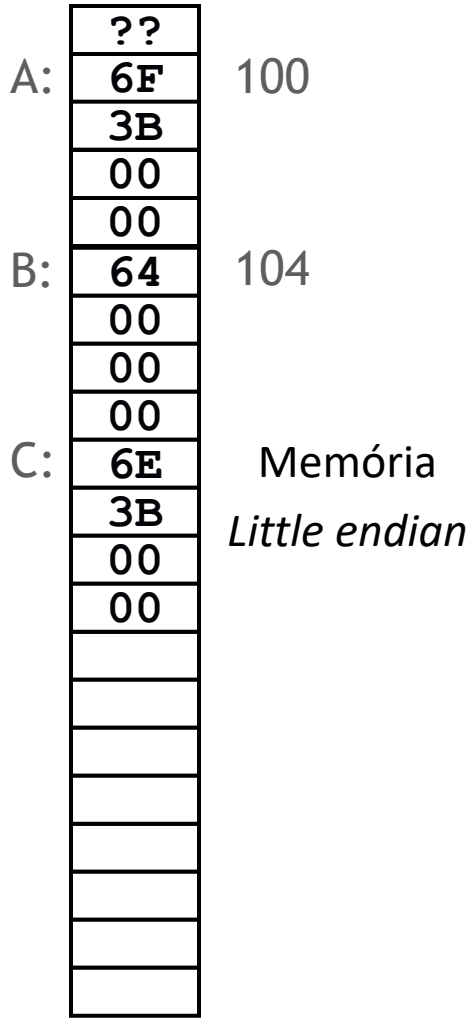
*B = *B + 2;

Tabela de símbolos do compilador:

A: 100 : 4 bytes

B: 104 : 4 bytes ← *arq. end. 32 bits*

C: 108 : 4 bytes



Referências para memória - C

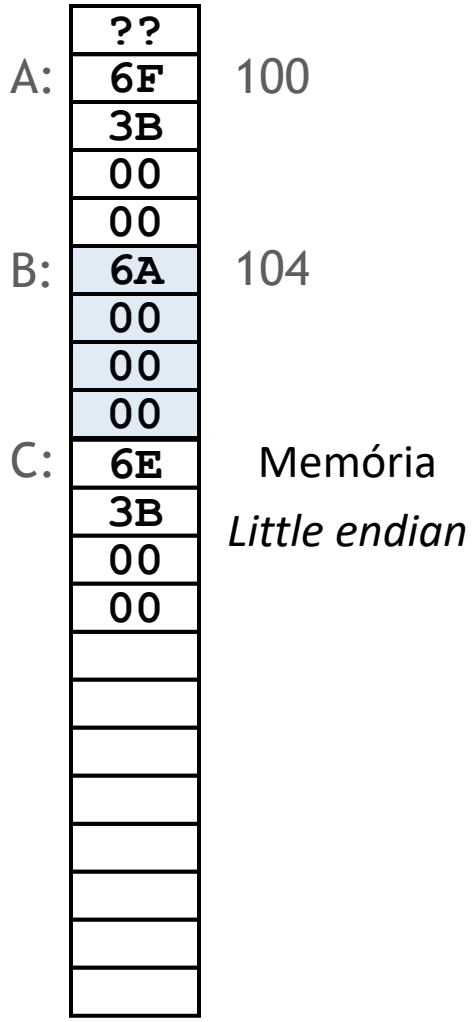
- Variável B refere A, guarda endereço de A, ou aponta para A:

```
int A = 15213;    // 0x00003B6D
int *B = &A;      // 100 = 0x00000064
```

```
int C = *B + 1;
*B = *B + 2;
```

- Já vimos que em C podemos usar expressões com apontadores:

```
B = B + 1;  // B passa para o próximo int
*B = *B + 2;  // 0x68 = 104
```



Convertendo entre tipos de dados (!)

- Com o *cast* de apontadores podemos "ignorar" os tipos de dados
  - ver a memória do *int* como se fosse *float* (copia bits de x para f):

```
int x = 3;
float f = *(float*)&x;
```

  - ver a memória do *float* como se fosse *int* (copia bits de f para x):

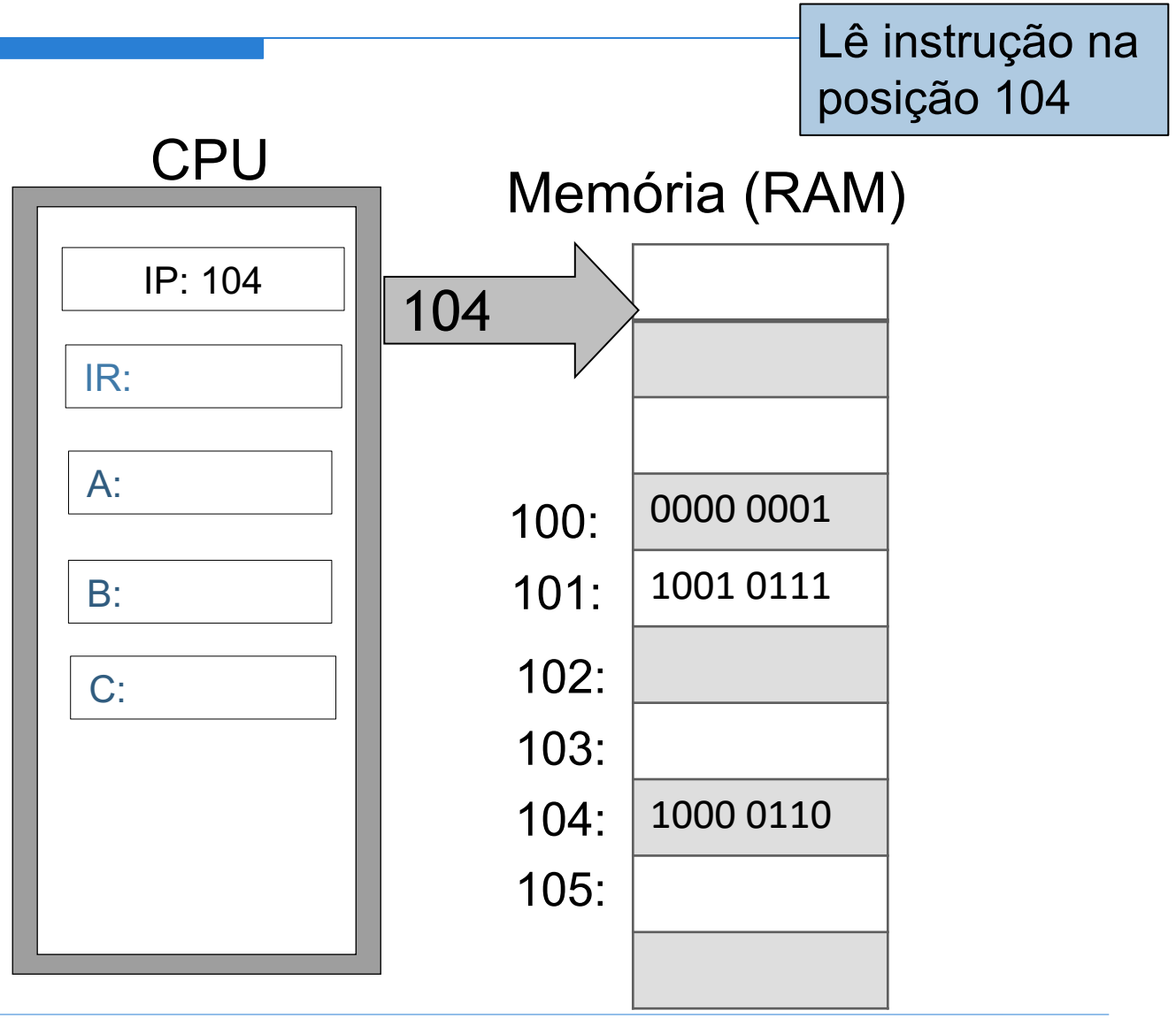
```
f = 1.5;
x = *(int*)&f;
```

CPU: Ciclo de Execução

- A Unidade de controlo do CPU:
  - Obtém a próxima instrução de memória (usa um índice para o programa: PC ou IP) → **fetch**
  - Incrementa IP (IP ← IP + tamanho_da_instrução)
  - Descodifica a instrução → **decode**
  - Emite os sinais de controlo, na micro-arquitectura, correspondentes ao encadeamento de ações necessárias para executar a instrução e as transferências de informação necessárias → **execute**
  - Volta ao início

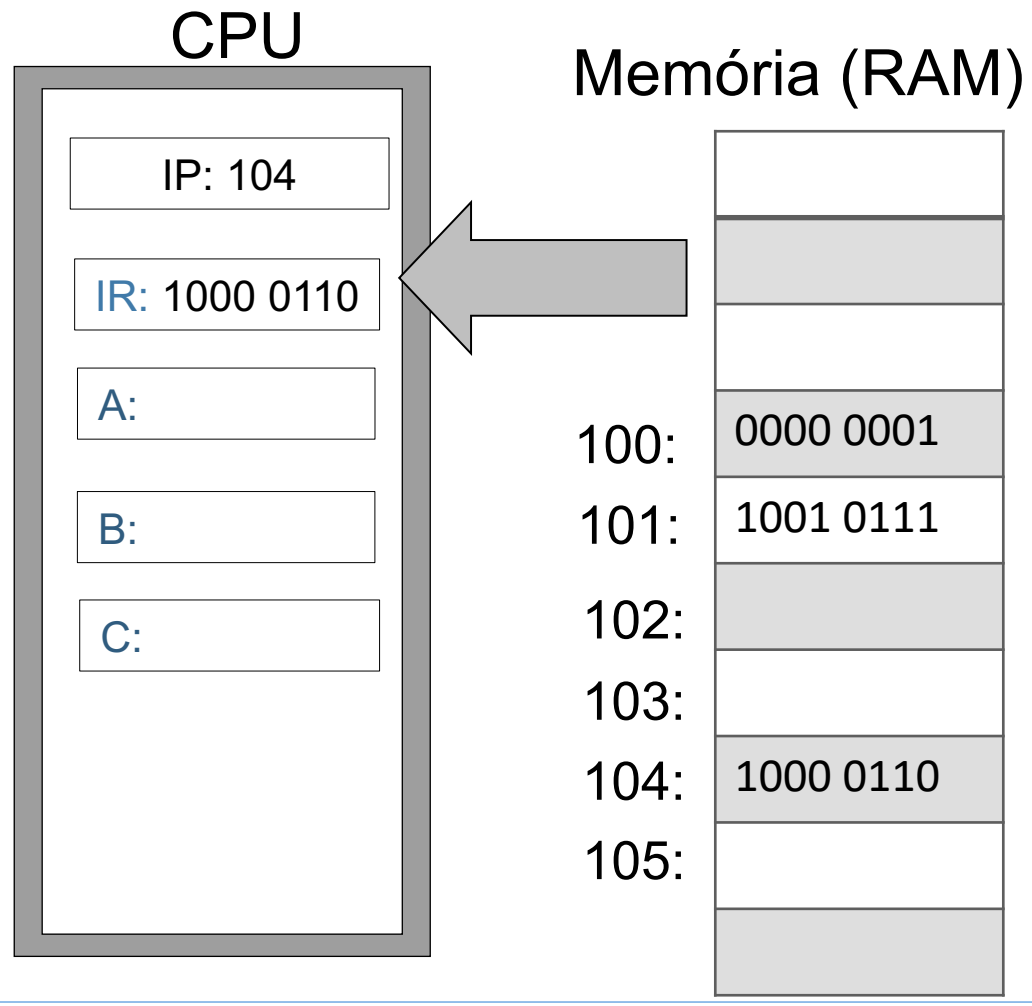
Funcionamento do CPU: FETCH

- O CPU executa as instruções guardadas na memória central, de uma forma sequencial.
- Em cada momento, o CPU mantém a posição de memória da instrução que está a executar.

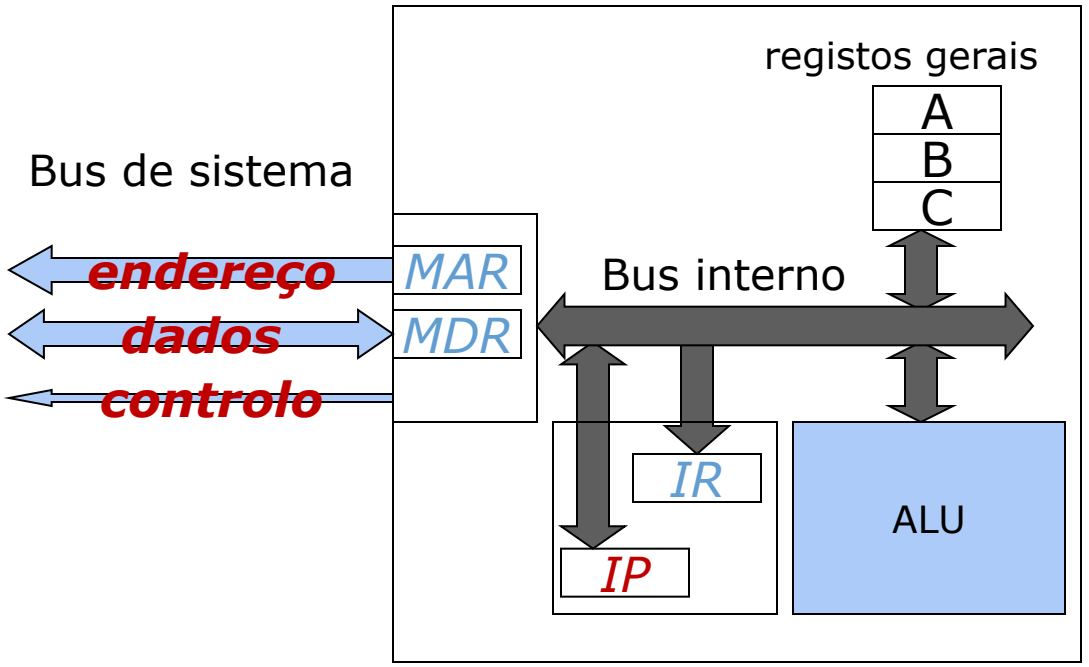


Funcionamento do CPU: FETCH

- O CPU executa as instruções guardadas na memória central, de uma forma sequencial.
- Em cada momento, o CPU mantém a posição de memória da instrução que está a executar.



Micro-ações do FETCH



ciclo de funcionamento:

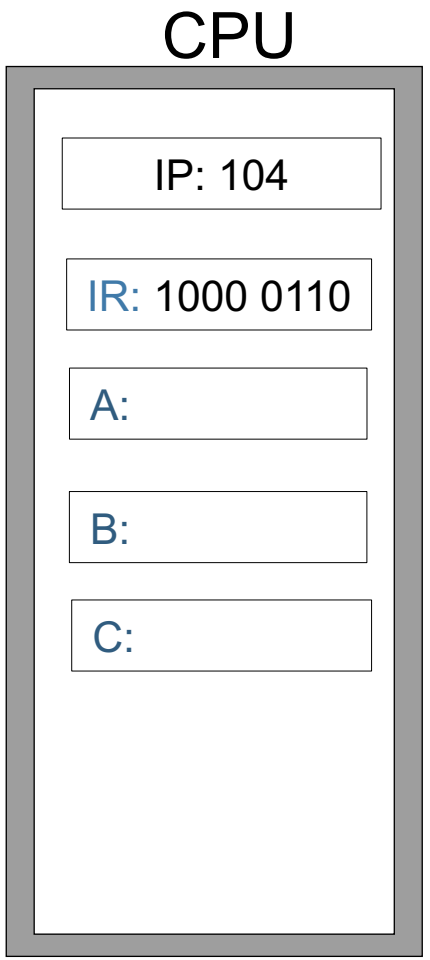
fetch (obter instrução da memória)
decode (descodifica)
execute (executa)

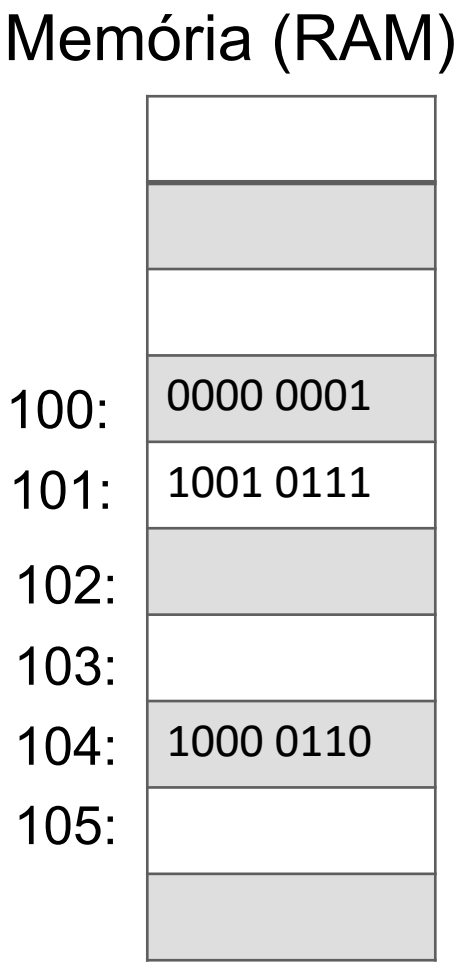
***fetch*: IR = mem[IP]**
MAR ← IP
controlo ← lêr memória
MDR ← Mem[MAR]
IR ← MDR

Nota: IP também é chamado PC

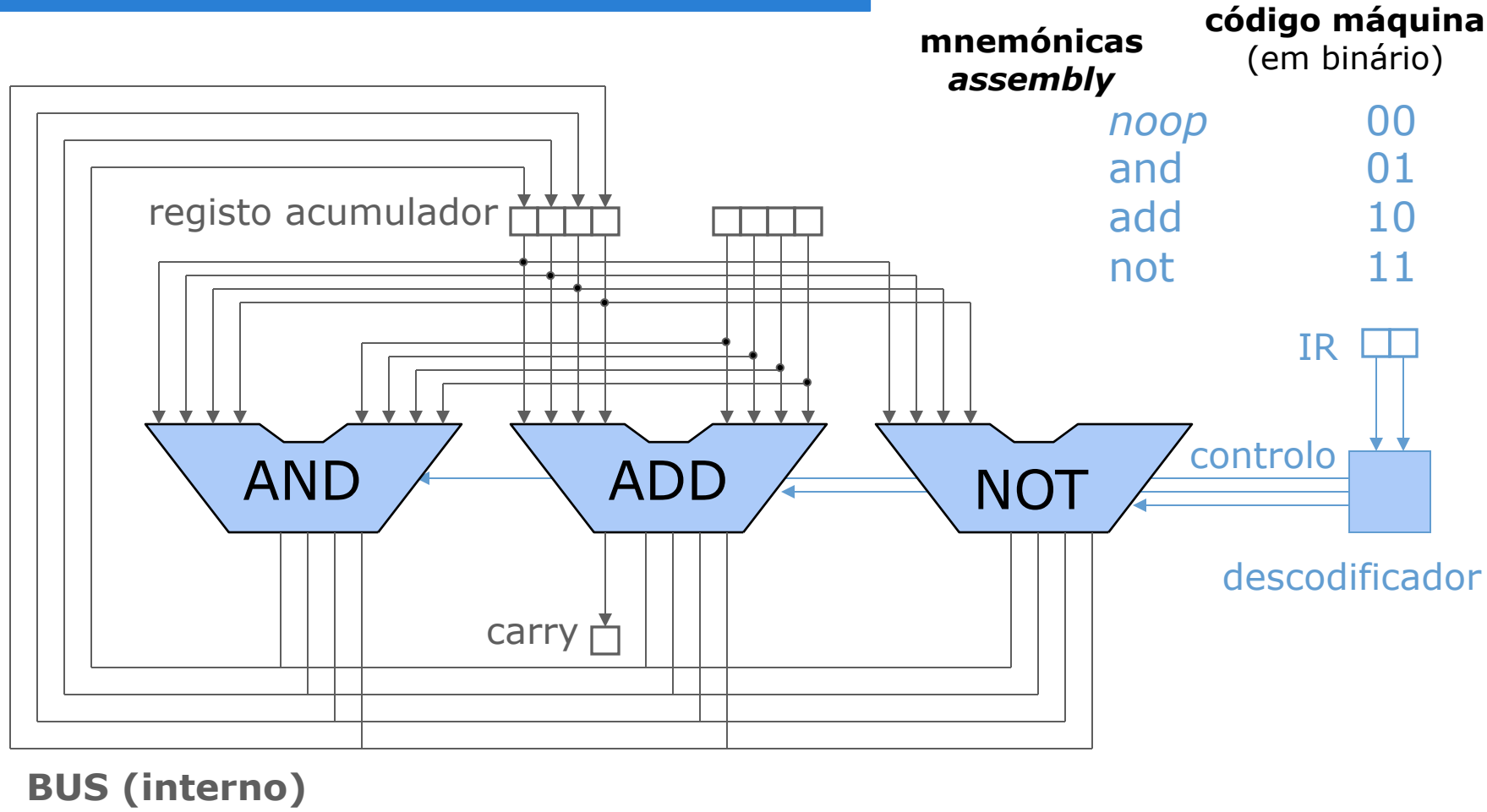
Funcionamento do CPU: DECODE

- A instrução no registo **IR** define a ação elementar a executar
- Para executá-la é preciso descoficar o conteúdo de IR
- Exemplo:
  - 2 bits para a operação:
    noop → 00
    and → 01
    add → 10
    not → 11
  - 2 bits para cada registo: A, B e C têm, respetivamente, identificadores 0, 1 e 2
  - Qual é a instrução representada por 1000 0110?
    add A B C





Descodificação e execução de instruções de 2bits e palavras de 4 bits



Muito simplificado!

Execução – tipos de instruções

- A execução de uma instrução pode envolver:
  - Operações aritméticas e lógicas (pela ALU) ou FP
    - A ALU e FPU operam sobre operandos na instrução ou em registos
    - ADD, OR, NOT, ...
  - Transferências reg/CPU ↔ Memória
    - Load/Store ou MOV
  - Transferências reg/CPU ↔ Periféricos
    - IN/OUT
  - Controlo da sequência de execução de instruções → alterar o valor em IP
    - Jump

Exemplo com o Assembly do Simulador

Programa em *assembly*:

```
.data
    .w 4
    .w 5
.code
   mov (0), r1
   mov (2), r2
   add r1, r2, r2
   mov 0, r1
   int 0
```

Programa em memória (código máquina e dados, assumindo dois espaços de endereçamento distintos):

Dados:

00 **04 00**
02 **05 00**

Código:

00 **00 00**
02 **00 11**
04 **82 09**
06 **40 00**
08 **c0 00**

O *assembly* é uma representação legível por humanos para um ISA

Execução do **`mov(0), r1`**

- Suponhamos que IP (ou PC) tem valor **0**
- O fetch carrega o valor **0000** para o IR
- Descodificação →
  - type = 0 (MEMORY),
  - opcode = 0 (LOAD)
  - address = 0
  - register = 0 (r1)
- Execução:

  registers[0] ← data[0]data[1] ou seja registers[0] ← 4

Execução do **`add r1, r2, r2`**

- Suponhamos que IP (ou PC) tem valor **4**
- O fetch carrega o valor **8209** para o IR
- Descodificação →
  - type = 2 ALU),
  - opcode = 0 (ADD)
  - size = 1 (WORD)
  - reg_in_1 = 0 (r1), reg_in_2 = 1 (r2), reg_out = 1 (r2)
- Execução:

  registers[1] ← registers[0] + registers[1] ou seja

  registers[1] ← 4 + 5

Desenho de uma Arquitectura

- Definição das instruções: para as boas soluções, contribuem:
  - A definição do conjunto de instruções a suportar
  - E todos os outros elementos da arquitectura: registos, bus, etc

- Uma boa solução é também fruto de compromissos:
  - Tecnologia
  - Custo
  - Funcionalidade
  - Desempenho
  - Leis do mercado

Capacidades das arquitecturas

- O número de bits nos endereços (p.e. IP) determina a capacidade máxima de memória endereçável
  - Logo o maior programa, de código e dados
  - O número de linhas no bus de endereços determina a capacidade máxima de memória realmente acessível
- O número de bits dos registos gerais determina o tamanho dos dados operados pelas instruções
- O número de linhas no bus de dados determina a capacidade máxima de transferência dos dados em cada ciclo de acesso à memória

Desenho das instruções do processador

- Os objectivos considerados são muitos:
  - Aproximar das linguagens de alto nível
  - Reduzir o tamanho dos programas
  - Oferecer instruções poderosas
  - Instruções simples, que facilitem a implementação eficiente
  - Oferecer só as instruções mais úteis
  - Oferecer diversos modos de endereçar as variáveis em memória (ex. referências, matrizes, registos)
  - Reduzir o número de acessos a memória central
  - Instruções mais complexas não são oferecidas, mas podem ser obtidas à custa de várias das existentes

Tamanho do endereço

- Palavras de memória que se pode endereçar (teoricamente):
  - 8 bits – 2^8 endereços =256 endereços
  - 16 bits – 2^{16} = 64 K
  - 32 bits – 2^{32} = 4 G
  - 64 bits – 2^{64} = 16 E
- Se memória endereçada ao byte:
  - 32 bits → 4 GBytes
- Se a memória endereçada por palavras de 2 bytes:
  - 32 bits → 8 GBytes

Múltiplos

- Nota sobre os unidades em bases 2 vs base 10 (exemplo com bytes):

Decimal			**Binary**				
Value	Metric		Value	IEC		*JEDEC*	
1000 (10^3)	KB	kilobyte	**1024 (2^{10})**	KiB	kibyte	***KB***	***kilobyte***
1000^2	MB	megabyte	**1024^2**	MiB	mebibyte	***MB***	***megabyte***
1000^3	GB	gigabyte	**1024^3**	GiB	gibibyte	***GB***	***gigabyte***
1000^4	TB	terabyte	**1024^4**	TiB	tebibyte	***TB***	***terabyte***
1000^5	PB	petabyte	**1024^5**	PiB	pebibyte	***PB***	***petabyte***
1000^6	EB	exabyte	**1024^6**	EiB	exbibyte	***EB***	***exabyte***
1000^7	ZB	zettabyte	**1024^7**	ZiB	zebibyte	***ZB***	***zettabyte***
1000^8	YB	yottabyte	**1024^8**	YiB	yobibyte	***YB***	***yottabyte***

Sincronizar os circuitos

- Os circuitos funcionam ao ritmo de um relógio:
  - marca com impulsos eléctricos o ritmo de funcionamento de todos os componentes (transições de estado)
- Marca quando os componentes podem interactuar
  - Garante que os vários sinais (bits) são enviados e recebidos no instante devido
    - Ex: um componente não lê o que está no BUS antes que algo válido lá tenha sido colocado
- O relógio gera um determinado número de impulsos por segundo: ciclos/segundo ou frequência (Hz)

Relógio vs velocidade do computador

- A frequência do relógio não é o único factor na velocidade de funcionamento de um computador!
- O tempo de execução de determinado programa depende de:
  - Número de instruções (tamanho do programa)
  - Tempo de execução de cada instrução (número de ciclos de relógio)
  - Tempos de espera pela memória
  - Tempos de espera pelas Entradas/Saídas

Ordem de tempos de acesso

- CPU: cada instrução → 1, 2, … ciclos de relógio
- Memória: cada acesso → dezenas de ciclos
- Periféricos: cada E/S → muitas dezenas, centenas, ou mais, de ciclos

Vendo noutras escalas

- Exemplo *clock*: 1GHz → 1 000 000 000 ciclos/s
  - Admitindo a execução de 1 instrução p/ciclo (1ns)

	escala: 1ns → 1s	**escala: 1ns → 1m**
Registo: 1 ns	1 s	1 m
Memória RAM: 20 ns	20 s	20 m
Disco: 8 ms – 20 ms	3 a 7,6 meses	8 000 – 20 000 km

1 ms = 1 000 000 ns