

Aula 1

- Conceitos fundamentais em Arquitetura de Computadores
 - Arquitetura básica de um sistema computacional
 - Arquitetura básica do CPU
 - O ciclo de execução de uma instrução
 - Níveis de representação
 - Codificação de instruções
 - *Instruction Set Architecture* (ISA)
 - Classes de instruções

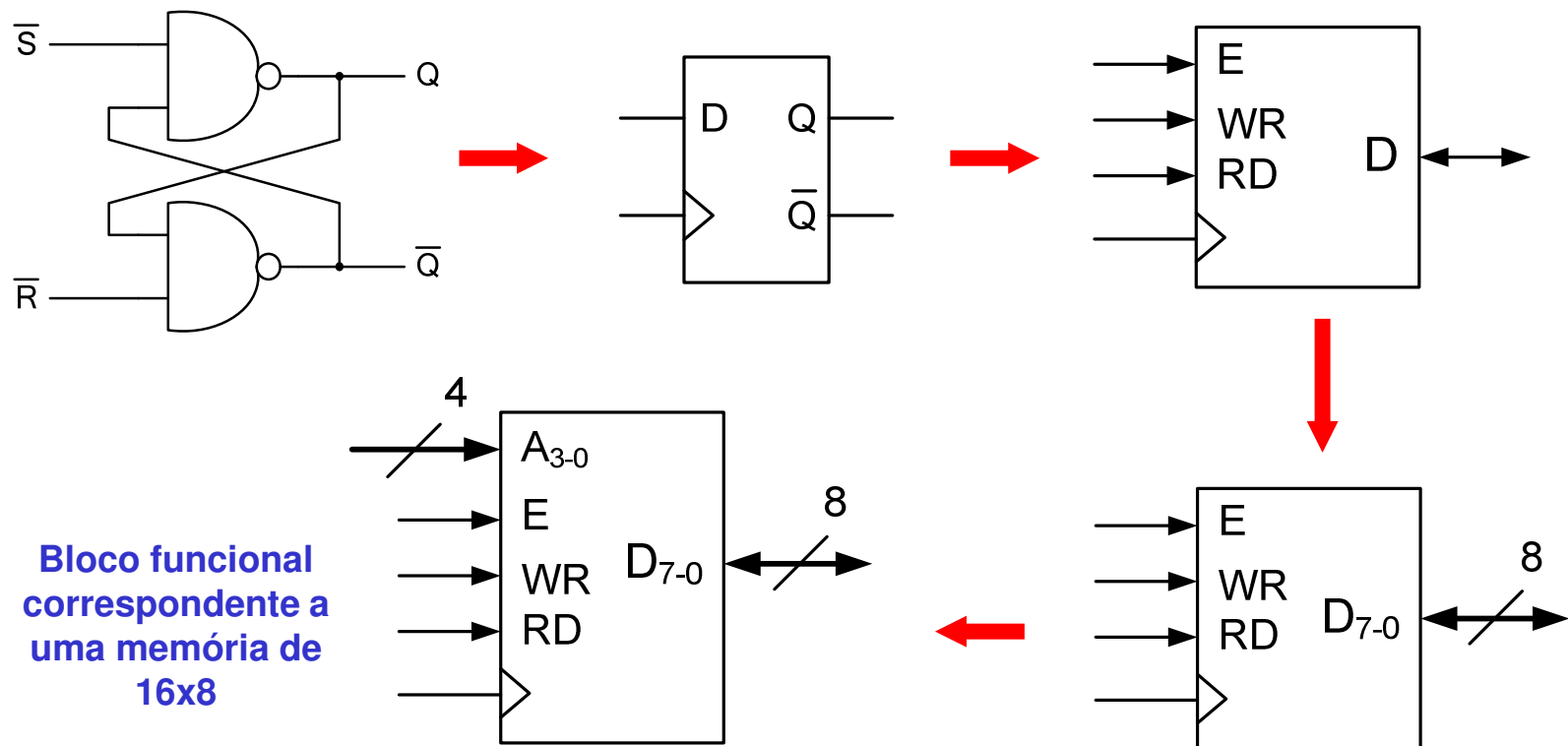
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

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**

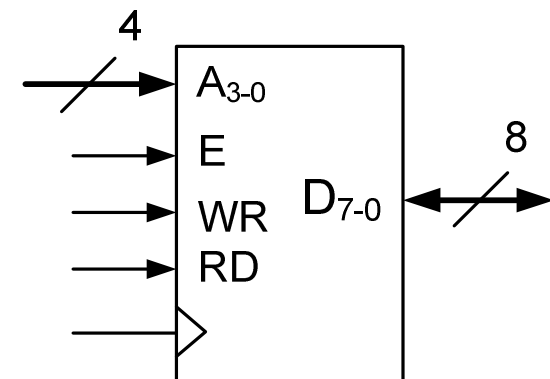


Exemplo: memória RAM 16x8 – VHDL

- 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;
```

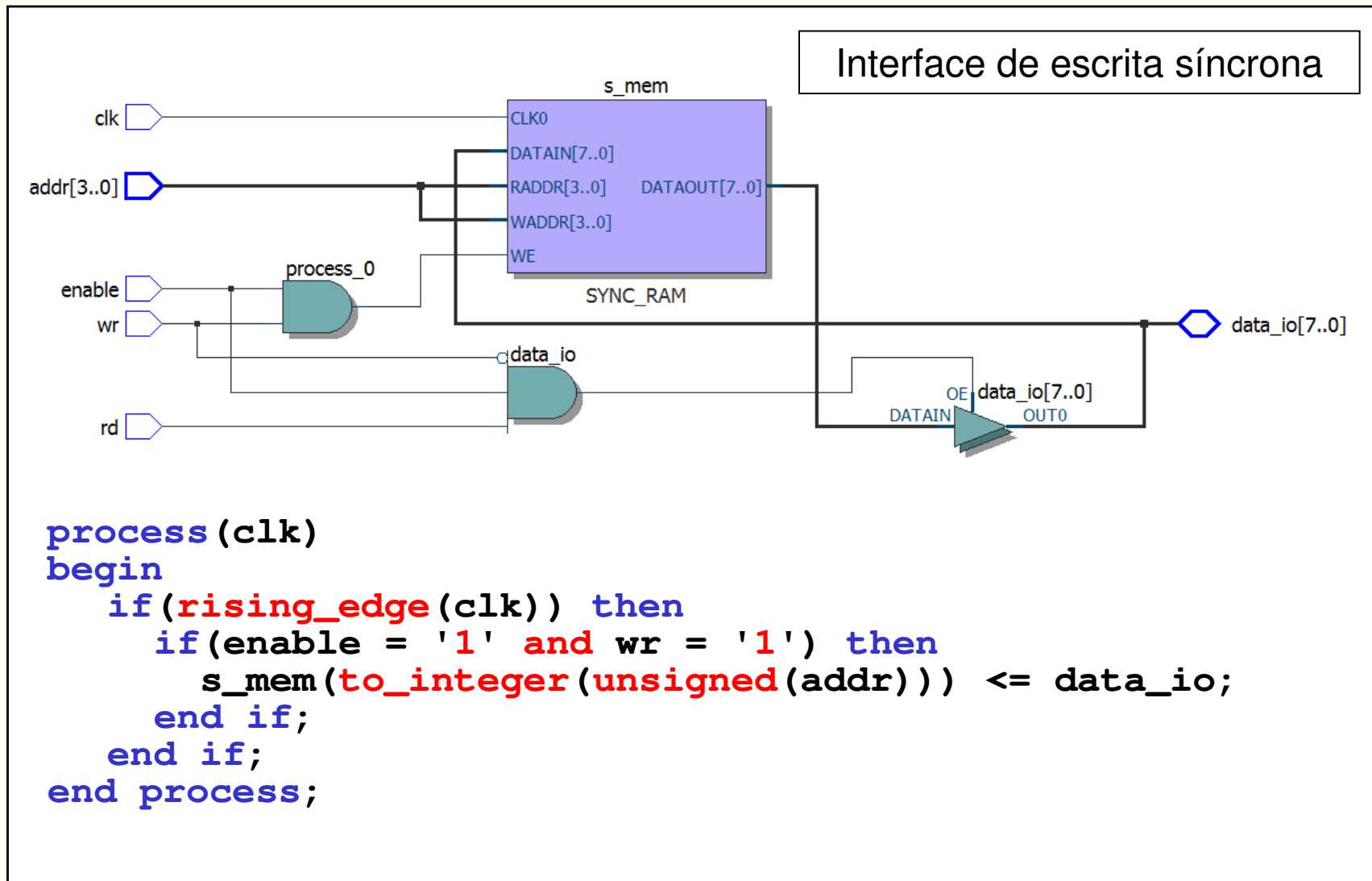
Escrita síncrona e leitura assíncrona.
O barramento de dados é bidirecional.



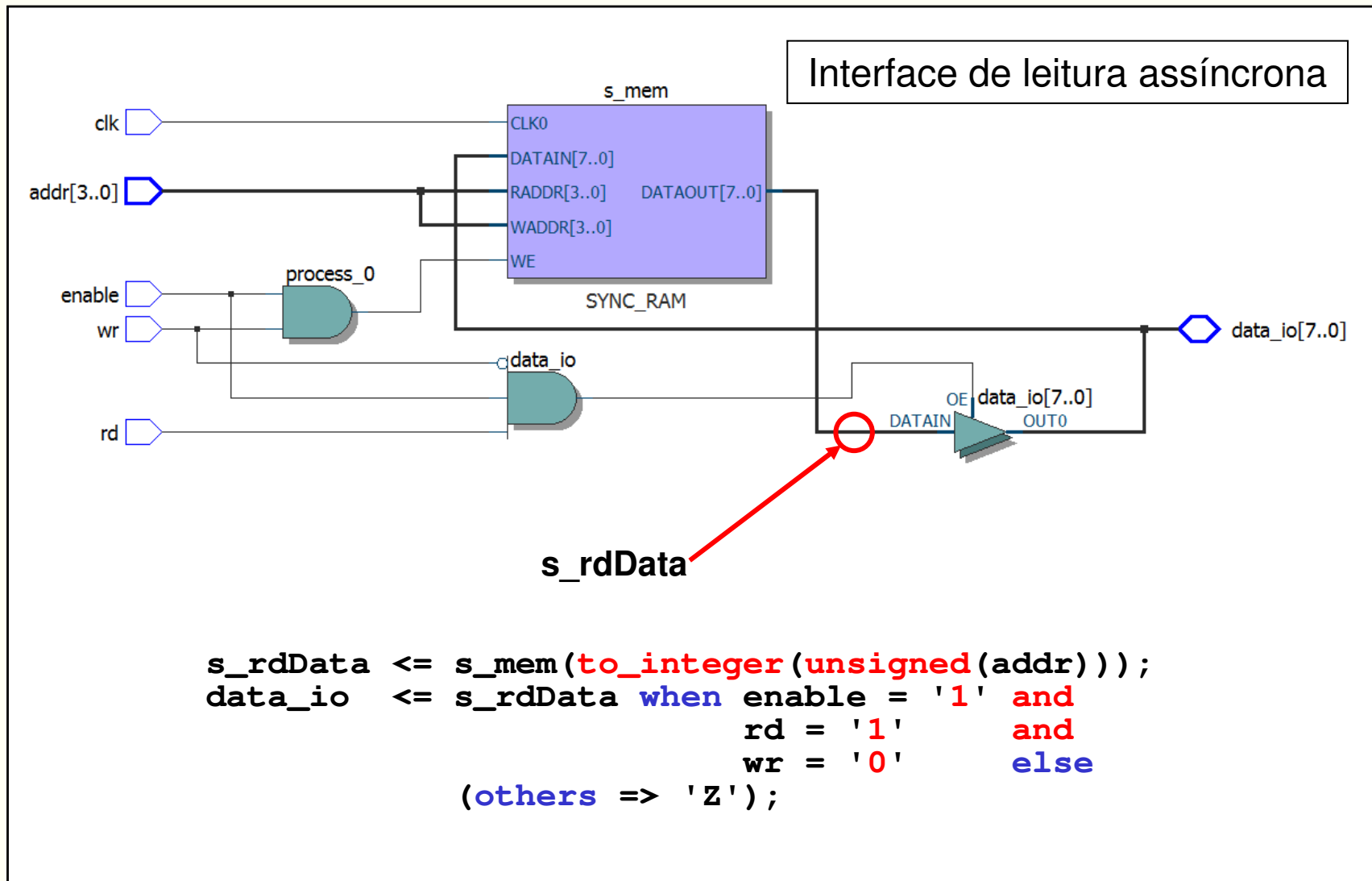
Exemplo: memória RAM 16x8 – VHDL

```
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



Exemplo: memória RAM 16x8 - síntese



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

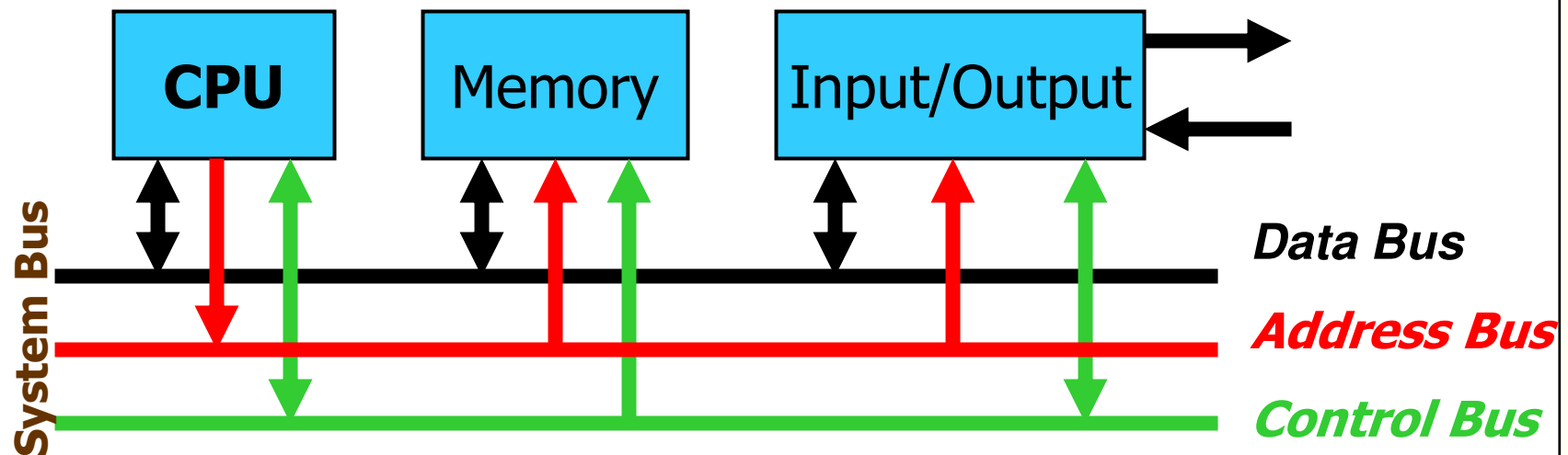
Arquitetura básica de um sistema computacional

- 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 (na memória ou nas unidades de input/output)
- **Control Bus:** sinais de protocolo que especificam o modo como a transferência de informação deve ser feita

Arquitetura básica de um sistema computacional

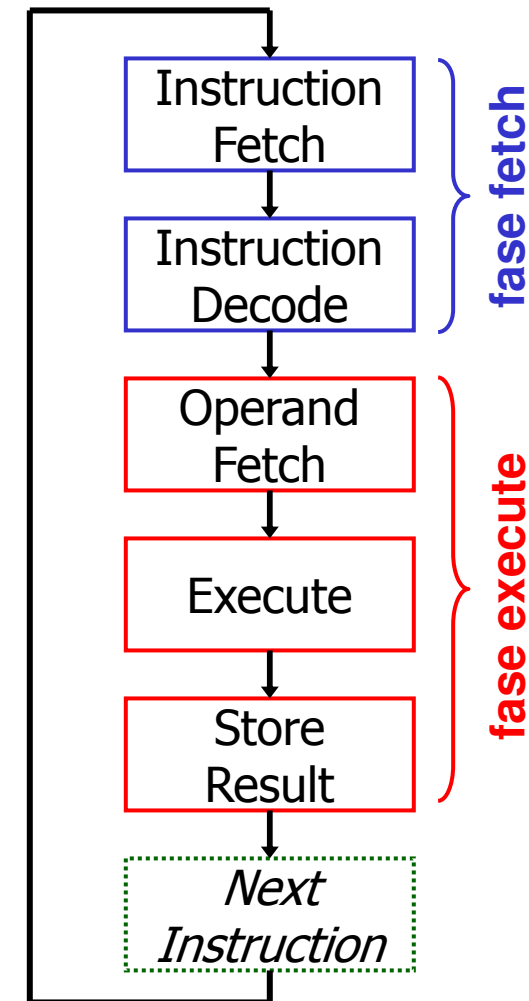
- **Endereço** (*address*) – um número (único) que identifica cada registo de memória. Os endereços são contados sequencialmente, começando em 0
 - Exemplo: o conteúdo da posição de memória 0x2000 é 0x32 – (0x2000 é o endereço, 0x32 o valor armazenado)
- **Espaço de endereçamento** (*address space*) – 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

High-level language
program (in C)

```
unsigned char toUpper(unsigned char c)
{
    if (c >= 'a' && c <= 'z')
        return (c - 0x20);
    else
        return c;
}
```

↓
Compiler

Assembly language
program (for MIPS)

```
toUpper:    addiu $5, $4, -97
            sltiu $1, $5, 26
            or    $2, $4, $0
            beq   $1, $0, else
            addiu $2, $4, -32
else:       jr    $31
```

↓
Assembler

Binary machine language
program (for MIPS)

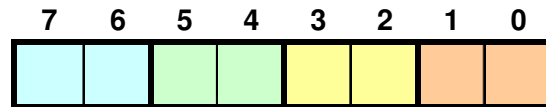
```
0010010010000100111111110011111  (0x2485ff9f)
00101100101000010000000000011010  (0x2ca1001a)
00000000100000000001000000100101  (0x00801025)
00010000001000000000000000000001  (0x10200001)
0010010010000010111111111100000  (0x2482ffe0)
00000011111000000000000000001000  (0x03e00008)
```

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?
 - **Registos 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

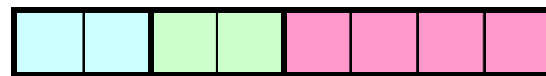
Exemplo - CPU hipotético

Formato de codificação das instruções (8 bits)



Formato 1

Oper. Rdest Rop1 Rop2



Formato 2

Oper. Reg. End. Memória

Registos 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

Hex

Binary

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

00001111

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

Qual é a expressão aritmética implementada neste programa?

Arquitetura do Conjunto de Instruções (ISA)

- **Instruction Set**: coleção de todas as operações/instruções que o processador pode executar
- **Instruction Set Architecture (ISA)**
 - ... *os atributos de um sistema computacional tal como são vistos pelo programador, i.e. a estrutura concetual 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
- Arquitetura de Computadores =
Arquitetura do Conjunto de Instruções (ISA) + Organização da Máquina

Arquitetura do Conjunto de Instruções (ISA)

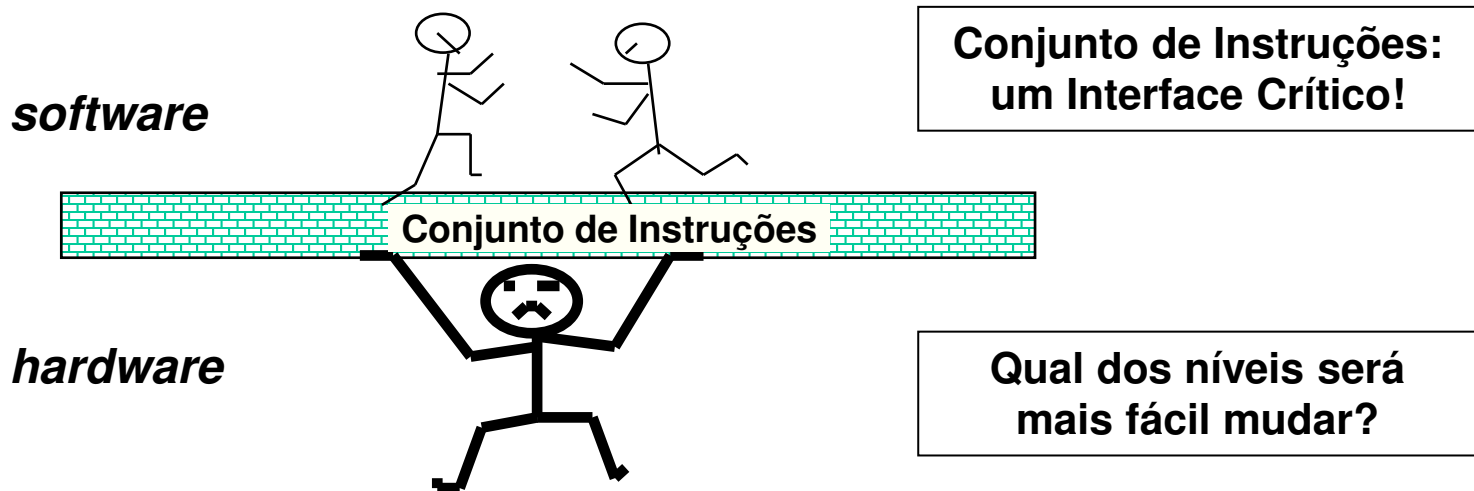
- Também designada por "modelo de programação"
- Descreve tudo o que o programador necessita de saber para programar corretamente, em *assembly*, um determinado processador
- Uma importante abstração que representa a **interface entre o nível mais básico de software e o hardware**
- Descreve a funcionalidade, de forma independente do hardware 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 físicas 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)

- Requisitos básicos da Arquitetura do Conjunto de Instruções:
 - Fácil de entender e programar
 - Desenvolvimento de compiladores eficientes
 - Implementação simples e eficiente em hardware
 - Com o melhor desempenho possível
 - Eficiente do ponto de vista energético
 - Com o menor custo possível



Classes de instruções

- Um dada arquitetura pode ter um ISA com centenas de instruções
- É possível, no entanto, considerar a existência de um grupo limitado de classes de instruções comuns à generalidade das arquiteturas
- Classes de instruções:
 - **Processamento**
 - Aritméticas e lógicas
 - **Transferência de informação**
 - Cópia entre registos internos e entre registos internos e memória
 - **Controlo de fluxo de execução**
 - Alteração da sequência de execução (estruturas condicionais, ciclos, chamadas a funções,...)

Questões

- O que é um endereço?
- O que é o espaço de endereçamento de um processador?
- Como se organiza internamente um processador? Quais são os blocos fundamentais da secção de dados? Para que serve a unidade de controlo?
- O que é o conceito "stored-program"?
- Como se codifica uma instrução? Que informação fundamental deverá ter o código de uma instrução?
- O que é o ISA?
- Quais são as classes de instruções que agrupam as instruções de uma arquitetura?

Aula 2

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

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Instruções e implementação hardware

- No projeto de um processador a definição do ***instruction set*** exige um delicado compromisso entre múltiplos aspetos, 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

ISA – formato e codificação das instruções

- Codificação das instruções com um número de bits variável
 - Código mais pequeno
 - Maior flexibilidade
 - *Instruction fetch* em vários passos
- Codificação das instruções com um número de bits 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
 - Algumas variáveis dos programas podem residir 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
 - **add a** **# acc ← acc + a**
- Arquiteturas baseadas em **Stack**
 - Operandos e resultado armazenados numa *stack* (pilha) de registos
 - **add** **# tos ← tos + next**
 (tos = top of stack)

ISA – localização dos operandos das instruções

- Arquiteturas **Register-Memory**

- Operandos das instruções aritméticas e lógicas residem em registos internos do CPU ou em memória

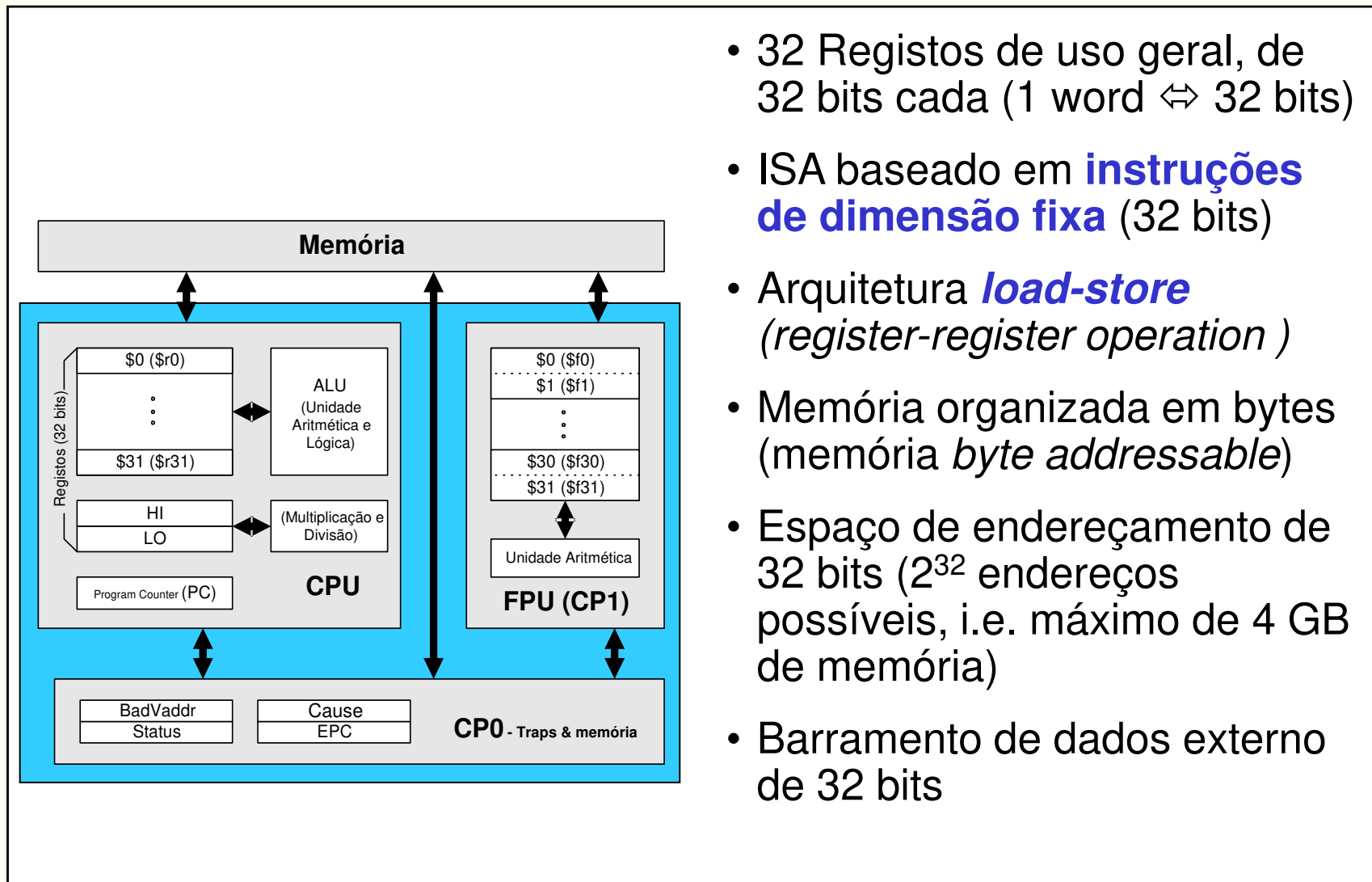
- **load** **r1, [a]** **# r1 ← mem[a]**
 - **add** **r1, [b]** **# r1 ← r1 + mem[b]**
 - **store** **[c], r1** **# mem[c] ← r1**

- Arquiteturas **Load-store**

- Operandos das instruções aritméticas e lógicas residem em registos internos do CPU de uso geral (mas nunca na memória).

- **load** **r1, [a]** **# r1 ← mem[a]**
 - **load** **r2, [b]** **# r2 ← mem[b]**
 - **add** **r3, r1, r2** **# r3 ← r1 + r2**
 - **store** **[c], r3** **# mem[c] ← r3**

Aspetos chave da arquitetura MIPS



- 32 Registos de uso geral, de 32 bits cada (1 word \Leftrightarrow 32 bits)
- ISA baseado em **instruções de dimensão fixa** (32 bits)
- Arquitetura **load-store** (*register-register operation*)
- 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

Instruções aritméticas - SOMA

Formato da instrução *Assembly* do MIPS:



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

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 expressão 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 expressão $z = (a + b) - (c + d)$
tem de ser decomposta em:

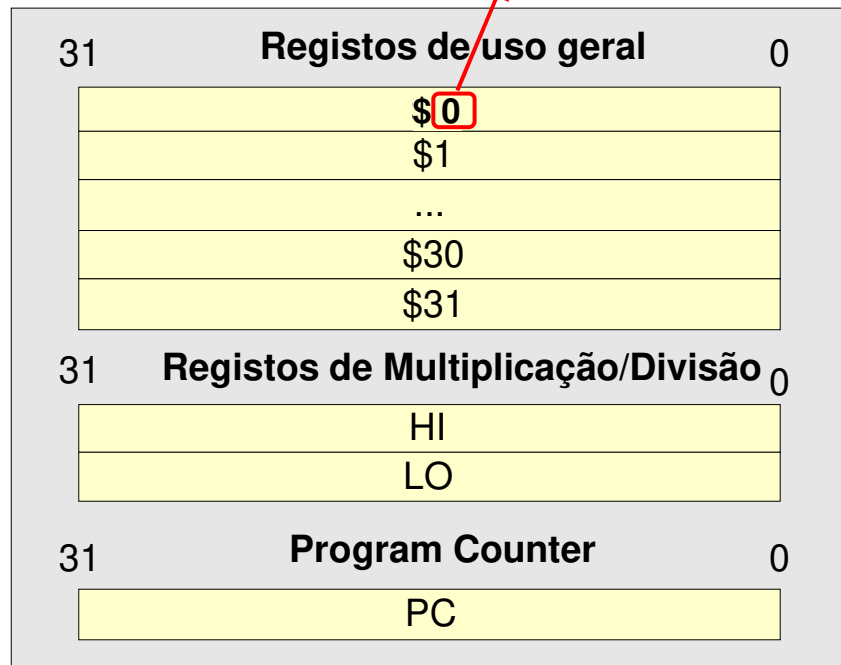
add **x**, **a**, **b** # Soma **a** com **b**, resultado em **x**

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

sub **z**, **x**, **y** # Subtrai **y** a **x**, e coloca o resultado em **z**

Os registos internos do MIPS

Endereço do registo (0 a 31)



Program Counter: registo que contém o endereço de memória onde está armazenado o código da próxima instrução a executar

- 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)

- Registo **\$0** tem sempre o valor **0x00000000** (apenas pode ser lido)

Exemplo de tradução de C para *Assembly* MIPS

- Programa em C:

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

- Em *assembly* (supondo que 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
```

Exemplo de tradução de C para *Assembly* MIPS

- Programa em C:

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

```
# a: $17, b: $18, c: $19, d: $20, z: $16
```

```
...
```

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

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

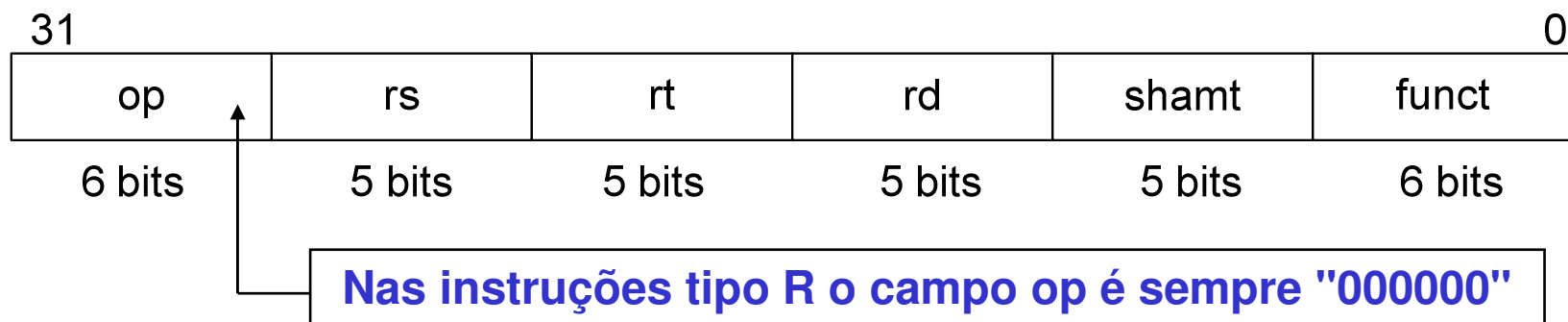
```
sub  $16, $8, $9    # z = (a + b) - (c + d);
```

```
...
```

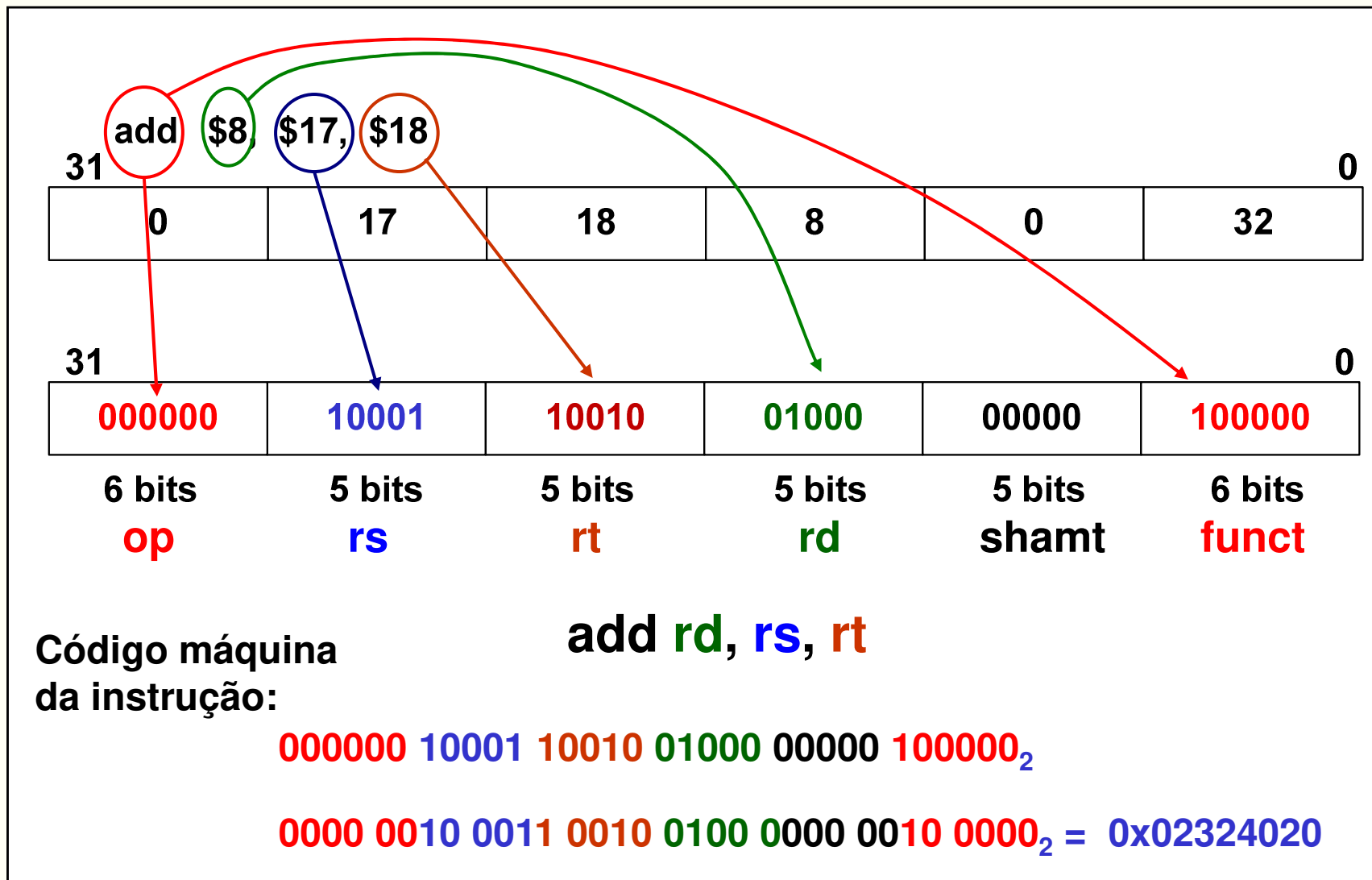
- A linguagem C é uma excelente forma de comentar programas em *Assembly* uma vez que permite uma interpretação direta e mais simples do(s) algoritmo(s) implementado(s).

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

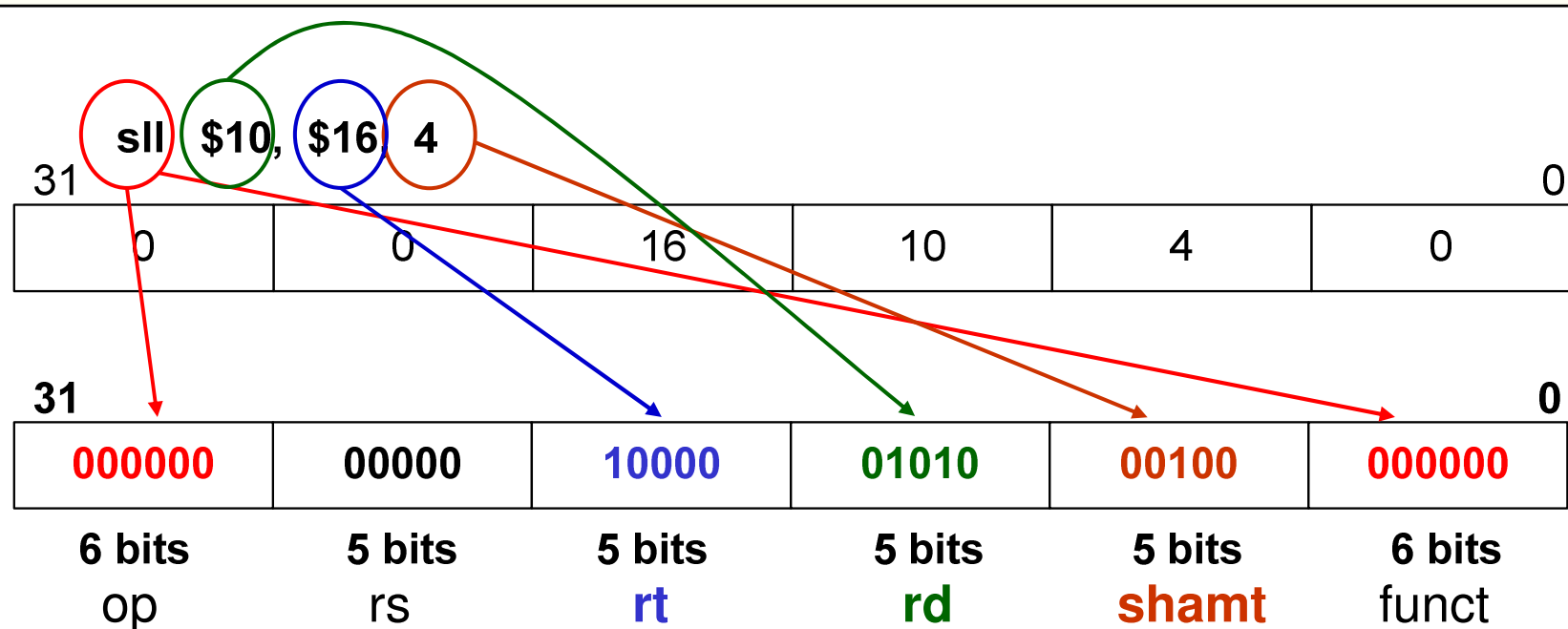
- Operadores lógicos bit a bit (*bitwise operators*) em C:
 - **&** (AND), **|** (OR), **^** (XOR), **~** (NOT)
- A operação indicada é realizada bit a bit nos dois operandos, no caso do AND, do OR e do XOR e é feita a negação de todos os bits do operando no caso do NOT.
- Os operadores bit a bit "**&**" e "**|**" não devem ser confundidos com os operadores lógicos "**&&**" e "**||**".
- **Exercício**: determine os resultados deste programa:

```
void main(void)
{
    int a = 10;
    int b = 9;
    printf("a & b = %d\n", a & b);    // ?
    printf("a && b = %d\n", a && b);   // ?
    printf("a | b = %d\n", a | b);    // ?
    printf("a || b = %d\n", a || b);  // ?
}
```

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 = ~(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



sll **rd**, **rt**, **shamt**

**Código máquina
da instrução:**

00000000000100000101000100000000₂ = 0x00105100

O que faz a instrução cujo código máquina é: 0x00000000 ?

Instruções de transferência entre registos internos

- Transferência entre registos internos: $R_{dst} = R_{src}$
- 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, Rsrc, \$0** # $R_{dst} = (R_{src} | 0) = R_{src}$
 - Exemplo: **or \$t1, \$t2, \$0** # $\$t1 = \$t2$
- 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** # $R_{dst} = R_{src}$
 - Exemplo: **move \$t1, \$t2** # $\$t1 = \$t2$ (or $\$t1, \$t2, \$0$)

Questões

- O que caracteriza as arquiteturas "register-memory" e "load-store"? De que tipo é a arquitetura MIPS?
- Com quantos bits são codificadas as instruções no MIPS? Quantos registros internos tem o MIPS? O que diferencia o registro `$0` dos restantes? Qual o número do registro interno do MIPS a que corresponde o registro `$ra`?
- Quais os campos em que se divide o formato de codificação **R**? Qual o significado de cada um desses campos? Qual o valor do campo **opCode** nesse formato?
- O que faz a instrução cujo código máquina é: `0x00000000`?
- O símbolo `>>` da linguagem C significa deslocamento à direita e é traduzido por **SRL** ou **SRA** (no caso do MIPS). Quando é que usado **SRL** e quando é que é usado **SRA**?
- Qual a instrução nativa do MIPS em que é traduzida a instrução virtual `"move $4, $15"`?

Exercícios

- Determine o código máquina das seguintes instruções:
xor \$5, \$13, \$24 - sub \$30, \$14, 8 - sll \$3, \$9, 7
sra \$18, \$9, 8
- Traduza para instruções *assembly* do MIPS a seguinte expressão aritmética, supondo **x** e **y** inteiros e residentes em **\$t2** e **\$t5**, respetivamente (apenas pode usar instruções nativas e não deverá usar a instrução de multiplicação):
y = -3 * x + 5;
- Traduza para instruções *assembly* do MIPS o seguinte trecho de código:
int a, b, c; //a:\$t0, b:\$t1, c:\$t2
unsigned int x, y, z; //x:\$a0, y:\$a1, z:\$a2
z = x >> 2 + y;
c = a >> 5 - 2 * b;

Aulas 3 e 4

- 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

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Instruções de controlo de fluxo de execução

- Durante a execução de um programa há necessidade de tomar decisões com base em valores que só são conhecidos durante a execução do mesmo
- As instruções que permitem a tomada de decisões pertencem à classe "controlo de fluxo de execução"
- No MIPS as instruções básicas de controlo de fluxo de execução são:

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

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

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

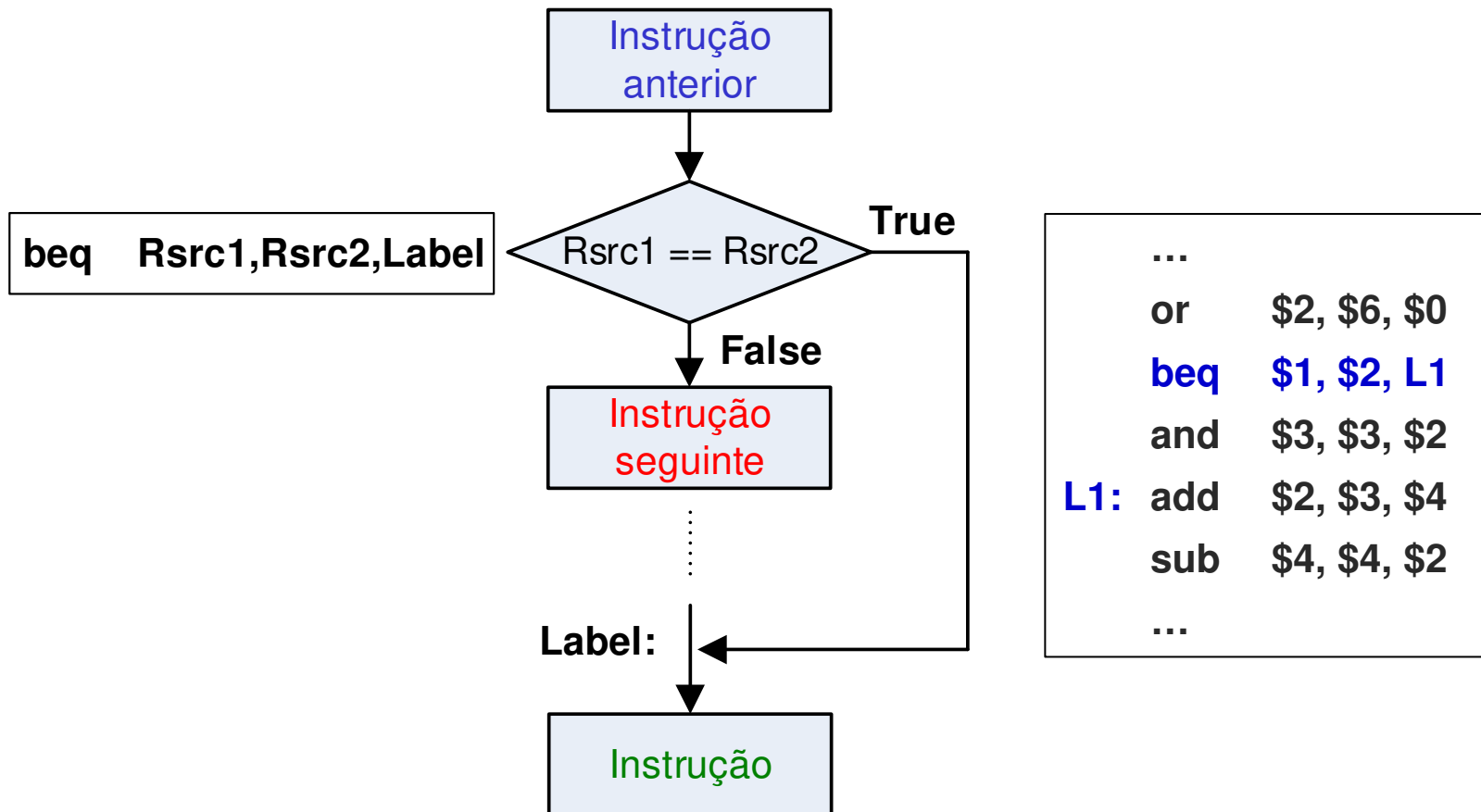
Instruções de controlo de fluxo de execução – BEQ

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*)

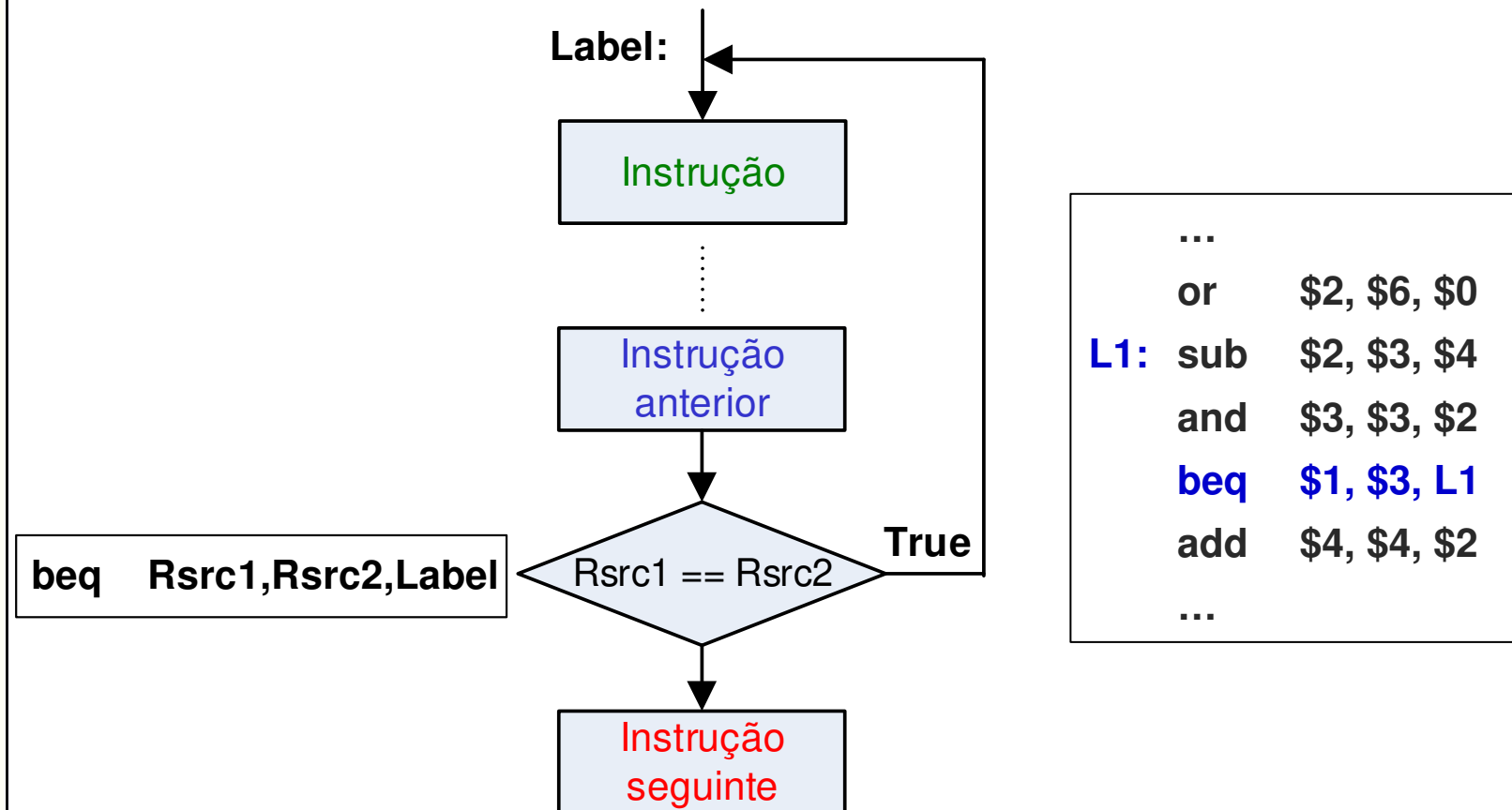
Instruções de *branch* – como funcionam?

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



Instruções de *branch* – como funcionam?

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



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
 - A instrução que é executada de seguida é a que se situa imediatamente a seguir à instrução de *branch*

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:

```
        or      $2, $6, $0
        bne     $1, $2, L1  # se "branch taken" (i.e. $1 != $2)
                           # a próxima instrução a executar
                           # está em L1 (add $2,$3,$4)
        and     $3, $3, $2  # se "branch not taken" a sequência
                           # de execução não é alterada
L1:     add     $2, $3, $4  #
        sub     $4, $4, $2  #
```

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

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  $\equiv$  "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  $\equiv$  "set if less than"  
                                # set Rdst if Rsrc1 < Imm
```

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

Instruções virtuais de *branch* do MIPS

- Nos programas *Assembly*, podem ser utilizadas instruções de salto não diretamente suportadas pelo MIPS (**instruções virtuais**), mas que são **decompostas pelo *assembler* em instruções nativas**.

Essas instruções são:

- **blt** **Rsrc1**, **Rsrc2**, **Label** # Branch if Rsrc1 < Rsrc2
- **ble** **Rsrc1**, **Rsrc2**, **Label** # Branch if Rsrc ≤ Rsrc2
- **bgt** **Rsrc1**, **Rsrc2**, **Label** # Branch if Rsrc > Rsrc2
- **bge** **Rsrc1**, **Rsrc2**, **Label** # Branch if Rsrc ≥ Rsrc2

- 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 decompostas estas instruções?

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
```

Decomposição das instruções virtuais BGT e BGE

Sobre este tema, pode encontrar informação complementar e mais detalhada no documento:

Como decompor instruções de logica relacional em instruções nativas

disponível na secção “Documentos de apoio às aulas teóricas e práticas” do moodle da UC.

Instrução de salto incondicional

- As arquiteturas disponibilizam também instruções de **salto incondicional**
- Numa instrução de **salto incondicional** não existe o teste de qualquer condição: **o salto é sempre realizado**
- No ISA do MIPS a instrução de salto incondicional tem a mnemónica "**j**", que significa *jump*

j label

- O fluxo de execução é desviado, de forma incondicional, para "label"

Estruturas de controlo de fluxo em C

- Exemplos

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

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

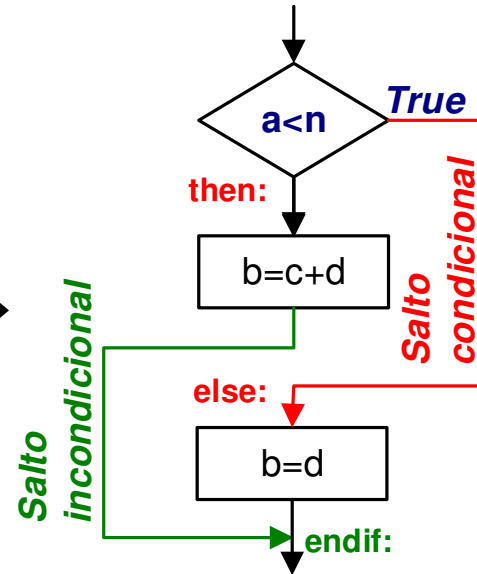
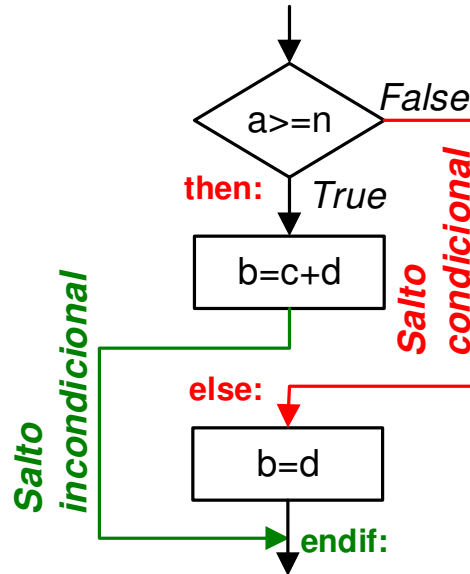
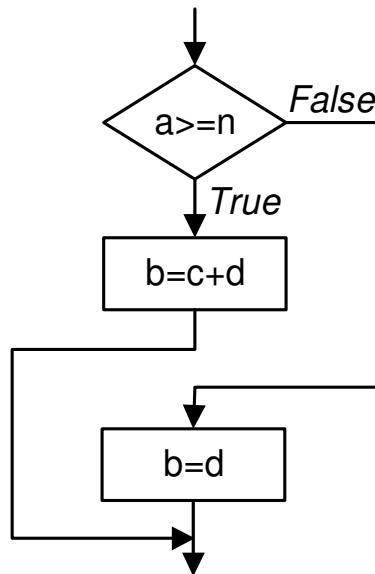
```
n = 0;  
do {  
    a = a + b[n];  
    n++;  
} while (n < 100);  
...
```

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

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

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

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

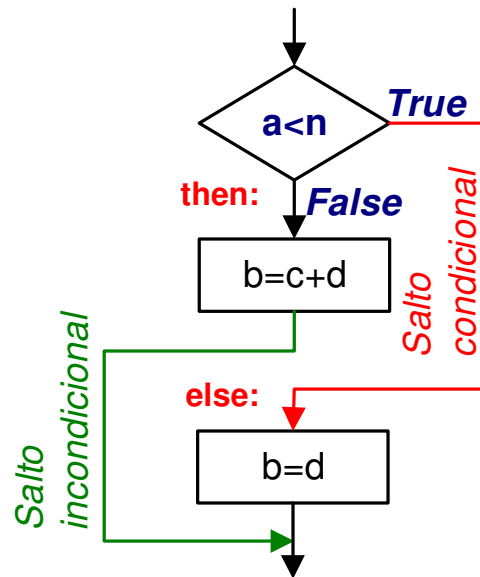


Tradução para *Assembly* do MIPS – *if()... then... else*

```
if (a >= n) {
    b = c + d;
} 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) {
    add    $t3,$t2,$t4  #     b = c + d;
    j      endif        # }
else:     # else {
    or     $t3,$t4,$0   #     b = d;
endif:    ...          # }

```

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

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++) {  
    ...  
}
```

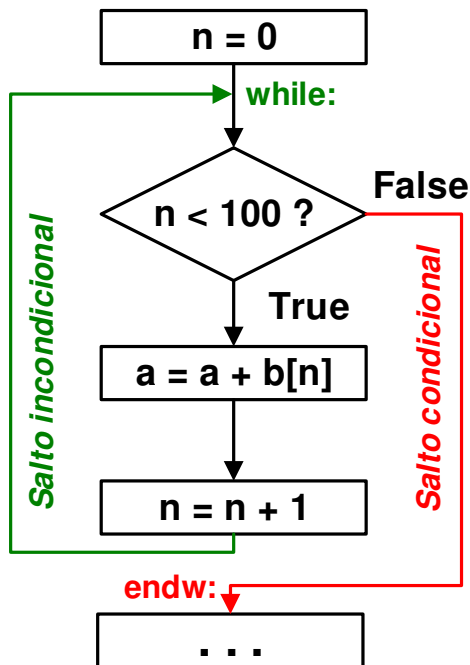
Qual o resultado deste código?

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

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

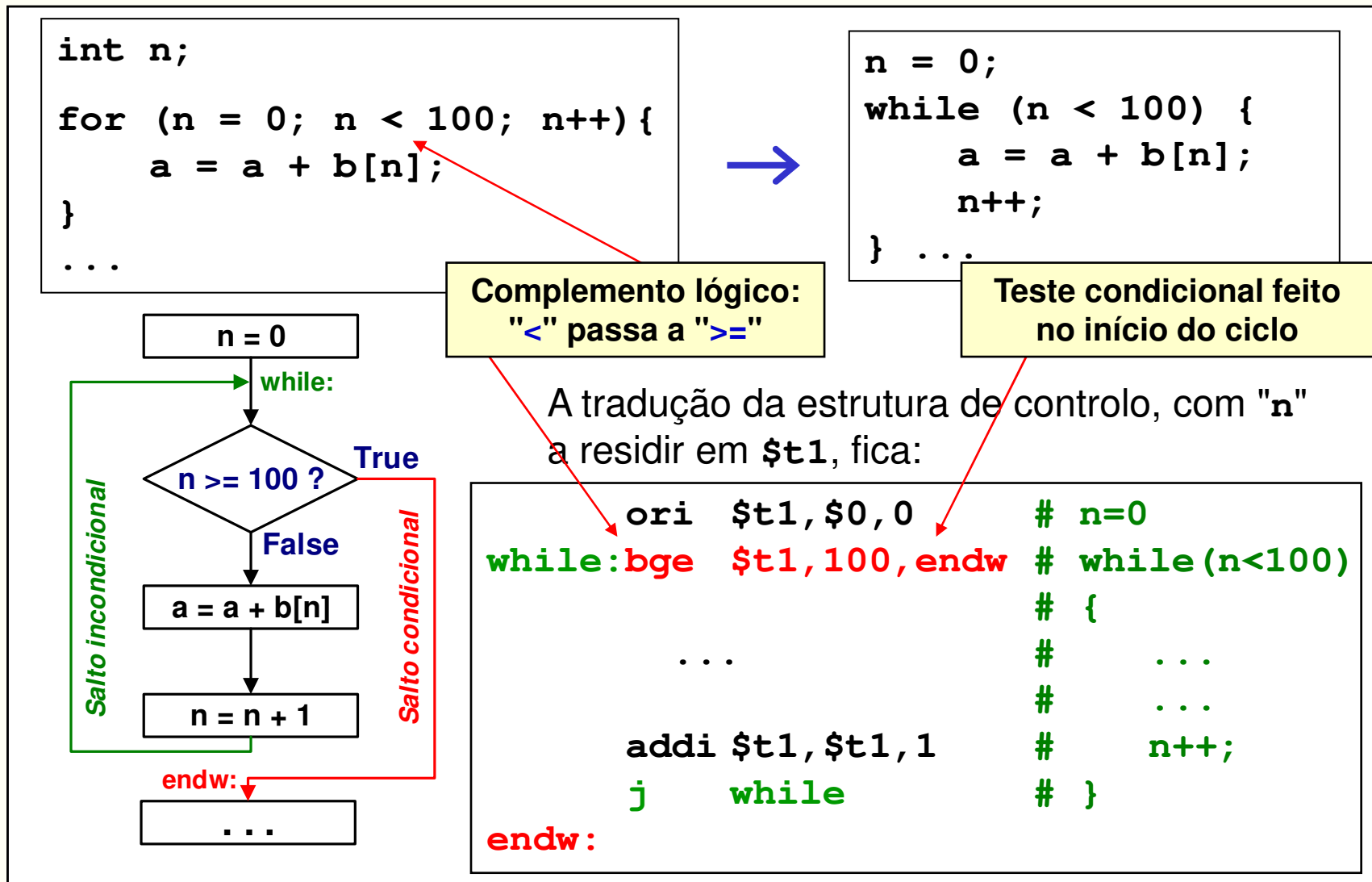


```
n = 0;  
while (n < 100) {  
    a = a + b[n];  
    n++;  
} ...
```



- 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 ">=")

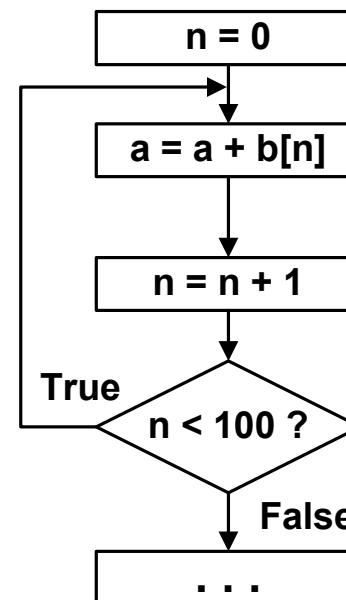
Tradução para *Assembly* do MIPS - ciclos *for()* e *while()*



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);  
...
```

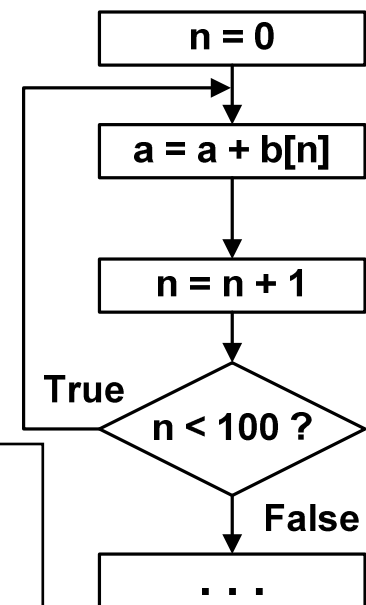


Tradução para *Assembly* do MIPS - ciclo *do ... while()*

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

A tradução para *Assembly* do MIPS fica (com "n" a residir em \$t1):

```
ori    $t1,$0,0      # n = 0;  
do:    # do {  
  
    ...              # a = a + b[n]  
  
addi   $t1,$t1,1      # n = n + 1;  
blt    $t1,100,do     # } while(n < 100);  
...
```



Teste condicional feito no fim do ciclo. Mesma condição do código original.

Conclusão

- As estruturas do tipo ciclo incluem, geralmente, uma ou mais instruções de inicialização de variáveis, executadas antes e fora do mesmo
- No caso dos ciclos **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, o que significa que o corpo do ciclo é executado pelo menos uma vez
- Na tradução de um ciclo **for()** para *Assembly*, o terceiro campo é codificado no fim do corpo do ciclo, ou seja, é a última instrução do ciclo antes do **j** que fecha o ciclo.

Questões / Exercícios

- Qual a função da instrução **"slt"**?
- Qual o valor armazenado no registo **\$1** na execução da instrução **"slt \$1, \$3, \$7"**, admitindo que: a) **\$3=5** e **\$7=23** e b) **\$3=0xFE** e **\$7=0x913D45FC**
- Com que registo comparam as instruções **"bltz"**, **"blez"**, **"bgtz"** e **"bgez"**?
- Decomponha em instruções nativas do MIPS as seguintes instruções virtuais:
 - **blt \$15, \$3, exit**
 - **ble \$6, \$9, exit**
 - **bgt \$5, 0xA3, exit**
 - **bge \$10, 0x57, exit**
 - **blt \$19, 0x39, exit**
 - **ble \$23, 0x16, exit**

Exercícios

- Traduza para *assembly* do MIPS os seguintes trechos de código de linguagem C (admita que **a**, **b** e **c** residem nos registos \$4, \$7 e \$13, respetivamente):

```
1)    if(a > b && b != 0)
        c = b << 2;
    else
        c = (a & b) ^ (a | b);
```

```
2)    if(a > 3 || b <= c)
        c = c - (a + b);
    else
        c = c + (a - 5);
```

- Na tradução para *assembly*, que diferenças encontra entre um ciclo do tipo "**while**(...) {...}" e um do tipo "**do**{...}**while**(...);"

Exercícios

- Traduza para *assembly* do MIPS os seguintes trechos de código de linguagem C (atribua registos internos para o armazenamento das variáveis **i** e **k**) :

1) `int i, k;
 for(i=5, k=0; i < 20; i++, k+=5);`

2) `int i=100, k=0;
 for(; i >= 0;)
 {
 i--;
 k -= 2;
 }`

3) `unsigned int k=0;
 for(; ;)
 {
 k += 10;
 }`

4) `int k=0, i=100;
 do
 {
 k += 5;
 } while(--i >= 0);`

Aulas 5 e 6

- Armazenamento de informação na memória externa
- Endereçamento indireto por registo com deslocamento
- Instruções de acesso a words de 32 bits armazenadas na memória externa: LW e SW
- Codificação das instruções de acesso à memória: formato I
- Restrições de alinhamento nos endereços das variáveis
- Instruções de acesso a bytes na memória externa: LB, LBU e SB
- Organização de informação em memória: "little-endian" *versus* "big-endian"
- Diretivas do *assembler* MARS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Armazenamento de informação – registos internos

- Nos exemplos das aulas anteriores apenas se fez uso de registos internos do CPU para o armazenamento de informação (variáveis):

```
int a, b, c, d, z;           // a, b, c, d e z residem, respetivamente, em:
z = (a + b) - (c + d);       // $17, $18, $19, $20 e $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
```

- E se a informação a processar residir na memória externa (por exemplo um *array* de inteiros) ?
- Recorde-se que a arquitetura MIPS é do tipo **load-store**, ou seja, não é possível operar diretamente sobre o conteúdo da memória externa
- Terão que existir instruções para transferir informação entre os registos do CPU e a memória externa

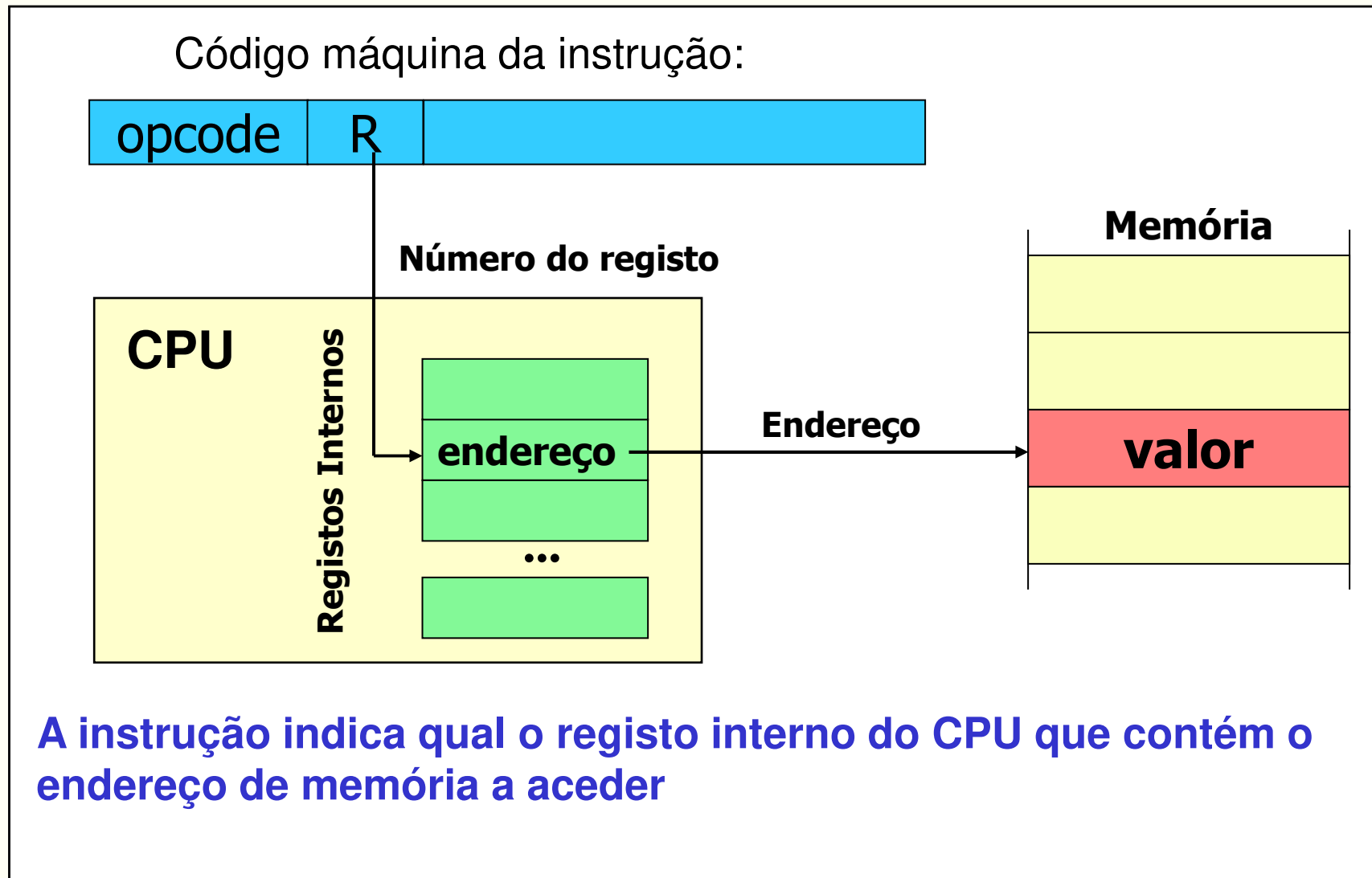
Modos de endereçamento

- 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 (codificadas com o formato R), os operandos residem ambos em registos internos
 - 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

- O acesso à memória externa implica sempre especificar o endereço da posição que se quer ler ou escrever
- O espaço de endereçamento do MIPS é de 32 bits, pelo que um endereço de memória é representado por 32 bits
- 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?
 - Para codificar apenas o endereço já seriam necessários 32 bits...
- **Solução**: em vez do endereço, a instrução indica um registo que contém o endereço de memória a aceder (no MIPS um registo interno permite armazenar 32 bits):
 - **endereçamento indireto por registo**

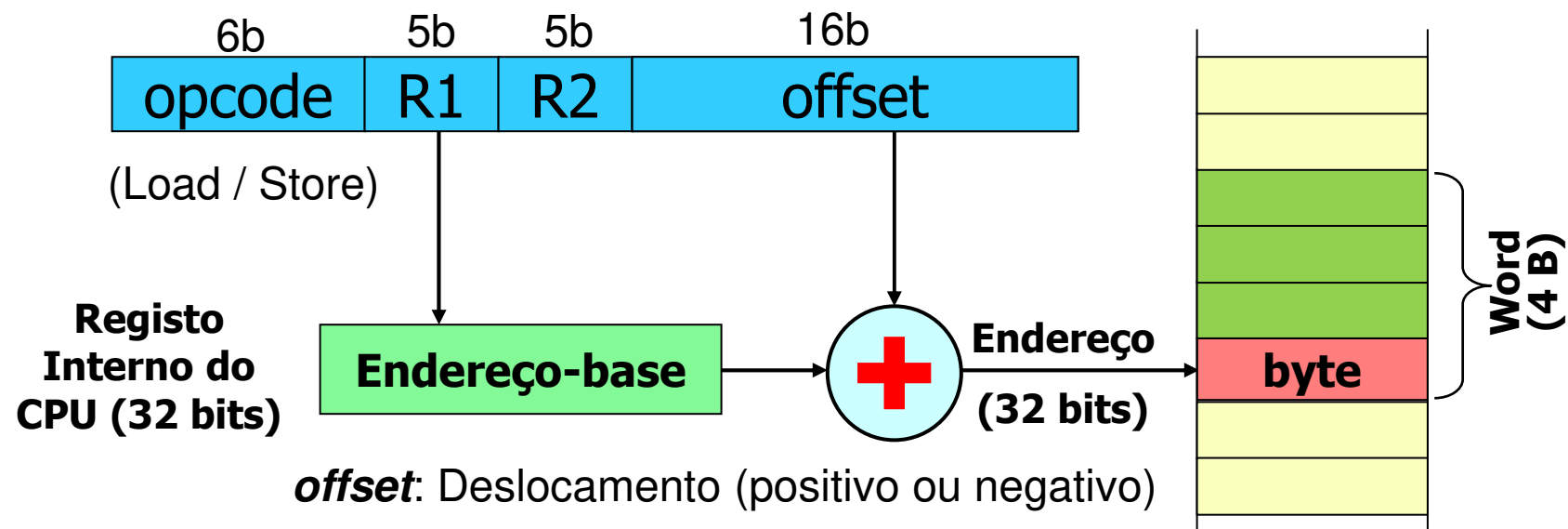
Endereçamento indireto por registo



Endereçamento indireto por registo com deslocamento

- A solução do MIPS

Código máquina da instrução:



R1: Registo de endereçamento indireto

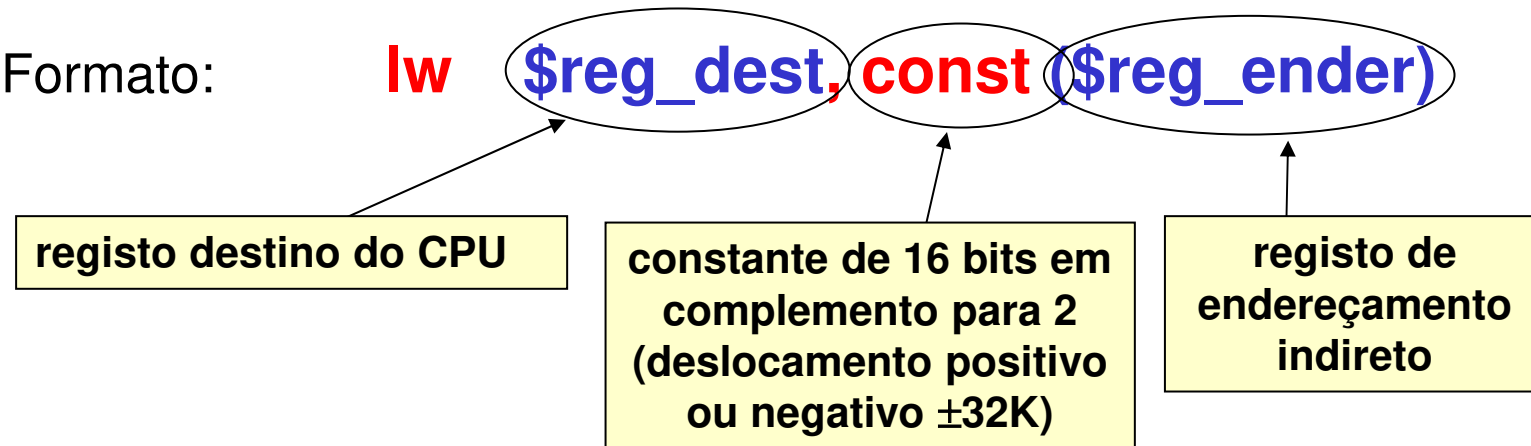
R2: Registo de dados: destino / origem

O endereço de acesso à memória é calculado pela **soma algébrica do conteúdo do registo com o *offset*** (estendido, 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:



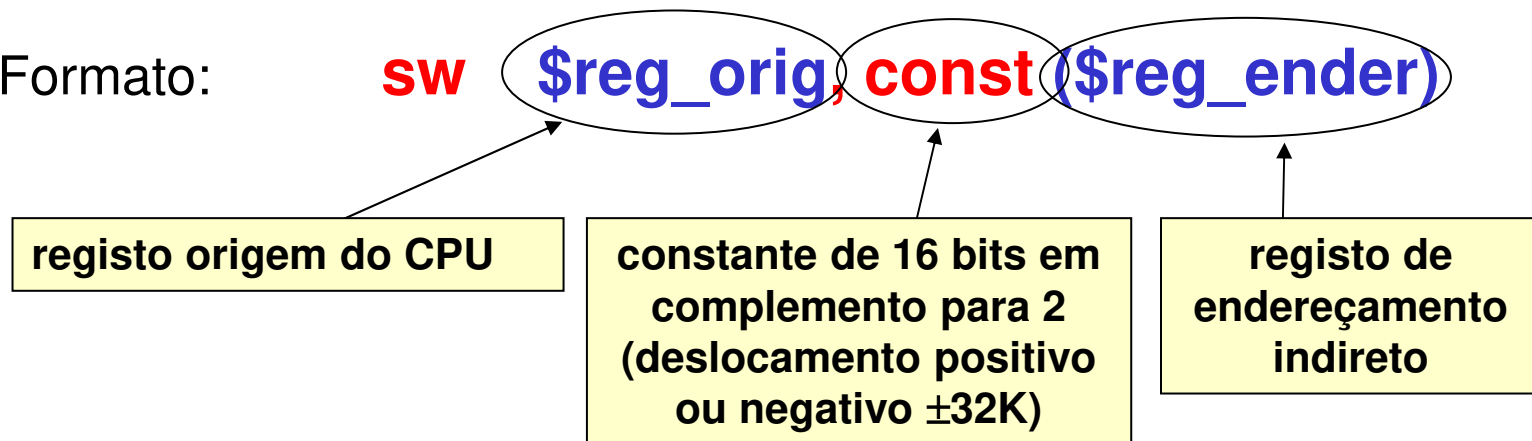
Exemplo:

lw \$5, 4 (\$2) # copia para o registo \$5 a *word* de 32 bits 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 registro 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 registro \$7 para a
memória, a partir do endereço calculado como:
addr = (conteúdo do registro \$4) - 8

Acesso à memória

- Exemplo

Address	Data	
0x00000020	0x45	word
0x00000021	0x12	
0x00000022	0x3A	
0x00000023	0xF3	
0x00000024	0xC9	word
0x00000025	0x7D	
0x00000026	0xB3	
0x00000027	0x9D	
0x00000028	0x47	word
0x00000029	0x5F	
0x0000002A	0x6D	
0x0000002B	0x4A	
0x0000002C	0xFD	word
0x0000002D	0xC0	
0x0000002E	0x5A	
0x0000002F	0x7C	
0x00000030	0x1D	
...	...	

`int A[4]; // array de inteiros`

Address	Data	
0x00000020	0x45	A[0]
0x00000021	0x12	
0x00000022	0x3A	
0x00000023	0xF3	
0x00000024	0xC9	A[1]
0x00000025	0x7D	
0x00000026	0xB3	
0x00000027	0x9D	
0x00000028	0x47	A[2]
0x00000029	0x5F	
0x0000002A	0x6D	
0x0000002B	0x4A	
0x0000002C	0xFD	A[3]
0x0000002D	0xC0	
0x0000002E	0x5A	
0x0000002F	0x7C	
0x00000030	0x1D	
...	...	

Endereço inicial do *array* A = endereço de A[0]: **0x20** (0x00000020)

Endereço de A[1]: **0x24**, endereço de A[2]: **0x28**, endereço de A[3]: **0x2C**

Acesso à memória: exemplo 1

- Considere-se o seguinte exemplo:

$g = h + A[3];$

assumindo que ***g***, ***h*** e o **endereço de início do array *A*** residem nos registros ***\$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 inteiros, i.e. 32 bits):

`lw $8, 12($19) # Lê A[3] da memória`
`add $17, $18, $8 # Calcula novo valor de g`

Variável temporária (destino)

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

Acesso à memória: exemplo 1

- Na primeira instrução da sequência anterior

`lw $8, 12 ($19) # Lê A[3] da memória`

O endereço da memória é calculado pelo processador somando o conteúdo do registo indicado entre parêntesis com a constante explicitada na instrução

- Se, por exemplo, o conteúdo de **\$19** for **0x00000020** o endereço da memória a que a instrução acede é:

`lw $8, 12 ($19) # Lê A[3] da memória`

0x0000000C + 0x00000020 = 0x0000002C ← **Endereço resultante**

- Se o valor armazenado em \$19 corresponder ao endereço do primeiro elemento do *array* de inteiros e como cada elemento do *array* ocupa quatro *bytes*, o elemento acedido é A[3]

Acesso à memória: exemplo 2

- Se se pretendesse obter:

$$A[3] = h + A[3]$$

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, 12 (\$19)	# Lê A[3] da memória
add	\$8, \$18, \$8	# Calcula novo valor
sw	\$8, 12 (\$19)	# Escreve resultado em A[3]

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**



Formato I

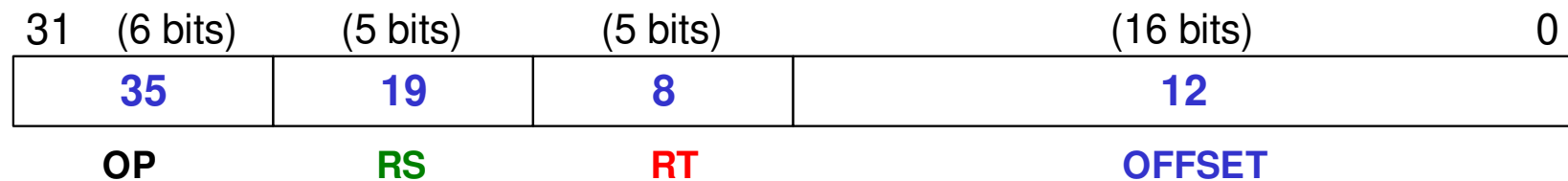
Codificado em complemento para 2 ($\pm 32K$)

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

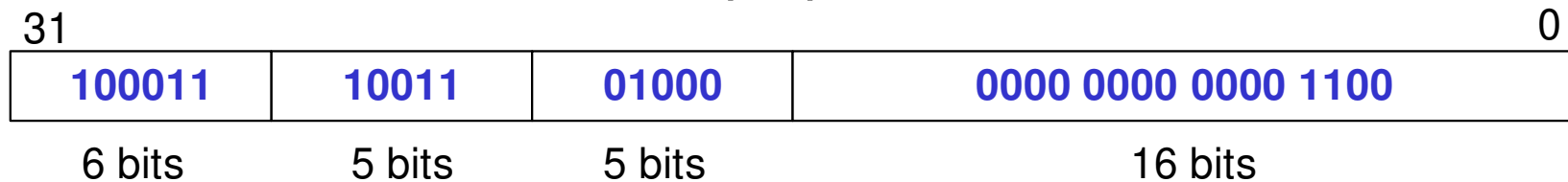
Codificação da instrução LW (Load Word)

lw **\$8,** **12(\$19)** **#Lê A[3] da memória**

Corresponderia à seguinte instrução máquina:



LW **RT, OFFSET(RS)**



10001110011010000000000000001100₂

1000 1110 0110 1000 0000 0000 0000 1100 = 0x8E68000C

Codificação da instrução SW (Store Word)

sw

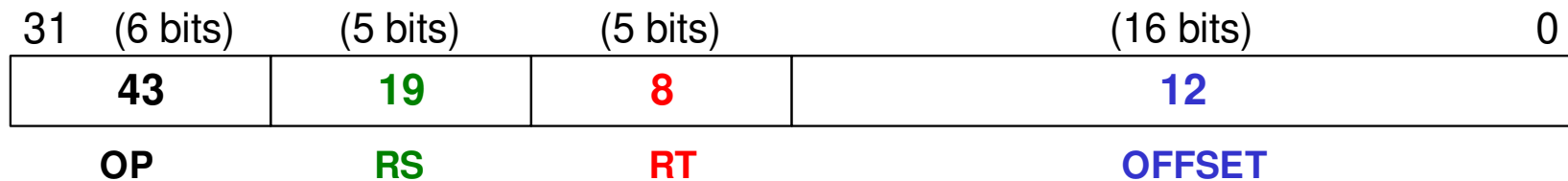
\$8,

12

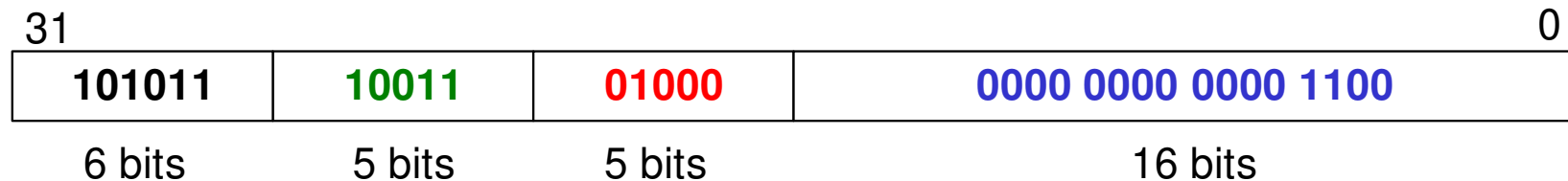
(\$19)

#Escreve result. em A[3]

Corresponderia à seguinte instrução máquina:



SW **RT, OFFSET(RS)**



10101110011010000000000000001100₂

1010 1110 0110 1000 0000 0000 0001 0100 = 0xAE68000C

Exemplo de codificação

- O seguinte trecho de código *assembly*:

```
lw    $8, 12 ($19)      # Lê A[3] da memória
add   $8, $18, $8        # Calcula novo valor
sw    $8, 12 ($19)      # Escreve resultado em A[3]
```

Corresponde à codificação:

31						0	
0x23	0x13	0x08	0x000C			Formato I	
0x00	0x12	0x08	0x08	0x00	0x20	Formato R	
0x2B	0x13	0x08	0x000C			Formato I	

- Resultando no código máquina:

$10001110011010000000000000001100_2 = 0x8E68000C$

$00000010010010000100000000100000_2 = 0x02484020$

$10101110011010000000000000001100_2 = 0xAE68000C$

Restrições de alinhamento nas instruções LW e SW

word {

Address	Data
0x00000020	0x45
0x00000021	0x12
0x00000022	0x3A
0x00000023	0xF3
0x00000024	0xC9
0x00000025	0x7D
0x00000026	0xB3
0x00000027	0x9D
0x00000028	0x47
0x00000029	0x5F
0x0000002A	0x6D
0x0000002B	0x4A
0x0000002C	0xFD
0x0000002D	0xC0
0x0000002E	0x5A
0x0000002F	0x7C
0x00000030	0x1D
...	...

} X



Address	Data
0x00000020	0x45123AF3
0x00000024	0xC97DB39D
0x00000028	0x475F6D4A
0x0000002C	0xFDC05A7C
0x00000030	...

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

Restrições de alinhamento nas instruções LW e SW

- Externamente o barramento de endereços do MIPS só tem disponíveis 30 dos 32 bits: $A_{31}...A_2$. 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 endereço calculado por uma instrução de leitura/escrita de uma **word** da memória, não é um 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*) ?

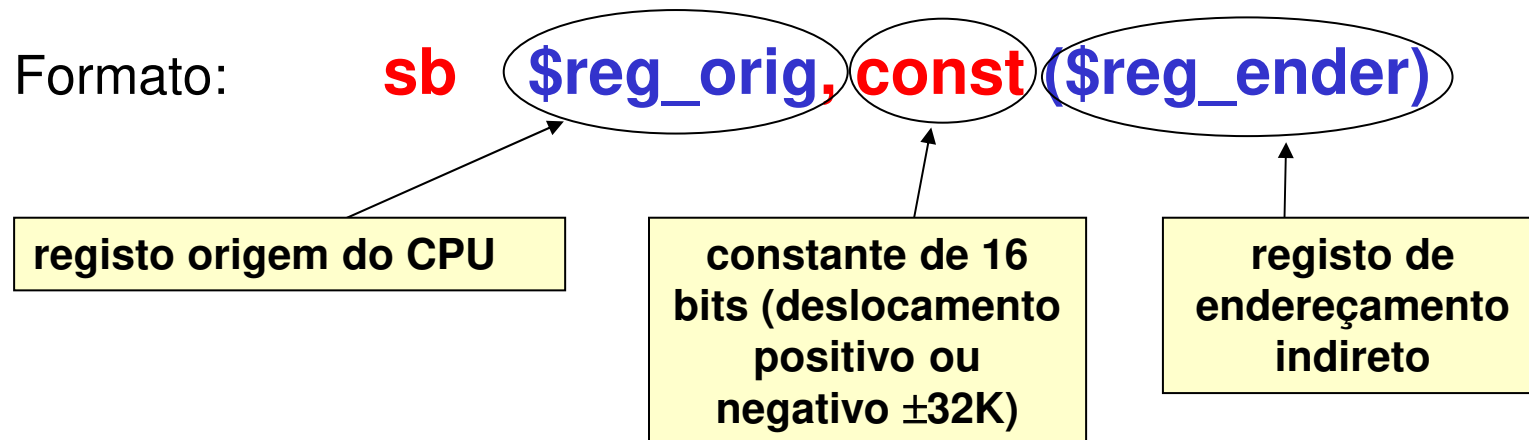
Restrições de alinhamento nos endereços das variáveis

- Se, numa instrução de leitura/escrita **de uma word**, for calculado um endereço **não múltiplo de 4**, quando o MIPS tenta aceder em memória a esse endereço 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**)
- Como se pode verificar facilmente que um endereço de 32 bits é múltiplo de 4? E múltiplo de 8?

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:



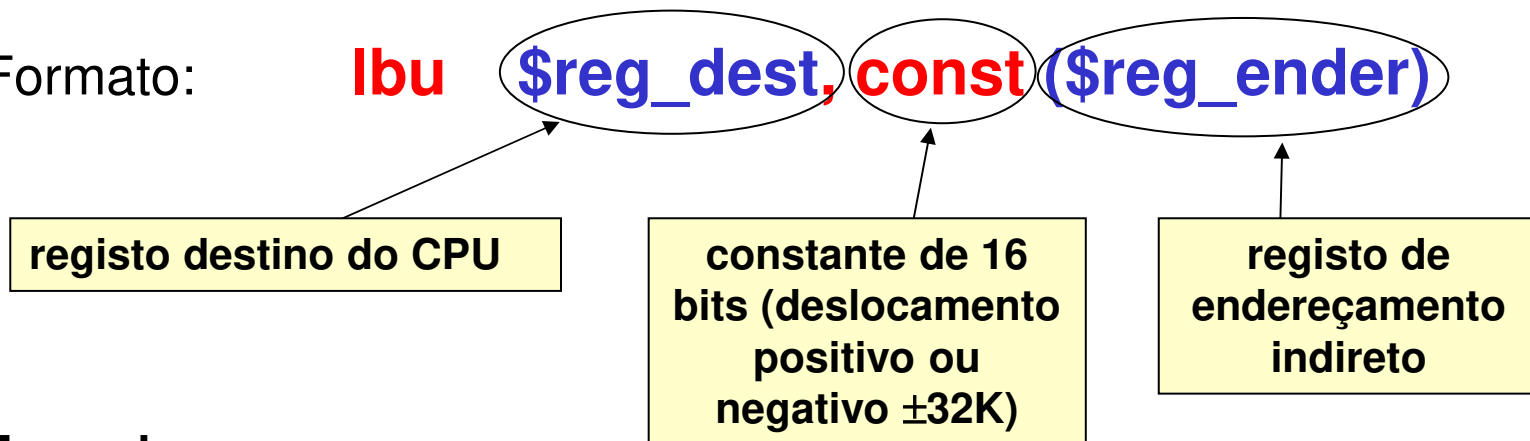
Exemplo:

sb \$7, 5 (\$4) # transfere o *byte* armazenado no registo \$7 (8
bits menos significativos) para o endereço de
memória calculado como:
addr = (conteúdo do registo \$4) + 5

Instrução de leitura de 1 *byte* na memória - LBU

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

Formato:



Exemplo:

Ibu \$5, -3 (\$2) # transfere para o registro \$5 o *byte* armazenado
no endereço de memória calculado como:
addr = (conteúdo do registro \$2) - 3
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:

Ib \$5, 0 (\$2)

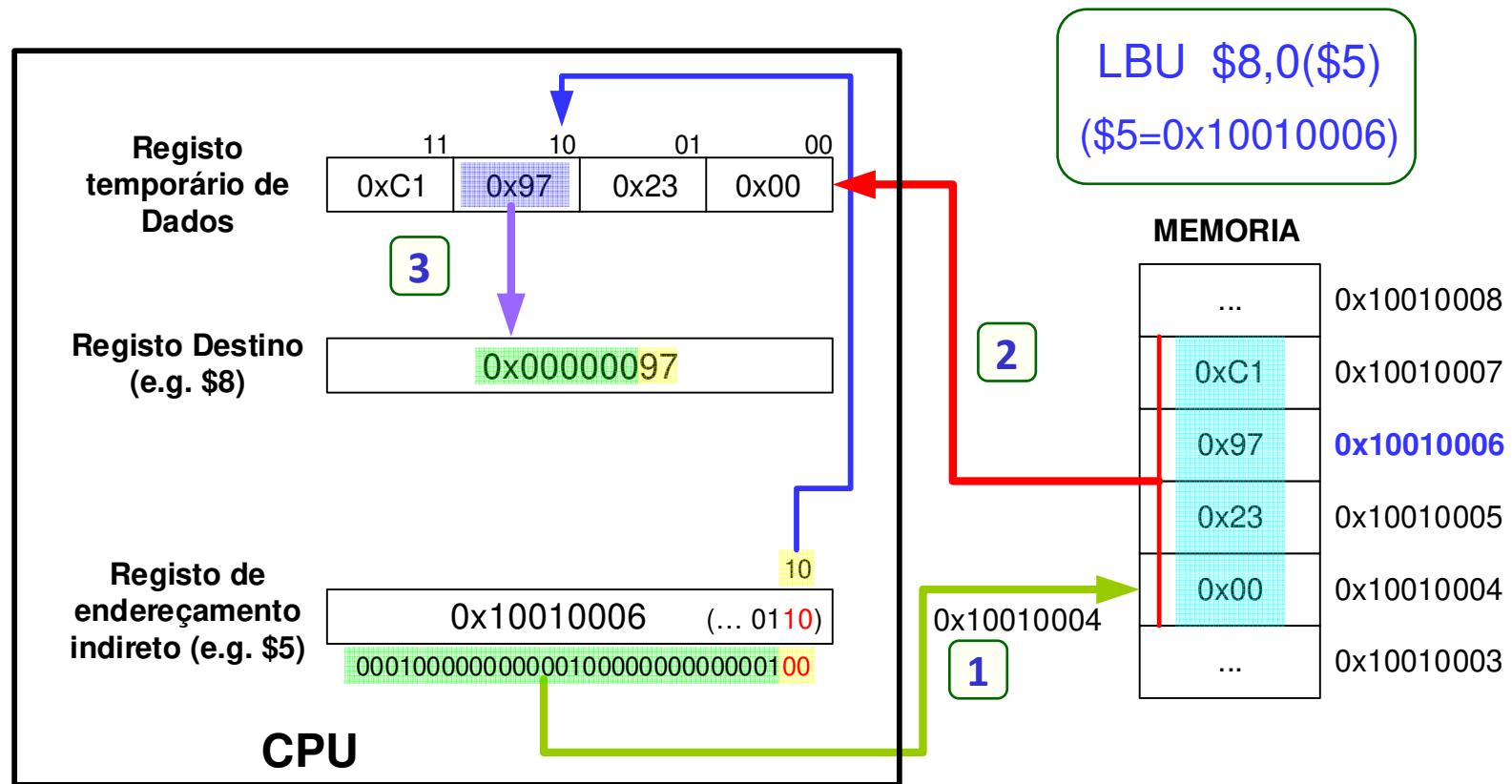
- # transfere para o registo \$5 o *byte* armazenado
- # no endereço de memória calculado como:
- # **$addr = (\text{conteúdo do registo } \$2) + 0$**
- # o bit mais significativo do *byte* transferido é
- # replicado nos 24 bits mais significativos de \$5

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* de 32 bits inclui o endereço pretendido
 - No caso de **Leitura** (instruções **lb**, **lbu**):
 - 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**"

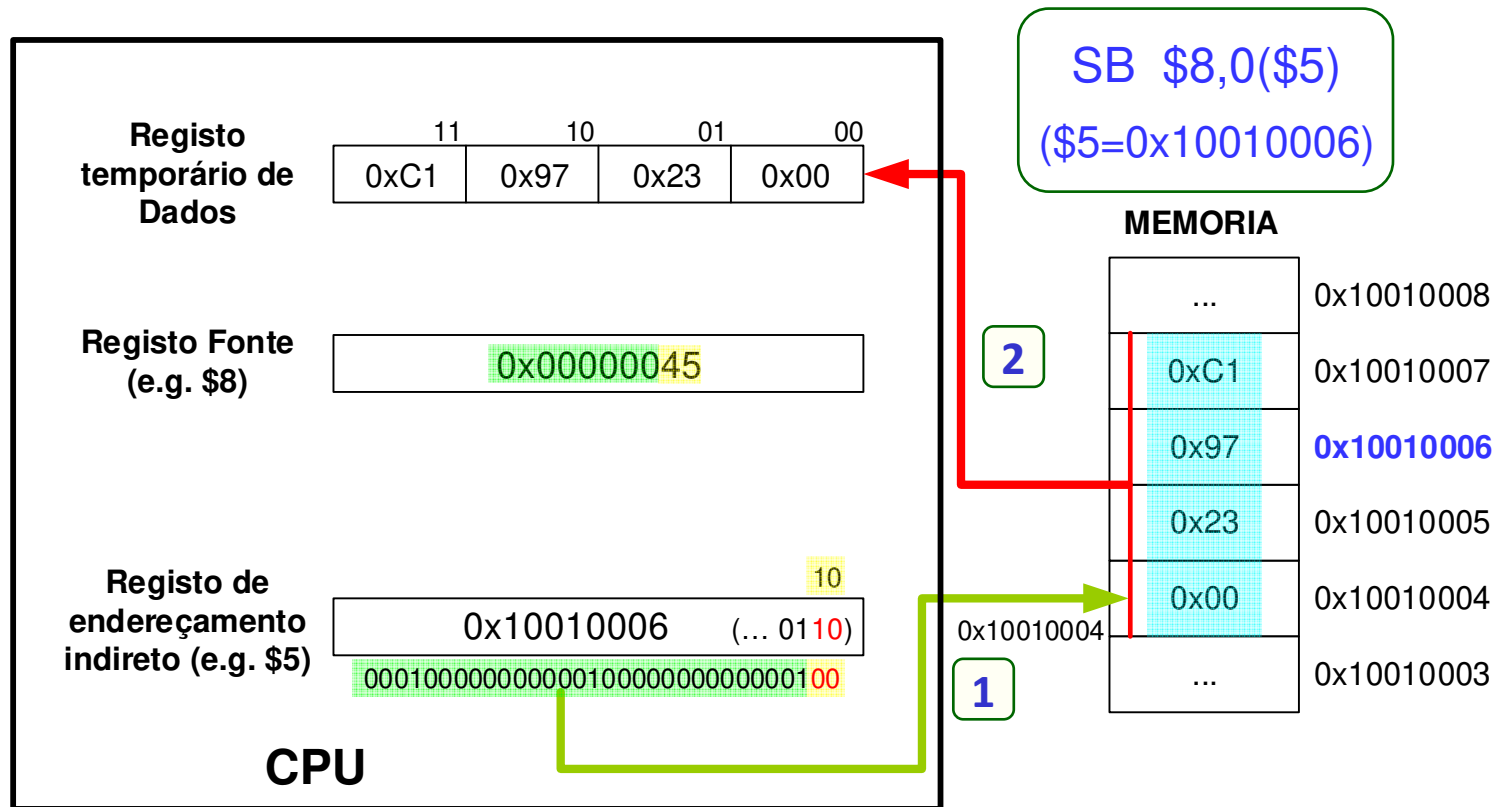
Exemplo: leitura de 1 *byte* da memória

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



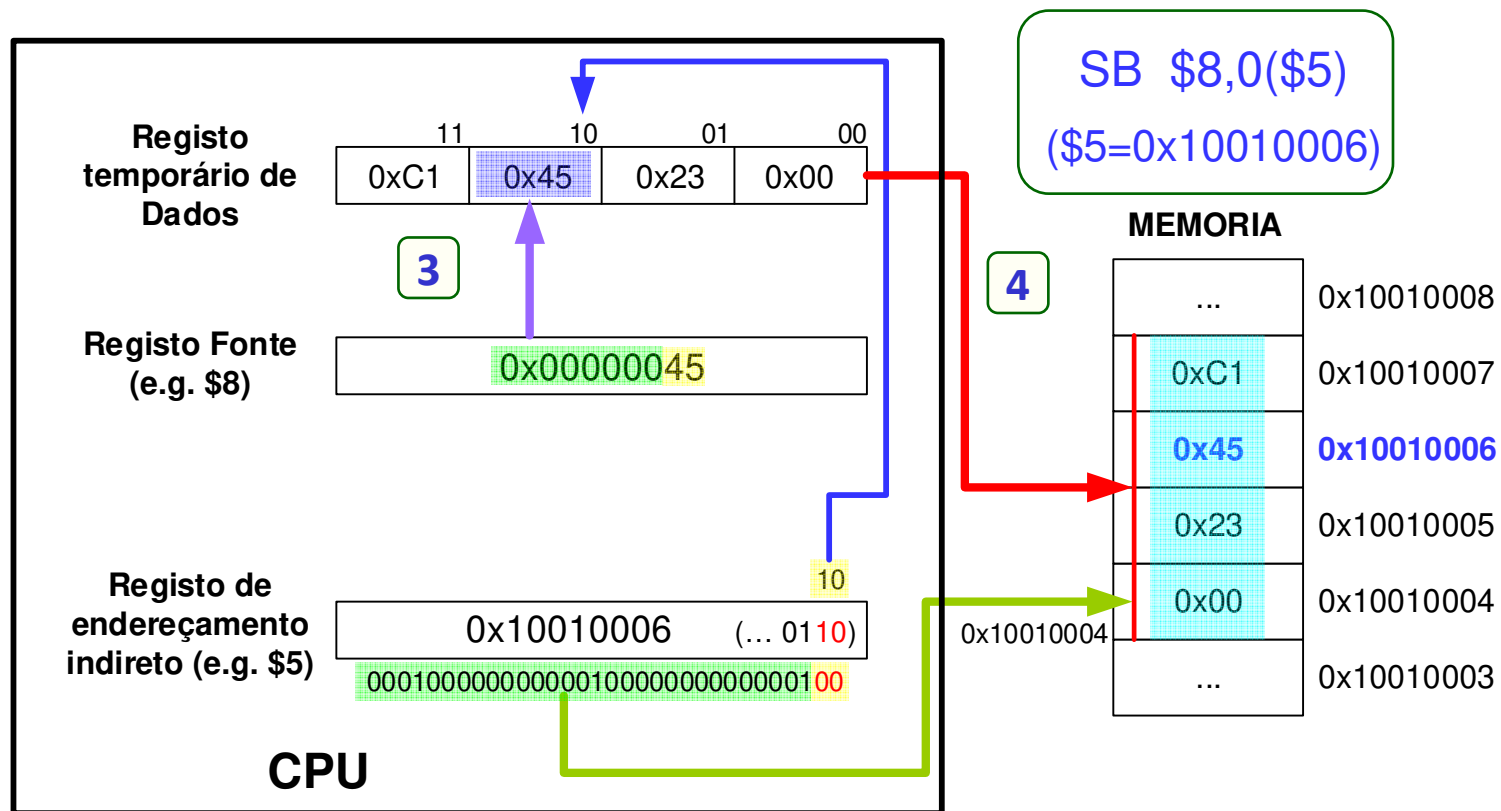
Exemplo: escrita de 1 *byte* da memória (fase 1)

- Exemplo para o caso da escrita (instrução **sb** a copiar um valor para a posição de memória **0x10010006**) - **READ**



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

- Exemplo para o caso da escrita (instrução **sb** a copiar um valor para a posição de memória **0x10010006**) - **MODIFY / WRITE**



Organização das *words* de 32 bits 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 (***big-endian***)
 - *byte* menos significativo armazenado no endereço mais baixo da memória (***little-endian***)

Organização das words de 32 bits na memória

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

Address	Data
0x10010008	?
0x10010009	?
0x1001000A	?
0x1001000B	?
0x1001000C	0x01
0x1001000D	0x23
0x1001000E	0x87
0x1001000F	0xA5
0x10010010	?
0x10010011	?
0x10010012	?
0x10010013	?
0x10010014	?
...	...



Big-Endian

Address	Data
0x10010008	?
0x10010009	?
0x1001000A	?
0x1001000B	?
0x1001000C	0xA5
0x1001000D	0x87
0x1001000E	0x23
0x1001000F	0x01
0x10010010	?
0x10010011	?
0x10010012	?
0x10010013	?
0x10010014	?
...	...



Little-Endian

- O simulador MARS implementa "little-endian"

Diretivas do *Assembler*

- Diretivas são comandos especiais colocados num programa em linguagem *assembly* destinados a instruir o *assembler* a executar uma determinada tarefa ou função
- 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 MARS

.ASCIIZ *str*

Reserva espaço e armazena a string *str* em sucessivas posições de memória; acrescenta o terminador '`\0`' (**NUL**)

Ex:

msg1: .ASCIIZ "Arquitetura de Computadores"

.SPACE *n*

Reserva *n* posições consecutivas (endereços) de memória, sem inicialização

Ex:

array: .SPACE 20

.BYTE b_1, b_2, \dots, b_n Reserva espaço e armazena os bytes b_1, b_2, \dots, b_n em sucessivas posições de memória

Ex:

array: .BYTE 0x41, 0x43, 0x31, 0x00

Diretivas do *Assembler* do MARS

.WORD w_1, w_2, \dots, w_n Reserva espaço e armazena as *words* w_1, w_2, \dots, w_n em sucessivas posições de memória (cada *word* em 4 endereços consecutivos)

Ex:

array: .WORD 0x012387A5, 0xF34, 0x678AC

.ALIGN n Alinha o próximo item num endereço múltiplo de 2^n

Ex:

.ALIGN 2 # próximo item está alinhado num endereço
múltiplo de 4

.EQV *symbol, val* Atribui a um símbolo um valor. No programa o assembler substitui as ocorrências de ***symbol*** por ***val***

Ex:

.EQV TRUE, 1
.EQV FALSE, 0

Diretivas do Assembler - exemplo

.DATA	# 0x10010000	0x10010017	??
		0x10010016	??
STR1: .ASCIIZ	"AULA5"	0x10010015	??
		VARW 0x10010014	??
ARR1: .WORD	0x1234, MAIN	0x10010013	?? (unused)
		0x10010012	?? (unused)
VARB: .BYTE	0x12	0x10010011	?? (unused)
		VARB 0x10010010	0x12
.ALIGN	2	0x1001000F	0x00
VARW: .SPACE	4 #space for 1 word	0x1001000E	0x40
		0x1001000D	0x00
.TEXT	# 0x00400000	0x1001000C	0x00
		0x1001000B	0x00
.GLOBL	MAIN	0x1001000A	0x00
MAIN:		0x10010009	0x12
		ARR1 0x10010008	0x34
		0x10010007	?? (unused)
		0x10010006	?? (unused)
		0x10010005	'\0' (0x00)
		0x10010004	'5' (0x35)
		0x10010003	'A' (0x41)
		0x10010002	'L' (0x4C)
		0x10010001	'U' (0x55)
		STR1 0x10010000	'A' (0x41)

- Utilizar a diretiva **".align"** sempre que se pretender que o endereço subsequente esteja alinhado
- A diretiva **".word"** alinha automaticamente num endereço múltiplo de 4

Questões / exercícios

- Qual o modo de endereçamento usado pelo MIPS para acesso a quantidades residentes na memória externa?
- Na instrução "**lw** \$3, 0x24(\$5)" qual a função dos registos \$3 e \$5 e da constante 0x24?
- Qual o formato de codificação das instruções de acesso à memória no MIPS e qual o significado de cada um dos seus campos?
- Qual a diferença entre as instruções "**sw**" e "**sb**"? O que distingue as instruções "**lb**" e "**lbu**"?
- O que acontece quando uma instrução **lw/sw** acede a um endereço que não é múltiplo de 4?
- Sabendo que o *opcode* da instrução "**lw**" é 0x23, determine o código máquina, expresso em hexadecimal, da instrução "**lw** \$3, 0x24(\$5)".

Questões / exercícios

- Suponha que a memória externa foi inicializada, a partir do endereço **0x10010000**, com os valores **0x01, 0x02, 0x03, 0x04, 0x05, ...**. Suponha ainda que **\$3=0x1001** e **\$5=0x10010000**. Qual o valor armazenado no registo destino após a execução da instrução **"lw \$3, 0x24(\$5)"**?
- Nas condições anteriores qual o valor armazenado no registo destino pelas instruções: **"lbu \$3, 0xA3(\$5)"** e **"lb \$4, 0xA3(\$5)"**?
- Quantos bytes são reservados em memória por cada uma das diretivas:
L1: .asciiz "Aulas5&6T"
L2: .word 5, 8, 23
L3: .byte 5, 8, 23
L4: .space 8
- Acrescente a diretiva **".align 2"** a seguir a L3. Desenhe esquematicamente a memória e preencha-a com o resultado das diretivas anteriores.
- Supondo que **"L1"** corresponde ao endereço inicial do segmento de dados, e que esse endereço é **0x10010000**, determine os endereços a que correspondem os *labels* **"L2"**, **"L3"** e **"L4"**, nas condições da questão anterior.

Aula 7

- Utilização de ponteiros em linguagem C
- Acesso sequencial aos elementos de um *array*:
 - acesso indexado
 - acesso com ponteiro
- Tradução para *assembly* do MIPS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Introdução

- Em linguagem C, em muitas situações é necessário saber qual o endereço de memória onde reside uma variável, para então aceder ao seu conteúdo
- Por exemplo, para imprimir no ecrã os caracteres de uma *string* (array de caracteres) é necessário ler sequencialmente cada uma das posições de memória onde a *string* se encontra alojada (que são identificadas pelo seu endereço)
- O acesso a cada um dos caracteres é feito indiretamente através do endereço onde residem:
 - conhecido o endereço inicial da *string* em memória, o acesso sequencial é garantido pelo incremento sucessivo do endereço
- A linguagem C providencia um mecanismo para acesso a variáveis residentes na memória externa através da utilização de ponteiros

Linguagem C: ponteiros e endereços – o operador &

- Um **ponteiro** é uma **variável que contém o endereço de outra variável** – o acesso à 2ª variável pode fazer-se indiretamente através do ponteiro
- Se **var** é uma variável, então **&var** dá-nos o seu endereço
- 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);
```

Linguagem C: ponteiros e endereços – o operador &

- Exemplo:

```
#include <stdio.h>
void main(void)
{
    int age = 59;
    printf("Value of variable age is: %d\n", age);
    printf("Address of variable age is: %p\n", &age);
}
```

- O primeiro `printf()` imprime o valor da variável: 59
- O segundo `printf()` imprime o endereço da posição de memória onde reside a variável:
 - se este código executar num processador com arquitetura MIPS é um valor de 32 bits

Ponteiros e endereços – o operador *

- O operador "*":
 - trata o seu operando como um endereço
 - permite o acesso ao endereço para obter ou alterar 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 a expressão: **y = x;**

- O operador "*", é designado por **operador de indireção**

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, ou seja,
           // não há qualquer inicialização
```

- 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
```

Ponteiros e endereços – declaração de variáveis

- Exemplo:

```
#include <stdio.h>
void main(void)
{
    int age = 59;
    int *p = &age;

    printf("Value of variable age is: %d\n", age);
    printf("Value pointed by p is: %d\n", *p);
}
```

- O segundo **printf()** imprime o conteúdo da variável apontada pelo ponteiro "p"
- O valor impresso pelos dois **printf()** é o mesmo

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 das variáveis originais)
- Assim, uma função chamada não pode alterar diretamente o valor de uma variável da função chamadora
- Então, no código seguinte:

```
void change(int a)
{
    a = 10;
}

void main(void)
{
    int b = 25;
    change(b);           // Para a função é passado o valor
                        // da variável (passagem por valor)
    printf("%d", b);    // Imprime o valor de b
}
```

- Qual o valor de "b" após a chamada à função "change ()"?
- A função alterou o valor da cópia da variável, pelo que, "b" mantém o valor original, ou seja, 25

Ponteiros como argumentos de funções

- Se pretendermos que a função altere o valor da variável da função chamadora, então teremos que passar como argumento da função o endereço da variável, ou seja, um ponteiro para a variável

```
void change(int *a)      // argumento de entrada é um
{                          // ponteiro para um inteiro
    *a = 10;
}

void main(void)
{
    int b = 25;
    change(&b); // Para a função é passado o endereço
                // da variável (passagem por referência)
    printf("%d", b);
}
```

- Qual o valor de "b" após a chamada à função "change ()"?
- A função acedeu à variável da função chamadora através do seu endereço, pelo que "b" passa a ter o valor 10

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* representa 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 (considerando **pa** um ponteiro para inteiro)

Exemplo 1

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

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

    int i = 0;

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

H	e	l	l	o	\0
---	---	---	---	---	----

```
// "s" é um array de
// caracteres (string)
// terminado com o
// caractere '\0' (0x00)
```

Exemplo 2

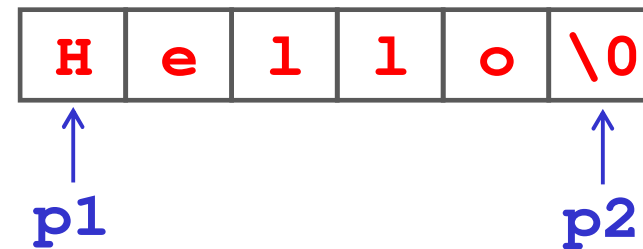
```
void main(void)
{
    char s[] = "Hello";
    char *p;    // Declara um ponteiro para
                // carater (reserva espaço)
    p = s;      // Inicializa o ponteiro com o
                // endereço inicial do array
    while(*p != '\0')
    {
        printf("%c", *p); // imprime carater
        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 carácter a imprimir (*p)
- O ponteiro "p" é incrementado após o acesso ao conteúdo (pós-incremento).
- Esta versão é equivalente à do exemplo 2
- Qual seria o resultado do programa se *p++ fosse substituído por *(++p) ?

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 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:

`v = 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:

`v = *p;` // com `p` = endereço de `a[i]` (i.e. **`p = &a[i]`**)

- Estas 2 formas de acesso traduzem-se em **implementações distintas** em *assembly*

Acesso sequencial a elementos de um *array*

- **Acesso indexado**

- $v = a[i];$ // 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***
- Por exemplo, se se tratar de um *array* de inteiros, o endereço do elemento 2 está 8 endereços à frente do endereço do elemento 0

$\&a[0] \rightarrow$

Address	Data
0x00000020	0x45
0x00000021	0x12
0x00000022	0x3A
0x00000023	0xF3
0x00000024	0xC9
0x00000025	0x7D
0x00000026	0xB3
0x00000027	0x9D
0x00000028	0x47
0x00000029	0x5F
0x0000002A	0x6D
0x0000002B	0x4A
0x0000002C	0xFD
0x0000002D	0xC0
0x0000002E	0x5A
0x0000002F	0x7C
0x00000030	0x1D
...	...

$\&a[2] \rightarrow$

Diagram illustrating memory access for an array *a*. The array is represented as a table of Address and Data. The elements are grouped into four sets: *a*[0] (addresses 0x00000020 to 0x00000023), *a*[1] (addresses 0x00000024 to 0x00000027), *a*[2] (addresses 0x00000028 to 0x0000002B), and *a*[3] (addresses 0x0000002C to 0x0000002F). The pointer $\&a[0]$ points to the first address (0x00000020), and $\&a[2]$ points to the first address of the third group (0x00000028).

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

Acesso sequencial a elementos de um *array*

- **Acesso por ponteiro**

- $v = *p;$
- O endereço do elemento a aceder está armazenado num registo

Address	Data
0x00000020	0x45
0x00000021	0x12
0x00000022	0x3A
0x00000023	0xF3
0x00000024	0xC9
0x00000025	0x7D
0x00000026	0xB3
0x00000027	0x9D
0x00000028	0x47
0x00000029	0x5F
0x0000002A	0x6D
0x0000002B	0x4A
0x0000002C	0xFD
0x0000002D	0xC0
0x0000002E	0x5A
0x0000002F	0x7C
0x00000030	0x1D
...	...

Diagram illustrating sequential access to array elements using pointers:

- $p \rightarrow$ points to address 0x00000024 (start of $a[1]$)
- $p+1 \rightarrow$ points to address 0x00000028 (start of $a[2]$)
- $p+2 \rightarrow$ points to address 0x0000002C (start of $a[3]$)

Array elements and their corresponding addresses:

- $a[0]$: 0x00000020 to 0x00000023
- $a[1]$: 0x00000024 to 0x00000027
- $a[2]$: 0x00000028 to 0x0000002B
- $a[3]$: 0x0000002C to 0x0000002F

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])
```

```
.data
array: .space 40
      .eqv  SIZE, 10
```

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

```
.text
.globl main
```

```
main:  li    $t0, 0
```

```
# i = 0;
```

```
for:   bge   $t0, SIZE, 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
```

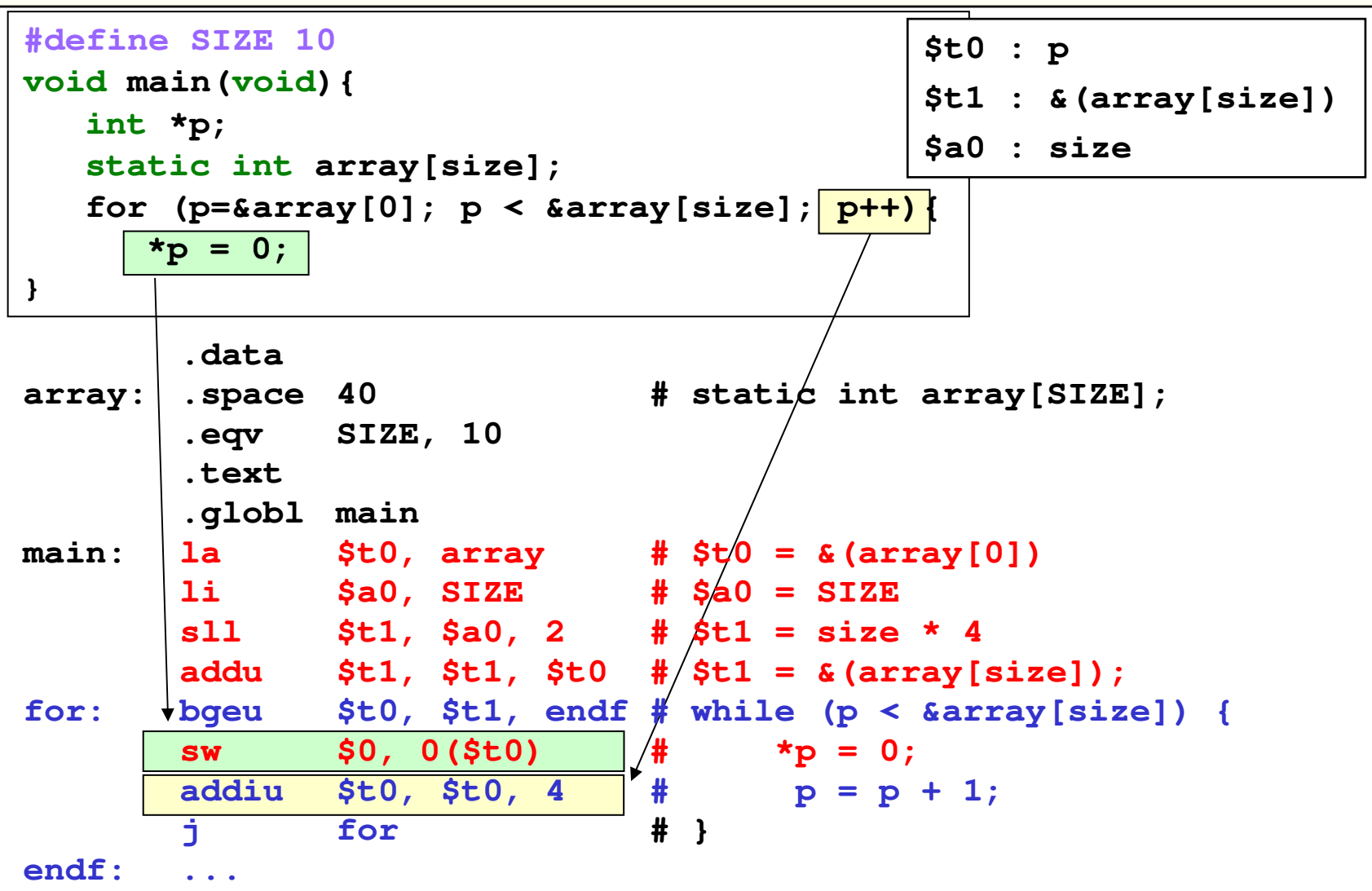
```
# i = i + 1;
```

```
    j     for
```

```
# }
```

```
endf:  ...
```

Acesso sequencial a arrays – exemplo 2



Questões

- O que significa a declaração `int *ac;`? Qual a diferença entre essa declaração e `int ac`?

O que significa a declaração `char *ac;`?

- A partir das declarações de `a` e `b`:

```
int a;
```

```
int *b;
```

identifique quais das seguintes atribuições são válidas:

```
a=b;      b=*a;      b=&(a+1);  a=&b;      b=&a;
b=*a+1;  b=*(a+1);  a=*b;      a=*(b+1);  a=*b+1;
```

- Identifique as operações, e respetiva sequência, realizadas nas seguintes instruções C:

```
a=*b++;  a=*(b)++;  a=*(++b);
```

- Suponha que `p` está declarado como `int *p;`. Supondo que a organização da memória é do tipo *byte-addressable*, qual o incremento no endereço que é obtido pela operação `p=p+2;` ?

Questões

- Suponha que "**b**" é um *array* declarado como "**int b[25];**". Como é obtido o endereço inicial do *array*, i.e., o endereço da sua primeira posição? Supondo uma memória "*byte-addressable*", como é obtido o endereço do elemento "**b[6]**"?
- Dada a seguinte sequência de declarações:

```
int b[25];  
int a;  
int *p = b;
```

Identifique qual ou quais das seguintes atribuições permitem aceder ao elemento de índice 5 do *array* "**b**":

```
a = b[5];           a = *p + 5;  
a = *(p + 5); a = *(p + 20);
```


Exercício

- Pretende-se escrever uma função para a troca do conteúdo de duas variáveis (**troca(a, b);**). Isto é, se, antes da chamada à função, a=2 e b=5, então, após a chamada à função, os valores de a e b devem ser: a=5 e b= 2

Uma solução incorreta para o problema é a seguinte:

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

    aux = x;
    x = y;
    y = aux;
}
```

- Identifique o erro presente no trecho de código e faça as necessárias correções para que a função tenha o comportamento pretendido

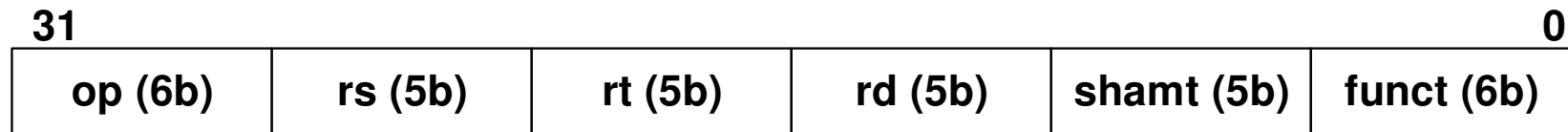
Aula 8

- 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

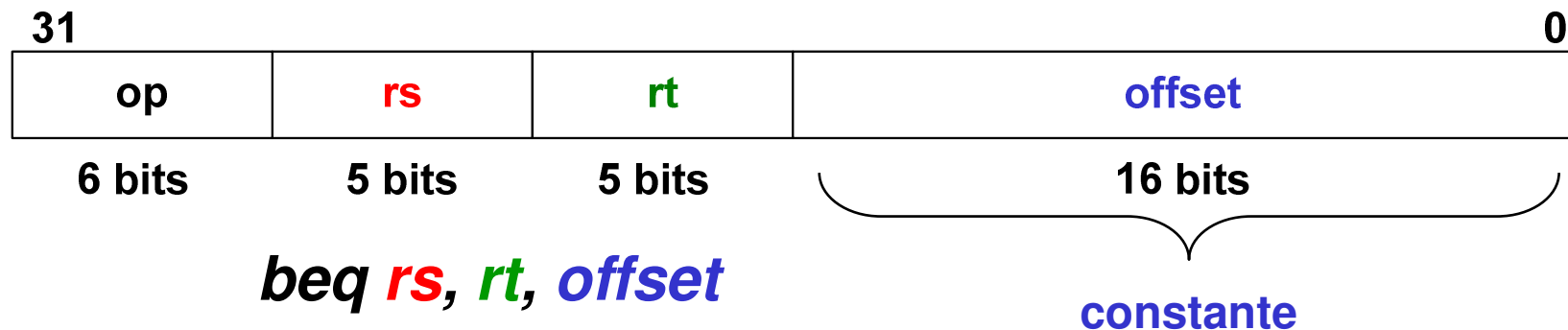
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Formatos de codificação no MIPS

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

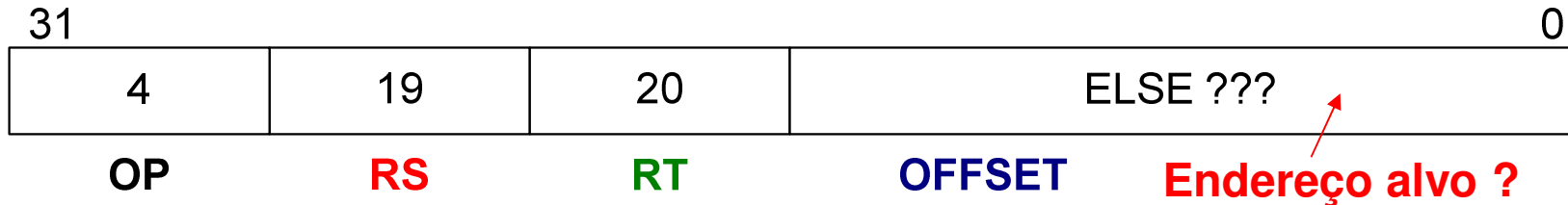


- A necessidade de codificação do **endereço-alvo** das instruções de salto condicional obriga a que estas instruções sejam codificadas recorrendo ao **formato I**



Codificação de *branches* – método geral

Exemplo: **beq \$19, \$20, ELSE** # "ELSE" representa o endereço-alvo



- Se o endereço alvo fosse codificado diretamente nos 16 bits menos significativos da instrução, isso significaria que o programa não poderia ter uma dimensão superior a 2^{16} (64K)...
- Em vez de um endereço absoluto, o campo *offset* pode ser usado para codificar a **diferença** entre o valor do endereço-alvo e o endereço onde está armazenada a instrução de *branch*
- O *offset* é interpretado como um **valor em complemento para dois**, permitindo o salto para **endereços anteriores** (*offset* negativo) ou **posteriores** (*offset* positivo) ao PC
- Durante a execução da instrução de *branch* o seu endereço está disponível no registo PC, pelo que o processador pode calcular o endereço-alvo como: **Endereço-alvo = PC + offset**
- Endereçamento relativo (**PC-relative addressing**)

Codificação de *branches* no MIPS

- No MIPS, na fase de execução de um *branch*, o PC corresponde ao endereço da instrução seguinte (o PC é incrementado na fase “*fetch*” da instrução)
- Por essa razão, na codificação de uma instrução de *branch*, **a referência para o cálculo do *offset* é o endereço da instrução seguinte**
- As instruções estão armazenadas em memória em endereços múltiplos de 4 (e.g., **0x00400004**, **0x00400008**,...) pelo que o *offset* é também um valor múltiplo de 4 (2 bits menos significativos são sempre 0)
- De modo a otimizar o espaço disponível para o *offset* na instrução, os dois bits menos significativos não são representados

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:
    
```

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

O endereço correspondente ao
label ELSE é 0x0040000C

O "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 "offset" é também múltiplo de 4 Logo:

"offset" = $0x08 / 4 = 0x02$ (offset em número de instruções!!!)

31				0
	5	19	20	0x0002

Código máquina: **0001011001110100000000000000010** = **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.

Execução de uma instrução de *branch*

- O campo *offset* do código máquina da instrução de *branch* é então usado para codificar a **diferença** entre o valor do endereço-alvo e o valor do endereço seguinte ao da instrução de *branch*, **dividida por 4**
- Durante a execução da instrução, o processador calcula o endereço-alvo como:

$$\text{Endereço_alvo} = \text{PC_atual} + (\text{offset} * 4)$$

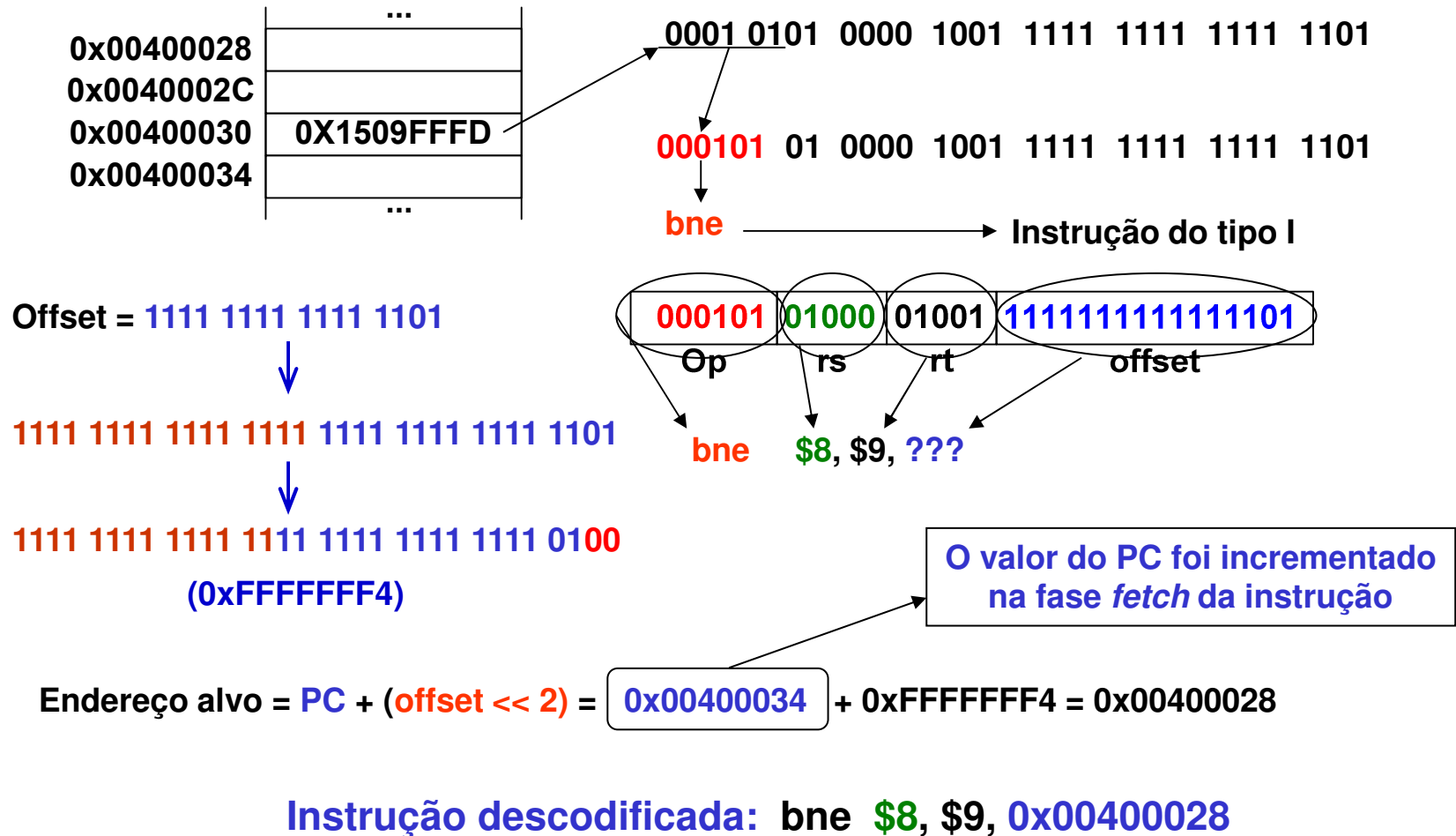
ou:

$$\text{Endereço_alvo} = \text{PC_atual} + (\text{offset} \ll 2)$$

(o offset de 16 bits é estendido com sinal para 32 bits, antes do *shift*)

Interpretação de uma instrução de *branch* no MIPS

Exemplo



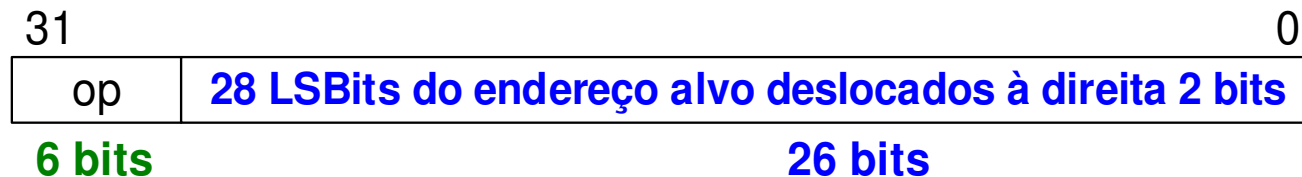
Codificação da instrução de salto incondicional

- 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**

- Formato J:



- Endereço alvo da instrução "j" é sempre múltiplo de 4 (2 bits menos significativos são sempre 0)



Codificação da instrução de salto incondicional

- Exemplo: **j Label** # com Label = 0x001D14C8

0x001D14C8: 0000 0000 0001 1101 0001 0100 1100 1000



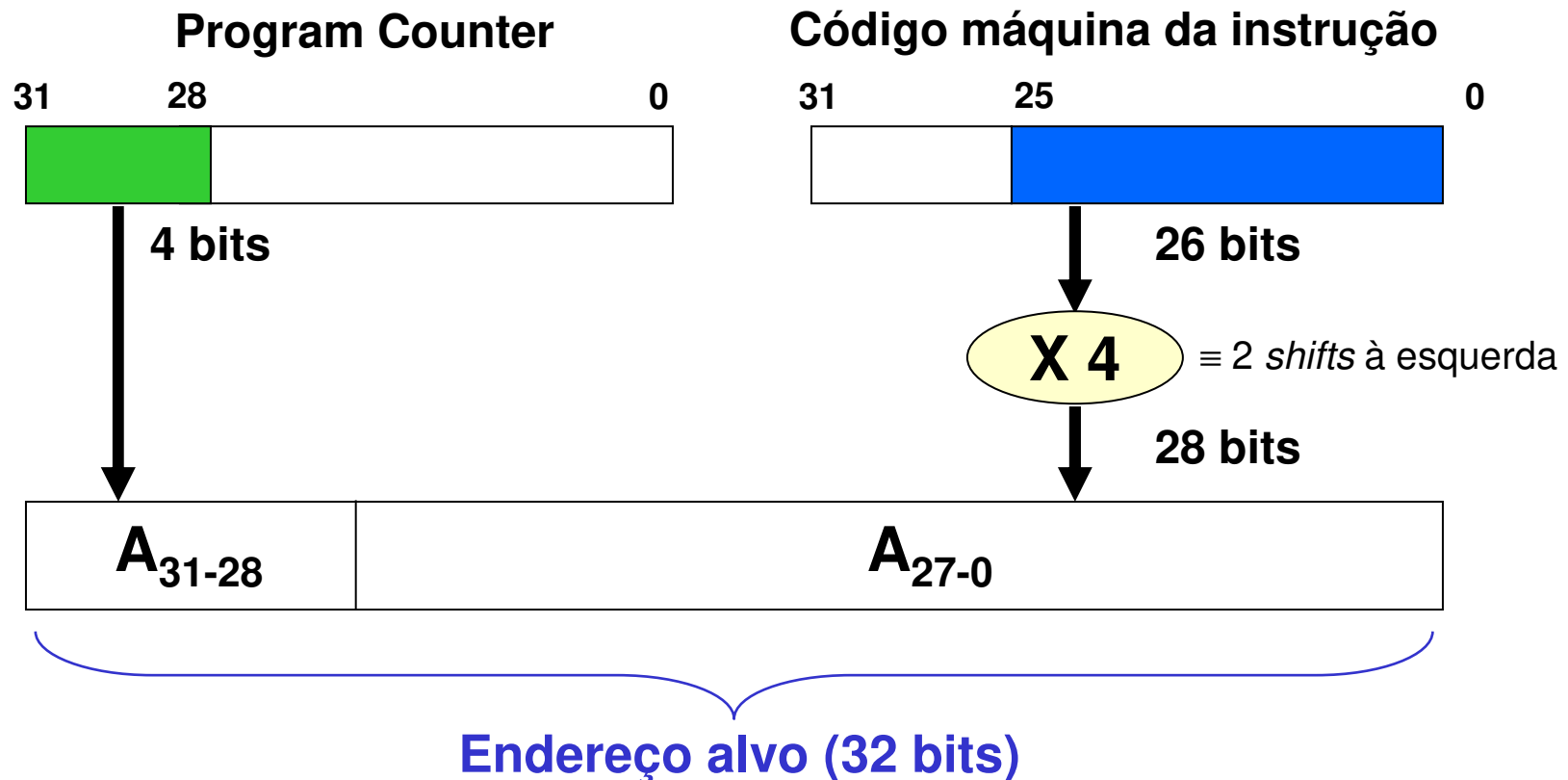
(26 bits) 00 0000 0111 0100 0101 0011 0010

- Código máquina (opcode do "j" é 0x02):

0000 1000 0000 0111 0100 0101 0011 0010 = 0x08074532

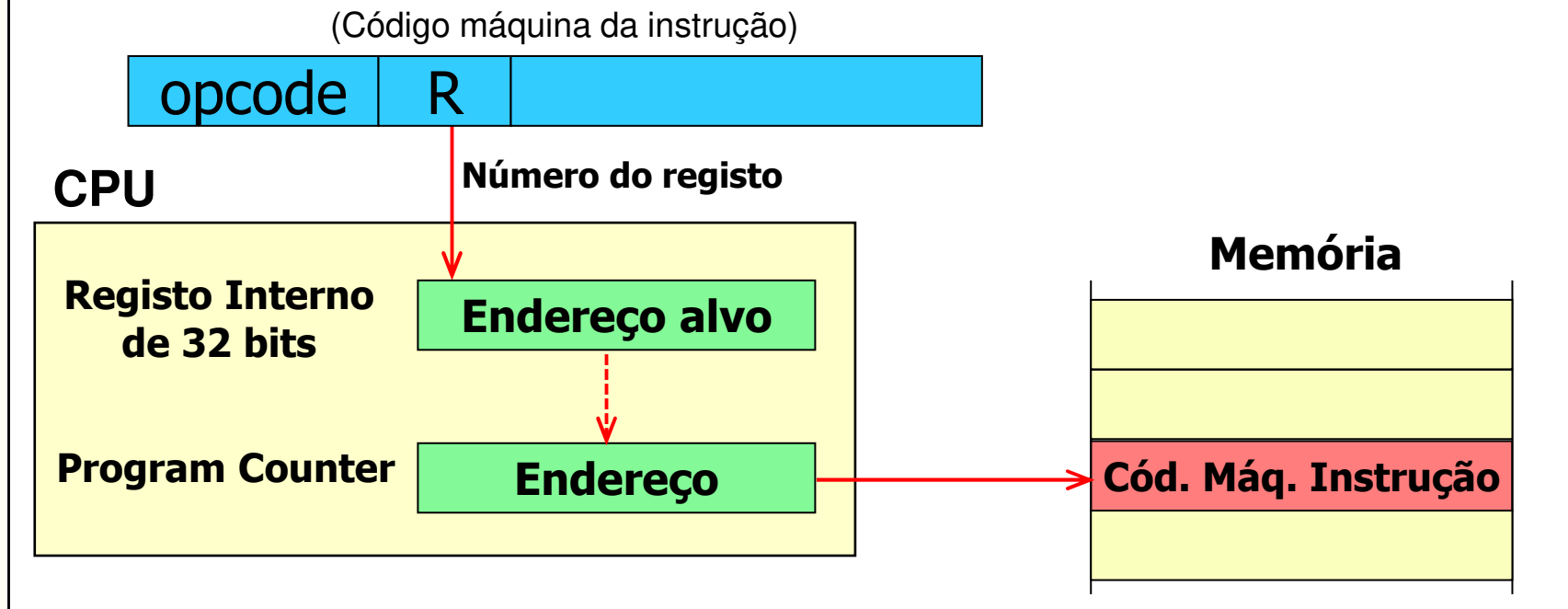
Cálculo do endereço-alvo de uma instrução J

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 que realize um 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** - Jump register)



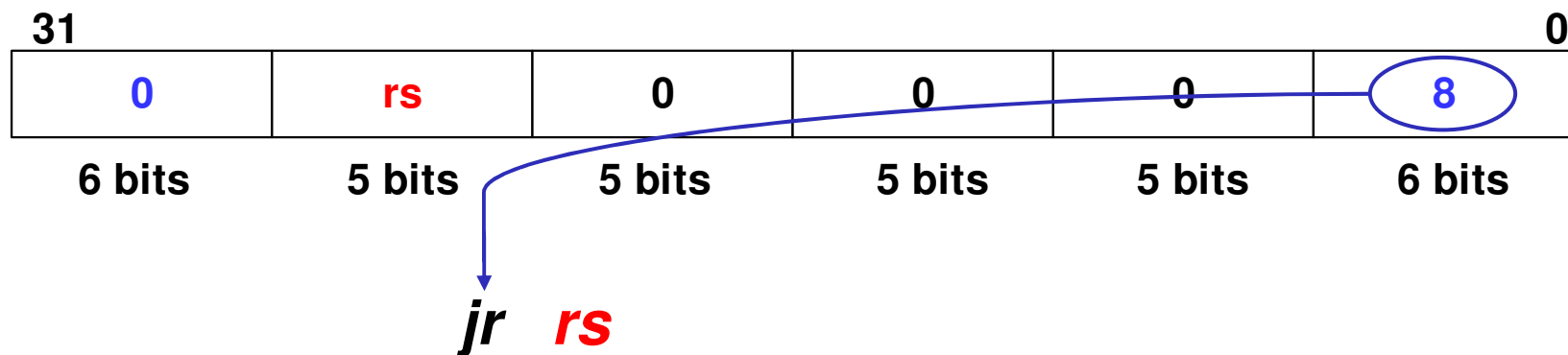
Instrução JR (jump on register)

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:



Manipulação de constantes

- Constante é um valor determinado com antecedência (quando o programa é escrito) e que não se pretende que seja ou possa ser mudado durante a execução do programa
- 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
- Para aumentar a eficiência, as arquiteturas disponibilizam 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: “**endereço imediato**”

Manipulação de constantes no MIPS

- As instruções aritméticas e lógicas que manipulam constantes (do tipo imediato) são identificadas pelo sufixo “i”:

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

- Estas instruções são codificados usando o **formato I**. Logo apenas **16 bits** podem ser usados para codificar a constante
- 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...

Manipulação de constantes no MIPS

- 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 0xFFFC
addi $4, $2, 0x15     # 2110
slti $6, $7, 0xFFFF  # -110
```

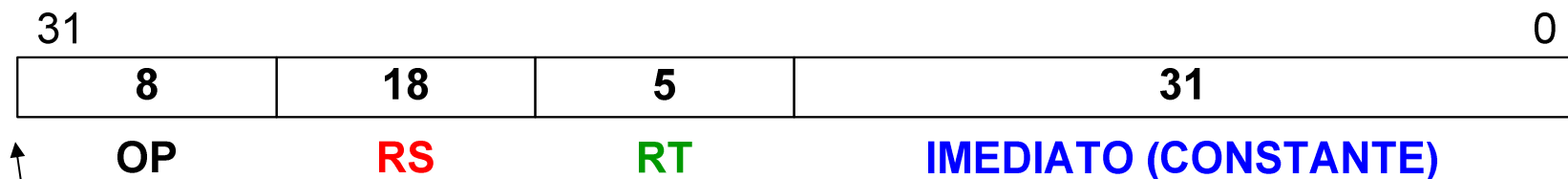
- Gama de representação da constante: **[-32768, +32767]**
 - A constante de 16 bits é estendida para 32 bits, preservando o sinal (ex: para -4, **0xFFFC** é estendido para **0xFFFFF**)
- Existem também instruções em que a constante deve ser entendida como uma quantidade inteira sem sinal. Estão neste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

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

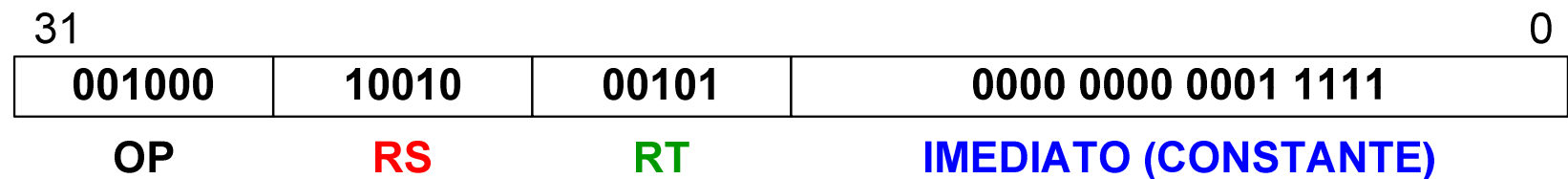
- Gama de representação da constante: **[0, 65535]**
- A constante de 16 bits é estendida para 32 bits, sendo os 16 mais significativos **0x0000** (para o exemplo: **0x0000FFFF**)

Codificação das instruções que usam constantes

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



addi rt, rs, immediate



Cod. Máquina: 00100010010001010000000000011111 = 0x2245001F

Manipulação de constantes de 32 bits – LUI

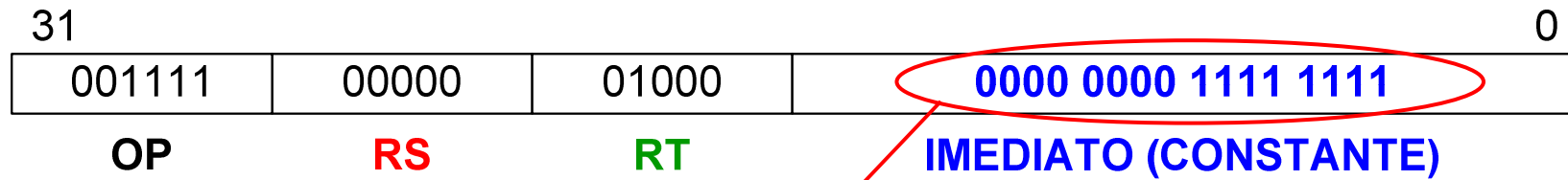
- Em alguns casos pode ser necessário manipular constantes que necessitem de um espaço de armazenamento com mais do que 16 bits (e.g., a referência explícita a um endereço)
- Como lidar com esses casos?
- Para permitir a manipulação de constantes com mais de 16 bits, o ISA do MIPS inclui a seguinte instrução, também codificada com o formato I:

lui \$reg, immediate

- A instrução **lui** ("Load Upper Immediate"), coloca a constante "immediate" nos **16 bits mais significativos do registo destino** (\$reg)
- Os 16 bits menos significativos ficam com **0x0000**

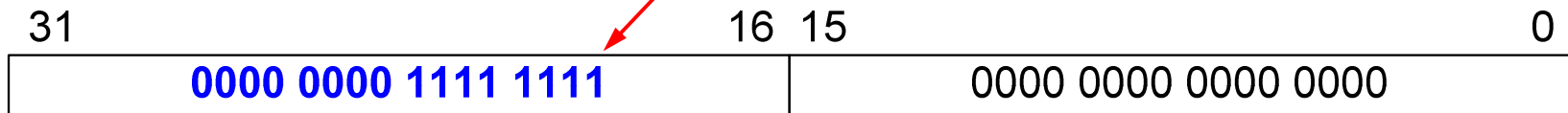
Manipulação de constantes de 32 bits – LUI

Exemplo: `lui $8, 255 # 25510 = 0xFF`



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

- Exemplo: inicializar o registo `$6` com o valor `0xF32864D9`

`lui $6, 0xF328 # $6 = 0xF3280000`

`ori $6, $6, 0x64D9 # $6 = 0xF3280000 | 0x000064D9 = 0xF32864D9`

Manipulação de constantes de 32 bits – LA / LI

A instrução virtual "load address"

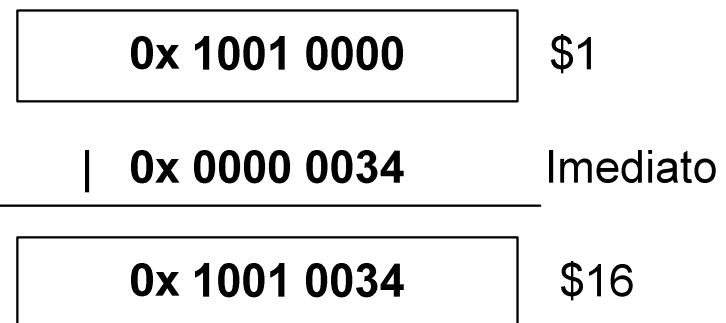
```
la    $16, MyData #Ex. MyData = 0x10010034
      # (segmento de dados em 0x1001000)
```

é executada no MIPS pela sequência de **instruções nativas**:

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

Notas:

- O **registro \$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”

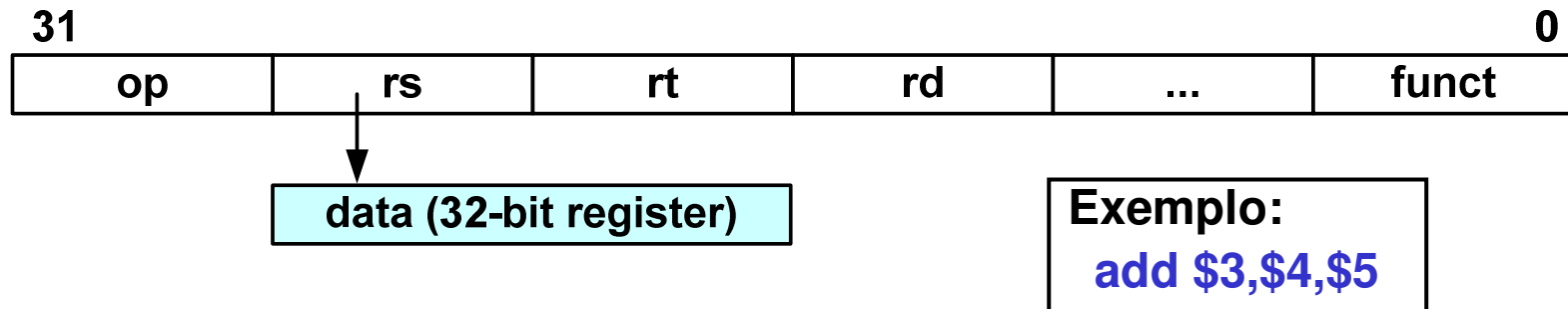


Modos de endereçamento no MIPS (resumo)

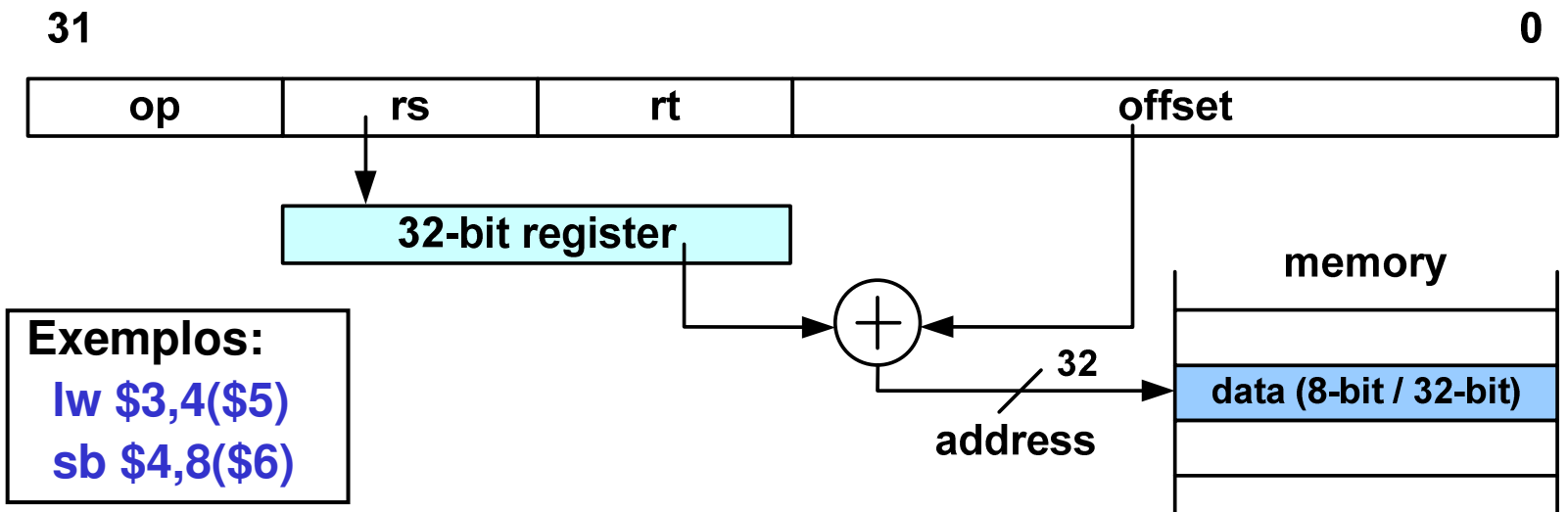
- Instruções aritméticas e lógicas: **endereçamento tipo registo**
- Instruções aritméticas e lógicas com constantes: **endereçamento imediato**
- Instruções de acesso à memória: **endereçamento indireto por registo com deslocamento**
- Instruções de salto condicional (*branches*): **endereçamento relativo ao PC**
- Instrução de salto incondicional através de um registo (instrução **JR**): **endereçamento indireto por registo**
- Instrução de salto incondicional (**J**): **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 do MIPS (resumo)

- Register Addressing (endereço tipo registro):

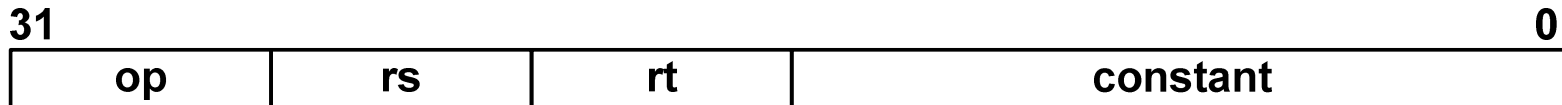


- Base addressing (indireto por registro com deslocamento):



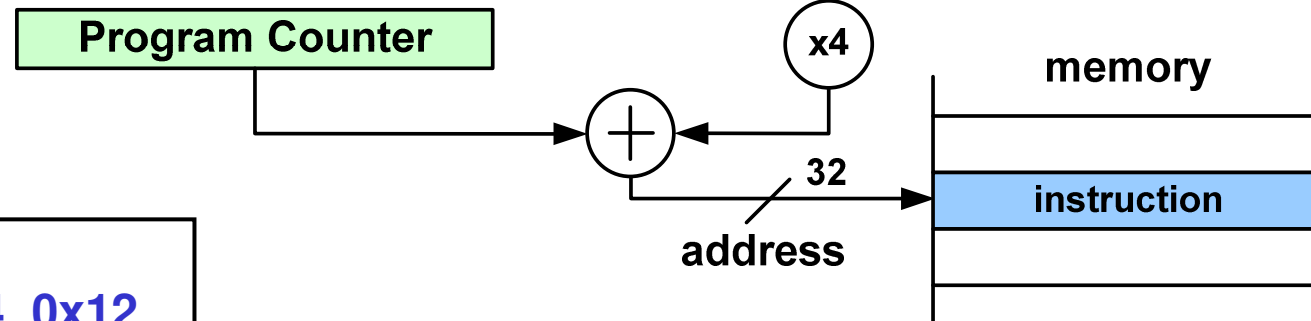
Modos de endereçamento do MIPS (resumo)

- Immediate Addressing (endereço imediato):



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

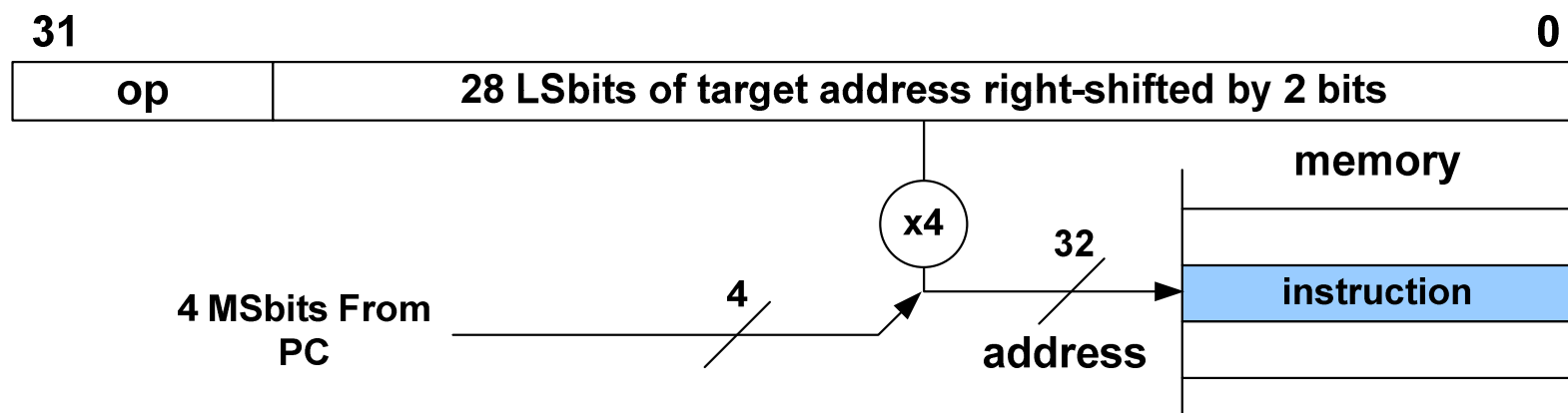
- PC-relative Addressing (endereço relativo ao PC):



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

Modos de endereçamento do MIPS (resumo)

- **Pseudo-direct Addressing (endereço pseudo-direto):**



Exemplos:

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

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

Questões / exercícios

- Qual o formato de codificação de cada uma das seguintes instruções: "**beq/bne**", "**j**", "**jr**"?
- O que é codificado no campo *offset* do código máquina das instruções "**beq/bne**" ?
- A partir do código máquina de uma instrução "**beq/bne**", como é formado o endereço-alvo (*Branch Target Address*)?
- A partir do código máquina de uma instrução "**j**", como é formado o endereço-alvo (*Jump Target Address*)?
- Na instrução "**jr \$ra**", como é obtido o endereço-alvo?
- Qual o endereço mínimo e máximo para onde uma instrução "**j**", residente no endereço de memória **0x5A18F34C**, pode saltar?
- Qual o endereço mínimo e máximo para onde uma instrução "**beq**", residente no endereço de memória **0x5A18F34C**, pode saltar?
- Qual o endereço mínimo e máximo para onde uma instrução "**jr**", residente no endereço de memória **0x5A18F34C** pode saltar?

Questões / exercícios

- Qual a gama de representação da constante nas instruções aritméticas imediatas?
- Qual a gama de representação da constante nas instruções lógicas imediatas?
- Porque razão não existe no ISA do MIPS uma instrução que permita manipular diretamente uma constante de 32 bits?
- Como é que no MIPS se podem manipular constantes de 32 bits?
- Apresente a decomposição em instruções nativas das seguintes instruções virtuais:

```
li      $6, 0x8B47BE0F
xori    $3, $4, 0x12345678
addi    $5, $2, 0xF345AB17
beq     $7, 100, L1
blt     $3, 0x123456, L2
```

Aulas 9 e 10

- Sub-rotinas: chamada e retorno
- Caracterização das sub-rotinas na perspetiva do "chamador" e do "chamado"
- Convenções adotadas na arquitetura MIPS quanto à:
 - passagem de parâmetros para sub-rotinas
 - devolução de valores de sub-rotinas
 - utilização e salvaguarda de registos
- A *stack* - conceito e operações básicas
- Utilização da *stack* na arquitetura MIPS
- Análise de um exemplo.

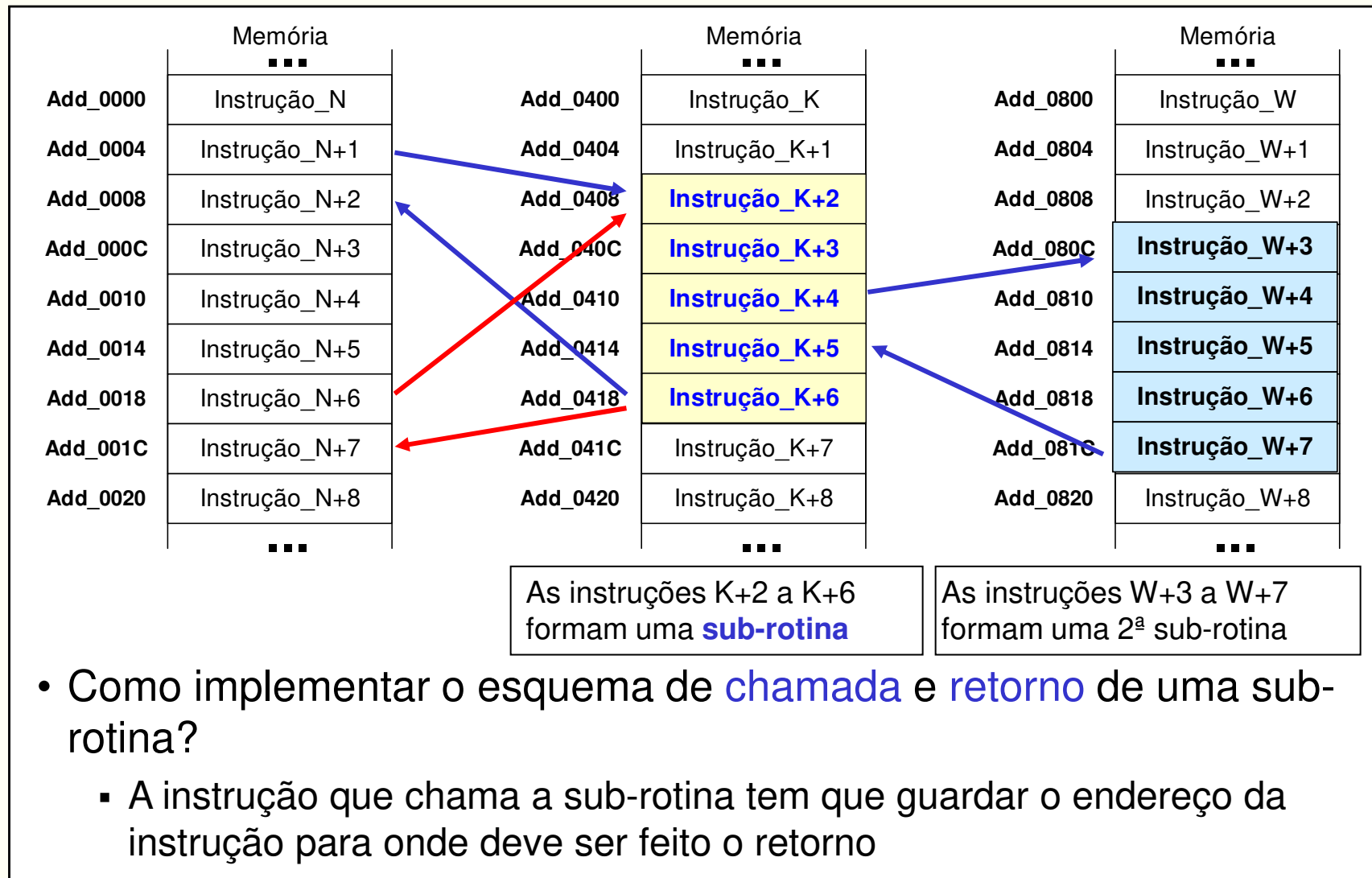
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Porque se usam funções (sub-rotinas)?

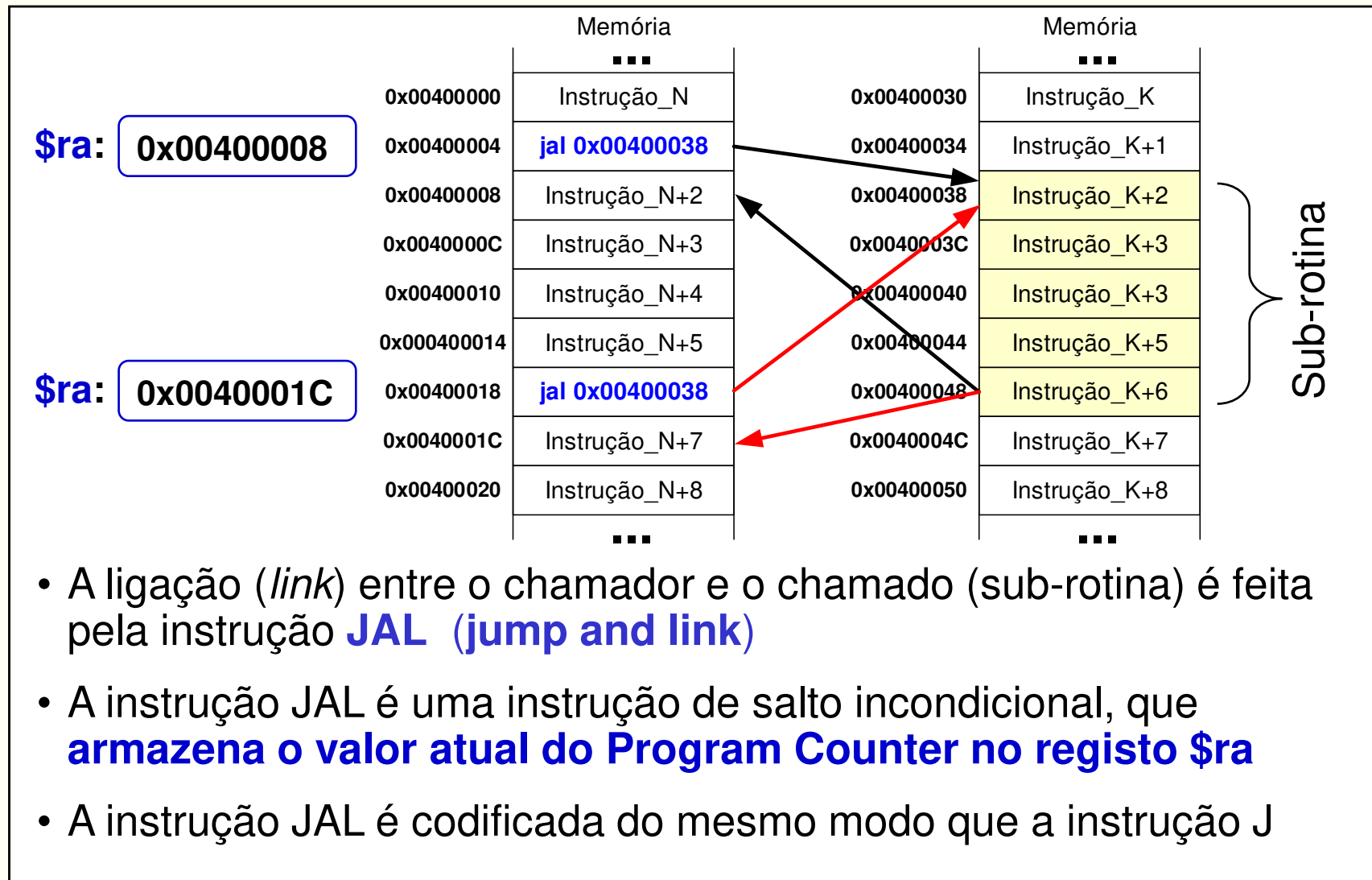
- Há três razões principais que justificam a existência de funções*:
 - A **reutilização no contexto de um determinado programa** - aumento da eficiência na dimensão do código, substituindo a repetição de um mesmo trecho de código por um único trecho evocável de múltiplos pontos do programa
 - A **reutilização 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

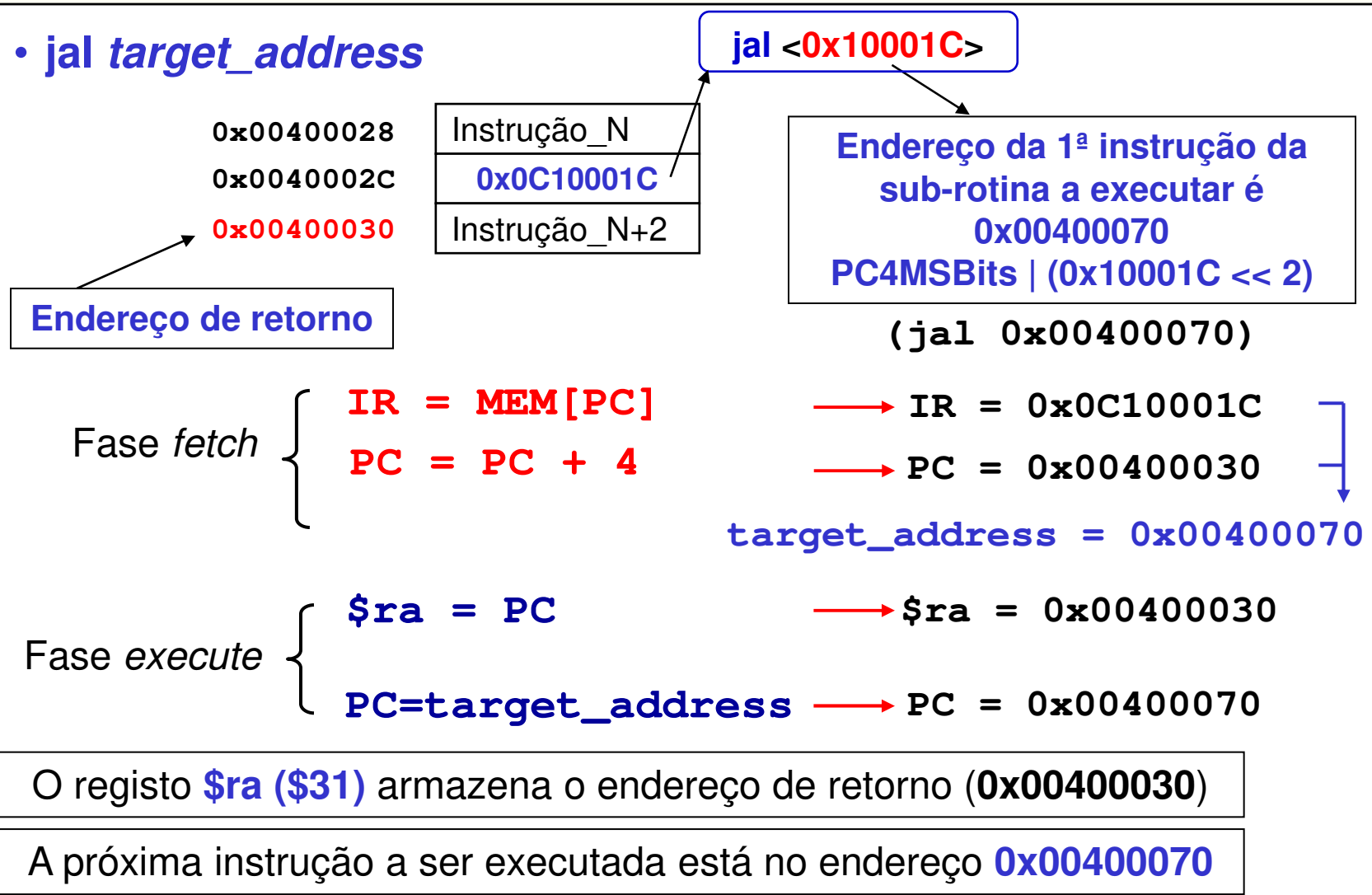


Sub-rotinas no MIPS: a instrução JAL



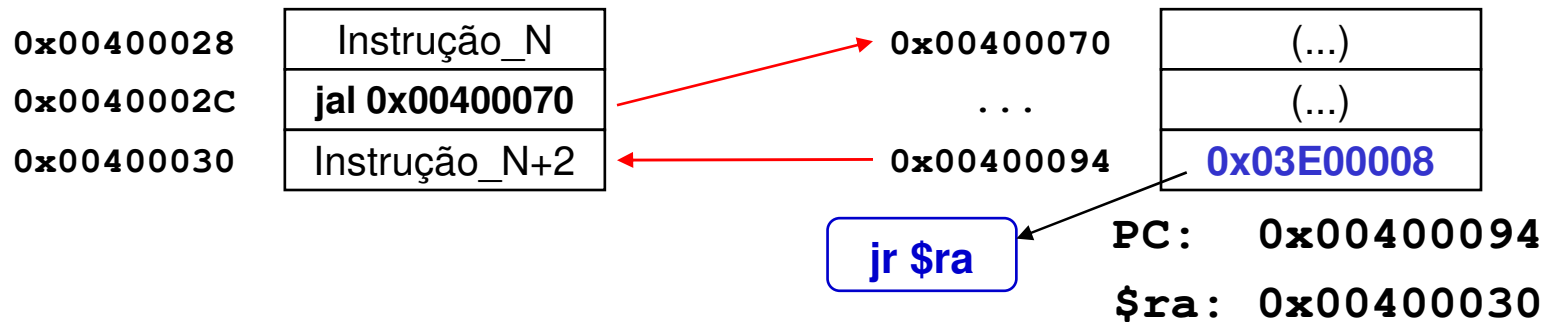
Ciclo de execução da instrução JAL

• *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**")

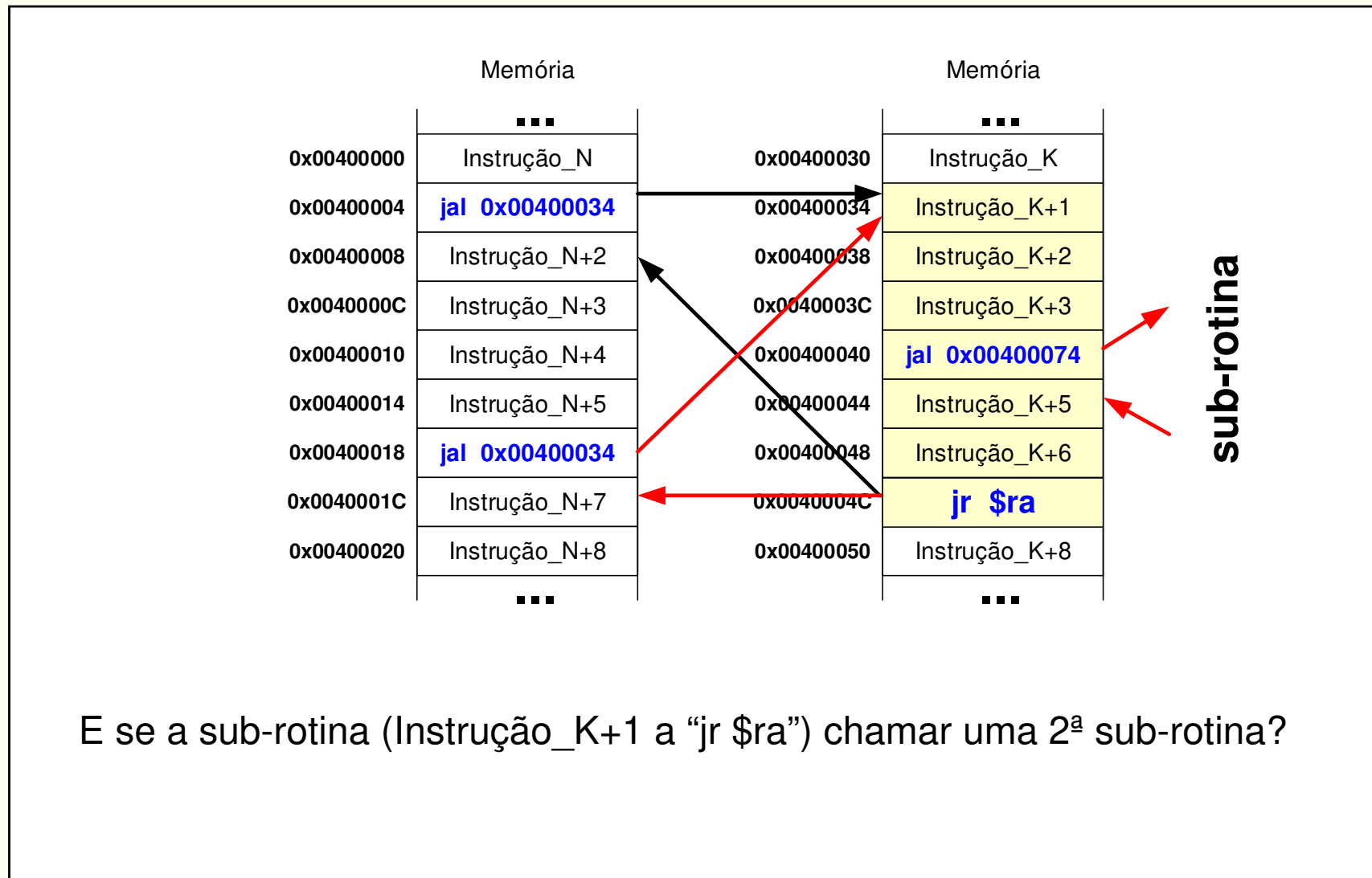


Fase *fetch* { **IR = MEM[PC]** → IR = 0x03E00008
PC = PC + 4 → PC = 0x00400098

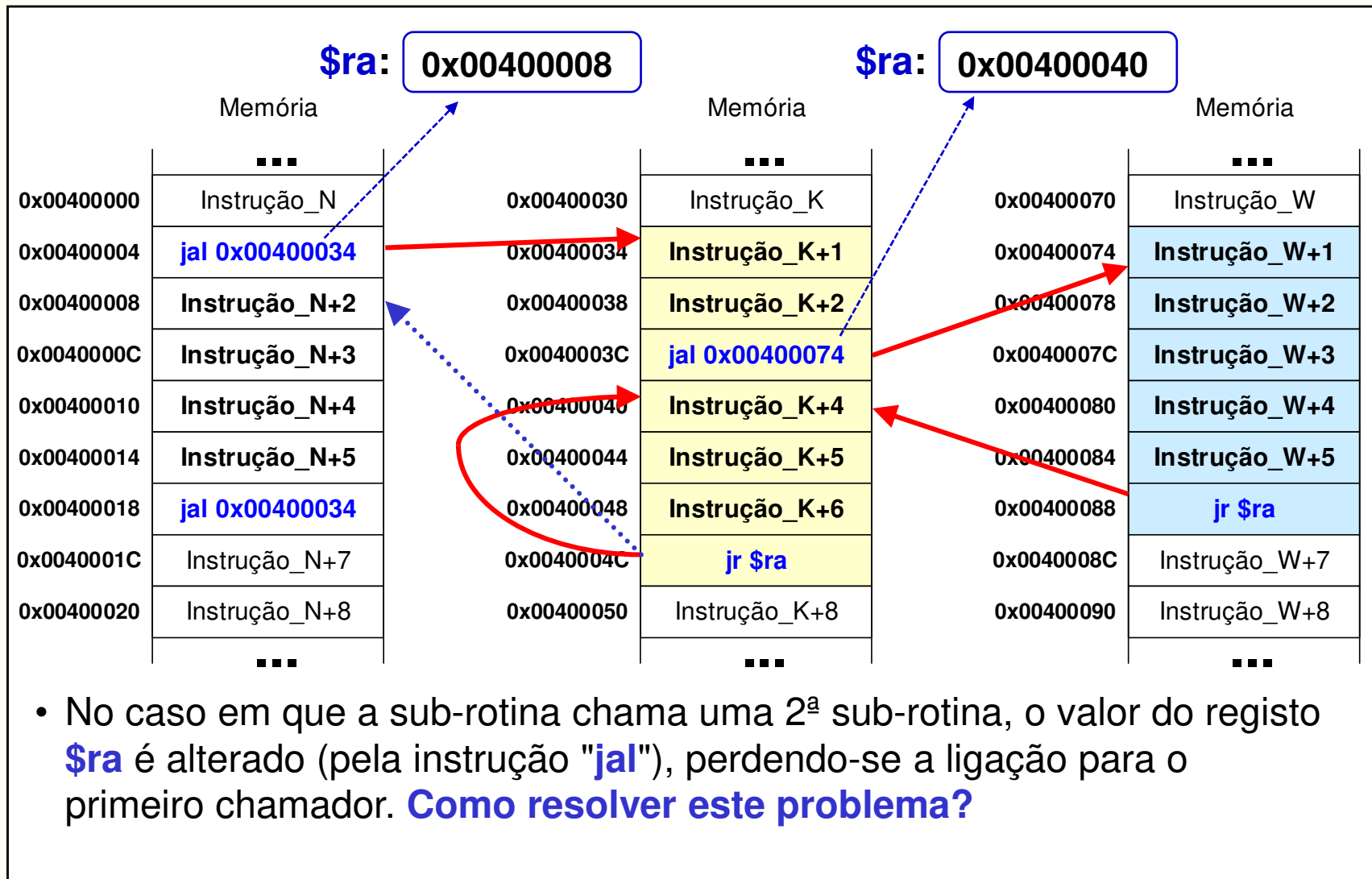
Fase *execute* { **PC = \$ra** → PC = 0x00400030

A próxima instrução a ser executada está no endereço **0x00400030**

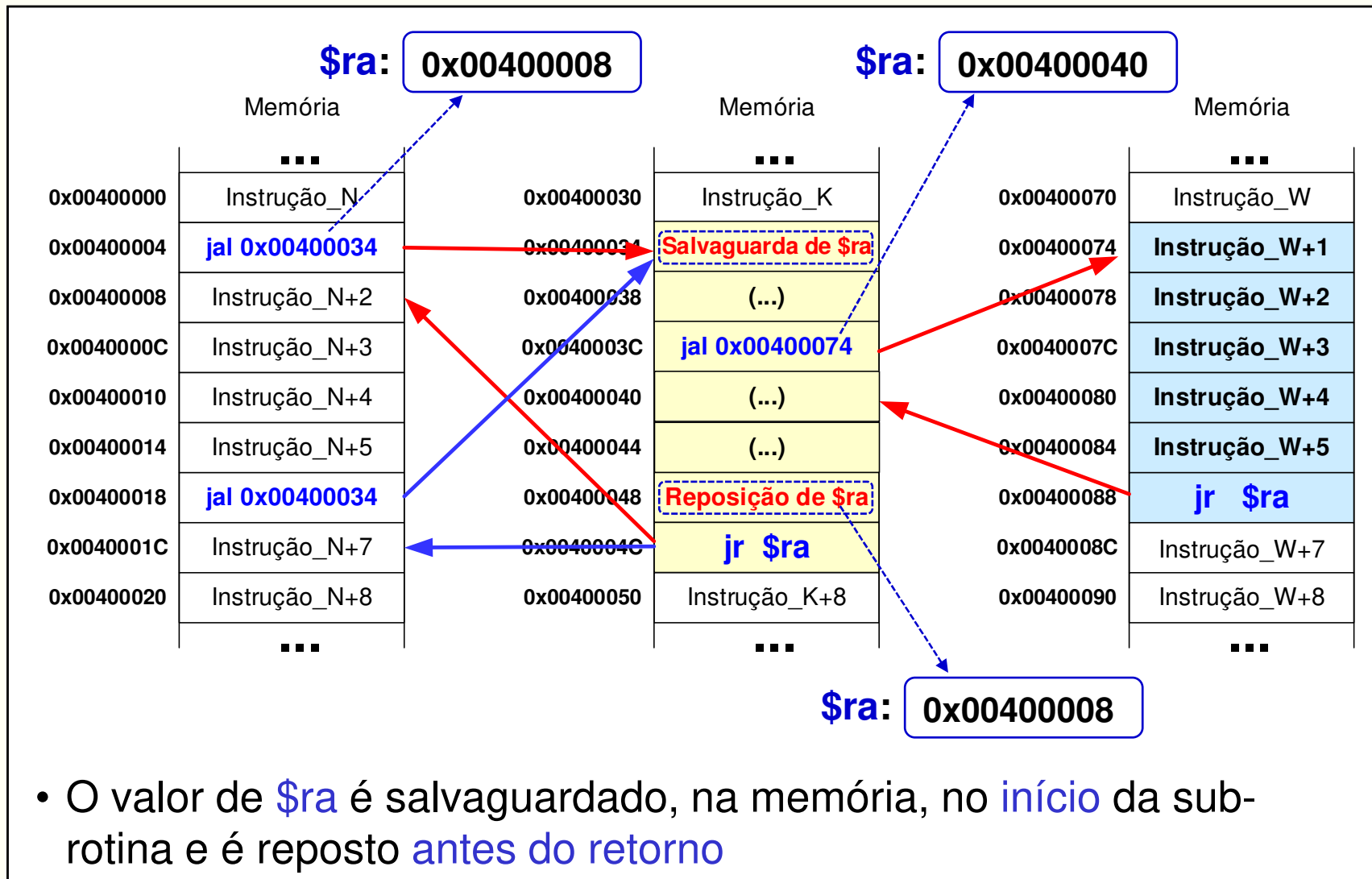
Chamada a uma sub-rotina a partir de outra sub-rotina



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



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



Instruções JAL e JALR

- A instrução "**jal**" é codificada do mesmo modo que a instrução "**j**": formato j em que os 26 bits menos significativos são obtidos dos 28 bits menos significativos do endereço-alvo, deslocados à direita dois bits
- Durante a execução, a obtenção do endereço-alvo é feita do mesmo modo que na instrução "**j**"
- A especificação de um endereço-alvo de 32 bits é possível através da utilização da instrução "**jalr**" (**jump and link register**)
- A instrução "**jalr**" funciona de modo idêntico à instrução "**jal**", exceto na obtenção do endereço-alvo:
 - o endereço da sub-rotina é lido do registo especificado na instrução (endereçamento indireto por registo); ex: **jalr \$t2**
- A instrução "**jalr**" é codificada com o formato R

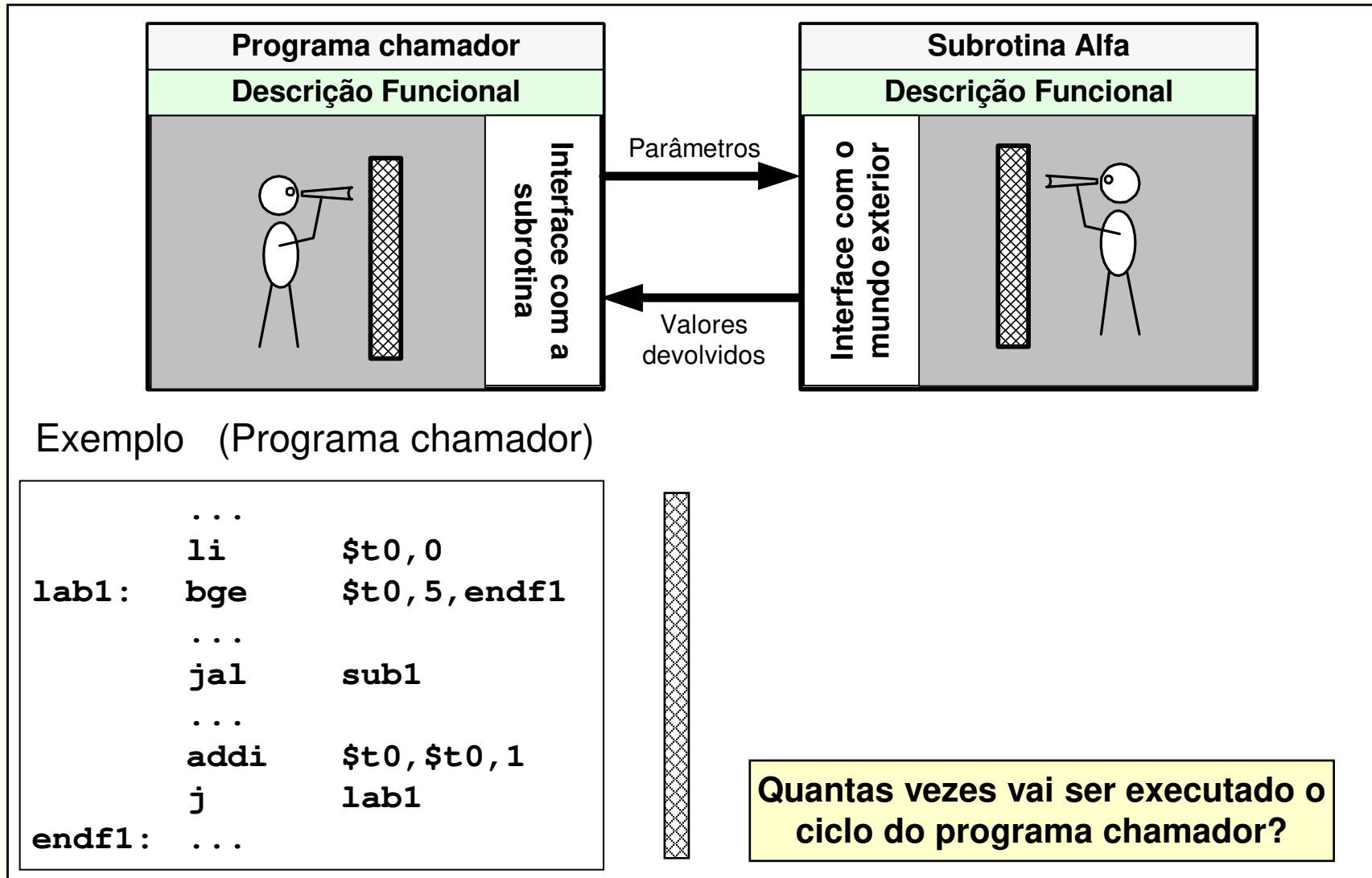
Sub-rotinas

- A **reutilização de sub-rotinas** é essencial em programação, 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 frequentemente agrupadas em **bibliotecas**, a partir das quais podem ser evocadas por qualquer programa
- A utilização de sub-rotinas escritas por outros para serviço dos nossos programas, **não deverá implicar 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

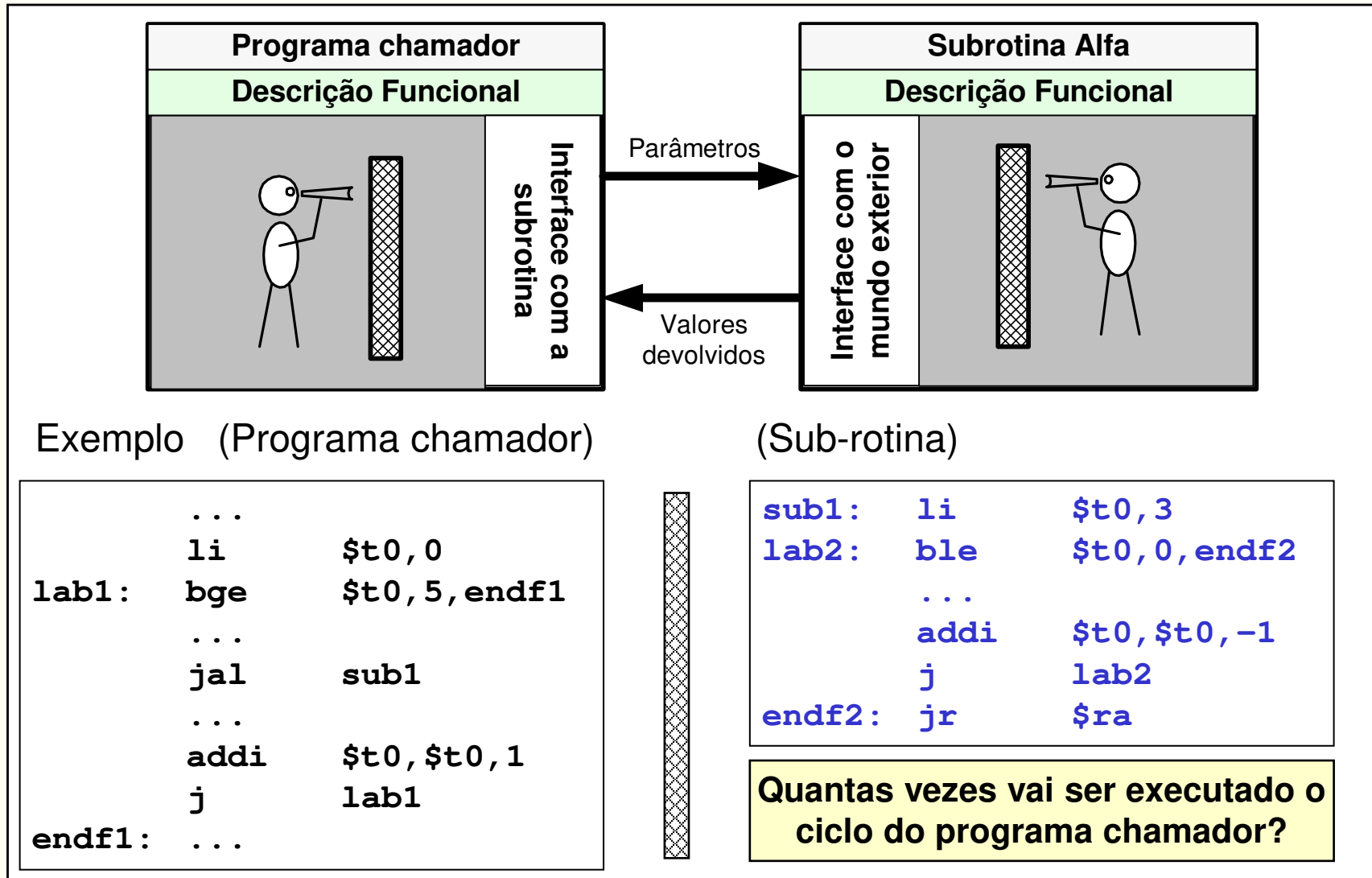
Sub-rotinas

- 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 uma interface que ele próprio pode determinar em função das necessidades
- O facto de uma sub-rotina ser escrita para poder ser reutilizada implica que o programador não conhece antecipadamente as características do programa que a irá chamar
- Torna-se óbvia a necessidade de definir um conjunto de **regras que regulem a relação entre o programa “chamador” e a sub-rotina “chamada”**:
 - definição da interface entre ambos, i.e., quais os parâmetros de entrada e como os passar para a sub-rotina e como devolver resultados ao programa chamador
 - princípios que assegurem uma “sã convivência” entre os dois, de modo a que um não destrua os dados do outro

Sub-rotinas



Sub-rotinas



Regras a definir entre chamador e a sub-rotina chamada

- Ao nível da interface:
 - Como **passar parâmetros** do “chamador” para a sub-rotina, identificar quantos e onde são passados
 - Como **devolver**, para o “chamador”, resultados produzidos pela sub-rotina
- Ao nível das regras de “sã convivência”:
 - Que registos do CPU podem “chamador” e sub-rotina usar, sem que haja alteração indevida de informação por parte da sub-rotina (por exemplo a sub-rotina alterar o conteúdo de um registo que é usado pelo chamador e que tem informação que não pode ser perdida)
 - Como partilhar a memória usada para armazenar dados, sem risco de sobreposição (e consequente perda de informação armazenada)

Convenções do MIPS (passagem e devolução de valores)

- 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 seja superior a quatro, os 4 primeiros são passados nos registos \$ai e os restantes (pela ordem em que são declarados) deverão ser passados na stack*
- 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)**

Exemplo (chamador)

```
int max(int, int);
void main(void)
{
    int val1=19;
    int val2=35;
    int maxVal;
    ...
    maxVal=max(val1, val2);
    print_int10(maxVal);
}
```

parâmetros passados em \$a0 e \$a1

chamada da sub-rotina

valor devolvido em \$v0

```
val1:    $t2
val2:    $t3
maxVal:  $a0
```

```
.text
main:   (...)    #Salvaguarda $ra
        li      $t2, 19
        li      $t3, 35
        ...
        move     $a0, $t2
        move     $a1, $t3
        jal      max
        move     $a0, $v0
        li      $v0, 1
        syscall
        (...)    #Repõe $ra
        jr      $ra
```

- Para escrever o programa “chamador”, não é necessário conhecer os detalhes de implementação da sub-rotina

Exemplo (sub-rotina)

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

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

vmax: \$t0

Em Assembly:

```
max:  move    $t0, $a0
      ble     $a1, $t0, endif
      move    $t0, $a1
endif: move    $v0, $t0
      jr      $ra
```

parâmetros

Valor a devolver

regresso ao chamador

- Para escrever o código da sub-rotina, não é necessário conhecer os detalhes de implementação do “chamador”
- 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 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
 - Essa responsabilidade pode ser atribuída ao chamador ou à sub-rotina (ou aos dois)

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 chamar a sub-rotina
 - Cabe-lhe também a tarefa de repor posteriormente o seu valor
 - No limite, é admissível que o “chamador” salvaguarde apenas os registos de cujo conteúdo venha a precisar mais tarde
- Estratégia “**callee-saved**”
 - Entrega-se à sub-rotina a responsabilidade pela prévia salvaguarda dos registos que vai usar
 - Assegura, igualmente, a tarefa de repor o seu valor imediatamente antes de regressar ao programa “chamador”

Convenção para salvaguarda de registos no MIPS

- Os registos **\$t0** a **\$t9**, **\$v0** e **\$v1**, e **\$a0** a **\$a3** podem ser livremente utilizados e alterados pelas sub-rotinas
- Os registos **\$s0** a **\$s7** não podem, **na perspetiva do chamador**, ser alterados pelas sub-rotinas
 - Se uma dada sub-rotina precisar de usar qualquer um dos registos **\$s0** a **\$s7** compete a essa sub-rotina **salvaguardar previamente o seu conteúdo**, repondo-o imediatamente antes de terminar
 - Ou seja, é seguro para o programa chamador usar um registo **\$sn** para armazenar um valor que vai necessitar após a chamada à sub-rotina, uma vez que tem a garantia que esta não o modifica

Considerações práticas sobre a utilização da convenção

- **sub-rotinas terminais** (sub-rotinas folha, i.e., que não chamam qualquer sub-rotina)
 - Só devem utilizar registos que não têm a responsabilidade de salvar guardar (**\$t0..\$t9**, **\$v0..\$v1** e **\$a0..\$a3**)
- **sub-rotinas intermédias** (sub-rotinas que chamam outras sub-rotinas)
 - Devem utilizar os registos **\$s0..\$s7** para o armazenamento de valores que pretendam preservar durante a chamada à sub-rotina seguinte
 - A utilização de qualquer um dos registos **\$s0** a **\$s7** 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 13 e 14 pode facilmente ser resolvido se a convenção de salvaguarda de registos for aplicada

O código da sub-rotina é desconhecido do programador do “chamador” e vice-versa

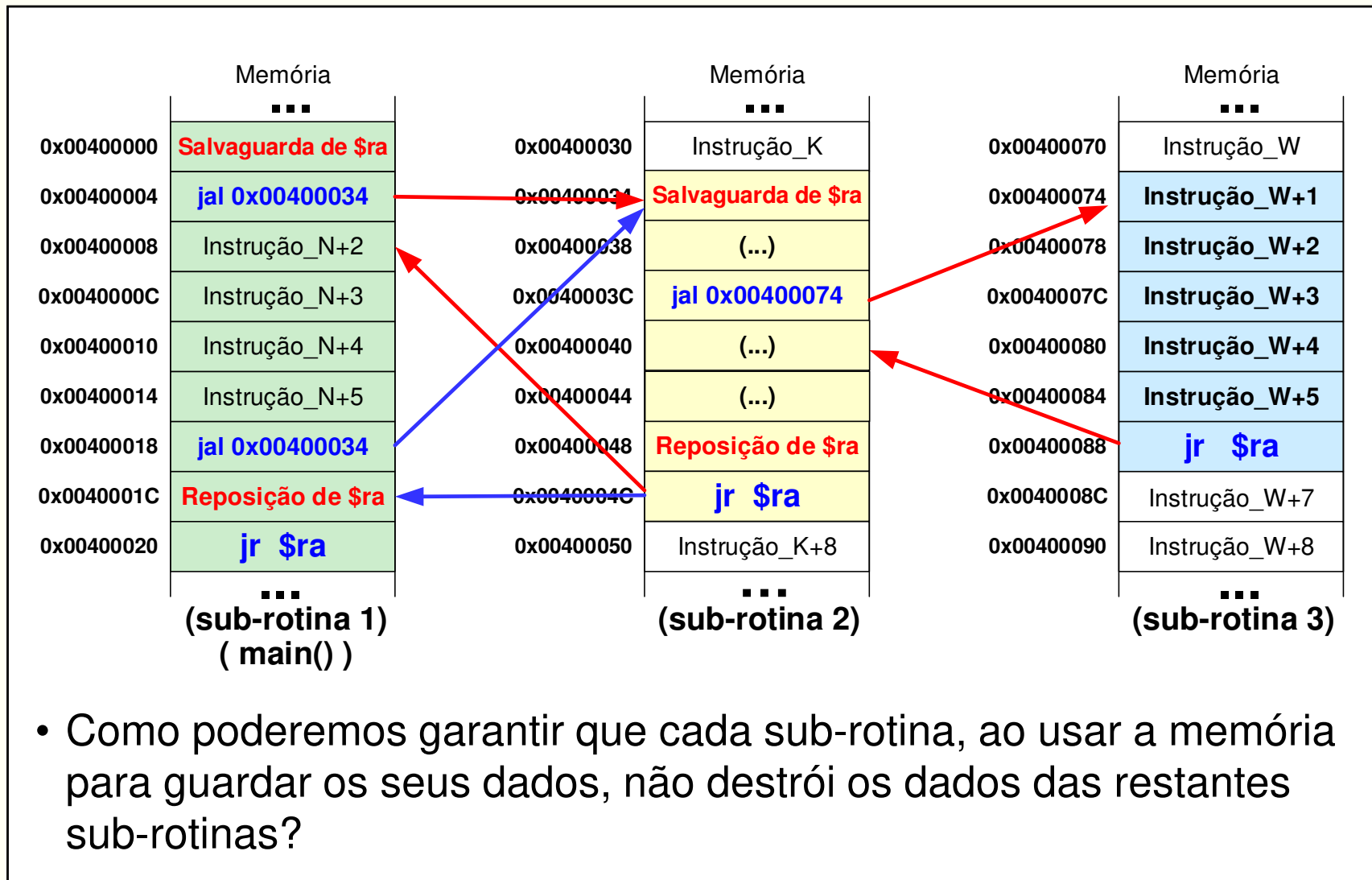
```
(...) # Salv. $s0
...
li      $s0, 0
lab1:   bge    $s0, 5, endf1
...
jal     sub1
...
addi    $s0, $s0, 1
j       lab1
endf1:  ...
(...) # Repoe $s0
```

```
sub1:   li      $t0, 3
lab2:   ble     $t0, 0, endf2
...
        addi    $t0, $t0, -1
        j       lab2
endf2:  jr      $ra
```

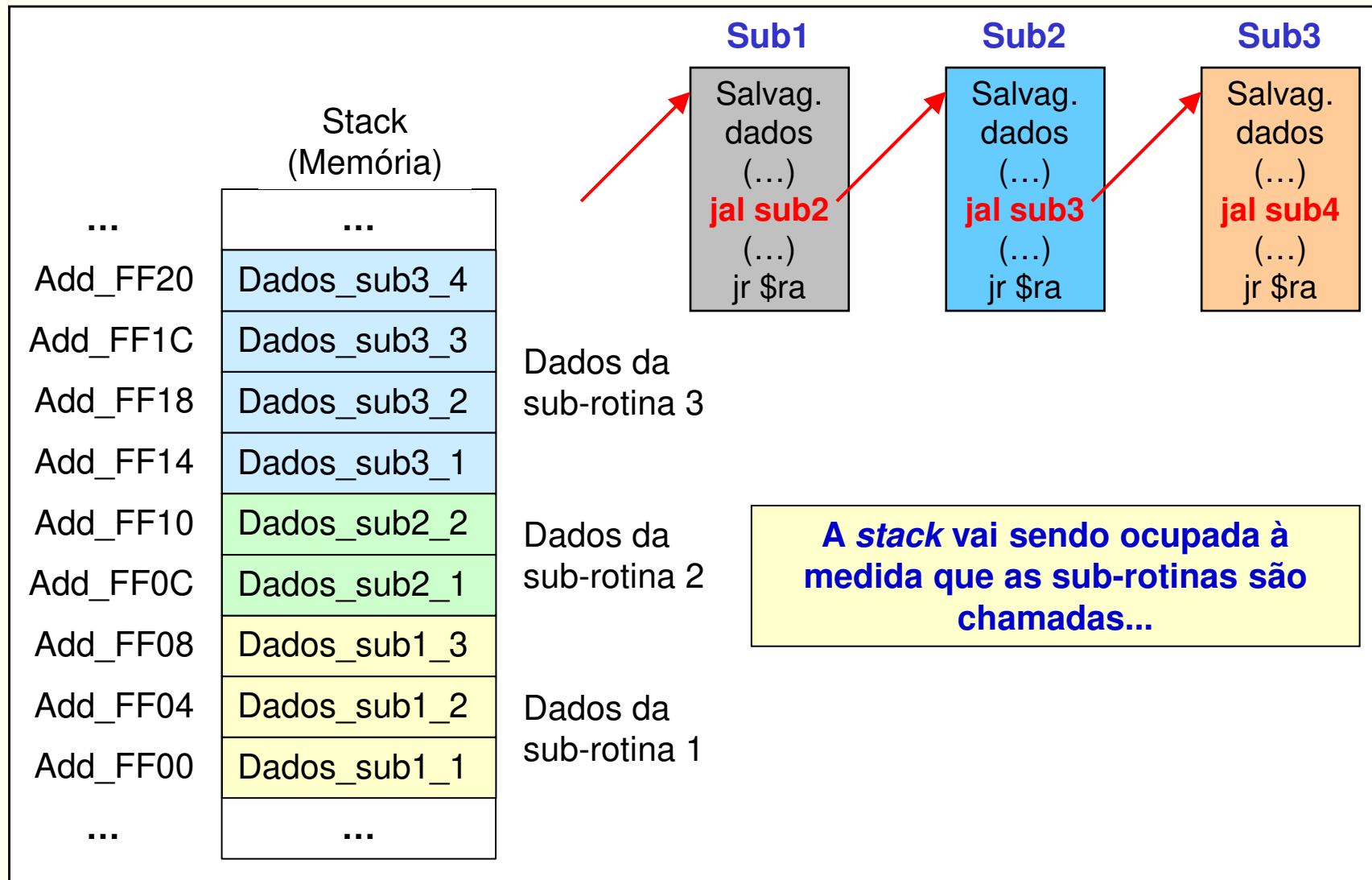
Quantas vezes vai ser executado o ciclo do programa chamador?

- A variável de controlo do ciclo do chamador deverá residir num registo **\$sn** (por exemplo no \$s0) – registo que, **garantidamente**, a sub-rotina não vai alterar

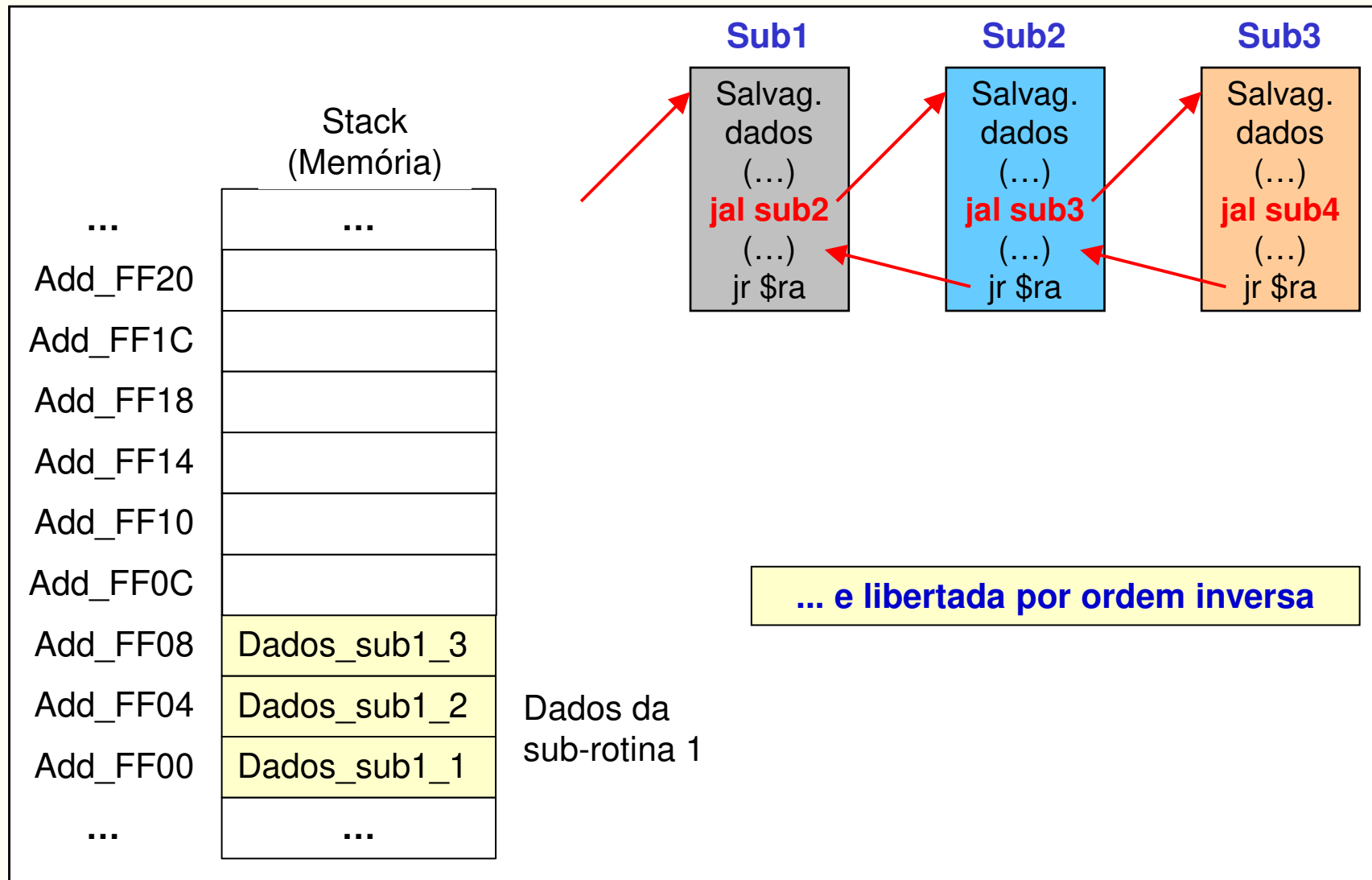
Armazenamento temporário de informação



Stack: espaço de armazenamento temporário



Stack: espaço de armazenamento temporário

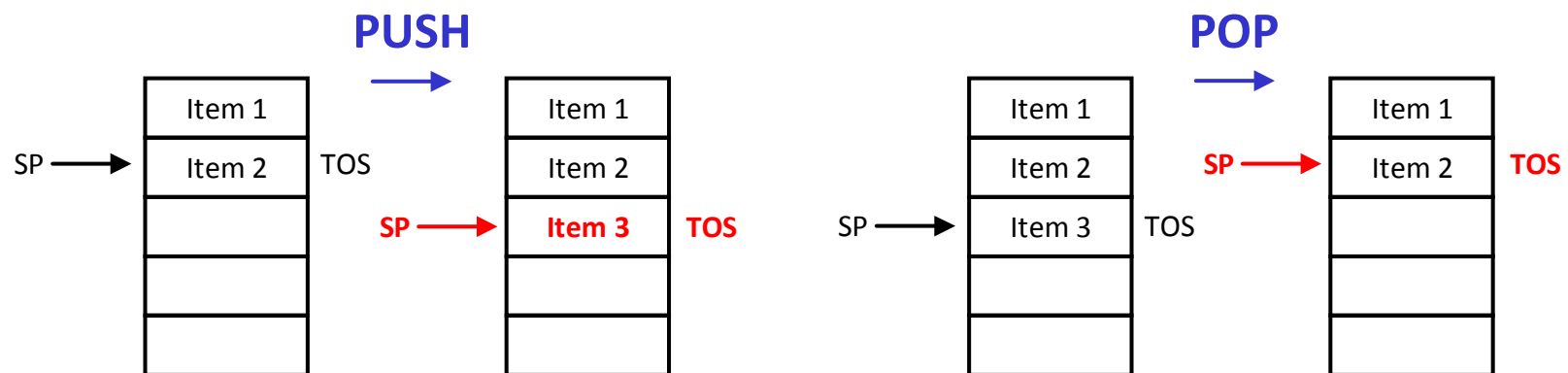


Stack: espaço de armazenamento temporário

- 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 **stack** ("pilha")
- As *stacks* são de tal forma importantes que muitas arquiteturas suportam diretamente instruções específicas para a sua manipulação (por exemplo a arquitetura Intel 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*

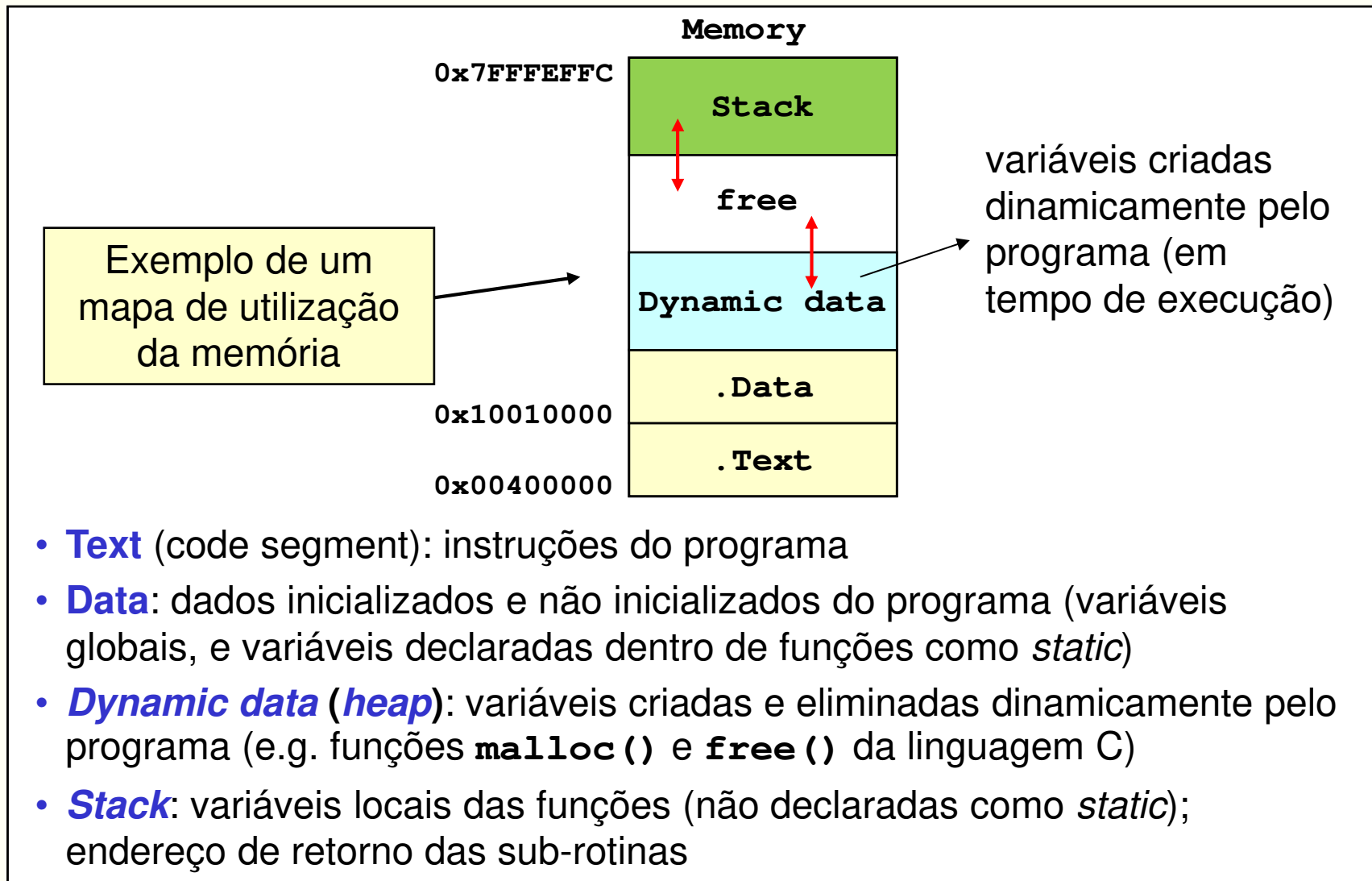
- Estas operações têm associado um registo designado por **Stack Pointer (SP)**
- O registo **Stack Pointer** mantém, de forma permanente, o **endereço do topo da stack (TOS - top of stack)** e aponta sempre **para o último endereço ocupado**
 - Numa operação de **PUSH** é necessário pré-atualizar o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



Atualização do *stack pointer*

- A atualização do ***stack pointer***, numa operação de *push* (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
- A 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

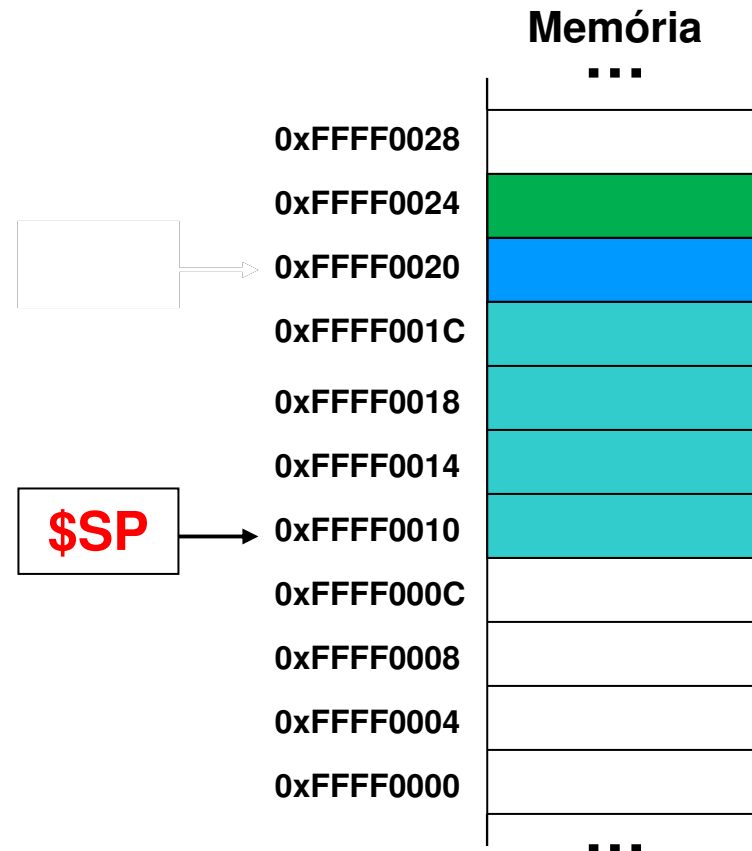
1. O registo **\$sp** (*stack pointer*) contém o endereço da **última posição ocupada** da *stack*

\$sp = \$29

\$SP: **0xFFFF0010**

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

3. A *stack* **está organizada em words de 32 bits**



Regras de utilização da *stack* na arquitetura MIPS

- Exemplo (salvaguarda de 4 registos)

sb1: **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*

jr \$ra # Retorna

\$SP

\$SP

Memória

...

0xFFFF0028

0xFFFF0024

0xFFFF0020

0xFFFF001C

0xFFFF0018

0xFFFF0014

0xFFFF0010

0xFFFF000C

0xFFFF0008

0xFFFF0004

0xFFFF0000

...

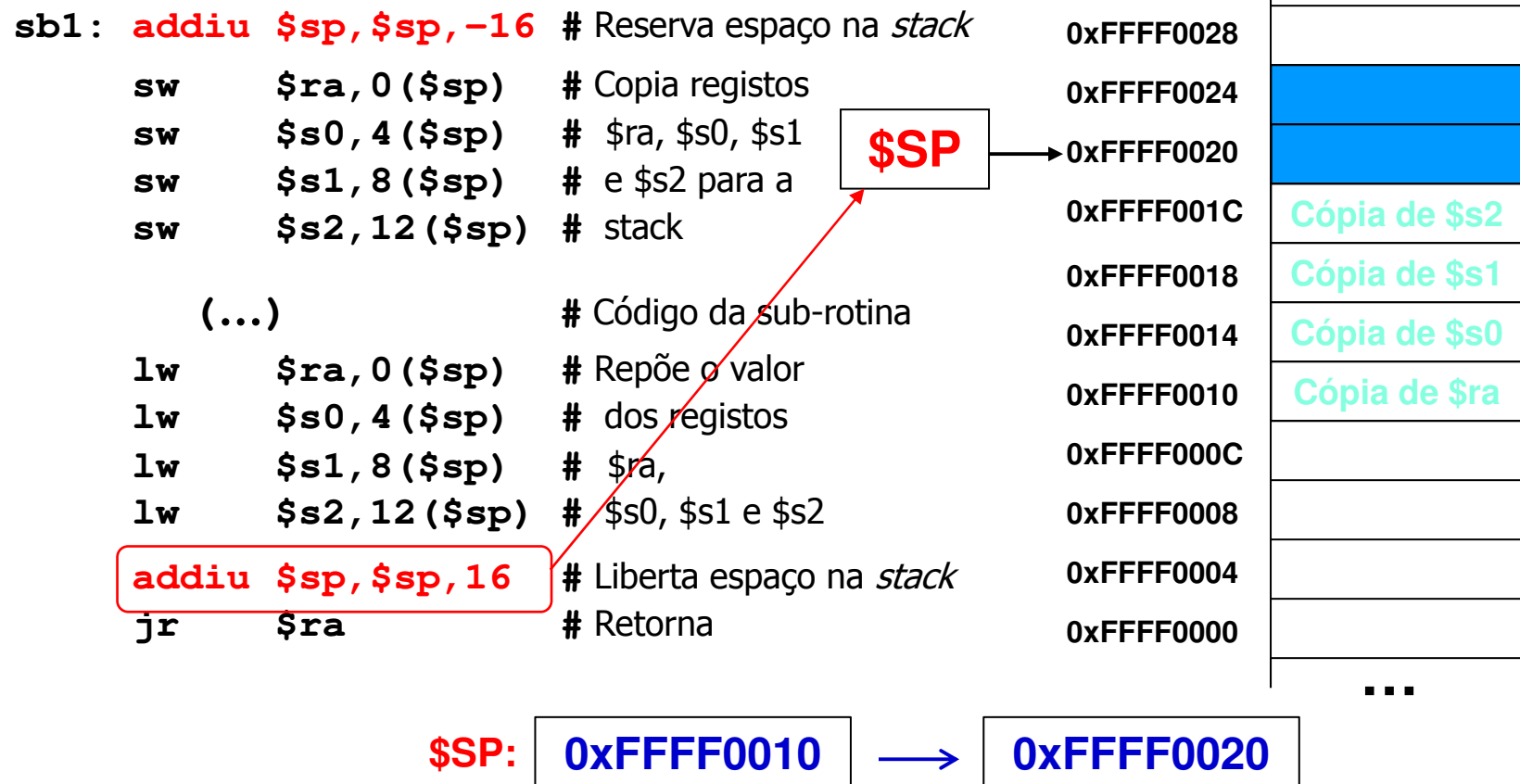
\$SP: 0xFFFF0020



0xFFFF0010

Regras de utilização da *stack* na arquitetura MIPS

- Exemplo (salvaguarda de 4 registos)



Regras de utilização da *stack* na arquitetura MIPS

- Exemplo

Prólogo

```
sb1: 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
```

\$SP

(...)

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
      jr    $ra          # Retorna
```

Epílogo

Memória

0xFFFF0028

0xFFFF0024

0xFFFF0020

0xFFFF001C

0xFFFF0018

0xFFFF0014

0xFFFF0010

0xFFFF000C

0xFFFF0008

0xFFFF0004

0xFFFF0000

Cópia de \$s2

Cópia de \$s1

Cópia de \$s0

Cópia de \$ra

Análise de um exemplo completo

Considere-se o seguinte código C:

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

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

```
    static int temp[100]; //reside em memória
```

```
    int result;
```

```
    ...                // código de inicialização do array
```

```
    result = soma(temp, 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

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

Análise de um exemplo completo

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

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

- A função **main()** é uma função intermédia (chama a função **soma()**); **registo \$ra** tem que ser salvaguardado
- A variável "**result**" não tem atribuído qualquer valor que seja necessário depois da chamada à função **soma()**; **deve residir num registo \$tn, \$vn ou \$an** (pode ser usado **\$t0**)

Código correspondente em Assembly do MIPS

```
# result: $t0
#
```

```
        .data
temp:    .space 400                # Reserva de espaço p/ o array
                                           # (100 words => 400 bytes)

        .eqv    print_int10, 1    #
        .text
        .globl  main
main:    addiu    $sp, $sp, -4      # Reserva espaço na stack
        sw       $ra, 0($sp)      # Salva o registo $ra
        la       $a0, temp        # inicialização dos registos
        li       $a1, 100         # que vão passar os parâmetros
        jal      soma             # soma(temp, 100)
        move     $t0, $v0         # result = soma(temp, 100)
        move     $a0, $t0         #
        li       $v0, print_int10 #
        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 temp[100];
    int result;
    result = soma(temp, 100);
    print_int(result);
}
```

Código da função soma()

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

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

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

A mesma função usando ponteiros:

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


Código da função soma() usando ponteiros:

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

- A função **soma ()** é uma **função terminal**:
 - não é necessário salvar **\$ra**
 - só devem ser usados registros **\$t_n**, **\$v_n** ou **\$a_n**

Código correspondente em *Assembly* do MIPS

- Versão com ponteiros

```
# res:    $v0
# p:      $t1
# array:  $a0
# n:      $a1
#
```

```
soma:  li      $v0, 0           # res = 0;
       move    $t1, $a0        # p = array;
       sll     $a1, $a1, 2      # n *= 4;
       addu    $a0, $a0, $a1    # $a0 = array + n;
for:   bgeu    $t1, $a0, endf   # while(p < &(array[n])){
       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 n)
{
    int res = 0;
    int *p = array;
    for (; p < &(array[n]); p++)
        res += (*p);
    return res;
}
```

A sub-rotina não chama nenhuma outra e não são usados registos **\$sn**, pelo que não é necessário salvar qualquer registo

Código de uma função para cálculo da média

```
int media (int *array, int n)
{
    int res;

    res = soma(array, n);
    return res / n;
}
```

- A função `media()` é uma **função intermédia**:
 - é necessário **salvar \$ra**
- O valor da variável "`res`" só é definido após a chamada à função; deve residir num registo de utilização livre, por exemplo **\$t0**
- O número de elementos do `array` "`n`" (**\$a1**), é necessário após a chamada à função `soma()`; **o registo \$a1 tem que ser copiado para um registo \$sn** (por exemplo **\$s1**)

Exemplo – função para cálculo da média

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

chama função soma()

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

```
# res: $t0, array: $a0, n: $a1 -> $s1
```

```
media: addiu $sp,$sp,-8      # Reserva espaço na stack
       sw    $ra,0($sp)     # salvaguarda $ra e $s1
       sw    $s1,4($sp)     # guarda valor $s1 antes de o usar
       move  $s1,$a1        # "n" é necessário depois
                               # da chamada à função soma
       jal   soma           # soma(array,n);
       move  $t0,$v0        # res = soma(array,n);
       div   $v0,$t0,$s1    # res/n
       lw    $ra,0($sp)     # recupera valor de $ra
       lw    $s1,4($sp)     # e $s1
       addiu $sp,$sp,8      # Liberta espaço na stack
       jr    $ra           # retorna
```

Questões

- O que é uma sub-rotina? Qual a instrução do MIPS usada para saltar para uma sub-rotina? Porque razão não pode ser usada a instrução "j"?
- Quais as operações realizadas, e relativa sequência, na execução de uma instrução "jal"? Qual o nome virtual e o número do registo associado à execução dessa instrução?
- No caso de uma sub-rotina ser simultaneamente chamada e chamadora (sub-rotina intermédia) que operações é obrigatório realizar nessa sub-rotina?
- Qual a instrução usada para retornar de uma sub-rotina? Que operação fundamental é realizada na execução dessa instrução?
- De acordo com a convenção de utilização de registos no MIPS:
 - Que registos são usados para passar parâmetros e para devolver resultados de uma sub-rotina?
 - Quais os registos que uma sub-rotina pode livremente usar e alterar sem necessidade de prévia salvaguarda?
 - Quais os registos que uma sub-rotina tem de preservar? Quais os registos que uma sub-rotina chamadora tem a garantia que a sub-rotina chamada não altera?
 - Em que situação devem ser usados registos `$sn`? Em que situação devem ser usados os restantes: `$tn`, `$an` e `$vn`?

Questões

- O que é a *stack*? Qual a utilidade do *stack pointer*?
- Como funcionam as operações de *push* e *pop*?
- Porque razão a *stack* cresce tipicamente no sentido dos endereços mais baixos?
- Quais as regras para a implementação em software de uma *stack* no MIPS? Qual o registo usado como *stack pointer*?
- De acordo com a convenção de utilização de registos do MIPS:
 - Que registos devem preferencialmente ser usados numa sub-rotina intermédia, para armazenar variáveis cujo tempo de vida inclui a chamada de sub-rotinas? Que cuidados se deve ter na utilização desses registos?
 - Que registos devem preferencialmente ser usados numa sub-rotina intermédia, para armazenar variáveis cujo tempo de vida não inclui a chamada de sub-rotinas?
 - Que registos devem preferencialmente ser usados numa sub-rotina terminal para armazenar variáveis?
- Para a função com o protótipo seguinte, indique, para cada um dos parâmetros de entrada e para o valor devolvido, qual o registo do MIPS usado para a passagem dos respetivos valores:

char fun(int a, unsigned char b, char *c, int *d)

Exercício

- Traduza para *assembly* do MIPS a seguinte função **fun1()**, aplicando a convenção de passagem de parâmetros e salvaguarda de registos:

```
char *fun2(char *, char);

char *fun1(int n, char *a1, char *a2)
{
    int j = 0;
    char *p = a1;

    do
    {
        if((j % 2) == 0)
            fun2(a1++, *a2++);
    } while(++j < n);

    *a1 = '\0';
    return p;
}
```

Aula 11

- Representação de números inteiros com sinal: complemento para dois. Exemplos de operações aritméticas
- *Overflow* e mecanismos para a sua deteção
- Construção de uma ALU de 32 bits
- Multiplicação de inteiros no MIPS
- Divisão de inteiros no MIPS. Divisão de inteiros com sinal

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Representação de inteiros

- Sendo um computador um sistema digital binário, a representação de inteiros faz-se sempre em base 2 (símbolos 0 e 1).
- **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:

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

Representação de inteiros

- Os circuitos que realizam operações aritméticas estão igualmente limitados a um número finito de bits, geralmente igual à dimensão dos registos internos do CPU
- Os circuitos aritméticos operam assim em aritmética modular, ou seja em **mod (2^n)** em que "**n**" é o número de bits da 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 uma ALU de 8 bits, por exemplo, o resultado da soma dos números **11001011** e **00110111** seria:

$$11001011 + 00110111 = \text{1} \boxed{00000010}$$

Diagram illustrating the addition of two 8-bit numbers:

The sum of **11001011** and **00110111** is **1** followed by **00000010**.

The **1** is labeled **carry** (indicated by an arrow).

The **00000010** is labeled **resultado com 8 bits** (indicated by an arrow).

- No caso em que os operandos são do tipo **unsigned**, o bit **carry**, se igual a '1', 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**, por si só, não tem qualquer significado, e não faz parte do resultado

Representação em complemento para dois

- O método usado em sistemas computacionais para a codificação de quantidades inteiras com sinal (*signed*) é "complemento para dois"
- **Definição:** Se K é um número positivo, então K^* é o seu complemento para 2 (complemento verdadeiro) e é dado por:

$$K^* = 2^n - K$$

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

- **Exemplo:** determinar a representação de -5, com 4 bits

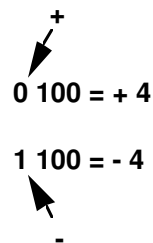
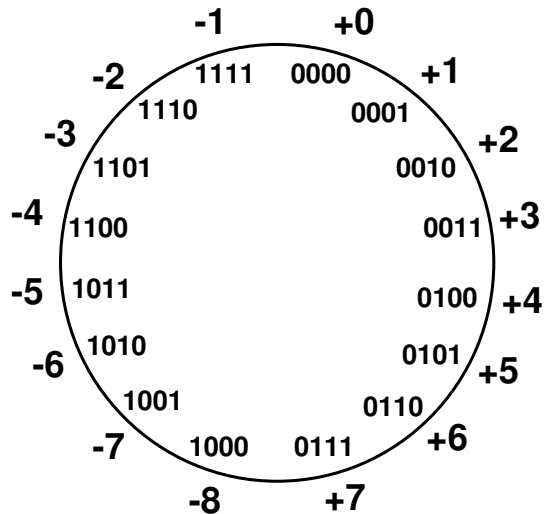
$$N = 5_{10} = 0101_2$$

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

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

- **Método prático:** inverter todos os bits do valor original e somar 1 (0101 \Rightarrow 1010; 1010 + 1 = 1011)
 - Este método é reversível: $C_1(1011) = 0100$; $0100 + 1 = 0101$

Representação em complemento para dois



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 realizada através de uma operação de soma com o complemento para 2 do 2.º operando: **$(a-b) = (a+(-b))$**
- Uma quantidade de N bits codificada em complemento para 2 pode ser representada pelo seguinte polinómio:

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

Representação em complemento para dois

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

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

Onde o bit indicador de sinal (a_{N-1}) é multiplicado por -2^{N-1} e os restantes pela versão positiva do respetivo peso

- **Exemplo:** Qual o valor representado em base 10 pela quantidade 10100101_2 , supondo uma representação em complemento para 2 com 8 bits?

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

- **R2:** O valor é negativo, calcular o módulo (simétrico de 10100101): $01011010 + 1 = 01011011_2 = 5B_{16} = 91_{10}$
 o módulo da quantidade é 91; logo o valor representado é -91_{10}

Representação em complemento para dois

- Exemplos de operações, com 4 bits

$$\begin{array}{rcl} (4 + 3) & 4 & 0100 \\ & + 3 & 0011 \\ \hline & 7 & 0111 \end{array}$$

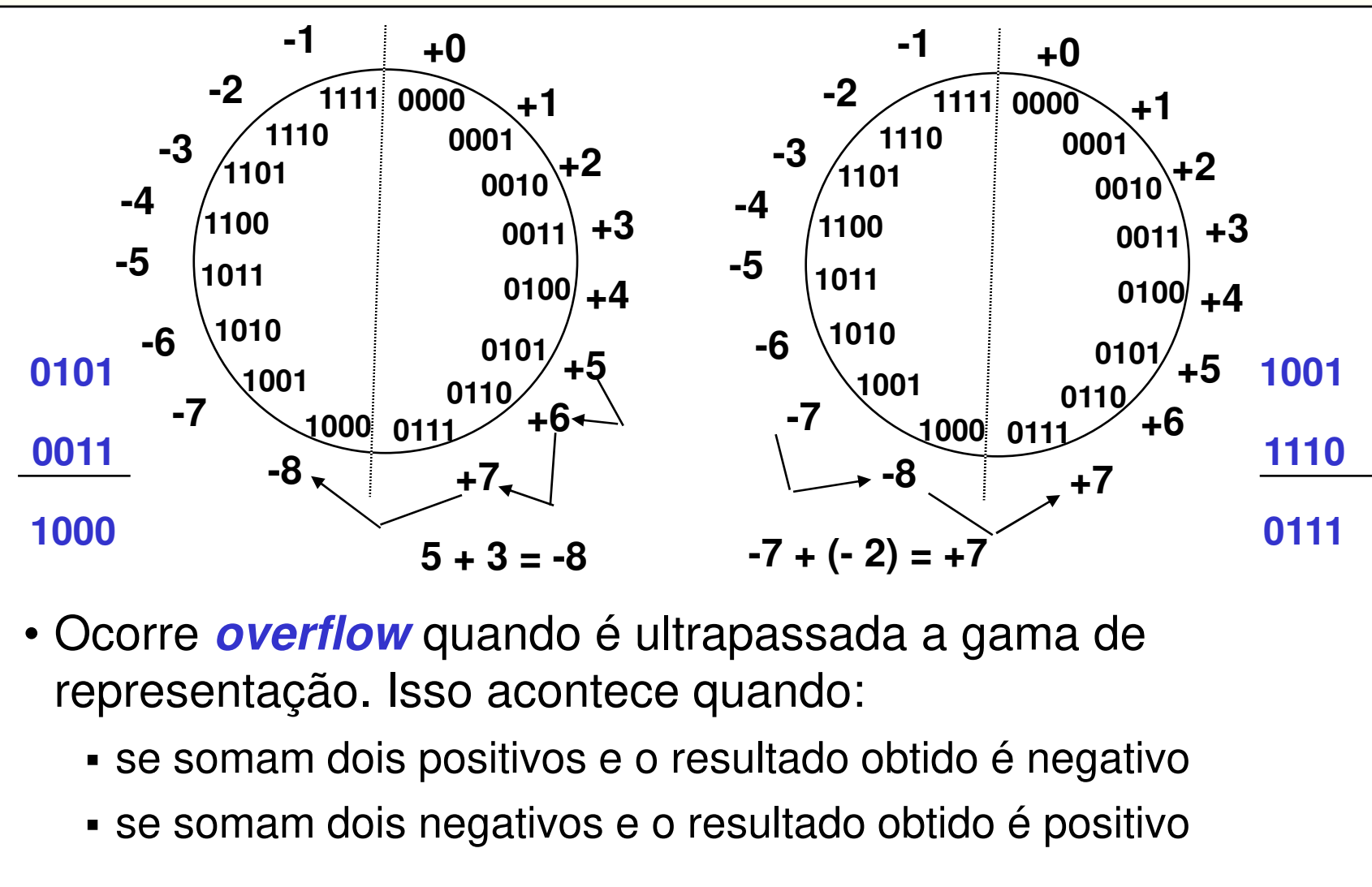
$$\begin{array}{rcl} (-4 - 3) & -4 & 1100 \\ & + (-3) & 1101 \\ \hline & -7 & 11001 \end{array}$$

$$\begin{array}{rcl} (4 - 3) & 4 & 0100 \\ & + (-3) & 1101 \\ \hline & 1 & 10001 \end{array}$$

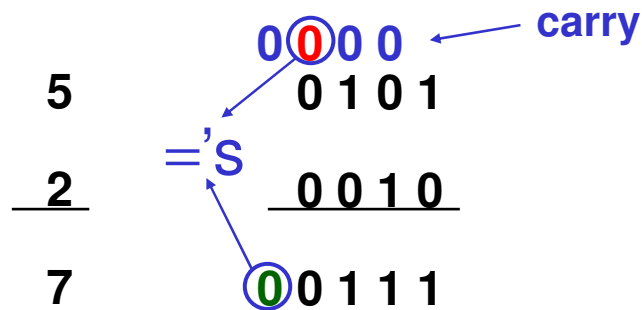
$$\begin{array}{rcl} (-4 + 3) & -4 & 1100 \\ & + 3 & 0011 \\ \hline & -1 & 1111 \end{array}$$

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

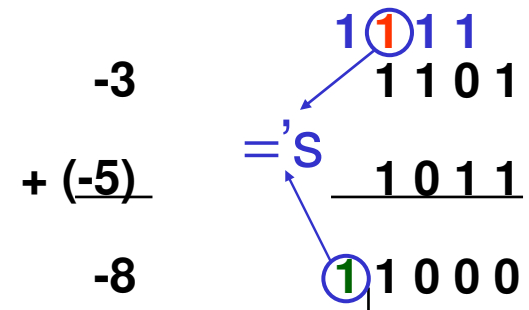
Overflow em complemento para 2



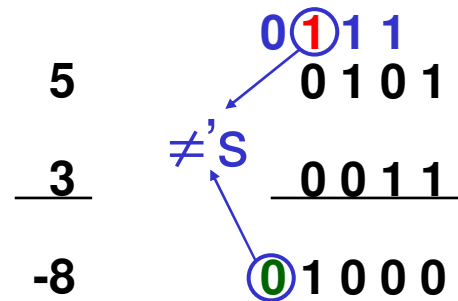
Overflow em complemento para 2



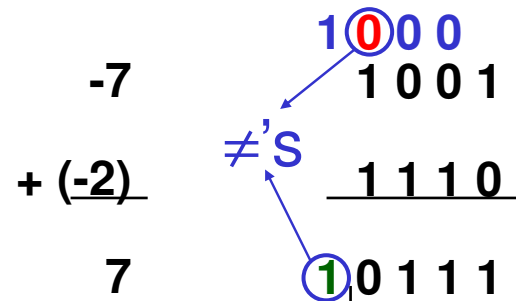
Sem overflow



Sem overflow



Overflow



Overflow

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

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

Overflow em operações aritméticas

- Operandos interpretados em complemento para 2 (i.e. **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*, ($R = A + B$):

- $OVF = R_{n-1} \cdot \overline{A_{n-1}} \cdot \overline{B_{n-1}} + \overline{R_{n-1}} \cdot A_{n-1} \cdot B_{n-1}$

- Operandos interpretados **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*

- O MIPS apenas deteta *overflow* nas operações de adição com sinal (ADD, SUB, ADDI) e, quando isso acontece, gera uma exceção. ADDU, SUBU e ADDIU não detetam *overflow*

Construção de uma ALU de 32 bits

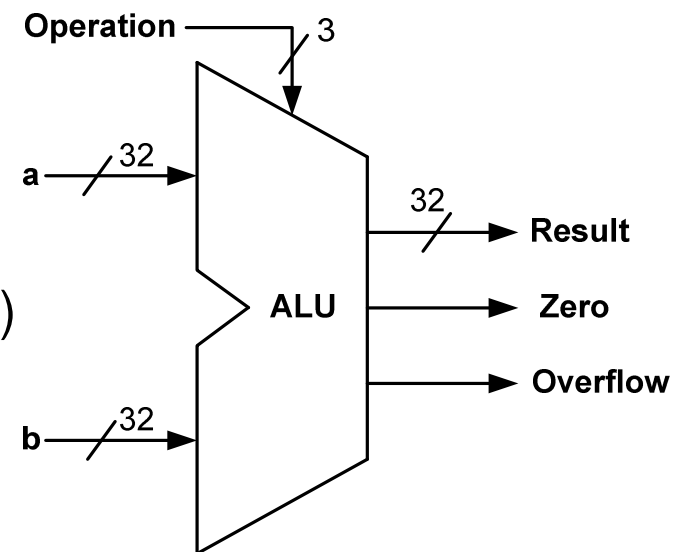
- A ALU deverá realizar as operações:

- ADD, SUB
- AND, OR
- SLT (set if less than)

- Deverá ainda:

- Detetar e sinalizar *overflow* (operandos em complemento para 2)
- 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 – 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;
```

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

```
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); -- simétrico de b (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)
```

```
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 <= X"00000000";
```

```
      end if;
```

```
    when others =>
```

```
      s_res <= (others => '-');
```

```
  end case;
```

```
end process;
```

```
end Behavioral;
```

Construção de uma ALU de
32 bits (continuação)

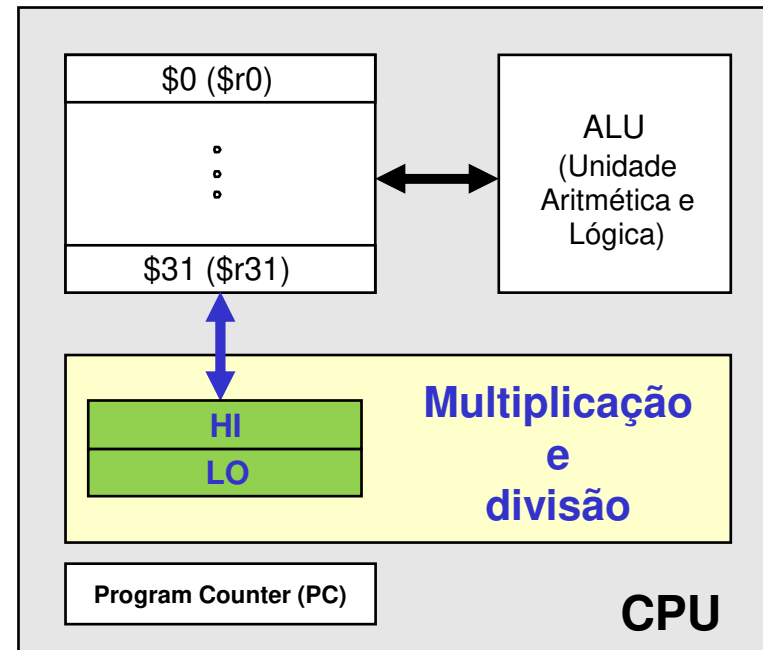
Operation	ALU Action
0 0 0	And
0 0 1	Or
0 1 0	Add
1 1 0	Subtract
1 1 1	Set if less than

Multiplicação de inteiros

- 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
- Multiplicação de quantidades **sem sinal**: algoritmo clássico que é usado na multiplicação em decimal
- Multiplicação de quantidades **com sinal** (representadas em complemento para dois): algoritmo de Booth
- Uma multiplicação que envolva **dois operandos de N bits** carece de um espaço de armazenamento, para o resultado, de **$2*N$ bits**

A Multiplicação de inteiros no MIPS

- No MIPS, a multiplicação e a divisão são asseguradas por um módulo independente da ALU
- Os operandos são registos de 32 bits. Na multiplicação, tal implica que o **resultado** tem de ser armazenado com **64 bits**
- Os resultados são armazenados num par de registos especiais designados por **HI** e **LO**, cada um com 32 bits
- Estes registos são de uso específico da unidade de multiplicação e divisão de inteiros



$$\boxed{\text{Rsrc1}} \times \boxed{\text{Rsrc2}} = \boxed{\text{hi}} \boxed{\text{lo}}$$

A Multiplicação de inteiros no MIPS

- O registo **HI** armazena os **32 bits mais significativos do resultado**
- O registo **LO** armazena os **32 bits menos significativos do resultado**
- 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 **Rdst** # **move from hi**: copia HI para Rdst

mflo **Rdst** # **move from lo**: copia LO para Rdst

- A unidade de multiplicação pode operar considerando os operandos com sinal (multiplicação *signed*) ou sem sinal (multiplicação *unsigned*); a distinção é feita através da mnemónica da instrução:
 - **mult** – multiplicação "signed"
 - **multu** – multiplicação "unsigned"

A Multiplicação de inteiros no MIPS

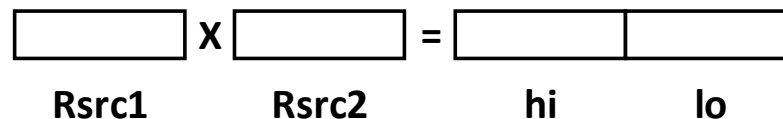
- Em *Assembly*, a multiplicação é então efetuada pelas instruções

mult **Rsrc1**, **Rsrc2** # Multiply (signed)

multu **Rsrc1**, **Rsrc2** # Multiply unsigned

em que **Rsrc1** e **Rsrc2** são os dois registos a multiplicar

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



- Exemplo:** Multiplicar os registos \$t0 e \$t1 e colocar o resultado nos registos \$a1 (32 bits mais significativos) e \$a0 (32 bits menos significativos); os operandos devem ser interpretados com sinal

mult **\$t0, \$t1** # resultado em hi e lo

mfhi **\$a1** # copia hi para registo \$a1

mflo **\$a0** # copia lo para registo \$a0

Instruções virtuais de multiplicação

Multiplicação *signed*

mul	Rdst, Rsrc1, Rsrc2
mult	Rsrc1, Rsrc2
mflo	Rdst

Multiplicação *unsigned*

mulu	Rdst, Rsrc1, Rsrc2
multu	Rsrc1, Rsrc2
mflo	Rdst

Multiplicação *unsigned* com detecção de overflow

mulou	Rdst, Rsrc1, Rsrc2
multu	Rsrc1, Rsrc2
mfhi	\$1
beq	\$1, \$0, cont
break	
cont:	mflo Rdst

Multiplicação *signed* com detecção de overflow

mulo	Rdst, Rsrc1, Rsrc2
mult	Rsrc1, Rsrc2
mfhi	\$1
mflo	Rdst
sra	Rdst, Rdst, 31
beq	\$1, Rdst, cont
break	
cont:	mflo Rdst

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:
 - Divide-se dividendo por divisor, em módulo
 - O quociente tem sinal negativo se os sinais do dividendo e do divisor forem diferentes
 - O resto tem o mesmo sinal do dividendo
- Exemplo 1 (**dividendo = -7, divisor = 3**):

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

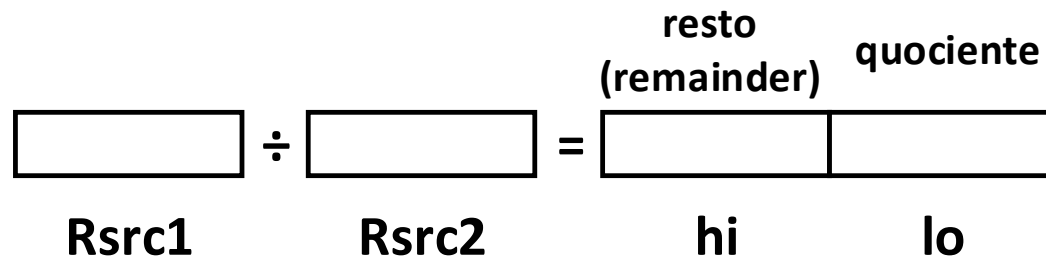
- Exemplo 2 (dividendo = 7, divisor = -3):

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

Note que: **Dividendo = Divisor * Quociente + Resto**

A Divisão de inteiros no MIPS

- Tal como na multiplicação, continua a existir a necessidade de um registo de 64 bits para armazenar o resultado final na forma de um quociente e de um resto
- Os mesmos registos, **HI** e **LO**, que tinham já sido usados para a 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



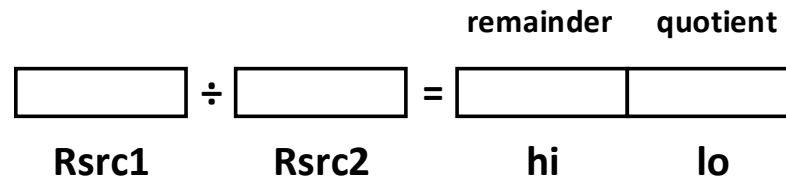
A Divisão de inteiros no MIPS

- No MIPS, as instruções *Assembly* de divisão são:

div **Rsrc1**, **Rsrc2** # Divide (signed)

divu **Rsrc1**, **Rsrc2** # Divide unsigned

- em que **Rsrc1** é o dividendo e **Rsrc2** o divisor. O **resultado** fica armazenado nos registos **HI (resto)** e **LO (quociente)**.



- Exemplo:** obter o resto da divisão inteira entre os valores armazenados em \$t0 e \$t5, colocando o resultado em \$a0

div **\$t0, \$t5** # **hi = \$t0 % \$t5**

lo = \$t0 / \$t5

mfhi **\$a0** # **\$a0 = hi**

Instruções virtuais de divisão

Divisão *signed*

div	Rdst, Rsrc1, Rsrc2
div	Rsrc1, Rsrc2
mflo	Rdst

Divisão *unsigned*

divu	Rdst, Rsrc1, Rsrc2
divu	Rsrc1, Rsrc2
mflo	Rdst

Resto da divisão *signed*

rem	Rdst, Rsrc1, Rsrc2
div	Rsrc1, Rsrc2
mfhi	Rdst

Resto da divisão *unsigned*

remu	Rdst, Rsrc1, Rsrc2
divu	Rsrc1, Rsrc2
mfhi	Rdst

Exercícios

- Para uma codificação em complemento para 2, apresente a gama de representação que é possível obter com 3, 4, 5, 8 e 16 bits (indique os valores-limite da representação em binário, hexadecimal e em decimal com sinal e módulo).
- Determine a representação em complemento para 2 com 16 bits das seguintes quantidades:
 - 5, -3, -128, -32768, 31, -8, 256, -32
- Determine o valor em decimal representado por cada uma das quantidades seguintes, supondo que estão codificadas em complemento para 2 com 8 bits:
 - 00101011_2 , 0xA5, 10101101_2 , 0x6B, 0xFA, 0x80
- Determine a representação das quantidades do exercício anterior em hexadecimal com 16 bits (também codificadas em complemento para 2).

Exercícios

- Como é realizada a detecção de *overflow* em operações de adição com quantidades sem sinal? E com quantidades com sinal (codificadas em complemento para 2)?
- Para a multiplicação de dois operandos de "**m**" e "**n**" bits, respetivamente, qual o número de bits necessário para o armazenamento do resultado?
- Apresente a decomposição em instruções nativas da instrução virtual **mul \$5, \$6, \$7**
- Determine o resultado da instrução anterior, quando **\$6=0xFFFFFFF** e **\$7=0x00000005**.
- Apresente a decomposição em instruções nativas das instruções virtuais **div \$5, \$6, \$7** e **rem \$5, \$6, \$7**
- Determine o resultado das instruções anteriores, quando **\$6=0xFFFFFFF0** e **\$7=0x00000003**

Exercícios

- As duas sub-rotinas do slide seguinte permitem detetar *overflow* nas operações de adição com e sem sinal, no MIPS. Analise o código apresentado e determine o resultado produzido, pelas duas sub-rotinas, nas seguintes situações:
 - `$a0=0x7FFFFFFF1`, `$a1=0x0000000E`;
 - `$a0=0x7FFFFFFF1`, `$a1=0x0000000F`;
 - `$a0=0xFFFFFFFF1`, `$a1=0xFFFFFFFF`;
 - `$a0=0x80000000`, `$a1=0x80000000`;
- Ainda no código das sub-rotinas, qual a razão para não haver salvaguarda de qualquer registo na *stack*?

Exercícios

```
# Overflow detection, signed
# int isovf_signed(int a, int b);
isovf_signed:  ori  $v0,$0,0
               xor  $1,$a0,$a1
               slt  $1,$1,$0
               bne  $1,$0,notovf_s
               addu $1,$a0,$a1
               xor  $1,$1,$a0
               slt  $1,$1,$0
               beq  $1,$0,notovf_s
               ori  $v0,$0,1
notovf_s:      jr   $ra

# Overflow detection, unsigned
# int isovf_unsigned(unsigned int a, unsigned int b);
isovf_unsigned:ori  $v0,$0,0
               nor  $1,$a1,$0
               sltu $1,$1,$a0
               beq  $1,$0,notovf_u
               ori  $v0,$0,1
notovf_u:      jr   $ra
```

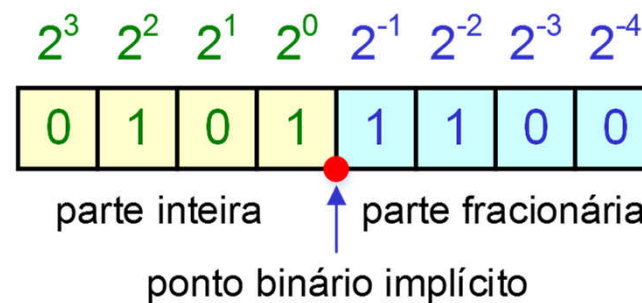
Aulas 12 e 13

- Representação de números em vírgula flutuante
- A norma IEEE 754
 - Operações aritméticas em vírgula flutuante
 - Precisão simples e precisão dupla
 - Casos particulares
 - Representação desnormalizada
 - Arredondamentos
- Unidade de vírgula flutuante do MIPS
 - Instruções da FPU do MIPS
 - Análise de um exemplo de tradução de C para assembly

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Representação de quantidades fracionárias

- A codificação de quantidades numéricas com que trabalhamos até agora esteve sempre associada à representação de números inteiros
- A representação posicional de inteiros pode também ser usada para representar números racionais considerando-se potências negativas da base
- Por exemplo a representação da quantidade 5.75 em base 2 com 4 bits para a parte inteira e 4 bits para a parte fracionária poderia ser:



- Esta representação designa-se por "**representação em vírgula fixa**"

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
- Quantos bits devem ser reservados para a **parte inteira** e quantos para a **parte fracionária**, sabendo nós que o espaço de armazenamento é limitado?
- O número de bits da parte inteira determina a **gama de valores representáveis** (2^4 , no exemplo anterior)
- O número de bits da parte fracionária, determina a **precisão** da representação (passos de $2^{-4} = 0.0625$, no exemplo anterior)

Representação de números em Vírgula Flutuante

- **Exemplo: -23.45129** (vírgula fixa). A mesma quantidade pode também ser representada recorrendo à notação científica:

$$-2.345129 \times 10^1$$

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

$$-0.2345129 \times 10^2$$

$$-(0 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} + \dots + 9 \times 10^{-7}) \times 10^2$$

- 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 no slide anterior, mas usando agora a base dois:

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

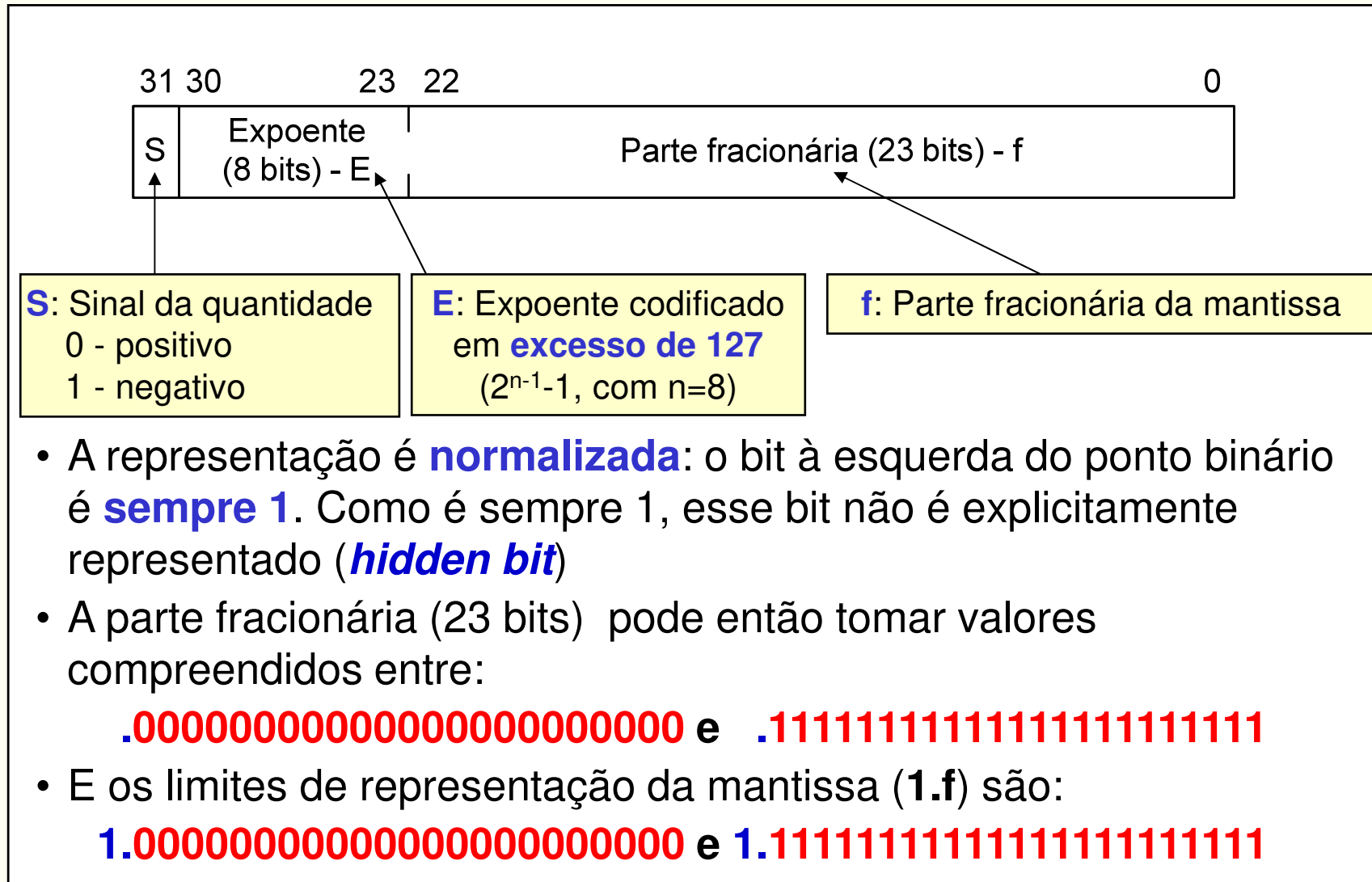
(representação em binário de uma quantidade real, no formato de **vírgula flutuante normalizada**)

- 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)

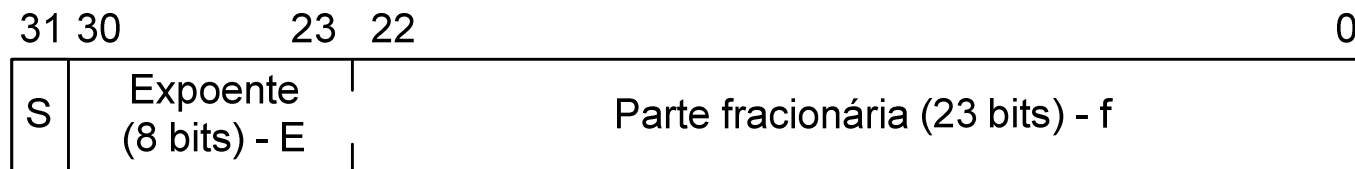


- O expoente é codificado em **excesso de 127** ($2^{n-1}-1$, $n=8$ bits). Ou seja, é somado ao expoente verdadeiro (**Exp**) o valor 127 para obter o código de representação (i.e. **$E = \text{Exp} + 127$** , em que E é o expoente codificado)

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

- 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: Qual o valor, em decimal, representado em **0x41580000**?

0 10000010 101100000000000000000000

Sinal = 0 (quantidade positiva)

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

Mantissa = $(1 + \text{parte fracionária}) = 1 + .1011 = 1.1011$

A quantidade representada (R) será então: **$+1.1011 \times 2^3$**

$$R = +1.1011 \times 2^3 = (1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}) \times 2^3$$

$$= +1.6875 \times 8 = +13.5 \quad (+1.1011 \times 2^3 = +1101.1_2 = +13.5)$$

Norma IEEE 754 (precisão simples)

- Exemplo:** codificar no formato vírgula flutuante IEEE 754 precisão simples, o valor $-12593.75_{10} \times 10^{-3}$

$$-12593.75 \times 10^{-3} = -12.59375$$

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

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

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

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

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

1 **10000010** **100100110000000000000000**

0xC1498000

	0.59375
	× 2
MSb	1 .18750
	0.18750
	× 2
	0 .37500
	0.37500
	× 2
	0 .75000
	0.75000
	× 2
	1 .50000
	0.50000
	× 2
LSb	1 .00000

Norma IEEE 754 (precisão simples)

- A gama de representação suportada por este formato será portanto:

$$\pm [1.000000000000000000000000 \times 2^{-126}, 1.111111111111111111111111 \times 2^{+127}]$$

$$\pm [1.175494 \times 10^{-38}, 3.402824 \times 10^{+38}]$$

- 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:

$$m = \left\lfloor n \frac{\log r}{\log s} \right\rfloor \quad \lfloor . \rfloor \text{ é o operador } floor$$

- 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 n° de bits por casa decimal = $\log_2(10) \cong 3.3$, o número de casas decimais é $\lfloor 23 / 3.3 \rfloor = \mathbf{6 \text{ casas decimais}}$

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 → **$E > 254$**)

$$N_{\text{resultado}} > 1.111111111111111111111111 \times 2^{+127}$$

- Underflow**: caso em que o expoente é tão pequeno que também não é representável → **$E < 1$**)

$$0 < N_{\text{resultado}} < 1.000000000000000000000000 \times 2^{-126}$$

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)

$$N = 1.0000\underline{110} \times 2^1 = 1.0001 \times 2^1$$

1.0000	11
+ 0.0000	10
<hr/>	
1.0001	01

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

$$M_r = 1.1100 \times 1.1001 = 10.101111$$

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)

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

1.0101	111
+ 0.0000	100
<hr/>	
1.0110	011

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

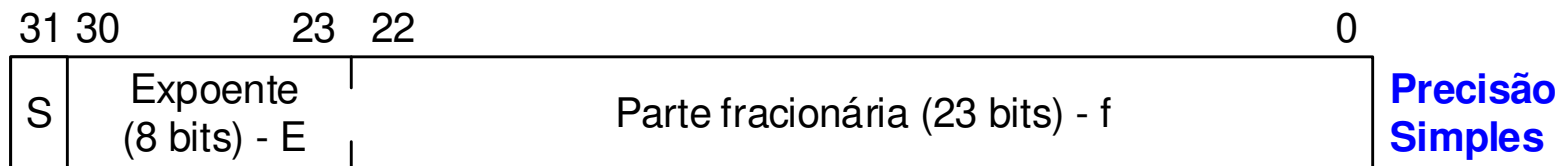
$$N = 1.1 \times 2^1 = 1.1000 \times 2^1$$

1.1000		0
+ 0.0000		1
<hr/>		
1.1000		1

Exemplo com 4 bits fracionários

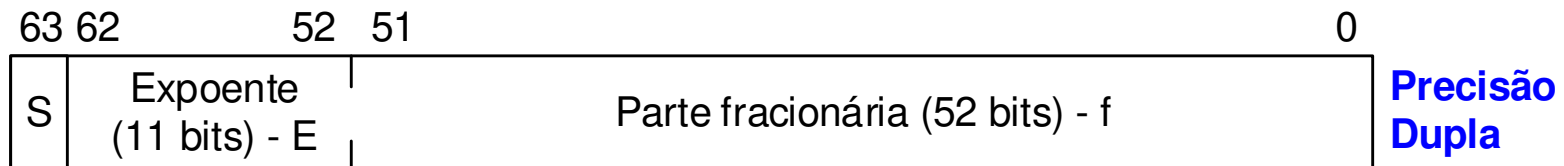
Norma IEEE 754 (precisão dupla)

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



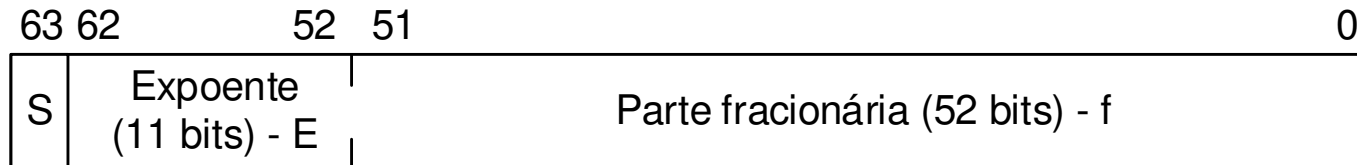
$$N = (-1)^S 1.f \times 2^{(E - 127)} \quad (\text{Precisão simples - tipo float})$$

- e em **precisão dupla (64 bits)**



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

Norma IEEE 754 (precisão dupla)



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

- Na codificação do expoente, **os códigos 0 e 2047 são reservados**. O expoente pode então tomar valores entre **-1022 e +1023** [códigos 1 a 2046]
- A gama de representação suportada pelo formato de precisão dupla será:

$$\pm [1.000000000000000000000000 \times 2^{-1022}, 1.111111111111111111111111 \times 2^{+1023}]$$

$$\pm [2.225073858507201 \times 10^{-308}, 1.797693134862316 \times 10^{+308}]$$

- 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 (inf)**. Gama de representação excedida; divisão por 0. Exemplos: $1.0 / 0.0$, $-1.0 / 0.0$
 - Resultados não numéricos (**NaN – Not a Number**). Exemplo: $0.0 / 0.0$, inf / inf , $\text{nan} * 2$
 - Afim de 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

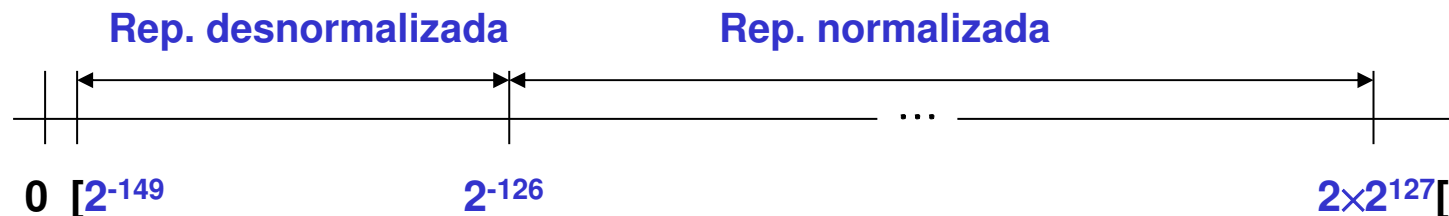
Precisão Simples		Precisão Dupla		Representa
Expoente	Parte Frac.	Expoente	Parte Frac.	
0	0	0	0	0
0	$\neq 0$	0	$\neq 0$	Quantidade desnormalizada
<i>1 a 254</i>	<i>qualquer</i>	<i>1 a 2046</i>	<i>qualquer</i>	<i>Nº em vírgula flutuante normalizado</i>
255	0	2047	0	Infinito
255	$\neq 0$	2047	$\neq 0$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

Norma IEEE 754 – representação desnormalizada

- Representação com mantissa desnormalizada: assume-se que o bit à esquerda do ponto binário é 0
- O expoente codificado é 0; **o expoente verdadeiro é -126 (precisão simples) ou -1022 (precisão dupla)**
- Permite a representação de quantidades cada vez mais pequenas (*underflow* gradual)
- Gama de representação com mantissa desnormalizada, em precisão simples:

[illegible]

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



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

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 bits na parte fracionária: $d=2$)

val	Trunc(val)	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 bits na parte fracionária: $d=2$)

val	Arred(val)	Erro
x.00	x	0
x.01	x	$x - x.25 = -1/4$
x.10	$x + 1$	$(x+1) - x.5 = +1/2$
x.11	$x + 1$	$(x+1) - x.75 = +1/4$

Erro médio

$$= (0 - 1/4 + 1/2 + 1/4) / 4$$

$$= +1/8$$

- Soma-se 1 ao 1º bit à direita do ponto binário e trunca-se o resultado (**arred(val) = trunc(val + 0.5)**)

$$\begin{array}{r} x.00 \\ + 0.1 \\ \hline x.10 \end{array}$$

$$\begin{array}{r} x.01 \\ + 0.1 \\ \hline x.11 \end{array}$$

$$\begin{array}{r} x.10 \\ + 0.1 \\ \hline x+1.00 \end{array}$$

$$\begin{array}{r} x.11 \\ + 0.1 \\ \hline x+1.01 \end{array}$$

- 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 bits na parte fracionária: $d=2$)

val	Arred(val)	Erro	val	Arred(val)	Erro
x0.00	x0	0	x1.00	x1	0
x0.01	x0	-1/4	x1.01	x1	-1/4
x0.10	x0	-1/2	x1.10	x1 + 1	+1/2
x0.11	x1	+1/4	x1.11	x1 + 1	+1/4

- Semelhante à técnica de arredondamento simples, mas decidindo, para o caso “**xx.10**”, em função do primeiro bit à esquerda do ponto binário

$$\begin{array}{r} x0.10 \\ + 0.0 \\ \hline x0.10 \end{array}$$

$$\begin{array}{r} x1.10 \\ + 0.1 \\ \hline x1 + 1.00 \end{array}$$

- **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

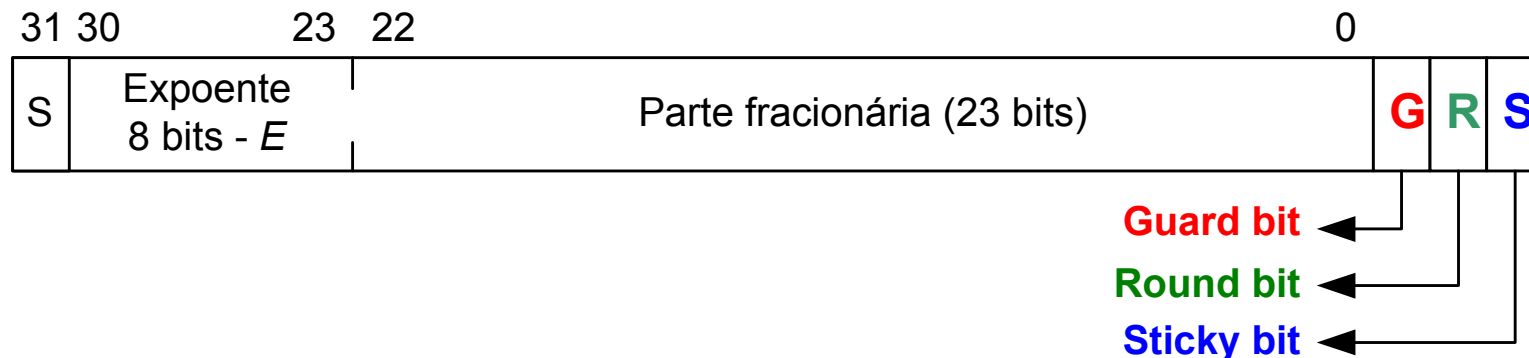
O que fica à direita de b_{23}	Exemplo	Resultado
< 0.5	$1.b_1b_2 \dots b_{22}b_{23} \text{ 011}$	<i>Round down</i> : bits à direita de b_{23} são descartados
> 0.5	$1.b_1b_2 \dots b_{22}b_{23} \text{ 101}$	<i>Round up</i> : soma-se 1 a b_{23} (propagando o <i>carry</i>)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 1 100}$	<i>Round up</i> : soma-se 1 a b_{23} (propagando o <i>carry</i>) (*)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 0 100}$	<i>Round down</i> : bits à direita de b_{23} são descartados (*)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 1 100}$	<i>Round down</i> : bits à direita de b_{23} são descartados (**)
$= 0.5$	$1.b_1b_2 \dots b_{22} \text{ 0 100}$	<i>Round up</i> : soma-se 1 a b_{23} (propagando o <i>carry</i>) (**)

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

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

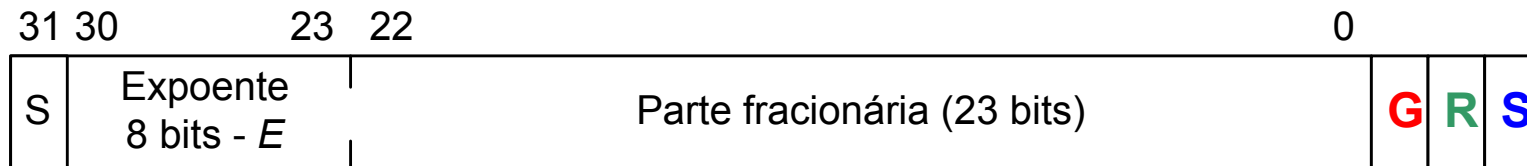
Norma IEEE 754 – arredondamentos

- Os valores resultantes de cada fase intermédia do cálculo 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})



- Objetivos: 1) ter bits suplementares para a pós-normalização e 2) minimizar o erro introduzido pelo processo de arredondamento
 - **G – Guard Bit;**
 - **R – Round bit**
 - **S – Sticky bit** – Resultado 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, $A + B$)

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

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

$$\text{Mant}(A+B) = 1.11010 + 0.0100100 \quad \text{Expoente}(A+B) = 0$$

$$= 10.00011 \text{ 000 } G=0, R=0, S=0$$

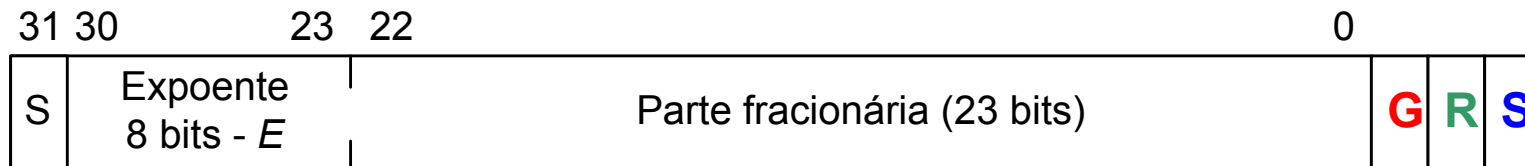
$$\text{Mant}(A+B)_{\text{norm}} = 1.00001 \text{ 100 } G=1, R=0, S=0 \quad \text{Expoente}(A+B) = 1$$

Arredondamento:

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

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

Norma IEEE 754 – arredondamentos



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

Guard bit

Round bit

Sticky bit

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

$$\text{Mant}(A/B) = 1.00001 / 1.11111$$

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

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

$$= 0.10000 \mathbf{1} 11$$

$$\text{Mant}(A/B)_{\text{norm}} = 1.0000 \mathbf{1} \mathbf{1} 10$$

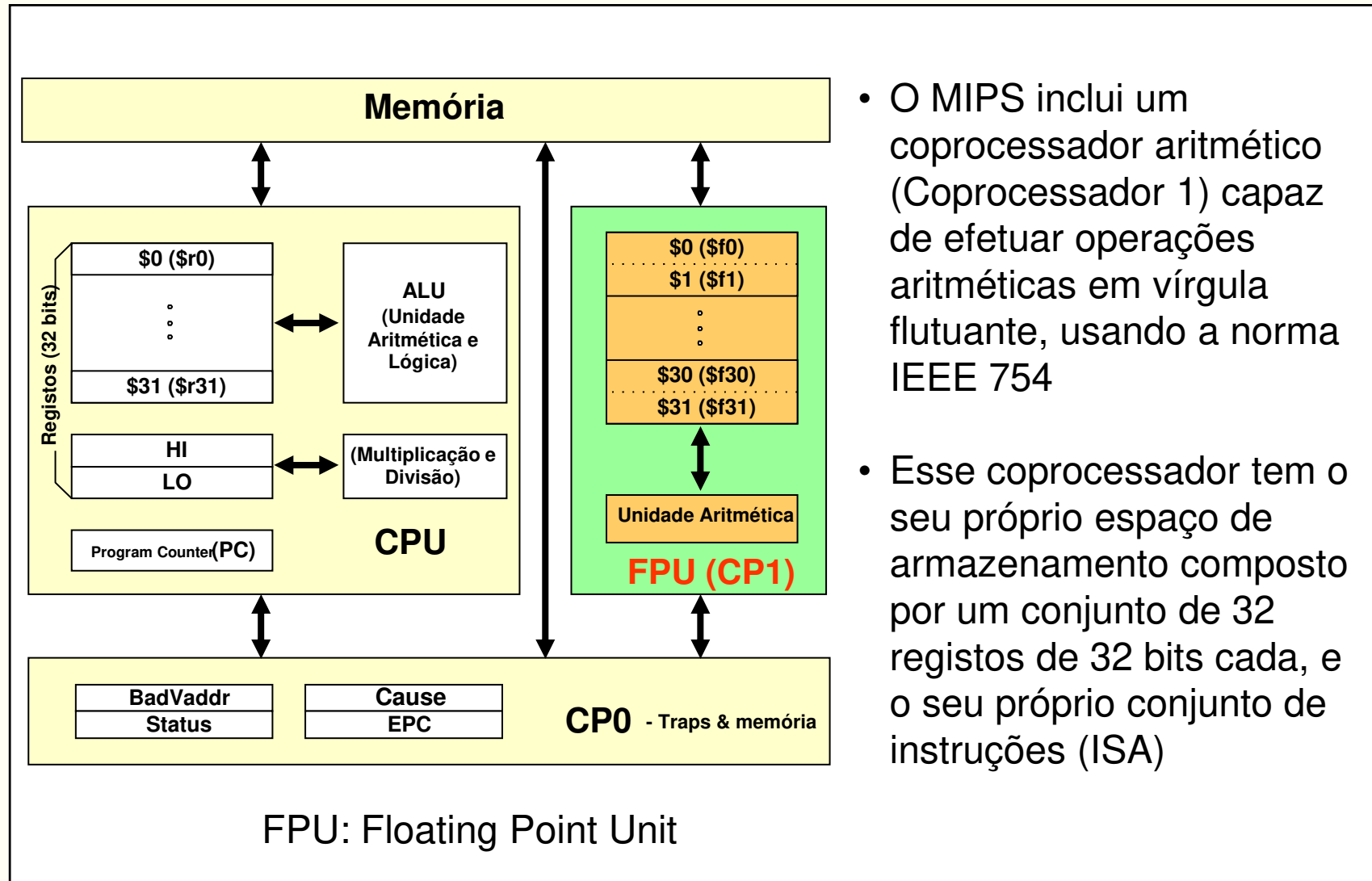
$$\text{Arred}(1, 11_2) = 10_2$$

$$\text{Expoente}(A/B) = 2$$

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

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

Cálculo em Vírgula Flutuante no MIPS



Vírgula Flutuante no MIPS – registos

- Os registos do coprocessador 1 são designados por **\$fn**, em que o índice **n** toma valores entre 0 e 31 (\$f0, \$f1, \$f2, ...)
- 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**.
- A referência ao conjunto de 2 registos faz-se sempre indicando como operando o **registo par** (\$f0, \$f2, \$f4,...)
- **Apenas os registos de índice par podem ser usados no contexto das instruções**

Vírgula Flutuante no MIPS – instruções aritméticas

<code>abs.p</code>	<code>FPdst, FPsrc</code>	<code>#Absolute Value</code>
<code>neg.p</code>	<code>FPdst, FPsrc</code>	<code>#Negate</code>
<code>div.p</code>	<code>FPdst, FPsrc1, FPsrc2</code>	<code>#Divide</code>
<code>mul.p</code>	<code>FPdst, FPsrc1, FPsrc2</code>	<code>#Multiply</code>
<code>add.p</code>	<code>FPdst, FPsrc1, FPsrc2</code>	<code>#Addition</code>
<code>sub.p</code>	<code>FPdst, FPsrc1, FPsrc2</code>	<code>#Subtract</code>

- O sufixo `.p` representa a **precisão** com que é efetuada a operação (simples ou dupla); na instrução é substituído pelas letras `.s` ou `.d` respetivamente
- Exemplos:
 - `add.s $f0, $f4, $f6` `#$f0=$f4 + $f6`
 - `div.d $f4, $f0, $f8` `#$f4 ($f5)=$f0 ($f1) / $f8 ($f9)`

Vírgula Flutuante no MIPS – conversão entre tipos

<code>cvt.d.s</code>	<code>FPdst,FPsrc</code>	<code>#Convert Float to Double</code>
<code>cvt.d.w</code>	<code>FPdst,FPsrc</code>	<code>#Convert Integer to Double</code>
<code>cvt.s.d</code>	<code>FPdst,FPsrc</code>	<code>#Convert Double to Float</code>
<code>cvt.s.w</code>	<code>FPdst,FPsrc</code>	<code>#Convert Integer to Float</code>
<code>cvt.w.d</code>	<code>FPdst,FPsrc</code>	<code>#Convert Double to Integer</code>
<code>cvt.w.s</code>	<code>FPdst,FPsrc</code>	<code>#Convert Float to Integer</code>

- A letra mais à direita especifica o formato original; a letra do meio, especifica o formato do resultado - **s**: float (single), **d**: double, **w**: inteiro
- As **conversões** entre tipos de representação são efetuadas pela FPU: **os registos operando e destino das instruções são obrigatoriamente registos da FPU**

Conversão entre tipos – exemplos

[illegible]

```
cvt.d.s $f6,$f0 #Convert Float to Double
```

$$\mathbf{E} = (129-127) + 1023 = 1025 = 10000000001_2$$

\$f6=0x00000000 \$f7=**1** **100000000001** **1010000...**0

```
$f6=0x00000000 $f7=0xC01A0000
```

```
cvt.w.s $f8,$f0 #Convert Float to Integer
```

$$\mathbf{Exp} = (129-127) = 2$$
$$\mathbf{Val} = -1.625 \times 2^2 = -6.5$$

Resultado: `(int) (-6.5) = trunc(-6.5) = -6`

\$f8=0xFFFFFFFFA (-6 em complemento para 2)

Vírgula Flutuante no MIPS – instruções de transferência

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

Registo do CPU	Registo da FPU	
mtc1	CPUSrc, FPdst	#Move <u>to</u> Coprocessor 1 #Ex: mtc1 \$t0, \$f4
mfc1	CPUdst, FPsrc	#Move <u>from</u> Coprocessor 1 #Ex: mfc1 \$a0, \$f6
mov.s	FPdst, FPsrc	#Move from FPsrc to FPdst (single) #Ex: mov.s \$f4, \$f8
mov.d	FPdst, FPsrc	#Move from FPsrc to FPdst (double) #Ex: mov.d \$f2, \$f0

- Estas instruções copiam o conteúdo integral do registo fonte para o registo destino
- **Não fazem 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

Registo da FPU	Endereço de memória	
↓	↓	
l.s	FPdst , offset (CPUreg)	#Load Float from memory #Ex: l.s \$f0,4(\$a0)
s.s	FPsrc , offset (CPUreg)	#Store Float into memory #Ex: s.s \$f0,0(\$a0)
l.d	FPdst , offset (CPUreg)	#Load Double from memory #Ex: l.d \$f4,8(\$a1)
s.d	FPsrc , offset (CPUreg)	#Store Double into memory #Ex: s.d \$f4,16(\$t0)

Instruções nativas (só muda a mnemónica):

lwc1	FPdst , offset (CPUreg)	#Load Float from memory
swc1	FPsrc , offset (CPUreg)	#Store Float into memory
ldc1	FPdst , offset (CPUreg)	#Load Double from memory
sdc1	FPsrc , offset (CPUreg)	#Store Double into memory

Vírgula Flutuante no MIPS – Manipulação de constantes

- Nas instruções da FPU do MIPS os operandos têm que residir em registos internos, o que significa que **não há suporte para a manipulação direta de constantes**. Como lidar então com operandos que são constantes?
- **Método 1:**
 - Determinar, manualmente, o valor que codifica a constante (32 bits para precisão simples ou 64 bits para precisão dupla)
 - Carregar essa constante em 1 ou 2 registos do CPU e copiar o(s) seu(s) valor(es) para o(s) registo(s) da FPU
- **Método 2:**
 - Usar as directivas “**.float**” ou “**.double**” para definir em memória o valor da constante: 32 bits (**.float**) ou 64 bits (**.double**)
 - Ler o valor da constante da memória para um registo da FPU usando as instruções de acesso à memória (**l.s** ou **l.d**)

Vírgula Flutuante no MIPS – Manipulação de constantes

- O MARS disponibiliza duas instruções virtuais que permitem usar o método 2 (definição da constante em memória) de forma simplificada. Essas instruções têm o seguinte formato:

l.s **FPdst**, **label** **#Ex:** **l.s** **\$f0**, **K1**

l.d **FPdst**, **label** **#Ex:** **l.d** **\$f4**, **K1**

em que “**label**” representa o endereço onde a constante está armazenada em memória.

- A decomposição em instruções nativas destas instruções é (admitindo, por exemplo, que K1 corresponde ao endereço **0x10010008**):

```
l.s    $f0, k1
      lui   $1, 0x1001
      l.s   $f0, 0x0008 ($1)
```

```
l.d    $f4, k1
      lui   $1, 0x1001
      l.d   $f4, 0x0008 ($1)
```

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.xx.s FPUreg1, FPUreg2 # compare float`

`c.xx.d FPUreg1, FPUreg2 # compare double`

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

EQ - equal

LT - less than

LE - less or equal

Exemplos:

`c.eq.s $f0, $f2 / c.le.d $f4, $f8`

- Instruções de salto:

`bc1t label # branch if true`

`bc1f label # branch if false`

Vírgula Flutuante no MIPS – instruções de decisão

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

```
# a: $f0  
# b: $f2  
...  
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 de utilização dos registos

- Registos para **passar parâmetros** para sub-rotinas (do tipo float ou double):
 - **\$f12** (\$f13), **\$f14** (\$f15), por esta ordem
- Registos para **devolução de resultados** das sub-rotinas:
 - **\$f0** (\$f1)
- Registos que **podem** ser livremente usados e alterados pelas sub-rotinas ("caller-saved"):
 - **\$f0** (\$f1) a **\$f18** (\$f19)
- Registos que **não podem** ser alterados pelas sub-rotinas ("callee-saved"):
 - **\$f20** (\$f21) a **\$f30** (\$f31)

Tradução C / Assembly – Exemplo 1

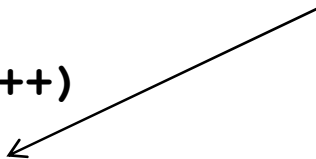
```
#define SIZE 25
double average(double *, int);

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

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

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

Conversão entre tipos
(inteiro para double)



Tradução C / Assembly – Exemplo 1

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

double average(double *, int)

```
        .data
array:   .space 200           # 8*SIZE (alinhado múltiplo 8)
        .eqv SIZE,25
        .text
        .globl main          # avg: $f12
main:    ...                  # Salvaguarda $ra
        la      $a0, array    #
        li      $a1, SIZE     #
        jal     average       #
        mov.d   $f12, $f0     # avg = average(array, SIZE)
        li      $v0, 3        #
        syscall              # print_double(avg)
        ...                  # Repõe $ra
        jr      $ra           #
```

Tradução C / Assembly – Exemplo 1

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

```
# sum: $f0 / tmp1: $f4 / i: $t0 / tmp2: $t1
average: mtc1      $0, $f0          #
          cvt.d.w  $f0, $f0          # sum = 0.0
          li       $t0, 0           # i = 0
for:      bge      $t0, $a1, endf    # while(i < N) {
          sll      $t1, $t0, 3       # tmp = i * 8
          addu     $t1, $t1, $a0     # $t1 = &v[i]
          l.d      $f4, 0($t1)      # $f4 = v[i]
          add.d    $f0, $f0, $f4     # sum += 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     # return sum / (double)N
          jr       $ra              #
```

Tradução C / Assembly – Exemplo 2

```
float fun(float, int);

void main(void)
{
    float res;

    res = fun( 12.5E-2, 2 );
    print_float( res );    // syscall 2
}
```

```
float fun(float a, int m)
{
    float val;
    if( a >= -5.6 )
        val = (float)m * (a - 32.0);
    else
        val = 0.0;
    return val;
}
```

Tradução C / Assembly – Exemplo 2

```
void main(void)
{
    float res;

    res = fun( 12.5E-2, 2 );
    print_float( res );    // syscall 2
}
```

```
float fun(float a, int k)
```

```
.data
k1:    .float 12.5E-2        # 12.5 x 10-2
k2:    .float -5.6
k3:    .float 32.0
k4:    .float 0.0
.text
.globl main                  # res: $f12
main:  ...                  # salvaguarda $ra
      l.s    $f12, k1        # $f12 = 12.5E-2
      li     $a0, 2          # $a0 = 2
      jal    fun             #
      mov.s  $f12, $f0        # res = fun(12.5E-2, 2)
      li     $v0, 2          #
      syscall                # print_float(res)
      ...                  # repõe $ra
      jr     $ra             #
```


Tradução C / Assembly – Exemplo 2

```
float fun(float a, int m)
{
    float val;
    if( a >= -5.6)
        val = (float)m * (a - 32.0);
    else
        val = 0.0;
    return val;
}
```

```
.data
k1: .float 12.5E-2
k2: .float -5.6
k3: .float 32.0
k4: .float 0.0
```

```
# val: $f2 / a: $f12 / m: $a0
```

```
fun:  l.s      $f0, k2          # $f0 = -5.6
      c.lt.s   $f12, $f0       # if( a >= -5.6 )
      bclt     else           # {
      l.s      $f2, k3         #     val = 32.0
      sub.s    $f2, $f12, $f2  #     val = a - 32.0
      mtc1     $a0, $f0        #     $f0 = m
      cvt.s.w  $f0, $f0        #     $f0 = (float)m
      mul.s    $f2, $f0, $f2   #     val = (float)m * val
      j        endif          # } else
else:  l.s      $f2, k4         #     val = 0.0
endif: mov.s   $f0, $f2        # return val;
      jr      $ra             #
```

Exercícios

- Na conversão de uma quantidade codificada em formato IEEE754 precisão simples para decimal, qual o número máximo de casas decimais com que o resultado deve ser apresentado? E se o valor original estiver representado em formato IEEE754 precisão dupla?
- Determine a representação em formato IEEE754 precisão simples da quantidade real **19,1875**. Determine a representação da mesma quantidade em precisão dupla
- Determine o valor em decimal da quantidade representada em formato IEEE754, precisão simples, como **0xC19AB000**
- Determine o valor em decimal da quantidade representada em formato IEEE754, precisão simples, como **0x80580000**

Exercícios

- Considere que o conteúdo dos dois seguintes registros da FPU representam a codificação de duas quantidades reais no formato IEEE754 precisão simples:

- `$f0 = 0x416A0000`

- `$f2 = 0xC0C00000`

Calcule o resultado das instruções seguintes, apresentando o resultado em hexadecimal:

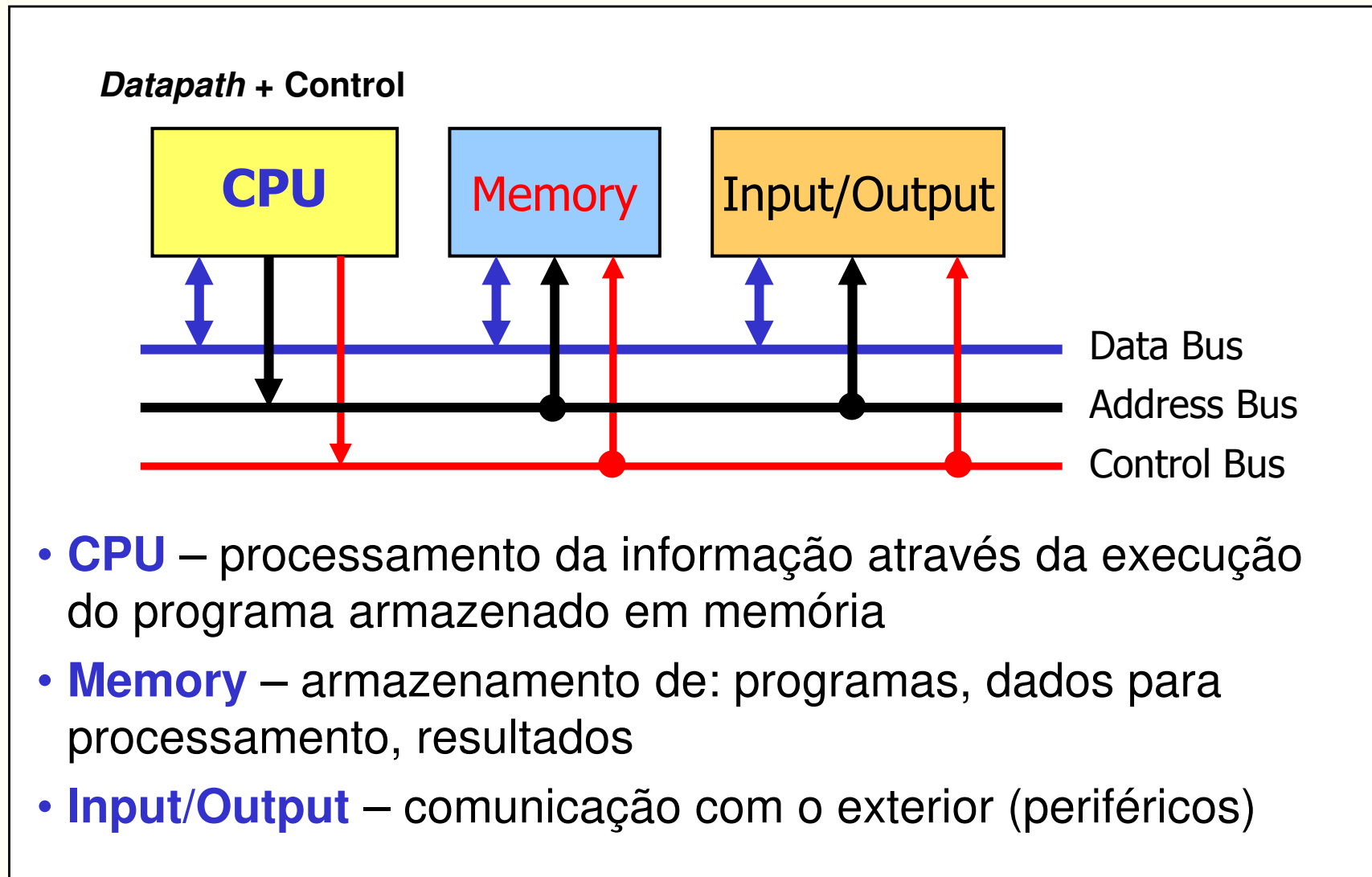
▪ <code>abs.s</code>	<code>\$f4, \$f2</code>	# <code>\$f4 = abs(\$f2)</code>
▪ <code>neg.s</code>	<code>\$f6, \$f0</code>	# <code>\$f6 = neg(\$f0)</code>
▪ <code>sub.s</code>	<code>\$f8, \$f0, \$f2</code>	# <code>\$f8 = \$f0 - \$f2</code>
▪ <code>sub.s</code>	<code>\$f10, \$f2, \$f0</code>	# <code>\$f10 = \$f2 - \$f0</code>
▪ <code>add.s</code>	<code>\$f12, \$f0, \$f2</code>	# <code>\$f12 = \$f0 + \$f2</code>
▪ <code>mul.s</code>	<code>\$f14, \$f0, \$f2</code>	# <code>\$f14 = \$f0 * \$f2</code>
▪ <code>div.s</code>	<code>\$f16, \$f0, \$f2</code>	# <code>\$f16 = \$f0 / \$f2</code>
▪ <code>div.s</code>	<code>\$f18, \$f2, \$f0</code>	# <code>\$f18 = \$f2 / \$f0</code>
▪ <code>cvt.d.s</code>	<code>\$f20, \$f2</code>	# Convert single to double
▪ <code>cvt.w.s</code>	<code>\$f22, \$f0</code>	# Convert single to integer

Aulas 14, 15 e 16

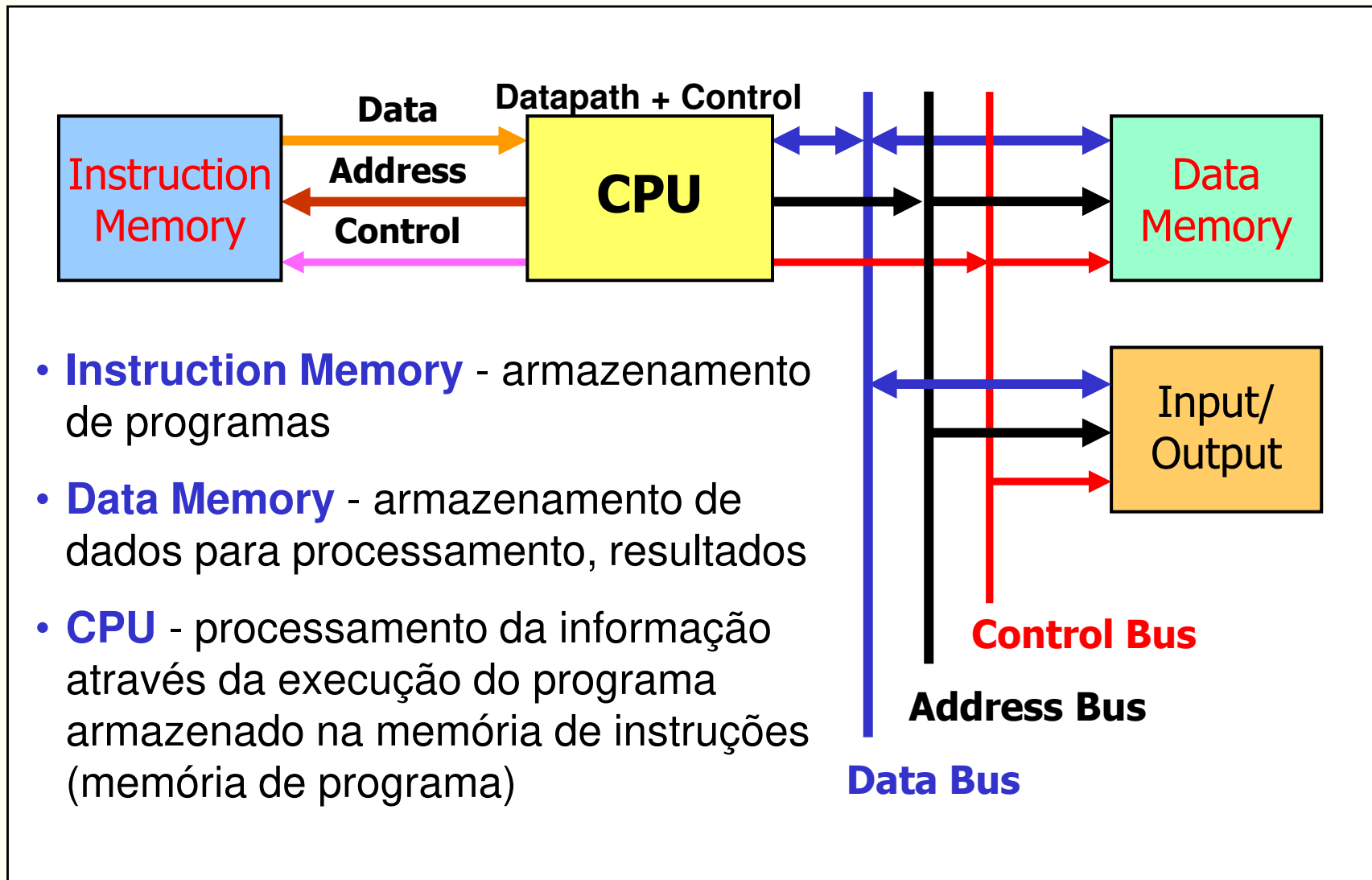
- Modelos de Harvard e Von Neumann
- Blocos constituintes de um *datapath* genérico para uma arquitetura tipo MIPS
- Análise dos blocos necessários à execução de um subconjunto de instruções do MIPS, de cada classe de instruções:
 - Aritméticas e lógicas (add, addi, sub, and, or, slt, slti)
 - Acesso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

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Modelo de von Neumann



Modelo de Harvard



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 comprimentos de palavra diferentes

Implementação de um *Datapath*

- O CPU consiste, fundamentalmente, em duas secções:
 - **Secção de dados** (*datapath*) - elementos operativos/funcionais para armazenamento, processamento e encaminhamento da informação:
 - Registos
 - 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 agrupados num banco de registos, ou outros 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 nestas aulas, e por uma questão de legibilidade dos diagramas, considera-se a memória externa como um elemento operativo integrante do *datapath* (elemento de estado)

Implementação de um *Datapath*

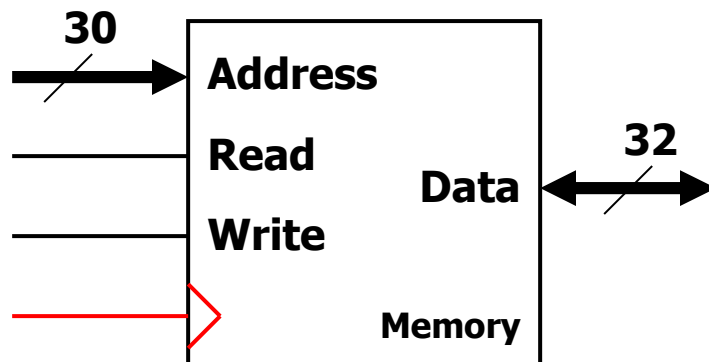
- Para além do sinal de relógio, um elemento de estado pode ainda ter sinais de controlo adicionais:
 - **Um sinal de leitura (read)**, que permite (quando ativo) que a informação armazenada seja disponibilizada na saída (leitura assíncrona)
 - **Um sinal de escrita (write)**, que autoriza (quando ativo) 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, por uma questão de simplificação dos diagramas, o sinal de relógio pode não ser sempre explicitamente representado

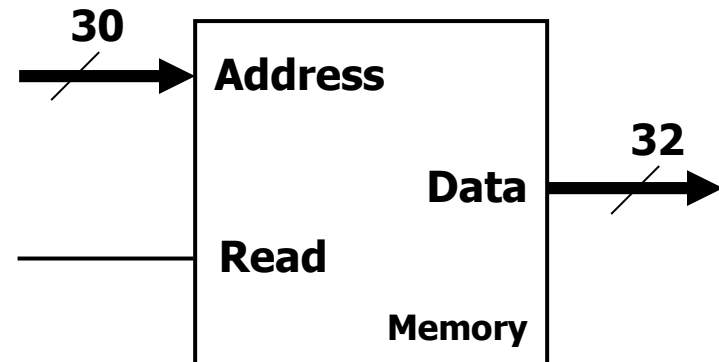
Implementação de um *Datapath*

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

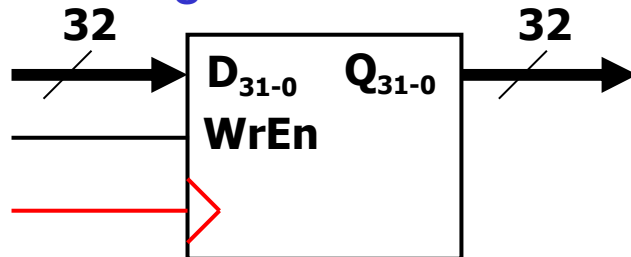
Memória para escrita e leitura
(2^{30} words de 32 bits)



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



Registo 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* - MIPS

- 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 e slti**)
 - Instruções de acesso à memória: load word (**lw**) e store word (**sw**)
 - As instruções de salto condicional (**beq**) e salto incondicional (**j**)
- 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* - MIPS

- No caso do MIPS, para qualquer instrução que compõe o *set* de instruções, **as duas primeiras operações necessárias à sua execução são sempre as mesmas:**
 1. Usar o conteúdo do registo *Program Counter* (PC) como endereço da memória do qual vai ser lido o código máquina da 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 “**j**”)
- **Depois destas operações genéricas, realizam-se as ações específicas para completar a execução da instrução em causa**

Implementação de um *Datapath* - MIPS

- 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 instruções** (à exceção do salto incondicional) **utilizam a ALU depois da leitura dos registos**:
 - as instruções aritméticas e lógicas para a operação correspondente à 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) com o endereço-alvo
 - o endereço-alvo é obtido a partir dos 26 LSbits do código máquina da instrução e dos 4 bits mais significativos do valor atual do PC

Implementação de um *Datapath* - MIPS

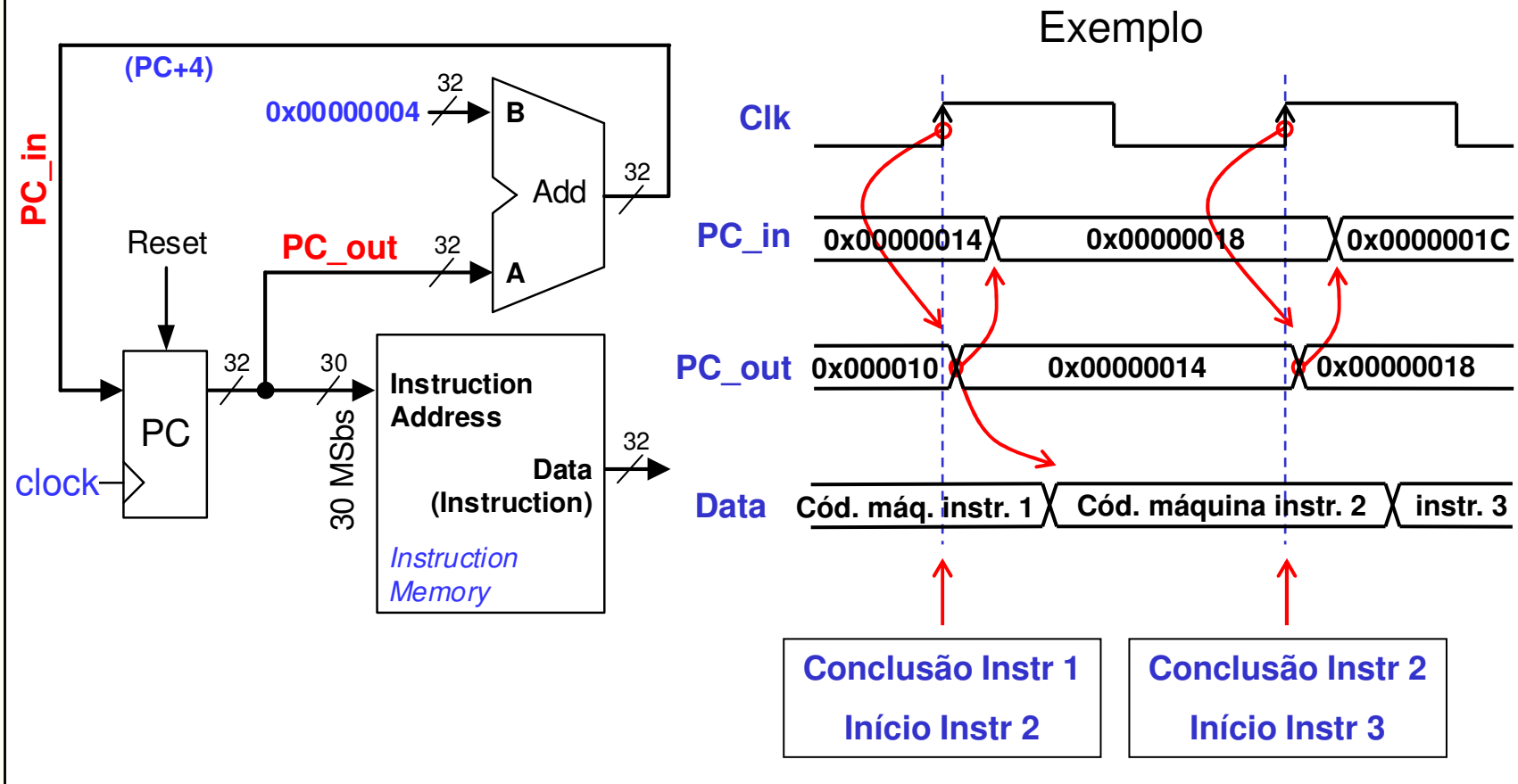
- Depois de utilizar a ALU, as ações que completam as várias classes de instruções diferem:
 - as instruções **aritméticas e lógicas** armazenam o resultado à saída da ALU no registo destino especificado na instrução
 - a instrução "**sw**" acede à memória para escrita do valor do registo lido anteriormente (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 "**beq**" 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 em teste 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***
- 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



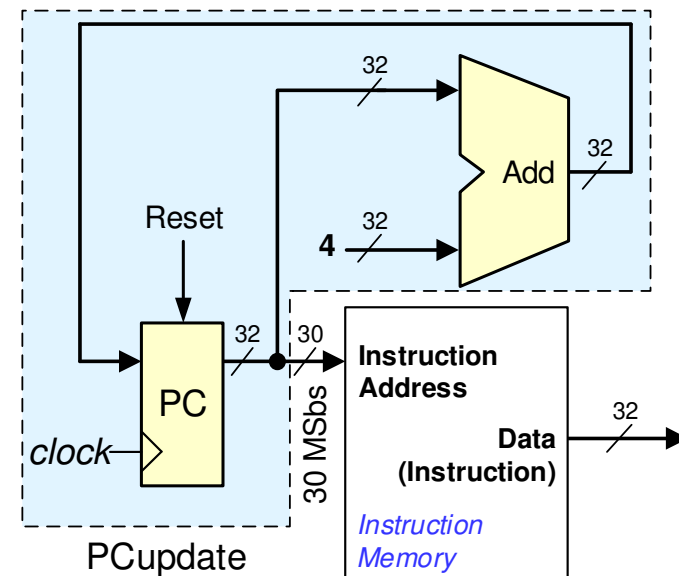
Implementação de um *Datapath* – Atualização do PC

```

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"8C610004",  -- lw    $1,4($3)
                                     X"20210004",  -- addi  $1,$1,4
                                     X"AC610008",  -- sw    $1,8($0)
                                     others => X"00000000");

begin
    readData <= s_memory(to_integer(unsigned(address)));
end Behavioral;

```

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**)
 - Instrução de salto condicional (Tipo I: **beq**)

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 no decurso da execução de uma instrução 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**”)

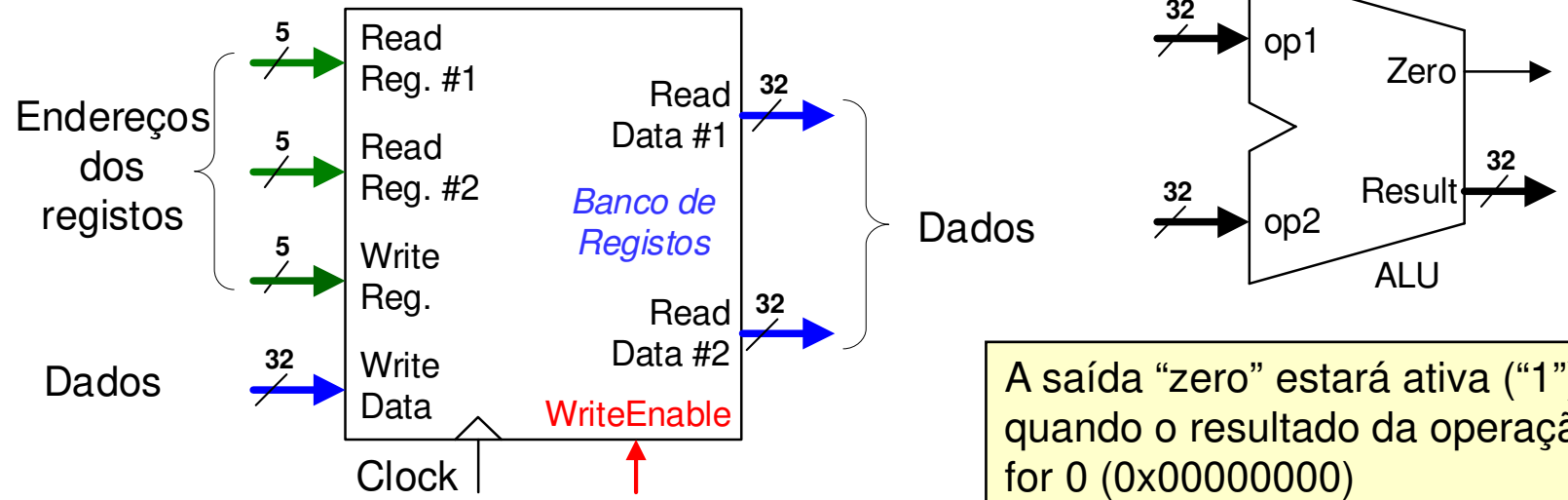
Exemplo: **add** \$2, \$3, \$4



Código máquina: 0x00641020

Implementação de um *Datapath* – instruções tipo R

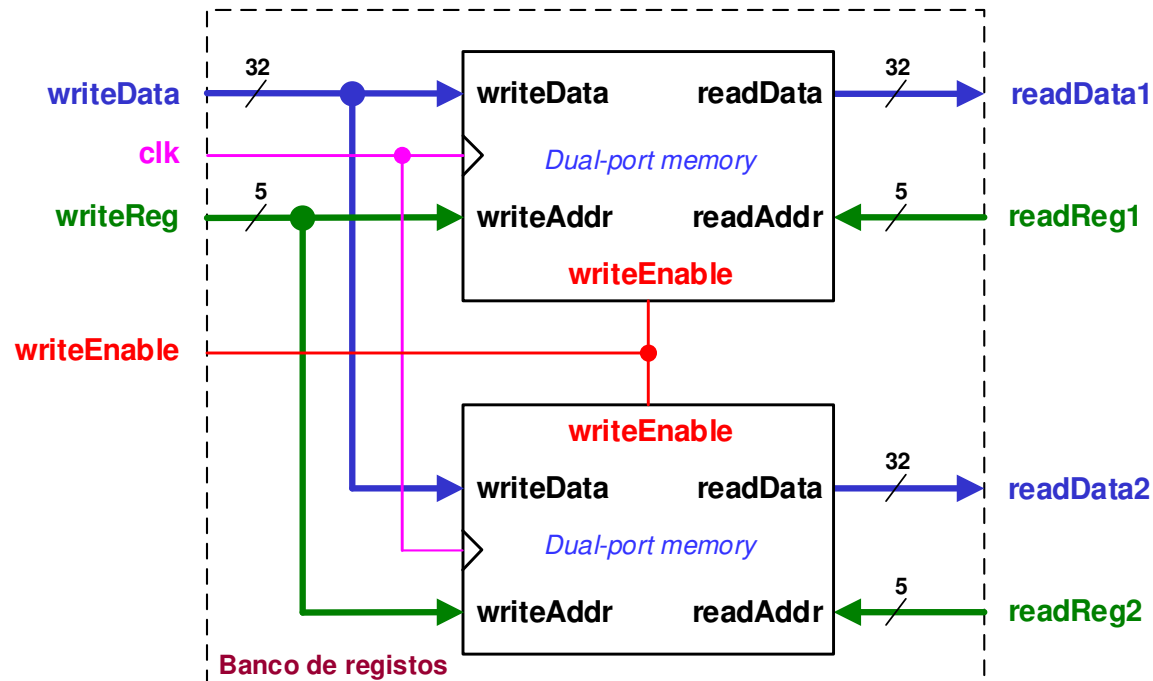
- Os elementos necessários à execução das instruções aritméticas e lógicas (tipo R) são:
 - Uma ALU de 32 bits
 - Um conjunto de registos internos (Banco de registos com 32 registos de 32 bits cada)



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

Banco de Registos

- 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;
```

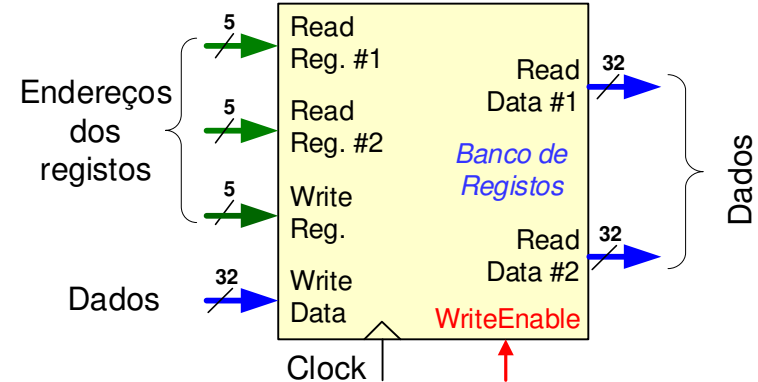
Banco de registros – VHDL

```

library ieee;
use ieee.std_logic_1164.all;

entity RegFile is
  port (clk      : in  std_logic;
        -- synchronous write port
        writeEnable : in  std_logic;
        writeReg    : in  std_logic_vector( 4 downto 0);
        writeData   : in  std_logic_vector(31 downto 0);
        -- asynchronous read port #1
        readReg1    : in  std_logic_vector( 4 downto 0);
        readData1   : out std_logic_vector(31 downto 0);
        -- asynchronous read port #2
        readReg2    : in  std_logic_vector( 4 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)

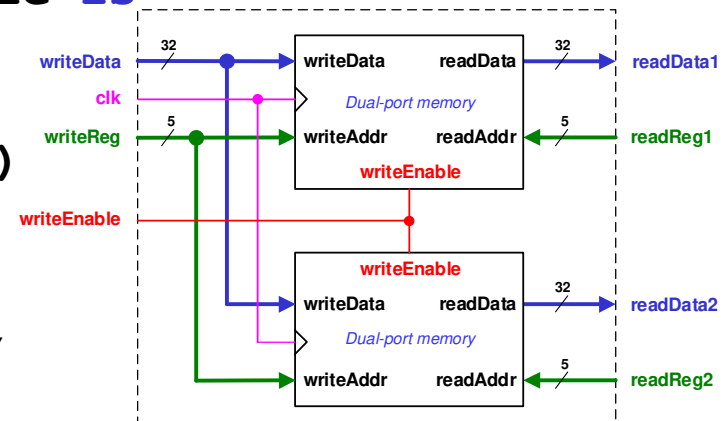
```
port map (clk      => clk,
          readAddr  => readReg1,
          readData  => readData1,
          writeAddr  => writeReg,
          writeData  => writeData,
          writeEnable => writeEnable);
```

rt_mem:

entity work.DP_Memory (Behavioral)

```
port map (clk      => clk,
          readAddr  => readReg2,
          readData  => readData2,
          writeAddr  => writeReg,
          writeData  => writeData,
          writeEnable => writeEnable);
```

end Structural;



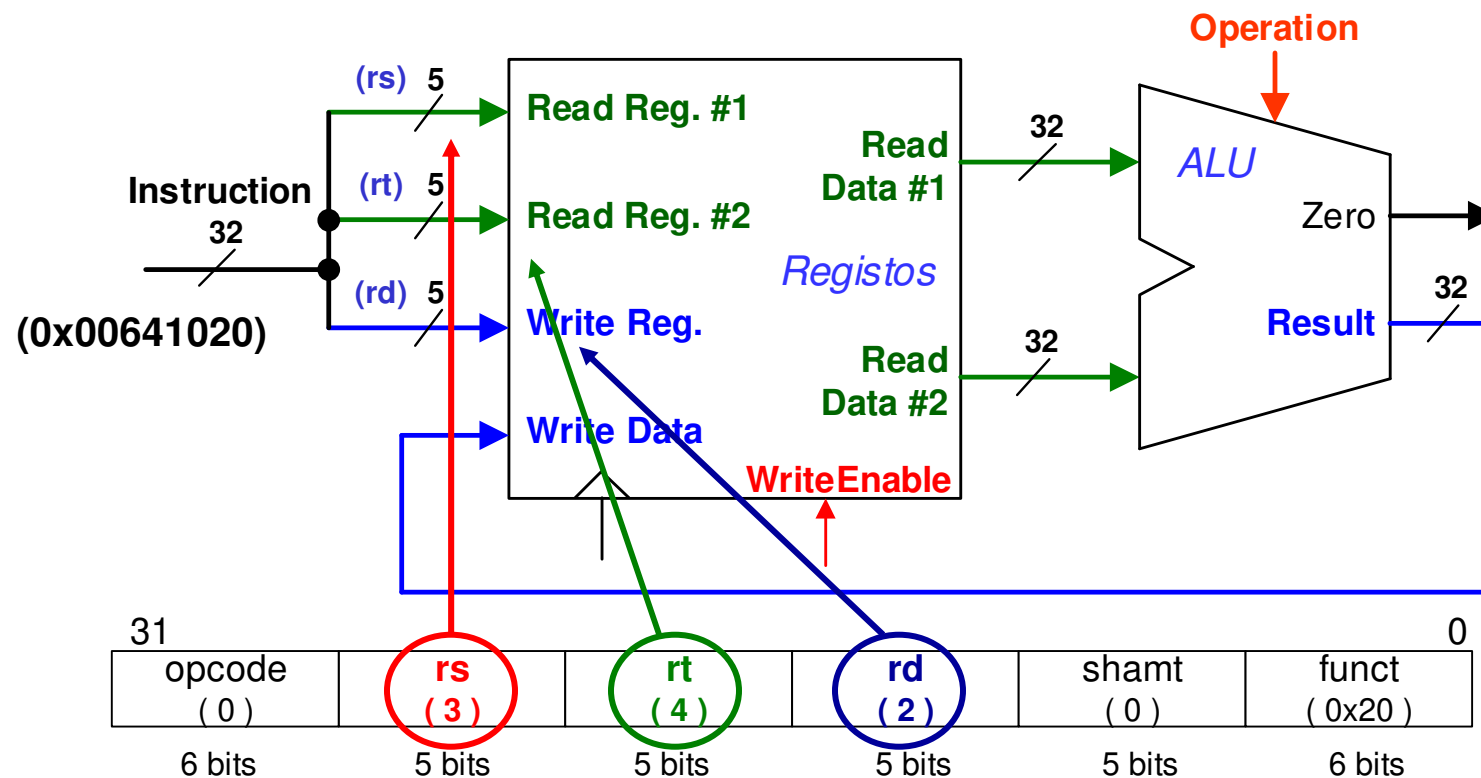
```
entity RegFile is
port ( clk
      writeEnable
      writeReg
      writeData
      readReg1
      readData1
      readReg2
      readData2
end RegFile;
```

Implementação de um *Datapath* – instruções tipo R

- Interligação dos elementos operativos para a execução de uma instrução tipo R:

Ex.: **add** \$2, \$3, \$4

00000000011001000001000000100000



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 (0x2B)	rs (4)	rt (2)	offset (0x24)
--------------------	--------------------	--------------------	---------------------------

Implementação de um *Datapath* (Instrução LW)

- Operações realizadas na execução de uma instrução “**lw**”
 - *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)

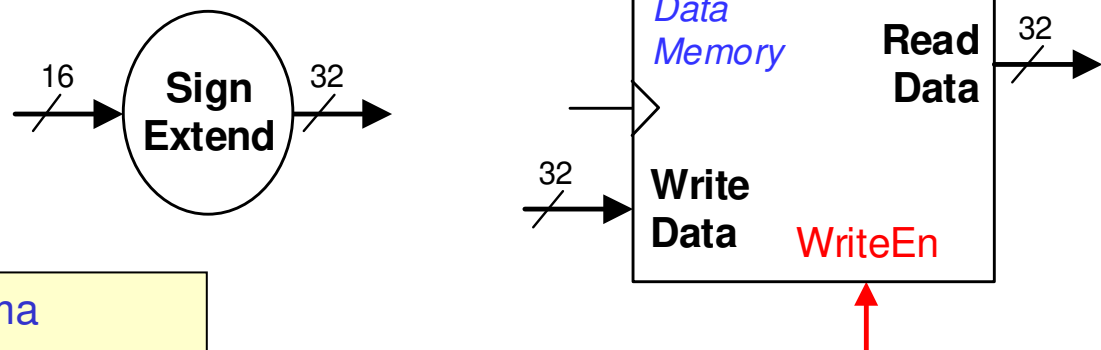
Exemplo: **lw** \$4, **0x2F(\$15)**

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

opcode (0x23)	rs (15)	rt (4)	offset (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

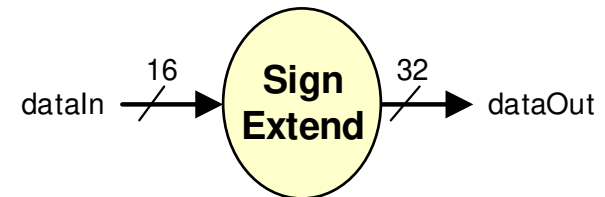


O **extensor de sinal** cria uma constante de 32 bits em complemento para 2, a partir dos 16 bits menos significativos da instrução (o bit 15 é replicado nos 16 mais significativos da constante de saída)

Por uma questão de conveniência de desenho dos diagramas, o barramento de dados da memória (bidirecional) está separado em dados para escrita e dados de leitura

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;
```

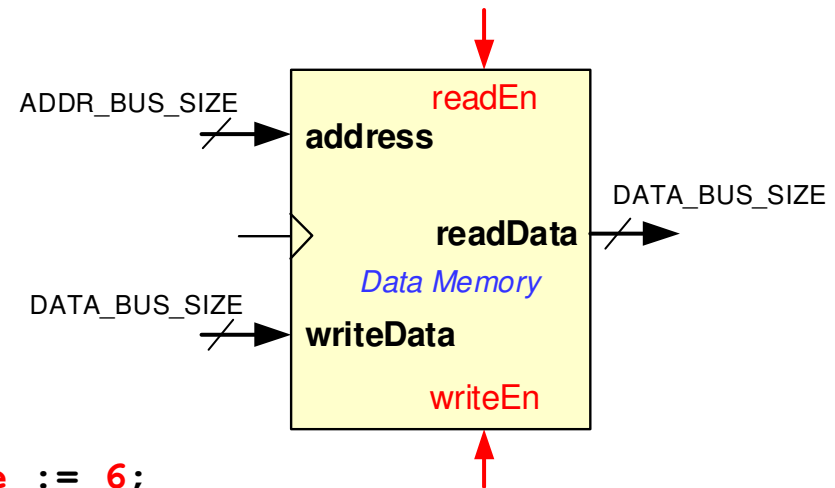
```
architecture Behavioral of SignExtend is  
begin  
    dataOut(31 downto 16) <= (others => dataIn(15));  
    dataOut(15 downto 0)  <= dataIn;  
end Behavioral;
```


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 => 'Z');

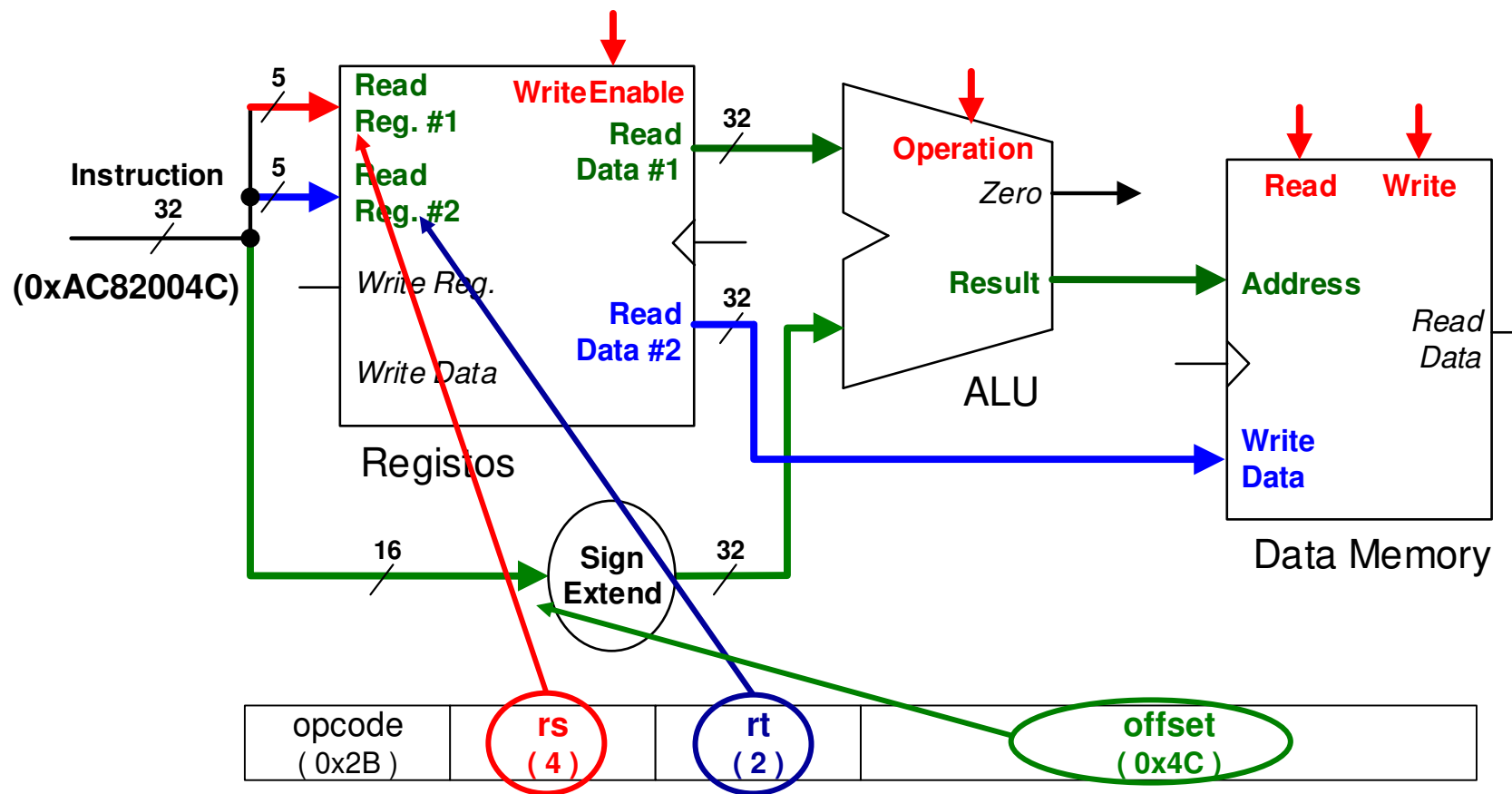
end Behavioral;
```

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

- Interligação dos elementos operativos para a execução do “**sw**”:

Ex: **sw** \$2, 0x4C(\$4)

10101100100000100000000001001100

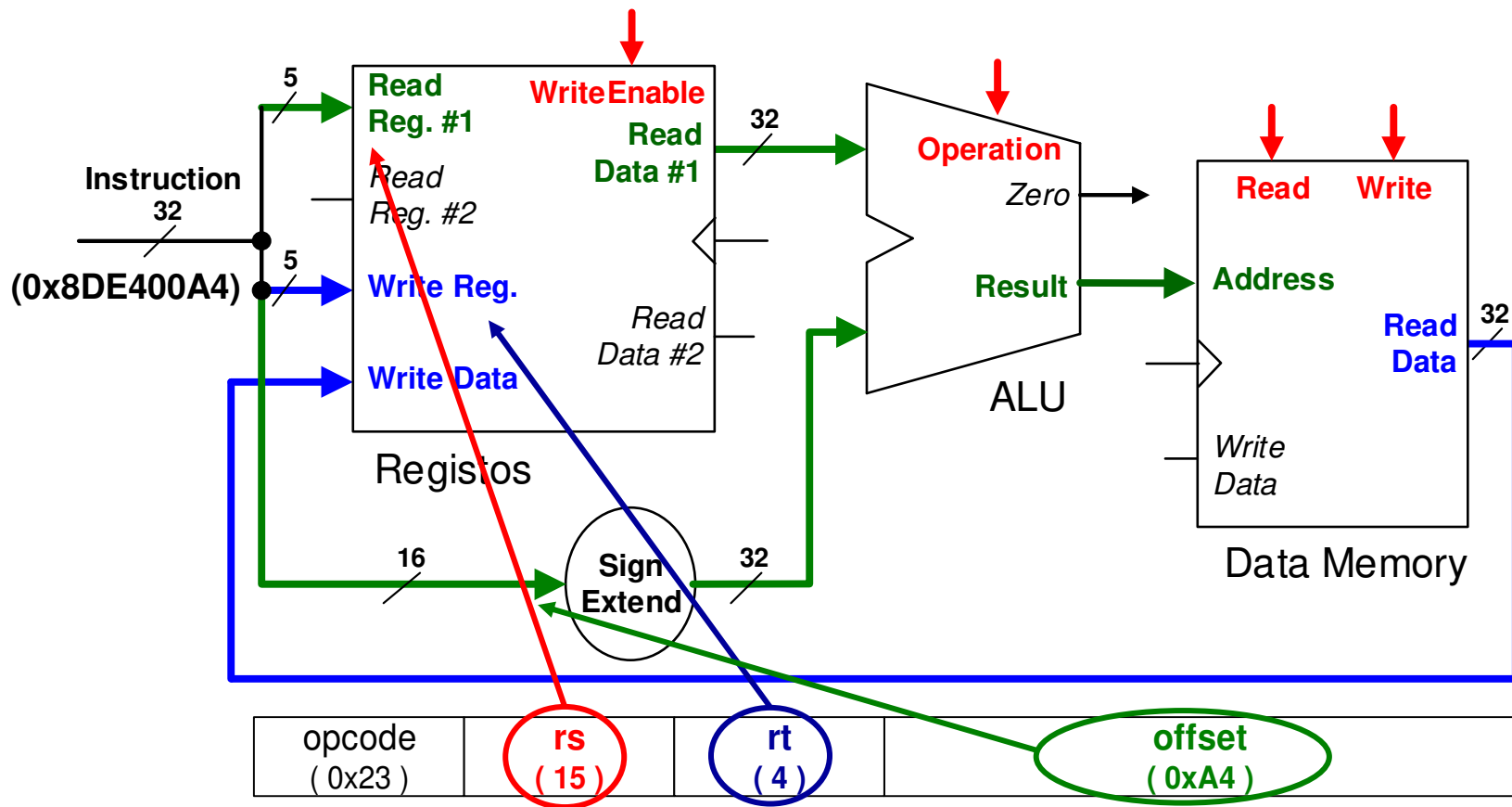


Implementação de um *Datapath* (Instruções lw e sw)

- Interligação dos elementos operativos para a execução do “lw”:

Ex: lw \$4, 0xA4(\$15)

10001101111001000000000010100100

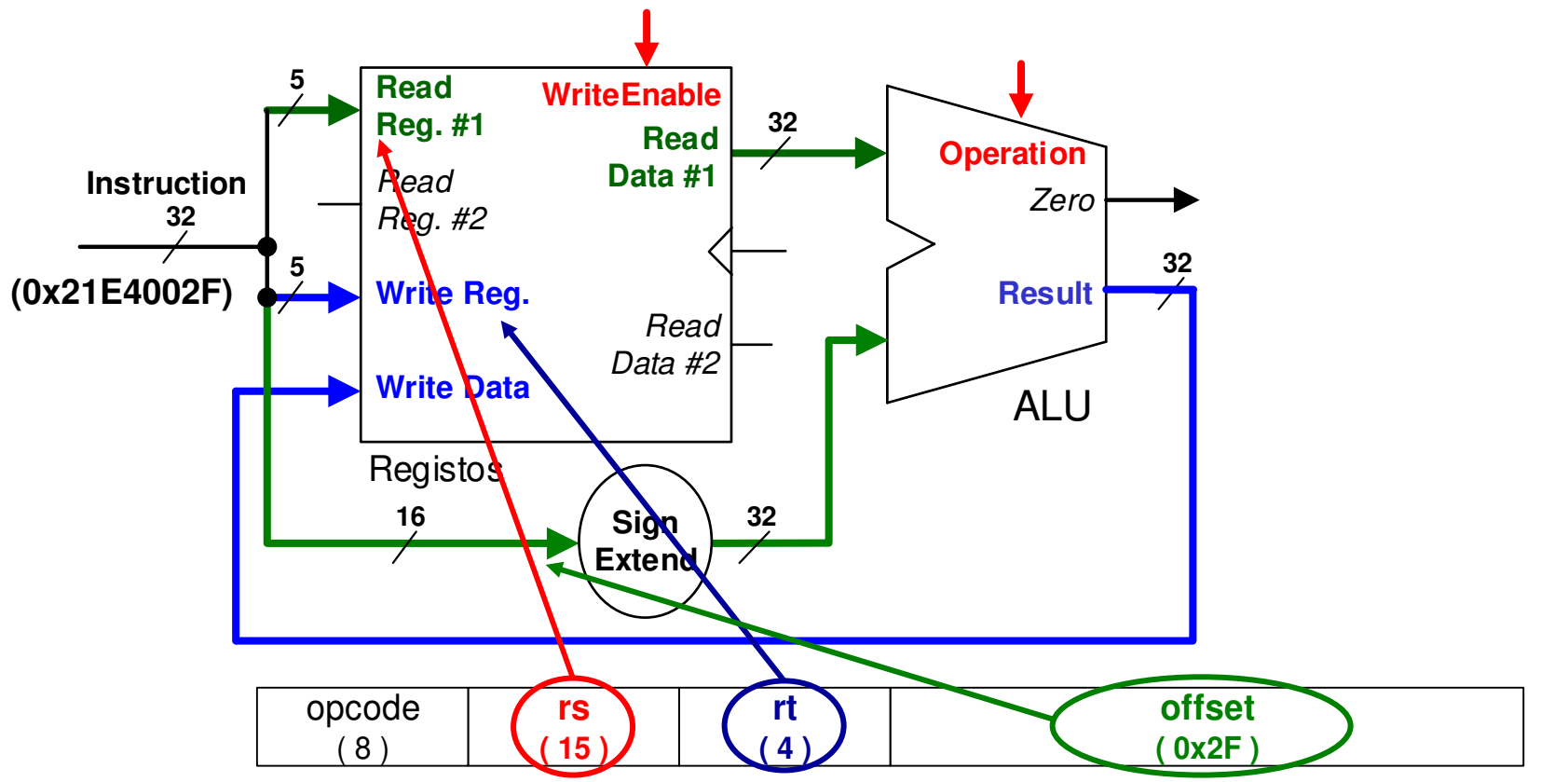


Implementação de um *Datapath* (Instruções “imediatas”)

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

Ex: **addi** \$4, \$15, 0x2F

00100001111100100000000000000101111



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 de dois registos, do banco de registos
 - Comparação dos conteúdos dos registos (realização de uma operação de subtração na ALU)
 - Cálculo do endereço-alvo da instrução de *branch* (*Branch Target Address* - BTA)

$$\text{BTA} = (\text{PC}+4) + (\text{instruction_offset} \ll 2)$$

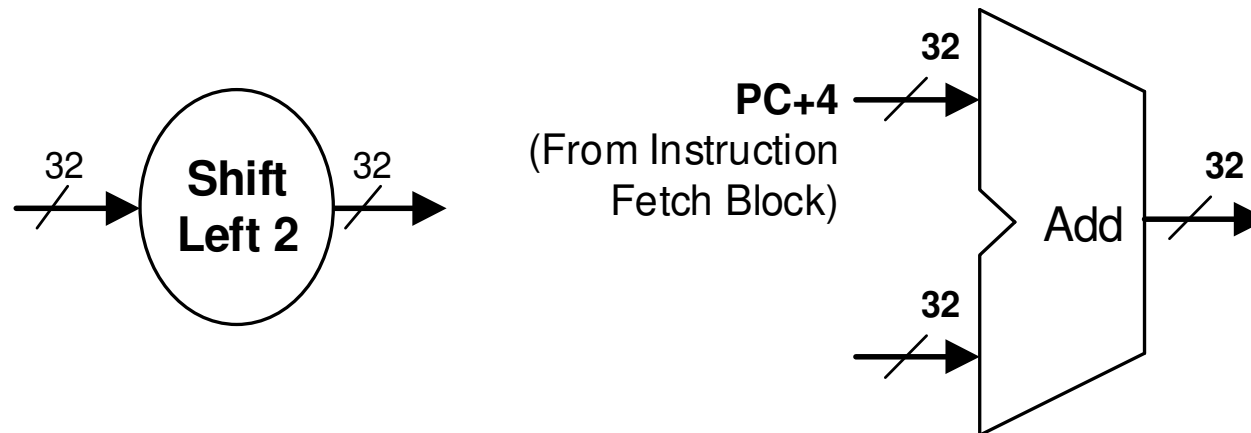
- Alteração do valor do registo PC:
 - se a condição testada pelo *branch* for verdadeira PC = BTA
 - se a condição testada pelo *branch* for falsa PC = PC + 4

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

opcode (4)	rs (2)	rt (3)	instruction_offset (0x20)
-----------------	-------------	-------------	--------------------------------

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

- 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 da diferença de endereços que são desprezados no momento da codificação da instrução

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));
```

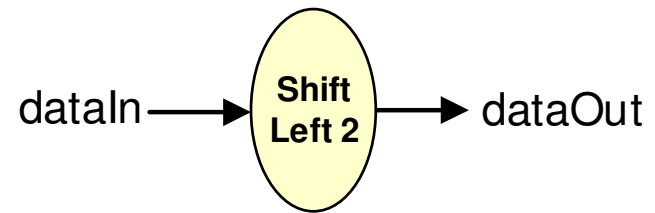
```
end LeftShifter2;
```

```
architecture Behavioral of LeftShifter2 is
```

```
begin
```

```
    dataOut <= dataIn(29 downto 0) & "00";
```

```
end Behavioral;
```

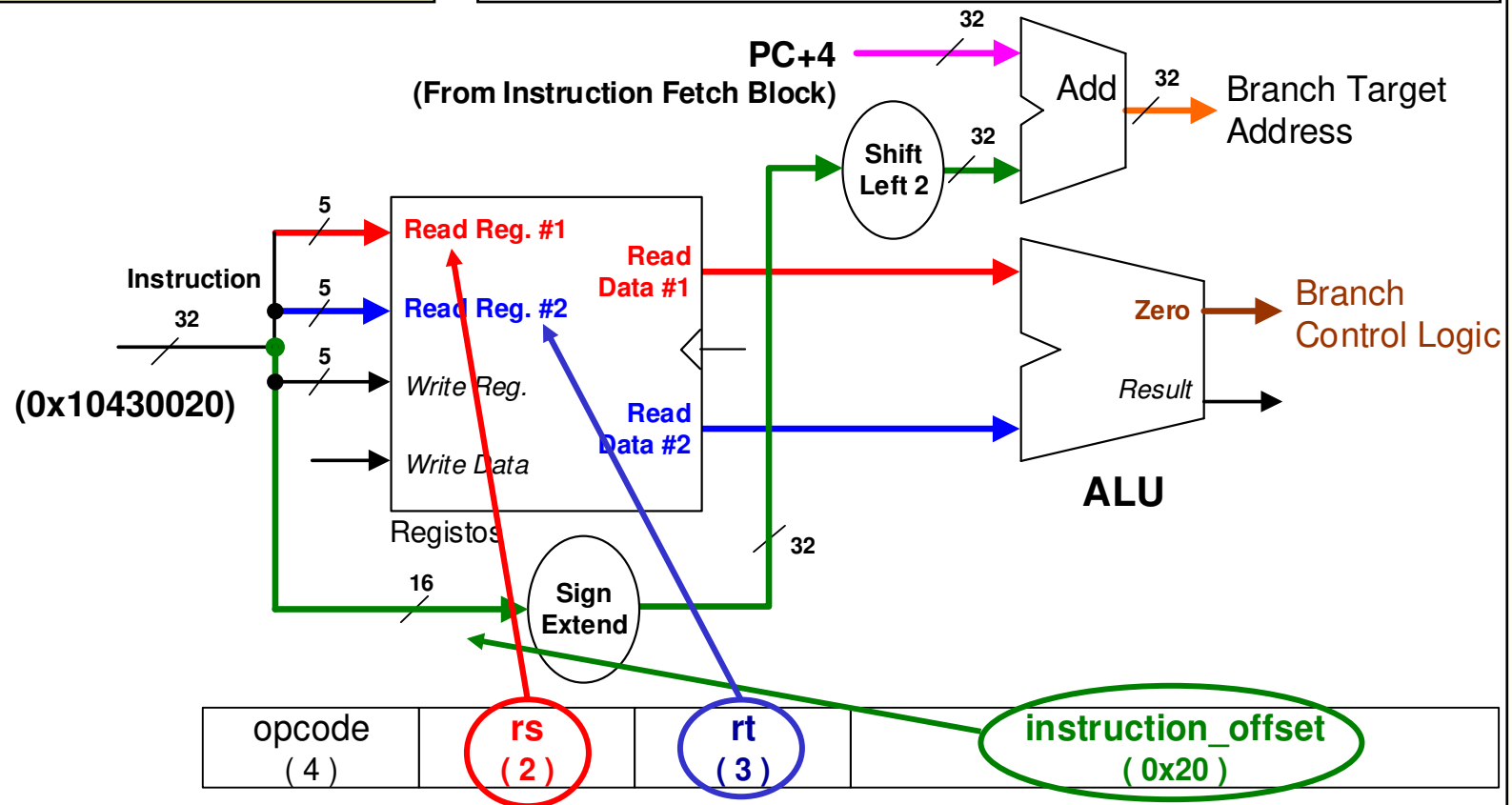


Implementação de um *Datapath* (Instruções de *branch*)

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

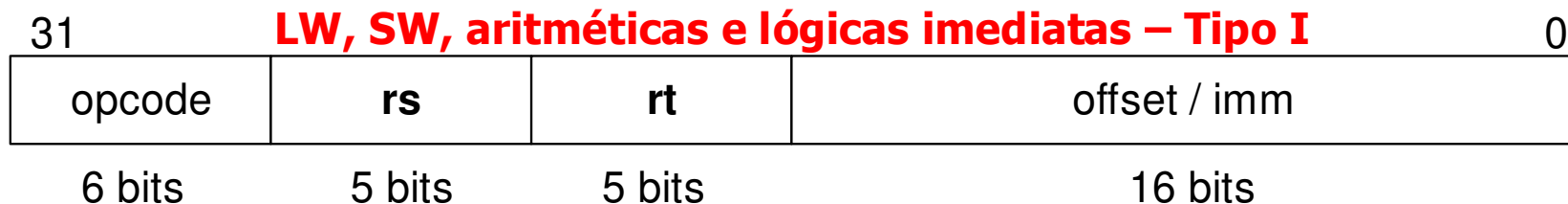
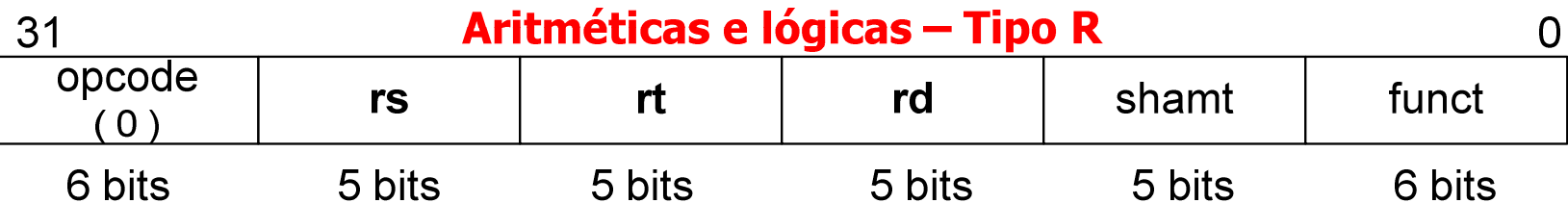
Ex.: **beq** \$2, \$3, 0x20

00010000010000110000000000010000



Implementação de um *Datapath* – juntando tudo

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

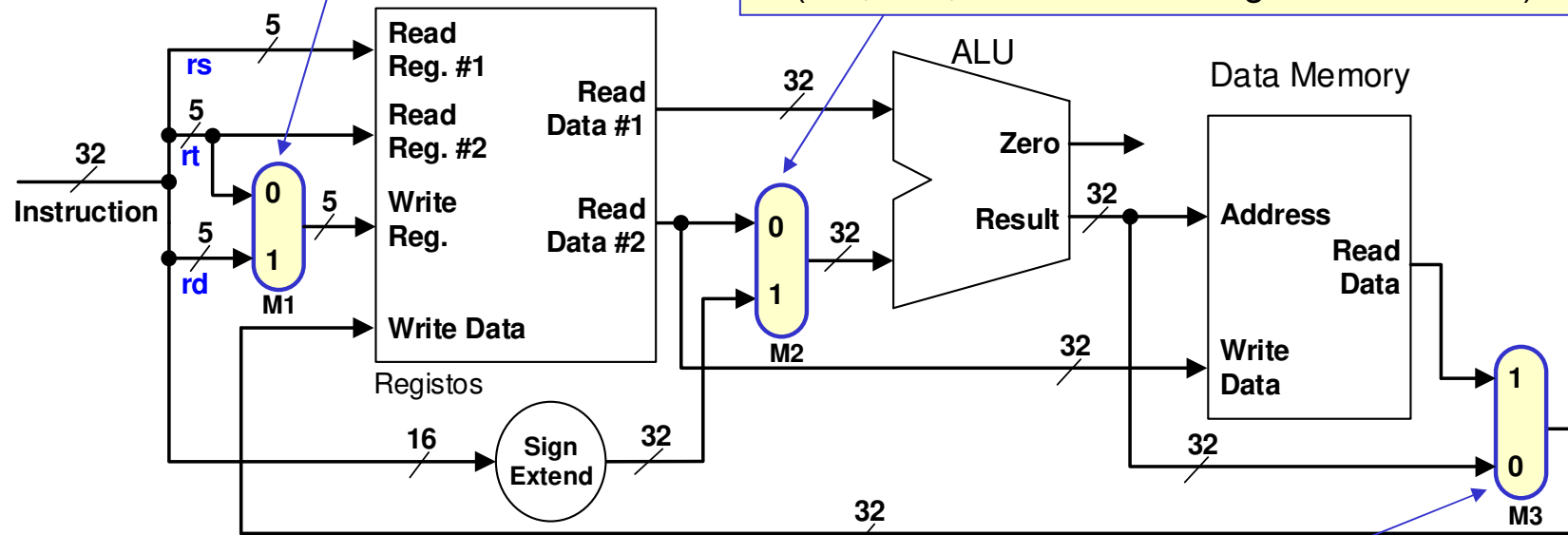


Implementação de um *Datapath* – juntando tudo

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

Seleção do registo destino: **rd** (instruções tipo R), **rt** (LW e nas aritméticas e lógicas imediatas)

Seleção do 2º operando da ALU: **conteúdo de um registo** (instruções tipo R e branches); **offset estendido para 32 bits** (LW, SW, aritméticas e lógicas imediatas)

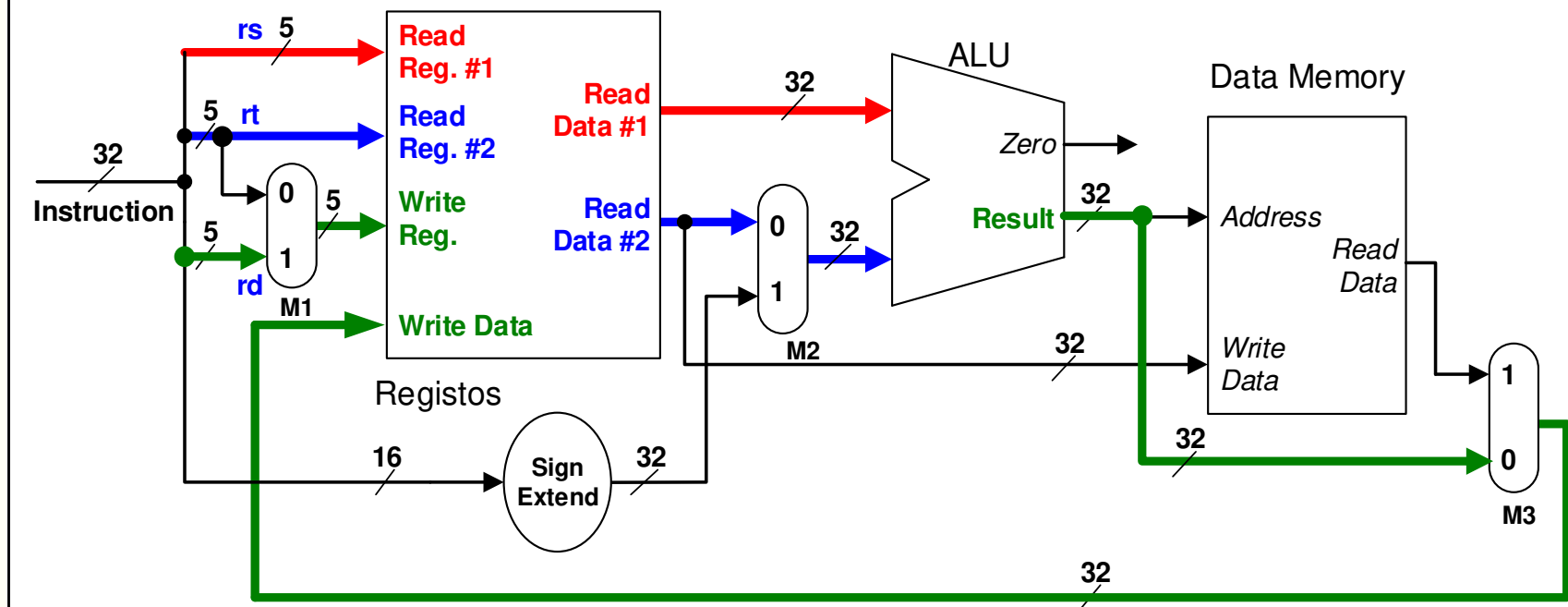


Seleção do valor a escrever no banco de registos: **valor lido da memória** (LW), **valor calculado na ALU** (instruções tipo R, aritméticas e lógicas imediatas)

Implementação de um *Datapath* – juntando tudo

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

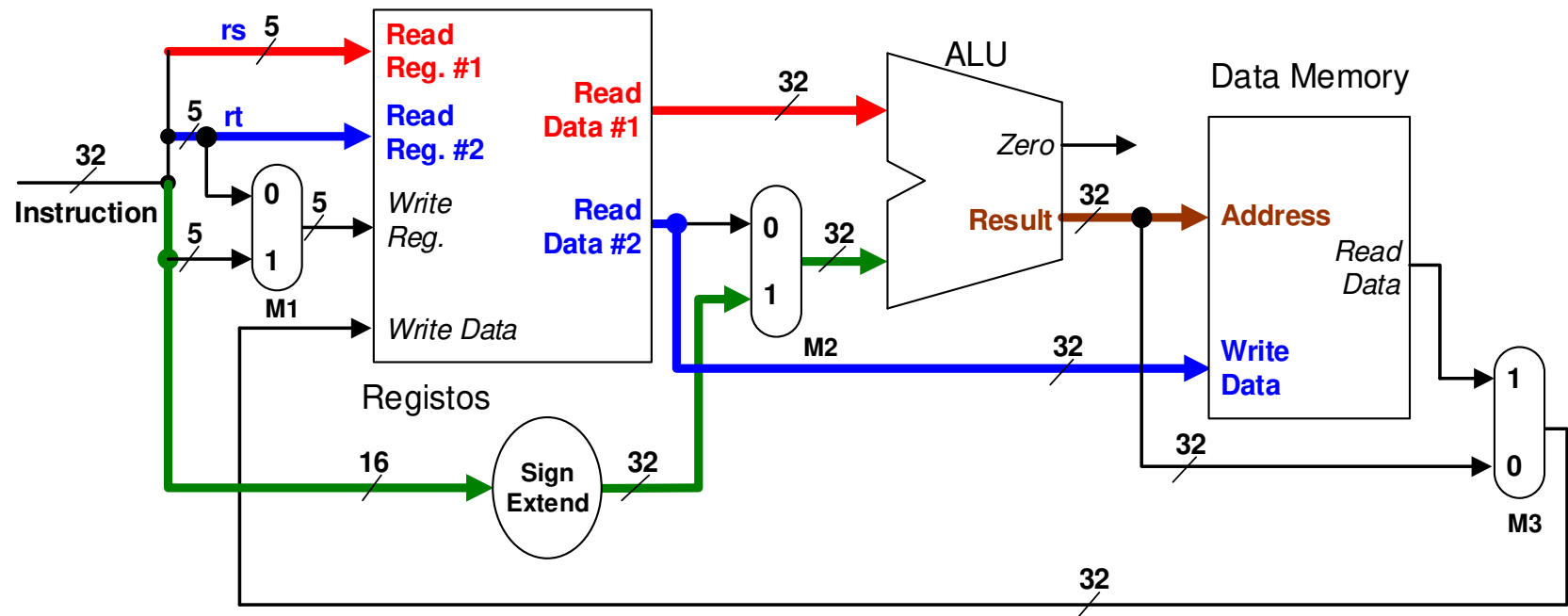
opcode (0)	rs (3)	rt (4)	rd (2)	shamt (0)	funct (32)
-----------------	-------------	-------------	-------------	----------------	-----------------



Implementação de um *Datapath* – juntando tudo

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

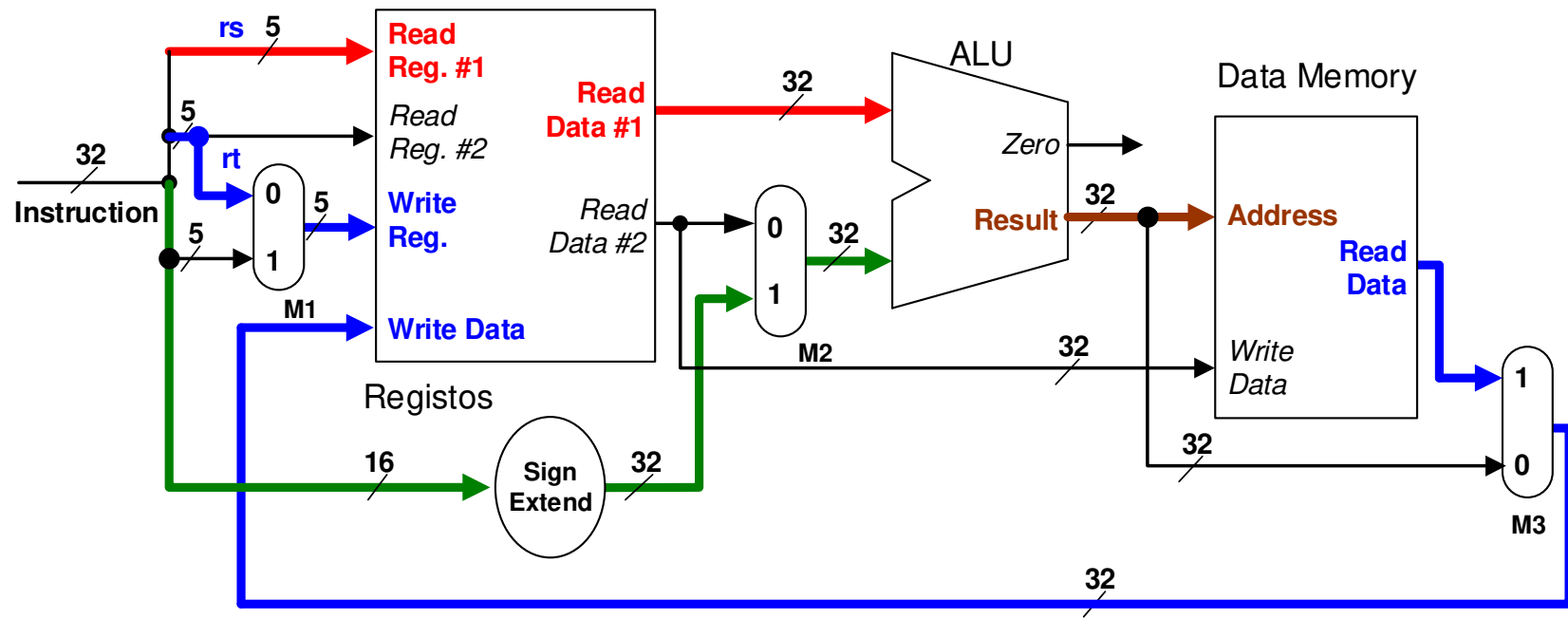
opcode (43)	rs (4)	rt (2)	offset (0x24)
------------------	--------------------	--------------------	---------------------------



Implementação de um *Datapath* – juntando tudo

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

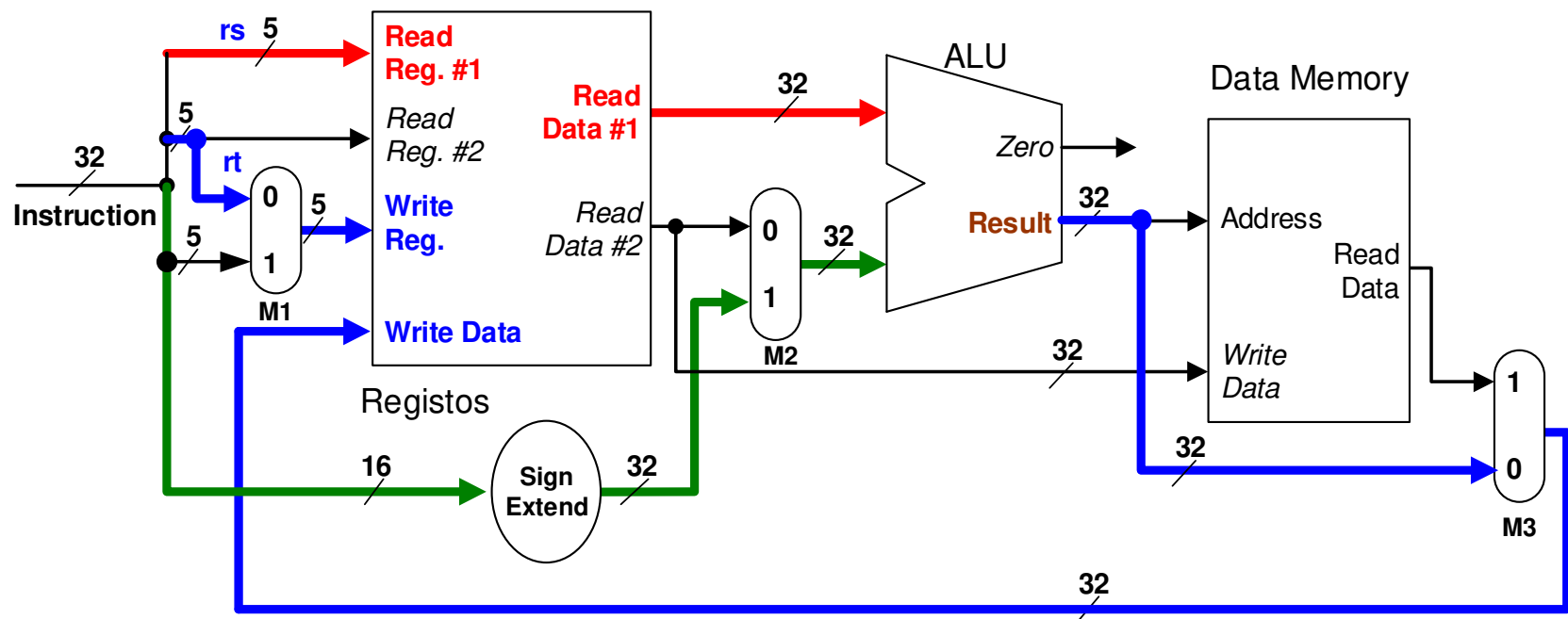
opcode (35)	rs (15)	rt (4)	offset (0x2F)
------------------	---------------------	--------------------	---------------------------



Implementação de um *Datapath* – juntando tudo

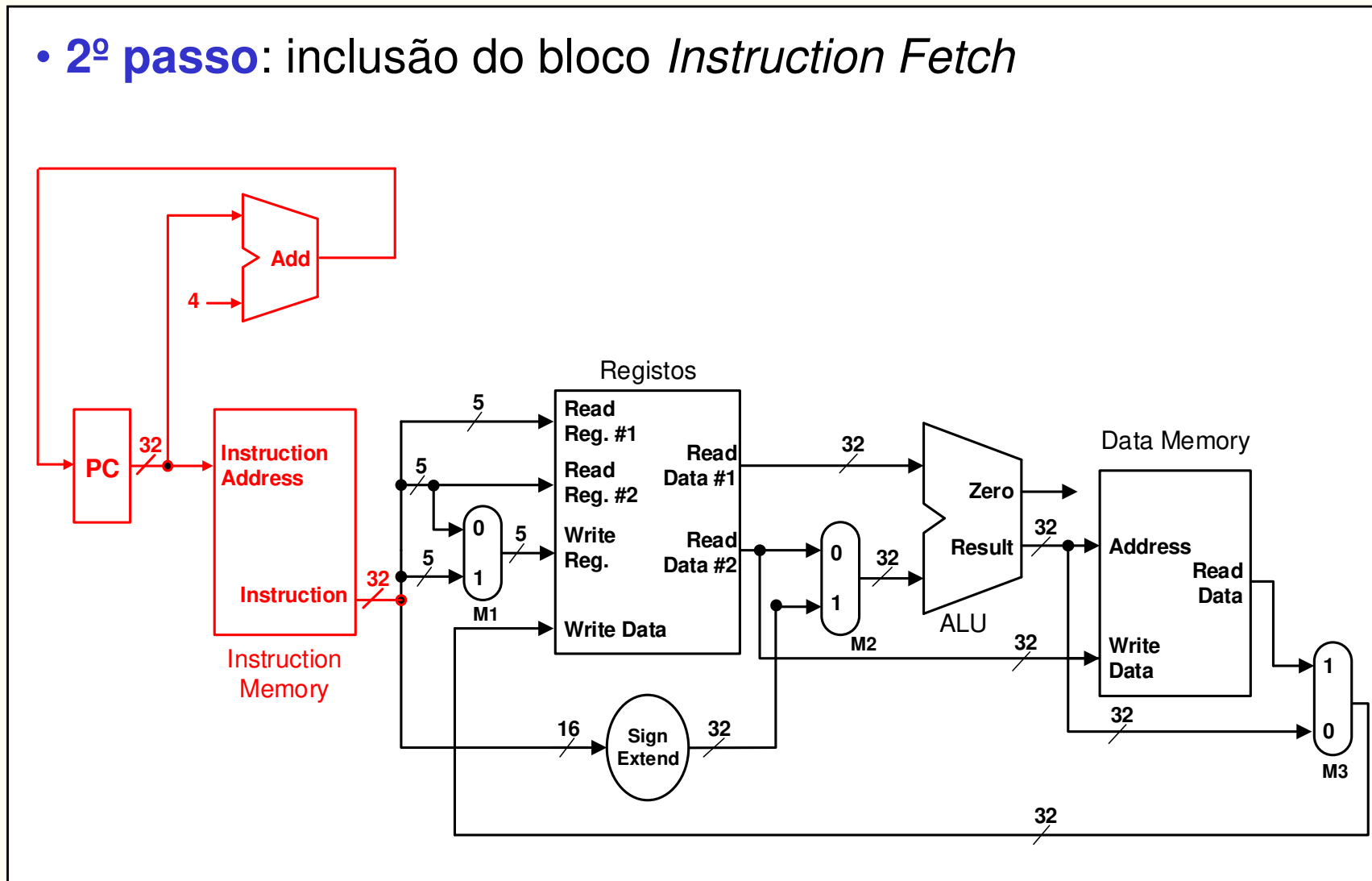
- Fluxo da informação na execução das instruções imediatas.
Exemplo: **addi \$4, \$15, 0x2F**

opcode (8)	rs (15)	rt (4)	offset (0x2F)
-----------------	---------------------	--------------------	---------------------------



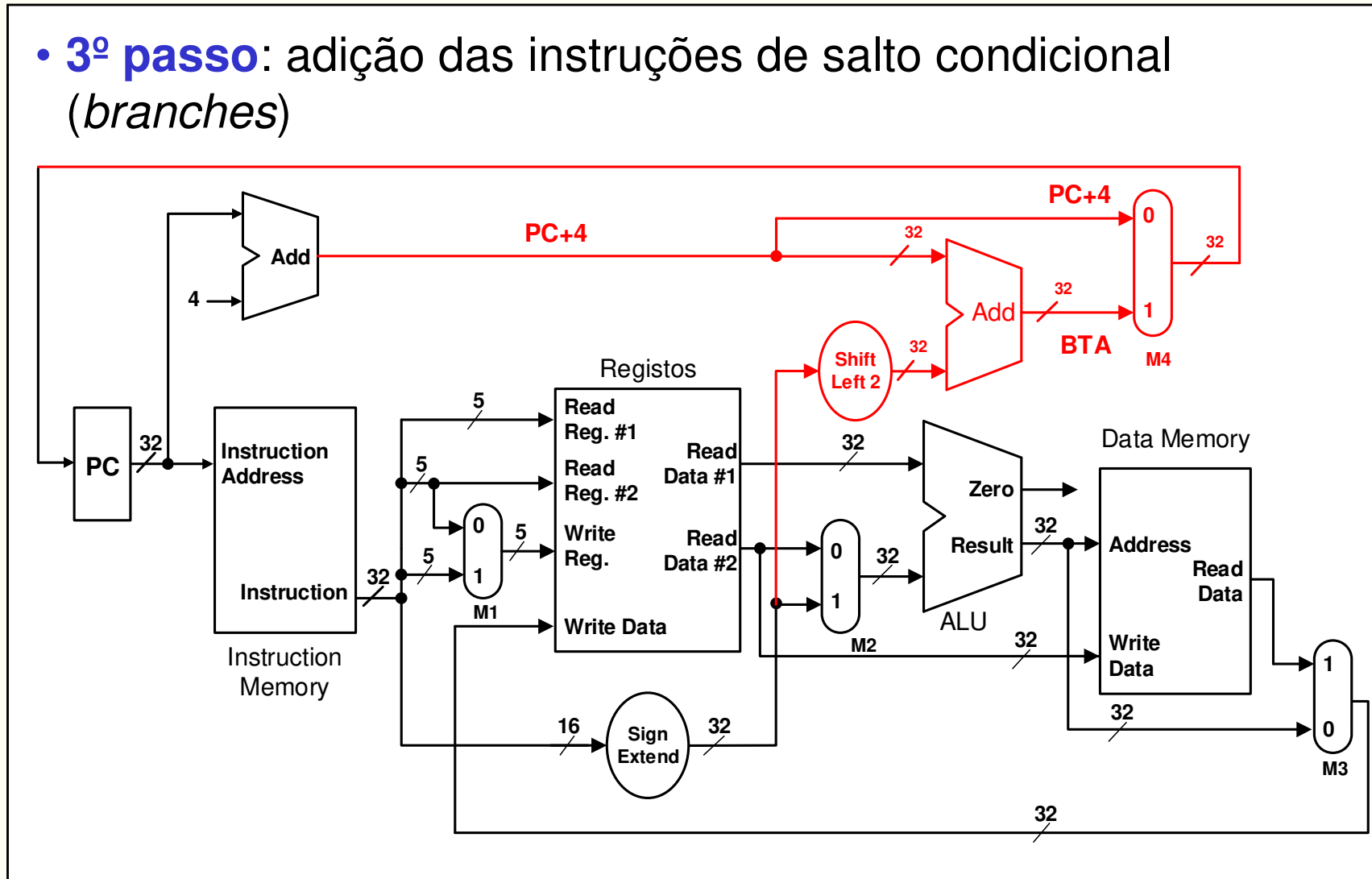
Implementação de um *Datapath* – juntando tudo

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



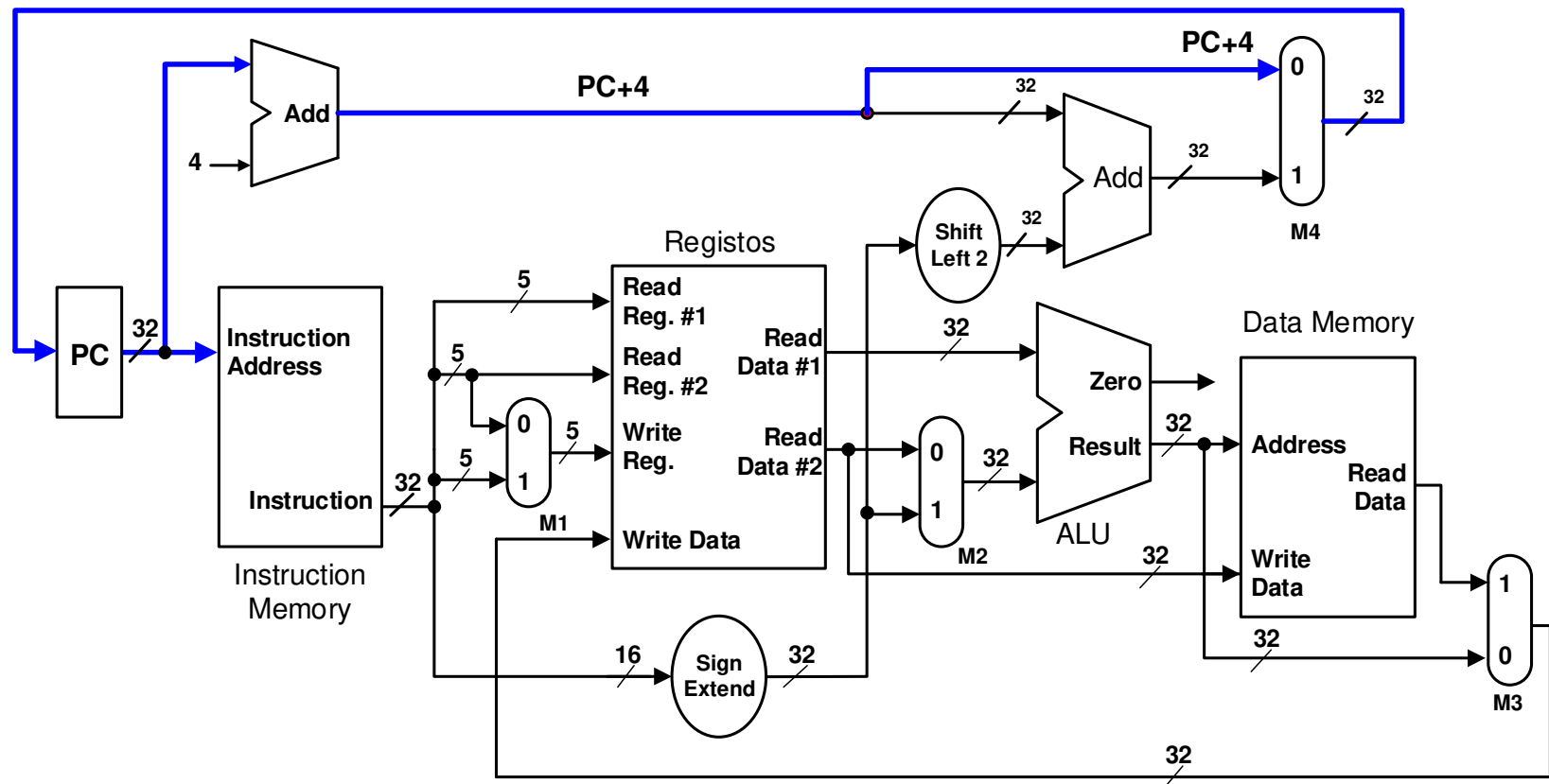
Implementação de um *Datapath* – juntando tudo

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



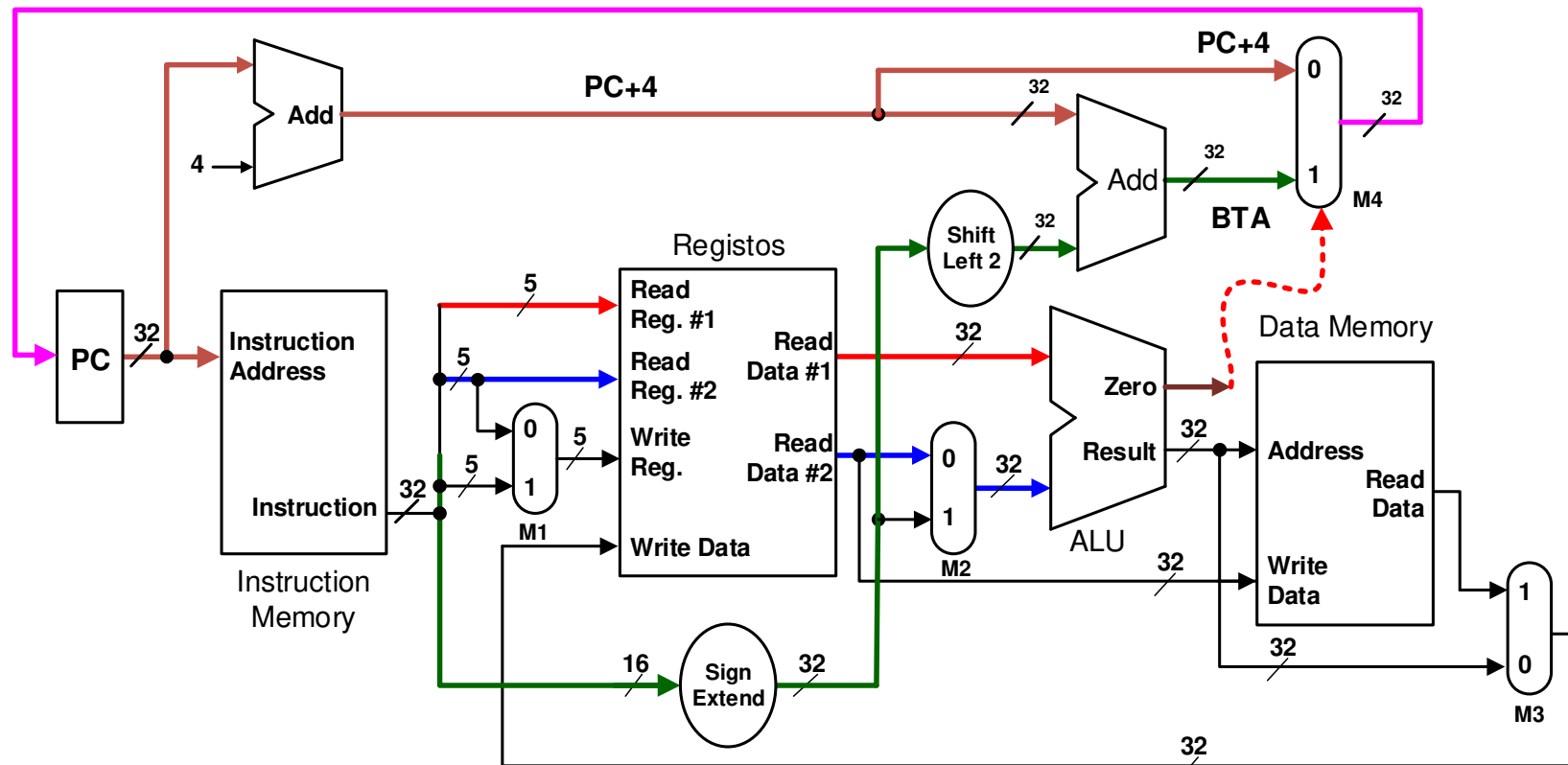
Implementação de um *Datapath* – juntando tudo

- Fluxo da informação durante o *instruction fetch*



Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução de *branch* (**beq**)

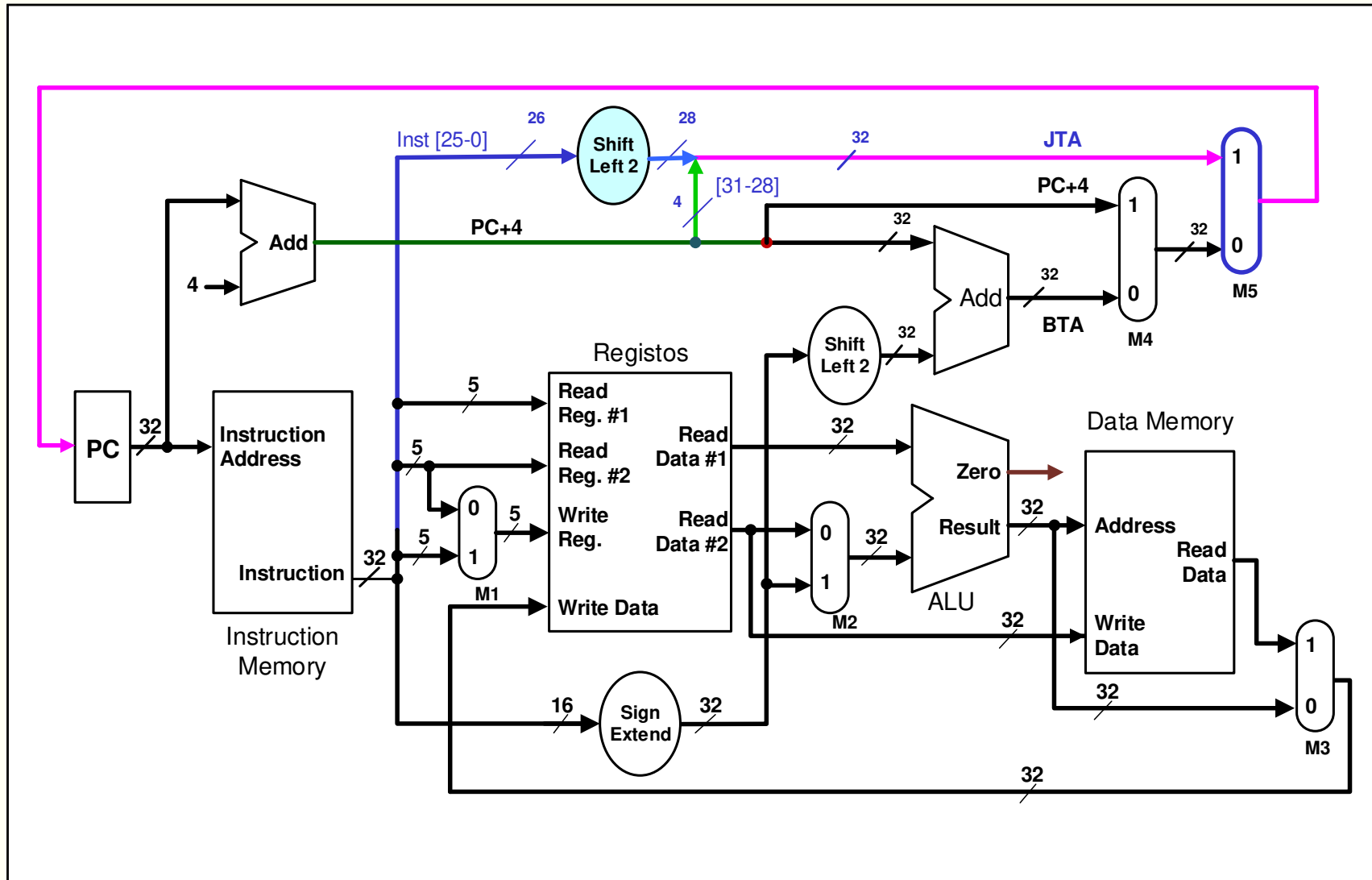


O valor a ser escrito no registo PC, no próximo flanco ativo do relógio, depende da saída "zero" da ALU: **"PC+4" se zero=0; BTA se zero=1**

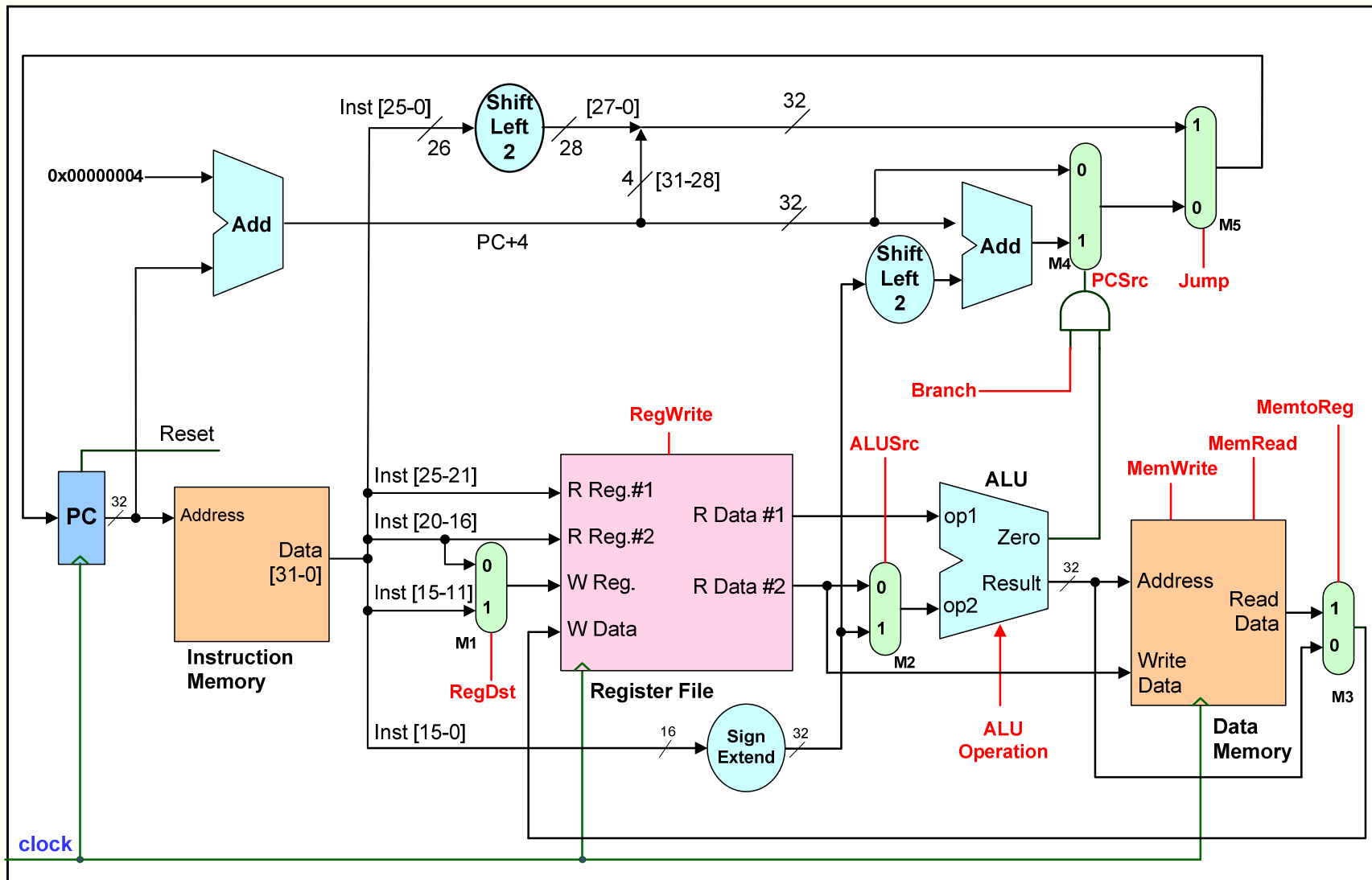
Datapath com suporte para a instrução "j" (jump)

- A instrução “j” é codificada com um caso particular de codificação, o formato J
- No formato J existem apenas dois campos:
 - o campo opcode (**bits 31-26**) e o
 - campo de endereço (**bits 25-0**)
- Na instrução “j”, o endereço alvo (*Jump Target Address - JTA*) 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 (2 *shifts* à esquerda)
- No próximo flanco ativo do relógio, o valor do PC será **incondicionalmente** alterado com o valor do JTA

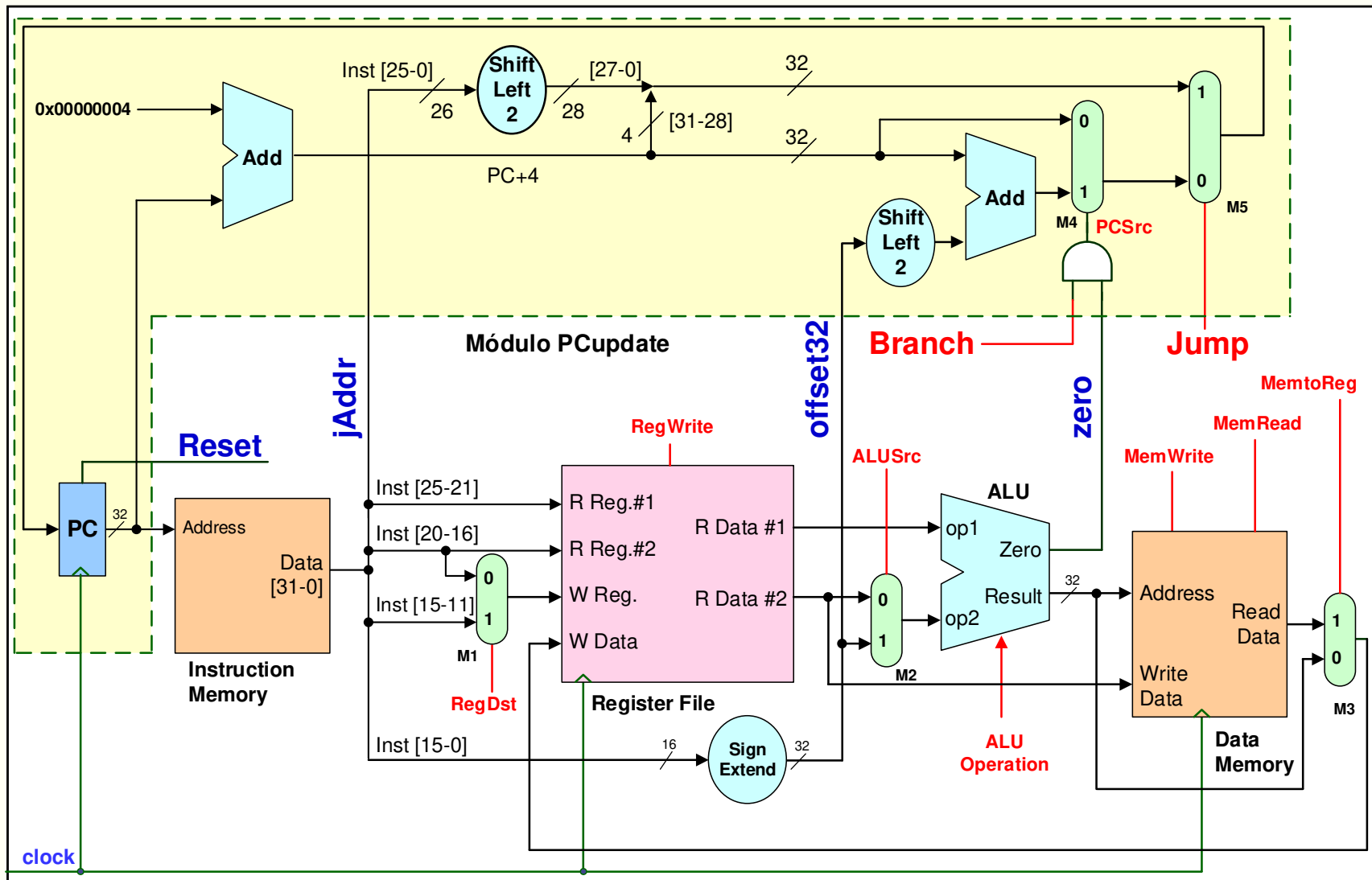
Datapath com suporte para a instrução "jump" (j)



Datapath single-cycle completo (com sinais de controlo)



Módulo de atualização do PC para o DP completo



Módulo de atualização do PC para o DP 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;
        offset32 : 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 para o DP completo

```
architecture Behavioral of PCupdate is
    signal s_pc, s_pc4, s_offset32 : unsigned(31 downto 0);
begin
    s_offset32 <= unsigned(offset32(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_offset32;    -- Branch Target Address
                else
                    s_pc <= s_pc4;
                end if;
            end if;
        end if;
    end process;
    pc <= std_logic_vector(s_pc);
end Behavioral;
```

Exercícios

- Quais as diferenças entre uma arquitetura Harvard e uma arquitetura von Neumann?
- Suponha um sistema baseado numa arquitetura von Neumann, com um barramento de endereços de 20 bits e com uma organização de memória do tipo *byte-addressable*. Qual a dimensão máxima, em bytes, que os programas a executar neste sistema (instruções+dados+stack) podem ter?
- Num processador baseado numa arquitetura Harvard, a memória de instruções está organizada em *words* de 32 bits, a memória de dados em words de 8 bits (*byte-addressable*) e os barramentos de endereços respetivos têm uma dimensão de 24 bits. Qual a dimensão, em bytes, dos espaços de endereçamento de instruções e de dados?
- O que significa um elemento de estado ter escrita síncrona?
- Considere um elemento de estado, com leitura assíncrona, que apenas tem o sinal de *clock*, na sua interface de controlo. O que pode concluir-se relativamente à escrita?

Exercícios

- Suponha um elemento de estado, com escrita síncrona e leitura assíncrona, que apresenta, na sua interface de controlo, um sinal "read", um sinal "write" e um sinal de *clock*. Indique que sinal ou sinais têm que estar ativos para que se realize: a) uma operação de leitura; b) uma operação de escrita.
- Qual a capacidade de armazenamento, expressa em bytes, de uma memória com uma organização interna em *words* de 32 bits e um barramento de endereços de 30 bits?
- Quais as operações realizadas no *datapath* que são comuns a todas as instruções?
- Identifique a operação realizada na ALU na realização de cada uma das seguintes instruções: tipo R, *addi*, *slti*, *lw*, *sw* e *beq*.
- Indique qual a operação realizada na conclusão de cada uma das seguintes instruções: tipo R, *addi*, *slti*, *lw*, *sw*, *beq* e *j*.
- Suponha que o *datapath* está a executar a instrução **add \$3, \$4, \$5**. Que operações serão realizadas na próxima transição ativa do sinal de relógio?
- No *datapath single-cycle* que tipo de informação é armazenada na memória cujo endereço é a saída do registo PC?

Exercícios

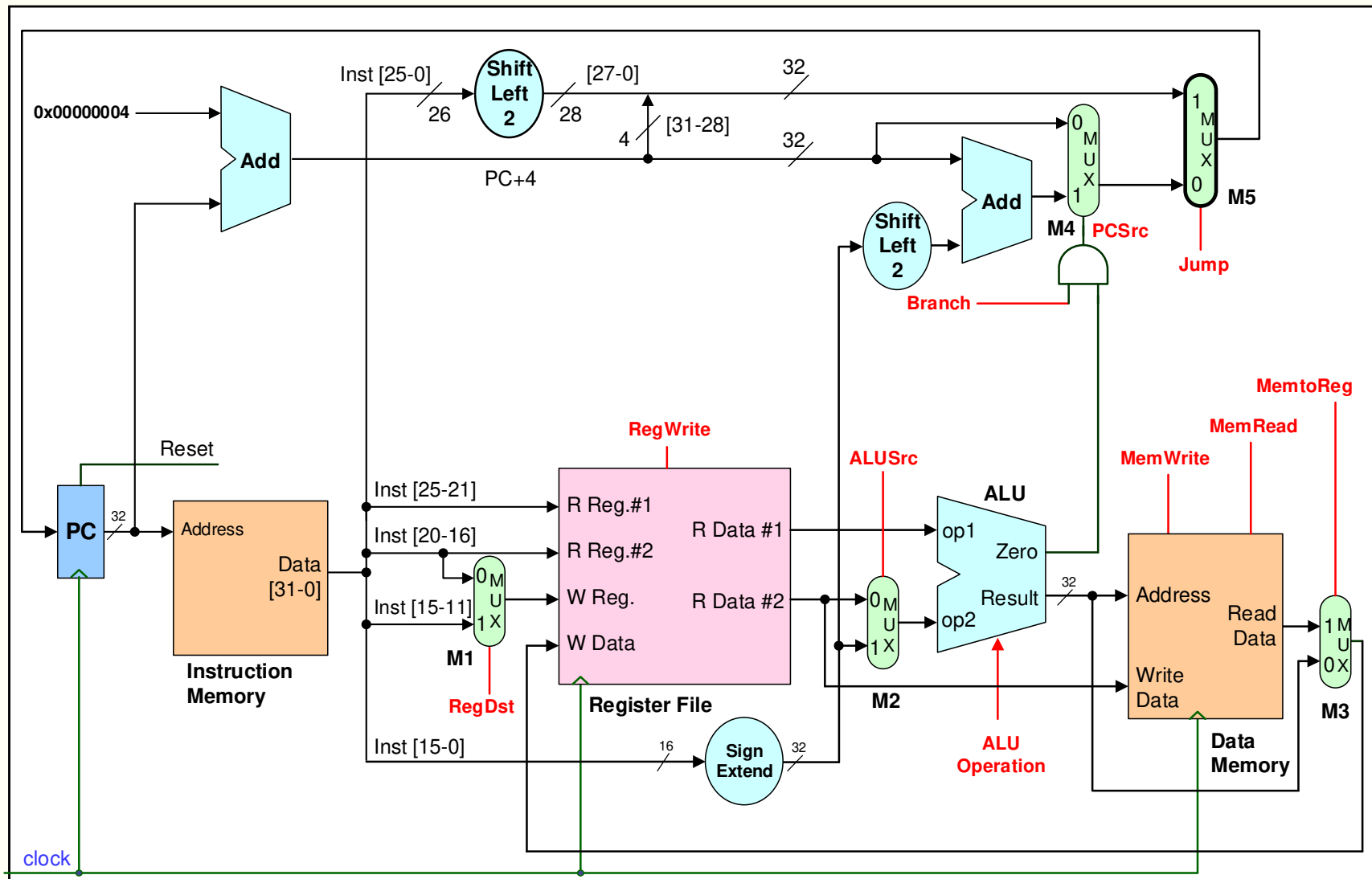
- Qual o endereço de memória onde deve estar armazenada a primeira instrução do programa para que a execução possa ser reiniciada sempre que se ative o sinal de "reset" do registo PC?
- Suponha que cada registo do banco de registos foi inicializado com um valor igual a: (32-número do registo). Indique o valor presente nas entradas do banco de registos **ReadReg1**, **ReadReg2** e **WriteReg**, e o valor presente nas saídas **ReadData1** e **ReadData2**, durante a execução das instruções com o código máquina: **0x00CA9820**, **0x8D260018** (**lw**) e **0xAC6A003C** (**sw**).
- Considerando ainda a inicialização do banco de registos da questão anterior, indique qual o valor calculado pela ALU durante a execução das instruções **LW** com o código máquina **0x8CA40005** e **0x8CE6FFF3**.
- Qual o valor à saída do somador de cálculo do BTA durante a execução da instrução cujo código máquina é **0x10430023**, supondo que o valor à saída do registo PC é **0x00400034**?

Aulas 17 e 18

- A unidade de controlo principal do *datapath single-cycle*
- A unidade de controlo da ALU
- Implementação das unidades de controlo do *datapath* e da ALU
- Exemplos de funcionamento do *datapath* com unidade de controlo

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Datapath single-cycle completo



Unidade de controlo

- A unidade de controlo deve gerar os sinais (identificados a vermelho) para:
 - 1) controlar a escrita e/ou a leitura em elementos de estado: **banco de registos** e **memória de dados**
 - 2) definir a operação dos elementos combinatórios: **ALU** e **multiplexers**
- A operação na ALU é definida com 3 bits (ALU Control):

ALU operation	ALU Control
AND	000
OR	001
ADD	010
SUB	110
SLT	111

Unidade de controlo

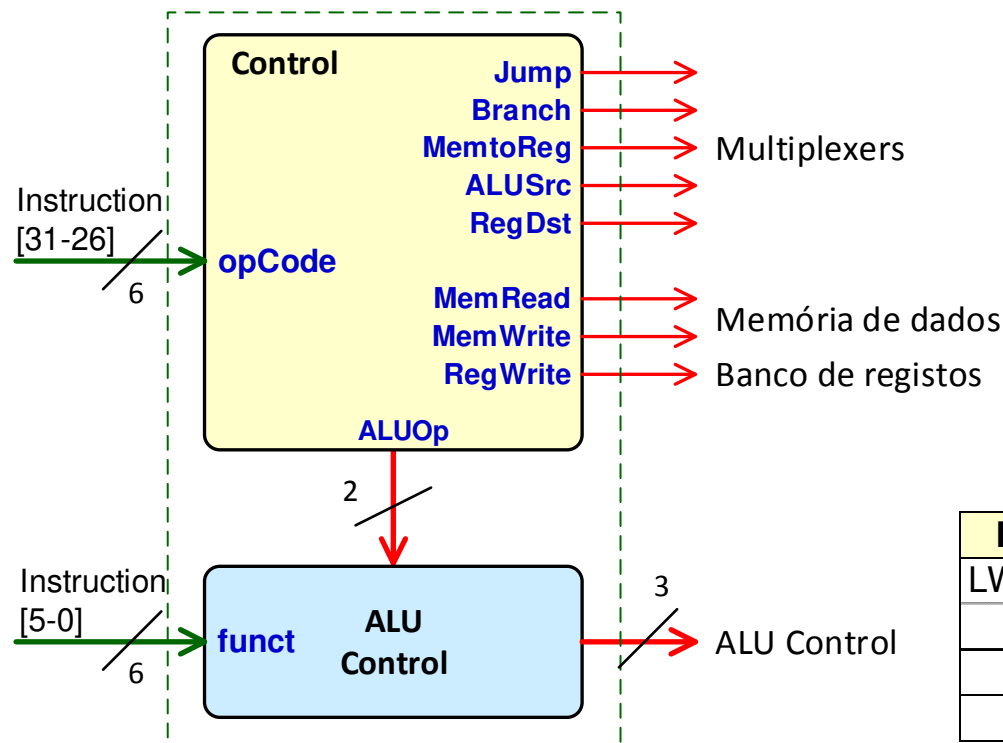
- Alguns dos elementos de estado do *datapath* são acedidos em todos os ciclos de relógio (PC e memória de instruções)
 - Nestes casos não há necessidade de explicitar um sinal de controlo
- Outros elementos de estado podem ser lidos ou escritos dependendo da instrução que estiver a ser executada (memória de dados e banco de registos)
 - Para estes é necessário explicitar os respetivos sinais de controlo
- Nos elementos de estado:
 - a **escrita** é sempre realizada de forma síncrona
 - a **leitura** é sempre realizada de forma assíncrona

Unidade de controlo

- Todas as instruções (exceto o "j") usam a ALU:
 - **LW e SW** – para calcular o endereço da memória externa (soma)
 - **Branch if equal / not equal** – para determinar se os operandos são iguais ou diferentes (subtração)
 - **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 (opcode=0):
 $ALUControl = f(opcode, funct)$
 - do campo **opcode** nas restantes instruções:
 $ALUControl = f(opcode)$

Unidade de controlo

- A unidade de controlo pode ser sub-dividida em duas: 1) controlo de multiplexers e elementos de estado; 2) controlo da ALU

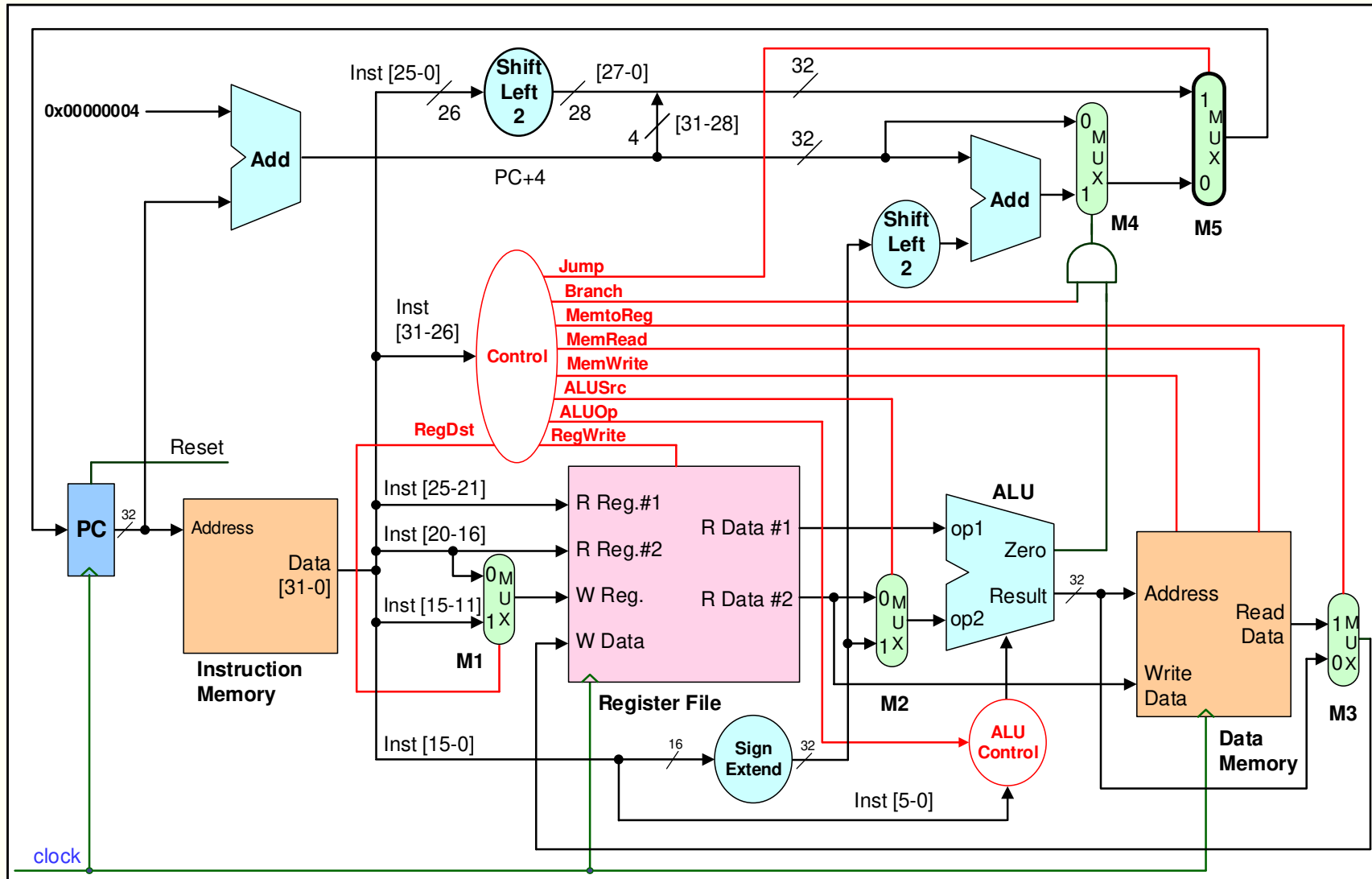


ALU Control	ALU operation
000	AND
001	OR
010	ADD
110	SUB
111	SLT

Instruction	ALUOp	ALU operation
LW, SW, ADDI	00	ADD
BEQ	01	SUB
R-Type	10	Depends on "funct"
SLTI	11	SLT

- A operação da ALU é definida em conjunto com a unidade de controlo principal, em função dos campos "opcode" e "funct"

Datapath single-cycle com unidade de controlo



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 tabela seguinte

ALU Control	ALU operation
000	AND
001	OR
010	ADD
110	SUB
111	SLT

Instruction	opcode	funct	ALU Operation	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
jump	000010 ("j")	xxxxxx	-	xx	xxx

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"; -- ADD
      when "01" => -- BEQ
        ALUcontrol <= "110"; -- SUB
      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 <= "---";
        end case;
      when "11" => -- SLTI
        ALUcontrol <= "111";
    end case;
  end process;
end Behavioral;

```

ALU Control	ALU operation
0 0 0	AND
0 0 1	OR
0 1 0	ADD
1 1 0	SUB
1 1 1	SLT

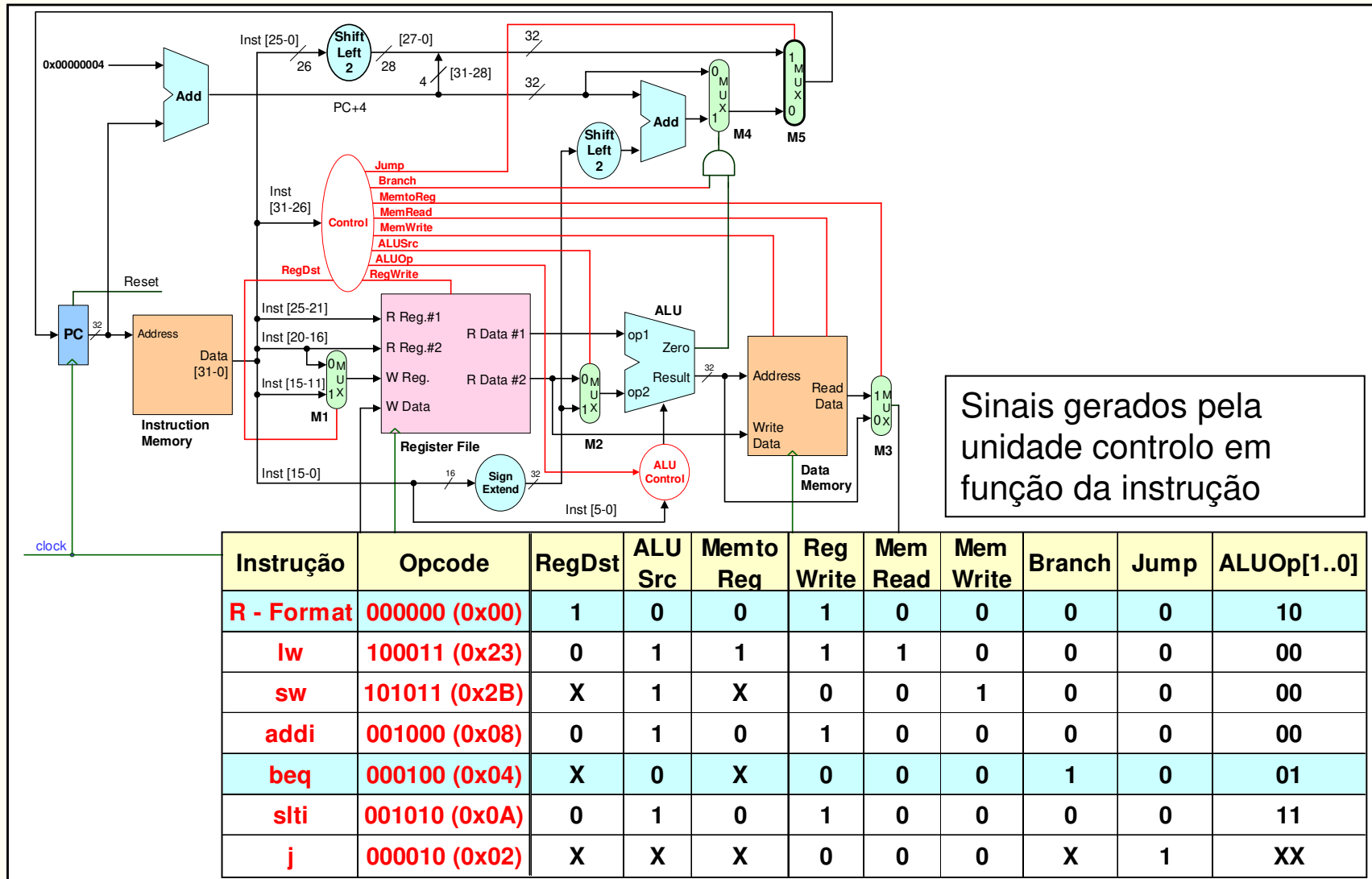
Instruction	ALUOp	ALU operation
LW, SW, ADDI	00	ADD
BEQ	01	SUB
R-Type	10	Depends on "funct"
SLTI	11	SLT

Unidade de controlo principal

- É necessário especificar um total de oito sinais de controlo (para além do ALUOp):

Sinal	Efeito quando não ativo ('0')	Efeito quando ativo ('1')
MemRead	Nenhum (barramento de dados da memória em alta impedância)	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
RegWrite	Nenhum	O registo indicado no endereço de escrita é alterado pelo valor presente na entrada de dados
RegDst	O endereço do registo destino provém do campo "rt"	O endereço do registo destino provém do campo "rd"
ALUSrc	O segundo operando da ALU provém da segunda saída do <i>Register File</i>	O segundo operando da ALU provém dos 16 bits menos significativos da instrução após extensão do sinal
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
Branch	Nenhum	Indica que a instrução é um branch condicional
PCSrc	O PC é substituído pelo seu valor actual mais 4	O PC é substituído pelo resultado do somador que calcula o endereço alvo do <i>branch</i> condicional
Jump	Nenhum	Indica que a instrução é um <i>jump</i> incondicional

Unidade de controlo principal



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;
        Jump        : 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'; Jump  <= '0';
    ALUOp    <= "00";
    case OpCode is
      when "000000" => -- R-Type instructions
        ALUOp  <= "10"; RegDst <= '1'; RegWrite <= '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 "000100" => -- BEQ
        ALUOp  <= "01"; Branch <= '1';
      when "001010" => -- SLTI
        ALUOp  <= "11"; ALUSrc <= '1'; RegWrite <= '1';
      when "000010" => -- J
        Jump    <= '1';
      when others =>
    end case;
  end process;
end Behavioral;

```

Instrução	Opcode	RegDst	ALU Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALUOp[1..0]
R - Format	000000 (0x00)	1	0	0	1	0	0	0	0	10
lw	100011 (0x23)	0	1	1	1	1	0	0	0	00
sw	101011 (0x2B)	X	1	X	0	0	1	0	0	00
addi	001000 (0x08)	0	1	0	1	0	0	0	0	00
beq	000100 (0x04)	X	0	X	0	0	0	1	0	01
slti	001010 (0x0A)	0	1	0	1	0	0	0	0	11
j	000010 (0x02)	X	X	X	0	0	0	X	1	XX

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, considerar que a utilização dos vários elementos operativos ocorre em sequência e decorre ao longo de um conjunto de operações
- A sequência de operações culmina 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

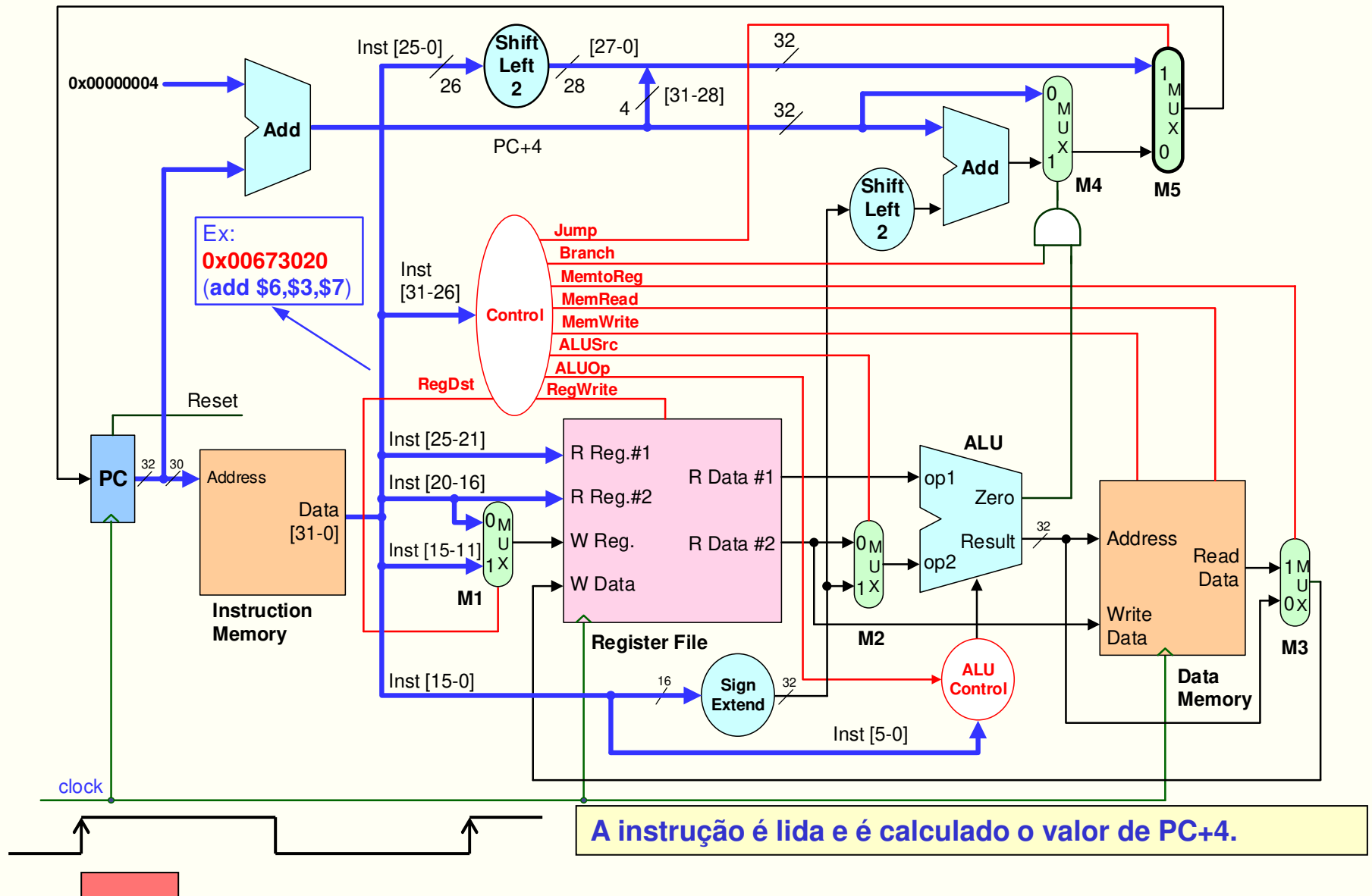
Análise do funcionamento do *datapath* – operações

- *Fetch* de uma instrução e cálculo do endereço da próxima instrução
- Leitura de dois registros do Banco de Registos
- A ALU opera sobre dois valores (a origem do segundo operando 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**): BTA ou PC+4

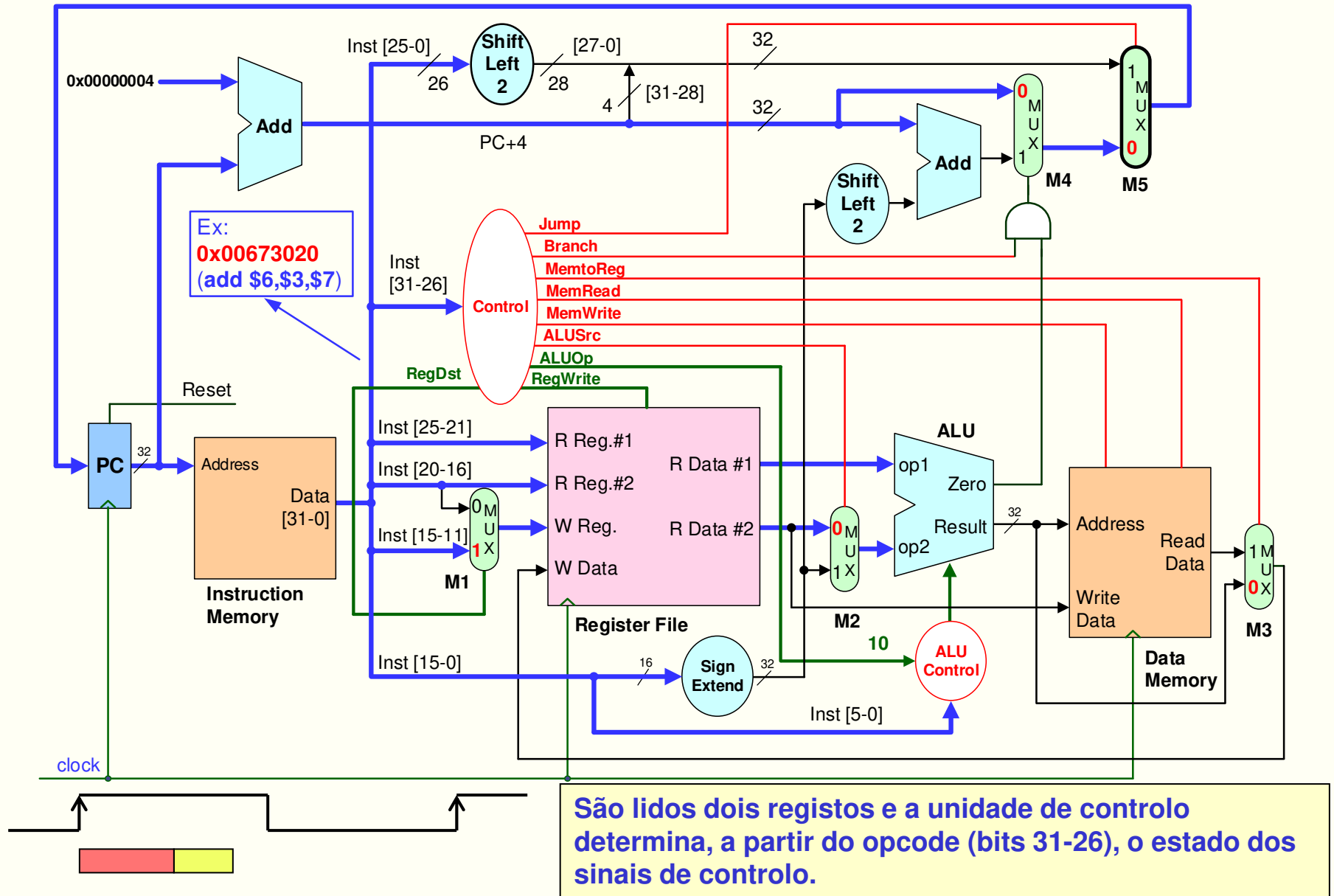
Funcionamento do *datapath* nas instruções tipo R

- 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 no campo *funct* (**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

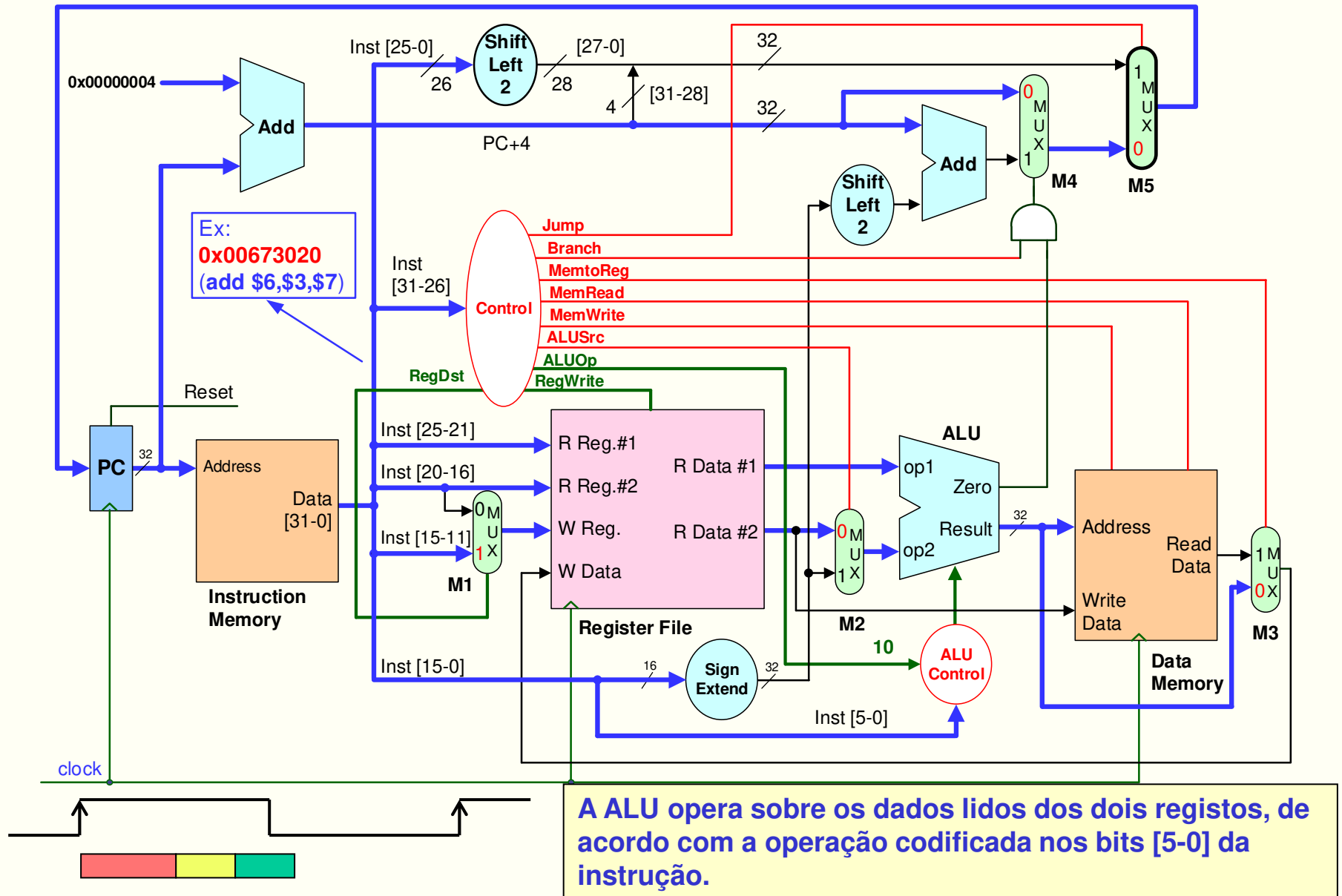
Funcionamento do *datapath* nas instruções tipo R (1)



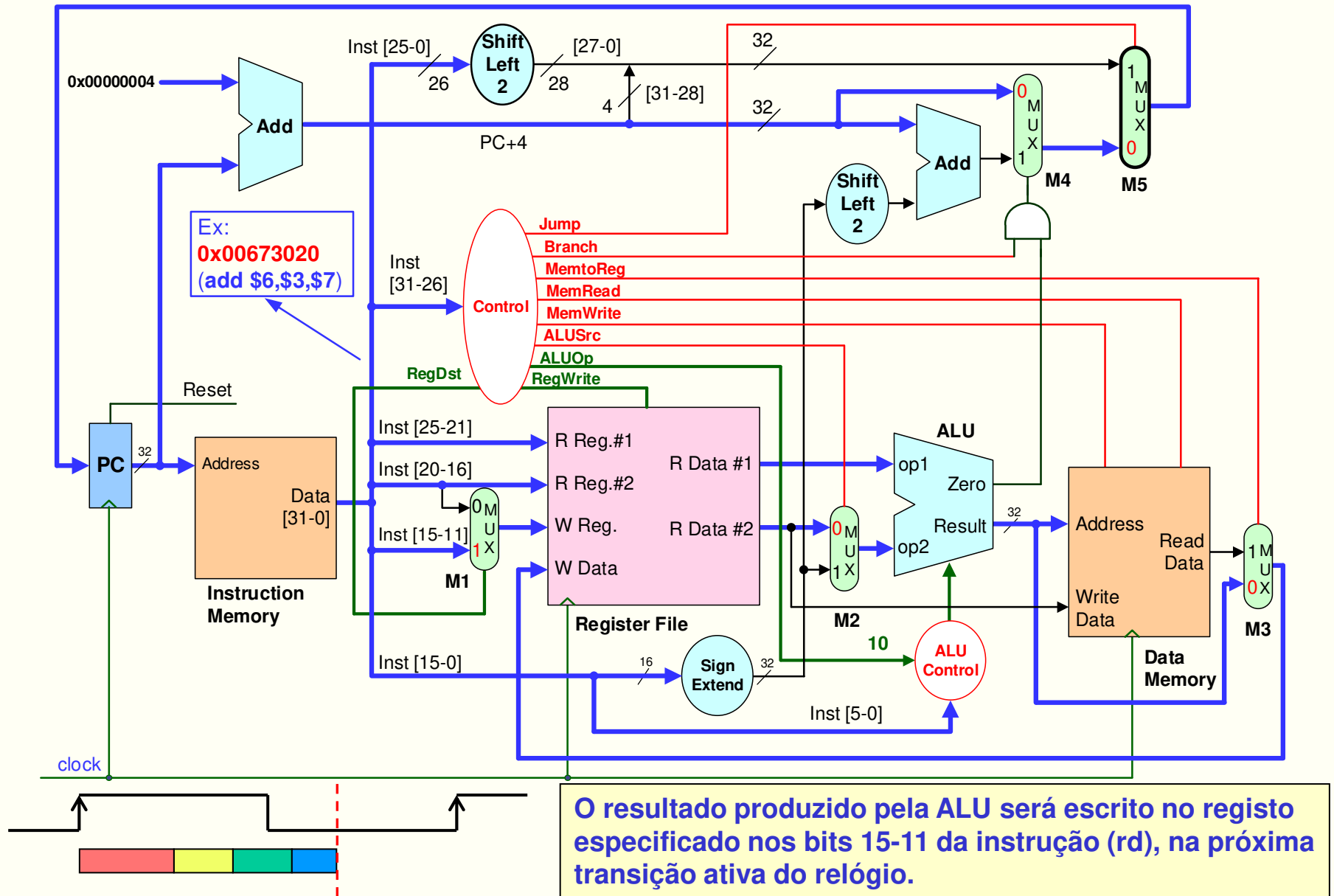
Funcionamento do *datapath* nas instruções tipo R (2)



Funcionamento do *datapath* nas instruções tipo R (3)



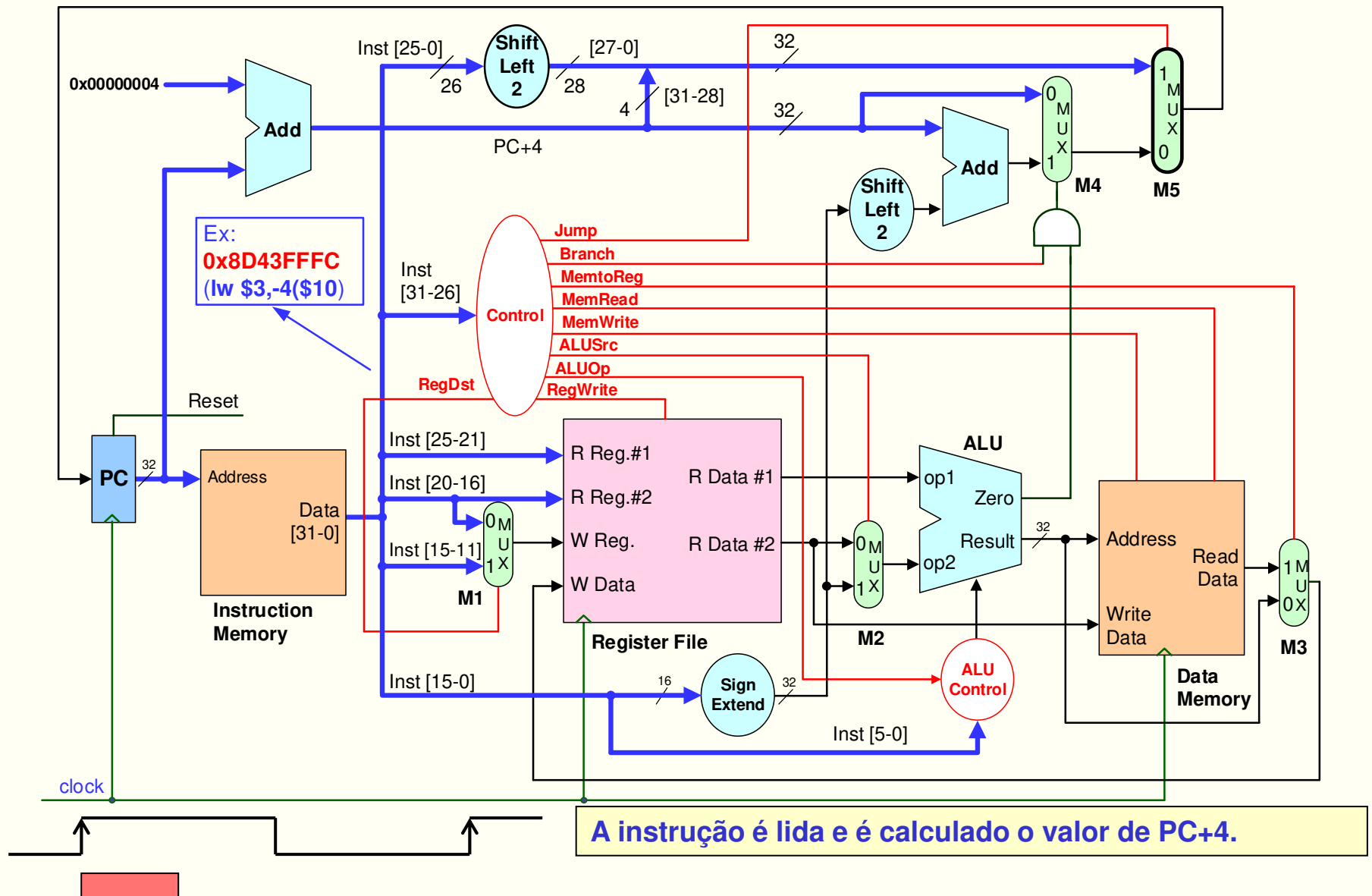
Funcionamento do *datapath* nas instruções tipo R (4)



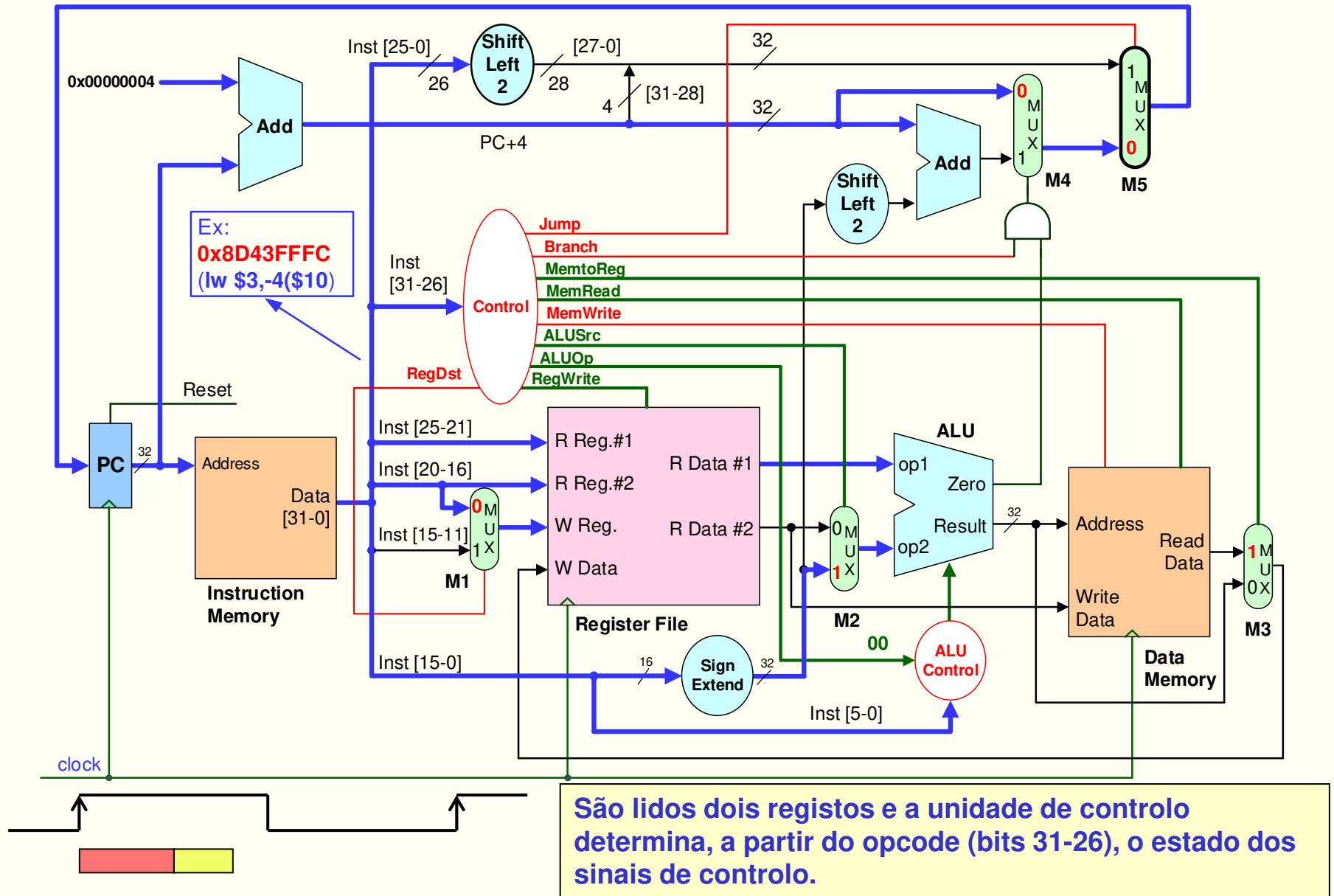
Funcionamento do *datapath* na instrução LW

- 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 **bits 25-21** ("**rs**") com os 16 bits (estendidos com sinal para 32) do campo *offset* da instrução (**bits 15-0**).
- O resultado produzido pela ALU constitui o endereço de acesso à memória de dados. A memória é lida nesse endereço (leitura assíncrona).
- A *word* lida da memória será escrita no registo especificado nos **bits 20-16** da instrução ("**rt**"), na próxima transição ativa do relógio.

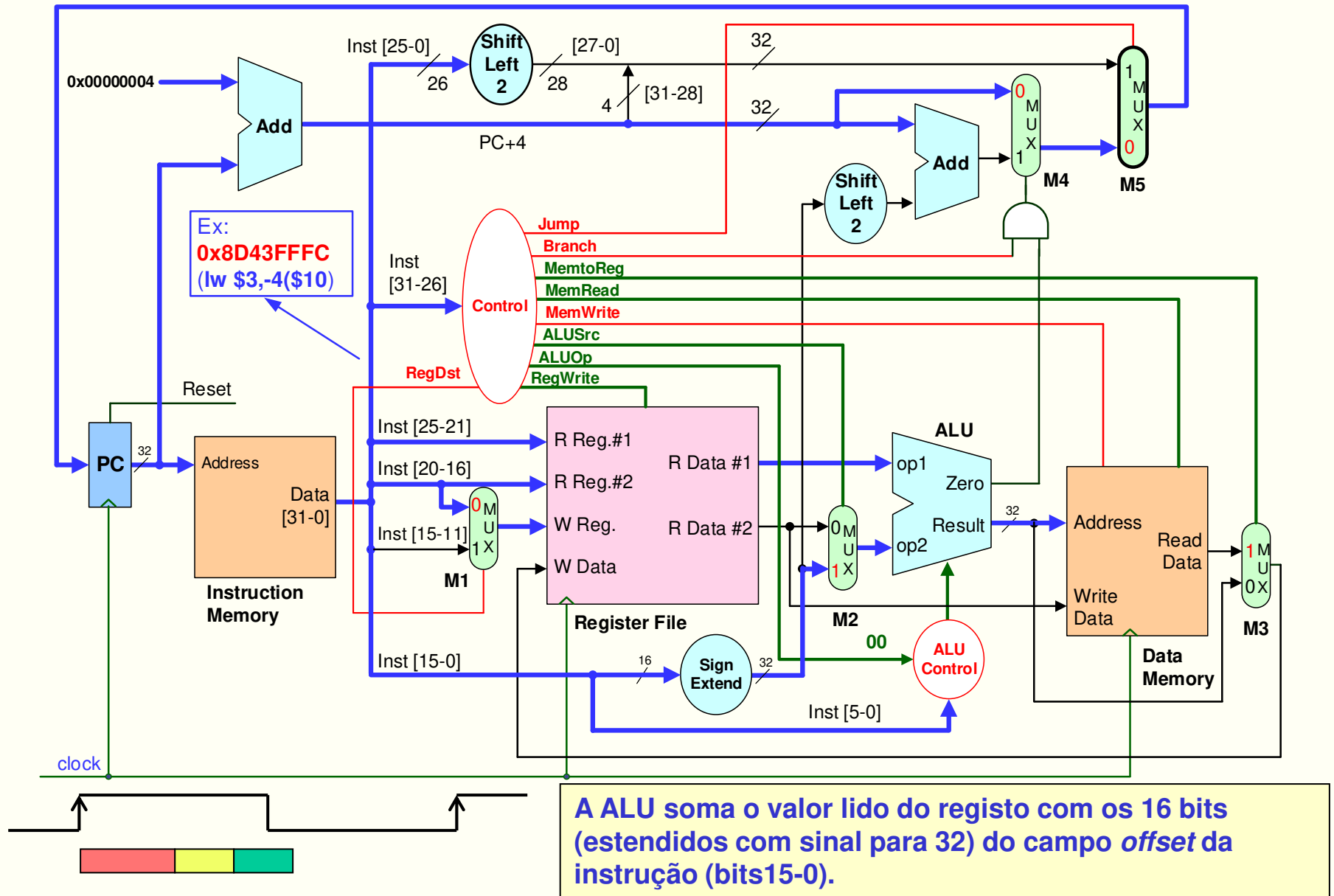
Funcionamento do *datapath* na instrução LW (1)



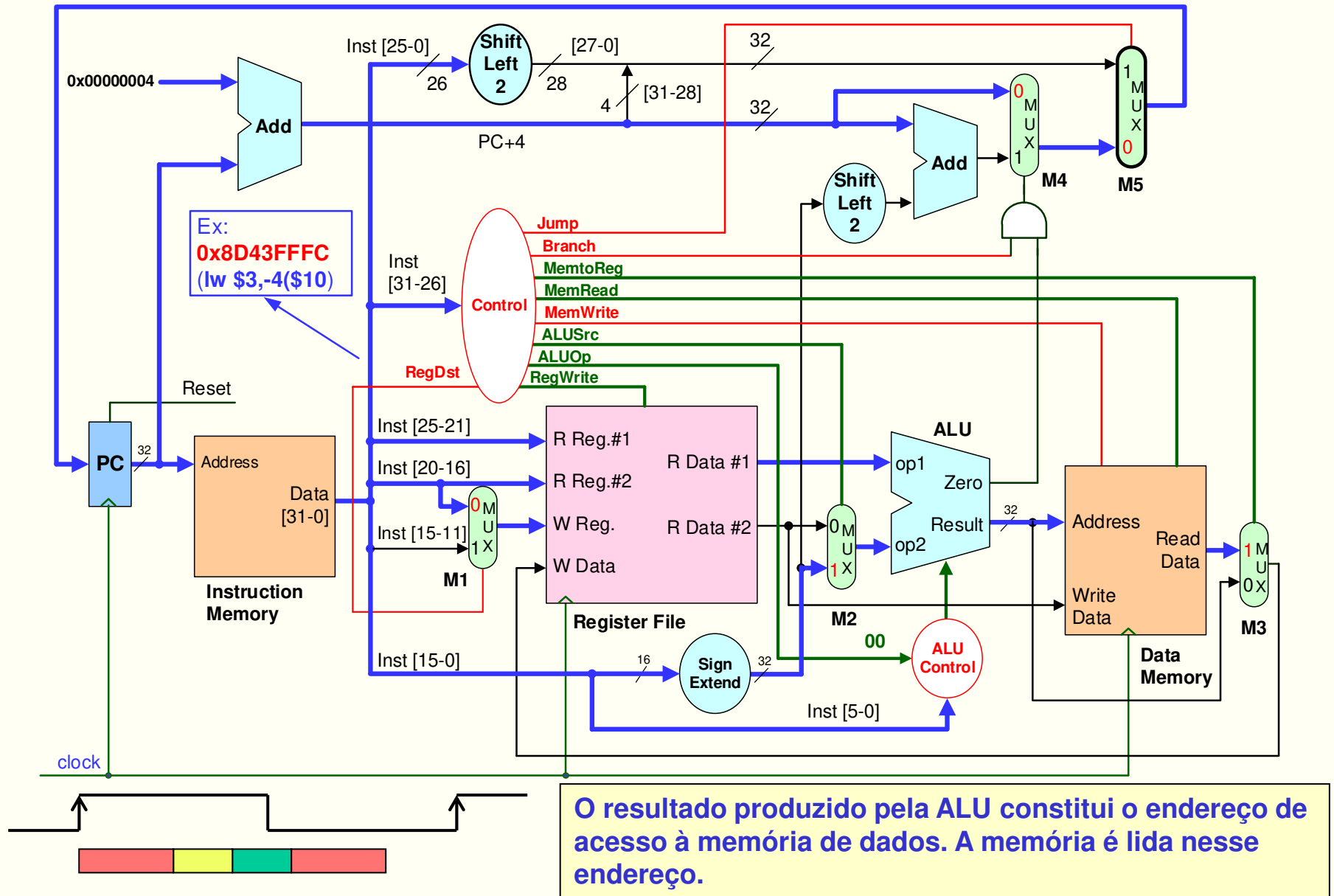
Funcionamento do *datapath* na instrução LW (2)



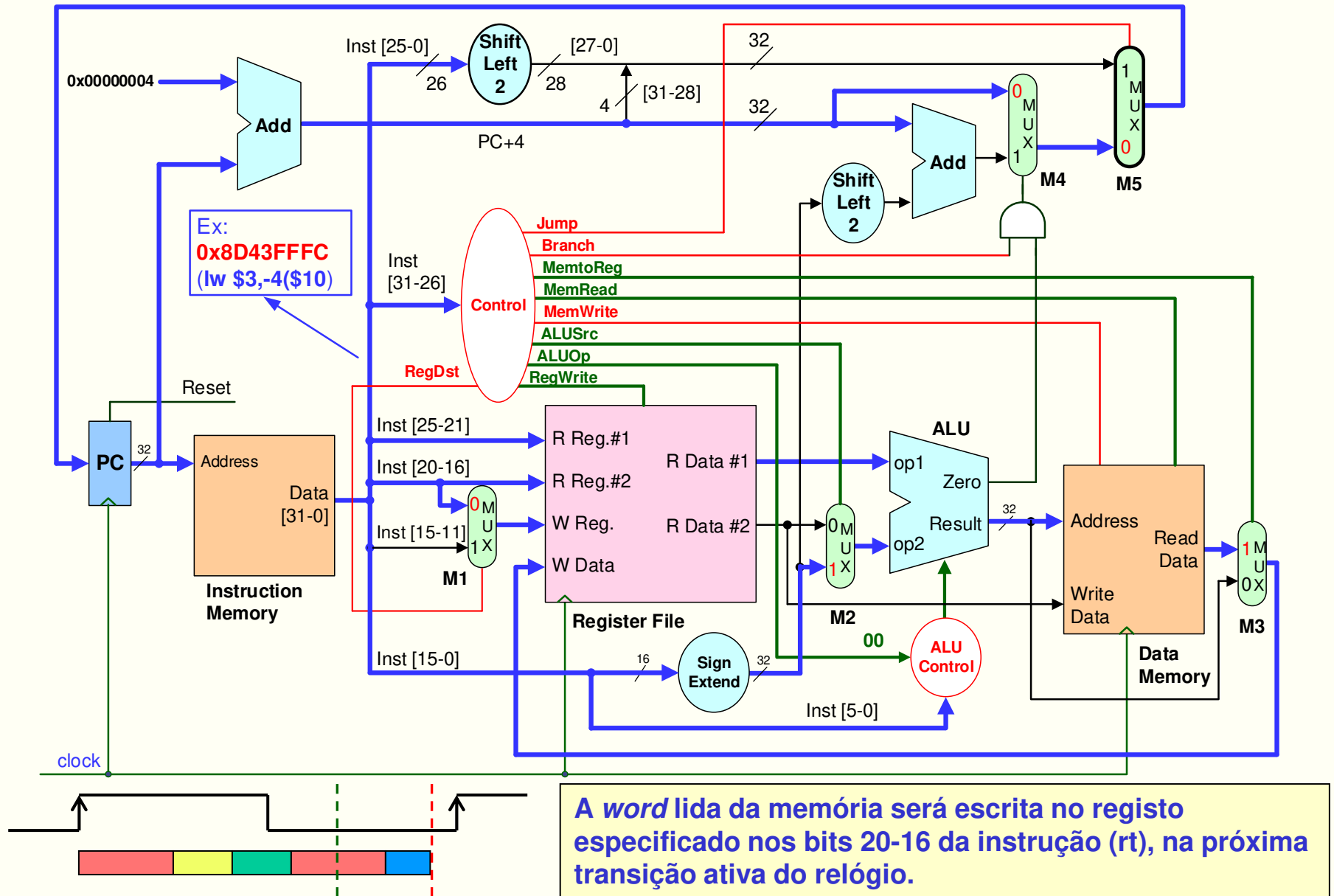
Funcionamento do *datapath* na instrução LW (3)



Funcionamento do *datapath* na instrução LW (4)



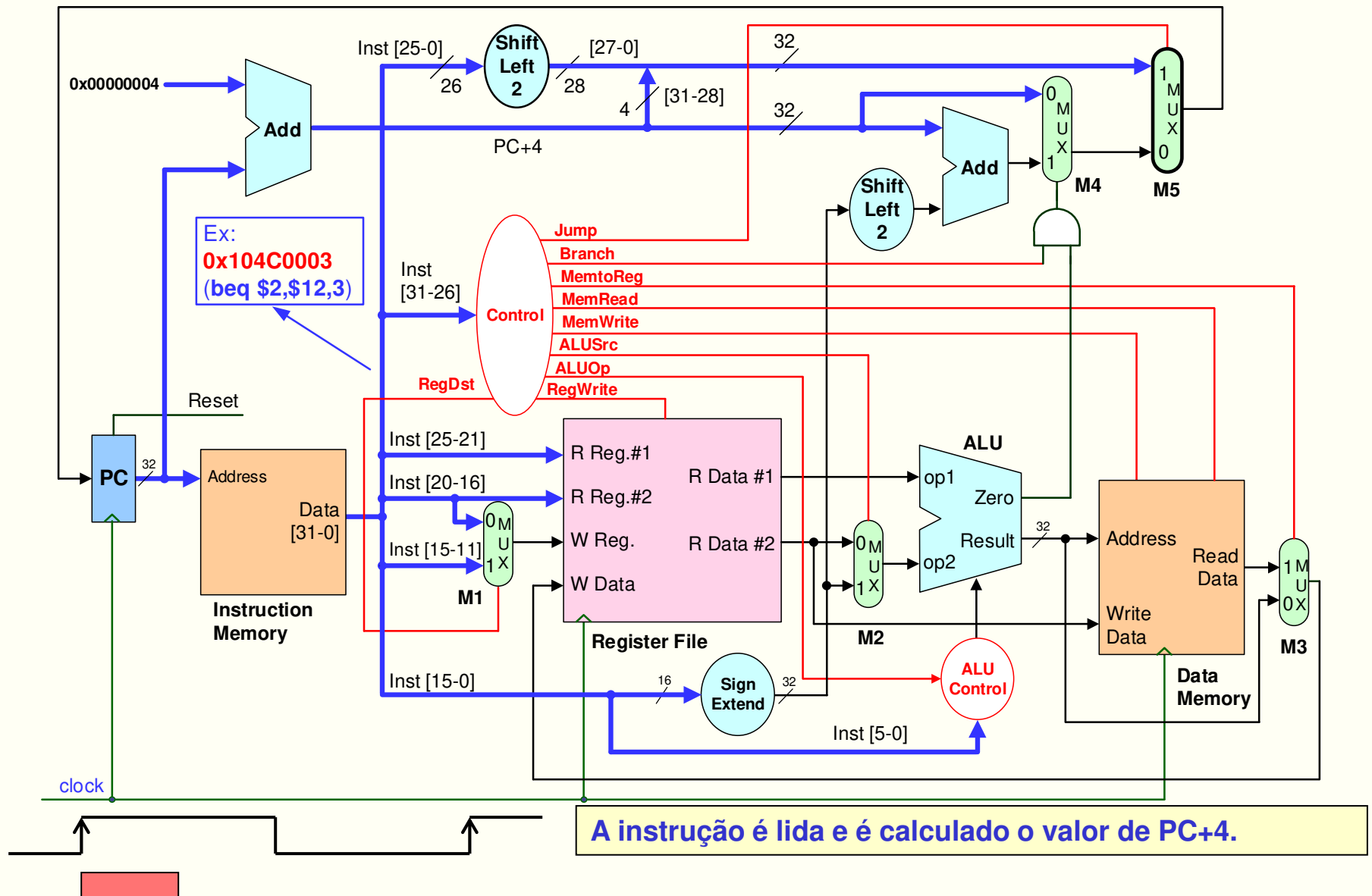
Funcionamento do *datapath* na instrução LW (5)



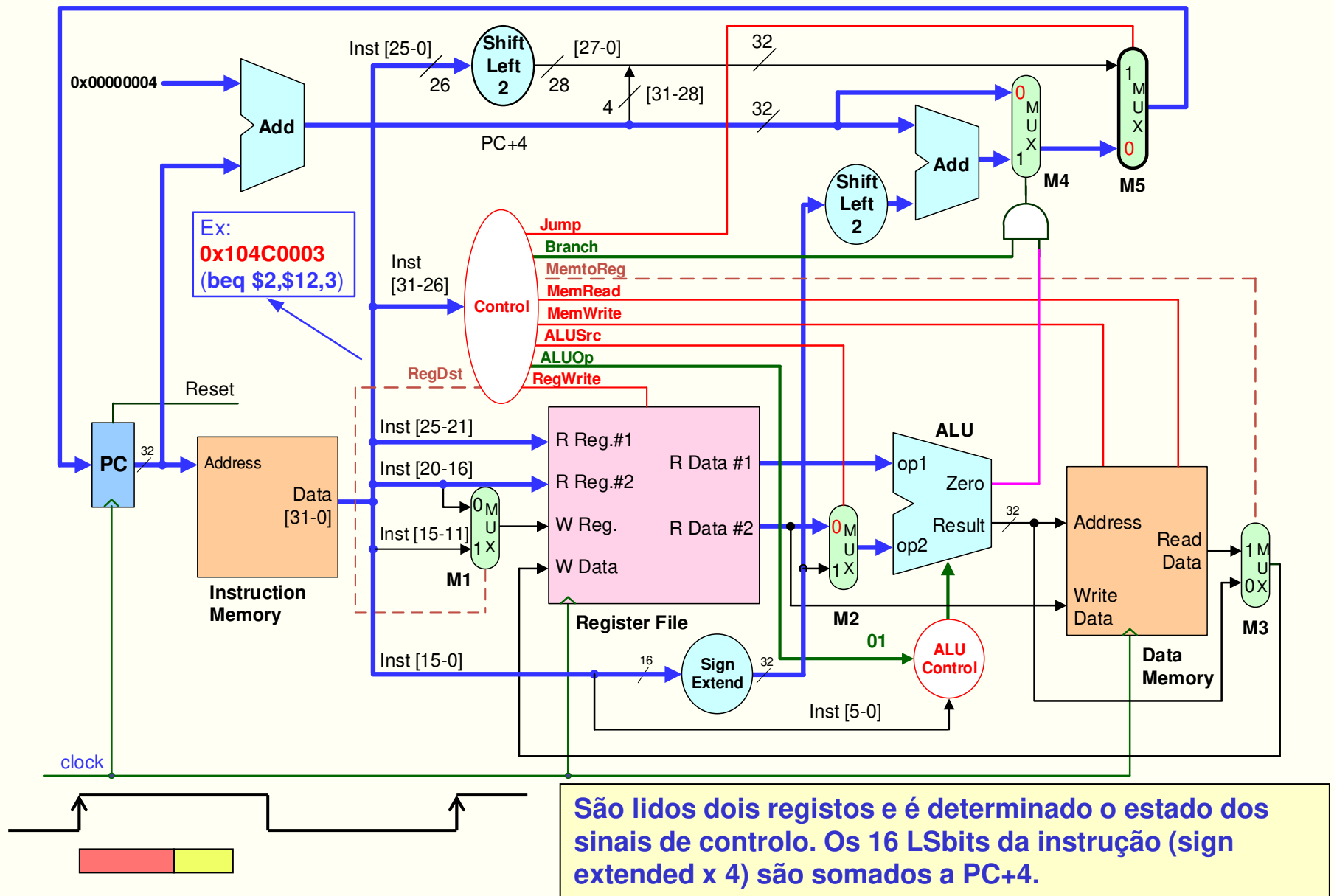
Funcionamento do *datapath* na instrução BEQ

- 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 16 LSbits 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

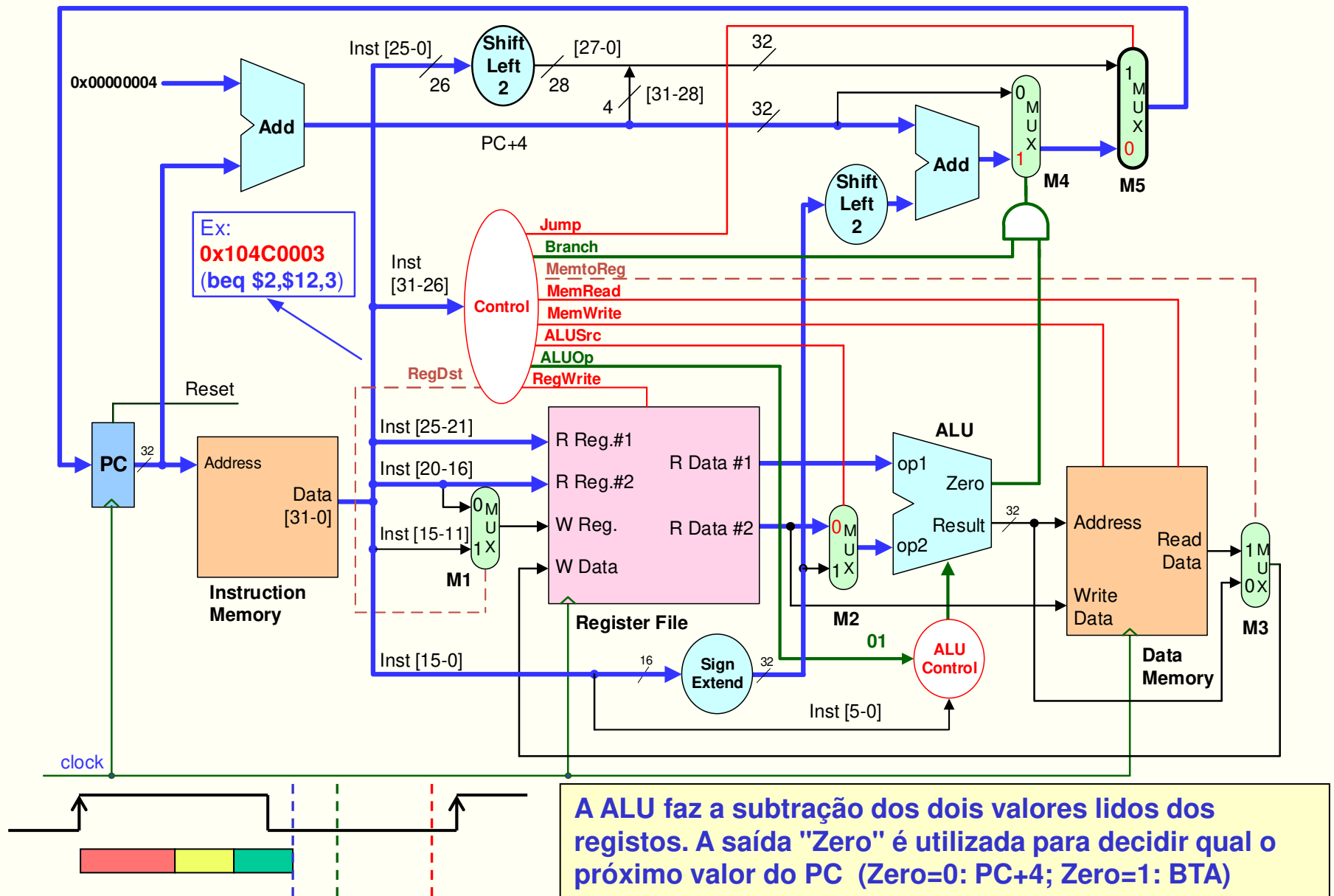
Funcionamento do *datapath* na instrução BEQ (1)



Funcionamento do *datapath* na instrução BEQ (2)



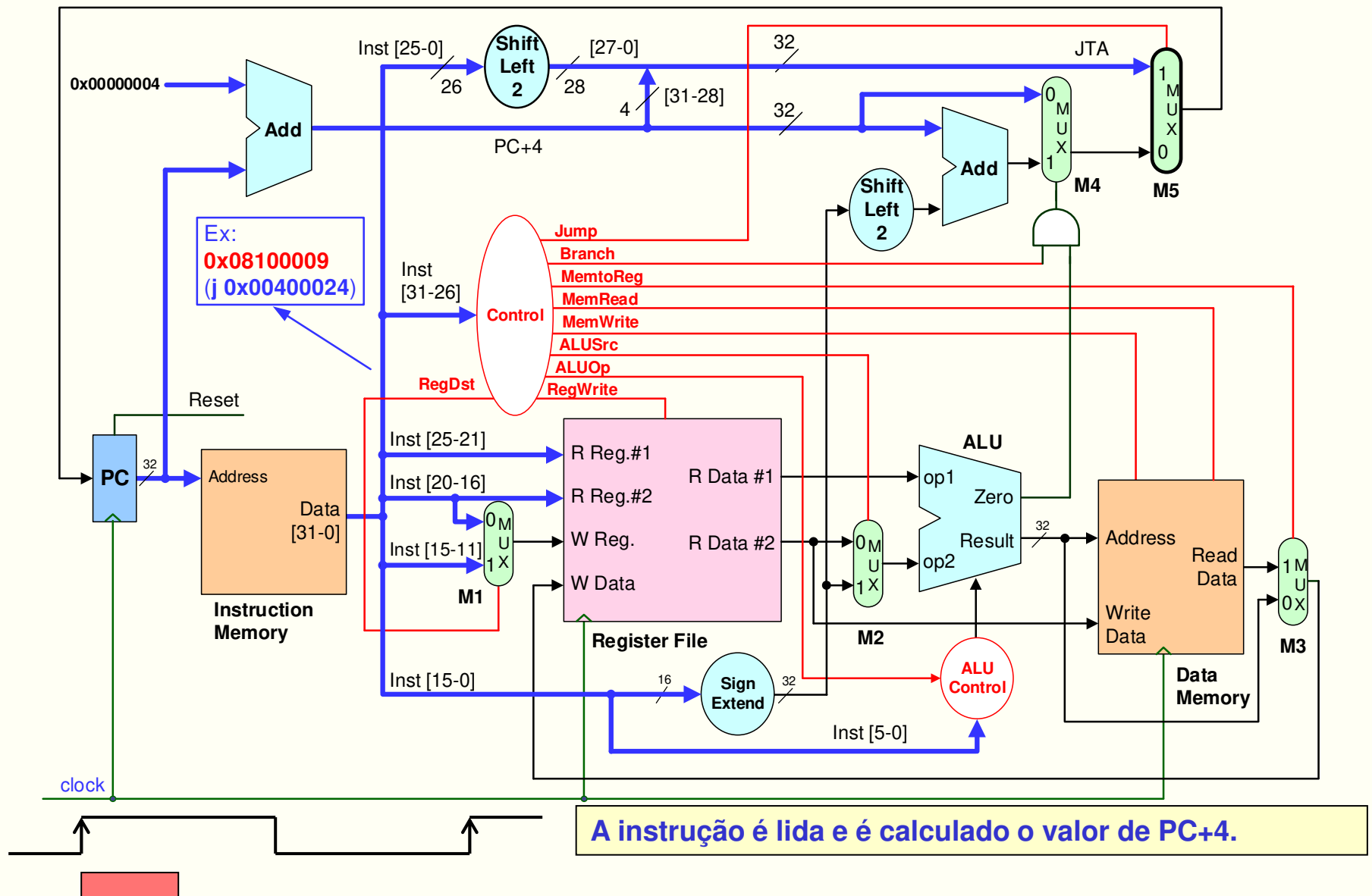
Funcionamento do *datapath* na instrução BEQ (3)



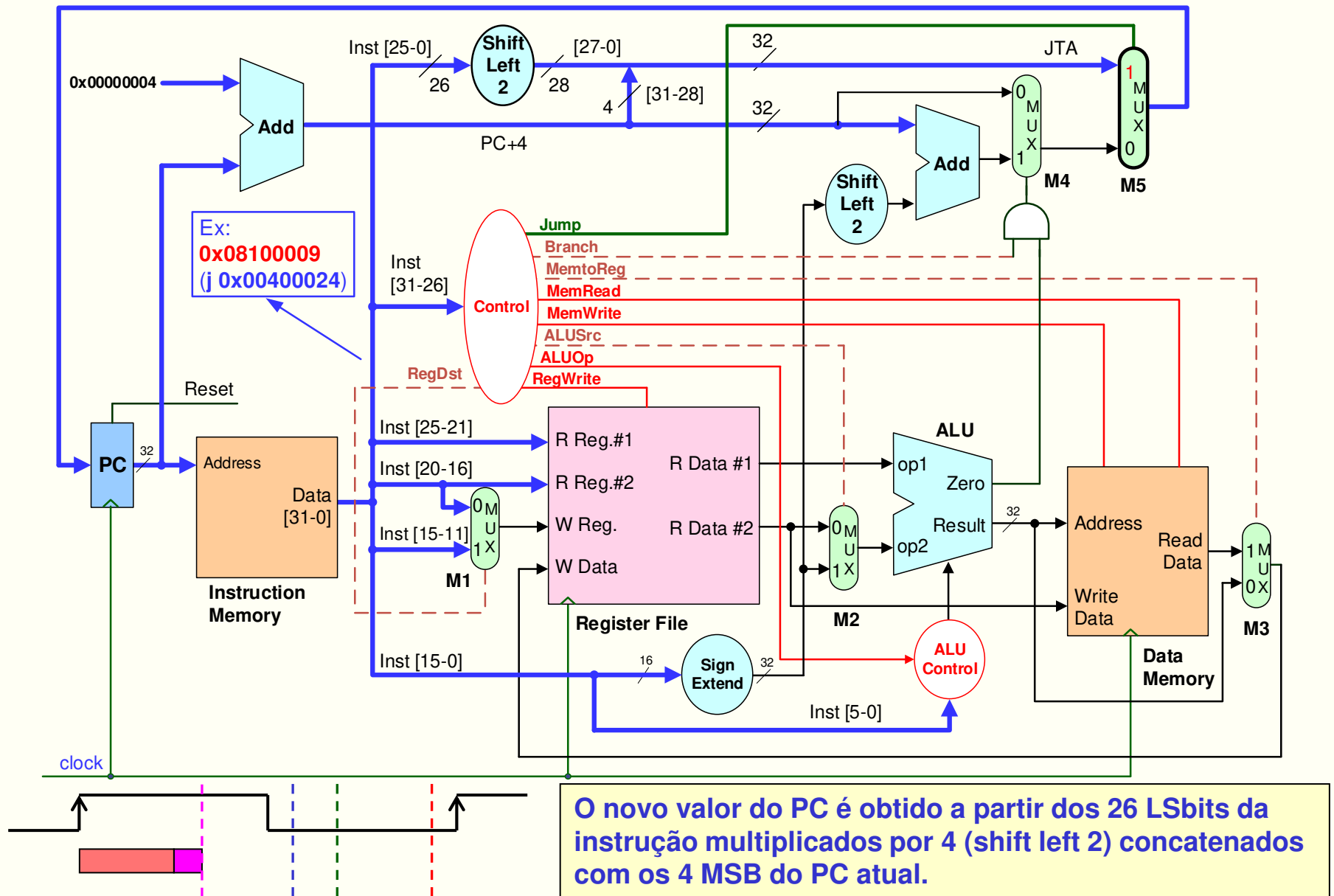
Funcionamento do *datapath* na instrução J

- A instrução é lida e é calculado o valor de PC+4
- São determinados os sinais de controlo. O endereço alvo é obtido a partir dos 26 LSbits da instrução multiplicados por 4 (*shift left 2*) concatenados com os 4 bits mais significativos do PC+4

Funcionamento do *datapath* na instrução J (1)



Funcionamento do *datapath* na instrução J (2)



Execução de uma instrução no DP *single-cycle* – exemplo

- Vai iniciar-se o *instruction fetch* da instrução apontada pelo Program Counter (PC: **0x00400024**). Nesse instante o conteúdo dos registos do CPU e da memória de dados e instruções é o indicado na figura. **Qual o conteúdo dos registos após a execução da instrução?**

Memória de dados

Endereço	Valor
(...)	(...)
0x10010030	0x63F78395
0x10010034	0xA0FCF3F0
0x10010038	0x147FAF83
(...)	(...)

CPU antes

PC	0x00400024
\$3	0x7F421231
\$4	0x15A73C49
\$5	0x10010010

Memória de instruções

Endereço	Código máquina
(...)	(...)
0x00400020	0x00E82820
0x00400024	0x8CA30024
0x00400028	0x00681824
(...)	(...)

CPU depois

PC	0x00400028
\$3	0xA0FCF3F0
\$4	0x15A73C49
\$5	0x10010010



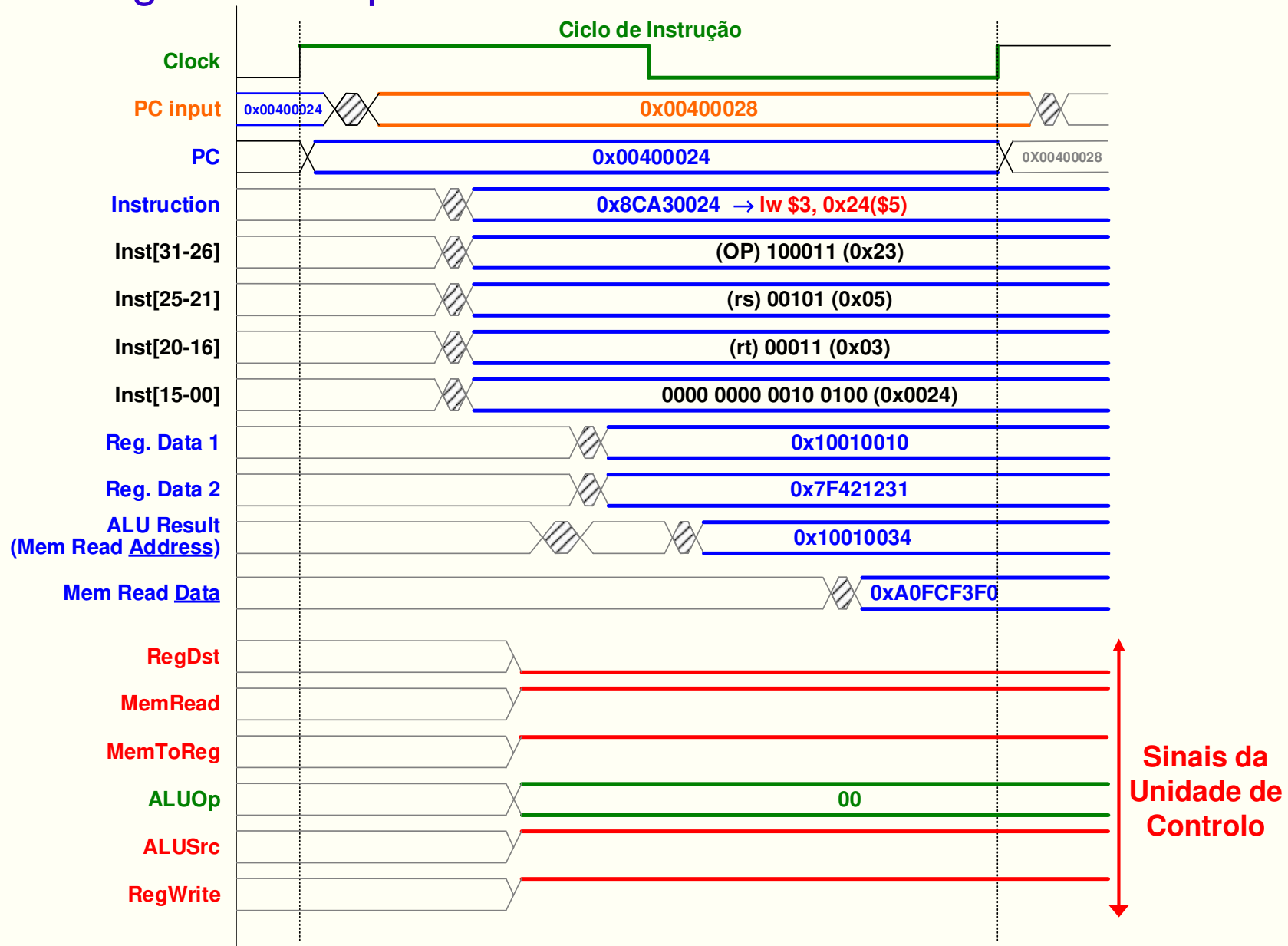
0x8CA30024 → lw \$3, 0x24(\$5)

Mem Addr: 0x10010010 + 0x24 = 0x10010034

1000110010100011000000000100100

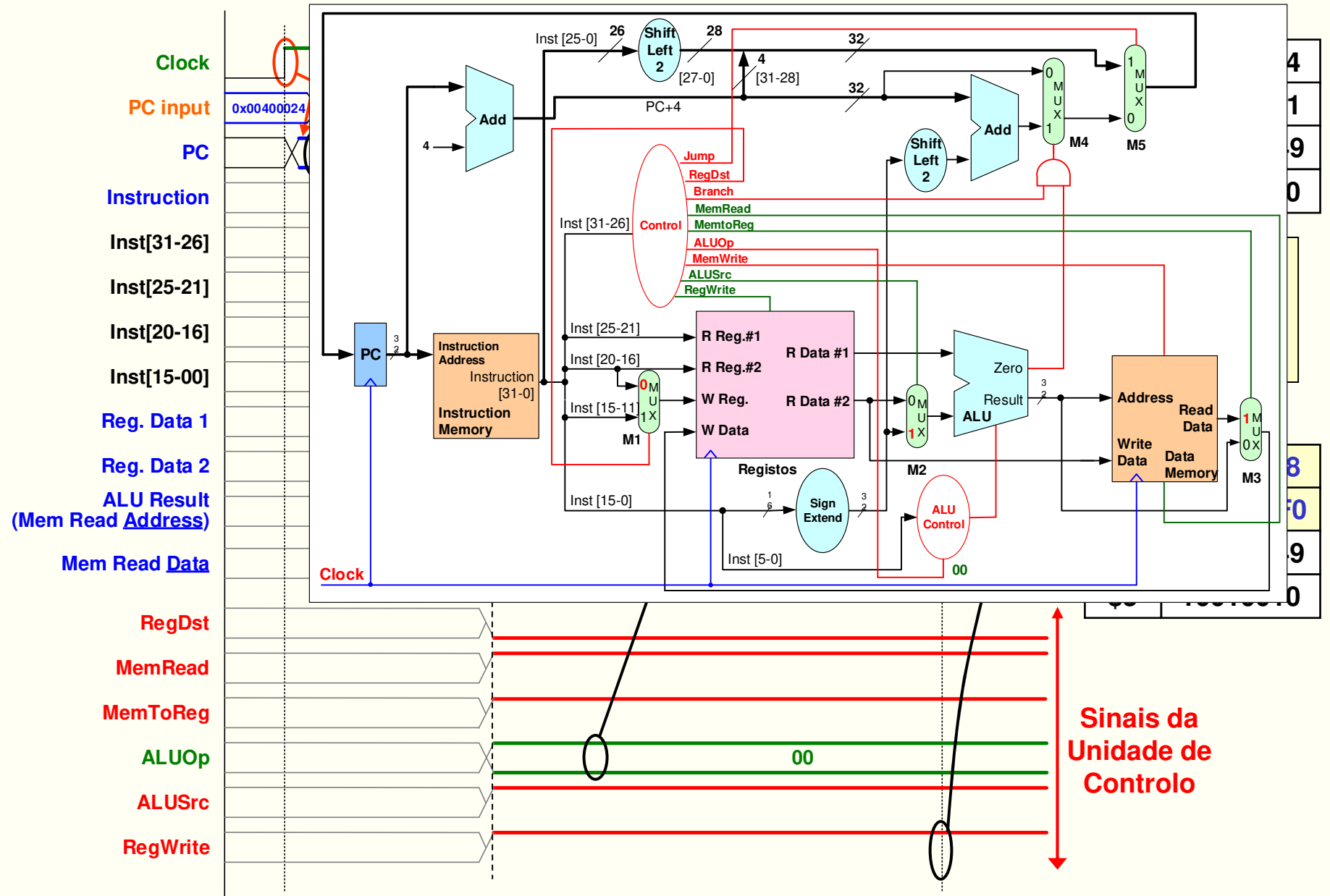
\$3 = [0x10010034] = 0xA0FCF3F0

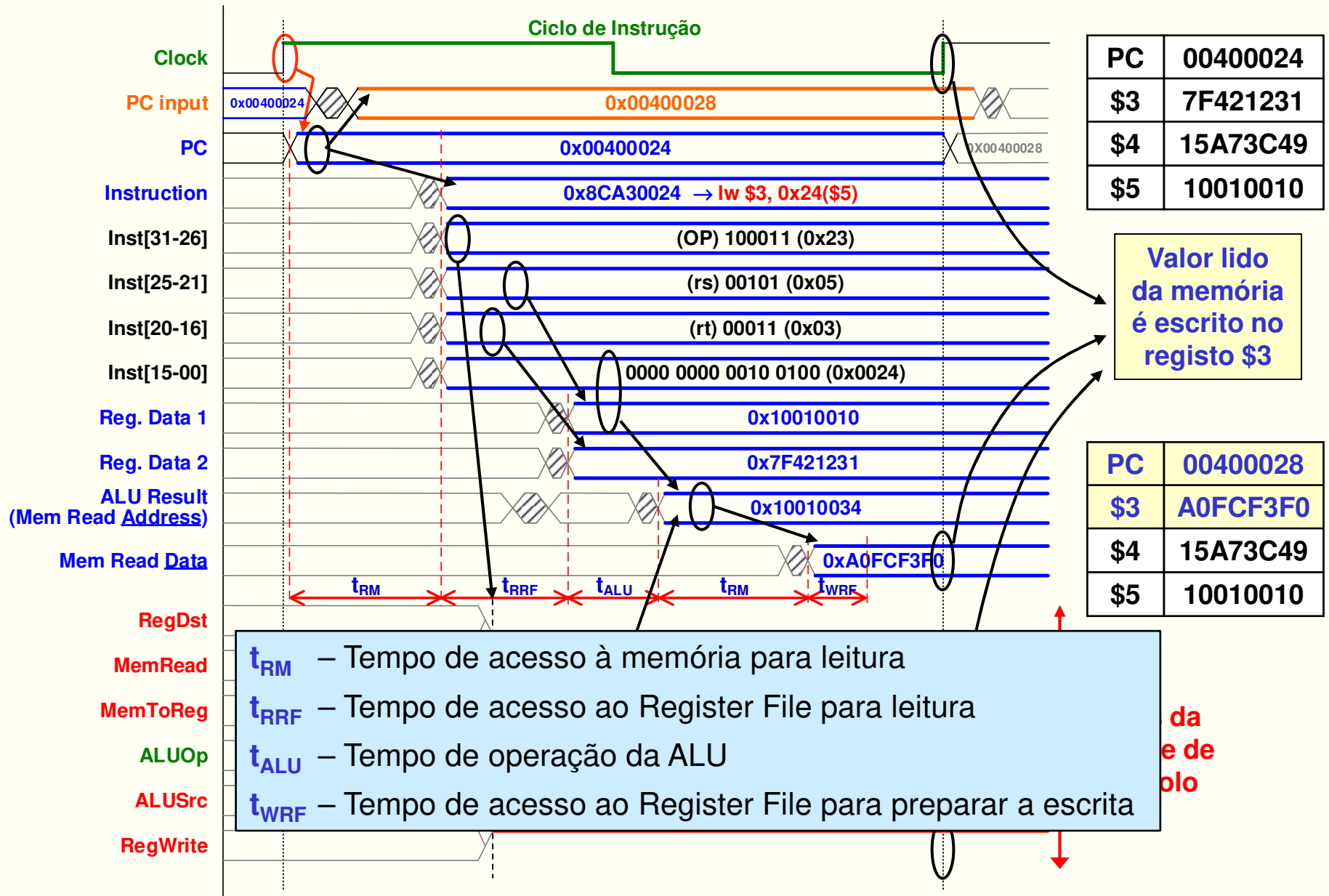
Execução de uma instrução no DP *single-cycle* – diagrama temporal



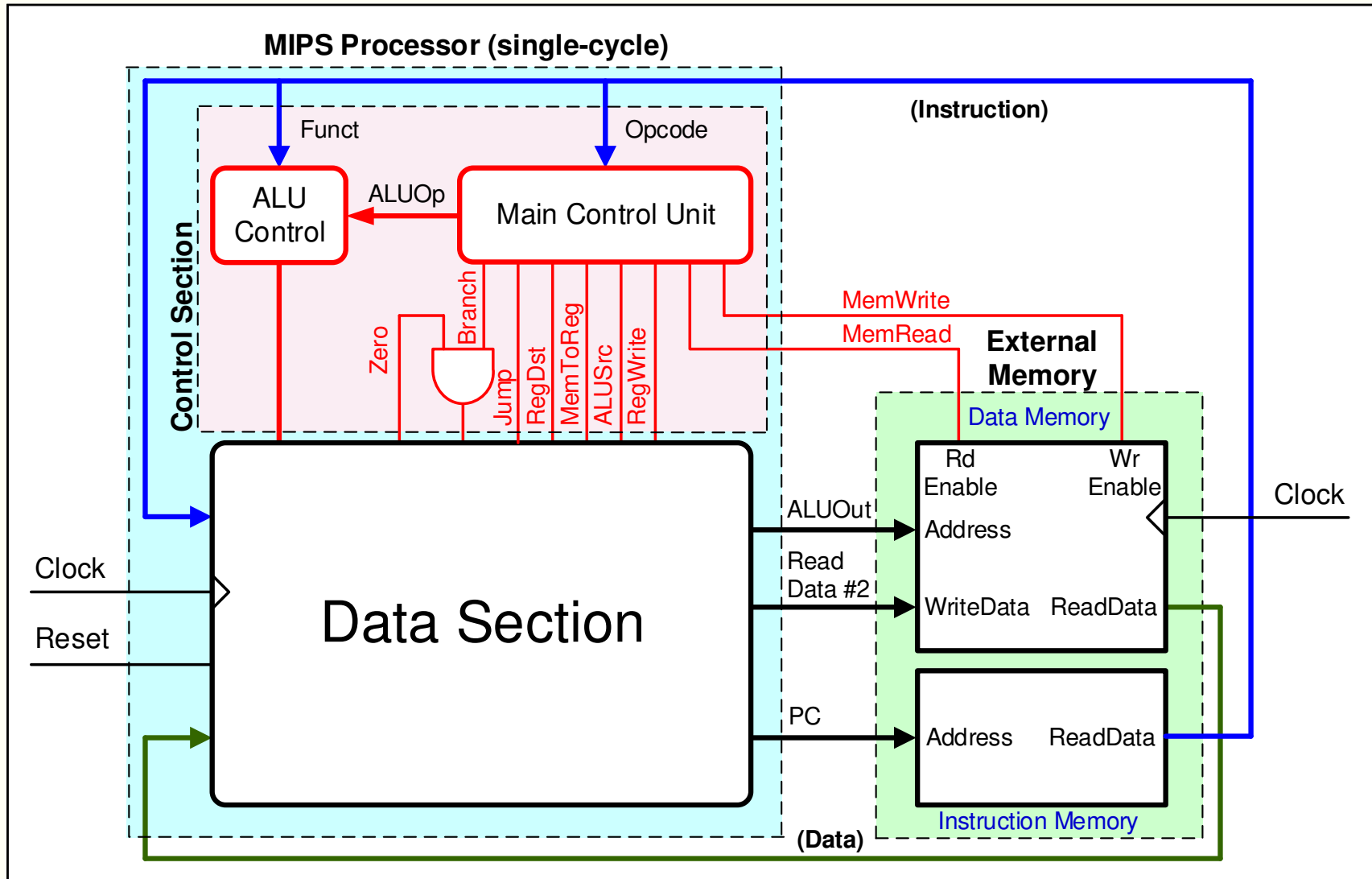
0x00400024	0x8CA30024
0x10010034	0xA0FCF3F0

op	rs	rt	offset
100011	00101	100011	0000000000100100





Visão global do processador



Exercícios

- De que tipo é a unidade de controlo principal do *datapath single-cycle*?
- Como calcularia o tempo mínimo necessário para executar cada uma das instruções anteriormente analisadas?
- O que limita a frequência máxima do relógio do *datapath single-cycle*?
- Que alterações é necessário fazer ao *datapath single-cycle* para permitir a execução das instruções:
 - **"bne"** – branch not equal
 - **"jal"** – jump and link
 - **"jr"** – jump register
 - **"nor"**, **"xor"** e **"sltu"** (todas tipo R)
- Analise o *datapath* e identifique que instruções deixariam de funcionar corretamente se a unidade de controlo bloqueasse o sinal **RegWrite** a '1'.
- Repita o exercício anterior para cada uma das seguintes situações:
RegWrite='0', MemRead='0', MemWrite='0', ALUOp="00",
RegDst='1', ALUSrc='0', MemtoReg='0', MemtoReg='1'
- Que consequência teria para o funcionamento do *datapath* o bloqueio do sinal **Branch** a '1'?

Aulas 19, 20 e 21

- 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

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Tempo de execução das instruções (*single-cycle*)

- A **frequência máxima** do relógio de sincronização do *datapath single-cycle* está limitada pelo **tempo de execução da instrução “mais longa”**
- O **tempo de execução** de uma instrução corresponde ao **somatório dos atrasos introduzidos por cada um dos elementos operativos envolvidos na sua execução**
- Note-se que apenas os elementos operativos que se encontram em **série** contribuem para aumentar o tempo necessário para concluir a execução da instrução (caminho crítico)

Tempo de execução das instruções

- Consideremos os seguintes tempos de atraso introduzidos por cada um dos elementos operativos 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 da

- Considerando os tempos de ativação das várias instruções suportadas pelo processador

Instruções tipo R:

- $t_{EXEC} = t_{RM} + \max(t_{RRF}, t_{CNTL})$

Instrução SW:

- $t_{EXEC} = t_{RM} + \max(t_{RRF}, t_{CNTL})$

Instrução LW:

- $t_{EXEC} = t_{RM} + \max(t_{RRF}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{RM} + t_{WRF}$

Instrução BEQ:

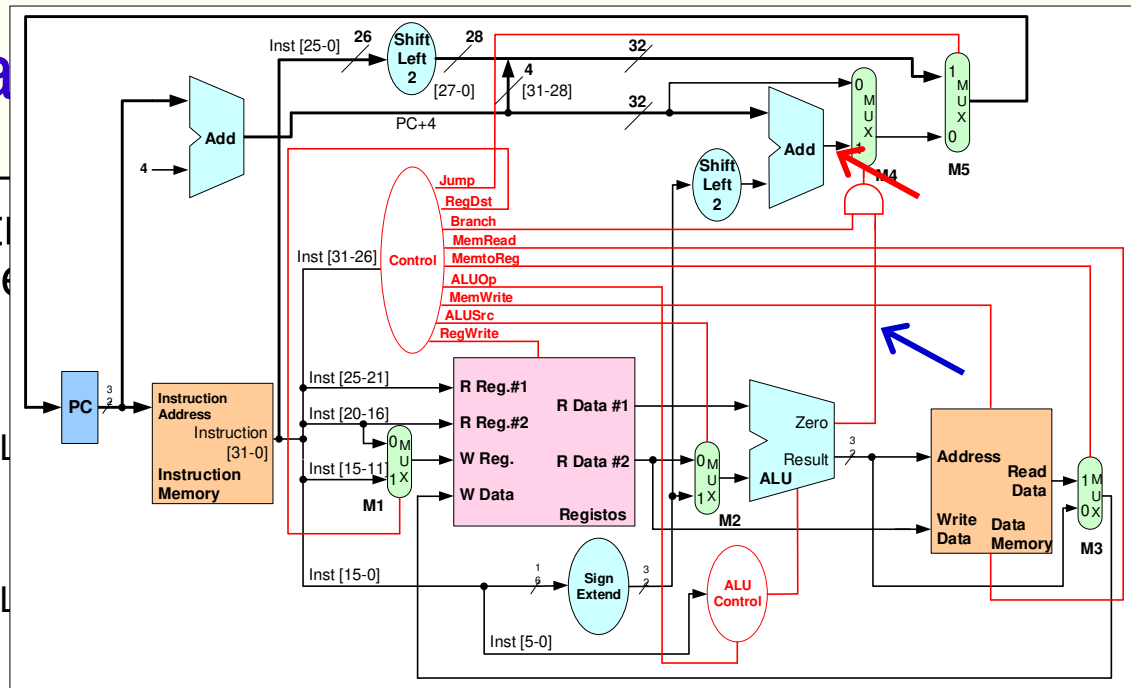
- $t_{EXEC} = t_{RM} + \max(\underbrace{\max(t_{RRF}, t_{CNTL}) + t_{ALU}}_{\text{comparação}}, \underbrace{t_{SE} + t_{SL2} + t_{ADD}}_{\text{cálculo do BTA}}) + t_{stPC}$

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

- 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:

- $t_{EXEC} = t_{RM} + \max(\underbrace{\max(t_{RRF}, t_{CNTL}) + t_{ALU}}_{\text{comparação}}, \underbrace{t_{SE} + t_{SL2} + t_{ADD}}_{\text{cálculo do BTA}}) + t_{stPC}$

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 operativos do *datapath single-cycle*:

▪ Acesso à memória para leitura (t_{RM}):	5ns
▪ Acesso à memória para preparar escrita (t_{WM}):	5ns
▪ Acesso ao <i>register file</i> para leitura (t_{RRF}):	3ns
▪ Acesso ao <i>register file</i> para preparar escrita (t_{WRF}):	3ns
▪ Operação da ALU (t_{ALU}):	4ns
▪ Operação de um somador (t_{ADD}):	1ns
▪ <i>Multiplexers</i> e restantes elementos operativos:	0ns
▪ Unidade de controlo (t_{CNTL}):	1ns
▪ Tempo de <i>setup</i> do PC (t_{stPC}):	1ns

Tempo de execução das instruções - exemplo

- **Instruções tipo R:**

- $t_{EXEC} = t_{RM} + \max(t_{RFR}, t_{CNTL}) + t_{ALU} + t_{WFR}$
 $= 5 + \max(3, 1) + 4 + 3 = \mathbf{15\ ns}$

- **Instrução SW:**

- $t_{EXEC} = t_{RM} + \max(t_{RFR}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{WM}$
 $= 5 + \max(3, 1, 0) + 4 + 5 = \mathbf{17\ ns}$

- **Instrução LW:**

- $t_{EXEC} = t_{RM} + \max(t_{RFR}, t_{CNTL}, t_{SE}) + t_{ALU} + t_{RM} + t_{WFR}$
 $= 5 + \max(3, 1, 0) + 4 + 5 + 3 = \mathbf{20\ ns}$

- **Instrução BEQ:**

- $t_{EXEC} = t_{RM} + \max(\max(t_{RFR}, t_{CNTL}) + t_{ALU}, t_{SE} + t_{SL2} + t_{ADD}) + t_{stPC}$
 $= 5 + \max(\max(3, 1) + 4, 0 + 0 + 1) + 1 = \mathbf{13\ ns}$

- **Instrução J:**

- $t_{EXEC} = t_{RM} + \max(t_{CNTL}, t_{SL2}) + t_{stPC} = 5 + \max(1, 0) + 1 = \mathbf{7\ ns}$

Limitações das soluções *single-cycle*

- Face à análise anterior, a máxima frequência de trabalho seria:

$$f_{\max} = 1 / 20\text{ns} = 50\text{MHz}$$

- 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 drasticamente a eficiência do *datapath*, mesmo que as instruções de multiplicação ou divisão sejam raramente utilizadas
- Uma solução ingênua, seria usar um relógio de frequência variável, ajustável em função da instrução que está em execução – é uma solução tecnicamente inviável

Limitações das soluções *single-cycle*

- O tempo de execução de um programa pode ser calculado como:

$$T_{exec_{CPU}} = \# Instruções \times CPI \times Clock_Cycle_{CPU}$$

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:

$$T_{exec_{CPU}} = \# Instruções \times Clock_Cycle_{CPU}$$

- Define-se ainda:

$$Desempenho_{CPU} = 1/T_{exec_{CPU}}$$

- O desempenho 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*

- **Exercício:** 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 seria calculado como:

$$T_{MED_INSTR} = 0,2 \times 20 + 0,1 \times 17 + 0,5 \times 15 + 0,15 \times 13 + 0,05 \times 7 = 15,5ns$$

- O ganho de desempenho do CPU com *clock* variável relativamente a um com *clock* fixo seria então:

$$\frac{Des_{CPU_CLOCK_VARIÁVEL}}{Des_{CPU_CLOCK_FIXO}} = \frac{\# Instruções \times 20}{\# Instruções \times T_{MED_INSTR}} = 1,29$$

A implementação com *clock* variável não é viável mas permite entender o que está a ser sacrificado quando todas as instruções têm que ser executadas num único ciclo de relógio com tempo fixo

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 uma 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 contraria um dos princípios-chave de desenho: *make the common case fast* (o que é mais comum deve ser mais rápido)
- Elementos operativos que estejam envolvidos na execução de uma 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, ...)

O datapath Multi-cycle

- Uma solução para os problemas enumerados passa por 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
 - Em cada ciclo de relógio poderão ser realizadas **várias operações, desde que sejam independentes** (por exemplo, *instruction fetch* e cálculo de PC+4 ou *operand fetch* e cálculo do BTA)
 - Cada uma dessas operações faz uso de um elemento operativo fundamental: memória, *register file* ou ALU
- Desta forma, o período de relógio fica apenas limitado pelo maior dos tempos de atraso de cada um dos elementos operativos fundamentais
- Para os tempos de atraso que considerámos anteriormente, a máxima frequência de relógio seria assim: **$f_{\max} = 1 / t_{\text{RM}} = 1 / 5\text{ns} = 200\text{MHz}$**

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 ações fundamentais da execução de uma instrução:
 - Acesso à memória externa (uma leitura ou uma escrita)
 - Acesso ao *Register File* (uma leitura ou uma escrita)
 - 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**

Alternativa às soluções *single-cycle*

- Uma outra vantagem duma solução de execução em vários ciclos de relógio (*multi-cycle*) é que um **mesmo elemento operativo** pode ser utilizado mais do que uma vez, no contexto da execução de uma 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 – fases de execução

Fase 1 (memória, ALU):

- *Instruction fetch* e cálculo de PC+4

Fase 2 (register file, ALU, unidade de controlo):

- *Operand fetch* e cálculo do *branch target address* e *Instruction decode*

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)

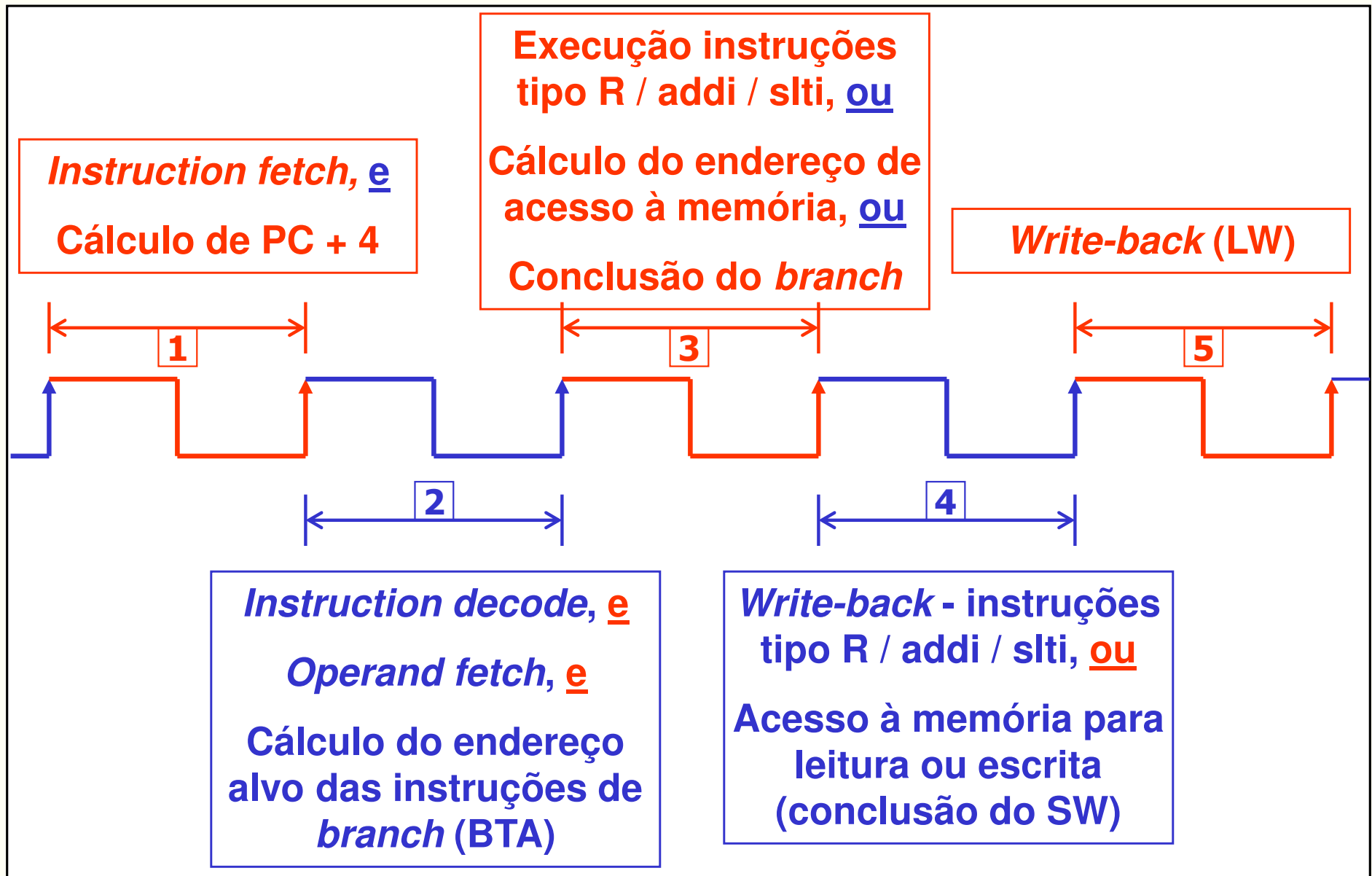
Fase 4 (memória, register file):

- 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**)

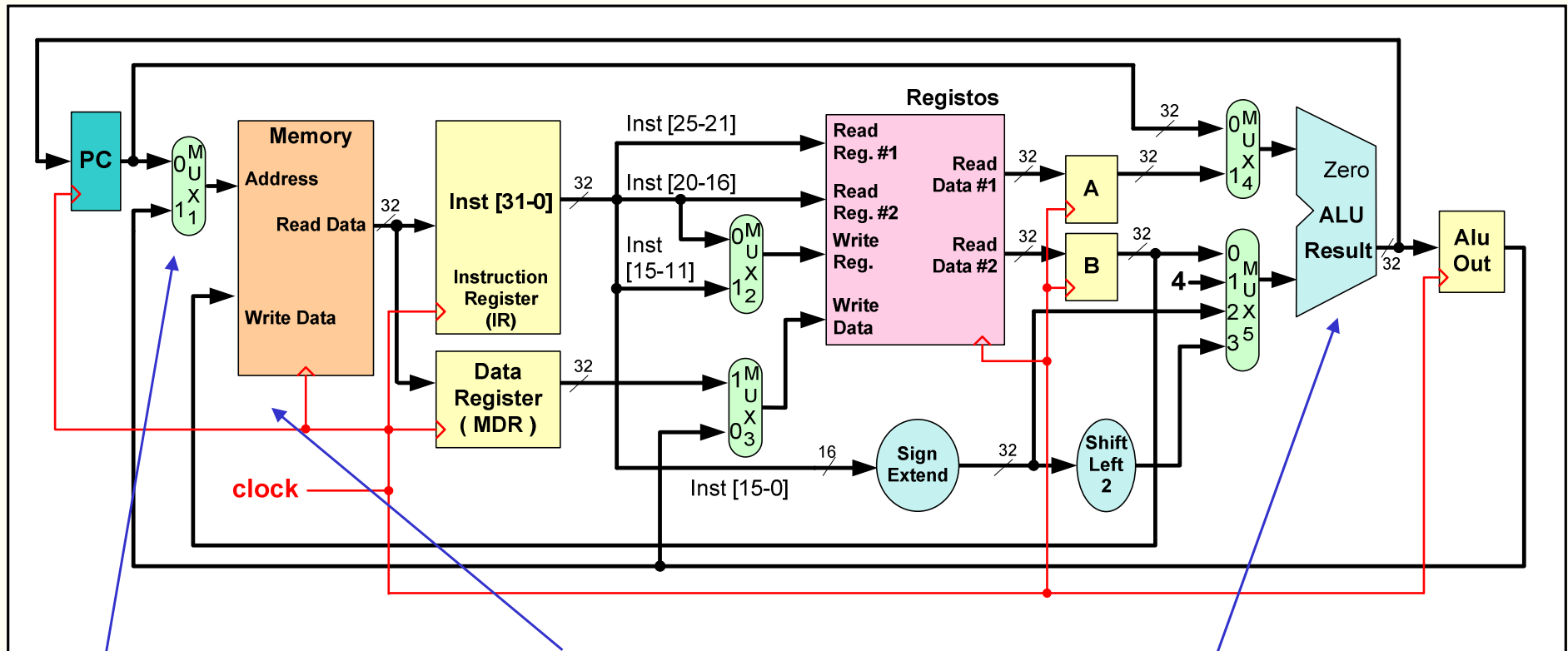
Fase 5 (register file):

- Escrita no *Register File* (conclusão da instrução LW: **write-back**)

O datapath Multi-cycle – fases de execução

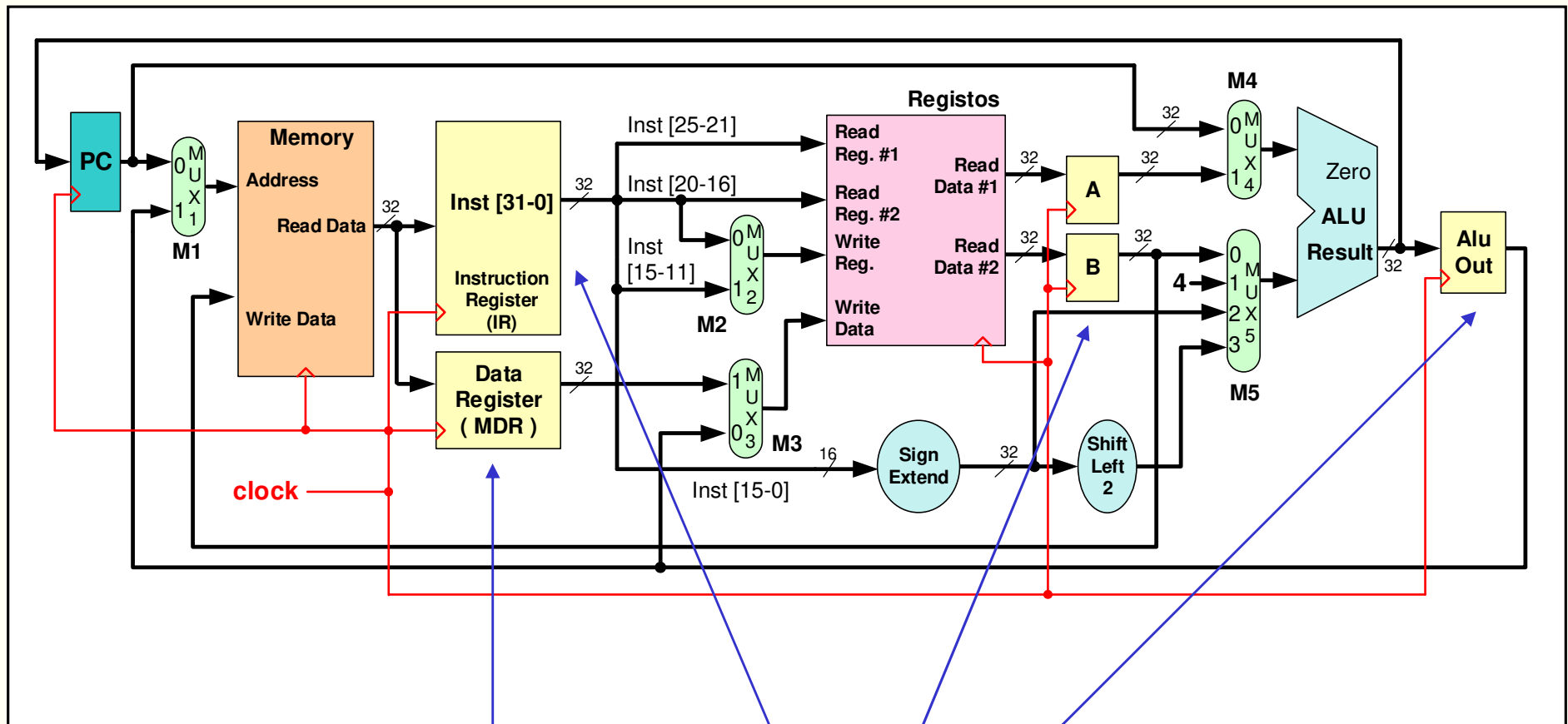


O *datapath Multi-cycle* (sem BEQ e J)



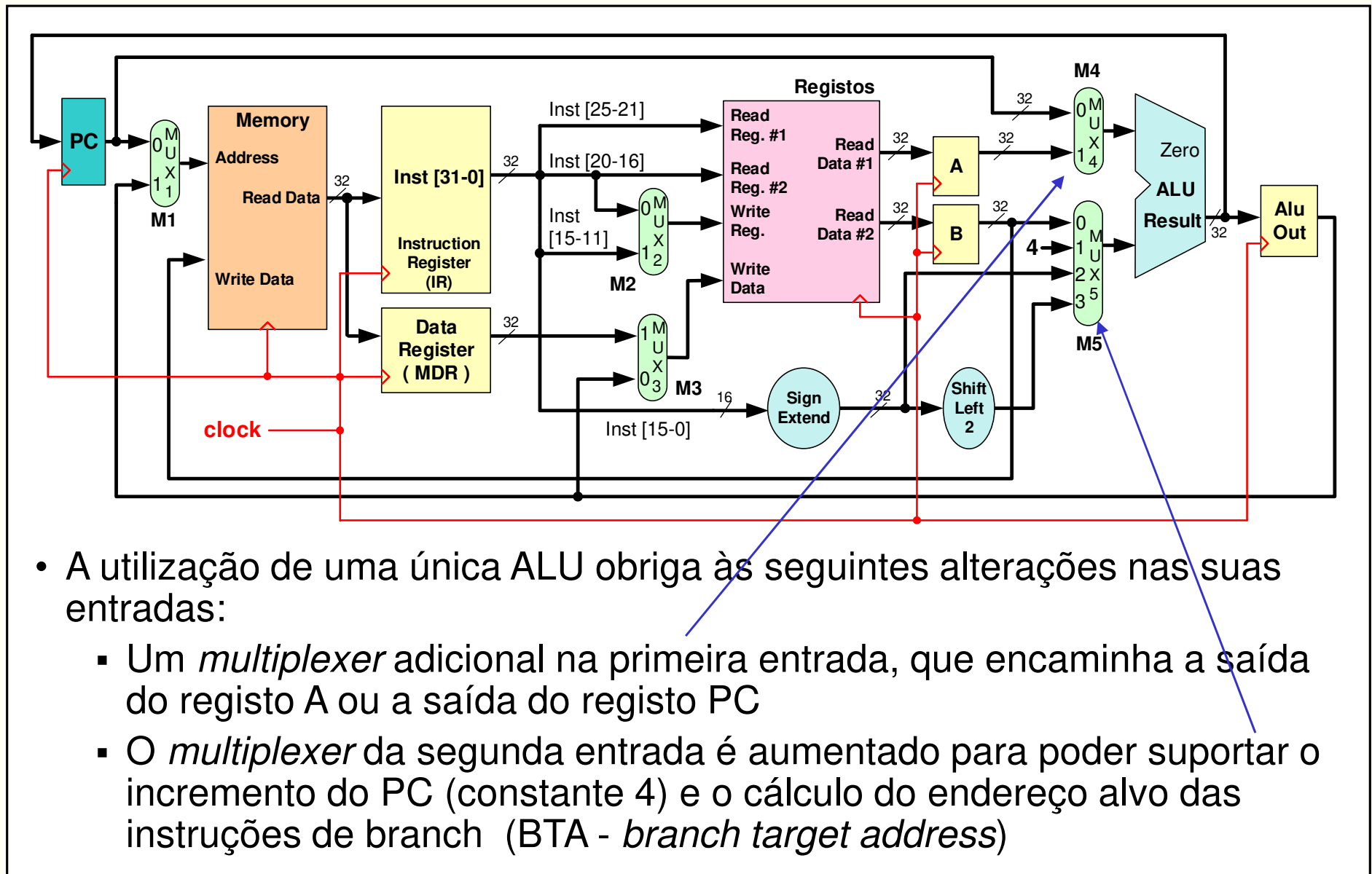
- **Uma única memória para programa e dados**
 - Um *multiplexer* no barramento de endereços da memória permite selecionar o endereço a usar:
 - o conteúdo do PC (para leitura da instrução) ou
 - o valor calculado na ALU (para acesso de leitura/escrita de dados nas instruções LW/SW)
- **Uma única ALU** (em vez de uma ALU e dois somadores)

O datapath Multi-cycle (sem BEQ e J)



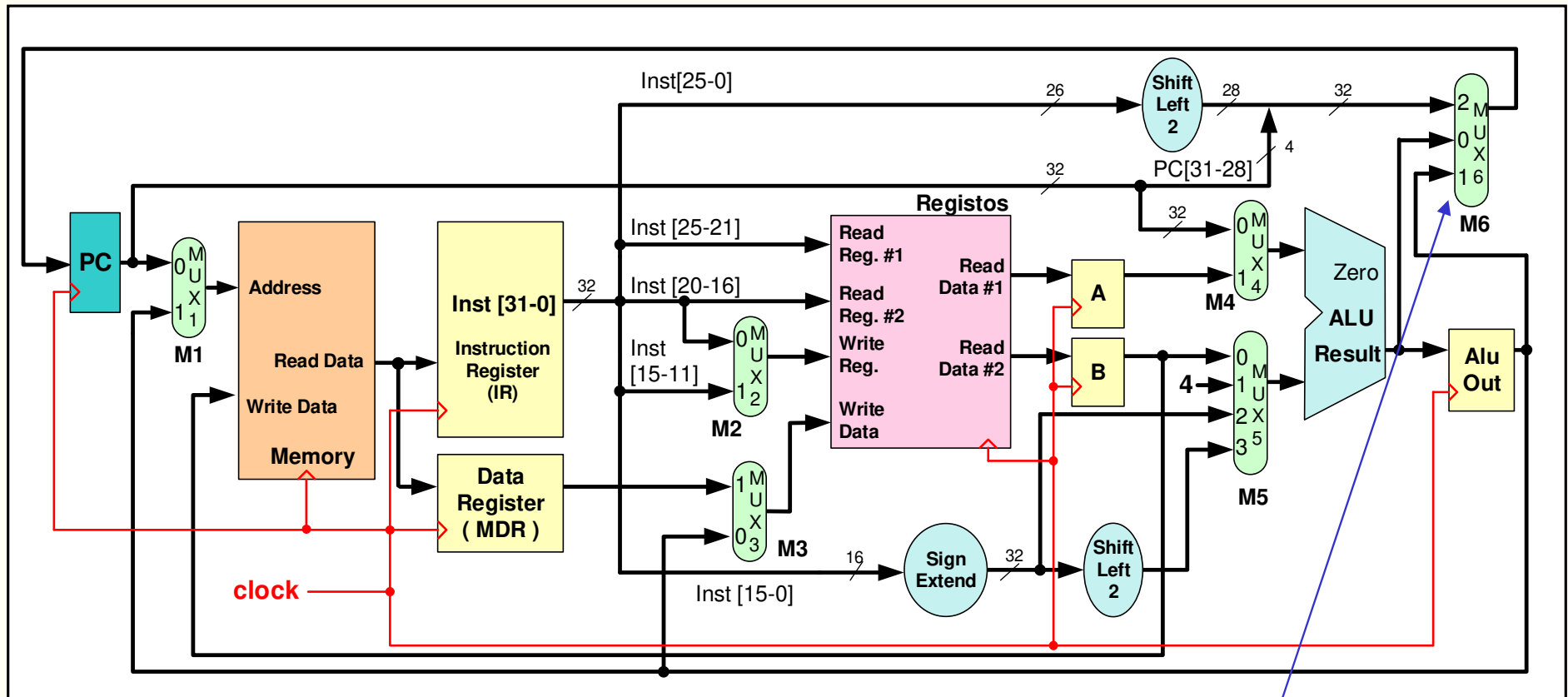
- Registos adicionados à saída dos elementos operativos fundamentais para armazenamento da informação obtida/calculada durante o ciclo de relógio corrente e que será utilizada no ciclo de relógio seguinte

O datapath Multi-cycle (sem BEQ e J)



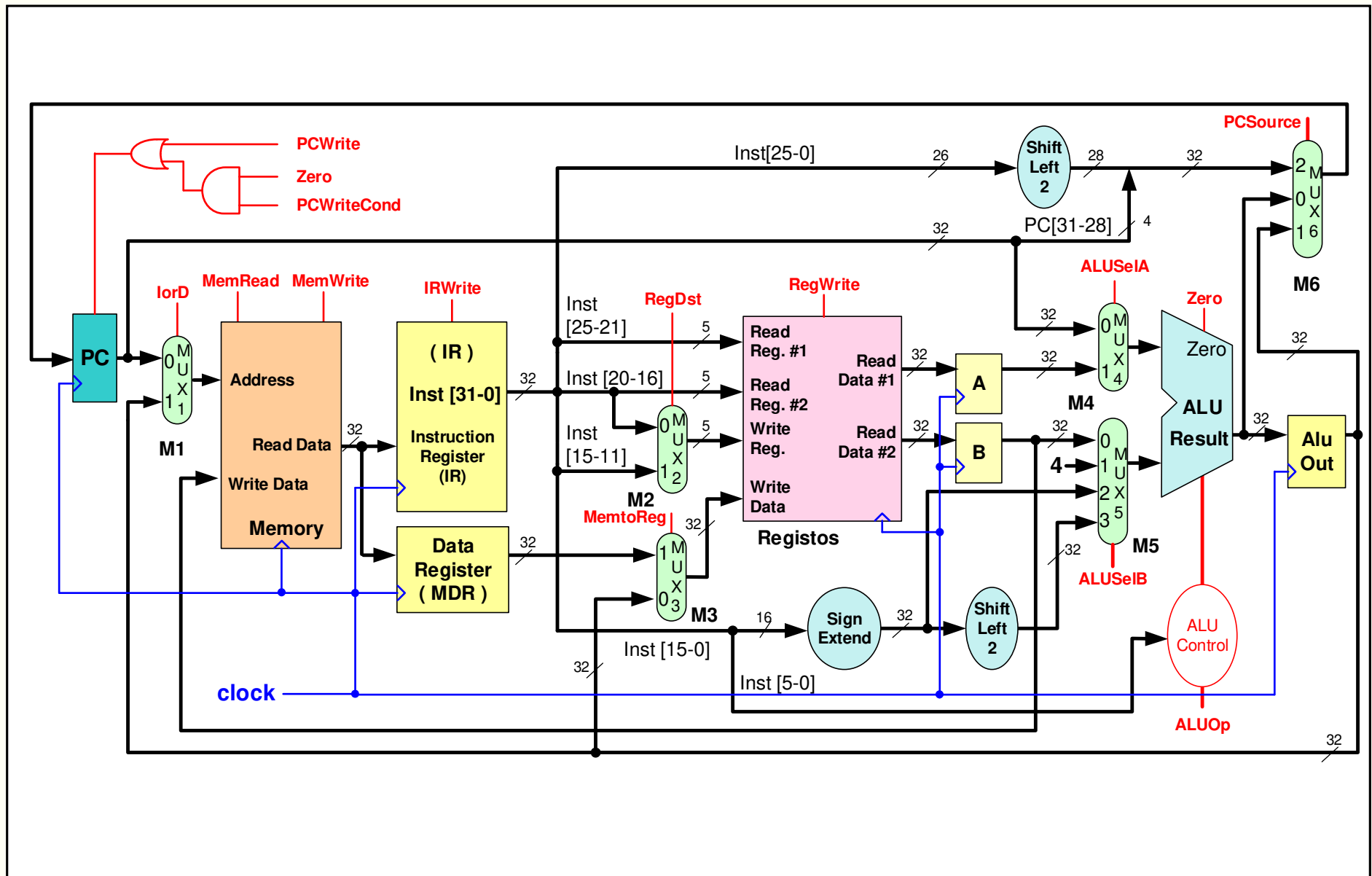
- A utilização de uma única ALU obriga às seguintes alterações nas suas entradas:
 - Um *multiplexer* adicional na primeira entrada, que encaminha a saída do registo A ou a saída do registo PC
 - O *multiplexer* da segunda entrada é aumentado para poder suportar o incremento do PC (constante 4) e o cálculo do endereço alvo das instruções de branch (BTA - *branch target address*)

O datapath Multi-cycle com as instruções BEQ e J



- 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 1ª fase)
 - A saída do registro **ALUOut** que armazena o endereço alvo das instruções de *branch* (BTA) calculado na ALU (na 2ª 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 1ª 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')
MemRead	Nenhum (barramento de dados da memória em alta impedância)	O conteúdo da memória no endereço indicado é apresentado à saída
MemWrite	Nenhum	O conteúdo do registo de memória, cujo endereço é fornecido, é substituído pelo valor apresentado à entrada
RegWrite	Nenhum	O registo indicado no endereço de escrita é alterado pelo valor presente na entrada de dados
IRWrite	Nenhum	O valor lido da memória externa é escrito no Instruction Register
PCWrite	Nenhum	O PC é atualizado incondicionalmente na próxima transição ativa do sinal de relógio
PCWriteCond	Nenhum	O PC é atualizado condicionalmente na próxima transição ativa do relógio
ALUSelA	O primeiro operando da ALU é o PC	O primeiro operando da ALU provém do registo indicado no campo rs
RegDst	O endereço do registo destino provém do campo rt	O endereço do registo destino provém do campo rd
MemtoReg	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
lorD	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

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 estendido dos 16 bits menos significativos do IR (instruction register)
	11	A segunda entrada da ALU é a versão de sinal estendido 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 "funct" 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

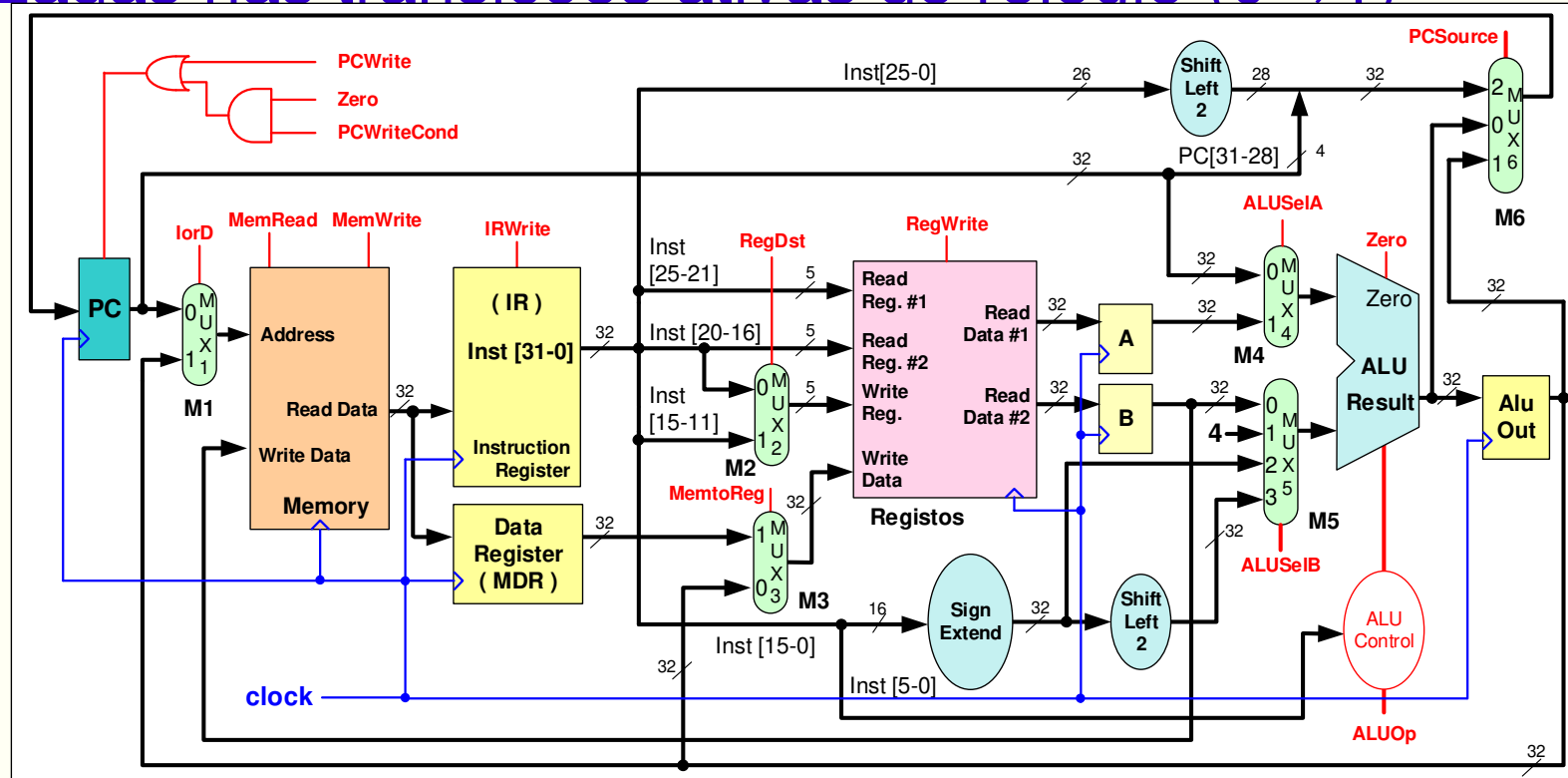
Ações realizadas nas transições ativas do relógio (0→1)

Início da execução da instrução

IR = Memory[PC]

PC = PC + 4

1

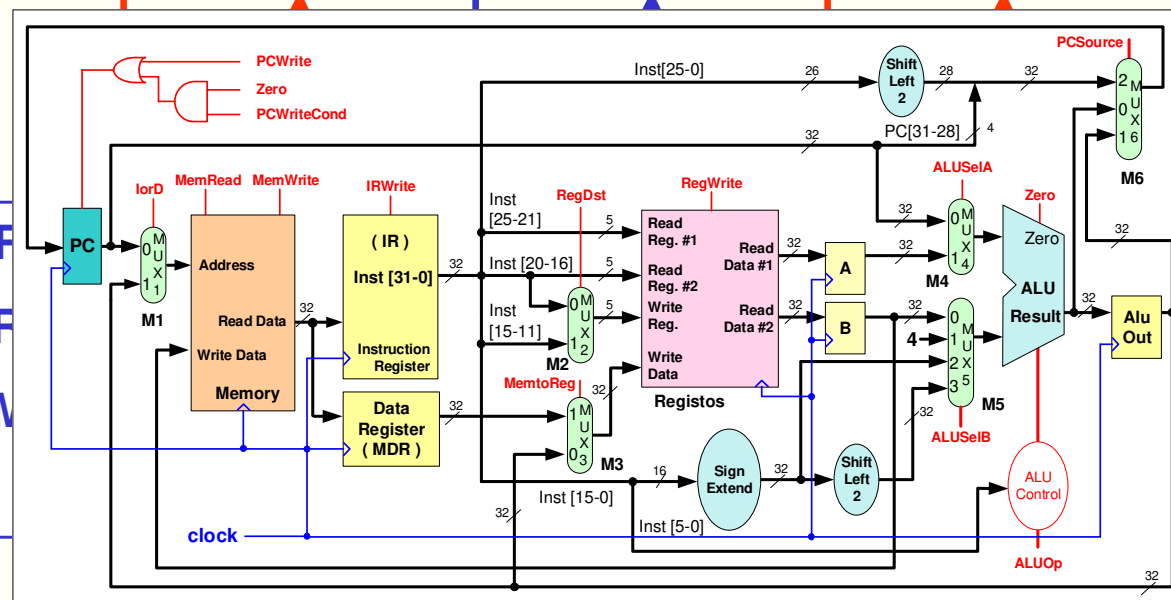


2

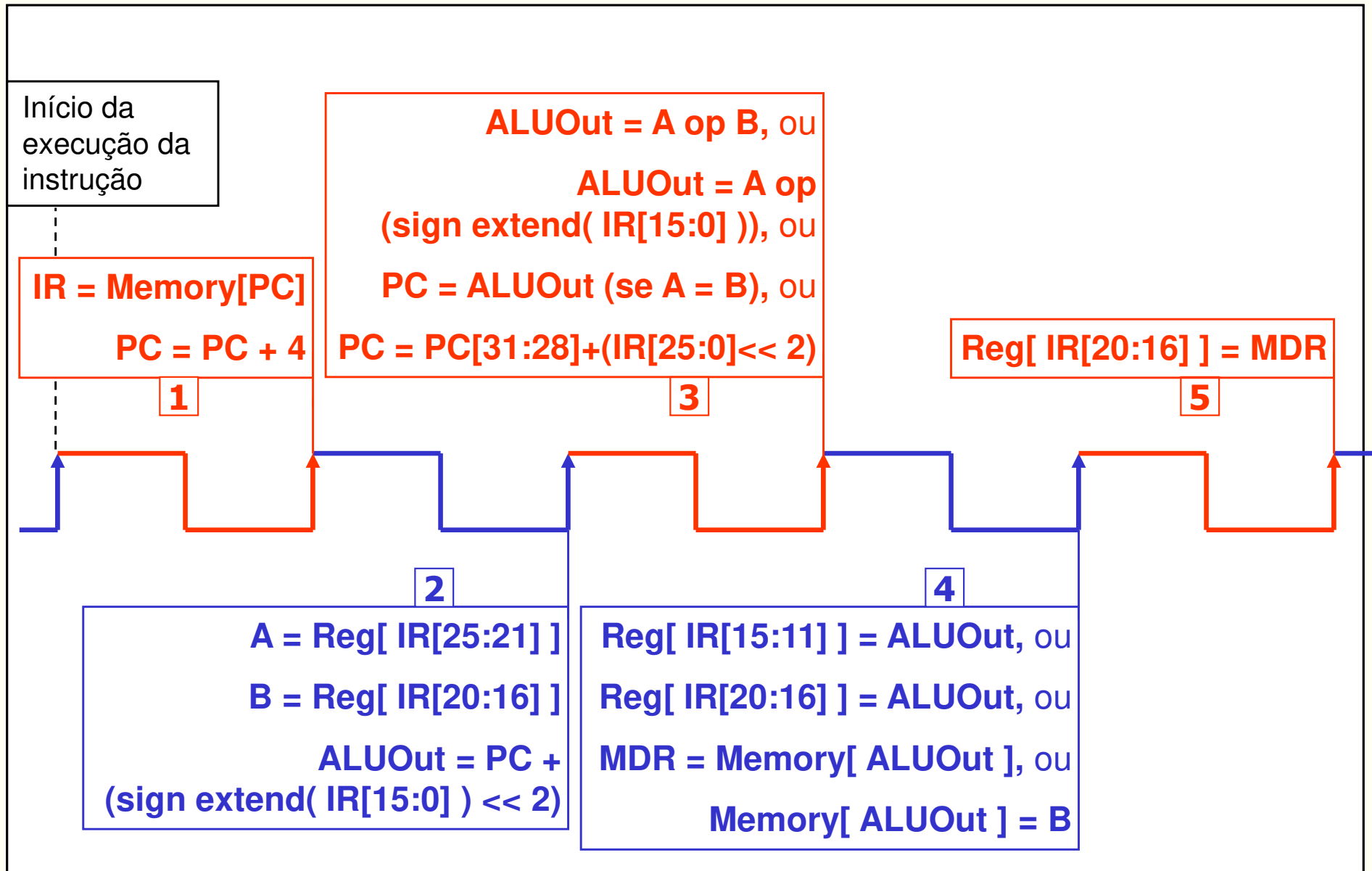
A = Reg[IR[25:21]]

B = Reg[IR[20:16]]

**ALUOut = PC +
(sign extend(IR[15:0]) << 2)**



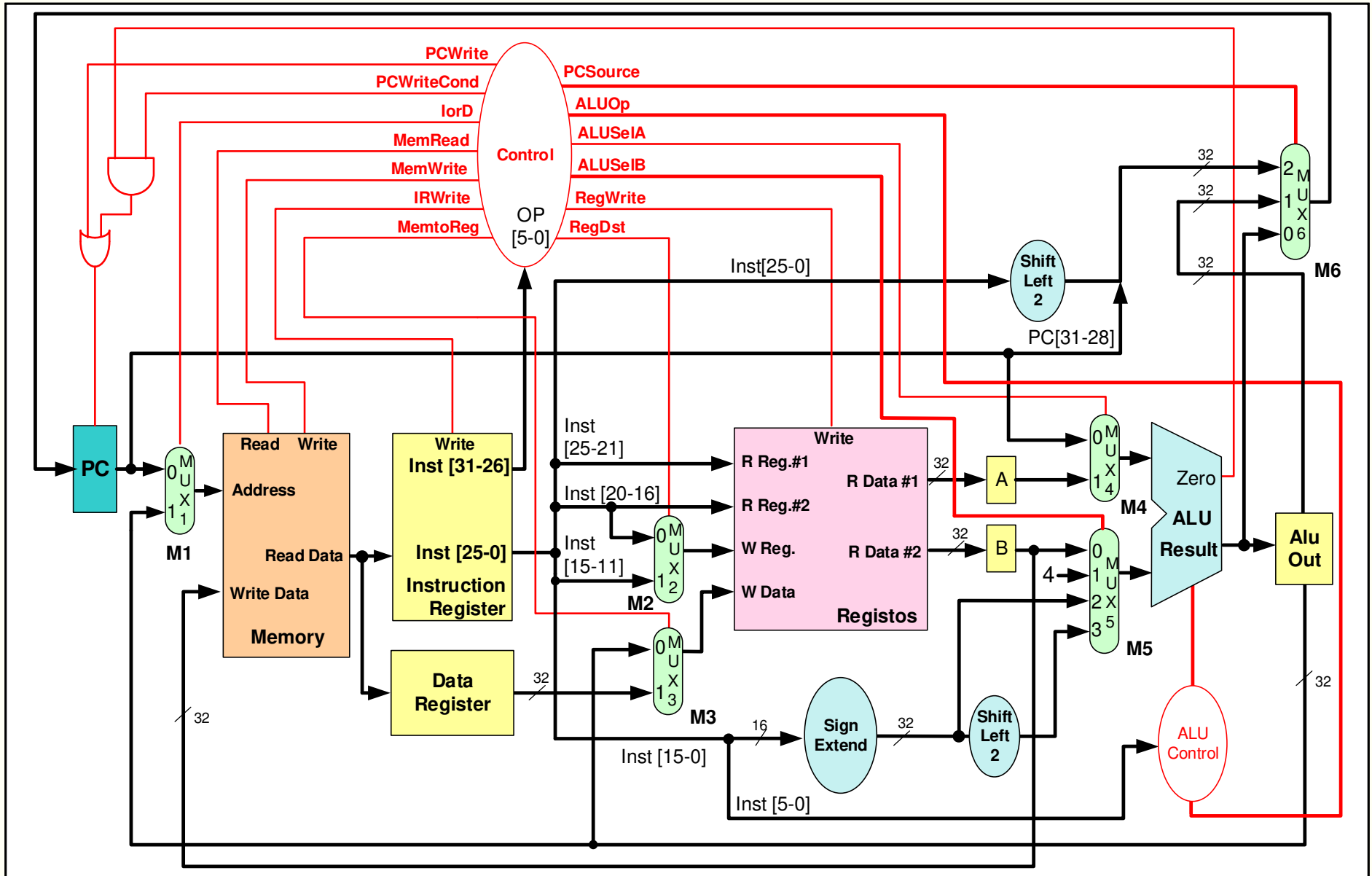
Ações realizadas nas transições ativas do relógio (0→1)



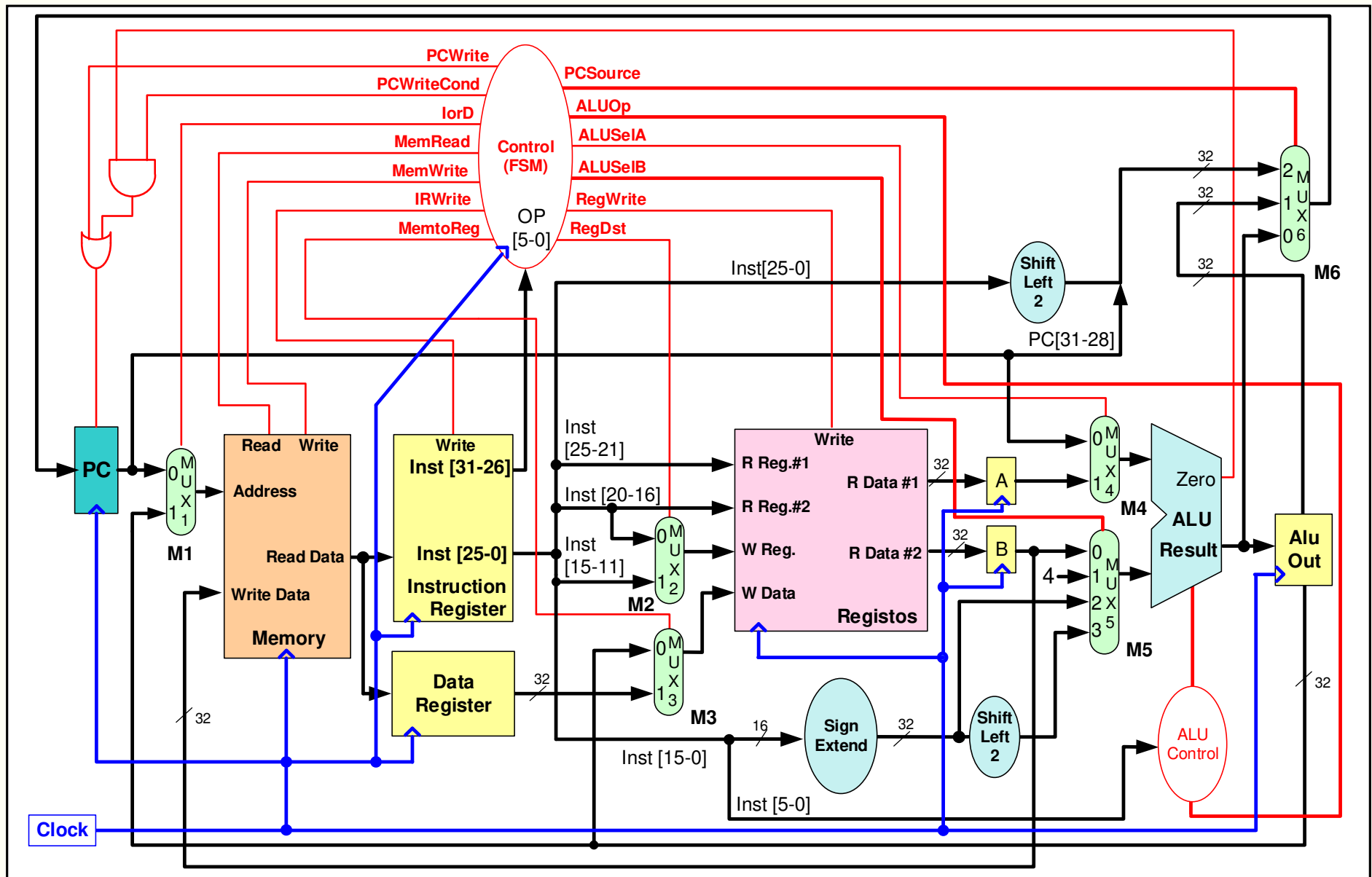
Ações realizadas nas transições ativas do relógio (0→1)

Passo	Ação p/ as R-Type / ADDI / SLTI	Ação p/ instruções que referenciam a memória	Ação p/ os branches
Instruction fetch	$IR = \text{Memory}[PC]$ $PC = PC + 4$		
Instruction decode, register fetch, cálculo do BTA	$A = \text{Reg}[IR[25:21]]$ $B = \text{Reg}[IR[20:16]]$ $ALUOut = PC + (\text{sign extended}(IR[15:0]) \ll 2)$		
Execução (tipo R/addi/slti), cálculo de endereços ou conclusão dos branches	$ALUOut = A \text{ op } B$ ou $ALUOut = A \text{ op extend}(IR[15:0])$	$ALUOut = A + \text{sign-extended}(IR[15:0])$	If (A == B) then PC = ALUOut
Acesso à memória (leitura-LW; ou escrita-SW) ou escrita no File Register (write-back, instruções tipo R/addi/slti)	Tipo R: $\text{Reg}[IR[15:11]] = ALUOut$ ADDI / SLTI: $\text{Reg}[IR[20:16]] = ALUOut$	$MDR = \text{Memory}[ALUOut]$ ou $\text{Memory}[ALUOut] = B$	
Escrita no File Register (write-back, instrução LW)		$\text{Reg}[IR[20:16]] = MDR$	

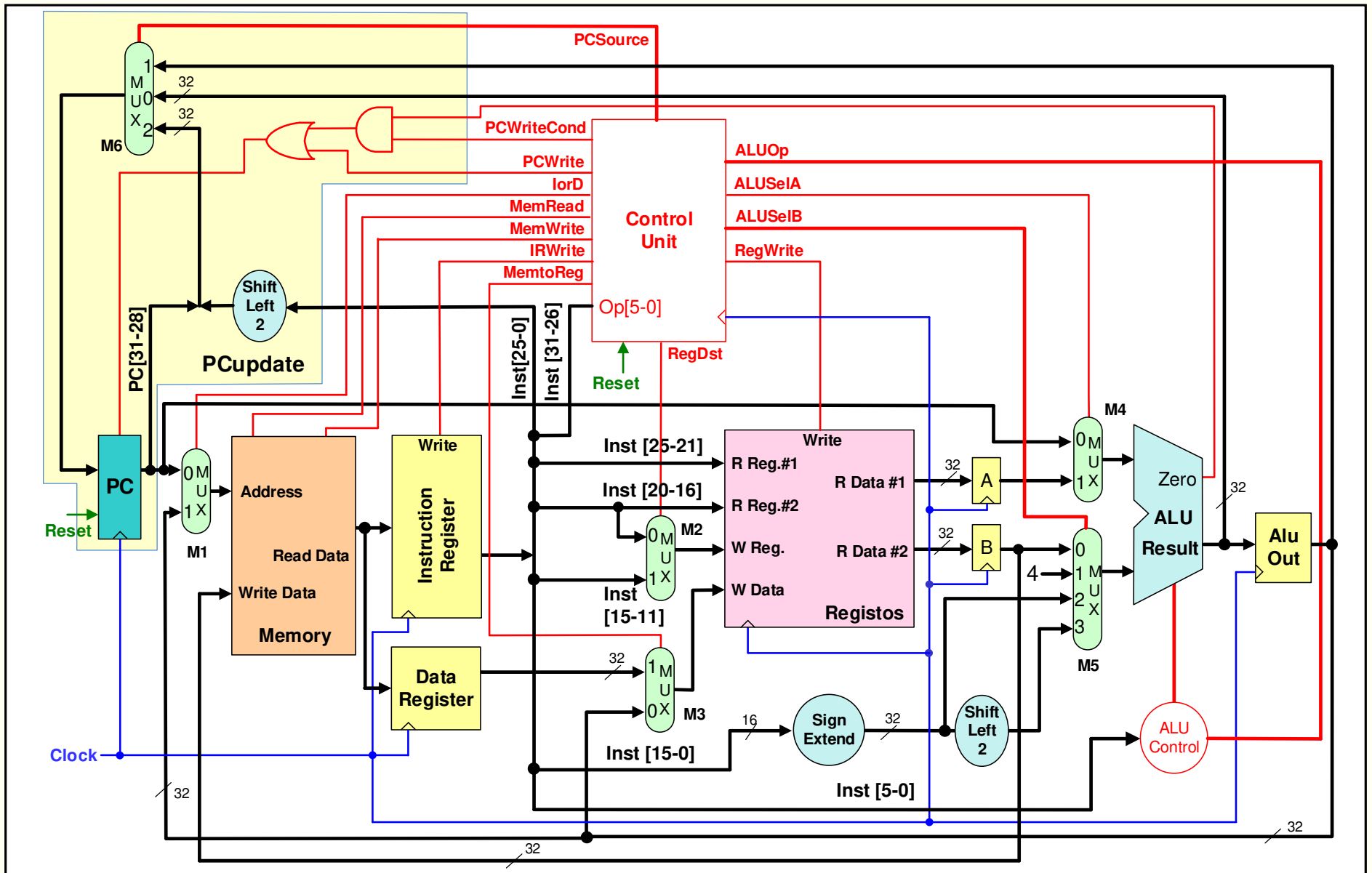
O datapath Multi-cycle completo



O datapath Multi-cycle completo



Módulo de atualização do PC



Módulo de atualização do PC – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

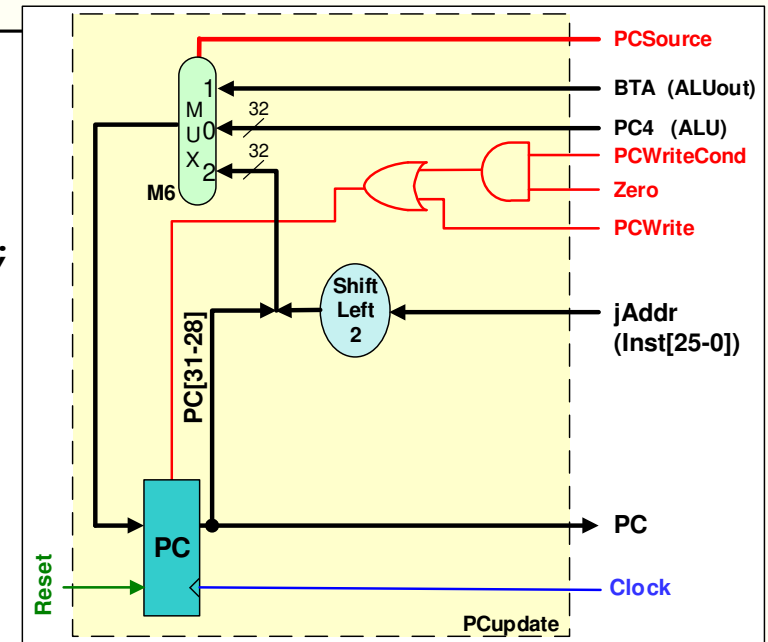
entity PCupdate is
  port (clk      : in  std_logic;
        reset    : in  std_logic;
        zero     : in  std_logic;
        PCSource : in  std_logic_vector(1 downto 0);
        PCWrite  : in  std_logic;
        PCWriteCond : in std_logic;
        PC4      : in  std_logic_vector(31 downto 0);
        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;

```



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 de cor → Usados no contexto do ciclo da instrução
 - fundo branco → Não usados no contexto do ciclo da instrução
- Elementos de estado:
 - fundo de cor com textura → Escritos no final do ciclo de relógio corrente

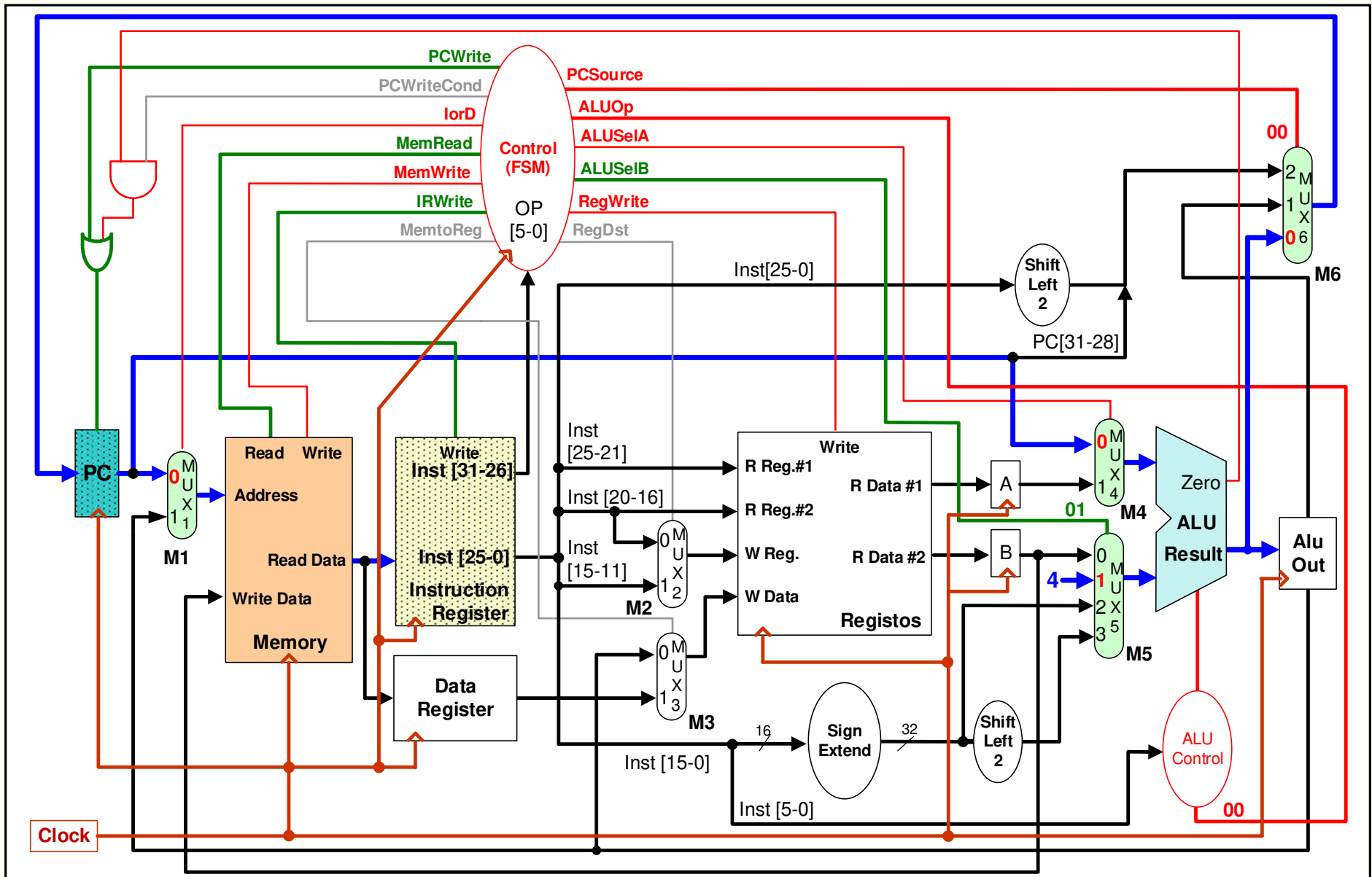
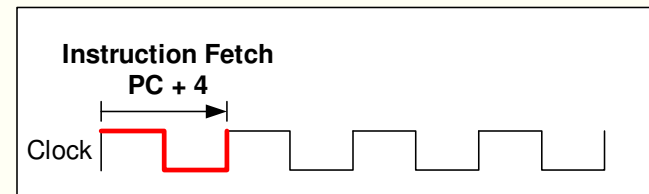
Funcionamento do *datapath* nas instruções do tipo R

- 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*

Exemplo: add \$5,\$8,\$6

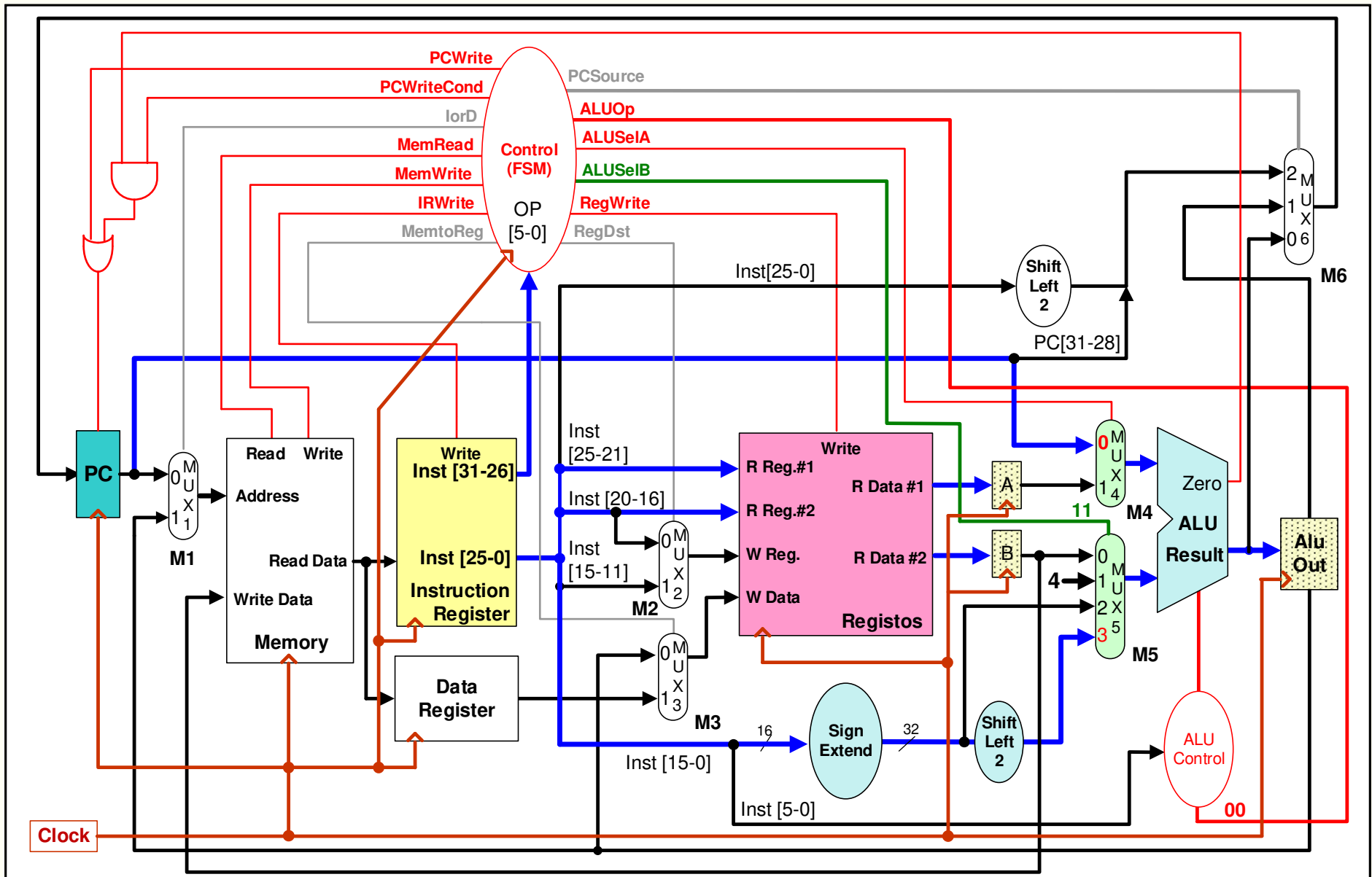
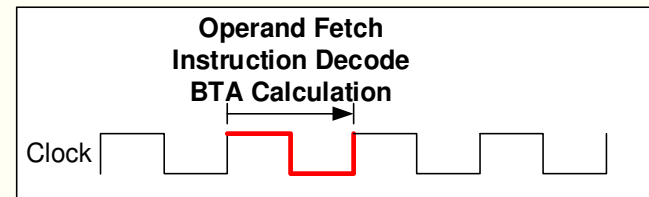
Instruções do tipo R

Fase 1



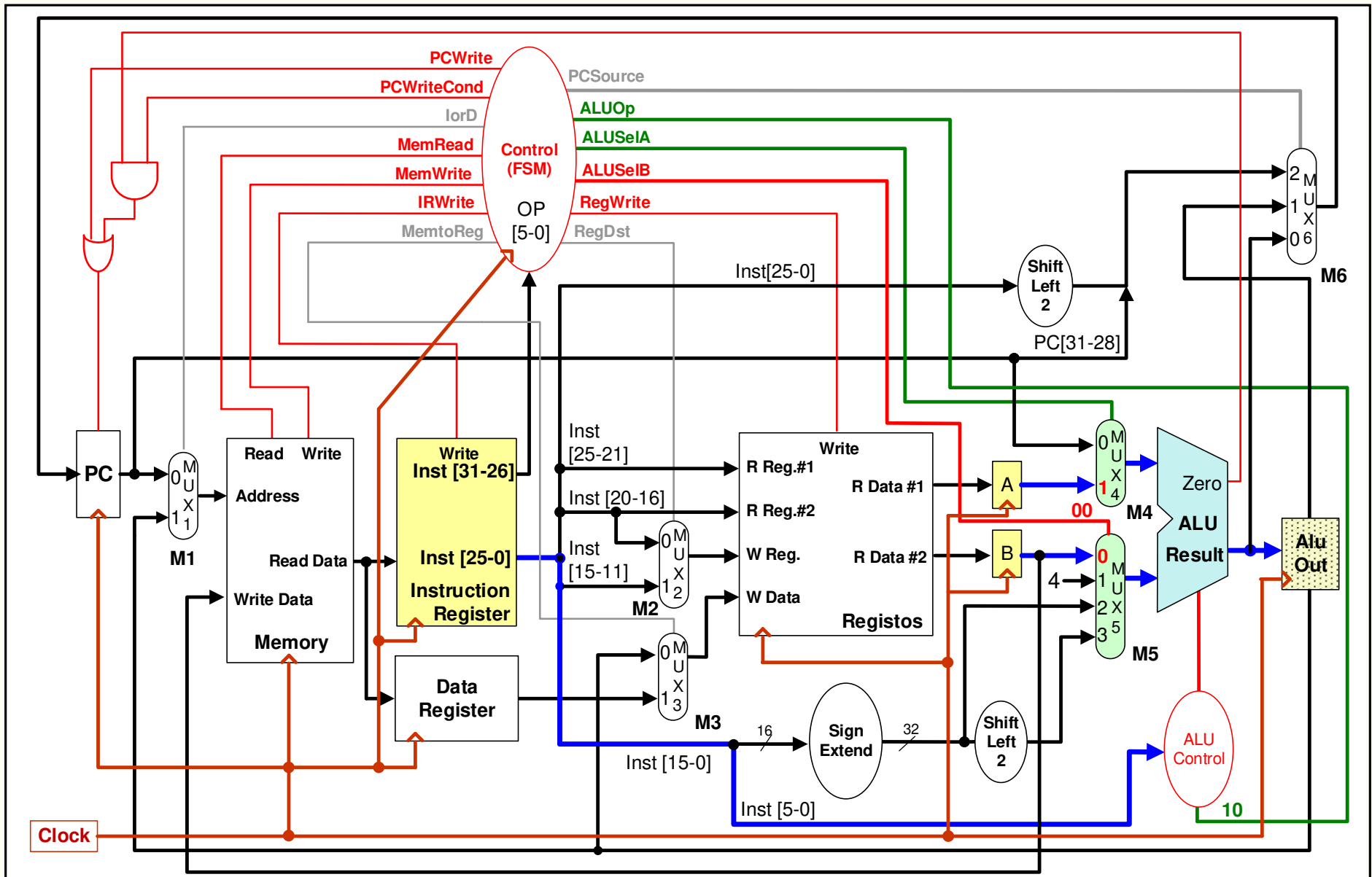
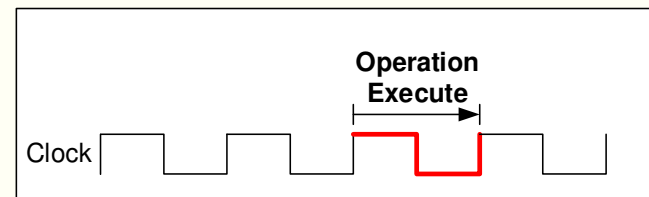
Instruções do tipo R

Fase 2



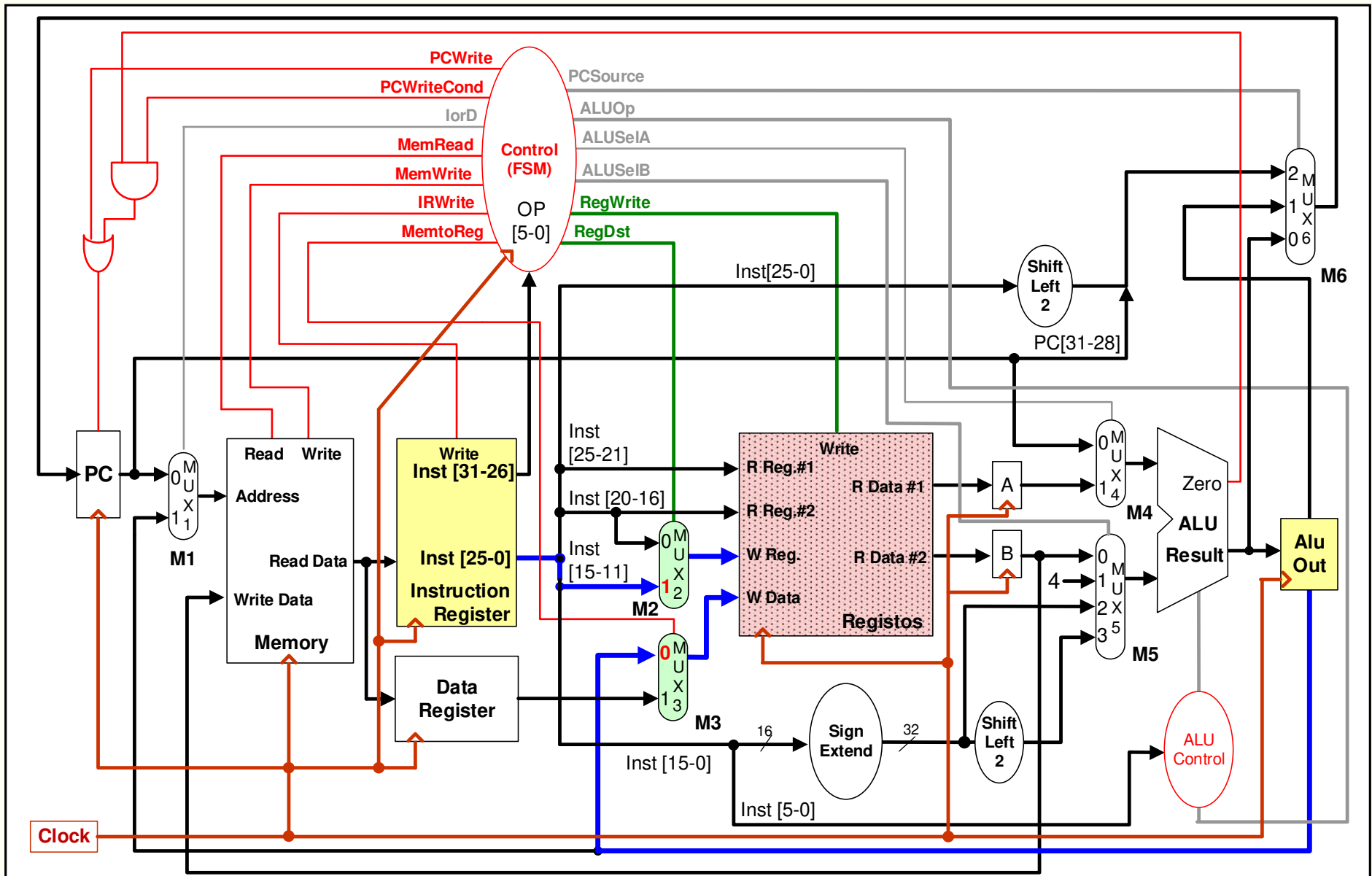
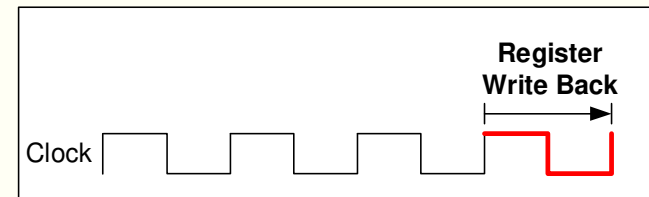
Instruções do tipo R

Fase 3



Instruções do tipo R

Fase 4



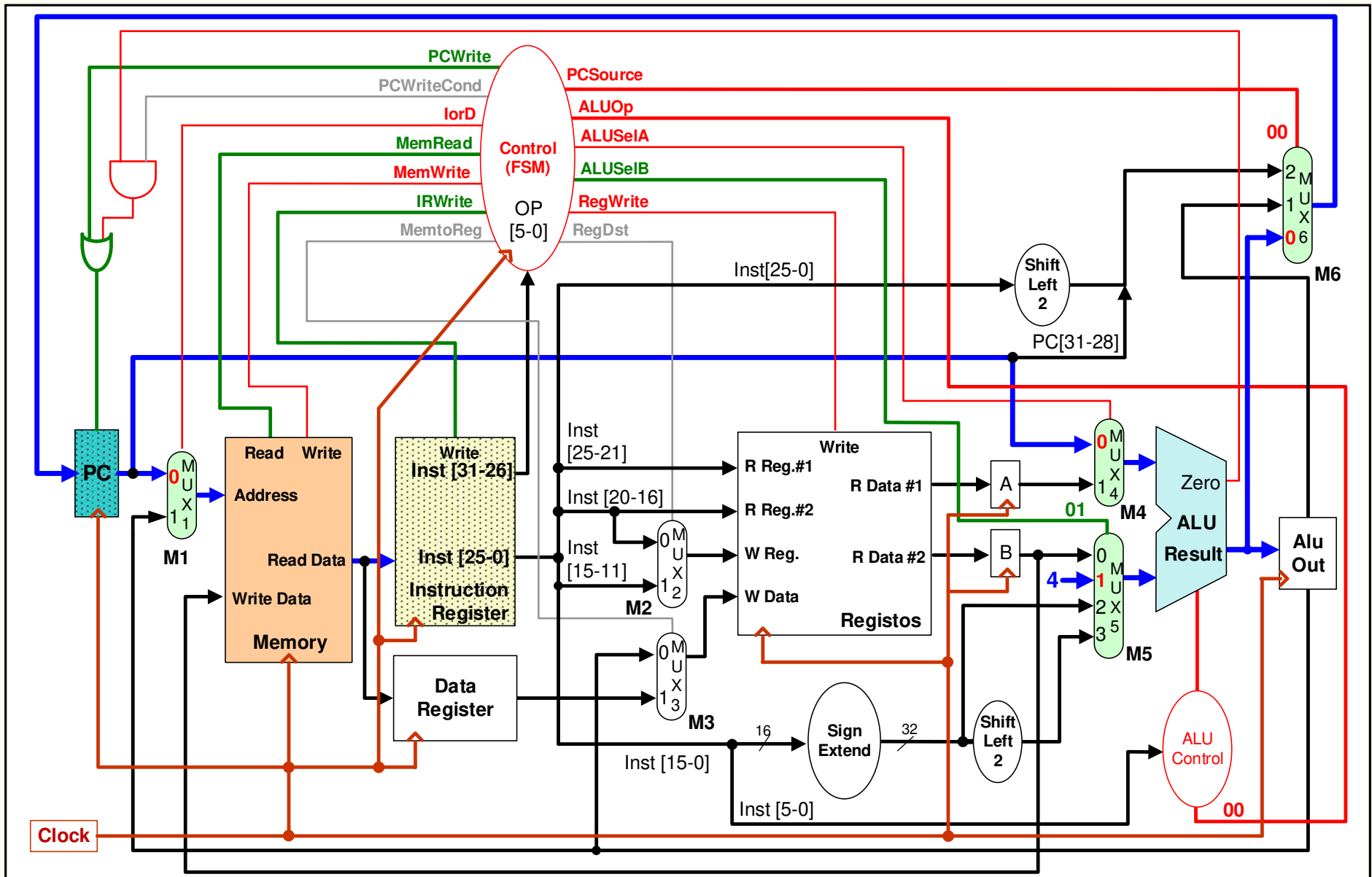
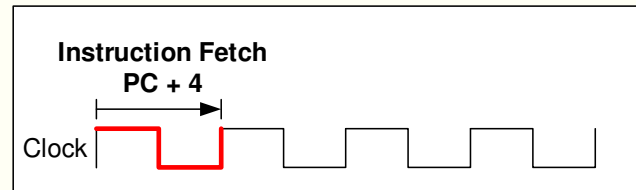
Funcionamento do *datapath* na instrução LW

- Fase 1:
 - *Instruction fetch*
 - Cálculo de PC+4
- Fase 2:
 - Leitura dos registros
 - 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*

Exemplo: lw \$3,0x0014(\$6)

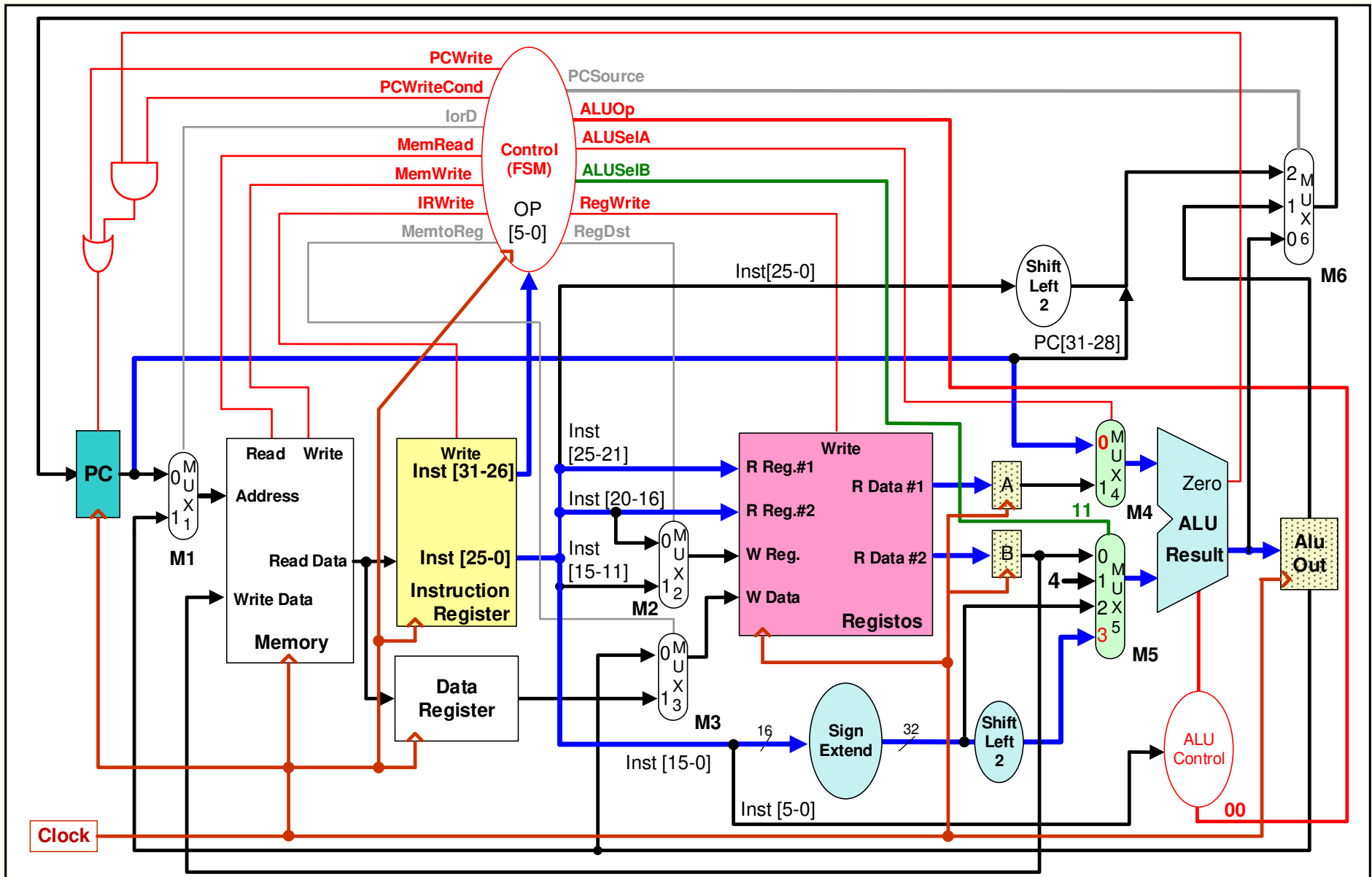
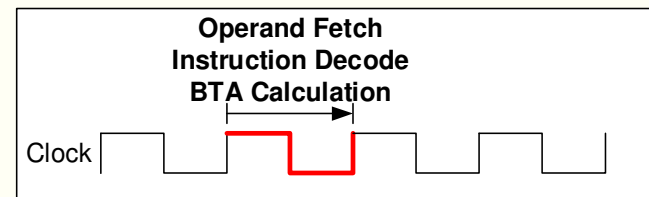
Instrução LW

Fase 1



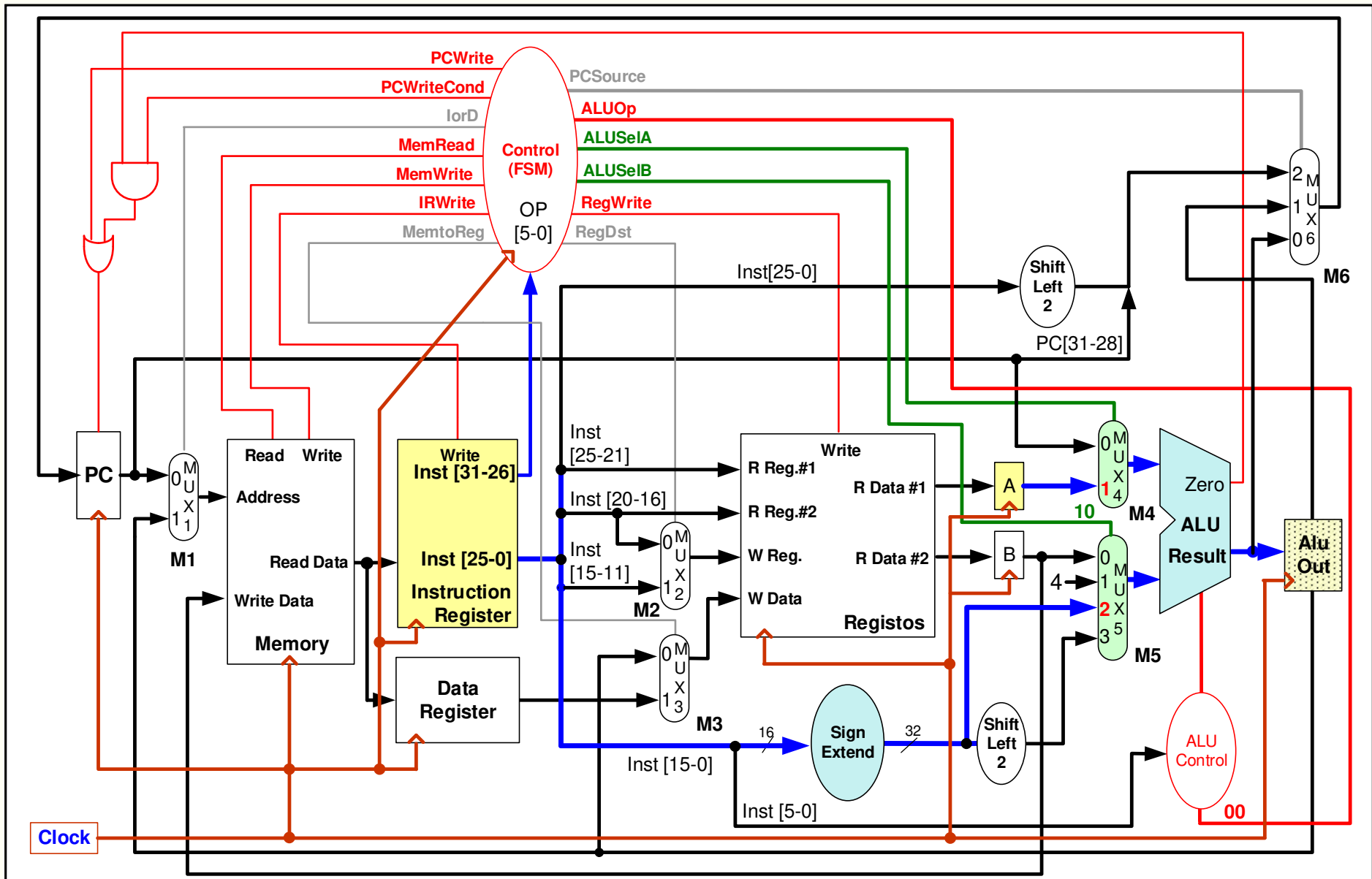
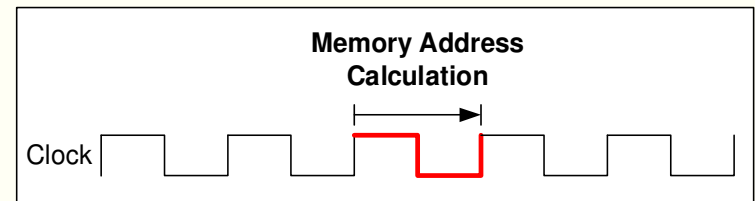
Instrução LW

Fase 2



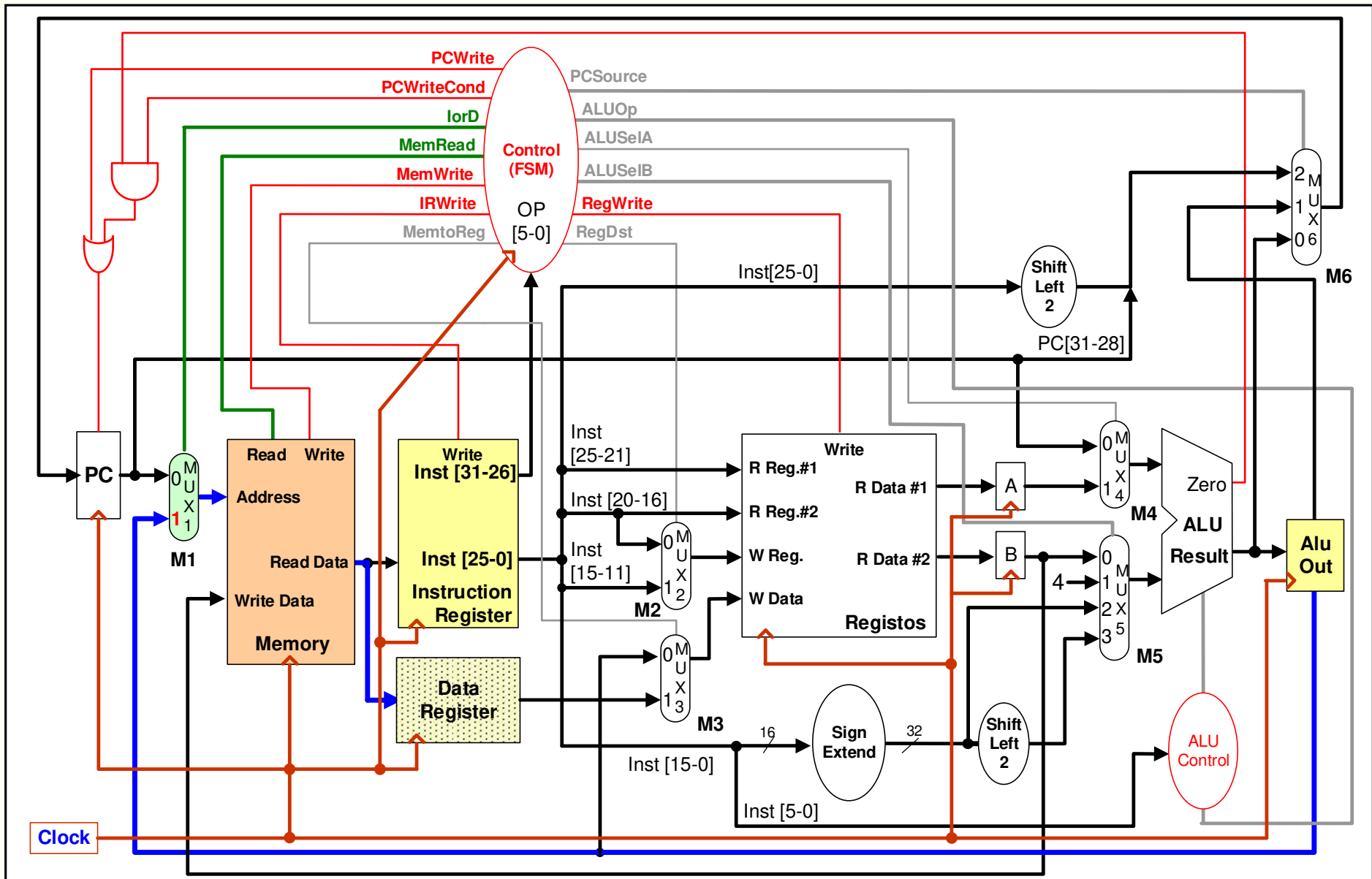
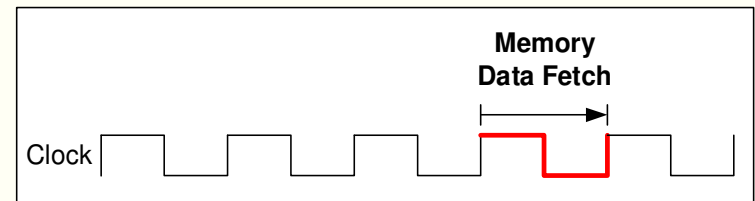
Instrução LW

Fase 3



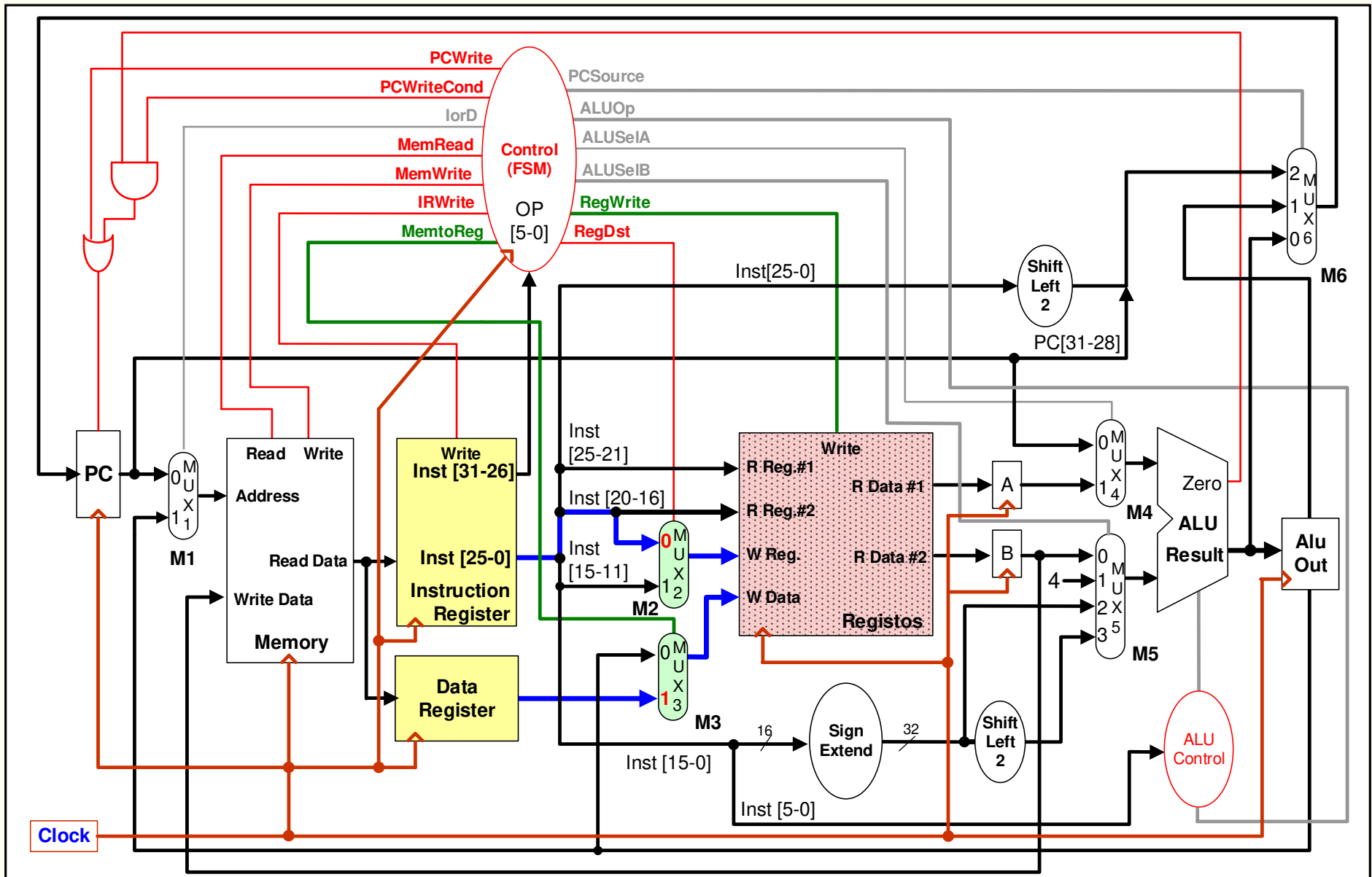
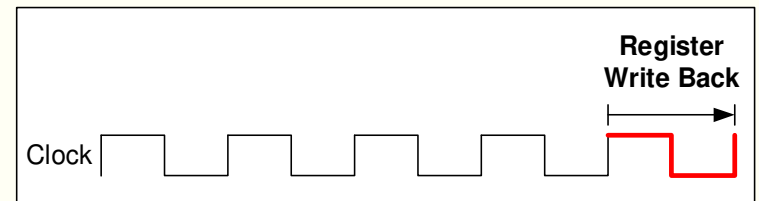
Instrução LW

Fase 4



Instrução LW

Fase 5



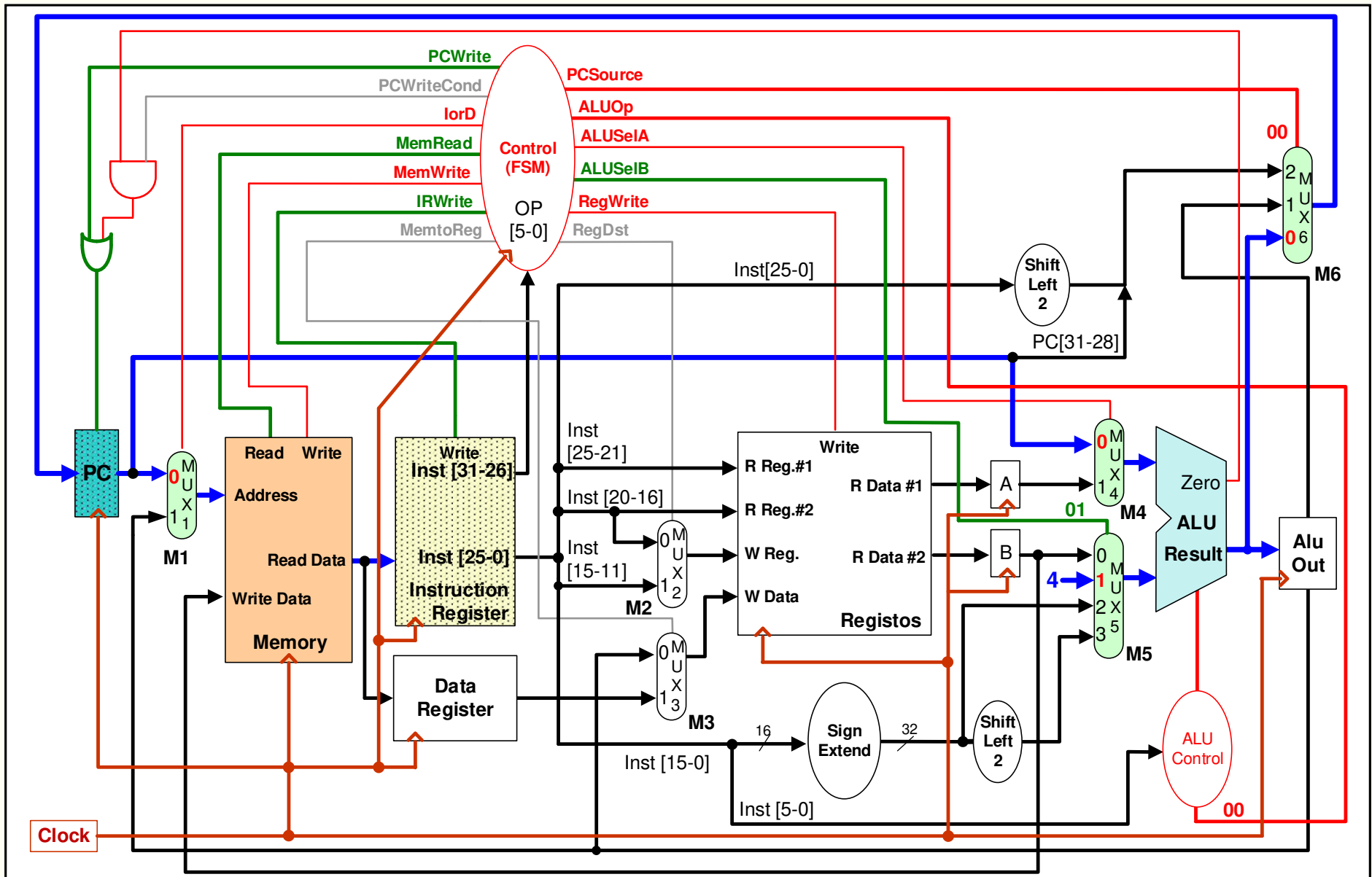
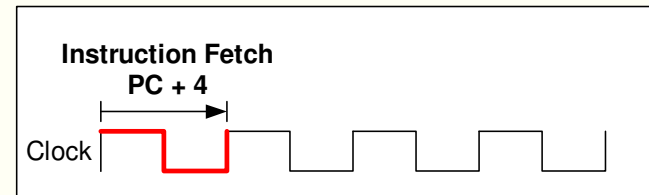
Funcionamento do *datapath* na instrução BEQ

- 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

Exemplo: beq \$3,\$6,endif

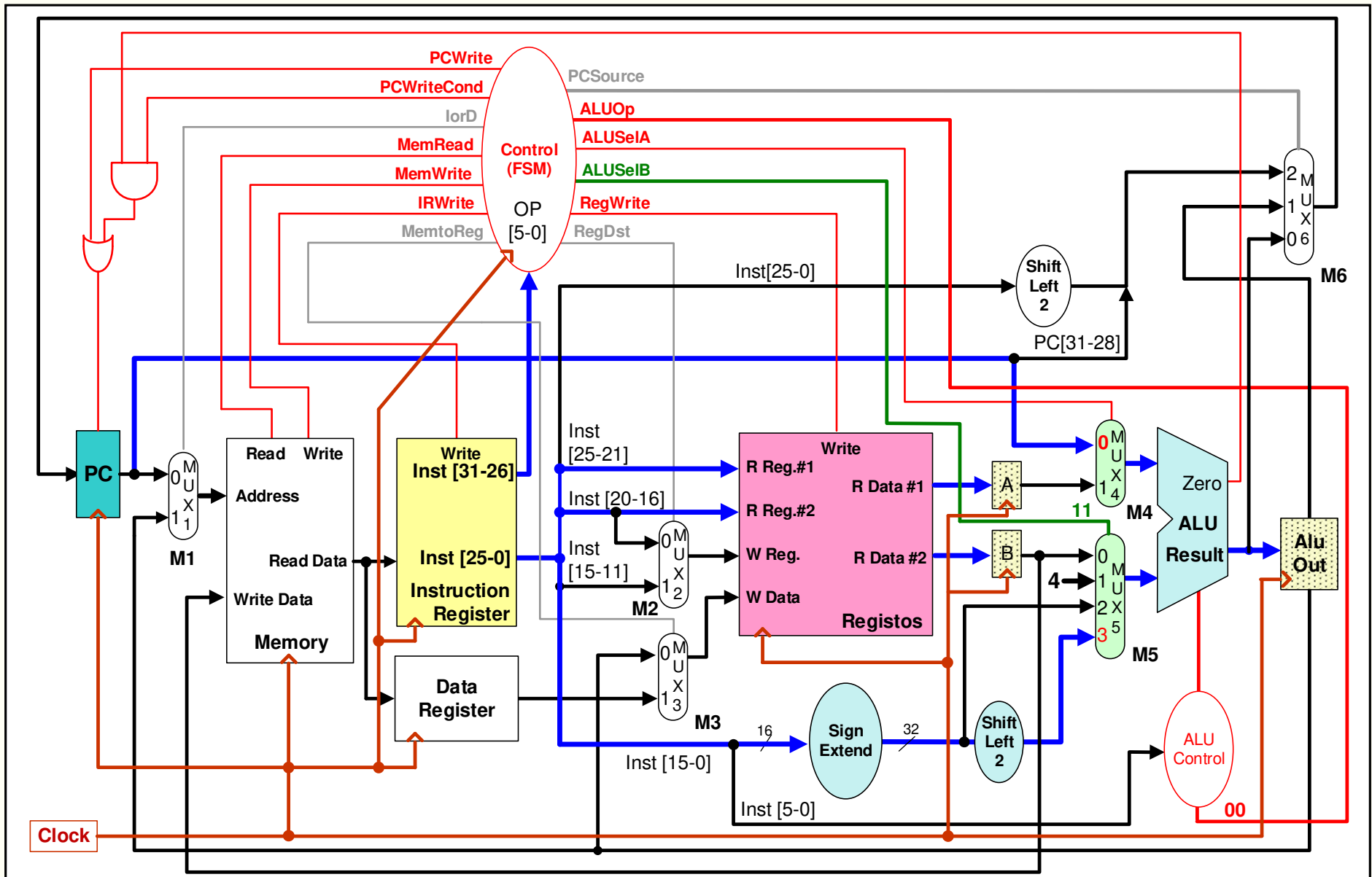
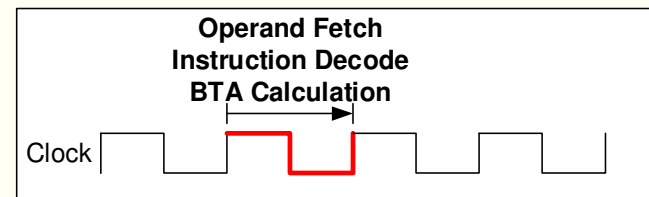
Instrução BEQ

Fase 1



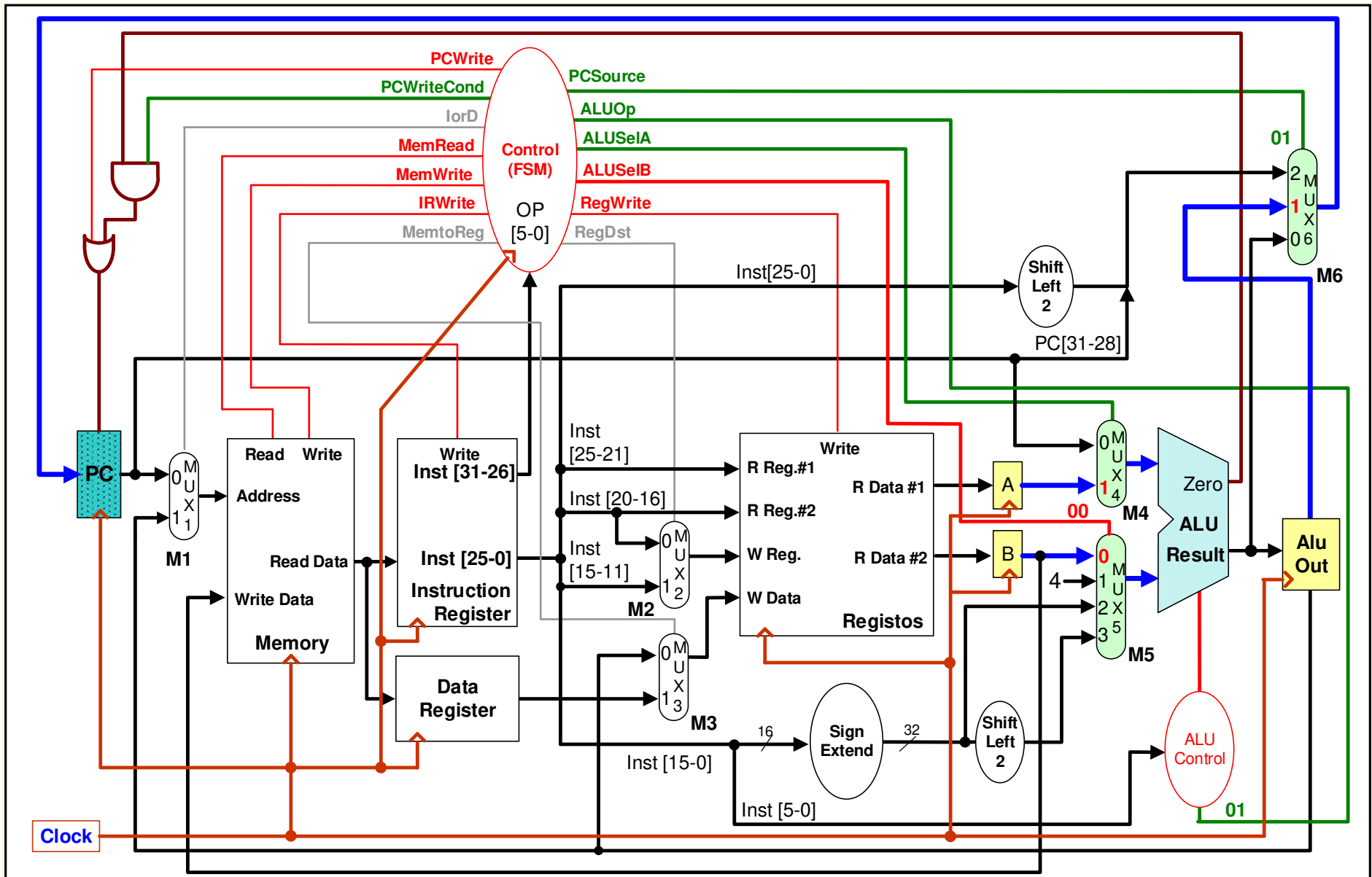
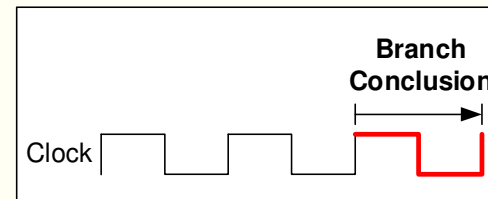
Instrução BEQ

Fase 2



Instrução BEQ

Fase 3



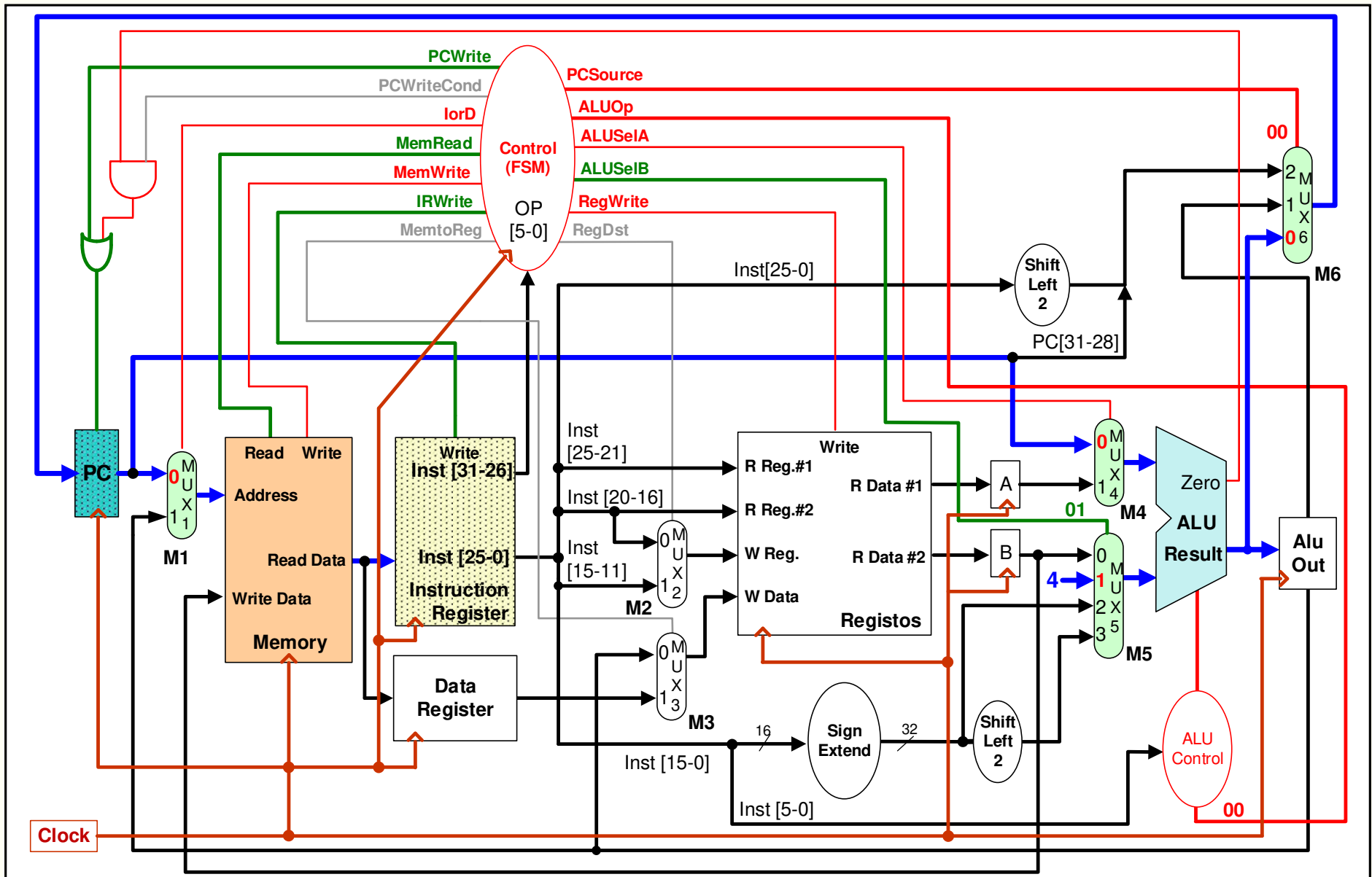
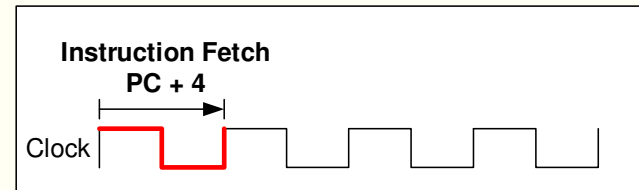
Funcionamento do *datapath* na instrução J

- 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

Exemplo: j loop

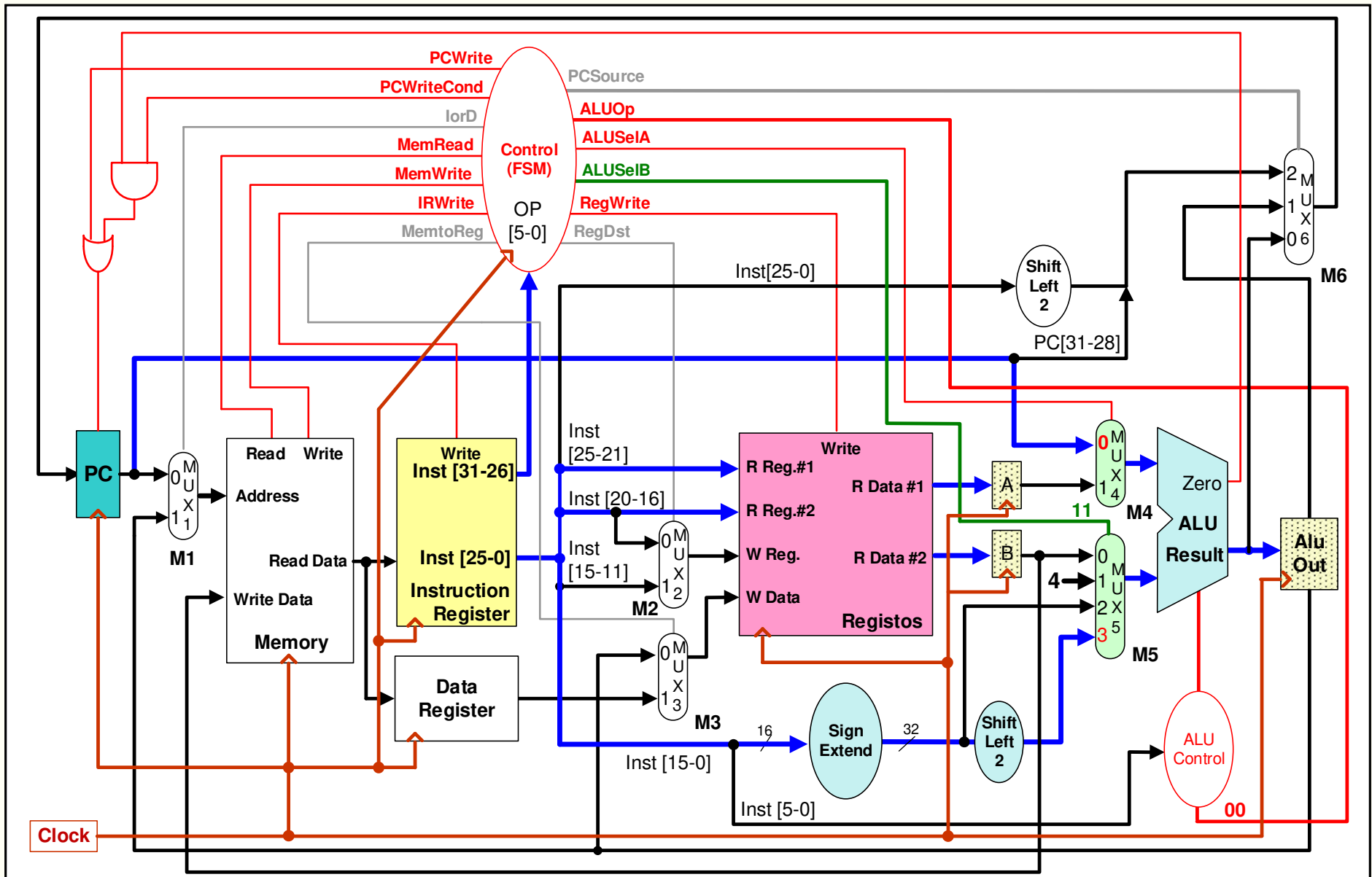
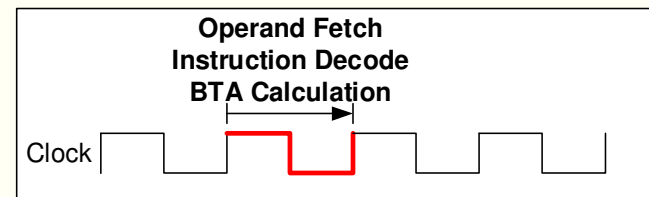
Instrução J

Fase 1



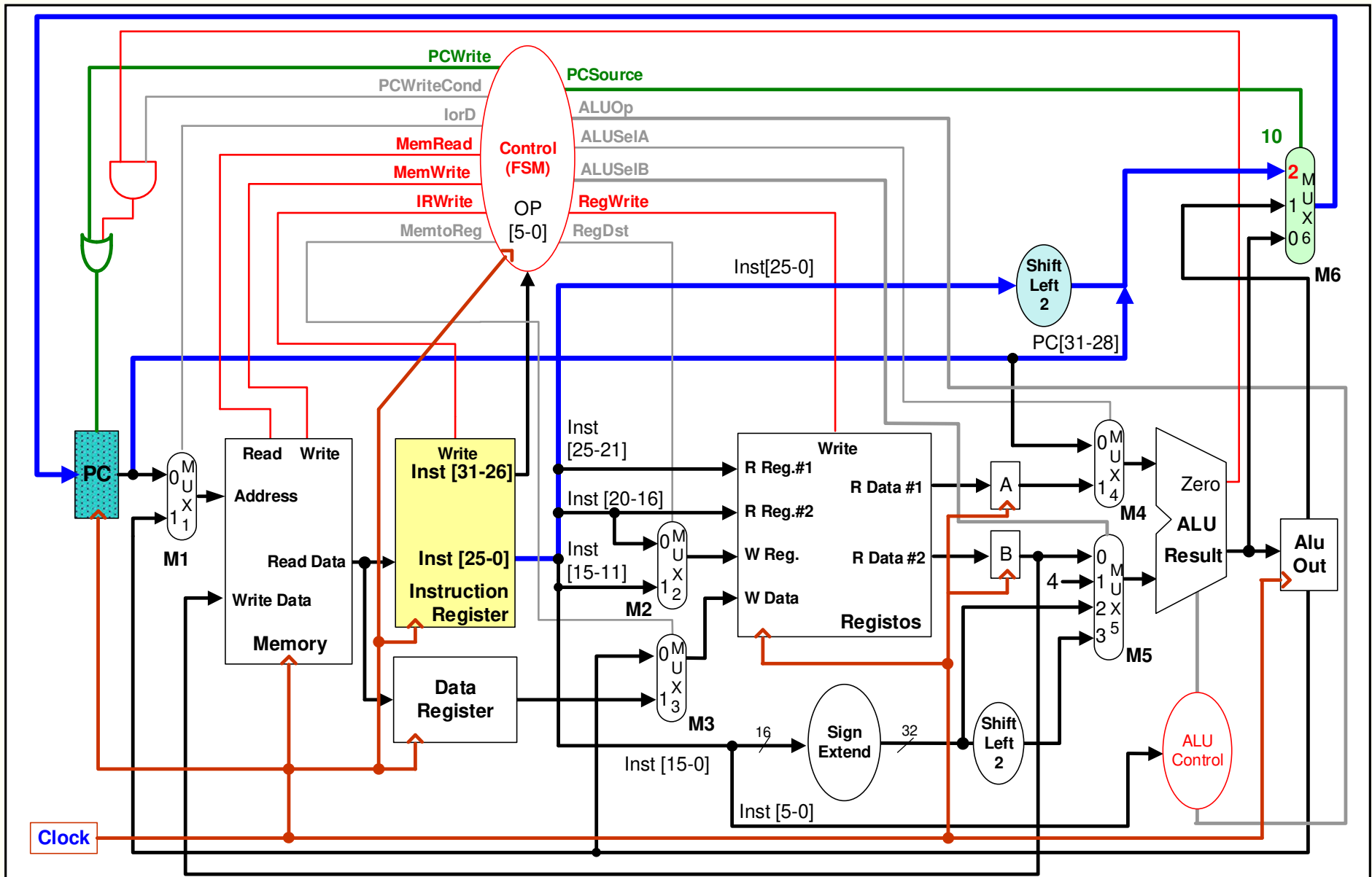
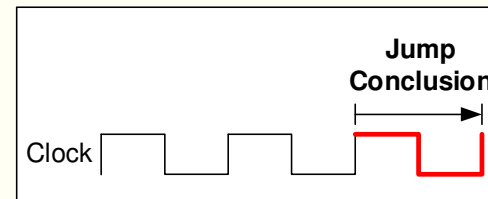
Instrução J

Fase 2



Instrução J

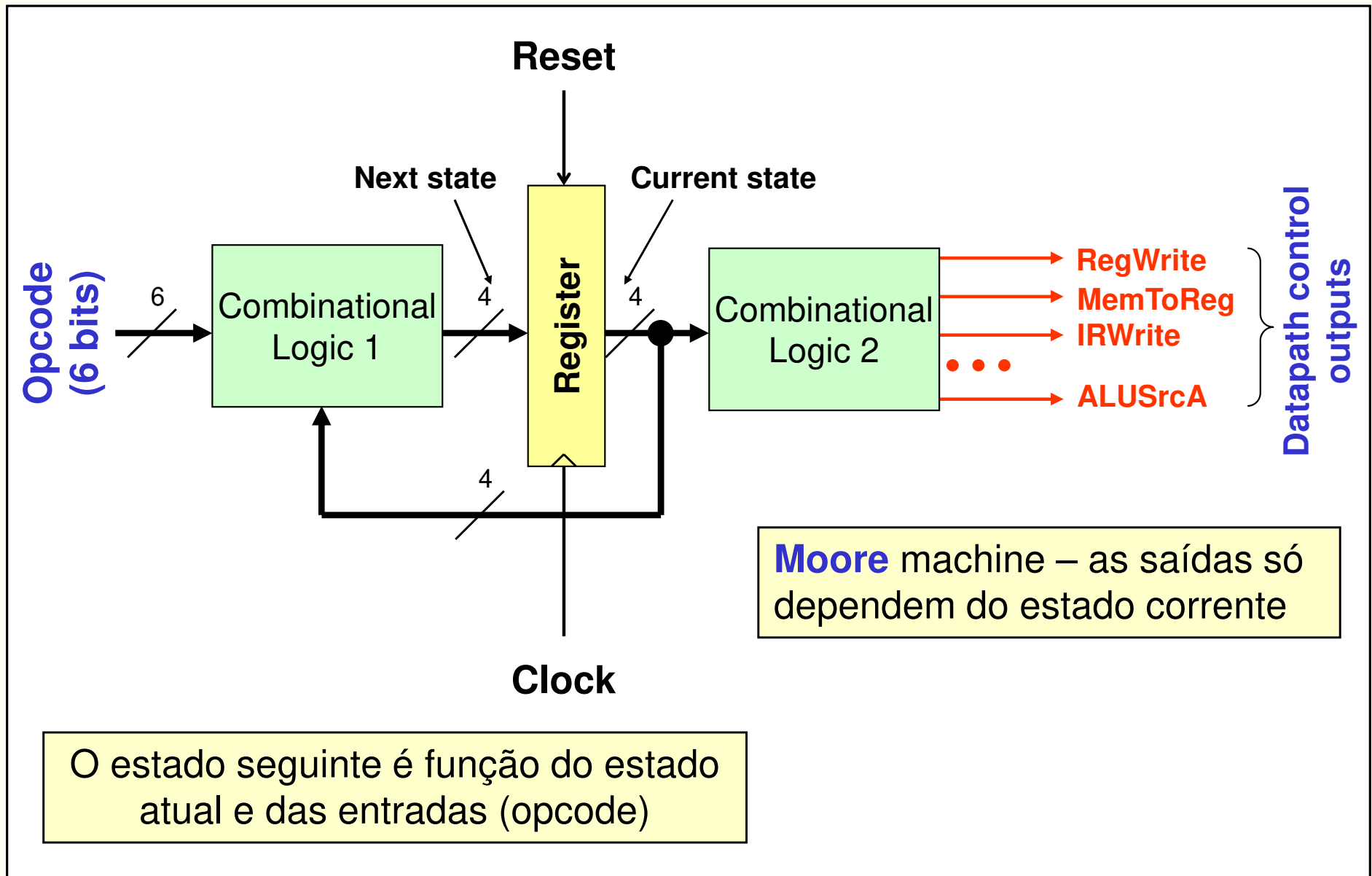
Fase 3



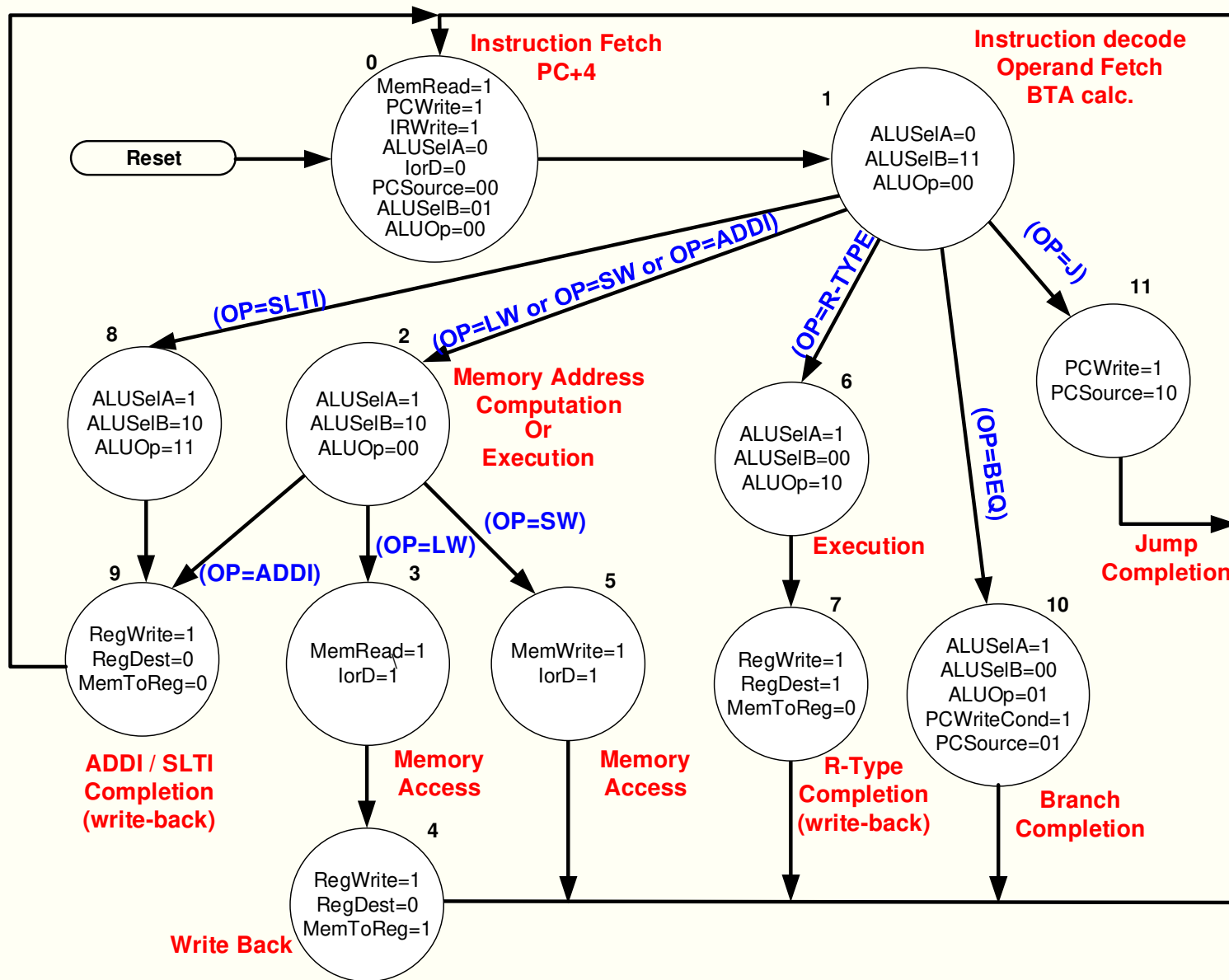
A unidade de controlo do *datapath Multi-cycle*

- 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:
 - os sinais de controlo diferem de ciclo de relógio para ciclo de relógio e após o segundo ciclo diferem de instrução para instrução.
 - 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 sinais de saída não explicitados em cada estado ou 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* - 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;

```

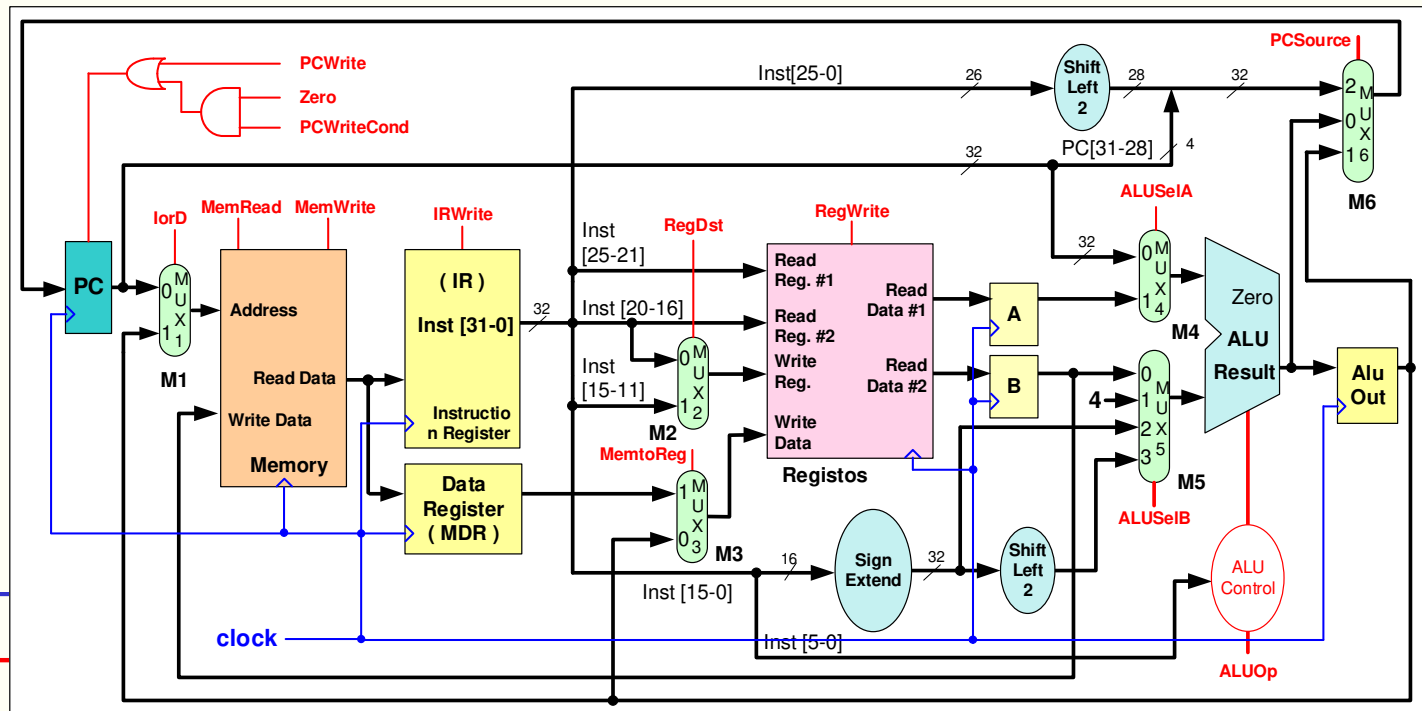
Processo combinatório

Funcionamento do DP

add	\$2, \$3, \$4
sw	\$2, -4(\$6)
or	\$4, \$6, \$3

Sinais de controlo na execução sequencial das três instruções:

add \$2, \$3, \$4



PCWriteCond	0	X	0	0	0	X	0
PCWrite	0	1	0	0	0	1	0
MemWrite	0	0	0	0	1	0	0
MemRead	0	1	0	0	0	1	0
MemToReg	0	X	X	X	X	X	X
IRWrite	0	1	0	0	0	1	0
ALUSelA	X	0	0	1	X	0	0
ALUSelB	XX	01	11	10	XX	01	11
ALUOp	XX	00	00	00	XX	00	00
IorD	X	0	X	X	1	0	X
PCSource	XX	00	XX	XX	XX	00	XX
RegWrite	1	0	0	0	0	0	0
RegDst	1	X	X	X	X	X	X

DETI-UA

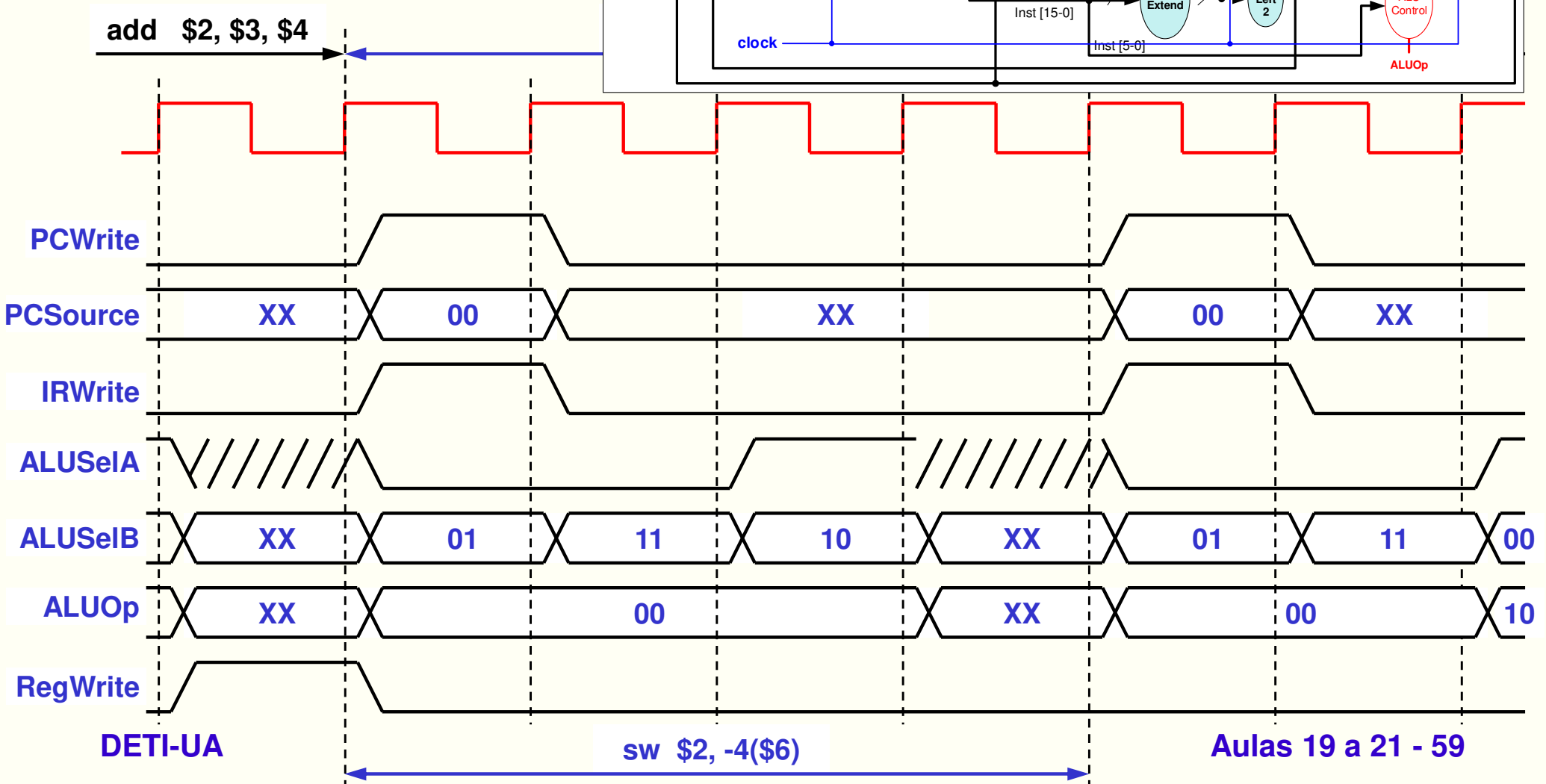
sw \$2, -4(\$6)

Aulas 19 a 21 - 58

Funcionamento do DP

add	\$2, \$3, \$4
sw	\$2, -4(\$6)
or	\$4, \$6, \$3

Sinais de controlo: diagrama temporal



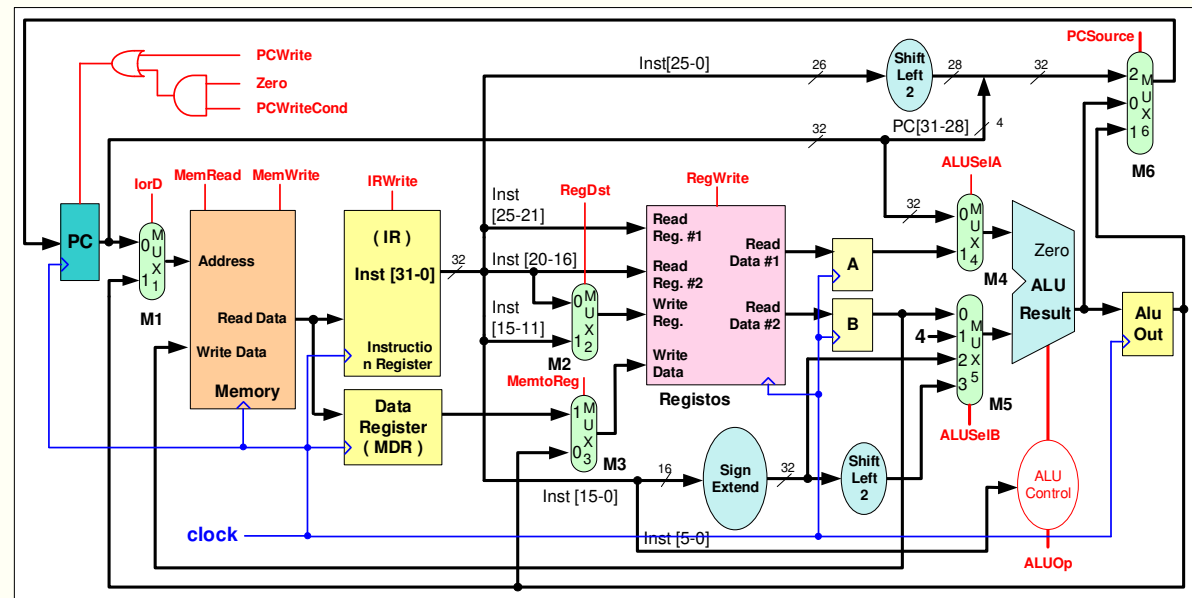
Funcionamento do DP

00400048 add \$2, \$3, \$4 # 00641020
 0040004C sw \$2, -4(\$6) # ACC2FFFC
 00400050 or \$4, \$6, \$3 # 00C32025

(valores em hexadecimal)

\$3	20001FA6
\$4	81002378
\$6	10012480

Valores calculados /
 obtidos em cada ciclo
 de relógio:



add \$2, \$3, \$4

sw \$2, -4(\$6)

or \$4, \$6, \$3

PC									
IR									
MDR									
ALU Res									
ALU Out									
ALU Zero									
A									
B									
ALUOp									

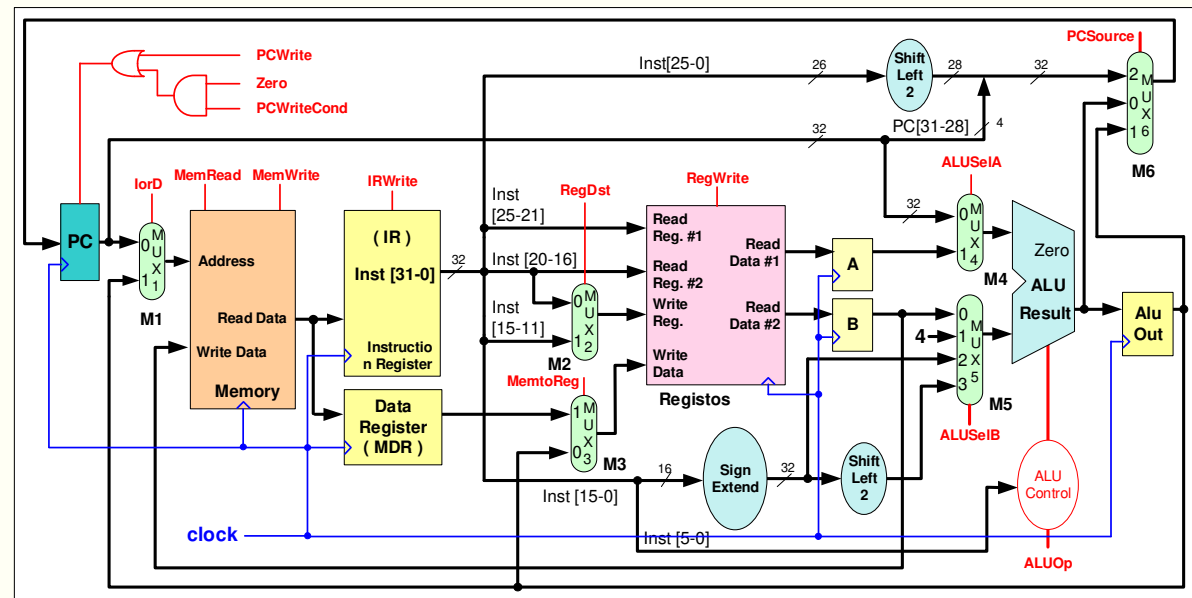
Funcionamento do DP

00400048 add \$2, \$3, \$4 # 00641020
 0040004C sw \$2, -4(\$6) # ACC2FFFC
 00400050 or \$4, \$6, \$3 # 00C32025

(valores em hexadecimal)

\$3	20001FA6
\$4	81002378
\$6	10012480

Valores calculados /
 obtidos em cada ciclo
 de relógio:



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

DETI-UA

Opcodes: SW - 0x2B, ADD - 0x20, OR - 0x25

Aulas 19 a 21 - 61

Exercícios

- Considere um programa que executa em 10s num computador "A" com uma frequência de 4GHz. Pretende-se desenvolver um computador "B" que execute o programa em 6s. O hardware *designer* verificou que é possível um aumento da frequência de trabalho do CPU do computador "B", mas isso acarreta um acréscimo do número total de ciclos de relógio de 1,2 vezes relativamente a A. Qual a frequência de trabalho que deverá ter o CPU da máquina "B"?
- Considere duas máquinas com implementações distintas da mesma arquitetura do conjunto de instruções (ISA). Para um dado programa,
 - Máquina A: Clock_cycle = 350 ps; CPI = 2,0
 - Máquina B: Clock_cycle = 400 ps; CPI = 1,5Qual a máquina mais rápida? Qual a relação de desempenho?
- Considere duas máquinas ("A" e "B") com implementações distintas da mesma arquitetura do conjunto de instruções (ISA). Para um mesmo programa, a máquina "A" apresenta um CPI de 2,0 e a "B" de 3,125. Usando a métrica tempo de execução, verificou-se que a máquina "A" é mais rápida que a máquina "B" por um fator de 1,25. Calcule a relação entre as frequências de relógio das máquinas "A" e "B".

Exercícios

- Considerando os seguintes tempos de atraso dos elementos operativos do *datapath single-cycle* que estudou:
 - acesso à memória para leitura: 5ns; acesso à memória para preparar a escrita: 2ns; acesso ao *register file* para leitura: 3ns; acesso ao *register file* para preparar a escrita: 2ns; operação da ALU: 4ns; operação de um somador: 2ns; unidade de controlo: 2ns; tempo de *setup* do PC: 1ns; extensor de sinal: 1ns; *left shifter*: 1ns; multiplexers: 0ns;
 - Q1: calcule o tempo mínimo de execução para cada uma das instruções suportadas.
 - Q2: calcule a frequência máxima de funcionamento do *datapath single-cycle*.
- O que limita a frequência máxima do relógio do *datapath multi-cycle*?

Exercícios

- Quantos ciclos de relógio demora, no *datapath multi-cycle*, a execução de cada uma das instruções consideradas (r-type, lw, sw, addi, slti, beq e j)?
- Para os tempos de atraso apresentados no exercício anterior, qual a frequência máxima de funcionamento do *datapath multi-cycle*?
- Considere um programa com 100.000 instruções, com o seguinte padrão: 10% de lw, 10% de sw, 60% de tipo R, 10% de addi/slti, 5% de branches e 5% de jumps. Usando os valores de frequência que calculou anteriormente, determine o tempo de execução desse programa: a) num *datapath single-cycle*; b) num *datapath multi-cycle*. Calcule, para esse programa, o ganho de desempenho da arquitetura *multi-cycle* relativamente à arquitetura *single-cycle*.

Exercícios

- Calcule o número de ciclos de relógio que o programa seguinte demora a executar, desde o *Instruction Fetch* da 1ª instrução até à conclusão da última instrução, tendo em atenção os valores da memória de dados apresentados:

1) num *datapath single-cycle*, 2) num *datapath multi-cycle*

main:

lw \$1, 0(\$0)

add \$4, \$0, \$0

lw \$2, 4(\$0)

loop:

lw \$3, 0(\$1)

add \$4, \$4, \$3

sw \$4, 36(\$1)

addi \$1, \$1, 4

slt \$5, \$1, \$2

bne \$5, \$0, loop

sw \$4, 8(\$0)

lw \$1, 12(\$0)

Memória de dados

Address	Value
0x00000000	0x10
0x00000004	0x20

Exercícios

- Calcule o número de ciclos de relógio que o programa seguinte demora a executar, desde o *Instruction Fetch* da 1ª instrução até à conclusão da última instrução, tendo em atenção os valores da memória de dados apresentados:

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;
    addi    $1, $1, 4                #     p1++;
    slt     $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

Exercícios

- Suponha que no endereço de memória **0x00400038** está armazenada a instrução **"lw \$5, -12(\$7)"**; considere ainda que o conteúdo dos registros **\$5** e **\$7** é, respetivamente, **0x10013CA4** e **0x10010098**. Calcule os valores que estão disponíveis à saída do registo **"ALUOut"** durante as 2^a, 3^a e 4^a fases de execução dessa instrução.
- Preencha as tabelas dos slides 57 e 59 para a execução da instrução **"xor \$10, \$3, \$17"**, supondo que está armazenada no endereço **0x004000A0** e que o valor dos registos é: **\$10=0xF3A431**, **\$3=0xA1234**, **\$17=0xFF0C8**.
- Complete o código VHDL da unidade de controlo apresentado nos slides 55 e 56 para todas as instruções definidas.

Aulas 22 a 26

- *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 simplificadas de *forwarding* e *stalling*

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

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

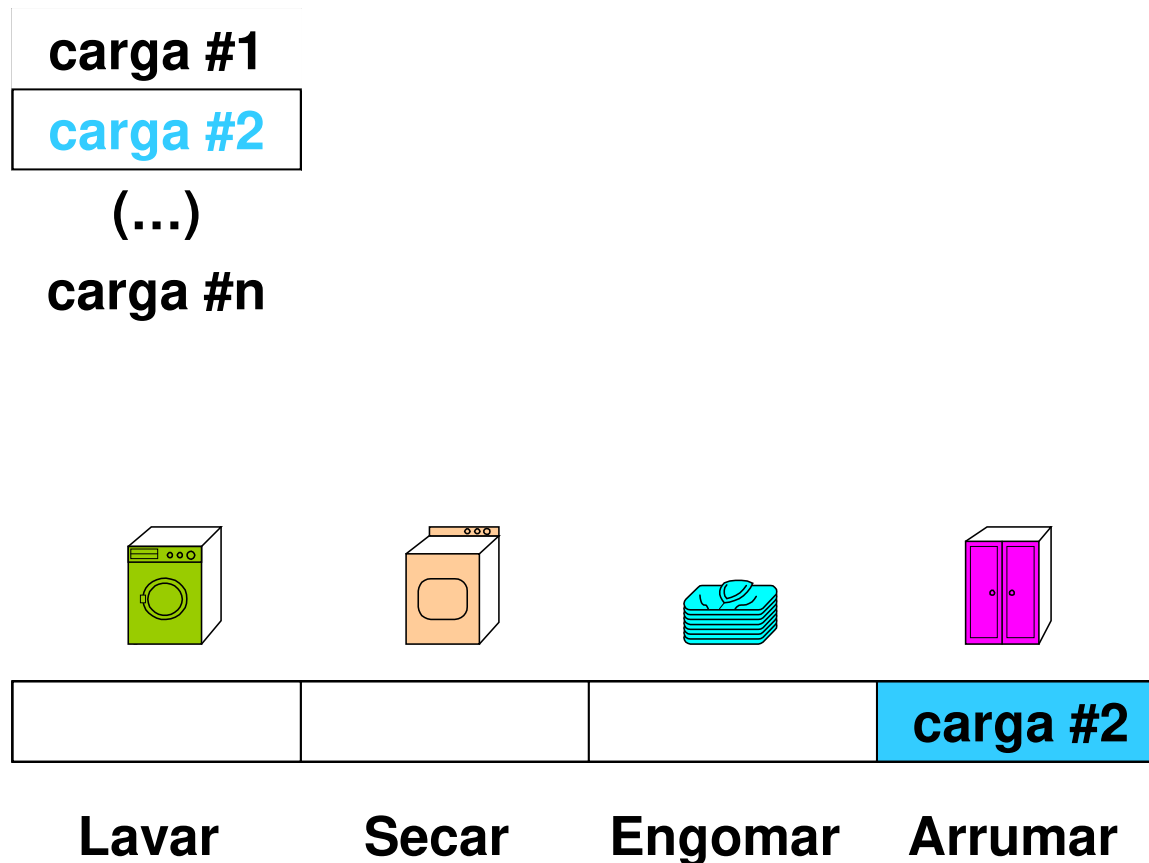
- O exemplo de *pipelining* que iremos observar de seguida apoia-se num conjunto de tarefas simples e intuitivas: o processo de tratamento da roupa suja 😊



- 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

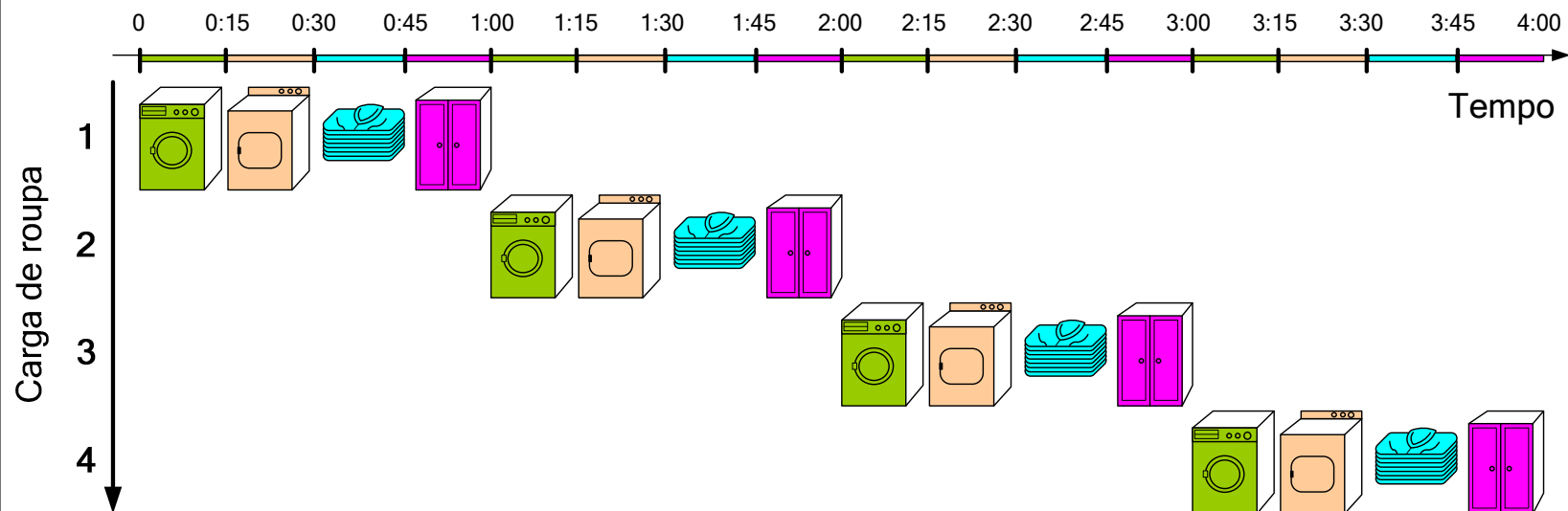
Pipelining - exemplo por analogia

- Numa versão não *pipelined*, o processamento de N cargas de roupa seria, para cada carga: (Lavar > Secar > Engomar > Arrumar)



Pipelining - exemplo por analogia

- Este processo pode então ser descrito temporalmente do seguinte modo:



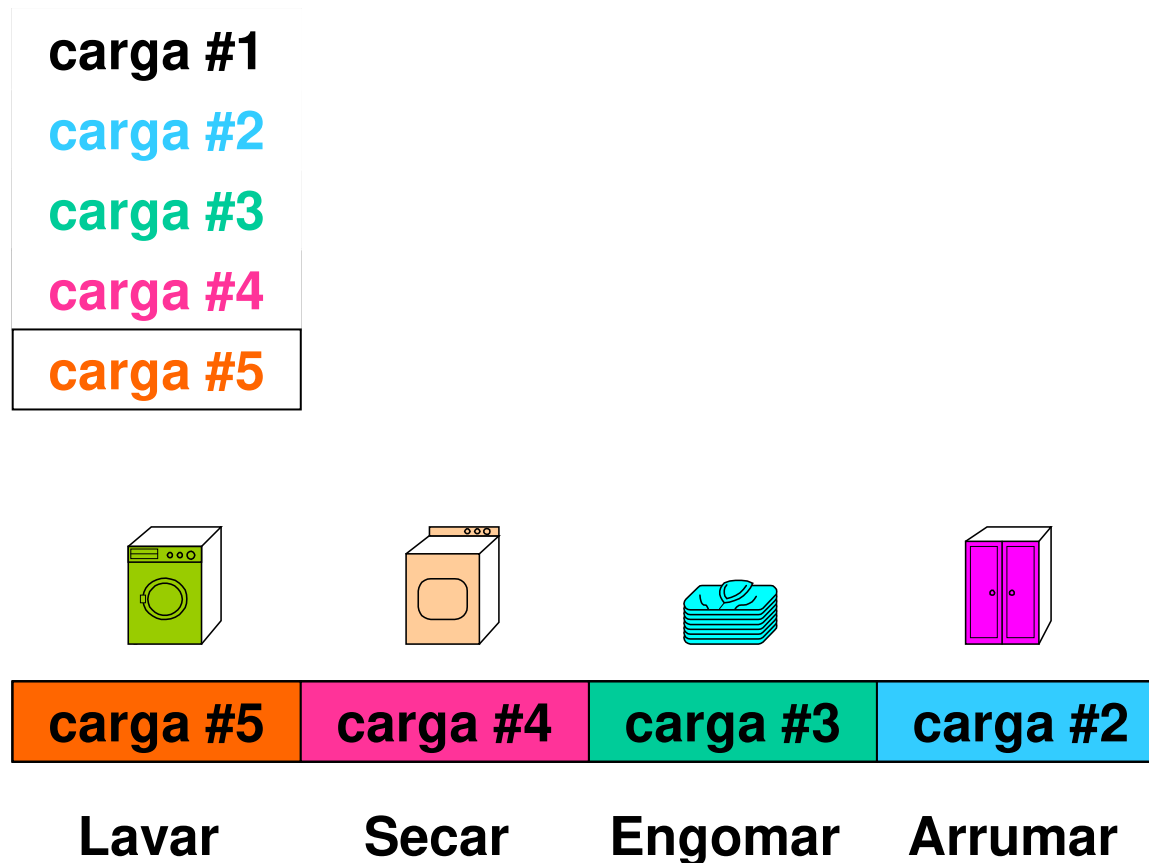
Se o tempo para tratar uma carga de roupa for uma hora, tratar quatro cargas demorará **quatro horas**.

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

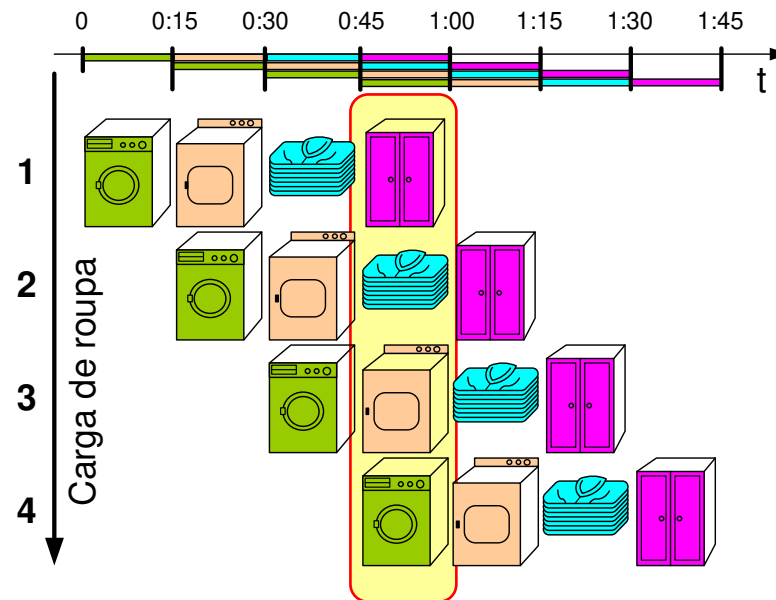
Pipelining - exemplo por analogia

- Na versão *pipelined*, o processamento das cargas de roupa seria (admitindo tempo nulo entre a comutação de tarefas):



Pipelining - exemplo por analogia

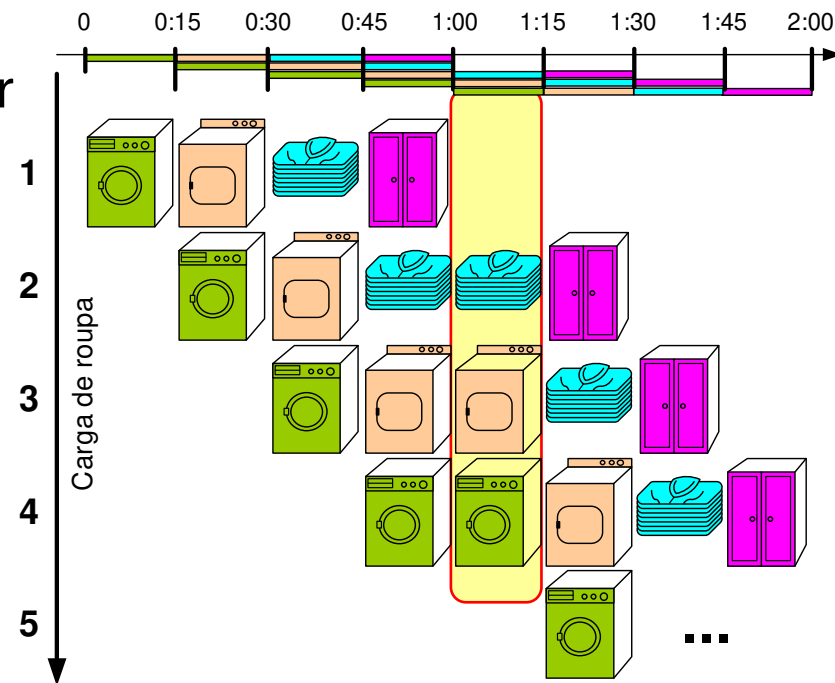
- O processo de tratamento da versão *pipelined* pode então ser descrito temporalmente do seguinte modo:



- Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45 (ou seja 135 minutos menos (240 – 105)).
- O que acontece se, por exemplo, a carga 2 não precisar de ser engomada?

Pipelining - exemplo por analogia

- O que acontece se a carga 2 tiver roupa que, por alguma razão, demora mais tempo a engomar?
- É necessária uma segunda "slot" de 15 min para completar a engomagem da carga 2
- A carga 3 não pode avançar para a engomagem e permanece na máquina de secar
- A carga 4 não pode avançar para a máquina de secar e permanece na máquina de lavar
- A carga 5 só é colocada na máquina de lavar no minuto 75

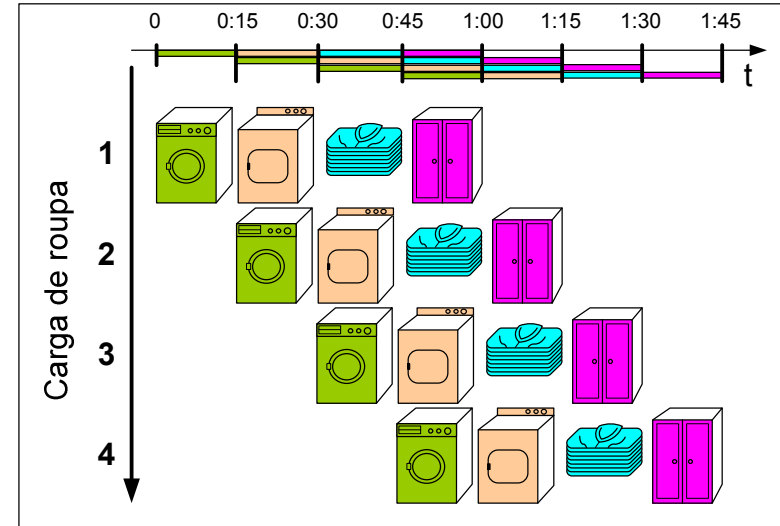


Pipelining - exemplo por analogia

- O paradoxo aparente da solução *pipelined* é que o tempo necessário para o processamento completo de uma carga de roupa não difere do tempo 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 de desempenho
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: $\text{Ganho} \approx \frac{N \times F}{N} = F$

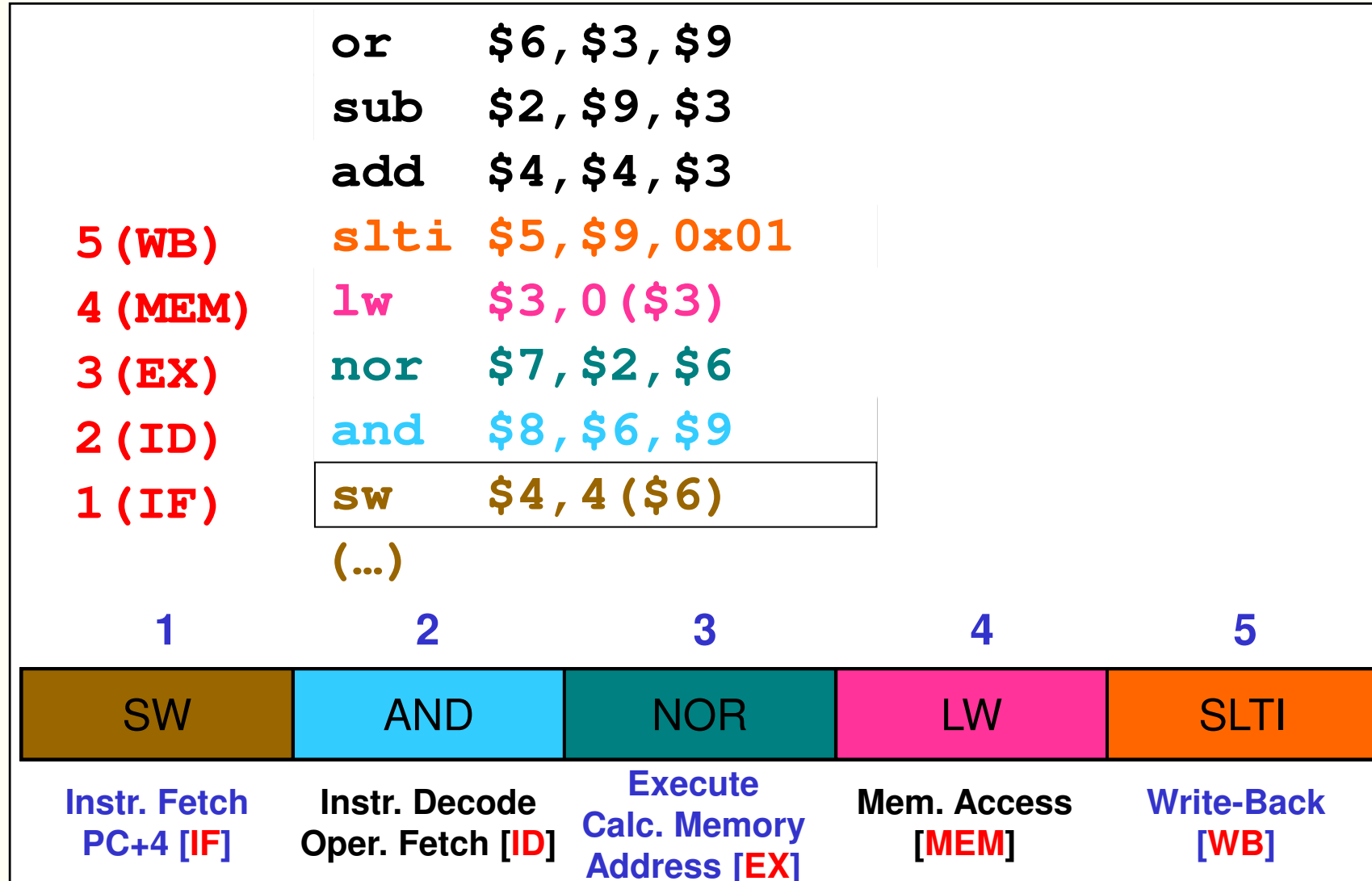
Pipelining – ganho de desempenho

- 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 noutro 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 ganho teórico estabelece que a solução *pipelined* é quatro vezes mais rápida do que a solução não *pipelined*
- A adoção de *pipelines* muito longos (com muitos estágios) pode, contudo, limitar drasticamente a eficiência global

Pipelining no MIPS

- Os mesmos princípios observados no exemplo do tratamento da roupa, podem igualmente ser aplicados aos processadores
- Para o MIPS, como já analisado, a execução da instrução mais longa (LW) pode ser dividida genericamente em **cinco fases**
- 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:
 1. **Instruction fetch [IF]** - ler a instrução da memória, incremento do PC
 2. **Operand fetch [ID]** - ler os registos e descodificar a instrução (os formatos das instruções do MIPS permitem que estas duas tarefas possam ser executadas em paralelo)
 3. **Execute [EX]** - executar a operação ou calcular um endereço
 4. **Memory access [MEM]** - aceder à memória de dados para leitura ou escrita
 5. **Write-Back [WB]** - escrever o resultado no registo destino

Pipelining no MIPS



Pipelining – Problemas (exemplo 1)

lw \$1, 0 (\$9)

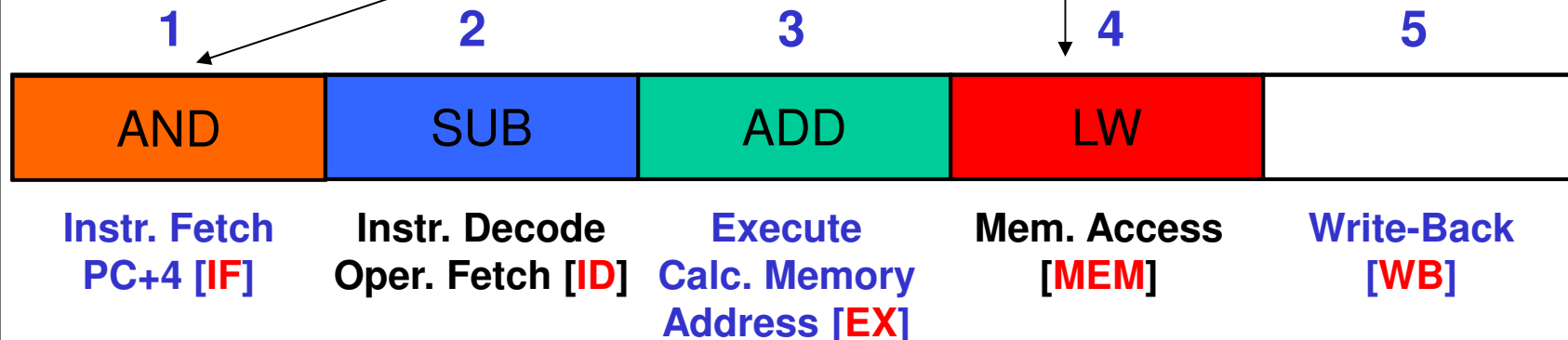
add \$2, \$3, \$4

sub \$3, \$4, \$5

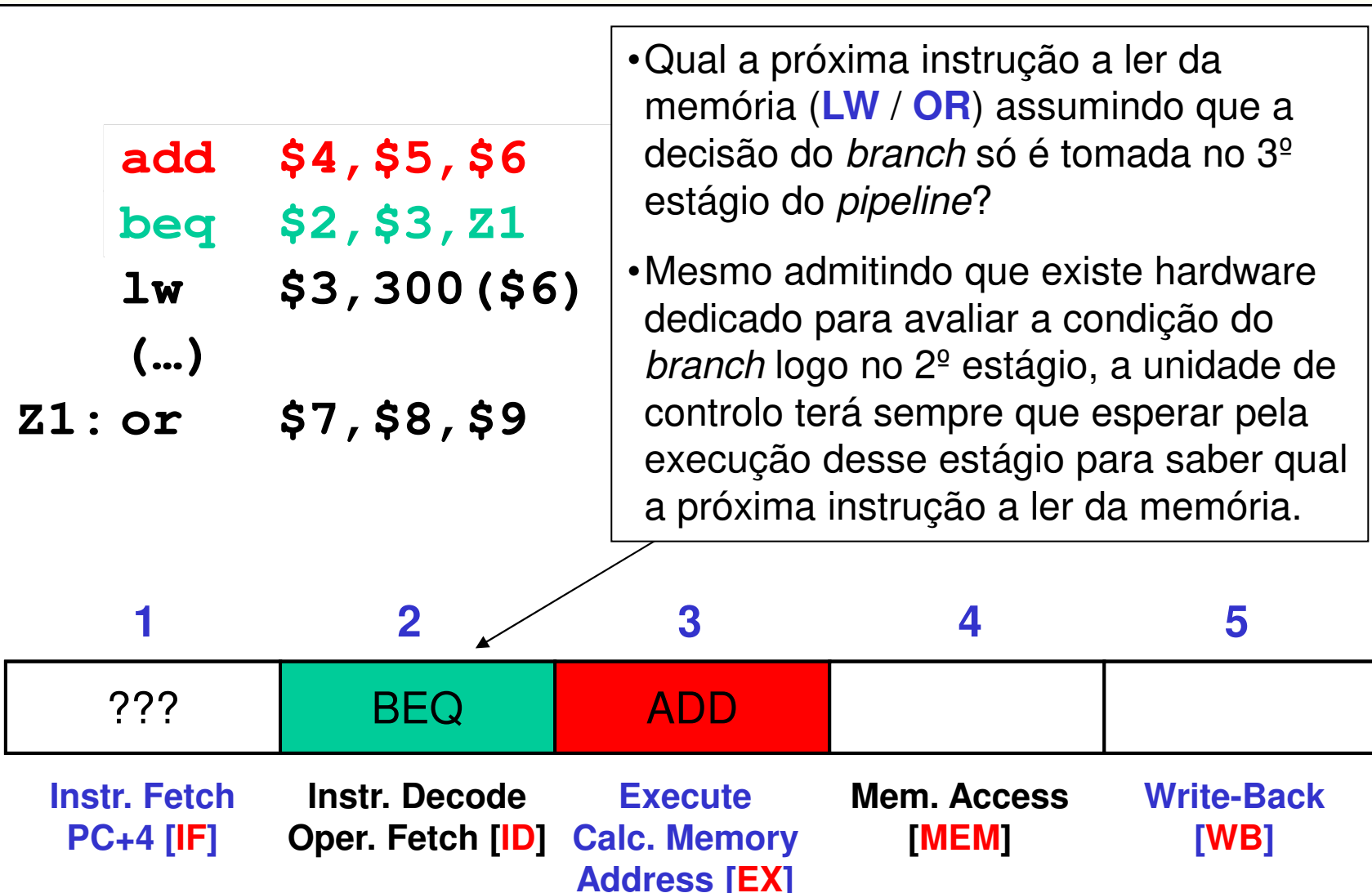
and \$4, \$5, \$6

lw \$5, 16 (\$9)

- 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 ao mesmo tempo



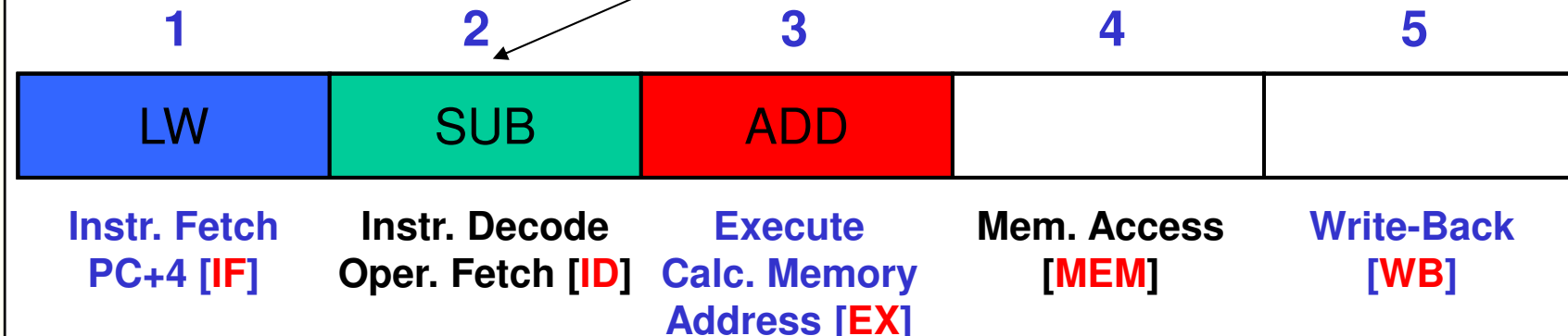
Pipelining – Problemas (exemplo 2)



Pipelining – Problemas (exemplo 3)

add **\$3**, \$4, \$5
sub \$2, **\$3**, \$6
lw \$4, 0 (\$5)

A instrução de subtração (SUB) não pode avançar para o estágio seguinte (EX) uma vez que o valor de um dos seus operandos ainda não foi calculado e armazenado no registo destino, o **\$3**, pela instrução anterior.



Datapath pipelined para o MIPS

- Nos slides seguintes vamos construir e analisar um *datapath pipeline* que suporte a execução das instruções do MIPS que já considerámos anteriormente, isto é:
 - Acesso à memória: **lw** (*load word*) e **sw** (*store word*)
 - Tipo R: **add**, **sub**, **and**, **or** e **slt**
 - Imediatas: **addi** e **slti**
 - Alteração do fluxo de execução: **beq** e **j**
- Na comparação dos tempos de execução destas instruções num *DP single cycle* e num *DP pipelined*, tomamos como referência os seguintes tempos de execução de cada uma das fases:

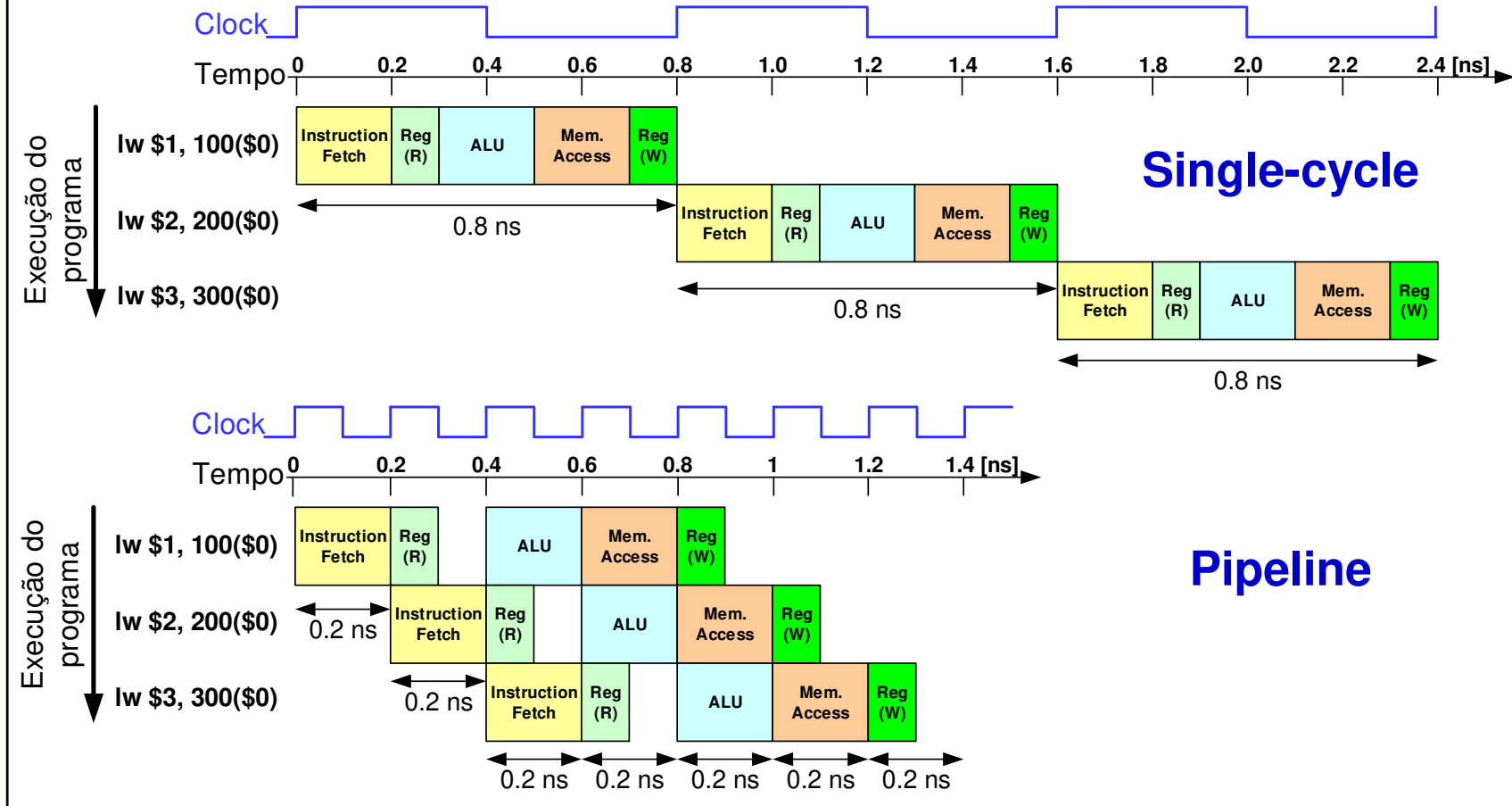
Instruction	Instruction Fetch	Register Read	ALU Operation	Memory Access	Register Write	Tempo total
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-Type (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps
Immediate (addi, slti)	200 ps	100 ps	200 ps		100 ps	600 ps

Datapath pipelined para o MIPS

- Num *datapath single cycle* o período do sinal de relógio terá que ser ajustado de modo a permitir a execução da instrução mais lenta (lw)
- Na solução *single cycle* o período de relógio deve então ser, no mínimo, 800 ps, ou seja, todas as instruções, independentemente do tempo mínimo que poderiam durar, serão executadas num tempo de 800 ps
- Num *datapath pipelined*, embora alguns estágios pudessem executar em menos tempo, o período do relógio tem que ser ajustado para o atraso de propagação do elemento operativo mais lento, 200 ps no exemplo

Datapath pipelined para o MIPS

- Exemplo de execução de 3 instruções LW nos *datapaths* *single-cycle* e *pipelined*



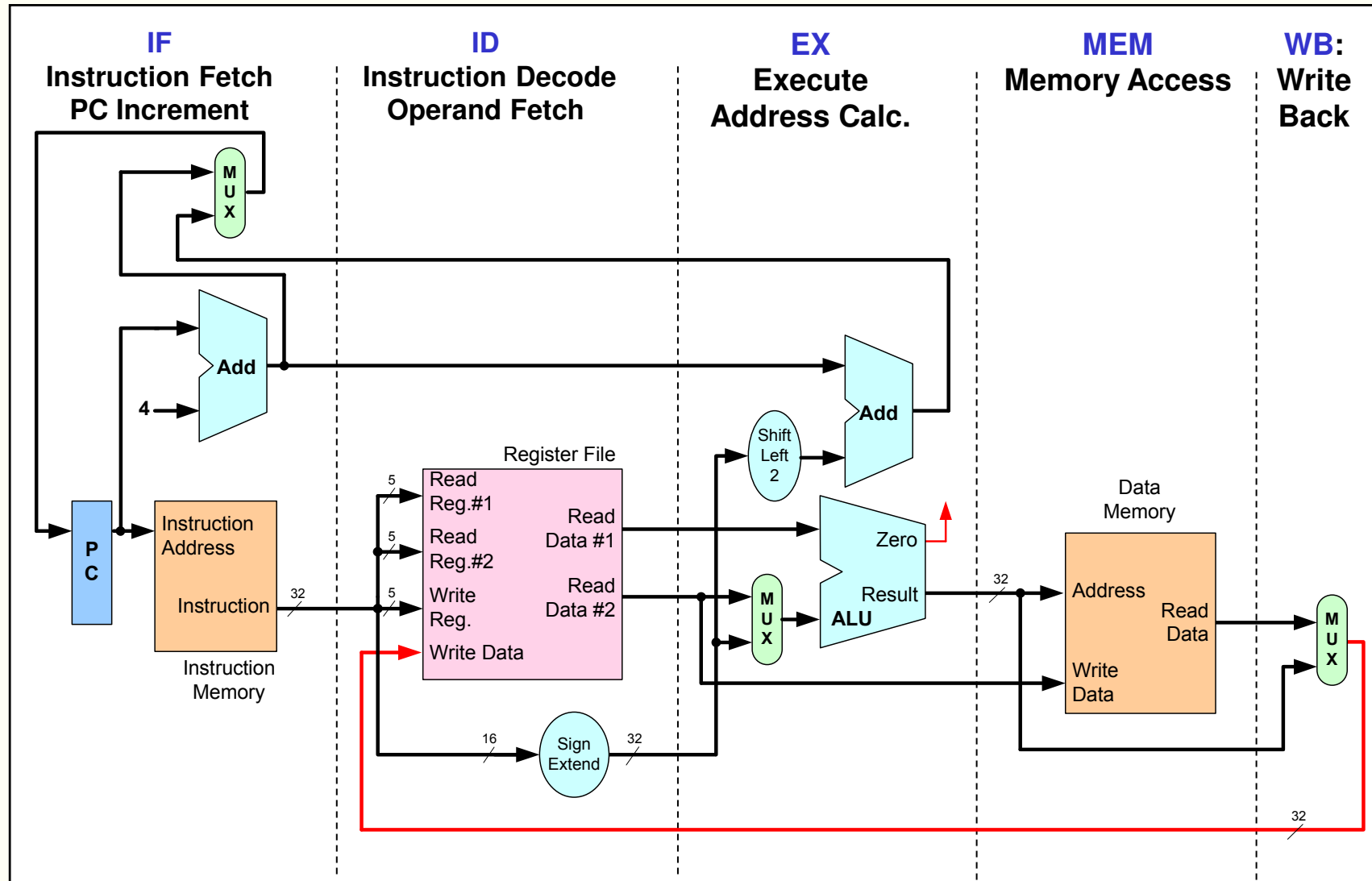
Datapath pipelined para o MIPS

- 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 (a unidade de controlo não tem que ter em consideração a dimensão da instrução decodificada)
 - **Poucos formatos de instrução**, com a referência aos registos a ler sempre nos mesmos campos (isso permite que os registos sejam lidos no segundo estágio ao mesmo tempo que a instrução é decodificada pela unidade de controlo)
 - **Referências à memória só aparecem em instruções de load/store**: o terceiro estágio pode ser usado para calcular o resultado da operação na ALU ou para calcular o endereço de memória, permitindo o acesso à memória no estágio seguinte
 - Os **operandos em memória têm que estar alinhados**: qualquer operação de leitura/escrita da memória pode ser feita num único estágio

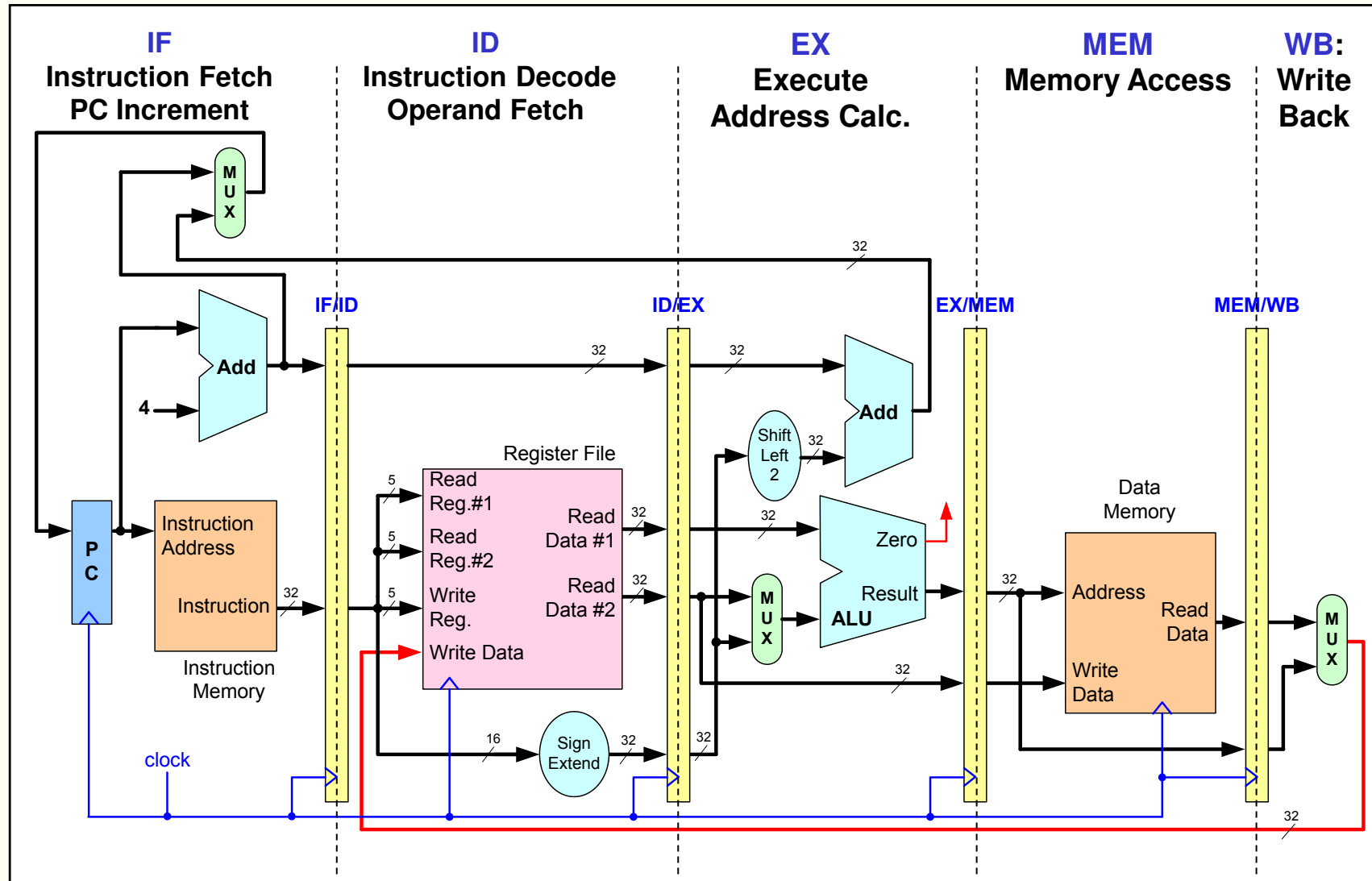
Datapath pipelined para o MIPS

- O *pipeline* 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)
- A solução *pipelined* para o MIPS parte do modelo do *datapath single-cycle*
- Na solução apresentada no slide seguinte não são identificados os sinais de controlo nem a respetiva unidade de controlo

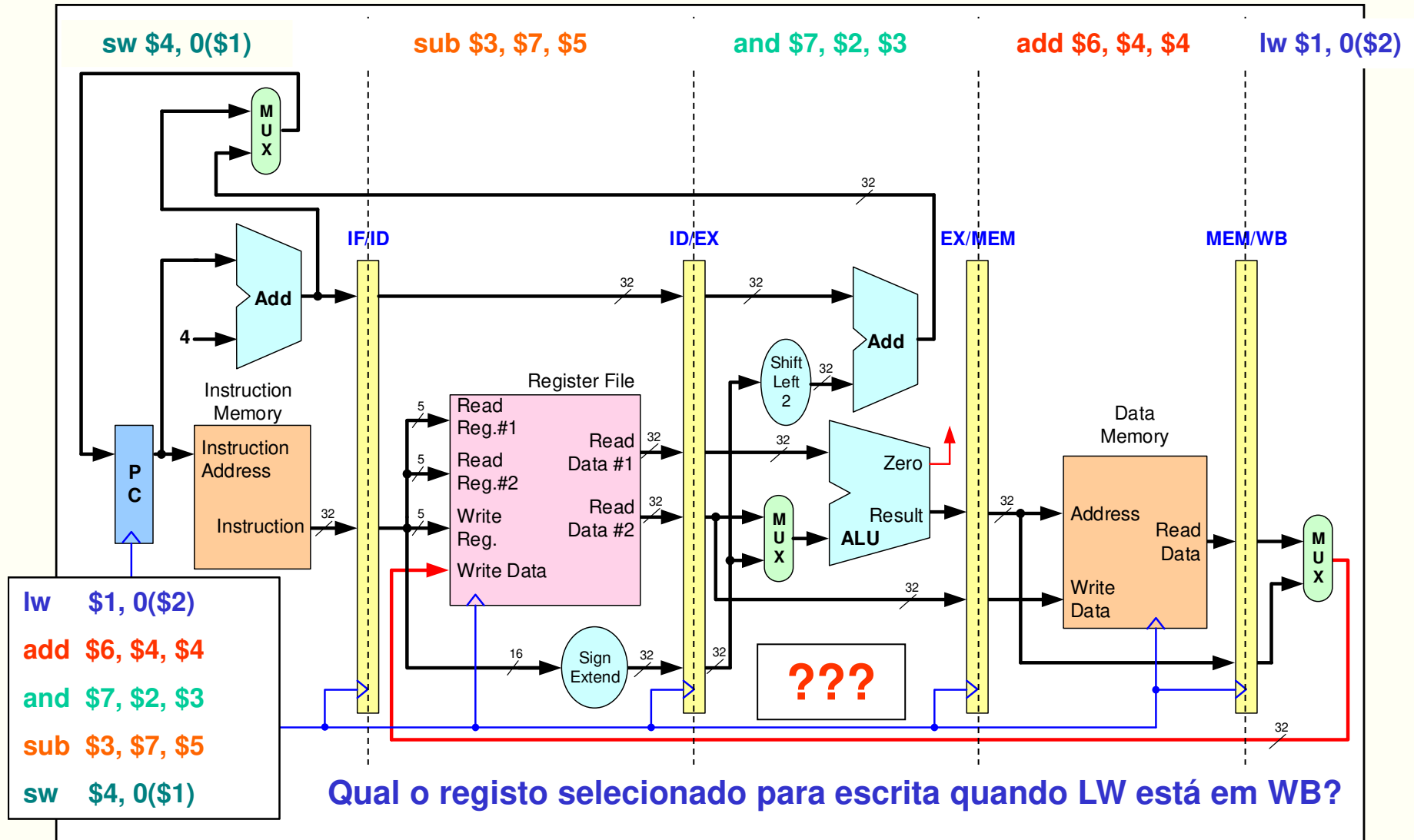
Divisão em fases de execução



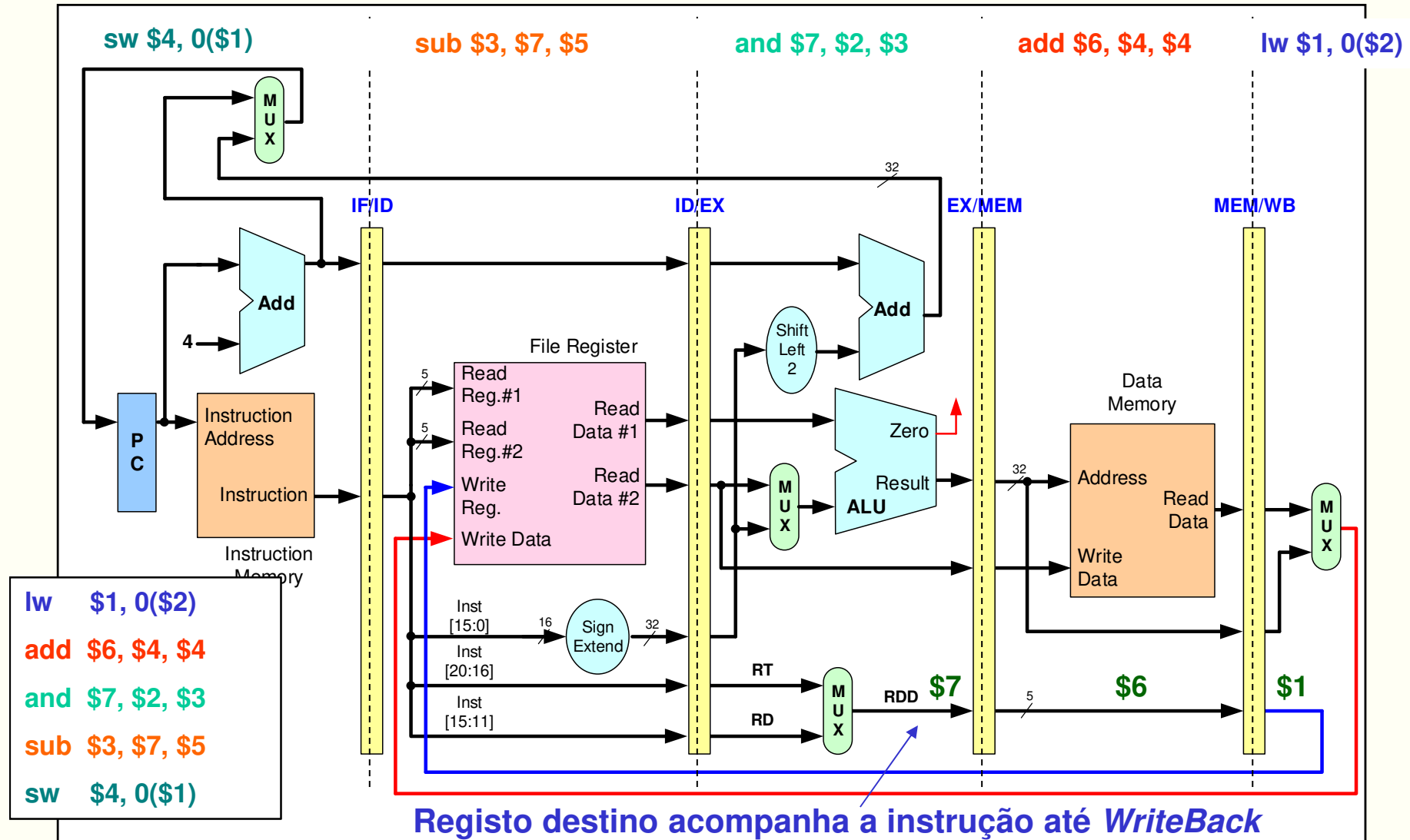
Divisão em fases de execução – registos de *pipeline*



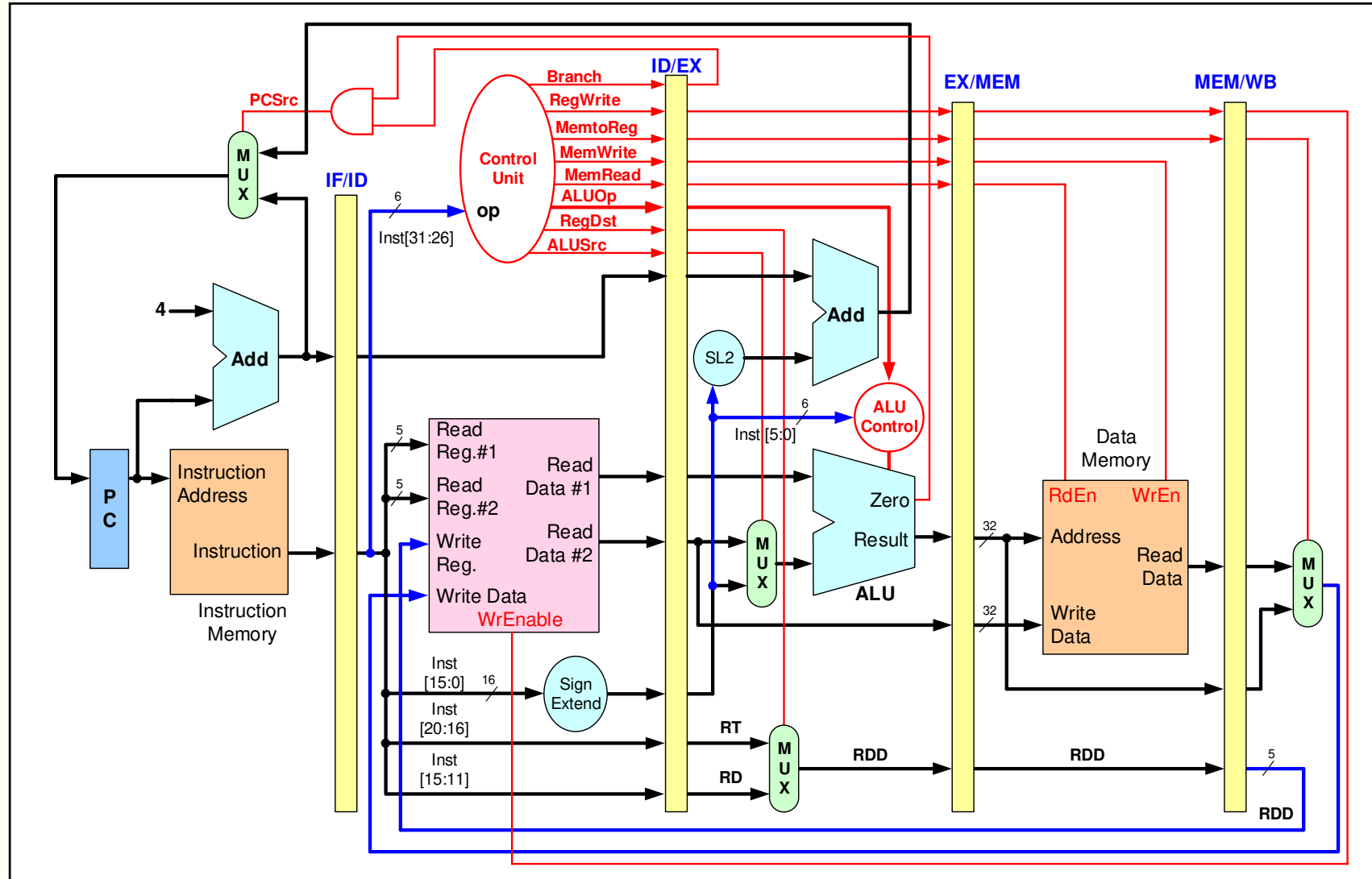
Execução de instruções



Datapath pipelined – 1ª versão



Datapath pipelined com 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
 - Os sinais **MemRead** e **MemWrite** são propagados até à fase MEM, onde controlam o acesso à memória
 - 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)

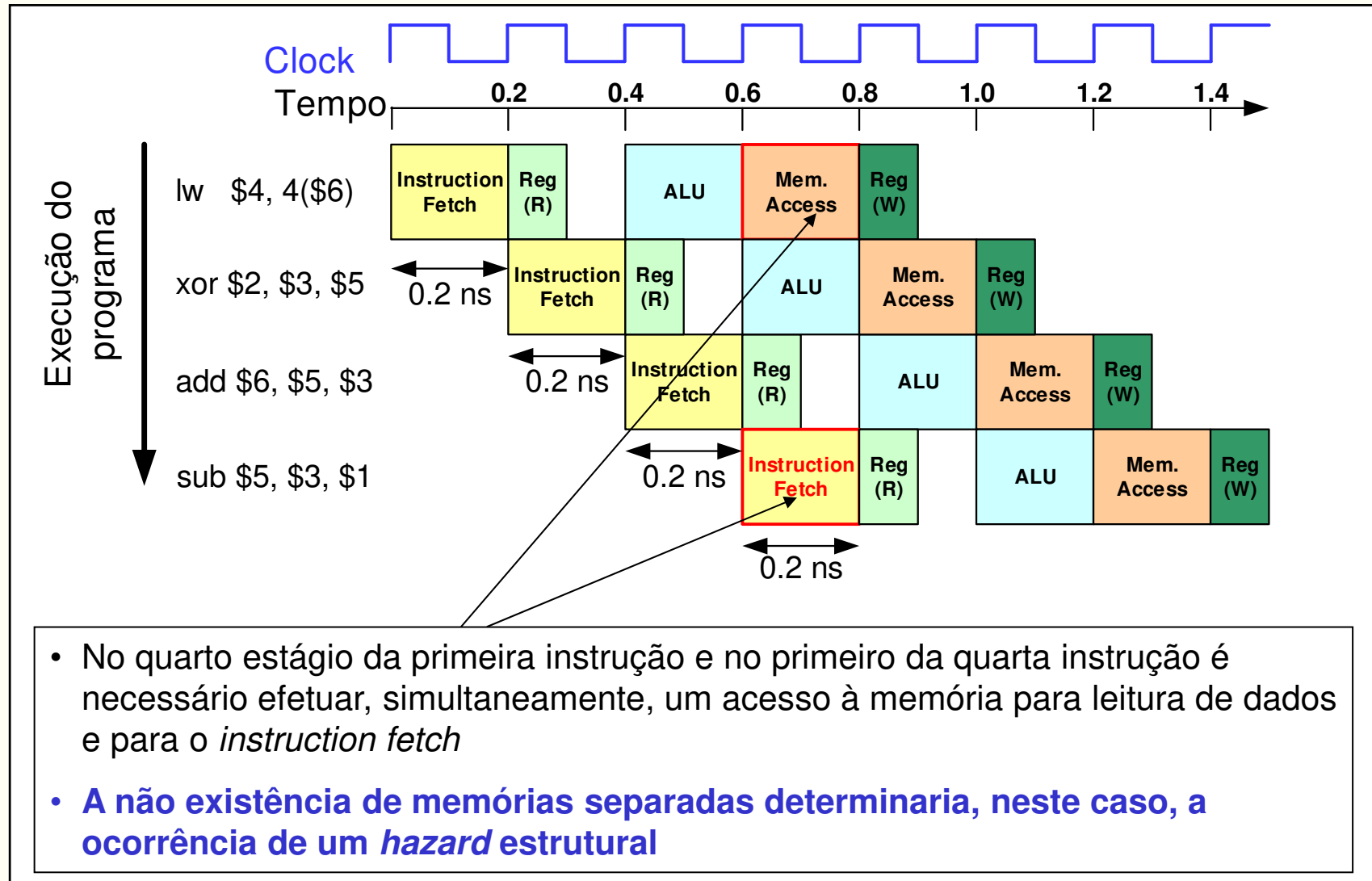
Pipeline Hazards

- 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 aspetos ao nível da implementação da arquitetura *pipelined* do MIPS

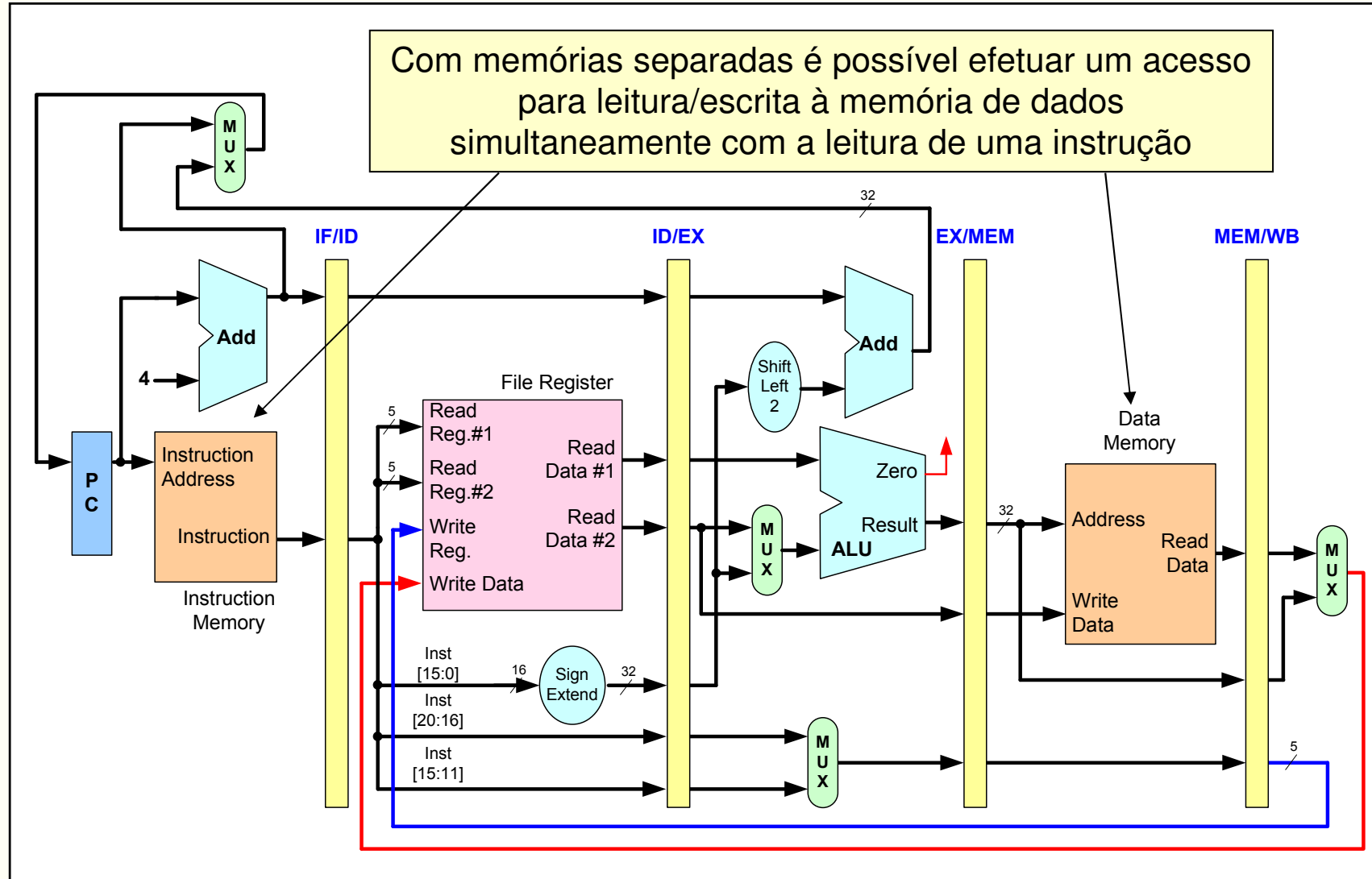
Hazards Estruturais

- 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 instruções 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, usando outro elemento operativo diferente da ALU

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

- Exemplo:

beq \$5, \$6, **next**

add \$2, \$3, \$4

...

next: **lw** \$3, 0(\$4)

...

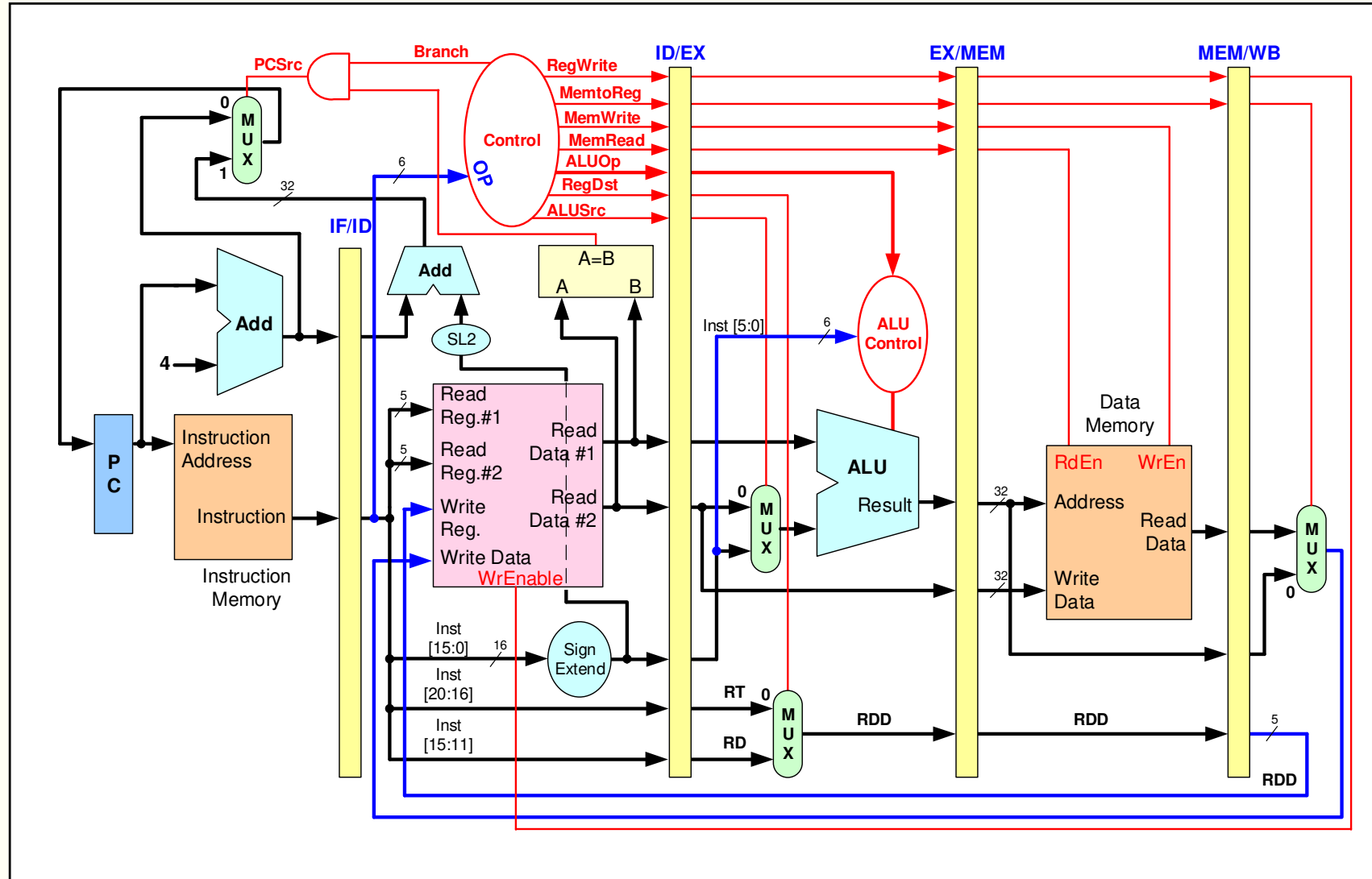
Qual a instrução que deve entrar no *pipeline* a seguir à instrução "beq"?

- No caso do MIPS, as situações de *hazard* de controlo surgem com as instruções de salto, (*jumps* e *branches*: j, jal, jalr, jr, beq, ...)

Hazards de Controlo

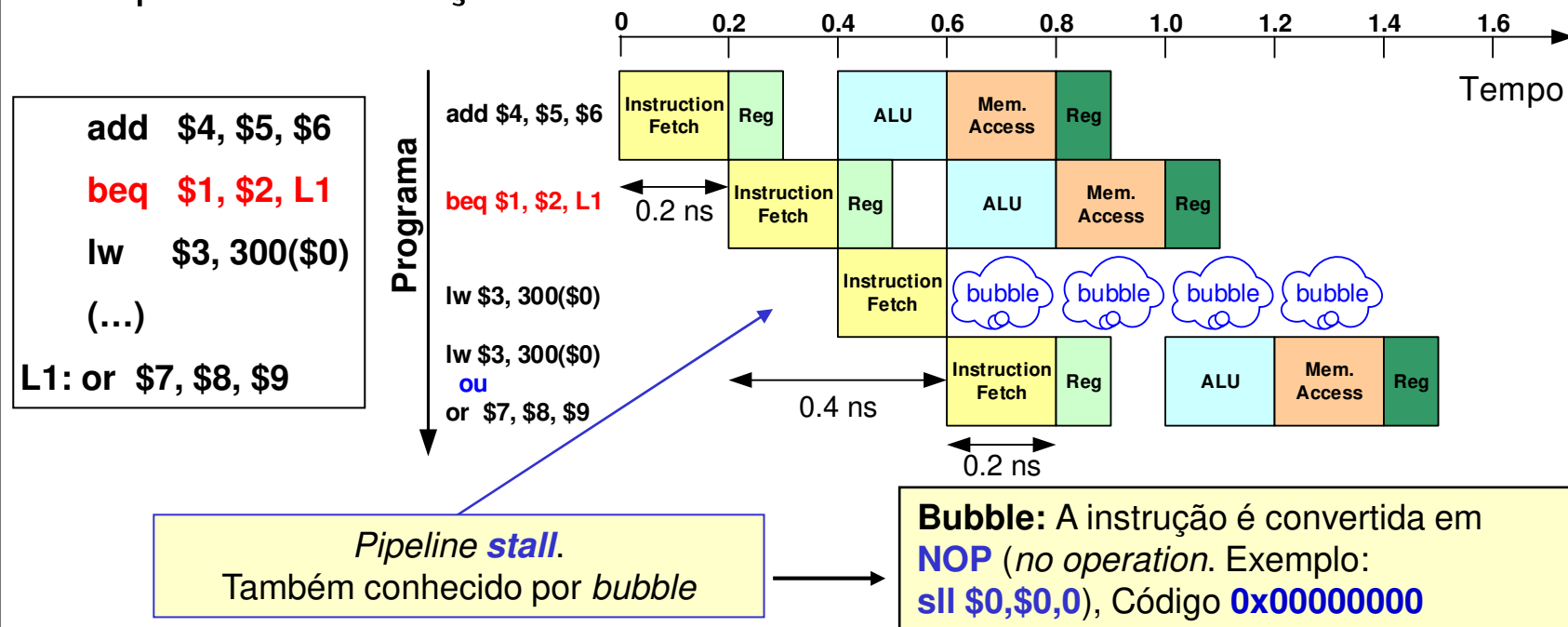
- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de instruções
- A resolução dos *branches* em ID minimiza o problema, e por isso a **comparação dos operandos passa a ser 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



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

Exercício

- Se 15% das instruções de um dado programa forem *branches* e *jumps*, qual o efeito da solução de *stalling* no desempenho da arquitetura, admitindo que essas instruções são resolvidas em ID?

Sem *stalls*: $CPI = 1$

Com *stalls*: $CPI = 0,85 * 1 + 0,15 * 2 = 1,15$

Relação de desempenho = $1 / 1,15 = 0,87$

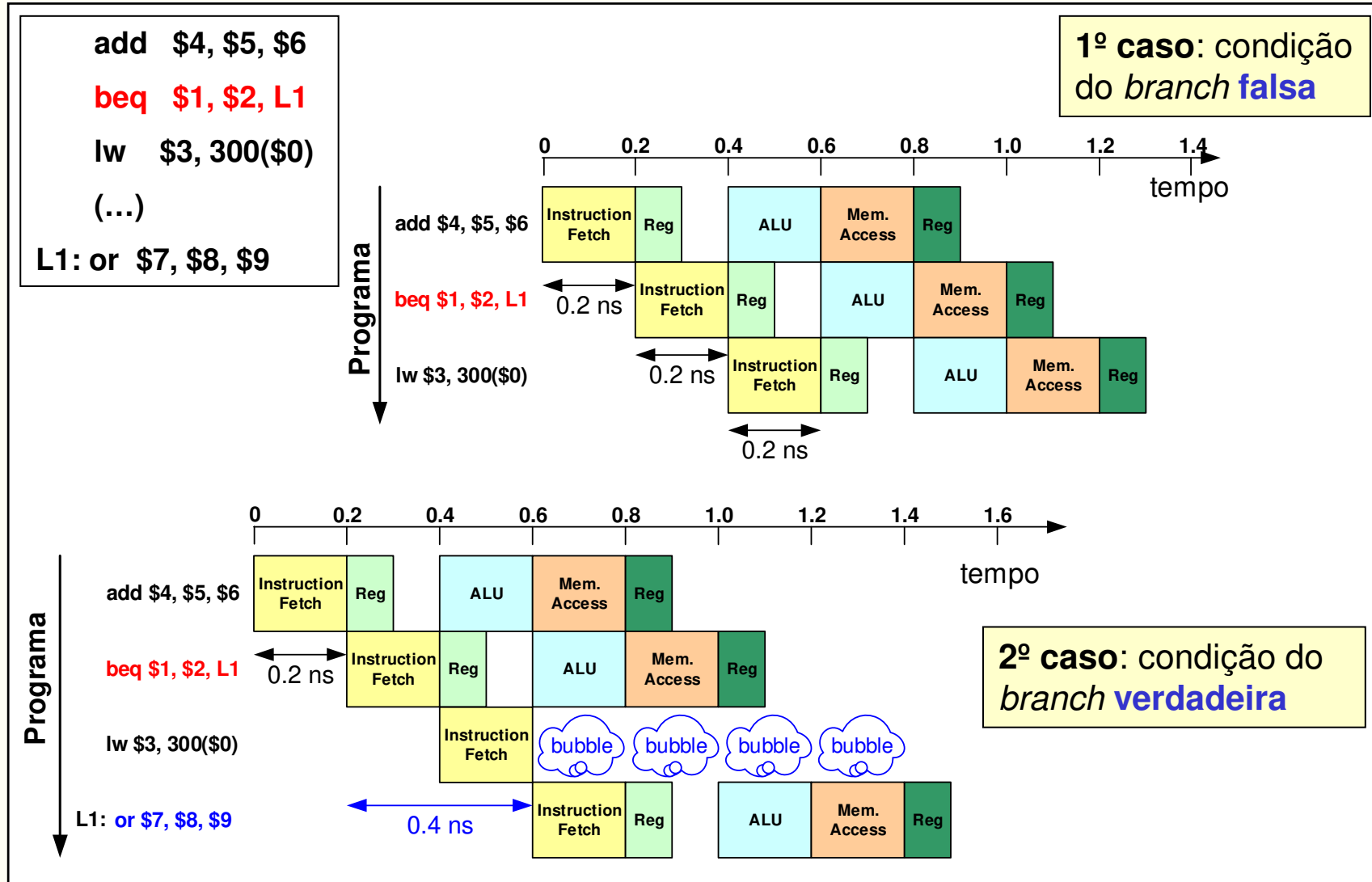
- A degradação do desempenho é tanto maior quanto mais tarde for resolvida a instrução que altera o fluxo de execução.
- Na mesma situação, se o *branch* / *jump* for resolvido em EX, a relação passa a ser:

Relação de desempenho = $1 / (0,85 * 1 + 0,15 * 3) = 0,77$

Hazards de controlo

- Uma solução alternativa ao *pipeline stalling* é designada por **previsão** (*prediction*):
 - Prevê-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 anulada (convertida em **nop**), continuando o *instruction fetch* na instrução correta
- Se a previsão estiver certa, esta estratégia permite poupar tempo
- Para o exemplo do slide anterior, se a previsão for correta 50% das vezes, a relação de desempenho passa a ser:
$$\text{CPI} = 1 + 1 * 0,15 / 2 = 1,075$$
$$\text{Relação de desempenho} = 1 / 1,075 = 0,93$$

Hazards de controlo – previsão estática *not taken*



Hazards de controlo – previsão

- Os previsores usados nas arquiteturas atuais são mais elaborados
- **Previsores estáticos**: o resultado da previsão não depende do histórico da execução das instruções de *branch / jump*:
 - **Previsor *Not taken***
 - **Previsor *Taken***
 - **Previsor *Backward taken, Forward not taken*** (BTFNT)
- **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 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 solução, o processador **executa sempre a instrução que se segue ao *branch*** (ou *jump*), independentemente de a condição ser verdadeira ou falsa
- Esta técnica é implementada com a ajuda do **compilador/assembler** que:
 - reorganiza 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^{as}
instruções

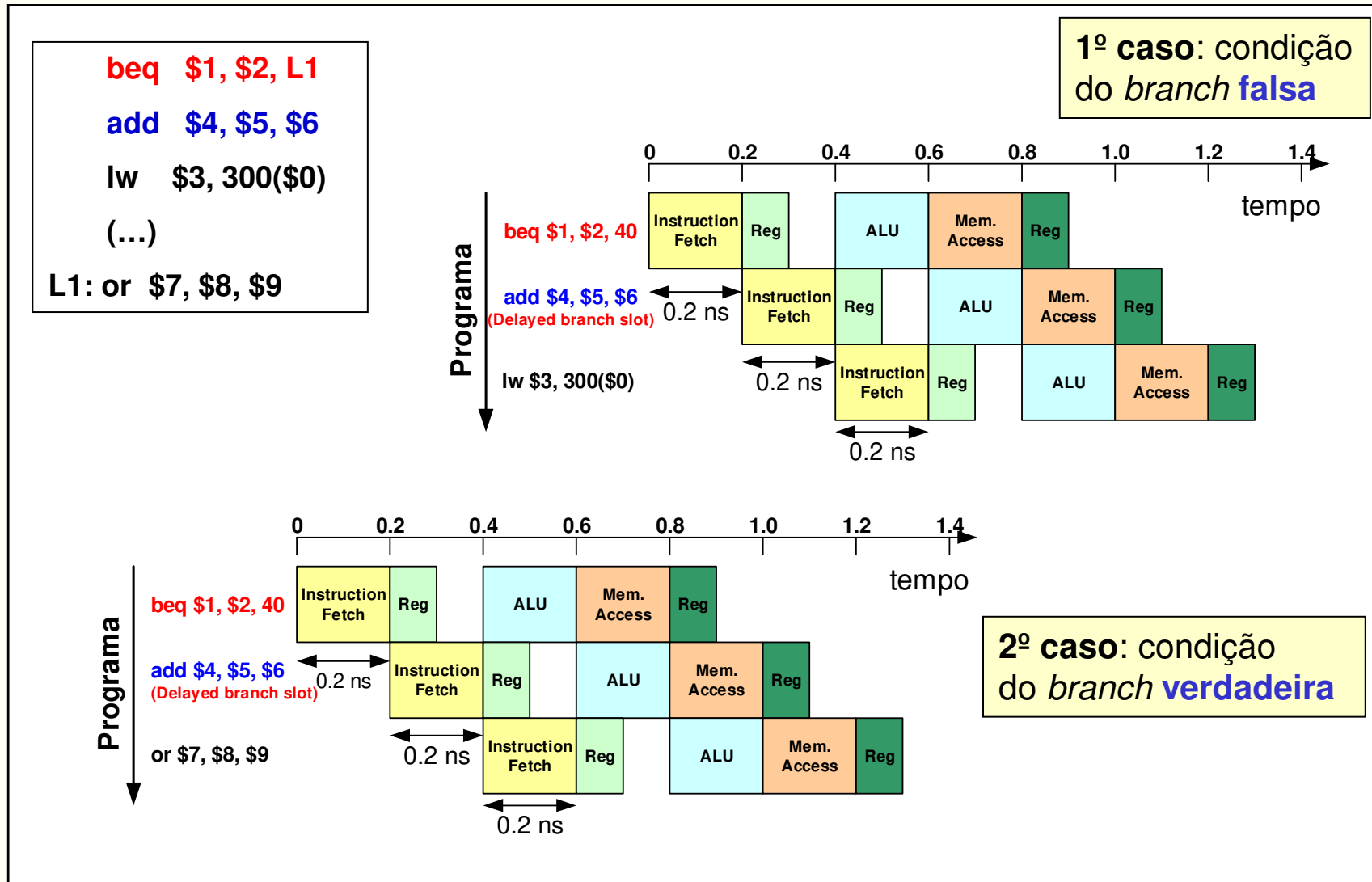


Código reordenado

```
beq  $1, $2, L1
add  $4, $5, $6
lw   $3, 300($0)
(...)
L1: or  $7, $8, $9
```

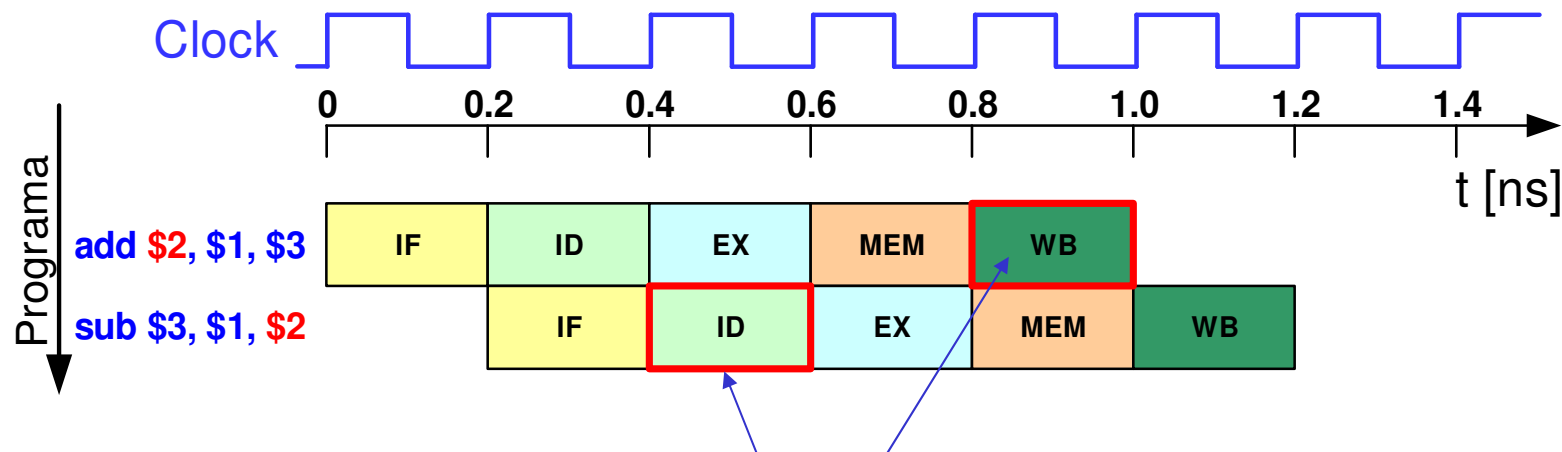
- Neste exemplo a instrução "**beq**" não depende do resultado produzido pela instrução "**add**", logo a troca das duas não altera o resultado final do programa
- No código reordenado (que é o executado no processador) a instrução "**add**" é sempre executada, independentemente do resultado *taken/not taken* do "**beq**"

Hazards de controlo – *delayed branch*



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 \$2, \$1, \$3**
sub \$3, \$1, \$2



A instrução "**sub \$3,\$1,\$2**" não pode avançar no *pipeline* antes de o valor de **\$2** ser calculado e armazenado pela instrução anterior (o valor é necessário em **t = 0.4**, mas só vai ser escrito no registo destino em **t = 1**)

Hazards de dados

- Se o resultado 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**)
- No exemplo anterior, a instrução SUB só poderia prosseguir para a fase EX em $t=1.2$
- O problema pode ser minorado, se a **escrita no banco de registos for feita a meio do ciclo de relógio** (i.e., na transição descendente)
 - a instrução que está na fase ID e que necessita do valor, poderá prosseguir na transição de relógio seguinte, já com o valor do registo atualizado
 - poupa-se 1 ciclo de relógio (no exemplo anterior, a instrução "**sub \$3,\$1,\$2**" poderá prosseguir para a fase EX em $t=1.0$)

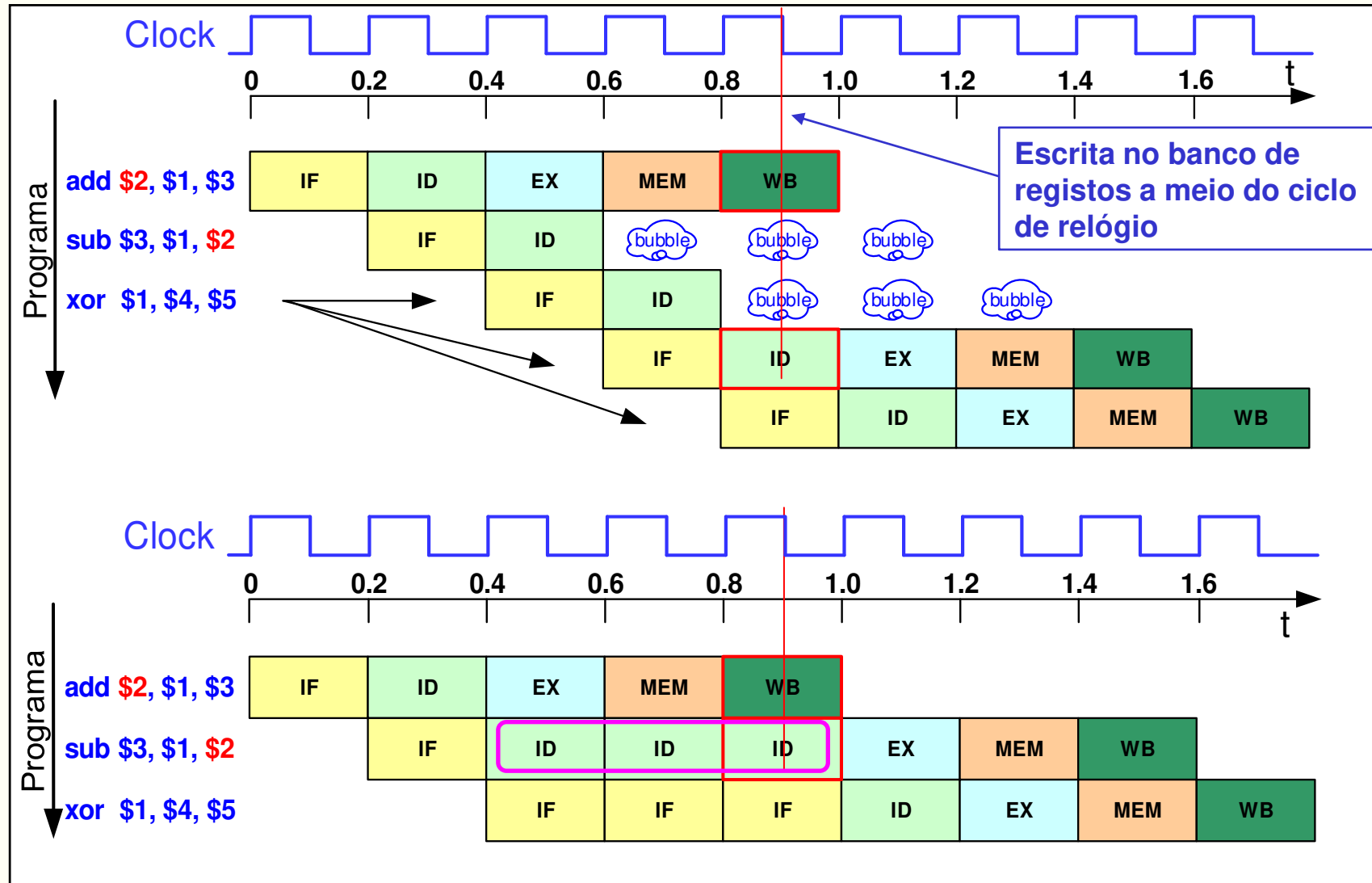
Hazards de dados – pipeline stalling

- Para o exemplo anterior, parar a progressão da instrução SUB no estágio ID durante 2 ciclos de relógio é equivalente a introduzir dois NOP entre as duas instruções

```
add    $2, $1, $3    # WB
nop                    # MEM
nop                    # EX
sub     $3, $1, $2    # ID
```

- Com a escrita no banco de registos feita a meio do ciclo de relógio, a instrução SUB lê o valor atualizado de \$2, quando a instrução ADD está na fase WB
- Primeira solução – ***stall do pipeline***:
 - ***parar a progressão no pipeline (stall)*** da instrução que necessita do valor (e das anteriores), no estágio **ID**, até que a instrução que produz o resultado chegue ao estágio **WB**

Hazards de dados – pipeline stalling



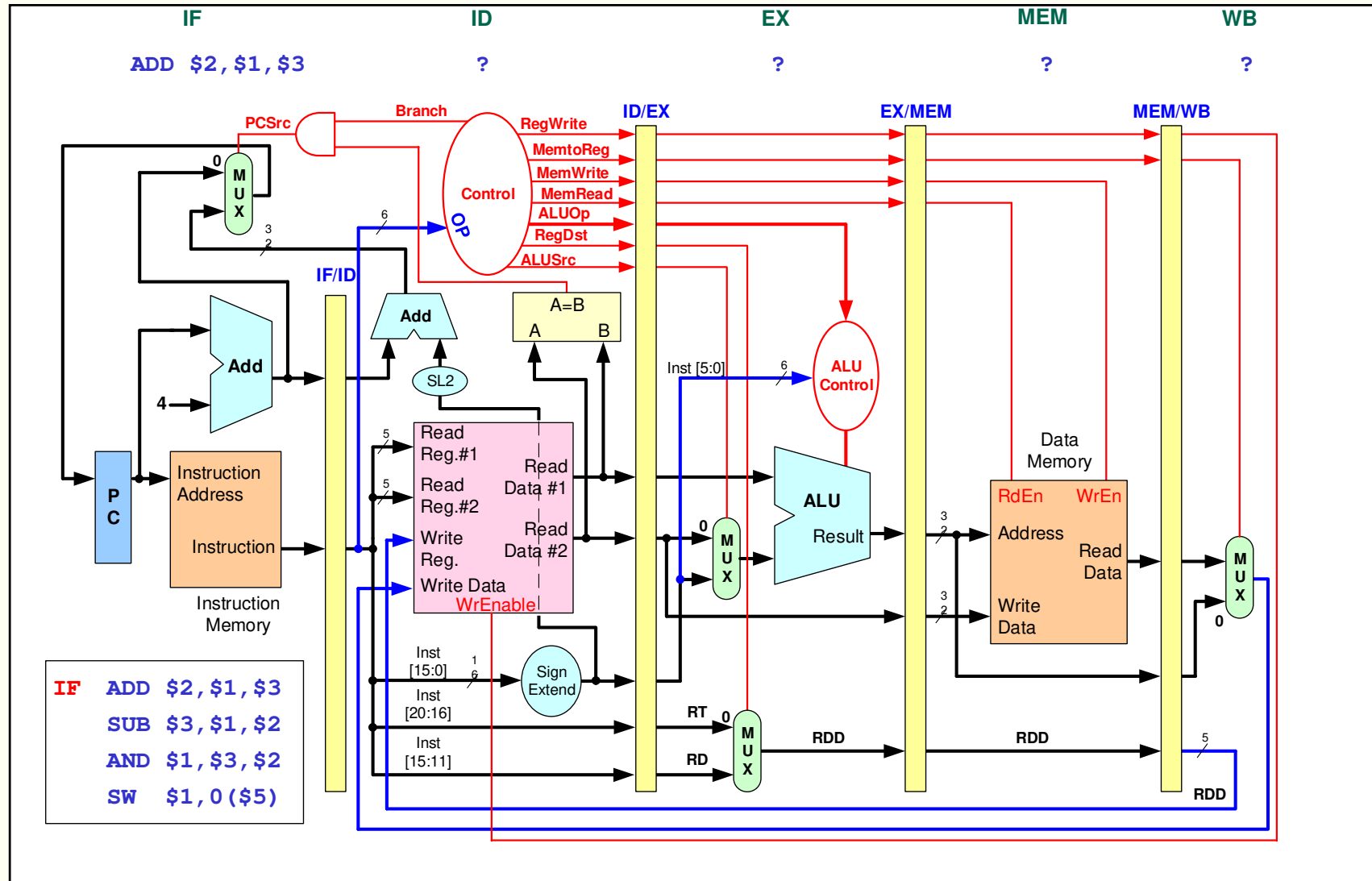
Hazards de dados resolvidos com *stalling* (1)

- Exemplo:

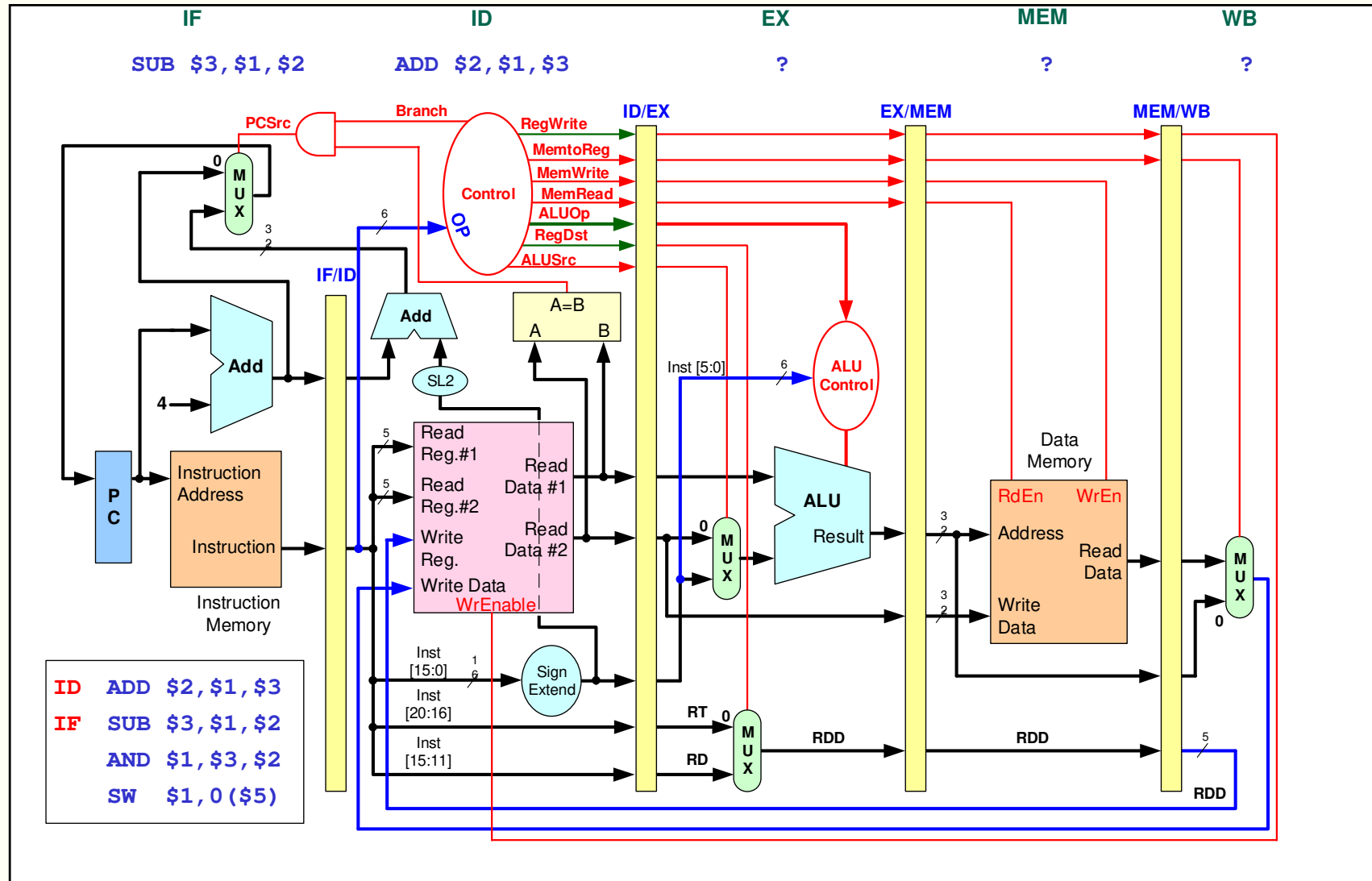
```
ADD  $2, $1, $3  #  
SUB  $3, $1, $2  #  
AND  $1, $3, $2  #  
SW   $1, 0 ($5)  #
```

- A sequência de instruções apresenta 3 *hazards* de dados:
 - Na instrução SUB, resultante da dependência do registro \$2
 - Na instrução AND, resultante da dependência do registro \$3
 - Na instrução SW, resultante da dependência do registro \$1
- Cada uma destas situações de *hazard* de dados obriga a fazer o *stall* do *pipeline* durante 2 ciclos de relógio

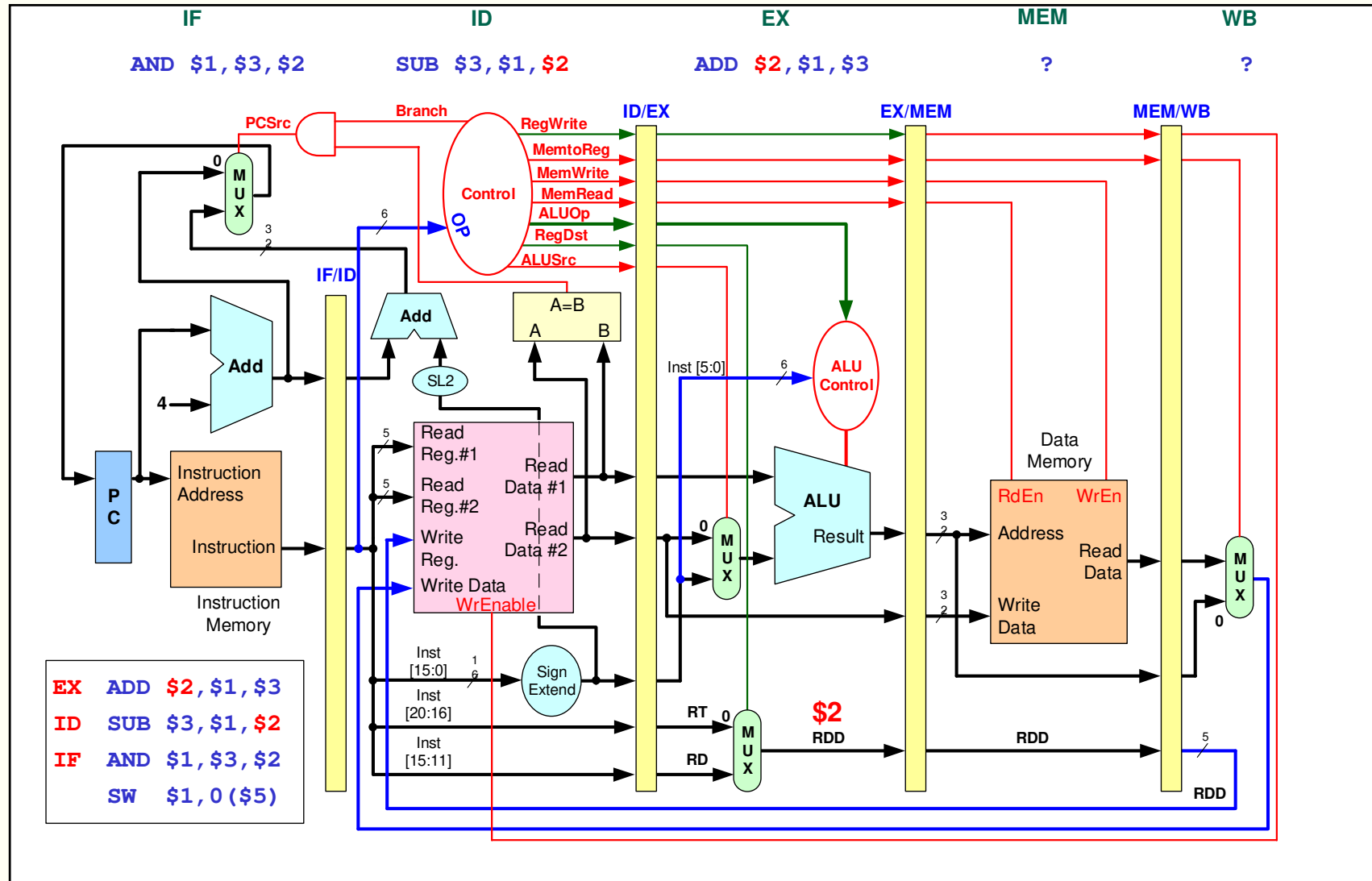
Hazards de dados resolvidos com *stalling* (2)



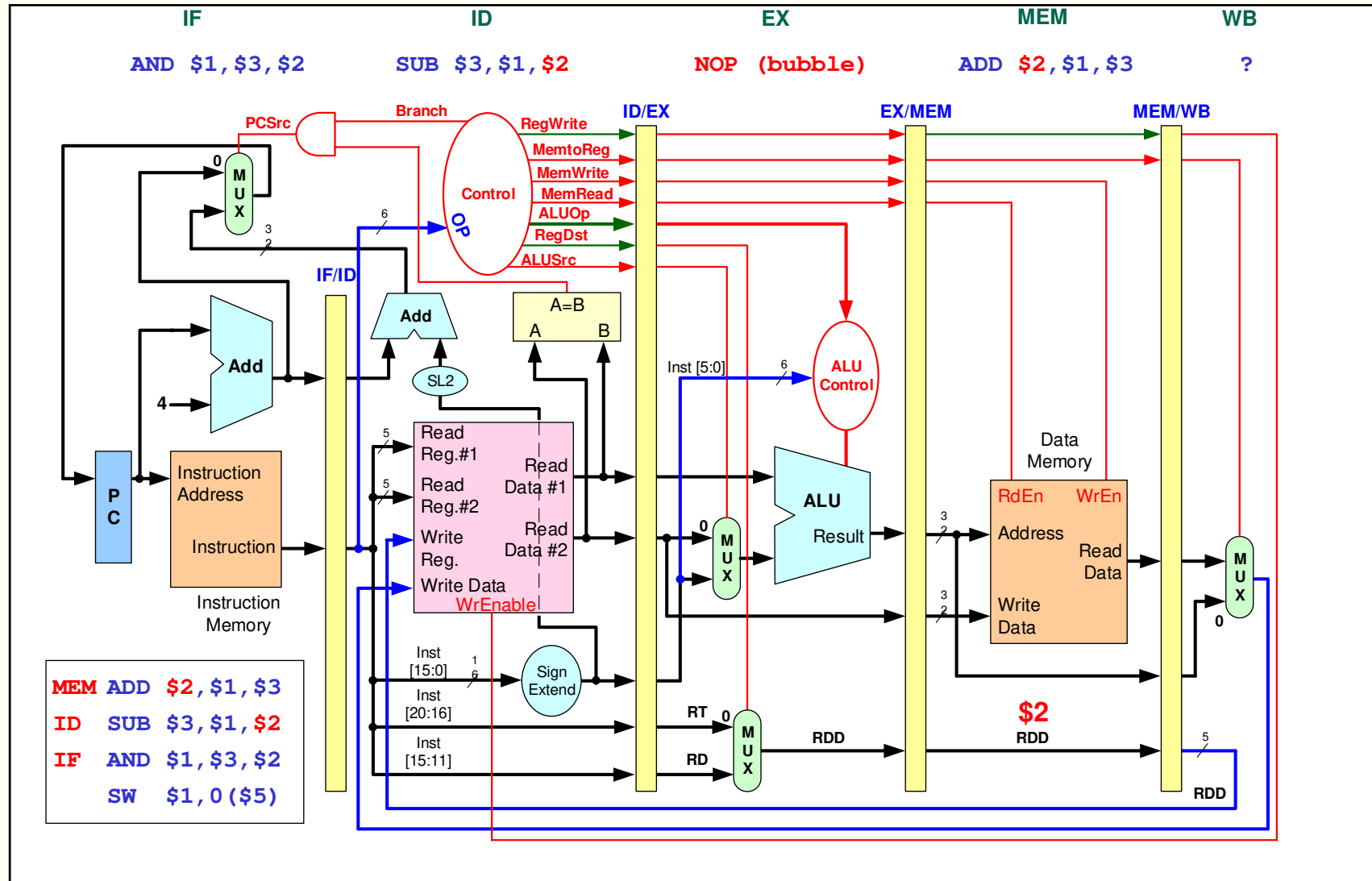
Hazards de dados resolvidos com *stalling* (3)



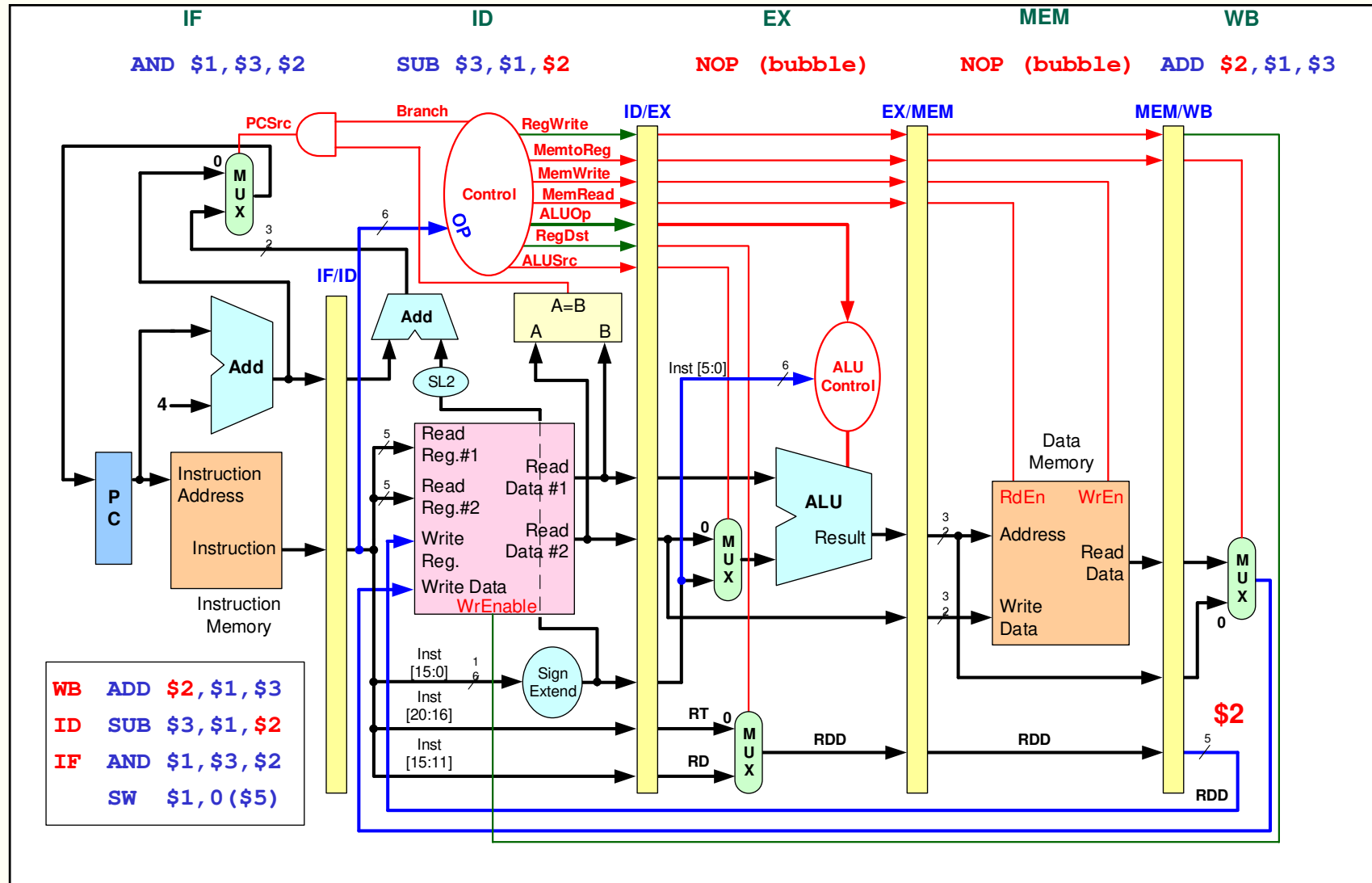
Hazards de dados resolvidos com *stalling* (4)



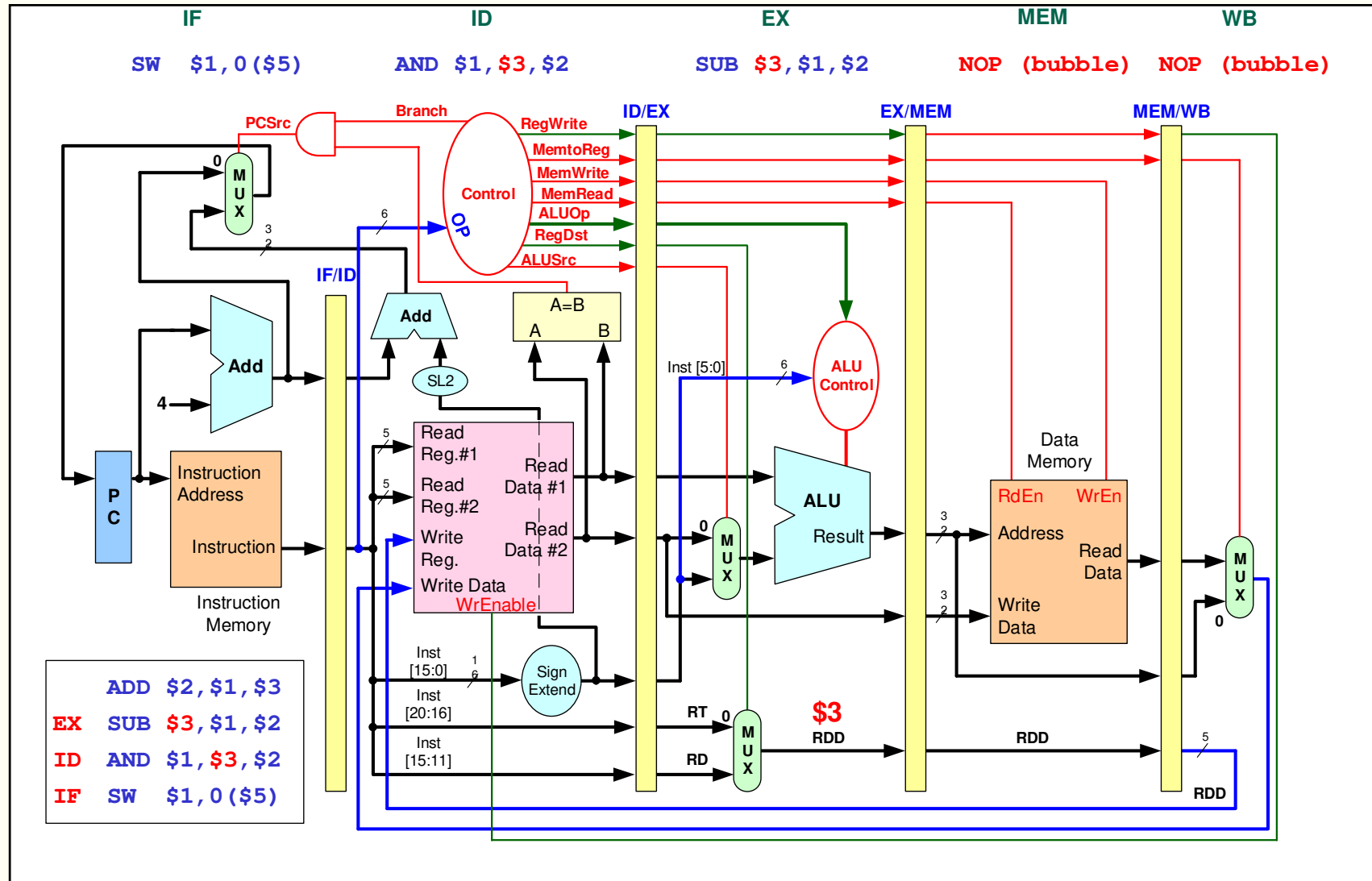
Hazards de dados resolvidos com *stalling* (5)



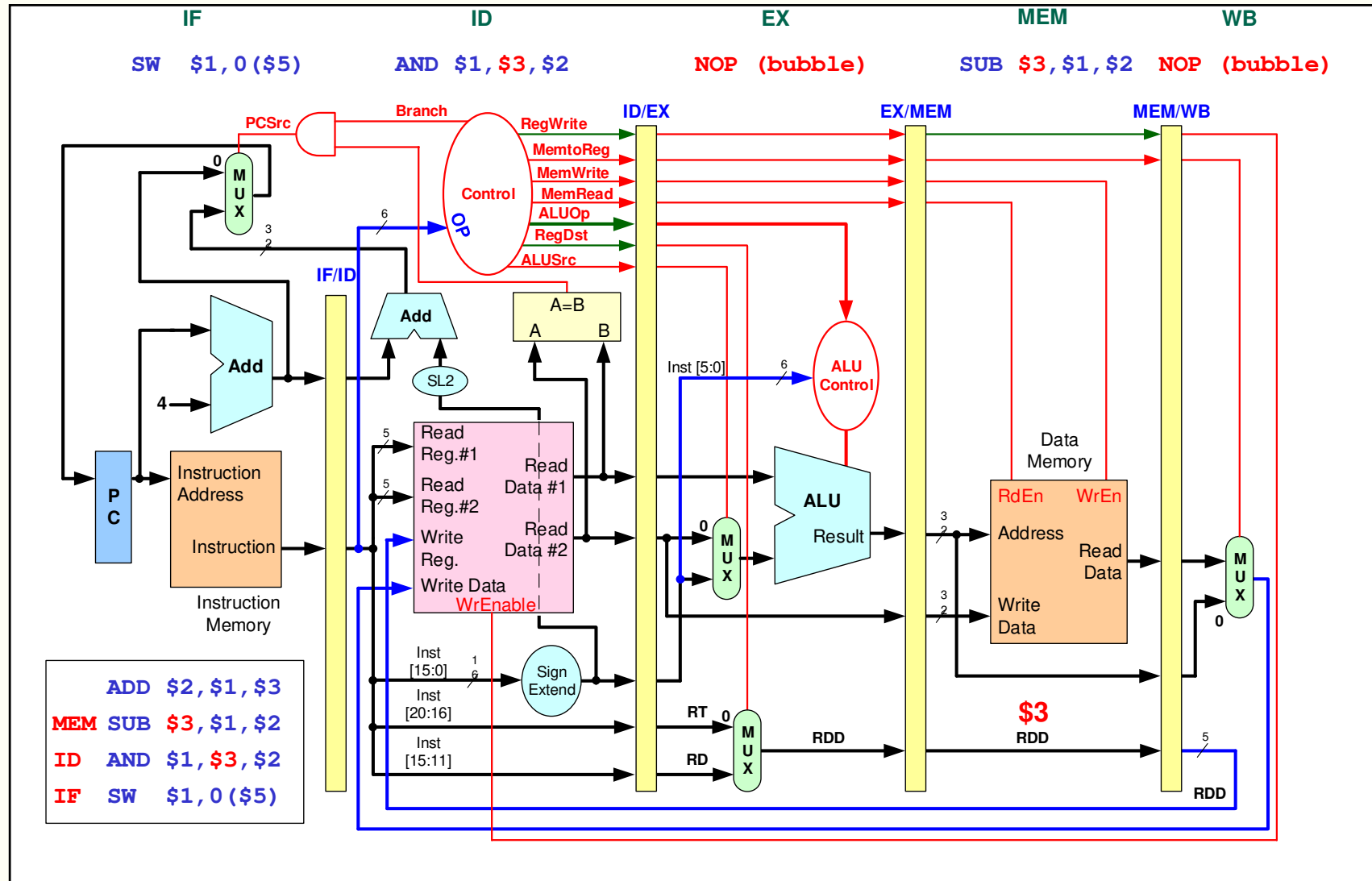
Hazards de dados resolvidos com *stalling* (6)



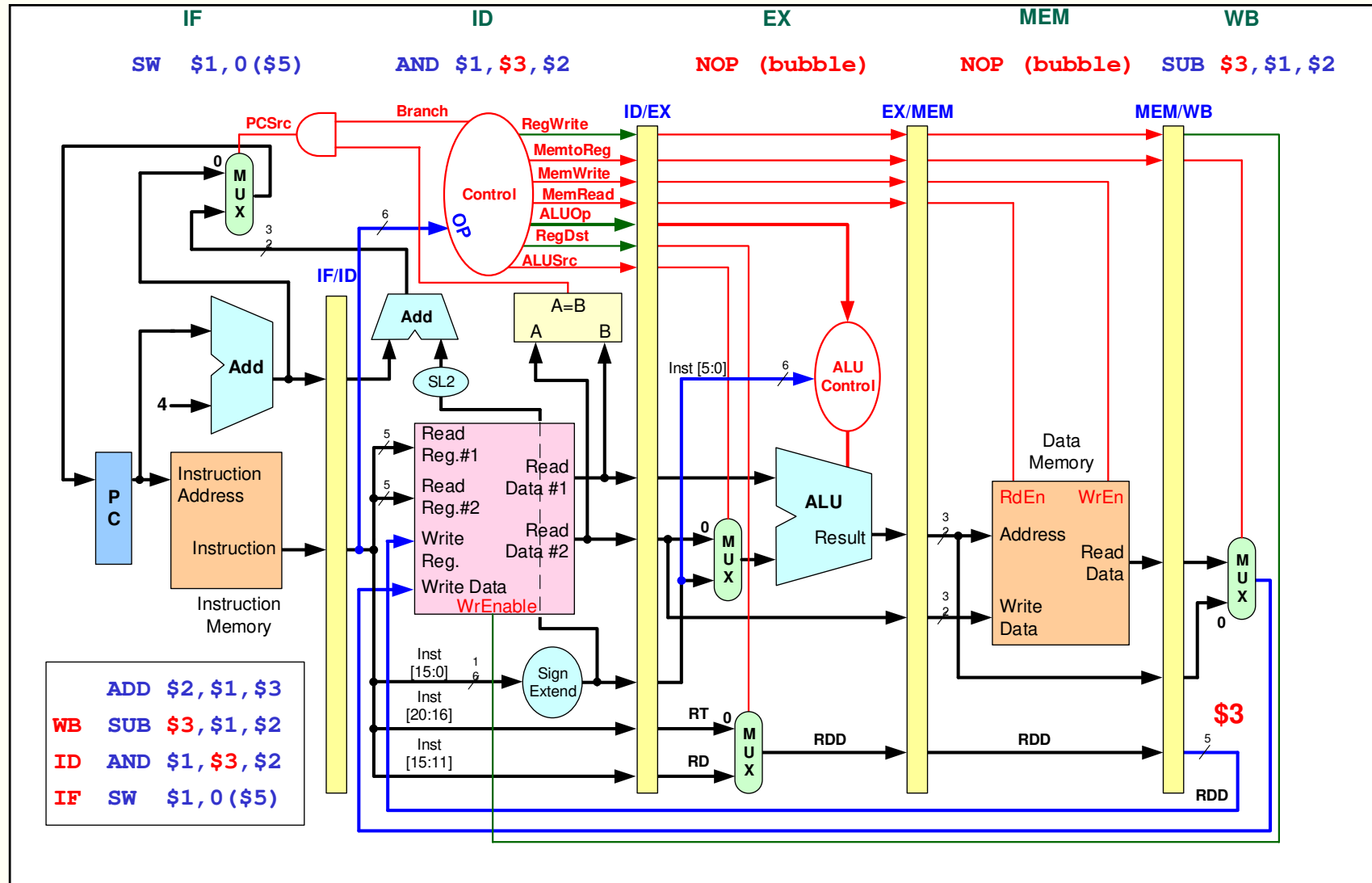
Hazards de dados resolvidos com *stalling* (7)



Hazards de dados resolvidos com *stalling* (8)



Hazards de dados resolvidos com *stalling* (9)



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, até, 2 ciclos de relógio (2 T); para o exemplo anterior, 6 T no total (3 *hazards* de dados):

$$\text{Texec_sem_stalls} = F + (N-1) = 5 + (4-1) = 8 \text{ T}$$

$$\text{Texec} = F + (N-1) + \text{Nr_stalls} = 5 + (4-1) + 6 = 14 \text{ T}$$

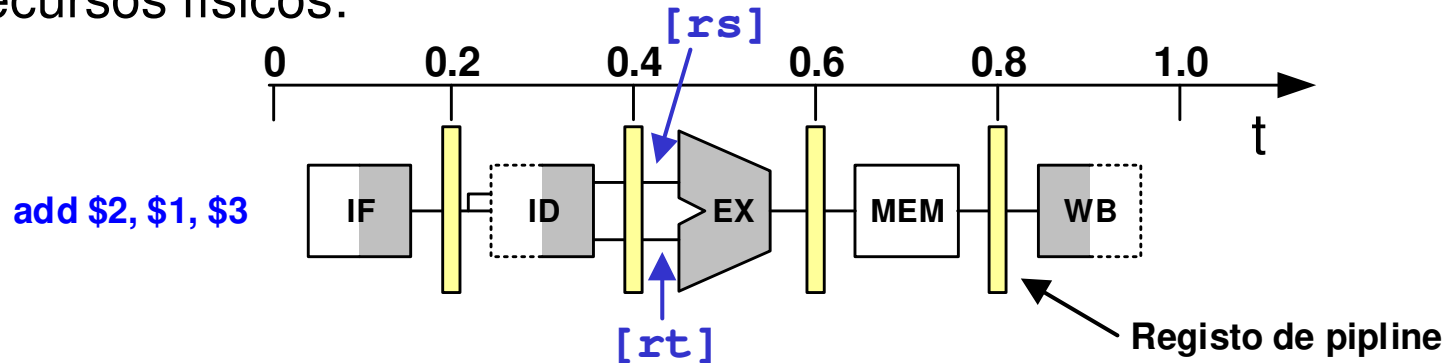
- Qual será então a solução?

Hazards de dados

- 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 instrução que produz o resultado para resolver o *hazard*
- Por exemplo, para as instruções do tipo R, logo que a operação da instrução que vai à frente seja realizada na ALU, (**EX**, 3º estágio), o resultado pode ser disponibilizado para a instrução seguinte
- Esta técnica de disponibilizar um resultado de uma instrução que ainda não terminou para uma instrução que vem a seguir na cadeia de *pipelining*, é designada por **forwarding** (ou *bypassing*)

Representação gráfica da cadeia de *pipelining*

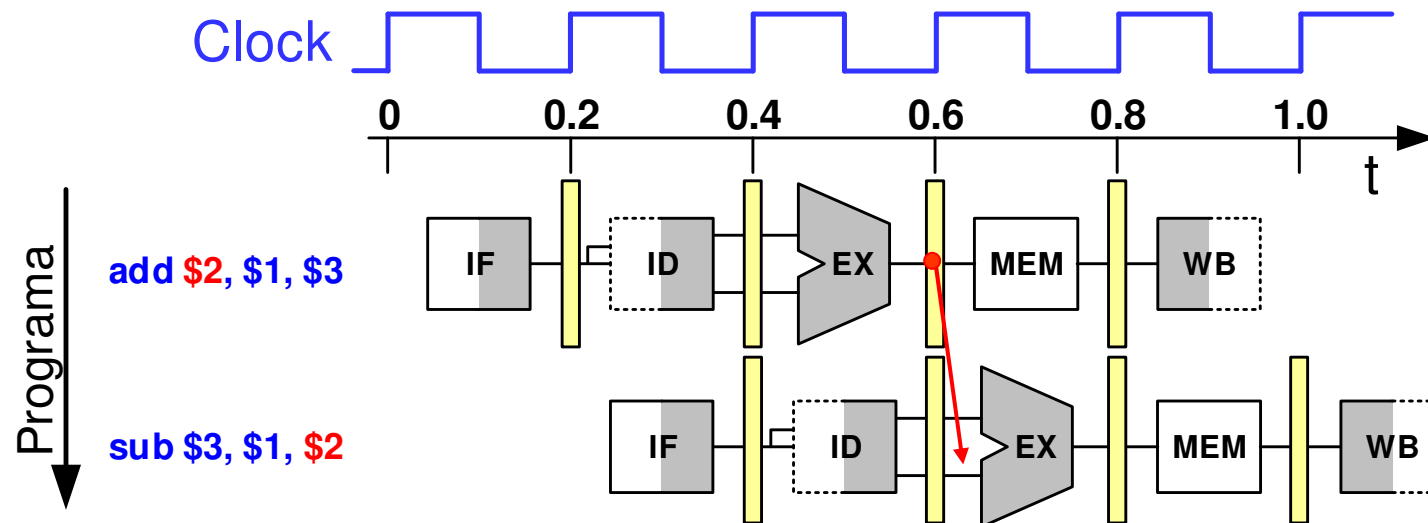
- Nesta representação gráfica usamos símbolos para representar os recursos físicos:



- IF (memória de instruções), ID (banco de registros), EX (ALU), MEM (memória de dados), WB (banco de registros)
- Metade cinzenta à direita indica uma operação de leitura do elemento de estado
- Metade cinzenta à esquerda indica uma operação de escrita no elemento de estado
- Um quadrado branco indica que o elemento de estado não está envolvido na execução da instrução

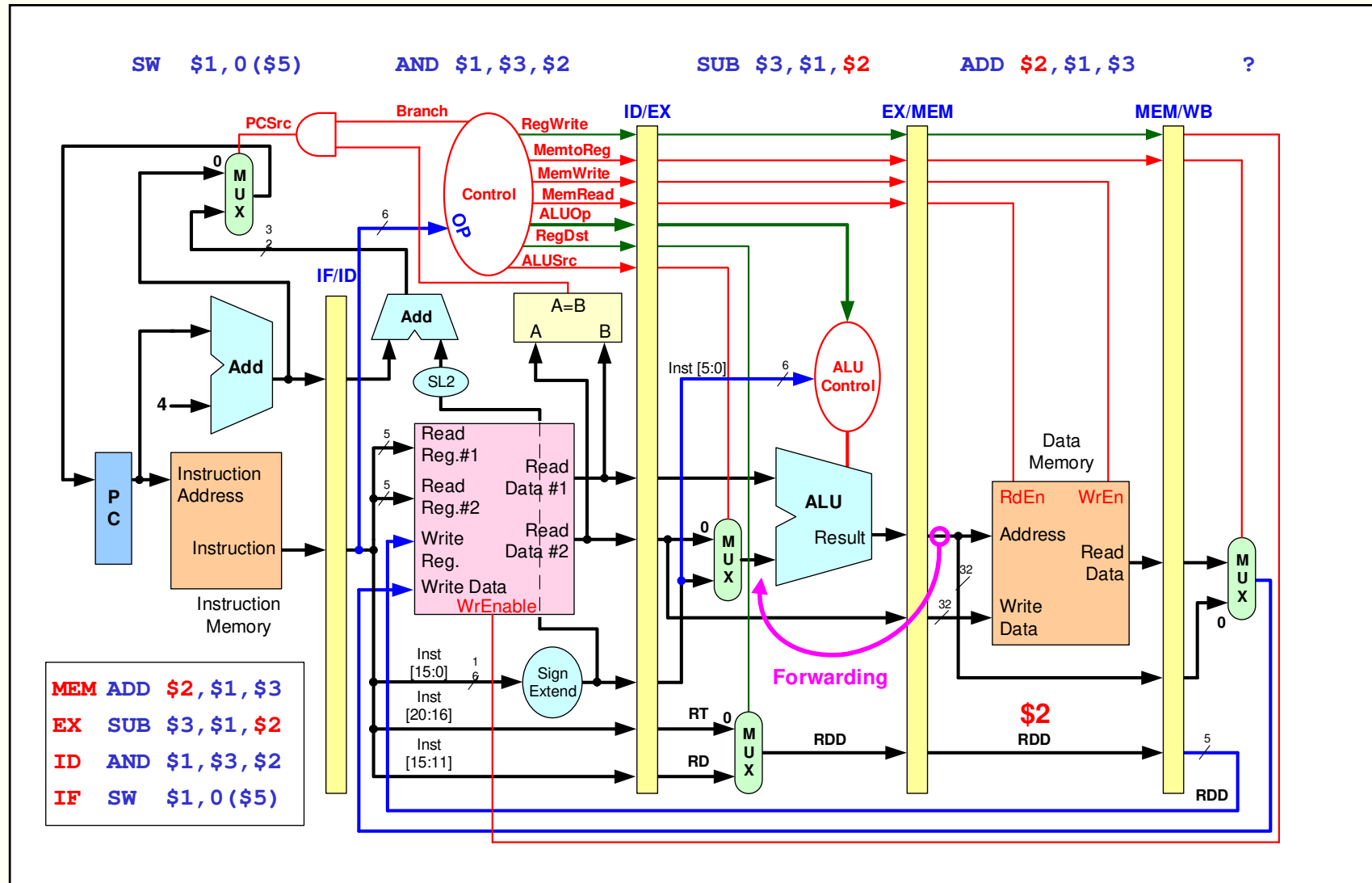
Hazards de dados

- Um exemplo anterior, em que se observou a existência de um *hazard* de dados, pode então ser representado graficamente por:



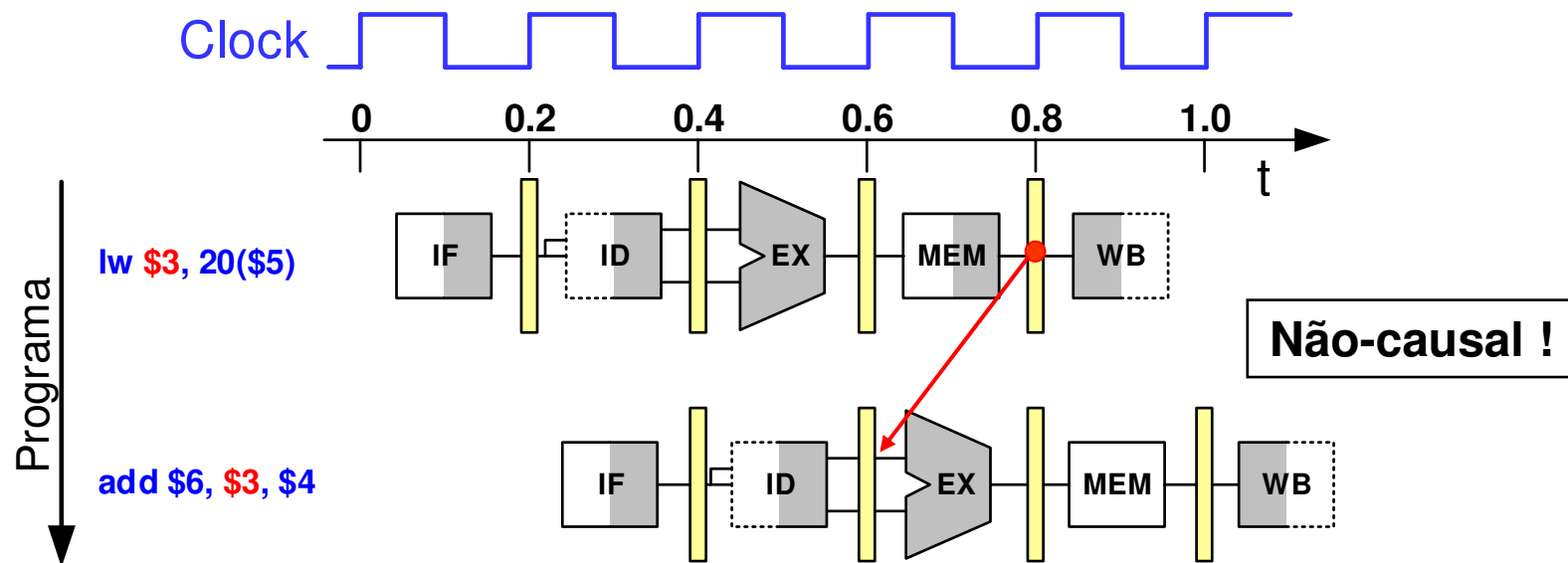
- O **forwarding** do valor presente no registo **EX/MEM** (resultado da instrução ADD) para a segunda entrada da ALU (estágio **EX**, instrução SUB) resolve o *hazard* de dados
 - Designamos esta operação por "*forwarding* de **EX/MEM** para **EX**" (para a segunda entrada da ALU)

Hazards de dados - forwarding



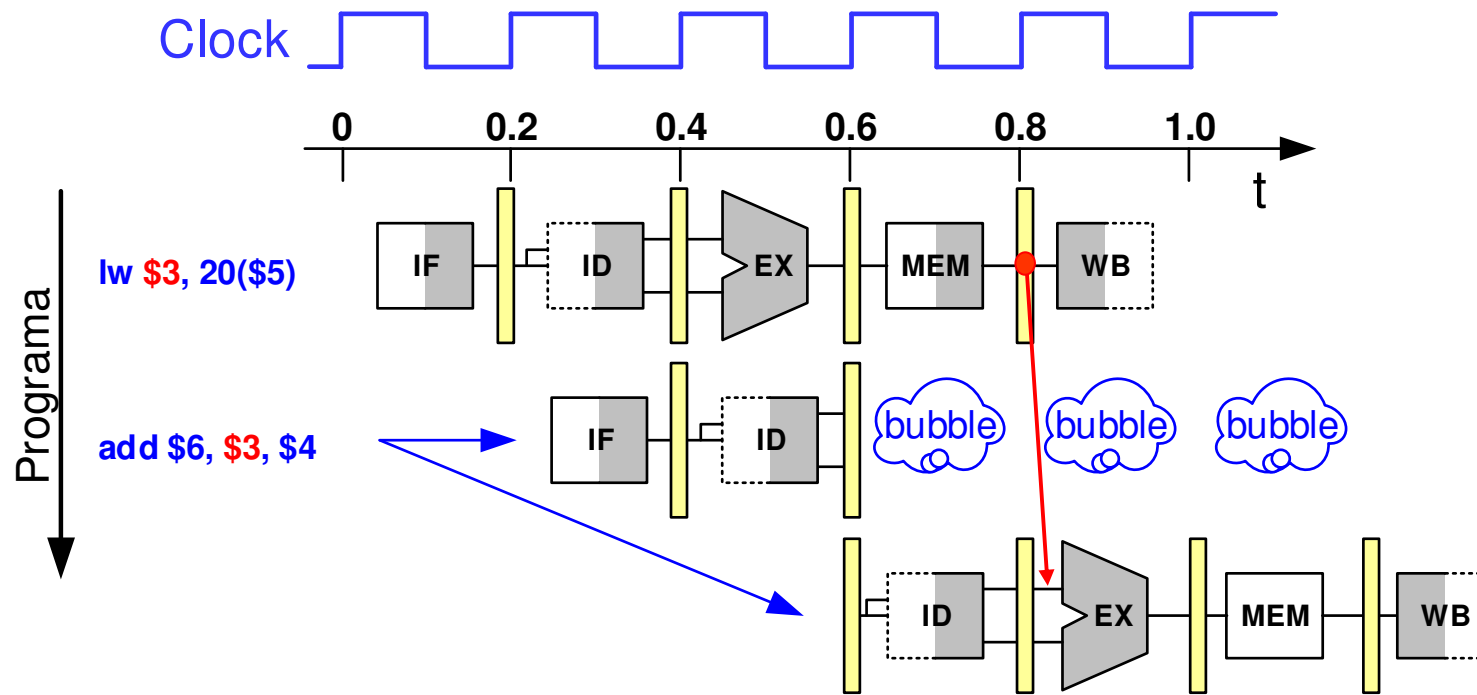
Hazards de dados

- 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 a situação do slide anterior, 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 primeira entrada da ALU

Hazards de dados – reordenação de instruções

- Algumas situações de *hazards* de dados podem ser atenuadas ou **resolvidas pelo compilador/assembler**, através da reordenação de instruções
- A reordenação não pode comprometer o resultado final
- Código original (exemplo):

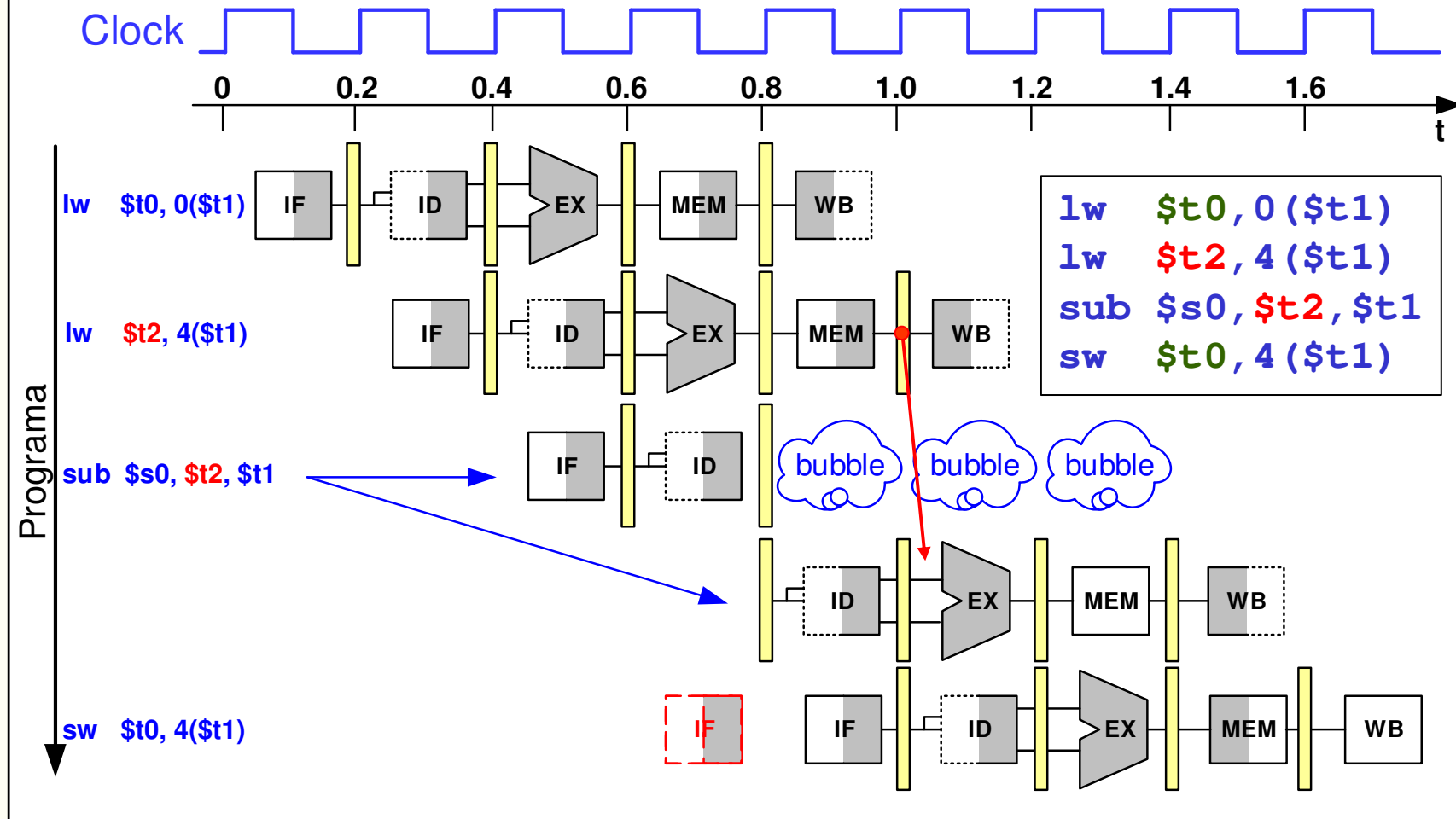
```
lw    $t0, 0($t1)
lw    $t2, 4($t1)
sub    $s0, $t2, $t1 # Stalling por hazard de dados (1T)
sw    $t0, 4($t1)
```

- 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)
```

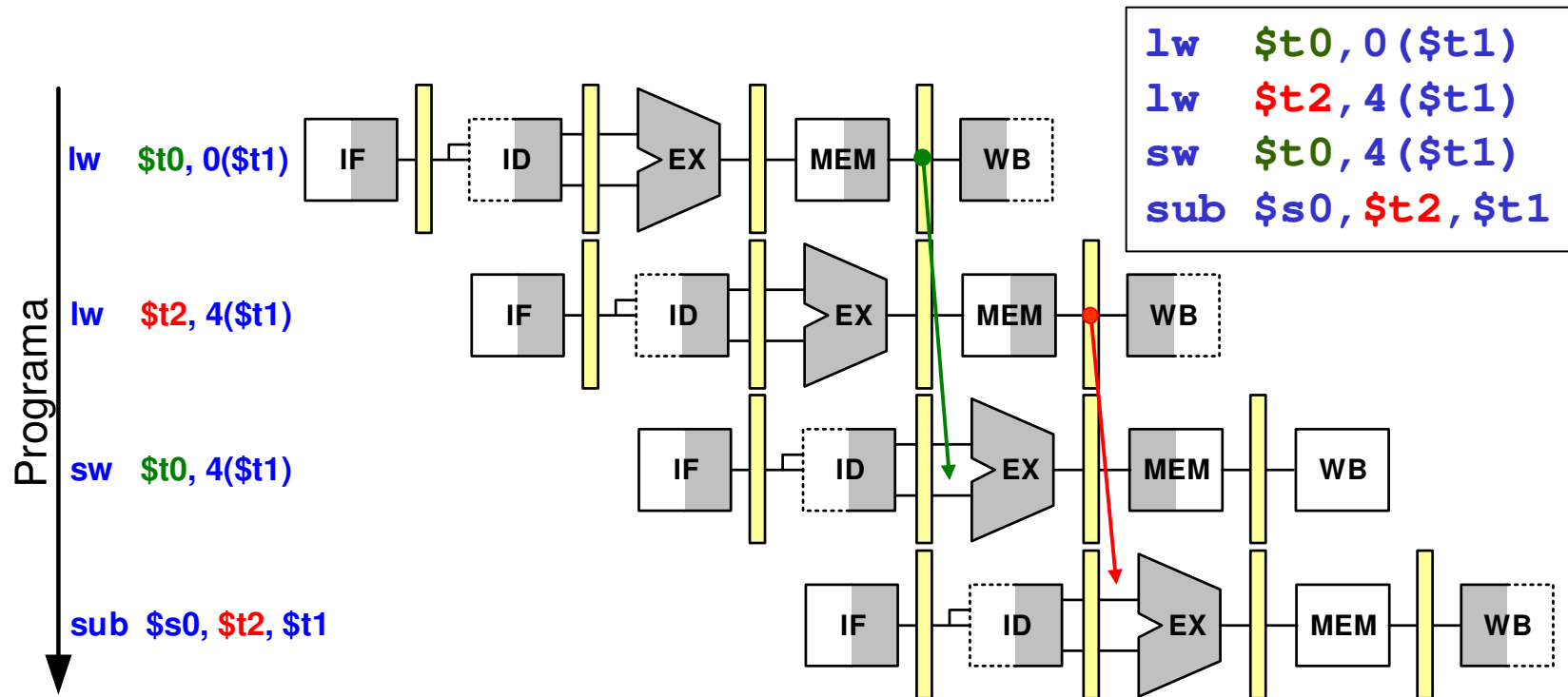
Hazards de dados – exemplo que gera *stalling*

- Situação de *stalling* por *hazard* de dados



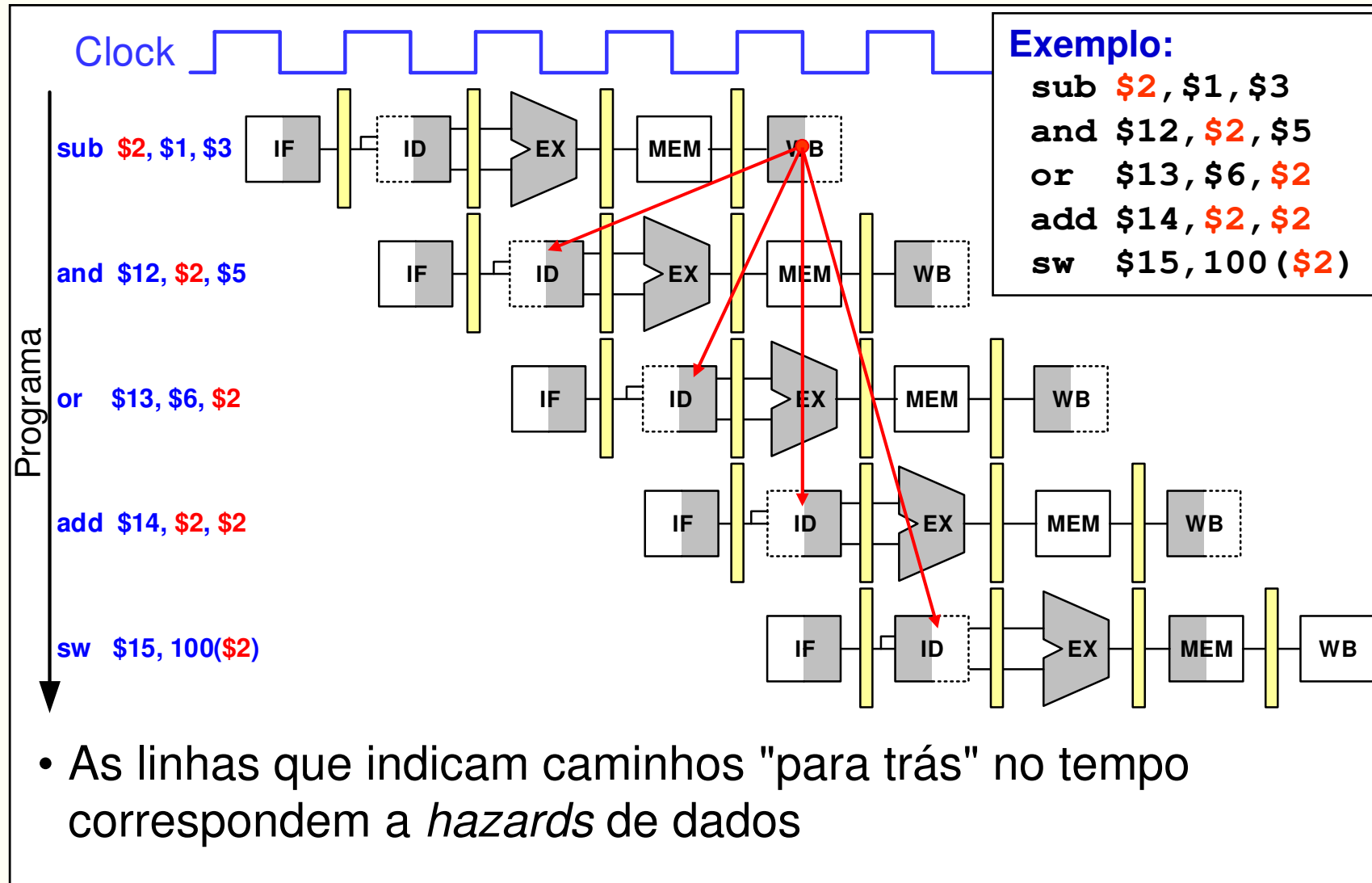
Hazards de dados

- A situação de *stalling* foi evitada pelo compilador/*assembler* através de reordenação. A reordenação gera um segundo *hazard* de dados que também é resolvido por *forwarding*

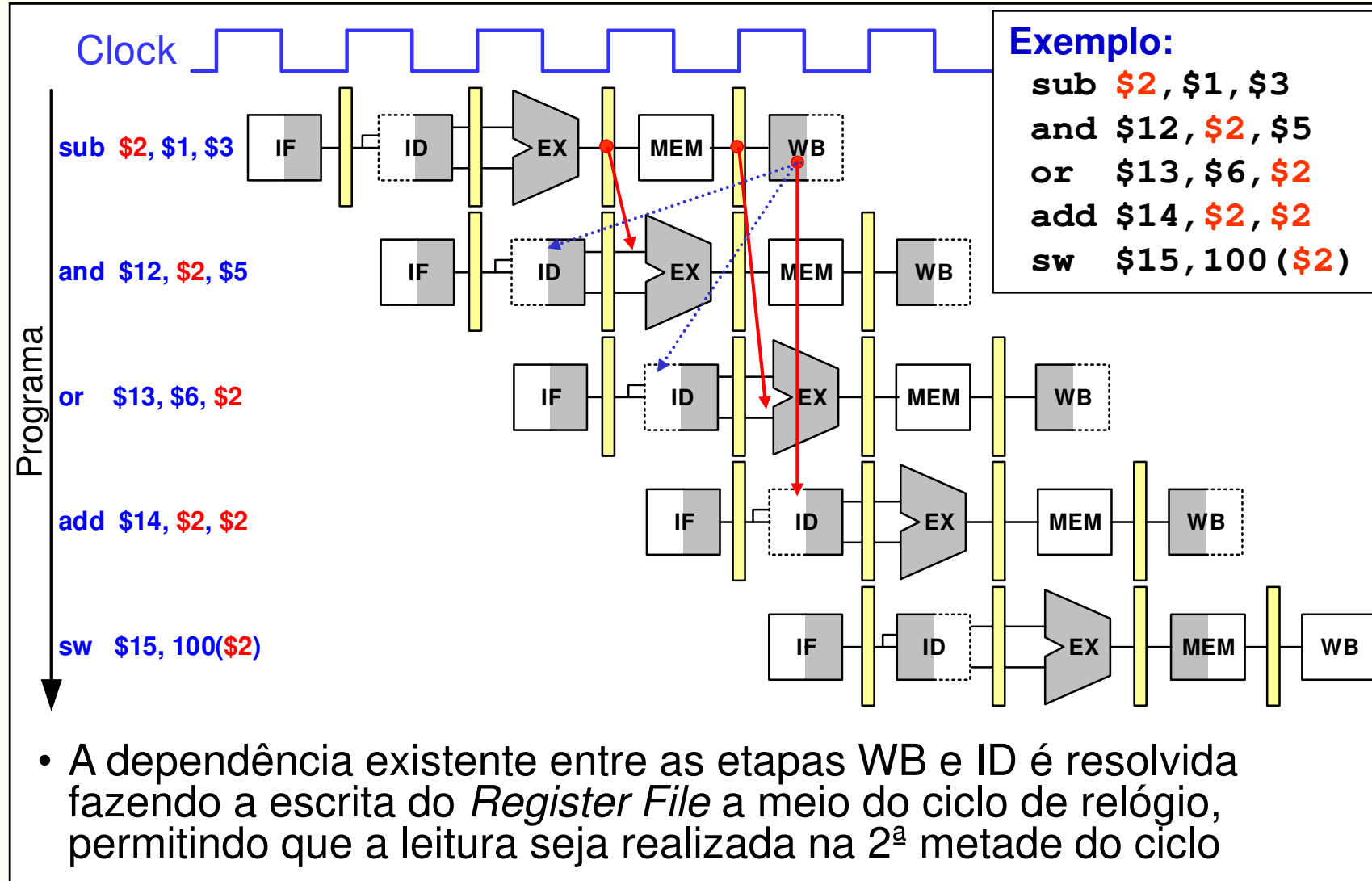


- A sequência reordenada executa em menos 1 ciclo de relógio

Hazards de dados – exemplo



Hazards de dados – exemplo

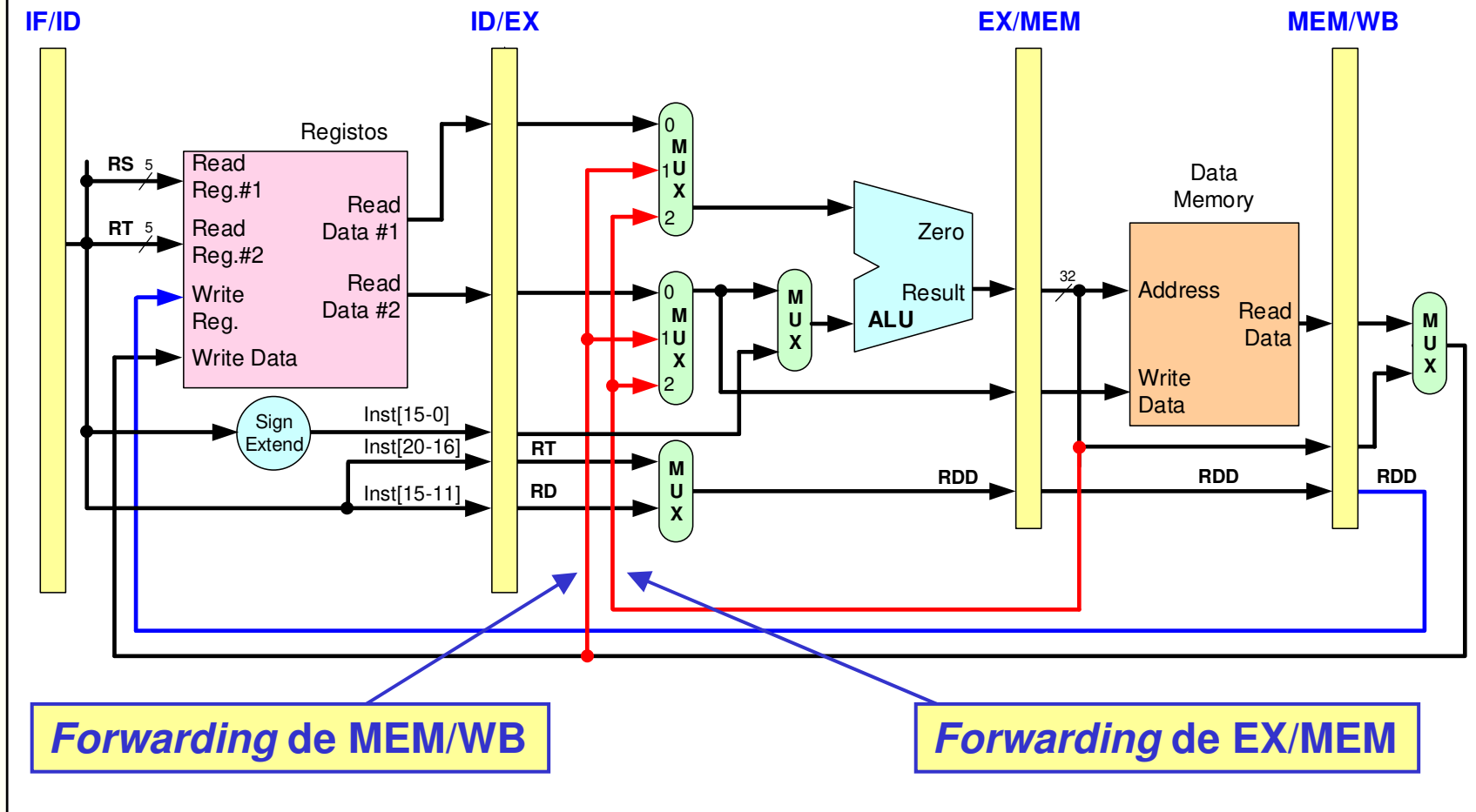


Hazards de dados – implementação do forwarding

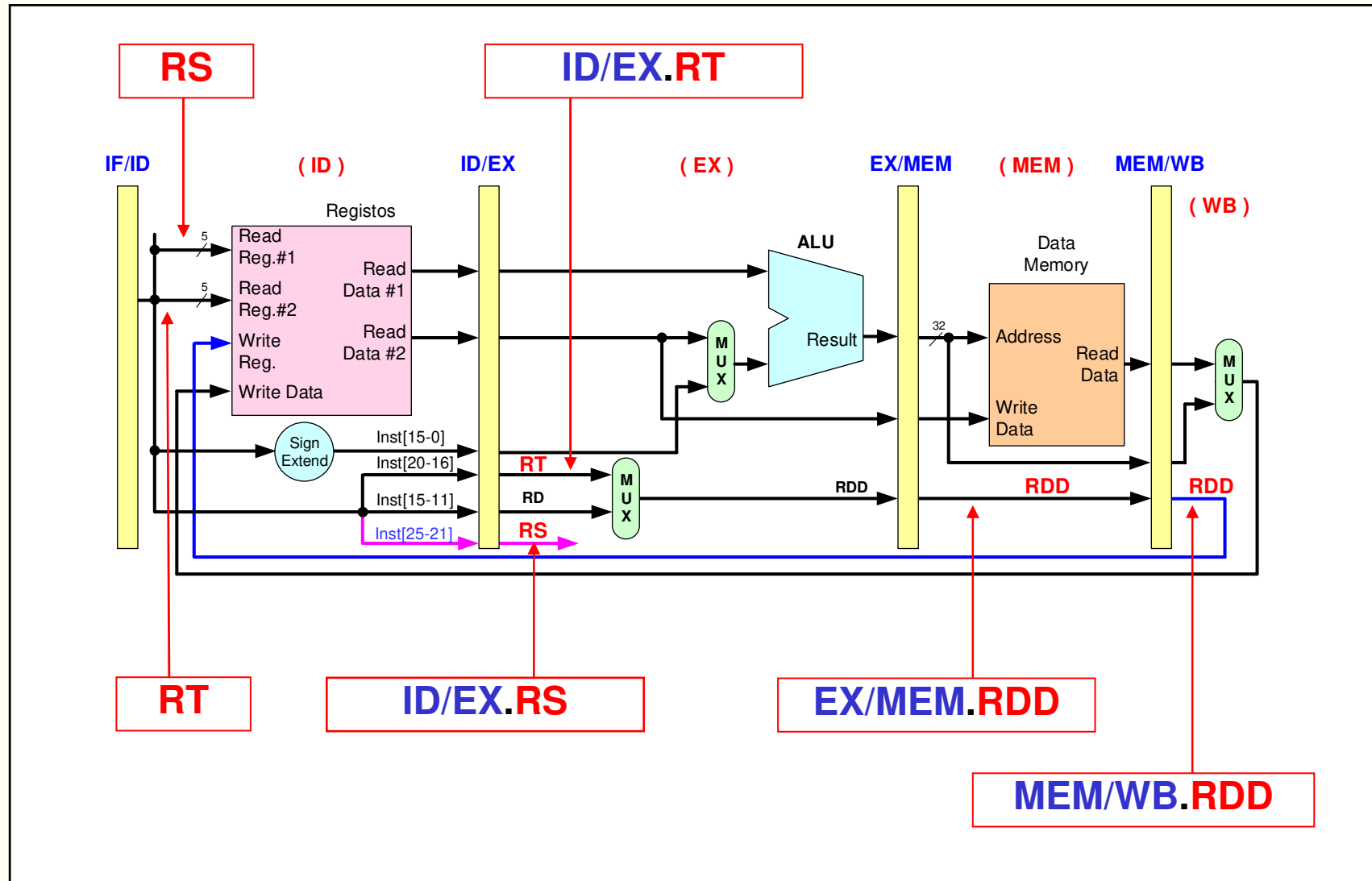
- Para resolver um *hazard* de dados através de **forwarding** é necessário:
 - **Detetar** a situação de *hazard*
 - **Encaminhar** o valor que se encontra num estágio mais avançado do *pipeline* para onde ele é necessário
- A resolução de uma parte significativa dos *hazards* de dados é feita através do encaminhando de valores para o estágio **EX**:
 - *forwarding* de **EX/MEM** para **EX** e de **MEM/WB** para **EX**
- As instruções de *branch* necessitam dos valores corretos dos registos no estágio **ID**
 - *forwarding* de **EX/MEM** para **ID**

Hazards de dados – encaminhamento

- Forwarding de **EX/MEM** para **EX** e de **MEM/WB** para **EX**



Hazards de dados – detecção



Hazards de dados – deteção

- As situações de *hazard* de dados, em que há necessidade de encaminhar valores para o estágio **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**

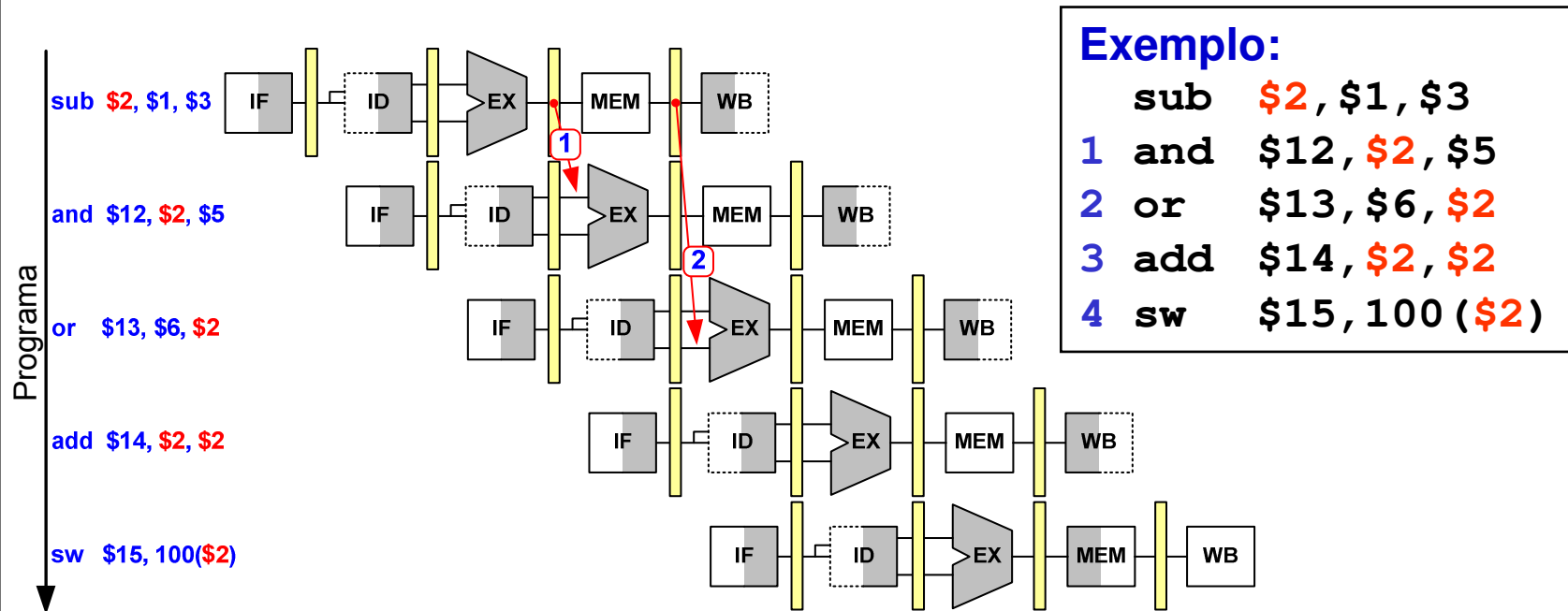
WB add **\$1**, \$2, \$3

M add \$6, \$2, \$3

EX sub \$4, \$5, **\$1**

Hazards de dados – detecção

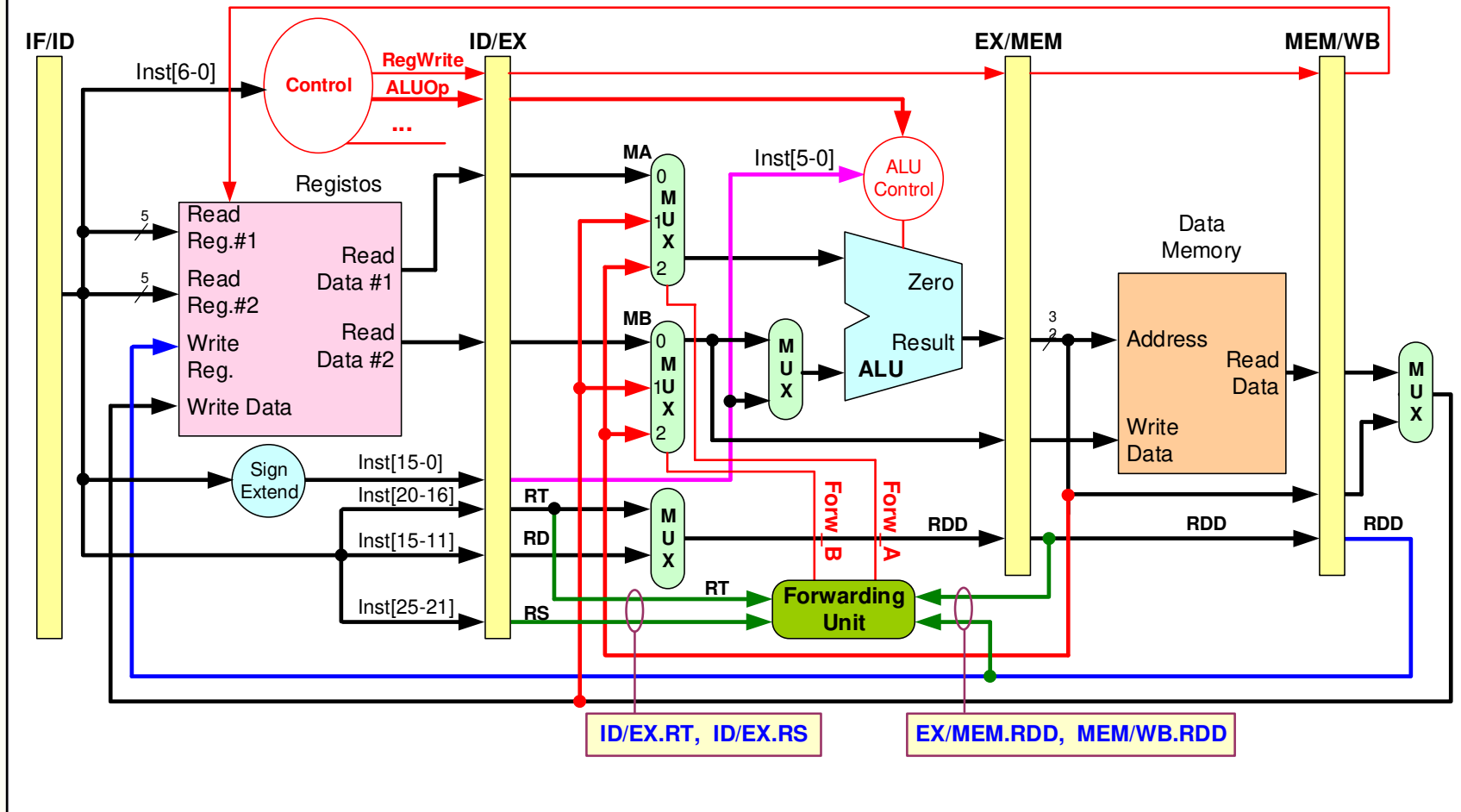
- A situação 3 é resolvida sem *forwarding* (não se considera *hazard*)
- A situação 4 não corresponde a um *hazard* de dados



- As situações de *hazard* de dados 1 e 2 podem ser detetadas por:
 1. **EX/MEM.RDD == ID/EX.RS** (**EX/MEM.RDD** = \$2, **ID/EX.RS** = \$2)
 2. **MEM/WB.RDD == ID/EX.RT** (**MEM/WB.RDD** = \$2, **ID/EX.RT** = \$2)

Hazards de dados – unidade de controlo de *forwarding*

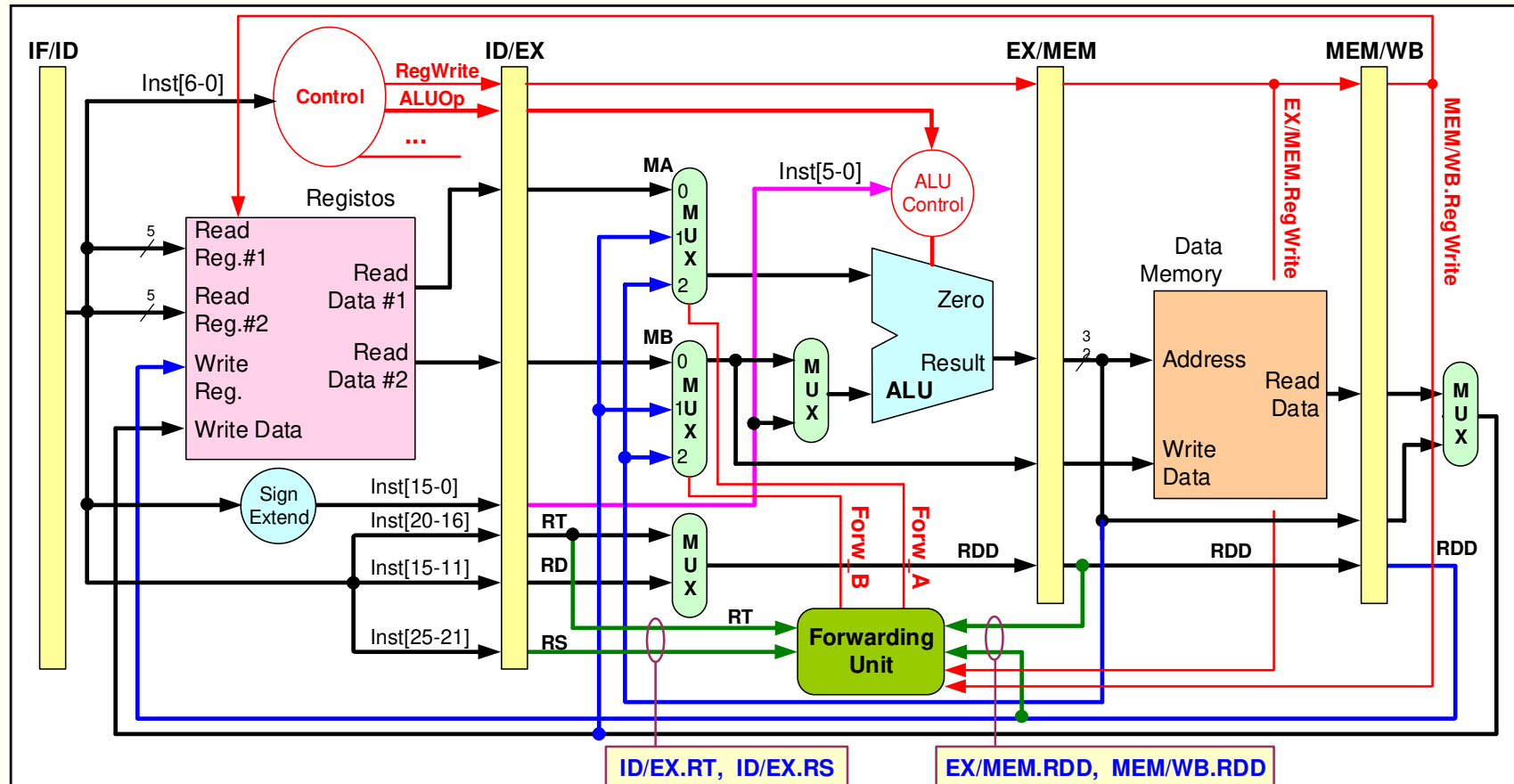
- Unidade de controlo de *forwarding*, simplificada



Hazards de dados – unidade de controlo de forwarding

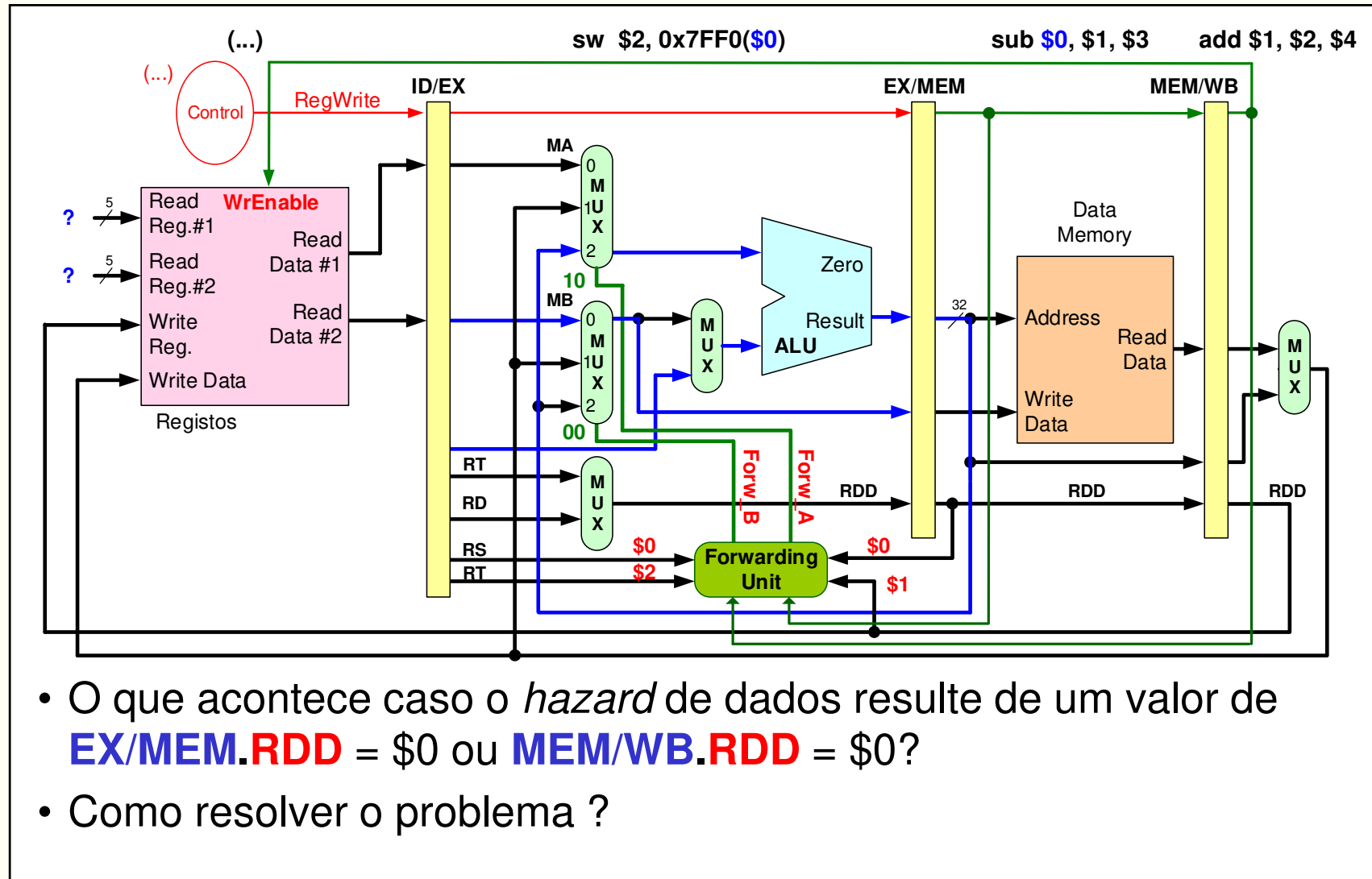
- A simples comparação dos registos não é suficiente para a correta deteção das situações de *hazard* de dados
- O sinal de controlo que permite a escrita no banco de registos (RegWrite) tem igualmente que ser avaliado:
 - Instrução na fase **MEM** que escreve o resultado num registo (**RegWrite='1'**) igual ao 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 r

```

architecture Behavioral of ForwardingUnit is
begin
  process (all)
  begin
    Forw_A <= "00"; -- no hazard
    Forw_B <= "00"; -- no hazard
    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;

```

```

1  add  $2, $3, $5
2  add  $2, $2, $6
3  sub  $4, $2, $7

```

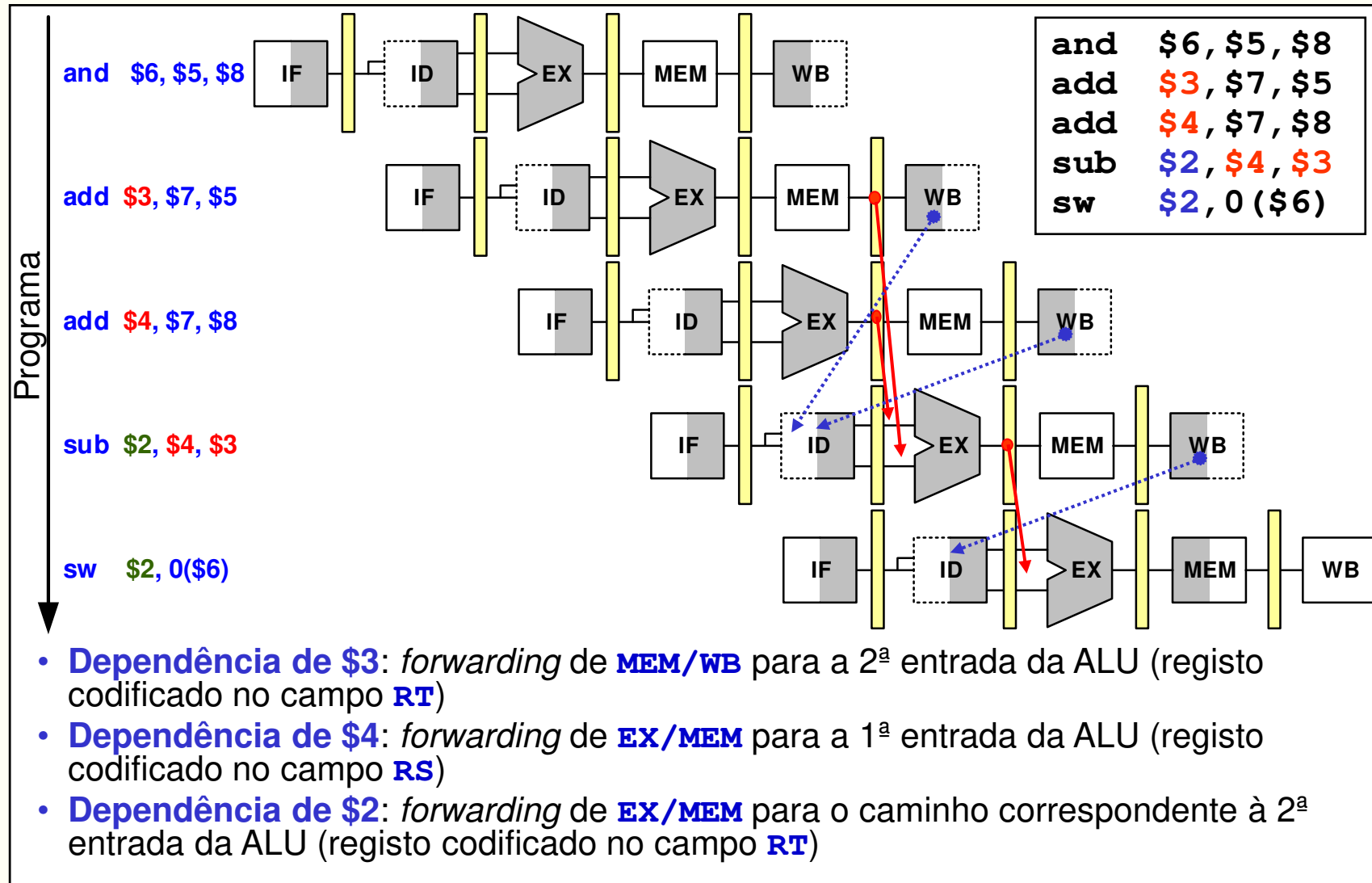
00 – encaminhar valor lido do banco de registos
 01 – encaminhar o valor proveniente do registo **MEM/WB**
 10 – encaminhar o valor proveniente do registo **EX/MEM**

Exemplo de *forwarding*

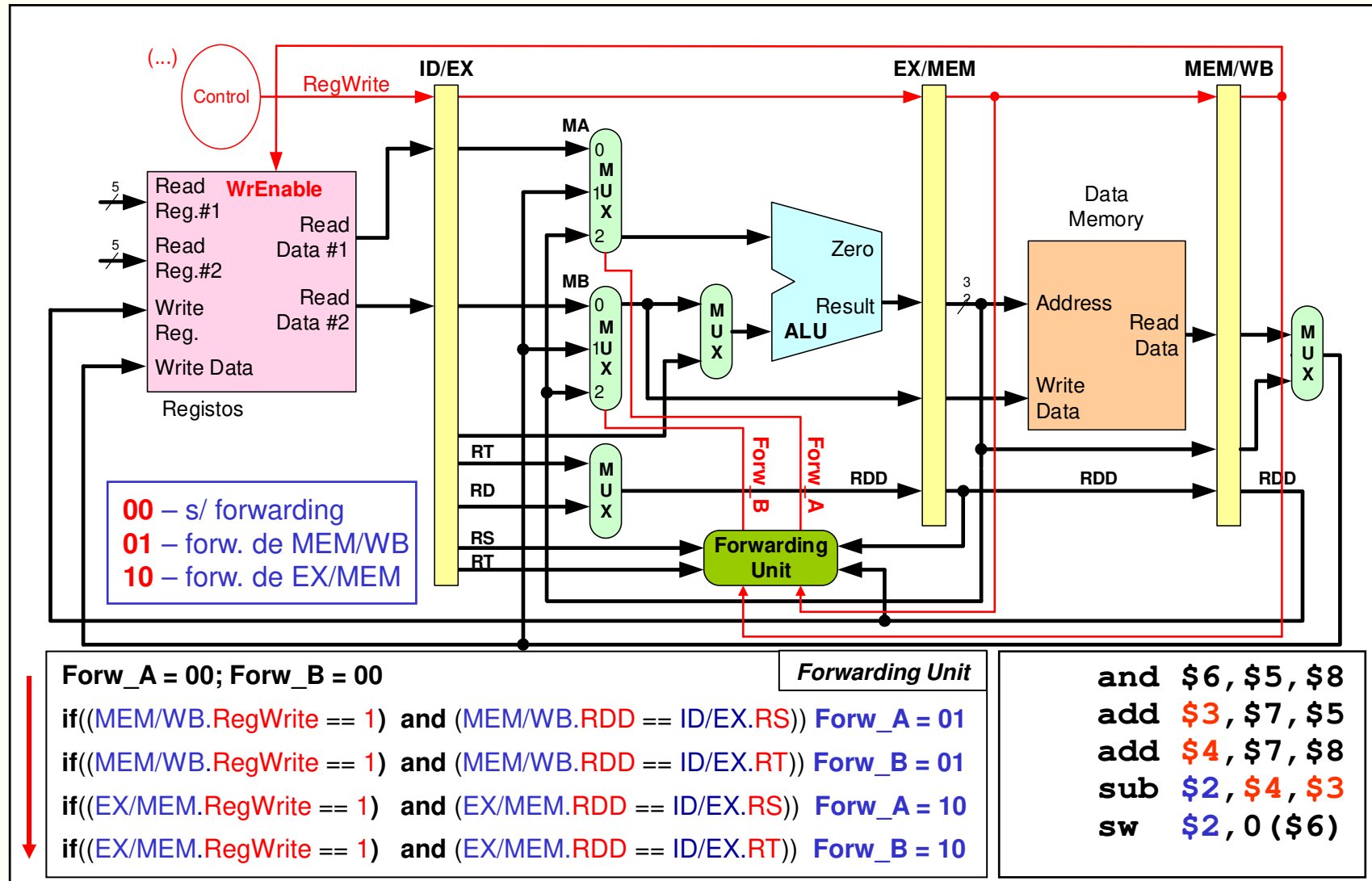
```
and $6, $5, $8
add $3, $7, $5
add $4, $7, $8
sub $2, $4, $3    # Hazard de dados: $3, $4
sw  $2, 0($6)     # Hazard de dados: $2
```

- A instrução "**sub** \$2, \$4, \$3" apresenta duas situações de *hazards* de dados:
 - dependência do registro \$4 (**add** \$4, \$7, \$8)
 - dependência do registro \$3 (**add** \$3, \$7, \$5)
- A instrução "**sw** \$2, 0(\$6)" apresenta igualmente uma situação de *hazard* de dados (dependência em \$2, **sub** \$2, \$4, \$3)

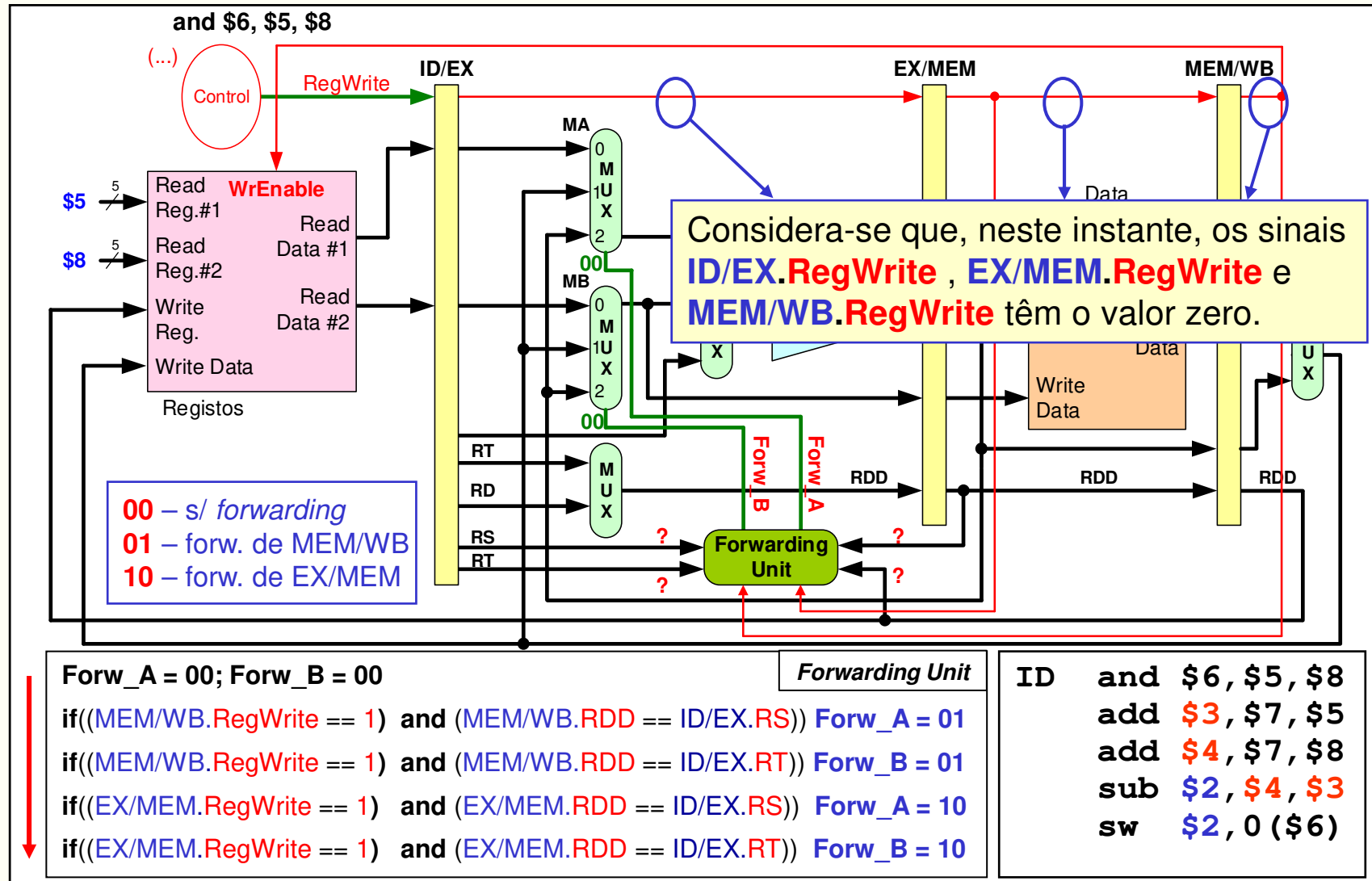
Exemplo de *forwarding*



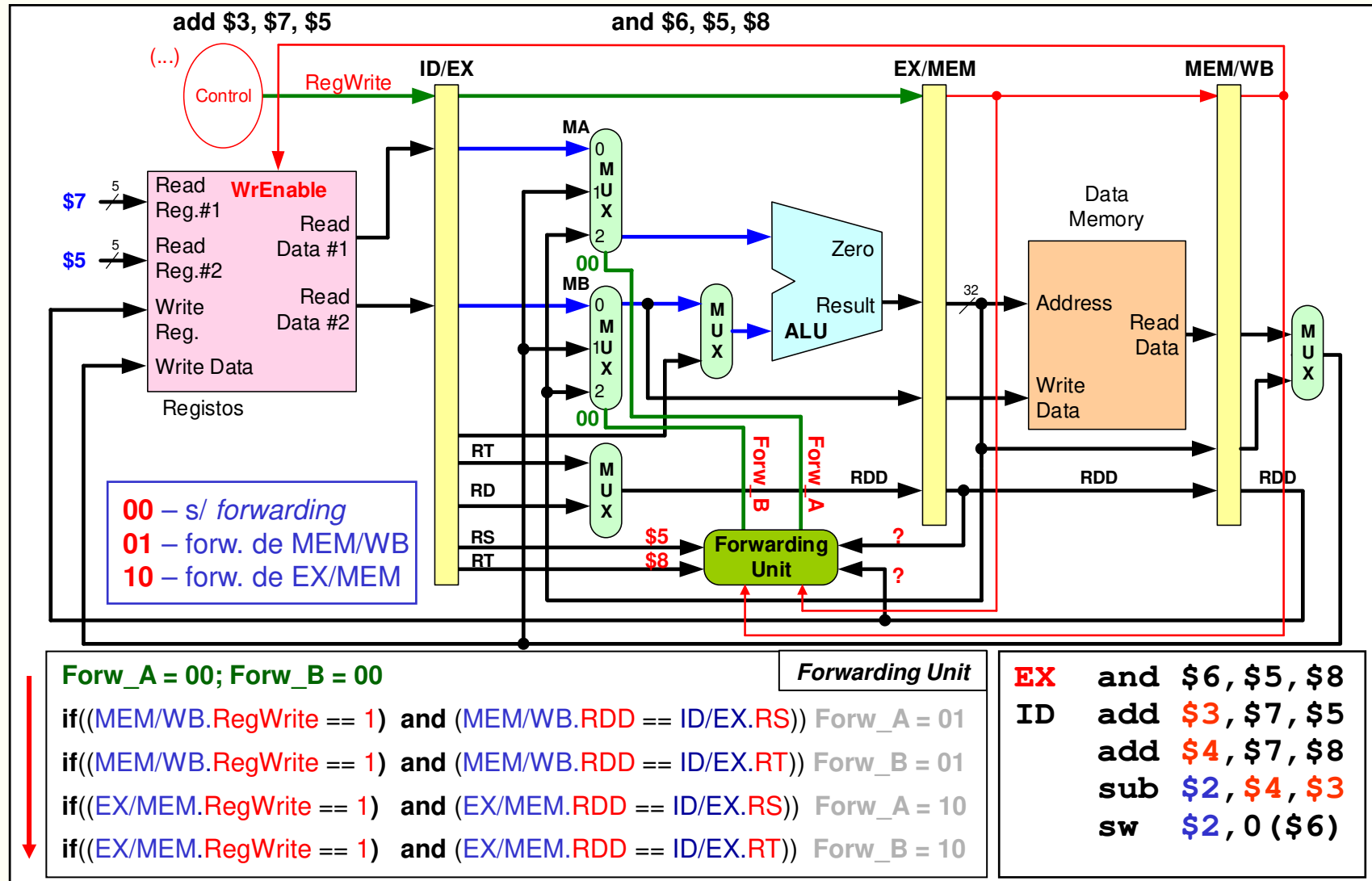
Exemplo de forwarding



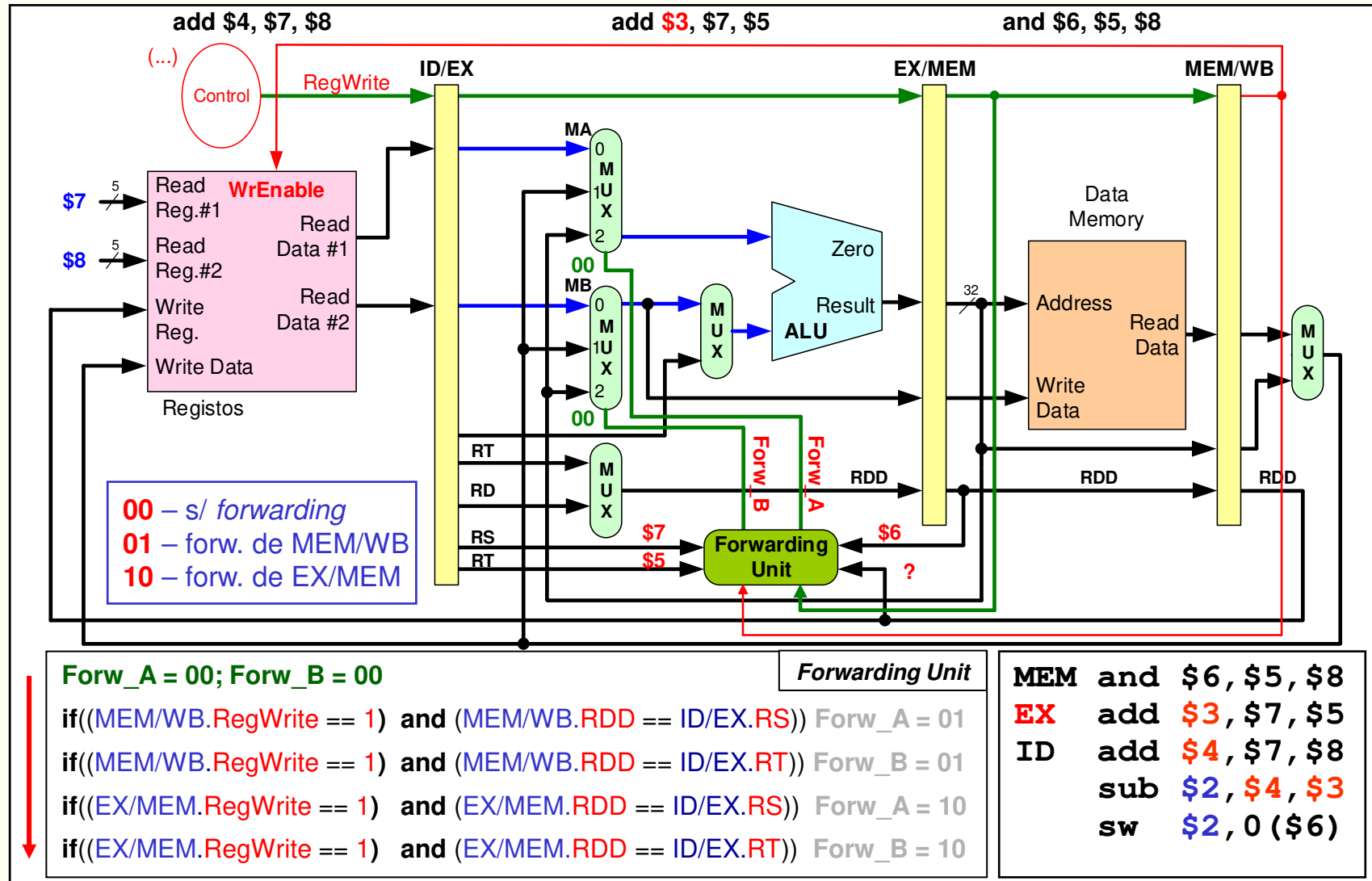
Exemplo de forwarding



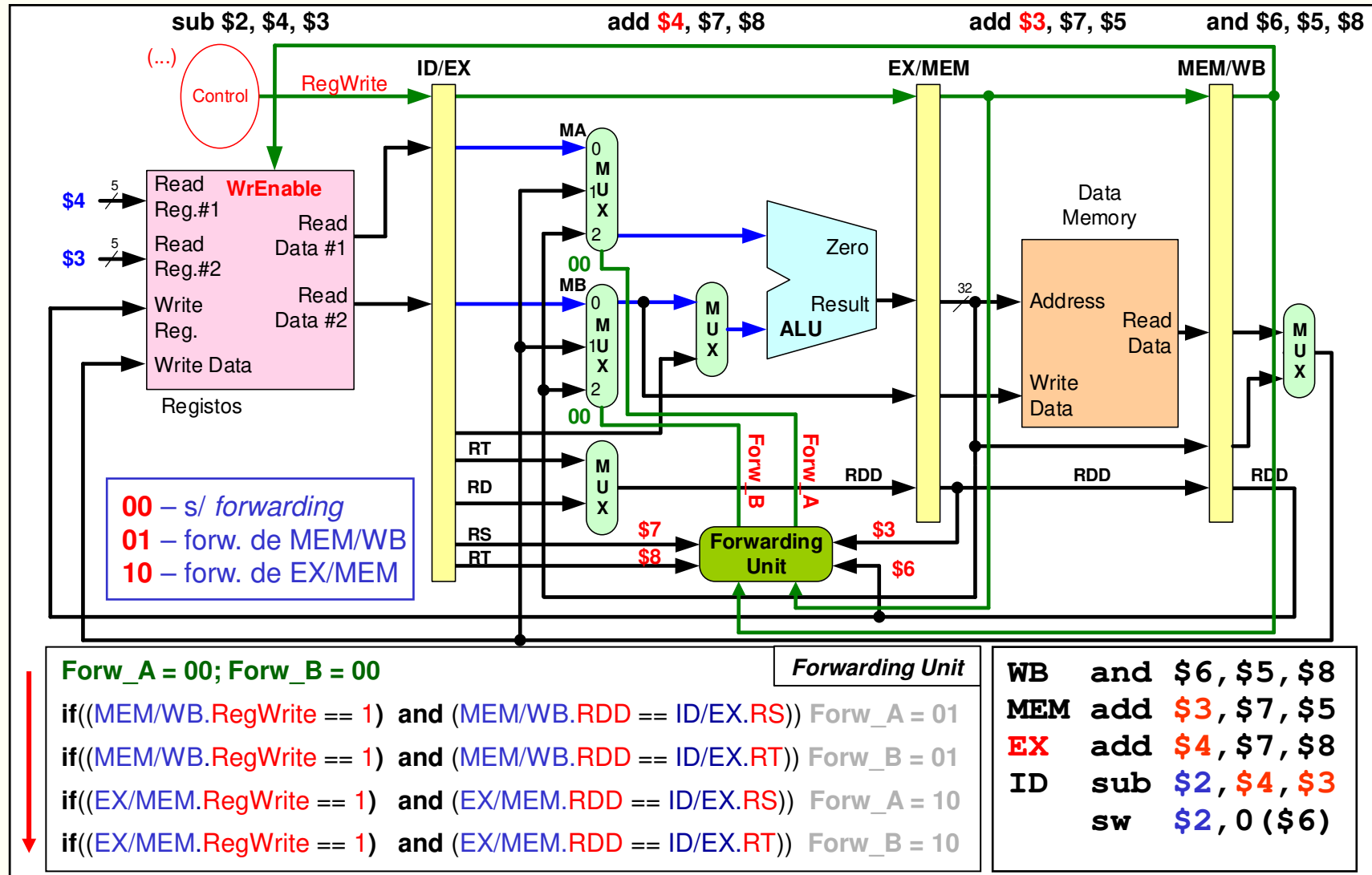
Exemplo de forwarding



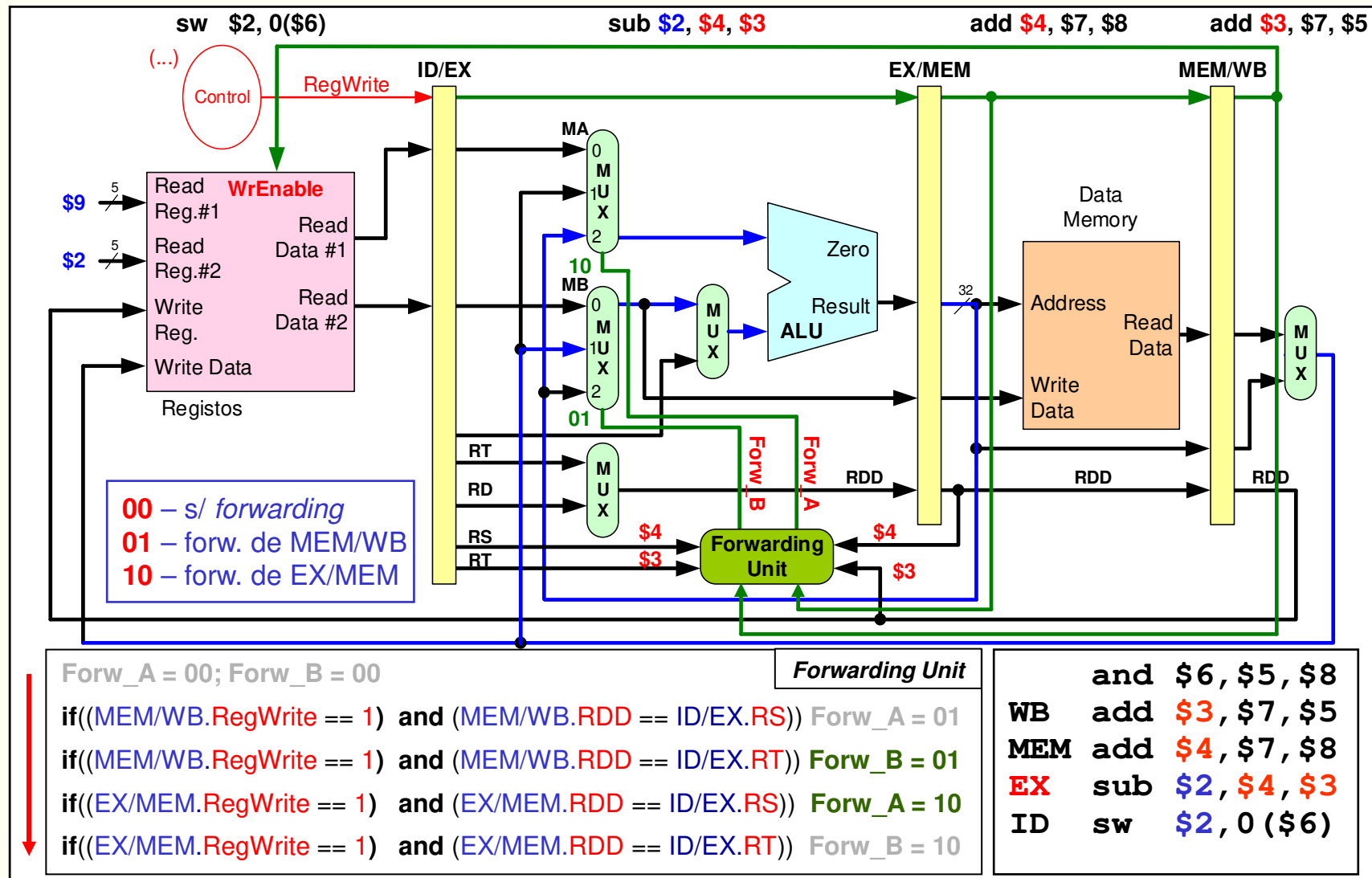
Exemplo de forwarding



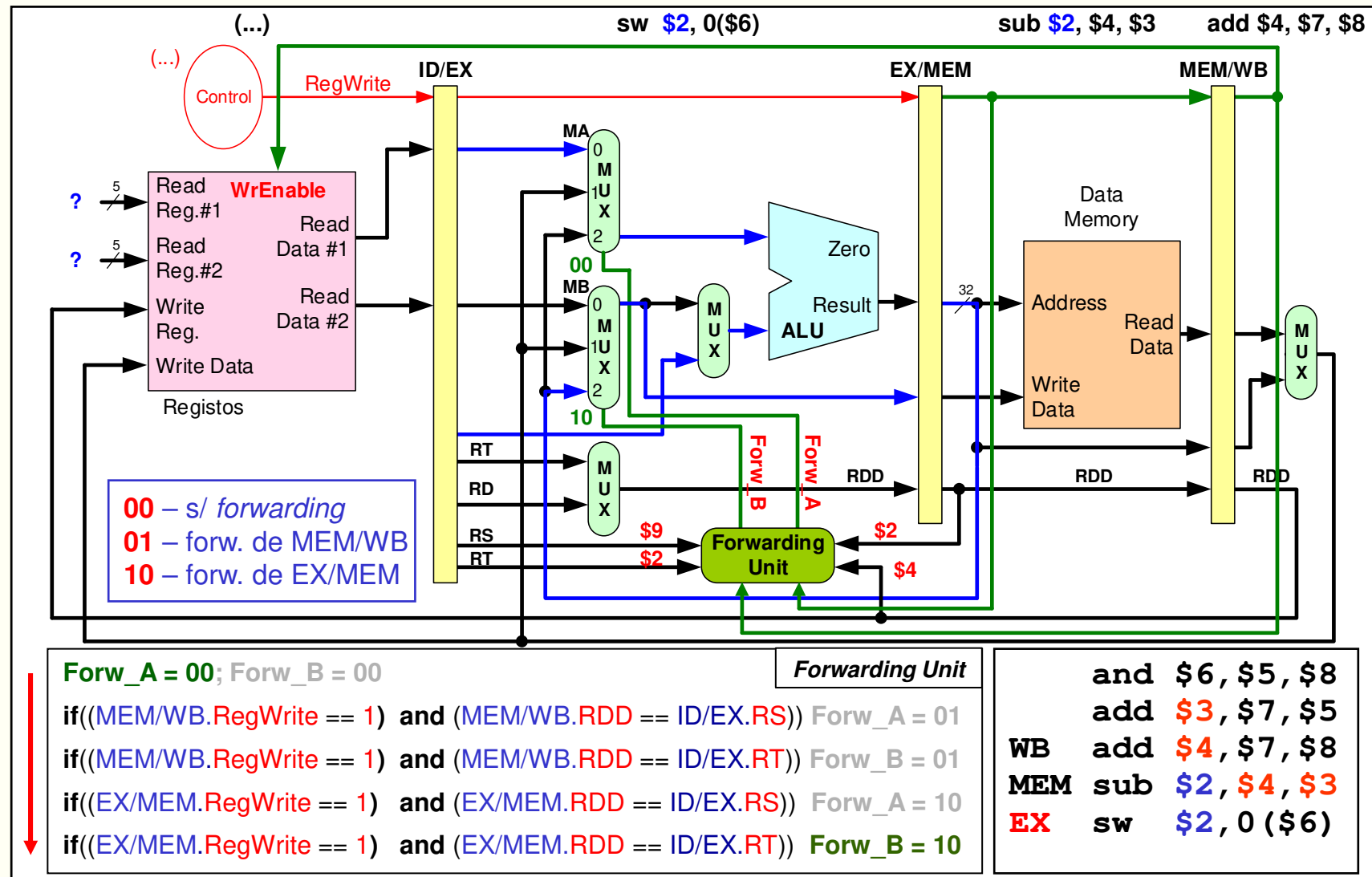
Exemplo de forwarding



Exemplo de forwarding



Exemplo de *forwarding*



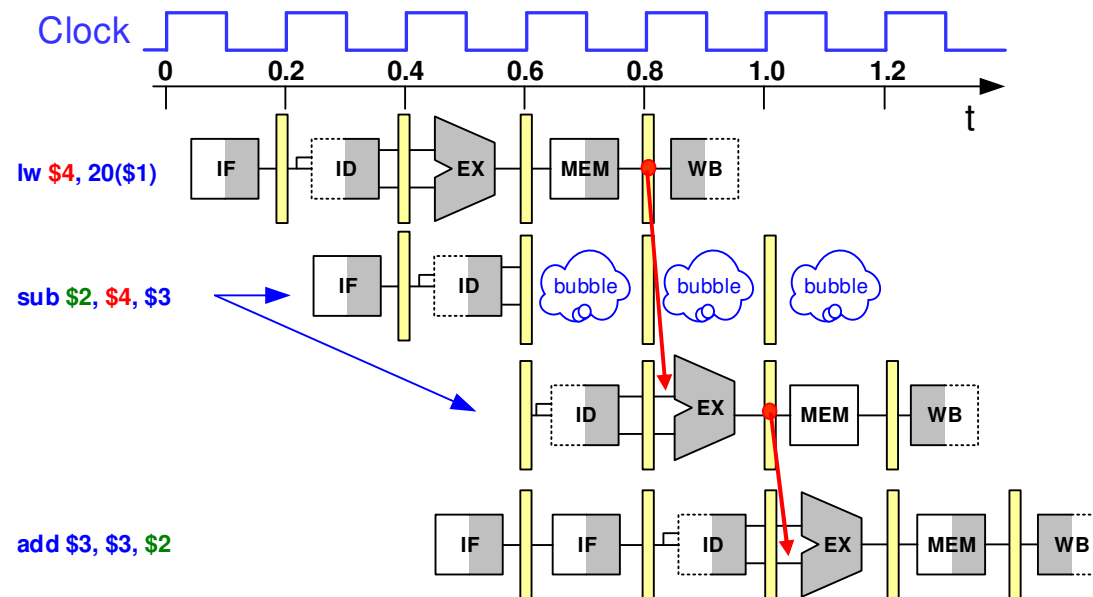
Dependência que obriga a *stalling*

- Como já observado anteriormente, uma situação de dependência que obriga a *stalling* é a que resulta de uma instrução aritmética ou lógica executada a seguir e na dependência de uma instrução LW:

lw \$4, 20(\$1) # valor disponível em WB

sub \$2, \$4, \$3 # Stall 1T, Forw. MEM/WB > EX

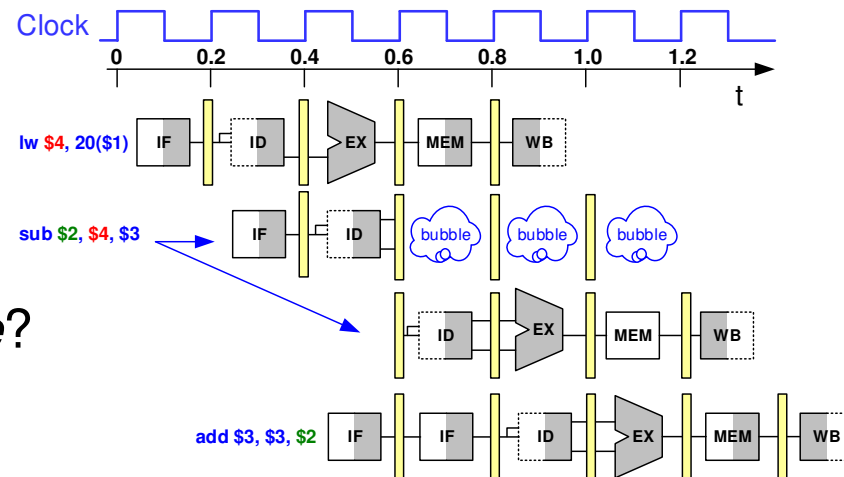
add \$3, \$3, \$2 # Forw. EX/MEM > EX



Dependência que obriga a *stalling*

- A situação de *stalling* tem que ser detetada quando a instrução tipo R está na sua fase ID (e o LW na fase EX). Como detetar?
- De forma simplificada:

(ID/EX.MemRead == 1) and (ID/EX.RDD == RS or ID/EX.RDD == RT)

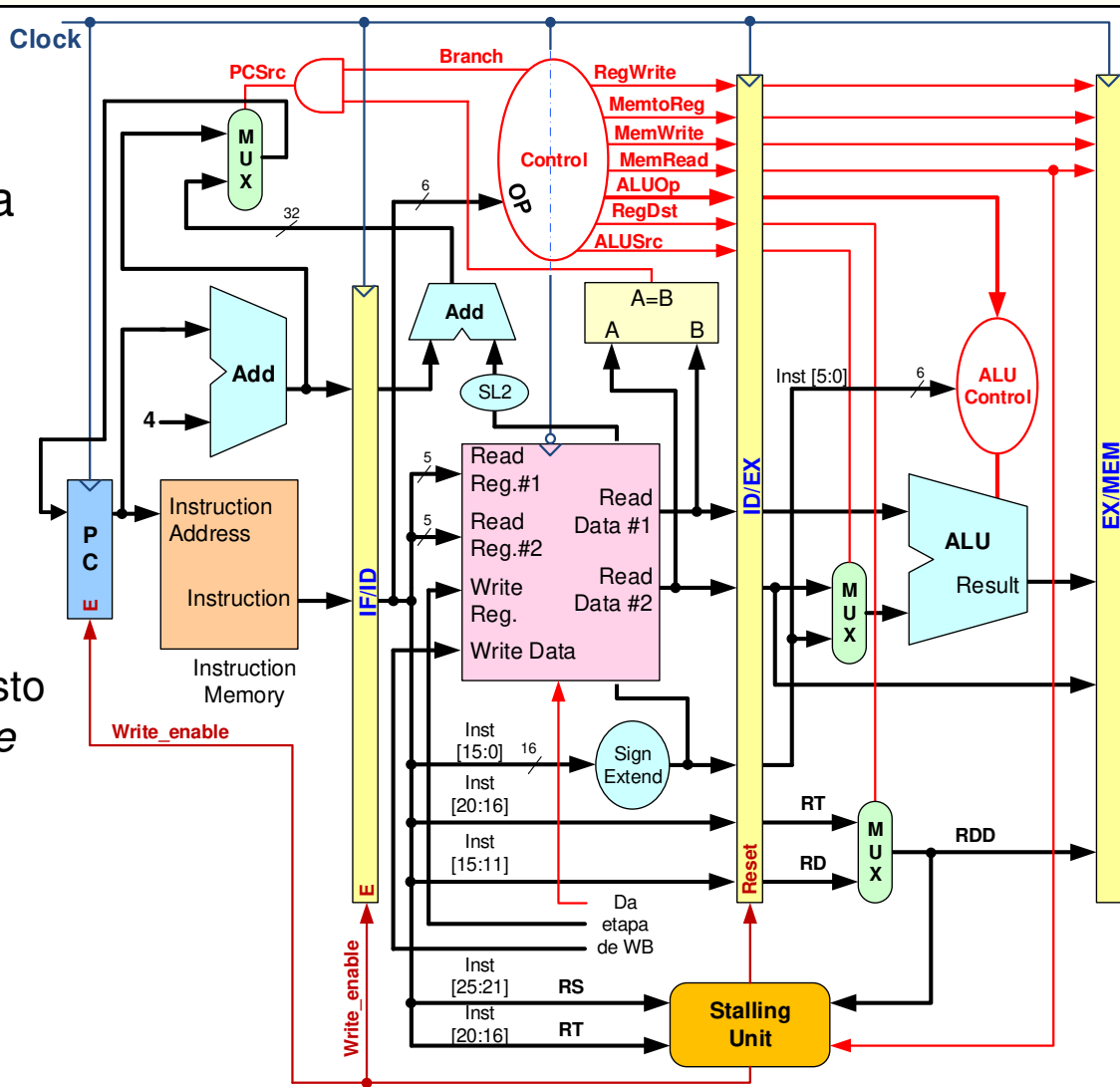


- Como fazer o *stall* do pipeline?

- **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 *stalling*

- Inserir *bubble* na etapa EX:
 - Ativar o *Reset* (síncrono) no registo ID/EX
- Congelar, durante 1 ciclo de relógio, as etapas IF e ID
 - Impedir a escrita no registo IF/ID - desativar o *Enable* do registo IF/ID
 - Impedir que seja feita a atualização do PC - desativar o *Enable* do registo PC



Unidade de controlo de *stalling* – VHDL

- Unidade de controlo de *stalling* simplificada, que contempla apenas a situação de dependência entre uma instrução LW e uma instrução tipo R
 - Instrução do tipo R: **(ALUOp == "10") and (RegWrite == 1)**

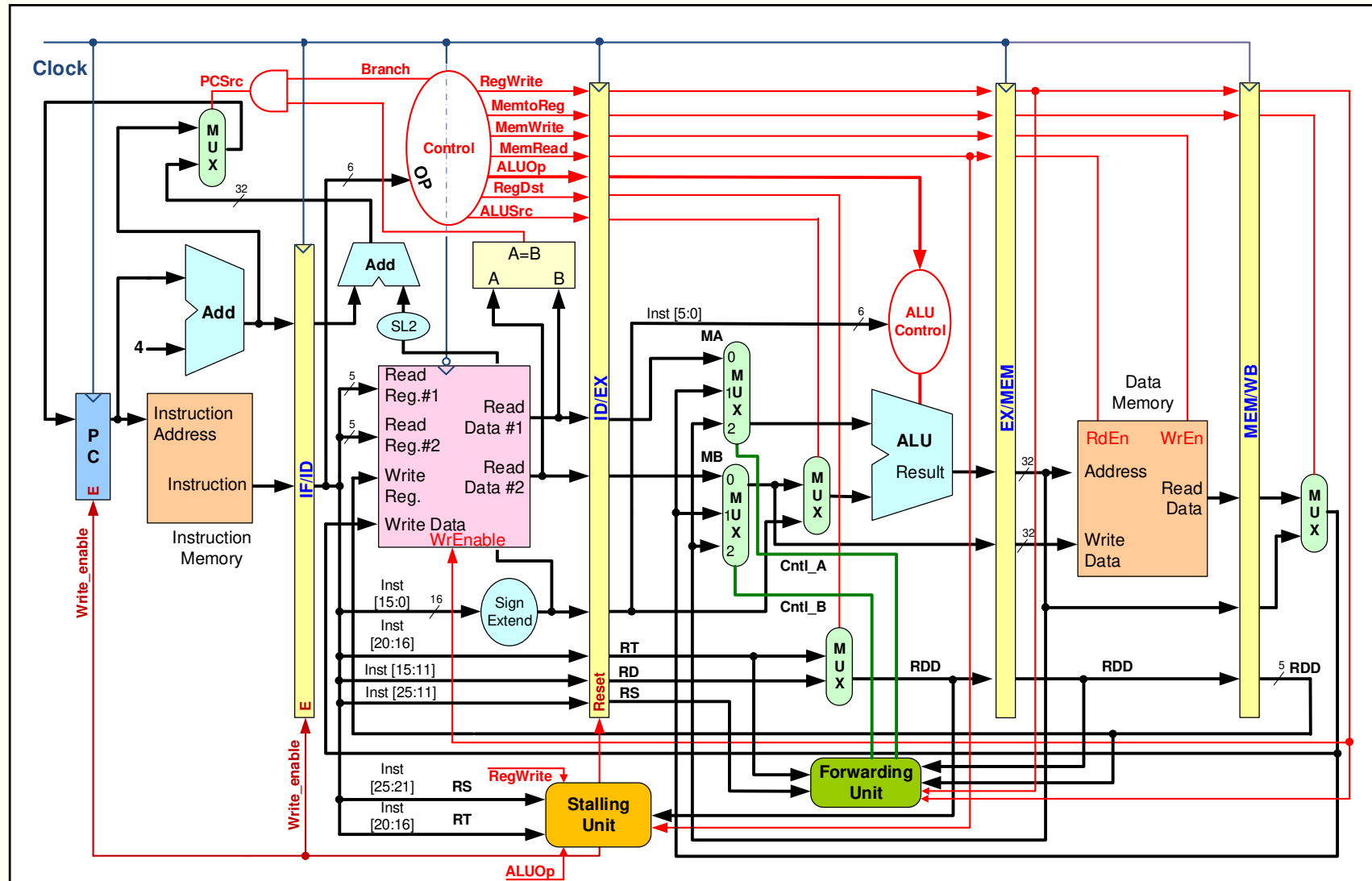
```
library ieee;
use ieee.std_logic_1164.all;

entity StallingUnit is
    port ( RS          : in  std_logic_vector(4 downto 0);
          RT          : in  std_logic_vector(4 downto 0);
          RegWrite     : in  std_logic;
          ALUOp        : in  std_logic_vector(1 downto 0);
          Ex_RDD       : in  std_logic_vector(4 downto 0);
          IdEx_MemRead : in  std_logic;
          Reset_IdEx   : out std_logic;
          Enable_IfId  : out std_logic;
          Enable_PC     : out std_logic);
end StallingUnit;
```

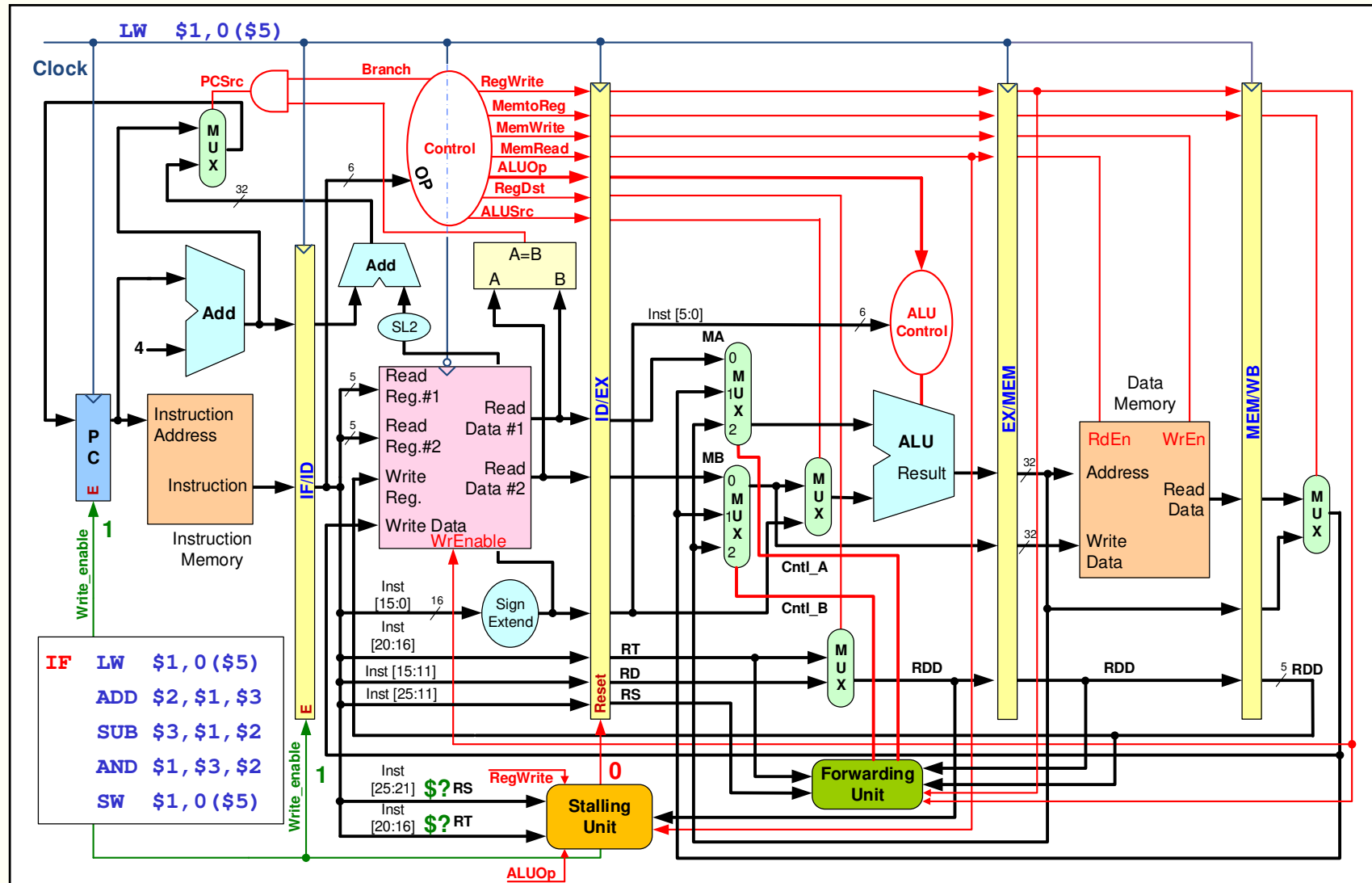
Unidade de controlo de *stalling* – VHDL

```
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 Ex_RDD /= "00000") then
      if(RegWrite = '1' and ALUOp = "10") then
        if(Ex_RDD = RS or Ex_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;
```

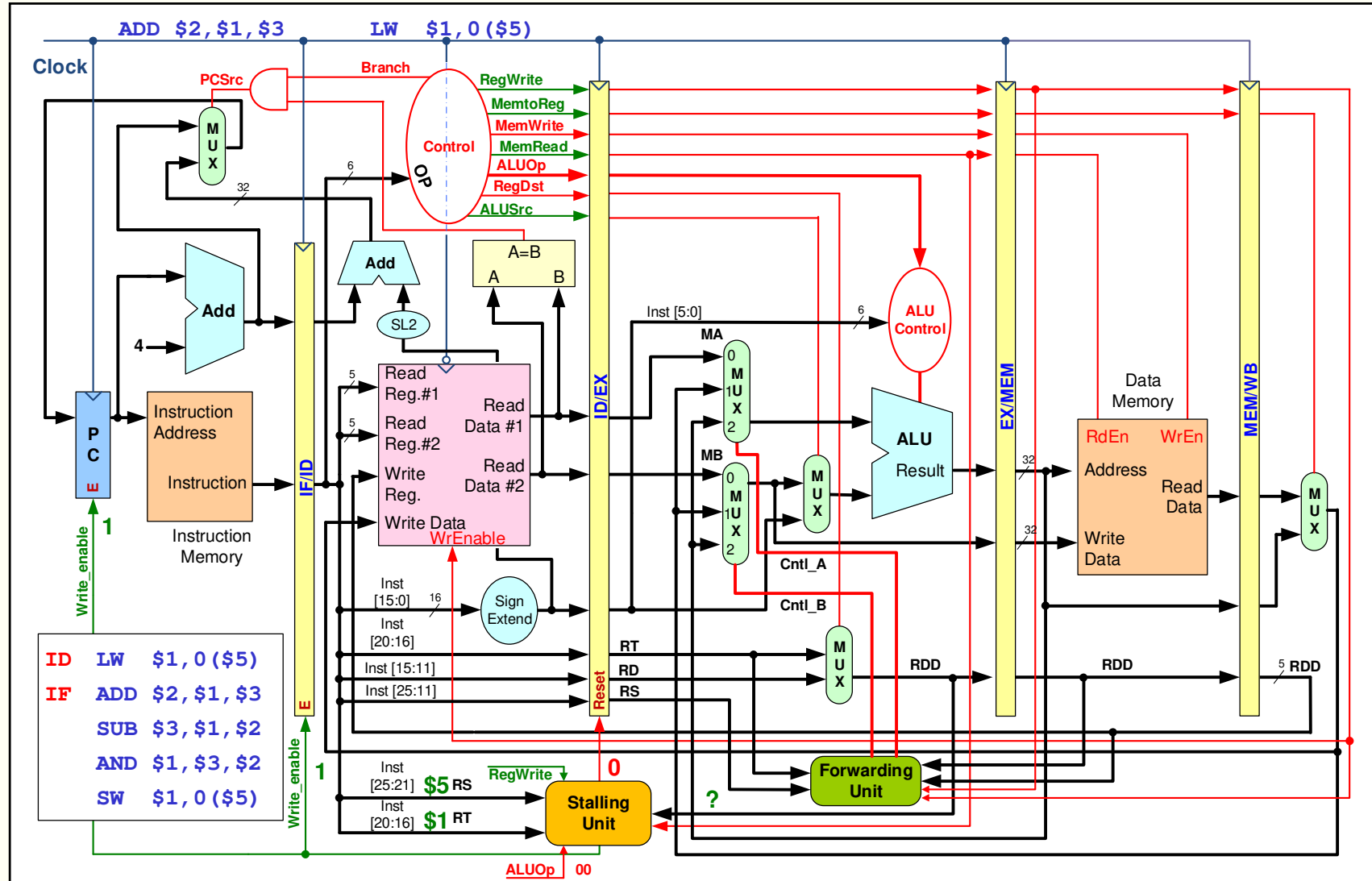
Datapath pipelining completo (com forwarding para EX, sem j)



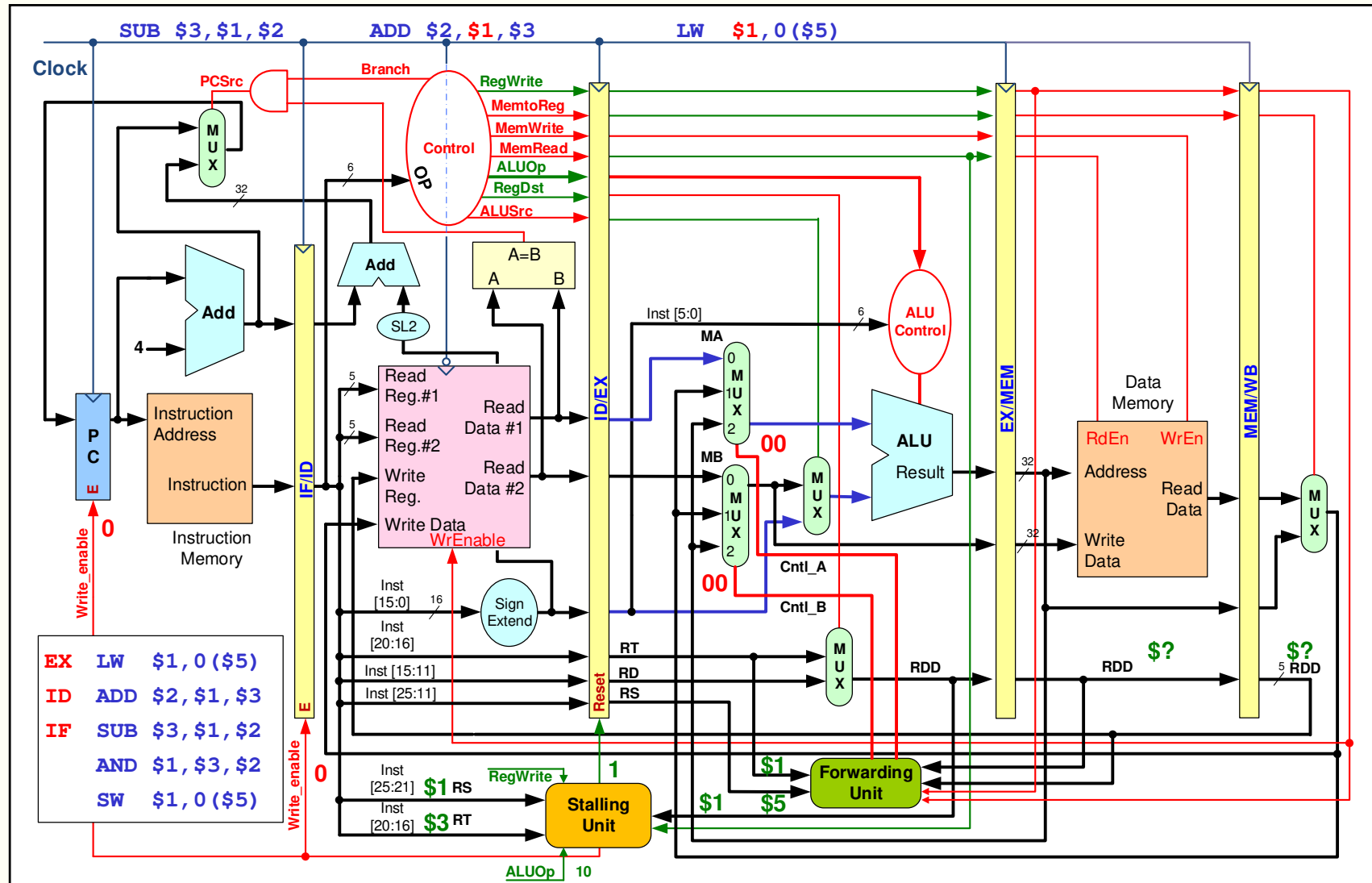
Datapath pipelining completo – exemplo de execução (1)



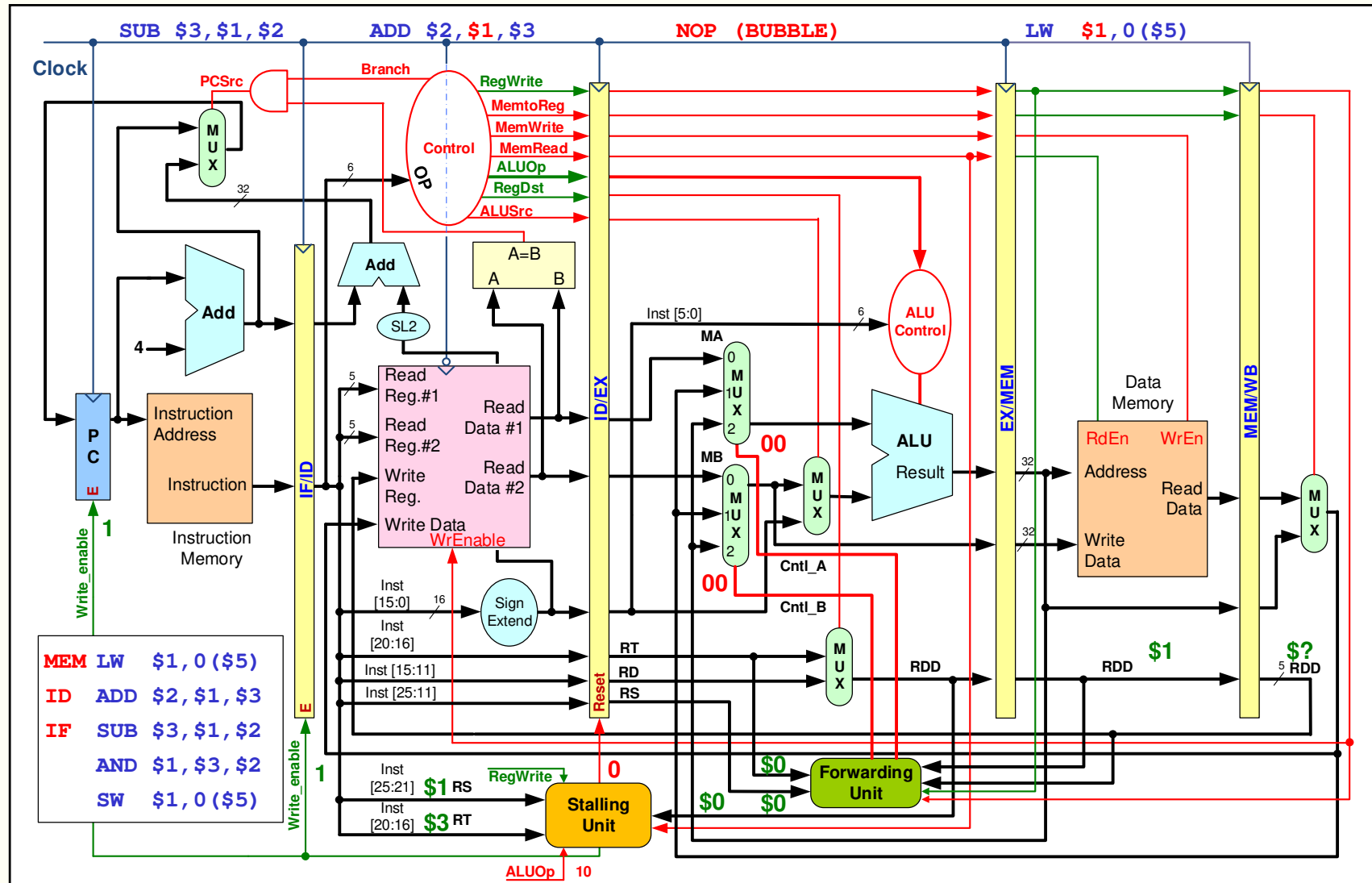
Exemplo de execução (2) (Normal Flow)



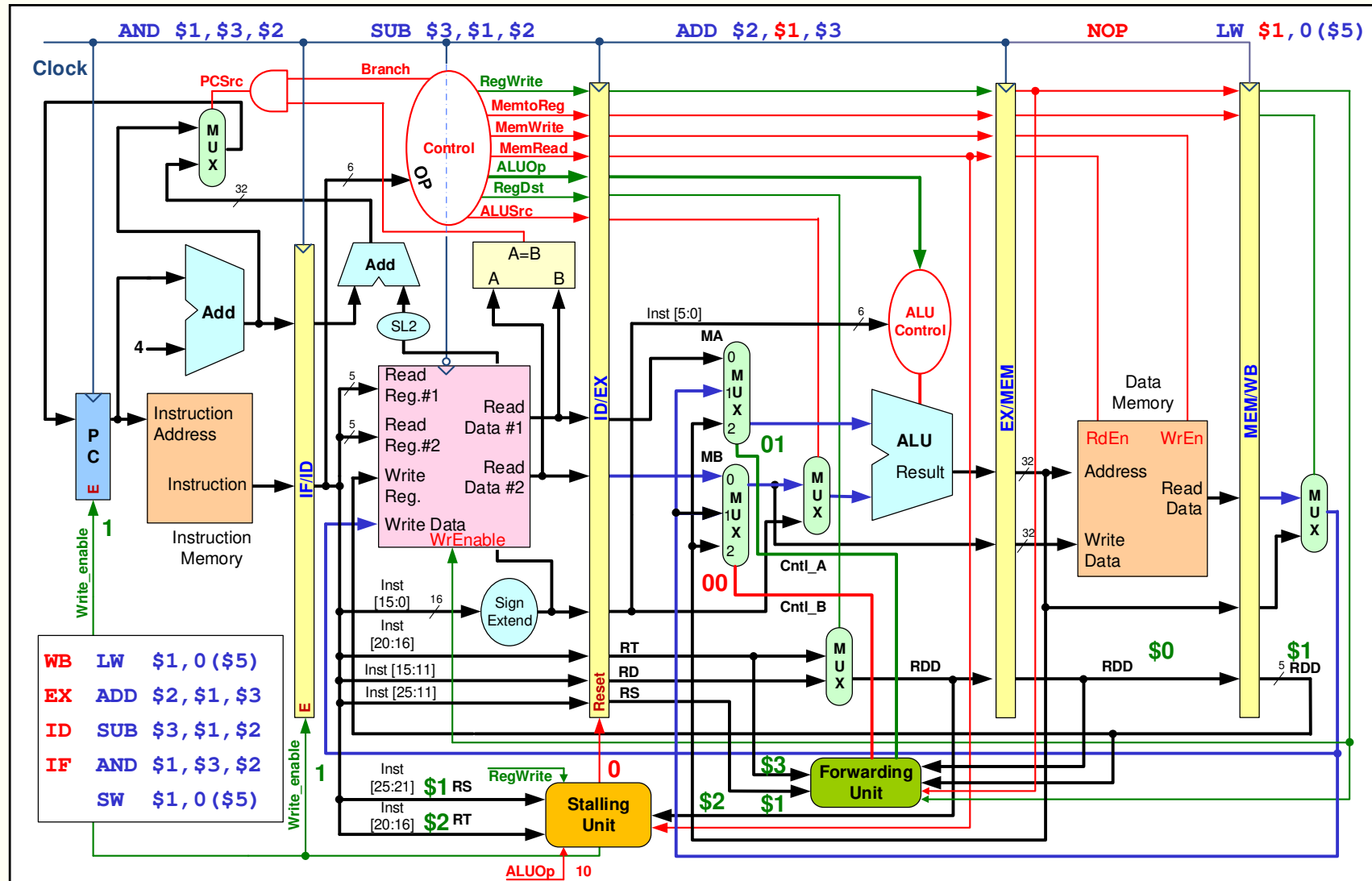
Exemplo de execução (3) (Normal Flow)



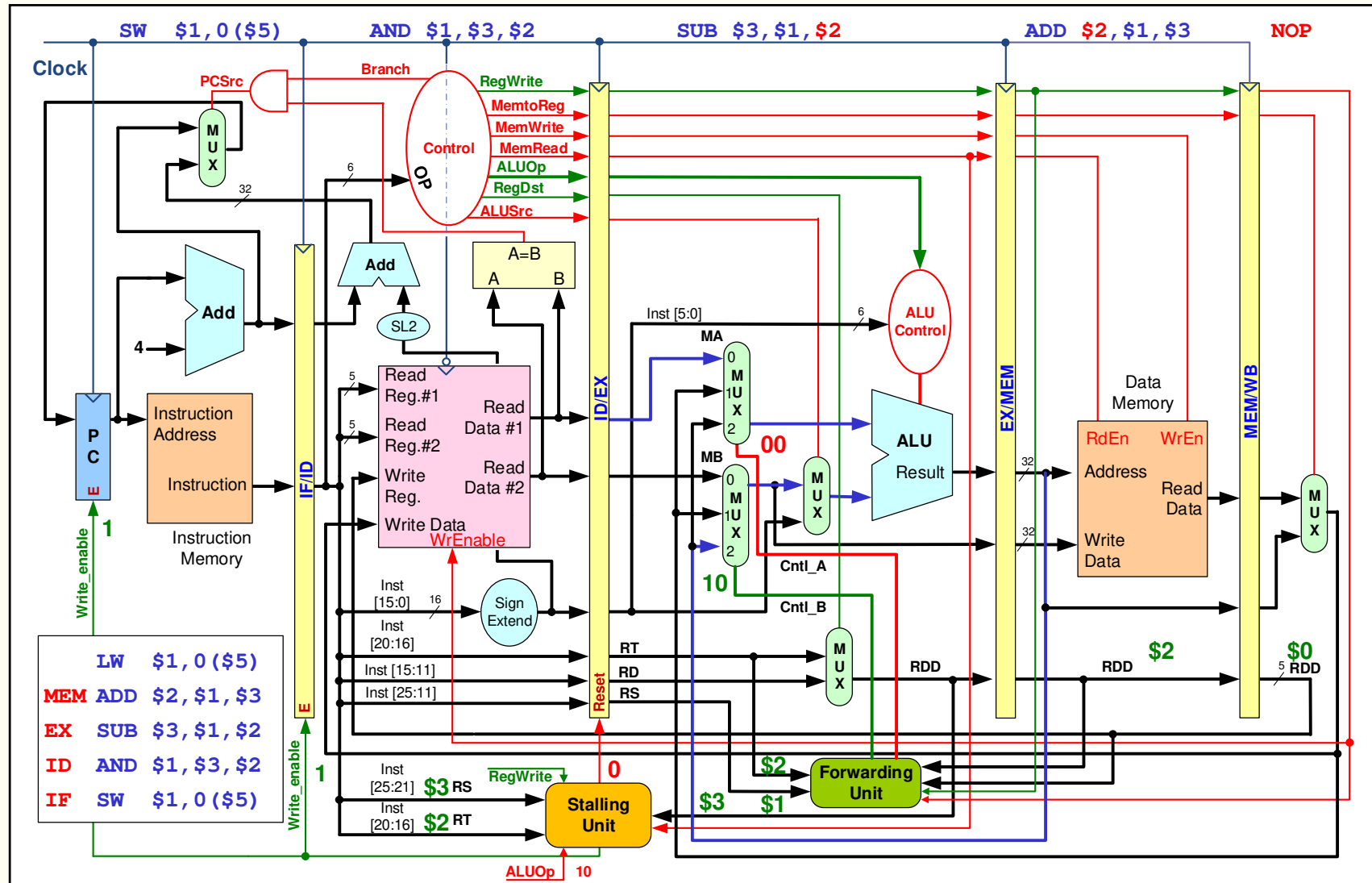
Exemplo de execução (4) (STALL)



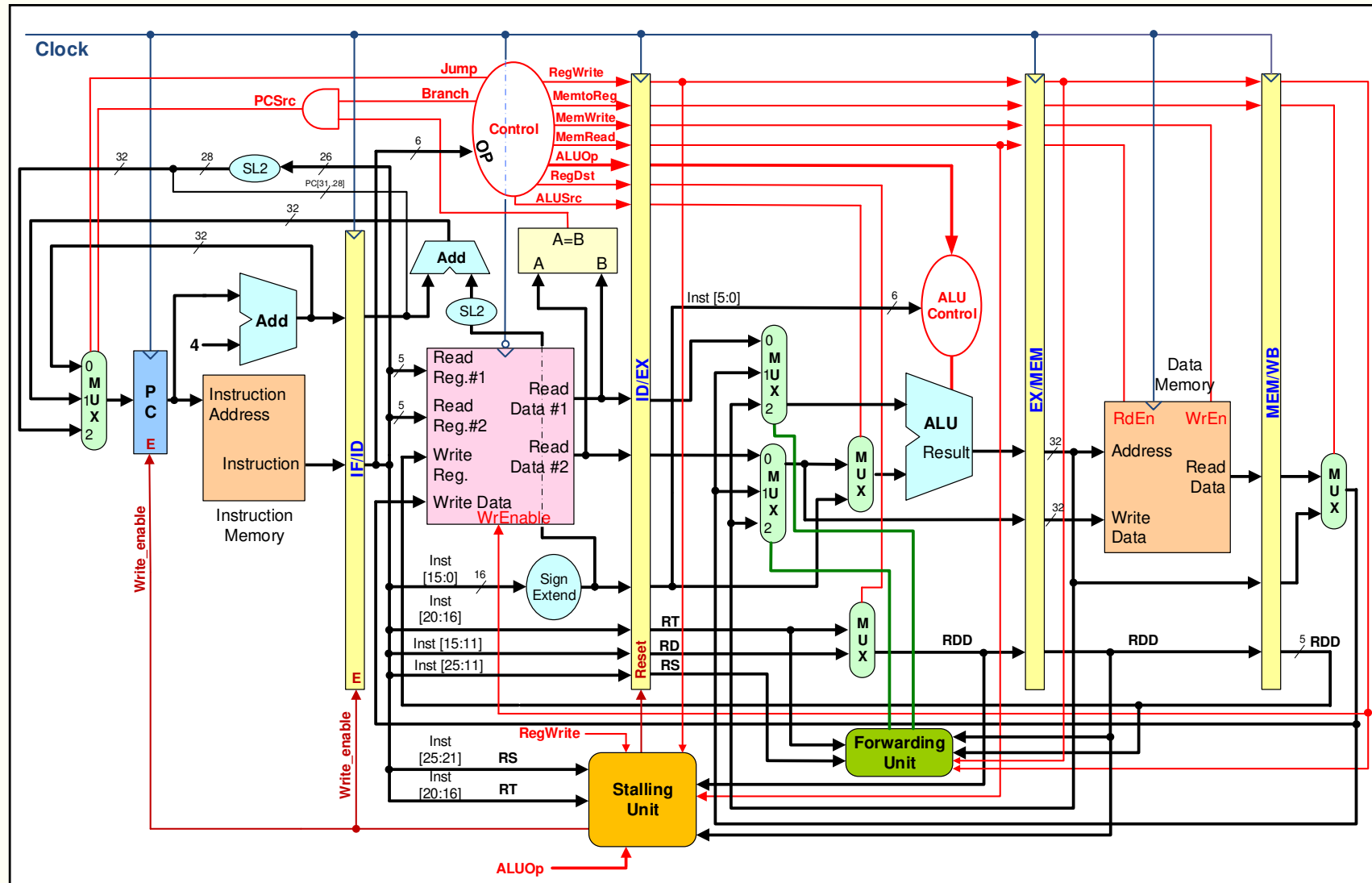
Exemplo de execução (5) (Fwd: MEM/WB > EX, rs)



Exemplo de execução (6) (Fwd: EX/MEM > EX, rt)



Datapath pipelining completo, com Jump



Stalling seguido de forwarding para EX

- Para além da sequência descrita anteriormente, há outras situações que também são resolvidas com *forwarding* para EX, e que obrigam a *stall* do *pipeline*. Exemplos:

lw	\$1, 0 (\$3)	#
sw	\$4, 8 (\$1)	# Stall 1T, FW MEM/WB > EX (RS)

lw	\$2, 0 (\$3)	#
lw	\$4, 8 (\$2)	# Stall 1T, FW MEM/WB > EX (RS)

lw	\$3, 0 (\$6)	#
addi	\$4, \$3, 0x12	# Stall 1T, FW MEM/WB > EX (RS)

- Se a arquitetura apenas implementar *forwarding* para EX:

lw	\$4, 0 (\$5)	#
sw	\$4, 4 (\$4)	# Stall 1T, FW MEM/WB > EX (RT)

Forwarding para ID (EX/MEM para ID)

- Os *hazards* de dados que ocorrem em instruções de *branch* determinam que a arquitetura deva também implementar *forwarding* para ID
- Exemplos:

(MEM)	add	\$1, \$2, \$3	#
(EX)	sub	\$2, \$4, \$6	#
(ID)	beq	\$1, \$5, lab	# FW EX/MEM > ID (RS)

(MEM)	addi	\$1, \$3, 0x25	#
(EX)	sub	\$2, \$1, \$4,	# FW EX/MEM > EX (RS)
(ID)	beq	\$5, \$1, lab	# FW EX/MEM > ID (RT)

Stalling seguido de forwarding para ID

- Mesmo supondo que a arquitetura implementa *forwarding* para ID (**EX/MEM** para 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
```

Forwarding para MEM (MEM/WB para MEM)

- Uma dependência originada por uma sequência do tipo:

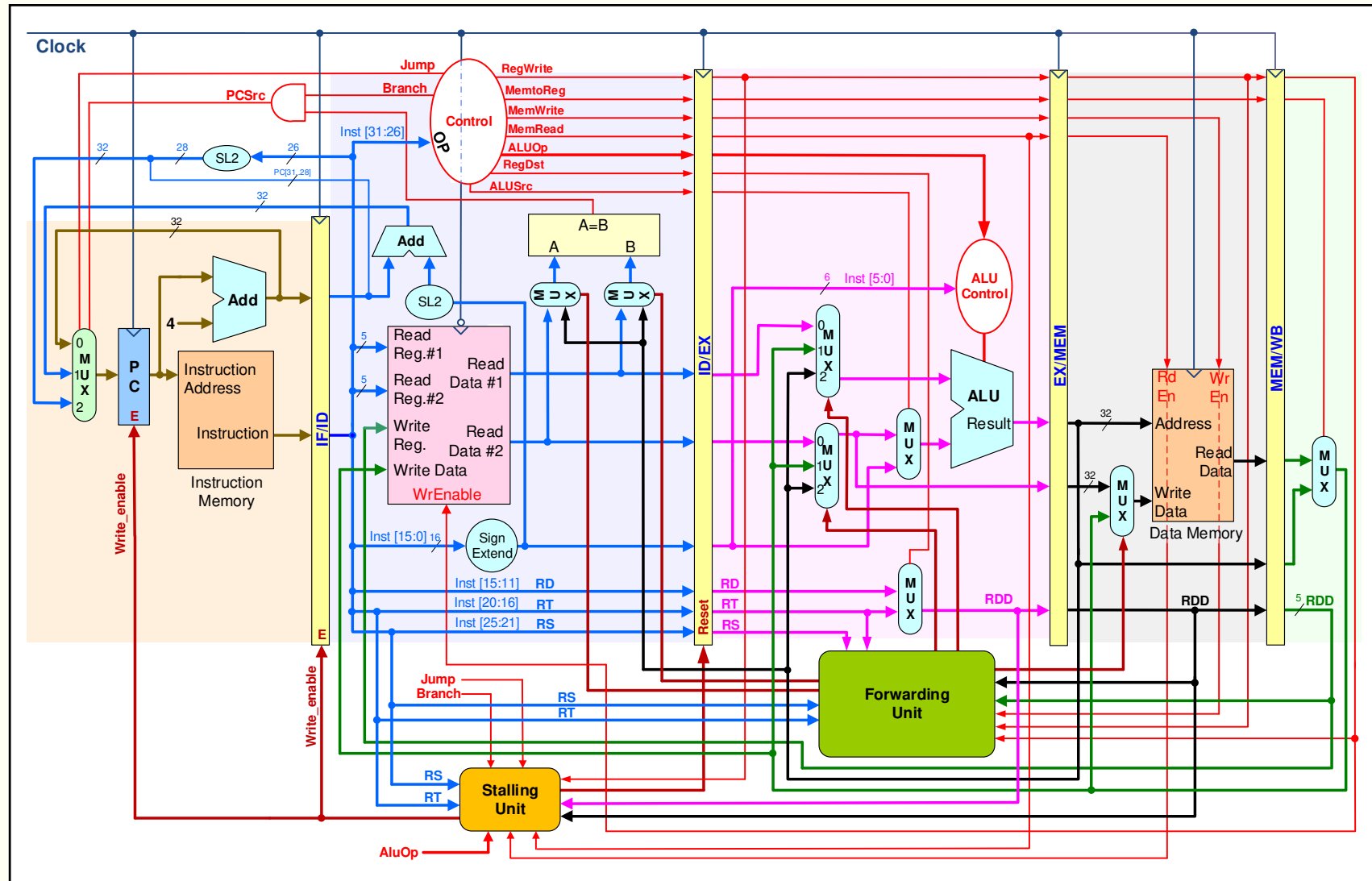
lw \$1, 0 (\$5) #

sw \$1, 4 (\$4) # Stall 1T, FW MEM/WB > EX (RT)

Pode ser resolvida com *stall* durante 1 ciclo de relógio seguido de *forwarding* de MEM/WB para EX (rt)

- 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 completo, com forwarding para MEM, EX 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 1ª 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)  
      addi    $1, $1, 4    #  
      slt     $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  
      addi    $1, $1, 4    #  
      slt     $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 1ª 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
      addi    $1, $1, 4    #
      slt     $5, $1, $2   # Stall 2T
      bne     $5, $0, loop # Stall 2T
      sw      $4, 8($0)    #
      lw      $1, 12($0)   #
  
```

Memória de dados

Addr	Value
0x00000000	0x10
0x00000004	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)
      addi    $1, $1, 4    #
      slt     $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 1ª 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)
      addi    $1, $1, 4    #
      slt     $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)   #

```

$$\begin{aligned}
 \text{Nr_cycles} &= F + (\text{Nr_instructions} - 1) + \text{Nr_Cycle_Stalls} \\
 &= 5 + (28 + 4 - 1) + 8 = 44 \text{ T} \quad (\text{nr of cycle stalls} = 4 * 2 = 8\text{T})
 \end{aligned}$$

Exercício 3

Relativamente ao programa da página seguinte a executar num *pipeline* de 5 fases, com *branches* resolvidos em ID e *delayed branch slot*:

1. Identifique todas as situações de *hazard* de dados e de controlo
2. Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o *pipeline* não implementa *forwarding*
3. Calcule o número de ciclos de relógio que o programa demora a executar (na situação da questão anterior), desde o IF da 1ª instrução até à conclusão da última instrução
4. Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o *pipeline* implementa *forwarding* para ID, EX e MEM
5. Calcule o número de ciclos de relógio que o programa demora a executar (na situação da questão anterior), desde o IF da 1ª instrução até à conclusão da última instrução

Para o mesmo programa, apresente os valores presentes nos registos \$1, \$2, \$3, \$4 e \$5 no final da execução do mesmo.

Exercício 3 (programa)

```
        .data                                # Segmento de dados: 0x00000000
A1:      .word  0x41, 0x43, 0x31, 0x2D, 0x32, 0x30, 0x31, 0x38
A2:      .space  48                          #
        .text                                #
        .globl  main                         #
main:    addi    $5, $0, 0                    #
        addi    $4, $0, 0x20                 #
        addi    $2, $0, 7                    #
        add     $2, $2, $2                    #
        add     $2, $2, $2                    #
        add     $2, $2, $4                    #
C1:      lw      $3, 0($5)                    #
        sw      $3, 0($2)                    #
        addi    $2, $2, -4                   #
        slt     $1, $2, $4                   #
        beq     $1, $0, C1                   #
        addi    $5, $5, 4                    #
C2:      addi    $0, $0, 0                    #
```