

# Aula 1

- Conceitos fundamentais em Arquitetura de Computadores
  - O Computador como sistema digital
  - Os elementos básicos de um computador
  - O ciclo básico de execução de uma instrução
- Arquitetura de Computadores
  - *Instruction Set Architecture* (ISA)
  - Organização
  - Níveis de Representação

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



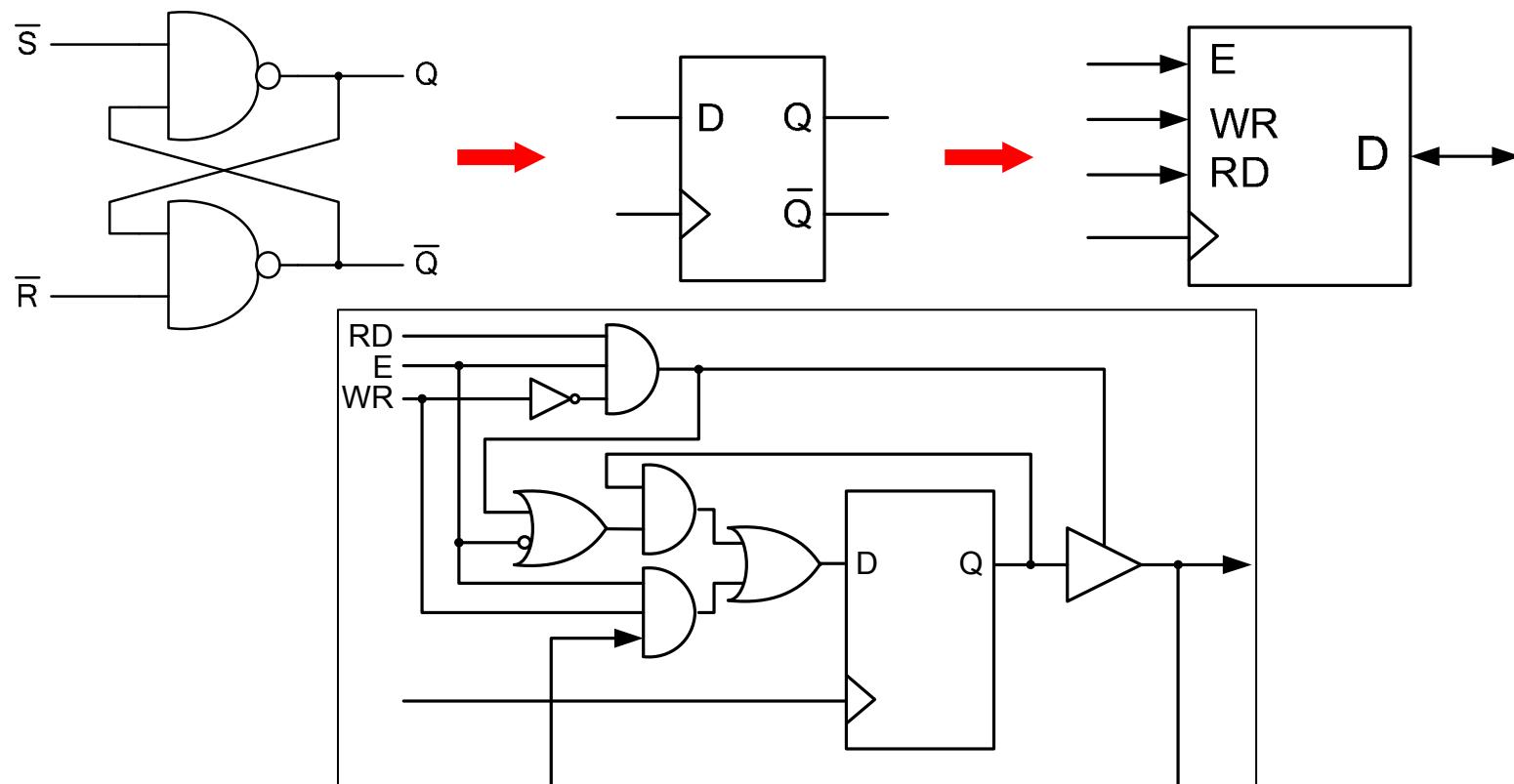
# Arquitetura de Computadores e Sistemas Digitais

- Arquitetura de Computadores é uma das áreas de aplicação direta dos conceitos, técnicas e metodologias apreendidas nas duas UCs de Sistemas Digitais
- Em Arquitetura de Computadores, contudo, trabalha-se num nível de abstração diferente
- Recorre-se, na maior parte das vezes, a **blocos funcionais complexos** com cuja síntese, normalmente, não temos que nos preocupar (isso não significa que a sua funcionalidade não tenha que ser totalmente compreendida)



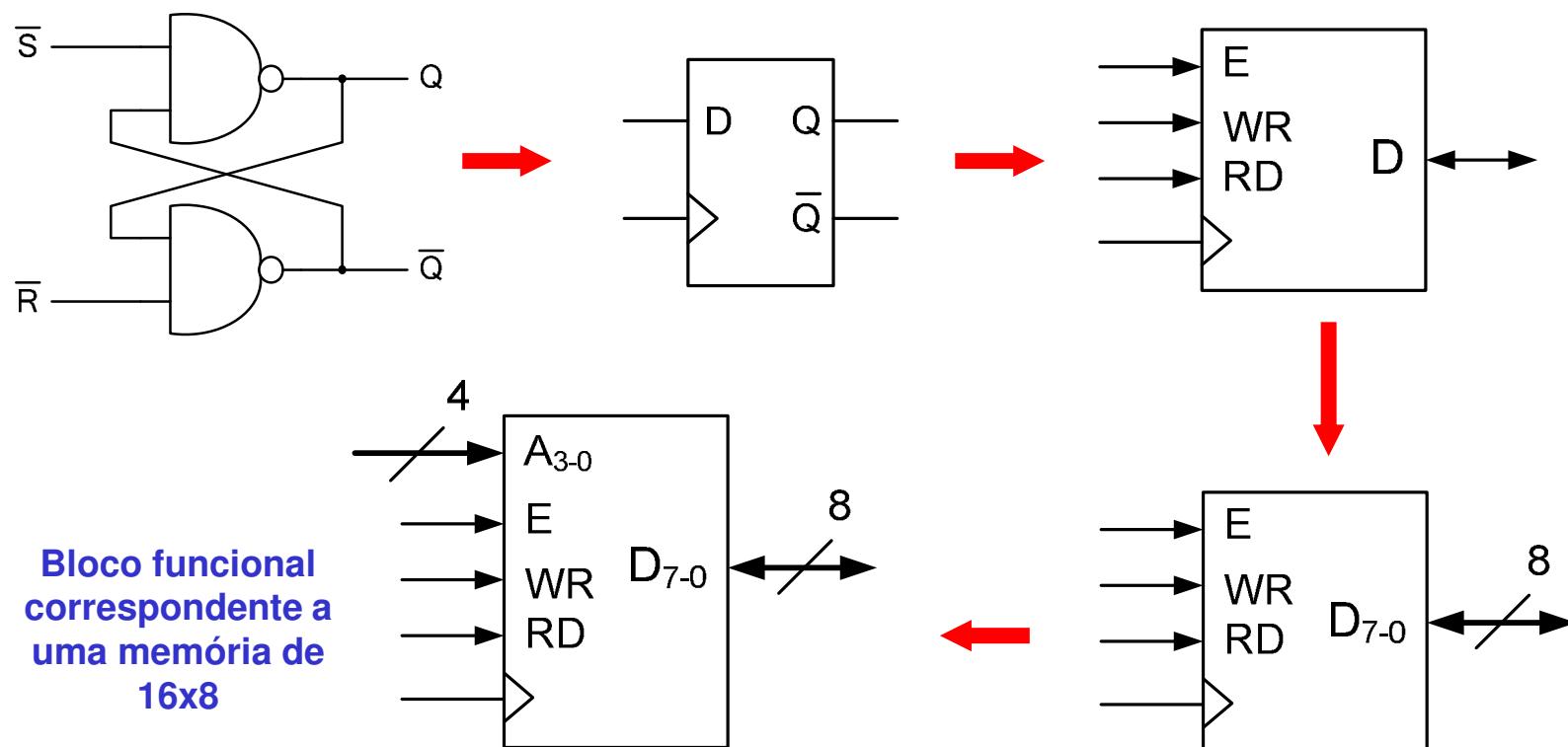
# Exemplo: memória RAM 16x8

- Por exemplo, uma "Memória" (um dispositivo com capacidade para armazenar informação digital binária) pode ser construída à custa de blocos básicos bem conhecidos dos sistemas digitais: **flip-flops**



# Exemplo: memória RAM 16x8

- Por exemplo, uma "Memória" (um dispositivo com capacidade para armazenar informação digital binária) pode ser construída à custa de blocos básicos bem conhecidos dos sistemas digitais: **flip-flops**



# Arquitetura básica de um sistema computacional

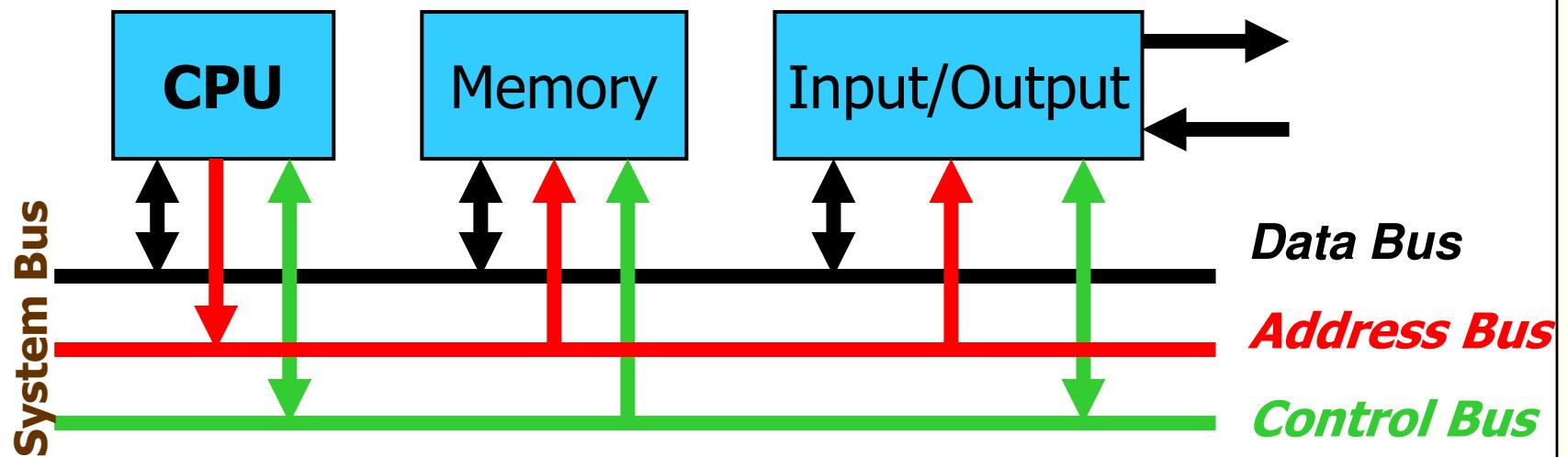
- Unidades fundamentais que constituem um computador
  - **CPU** – responsável pelo processamento da informação através da execução de uma sequência de instruções (programa) armazenadas em memória
  - **Memória** – responsável pelo armazenamento de:
    - Programas
    - Dados para processamento
    - Resultados
  - **Unidades de I/O** – responsáveis pela comunicação com o exterior
    - **Unidades de entrada** – permitem a receção de informação vinda do exterior (dados, programas) e que é armazenada em memória
    - **Unidades de saída** – permitem o envio de resultados para o exterior
- Um computador é um sistema digital complexo

Cada um destes blocos é  
um sistema digital!



# Arquitetura básica de um sistema computacional

- Modelo de von Neumann



- **Data Bus**: barramento de transferência de informação (CPU↔memória, CPU↔Input/Output)
- **Address Bus**: identifica a origem/destino da informação
- **Control Bus**: sinais de protocolo que especificam o modo como a transferência de informação deve ser feita



# Arquitetura básica de um sistema computacional

- **Endereço (address)** – um número (único) que identifica cada registo de memória. Os endereços são contados sequencialmente, começando em 0
  - Exemplo: o conteúdo da posição de memória 0x2000 é 0x32 – (0x2000 é o endereço, 0x32 o valor armazenado)
- **Espaço de endereçamento (address space)** – a gama total de endereços que o CPU consegue referenciar (depende da dimensão do barramento de endereços).
  - Exemplo: um CPU com um barramento de endereços de 16 bits pode gerar endereços na gama: 0x0000 a 0xFFFF (i.e., 0 a  $2^{16}-1$ )
  - Qual o espaço de endereçamento de um processador com um barramento de endereços de 32 bits?



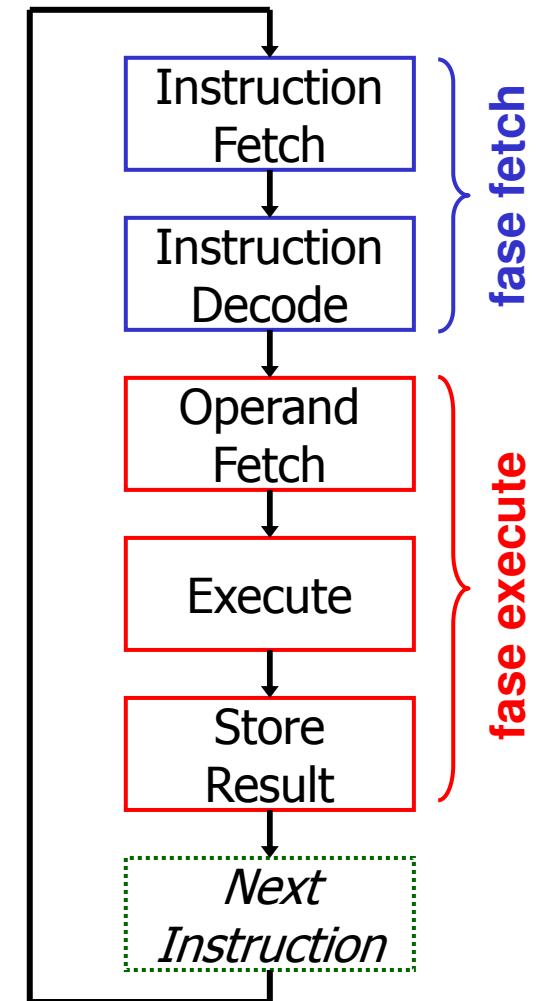
# Arquitetura básica do CPU

- **Secção de dados** (*datapath*) – elementos operativos/funcionais para encaminhamento, processamento e armazenamento de informação
  - Multiplexers
  - Unidade Aritmética e Lógica (ALU) – Add, Sub, And, Or...
  - Registos internos
- **Unidade de controlo** – responsável pela coordenação dos elementos do *datapath*, durante a execução de um programa
  - Gera os sinais de controlo que adequam a operação de cada um dos recursos da secção de dados às necessidades da instrução que estiver a ser executada
  - Dependendo da arquitetura, pode ser uma máquina de estados ou um elemento meramente combinatório
- Independentemente da Unidade de Controlo ser combinatória ou sequencial, **o CPU é sempre uma máquina de estados síncrona**

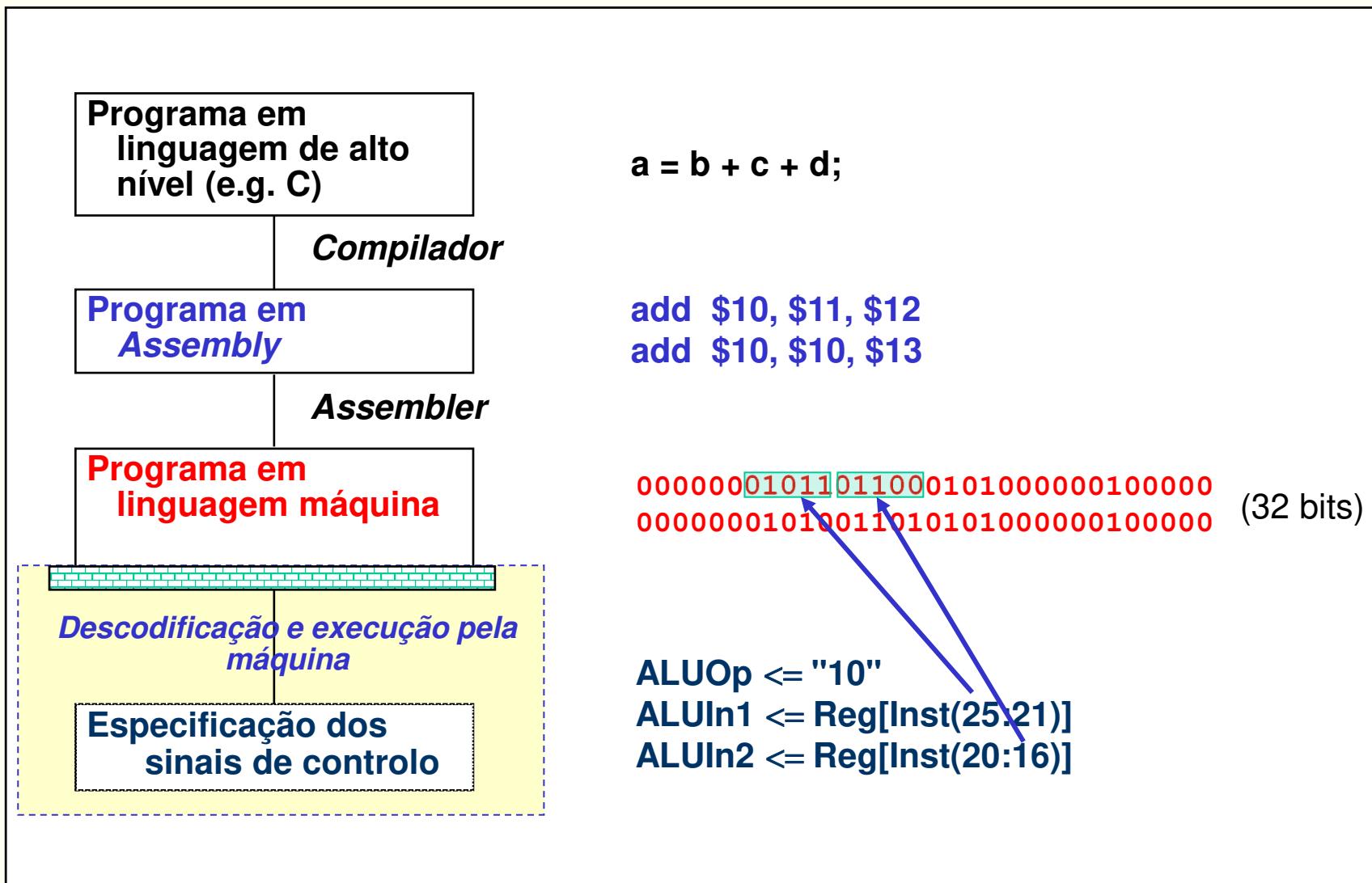


# Ciclo-base de execução de uma instrução

- **Instruction fetch:** leitura do código máquina da instrução (instrução reside em memória)
- **Instruction decode:** descodificação da instrução pela unidade de controlo
- **Operand fetch:** leitura do(s) operando(s)
- **Execute:** execução da operação especificada pela instrução
- **Store result:** armazenamento do resultado da operação no destino especificado na instrução



# Níveis de Representação



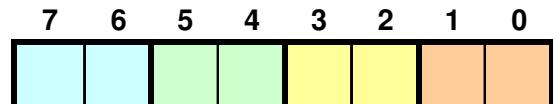
# Codificação das instruções

- A **codificação de uma instrução**, sob a forma de um número expresso em binário, terá que ter toda a informação de que o CPU necessita para a sua execução
- Qual a operação a realizar ?
- Qual a localização dos operandos (se existirem) ?
  - podem estar em **registos internos do CPU** ou na **memória externa**. No 1º caso deverá ser especificado o número de um registo; no 2º um endereço de memória
- Onde colocar o resultado ?
  - **Registros internos / memória**
- Qual a próxima instrução a executar?
  - em condições normais é a instrução seguinte na sequência e, portanto, não é, normalmente, explicitamente mencionada
  - em instruções que **alteram a sequência de execução** a instrução deverá fornecer o endereço da próxima instrução a ser executada



# Exemplo - CPU hipotético

## Formato de codificação das instruções (8 bits)



Formato 1



Formato 2

## Registros Internos do CPU:

|    |        |
|----|--------|
| 00 | Reg. 0 |
| 01 | Reg. 1 |
| 10 | Reg. 2 |
| 11 | Reg. 3 |

## Operações possíveis:

- 00 Somar o conteúdo de dois registos
- 01 Ler da memória para um registo interno do CPU (LOAD)
- 10 Escrever o conteúdo de um registo interno na memória (STORE)
- 11 Não definida (N.D.)

Exemplo de programa em código máquina  
para este processador

Qual é a expressão aritmética  
implementada neste programa?

| Hex  | Binário  |  |
|------|----------|--|
| 0x58 | 01011000 | Ler o conteúdo da posição de memória 8 para o registo interno 1        |
| 0x79 | 01111001 | Ler o conteúdo da posição de memória 9 para o registo interno 3        |
| 0x15 | 00010101 | Somar o conteúdo do reg. 1 c/ o reg. 1 e depositar o result. no reg. 1 |
| 0x07 | 00000111 | Somar o conteúdo do reg. 1 c/ o reg. 3 e depositar o result. no reg. 0 |
| 0x8A | 10001010 | Escrever o conteúdo do reg. 0 na posição de memória 10                 |



# Arquitetura do Conjunto de Instruções (ISA)

- **ISA**: Instruction Set Architecture
- **Instruction Set**: coleção de todas as operações/instruções que o processador pode executar
- Arquitetura de Computadores =  
**Arquitetura do Conjunto de Instruções (ISA) + Organização da Máquina**



# Arquitetura do Conjunto de Instruções (ISA)

- Também designada por "modelo de programação"
- Uma importante abstração que representa a **interface entre o h/w e o nível mais básico de s/w**
- Descreve tudo o que o programador necessita de saber para programar corretamente, em linguagem máquina, um determinado processador
- Descreve a funcionalidade, independentemente do h/w que a implementa. Pode assim falar-se de "**arquitetura**" e "**implementação de uma arquitetura**"
- Exemplo em que a mesma arquitetura do conjunto de instruções tem 2 implementações distintas:
  - Processadores AMD compatíveis com Intel x86



# Arquitetura do Conjunto de Instruções (ISA)

- Alguns exemplos de ISAs:
  - MIPS
  - ARM (Nintendo DS, iPod, Canon PowerShot, smartphones, ...)
  - Intel x86 (PCs, MACs)
  - PowerPC
  - Cell (playstation 3)

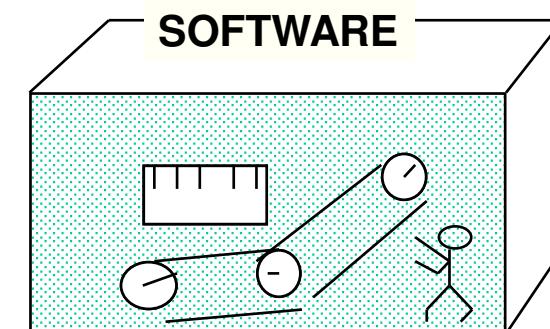


# Arquitetura do Conjunto de Instruções (ISA)

- ... os atributos de um sistema computacional tal como são vistos pelo programador, i.e. a estrutura conceitual e o comportamento funcional, de forma distinta e independente da organização do fluxo de informação e dos respetivos elementos de controlo, do desenho lógico e da implementação física.

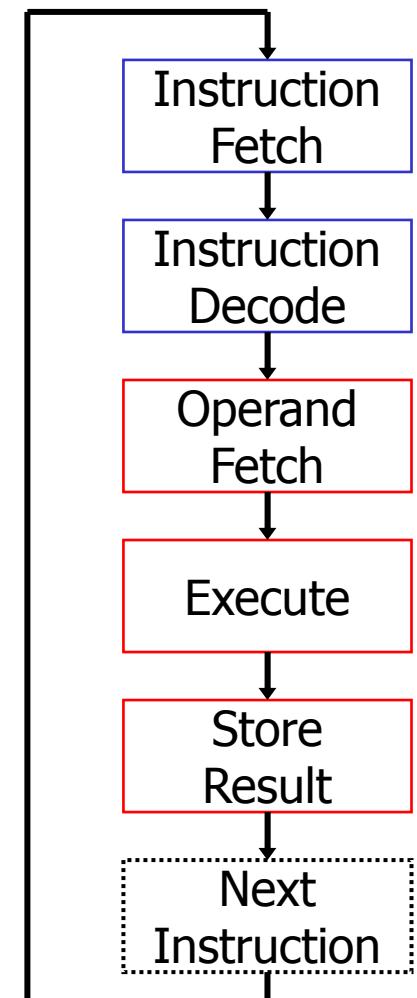
— Amdahl, Blaaw, and Brooks, 1964

- Conjunto de Instruções
- Organização da memória
- Número e tipos de Registros
- Tipos de dados e estruturas de dados (Codificação e representação)
- Modos de endereçamento (acesso a dados e instruções)
- Formato/codificação das Instruções
- Condições que podem desencadear "exceções"



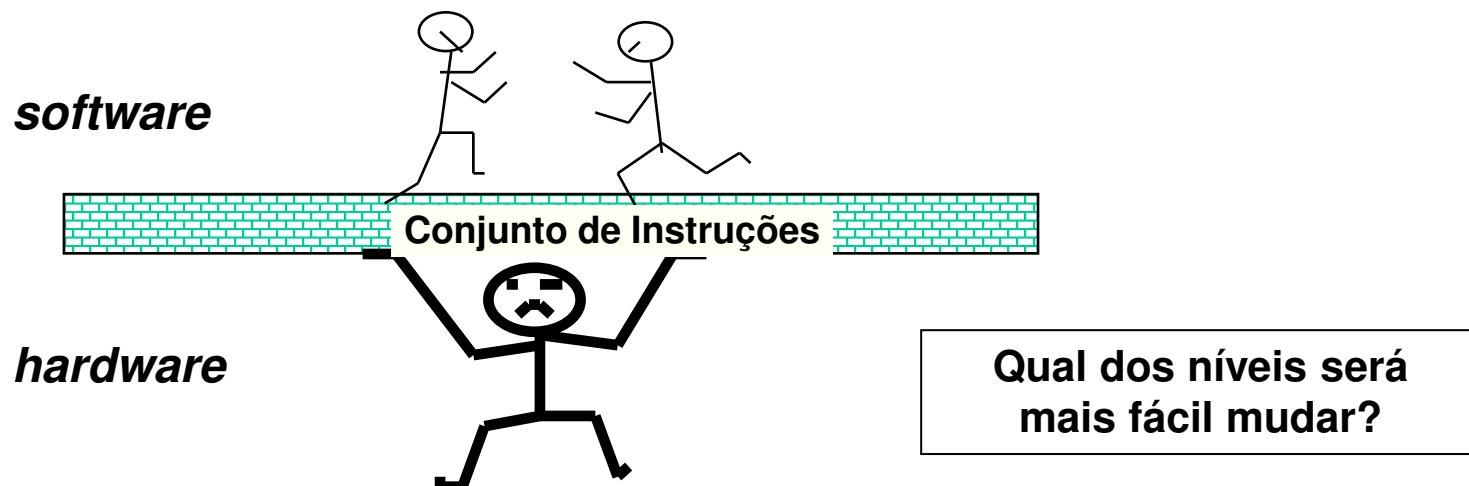
# Arquitetura do Conjunto de Instruções (ISA)

- Formato e codificação das instruções
  - como são descodificadas?
- Operандos das instruções e resultados
  - onde podem residir?
  - quantos operandos explícitos?
  - como referenciar?
  - quais podem residir na memória externa?
- Tipo e dimensão dos dados
- Operações
  - quais são suportadas?
  - instruções auxiliares: jumps, conditions, branches (para controlo do fluxo de execução)



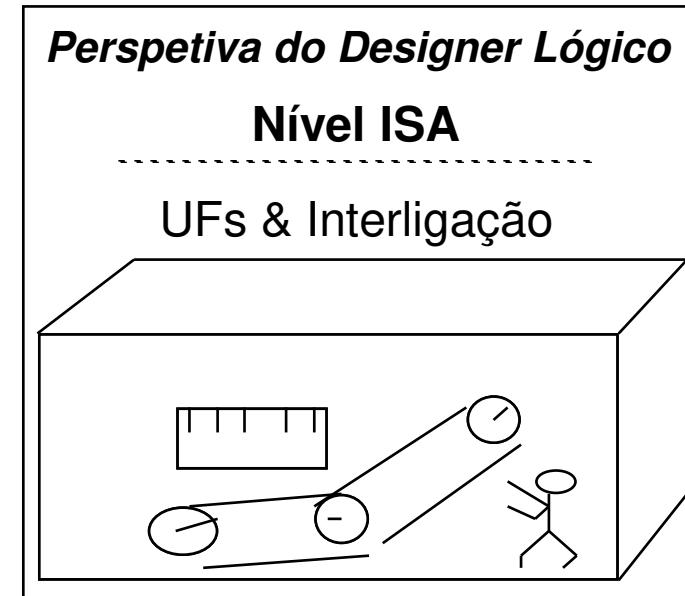
# Arquitetura do Conjunto de Instruções (ISA)

- Requisitos básicos da Arquitetura do Conjunto de Instruções:
  - Implementação simples e eficiente em hardware
  - Fácil de entender e programar
  - Desenvolvimento de compiladores eficientes
- Conjunto de Instruções: um Interface Crítico

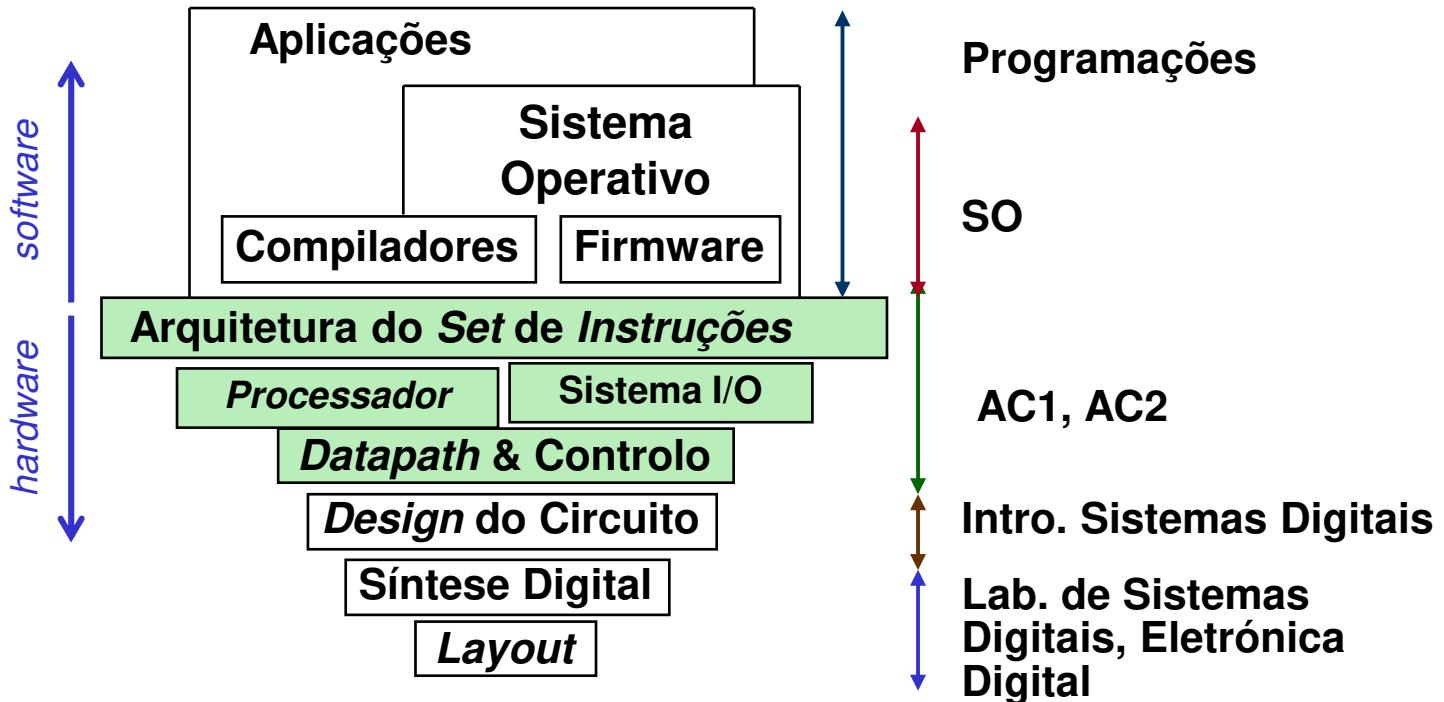


# Organização da máquina

- Características operativas e de desempenho das principais unidades funcionais (ALU, Registros, Shifters, Unidades Lógicas, ...)
- De que modo esses componentes são interligados
- Fluxo de informação entre componentes
- Lógica e meios através dos quais esse fluxo é controlado
- Coreografia das Unidades Funcionais para implementar a ISA



# Arquitetura dos Sistemas Computacionais

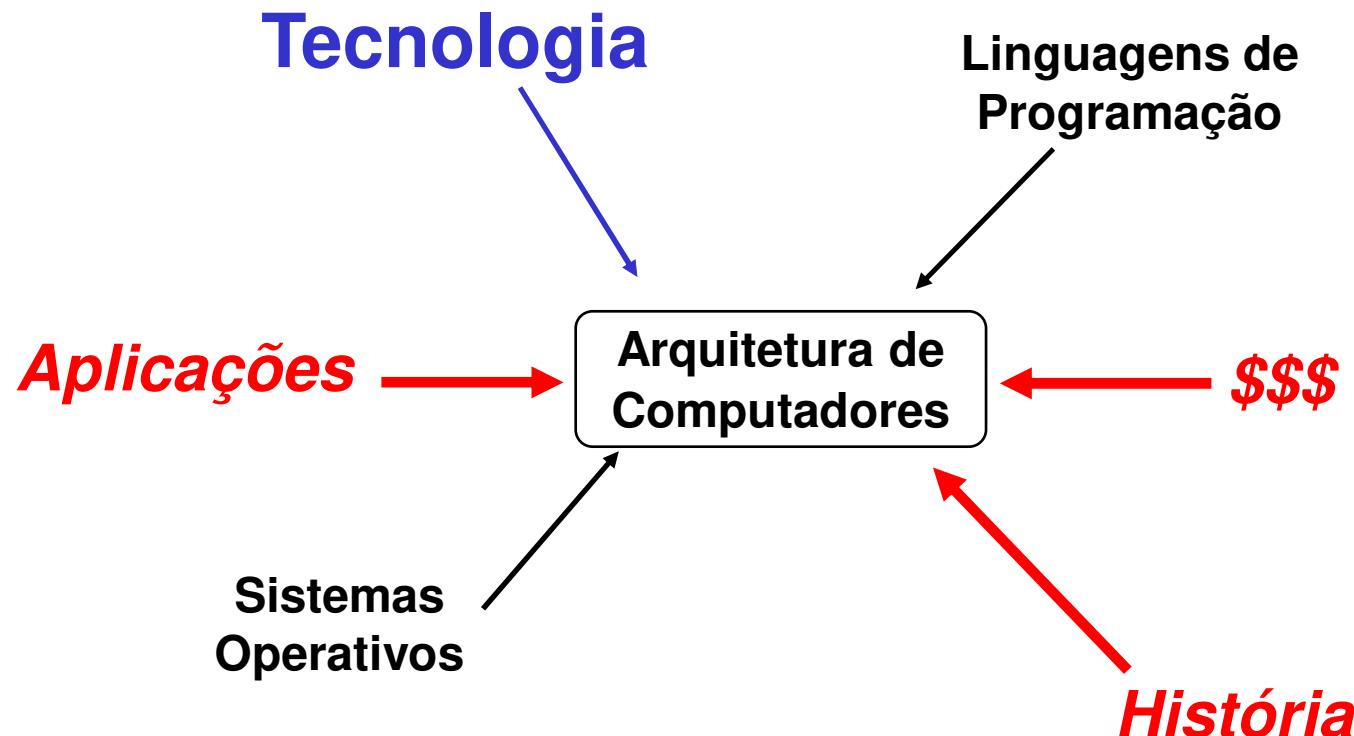


- Coordenação de muitos níveis de abstração...
- Sob o efeito de um conjunto de forças em permanente mutação
- Design, Medida e Avaliação



# Conclusão

- A evolução da arquitetura de computadores depende de múltiplos fatores

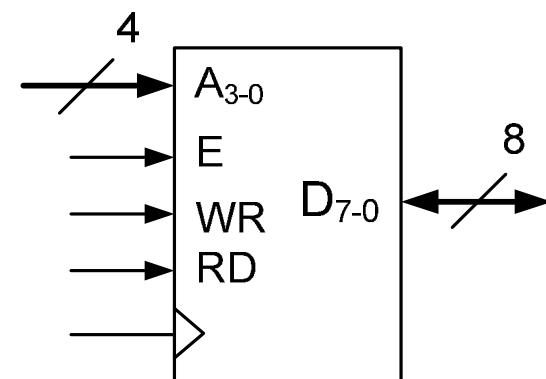


# Exemplo: memória RAM 16x8 – VHDL

- O mesmo bloco pode, contudo, ser modelado numa linguagem de descrição de hardware, por exemplo VHDL, usando para isso uma mera descrição comportamental:

```
entity RAM_16_8 is
    port(clk      : in std_logic;
          addr     : in std_logic_vector(3 downto 0);
          enable   : in std_logic;
          wr       : in std_logic;
          rd       : in std_logic;
          data_io  : inout std_logic_vector(7 downto 0));
end RAM_16_8;
```

Escrita síncrona e leitura assíncrona.  
O barramento de dados é bidirecional.

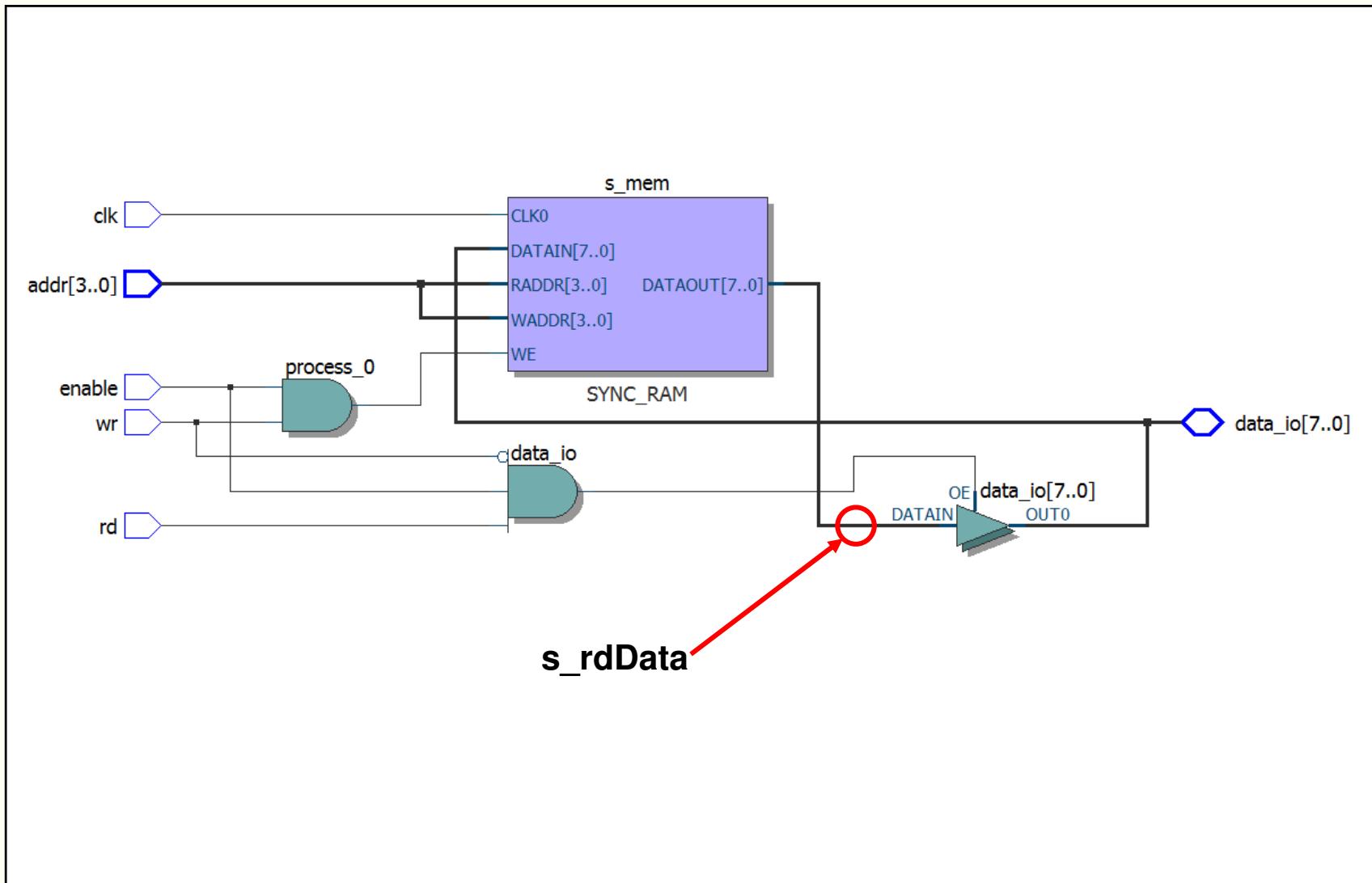


# Exemplo: memória RAM 16x8 – VHDL

```
architecture Behavioral of RAM_16_8 is
    subtype TData is std_logic_vector(7 downto 0);
    type TMemory is array(0 to 15) of TData;
    signal s_mem : TMemory;
    signal s_rdData : std_logic_vector(7 downto 0);
begin
    process(clk)
    begin
        if(rising_edge(clk)) then
            if(enable = '1' and wr = '1') then
                s_mem(to_integer(unsigned(addr))) <= data_io;
            end if;
        end if;
    end process;
    s_rdData <= s_mem(to_integer(unsigned(addr)));
    data_io <= s_rdData when enable = '1' and rd = '1' and
        wr = '0' else (others => 'Z');
end Behavioral;
```



# Exemplo: memória RAM 16x8 - síntese



## Aula 2

- Instruções e classes de instruções
- Princípios básicos de projeto de uma Arquitetura
- Aspetos chave da arquitetura MIPS
- Instruções aritméticas
- Instruções lógicas e de deslocamento
- Codificação de instruções no MIPS: formato R

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

# Introdução: a máquina e a sua linguagem

- Princípios básicos dos computadores atuais:
  - As instruções são representadas da mesma forma que os números
  - Os programas são armazenados em memória, para serem lidos e escritos, tal como os números
- Estes princípios formam os fundamentos do conceito da arquitetura "**stored-program**"
  - O conceito "stored-program" implica que na memória possa residir, ao mesmo tempo, informação de natureza tão variada como: o código fonte de um programa em C, um editor de texto, um compilador, e o próprio programa resultante da compilação

# ISA: Instruções e classes de instruções

***“It is easy to see by formal-logical methods that there exist certain instruction sets that are in abstract adequate to control and cause the execution of any sequence of operations...***

*The really decisive considerations from the present point of view, in selecting an instruction set, are more of a practical nature: simplicity of the equipment demanded by the instruction set, and the clarity of its application to the actually important problems together with the speed of its handling of those problems”*

*Burks, Goldstine and von Neumann, 1947*

# Instruções e classes de instruções

Será, portanto, possível considerar a existência de um grupo limitado de classes de instruções que possam ser consideradas comuns à generalidade das arquiteturas?

*There must certainly be instructions for performing the fundamental arithmetic operations*

*Burks, Goldstine and von Neumann, 1947*

- **Classes de instruções:**

- Processamento (aritméticas e lógicas)
- Transferência de informação
- Controlo de fluxo de execução

# Instruções e implementação hardware

- No projeto de um processador a definição do ***instruction set*** exige um delicado compromisso entre múltiplos aspectos, nomeadamente:
  - as facilidades oferecidas aos programadores (por ex. instruções de manipulação de *strings*)
  - a complexidade do hardware envolvido na sua implementação
- Quatro princípios básicos estão subjacentes a um bom design ao nível do hardware:
  - A regularidade favorece a simplicidade
  - Quanto mais pequeno mais rápido
  - O que é mais comum deve ser mais rápido
  - Um bom design implica compromissos adequados

# Instruções e implementação hardware

- **A regularidade favorece a simplicidade**
  - Ex1: todas as instruções do *instruction set* são codificadas com o mesmo número de bits
  - Ex2: instruções aritméticas operam sempre sobre registos internos e depositam o resultado também num registo interno
- **Quanto mais pequeno mais rápido**
- **O que é mais comum deve ser mais rápido**
  - Ex: quando o operando é uma constante esta deve fazer parte da instrução (é vulgar que mais de 50% das instruções que envolvem a ALU num programa utilizem constantes)
- **Um bom *design* implica compromissos adequados**
  - Ex: o compromisso que resulta entre a possibilidade de se poder codificar constantes de maior dimensão nas instruções e a manutenção da dimensão fixa nas instruções

## Arquitetura do set de instruções (ISA). Relembrando...

- Formato e codificação das instruções
  - como são descodificadas?
- Operandos das instruções e resultados
  - onde podem residir?
  - quantos operandos explícitos?
  - como localizar?
  - quais podem residir na memória externa?
- Tipo e dimensão dos dados
- Operações
  - quais são suportadas?
- Instruções auxiliares
  - jumps, conditions, branches

# ISA – formato e codificação das instruções

- Tamanho variável
  - Código mais pequeno
  - Maior flexibilidade
  - *Instruction fetch* em vários passos
- Tamanho fixo
  - *Instruction fetch* e *decode* mais simples
  - Mais simples de implementar em *pipeline*

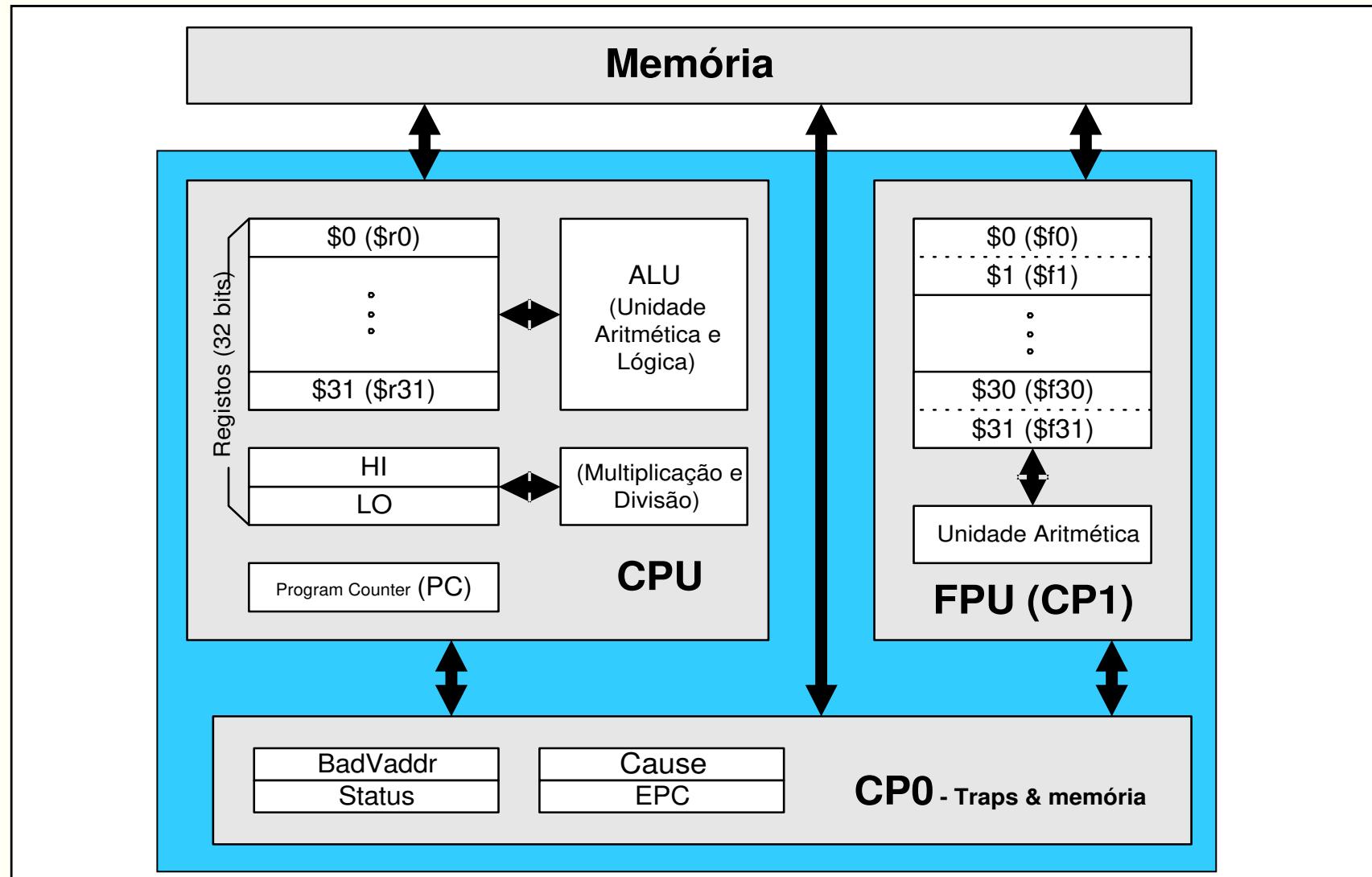
# ISA – número de registos internos do CPU

- Vantagens de um número pequeno de registos
  - Menos hardware
  - Acesso mais rápido
  - Menos bits para identificação do registo
  - Mudança de contexto mais rápida
- Vantagens de um número elevado de registos
  - Menos acessos à memória
  - Variáveis em registos
  - Certos registos podem ter restrições de utilização

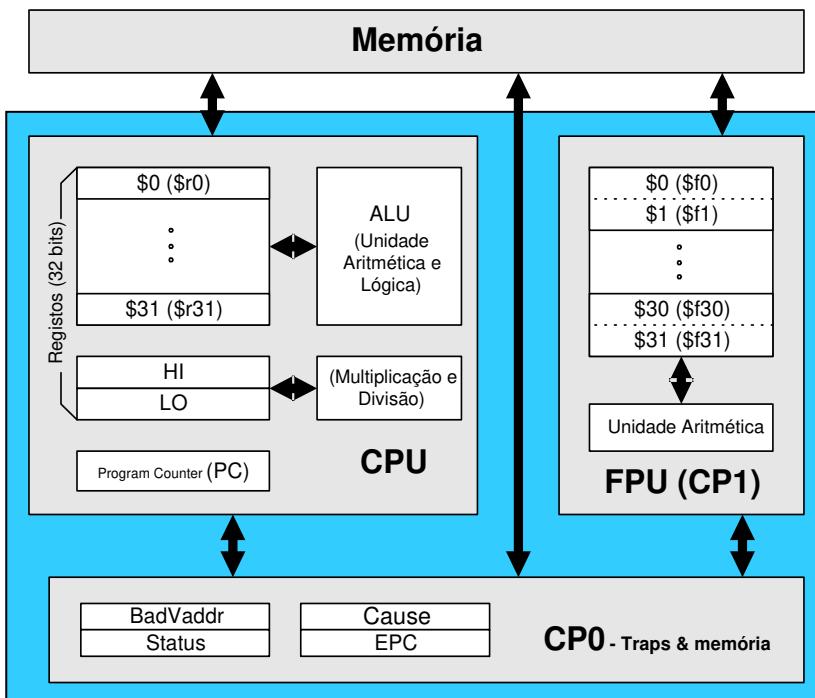
# ISA – localização dos operandos das instruções

- Arquiteturas baseadas em **acumulador**
  - Resultado das operações é armazenado num registo especial designado de acumulador
- Arquiteturas baseadas em **Stack**
  - Operandos e resultado armazenados numa *stack* (pilha) de registos
- Arquiteturas **Register-Memory**
  - Operandos residem em registos internos do CPU ou em memória
- Arquiteturas **Load-store**
  - Operandos das instruções residem em registos internos do CPU de uso geral (mas nunca na memória).

# Arquitetura MIPS (Microprocessor without Interlocked Pipeline Stages)



# Aspetos chave da arquitetura MIPS



- 32 Registros de uso geral, de 32 bits cada (1 word  $\Leftrightarrow$  32 bits)
- ISA baseado em **instruções de dimensão fixa** (32 bits)
- Memória organizada em bytes (memória *byte addressable*)
- Espaço de endereçamento de 32 bits ( $2^{32}$  endereços possíveis, i.e. máximo de 4 GB de memória)
- Barramento de dados externo de 32 bits
- Arquitetura ***load-store*** (*register-register operation* )

# Instruções aritméticas - SOMA

Formato da instrução Assembly do Mips:

**add a, b, c** # Soma **b** com **c** e armazena o resultado  
# em **a** ( $a = b + c$ )

Uma adição do tipo

$$z = a + b + c + d$$

Tem de ser decomposta em:

**add z, a, b** # Soma a com b, resultado em z  
**add z, z, c** # Soma z com c, resultado em z  
**add z, z, d** # Soma z com d, resultado em z

# Instruções aritméticas - SUBTRAÇÃO

Formato da instrução Assembly do Mips:

**sub a, b, c** # Subtrai **c** a **b** e armazena o resultado  
# em **a** ( $a = b - c$ )

**Exemplo:** A operação  $z = (a + b) - (c + d)$

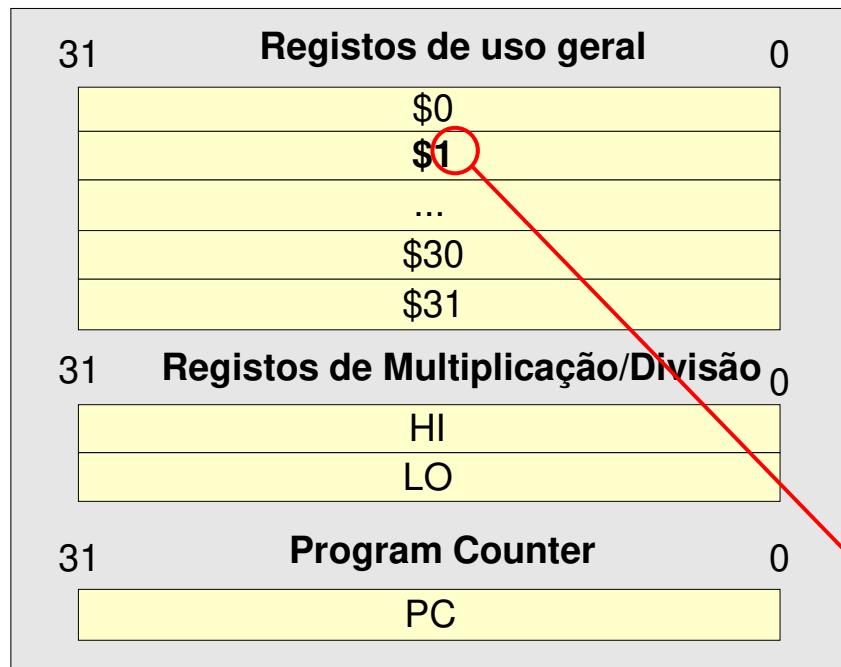
tem de ser decomposta em:

**add t1, a, b** # Soma **a** com **b**, resultado em **t1**

**add t2, c, d** # Soma **c** com **d**, resultado em **t2**

**sub z, t1, t2** # Subtrai **t2** a **t1**, resultado em **z**

# Os registos internos do MIPS



Em *assembly* são, normalmente, usados nomes alternativos para os registos (nomes virtuais):

- \$zero (\$0)
- \$at (\$1)
- \$v0 e \$v1 (\$2 e \$3)
- \$a0 a \$a3
- \$t0 a \$t9
- \$s0 a \$s7
- \$sp (\$29)
- \$ra (\$31)

Porquê 32 registos ?

Endereço do registo  
(0 a 31)

# Exemplo de tradução de C para Assembly MIPS

- Programa em C:

```
int a, b, c, d, z;  
z = (a + b) - (c + d);
```

- Em assembly (a, b, c, d, z residem em  
**a > \$17, b > \$18, c > \$19, d > \$20 e z > \$16**):

```
add $8, $17, $18 # Soma $17 com $18 e armazena o  
# resultado em $8  
add $9, $19, $20 # Soma $19 com $20 e armazena o  
# resultado em $9  
sub $16, $8, $9 # Subtrai $9 a $8 e armazena o  
# resultado em $16
```

# Como comentar um programa *Assembly*

```
# a > $17, b > $18
```

```
# c > $19, d > $20, z > $16
```

```
...
```

```
add    $8, $17, $18  # r1 = a + b;
```

```
add    $9, $19, $20  # r2 = c + d;
```

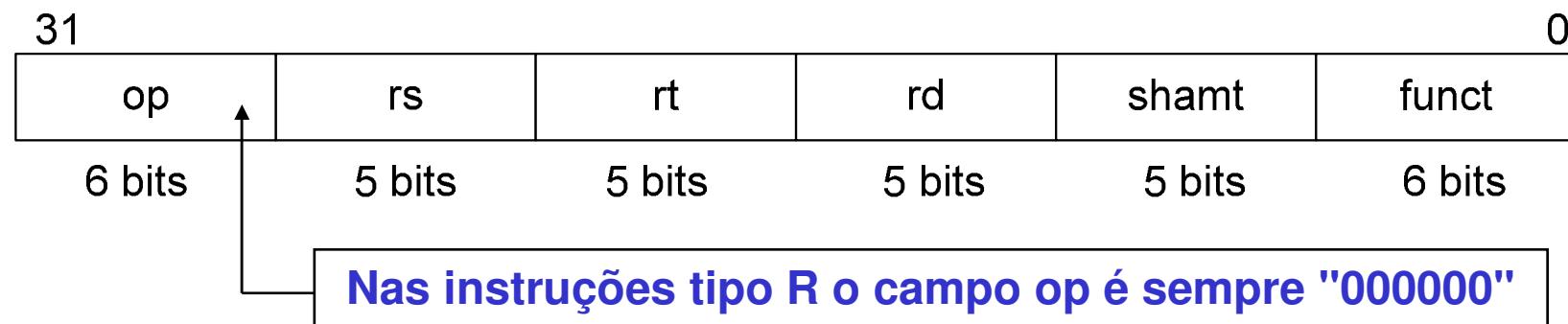
```
sub    $16, $8, $9   # z = (a + b) - (c + d);
```

```
...
```

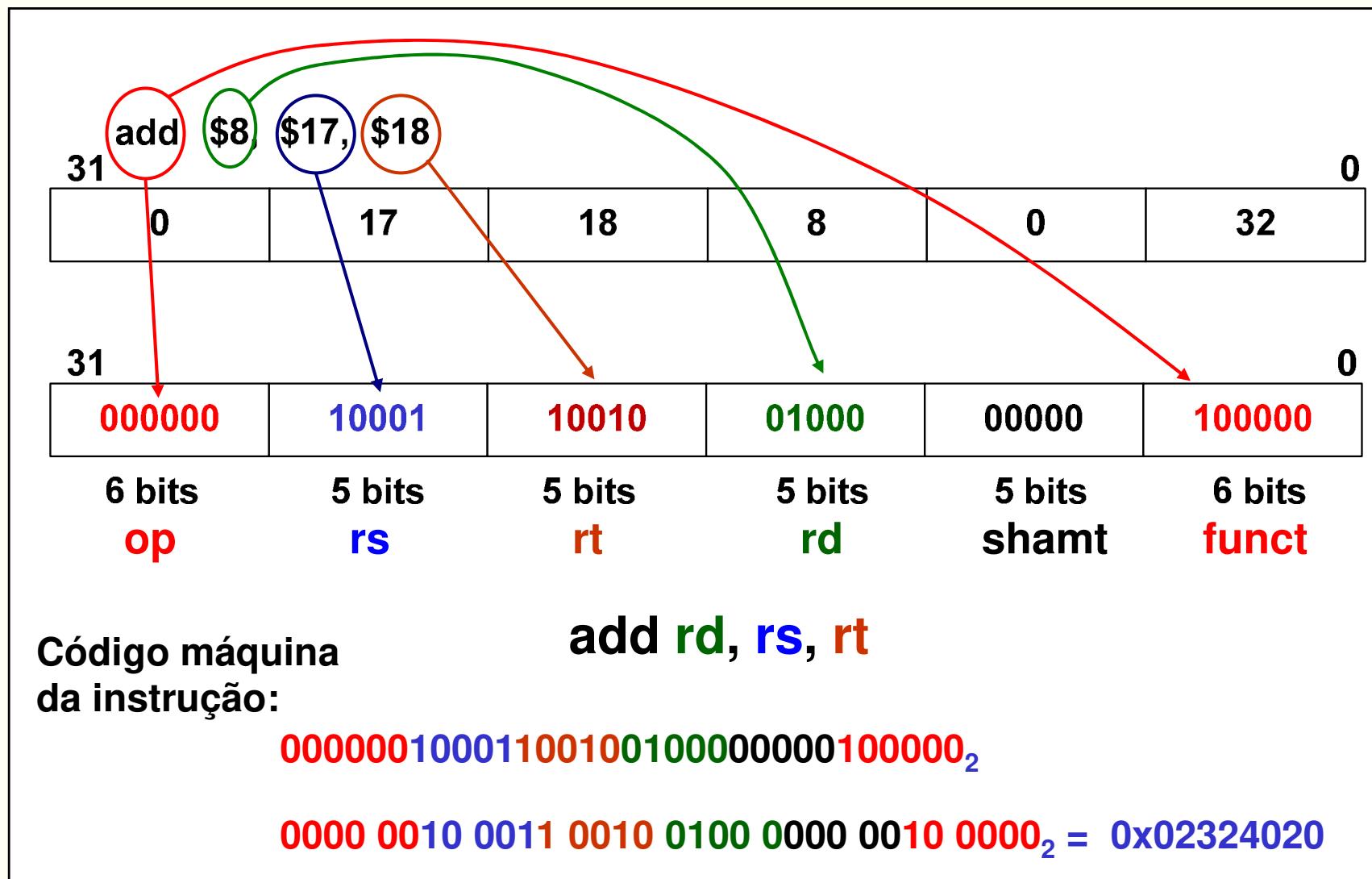
- A linguagem C é uma excelente forma de comentar programas em *Assembly* uma vez que permite uma interpretação direta e mais simples do(s) algoritmo(s) implementado(s).

# Codificação de instruções no MIPS – formato R

- O formato R é um dos três formatos de codificação de instruções no MIPS
- Campos da instrução:
  - op:** *opcode* (é sempre zero nas instruções tipo R)
  - rs:** Endereço do registo que contém o 1º operando fonte
  - rt:** Endereço do registo que contém o 2º operando fonte
  - rd:** Endereço do registo onde o resultado vai ser armazenado
  - shamt:** *shift amount* (útil apenas em instruções de deslocamento)
  - funct:** código da operação a realizar



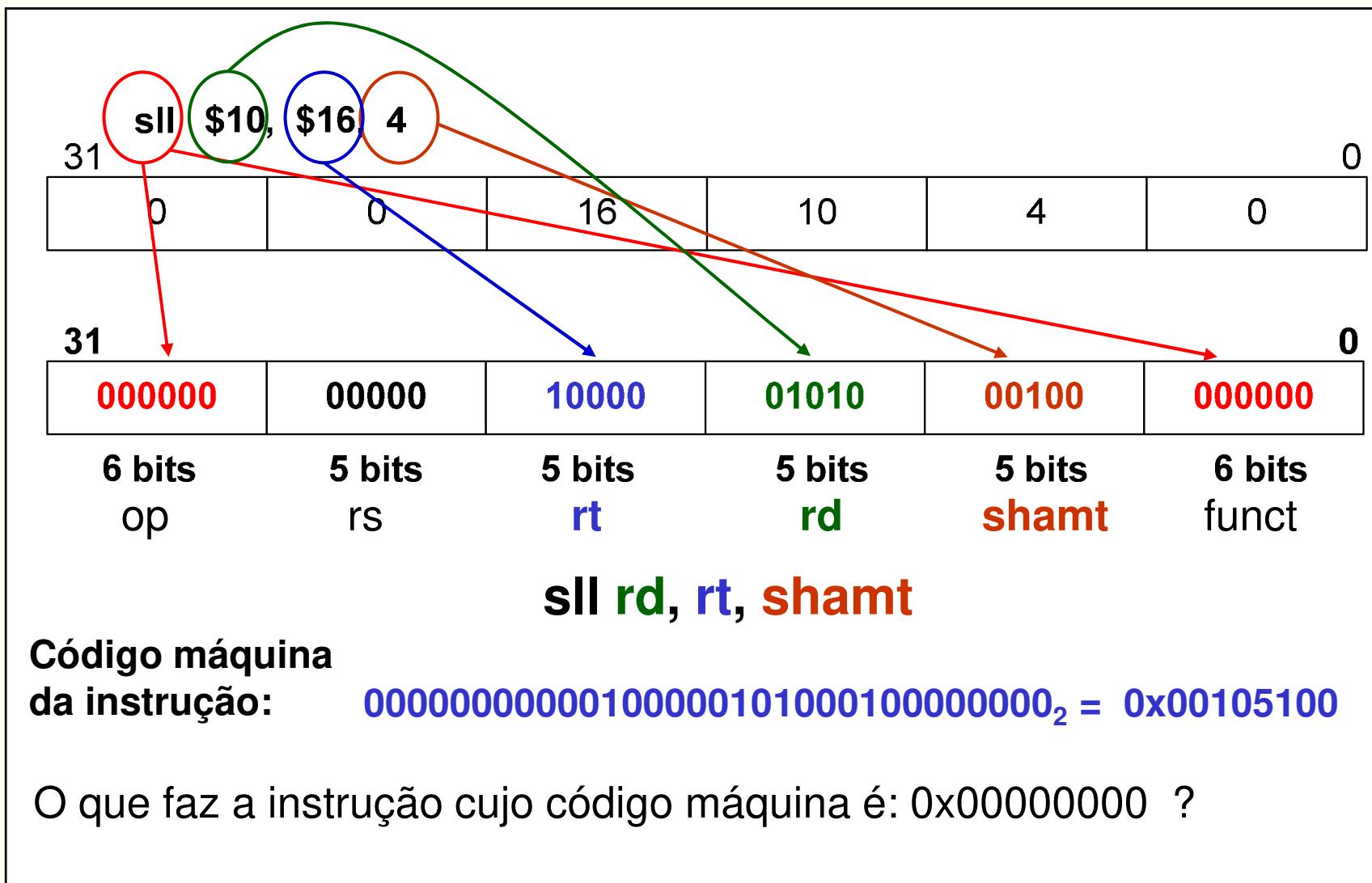
# Codificação de instruções no MIPS – formato R



# Instruções lógicas e de deslocamento

- Operadores lógicos bitwise em C:
  - `&` (AND), `|` (OR), `^` (XOR), `~` (NOT)
- Instruções lógicas do MIPS
  - `and Rdst, Rsrc1, Rsrc2` #  $Rdst = Rsrc1 \& Rsrc2$
  - `or Rdst, Rsrc1, Rsrc2` #  $Rdst = Rsrc1 | Rsrc2$
  - `nor Rdst, Rsrc1, Rsrc2` #  $Rdst = \sim(Rsrc1 | Rsrc2)$
  - `xor Rdst, Rsrc1, Rsrc2` #  $Rdst = (Rsrc1 ^ Rsrc2)$
- Operadores de deslocamento em C:
  - `<<` shift left
  - `>>` shift right, **lógico** ou **aritmético**, dependendo da variável ser do tipo **unsigned** ou **signed**, respetivamente
- Instruções de deslocamento do MIPS
  - `sll Rdst, Rsrc, k` #  $Rdst = Rsrc << k$ ; (shift left logical)
  - `srl Rdst, Rsrc, k` #  $Rdst = Rsrc >> k$ ; (shift right logical)
  - `sra Rdst, Rsrc, k` #  $Rdst = Rsrc >> k$ ; (shift right arithmetic)

# Codificação de instruções no MIPS – formato R



## Instruções de transferência entre registos internos

- Transferência entre registos internos:  $Rdst = Rsrc$
- Registo **\$0** do MIPS tem sempre o valor **0x00000000** (apenas pode ser lido)
- Utilizando o registo **\$0** e a instrução lógica OR é possível realizar uma operação de transferência entre registos internos:
  - **or Rdst, \$0, Rsrc** #  $Rdst = (0 \mid Rsrc) = Rsrc$
  - Exemplo: **or \$t1, \$0, \$t2** #  $$t1 = $t2$
- Para esta operação é habitualmente usada uma **instrução virtual** que melhora a legibilidade dos programas - "**move**".
- No processo de geração do código máquina, o *assembler* substitui essa instrução pela instrução nativa anterior:
  - **move Rdst, Rsrc** #  $Rdst = Rsrc$
  - Exemplo: **move \$t1, \$t2** #  $$t1 = $t2$  (or \$t1, \$0, \$t2)

## Aula 3

- Instruções de controlo de fluxo de execução
- Estruturas de controlo de fluxo de execução:
  - if()...then...else
  - Ciclos “for()”, “while()” e “do...while()”
- Tradução das estruturas de controlo de fluxo de execução para *Assembly* do MIPS

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Instruções de controlo de fluxo de execução

- "O que distingue um computador de uma calculadora básica (não programável) é a capacidade de decidir com base em valores que não são conhecidos *a priori*."
- A capacidade de decidir e realizar uma de várias tarefas com base num critério de verdade ou falsidade determinado durante a execução - conhecido como instrução if() nas linguagens de alto nível - é possibilitado no *assembly* do MIPS pelas instruções:

**beq Rsrc1, Rsrc2, Label** # branch if equal  
**bne Rsrc1, Rsrc2, Label** # branch if not equal

e são conhecidas como “**branches**” (saltos / jumps)  
**condicionais**



## Instruções de controlo de fluxo de execução – BEQ

**beq Rsrc1, Rsrc2, Label # branch if equal**

- Se os conteúdos dos registos **Rsrc1** e **Rsrc2** forem iguais é realizado um salto, i.e., a execução continua na **instrução situada no endereço representado por "Label"** (**branch taken**)
- A execução continua na **instrução seguinte** se os conteúdos dos 2 registos forem diferentes (**branch not taken**)
- O endereço para onde o salto é efetuado (no caso de a condição ser verdadeira) designa-se por **endereço-alvo** da instrução de *branch* (**branch target address**)



## Instruções de controlo de fluxo de execução – BNE

**bne Rsrc1, Rsrc2, Label # branch if not equal**

- Se os conteúdos dos registos **Rsrc1** e **Rsrc2** forem diferentes é realizado um salto, i.e., a execução continua na **instrução situada no endereço representado por "Label"** (**branch taken**)
- A execução continua na instrução seguinte se os conteúdos dos 2 registos forem iguais (**branch not taken**)
- Exemplo:

|                       |   |
|-----------------------|---|
| bne \$1, \$2, L1      | # se <i>branch taken</i> (i.e. \$1 != \$2)                  |
| add \$3, \$5, \$5     | # a próxima instrução está em L1                            |
| L1: add \$2, \$3, \$4 | # caso contrário a instrução<br># seguinte é executada<br># |



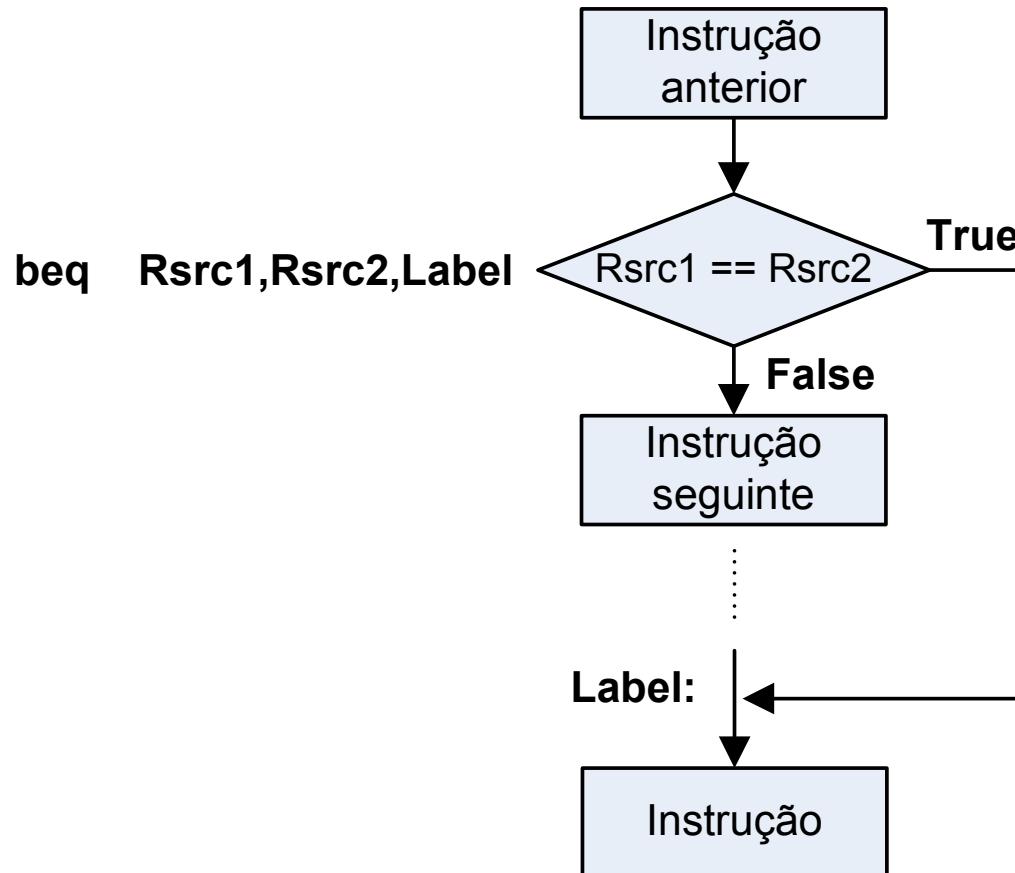
# Instruções de *branch* – como funcionam?

- Se a **condição** testada na instrução **for verdadeira** (no caso do "beq" **Rsrc1=Rsrc2**, isto é **Rsrc1 – Rsrc2 = 0**), o valor corrente do PC (**Program Counter**) é substituído pelo endereço a que corresponde "Label" (endereço-alvo)
  - A instrução que é executada de seguida é a que se situa no endereço-alvo
- Se a **condição for falsa**, a sequência de execução não é alterada
  - Neste caso, a instrução que é executada de seguida é a que se situa imediatamente a seguir à instrução de *branch*



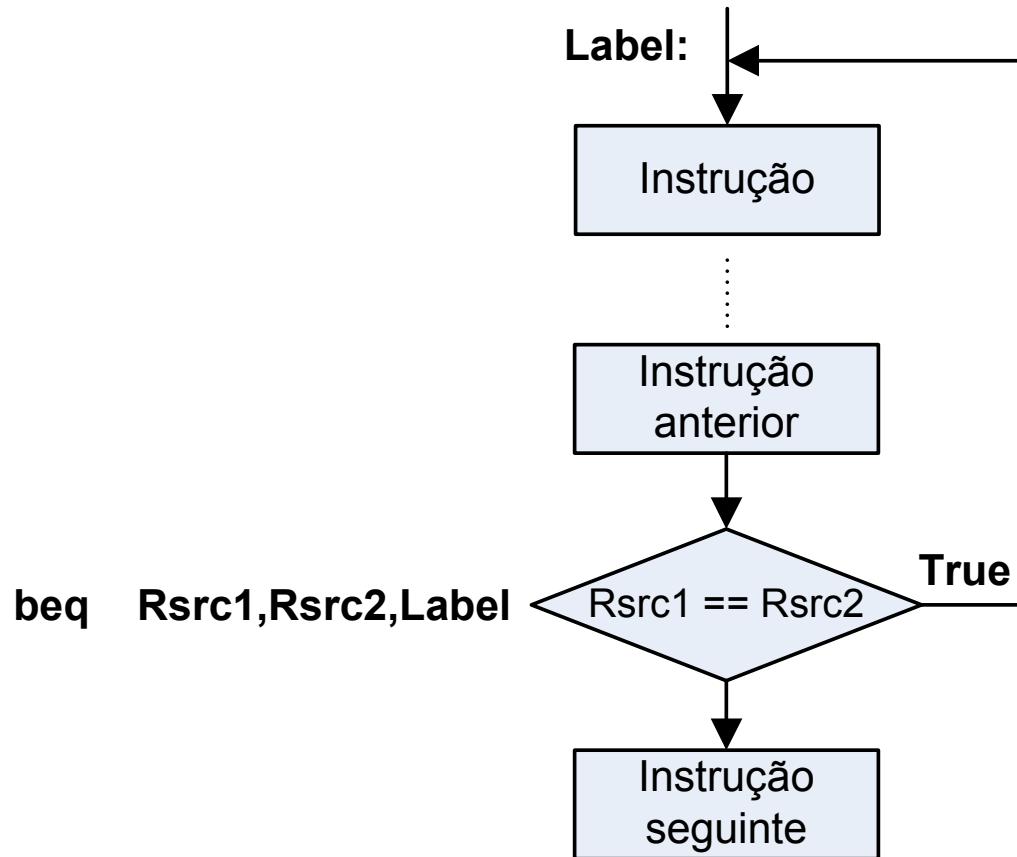
# Instruções de *branch* – como funcionam?

- **beq Rsrc1, Rsrc2, Label** # branch if equal



# Instruções de *branch* – como funcionam?

- **beq Rsrc1, Rsrc2, Label** # branch if equal



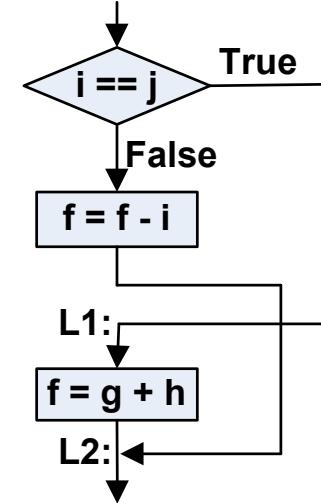
# Exemplo de tradução C / Assembly

Considere-se o seguinte trecho de código C  
(escrito de forma inadequada):

```
if (i == j) goto L1; // ☺
f = f - i;
goto L2; // ☺
L1: f = g + h;
L2: ...
```

O código equivalente em *Assembly* do MIPS seria  
(considerando  $i > \$19$ ,  $j > \$20$ ,  $f > \$16$ ,  $g > \$17$ , e  $h > \$18$ ):

```
beq $19,$20,L1 # if(i == j) goto L1;
sub $16,$16,$19 # f = f - i;
j L2             # goto L2;
L1: add $16,$17,$18 # f = g + h;
L2: ...
```



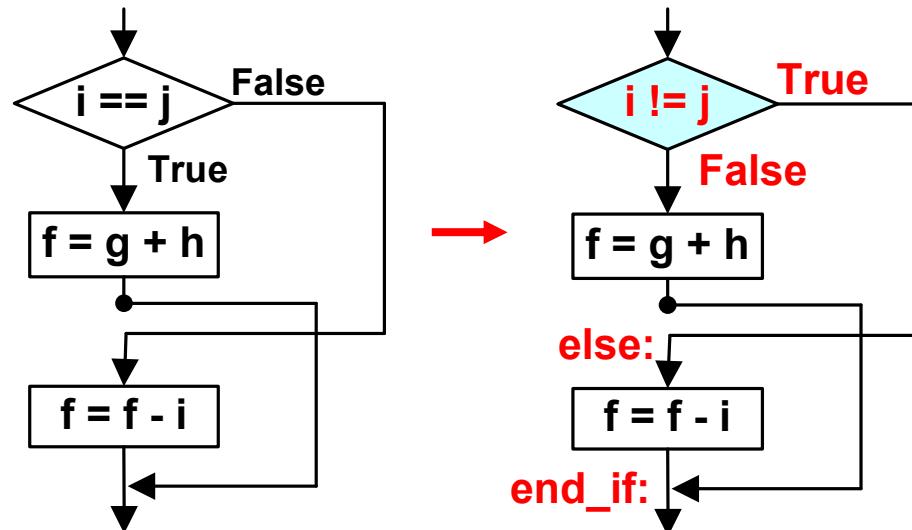
*j* significa *jump* e representa um  
salto incondicional para o "label" indicado



# Exemplo de tradução C / Assembly

O exemplo anterior escrito de forma correta:

```
if (i == j)
{
    f = g + h;
}
else
{
    f = f - i;
}
```



A que corresponde o código *Assembly*:

```
bne    $19,$20,ELSE      # if (i == j) {
add    $16,$17,$18        #     f = g + h;
j      END_IF             #
ELSE:
    sub    $16,$16,$19      # } else {
END_IF:                      # }
```



# Outras instruções de *branch* do MIPS

- O ISA do MIPS suporta ainda um conjunto de instruções que **comparam diretamente com zero**:
  - **bltz Rsrc, Label** # Branch if Rsrc < 0
  - **blez Rsrc, Label** # Branch if Rsrc ≤ 0
  - **bgtz Rsrc, Label** # Branch if Rsrc > 0
  - **bgez Rsrc, Label** # Branch if Rsrc ≥ 0
- Nestas instruções **o registo \$0 está implícito** como o segundo registo a comparar
- Exemplos:
  - **blez \$1, L2** # if \$1 <= 0 then goto L2
  - **bgtz \$2, L3** # if \$2 > 0 then goto L3



# Outras instruções de *branch* do MIPS

- Podem ainda ser utilizadas, nos programas Assembly, instruções de salto não diretamente suportadas pelo MIPS (**instruções virtuais**), mas que são **decompostas pelo assembler em instruções nativas**, nomeadamente:

|              |                            |                                     |
|--------------|----------------------------|-------------------------------------|
| ▪ <b>blt</b> | <b>Rsrc1, Rsrc2, Label</b> | <b># Branch if Rsrc1 &lt; Rsrc2</b> |
| ▪ <b>ble</b> | <b>Rsrc1, Rsrc2, Label</b> | <b># Branch if Rsrc ≤ Rsrc2</b>     |
| ▪ <b>bgt</b> | <b>Rsrc1, Rsrc2, Label</b> | <b># Branch if Rsrc &gt; Rscr2</b>  |
| ▪ <b>bge</b> | <b>Rsrc1, Rscr2, Label</b> | <b># Branch if Rsrc ≥ Rscr2</b>     |

- Nestas instruções **Rsrc2** pode ser substituído por uma **constante**.

- Exemplos:

- **blt \$1,\$2,L2 # if \$1 < \$2 goto L2**
  - **bgt \$1,100,L3 # if \$1 > 100 goto L3**

Como são compostas estas instruções?



# Instrução SLT

Para além das instruções de salto com base no critério de igualdade e desigualdade, o MIPS suporta ainda a instrução:

```
slt Rdst, Rsrc1, Rsrc2    # slt ≡ "set if less than"  
                                # set Rdst if Rsrc1 < Rsrc2
```

**Descrição:** O registo "Rdst" toma o valor "1" se o conteúdo do registo "Rsrc1" for inferior ao do registo "Rsrc2". Caso contrário toma o valor "0".

```
slti Rdst, Rsrc1, Imm    # slt ≡ "set if less than"  
                                # set Rdst if Rsrc1 < Imm
```

A utilização das instruções "**bne**", "**beq**", "**slti**" e "**slt**", em conjunto com o registo **\$0**, permitem a implementação de todas as condições de comparação entre dois registos e também entre um registo e uma constante: (**A = B**), (**A ≠ B**), (**A > B**), (**A ≥ B**), (**A < B**), (**A ≤ B**)



# Decomposição das instruções virtuais BGT e BGE

A **instrução virtual "bge"** (*branch if greater or equal than*):

```
bge    $4, $7, exit  # if $4 ≥ $7 goto exit  
          # (i.e. goto exit if !( $4 < $7 ))
```

É decomposta nas **instruções nativas**:

```
slt    $1, $4, $7  # $1 = 1 if $4 < $7 ($1=0 if $4 ≥ $7)  
beq    $1, $0, exit # if $1 = 0 goto exit
```

De modo similar, a **instrução virtual "bgt"** (*branch if greater than*):

```
bgt    $4, $7, exit  # if $4 > $7 goto exit  
          # (i.e. goto exit if $7 < $4)
```

É decomposta nas **instruções nativas**:

```
slt    $1, $7, $4  # $1 = 1 if $7 < $4 ($1=1 if $4 > $7)  
bne    $1, $0, exit # if $1 ≠ 0 goto exit
```



# Estruturas de controlo de fluxo em C

- Exemplos

```
if (a >= n) {  
    b = c;  
} else {  
    b = d;  
} ...
```

```
for (n = 0; n < 100; n++) {  
    a = a + b[n];  
}  
...
```

```
n = 0;  
do {  
    a = a + b[n];  
    n++;  
} while (n < 100);  
...
```

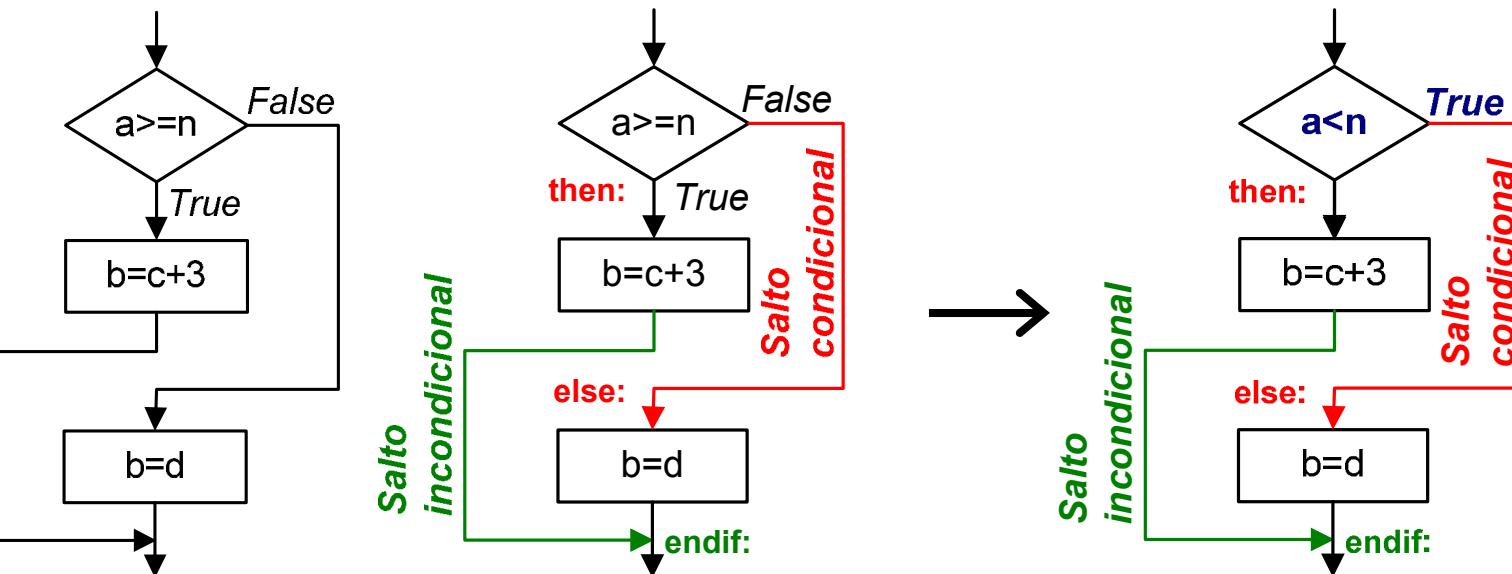
```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
}  
...
```



# Tradução para Assembly do MIPS (if()... then... else)

```
if (a >= n) {  
    b = c + 3;  
} else {  
    b = d;  
}
```

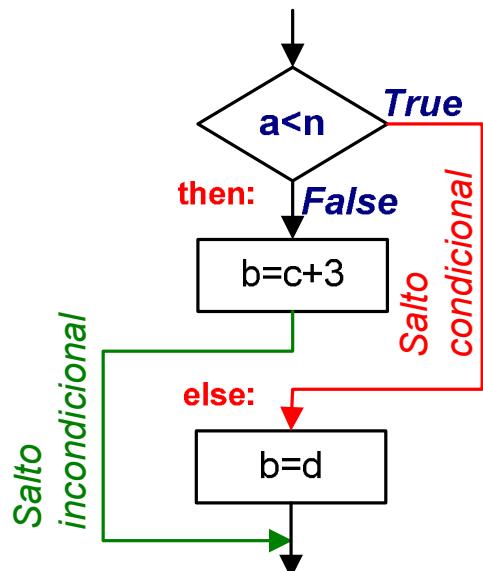
- Transformando o código apresentado no fluxograma equivalente, é possível identificar facilmente a ocorrência de um **salto condicional** e de um **salto incondicional**
- E adaptar o salto condicional para que este se efetue quando a condição for verdadeira (tal como nos *branches*).



## Tradução para Assembly do MIPS – *if()... then... else*

```
if (a >= n) {  
    b = c + 3;  
} else {  
    b = d;  
}
```

```
a > $t0  
n > $t1  
c > $t2  
b > $t3  
d > $t4
```



Supondo que as variáveis residem nos registos \$t0 a \$t4, a tradução para Assembly fica:

```
blt $t0, $t1, else # if (a >= n) {  
addi $t3, $t2, 3    #     b = c + 3;  
j endif            # }  
else:  
move $t3, $t4        #     b = d;  
endif: ...           # }
```



## Tradução para Assembly do MIPS - ciclos **for()** e **while()**

```
for (n = 0; n < 100; n++) {  
    a = a + b[n];  
}  
...
```

```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
}
```

Estes dois exemplos são  
**funcionalmente equivalentes!**

Operações a executar **antes do corpo do ciclo** (inicializações)

Condição de continuação da execução do ciclo

Operações a realizar no **fim do corpo do ciclo**

Os 3 campos do ciclo "**for**" são opcionais. Exemplo:

```
for( ; ; i++) {  
    ...  
}
```

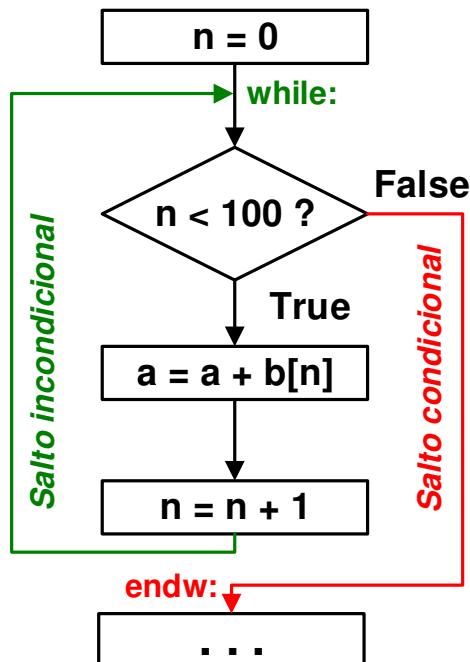
Qual o resultado deste código?



## Tradução para Assembly do MIPS - ciclos **for()** e **while()**

```
for (n = 0; n < 100; n++) {  
    a = a + b[n];  
}  
...
```

```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
} ...
```



- É possível identificar a ocorrência de um **salto condicional** e de um **salto incondicional**
- O salto condicional necessita de ser modificado de forma a ser efetuado quando a condição for verdadeira
- Para isso usa-se o **complemento lógico** da condição presente no código original (para o exemplo, "`<`" passa a ser "`>=`")

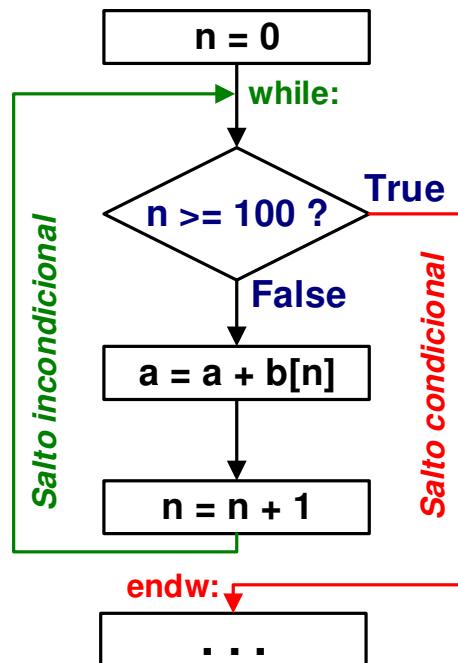


## Tradução para Assembly do MIPS - ciclos **for()** e **while()**

```
for (n = 0; n < 100; n++) {  
    a = a + b[n];  
}  
...
```



```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
} ...
```



A tradução da estrutura de controlo, com "n" a residir em \$t1, fica:

```
ori $t1, $0, 0      #n=0  
while: bge $t1, 100, endw #while(n<100)  
# {  
#     ...  
# }  
addi $t1, $t1, 1    #     n++;  
j     while        # }  
endw:
```



## Tradução para Assembly do MIPS - ciclos **for()** e **while()**

Complemento lógico ("<" passa a ">=")

```
for (n = 0; n < 100; n++) {  
    a = a + b[n];  
}  
...
```

Teste condicional feito no início do ciclo

```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
} ...
```

A tradução para Assembly MIPS (assumindo n > \$t1, a > \$t2, b > \$t3) fica:

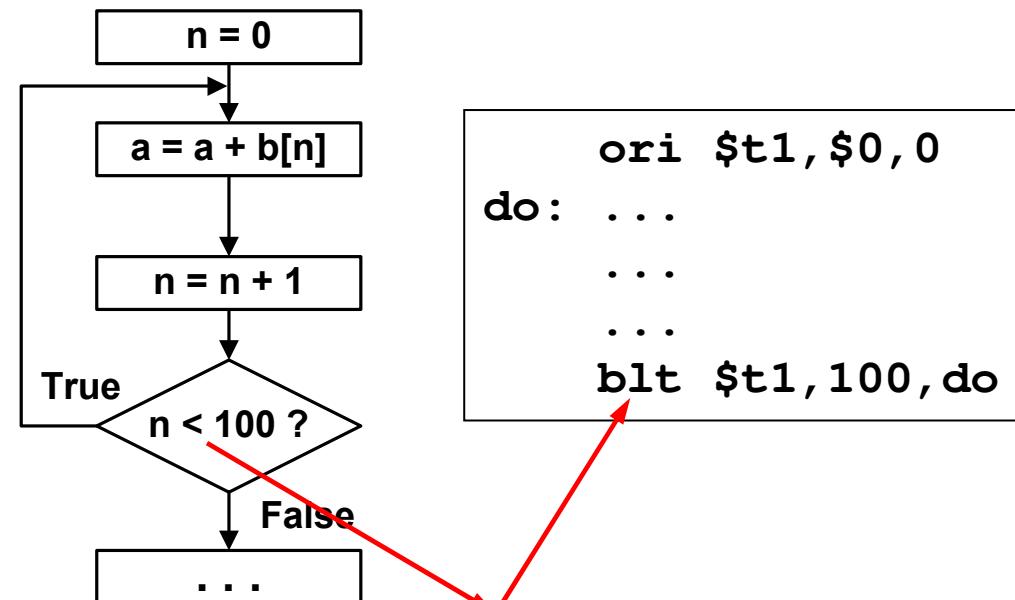
```
ori  $t1,$0,0      # n = 0;  
while: bge $t1,100,endw  # while(n < 100) {  
    sll  $t0,$t1,2      # temp = 4 * n;  
    add  $t0,$t3,$t0      # temp = temp + b;  
    lw   $t0,0($t0)      # temp = *temp;  
    add  $t2,$t2,$t0      # a = a + temp;  
    addi $t1,$t1,1      # n++;  
    j    while           # }  
  
endw: ...
```



## Tradução para Assembly do MIPS - ciclo **do ... while()**

- Ao contrário do **for()** e do **while()**, o corpo do ciclo **do...while()** é executado incondicionalmente pelo menos uma vez!
- O teste da condição é efetuado no fim do ciclo

```
n = 0;  
do  
{  
    a = a + b[n];  
    n++;  
}while (n < 100);  
...
```



Neste caso o branch é executado quando a condição é verdadeira. Logo não se aplica o complemento lógico à condição

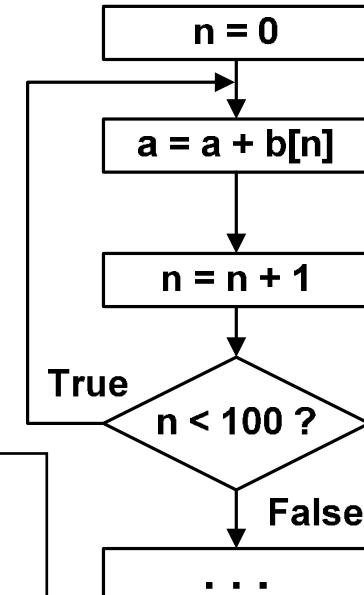


## Tradução para Assembly do MIPS - ciclo *do ... while()*

```
n = 0;  
do  
{  
    a = a + b[n];  
    n++;  
}while (n < 100);  
...
```

A tradução completa para Assembly do MIPS fica (com  $n > \$t1$ ,  $a > \$t2$ ,  $b > \$t3$ ):

```
ori $t1,$0,0      # n = 0;  
do:  
    sll $t0,$t1,2    # temp = 4 * n;  
    add $t0,$t3,$t0  # temp = temp + b;  
    lw $t0,0($t0)    # temp = *temp;  
    add $t2,$t2,$t0  # a = a + temp;  
    addi $t1,$t1,1   # n = n + 1;  
    blt $t1,100,do   # } while(n < 100);
```



o teste condicional é efetuado no fim do ciclo



# Conclusão

- As estruturas do tipo ciclo incluem, geralmente, uma ou mais instruções de inicialização de variáveis, executadas antes e fora do mesmo
- No caso do **for()** e do **while()** o teste condicional é executado no início do ciclo
- No caso do **do...while()** o teste condicional é efetuado no fim do ciclo
- Na tradução de um **for()** para *Assembly*, o terceiro campo é codificado no fim do corpo do ciclo.



## Aula 4

- Armazenamento de informação na memória externa
- Endereçamento indireto por registo com deslocamento
- Instruções de acesso a informação residente na memória externa: LW, SW, LB, LBU, SB
- Codificação das instruções de acesso à memória: formato I
- Restrições de alinhamento nos endereços das variáveis
- Organização de informação em memória: "little-endian" *versus* "big-endian"

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



## Armazenamento de informação – registros internos

- Exemplo de um trecho de código que apenas faz uso de registros internos do CPU para o armazenamento de informação:

```
add $8, $17, $18      # Soma $17 com $18 e armazena o resultado em $8
add $9, $19, $20      # Soma $19 com $20 e armazena o resultado em $9
sub $16, $8, $9       # Subtrai $9 a $8 e armazena o resultado em $16
```

- Sendo o equivalente em C:

```
// a, b, c, d e z residem, respectivamente, em:
// $17, $18, $19, $20 e $16
// $8 e $9 representam variáveis temporárias não explicitadas em C
```

```
int a, b, c , d , z;
z = (a + b) - (c + d);
```



## Armazenamento de informação – memória externa

- E se se pretendesse somar os elementos de um *array* composto por N elementos?
- Se N for maior do que o número de registos disponíveis no CPU seria necessário recorrer a recursos externos – a memória
- Por outro lado, a arquitetura do MIPS é do tipo **load-store**, pelo que não é possível operar diretamente sobre o conteúdo da memória externa
- Deverão existir, portanto, instruções para transferir informação entre os registos do CPU e a memória externa



# Modos de endereçamento

- O **método** usado pela arquitetura para **aceder ao elemento que contém a informação** que irá ser processada por uma dada instrução é genericamente designado por “**Modo de Endereçamento**”
- Nas instruções aritméticas e lógicas, os operandos residem em registos internos (um dos operandos também pode ser uma constante)
- Os endereços dos registos internos envolvidos na operação são especificados diretamente na própria instrução, em campos de 5 bits: ***rs*** e ***rt***
- Este modo é designado por **endereçamento tipo registo**



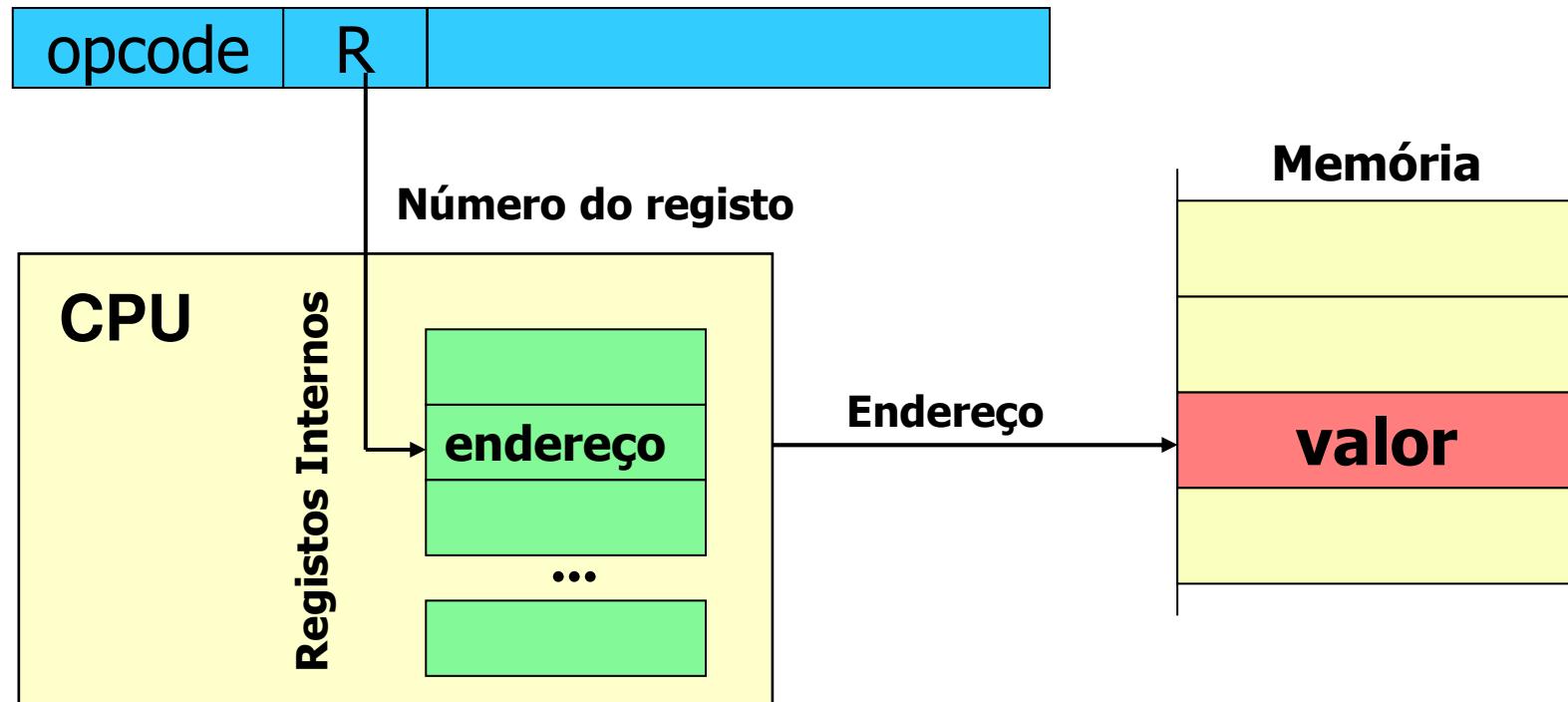
## Acesso a informação residente na memória externa

- Como será então possível codificar as instruções de acesso à memória externa (para escrita e leitura), sabendo que as instruções do MIPS ocupam, todas, exatamente 32 bits?
- Note-se que um endereço de memória no MIPS é representado por 32 bits, pelo que ele sozinho ocuparia a totalidade do código máquina da instrução
- **Solução:** em vez do endereço, a instrução indica um registo que contém o endereço de memória a aceder (recordar-se que um registo interno permite armazenar 32 bits):
  - **endereçamento indireto por registo**



# Endereçamento indireto por registo

Código máquina da instrução:



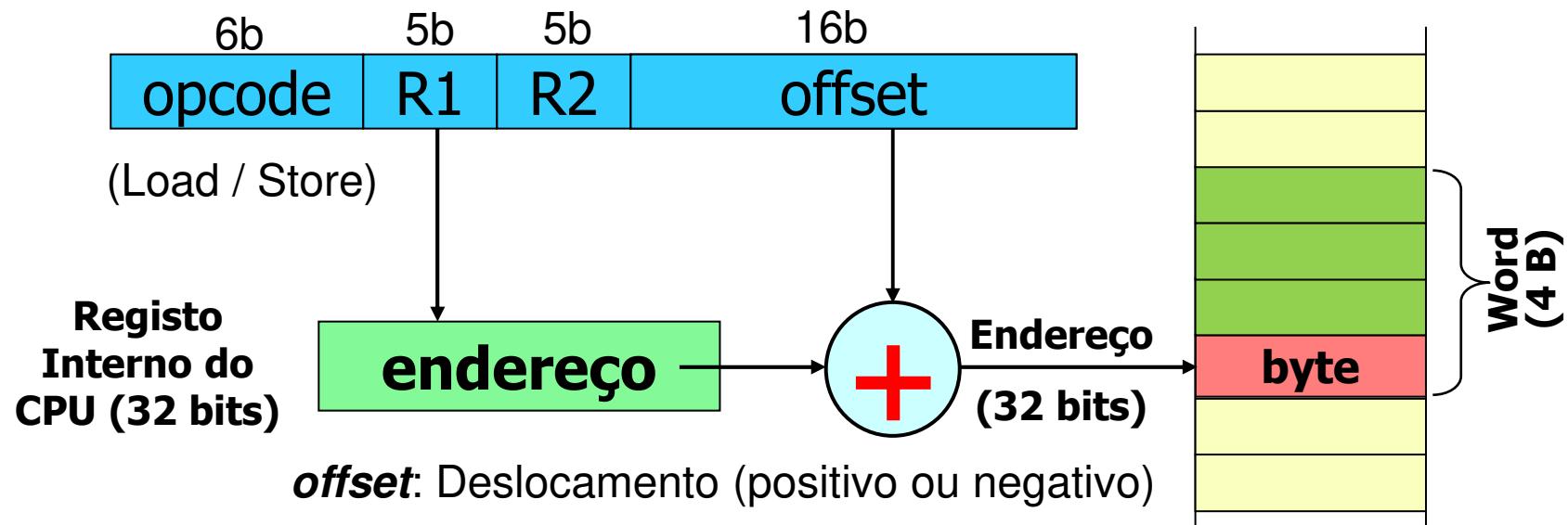
A instrução indica qual o registo interno do CPU que contém o endereço de memória a aceder



# Endereçamento indireto por registo com deslocamento

- A solução do MIPS

Código máquina da instrução:



**offset**: Deslocamento (positivo ou negativo)

**R1**: Registo de endereçamento

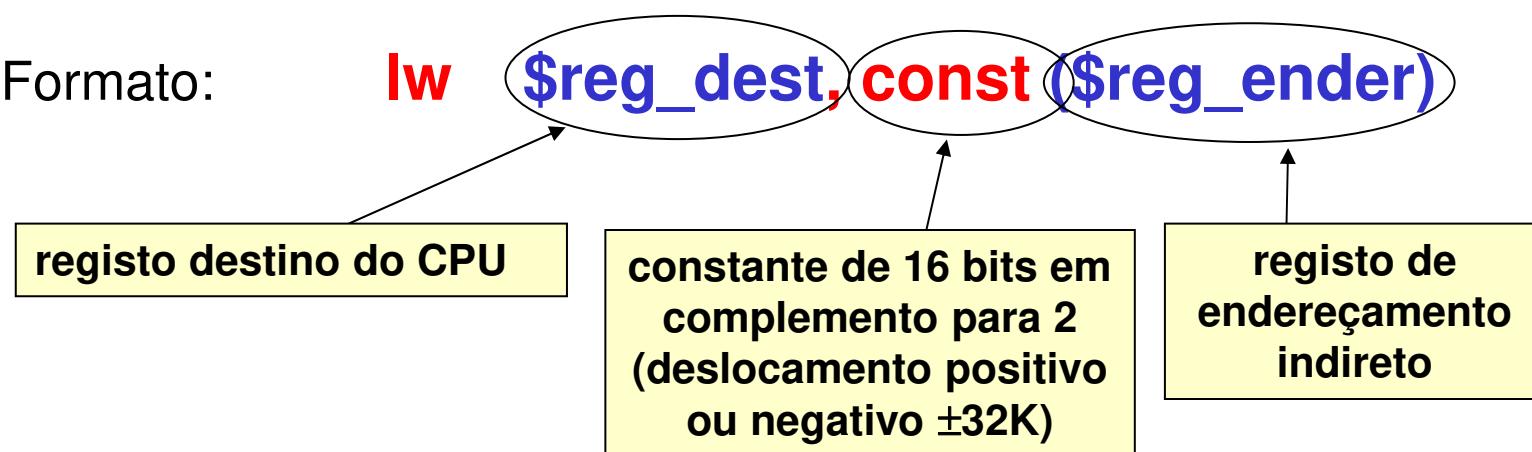
**R2**: Registo de dados: destino / origem

O endereço de acesso à memória é calculado pela **soma algébrica do conteúdo do registo com o offset** (extended, com sinal, para a dimensão do registo, i.e., para 32 bits)

# Leitura da memória – instrução LW

- **LW** - (*load word*) transfere uma palavra de 32 bits da memória para um registo interno do CPU (**1 word é armazenada em 4 posições de memória consecutivas**)

Formato:



Exemplo:

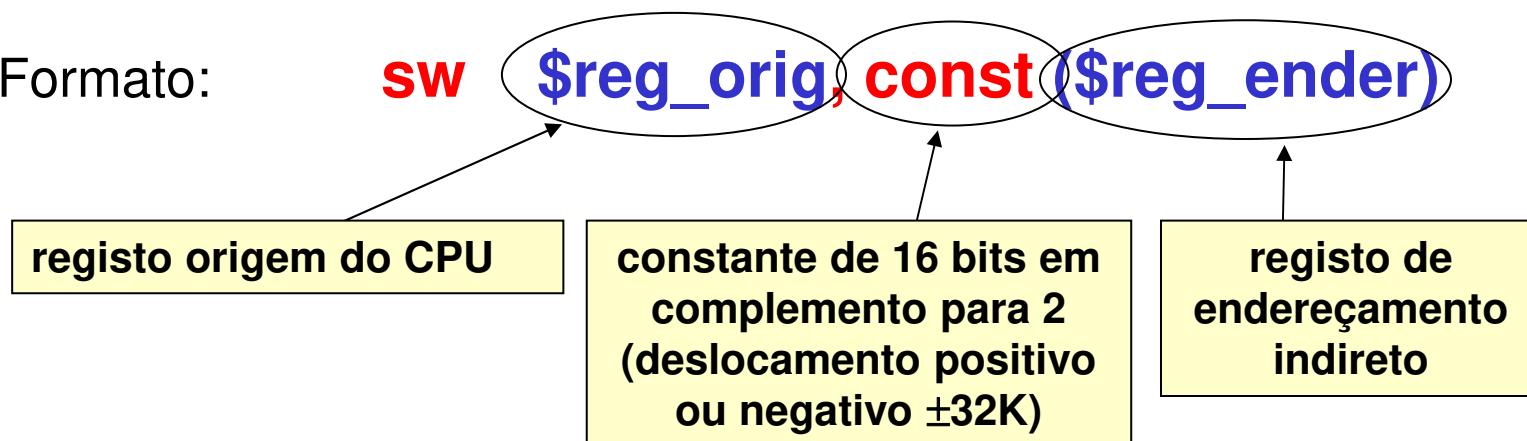
**lw \$5, 4 (\$2)**    **# copia para o registo \$5 a word armazenada**  
                     **# a partir do endereço de memória calculado como:**  
                     **#      addr = (conteúdo do registo \$2) + 4**



# Escrita na memória – instrução SW

- **SW** - (*store word*) transfere uma palavra de 32 bits de um registo interno do CPU para a memória (1 word é armazenada em 4 posições de memória consecutivas)

Formato:



Exemplo:

```
sw $7, -8 ($4)      # copia a word armazenada no registo $7 para a
                      # memória, a partir do endereço calculado como:
                      #     addr = (conteúdo do registo $4) - 8
```



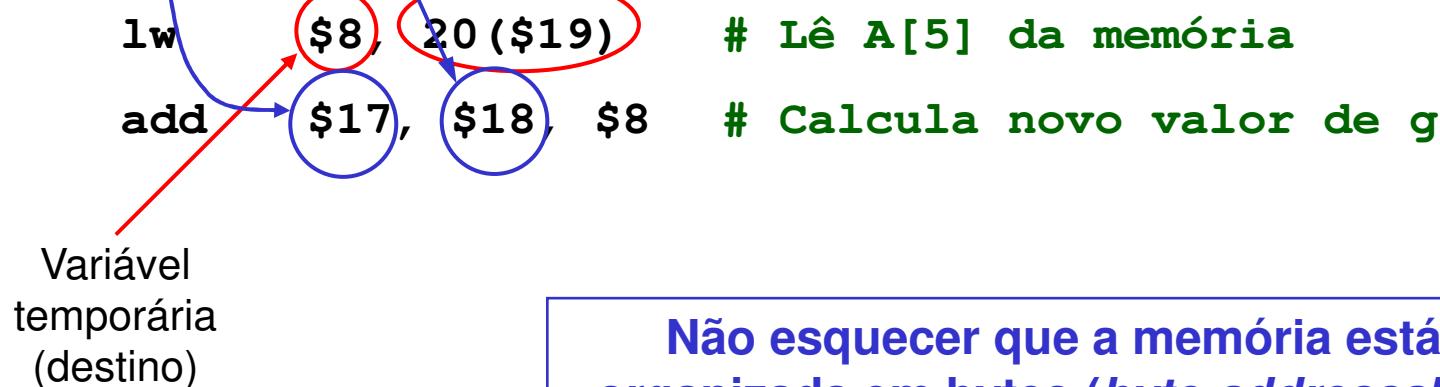
# Acesso à memória: exemplo 1

- Considere-se o seguinte exemplo:

$$g = h + A[5]$$

assumindo que ***g***, ***h*** e o **endereço de início do array *A*** residem nos registos **\$17**, **\$18** e **\$19**, respetivamente

- Usando instruções do *Assembly* do MIPS, a expressão anterior tomaria a seguinte forma (supondo que *A* é um *array* de *words*, i.e. 32 bits):



**Não esquecer que a memória está organizada em bytes (*byte-addressable*)**

# Acesso à memória: exemplo 1

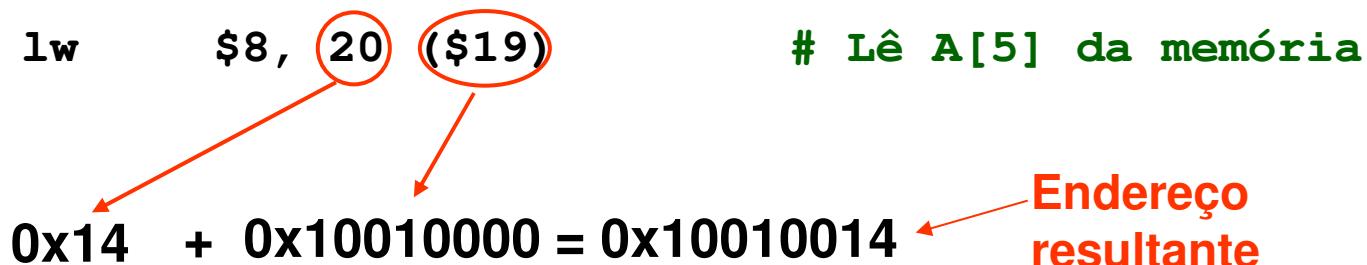
- Retomemos a primeira instrução:

lw \$8, 20(\$19) # Lê A[5] da memória

- O endereço da memória é calculado somando o conteúdo do registo indicado entre parêntesis com a constante explicitada na instrução. Se o conteúdo de **\$19** for **0x10010000** o endereço da memória será:

$$\text{lw } \$8, 20(\$19) \quad \# \text{ Lê A[5] da memória}$$

$0x14 + 0x10010000 = 0x10010014$  ← Endereço resultante



- Como cada elemento do *array* ocupa quatro *bytes* (*array de words*), o elemento acedido será A[5]



## Acesso à memória: exemplo 2

- Se se pretendesse obter:

$$A[5] = h + A[5]$$

- Assumindo mais uma vez que *h* e o endereço inicial do array residem nos registos **\$18** e **\$19**, respetivamente
- Poderíamos fazê-lo com o seguinte código:

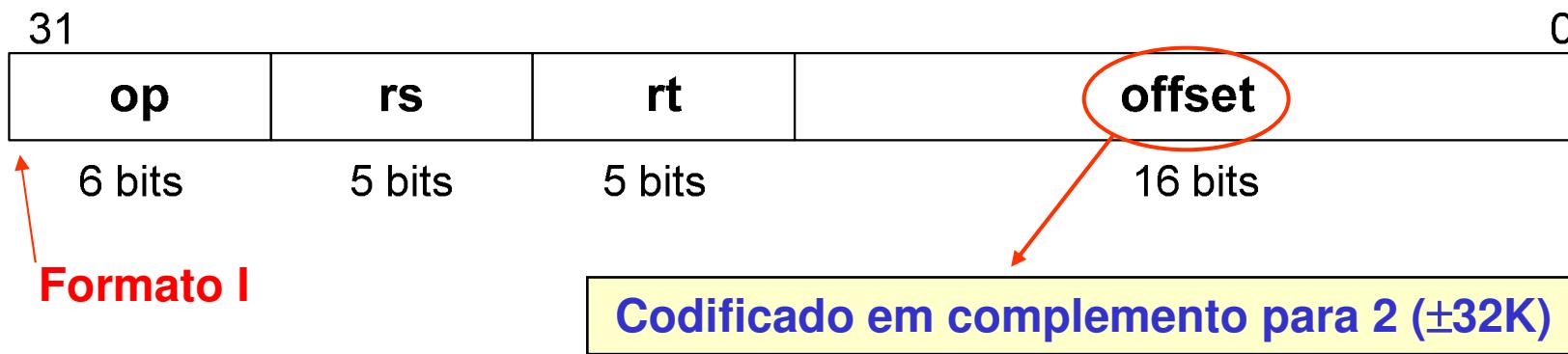
```
lw      $8, 20($19)    # Lê A[5] da memória  
add    $8, $18, $8     # Calcula novo valor  
sw      $8, 20($19)    # Escreve resultado em A[5]
```

**Arquitetura load/store:** as operações aritméticas e lógicas só podem ser efetuadas sobre registos internos do CPU



## Codificação das instruções de acesso à memória no MIPS

- A necessidade de codificação de uma constante de 16 bits, obriga à definição de um novo formato de codificação, o **formato I**



- Gama de representação da constante de 16 bits
  - [-32768, +32767]



# Codificação da instrução LW (Load Word)

lw \$8, 20(\$19) #Lê A[5] da memória

Corresponderia à seguinte instrução máquina:

|    |          |          |          |           |   |
|----|----------|----------|----------|-----------|---|
| 31 | (6 bits) | (5 bits) | (5 bits) | (16 bits) | 0 |
|    | 35       | 19       | 8        | 20        |   |
| OP | RS       | RT       | OFFSET   |           |   |

LW RT, OFFSET(RS)

|        |        |        |         |                     |
|--------|--------|--------|---------|---------------------|
| 31     |        |        |         | 0                   |
|        | 100011 | 10011  | 01000   | 0000 0000 0001 0100 |
| 6 bits | 5 bits | 5 bits | 16 bits |                     |

100011100110100000000000000010100<sub>2</sub>

1000 1110 0110 1000 0000 0000 0001 0100 = 0x8E680014



# Codificação da instrução SW (Store Word)

**sw**

\$8, 20(\$19)

#Escreve result. em A[5]

Corresponderia à seguinte instrução máquina:

| 31 | (6 bits) | (5 bits) | (5 bits) | (16 bits) | 0 |
|----|----------|----------|----------|-----------|---|
| OP | RS       | RT       | OFFSET   |           |   |
|    | 43       | 19       | 8        | 20        |   |

**SW RT, OFFSET(RS)**

| 31     |        |        |                     | 0 |
|--------|--------|--------|---------------------|---|
| 101011 | 10011  | 01000  | 0000 0000 0001 0100 |   |
| 6 bits | 5 bits | 5 bits | 16 bits             |   |

101011100110100000000000000010100<sub>2</sub>

1010 1110 0110 1000 0000 0000 0001 0100 = 0xAE680014



# Exemplo de codificação

- O seguinte trecho de código assembly:

```
lw $8, 20($19)          # Lê A[5] da memória  
add $8, $18, $8         # Calcula novo valor  
sw $8, 20($19)          # Escreve resultado em A[5]
```

Corresponde à codificação:

| 31 0 |      |      |        |      |      |
|------|------|------|--------|------|------|
| 0x23 | 0x13 | 0x08 | 0x0014 |      |      |
| 0x00 | 0x12 | 0x08 | 0x08   | 0x00 | 0x20 |
| 0x2B | 0x13 | 0x08 | 0x0014 |      |      |

Formato I      Formato R      Formato I

- Resultando no código máquina:

$100011100110100000000000000010100_2 = 0x8E680014$

$000000100100100010000000000000000000_2 = 0x02484020$

$101011100110100000000000000010100_2 = 0xAE680014$



## Restrições de alinhamento nos endereços das variáveis

- Externamente o barramento de endereços do MIPS só tem disponíveis 30 dos 32 bits:  $A_{31} \dots A_2$ . Ou seja, qualquer combinação nos bits  $A_1$  e  $A_0$  é ignorada no barramento de endereços exterior.
- Assim, do ponto de vista externo, só são gerados endereços **múltiplos de  $2^2 = 4$**  (ex: ...0000, ...0100, , ...1000, ...1100)

**O acesso a words só é possível em endereços múltiplos de 4**

- **Questão 1:** O que acontece quando o MIPS tenta executar uma instrução de leitura/escrita de uma ***word*** da memória, num endereço não múltiplo de 4 ?
- **Questão 2:** Como é possível a leitura/escrita de 1 byte de informação uma vez que o ISA do MIPS define que a memória é organizada em bytes (*byte-addressable*) ?



## Restrições de alinhamento nos endereços das variáveis

- Se, numa instrução de leitura/escrita de uma *word*, for especificado um endereço **não múltiplo de 4**, quando o MIPS a tenta executar verifica que o endereço é inválido e **gera uma exceção**, terminando aí a execução do programa
- Como se evita o problema ?
  - Garantindo que as variáveis do tipo *word* estão armazenadas num endereço múltiplo de 4
  - Diretiva **.align n** do *assembler* (força o alinhamento do endereço de uma variável num valor **múltiplo de  $2^n$** )



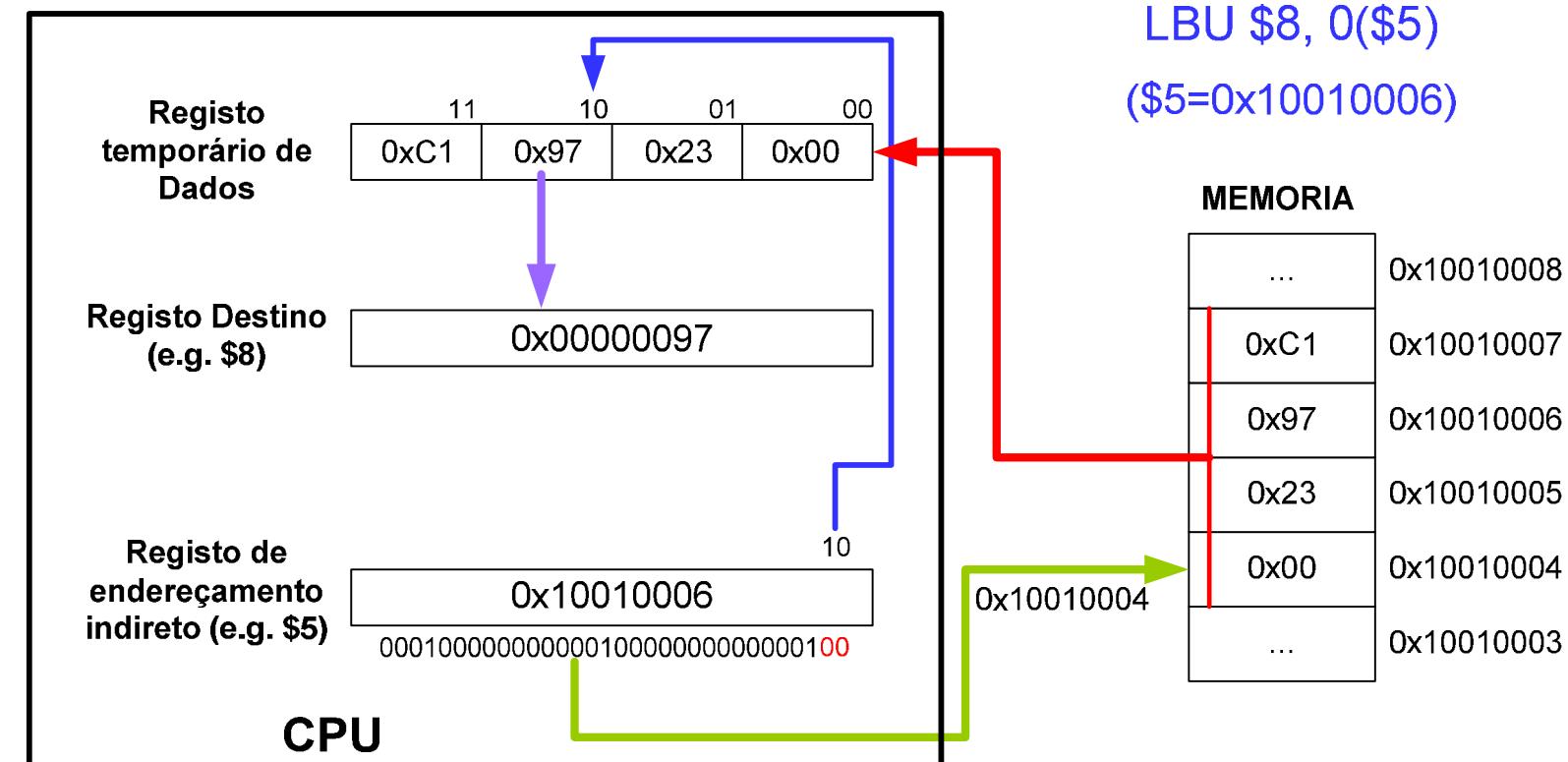
# Escrita / leitura de 1 byte na memória

- Na leitura/escrita de 1 byte de informação o problema do alinhamento, do ponto de vista do programador, não se coloca
- Como é que o MIPS resolve o acesso?
  - O MIPS gera o endereço múltiplo de 4 (EM4) que, no acesso a uma *word*, inclui o endereço pretendido
  - No caso de **Leitura** (instruções **Ib**, **Ibu**):
    - Executa uma instrução de leitura de 1 *word* do endereço EM4 e, dos 32 bits lidos, retira os 8 bits correspondentes ao endereço pretendido
    - No caso de **Escrita** (instrução **sb**):
      - Executa uma instrução de leitura de 1 *word* do endereço EM4
      - De entre os 32 bits lidos substitui os 8 bits que correspondem ao endereço pretendido
      - Escreve a *word* modificada em EM4
      - Sequência conhecida como "**read-modify-write**"



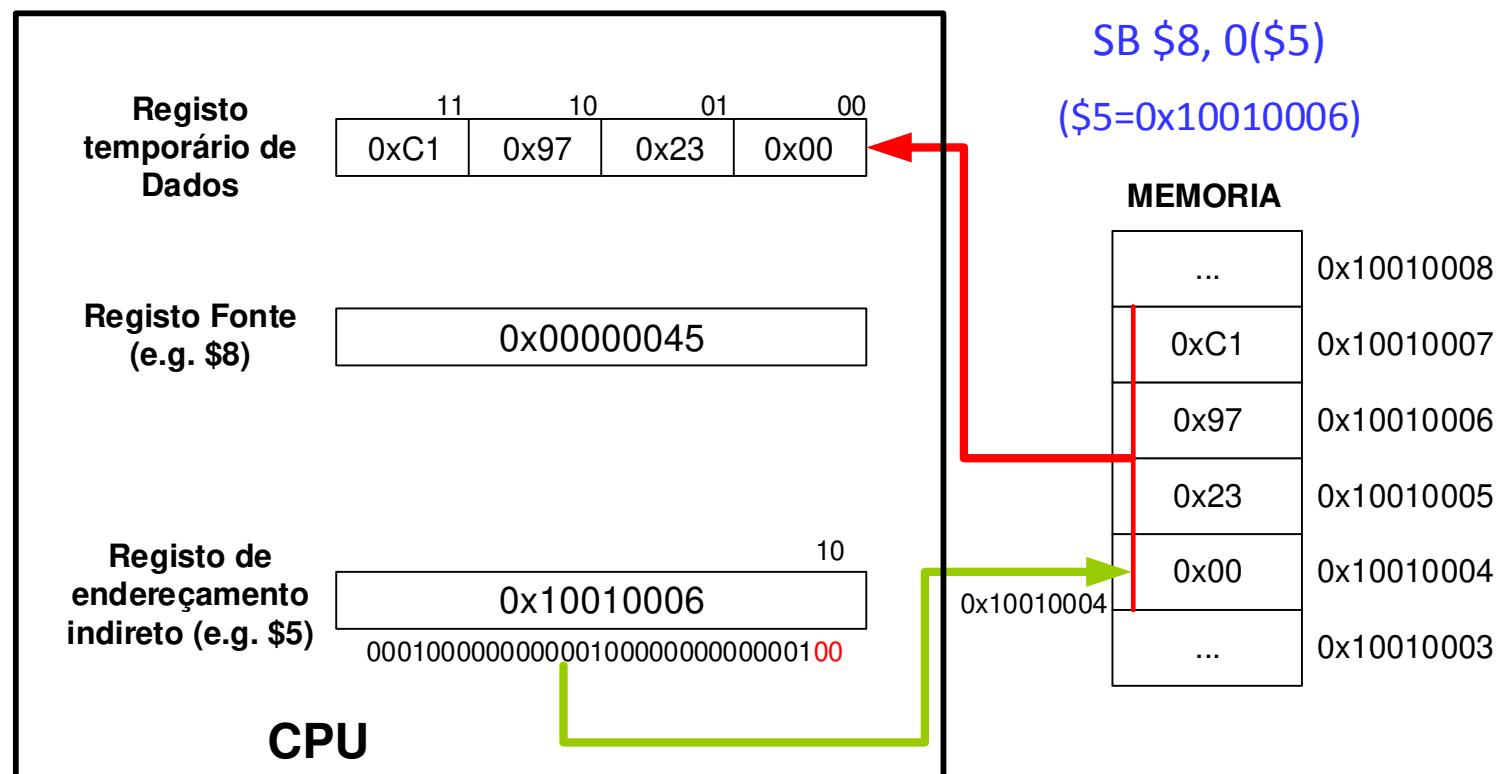
# Exemplo: leitura de 1 byte da memória

- Exemplo para o caso da leitura (instrução **Lbu** a ler o conteúdo da posição de memória **0x10010006**)



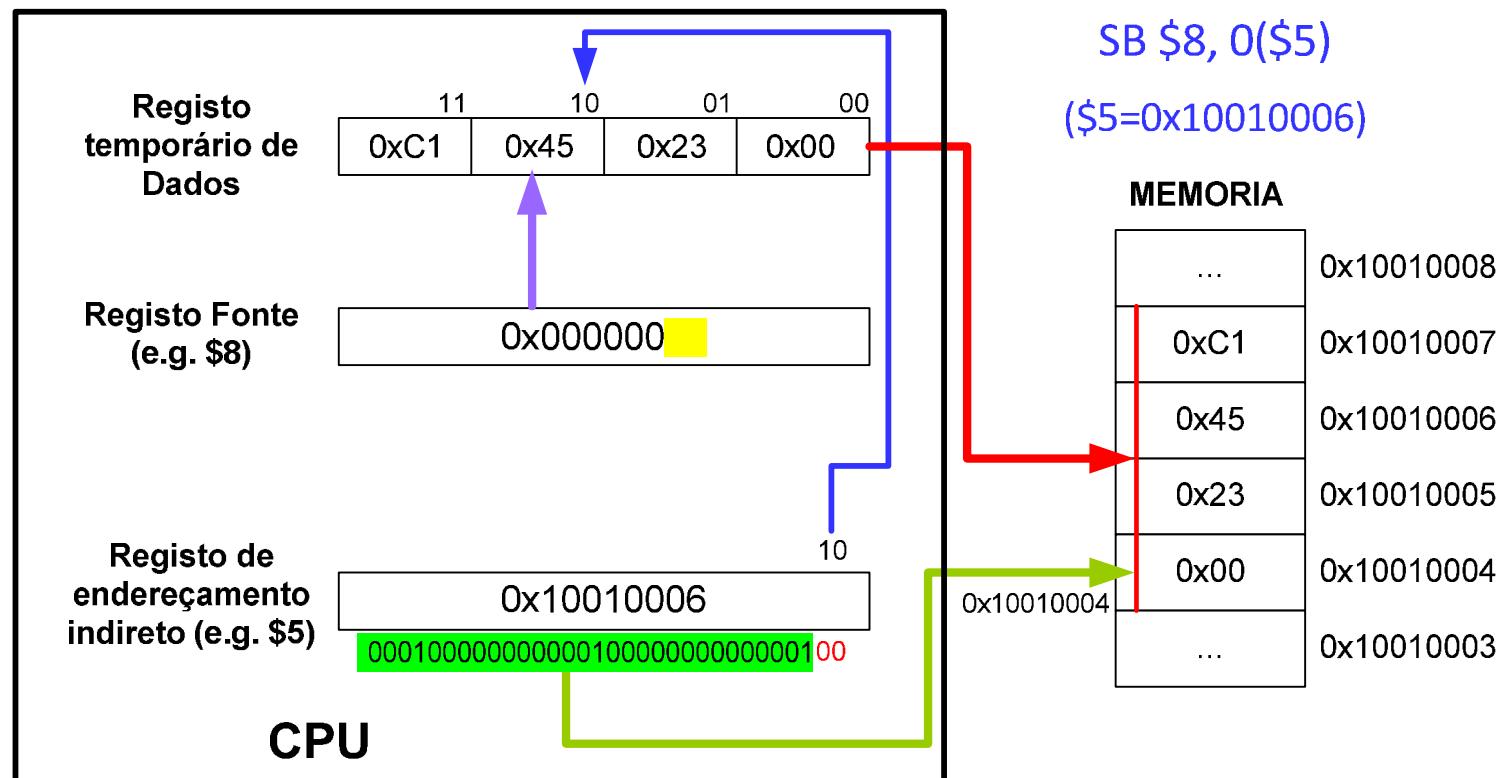
# Exemplo: escrita de 1 byte da memória (fase 1)

- Exemplo para o caso da escrita (instrução **sb** a escrever o conteúdo da posição de memória **0x10010006**) - READ



# Exemplo: escrita de 1 byte da memória (fase 2)

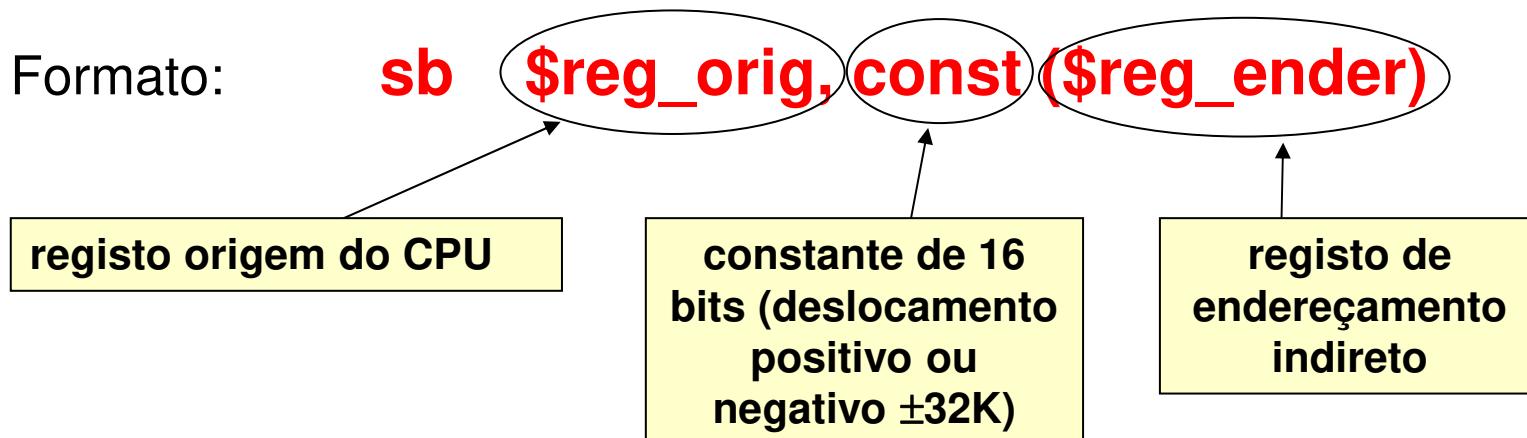
- Exemplo para o caso da escrita (instrução **sb** a escrever o conteúdo da posição de memória **0x10010006**) – MODIFY / WRITE



# Instrução de escrita de 1 byte na memória - SB

- **SB** - (store byte) transfere um *byte* de um registo interno para a memória – só são usados os 8 bits menos significativos

Formato:



Exemplo:

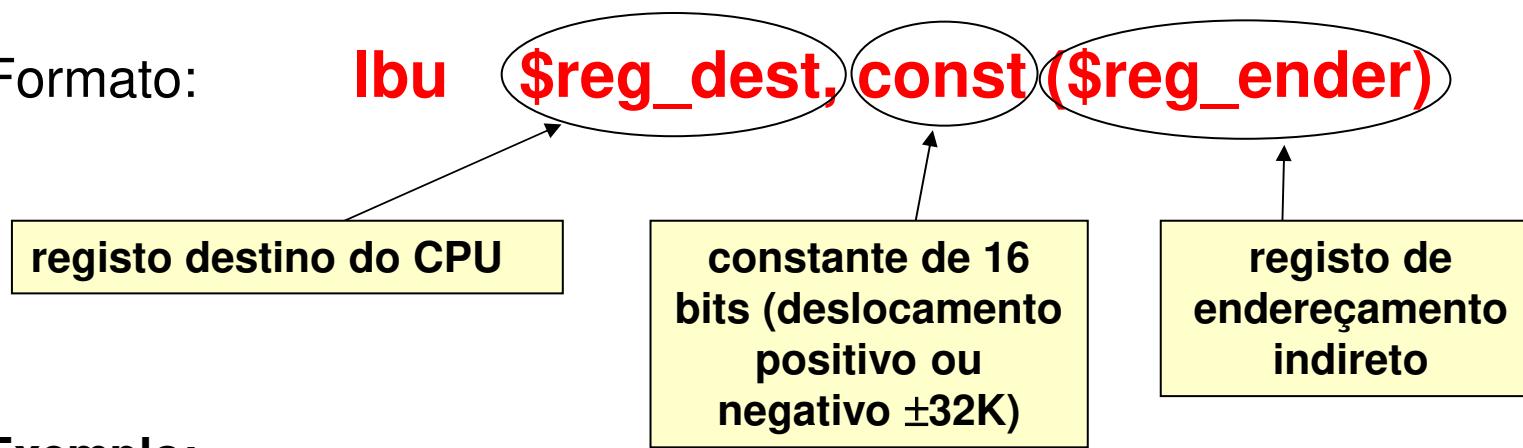
**sb \$7, 8 (\$4)** # transfere o *byte* armazenado no registo \$7 (8 bits menos significativos) para o endereço de memória calculado como:  
# addr = (conteúdo do registo \$4) + 8



# Instrução de leitura de 1 byte na memória - LBU

- **LBU** - (load byte unsigned) transfere um *byte* da memória para um registo interno - os 24 bits mais significativos do registo destino são colocados a 0

Formato:



Exemplo:

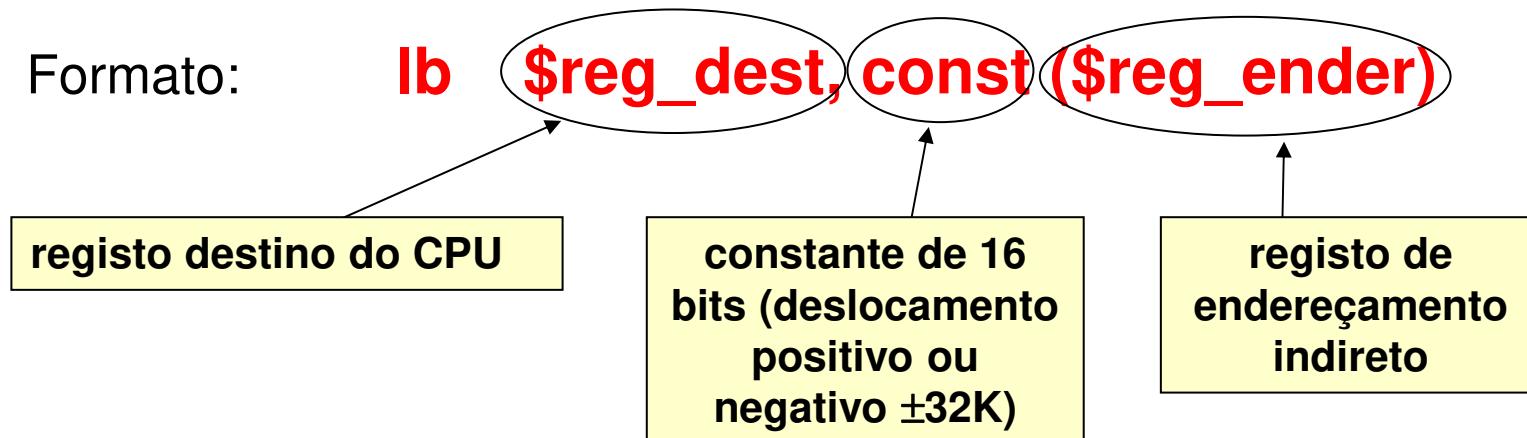
Ibu \$5, -4 (\$2)    # transfere para o registo \$5 o byte armazenado  
                      # no endereço de memória calculado como:  
                      #     addr = (conteúdo do registo \$2) - 4  
                      # os 24 bits mais significativos de \$5 são  
                      # colocados a zero



# Instrução de leitura de 1 byte na memória - LB

- LB - (load byte) transfere um byte da memória para um registo interno, fazendo extensão de sinal do valor lido de 8 para 32 bits

Formato:



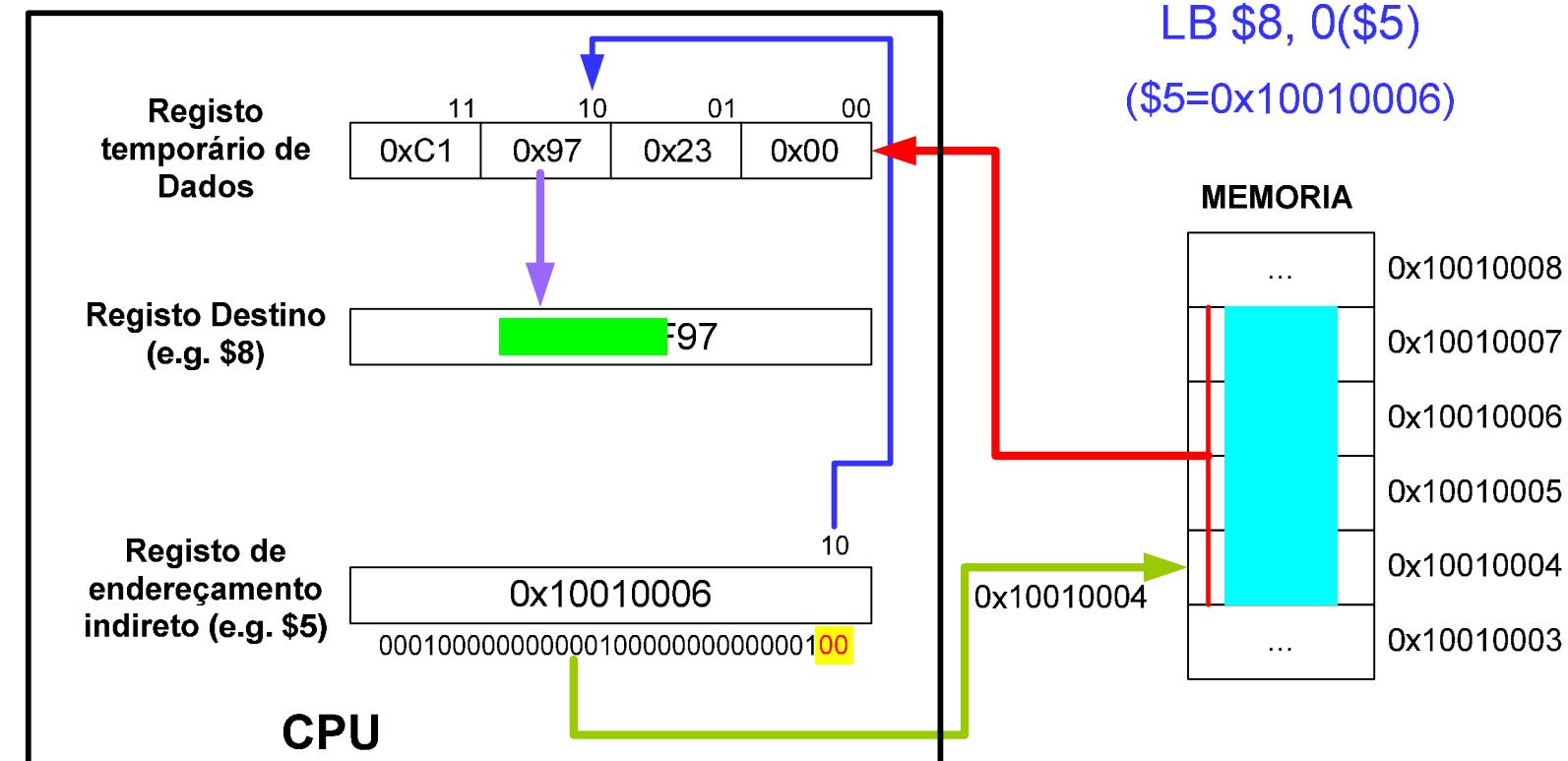
Exemplo:

lb \$5, 0 (\$2)      # transfere para o registo \$5 o byte armazenado  
# no endereço de memória calculado como:  
#      addr = (conteúdo do registo \$2) + 0  
# o bit mais significativo do byte transferido é  
# replicado nos 24 bits mais significativos de \$5



# Exemplo: leitura de 1 byte da memória

- Exemplo para o caso da leitura (instrução **lb** a ler o conteúdo da posição de memória **0x10010006**)



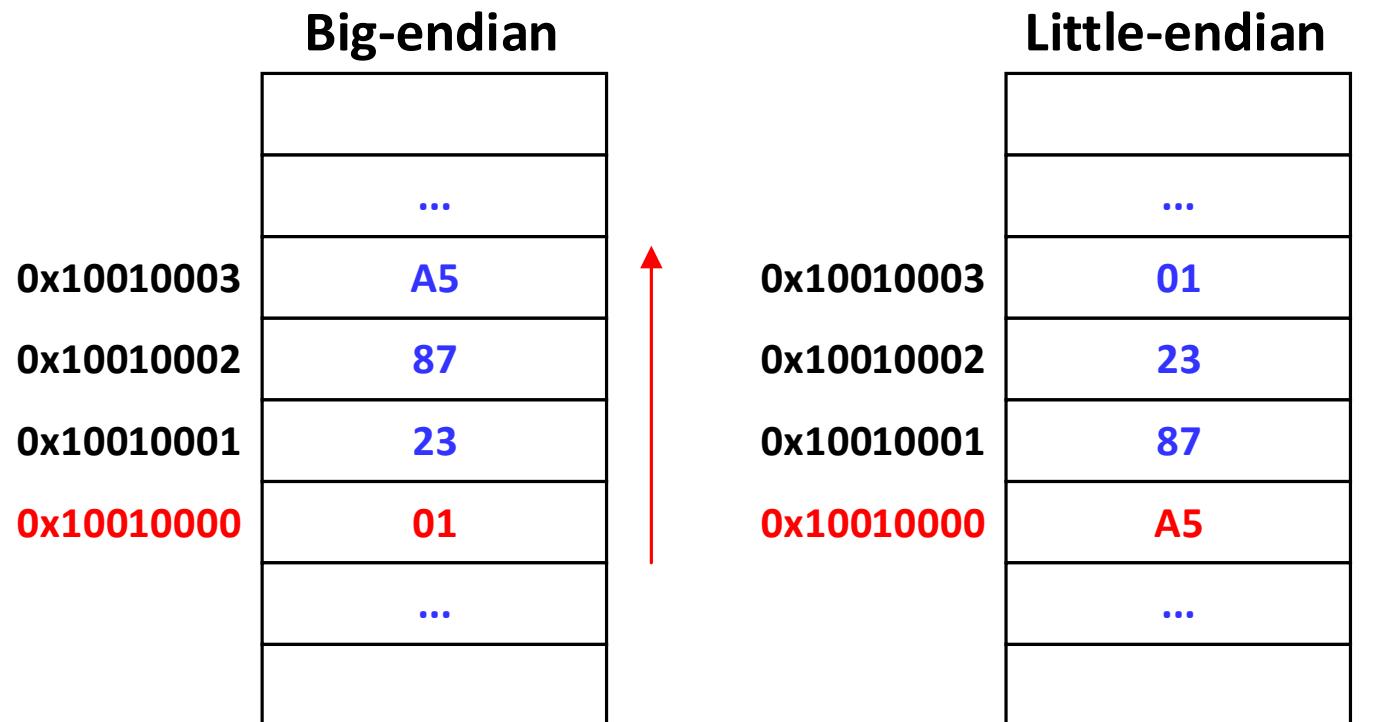
# Organização da informação na memória

- A memória no MIPS está organizada em *bytes* (*byte-addressable memory*)
- Se a quantidade a armazenar tiver uma dimensão superior a 8 bits vão ser necessárias várias posições de memória consecutivas (por exemplo, para uma *word* de 32 bits são necessárias 4 posições de memória)
- Exemplo: **0x012387A5** (4 bytes: 01 23 87 A5)
- Qual a ordem de armazenamento dos *bytes* na memória? Duas alternativas:
  - *byte* mais significativo armazenado no endereço mais baixo da memória (formato ***big-endian***)
  - *byte* menos significativo armazenado no endereço mais baixo da memória (formato ***little-endian***)



# Organização da informação na memória

- Exemplo: 0x012387A5 (0x01 23 87 A5)



- O simulador MARS (usado nas aulas práticas) implementa o formato "little-endian"



## Aula 5

- Diretivas do *assembler* do MIPS
- Introdução à utilização de ponteiros em linguagem C
- Acesso sequencial a elementos de um *array* residente em memória: acesso indexado e acesso com ponteiros

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Diretivas do Assembler

- Diretivas são códigos especiais colocados num programa em linguagem *assembly* destinados a instruir o *assembler* para que este execute uma determinada tarefa
- Diretivas **não são instruções** da linguagem *assembly* (não fazem parte do ISA), não gerando qualquer código máquina
- As diretivas podem ser usadas com diversas finalidades:
  - reservar e inicializar espaço em memória para variáveis
  - controlar os endereços reservados para variáveis em memória
  - especificar os endereços de colocação de código e dados na memória
  - definir valores simbólicos
- As diretivas são específicas para um dado *assembler* (em AC1 usaremos as diretivas definidas pelo *assembler* do simulador MARS)



# Diretivas do Assembler do MIPS

|  |  |
|--|--|
| <b>.ASCIIIZ</b> <i>str</i>   | Reserva espaço e armazena a string <i>str</i> em sucessivas posições de memória; acrescenta o terminador '\0' ( <b>NUL</b> )   |
| <b>.SPACE</b> <i>n</i>   | Reserva <i>n</i> posições consecutivas (endereços) de memória, sem inicialização   |
| <b>.BYTE</b> <i>b</i> <sub>1</sub> , <i>b</i> <sub>2</sub> , ..., <i>b</i> <sub><i>n</i></sub> | Reserva espaço e armazena os bytes <i>b</i> <sub>1</sub> , <i>b</i> <sub>2</sub> , ..., <i>b</i> <sub><i>n</i></sub> em sucessivas posições de memória   |
| <b>.WORD</b> <i>w</i> <sub>1</sub> , <i>w</i> <sub>2</sub> , ..., <i>w</i> <sub><i>n</i></sub> | Reserva espaço e armazena as words <i>w</i> <sub>1</sub> , <i>w</i> <sub>2</sub> , ..., <i>w</i> <sub><i>n</i></sub> em sucessivas posições de memória (cada word em 4 endereços consecutivos) |
| <b>.ALIGN</b> <i>n</i>   | Alinha o próximo item num endereço múltiplo de 2 <sup><i>n</i></sup>   |
| <b>.EQV</b> <i>symbol</i> , <i>val</i>   | Substitui as ocorrências de <b><i>symbol</i></b> no programa por <b><i>val</i></b>   |



# Diretivas do Assembler - exemplo

```
.DATA # 0x10010000
STR1: .ASCIIIZ "AULA6"
ARR1: .WORD 0x1234, MAIN
VARB: .BYTE 0x12
.ALIGN 2
VARW: .SPACE 4 #space for 1 word
.TEXT # 0x00400000
.GLOBL MAIN

MAIN:
• Utilizar a diretiva ".align" sempre que se
pretender que o endereço subsequente
esteja alinhado
• A diretiva ".word" alinha automaticamente
num endereço múltiplo de 4
```

|                 |             |
|-----------------|-------------|
| 0x10010017      | ??          |
| 0x10010016      | ??          |
| 0x10010015      | ??          |
| VARW 0x10010014 | ??          |
| 0x10010013      | ?? (unused) |
| 0x10010012      | ?? (unused) |
| 0x10010011      | ?? (unused) |
| VARB 0x10010010 | 0x12        |
| 0x1001000F      | 0x00        |
| 0x1001000E      | 0x40        |
| 0x1001000D      | 0x00        |
| 0x1001000C      | 0x00        |
| 0x1001000B      | 0x00        |
| 0x1001000A      | 0x00        |
| 0x10010009      | 0x12        |
| ARR1 0x10010008 | 0x34        |
| 0x10010007      | ?? (unused) |
| 0x10010006      | ?? (unused) |
| 0x10010005      | '\0' (0x00) |
| 0x10010004      | '6' (0x37)  |
| 0x10010003      | 'A' (0x41)  |
| 0x10010002      | 'L' (0x4C)  |
| 0x10010001      | 'U' (0x55)  |
| STR1 0x10010000 | 'A' (0x41)  |

## Linguagem C: ponteiros e endereços – o operador &

- Um **ponteiro** é uma **variável que contém o endereço de outra variável** – o acesso à 2<sup>a</sup> variável pode fazer-se indiretamente através do ponteiro
- Exemplo:
  - **x** é uma variável (por ex. um inteiro) e **px** é um ponteiro. O **endereço da variável x** pode ser obtido através do **operador &**, do seguinte modo:
  - **px = &x; // Atribui o endereço de "x" a "px"**
  - Diz-se que **px é um ponteiro que aponta para x**
- O operador **&** apenas pode ser utilizado com variáveis e elementos de arrays.
  - Exemplos de utilizações **erradas**:
  - **&5;      &(x+1);**



# Ponteiros e endereços – o operador \*

- O operador "\*":

- trata o seu operando como um endereço
  - permite o acesso ao endereço para obter o respetivo conteúdo

- Exemplo:

```
y = *px; // Atribui o conteúdo da variável  
           // apontada por "px" a "y"
```

- A sequência:

```
px = &x; // px é um ponteiro para x  
y = *px; // *px é o valor de x
```

Atribui a **y** o mesmo valor que: **y = x;**



# Ponteiros e endereços – declaração de variáveis

- As variáveis envolvidas têm que ser declaradas
- Para o exemplo anterior, supondo que se tratava de variáveis inteiras:

```
int x, y; // x, y - variáveis do tipo inteiro
int *px; // ou int* px; (ponteiro para inteiro)
           // Esta declaração apenas reserva o
           // espaço para o ponteiro
```

- A declaração do ponteiro (**int \*px;** ou **int\* px;**) deve ser entendida como uma mnemónica e significa que **px é um ponteiro** e que o conjunto **\*px é do tipo inteiro**
- Exemplos de **declarações de ponteiros**:

```
char *p; // p é um ponteiro para caracter
double *v; // v é um ponteiro para double
```



# Manipulação de ponteiros em expressões

- Exemplo: supondo que **px** aponta para **x** (**px = &x;**), a expressão **y = \*px + 1**; atribui a **y** o valor de **x** acrescido de 1
- Os ponteiros podem igualmente ser utilizados na parte esquerda de uma expressão. Por exemplo, (supondo que **px = &x;**)

**\*px = 0;** // equivalente a **x=0**

ou

**\*px = \*px + 1;** // equivalente a **x = x + 1**

**\*px += 1;** // o mesmo que **\*px = \*px + 1**

**(\*px)++;** // o mesmo que **\*px = \*px + 1**



# Ponteiros como argumentos de funções

- Em C os argumentos das funções são passados por valor (cópia do conteúdo da variável original)
- Isso significa que a função chamada só pode alterar diretamente o valor da cópia da variável original, isto é, uma função chamada não pode alterar diretamente o valor de uma variável da função chamadora
- Se tal for necessário, a solução reside na utilização de ponteiros
- **Exemplo:** implementar uma função para a troca do conteúdo de duas variáveis (**troca(a, b);**):
  - Se, antes da chamada à função,  $a=2$  e  $b=5$ , então
  - Após a chamada à função:  $a=5$  e  $b= 2$



# Ponteiros como argumentos de funções

```
void troca(int, int);

void main(void)
{
    int a, b;
    (...)

    if(a < b)
        troca(a, b);
    (...)

}
```

```
void troca(int *,int *);

void main(void)
{
    int a, b;
    (...)

    if(a < b)
        troca(&a, &b);
    (...)

}
```

```
void troca(int x, int y)
{
    int aux;
    aux = x;
    x = y;
    y = aux;
}
```

```
void troca(int *x, int *y)
{
    int aux;
    aux = *x;    // aux = a
    *x = *y;    // a = b
    *y = aux;   // b = aux
}
```



# Ponteiros e arrays

- Sejam as declarações

```
int a[10]; // array de inteiros "a" com  
           // 10 elementos  
  
int *pa;   // ponteiro para um inteiro  
  
int v;     // variável do tipo inteiro
```

- A expressão `pa = &a[0]`; atribui a `pa` o endereço do 1º elemento do array. Então, a expressão `v = *pa`; atribui a `v` o valor de `a[0]`
- Se `pa` aponta para um dado elemento do array, `pa+1` aponta para o seguinte
- Se `pa` aponta para o primeiro elemento do array, então `(pa+i)` aponta para o elemento `i` e `* (pa+i)` refere-se ao seu conteúdo
- A expressão `pa = &a[0]`; pode também ser escrita como `pa=a`; isto é, o nome do array é o endereço do seu primeiro elemento



# Aritmética de Ponteiros

- Se **pa** é um ponteiro, então a expressão **pa++**; incrementa **pa** de modo a apontar para o elemento seguinte (seja qual for o tipo de variável para o qual **pa** aponta)
- Do mesmo modo **pa = pa + i**; incrementa **pa** para apontar para **i** elementos à frente do elemento atual
- **A tradução das expressões anteriores para Assembly tem que ter em conta o tipo de variável para o qual o ponteiro aponta**
  - Por exemplo, se um inteiro for definido com 4 bytes (32 bits), então a expressão **pa++**; implica adicionar 4 ao valor atual do endereço correspondente



## Exemplo 1

- Analise o código C deste e dos slides seguintes e determine o resultado produzido

```
void main(void)
{
    char s[] = "Hello"; // "s" é um array de
                        // carateres (string)
                        // terminado com o
                        // carater '\0' (0x00)
    int i = 0;

    while(s[i] != '\0')
    {
        printf("%c", s[i]); // imprime carater
        i++;
    }
}
```



## Exemplo 2

```
void main(void)
{
    char s[] = "Hello";
    char *p; // Declara um ponteiro para
              // carater (reserva espaço)
    p = s;   // Inicializa o ponteiro com
              // endereço inicial do array
    while(*p != '\0')
    {
        printf("%c", *p); // imprime carater
        p++; // incrementa o ponteiro
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (\*p)
- O ponteiro é depois incrementado, i.e., fica a apontar para o carater seguinte do array



## Exemplo 3

```
void main(void)
{
    char s[] = "Hello";
    char *p1 = s; // p1 = &s[0]
    char *p2 = s; // p2 = &s[0]

    while(*p2 != '\0') p2++;
    while(p1 < p2)
    {
        printf("%c", *p1);
        p1++;
    }
}
```

- Após o primeiro "while" o ponteiro "p2" aponta para o fim da *string* (i.e., para o carater '\0')
- O ponteiro "p1" é usado pelo "printf()" para aceder ao carater a imprimir (\*p1); o ponteiro "p1" é incrementado na linha seguinte



## Exemplo 4

```
void main(void)
{
    char s[] = "Hello";
    char *p = s;

    while(*p != '\0')
    {
        printf("%c", *p++);
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (\*p)
- O ponteiro "p" é incrementado após o acesso ao conteúdo (pós-incremento)
- Esta construção é equivalente à do exemplo 2



## Exemplo 5

```
void main(void)
{
    char s[] = "Hello";
    char *p = s;
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("%c", (*p)++);
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (\*p)
- A operação de pos-incremento está a ser aplicada à variável apontada pelo ponteiro
- Neste exemplo o ponteiro "p" nunca é incrementado
- Qual a sequência de carateres impressa?



# Acesso sequencial a elementos de um *array*

- O acesso sequencial a elementos de um *array* apoia-se em uma de duas estratégias:
  1. Acesso indexado, isto é, endereçamento a partir do nome do *array* e de um índice que identifica o elemento a que se pretende aceder:  
 **$f = a[i];$**
  2. Utilização de um ponteiro (endereço armazenado num registo) que identifica em cada instante o endereço do elemento a que se pretende aceder:  
 **$f = *pt;$**  // com  $pt$  = endereço de  $a[i]$  ( i.e.  **$pt = &a[i]$**  )
- Estas 2 formas de acesso traduzem-se em **implementações distintas** em *assembly*



# Acesso sequencial a elementos de um *array*

- Acesso indexado

- $f = a[i];$  // Com  $i \geq 0$
- Para aceder ao elemento "*i*" do array "*a*", o programa começa por calcular o respetivo endereço, a partir do endereço inicial do array:

**endereço do elemento a aceder = endereço inicial do array +  
(índice \* dimensão em bytes de cada posição do array)**

- Acesso por ponteiro

- $f = *pt;$
- O endereço do elemento a aceder está armazenado num registo

**endereço do elemento seguinte = endereço actual +  
dimensão em bytes de cada posição do array**



# Exemplos de acesso sequencial a arrays

```
// Exemplo 1
int i;
static int array[size];

for(i = 0; i < size; i++) {
    array[i] = 0;
}
```

Acesso indexado

```
// Exemplo 2
int *p;
static int array[size];

for(p=&array[0]; p < &array[size]; p++)
{
    *p = 0;
}
```

Acesso por ponteiro

Também pode ser escrito como: `for(p=array; p < array+size; p++)`



# Acesso sequencial a arrays – exemplo 1

```
#define SIZE 10
```

```
void main(void) {
```

```
    int i;
```

```
    static int array[SIZE];
```

```
    for (i = 0; i < SIZE; i++)
```

```
        array[i] = 0;
```

```
}
```

\$t0 > i

\$t1 > temp

\$t2 > &(array[0])

\$a0 > SIZE

```
.DATA
```

```
array: .SPACE 40 # SIZE*4 não é suportado pelo MARS
```

```
.EQV SIZE, 10
```

```
.TEXT
```

```
.GLOBL main
```

```
main: li $a0, SIZE
```

```
li $t0, 0 # i = 0;
```

```
loop: bge $t0, $a0, endf # while (i < size) {
```

```
    la $t2, array # $t2 = &(array[0]);
```

```
    sll $t1, $t0, 2 # temp = i * 4;
```

```
    addu $t1, $t2, $t1 # temp = &(array[i])
```

```
    sw $0, 0($t1) # array[i] = 0;
```

```
    addi $t0, $t0, 1 # i = i + 1;
```

```
    j loop # }
```

```
endf: ...
```



# Acesso sequencial a arrays – exemplo 2

```
#define SIZE 10
void main(void) {
    int *p;
    static int array[size];
    for (p=&array[0]; p < &array[size]; p++) {
        *p = 0;
    }
}
```

```
.DATA
array: .SPACE 40          #SIZE * 4      $t0 > p
       .EQV   SIZE, 10
.TEXT
.GLOBL main
main:  li     $a0, SIZE      #
       la     $t0, array      # $t0 = &(array[0]);
       sll   $t1, $a0, 2      # $t1 = size * 4;
       addu  $t1, $t1, $t0      # $t1 = &(array[size]);
loop:  bgeu  $t0, $t1, endf  # while (p < &array[size]) {
       sw    $0, 0($t0)      #     *p = 0;
       addiu $t0, $t0, 4      #     p = p + 1;
       j    loop             # }
endf: ...
```



## Aula 6

- Métodos de endereçamento em saltos condicionais e incondicionais
- Codificação das instruções de salto condicional no MIPS
- Codificação das instruções de salto incondicional no MIPS: o formato J
- Endereçamento imediato e uso de constantes
- Resumo dos modos de endereçamento do MIPS

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

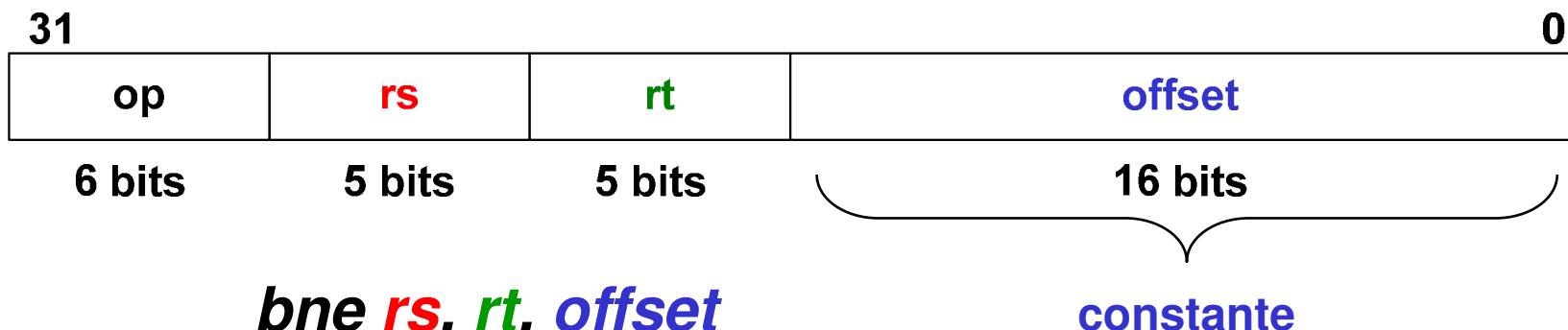


## Codificação das instruções de salto condicional no MIPS

- As instruções aritméticas e lógicas no MIPS são codificadas no **formato R**



- Por outro lado, a necessidade de codificação do **endereço-alvo** nas instruções de salto condicional obriga a que estas instruções sejam codificadas recorrendo ao **formato I**



# Codificação de *branches* no MIPS

Exemplo: **bne \$8, \$21, Exit**

|    |   |   |    |   |
|----|---|---|----|---|
| 31 | 5 | 8 | 21 | 0 |
|----|---|---|----|---|



***bne rs, rt, offset***

- Se o endereço alvo for codificado nos 16 bits menos significativos da instrução, isso significa que o programa não pode ter uma dimensão superior a  $2^{16}$  (64K). Será essa uma opção realista?
- Uma alternativa mais interessante poderia passar por especificar um registro interno cujo conteúdo pudesse ser somado à constante codificada na instrução (*offset*), de tal modo que **PC = Reg + Offset**
- Desta forma a dimensão máxima do programa já poderia ser  $2^{32}$



# Codificação de *branches* no MIPS

## Note-se contudo que:

- A maioria das instruções de salto condicional realizam esse salto para a vizinhança da própria instrução
- Como exemplo verifica-se, estatisticamente, que no gcc (GNU C Compiler) quase metade de todos os saltos condicionais são para endereços correspondentes a uma gama de  $\pm 16$  instruções)
- Com 16 bits é possível endereçar  $2^{16}$  endereços distintos (64K endereços)
- **No MIPS todas as instruções são armazenadas em endereços múltiplos de 4** (e.g. 0x00400000, 0x00400004, 0x00400008, ...)
- Qual deverá ser então o registo-base a usar?



# Codificação de *branches* no MIPS

- **Solução:**
  - Utilizar o registo PC (Program Counter)
  - Usar **endereçamento relativo**: o valor do endereço alvo é calculado somando algebraicamente o *offset* de 16 bits, codificado na instrução, ao valor corrente do PC (o valor do *offset* é extendido com sinal para 32 bits)
- No MIPS o valor do PC, na fase de execução de um "branch", corresponde ao endereço da instrução seguinte, uma vez que esse registo é incrementado na fase “*fetch*” da instrução
- Assim, o endereço-alvo (novo PC) é calculado como:

$$\text{Novo\_PC} = \text{PC\_atual} + \text{offset}$$

O *offset* de 16 bits é interpretado como um **valor em complemento para dois**, permitindo o salto para **endereços anteriores (offset negativo) ou posteriores (offset positivo)** ao PC



# Codificação de *branches* no MIPS

Considere-se o seguinte exemplo:

|            |         |                      |
|------------|---------|----------------------|
| 0x00400000 | bne     | \$19, \$20, ELSE     |
| 0x00400004 | add     | \$16, \$17, \$18     |
| 0x00400008 | j       | END_IF               |
| 0x0040000C | ELSE:   | sub \$16, \$16, \$19 |
| 0x00400010 | END_IF: |                      |

O endereço correspondente ao label ELSE é 0x0040000C

Durante o *instruction fetch* o PC é incrementado (i.e. PC=0x00400004)

O "branch\_offset" seria portanto:  
ELSE - [PC] =  
0x0040000C - 0x00400004 = 0x08

No entanto, como **cada instrução ocupa sempre 4 bytes** na memória (a partir de um endereço múltiplo de 4), o "branch\_offset" é **calculado em instruções**. Logo:

"branch\_offset" = 0x08 / 4 = 0x02

|    |   |    |    |        |
|----|---|----|----|--------|
| 31 | 5 | 19 | 20 | 0      |
|    |   |    |    | 0x0002 |

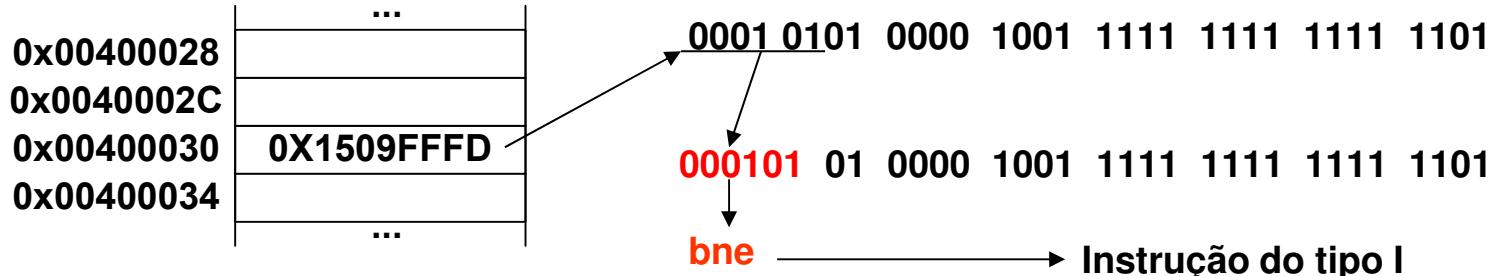
Código máquina: 00010110011101000000000000000010 = 0x16740002

Uma instrução de salto condicional pode referenciar qualquer endereço de uma outra instrução que se situe até **32K instruções** antes ou depois dela própria.



# Interpretação de uma instrução de *branch* pelo CPU

## Exemplo

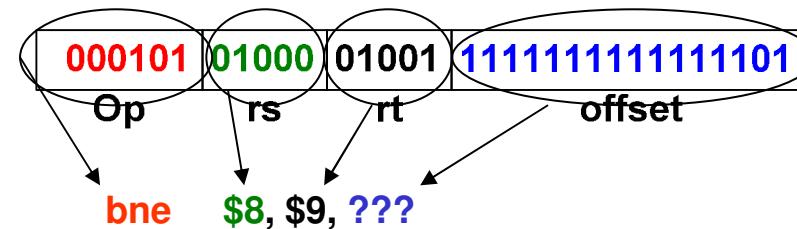


Offset = 1111 1111 1111 1101

negativo

complemento para dois

- 0000 0000 0000 0011 = -3<sub>10</sub>



O valor do PC foi incrementado na fase *fetch* da instrução

Endereço alvo = PC + offset \* 4 = 0x00400034 + (-3 \* 4) = 0x00400034 - 0x0C

Endereço alvo = 0x00400028

Instrução descodificada: bne \$8, \$9, 0x00400028



## Codificação da instrução de salto incondicional no MIPS

- No caso da instrução de salto incondicional ("j"), é usado **endereçamento pseudo-direto**, i.e. o **código máquina** da instrução **codifica diretamente parte do endereço alvo** (endereço do qual será lida a próxima instrução)
- **Exemplo:** a instrução **j Label #se Label=0x001D14C8**

será codificada como:

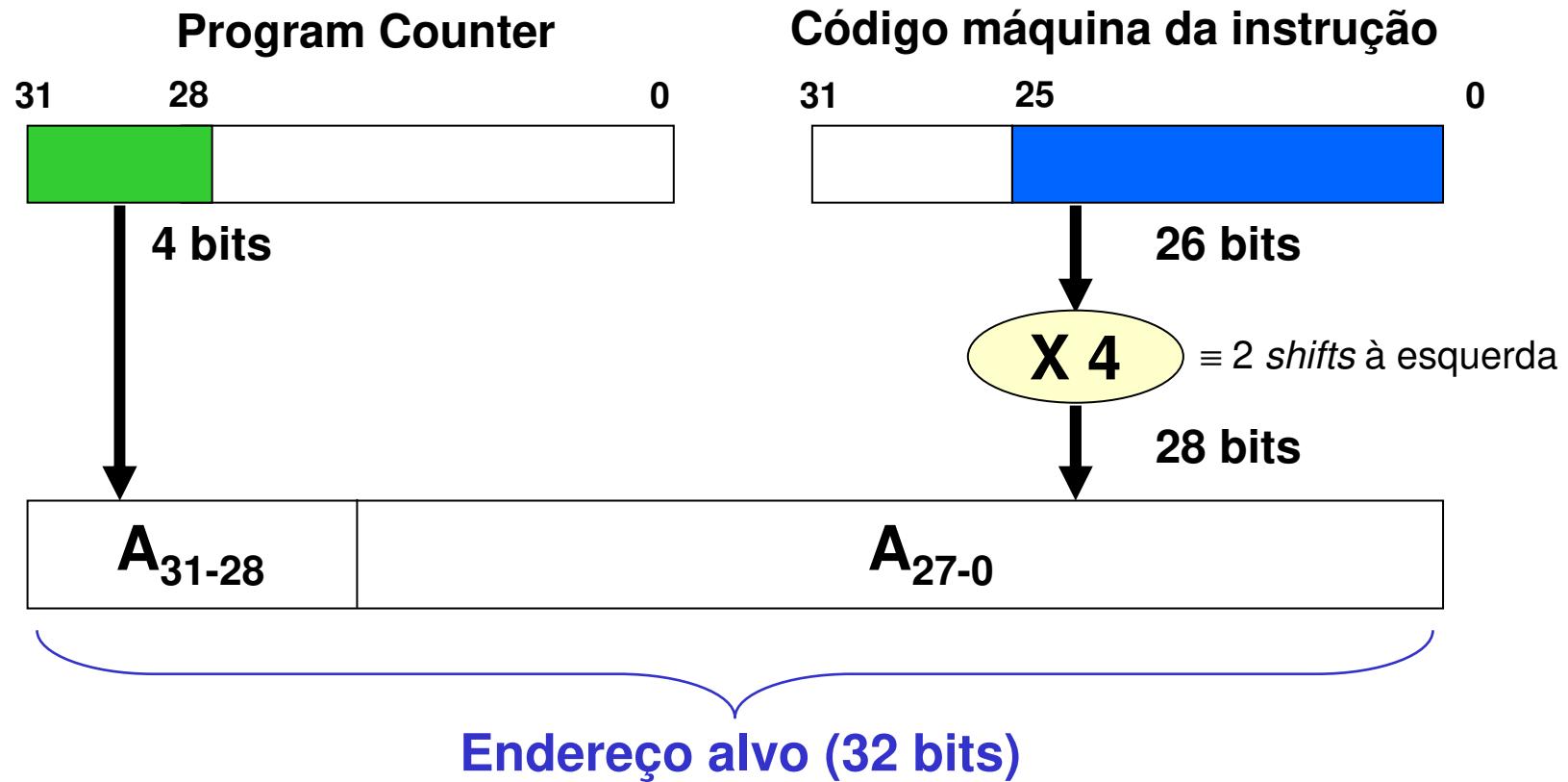


**Formato J**

Código Máquina: **0000100000001110100010100110010<sub>2</sub> = 0x08074532**

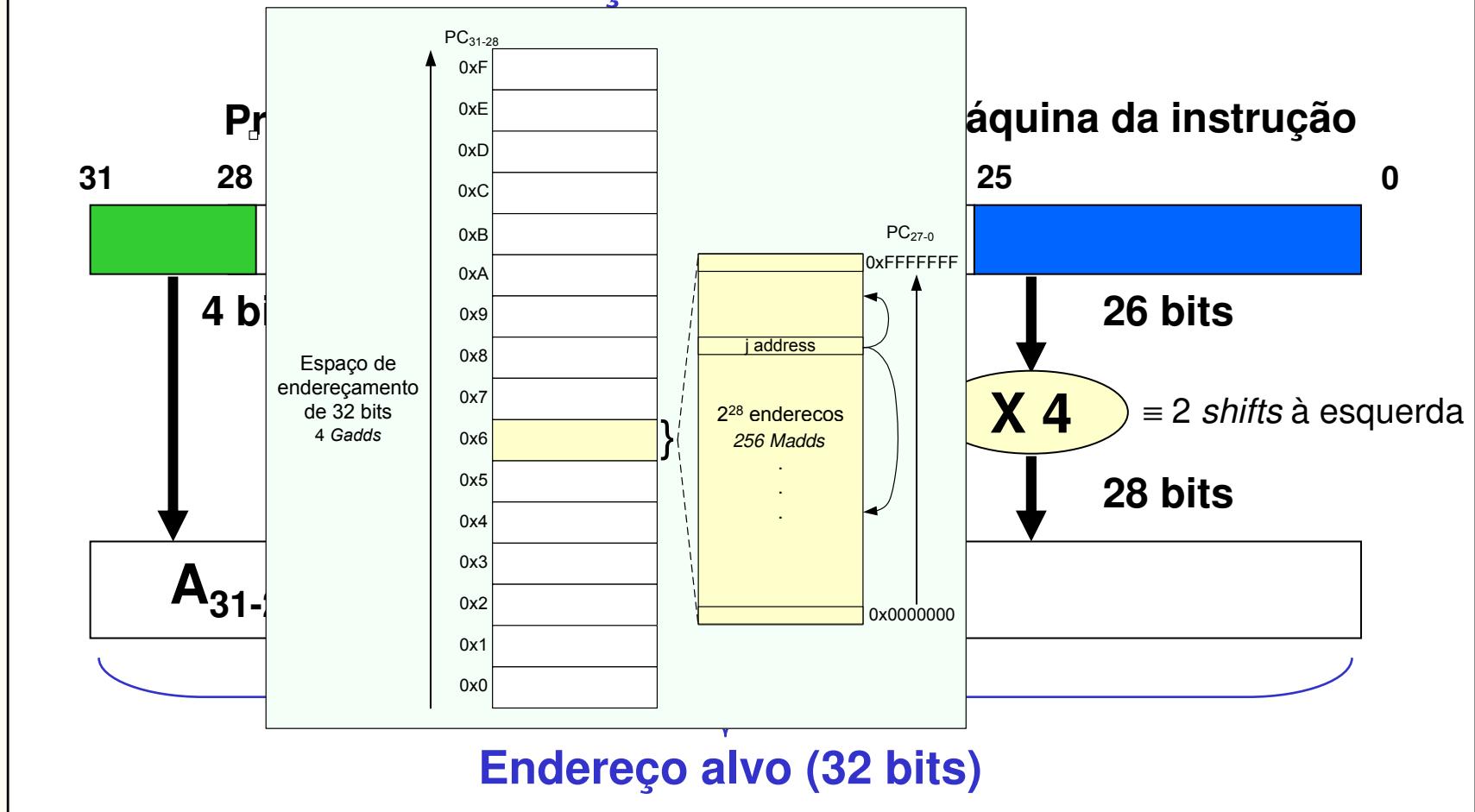
## Cálculo, no CPU, do endereço-alvo de uma instrução J

Se a instrução só codifica 28 bits (26 explícitos + 2 implícitos),  
**como é formado o endereço final de 32 bits?**



# Cálculo, no CPU, do endereço-alvo de uma instrução J

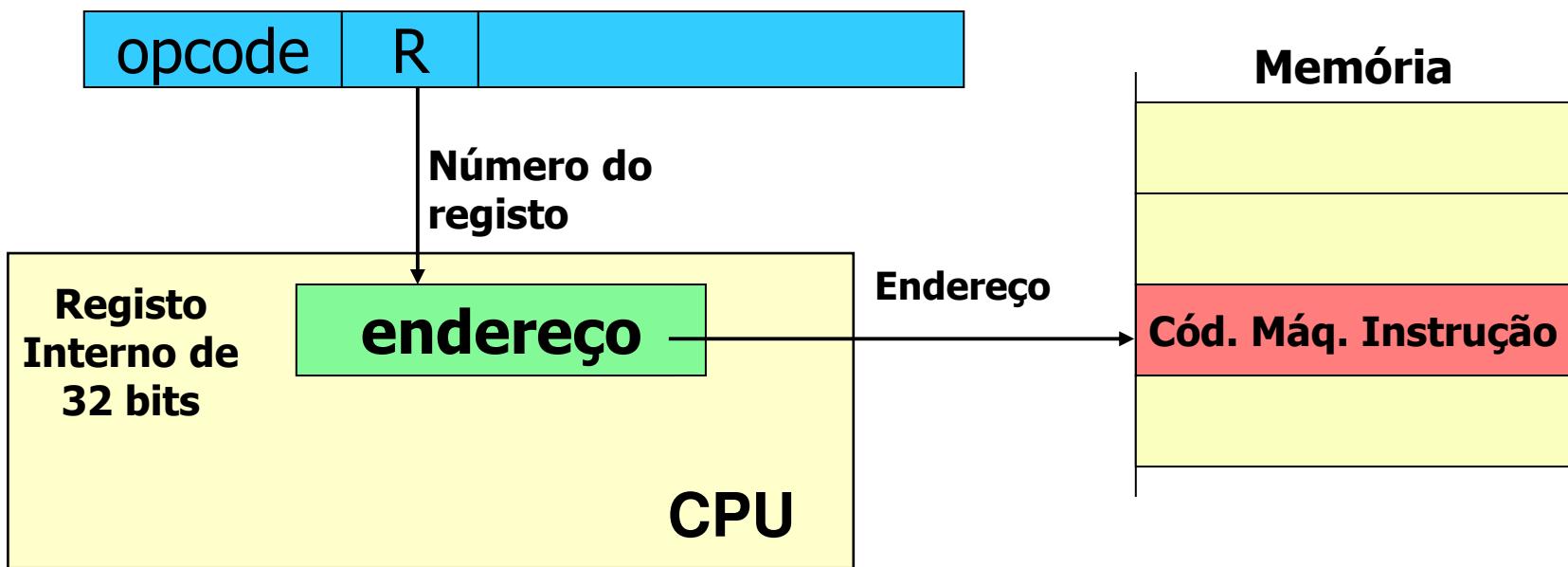
Se a instrução só codifica 28 bits (26 explícitos + 2 implícitos),  
**como é formado o endereço final de 32 bits?**



## Salto incondicional – endereçamento indireto por registo

- Haverá maneira de especificar, numa instrução de salto incondicional, um endereço-alvo de 32 bits?
- Há! Utiliza-se **endereçamento indireto por registo**. Ou seja, um registo interno (de 32 bits) armazena o endereço alvo da instrução de salto (**instrução JR**)

(Código máquina da instrução)



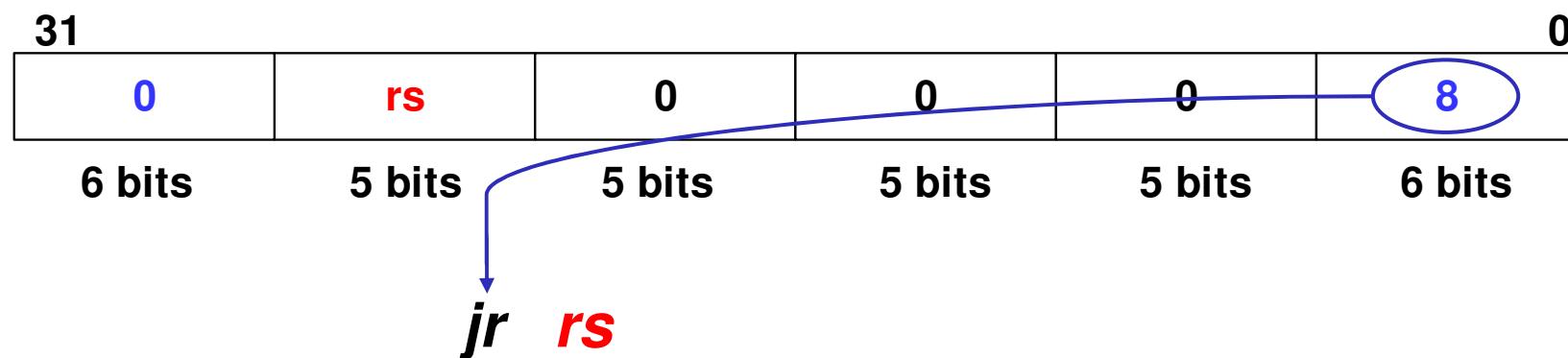
# Instrução JR (jump on register)

```
jr    Rsrc  # salta para o endereço que  
          # se encontra armazenado no registo Rsrc
```

Exemplo:

```
jr    $ra   # Salta para o endereço que está  
          # armazenado no registo $ra
```

O formato de codificação da instrução JR é o formato R:



# Modos de endereçamento no MIPS

- Instruções aritméticas e lógicas: **endereçamento tipo registo**
- Instruções de acesso à memória : **endereçamento indireto por registo com deslocamento**
- Instruções de salto condicional: **endereçamento relativo ao PC**
- Na instrução de salto incondicional através de um registo (instrução **JR** ) é usado um registo interno do processador para armazenar o endereço-alvo da instrução de salto: **endereçamento indireto por registo**
- Na instrução de salto incondicional (**J**) é usado **endereçamento direto** (uma vez que o endereço não é especificado na totalidade, esse tipo de endereçamento é normalmente designado por "pseudo-direto")



# Modos de endereçamento no MIPS

- Para além dos modos de endereçamento registo e indireto por registo, o MIPS suporta ainda um outro tipo de endereçamento, designado por “**endereçamento imediato**”.
- Relembrando os quatro princípios básicos subjacentes ao design de uma arquitetura
  - A simplicidade favorece a regularidade
  - Quanto mais pequeno mais rápido
  - Um bom design implica compromissos adequados
  - **O que é mais comum deve ser mais rápido**
- O último ponto determina que a capacidade de tornar mais rápida a execução das operações que ocorrem mais vezes, resulta num aumento global do desempenho!



# Endereçamento imediato

- Pode verificar-se, estatisticamente, que um número muito significativo de instruções em que está envolvida uma operação aritmética usa uma **constante** como um dos seus operandos
- É vulgar que este número seja superior a 50% do total das instruções que envolvem a ALU num determinado programa.
  - Chama-se **constante** a um valor determinado com antecedência (na altura em que o programa é escrito) e que não se pretende que seja ou possa ser mudado durante a execução do programa.



# Endereçamento imediato

- A constante “zero” é tão usada, que o MIPS tem um registo interno onde esse valor está permanentemente disponível (**\$0**)
- A constante “um”, por outro lado, também é muito utilizada em operações de incremento ou decremento de variáveis de contagem usadas em ciclos
- As constantes poderiam ser armazenadas na memória externa. Nesse caso, a sua utilização implicaria sempre o recurso a duas instruções:
  - leitura do valor residente em memória para um registo interno
  - operação com essa constante



# Endereçamento imediato

- Para aumentar a eficiência, as arquiteturas disponibilizam, habitualmente, um conjunto de instruções em que as **constantes se encontram armazenadas na própria instrução**
- Desta forma o acesso à constante é “**imediato**”, sem necessidade de recorrer a uma operação prévia de leitura da memória
- No caso do MIPS as instruções aritméticas e lógicas do tipo imediato são identificadas pelo sufixo “**i**”:

```
addi $3,$5,4      # $3 = $5 + 0x0004
andi $17,$18,0x3AF5 # $17 = $18 & 0x3AF5
ori $12,$10,0x0FA2 # $12 = $10 | 0x0FA2
slti $2,$12,16     # $2 = 1 se $12 < 16
                    # ($2 = 0 se $12 ≥ 16)
```



## Endereçamento imediato – gama de representação

- Se todas as instruções do MIPS ocupam um espaço de armazenamento de 32 bits, **quantos desses 32 bits são dedicados a armazenar o “valor imediato”?**
- Estas instruções são codificados usando o **formato I**. Logo a resposta é **16 bits**
- Este espaço é geralmente suficiente para armazenar as constantes mais frequentemente utilizadas (geralmente valores pequenos)
- Se há apenas 16 bits dedicados ao armazenamento da constante, qual será a **gama de representação** dessa constante?
  - Depende da instrução...



## Endereçamento imediato – gama de representação

- No caso mais geral, a constante representa uma quantidade inteira, positiva ou negativa, codificada em **complemento para dois**. É o caso das instruções:

```
addi $3, $5, -4      # equivalente a 0xFFFFC  
addi $4, $2, 0x15    # 2110  
slti $6, $7, 0xFFFFF # -110
```

- Gama de representação da constante: **[-32768, +32767]**
- A constante de 16 bits é extendida para 32 bits, preservando o sinal (para a constante -4, o valor do operando é **0xFFFFFFF**)
- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Estão neste grupo todas as instruções lógicas:

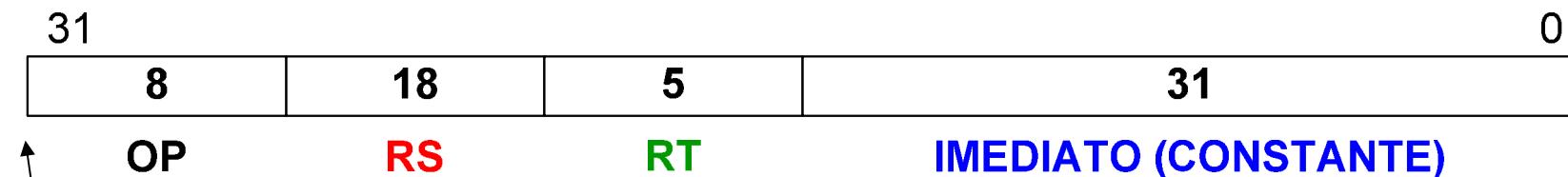
```
andi $3, $5, 0xFFFF
```

- Gama de representação da constante: **[0, 65535]**
- A constante de 16 bits é extendida para 32 bits, sendo os 16 mais significativos **0x0000** (para o exemplo: **0x0000FFFF**)



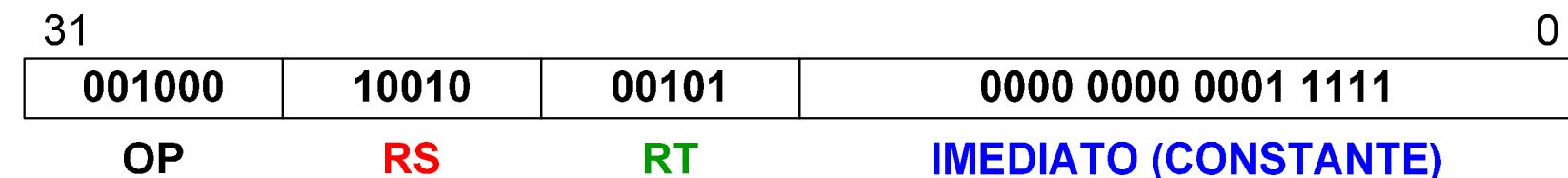
# Codificação das instruções que usam constantes

Exemplo: **addi \$5, \$18, 31**



Instrução do tipo I

***addi rt, rs, immediate***



**Cod. Máquina:** 001000**10010**00101**0000000000011111** = 0x2245001F



# Manipulação de constantes de 32 bits – LUI

- Em alguns casos pode ser necessário manipular constantes que necessitem de um espaço de armazenamento com mais do que 16 bits (como, por exemplo, a referência explícita a um endereço). Como lidar com esses casos?
- Para facilitar a manipulação de imediatos com mais de 16 bits, o ISA do MIPS inclui a seguinte instrução:

**lui        \$reg, immediate**

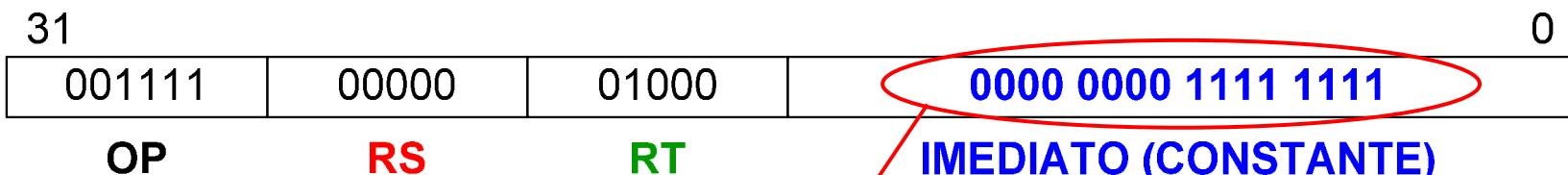
- A instrução **lui** ("Load Upper Immediate"), coloca a constante "immediate" nos **16 bits mais significativos do registo destino** (também é uma instrução do tipo I)
- Os 16 bits menos significativos ficam com **0x0000**



# Manipulação de constantes de 32 bits – LUI

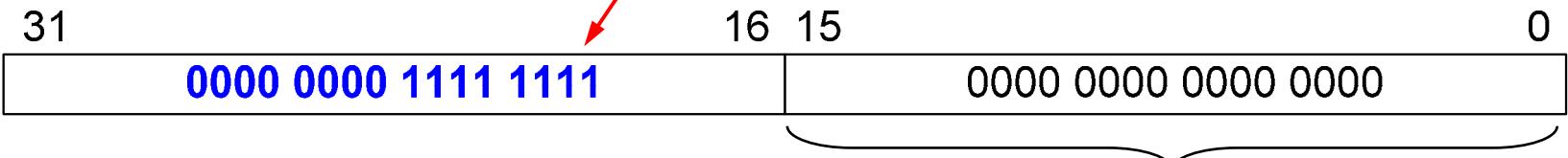
Exemplo:

**lui \$8, 255 #  $255_{10} = 0x00FF$**



***lui rt, immediate***

Conteúdo do registo \$8 após a execução da instrução:



**Valor que fica armazenado  
em \$8 = 0x00FF0000**

**Os 16 bits menos significativos  
ficam com o valor 0**



# Manipulação de constantes de 32 bits – LA / LI

A instrução virtual "load address"

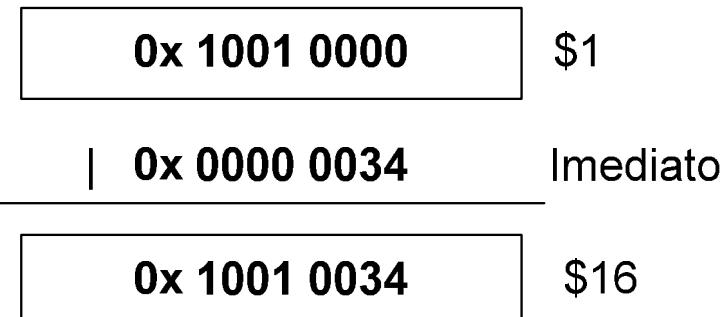
```
la $16, MyData # Ex. MyData = 0x10010034  
# Segmento de dados em 0x1001000
```

é executada no MIPS pela sequência de instruções nativas:

```
lui $1, 0x1001      # $1    = 0x10010000  
ori $16, $1, 0x0034 # $16   = 0x10010000 | 0x00000034
```

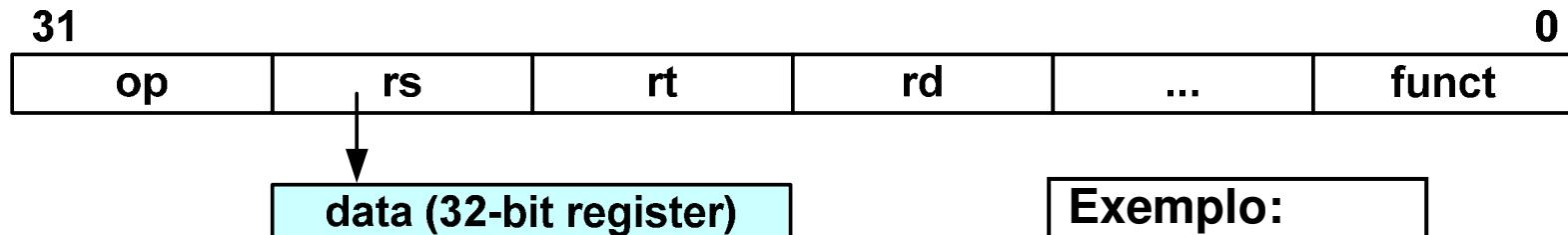
Notas:

- O registo \$1 (\$at) é reservado para o Assembler, para permitir este tipo de decomposição de instruções virtuais em instruções nativas.
- A instrução “li” (*load immediate*) é decomposta em instruções nativas de forma análoga à instrução “la”

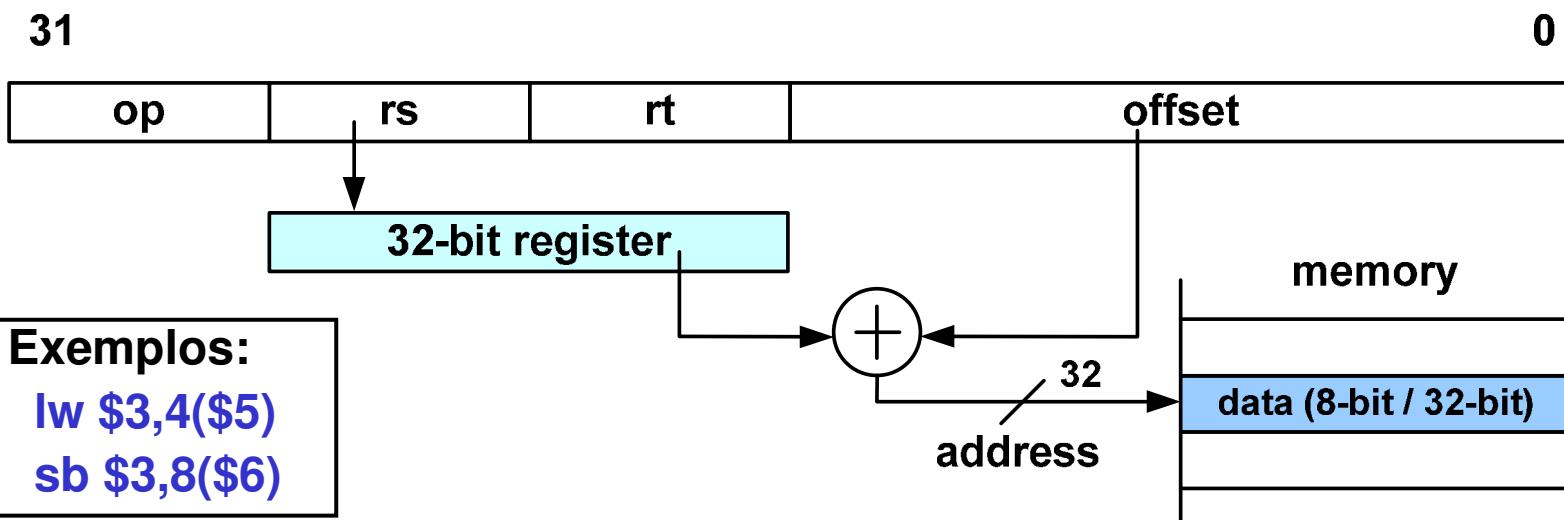


# Modos de endereçamento do MIPS (resumo)

- 1 - Register Addressing (endereçamento tipo registo):



- 2 - Base addressing (indireto por registo com deslocamento):



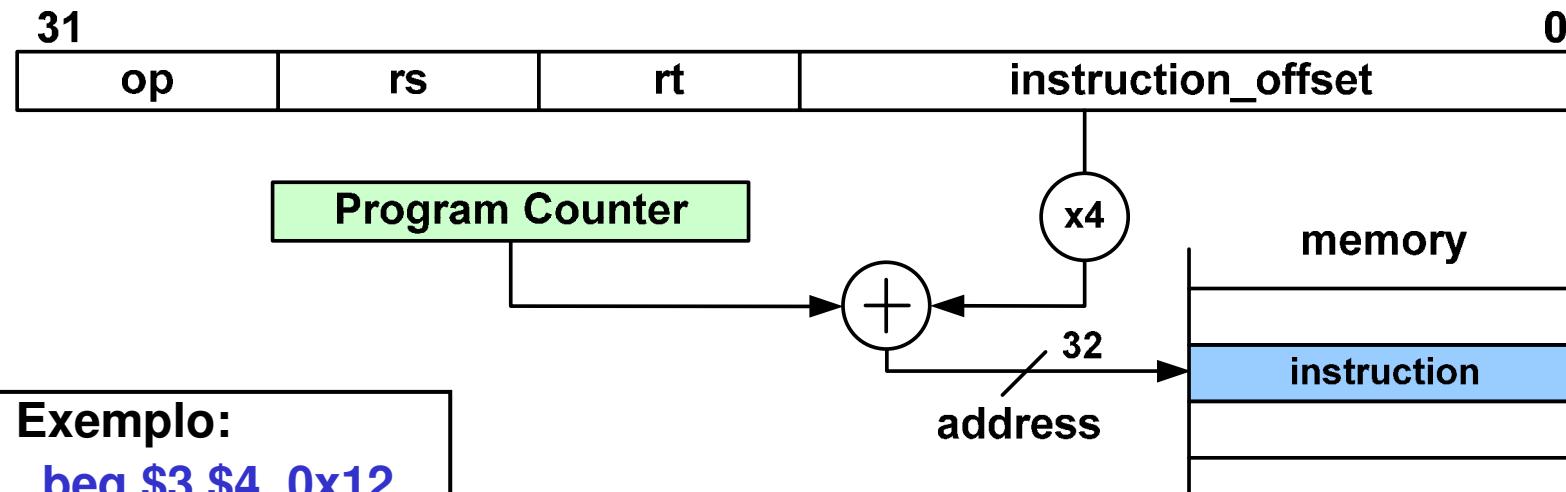
# Modos de endereçamento do MIPS (resumo)

- 3 - Immediate Addressing (endereçamento imediato):



Exemplo:  
addi \$3,\$4,0x3F

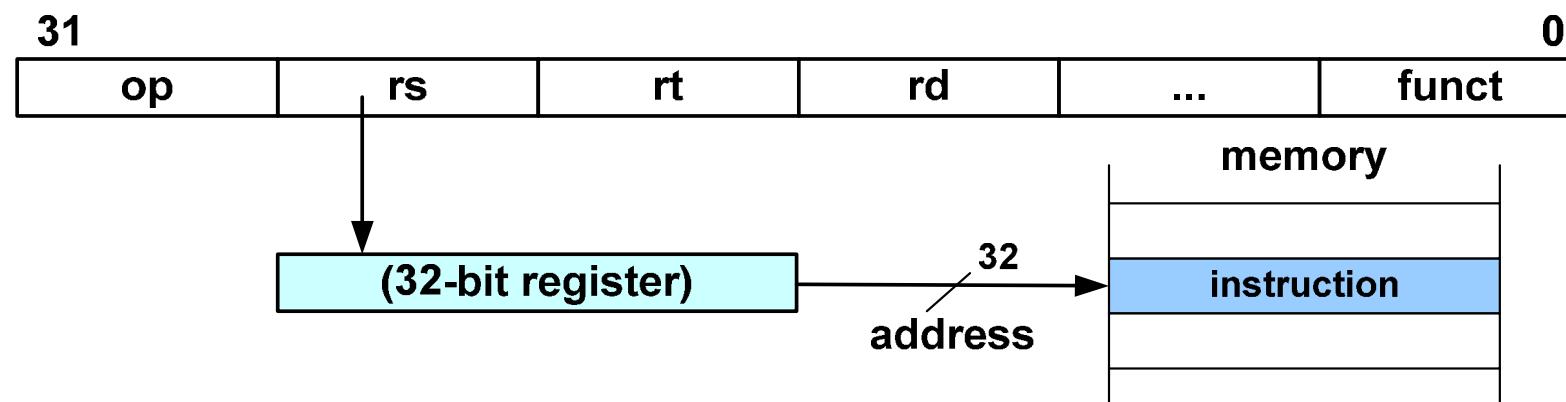
- 4 - PC-relative Addressing (endereçamento relativo ao PC):



Exemplo:  
beq \$3,\$4, 0x12

# Modos de endereçamento do MIPS (resumo)

- 5 – Indirect Register Addressing (endereçamento indireto por registo):



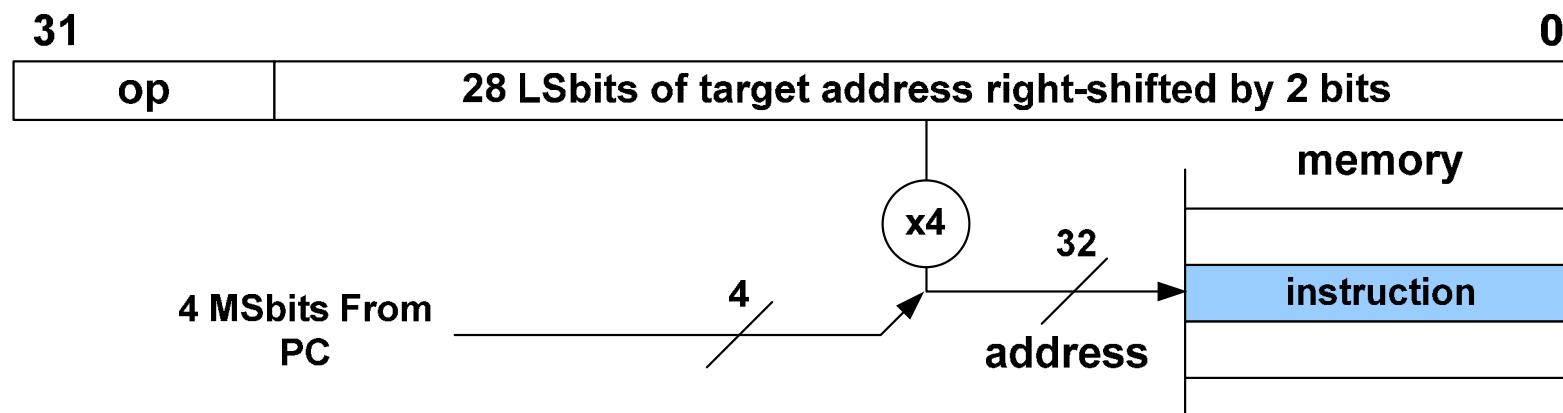
**Exemplo:**

**jr \$ra # target address is [\$ra]**



# Modos de endereçamento do MIPS (resumo)

- 6 - Pseudo-direct Addressing (endereçamento pseudo-direto):



**Exemplos:**

```
j 0x0010000B # target address is 0x0040002C  
jal 0x0010048E # target address is 0x00401238
```

(target calculado supondo que PC = 0x0...)



## Aula 7

- Sub-rotinas: evocação e retorno
- Caraterização das sub-rotinas na perspetiva do "chamador" e do "chamado"
- Convenções adotadas quanto à:
  - passagem de parâmetros para sub-rotinas
  - devolução de valores de sub-rotinas
  - salvaguarda de registos: "*caller-saved*" *versus* "*callee-saved*"

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



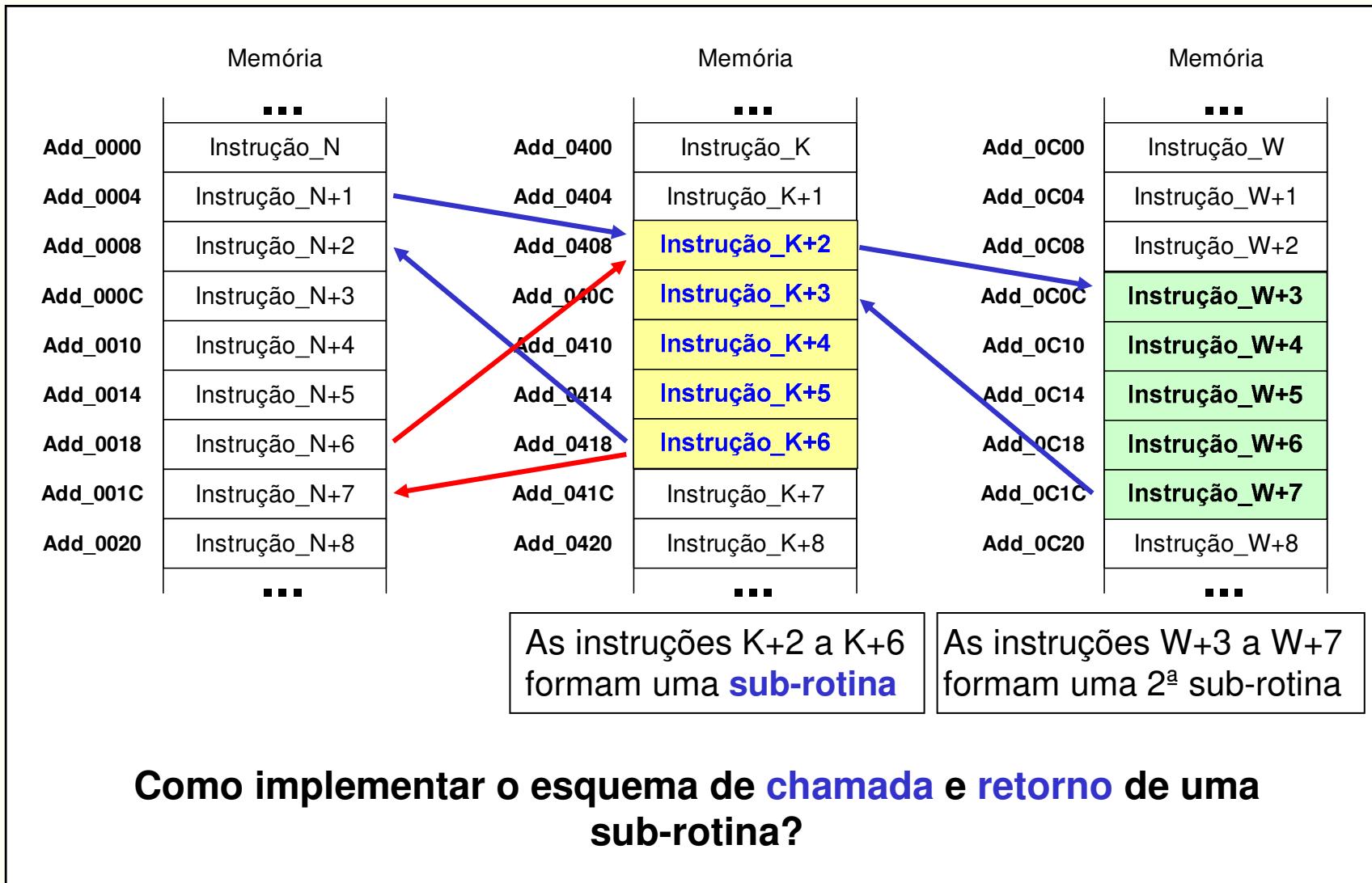
# Porque se usam funções (sub-rotinas)?

- Há três razões principais que justificam a existência de funções\*:
  - A **reutilização no contexto de um determinado programa** - aumento da eficiência na dimensão do código, substituindo a repetição de um mesmo trecho de código por um único trecho evocável de múltiplos pontos do programa
  - A **reutilização no contexto de um conjunto de programas**, permitindo que o mesmo código possa ser reaproveitado (bibliotecas de funções)
  - A **organização e estruturação do código**

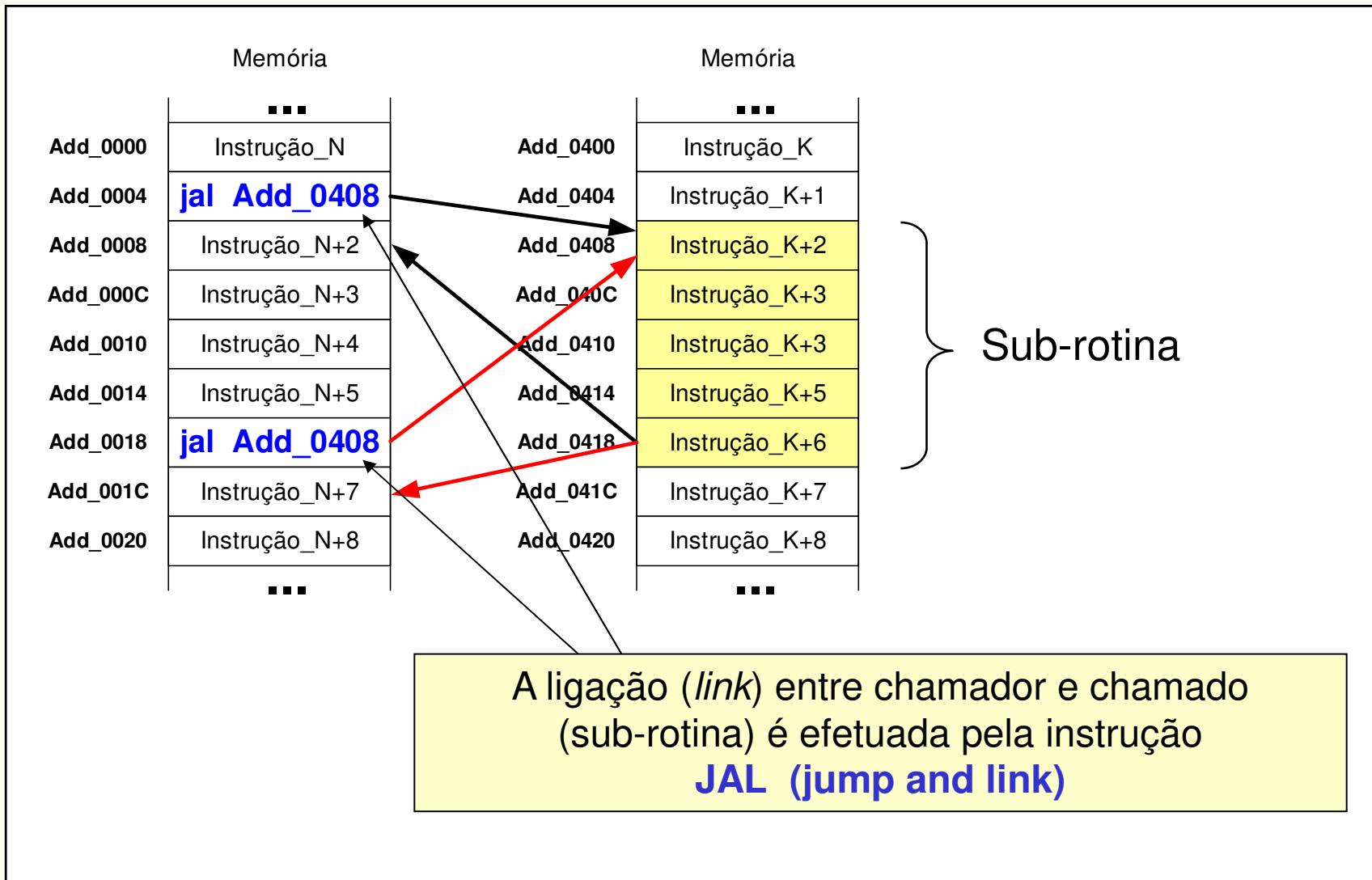
(\*) No contexto da linguagem *Assembly*, as funções e os procedimentos são genericamente conhecidas por **sub-rotinas!**



# Sub-rotinas: exemplo

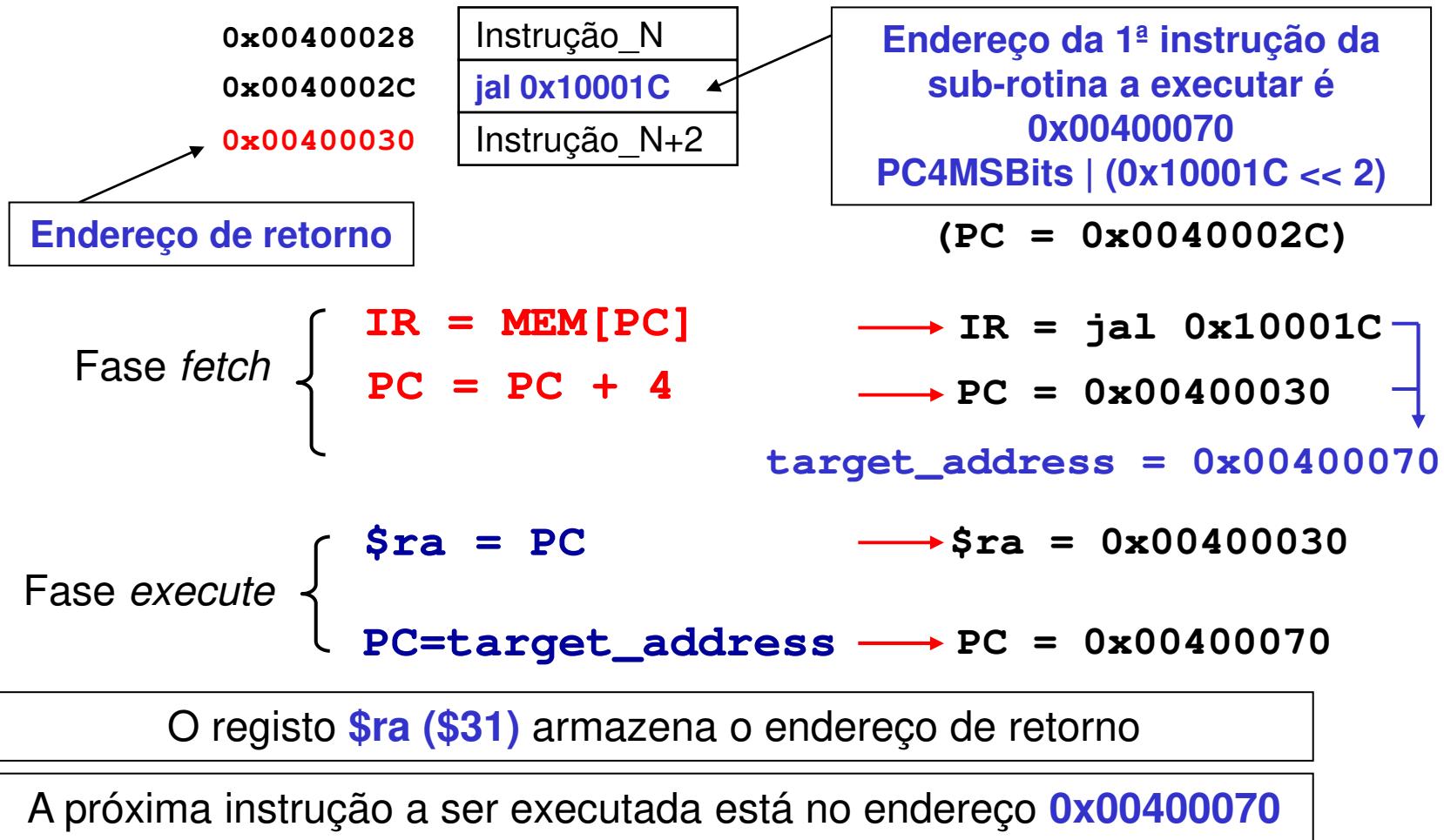


# Sub-rotinas: instrução JAL



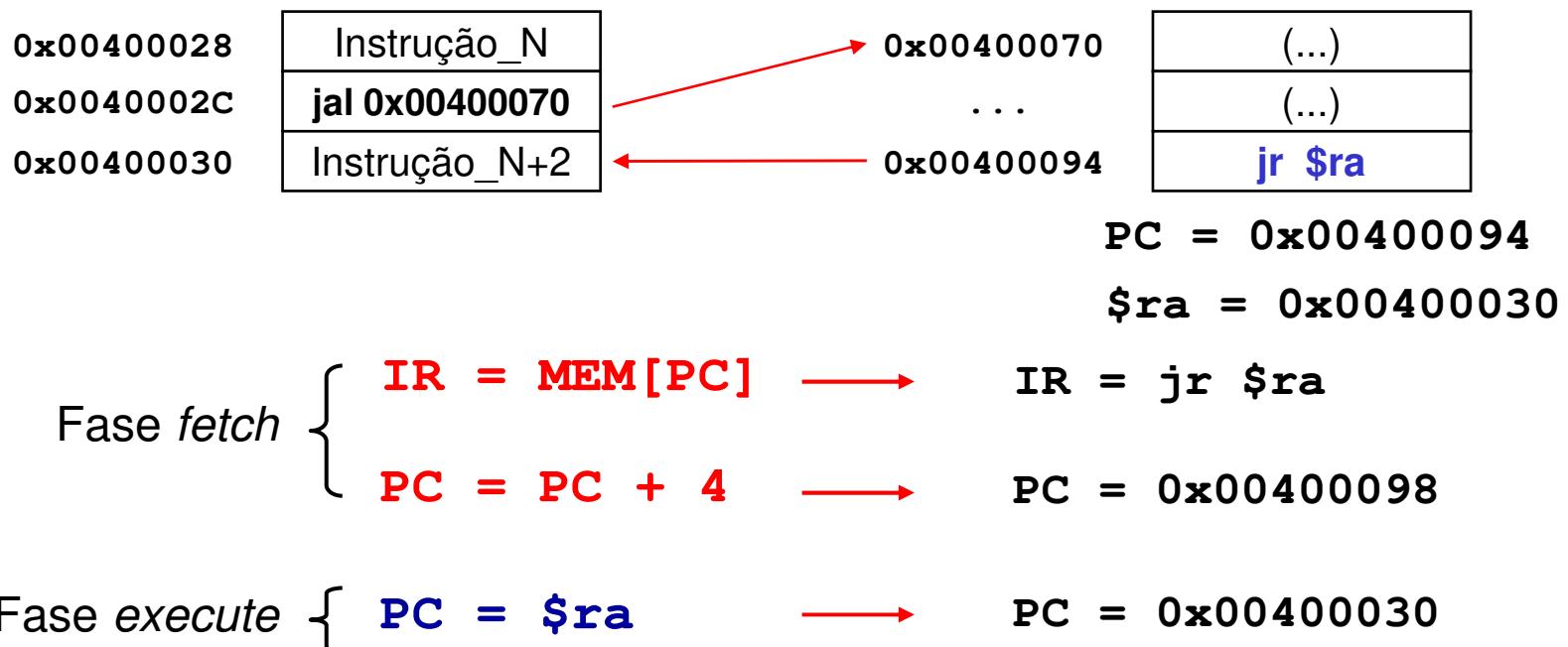
# Ciclo de execução da instrução JAL

- **jal target\_address**

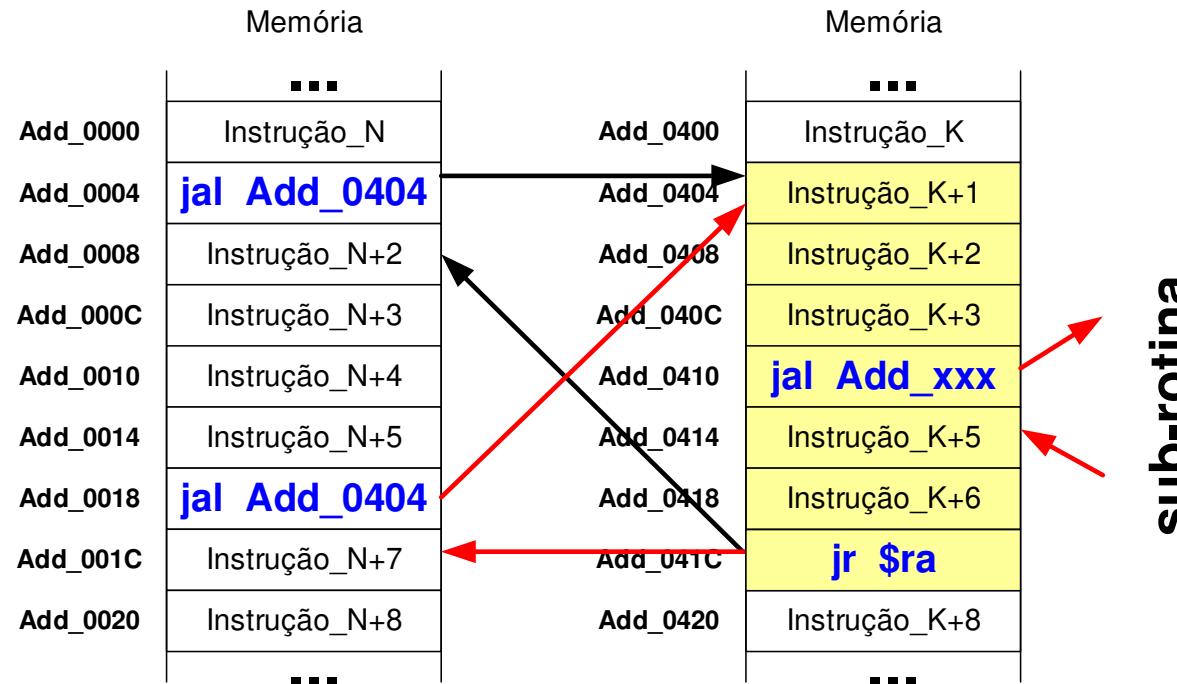


# Ciclo de execução da instrução JR

- Como **regressar** à instrução que sucede à instrução "jal" ?
- Aproveita-se o endereço de retorno armazenado em **\$ra** durante a execução da instrução "jal" (instrução "**jr register**")



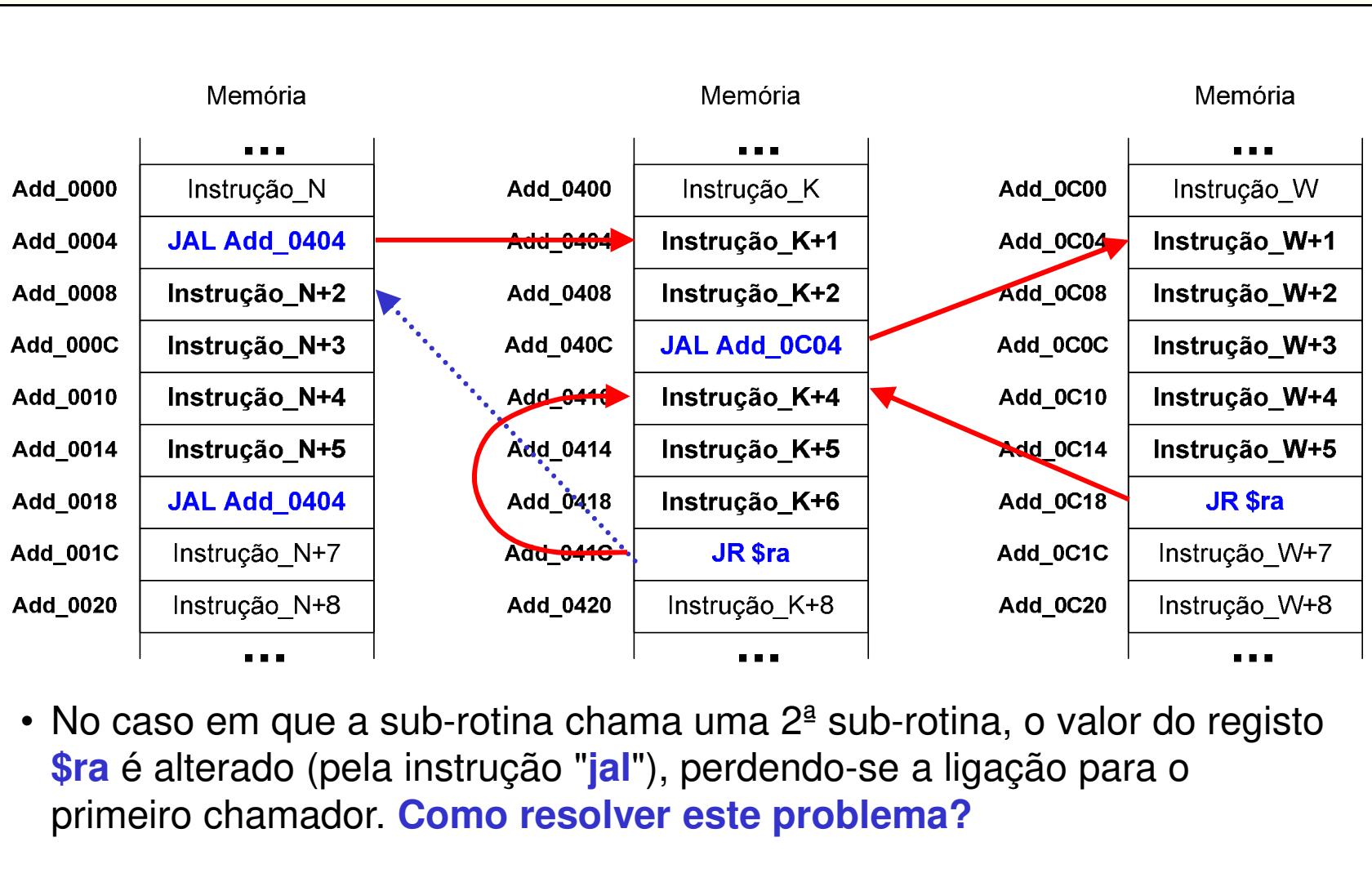
# Chamada a uma sub-rotina a partir de outra sub-rotina



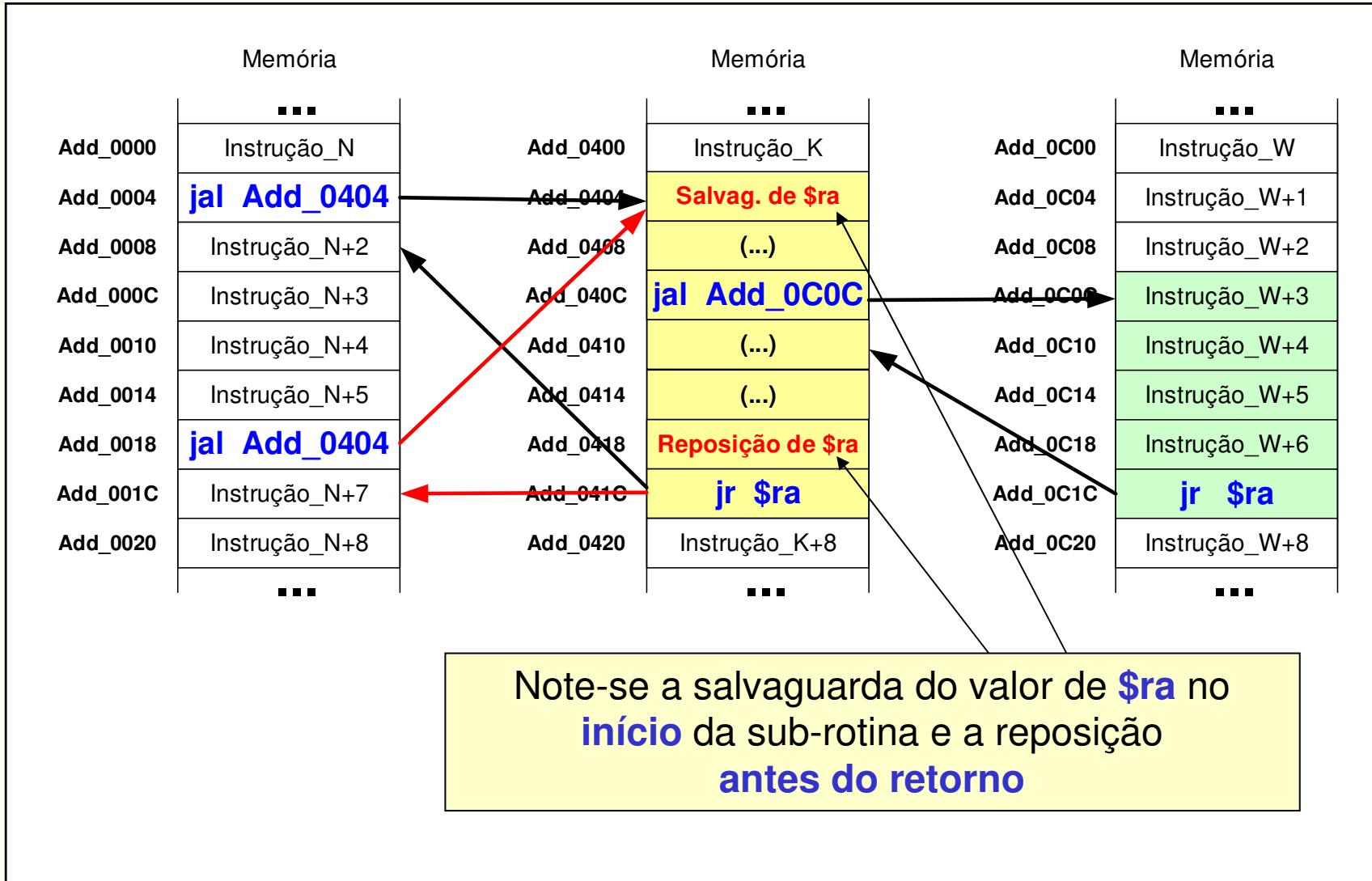
E se a sub-rotina (**Instrução\_K+1** a “**jr \$ra**”) chamar uma 2ª sub-rotina?



# Chamada a uma sub-rotina a partir de outra sub-rotina



# Chamada a uma sub-rotina a partir de outra sub-rotina



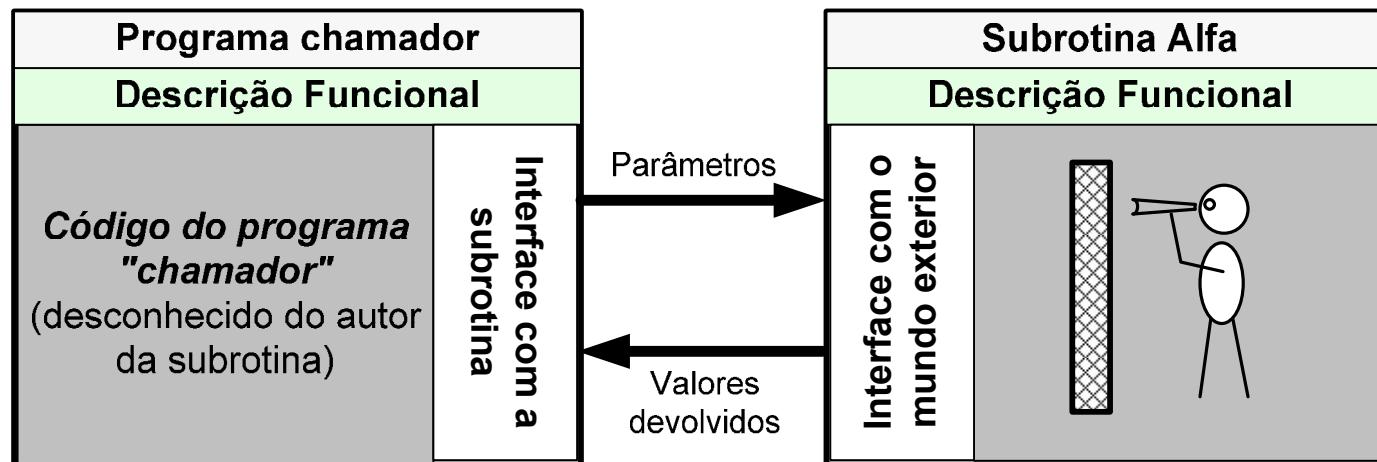
## Sub-rotinas – perspetiva do utilizador (programador)

- A **reutilização das sub-rotinas** torna-as particularmente atrativas, em especial quando suportam funcionalidades básicas, quer do ponto de vista computacional como do ponto de vista do interface entre o computador, os periféricos e o utilizador humano
- As sub-rotinas surgem, assim, frequentemente agrupadas em **bibliotecas**, a partir das quais podem ser evocadas por qualquer programa externo
- Este facto determina que o recurso a sub-rotinas escritas por outros para serviço dos nossos programas, **não deverá implicar necessariamente o conhecimento dos detalhes da sua implementação**
- Geralmente, o acesso ao código fonte da sub-rotina (conjunto de instruções originalmente escritas pelo programador) não é sequer possível, a menos que o mesmo seja tornado público pelo seu autor



# Sub-rotinas – perspetiva do programador

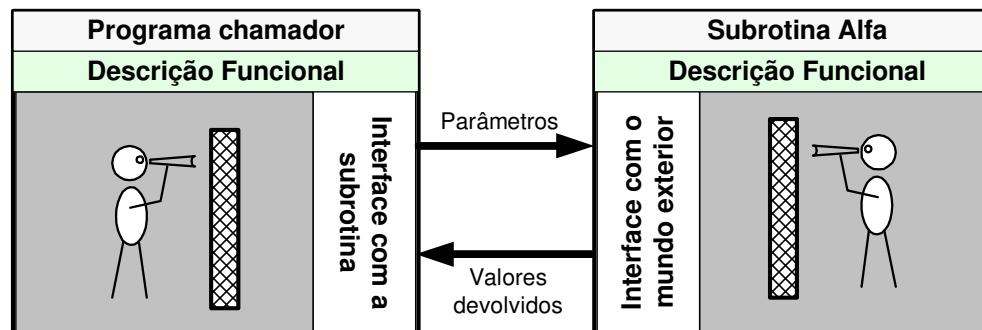
- Na perspetiva do programador, a sub-rotina que este tem a responsabilidade de escrever é um **trecho de código isolado**, com uma funcionalidade bem definida, e com um interface que ele próprio pode determinar em função das necessidades
- O facto de a sub-rotina ter de ser escrita para ser reutilizada implica que o programador não conhece antecipadamente as características do programa que irá evocar o seu código



# Regras a definir entre chamador e a sub-brotina chamada

- Torna-se assim óbvia a necessidade de definir um conjunto de **regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”**: a) definição do interface entre ambos e b) princípios que assegurem uma “sã convivência” entre os dois!

Exemplo



```
...
li      $t0, 0
11:    bge   $t0, 5, endf1
...
jal    sub1
...
addi   $t0, $t0, 1
j      11
endf1: ...
```

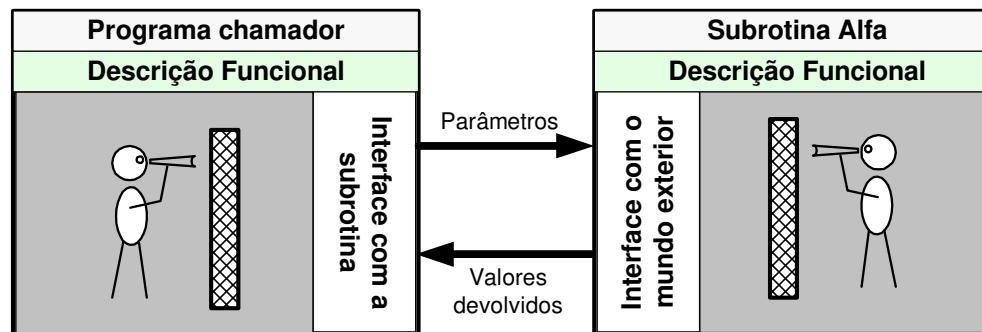
Quantas vezes vai ser executado o ciclo do programa chamador?



# Regras a definir entre chamador e a sub-brotina chamada

- Torna-se óbvia a necessidade de definir um conjunto de **regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”**: a) definição do interface entre ambos e b) princípios que assegurem uma “sã convivência” entre os dois!

Exemplo



```
...
li      $t0,0
11:    bge   $t0,5,endf1
...
jal    sub1
...
addi   $t0,$t0,-1
j      11
endf1: ...
```

```
sub1: li      $t0,3
12:   ble   $t0,0,endf2
...
addi   $t0,$t0,-1
j      12
endf2: jr   $ra
```

Quantas vezes vai ser executado o ciclo do programa chamador?



# Regras a definir entre chamador e a sub-brotina chamada

- Ao nível do interface:
  - Como **passar parâmetros** do “chamador” para o “chamado”, quantos e onde
  - Como **receber**, do lado do “chamador”, **valores devolvidos** pelo “chamado”
- Ao nível das regras de “sã convivência”:
  - Que registos do CPU podem “chamador” e “chamado” usar, sem que haja alteração indevida de informação (por exemplo um alterar o conteúdo de um registo que está simultaneamente a ser usado pelo outro)
  - Como partilhar a memória usada para armazenar dados, sem risco de sobreposição (e consequente perda de informação armazenada)



# Convenção para a passagem de parâmetros no MIPS

- Os parâmetros que possam ser armazenados na dimensão de um registo (32 bits, i.e., **char, int, ponteiros**) devem ser passados à sub-rotina nos registos **\$a0 a \$a3** (\$4 a \$7) por esta ordem
  - o **primeiro parâmetro sempre em \$a0**, o **segundo em \$a1** e assim sucessivamente
- Caso o número de parâmetros a passar nos registos  $\$a_i$  seja superior a quatro, os restantes (pela ordem em que são declarados) deverão ser passados na stack
- No caso de um ou mais parâmetros serem do **tipo float ou double**, os registos utilizados para os passar serão os registos **\$f12 e \$f14** do co-processador de vírgula flutuante (ponteiros são passados nos registos  $\$a_i$ )



# Convenção para a devolução de valores no MIPS

- A sub-rotina pode devolver um valor de 32 bits ou um de 64 bits:
  - Se o valor a devolver é de **32 bits** é utilizado o registo **\$v0**
  - Se o valor a devolver é de **64 bits**, são utilizados os registos **\$v1 (32 bits mais significativos) e \$v0 (32 bits menos significativos)**
- No caso de o valor a devolver ser do tipo **float ou double**, o registo a utilizar será o registo **\$f0** do co-processador de vírgula flutuante



# Exemplo (chamador)

```
int max(int, int);

void main(void)
{
    static int maxVal;
    maxVal = max(19, 35);
}
```

Em Assembly:

```
.data
maxVal:.space 4
.text
main: (...) # Salvag. $ra
    li      $a0, 19
    li      $a1, 35
    jal    max
    la      $t0, maxVal
    sw      $v0, 0($t0)
    (...) # Repõe $ra
    jr      $ra
```

Note-se que, para escrever o programa “chamador”, não é necessário conhecer os detalhes de implementação da sub-rotina

parâmetros

evocação da sub-rotina

valor devolvido



# Exemplo (sub-rotina)

```
int max(int a, int b)
{
    int vmax = a;

    if(b > vmax)
        vmax = b;
    return vmax;
}
```

Note-se que , para escrever o código da sub-rotina, não é necessário conhecer os detalhes de implementação do “chamador”

Em Assembly:

```
max: move $v0, $a0
      ble $a1, $v0, endif
      move $v0, $a1
endif: jr $ra
```

regresso ao chamador

parâmetros

Valor a devolver

Será necessário salvaguardar o valor de \$ra?



# Estratégias para a salvaguarda de registos

- Que registos pode usar uma sub-rotina, sem que se corra o risco de que os mesmos registos estejam a ser usados pelo programa “chamador”, potenciando assim a destruição de informação vital para a execução do programa como um todo?
- Uma hipótese seria dividir, de forma estática, os registos existentes entre “chamador” e “chamado”! Nesse caso, o que fazer quando o “chamado” é simultaneamente “chamador” (sub-rotina que chama outra sub-rotina)?
- Outra hipótese, mais praticável, consiste em atribuir a um dos “parceiros” a responsabilidade de copiar previamente para a memória externa o conteúdo de qualquer registo que pretenda utilizar (**salvaguardar o registo**) e repor, posteriormente, o valor original lá armazenado



# Estratégias para a salvaguarda de registos

- Estratégia “**caller-saved**”
  - Deixa-se ao cuidado do programa “chamador” a responsabilidade de salvaguardar o conteúdo da totalidade dos registos antes de evocar a sub-rotina
  - Cabe-lhe também a tarefa de repor posteriormente o seu valor
  - No limite, é admissível que o “chamador” salvaguarde apenas o conteúdo dos registos de que venha a precisar mais tarde
- Estratégia “**callee-saved**”
  - Entrega-se à sub-rotina a responsabilidade pela prévia salvaguarda dos registos de que possa necessitar
  - Assegura, igualmente, a tarefa de repor o seu valor imediatamente antes de regressar ao programa “chamador”



## Convenção para salvaguarda de registos no MIPS

- No caso do MIPS, a estratégia adotada é uma versão mista das anteriores, e baseia-se nas duas regras seguintes:
  - Os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** podem ser livremente utilizados e alterados pelas sub-rotinas
  - Os valores dos registos **\$s0..\$s7** não podem, **na perspetiva do chamador**, ser alterados pelas sub-rotinas
- Então, se os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** podem ser livremente utilizados e alterados pelas sub-rotinas
  - Um programa “chamador” que esteja a usar um ou mais destes registos, deverá salvaguardar o seu conteúdo antes de evocar uma sub-rotina, sob pena de que esta os venha a alterar



## Convenção para salvaguarda de registos no MIPS

- Os valores dos registos **\$s0..\$s7** não podem, na perspetiva do chamador, ser alterados pelas sub-rotinas
  - Se uma dada sub-rotina precisar de usar um registo do tipo **\$sn**, compete a essa sub-rotina **copiar previamente o seu conteúdo** para um lugar seguro (memória externa), repondo-o imediatamente antes de terminar
  - Dessa forma, do ponto de vista do programa “chamador” (que não “vê” o código da sub-rotina) é como se esse registo não tivesse sido usado ou alterado



# Considerações práticas sobre a utilização da convenção

- **sub-rotinas terminais** (sub-rotinas folha, i.e., que não chamam qualquer sub-rotina)
  - Só devem utilizar (preferencialmente) registo que não necessitam de ser salvaguardados (**\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3**)
- **sub-rotinas que chamam outras sub-rotinas**
  - Devem utilizar os registos **\$s0..\$s7** para o armazenamento de valores que se pretenda preservar. A utilização destes registos implica a sua prévia salvaguarda na memória externa logo no início da sub-rotina e a respetiva reposição no final
  - Devem utilizar os registos **\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3** para os restantes valores



# Utilização da convenção - exemplo

- O problema detetado na codificação do programa chamador e da sub-rotina dos slides 12 e 13 pode facilmente ser resolvido se a convenção de salvaguarda de registos for aplicada
- A variável índice do ciclo do programa chamador passará a residir num registo **\$sn** (por exemplo no \$s0) – registo que, **garantidamente**, a sub-rotina não vai alterar

O código da subrotina é desconhecido do programador  
do “programa chamador” e vice-versa

```
(...) # Salv. $s0
...
li      $s0, 0
11:    bge   $s0, 5, endf1
...
jal    sub1
...
addi   $s0, $s0, 1
j     11
endf1: ...
(...) # Repoe $s0
```

```
sub1:  li      $t0, 3
12:    ble   $t0, 0, endf2
...
addi   $t0, $t0, -1
j     12
endf2: jr    $ra
```

Quantas vezes vai ser executado o  
ciclo do programa chamador?



## Aula 8

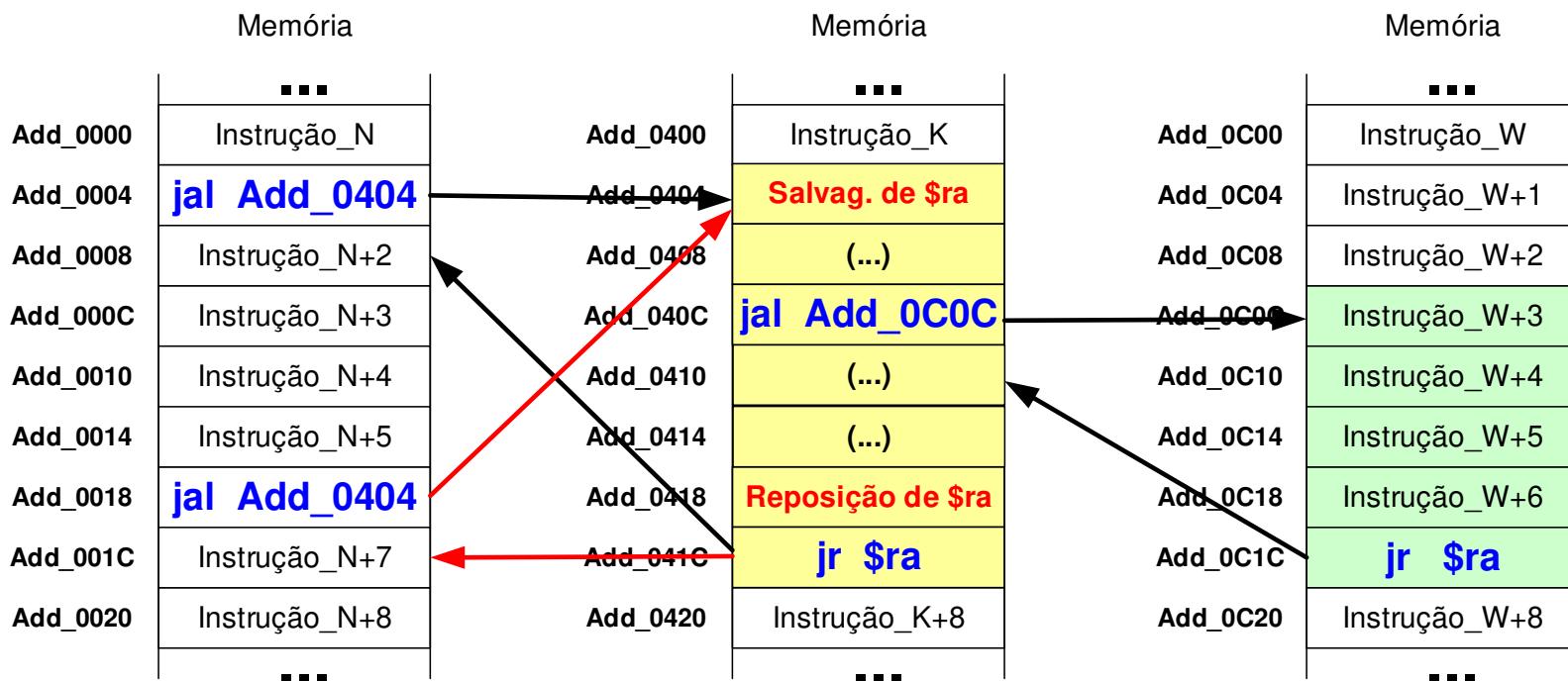
- Utilização de *stacks*
- Conceito e regras básicas de utilização
- Utilização da *stack* nas arquiteturas MIPS
- Recursividade
- Análise de um exemplo, incluindo uma sub-rotina recursiva

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

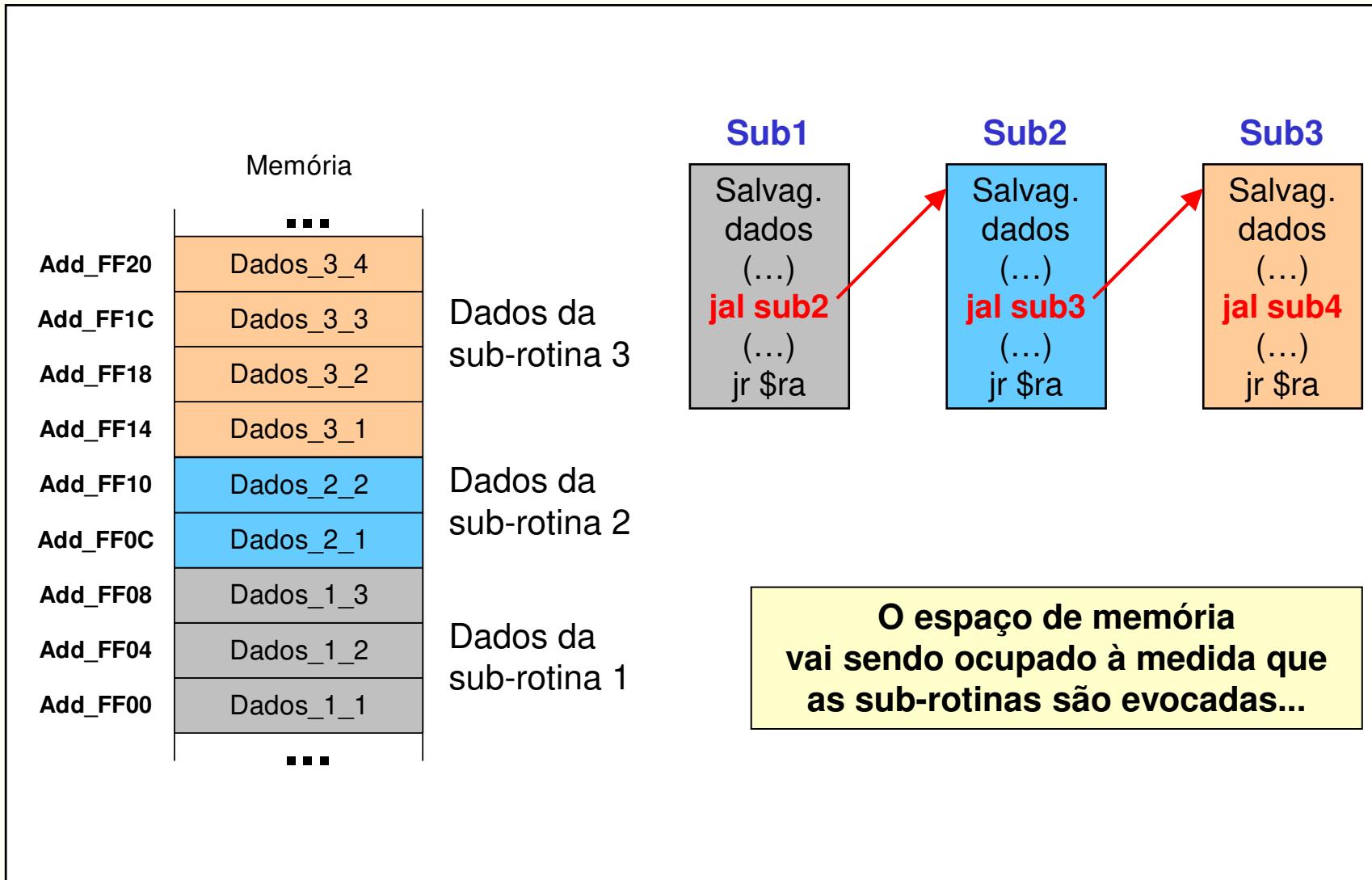


# Armazenamento temporário de informação

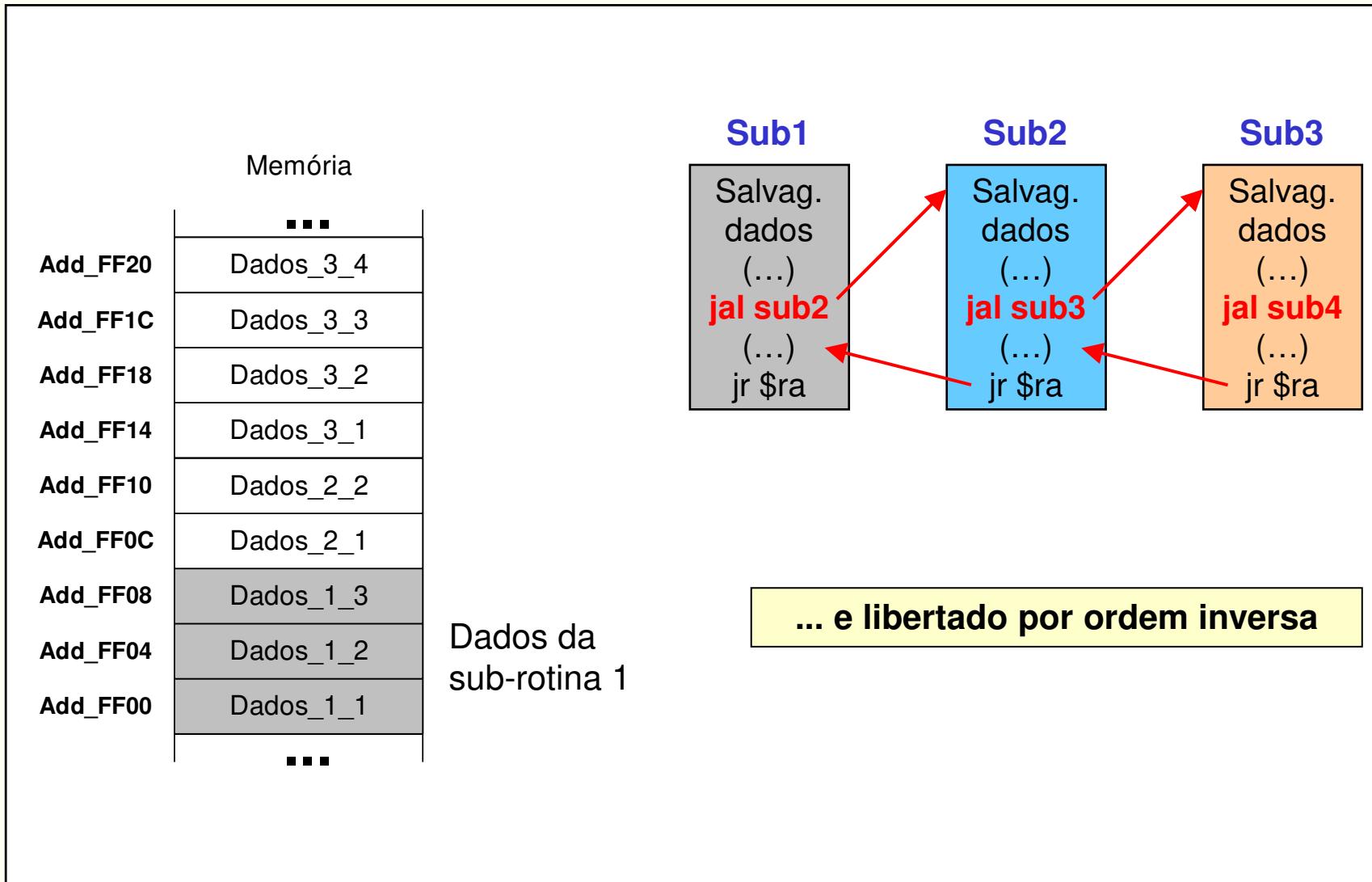
Como poderemos garantir que os dados, residentes em memória e manipulados por cada sub-rotina não interferem com os dados das restantes?



# Stack: espaço de armazenamento temporário



# Stack: espaço de armazenamento temporário



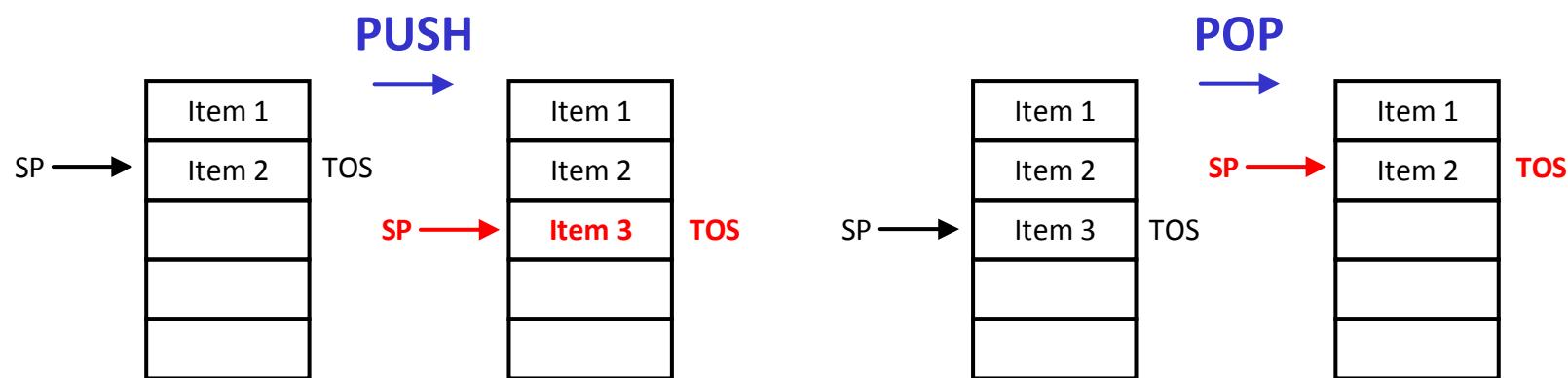
# *Stack: espaço de armazenamento temporário*

- A estratégia de gestão dinâmica do espaço de memória - em que a última informação acrescentada é a primeira a ser retirada – é designada por **LIFO** (*Last In First Out*)
- A estrutura de dados correspondente é conhecida por “pilha” - **STACK**
- As *stacks* são de tal forma importantes que a maioria das arquiteturas suportam diretamente instruções específicas para manipulação de *stacks* (por exemplo a x86)
- A operação que permite acrescentar informação à *stack* é normalmente designada por **PUSH**, enquanto que a operação inversa é conhecida por **POP**



# Stack: operações *push* e *pop*

- Estas operações têm associado um registo designado por **Stack Pointer (SP)**
- O registo **Stack Pointer** mantém, de forma permanente, o **endereço do topo da stack (TOS - top of stack)** e aponta sempre para o último endereço ocupado
  - Numa operação de **PUSH** é necessário pré-atualizar o *stack pointer* antes de uma nova operação de escrita na *stack*
  - Numa operação de **POP** é feita uma leitura da *stack* seguida de atualização do *stack pointer*

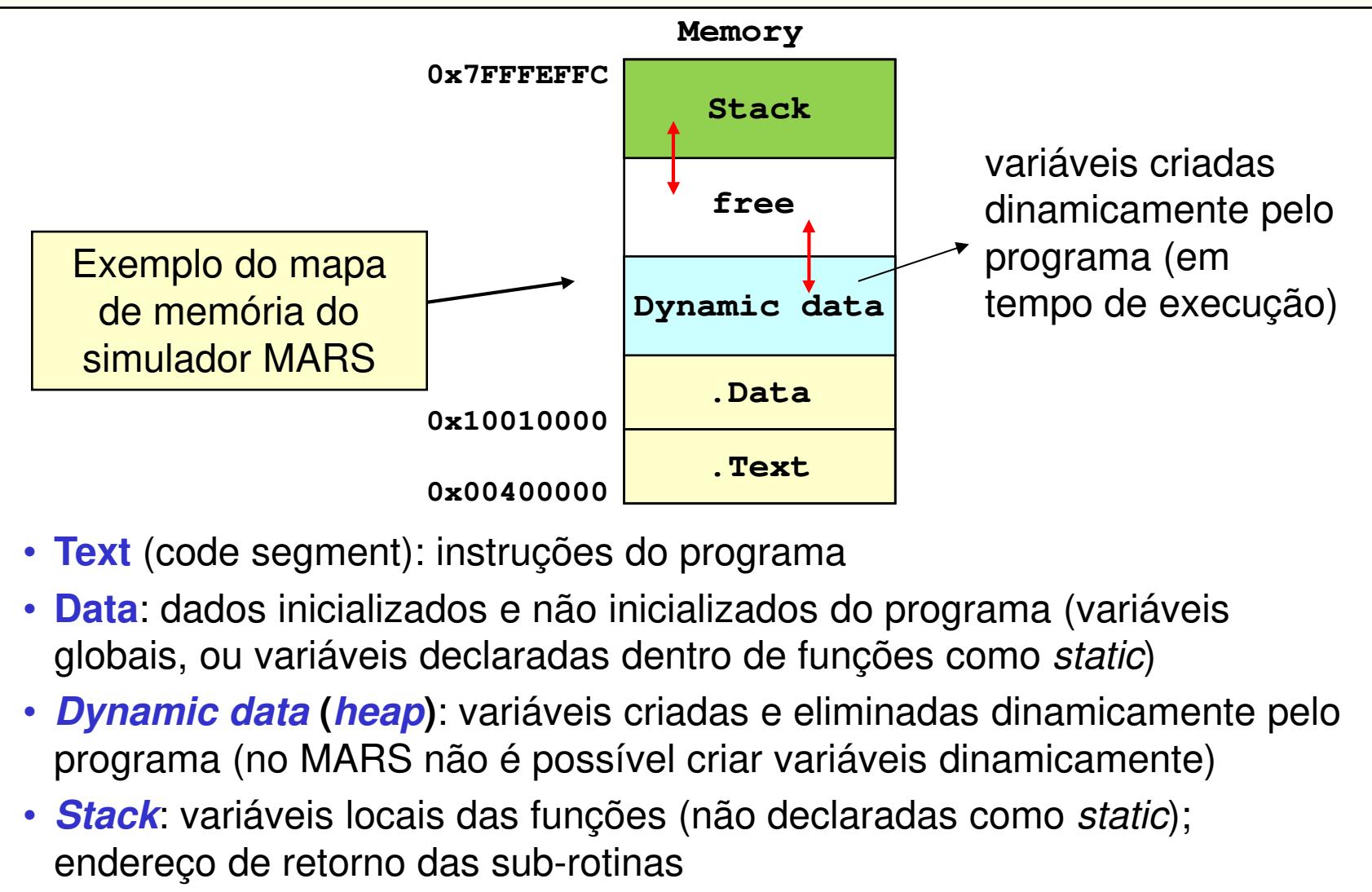


# Atualização do *stack pointer*

- A atualização do ***stack pointer***, durante a fase de escrita de informação, pode seguir uma de duas estratégias:
  - Ser incrementado, fazendo crescer a *stack* no sentido crescente dos endereços
  - Ser decrementado, fazendo crescer a *stack* no sentido decrescente dos endereços
- A estratégia de crescimento da *stack* no sentido dos endereços mais baixos é, geralmente, a adotada
- Esta estratégia (de crescimento da *stack* no sentido dos endereços mais baixos) permite uma gestão simplificada da fronteira entre os segmentos de dados e de *stack*



# Atualização do *stack pointer*



# Regras de utilização da stack na arquitetura MIPS

1. O registo **\$sp** (*stack pointer*) contém o endereço da **última posição ocupada** da stack

2. A stack **cresce** no **sentido decrescente** dos endereços da memória

$$\$sp = \$29$$



# Regras de utilização da stack na arquitetura MIPS

- Exemplo

```
lab: addiu $sp, $sp, -16 # Reserva espaço na stack
```

```
sw    $ra, 0($sp)   # Copia registos  
sw    $s0, 4($sp)   # $ra, $s0, $s1  
sw    $s1, 8($sp)   # e $s2 para a  
sw    $s2, 12($sp)  # stack
```

```
(...)          # Código da sub-rotina
```

```
lw    $ra, 0($sp)   # Repõe o valor  
lw    $s0, 4($sp)   # dos registos  
lw    $s1, 8($sp)   # $ra,  
lw    $s2, 12($sp)  # $s0, $s1 e $s2
```

```
addiu $sp, $sp, 16 # Liberta espaço na  
# stack
```

\$SP

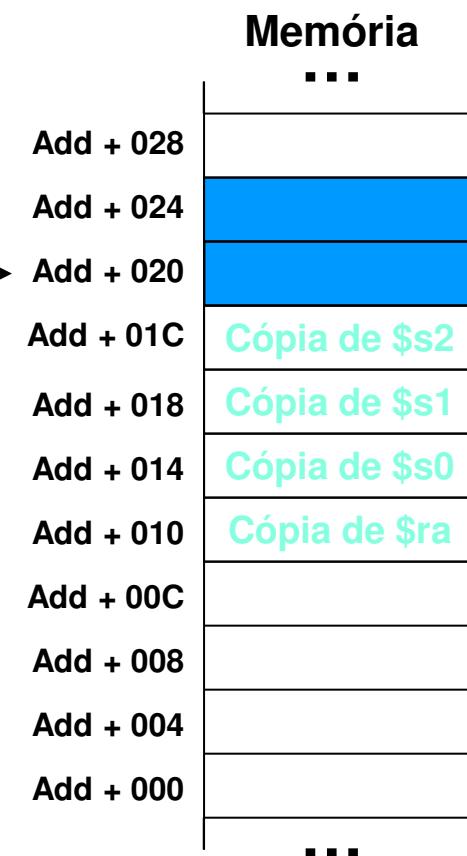
\$SP



# Regras de utilização da stack na arquitetura MIPS

- Exemplo

```
lab: addiu $sp, $sp, -16 # Reserva espaço na stack  
      sw    $ra, 0($sp)   # Copia registos  
      sw    $s0, 4($sp)   # $ra, $s0, $s1  
      sw    $s1, 8($sp)   # e $s2 para a  
      sw    $s2, 12($sp)  # stack  
  
      (...)           # Código da sub-rotina  
      lw    $ra, 0($sp)   # Repõe o valor  
      lw    $s0, 4($sp)   # dos registos  
      lw    $s1, 8($sp)   # $ra,  
      lw    $s2, 12($sp)  # $s0, $s1 e $s2  
  
      addiu $sp, $sp, 16 # Liberta espaço na  
                          # stack
```



# Análise de um exemplo completo

Considere-se o seguinte código C:

```
int soma(int *, int);

void main(void)
{
    static int array[100]; // reside em memória
    int result;
    ...
    // código de inicialização do array
    result = soma(array, 100);
    print_int10(result); // syscall
}
```

Declaração de um *array static*  
(reside no “data segment”)

Declaração de uma variável  
inteira (pode residir num registo  
interno)

Afixação do resultado  
no ecrã

Evocação de uma função e  
atribuição do valor devolvido à  
variável inteira



# Código correspondente em Assembly do MIPS

```
# $t0 > variável "result"
#
        .data
array: .space 400          # Reserva de espaço p/ o array
                           # (100 words => 400 bytes)
        .eqv    print_int, 1   #
        .text
        .globl  main
main:  addiu  $sp, $sp, -4  # Reserva espaço na stack
        sw     $ra, 0($sp)   # Salvaguarda o registo $ra
        la     $a0, array     # inicialização dos registos
        li     $a1, 100       # que vão passar os parâmetros
        jal    soma          # soma(array, 100)
        move   $t0, $v0       # result = soma(array, 100)
        move   $a0, $t0       #
        li     $v0, print_int
        syscall            # print_int(result)
        lw     $ra, 0($sp)   # Recupera o valor do reg. $ra
        addiu $sp, $sp, 4    # Liberta espaço na stack
        jr     $ra             # Retorno
```

```
void main(void) {
    static int array[100];
    int result;
    result = soma(array, 100);
    print_int(result);
}
```

## Código da função soma()

```
int soma (int *array, int nelem)
{
    int n, res;
    for (n = 0, res = 0; n < nelem; n++)
    {
        res = res + array[n];
    }
    return res;
}
```

## A mesma função usando ponteiros:

```
int soma (int *array, int nelem)
{
    int res = 0;
    int *p = array;
    for (; p < &(array[nelem]); p++) // ou: ; p < (array + nelem);
    {
        res += (*p);
    }
    return res;
}
```

Esta função recebe dois parâmetros (um ponteiro para inteiro e um inteiro) e calcula o seguinte resultado:

$$res = \sum_{n=0}^{nelem-1} (array[n])$$

# Código correspondente em *Assembly* do MIPS

- Versão com ponteiros

```
# $t1 > p
# $v0 > res
#
soma: li      $v0, 0          # res = 0;
      move   $t1, $a0          # p = array;
      sll    $a1, $a1, 2        # nelem *= 4;
      addu   $a0, $a0, $a1        # $a0 = array + nelem;
for:   bgeu   $t1, $a0, endf     # while(p < &(array[nelem])){
      lw     $t2, 0($t1)        #
      add    $v0, $v0, $t2        #     res = res + (*p);
      addiu  $t1, $t1, 4         #     p++;
      j     for                  # }
endf:  jr    $ra                # return res;
```

```
int soma (int *array, int nelem)
{
    int res = 0;
    int *p = array;
    for (; p < &(array[nelem]); p++)
        res += (*p);
    return res;
}
```

A sub-rotina não evoca nenhuma outra e não são usados registos \$Sn, pelo que não é necessário salvaguardar qualquer registo

# Exemplo – função para cálculo da média

```
int media (int *array, int nelem)
{
    int res;
    res = soma(array, nelem);
    return res / nelem;
}
```

chama função soma()

Valor de *nelem* é necessário depois  
de chamada a função “soma”!

```
# res > $t0, array > $a0, nelem > $a1
media: addiu $sp,$sp,-8      # Reserva espaço na stack
        sw    $ra,0($sp)      # salvaguarda $ra
        sw    $s0,4($sp)      # guarda valor $s0 antes de o usar
        move $s0,$a1          # nelem é necessário depois
                                # da chamada à função soma
        jal   soma            # soma(array,nelem);
        move $t0,$v0          # res = retorno de soma()
        div   $v0,$t0,$s0      # res/nelem
        lw    $ra,0($sp)      # recupera valor de $ra
        lw    $s0,4($sp)      # e $s0
        addiu $sp,$sp,8       # Liberta espaço na stack
        jr   $ra              # retorna
```

# Recursividade – função soma()

$$\text{res} = \sum_{n=0}^{\text{nelem}-1} (\text{array}[n])$$

O resultado do somatório pode também ser obtido da seguinte forma:

$$\text{res} = \text{array}[0] + \sum_{n=1}^{\text{nelem}-1} (\text{array}[n])$$

$$\text{array}[1] + \sum_{n=2}^{\text{nelem}-1} (\text{array}[n])$$

$$\text{array}[2] + \sum_{n=3}^{\text{nelem}-1} (\text{array}[n])$$

(...)

$$\text{array}[\text{nelem} - 1]$$

$$\sum_{n=i}^{\text{nelem}-1} (\text{array}[n]) = \text{array}[i] + \sum_{n=i+1}^{\text{nelem}-1} (\text{array}[n])$$



# Recursividade – função soma()

$$res = \sum_{n=0}^{nelem-1} (\text{array}[n])$$

**res = soma (array, 0, nelem);**



$$\text{array}[0] + \sum_{n=1}^{nelem-1} (\text{array}[n])$$

**array[0] + soma (array, 1, nelem);**



$$\text{array}[1] + \sum_{n=2}^{nelem-1} (\text{array}[n])$$

**array[1] + soma (array, 2, nelem);**

```
int soma(int *array, int i, int nelem);
```

```
int soma(int *array, int i, int nelem)
{
    return array[i] + soma(array, i+1, nelem);
}
```

**O que falta nesta função?**

$$\sum_{n=i}^{nelem-1} (\text{array}[n]) = \text{array}[i] + \sum_{n=i+1}^{nelem-1} (\text{array}[n])$$



# Recursividade – função soma()

$$\sum_{n=i}^{nelem-1} (\text{array}[n]) = \text{array}[i] + \sum_{n=i+1}^{nelem-1} (\text{array}[n])$$

A função **soma()** pode, assim, ser escrita de forma **recursiva**:

O valor devolvido é posteriormente adicionado com o valor armazenado na posição **i** do *array*

```
int soma_rec (int *array, int i, int nelem)
{
    if (i != nelem)
        return array[i] + soma_rec (array, i + 1, nelem);
    else
        return 0;
}
```

**A função evoca-se a si mesma**, passando como primeiro parâmetro o endereço do início do *array*, como segundo parâmetro o índice a partir do qual se pretende obter a soma e como terceiro parâmetro o número de elementos do *array*



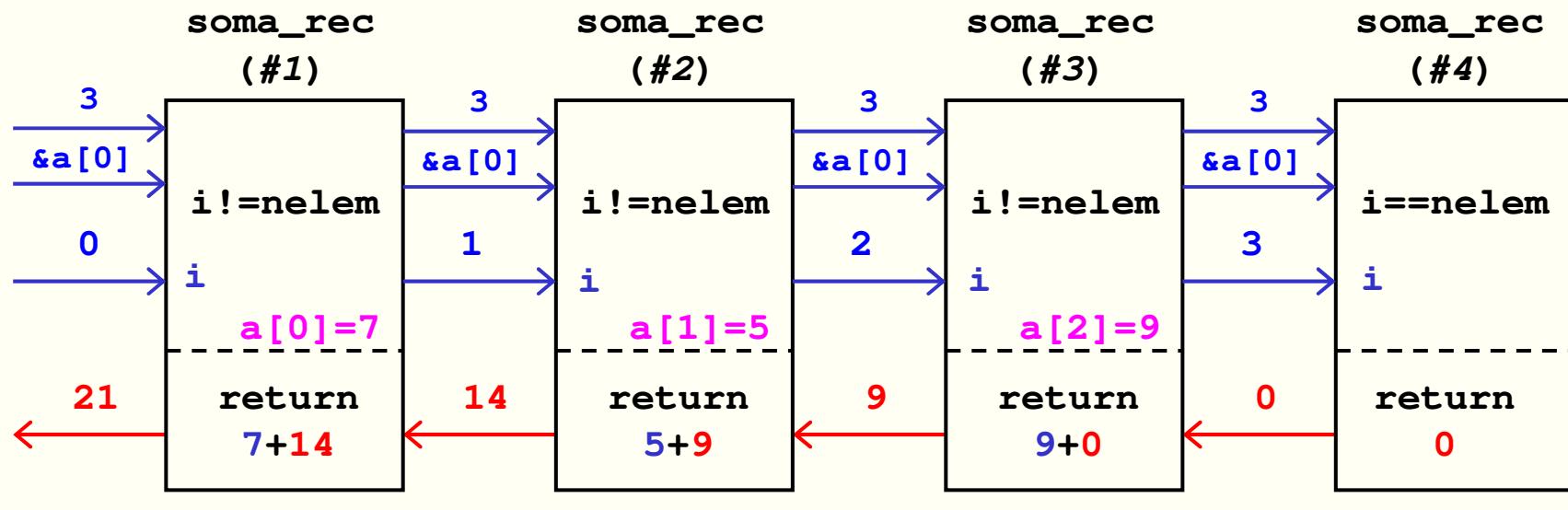
# Recursividade – função soma()

```
int soma_rec (int *array, int i, int nelem)
{
    if (i != nelem)
        return array[i] + soma_rec (array, i + 1, nelem);
    else
        return 0;
}
```

**Exemplo:**

Nº elementos do array “a”: 3

Array inicializado com:  $a[0]=7$ ,  $a[1]=5$ ,  $a[2]=9$



# Recursividade – função soma()

```
int soma_rec (int *array, int i, int nelem)
{
    if (i != nelem)
        return array[i] + soma_rec (array, i + 1, nelem);
    else
        return 0;
}
```

A função **soma\_rec()** pode ser simplificada, utilizando um **ponteiro para a posição do array** a partir da qual se pretende obter a soma (em vez do índice) e o **número de elementos do array que falta visitar** (em vez do número total de elementos).

```
int soma_rec (int *array, int nelem)
{
    if (nelem != 0)
        return *array + soma_rec (array + 1, nelem - 1);
    else
        return 0;
}
```

O segundo parâmetro representa o **número de elementos do array ainda não visitados**



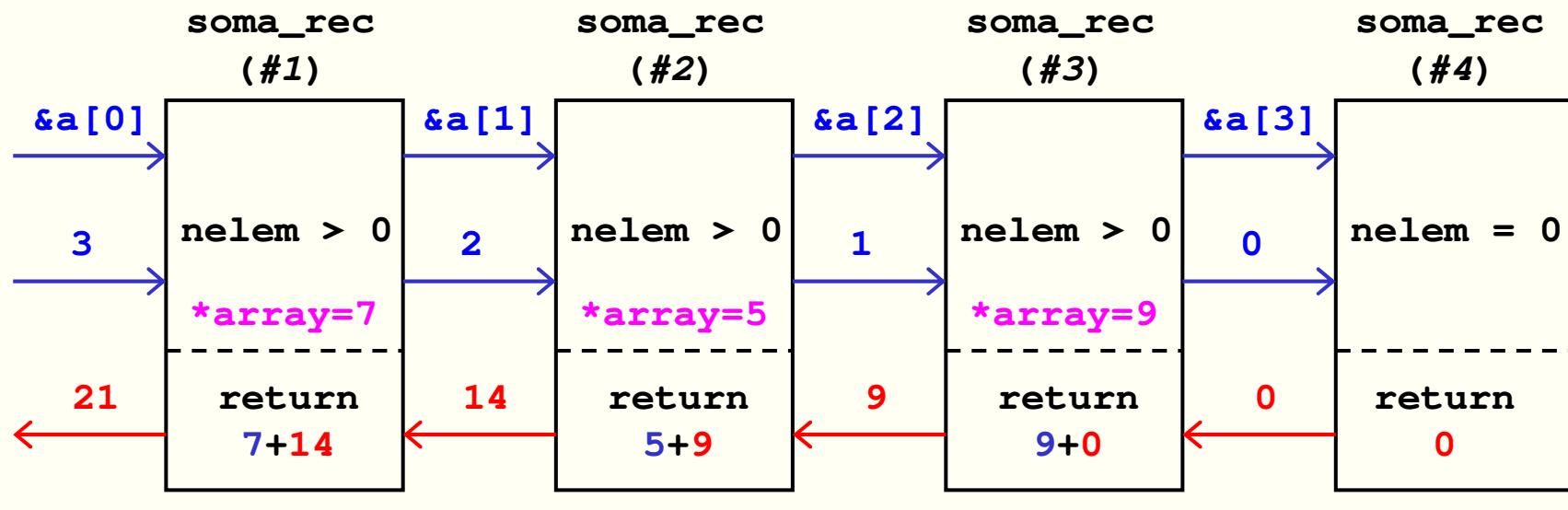
# Recursividade – função soma()

```
int soma_rec (int *array, int nelem)
{
    if (nelem != 0)
        return *array + soma_rec (array + 1, nelem - 1);
    else
        return 0;
}
```

Exemplo:

Nº elementos do array “a”: 3

Array inicializado com:  $a[0]=7$ ,  $a[1]=5$ ,  $a[2]=9$



## Código correspondente em Assembly do MIPS

```
int soma_rec (int *array, int nelem)
{
    if (nelem != 0)
        return *array+soma_rec(array+1, nelem-1);
    else
        return 0;
}
```

soma\_rec:

```
beq    $a1, $0, else # if (nelem != 0) {
addiu $sp, $sp, -8   # stack allocation
sw    $ra, 0($sp)    # save $ra
sw    $s0, 4($sp)    # save $s0
move  $s0, $a0         # $s0 = array
addiu $a0, $a0, 4      # array + 1;
sub   $a1, $a1, 1      # nelem=nelem-1;
jal   soma_rec        # soma_rec(array+1, nelem-1);
lw    $t0, 0($s0)      # aux = *array;
add   $v0, $v0, $t0      # val = val + aux;
lw    $ra, 0($sp)      # restore $ra
lw    $s0, 4($sp)      # restore $s0
addiu $sp, $sp, 8       # free stack
jr   $ra                 # return val;
                           # }
                           # else {
li    $v0, 0               #
jr   $ra                 # return 0; }
```

Salvag. **\$ra** (a sub-rotina não é terminal)

**array** é necessário depois da chamada à sub-rotina (cópia para **\$s0**)

O **stack pointer** tem obrigatoriamente que ser **atualizado antes de terminar a sub-rotina**

## Aula 9

- Representação de números inteiros com sinal (revisão)
  - Sinal e módulo
  - Complemento para um
  - Complemento para dois
- Exemplos de operações aritméticas
- *Overflow* e mecanismos para a sua deteção
- Construção de uma ALU de 32 bits

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



## Era uma vez...

- Era uma vez, num país bué, bué longe, um conselho de ministros...
- O Sr. Primeiro Ministro dirigiu-se aos Srs. Ministros, questionando-os sobre quantos milhares de milhões deveria atribuir ao NewBank no orçamento de 2020, para equilibrar ☺☺☺ os desvarios do mesmo.

***Dizei-me senhores ministros  
No vosso saber dotado  
Qual deve ser o valor  
Pr'a dar ao NewBank, coitado***



## Era uma vez...

- Fácil, disse o Ministro da Cultura: eu acho que devia ser  
**00000101 G€**
- Não, retorquiu o Ministro dos Negócios Estrangeiros, para mim  
devia ser **01000011 01001001 01001110 01000011 01001111 G€**
- Não concordo, contestou o Ministro do Trabalho, eu cá acho que  
devia ser **01010110 G€**
- Nada disso, insurgiu-se o Ministro da Economia. Para sermos  
justos é necessário um aumento de  
**01000000101000000000000000000000000000000 G€**
- Por todos os deuses – levantou-se o Ministro das Finanças irritado  
– só um cego não vê que o orçamento só suporta um aumento de  
**10000100 G€**



## Era uma vez...

- O Primeiro Ministro desse país longínquo, conhecido como um hábil negociador, nada sabia de códigos de representação e ficou bastante irritado com as respostas dos seus ministros
- No entanto, não havia razão para tal, uma vez que houve unanimidade nas respostas. A resposta dada por cada ministro foi, na realidade, a mesma; apenas usaram uma linguagem (código) diferente
- A extração da informação requer, assim, o conhecimento do código usado, sob pena de as mensagens não passarem de coleções de bits sem sentido



## Era uma vez...

O Ministro da Cultura codificou a sua resposta em **binário**:

$$00000101_2 = 5_{10} \text{ G€}$$

O Ministro dos Negócios Estrangeiros usou **ASCII**:

**01000011 01001001 01001110 01000011 01001111** = "CINCO" G€

O Ministro do Trabalho usou **ASCII** mas para representar numeração romana:

$$01010110 = "V" = 5 \text{ G€}$$

O Ministro da Economia usou representação em **vírgula flutuante**:

$$01000000101000000000000000000000 = 1.01_2 \times 2^2 = 5 \text{ G€}$$

O Ministro das Finanças usou **excesso de  $2^{n-1}-1$**  (com n=8, excesso de 127):

$$10000100_2 = 5_{10} \text{ G€}$$



# Representação de inteiros

- No sistema árabe, cada algarismo que compõe um dado número tem um peso que é função quer da sua posição no número quer do número de símbolos do alfabeto usado.
- Um número com  $n$  dígitos  $d_{n-1} d_{n-2} \dots d_1 d_0$  representado neste sistema, pode ser decomposto num polinómio da forma
$$d_{n-1} \cdot b^{n-1} + d_{n-2} \cdot b^{n-2} + \dots + d_1 \cdot b^1 + d_0 \cdot b^0$$
em que  $b$  é a base de representação e corresponde à dimensão do alfabeto
- Exemplos:

$$1230_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10 + 0$$

$$110101_2 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 1 = 53_{10}$$

$$721_8 = 7 \times 8^2 + 2 \times 8 + 1 = 465_{10}$$

$$5A8_{16} = 5 \times 16^2 + A \times 16 + 8 = 1448_{10}$$



# Representação de inteiros

- Sendo um computador um sistema digital binário, a representação de inteiros faz-se sempre em base 2 (símbolos 0 e 1).
- Por outro lado, como o espaço de armazenamento de informação (numérica ou não) é limitado, a representação de inteiros é também necessariamente limitada.
- **Tipicamente, um inteiro pode ocupar um número de bits igual à dimensão de um registo interno do CPU.**
- A gama de valores inteiros representáveis é, assim, finita, e corresponde ao número máximo de combinações que é possível obter com o número de bits de um registo interno.
- No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representável é:

$$N_{\text{inteiros}} = 2^{32} = 4.294.967.296_{10} = [0 \dots 4.294.967.295_{10}]$$



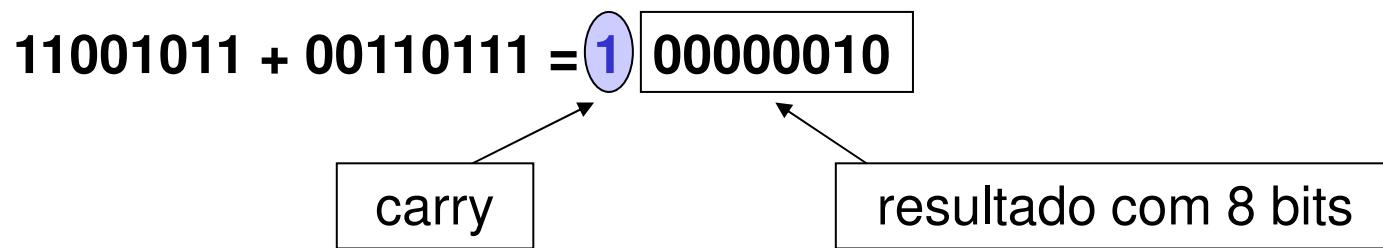
# Representação de inteiros

- Os circuitos que realizam operações aritméticas estão igualmente limitados a um número finito de dígitos (bits), geralmente igual à dimensão dos registos internos do CPU.
- Os circuitos aritméticos operam assim em aritmética modular, ou seja em  $\text{mod}(2^n)$  em que 'n' é o número de bits de representação.
- O maior valor que um resultado aritmético pode tomar será portanto  $2^n - 1$ , sendo o valor inteiro imediatamente a seguir o valor zero (representação circular).



# Representação de inteiros

- Num CPU com registos de 8 bits, por exemplo, o resultado da soma dos números 11001011 e 00110111 seria:



- No caso em que os operandos são do tipo ***unsigned***, o bit ***carry*** sinaliza que o resultado não cabe num registo de 8 bits, ou seja sinaliza a ocorrência de ***overflow***
- No caso em que os operandos são do tipo ***signed*** (codificados em complemento para 2) o bit de ***carry*** não tem qualquer significado e é ignorado.

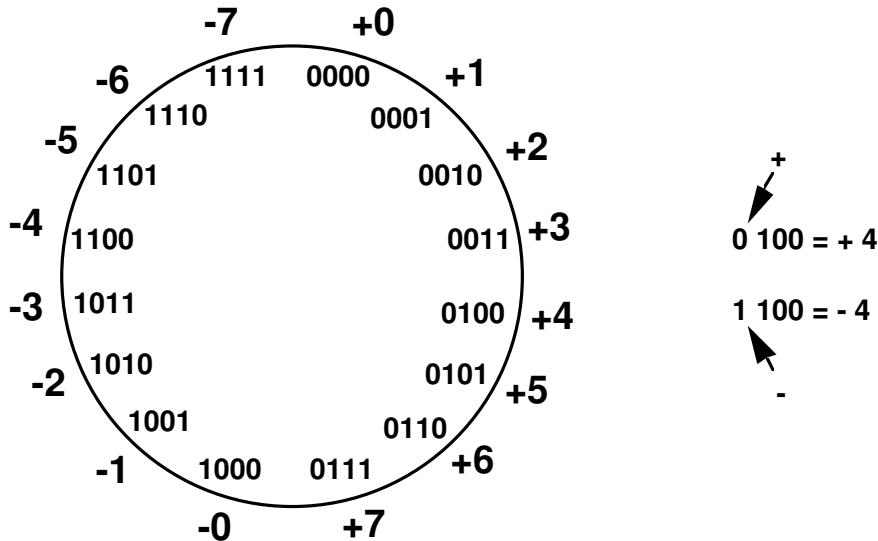


# Representação de inteiros negativos

- A representação de números positivos é a mesma na maioria dos sistemas numéricos
- Os maiores problemas colocam-se quando se procura uma forma de representar quantidades negativas
- Os três esquemas mais usados são:
  - sinal e módulo
  - complemento para um
  - complemento para dois
- Por uma questão de simplicidade vamos admitir, na discussão subsequente, que a dimensão do registo interno do CPU é de 4 bits



# Representação em sinal e módulo



- O bit mais significativo é usado para representar o sinal:
  - 0 = positivo (ou zero), 1 = negativo
- A magnitude é representada pelos 3 LSBits: 0 (000) a 7 (111)
- Gama de representação para n bits =  $+/-2^{n-1}-1$
- 2 representações para 0



# Representação em sinal e módulo

- Este método de representação de inteiros apresenta os seguintes problemas do ponto de vista da implementação numa ALU:
  - Existem duas representações distintas para um mesmo valor (zero)
  - É necessário comparar as magnitudes dos operandos para determinar o sinal do resultado
  - É necessário implementar um somador e um subtrator distintos
  - O bit de sinal tem de ser tratado independentemente dos restantes



# Representação em complemento para um

- **Definição:** Se  $N$  é um número positivo, então  $\bar{N}$  é negativo e o seu complemento para 1 (complemento falso) é dado por:

$$\bar{N} = (2^n - 1) - N$$

em que  $n$  é o número de bits da representação

- **Exemplo:** determinar o complemento para 1 de 5 (com 4 bits)

$$N = 5_{10} = 0101_2$$

$$2^n = 2^4 = 10000$$

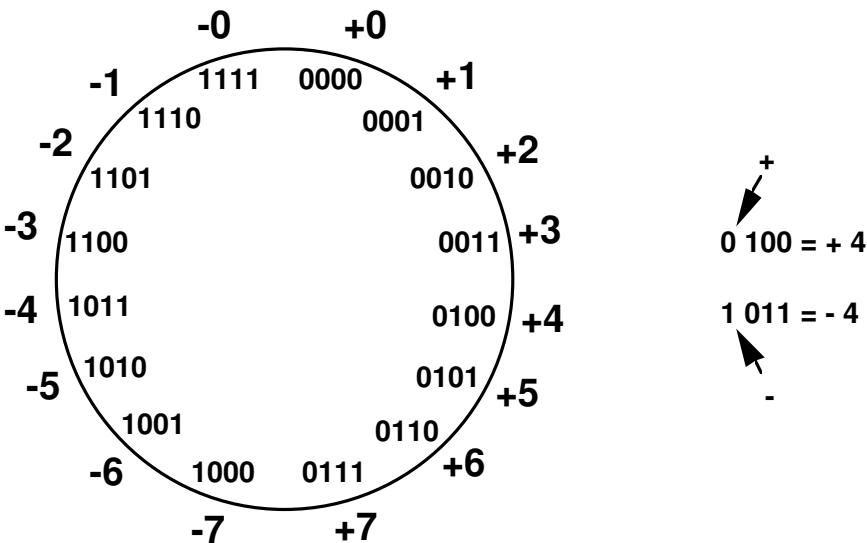
$$(2^n - 1) = 10000 - 1 = 1111$$

$$(2^n - 1) - N = 1111 - 0101 = 1010$$

- **Método prático:** inverter todos os bits do valor original



# Representação em complemento para um



$$\begin{array}{l} + \\ \hline 0\ 100 = +4 \\ - \\ 1\ 011 = -4 \end{array}$$

- O bit mais significativo também pode ser interpretado como sinal: 0 = valor positivo, 1 = valor negativo
- A subtração faz-se adicionando o complemento para 1
- Há 2 representações para 0 (tem implicações no modo como as operações são realizadas)



# Representação em complemento para dois

- **Definição:** Se  $N$  é um número positivo, então  $N^*$  é o seu complemento para 2 (complemento verdadeiro) e é dado por:

$$N^* = 2^n - N$$

em que “n” é o número de bits da representação

- **Exemplo:** determinar o complemento para 2 de 5 (com 4 bits)

$$N = 5_{10} = 0101_2$$

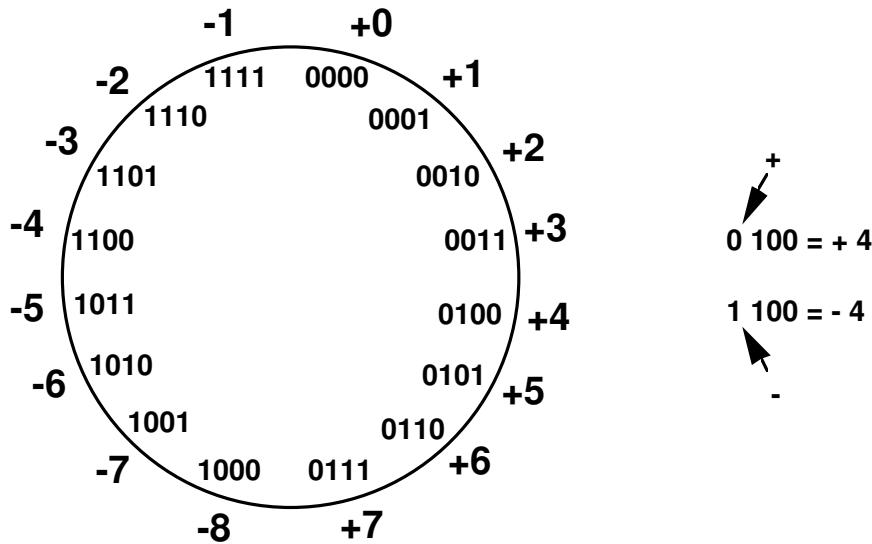
$$2^n = 2^4 = 10000$$

$$2^n - N = 10000 - 0101 = 1011 = N^*$$

- **Método prático:** inverter todos os bits do valor original e somar 1



# Representação em complemento para dois



$$\begin{array}{l} + \\ \hline 0\ 100 = +4 \\ - \\ 1\ 100 = -4 \end{array}$$

- O bit mais significativo também **pode ser interpretado como sinal**: 0 = valor positivo, 1 = valor negativo
- Uma única representação para 0
- Codificação assimétrica (mais um negativo do que positivos)
- A subtração é obtida através da soma com o complemento para 2  $(a - b) = (a + (-b))$



# Representação em complemento para dois

- Uma quantidade de 32 bits codificada em complemento para 2 pode ser representada pelo seguinte polinómio:

$$-(a_{31} \cdot 2^{31}) + (a_{30} \cdot 2^{30}) + \dots + (a_1 \cdot 2^1) + (a_0 \cdot 2^0)$$

Onde o bit de sinal ( $a_{31}$ ) é multiplicado por  $-2^{31}$  e os restantes pela versão positiva do respetivo peso

- **Exemplo:** Qual o valor representado pela quantidade  $10100101_2$ , supondo uma representação com 8 bits e uma codificação em complemento para 2?

- R1:  $10100101_2 = -(1 \cdot 2^7) + (1 \cdot 2^5) + (1 \cdot 2^2) + (1 \cdot 2^0)$   
 $= -128 + 32 + 4 + 1 = -91_{10}$

- R2: Complemento para 2 de  $10100101 = 01011010 + 1$   
 $= 01011011_2 = 5B_{16} = 91_{10}$ . Ou seja, o valor representado em sinal e módulo, base 10, é  $-91_{10}$



# Representação em complemento para dois

- Exemplos de operações

$$\begin{array}{r} 4 \quad 0100 \\ + 3 \quad \underline{0011} \\ \hline 7 \quad 0111 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + (-3) \quad \underline{1101} \\ \hline -7 \quad \underline{11001} \end{array}$$

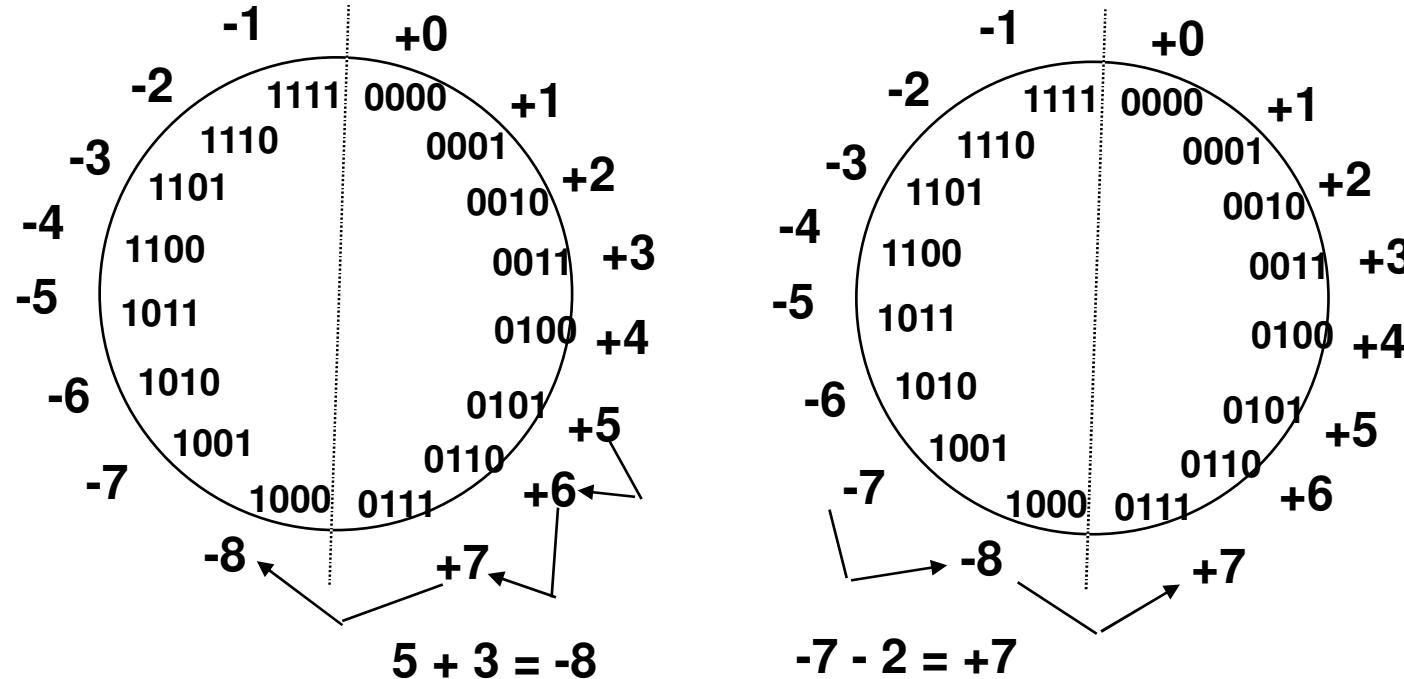
$$\begin{array}{r} 4 \quad 0100 \\ - 3 \quad \underline{1101} \\ \hline 1 \quad \underline{10001} \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + 3 \quad \underline{0011} \\ \hline -1 \quad 1111 \end{array}$$

- Este esquema simples de adição com sinal torna o complemento para 2 o preferido para representação de inteiros em arquitetura de computadores



# Overflow em complemento para 2



- Ocorre **overflow** quando é ultrapassada a gama de representação. Isso acontece quando:
  - se somam dois positivos e o resultado obtido é negativo
  - se somam dois negativos e o resultado obtido é positivo

# Overflow em complemento para 2

|  |  |   |
|--|--|---|
| $\begin{array}{r} 5 \\ - 2 \\ \hline 7 \end{array}$        | $\begin{array}{r} 0000 \\ 0101 \\ \hline 0010 \\ 0111 \end{array}$<br>Sem overflow | $\begin{array}{r} -3 \\ -5 \\ -8 \\ \hline -10 \\ =S \end{array}$<br>$\begin{array}{r} 111 \\ 1101 \\ \hline 1011 \\ 1000 \end{array}$<br>Sem overflow  |
| $\begin{array}{r} 5 \\ - 3 \\ -8 \\ \hline -8 \end{array}$ | $\begin{array}{r} 0111 \\ 0101 \\ \hline 0011 \\ 1000 \end{array}$<br>Overflow     | $\begin{array}{r} -7 \\ -2 \\ 7 \\ \hline 0111 \\ \neq S \end{array}$<br>$\begin{array}{r} 1000 \\ 1001 \\ \hline 1110 \\ 0111 \end{array}$<br>Overflow |

A situação de **overflow** ocorre quando o *carry-in* do bit de sinal não é igual ao *carry-out*, ou seja, quando:

$$C_{n-1} \oplus C_n = 1$$



# Overflow em operações aritméticas

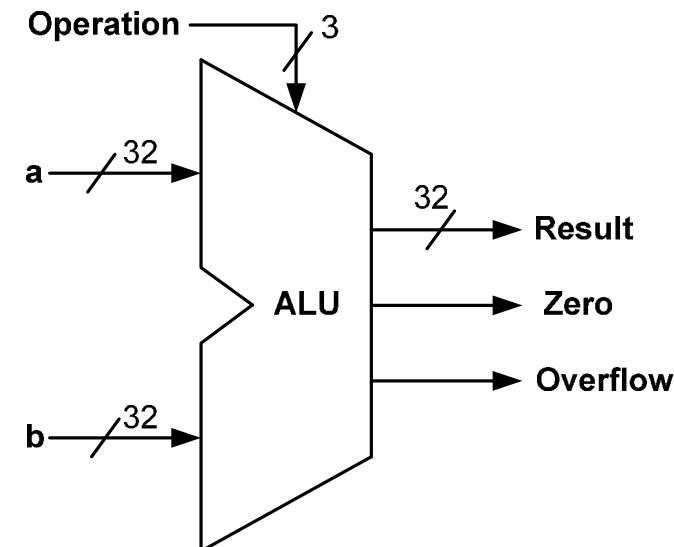
- Em operações sem sinal:
  - Quando  $A+B > 2^{n-1}$  ou  $A-B$  c/  $B>A$
  - O bit de carry  $C_n = 1$  sinaliza a ocorrência de *overflow*
- Em operações com sinal:
  - Quando  $A + B > 2^{n-1}-1$  ou  $A + B < -2^{n-1}$ 
    - $OVF = (C_{n-1} \cdot \overline{C_n}) + (\overline{C_{n-1}} \cdot C_n) = C_{n-1} \oplus C_n$
  - Alternativamente, não tendo acesso aos bits intermédios de carry, ( $R = A + B$ ):
    - $OVF = R_{n-1} \cdot \overline{A_{n-1}} \cdot \overline{B_{n-1}} + \overline{R_{n-1}} \cdot A_{n-1} \cdot B_{n-1}$
- Como fazer a deteção de *overflow* em operações com e sem sinal no MIPS?



# Construção de uma ALU de 32 bits

- A ALU deverá implementar as operações:
  - AND, OR
  - ADD, SUB
  - SLT (set if less than)
- Deverá ainda:
  - Detetar e sinalizar *overflow*
  - Sinalizar resultado igual a zero

| Operation | ALU Action       |
|-----------|------------------|
| 0 0 0     | And              |
| 0 0 1     | Or               |
| 0 1 0     | Add              |
| 1 1 0     | Subtract         |
| 1 1 1     | Set if less than |

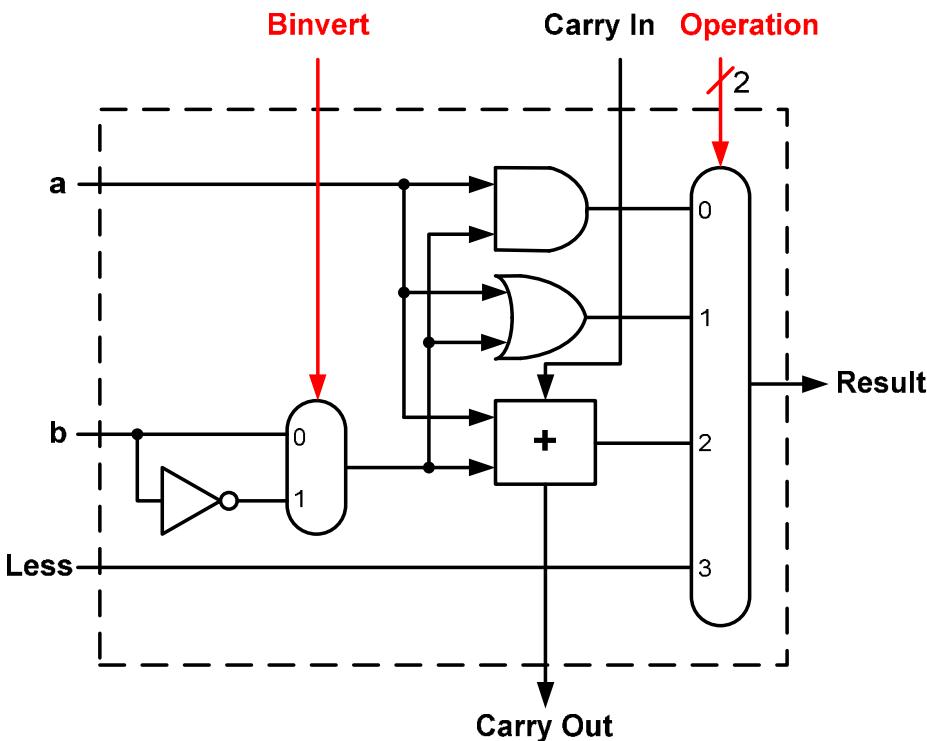


Bloco funcional  
correspondente a uma  
ALU de 32 bits



# Construção de uma ALU de 32 bits

## Construção de uma ALU básica de 1 bit:

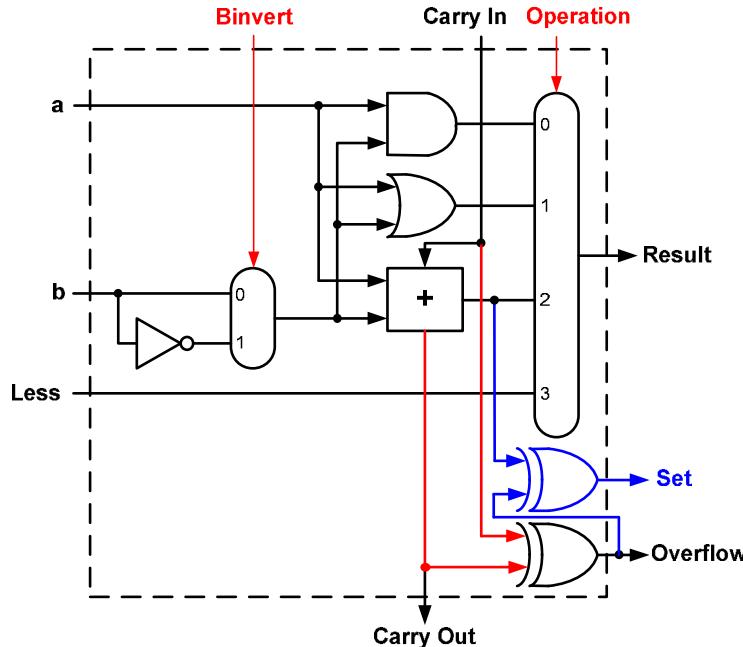


- Esta ALU permite efetuar as operações aritméticas de **adição** e **subtração**, e as operações lógicas **AND** e **OR**
- A operação é selecionada pelo sinal **Operation** (2 bits: **00 – AND**, **01 – OR**, **10 – ADD**, **11 – SLT**)
- A subtração obtém-se colocando um “**1**” em **Binvert** e **Carry In**



# Construção de uma ALU de 32 bits

## ALU básica de 1 bit, com detecção de *overflow*:



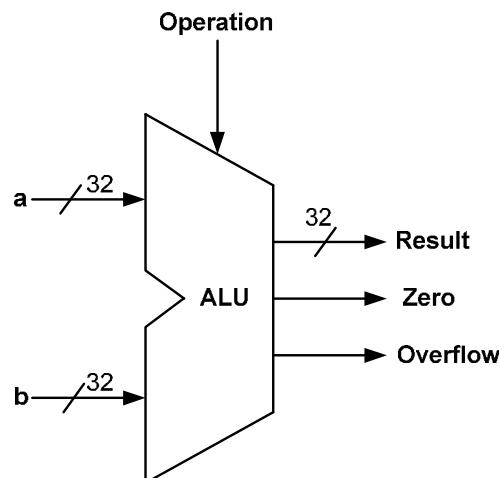
- Detecção de overflow:  
 $\text{overflow} = \text{carry\_in} \oplus \text{carry\_out}$ , no bit mais significativo da ALU
- Esta ALU permite ainda efectuar a operação **SLT** (*set if less than*)
- A operação SLT é realizada através da operação (*a-b*):
  - saída **Set=1** se  $a < b \rightarrow (a-b) < 0$
  - saída **Set=0** se  $a \geq b \rightarrow (a-b) \geq 0$

- Na operação de subtração ( $a+(-b)$ ) o bit mais significativo do resultado é “1” se  $a < b$  e “0” se  $a \geq b$ . Esse bit pode, assim, ser usado para a implementação da instrução **SLT**
- No entanto, quando ocorre **overflow**, o bit mais significativo do resultado vem trocado, pelo que, nessa situação, é necessário **negá-lo** para que a saída **set** seja correcta

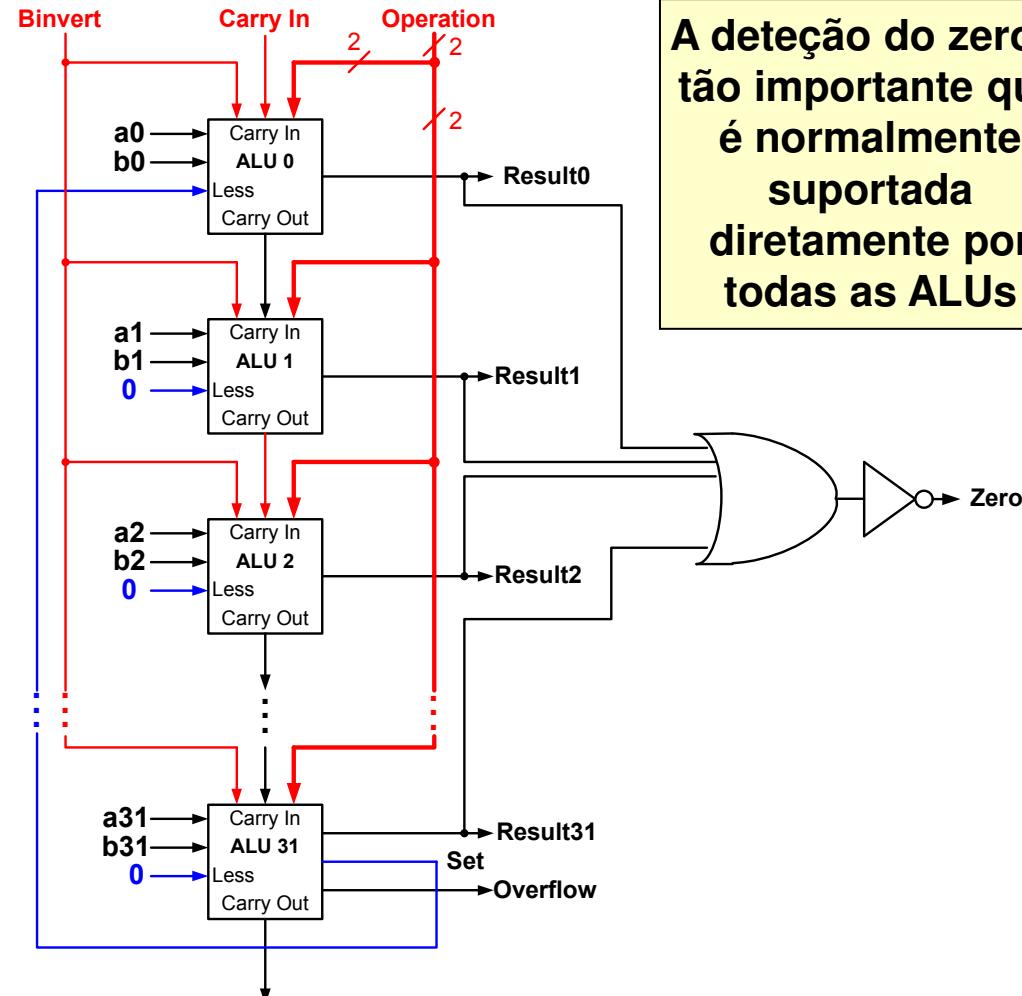


# Construção de uma ALU de 32 bits

**Expansão para 32 bits em *ripple carry*:**



**Bloco funcional correspondente a uma ALU de 32 bits**



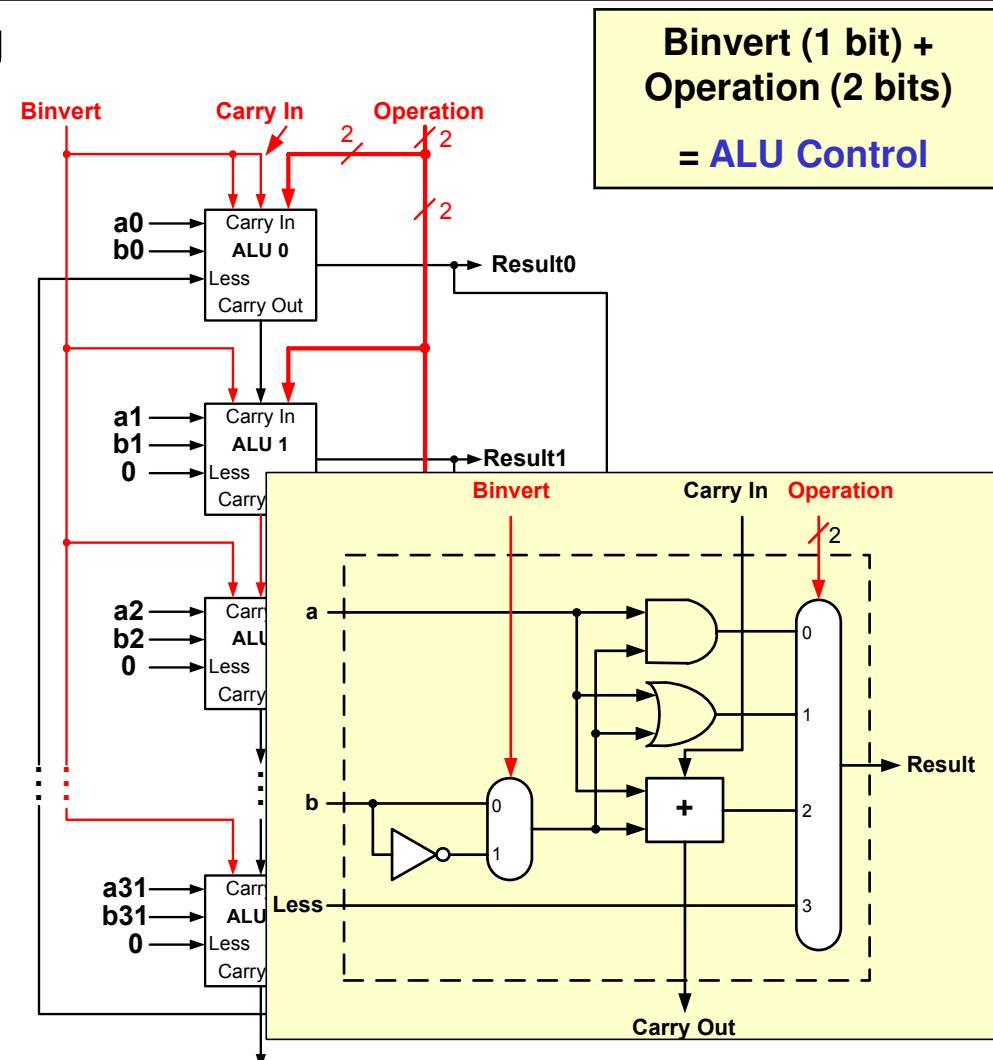
# Construção de uma ALU de 32 bits

## Sinais de controlo da ALU

Os sinais de controlo directo da ALU podem ser reduzidos a três, uma vez que os sinais *Binvert* e *Carry In* podem ser combinados num só.

| ALU Control | ALU Action       |
|-------------|------------------|
| 0 0 0       | And              |
| 0 0 1       | Or               |
| 0 1 0       | Add              |
| 1 1 0       | Subtract         |
| 1 1 1       | Set if less than |

Bit "**Binvert**"



# Construção de uma ALU de 32 bits – VHDL

```
entity alu32 is
    port( a      : in  std_logic_vector(31 downto 0);
          b      : in  std_logic_vector(31 downto 0);
          oper   : in  std_logic_vector(2  downto 0);
          res    : out std_logic_vector(31 downto 0);
          zero   : out std_logic;
          ovf    : out std_logic);
end alu32;

architecture Behavioral of alu32 is
    signal s_res : std_logic_vector(31 downto 0);
    signal s_b   : unsigned(31 downto 0);
begin
    s_b  <= not(unsigned(b)) + 1 when oper = "110" else
          unsigned(b); -- complemento para 2 (se subtração)
    res  <= s_res;
    zero <= '1' when s_res = X"00000000" else '0';
    ovf  <= (not a(31) and not s_b(31) and s_res(31)) or
          (a(31) and s_b(31) and not s_res(31));
-- (continua)
```

| Operation | ALU Action       |
|-----------|------------------|
| 0 0 0     | And              |
| 0 0 1     | Or               |
| 0 1 0     | Add              |
| 1 1 0     | Subtract         |
| 1 1 1     | Set if less than |



```

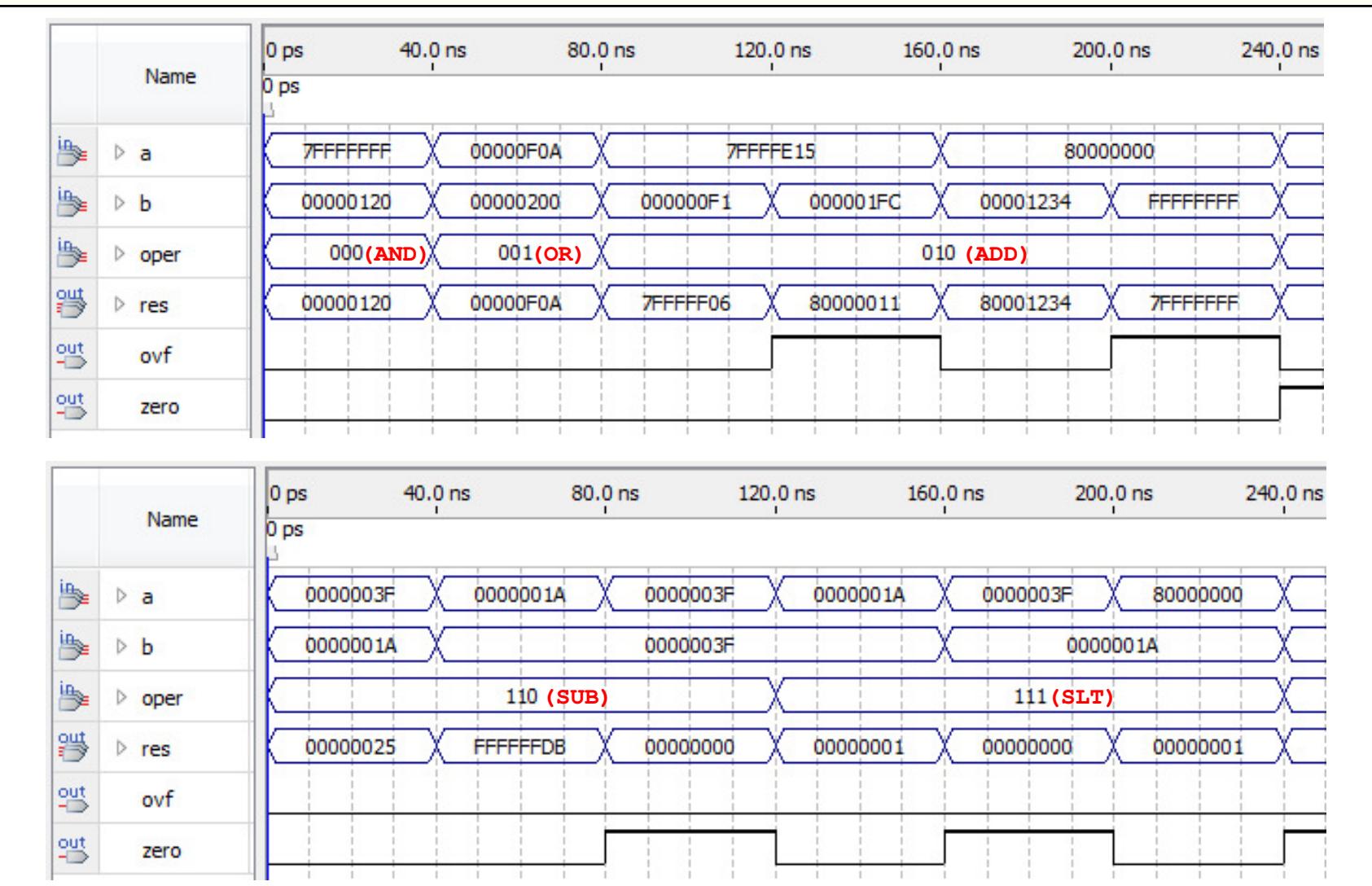
process(oper, a, b, s_b)
begin
    case oper is
        when "000" => -- AND
            s_res <= a and b;
        when "001" => -- OR
            s_res <= a or b;
        when "010" => -- ADD
            s_res <= std_logic_vector(unsigned(a) + s_b);
        when "110" => -- SUB
            s_res <= std_logic_vector(unsigned(a) - s_b);
        when "111" => -- SLT
            if(signed(a) < signed(b)) then
                s_res <= X"00000001";
            else
                s_res <= (others => '0');
            end if;
        when others =>
            s_res <= (others => '-');
    end case;
end process;
end Behavioral;

```

Construção de uma ALU de  
32 bits (continuação)

| Operation | ALU Action       |
|-----------|------------------|
| 0 0 0     | And              |
| 0 0 1     | Or               |
| 0 1 0     | Add              |
| 1 1 0     | Subtract         |
| 1 1 1     | Set if less than |

# ALU – resultado da simulação



## Aula 10

- Arquitetura de um multiplicador de inteiros
- A multiplicação de inteiros no MIPS
  
- Arquitetura de um divisor de inteiros
- Divisão de inteiros com sinal
- Divisão de inteiros no MIPS

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Arquitetura de um Multiplicador

- Devido ao aumento de complexidade que daí resulta, nem todas as arquiteturas suportam, ao nível do *hardware*, a capacidade para efetuar operações aritméticas de multiplicação e divisão de inteiros
- No caso do MIPS, essas operações são asseguradas por uma unidade especial de multiplicação e divisão de inteiros
- Note-se que uma multiplicação que envolva **dois operandos de  $n$  bits** carece de um espaço de armazenamento, para o resultado, de  **$2*n$  bits**
- Tal implica que o **resultado**, no caso do MIPS, deverá ser armazenado com **64 bits**, o que determina a existência de **registos especiais** para esse mesmo armazenamento



# Arquitetura de um Multiplicador

- A arquitetura de um multiplicador utiliza, em grande parte, o algoritmo da multiplicação que todos aprendemos a usar no 1º ciclo

$$\begin{array}{r} 0101 \\ \times 0110 \\ \hline 0000 \\ 01010 \\ 010100 \\ +0000000 \\ \hline 0011110 \end{array}$$

- *Para além da solução iterativa estudada nesta aula é também possível construir multiplicadores combinatórios, tal como foi estudado em Sistemas Digitais*



# Arquitetura de um Multiplicador

- Esse algoritmo tira partido da propriedade distributiva em relação à adição, permitindo que a multiplicação seja decomposta numa sucessão de somas de produtos parciais
- Considere-se o seguinte produto, em que **M representa o multiplicando** e **m o multiplicador** representados com 4 bits:  
$$R = M \cdot m$$

$$M \cdot m = M \cdot (m_3 \cdot 2^3 + m_2 \cdot 2^2 + m_1 \cdot 2^1 + m_0 \cdot 2^0)$$

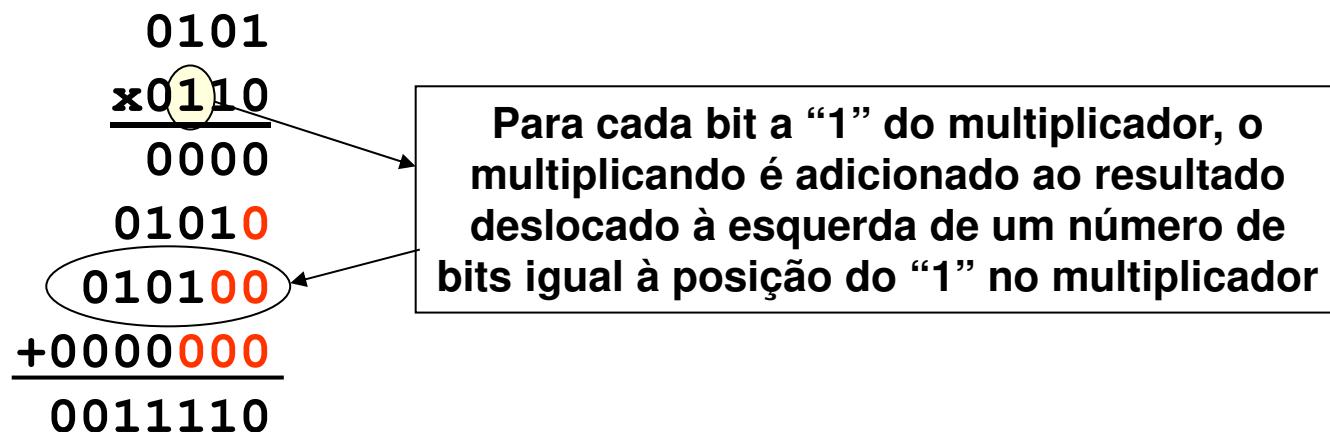
$$M \cdot m = (M \cdot 2^3 \cdot m_3) + (M \cdot 2^2 \cdot m_2) + (M \cdot 2^1 \cdot m_1) + (M \cdot 2^0 \cdot m_0)$$

$$M \cdot m = ((M \cdot 2^3) \cdot m_3) + ((M \cdot 2^2) \cdot m_2) + ((M \cdot 2^1) \cdot m_1) + ((M \cdot 2^0) \cdot m_0)$$

# Arquitetura de um Multiplicador

$$M \cdot m = ((M \cdot 2^3) \cdot m_3) + ((M \cdot 2^2) \cdot m_2) + ((M \cdot 2^1) \cdot m_1) + ((M \cdot 2^0) \cdot m_0)$$

- Multiplicar por dois (ou por uma potência de dois) corresponde a deslocar o número multiplicado à esquerda (**shift left**) tantos bits quantos o valor do expoente da potência de dois envolvida
- Por outro lado, se  $m_n$  for igual a "0", o produto parcial correspondente também será zero, e se for "1", o mesmo produto parcial será igual ao multiplicando deslocado à esquerda de n bits



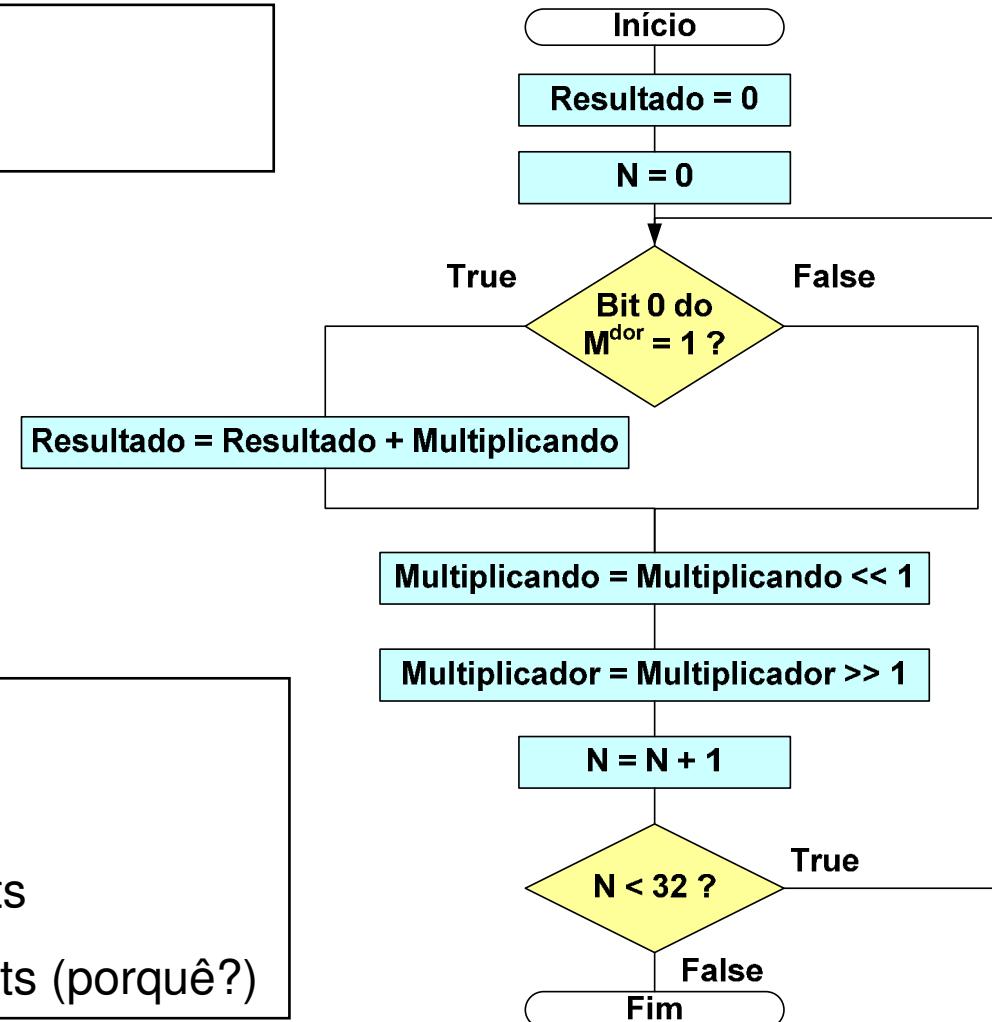
# Multiplicação de inteiros de 32 bits - algoritmo

Operandos: 32 bits

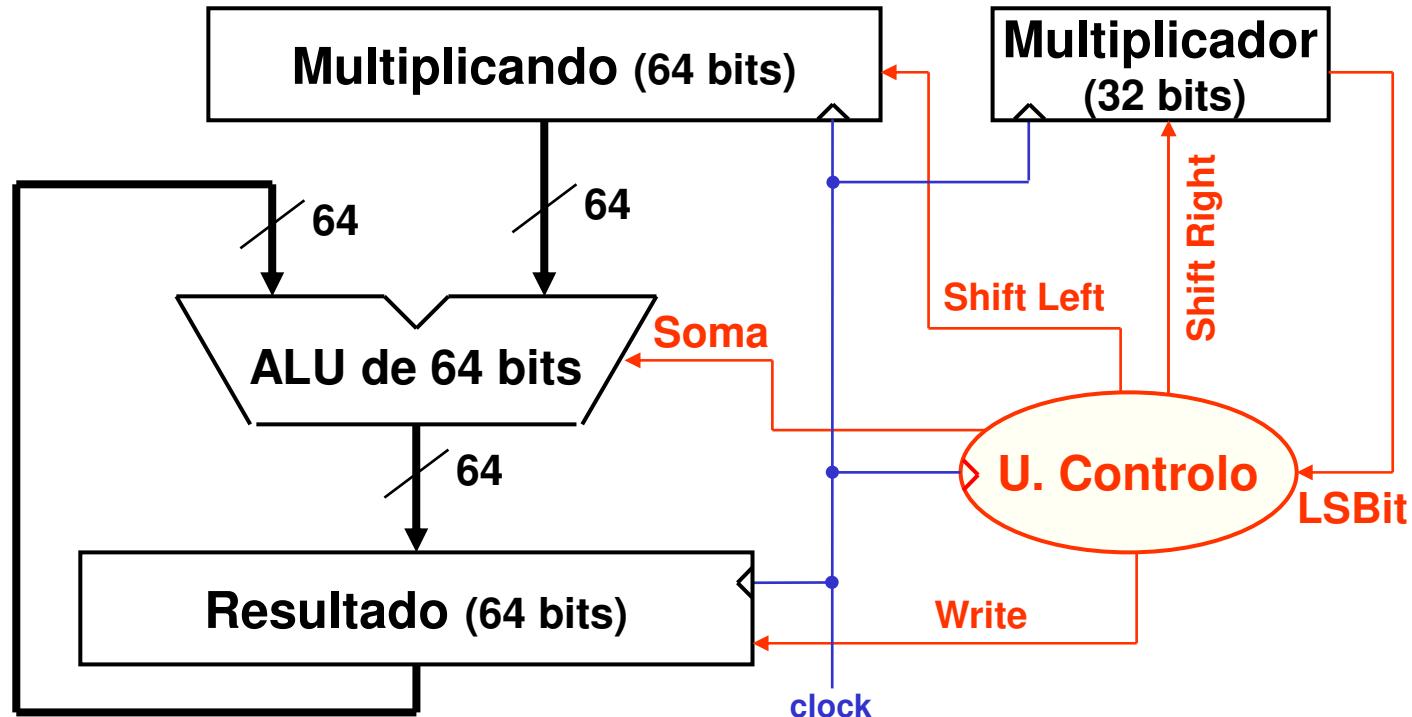
Resultado: 64 bits

Registros necessários:

- **Resultado**: 64 bits
- **Multiplicador**: 32 bits
- **Multiplicando**: 64 bits (porquê?)



# Arquitetura de um Multiplicador de 32 bits (1<sup>a</sup> versão)



- Nesta 1<sup>a</sup> versão do multiplicador, a **ALU** e os registos **Multiplicando** e **Resultado** operam com 64 bits
- *Não estão representados os sinais para carregar (load) os valores do multiplicando e do multiplicador e reset do resultado*



# Multiplicação de inteiros – exemplo com 4 bits

- Com operandos de 4 bits, o resultado terá uma dimensão máxima de 8 bits
- Para a implementação de um multiplicador de 4 bits, que aplique o algoritmo do slide anterior, os registos necessários são:
  - **resultado**: registo de 8 bits
  - **multiplicando**: registo de 8 bits (inicialmente os bits mais significativos são colocados a 0000)
  - **multiplicador**: registo de 4 bits

$$\begin{array}{r} & \begin{array}{cccc} 0 & 1 & 0 & 1 \end{array} \\ \times & \begin{array}{ccccc} 0 & 1 & 1 & 0 \end{array} \\ \hline \end{array}$$
$$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\ + 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \\ + 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \end{array} \begin{array}{l} \text{Res. Inicial} \\ 0.\text{mdo.}2^0 \\ 1.\text{mdo.}2^1 \\ 1.\text{mdo.}2^2 \\ 0.\text{mdo.}2^3 \\ \text{Res. FINAL} \end{array}$$



## Arquitetura de um Multiplicador de 32 bits (2<sup>a</sup> versão)

- Por cada nova iteração obtém-se o valor final de um novo bit do resultado (as iterações seguintes somam 0 a esse bit)
- Este algoritmo pode ser melhorado:
  - Deslocar, a cada iteração, o resultado para a direita, em vez de o multiplicando para a esquerda (o movimento relativo dos dois é o mesmo, logo o mesmo efeito algorítmico é obtido)
  - O multiplicador continua a ser deslocado à direita a cada iteração
- Para cada nova adição é suficiente operar apenas sobre 32 bits dos 64 bits do resultado final
- O registo utilizado para armazenar o multiplicando pode ter apenas 32 bits (em vez de 64 bits) e a ALU é também de 32 bits



# Multiplicação de inteiros – exemplo com 4 bits

- Uma alternativa ao algoritmo inicial será portanto deslocar o resultado à direita por cada nova iteração
- Vantagens desta nova abordagem:
  - Para cada nova adição é suficiente operar apenas sobre 4 bits dos 8 bits do resultado final
  - O registo utilizado para armazenar o multiplicando pode ter apenas 4 bits (em vez de 8 bits)

|  |         |              |
|--|---------|--------------|
| $\begin{array}{r} 0 \ 1 \ 0 \ 1 \\ \times \ 0 \ 1 \ 1 \ 0 \\ \hline \end{array}$ |         |              |
| 0 0 0 0  | 0 0 0 0 | Res. Inicial |
| + 0 0 0 0  |         | 0.mdo        |
| 0 0 0 0  | 0 0 0 0 |              |
| 0 0 0 0  | 0 0 0 0 | Após >> 1    |
| + 0 1 0 1  |         | 1.mdo        |
| 0 1 0 1  | 0 0 0 0 |              |
| 0 0 1 0  | 1 0 0 0 | Após >> 1    |
| + 0 1 0 1  |         | 1.mdo        |
| 0 1 1 1  | 1 0 0 0 |              |
| 0 0 1 1  | 1 1 0 0 | Após >> 1    |
| + 0 0 0 0  |         | 0.mdo        |
| 0 0 1 1  | 1 1 0 0 |              |
| 0 0 0 1  | 1 1 1 0 | Após >> 1    |
|  |         | Res. FINAL   |

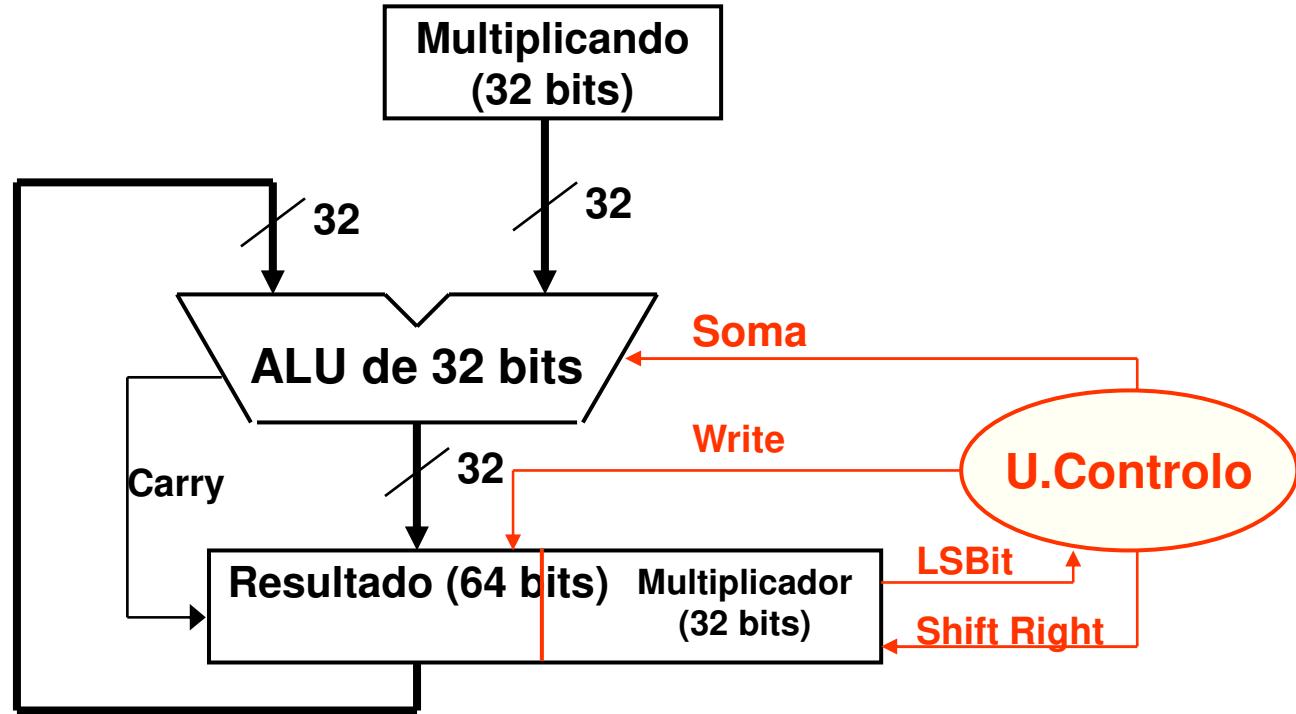


## Arquitetura de um Multiplicador de 32 bits (versão final)

- Os 32 bits menos significativos do resultado no início não têm qualquer informação útil (vão sendo preenchidos a cada nova iteração)
- O sucessivo deslocamento à direita do resultado, é acompanhado por um deslocamento idêntico do multiplicador
- É então possível utilizar a parte menos significativa do resultado (32 bits) para armazenar o valor inicial do multiplicador
- Otimiza-se, deste modo, o espaço total de armazenamento e os recursos necessários à arquitetura de multiplicação:
  - Registo de 64 bits para o resultado que armazena inicialmente o multiplicador
  - Registo de 32 bits para o armazenamento do multiplicando
  - ALU de 32 bits



# Arquitetura de um Multiplicador de 32 bits (versão final)



Na versão otimizada do multiplicador, o registo **Multiplicando** e a **ALU** passam a ser de 32 bits. O registo **Multiplicador** desaparece, sendo substituído pela parte menos significativa do **Resultado**.

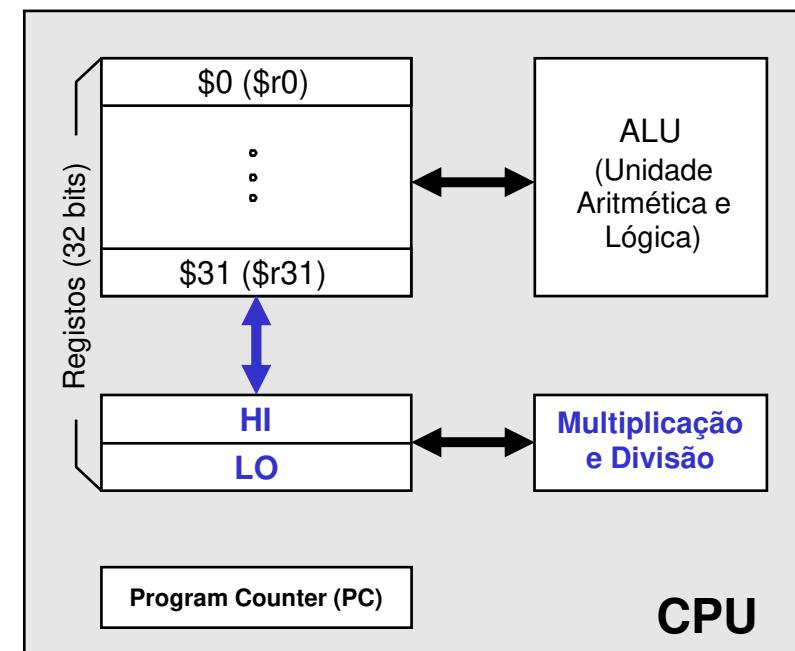
# Multiplicação de inteiros com sinal

- A arquitetura de multiplicação que obtivemos no slide anterior apenas opera corretamente sobre quantidades em binário natural (inteiros sem sinal)
- É portanto uma arquitetura útil para a realização de operações de **multiplicação *unsigned***
- Para a multiplicação de valores em binário com sinal (**multiplicação *signed***), codificados em complemento para dois, utiliza-se o **algoritmo de Booth** (não apresentado nestas aulas) consultar - J.Hennessy, D.A.Patterson, *Computer Organization and Design – the hardware/software interface*, Elsevier.
- A arquitetura que usa o algoritmo de Booth é semelhante à apresentada anteriormente, diferindo apenas na Unidade de Controlo



# A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação é assegurada por uma arquitetura semelhante à anteriormente descrita
- Para o armazenamento do multiplicador e do resultado final, os arquitetos do MIPS incluíram um par de registros especiais designados, respetivamente, por **HI** e **LO**
- Estes registros são de uso específico da unidade de multiplicação e divisão de inteiros



$$\boxed{\text{op1}} \times \boxed{\text{op1}} = \boxed{\text{hi}} \quad \boxed{\text{lo}}$$

# A Multiplicação de inteiros no MIPS

- O registo **HI** armazena os **32 bits mais significativos do resultado**
- O registo **LO** armazena, inicialmente, o multiplicador e, após a execução da operação, os **32 bits menos significativos do resultado**
- A transferência do multiplicador para o registo LO é efetuada automaticamente (por hardware) no início da execução da operação
- A unidade de multiplicação pode operar considerando os operandos sem sinal (multiplicação *unsigned*) ou com sinal (multiplicação *signed*); a distinção é feita através da mnemónica da instrução



# A Multiplicação de inteiros no MIPS

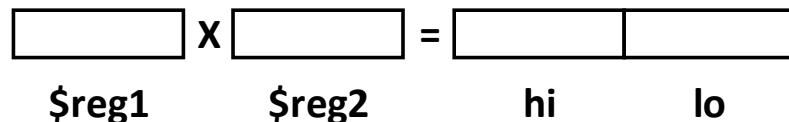
- Em Assembly, a multiplicação é efetuada pela instrução

**mult \$reg1, \$reg2** # Multiply (signed)

**multu \$reg1, \$reg2** # Multiply unsigned

em que **\$reg1** é o multiplicando e **\$reg2** o multiplicador

- O **resultado** fica armazenado nos **registos HI e LO**



- A transferência de informação entre os registos HI e LO e os restantes registos de uso geral faz-se através das instruções **mfhi** e **mflo**:

**mfhi \$reg** # **move from hi** - Copia HI para \$reg

**mflo \$reg** # **move from lo** - Copia LO para \$reg

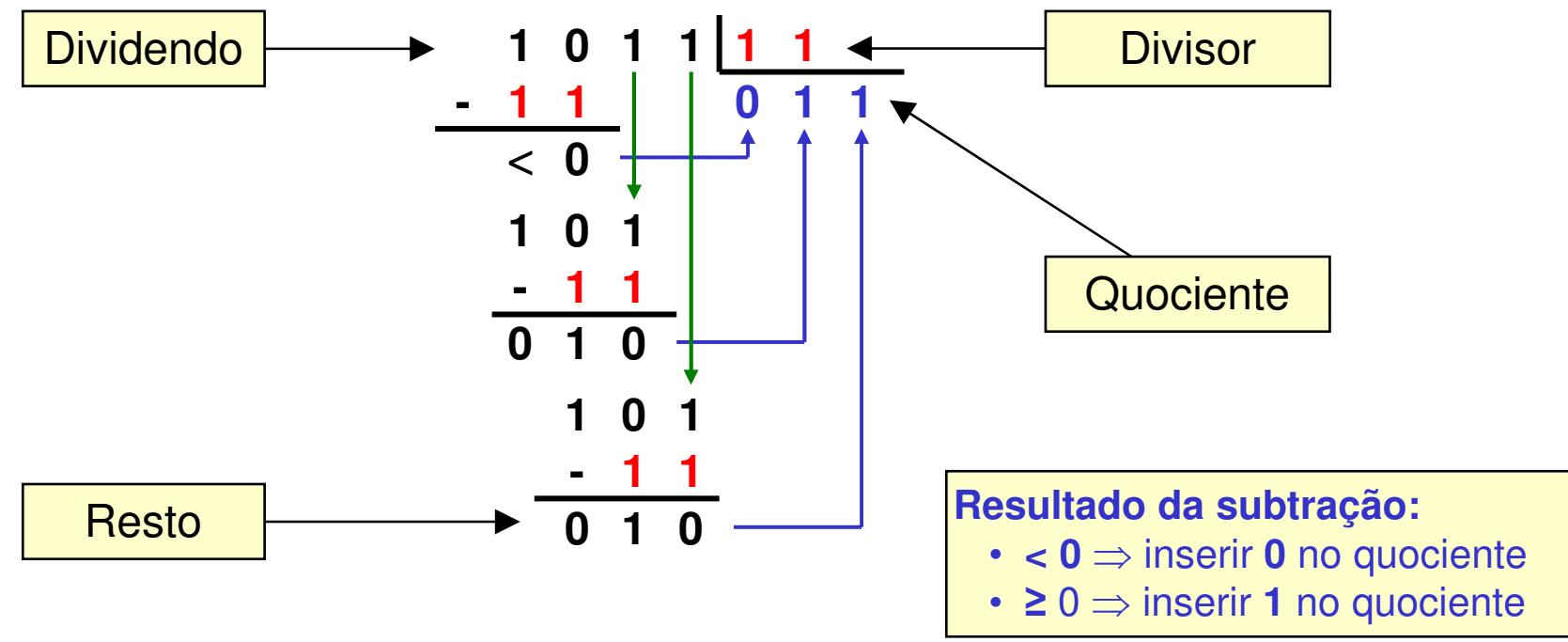
**mthi \$reg** # **move to hi** - Copia \$reg para HI

**mtlo \$reg** # **move to lo** - Copia \$reg para LO

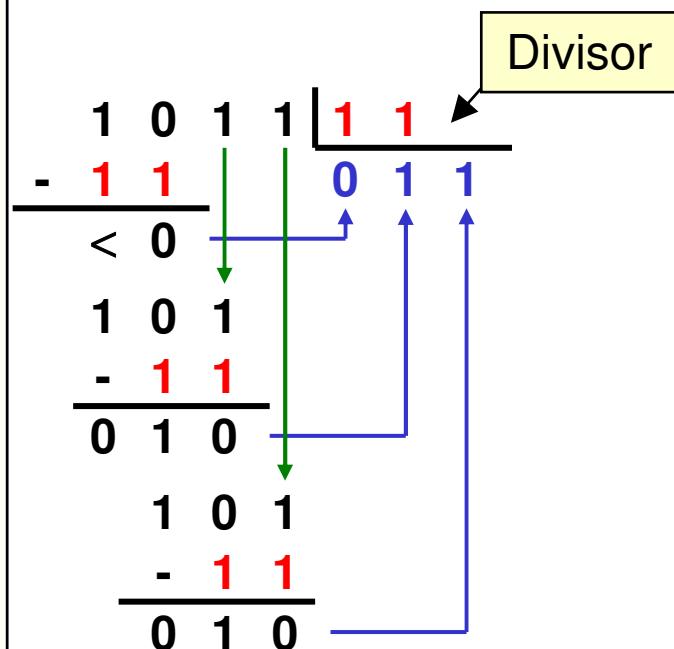


# Divisão de inteiros em binário

- Tal como acontecia com a multiplicação, também na divisão se usa uma arquitetura que aproveita o algoritmo que se ensina nos primeiros anos do ensino básico.
- Tomemos como exemplo  $1011 \div 0011$



# Divisão de inteiros em binário



1. Começa-se por alinhar o Divisor à esquerda com o Dividendo
2. **Subtrai-se o Divisor do Dividendo**
  - Se o resultado for **positivo** (i.e. Dividendo  $\geq$  Divisor) acrescenta-se "**1**" no Quociente
  - Se o resultado for **negativo** (i.e. Dividendo  $<$  Divisor) acrescenta-se "**0**" no Quociente e repõe-se o dividendo
3. Se o Divisor ainda não está alinhado à direita com o Dividendo, então desloca-se o Divisor 1 bit para a direita
4. Repete-se desde 2

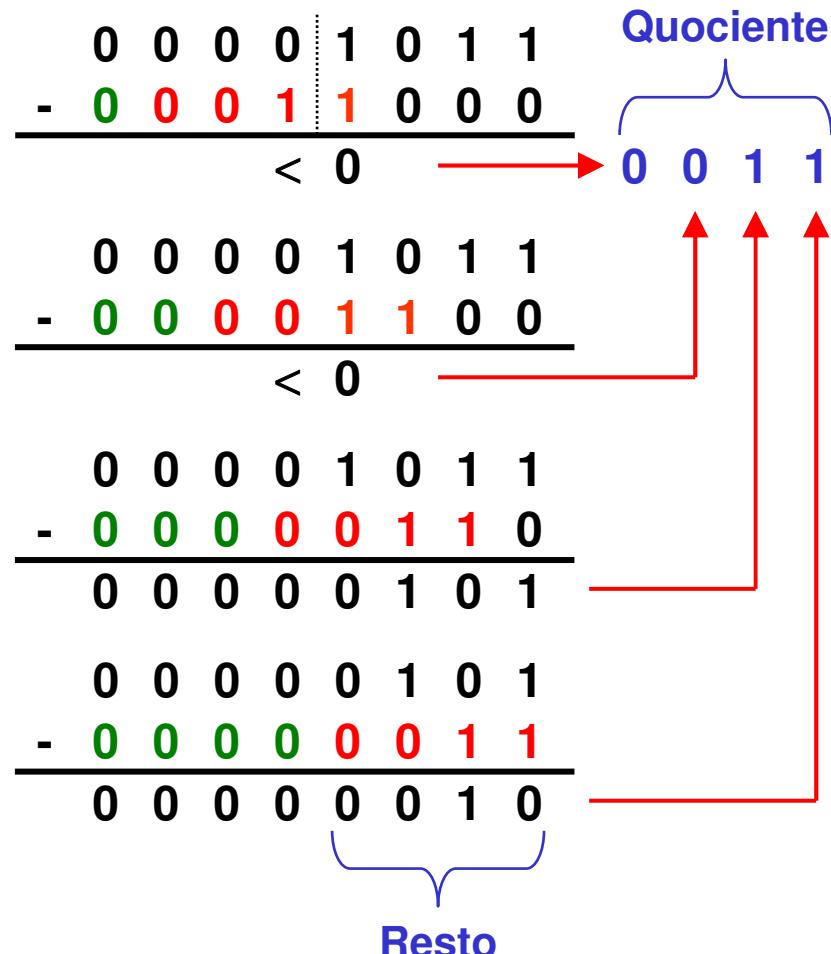
- Como fazer o alinhamento do divisor à esquerda de forma automática?
- Quantas iterações são necessárias, no caso geral, para fazer a divisão?



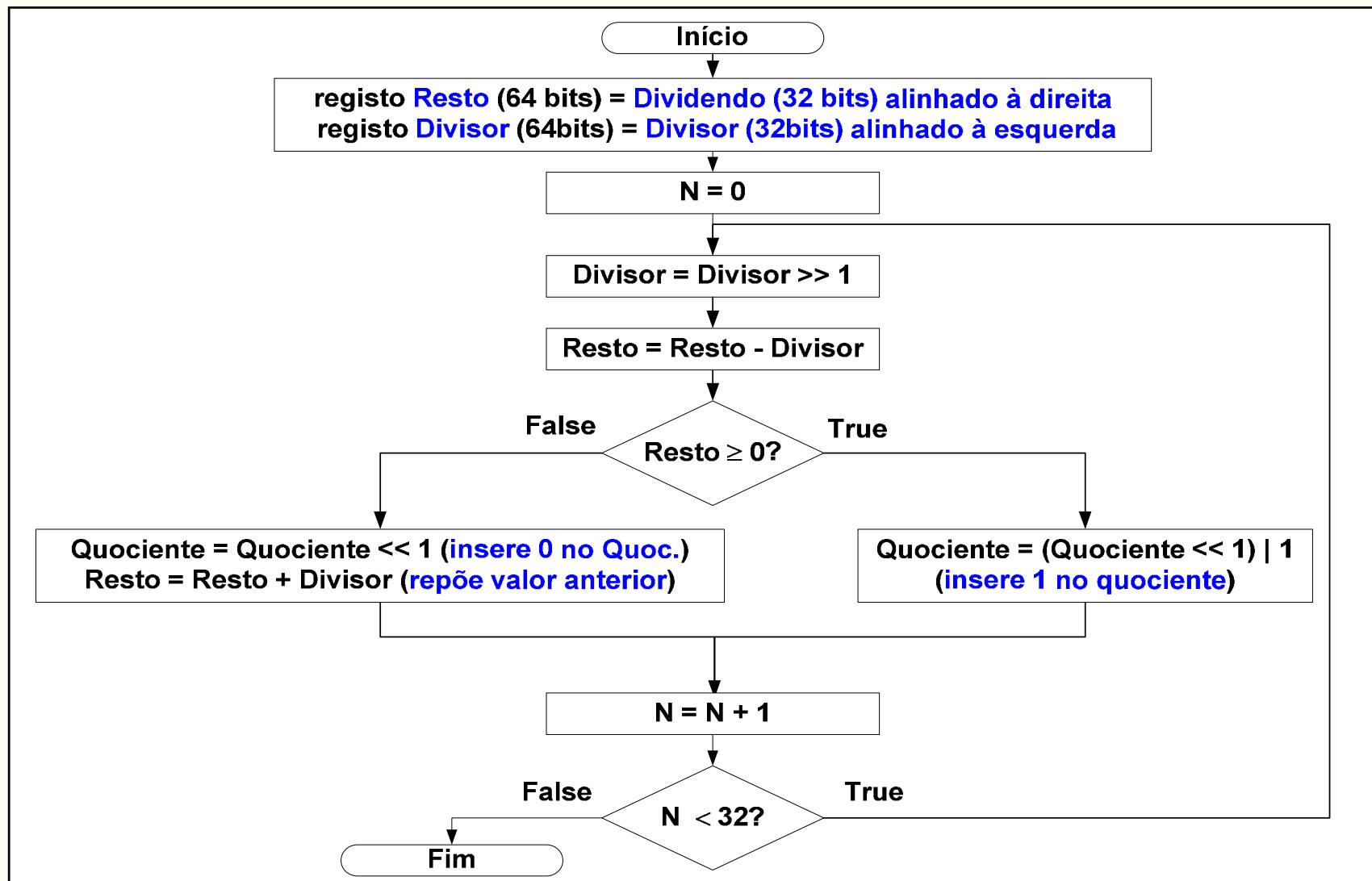
## Divisão de inteiros em binário – exemplo com 4 bits

1 0 1 1 | 0 0 1 1  
0 0 0 0 | 1 0 1 1      Dividendo  
0 0 1 1 | 0 0 0 0      Divisor

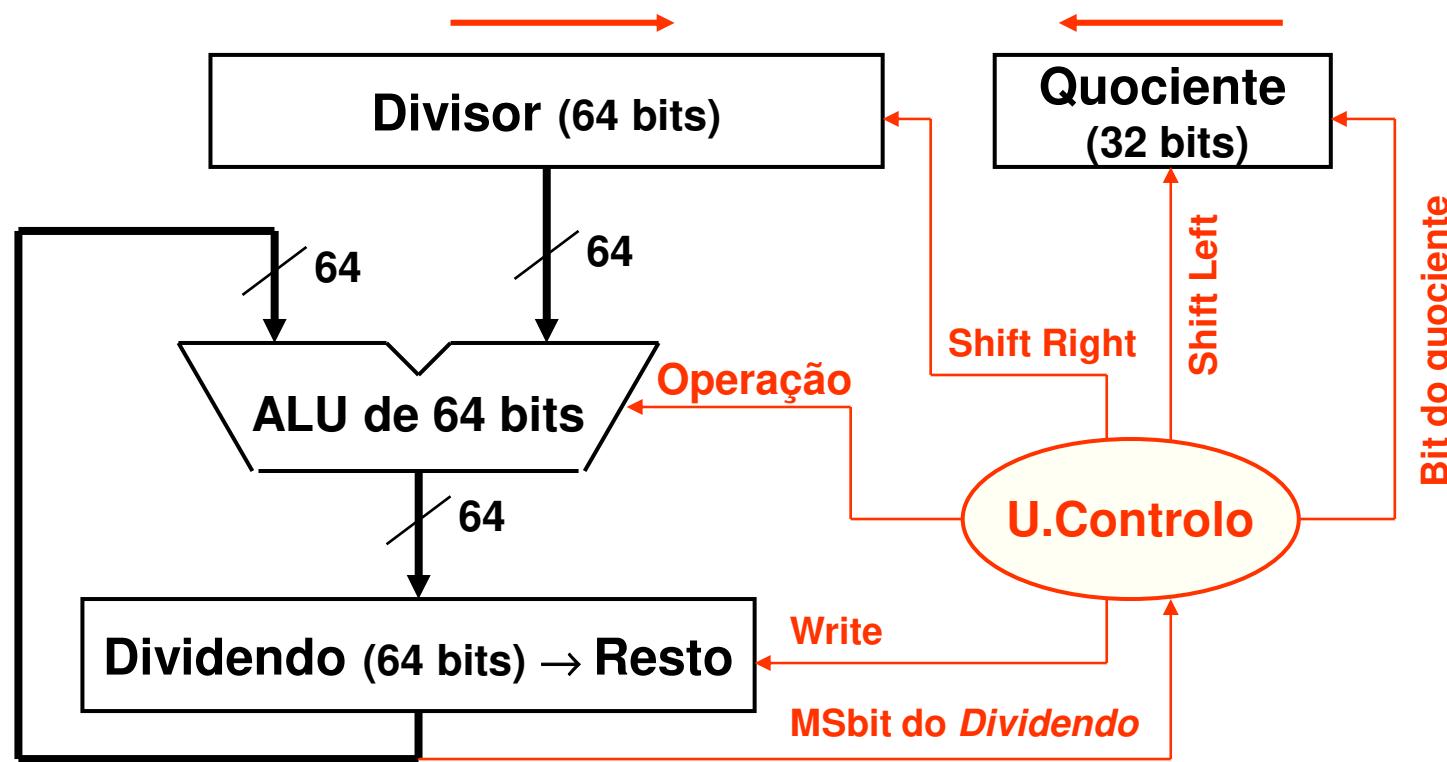
- Com operandos de 4 bits os registos para alojar o **dividendo e o divisor têm 8 bits**
- O **dividendo** é alinhado à **direita** (os 4 MSbits são colocados a 0)
- O **divisor** é alinhado à **esquerda** (os 4 LSbits são colocados a 0)
- Por cada nova iteração o **divisor é deslocado à direita 1 bit**
- O **número total de iterações** é igual ao **número de bits do dividendo original**



# Divisão de inteiros em binário – algoritmo



# Arquitetura de um Divisor de 32 bits (1<sup>a</sup> versão)



- Nesta 1<sup>a</sup> versão, a **ALU** e os registos **Divisor** e **Dividendo/Resto** operam com 64 bits.
- *O sinal de relógio não está representado*

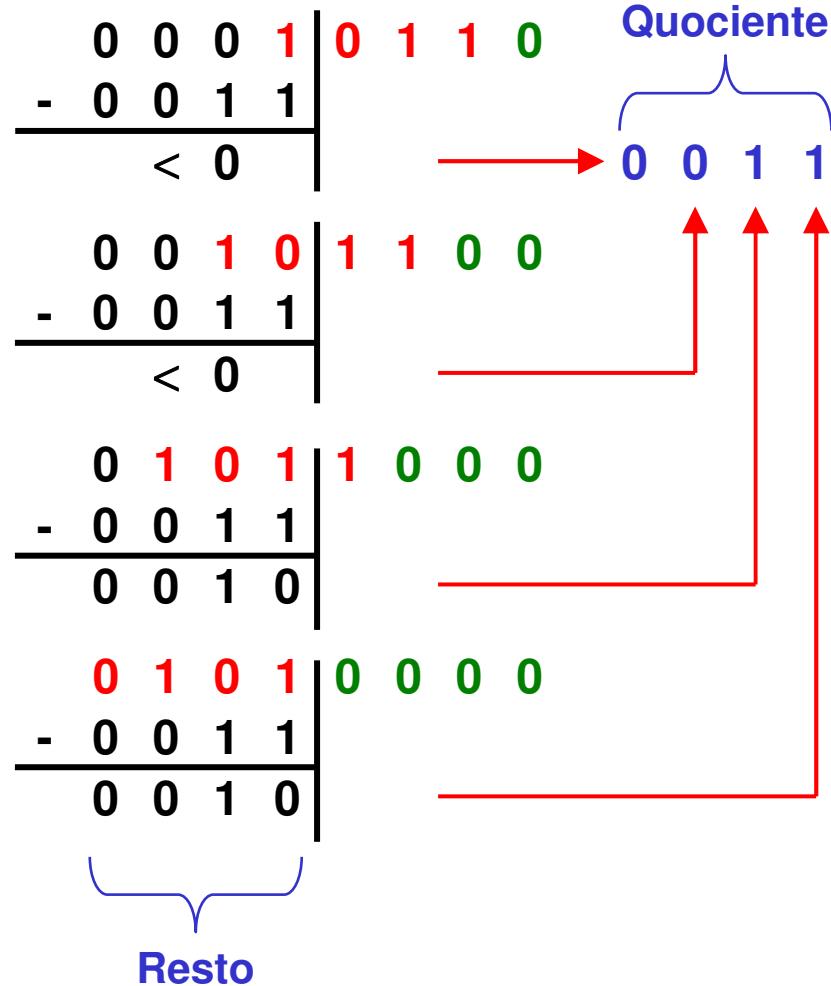


## Divisão de inteiros em binário – exemplo com 4 bits

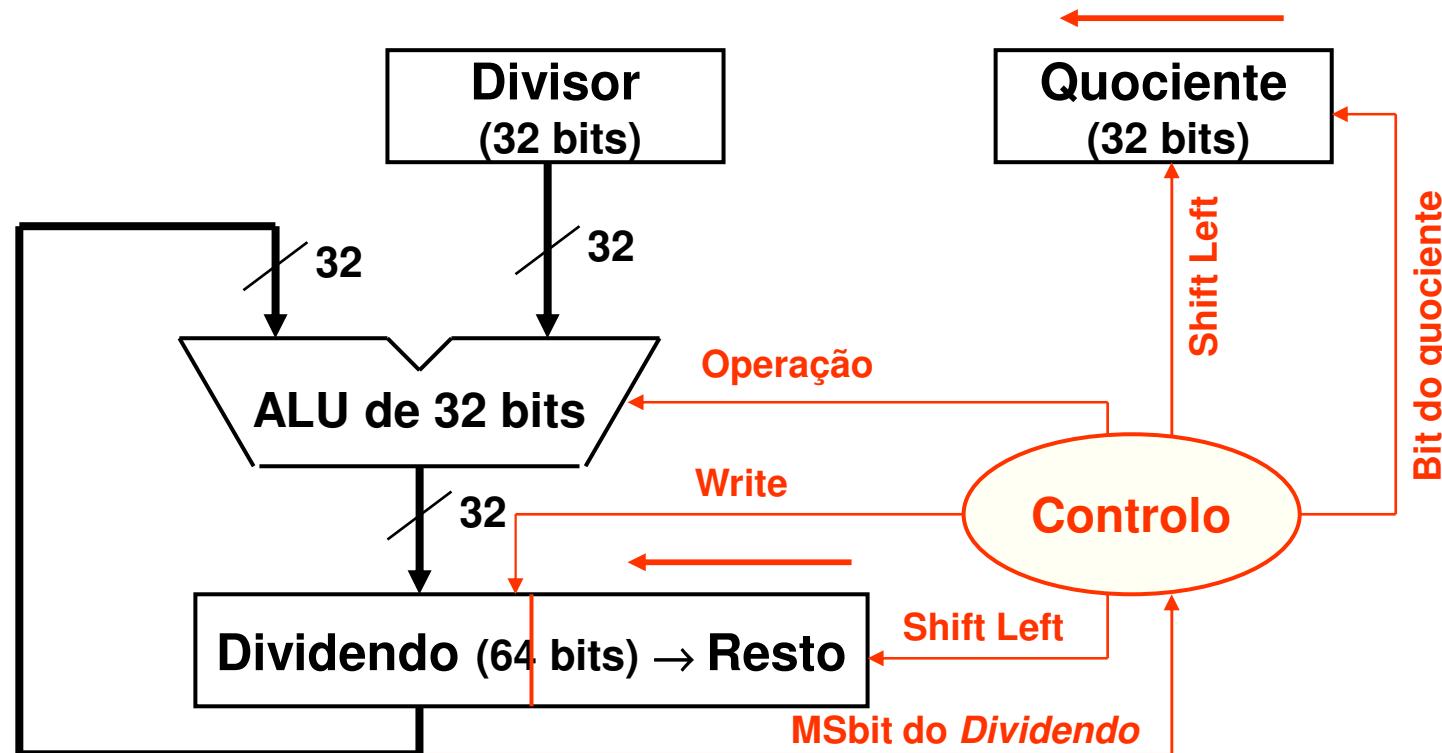
1 0 1 1 | 0 0 1 1

0 0 0 0 | 1 0 1 1      Dividendo  
0 0 1 1      Divisor

- O movimento relativo do **Dividendo/Resto** e do **Divisor** mantém-se se se fixar o *Divisor* e deslocar para a **esquerda** o **Dividendo/Resto**
- O registo *Divisor* mantém assim a dimensão original (4 bits no exemplo)
- A **subtração** (entre dividendo e divisor) pode também ser feita apenas com **4 bits**, reduzindo-se para metade a dimensão da ALU.



# Arquitetura de um Divisor de 32 bits (2<sup>a</sup> versão)



Nesta 2<sup>a</sup> versão do divisor, a **ALU** e o registo **Divisor** operam com 32 bits.

**1 0 1 1 | 0 0 1 1**

**0 0 0 0 1 0 1 1 Dividendo**

**0 0 0 1 0 1 1 0 Dividendo após << 1**  
**0 0 1 1 | Divisor**

- Pode verificar-se que o deslocamento à esquerda do conteúdo do *Dividendo*, é acompanhado por um deslocamento idêntico do *Quociente*
- Em cada deslocamento à esquerda do *Dividendo* é introduzido um zero (não útil) no bit menos significativo
- **Esse espaço pode ser aproveitado para guardar o próximo bit do quociente**
- Desta forma poupa-se ainda o espaço que seria necessário para armazenar esse quociente

$$\begin{array}{r} 0 0 0 1 | 0 1 1 0 \\ - 0 0 1 1 \\ \hline < 0 \end{array}$$

$$\begin{array}{r} 0 0 1 0 1 1 0 0 \\ \hline \end{array}$$

$$\begin{array}{r} 0 0 1 0 | 1 1 0 0 \\ - 0 0 1 1 \\ \hline < 0 \end{array}$$

$$\begin{array}{r} 0 1 0 1 1 0 0 0 \\ \hline \end{array}$$

$$\begin{array}{r} 0 1 0 1 | 1 0 0 0 \\ - 0 0 1 1 \\ \hline 0 0 1 0 \end{array}$$

$$\begin{array}{r} 0 1 0 1 0 0 0 1 \\ \hline \end{array}$$

$$\begin{array}{r} 0 1 0 1 | 0 0 0 1 \\ - 0 0 1 1 \\ \hline 0 0 1 0 \end{array}$$

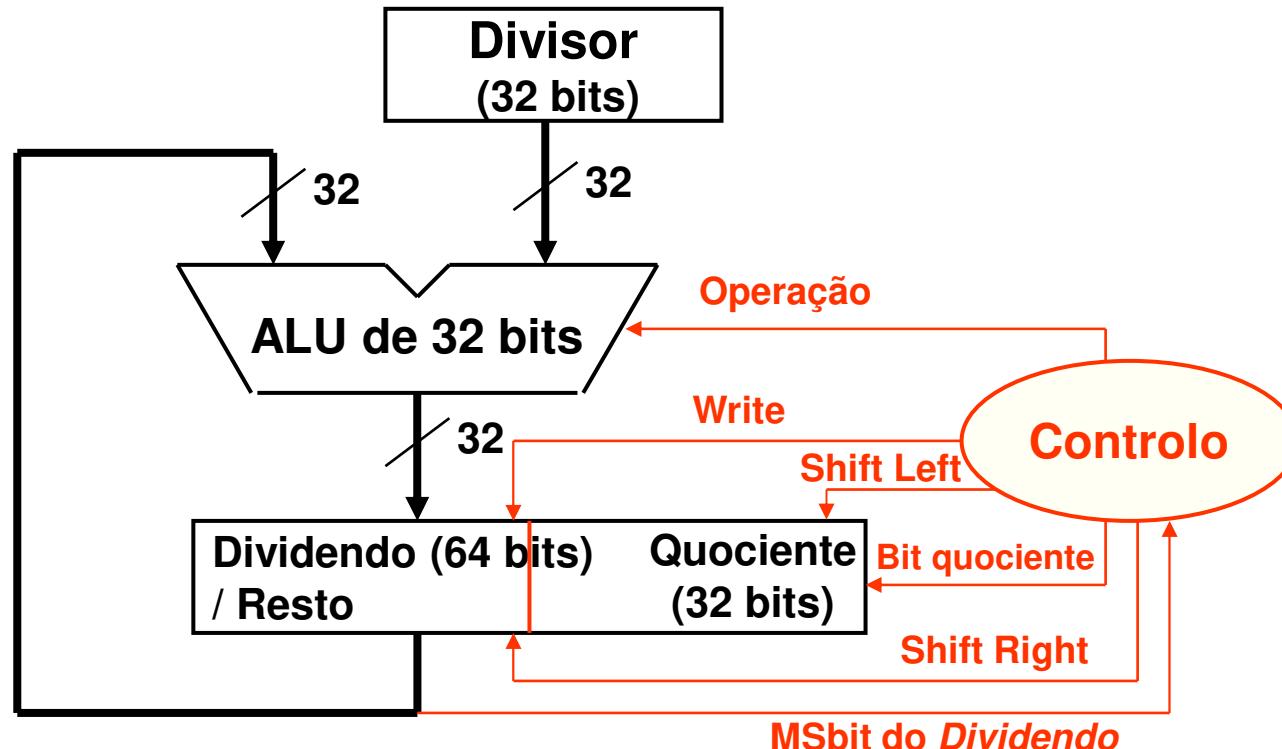
$$\begin{array}{r} 0 1 0 0 0 0 1 1 \\ \hline \end{array}$$

$$\begin{array}{r} 0 0 1 0 0 0 1 1 \\ \hline \end{array}$$

**Resto**

**Quociente**

# Arquitetura de um Divisor (versão final)



Na versão final do divisor, o registo **Quociente** desaparece, sendo substituído pela parte menos significativa do registo **Dividendo/Resto**.

# Divisão de inteiros com sinal

- A divisão de inteiros com sinal faz-se, do ponto de vista algorítmico, em sinal e módulo
- Nas divisões com sinal aplicam-se as seguintes regras:
  - Dividem-se dividendo por divisor em módulo
  - O quociente tem sinal negativo se os sinais de dividendo e divisor forem diferentes
  - O resto tem o mesmo sinal do dividendo
- Exemplos:

$$-7 / 3 = -2 \quad \text{resto} = -1$$

$$7 / -3 = -2 \quad \text{resto} = 1$$

- Notar que **Dividendo = Divisor \* Quociente + Resto**



# A Divisão de inteiros no MIPS

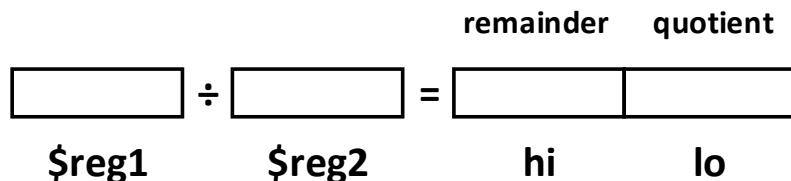
- No MIPS, a divisão é assegurada por uma arquitetura hardware semelhante à anteriormente descrita para a multiplicação (a unidade de controlo é que estabelece a diferença)
- Tal como acontecia na multiplicação, continua a existir a necessidade de um registo de 64 bits para armazenar o valor inicial do dividendo, e bem assim o resultado final na forma de um quociente e de um resto.
- Os mesmos registo, **HI** e **LO**, que tinham já sido apresentados para o caso da multiplicação, são igualmente utilizados para a divisão:
  - o registo **HI armazena o resto da divisão inteira**
  - o registo **LO armazena o quociente da divisão inteira**

$$\begin{array}{c} \text{remainder} \quad \text{quotient} \\ \boxed{\phantom{0}} \div \boxed{\phantom{0}} = \boxed{\phantom{0}} \boxed{\phantom{0}} \\ \text{op1} \qquad \text{op2} \qquad \text{hi} \qquad \text{lo} \end{array}$$



# A Divisão de inteiros no MIPS

- Em *Assembly*, a divisão é efetuada pela instrução
  - div      \$reg1, \$reg2** # Divide (signed)
  - divu    \$reg1, \$reg2** # Divide unsigned
- em que **\$reg1** é o dividendo e **\$reg2** o divisor. O **resultado** fica armazenado nos registos **HI (resto)** e **LO (quociente)**.



- A transferência de informação entre os registos HI e LO e os restantes registos de uso geral faz-se através das instruções **mfhi** e **mflo**:

**mfhi    \$reg**    # move from hi - Copia HI para \$reg  
**mflo    \$reg**    # move from lo - Copia LO para \$reg



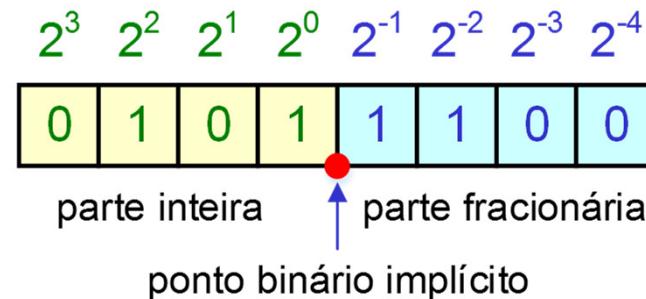
## Aulas 11 e 12

- Representação de números em vírgula flutuante
- A norma IEEE 754
  - Operações aritméticas em vírgula flutuante
  - Precisão simples e precisão dupla
  - Arredondamentos
- Unidade de vírgula flutuante do MIPS
  - Instruções da FPU do MIPS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira



# Representação de quantidades fracionárias



# Representação de quantidades fracionárias

- A representação de quantidades fracionárias em vírgula fixa coloca de imediato a questão da divisão do espaço de armazenamento para as partes inteira e fracionária
- O número de dígitos da parte inteira determina a **gama de valores representáveis**
- O número de dígitos da parte fracionária, determina a **precisão** da representação (no exemplo anterior, a menor quantidade representável é  $2^{-4} = 0,0625$ )
- No caso geral, quantos dígitos devem então ser reservados para a **parte inteira** e quantos para a **parte fracionária**, sabendo nós que o espaço de armazenamento é limitado?



# Representação de números em Vírgula Flutuante

- **Exemplo:** **-23.45129876** (vírgula fixa). A mesma quantidade pode também ser representada recorrendo à notação científica:

$$-2.345129876 \times 10^1 \quad -(2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3} + \dots + 6 \times 10^{-9}) \times 10^1$$

$$-0.2345129876 \times 10^2 \quad -(0 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} + \dots + 6 \times 10^{-10}) \times 10^2$$

- São representações do mesmo valor em que a posição da vírgula tem de ser ponderada, na interpretação numérica da quantidade, pelo valor do expoente de base 10
- Esta técnica, em que a vírgula pode ser deslocada sem alterar o valor representado, designa-se também por **representação em vírgula flutuante (VF)**
- A representação em VF tem a vantagem de não desperdiçar espaço de armazenamento com os zeros à esquerda da quantidade representada
- No primeiro exemplo, o número de dígitos diferentes de zero à esquerda da vírgula é igual a um: diz-se que a **representação está normalizada**



# Representação de números em Vírgula Flutuante

- A representação de quantidades em vírgula flutuante, em sistemas computacionais digitais, faz-se recorrendo à estratégia descrita nos slides anteriores, mas usando agora a base dois:

$$N = (+/-) 1.f \times 2^{\text{Exp}}$$

(representação **normalizada** de uma quantidade binária em vírgula flutuante)

- Em que:
  - f** – parte **fracionária** representada por **n** bits
  - 1.f** – **mantissa** (também designada por significando)
  - Exp** – **expoente** da potência de base 2 representado por **m** bits

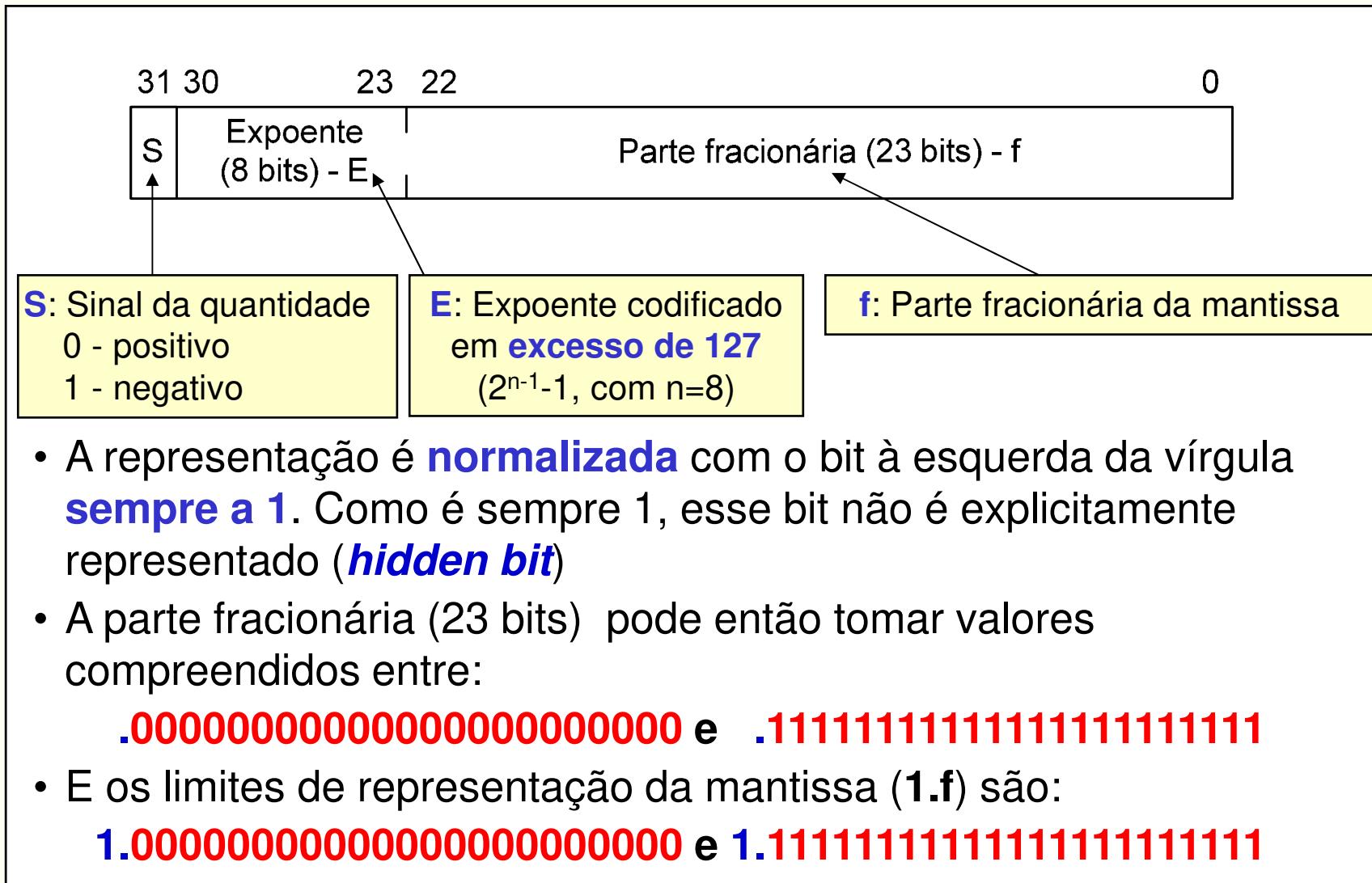


# Representação de números em Vírgula Flutuante

- O problema da divisão do espaço de armazenamento coloca-se também neste caso, mas agora na determinação do **número de bits** ocupados pela **parte fracionária** e pelo **expoente**
- Essa divisão é um **compromisso** entre **gama de representação** e **precisão**:
  - Aumento do número de bits da parte fracionária  $\Rightarrow$  maior precisão na representação
  - Aumento do número de bits do expoente  $\Rightarrow$  maior gama de representação
- Um bom design implica compromissos adequados!



# Norma IEEE 754 (precisão simples)



## Norma IEEE 754 (precisão simples)



- O expoente é codificado em **excesso de 127** ( $2^{n-1}-1$ , n=8 bits). Ou seja, é somado ao expoente verdadeiro (**Exp**) o valor 127 para obter o código de representação (i.e.  $E = Exp + 127$ , em que E é o expoente codificado)

$$N = (-1)^S \cdot 1.f \times 2^E = (-1)^S \cdot 1.f \times 2^{E-127}$$

- O código 127 representa, assim, o expoente zero; códigos maiores do que 127 representam expoentes positivos e códigos menores que 127 representam expoentes negativos
- **Os códigos 0 e 255 são reservados.** O expoente pode, desta forma, tomar valores entre **-126** e **+127** [códigos 1 a 254].



# Norma IEEE 754 (precisão simples)



**Exemplo:** 0 1000010 1011000000000000000000000 (0x41580000)

**Sinal** = 0 (quantidade positiva)

**Expoente** = 130 – offset = 130 – 127 = 3  $\Leftrightarrow$  (Exp = E – offset)

**Mantissa** = (1 + parte fracionária) = 1 + .1011 = 1.1011

A quantidade representada (R) se

$$\begin{aligned} R &= +1.1011 \times 2^3 = (1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}) \times 2^3 \\ &= 1.6875 \times 8 = 13.5 \end{aligned}$$

$$R = +1.1011 \times 2^3$$

$$R = 1101.1 \text{ (vírgula fixa)}$$

$$(i = 1101 = 13_{10} \quad f = 0.1 = 0.5_{10})$$

$$R = 13.5_{10}$$



# Norma IEEE 754 (precisão simples)

- **Exemplo:** codificar no formato vírgula flutuante IEEE 754 precisão simples, o valor  $-12.59375_{10}$

Parte inteira:  $12_{10} = 1100_2$

Parte fracionária:  $0.59375_{10} = 0.10011_2$

$$12.59375_{10} = 1100.10011_2 \times 2^0$$

$$\text{Normalização: } 1100.10011_2 \times 2^0 = 1.10010011_2 \times 2^3$$

$$\text{Expoente codificado: } +3 + 127 = 130_{10} = 10000010_2$$

**1 | 10000010 | 100100110000000000000000**

**0xC1498000**

MSb    ↓    LSb

$$\begin{array}{r} 0.59375 \\ \times 2 \\ \hline 1.18750 \\ 0.18750 \\ \times 2 \\ \hline 0.37500 \\ 0.37500 \\ \times 2 \\ \hline 0.75000 \\ 0.75000 \\ \times 2 \\ \hline 1.50000 \\ 0.50000 \\ \times 2 \\ \hline 1.00000 \end{array}$$

# Norma IEEE 754 (precisão simples)

- A gama de representação suportada por este formato será portanto:  
 $\pm [1.00000000000000000000000000000000 \times 2^{-126}, 1.111111111111111111111111111111 \times 2^{+127}]$   
 $\pm [1.175494 \times 10^{-38}, 3.402824 \times 10^{+38}]$
- Qual o número de dígitos à direita da vírgula na representação em decimal (casas decimais)?
- Partindo de uma representação com "**n**" dígitos fracionários na base "**r**", o número máximo de dígitos na base "**s**" que garante que a mudança de base não acrescenta precisão à representação original é:  
$$m = \left\lfloor n \frac{\log r}{\log s} \right\rfloor$$
       $\lfloor . \rfloor$  é o operador *floor*
- Assim, de modo a não exceder a precisão da representação original, a **representação em decimal** deve ter, no máximo, 6 casas decimais:  
$$m = \left\lfloor n \frac{\log r}{\log s} \right\rfloor = \left\lfloor 23 \frac{\log 2}{\log 10} \right\rfloor = 6$$
- Ou, sabendo que o nº de bits por casa decimal =  $\log_2(10) \approx 3.3$ , o número de casas decimais é  $\lfloor 23 / 3.3 \rfloor = 6$  **casas decimais**

# Norma IEEE 754 (precisão simples)



- Nas operações com quantidades representadas neste formato podem ocorrer situações de **overflow** e de **underflow**:
  - **Overflow**: quando o expoente do resultado não cabe no espaço que lhe está destinado →  $E > 254$ )  
 $N_{resultado} > 1.11111111111111111111111 \times 2^{+127}$
  - **Underflow**: caso em que o expoente é tão pequeno que também não é representável →  $E < 1$ )  
 $0 < N_{resultado} < 1.00000000000000000000000 \times 2^{-126}$



# Norma IEEE 754 – Adição / Subtração

Exemplo:  $N = 1.1101 \times 2^0 + 1.0010 \times 2^{-2}$

**1º Passo:** Igualar os expoentes ao maior dos expoentes

$$a = 1.1101 \times 2^0 \quad b = 0.010010 \times 2^0$$

**2º Passo:** Somar / subtrair as mantissas mantendo os expoentes

$$N = 1.1101 \times 2^0 + 0.010010 \times 2^0 = 10.000110 \times 2^0$$

**3º Passo:** Normalizar o resultado

$$N = 10.000110 \times 2^0 = 1.0000110 \times 2^1$$

**4º Passo:** Arredondar o resultado e renormalizar (se necessário)

$$N = 1.0000110 \times 2^1 = 1.0001 \times 2^1$$

**Exemplo com 4 bits fracionários**



# Norma IEEE 754 – Multiplicação

Exemplo:  $N = (1.1100 \times 2^0) \times (1.1001 \times 2^{-2})$

**1º Passo: Somar** os expoentes

$$\text{Exp. Resultado} = 0 + (-2) = -2$$

**2º Passo: Multiplicar** as mantissas

$$Mr = 1.1100 \times 1.1001 = 10.101111$$

**3º Passo: Normalizar** o resultado

$$N = 10.101111 \times 2^{-2} = 1.0101111 \times 2^{-1}$$

**4º Passo: Arredondar** o resultado e renormalizar (se necessário)

$$N = 1.0101111 \times 2^{-1} = 1.0110 \times 2^{-1}$$

**Exemplo com 4 bits fracionários**



# Norma IEEE 754 – Divisão

Exemplo:  $N = (1.0010 \times 2^0) / (1.1000 \times 2^{-2})$

**1º Passo: Subtrair os expoentes**

$$\text{Exp. Resultado} = 0 - (-2) = 2$$

**2º Passo: Dividir as mantissas**

$$M_r = 1.0010 / 1.1000 = 0.11$$

**3º Passo: Normalizar o resultado**

$$N = 0.11 \times 2^2 = 1.1 \times 2^1$$

**4º Passo: Arredondar o resultado**

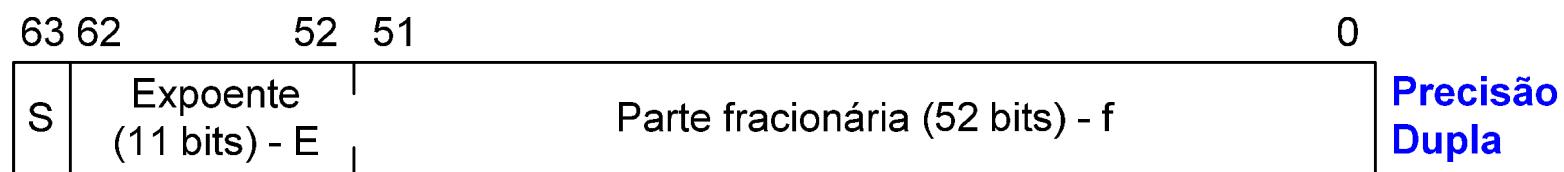
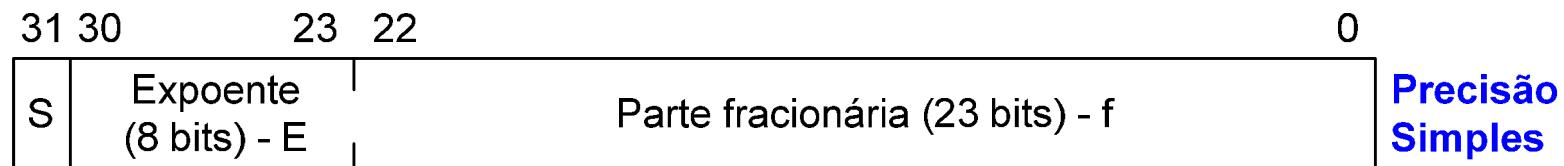
$$N = 1.1 \times 2^1 = 1.1000 \times 2^1$$

**Exemplo com 4 bits fracionários**



# Norma IEEE 754 (precisão dupla)

- A norma IEEE 754 suporta a representação de quantidades em **precisão simples (32 bits)** e em **precisão dupla (64 bits)**



$$N = (-1)^S \cdot 1.f \times 2^{(E - 127)} \quad (\text{Precisão simples - tipo float})$$

$$N = (-1)^S \cdot 1.f \times 2^{(E - 1023)} \quad (\text{Precisão dupla - tipo double})$$



## Norma IEEE 754 (precisão dupla)



$$N = (-1)^S \cdot 1.f \times 2^{\text{Exp}} = (-1)^S \cdot 1.f \times 2^{E-1023}$$

- A gama de representação suportada pelo formato de precisão dupla será:  
 $\pm [1.000000000000000...000 \times 2^{-1022}, 1.111111111111111...111 \times 2^{+1023}]$   
 $\pm [2.225073858507201 \times 10^{-308}, 1.797693134862316 \times 10^{+308}]$
- De modo a não exceder a precisão da representação original, a **representação em decimal** deve ter, no máximo,  $\lfloor 52 / \log_2(10) \rfloor = 15$  casas decimais



## Norma IEEE 754 – casos particulares

- A norma IEEE 754 suporta ainda a representação de alguns casos particulares:
  - A **quantidade zero**; essa quantidade não seria representável de acordo com o formato descrito até aqui
  - **+/-infinito**. Exemplos:  $1.0 / 0.0$ ,  $-1.0 / 0.0$
  - Resultados não numéricos (**NaN – Not a Number**). Exemplo:  $0.0 / 0.0$
  - Por forma a aumentar a resolução (menor quantidade representável) é ainda possível usar um formato de **mantissa desnormalizada** no qual o bit à esquerda do ponto binário é zero



# Norma IEEE 754 – casos particulares

| Precisão Simples |             | Precisão Dupla |             | Representa                             |
|------------------|-------------|----------------|-------------|--|
| Expoente         | Parte Frac. | Expoente       | Parte Frac. |  |
| 0                | 0           | 0              | 0           | 0                                      |
| 0                | $\neq 0$    | 0              | $\neq 0$    | Quantidade<br>desnormalizada           |
| 1 a 254          | qualquer    | 1 a 2046       | qualquer    | Nº em vírgula flutuante<br>normalizado |
| 255              | 0           | 2047           | 0           | Infinito                               |
| 255              | $\neq 0$    | 2047           | $\neq 0$    | NaN (Not a Number)                     |



# Norma IEEE 754 – representação desnormalizada

- Permite a representação de quantidades cada vez mais pequenas (*underflow gradual*)
- A gama de representação suportada pelo formato de mantissa desnormalizada, em precisão simples, é:

$$\pm [0.00000000000000000000000000000001 \times 2^{-126}, 0.11111111111111111111111111111111 \times 2^{-126}]$$

$$\pm [1 \times 2^{-23} \times 2^{-126}, 1.0 \times 2^{-126}[$$

$$\pm [1.401299 \times 10^{-45}, 1.175494 \times 10^{-38}[$$



# Técnicas de arredondamento do resultado

- As operações aritméticas são efetuadas com um número de bits da parte fracionária superior ao disponível no espaço de armazenamento
- Desta forma, na conclusão de qualquer operação aritmética é necessário proceder ao arredondamento do resultado por forma a assegurar a sua adequação ao espaço que lhe está destinado
- As técnicas mais comuns no processo de **arredondamento do resultado** (o qual introduz um erro) são:
  - Truncatura
  - Arredondamento simples
  - Arredondamento para o par (ímpar) mais próximo



# Técnicas de arredondamento do resultado

- **Truncatura** (exemplo com 2 dígitos na parte fracionária: d=2)

| Número | Trunc(x) | Erro |
|--------|----------|------|
| x.00   | x        | 0    |
| x.01   | x        | -1/4 |
| x.10   | x        | -1/2 |
| x.11   | x        | -3/4 |

$$\begin{aligned}\text{Erro médio} &= (0 - 1/4 - 1/2 - 3/4) / 4 \\ &= -3/8\end{aligned}$$

- Mantém-se a parte inteira, desprezando qualquer informação que exista à direita do ponto binário



# Técnicas de arredondamento do resultado

- **Arredondamento simples** (exemplo com 2 dígitos na parte fracionária:  $d=2$ )

| Número | Arred(x) | Erro |
|--------|----------|------|
| x.00   | x        | 0    |
| x.01   | x        | -1/4 |
| x.10   | x + 1    | +1/2 |
| x.11   | x + 1    | +1/4 |

$$\begin{aligned}\text{Erro médio} &= (0 - 1/4 + 1/2 + 1/4) / 4 \\ &= +1/8\end{aligned}$$

- Mantém-se a parte inteira quando o 1º dígito à direita do ponto binário for 0 ou soma-se “1” à parte inteira quando aquele for “1”:

$$\text{arred}(x) = \text{trunc}(x + 0.5)$$

- O erro médio é mais próximo de zero do que no caso da truncatura, mas ligeiramente polarizado do lado positivo



# Técnicas de arredondamento do resultado

- **Arredondamento para o par mais próximo** (exemplo com 2 dígitos na parte fracionária:  $d=2$ )

| Número       | Arred(x)  | Erro        | Número       | Arred(x)      | Erro        |
|--------------|-----------|-------------|--------------|---------------|-------------|
| x0.00        | x0        | 0           | x1.00        | x1            | 0           |
| x0.01        | x0        | -1/4        | x1.01        | x1            | -1/4        |
| <b>x0.10</b> | <b>x0</b> | <b>-1/2</b> | <b>x1.10</b> | <b>x1 + 1</b> | <b>+1/2</b> |
| x0.11        | X0 + 1    | +1/4        | x1.11        | x1 + 1        | +1/4        |

- Semelhante à técnica de arredondamento, mas decidindo, para o caso “**xx.10**”, em função do primeiro dígito à esquerda do ponto binário
- **Erro médio** =  $(0 - 1/4 - 1/2 + 1/4) / 4 + (0 - 1/4 + 1/2 + 1/4) / 4$   
 $= -1/8 + 1/8 = 0$



# Técnicas de arredondamento do resultado

| O que fica à direita de $b_{23}$ | Exemplo  | Resultado   |
|----------------------------------|--|---|
| < 0.5                            | $1.b_1b_2 \dots b_{22}b_{23} \textcolor{red}{0111}$              | <i>Round down</i> : bits à direita de $b_{23}$ são descartados      |
| > 0.5                            | $1.b_1b_2 \dots b_{22}b_{23} \textcolor{red}{1001}$              | <i>Round up</i> : soma-se 1 a $b_{23}$ (propagando o carry)         |
| = 0.5                            | $1.b_1b_2 \dots b_{22}\textcolor{blue}{1} \textcolor{red}{1000}$ | <i>Round up</i> : soma-se 1 a $b_{23}$ (propagando o carry) (*)     |
| = 0.5                            | $1.b_1b_2 \dots B_{22}\textcolor{blue}{0} \textcolor{red}{1000}$ | <i>Round down</i> : bits à direita de $b_{23}$ são descartados (*)  |
| = 0.5                            | $1.b_1b_2 \dots B_{22}\textcolor{blue}{1} \textcolor{red}{1000}$ | <i>Round down</i> : bits à direita de $b_{23}$ são descartados (**) |
| = 0.5                            | $1.b_1b_2 \dots b_{22}\textcolor{blue}{0} \textcolor{red}{1000}$ | <i>Round up</i> : soma-se 1 a $b_{23}$ (propagando o carry) (**)    |

(\*) Arredondamento para o **par mais próximo**.

(\*\*) Arredondamento para o **ímpar mais próximo**.



## Norma IEEE 754 – arredondamentos

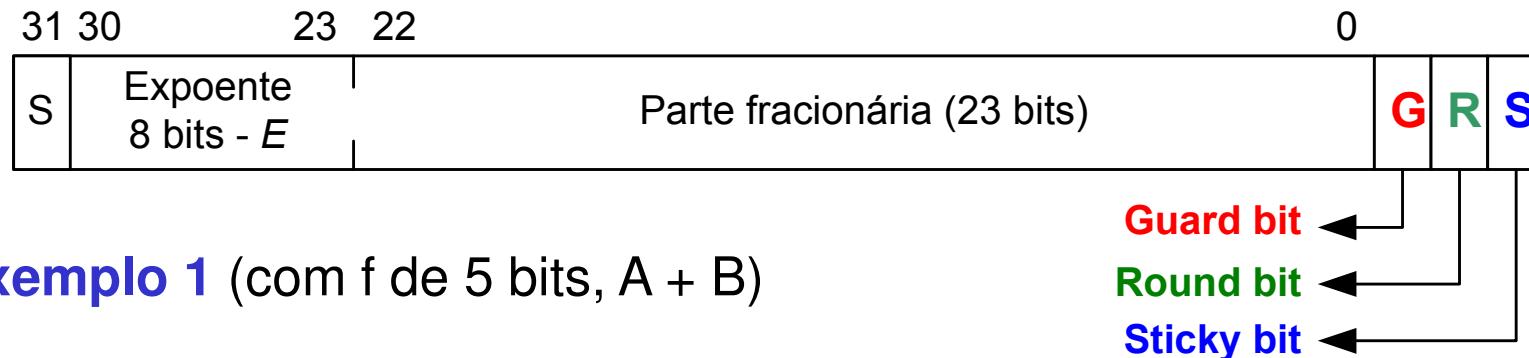
- Os valores resultantes de cada fase intermédia da execução de uma operação aritmética são armazenados com três bits adicionais, à direita do bit menos significativo da mantissa (i.e., para o caso de precisão simples, com pesos  $2^{-24}$ ,  $2^{-25}$  e  $2^{-26}$ )

? .????????????????????????? **G R S** (f c/ 26 bits)

- Objetivos: 1) minimizar o erro introduzido pelo processo de arredondamento e 2) ter bits suplementares para a pós-normalização
- **G – Guard Bit**
- **R – Round bit**
- **S – Sticky bit** – Bit que resulta da soma lógica de todos os bits à direita do bit R (i.e., se houver à direita de R pelo menos 1 bit a ‘1’, então S=‘1’)



# Norma IEEE 754 – arredondamentos



**Exemplo 1** (com f de 5 bits, A + B)

$$A = 1.11010 \times 2^0 \quad B = 1.00100 \times 2^{-2}$$

$$B = 0.0100100 \times 2^0 \text{ (igualar ao maior dos expoentes)}$$

$$\begin{aligned} \text{Mant}(A+B) &= 1.11010 + 0.0100100 & \text{Expoente}(A+B) &= 0 \\ &= 10.00011 \text{ } \color{blue}{000} & \text{G} &= 0, \text{R} &= 0, \text{S} &= 0 \end{aligned}$$

$$\text{Mant}(A+B)_{\text{norm}} = 1.00001 \boxed{100} \quad \text{G} = 1, \text{R} = 0, \text{S} = 0$$

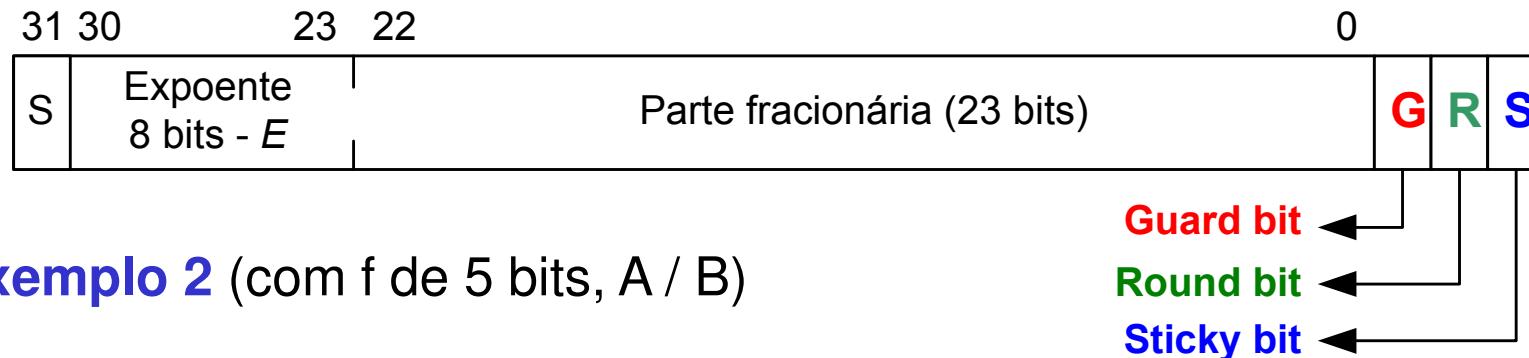
**Arredondamento:**

Mant(A + B) = 1.00010, se arred. para o par mais próximo ( $R=1.00010 \times 2^1$ )

Mant(A + B) = 1.00001, se arred. para o ímpar mais próximo ( $R=1.00001 \times 2^1$ )



# Norma IEEE 754 – arredondamentos



**Exemplo 2** (com f de 5 bits, A / B)

$$A = 1.00001 \times 2^2 \quad B = 1.11111 \times 2^{-1}$$

$$\text{Mant}(A/B) = 1.00001 / 1.11111 \quad \text{Expoente}(A/B) = 2 - (-1) = 3$$

$$= 0.10000 \textcolor{red}{1}100001 \quad G = 1, R = 1, S = \text{OR}(00001) = 1$$

$$= 0.10000 \textcolor{red}{1}11$$

$$\text{Mant}(A/B)_{\text{norm}} = 1.00001 \textcolor{red}{1}10$$

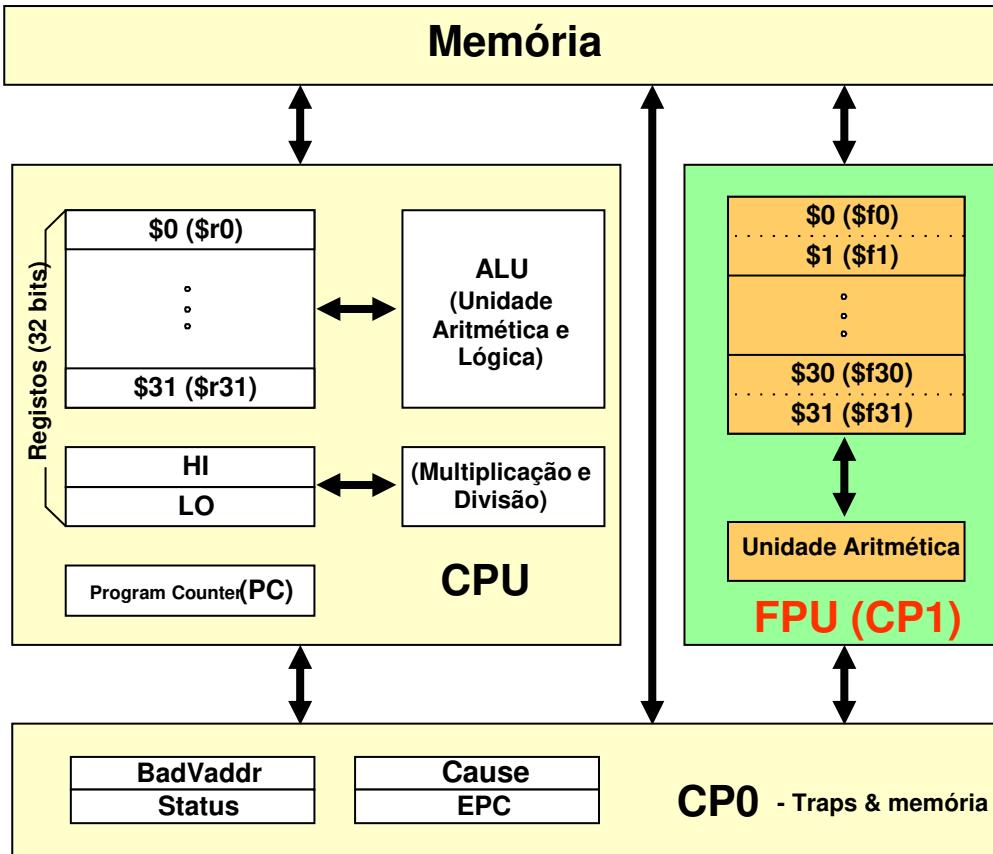
$$\text{Arred}(1, 11_2) = 10_2$$

$$\text{Arredondamento} \Rightarrow \text{Mant}(A/B) = 1.00010$$

$$A/B = 1.00010 \times 2^2$$



# Cálculo em Vírgula Flutuante no MIPS



- O MIPS inclui um coprocessador aritmético (Coprocessador 1) capaz de efetuar operações aritméticas em vírgula flutuante, usando a norma IEEE 754
- Esse coprocessador tem o seu próprio espaço de armazenamento composto por um conjunto de 32 registros de 32 bits cada, e o seu próprio conjunto de instruções (ISA)



## Vírgula Flutuante no MIPS – registos

- Os registos do coprocessador aritmético são designados, no *Assembly* do MIPS, pelas letras **\$fn**, em que o índice **n** toma valores entre 0 e 31
- Cada par de registos consecutivos **[\$fn,\$fn+1]** (**com n par**) pode funcionar como um registo de 64 bits para armazenar valores em **precisão dupla**.
- Em *Assembly* a referência ao par de registos faz-se indicando como operando o **registro par**
- **Apenas os registos de índice par** podem ser usados no contexto das instruções



# Vírgula Flutuante no MIPS – instruções aritméticas

|              |  |                         |
|--------------|--|-------------------------|
| <b>abs.p</b> | <b>FPdst , FPs<sub>rc</sub></b>                      | <b># Absolute Value</b> |
| <b>neg.p</b> | <b>FPdst , FPs<sub>rc</sub></b>                      | <b># Negate</b>         |
| <b>div.p</b> | <b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b> | <b># Divide</b>         |
| <b>mul.p</b> | <b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b> | <b># Multiply</b>       |
| <b>add.p</b> | <b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b> | <b># Addition</b>       |
| <b>sub.p</b> | <b>FPdst , FPs<sub>rc1</sub> , FPs<sub>rc2</sub></b> | <b># Subtract</b>       |

O sufixo **.p** representa a **precisão** com que é efetuada a operação (simples ou dupla). Deverá, na instrução, ser substituído pelas letras **.s** ou **.d** respetivamente.



# Vírgula Flutuante no MIPS – conversão entre tipos

|                     |                             |
|---------------------|-----------------------------|
| cvt.d.s FPdst,FPsrc | # Convert Single to Double  |
| cvt.d.w FPdst,FPsrc | # Convert Integer to Double |
| cvt.s.d FPdst,FPsrc | # Convert Double to Single  |
| cvt.s.w FPdst,FPsrc | # Convert Integer to Single |
| cvt.w.d FPdst,FPsrc | # Convert Double to Integer |
| cvt.w.s FPdst,FPsrc | # Convert Single to Integer |

Formato do resultado

Formato original

As **conversões** entre tipos de representação **são efetuadas pela FPU** pelo que apenas podem ter como operandos/destinos registos da FPU



## Conversão entre tipos – exemplos

```
$f0=0xC0D00000 = 11000000110100000000000000000000  
= 11000000110100000000000000000000  
= -1.625 x 22 = -6.5
```

**cvt.d.s \$f6,\$f0**

Exp = (129-127) + 1023 = 1025 = 1000000001

\$f6=0x00000000 \$f7=1 1000000001 101000...0

\$f6=0x00000000 \$f7=0xC01A0000

**cvt.w.s \$f8,\$f0**

Exp = (129-127) = 2

Val = -1.625 x 2<sup>2</sup> = -6.5, (int) (-6.5) = -6

\$f8=0xFFFFFFF8



# Vírgula Flutuante no MIPS – instruções de transferência

- **Transferência de informação** entre regtos do CPU e da FPU, e entre regtos da FPU



```
mtc1  CPUSrc,FPdst    # Move to Coprocessor 1
mfc1  CPUDst,FPSrc   # Move from Coprocessor 1
mov.s  FPdst, FPSrc   # Move from FPSrc to FPdst (single)
mov.d  FPdst, FPSrc   # Move from FPSrc to FPdst (double)
```

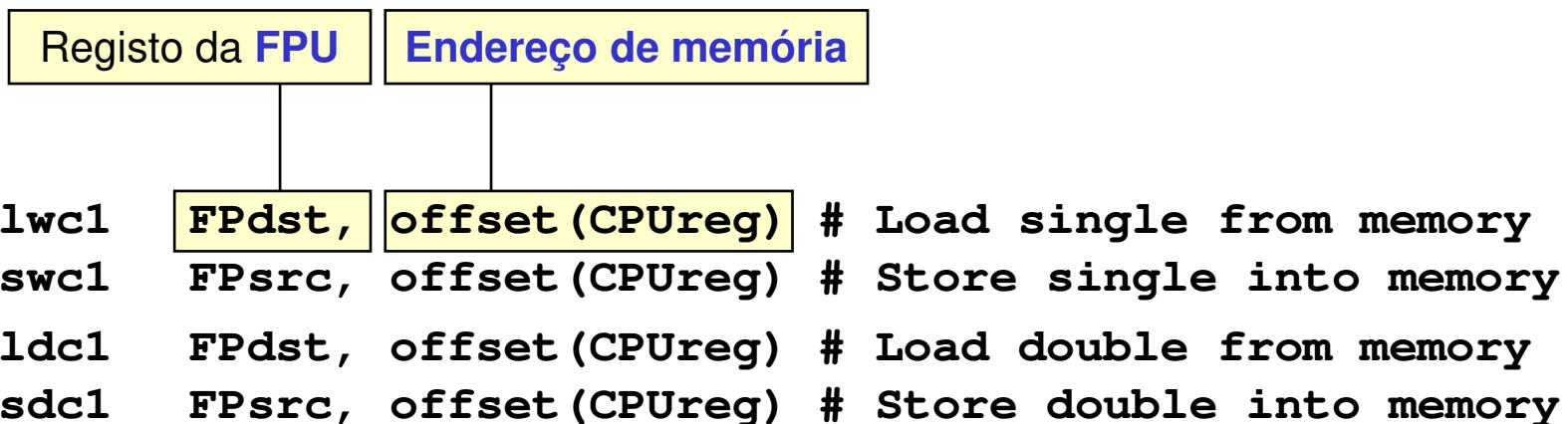
Estas instruções copiam o conteúdo integral do registo fonte para o registo destino.

**Não efetuam qualquer tipo de conversão entre tipos de informação.**



# Vírgula Flutuante no MIPS – instruções de transferência

- Transferência de informação entre registos da FPU e a memória



Instruções virtuais (apenas muda a mnemónica):

- l.s** **FPdst, offset (CPUREg)** # Load single from memory
- s.s** **FPSrc, offset (CPUREg)** # Store single into memory
- l.d** **FPdst, offset (CPUREg)** # Load double from memory
- s.d** **FPSrc, offset (CPUREg)** # Store double into memory



# Vírgula Flutuante no MIPS – instruções de decisão

- A tomada de decisões envolvendo quantidades em vírgula flutuante realiza-se de forma distinta da utilizada para o mesmo tipo de operação envolvendo quantidades inteiras
- Para quantidades em vírgula flutuante são necessárias duas instruções em sequência: uma **comparação das duas quantidades, seguida da decisão** (que usa a informação produzida pela comparação):
  - A instrução de comparação coloca a **True** ou **False** uma *flag* (1 bit), dependendo de a condição em comparação ser verdadeira ou falsa, respetivamente
  - Em **função do estado dessa flag** a instrução de decisão (instrução de salto) pode alterar a sequência de execução



# Cálculo em Vírgula Flutuante no MIPS

- Instruções de comparação:

```
c.x.s    FPUREG1, FPUREG2  # compare single  
c.x.d    FPUREG1, FPUREG2  # compare double
```

Em que **X** pode ser uma das seguintes condições:

**EQ** – **equal**

**LT** – **less than**

**LE** – **less or equal**

- Instruções de salto:

```
bc1t    # branch if true
```

```
bc1f    # branch if false
```



# Vírgula Flutuante no MIPS – instruções de decisão

```
float a, b;  
...  
  
if( a > b)  
    a = a + b;  
else  
    a = a - b;
```

```
# $f0 ← a  
# $f2 ← b  
...  
if:  c.le.s $f0, $f2          # if(a > b)  
     bc1t else                 # {  
     add.s  $f0, $f0, $f2      #     a = a + b;  
     j      endif               # }  
                                # else  
else: sub.s $f0, $f0, $f2    #     a = a - b;  
endif:...
```



# Convenções quanto à utilização de registos

- Registos para **passar parâmetros** para funções:
  - \$f12 (\$f13), \$f14 (\$f15)
- Registos para **devolução de resultados** das funções:
  - \$f0 (\$f1), \$f2 (\$f3)
- Registos para **caller saved** não preservados pelas funções:
  - \$f4 (\$f5) a \$f18 (\$f19)
- Registos para **callee saved** preservados pelas funções:
  - \$f20 (\$f21) a \$f30 (\$f31)



# Diretivas do Assembler em vírgula flutuante

```
.data  
label1: .float f1.s1, f2.s2, ...  
label2: .double d1.f1, d2.f2, ...
```

Nota: “label2” tem de ser um endereço múltiplo de 8

**.float** -> aloca espaço no segmento de dados e inicializa esse espaço com a lista de valores float (precisão simples) indicado

**.double** -> aloca espaço no segmento de dados e inicializa esse espaço com a lista de valores double (precisão dupla) indicado



# Cálculo em Vírgula Flutuante no MIPS – Exemplo

```
double average(double *, int);

void main(void)
{
    static double array[SIZE];
    double avg;
    ...
    avg = average( array, SIZE );
    printDouble( avg );      // syscall 3
}

double average(double *v, int N)
{
    double av = 0.0;
    int i;

    for(i = 0; i < N; i++)
        av += v[i];
    return av / (double)N;
}
```



# Tradução C / Assembly

```
void main(void)
{
    static double array[SIZE];
    double avg;
    ...
    avg = average( array, SIZE );
    printDouble( avg );           // syscall 3
}
```

```
.data
array: .space n          # n = 8 * SIZE
.text
.globl main
main: ...
        # void main(void) {
        la    $a0, array      #
        li    $a1, SIZE       #
        jal   average        #
        mov.d $f12, $f0        #     avg = average(array, SIZE)
        li    $v0, 3           #
        syscall                #     print_double(avg)
        ...
        jr    $ra             # }
```



# Tradução C / Assembly

```
double average(double *v, int N)
{
    double av = 0.0;
    int i;
    for(i = 0; i < N; i++)
        av += v[i];
    return av / (double)N;
}
```

```
# $f0 ← av
func: mtc1    $0,  $f0          #
       cvt.d.w $f0,  $f0          # av = 0.0
       li      $t0,  0            # i = 0
for:   bge     $t0,  $a1, endf # while(i < N) {
       sll     $t1,  $t0,  3          #
       addu   $t1,  $t1,  $a0          # $t1 = &v[i]
       l.d    $f4,  0($t1)          # $f0 = v[i]
       add.d   $f0,  $f0,  $f4          # av += v[i]
       addi   $t0,  $t0,  1            # i++
       j      for                  # }
endf:  mtc1    $a1,  $f4          #
       cvt.d.w $f4,  $f4          # $f4 = (double)N
       div.d   $f0,  $f0,  $f4          #
       jr     $ra                  #
```



## Aulas 13, 14 e 15

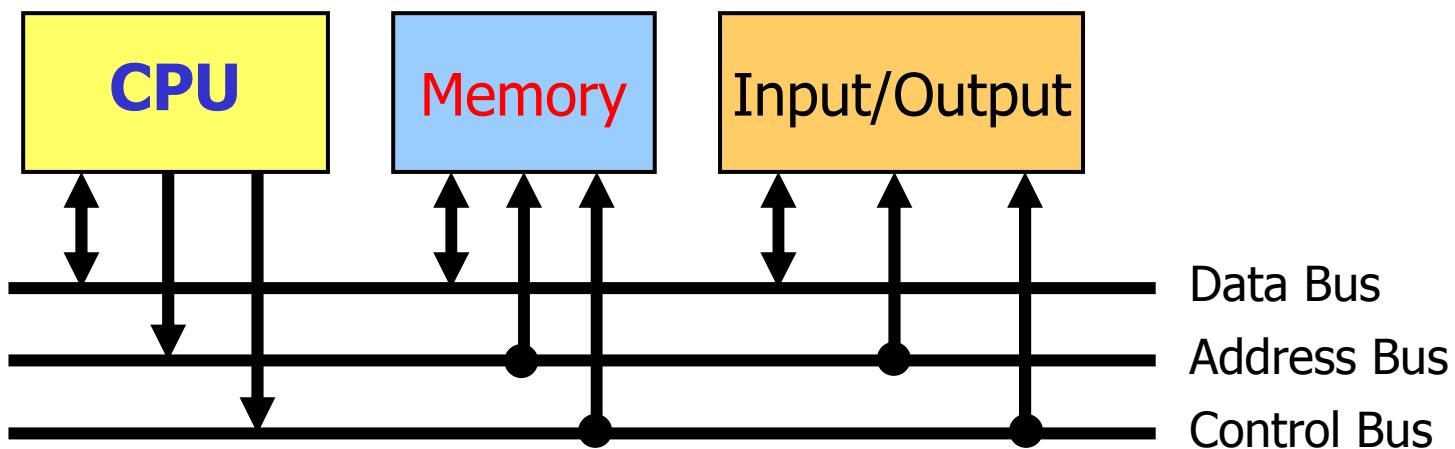
- Modelos de Harvard e Von Neumann
- Pressupostos para a construção de um *datapath* genérico para uma arquitetura tipo MIPS
- Análise dos blocos constituintes necessários à execução de um subconjunto de instruções de cada uma das classes de instruções:
  - Aritméticas e lógicas (add, addi, sub, and, or, slt, slti)
  - Acesso à memória (lw, sw)
  - Controlo de fluxo de execução (beq, bne, j)
- Montagem de um *datapath* completo para execução de instruções num único ciclo de relógio (*single-cycle*)

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Modelo de von Neumann

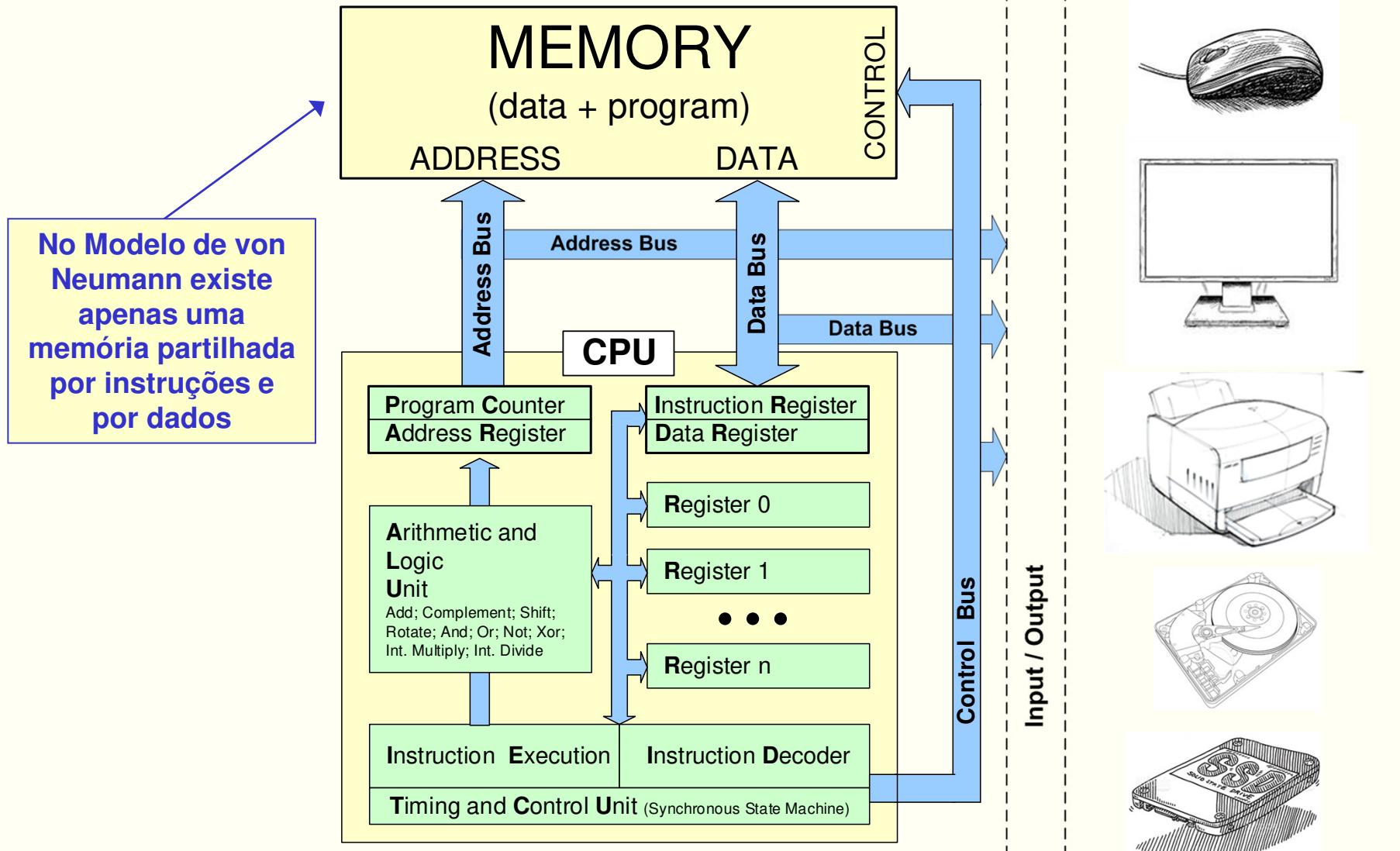
Datapath + Control



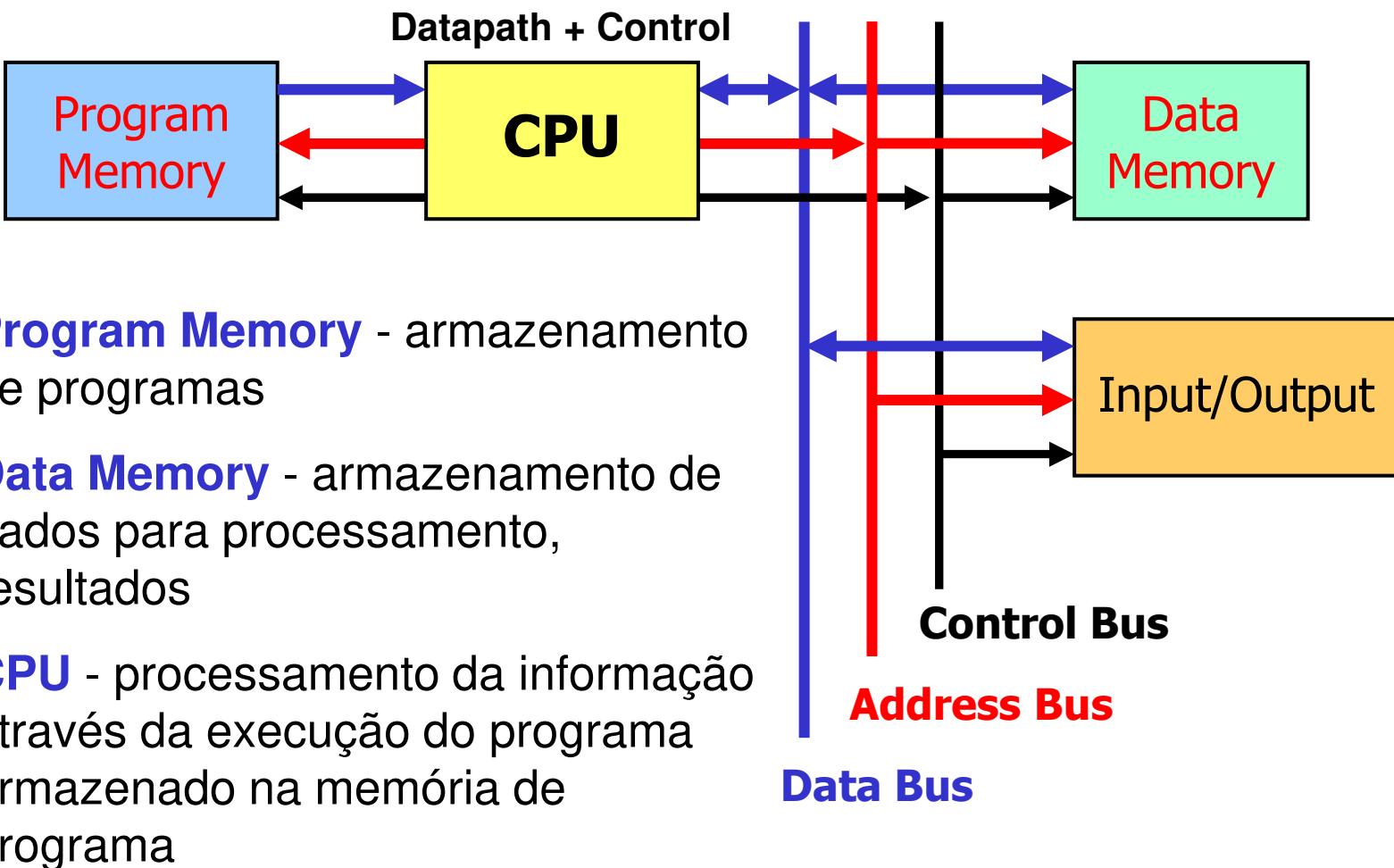
- **Memory** – armazenamento de: programas, dados para processamento, resultados
- **CPU** – processamento da informação através da execução do programa armazenado em memória
- **Input/Output** – comunicação com dispositivos periféricos



# Modelo de von Neumann

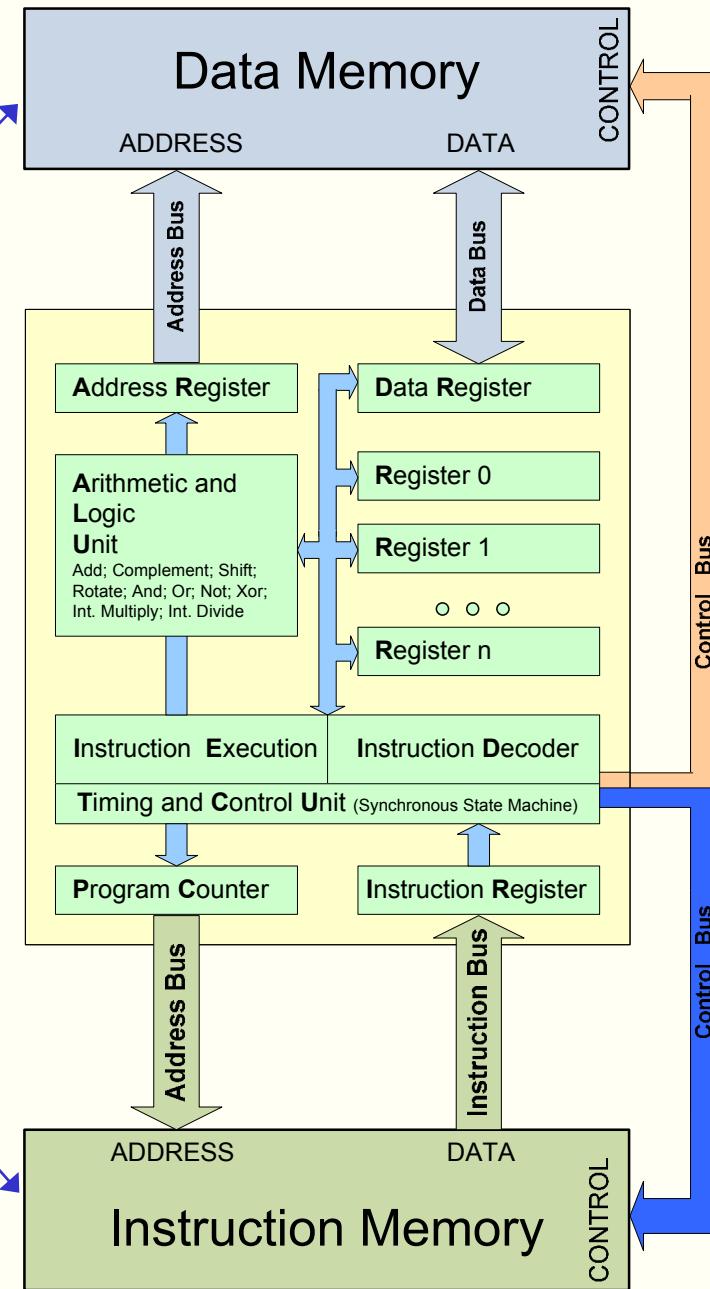


# Modelo de Harvard



# Modelo de Harvard

No Modelo de Harvard existem duas memórias independentes (uma para instruções e outra para dados) que podem ser acedidas simultaneamente pelo CPU



# von Neumann *versus* Harvard – resumo

- **Modelo de von Neumann**

- um único espaço de endereçamento para instruções e dados (i.e. uma única memória)
- acesso a instruções e dados é feito em ciclos de relógio distintos

- **Modelo de Harvard**

- dois espaços de endereçamento separados: um para dados e outro para instruções (i.e. duas memórias independentes)
- possibilidade de acesso, no mesmo ciclo de relógio, a dados e instruções (i.e. CPU pode fazer o *fetch* da instrução e ler os dados que a instrução vai manipular no mesmo ciclo de relógio)
- memórias de dados e instruções podem ter dimensões de palavra diferentes



# Implementação de um *Datapath*

- O CPU consiste, fundamentalmente, em duas secções:
  - **Secção de dados** - elementos operativos/funcionais para armazenamento, processamento e encaminhamento da informação:
    - Registos internos
    - Unidade Aritmética e Lógica (ALU)
    - Elementos de encaminhamento (multiplexers)
  - **Unidade de controlo**: responsável pela coordenação dos elementos da secção de dados, durante a execução de cada instrução



# Implementação de um *Datapath*

- As unidades funcionais que constituem o *datapath* são de dois tipos:
  - **Elementos combinatórios** (por exemplo a ALU)
  - **Elementos de estado**, isto é, que têm capacidade de armazenamento (por exemplo os registos internos \*)
- Um elemento de estado possui, pelo menos, duas entradas:
  - Uma para os **dados** a serem armazenados
  - Outra para o **relógio**, que determina o instante em que os dados são armazenados (**interface síncrona**)
- Um elemento de estado pode ser lido em qualquer momento
- A saída de um elemento de estado disponibiliza a informação armazenada na última transição ativa do relógio

(\*) Na abordagem que se faz a seguir considera-se a memória externa ao CPU como um elemento operativo integrante do *datapath* (elemento de estado)



# Implementação de um *Datapath*

- Para além do sinal de relógio, um elemento de estado pode ainda ter sinais de controlo adicionais:
  - **Um sinal de leitura (read)**, que permite (quando ativo) que a informação armazenada seja disponibilizada na saída (leitura assíncrona)
  - **Um sinal de escrita (write)**, que autoriza (quando ativo) a escrita de informação na próxima transição ativa do relógio (escrita síncrona)
- Se algum destes dois sinais não estiver explicitamente representado, isso significa que a operação respetiva é sempre realizada. No caso da operação de escrita ela é realizada uma vez por ciclo, e coincide com a transição ativa do sinal de relógio

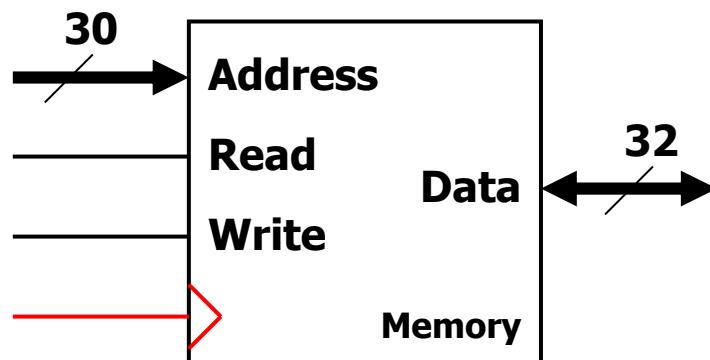
NOTA: Nos slides seguintes, havendo um sinal de relógio comum, e por uma questão de simplificação dos diagramas, o sinal de relógio pode não ser explicitamente representado



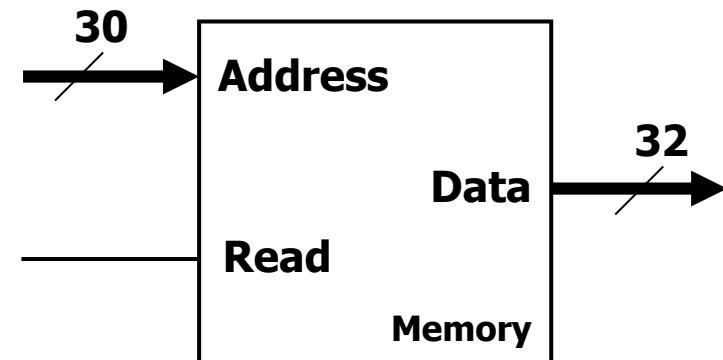
# Implementação de um *Datapath*

- Exemplos de representação gráfica de blocos funcionais correspondentes a elementos de estado

**Memória para escrita e leitura  
( $2^{30}$  words de 32 bits)**



**Memória apenas para leitura  
( $2^{30}$  words de 32 bits)**



O sinal “**Read**” pode não existir. Nesse caso a informação de saída estará sempre disponível e corresponderá ao conteúdo da posição de memória especificada na entrada “address”



# Implementação de um *Datapath*

- Nos próximos slides faz-se uma abordagem à implementação de um *datapath* capaz de interpretar e executar o seguinte subconjunto de instruções do MIPS:
  - As instruções aritméticas e lógicas (**add**, **addi**, **sub**, **and**, **or**, **slt**, **slti**)
  - Instruções de acesso à memória: load word (**lw**) e store word (**sw**)
  - As instruções de salto condicional (**beq**, **bne**) e salto incondicional (**j**)
- Como se verá, independentemente da quantidade e tipo de instruções suportadas por uma dada arquitetura, **uma parte importante do trabalho realizado pelo CPU e da infra-estrutura necessária para executar essas instruções é comum a praticamente todas elas**



# Implementação de um *Datapath*

- No caso particular do MIPS, para qualquer instrução que compõe o *set* de instruções, **as duas primeiras operações necessárias à sua realização são sempre as mesmas:**
  1. Usar o conteúdo do registo *Program Counter* (PC) para indicar o endereço da memória do qual vai ser lida a próxima instrução e efetuar essa leitura
  2. Ler dois registos internos, usando para isso os índices obtidos nos respetivos campos da instrução (rs e rt):
    - Nas instruções de transferência memória→registo (“lw”) e nas instruções que operam com constantes (immediatos) apenas o conteúdo de um registo é necessário (codificado no campo rs)
    - Em todas as outras é sempre necessário o conteúdo de dois registos (exceto na instrução “jump”)
- **Depois destas operações genéricas, realizam-se as ações específicas para completar a execução da instrução em causa**



# Implementação de um *Datapath*

- As ações específicas necessárias para executar as instruções de cada uma das três classes de instruções descritas anteriormente são, em grande parte, semelhantes, independentemente da instrução exata em causa
- Por exemplo, **todas as classes de instruções** (à exceção do salto incondicional) **utilizam a ALU depois da leitura dos registos**:
  - as instruções aritméticas e lógicas para a execução da instrução
  - as instruções de acesso à memória usam a ALU para calcular o endereço de memória
  - a instrução de *branch* para efetuar a subtração que permite determinar se os operandos são iguais ou diferentes
- A execução da instrução de salto incondicional ("j") resume-se à alteração incondicional do registo Program Counter (PC) – o novo valor é obtido a partir dos 26 LSB do código máquina da instrução e dos 4 bits mais significativos do valor atual do PC (ver aula 6)



# Implementação de um *Datapath*

- Depois de utilizar a ALU, as ações que completam as várias classes de instruções diferem:
  - as instruções **aritméticas e lógicas** armazenam o resultado à saída da ALU no registo destino especificado na instrução
  - a instrução **sw** acede à memória para escrita do valor do registo lido anteriormente (codificado no campo rt)
  - a instrução **lw** acede à memória para leitura; o valor lido da memória é, de seguida, escrito no registo destino especificado na instrução (codificado no campo rt)
  - a instrução de **branch** pode ter que alterar o conteúdo do registo Program Counter (i.e. o endereço onde se encontra a próxima instrução a ser executada) no caso de a condição testada ser verdadeira



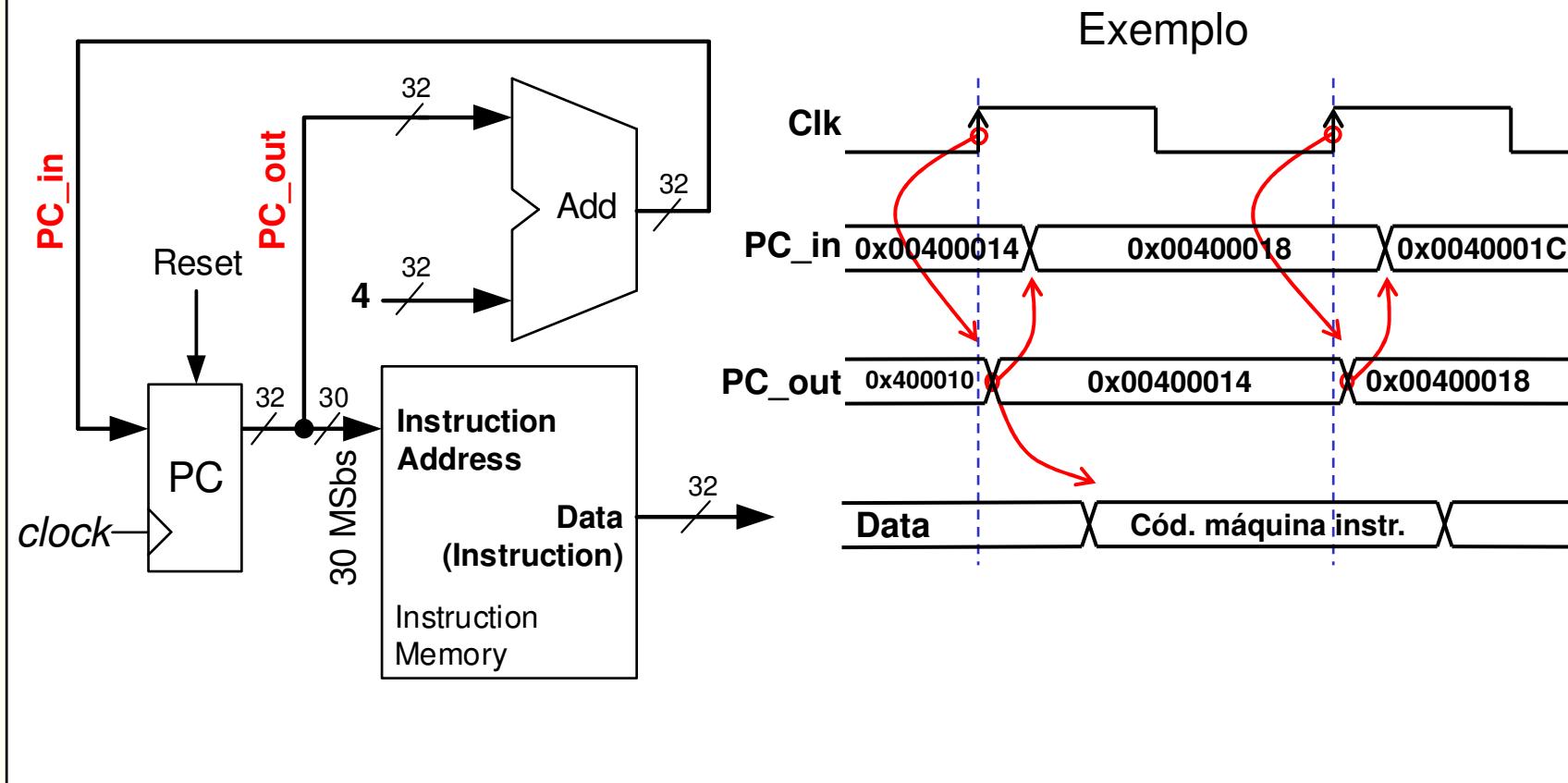
## Implementação de um *Datapath* – *Instruction Fetch*

- O processo de acesso à memória para leitura da próxima instrução é genericamente designado por ***Instruction Fetch***
- Por uma questão de simplificar a organização da informação, as instruções que compõem um programa são armazenadas sequencialmente na memória:
  - se a instrução  $n$  se encontra armazenada no endereço  $k$ , então a instrução  $n+1$  encontra-se armazenada no endereço  $k+x$ , em que  $x$  é a dimensão da instrução  $n$ , medida em bytes
  - no MIPS, a dimensão das instruções é fixa e igual a 4 bytes; o endereço  $k$  é sempre um **múltiplo de 4**
- **O processo de *Instruction Fetch* deverá, uma vez concluído, deixar o conteúdo do PC pronto para endereçar a próxima instrução**
  - No caso do MIPS, tal corresponde a adicionar a constante 4 ao valor atual do PC



## Implementação de um *Datapath – Instruction Fetch*

- A parte do *Datapath* necessária à execução de um *Instruction Fetch* toma, assim, a seguinte configuração



# Implementação de um *Datapath*

- Que outros elementos operativos básicos serão necessários para suportar a execução das várias classes de instruções que estamos a considerar?
  - Instruções aritméticas e lógicas
    - Tipo R: **add**, **sub**, **and**, **or**, **slt**
    - Tipo I: **addi**, **slti**
  - Instruções de leitura e escrita da memória (Tipo I: **lw**, **sw**)
  - Instruções de salto condicional (Tipo I: **beq**, **bne**)

Na análise que se segue, não se explicita a Unidade de Controlo. Esta unidade é responsável pela geração dos sinais de controlo que asseguram a coordenação dos elementos do *datapath* durante a execução de uma instrução



## Implementação de um *Datapath* – instruções tipo R

- Operações realizadas na execução de uma instrução do tipo R:
  - **Instruction Fetch** (leitura da instrução, cálculo de PC+4)
  - **Leitura dos registos** operando (registos especificados nos campos “rs” e “rt” da instrução)
  - **Realização da operação** na ALU (especificada no campo “funct”)
  - **Escrita do resultado** no registo destino (especificado no campo “rd”)

Exemplo: **add \$2, \$3, \$4**

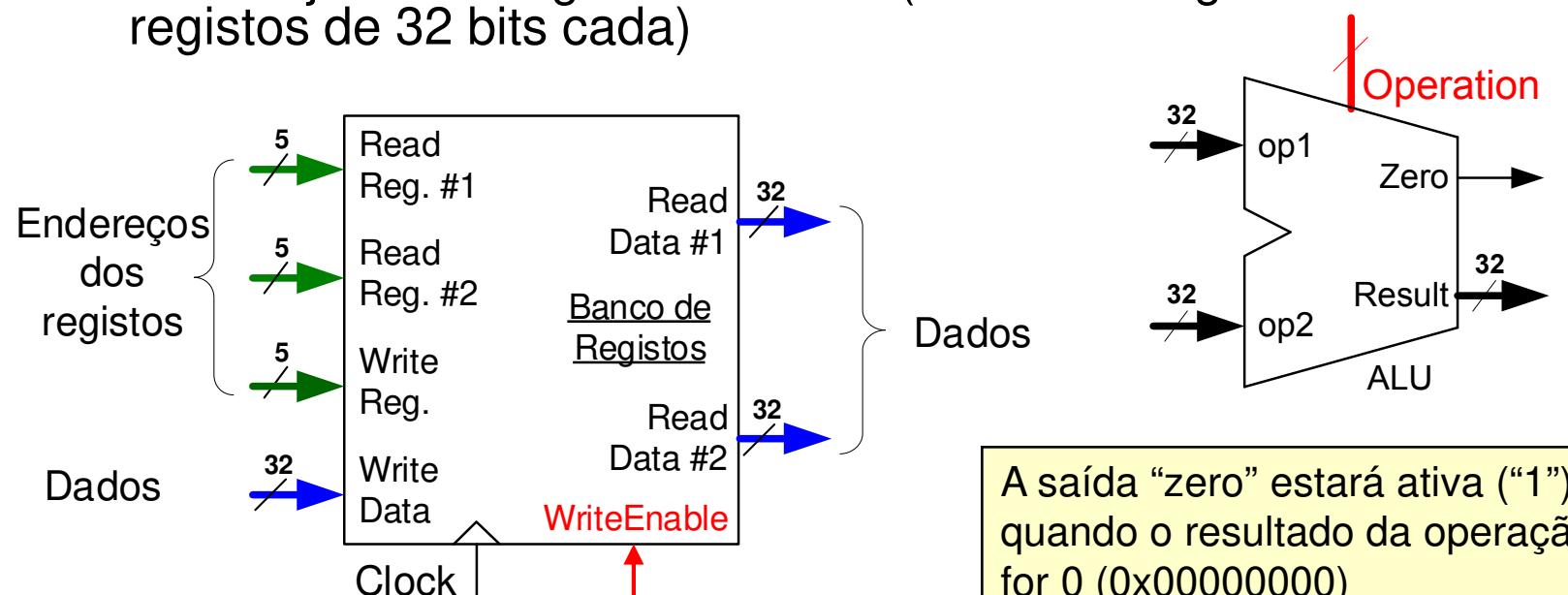
| 31 | opcode<br>( 0 ) | rs<br>( 3 ) | rt<br>( 4 ) | rd<br>( 2 ) | shamt<br>( 0 ) | 0      | funct<br>( 32 ) |
|----|-----------------|-------------|-------------|-------------|----------------|--------|-----------------|
|    | 6 bits          | 5 bits      | 5 bits      | 5 bits      | 5 bits         | 6 bits |                 |

Código máquina: 0x00641020



## Implementação de um *Datapath* – instruções tipo R

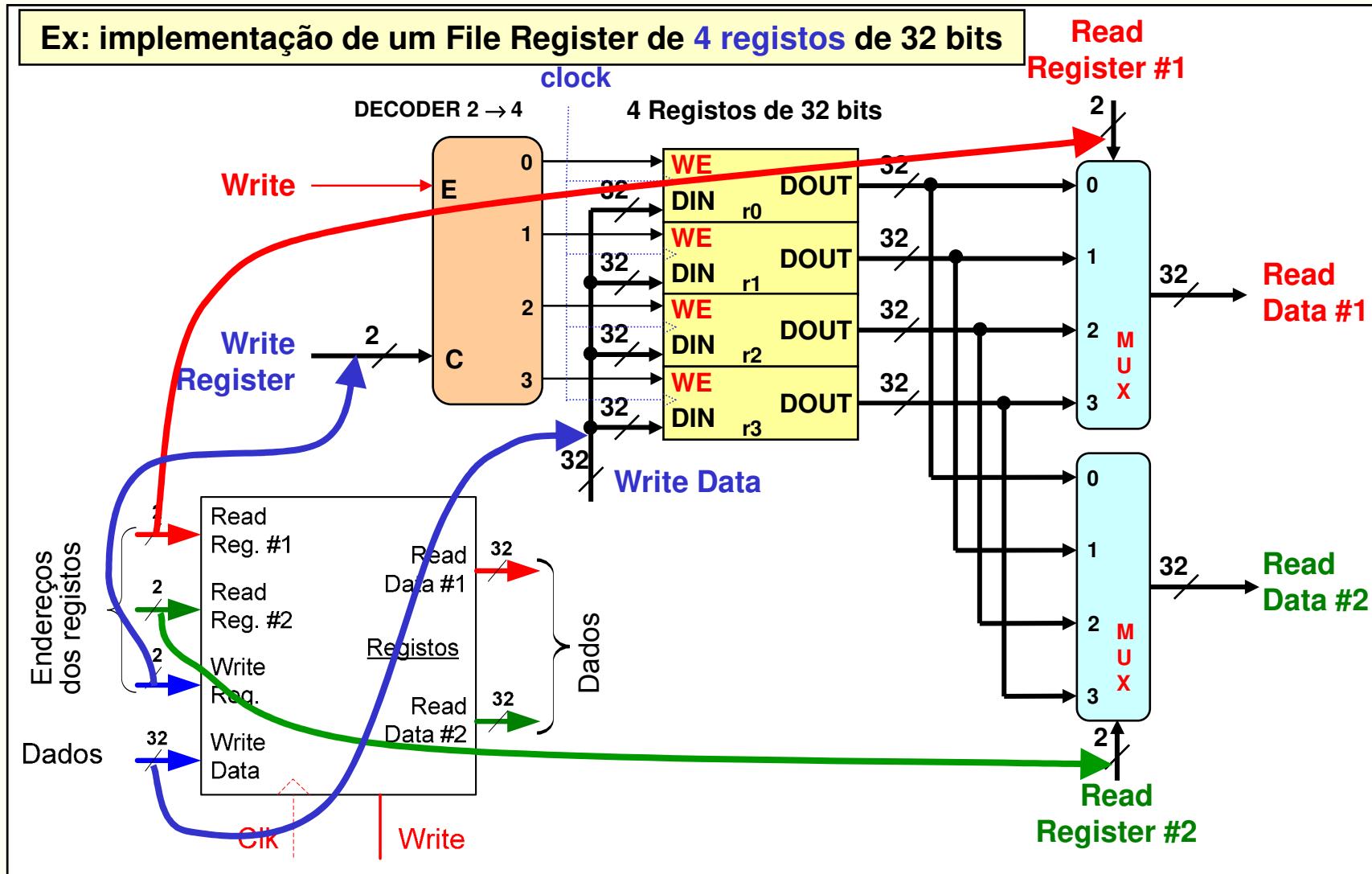
- Os elementos necessários à execução das instruções aritméticas e lógicas (tipo R) são:
  - Uma ALU de 32 bits
  - Um conjunto de registos internos (Banco de registos com 32 registos de 32 bits cada)



- 1 porto de escrita síncrona
- 2 portos de leitura assíncrona



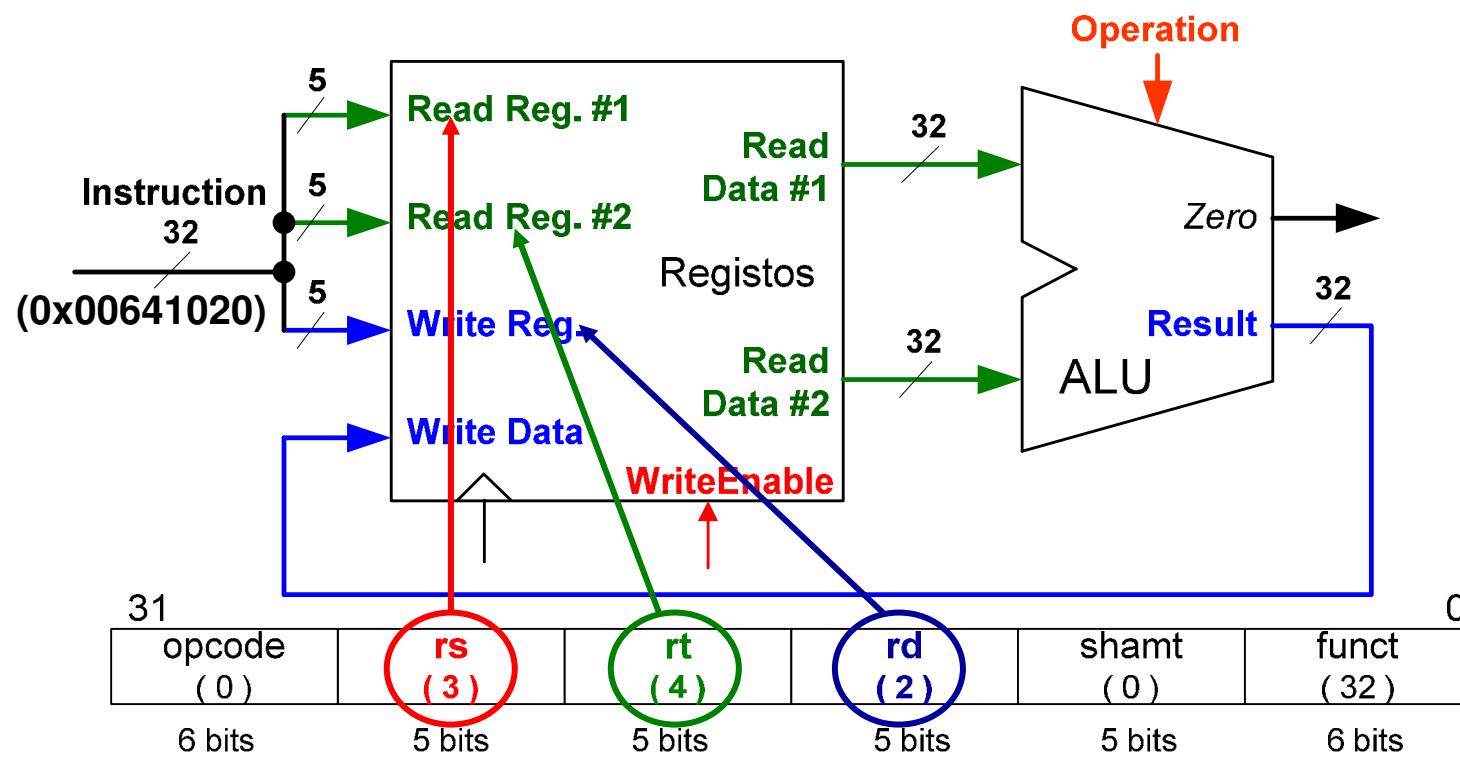
# Implementação de um Datapath – instruções tipo R



# Implementação de um *Datapath* – instruções tipo R

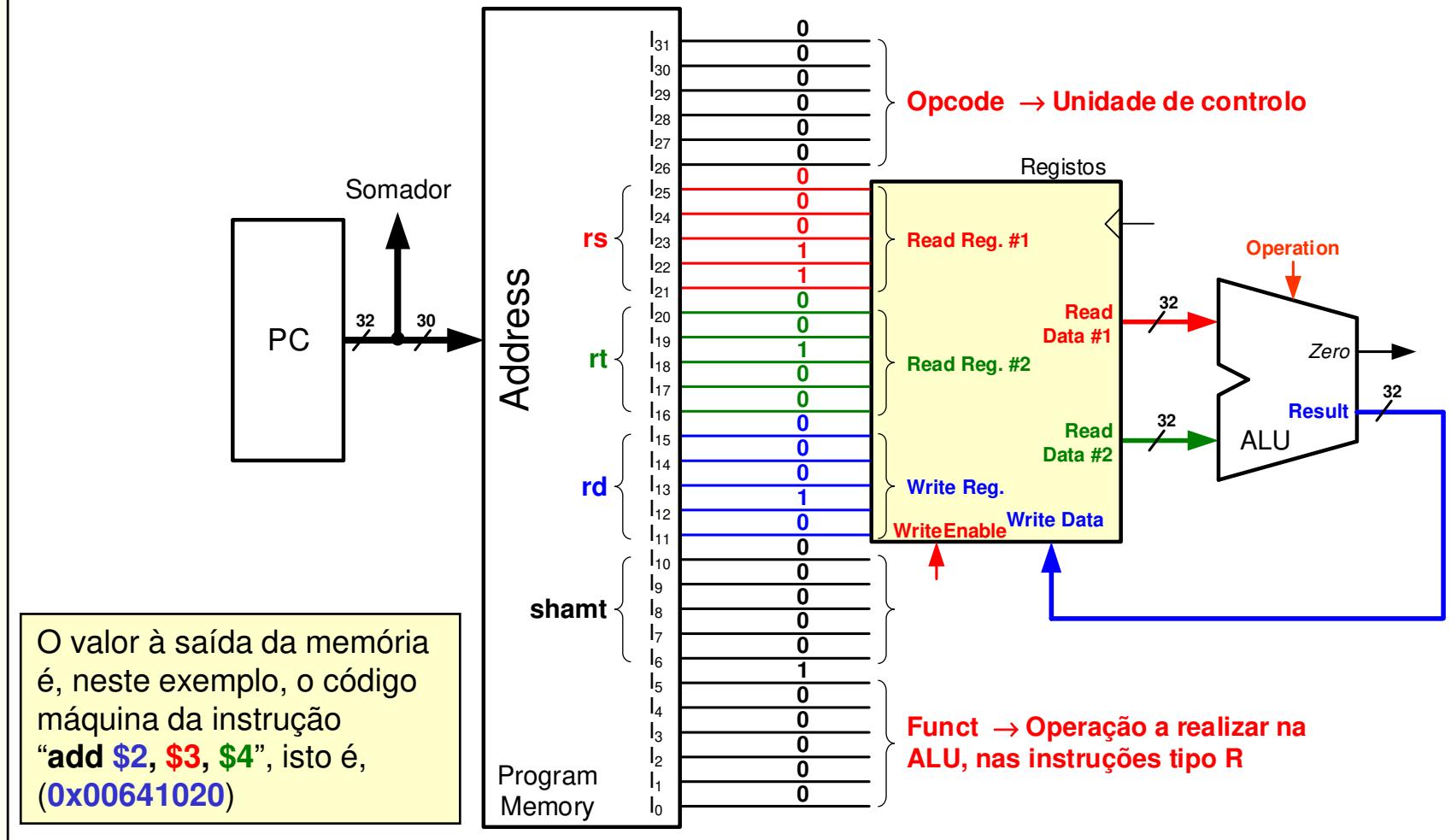
- A interligação dos elementos operativos será:

Exemplo: **add \$2, \$3, \$4**



# Implementação de um Datapath – instruções tipo R

- Ligaçāo entre a memória de código e o Banco de Registos (Instruções tipo R)



# Implementação de um *Datapath* (Instrução SW)

- Operações realizadas na execução de uma instrução “sw”:
  - *Instruction Fetch* (leitura da instrução, cálculo de PC+4)
  - Leitura dos registos que contêm o **endereço-base** e o **valor a transferir** (registos especificados nos campos “rs” e “rt”da instrução, respetivamente)
  - Cálculo, na ALU, do endereço de acesso (soma algébrica entre o conteúdo do registo “rs” e o **offset** especificado na instrução)
  - Escrita na memória

Exemplo: sw \$2, 0x24( \$4 )

Endereço inicial da memória onde vai ser escrita a word de 32 bits armazenada no registo \$2

|                  |             |             |                    |
|------------------|-------------|-------------|--------------------|
| opcode<br>( 43 ) | rs<br>( 4 ) | rt<br>( 2 ) | offset<br>( 0x24 ) |
|------------------|-------------|-------------|--------------------|



# Implementação de um *Datapath* (Instrução LW)

- Operações realizadas na execução de uma instrução “**Iw**”
  - *Instruction Fetch* (leitura da instrução, cálculo de PC+4)
  - Leitura do registo que contém o endereço base (registo especificado no campo “**rs**” da instrução)
  - Cálculo, na ALU, do endereço de acesso (soma algébrica entre o conteúdo do registo “**rs**” e o **offset** especificado na instrução)
  - Leitura da memória
  - Escrita do valor lido da memória no registo destino (especificado no campo “**rt**” da instrução)

Exemplo: **Iw \$4, 0x2F( \$15 )**

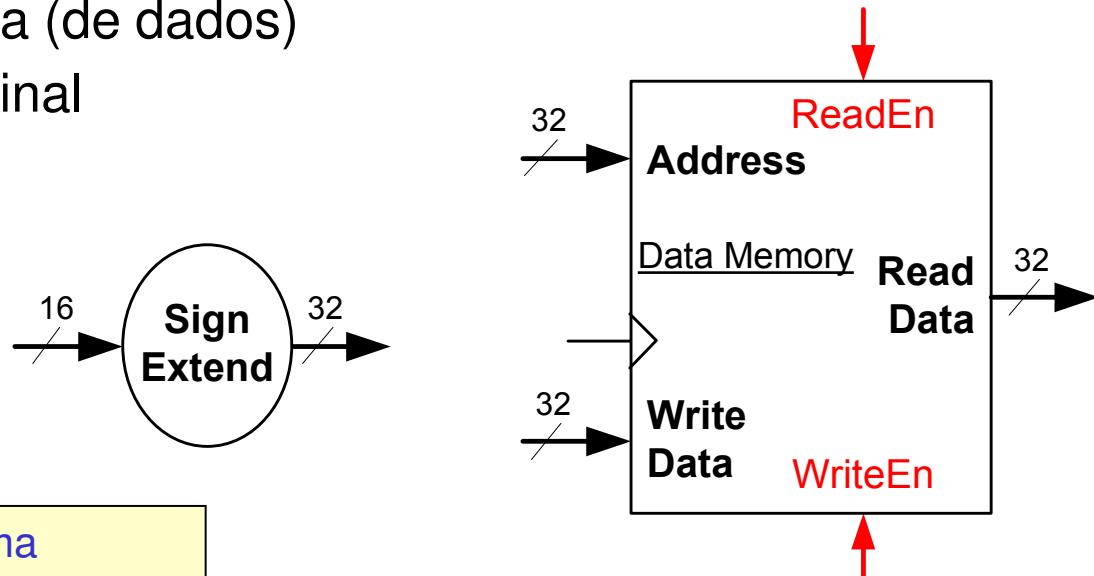
Endereço inicial da memória para leitura de uma word de 32 bits (vai ser escrita no registo \$4)

|                  |                     |                    |                           |
|------------------|---------------------|--------------------|---------------------------|
| opcode<br>( 35 ) | <b>rs</b><br>( 15 ) | <b>rt</b><br>( 4 ) | <b>offset</b><br>( 0x2F ) |
|------------------|---------------------|--------------------|---------------------------|



## Implementação de um *Datapath* (Instruções *lw* e *sw*)

- Os elementos necessários à execução das instruções de transferência de informação entre registos e memória (*load* e *store*) são, para além da ALU e do Banco de Registos:
  - A memória externa (de dados)
  - Um extensor de sinal

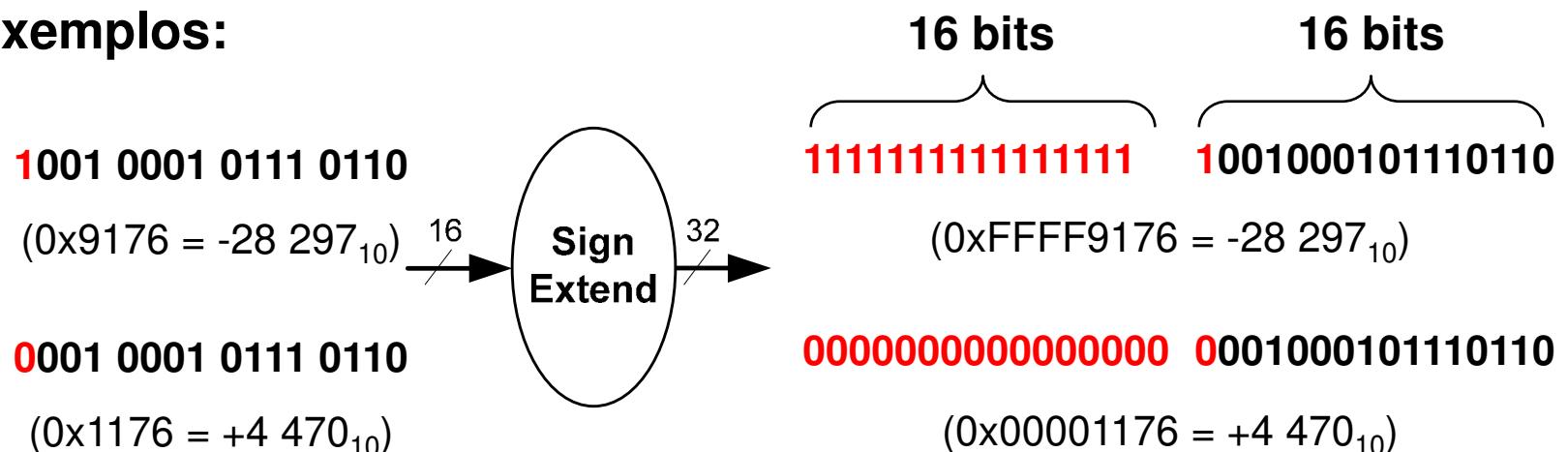


# Implementação de um *Datapath* (Instruções *lw* e *sw*)

## Módulo de extensão de sinal

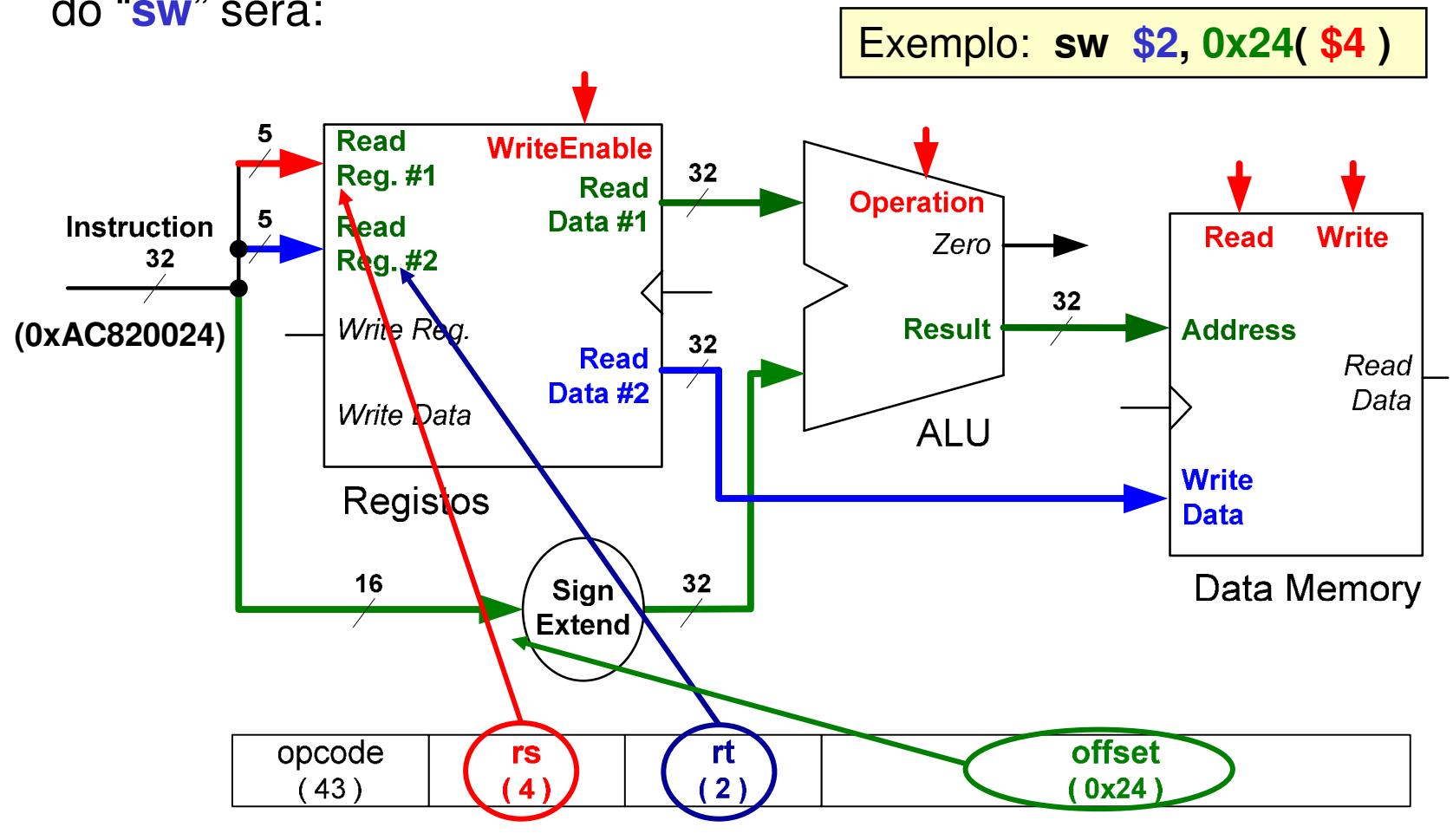
- 16 bits são apresentados na entrada
- 32 bits são disponibilizados na saída

### Exemplos:



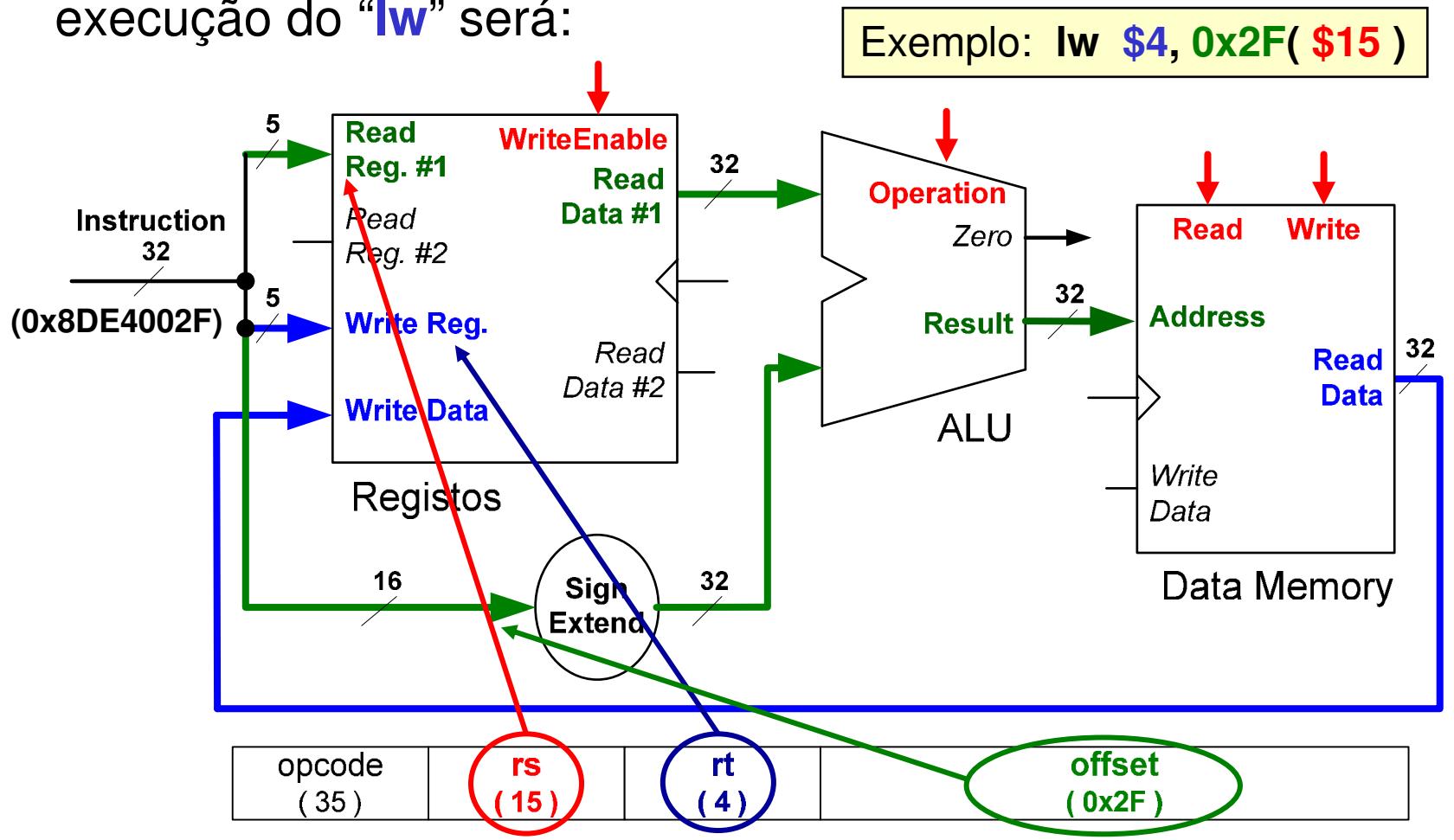
# Implementação de um Datapath (Instruções lw e sw)

- A interligação dos elementos operativos necessários à execução do “sw” será:



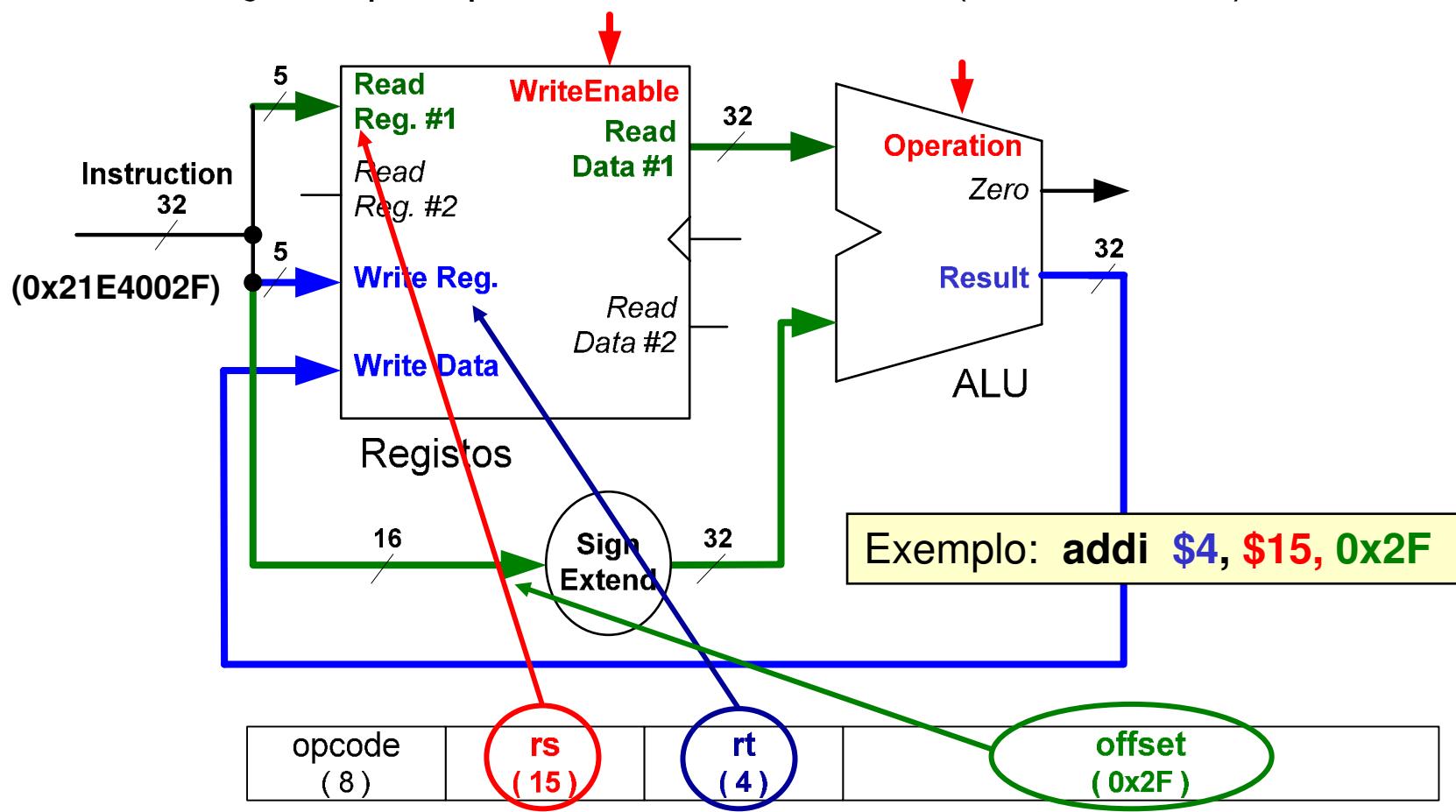
# Implementação de um Datapath (Instruções lw e sw)

- A interligação dos elementos operativos necessários à execução do “lw” será:



# Implementação de um Datapath (Instruções “imediatas”)

- A interligação dos elementos operativos necessários à execução de instruções que operam com constantes (“**addi**”, “**slti**”) será:



# Implementação de um *Datapath* (Instruções de *branch*)

- Operações realizadas na execução de uma instrução de *branch*:
  - *Instruction Fetch* (leitura da instrução, cálculo de PC+4)
  - Leitura dos registos a comparar
  - Comparação dos valores dos registos (realização de uma operação de subtração na ALU)
  - Cálculo do endereço-alvo da instrução de *branch* (*Branch Target Address* - BTA) - ver aula 6
$$\text{BTA} = (\text{PC} + 4) + (\text{instruction\_offset} * 4)$$
  - Alteração do valor do registo PC:
    - se a condição testada pelo *branch* for verdadeira PC = BTA
    - se a condição testada pelo *branch* for falsa PC = PC + 4

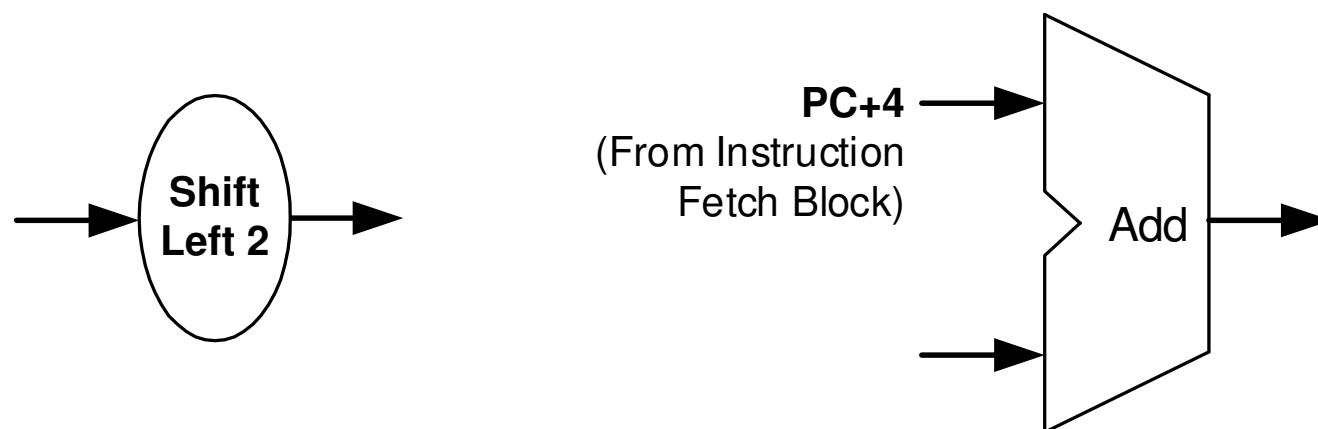
Exemplo: **beq \$2, \$3, 0x20**

|                 |             |             |                                |
|-----------------|-------------|-------------|--------------------------------|
| opcode<br>( 4 ) | rs<br>( 2 ) | rt<br>( 3 ) | instruction_offset<br>( 0x20 ) |
|-----------------|-------------|-------------|--------------------------------|



## Implementação de um *Datapath* (Instruções de branch)

- Finalmente, os elementos necessários à execução das instruções de salto condicional implicam a inclusão dos seguintes elementos:
  - left shifter* (2 bits)
  - um somador



O *left shifter* recupera os 2 bits menos significativos do *instruction offset* que são desprezados no momento da codificação da instrução (ver aula 6)

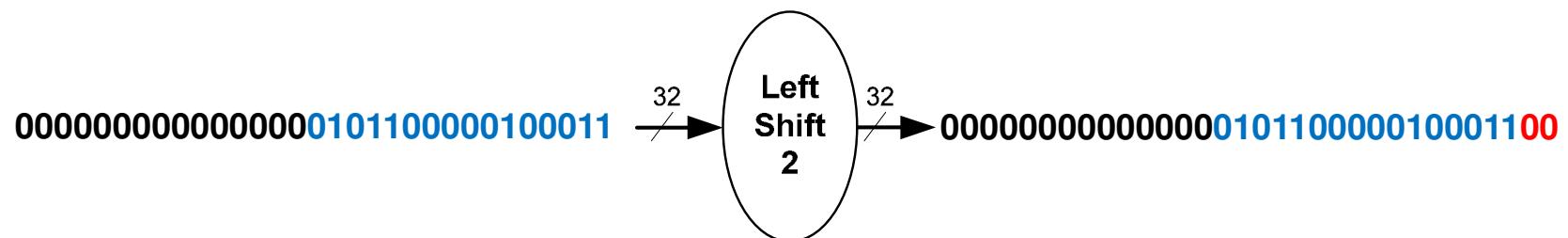


# Implementação de um *Datapath* (Instruções lw e sw)

## Módulo de left shift

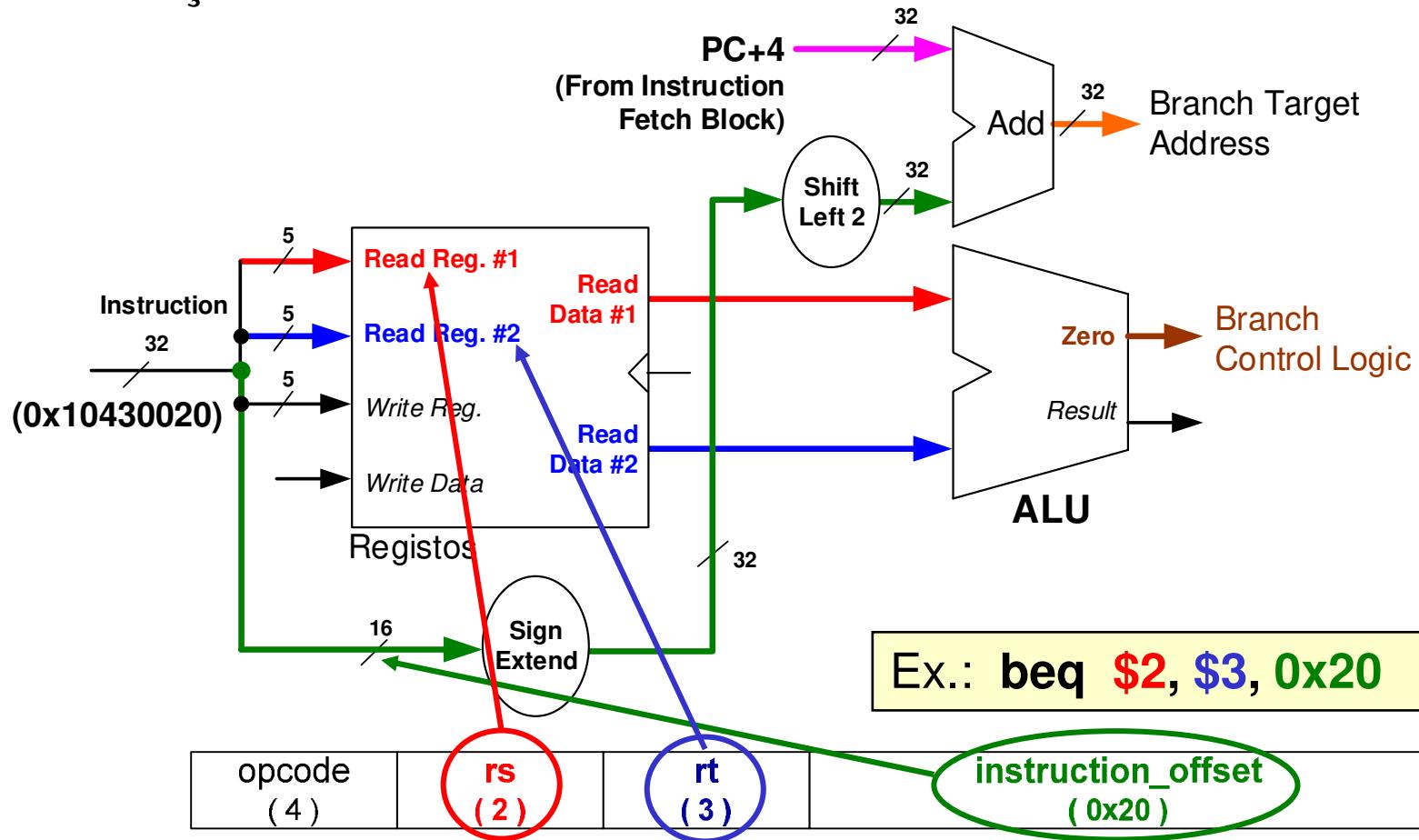
- 32 bits são apresentados na entrada
- Saída = 32 bits à entrada \* 4

### Exemplo:



# Implementação de um Datapath (Instruções de *branch*)

- Interligação dos elementos operativos necessários à execução de uma instrução de **branch**:



# Implementação de um *Datapath* – juntando tudo

- Nos slides anteriores identificaram-se, separadamente, os blocos básicos constituintes do *Datapath* necessários à execução dos vários tipos de instruções
- **Como juntar e interligar os diversos blocos, por forma a servir todas as instruções?**
  - Identificação dos blocos que podem ser partilhados pelos vários tipos de instruções
  - Desenvolvimento de uma estratégia que permita que os mesmos possam ser “configurados” para cada caso
- (o suporte para a instrução J (jump) será introduzido mais tarde)



# Implementação de um *Datapath* – juntando tudo

- Relembremos o formato de codificação dos três tipos de instruções:

| Aritméticas e lógicas – Tipo R |               |        |        |        |        |
|--------------------------------|---------------|--------|--------|--------|--------|
| 31                             | opcode<br>(0) | rs     | rt     | rd     | shamt  |
|                                | 6 bits        | 5 bits | 5 bits | 5 bits | 5 bits |

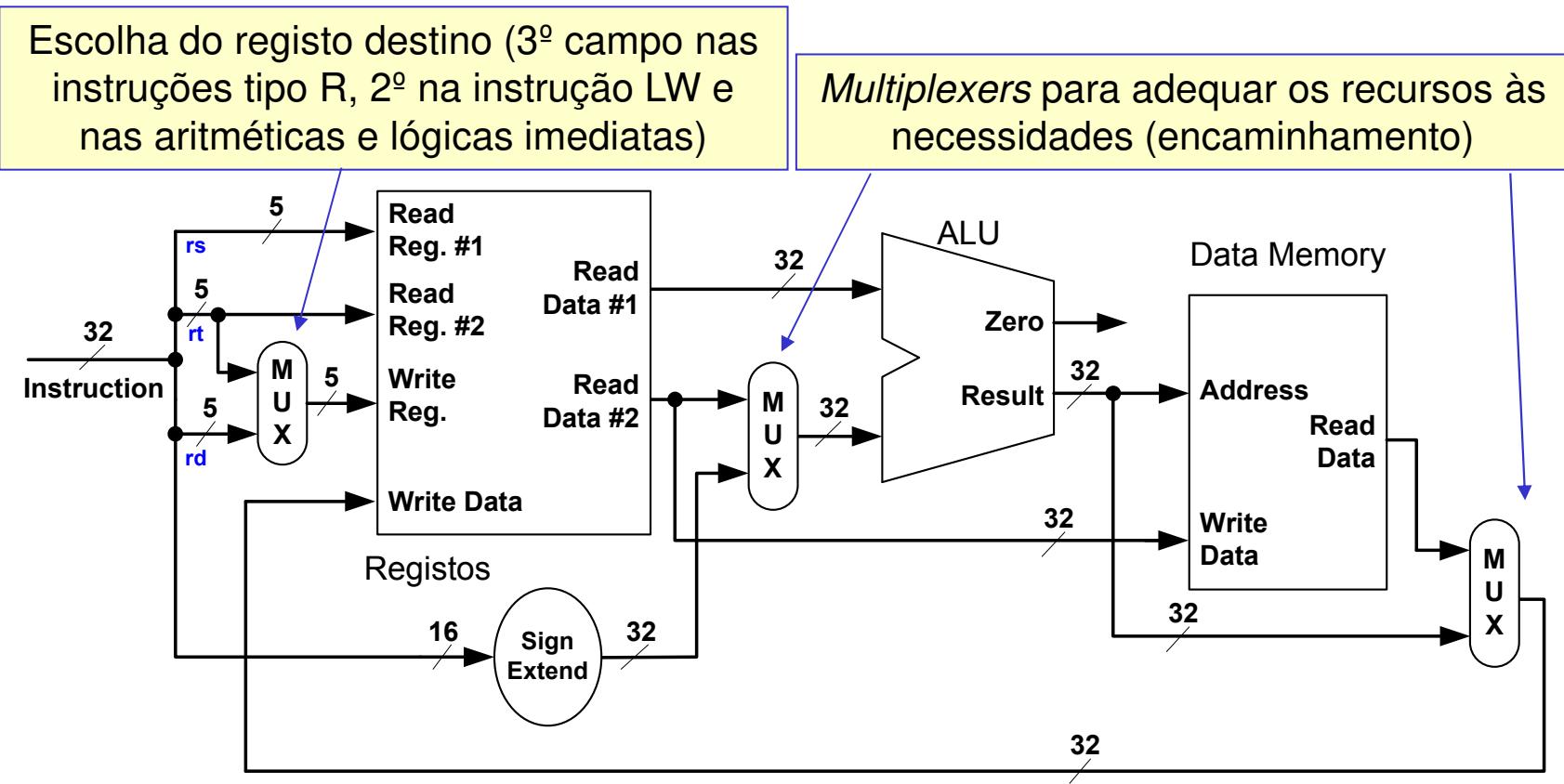
| LW, SW, aritméticas imediatas – Tipo I |        |        |        |
|--|--------|--------|--------|
| 31                                     | opcode | rs     | rt     |
|  | 6 bits | 5 bits | 5 bits |

| Branches – Tipo I |        |        |        |
|-------------------|--------|--------|--------|
| 31                | opcode | rs     | rt     |
|                   | 6 bits | 5 bits | 5 bits |



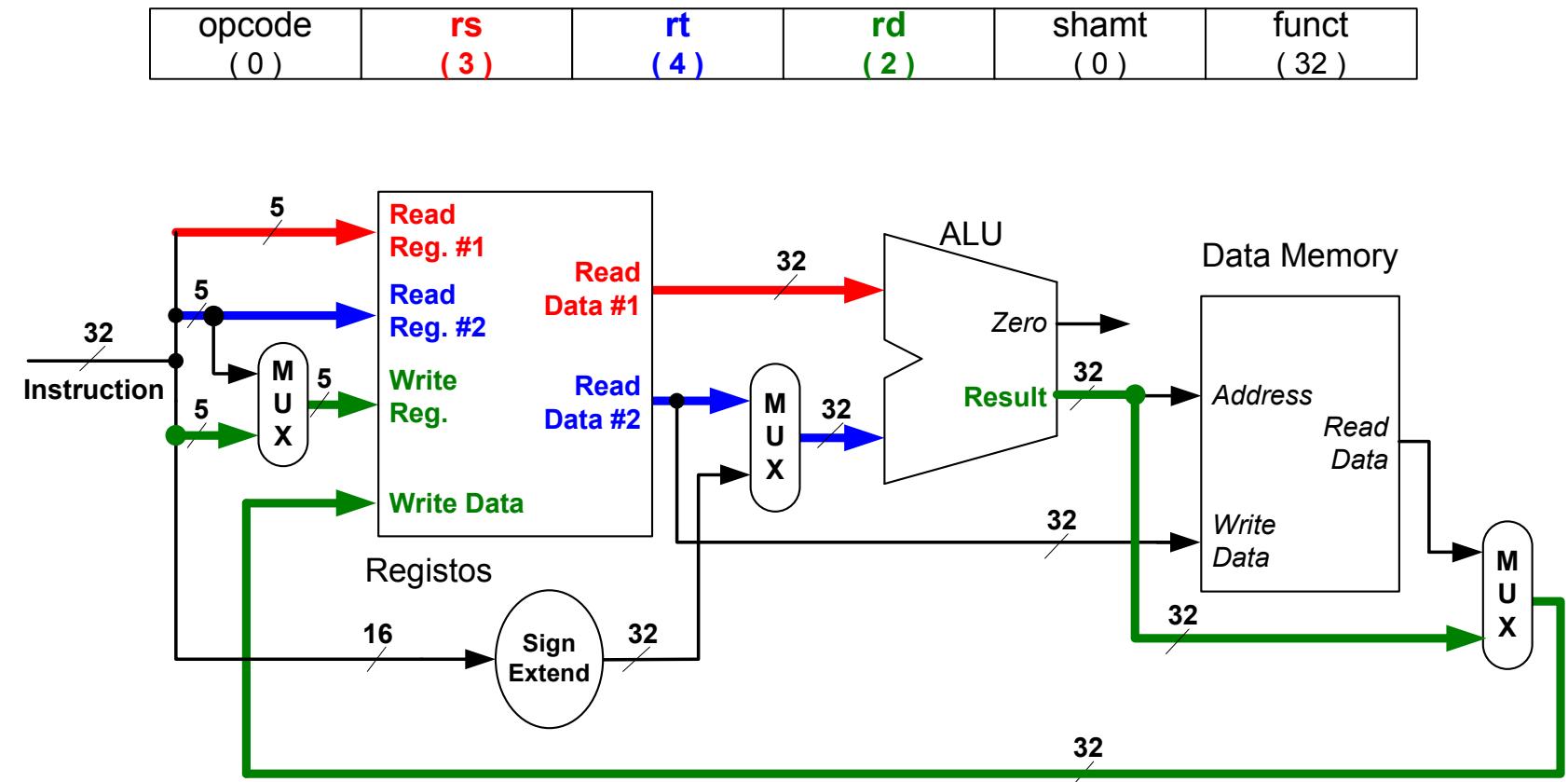
# Implementação de um *Datapath* – juntando tudo

- **1º passo:** combinação das instruções de acesso à memória com as instruções aritméticas e lógicas do tipo R e do tipo I:



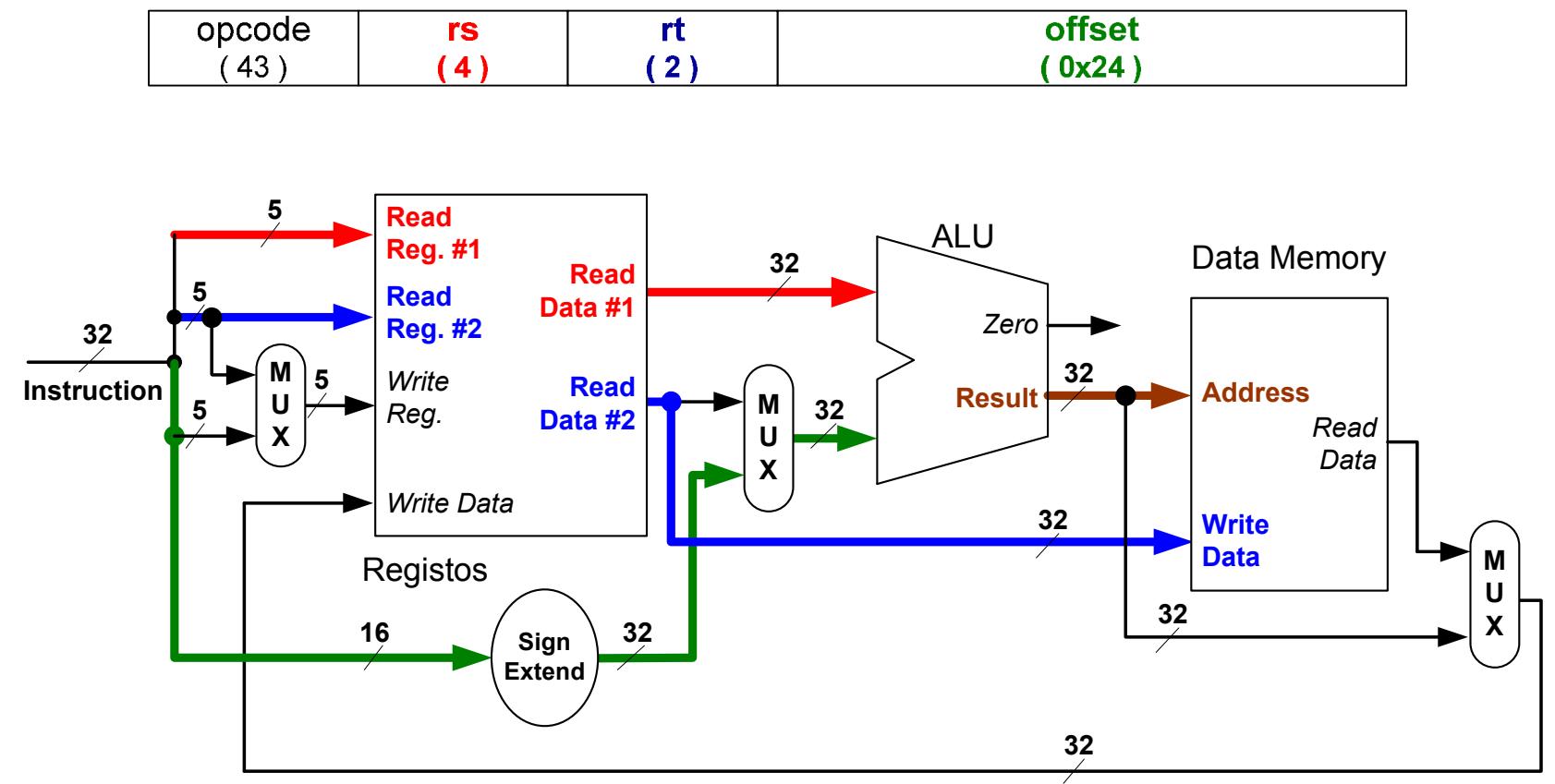
# Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução do tipo R. Exemplo: **add \$2, \$3, \$4**



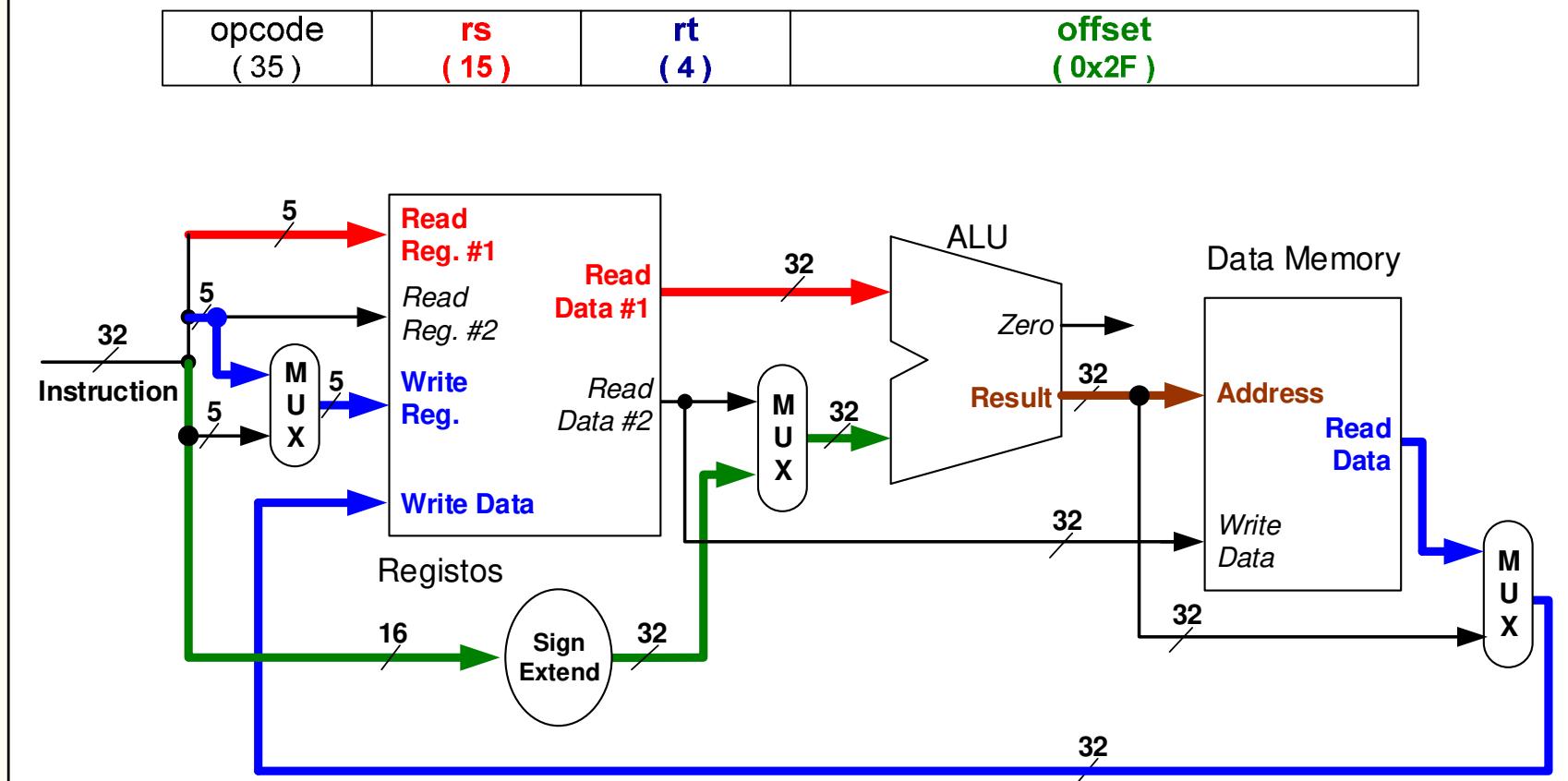
# Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução SW (*store word*). Exemplo: **sw \$2, 0x24 (\$4)**



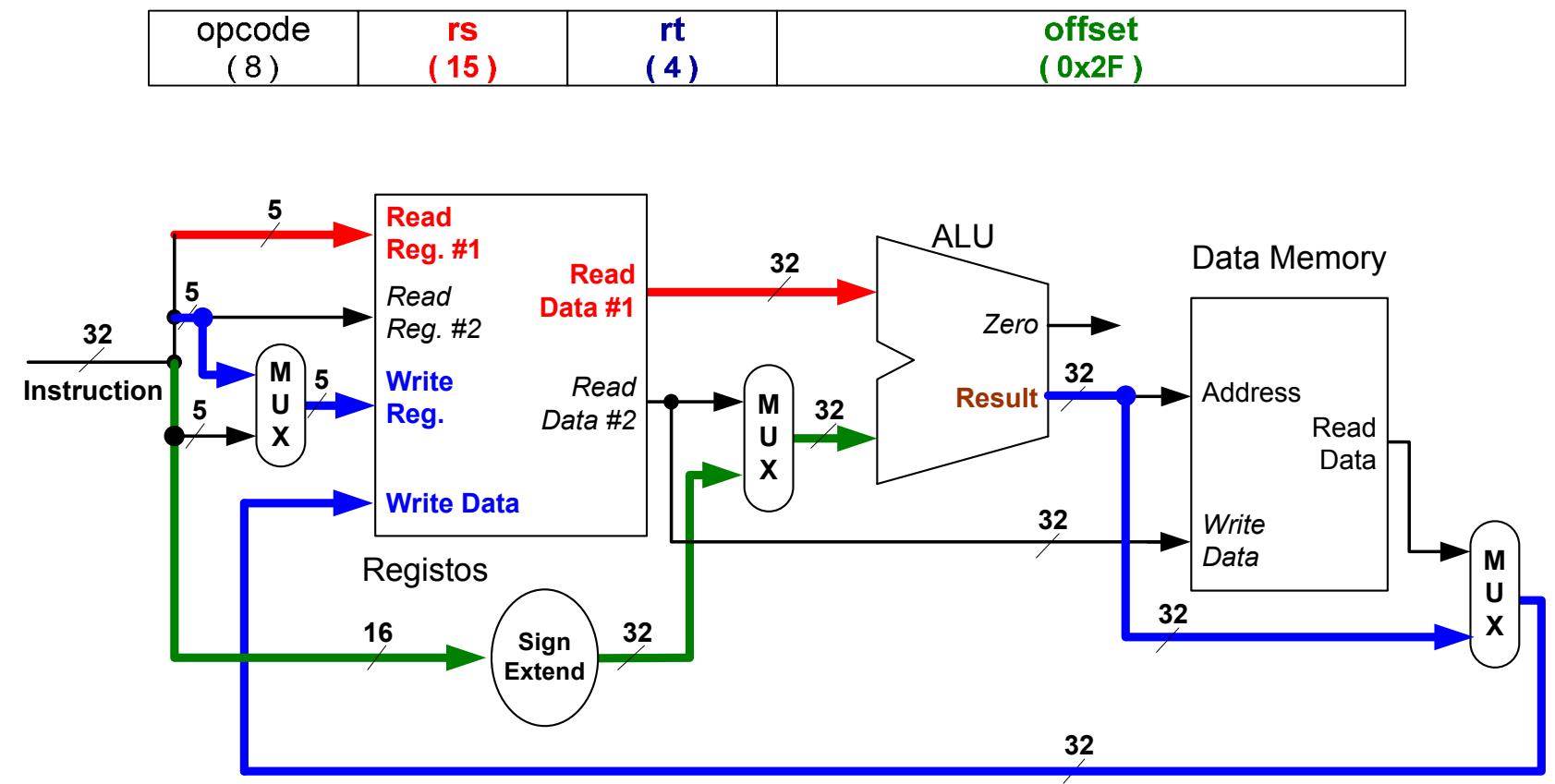
# Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução LW (*load word*). Exemplo: `lw $4, 0x2F($15)`



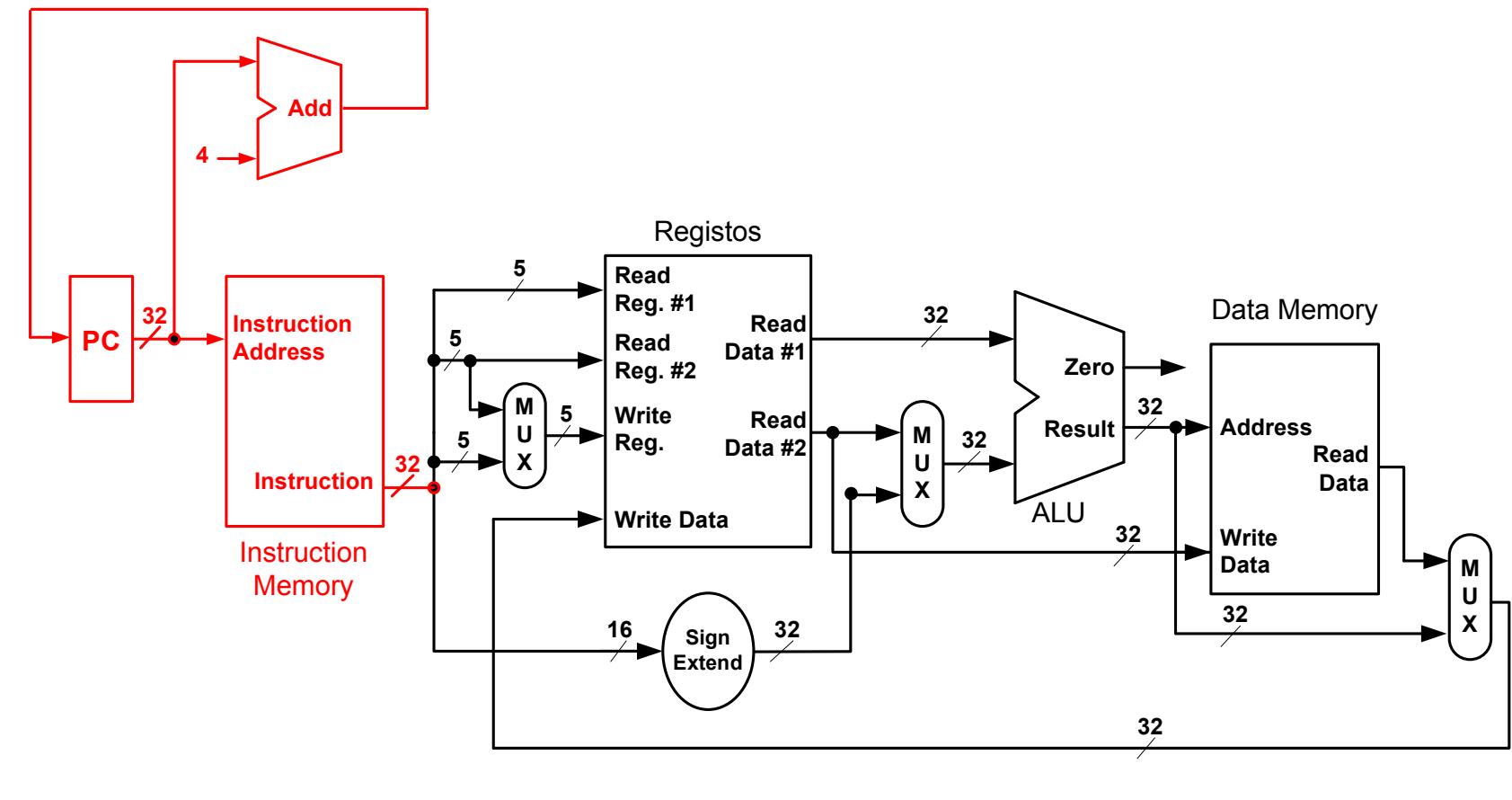
# Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução aritmética imediata. Exemplo: **addi \$4, \$15, 0x2F**



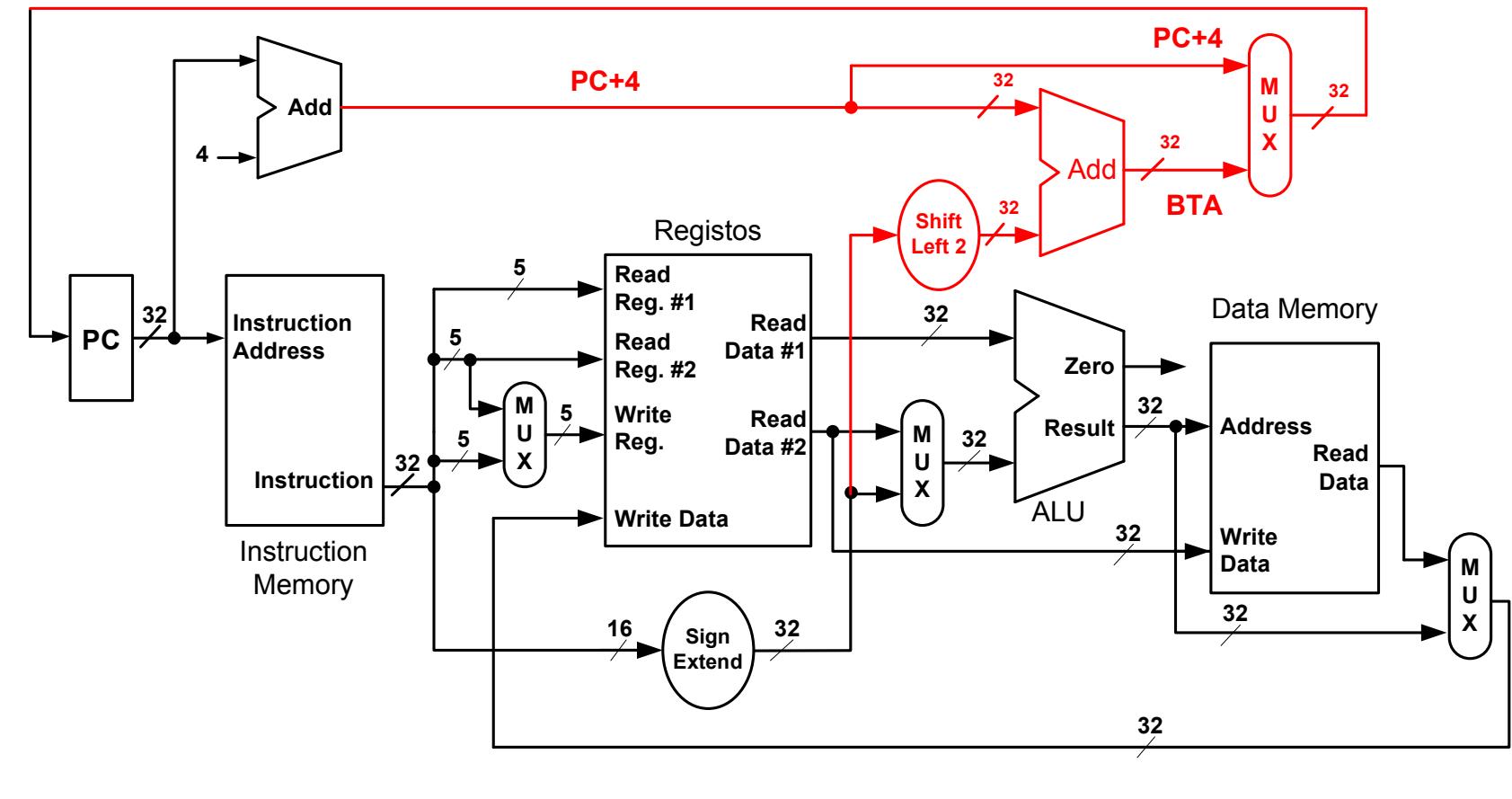
# Implementação de um *Datapath* – juntando tudo

- **2º passo:** inclusão do bloco *Instruction Fetch*



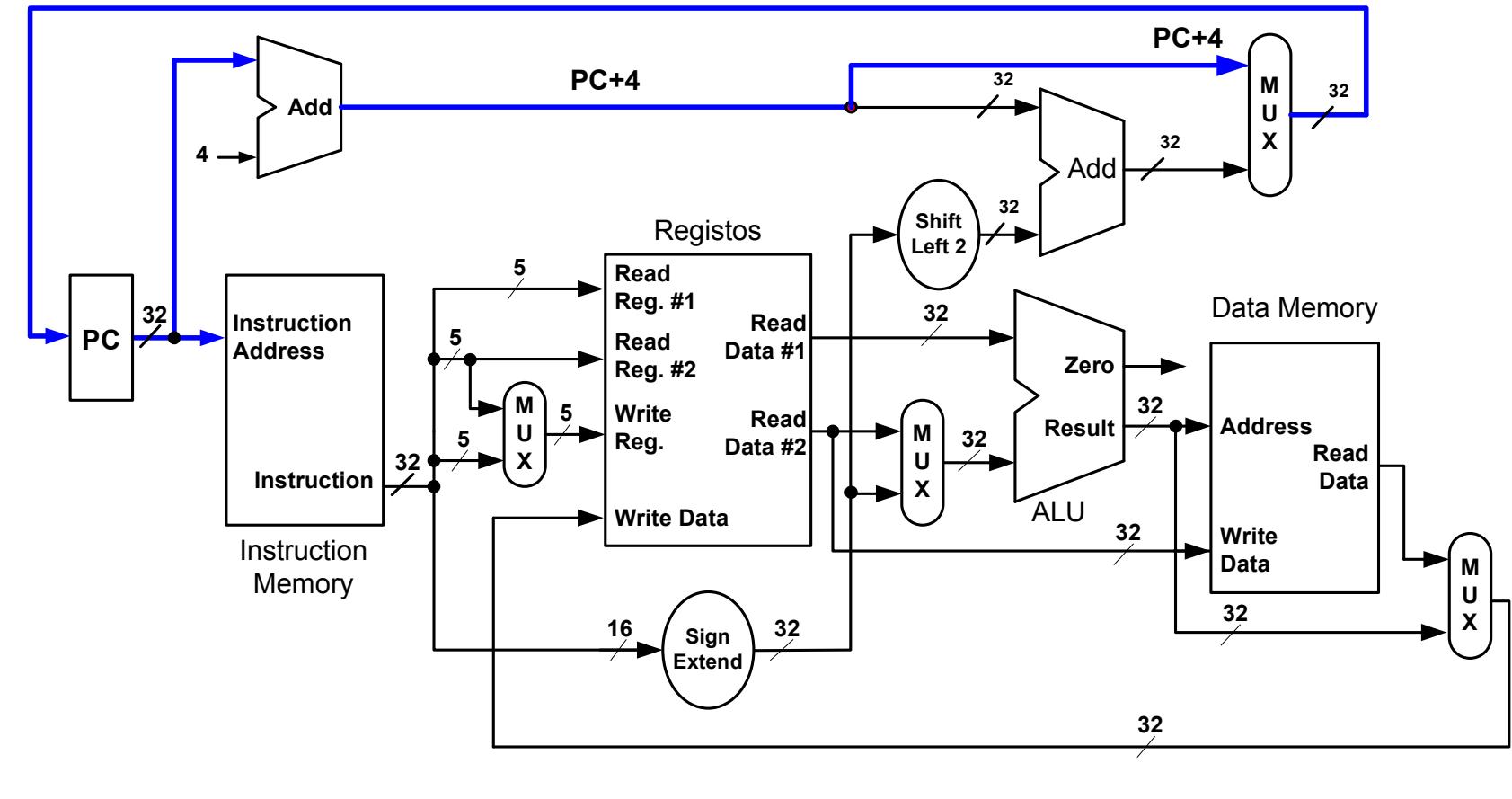
# Implementação de um *Datapath* – juntando tudo

- **3º passo:** adição das instruções de salto condicional (*branches*)



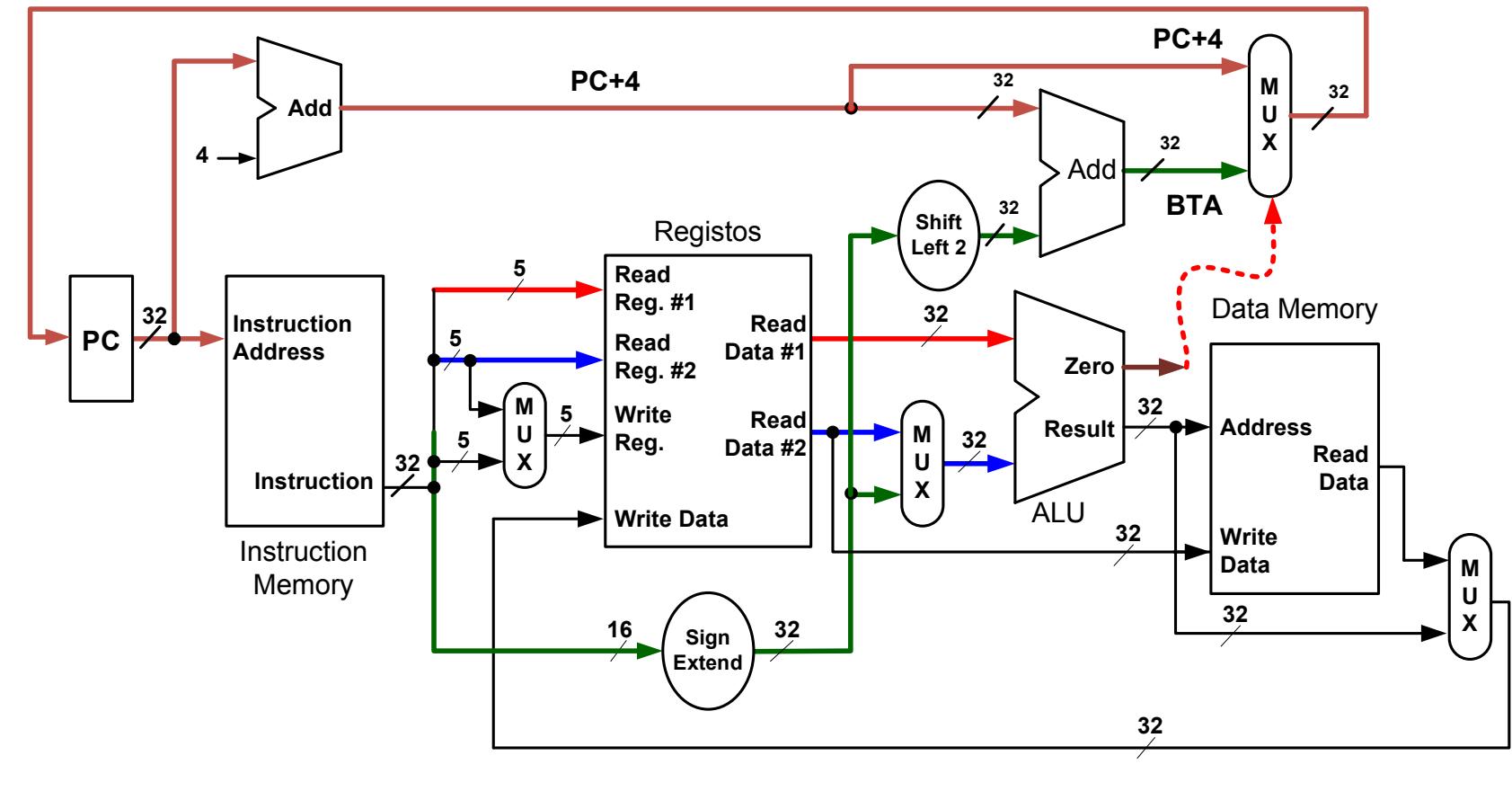
# Implementação de um *Datapath* – juntando tudo

- Fluxo da informação durante o *instruction fetch*



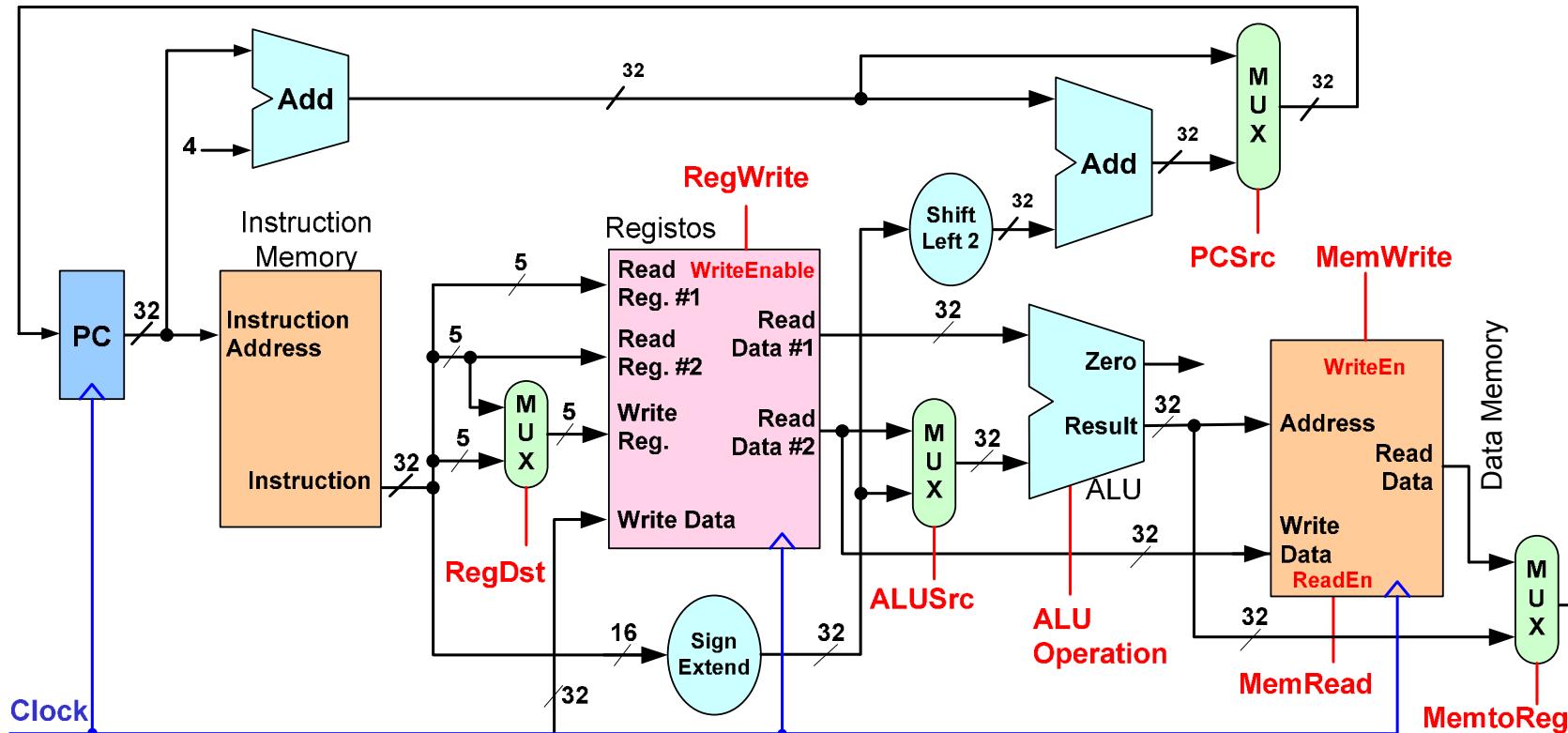
# Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução de *branch*



# Datapath single-cycle completo

- Datapath completo com identificação dos sinais de controlo



A utilização de memórias distintas para dados e código, permite a execução de cada instrução num único ciclo de relógio

Que alterações é necessário fazer para incluir a execução do “bne” ?



## Aulas 13, 14 e 15

- Instanciação dos vários módulos do datapath recorrendo à linguagem de programa hardware VHDL

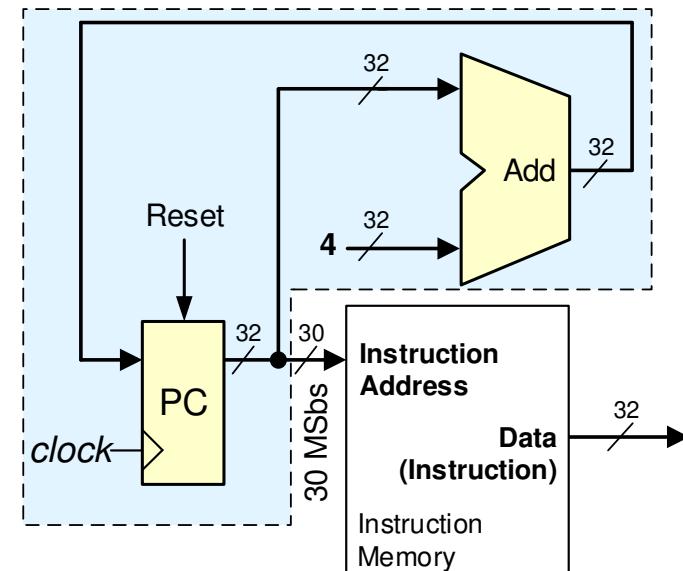
José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Implementação de um *Datapath* – Atualização do PC

```
entity PCupdate is
    port( clk      : in std_logic;
          reset   : in std_logic;
          pc      : out std_logic_vector(31 downto 0));
end PCupdate;

architecture Behavioral of PCupdate is
    signal s_pc : unsigned(31 downto 0);
begin
    process(clk)
    begin
        if(rising_edge(clk)) then
            if(reset = '1') then
                s_pc <= (others => '0');
            else
                s_pc <= s_pc + 4;
            end if;
        end if;
    end process;
    pc <= std_logic_vector(s_pc);
end Behavioral;
```



# Implementação de um *Datapath – Instruction Memory*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

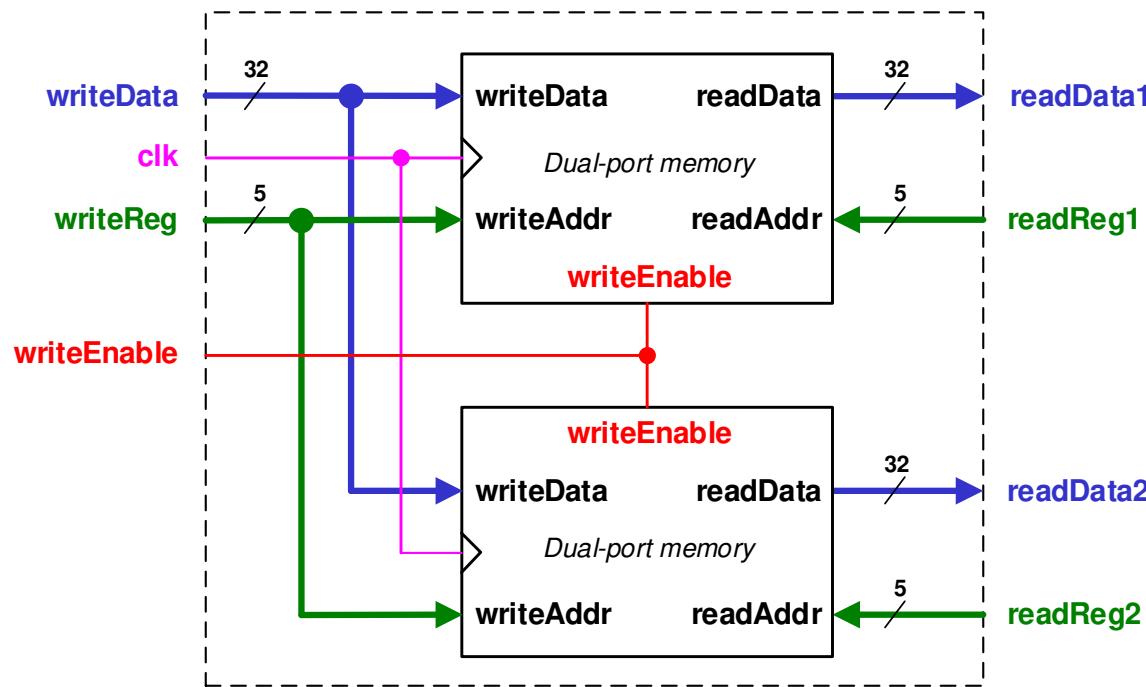
entity instructionMemory is
    generic(ADDR_BUS_SIZE : positive := 6);
    port( address : in std_logic_vector(ADDR_BUS_SIZE-1 downto 0);
          readData : out std_logic_vector(31 downto 0));
end instructionMemory;

architecture Behavioral of instructionMemory is
    constant NUM_WORDS : positive := (2 ** ADDR_BUS_SIZE );
    subtype TData is std_logic_vector(31 downto 0);
    type TMemory is array(0 to NUM_WORDS - 1) of TData;
    constant s_memory : TMemory := (X"8C010000", -- lw $1,0($0)
                                    X"20210004", -- addi $1,$1,4
                                    X"AC010004", -- sw $1,4($0)
                                    others => X"00000000");
begin
    readData <= s_memory(to_integer(unsigned(address)));
end Behavioral;
```



# Banco de Registros

- O banco de registos pode ser implementado com duas memórias de duplo porto (um porto de escrita e um porto de leitura):



- o porto de escrita do banco de registos é comum às duas memórias (i.e. a escrita é feita simultaneamente nas duas memórias)
- cada memória fornece um porto de leitura independente



# Banco de registros (dual-port memory) – VHDL

```
entity DP_Memory is
    generic(WORD_BITS : integer range 1 to 128 := 32;
            ADDR_BITS : integer range 1 to 10 := 5);
    port(
        clk          : in  std_logic;
        -- asynchronous read port
        readAddr : in std_logic_vector(ADDR_BITS-1 downto 0);
        readData  : out std_logic_vector(WORD_BITS-1 downto 0);

        -- synchronous write port
        writeAddr : in std_logic_vector(ADDR_BITS-1 downto 0);
        writeData : in std_logic_vector(WORD_BITS-1 downto 0);
        writeEnable : in std_logic);
end DP_Memory;
```



# Banco de registros (dual-port memory) – VHDL

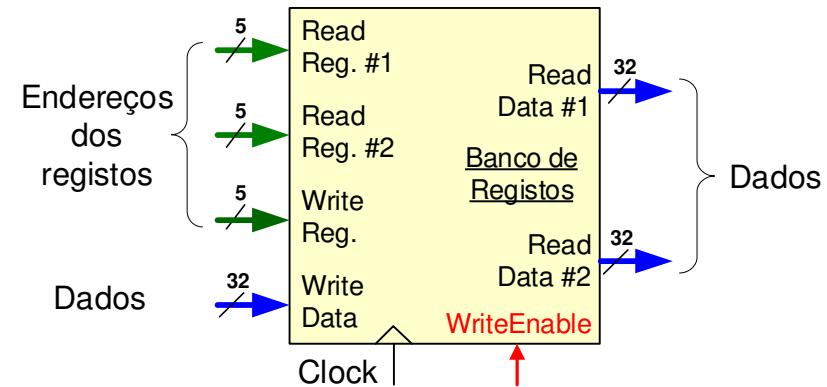
```
architecture Behavioral of DP_Memory is
    subtype TDataWord is std_logic_vector(WORD_BITS-1 downto 0);
    type TMem is array (0 to 2**ADDR_BITS-1) of TDataWord;
    signal s_memory : TMem := (others => (others => '0'));
begin
    process(clk, writeEnable) is
    begin
        if(rising_edge(clk) ) then
            if(writeEnable = '1') then
                s_memory(to_integer(unsigned(writeAddr))) <= writeData;
            end if;
        end if;
    end process;
    readData <= (others => '0') when
        (to_integer(unsigned(readAddr)) = 0) else
        s_memory(to_integer(unsigned(readAddr)));
end Behavioral;
```



# Banco de registros – VHDL

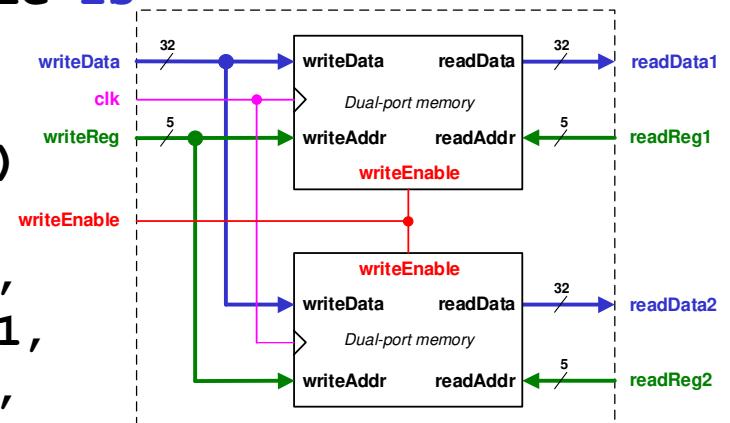
```
library ieee;
use ieee.std_logic_1164.all;

entity RegFile is
    port(clk           : in  std_logic;
          writeEnable   : in  std_logic;
          writeReg      : in  std_logic_vector( 4 downto 0);
          writeData     : in  std_logic_vector(31 downto 0);
          readReg1     : in  std_logic_vector( 4 downto 0);
          readReg2     : in  std_logic_vector( 4 downto 0);
          readData1    : out std_logic_vector(31 downto 0);
          readData2    : out std_logic_vector(31 downto 0));
end RegFile;
```



# Banco de registros – VHDL

```
architecture Structural of RegFile is
begin
rs_mem:
entity work.DP_Memory(Behavioral)
port map(clk      => clk,
         readAddr  => readReg1,
         readData   => readData1,
         writeAddr  => writeReg,
         writeData  => writeData,
         writeEnable => writeEnable);
rt_mem:
entity work.DP_Memory(Behavioral)
port map(clk      => clk,
         readAddr  => readReg2,
         readData   => readData2,
         writeAddr  => writeReg,
         writeData  => writeData,
         writeEnable => writeEnable);
end Structural;
```



# Módulo para separação dos campos da instrução – VHDL

```
entity InstrSplitter is
    port( instruction : in  std_logic_vector(31 downto 0);
          opcode      : out std_logic_vector(5  downto 0);
          rs          : out std_logic_vector(4  downto 0);
          rt          : out std_logic_vector(4  downto 0);
          rd          : out std_logic_vector(4  downto 0);
          shamt       : out std_logic_vector(4  downto 0);
          funct       : out std_logic_vector(5  downto 0);
          imm         : out std_logic_vector(15 downto 0);
          jAddr       : out std_logic_vector(25 downto 0));
end InstrSplitter;
architecture Behavioral of InstrSplitter is
begin
    opcode  <= instruction(31 downto 26);
    rs      <= instruction(25 downto 21);
    rt      <= instruction(20 downto 16);
    rd      <= instruction(15 downto 11);
    shamt   <= instruction(10 downto 6);
    funct   <= instruction( 5 downto 0);
    imm     <= instruction(15 downto 0);
    jAddr   <= instruction(25 downto 0);
end Behavioral;
```

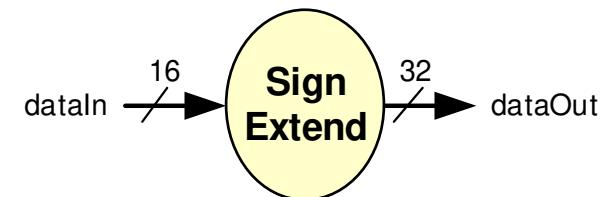


# Módulo de extensão de sinal – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

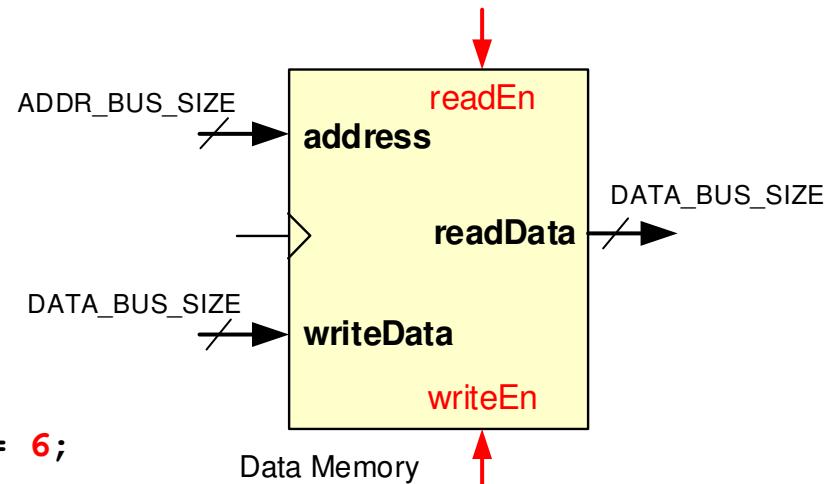
entity SignExtend is
    port(dataIn : in std_logic_vector(15 downto 0);
         dataOut : out std_logic_vector(31 downto 0));
end SignExtend;

architecture Behavioral of SignExtend is
begin
    dataOut(31 downto 16) <= (others => dataIn(15));
    dataOut(15 downto 0) <= dataIn;
end Behavioral;
```



# Módulo de memória RAM – VHDL

```
entity RAM is
    generic( ADDR_BUS_SIZE : positive := 6;
              DATA_BUS_SIZE : positive := 32);
    port(clk          : in  std_logic;
         readEn       : in  std_logic;
         writeEn      : in  std_logic;
         address      : in  std_logic_vector(ADDR_BUS_SIZE-1 downto 0);
         writeData    : in  std_logic_vector(DATA_BUS_SIZE-1 downto 0);
         readData     : out std_logic_vector(DATA_BUS_SIZE-1 downto 0));
end RAM;
```



# Módulo de memória RAM – VHDL

```
architecture Behavioral of RAM is
    constant NUM_WORDS : positive := (2 ** ADDR_BUS_SIZE );
    subtype TData is std_logic_vector(DATA_BUS_SIZE-1 downto 0);
    type TMemory is array(0 to NUM_WORDS - 1) of TData;
    signal s_memory : TMemory;
begin

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(writeEn = '1') then
                s_memory(to_integer(unsigned(address))) <= writeData;
            end if;
        end if;
    end process;
    readData <= s_memory(to_integer(unsigned(address))) when
        readEn = '1' else (others => '-');
end Behavioral;
```

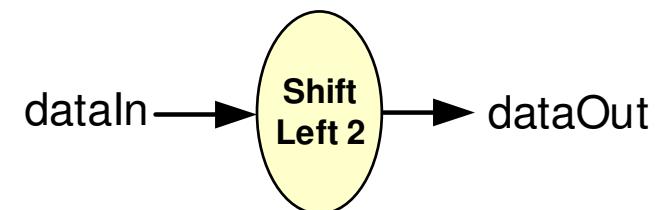


# Módulo "left shifter" – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity LeftShifter2 is
    port(dataIn : in std_logic_vector(31 downto 0);
         dataOut: out std_logic_vector(31 downto 0));
end LeftShifter2;

architecture Behavioral of LeftShifter2 is
begin
    dataOut <= dataIn(29 downto 0) & "00";
end Behavioral;
```



## Aulas 16 e 17

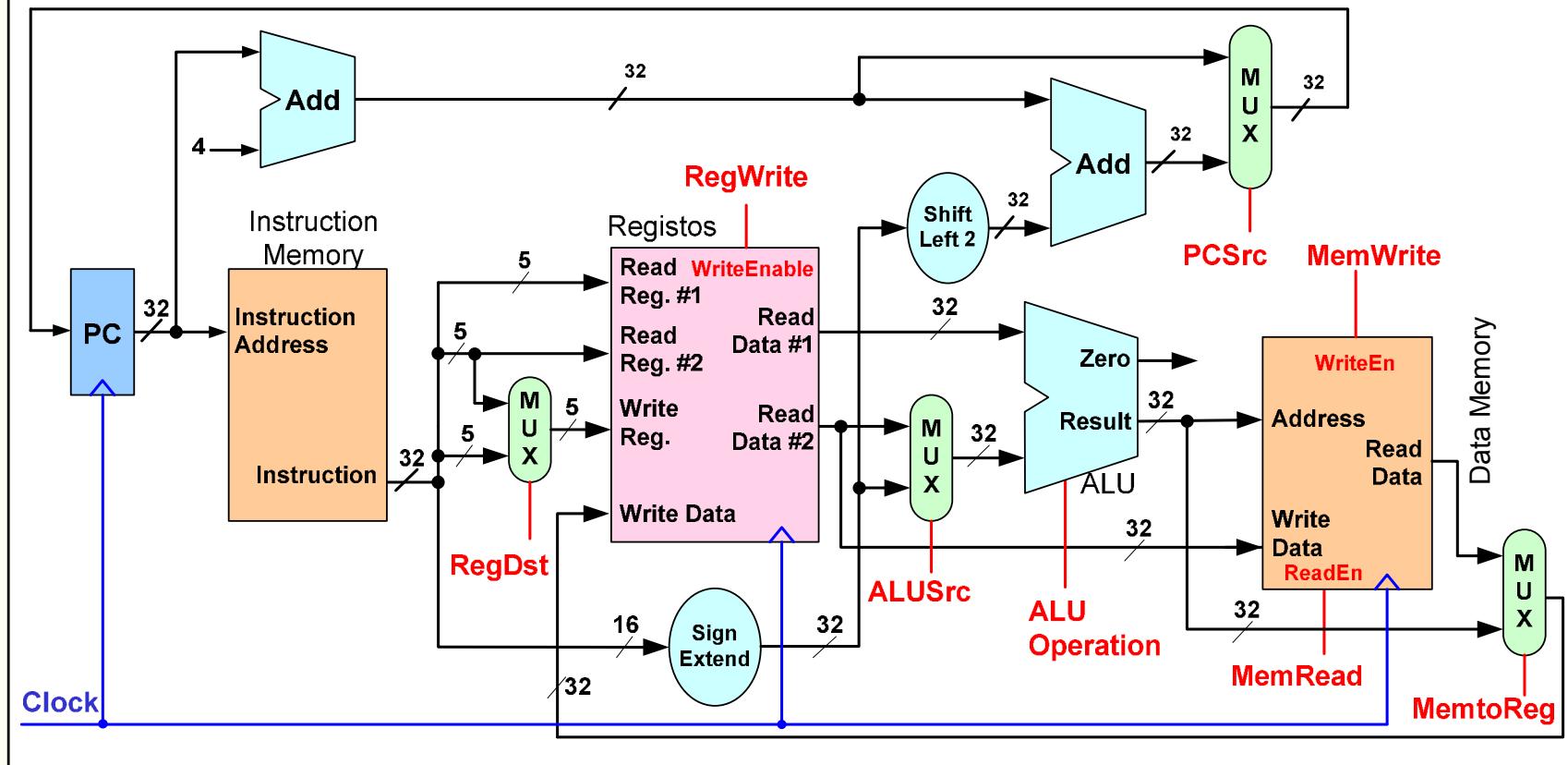
- A unidade de controlo principal do *datapath single-cycle*
- A unidade de controlo da ALU
- Desenho das unidades de controlo do *datapath* e da ALU
- Exemplos de funcionamento do *datapath* com unidade de controlo
- Suporte para a instrução *jump (j)*

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Datapath – unidade de controlo

- A unidade de controlo deve gerar os sinais de controlo (identificados a vermelho) para: 1) elementos de estado: banco de registos e memória de dados; 2) *multiplexers* e ALU

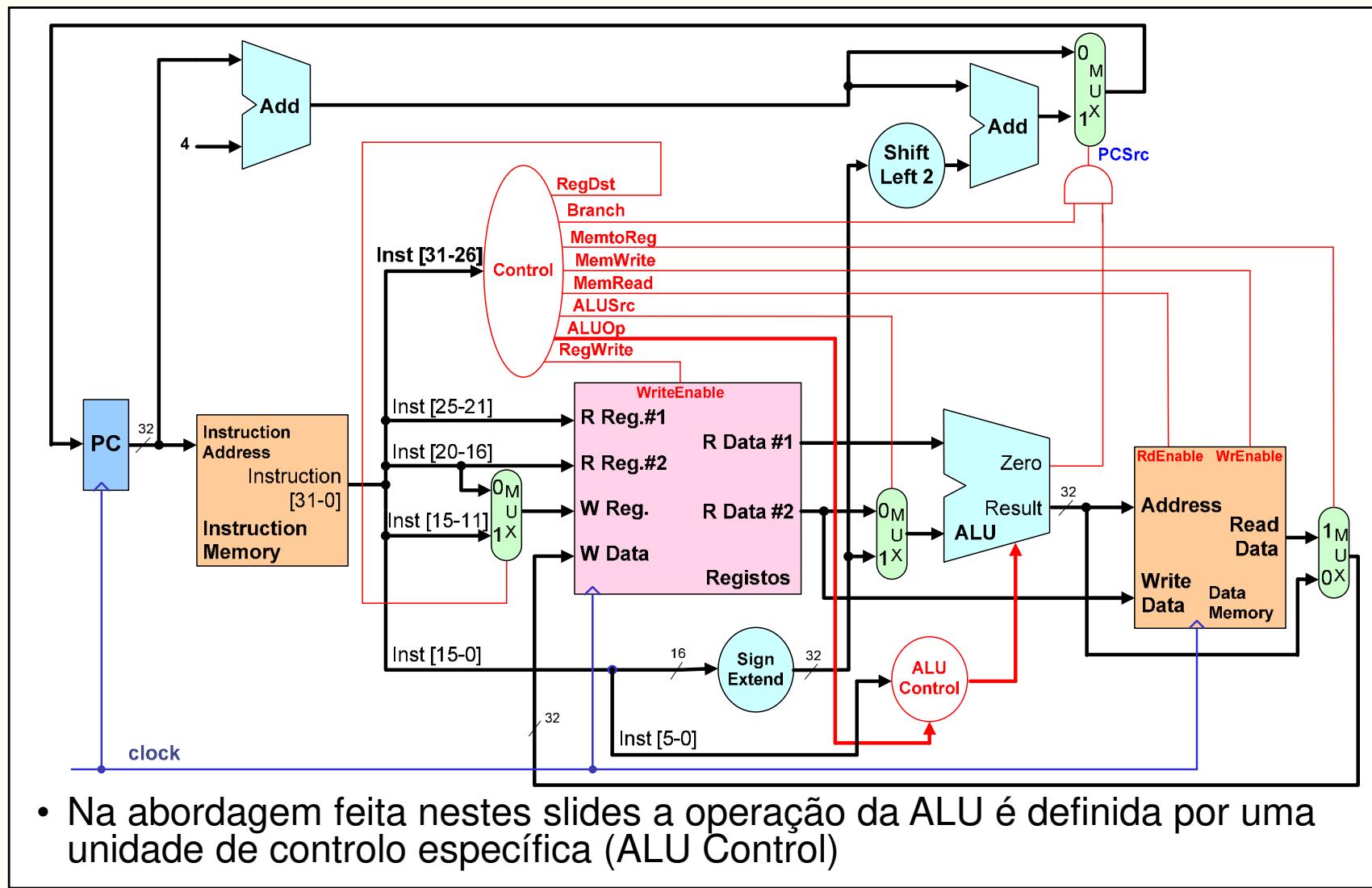


## Datapath – unidade de controlo

- Alguns dos elementos de estado presentes no *datapath* são acedidos em todos os ciclos de relógio (PC e memória de instruções). Nestes casos não há necessidade de explicitar um sinal de controlo
- Outros elementos de estado podem ser lidos ou escritos dependendo da instrução que estiver a ser executada (memória de dados e banco de registos). Para estes é necessário explicitar os respetivos sinais de controlo
- Para a ALU e para os elementos combinatórios que fazem o encaminhamento da informação (*multiplexers*) também é necessário definir os respetivos sinais de controlo
- A **escrita** nos elementos de estado é sempre realizada de forma síncrona



# *Datapath – unidade de controlo*



# Unidade de controlo da ALU

- As instruções básicas que fazem uso da ALU são:
  - **Load e store** – para calcular o endereço da memória externa
  - **Branch if equal / not equal** – para determinar se os operandos são iguais ou diferentes
  - **Aritméticas e lógicas** – para efetuar a respetiva operação
- A operação a realizar na ALU depende:
  - dos campos **opcode** e **funct** nas instruções aritméticas e lógicas de tipo R: **ALUControl = f(opcode, funct)**
  - do campo **opcode** nas restantes instruções:  
**ALUControl = f(opcode)**
- Assim, a geração dos sinais de controlo da ALU pode ser realizada em dois níveis:
  - Nível 1: **ALUOp = g (opcode)**
  - Nível 2: **ALUControl = f (ALUOp, funct)**



# Unidade de controlo da ALU

- A relação entre o tipo de instruções, o campo “funct”, a operação efetuada pela ALU e os sinais de controlo da mesma, pode ser resumida pela seguinte tabela

| ALU Control | ALU Action       |
|-------------|------------------|
| 0 0 0       | And              |
| 0 0 1       | Or               |
| 0 1 0       | Add              |
| 1 1 0       | Subtract         |
| 1 1 1       | Set if Less Than |

| Instruction          | OpCode           | Funct  | ALU Action       | ALUOp | ALU Control |
|----------------------|------------------|--------|------------------|-------|-------------|
| load word            | 100011 ("lw")    | xxxxxx | add              | 00    | 010         |
| store word           | 101011 ("sw")    | xxxxxx | add              | 00    | 010         |
| addi                 | 001000 ("addi")  | xxxxxx | add              | 00    | 010         |
| branch if equal      | 000100 ("beq" )  | xxxxxx | subtract         | 01    | 110         |
| add                  | 000000 (R-Type)  | 100000 | add              | 10    | 010         |
| subtract             | 000000 (R-Type)  | 100010 | subtract         | 10    | 110         |
| and                  | 000000 (R-Type)  | 100100 | and              | 10    | 000         |
| or                   | 000000 (R-Type)  | 100101 | or               | 10    | 001         |
| set if less than     | 000000 (R-Type)  | 101010 | set if less than | 10    | 111         |
| set if less than imm | 001010 ("slti" ) | xxxxxx | set if less than | 11    | 111         |

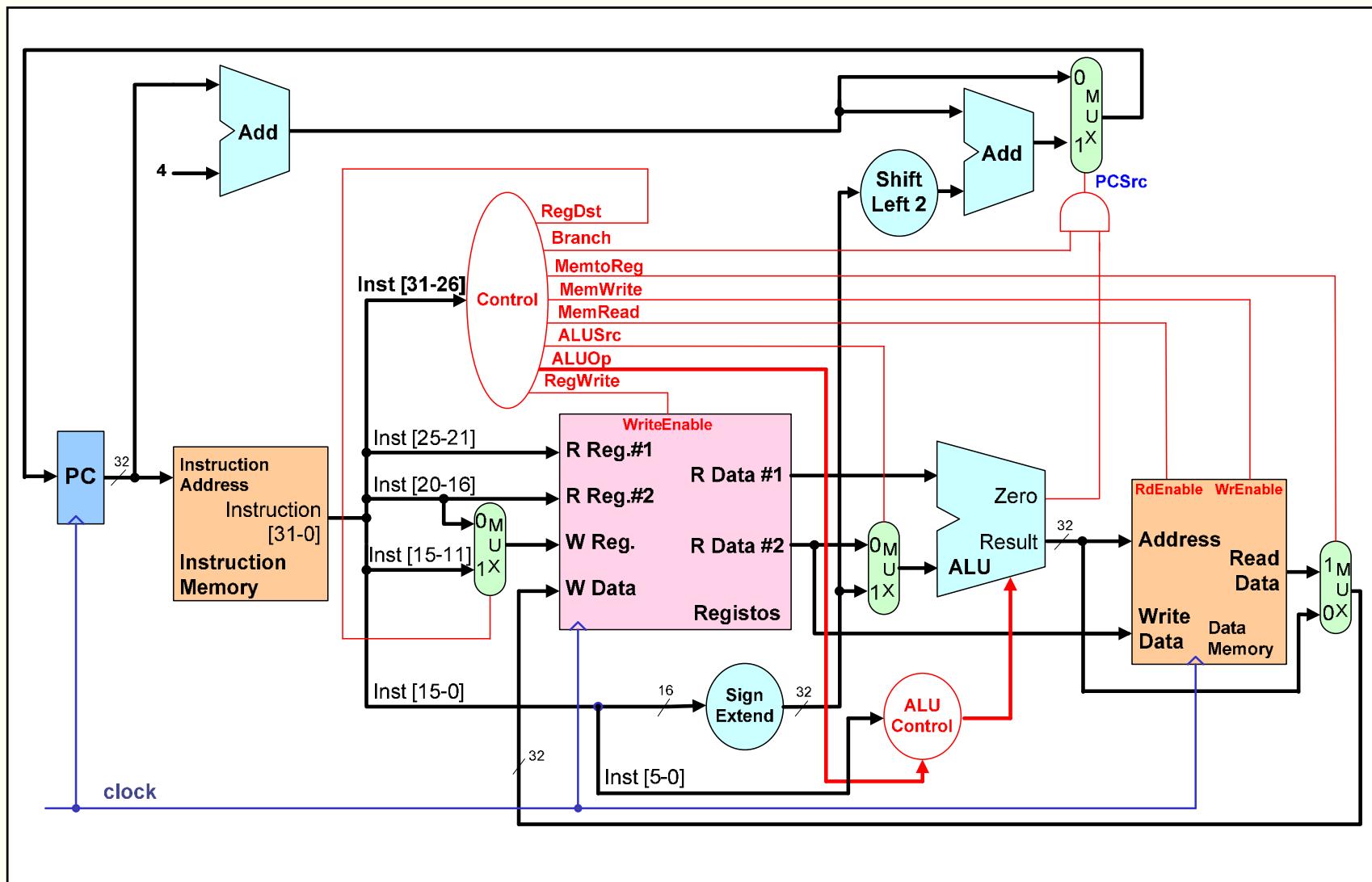


# Unidade de controlo principal

- O desenho da unidade de controlo principal do nosso CPU simplificado apoia-se na observação de um conjunto de factos que decorrem da forma como são codificadas as instruções do MIPS:
  - O campo **op** (Operation Code) está situado nos bits **31-26 de todas as instruções**
  - Os índices dos 2 registos que devem ser lidos (nas instruções em que tal se aplica), surgem sempre nos bits **25-21 (rs)** e **20-16 (rt)**.
  - Nas instruções **load/store**, o **registro base de endereçamento** está sempre nos bits **25-21 (rs)**
  - As **constantes ou offsets** surgem sempre nos bits **15-0** da instrução (à excepção do “j” em que a constante surge nos bits 25-0)
  - O **registro destino** (quando se aplique) pode aparecer em um de dois campos: nos **bits 20-16** (lw, addi, slti), ou nos **bits 15-11** (instruções aritméticas e lógicas de tipo R)



# Unidade de controlo principal



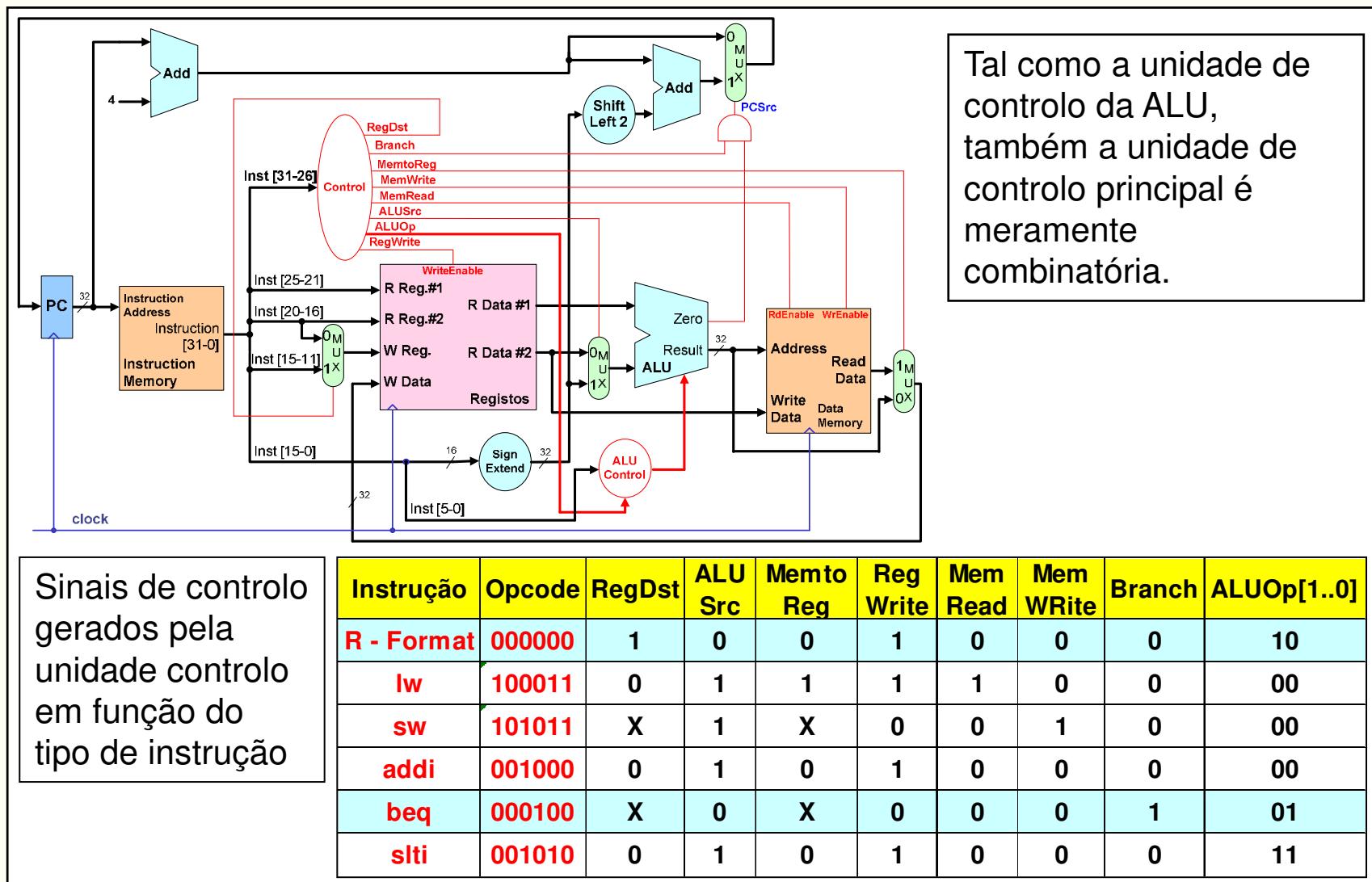
# Unidade de controlo principal

- Teremos assim de especificar um total de sete (+1) sinais de controlo (para além do ALUOp). São eles:

| Sinal    | Efeito quando não ativo ('0')   | Efeito quando ativo ('1')  |
|----------|---|--|
| MemRead  | Nenhum  | O conteúdo da memória de dados no endereço indicado é apresentado à saída  |
| MemWrite | Nenhum  | O conteúdo do registo de memória de dados cujo endereço é fornecido é substituído pelo valor apresentado à entrada |
| ALUSrc   | O segundo operando da ALU provém da segunda saída do <i>File Register</i> | O segundo operando da ALU provém dos 16 bits menos significativos da instrução após extensão do sinal              |
| RegDst   | O endereço do registo destino provém do campo <b>rt</b>                   | O endereço do registo destino provém do campo <b>rd</b>  |
| RegWrite | Nenhum  | O registo indicado no endereço de escrita é alterado pelo valor presente na entrada de dados                       |
| MemtoReg | O valor apresentado para escrita no registo destino provém da ALU         | O valor apresentado na entrada de dados dos registos internos provém da memória externa                            |
| PCSrc    | O PC é substituído pelo seu valor actual mais 4                           | O PC é substituído pelo resultado do somador que calcula o endereço target do <i>branch</i> condicional            |
| Branch   | Nenhum  | Indica que a instrução é um branch condicional   |



# Unidade de controlo principal



# Análise do funcionamento do *datapath*

- A execução de qualquer uma das instruções suportadas ocorre no intervalo de tempo correspondente a um único ciclo de relógio: tem início numa transição ativa do relógio e termina na transição ativa seguinte
- Para simplificar a análise podemos, no entanto, admitir que a utilização dos vários elementos operativos é “sequencial” e decorre ao longo de um conjunto de operações que culminam com:
  - escrita no Banco de Registos: instruções tipo R, LW, ADDI, SLTI
  - escrita na Memória de Dados: SW
- O *Program Counter* é sempre atualizado com:
  - endereço-alvo da instrução BEQ, se os registos forem iguais (*branch taken*), ou PC+4 se forem diferentes (*branch not taken*)
  - endereço-alvo da instrução J
  - PC+4 nas restantes instruções



## Análise do funcionamento do *datapath* – operações

- *Fetch* de uma instrução e cálculo do endereço da próxima instrução
- Leitura de dois registo do Banco de Registo
- A ALU opera sobre dois valores (a fonte dos valores a operar depende do tipo de instrução que estiver a ser executada)
- O resultado da operação efetuada na ALU:
  - é escrito no Banco de Registos (**R-Type**, **addi** e **slti**)
  - é usado como endereço para escrever na memória de dados (**sw**)
  - é usado como endereço para fazer uma leitura da memória de dados (**lw**) - o valor lido da memória de dados é depois escrito no Banco de Registos
  - é usado para decidir qual o próximo valor do PC (**beq** / **bne**)

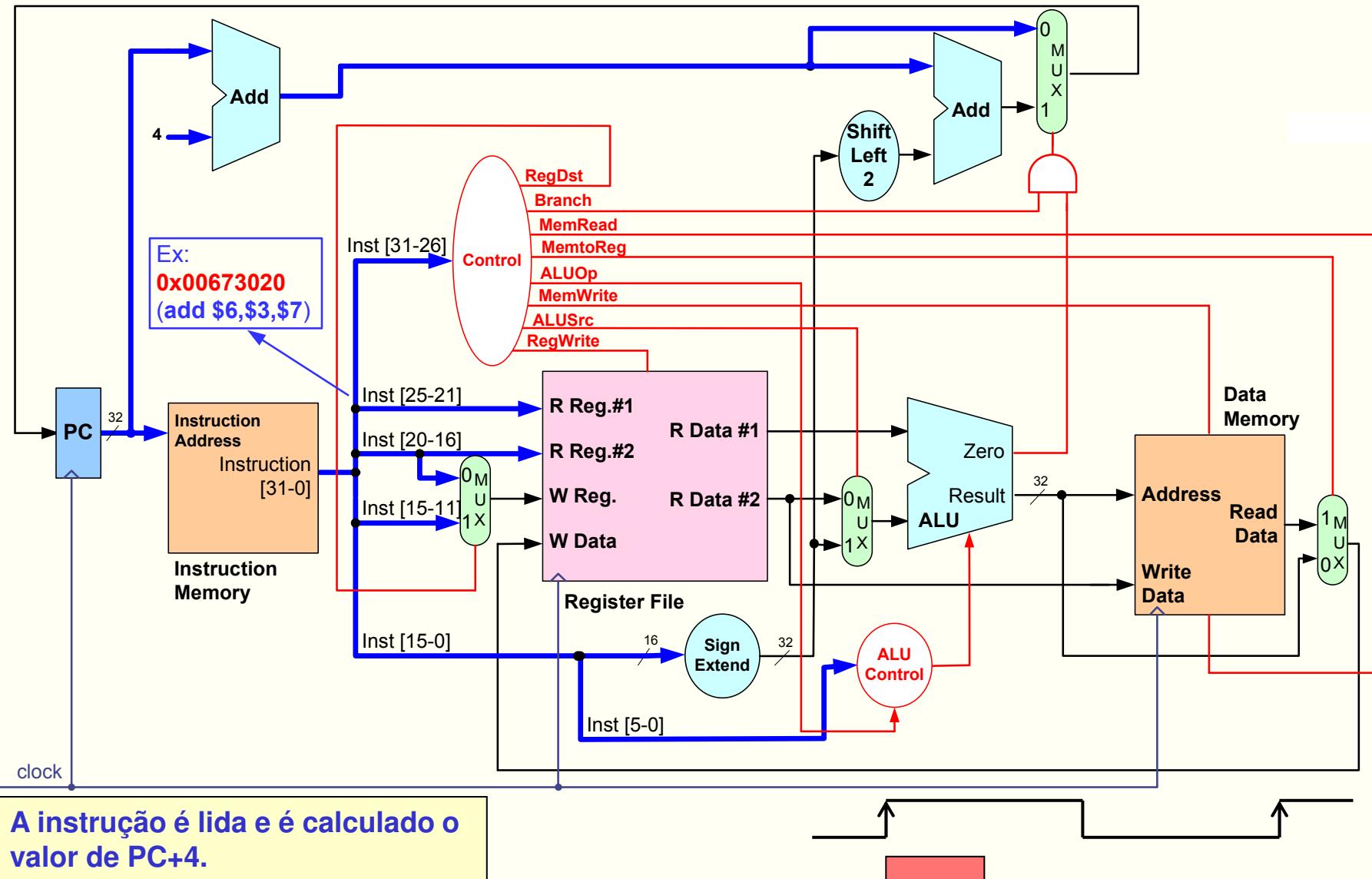


## Funcionamento do *datapath* nas instruções do tipo R

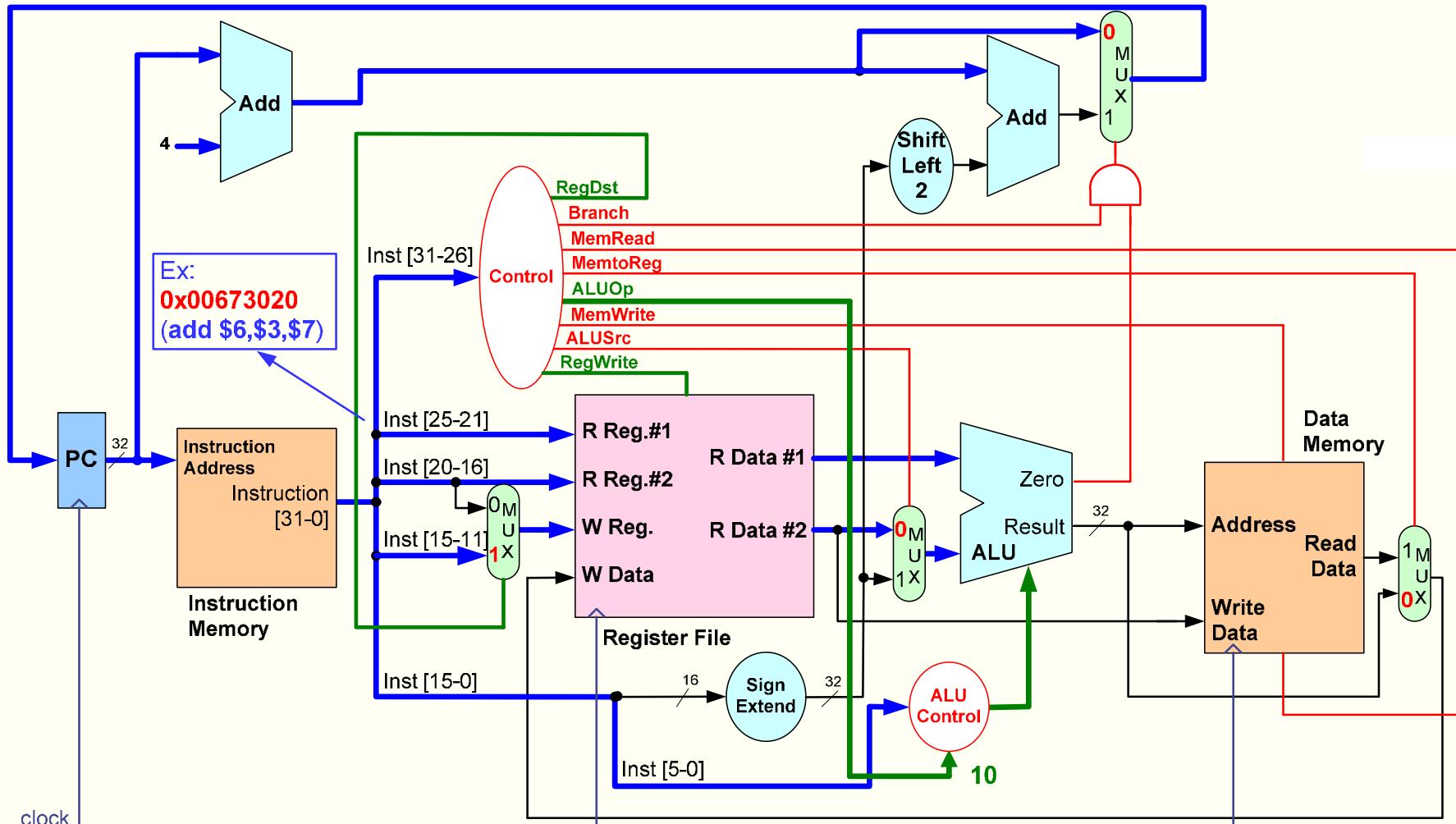
- A instrução é lida e é calculado o valor de PC+4.
- São lidos dois registos e a unidade de controlo determina, a partir do *opcode* (**bits 31-26**), o estado dos sinais de controlo.
- A ALU opera sobre os dados lidos dos dois registos, de acordo com a função codificada **nos bits 5-0** da instrução.
- O resultado produzido pela ALU será escrito no registo especificado nos **bits 15-11** da instrução (rd), na próxima transição ativa do relógio.



# Funcionamento do datapath nas instruções tipo R (1)



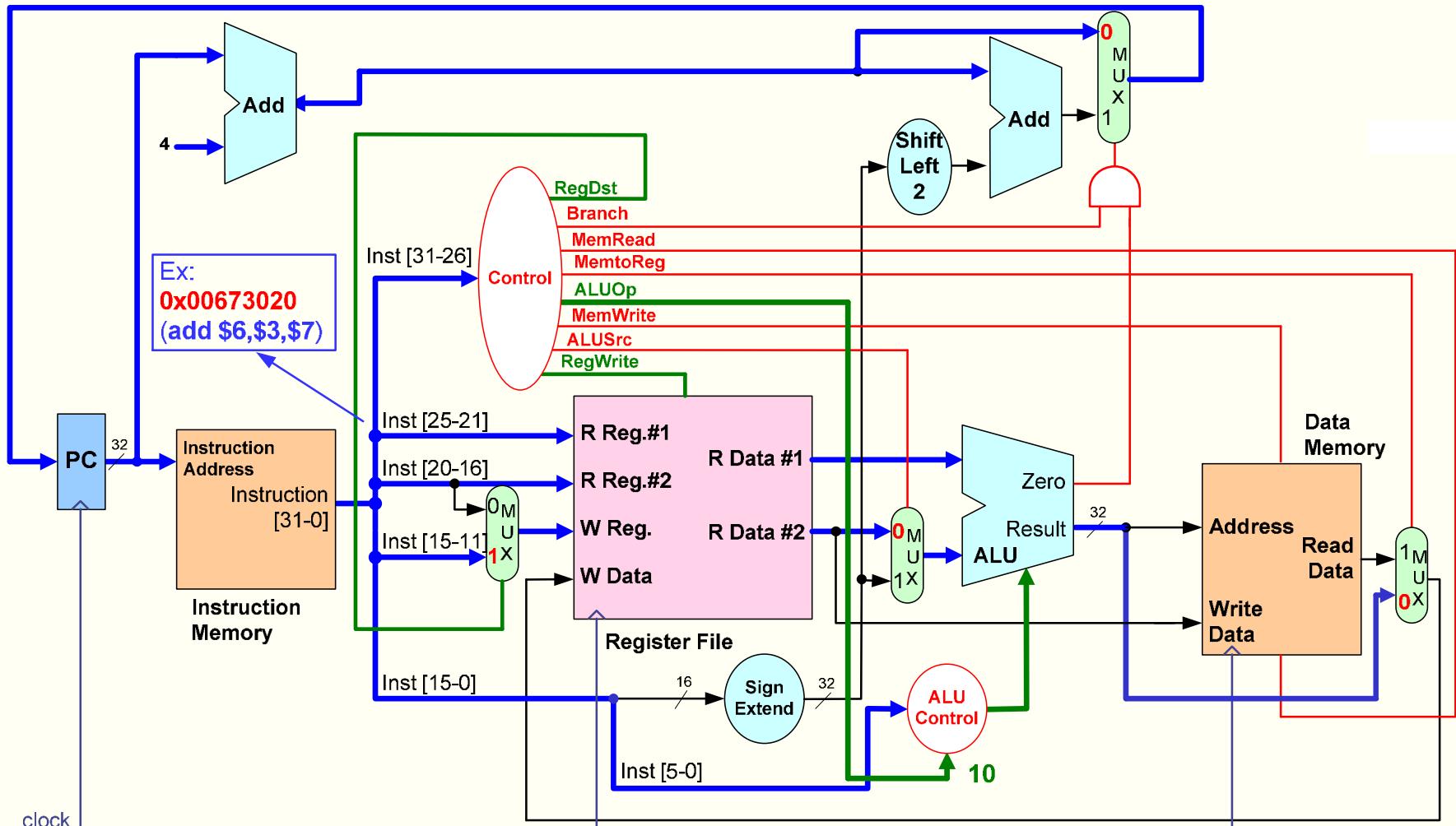
## Funcionamento do datapath nas instruções tipo R (2)



São lidos dois registos e a unidade de controlo determina, a partir do opcode (bits 31-26), o estado dos sinais de controlo.

Aulas 16,17 - 15

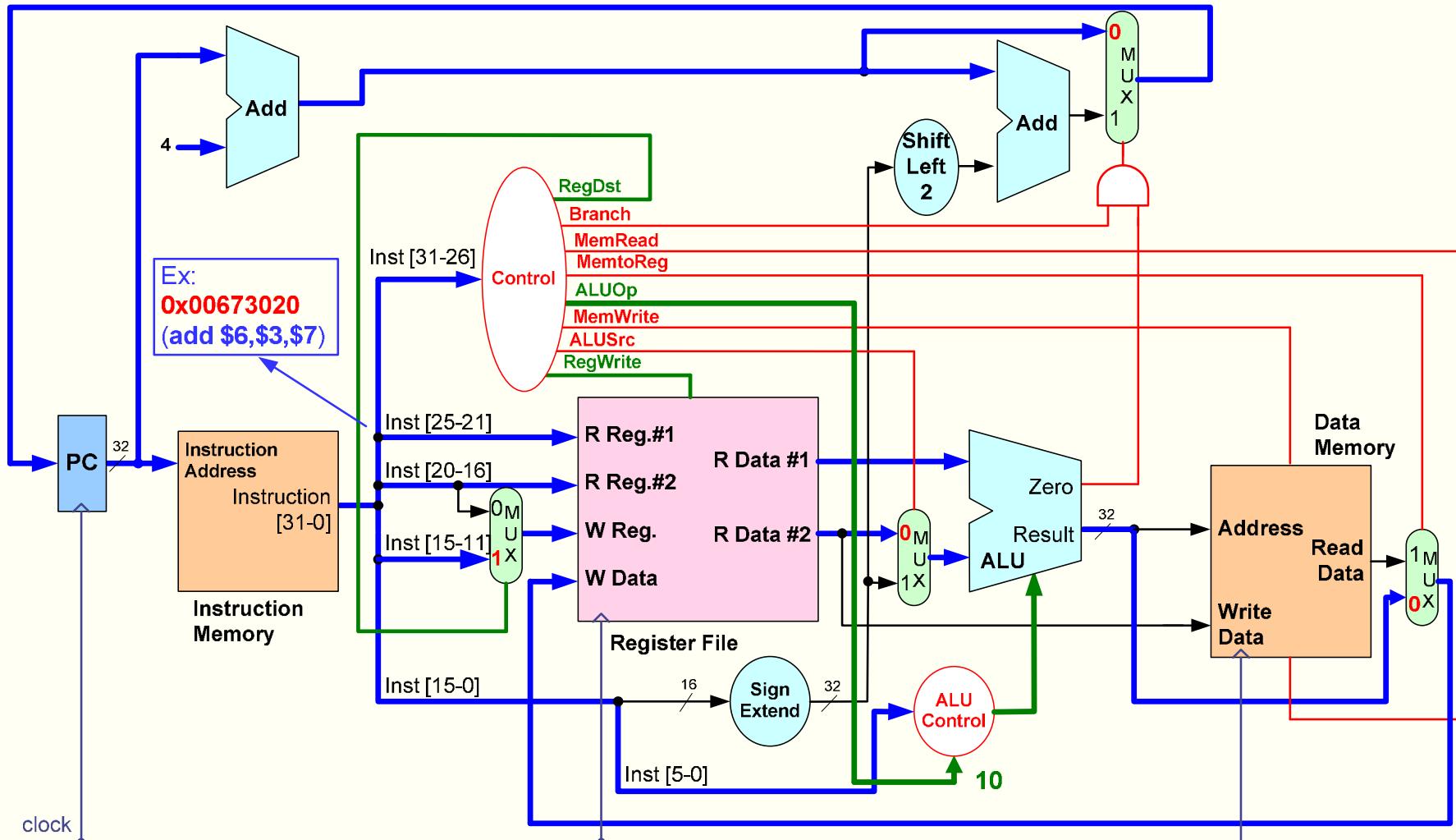
## Funcionamento do datapath nas instruções tipo R (3)



A ALU opera sobre os dados lidos dos dois registos, de acordo com a função codificada nos bits [5-0] da instrução.

Aulas 16,17 - 16

## Funcionamento do datapath nas instruções tipo R (4)



O resultado produzido pela ALU será escrito no registro especificado nos bits 15-11 da instrução (rd), na próxima transição ativa do relógio.

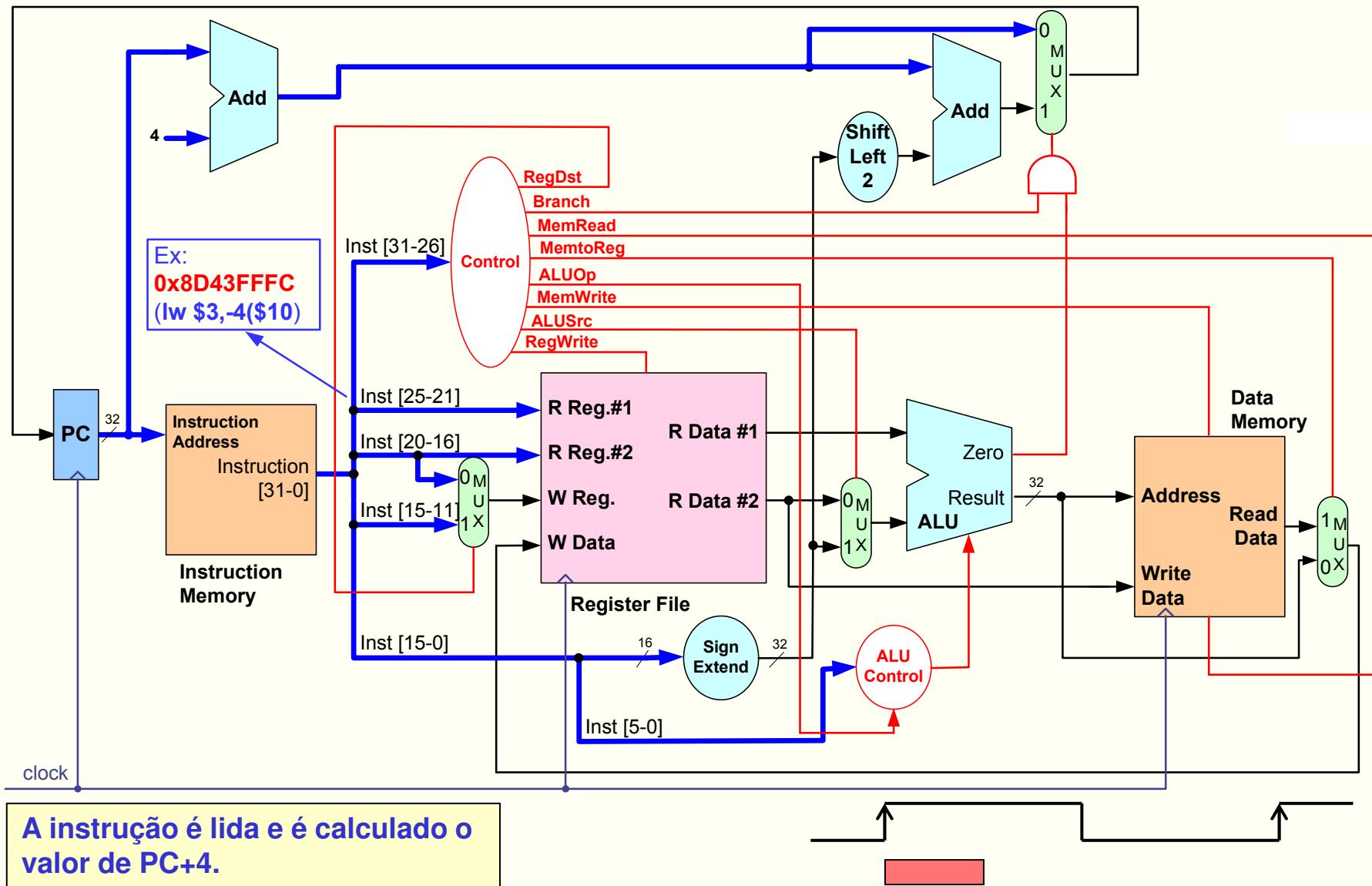
Aulas 16,17 - 17

## Funcionamento do *datapath* na instrução *load word* ("lw")

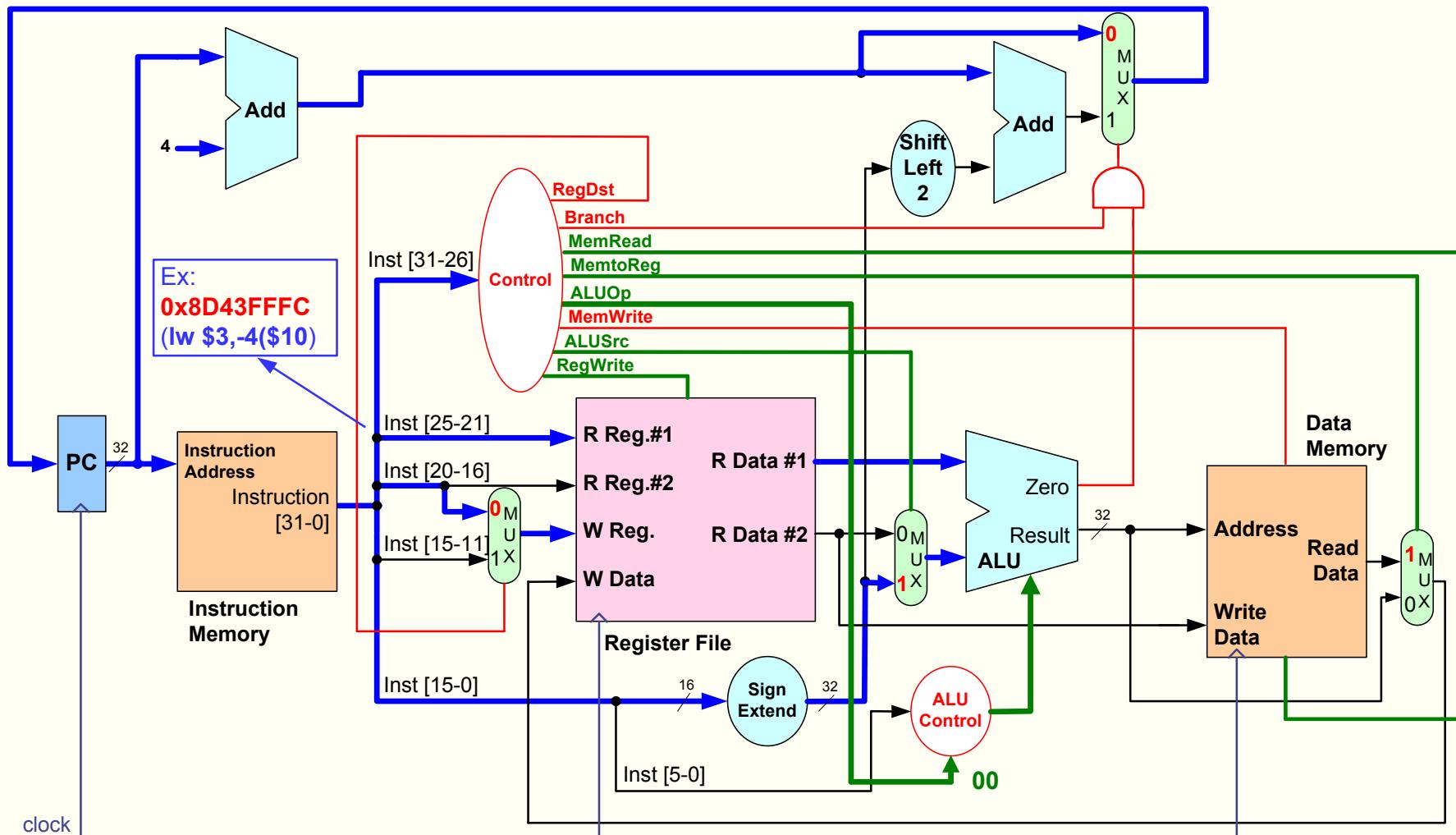
- A instrução é lida e é calculado o valor de PC+4.
- É lido um registo e a unidade de controlo determina, a partir do *opcode*, o estado dos sinais de controlo.
- A ALU soma o valor lido do registo especificado nos **bites 25-21** (rs) com os 16 bits (extendidos com sinal para 32) do campo *offset* da instrução (**bites15-0**).
- O resultado produzido pela ALU constitui o endereço de acesso à memória de dados. A memória é lida nesse endereço.
- A *word* lida da memória será escrita no registo especificado nos **bites 20-16** da instrução (rt), na próxima transição ativa do relógio.



# Funcionamento do datapath na instrução *load word* (1)



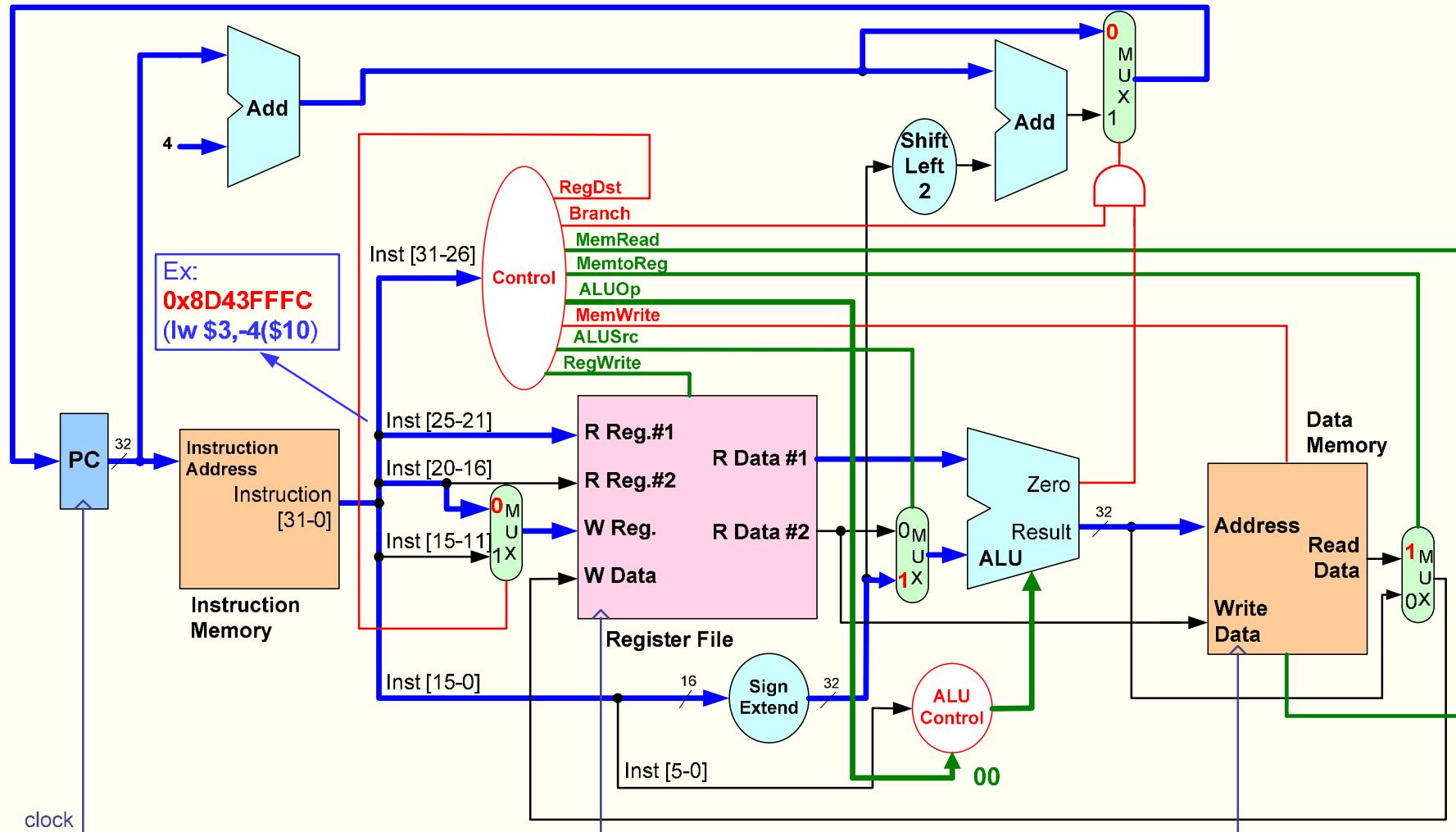
## Funcionamento do datapath na instrução *load word* (2)



É lido um registo e a unidade de controlo determina, a partir do opcode, o estado dos sinal de controlo.

Aulas 16,17 - 20

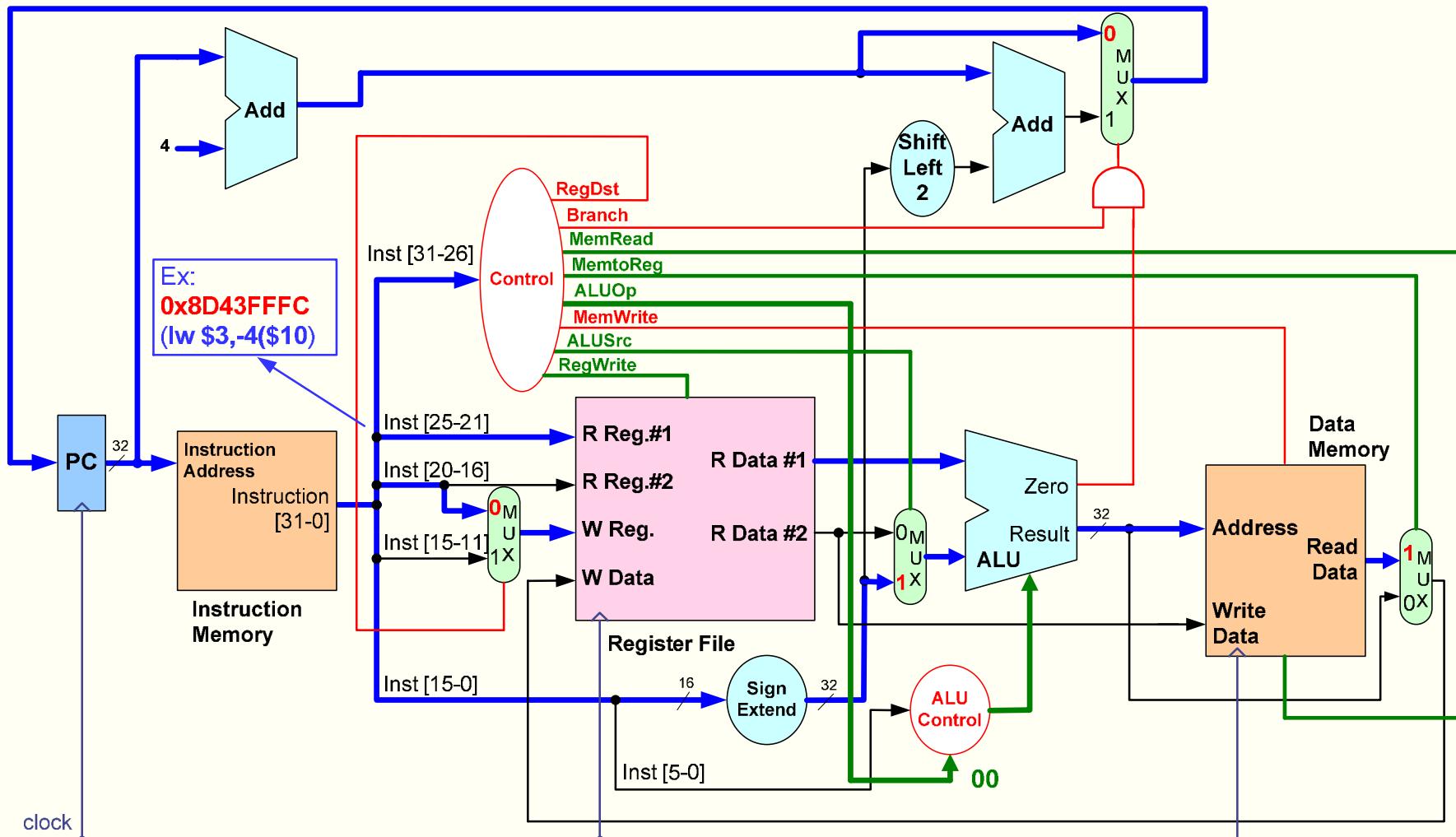
## Funcionamento do datapath na instrução *load word* (3)



A ALU soma o valor lido do registo com os 16 bits (extendidos com sinal para 32) do campo offset da instrução (bits15-0).

Aulas 16,17 - 21

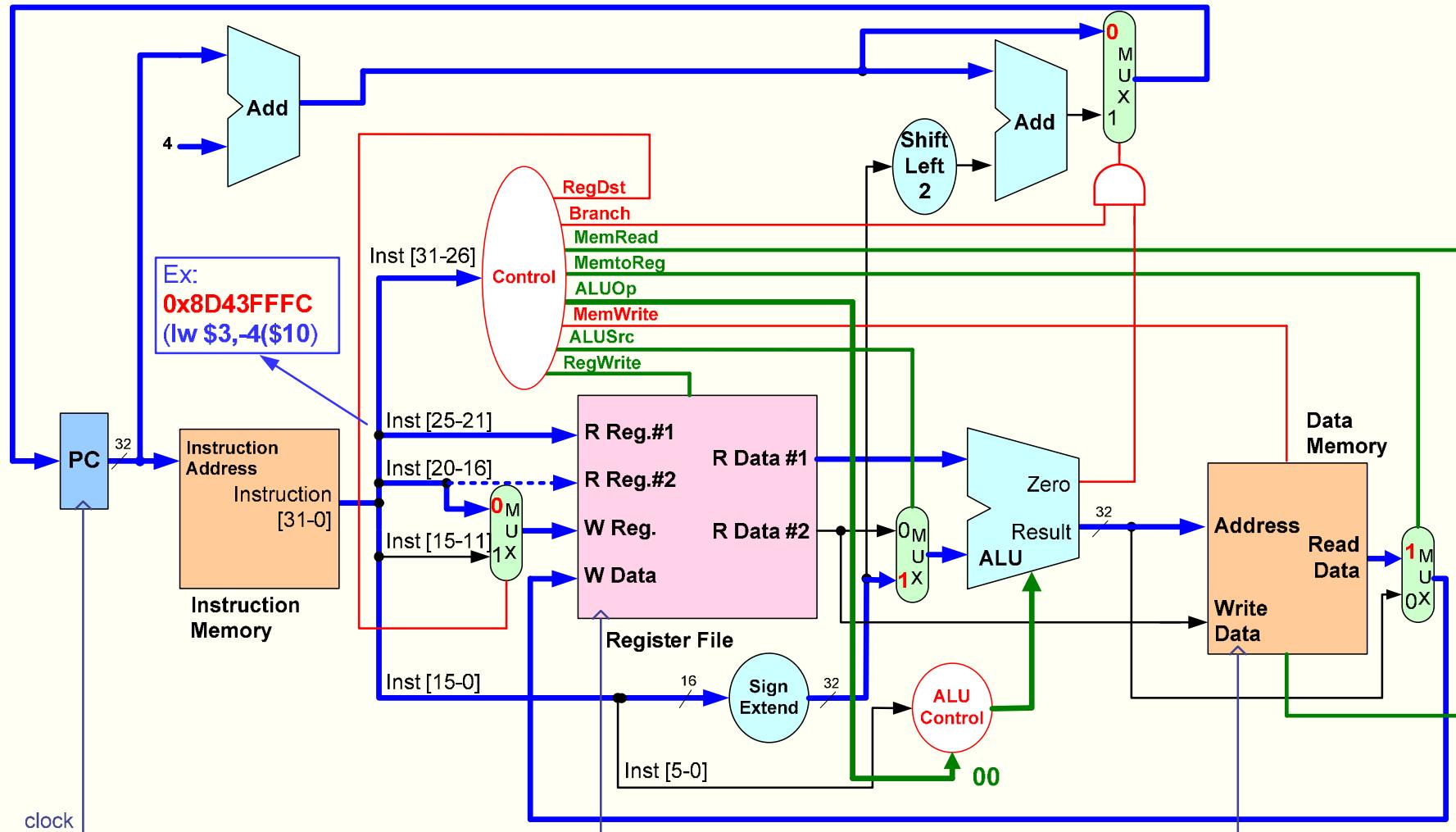
## Funcionamento do datapath na instrução *load word* (4)



O resultado produzido pela ALU constitui o endereço de acesso à memória de dados.  
A memória é lida nesse endereço.

Aulas 16,17 - 22

## Funcionamento do datapath na instrução *load word* (5)



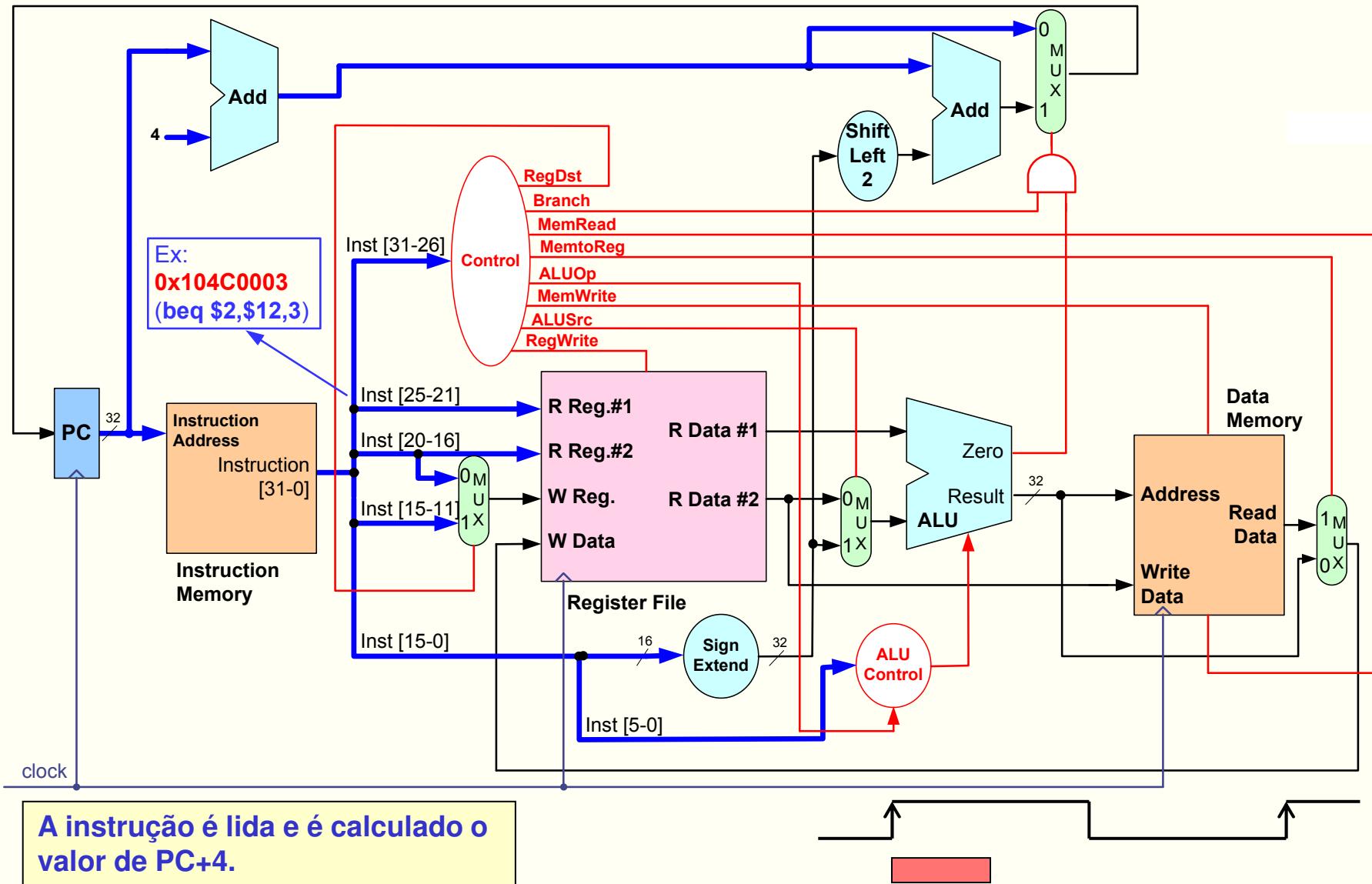
A word lida da memória será escrita no registro especificado nos bits 20-16 da instrução (rt), na próxima transição ativa do relógio.

## Funcionamento do *datapath* na instrução *branch if equal*

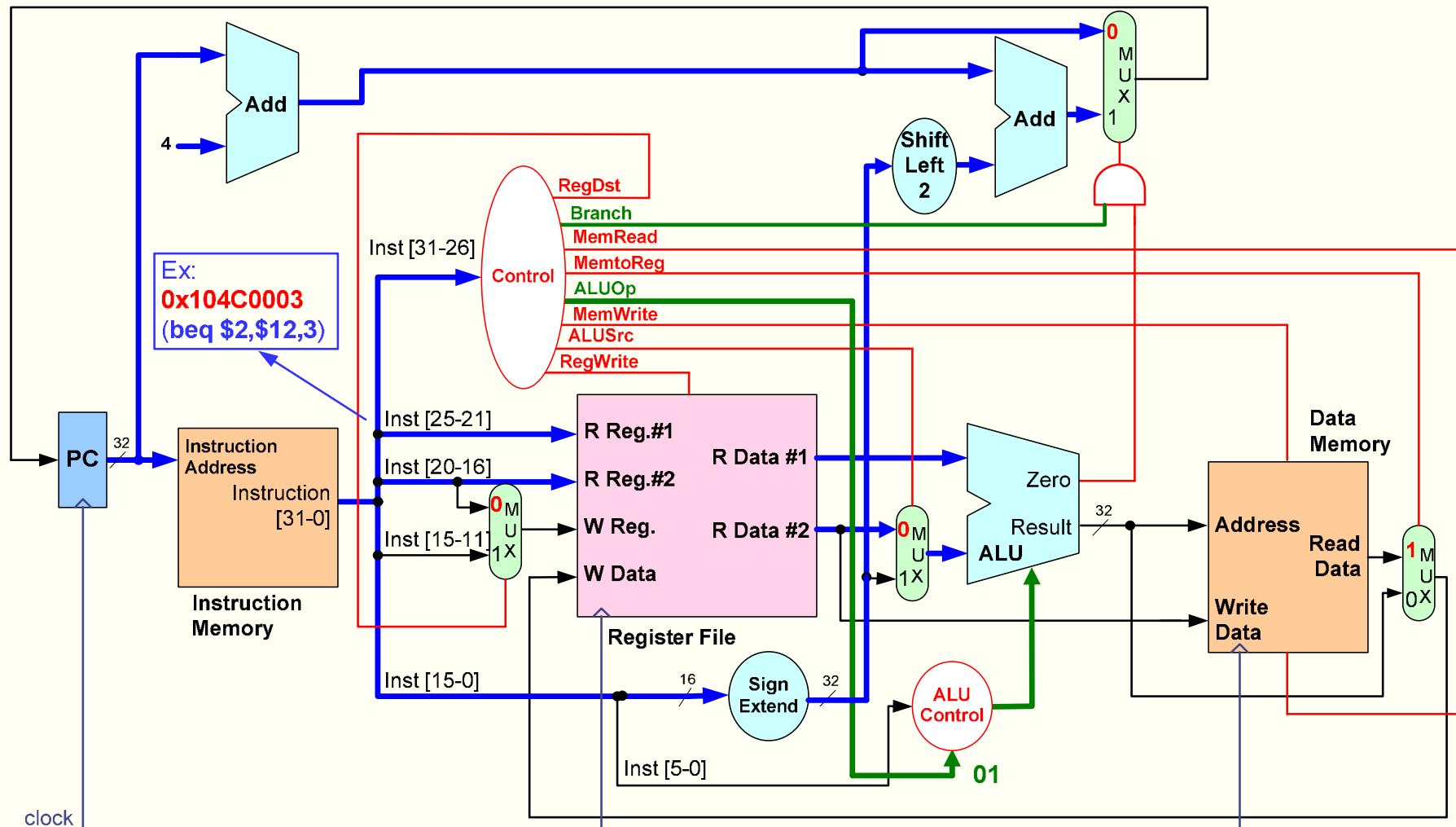
- A instrução é lida e é calculado o valor de PC+4.
- São lidos dois registos e é determinado o estado dos sinais de controlo. Os 16LSBs da instrução (sign extended x 4) são somados a PC+4 (BTA).
- A ALU faz a subtração dos dois valores lidos dos registos.
- A saída "Zero" da ALU é utilizada para decidir qual o próximo valor do PC, que será atualizado na próxima transição ativa do relógio.



# Funcionamento do datapath na instrução "beq" (1)

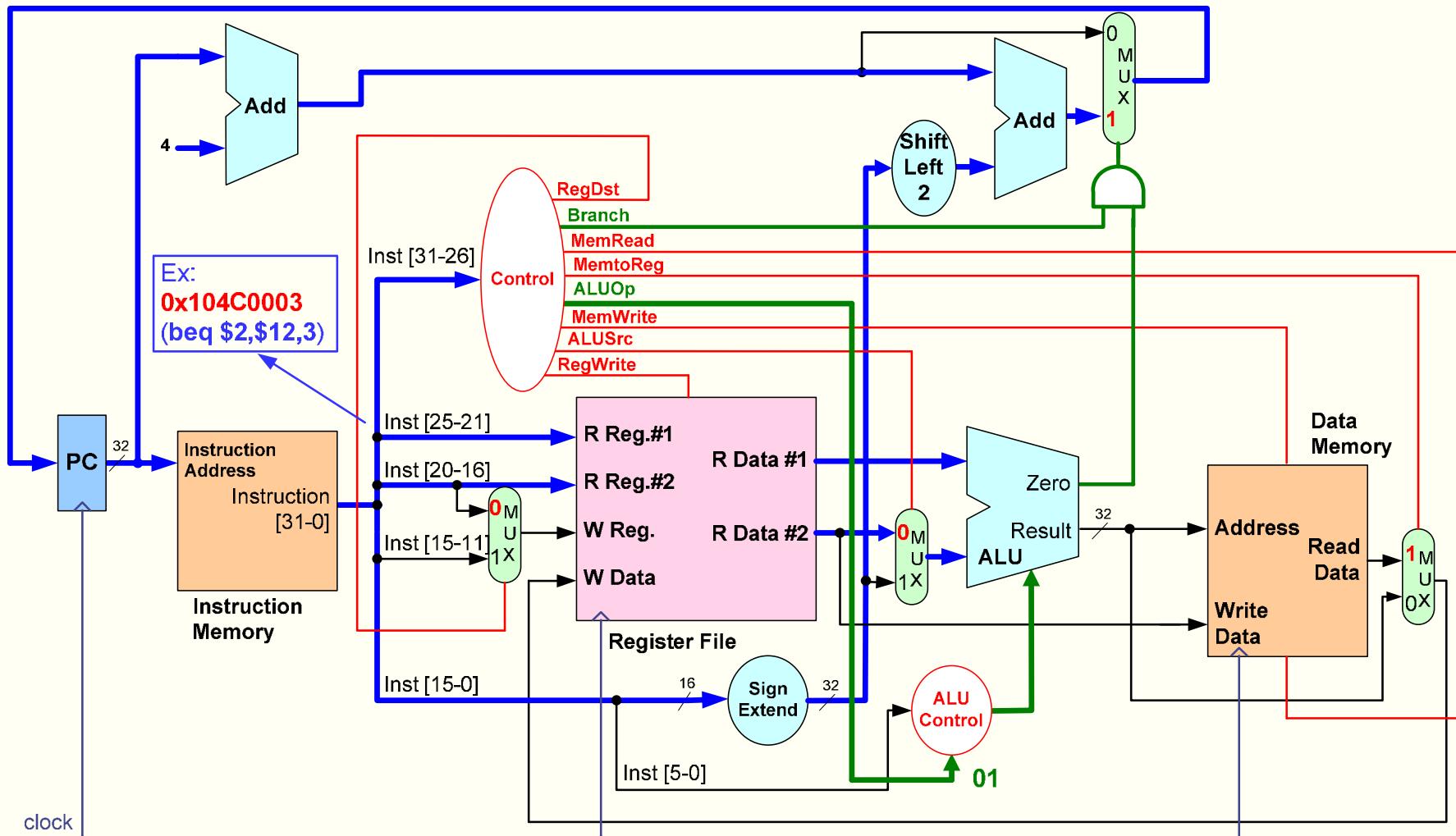


## Funcionamento do datapath na instrução "beq" (2)



São lidos dois registos e é determinado o estado dos sinais de controlo. Os 16LSBs da instrução (sign extended x 4) são somados a PC+4.

## Funcionamento do datapath na instrução "beq" (3)



A ALU efetua a subtração dos dois valores lidos dos registos. A saída "Zero" da ALU é utilizada para decidir qual o próximo valor do PC, que será atualizado na próxima transição ativa do relógio.

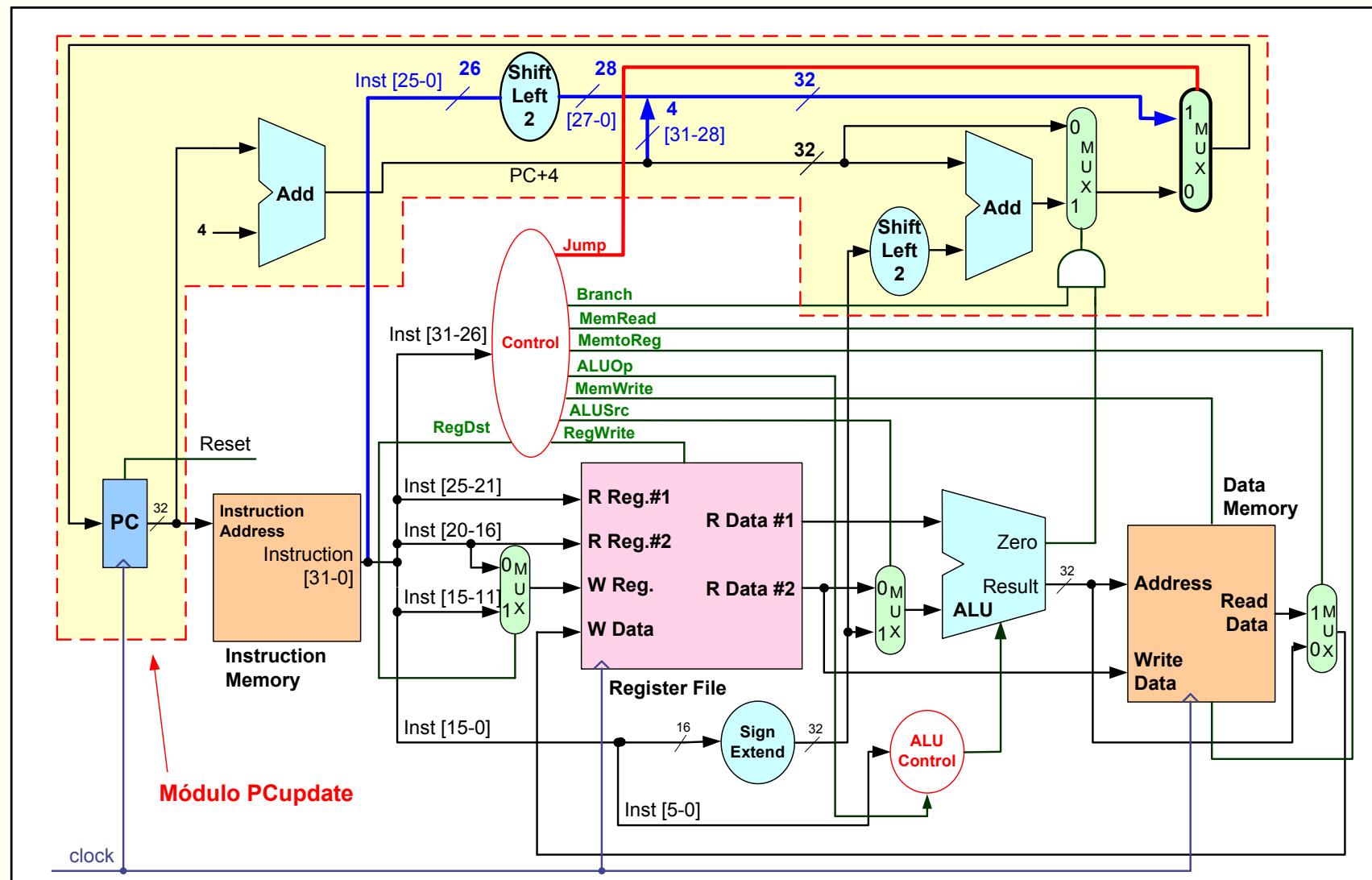
Aulas 16,17 - 27

## Datapath com suporte para a instrução "jump"

- A instrução “jump” corresponde a um caso particular de codificação (instruções do tipo J). Nestas instruções existem apenas dois campos: o campo op (**bits 31-26**) e o campo de endereço (**bits 25-0**)
- Nas instruções de “jump”, o endereço alvo (*Jump Target Address*) obtém-se pela **concatenação** dos bits **31-28** do PC+4 com os bits do campo de endereço da instrução (26 bits) multiplicados por 4.
- Será necessário acrescentar ainda um bit de saída à unidade de controlo para selecionar a fonte de informação disponibilizada à entrada do PC
- O *datapath* simplificado, com suporte para a instrução “j” (“jump”), fica com a configuração do slide seguinte



# Datapath com suporte para a instrução “jump”

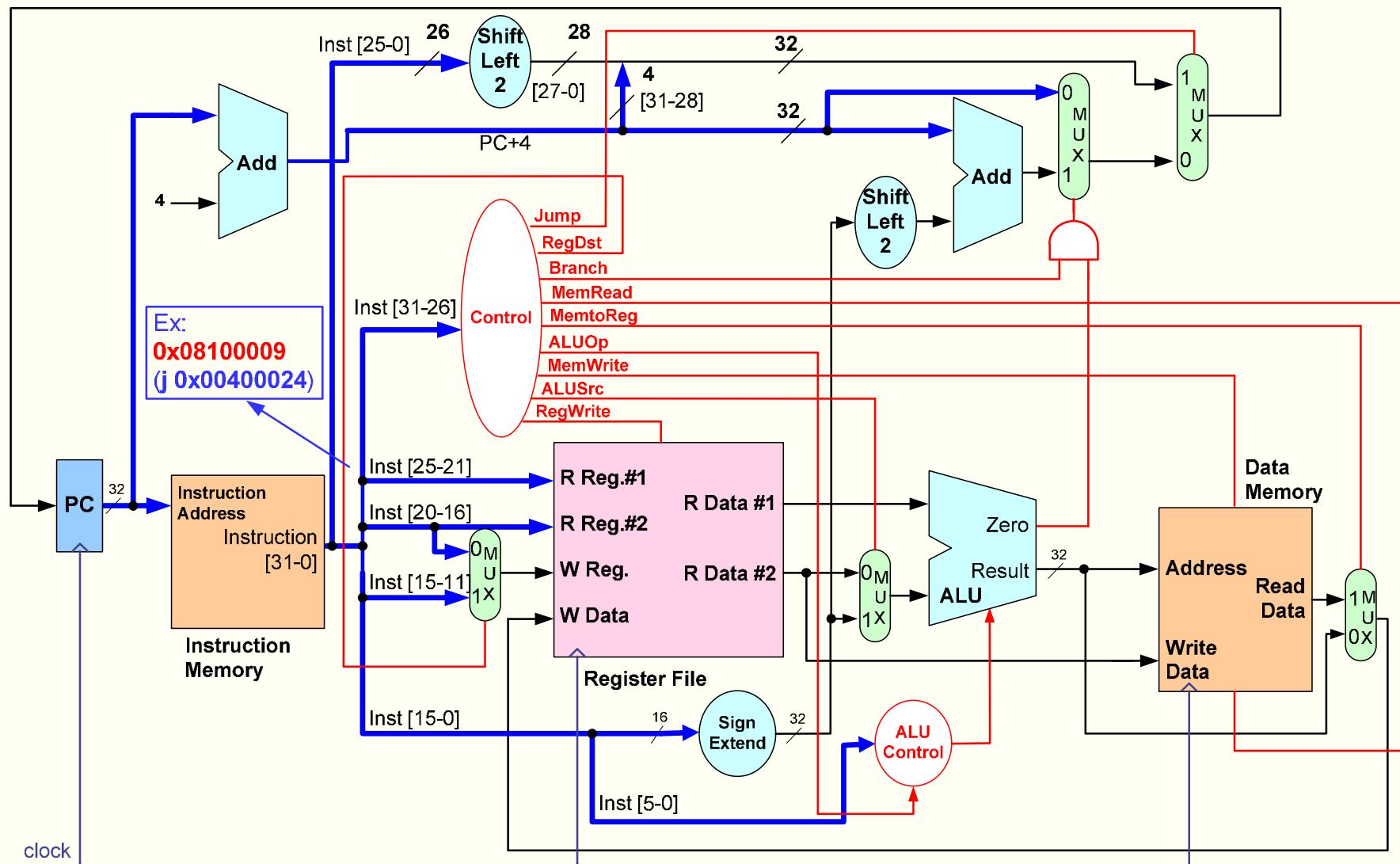


# Funcionamento do *datapath* na instrução J

- A instrução é lida e é calculado o valor de PC+4.
- São determinados os sinais de controlo. O novo valor do PC é obtido a partir dos 26 LSB da instrução multiplicados por 4 (*shift left 2*) concatenados com os 4 bits mais significativos do PC atual.



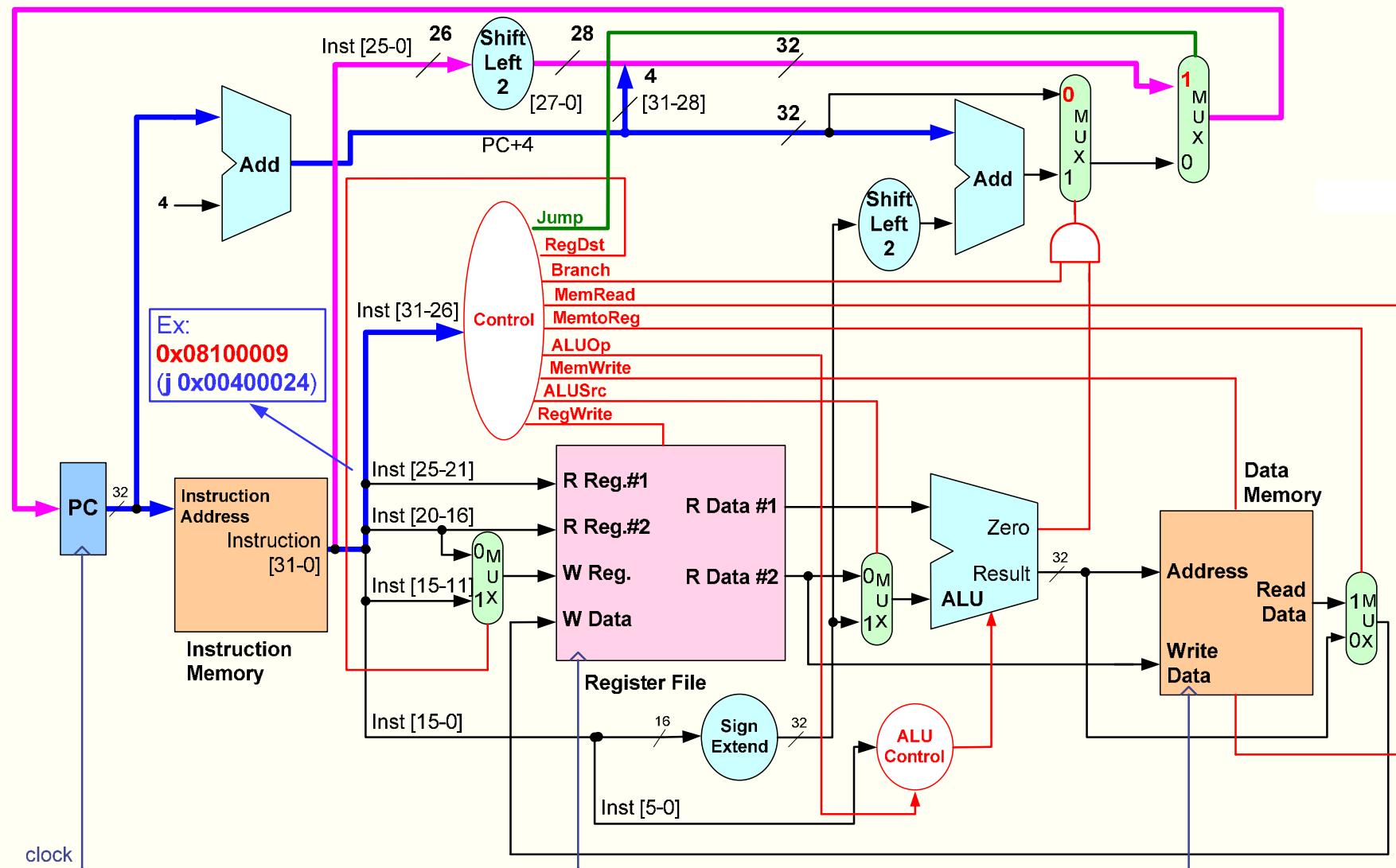
## Funcionamento do datapath na instrução J (1)



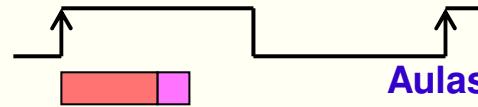
**A instrução é lida e é calculado o valor de PC+4.**

Aulas 16,17 - 31

## Funcionamento do datapath na instrução J (2)

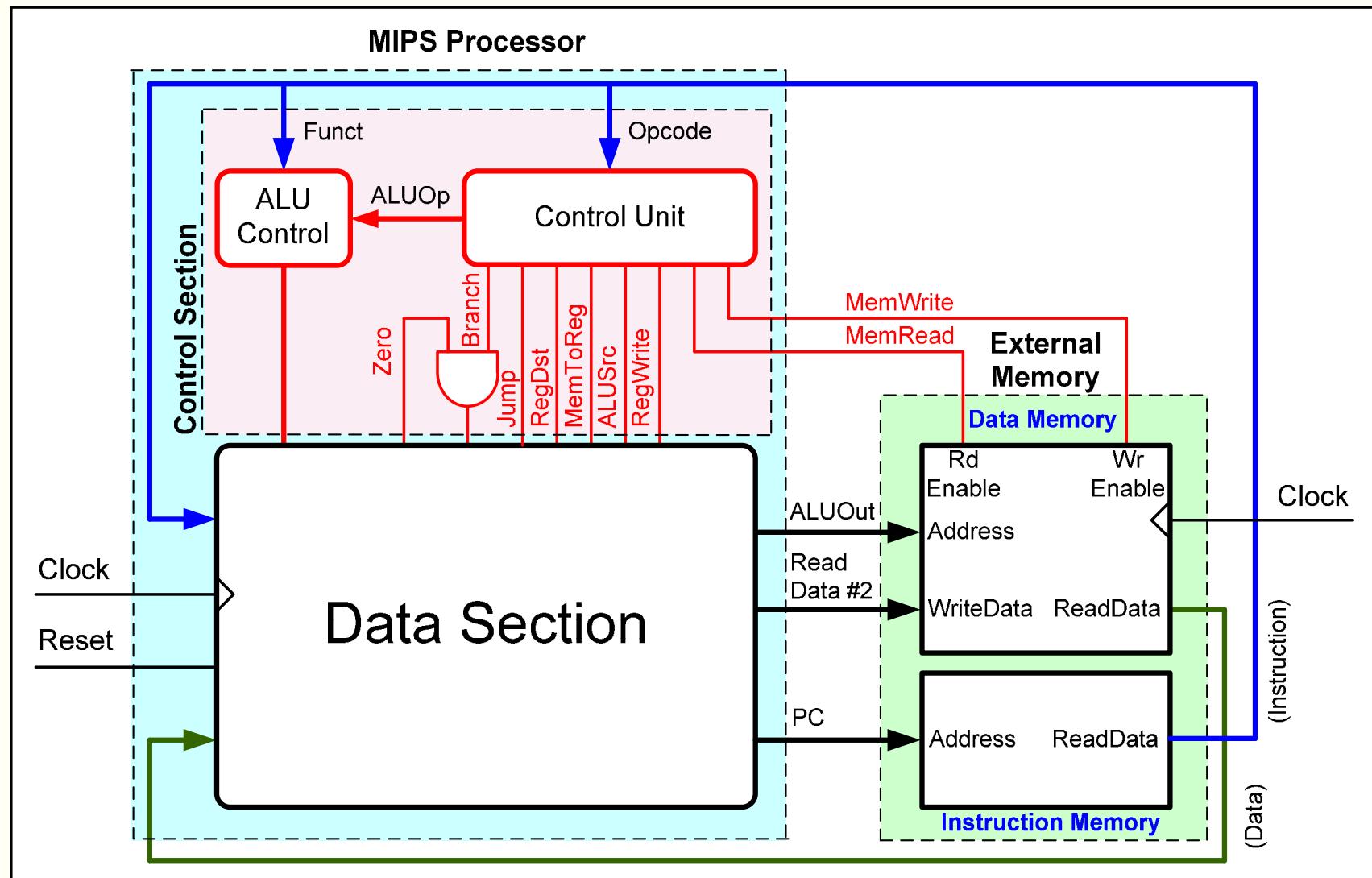


O novo valor do PC é obtido a partir dos 26 LSB da instrução multiplicados por 4 (shift left 2) concatenados com os 4 MSB do PC atual.



Aulas 16,17 - 32

# Visão global do processador



## Execução de uma instrução no *datapath single-cycle* – exemplo

- Vai iniciar-se o *instruction fetch* da instrução apontada pelo registo \$PC (0x00400024). Nesse instante o conteúdo dos registos do CPU e da memória de dados é o indicado. **Qual o conteúdo dos registos após a execução da instrução?**

| Endereço   | Valor      |
|------------|------------|
| (...)      | (...)      |
| 0x10010030 | 0x63F78395 |
| 0x10010034 | 0xA0FCF3F0 |
| 0x10010038 | 0x147FAF83 |
| (...)      | (...)      |



| Endereço   | Código máquina |
|------------|----------------|
| (...)      | (...)          |
| 0x00400020 | 0x00E82820     |
| 0x00400024 | 0x8CA30024     |
| 0x00400028 | 0x00681824     |
| (...)      | (...)          |

|      |            |
|------|------------|
| \$PC | 0x00400024 |
| \$3  | 0x7F421231 |
| \$4  | 0x15A73C49 |
| \$5  | 0x10010010 |

|      |            |
|------|------------|
| \$PC | 0x00400028 |
| \$3  | 0xA0FCF3F0 |
| \$4  | 0x15A73C49 |
| \$5  | 0x10010010 |

0x8CA30024 → lw \$3, 0x24(\$5)

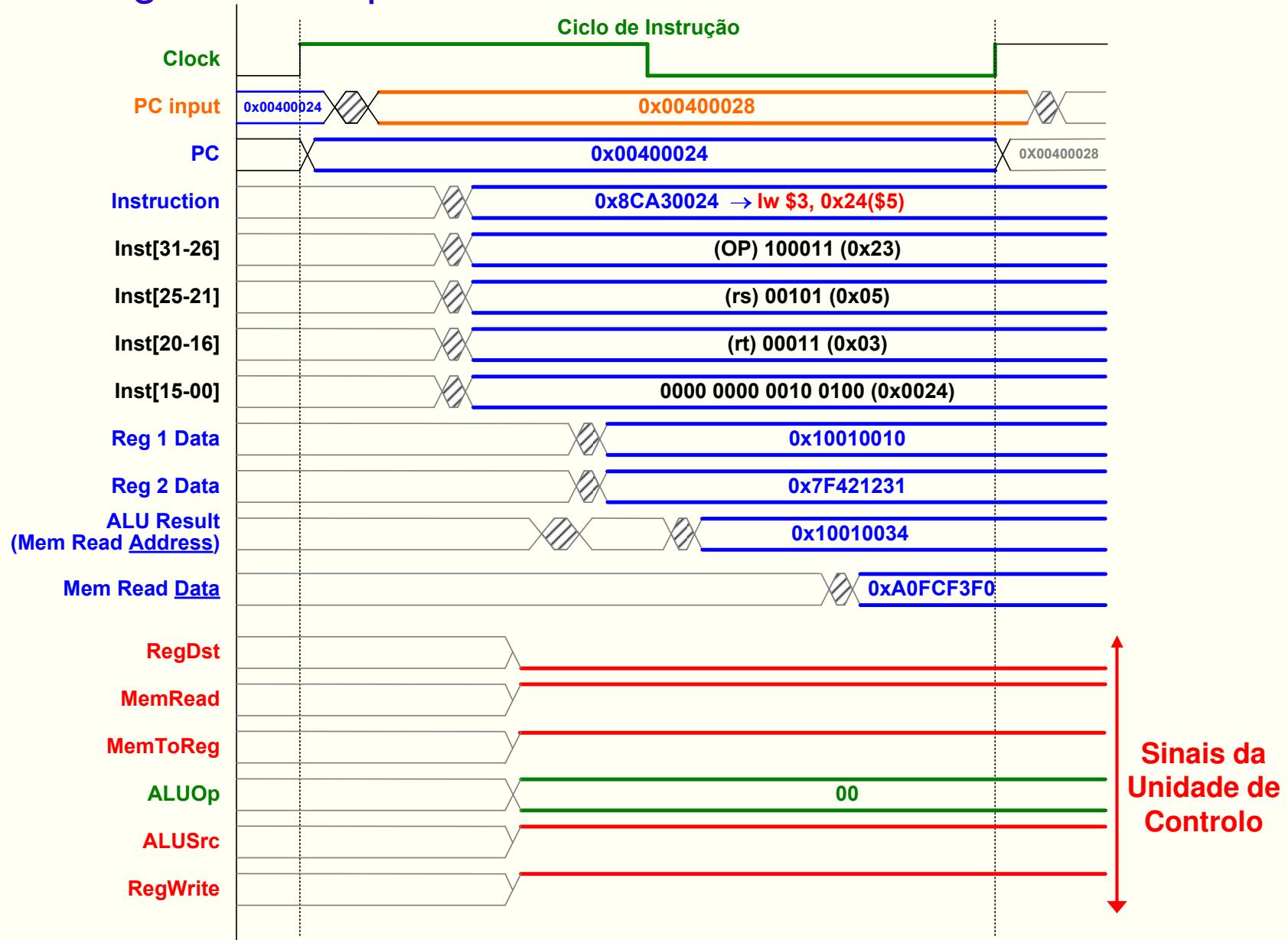
Mem Addr: 0x10010010 + 0x24 = 0x10010034

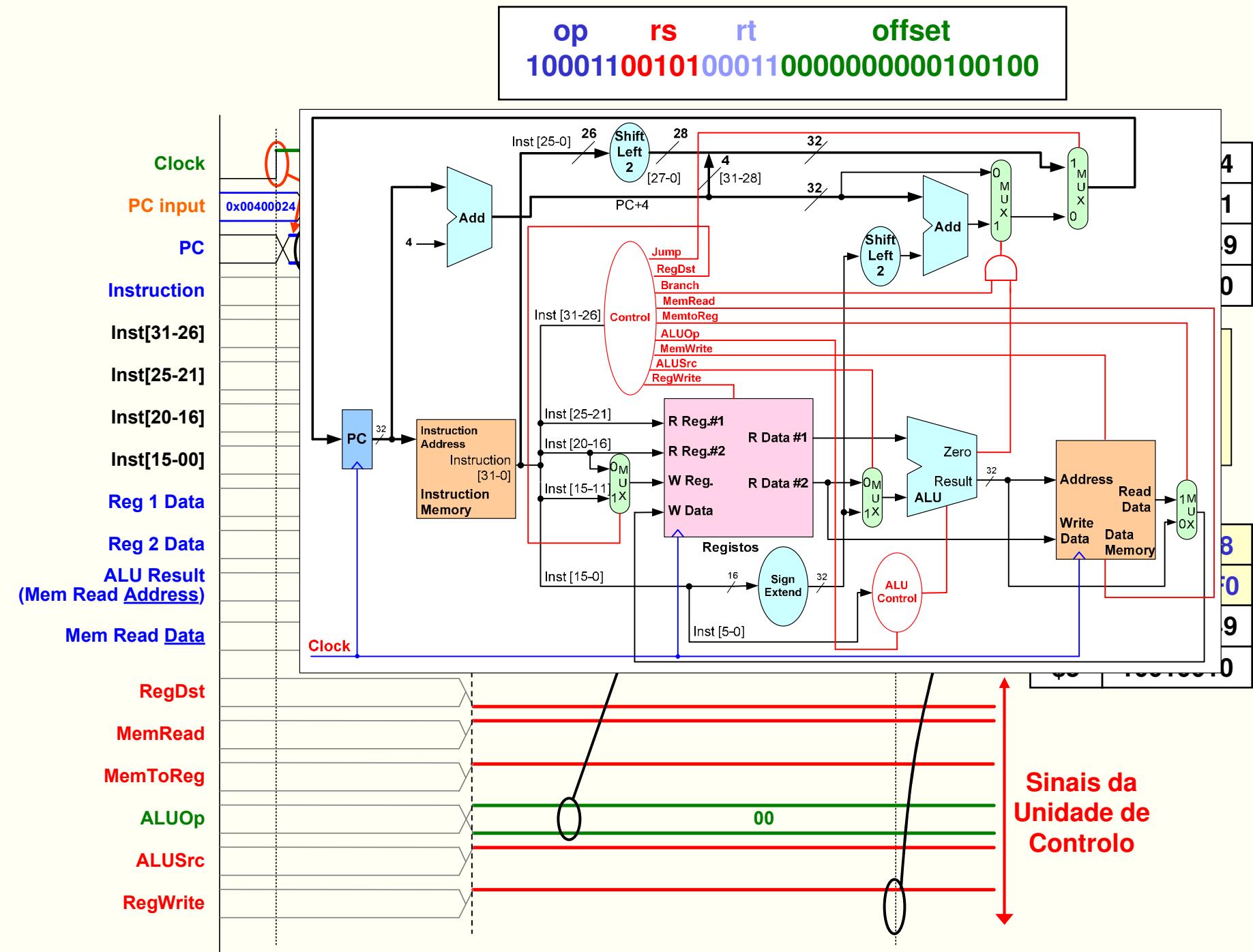
10001100101000110000000000100100

\$3 = [0x10010034] = 0xA0FCF3F0

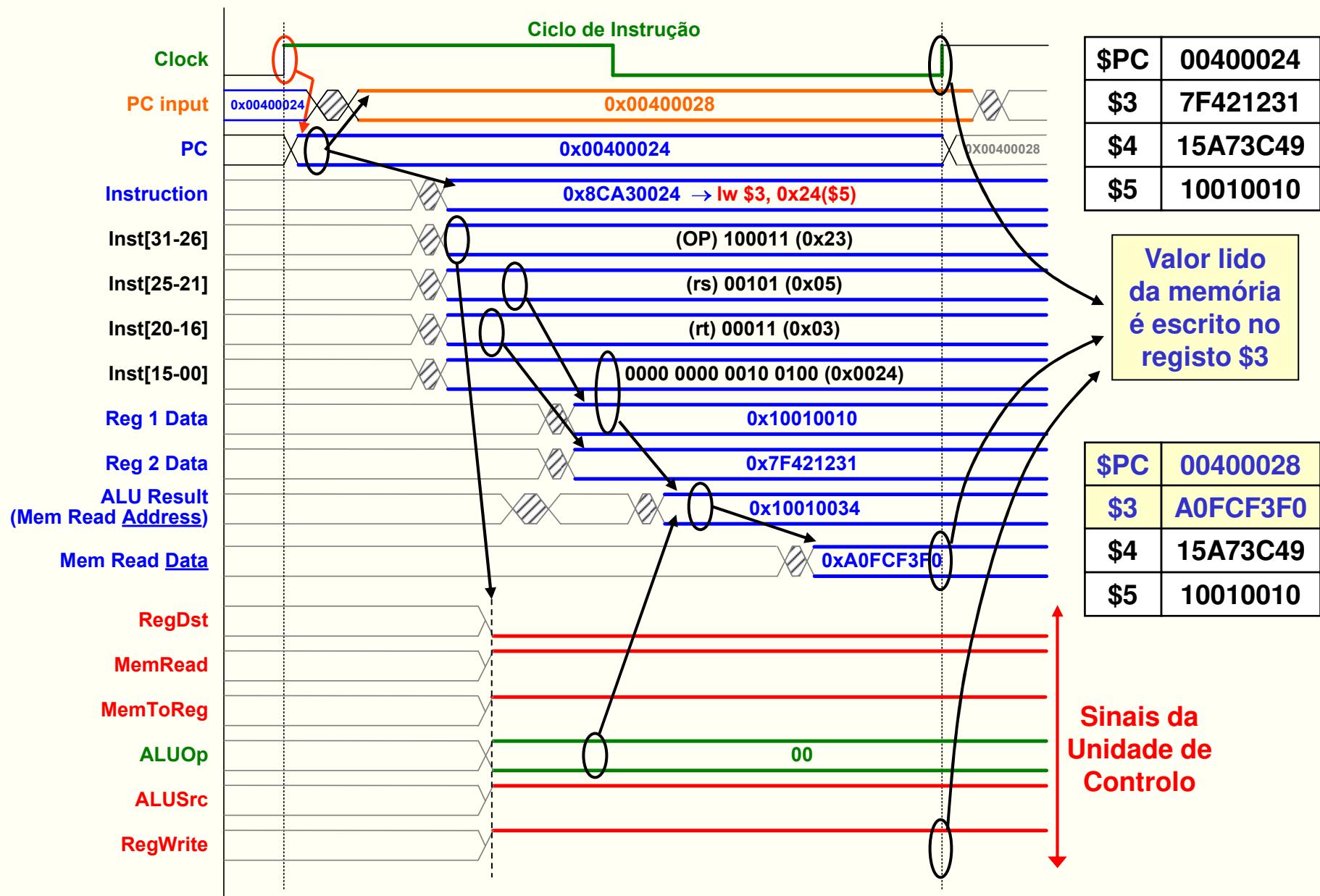


# Execução de uma instrução no *datapath single-cycle* – diagrama temporal





| op     | rs    | rt    | offset           |
|--------|-------|-------|------------------|
| 100011 | 00101 | 00011 | 0000000000100100 |



## Aulas 16 e 17

- Instanciação de vários módulos do datapath recorrendo à linguagem de programa hardware VHDL:
  - Unidade de controlo da ALU
  - Unidade de controlo principal
  - Atualização do PC (versão completa)

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Unidade de controlo da ALU

```
library ieee;
use ieee.std_logic_1164.all;

entity ALUControlUnit is
    port(ALUop      : in  std_logic_vector(1 downto 0);
          funct      : in  std_logic_vector(5 downto 0);
          ALUcontrol : out std_logic_vector(2 downto 0));
end ALUControlUnit;
```



# Unidade de controlo da ALU

```
architecture Behavioral of ALUControlUnit is
begin
    process(ALUop, funct)
    begin
        case ALUop is
            when "00" => -- LW, SW, ADDI
                ALUcontrol <= "010";
            when "01" => -- BEQ
                ALUcontrol <= "110";
            when "10" => -- R-Type instructions
                case funct is
                    when "100000" => ALUcontrol <= "010";      -- ADD
                    when "100010" => ALUcontrol <= "110";      -- SUB
                    when "100100" => ALUcontrol <= "000";      -- AND
                    when "100101" => ALUcontrol <= "001";      -- OR
                    when "101010" => ALUcontrol <= "111";      -- SLT
                    when others     => ALUcontrol <= "010";
                end case;
            when "11" => -- SLTI
                ALUcontrol <= "111";
        end case;
    end process;
end Behavioral;
```

| ALU Control | ALU Action       |
|-------------|------------------|
| 0 0 0       | And              |
| 0 0 1       | Or               |
| 0 1 0       | Add              |
| 1 1 0       | Subtract         |
| 1 1 1       | Set if Less Than |

| ALUOp | ALU Action       |
|-------|------------------|
| 00    | Add              |
| 01    | Subtract         |
| 10    | R-Type           |
| 11    | Set if Less Than |



# Unidade de controlo principal

```
library ieee;
use ieee.std_logic_1164.all;

entity ControlUnit is
    port(OpCode      : in std_logic_vector(5 downto 0);
          RegDst      : out std_logic;
          Branch      : out std_logic;
          MemRead     : out std_logic;
          MemWrite    : out std_logic;
          MemToReg   : out std_logic;
          ALUsrc     : out std_logic;
          RegWrite   : out std_logic;
          ALUop       : out std_logic_vector(1 downto 0));
end ControlUnit;
```



```

architecture Behavioral of ControlUnit is
begin
  process(OpCode)
  begin
    RegDst  <= '0'; Branch <= '0'; MemRead  <= '0'; MemWrite <= '0';
    MemToReg <= '0'; ALUsrc <= '0'; RegWrite <= '0';
    ALUop   <= "00";
    case OpCode is
      when "000000" =>    -- R-Type instructions
        ALUop     <= "10";
        RegDst    <= '1';
        RegWrite <= '1';
      when "000100" =>    -- BEQ
        ALUop     <= "01";
        Branch   <= '1';
      when "100011" =>    -- LW
        ALUsrc    <= '1';
        MemToReg <= '1';
        MemRead   <= '1';
        RegWrite <= '1';
      when "101011" =>    -- SW
        ALUsrc    <= '1';
        MemWrite <= '1';
      when "001000" =>    -- ADDI
        ALUsrc    <= '1';
        RegWrite <= '1';
      when "001010" =>    -- SLTI
        ALUop     <= "11";
        ALUsrc    <= '1';
        RegWrite <= '1';
      when others =>
    end case;
  end process;
end Behavioral;

```

# Módulo de atualização do PC, completo

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity PCupdate is
    port(clk      : in std_logic;
          reset    : in std_logic;
          branch   : in std_logic;
          jump     : in std_logic;
          zero     : in std_logic;
          offset   : in std_logic_vector(31 downto 0);
          jAddr    : in std_logic_vector(25 downto 0);
          pc       : out std_logic_vector(31 downto 0));
end PCupdate;
```



# Módulo de atualização do PC, completo

```
architecture Behavioral of PCupdate is
    signal s_pc, s_pc4, s_offset : unsigned(31 downto 0);
begin
    s_offset <= unsigned(offset(29 downto 0)) & "00"; -- Left shift
    s_pc4 <= s_pc + 4;
    process(clk)
    begin
        if(rising_edge(clk)) then
            if(reset = '1') then
                s_pc <= (others => '0');
            else
                if(jump = '1') then          -- Jump Target Address
                    s_pc <= s_pc4(31 downto 28) & unsigned(jAddr) & "00";
                elsif(branch = '1' and zero = '1') then
                    s_pc <= s_pc4 + s_offset;   -- Branch Target Address
                else
                    s_pc <= s_pc4;
                end if;
            end if;
        end if;
    end process;
    pc <= std_logic_vector(s_pc);
end Behavioral;
```



## Aulas 18, 19 e 20

- Limitações das arquiteturas *single-cycle*
- Versão de referência de uma arquitetura *multi-cycle*
- Exemplos de funcionamento numa arquitetura *multi-cycle*:
  - Instruções tipo R
  - Acesso à memória – LW
  - Salto condicional – BEQ
  - Salto incondicional – J
- Unidade de controlo para o *datapath multi-cycle*
  - Diagrama de estados da unidade de controlo
- Sinais de controlo e valores do *datapath multi-cycle*
  - Exemplo com execução sequencial de três instruções

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Tempo de execução das instruções (*single-cycle*)

- Como visto anteriormente, a **frequência máxima** do relógio de sincronização está limitada pelo **tempo de execução da instrução “mais longa”**
- Os **tempos de execução** das várias instruções suportadas pelo *datapath single-cycle* correspondem ao **somatório dos atrasos introduzidos por cada um dos elementos funcionais envolvidos na execução da instrução**
- Note-se que apenas os elementos funcionais que se encontram em **série** contribuem para aumentar o tempo necessário para concluir a execução da instrução (caminho crítico)



# Tempo de execução das instruções

- Consideraremos os seguintes tempos de atraso introduzidos por cada um dos elementos funcionais do *datapath single-cycle*:
  - Acesso à memória para leitura -  $t_{RM}$
  - Acesso à memória para preparar a escrita -  $t_{WM}$
  - Acesso ao *register file* para leitura -  $t_{RRF}$
  - Acesso ao *register file* para preparar a escrita -  $t_{WRF}$
  - Operação da ALU -  $t_{ALU}$
  - Operação de um somador -  $t_{ADD}$
  - Unidade de controlo -  $t_{CNTL}$
  - Extensor de sinal -  $t_{SE}$
  - Shift Left 2 -  $t_{SL2}$
  - Tempo de *setup* do PC -  $t_{stPC}$



# Tempo de execução das instruções

- Considerando os tempos de atraso anteriores, os tempos de execução das várias instruções suportadas pelo datapath single cycle serão:

## Instruções tipo R:

- $t_{EXEC} = t_{RM} + \max(t_{RRF}, t_{CNTL}) + t_{ALU} + t_{WRF}$

## Instrução SW:

- $t_{EXEC} = t_{RM} + \max(t_{RRF}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{WM}$

## Instrução LW:

- $t_{EXEC} = t_{RM} + \max(t_{RRF}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{RM} + t_{WRF}$

## Instrução BEQ:

- $t_{EXEC} = t_{RM} + \max(\underbrace{\max(t_{RRF}, t_{CNTL}) + t_{ALU}}_{comparação}, \underbrace{t_{SE} + t_{SL2} + t_{ADD}}_{cálculo do BTA}) + t_{stPC}$

## Instrução J:

- $t_{EXEC} = t_{RM} + \max(t_{CNTL}, t_{SL2}) + t_{stPC}$

## Notas:

- Considera-se que o tempo de cálculo de PC+4 é muito inferior ao somatório dos restantes tempos envolvidos na execução da instrução
- O tempo  $t_{CNTL}$  inclui o tempo de atraso da unidade de controlo da ALU
- Desprezam-se os tempos de atraso introduzidos pelos *multiplexers*
- Só se considera o  $t_{stPC}$  nas instruções de controlo de fluxo.



# Tempo de execução das instruções - exemplo

- Considerem-se os seguintes valores hipotéticos para os tempos de atraso introduzidos por cada um dos elementos funcionais do *datapath single-cycle*:
  - Acesso à memória para leitura ( $t_{RM}$ ): 5ns
  - Acesso à memória para preparar escrita ( $t_{WM}$ ): 5ns
  - Acesso ao *register file* para leitura ( $t_{RRF}$ ): 3ns
  - Acesso ao *register file* para preparar escrita ( $t_{WRF}$ ): 3ns
  - Operação da ALU ( $t_{ALU}$ ): 4ns
  - Operação de um somador ( $t_{ADD}$ ): 1ns
  - *Multiplexers* e restantes elementos funcionais: 0ns
  - Unidade de controlo ( $t_{CNTL}$ ): 1ns
  - Tempo de *setup* do PC ( $t_{stPC}$ ): 1ns



# Tempo de execução das instruções - exemplo

- Instruções tipo R:

- $t_{EXEC} = t_{RM} + \max(t_{RFR}, t_{CNTL}) + t_{ALU} + t_{WFR}$   
 $= 5 + \max(3, 1) + 4 + 3 = 15 \text{ ns}$

- Instrução SW:

- $t_{EXEC} = t_{RM} + \max(t_{RFR}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{WM}$   
 $= 5 + \max(3, 1, 0) + 4 + 5 = 17 \text{ ns}$

- Instrução LW:

- $t_{EXEC} = t_{RM} + \max(t_{RFR}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{RM} + t_{WFR}$   
 $= 5 + \max(3, 1, 0) + 4 + 5 + 3 = 20 \text{ ns}$

- Instrução BEQ:

- $t_{EXEC} = t_{RM} + \max(\max(t_{RFR}, t_{CNTL}) + t_{ALU}, t_{SE} + t_{SL2} + t_{ADD}) + t_{stPC}$   
 $= 5 + \max(\max(3, 1) + 4, 0 + 0 + 1) + 1 = 13 \text{ ns}$

- Instrução J:

- $t_{EXEC} = t_{RM} + \max(t_{CNTL}, t_{SL2}) + t_{stPC} = 5 + \max(1, 0) + 1 = 7 \text{ ns}$



# Limitações das soluções *single-cycle*

- Face à análise anterior, a máxima frequência de trabalho seria:  
**fmax = 1 / 20ns = 50MHz**
- Com a mesma tecnologia, contudo, uma multiplicação ou divisão poderia demorar um tempo da ordem dos 150ns
- Para poder suportar uma ALU com capacidade para efetuar operações de multiplicação/divisão, a frequência de relógio máxima do nosso *datapath* baixaria para 6.66Mhz
- Esta frequência máxima limitaria a eficiência de todas as outras instruções, mesmo que as instruções de multiplicação ou divisão sejam raramente utilizadas
- Uma solução possível, mas tecnicamente complicada, seria usar um relógio de frequência variável, ajustável em função da instrução que vai ser executada



# Limitações das soluções *single-cycle*

- Genericamente, o tempo de execução de um programa pode ser calculado como:

$$T_{exec_{CPU}} = \# \text{ Instruções} \times CPI \times Clock\_Cycle_{CPU}$$

sendo CPI o número médio de ciclos de relógio por instrução na execução do programa em causa; no caso da implementação *single-cycle* o CPI é 1, logo:

$$T_{exec_{CPU}} = \# \text{ Instruções} \times Clock\_Cycle_{CPU}$$

- O desempenho relativo de um CPU ( $CPU_{ANALISE}$ ) relativamente a outro ( $CPU_{REFERENCIA}$ ) pode ser expresso por:

$$\frac{Desempenho_{CPU\_ANALISE}}{Desempenho_{CPU\_REFERENCIA}} = \frac{T_{exec_{CPU\_REFERENCIA}}}{T_{exec_{CPU\_ANALISE}}}$$



# Limitações das soluções *single-cycle*

- Calcular o ganho de desempenho que se obteria com uma implementação de *clock* variável relativamente a uma com o *clock* fixo, na execução de um programa com o seguinte mix de instruções:
  - 20% de lw, 10% de sw, 50% de tipo R, 15% de branches e 5% de jumps
  - assumindo os tempos execução determinados anteriormente para os vários tipos de instruções (LW: 20ns, SW: 17ns, R-Type: 15ns, BEQ: 13ns, J: 7ns)
- Para este exemplo, o tempo médio de execução de cada instrução num CPU com *clock* variável é calculado como:  
$$T_{INSTR} = 0,2*20 + 0,1*17 + 0,5*15 + 0,15*13 + 0,05*7$$
- O ganho de desempenho do CPU com *clock* variável relativamente a um com *clock* fixo seria então:

$$\frac{Des_{CPU\_CLOCK\_VARIAVEL}}{Des_{CPU\_CLOCK\_FIXO}} = \frac{\# Instruções \times 20}{\# Instruções \times (0,2 \times 20 + 0,1 \times 17 + 0,5 \times 15 + 0,15 \times 13 + 0,05 \times 7)} = 1,29$$

A implementação com *clock* variável não é viável mas permite-nos entender o que está a ser sacrificado quando todas as instruções têm que ser executadas num único ciclo de relógio com dimensão fixa



## Limitações das soluções *single-cycle* - conclusões

- Num *datapath* que suporte instruções com complexidade variável, é a **instrução mais lenta que determina a máxima frequência de trabalho**, mesmo que seja um instrução pouco frequente
- Uma vez que o ciclo de relógio é igual ao maior tempo de atraso de todas as instruções, não é útil usar técnicas que reduzam o atraso do caso mais comum mas que não melhorem o maior tempo de atraso (isto é, o atraso do caminho crítico)
  - Isto contraria um dos princípios-chave de desenho: *make the common case fast* (o que é mais comum deve ser mais rápido)
- Elementos funcionais que estejam envolvidos na execução de uma mesma instrução **não podem ser usados para mais do que uma operação por ciclo de relógio** (ex: memória de instruções e de dados, ALU e somadores, ...)



# Alternativa às soluções *single-cycle*

- Em vez de desenvolver uma estratégia baseada num relógio de frequência variável, é preferível abdicar do princípio de que todas as instruções devem ser executadas num único ciclo de relógio
- Em alternativa, as várias instruções que compõem o *set* de instruções podem ser executadas em vários ciclos de relógio (*multi-cycle*):
  - A execução da instrução é decomposta num conjunto de operações
  - Cada uma dessas operações faz uso de um elemento funcional fundamental: memória, *register file* ou ALU
  - **Em cada ciclo de relógio poderá ser realizada uma ou mais operações, desde que sejam independentes** (por exemplo, *instruction fetch* e cálculo de PC+4 ou *operand fetch* e cálculo do BTA)
- Desta forma, o período de relógio fica apenas limitado pelo maior dos tempos de atraso de cada um dos elementos funcionais fundamentais
- Para os tempos de atraso que considerámos anteriormente, a máxima frequência de relógio seria assim:  **$f_{max} = 1 / t_{RM} = 1 / 5\text{ns} = 200\text{MHz}$**



# Alternativa às soluções *single-cycle*

- Uma outra vantagem duma solução de execução em vários ciclos de relógio (*multi-cycle*) é que um **mesmo elemento funcional** pode ser utilizado mais do que uma vez, no contexto da execução duma mesma instrução, desde que em ciclos de relógio distintos:
  - A memória externa poderá ser partilhada por instruções e dados
  - A mesma ALU poderá ser usada, para além das operações que já realizava na implementação *single-cycle*, para:
    - Calcular o valor de PC+4
    - Calcular o endereço alvo das instruções de salto condicional (BTA)
- A versão ***multi-cycle*** passará assim a ter:
  - **Uma única memória** para programa e dados (arquitetura Von Neumann)
  - **Uma única ALU**, em vez de uma ALU e dois somadores



# O datapath Multi-cycle

- A arquitetura *multi-cycle* do MIPS que vamos analisar adota um ciclo de instrução composto por um **máximo de cinco passos distintos**, cada um deles executado em 1 ciclo de relógio
- A distribuição das operações por estes 5 passos tenta distribuir equitativamente o trabalho a realizar em cada ciclo
- Na definição destes passos pressupõe-se que durante um ciclo de relógio apenas é possível efetuar uma das seguintes operações fundamentais:
  - 1 acesso à memória externa (**uma escrita ou uma leitura**)
  - 1 acesso ao *Register File* (**uma escrita ou uma leitura**)
  - 1 operação na ALU
- No **mesmo ciclo de relógio**, podem ser realizadas operações em elementos operativos distintos, desde que sejam independentes
  - Exemplos: **um acesso à memória externa e uma operação na ALU**, ou **um acesso ao *Register File* e uma operação na ALU**



# O datapath Multi-cycle – fases de execução

## Fase 1 (memória, ALU):

- *Instruction fetch* e cálculo de PC+4

## Fase 2 (unidade de controlo, register file, ALU):

- *Instruction decode*, *operand fetch* e cálculo do *branch target address*

## Fase 3 (ALU):

- Execução da operação na ALU (instruções tipo R / addi / slti), **ou**
- Cálculo do endereço de memória (instr. de acesso à memória), **ou**
- Comparação dos operandos - instrução *branch* (conclusão da instrução)

## Fase 4 (memória, register file):

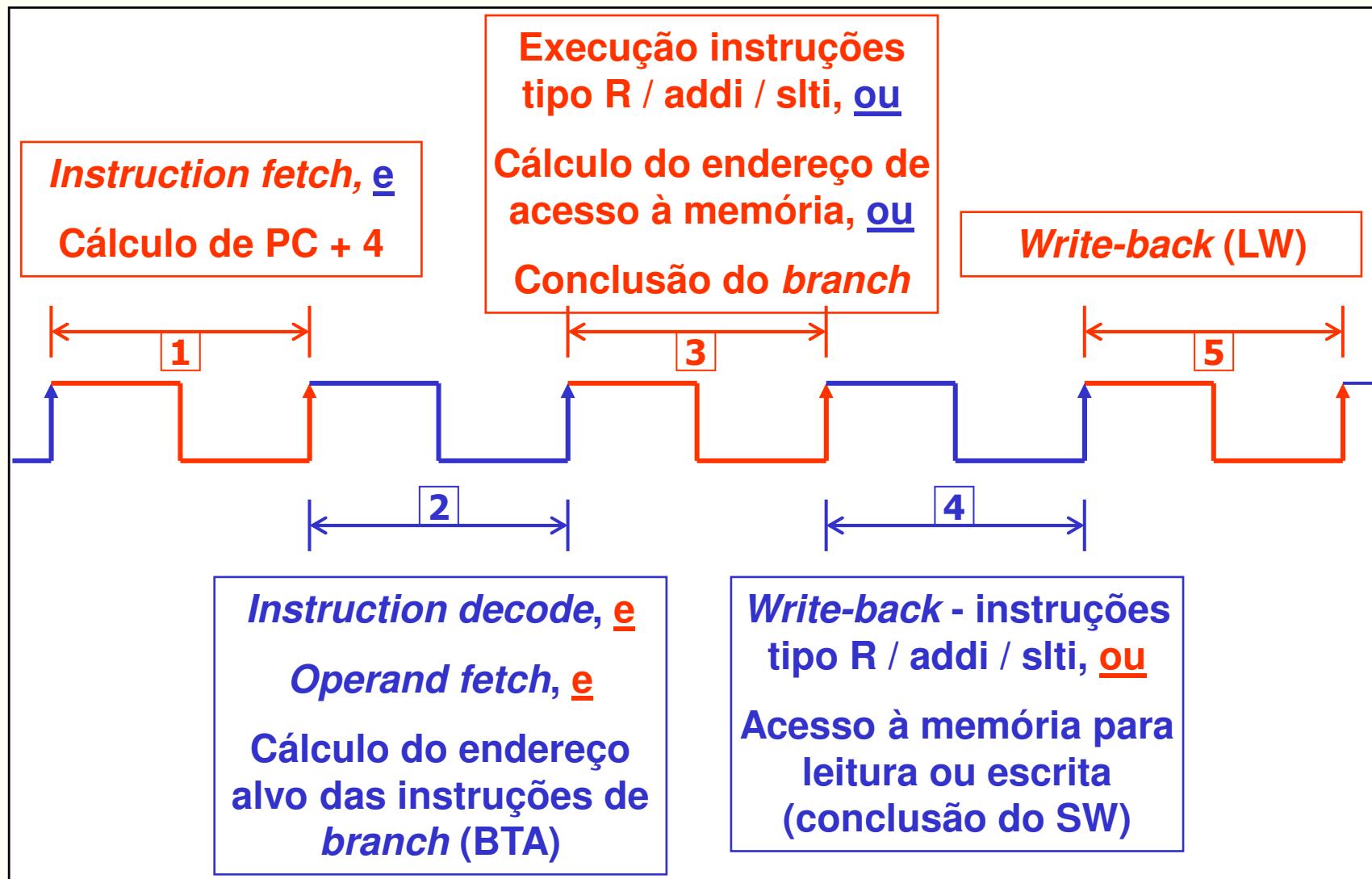
- Acesso à memória para leitura (instrução LW), **ou**
- Acesso à memória para escrita (conclusão da instrução SW), **ou**
- Escrita no *Register File* (conclusão das instruções tipo R / addi / slti: **write-back**)

## Fase 5 (register file):

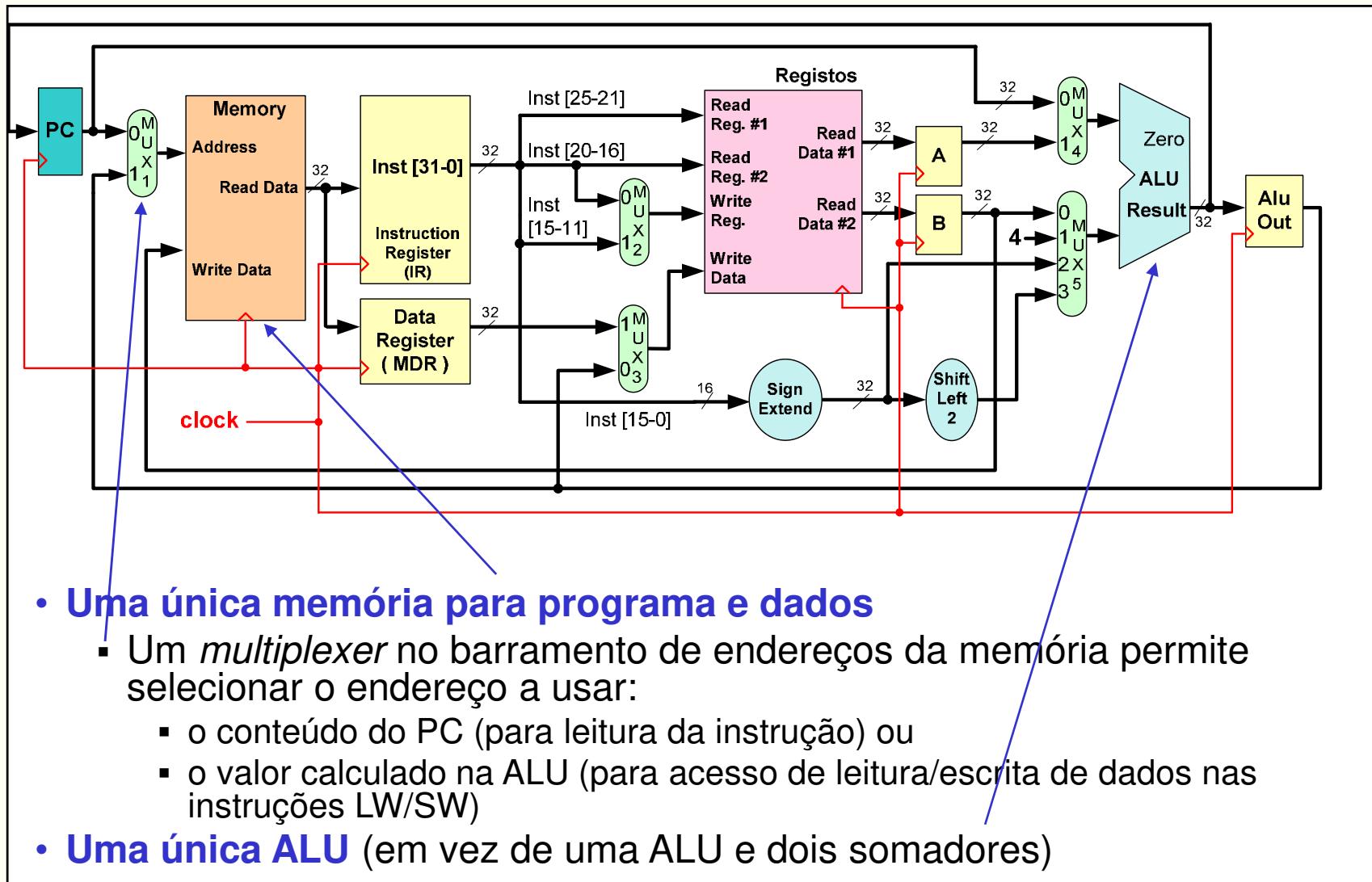
- Escrita no *Register File* (conclusão da instrução LW: **write-back**)



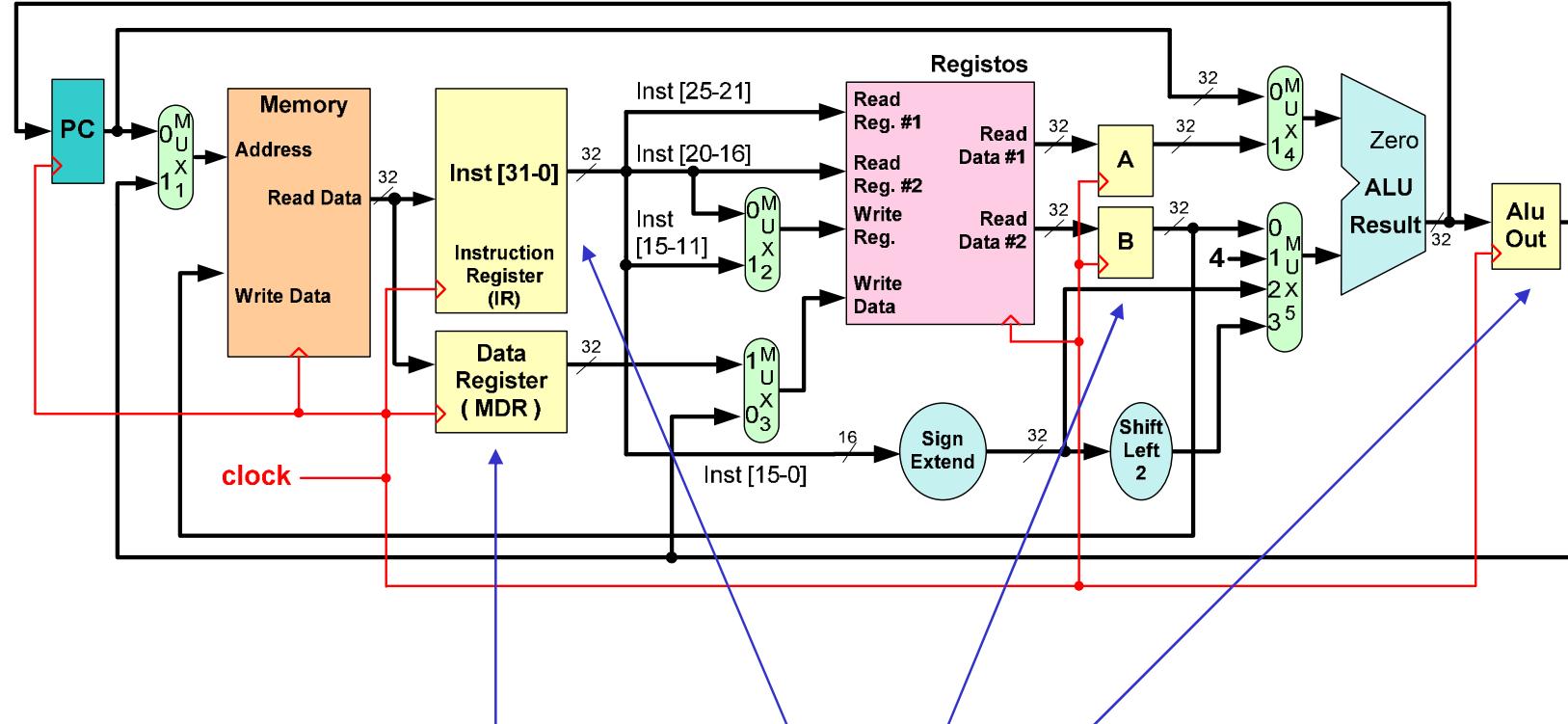
# O datapath Multi-cycle



# O datapath Multi-cycle (sem BEQ e J)

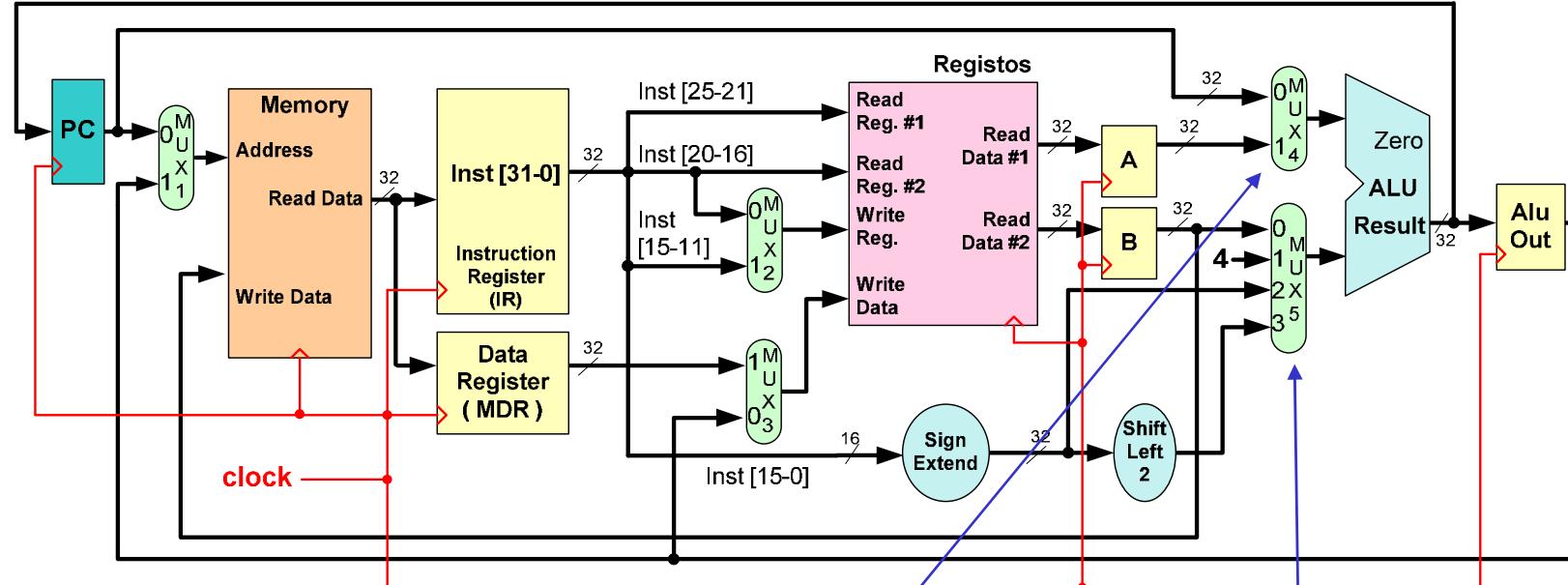


## O datapath Multi-cycle (sem BEQ e J)



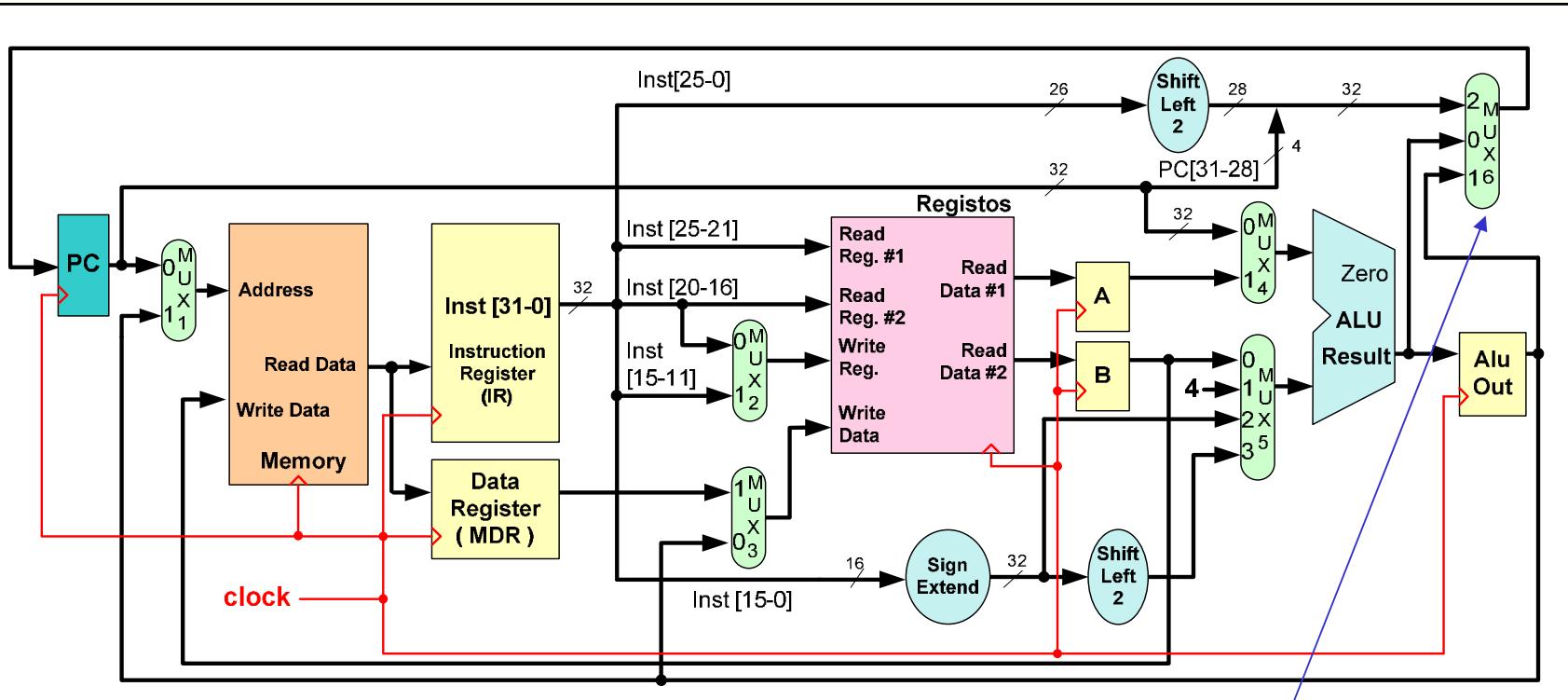
- Registos adicionados à saída dos elementos funcionais fundamentais para armazenamento de informação obtida/calculada durante o ciclo de relógio corrente e que será utilizada no ciclo de relógio seguinte

# O datapath Multi-cycle (sem BEQ e J)



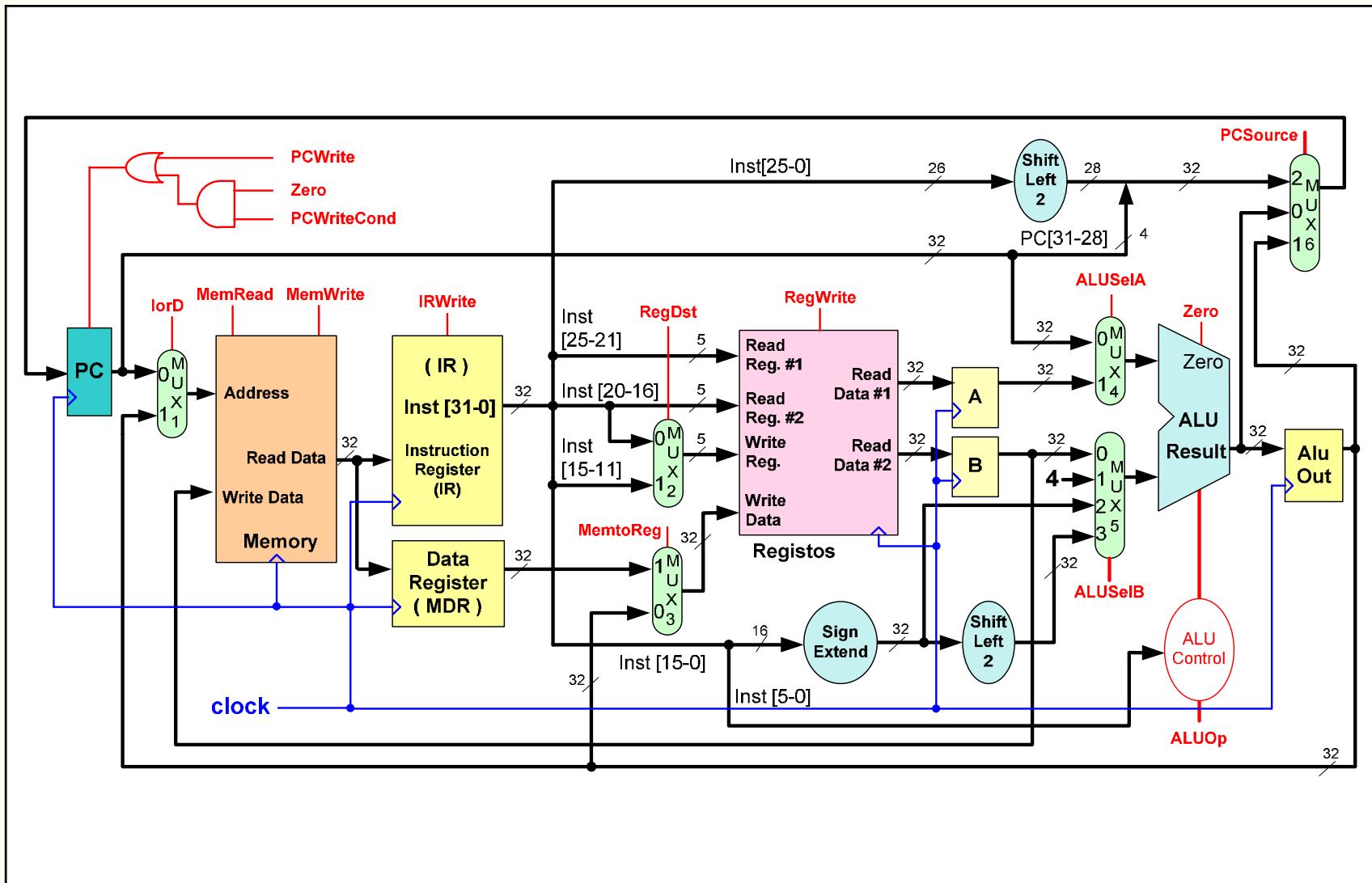
- A utilização de uma única ALU obriga às seguintes alterações nas suas entradas:
  - Um *multiplexer* adicional na primeira entrada, que escolhe entre a saída do registo A e a saída do PC
  - O *multiplexer* da segunda entrada é aumentado para poder suportar o incremento do PC (constante 4) e o cálculo do endereço alvo das instruções de branch (BTA - *branch target address*)

O datapath Multi-cycle com as instruções de salto



- Com as instruções de salto, o **registro PC** pode ser atualizado com um dos valores:
    - A **saída da ALU** que contém o PC+4 calculado durante o *instruction fetch* (na 1<sup>a</sup> fase)
    - A saída do registo **ALUOut** que armazena o endereço alvo das instruções de *branch* (BTA) calculado na ALU (na 2<sup>a</sup> fase)
    - **Jump Target Address** - 26 LSB da instrução multiplicados por 4 (*shift left 2*) concatenados com os 4 MSB do PC atual (o PC foi já incrementado na 1<sup>a</sup> fase)

# O datapath Multi-cycle, com os sinais de controlo



# O datapath Multi-cycle – sinais de controlo

| Sinal              | Efeito quando não activo ('0')                                    | Efeito quando activo ('1')  |
|--------------------|---|---|
| <b>MemRead</b>     | Nenhum  | O conteúdo da memória no endereço indicado é apresentado à saída  |
| <b>MemWrite</b>    | Nenhum  | O conteúdo do registo de memória, cujo endereço é fornecido, é substituído pelo valor apresentado à entrada |
| <b>ALUSelA</b>     | O primeiro operando da ALU é o PC                                 | O primeiro operando da ALU provém do registo indicado no campo rs   |
| <b>RegDst</b>      | O endereço do registo destino provém do campo rt                  | O endereço do registo destino provém do campo rd  |
| <b>RegWrite</b>    | Nenhum  | O registo indicado no endereço de escrita é alterado pelo valor presente na entrada de dados                |
| <b>MemtoReg</b>    | O valor apresentado para escrita no registo destino provém da ALU | O valor apresentado na entrada de dados do Register File provém do Data Register                            |
| <b>IorD</b>        | O PC é usado para fornecer o endereço à memória externa           | A saída do registo AluOut é usada para providenciar um endereço para a memória externa                      |
| <b>IRWrite</b>     | Nenhum  | O valor lido da memória externa é escrito no Instruction Register   |
| <b>PCWrite</b>     | Nenhum  | O PC é actualizado <b>incondicionalmente</b> na próxima transição activa do sinal de relógio                |
| <b>PCWriteCond</b> | Nenhum  | O PC é actualizado <b>condicionalmente</b> na próxima transição activa do relógio                           |

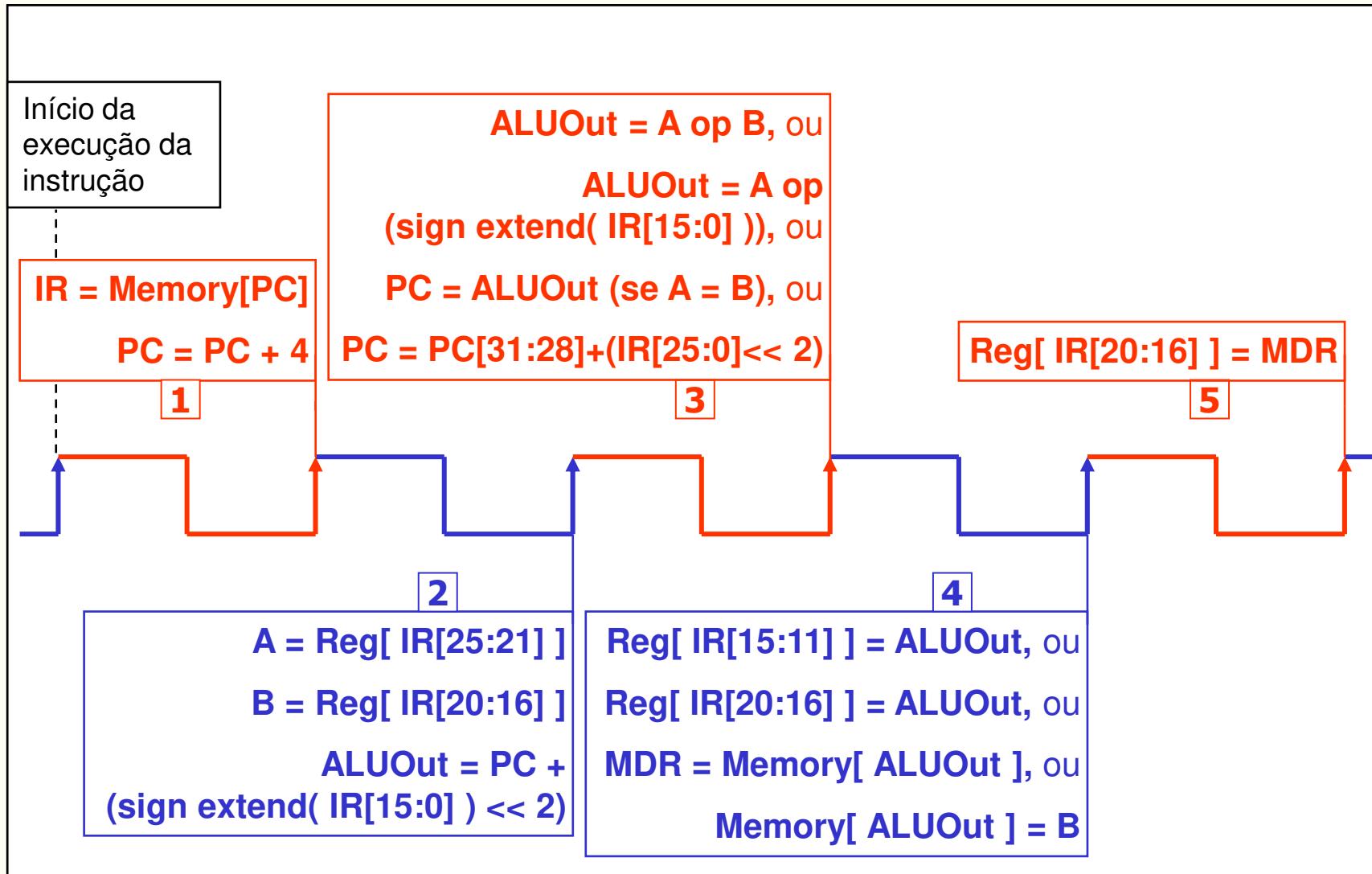


# O datapath Multi-cycle – sinais de controlo

| Sinal    | Valor | Efeito   |
|----------|-------|--|
| ALUSelB  | 00    | A segunda entrada da ALU provém do registo indicado pelo campo rt  |
|          | 01    | A segunda entrada da ALU é a constante 4   |
|          | 10    | A segunda entrada da ALU é a versão de sinal extendido dos 16 bits menos significativos do IR (instruction register)                           |
|          | 11    | A segunda entrada da ALU é a versão de sinal extendido e deslocada de dois bits, dos 16 bits menos significativos do IR (instruction register) |
| ALUOp    | 00    | ALU efetua uma adição  |
|          | 01    | ALU efetua uma subtração   |
|          | 10    | O campo "function code" da instrução determina qual a operação da ALU  |
|          | 11    | ALU efetua um SLT  |
| PCSource | 00    | O valor do PC é atualizado com o resultado da ALU (IF)   |
|          | 01    | O valor do PC é atualizado com o resultado da AluOut (Branch)  |
|          | 10    | O valor do PC é atualizado com o valor target do Jump  |
|          | 11    | Não usado  |



# Ações realizadas nas transições ativas do relógio ( $0 \rightarrow 1$ )

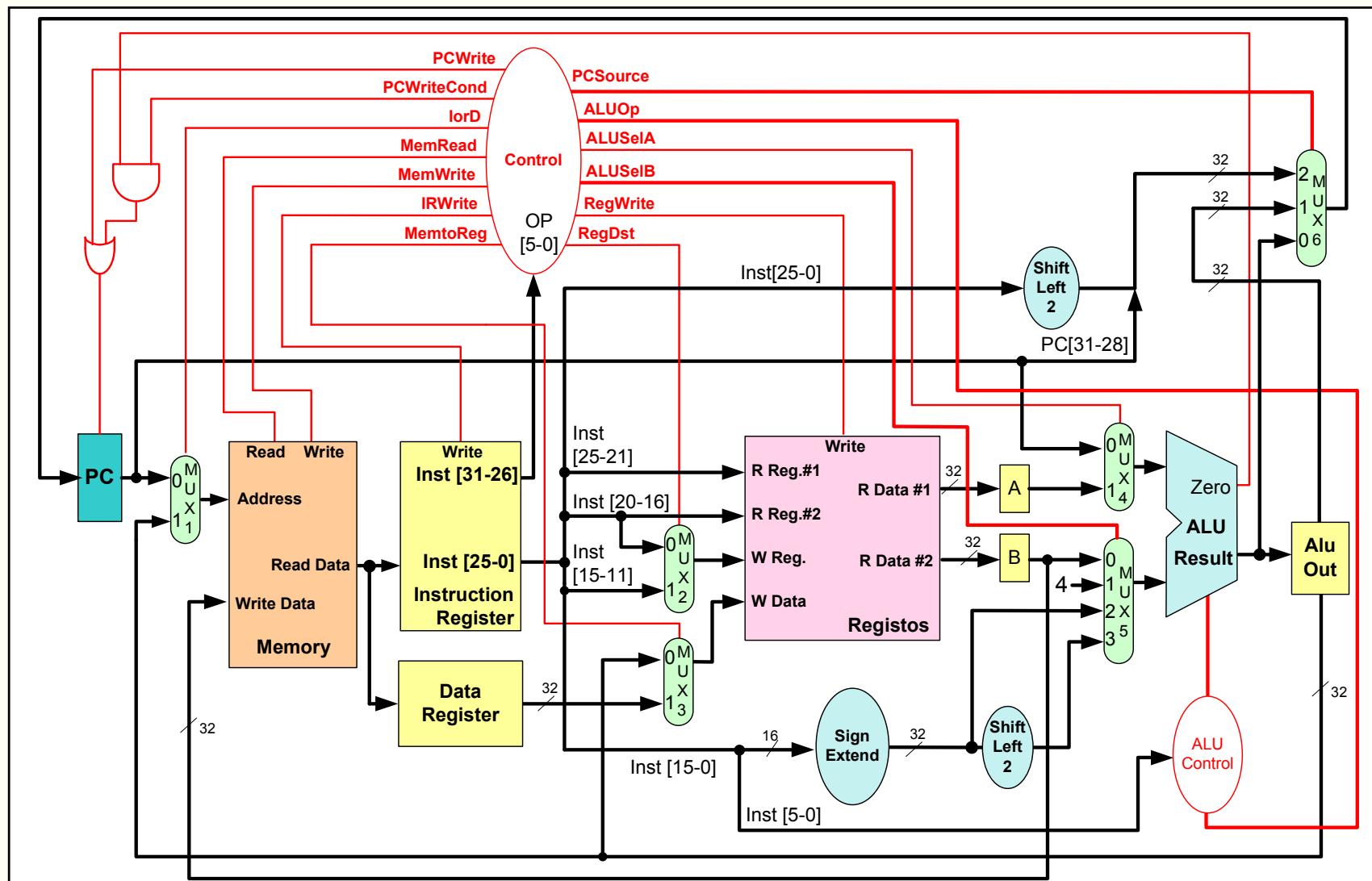


# Ações realizadas nas transições ativas do relógio ( $0 \rightarrow 1$ )

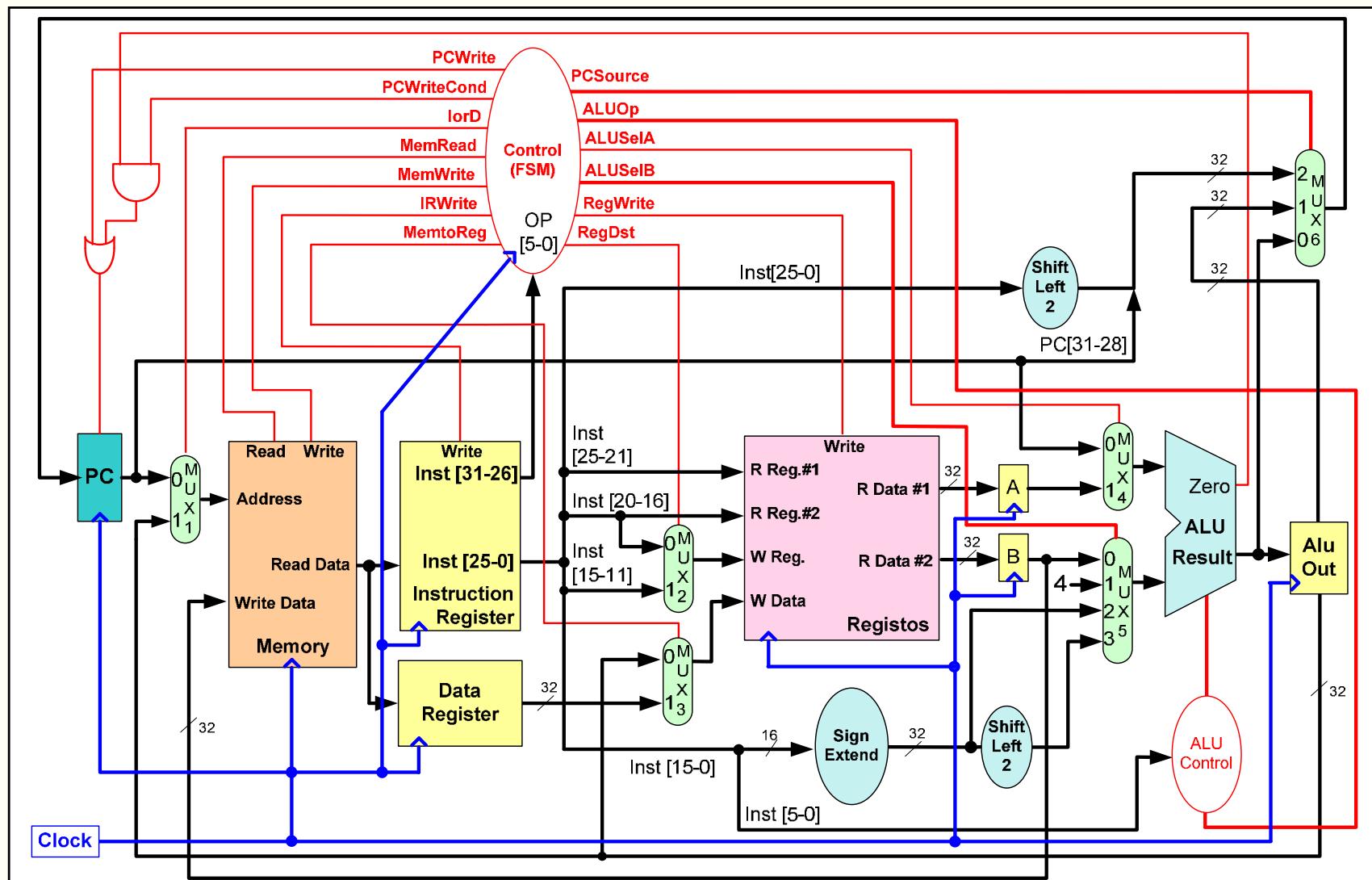
| Passo   | Ação p/ as R-Type / ADDI / SLTI  | Ação p/ instruções que referenciam a memória   | Ação p/ os branches                 |
|---|--|--|-------------------------------------|
| Instruction fetch   |  | $IR = Memory[PC]$<br>$PC = PC + 4$   |                                     |
| Instruction decode,<br>register fetch, cálculo<br>do BTA  |  | $A = Reg[ IR[25:21] ]$<br>$B = Reg[ IR[20:16] ]$<br>$ALUOut = PC + (\text{sign extended}( IR[15:0] ) \ll 2)$ |                                     |
| Execução<br>(tipoR/addi/slti), cálculo<br>de endereços ou<br>conclusão dos branches                                   | $ALUOut = A \text{ op } B \text{ ou}$<br>$ALUOut = A \text{ op }$<br>$\text{extend}(IR[15:0])$ | $ALUOut = A + \text{sign-extended}( IR[15:0] )$  | If $(A == B)$ then<br>$PC = ALUOut$ |
| Acesso à memória<br>(leitura-LW; ou escrita-SW) ou escrita no File Register (write-back, instruções tipo R/addi/slti) | Tipo R:<br>$Reg[ IR[15:11] ] = ALUOut$<br><br>ADDI / SLTI:<br>$Reg[ IR[20:16] ] = ALUOut$      | $MDR = Memory[ALUOut]$ ou<br>$Memory[ALUOut] = B$  |                                     |
| Escrita no File Register<br>(write-back, instrução LW)  |  | $Reg[ IR[20:16] ] = MDR$   |                                     |



# O datapath Multi-cycle completo



# O datapath Multi-cycle completo



# Exemplos de funcionamento

- Nos exemplos seguintes as cores indicam o estado, o valor ou a utilização dos sinais de controlo, barramentos e elementos de estado/combinatórios. O significado atribuído a cada cor é:
- Sinais de controlo:
  - **vermelho** → 0
  - **verde** → diferente de zero
  - **cinzento** → “don’t care”
- Barramentos:
  - **azul** → Relevantes no contexto do ciclo da instrução
  - **preto** → Não relevantes no contexto do ciclo da instrução
- Elementos de estado / combinatórios:
  - fundo branco → Não usados no contexto do ciclo da instrução
  - fundo de cor → Usados no contexto do ciclo da instrução
- Elementos de estado:
  - fundo de cor com textura → Escritos no final do ciclo de relógio corrente

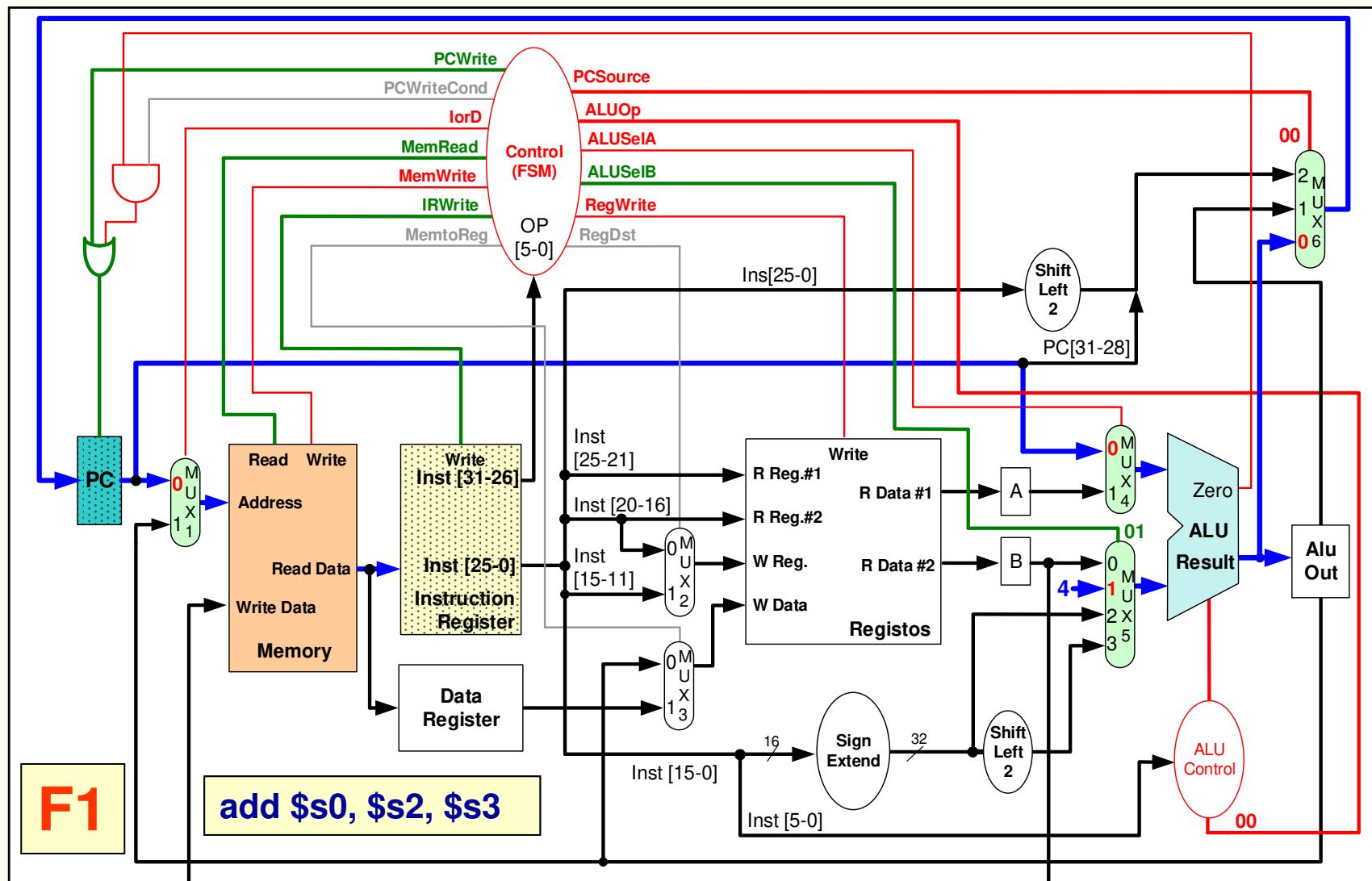
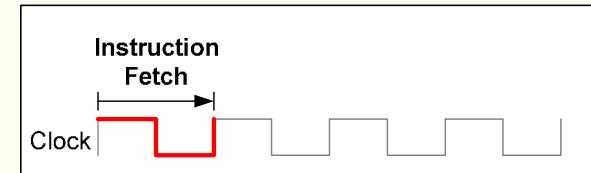


# Funcionamento do *datapath* nas instruções do tipo R

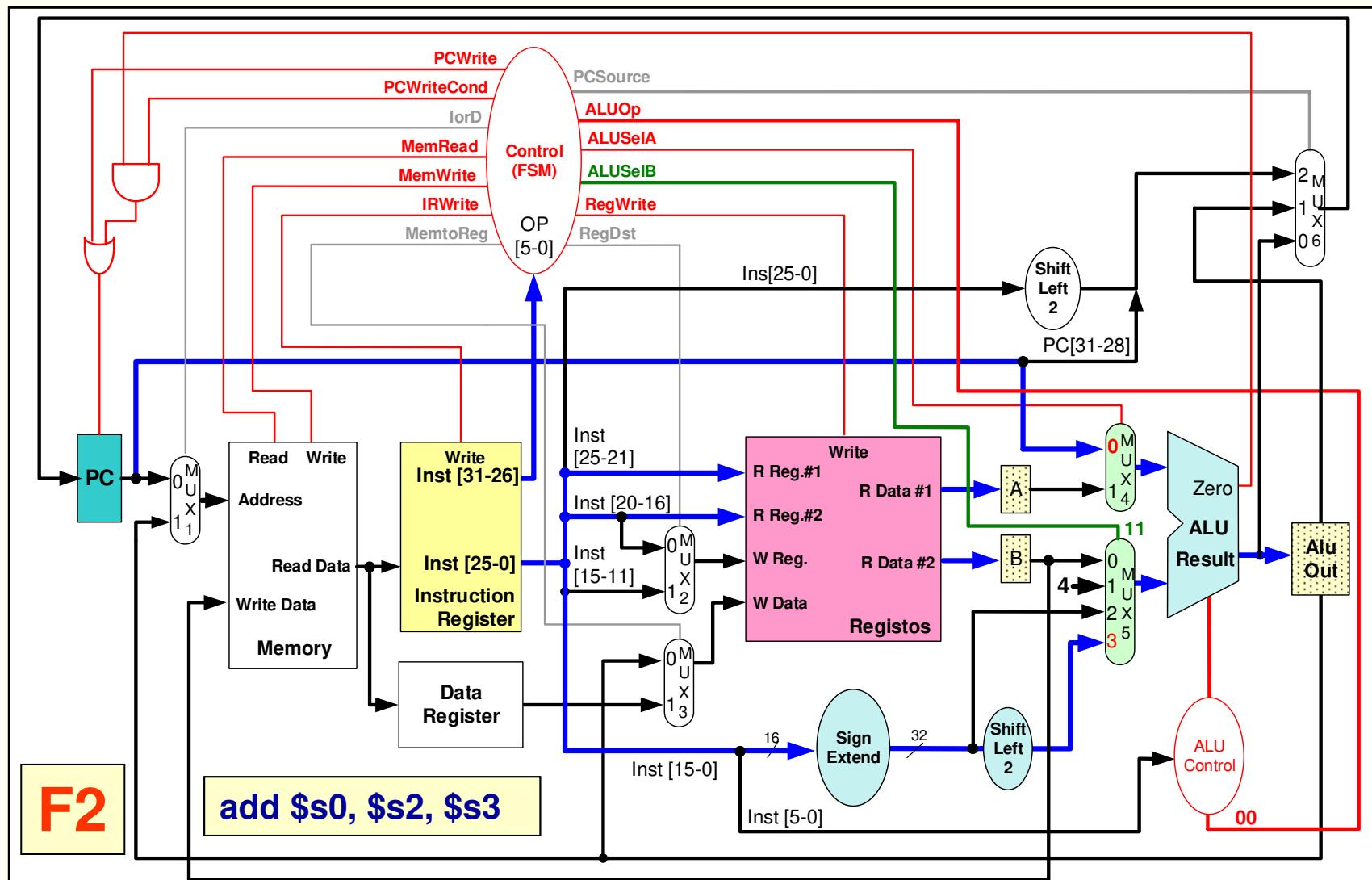
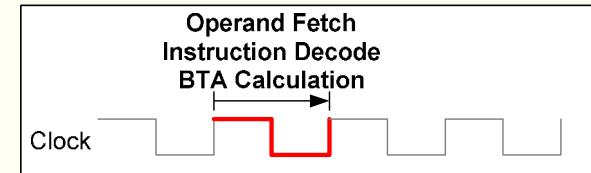
- Fase 1:
  - *Instruction fetch*
  - Cálculo de PC+4
- Fase 2:
  - Leitura dos registos
  - Descodificação da instrução
  - Cálculo do endereço-alvo das instruções de *branch*
- Fase 3:
  - Cálculo da operação na ALU
- Fase 4:
  - *Write-back*



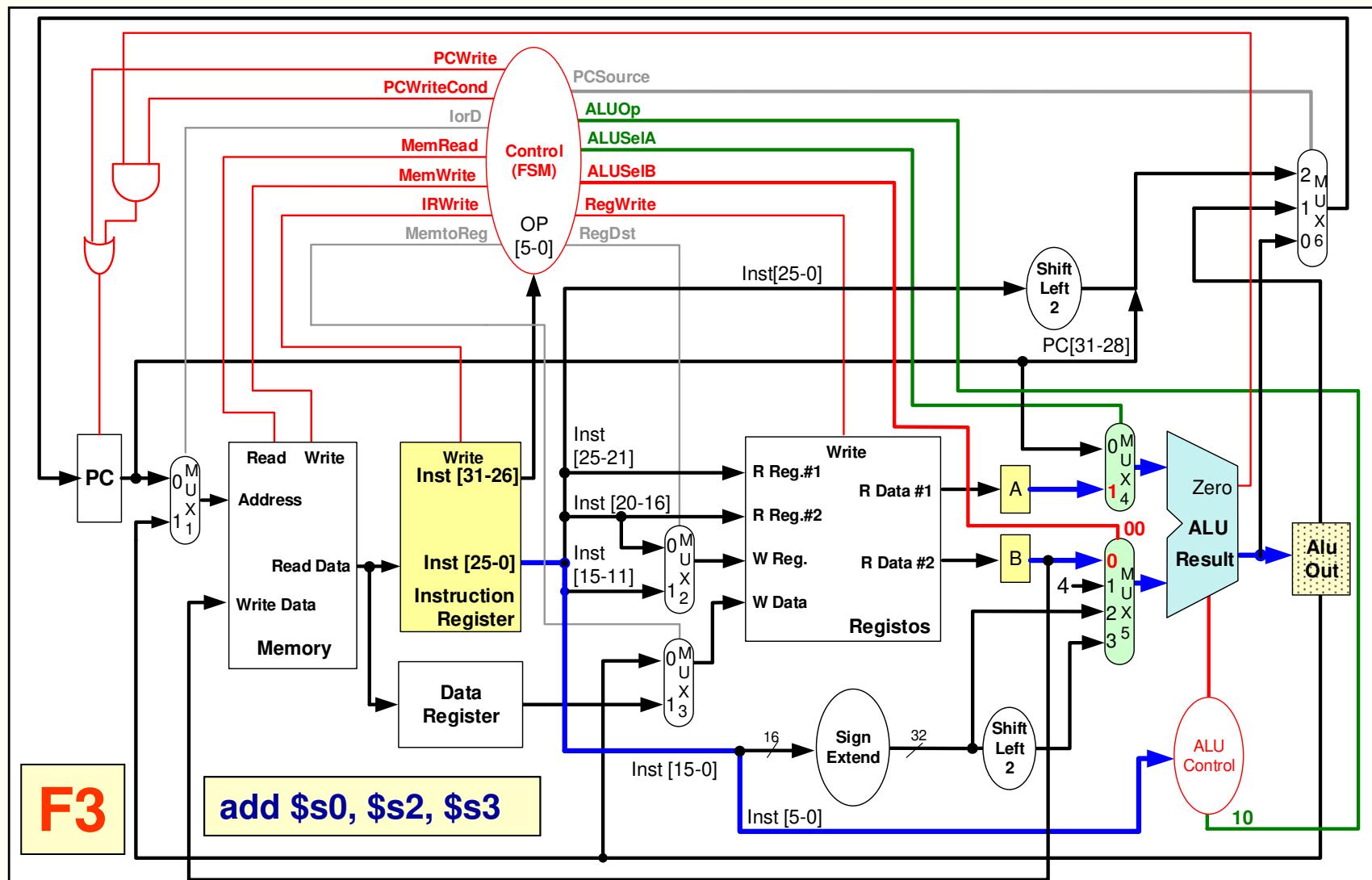
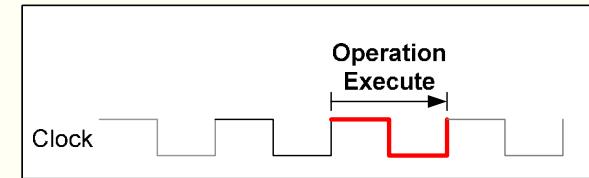
# Instruções do tipo R



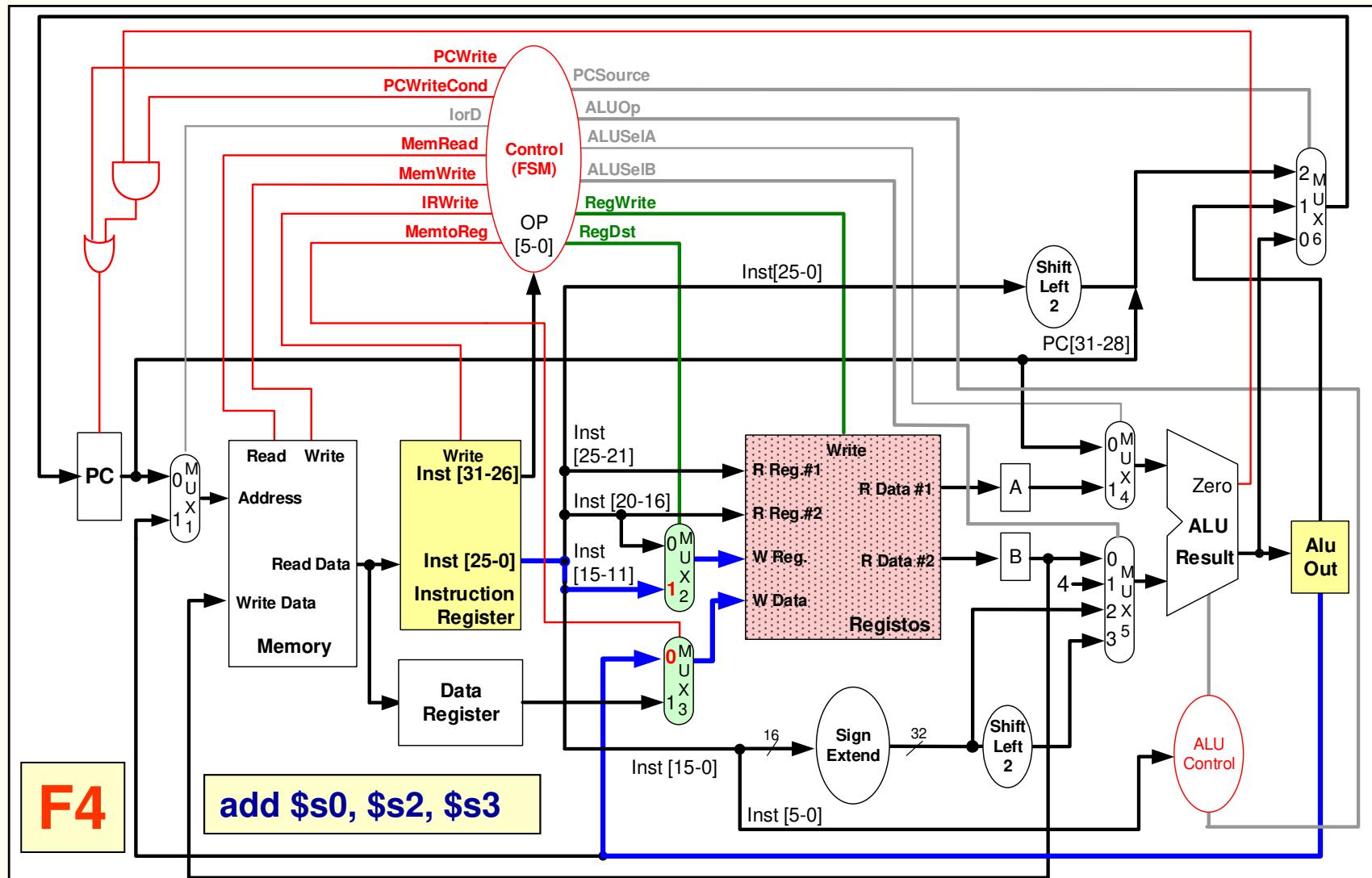
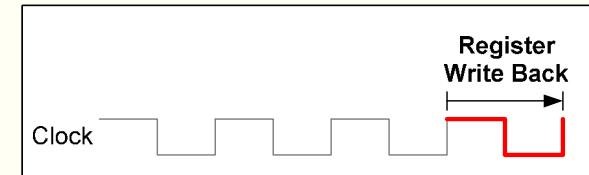
# Instruções do tipo R



## Instruções do tipo R



# Instruções do tipo R

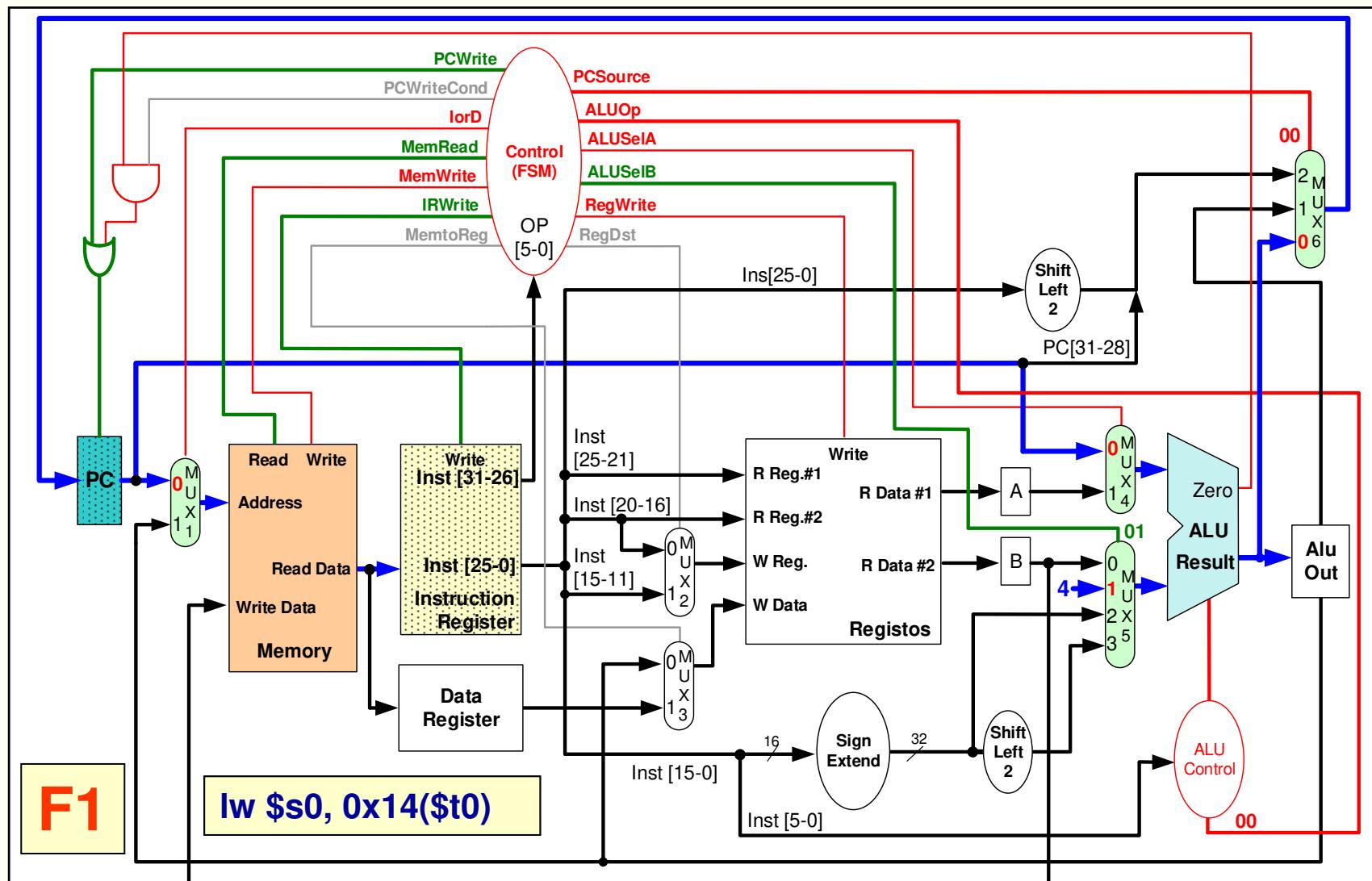
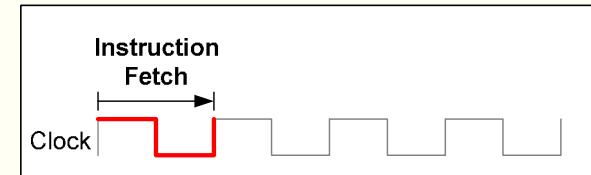


# Funcionamento do *datapath* na instrução LW

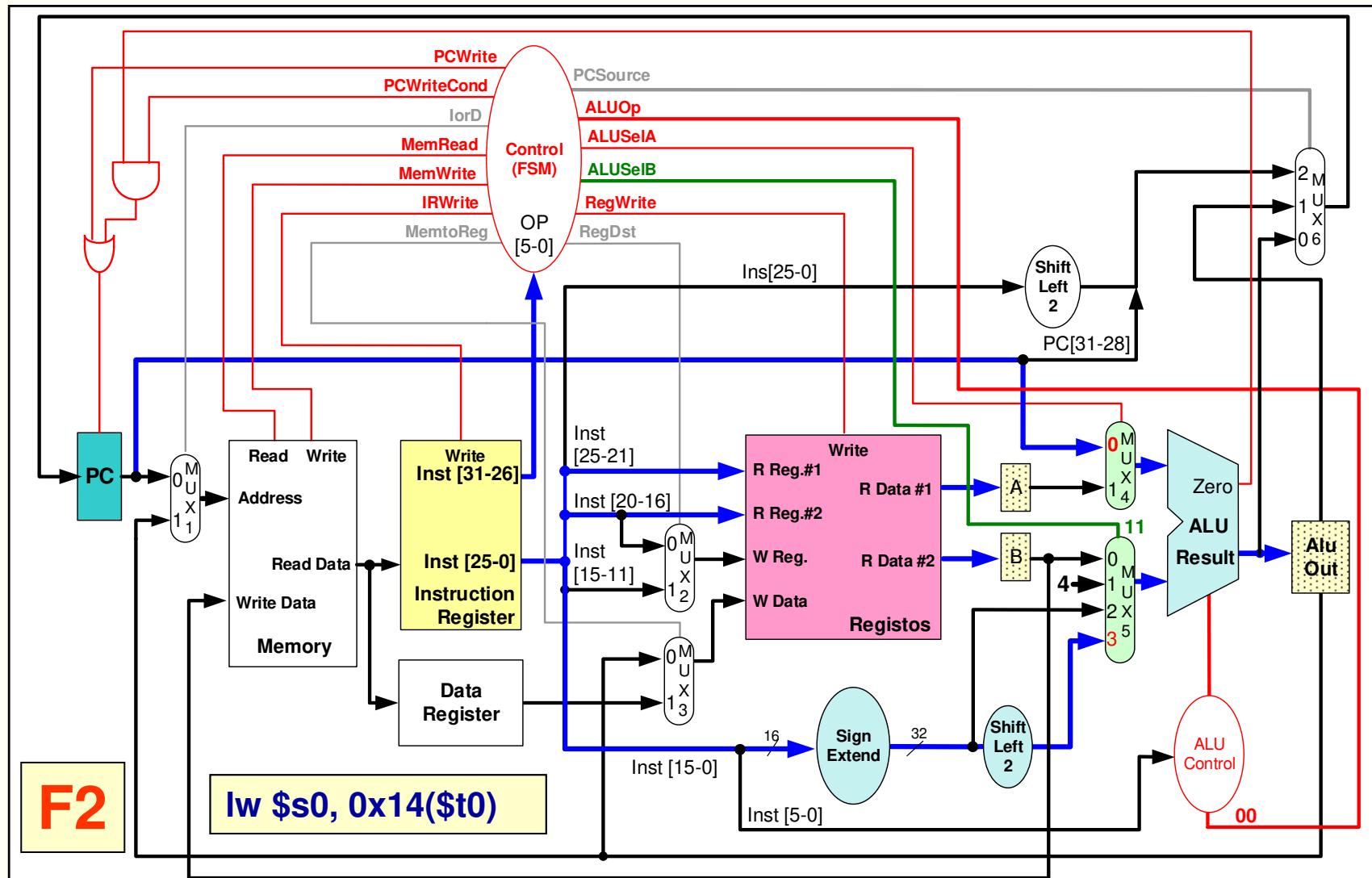
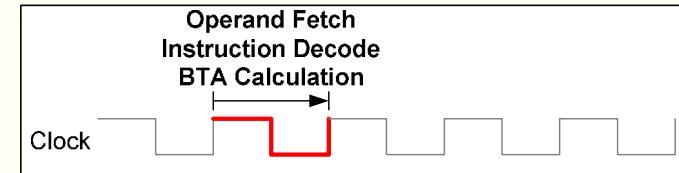
- Fase 1:
  - *Instruction fetch*
  - Cálculo de PC+4
- Fase 2:
  - Leitura dos registos
  - Descodificação da instrução
  - Cálculo do endereço-alvo das instruções de *branch*
- Fase 3:
  - Cálculo na ALU do endereço a aceder na memória
- Fase 4:
  - Leitura da memória
- Fase 5:
  - *Write-back*



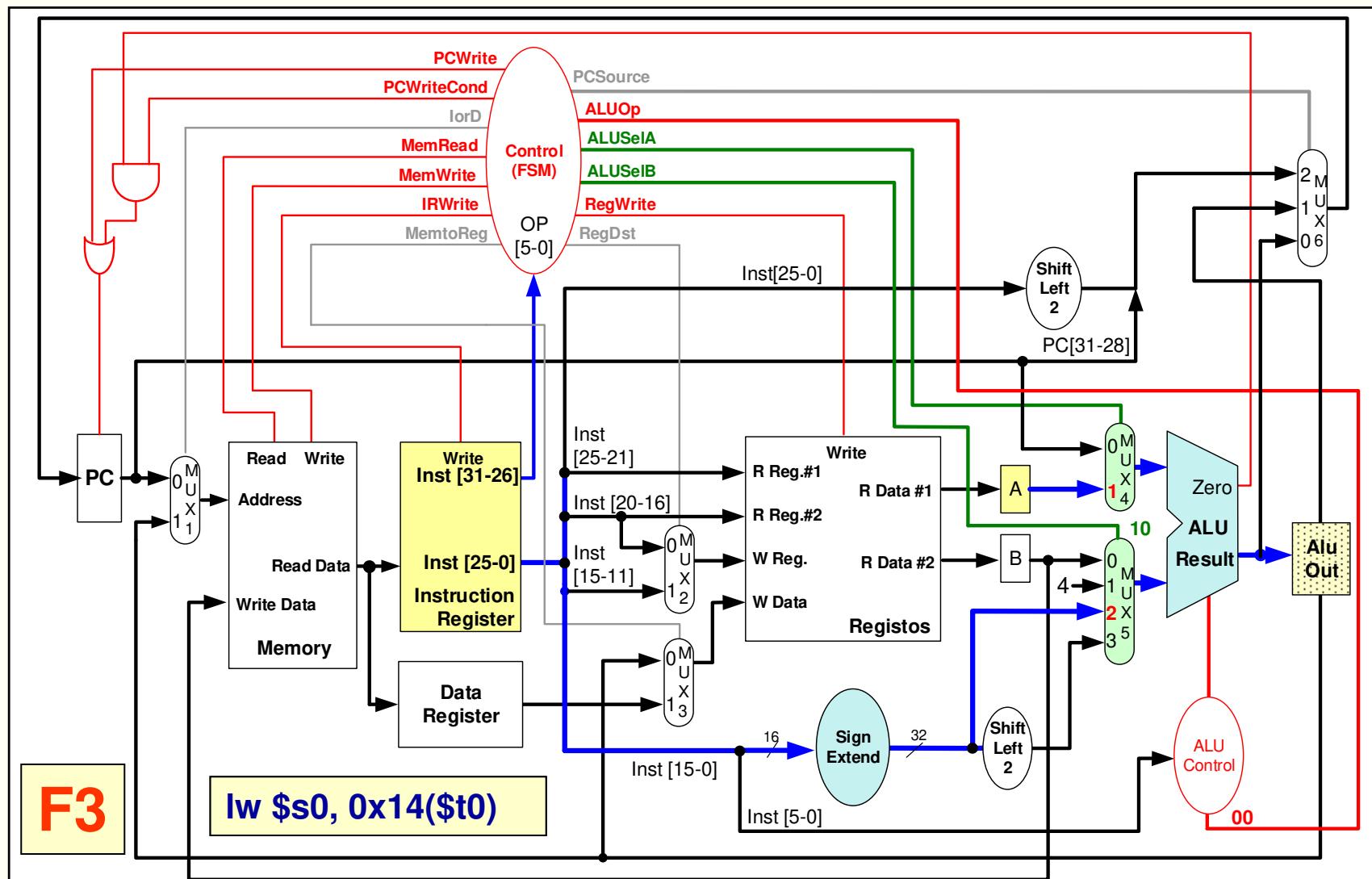
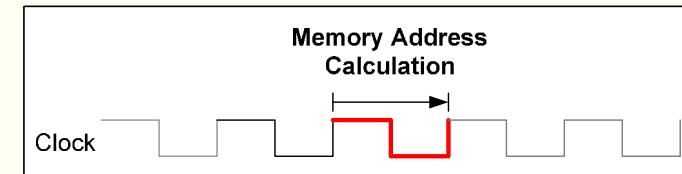
# Instrução LW



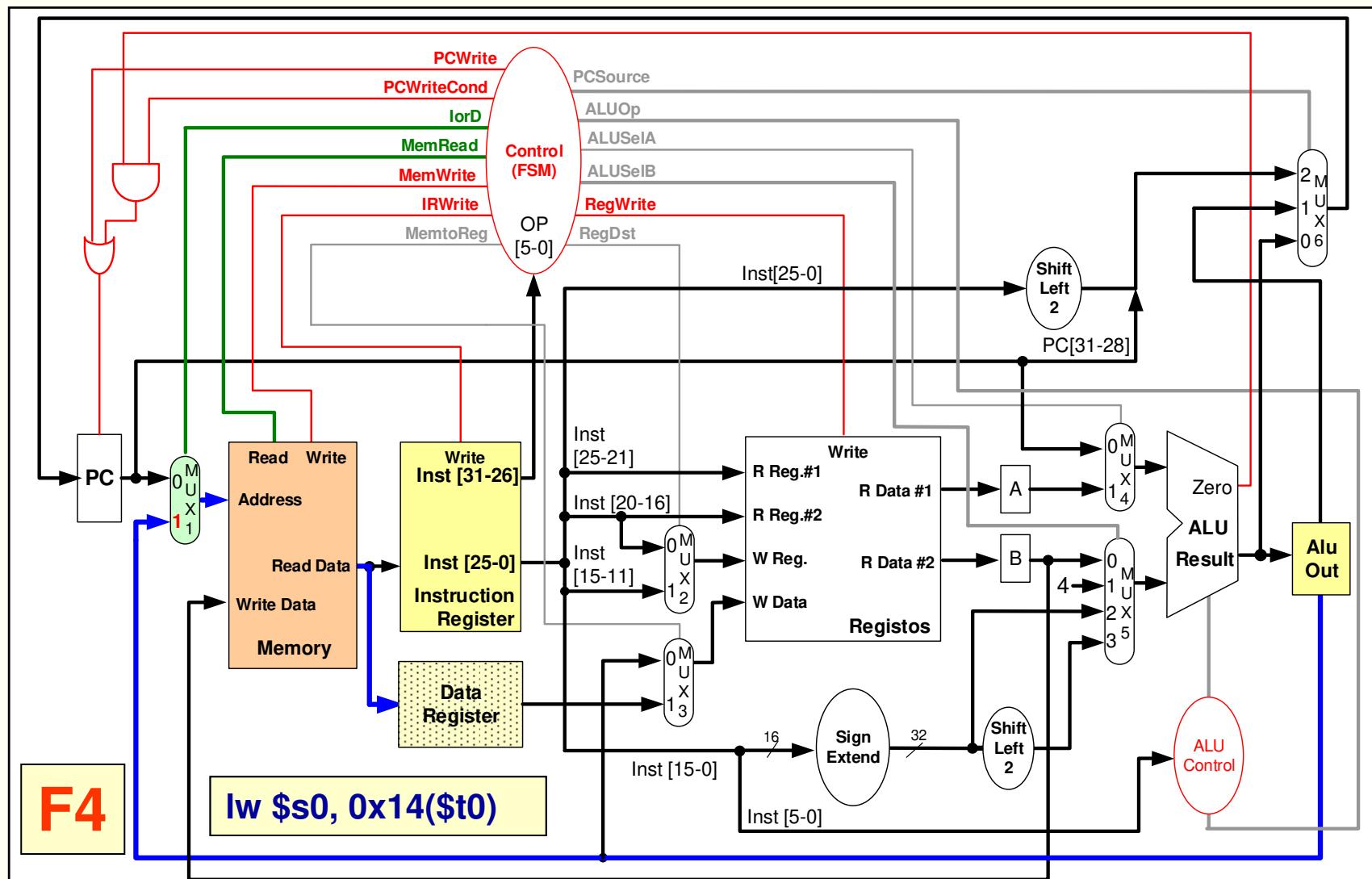
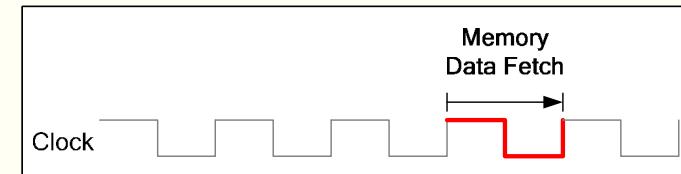
# Instrução LW



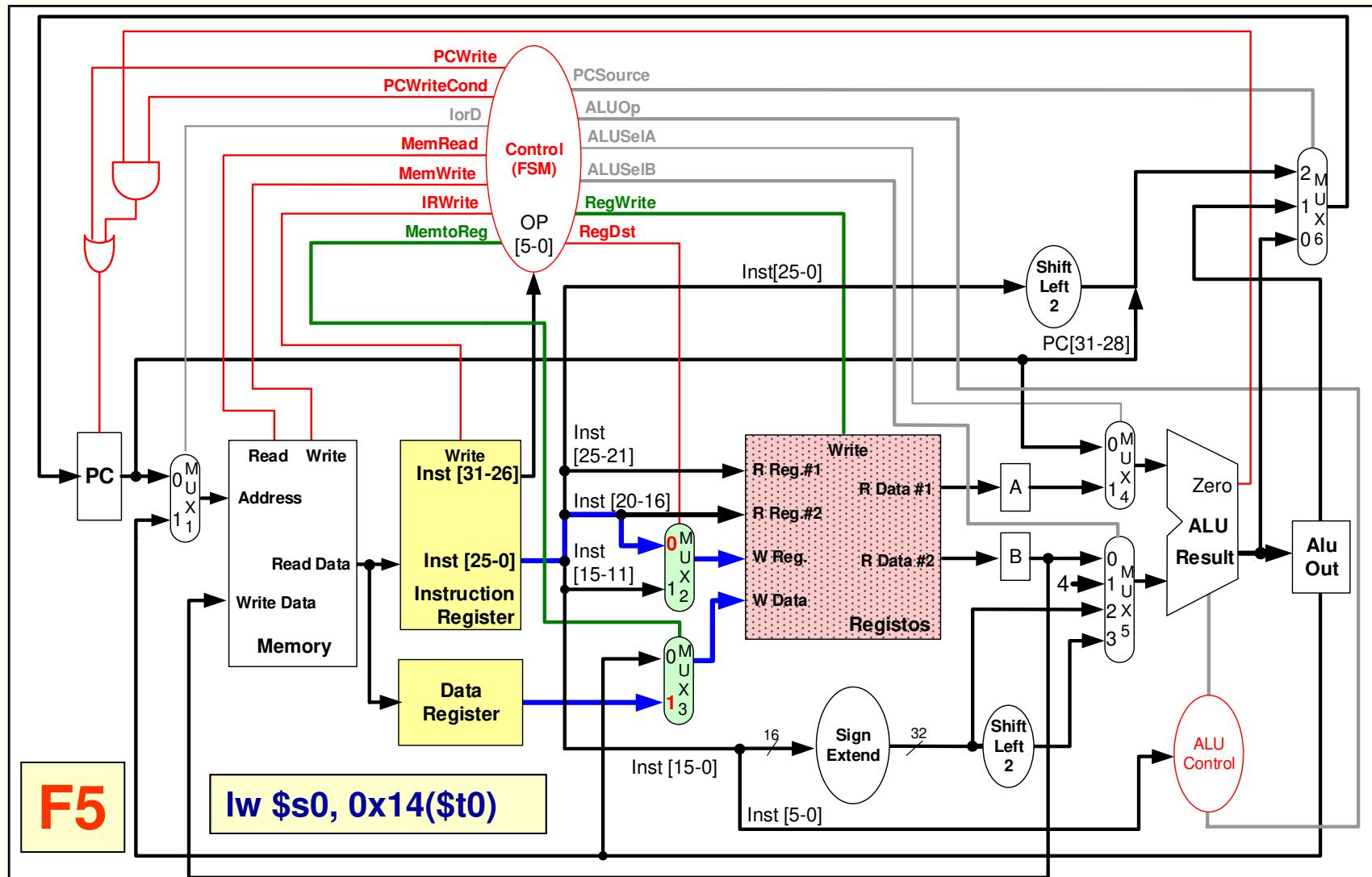
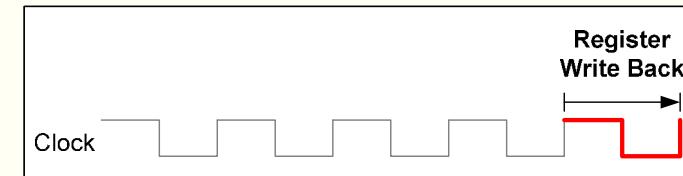
# Instrução LW



# Instrução LW



# Instrução LW

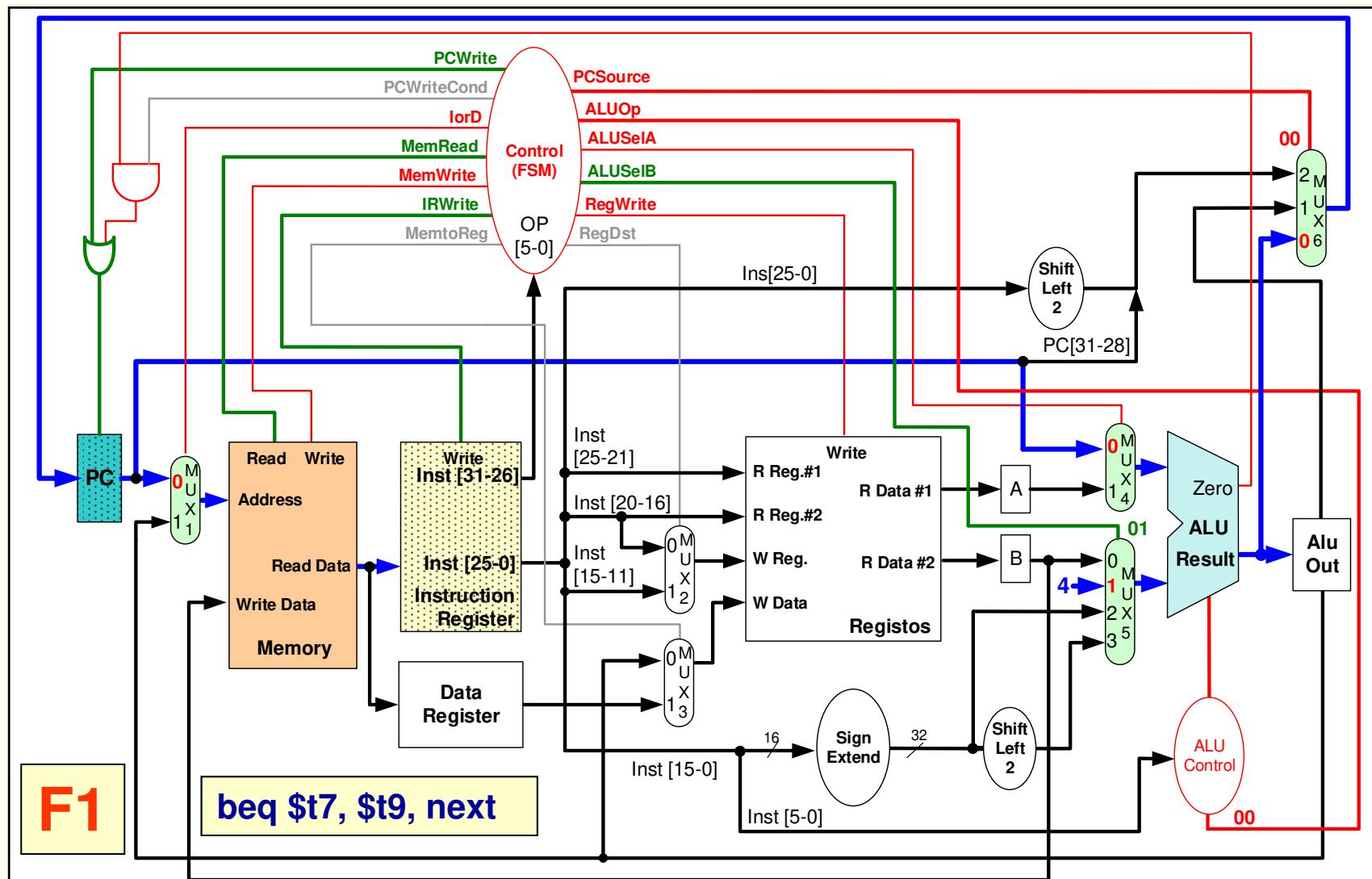
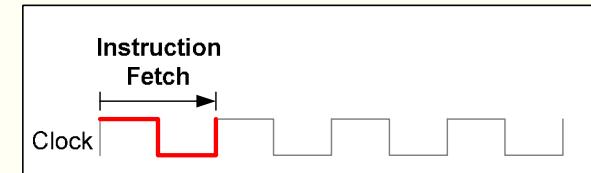


# Funcionamento do *datapath* na instrução BEQ

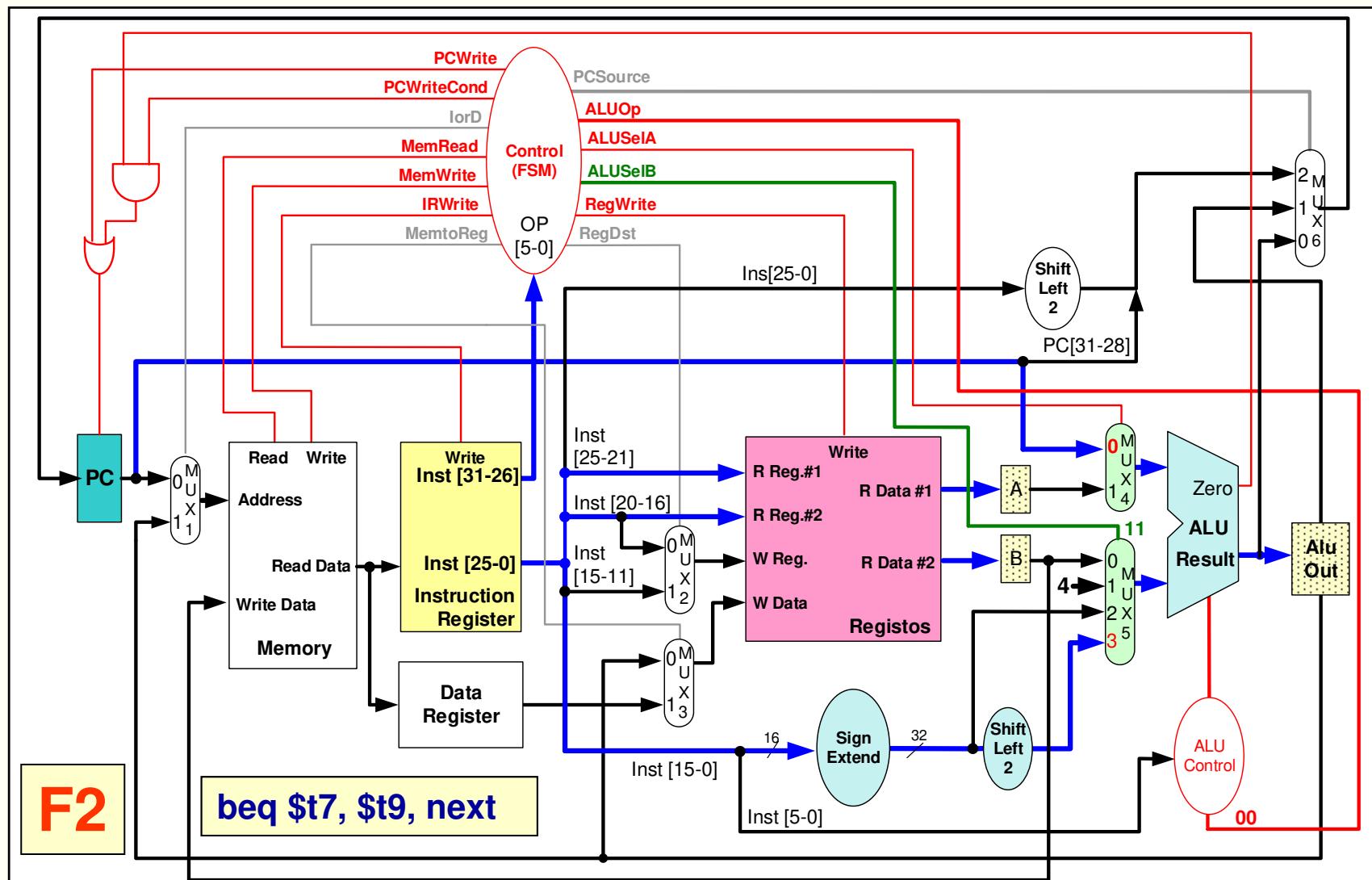
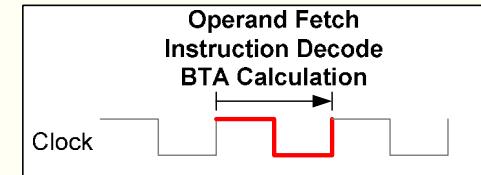
- Fase 1:
  - *Instruction fetch*
  - Cálculo de PC+4
- Fase 2:
  - Leitura dos registos
  - Descodificação da instrução
  - Cálculo do endereço-alvo das instruções de *branch* (BTA)
- Fase 3:
  - Comparação dos dois registos na ALU (subtração)
  - Conclusão da instrução de *branch* com eventual escrita do registo PC com o BTA



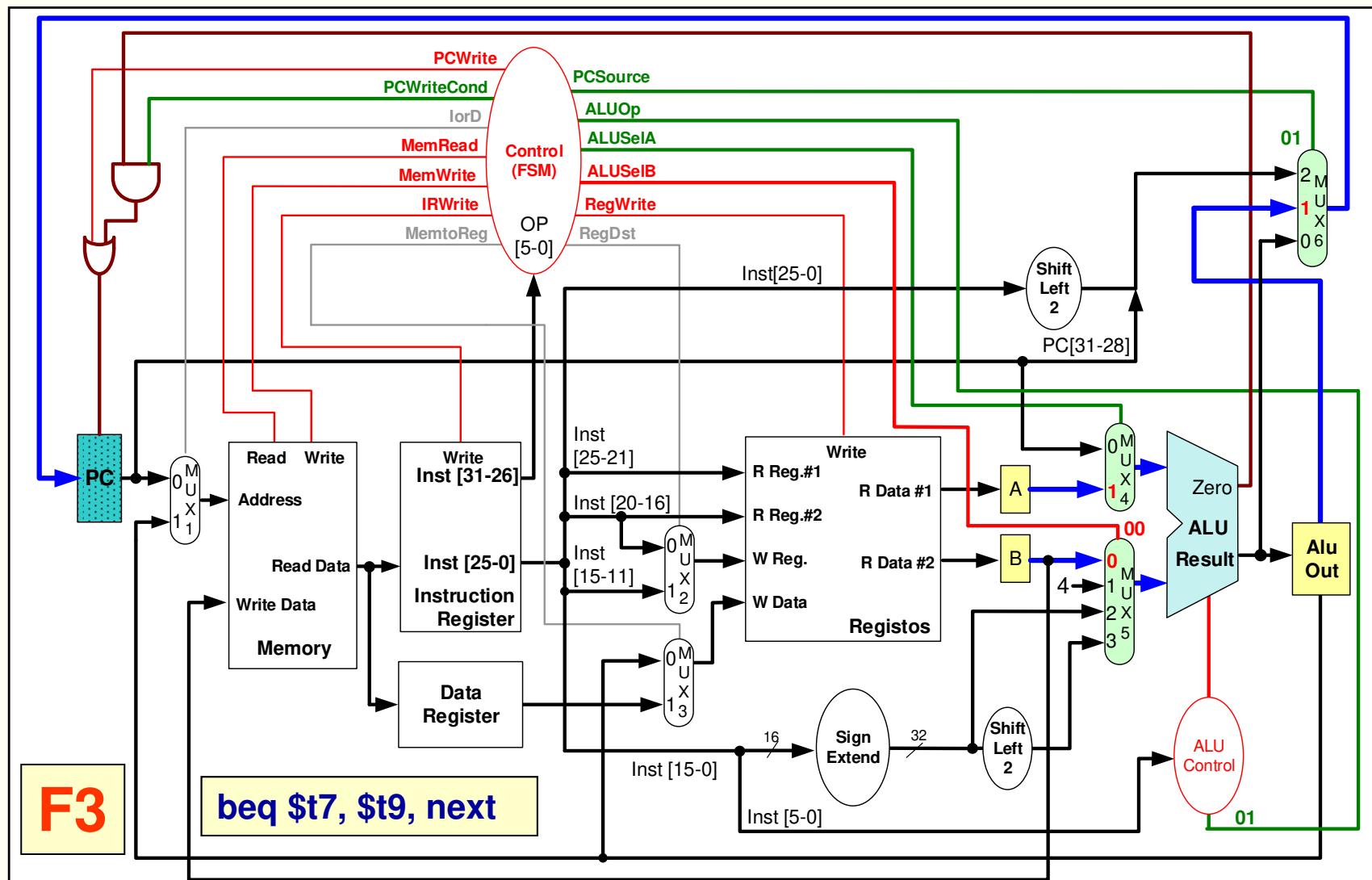
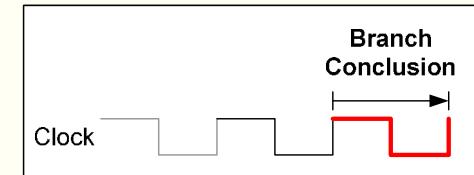
# Instrução BEQ



# Instrução BEQ



# Instrução BEQ

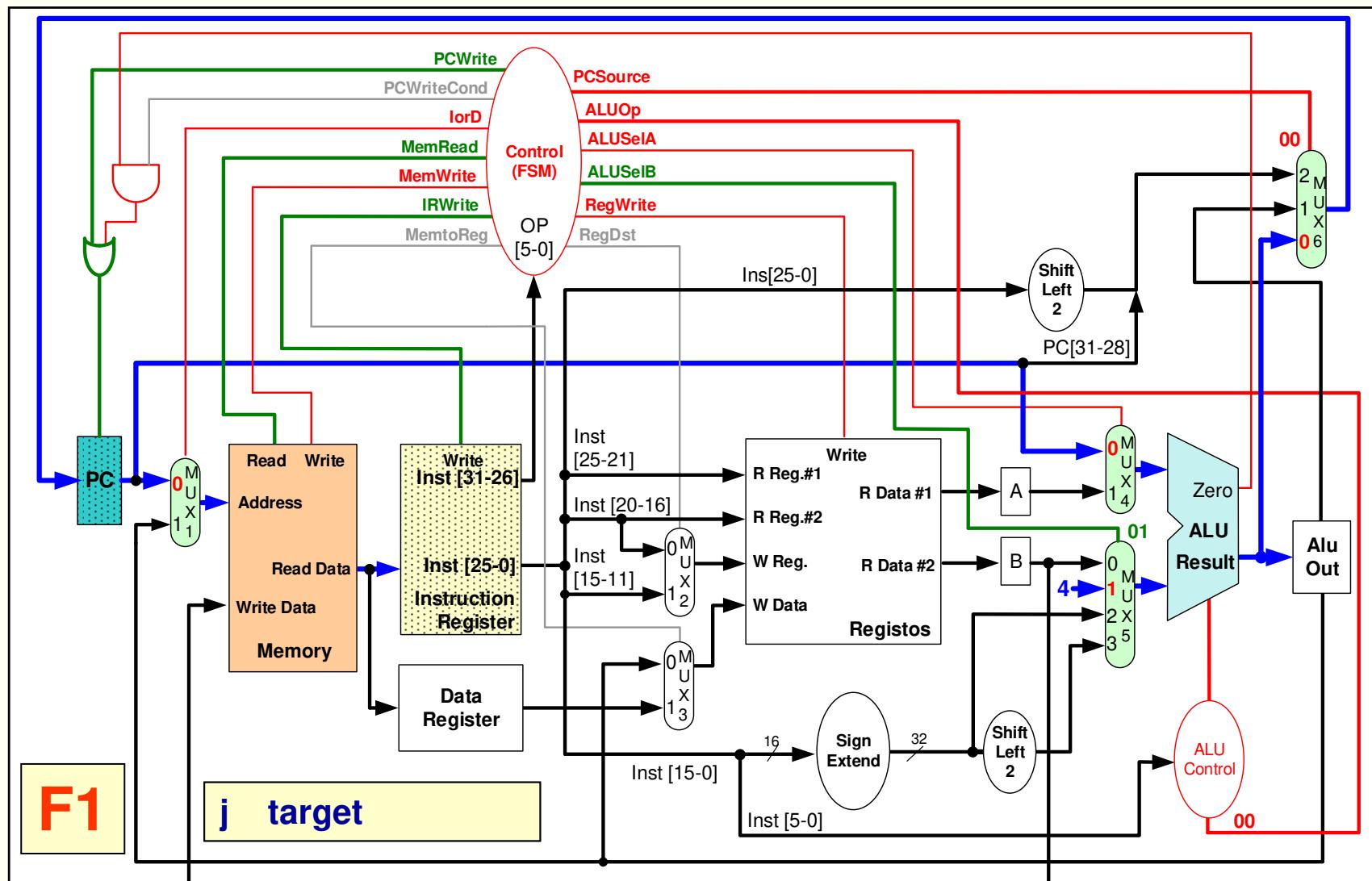
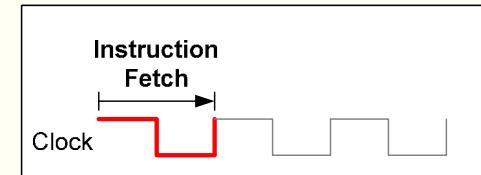


# Funcionamento do *datapath* na instrução J

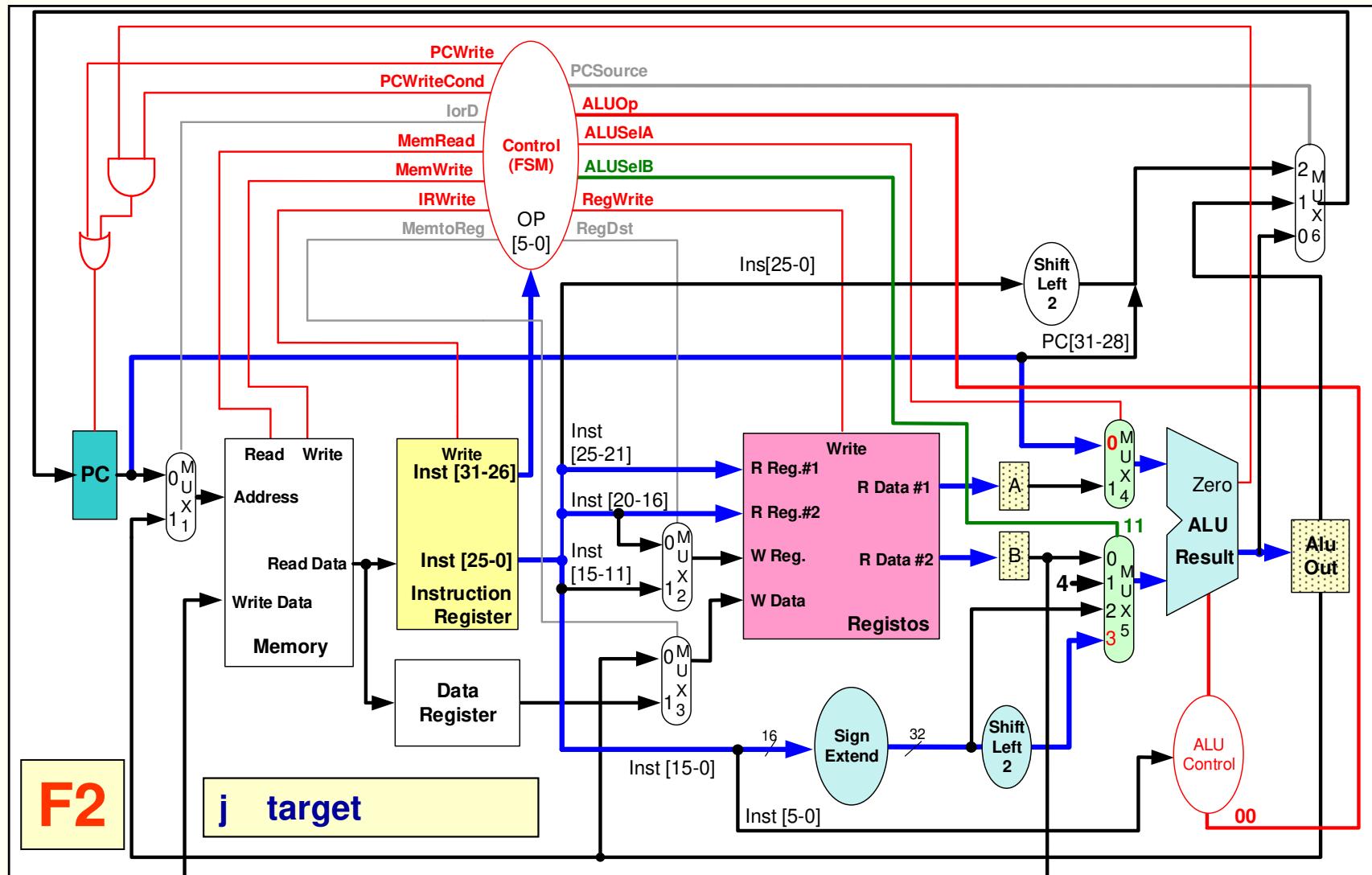
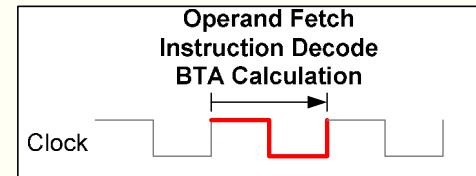
- Fase 1:
  - *Instruction fetch*
  - Cálculo de PC+4
- Fase 2:
  - Leitura dos registos
  - Descodificação da instrução
  - Cálculo do endereço-alvo das instruções de *branch*
- Fase 3:
  - Conclusão da instrução J com a seleção do JTA como próximo endereço do PC



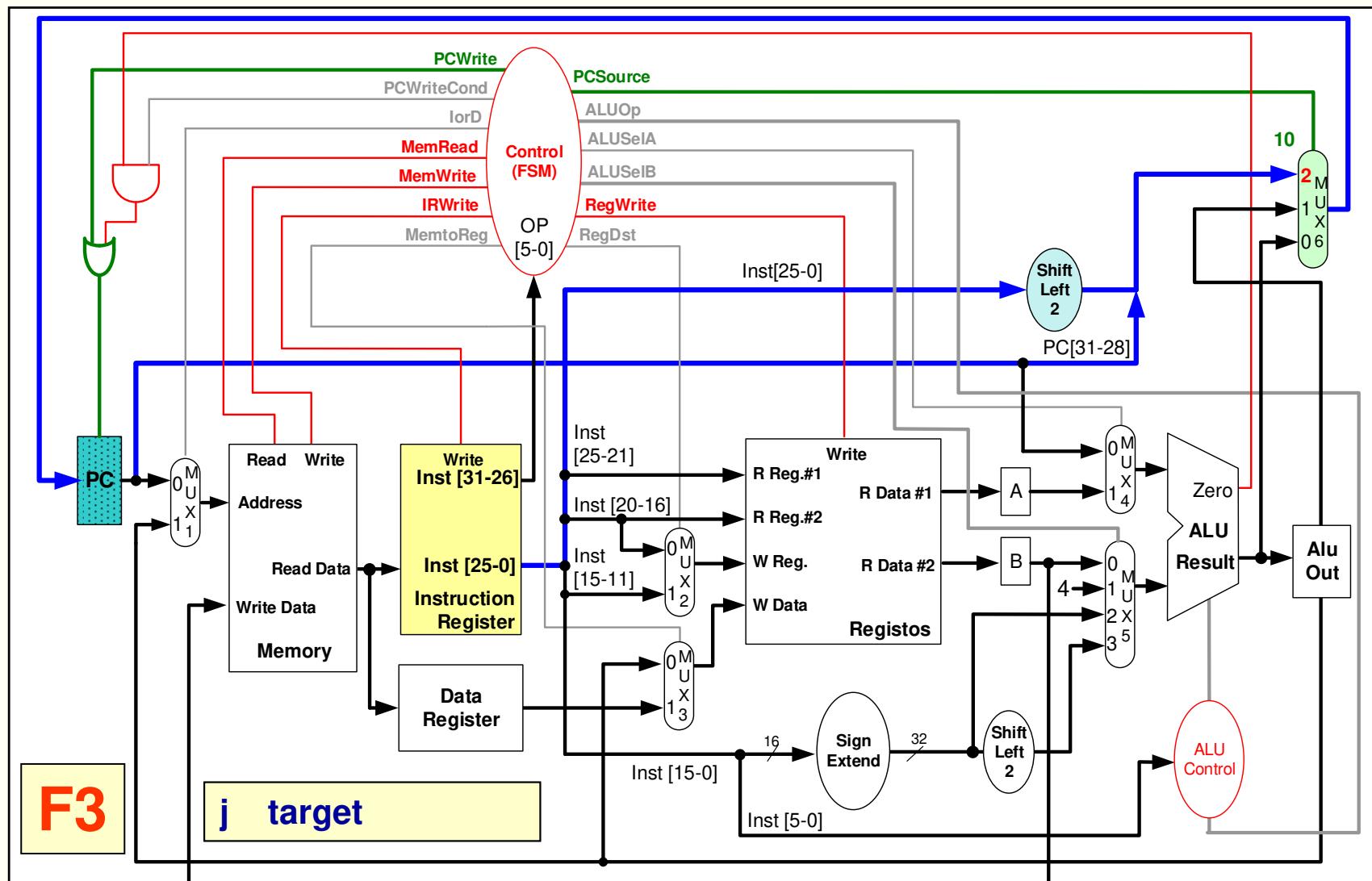
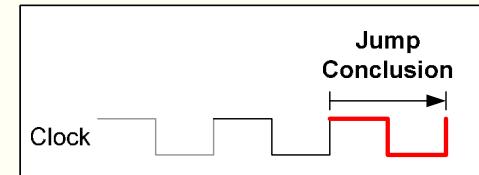
# Instrução J



# Instrução J



# Instrução J

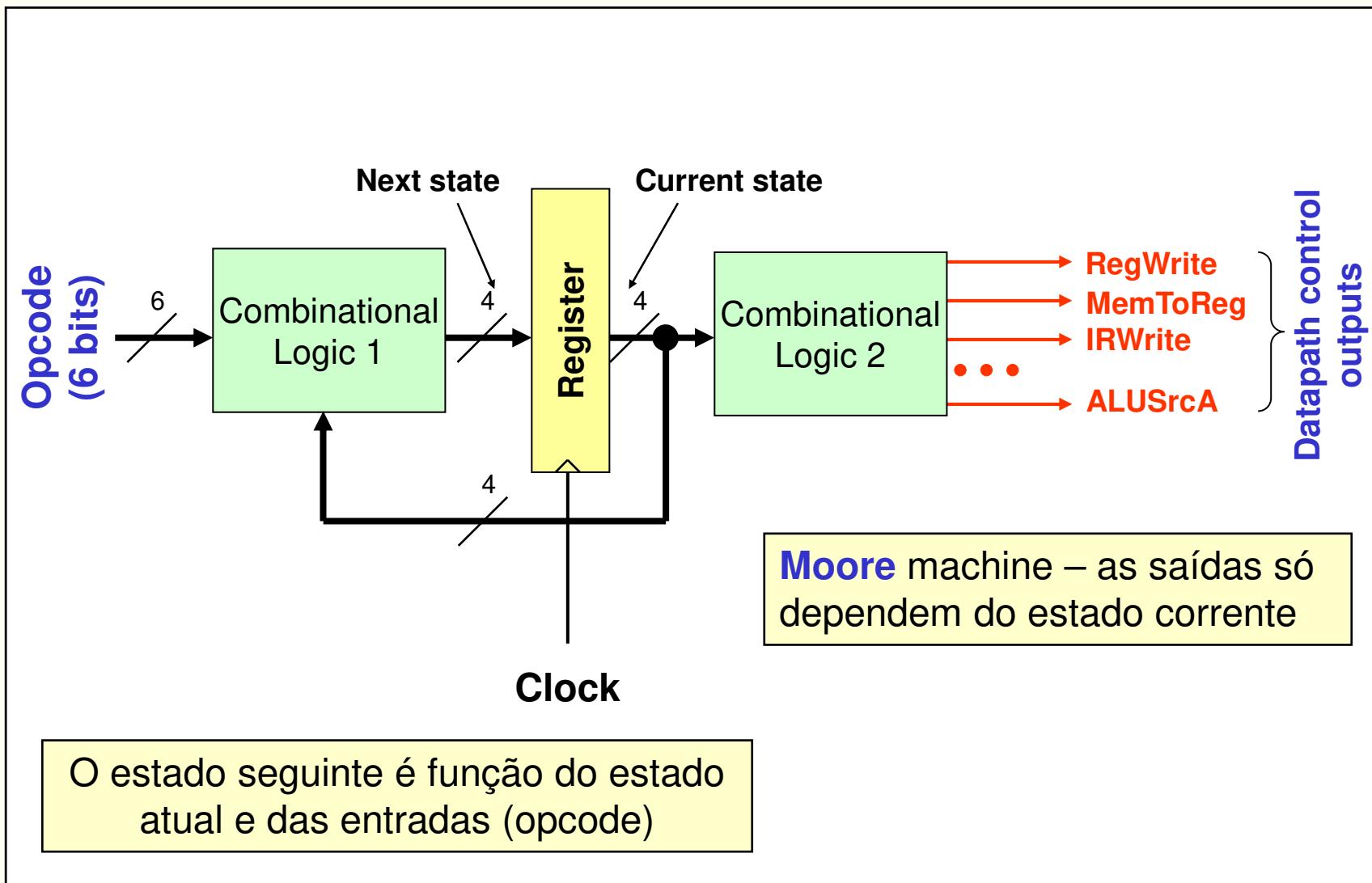


# A unidade de controlo do datapath *Multi-cycle*

- No datapath ***single-cycle***, cada instrução é executada num único ciclo de relógio:
  - a unidade de controlo é responsável pela geração de um conjunto de sinais que não se alteram durante a execução de cada instrução.
  - a relação entre os sinais de controlo e o código de operação pode assim ser gerado por um circuito meramente combinatório.
- No datapath ***multi-cycle***, cada instrução é decomposta num conjunto de ciclos de execução, correspondendo cada um destes a um período de relógio distinto:
  - a geração dos sinais de controlo ao longo do conjunto de ciclos em que é decomposta cada instrução depende da instrução particular que está a ser executada.
  - a solução combinatória deixa portanto de poder ser utilizada neste caso, sendo necessário recorrer a uma máquina de estados.

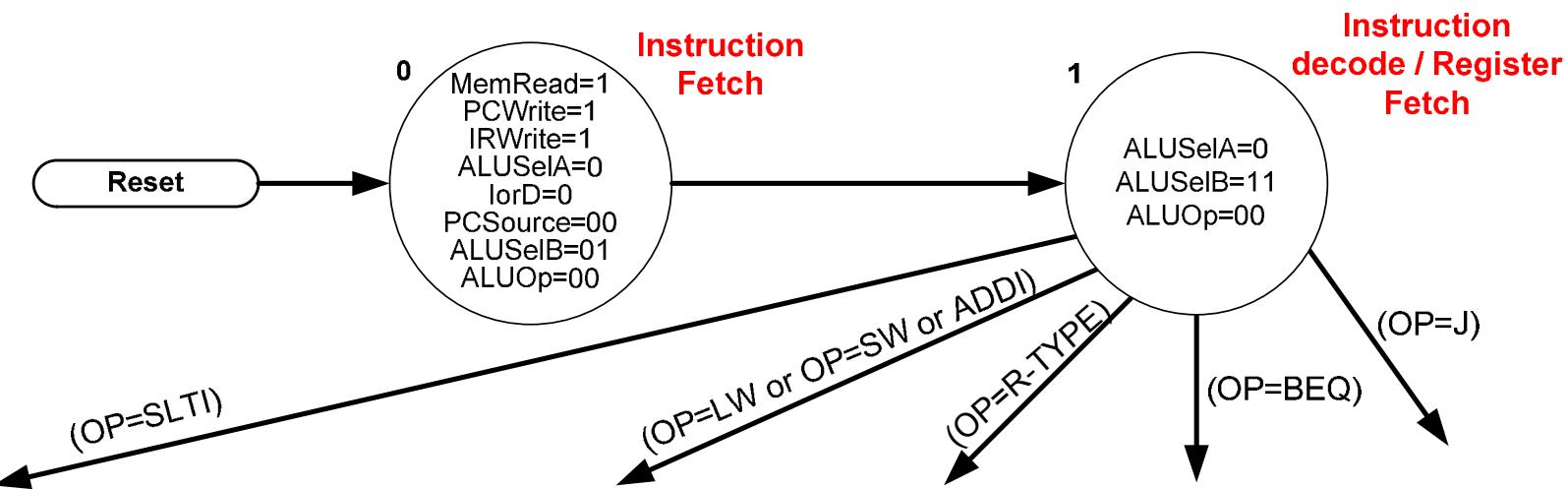


# A unidade de controlo do datapath Multi-cycle



# A unidade de controlo do datapath Multi-cycle

- Os dois primeiros ciclos de instrução são comuns a todas as instruções
- Correspondem a dois estados únicos, sendo a transição entre ambos incondicional e independente de qualquer sinal de entrada



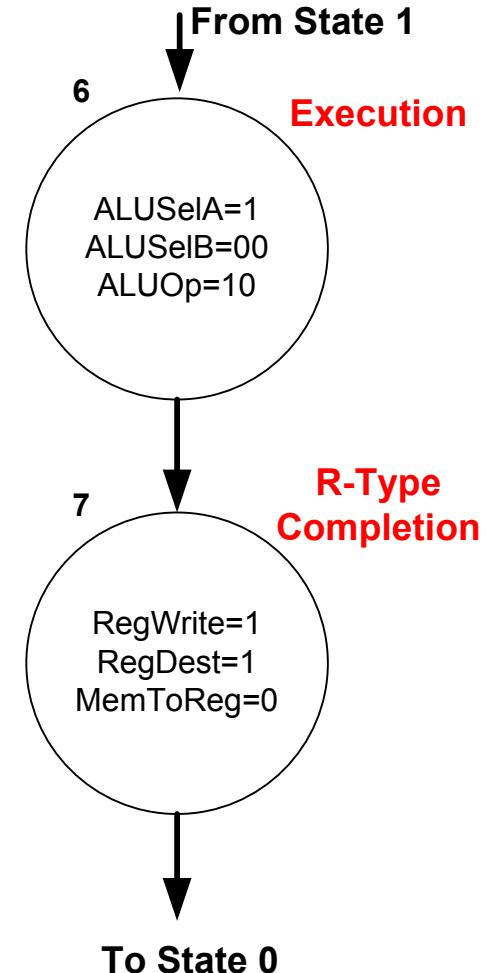
- O segundo estado tem cinco destinos distintos, dependendo do valor do campo OP da instrução

Nos diagramas apresentados os sinais de saída não explicitados em cada estado são irrelevantes (e.g. multiplexers) ou encontram-se no estado não ativo (controlo de elementos de estado).



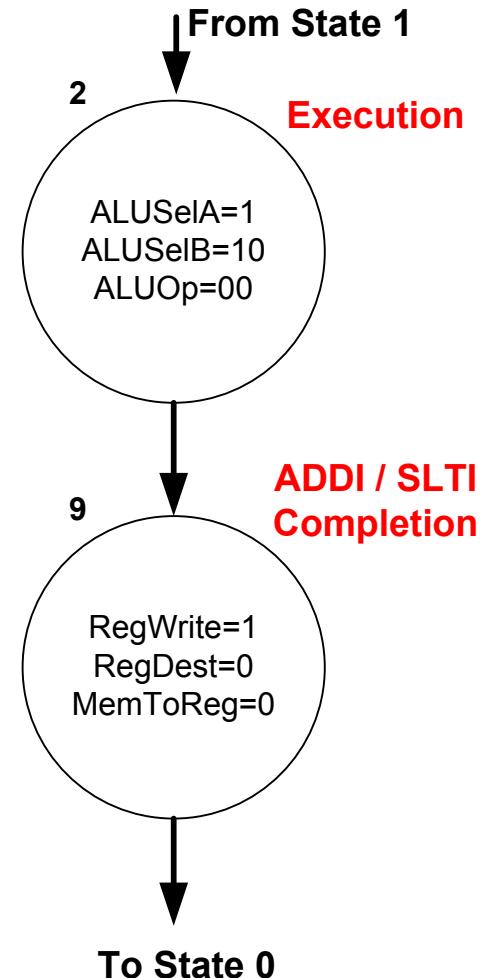
# A unidade de controlo do datapath *Multi-cycle*

- Nas **instruções do tipo "R"**, são necessários mais dois estados:
  - Um para controlar a execução da operação aritmética ou lógica e para assegurar o encaminhamento do 2º operando para a ALU
  - Outro para escrever o resultado no registo destino



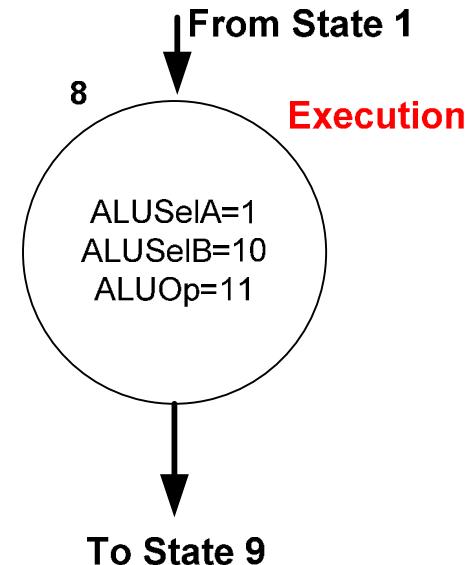
# A unidade de controlo do datapath *Multi-cycle*

- Na **instrução “ADDI”**, são necessários mais dois estados:
  - Um para definir a operação a realizar na ALU e para assegurar o encaminhamento do 2º operando para a ALU
  - Outro para escrever o resultado no registo destino



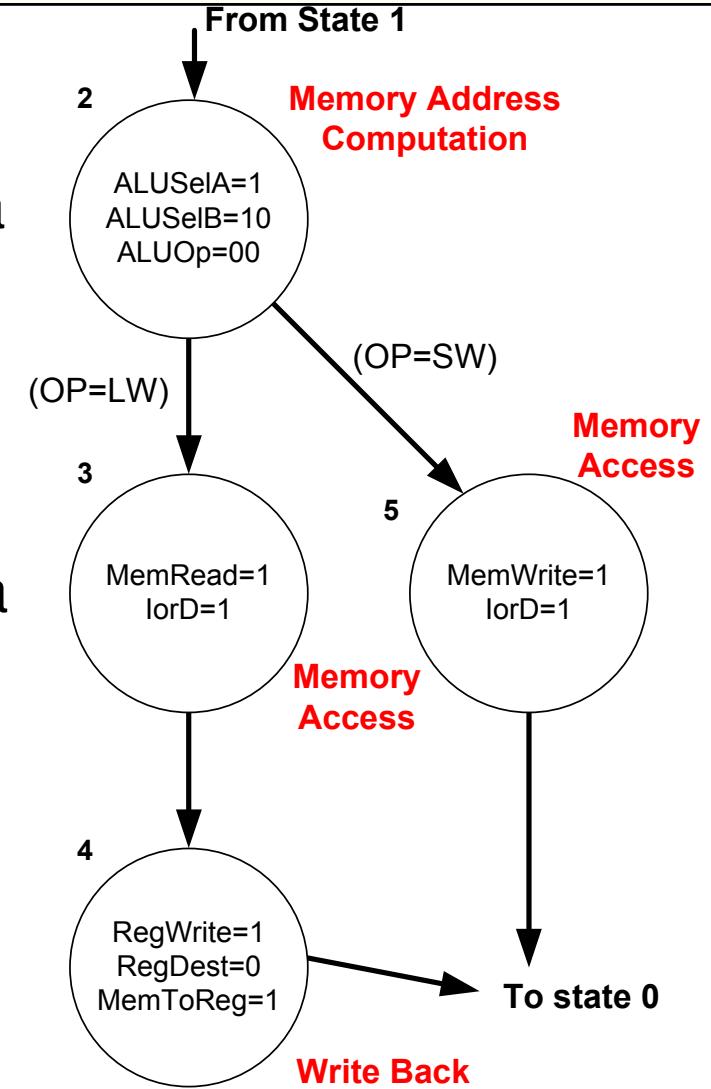
# A unidade de controlo do datapath *Multi-cycle*

- Para a **instrução “SLTI”**, é necessário mais um estado:
  - Para definir a operação a realizar na ALU e para assegurar o encaminhamento do 2º operando para a ALU
  - A conclusão desta instrução é igual à instrução “ADDI” (daí a partilha do estado 9)



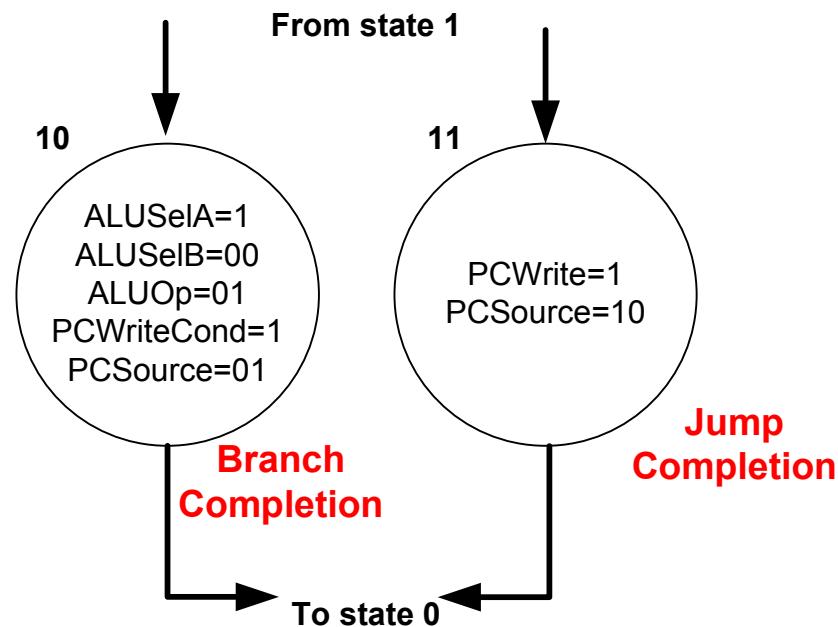
# A unidade de controlo do datapath *Multi-cycle*

- Nas **instruções de “load / store”**, o estado dois é dedicado a determinar o endereço da memória externa sobre a qual será efetuada a operação de escrita ou leitura
- A instrução de "load" obriga a um estado suplementar, face à instrução de "store", para permitir a escrita do valor lido no registo destino

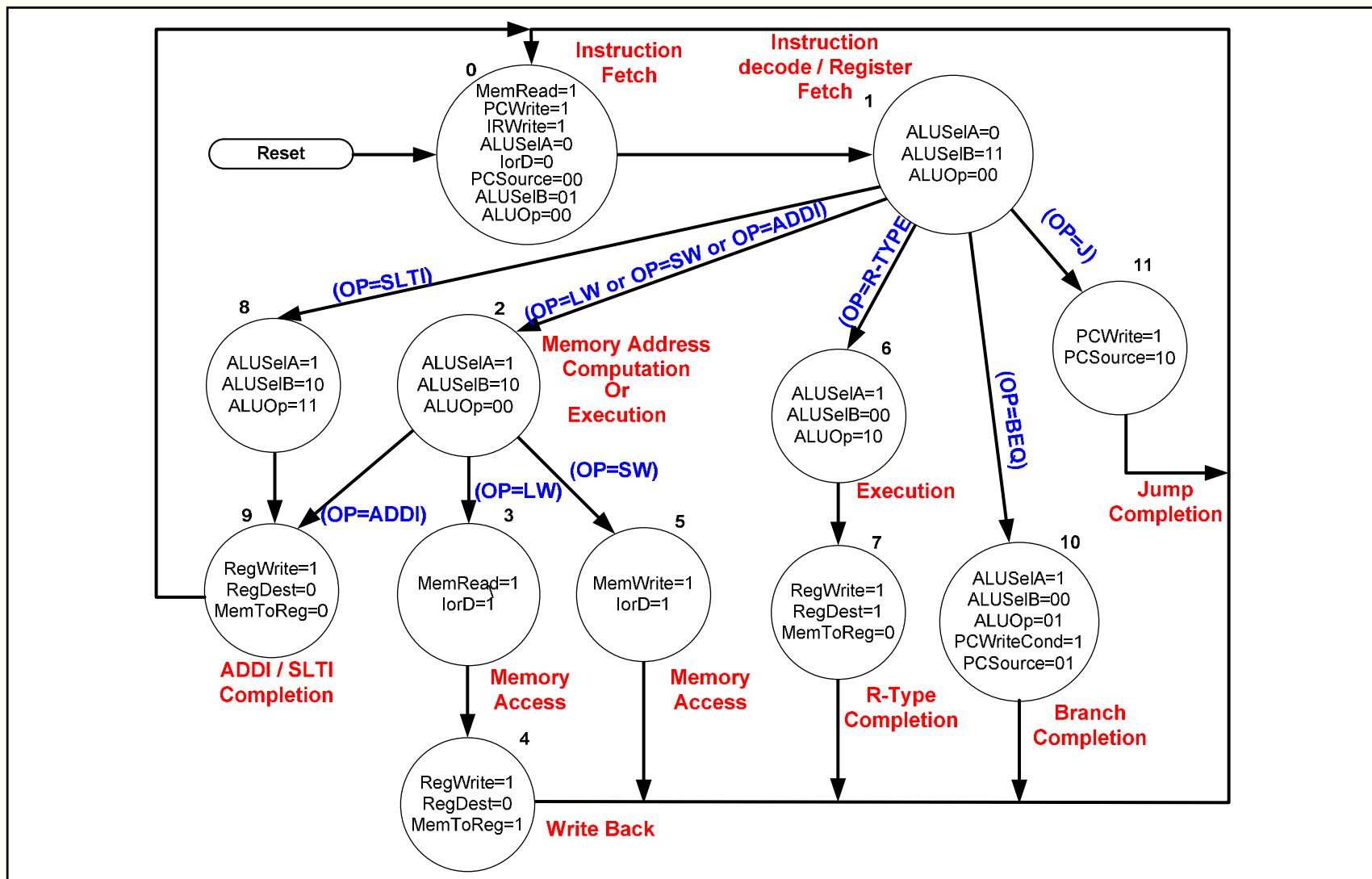


# A unidade de controlo do datapath *Multi-cycle*

- As **instruções de "branch"** condicional e as instruções de **"jump"**, finalmente, carecem apenas de mais um estado para poderem ser completadas



# A unidade de controlo do datapath Multi-cycle



# A unidade de controlo do *datapath Multi-cycle*

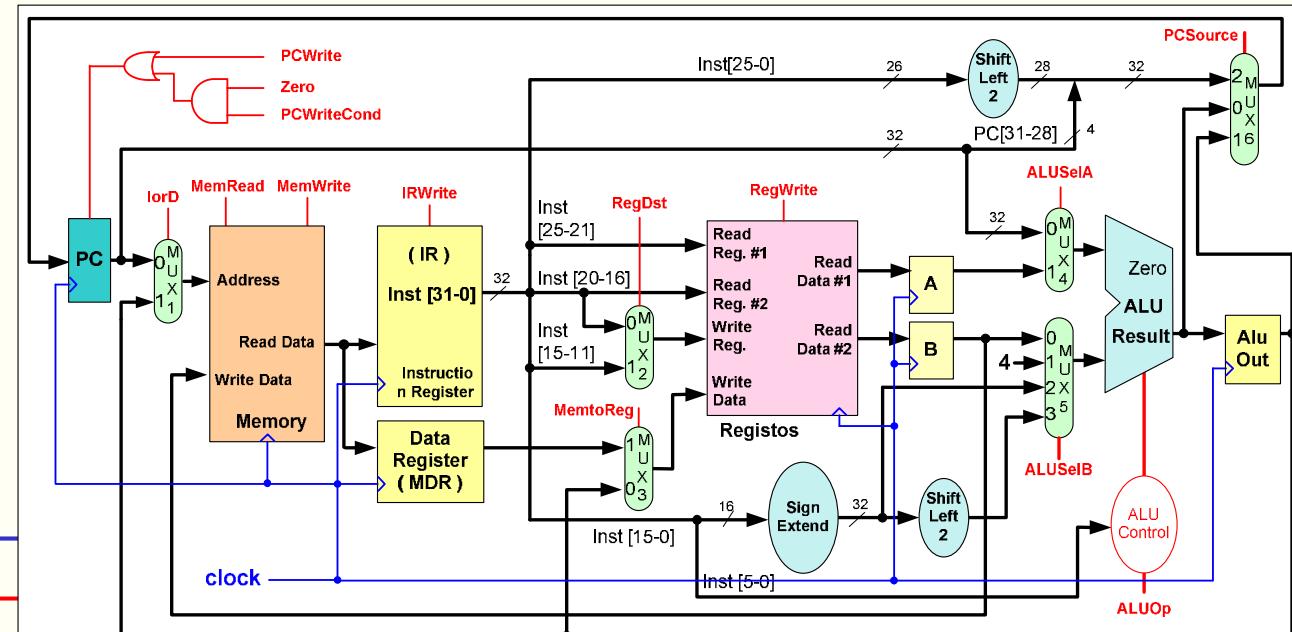
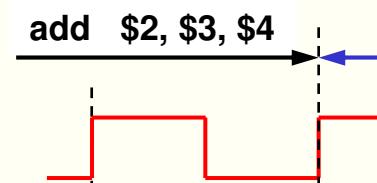
- A unidade de controlo que acabamos de desenhar tem apenas 12 estados (4 variáveis de estado). A representação do seu aspeto funcional na forma de um diagrama de estados é portanto perfeitamente razoável
- Uma versão completa do *datapath* do MIPS (mais de cem instruções distintas) pode implicar ciclos de execução que variem entre dois e vinte períodos de relógio, complicando significativamente o diagrama de estados
- Em arquiteturas do Set de Instruções mais complexas, com um número muito superior de instruções agrupadas num número muito variado de classes, a unidade de controlo pode requerer milhares de estados agrupados em centenas de sequências distintas
- Nestes casos, o recurso a uma representação gráfica da máquina de estados é não só inapropriada como virtualmente impossível de realizar. A **micro-programação** é uma forma alternativa de representar a unidade de controlo do ponto de vista funcional



## Funcionamento do DP

**add** \$2, \$3, \$4  
**sw** \$2, -4(\$6)  
**or** \$4, \$6, \$3

Sinais de controlo na execução sequencial das três instruções:



|                    | 0  | X  | 0  | 0  | 0  | X  | 0  |
|--------------------|----|----|----|----|----|----|----|
| <b>PCWriteCond</b> | 0  | X  | 0  | 0  | 0  | X  | 0  |
| <b>PCWrite</b>     | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
| <b>MemWrite</b>    | 0  | 0  | 0  | 0  | 1  | 0  | 0  |
| <b>MemRead</b>     | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
| <b>MemToReg</b>    | 0  | X  | X  | X  | X  | X  | X  |
| <b>IRWrite</b>     | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
| <b>ALUSelA</b>     | X  | 0  | 0  | 1  | X  | 0  | 0  |
| <b>ALUSelB</b>     | XX | 01 | 11 | 10 | XX | 01 | 11 |
| <b>ALUOp</b>       | XX | 00 | 00 | 00 | XX | 00 | 00 |
| <b>IorD</b>        | X  | 0  | X  | X  | 1  | 0  | X  |
| <b>PCSource</b>    | XX | 00 | XX | XX | XX | 00 | XX |
| <b>RegWrite</b>    | 1  | 0  | 0  | 0  | 0  | 0  | 0  |
| <b>RegDst</b>      | 1  | X  | X  | X  | X  | X  | X  |

**sw \$2, -4(\$6)**

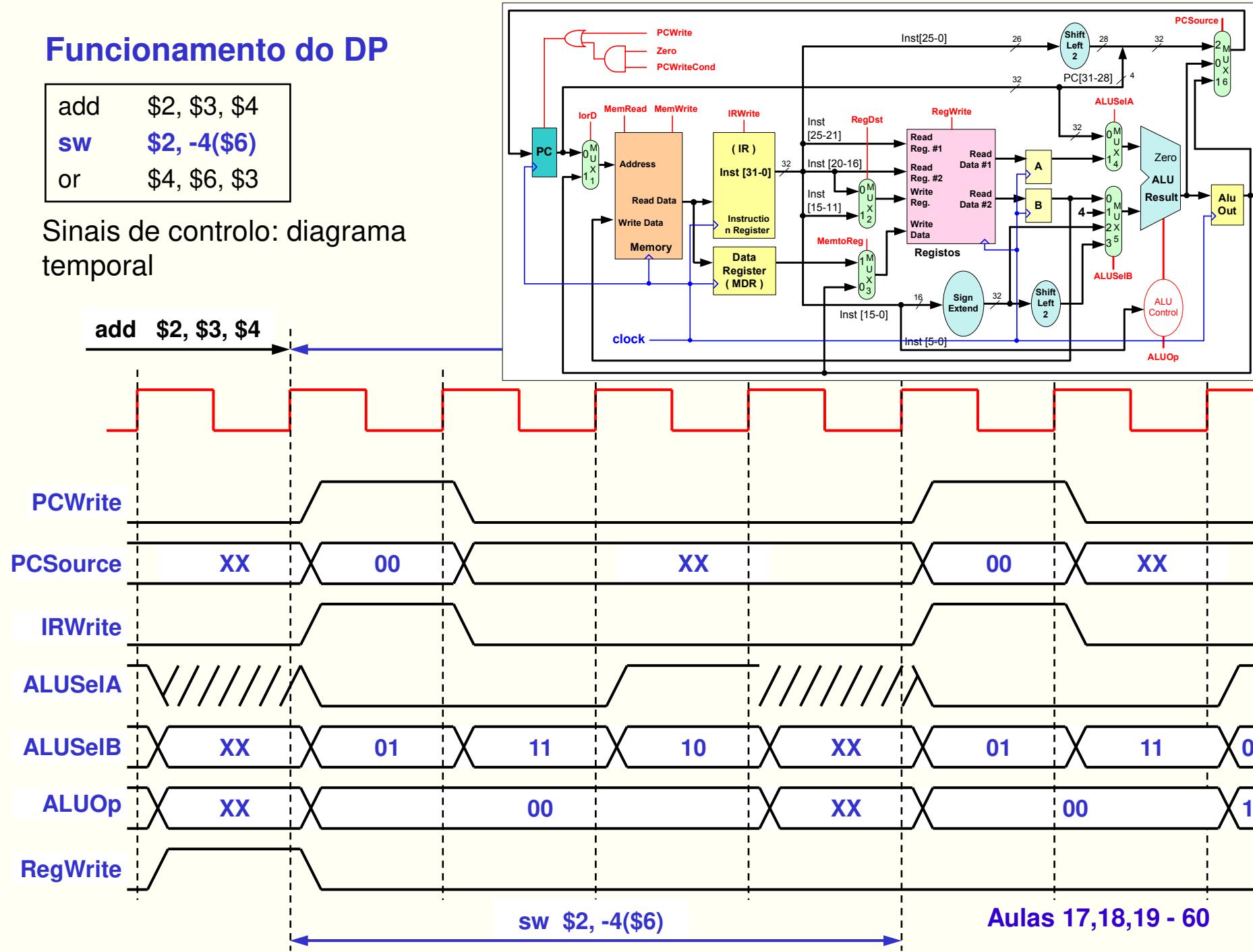
Aulas 17,18,19 - 59

## Funcionamento do DP

**add** \$2, \$3, \$4  
**sw** \$2, -4(\$6)  
**or** \$4, \$6, \$3

Sinais de controlo: diagrama temporal

**add \$2, \$3, \$4**



## Funcionamento do DP

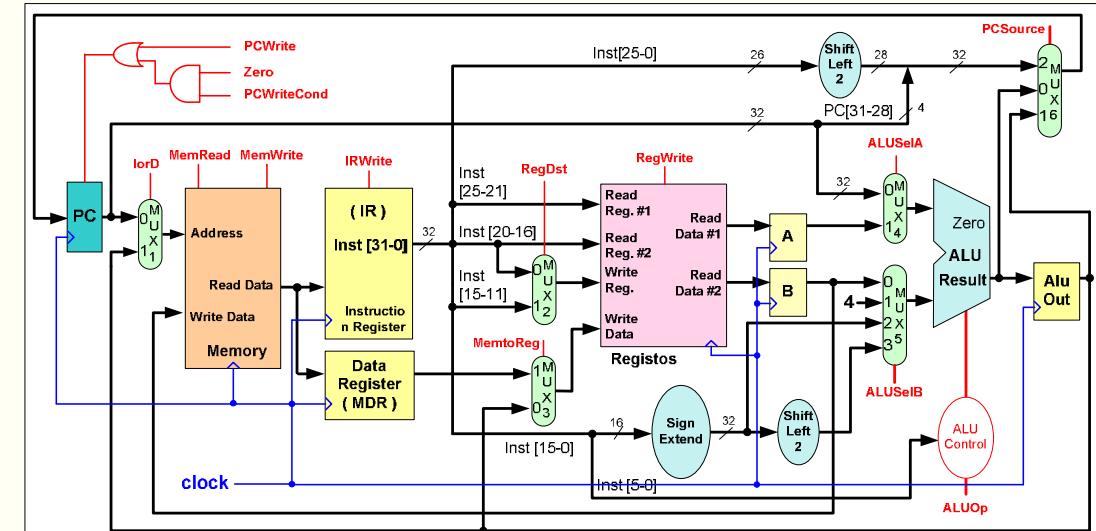
00400048 add \$2, \$3, \$4 # 00641020

0040004C sw \$2, -4(\$6) # ACC2FFFC

00400050 or \$4, \$6, \$3 # 00C32025

Valores calculados / obtidos em cada ciclo de relógio:

|     |          |
|-----|----------|
| \$3 | 20001FA6 |
| \$4 | 81002378 |
| \$6 | 10012480 |



|          | add \$2, \$3, \$4 | sw \$2, -4(\$6) | or \$4, \$6, \$3 |          |          |          |          |
|----------|-------------------|-----------------|------------------|----------|----------|----------|----------|
| PC       | 0040004C          | 0040004C        | 00400050         | 00400050 | 00400050 | 00400050 | 00400054 |
| IR       | 00641020          | 00641020        | ACC2FFFC         | ACC2FFFC | ACC2FFFC | ACC2FFFC | 00C32025 |
| MDR      | ?                 | ?               | ACC2FFFC         | ?        | ?        | ?        | 00C32025 |
| ALU Res  | ?                 | 00400050        | 00400040         | 1001247C | ?        | 00400054 | 004080E8 |
| ALU Out  | A100431E          | ?               | 00400050         | 00400040 | 1001247C | ?        | 00400054 |
| ALU Zero | ?                 | 0               | 0                | 0        | ?        | 0        | 0        |
| A        | 20001FA6          | 20001FA6        | 20001FA6         | 10012480 | 10012480 | 10012480 | 10012480 |
| B        | 81002378          | 81002378        | 81002378         | A100431E | A100431E | A100431E | A100431E |
| ALUOp    | XX                | 00              | 00               | 00       | XX       | 00       | 00       |

Opcodes: SW - 0x2B, ADD - 0x20, OR - 0x25

Aulas 17,18,19 - 61

# Exercício

- Calcule o número de ciclos de relógio que o programa seguinte demora a executar, desde o *Instruction Fetch* da 1<sup>a</sup> instrução até à conclusão da última instrução:
  - 1) num *datapath single-cycle*, 2) num *datapath multi-cycle*

**main:**

```
lw    $1, 0($0)
add $4, $0, $0
lw    $2, 4($0)
```

**Memória de dados**

| <b>Address</b> | <b>Value</b> |
|----------------|--------------|
| 0x00000000     | 0x10         |
| 0x00000004     | 0x20         |

**loop:**

```
lw    $3, 0($1)
add $4, $4, $3
sw    $4, 36($1)
addiu $1, $1, 4
sltu $5, $1, $2
bne  $5, $0, loop
sw    $4, 8($0)
lw    $1, 12($0)
```



# Exercício

- Calcule o número de ciclos de relógio que o programa seguinte demora a executar, desde o *Instruction Fetch* da 1<sup>a</sup> instrução até à conclusão da última instrução:
  - 1) num datapath single-cycle, 2) num datapath multi-cycle

```
main:          # p0 = 0;
    lw    $1,0($0)   # p1 = *p0 = 0x10;
    add   $4,$0,$0   # v = 0;
    lw    $2,4($0)   # p2=*(p0+1)=0x20;
loop:          # do {
    lw    $3,0($1)   # aux1 = *p1;
    add   $4,$4,$3   # v = v + *p1;
    sw    $4,36($1)   # *(p1 + 9) = v;
    addiu $1,$1,4    # p1++;
    sltu  $5,$1,$2    #
    bne   $5,$0,loop # } while(p1 < p2);
    sw    $4,8($0)   # *(p0 + 2) = v;
    lw    $1,12($0)   # aux2 = *(p0 + 3);
```

| Memória de dados |       |
|------------------|-------|
| Address          | Value |
| 0x00000000       | 0x10  |
| 0x00000004       | 0x20  |



## Aulas 18, 19 e 20

- Instanciação de vários módulos do datapath recorrendo à linguagem de programa hardware VHDL:
  - Módulo de atualização do PC
  - Unidade de controlo do DP multi-cycle

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Módulo de atualização do PC – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity PCupdate is
    port(clk      : in  std_logic;
          reset     : in  std_logic;
          zero      : in  std_logic;
          PCSource: in  std_logic_vector(1 downto 0);
          PCWrite  : in  std_logic;
          PCWriteCond: in std_logic;
          PC4       : in  std_logic_vector(31 downto 0);
          BTA       : in  std_logic_vector(31 downto 0);
          jAddr    : in  std_logic_vector(25 downto 0);
          pc        : out std_logic_vector(31 downto 0));
end PCupdate;
```



# Módulo de atualização do PC – VHDL

```
architecture Behavioral of PCUpdate is
    signal s_pc : std_logic_vector(31 downto 0);
    signal s_pcEnable : std_logic;
begin
    s_pcEnable <= PCWrite or (PCWriteCond and zero);
    process(clk)
    begin
        if(rising_edge(clk)) then
            if(reset = '1') then
                s_pc <= (others => '0');
            elsif(s_pcEnable = '1') then
                case PCSource is
                    when "01" => -- BTA
                        s_pc <= BTA;
                    when "10" => -- JTA
                        s_pc <= s_pc(31 downto 28) & jAddr & "00";
                    when others => -- PC + 4
                        s_pc <= PC4;
                end case;
            end if;
        end if;
    end process;
    pc <= s_pc;
end Behavioral;
```



# A unidade de controlo do *datapath Multi-cycle* - VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ControlUnit is
    port( Clock      : in std_logic;
          Reset      : in std_logic;
          OpCode     : in std_logic_vector(5 downto 0);
          PCWrite    : out std_logic;
          IRWrite    : out std_logic;
          IorD       : out std_logic;
          PCSource   : out std_logic_vector(1 downto 0);
          RegDest    : out std_logic;
          PCWriteCond : out std_logic;
          MemRead    : out std_logic;
          MemWrite   : out std_logic;
          MemToReg   : out std_logic;
          ALUSelA    : out std_logic;
          ALUSelB    : out std_logic_vector(1 downto 0);
          RegWrite   : out std_logic;
          ALUop      : out std_logic_vector(1 downto 0));
end ControlUnit;
```



# A unidade de controlo do *datapath Multi-cycle* - VHDL

```
architecture Behavioral of ControlUnit is
    type TState is ( E0, E1, E2, E3, E4, E5, E6 , E7, E8, E9,
                      E10, E11);
    signal CS, NS : TState;
begin
    -- processo síncrono da máquina de estados (ME)
    process(Clock) is
    begin
        if(rising_edge(Clock)) then
            if(Reset = '1') then
                CS <= E0;
            else
                CS <= NS;
            end if;
        end if;
    end process;
    -- processo combinatório da ME na próxima página
end Behavioral;
```



```

process(CS, OpCode) is
begin
    PCWrite <= '0'; IRWrite <= '0'; IorD <= '0'; RegDest <= '0';
    PCWriteCond <= '0'; MemRead <= '0'; MemWrite <= '0'; MemToReg <= '0';
    RegWrite <= '0'; PCSource <= "00"; ALUop <= "00"; ALUSelA <= '0';
    ALUSelB <= "00";
    NS <= CS;
    case CS is
        when E0 =>
            MemRead <= '1'; PCWrite <= '1'; IRWrite <= '1'; ALUSelB <= "01";
            NS <= E1;
        when E1 =>
            ALUSelB <= "11";
            if(OpCode = "000000") then NS <= E6;      -- R-Type instructions
            elsif(OpCode = "100011" or OpCode = "101011" or
                  OpCode = "001000") then                -- LW, SW, ADDI
                NS <= E2;
            elsif(OpCode = "001010") then NS <= E8; -- SLTI
            elsif(OpCode = "000100") then NS <= E10;-- BEQ
            elsif(OpCode = "000010") then NS <= E11;-- J
            end if;
        when E6 =>      -- R-Type instructions
            ALUSelA <= '1'; ALUop <= "10";
            NS <= E7;
        when E7 =>      -- R-Type instructions
            RegWrite <= '1'; RegDest <= '1';
            NS <= E0;
            -- (...)

        end case;
    end process;

```

Processo combinatório



# Aulas 19, 20, 21 e 22

- *Pipelining*
  - Definição - exemplo prático por analogia
  - Adaptação do conceito ao caso do MIPS
  - Problemas da solução *pipelined*
- Construção de um *datapath* com *pipelining*
  - Divisão em fases de execução
  - Execução das instruções
- *Pipelining hazards*
  - *Hazards* estruturais: replicação de recursos
  - *Hazards* de controlo: *stalling*, previsão, *delayed branch*
  - *Hazards* de dados: *stalling*, *forwarding*
- *Datapath* para o MIPS com unidades de *forwarding* e *stalling*

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



# Introdução

- **Pipelining** é uma técnica de implementação de arquiteturas do set de instruções (ISA), através da qual múltiplas instruções são executadas com algum grau de **sobreposição temporal**
- O objetivo é aproveitar, de forma o mais eficiente possível, os recursos disponibilizados pelo *datapath*, por forma a **maximizar a eficiência global do processador**



## *Pipelining - exemplo por analogia*

- O exemplo de *pipelining* que iremos observar de seguida apoia-se num conjunto de tarefas simples e intuitivas: o processo de tratamento da roupa suja ☺



- Neste exemplo, o tratamento da roupa suja desencadeia-se nas seguintes quatro fases:
  1. Lavar uma carga de roupa na máquina respetiva
  2. Secar a roupa lavada na máquina de secar
  3. Passar a ferro e dobrar a roupa
  4. Arrumar a roupa dobrada no guarda roupa respetivo

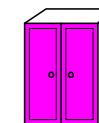
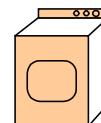


## *Pipelining - exemplo por analogia*

- Numa versão não *pipelined*, o processamento de N cargas de roupa seria:

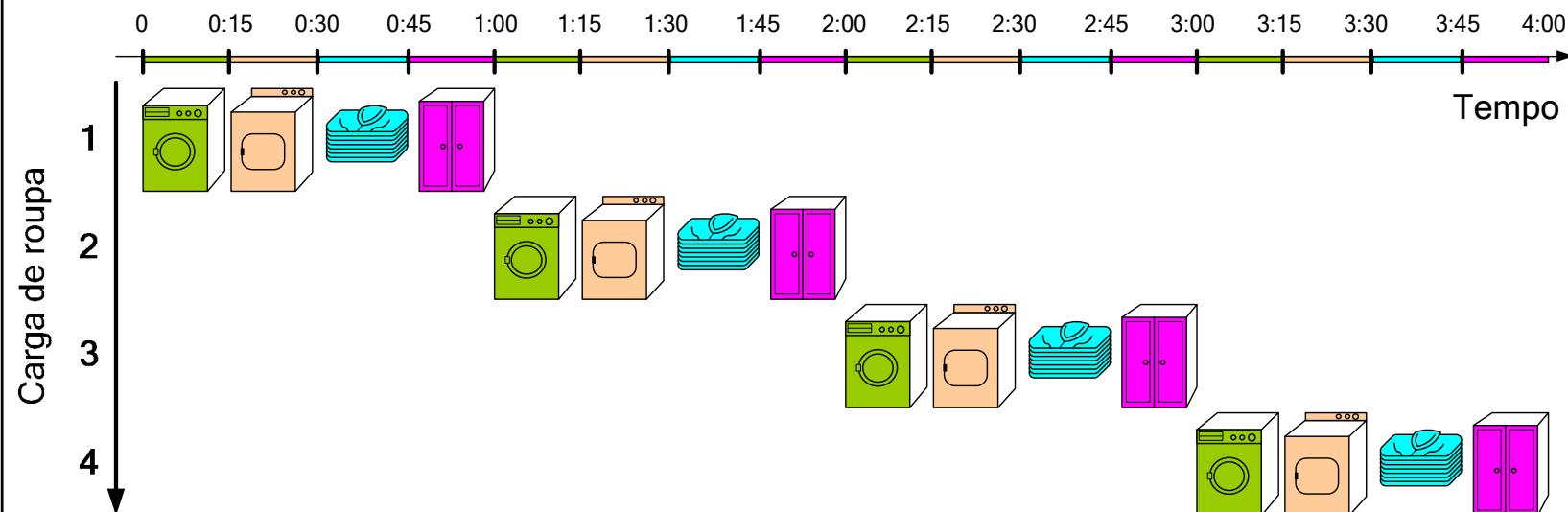


carga #n



# Pipelining - exemplo por analogia

- Este processo pode então ser descrito temporalmente do seguinte modo:



Se o tempo para tratar uma carga de roupa for uma hora,  
tratar quatro cargas demorará **quatro horas**.



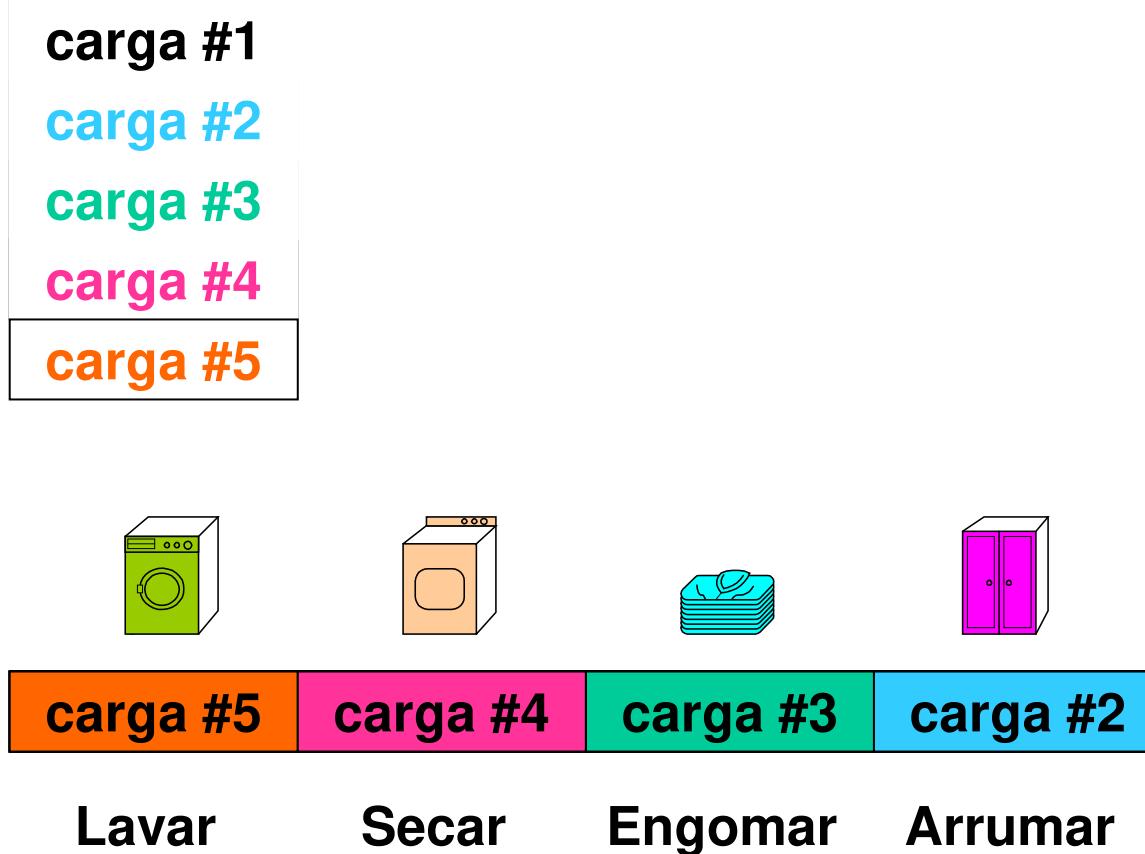
## *Pipelining* - exemplo por analogia

- Na versão *pipelined*, aproveita-se para carregar uma nova carga de roupa na máquina de lavar mal esteja concluída a lavagem da primeira carga
- O mesmo princípio se aplica a cada uma das restantes três tarefas
- Quando se inicia a arrumação da primeira carga, todos os passos (chamados **estágios** ou **fases** em *pipelining*) estão a funcionar em paralelo
- Maximiza-se assim a utilização dos recursos disponíveis



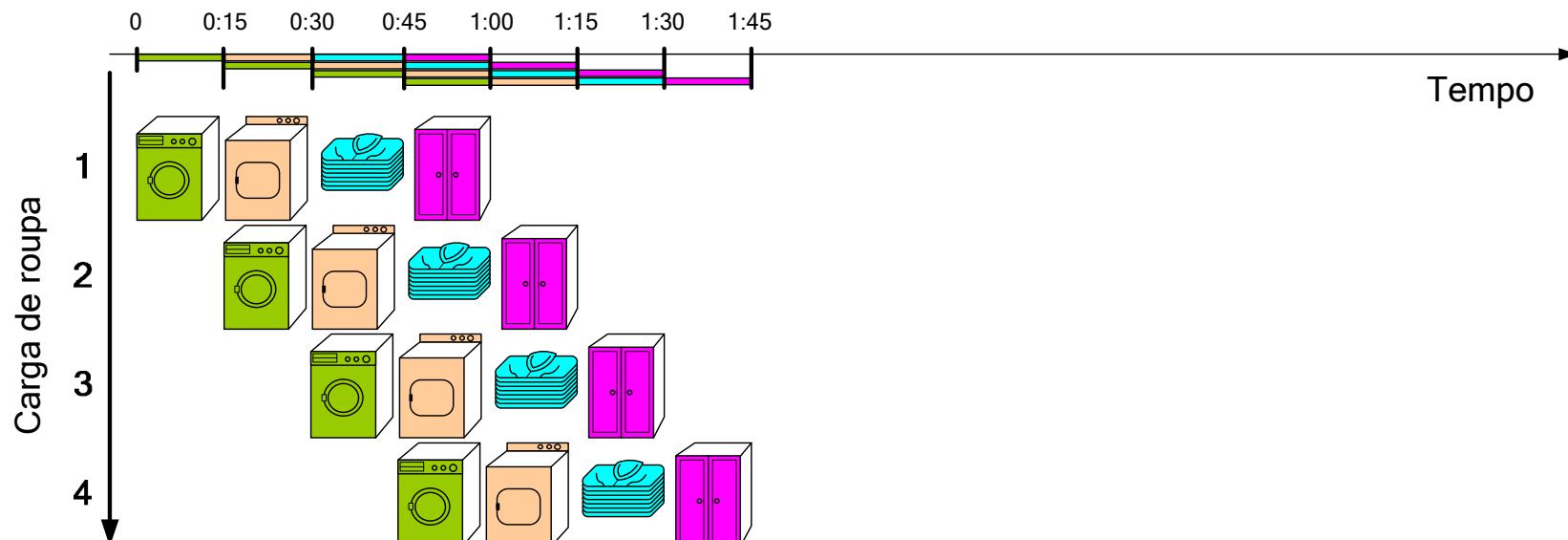
## *Pipelining - exemplo por analogia*

- Na versão *pipelined*, o processamento das cargas de roupa seria (admitindo tempo nulo entre a comutação de tarefas):



# Pipelining - exemplo por analogia

- O processo de tratamento da versão *pipelined* pode então ser descrito temporalmente do seguinte modo:



Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45. Ou seja 135 minutos menos ( $240 - 105$ ).



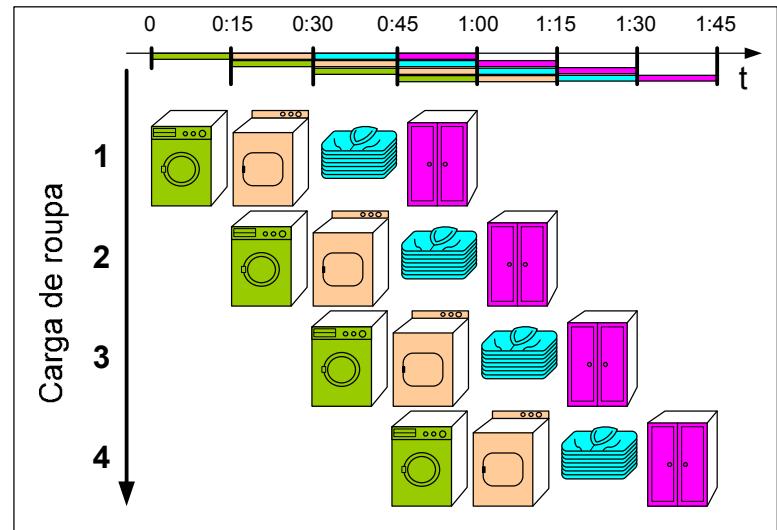
## *Pipelining* - exemplo por analogia

- O paradoxo aparente da solução *pipelined* é que o tempo necessário para o processamento completo de uma carga de roupa não difere do tempo de execução da solução não *pipelined*
- A eficiência da solução com *pipelining* decorre do facto de, para um número grande de cargas de roupa, todos os passos intermédios estarem a executar em paralelo
- O resultado é o aumento do número total de cargas de roupa processadas por unidade de tempo (***throughput***)
- Qual o **ganho de desempenho** que se obtém com o sistema *pipelined* relativamente ao sistema normal?



# Pipelining – ganho de desempenho

- O tratamento de  $N$  cargas de roupa num sistema com  $F$  fases demorará idealmente (admitindo que cada fase demora 1 unidade de tempo):



Sistema não *pipelined*:  $T_{\text{NON-PIPELINE}} = N \times F$

Sistema *pipelined*:  $T_{\text{PIPELINE}} = F + (N - 1) = (F - 1) + N$

Ganho obtido com a solução *pipelined*:

$$\frac{\text{Desempenho}_{\text{PIPELINE}}}{\text{Desempenho}_{\text{NON-PIPELINE}}} = \frac{T_{\text{NON-PIPELINE}}}{T_{\text{PIPELINE}}} = \frac{N \times F}{(F - 1) + N}$$

Se  $N \gg (F-1)$ , então:

$$\text{Ganho} \approx \frac{N \times F}{N} = F$$



# Pipelining – ganho de desempenho

- No limite, para um número de cargas de roupa muito elevado, o ganho de desempenho (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e outro modelo) é da ordem do **número de tarefas realizadas em paralelo** (isto é, igual ao número de fases do processo)
- Genericamente, poderíamos afirmar que o ganho em velocidade de execução é igual ao número de estágios do *pipeline* ( $F$ )
- No exemplo observado, o limite teórico estabelece que a solução *pipelined* é quatro vezes mais rápida do que a solução não *pipelined*
- A adopção de *pipelines* muito longos (com muitos estágios) pode, contudo, como veremos mais tarde, limitar drasticamente a eficiência global

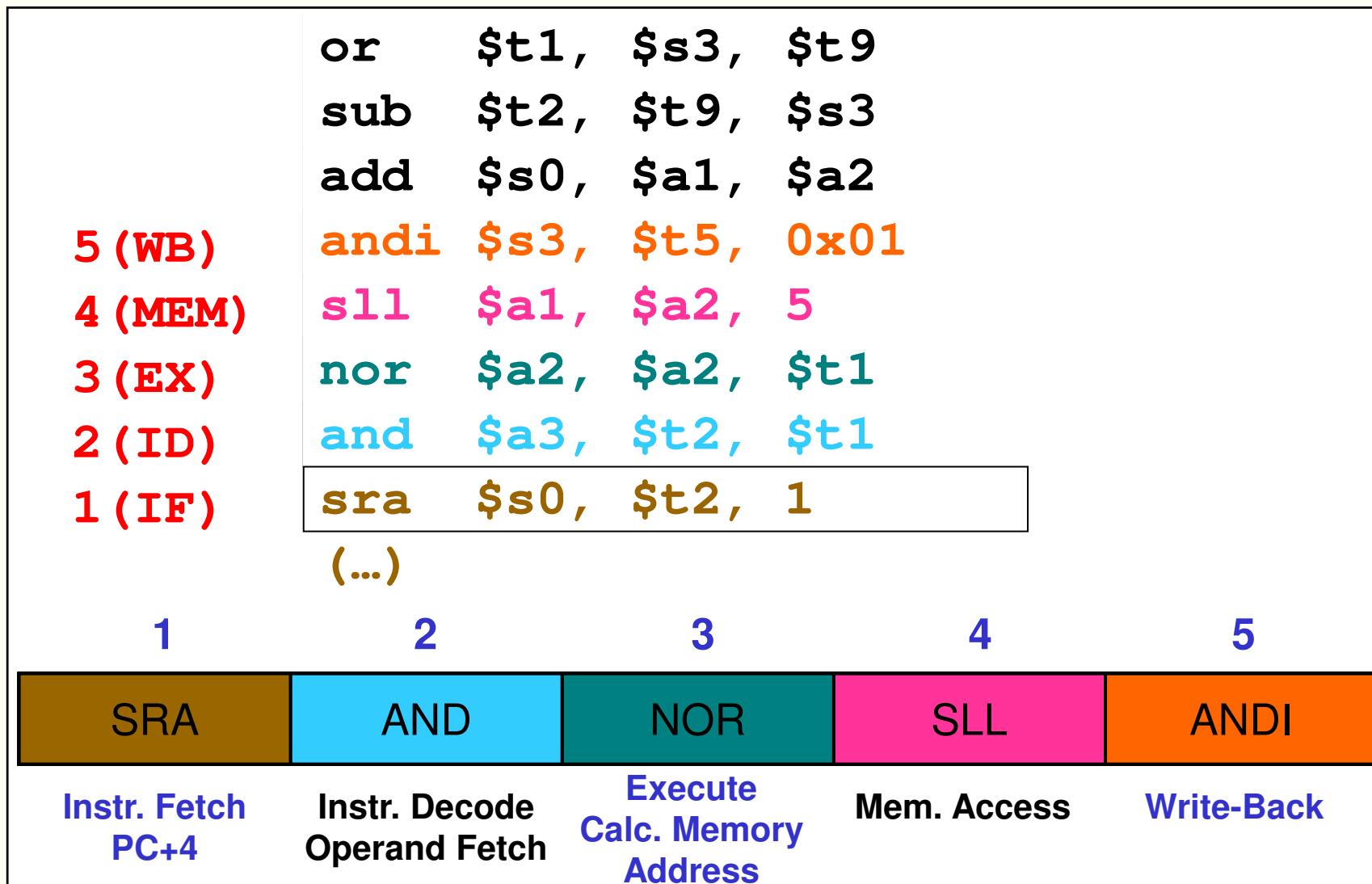


# Pipelining no MIPS

- Os mesmos princípios que observámos para o caso do tratamento da roupa, podem igualmente ser aplicados aos processadores
- No caso do MIPS, como já sabemos, as instruções podem ser divididas genericamente em **cinco fases** (estágios, etapas):
  1. **Instruction fetch** (ler a instrução da memória), incremento do PC
  2. **Operand fetch** (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
  3. **Execute** (executar a operação ou calcular um endereço)
  4. **Memory access** (aceder à memória de dados para leitura ou escrita)
  5. **Write-Back** (escrever o resultado no registo destino)
- Parece assim razoável admitir a construção de uma solução *pipelined* do *datapath* do MIPS que implemente cinco estágios distintos, um para cada fase da execução das instruções



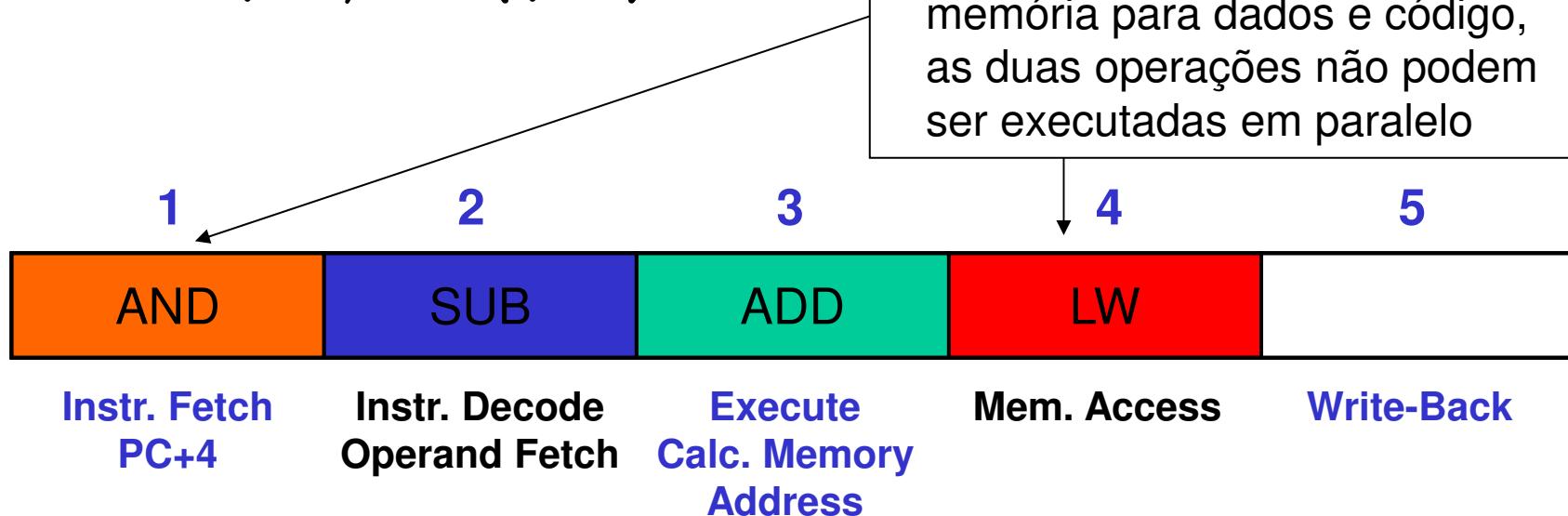
## *Pipelining no MIPS*



# Pipelining – Problemas (exemplo 1)

```
lw    $t1, 0($t9)
add $t2, $t3, $t4
sub $t3, $t4, $t5
and $t4, $t5, $t6
lw    $t5, 16($t9)
```

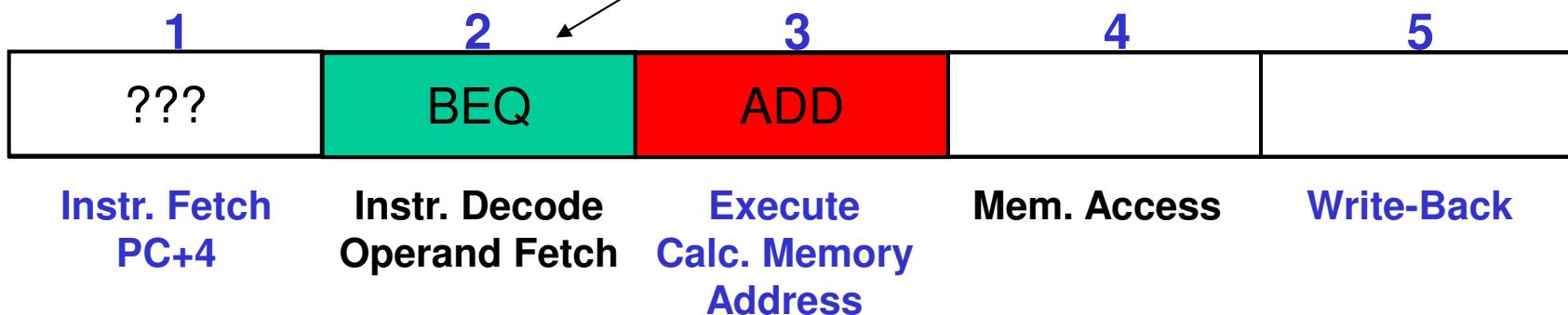
- No quarto estágio da primeira instrução e no primeiro da quarta instrução é necessário efetuar, simultaneamente, um acesso à memória para **leitura de dados** e para o **instruction fetch**
- Se existir apenas uma memória para dados e código, as duas operações não podem ser executadas em paralelo



## Pipelining – Problemas (exemplo 2)

```
add $t4, $t5, $t6  
beq $t1, $t2, Z1  
lw $s3, 300($a0)  
(...)  
Z1: or $t7, $t8, $t9
```

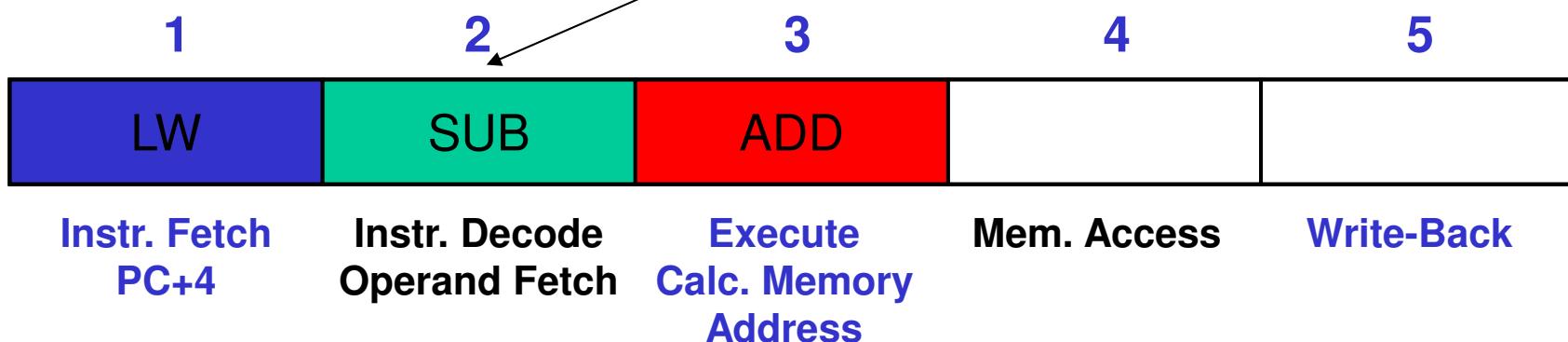
- Qual a próxima instrução a ler da memória (**LW / OR**) assumindo que a decisão do *branch* só é tomada no 3º estágio do *pipeline*?
- Mesmo admitindo que existe h/w dedicado para avaliar a condição do *branch* logo no 2º estágio, a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória.



## Pipelining – Problemas (exemplo 3)

add \$s0, \$t0, \$t1  
sub \$t2, \$s0, \$t3  
lw \$t4, 0(\$s2)

A instrução de subtração não pode avançar para o estágio seguinte uma vez que o seu operando, **\$s0**, ainda não foi calculado e armazenado no registo destino pela instrução anterior.



# Datapath pipelined para o MIPS

- Para tornar a discussão mais concreta, vamos construir um *datapath* que implemente um *pipeline*, que suporte as instruções que já considerámos anteriormente, isto é:
  - acesso à memória: load word (**lw**) e store word (**sw**)
  - instruções tipo R: **add**, **sub**, **and**, **or** e **slt**
  - Instruções imediatas: **addi** e **slti**
  - branch if equal (**beq**)
- Começamos por comparar os tempos necessários à execução destas instruções num *datapath single cycle* e num *datapath pipelined*, tomando como referência os seguintes tempos de execução de cada um das fases:

| Instruction                     | Instruction Fetch | Register Read | ALU Operation | Memory Access | Register Write |
|---------------------------------|-------------------|---------------|---------------|---------------|----------------|
| Load word (lw)                  | 2 ns              | 1 ns          | 2 ns          | 2 ns          | 1 ns           |
| Store word (sw)                 | 2 ns              | 1 ns          | 2 ns          | 2 ns          |                |
| R-Type (add, sub, and, or, slt) | 2 ns              | 1 ns          | 2 ns          |               | 1 ns           |
| Branch (beq)                    | 2 ns              | 1 ns          | 2 ns          |               |                |
| Immediate (addi, slti)          | 2 ns              | 1 ns          | 2 ns          |               | 1 ns           |

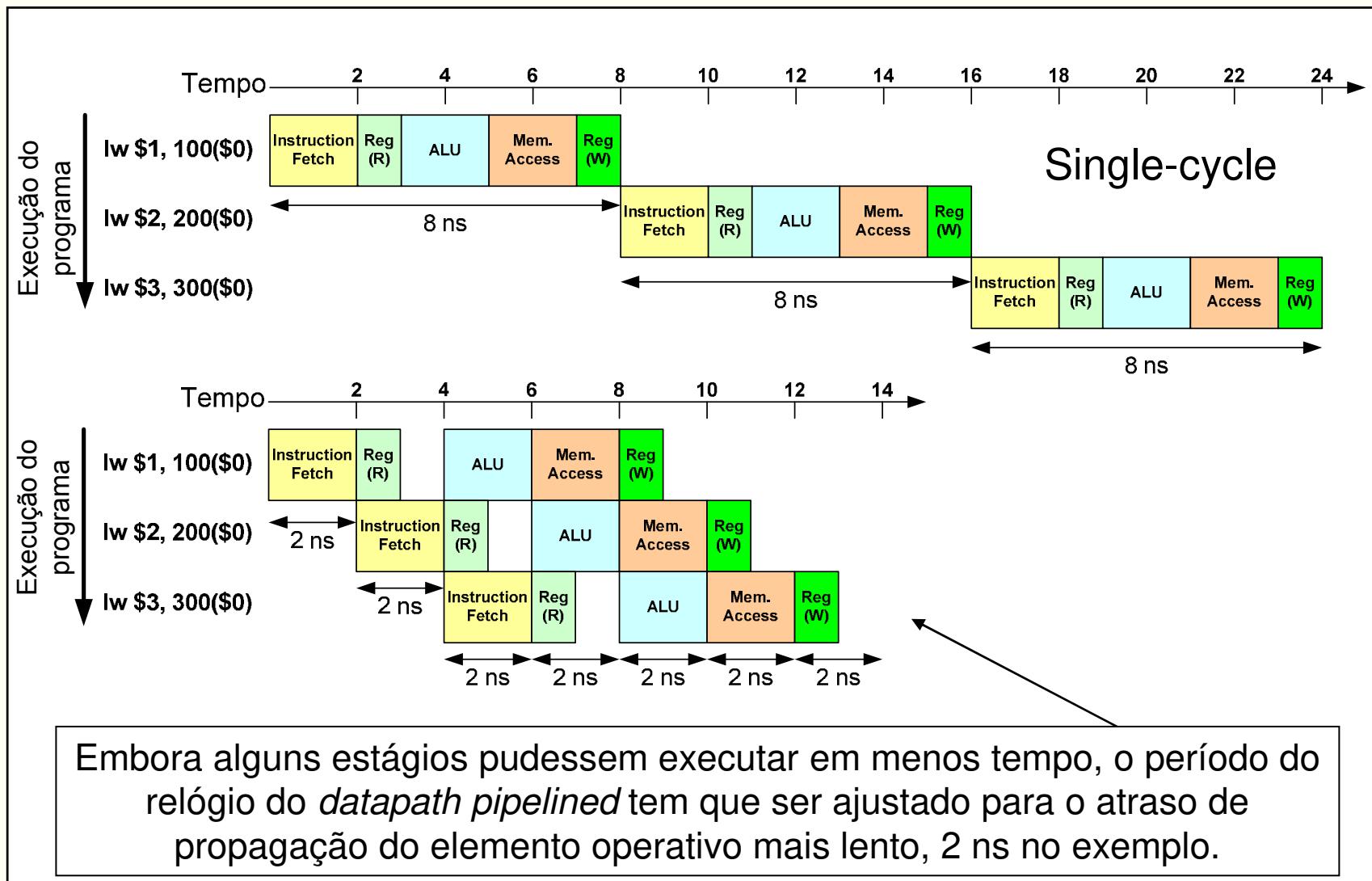


## *Datapath pipelined para o MIPS*

- De acordo com a tabela fornecida, e para a solução *single cycle*, teremos que ajustar o período do relógio ao tempo necessário para executar a instrução mais lenta (*lw*)
- Ou seja, na solução *single cycle* todas as instruções, independentemente do tempo mínimo que poderiam durar, serão executadas num tempo de 8ns
- Para verificarmos como comparar o tempo de execução de um trecho de código por cada uma das soluções (*pipelined* e não *pipelined*), observemos o exemplo do slide seguinte



# Datapath pipelined para o MIPS



# Datapath pipelined para o MIPS

- O *instruction set* do **MIPS** (**Microprocessor without Interlocked Pipeline Stages**) foi concebido para uma implementação em *pipeline*. Os aspetos fundamentais a considerar são:
  - **Instruções de comprimento fixo.** *Instruction Fetch* e *Instruction Decode* podem ser feitos em estágios sucessivos uma vez que a unidade de controlo não tem que se preocupar com a dimensão da instrução descodificada
  - **Poucos formatos de instrução**, com a referência aos registos a ler sempre no mesmo campo. Isto permite que os registos sejam lidos no segundo estágio ao mesmo tempo que a instrução é descodificada pela unidade de controlo
  - **Referências à memória só aparecem em instruções de load/store.** O terceiro estágio pode assim ser usado para executar a instrução ou para calcular o endereço de memória, permitindo o acesso à memória no estágio seguinte
  - Os **operандos em memória têm que estar alinhados**. Desta forma qualquer operação de leitura/escrita da memória pode ser feita num único estágio

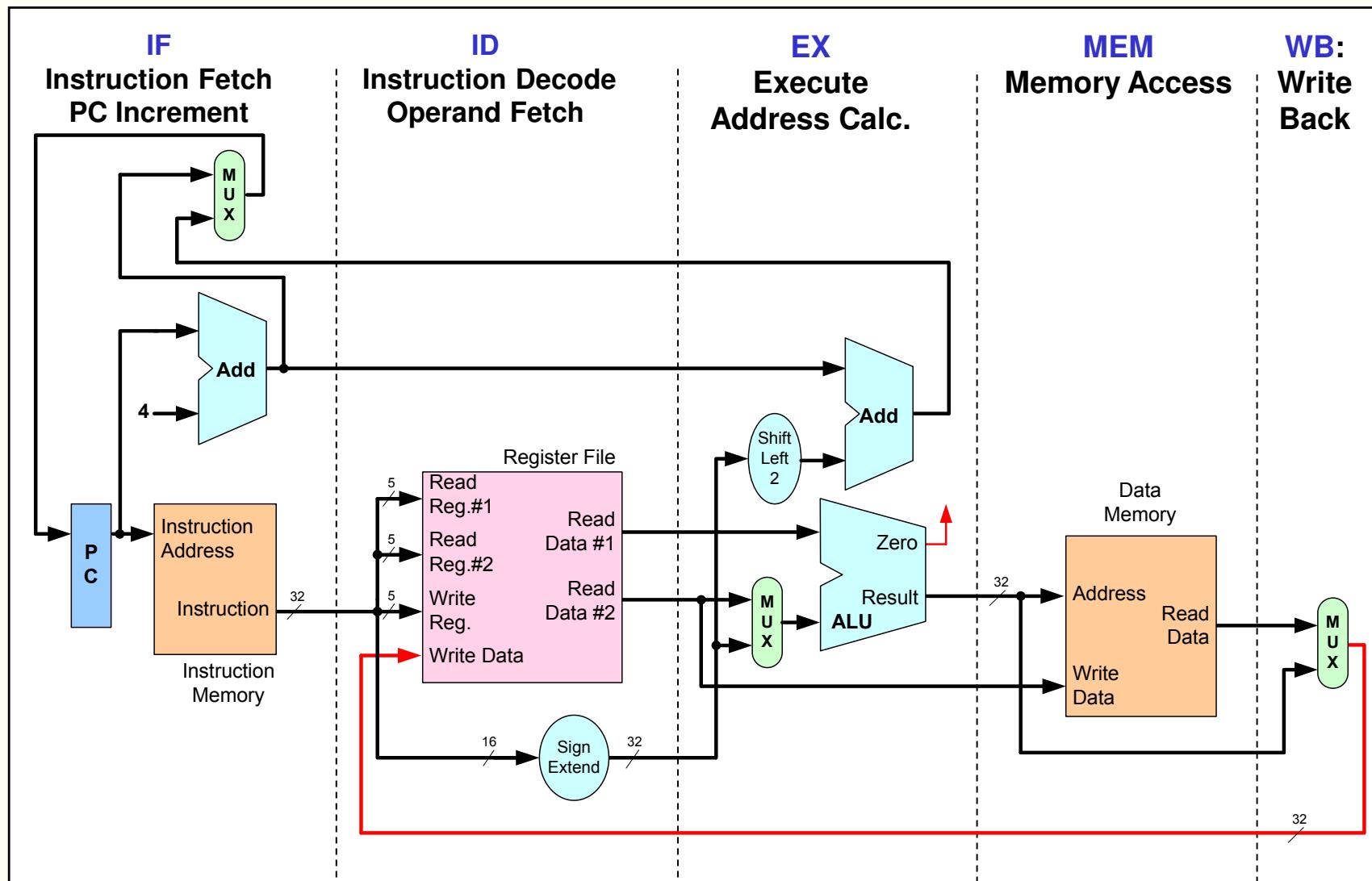


# Datapath pipelined para o MIPS

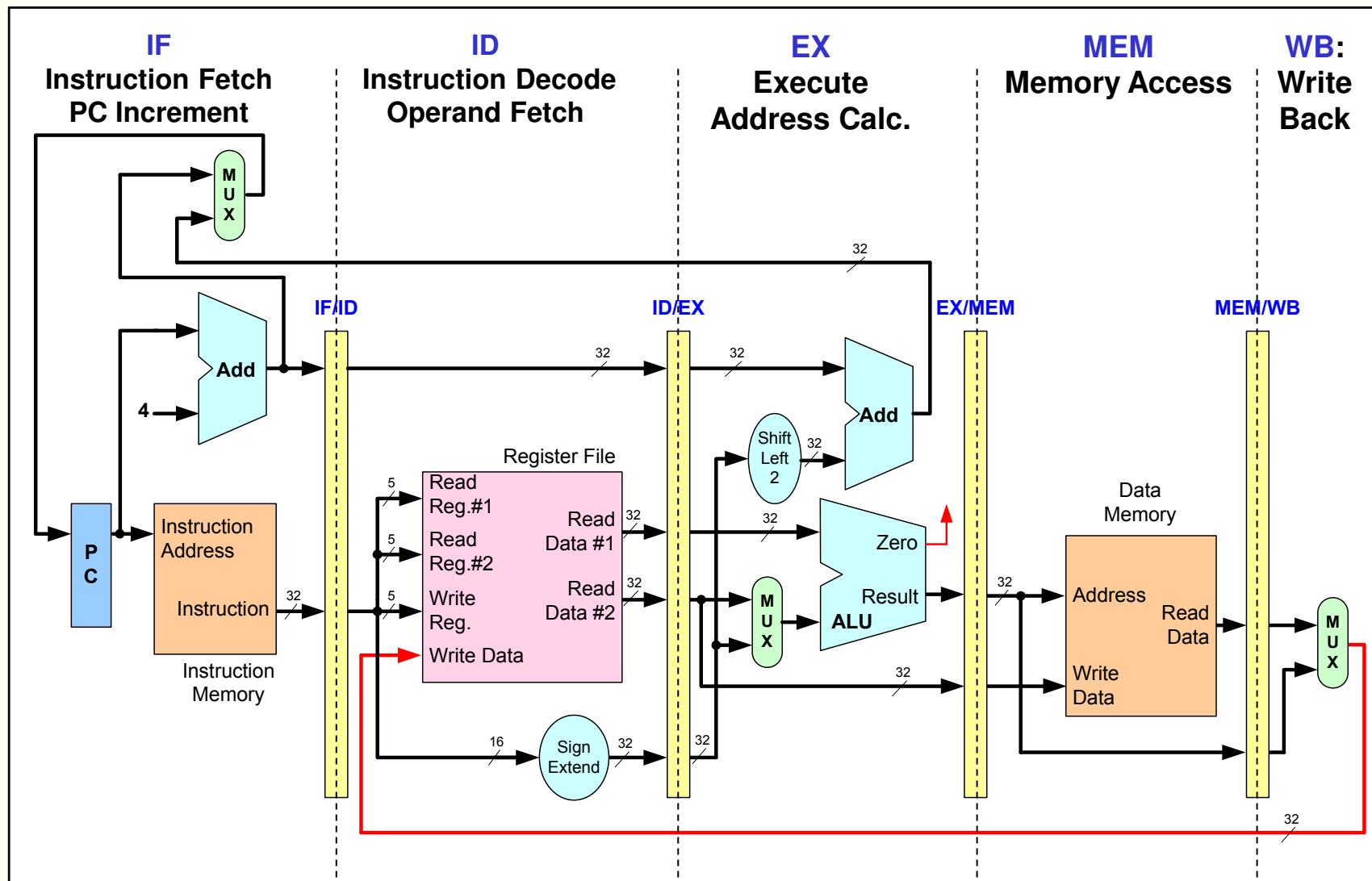
- A solução *pipelined* para o MIPS parte do modelo do *datapath single-cycle*
- A organização implementa as cinco fases sequenciais em que são decomponíveis as instruções:
  1. (**IF**) - *Instruction fetch* (ler a instrução da memória), incremento do PC
  2. (**ID**) - *Operand fetch* (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
  3. (**EX**) - Executar a operação ou calcular um endereço
  4. (**MEM**) - *Memory access* (aceder à memória de dados para leitura ou escrita)
  5. (**WB**) - *Write-back* (escrever o resultado no registo destino)
- Na solução apresentada no slide seguinte não são identificados os sinais de controlo nem a respetiva unidade de controlo



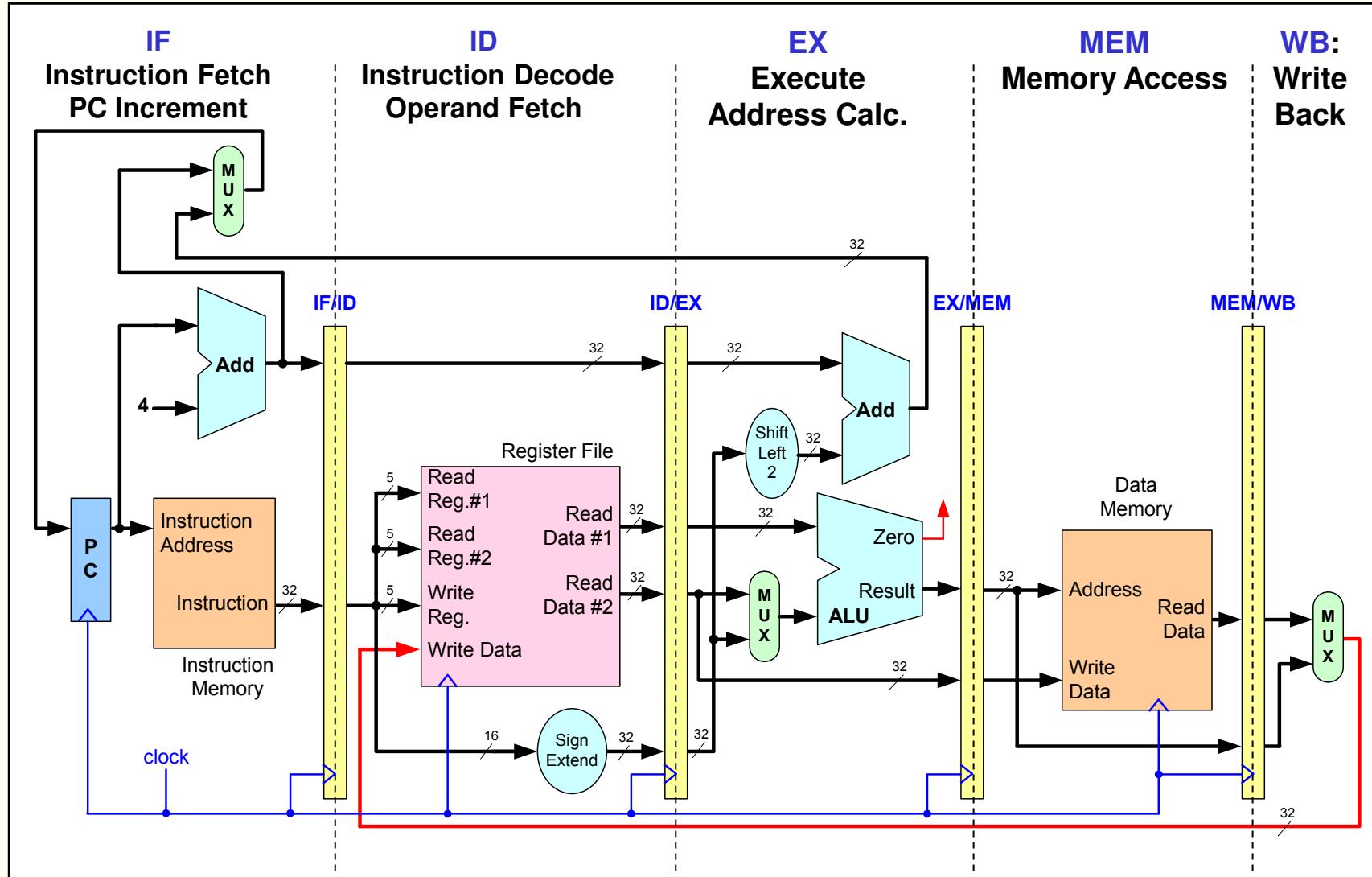
# Divisão em fases de execução



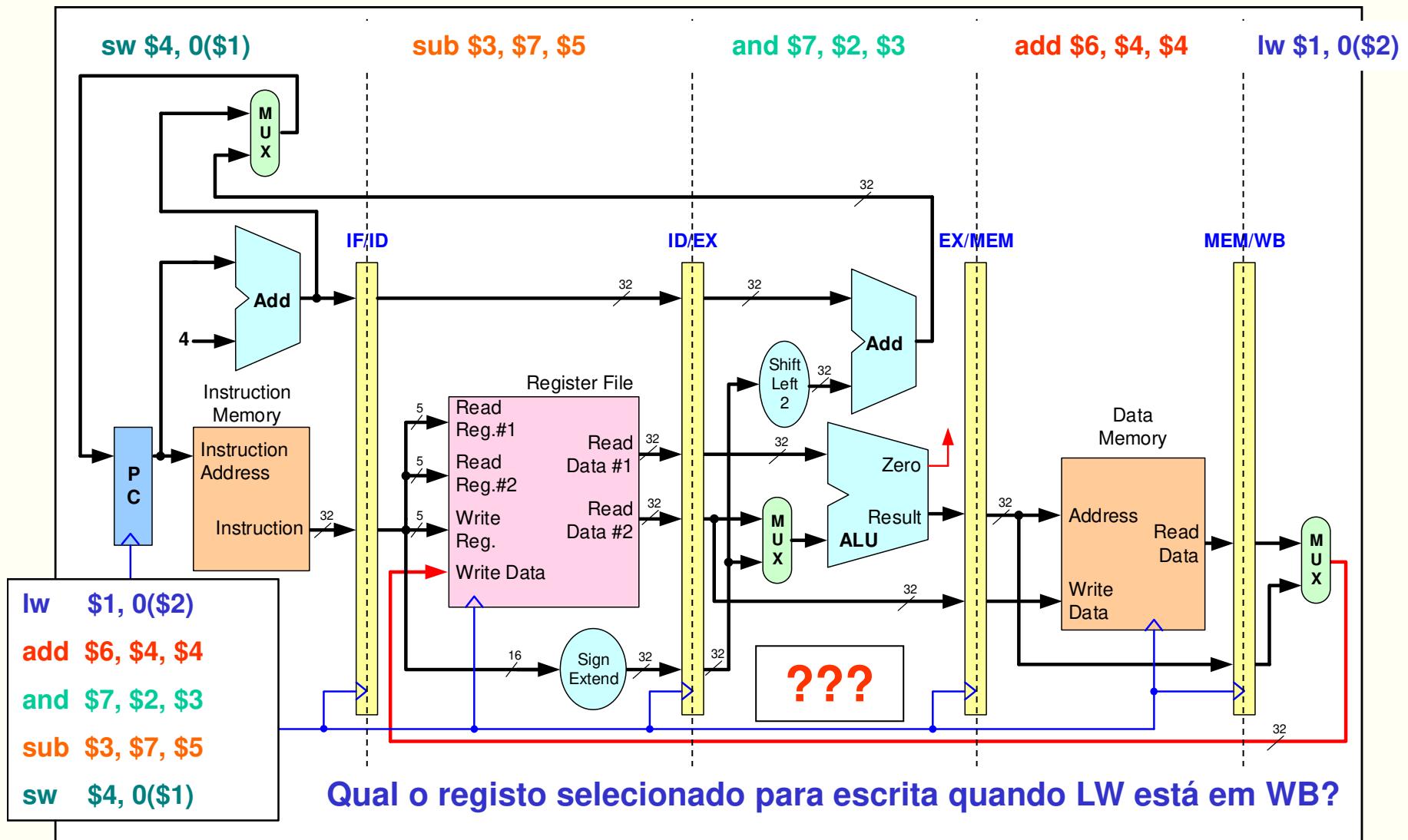
# Divisão em fases de execução



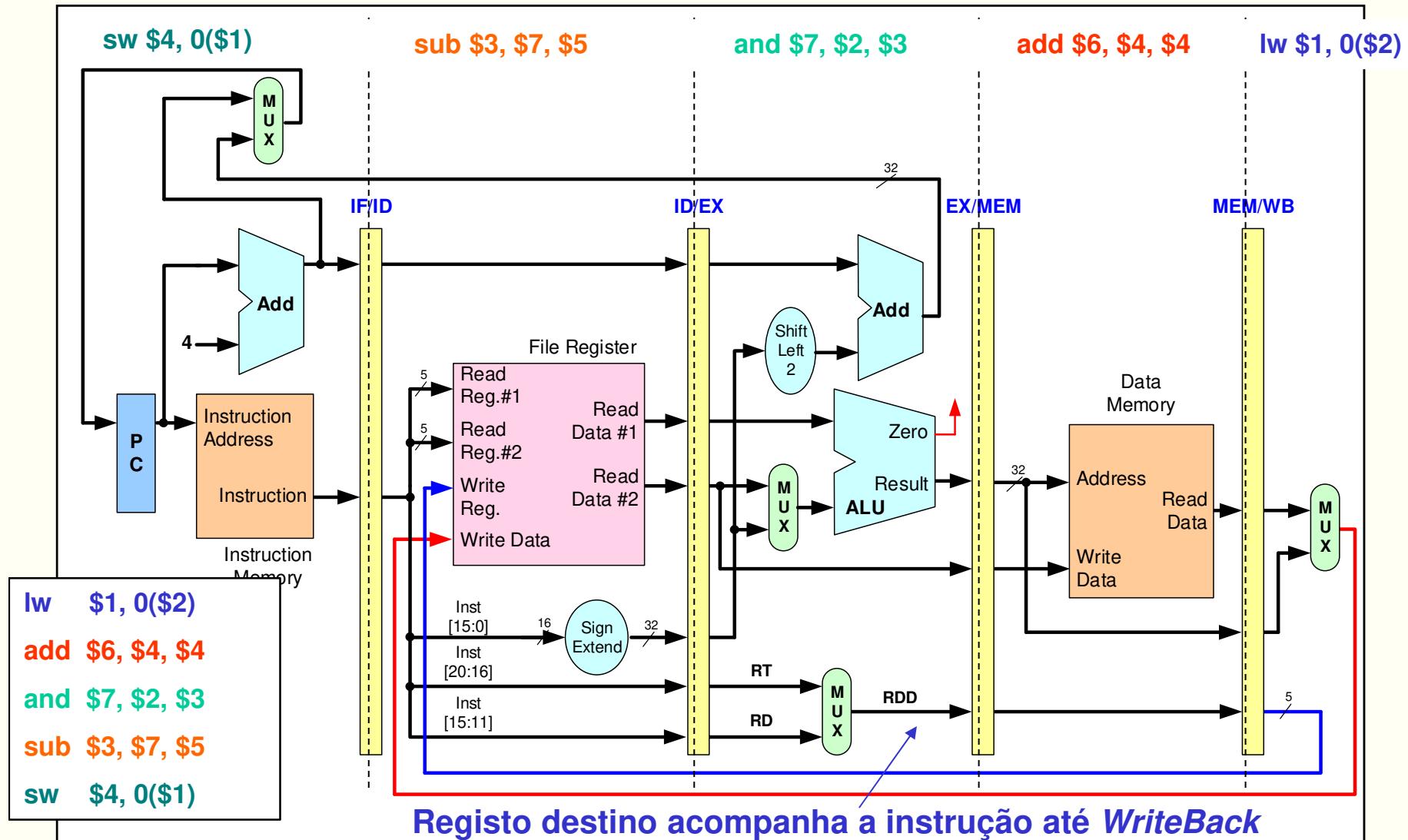
# Divisão em fases de execução (com o sinal de relógio)



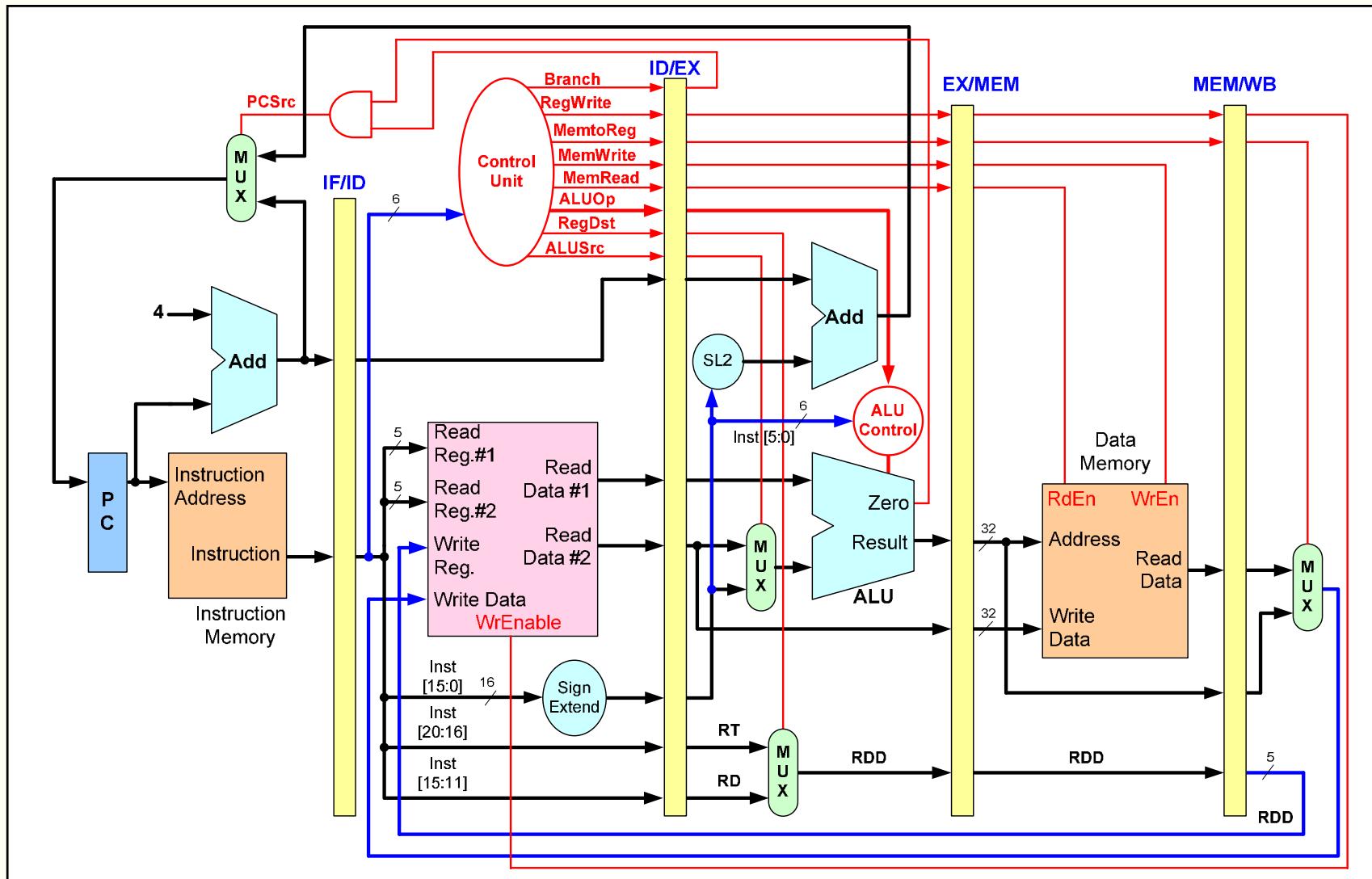
# Execução de instruções



# Datapath pipelined – 1<sup>a</sup> versão



# Unidade de controlo



# Unidade de controlo

- A implementação *pipeline* do MIPS usa os mesmos sinais de controlo da versão *single-cycle*
- A unidade de controlo é, assim, uma **unidade combinatória** que gera os sinais de controlo em função do código da instrução (6 bits mais significativos da instrução, i.e., *opcode*) presente na fase ID
- Os sinais de controlo relevantes avançam no *pipeline* a cada ciclo de relógio (assim como os dados) estando, portanto, sincronizados com a instrução
- O sinal **RegWrite** é propagado até *WriteBack* e daí controla a escrita no *Register File* (fase ID)
- O sinal **Branch** é propagado até à fase EX (nesta versão o *branch* é resolvido nessa fase)



# Exercício 1

- Determine o número de ciclos de relógio que o trecho de código seguinte demora a executar num *pipeline* de 5 fases, desde o instante em que é feito o *Instruction Fetch* da 1<sup>a</sup> instrução, até à conclusão da última:

```
add    $1, $2, $3
lw     $2, 0($4)
sub    $3, $4, $3
addi   $4, $4, 4
and    $5, $1, $5 # "and" em ID, "add" já terminou
sw     $2, 0($1) # "sw" em ID, "add" e "lw"
          # já terminaram
```

$$\begin{aligned}\text{Nr_Cycles} &= F + (\text{Number\_of\_executed\_instructions} - 1) \\ &= 5 + (6 - 1) = 10 \text{ T}\end{aligned}$$

Num *datapath single-cycle* o mesmo código demoraria 6 ciclos de relógio a executar. Então porque razão é a execução no *datapath pipelined* mais rápida?

Quantos ciclos de relógio demora a execução num *datapath multi-cycle*?



# Pipeline Hazards

- Existe um conjunto de situações particulares que podem condicionar a progressão das instruções no *pipeline* no próximo ciclo de relógio
- Estas situações são designadas genericamente por ***hazards***, e podem ser agrupadas em três classes distintas:
  - ***Hazards estruturais***
  - ***Hazards de controlo***
  - ***Hazards de dados***
- Nos próximos slides serão discutidas, para cada tipo de *hazard*, as origens e as consequências, mapeando depois esses aspectos ao nível da arquitetura *pipelined* do MIPS

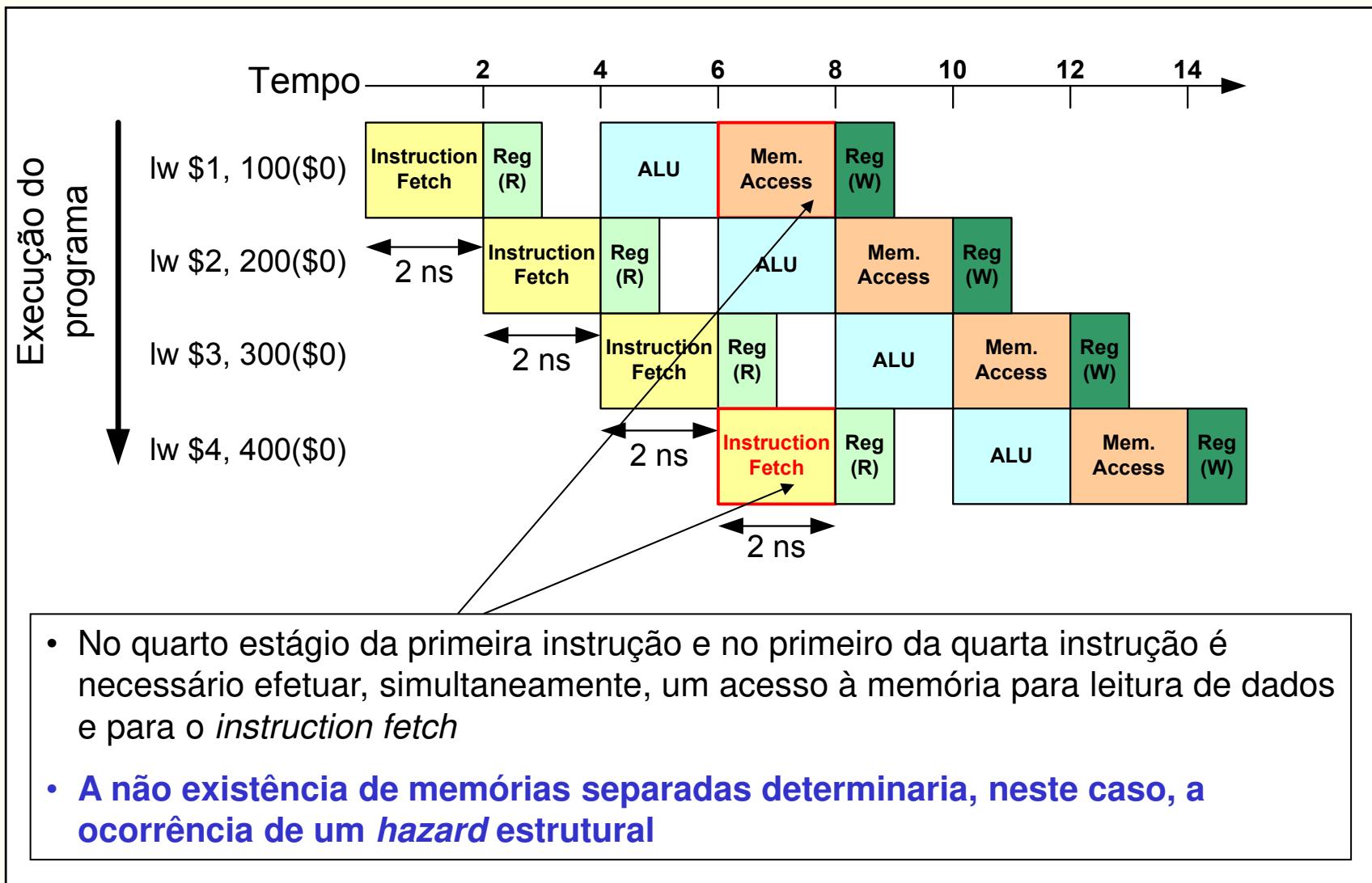


# Hazards Estruturais

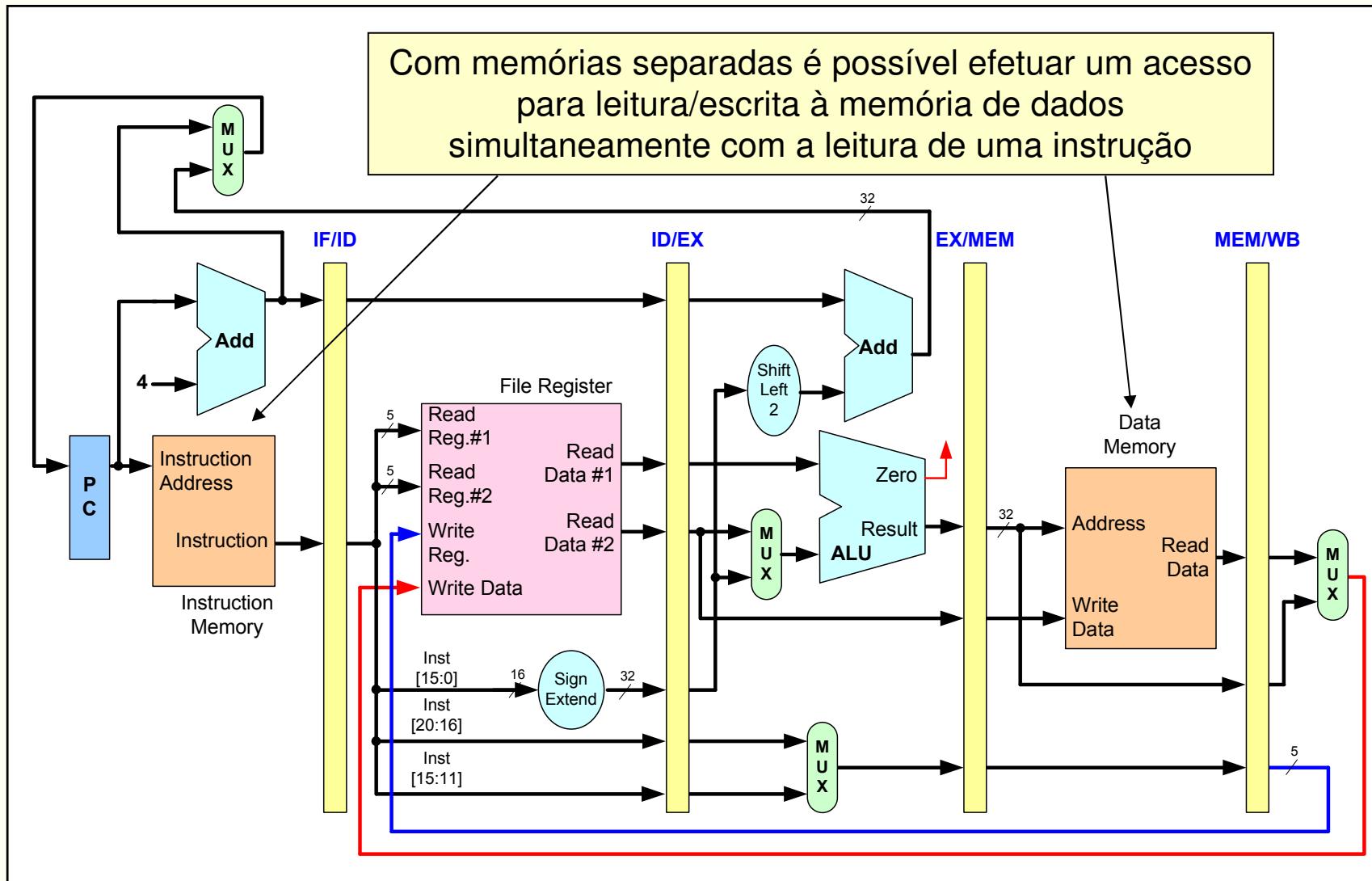
- Um **hazard estrutural** ocorre quando mais do que uma instrução necessita de aceder ao mesmo hardware
- Ocorre quando: 1) apenas existe uma memória ou 2) há instruções no *pipeline* com diferentes tempos de execução
- No primeiro caso o *hazard* estrutural é evitado duplicando a memória, i.e., uma memória de programa e uma memória de dados (acesso em IF não conflitua com possível acesso em MEM)
- O segundo caso está fora da análise feita nestes slides; como exemplo pode pensar-se na implementação de uma instrução mais complexa que demore 2 ciclos de relógio na fase EX



# Hazards Estruturais



# Hazards Estruturais



# Hazards de Controlo

- Um *hazard* de controlo ocorre quando é necessário fazer o *instruction fetch* de uma nova instrução e existe numa etapa mais avançada do *pipeline* uma instrução que pode alterar o fluxo de execução e que ainda não terminou
- No caso do MIPS, as situações de *hazard* de controlo surgem com as instruções de salto, (*jumps* e *branches*)
- Exemplo:

**beq \$5, \$6, next**

**add \$2, \$3, \$4**

...

**next: lw \$3, 0(\$4)**

...

Qual a instrução que deve entrar no *pipeline* a seguir à instrução "beq"?

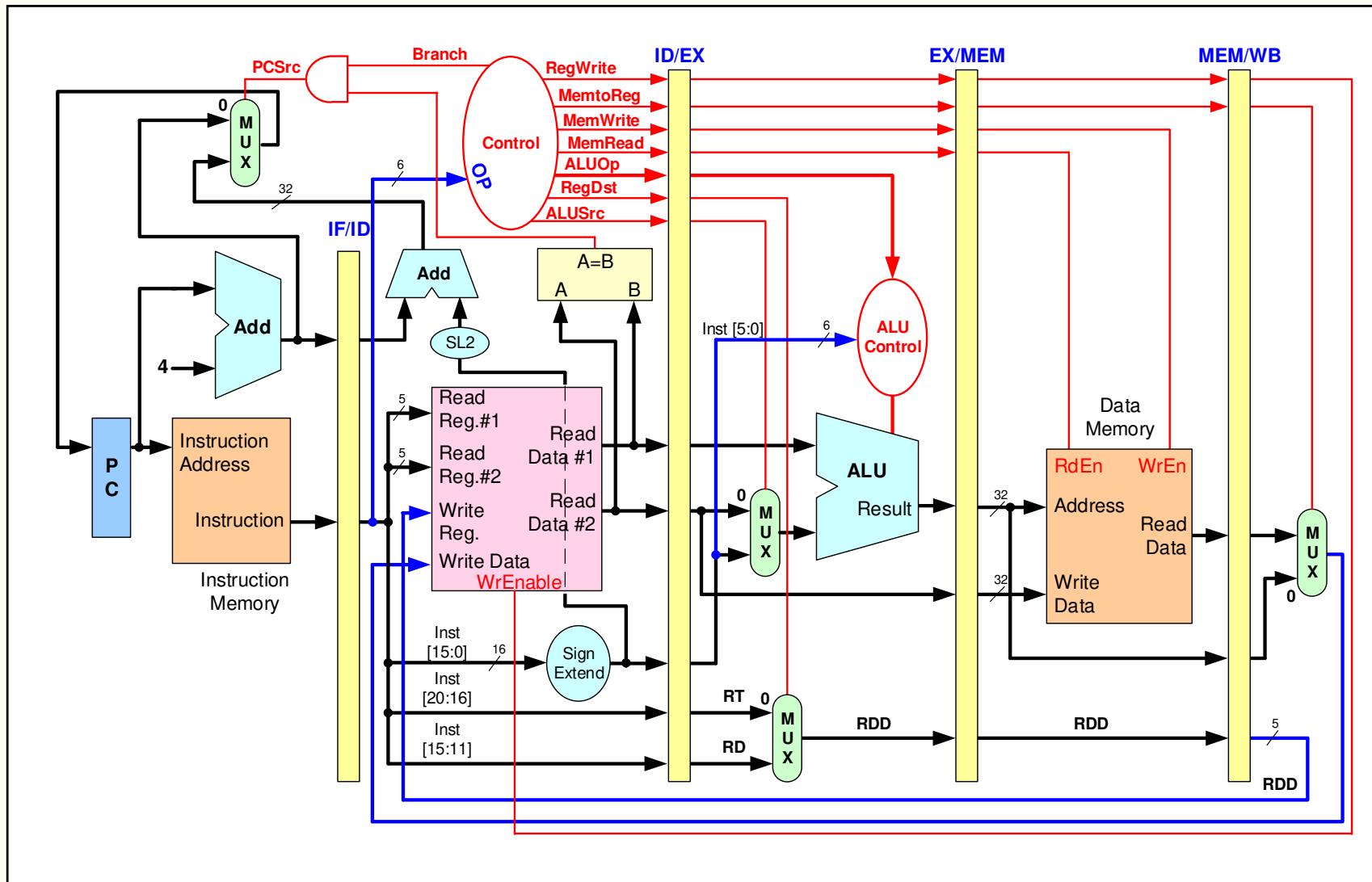


## Hazards de Controlo

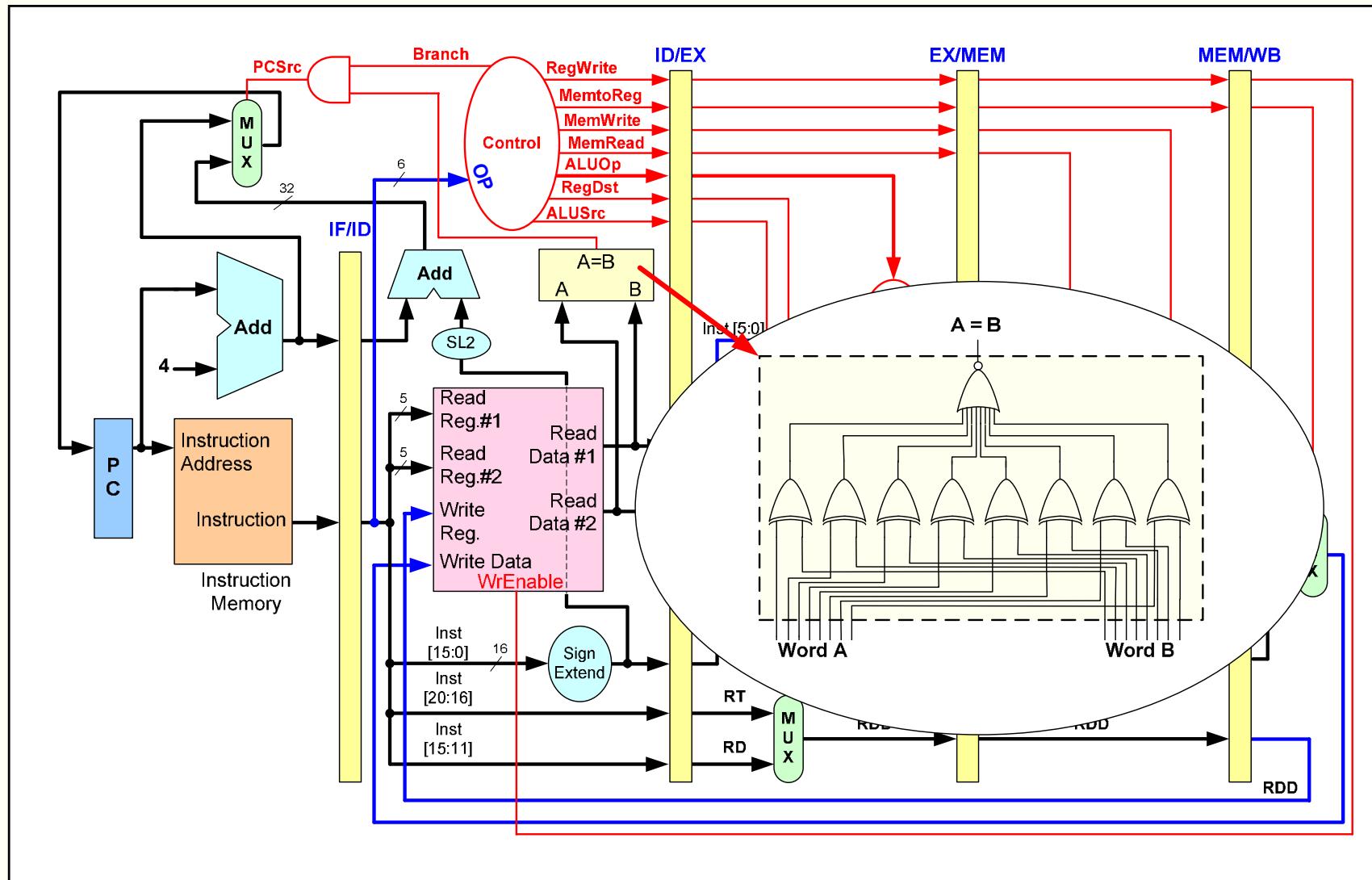
- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de código
- Na análise que se segue supõe-se que a **comparação dos operandos é efetuada no 2º estágio (ID)**, através de hardware adicional
- Do mesmo modo, **o cálculo do *Branch Target Address* passa também a ser efetuado em ID**



# Datapath com branches resolvidos em ID

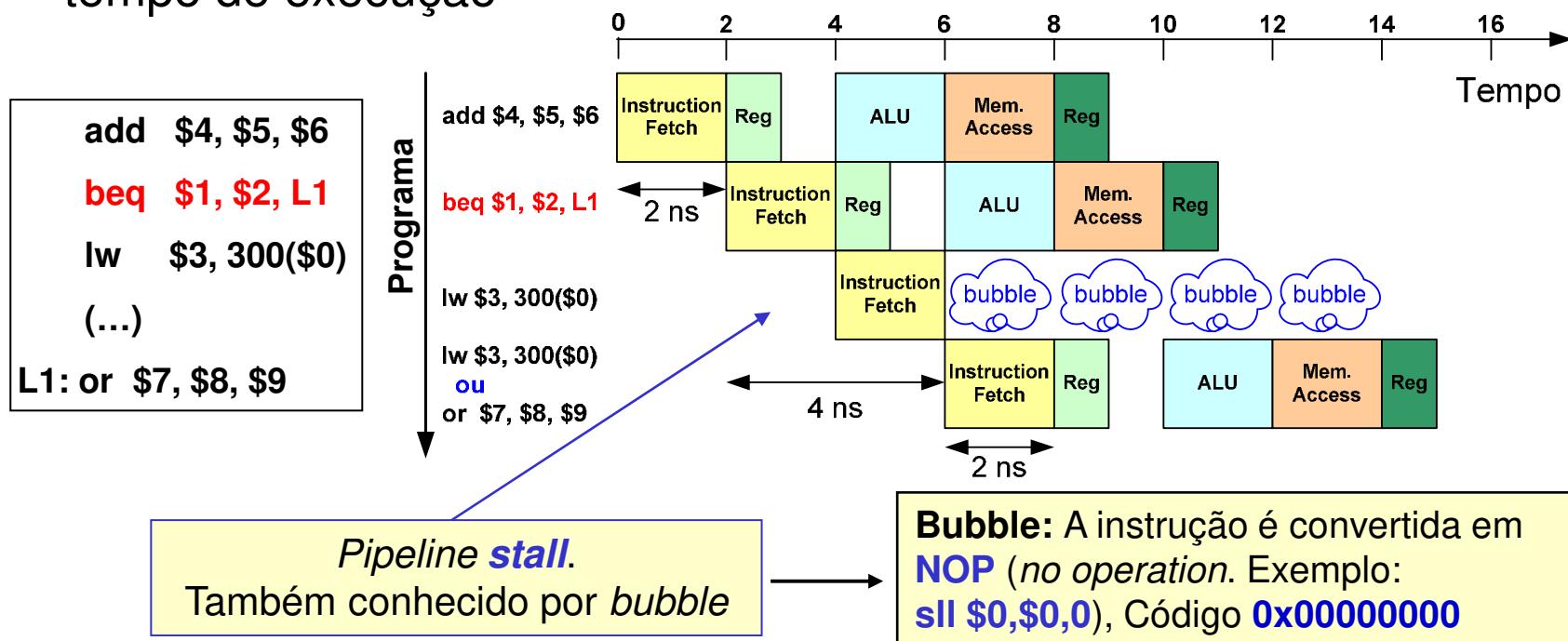


# Datapath com branches resolvidos em ID



# Hazards de controlo

- Há mais do que uma solução para lidar com os *hazards* de controlo. A primeira que vamos analisar é designada por *stalling* ("parar o progresso de...")
- Nesta estratégia a unidade de controlo atrasa a entrada no *pipeline* da próxima instrução até saber o resultado do *branch* condicional
- É uma solução conservativa que tem um preço em termos de tempo de execução



## Hazards de controlo - *Stalling*

- Se 15% das instruções de um dado programa forem *branches*, qual o efeito desta estratégia no desempenho da arquitetura, admitindo que os *branches* são resolvidos em ID?

Sem *stalls*: CPI = 1

Com *stalls*: CPI = 1 + 1 \* 0,15 = 1,15

Relação de desempenho = 1 / 1,15 = 0,87

- A degradação do desempenho é tanto maior quanto mais tarde for resolvida a instrução de *branch*. Na mesma situação, se o *branch* for resolvido em EX, a relação passa a ser:

Relação de desempenho = 1 / (1 + 2 \* 0,15) = 0,77

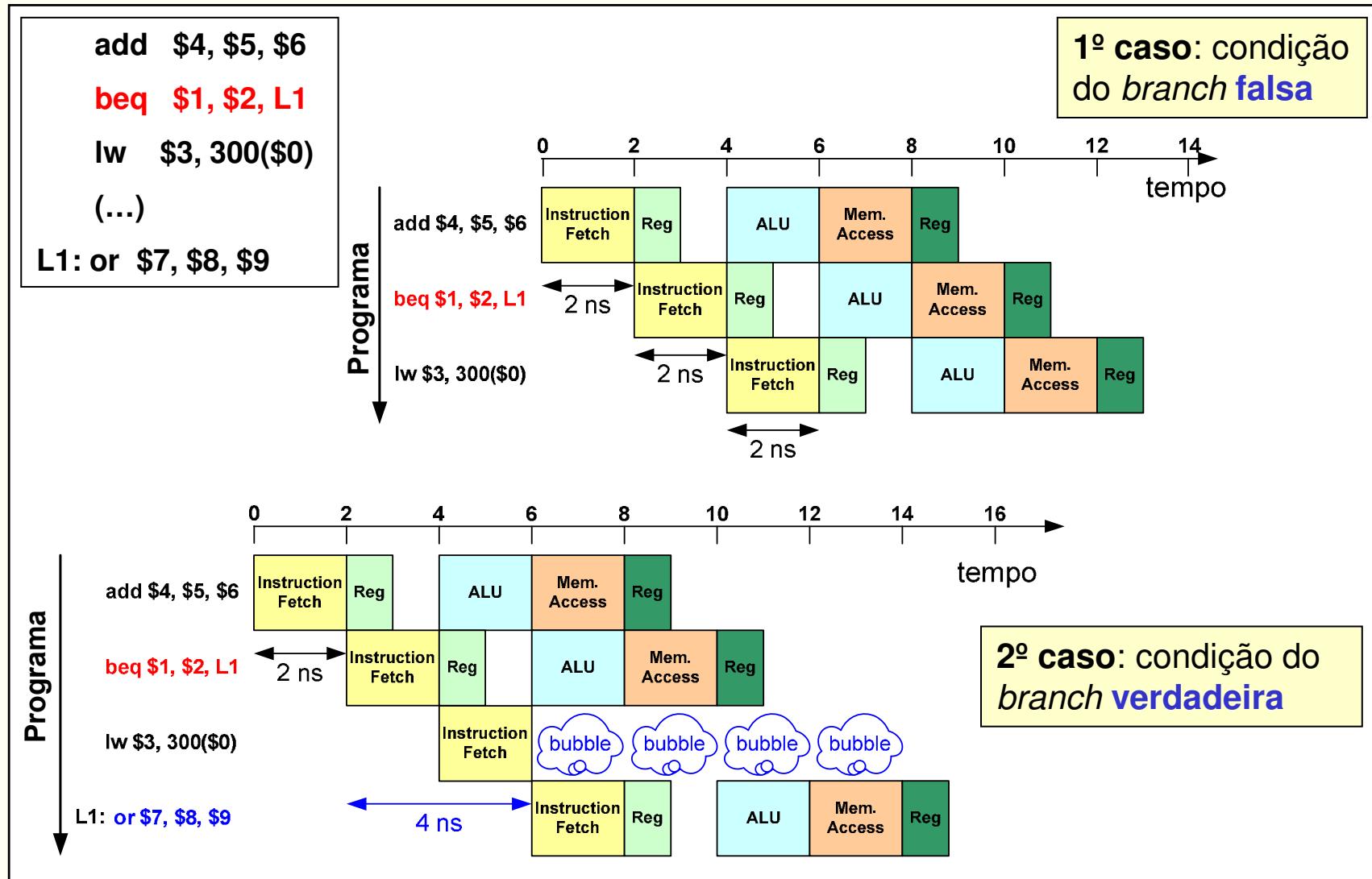


# Hazards de controlo

- Uma solução alternativa ao *pipeline stalling* é designada por **previsão** (*prediction*):
  - Assume-se que a condição do *branch* é falsa (*branch not taken*), pelo que a próxima instrução a ser executada será a que estiver em PC+4 – estratégia designada por **previsão estática not taken**
  - Se a previsão falhar, a instrução entretanto lida (a seguir ao *branch*) é descartada (convertida em **nop**), continuando o *instruction fetch* na instrução correta
- Se a previsão estiver certa esta estratégia permite poupar tempo; para o exemplo anterior, se a previsão for correta 50% das vezes, a relação de desempenho passa a ser:  
$$\text{Ganho} = 1 / (1 + 1 * 0,15 / 2) = 0,93$$



# Hazards de controlo – previsão *not taken*



# Hazards de controlo – previsão

- Os previsores usados nas arquiteturas mais recentes são mais elaborados
- **Previsores estáticos**: o resultado da previsão não é dependente do resultado da execução das instruções:
  - Previsor *Not taken*
  - Previsor *Taken*
  - Previsor *Backward taken, Forward not taken* (BTFTNT)
- **Previsores dinâmicos**: o resultado da previsão depende da história de *branches* anteriores:
  - Guardam informação do resultado *taken/not taken* de *branches* anteriores e do *target address*
  - A previsão é feita com base na informação estatística guardada



# Hazards de controlo – a solução do MIPS

- Uma outra alternativa para resolver os *hazards* de controlo - adotada no MIPS - é designada por ***delayed branch***
- Nesta abordagem, o processador **executa sempre a instrução que se segue ao *branch***, independentemente de a condição ser verdadeira ou falsa
- Esta técnica é implementada com a ajuda do **compilador/assembler** que:
  - organiza as instruções do programa por forma a trocar a ordem do *branch* com uma instrução anterior (desde que não haja dependência entre as duas), ou
  - não sendo possível efetuar a troca de instruções introduz um **NOP** ("no operation"; ex.: **sll, \$0, \$0, 0**) a seguir ao *branch*
- Não é uma técnica comum nos processadores modernos



## Hazards de controlo – delayed branch

- Esta técnica é escondida do programador pelo compilador/assembler:

### Código original

```
add $4, $5, $6  
beq $1, $2, L1  
lw $3, 300($0)  
(...)  
L1: or $7, $8, $9
```

Assembler troca a  
ordem das duas 1<sup>as</sup>  
instruções

### Código reordenado

```
beq $1, $2, L1  
add $4, $5, $6  
lw $3, 300($0)  
(...)  
L1: or $7, $8, $9
```

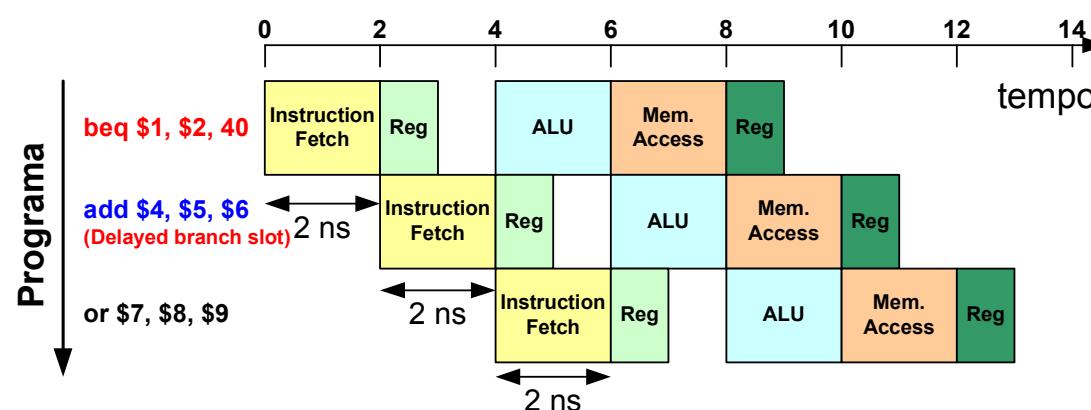
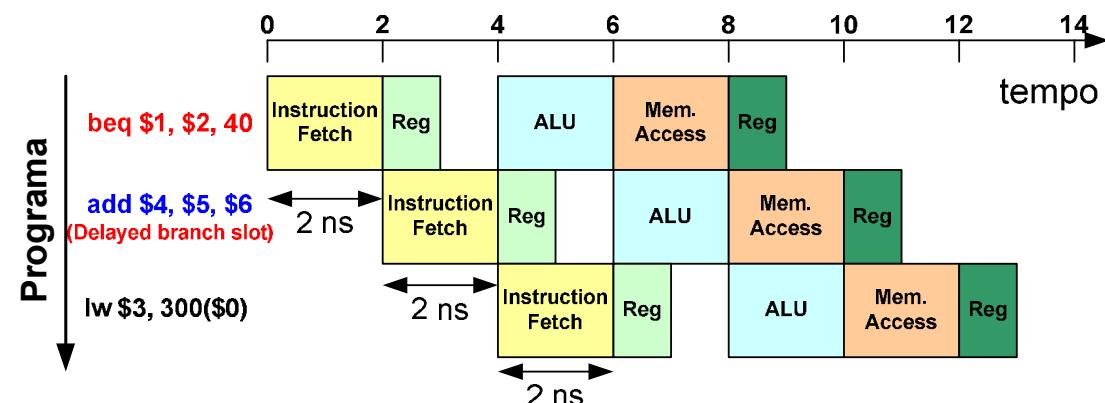
- Neste exemplo a instrução "beq" não depende do resultado produzido pela instrução "add", logo a troca das duas não altera o resultado final do programa
- A instrução "add" é executada independentemente do resultado do "beq"



# Hazards de controlo – delayed branch

```
beq $1, $2, L1  
add $4, $5, $6  
lw $3, 300($0)  
(...)  
L1: or $7, $8, $9
```

1º caso: condição do branch falsa



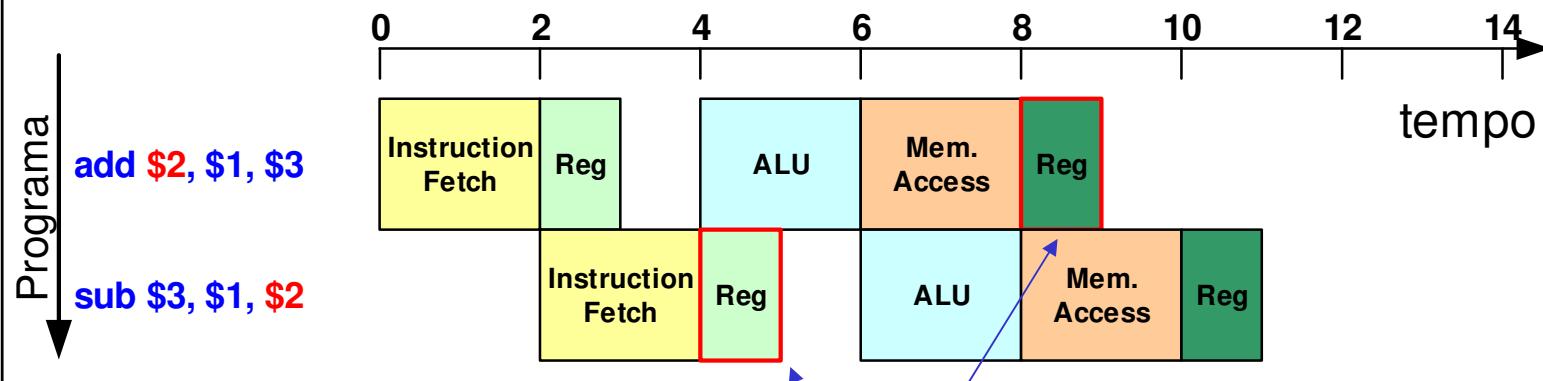
2º caso: condição do branch verdadeira



# Hazards de dados

- O terceiro tipo de *hazards* resulta da **dependência** existente entre o resultado calculado por uma instrução e o operando usado por outra que segue mais atrás no *pipeline* (i.e., mais recente)
- Exemplo:

|     |                       |
|-----|-----------------------|
| add | <b>\$2</b> , \$1, \$3 |
| sub | \$3, \$1, <b>\$2</b>  |



A instrução "**sub \$3,\$1,\$2**" não pode ser executada antes de o valor de **\$2** ser calculado e armazenado pela instrução anterior (o valor é necessário em **t = 4**, mas só vai ser escrito no registo destino em **t = 10**)



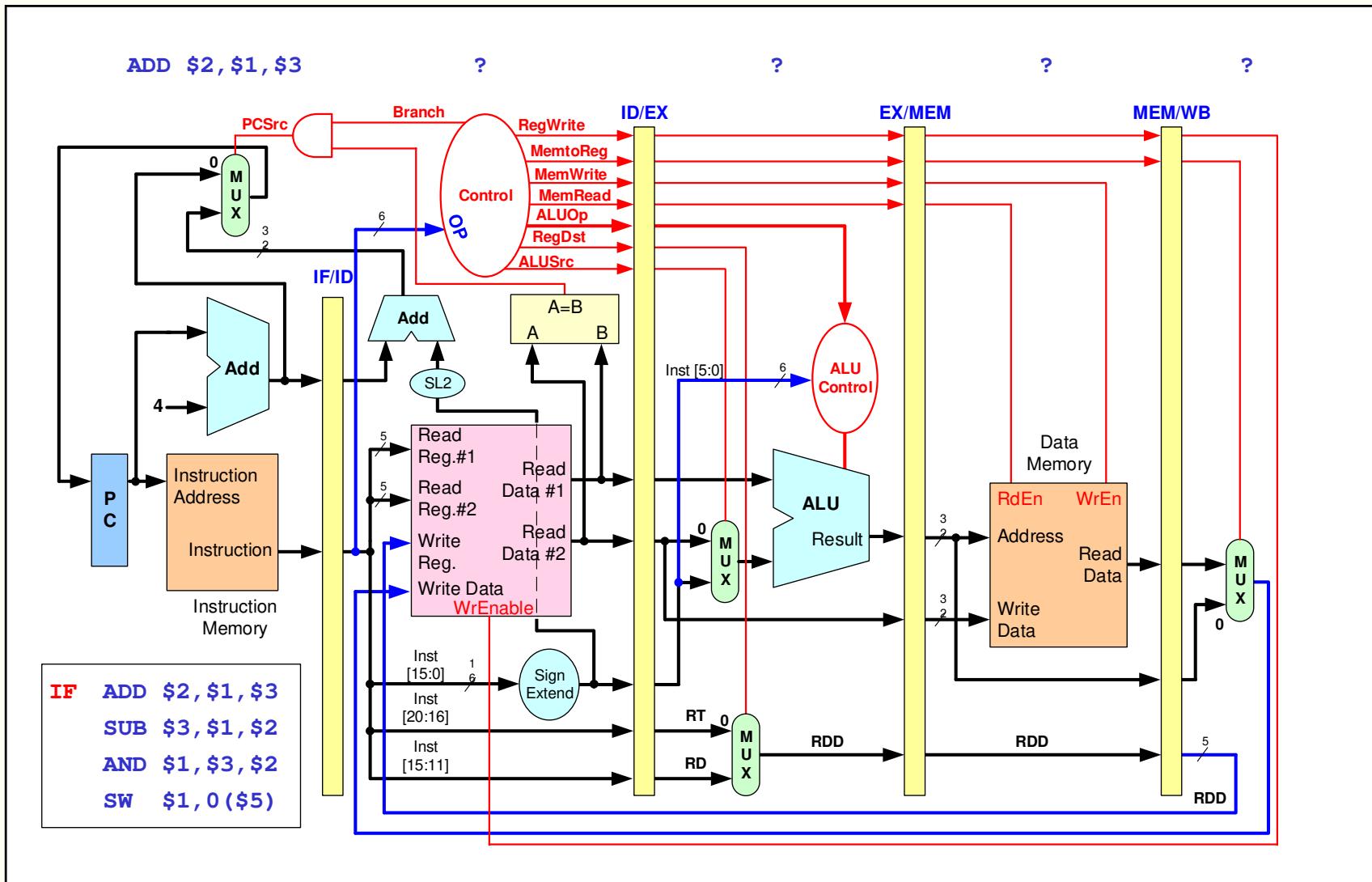
# Hazards de dados

- Se o resultado que vai ser necessário para a instrução mais recente ainda não tiver sido armazenado, então essa instrução não poderá prosseguir porque irá tomar como operando um valor incorreto (**a escrita no registo só é feita quando a instrução chega a WB**)
- Exemplo:

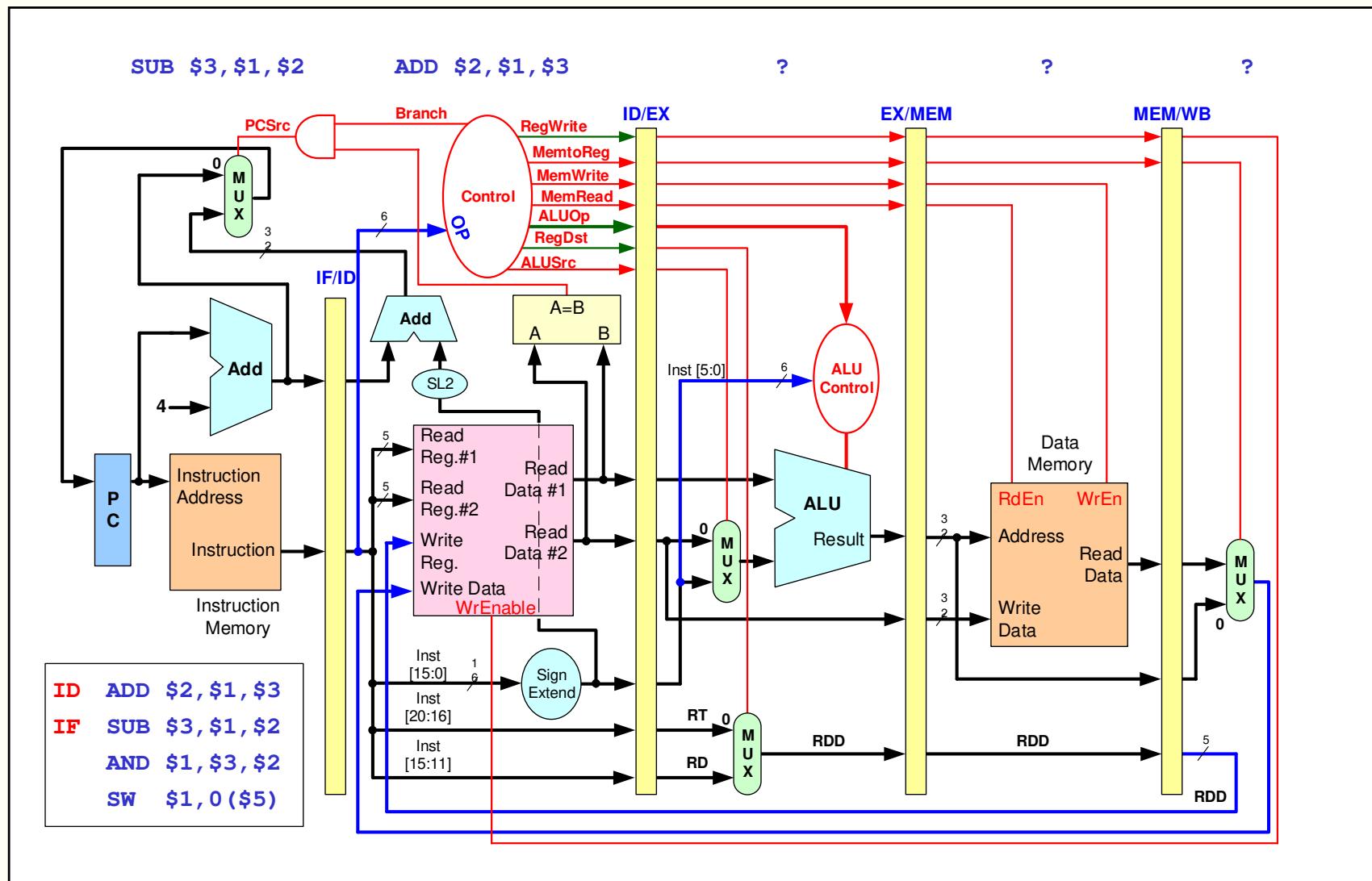
```
ADD $2,$1,$3    #
SUB $3,$1,$2    # Hazard de dados ($2)
AND $1,$3,$2    # Hazard de dados ($3)
SW  $1,0($5)    # Hazard de dados ($1)
```
- Primeira solução: ***stall do pipeline***
  - parar a progressão da instrução que necessita do valor (e das anteriores, no pipeline), na etapa **ID**, até que a instrução que produz o resultado chegue à etapa **WB**
  - se a escrita no banco de registos for feita a meio do ciclo de relógio, então a instrução que necessita do valor poderá prosseguir na transição de relógio seguinte, já com o valor do registo atualizado



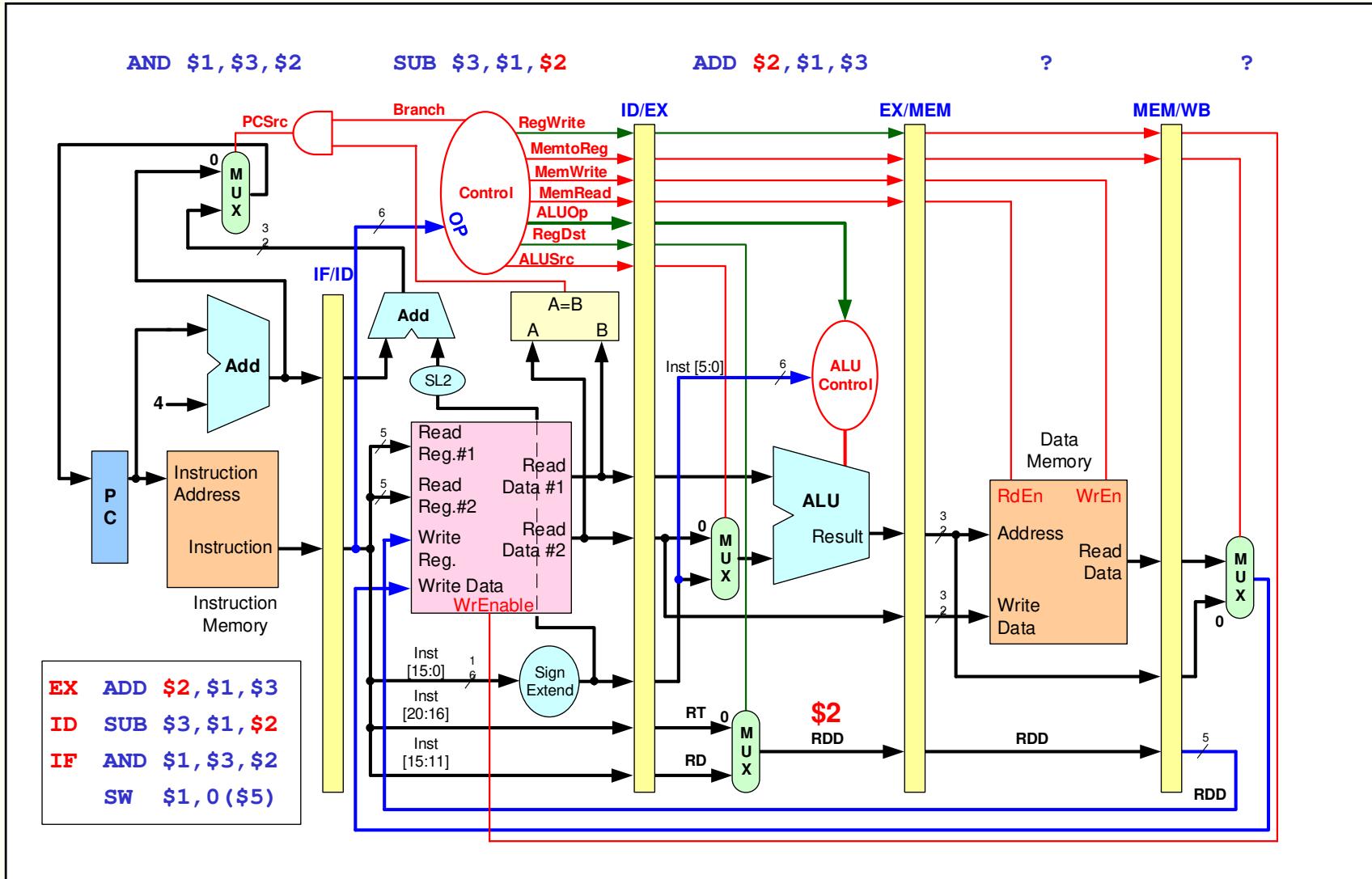
# Hazards de dados resolvidos com *stalling* (1)



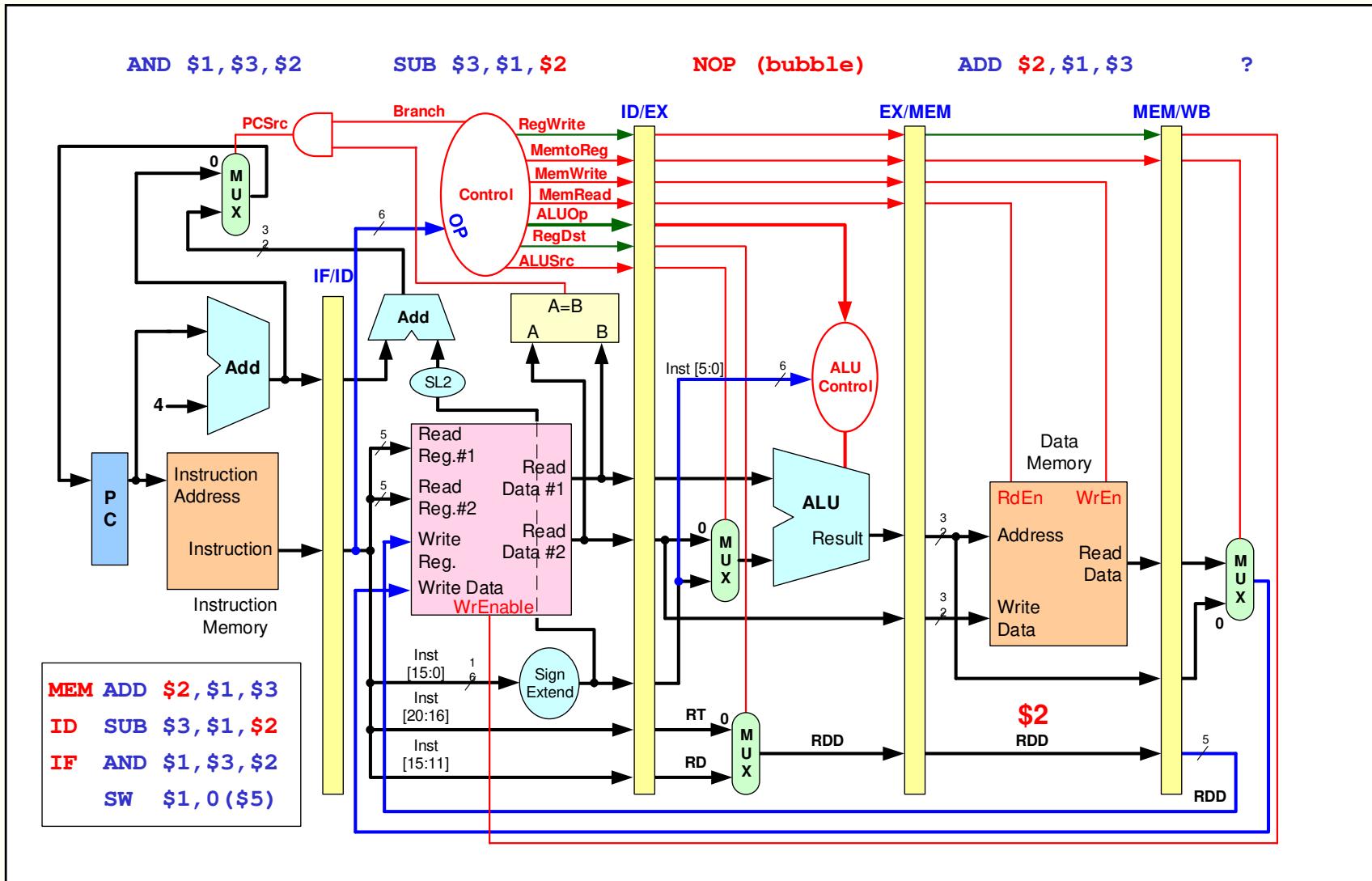
## Hazards de dados resolvidos com *stalling* (2)



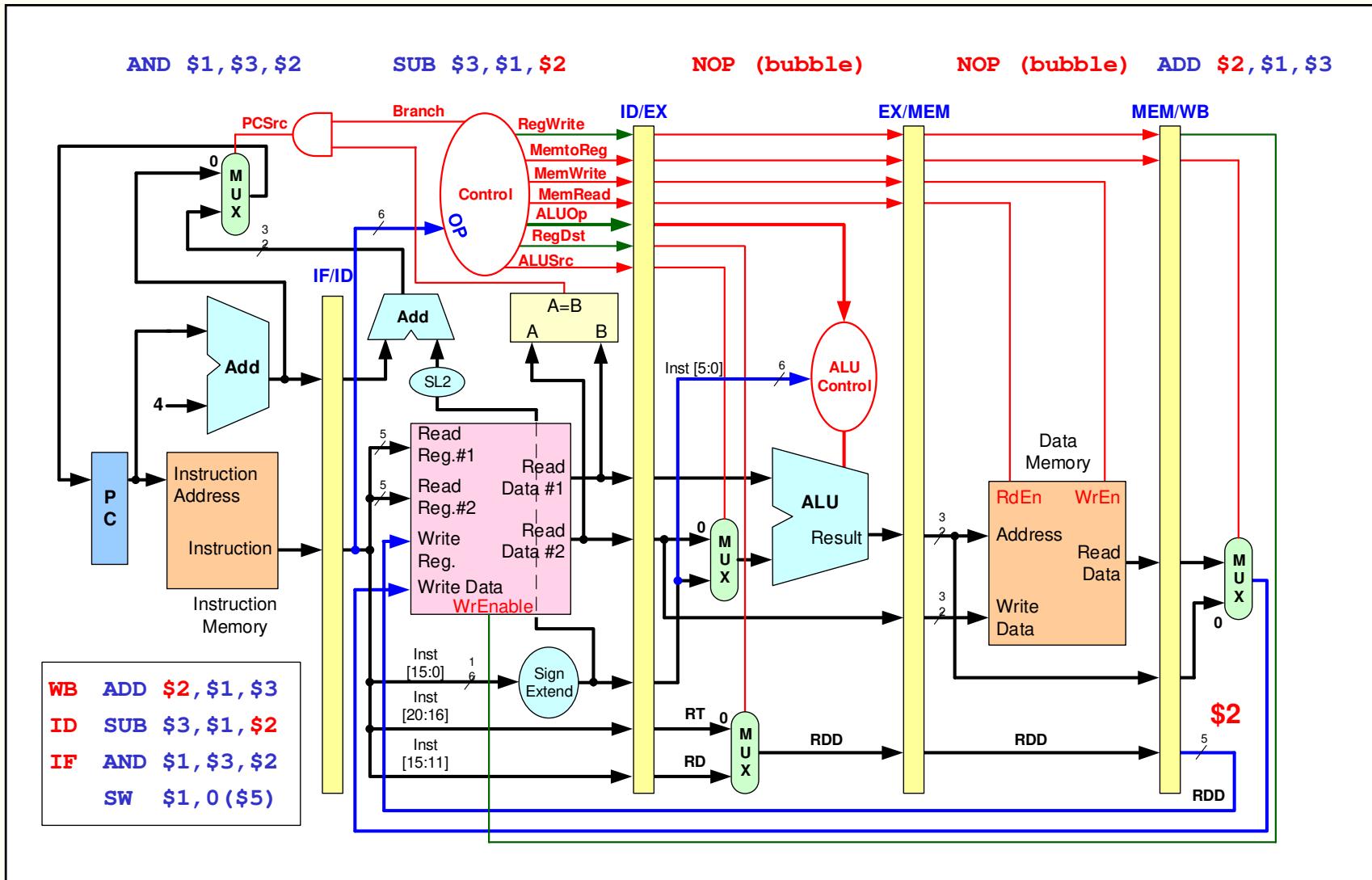
# Hazards de dados resolvidos com stalling (3)



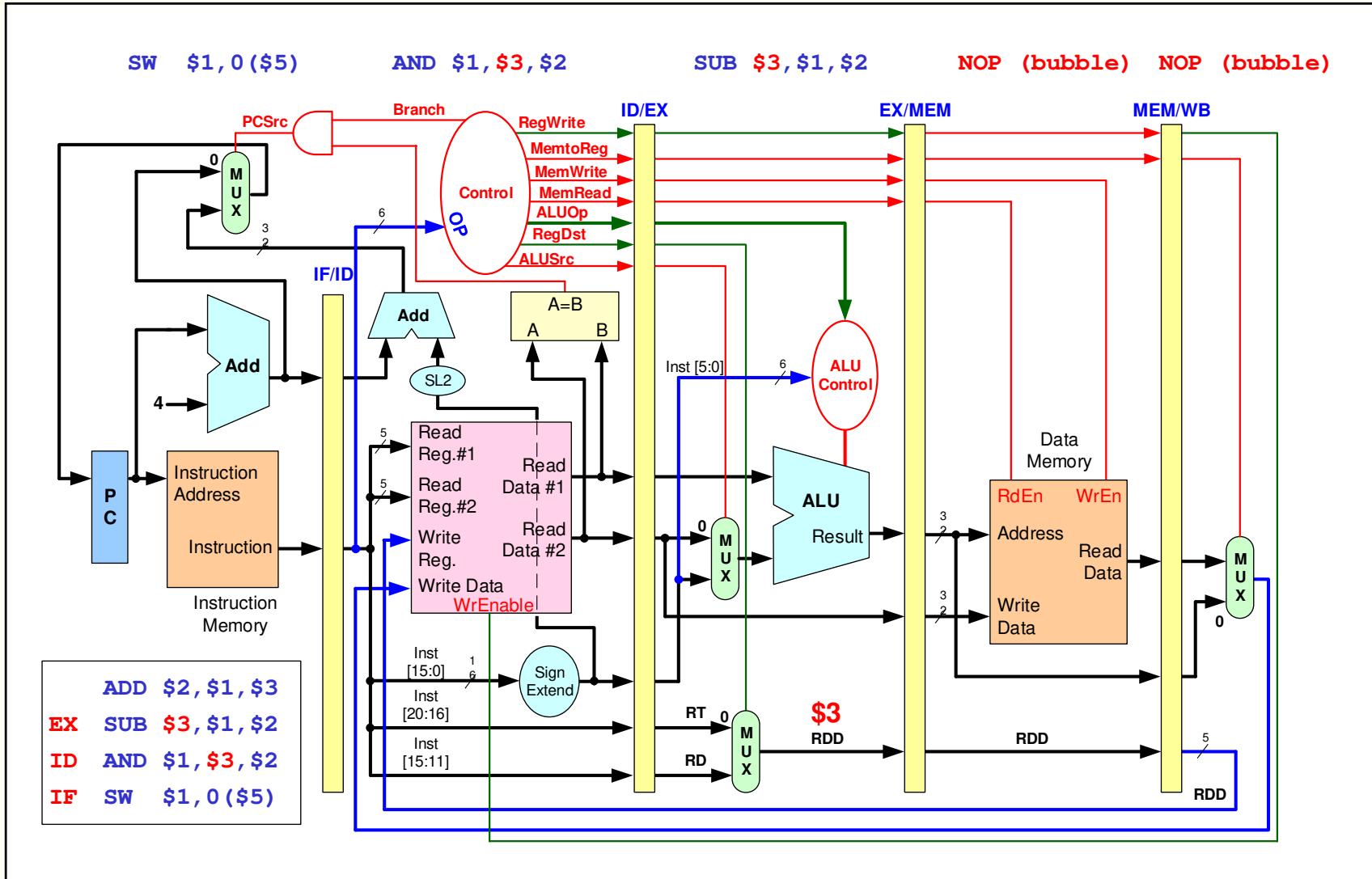
# Hazards de dados resolvidos com stalling (4)



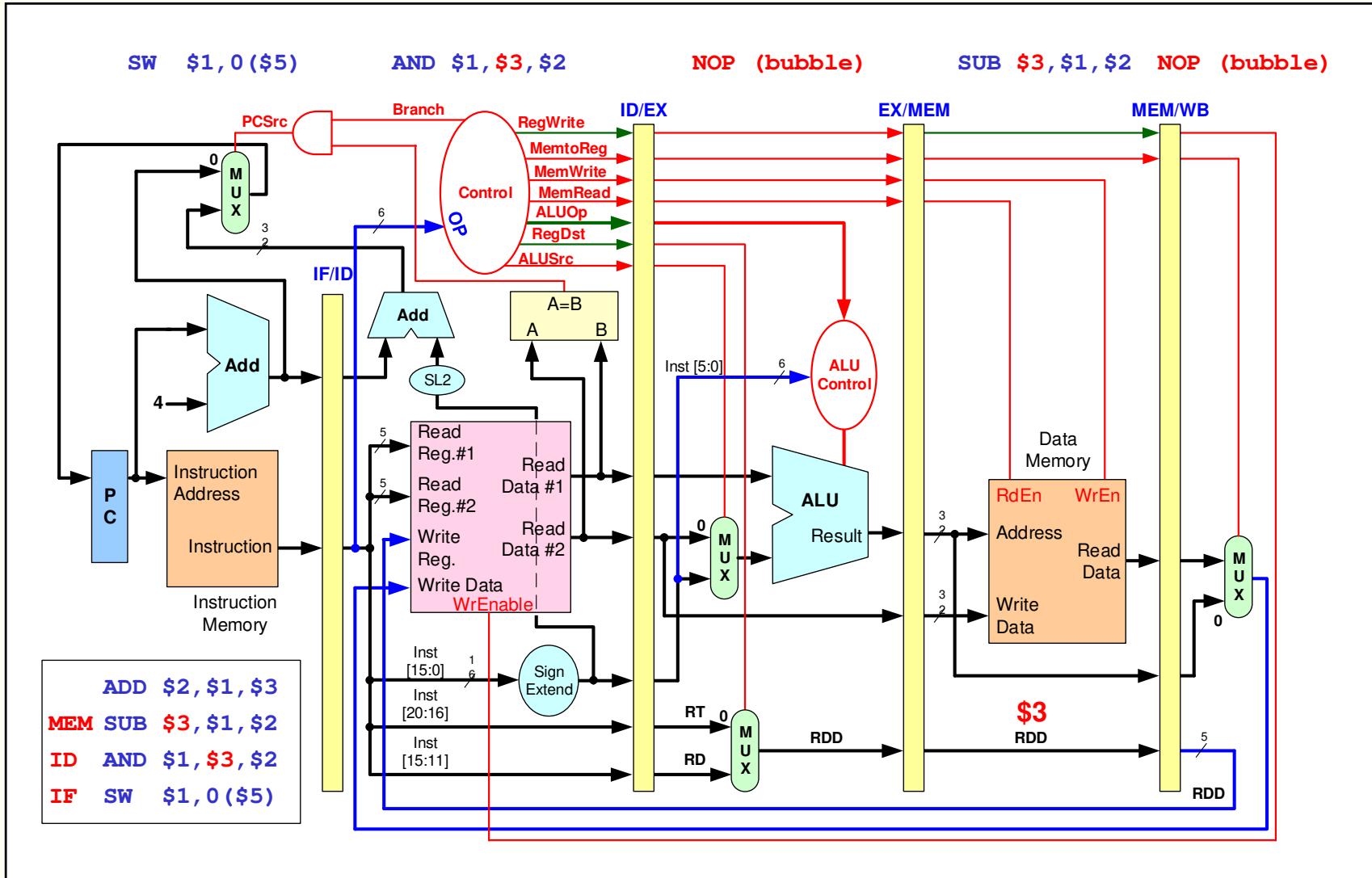
# Hazards de dados resolvidos com stalling (5)



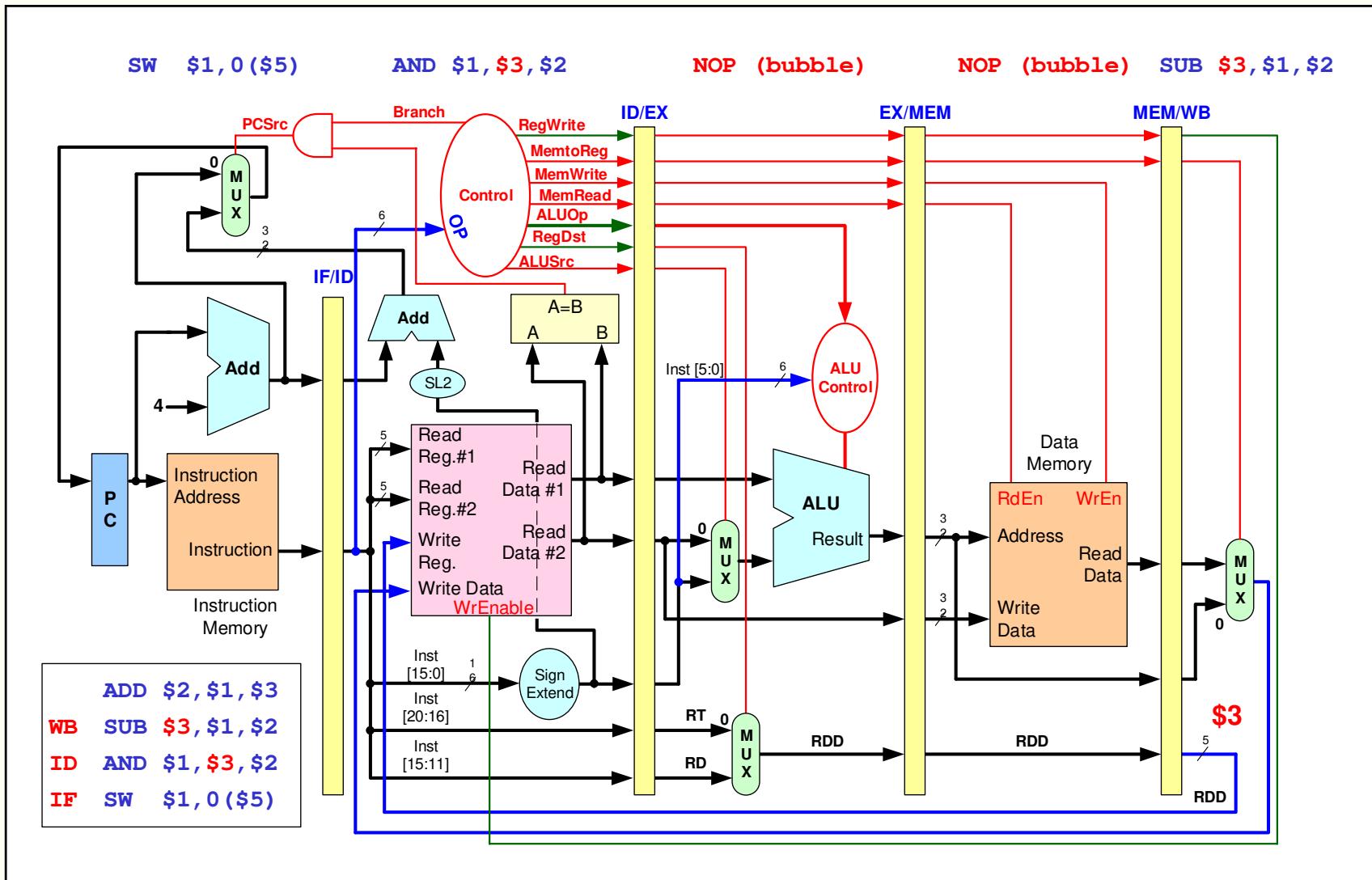
# Hazards de dados resolvidos com stalling (6)



# Hazards de dados resolvidos com stalling (7)



# Hazards de dados resolvidos com stalling (8)



# Hazards de dados

- Esperar pela conclusão da instrução que produz o resultado (através de *stalling*) tem um impacto elevado no desempenho...
- Cada instrução com dependência atrasa a progressão do *pipeline* em 2 ciclos de relógio

$$\text{Texec\_sem\_stalls} = F + (N-1) = 5 + (4-1) = 8 \text{ ciclos de relógio}$$

$$\text{Texec} = F + (N-1) + \text{Nr\_stalls} = 5 + (4-1) + 6 = 14 \text{ ciclos de relógio}$$

- Qual será então a solução?
- A principal solução para a resolução de situações de *hazards* de dados resulta da observação de que não é necessário, na maioria dos casos, esperar pela conclusão da primeira instrução para resolver o *hazard*



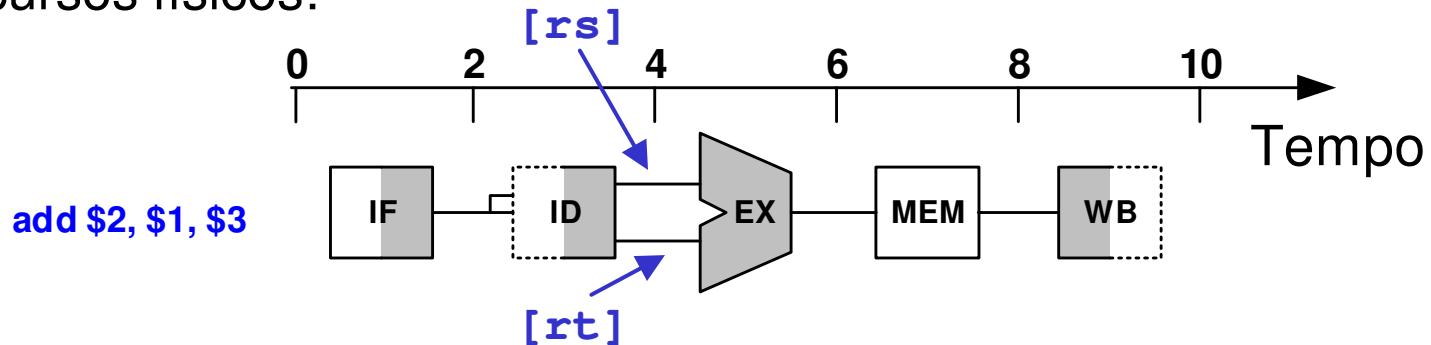
# Hazards de dados

- Para as instruções do tipo R, logo que a operação seja realizada na ALU, (**EX**, 3º estágio), o resultado pode ser disponibilizado para a instrução seguinte
- Esta técnica de disponibilizar um resultado para uma instrução subsequente, mais cedo na cadeia de *pipelining*, é conhecida por **forwarding** ou **bypassing**
- Para exemplificar uma situação de *forwarding*, e tornar mais clara esta técnica, começemos por apresentar uma versão gráfica simplificada da cadeia de *pipelining*



# Hazards de dados

- Nesta representação gráfica usamos símbolos para representar os recursos físicos:

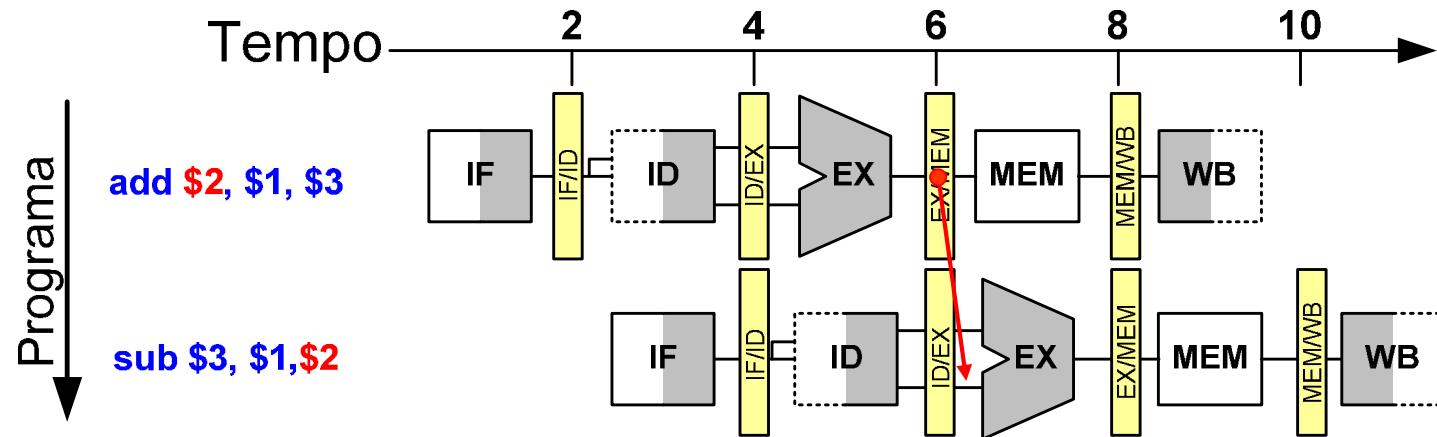


- IF corresponde ao estágio de *instruction fetch*, representando o quadrado a memória de instrução
- A metade cinzenta à direita tipifica uma operação de leitura
- Um quadrado branco (MEM) indica que esse elemento de estado não está envolvido na execução da instrução
- Quando a metade cinzenta está à esquerda, isso indica uma operação de escrita no elemento de estado respetivo (WB)



# Hazards de dados

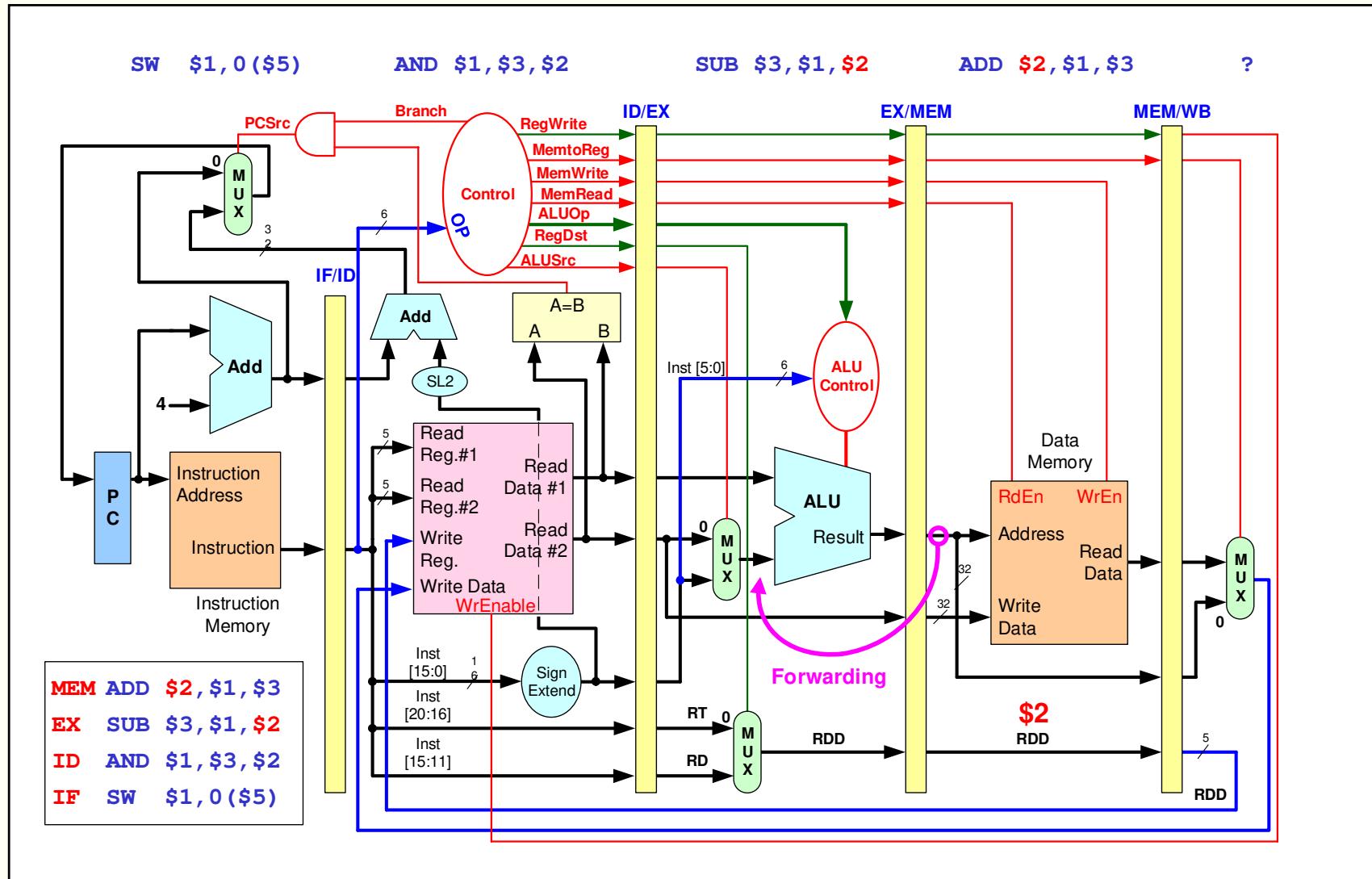
- O exemplo anterior, em que se observou a existência de um *hazard* de dados, pode então ser representado graficamente por:



- O **forwarding** do valor presente no registo **EX/MEM** (resultado da instrução ADD) para a segunda entrada da ALU (estágio **EX**, instrução SUB) resolve o *hazard* de dados
- Esta técnica só funciona, contudo, se o *forwarding* for efetuado para um estágio da instrução subsequente que ainda não tenha ocorrido (relação causal)

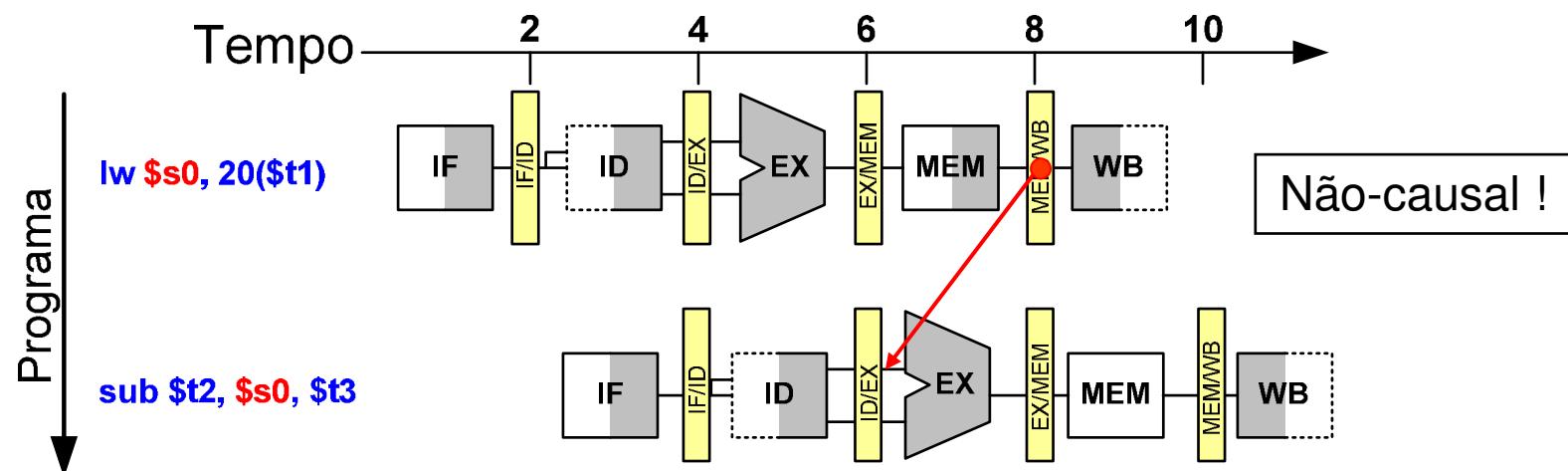


# Hazards de dados - forwarding



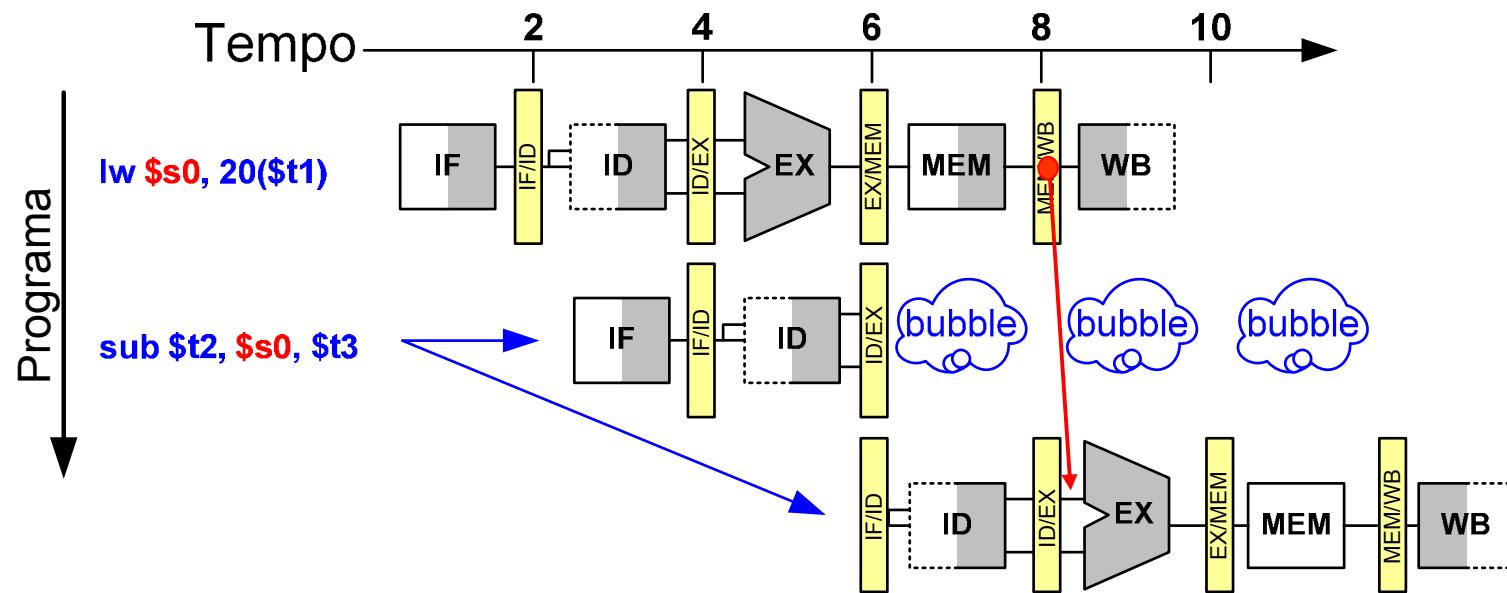
# Hazards de dados

- Há situações em que o *forwarding*, por si só, não resolve o *hazard de dados*
- Um exemplo é o que ocorre quando uma instrução aritmética/lógica depende do resultado de uma instrução de acesso à memória (LW) que ainda não terminou



# Hazards de dados – stalling

- Para resolver essa situação, é necessário:
  - Fazer o **stall** do *pipeline* durante um ciclo de relógio
  - Fazer o **forwarding** do registo **MEM/WB** para o estágio **EX**, para a entrada da ALU



# Hazards de dados – reordenação de instruções

- Parte das situações de *hazards* de dados podem ser atenuadas ou **resolvidas pelo compilador**, através da reordenação de instruções
- A reordenação não pode comprometer o resultado final
- Código original (exemplo):

```
lw    $t0,0($t1)
lw    $t2,4($t1)
sub  $s0,$t2,$t1 # Stalling por hazard de dados
sw    $t0,4($t1)
```

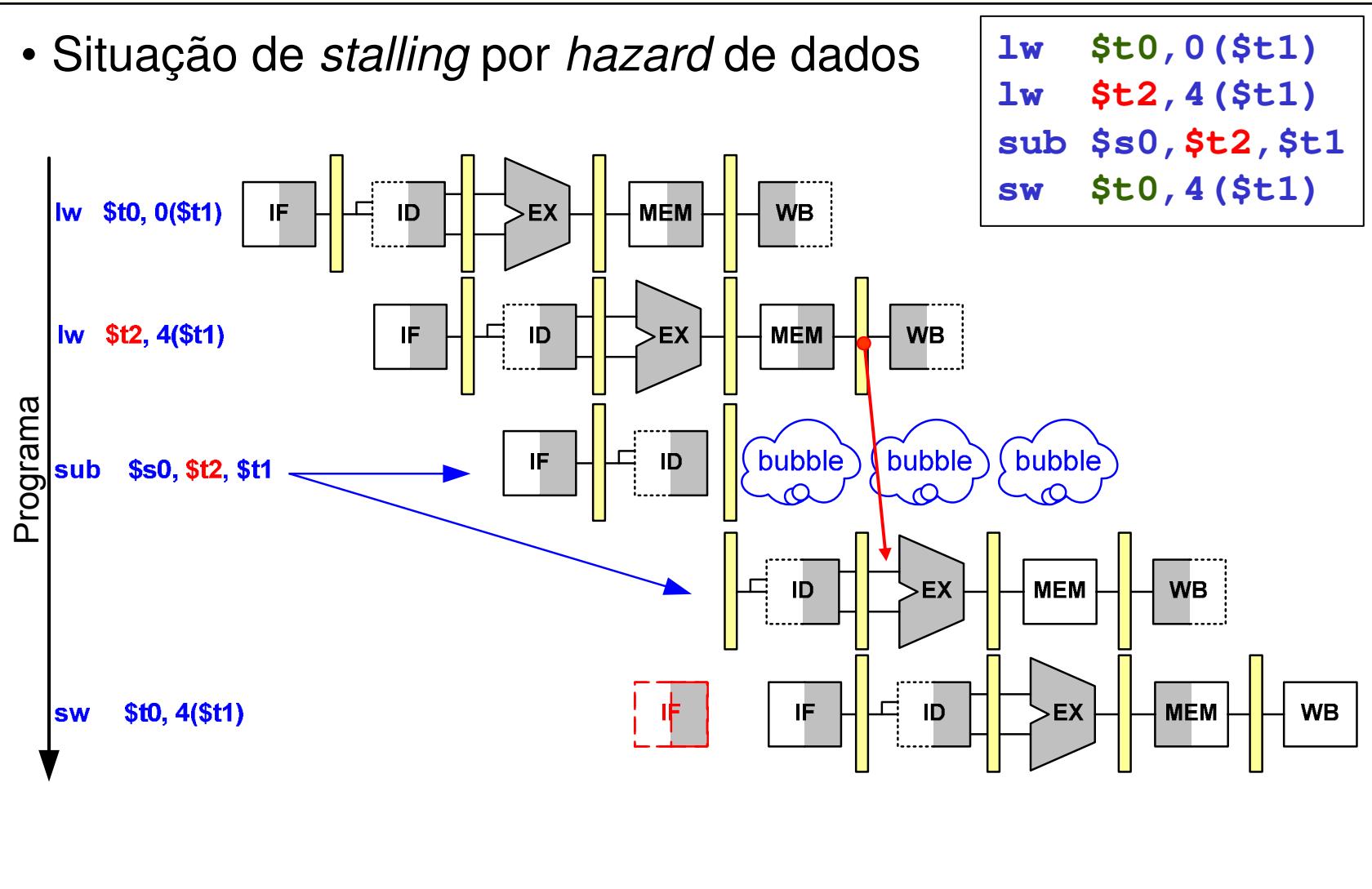
- Código reordenado pelo compilador/assembler:

```
lw    $t0,0($t1)
lw    $t2,4($t1)
sw    $t0,4($t1)  # FW: MEM/WB > EX (rt)
sub  $s0,$t2,$t1 # Stalling resolvido por reordenação
                  # FW: MEM/WB > EX (rs)
```



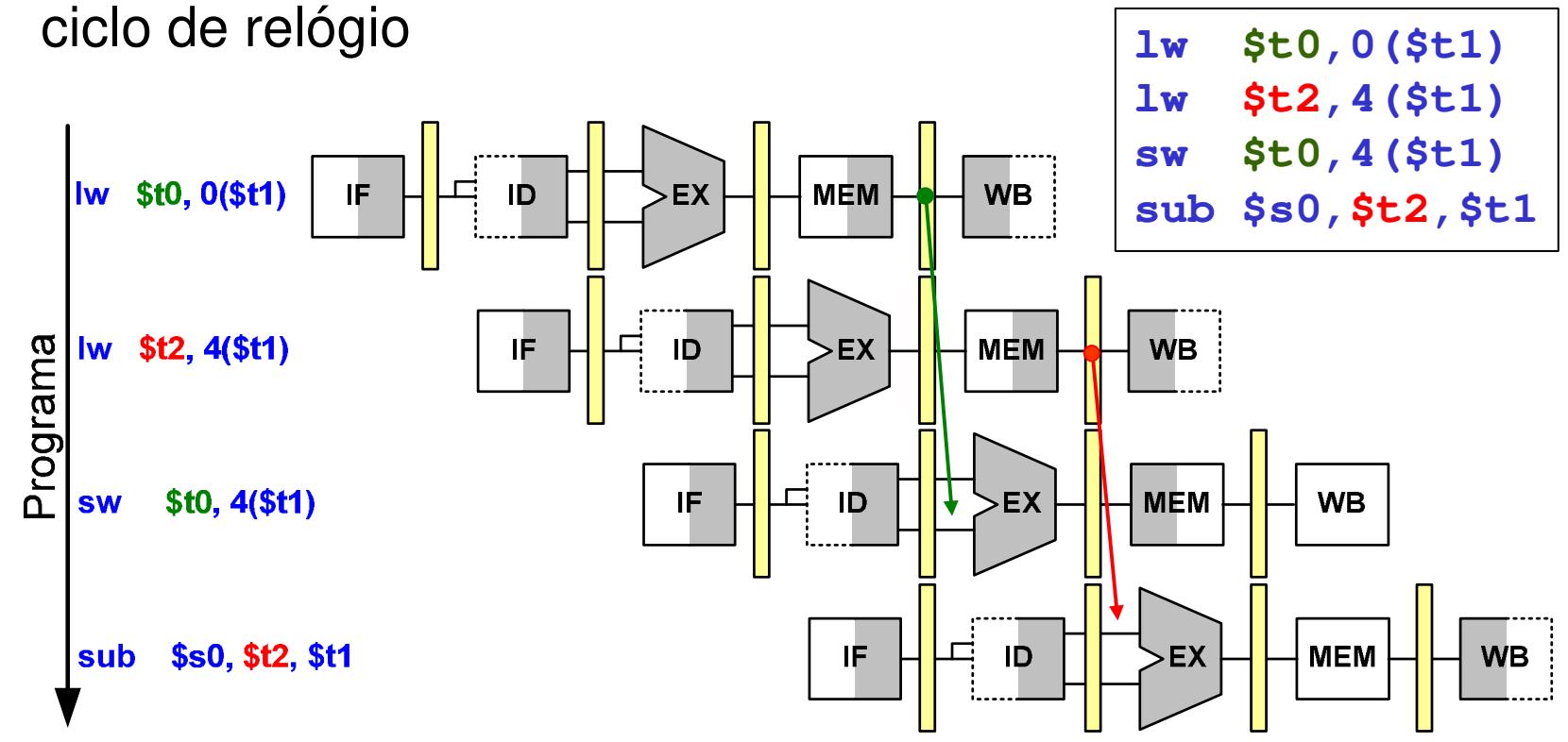
# Hazards de dados – exemplo que gera *stalling*

- Situação de *stalling* por *hazard de dados*

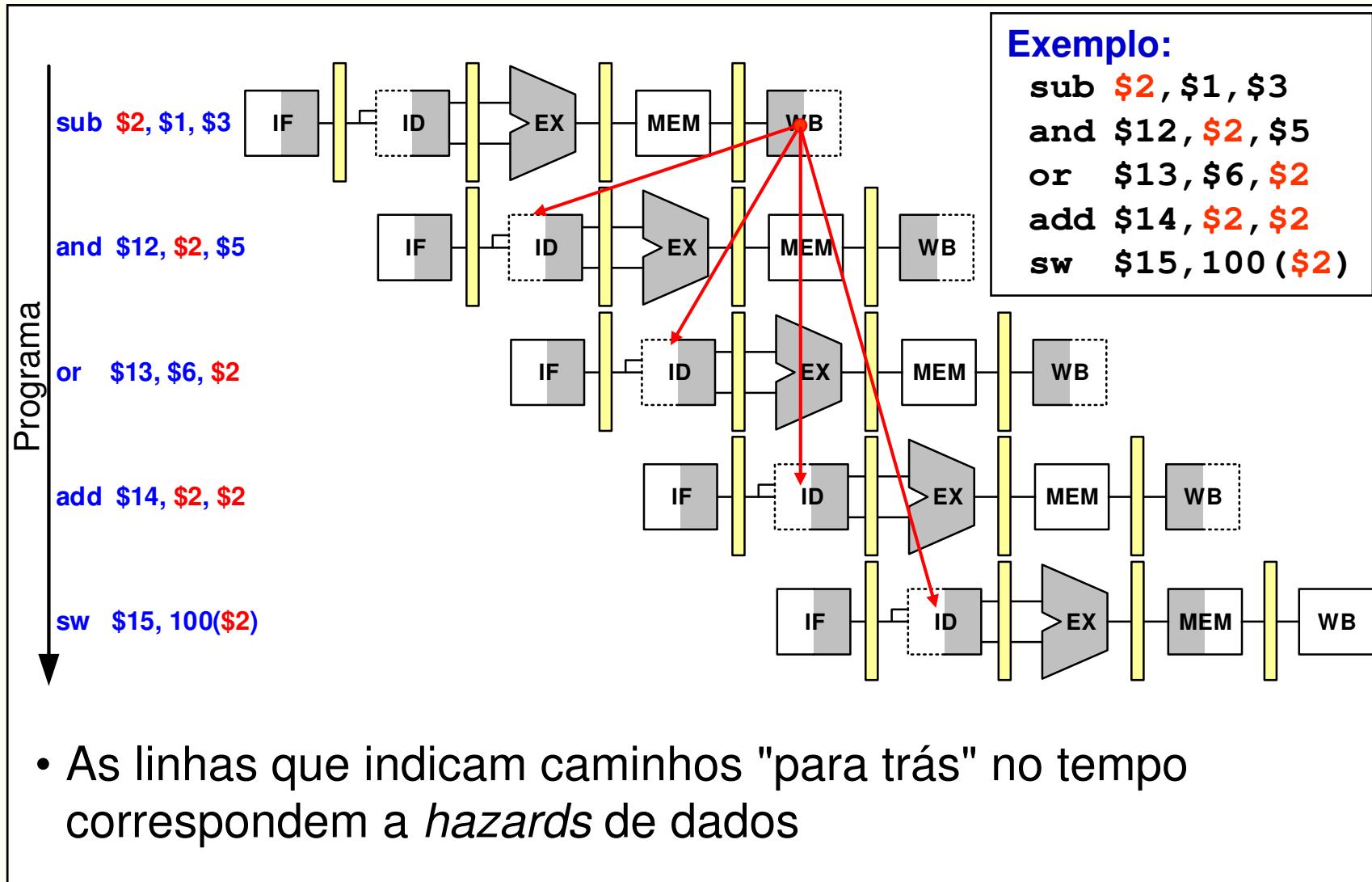


# Hazards de dados

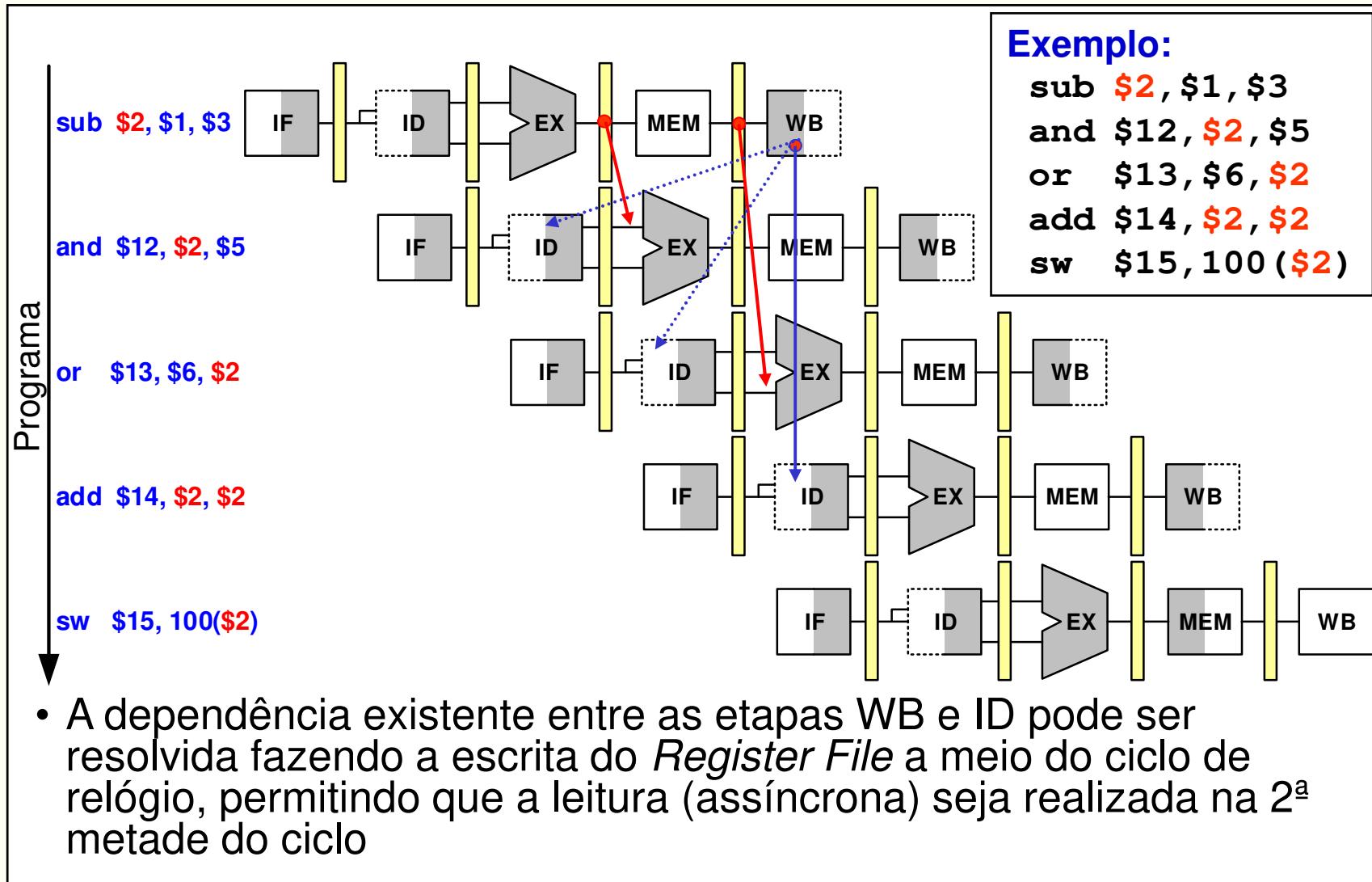
- A situação de *stalling* foi evitada pelo compilador/assembler através de reordenação. A reordenação gera um novo *hazard* de dados que é resolvido por *forwarding*
- A sequência de instruções reordenada executa em menos 1 ciclo de relógio



# Hazards de dados – exemplo



# Hazards de dados – exemplo

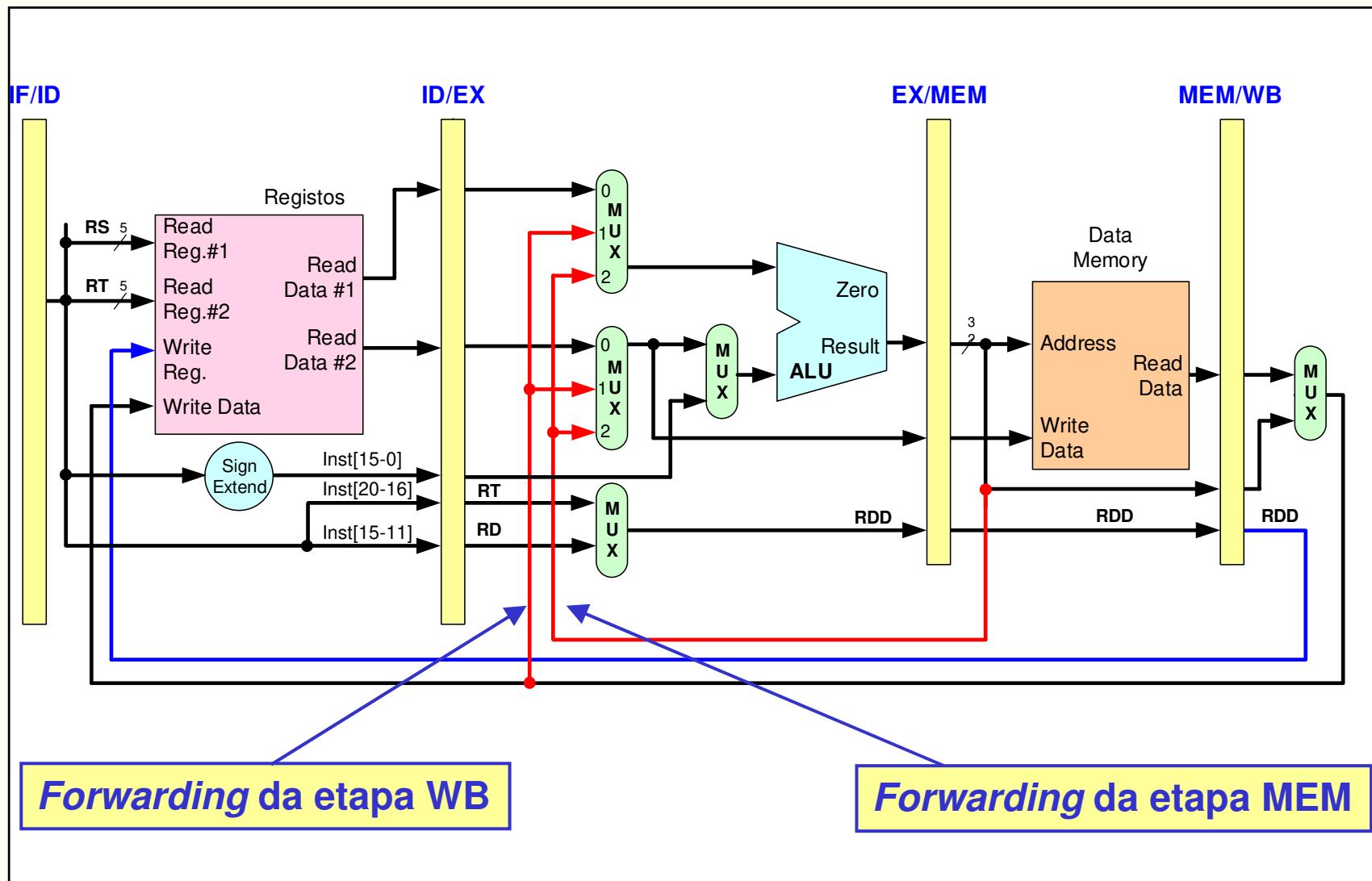


## Hazards de dados – implementação do *forwarding*

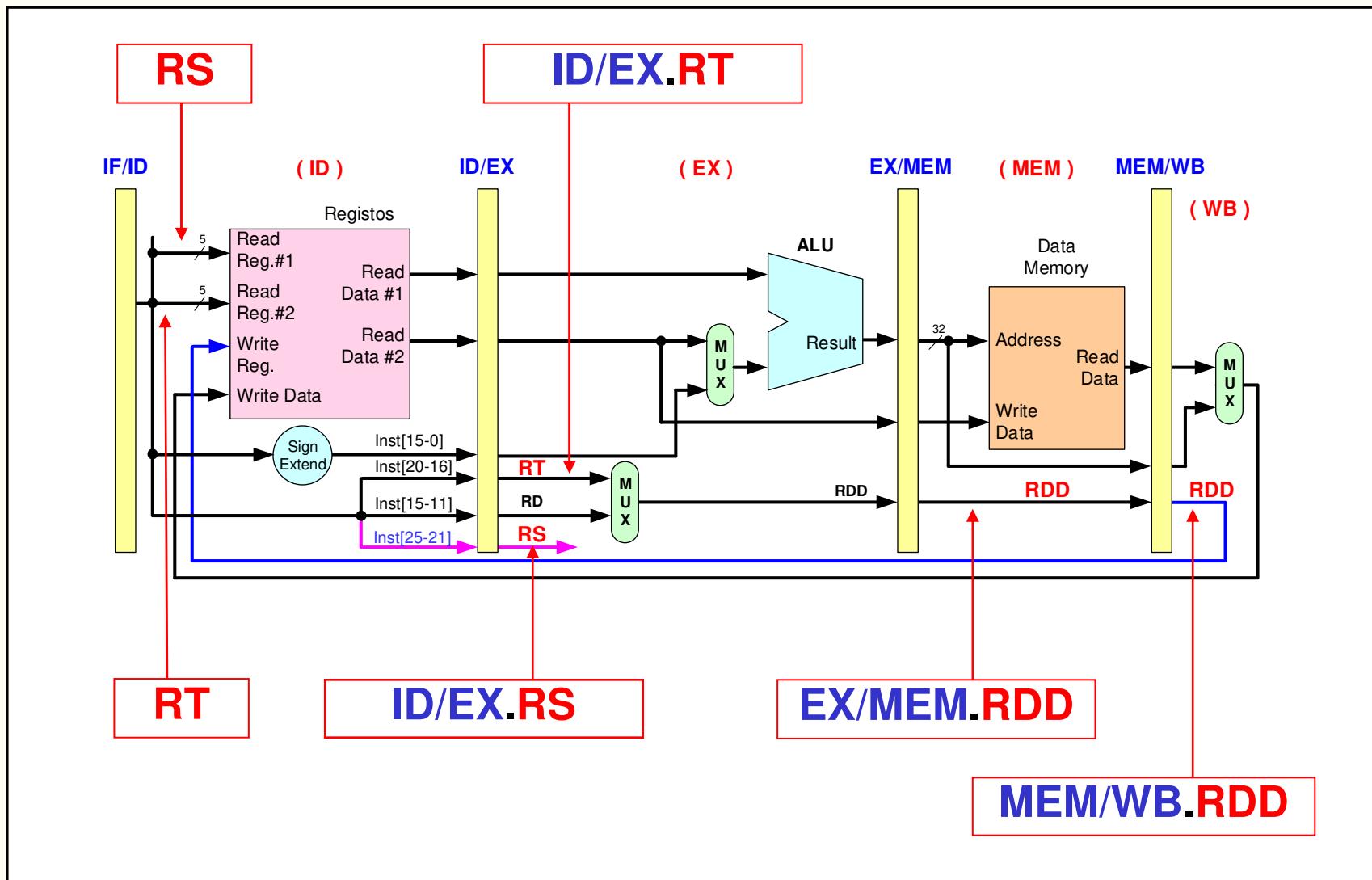
- Para resolver um *hazard* de dados através de **forwarding** é necessário:
  - **Detetar** a situação de *hazard*
  - **Encaminhar** o valor ou os valores que se encontram em fases mais avançadas do *pipeline* (que ainda não foram escritos no registo destino) para onde eles são necessários
- À exceção das instruções de *branch*, a generalidade das outras instruções necessitam dos valores corretos dos registos na fase de execução (**EX**)
- Assim, a resolução de uma parte significativa dos *hazards* de dados resolve-se encaminhando os valores que se encontram em fases mais avançadas do *pipeline* para as entradas da ALU (fase **EX**)



# Hazards de dados – encaminhamento



# Hazards de dados – detecção



# Hazards de dados – deteção

- As situações, correspondentes a *hazard* de dados, em que há necessidade de encaminhar valores para a fase **EX** são:

- Instrução na fase **MEM** cujo registo destino é um registo operando de uma instrução que se encontra na fase **EX**; de forma simplificada:

**EX/MEM.RDD == ID/EX.RS**, e/ou

**EX/MEM.RDD == ID/EX.RT**

**M add \$1, \$2, \$3**

**EX sub \$4, \$1, \$5**

- Instrução na fase **WB** cujo registo destino é um registo operando de uma instrução que se encontra na fase **EX**; de forma simplificada:

**MEM/WB.RDD == ID/EX.RS**, e/ou

**MEM/WB.RDD == ID/EX.RT**

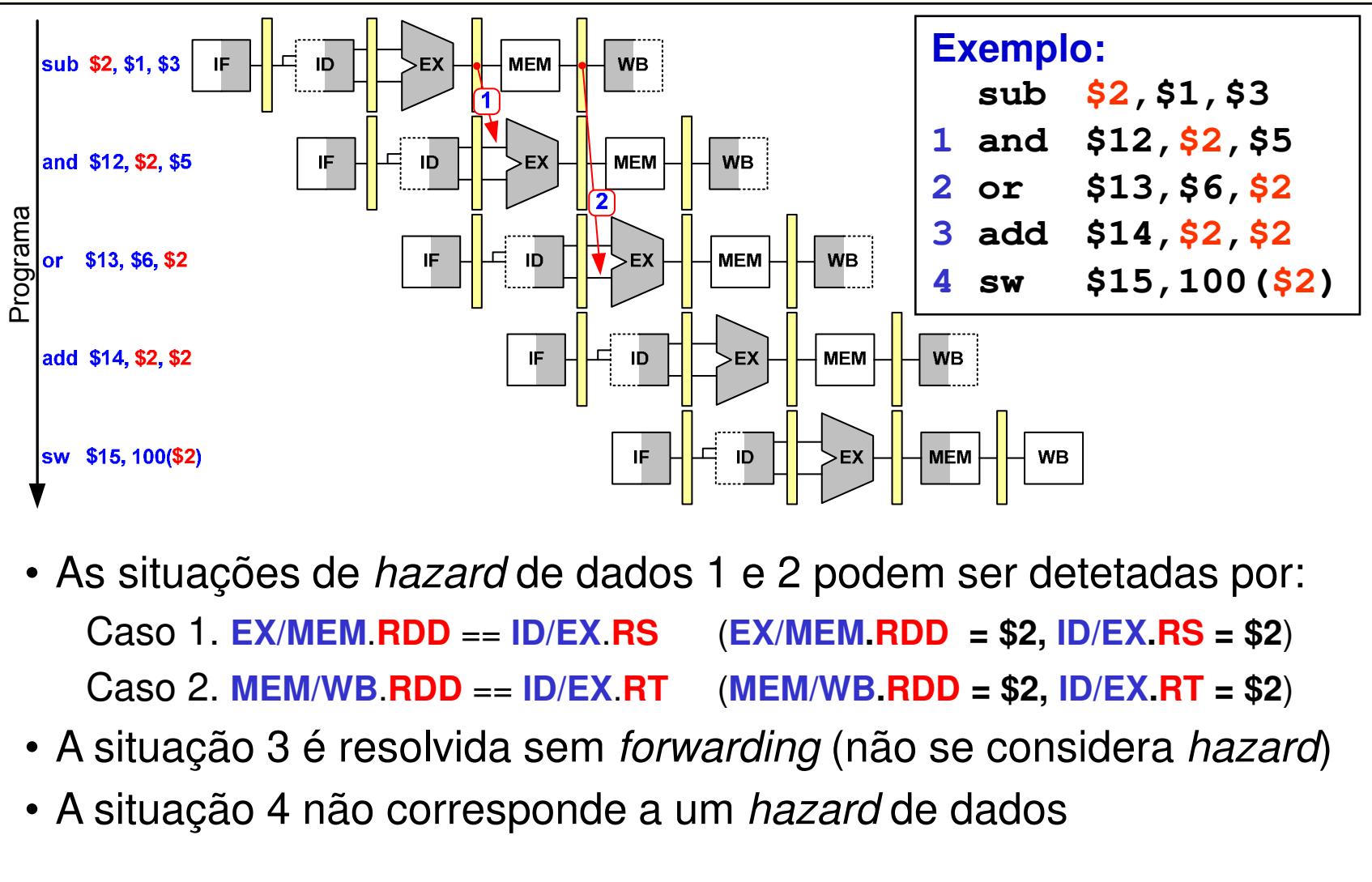
**WB add \$1, \$2, \$3**

**M add \$6, \$2, \$3**

**EX sub \$4, \$5, \$1**

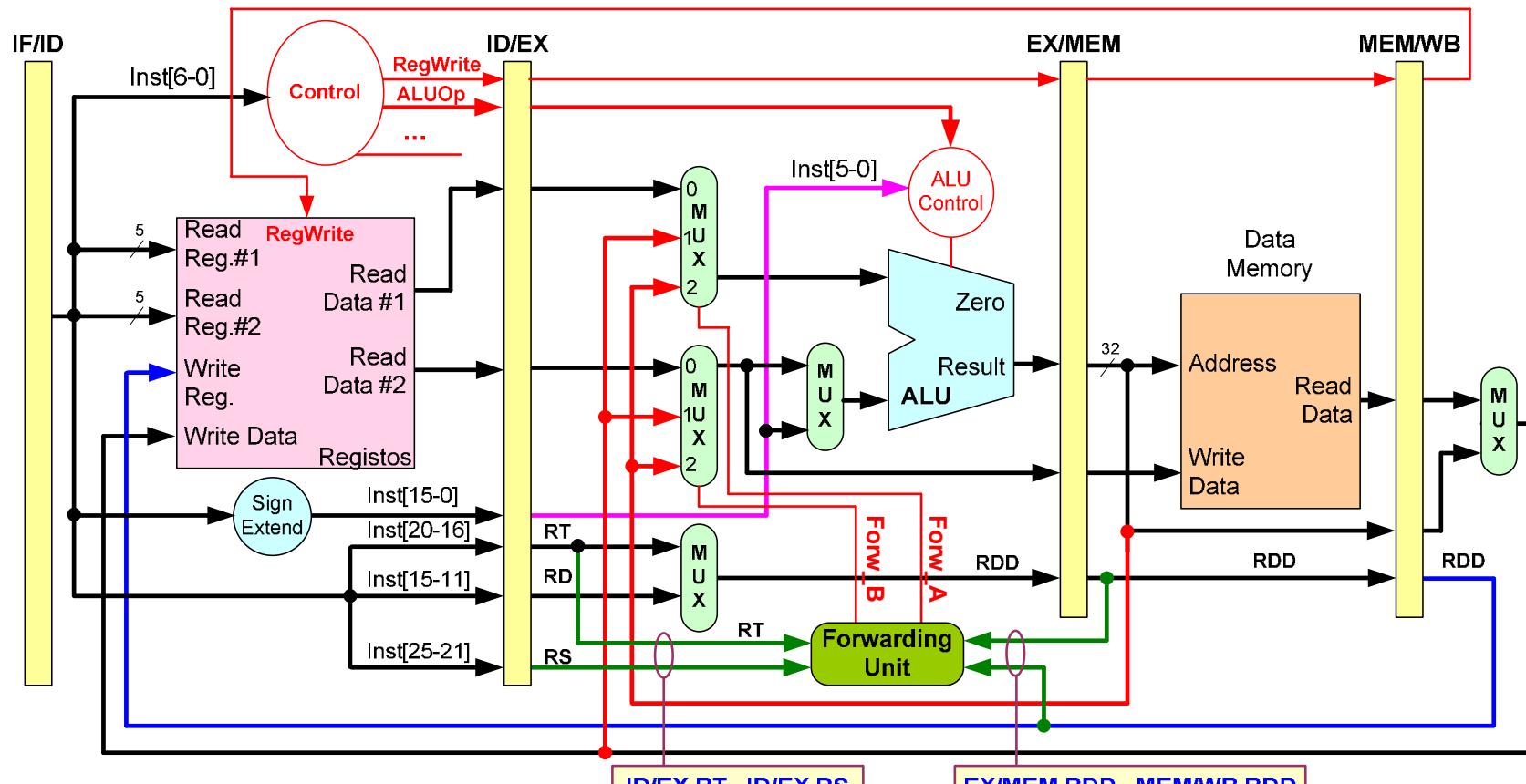


# Hazards de dados – deteção



## Hazards de dados – unidade de controlo de *forwarding*

- Unidade de controlo de *forwarding*, simplificada

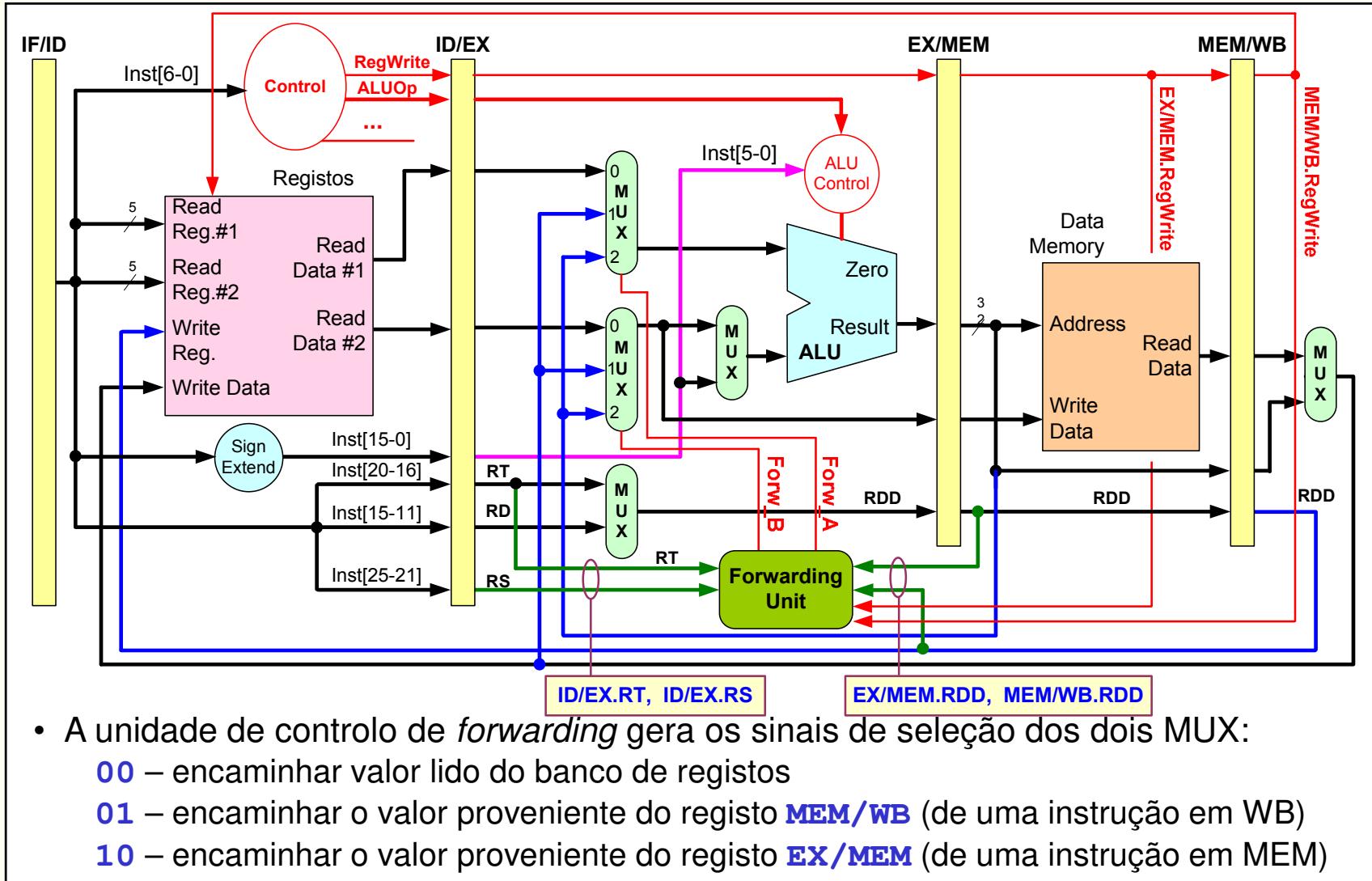


## Hazards de dados – unidade de controlo de *forwarding*

- A simples comparação dos registos não é suficiente para a correta deteção das situações de *hazard* de dados
- O sinal de controlo que permite a escrita no banco de registos (RegWrite) tem igualmente que ser avaliado:
  - Instrução na fase **MEM** que escreve o resultado num registo (**RegWrite = '1'**) e que tem como destino um registo operando de uma instrução que se encontra na fase EX:  
**(EX/MEM.RegWrite == 1) and (EX/MEM.RDD == ID/EX.RS)**  
e/ou  
**(EX/MEM.RegWrite == 1) and (EX/MEM.RDD == ID/EX.RT)**
  - Instrução na fase **WB** que escreve o resultado num registo (**RegWrite = '1'**) igual ao registo operando de uma instrução que se encontra na fase EX:  
**(MEM/WB.RegWrite == 1) and (MEM/WB.RDD == ID/EX.RS)**  
e/ou  
**(MEM/WB.RegWrite == 1) and (MEM/WB.RDD == ID/EX.RT)**



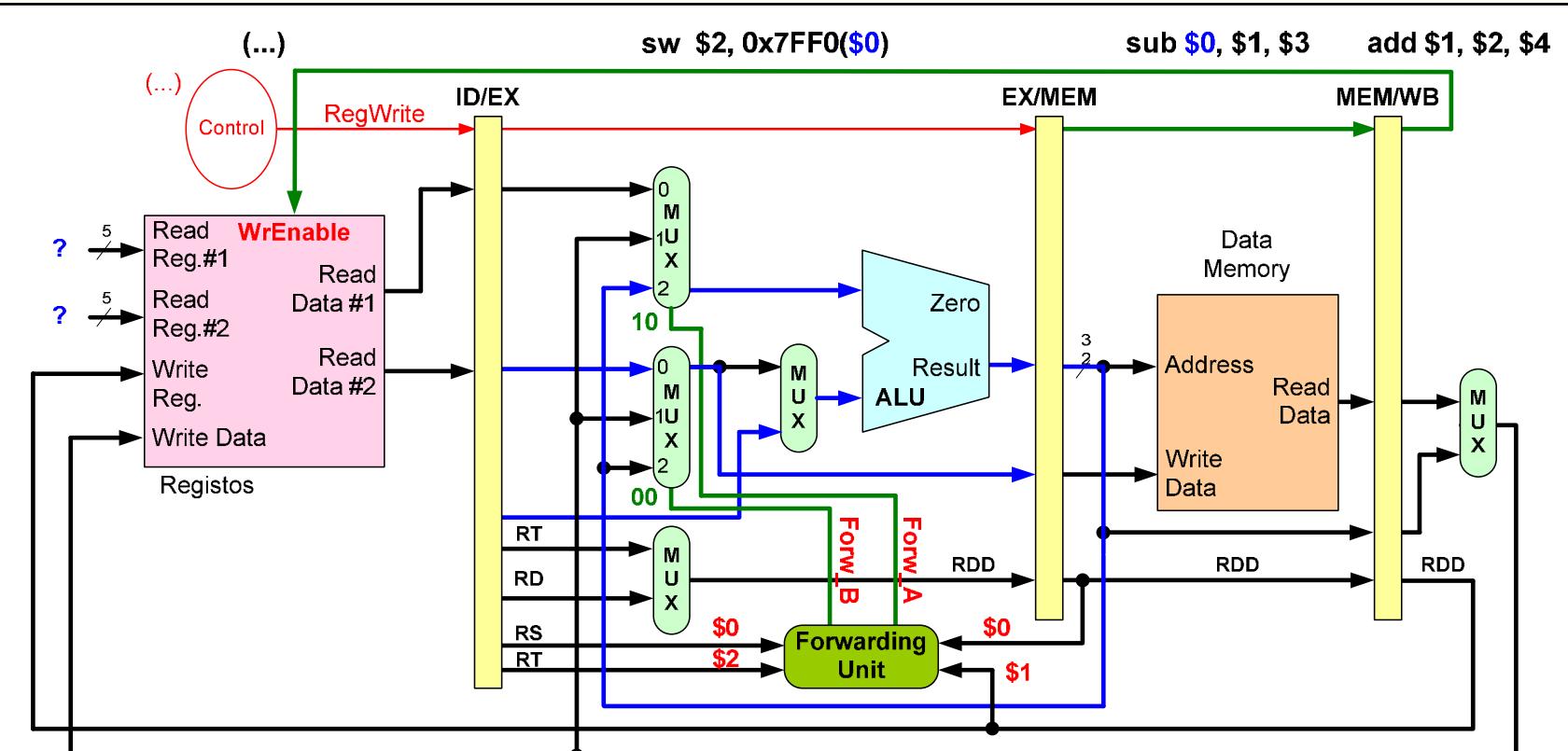
## Hazards de dados – unidade de controlo de *forwarding*



- A unidade de controlo de *forwarding* gera os sinais de seleção dos dois MUX:
  - 00 – encaminhar valor lido do banco de registos
  - 01 – encaminhar o valor proveniente do registo **MEM/WB** (de uma instrução em WB)
  - 10 – encaminhar o valor proveniente do registo **EX/MEM** (de uma instrução em MEM)



## Hazards de dados – unidade de controlo de *forwarding*



- O que acontece caso o *hazard* de dados resulte de um valor de **EX/MEM.RDD = \$0** ou **MEM/WB.RDD = \$0**?
- Como resolver o problema ?



## Unidade de controlo de *forwarding* (para EX) – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ForwardingUnit is
    port(ExMem_RegWrite : in std_logic;
          MemWb_RegWrite : in std_logic;
          IdEx_RS      : in  std_logic_vector(4 downto 0);
          IdEx_RT      : in  std_logic_vector(4 downto 0);
          ExMem_RDD    : in  std_logic_vector(4 downto 0);
          MemWb_RDD    : in  std_logic_vector(4 downto 0);
          Forw_A       : out std_logic_vector(1 downto 0);
          Forw_B       : out std_logic_vector(1 downto 0));
end ForwardingUnit;
```



# Unidade de controlo de *forwarding* (para EX) – VHDL

```
architecture Behavioral of ForwardingUnit is
begin
  process(all)
  begin
    Forw_A <= "00"; -- Op1 comes from Register File
    Forw_B <= "00"; -- Op2 comes from Register File

    if (MemWb_RegWrite = '1' and MemWb_RDD /= "00000") then
      if (MemWb_RDD = IdEx_RS) then Forw_A <= "01"; end if;
      if (MemWb_RDD = IdEx_RT) then Forw_B <= "01"; end if;
    end if;

    if (ExMem_RegWrite = '1' and ExMem_RDD /= "00000") then
      if (ExMem_RDD = IdEx_RS) then Forw_A <= "10"; end if;
      if (ExMem_RDD = IdEx_RT) then Forw_B <= "10"; end if;
    end if;
  end process;
end Behavioral;
```

00 – encaminhar valor lido do banco de registos  
01 – encaminhar o valor proveniente do registo **MEM/WB**  
10 – encaminhar o valor proveniente do registo **EX/MEM**



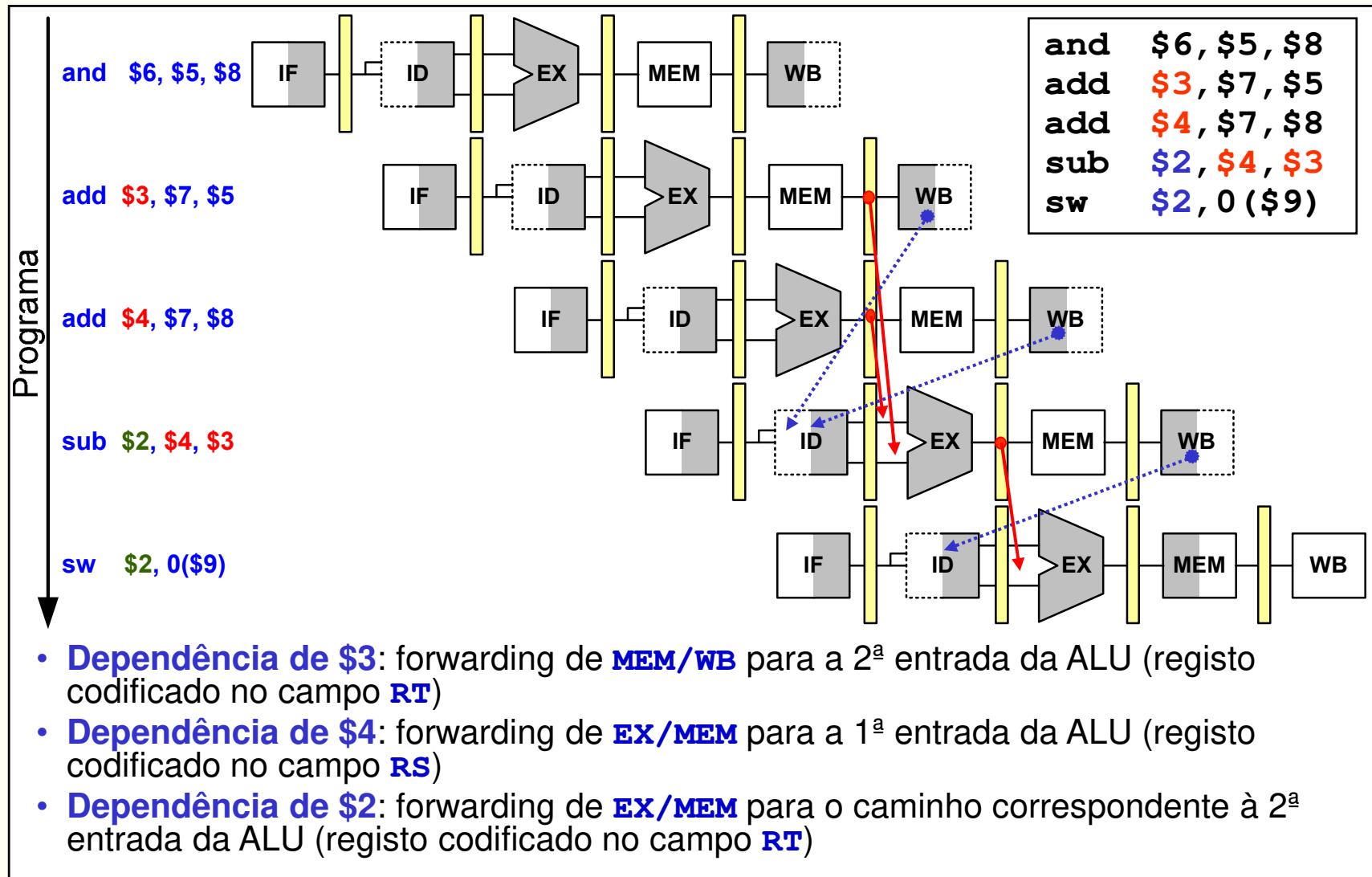
## Exemplo de *forwarding*

```
and $6, $5, $8
add $3, $7, $5
add $4, $7, $8
sub $2, $4, $3    # Hazard de dados: $3, $4
sw $2, 0($9)      # Hazard de dados: $2
```

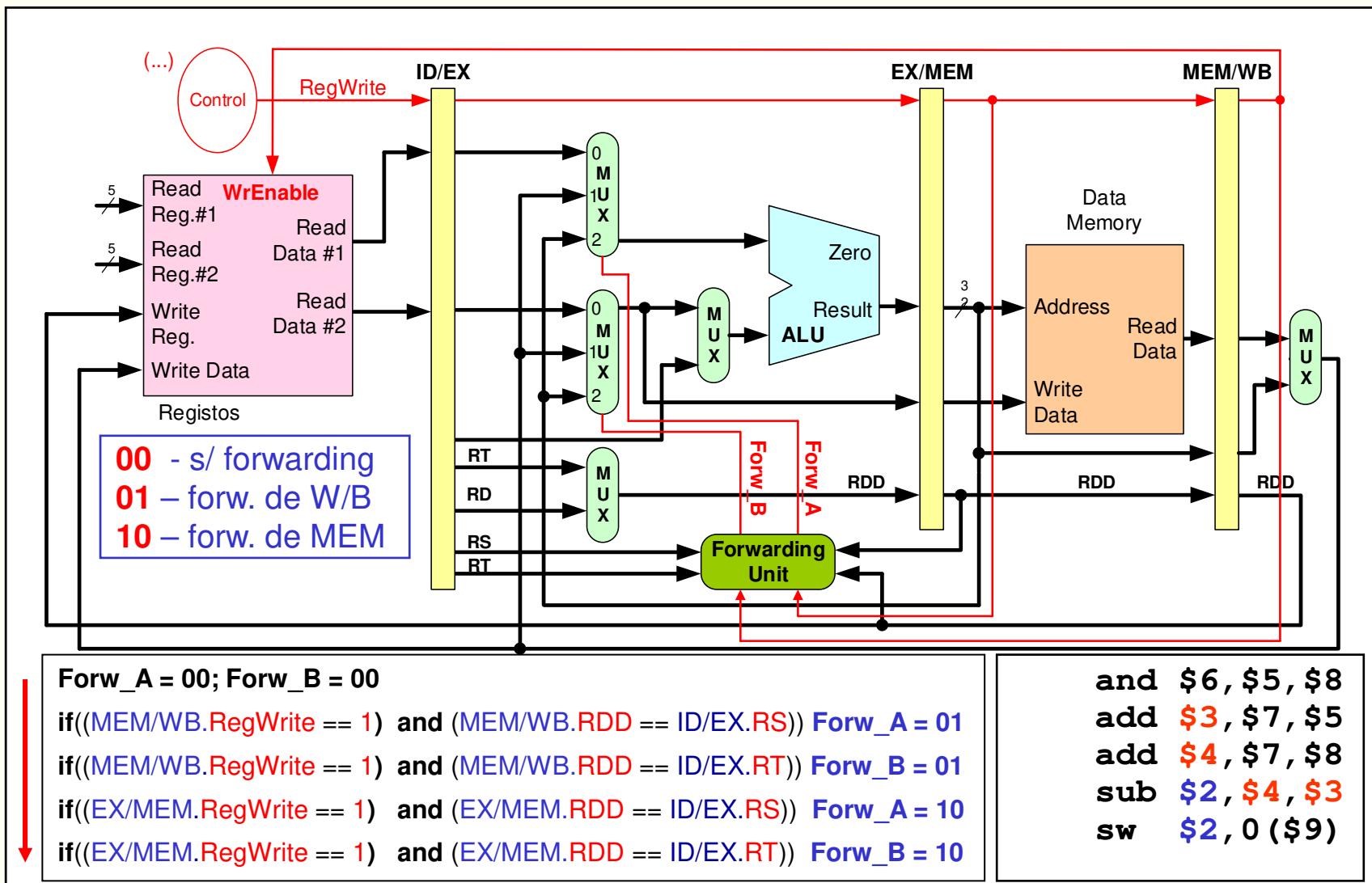
- A instrução "sub \$2, \$4, \$3" apresenta duas situações de *hazards* de dados:
  - dependência do valor de \$4 (add \$4, \$7, \$8)
  - dependência de valor de \$3 (add \$3, \$7, \$5)
- A instrução "sw \$2, 0(\$9)" apresenta igualmente uma situação de *hazard* de dados (dependência em \$2, sub \$2, \$4, \$3)



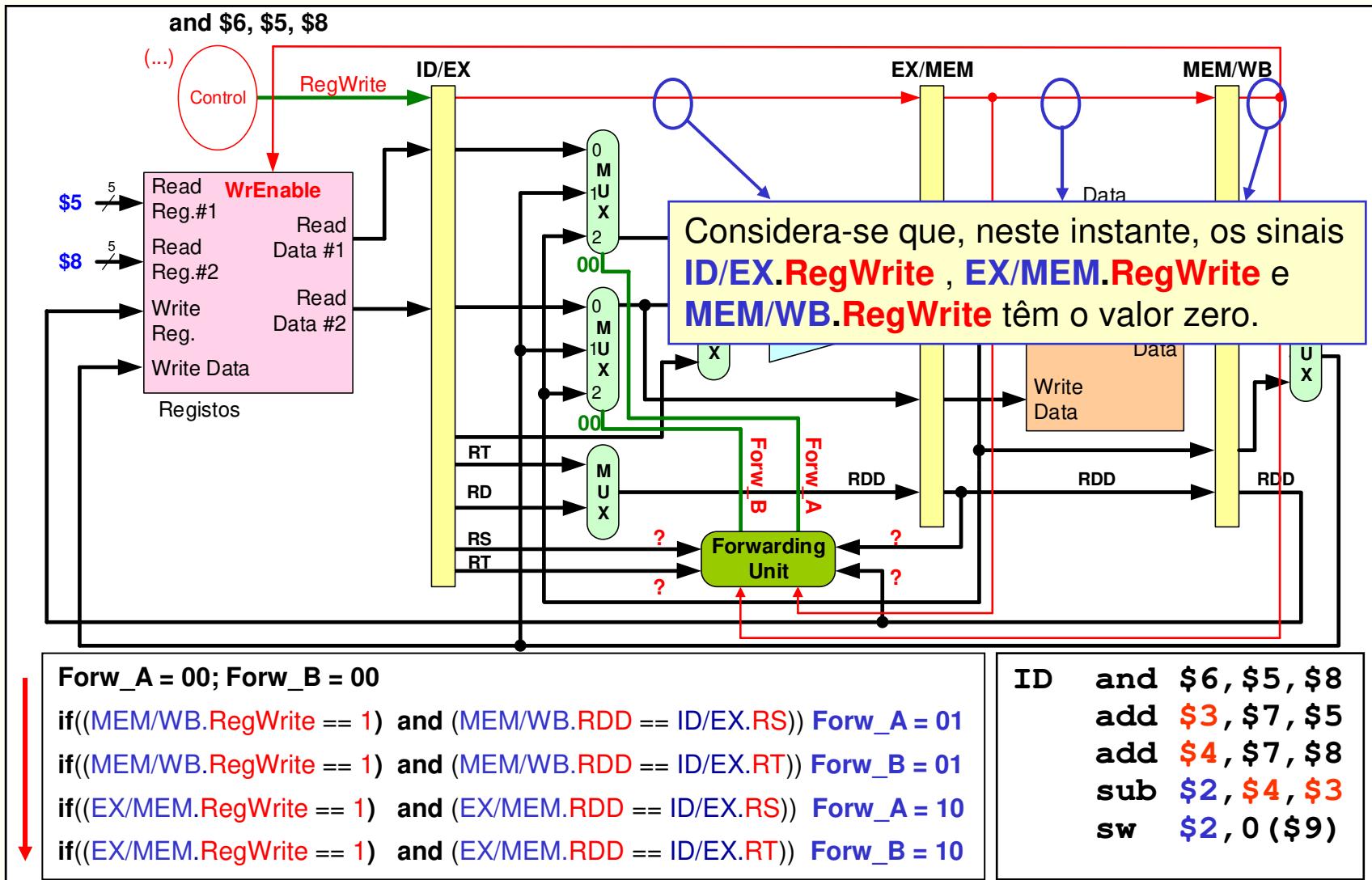
# Exemplo de *forwarding*



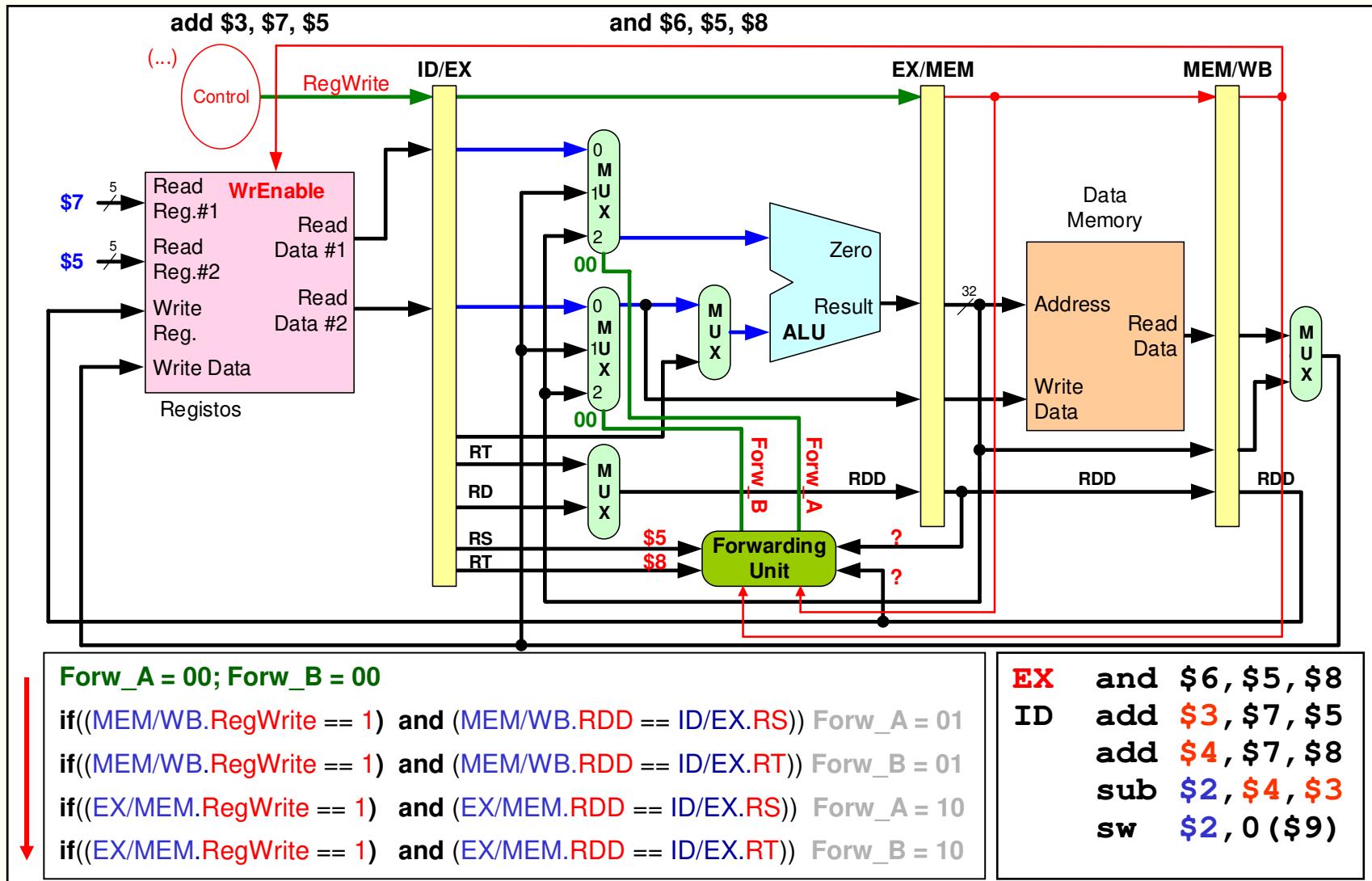
# Exemplo de *forwarding*



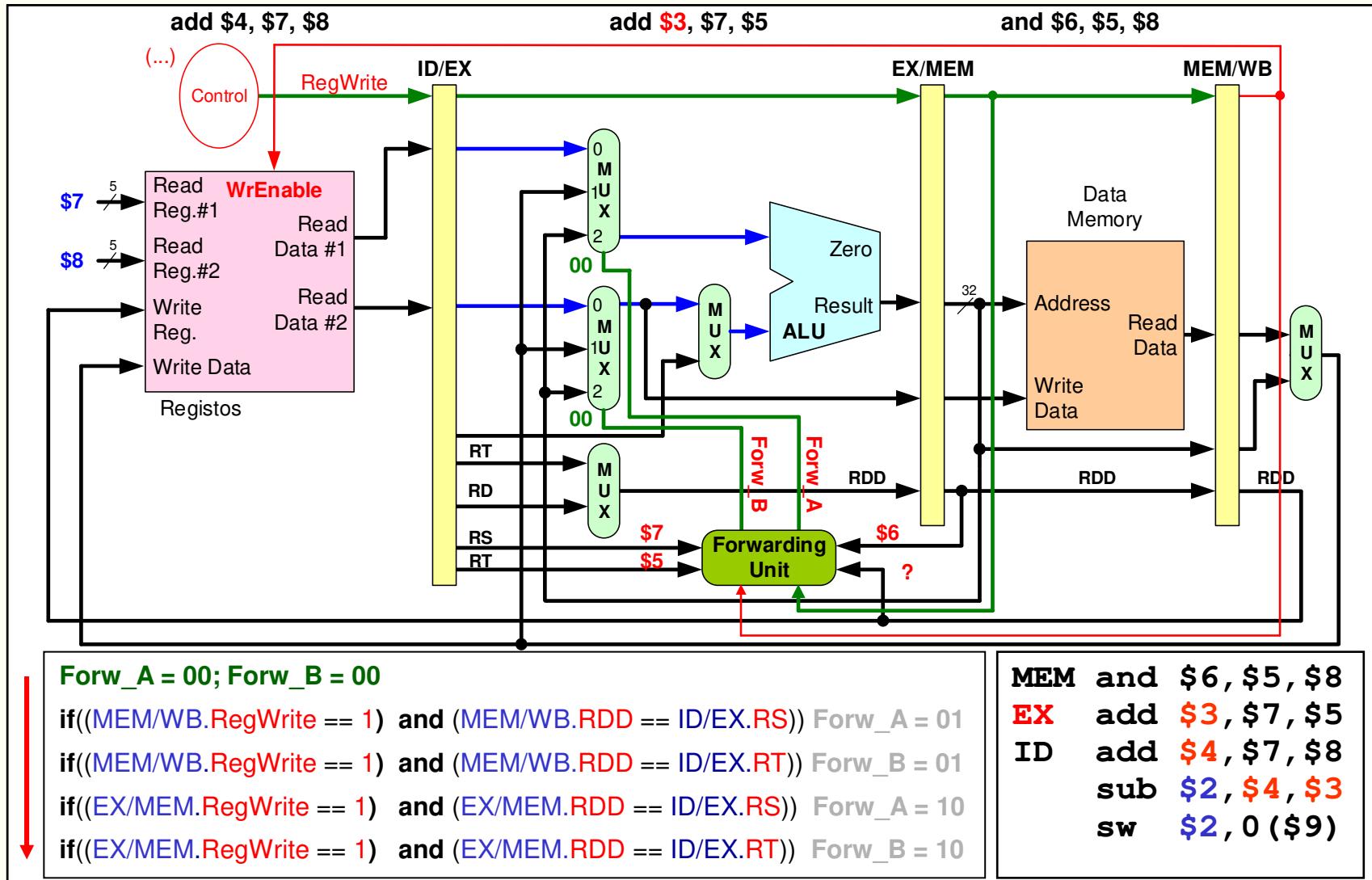
# Exemplo de *forwarding*



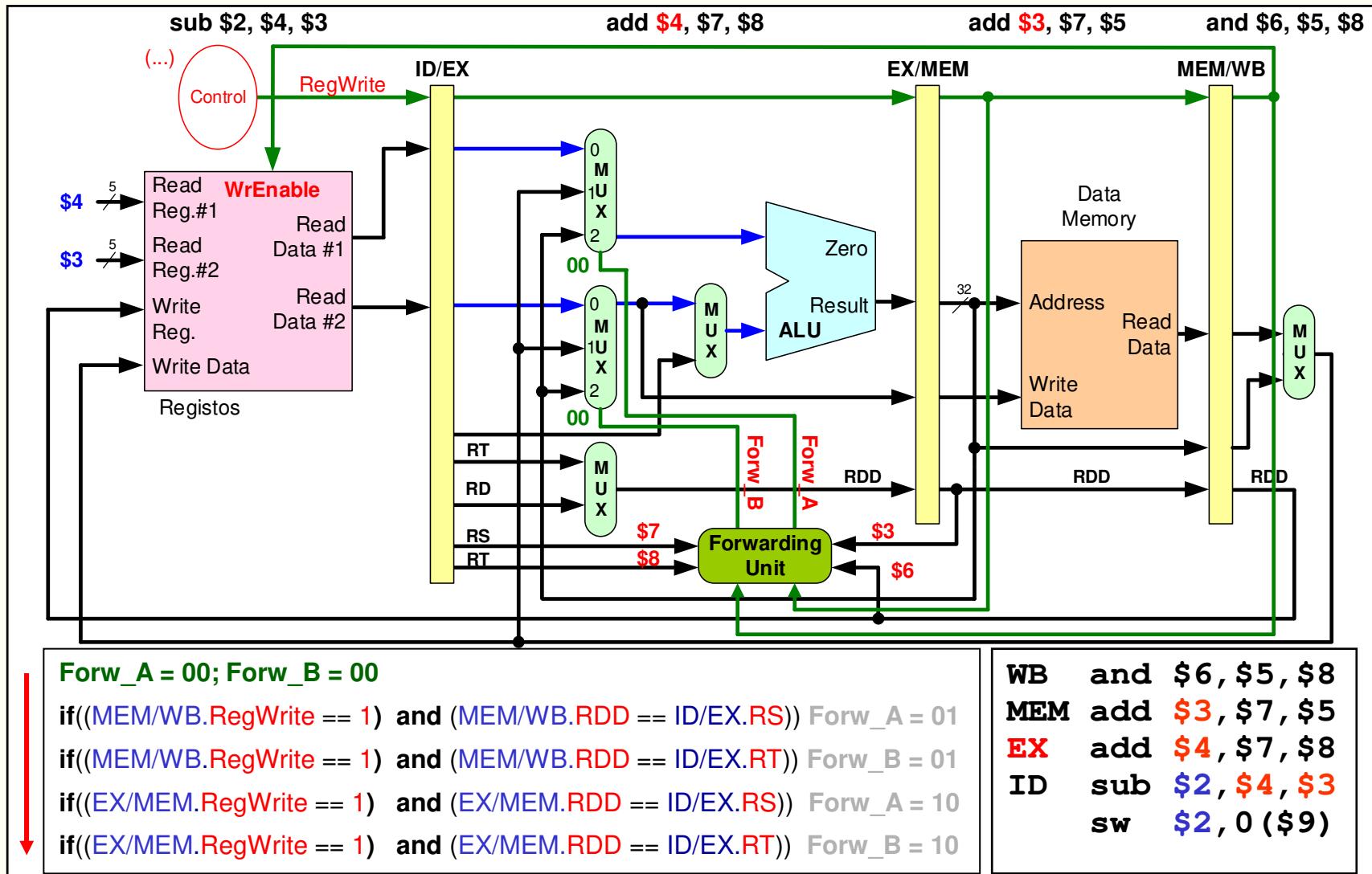
# Exemplo de *forwarding*



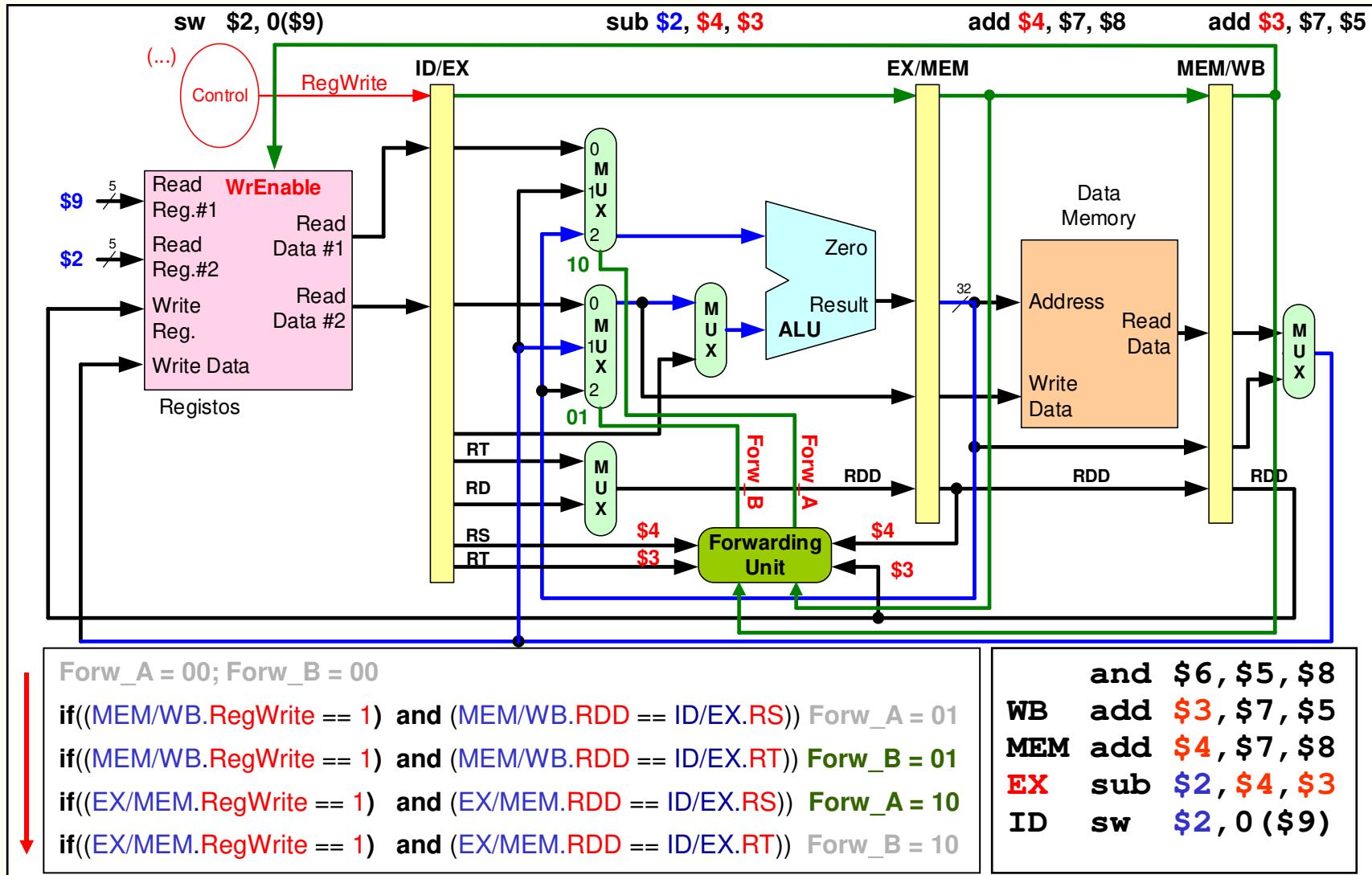
# Exemplo de *forwarding*



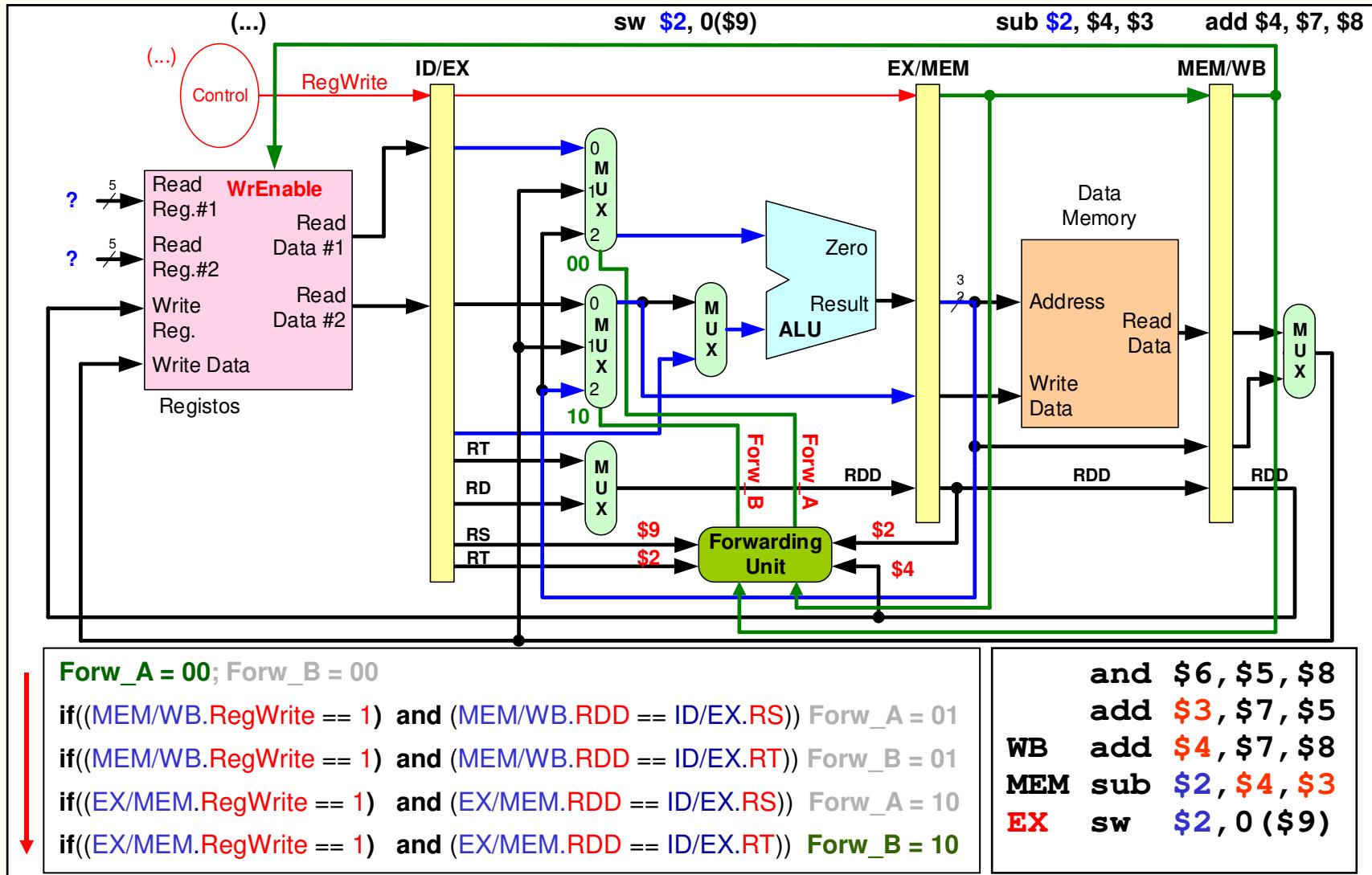
# Exemplo de *forwarding*



# Exemplo de *forwarding*



# Exemplo de *forwarding*



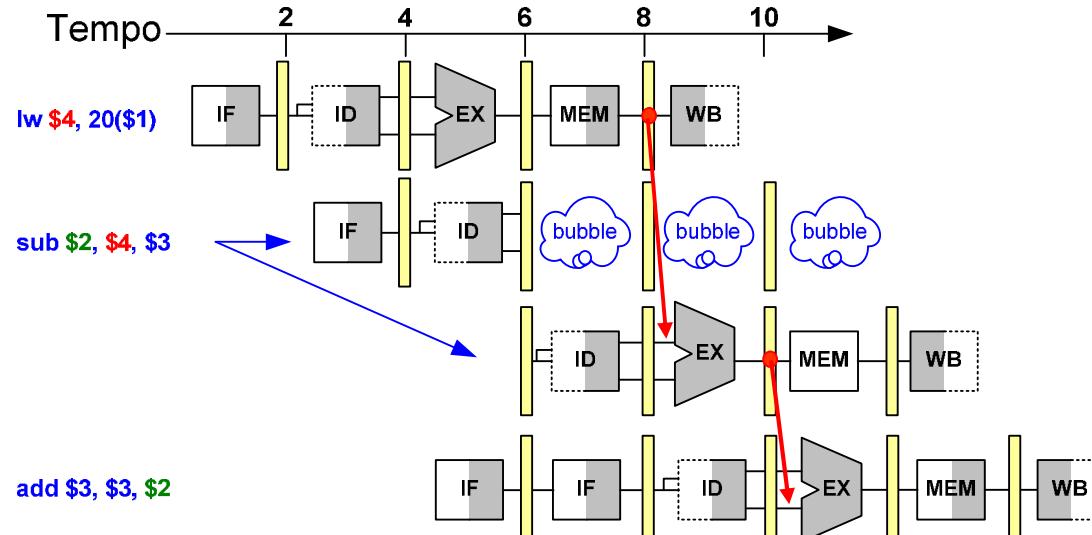
# Dependência que obriga a *stalling*

- Como já observado anteriormente, uma situação em que o *forwarding* não impede a ocorrência de *stalling* é a que resulta de uma instrução aritmética ou lógica executada a seguir e na dependência de uma instrução de load:

**lw \$4, 20(\$1) # valor disponível em WB**

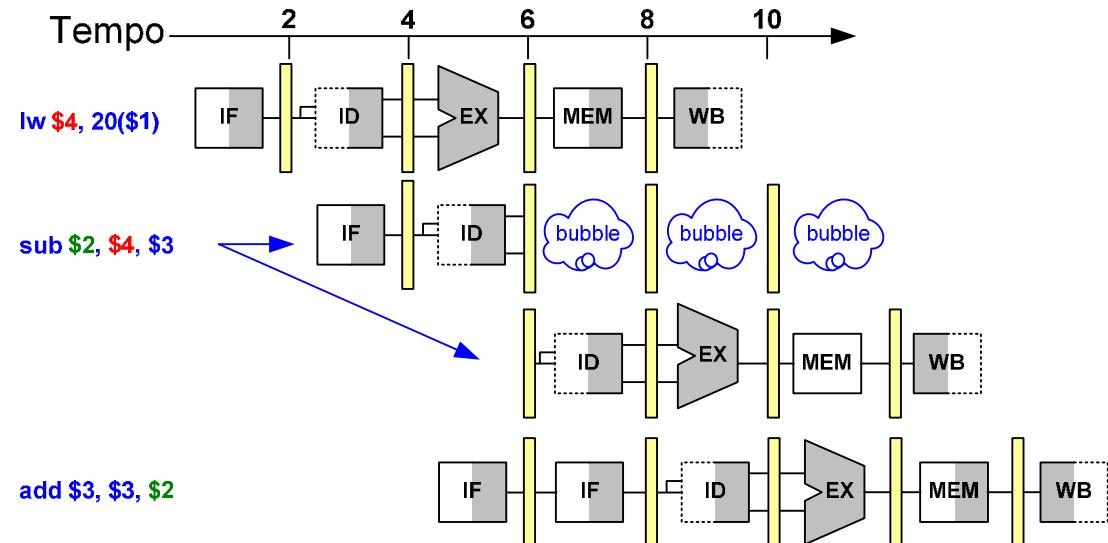
**sub \$2, \$4, \$3 # Stall 1T, Forw. MEM/WB > EX**

**add \$3, \$3, \$2 # Forw. EX/MEM > EX**



# Dependência que obriga a *stalling*

- A situação de *stalling* tem que ser desencadeada quando a instrução tipo R está na sua fase ID. Como fazer?
  - Inserir ***bubble*** na etapa **EX**: fazer o *reset* síncrono do registo **ID/EX**
  - **Congelar, durante 1 ciclo de relógio, as etapas IF e ID** (i.e. impedir a escrita no registo **IF/ID** e impedir que seja feita a atualização do PC)
- Como detetar?

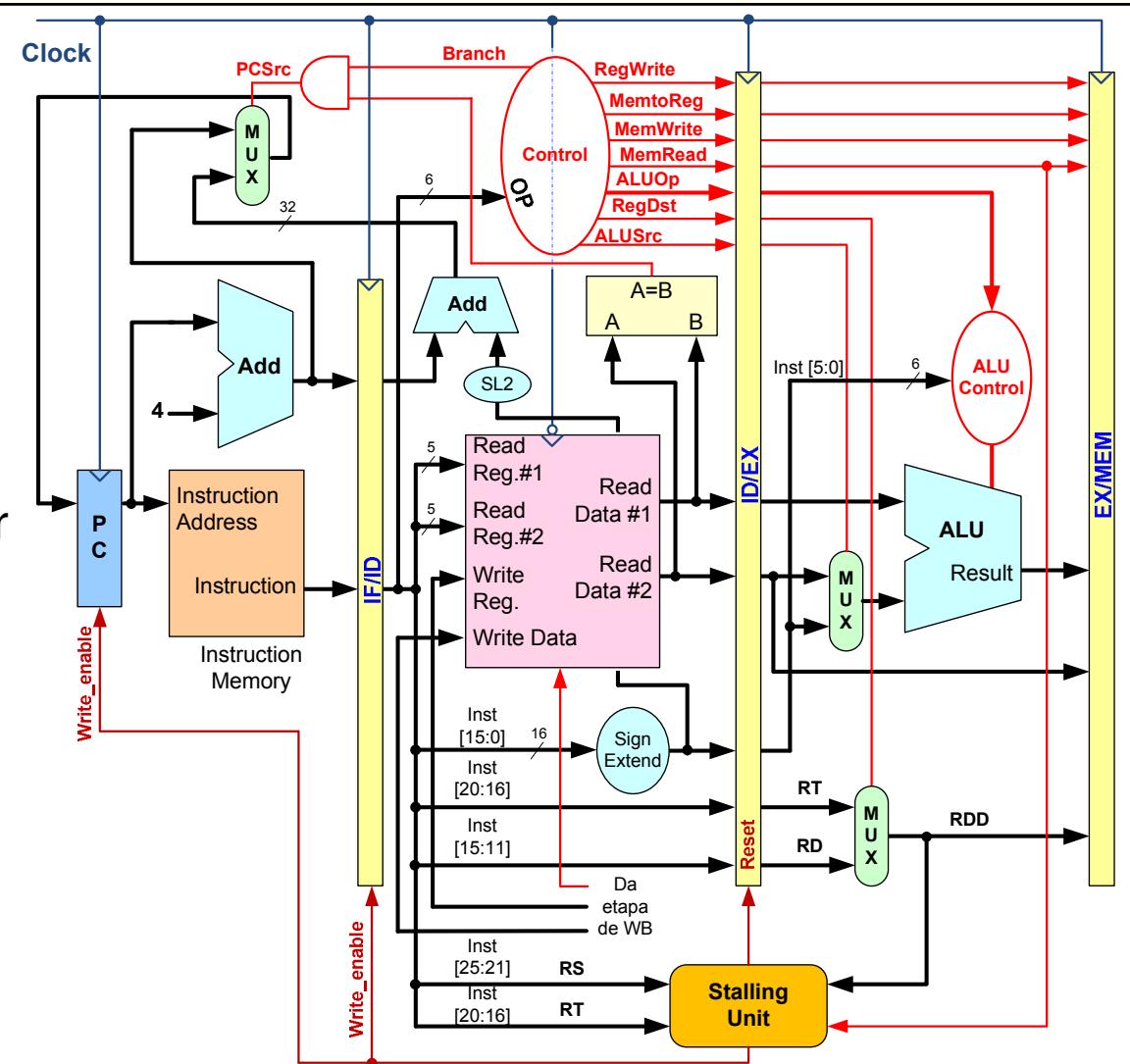


**(ID/EX.MemRead == 1) and (ID/EX.RDD == IF/ID.RS or ID/EX.RDD == IF/ID.RT)**



# Unidade de stalling

- Inserir **bubble** na etapa EX: fazer o *reset* síncrono do registo ID/EX
- Congelar, durante 1 ciclo de relógio, as etapas IF e ID (i.e. impedir a escrita no registo IF/ID e impedir que seja feita a atualização do PC)



# Unidade de controlo de *stalling* – VHDL (v1)

```
library ieee;
use ieee.std_logic_1164.all;

entity StallingUnit is
    port( RS           : in  std_logic_vector(4 downto 0);
          RT           : in  std_logic_vector(4 downto 0);
          RegWrite     : in  std_logic;
          IdEx_RDD    : in  std_logic_vector(4 downto 0);
          IdEx_MemRead: in  std_logic;
          Reset_IdEx  : out std_logic;
          Enable_PC   : out std_logic;
          Enable_IfId  : out std_logic);
end StallingUnit;
```

- Unidade de controlo de *stalling* simplificada, que contempla apenas, de forma incompleta, a situação de dependência entre uma instrução LW e uma instrução tipo R (semelhante ao exemplo apresentado anteriormente)



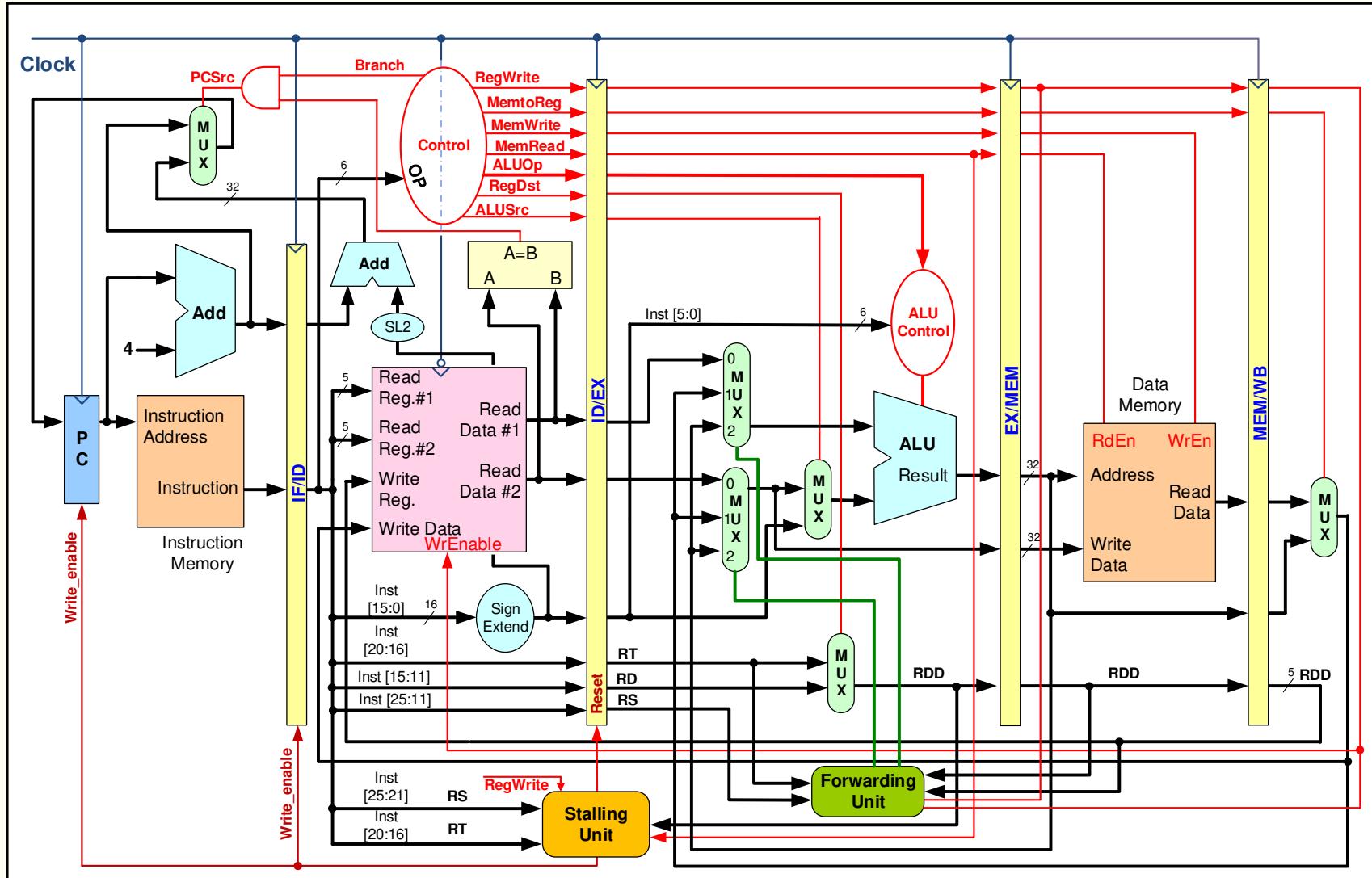
# Unidade de controlo de *stalling* – VHDL (v1)

```
architecture Behavioral of StallingUnit is
begin
    process(all)
    begin
        Enable_PC      <= '1';    -- Normal flow
        Enable_IfId    <= '1';
        Reset_IdEx    <= '0';
        if(IdEx_MemRead = '1' and IdEx_RDD /= "00000") then
            if(RegWrite = '1') then
                if(IdEx_RDD = RS or IdEx_RDD = RT) then
                    Enable_PC      <= '0'; -- Stall PC
                    Enable_IfId    <= '0'; -- Stall IF/ID
                    Reset_IdEx    <= '1'; -- Bubble in ID/EX
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

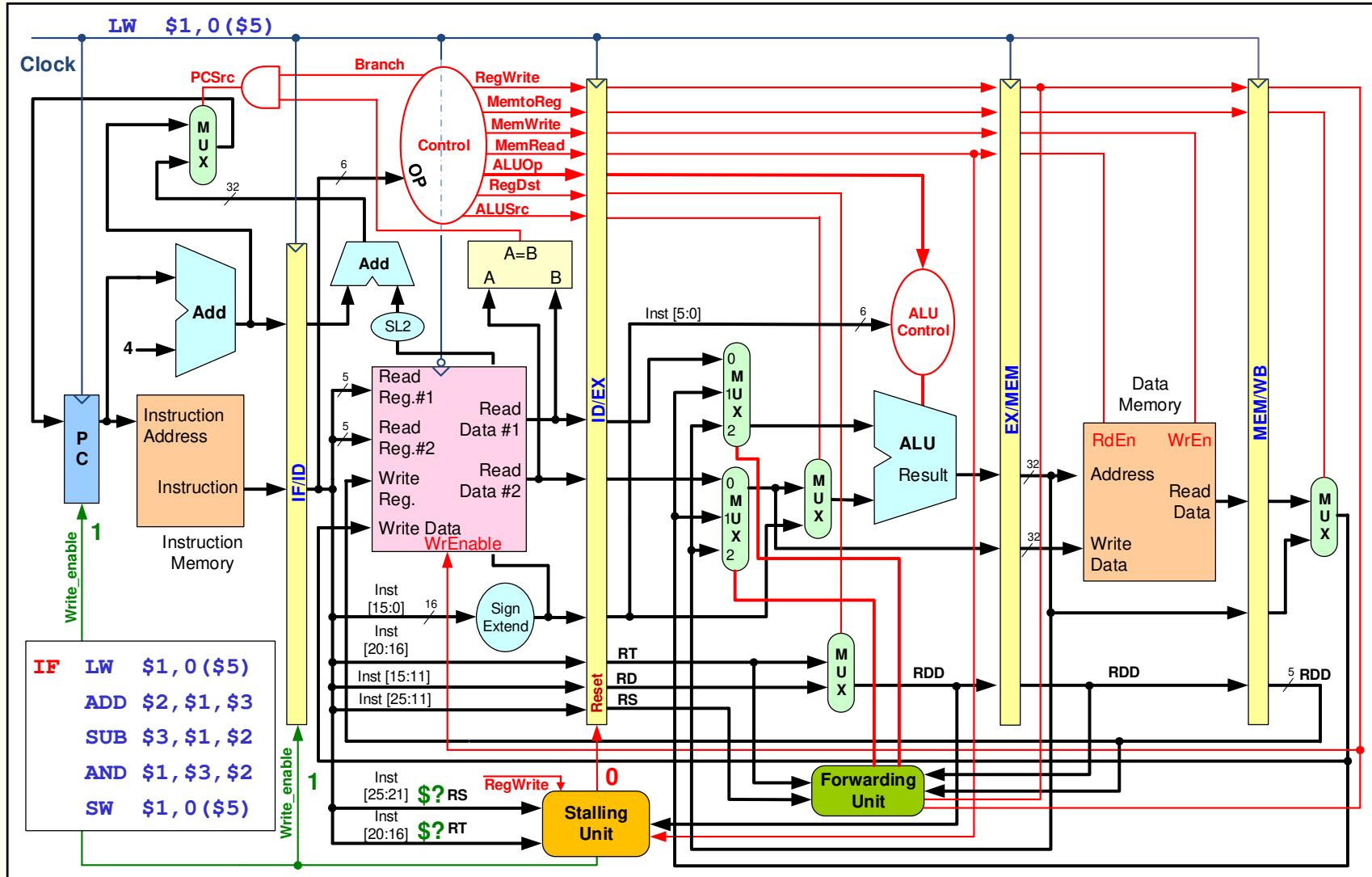
Com as condições explicitadas na estrutura condicional podemos ter a certeza que em ID está uma instrução tipo R?



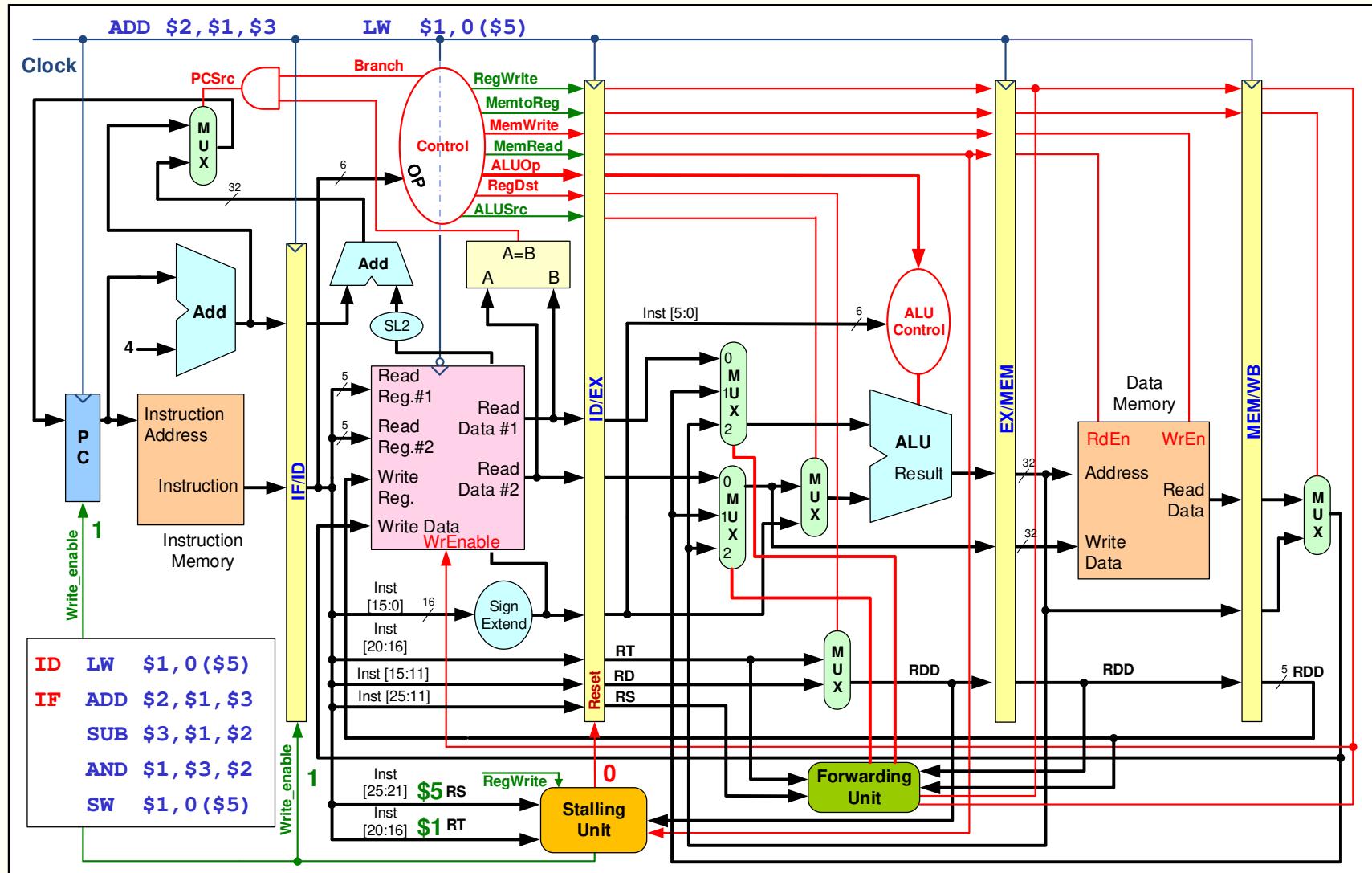
## Datapath pipelining completo (apenas com forwarding para EX)



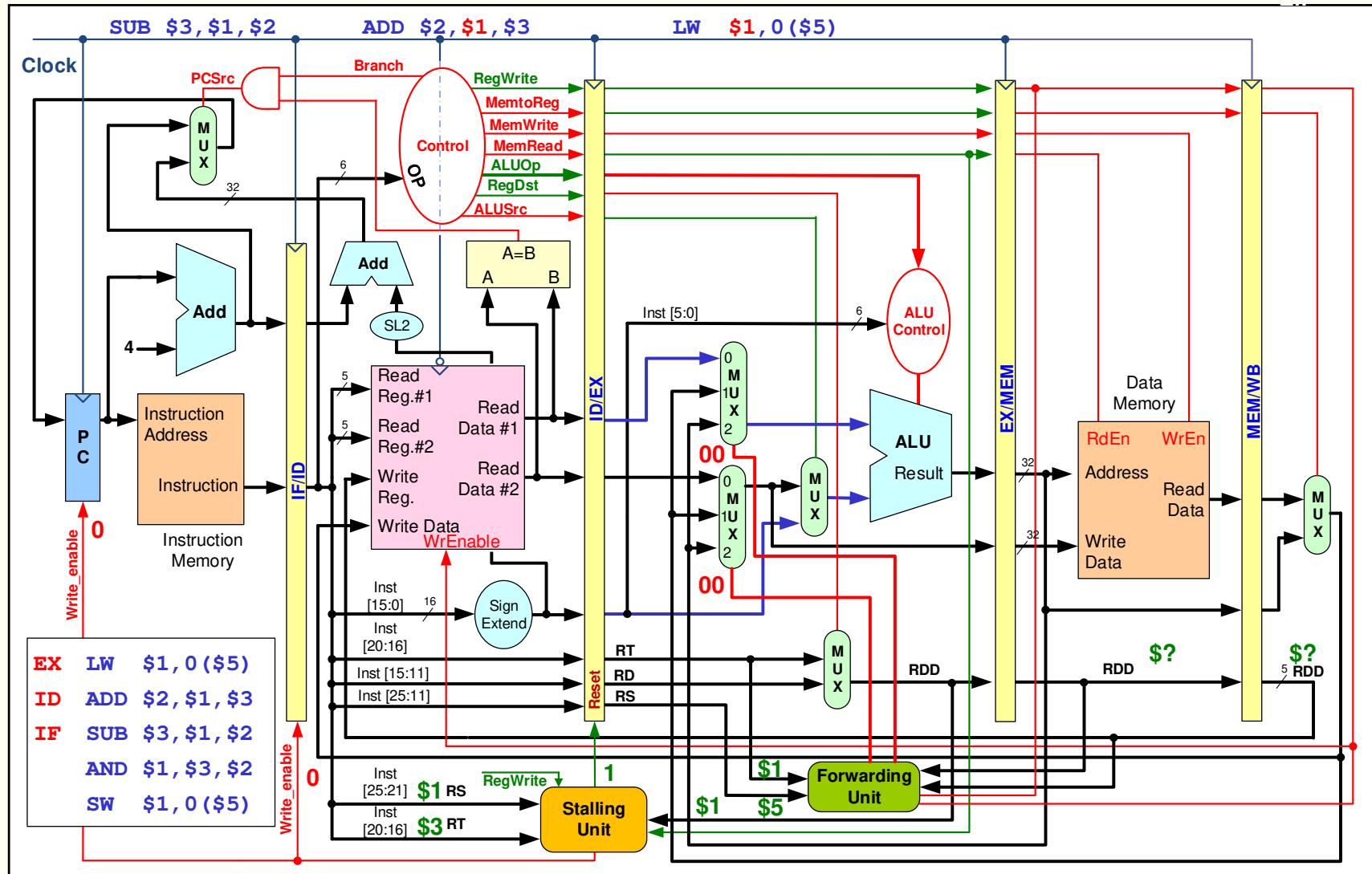
# Datapath pipelining completo – exemplo de execução (1)



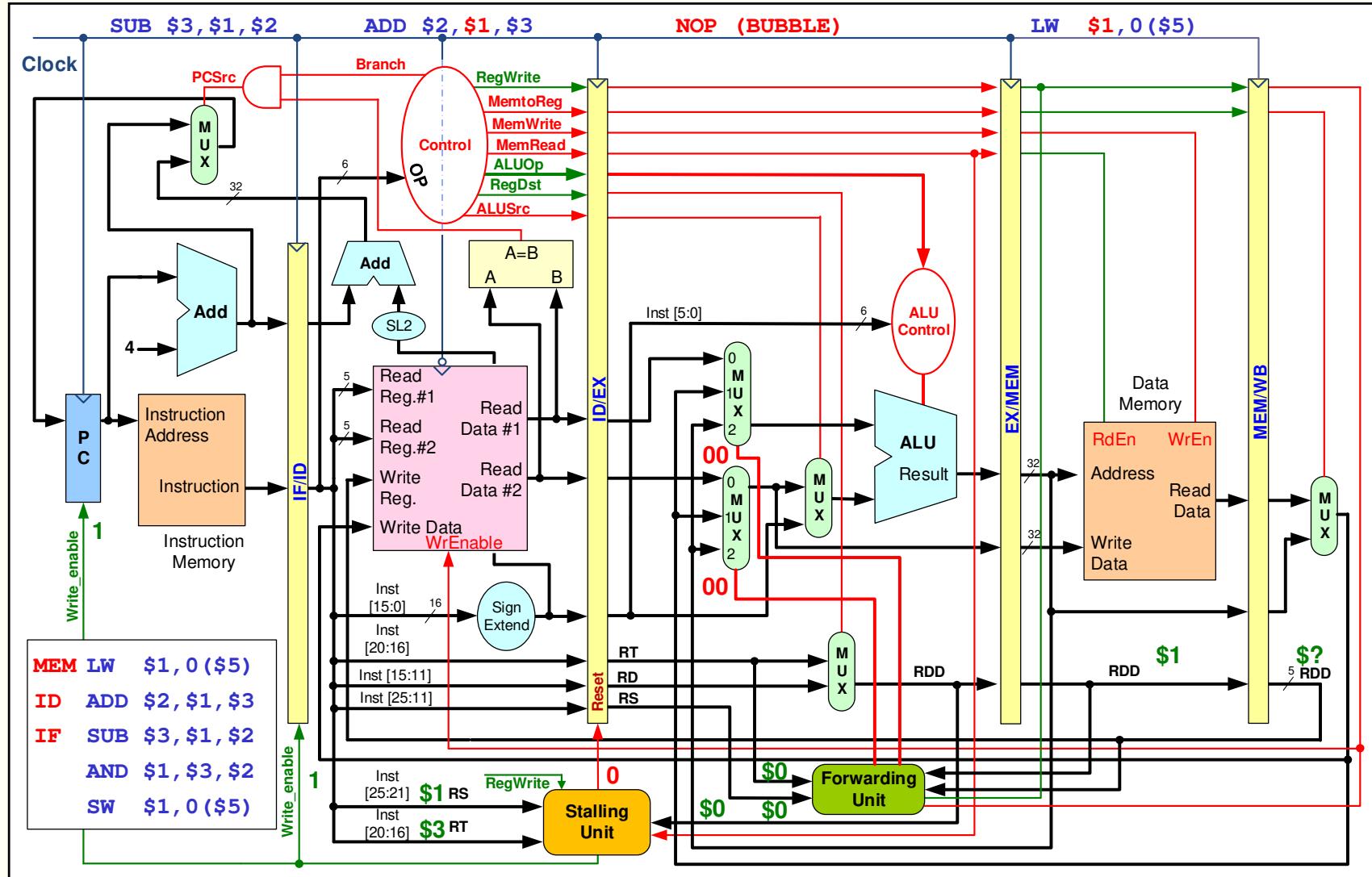
## Exemplo de execução (2) (Normal Flow)



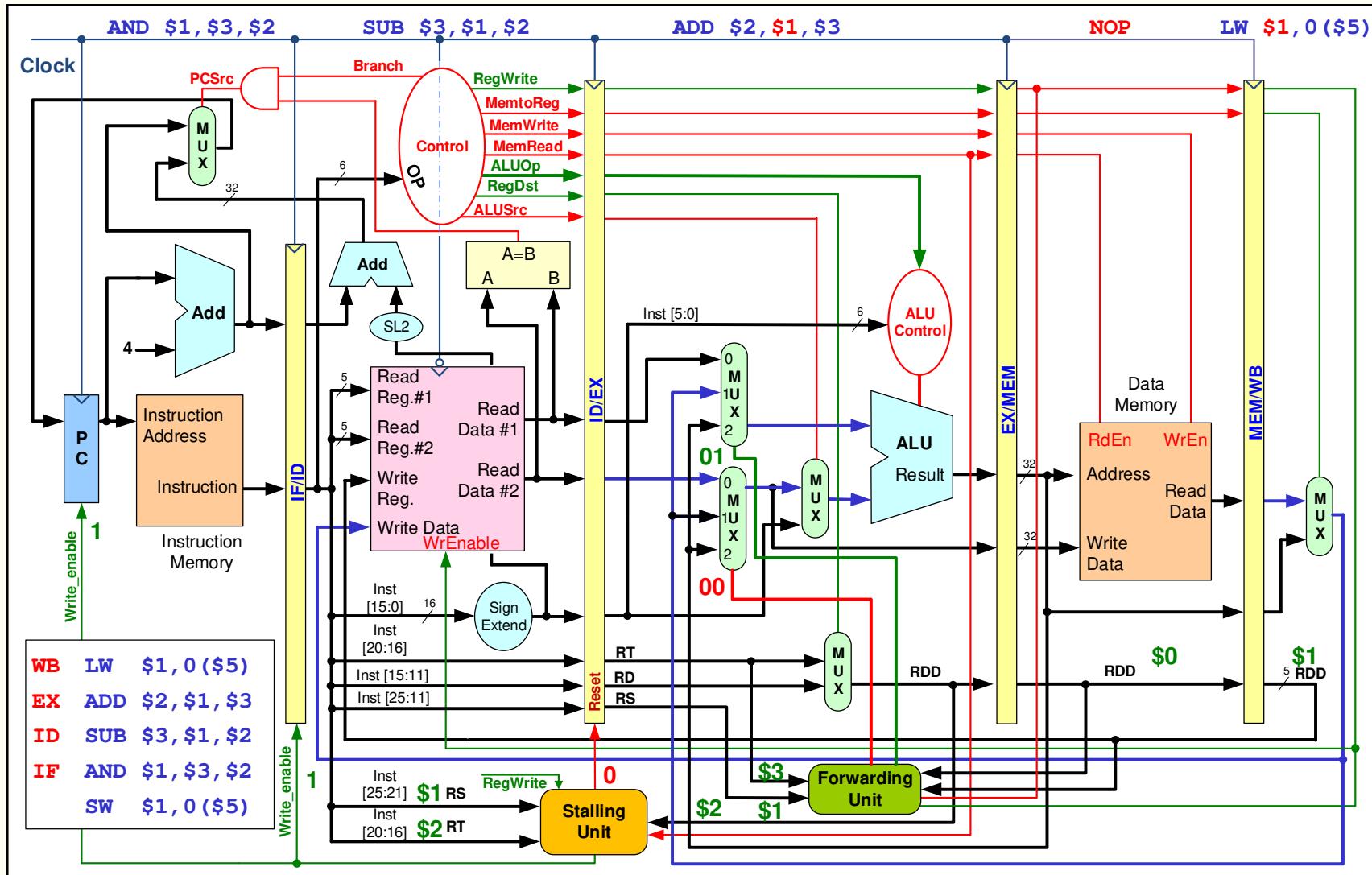
## Exemplo de execução (3) (Normal Flow)



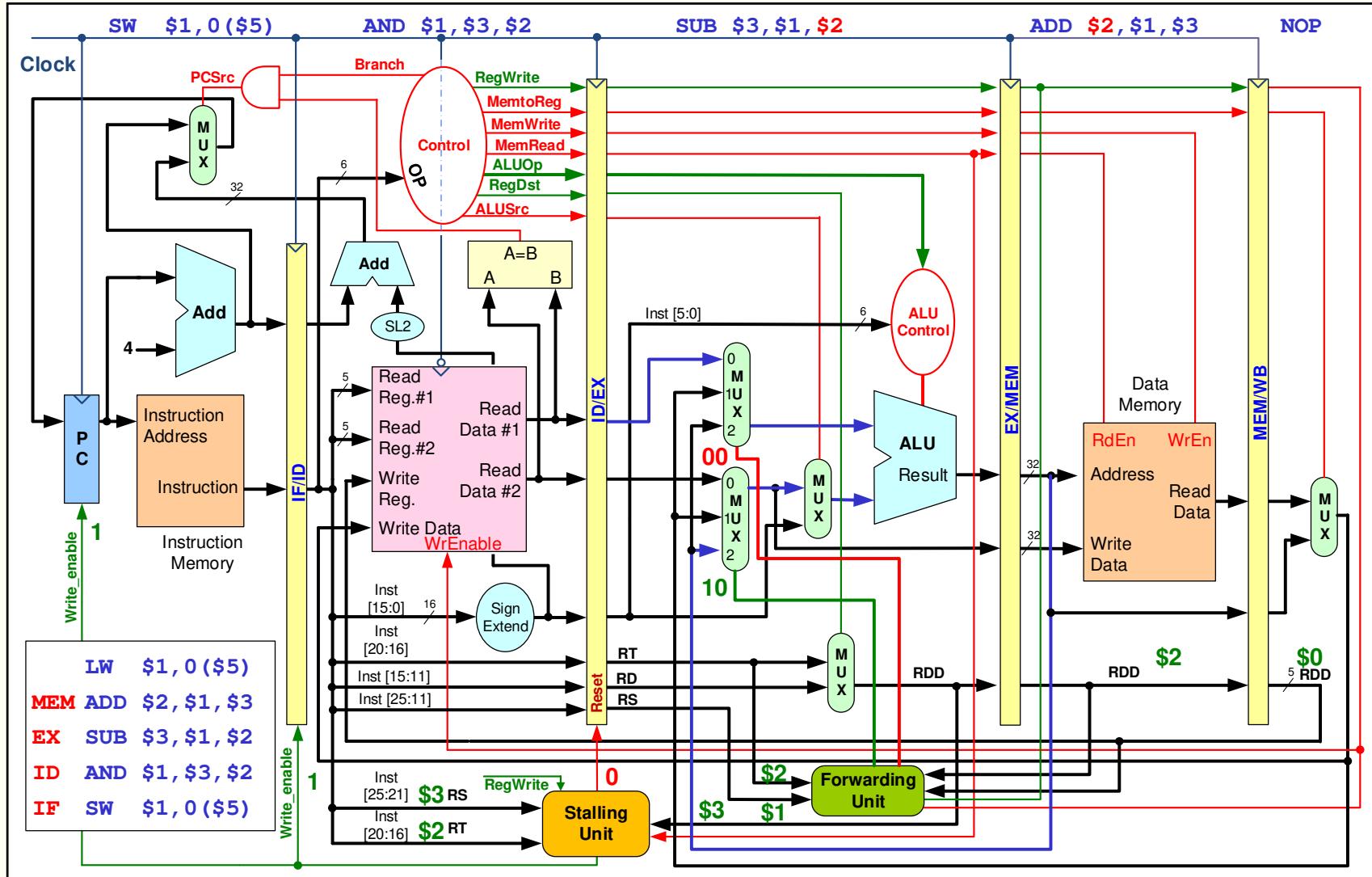
## Exemplo de execução (4) (STALL)



## Exemplo de execução (5) (Fwd: MEM/WB > EX, rs)



## Exemplo de execução (6) (Fwd: EX/MEM > EX, rt)



## Unidade de controlo de *stalling* (v2)

- Para além da sequência descrita anteriormente, há outras situações que obrigam a *stall* do *pipeline*. Exemplos (supondo que a arquitetura apenas implementa *forwarding* para **EX**):

|           |                     |   |
|-----------|---------------------|---|
| <b>lw</b> | <b>\$1, 0 (\$5)</b> | <b>#</b>                                  |
| <b>sw</b> | <b>\$1, 4 (\$4)</b> | <b># Stall 1T, FW MEM/WB &gt; EX (RT)</b> |

|           |                     |   |
|-----------|---------------------|---|
| <b>lw</b> | <b>\$1, 0 (\$3)</b> | <b>#</b>                                  |
| <b>sw</b> | <b>\$4, 8 (\$1)</b> | <b># Stall 1T, FW MEM/WB &gt; EX (RS)</b> |

|             |                       |   |
|-------------|-----------------------|---|
| <b>lw</b>   | <b>\$1, 0 (\$6)</b>   | <b>#</b>                                  |
| <b>addi</b> | <b>\$4, \$1, 0x12</b> | <b># Stall 1T, FW MEM/WB &gt; EX (RS)</b> |

|           |                     |   |
|-----------|---------------------|---|
| <b>lw</b> | <b>\$1, 0 (\$3)</b> | <b>#</b>                                  |
| <b>lw</b> | <b>\$4, 8 (\$1)</b> | <b># Stall 1T, FW MEM/WB &gt; EX (RS)</b> |



# Unidade de controlo de *stalling* – VHDL (v2)

```
library ieee;
use ieee.std_logic_1164.all;
entity StallingUnit is
    port( RS           : in std_logic_vector(4 downto 0);
          RT           : in std_logic_vector(4 downto 0);
          MemWrite     : in std_logic;
          Branch       : in std_logic;
          Jump         : in std_logic;
          AluOp        : in std_logic_vector(1 downto 0);
          IdEx_RDD    : in std_logic_vector(4 downto 0);
          IdEx_MemRead: in std_logic;
          Reset_IdEx  : out std_logic;
          Enable_PC   : out std_logic;
          Enable_IfId  : out std_logic);
end StallingUnit;
```

- Nesta versão da unidade de controlo de *stalling* é gerada uma *bubble* em todos os casos de dependência entre um LW e qualquer outra instrução à exceção de BEQ.



# Unidade de controlo de *stalling* – VHDL (v2)

```
architecture Behavioral of StallingUnit is
begin
    process(all)
    begin
        Enable_PC <= '1'; Enable_IfId <= '1'; Reset_IdEx <= '0';
        if(Branch = '0' and Jump = '0') then
            if(IdEx_MemRead = '1' and IdEx_RDD /= "00000") then
                if(RS = IdEx_RDD or -- R-type/lw/sw/addi/slti in ID (rs)
                   (RT = IdEx_RDD and AluOp = "10") or --R-type in ID (rt)
                   (RT = IdEx_RDD and MemWrite='1')) then --sw in ID (rt)
                    Enable_PC      <= '0';   -- Stall PC
                    Enable_IfId    <= '0';   -- Stall IF/ID
                    Reset_IdEx    <= '1';   -- Bubble in ID/EX
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```



# Unidade de controlo de *stalling* (v3)

- Mesmo supondo que a arquitetura implementa *forwarding* para **ID** (**EX/MEM** > **ID**) persistem situações em que há necessidade de fazer *stall* ao *pipeline*.
- Exemplos:

```
add    $1,$2,$3    #
beq    $1,$5,lab  # Stall 1T, FW EX/MEM > ID (RS)
```

```
addi   $1,$3,0x25 #
beq    $5,$1,lab  # Stall 1T, FW EX/MEM > ID (RT)
```

```
lw     $1,0($5)   #
beq    $1,$2,lab  # Stall 2T
```



# Unidade de controlo de *stalling* – VHDL (v3)

```
library ieee;
use ieee.std_logic_1164.all;

entity StallingUnit is
    port( RS           : in  std_logic_vector(4 downto 0);
          RT           : in  std_logic_vector(4 downto 0);
          MemWrite     : in  std_logic;
          AluOp        : in  std_logic_vector(1 downto 0);
          Branch       : in  std_logic;
          Jump         : in  std_logic;
          IdEx_RDD     : in  std_logic_vector(4 downto 0);
          IdEx_MemRead : in  std_logic;
          IdEx_RegWrite: in  std_logic;
          ExMem_RDD    : in  std_logic_vector(4 downto 0);
          ExMem_MemRead: in  std_logic;
          Reset_IdEx   : out std_logic;
          Enable_PC    : out std_logic;
          Enable_IfId   : out std_logic);
end StallingUnit;
```



# Unidade de controlo de stalling – VHDL (v3)

```
architecture Behavioral of StallingUnit is
begin
  process(all)
  begin
    Enable_PC <= '1'; Enable_IfId <= '1'; Reset_IdEx <= '0';
    if(branch = '1') then -- Branch instruction in ID
      if(IdEx_RegWrite = '1' and IdEx_RDD /= "00000") then
        if(RS = IdEx_RDD or RT = IdEx_RDD) then
          -- Stall
          Enable_PC <= '0'; Enable_IfId <= '0'; Reset_IdEx <= '1';
        end if;
      end if;
      if(ExMem_MemRead = '1' and ExMem_RDD /= "00000") then
        if(RS = ExMem_RDD or RT = ExMem_RDD) then
          -- Stall
          Enable_PC <= '0'; Enable_IfId <= '0'; Reset_IdEx <= '1';
        end if;
      end if;
    elsif(jump = '0') then -- R-type/lw/sw/addi/slti in ID
      -- see next page
```



# Unidade de controlo de *stalling* – VHDL (v3)

```
elsif(jump = '0') then -- R-type/LW/SW/ADDI/SLTI in ID
    if(IdEx_MemRead = '1' and IdEx_RDD /= "00000") then
        if(RS = IdEx_RDD or (RT = IdEx_RDD and AluOp = "10"))then
            -- Stall
            Enable_PC <= '0'; Enable_IfId <= '0'; Reset_IdEx <= '1';
        end if;
        if(RT = IdEx_RDD and MemWrite = '1') then
            -- Stall
            Enable_PC <= '0'; Enable_IfId <= '0'; Reset_IdEx <= '1';
        end if;
    end if;
end process;
end Behavioral;
```

Unidade de controlo de *stalling* completa (para o conjunto de instruções considerado), considerando que a arquitetura implementa *forwarding* para ID e para EX (e não para MEM)



## Forwarding de **MEM/WB** para **MEM**

- Nos slides anteriores considerou-se que uma dependência originada por uma sequência do tipo:

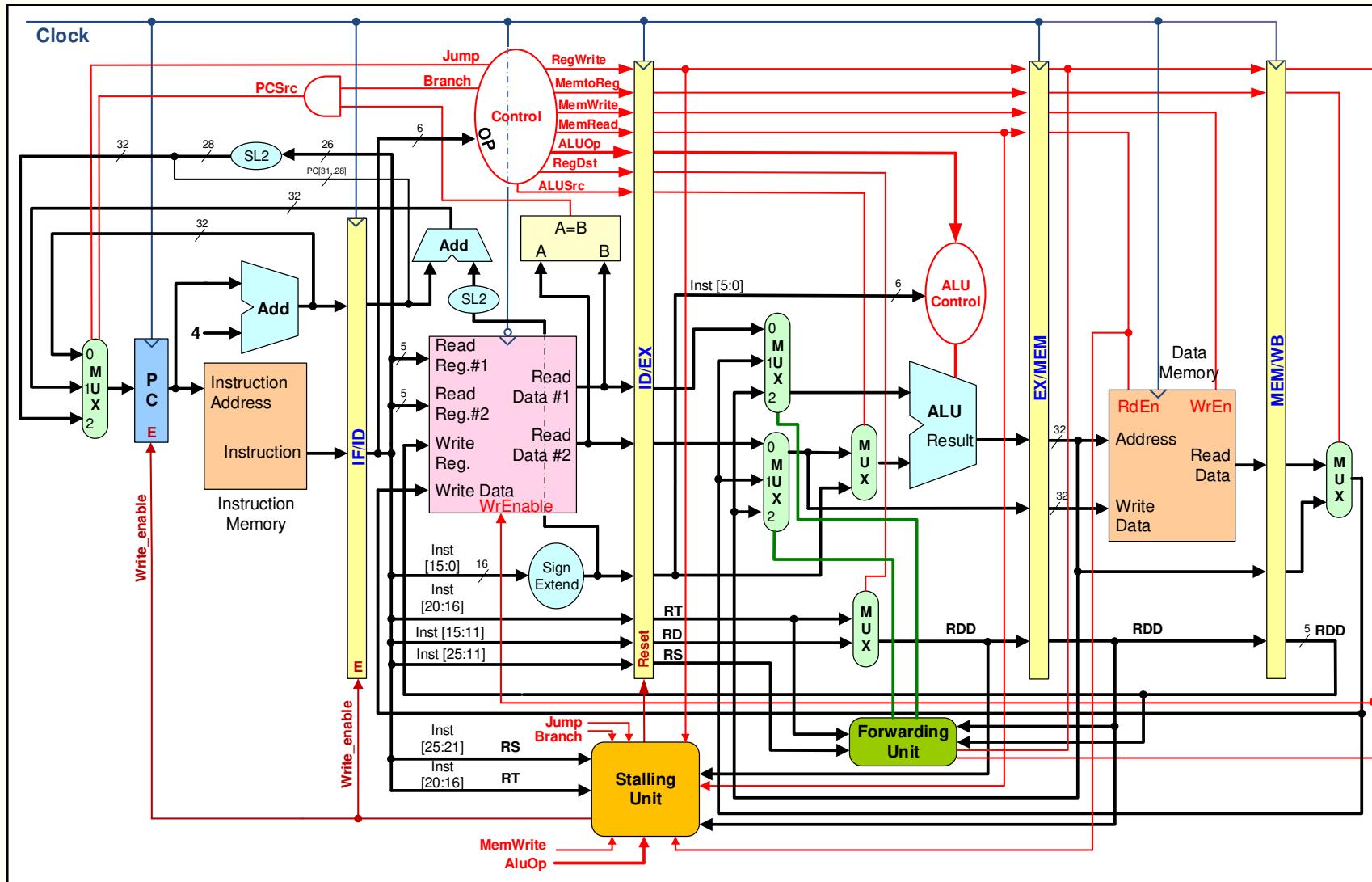
```
lw      $1, 0($5) #  
sw      $1, 4($4) # Stall 1T, FW MEM/WB > EX (RT)
```

é resolvida com *stall* durante 1 ciclo de relógio seguido de *forwarding* de **MEM/WB** para **EX**.

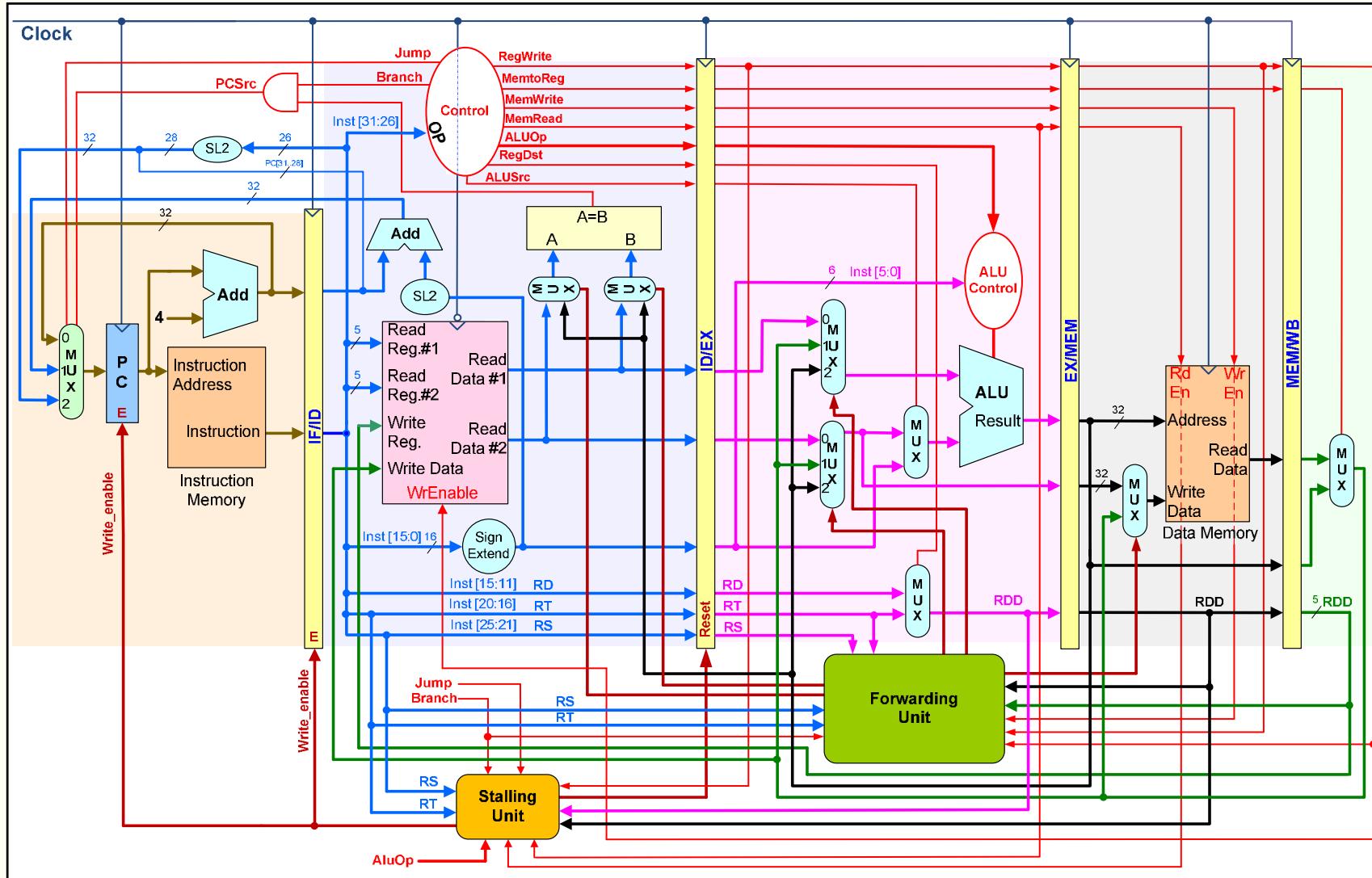
- Será mesmo necessário fazer o *stall* do pipeline?
- A instrução **sw** só necessita do valor de **\$1** no estágio **MEM** (**\$4** é necessário em **EX**), situação em que a instrução **lw** já se encontra em **WB**
- Esta situação particular pode então ser resolvida com *forwarding* de **MEM/WB** para **MEM**, evitando-se o *stall* do pipeline



# Datapath pipelining com Jump



## Datapath pipelining completo, com forwarding para MEM, EXE e ID



# Exercício 1

- Determine o número de ciclos de relógio que o trecho de código seguinte demora a executar num pipeline de 5 fases, desde o instante em que é feito o *Instruction Fetch* da 1<sup>a</sup> instrução, até à conclusão da última:

```
add  $1,$2,$3
lw   $2,0($4)
sub $3,$4,$3
addi $4,$4,4
and $5,$1,$5      # "and" em ID, "add" já terminou
sw   $2,0($1)      # "sw" em ID, "add" e "lw"
                  # já terminaram
```

$$\begin{aligned}\text{Nr_Cycles} &= F + (\text{Number\_of\_executed\_instructions} - 1) \\ &= 5 + (6 - 1) = 10 \text{ T}\end{aligned}$$

Num datapath *single-cycle* o mesmo código demoraria 6 ciclos de relógio a executar. Porque razão é a execução no *datapath pipelined* mais rápida? Quantos ciclos de relógio demora a execução num *datapath multi-cycle*?



## Exercício 2a

- Para o trecho de código seguinte identifique todas as situações de *hazard* de dados e de controlo que ocorrem na execução num *pipeline* de 5 fases, com *branches* resolvidos em ID:

```
main: lw    $1, 0($0)    #
      add   $4, $0, $0    #
      lw    $2, 4($0)    #
loop:  lw    $3, 0($1)    #
      add   $4, $4, $3    # hazard de dados ($3)
      sw    $4, 36($1)   # hazard de dados ($4)
      addiu $1, $1, 4     #
      sltu $5, $1, $2     # hazard de dados ($1)
      bne  $5, $0, loop  # haz. dados ($5) / haz. controlo
      sw    $4, 8($0)    #
      lw    $1, 12($0)   #
```



## Exercício 2b

- Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o **pipeline não implementa forwarding**:

```
main: lw    $1, 0($0)  #
      add   $4, $0, $0  #
      lw    $2, 4($0)  #
loop:  lw    $3, 0($1)  #
      add   $4, $4, $3  # Stall 2T
      sw    $4, 36($1)  # Stall 2T
      addiu $1, $1, 4   #
      sltu  $5, $1, $2   # Stall 2T
      bne   $5, $0, loop # Stall 2T
      sw    $4, 8($0)  #
      lw    $1, 12($0) #
```



## Exercício 2c

- Calcule o número de ciclos de relógio que o programa anterior demora a executar num *pipeline* de 5 fases, **sem forwarding, com branches resolvidos em ID e delayed branch**, desde o IF da 1<sup>a</sup> instrução até à conclusão da última instrução

```
main: lw    $1, 0($0)  # $1=0x10
      add   $4, $0, $0  # $4=0
      lw    $2, 4($0)  # $2=0x20
loop: lw    $3, 0($1)  #
      add   $4, $4, $3  # Stall 2T
      sw    $4, 36($1)  # Stall 2T
      addiu $1, $1, 4   #
      sltu $5, $1, $2   # Stall 2T
      bne  $5, $0, loop # Stall 2T
      sw    $4, 8($0)   #
      lw    $1, 12($0)  #
```

| Memória de dados |       |
|------------------|-------|
| Addr             | Value |
| 0x0000000        | 0x10  |
| 0x0000004        | 0x20  |

- O ciclo é executado 4 vezes:  $\$1 \in [0x10, 0x20[$
- Nr de instruções executadas no ciclo:  $4 * 7 = 28$
- Nr de instruções executadas fora do ciclo:  $3 + 1 = 4$
- Nr de cycle stalls =  $4 * 8 = 32$

$$\begin{aligned} \text{Nr_cycles} &= F + (\text{Nr_instructions} - 1) + \text{Nr_Cycle_Stalls} \\ &= 5 + (28 + 4 - 1) + 32 = 68 \text{ T} \end{aligned}$$



## Exercício 2d

- Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o **pipeline implementa forwarding para EX e para ID**:

```
main: lw      $1,0($0)    #
          add    $4,$0,$0    #
          lw      $2,4($0)    #
loop:  lw      $3,0($1)    #
          add    $4,$4,$3    # Stall 1T, FW MEM/WB > EX (RT)
          sw      $4,36($1)   # FW EX/MEM > EX (RT)
          addiu $1,$1,4     #
          sltu  $5,$1,$2     # FW EX/MEM > EX (RS)
          bne   $5,$0,loop # Stall 1T, FW EX/MEM > ID (RS)
          sw      $4,8($0)    #
          lw      $1,12($0)   #
```



## Exercício 2e

- Calcule o número de ciclos de relógio que o programa anterior demora a executar num *pipeline* de 5 fases, **com forwarding para EX e para ID, com branches resolvidos em ID e delayed branch**, desde o IF da 1<sup>a</sup> instrução até à conclusão da última instrução

```
main: lw    $1, 0($0)  #
      add   $4, $0, $0  #
      lw    $2, 4($0)  #
loop: lw    $3, 0($1)  #
      add   $4, $4, $3  # Stall 1T, FW MEM/WB > EX (RT)
      sw    $4, 36($1)  # FW EX/MEM > EX (RT)
      addiu $1, $1, 4   #
      sltu $5, $1, $2   # FW EX/MEM > EX (RS)
      bne  $5, $0, loop # Stall 1T, FW EX/MEM > ID (RS)
      sw    $4, 8($0)  #
      lw    $1, 12($0) #
```

$$\text{Nr\_cycles} = F + (\text{Nr\_instructions}-1) + \text{Nr\_Cycle\_Stalls}$$

$$= 5 + (28 + 4 - 1) + 8 = 44 \text{ T} \quad (\text{nr of cycle stalls} = 4 * 2 = 8 \text{ T})$$

