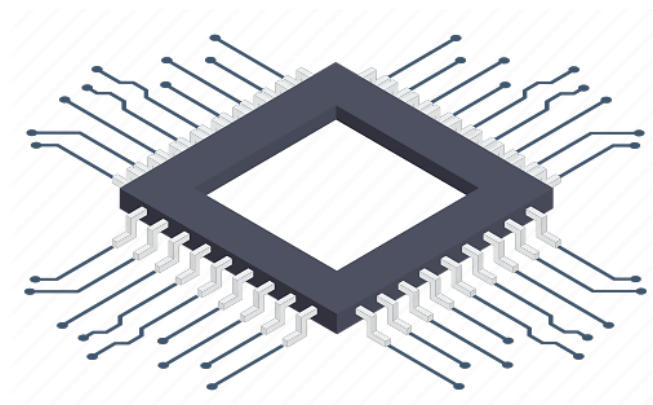


# **C208 – Arquitetura de Computadores**

## **Capítulo 1 – Introdução à Arquitetura de Computadores** (parte 2)



*Prof. Yvo Marcelo Chiaradia Masselli*

# Processadores

*Arquitetura Básica*

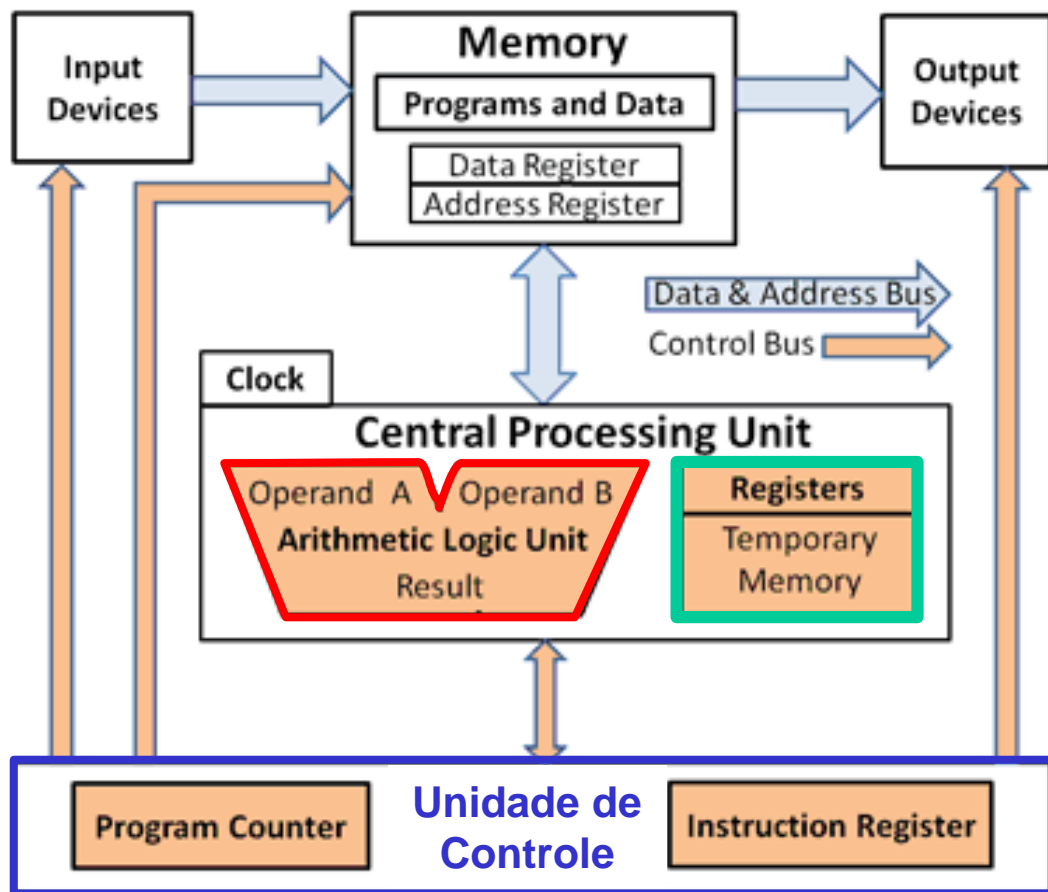
*Ciclo de Instrução*

*Conjunto de Instruções*

*Arquiteturas CISC x RISC*

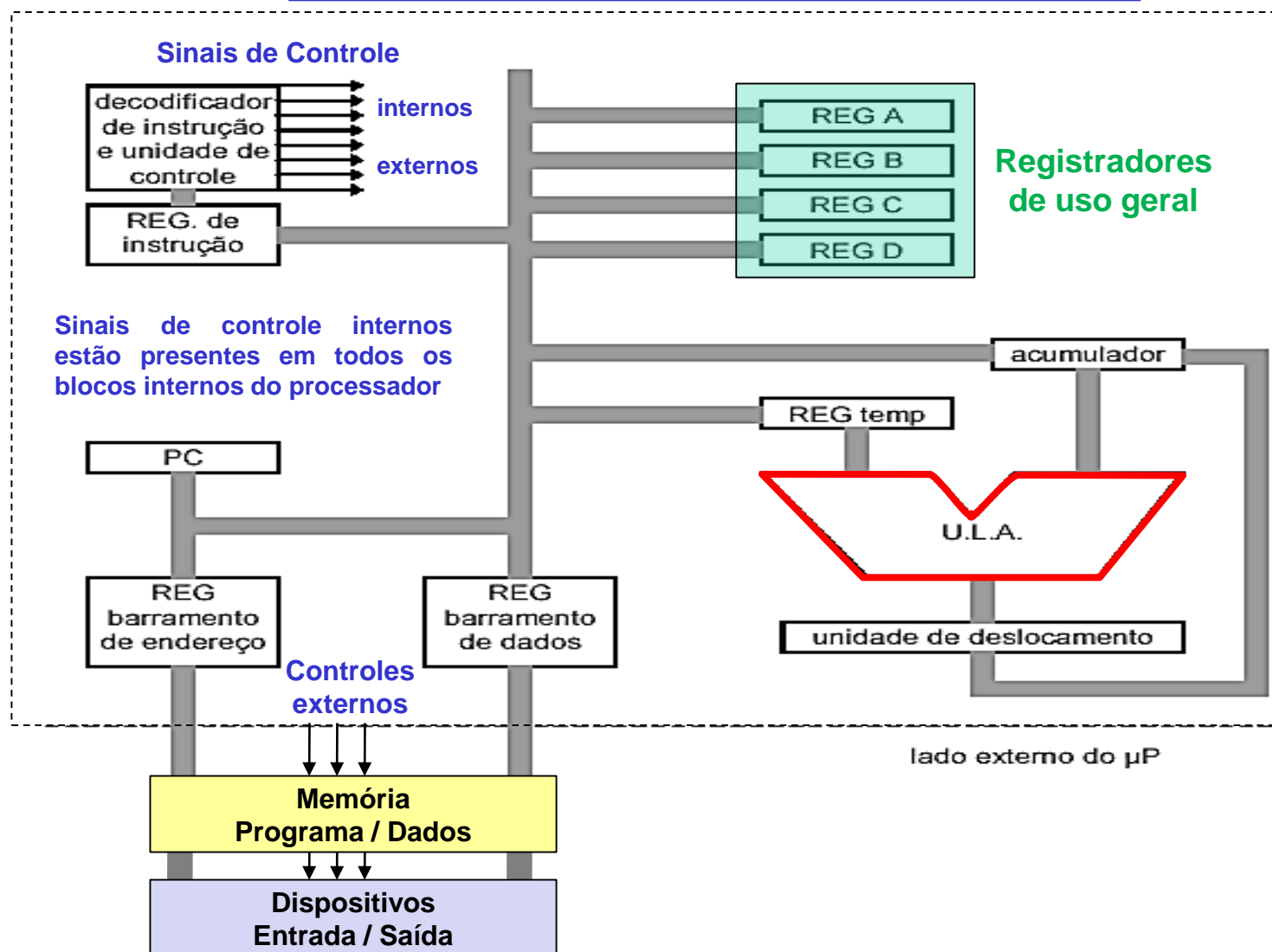
## Arquitetura Básica de um Processador

### Von Neumann Architecture



- **Unidade de Controle (UC)** → controla a operação interna e externa da CPU, sincronizando todo o sistema.
- **Unidade Lógica e Aritmética (ALU)** → realiza as operações lógicas e aritméticas da CPU.
- **Unidade de Registradores** → oferece armazenamento interno, temporário e de rápido acesso à CPU.

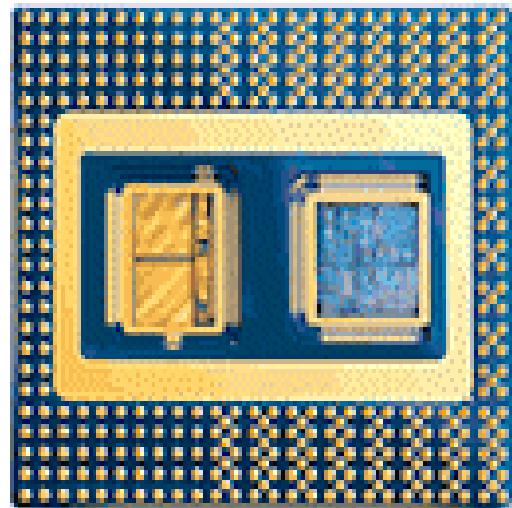
## Arquitetura Básica de um Processador



## Processamento das Instruções

• Instruções → São tarefas elementares que podem ser executadas em um processador, tais como:

- Somar 2 números;
- Transferir um número da Memória para a CPU;
- Transferir um dado da CPU para uma Porta de E/S;
- etc



**OBS:** Cada instrução do processador é representada por um código de operação (**OP-CODE**)

## Processamento das Instruções

### • Conjunto de Instruções (ou “SET” DE INSTRUÇÕES)

É o conjunto de todas as Instruções disponíveis do nível ISA.

Grupo	Sintaxe		Tipo	Op	Func	Comentário
Operações aritméticas	add	Rdest, Rsrc1, Rsrc2	R	00	0x20	addition (with overflow)
	addi	Rdest, Rsrc1, Imm16	I	08	-	addition imm. (with ov.)
	addu	Rdest, Rsrc1, Rsrc2	R	00	0x21	addition (without overflow)
	addiu	Rdest, Rsrc1, Imm16	I	09	-	addition imm. (without ov.)
	sub	Rdest, Rsrc1, Rsrc2	R	00	0x22	subtract (with overflow)
	subu	Rdest, Rsrc1, Rsrc2	R	00	0x23	subtract (without overflow)
	mult	Rsrc1, Rsrc2	R	00	0x18	multiply
	multu	Rsrc1, Rsrc2	R	00	0x19	unsigned multiply
	div	Rsrc1, Rsrc2	R	00	0x1a	divide
	divu	Rsrc1, Rsrc2	R	00	0x1b	unsigned divide

**OBS:** *cada processador tem o seu próprio “Set de Instruções”*

## Processamento das Instruções

- Acesso aos dados na memória externa

### Ordem de bytes na memória

Os números inteiros são normalmente armazenados em sequências de bytes (variável int em C tem 32 bits – 4 bytes), sendo que o valor é obtido por simples concatenação de bytes na memória. Os métodos mais comuns são:

Os bytes são guardados por **ordem decrescente do seu "peso numérico"** em **endereços sucessivos da memória** (extremidade maior primeiro ou *Big-endian*).

Os bytes são guardados por **ordem crescente do seu "peso numérico"** em **endereços sucessivos da memória** (extremidade menor primeiro ou *Little-endian*).

## Processamento das Instruções

- Acesso aos dados na memória externa

### Ordem dos bytes na memória

Ex: o valor  $90AB12CD_{(16)}$  armazenado na memória em:

#### *Big-endian*

Address	Value
1000	90
1001	AB
1002	12
1003	CD

#### *Little-endian*

Address	Value
1000	CD
1001	12
1002	AB
1003	90

#### Big Endian:

IBM 360/370, Motorola 68k, **MIPS**,  
Sparc, HP PA, entre outros

#### Little Endian:

Intel x86, Intel x64, DEC/Vax,  
DEC/Alpha, ARM, entre outros.



## Exercício:

Mostre como o dado **6151CE94h**, ficará armazenado no endereço de memória **0F40h**, nos modos **Big-endian** e **Little-endian**.

## Resposta

1) Big Endian:

0F40	61	51	CE	94
0F44	00	00	00	00

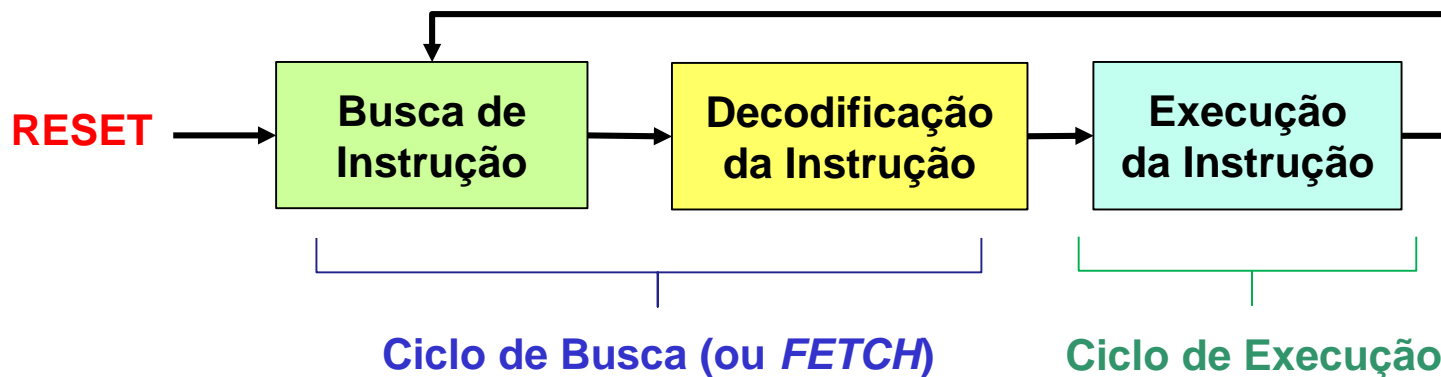
2) Little Endian:

0F40	94	CE	51	61
0F44	00	00	00	00

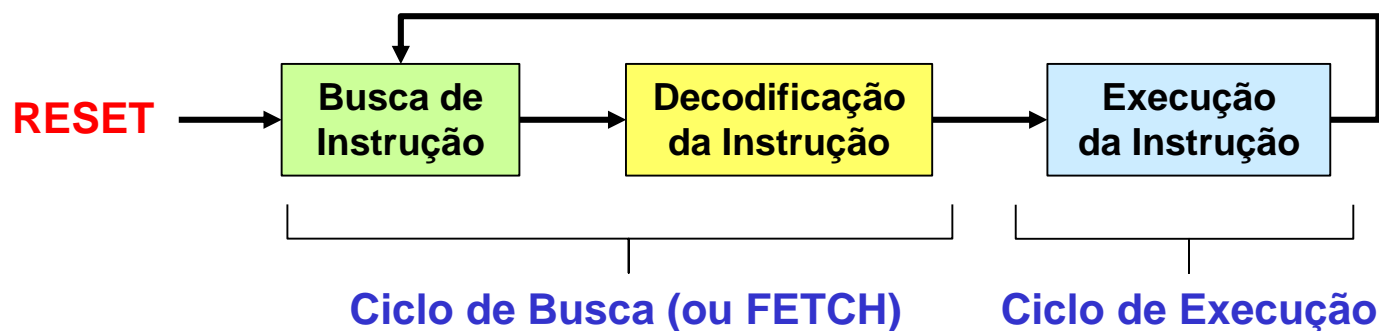
## Processamento das Instruções

O processamento de cada instrução é feito pela atuação da UC (Unidade de Controle) sobre todos os sinais de controles do Sistema (**Barramentos de controle interna e externo**), numa sequência de tempo bem definida (baseada na frequência do *clock*).

Esta sequência de controles para o processamento da Instrução é chamado de Ciclo de Instrução.



## Processamento das Instruções



Durante o processamento das instruções, dois registradores internos (uso específico) são muito importantes:

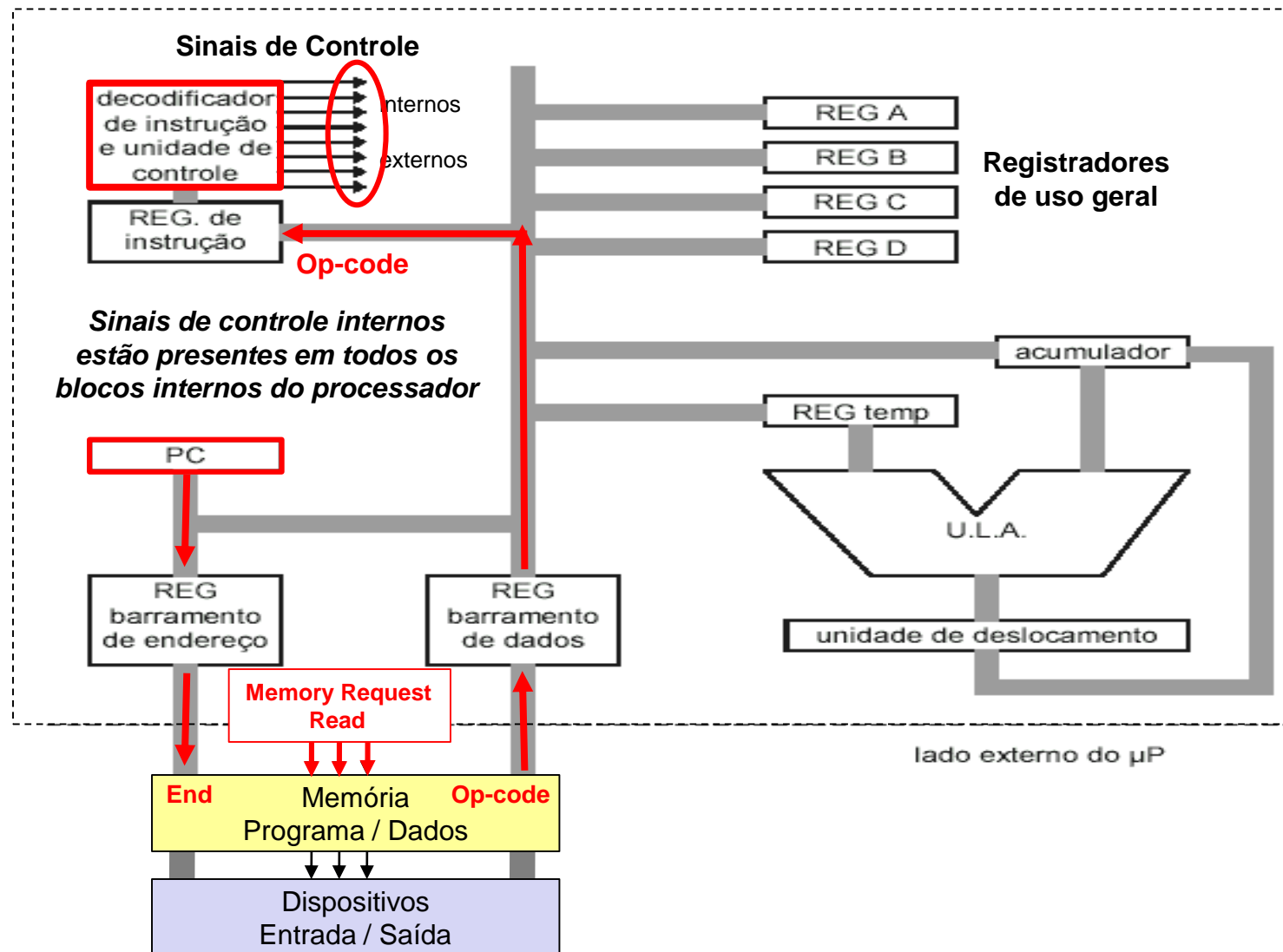
- **PC (Program Counter):** nele fica armazenado o endereço de memória de programa onde está a próxima instrução (op-code) a ser executada.
- **IR (Instruction Register):** armazena a instrução (op-code) em execução na Unidade de Controle (UC).

## Processamento das Instruções

### Ciclo de Busca (FETCH)

#### 1º. Passo:

Busca do Op-Code na Memória de Programa no endereço apontado pelo PC.

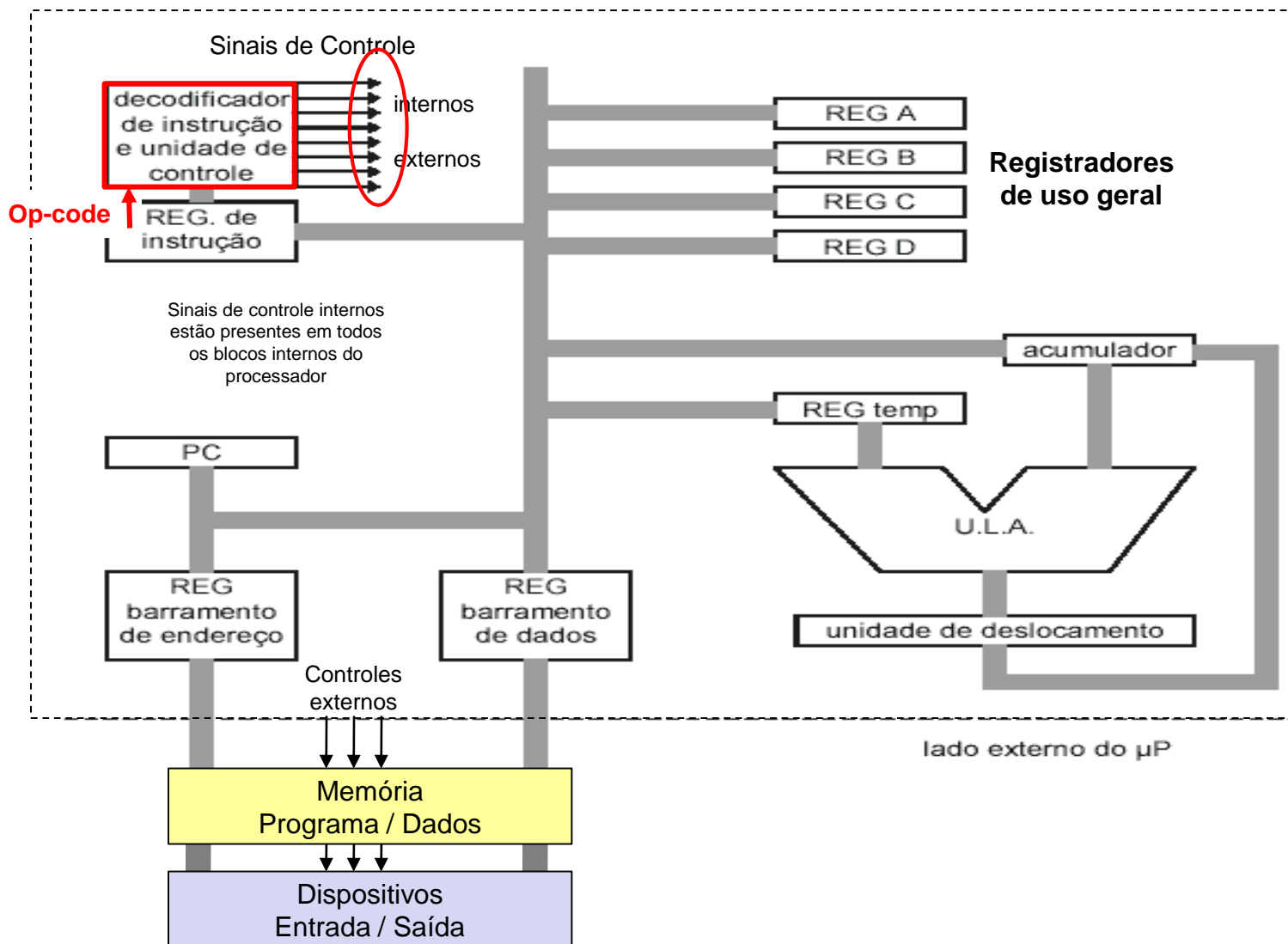


## Processamento das Instruções

### Ciclo de Busca (FETCH)

#### 2º. Passo:

Decodificação/  
Interpretação  
do Op-Code

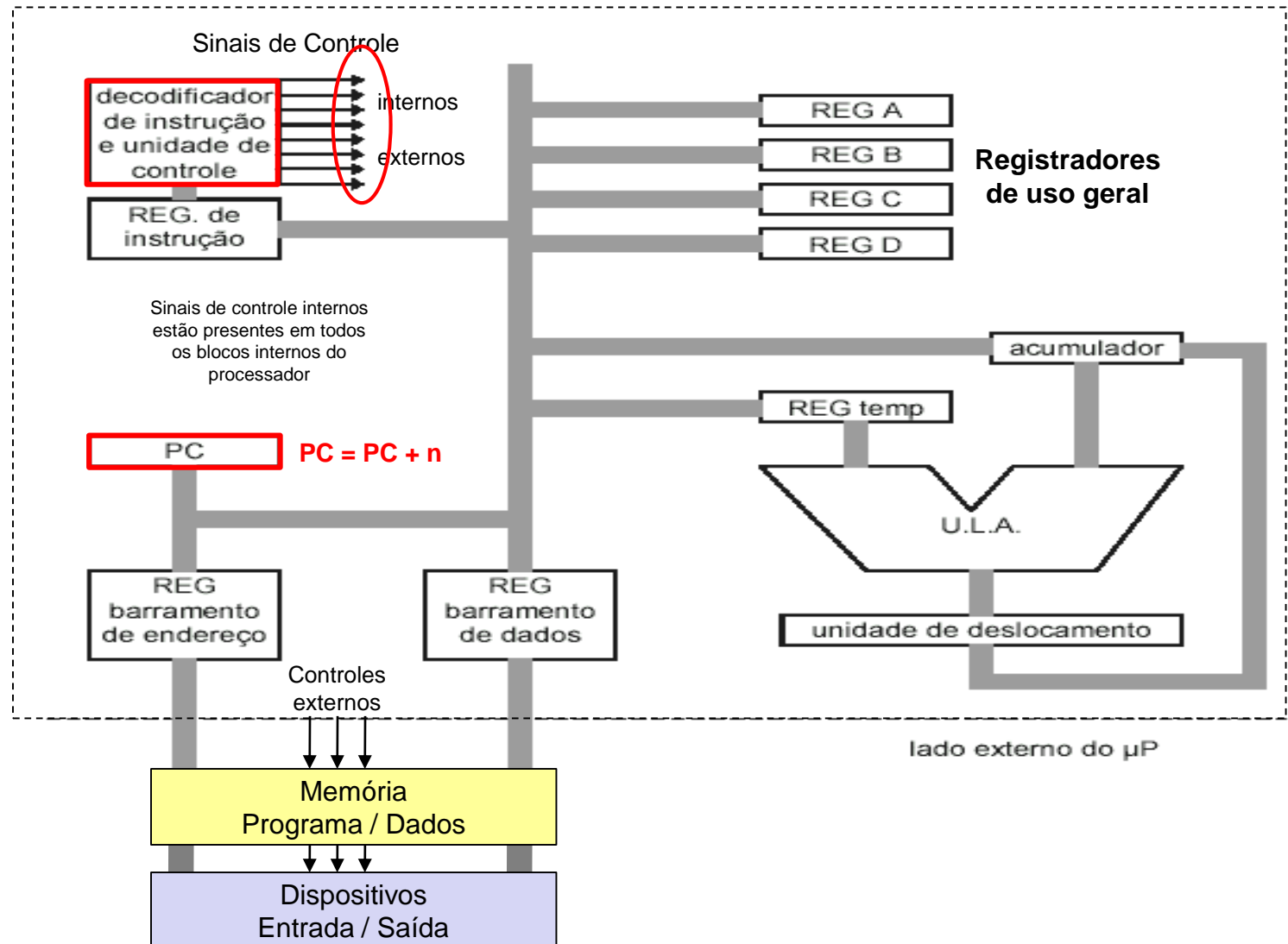


## Processamento das Instruções

### Ciclo de Busca (FETCH)

#### 3º. Passo:

Atualização do PC (PC aponta para o endereço da próxima instrução)

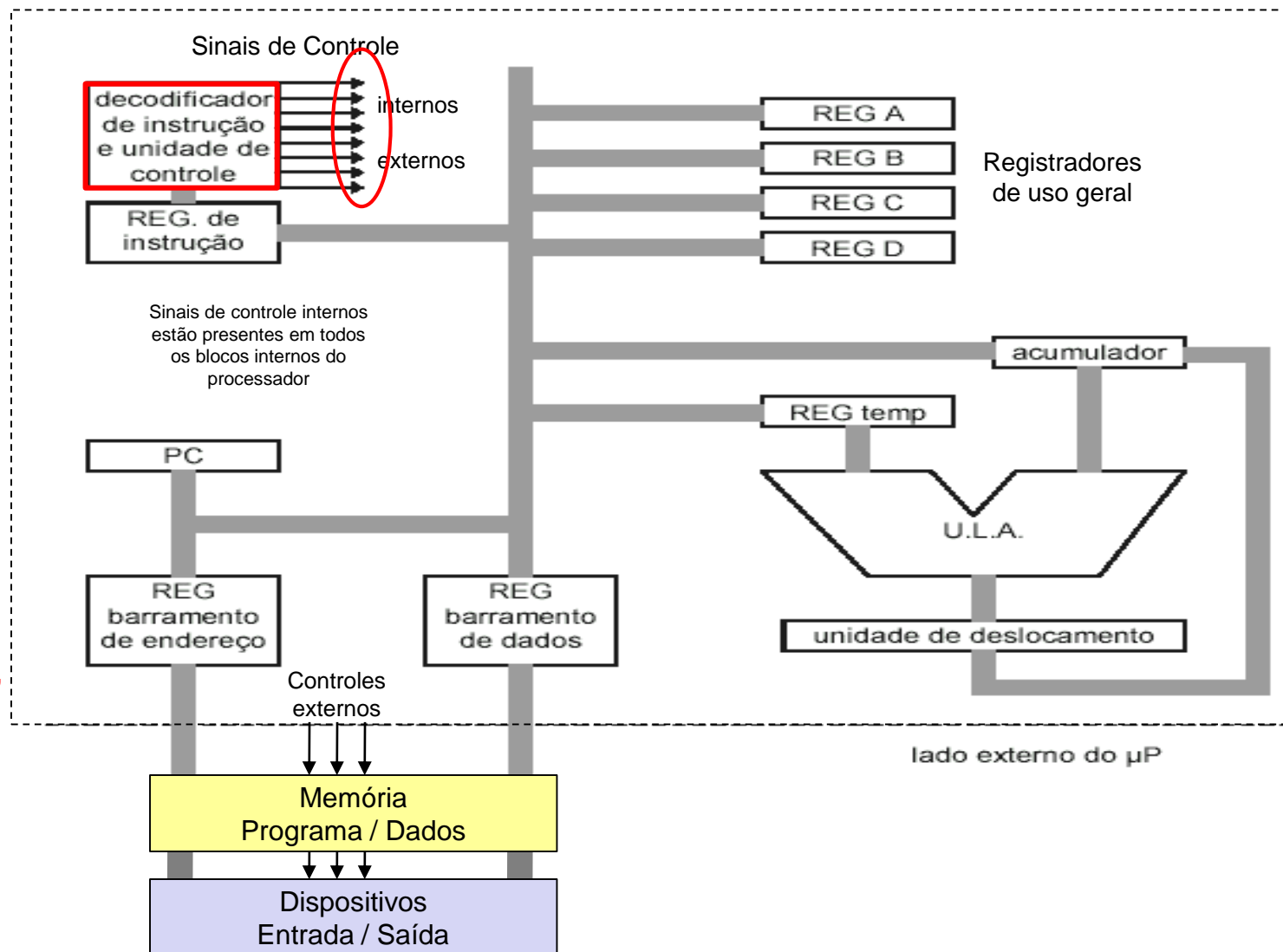


## Processamento das Instruções

### Ciclo de Execução da Instrução

**UC utiliza os sinais de controle (internos e/ou externos) para sequenciar todas as operações necessárias para a execução da instrução.**

Ex: Soma, subtração, leitura memória, etc





## Processamento das Instruções

Exemplo: execução da instrução **ADD A,B** ( $A = A+B$ )

Barramento de endereços  $\leftarrow$  PC  
Barramento Controle  $\leftarrow$  Memory Req + Read  
IR  $\leftarrow$  (memória) // IR  $\leftarrow$  op-code

UC  $\leftarrow$  op-code (decodificação/interpretação)

PC  $\leftarrow$  PC + 1 //considerando op-code de 1 byte

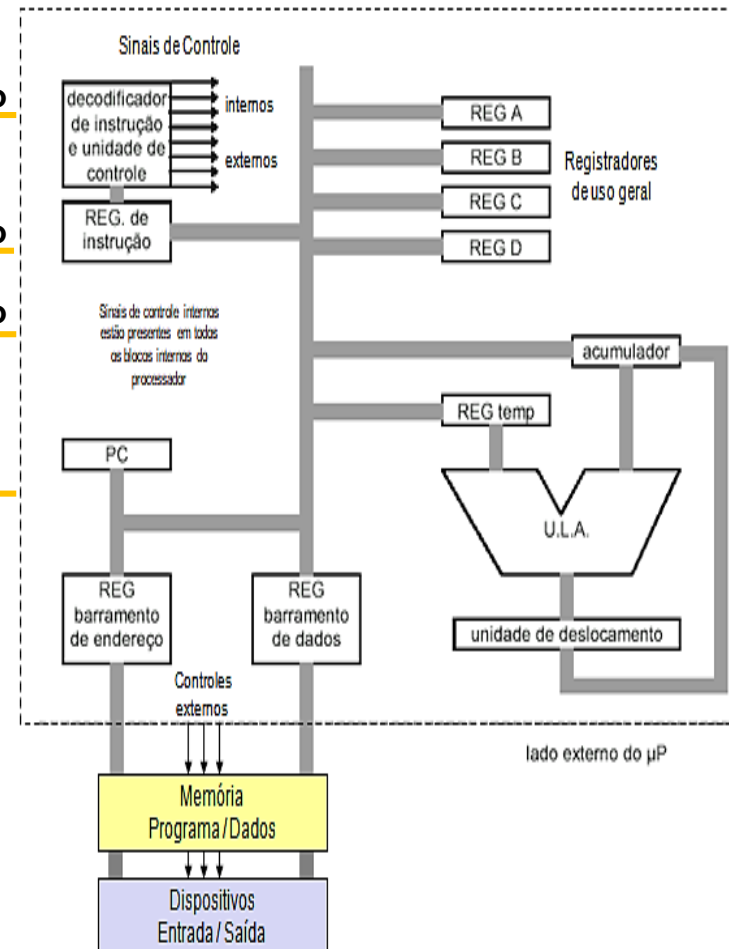
Acumulador  $\leftarrow$  REG A  
REG temp  $\leftarrow$  REG B  
Acumulador  $\leftarrow$  Acumulador ALU-SOMA REG temp  
REG A  $\leftarrow$  Acumulador

**FETCH – 1º. passo**

**FETCH – 2º. passo**

**FETCH – 3º. passo**

**Execução**



## Processamento das Instruções

### Exercícios:

Descreva os passos simbólicos para os ciclos de Busca e Execução das instruções abaixo (considere acesso à memória no formato **Little-endian**):

- |                  |                 |                |
|------------------|-----------------|----------------|
| a) ADD C,D       | //C = C+D       | op-code=1byte  |
| b) LD A,(3547h)  | //A = (3547h)   | op-code=3bytes |
| c) ADD A,(0258h) | //A = A+(0258h) | op-code=3bytes |
| d) ST (4455h),A  | //(4455h) = A   | op-code=3bytes |

## Arquiteturas dos Processadores

### Quanto ao Conjunto de Instruções (Nível ISA)

**CISC** – Complex Instructions Set Computers (Computadores com Conjunto de Instruções Complexas)

- Número maior de instruções (muitas instruções disponíveis a nível da ISA).
- Modos de endereçamento das instruções: por registrador e por memória
  - torna o processo de decodificação uma interpretação, onde o op-code tem que ser interpretado em vários passos de sequencia pela unidade de controle (microcódigo)
- Interpretação do op-code (por software – microcódigo) ao invés de decodificação (por hardware – sem microcódigo).
  - torna o processo de execução da instrução mais demorado (n ciclos de clock, um para cada passo da sequencia do microcódigo)
- Em geral tem op-codes menores, pois a complexidade de execução está na interpretação (microcódigo). Porém os op-codes podem ter tamanhos diferentes (depende do modo de endereçamento da instrução)

## Arquiteturas dos Processadores

### Quanto ao Conjunto de Instruções (Nível ISA)

**RISC** – Reduced Instructions Set Computers (Computadores com Conjunto de Instruções Reduzidas)

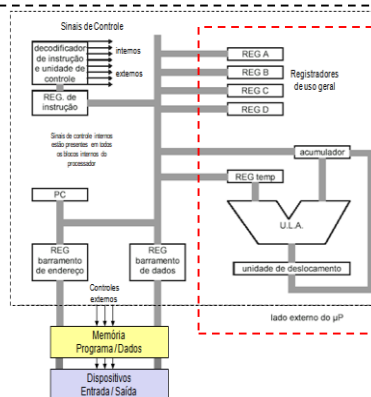
- Número pequeno de instruções (poucas instruções disponíveis a nível da ISA)
- Modos de endereçamento das instruções: somente por registrador (acesso a memória somente com instruções LOAD e STORE)
- Decodificação (por hardware) das instruções (muito rápido e sem execução de microcódigo). As instruções possuem tamanho fixo.
- Como as instruções são mais simples, muitas vezes é necessário várias instruções para fazer a mesma tarefa de uma instrução CISC.
  - porém como a execução das instruções RISC são muito rápidas, mesmo assim compensa e tem melhor desempenho.
- Processamento das Instruções é feito de forma muito rápida, geralmente 1 instrução por ciclo de clock.

## Exemplo de Execução CISC x RISC

### CISC

ADD (2352h),B

Op-code de 3 bytes na memória:  
13h (instrução)  
52h (operando Low)  
23h (operando High)



No ciclo de execução – microcódigo:

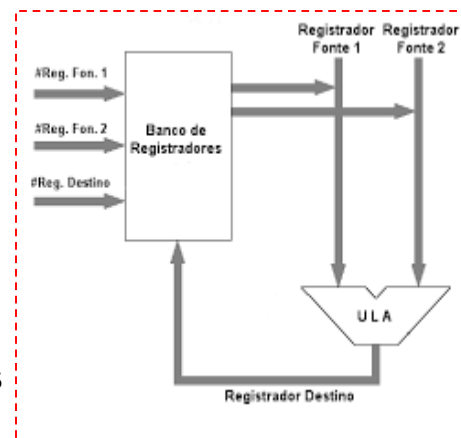
Reg-end = Reg-end+1 //aponta operando low-end  
Reg temp = leitura memória end. Reg-end  
Reg-end = Reg-end+1 //aponta operando high-end  
Reg-end high = leitura memória end. Reg-end  
Reg-end low = Reg temp  
Acumulador = leitura na memória end. Reg-end  
REG temp = REG B  
Acumulador = Acumulador ALU-SOMA REG temp  
Escrita na memória no end. Reg-end = Acumulador

Ciclo da instrução: 5 (fetch) + 9 (execução) = 14 cck

### RISC

LD (2352h)  
ADD B  
ST (2352h)

Op-codes: LD → 3 bytes:  
ADD → 3 bytes  
ST → 3 bytes



No ciclo de execução – decodificação:

REG temp = leitura memória end. 2352h  
REG temp = REG B ALU-SOMA Reg temp  
Escrita na memória no end. 2352h = REG temp

Ciclo da instrução: 1 cck por instrução  
Total: 3 cck

RISC:

Código na memória: maior

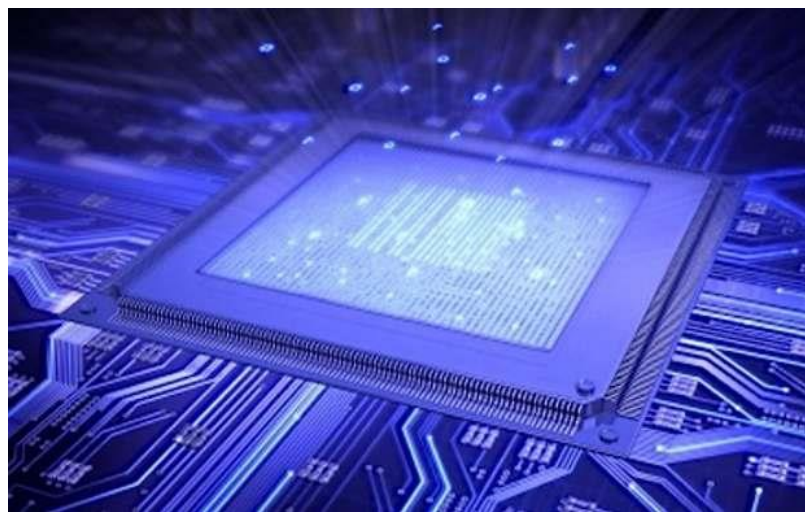
Op-code decodificado (não tem microcódigo)

Tempo de execução: mais rápido (menos cck)

## Arquiteturas dos Processadores

Atualmente no mercado de processadores existem muitos fabricantes adotando soluções HÍBRIDAS entre CISC e RISC.

- Por exemplo, a INTEL usa RISC para instruções de uso mais frequente (*Núcleo RISC*) e *interpretação (Núcleo CISC)* para instruções mais complexas e de uso menos frequente.



## Arquiteturas dos Processadores

Exercícios:

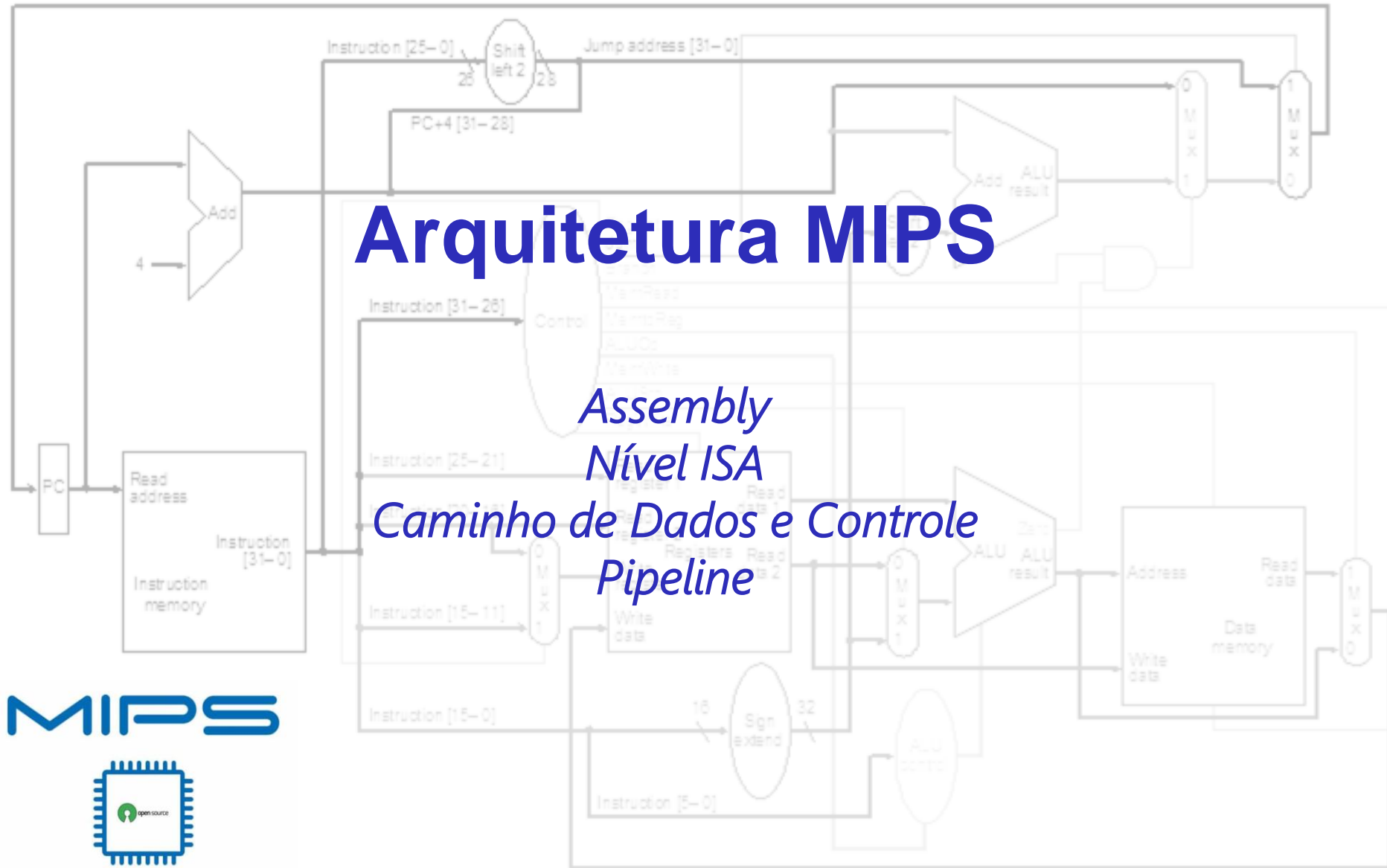
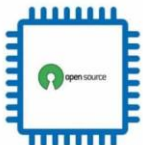
- 1) Descreva a função de cada um dos barramentos de uma arquitetura genérica (dados, endereços e controle).
- 2) Qual a influência do barramento de endereços na arquitetura de um processador? E do barramento de dados?
- 3) Faça uma comparação entre as arquiteturas CISC e RISC com relação aos aspectos a seguir e justifique a resposta:
  - Quanto ao tamanho do programa armazenado na memória;
  - Quanto ao tempo de execução das instruções;
  - Quanto a implementação da UC (Unidade de Controle)
- 4) Pode-se afirmar que na arquitetura CISC as instruções são “decodificadas” por software enquanto que na RISC são por hardware? Justifique.
- 5) Explique em detalhes o que você entende por microcódigo. Fica gravado em alguma memória? Em que local do Processador?

## Arquitetura MIPS

*Assembly  
Nível ISA*

*Caminho de Dados e Controle  
Pipeline*

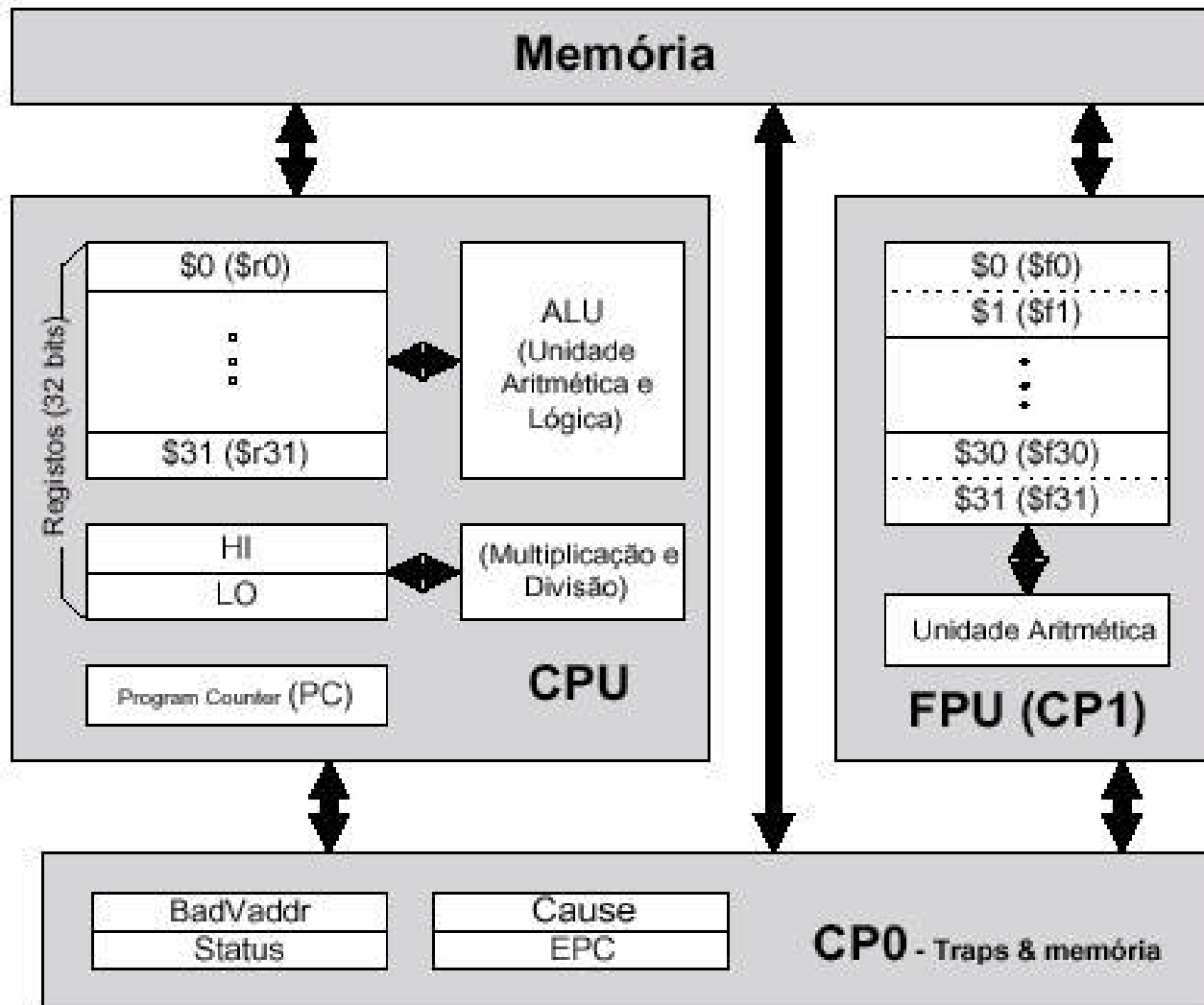
**MIPS**



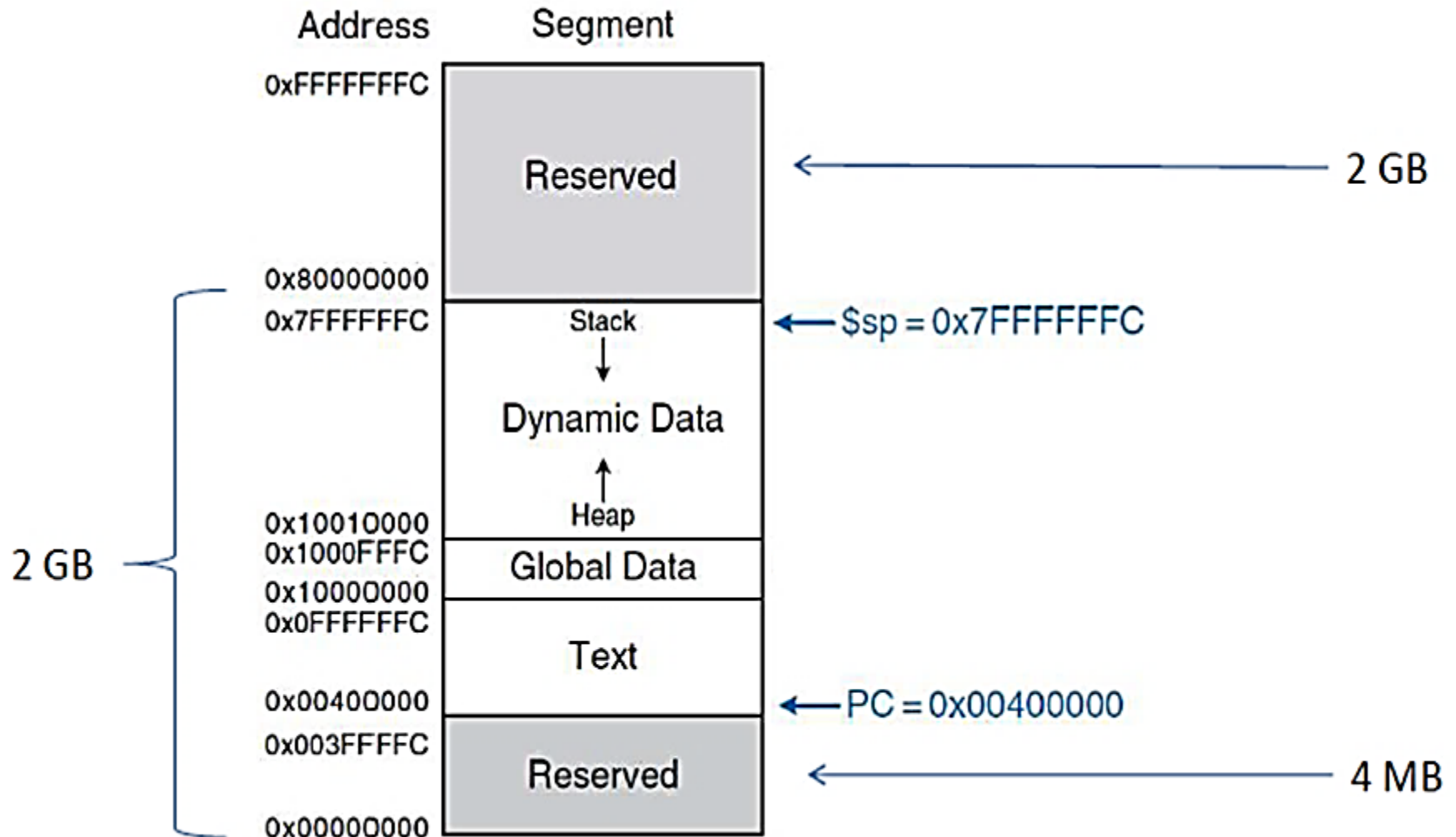




# MIPS – Arquitetura



# MIPS – Memória



## Assembly do MIPS - Estrutura

Um programa assembly é um texto a 3 colunas, em que a **primeira representa de uma forma simbólica endereços de memória** (labels), a **segunda coluna, as instruções** e a **última coluna, os comentários**.

Exemplo:

```
Var_1:      .data
            .word 1
            .text
            .globl main
main:       li $v0, 4           #Comentário
            lw $a0, Var_1
```

## Assembly do MIPS - Estrutura

**Comentários** → começam com o caracter "#" e vai até o final da linha.

**Etiquetas ou LABELs** → Identificadores que ficam no início de uma linha e que são sempre seguidos de dois pontos (:). Servem para dar um nome ao endereço de memória. Pode-se controlar o fluxo de execução do programa criando saltos para os Labels (etiquetas).

**Instruções** → instruções do programa

**Diretivas** → Instruções para o Montador Assembly a fim de informar como traduzir o programa. São identificadores reservados, e iniciam-se sempre por um ponto (.).

# Exercício:

Dado o seguinte programa em Assembly do MIPS, indique as etiquetas (*Labels*), as diretivas, as instruções e os comentários.

**.data**

**Var\_1: .word 1**

**.text**

**.global main**

**main: li \$v0, 4                   #chama a system call 4**  
      **lw \$a0, Var\_1**

## Assembly do MIPS - Diretivas

### Para criação de constantes e variáveis na Memória

#### **.ascii str**

Armazena uma *string* em memória sem acrescentar o terminador *NULL*.

#### **.asciiz str**

Armazena uma *string* em memória acrescentando o terminador *NULL*.

#### **.ascii “cadeia”**

A diretiva permite carregar, em posições de memória consecutivas (cada uma com dimensão de 1 byte), o código ASCII de cada um dos caracteres da cadeia.

#### **.byte b1, ..., bn**

Armazena as grandezas de 8 bits **b1, ..., bn** em sucessivos bytes de memória.

## Assembly do MIPS - Diretivas

### Para criação de constantes e variáveis na Memória

#### **.word** w1, ..., wn

Armazena as grandezas de 32 bits **w1, ..., wn** em sucessivas palavras de memória.

#### **.float** f1, ..., fn - armazena os números em ponto flutuante com precisão simples (32 bits)

**f1, ..., fn em posições de memória sucessivas.**

#### **.double** d1, ..., dn - armazena os números em ponto flutuante com precisão dupla (64 bits) **d1,..., dn em posições de memória sucessivas.**

#### **.space** n - reserva **n bytes** na memória de dados inicializando com 0.



# Assembly do MIPS - Diretivas

## Para controle dos Segmentos de Memória

### **.data <address>**

Os valores definidos após a diretiva devem ser colocados no segmento de dados do programa (0x10010000).

\* Opcionalmente podem começar a partir de um endereço especificado no campo <address>.

### Exemplo:

<b>.data</b>	<b># segmento de dados</b>
<b>palavra1: .word 13</b>	<b># decimal</b>
<b>palavra2: .word 0x15</b>	<b># hexadecimal</b>

## Exercícios:

- 1- Qual o valor dos *Labels* “palavra1” e “palavra2” no exemplo anterior?
- 2- Mostre como estes valores serão armazenados na memória (Big-endian) (lembre-se que o MIPS é um processador de 32 bits).
- 3- Existe alguma diferença no código a seguir em relação ao anterior?

```
.data                # segmento de dados  
palavra1: .word 13,0x15 #decimal,hexadecimal
```

- 4- Como os valores ficarão armazenados na memória de dados, no código abaixo (represente em Big-endian):

```
.data  
val1: .byte 0x10,0x20,0x30,0x40,0x50  
val2: .word 0x10203040
```

5- Como os valores ficarão armazenados na memória de dados, no código abaixo:

```
.data  
cadeia: .ascii "abcde"  
octeto: .byte 0xff  
string: .asciiz "teste"
```

6- Como os valores ficarão armazenados na memória de dados, no código abaixo:

```
.data  
cadeia: .ascii "abcde"  
          .word 0x345  
octeto: .byte 0xff  
string: .asciiz "teste"
```

# Assembly do MIPS - Diretivas

## Para controle dos Segmentos de Memória

### **.text** <address>

Os valores definidos a seguir devem ser colocados no segmento de código do programa, opcionalmente a partir do endereço <address>.

### Exemplo:

**.data**

**Var\_1: .word 1**

**.text**

**main:     li \$v0, 4               #chama a system call 4**

**lw \$a0, Var\_1**

# ISA (*Instruction Set Architecture*) MIPS

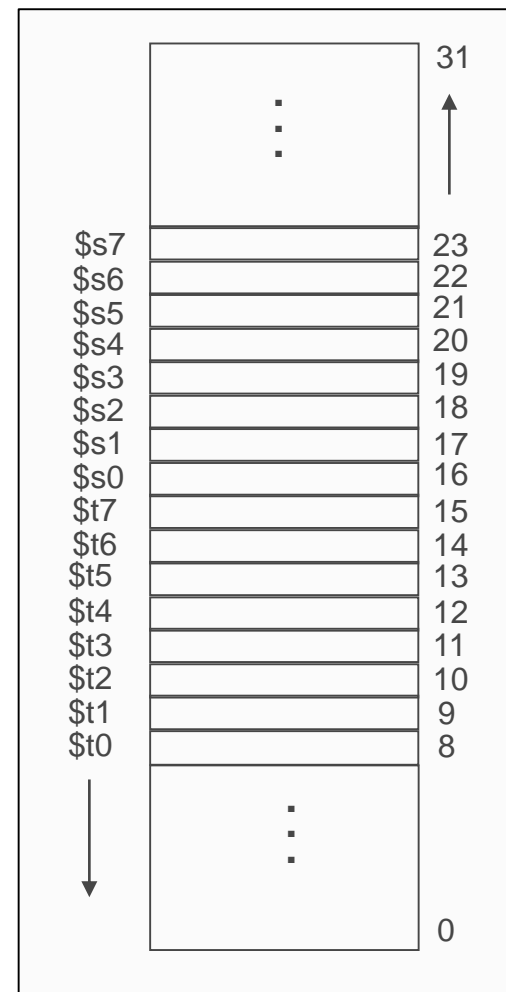
## - Registradores -

- MIPS: 32 registradores de 32 bits
- Cada registrador tem um nome
- Os nomes começam com \$

**\$s0, \$s1, \$s2... , \$s7** → correspondem às variáveis de programa

$\$t0, \$t1, \$t2..., \$t9 \rightarrow$  registradores temporários

Zero: \$zero ou \$0 → Constante 0



## ISA (*Instruction Set Architecture*) MIPS - Registradores -

Nome Real	Nome Lógico	Descrição
\$0	\$zero, \$r0	Sempre zero
\$1	\$at	Reservado para o assembler (assembler temporary)
\$2, \$3	\$v0, \$v1	Primeiro e segundo valores de retorno, respectivamente
\$4, ..., \$7	\$a0, ..., \$a3	Primeiros quatro argumentos para funções
\$8, ..., \$15	\$t0, ..., \$t7	Registradores temporários
\$16, ..., \$23	\$s0, ..., \$s7	Registradores salvos
\$24, \$25	\$t8, \$t9	Mais registradores temporários
\$26, \$27	\$k0, \$k1	Reservados para o Kernel do SO
\$28	\$gp	Global pointer (ponteiro global)
\$29	\$sp	Stack pointer (ponteiro para a pilha)
\$30	\$fp	Frame pointer (ponteiro para o frame)
\$31	\$ra	Return address (endereço de retorno)

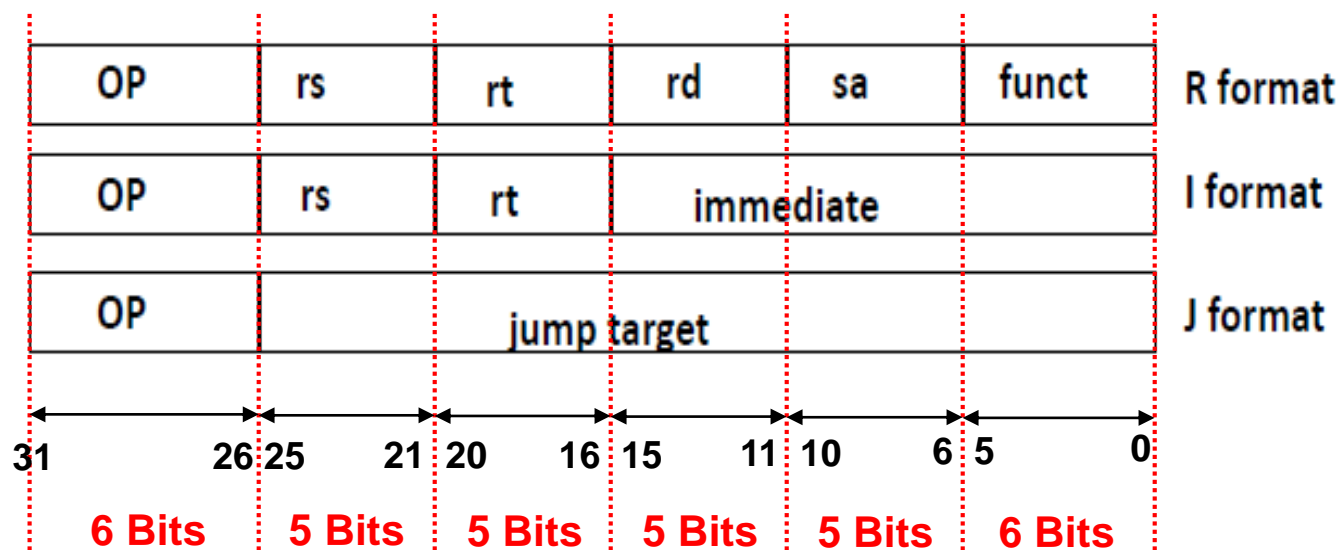
## ISA (Instruction Set Architecture) MIPS - Registradores -

### Tipos de Instruções (3 formatos):

**Tipo R** (R-format), envolvendo registrador-registrador

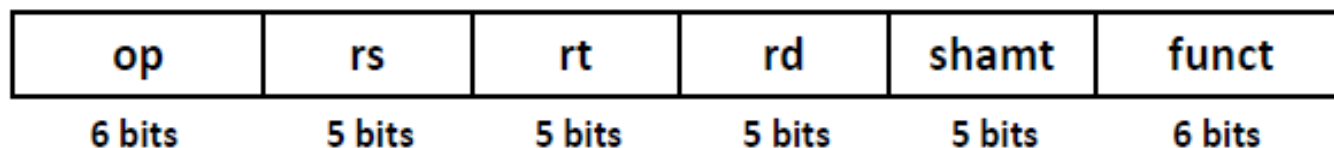
**Tipo I** (I-format), envolvendo valor imediato

**Tipo J** (J-format), de desvio



## ISA (Instruction Set Architecture) MIPS - Registradores -

### Tipo R:



### Campos:

op → código da operação (opcode)

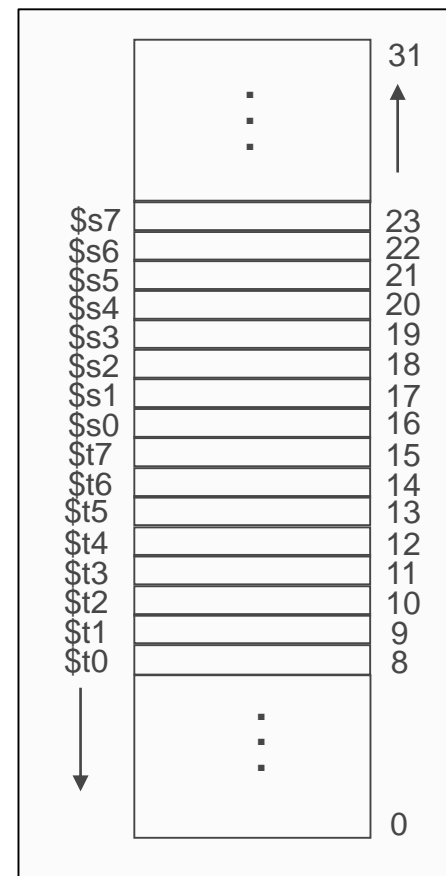
rs → endereço do primeiro registrador de origem

rt → endereço do segundo registrador de origem

rd → endereço do registrador destino

shamt → quantidade de bits a ser deslocado

funct → função específica a ser realizada





## ISA (Instruction Set Architecture) MIPS - Registradores -

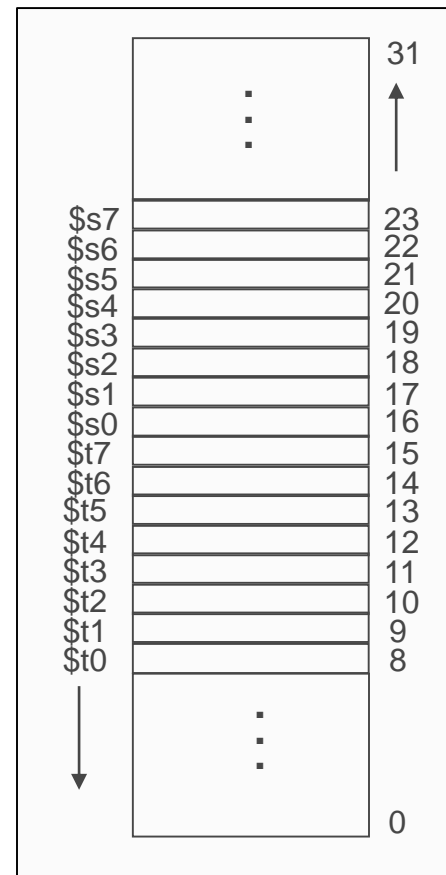
Tipo R - exemplos:

**add \$t2, \$t1, \$t0    # \$t2 = \$t1 + \$t0**

000000	01001	01000	01010	00000	100000
Op = 0x0	rs = 9	rt = 8	rd = 10	shamt	f = 0x20

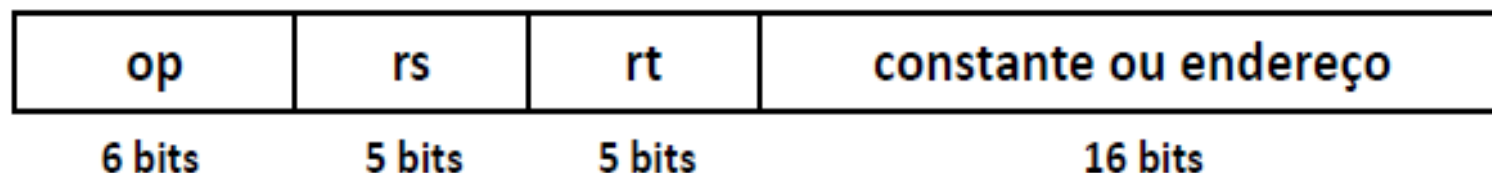
**sub \$s7, \$t8, \$zero    # \$s7 = \$t8 - 0**

000000	11000	00000	10111	00000	100010
Op = 0x0	rs = 24	rt = 0	rd = 23	shamt	f = 0x22



## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Tipo I:



### Campos:

op → código da operação (opcode)

rs → número do registrador base a ser operado com o valor constante

rt → número do registrador de destino ou operando

constante →  $-2^{15}$  a  $+2^{15}-1$

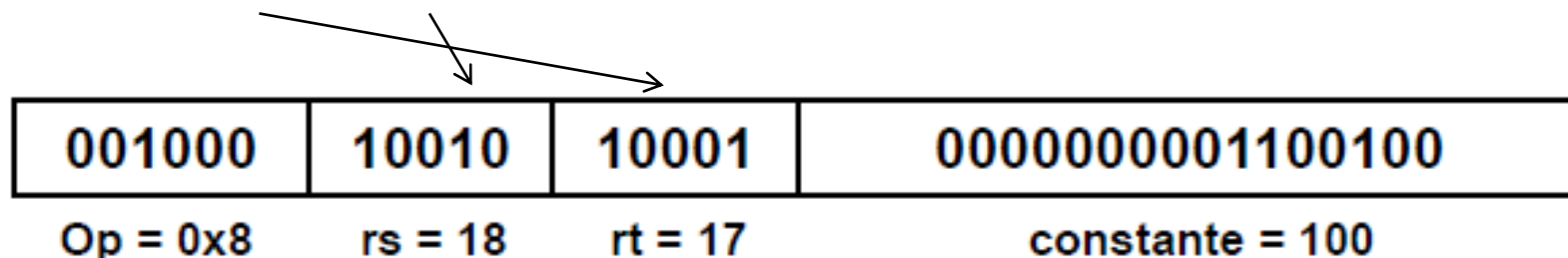
endereço →  $-2^{15}$  a  $+2^{15}-1$

## ISA (Instruction Set Architecture) MIPS - Registradores -

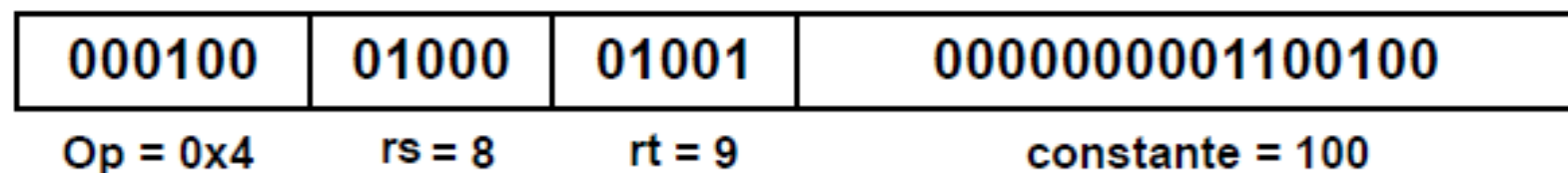
Tipo I - exemplos:

**addi \$s1, \$s2, 100**

**# \$s1 = \$s2 + 100**

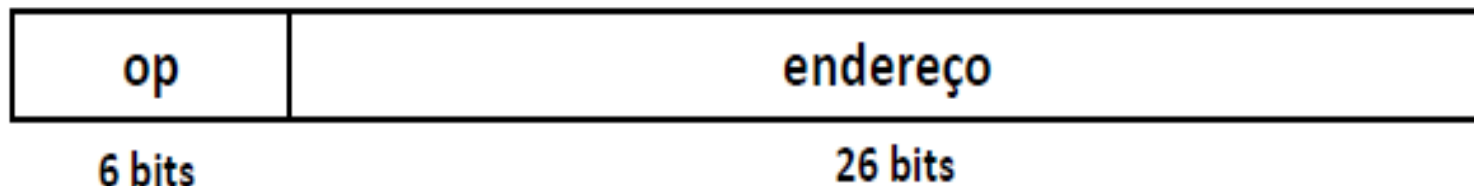


**beq \$t0, \$t1, 100**    **# se \$t0=\$t1 branch (salto) 100 endereços**



## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Tipo J:



### Campos:

op → código da operação (opcode)

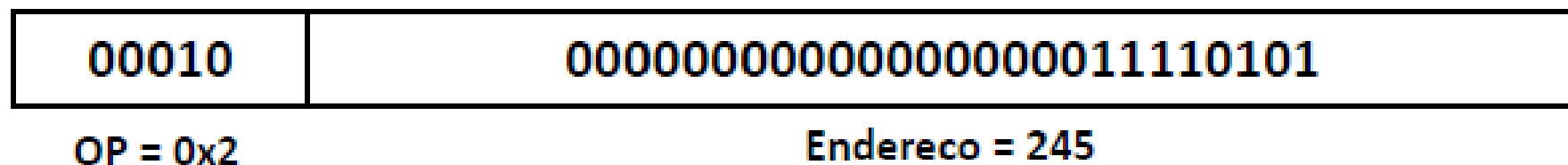
endereço → endereço destino

# ISA (Instruction Set Architecture) MIPS

## - Registradores -

## Tipo J - exemplo:

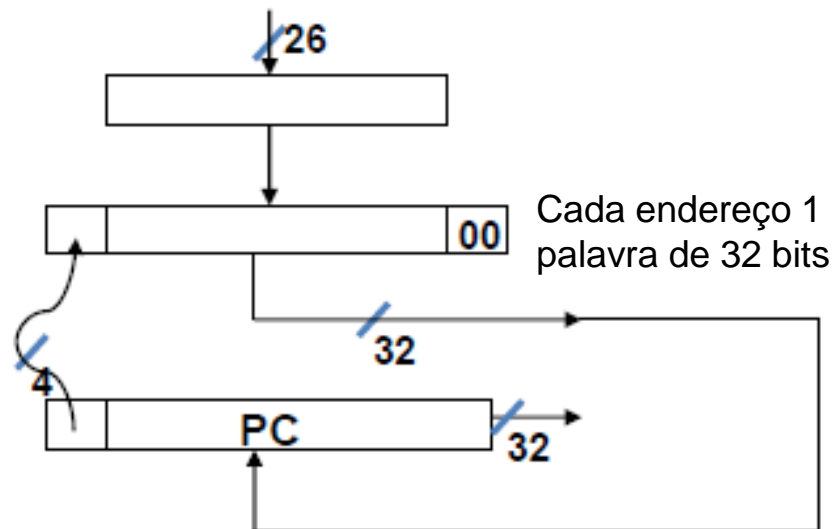
**j 245      #jump (salto) para o endereço absoluto 245**



**OBS:**

Campo de endereço = 26 bits  
PC = 32 bits

## Como transformar 26 bits de endereço em 32 bits?



## Arithmetic Operations

- |   |                  |  |                        |
|---|------------------|--|------------------------|
| • add                                   | add \$1,\$2,\$3  | $\$1 = \$2 + \$3$  | 3 operands             |
| • subtract                              | sub \$1,\$2,\$3  | $\$1 = \$2 - \$3$  | 3 operands             |
| • add immediate                         | addi \$1,\$2,5   | $\$1 = \$2 + 5 + \text{constant}$                            |                        |
| • add unsigned                          | addu \$1,\$2,\$3 | $\$1 = \$2 + \$3$  | 3 operands             |
| • subtract unsigned                     | subu \$1,\$2,\$3 | $\$1 = \$2 - \$3$  | 3 operands             |
| • add imm. unsign.                      | addiu \$1,\$2,5  | $\$1 = \$2 + 5 + \text{constant}$                            |                        |
| • multiply<br>signed product            | mult \$2,\$3     | Hi, Lo = $\$2 \times \$3$                                    | 64-bit                 |
| • multiply unsigned<br>unsigned product | multu \$2,\$3    | Hi, Lo = $\$2 \times \$3$                                    | 64-bit                 |
| • divide                                | div \$2,\$3      | Lo = $\$2 \div \$3$ , Hi = quotient,<br>Hi = $\$2 \bmod \$3$ |                        |
| • divide unsigned                       | divu \$2,\$3     | Lo = $\$2 \div \$3$ , Hi = Rem.                              |                        |
| • Move from Hi                          | mfhi \$1         | $\$1 = \text{Hi}$  | Used to get copy of Hi |
| • Move from Lo                          | mflo \$1         | $\$1 = \text{Lo}$  | Used to get copy of Lo |

## Logical Operations

• and	and \$1,\$2,\$3	\$1 = \$2 & \$3	Bitwise AND
• or	or \$1,\$2,\$3	\$1 = \$2   \$3	Bitwise OR
• xor	xor \$1,\$2,\$3	\$1 = \$2 XOR \$3	Bitwise XOR
• nor	nor \$1,\$2,\$3	\$1 = ~( \$2   \$3 )	Bitwise NOR
• and immediate const	andi \$1,\$2,10	\$1 = \$2 & 10	Bitwise AND reg,
• or immediate const	ori \$1,\$2,10	\$1 = \$2   10	Bitwise OR reg,
• xor immediate const	xori \$1, \$2,10	\$1 = ~\$2 & ~10	Bitwise XOR reg,
• shift left logical constant	sll \$1,\$2,10	\$1 = \$2 << 10	Shift left by
• shift right logical	srl \$1,\$2,10	\$1 = \$2 >> 10	Shift right by constant
• shift right arithm. extend)	sra \$1,\$2,10	\$1 = \$2 >> 10	Shift right (sign
• shift left logical	slv \$1,\$2,\$3	\$1 = \$2 << \$3	Shift left by var
• shift right logical	srsv \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right by var
• shift right arithm. var	srav \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right arith. by



## Data Transfer Operations

- Store word                      SW R3, 500(R4)                      # Memory[500+R4] = R3
- Store half word                SH R3, 502(R2)
- Store byte                      SB R2, 41(R3)
- Load Word                    LW R1, 30(R2)                # R1 = Memory[R2 + 30]
- Load Halfword                LH R1, 40(R3)
- Load Halfword Uns            LHU R1, 40(R3)
- Load Byte                    LB R1, 40(R3)
- Load Byte Unsigned           LBU R1, 40(R3)
- Load Upper Imm.            LUI R1, 40                      (16 bits shifted left by 16)

Pseudo-instruções: instruções que o montador traduz para uma ou várias instruções reais do conjunto ISA.

Pseudo-Commands (special commands that make things easier for humans, but that do not really exist in the ISA)

- Load Address                la \$t0,x                      # load address of label x into \$t0
- Load Immediate              li \$t0,56                    # load immediate value into \$t0



## Compare and Branch

- branch on equal      `beq $1,$2,100`      if ( $\$1 == \$2$ ) go to  $PC+4+100$
- branch on not eq.      `bne $1,$2,100`      if ( $\$1 \neq \$2$ ) go to  $PC+4+100*4$
- set on less than      `slt $1,$2,$3`      if ( $\$2 < \$3$ )  $\$1=1$ ; else  $\$1=0$
- set less than imm.      `slti $1,$2,100`      if ( $\$2 < 100$ )  $\$1=1$ ; else  $\$1=0$
- set less than uns.      `sltu $1,$2,$3`      if ( $\$2 < \$3$ )  $\$1=1$ ; else  $\$1=0$
- set l. t. imm. uns.      `sltiu $1,$2,100`      if ( $\$2 < 100$ )  $\$1=1$ ; else  $\$1=0$
- jump      `j 10000`      go to 40000
- jump register      `jr $31`      go to \$31
- jump and link      `jal 10000`       $\$31 = PC + 4$ ; go to 40000

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Operações Aritméticas e Lógicas:

- Exemplo:  $f = (g + h) - (i + j)$

`add $t0, $s1, $s2    # t0 = g + h`

`add $t1, $s3, $s4    # t1 = i + j`

`sub $s0, $t0, $t1    # f = t0 - t1`

- Exercício: mostre o OPPOSITE (em hexadecimal) de cada uma das instruções acima.

f	\$s0
g	\$s1
h	\$s2
i	\$s3
j	\$s4

## ISA (Instruction Set Architecture) MIPS - Registradores -

### Operações Aritméticas e Lógicas:

- Exemplo:  $a = (b + 10) + (c - 5) - (d - e)$

`addi $s0, $s1, 10`    #  $s0 = b + 10$

`addi $t0, $s2, -5`    #  $t0 = c + (-5)$

`sub $t1, $s3, $s4`    #  $t1 = d - e$

`add $s0, $s0, $t0`

`sub $s0, $s0, $t1`

a	\$s0
b	\$s1
c	\$s2
d	\$s3
e	\$s4

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Operações Aritméticas e Lógicas:

- Exemplo:  $a = a \mid (0x20)$

**ori   \$s0, \$s0, 0x20   # a = a | 0x20**

a	\$s0
b	\$s1
c	\$s2
d	\$s3
e	\$s4

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Operações Aritméticas e Lógicas:

- Exemplo:  $a = a \mid (1 \ll 5)$

**addi \$t0, \$zero, 1      # t0 = 0 + 1**

**sll \$t0, \$t0, 5      # t0 = 1 << 5**

**or \$s0, \$s0, \$t0      # a = a | t0**

a	\$s0
---	------

b	\$s1
---	------

c	\$s2
---	------

d	\$s3
---	------

e	\$s4
---	------

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Operações Aritméticas e Lógicas:

Exercícios:

1) Escreva os comandos em Assembly do MIPS para as seguintes expressões:

a)  $a = d - e + 40$

b)  $e = e \& \sim(0x80)$

2) Escreva os Opcodes gerados em cada uma das instruções das expressões acima

a	\$s0
b	\$s1
c	\$s2
d	\$s3
e	\$s4

## Transferência de dados (memória/registrador - registrador/memória):

- O acesso à memória no MIPS é feito semelhante a um vetor
- Cada acesso é feito em valores de 1 palavra do MIPS (32 bits)
- Especifica-se o endereço BASE e um deslocamento (offset) a partir deste endereço.
- O offset do endereço deve ser um valor múltiplo de 4 bytes (32 bits)

Conteúdo da Memória	
Endereço	Dados
7	F0 h
6	DE h
5	BC h
4	9A h
3	78 h
2	56 h
1	34 h
0	12 h

Offset →

Endereço base →

## ISA (Instruction Set Architecture) MIPS - Registradores -

**Load Word (lw)** → carrega um dado (32 bits) da memória para o registrador

Exemplo: **lw \$t0, 4(\$t5)**    # reg 13=\$t5, reg 08=\$t0

**Banco de Registradores**

No.	Valor
14	990A 6574
13	0000 0008
12	3159 D4F8
11	2F6D 851B
10	1F6C 2843
09	5502 0033
08	1234 5678

**Memória**

Endereço	Dados
00000000	9016 6000
00000004	FF66 6E30
00000008	0000 0000
0000000C	1234 5678
00000010	0000 012F
00000014	4500 5671
00000018	0000 0000
0000001C	0000 0000

\*\* Valores em Hexadecimal



## ISA (Instruction Set Architecture) MIPS - Registradores -

**Store Word (sw)** → carrega o valor do registrador para a memória

Exemplo: **sw \$t1, 8(\$t5)**    # reg 13=\$t5, reg 09=\$t1

**Banco de Registradores**

No.	Valor
14	990A 6574
13	0000 0008
12	3159 D4F8
11	2F6D 851B
10	1F6C 2843
09	5502 0033
08	1234 5678

**Memória**

Endereço	Dados
00000000	9016 6000
00000004	FF66 6E30
00000008	0000 0000
0000000C	1234 5678
00000010	5502 0033
00000014	4500 5671
00000018	0000 0000
0000001C	0000 0000

\*\* Valores em Hexadecimal

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

**Exemplo 1**: Qual o código Assembly para:  $A[12] = h + A[8]$  ?

OBS: Considerar que o valor da variável  $h$  esteja em  $\$s2$  e que o endereço base do vetor  $A$  esteja em  $\$s3$ .

```
lw  $t0, 32($s3)    # $t0 = A[8]
add $t0, $s2, $t0    # $t0 = h + A[8]
sw  $t0, 48($s3)    # A[12] = h + A[8]
```

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

**Exemplo 2:** Qual o código Assembly para trocar os valores de B[10] com B[11]?

OBS: Considerar que o endereço base do vetor B esteja em \$s4.

```
lw  $t0, 40($s4)    # $t0 = B[10]
lw  $t1, 44($s4)    # $t1 = B[11]
sw  $t0, 44($s4)    # B[11] = B[10]
sw  $t1, 40($s4)    # B[10] = B[11]
```

## Exercícios

Realize as seguintes operações utilizando a linguagem Assembly do MIPS:

a)  $A[15] = h + B[6];$

\$s0 : h

\$s1: endereço base de A

\$s2: endereço base de B

b)  $A[15] = A[5] + B[6] + B[0];$

\$s1: endereço base de A

\$s2: endereço base de B

c)  $B[4] = A[12] - A[5] + A[3]$

\$s1: endereço base de A

\$s2: endereço base de B

## ISA (*Instruction Set Architecture*) MIPS - *Registradores* -

### Desvio (branch) Condicional:

Branch if equal (beq):

*beq reg1, reg2, Label1*

- Se conteúdo do registrador1 igual ao conteúdo do registrador2, desvie para o *Label1*

Exemplo:

*beq \$t1, \$t0, Loop*

## ISA (*Instruction Set Architecture*) MIPS - Registradores -

### Desvio (branch) Condicional:

Branch if not equal (bne):

**bne reg1, reg2, Label1**

- Se conteúdo do registrador1 é diferente do conteúdo do registrador2, desvie para o *Label1*

Exemplo:

*bne \$t1, \$t0, volta*

## ISA (*Instruction Set Architecture*) MIPS - *Registradores* -

### Desvio Incondicional (jump):

**j Label**

- Salta (jump) para o *Label*1

Exemplo:

*j frente*

## ISA (Instruction Set Architecture) MIPS - Registradores -

### Desvio Condicional e Incondicional:

Exemplo: Como implementar em Assembly a estrutura a seguir:

```
if (a == 0)
    a = a+1;
else
    a = a-1;
```

Rearranjar o código



```
if (a == 0) goto L1;
    a = a - 1;
    goto L2;
L1: a = a + 1;
L2:
```



## ISA (Instruction Set Architecture) MIPS - Registradores -

### Desvio Condicional e Incondicional:

- Considerando A = \$s0

```
beq $s0, $zero, L1  # se a = 0 vai para L1
addi $s0, $s0, -1    # faz a = a - 1
j L2                 # vai para L2
```

```
L1: addi $s0,$s0,1    # a = a+1
```

```
L2:
```

```
if (a == 0)
    a = a+1;
else
    a = a-1;
```

```
if (a == 0) goto L1;
    a = a - 1;
    goto L2;
L1: a = a + 1;
L2:
```

## ISA (Instruction Set Architecture) MIPS - Registradores -

### Exercício:

*Escrever em código Assembly do MIPS a seguinte estrutura:*

```
do{  
    a = a + 1;  
}while (a != 10);
```

OBS: considerar a = \$s3

## Exercícios

Traduza os seguintes trechos de código para o assembly do MIPS (considere  $a = \$s0$ ,  $b = \$s1$  e  $c = \$s2$ ) :

a)  $b = 0;$   
do{  
     $b = b + 5;$   
}while( $b \neq 50$ );

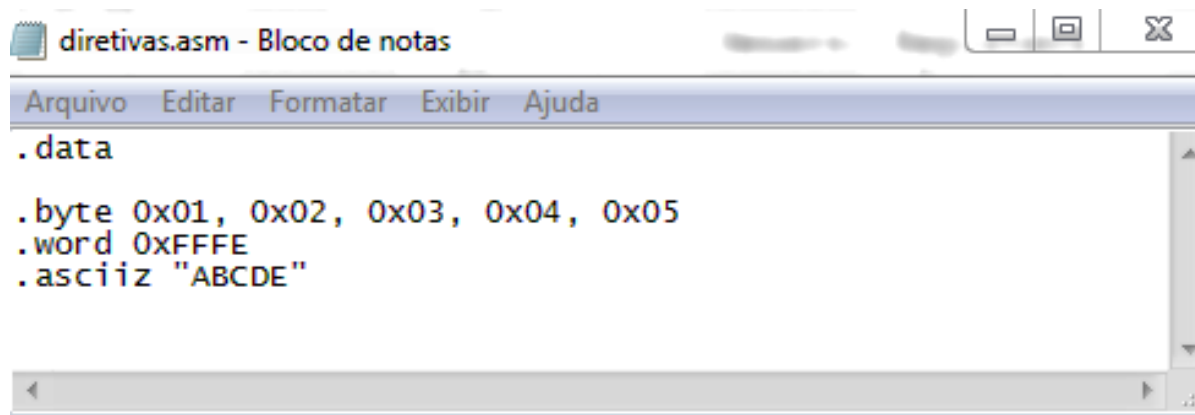
b) if ( $b == 10$ )  
     $b = b - 1;$   
else  
     $b = b * 2;$

c) if ( $a == b$ )  
     $a = a + 10;$   
else  
     $a = a - 8;$

d) while ( $a \neq 0$ )  
{  
     $b = b + c;$   
     $a = a + 1;$   
}

## Exercícios:

1- De acordo com as diretivas abaixo, pede-se:



```
diretivas.asm - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
.data

.byte 0x01, 0x02, 0x03, 0x04, 0x05
.word 0xFFFFE
.asciiz "ABCDE"
```

- a) Os dados foram armazenados a partir de qual endereço?
- b) Mostre como ficam os dados na memória

## Exercícios:

Analise o programa abaixo e responda:

```
.text  
main:  addi $t0, $zero, 3  
        addi $t1, $zero, 1  
        add $t2, $t0, $t1  
        jr $ra
```

- a) Mostre os valores dos registradores após a execução de cada instrução
- b) Qual a finalidade do jr \$ra no programa?
- c) Identifique no programa: registradores, constante, instruções, labels e diretivas.

## MIPS – Serviços do Sistema

O código do serviço deve estar no registrador \$v0

Serviço	Código de chamada do sistema	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (em \$v0)
read_float	6		float (em \$f0)
read_double	7		double (em \$f0)
read_string	8	\$a0 = buffer, \$a1 = tamanho	
sbrk	9	\$a0 = valor	endereço (em \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (em \$a0)
open	13	\$a0 = nome de arquivo (string), \$a1 = flags, \$a2 = modo	descriptor de arquivo (em \$a0)
read	14	\$a0 = descriptor de arquivo, \$a1 = buffer, \$a2 = tamanho	número de caracteres lidos (em \$a0)
write	15	\$a0 = descriptor de arquivo, \$a1 = buffer, \$a2 = tamanho	número de caracteres escritos (em \$a0)
close	16	\$a0 = descriptor de arquivo	
exit2	17	\$a0 = resultado	

## MIPS – Serviços do Sistema

### Exemplo de uso dos Serviços do Sistema

```
.data  
str: .asciiz "INATEL – C-208 - MIPS"
```

```
.text  
main:  
    li $v0,4  
    la $a0,str  
    syscall  
  
    li $v0,1  
    li $a0,0xF5  
    syscall  
  
    jr $ra
```

O **código do serviço** deve estar no registrador **\$v0**

Serviço	Código de chamada do sistema	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (em \$v0)

Console

```
INATEL - C-208 - MIPS245
```

## Exercícios:

1- Faça um programa em Assembly do MIPS que leia 2 números inteiros e mostre como resultado qual é o maior número

2- Faça um programa em Assembly do MIPS que leia 4 números (A,B,C e D) e execute a seguinte expressão:  $X = (A-B) + (C-D)$ . Mostre o resultado na tela.

3- Faça um programa em Assembly do MIPS para calcular o fatorial de um número. Mostrar o resultado na tela (ver algoritmo a seguir).

4- Faça um programa em Assembly do MIPS que leia 3 números inteiros e mostre como resultado a média aritmética destes números



## Algoritmo de Fatorial – Exercício 3

ALGORITMO FATORIAL

[declaração de Variáveis]

Inteiro FAT, NUM;

Início

FAT <- 1;

Escreva “Digite um numero inteiro e positivo:”;

Leia NUM;

Se (NUM > 0)

então Enquanto (NUM > 1) faça:

início

FAT <- FAT \* NUM;

NUM <- NUM - 1;

fim

Fim-se

Escreva “O fatorial de ”, NUM, “ = “, FAT;

FIM /