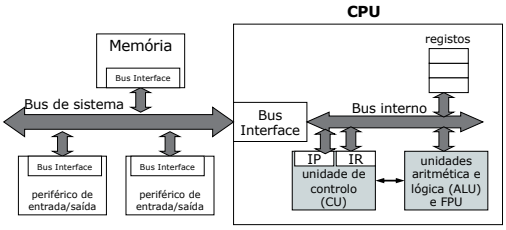
Arquitetura de Computadores

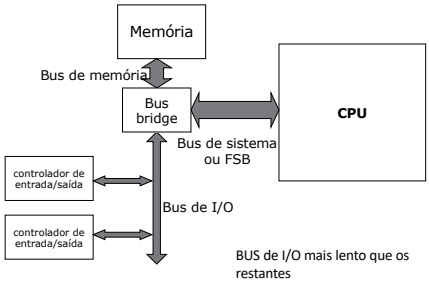
MIEI – 2017/18
DI-FCT/UNL
Aula 8

Arquitectura de Von Neumann



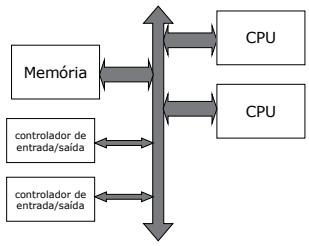
Arquitecturas de computadores

variantes



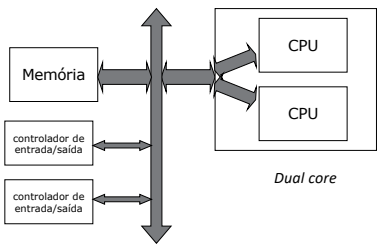
Arquitecturas de computadores (2)

variantes



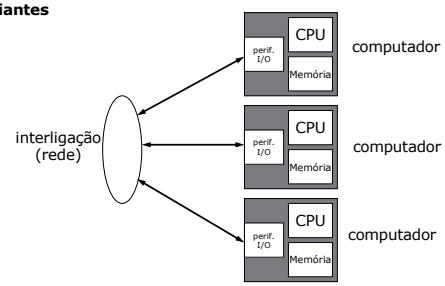
Arquitecturas de computadores (2)

variantes



Arquitecturas de computadores (3)

variantes

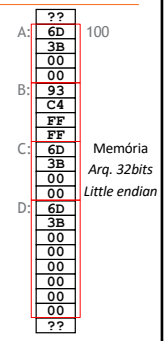


Variáveis na memória

- Variáveis são designações simbólicas para posições de memória:

```
int A = 15213;   // 0x3B6D
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

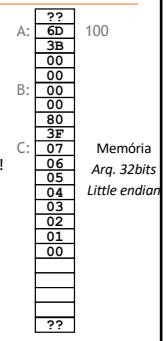


Variáveis na memória

- Qual o código produzido pelo compilador se:

```
int A = 15213;   // 0x00003B6D
float B = 1.0;   // 0x3F800000
long long C = 0x01020304050607;
A = C; //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 (float)
C: 108 : 8 bytes

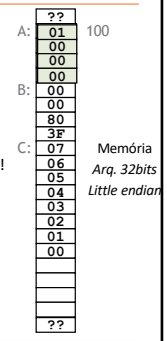


Variáveis na memória

- Qual o código produzido pelo compilador se:

```
int A = 15213;   // 0x00003B6D
float B = 1.0;   // 0x3F800000
long long C = 0x01020304050607;
A = C; //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 (float)
C: 108 : 8 bytes



Apontadores - C

- Apontadores (ou ponteiros) são variáveis cujo tipo representa referências para memória → **endereços**.
- Sintaxe para declarar a variável:
  Tipo * *var_pt*;
  Se var_pt não é iniciado pode conter "lixo"
  Dimensão de var_pt é a dos endereços (32 ou 64 bits)
- Operadores:
  & *nome_var* ← obter endereço de *nome_var*
  * *var_pt* ← obter o valor apontado
- Constante: **NULL** ← apontador inválido
  (semelhante ao ***null*** no Java)

Exemplo:

```
int X = 5;
int *Y;
```

X: 5
Y: ?
Y não iniciado

```
int a = *Y; // errado! (mas pode "executar")
Y=10;  // possivelmente errado!
```

```
Y= &X;  // Y aponta para X
int a = *Y;
//a recebe valor apontado por Y (ou seja, X)
```

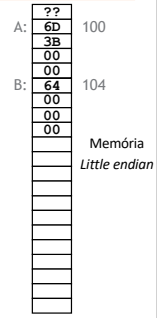
X: 5
Y: End de X
a: 5

Referências para memória - C

- Variável B refere A, guarda endereço de A, ou aponta para valor em 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*

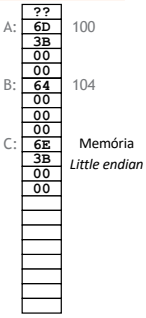


Referências para memória - C

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

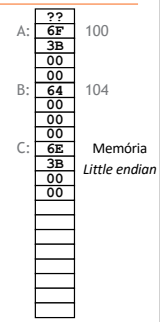


Referências para memória - C

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



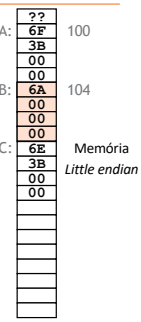
Referências para memória - C

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

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

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

- Em C podemos usar expressões com apontadores:

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



Ignorando os tipos de dados (!)

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

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

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

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


Uma variável, vários tipos

- Podemos declarar uma variável com vários tipos:

```
union { int i; float f; } var;
```

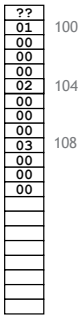
  - var pode ser vista como int ou float.
  - É só uma variável. Só são ocupados 4 bytes!
  - O padrão de bits em memória é sempre o mesmo!

```
var.f = 1.5;
printf("valor em var: %X\n", var.i);

valor em var: 3FC00000
```

Arrays em C e apontadores

- O nome do array vale por um apontador constante: v

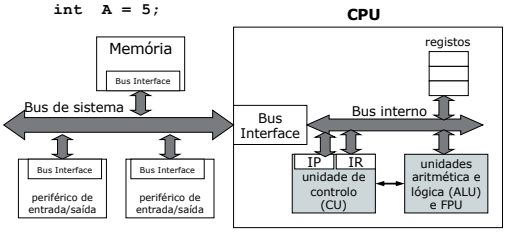
```
int v[3]={1, 2, 3};
// v tem tipo int*
// v representa end do vetor

v[2] = 5;    ⇔   *(v+2) = 5;
```



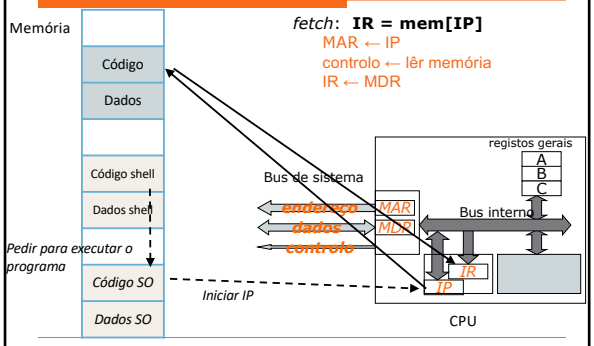
Arquitectura de Von Neumann

Como executar?

```
int  A = 5;
```



Execução de um Programa

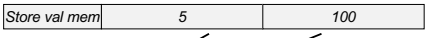


Execução da atribuição

int A = 5;

O compilador definiu um endereço de memória para A (exemplo: 100). Sabe a sua dimensão (4 bytes). Gerou a instrução máquina para colocar o valor 5 na memória respetiva.

- Fetch, obtém a instrução máquina para IR:

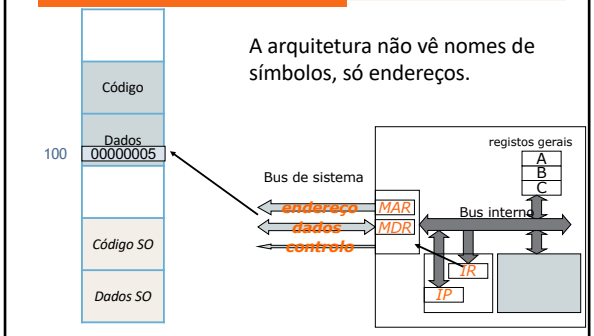


- Descodifica e Executa:

Store *val to memory:*

MDR ← 5
MAR ← 100
controlo ← escr. Memória

Execução de um Programa C



Helloworld em *assembly* Linux/Intel 32bits

```
.data           # secção de dados (variáveis)
msg:   .ascii    "Hello, world!\n"  # um vetor de caracteres
 len = 14   # len representa o tamanho do texto em msg

.text           # secção de código
.global _start   # exportar o simbolo _start (inicio programa)
_start:  movl    $len,%edx
         movl    $msg,%ecx
         movl    $1,%ebx
         movl    $4,%eax       # pedir write ao sistema
         int     $0x80         # chama o sistema

         movl    $0,%ebx
         movl    $1,%eax       # pedir o exit ao sistema
         int     $0x80         # chama o sistema
```

Vendo o resultado do assembler

```
$ objdump  -D hello.o
hello.o:     file format elf32-i386
Disassembly of section .text:
00000000 <_start>:
0:       ba 0e 00 00 00            movl    $0xe,%edx
5:       b9 00 00 00 00            movl    $0x0,%ecx
a:       bb 01 00 00 00            movl    $0x1,%ebx
f:       b8 04 00 00 00            movl    $0x4,%eax
14:      cd 80                     int     $0x80
16:      bb 00 00 00 00            movl    $0x0,%ebx
1b:      b8 01 00 00 00            movl    $0x1,%eax
20:      cd 80                     int     $0x80
Disassembly of section .data:
00000000 <msg>:
0:       48
1:       65
2:       6c
3:       6c
4:       6f
 . . . etc
```

Representação Hexadecimal do código máquina

Código executado pelo CPU?

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

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

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
  - Promover uma abordagem em que instruções mais complexas não são oferecidas, mas podem ser obtidas à custa da várias das existentes

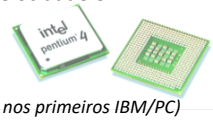
Intel e os circuitos integrados (CI)

- Intel- (Integrated Electronics Corporation)
  - Fundada em 1968 por funcionários vindos da Fairchild Semiconductor
    - Na origem (com a Texas Instruments) dos circuitos integrados (*microchip*)
  - Micro-processador
    - um único circuito integrado contendo todas as funcionalidades da unidade central de processamento (CPU) do computador
      - Intel 4004 – µ-proc. de 4bits para calculadoras

Nota: Já existiam processadores e computadores!

Alguns µ-processadores Intel

- Cada nova geração aumentou a velocidade e capacidade:



  Proc. – dados/endereços
  - 8080 – 8bits/16bits
  - 8086/8088 – 16bits/20bits *(usados nos primeiros IBM/PC)*
  - 80286 – 16bits/24bits *(usado no IBM/AT)*
  - 80386 – 32bits/32bits *(usado no IBM/PS2)*
  - 80486, Pentium (P5), Pentium Pro (P6), …
- Existe compatibilidade
  - continuam a existir instruções com dados de 8bits no 80386 e seguintes

Diagrama do 8086

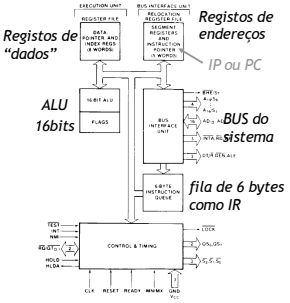
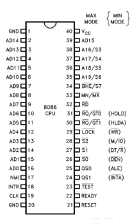


Figure 1. 8086 CPU Block Diagram

231455–1



40 Lead

231455–2

Figure 2. 8086 Pin Configuration