

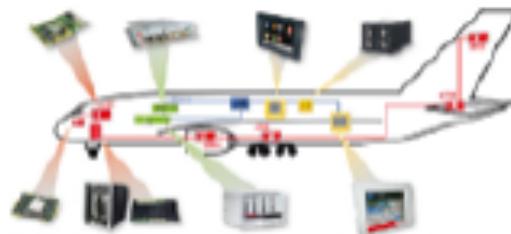
# Infraestrutura de HW

Introdução

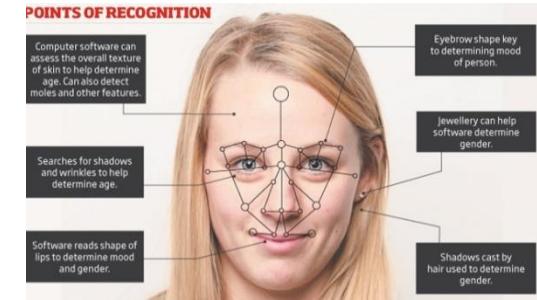
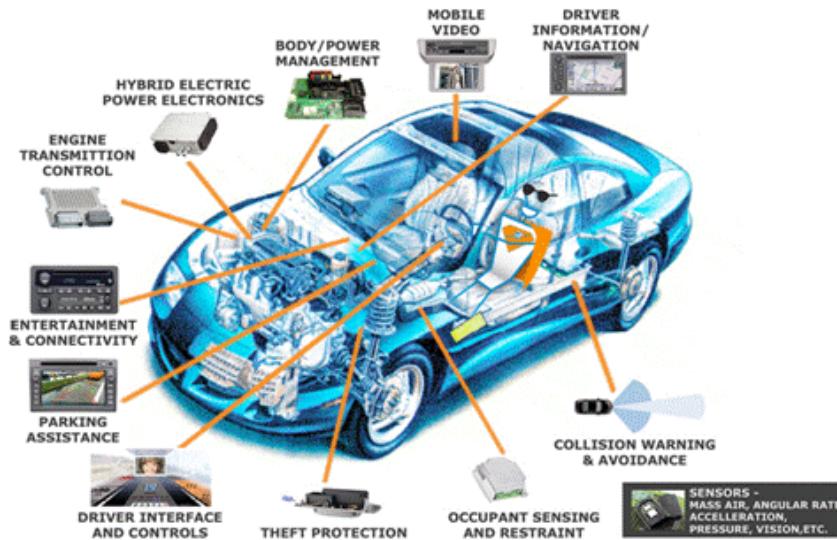
Prof. Adriano Sarmento

# Computadores no Mundo Atual

- Encontramos computadores em todo lugar!
  - Entretenimento, Transporte, Comunicação, Saúde, etc



# Demanda Crescente de Computadores



## Reconhecimento de Face

### Carro autônomo

- Aumento do número e complexidade das aplicações!

# Categorias de Computadores

- Desktops
- Servidores/Clusters
- Embarcados
- Dispositivos Móveis Pessoais (PMD)

# Desktops

- Impulsionou a popularização de computadores
- Computador pessoal que roda aplicativos genéricos
  - Exs: Editor de texto, browser, media player, jogos etc
- Alia bom desempenho a baixo custo
- Fati a importante do mercado de computadores
  - Impulsionou boa parte dos avanços tecnológicos dos últimos 40 anos



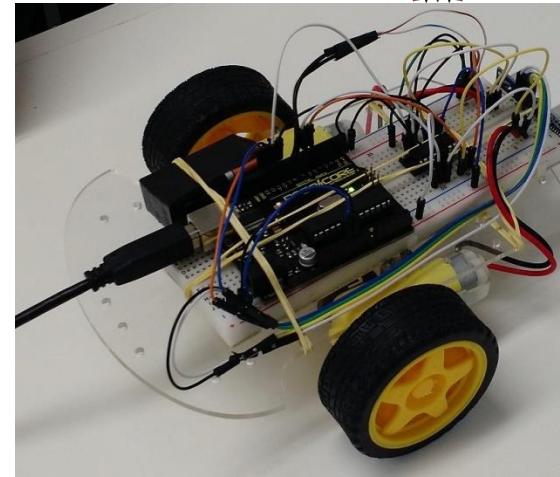
# Servidores/Clusters

- Roda aplicações complexas
  - Foco em disponibilidade, escalabilidade e throughput
- Usado para rodar aplicações que atendem muitos usuários simultaneamente
  - Exs: servidor web, sistema de gerenciamento de BD, “cloud computing”
- Acessados geralmente via rede
- Grande poder de processamento e armazenamento
  - Custo alto!



# Computadores Embarcados

- Estão em todo lugar!
  - Ex: carro, avião, televisão,
  - cameras digitais etc
- Rodam uma aplicação específica ou classe de aplicações relacionadas
  - Aplicações com forte integração com HW
- Aplicacões devem ser otimizadas para conseguir o máximo desempenho em um HW que deve ter custo e consumo de energia reduzido
- Devem ser robustos
  - Muito utilizados em sistemas críticos



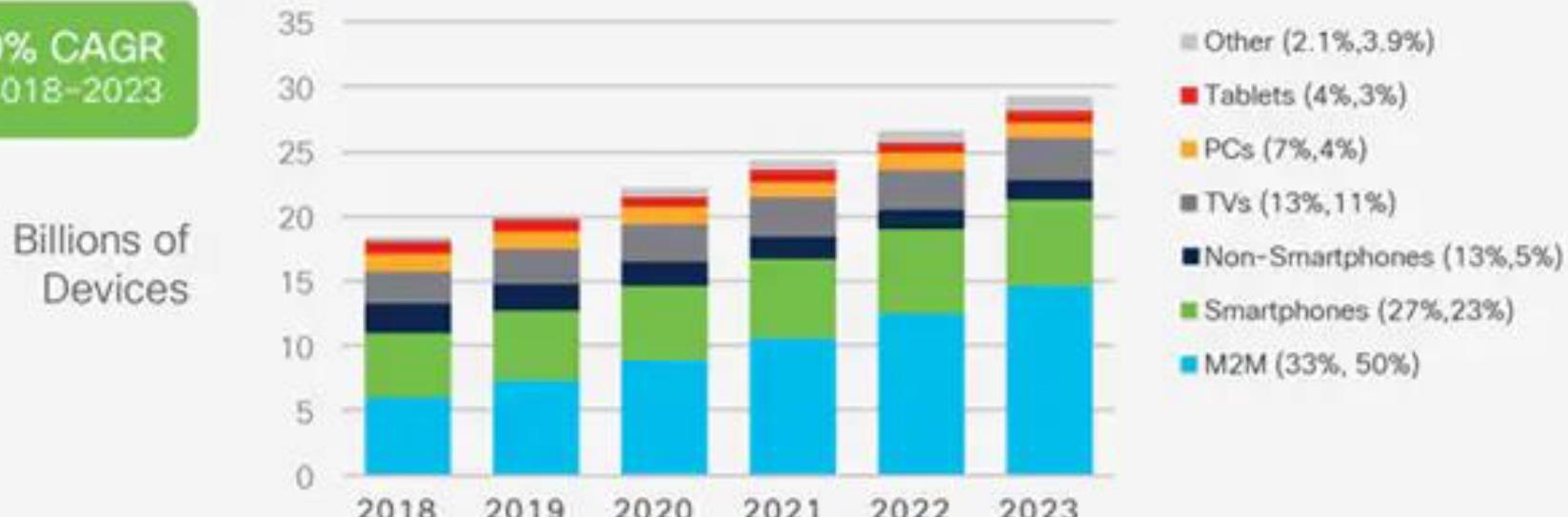
# Dispositivos Móveis Pessoais

- Dispositivos pequenos móveis capazes de executar diversos aplicativos
  - Ex: smartphone, tablets
- Têm como características marcantes:
  - Capacidade de comunicação com internet e outros dispositivos (wi-fi, bluetooth, GPS, etc)
  - Dependência de bateria
- Possuem características de desktops e embarcados
  - Variedade de dispositivos de E/S, capazes de rodar vários aplicativos diferentes
  - Restrições de memória e processamento, e otimiza consumo de energia



# Quantidade de Processadores Vendidos por Tipo

10% CAGR  
2018-2023



\* Figures (n) refer to 2018, 2023 device share

# Hardware e Software

- Computador = Hardware + Software

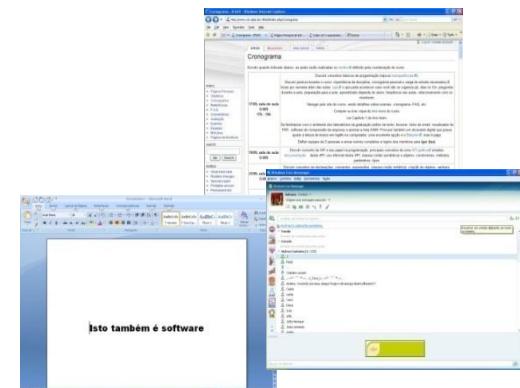
## ■ Hardware

- Parte física do computador
- Chips, monitores, teclado, etc



## ■ Software

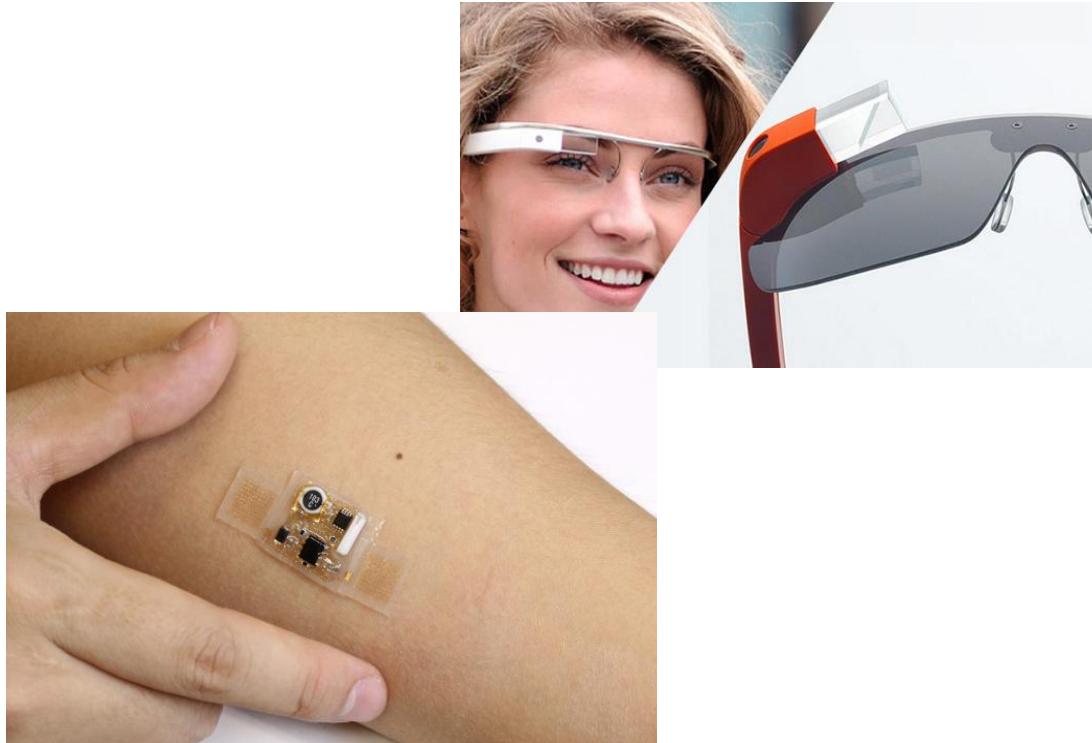
- Programas e dados
- Editores de texto, navegadores, sistemas operacionais, etc



# Por que Aprender Conceitos de Arquitetura e Organização de Computadores?

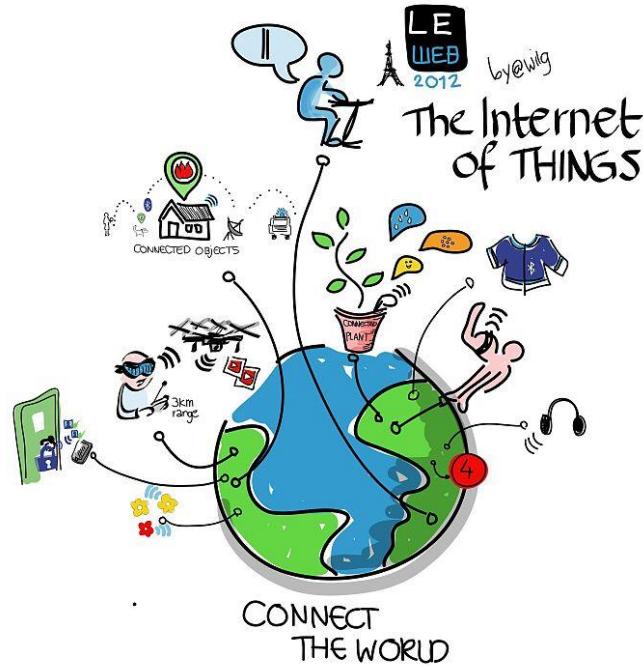
- Desempenho é um importante fator de qualidade para tornar software competitivo
- Desenvolver software com bom desempenho requer o entendimento de como um computador funciona
  - Componentes de um computador
  - Como os componentes interagem entre si
  - Como o software interage com os componentes

# Novas Tendências: Computadores Vestíveis (Wearable Computers)



- Computadores embarcados miniaturizados com poder de processamento e memória limitados que aderem ao corpo ou fazem parte da vestimenta

# Novas Tendências: Internet of Things (IoT)



- Conjunto de sensores e computadores embarcados com poder de processamento e memória limitados que estão conectados

# Por que Aprender Conceitos de Arquitetura e Organização de Computadores?



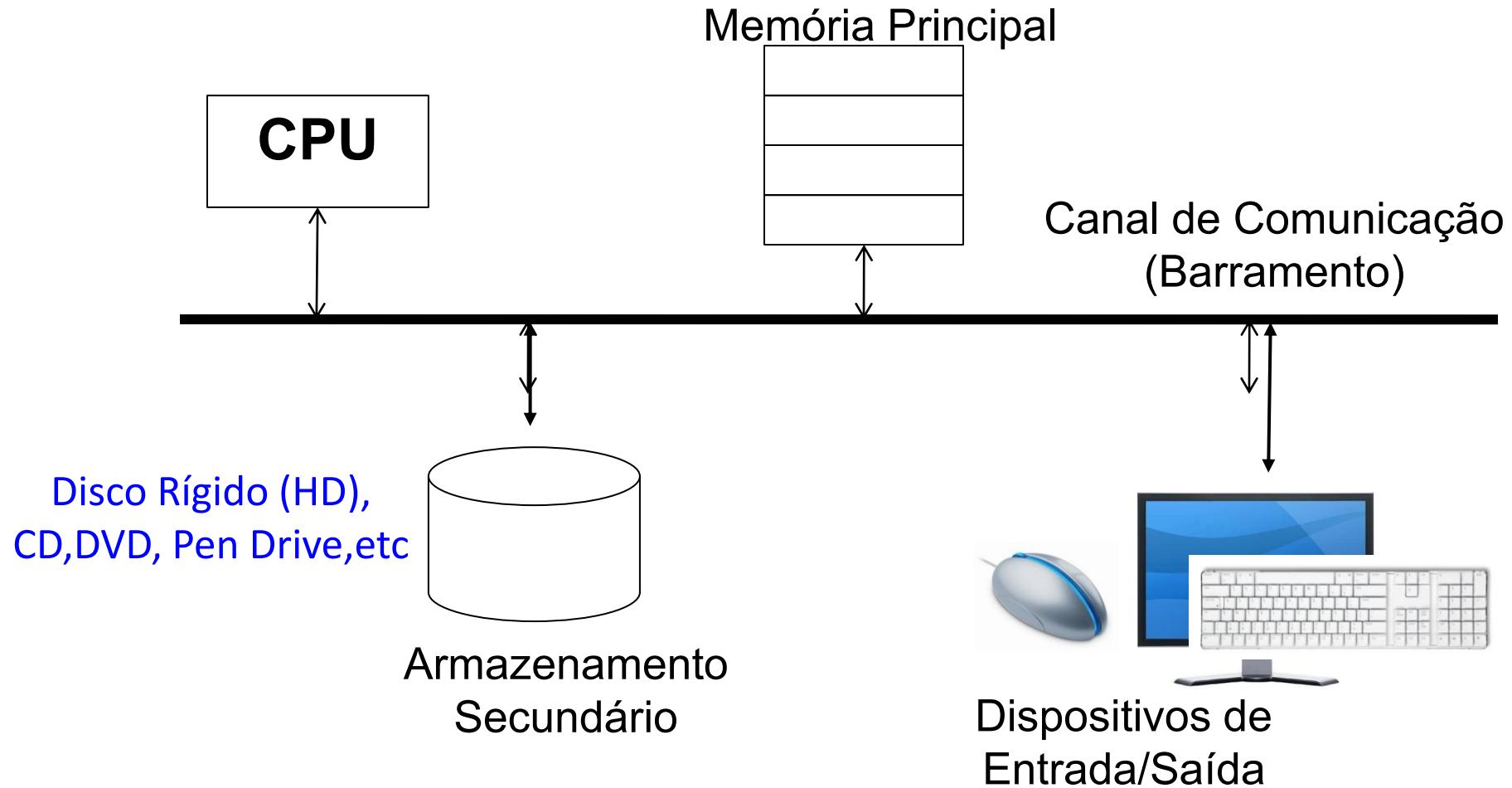
Aumento progressivo de venda de processadores para aplicações embarcadas

- Novas tendências exigirão o aumento de aplicações embarcadas
- Desenvolver aplicações embarcadas requerem bom conhecimento do HW

# Perguntas que Devem ser Respondidas ao Final do Curso

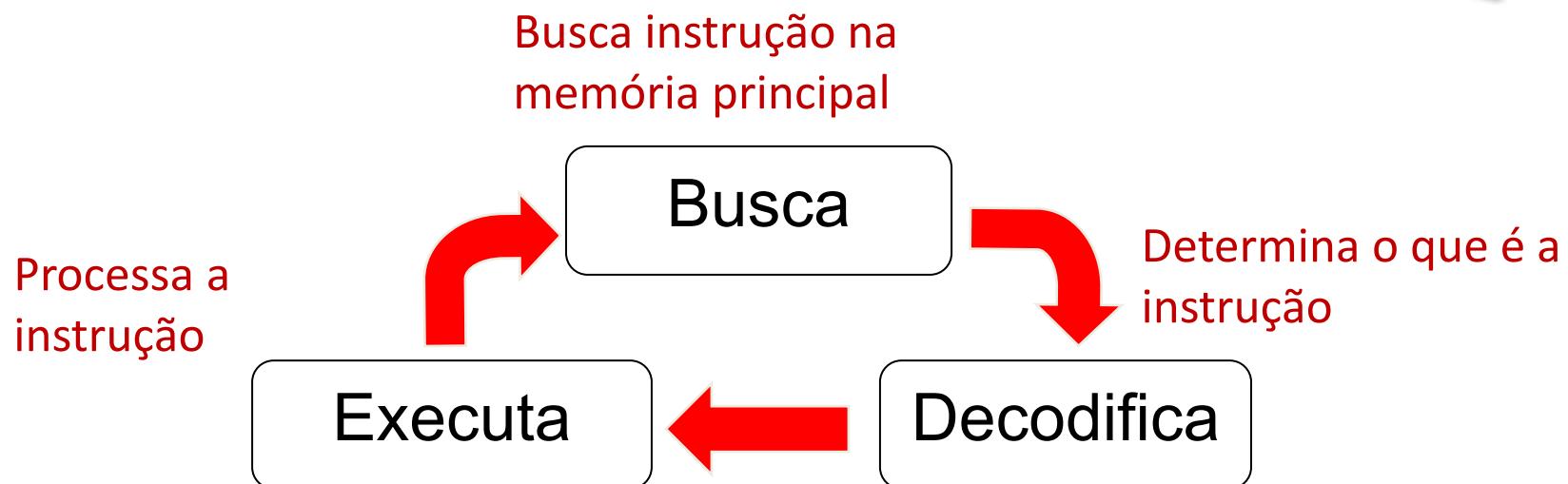
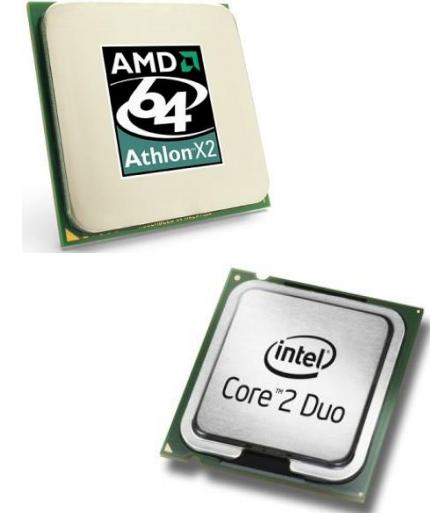
- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Modelo de um Computador

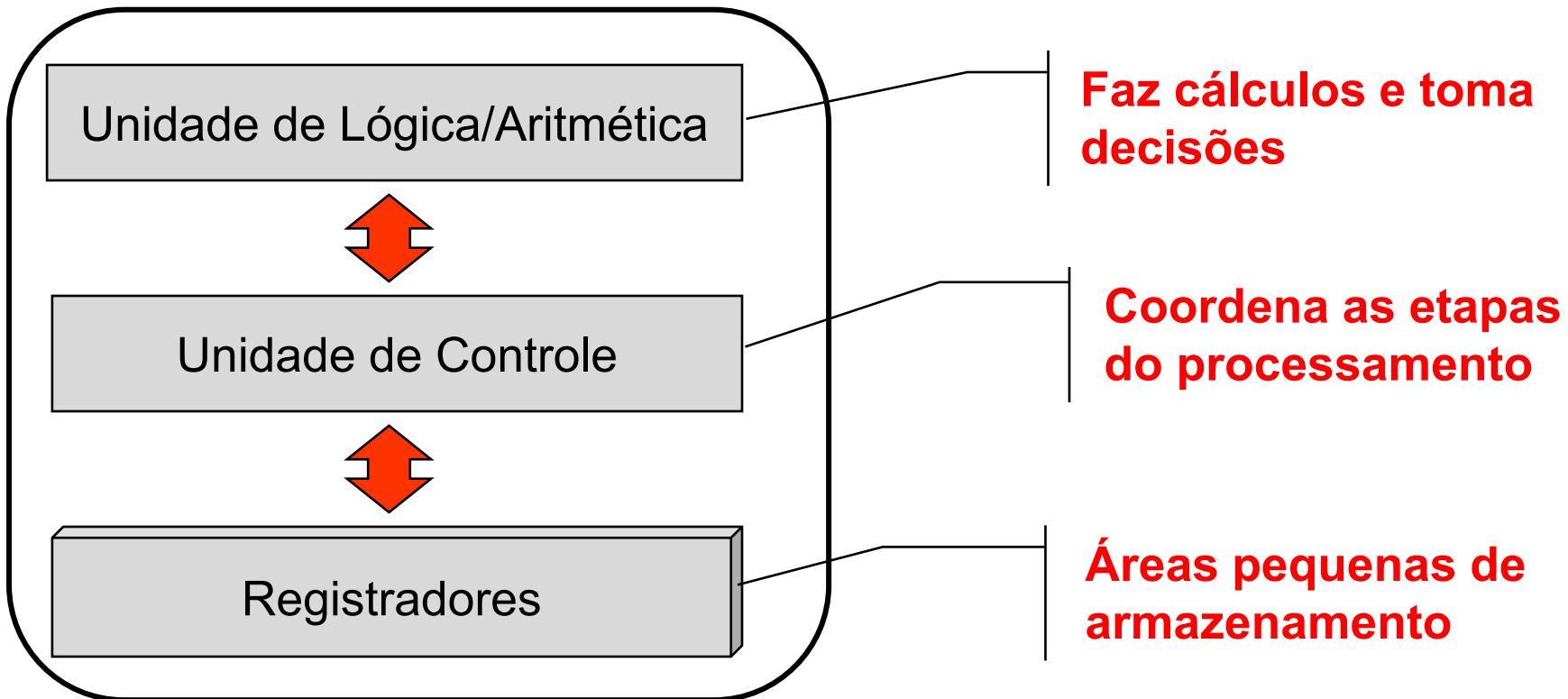


# Unidade Central de Processamento (CPU)

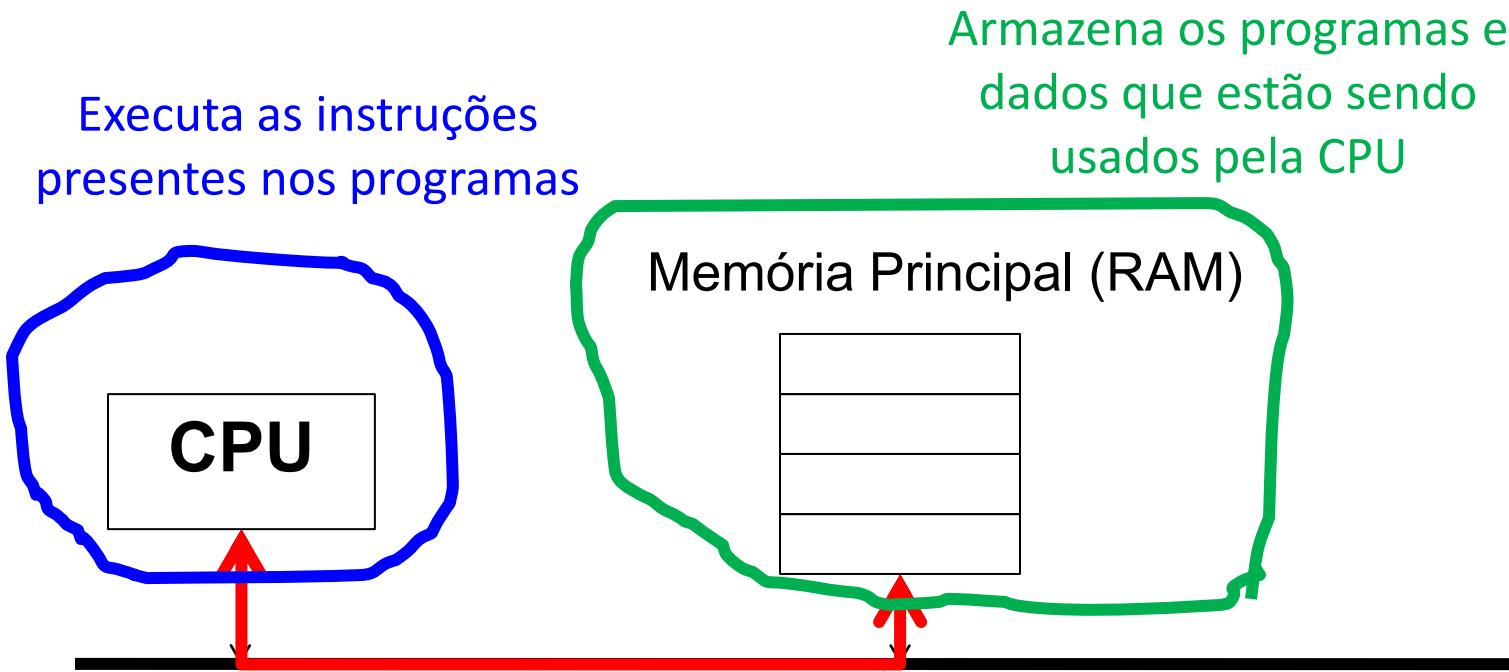
- A CPU é o “cérebro” do computador
- Implementado em um chip chamado de microprocessador
- Faz continuamente 3 ações:



# Componentes Principais de uma CPU



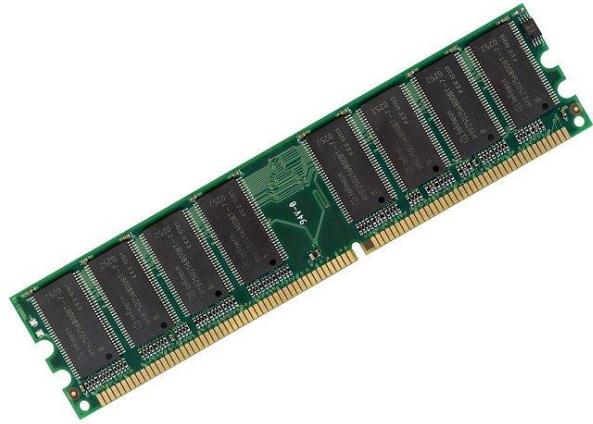
# CPU e Memória Principal



- CPU busca programas e dados residentes na memória
- CPU também armazena dados na memória

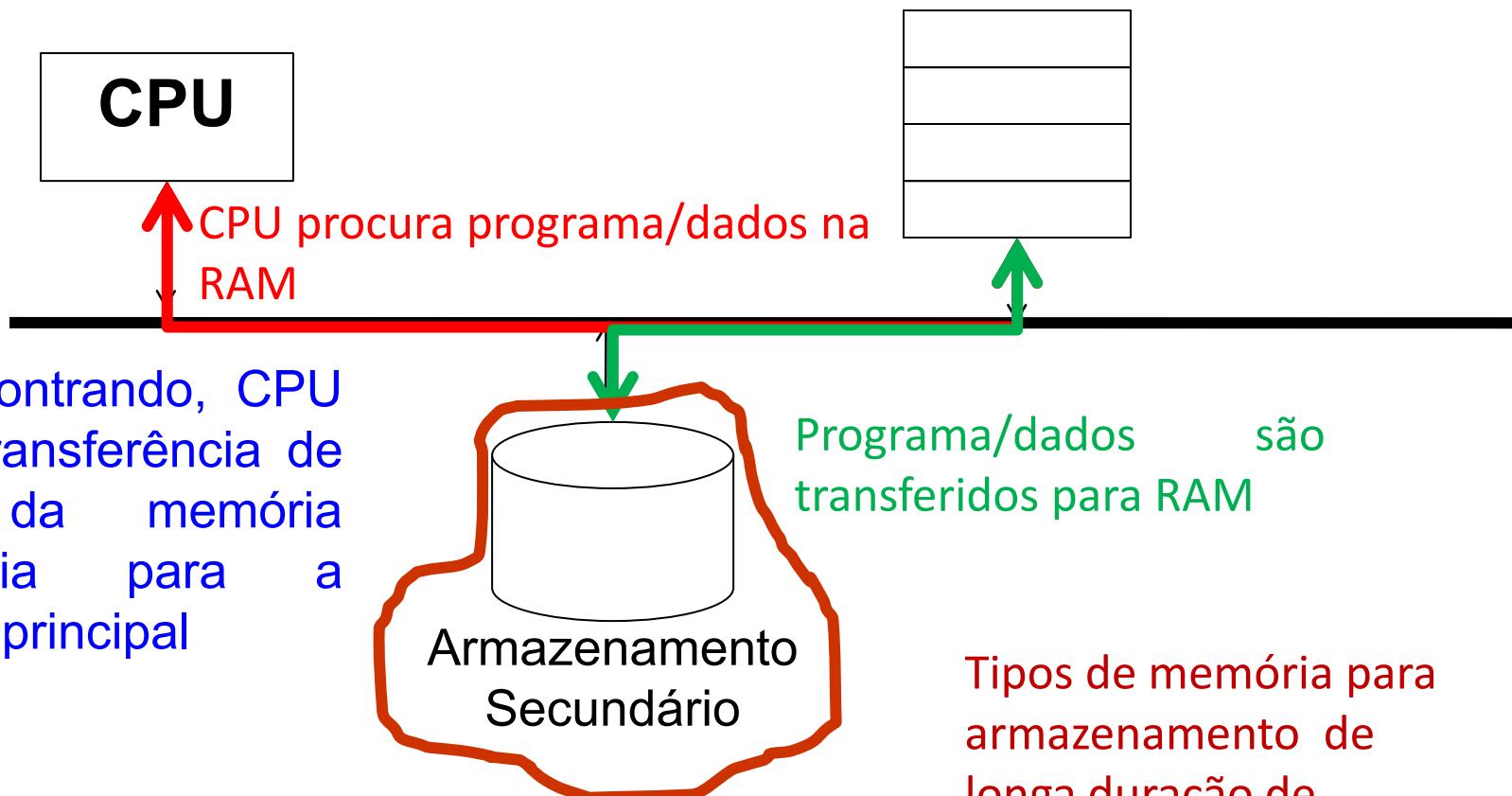
# Memória Principal

- Também chamada de memória RAM
  - Random Access Memory
    - Acesso aos endereços de memória podem ser feita de forma direta sem ter que passar por endereços anteriores
- Armazena dados e programas utilizados pelo processador num dado instante

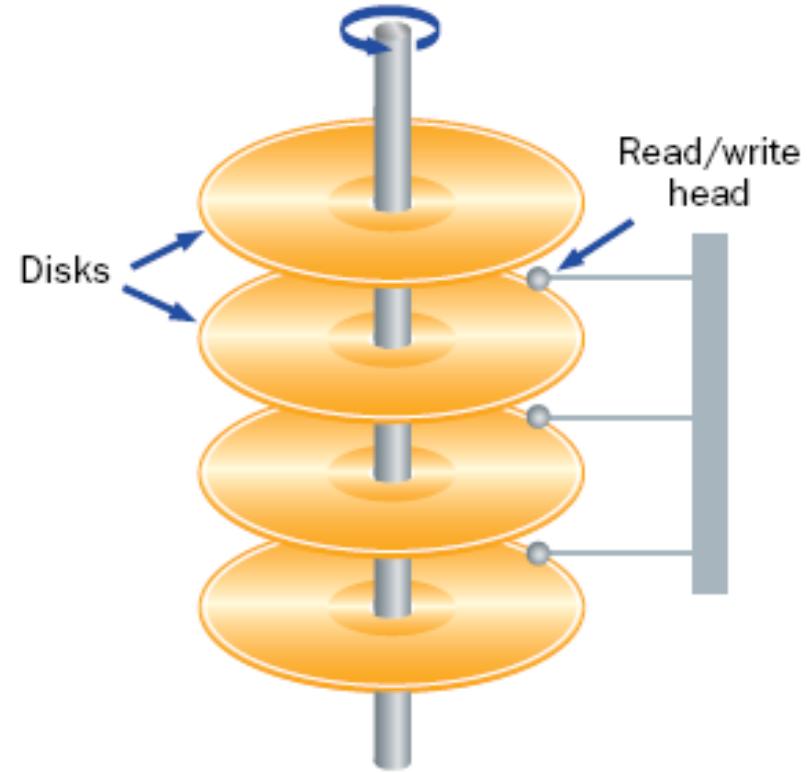


# Armazenamento Secundário

Memória Principal (RAM)



# Armazenamento Secundário (Disco Rígido)

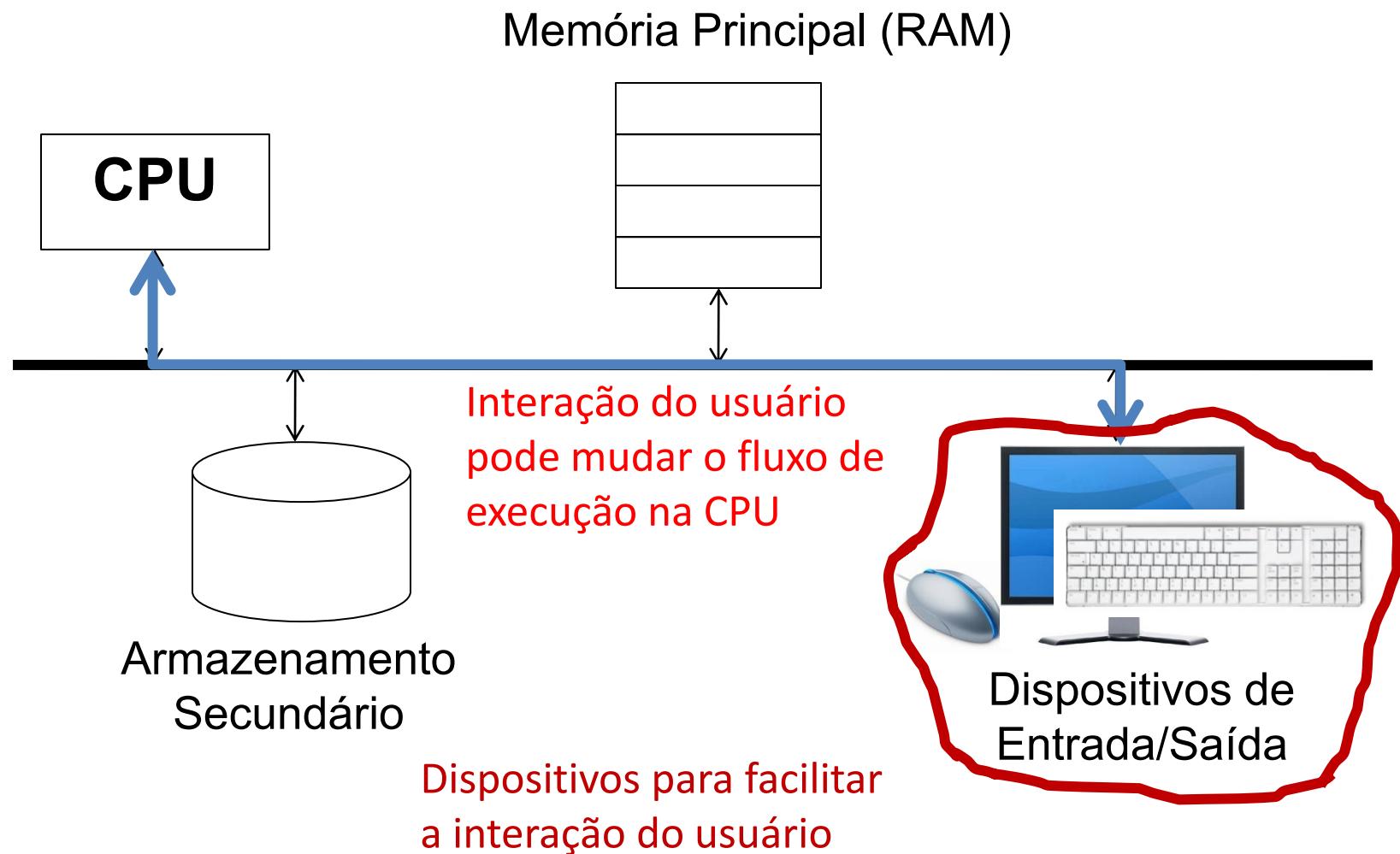


- Dispositivo magnético
- Partes que são gravadas são magnetizadas

# Memória Principal x Memória Secundária

- Memória RAM é mais rápida do que memórias secundárias
- Memória RAM é volátil
  - Informação é perdida quando não há corrente elétrica
- Memórias secundárias não são voláteis
- Memórias secundárias geralmente são mais baratas que a memória RAM
  - Por serem mais baratas, geralmente a capacidade de armazenamento é maior (Ex: Disco Rígido)

# Dispositivos de Entrada/Saída



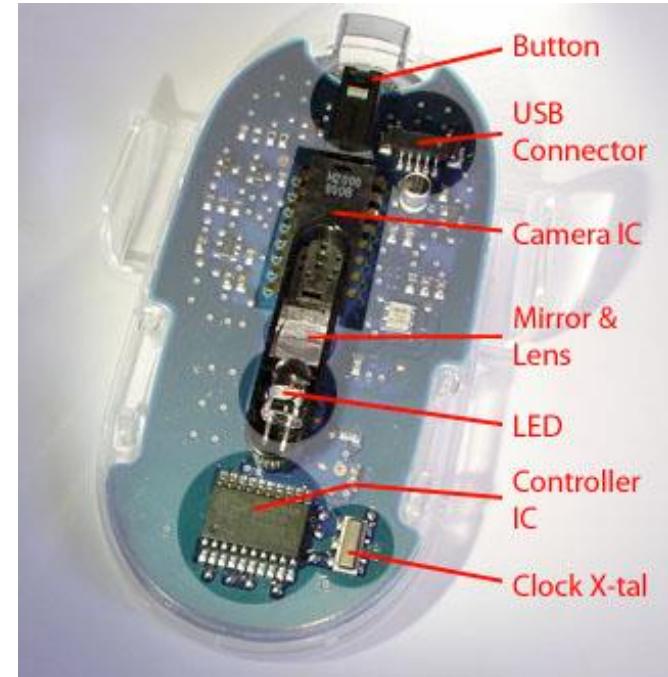
# Dispositivos de Entrada/Saída

- Teclado
- Mouse
- Leitor Óptico
- Joystick
- Monitor de vídeo
- Impressora

**Característica comum: Baixa Velocidade**

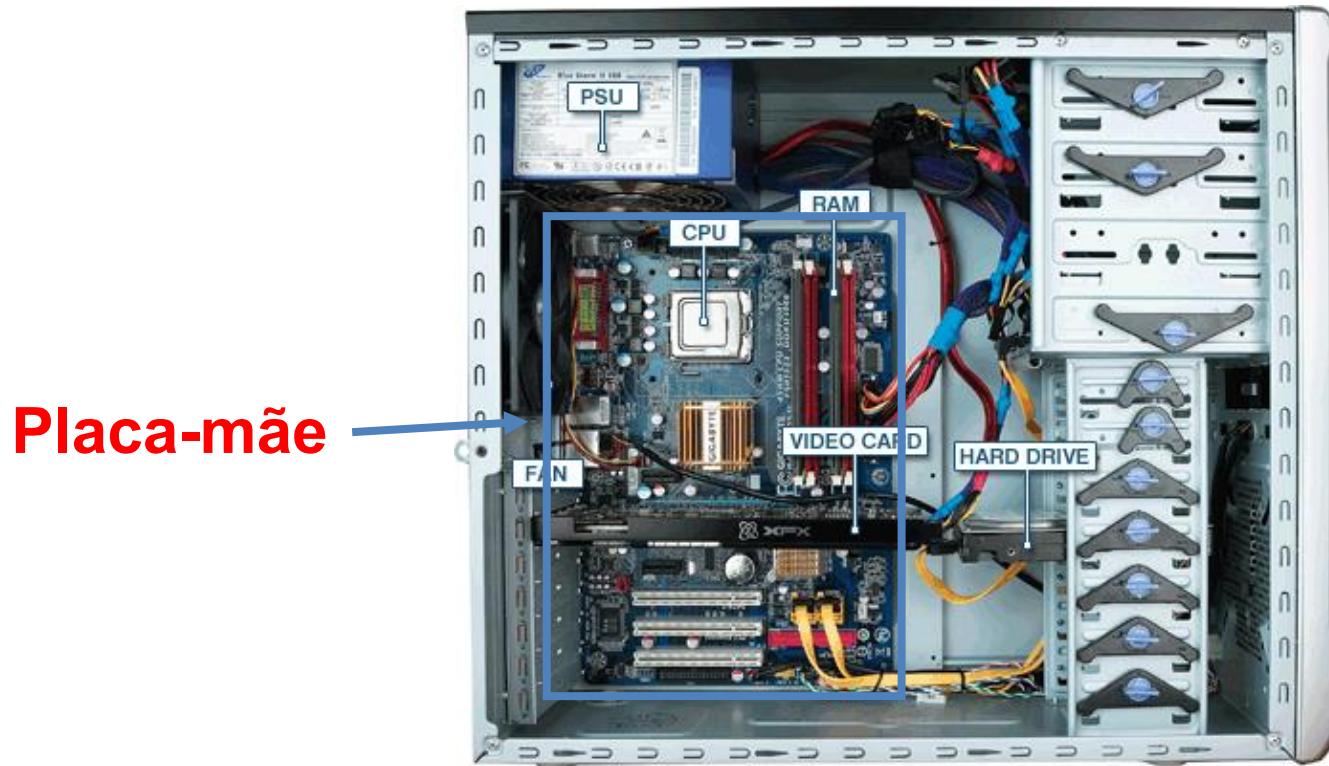
# Exemplo de Dispositivo de Entrada: Mouse Óptico

- Possui:
  - LED
  - Câmera preto e branco
  - Processador óptico (Controller IC)
- LED ilumina superfície, e câmera captura cerca de 1500 imagens por segundo e envia para processador óptico que calcula deslocamento



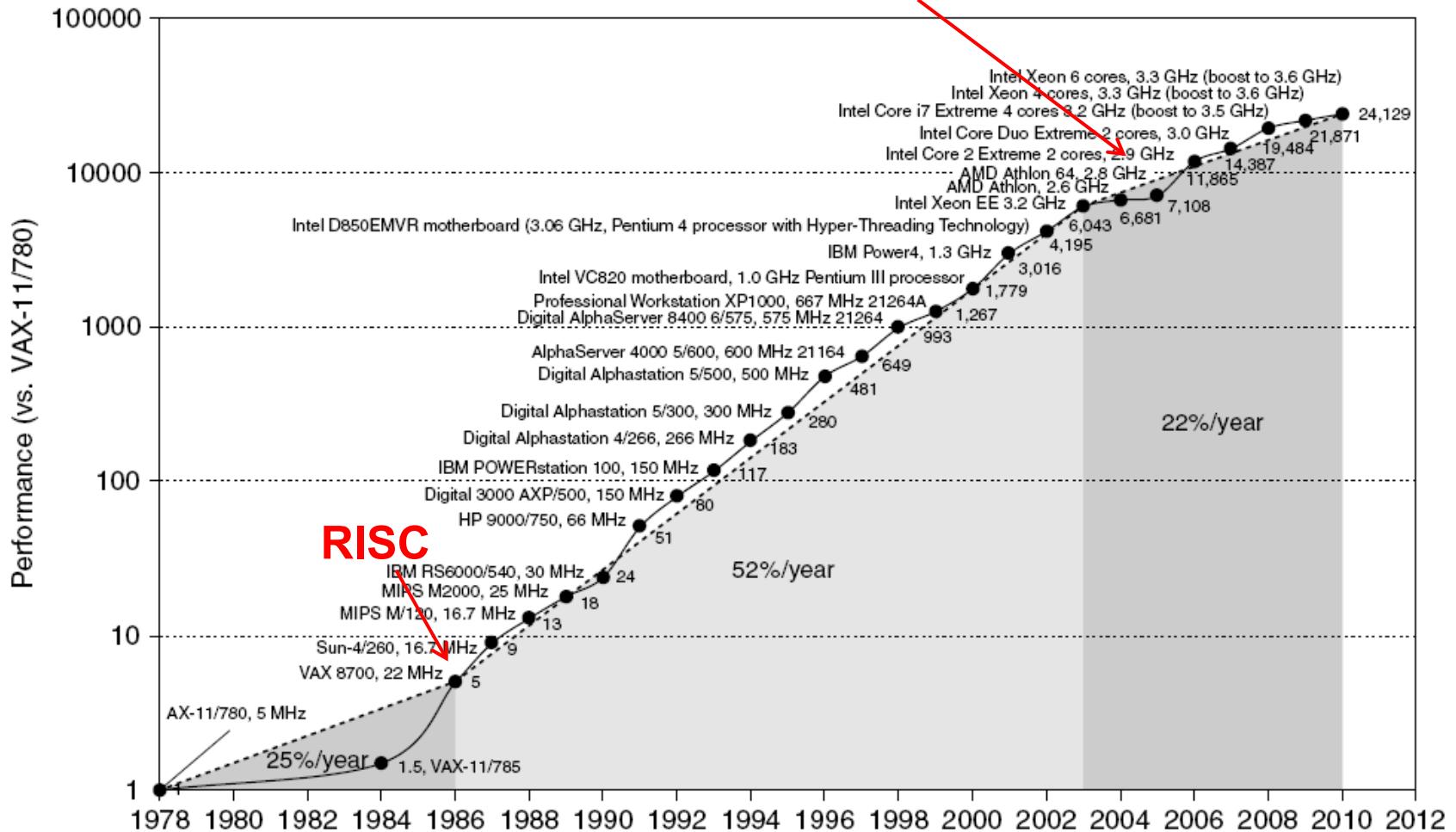
# Conectando Todos os Componentes de um Computador

- Placa-mãe é uma placa plástica dentro de um computador que contém chips, incluindo processador, caches, memória e conexões para dispositivos de E/S



# Avanços em Processamento

Mudança para multi-cores



# E Agora Para Onde Vamos?

- Projetistas de HW buscam maximizar desempenho e minimizar consumo de energia de processadores
  - Foco em dispositivos móveis

**Projetistas de SW devem desenvolver aplicações que maximizam uso eficiente das novas arquiteturas de HW**

# Programa

- Módulo 1:Conceitos Básicos de Arquitetura de Computadores
  - Introdução
  - Conceitos Básicos de Arquitetura
  - Usando o simulador MIPS
  - Implementação Mono-ciclo e Multi-ciclo

# Programa

- Módulo 2: Implementação em Pipeline e Superescalar, e Multiprocessadores
  - Implementação Pipeline
  - Resolução de Conflito de Dados e Controle
  - Implementação Superescalar
  - Multiprocessadores

# Programa

- Módulo 3: Hierarquia de Memória
  - Memória Cache
    - Tipos de Cache
    - Melhorando o desempenho de uma cache
  - Memória RAM
  - Memória Virtual

# Programa

- Módulo 4: Entrada/Saída
  - Entrada/Saída
    - Tipos de E/S
    - Componentes de um sistema de E/S

# Avaliação

- Provas, Projeto e Listas de Exercícios
  - Projeto e Listas feitas em grupo
- Nota Final =  $((\text{Projeto} + 0,1 \times \text{Lista1})) + ((\text{Prova1} + 0,1 \times \text{Lista2})) + ((\text{Prova2} + 0,1 \times \text{Lista3})) / 3$ 
  - **SE** a nota do projeto ou de qualquer uma das provas for 10,0, a bonificação extra da lista correspondente não é computada na nota final

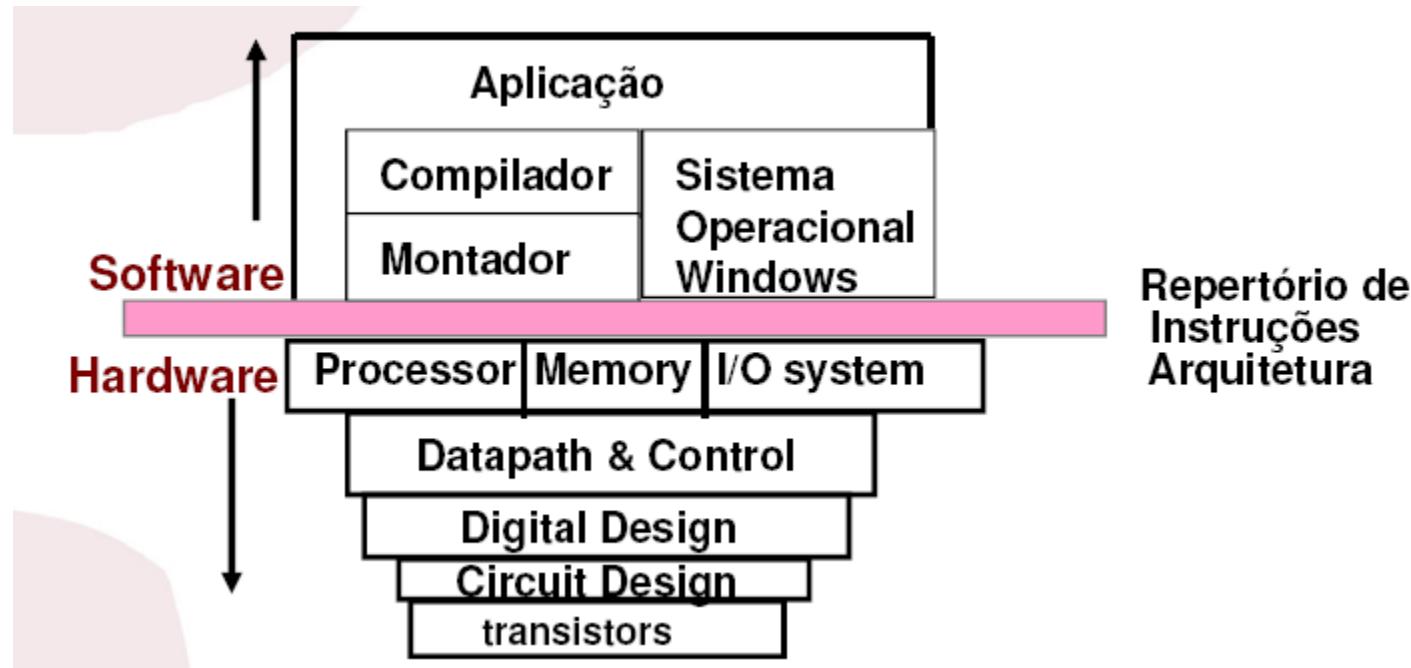
# Infraestrutura de Hardware

## Funcionamento de um computador

- Como um computador entende um programa

**Prof. Adriano Sarmento**

# Computador: Hardware + Software

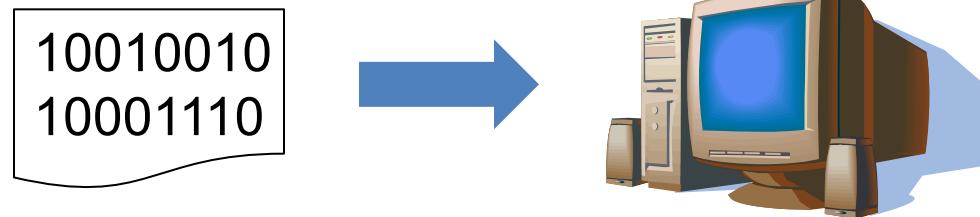


# Perguntas que Devem ser Respondidas ao Final do Curso

- **Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?**
- **Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?**
- **O que determina o desempenho de um programa e como ele pode ser melhorado?**
- **Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?**

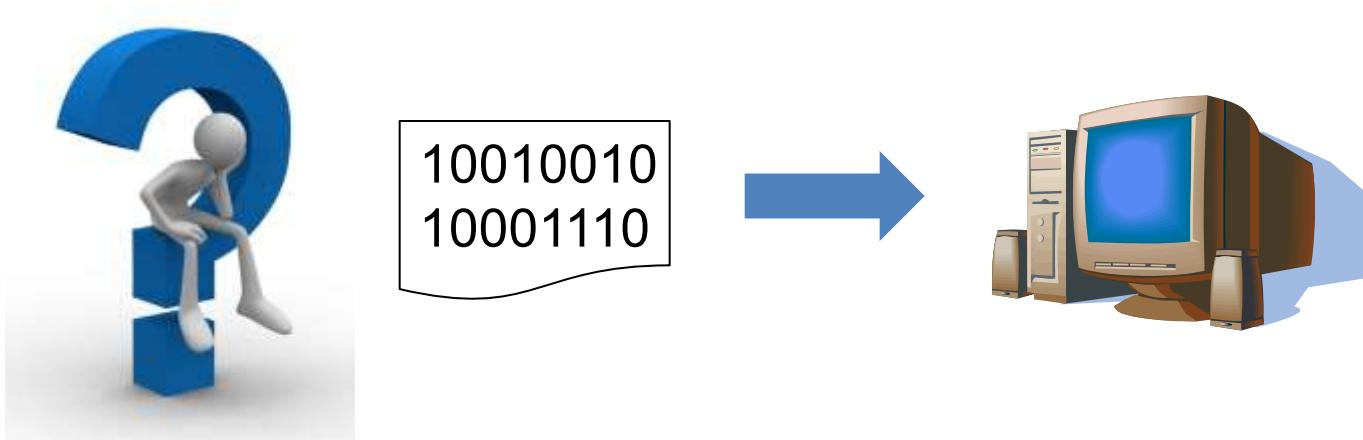
# Que Linguagem o HW entende?

- HW entende sinais elétricos
- Alfabeto da linguagem entendida por HW possui dois valores:
  - Desligado (Off) , Ligado (On),
  - Ou 0 e 1 (números binários)
- Instruções para o computador são sequências de números binários



# Abstraindo a Linguagem de Máquina

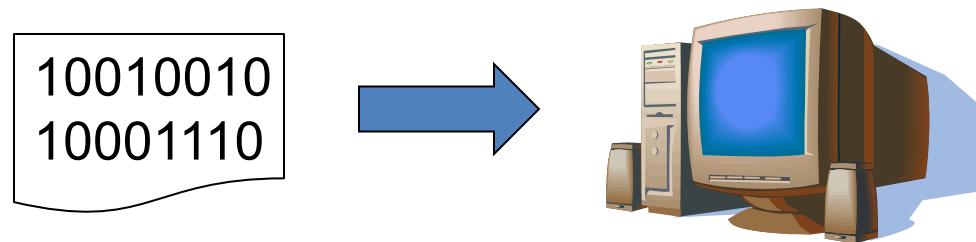
- Escrever um programa em linguagem de máquina é impraticável!



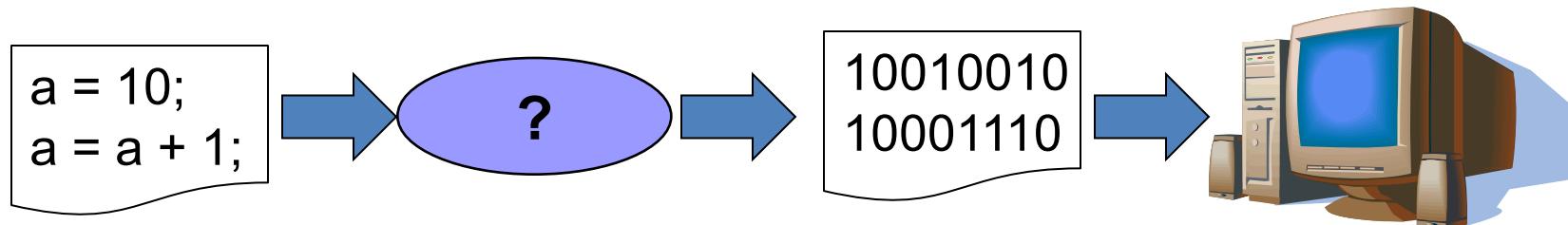
- Conceitos de HW foram abstraídos para que ser humano pudesse instruir o computador
- Criação de linguagens de programação

# Linguagens de Programação

- Os programas têm que ser escritos em uma linguagem de programação:
  - que possa ser entendida pelo computador



- que possa ser **traduzida** para a linguagem entendida pelo computador



# Níveis de Abstração de Linguagens

- Linguagens de programação variam de acordo com o seu nível de abstração
  - ↑ conhecimento da máquina onde programa será executado  
↓ nível de abstração
  - ↓ conhecimento da máquina onde programa será executado  
↑ nível de abstração
- Podem ser classificadas em 4 níveis:
  - Linguagem de máquina
  - Linguagem de montagem (assembly)
  - Linguagem de alto nível (Java, C, Pascal, C++, etc)
  - Linguagem de 4<sup>a</sup> geração (PL/SQL, IDL, MATLAB, etc)

# Níveis de Abstração de Linguagens

- Linguagem assembly é dependente da máquina, porém utiliza palavras reservadas para codificar instruções (mnemônicos)

Mnemônico

```
swap:  
    muli $2, $5,4  
    add  $2, $4,$2  
    lw   $15, 0($2)  
    lw   $16, 4($2)  
    sw   $16, 0($2)  
    sw   $15, 4($2)  
    jr   $31
```

- Outros níveis são independentes de máquina e facilitam leitura e escrita dos programas por parte do ser humano
  - Complexidade atual de programas exigem cada vez mais o emprego destas linguagens

# Como o Computador Entende um Programa?

- Deve-se traduzir um programa para a linguagem de máquina
- Um compilador é um programa que traduz um programa escrito (código fonte) em uma determinada linguagem de programação para outra linguagem (linguagem destino)
  - Se a linguagem destino for a de máquina, o programa pode, depois de compilado, ser executado
- Um interpretador é um programa que traduz instrução por instrução de um programa em linguagem de máquina e imediatamente executa a instrução

# Compilação x Interpretação

## Compilação

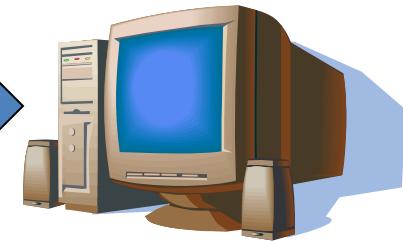
Código- fonte

```
a = 10;  
a = a + 1;
```

Compilador

Código de máquina

```
10010010  
10001110
```



## Interpretação

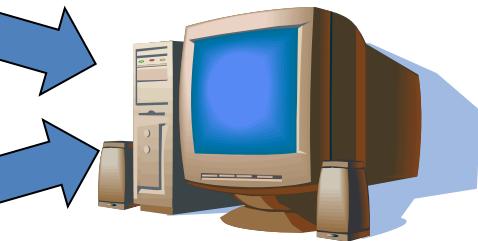
Código- fonte

```
a = 10;  
a = a + 1;
```

Interpretador

Código de máquina

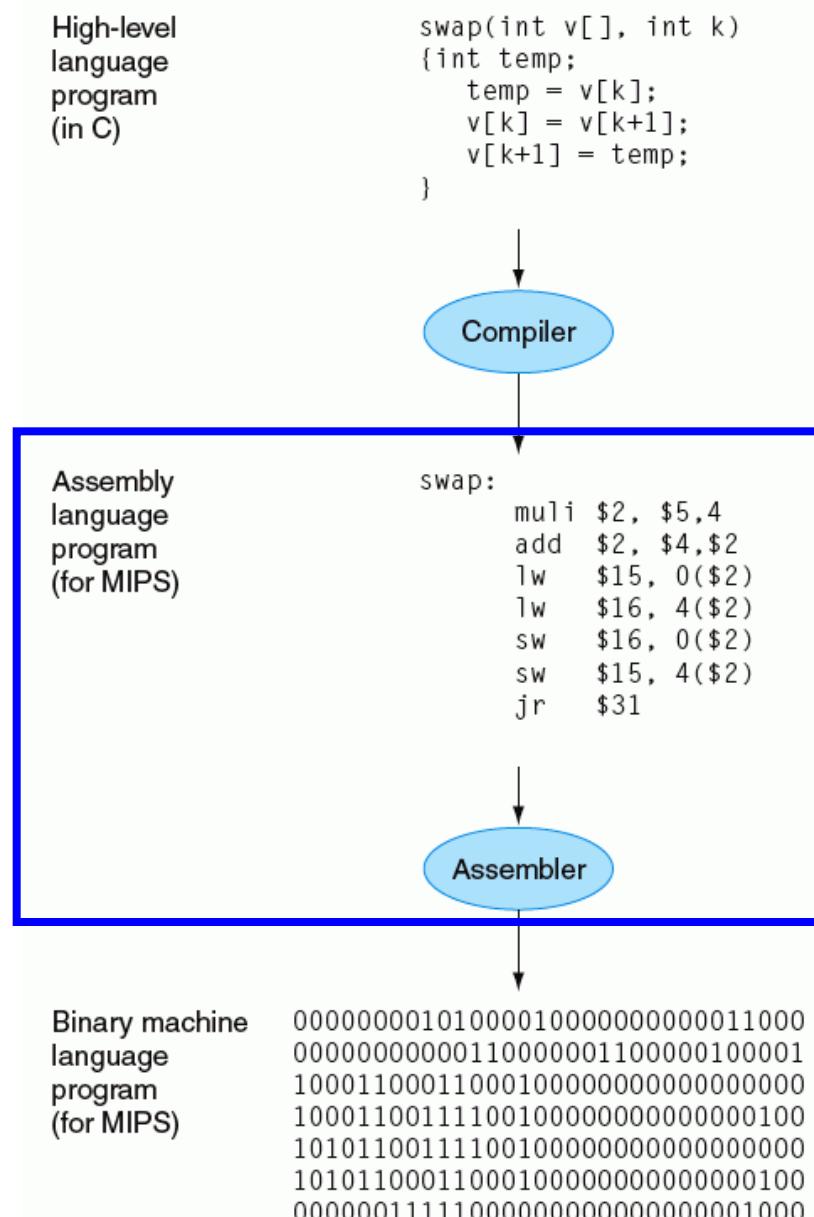
```
10010010  
10001110
```



# Compilação x Interpretação

- Existem vários exemplos tanto de linguagens interpretadas como de linguagens compiladas
- A linguagem C é um exemplo de linguagem compilada
- Python é uma linguagem interpretada
- Java é uma linguagem de programação que utiliza um processo híbrido de tradução
  - O compilador Java traduz o código-fonte em um formato intermediário independente de máquina chamado bytecode
  - Interpretador Java específico da máquina onde irá rodar o programa então traduz os bytecodes para linguagem de máquina e executa o código

# Exemplo de Compilação em 2 etapas

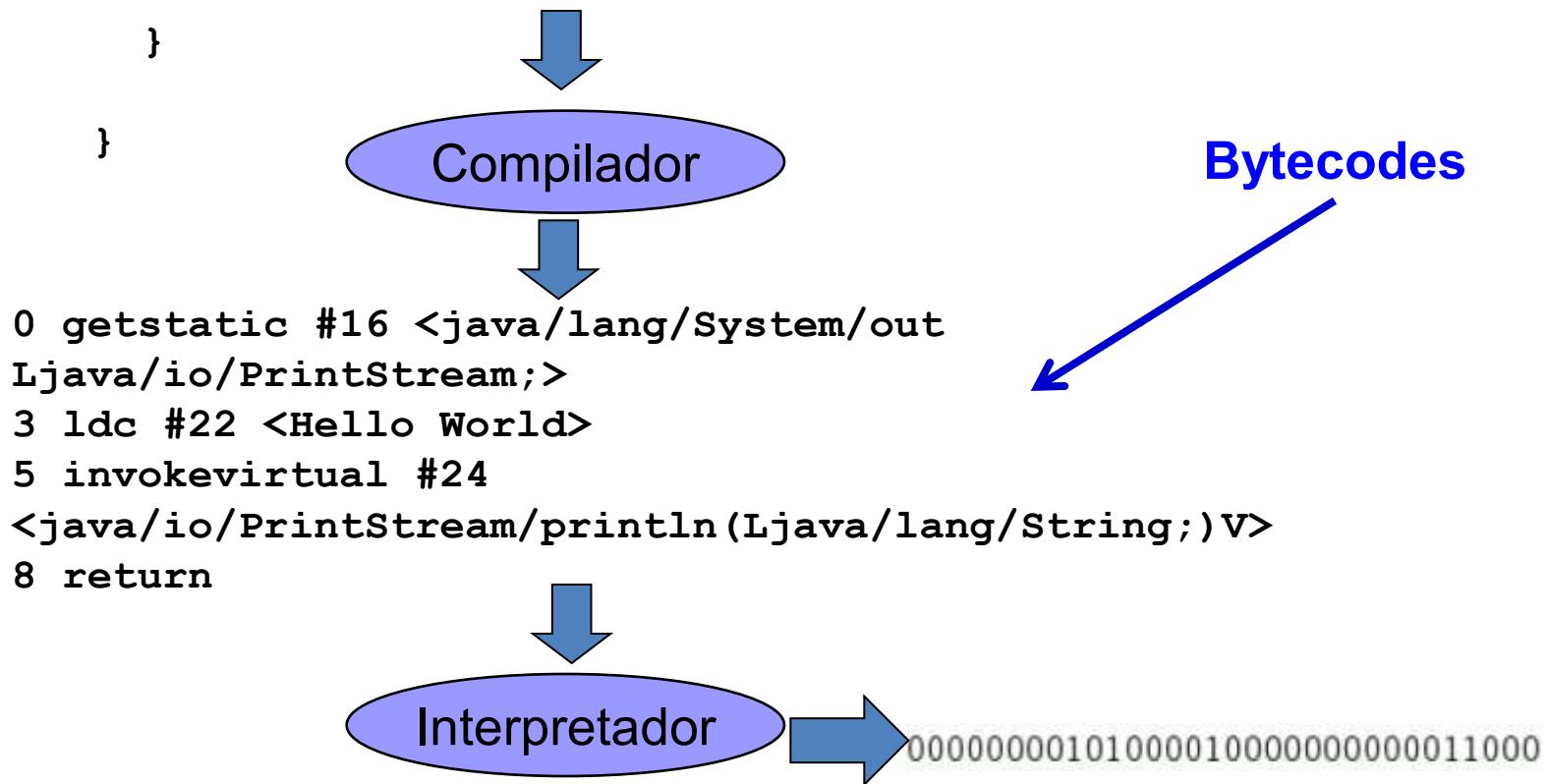


**Maioria dos compiladores C omitem esta parte. Compilam diretamente para linguagem de máquina**



# Exemplo de Compilação e Interpretação

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```



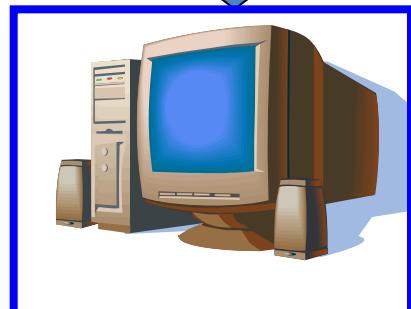
# Compilação em SW e Interpretação em HW

```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



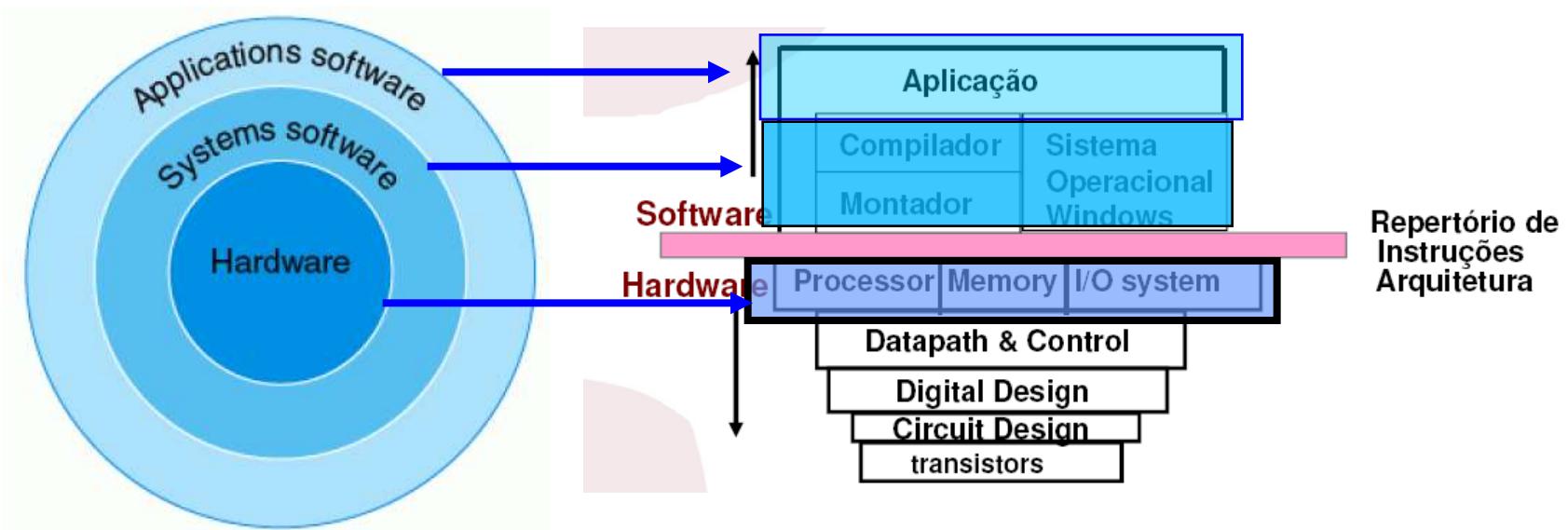
```
0000000001010000100000000000011000  
00000000000110000001100000100001  
1000110001100010000000000000000000  
1000110011110010000000000000000100  
10101100111100100000000000000000000  
1010110001100010000000000000000100  
00000011111000000000000000000001000
```

# HW interpreta instrução a instrução



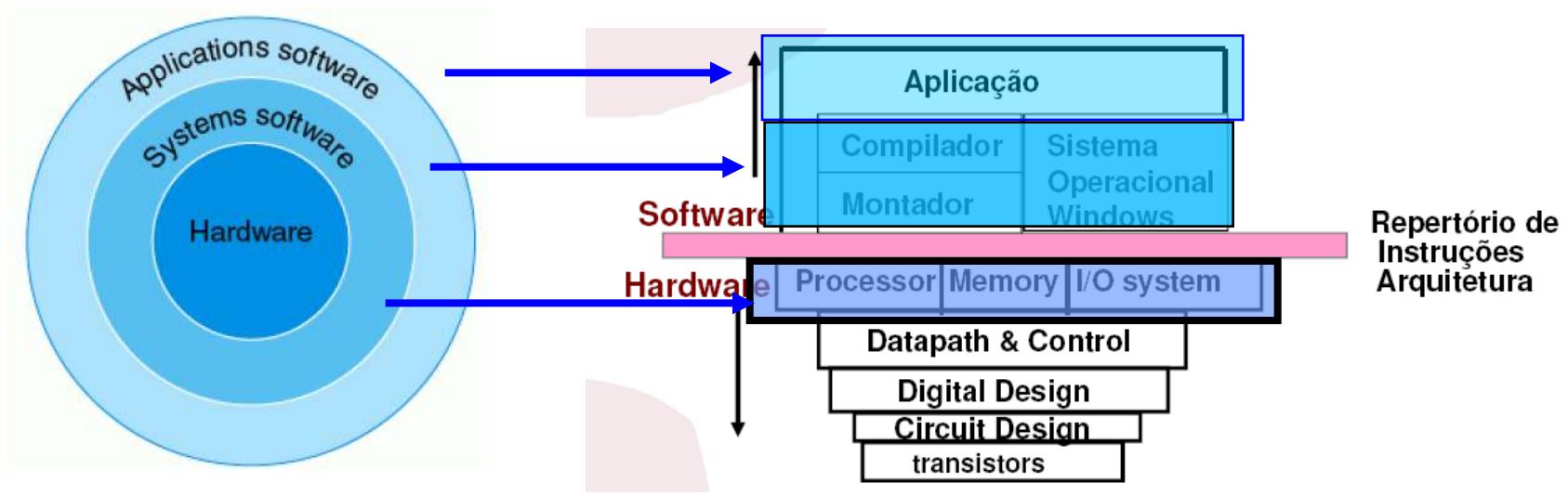
000000001010000100000000000011000

# Abstrações de um Computador



- Faz-se necessário a criação de camadas de abstrações que escondam detalhes de implementação de um computador para desenvolver as aplicações atuais cada vez mais complexas

# Abstrações de Software



- Aplicação: abstração de dados, armazenamento, procedural
- Softwares de sistema
  - Compiladores: abstração do repertório de instruções da máquina
  - Sistema Operacional: abstração de concorrência, recursos de HW, hierarquia de memória

# Obrigado!

# Infraestrutura de HW

Implementação Multiciclo de um Processador Simples  
(Unidade de Controle e Suporte a Exceções)

Prof. Adriano Sarmento

# Implementação da Unidade de Controle

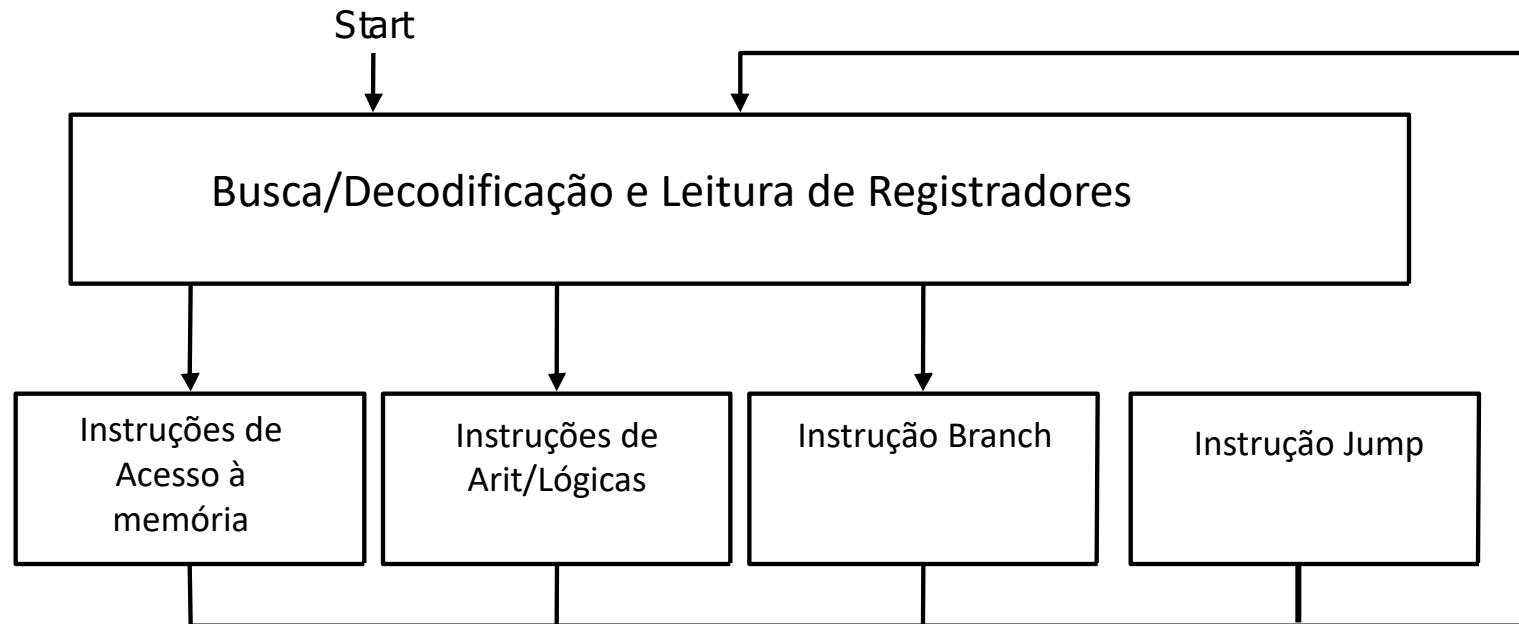
## ■ Monociclo

- Simplicidade de implementação
  - Tabela verdade usada para gerar lógica combinacional

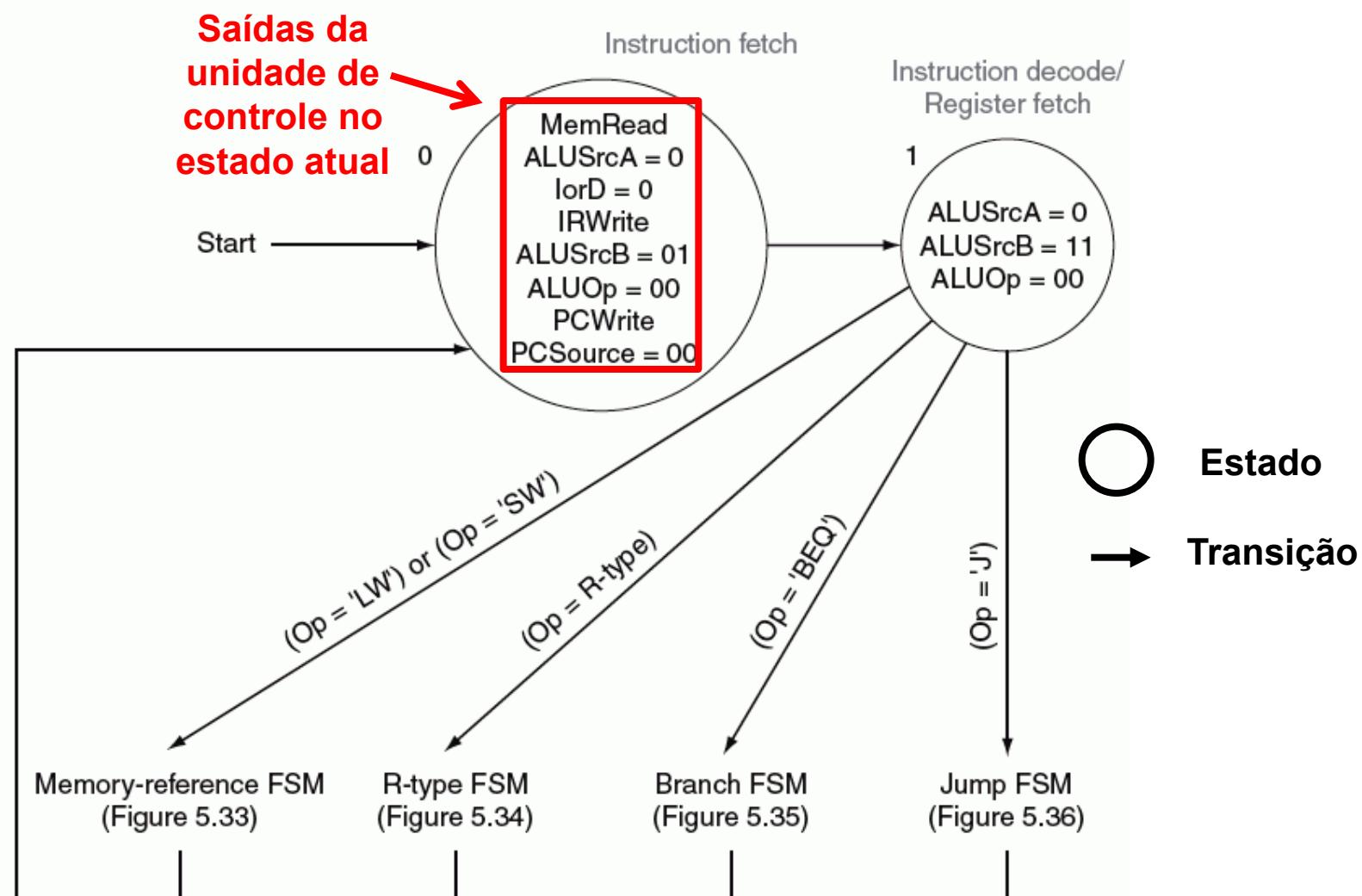
## ■ Multiciclo

- Complexa
  - Estado do processamento da instrução deve ser levado em conta
  - Pode ser projetada com o auxílio de uma máquina de estados finita (**Finite State Machine – FSM**)

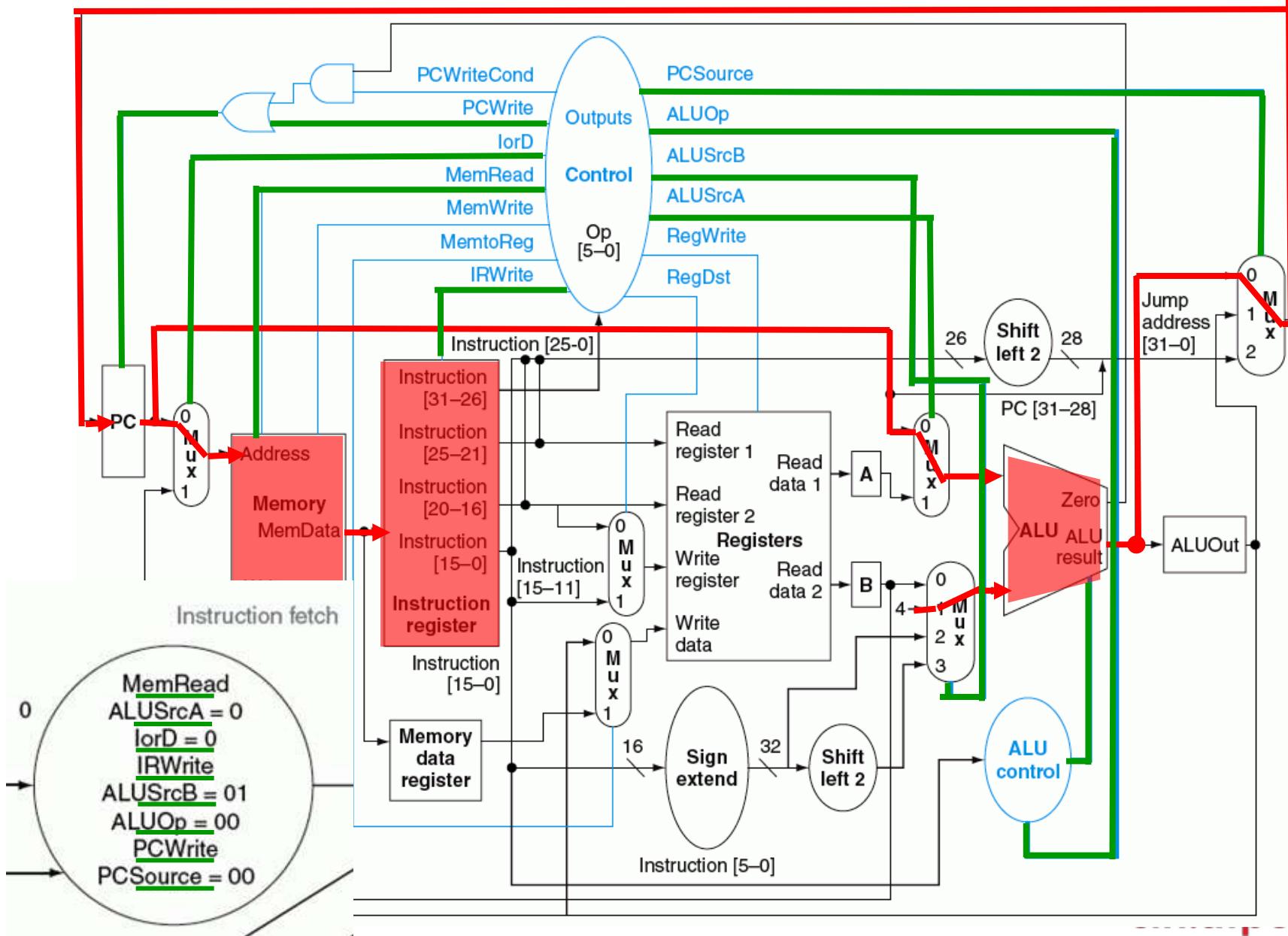
# FSM da Unidade de Controle – Visão Abstrata



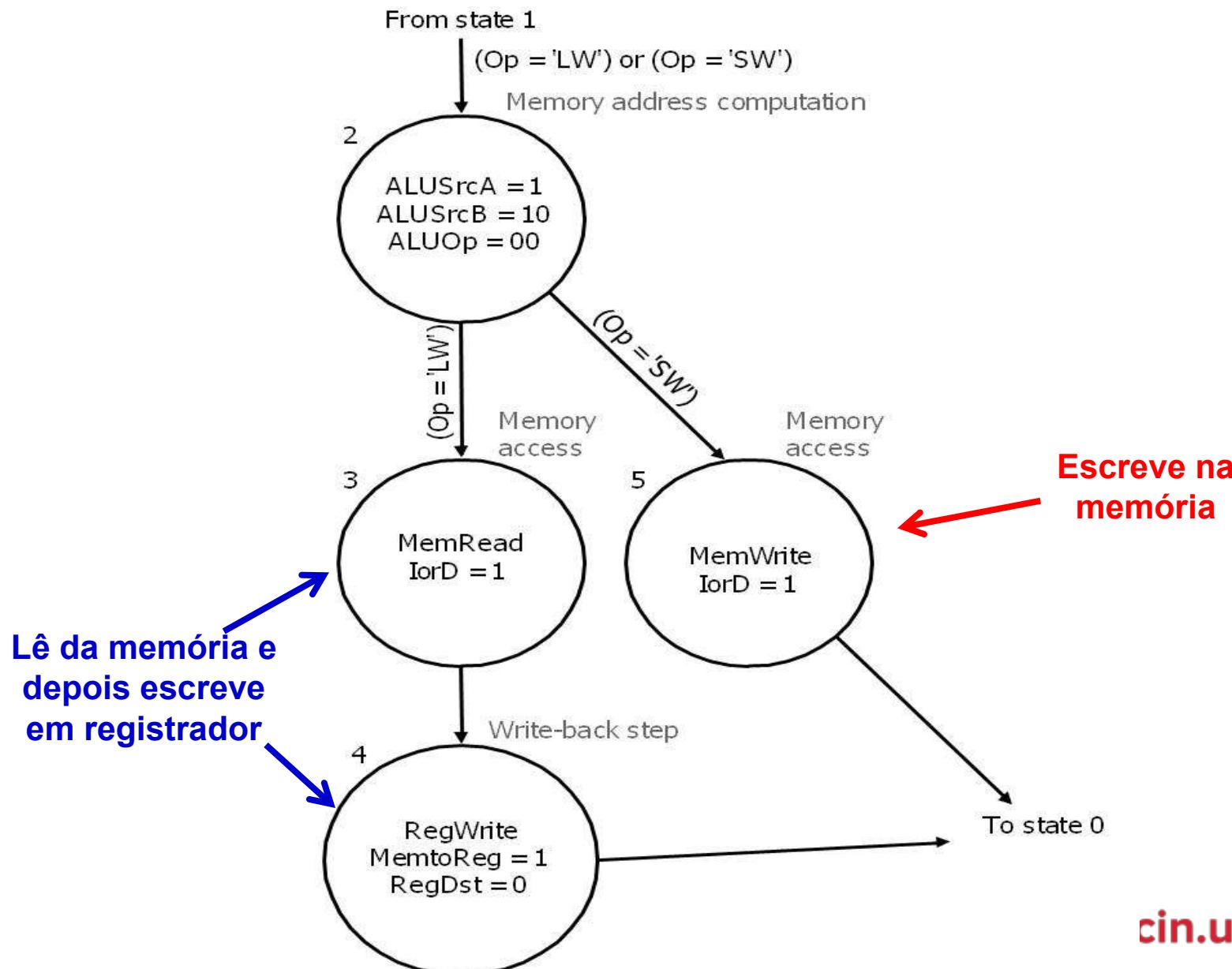
# FSM da Busca, Decodificação e Leitura de Registradores



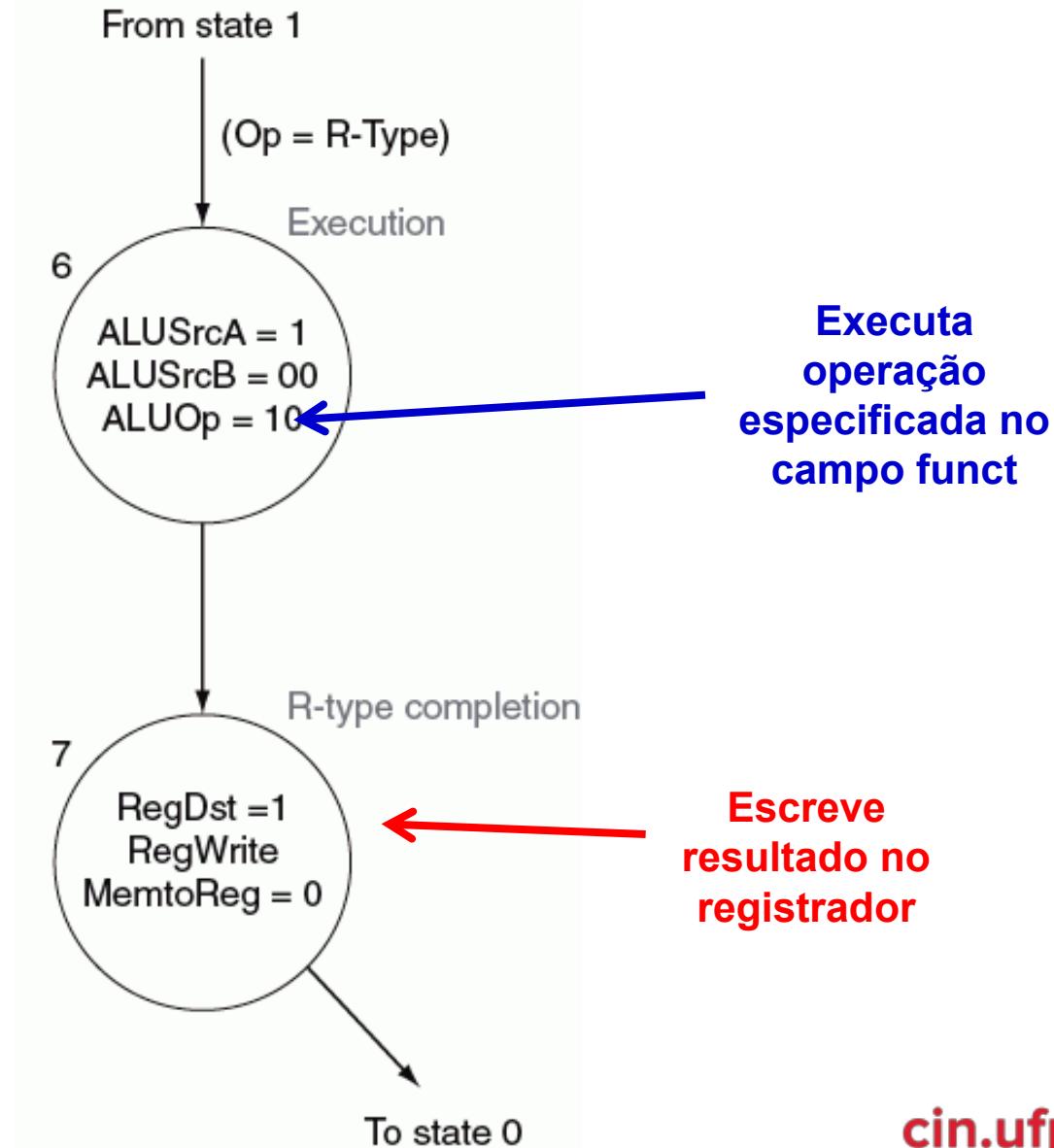
# Busca Instrução e Atualiza de PC



# Instruções de Acesso a Memória

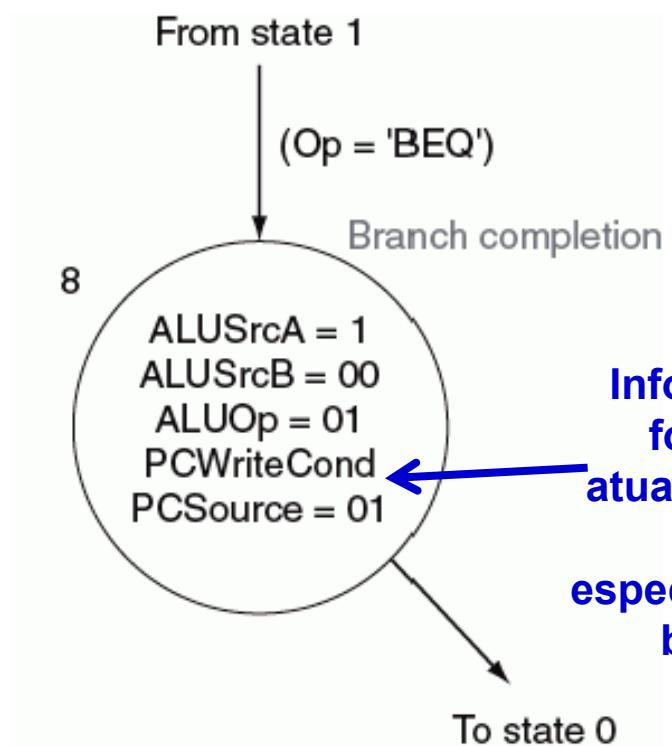


# Instruções de Aritméticas/Lógicas

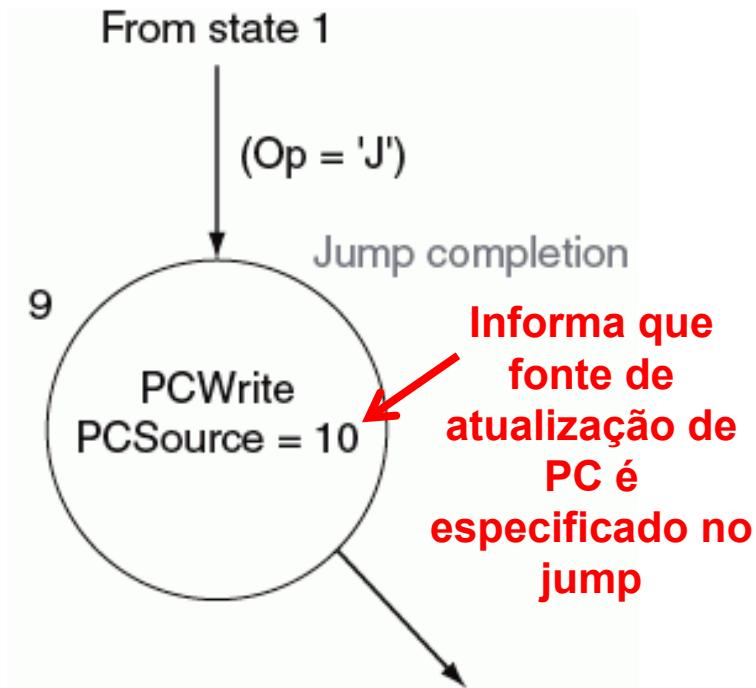


# Instruções de Branch e Jump

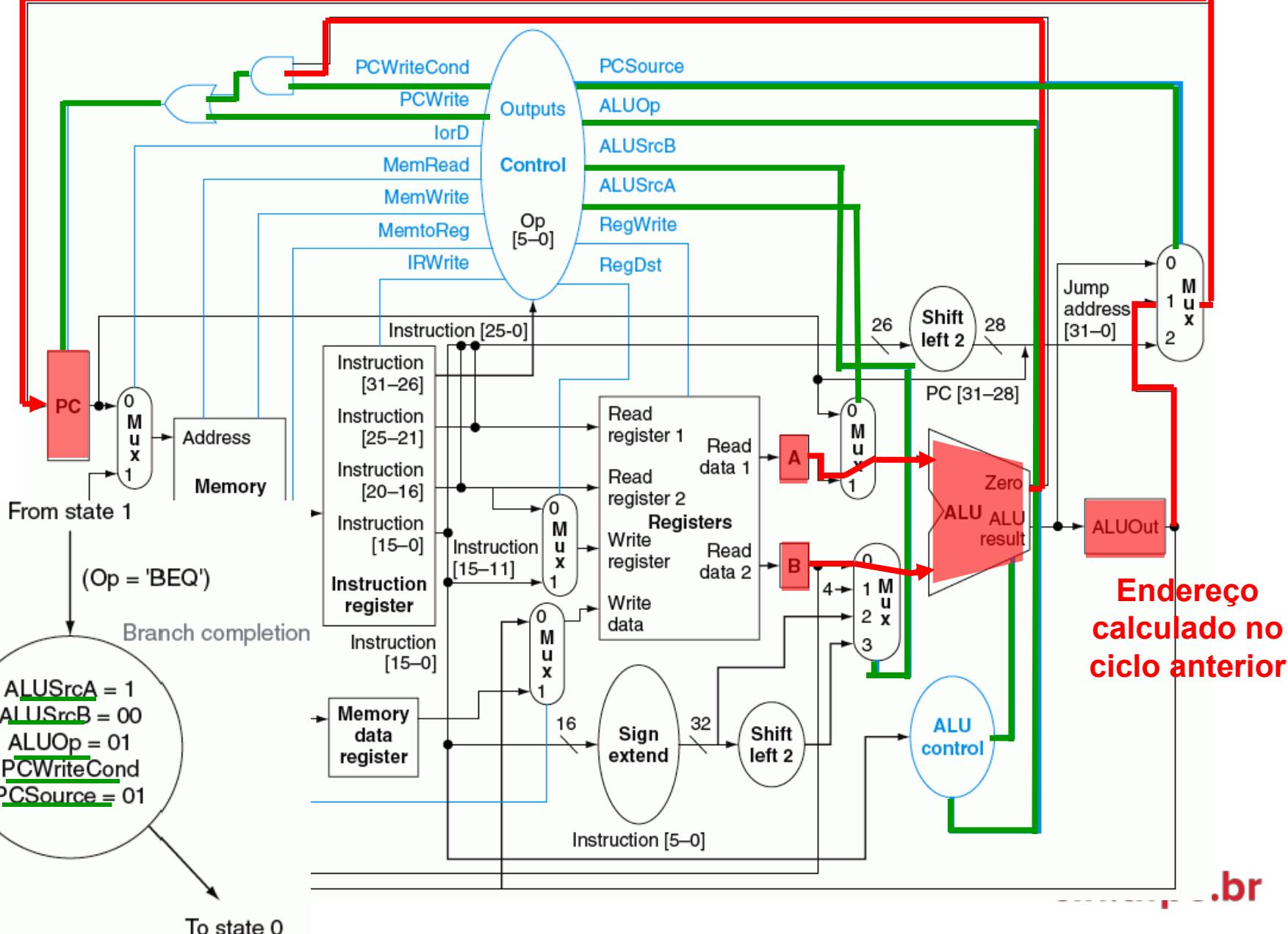
## Branch



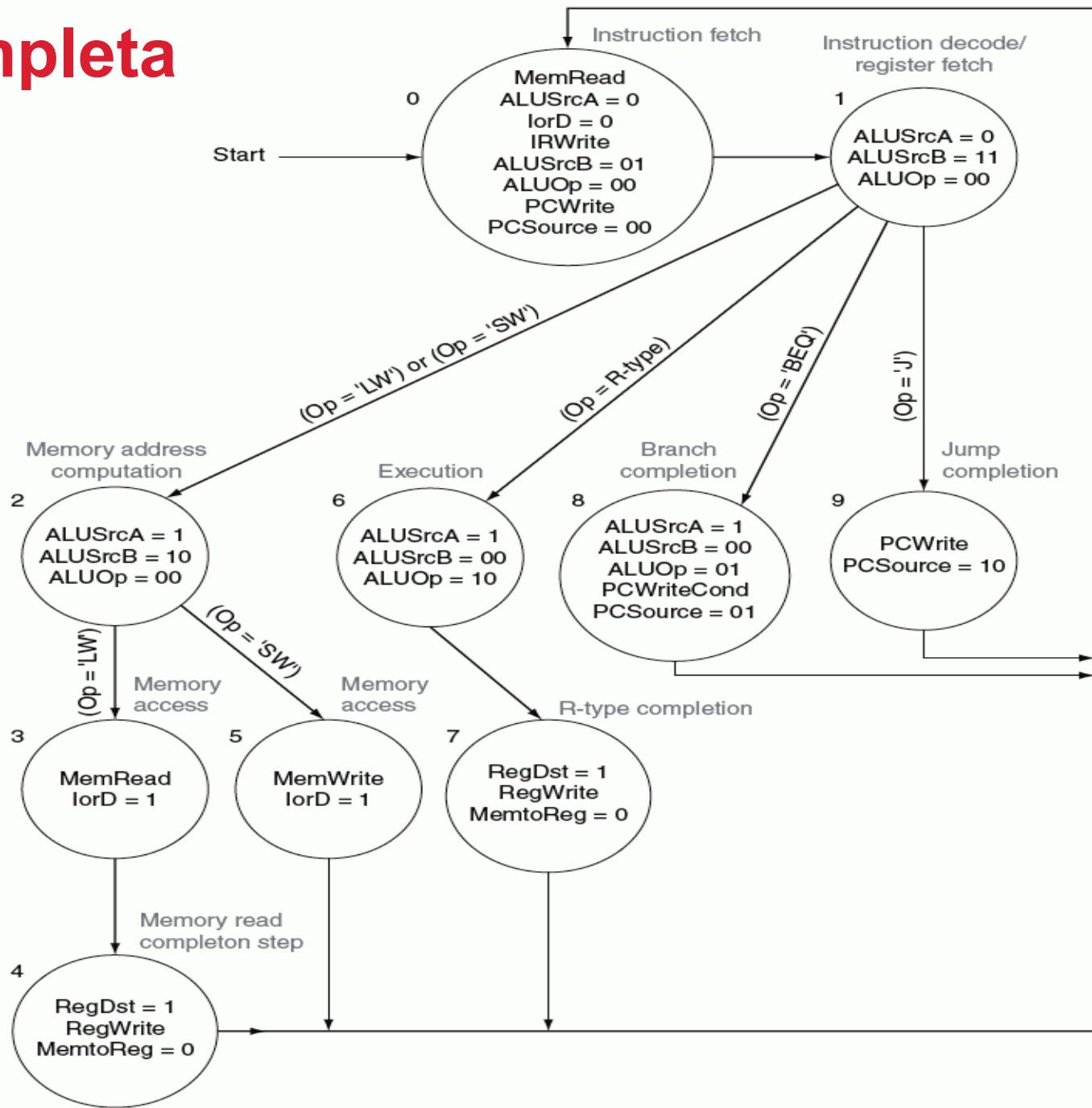
## Jump



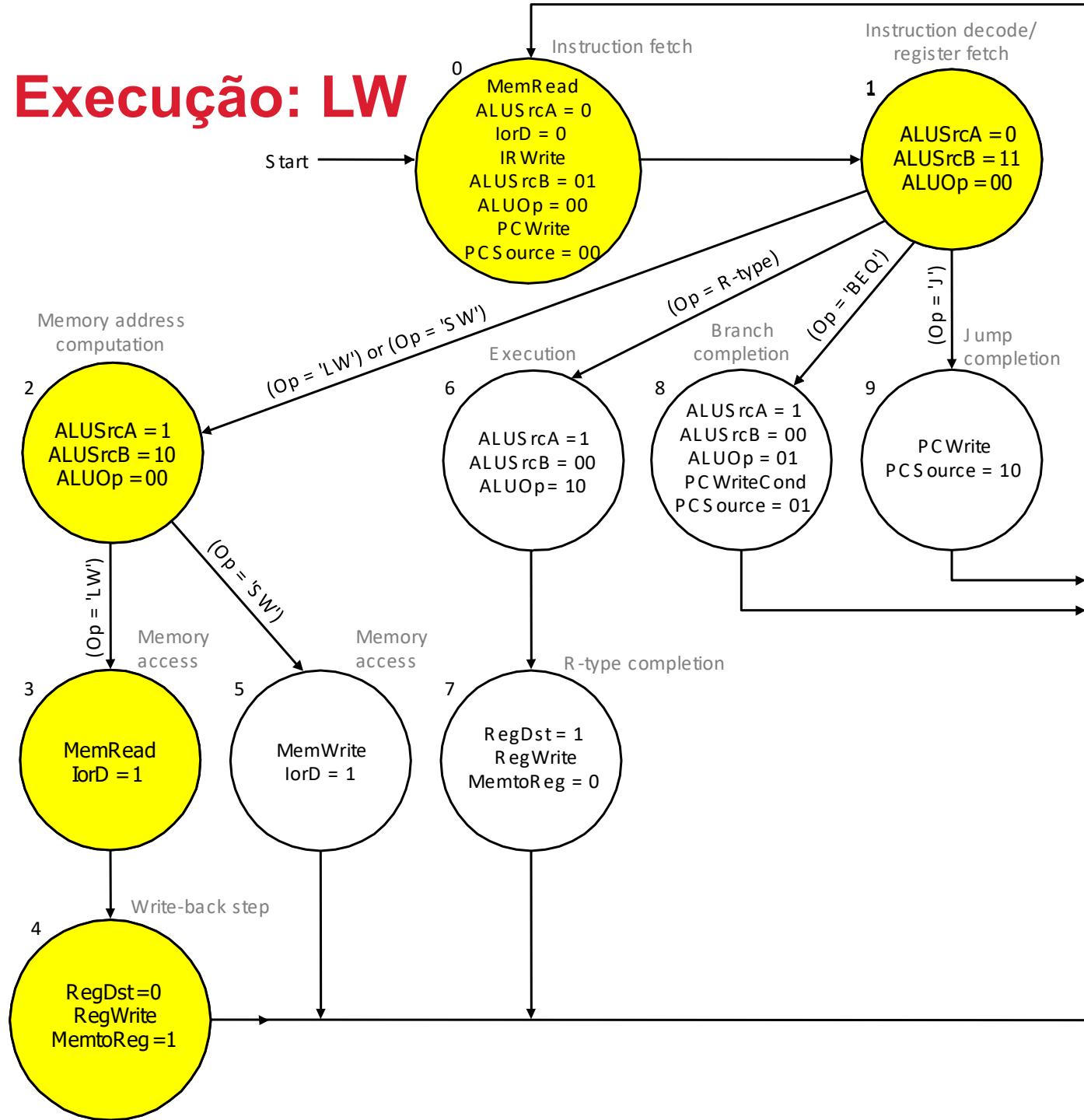
# Opera com a ALU: Branch



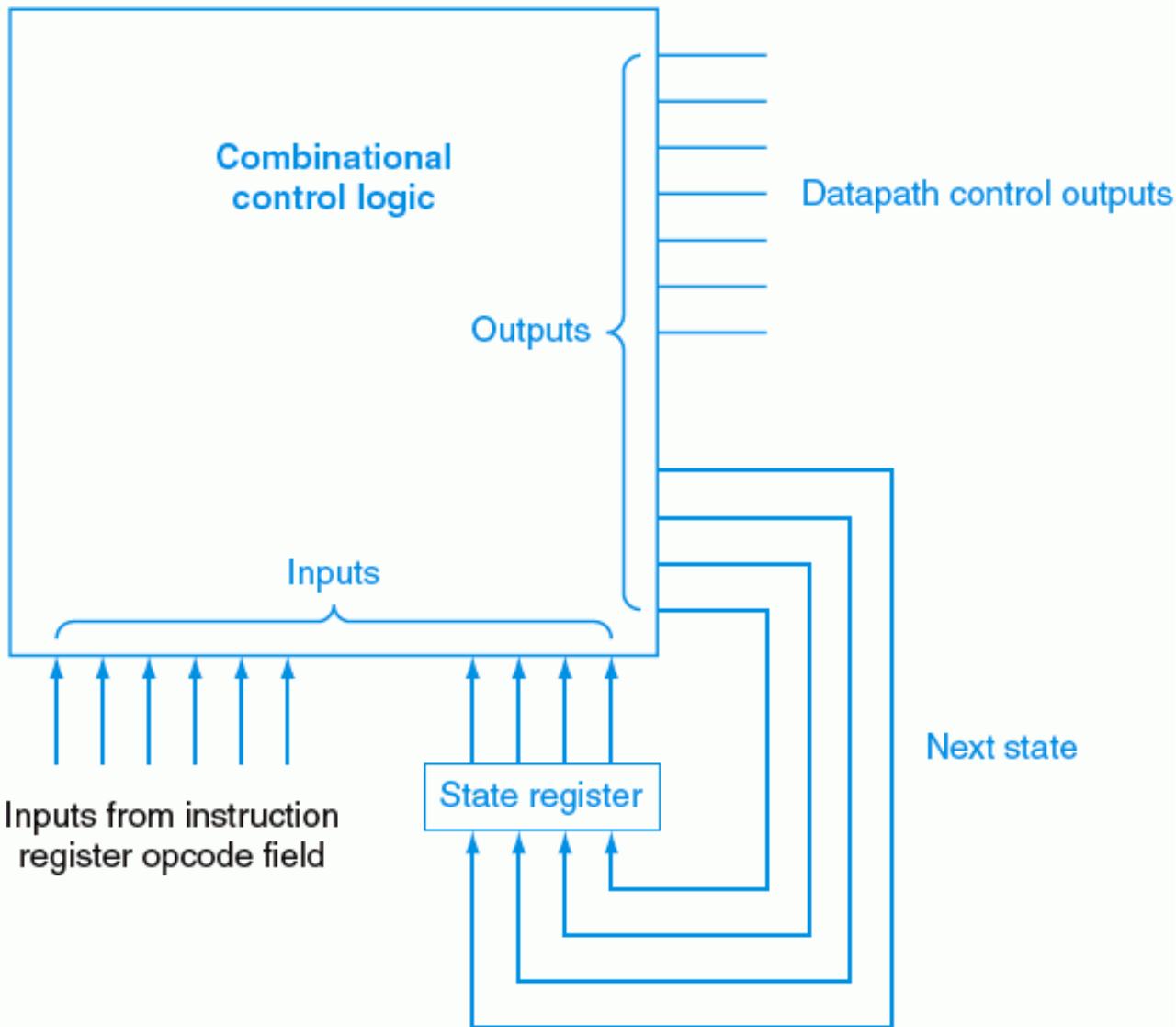
# FSM Completa



# Exemplo de Execução: LW



# Implementação de Unidade de Controle



# CPU: Exceções

- Sequência de execução é alterada devido a eventos não esperados:
  - internos:
    - opcode inexistente
    - overflow
    - divisão por zero
  - externos:
    - dispositivo de entrada/saída

# Exceções

- Execução do programa é interrompida e uma rotina de tratamento é executada
  - Valor do PC deve ser guardado
  - O endereço da rotina de tratamento deve ser carregado em PC

# Perguntas Importantes

- Onde guardar o endereço do PC?
  - Registrador
    - MIPS: EPC
  - Pilha
- Onde o endereço da sua subrotina de tratamento deve ser guardado?
  - Valor fixo – MIPS
    - 0x8000 0180
    - precisa-se saber a causa da exceção
  - Vetor de endereços na memória
    - Endereço pré-estabelecido informa a causa da exceção

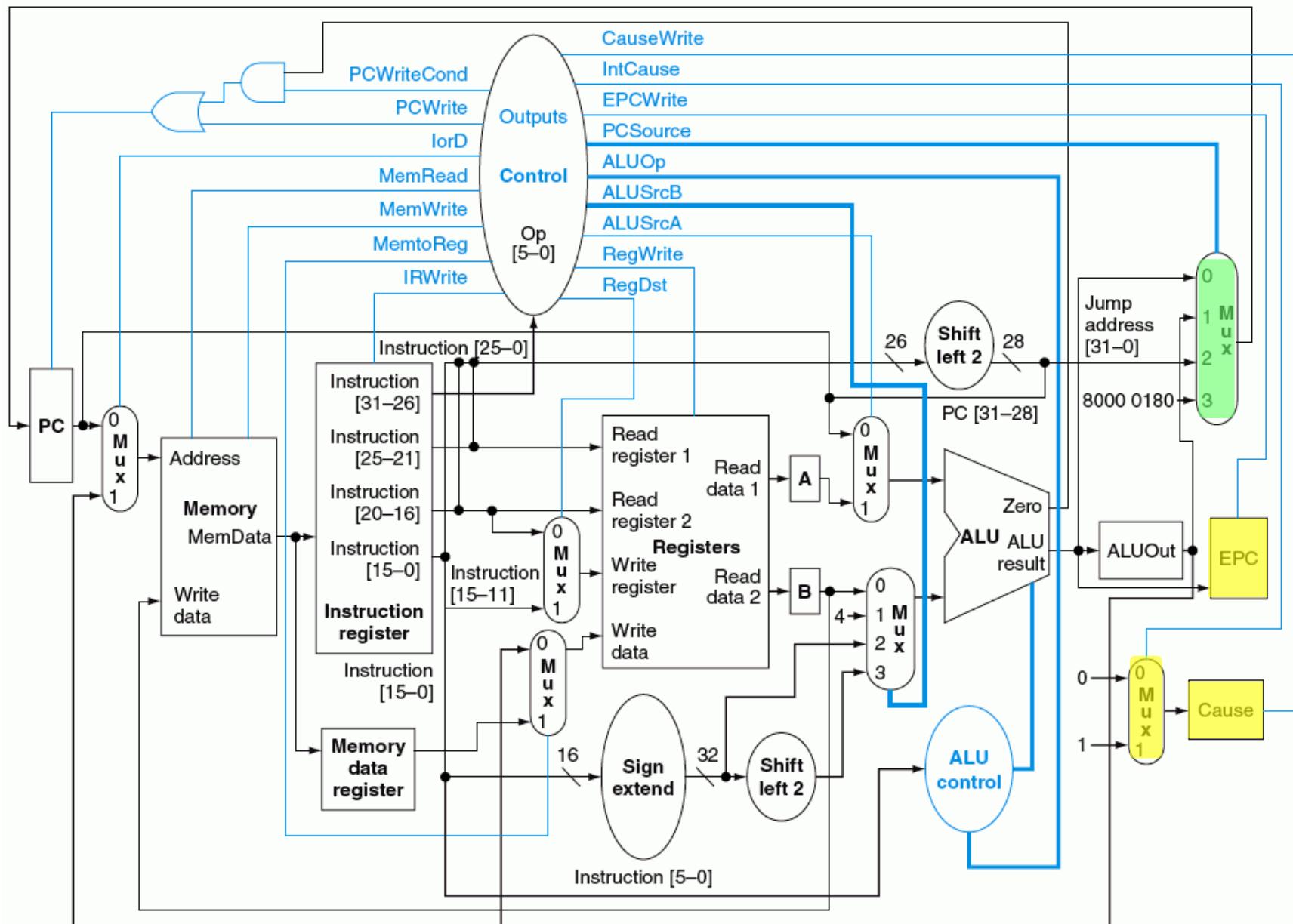
# Exceções - MIPS

- Tipos de Exceções:
  - Instrução Indefinida
  - Overflow aritmético
- Registrador EPC
  - guarda endereço da instrução afetada
- Registrador de Causa
  - identifica o tipo de evento que causou a exceção

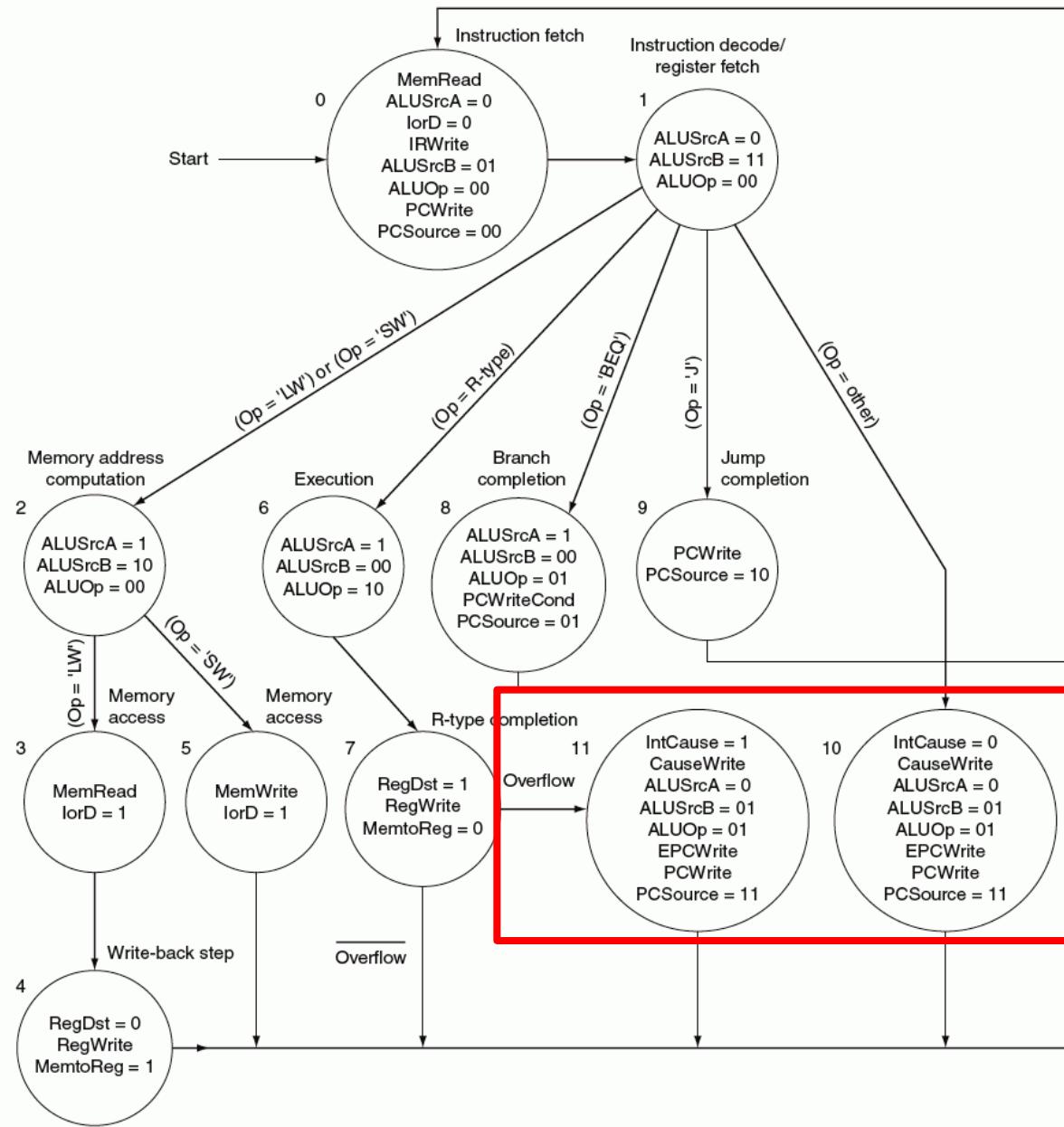
# Implementando Exceções

- Assumindo que a CPU trata dois tipos de exceção:
  - Instrução inexistente (código da causa = 0)
  - Overflow (código da causa = 1)
- Precisa-se mudar unidade de processamento para:
  - Armazenar causa
  - Armazenar endereço de instrução afetada (EPC)
  - Escrever no PC endereço da subrotina de tratamento
    - 0x8000 0180
- Precisa-se mudar unidade de controle para:
  - Permitir escrita no registrador de causa e EPC
  - Selecionar o que é escrito dependendo da exceção
  - Permitir a atualização do PC com endereço da rotina de tratamento

# CPU Trabalhando com Exceções

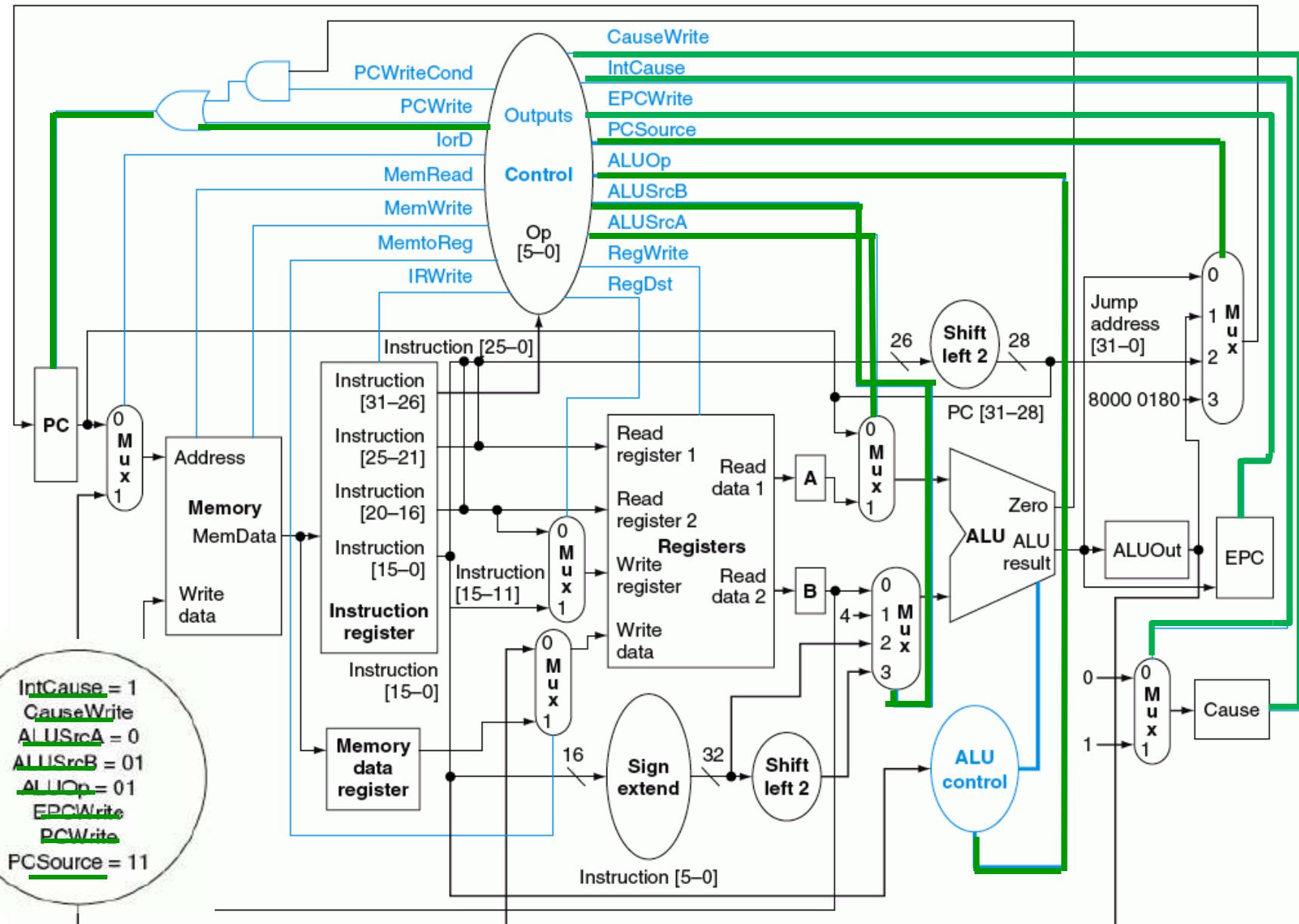


# Trabalhando com Exceções - Controle



Novos estados  
decorrentes do  
tratamento de  
exceção

# CPU Trabalhando com Exceções



# Infraestrutura de HW

Implementação Multiciclo de um Processador Simples

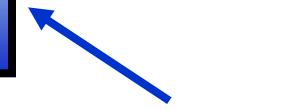
Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

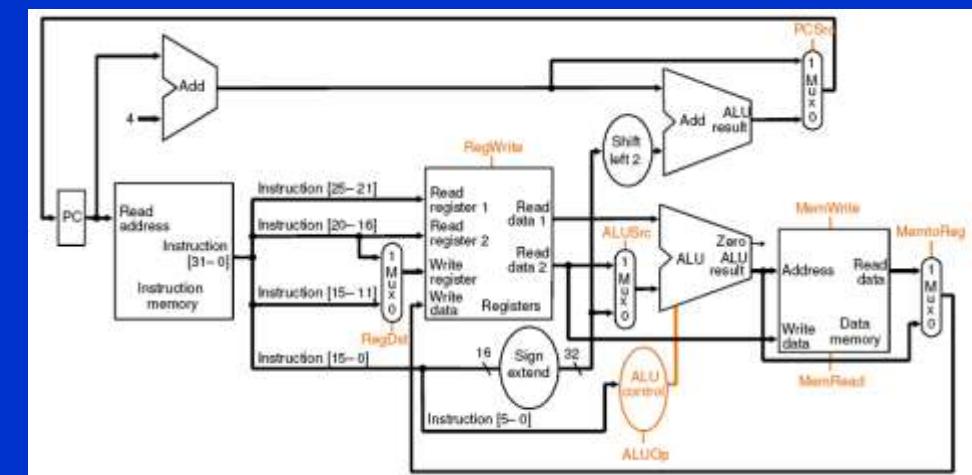
- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Interface HW/SW: ISA

Software

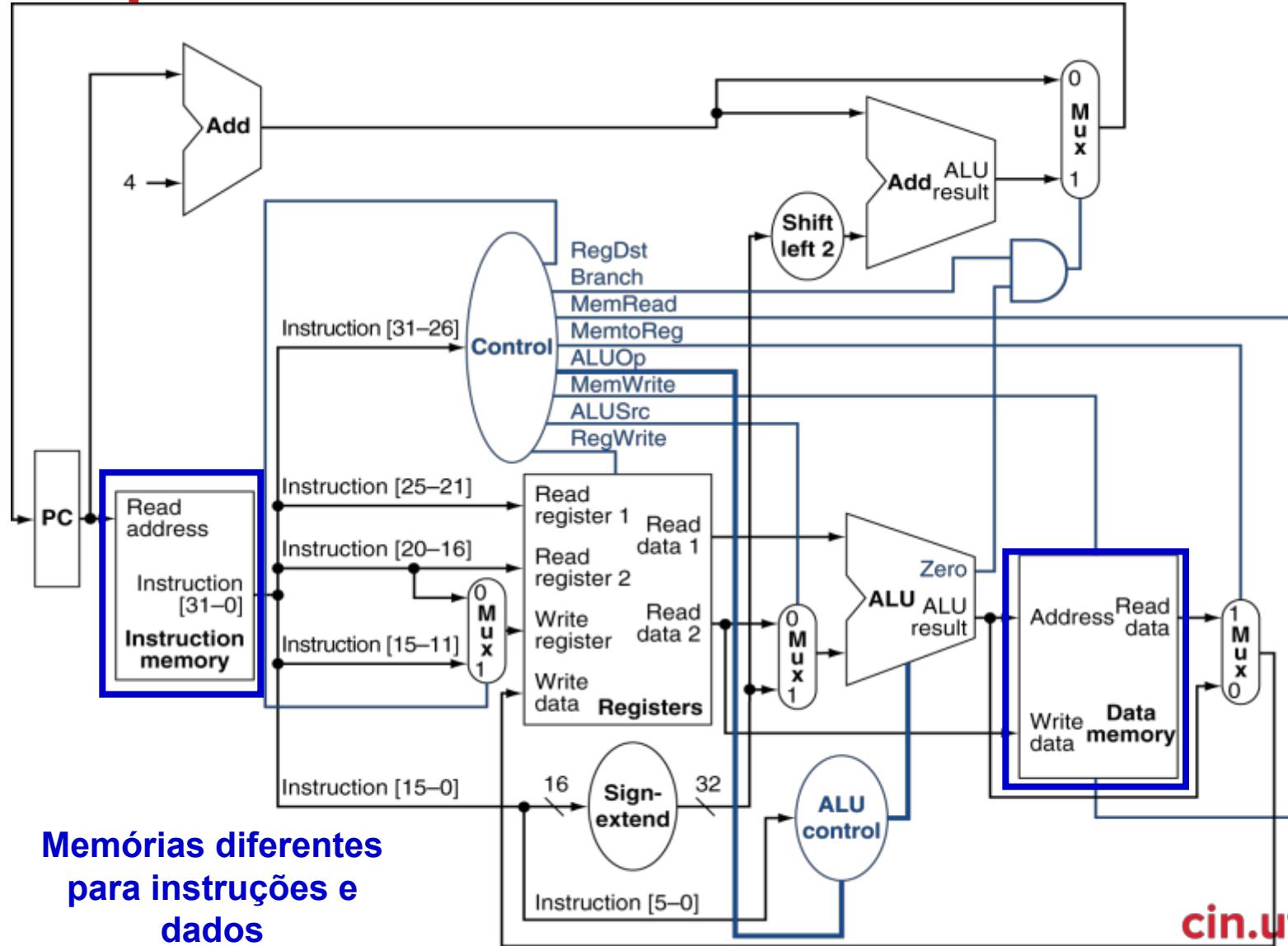


Repertório de  
Instruções da  
Arquitetura



Hardware

# Implementação Monociclo de CPU Simples



# Análise de Implementação Monociclo

- Vantagem:
  - Simplicidade de implementação
    - Datapath e Controle
- Desvantagens:
  - Custo de hardware
    - Duplicação de componentes devido a restrição de um ciclo para o processamento de instrução
  - Desempenho
    - Ciclo de clock longo para comportar instrução mais lenta
    - Instruções que demandam menos tempo deixam CPU ociosa
    - Implementação pouco eficiente

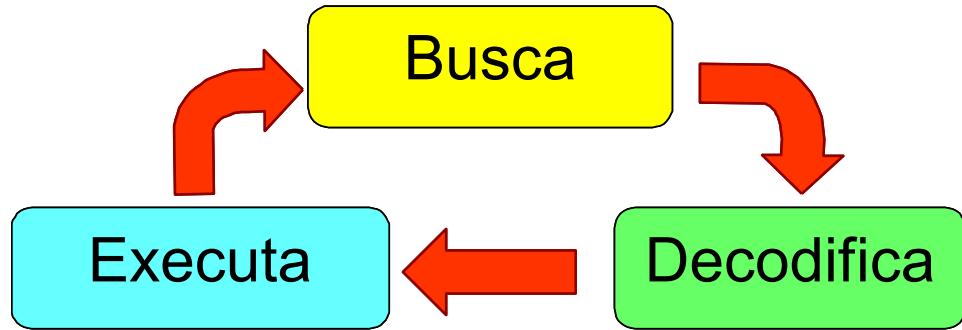
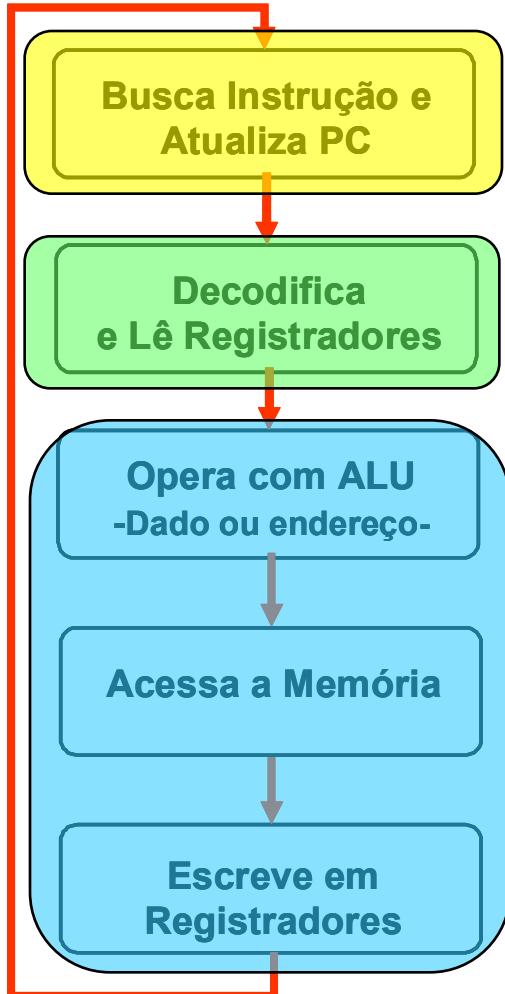
# Como Melhorar?

## ■ Implementação Multiciclo!

- Dividir processamento de instrução em diferentes estágios
- Cada estágio é executado em um ciclo de clock
  - Ciclo pode ser menor
- Diferentes instruções requerem quantidade de estágios diferentes
  - Algumas instruções podem ser processadas em menos tempo
  - Tempo médio de processamento pode melhorar
- Mesmo componente de HW pode ser utilizado em estágios diferentes

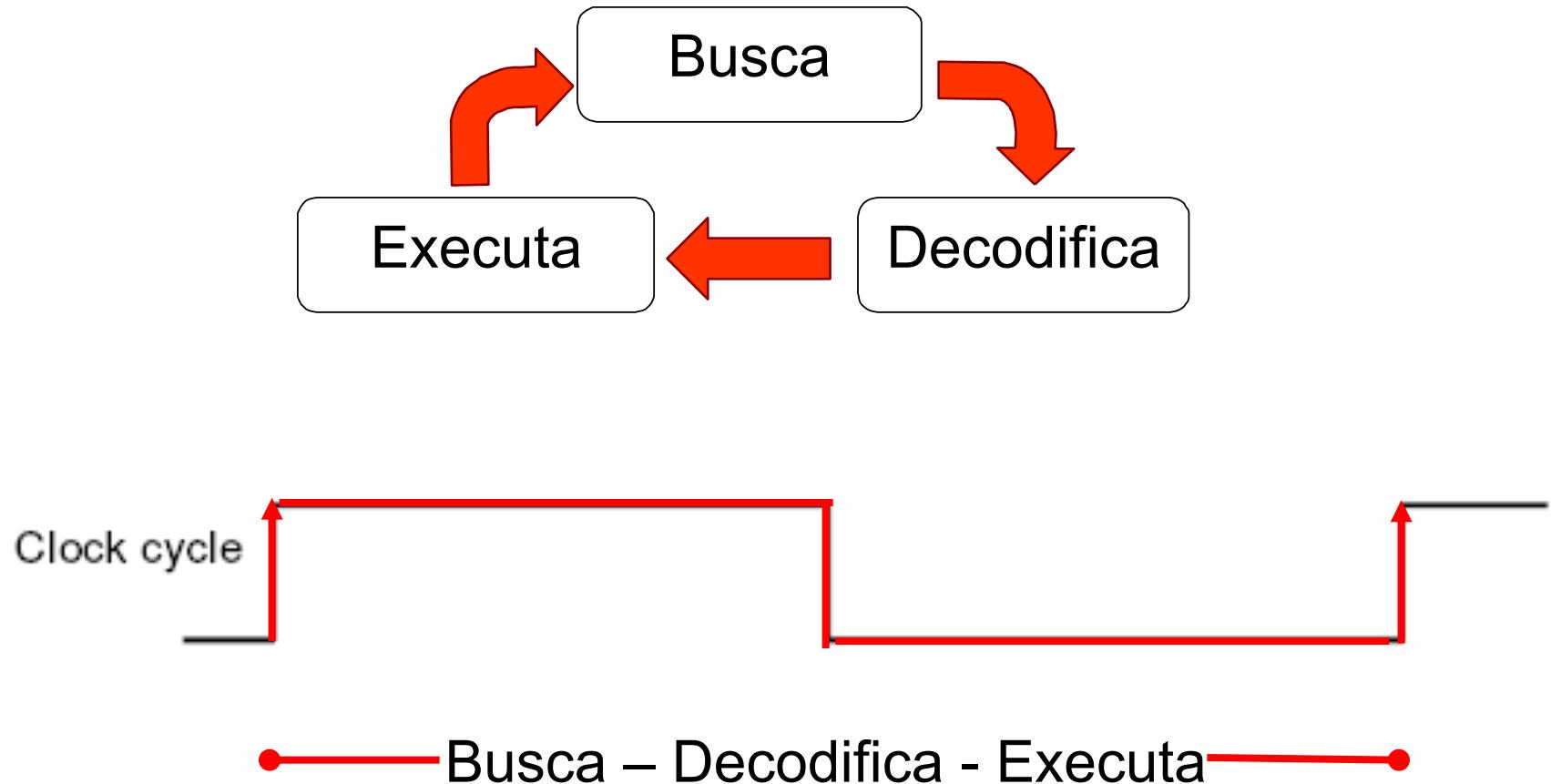
**Uso mais eficiente de HW!**

# Quebrando Processamento de Instrução em Estágios

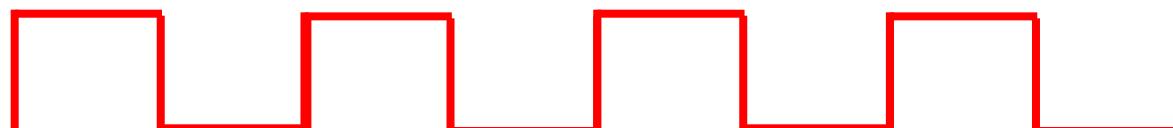


- Dois primeiros estágios sempre executados
- Dependendo da instrução, alguns dos outros estágios podem não ser executados

# Implementação Monociclo



# Implementação Multiciclo



Busca - Decodifica - ALU -

Memória – Escrita Reg.

# Instruções

## Repertório

Instrução	Descrição
<b>ADD rd, rs, rt</b>	$rd \leftarrow rs + rt$
<b>SUB rd, rs, rt</b>	$rd \leftarrow rs - rt$
<b>AND rd, rs, rt</b>	$rd \leftarrow rs \text{ and } rt$ (bit a bit)
<b>OR rd, rs, rt</b>	$rd \leftarrow rs \text{ or } rt$ (bit a bit)
<b>SLT rd, rs, rt</b>	Se $rs < rt$ , $rd \leftarrow 1$ , senão $rd \leftarrow 0$
<b>LW rt, desl(rs)</b>	Carrega palavra de mem. em registrador rt
<b>SW rt, desl(rs)</b>	Armazena conteúdo de registrador rt em mem.
<b>BEQ rs, rt, end</b>	Desvio para end, se $rs == rt$
<b>J end</b>	Desvio para end

## Formato

Aritméticos/Lógicos

op	rs	rt	rd	sa	funct
----	----	----	----	----	-------

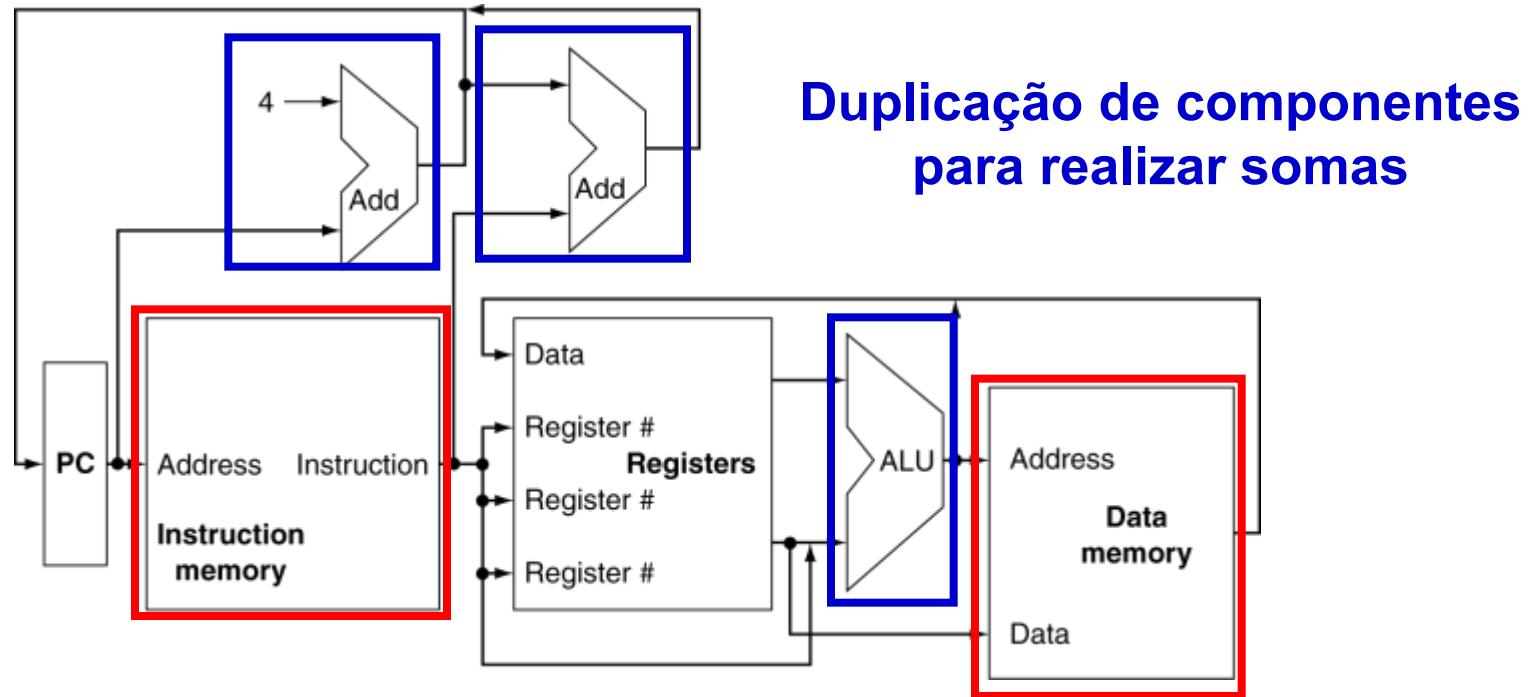
Armazenamento/Branch

op	rs	rt	deslocamento/endereço
----	----	----	-----------------------

Jump

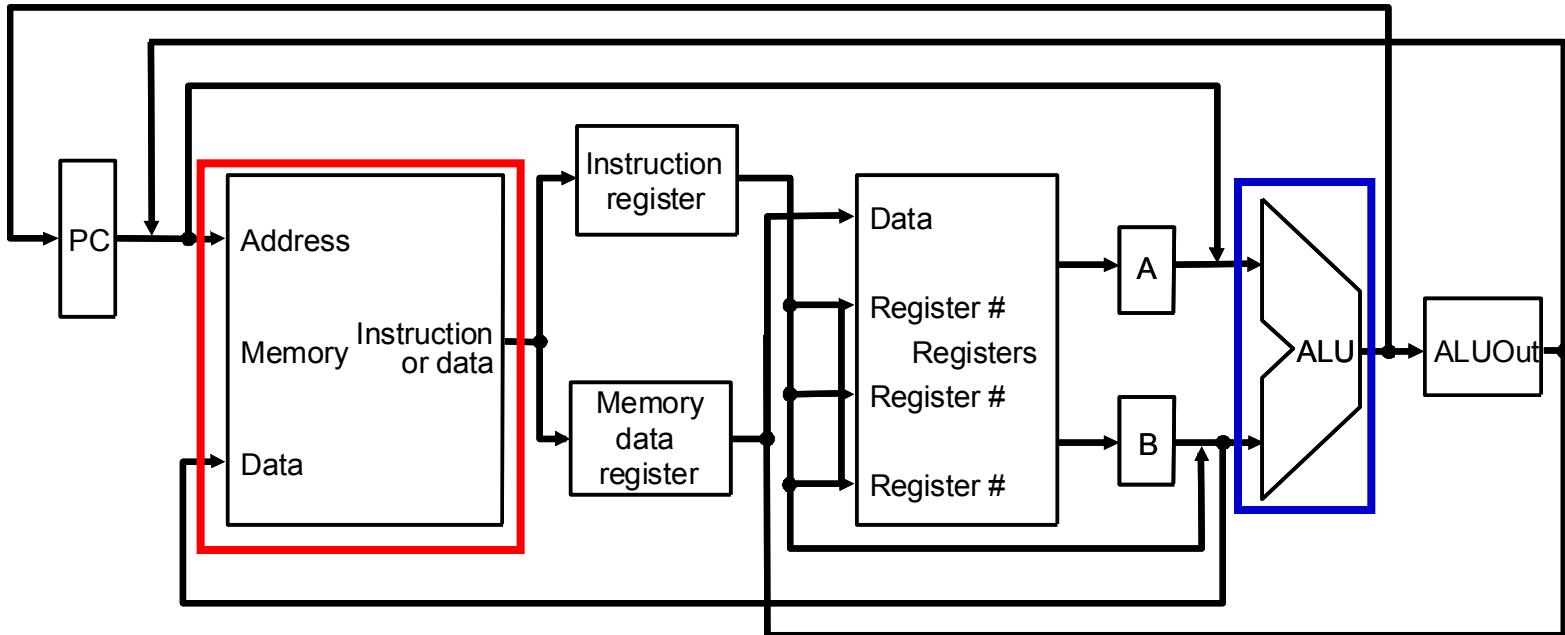
op	endereço
----	----------

# Visão Abstrata de Implementação Monociclo



Memórias diferentes para  
instruções e dados

# Visão Abstrata de Implementação Multiciclo

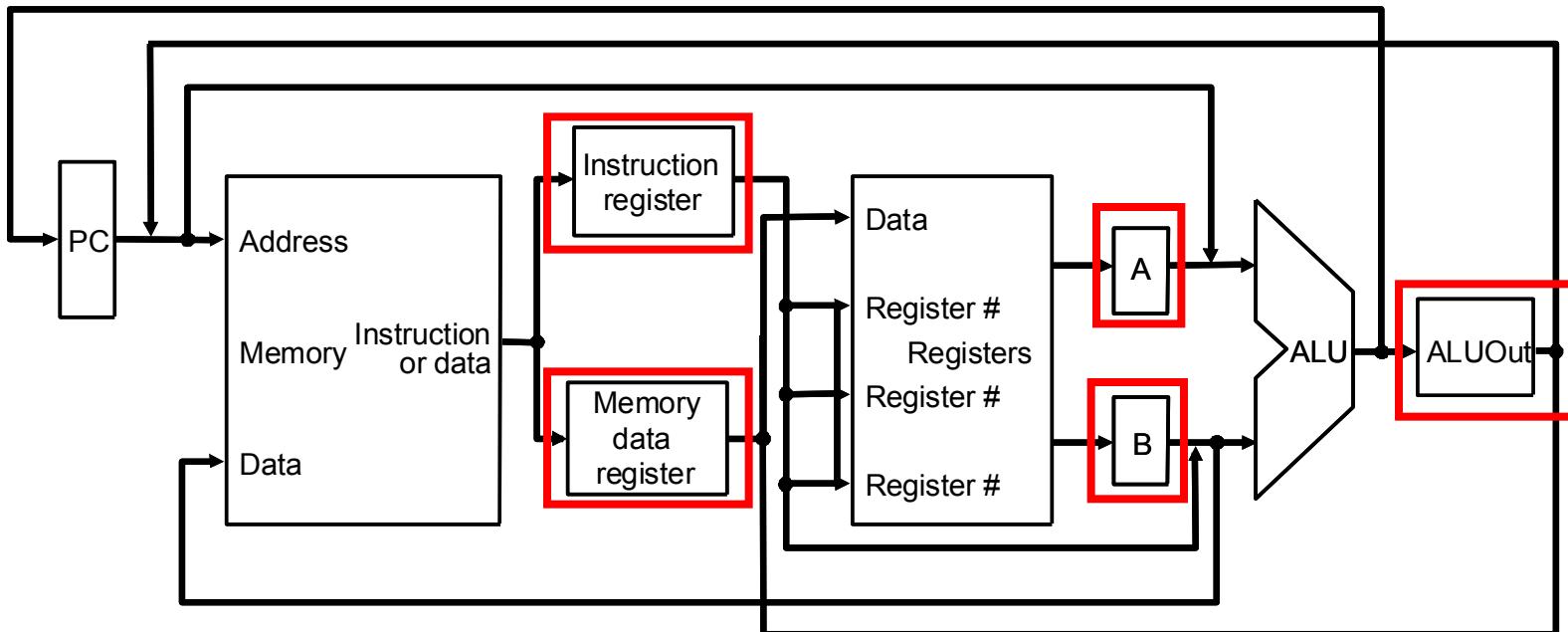


**Memória única para instruções e dados**

**Componente único para realizar operações**

- Compartilhamento de recursos
  - Utilizados em estágios diferentes (ciclos de clock diferentes)

# Mais Registradores...



**Registradores auxiliares  
entre unidades funcionais  
maiores**

- Valores que serão usados na mesma instrução, mas em ciclo de clock diferente devem ser armazenados nestes registradores

# Processamento na Implementação Multiciclo

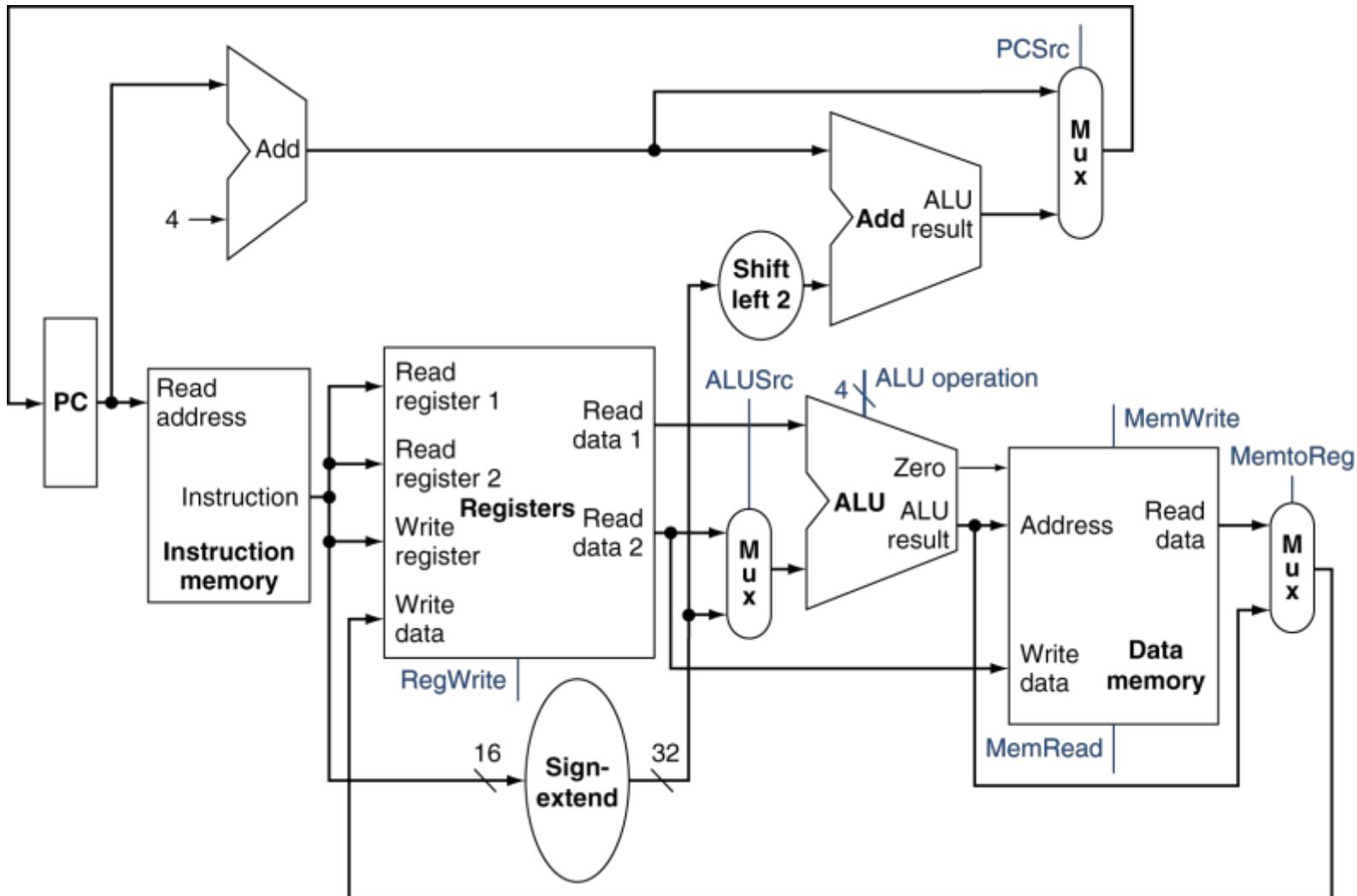
- Um ciclo de clock pode acomodar uma das operações:
  - Acesso de memória
  - Acesso ao banco de registradores (2 leituras ou 1 escrita)
  - Operação da ALU
- Unidades funcionais só podem ser utilizados uma vez durante um ciclo
  - ALU, memória, banco de registradores
  - Exceção: registradores auxiliares
    - IR – Armazena instrução lida da memória
    - MDR – Armazena dado lido da memória
    - A e B – Armazenam operandos lidos do banco de registradores
    - ALUOut – Armazena saída da ALU

# Processamento na Implementação Multiciclo

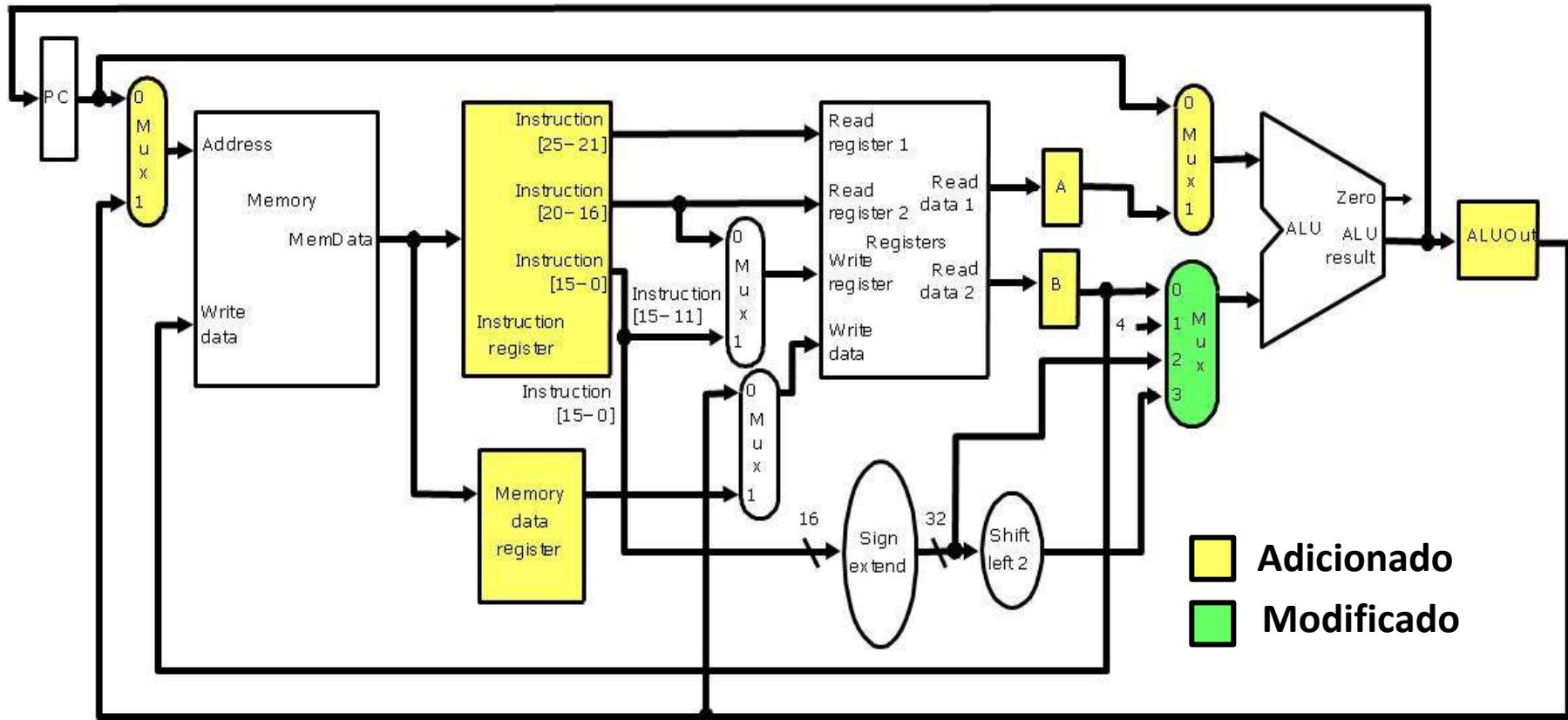
- Dados de uma instrução necessários para outra instrução são armazenados nas unidades visíveis ao programador
  - Memória
  - Banco de registradores
  - PC
- Dados necessários entre estágios (ciclos) diferentes de uma mesma instrução são armazenados nos registradores auxiliares

# Unidade de Processamento

## Monociclo

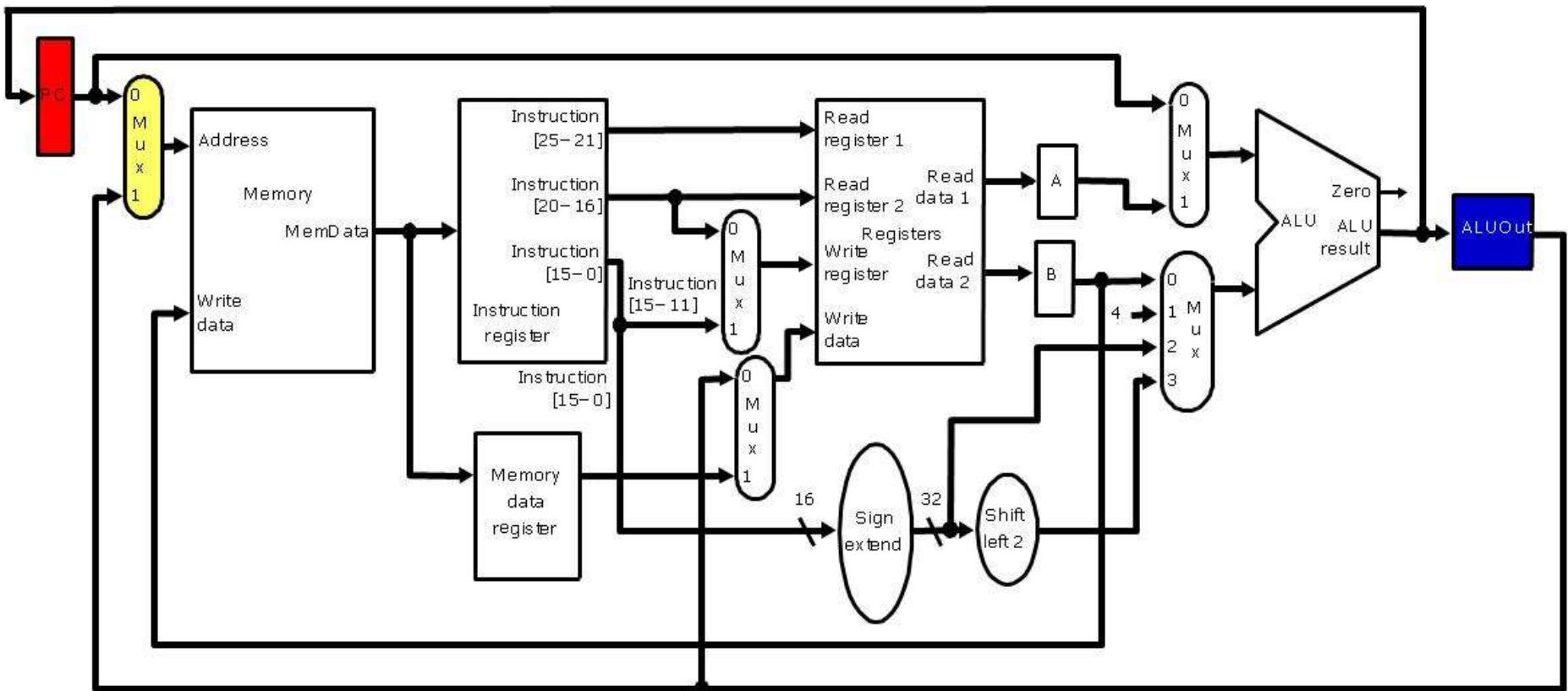


# Unidade de Processamento Multiciclo



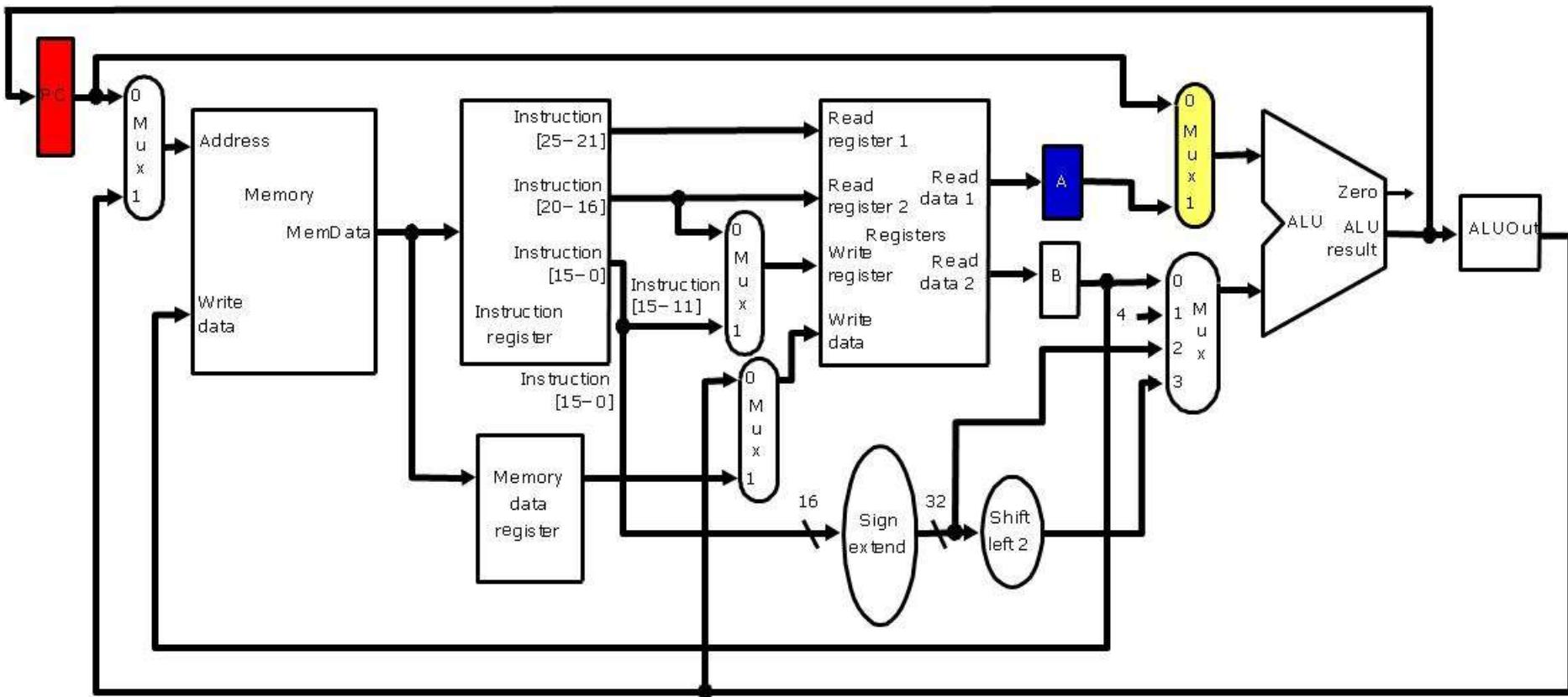
- Compartilhamento de recursos
  - Mais multiplexadores ou multiplexadores expandidos

# Entrada de Endereço da Memória



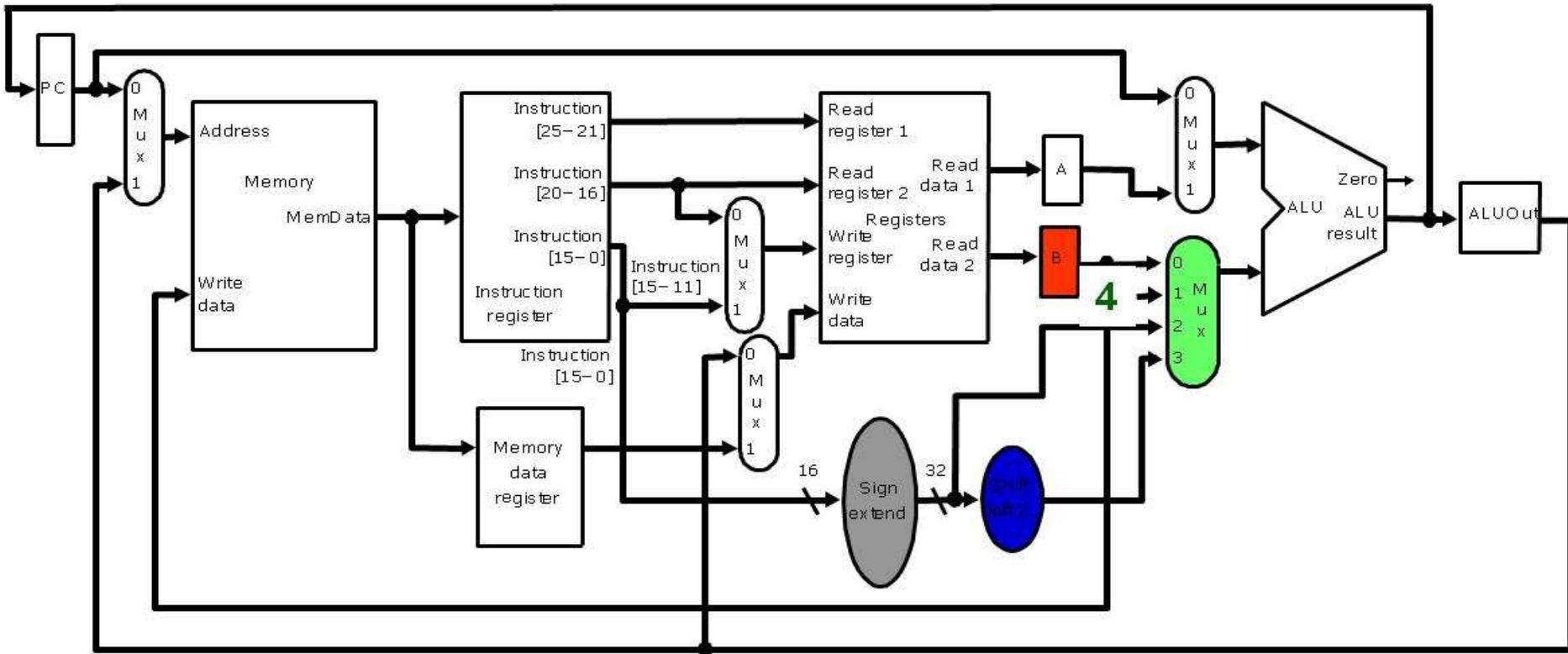
- Multiplexador para escolha entre endereço de instrução (PC) ou endereço de dado (saída da ALU - ALUOut)

# Primeiro Operando da ALU



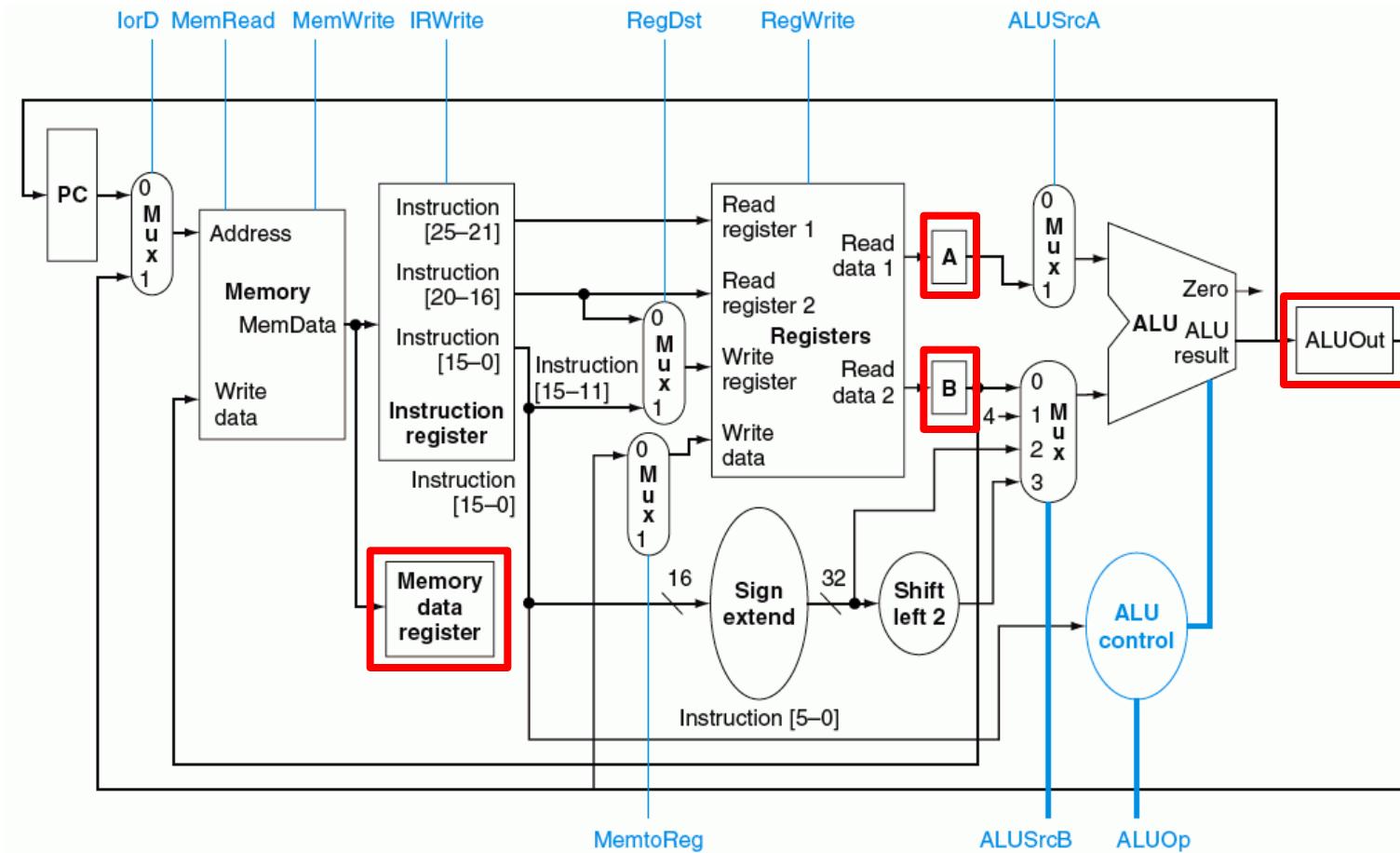
- Multiplexador para escolha entre **PC** ou operando da saída do banco de registradores (**registraror A**)

# Segundo Operando da ALU



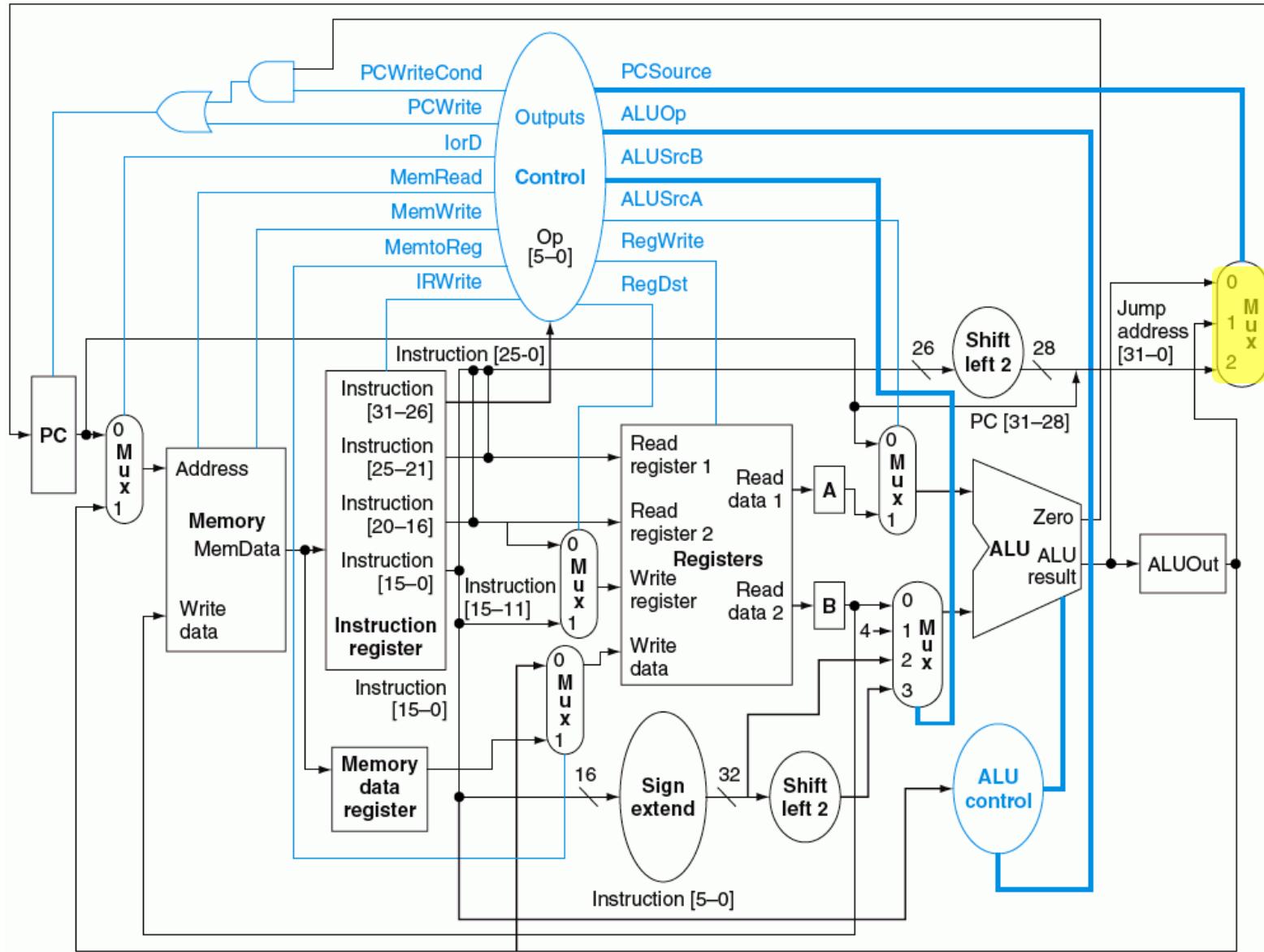
- Multiplexador expandido para escolha entre saída do banco de registradores (B), a constante 4 (para incrementar PC), o sinal estendido e deslocado de 2 bits (para beq) e o sinal simplesmente estendido (para lw/sw)

# Unidade de Processamento com Sinais de Controle

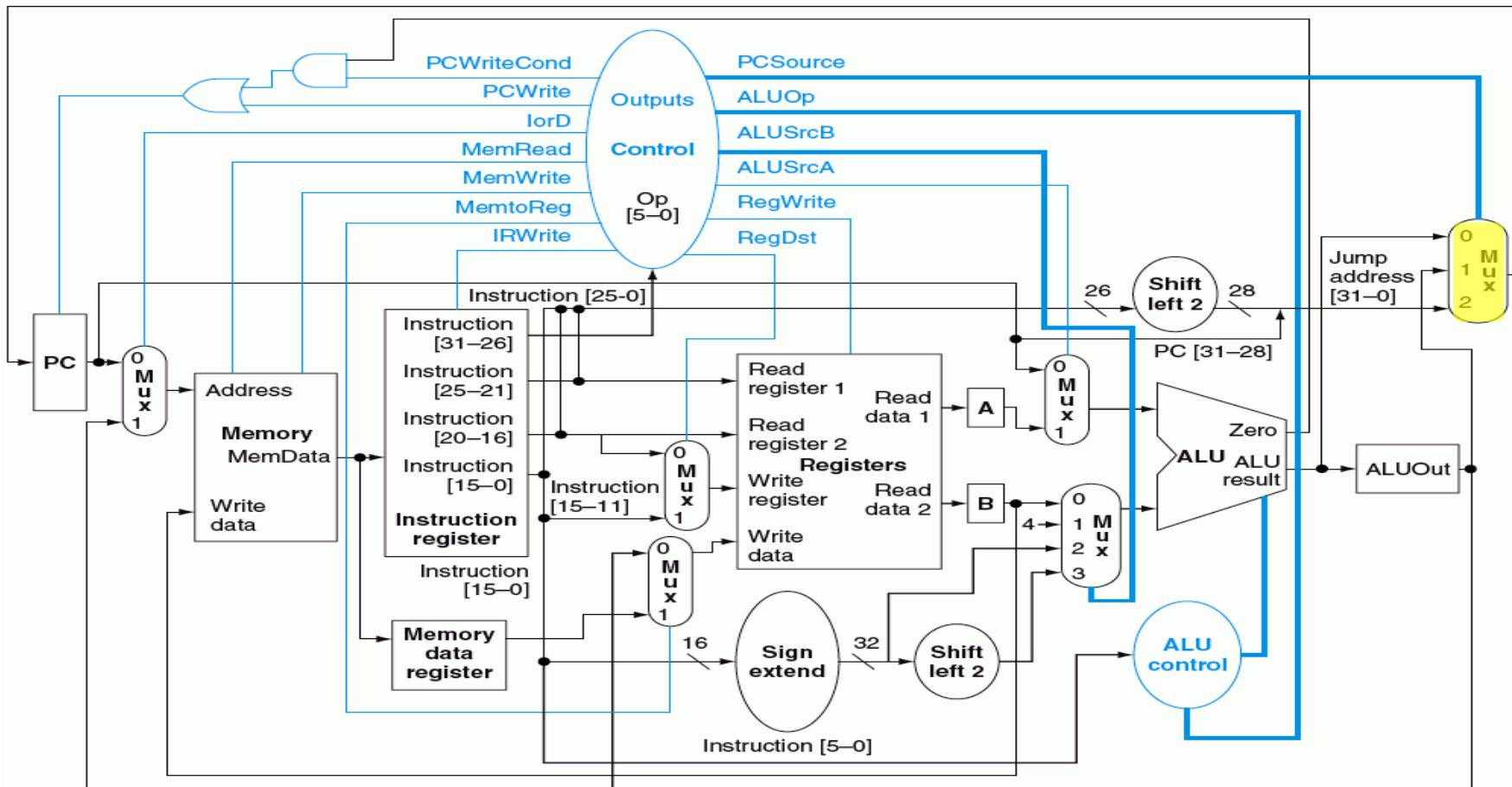


Atualizados em todo ciclo de clock, não precisam de sinal de controle de escrita

# Unidade de Processamento com Controle



# Atualização do PC



- Multiplexador para escolha entre 3 fontes de atualização: PC + 4, jump, branch

# Sinais de 1 bit da Unidade de Controle

Nome	Efeito quando 0	Efeito quando 1
RegDst	Número do registrador destino para escrita vem de rt	Número do registrador destino para escrita vem de rd
RegWrite	Nada	Registrador da entrada “Write register” recebe valor da entrada “Write data”
ALUSrcA	1º operando da ALU vem do PC	1º operando da ALU vem do registrador A
IRWrite	Nada	Saída da memória é escrita em IR
MemRead	Nada	Dado da memória relativo ao endereço especificado é colocado na saída
MemWrite	Nada	Dado da memória do endereço especificado é substituído pelo que tem no “Write data”
MemTo Reg	Valor escrito na entrada “Write data” dos registradores vem da ALUOut	Valor escrito na entrada “Write data” vem de MDR
PCWrite	Nada	Escrita em PC, controlado por PCSource
PCWriteCond	Nada	Escrita em PC se saída Zero de ALU ativo
IorD	PC fornece endereço para a memória	ALUOut fornece endereço para a memória

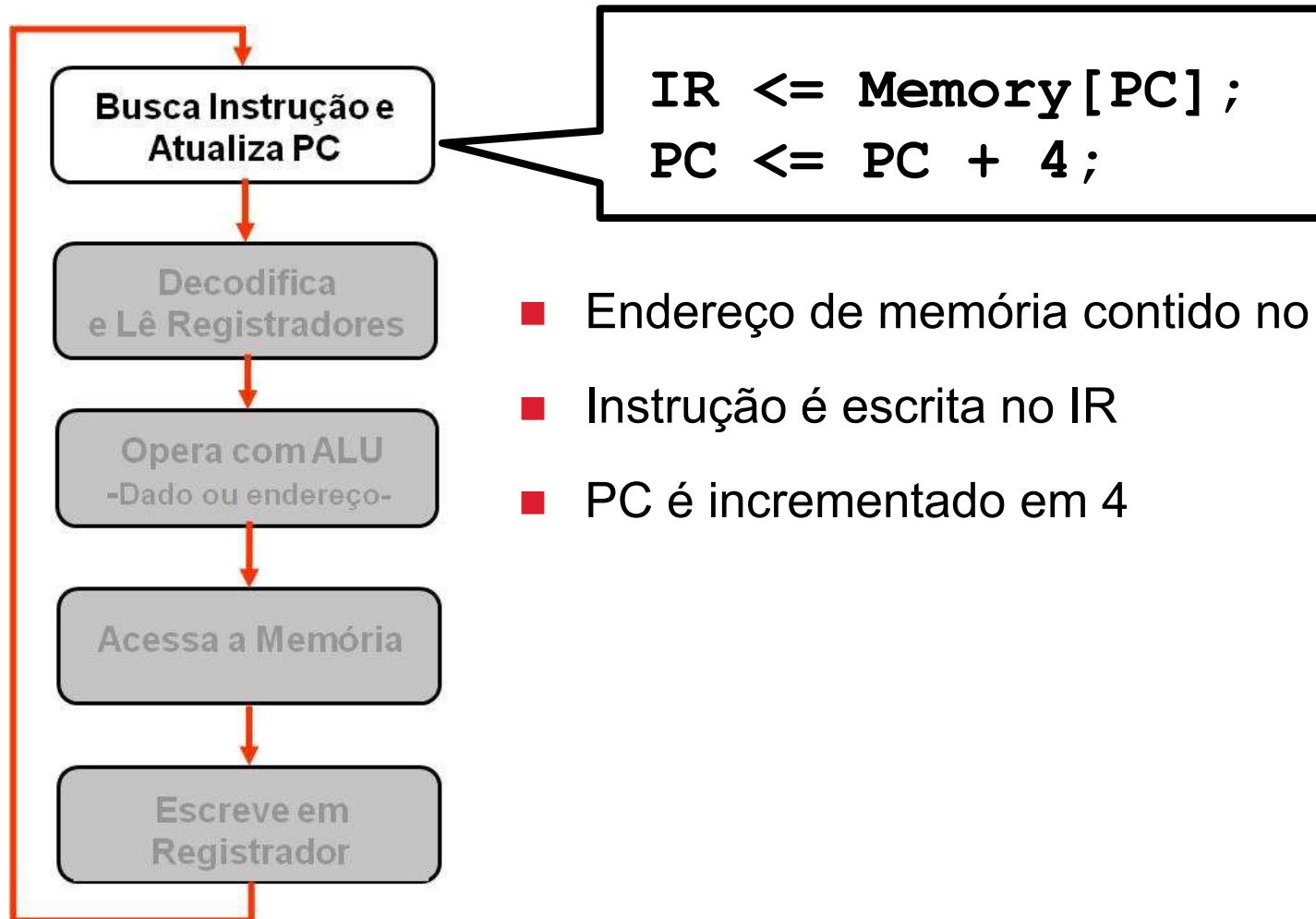
# Sinais de 2 bits da Unidade de Controle

Nome	Valor	Efeito
ALUOp	00	ALU faz soma
	01	ALU faz subtração
	10	Campo funct determina operação
ALUSrcB	00	2º Operando da ALU vem de B
	01	2º Operando da ALU é a constante 4
	10	2º Operando é o sinal estendido, 16 bits menos significantes de IR
	11	2º Operando é o sinal estendido, 16 bits menos significantes de IR, deslocados de 2 bits para a esquerda
PCSource	00	Saída de ALU (PC + 4) é enviado ao PC para escrita
	01	ALUOut (endereço destino do branch) é enviado ao PC para escrita
	10	Endereço destino do jump (26 bits menos significativos do IR deslocados de 2 bits para a esquerda concatenados com PC+4[31:28]) é enviado ao PC para escrita

# Quebrando Processamento de Instrução em Ciclos de Clock

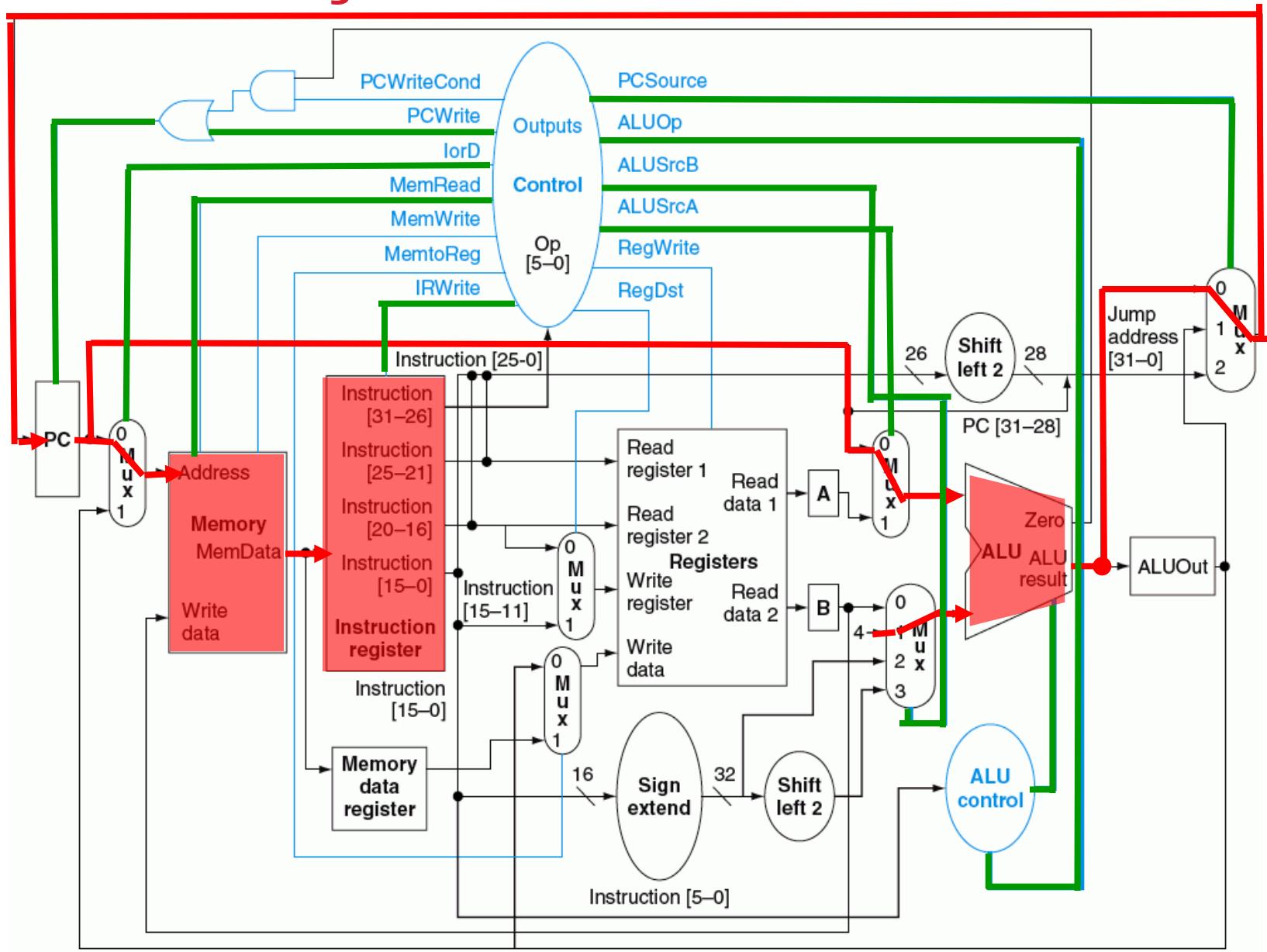


# Busca Instrução e Atualiza PC

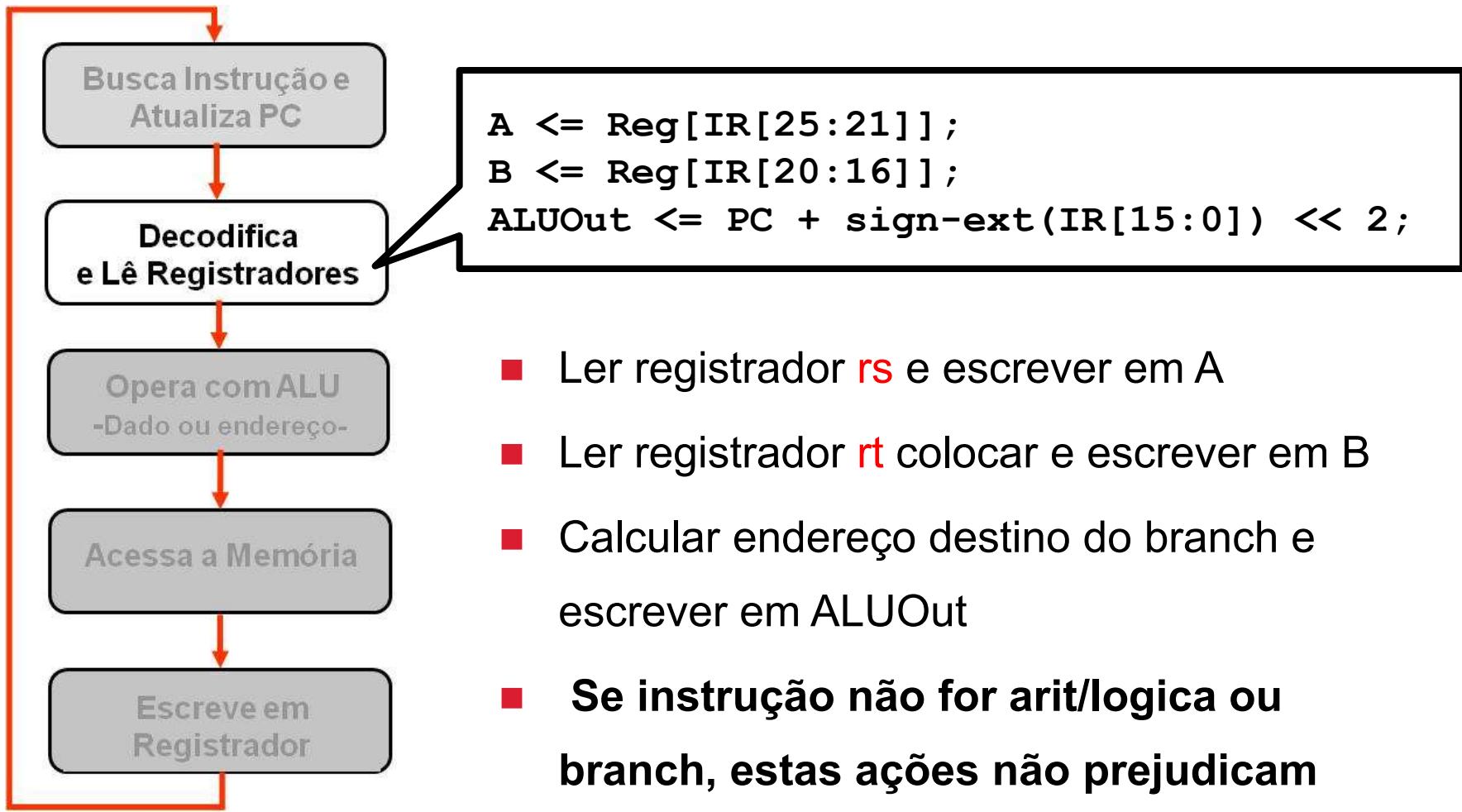


- Endereço de memória contido no PC é lido
- Instrução é escrita no IR
- PC é incrementado em 4

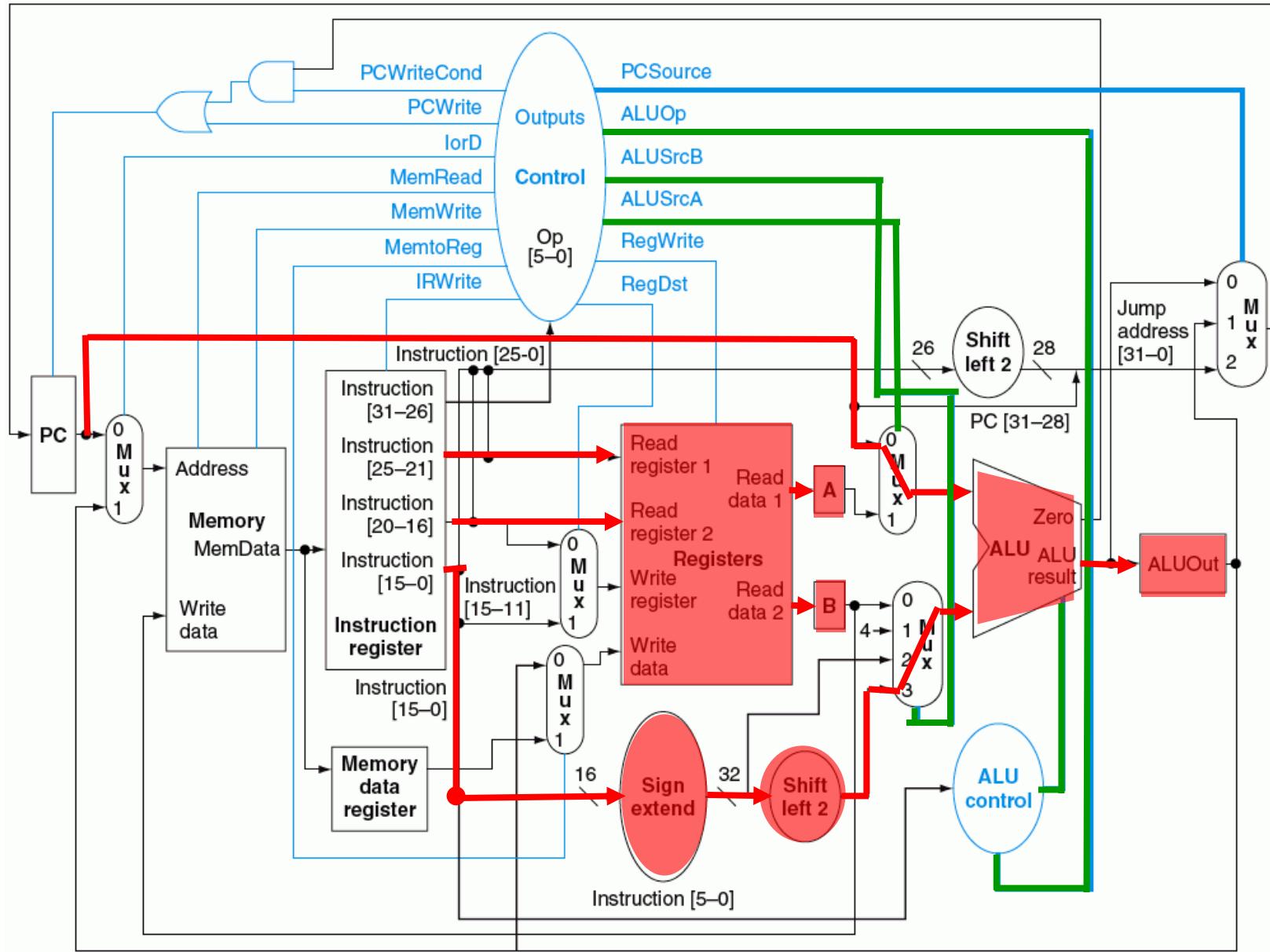
# Busca Instrução e Atualiza de PC



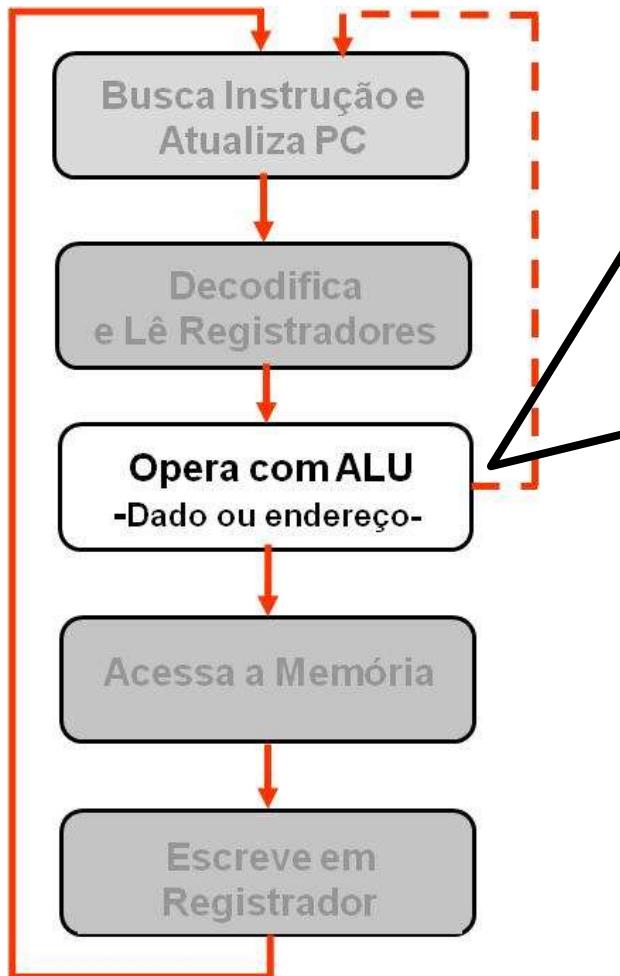
# Decodifica e Lê Registradores



# Decodifica e Lê Instruções



# Opera com a ALU



```

//Referência à memória
ALUOut <= A + sign-ext(IR[15:0]);

OU

//Operação Arit/Lógica
ALUOut <= A op B;

OU

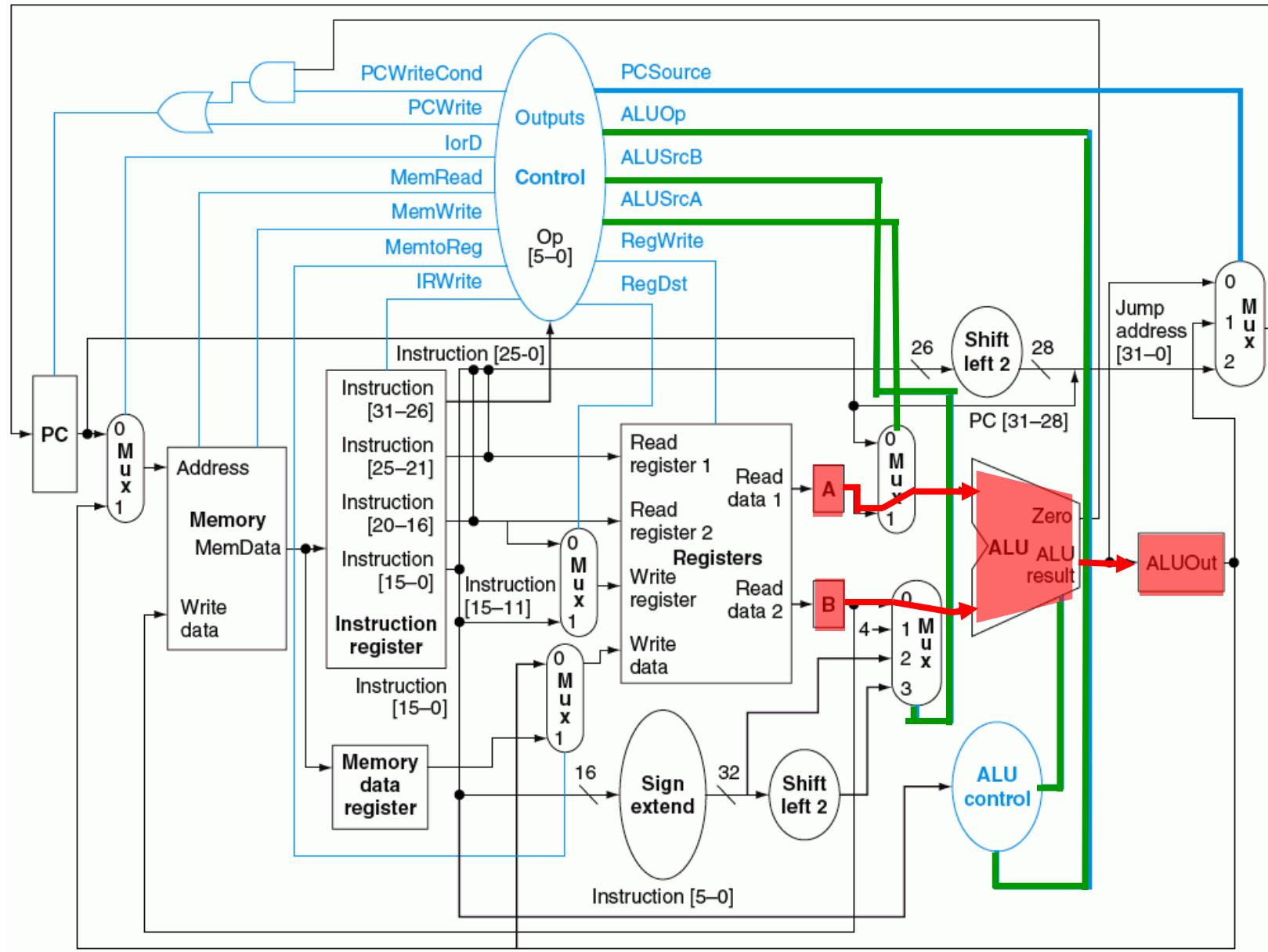
//Branch
if (A == B) PC <= ALUOut;

OU

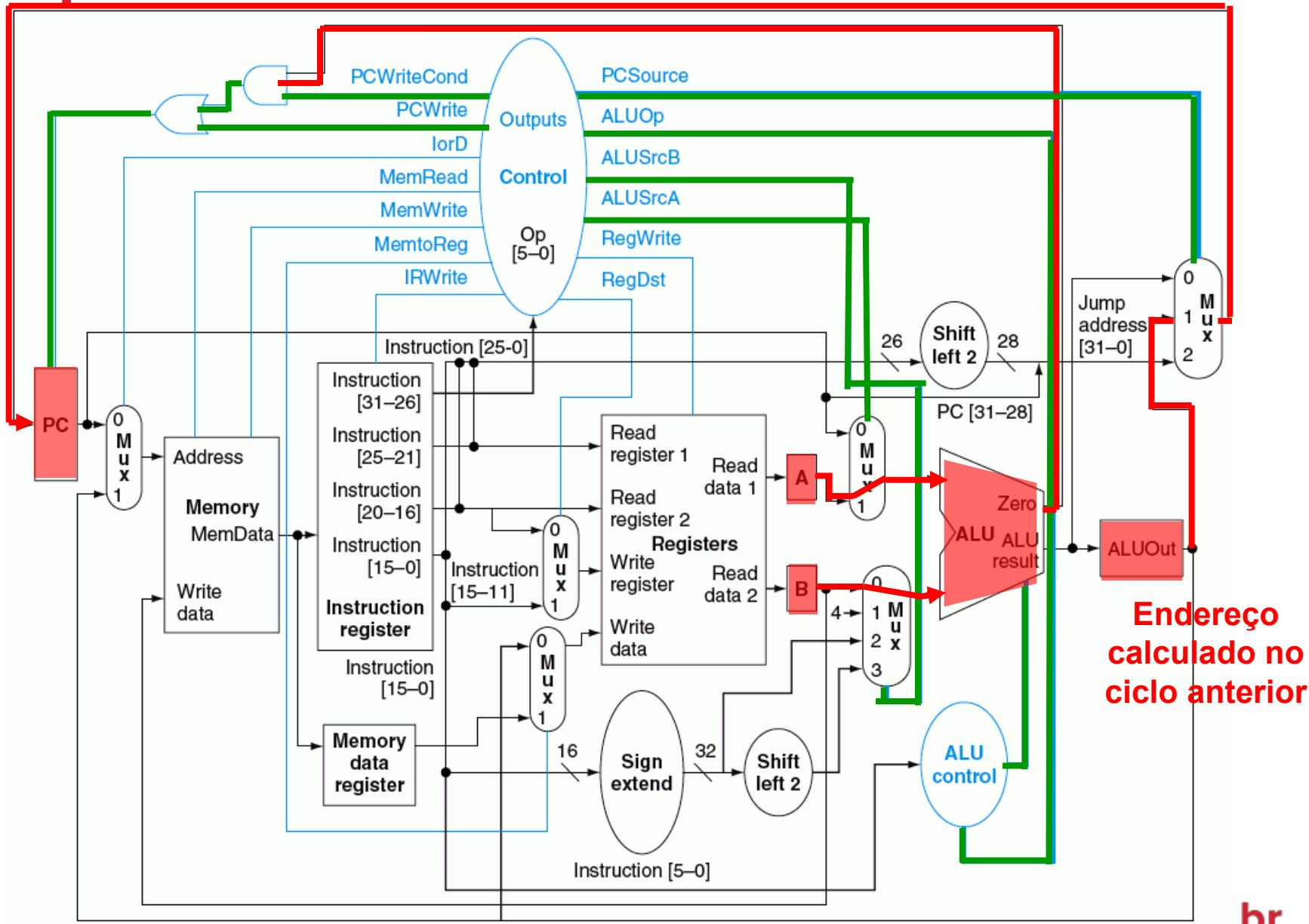
//Jump
PC <= conc(PC[31:28], IR[25:0] << 2)
  
```

- Lê registradores A e B
- Dependendo da instrução, executa diferentes ações com ALU
- **Branch e Jump completados**

# Opera com a ALU: Instruções Aritméticas



# Opera com a ALU: Branch



# Acessa a Memória (ou Escreve em Registrador)



- Instruções arit/lógicas e SW são completadas
  - **Arit/Lógicas escrevem no registrador**
- LW: dado da memória armazenado no endereço ALUOut é carregado em MDR

*//Referência à memória LW*

 $MDR \leftarrow Memory[ALUOut];$ 

**OU**

*//Referência à memória SW*

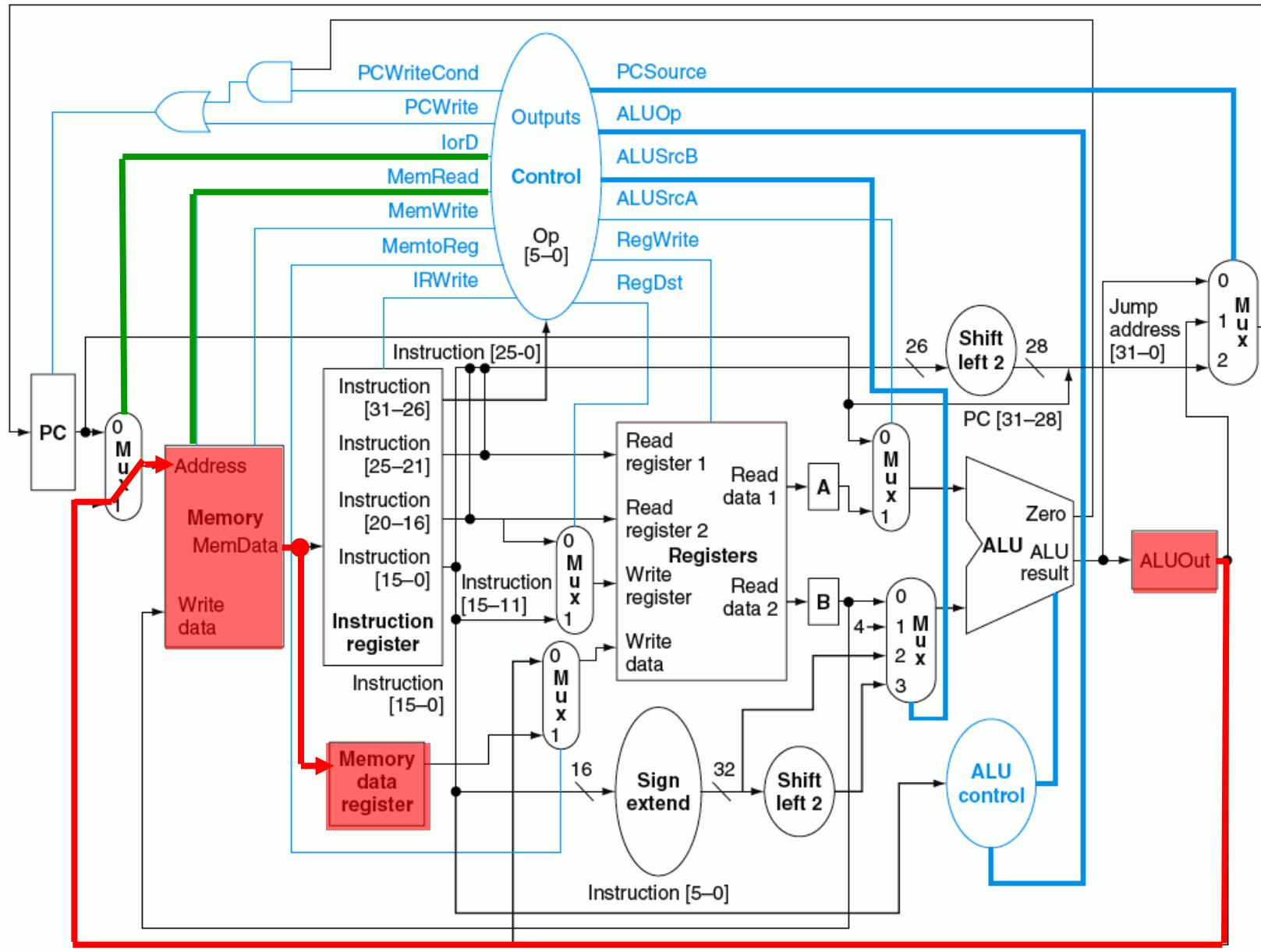
 $Memory[ALUOut] \leftarrow B;$ 

**OU**

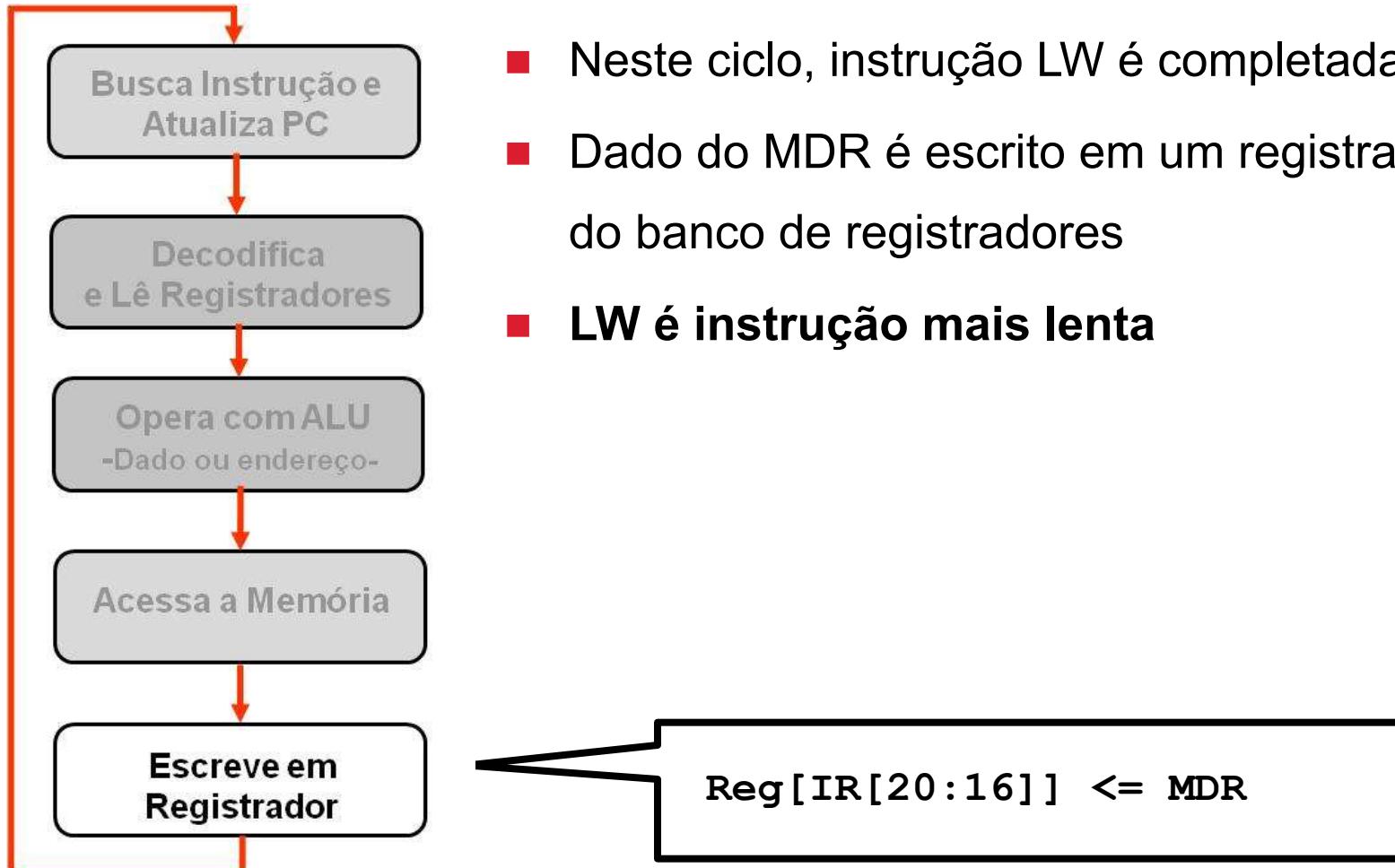
*//Operação Arit/Lógica*

 $Reg[IR[15:11]] \leftarrow ALUOut$

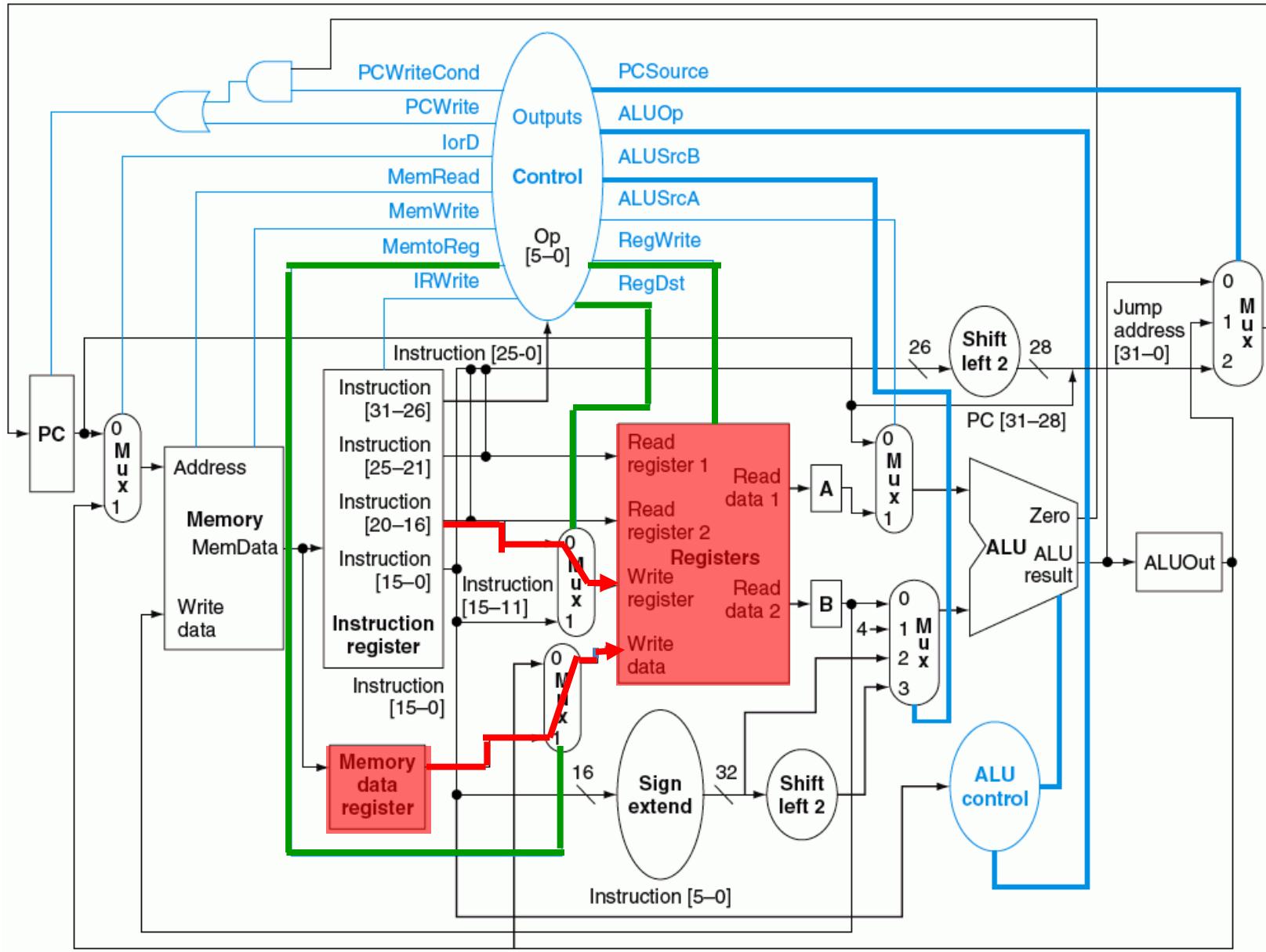
# Acessa a Memória: LW



# Escreve em Registrador



# Escreve em Registrador: LW

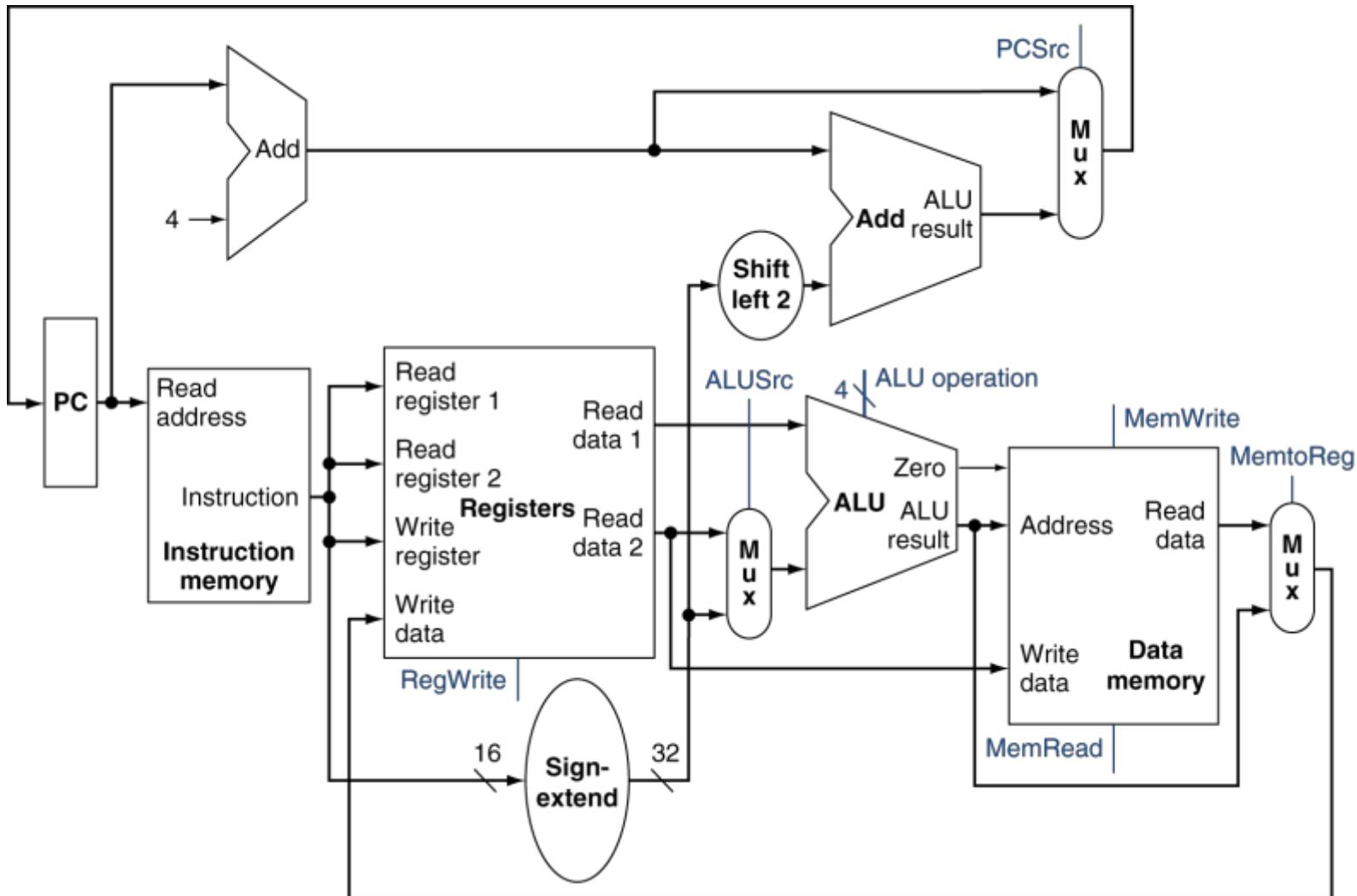


# Infraestrutura de HW

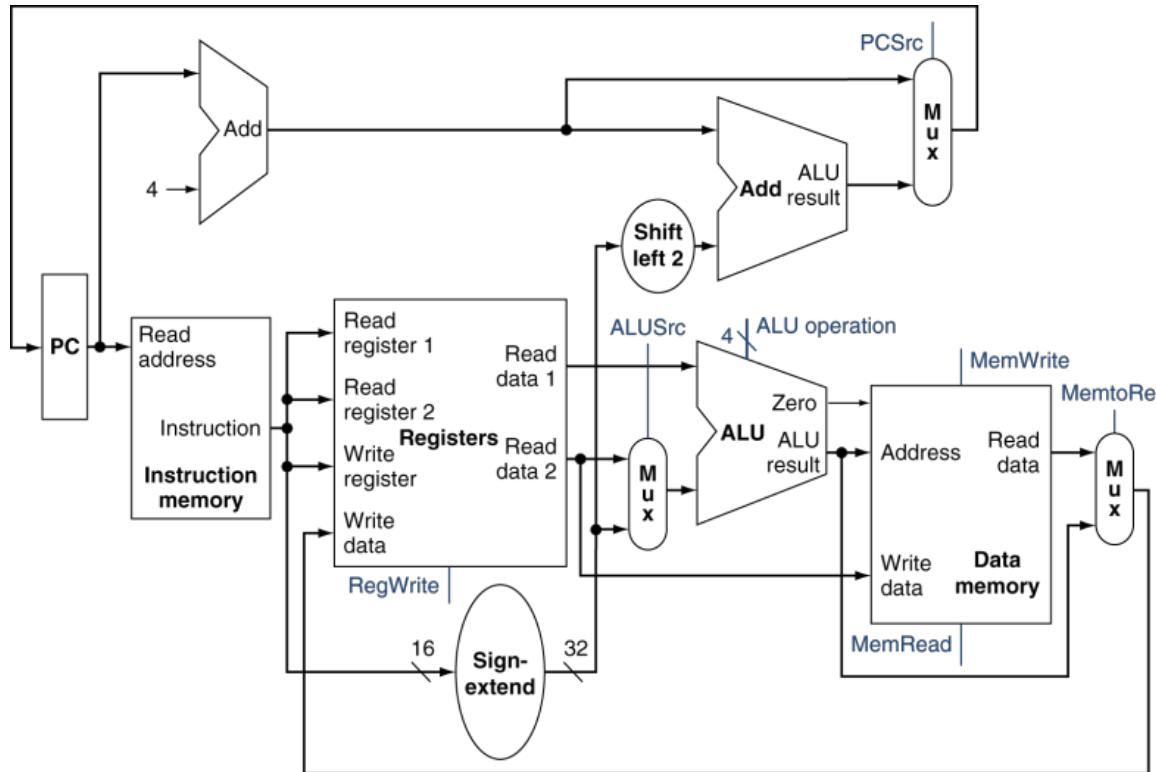
Implementação Monociclo de um Processador Simples  
(Unidade de Controle)

Prof. Adriano Sarmento

# Unidade de Processamento (Quase) Completa

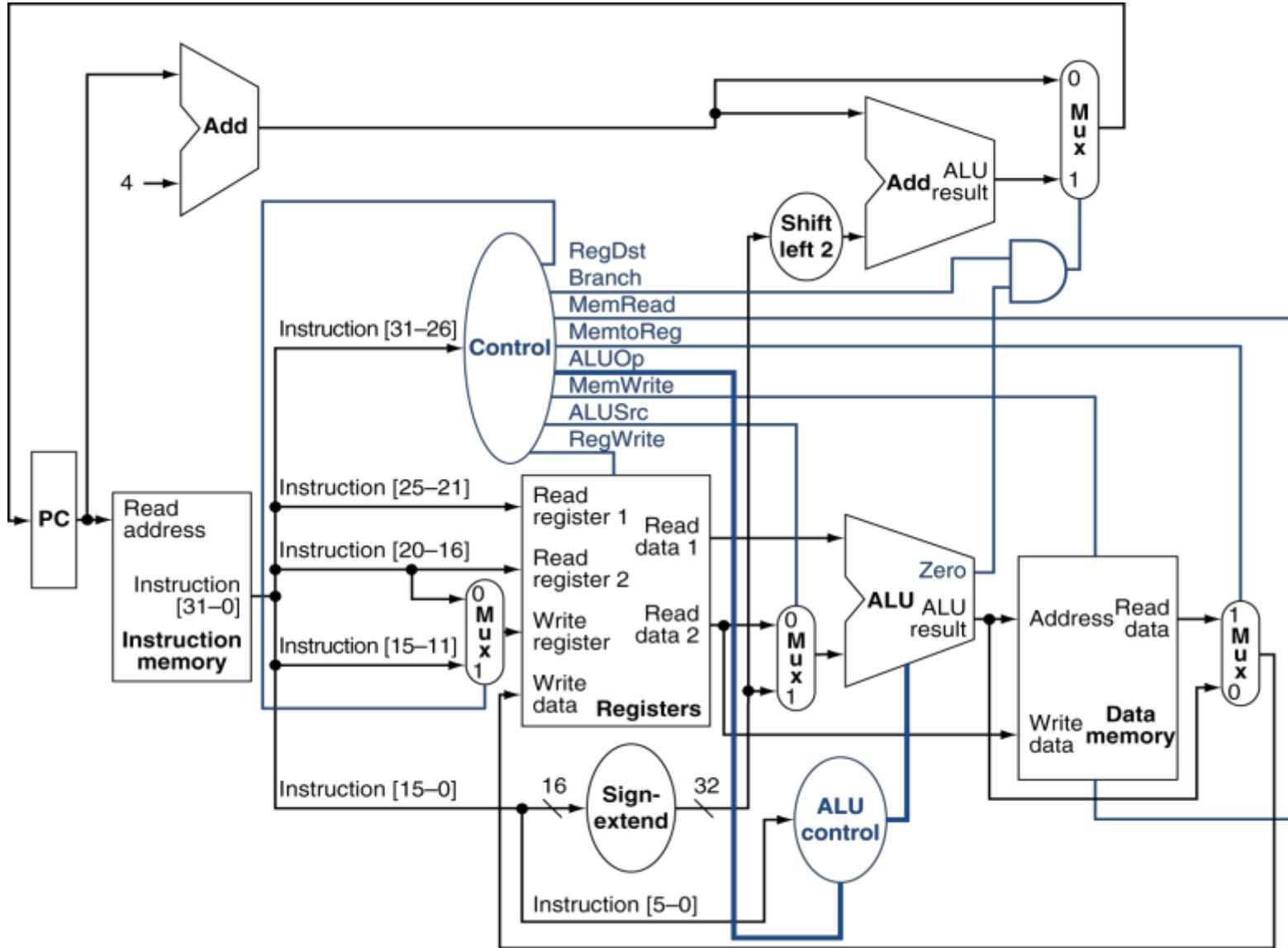


# O Que É Que Falta?



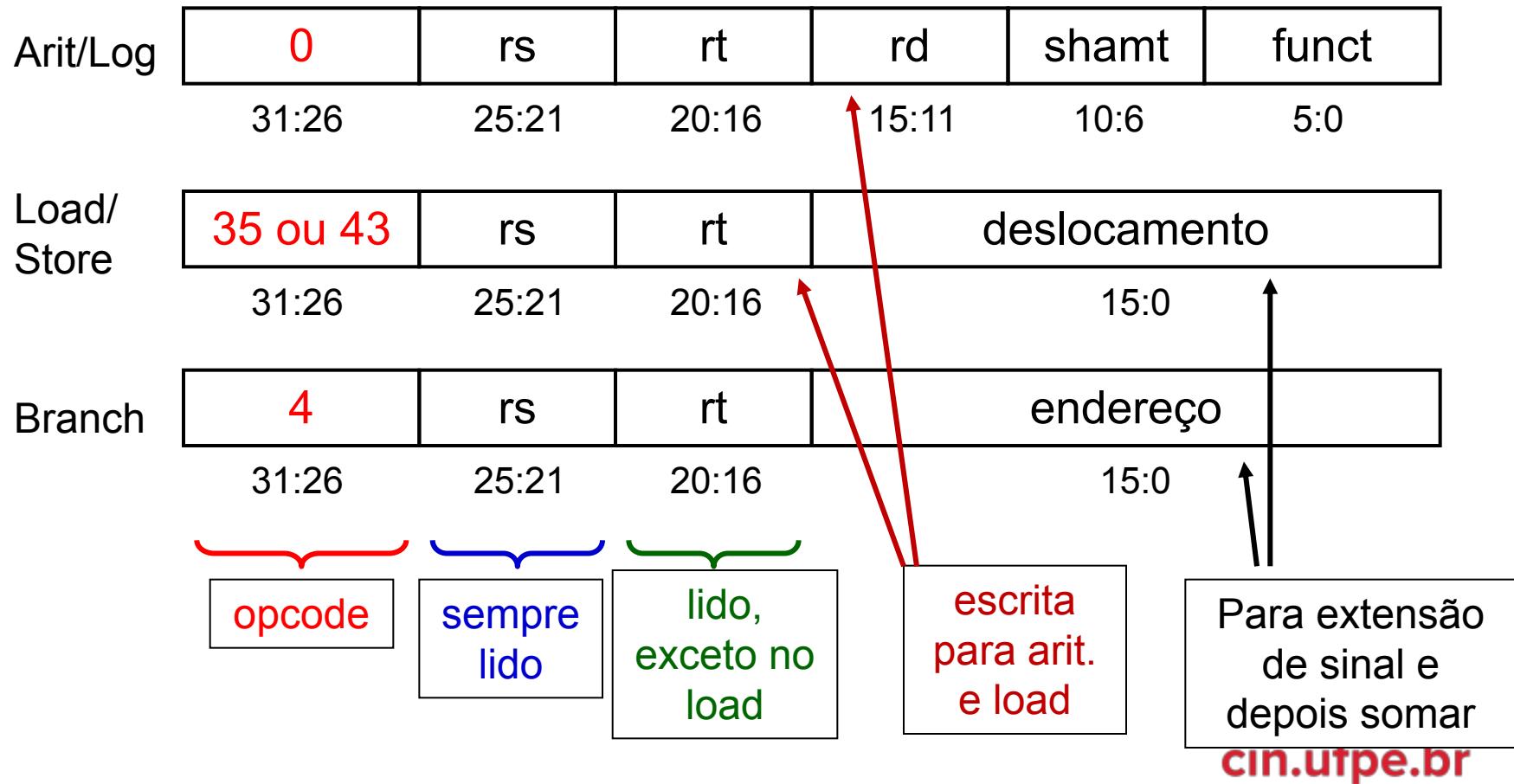
- Unidade que decodifique as instruções
  - Escolha operação da ALU, controle escrita em registrador,
  - controle leitura/escrita na memória, etc

# Unidade de Processamento Com Controle



# Unidade de Controle Principal

- Sinais de controle são derivados da instrução
  - Decodificação da instrução



# Sinais da Unidade de Controle

- Unidade de controle tem como entrada a instrução e como saída 8 sinais + ALUOp

Nome	Efeito quando 0	Efeito quando 1
RegDst	Número do registrador destino para escrita vem de rt	Número do registrador destino para escrita vem de rd
RegWrite	Nada	Registrador da entrada “Write register” recebe valor da entrada “Write data”
ALUSrc	2º operando da ALU vem da 2a saída do banco de registradores	2º operando da ALU vem do valor estendido contido na instrução
Branch	PC recebe valor de PC + 4 que vem do somador	PC recebe valor do endereço do branch se operandos =
MemRead	Nada	Dado da memória relativo ao endereço especificado é colocado na saída
MemWrite	Nada	Dado da memória relativo ao endereço especificado é substituído pelo que tem no “Write data”
MemTo Reg	Valor escrito na entrada “Write data” vem da ALU	Valor escrito na entrada “Write data” vem da memória

# Controle da ALU

- ALU usada para:
  - Load/Store: Função = soma
  - Branch: Função = subtração
  - Aritmética/Lógica: depende do campo funct

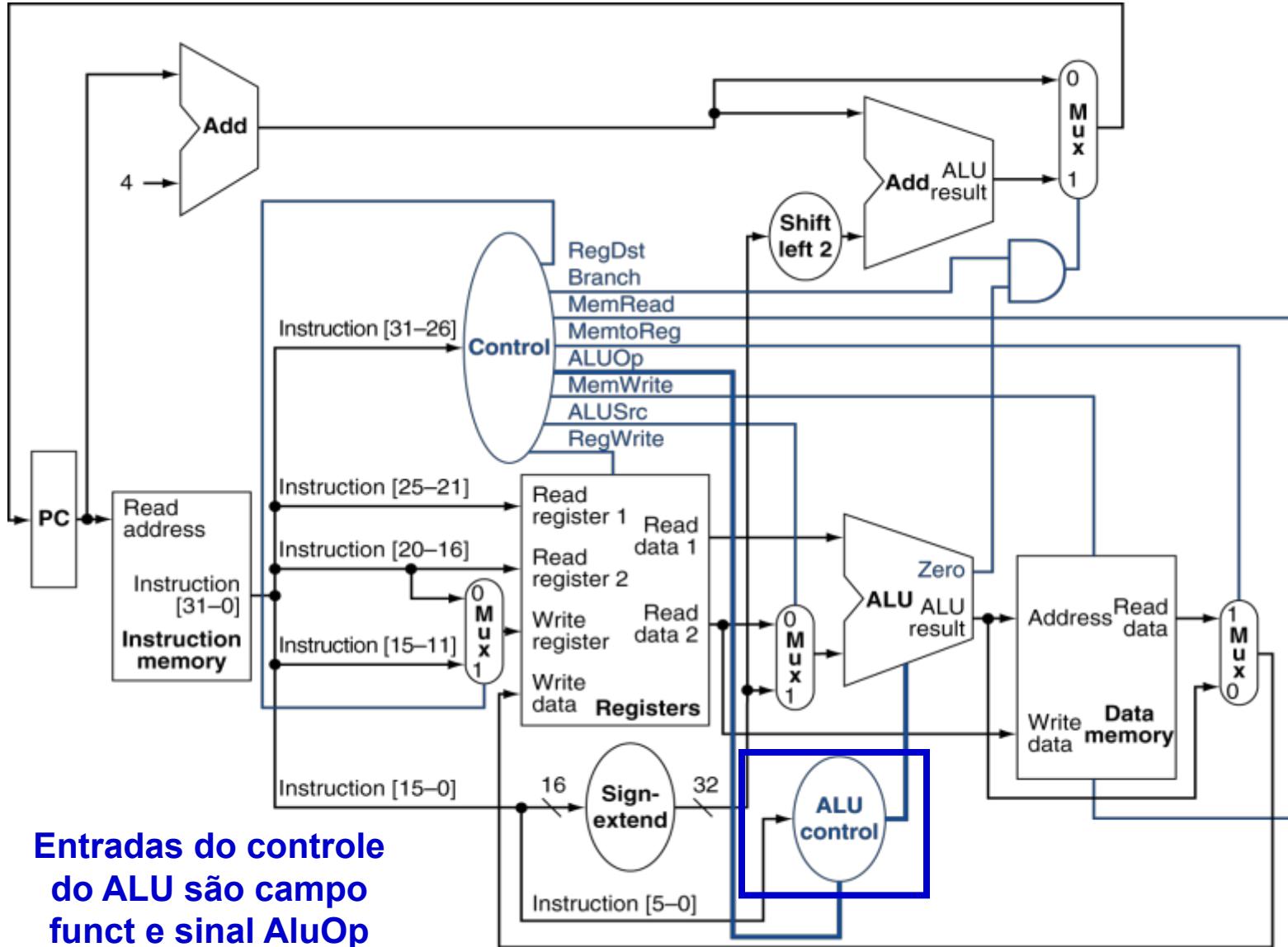
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

# Controle da ALU

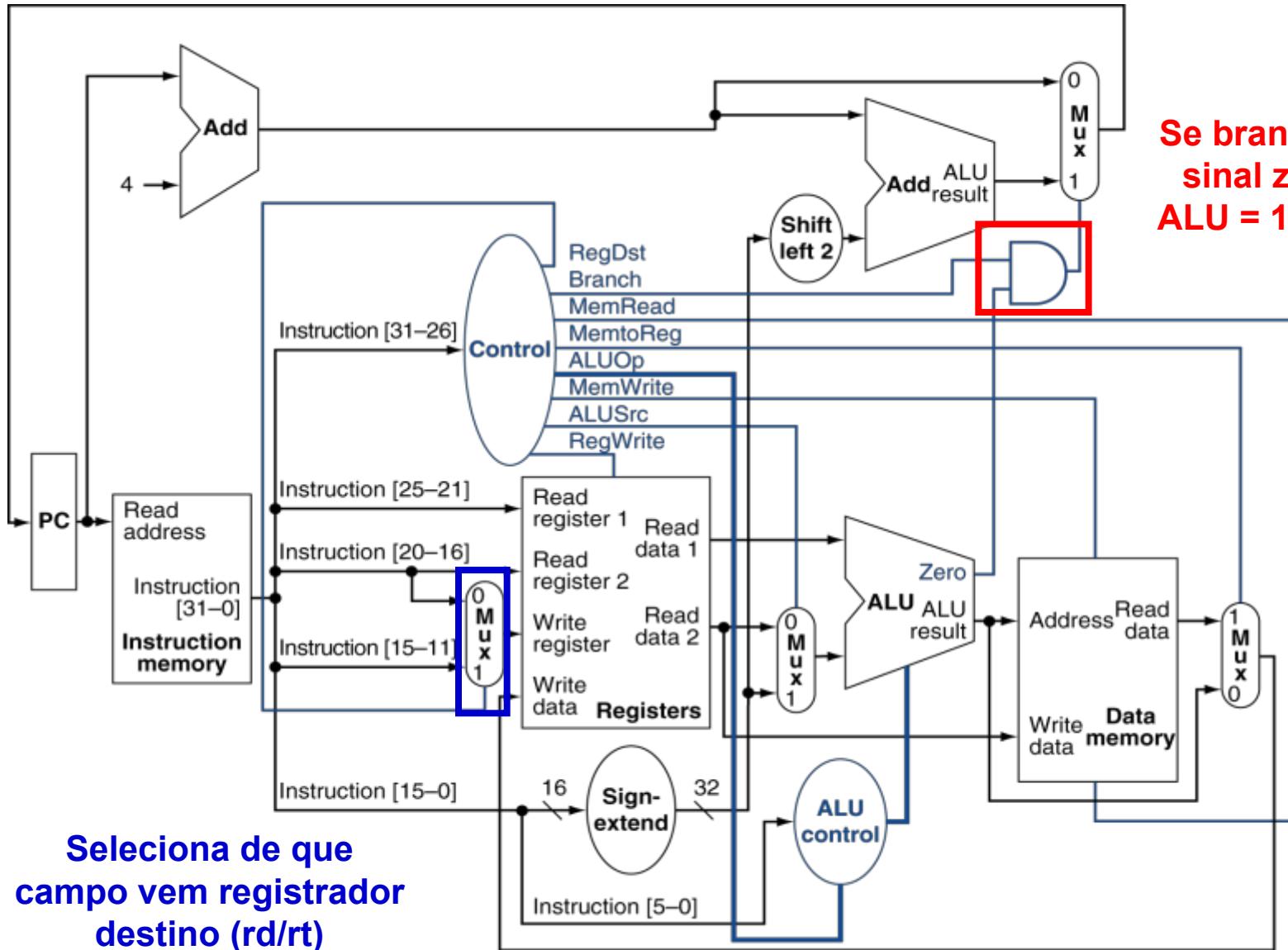
- Podemos derivar do opcode um sinal de 2 bits chamado ALUOp
  - Lógica combinacional deriva o controle da ALU

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
Beq	01	branch equal	XXXXXX	subtract	0110
Aritmética/ lógica	10	add	100000	add	0010
		subtract	100010	Subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

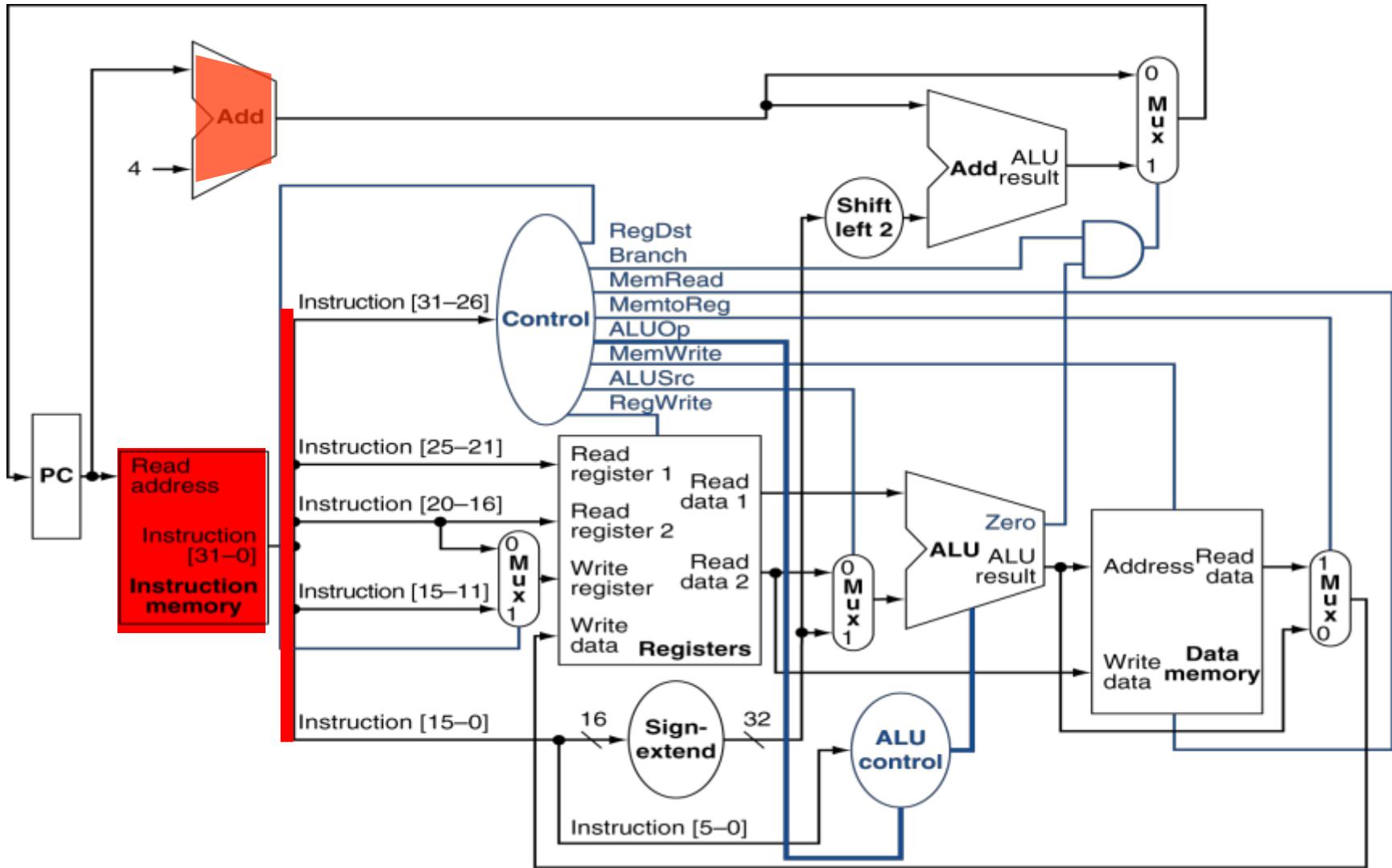
# Unidade de Processamento Com Controle



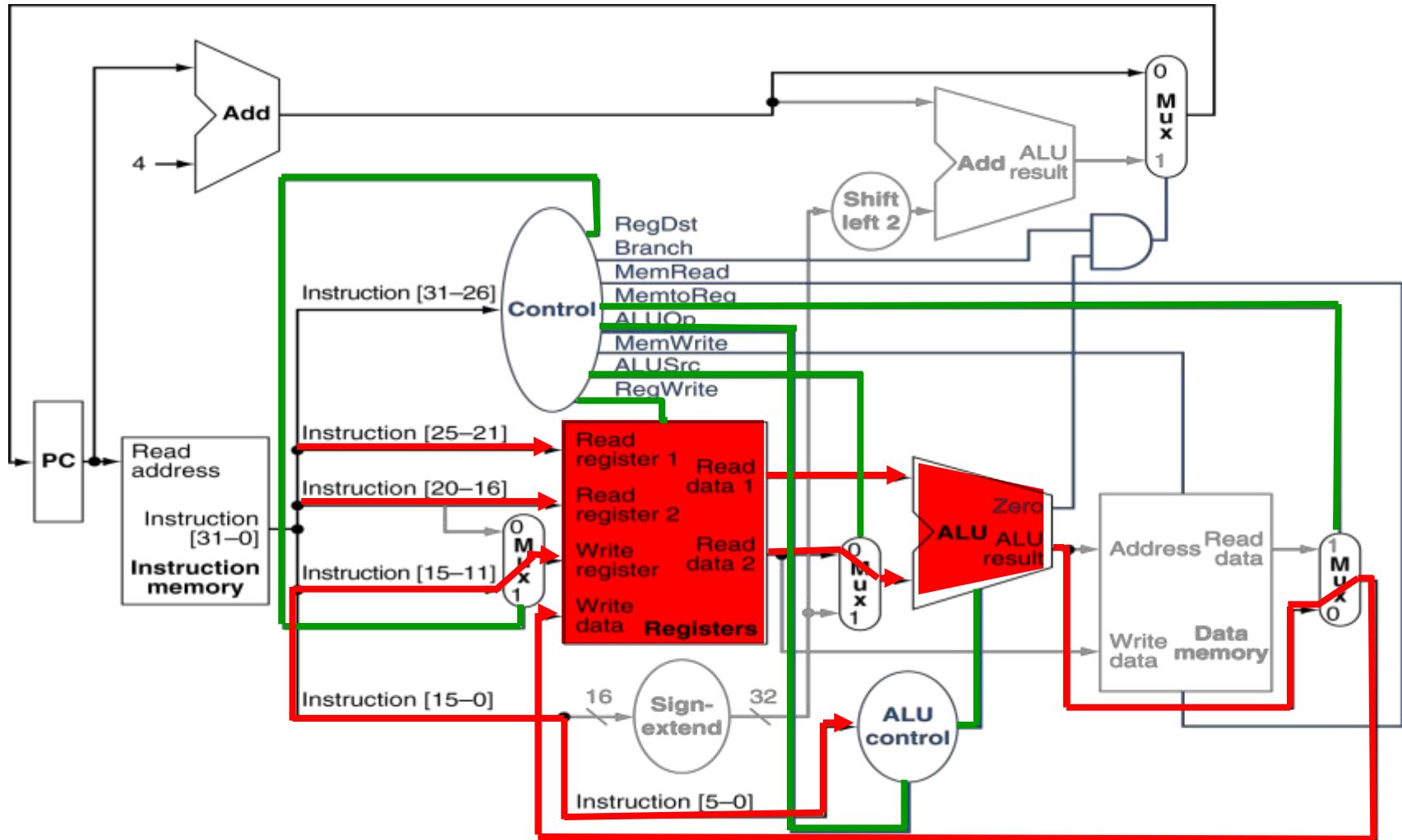
# Unidade de Processamento Com Controle



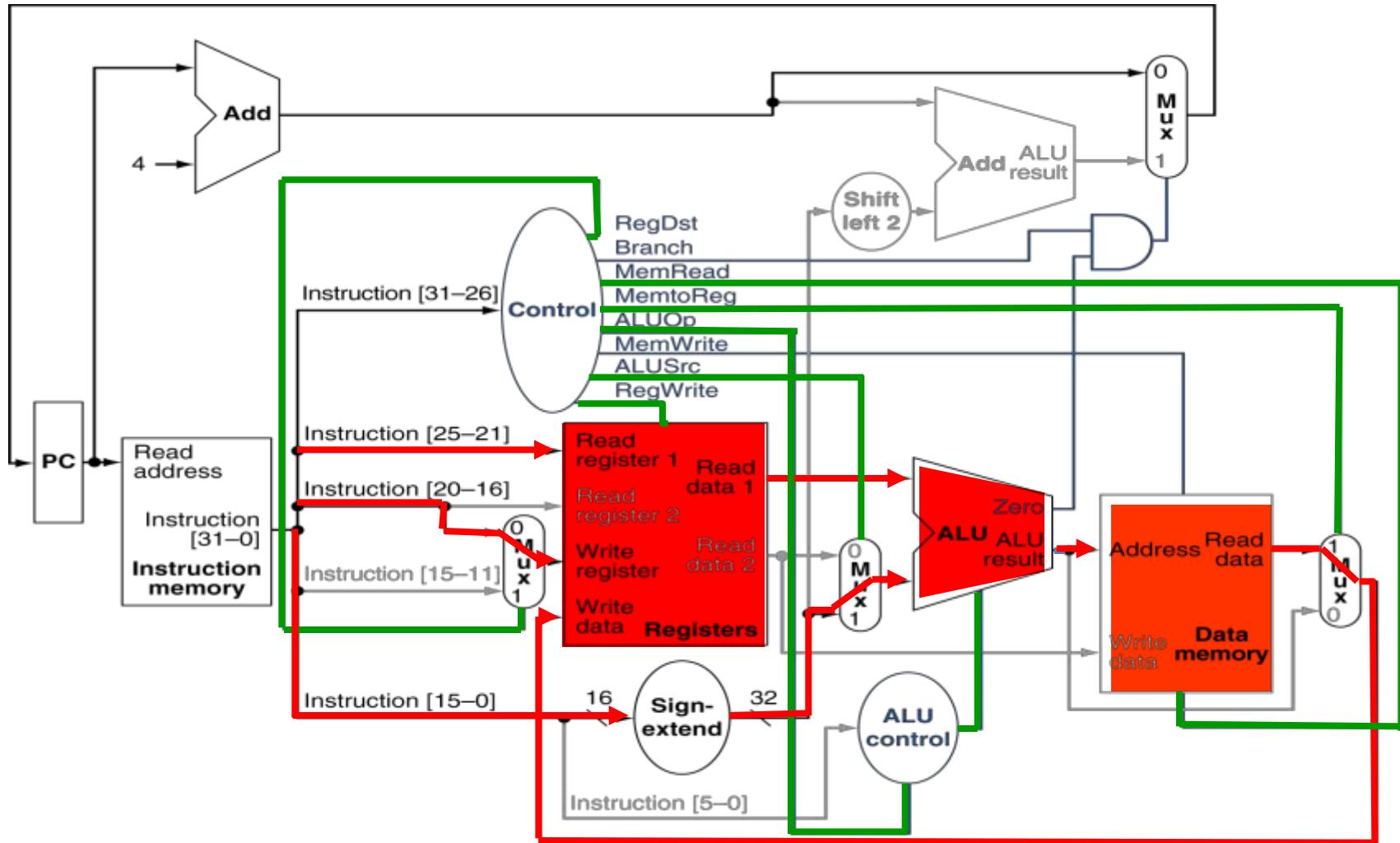
# Busca de Instrução



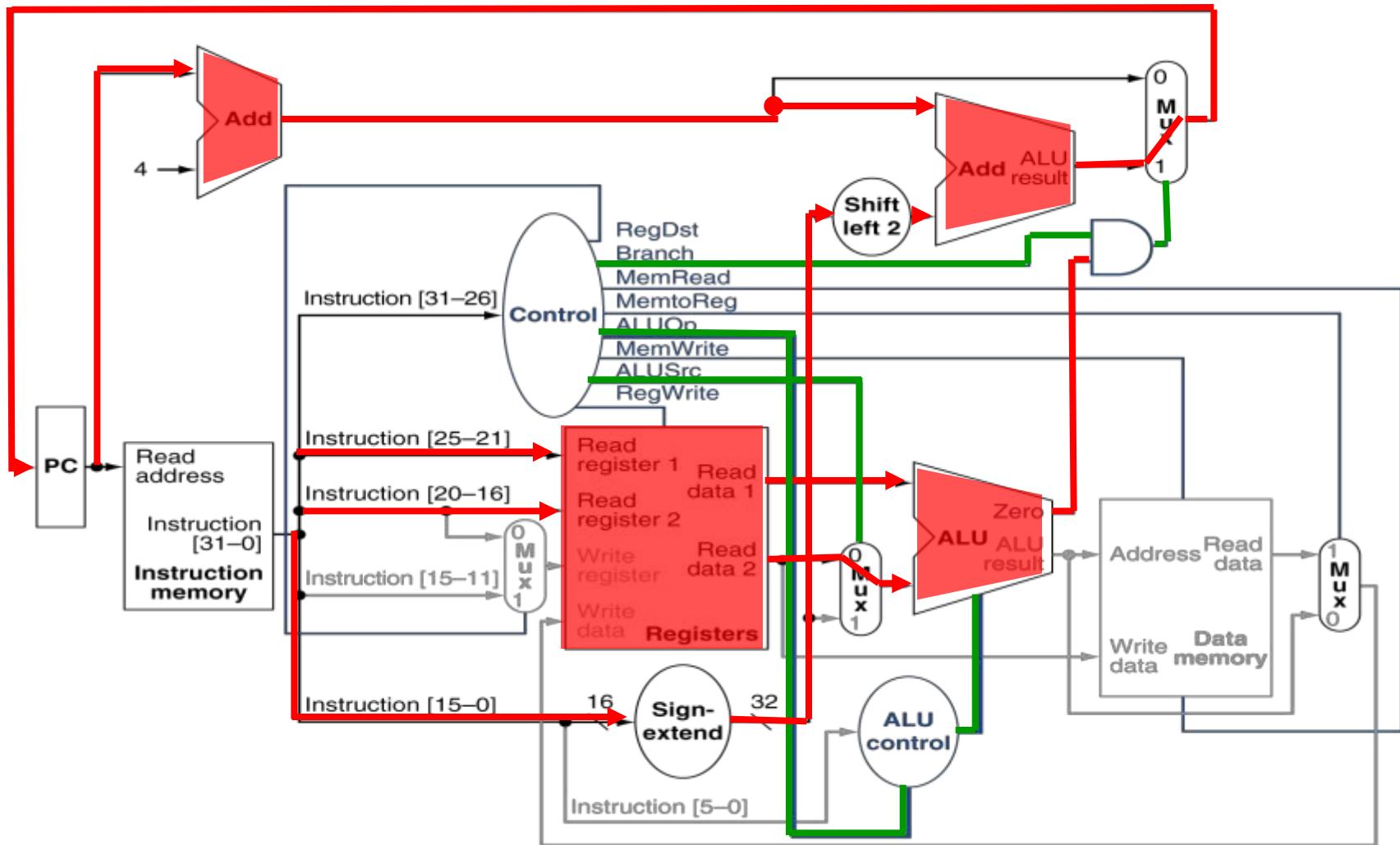
# Instruções Aritméticas/Lógicas



# Instrução LW



# Instrução BEQ



# Infraestrutura de HW

Implementação Monociclo de um Processador Simples

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

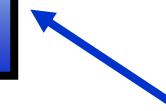
- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Interface HW/SW: ISA

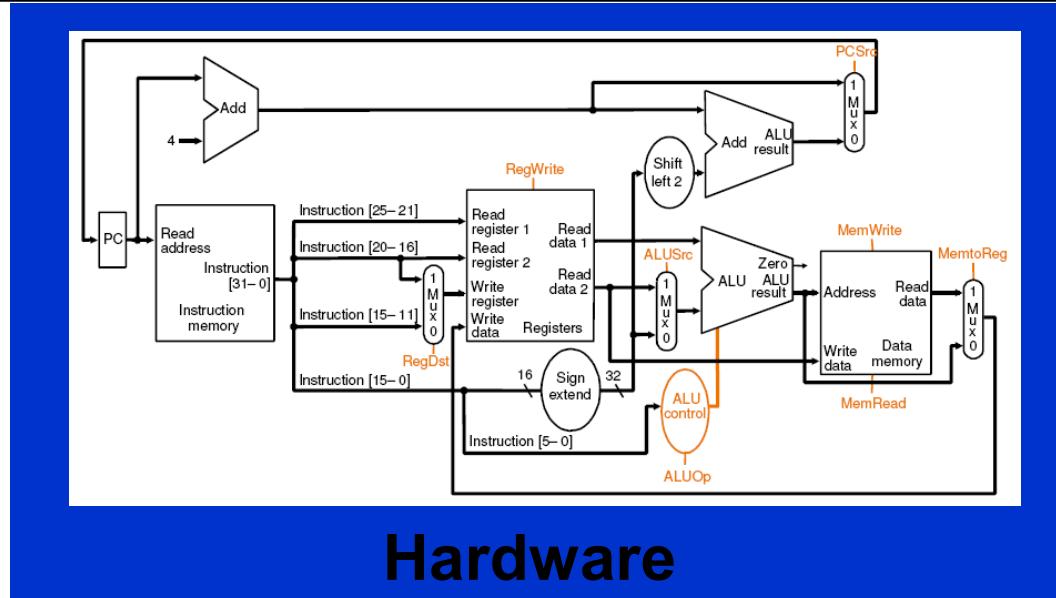
Software



ISA



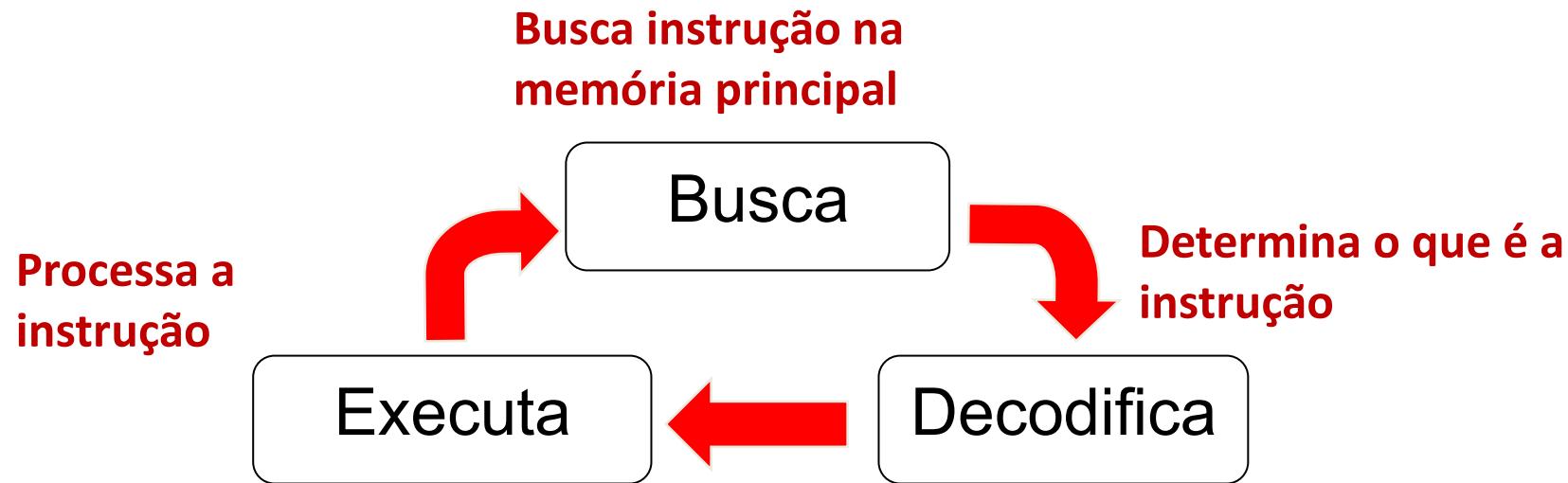
Repertório de  
Instruções da  
Arquitetura



Hardware

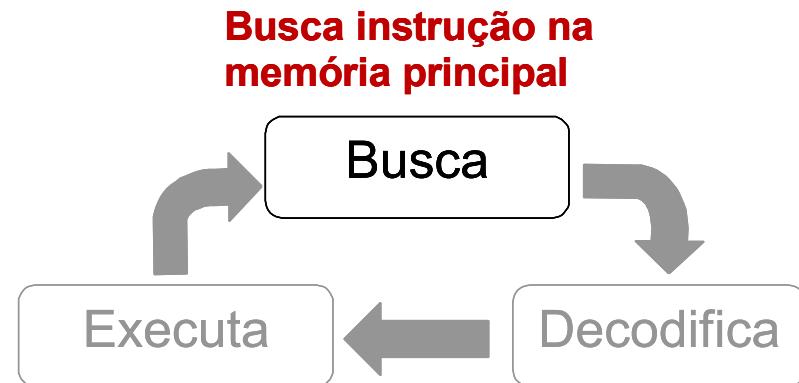
# Visão Simplificada de Processamento de Instrução

- CPU faz continuamente 3 ações:



# Buscando Instrução

- CPU deve saber qual é a instrução que deve ser buscada
  - Instrução na sequência?
  - Instrução especificada em um desvio?
- CPU deve calcular endereço da próxima instrução a ser executada



# Decodificando Instrução

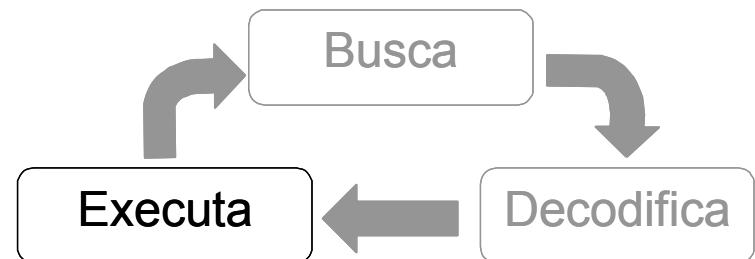
- CPU deve determinar o que é a instrução
  - Diferentes formatos de instrução
  - Como se identifica diferentes formatos?
  - Como se identifica diferentes instruções com mesmo formato?
  - Como se obtém os operandos?

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>sa</b>	<b>funct</b>	R format
<b>op</b>	<b>rs</b>	<b>rt</b>	<b>immediate</b>			I format
<b>op</b>	<b>jump target</b>					J format



# Executando Instrução

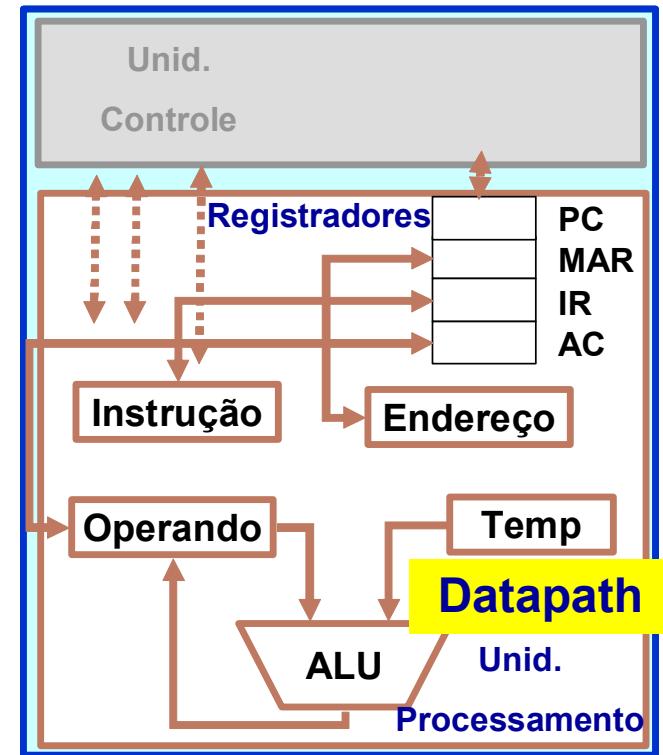
- CPU deve seguir o que a instrução manda
  - Diferentes tipos de instruções
    - Aritméticas/Lógicas
    - Acesso à memória
    - Desvios
  - Cálculos com dados e endereços
  - Acesso a diferentes componentes
    - Registradores, ALU, Memória ...
  - Coordenação da interação entre diferentes componentes



# Partes de uma CPU

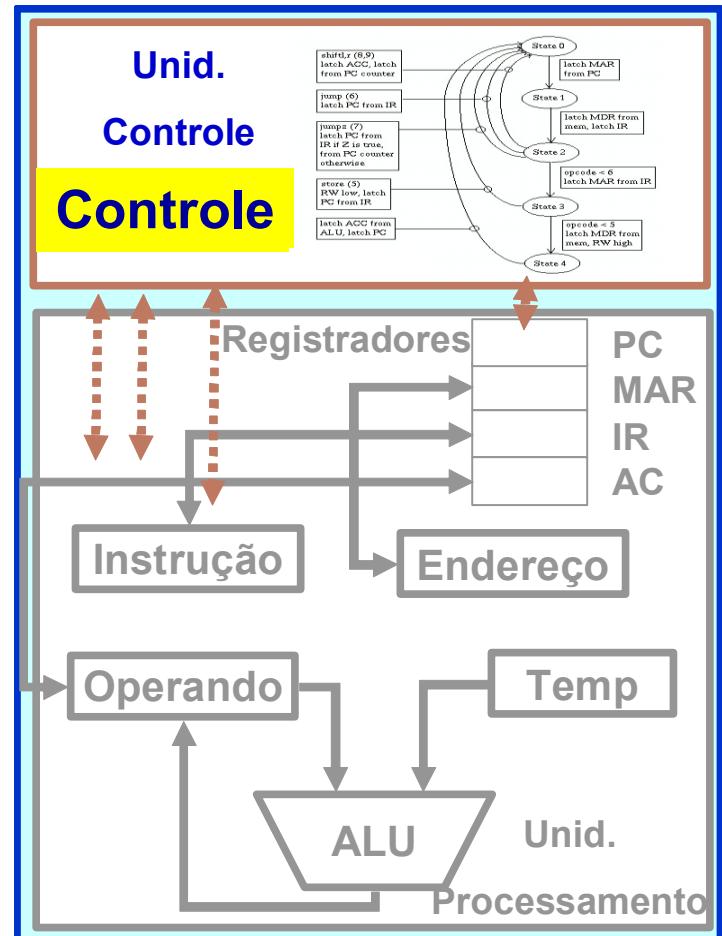
## ■ Datapath

- **Unidade de Processamento da instrução**
- Caminho onde os dados/endereços trafegam
- Composto por unidades funcionais, que operam sobre dados/endereços, e suas interconexões
  - Processamento: ALUs, Somadores, Multiplexadores ...
  - Armazenamento: Registradores e Memória



# Partes de uma CPU

- Controle
  - Unidade de Controle
  - Decodifica a instrução
  - Coordena as etapas de processamento da instrução
  - Ativa somente as unidades funcionais necessárias para execução da instrução
    - Decisão baseada na instrução



# Arquitetura de uma CPU

- Projeto de arquitetura de uma CPU requer a definição de:
  - Conjunto de registradores
  - Tipos de dados
  - Repertório de instruções
  - Formato de instruções

# Projetando uma CPU Simples

- Iremos ver como se pode projetar e implementar uma CPU simples
- CPU com um repertório de instruções composto por um subconjunto de instruções do MIPS
  - Aritméticas, lógicas, armazenamento e desvio
- Conjunto de registradores: mesmo do MIPS
- Operandos são do tipo **inteiro**
- Toda instrução deve ser executada em **um ciclo de clock** (Implementação **Monociclo**)
  - Irreal nos dias de hoje
  - Ciclo de clock deve longo o suficiente para que a instrução mais lenta respeite esta restrição

# Instruções

## Repertório

Instrução	Descrição
<b>ADD rd, rs, rt</b>	$rd \leftarrow rs + rt$
<b>SUB rd, rs, rt</b>	$rd \leftarrow rs - rt$
<b>AND rd, rs, rt</b>	$rd \leftarrow rs \text{ and } rt \text{ (bit a bit)}$
<b>OR rd, rs, rt</b>	$rd \leftarrow rs \text{ or } rt \text{ (bit a bit)}$
<b>SLT rd, rs, rt</b>	Se $rs < rt$ , $rd \leftarrow 1$ , senão $rd \leftarrow 0$
<b>LW rt, desl(rs)</b>	Carrega palavra de mem. em registrador rt
<b>SW rt, desl(rs)</b>	Armazena conteúdo de registrador rt em mem.
<b>BEQ rs, rt, end</b>	Desvio para end, se $rs == rt$

## Formato

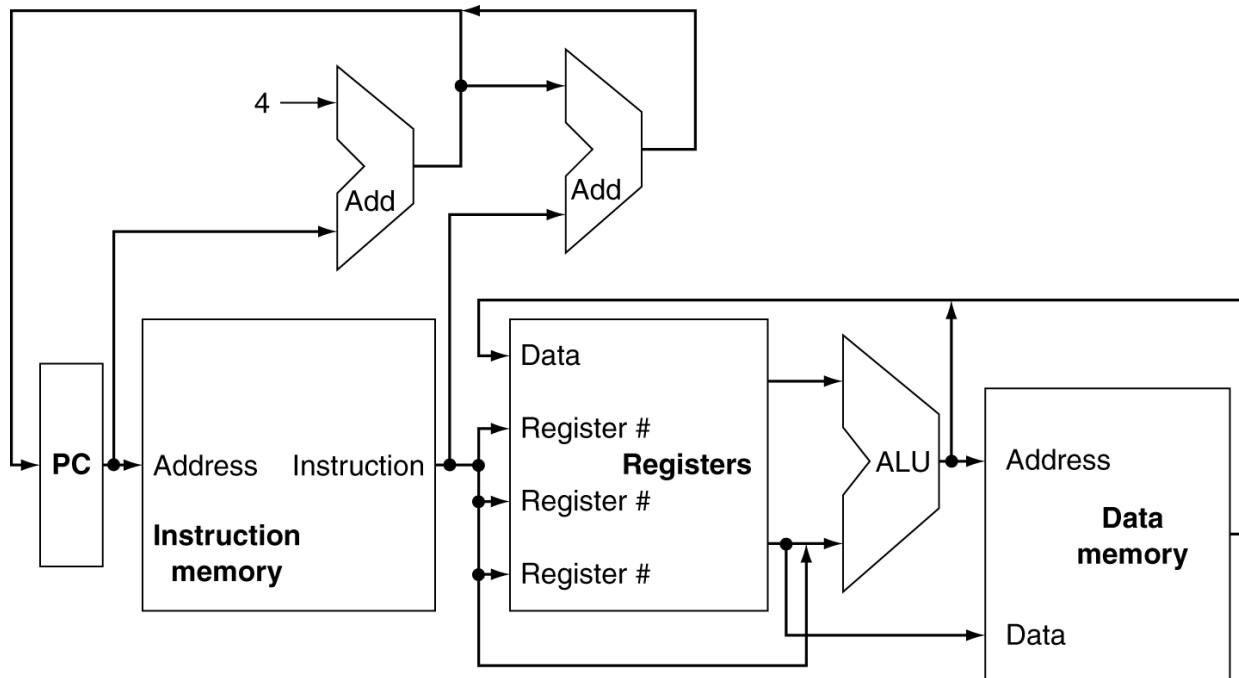
Aritméticos/Lógicos

op	rs	rt	rd	sa	funct
----	----	----	----	----	-------

Armazenamento/  
Desvio

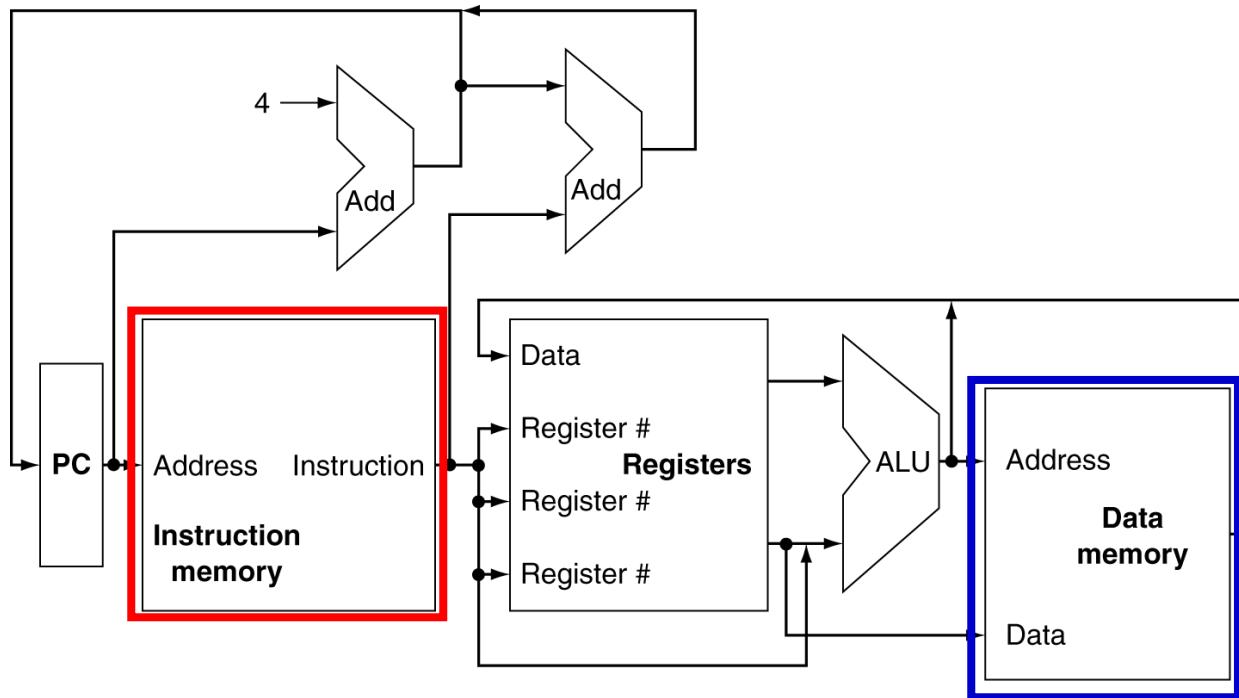
op	rs	rt	deslocamento
----	----	----	--------------

# MIPS: Visão Abstrata



- Na CPU simples, 2 etapas sempre executadas independente da instrução
  - Endereço do PC é enviado a memória para buscar instrução
  - Ler um ou dois registradores, especificados na instrução

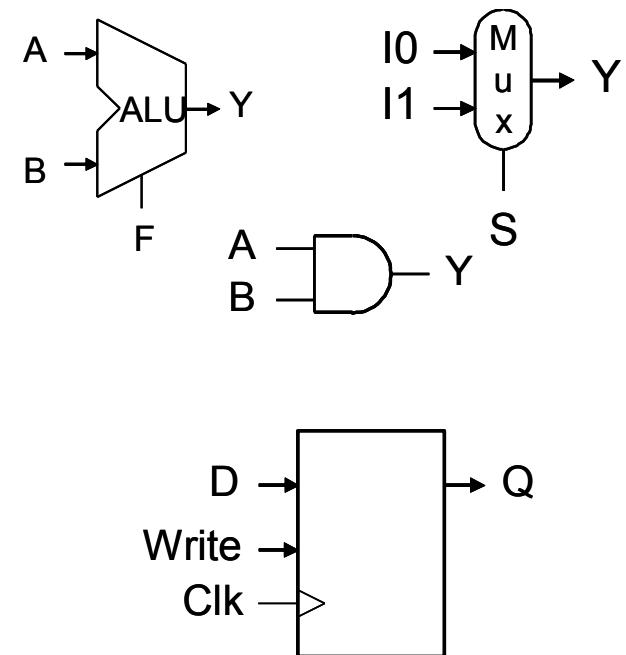
# Por Que Duas Memórias?



- Na CPU simples, uma instrução deve ser executada em um ciclo
  - Nenhuma unidade funcional pode ser utilizada mais de uma vez no mesmo ciclo, exceto banco de regs. (pode haver leitura e escrita)
  - Duplica-se a memória: **instruções** e **dados**

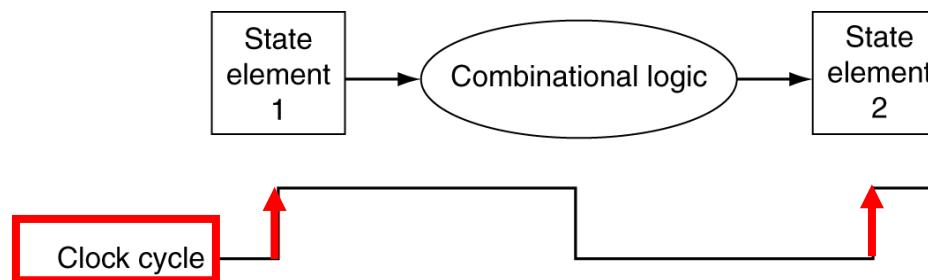
# Categorias de Elementos da CPU

- A CPU é composta por 2 categorias de elementos:
  - Combinacional
  - Sequencial
- Combinacional
  - Operam sobre dados
  - Saída é uma função da entrada
  - ALUs, multiplexadores...
- Sequencial
  - Possui um estado
  - Armazena informações
  - Registradores, memória...

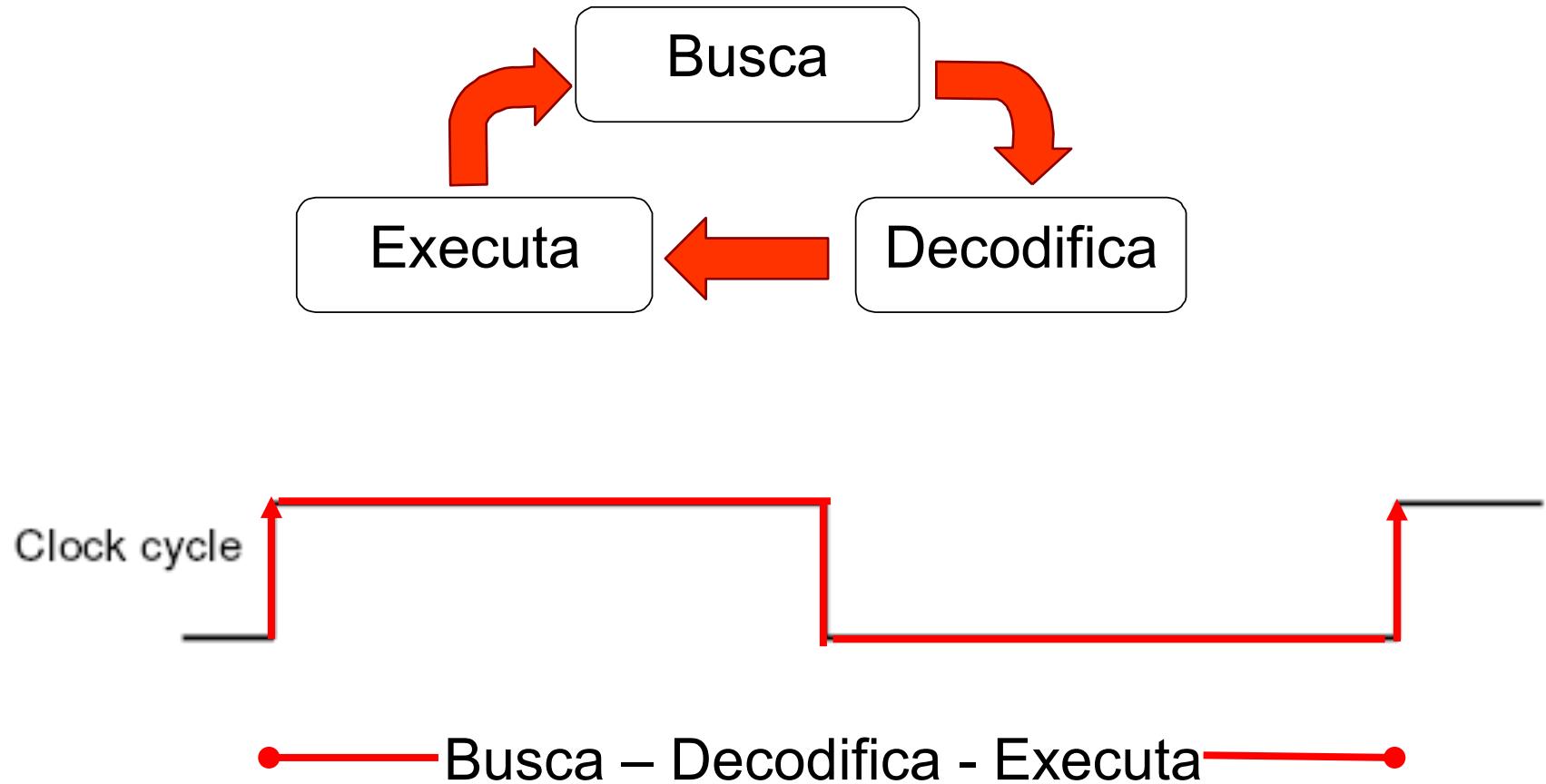


# Relógio (Clock)

- Lógica combinacional opera sobre dados durante ciclos de clock
  - Entre subidas ou descidas do clock (clock edges)
- Execução típica na CPU
  - Lê conteúdo de elementos sequenciais → envia valores para lógica combinacional → escreve resultados em um ou mais elementos sequenciais
- Mudança de estado (escritas) ocorre nos clock edges
  - Se não ocorrer, deve haver um sinal explícito para habilitar escrita



# Implementação Monociclo

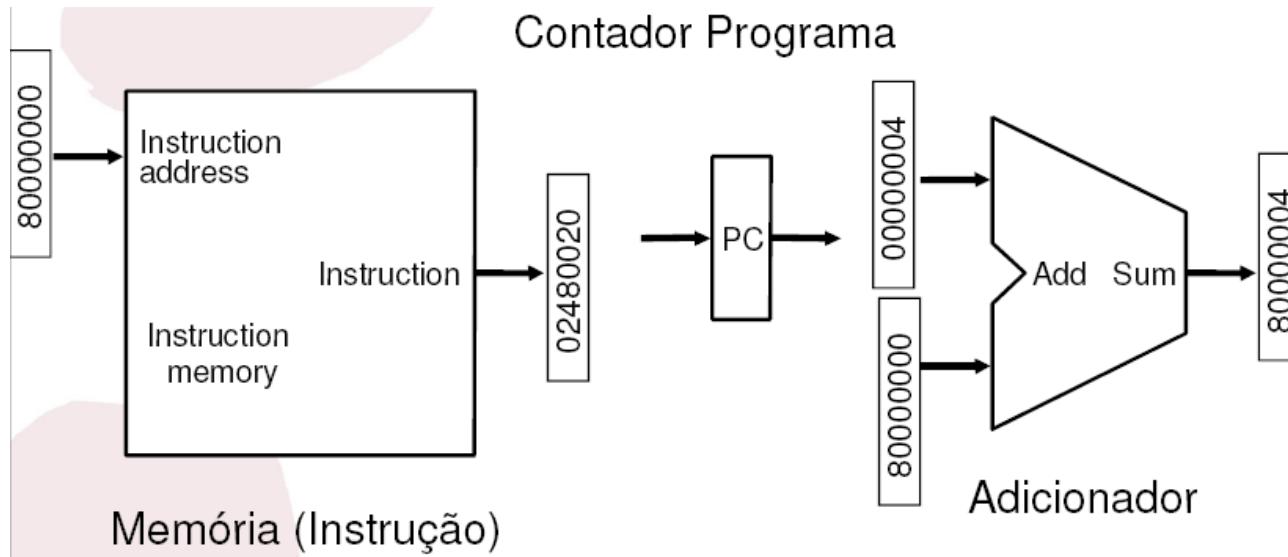


# Construindo a Unidade de Processamento (Datapath)

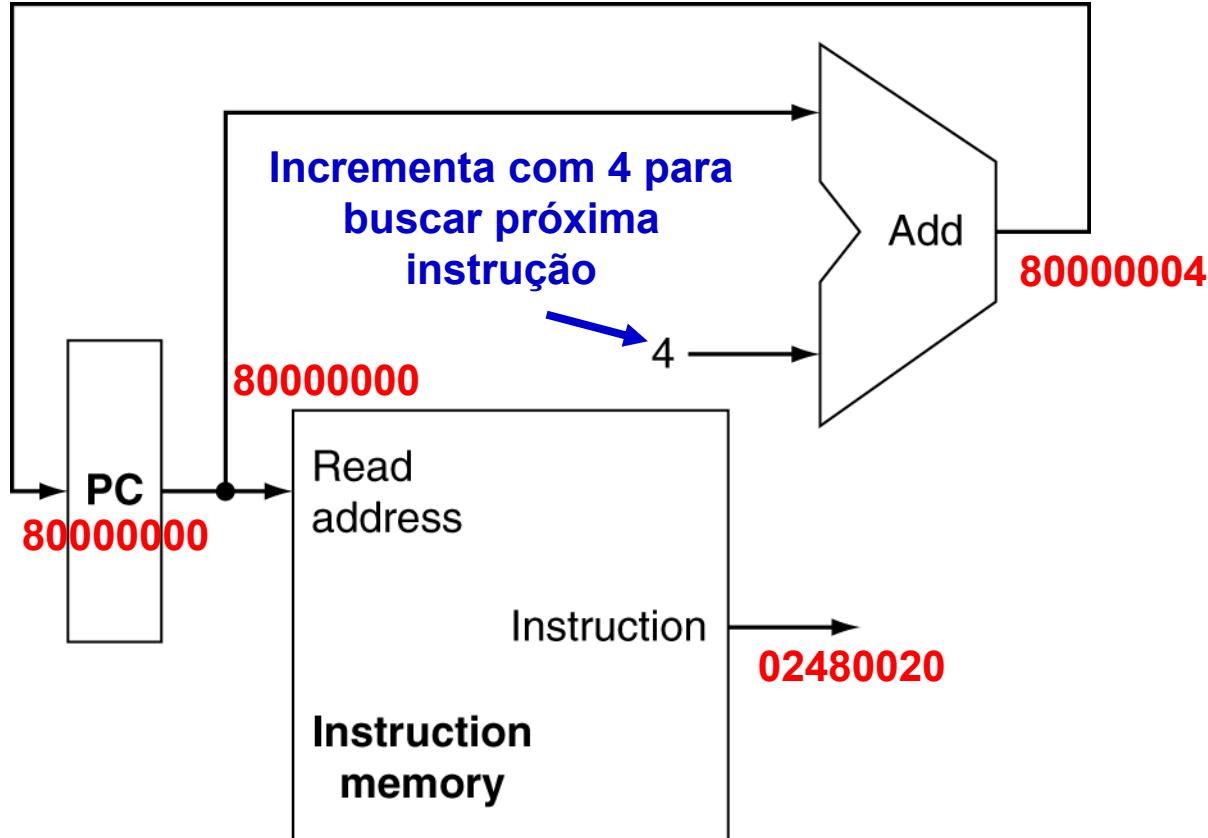
- Precisamos analisar o que cada etapa do ciclo de execução de uma instrução precisa
  - Busca e execução
- Precisamos analisar o que cada instrução do repertório precisa
  - `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`

# Componentes Básicos: Busca de Instrução

- Memória
  - Entrada: endereço , Saída: instrução
- Registrador PC de 32 bits
  - Contém endereço da instrução a ser executada
  - Atualizado todo ciclo de clock
- Somador
  - Soma PC com valor constante

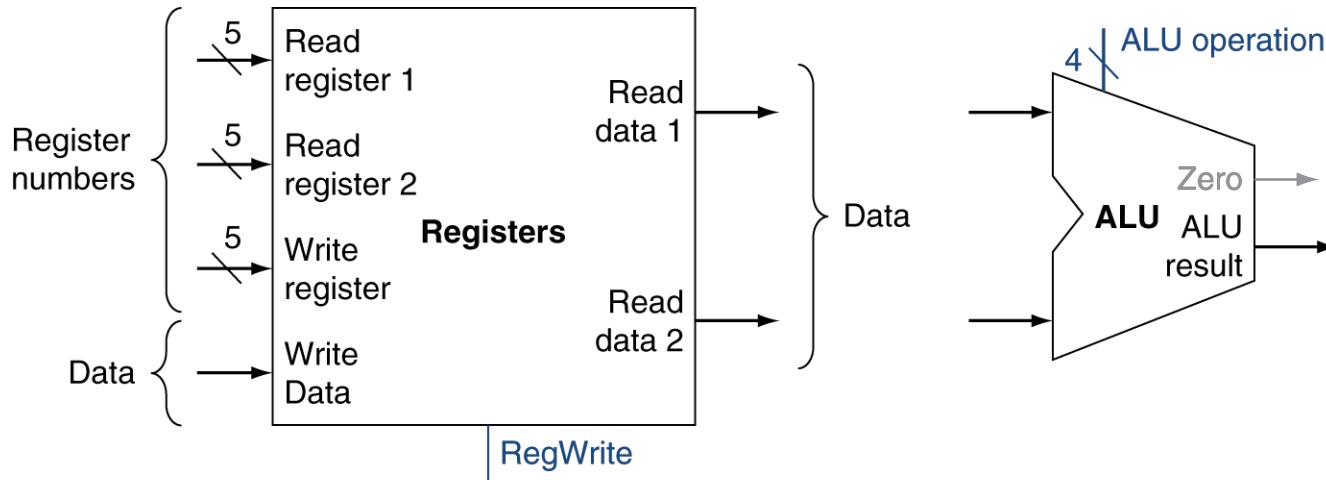


# Busca de Instrução



# Componentes Básicos: Instruções Aritméticas e Lógicas

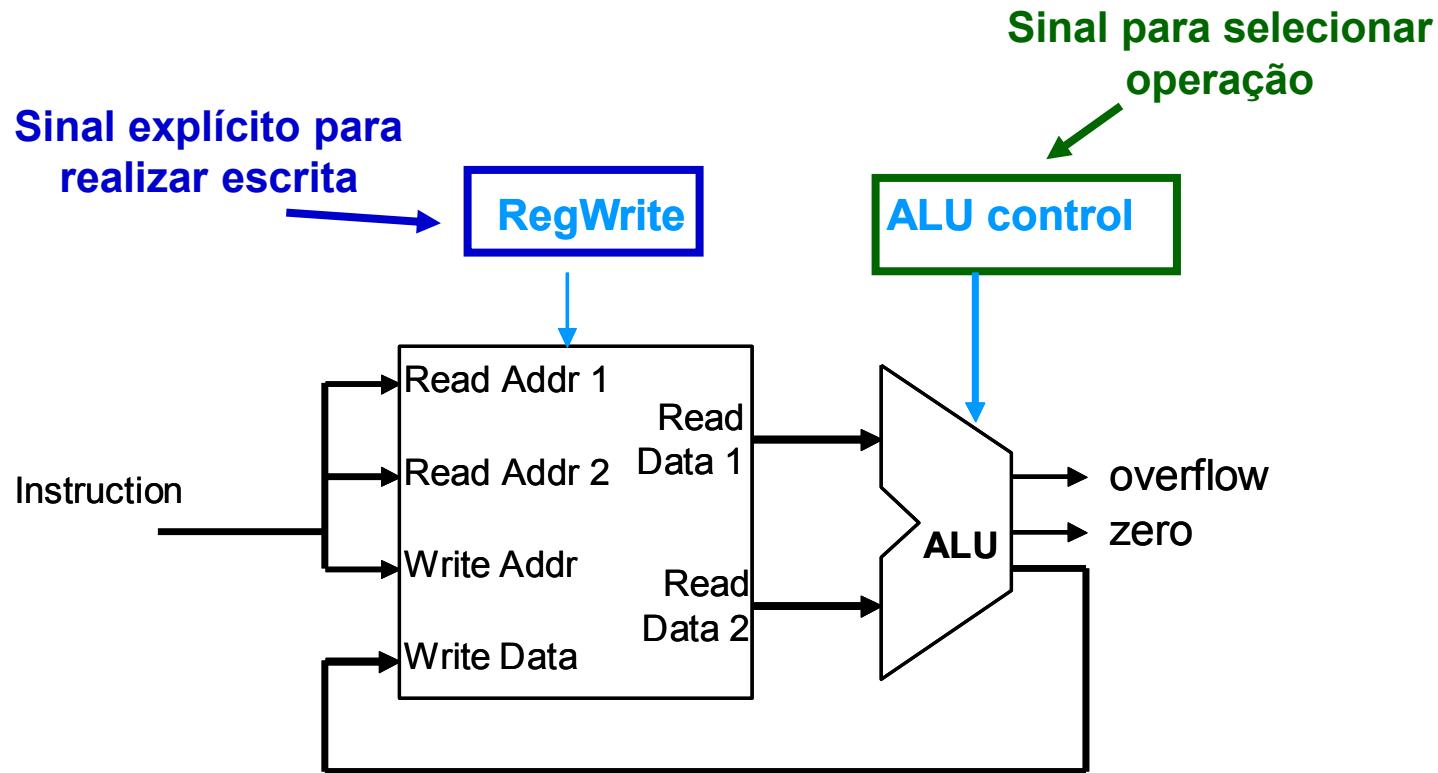
- Banco de registradores
  - Lê dois registradores
  - Armazena resultado em um registrador
- ALU calcula operandos vindos dos registradores



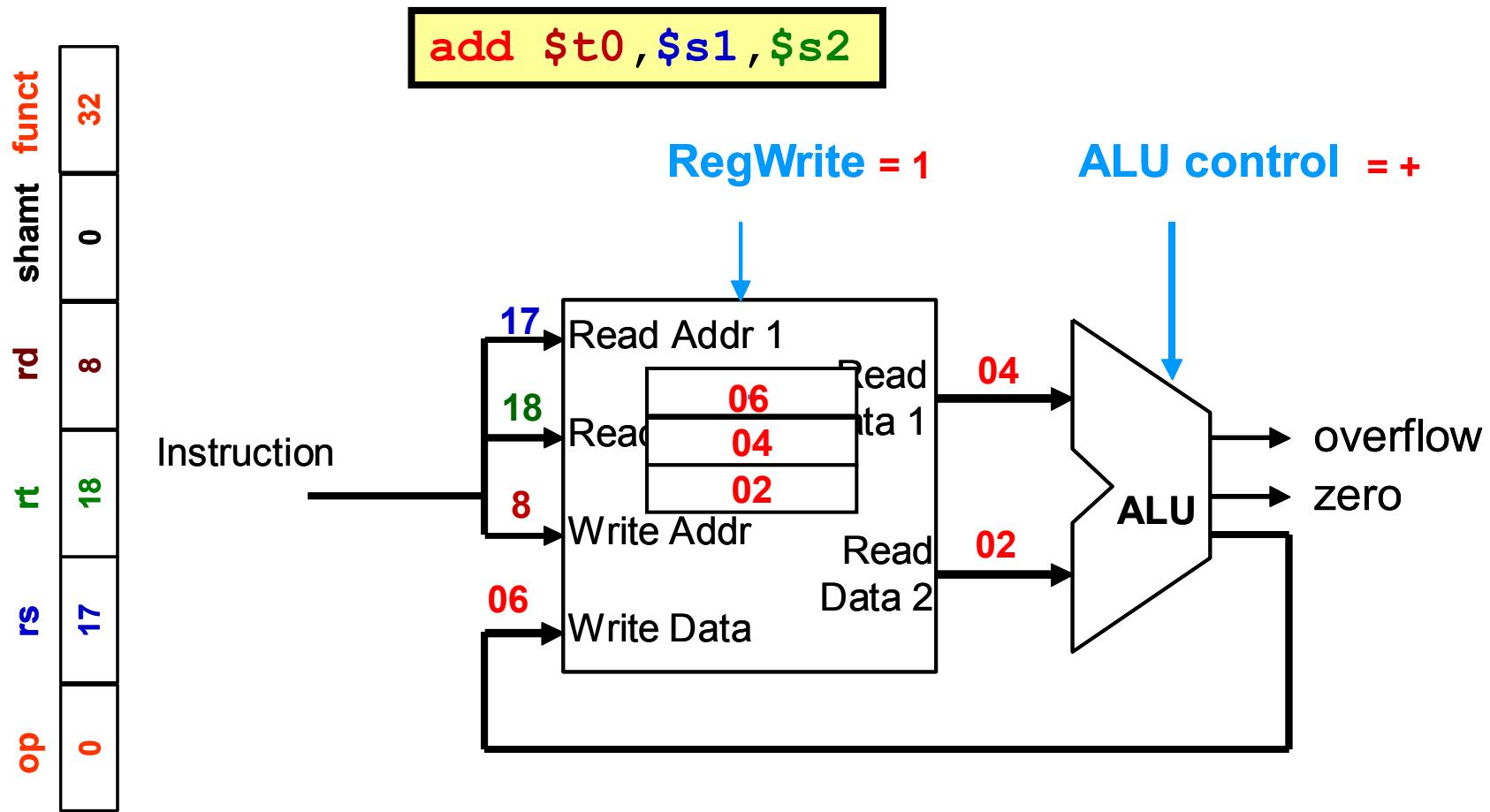
a. Registers

b. ALU

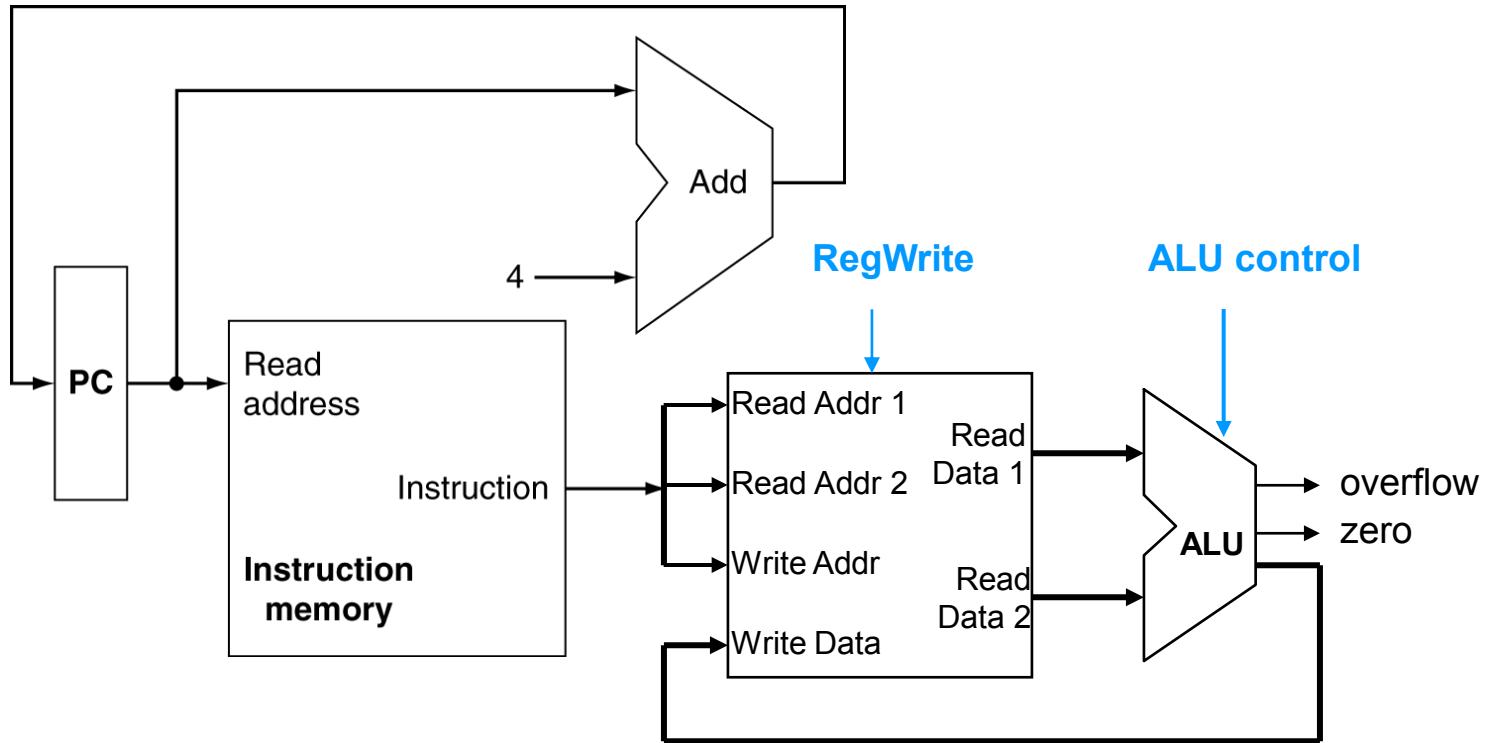
# Instruções Aritméticas e Lógicas



# Instruções Aritméticas e Lógicas

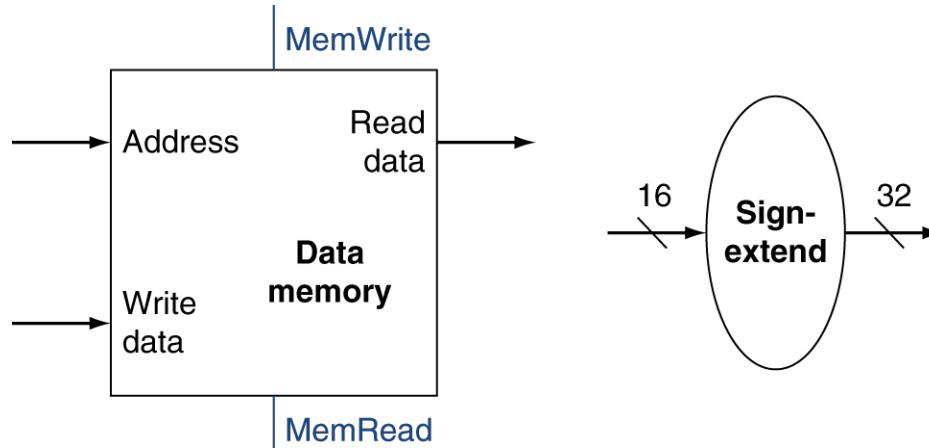


# Busca e Execução de Instruções Aritméticas e Lógicas



# Componentes Básicos: Instruções de Armazenamento

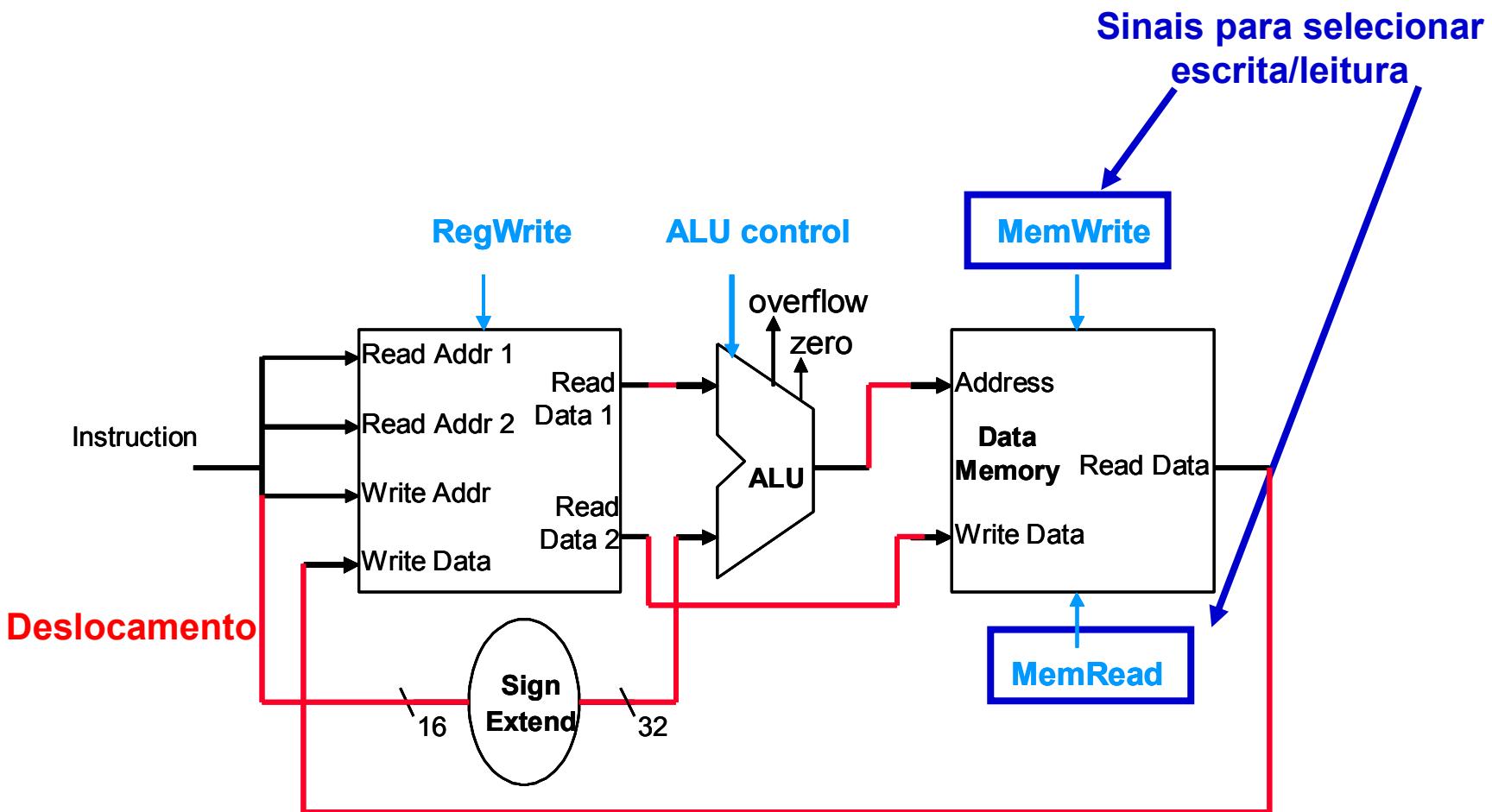
- **Memória de dados**
  - Leitura e escrita
- **Unidade de extensão de sinal**
  - Para transformar o deslocamento de 16 bits em 32 bits
- Registradores para ler e escrever
- ALU para calcular endereço: registrador base + deslocamento



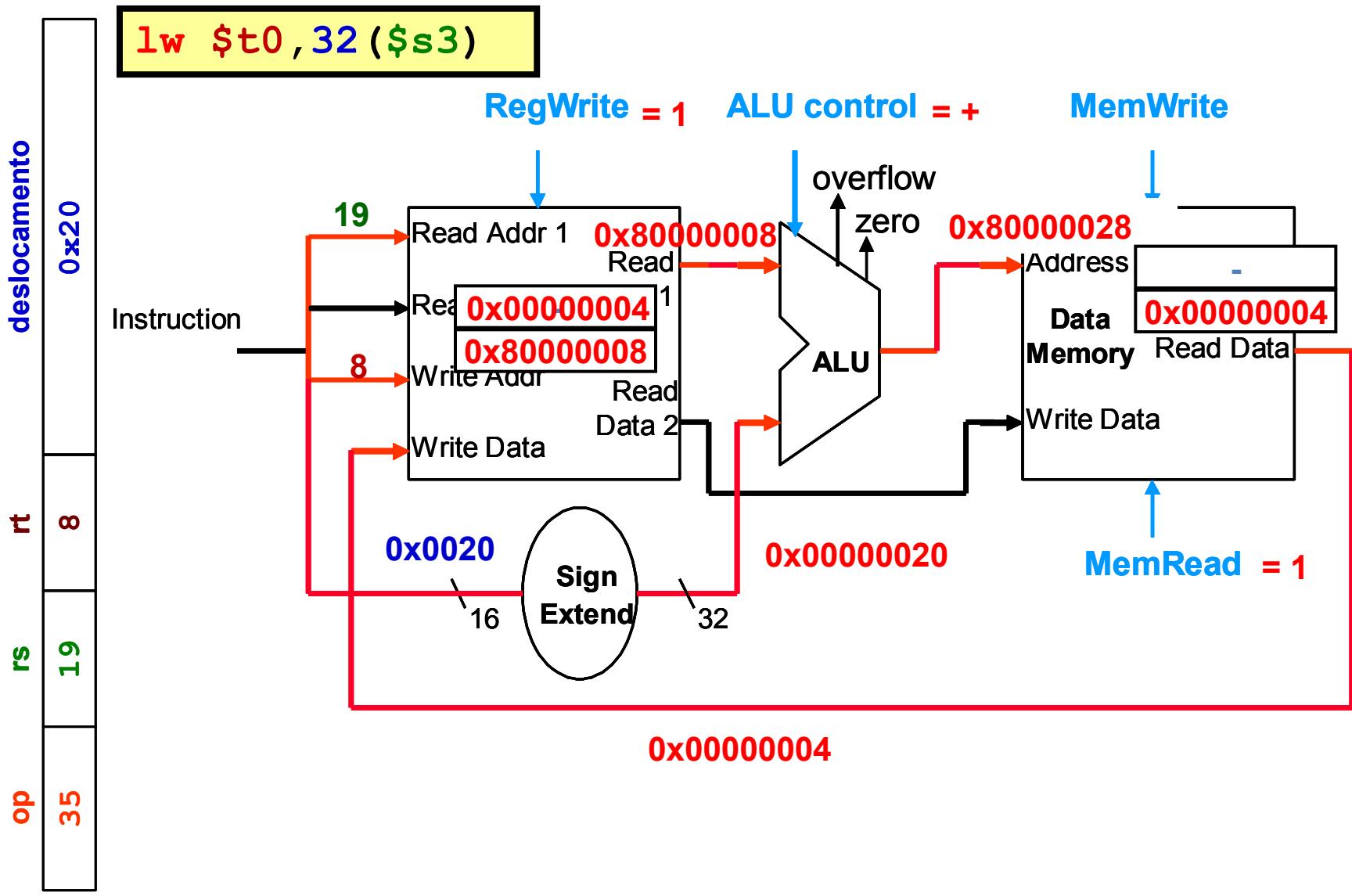
a. Data memory unit

b. Sign extension unit

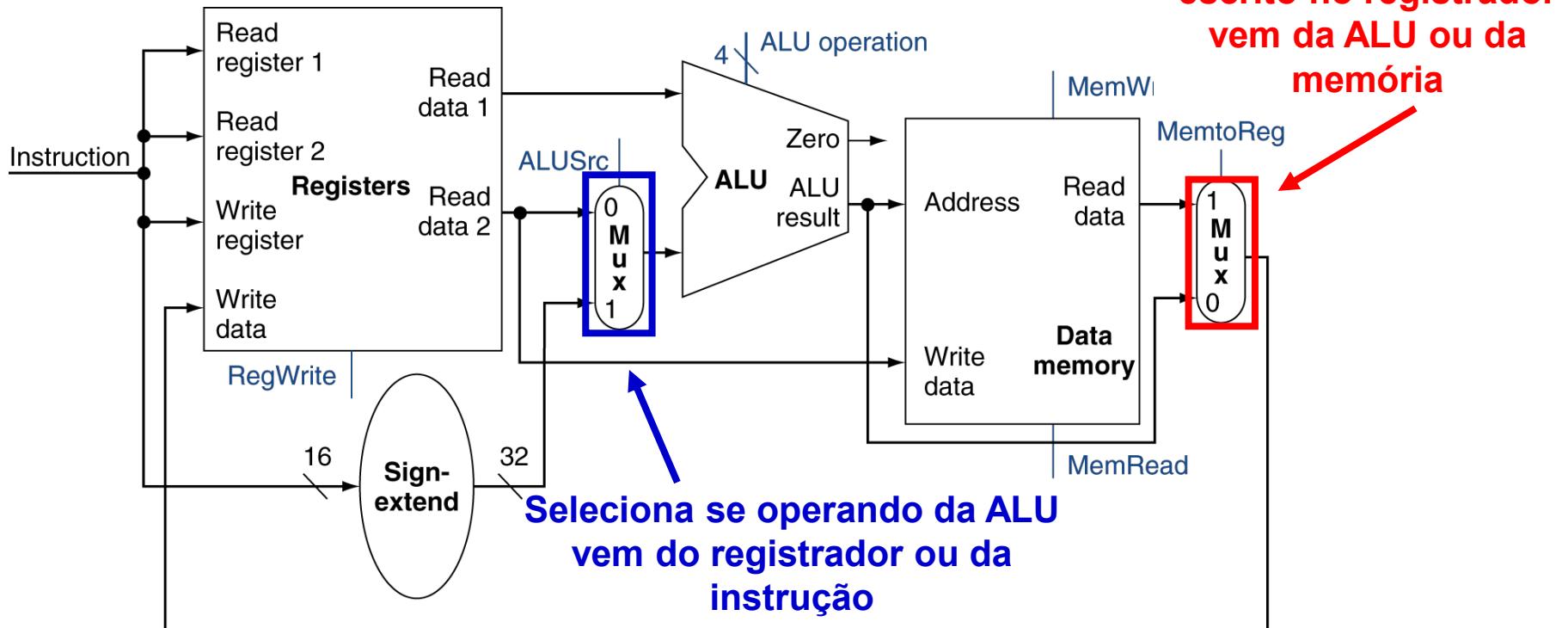
# Instruções de Armazenamento



# Instruções de Armazenamento

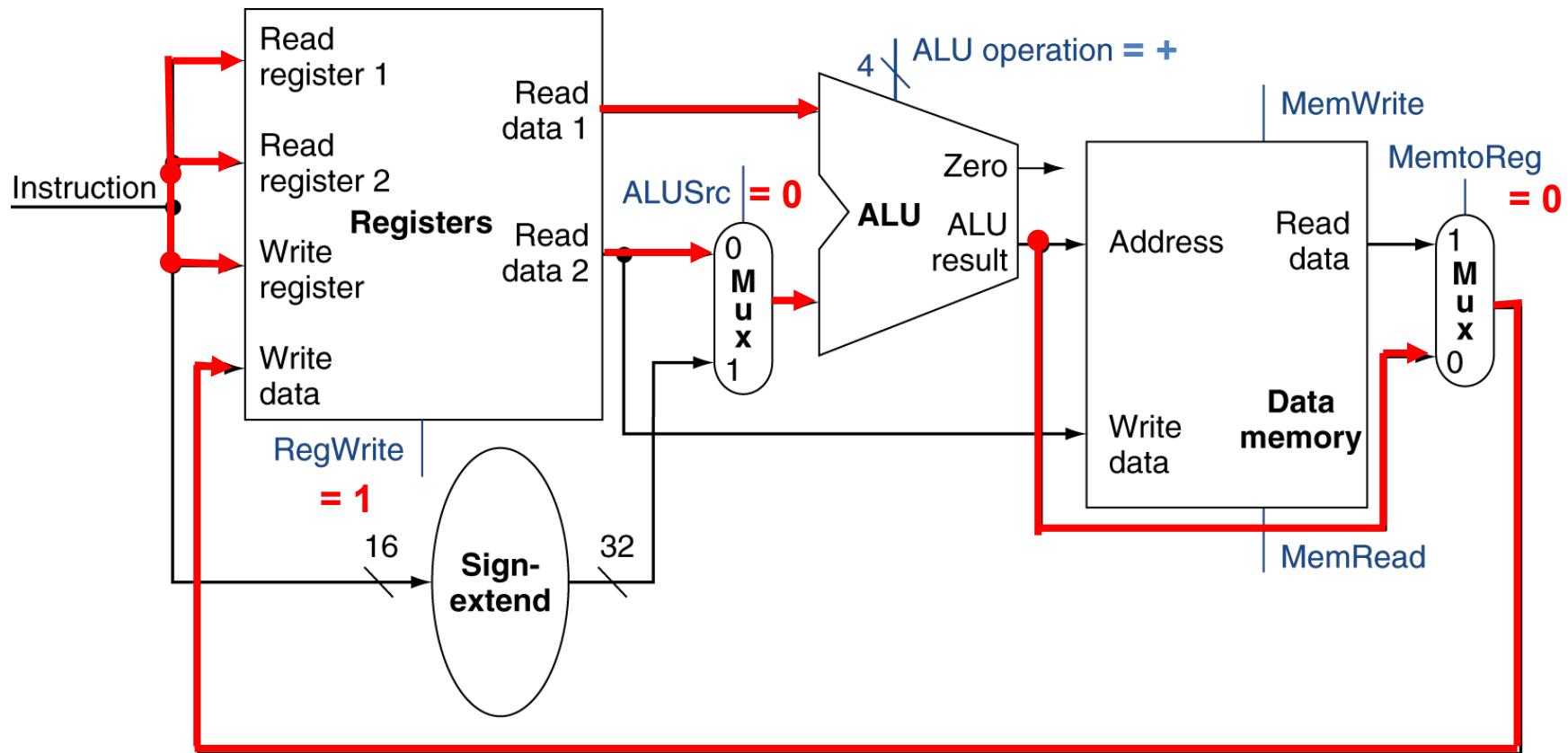


# Instruções Aritméticas e de Armazenamento

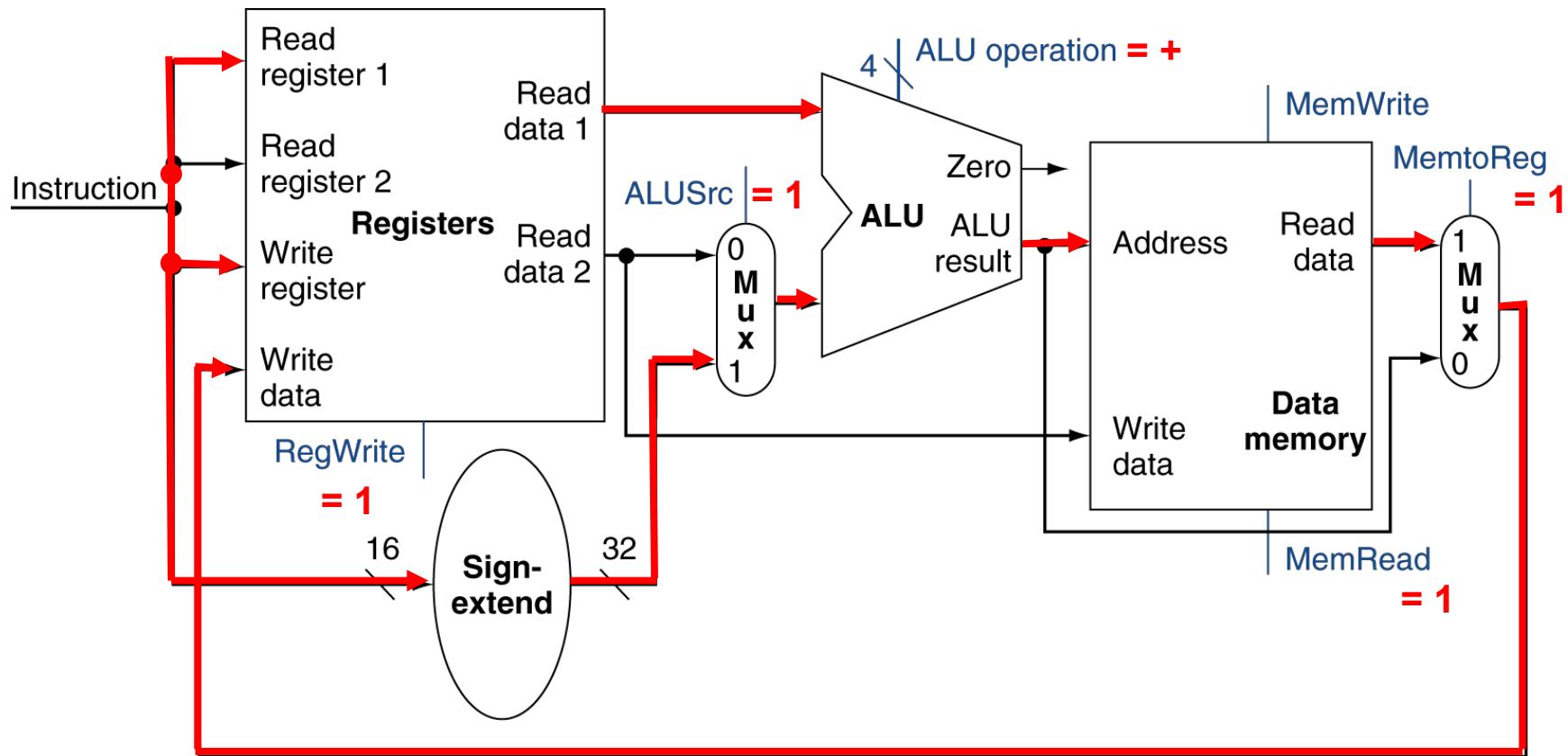


- Dois multiplexadores são necessários para selecionar entrada:
  - Registrador (aritmética) ou Deslocamento(armazenamento)
  - ALU (aritmética) ou Memória (armazenamento)

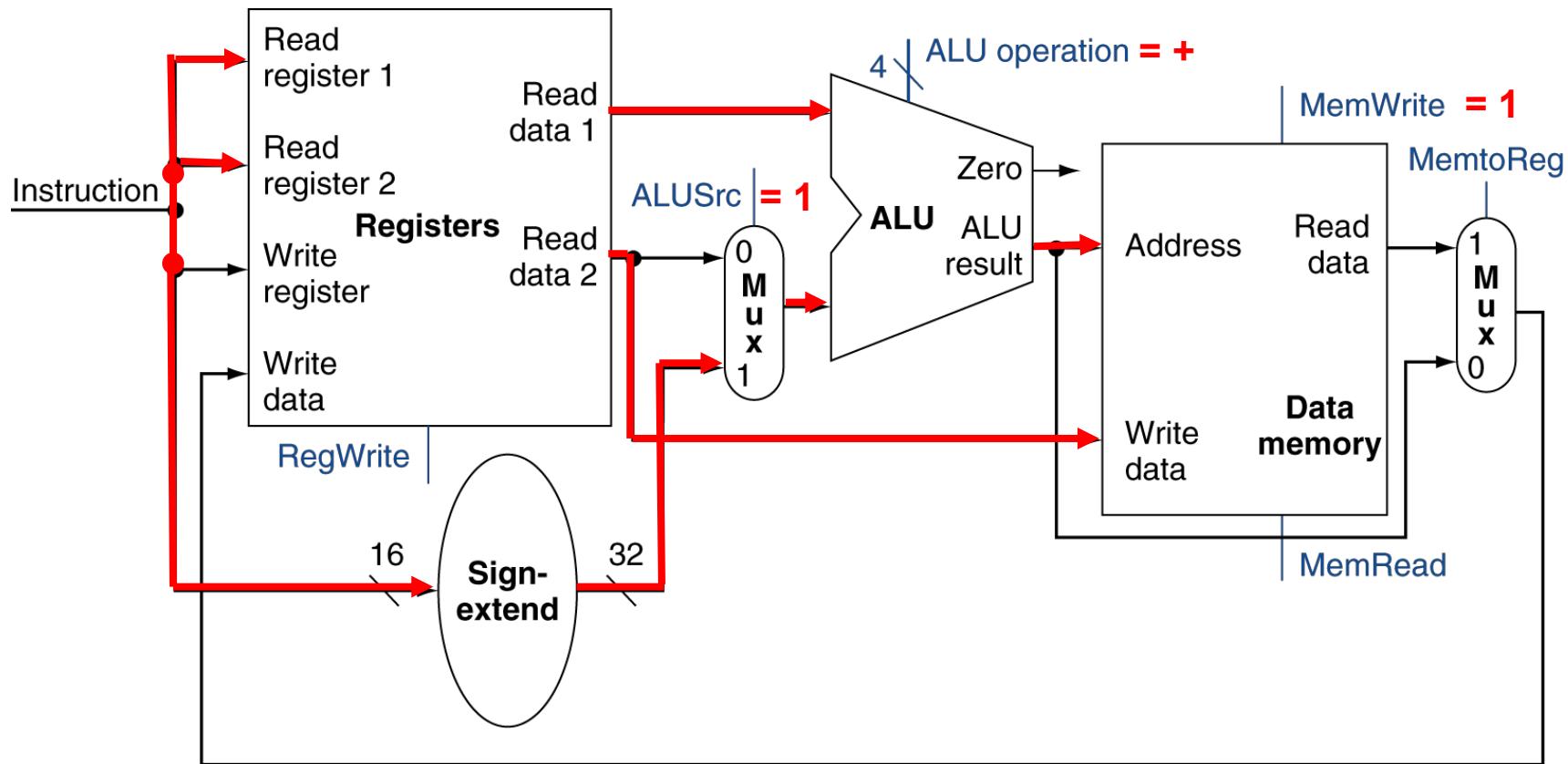
# Executando ADD



# Executando LW



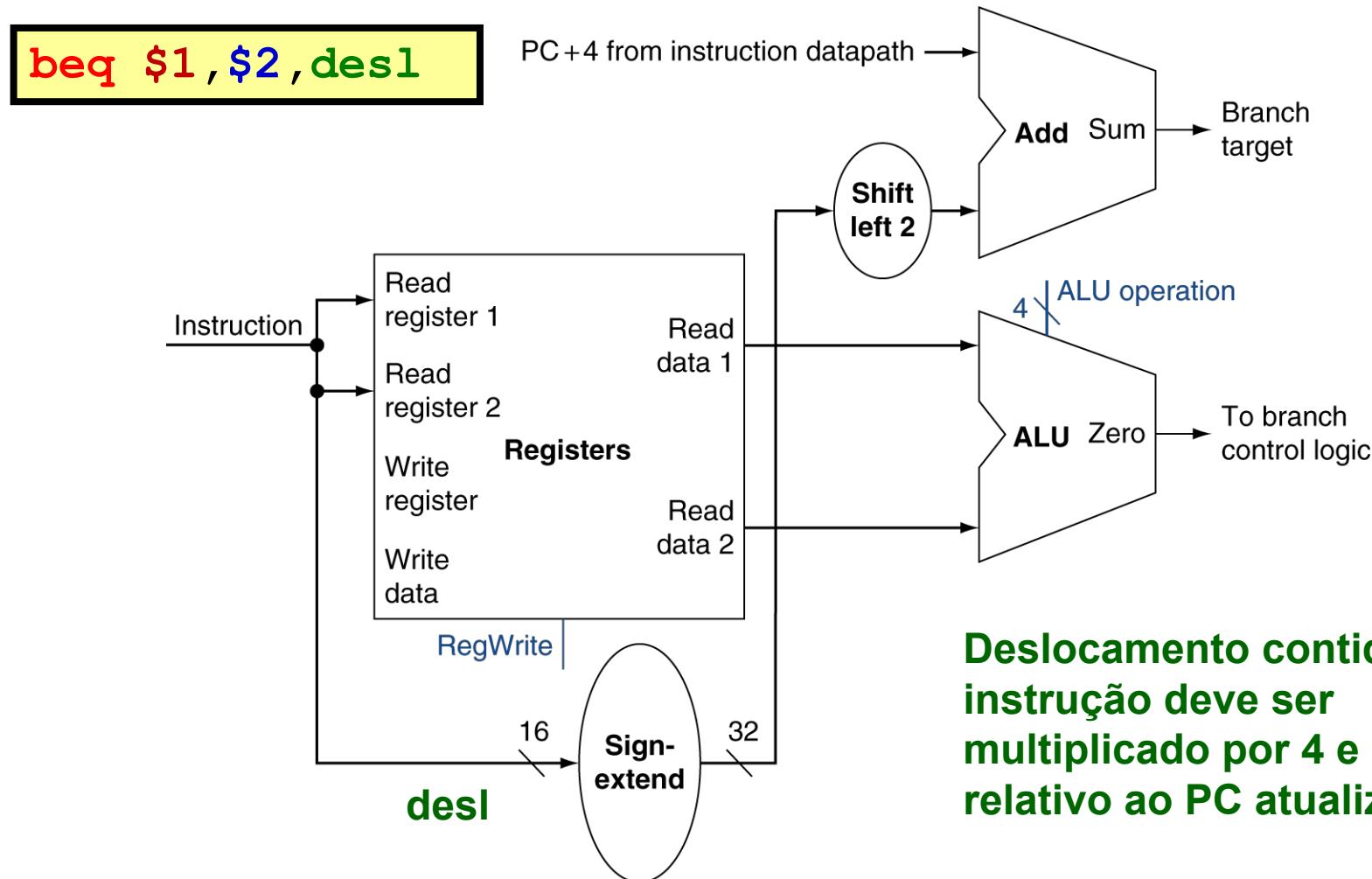
# Executando SW



# Componentes Básicos: Instruções de Branch

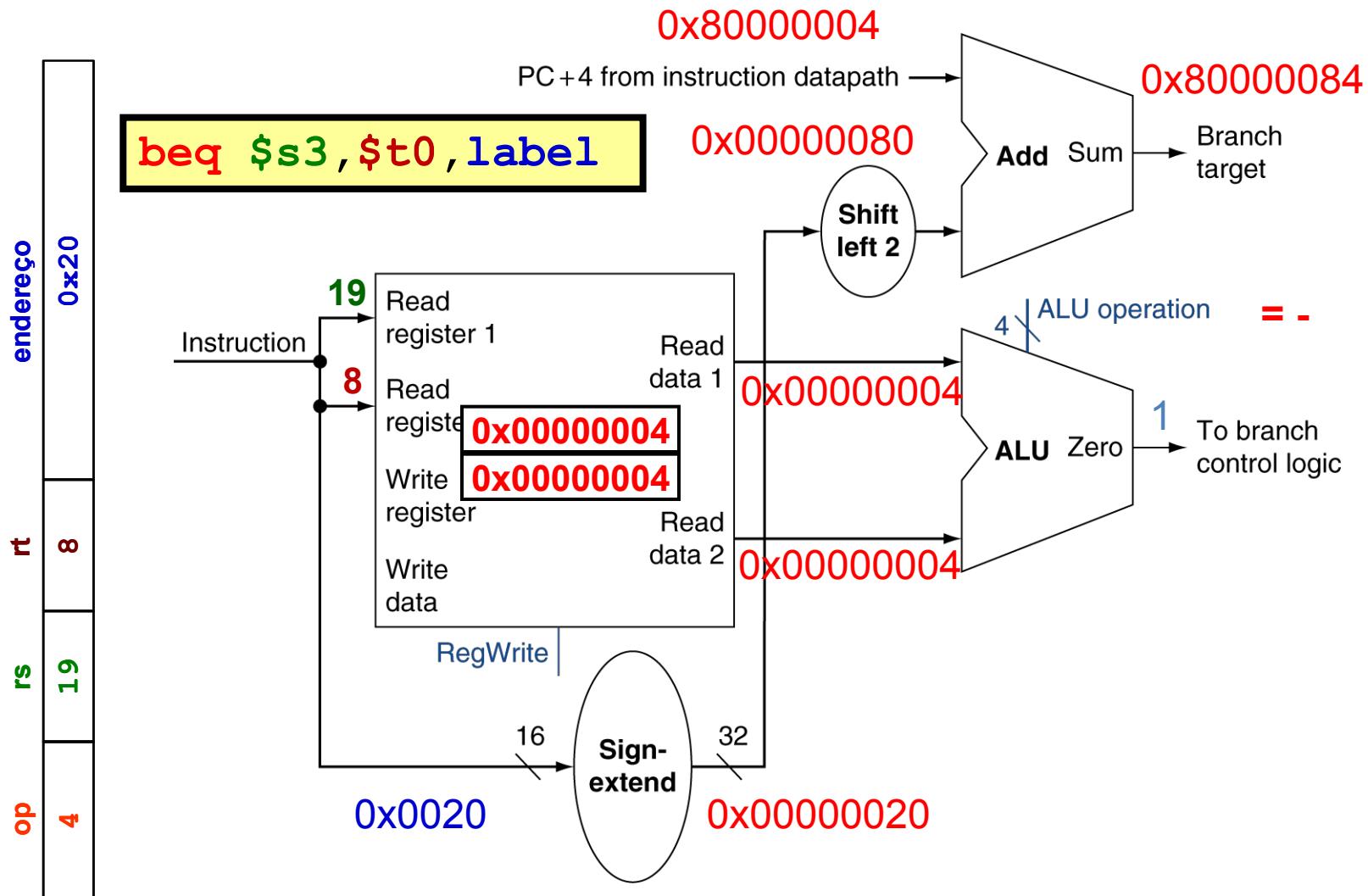
- Registradores para ler e escrever
- ALU para comparar operandos
  - Subtrai operandos e checa a saída 0
- Unidade de extensão de sinal
  - Para transformar o endereço contido na instrução (relativo) de 16 bits em 32 bits
- **Unidade para deslocar de 2 bits para esquerda o deslocamento relativo**
  - Multiplicar por 4
- Somador **para adicionar o deslocamento relativo x 4 ao endereço do PC**

# Instrução de Branch

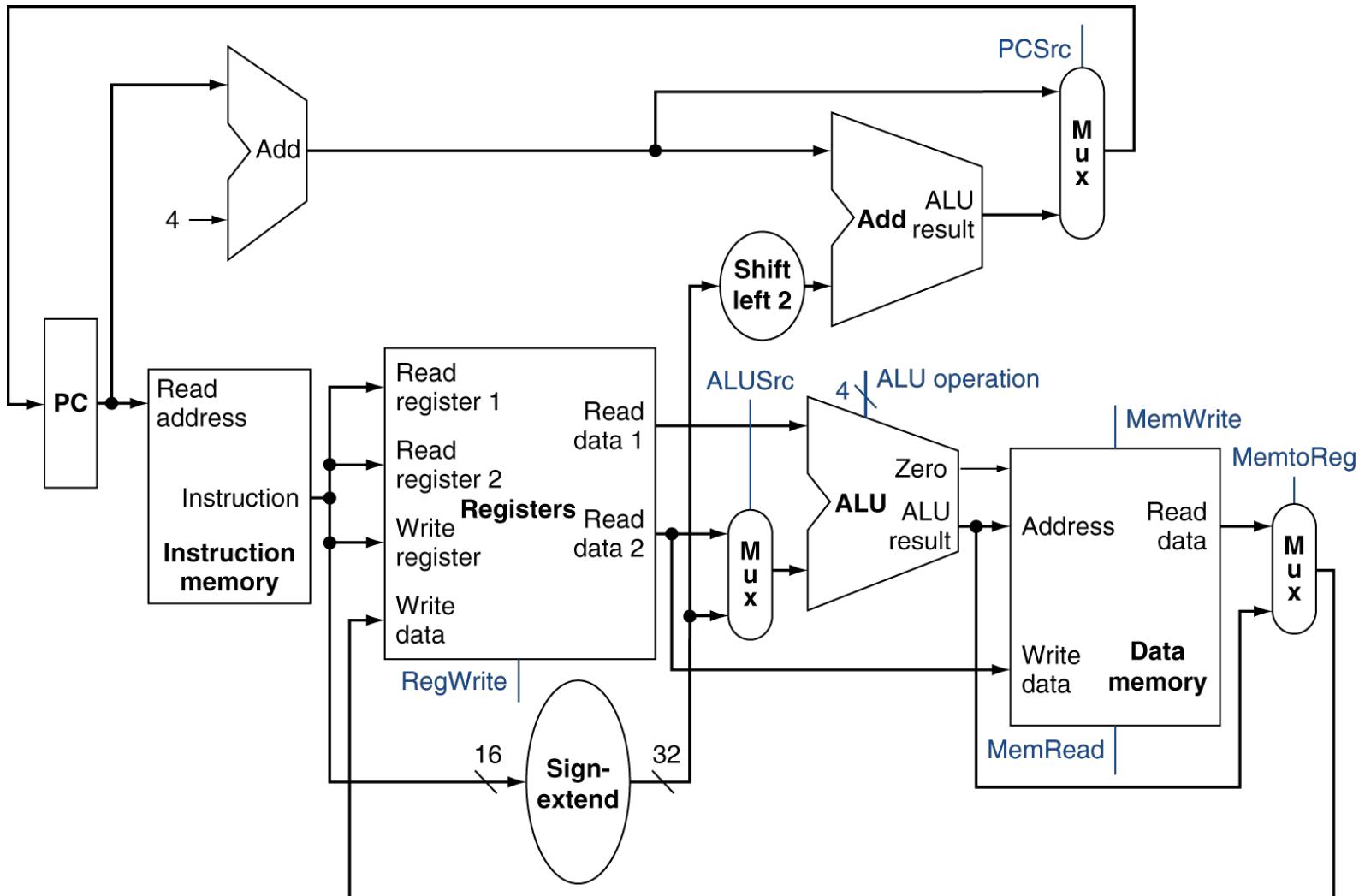


Deslocamento contido na instrução deve ser multiplicado por 4 e é relativo ao PC atualizado

# Executando BEQ



# Unidade de Processamento (Quase) Completa



# Infraestrutura de Hardware

Instruindo um Computador – Subrotinas aninhadas e  
Modos de Endereçamento

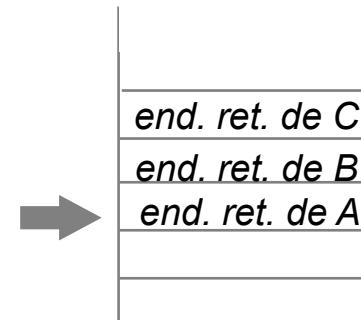
Prof. Adriano Sarmento

# Subrotinas Aninhadas

- Como executar subrotinas aninhadas?
  - Funções que chamam outras
  - Funções recursivas
- Subrotina que chama outra armazena na pilha:
  - seu endereço de retorno
  - registradores que utilize após o término da subrotina chamada

```
void C () {  
    B();  
}  
  
void B() {  
    A();  
}
```

*topo da  
pilha*



# Exemplo de Subrotina Aninhada

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Parâmetro **n** em **\$a0**
- Resultado em **\$v0**

# Assembly de Subrotina Aninhada

fact:

```

addi $sp, $sp, -8      # ajustando pilha p/ 2 itens
sw   $ra, 4($sp)       # salva endereço de retorno
sw   $a0, 0($sp)       # salva argumento
slti $t0, $a0, 1        # testa n < 1
beq  $t0, $zero, L1
addi $v0, $zero, 1      # se sim, resultado é 1
addi $sp, $sp, 8        # pop 2 itens da pilha
jr   $ra                 # e retorna
L1: addi $a0, $a0, -1    # senão decrementa n
jal  fact               # chamada recursiva
lw   $a0, 0($sp)       # restaura n original
lw   $ra, 4($sp)       # e endereço de retorno
addi $sp, $sp, 8        # pop 2 itens da pilha
mul $v0, $a0, $v0       # multiplica p/ obter resultado
jr   $ra                 # e retorna

```

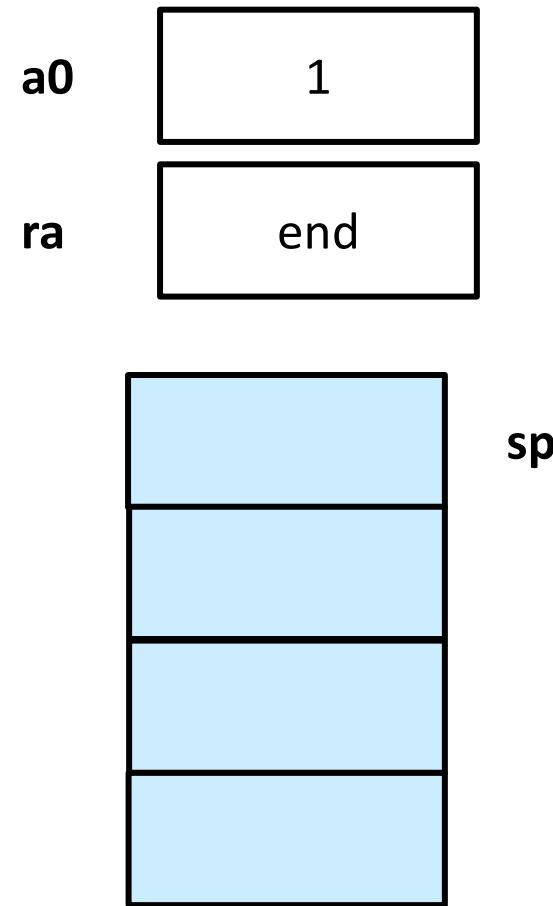
# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra

```



# Exemplo: Fatorial

**fact(1)**

```

1024 fact: addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi $v0, $zero, 1
1048     addi $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

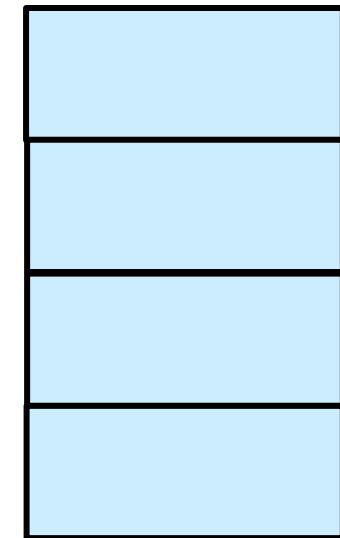


a0

1

ra

end



sp

# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi $v0, $zero, 1
1048     addi $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

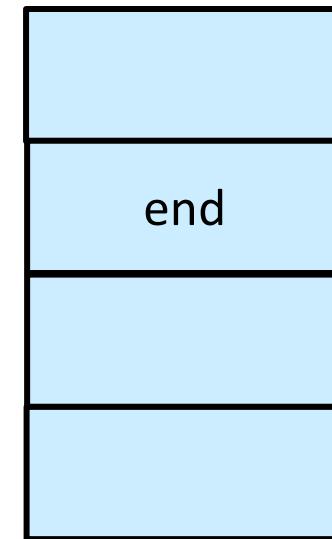


a0

1

ra

end



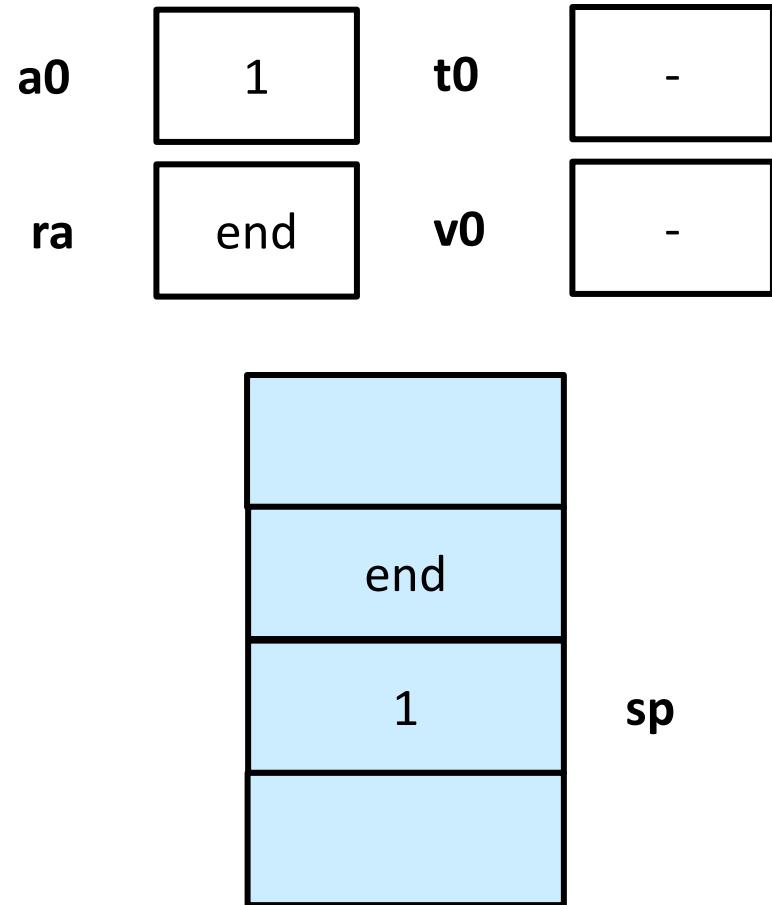
sp

# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp) ←
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

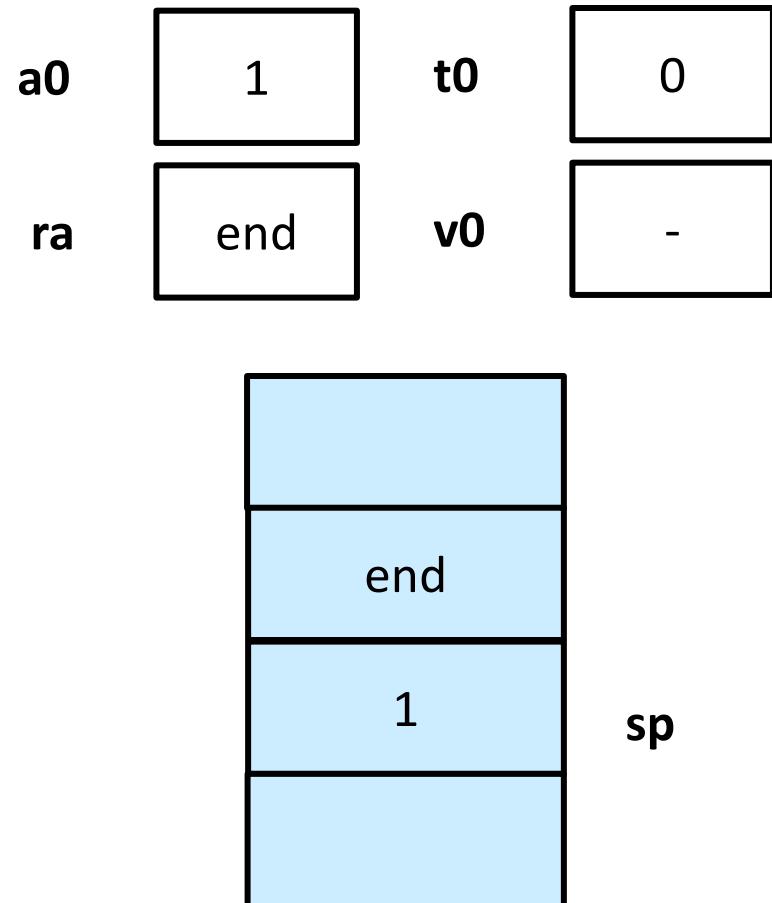


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

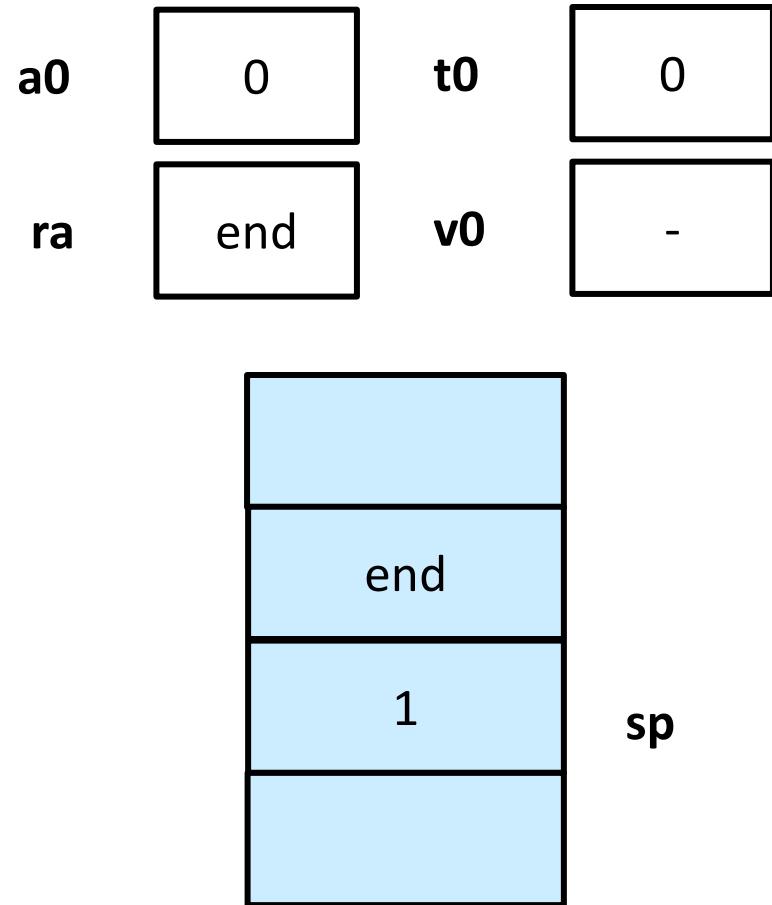


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```



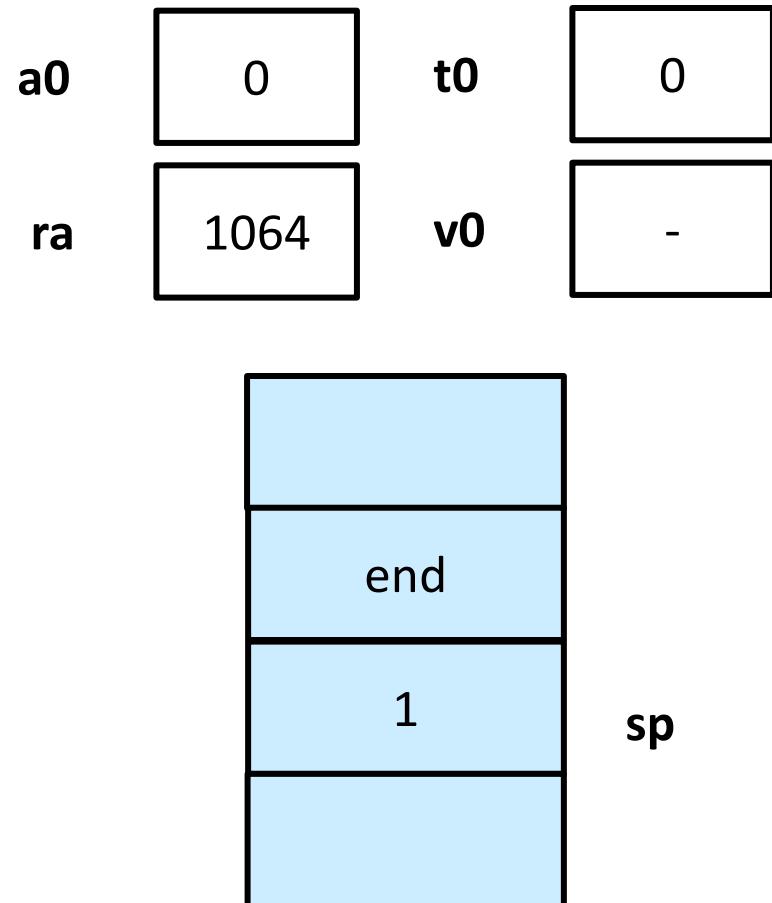
# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra

```



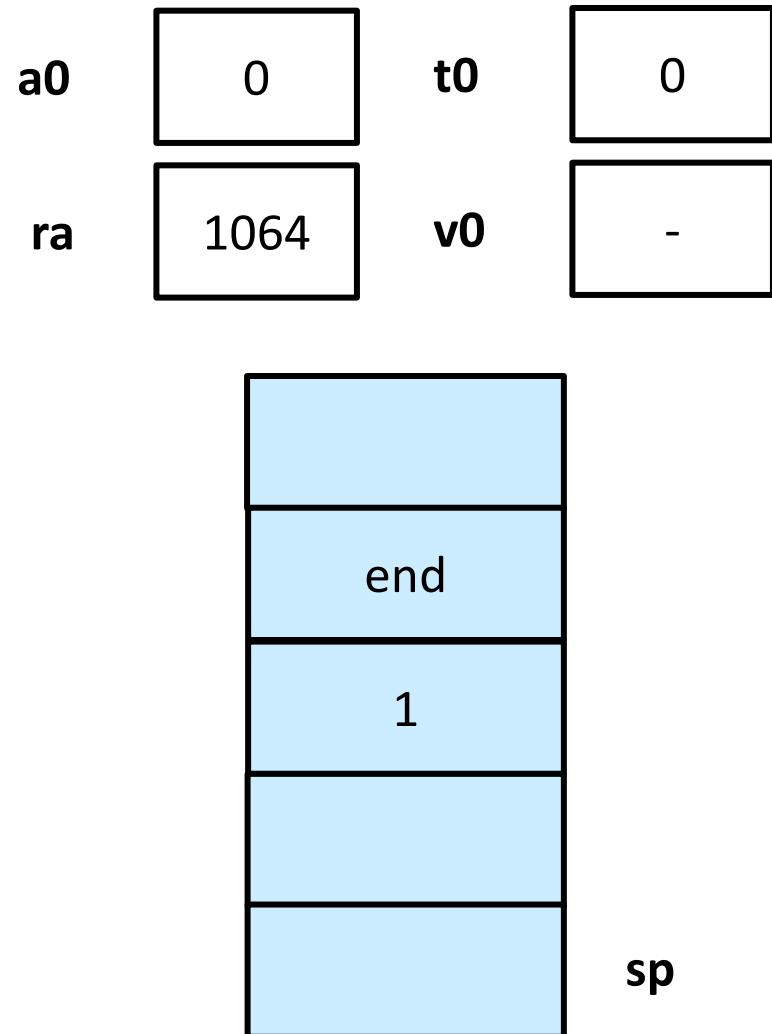
# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra

```

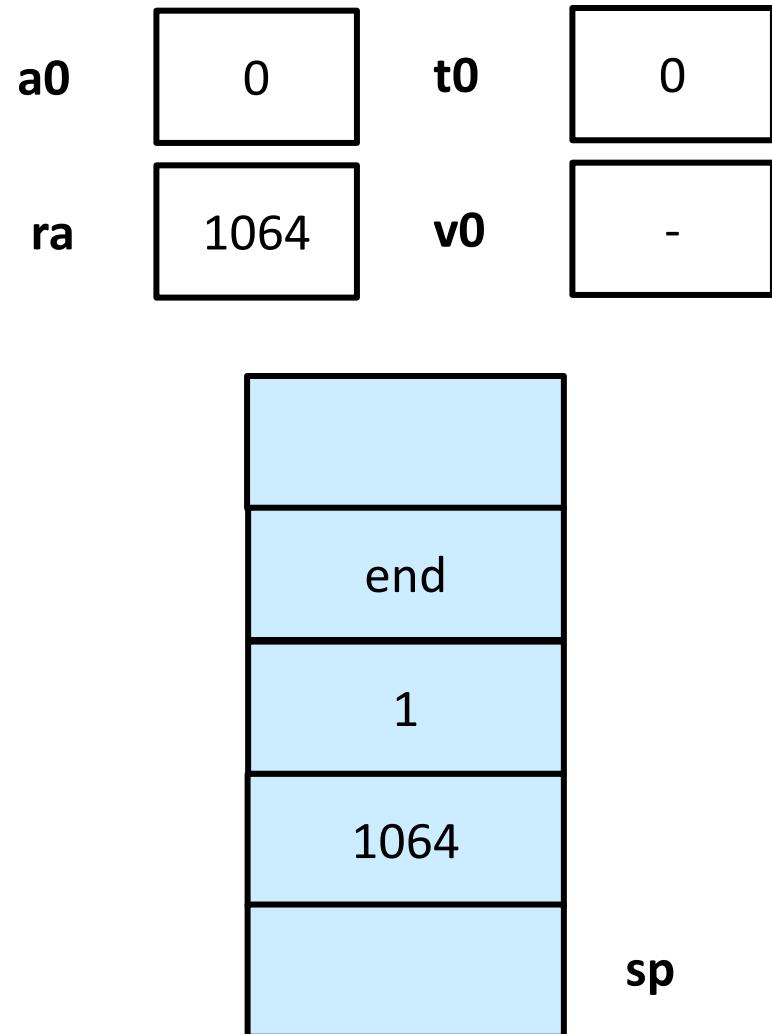


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi $v0, $zero, 1
1048     addi $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

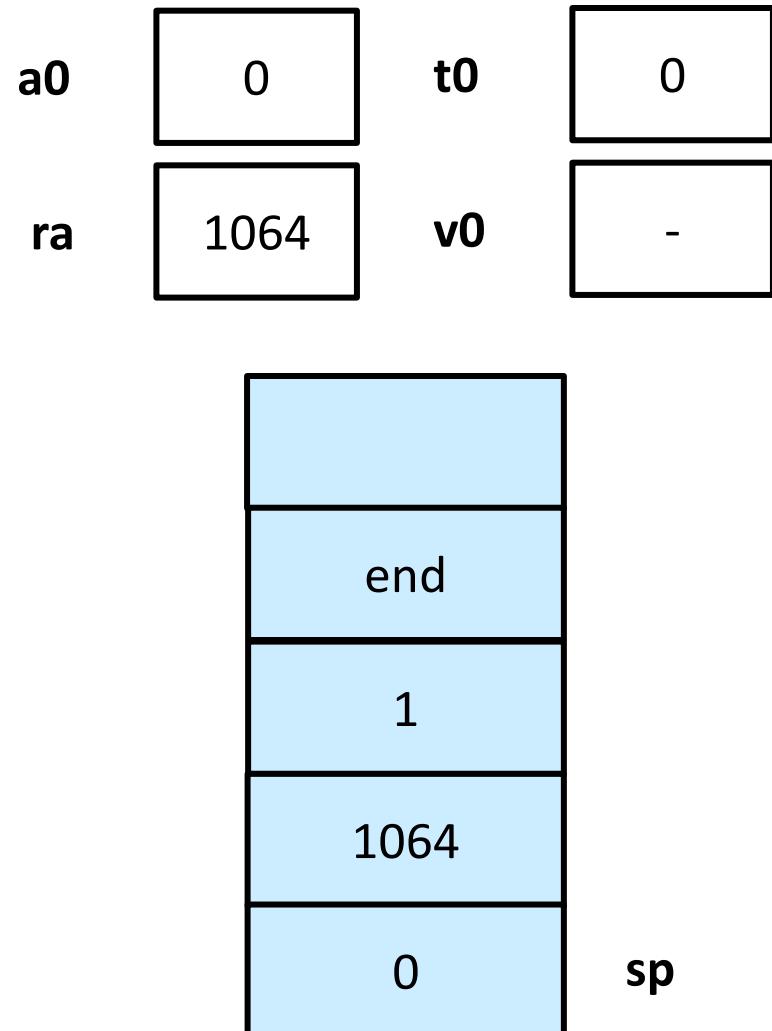


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp) ←
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```



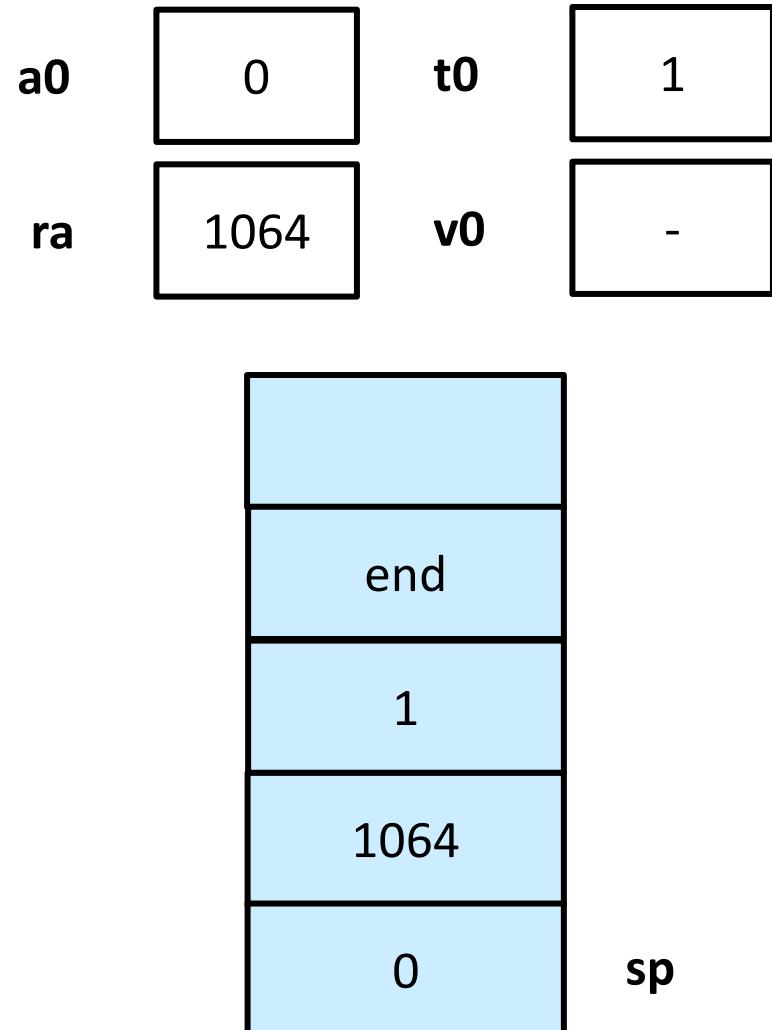
# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra

```

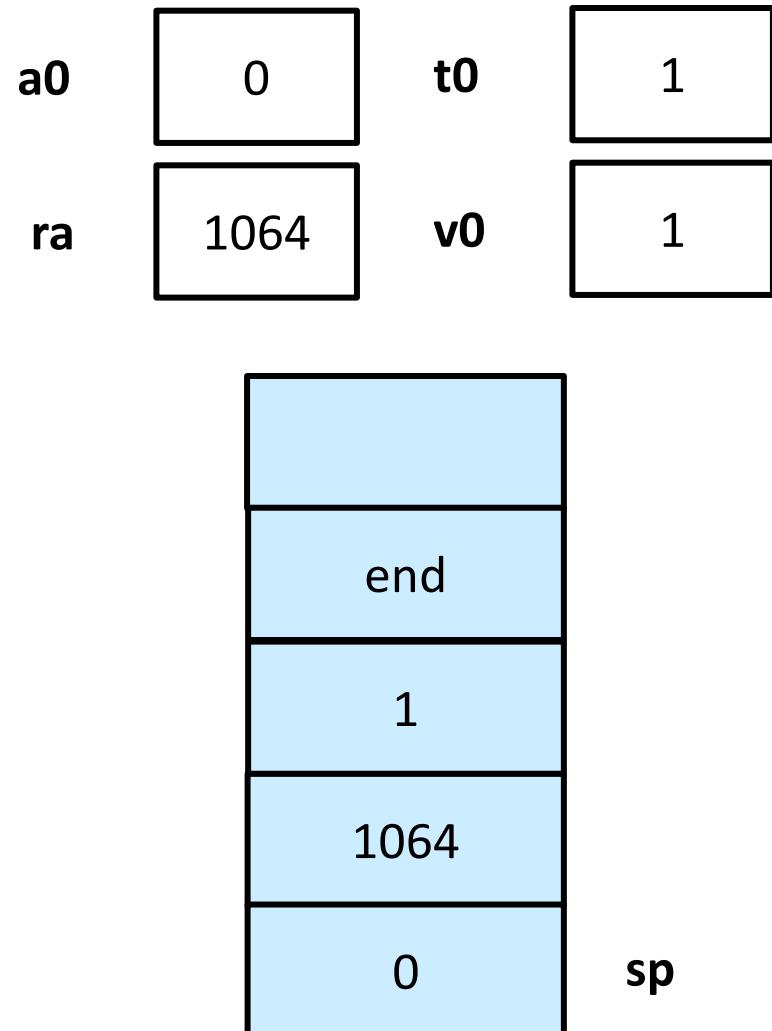


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi $v0, $zero, 1
1048     addi $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

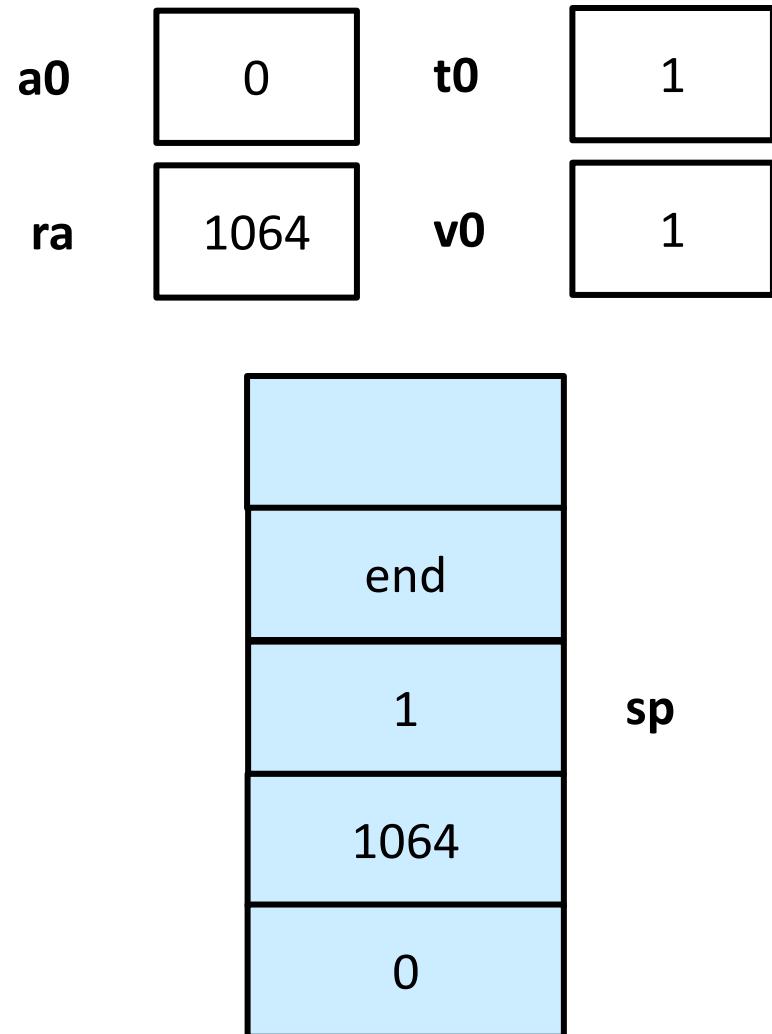


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

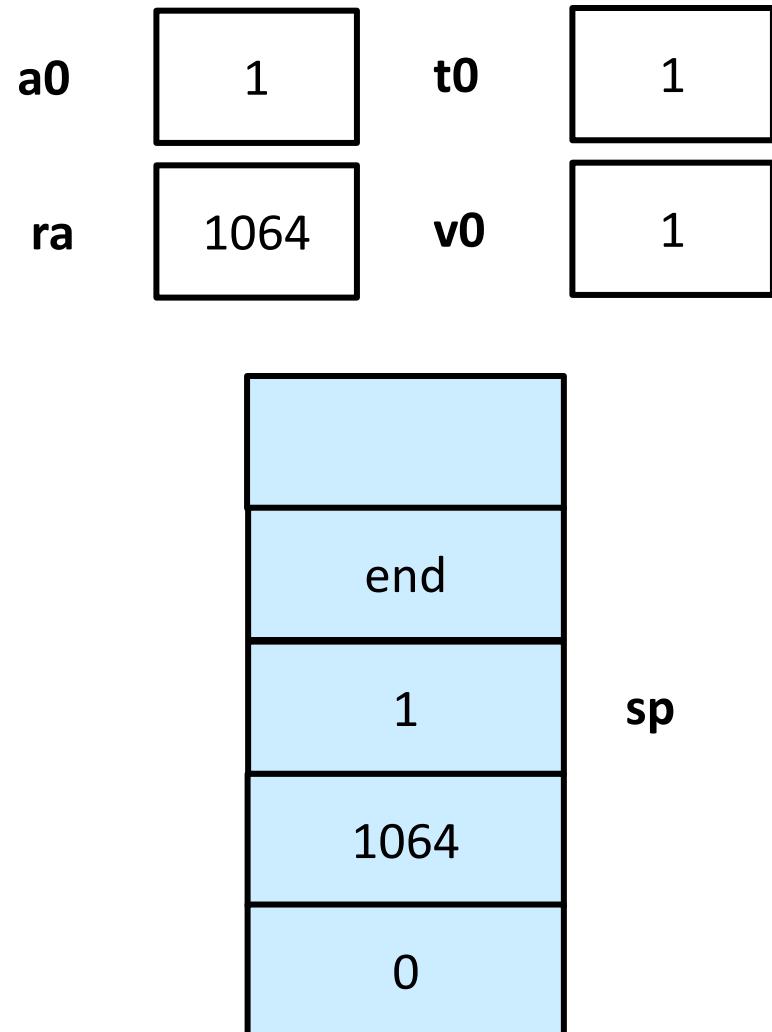


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp) ←
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

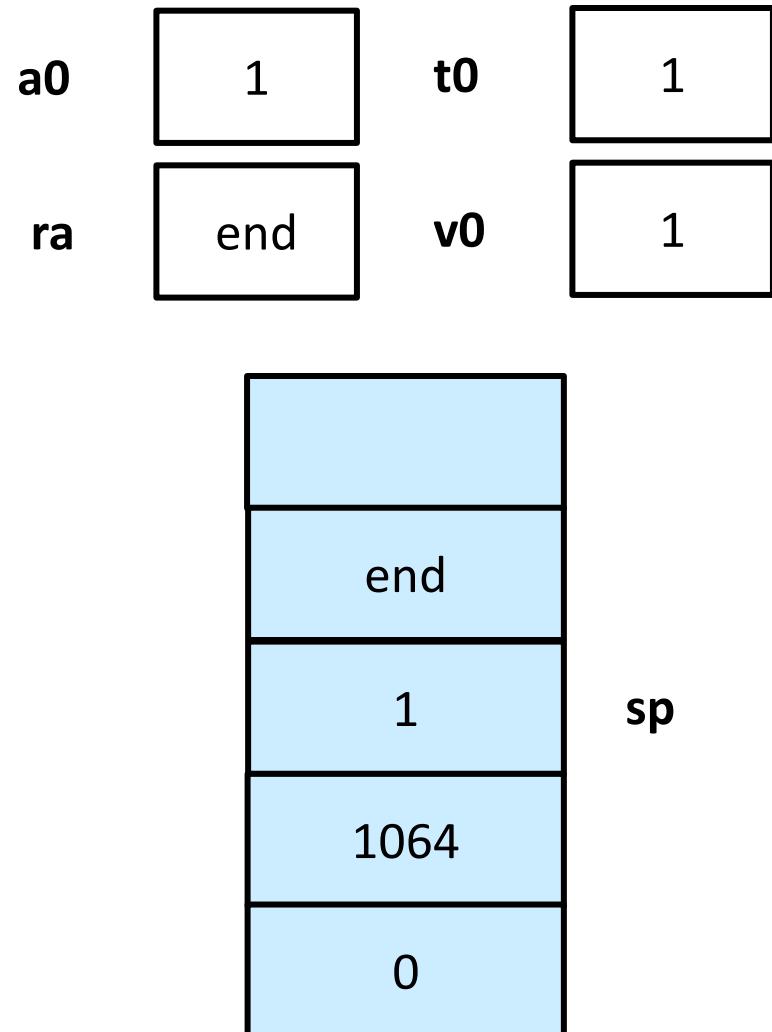


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

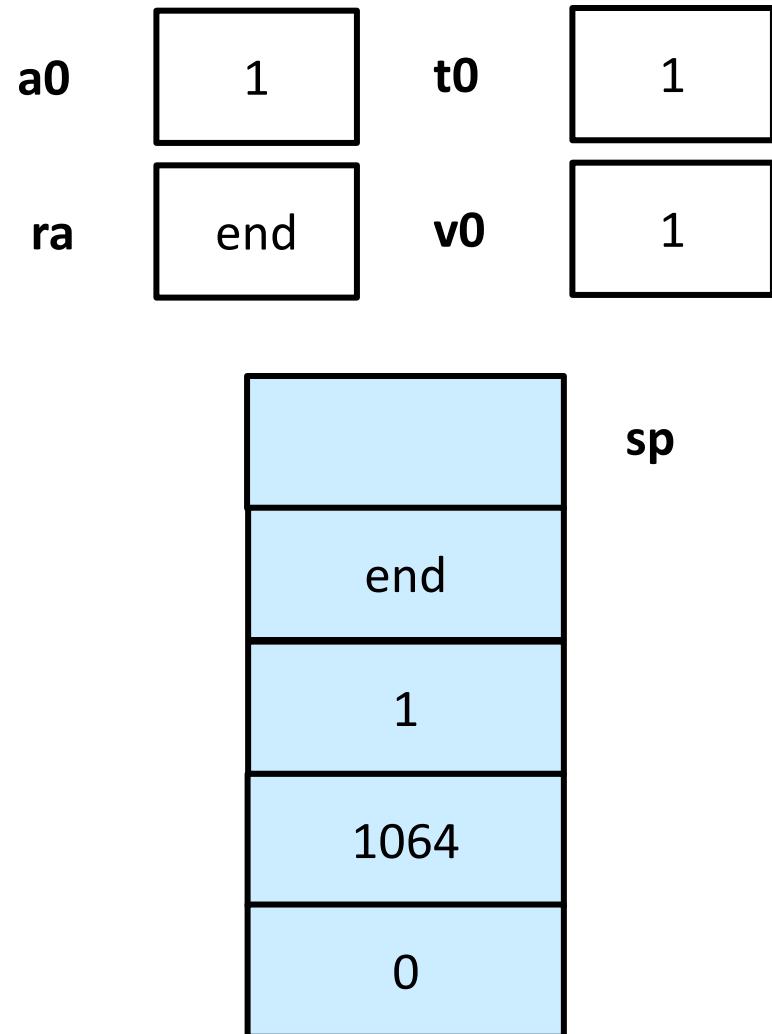


# Exemplo: Fatorial

**fact(1)**

```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```

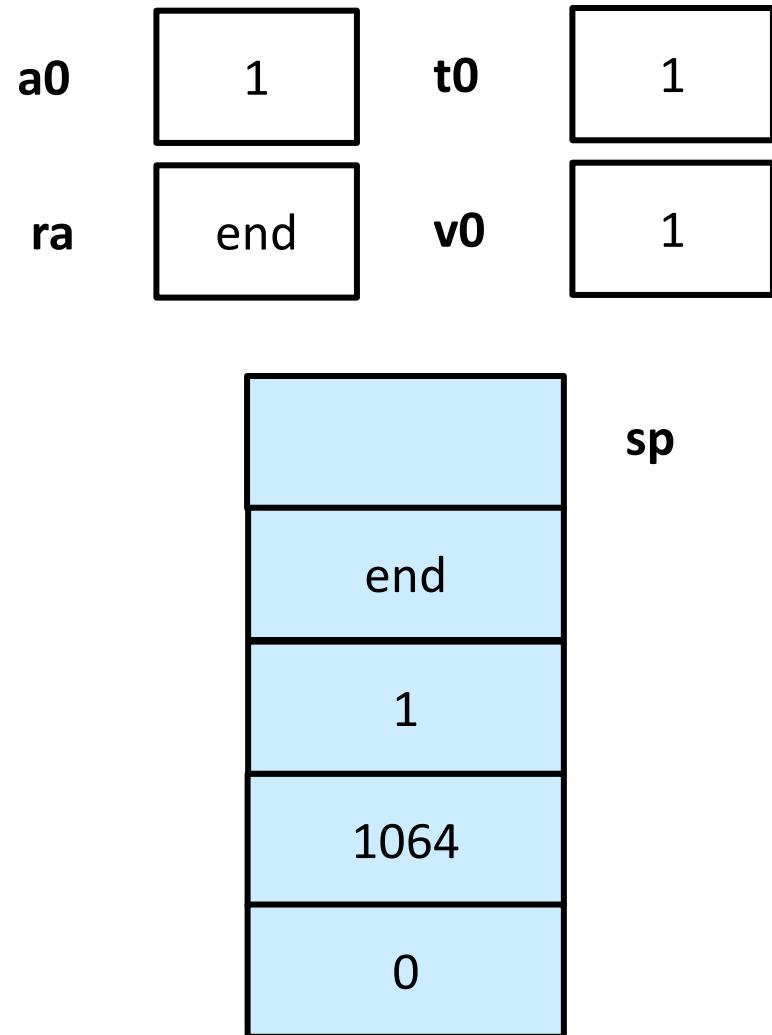


# Exemplo: Fatorial

**fact(1)**

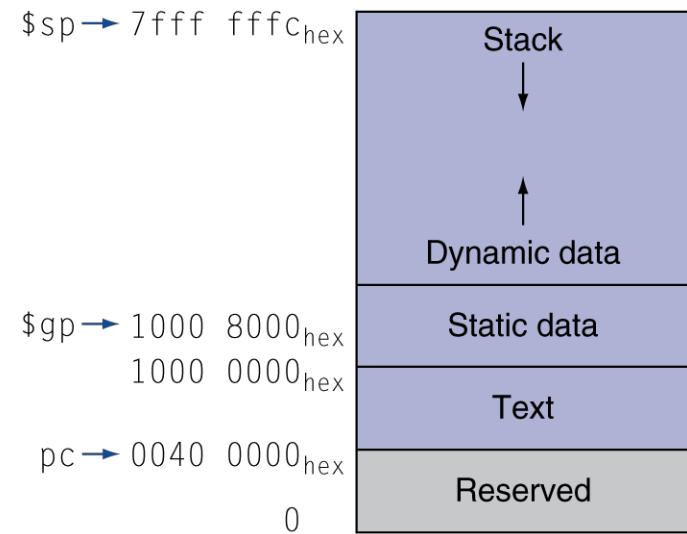
```

1024 fact:addi $sp, $sp, -8
1028     sw    $ra, 4($sp)
1032     sw    $a0, 0($sp)
1036     slti $t0, $a0, 1
1040     beq   $t0, $zero, L1
1044     addi  $v0, $zero, 1
1048     addi  $sp, $sp, 8
1052     jr    $ra
1056 L1: addi $a0, $a0, -1
1060     jal   fact
1064     lw    $a0, 0($sp)
1068     lw    $ra, 4($sp)
1072     addi $sp, $sp, 8
1076     mul   $v0, $a0, $v0
1080     jr    $ra
  
```



# Organização da Memória

- Segmento de texto (text) : código
- Dados estáticos (static data): variáveis globais
  - Variáveis estáticas
  - No MIPS, registrador **\$gp** guarda o início do segmento
- Dados dinâmicos : heap
  - Exs: malloc em C, new em Java
- Pilha (Stack)
  - Variáveis locais
  - Registradores



# Chamada de Subrotina em Outras Arquiteturas

## ■ Instruções:

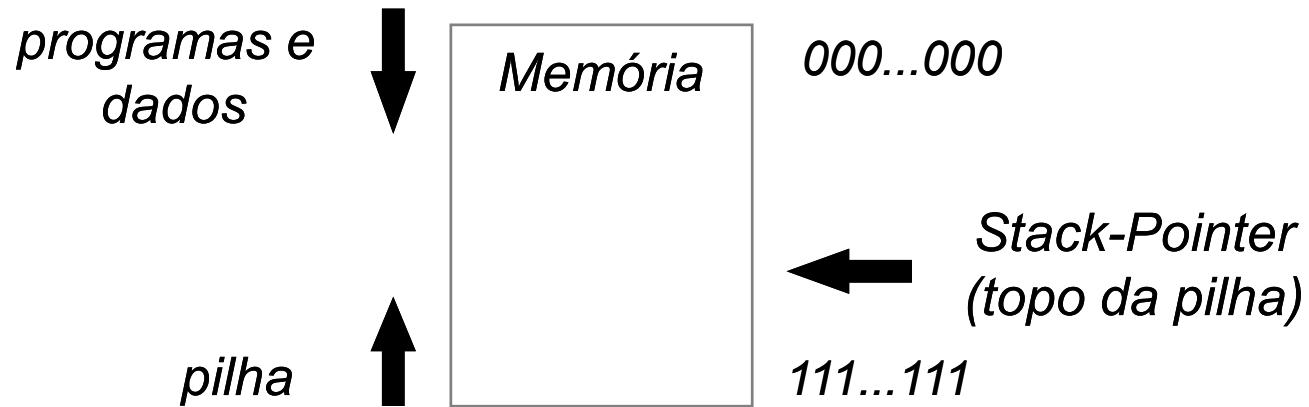
- call
  - empilha endereço de retorno
  - muda fluxo de controle
- ret
  - recupera endereço de retorno

## ■ Outras instruções de suporte...

- Salvar todos registradores na pilha
- alokar parte da pilha para armazenar variáveis locais e parâmetros

# Usando a Pilha em Outras Arquiteturas

- Utiliza parte da memória como pilha



- SP: Registrador adicional
- Instruções adicionais:
  - push reg: decrementa SP, mem(SP) reg ;
  - pop reg: reg mem(SP), incrementa SP;

# MIPS vs. Outras arquiteturas

- Endereço de retorno:
  - MIPS: registrador
  - Outras: Memória
  - Melhor desempenho
- Acesso à Pilha:
  - MIPS: instruções lw e sw
  - Outras: instruções adicionais
  - Menor complexidade na implementação
  - Compilador mais complexo
- Chamadas aninhadas ou recursivas
  - MIPS: implementada pelo compilador
  - Outras: suporte direto da máquina
  - Compilador mais complexo

# Modos de Endereçamento do MIPS

- Modo de endereçamento se refere às maneiras em que instruções de uma arquitetura especificam a localização do operando
  - Onde e como pode ser acessado
- No MIPS, operandos podem estar em:
  - Registradores
  - Memória
  - Na própria instrução

# Endereçamento de Registrador

## ■ Operações aritméticas:

- O operando está em um registrador e a instrução contem o número do registrador

**add R3 ,R3 ,R7**

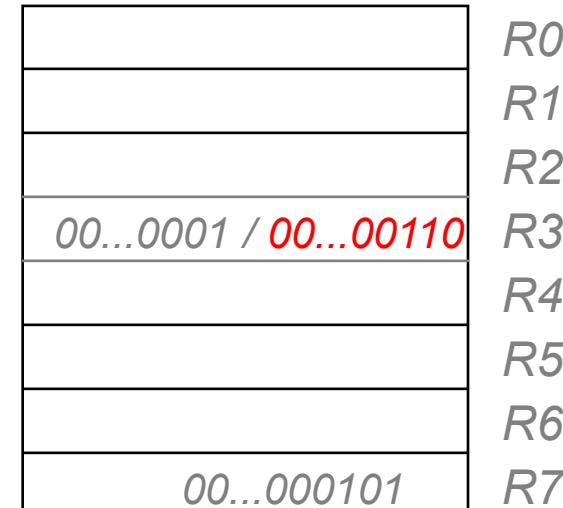


**R3 <- R3 + R7**

### Instrução

00101010	00011	00111	00011
----------	-------	-------	-------

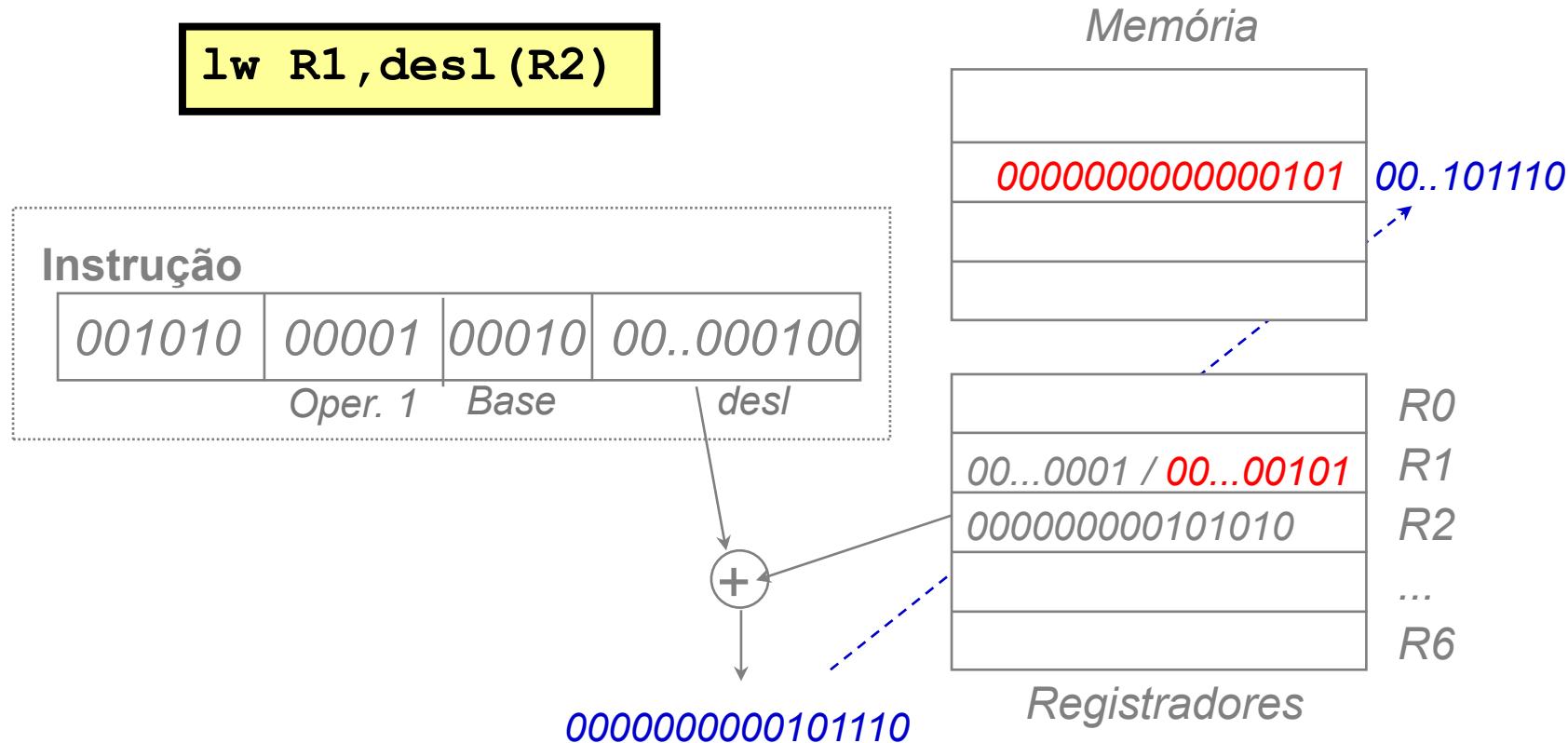
Opcode      Oper. 1    Oper.2      Destino



### Registradores

# Endereçamento Base

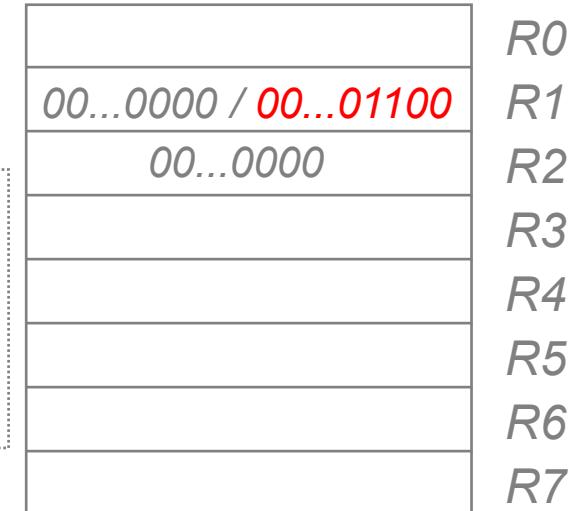
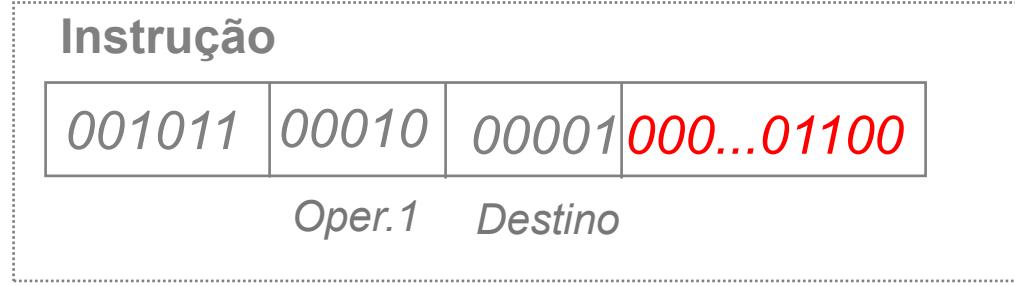
- Instruções de acesso à memória:
  - Instrução:deslocamento
  - Registrador de base:end- inicial



# Endereçamento imediato

- Operações aritméticas e de comparação:
  - O operando é especificado na instrução

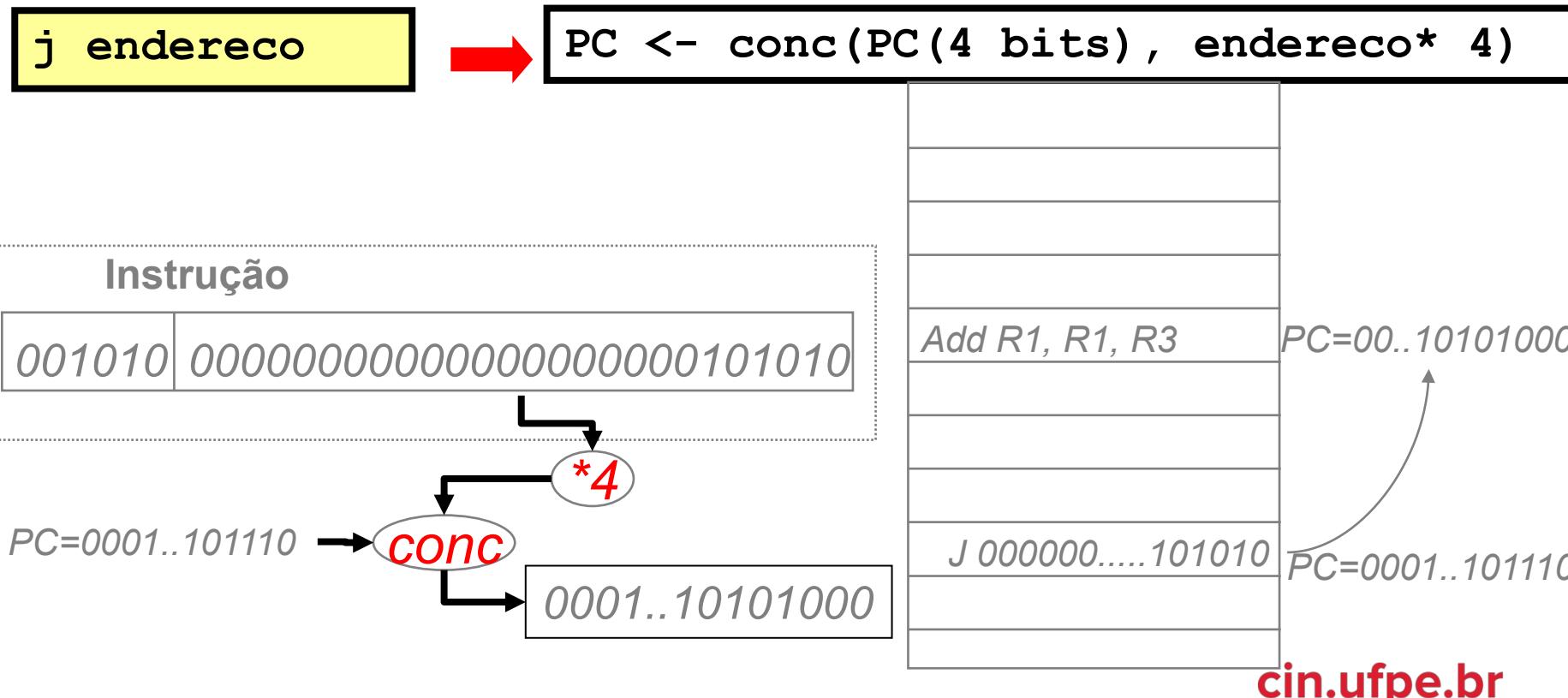
**addi R1 ,R2 , 12**



*Registradores*

# Endereçamento (Pseudo)Direto

- Instrução de Desvio Incondicional:
  - o (pseudo)endereço da próxima instrução (endereço da palavra) é especificado na instrução
  - 4 bits mais significativos do PC são concatenados ao endereço especificado multiplicado por 4



# Endereçamento Relativo a PC

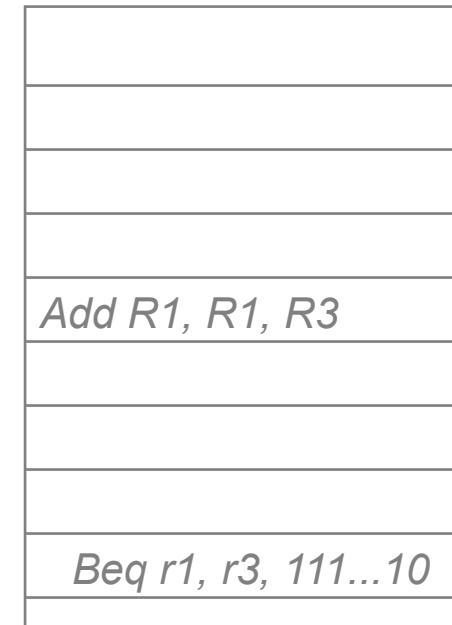
- Instrução de Desvio Condicional:
  - o número de instruções a serem puladas a partir da instrução é especificado na instrução

**beq R1 ,R2 ,desl**



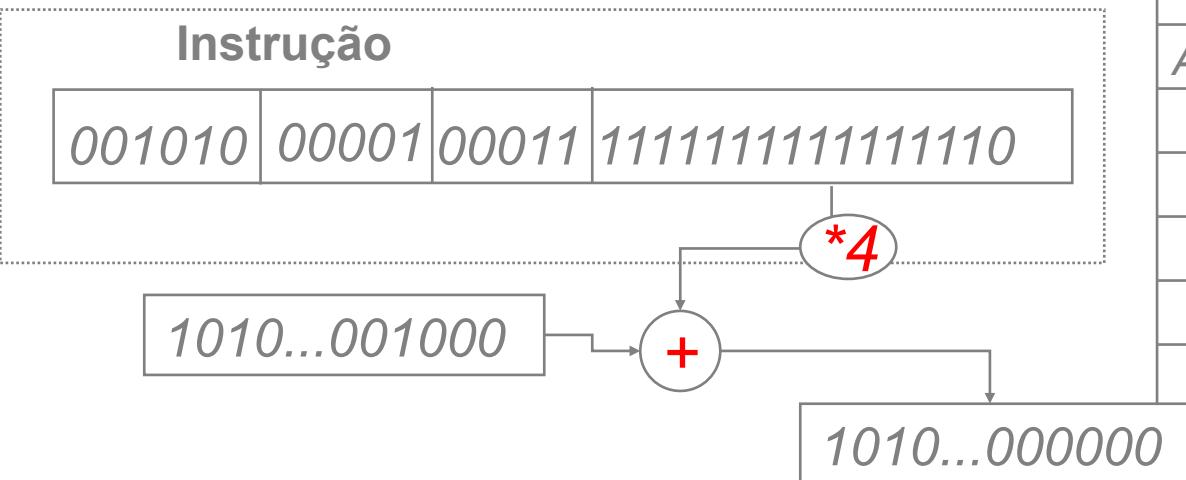
**PC <- PC + desl \* 4**

*Memória*



PC=1010...0000

PC=1010...1000



# Resumo dos Modos de Endereçamento do MIPS

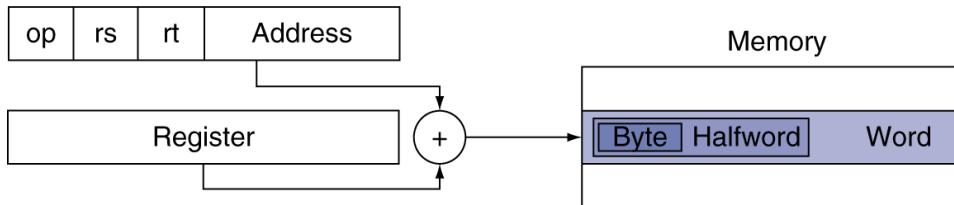
## 1. Immediate addressing



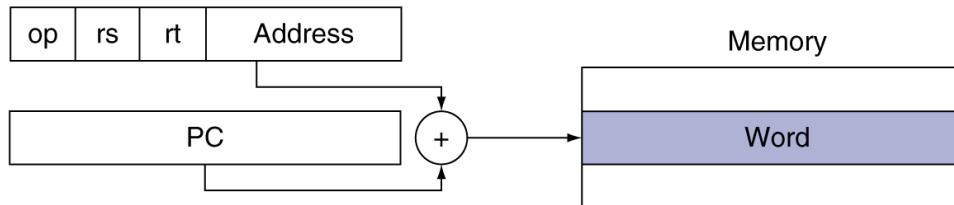
## 2. Register addressing



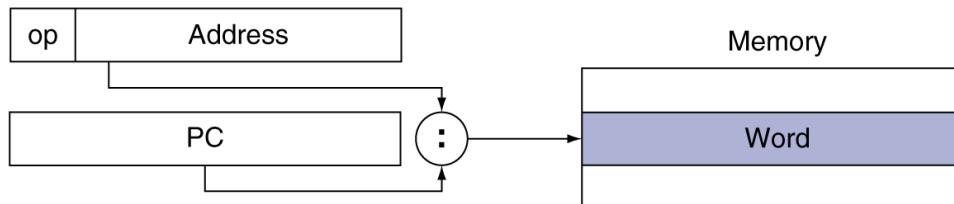
## 3. Base addressing



## 4. PC-relative addressing



## 5. Pseudodirect addressing



# Infraestrutura de Hardware

Instruindo um Computador – Suporte a Subrotinas

Prof. Adriano Sarmento

# Subrotinas

- Subrotinas são utilizadas para estruturar um programa
  - Facilita entendimento
  - Aumenta reuso de código
  - Exs: Procedimentos, funções e métodos
- Chamada de subrotina, faz com que programa execute as instruções contidas na subrotina
- Ao término da execução de uma subrotina, computador deve executar instrução seguinte à chamada de subrotina

# Seis Etapas da Execução de uma Subrotina

1. Rotina que faz a chamada coloca argumentos em um lugar onde a subrotina chamada pode acessá-los  
Passagem de argumentos
2. Rotina transfere controle para a subrotina
3. Subrotina adquire os recursos de armazenamento necessários
4. Subrotina executa suas instruções
5. Subrotina (quando é o caso) coloca o valor do resultado em um lugar que rotina pode acessá-lo
6. Subrotina retorna controle à rotina

# Seis Etapas da Execução de uma Subrotina

1. Rotina que faz a chamada coloca argumentos em um lugar onde a subrotina chamada pode acessá-los  
Passagem de argumentos
2. Rotina transfere controle para a subrotina
3. Subrotina adquire os recursos de armazenamento necessários
4. Subrotina executa suas instruções
5. Subrotina (quando é o caso) coloca o valor do resultado em um lugar que rotina pode acessá-lo
6. Subrotina retorna controle à rotina

# Passagem Argumentos

## Ling. alto nível

```
int media(int w, int x, int y, int z) {  
    ...  
    (corpo da função)  
    ...  
}  
/* Programa principal */  
int main() {  
    int m;  
    m = media(2,3,6,2);  
    ...  
}
```

← Passando Argumentos

- No MIPS, 4 registradores são destinados para armazenar argumentos
  - \$a0 - \$a3 – números 4 a 7

# Seis Etapas da Execução de uma Subrotina

1. Rotina que faz a chamada coloca argumentos em um lugar onde a subrotina chamada pode acessá-los  
Passagem de argumentos
2. Rotina transfere controle para a subrotina
3. Subrotina adquire os recursos de armazenamento necessários
4. Subrotina executa suas instruções
5. Subrotina (quando é o caso) coloca o valor do resultado em um lugar que rotina pode acessá-lo
6. Subrotina retorna controle à rotina

# Transferência de Controle Para Subrotina

## Ling. alto nível

```
int media(int w, int x, int y, int z) {  
    ...  
    (corpo da função) ←  
    ...  
}  
/* Programa principal */  
int main() {  
    int m;  
    m = media(2,3,6,2); →  
    ....  
}
```

Executa  
primeira  
instrução da  
subrotina

Controle deve passar  
para subrotina...  
mas como?

Retorno após  
a chamada

O endereço de  
retorno deve ser salvo...  
mas onde?

# Instrução Para Chamada de Subrotinas

- MIPS oferece uma instrução para fazer a chamada a subrotina
  - **Jump And Link**
- Instrução para chamar a subrotina possui um operando:
  - Label da subrotina

```
jal label
```

- Instrução pula para endereço inicial da subrotina e salva endereço de retorno (instrução após chamada)
  - **\$ra – return address** (número 31) – registrador que armazena endereço de retorno
  - Armazena **PC + 4**

# Seis Etapas da Execução de uma Subrotina

1. Rotina que faz a chamada coloca argumentos em um lugar onde a subrotina chamada pode acessá-los  
Passagem de argumentos
2. Rotina transfere controle para a subrotina
3. **Subrotina** adquire os recursos de armazenamento necessários
4. **Subrotina** executa suas instruções
5. **Subrotina** (quando é o caso) coloca o valor do resultado em um lugar que **rotina** pode acessá-lo
6. Subrotina retorna controle à rotina

# Armazenamento e Retorno de Valores

## Ling. alto nível

```

int media(int w, int x, int y, int z) {
    int result; ← Variável local
    result = (w + x + y + z)/4;
    return result; ← Retorna valor
}
/* Programa principal */
int main() {
    int m;
    m = media(2,3,6,2);
    ....
}

```

- Variáveis podem ser salvas em registradores disponíveis
- No MIPS, 2 registradores para valores retornados
  - \$v0 - \$v1 – números 2 a 3

# Seis Etapas da Execução de uma Subrotina

1. Rotina que faz a chamada coloca argumentos em um lugar onde a subrotina chamada pode acessá-los  
Passagem de argumentos
2. Rotina transfere controle para a subrotina
3. Subrotina adquire os recursos de armazenamento necessários
4. Subrotina executa suas instruções
5. Subrotina (quando é o caso) coloca o valor do resultado em um lugar que rotina pode acessá-lo
6. **Subrotina retorna controle à rotina**

# Retorno da Subrotina

Ling. alto nível

```
int media(int w, int x, int y, int z) {  
    int result;  
    result = (w + x + y + z)/4;  
    return result; }  
/* Programa principal */  
int main() {  
    int m;  
    m = media(2,3,6,2);  
    .... }  
    }
```



Controle deve  
voltar à instrução  
após chamada

# Instrução Para Retorno de Subrotinas

- MIPS oferece uma instrução que pode ser utilizado para retornar da subrotina
  - **Jump Register**
- Instrução para retornar da subrotina possui um operando:
  - Registrador que contém um endereço
- Instrução pula para endereço armazenado no registrador
  - No caso de retorno de subrotina, o registrador deve ser o \$ra

**jr registrador**

**jr \$ra**

# Formato de Instruções

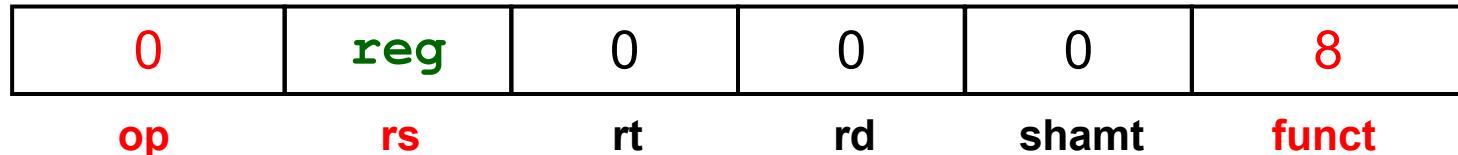
jal label

## Formato J de Instrução



jr reg

## Formato R de Instrução



# Código Assembly

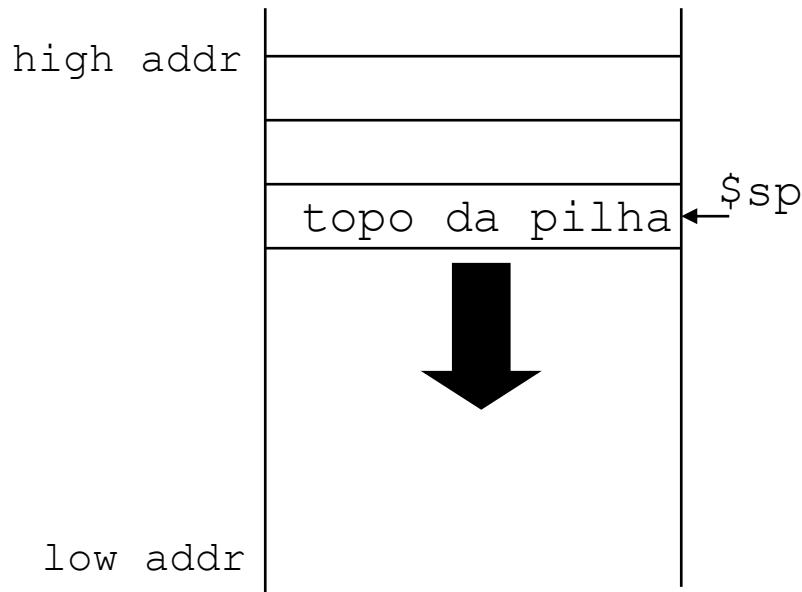
```
→media:add $t0,$a0,$a1 # $t0 = w + x
add $t1,$a2,$a3 # $t1 = y + z
add $v0,$t0,$t1 # $v0 = w + x + y + z
srl $v0,$v0,2 # $v0 = (w + x + y + z)/4
jr $ra #retornando para caller
# Programa principal
main: addi $a0, $zero,2 #$a0 o 1º argumento
addi $a1, $zero,3 #$a1 o 2º argumento
addi $a2, $zero,6 #$a2 o 3º argumento
addi $a3, $zero,2 #$a3 o 4º argumento
jal media #chamando media
→add $s0,$zero,$v0 # $s0 = media(2,3,6,2)
```

# E Se Precisarmos de Mais Registradores?

- É comum, precisar-se em uma subrotina de mais registradores que os específicos para as subrotinas
  - No MIPS:
    - 4 para argumentos e 2 para valores retornados
- Solução: **Pilha!**
- No MIPS, registradores que estão em uso fora da subrotina são salvos em uma pilha na memória
- Subrotina utiliza registradores cujos conteúdos foram salvos na pilha
  - Terminada a subrotina os valores antigos dos registradores são restaurados

# Implementando a Pilha

- Utiliza-se parte da memória como pilha
- Pilha cresce do maior para o menor endereço
- Um registrador guarda o endereço do topo
  - No MIPS: **\$sp** - *stack pointer* (número 29)



- **Push**
  - $\$sp = \$sp - 4$
  - Dado inserido no novo  $\$sp$
- **Pop**
  - Dado removido de  $\$sp$
  - $\$sp = \$sp + 4$

# Usando a Pilha – Salvando Registradores

## Código C

```
int media(int w, int x, int y, int z) {  
    int result;  
    result = (w + x + y + z)/4;  
    return result;  
}
```



## Código Assembly MIPS

```
addi $sp,$sp,-12 # reservando lugar para 3 itens  
sw $t1,8($sp) # salvando $t1 para uso posterior  
sw $t0,4($sp) # salvando $t0 para uso posterior  
sw $s0,0($sp) # salvando $s0 para uso posterior
```

# Usando a Pilha – Executando Subrotina

## Código C

```
int media(int w, int x, int y, int z) {  
    int result;  
    result = (w + x + y + z)/4;  
    return result;  
}
```



## Código Assembly MIPS

```
add $t0,$a0,$a1 # $t0 = w + x  
  
add $t1,$a2,$a3 # $t1 = y + z  
  
add $s0,$t0,$t1 # $s0 = w + x + y + z  
  
srl $s0,$s0,2    # $s0 = (w + x + y + z)/4  
  
add $v0,$s0,$zero # $v0 = $s0
```

# Usando a Pilha – Restaurando Registradores e Retornando

## Código C

```
int media(int w, int x, int y, int z) {  
    int result;  
    result = (w + x + y + z)/4;  
    return result;  
}
```



## Código Assembly MIPS

```
lw $s0,0($sp) # restaurando $s0  
lw $t0,4($sp) # restaurando $t0  
lw $t1,8($sp) # restaurando $t1  
addi $sp,$sp,12 # ajustando topo da pilha  
jr $ra          # retornando a quem chamou
```

# Infraestrutura de Hardware

Instruções lógicas e de desvio no MIPS

**Prof. Adriano Sarmento**

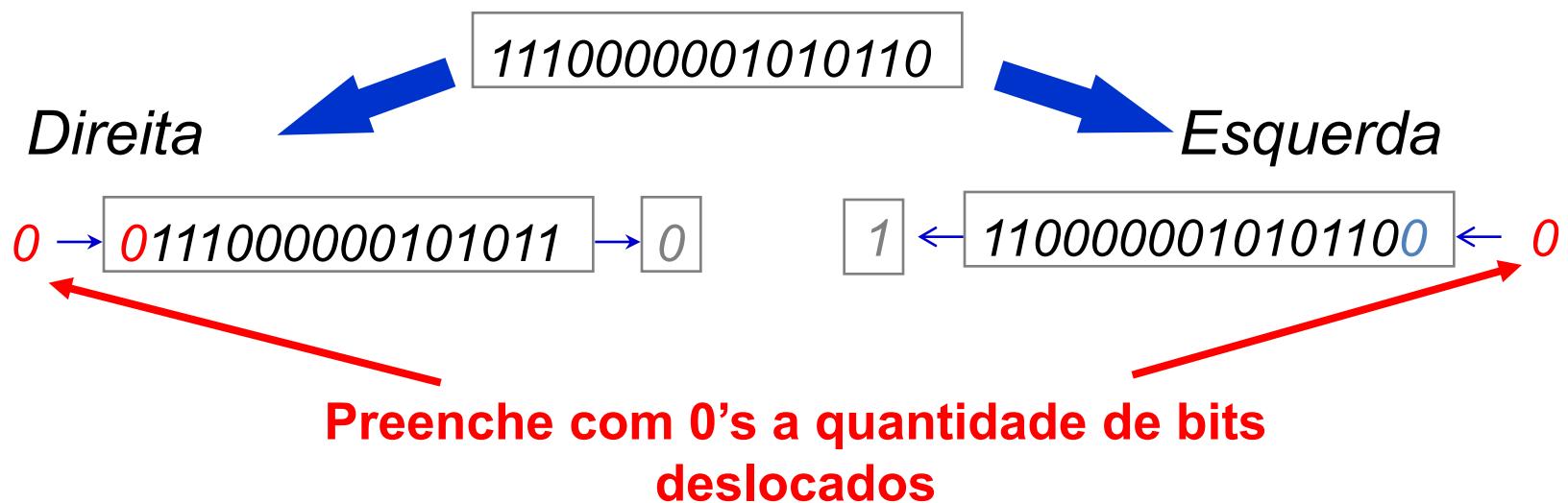
# Operações Lógicas

- Permitem manipulação bit a bit dos dados
- Úteis para extrair ou inserir um grupo de bits em uma palavra
  - Podem modificar o formato de um dado

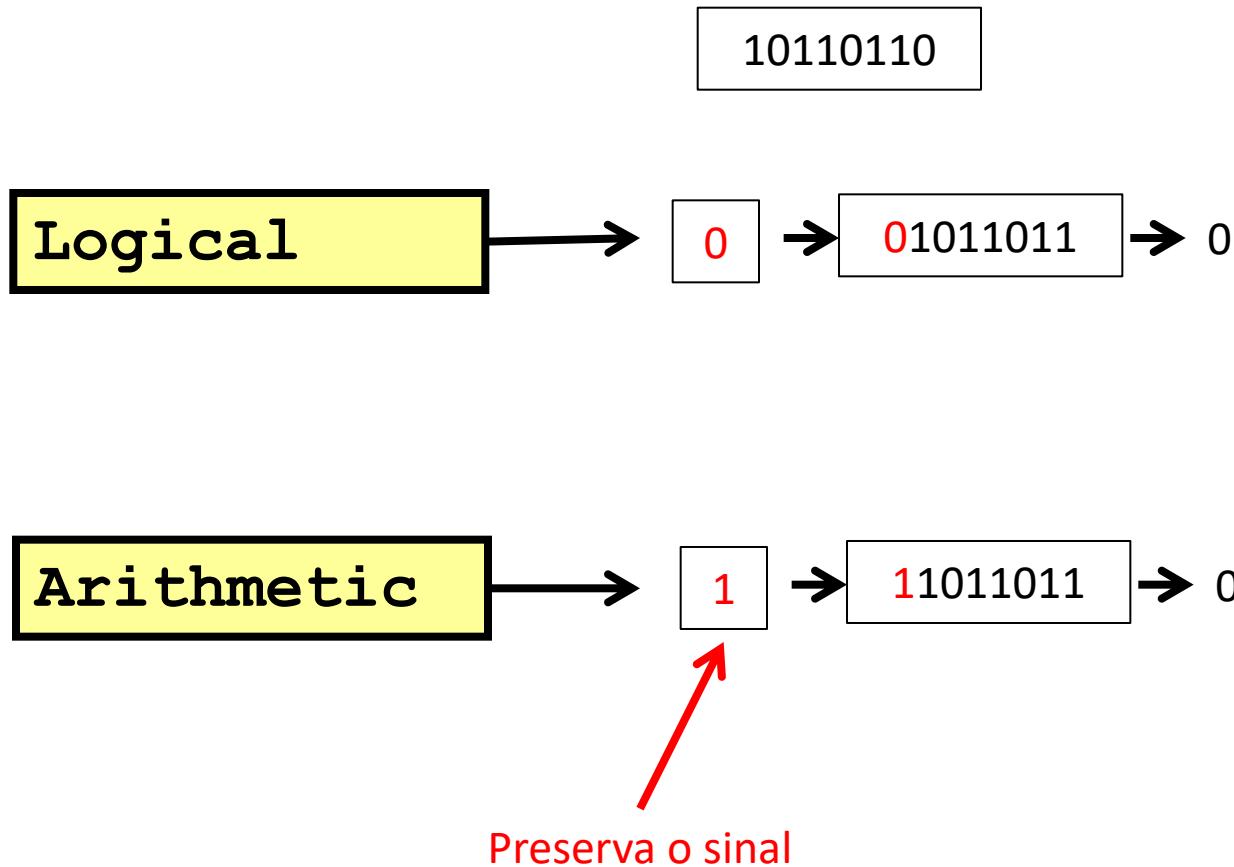
Operação	C	Java	MIPS
Logical Shift left	<<	<<	sll
Logical Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	a nor 0

# Operações Lógicas de Deslocamento (Shift)

- Afeta a localização dos bits em um dado
- Permite o deslocamento para esquerda ou direita de bits de um dado
- Insere grupo de bits no dado

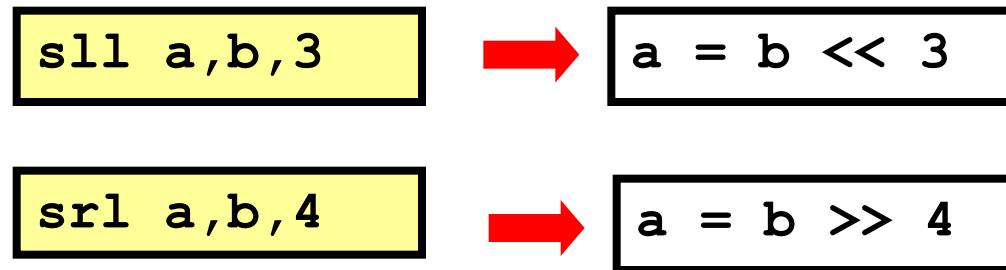


# Logical Shift Right x Arithmetic Shift Right



# Operações Lógicas de Deslocamento no MIPS

- Instruções de deslocamento no MIPS possuem 3 operandos:
  - destino, fonte, quantidade de bits deslocados



- Deslocamento para esquerda (direita) de  $i$  bits de um valor é equivalente a multiplicar (dividir) valor por  $2^i$

# Multiplicação com SLL

- Multiplicação do valor por 4 ( $2^2$ )

Código C

```
g = g * 4;
```



g em \$s1

Código Assembly MIPS

```
sll $s1, $s1, 2
```

g armazena valor 4  
( $100_2$ )

```
00000000000000100
```

g armazena valor 16  
( $10000_2$ )

```
0000000000010000
```

# Formato da Instrução SLL e SRL

## Formato R de Instrução

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op      opcode da instrução

rs      não utilizado para **sll** e **srl**

rt      registrador que contém operando fonte

rd      registrador destino que contém resultado

shamt    shift amount, quantidade de bits deslocados

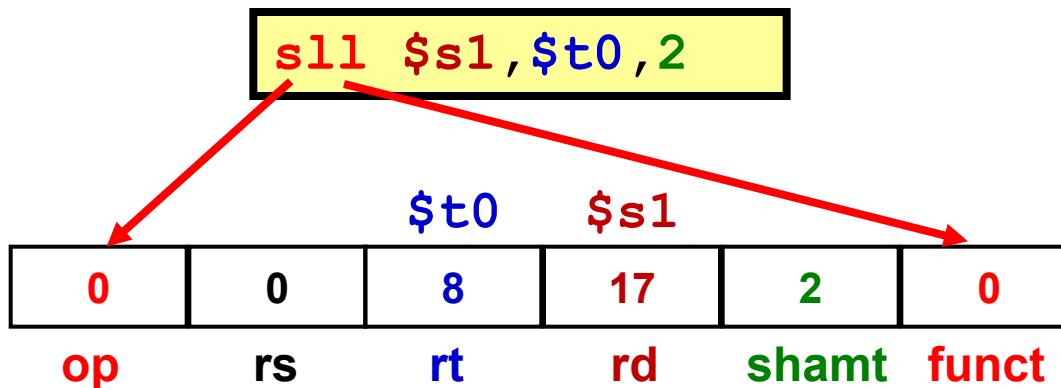
funct    função (function) que estende o opcode

# Representando SLL na Máquina

- Número dos registradores

- \$s0 - \$s7: 16 - 23
- \$t0 - \$t7 : 8-15
- \$t8-\$t9 : 24-25

## Código Assembly MIPS



# Outras Operações Lógicas

- AND ,OR, NOT (MIPS implementa como  $A \text{ NOR } 0$ )
- Úteis para extrair grupos de bits
  - “Máscara” para encontrar padrões de disposição de bits
- No MIPS, possui 3 operandos como (ADD) e tem formato R

and a,b,c



a = b & c

and \$t0,\$t1,\$t2

\$t2 0000 0000 0000 0000 0000 1101 1100 0000

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 0000 0000 0000 0000 0000 1100 0000 0000

# Operações de Controle de Fluxo

- Alterar a sequência de execução das instruções:

*Ling. alto nível*

- *If ...then ...else*
- *case*
- *loop*
- *go to*



*Linguagem máquina*

- *Desvio condicional a comparações entre variáveis e/ou valores*
- *Desvio incondicional*

# Desvios no MIPS

- Consegue-se através de dois tipos de instruções
  - **Branch** (desvio condicional)
  - **Jump** (desvio incondicional)
- Instruções do tipo *branch* tem 3 operandos
  - fonte 1, fonte 2, label de instrução

**beq a ,b ,L1**

se (**a ==b**) desvie para **L1**

**bne a ,b ,L2**

se (**a !=b**) desvie para **L2**

- Instruções do tipo *jump* tem um único operando, o label

**j L3**

desvie para **L3**

# Implementando Desvio Condicional - if

## Código C

```
if (i == h)
    i = g + h;
else
    i = g - h;
```

g em \$s1, h em  
\$s2, i em \$s3

Assembler calcula  
endereço



## Código Assembly MIPS

```
bne $s3, $s2, Else
add $s3, $s1, $s2
j Exit
Else: sub $s3,$s1,$s2
Exit:...
```

# Implementando Loops - while

## Código C

```
while (save[i] == h)  
    i += 1;
```

h em \$s2, i em  
\$s3 e endereço  
base de save  
em \$s4



## Código Assembly MIPS

```
Loop: sll $t1,$s3,2  
      add $t1, $t1, $s4  
      lw $t0,0($t1)  
      bne $t0,$s2,Exit  
      addi $s3, $s3,1  
      j Loop  
Exit:...
```

← \$t1= i \* 4  
← End\_base + \$t1

# Mais Sobre Branchs no MIPS

- O operando relativo ao label nas instruções de branch corresponde na verdade ao deslocamento em relação ao endereço da instrução contida no PC (Program Counter)
  - PC já incrementado de 4!
  - $PC = PC + (\text{deslocamento} * 4)$  se  $\text{reg1} == \text{reg2}$

```
beq rs,rt,deslocamento
```

## Formato I de Instrução

op	rs	rt	deslocamento
6 bits	5 bits	5 bits	16 bits

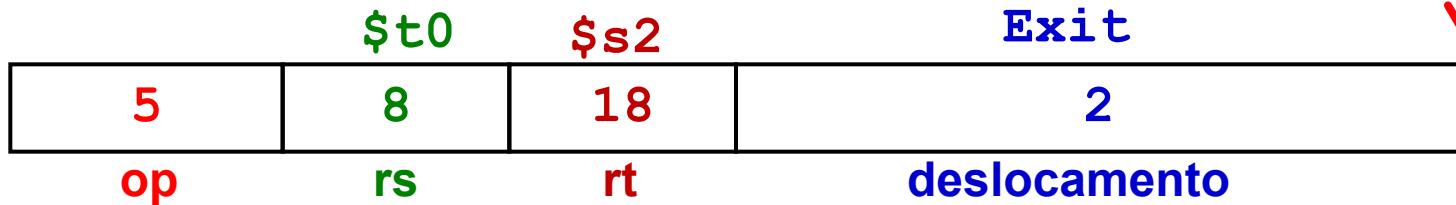
# Representando BNE na Máquina

- Número dos registradores

- \$s0 - \$s7: 16 - 23
- \$t0 - \$t7 : 8-15
- \$t8-\$t9 : 24-25

```

Loop: sll $t1,$s3,2
      add $t1, $t1, $s4
      lw $t0,0($t1)
      bne $t0,$s2,Exit
      addi $s3, $s3,1
      j Loop
      Exit:...
    
```



	\$t0	\$s2	Exit
op	5	8	2
rs			
rt			
deslocamento			

# Mais Sobre Jumps no MIPS

- O operando relativo ao label nas instruções de jumps corresponde na verdade ao endereço (número) da instrução a ser executada
  - **PC = endereço**

j **endereço**

## Formato J de Instrução

op	endereço
6 bits	26 bits

# Mais Operações Condicionais no MIPS

- Armazene 1 se condição é verdade, senão 0
  - Set less than

```
slt rd,rs,rt
```

se (**rs < rt**) **rd = 1**, senão **rd = 0**

- MIPS possui registrador que armazena valor 0
  - \$zero
- **slt** pode ser utilizada junto com **beq**, **bne**

```
slt $t0,$s1,$s2
```

```
bne $t0,$zero, L1
```

se (**\$s1 < \$s2**) desvie para L1

# Mais Operações Condicionais no MIPS

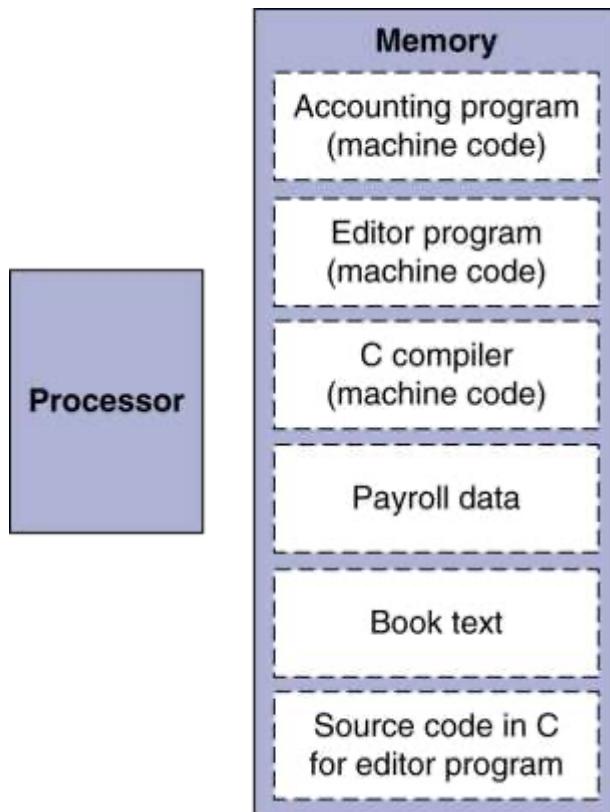
- Nenhuma instrução de desvio para <, >=, ... ????
  - Combinar em uma só instrução branch e comparações
  - ( >, <, >= ...) , requer mais trabalho por instrução
  - Clock mais lento
  - Penaliza todas as instruções
  - Branch se == ou != é o mais comum

# Infraestrutura de Hardware

Instruindo um processador real

**Prof. Adriano Sarmento**

# Computadores de Programas Armazenados

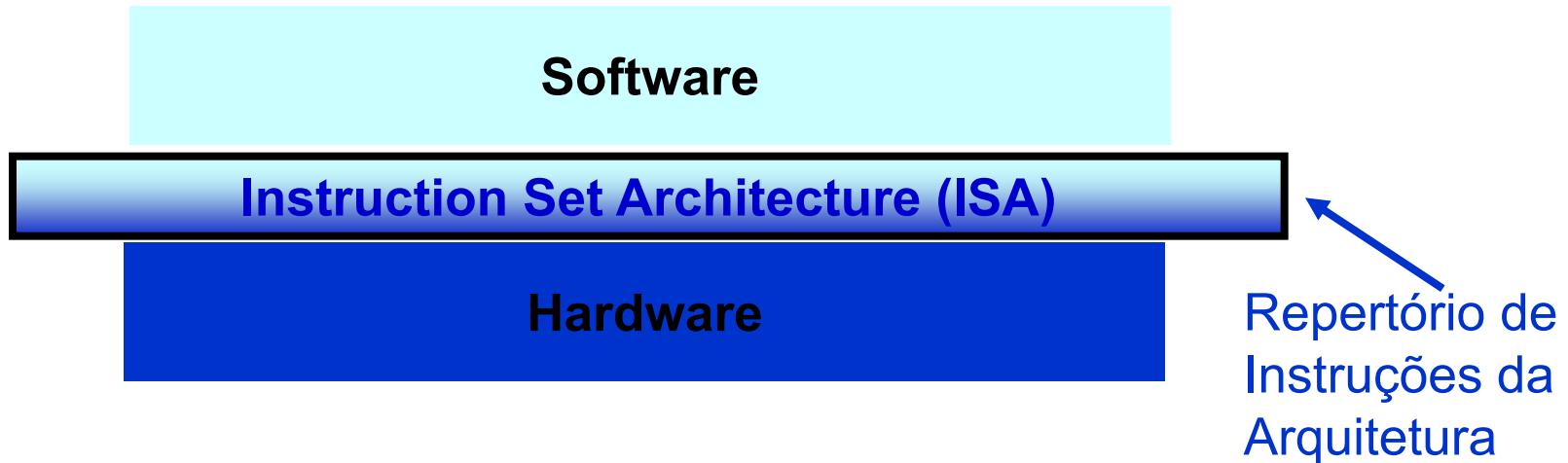


- Instruções e dados representados como números binários
- Instruções e dados armazenados em memória
- Programas podem interagir com outros programas
  - Ex: compiladores, linkers, ...
- Compatibilidade de formatos binários permite que programas compilados funcionem em diferentes computadores
  - ISAs padronizadas

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Interface HW/SW: Repertório de Instruções da Arquitetura



- Última abstração do HW vista pelo SW
- Computadores diferentes podem ter diferentes ISAs
  - Mas com muitos aspectos em comum
- Visando portabilidade de código, indústria se alinha em torno de quantidade pequena de ISAs diferentes

# Evolução de ISAs

- Até metade da década de 60 computadores tinham ISAs com quantidade reduzida de instruções e instruções simples
  - Simplifica implementação
- Fim da década de 60 surge ISAs com grande número de instruções complexas
  - **Complex Instruction Set Computer (CISC)**
  - Difícil implementação e existência de muitas instruções pouco usadas
- Começo da década de 80 ISAs com instruções simples voltam a ser comuns
  - **Reduced Instruction Set Computer (RISC)**

# Exemplos de Processadores CISC e RISC

## ■ CISC

- Intel x86, Pentium, AMDx86, AMD Athlon
- Muito utilizados em PCs



## ■ RISC

- MIPS, SPARC, ARM, PowerPC
- Muito utilizados em sistemas embarcados



## ■ Tendência hoje é termos processadores híbridos

- Ideias de RISC foram incorporados a CISC e vice-versa



# Repertório (ISA) do Processador MIPS

- Utilizado como exemplo nesta disciplina
- Desenvolvido no começo de 80, é um bom exemplo de uma arquitetura RISC
- Muito utilizado no mercado de sistemas embarcados
  - Aplicações em eletrônicos diversos, equipamento de rede/armazenamento, câmeras, impressoras, blue-rays, smart watches, etc

# Princípios de Projeto do MIPS (RISC)

- Simplicidade é favorecida pela regularidade
  - Instruções de tamanho fixo
  - Poucos formatos de instruções
  - Opcode sempre utiliza os primeiros 6 bits
- Quanto menor, mais rápido
  - Repertório de instruções limitados
  - Quantidade de registradores limitados
  - Número reduzido de modos de endereçamento
- Torne rápido o caso mais comum
  - Existência de instruções que contém operandos
- Bom projeto requer boas escolhas (compromissos)
  - Diferentes formatos de instruções complica decodificação, CONTUDO permite instruções de tamanho fixo

# Categoria e Formato de Instruções do MIPS

- Aritméticas
- Lógicas
- Transferência de dados
- Desvios de fluxo
- Gerenciamento de memória

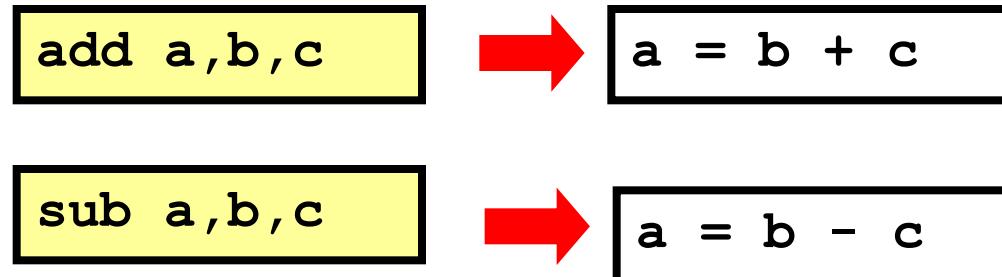
3 Formatos de Instrução: **todos com 32 bits**

op	rs	rt	rd	shamt	funct	R format
op	rs	rt		immediate		I format
op			jump target			J format

**Bom projeto requer boas escolhas (compromissos)**

# Operações Aritméticas

- Maioria das instruções aritméticas no MIPS possuem 3 operandos:
  - destino, fonte 1, fonte 2
- Cada instrução aritmética faz apenas uma operação



A simplicidade é favorecida pela regularidade

# Exemplo: Expressões Aritméticas

## Código C

```
f = (g + h) - (i + j);
```



## Pseudo-código Assembly MIPS

```
add t0,g,h #temp t0 = g + h
add t1,i,j #temp t1 = i + j
sub f,t0,t1 # f = t0 - t1
```

# Operandos nos Registradores

- Para melhorar desempenho, os operandos de uma instrução aritmética devem estar nos registradores
  - Acesso mais rápido em relação à memória
- Todos os registradores no MIPS possuem 32 bits
  - 32 bits é uma **palavra** (word) no MIPS
- Número de registradores no MIPS é reduzido: 32
  - Número grande de registradores pode penalizar desempenho
  - Impacto no tamanho da instrução

**Quanto menor, mais rápido**

# Registradores para Operandos no MIPS

- Nomes assembly dos registradores
  - \$s0, \$s1...\$s7 – armazenam variáveis dos programas
  - \$t0, \$t1...\$t9 – para valores temporários

## Código C

```
f = (g + h) - (i + j);
```



## Código Assembly MIPS

```
add $t0,$s1,$s2 # $s1 = g , $s2 = h
add $t1,$s3,$s4 # $s3 = i ,$s4 = j
sub $s0,$t0,$t1 # $s0 = f = t0 - t1
```

# Representação das instruções

- Informação tem uma representação numérica na base 2
  - Codificação das instruções
  - Mapeamento de nomes de registradores para números
    - \$s0 a \$s7 : 16 a 23
    - \$t0 a \$t7 : 8 a 15
    - \$t8-\$t9 : 24-25

# Formato da Instrução ADD e SUB

## Formato R de Instrução

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op      opcode da instrução

rs      registrador que contém 1º operando fonte (source)

rt      registrador que contém o 2º operando fonte

rd      registrador destino que contém resultado

shamt    shift amount , não utilizado para add e sub

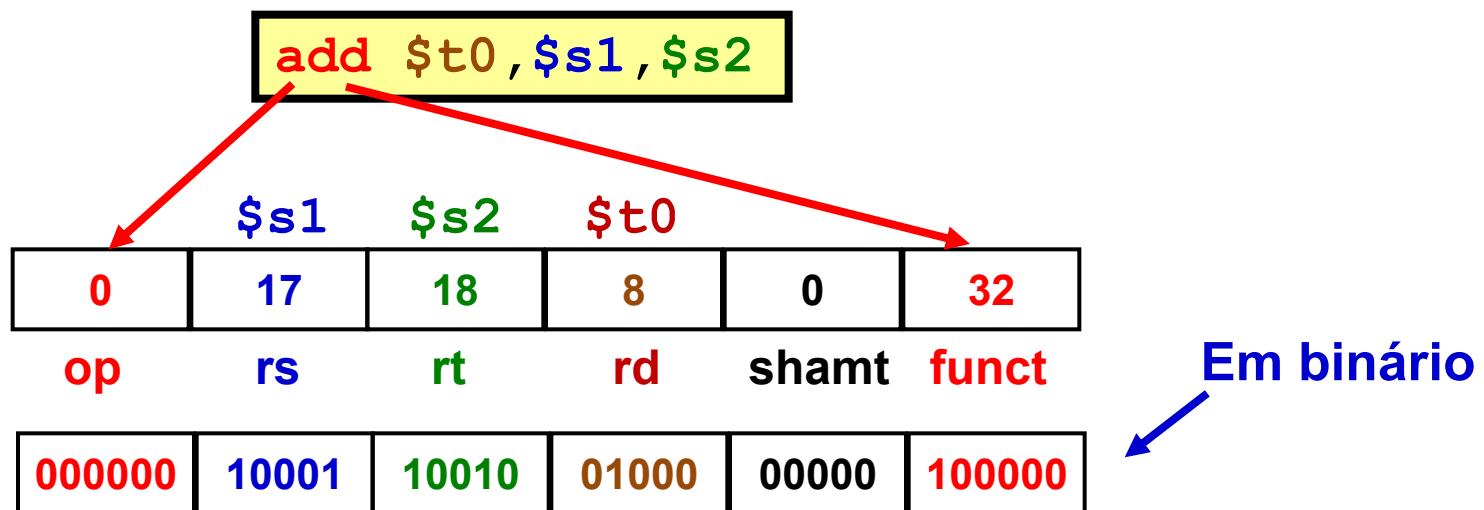
funct    função (function) que estende o opcode

# Representando ADD na Máquina

## ■ Número dos registradores

- \$s0 - \$s7: 16 - 23
- \$t0 - \$t7 : 8-15
- \$t8-\$t9 : 24-25

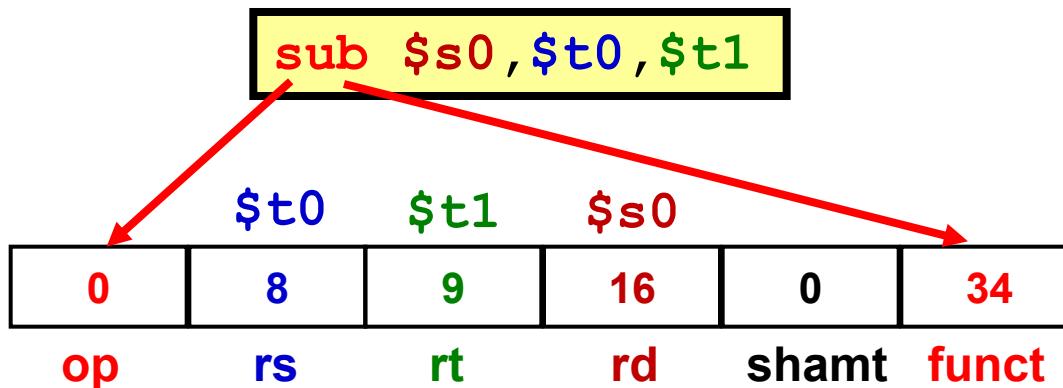
## Código Assembly MIPS



# Representando SUB na Máquina

- Número dos registradores
  - \$s0 - \$s7: 16 - 23
  - \$t0 - \$t7 : 8-15
  - \$t8-\$t9 : 24-25

## Código Assembly MIPS



# Operações de Transferência de Dados (Memória)

- Operações de transferência de dados entre processador e memória no MIPS tem dois operandos:
  - registrador, endereço de memória
  - Endereço de memória formado por um registrador de base (contém endereço inicial) e um deslocamento(offset)
- Duas operações básicas: **lw** (load word), **sw** (store word)

```
lw r, offset(end_inicial)
```

→ Carrega conteúdo de **offset + end\_inicial** no registrador **r**

```
sw r, offset(end_inicial)
```

→ Armazena conteúdo do registrador **r** em **offset + end\_inicial**

# Operandos na Memória

- Memória muito utilizada para armazenar dados compostos
  - Arrays, estruturas, dados dinâmicos
- Memória endereçada por byte (8 bits)
- Contudo memória é vista como uma sequência de palavras de 32 bits
  - **Endereços de palavras devem ser múltiplos de 4!**
- MIPS é Big Endian
  - Byte mais significativo tem menor endereço da palavra
  - Little Endian – byte mais significativo tem o maior endereço

# Endereços da Memória – Big Endian

- MIPS é BigEndian

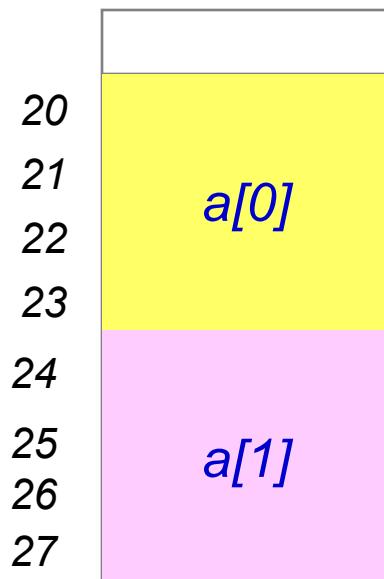
0	0x12
1	0x34
2	0x68
3	0x00

$\$s0 = 0$   
`lw $t0, 0($s0)`  
  
 $\$t0 = 0x12346800$

# Endereços da Memória

## ■ MIPS

- Inteiros de 32 bits (4 bytes)
- Array de inteiros chamado de *a*



$$\begin{aligned}\text{End}(a[0]) &= 20 \\ \text{End}(a[1]) &= 24\end{aligned}$$

$$\text{End}(a[i]) = \text{End-inicial} + i \times 4$$

# Utilizando Arrays no MIPS (Memória)

- Registrador de base guarda endereço inicial do array
  - Offset utilizado como índice do array

## Código C

```
a[12] = h + a[8];
```

h em \$s2,  
endereço base de  
a em \$s3



## Código Assembly MIPS

```
lw $t0,32($s3)  
add $t0,$s2,$t0  
sw $t0,48($s3)
```

Índice 8 requer offset de 32  
(4 x 8)

# Outro Exemplo de Arrays no MIPS

- Array com variável de indexação

## Código C

```
g = g + a[i];
```



g em \$s1, i em  
\$s2, endereço  
base de a em \$s3

## Código Assembly MIPS

```
add $t1, $s2, $s2
add $t1, $t1, $t1
add $t1, $t1, $s3
lw $t0, 0($t1)
add $s1, $s1, $t0
```

End(a [i]) =  
base + i + i + i + i

# Registradores x Memória

- Acesso a registradores é mais rápido
- Utilização da memória requer *loads* e *stores*
  - Mais instruções a serem executadas
- Compilador deve maximizar a utilização de registradores
  - **Otimização de registradores é importante!**

# Formato da Instrução LW e SW

## Formato I de Instrução

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

op            opcode da instrução

rs            registrador que neste caso contém endereço base

rt            registrador fonte ou destino

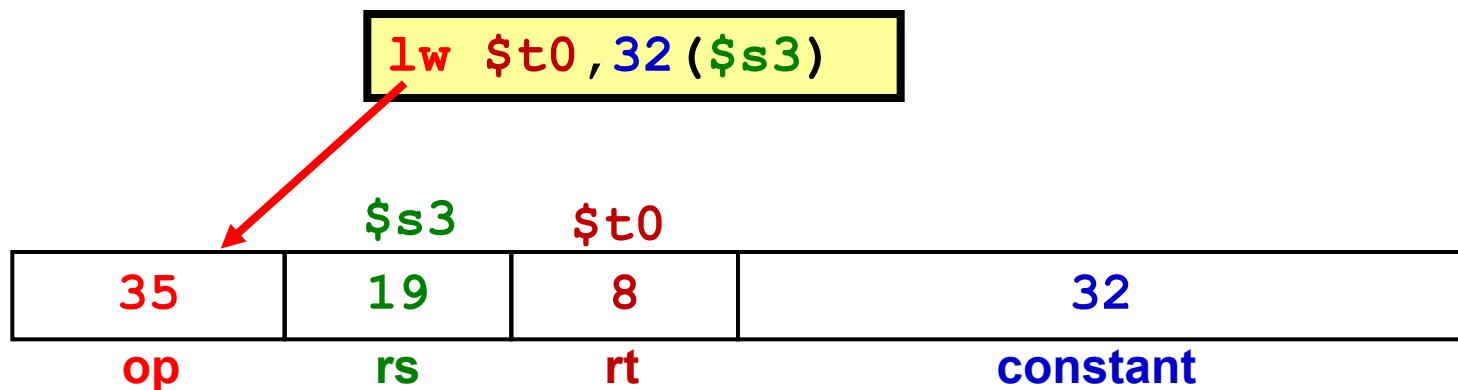
constant    constante que representa o offset

# Representando LW na Máquina

## ■ Número dos registradores

- \$s0 - \$s7: 16 - 23
- \$t0 - \$t7 : 8-15
- \$t8-\$t9 : 24-25

## Código Assembly MIPS

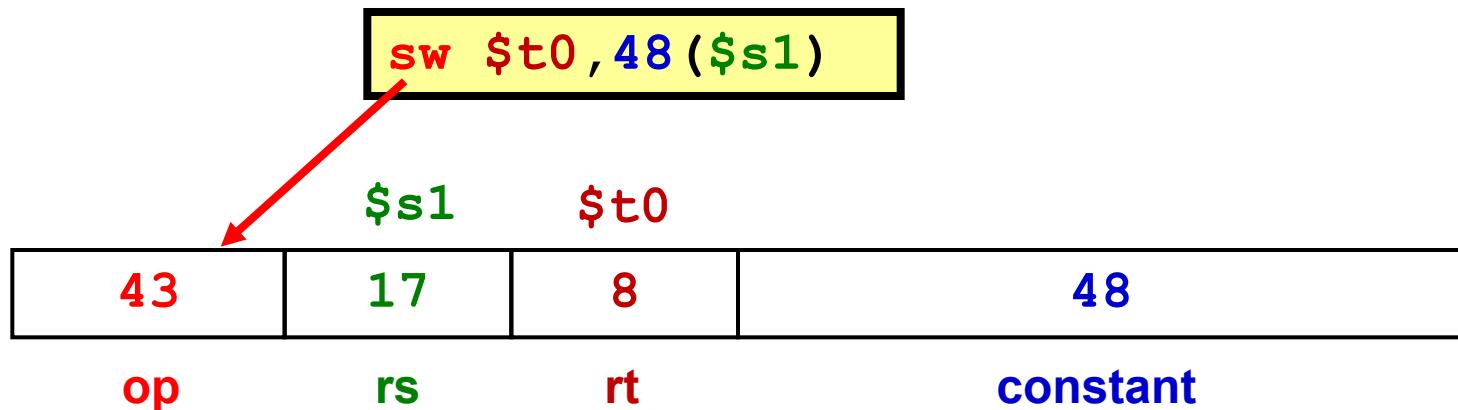


# Representando SW na Máquina

## ■ Número dos registradores

- \$s0 - \$s7: 16 - 23
- \$t0 - \$t7 : 8-15
- \$t8-\$t9 : 24-25

## Código Assembly MIPS



# Operandos Imediatos

- Frequentemente utilizamos pequenas constantes em programas
  - Ex: **a = a + 1;**
- Possíveis soluções
  - Armazenar constantes na memória e depois carregá-las
  - Ter registradores que armazenam a mesma constante
    - MIPS possui o registrador **\$zero** que armazena 0
  - **Ter instruções especiais que contêm constantes!**

# Instruções Imediatas

- MIPS oferece instruções onde uma constante está embutida na própria instrução
- Instruções imediatas contêm 3 operandos:
  - destino, fonte, constante

**addi a,b,2**



**a = b + 2**

- Existe a adição imediata (addi), mas não existe a subtração imediata
  - Subtração : Soma com uma constante negativa

**Torne rápido o caso mais comum**

# Adição Imediata

## Código C

```
a = b + 8;  
a = a - 2;
```

a em \$s1, b em  
\$s2

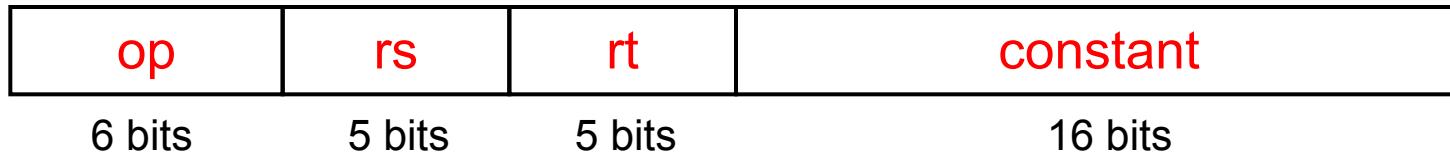


## Código Assembly MIPS

```
addi $s1, $s2, 8  
addi $s1, $s1, -2
```

# Formato da Instrução ADDI

## Formato I de Instrução



op            opcode da instrução

rs            registrador fonte

rt            registrador destino

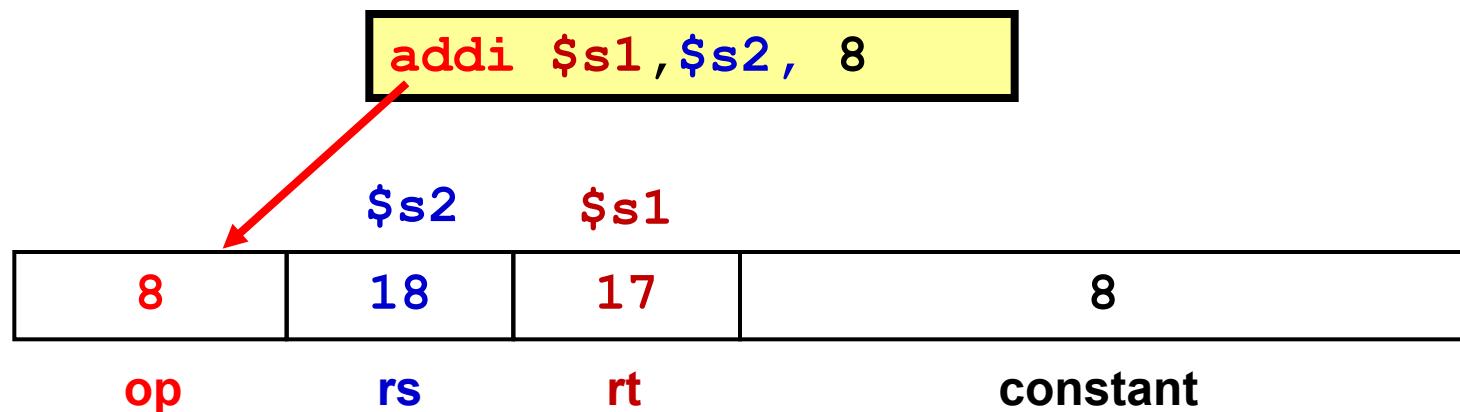
constant    constante embutida na instrução

# Representando ADDI na Máquina

## ■ Número dos registradores

- \$s0 - \$s7: 16 - 23
- \$t0 - \$t7 : 8-15
- \$t8-\$t9 : 24-25

## Código Assembly MIPS



# Infraestrutura de Hardware

**Funcionamento de um computador**

- Executando Instruções

**Prof. Adriano Sarmento**

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

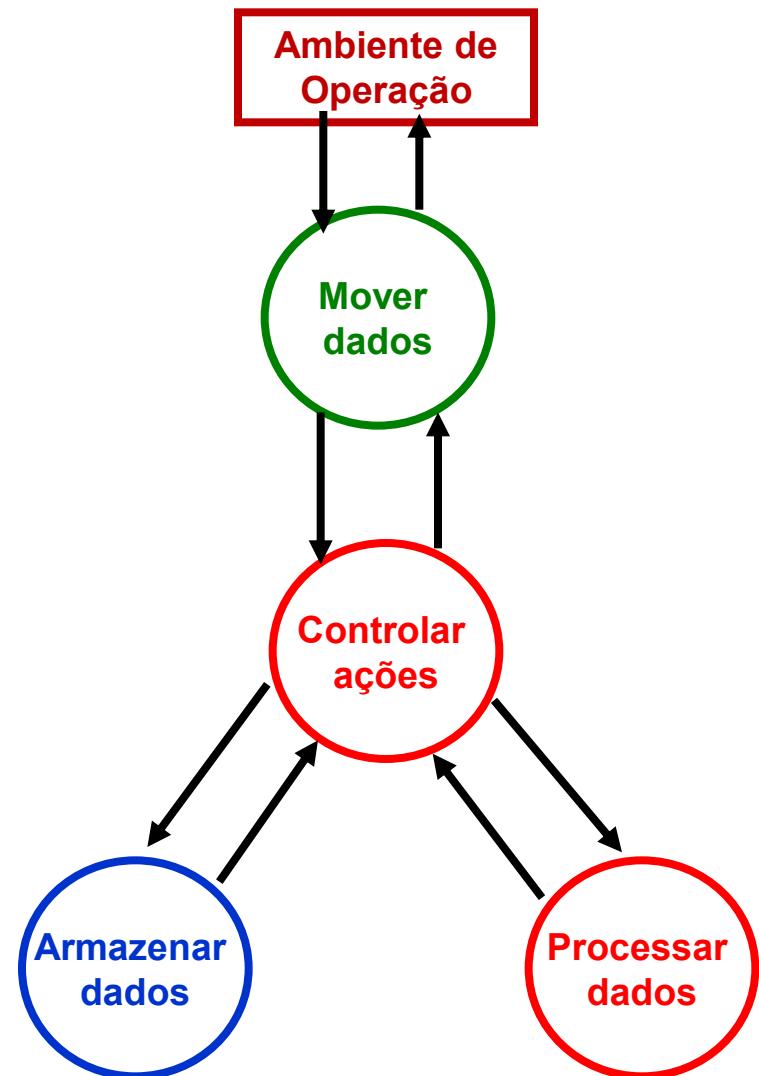
# Interface HW/SW: Repertório de Instruções da Arquitetura



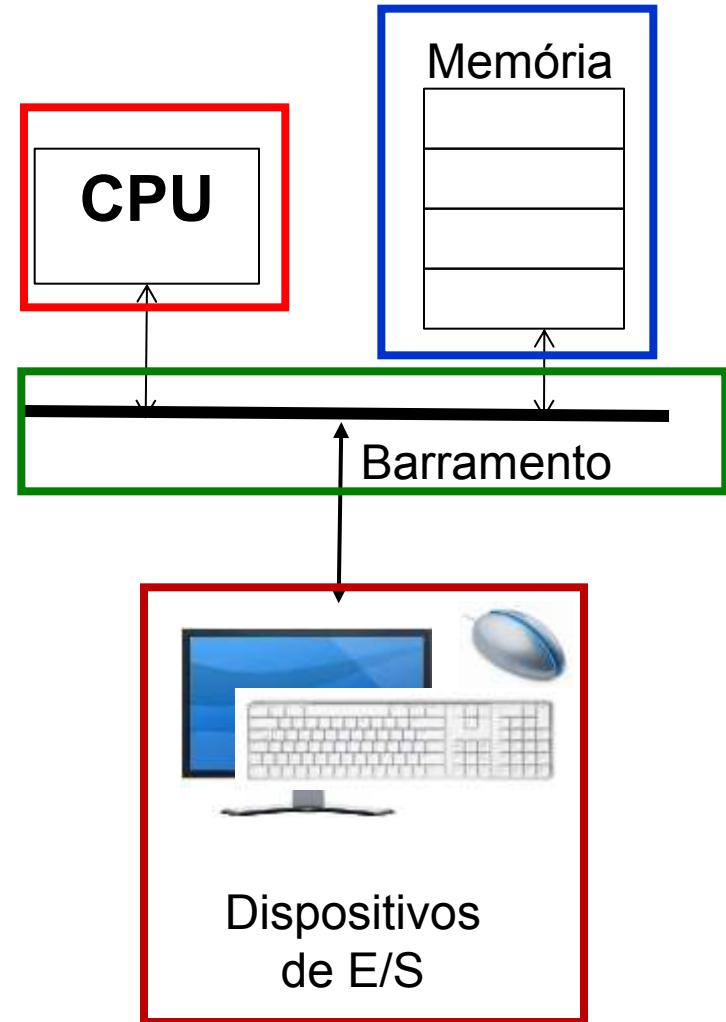
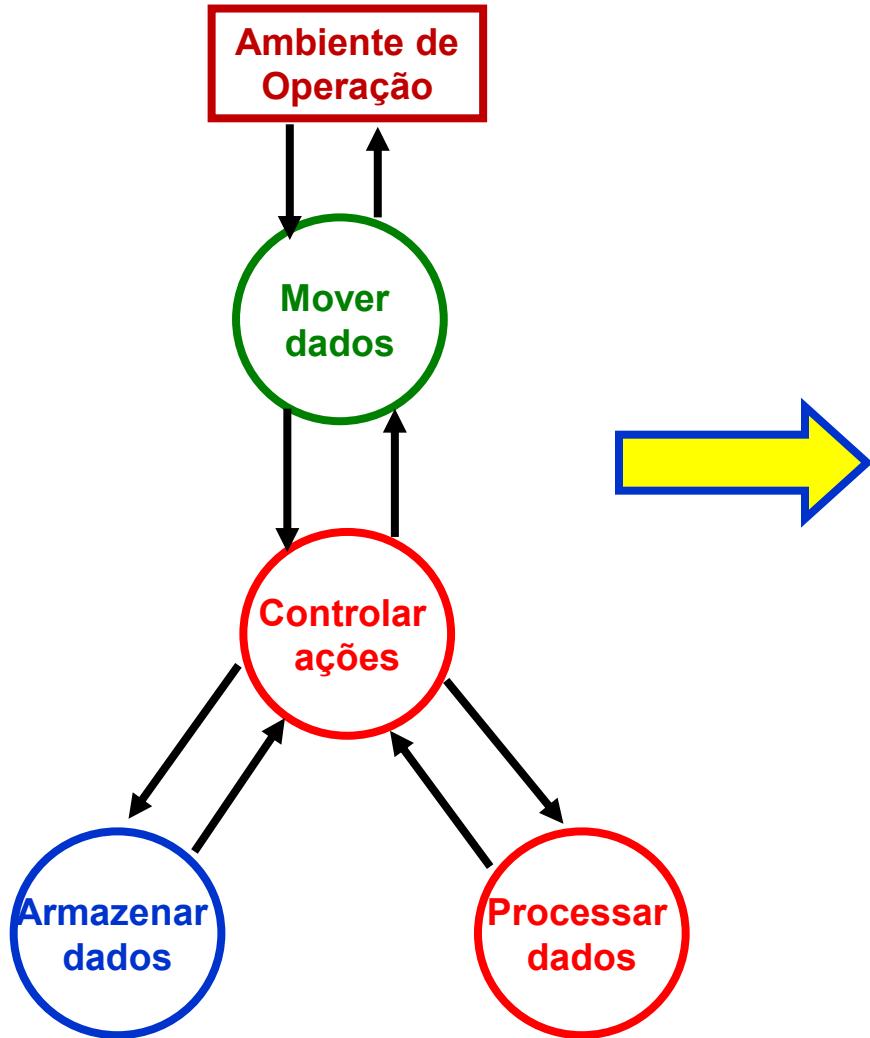
- Última abstração do HW vista pelo SW
- Provê a informação necessária para que se escreva um código em linguagem de máquina (ou montagem) que execute corretamente na arquitetura
  - Instruções, registradores, acesso a memória, entrada/saída, etc

# Visão Funcional de um Computador

- O HW de um computador deve realizar 4 ações:
  - Mover dados
  - Armazenar dados
  - Processar dados
  - Controlar as ações mencionadas



# Mapeando Funcionalidades em um Computador

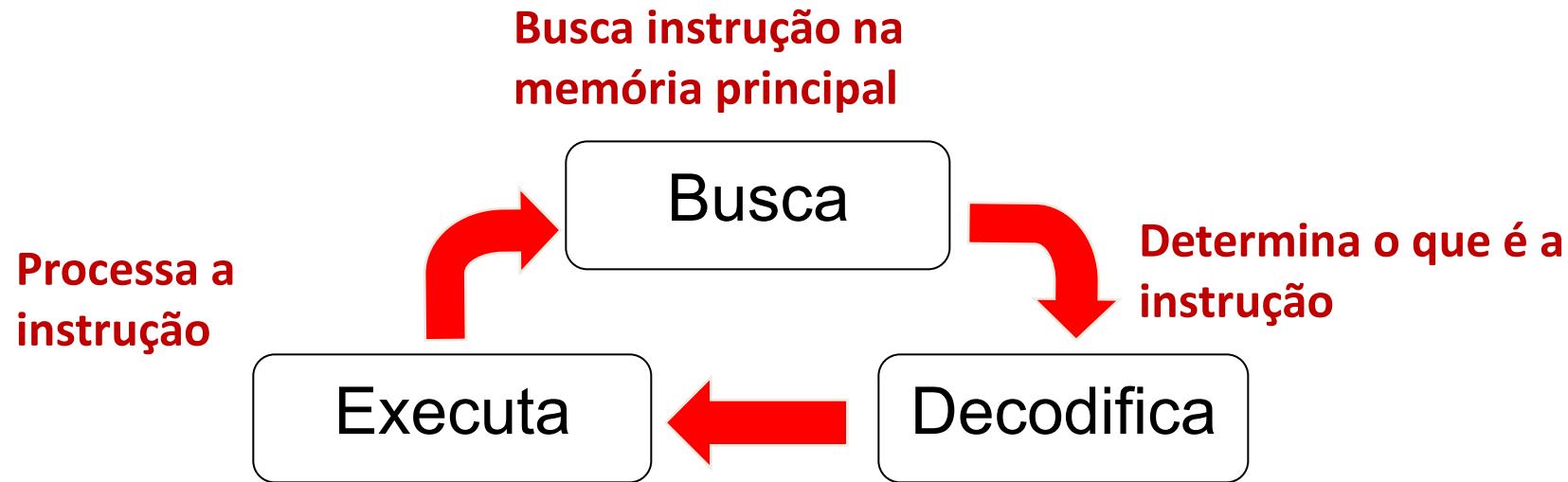


# Como Funciona um Computador?

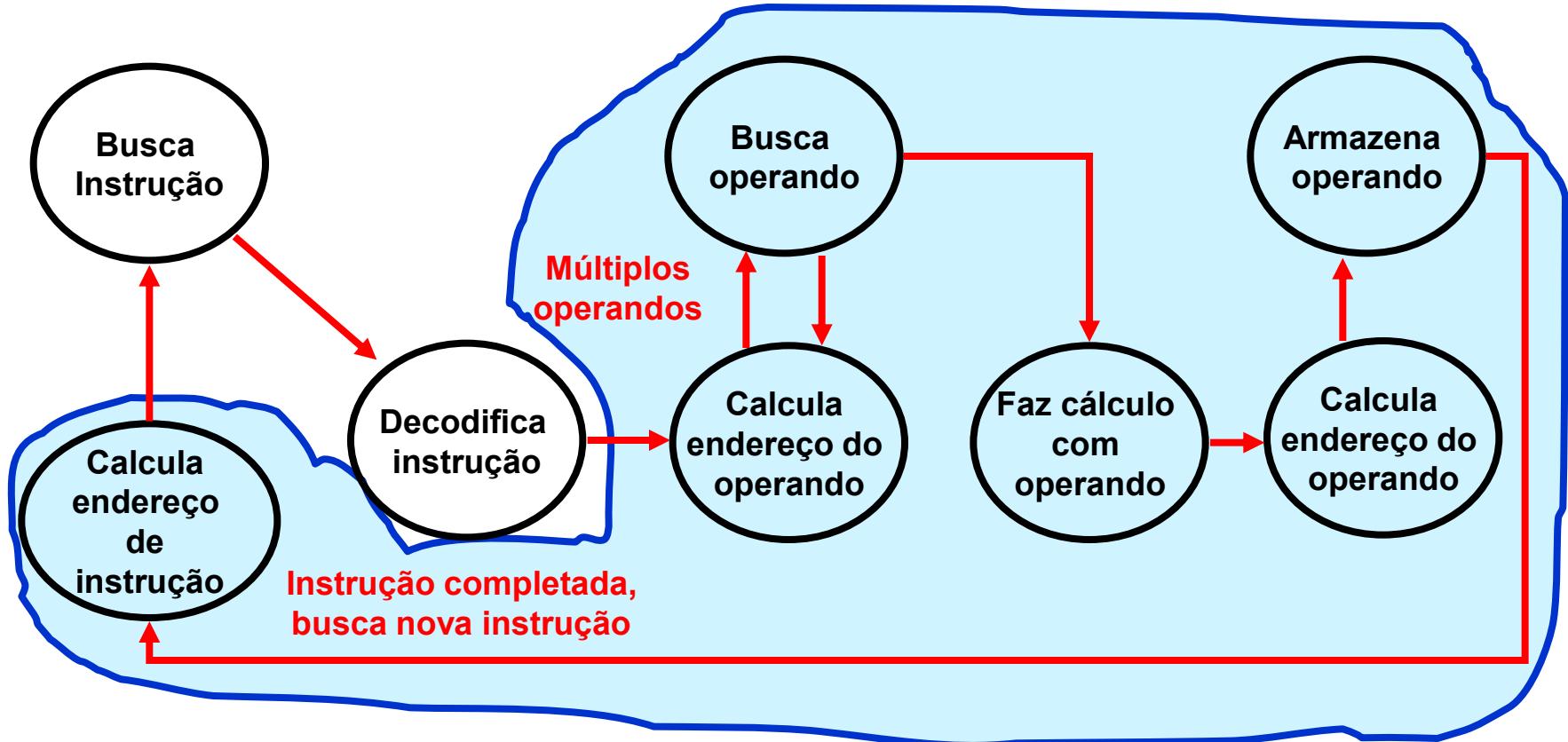
- Conceitos básicos para funcionamento de um computador:
  - Dados e instruções são armazenados na memória
    - Para simplificar, vamos considerar que é uma única memória para instruções e dados
  - Conteúdo da memória é acessado através de um endereço, não importando o tipo de dado armazenado
  - Execução ocorre de maneira sequencial (a não ser que seja explicitamente especificado), uma instrução após a outra

# Visão Simplificada de Processamento de Instrução

- CPU faz continuamente 3 ações:

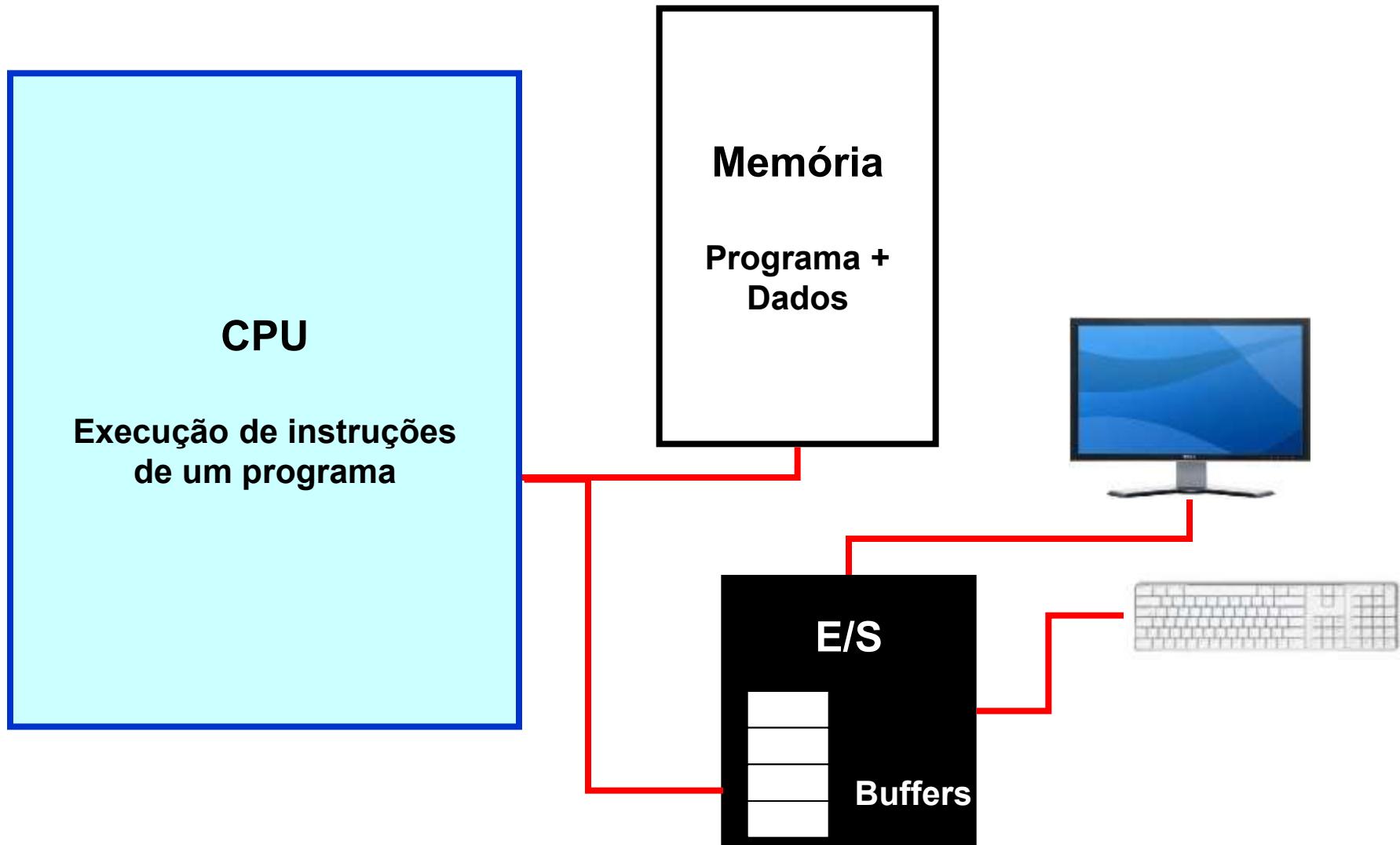


# Visão Detalhada da Execução de uma Instrução

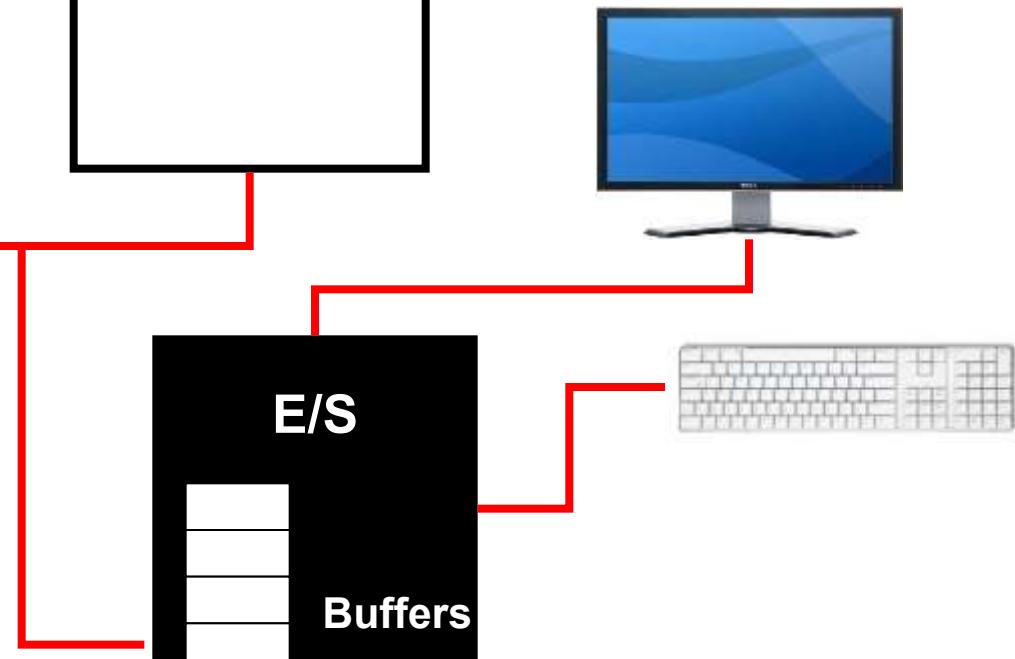
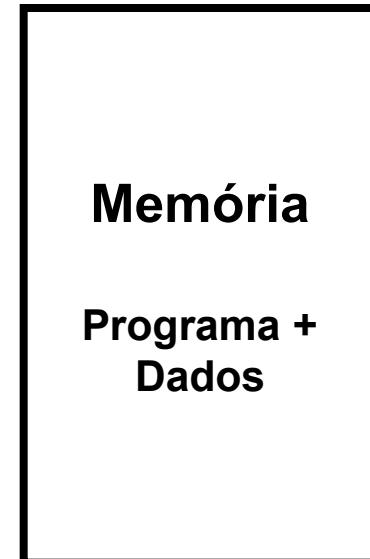
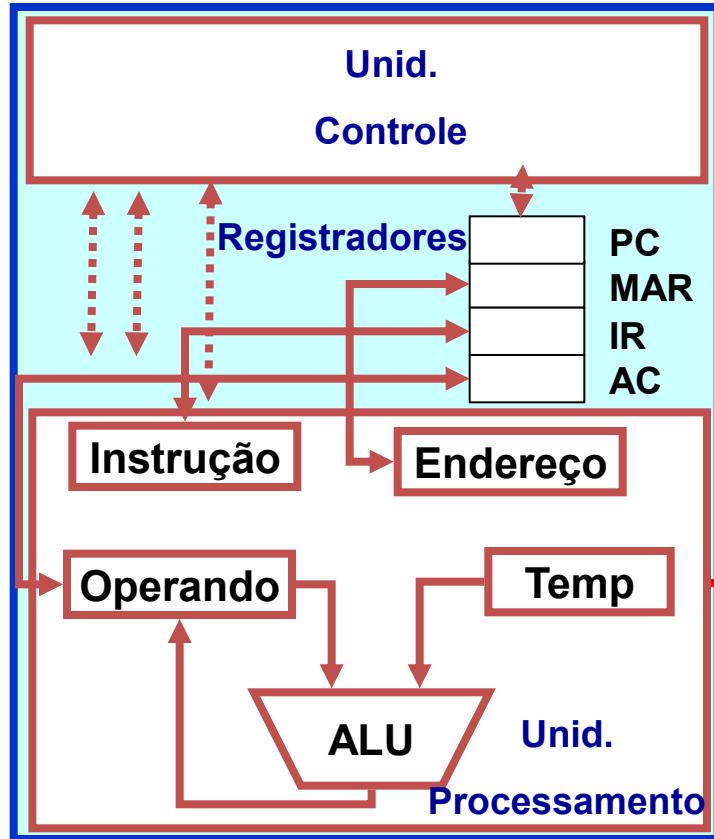


Etapa de execução de  
instrução

# Componentes de um Computador



# Mais Detalhes de uma CPU



**PC:** Program Counter

**MAR:** Memory Address Register

**IR:** Instruction Register

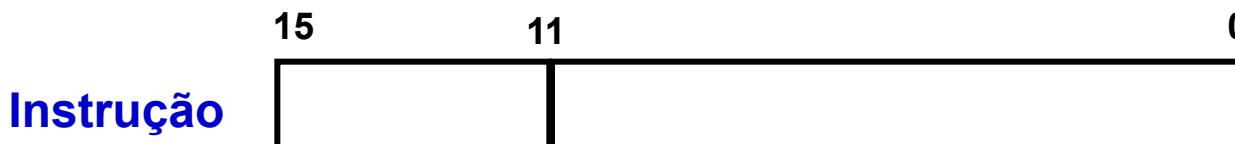
**AC:** Accumulator

# Executando um Programa em um Computador Hipotético

- Instruções e Dados ocupam 16 bits na memória
- Memória composta por **palavras** de 16 bits
- Formato de Dados e Instruções:



Sinal – 1 bit                      Magnitude – 15 bits



Opcode – 4 bits                      Endereço – 12 bits

- $2^4 = 16$  instruções possíveis nesta arquitetura

# Executando um Programa em um Computador Hipotético

- Por simplicidade, examinaremos 3 registradores
  - **PC** – Contém o endereço da instrução a ser executada
  - **AC** – Contém um operando
  - **IR** – Contém a instrução executada
- Repertório de Instruções

Opcode	Significado	Descrição
0001	AC $\leftarrow$ Mem	Carrega em AC conteúdo de memória
0010	Mem $\leftarrow$ AC	Salva na memória conteúdo de AC
0101	AC $\leftarrow$ AC + Mem	Soma a AC conteúdo de memória

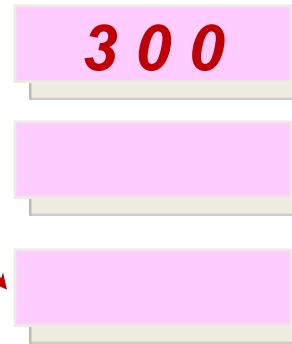
# Passo a Passo da Execução de um Programa

*Conteúdo de memória e registradores  
em hexadecimal*

*Memória*

300	1940
302	5942
304	2942
...	
940	0003
942	0002

*Registradores da CPU*



*PC (endereço)*

*AC (operando)*

*IR (Instrução)*

1940

*Opcode = 1 (0001)*

*Endereço = 940*

0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

# Passo a Passo da Execução de um Programa

*Memória*

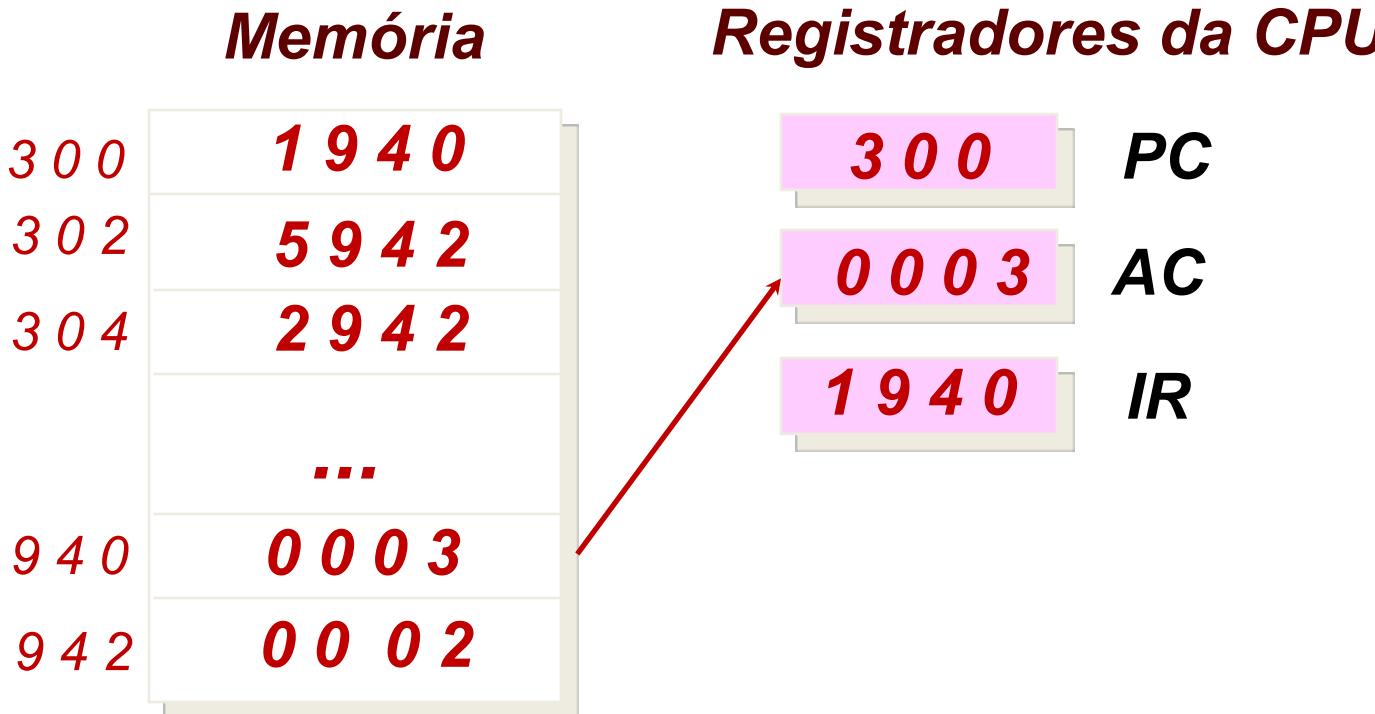
300	1940
302	5942
304	2942
	...
940	0003
942	0002

*Registradores da CPU*



0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

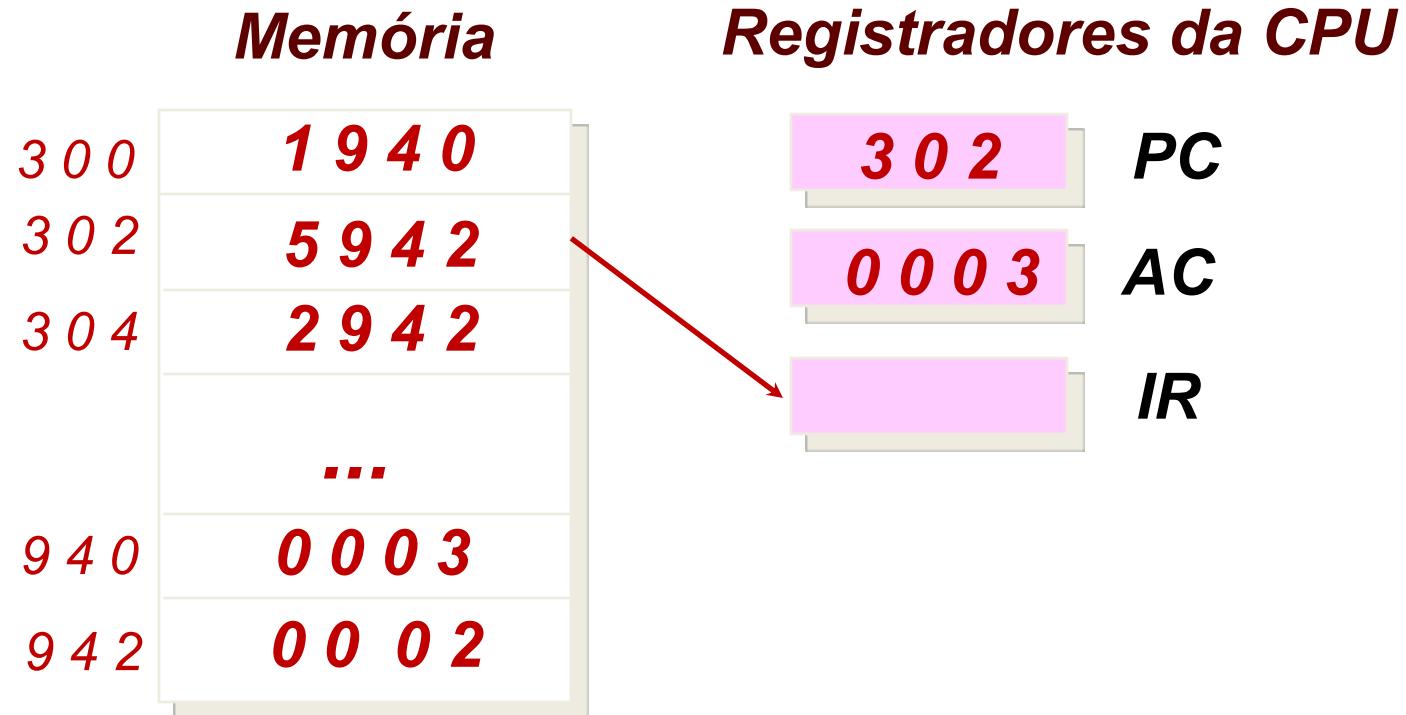
# Passo a Passo da Execução de um Programa



0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

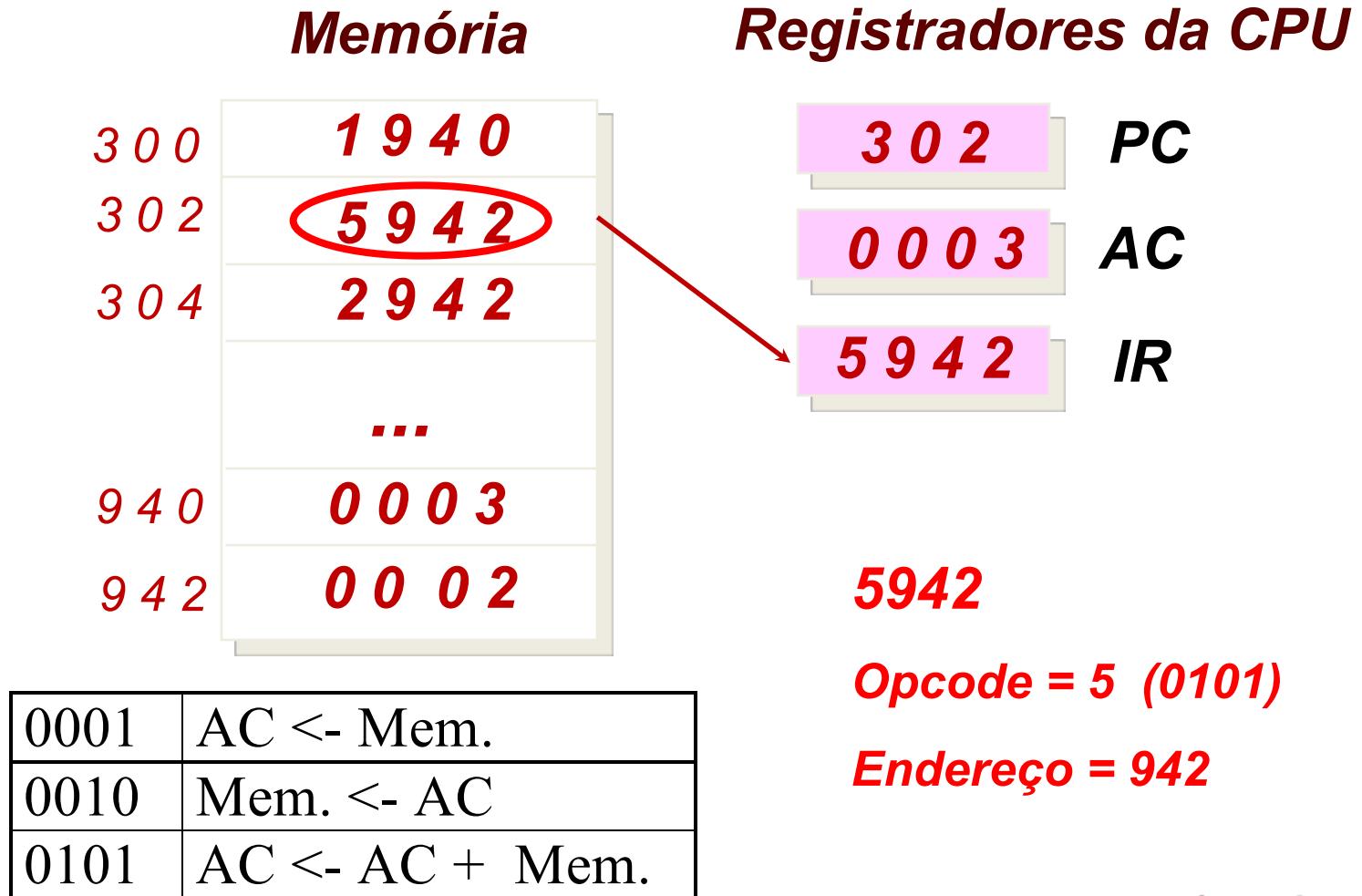


# Passo a Passo da Execução de um Programa

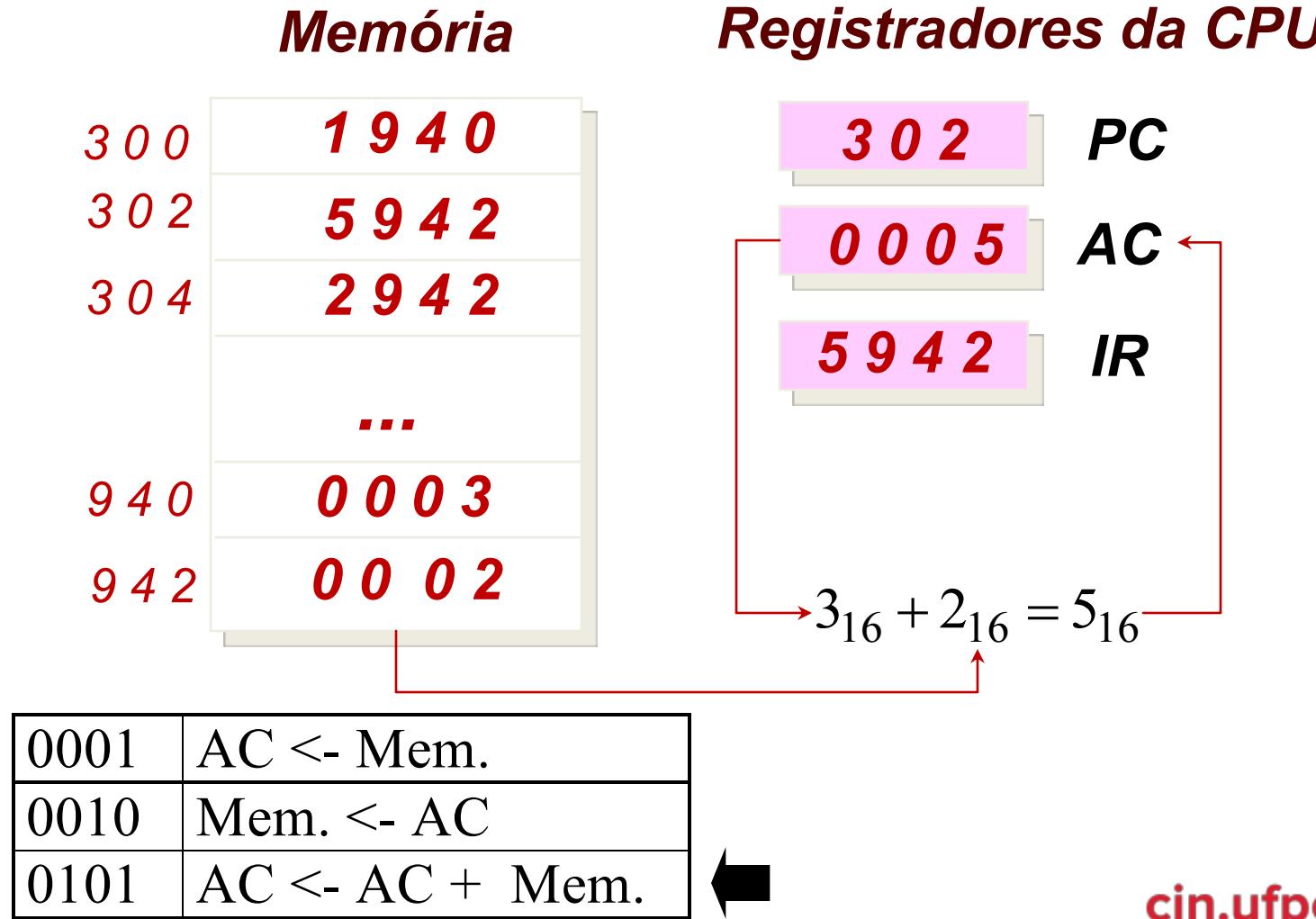


0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

# Passo a Passo da Execução de um Programa



# Passo a Passo da Execução de um Programa

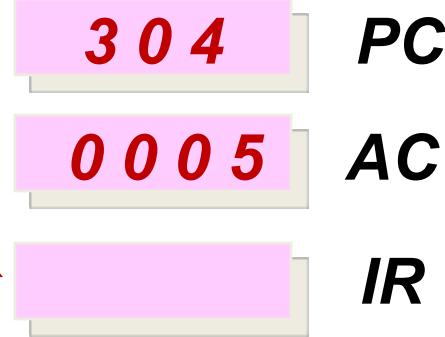


# Passo a Passo da Execução de um Programa

*Memória*

300	1940
302	5942
304	2942
	...
940	0003
942	0002

*Registradores da CPU*



0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

# Passo a Passo da Execução de um Programa

*Memória*

300	1940
302	5942
304	2942
	...
940	0003
942	0002

*Registradores da CPU*

304	PC
0005	AC
2942	IR

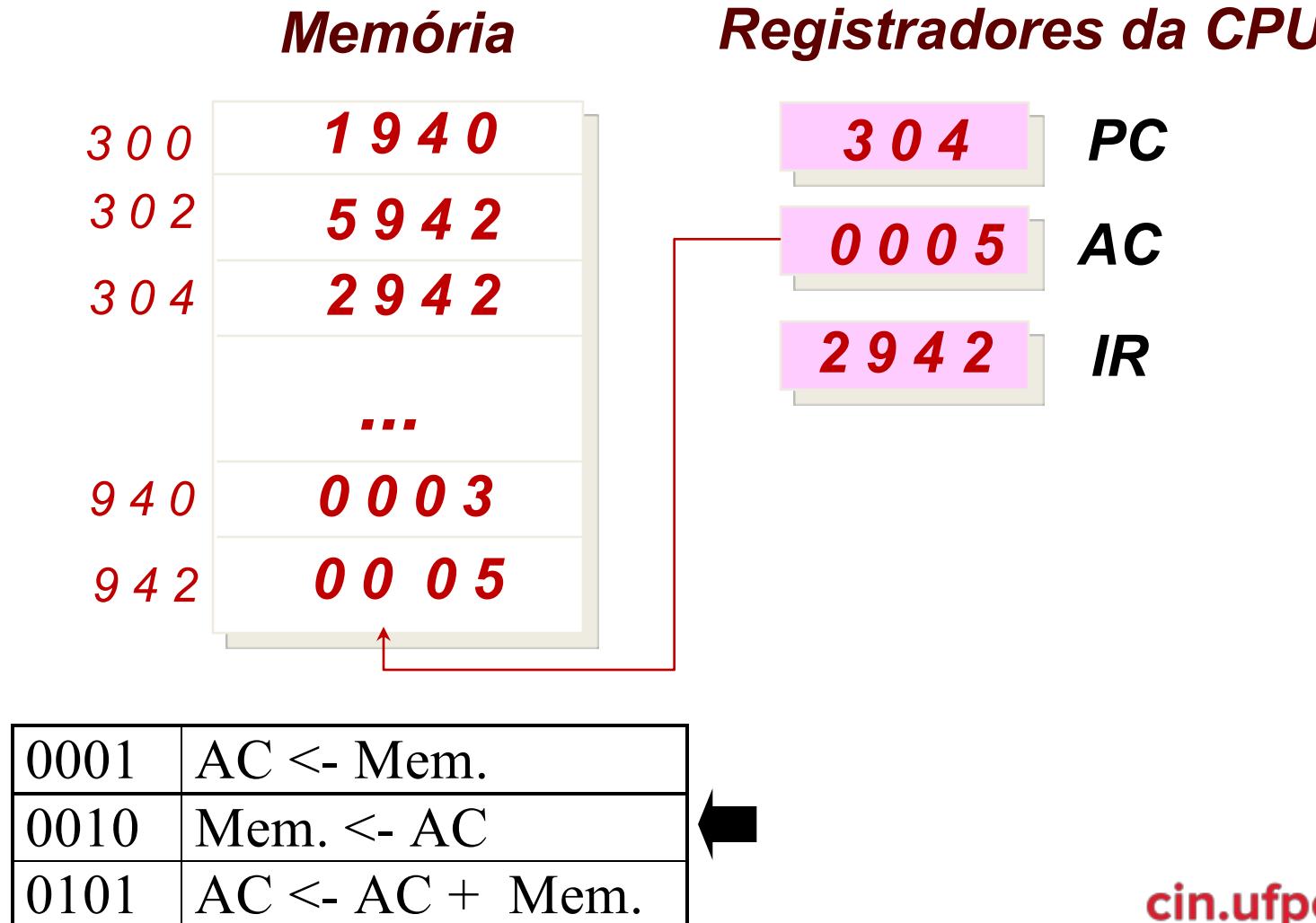
2942

Opcode = 2 (0010)

Endereço = 942

0001	AC <- Mem.
0010	Mem. <- AC
0101	AC <- AC + Mem.

# Passo a Passo da Execução de um Programa



# Arquitetura x Organização de Computadores

■ **Arquitetura** refere-se aos atributos do sistema visíveis ao programador

- Conjunto de registradores
- Tipos de Dados
- Modo de endereçamento
- Formato e Repertório de instruções

■ **Organização** refere-se às unidades operacionais que implementam a arquitetura

- Tecnologia de memória
- Interfaces
- Implementação das instruções
- Interconexões

# Obrigado!

# Infraestrutura de Hardware

## Desempenho

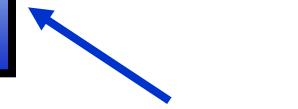
Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

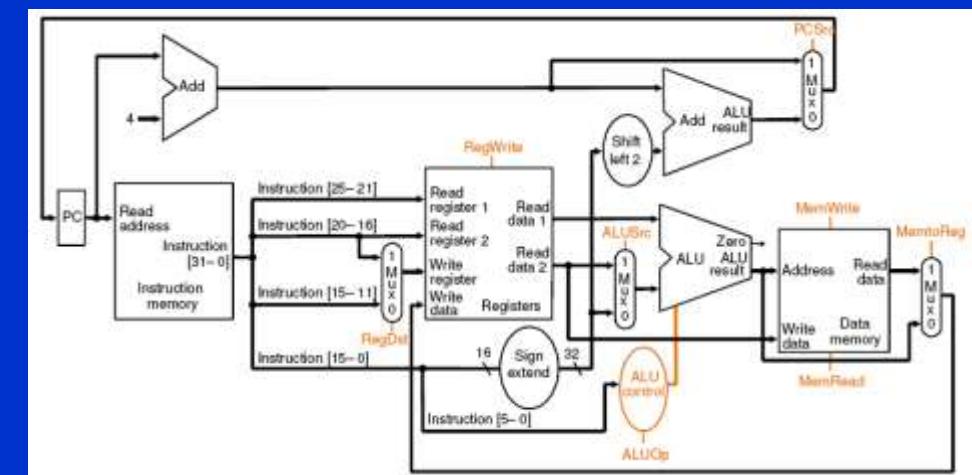
- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Interface HW/SW: ISA

Software



Repertório de  
Instruções da  
Arquitetura



Hardware

# Analisando Desempenho

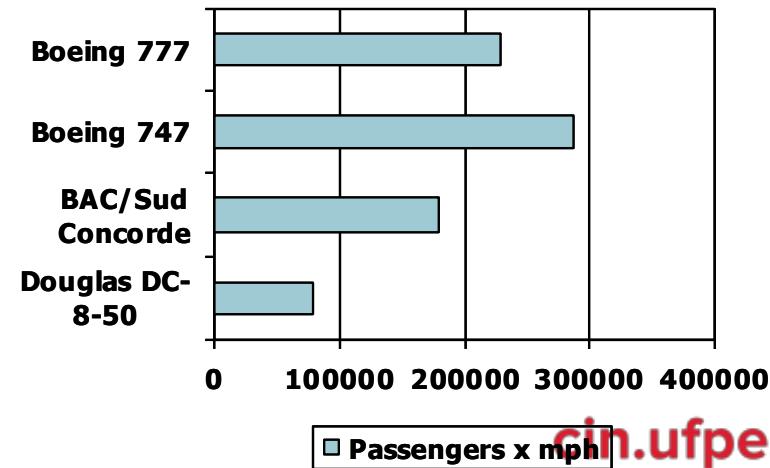
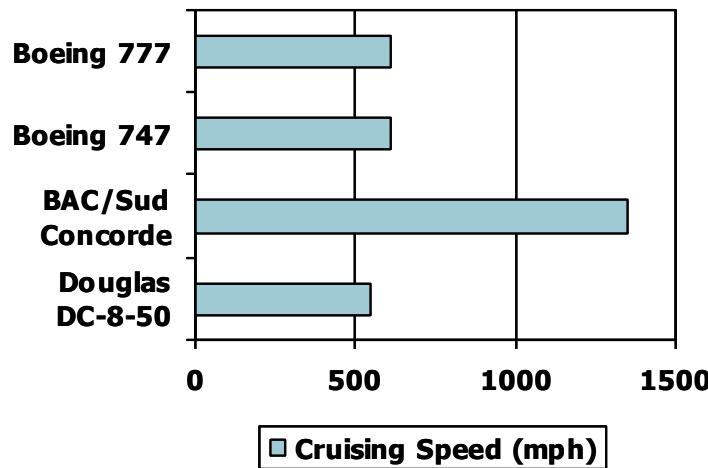
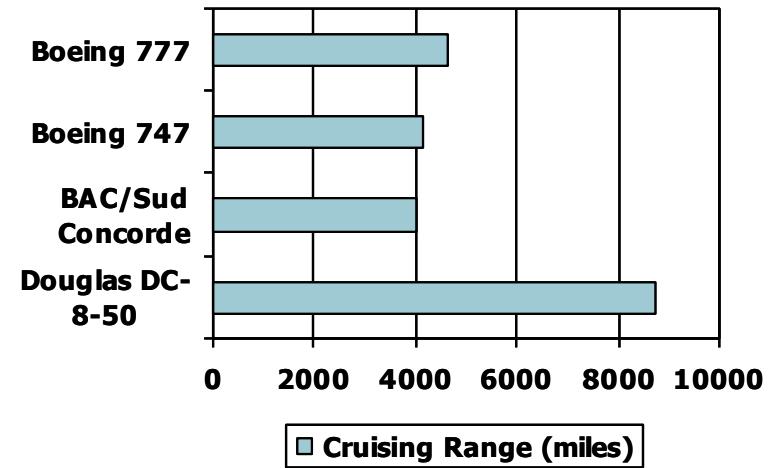
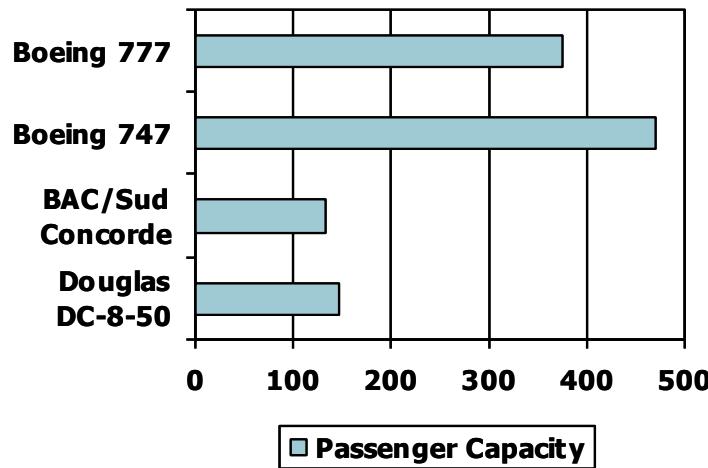
- Desempenho é um importante fator de qualidade de um sistema
- Análise difícil
  - Tamanho e complexidade de softwares atuais
  - Técnicas sofisticadas utilizadas no projeto do hardware para aumento de desempenho
- Diferentes métricas podem ser utilizadas dependendo do tipo de aplicação
- Diferentes aspectos do sistema podem ter impacto maior no desempenho
  - Entendimento de como os diferentes aspectos afetam o sistema é essencial para desenvolver software ou projetar hardware com melhor desempenho

# Fatores Que Influenciam Desempenho

- Algoritmo
  - Determina número de operações executadas
- Linguagem de Programação, compilador, arquitetura
  - Determina número de instruções de máquina executadas por operação
- Processador e estrutura de memória
  - Determina velocidade em que instruções são executadas
- E/S (incluindo Sistema Operacional)
  - Determina velocidade em que operações de E/S são executadas

# Definindo Desempenho

Qual avião tem melhor desempenho?



# Diferentes Métricas

- Tempo de Execução ou Resposta
  - Tempo que leva para completar uma tarefa
  - Importante para o usuário comum de um sistema
- Throughput
  - Trabalho total feito por unidade de tempo
    - Exemplos: tarefas ou transações por segundo ou hora
  - Importante para análise de máquinas servidoras
  - Frequentemente afetado pelo tempo de execução

**Analisaremos fatores que afetam tempo de execução**

# Desempenho Relativo

- Desempenho = 1/Tempo de Execução
- “X é  $n$  vezes mais rápido que Y”

$$\begin{aligned} \text{Desempenho}_x / \text{Desempenho}_y \\ = \text{Tempo de Execução}_y / \text{Tempo de Execução}_x = n \end{aligned}$$

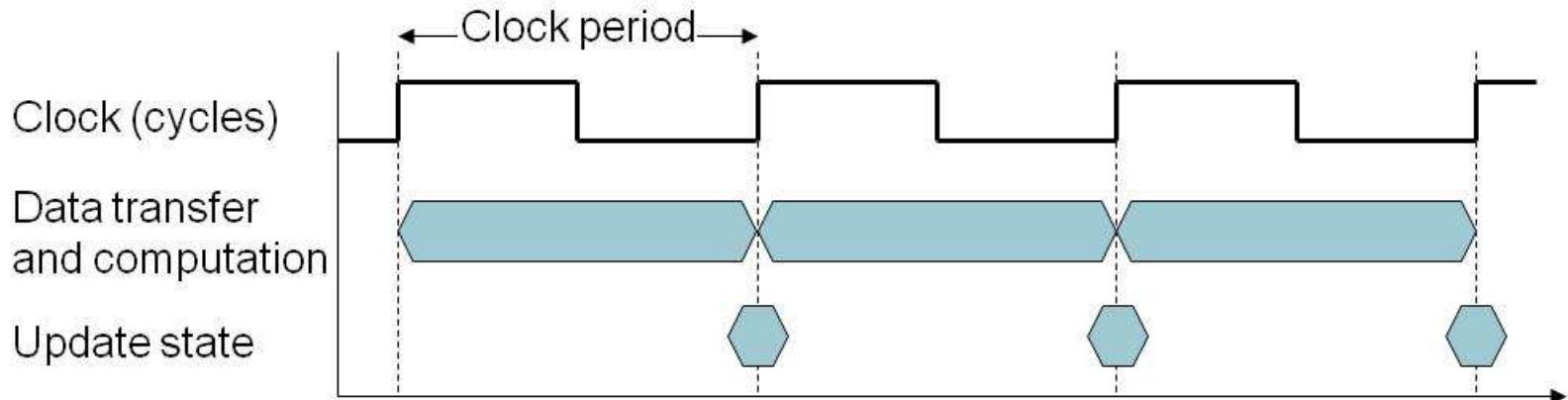
- **Exemplo: tempo para rodar um programa**
  - 10s em A, 15s em B
  - $\text{Tempo de Execução}_B / \text{Tempo de Execução}_A$   
 $= 15s / 10s = 1.5$
  - Então A é 1.5 vezes mais rápido que B

# Medindo Tempo de Execução

- Tempo Gasto (Elapsed time)
  - Tempo total de resposta, incluindo todos os aspectos
    - Processamento, E/S, overhead de sistema operacional, tempo esperando algum evento
  - Determina desempenho do sistema
  
- Tempo de CPU (CPU Time)
  - Tempo de processamento de um programa
    - Não leva em conta tempo de E/S, execução de outros programas, etc

# Clock

- Operações de uma CPU são governadas pelo clock



- Período do clock: duração de um ciclo de clock
  - Ex:  $250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$
- Frequência do Clock : ciclos por segundo
  - Ex:  $4.0\text{GHz} = 4000\text{MHz} = 4.0 \times 10^9\text{Hz}$

# Tempo de CPU

**Tempo de CPU = Ciclos de Clock (utilizados pela CPU) x Período de Clock**  
**= Ciclos de Clock Frequência**

- Desempenho pode ser melhorado
  - Reduzindo número de ciclos de clock
  - Aumentando frequência do clock
- Projetista de Hardware deve frequentemente fazer o *trade off* entre frequência e número de ciclos

# Tempo de CPU: Exemplo

- Computador A
  - Frequência de clock: 2GHz, Tempo de CPU: 10s
- Projetando Computador B
  - Objetivo: Tempo de CPU = 6s
  - Maior frequência do clock possível, mas isto causa um aumento no número de ciclos em 1,2x
- Qual deve ser então a frequência do clock para atingir o tempo de CPU necessário?

$$\text{Frequência}_B = \frac{\text{Ciclos de Clock}_B}{\text{Tempo de CPU}_B} = \frac{1,2 \times \text{Ciclos de Clock}_A}{6s}$$

$$\begin{aligned}\text{Ciclos de Clock}_A &= \text{Tempo de CPU}_A \times \text{Frequência}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Frequência}_B = \frac{1,2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

# Número de Instruções e CPI

**Ciclos de Clock = N° de Instruções x Ciclos por Instrução (CPI)**

**Tempo de CPU = N° de Instruções x CPI x Período de Clock**

$$= \frac{\text{Nº de Instruções} \times \text{CPI}}{\text{Frequência}}$$

- N° de Instruções de um programa
  - Determinado pelo algoritmo, ISA e compilador
- Quantidade média de ciclos por instrução
  - Determinado pelo hardware da CPU
  - Se diferentes instruções têm diferentes CPIs
    - Depende de como as diferentes instruções foram usadas no programa

# CPI : Exemplo

- Computador A: Tempo do ciclo = 250ps, CPI = 2,0
- Computador B: Tempo do ciclo = 500ps, CPI = 1,2
- Mesma ISA
- Qual é o **mais** rápido e **quão** mais rápido?

$$\begin{aligned}\text{Tempo de CPU}_A &= \text{Nº de Instruções} \times \text{CPI}_A \times \text{Tempo do Ciclo}_A \\ &= N \times 2,0 \times 250\text{ps} = N \times 500\text{ps}\end{aligned}$$

A é mais rápido

$$\begin{aligned}\text{Tempo de CPU}_B &= \text{Nº de Instruções} \times \text{CPI}_B \times \text{Tempo do Ciclo}_B \\ &= N \times 1,2 \times 500\text{ps} = N \times 600\text{ps}\end{aligned}$$

$$\frac{\text{Tempo de CPU}_B}{\text{Tempo de CPU}_A} = \frac{N \times 600\text{ps}}{N \times 500\text{ps}} = 1,2$$

1,2x mais rápido

# Mais Detalhes sobre CPI

- Se diferentes classes de instruções levam diferentes números de ciclos

$$\text{Ciclos de Clock} = \sum_{i=1}^n (\text{CPI}_i \times \text{Nº de instruções}_i)$$

- Média ponderada da CPI

$$\text{CPI} = \frac{\text{Ciclos de Clock}}{\text{Nº de Instruções}} = \sum_{i=1}^n \left( \text{CPI}_i \times \frac{\text{Nº de Instruções}_i}{\text{Nº de Instruções}} \right)$$

# Mais um Exemplo de CPI

- Uma mesma sequência de código foi compilado de 2 formas diferentes usando instruções das classes A, B, C

Classe	A	B	C
CPI da classe	1	2	3
Nº de Instruções na sequência 1	2	1	2
Nº de Instruções na sequência 2	4	1	1

- **Nº de Instruções na sequência 1 = 5**
  - Ciclos de Clock  
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$   
 $= 10$
  - Média CPI =  $10/5 = 2,0$
- **Nº de Instruções na sequência 2 = 6**
  - Ciclos de Clock  
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$   
 $= 9$
  - Média CPI =  $9/6 = 1,5$

# Desempenho: Resumindo

$$\text{Tempo de CPU} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de Clock}}{\text{Instrução}} \times \frac{\text{Segundos}}{\text{Ciclo de Clock}}$$

- Desempenho depende de:
  - Algoritmo: afeta nº de instruções, possivelmente CPI
  - Linguagem: afeta nº de instruções, CPI
  - Compilador: afeta nº de instruções, CPI
  - ISA: afeta nº de instruções, CPI, período do clock

# Infraestrutura de Hardware

Otimizando o Desempenho

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Otimizando Desempenho Através de Algoritmo: Arrays x Ponteiros

```
int clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i = i + 1)
        array[i] = 0;
}
```

```
addi $t0,$zero,0 # i = 0
L1:sll $t1,$t0,2    # $t1 = i * 4
    add $t2,$a0,$t1 # $t2= &array[i]
    sw $zero,0($t2) # array[i] = 0
    addi $t0,$t0,1    # i = i + 1
    slt $t3,$t0,$a1 # $t3=(i< size)
    bne $t3,$zero,L1
```

```
int clear2(int *array, int size) {
    int *p;
    for(p= &array[0];p< &array[size];
p = p + 1)
    *p = 0;
}
```

```
addi $t0,$a0,0 # p = & array[0]
    sll $t1,$a1,2    # $t1 = size * 4
    add $t2,$a0,$t1 # $t2=&array[size]
L2:sw $zero,0($t0) # *p = 0
    addi $t0,$t0,4    # p = p + 1 *4
    slt $t3,$t0,$t2 # $t3=
                                #(p<&array[size])
    bne $t3,$zero,L2
```

- Versão com array requer que shift seja dentro do loop
  - Para calcular endereço do elemento do índice i

# Arrays x Ponteiros

- Embora operações envolvendo arrays possam ser feitas com ponteiros, a escolha de uma forma de trabalhar ou outra pode ter impacto no desempenho
- Indexação de um array envolve:
  - Multiplicar índice pelo tamanho do tipo
  - Adicionar este valor ao endereço base do array para descobrir endereço do elemento indexado
- Ponteiros correspondem diretamente aos endereços de memória
  - Evita a complexidade extra da indexação

# Otimizando Desempenho Através do Compilador

## Dependencies

Language dependent;  
machine independent

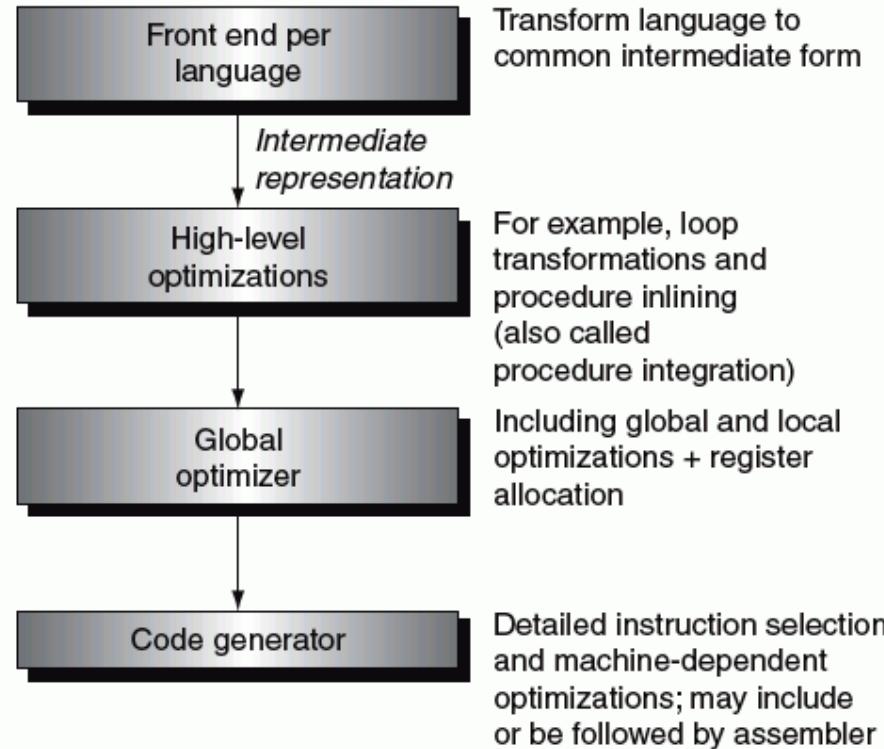
Somewhat language dependent;  
largely machine independent

Small language dependencies;  
machine dependencies slight  
(e.g., register counts/types)

Highly machine dependent;  
language independent

## Function

Transform language to  
common intermediate form



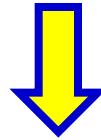
- **Diferentes tipos de otimização:**
  - Alto nível, local, global

# Exemplo de Otimização de Alto Nível

## ■ Loop Unrolling

- Consiste em replicar corpo do laço para reduzir número de iterações, evitando testes de fim de laço e jumps

```
for( i = 0; i < 10; i++) {  
    n = n + 5;  
}
```



```
for( i = 0; i < 5; i++) {  
    n = n + 5;  
    n = n + 5;  
}
```

# Outro Exemplo de Otimização de Alto Nível

## ■ Procedure Inlining

- Consiste em substituir chamadas a procedimento pelo corpo do procedimento, para reduzir overhead da chamada

```
void imprimeSituacao(float m) {
    if (m >= 7)
        printf("\nAprovado");
    else
        printf("\nReprovado");
}

int main() {
    float media;
    printf("Digite media: ");
    scanf("%f", &media);
    imprimeSituacao(media);
}
```



```
int main() {
    float media;
    printf("Digite media: ");
    scanf("%f", &media);
    if (media >= 7)
        printf("\nAprovado");
    else
        printf("\nReprovado");
}
```

# Otimização Local/Global

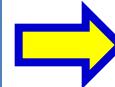
- Otimizações locais trabalham em cima de blocos simples individualmente
  - Ex: branches, expressões, loops, etc
- Otimizações globais trabalham sobre múltiplos blocos simples espalhados pelo código
- Exemplos de técnicas:
  - Strength Reduction
    - Substitui operações complexas por operações mais simples
    - Ex: Substitui instruções mult em assembly por sll quando for possível
  - Dead code elimination
    - Elimina código que não afeta resultado final do programa
    - Exs: variáveis não utilizadas, condições inalcansáveis de ifs

# Exemplo de Otimização Local/Global

## ■ Code Motion

- Consiste em identificar trechos de código dentro de um loop que calculam sempre o mesmo valor em todas as iterações (invariante do loop) e colocar estes trechos para fora do loop

```
int y;
int n = 0, x = 5;
for( i = 0; i < 10; i++) {
    y = x + 10;
    n = n + 5;
}
```

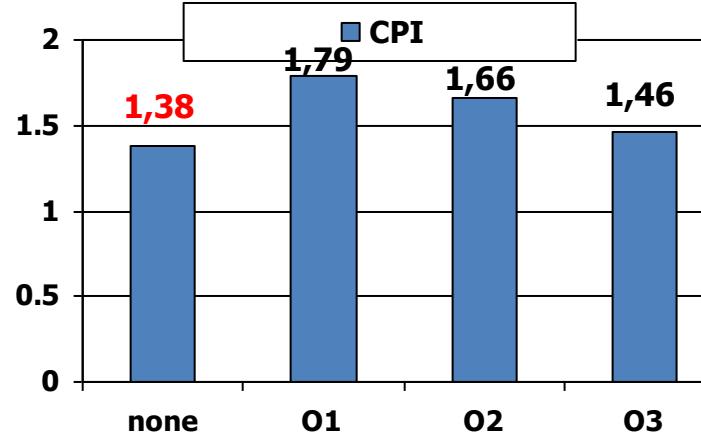
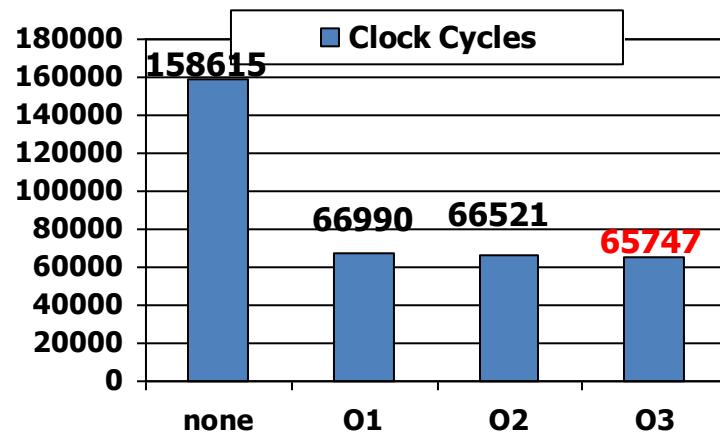
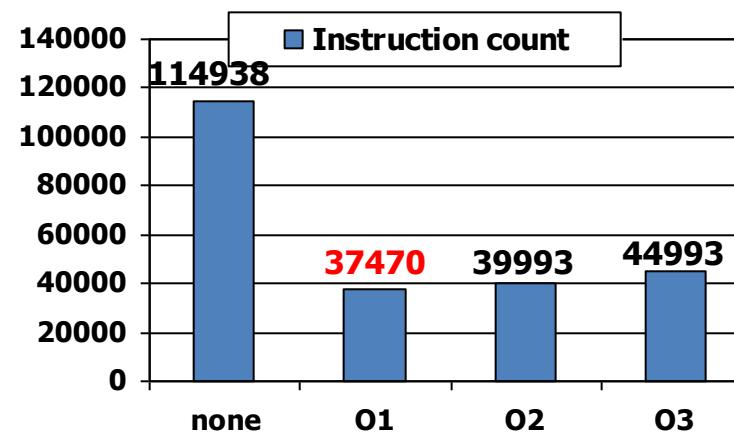
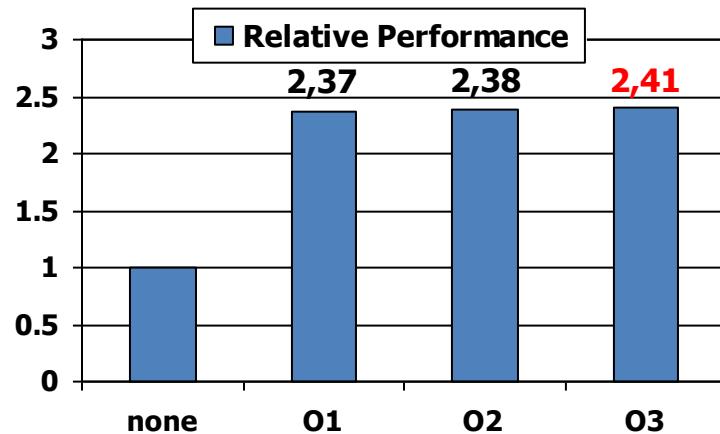


```
int y;
int n = 0, x = 5;
y = x + 10;
for( i = 0; i < 10; i++) {
    n = n + 5;
}
```

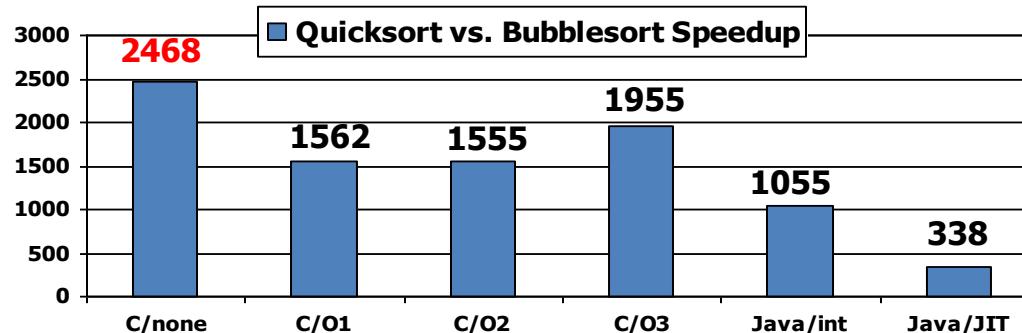
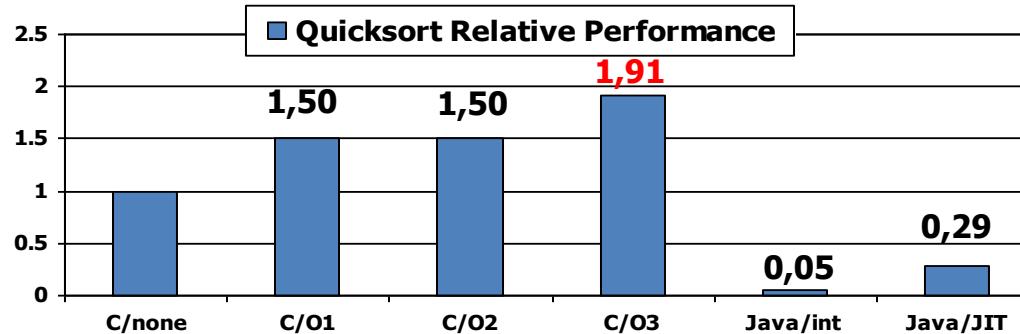
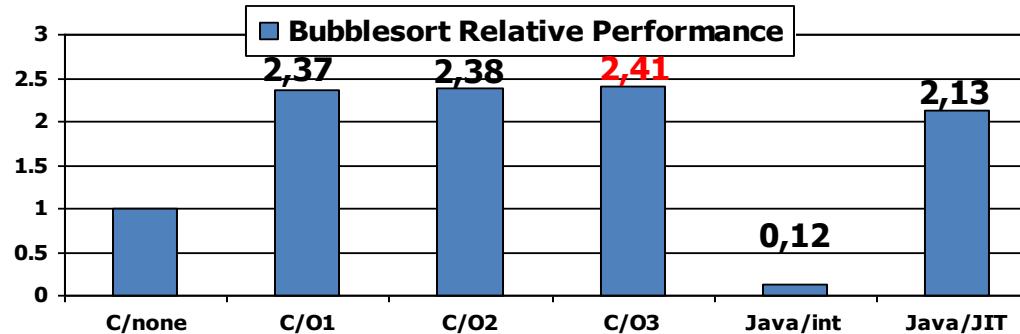
# Examinando Como Diferentes Fatores Podem Influenciar Desempenho

- Exemplo: Ordenação
  - Algoritmos : Bubble sort e Quicksort
  - Linguagem: C e Java
  - Compilador: gcc com 3 níveis de otimização para C (O1, O2 e O3) e interpretador e JIT para Java
  - ISA: Pentium 4 (com Linux)
- Comparação será feita em relação a códigos C não otimizados

# Efeito de Otimização do Compilador *gcc*



# Efeito da Linguagem e Algoritmo



# Analizando Resultados

- Número de instruções e CPI isoladamente não são bons indicadores de desempenho
- Otimizações de compilador são sensíveis ao algoritmo
- Otimizações nem sempre levam a um melhor desempenho
  - Pode aumentar tamanho do código, o que pode afetar desempenho, pois código pode ficar grande demais para caber em cache (**veremos depois**)
- Código compilado Java/JIT é significativamente mais rápido que código interpretado pelo JVM
  - Comparável a C otimizado em alguns casos
- Nenhuma otimização é capaz de compensar um algoritmo ruim!

# Como Podem Ser Comparados Diferentes Computadores?

- Benchmarks
  - Conjunto de aplicações padrões que são utilizados para avaliar o desempenho
  - Vários domínios:
    - Científicos
    - Banco de dados
    - Processamento de sinais
    - Rede
- SPEC (System Performance Evaluation Cooperative)
  - Iniciativa criada pela indústria para criar benchmarks para avaliar seus computadores

# Benchmarks da SPEC

- Benchmarks usando inteiros
  - Escritos em C, C++ e Fortran

SPECspeed®2017 Integer	Language [1]	KLOC [2]	Application Area
<a href="#">600.perlbench_s</a>	C	362	Perl interpreter
<a href="#">602.gcc_s</a>	C	1,304	GNU C compiler
<a href="#">605.mcf_s</a>	C	3	Route planning
<a href="#">620.omnetpp_s</a>	C++	134	Discrete Event simulation - computer network
<a href="#">623.xalancbmk_s</a>	C++	520	XML to HTML conversion via XSLT
<a href="#">625.x264_s</a>	C	96	Video compression
<a href="#">631.deepsjeng_s</a>	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
<a href="#">641.leela_s</a>	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
<a href="#">648.exchange2_s</a>	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
<a href="#">657.xz_s</a>	C	33	General data compression

# Benchmarks da SPEC

- Benchmarks usando ponto flutuante  
Escritos em C, C++ e Fortran

SPECspeed®2017 Floating Point	Language [1]	KLOC [2]	Application Area
<a href="#">603.bwaves_s</a>	Fortran	1	Explosion modeling
<a href="#">607.cactusBSSN_s</a>	C++, C, Fortran	257	Physics: relativity
	C++	8	Molecular dynamics
	C++	427	Biomedical imaging: optical tomography with finite elements
	C++, C	170	Ray tracing
<a href="#">619.lbm_s</a>	C	1	Fluid dynamics
<a href="#">621.wrf_s</a>	Fortran, C	991	Weather forecasting
	C++, C	1,577	3D rendering and animation
<a href="#">627.cam4_s</a>	Fortran, C	407	Atmosphere modeling
<a href="#">628.pop2_s</a>	Fortran, C	338	Wide-scale ocean modeling (climate level)
<a href="#">638.imagick_s</a>	C	259	Image manipulation
<a href="#">644.nab_s</a>	C	24	Molecular dynamics
<a href="#">649.fotonik3d_s</a>	Fortran	14	Computational Electromagnetics
<a href="#">654.roms_s</a>	Fortran	210	Regional ocean modeling

# Infraestrutura de Hardware

Implementação Pipeline de um  
Processador Simples

Prof. Adriano Sarmento

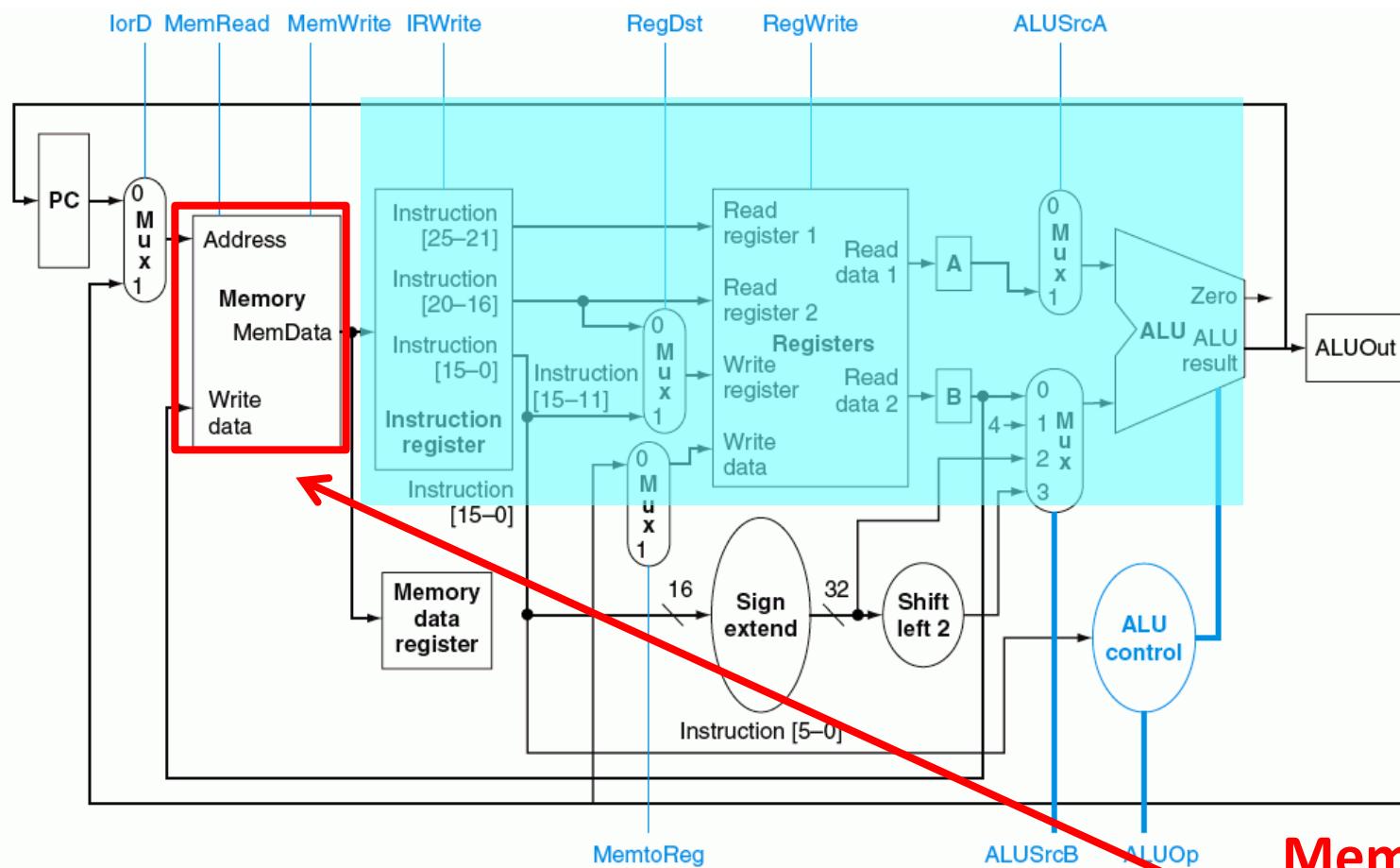
# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Diferentes Métricas de Desempenho

- Tempo de Execução (Latência)
  - Tempo que leva para completar uma tarefa
- Throughput
  - Trabalho total feito por unidade de tempo
    - Exemplos: transações ou instruções por segundo ou hora
- Uma métrica pode afetar a outra

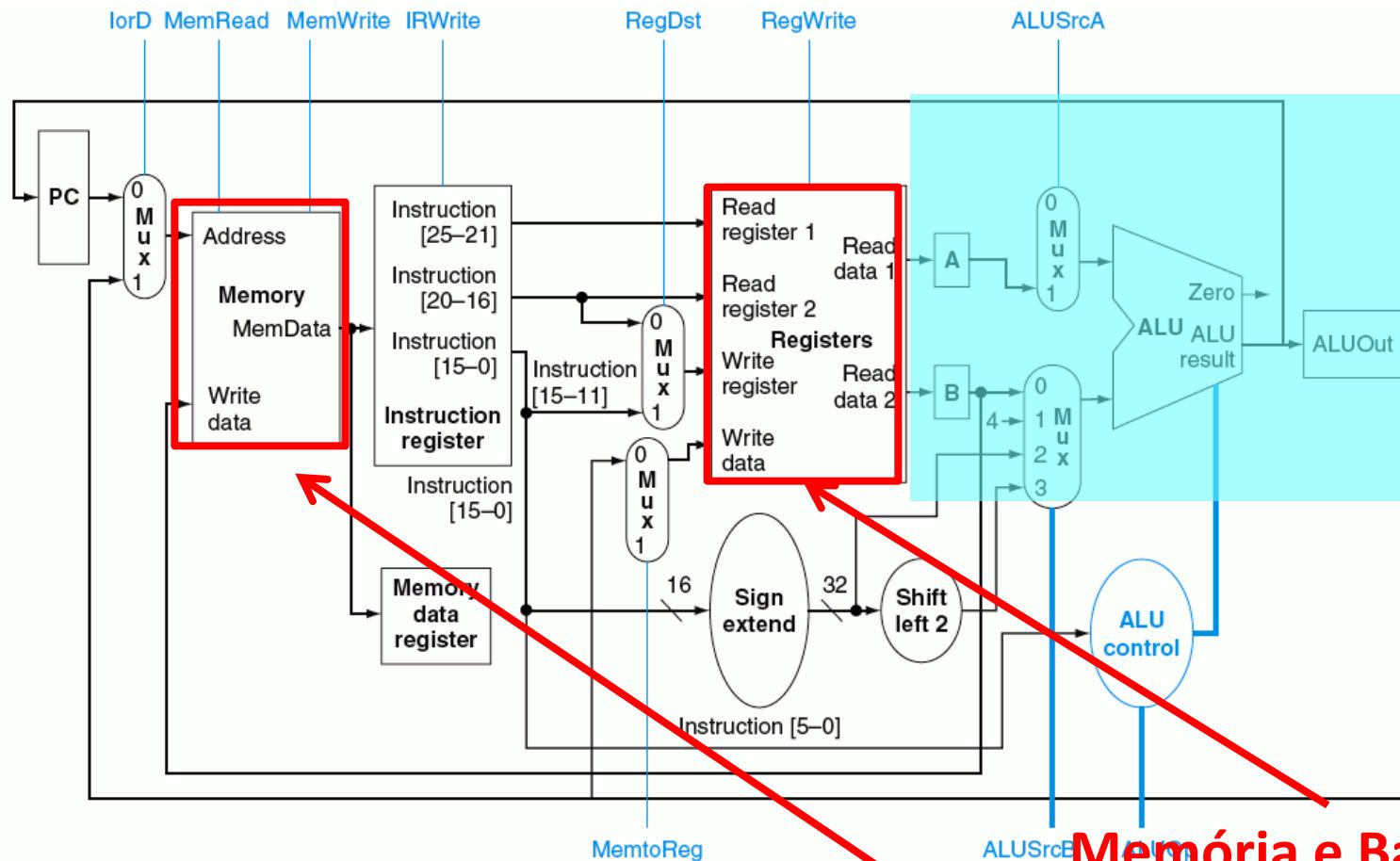
# Execução Serial de Uma Instrução ADD (Implementação Multiciclo)



**Etapa de decodificação e leitura de registradores**

**Memória não  
utilizada**

# Execução Serial de Uma Instrução ADD (Implementação Multiciclo)



Etapa de operação da ALU

Memória e Banco de  
Registradores não  
utilizados

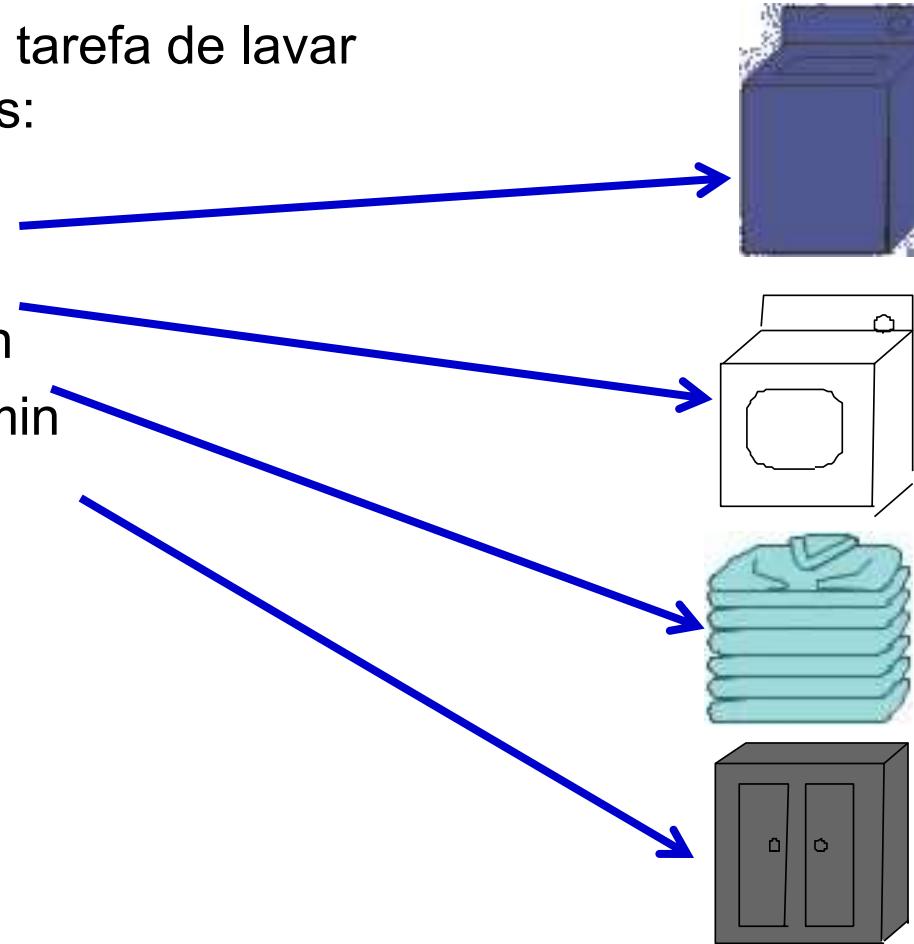
# Melhorando Uso de HW com Pipeline

- Em uma implementação serial de instruções, partes do hardware ficam ociosas durante a execução de uma instrução
- **Pipeline** é uma técnica de implementação que visa maximizar a utilização dos diferentes recursos de HW
- Pipeline permite que várias instruções sejam processadas simultaneamente com cada parte do HW atuando numa instrução distinta

# Analogia de um Pipeline: Lavanderia

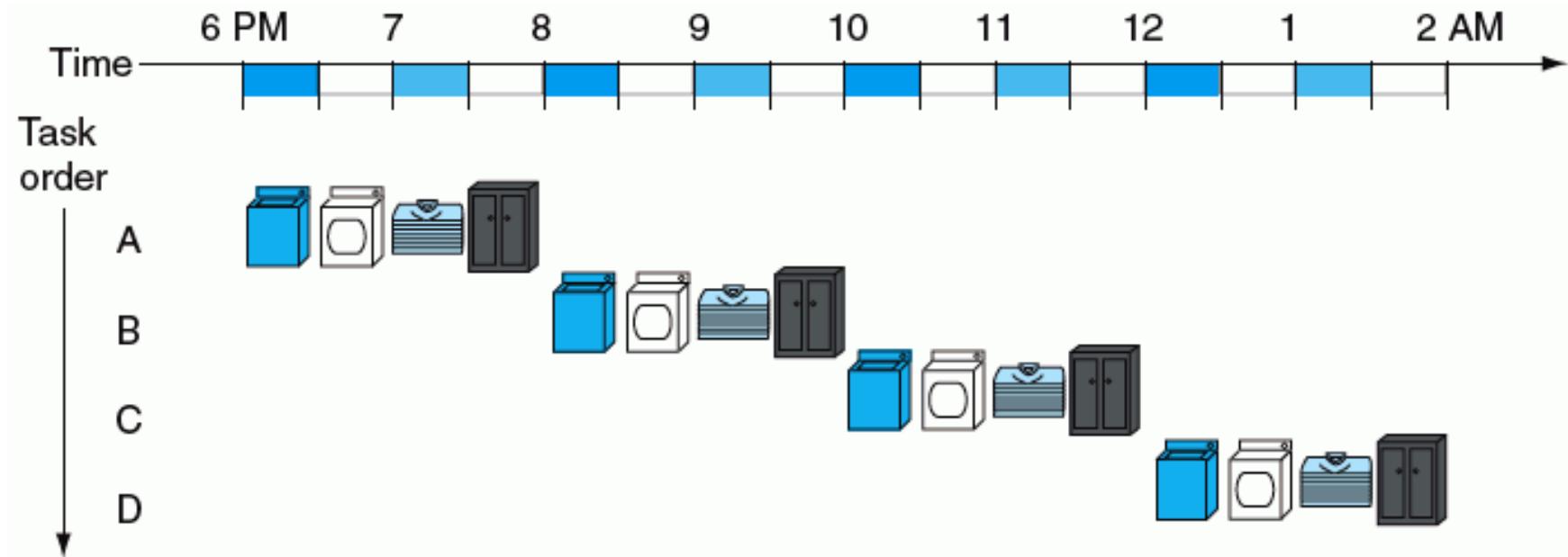
- Podemos dividir a tarefa de lavar roupa em 4 etapas:

- Lavar: 30 min
- Secar: 30 min
- Dobrar: 30 min
- Guardar : 30 min



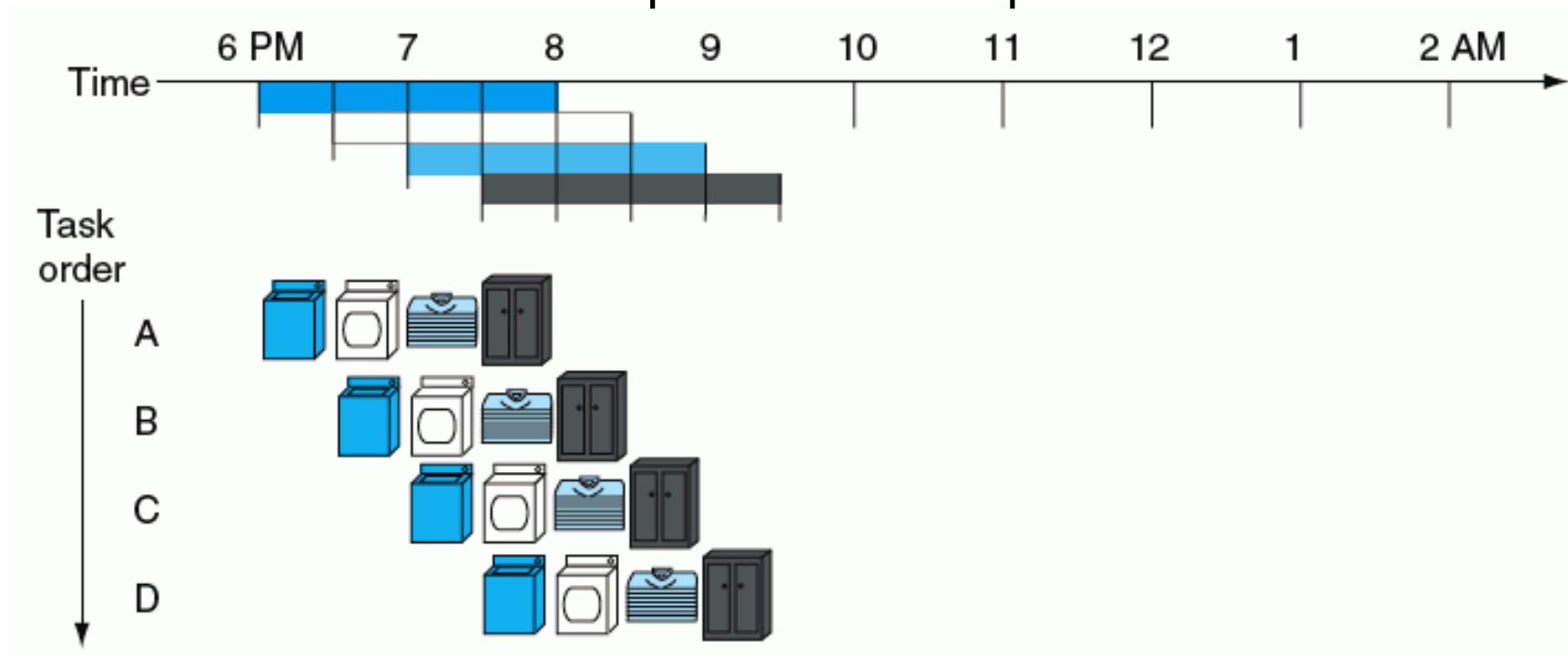
# Lavanderia: Execução Sequencial de Tarefas

- Para 4 lavagens de roupa:
  - 8 horas

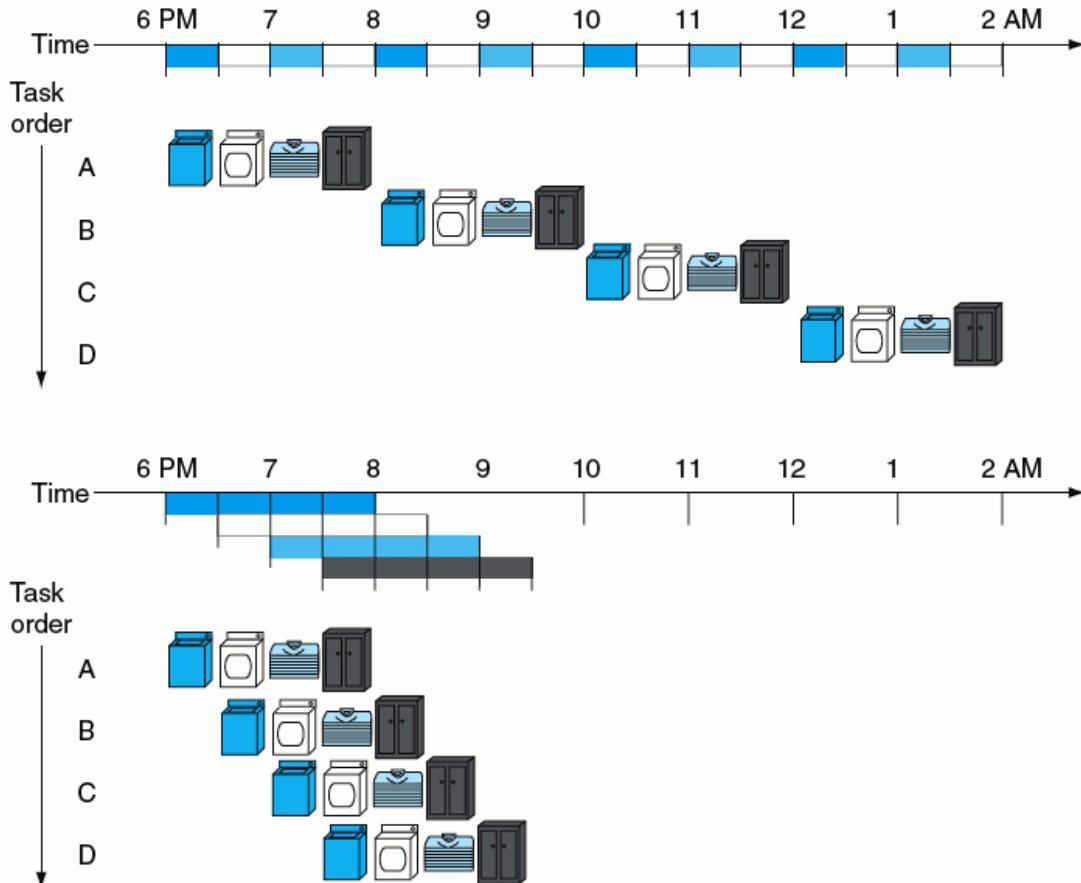


# Lavanderia: Execução Pipeline de Tarefas

- Para 4 lavagens de roupa:
  - 3,5 horas
- Diferentes lavagens podem ser executadas simultaneamente, desde que não utilizem o mesmo recurso em um mesmo período de tempo



# Lavanderia: Sequencial x Pipeline



- Para 4 lavagens de roupa:
  - Ganho de desempenho em  $8/3,5 = 2,3x$
- Tempo de execução de uma tarefa é o mesmo
- Pipeline melhora o throughput
  - Tempo de execução de um conjunto de tarefas

# Mais sobre Pipeline...

- Melhora no throughput → melhora no desempenho
- Aumento do número de estágios do pipeline → Aumento de desempenho
  - Mais execuções simultâneas
- Throughput é limitado pelo estágio mais lento do pipeline
  - Estágios devem ter a mesma duração
- Podem ocorrer dependências entre diferentes instâncias de execução, gerando espera
  - Reduz o desempenho

# Estágios de Processamento de Uma Instrução

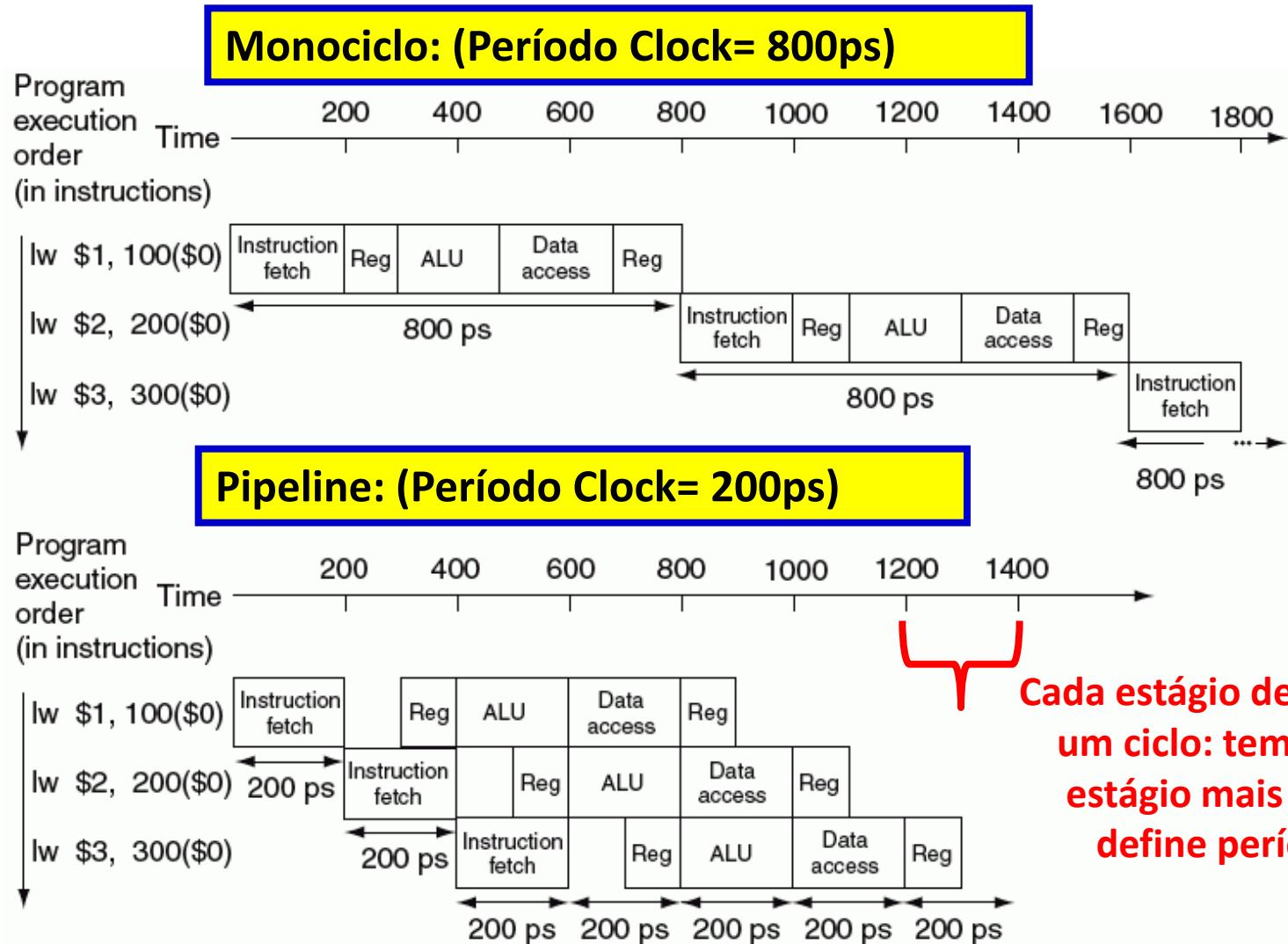


# Tempo de Execução de Cada Estágio

- Assuma que o tempo para os diferentes estágios são:
  - 100ps para leitura/escrita de registrador
  - 200ps para os outros estágios

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
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-format (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

# Desempenho : Monociclo x Pipeline

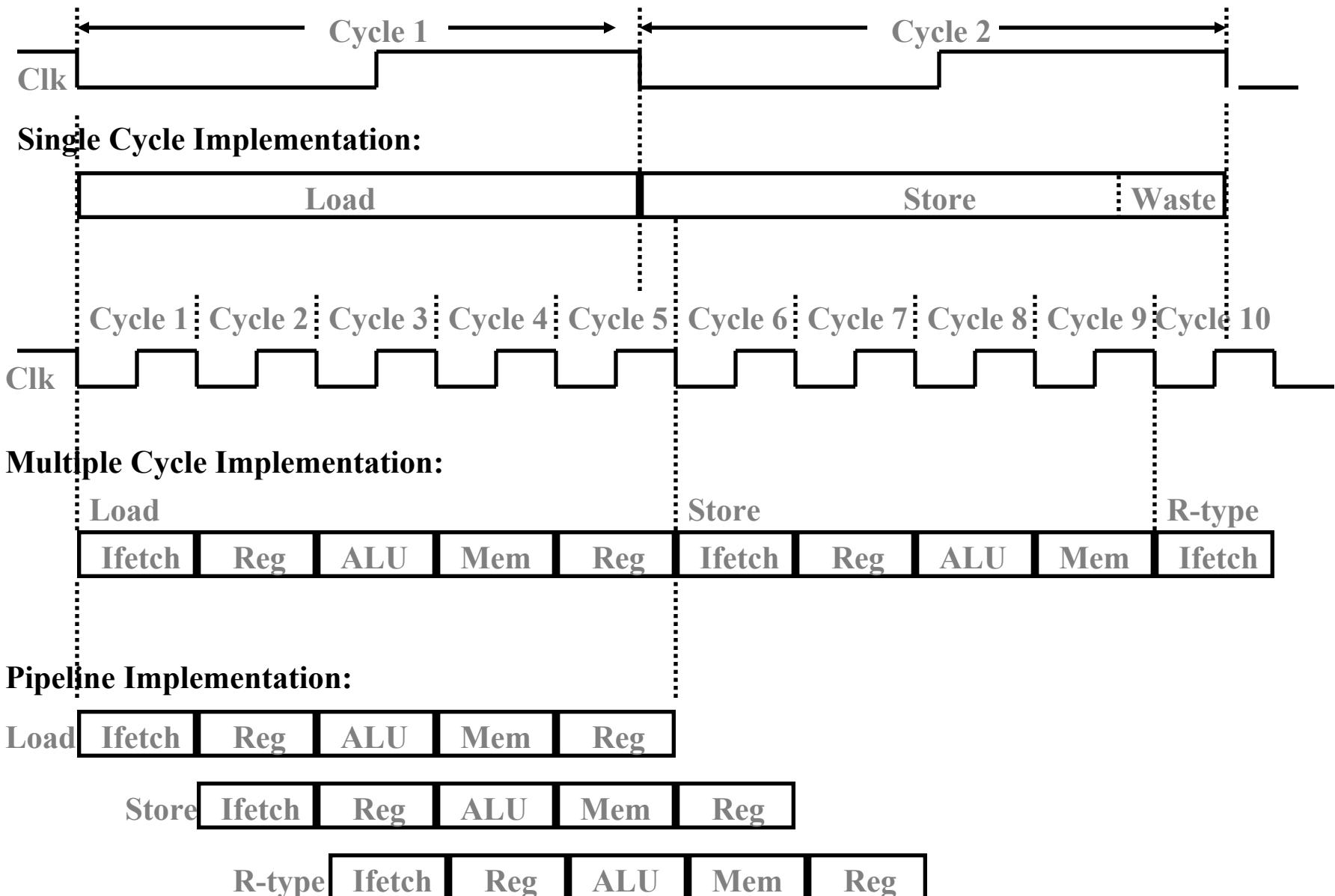


Cada estágio deve levar  
um ciclo: tempo do  
estágio mais lento  
define período

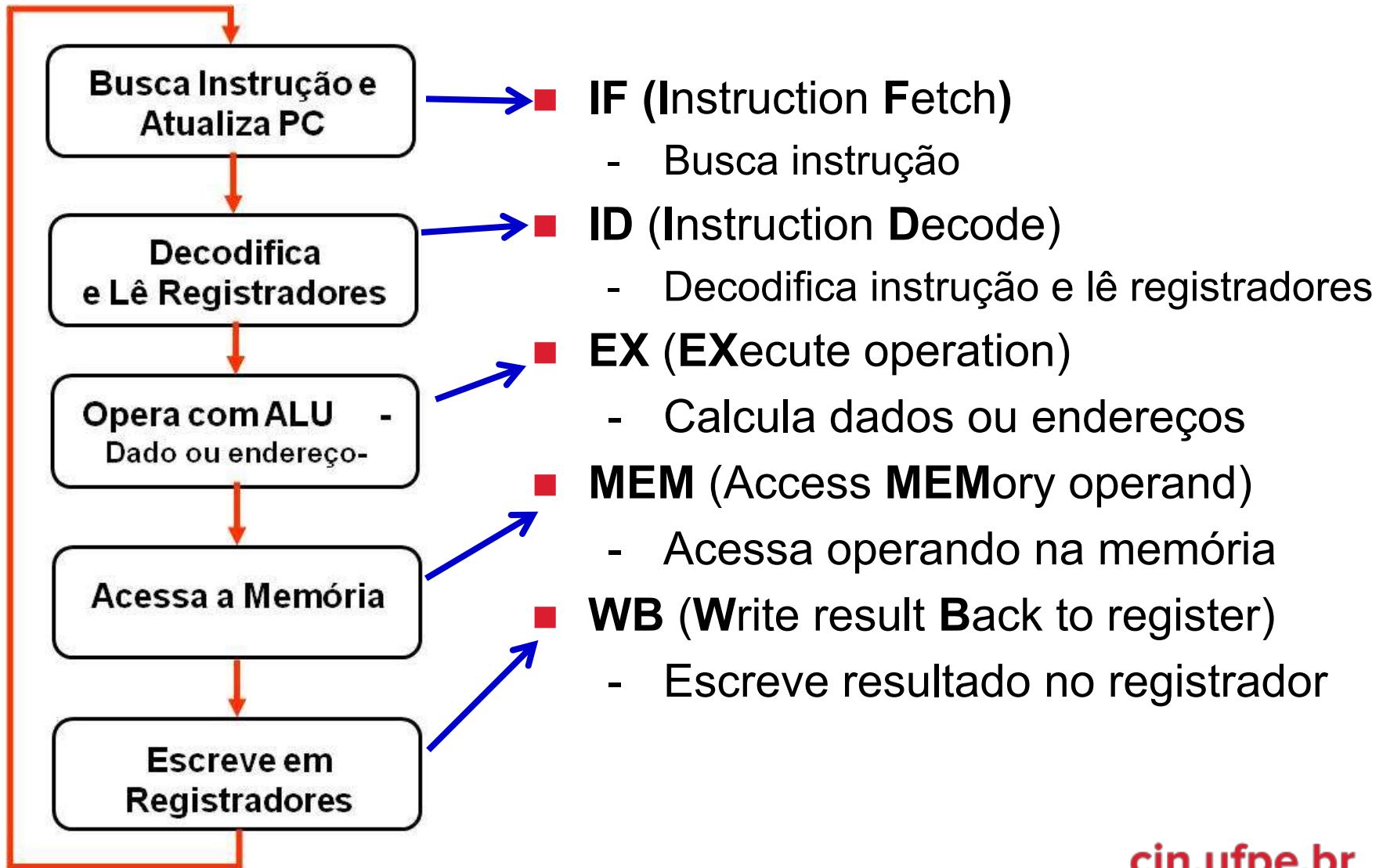
# Melhora de Desempenho do Pipeline

- Se todos os estágios balanceados, isto é, levam o mesmo tempo, e condições ideais
  - Tempo entre instruções<sub>pipelined</sub>  
$$= \frac{\text{Tempo entre instruções}_{\text{nonpipelined}}}{\text{Número de estágios}}$$
- Se não balanceado, aumento do desempenho é menor
- Latência (tempo de execução de cada instrução) não diminui com pipeline

# Mono, multi-ciclo e Pipeline



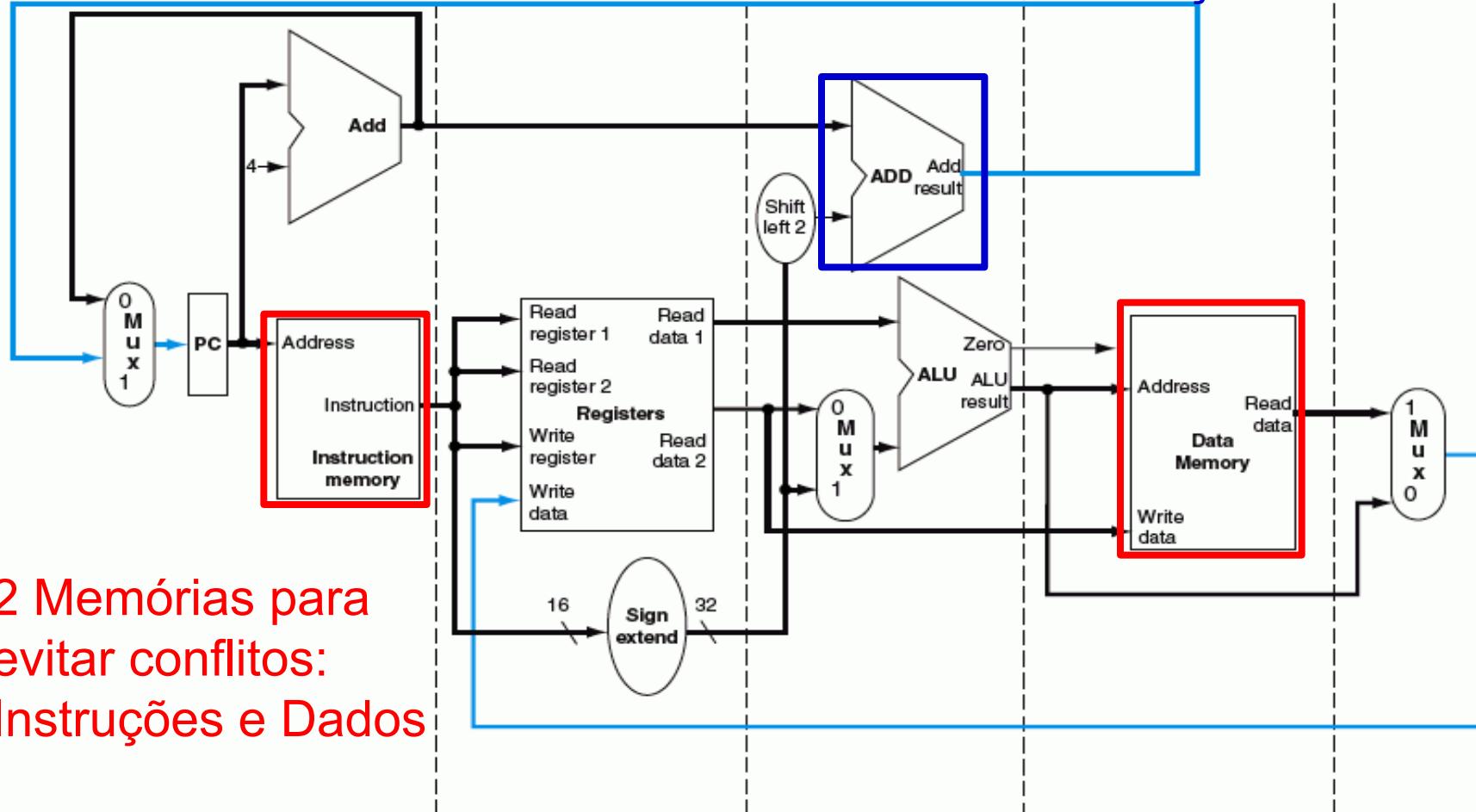
# Estágios de uma Implementação Pipeline de um Processador



# Implementação Pipeline do Datapath

IF: Instruction fetch    ID: Instruction decode/  
 register file read    EX: Execute/  
 address calculation    MEM: Memory access    WB: Write back

Para calcular  
endereço do branch



# Sentido dos Dados e Instruções em um Pipeline

IF: Instruction fetch

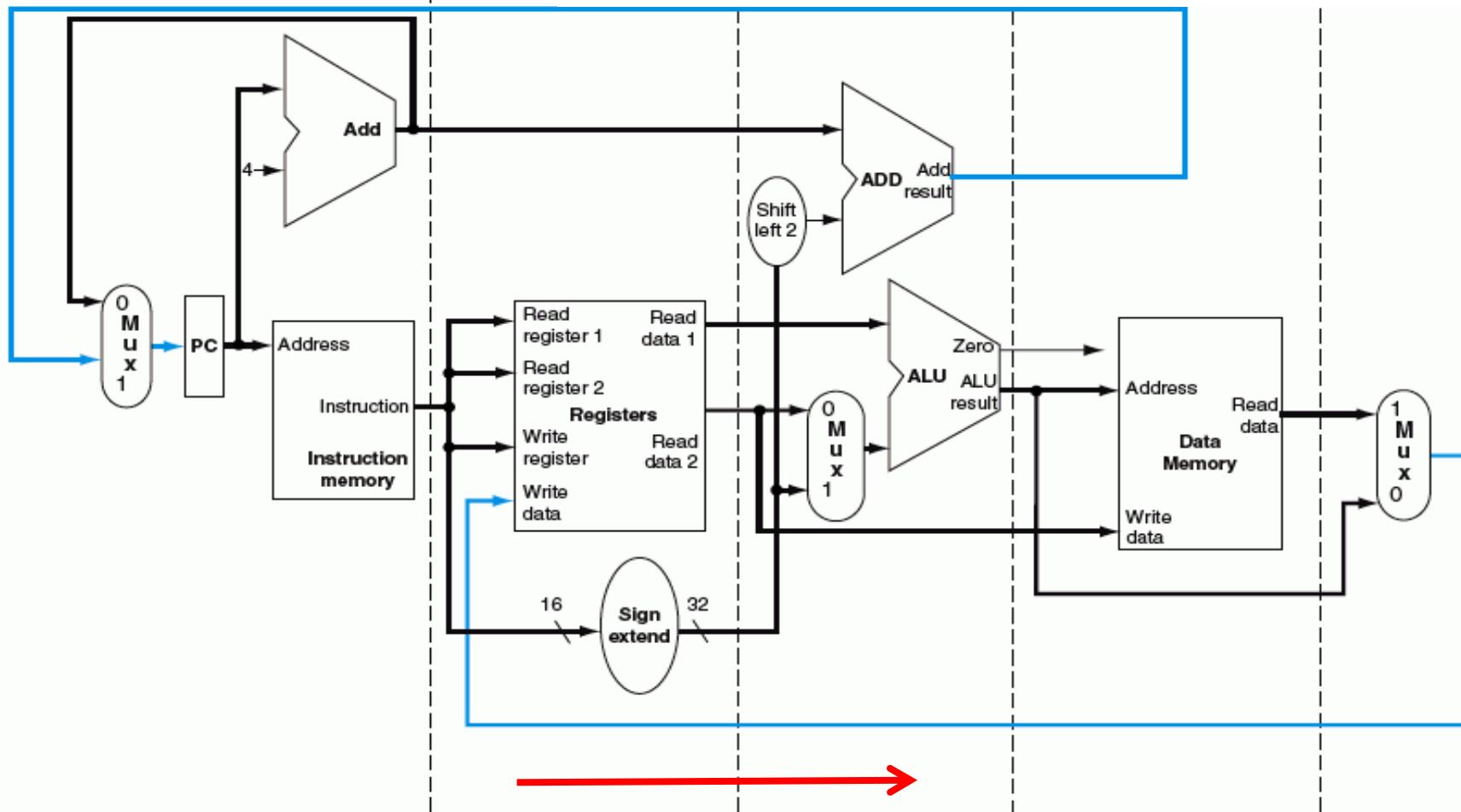
ID: Instruction decode/  
register file read

EX: Execute/  
address calculation

MEM: Memory access

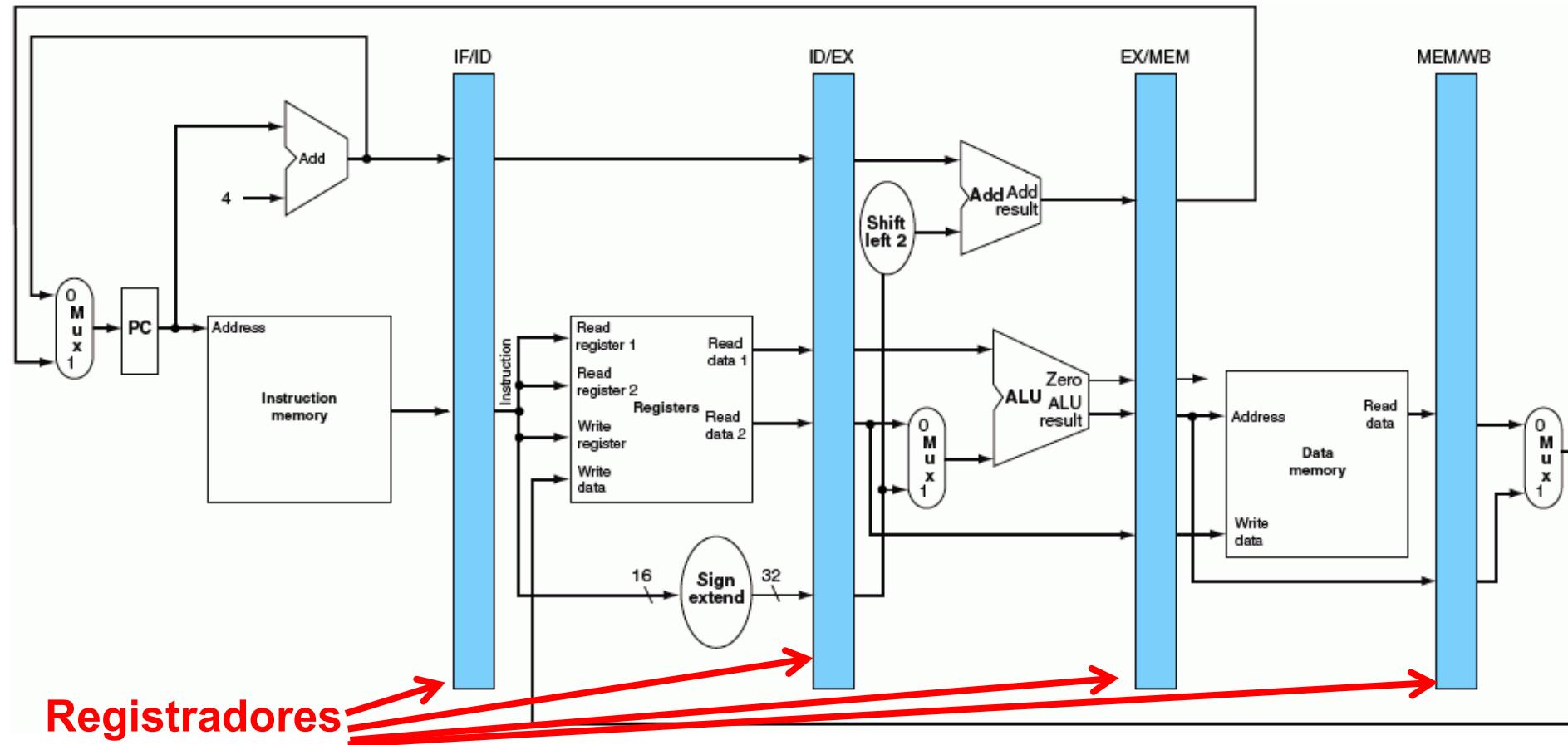
WB: Write back

Dados e instruções se movem da esquerda para a direita, exceto as linhas azuis marcadas



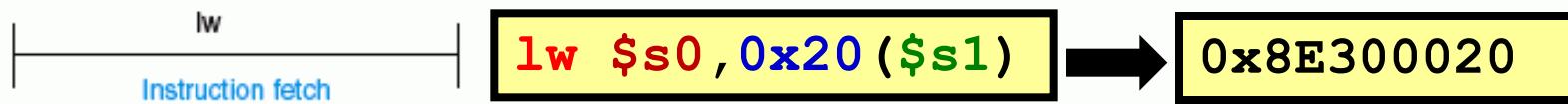
# Registradores do Pipeline

- São necessários registradores entre estágios
  - Para armazenar informação produzida no ciclo anterior

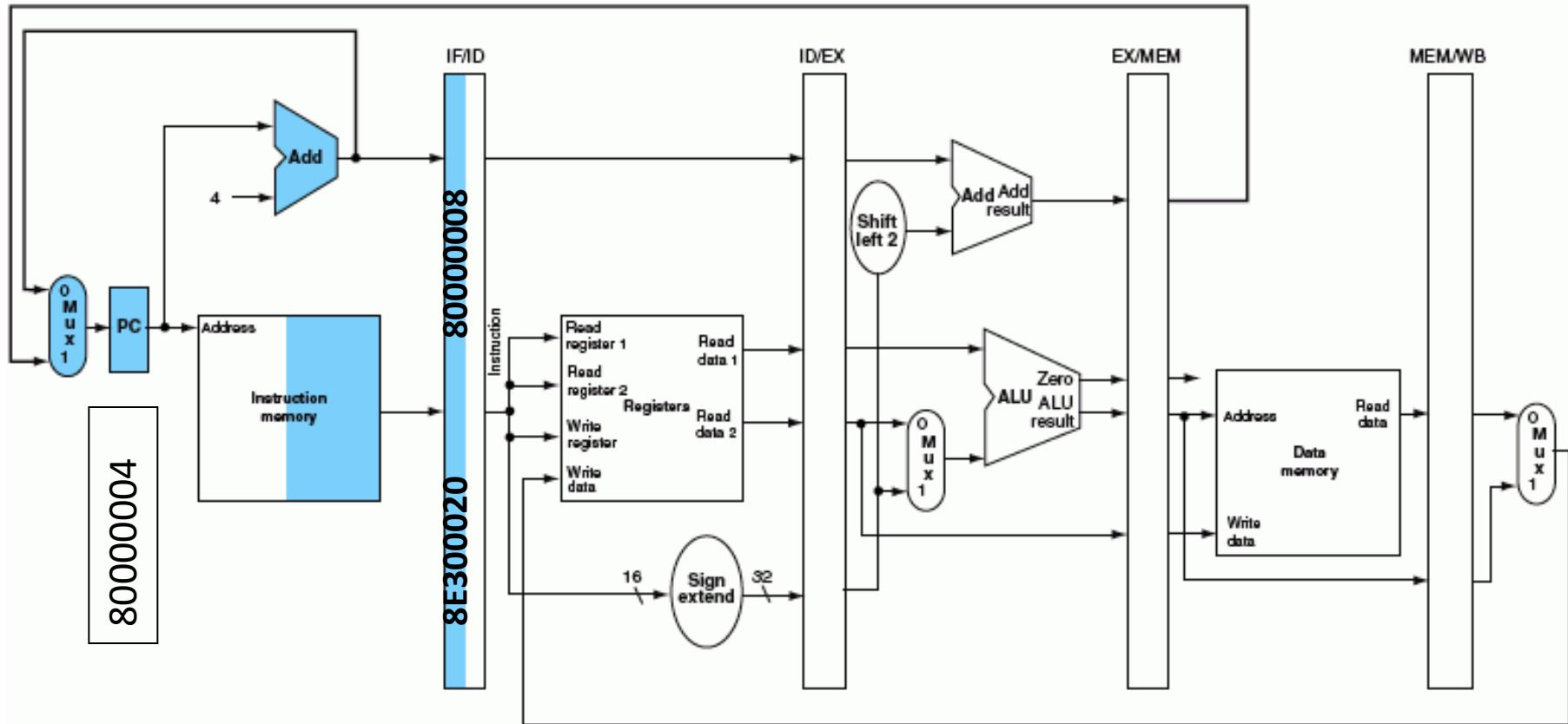


Registradores

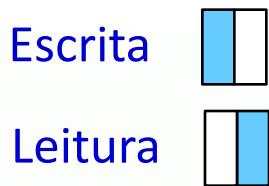
# Operação de um Pipeline: lw (Busca Instrução)



Escrita

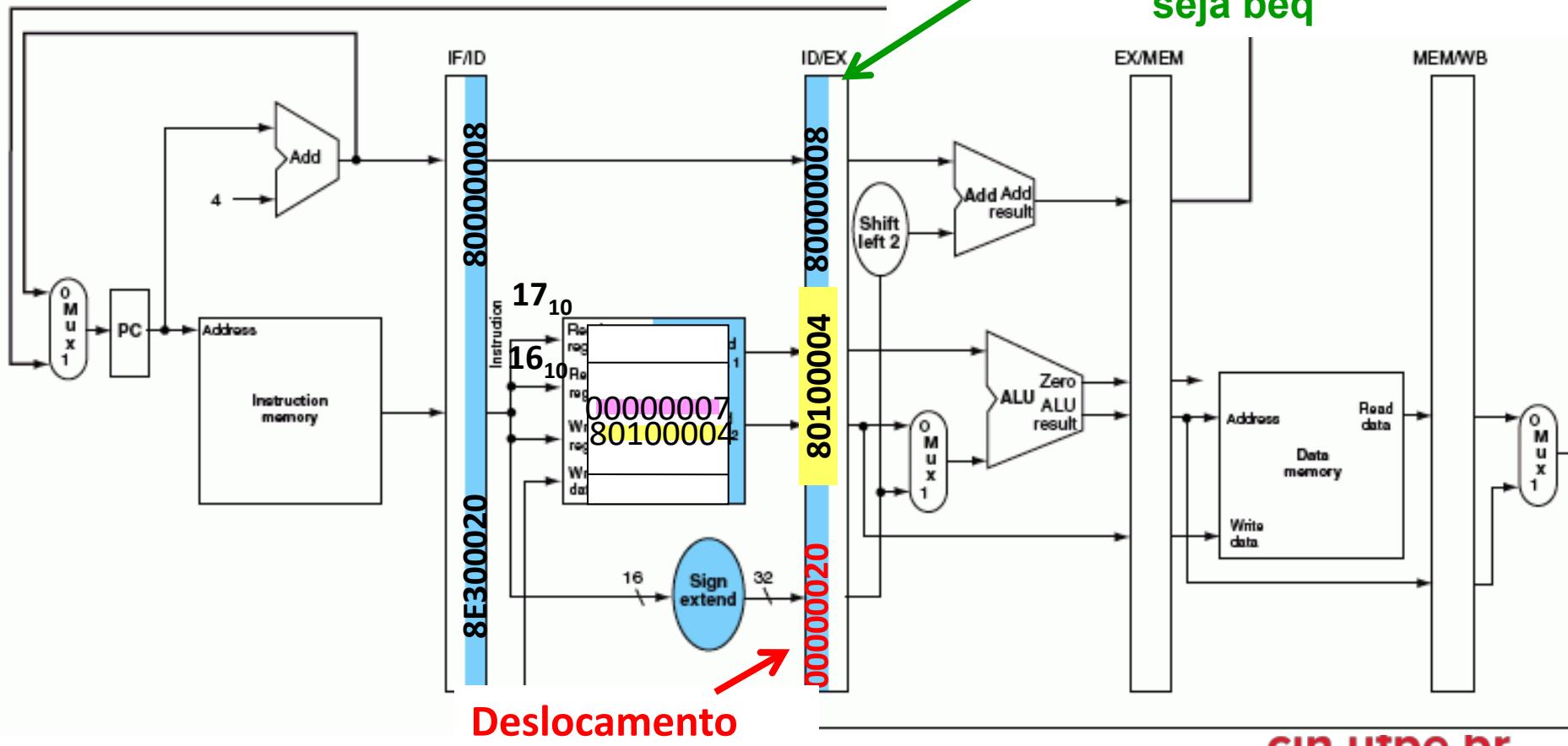


# Operação de um Pipeline: Iw (Decodificação)

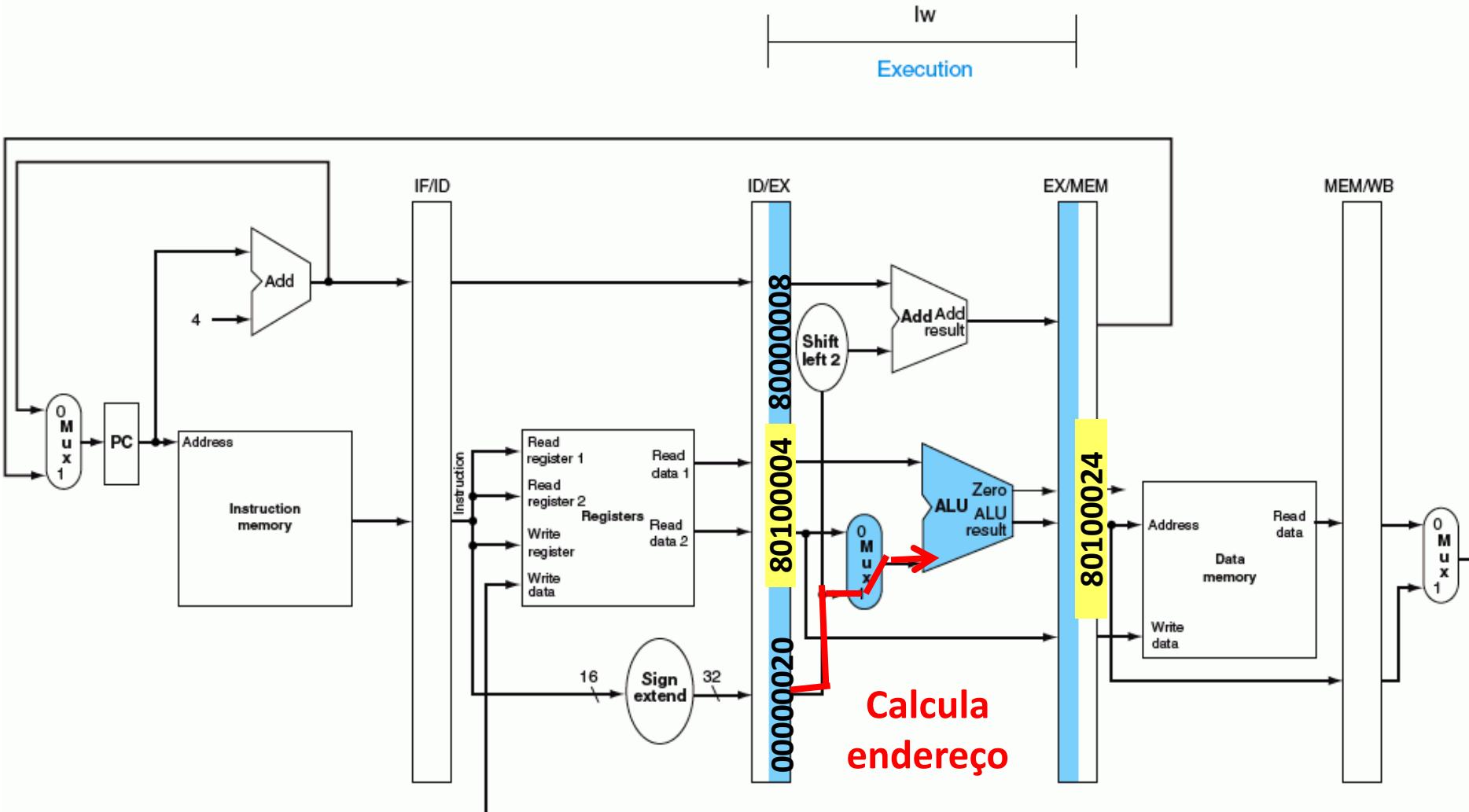


Iw  
Instruction decode

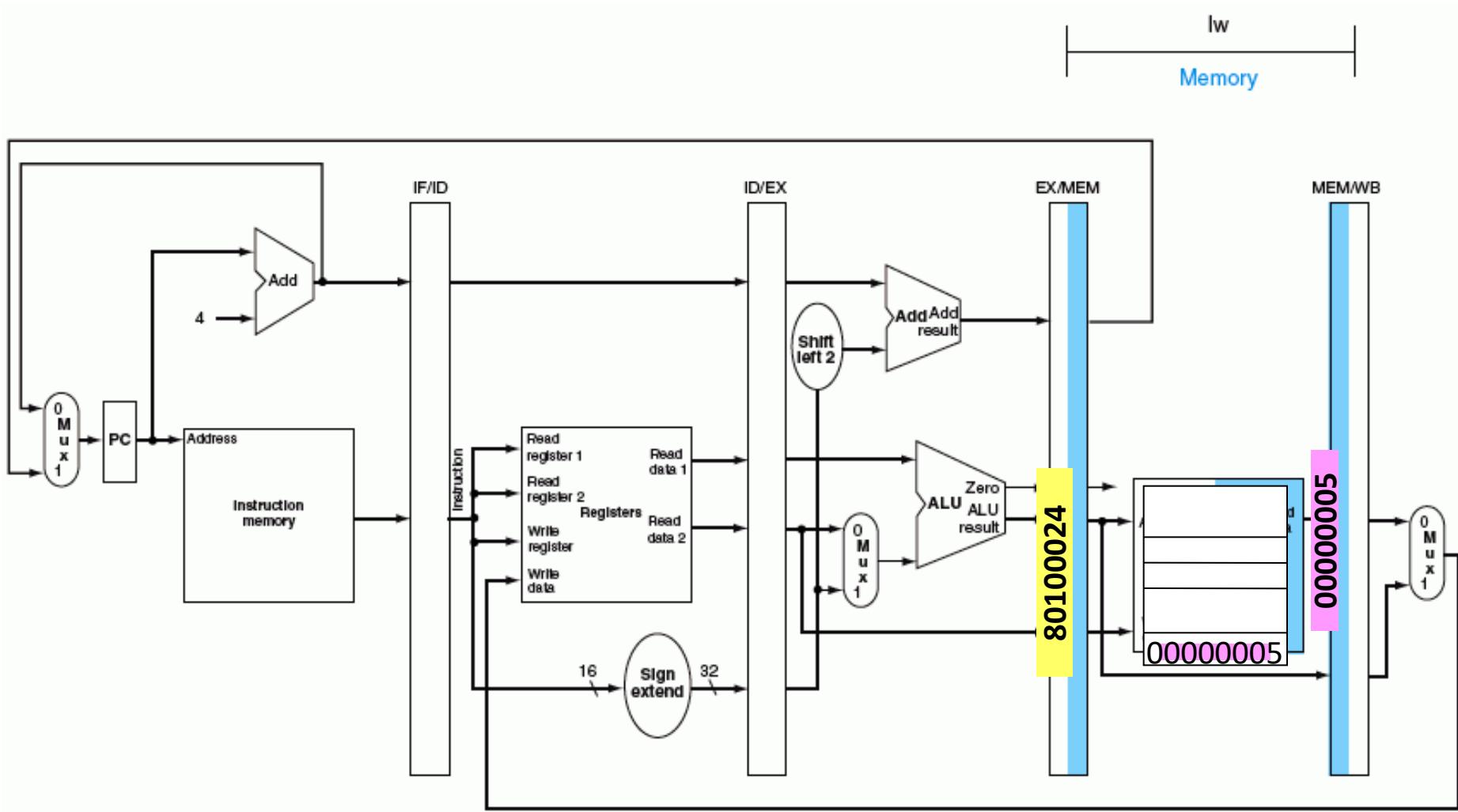
PC + 4 armazenado para ser usado, caso instrução corrente seja beq



# Operação de um Pipeline: lw (Execução)

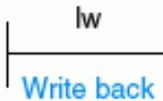


# Operação de um Pipeline: Iw (Acesso a Memória)



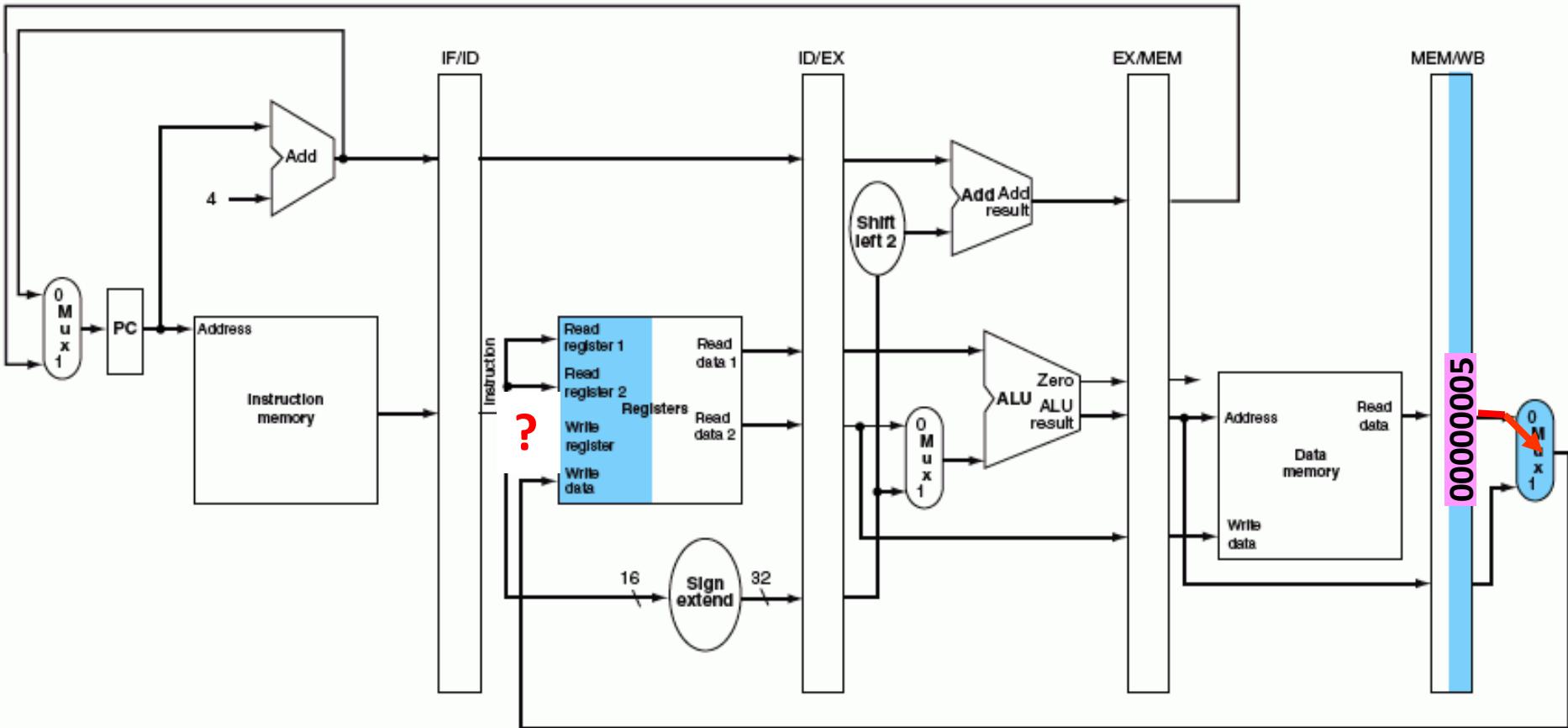
# Operação de um Pipeline: lw (Escrita em Registrador)

**Problema: Número de registrador de escrita não foi guardado no pipeline**



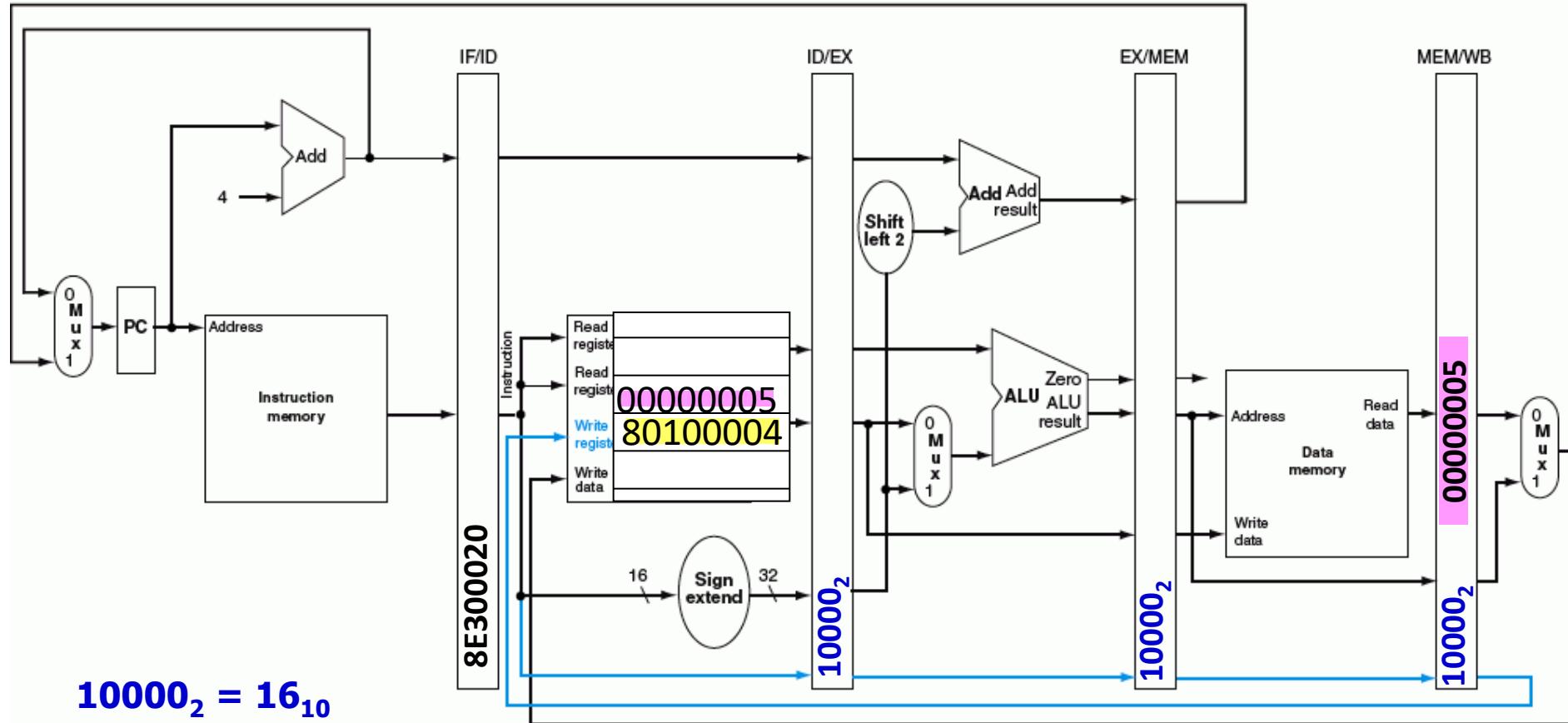
lw

Write back



# Operação de um Pipeline: Iw (Escrita em Registrador)

Corrigindo o problema: Número de registrador de escrita é armazenado nos registradores do pipeline em cada estágio do pipeline



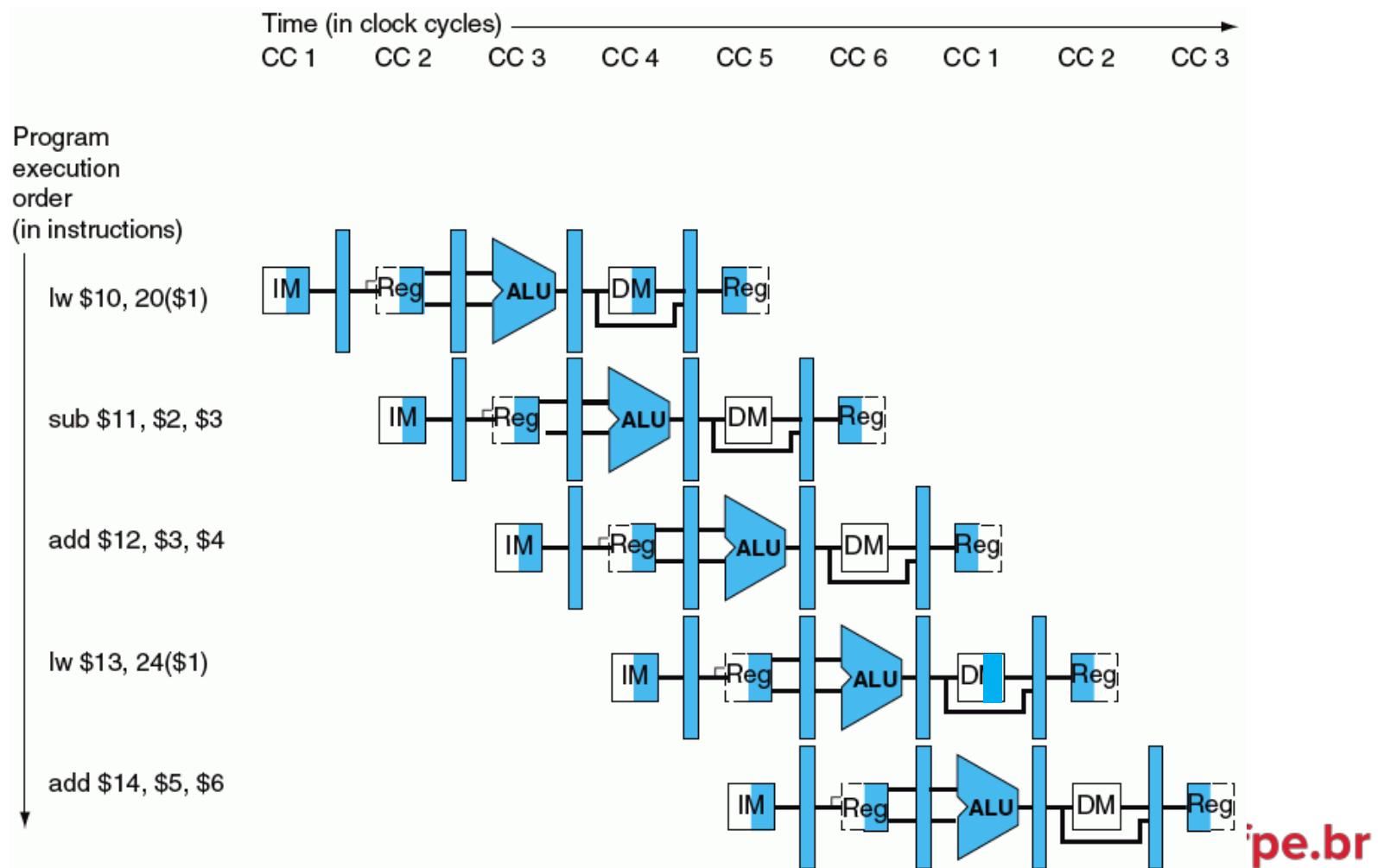
$$10000_2 = 16_{10}$$

# Representação Gráfica de Pipelines

- Fluxo de processamento em um pipeline pode ser difícil de entender
- É comum representar pipelines usando 2 tipos de diagramas
  - Diagrama de múltiplos ciclos de clock
    - Mostra a operação do pipeline ao longo do tempo
  - Diagrama de um único ciclo de clock
    - Mostra uso do pipeline em um único ciclo
    - Destaca recursos utilizados por cada instrução no pipeline em um determinado ciclo

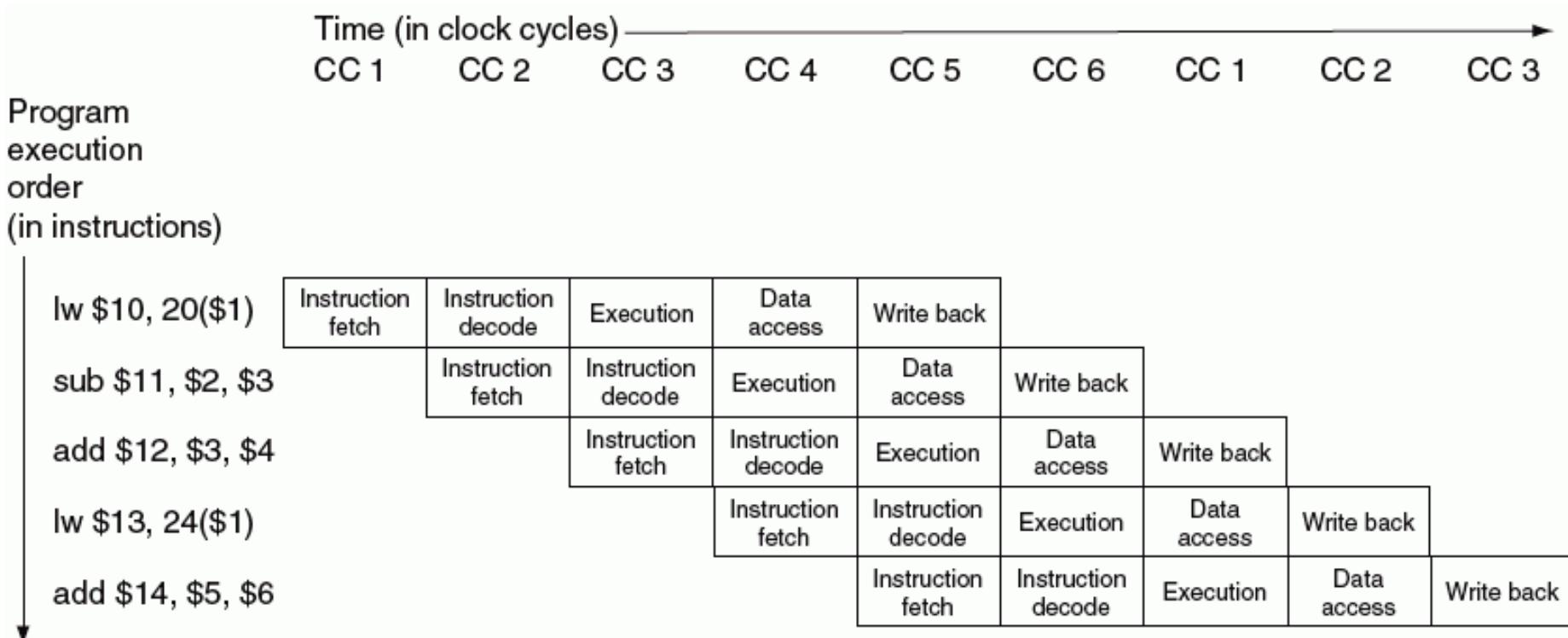
# Diagrama de Múltiplos Ciclos de Clock

- Versão não convencional que mostra a utilização de recursos em cada estágio



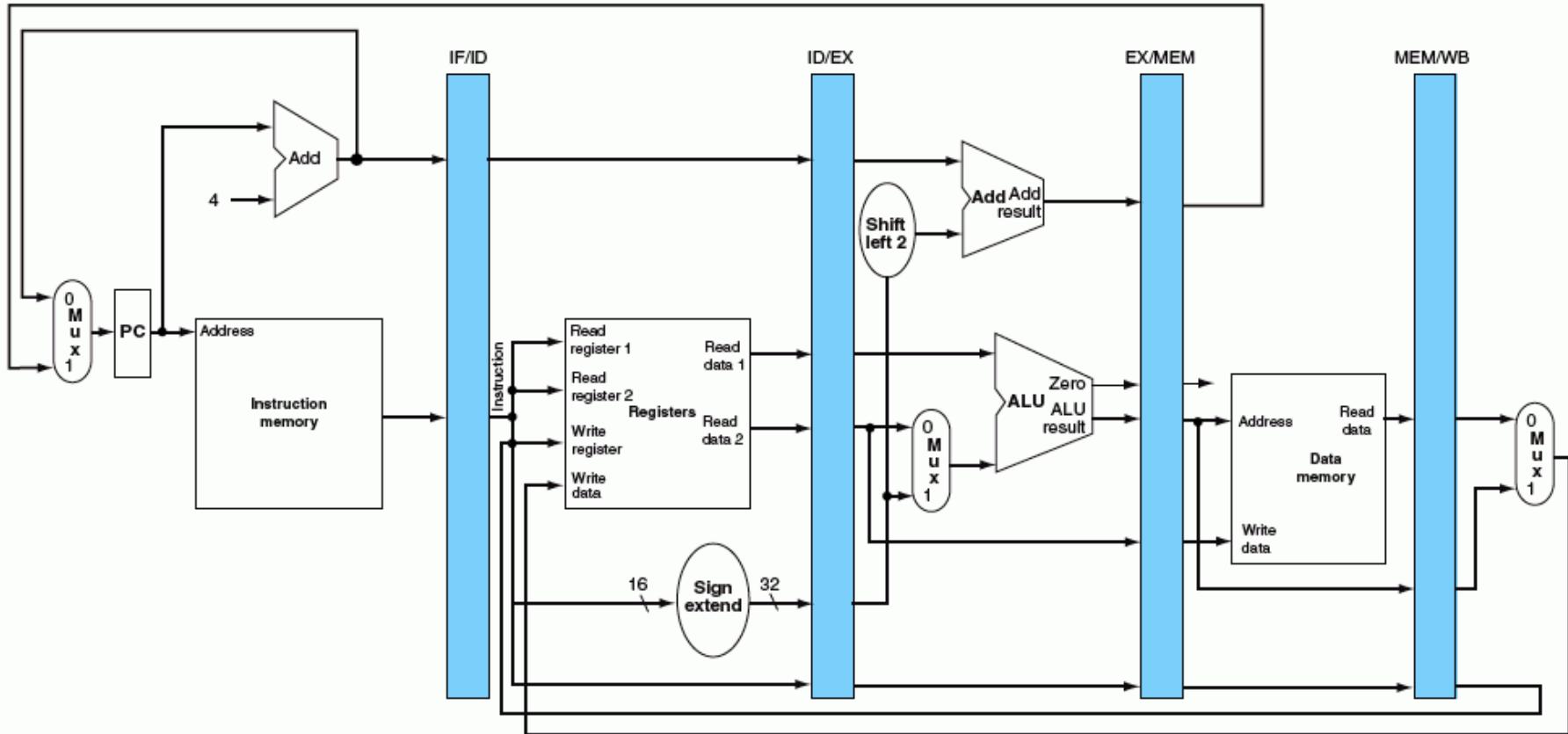
# Diagrama de Múltiplos Ciclos de Clock

- Versão tradicional identifica cada estágio pelo nome

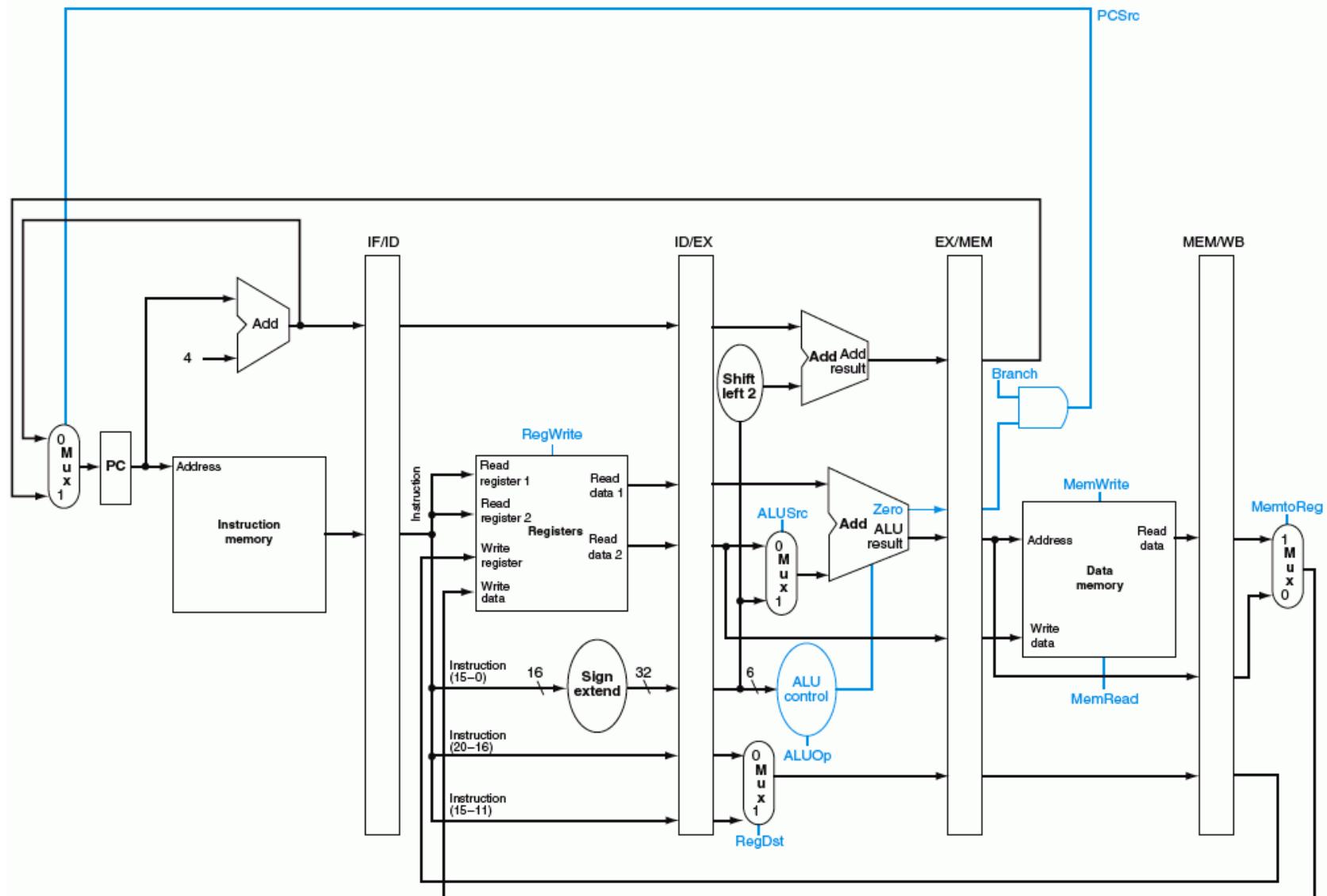


# Diagrama de Único Ciclo de Clock

- Mostra o estado do pipeline em um dado ciclo
  - Identifica recursos utilizados por cada instrução no dado ciclo



# Implementando Pipeline: Sinais de Controle



# Implementando o Controle do Pipeline

- Sinais de controle derivados da instrução
  - Como na implementação monociclo
- Boa parte dos sinais de controle da implementação monociclo serão aproveitados
  - ALU
  - Desvios
  - Multiplexadores para escolha do fonte do dado do registrador destino, etc

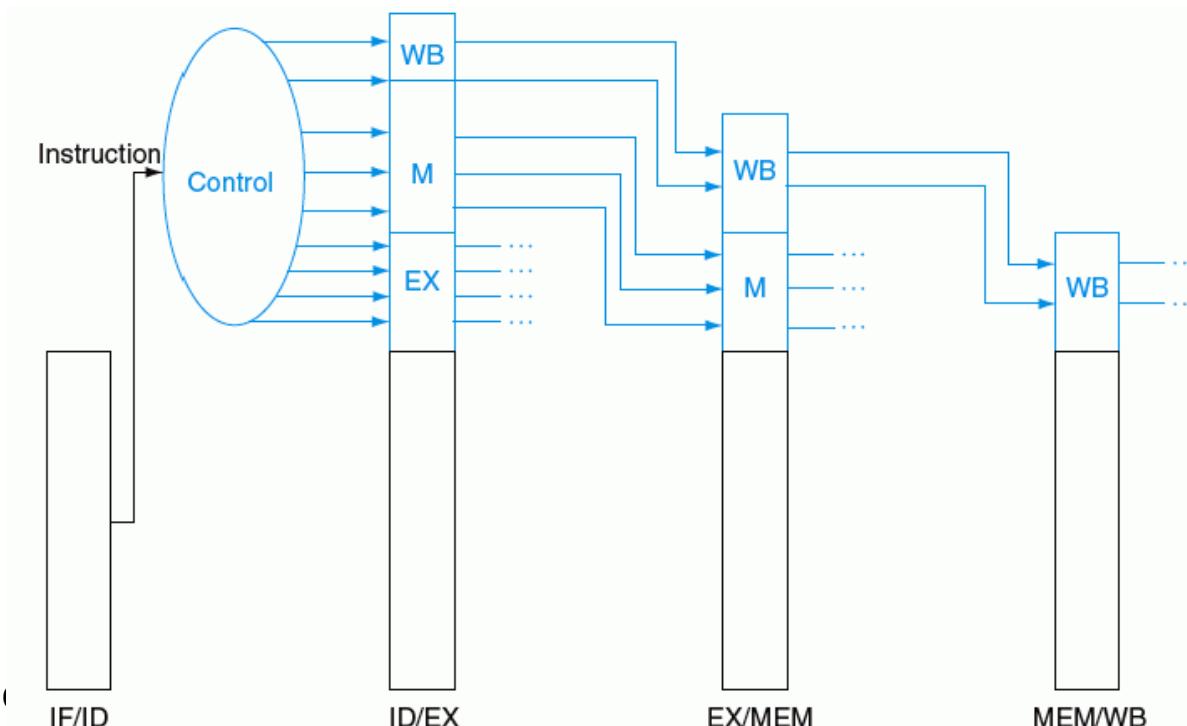
# Reagrupando Sinais de Controle da Implementação Monociclo

- Sinais de controle são essencialmente os mesmos, porém precisam ser repassados aos estágios juntamente com a instrução

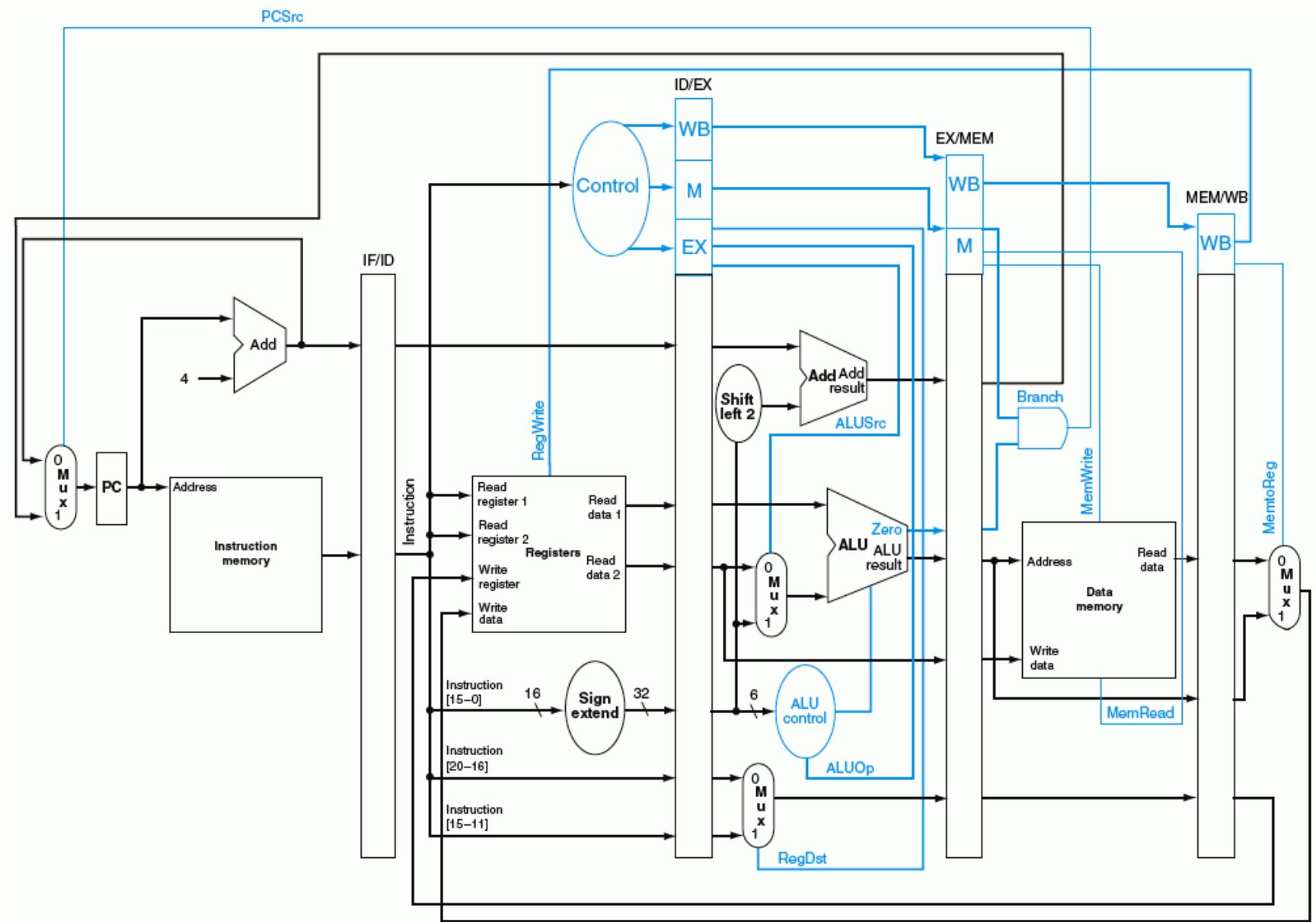
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

# Como Controlar Cada Estágio?

- Sinais de controle são repassados aos registradores do pipeline
  - Durante o estágio de decodificação, sinais de controle para o resto dos estágios podem ser gerados e armazenados
  - A cada ciclo do clock, o registrador corrente passa os sinais para o registrador do próximo estágio



# Processador Pipeline Completo



# Pipeline e Projeto de ISA

- ISA do MIPS projetada para pipeline
  - Todas as instruções são de 32 bits
    - Mais fácil de buscar e decodificar em um ciclo
    - Contra-exemplo Intel x86: tem instruções variando de 1 a 17 bytes
  - Poucas instruções e instruções regulares
    - Permite decodificação e leitura de registradores em um estágio
  - Endereçamento do Load/store
    - Permite cálculo de endereço no terceiro estágio e acesso no quarto estágio
- Projeto de ISA afeta a complexidade de implementação do pipeline

# Algumas Análises sobre Pipeline ...

- Pode aumentar o custo de hardware:
  - Duplicação de hardware para evitar que instruções diferentes utilizem o mesmo recurso em um ciclo
    - Ex: Memória
  - Maior quantidade de registradores para guardar estado de processamento da instrução
- Hardware pode ser utilizado mais eficientemente
  - Evita que recursos fiquem ociosos

# Algumas Análises sobre Pipeline ...

- Melhora de desempenho por utilizar pipeline:
  - estágios com a mesma duração
  - instruções independentes de resultados calculados em instrução anterior
  - execução sequencial das instruções

*... mas as características acima quase sempre não são satisfeitas:*

- *dependência de dados*
- *instruções de desvio*

# Infraestrutura de Hardware

## Conflitos no Pipeline

Prof. Adriano Sarmento

# Conflitos

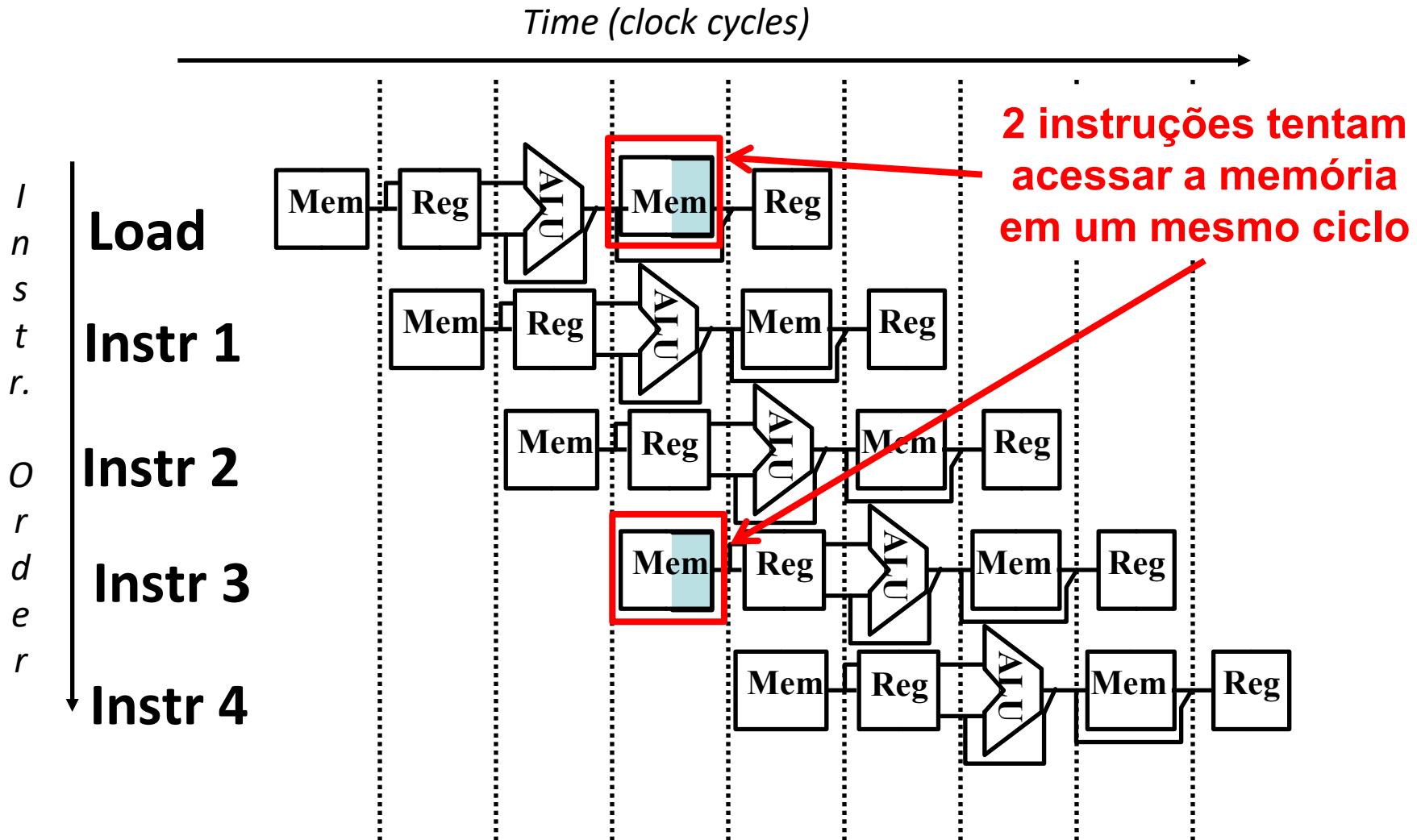
- Situações que evitam que uma nova instrução seja iniciada no próximo ciclo
- Tipos:
  - Estruturais
    - Recurso necessário para execução de uma instrução está ocupado
  - Dados
    - Dependência de dados entre instruções
  - Controle
    - Decisão da próxima instrução a ser executada depende de uma instrução anterior

# Conflitos Estruturais

- Conflito pelo uso de um recurso
- Hardware não permite que determinadas combinações de instruções sejam executadas em um pipeline
- Utilização de uma só memória para dados e instruções é um exemplo
  - Load/store requer acesso a dados
  - Estágio de busca de instrução teria que esperar o load/store terminar o acesso a memória para começar

**Solução comum:  
Replicar recursos**

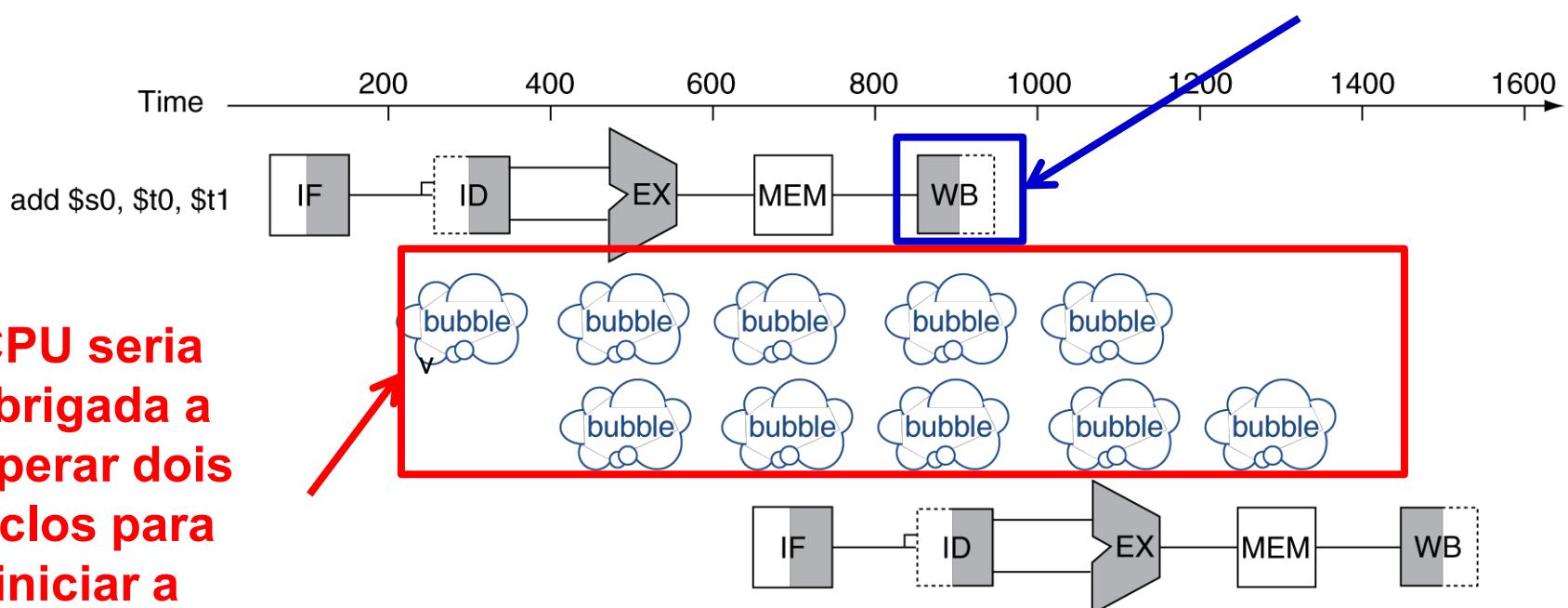
# Exemplo de Conflito Estrutural: Memória Única



# Conflito de Dados

- Uma instrução para ser executada depende de um dado gerado por uma instrução anterior
  - **add      \$s0 , \$t0 , \$t1**
  - sub      \$t2 , \$s0 , \$t3**

**Resultado da soma  
só será escrito em  
\$s0 neste ciclo**



# Resolvendo Conflitos de Dados

- Soluções em software (compilador/montador)
  - Inserção de NOPs
  - Re-arrumação de código
- Soluções em hardware
  - Método de Curto-circuito (Forwarding)
  - Inserção de retardos (stalls)

# Inserção de NOPs no Código

- Compilador/Montador deve identificar conflitos de dados e evitá-los inserindo NOPs no código

## Conflitos de dados

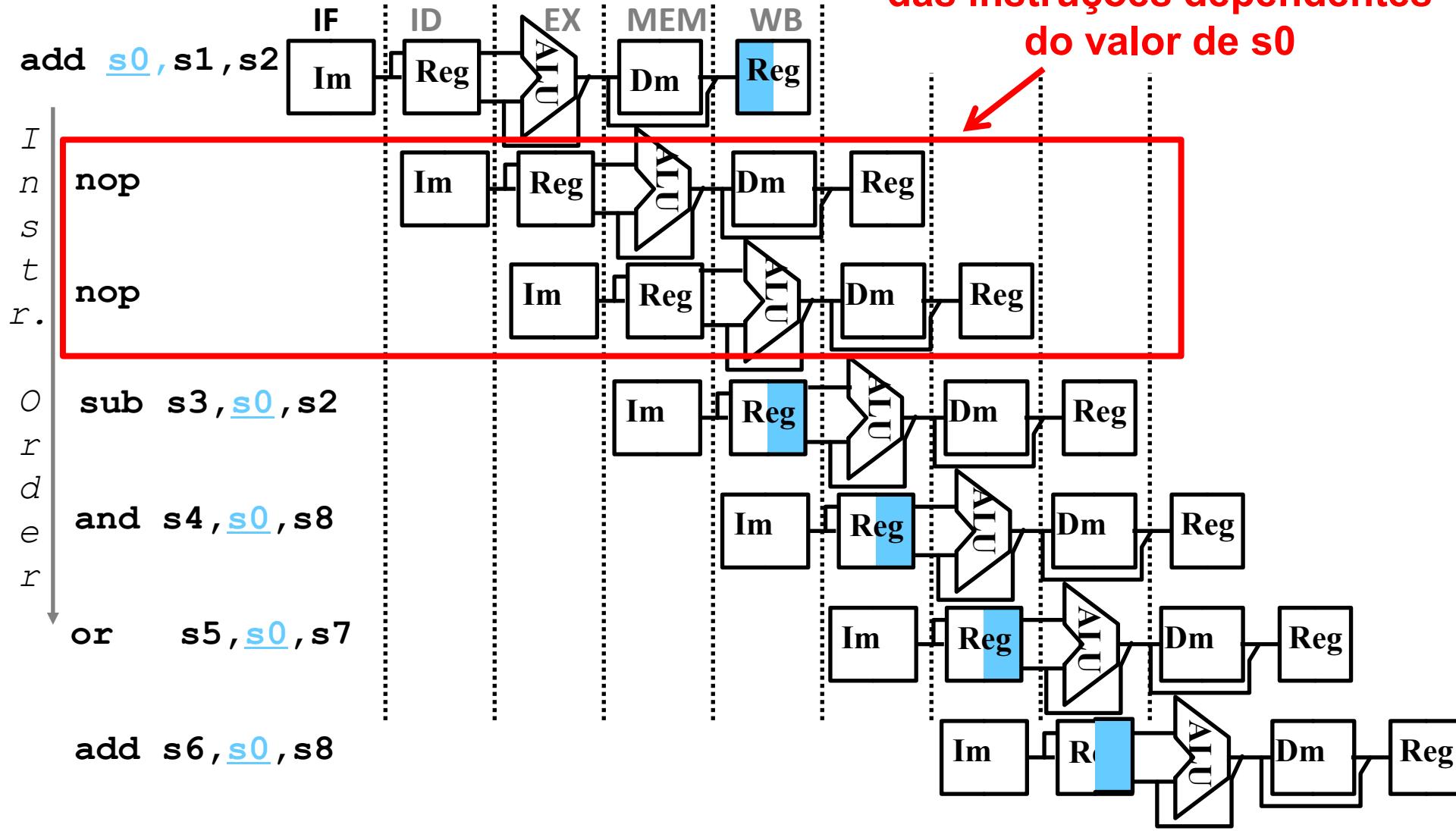
```
add $s0,$s1,$s2
sub $s3,$s0,$s2
and $s4,$s0,$s8
or $s5,$s0,$s7
add $s6,$s0,$s8
```



```
add $s0,$s1,$s2
nop
nop
sub $s3,$s0,$s2
and $s4,$s0,$s8
or $s5,$s0,$s7
add $s6,$s0,$s8
```

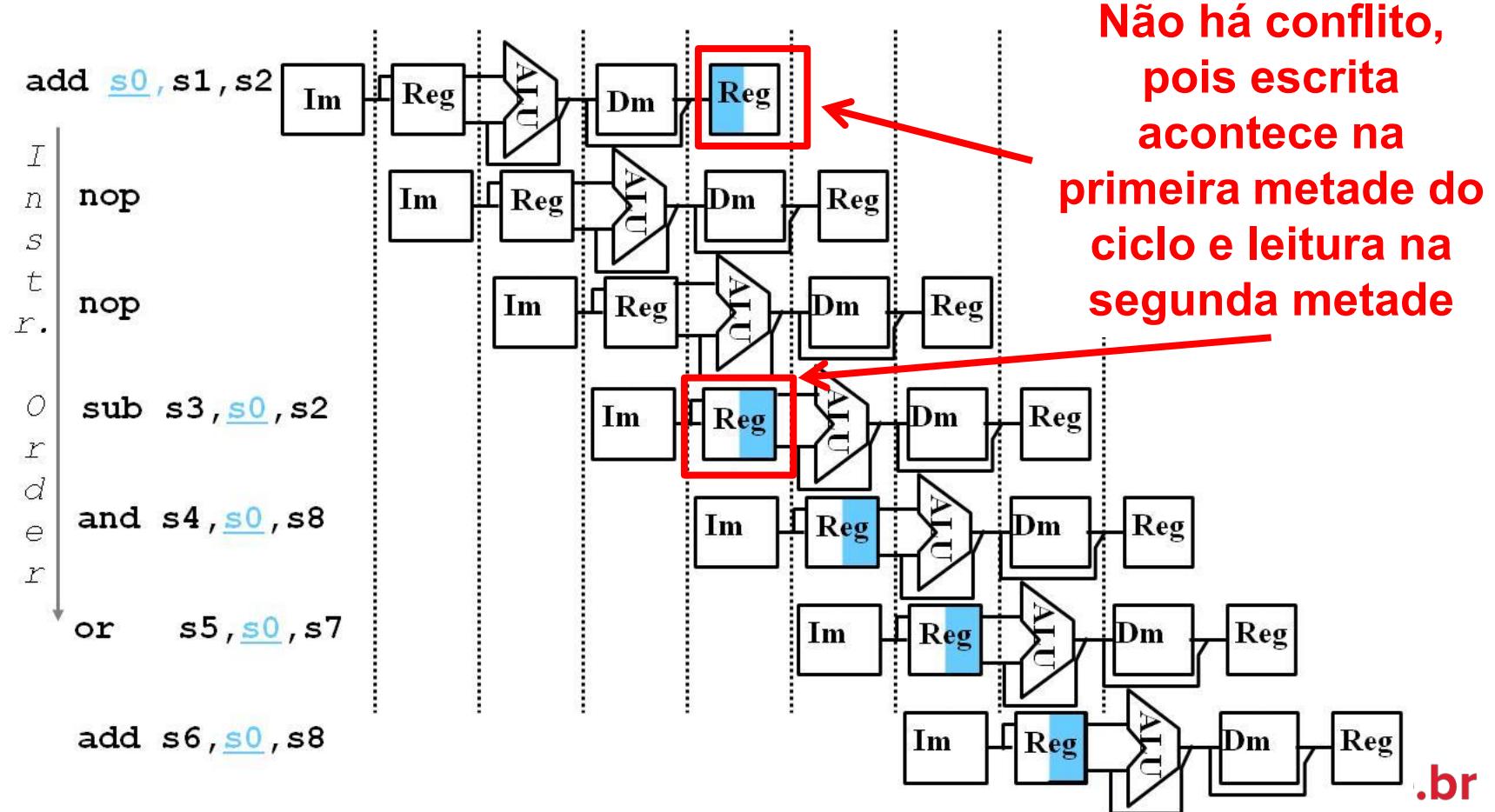
# Inserção de NOPs

NOPs retardam a execução das instruções dependentes do valor de s0



# Escrita e Leitura de Banco de Registradores no Mesmo Ciclo

- Banco de registradores permite a leitura de dois registradores e a escrita de um registrador simultaneamente



# Re-arrumação do Código

- Compilador/Montador deve identificar conflitos de dados e evitá-los re-arrumando o código
  - Executa instruções que não tem dependência de dados e que a ordem de execução não altera a corretude do programa

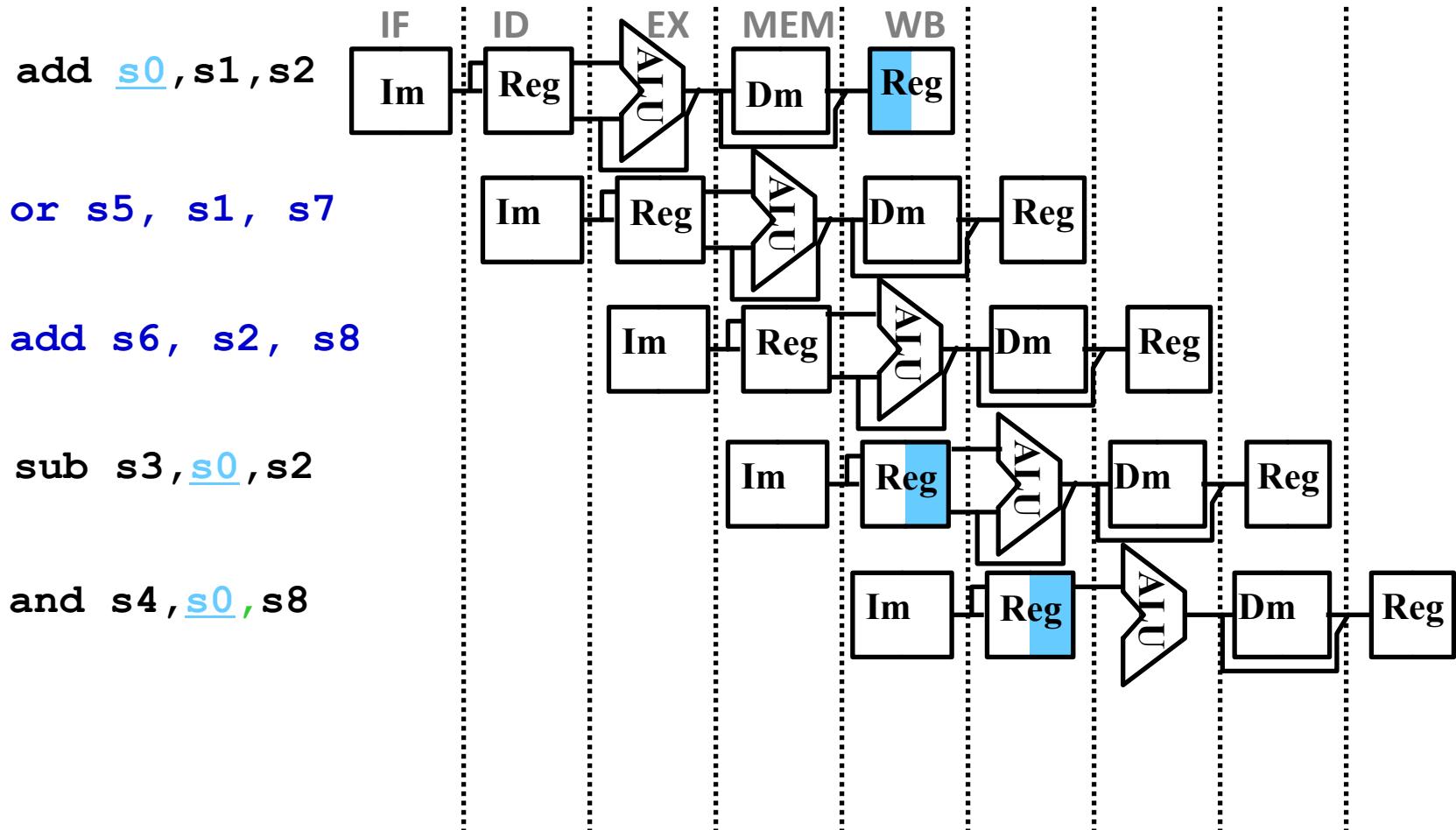
## Conflitos de dados

```
add $s0 $s1,$s2
sub $s3,$s0,$s2
and $s4,$s0 $s8
or $s5,$s1,$s7
add $s6,$s2,$s8
```



```
add $s0,$s1,$s2
or $s5,$s1,$s7
add $s6,$s2,$s8
sub $s3,$s0,$s2
and $s4,$s0,$s8
```

# Re-arrumação do Código

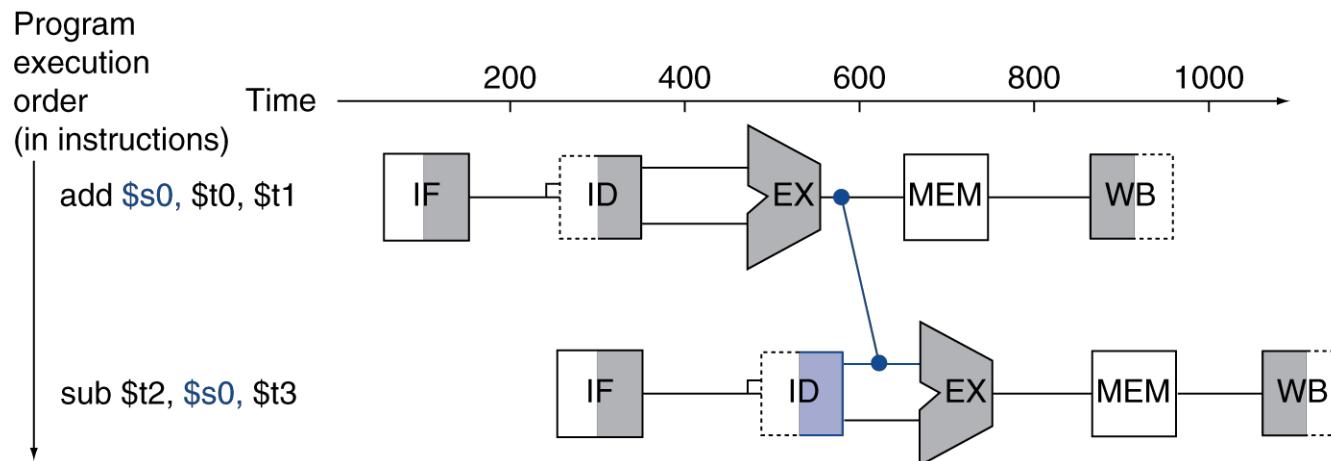


# Análise de Soluções em SW para Conflitos de Dados

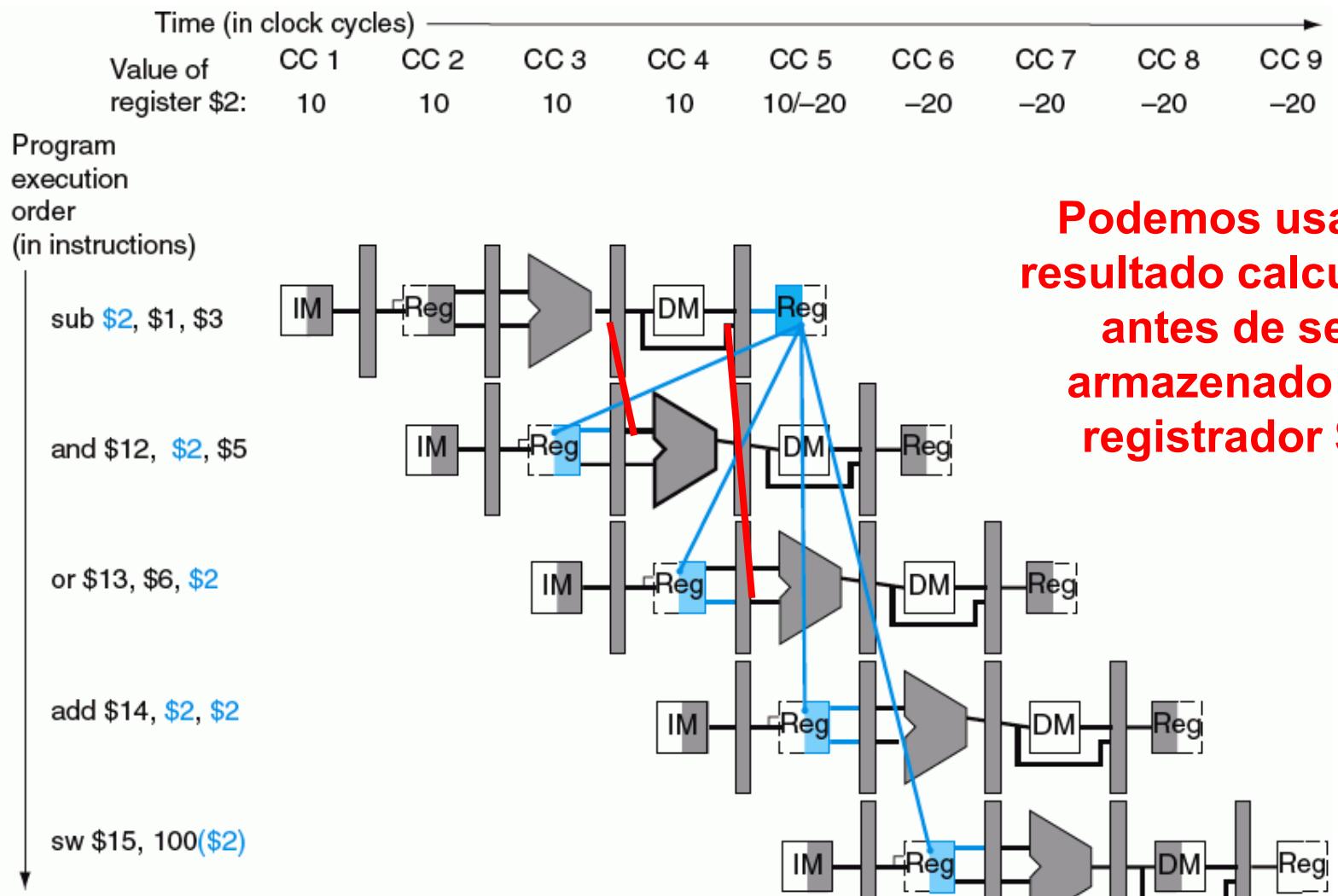
- Requerem compilador/montador “inteligente” que detecte conflitos de dados
  - Modifica o código para evitar conflitos
  - Mantem funcionamento correto do programa
  - Implementação do Compilador/Montador requer bom conhecimento do pipeline
- Inserção de NOPs
  - Insere retardos no programa
  - Degrada desempenho do sistema
- Re-arrumação de código
  - Não compromete desempenho
  - Mais complexidade na implementação do compilador/montador

# Método do Curto-Circuito (Forwarding ou Bypassing)

- Usa o resultado desejado assim que é computado
  - Não espera ser armazenado no registrador
  - Requer conexões extras na unidade de processamento



# Dependências e Forwarding



**Podemos usar o resultado calculado antes de ser armazenado no registrador \$2**

# Usando Forwarding (Representação Tradicional)

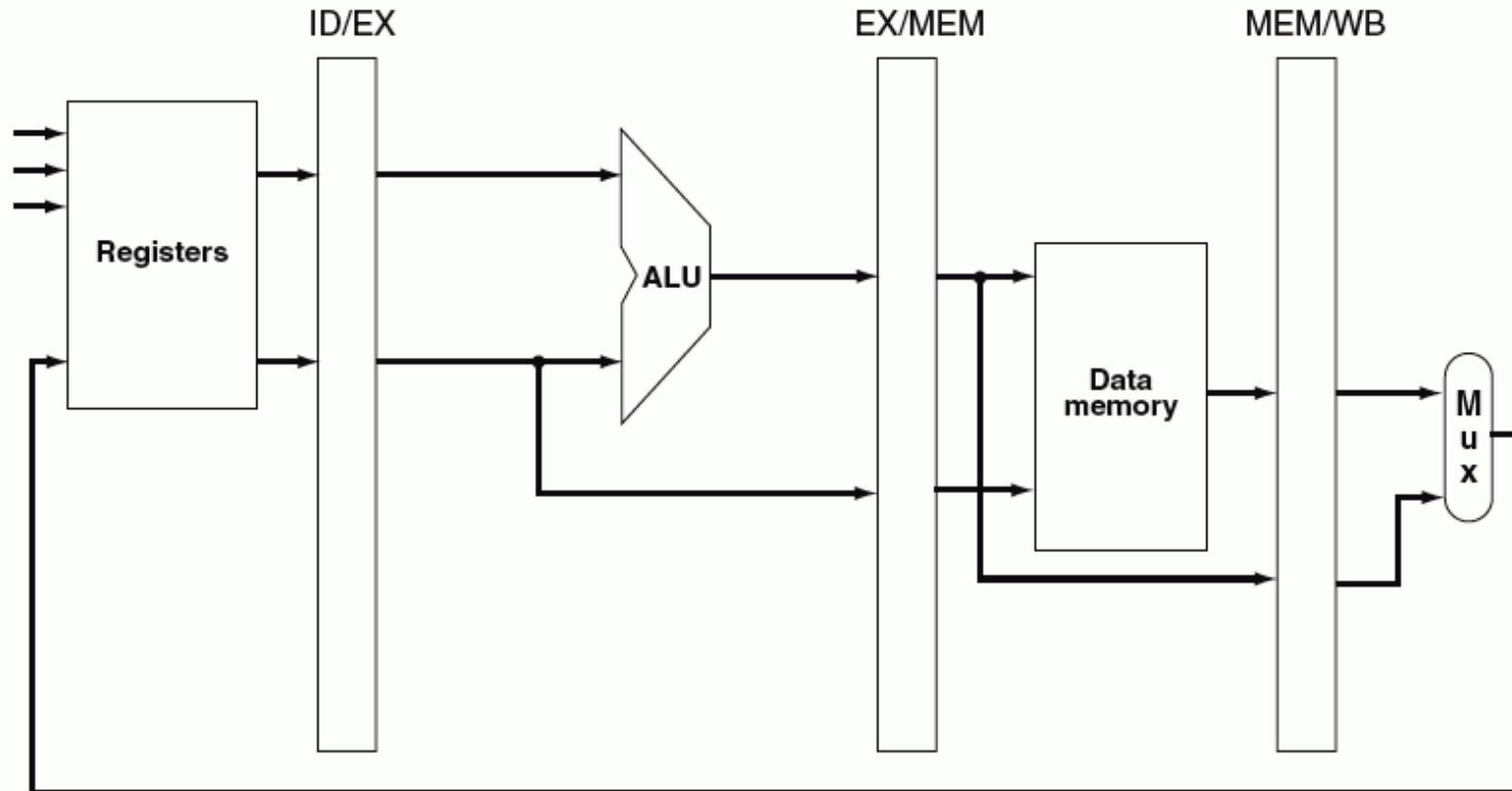
	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	

**I1:** sub \$2, \$1,\$3  
**I2:** and \$12, \$2, \$5  
**I3:** or \$13, \$6, \$2  
**I4:** and \$14, \$2, \$2

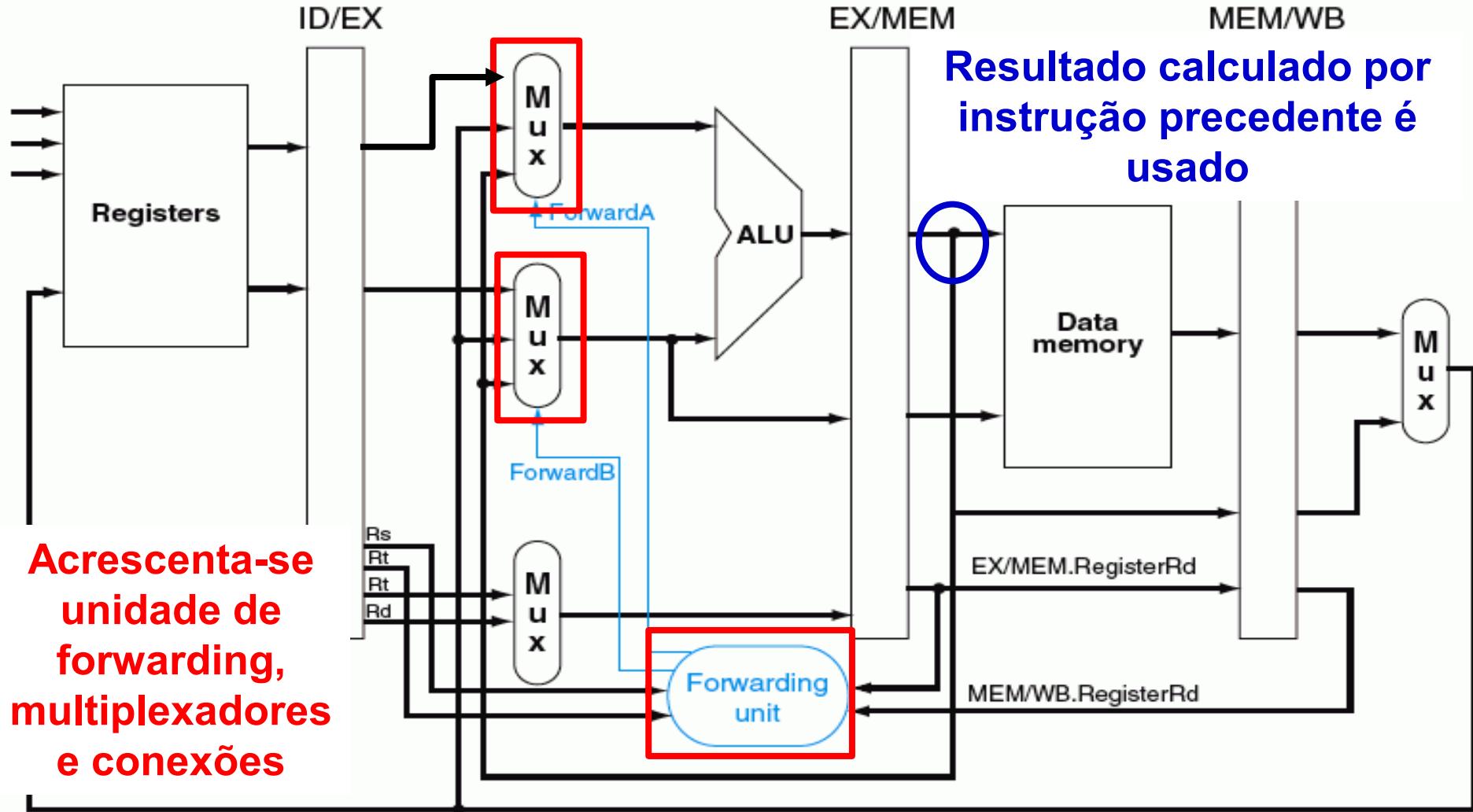
# Como Implementar Forwarding?

- Idéia é acrescentar HW com uma lógica capaz de detectar conflitos de dados e controlar unidade de processamento para realizar o forwarding
- Deve-se acrescentar mais conexões para permitir que resultados possam ser utilizados antes de escritos no banco de registradores
- Possivelmente, acrescenta-se multiplexadores para que outros estágios possam selecionar a fonte do operando
  - Banco de registradores ou resultado gerado por outra instrução anterior no pipeline

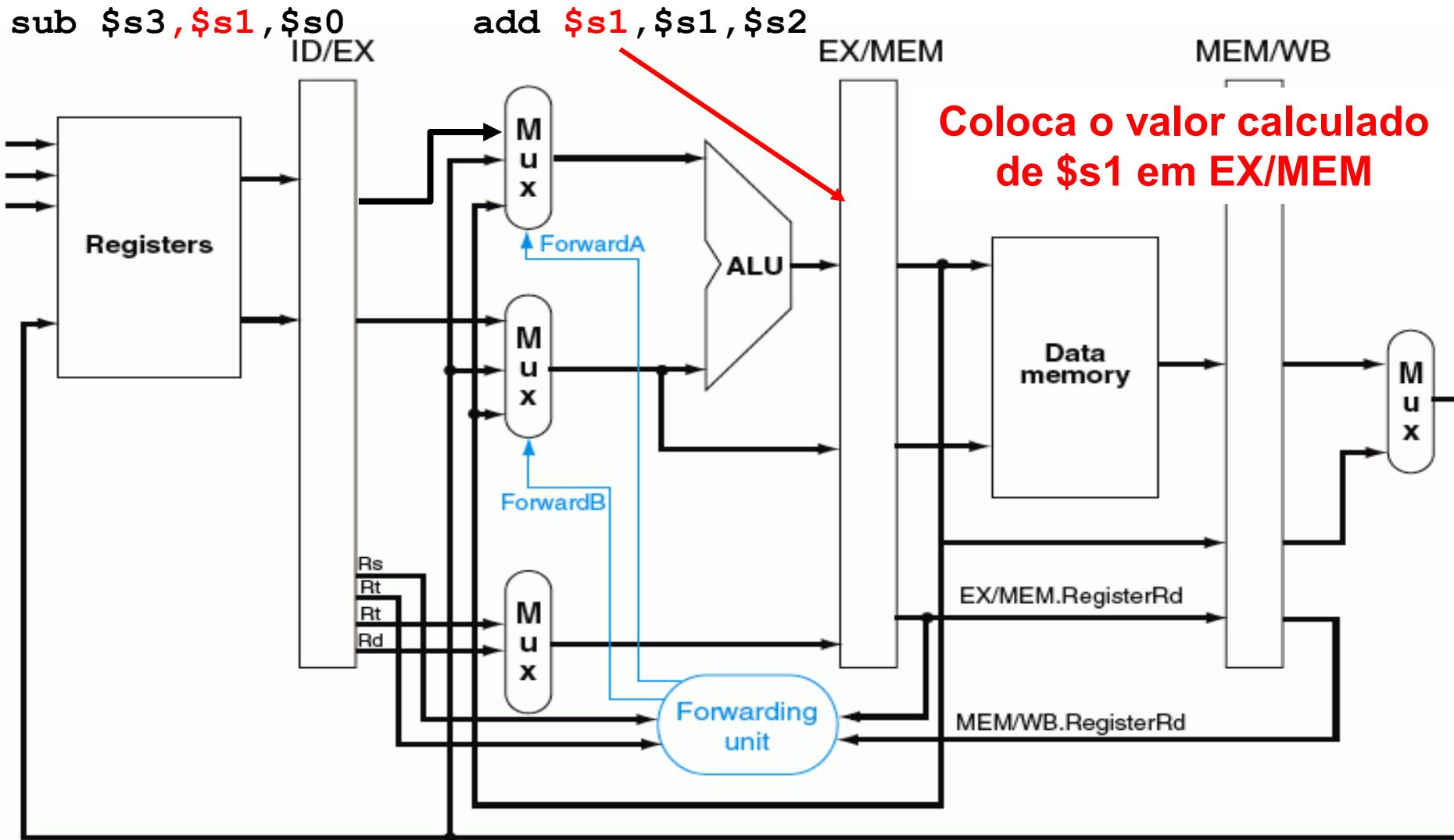
# Datapath Simplificado Sem Forwarding



# Datapath Simplificado Com Forwarding (Sem Considerar sw e Immediato)



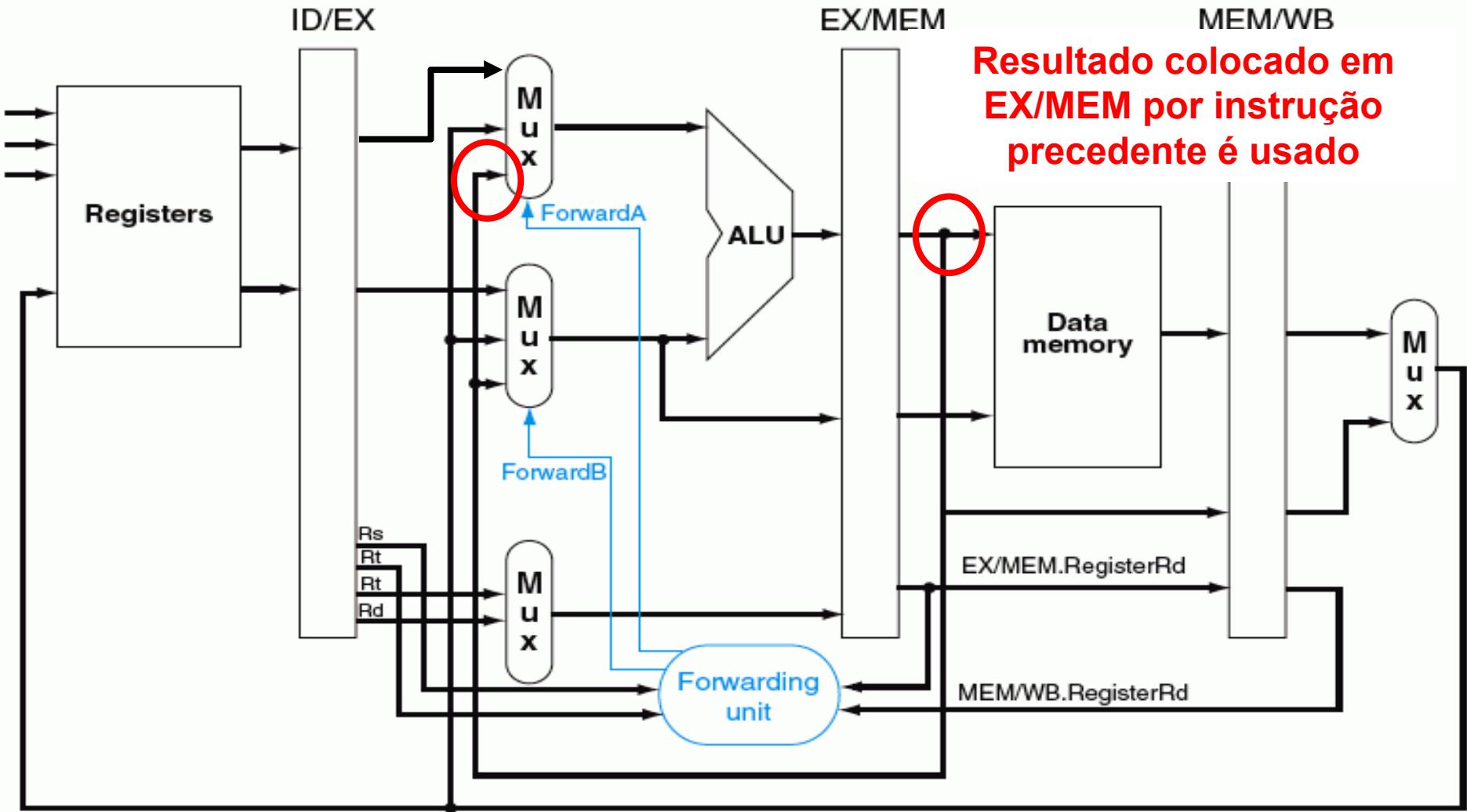
# Forwarding (Sem Considerar sw e Imediato)



# Forwarding (Sem Considerar sw e Imediato)

sub \$s3,\$s1,\$s0

add \$s1,\$s1,\$s2



# Considerando sw e o imediato

- Considere a sequência:

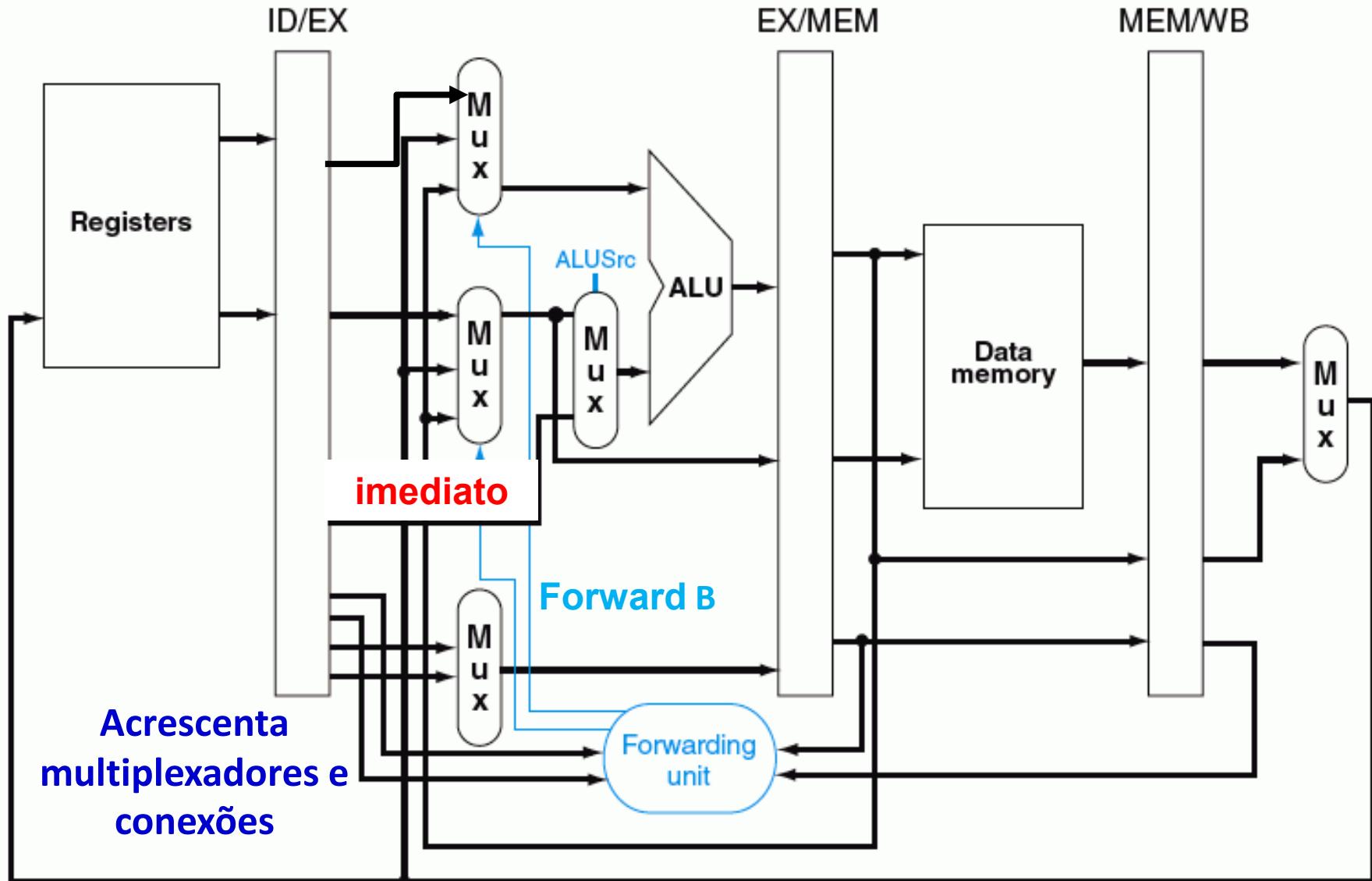
```
add $1,$1,$2
```

```
sw $1,16 ($3)
```

```
addi $2, $2, 256
```

- Faltam componentes no datapath anterior para fazer o forward para o **sw** ou para fazer uma simples instrução com uso do imediato.

# Datapath Simplificado Com Forwarding ( Considerando store (sw) e Imediato)



# Como Detectar Necessidade de Forward?

- Passar números de registradores ao longo do pipeline
  - Exemplo: ID/EX.RegisterRs = número do registrador Rs armazenado no registradorID/EX do pipeline
- Números dos registradores que são operandos da ALU no estágio EX são dados por
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Conflitos ocorrem quando
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

# Como Detectar Necessidade de Forward?

- Mas só vai dar um forward se a instrução for escrever em um registrador
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- E somente se o Rd para a instrução não for \$zero
  - EX/MEM.RegisterRd  $\neq 0$ ,
  - MEM/WB.RegisterRd  $\neq 0$

# Condições de Forwarding

- Forwarding do resultado do estágio EX
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
**ForwardA = 10**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
**ForwardB = 10**
- Forwarding do resultado do estágio MEM
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
**ForwardA = 01**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
**ForwardB = 01**

# Conflito Duplo de Dados

- Considere a sequência:

add \$1,\$1,\$2

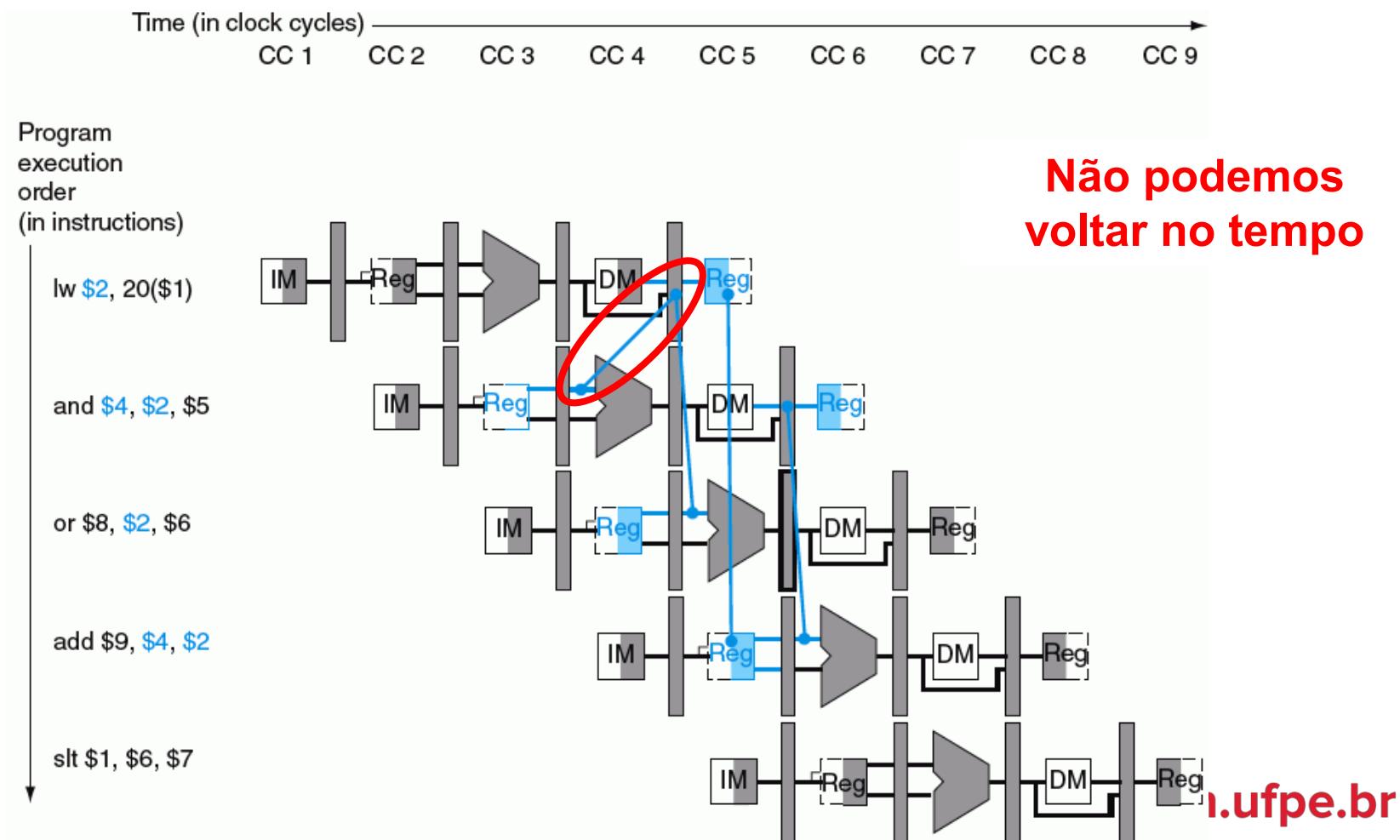
add \$1,\$1,\$3

add \$1,\$1,\$4

- O terceiro add deve pegar resultado do segundo add, não do primeiro
  - Resultado do primeiro add está em MEM/WB
  - Resultado do segundo add está em EX/MEM
- Deve-se reescrever condição de forward do estágio MEM
  - Somente se condição de forward de EX for falsa

# Conflito de Dados Pelo Uso do Load

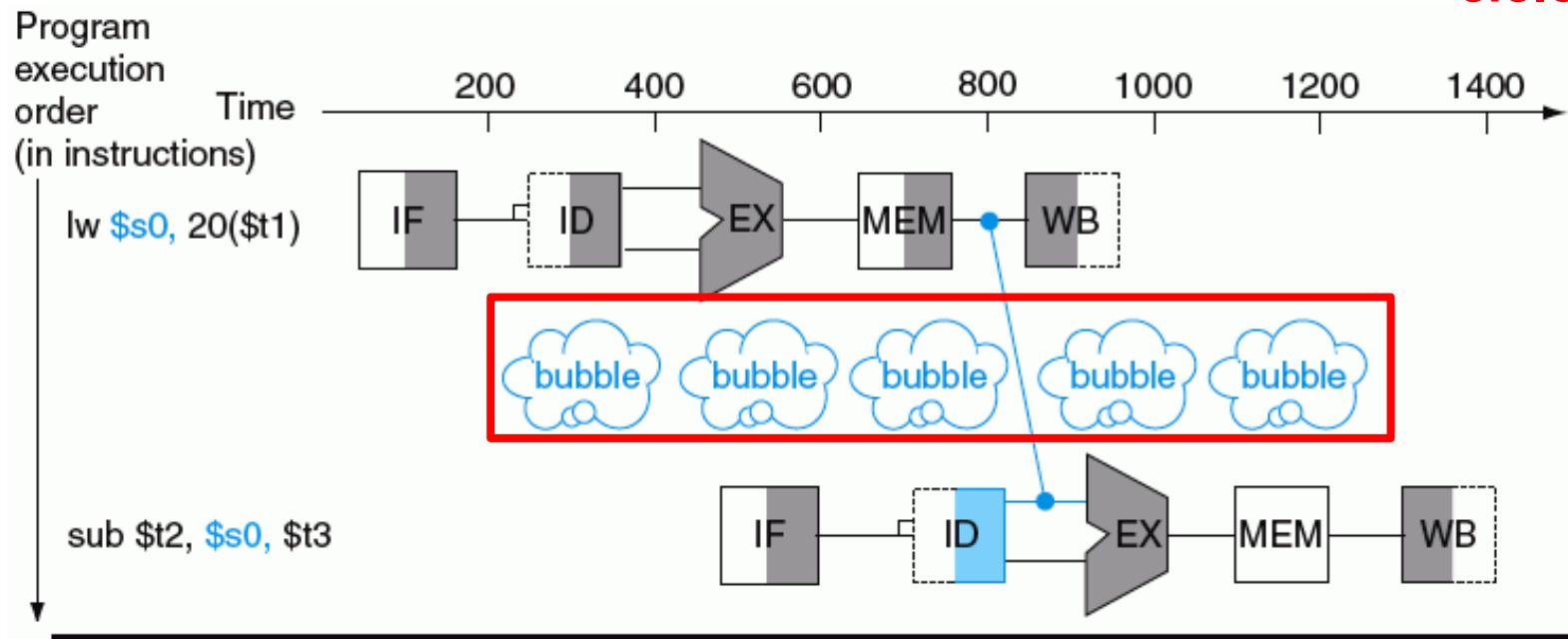
- Nem sempre se pode utilizar forwarding
  - Se o valor ainda não tiver sido computado quando necessário



# Inserção de Retardos

- Quando não podemos utilizar forwarding para resolver conflitos, inserimos retardos

**Retarda a execução  
da próxima  
instrução em um  
ciclo**



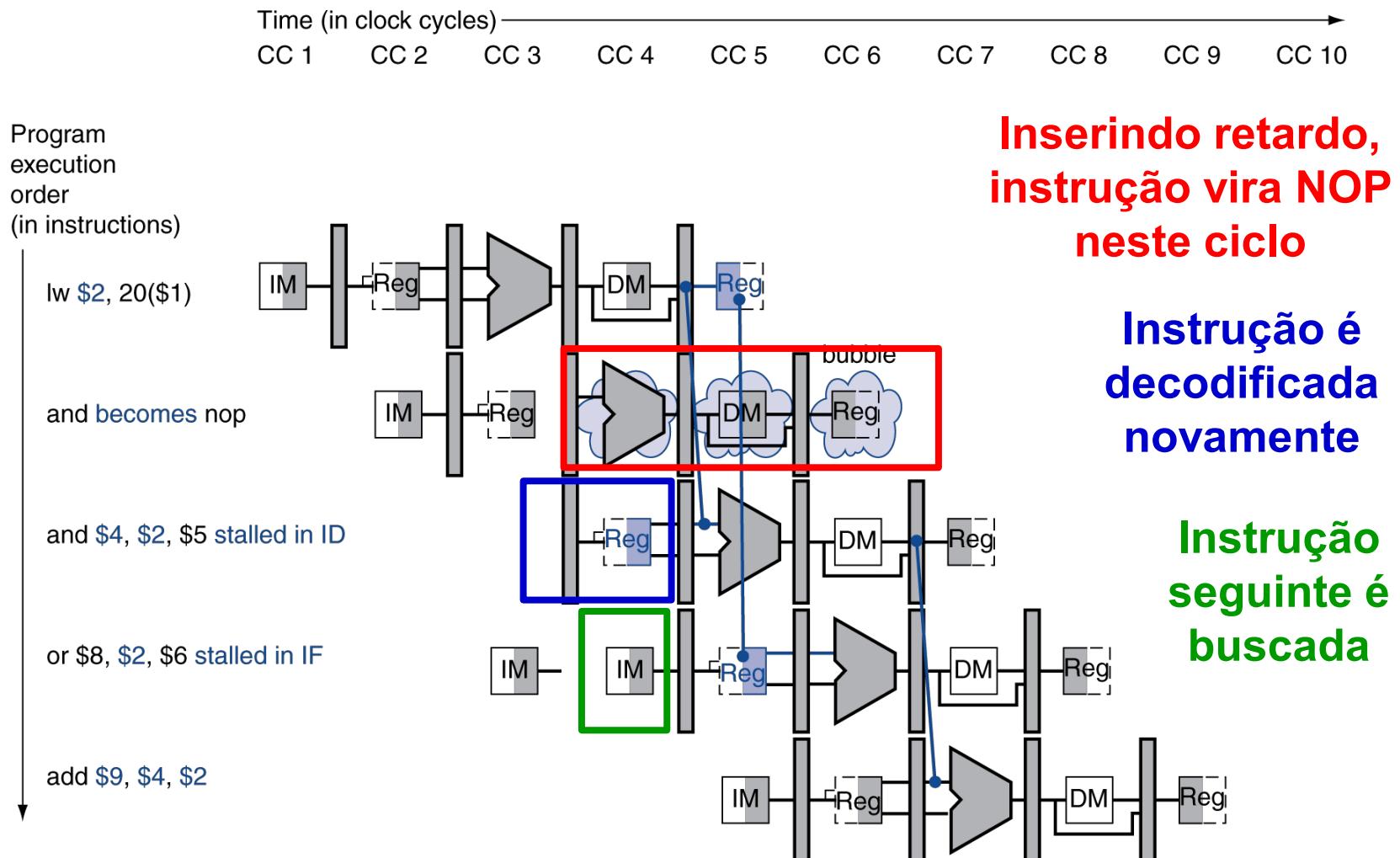
# Como Detectar Este Tipo de Conflito de Dado?

- Verificar se instrução depende do load no estágio ID
- Números dos registradores do operandos da ALU são dados por:
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Conflito acontece quando
  - ID/EX.MemRead and
$$((ID/EX.RegisterRt = IF/ID.RegisterRs) \text{ or } (ID/EX.RegisterRt = IF/ID.RegisterRt))$$
- Se detectado, insira um retardo

# Como Inserir Retardos em um Pipeline?

- Forçar sinais de controle no registrador ID/EX para terem valor 0
  - EX, MEM and WB
  - Instrução que depende do load se torna um nop
- Não permitir a atualização do PC e do registrador IF/ID
  - Instrução é decodificada de novo
  - Instrução seguinte é buscada novamente
  - Retardo de 1 ciclo permite que MEM leia dado do load
    - Depois se pode utilizar um forward do estágio MEM

# Inserindo um Retardo no Pipeline

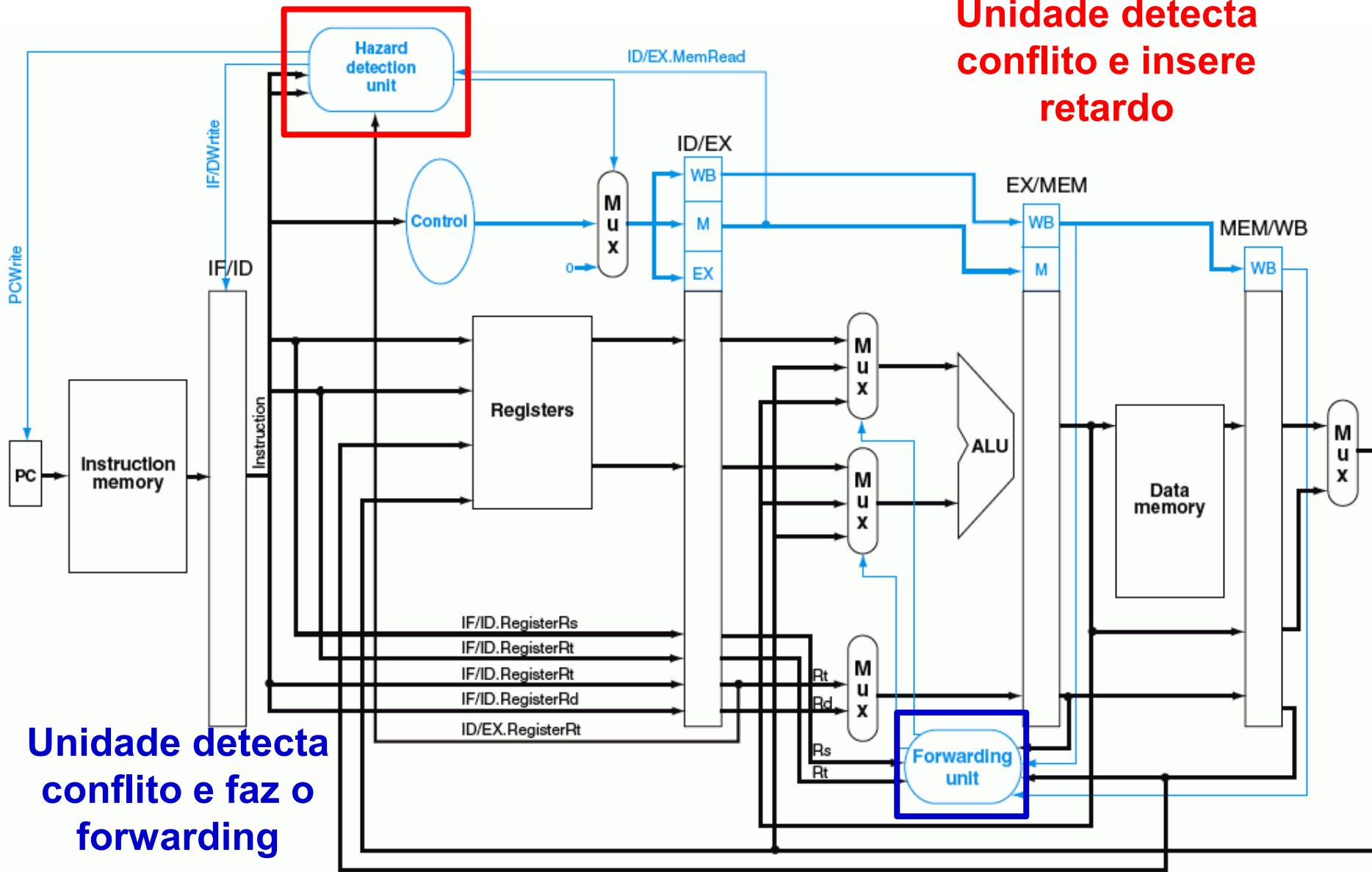


# Inserindo um Retardo no Pipeline (Representação Tradicional)

	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	X	EX	MEM	WB		
I3			IF	X	ID	EX	MEM	WB	
I4					IF	ID	EX	MEM	WB

**I1: lw \$2, 20(\$1)**  
**I2: and \$4, \$2, \$5**  
**I3: or \$8, \$2, \$6**  
**I4: add \$9, \$4, \$2**

# Datapath com Unidades de Retardo e Forwarding



# Análise de Soluções em HW para Conflitos de Dados

- Curto-circuito (Forwarding)
  - Não degrada desempenho
  - Requer custo um pouco maior de hardware
  -
- Inserção de Retardos
  - Similar a solução de SW de inserção de NOPs
  - Degrada desempenho do sistema
- Pode-se utilizar os dois métodos em conjunto
  - Deve-se usar inserção de retardos, somente quando não é possível utilizar forwarding

# Infraestrutura de Hardware

Melhorando o Desempenho do Pipeline –  
Processadores Superpipeline,  
Superescalares e VLIW

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Pipeline

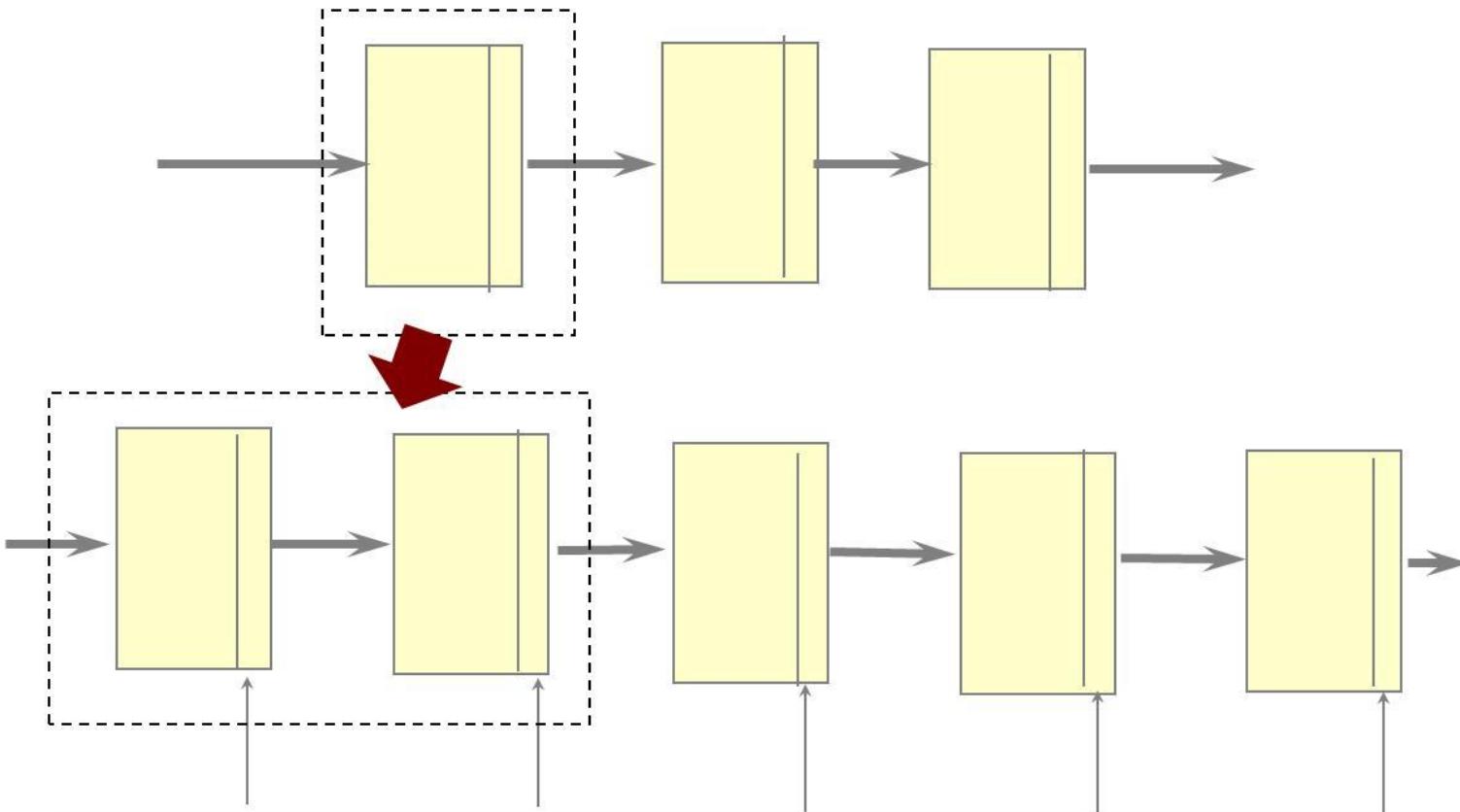
- Pipeline é uma técnica que visa aumentar o nível de paralelismo de execução de instruções
  - ILP (Instruction-Level Parallelism)
- Permite que várias instruções sejam processadas simultaneamente com cada parte do HW atuando numa instrução distinta
  - Instruções quebradas em estágios
  - Sobreposição temporal
- Visa aumentar desempenho
  - Latência de instruções é a mesma ou maior
  - Throughput aumenta
- Tempo de execução de instrução é o mesmo ou maior, MAS tempo de execução de programa é menor

# Como Melhorar Desempenho de Pipeline?

- Aumentando o número de estágios
  - Estágios de menor duração
  - Frequência do clock maior
  - **Superpipeline**
  
- Aumentando a quantidade de instruções que executam em paralelo
  - Paralelismo real
  - Replicação de recursos de HW
  - **Superescalar e VLIW**

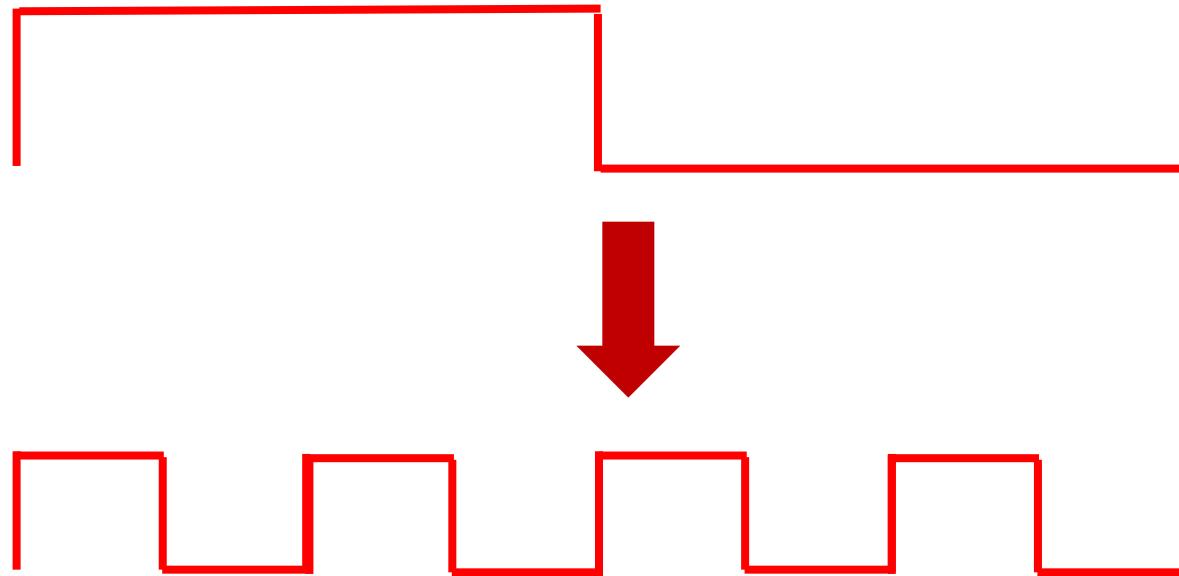
# Superpipeline

- Quebra estágios em subestágios (estágios menores)
  - Cada subestágio faz menos trabalho que estágio original
  - **Pipeline com maior profundidade**



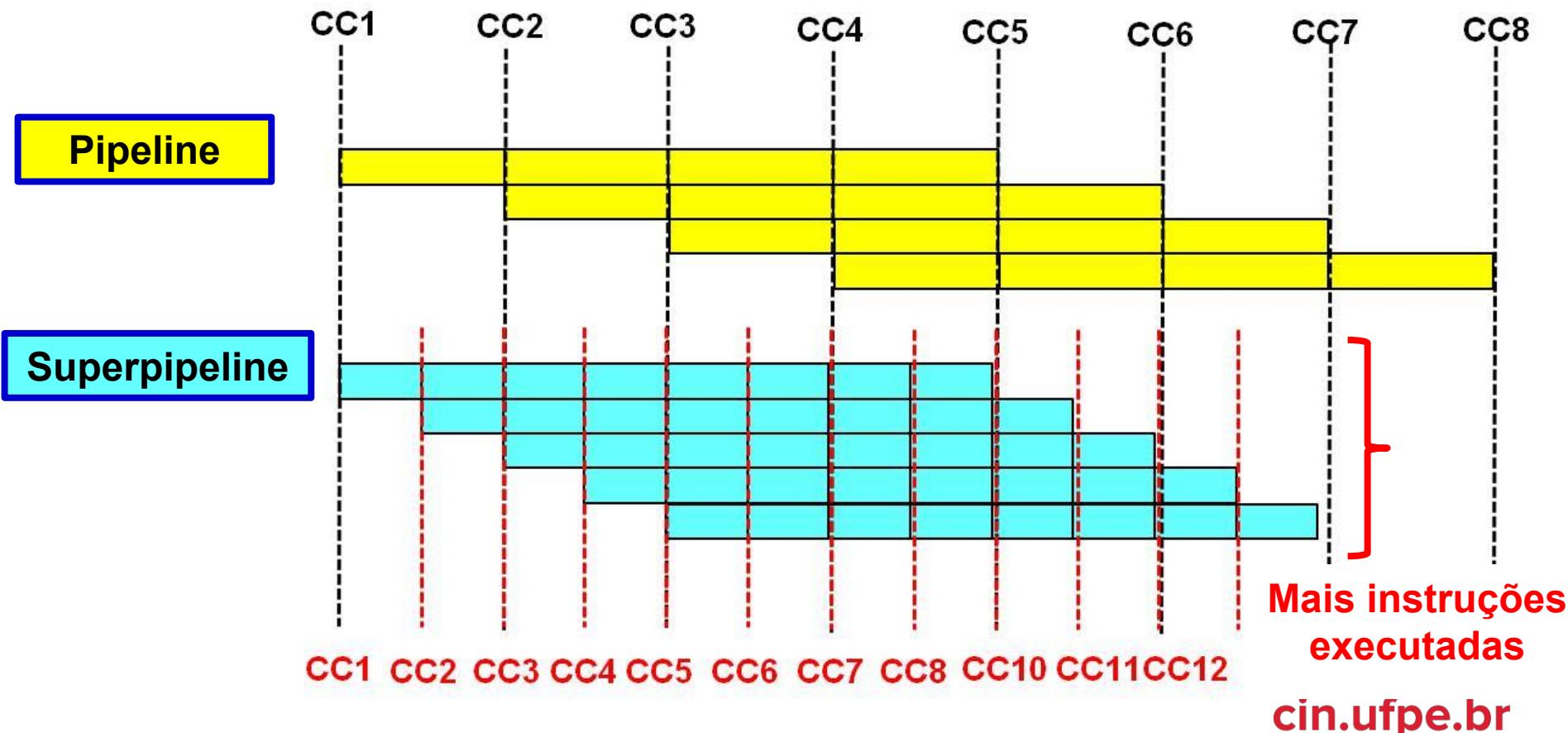
# Frequência do Clock em Superpipeline

- Estágios menores demandam menos tempo para serem executados
  - Período menor ↔ Frequência maior



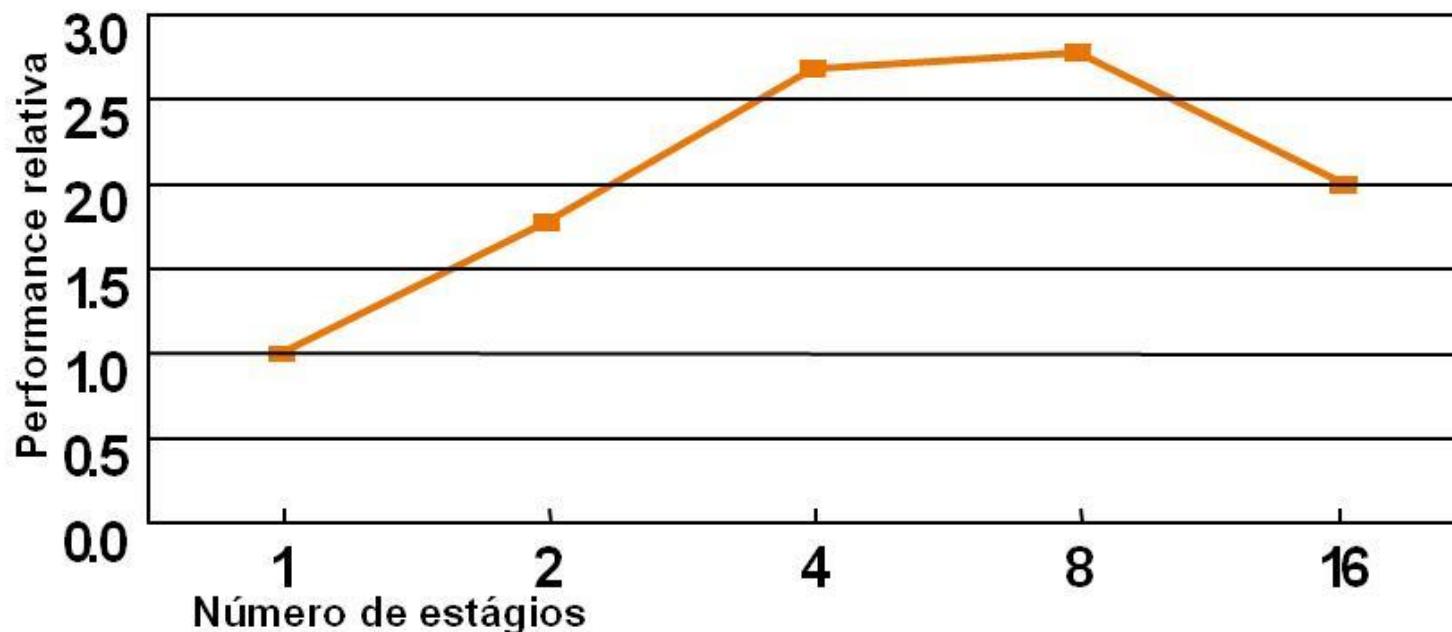
# Pipeline x Superpipeline

- Superpipeline: Maior throughput → Melhor desempenho
  - Maior números de estágios, frequência maior de clock
  - Mais instruções podem ser processadas simultaneamente



# Desempenho Relativo ao Número de Estágios

- Aumentar profundidade do pipeline nem sempre vai melhorar desempenho



# Mais Sobre Superpipeline

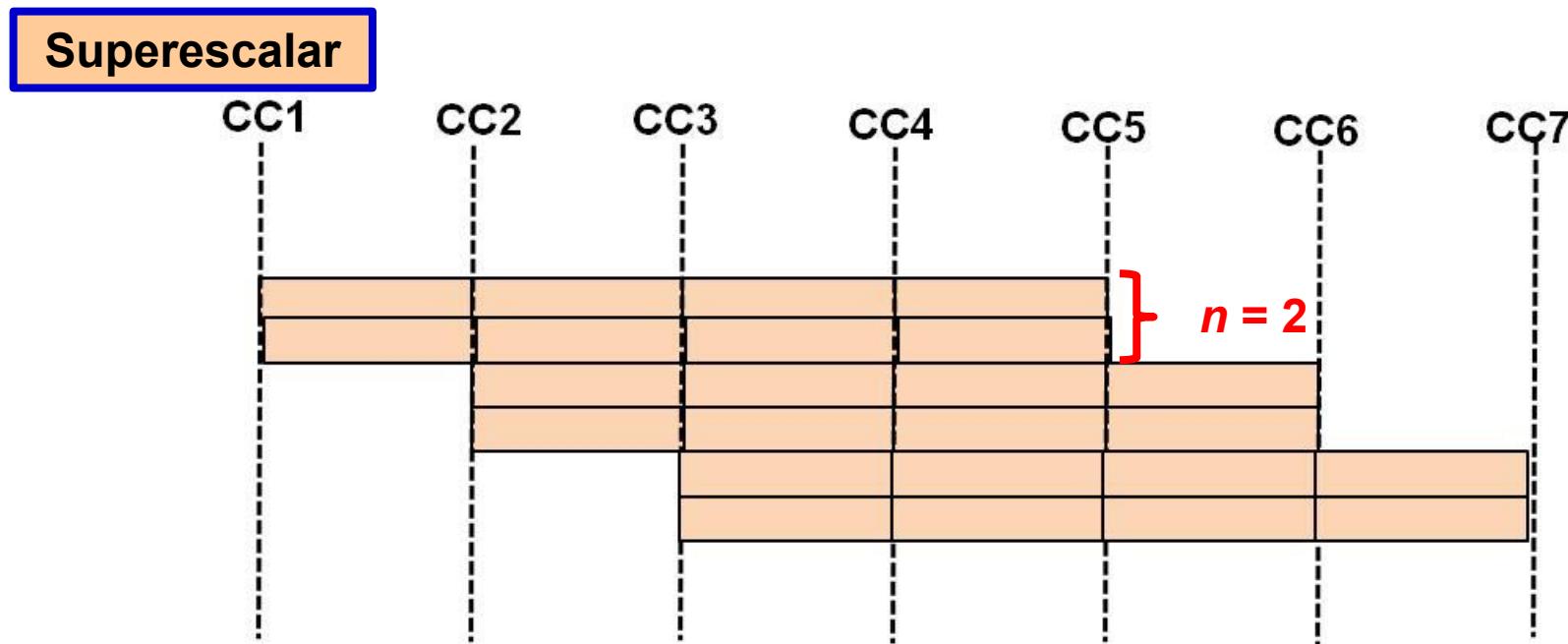
- Superpipeline visa diminuir tempo de execução de um programa
  - Dependências degradam desempenho
- Número de estágios excessivos penalizam desempenho
  - Conflito de dados
    - pipeline maior → mais dependências → mais retardos
  - Conflito de controle
    - pipeline maior → mais estágios para preencher
  - Tempo dos registradores do pipeline (entre estágios)
    - Limita tempo mínimo por estágio
- Maior custo de hardware

# Superescalar

- Processador com  $n$  pipelines de instrução replicados
  - $n$  dá o grau do pipeline superescalar
- Instruções diferentes podem iniciar a execução ao mesmo tempo
- Requer replicação de recursos de HW
- Aplicável a arquiteturas RISC e CISC
  - RISC : melhor uso efetivo
  - CISC : implementação mais difícil

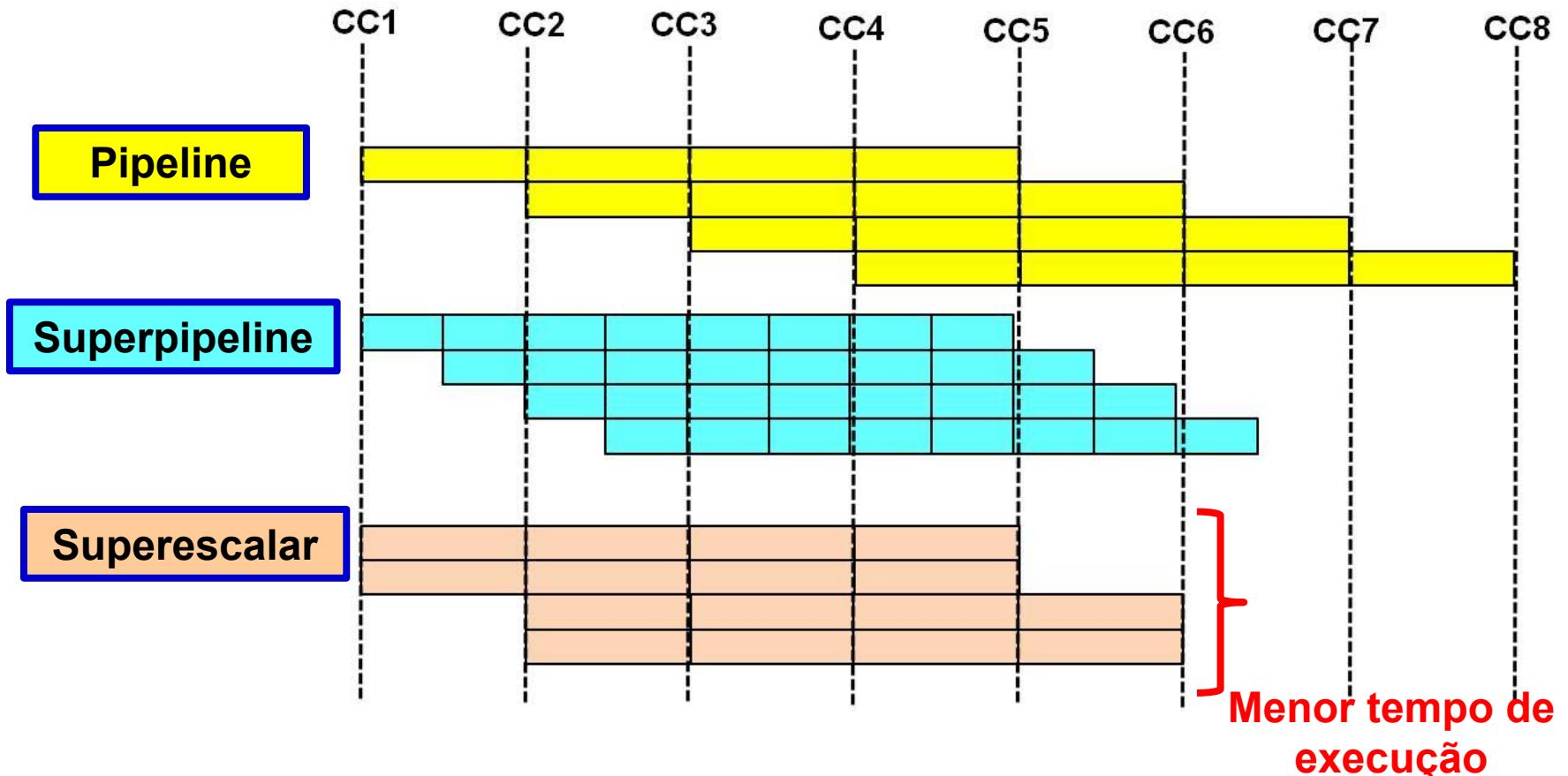
# Idéia Geral de Processadores Superescalares

- Grupos de instruções podem ser executados ao mesmo tempo
  - Número  $n$  de instruções por grupo define o grau do pipeline superescalar

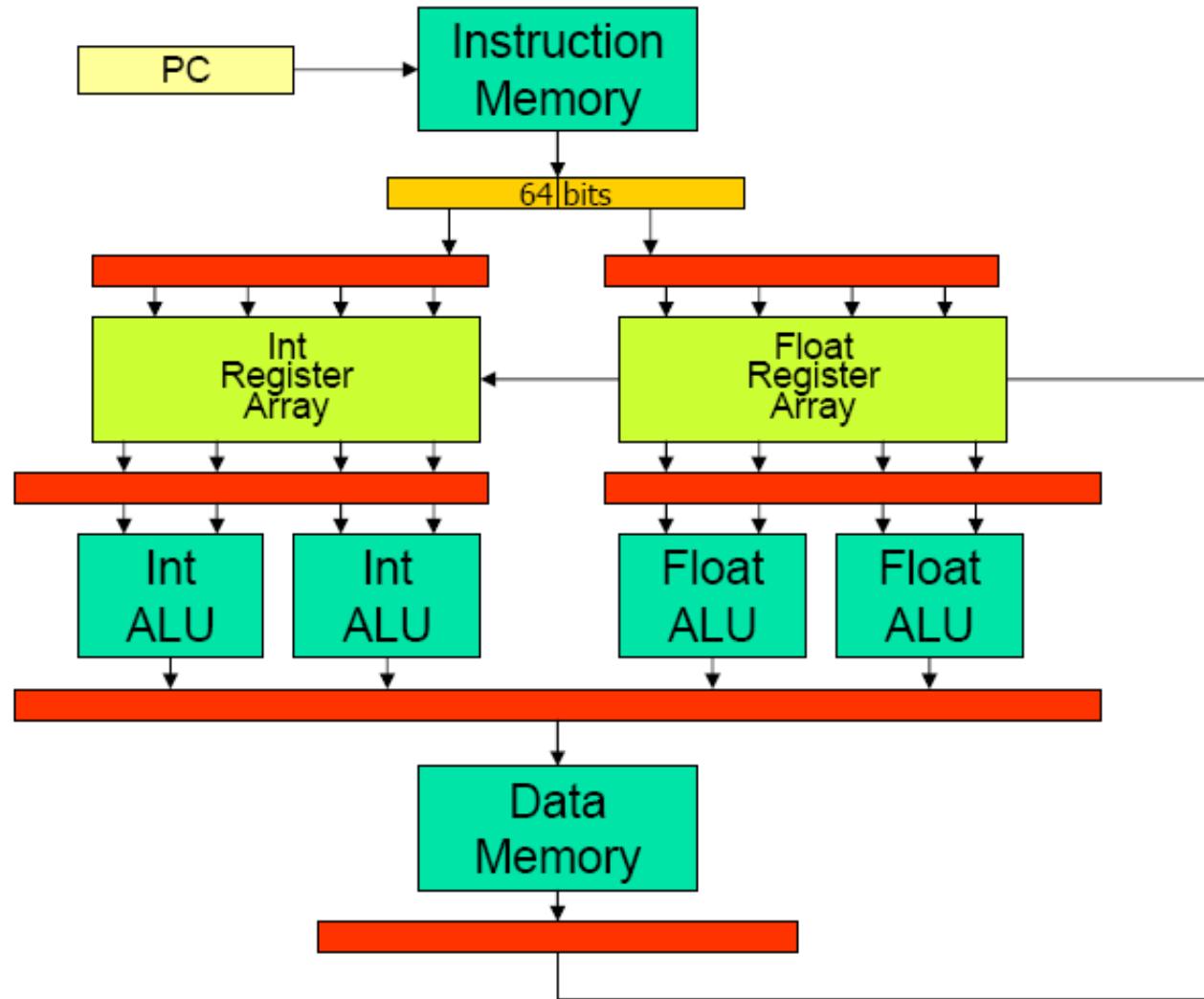


# Pipeline x Superpipeline x Superescalar

- Superescalar: Paralelismo real → Maior throughput → Melhor desempenho

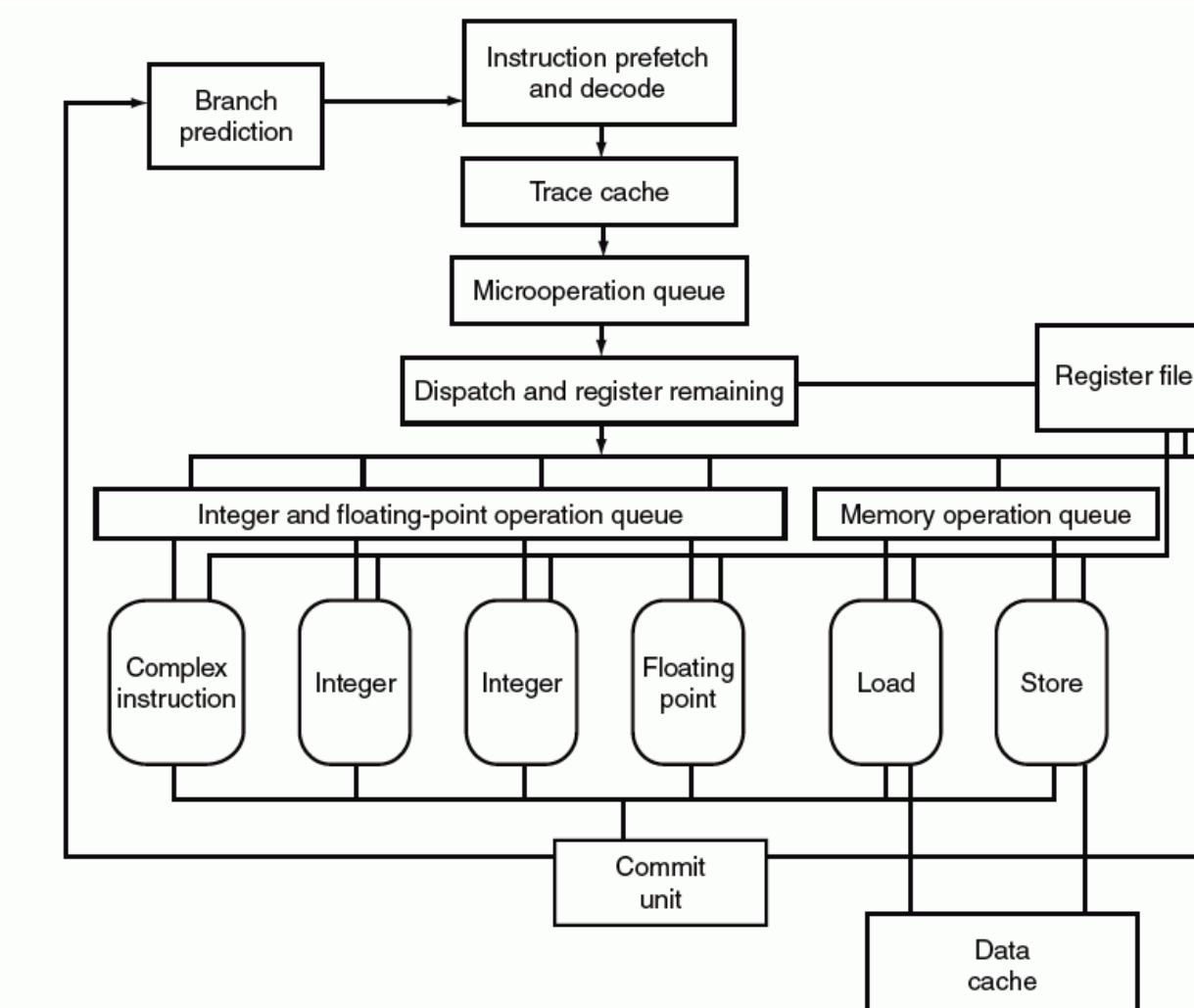


# Replicação de Recursos de HW

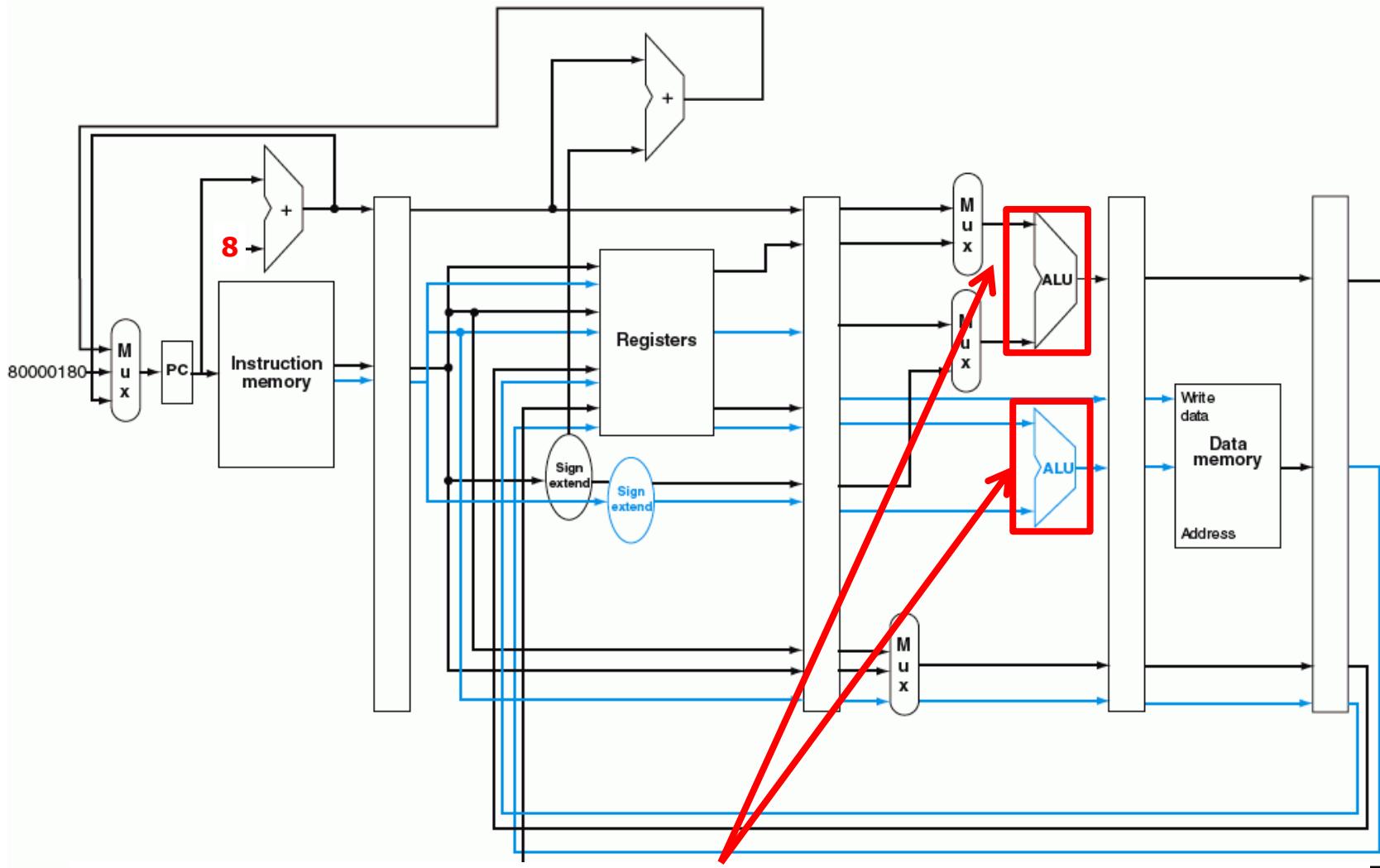


# Exemplo de Superescalar: Pentium 4

- Micro-arquitetura
  - Micro-instruções(operações) entram no pipeline superescalar

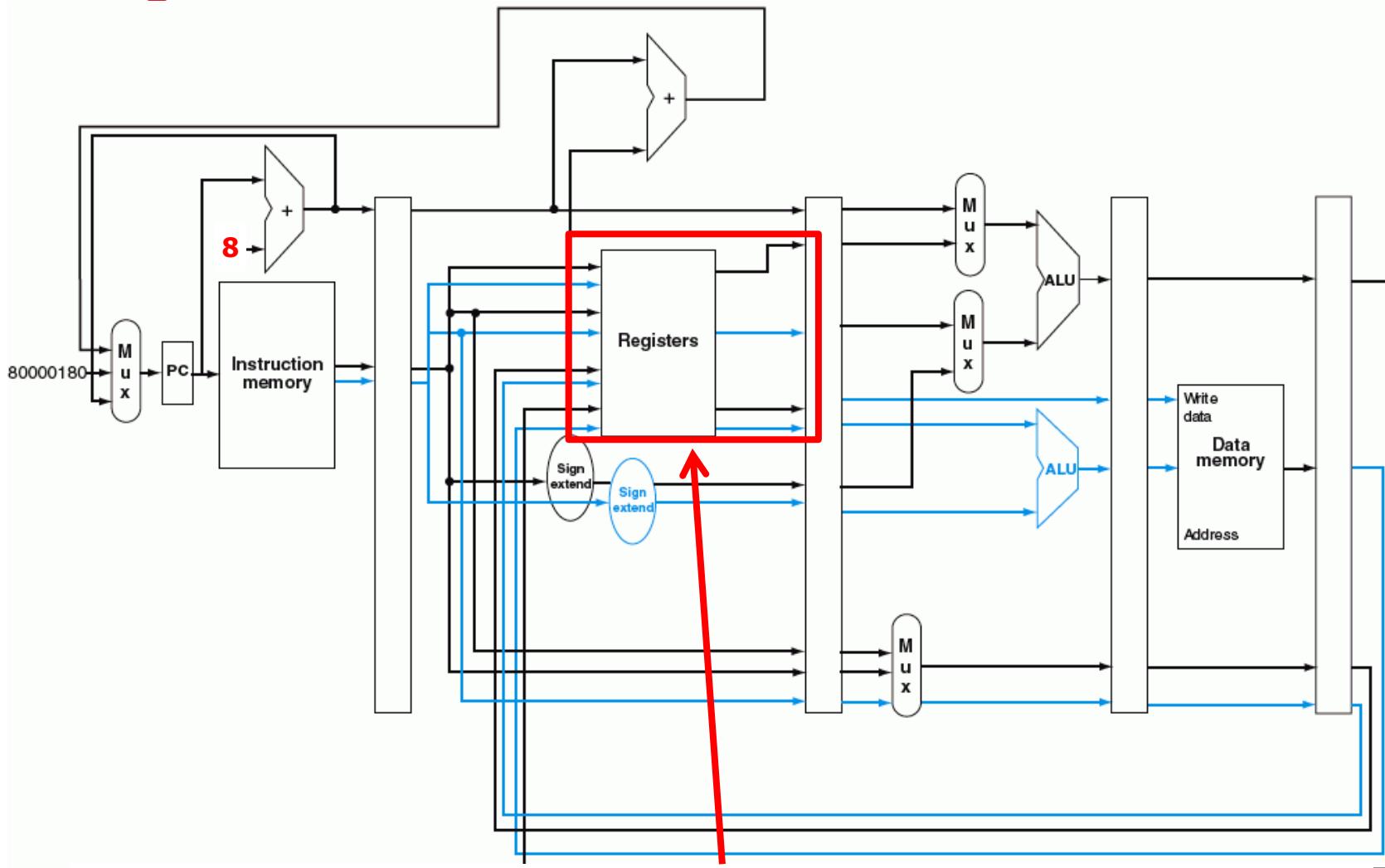


# Processador Superescalar com ISA do MIPS



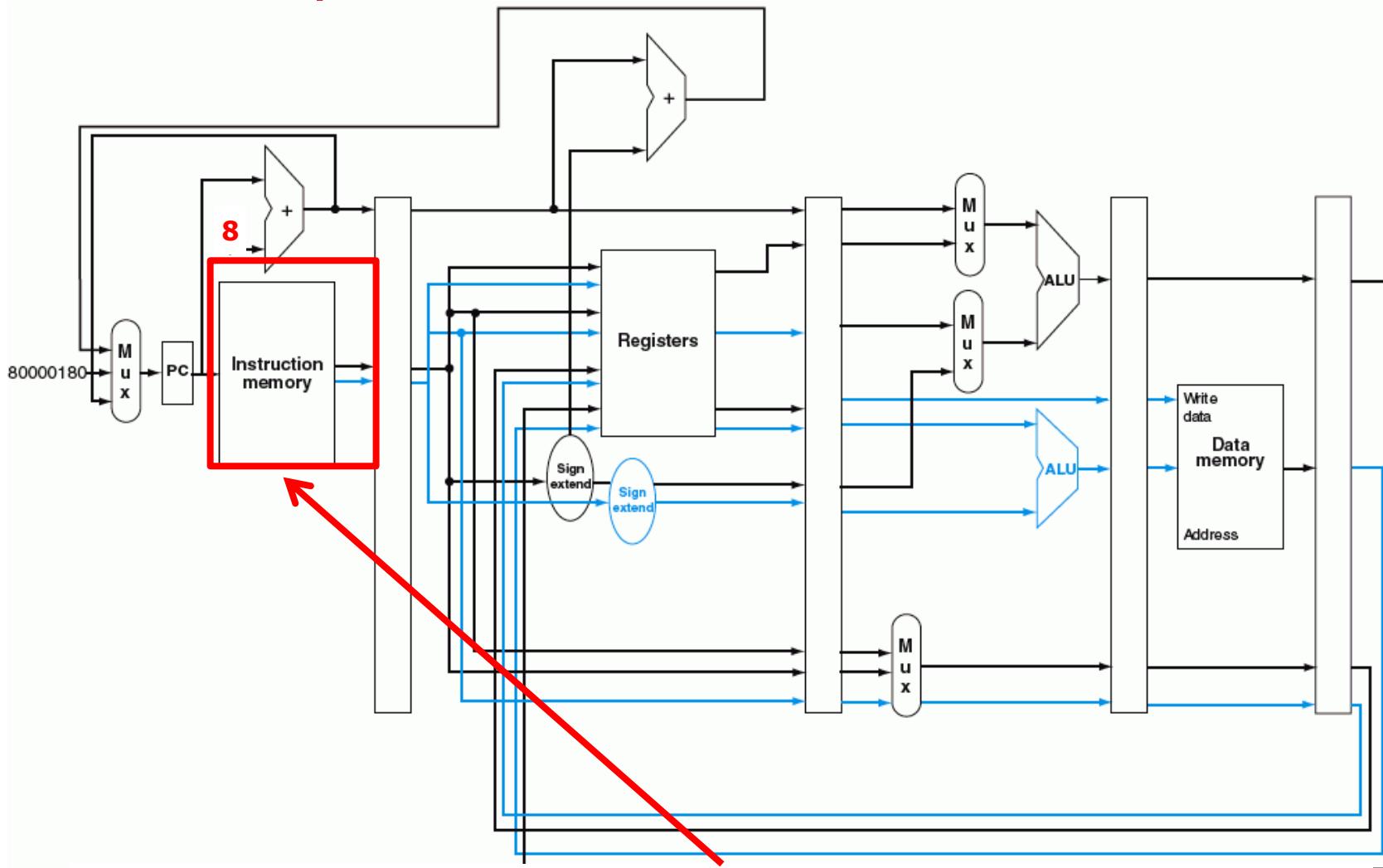
Replicação de ALU – Capacidade para realizar operações aritméticas/branches e de acesso a memória simultaneamente

# MIPS Superescalar – Banco de Registradores



Banco de registradores capazes de fazer 4 leituras e 2 escritas

# MIPS Superescalar – Memória de Instrução



Memória de Instrução permite leitura de 64 bits

# Operando com MIPS Superescalar

- Leitura da instrução de memória deve ser de 64 bits
  - Execução das instruções aos pares
  - Primeiros 32 bits, instrução aritmética/branch, e os outros 32 bits, instrução load/store

Instruction type	Pipe stages							
	IF	ID	EX	MEM	WB			
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

# Executando um Programa com o MIPS Superescalar

**Programa lê elementos do array na ordem inversa e soma valor contido em um registrador a cada elemento do array**

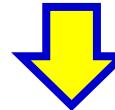
```
Loop: lw $t0,0($s1) #$t0 = elemento do array
      add $t0,$t0,$s2 # soma valor contido em $s2
      sw $t0,0($s1) # armazena resultado
      addi $s1,$s1,-4 # decrementa ponteiro
      bne $s1,$zero,Loop # desvia se $s1 != 0
```

# Mapeando o Programa para o MIPS Superescalar

```

Loop: lw $t0,0($s1) #$t0 = elemento do array
      add $t0,$t0,$s2 # soma valor contido em $s2
      sw $t0,0($s1) # armazena resultado
      addi $s1,$s1,-4 # decrementa ponteiro
      bne $s1,$zero,Loop # desvia se $s1 != 0
  
```

Dependência de dados obriga a rearrumação do código



	ALU ou Branch	Load ou Store	Clock
Loop	<b>nop</b>	<b>lw \$t0, 0(\$s1)</b>	1
	<b>addi \$s1, \$s1, -4</b>	<b>nop</b>	2
	<b>add \$t0, \$t0, \$s2</b>	<b>nop</b>	3
	<b>bne \$s1,\$zero,Loop</b>	<b>sw \$t0, 4(\$s1)</b>	4

# Desempenho do Programa com o MIPS Superescalar

	ALU ou Branch	Load ou Store	Clock
Loop	<b>nop</b>	<b>lw \$t0, 0(\$s1)</b>	1
	<b>addi \$s1, \$s1, -4</b>	<b>nop</b>	2
	<b>add \$t0, \$t0, \$s2</b>	<b>nop</b>	3
	<b>bne \$s1,\$zero,Loop</b>	<b>sw \$t0, 4(\$s1)</b>	4

- 4 ciclos por iteração (desprezando os ciclos para finalizar a execução da última instrução na última iteração)
- 4 ciclos para as 5 instruções  

$$CPI = 4/5 = 0,8$$
- Aproveita pouco o fato do processador ser superescalar  
 Muitos nops

# Otimizando o Desempenho do Programa

- Compiladores possuem técnicas avançadas de otimização
  - Loop Unrolling

Replica-se corpo do laço para reduzir número de iterações
- Supondo que índice de laço fosse múltiplo de 4, pode-se buscar 4 elementos do array a cada iteração
- Utilização mais eficiente do pipeline superescalar

Porém, código fica maior

# Otimizando o Desempenho do Programa

```
Loop: lw $t0,0($s1)
      add $t0,$t0,$s2
      sw $t0,0($s1)
      addi $s1,$s1,-4
      bne $s1,$zero,Loop
```

## Loop Unrolling



```
Loop: addi $s1,$s1,-16
      lw    $t0,0($s1)
      lw    $t1,12($s1) ←
      add $t0,$t0,$s2
      lw    $t2,8($s1) ←
      add $t1,$t1,$s2
      lw    $t3,4($s1) ←
      add $t2,$t2,$s2
      sw    $t0,16($s1)
      add $t3,$t3,$s2
      sw    $t1,12($s1)
      sw    $t2,8($s1)
      sw    $t3,4($s1)
      bne $s1,$zero,Loop
```

Carregando 4  
elementos por  
iteração

# Mapeando o Programa Otimizado para o MIPS Superescalar

	ALU ou Branch	Load ou Store	Clock
Loop	<b>addi \$s1, \$s1, -16</b>	<b>lw \$t0, 0(\$s1)</b>	1
	<b>nop</b>	<b>lw \$t1, 12(\$s1)</b>	2
	<b>add \$t0, \$t0, \$s2</b>	<b>lw \$t2, 8(\$s1)</b>	3
	<b>add \$t1, \$t1, \$s2</b>	<b>lw \$t3, 4(\$s1)</b>	4
	<b>add \$t2, \$t2, \$s2</b>	<b>sw \$t0, 16(\$s1)</b>	5
	<b>add \$t3, \$t3, \$s2</b>	<b>sw \$t1, 12(\$s1)</b>	6
	<b>nop</b>	<b>sw \$t2, 8(\$s1)</b>	7
	<b>bne \$s1,\$zero,Loop</b>	<b>sw \$t3, 4(\$s1)</b>	8

# Desempenho do Programa Otimizado

	ALU ou Branch	Load ou Store	Clock
Loop	<code>addi \$s1, \$s1, -16</code>	<code>lw \$t0, 0(\$s1)</code>	1
	<code>nop</code>	<code>lw \$t1, 12(\$s1)</code>	2
	<code>add \$t0, \$t0, \$s2</code>	<code>lw \$t2, 8(\$s1)</code>	3
	<code>add \$t1, \$t1, \$s2</code>	<code>lw \$t3, 4(\$s1)</code>	4
	<code>add \$t2, \$t2, \$s2</code>	<code>sw \$t0, 16(\$s1)</code>	5
	<code>add \$t3, \$t3, \$s2</code>	<code>sw \$t1, 12(\$s1)</code>	6
	<code>nop</code>	<code>sw \$t2, 8(\$s1)</code>	7
	<code>bne \$s1,\$zero,Loop</code>	<code>sw \$t3, 4(\$s1)</code>	8

- 8 ciclos para 4 iterações ou 2 ciclos por iteração
- 14 instruções

$$CPI = 8/14 \text{ ou } 0,57$$

# Escolhendo as Instruções que são Executadas em Paralelo

- Escolha de quais instruções serão executadas em paralelo pode ser feita por hardware ou software
- Hardware
  - Lógica especial deve ser inserida no processador
  - Decisão em tempo de execução (escolha dinâmica)
- Software
  - Compilador
    - Rearruma código e agrupa instruções
  - Decisão em tempo de compilação (escolha estática)

# Escolha pelo HW

- O termo **Superescalar** é mais associado a processadores que utilizam o hardware para fazer esta escolha
- CPU decide se 0, 1, 2, ... instruções serão executadas a cada ciclo
  - Escalonamento de instruções
  - Evitando conflitos
- Evita a necessidade de escalonamento de instruções por parte do compilador
  - Embora o compilador possa ajudar
  - Semântica do código é preservada pela CPU

# Escalonamento Dinâmico pelo HW

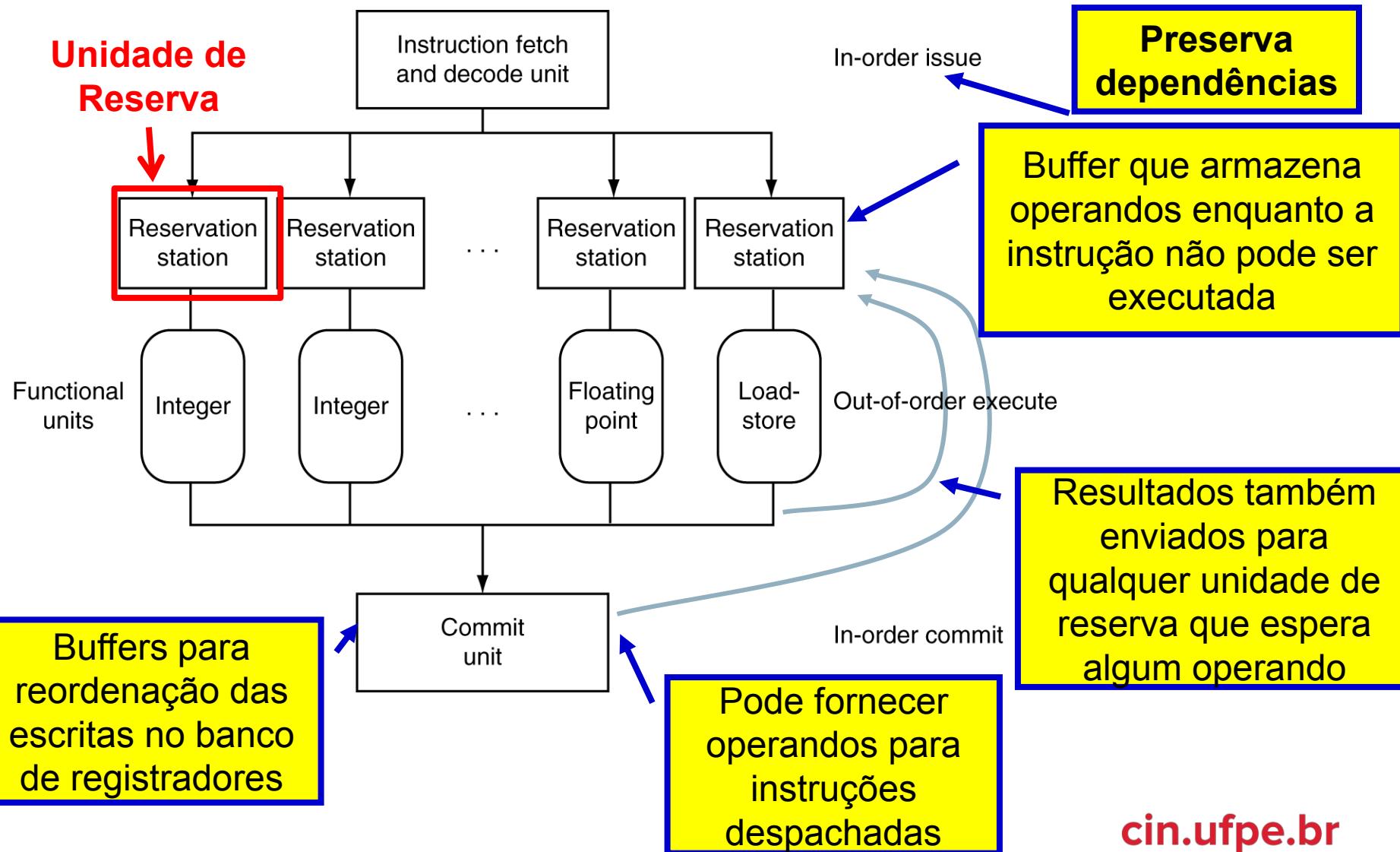
- Permite que CPU execute instruções fora de ordem para evitar retardos
  - Escrita nos bancos de registradores para completar instrução é feita em ordem
    - Encontram-se processadores que permitem escrita fora de ordem também
- Exemplo

```
lw      $t0, 20($s2)
add   $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```



**sub** pode começar enquanto **add** está esperando  
por **lw**

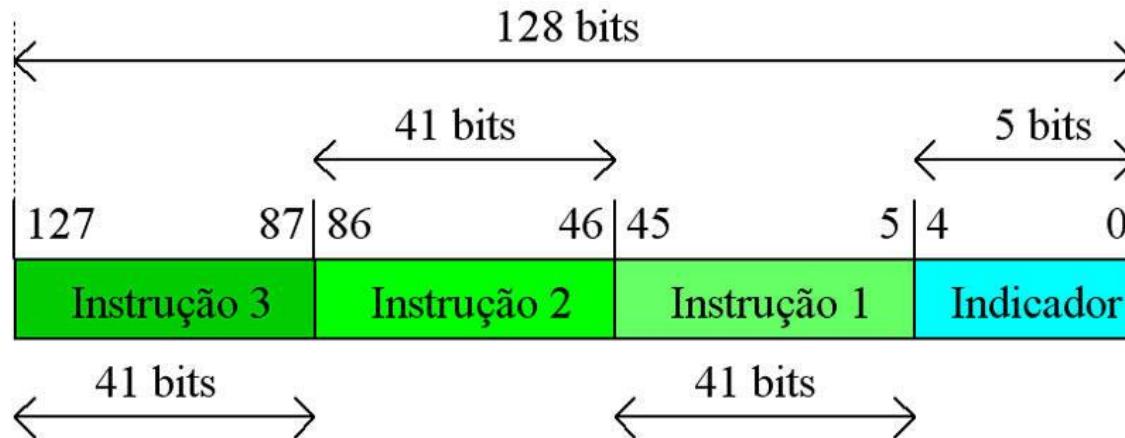
# Implementação de Escalonamento Dinâmico



# Escolha de Instruções pelo SW (Processadores VLIW)

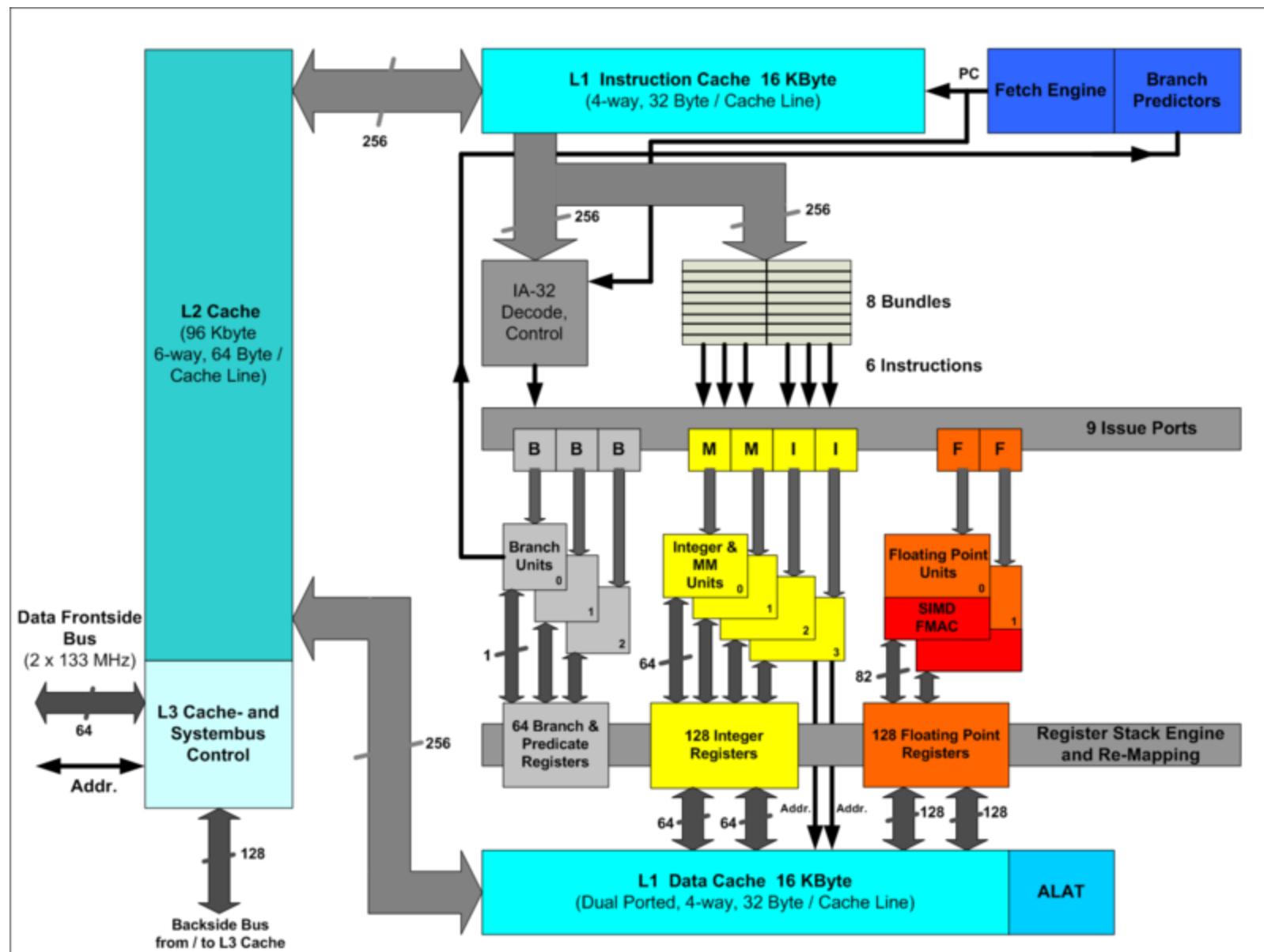
- O termo **VLIW (Very Long Instruction Word)** é associado a processadores parecidos com superescalares mas que dependem do software(compilador) para fazer esta escolha
- O compilador descobre as instruções que podem ser executadas em paralelo e as agrupa formando uma longa instrução (**Very Long Instruction Word**) que será despachada para a máquina
  - Escalonamento de instruções
  - Evitando conflitos

# Exemplo de VLIW: Intel Itanium



- Instruções são empacotadas
    - Cada pacote (*bundle*) contém 3 instruções e 128 bits
    - Cada instrução tem 41 bits
    - 5 bits são utilizados para informar quais unidades funcionais serão usadas pelas instruções

# Arquitetura do Intel Itanium



# Revendo Dependências de Dados

```
r3:= r0 op1 r5      (i1)
r4:= r3 op2 1       (i2)
r3:= r5 op3 1       (i3)
r7:= r3 op4 r4      (i4)
```

- Dependência Verdadeira (*Read-After-Write – RAW*)
  - i2 e i1, i4 e i3, i4 e i2
- Anti-dependência (*Write-After-Read - WAR*)
  - i3 não pode terminar antes de i2 iniciar
- Dependência de Saída (*Write-After-Write – WAW*)
  - i3 não pode terminar antes de i1

# Tipos de Dependências de Dados em Pipeline

```
r3:= r0 op1 r5      (i1)
r4:= r3 op2 1        (i2)
r3:= r5 op3 1        (i3)
r7:= r3 op4 r4       (i4)
```

- Único tipo de dependência que causa problemas em um pipeline é a Dependência Verdadeira (*RAW*)
- Anti-dependência (*WAR*) não causa nenhum problema em um pipeline , pois instrução que escreve sempre o faz em um estágio posterior ao da leitura do registrador pela instrução anterior
  - Idem para WAW

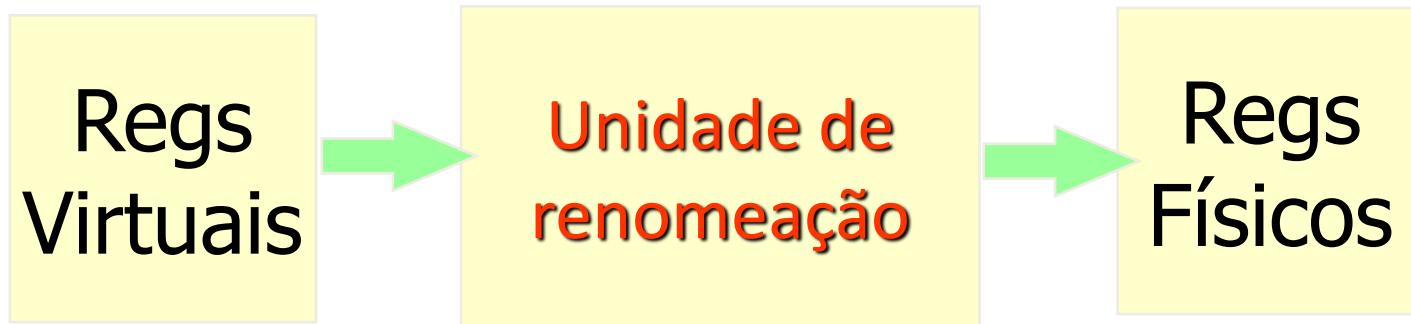
# Resolvendo Dependências WAR e WAW

- Mesmo não causando nenhum problema para o pipeline, estas dependências podem “iludir” o compilador ou HW, devendo portanto ser eliminadas
- Solução: Renomeação de registradores

<b>WAW</b> <b>WAR</b>	$r3 := r3 \text{ op}_1 r5$ $r3b := r3a \text{ op}_1 r5$ $r4 := r3 \text{ op}_2 1$ $r4 := r3b \text{ op}_2 1$ $r3 := r5 \text{ op}_3 1 \Rightarrow r3c := r5 \text{ op}_3 1$ $r7 := r3 \text{ op}_4 r4$ $r7 := r3c \text{ op}_4 r4$
--------------------------	---

# Renomeação de Registradores

- Pode ser feita tanto pelo compilador (SW) ou pelo HW



# Renomeação de Registradores por SW

```
Loop: lw $t0,0($s1)
      addu $t0,$t0,$s2
      sw $t0,0($s1)
      addi $s1,$s1,-4
      bne $s1,$zero,Loop
```

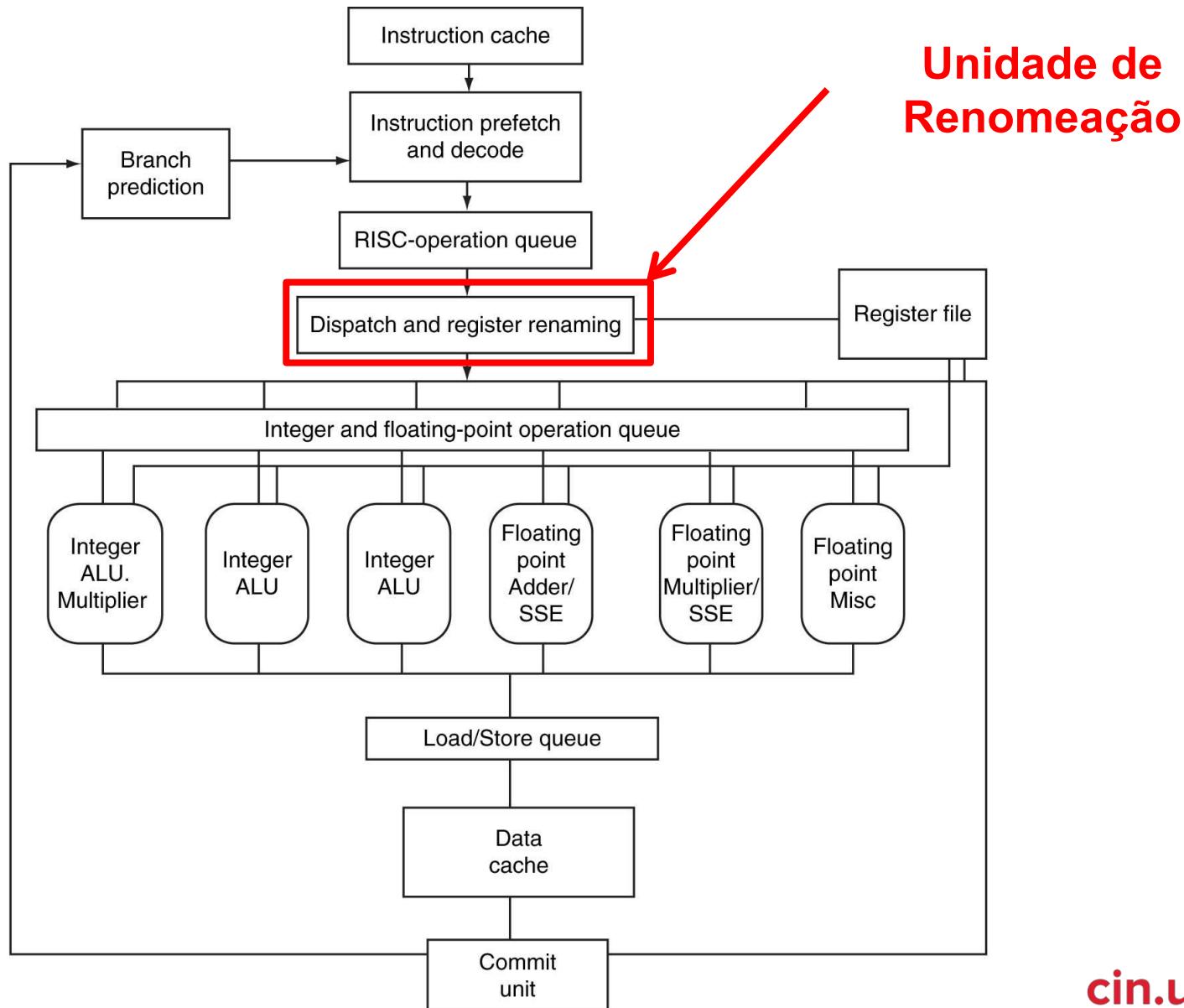
## Loop Unrolling



```
Loop: addi $s1,$s1,-16
      lw    $t0,0($s1)
      lw    $t1,12($s1) ← Red box
      addu $t0,$t0,$s2
      lw    $t2,8($s1)   ← Red box
      addu $t1,$t1,$s2
      lw    $t3,4($s1)   ← Red box
      addu $t2,$t2,$s2
      sw    $t0,16($s1)
      addu $t3,$t3,$s2
      sw    $t1,12($s1)
      sw    $t2,8($s1)
      sw    $t3,4($s1)
      bne   $s1,$zero,Loop
```

**Renomeação para  
carregar 4  
elementos  
independentes**

# Renomeação por HW: AMD Opteron X4



# Analizando o Superpipeline

- Superpipeline visa diminuir tempo de execução de um programa
  - Aumento da frequência do clock
  - Dependências degradam desempenho
- Número de estágios excessivos podem penalizar desempenho
  - Conflito de dados, controle
  - Overhead de passagem de estágios
- Maior custo de hardware
  - Mais registradores, mais lógica

# Analisando o Superescalar...

- Visa reduzir o número médio de ciclos por instrução (CPI)
  - CPI < 1
  - Instruções podem iniciar ao mesmo tempo
- HW encarregado de escalonar instruções e detectar conflitos
  - Permite execução fora de ordem de instruções
  - Considera aspectos dinâmicos de execução
  - Lógica em HW mais complexa
- Maior custo de HW
  - Unidades de Reserva, lógica de escalonamento

# Analizando o VLIW

- Simplifica HW transferindo para o compilador a lógica de detecção do paralelismo
  - Circuitos mais simples
  - Clock mais rápido
- Aspectos dinâmicos não são analisados pelo compilador, o que pode causar retardos inesperados
  - Exemplo: Se dado não estiver na cache
- Mudanças no pipeline de um VLIW requer mudanças no compilador

# Infraestrutura de Hardware

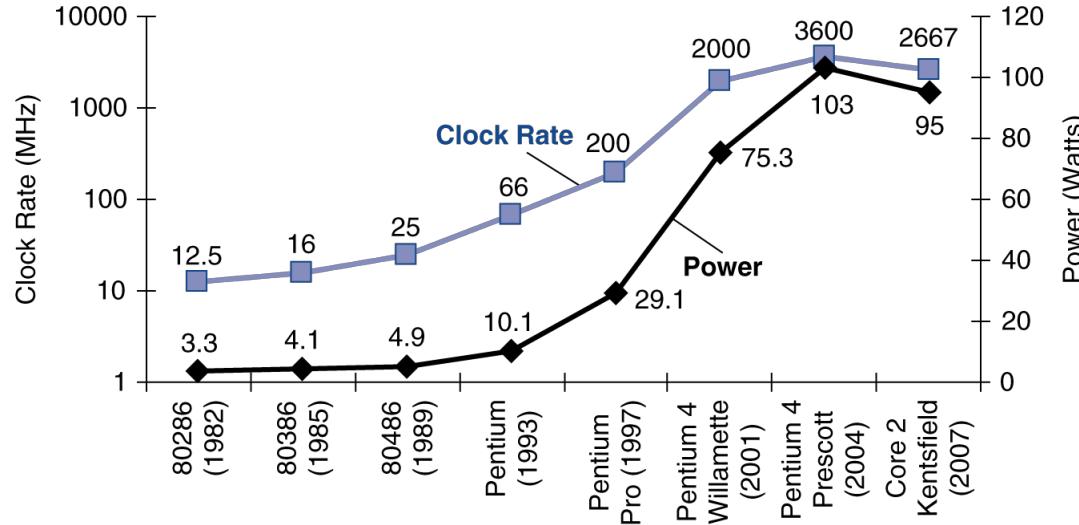
Processamento Paralelo: Desafios, Multicore  
e Classificação de Arquiteturas

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Obstáculo para Desempenho: Potência



- Tecnologia CMOS de circuitos integrados

$$\text{Potencia} = \text{Capacitancia} \times \text{Voltagem}^2 \times \text{Frequencia}$$

$\times 30$

No intervalo de  
25 anos

$5V \rightarrow 1V$

$\times 1000$

# Reduzindo Potência e Energia

- Quanto maior a frequência, maior a potência dissipada
  - Sistema de esfriamento do processador é impraticável para frequências muito altas
- Devem ser mais eficientes em termos de energia
  - Para dispositivos móveis que dependem de bateria
- **Power wall**
  - Não se consegue reduzir ainda mais a voltagem
  - Sistema de esfriamento não consegue eliminar tanto calor

**Como aumentar então desempenho de uma CPU?**

# Multiprocessadores

- Múltiplos processadores menores, mas eficientes trabalhando em conjunto
- Melhoria no desempenho:
  - Paralelismo ao nível de processo
    - Processos independentes rodando simultaneamente
  - Paralelismo ao nível de processamento de programa
    - Único programa rodando em múltiplos processadores
- Outros Benefícios:
  - Escalabilidade, Disponibilidade, Eficiência no Consumo de Energia

# Interesse em Multiprocessadores

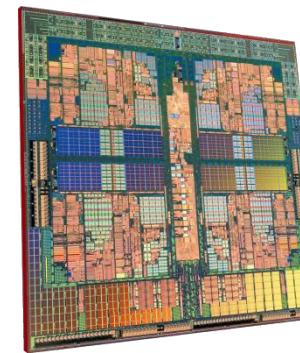
- Uma crescente utilização de servidores
- Um crescimento em aplicações data-intensive
- Um melhor entendimento de como usar multiprocessadores para explorar thread-level paralelismo
- Investimento em replicação é mais atrativo que investimento em um projeto exclusivo.

# Melhorando Desempenho com Multiprocessamento

- Alguns problemas que requerem alto desempenho podem ser solucionados com o uso de clusters
  - Servidores independentes conectados por rede que funcionam como um grande e único multiprocessador
    - Search engines, servidores web, servidores de email, banco de dados, etc
- Desafio é desenvolver programas paralelos que tenham alto desempenho em multiprocessadores a medida que a quantidade de processadores aumenta

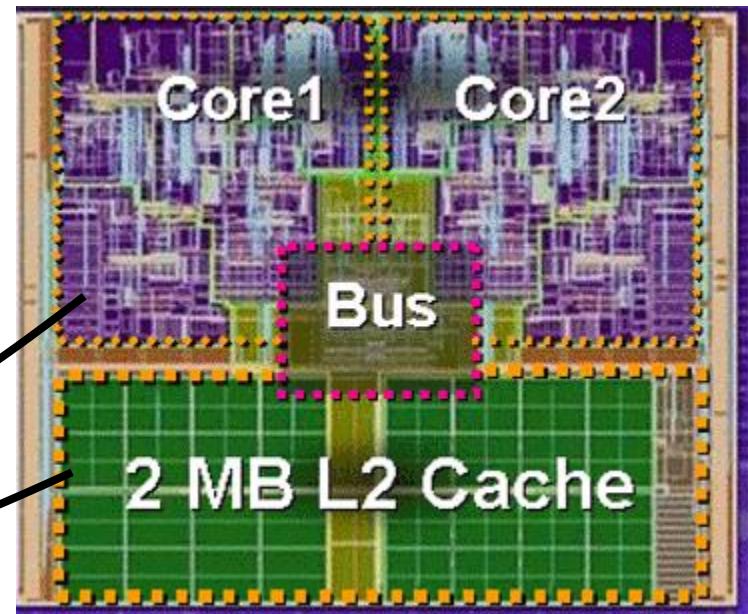
# Multicores

- Microprocessadores multicores
  - Mais de um processador por chip
  - Cada processador é chamado de core
  - Capacidade de integração de transistor permitiu maior quantidade de cores por chip
  - Menor consumo de energia por core
- Cada core pode rodar com frequências menores
  - Menor potência
  - Desempenho alcançado com aumento do throughput



# Exemplo: Intel Core 2 Duo

- Cores homogêneos
  - Superscalares
    - (escalonamento dinâmico, especulação, superescalar)
- Interconexão baseada em barramento
- Cada “core” tem cache local (L1)
- Memória compartilhada
  - (cache L2) no chip



Source: Intel Corp.

# Classificação de Arquiteturas Segundo a Taxonomia de Flynn

M.J. Flynn, "Very High-Speed Computers",  
*Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

		Data Streams	
		Single	Multiple
Instruction Streams	Single	<b>SISD:</b> Intel Pentium 4	<b>SIMD:</b> instruções SSE do x86
	Multiple	<b>MISD:</b> Nenhum exemplo	<b>MIMD:</b> Clusters, Intel i7

- Flynn classificou de acordo com streams de dado e controle em 1966
- SIMD ⇒ Paralelismo ao nível de dados
- MIMD ⇒ Paralelismo ao nível de threads
- MIMD mais popular
  - Flexibilidade: N programas multithreaded

# Processamento Paralelo - Hardware e Software

		Software	
		Sequencial	Concorrente
Hardware	Serial	Multiplicação de matrizes em Java no Intel Pentium 4	Windows 10 no Intel Pentium 4
	Paralelo	Multiplicação de matrizes em Java no AMD Ryzen 5	Windows 10 no AMD Ryzen 5

- Software sequencial/concorrente pode executar em hardware serial/paralelo
  - Desafio: fazer uso efetivo de hardware paralelo

# Programação Paralela

- Desenvolver software para executar em HW paralelo
- Necessidade de melhoria significativa de desempenho
  - Senão é melhor utilizar processador com único core rápido, pois é mais fácil de escrever o código
- Dificuldades
  - Programar visando desempenho de modo que seja mais transparente para o programador
  - Particionamento de tarefas (Balanceamento de carga)
  - Coordenação
  - Overhead de comunicação

# Analisando Desempenho com Multicores

- Podemos analisar a melhoria provocado pelo aumento de cores
- Lei de Amdahl
  - $T_m$  = Tempo de Execução com melhoria
  - $T_a$  = Tempo de execução afetado pela melhoria
  - $Q_c$  = quantidade de cores
  - $T_{sm}$  = Tempo de Execução não afetado pela melhoria

$$T_m = T_a/Q_c + T_{sm}$$

# Exemplo: Ganho de Desempenho com Multicores

## Problema:

Suponha que queiramos fazer dois tipos de soma: somar 10 variáveis escalares em sequencia e somar duas matrizes  $10 \times 10$ . Qual o ganho com 10 cores e com 100 cores? Considere que uma soma leva um tempo t.

## Solução:

Se só houvesse um core o tempo total seria  $100t$  (soma de duas matrizes  $10 \times 10$ ) +  $10t$  (soma de 10 escalares) =  $110t$

Soma de 10 escalares deve ser sequencial, portanto leva  $10t$  ou seja  $T_{sm} = 10t$

Soma de 2 matrizes pode ser feita em paralelo.

Para 10 cores:

$$T_m = T_a/Q_c + T_{sm} = 100t/10 + 10t = 20t$$

$$\text{Ganho} = 110t/20t = 5,5$$

# Exemplo: Ganho de Desempenho com Multicores

## Problema:

Suponha que queiramos fazer dois tipos de soma: somar 10 variáveis escalares em sequencia e somar duas matrizes  $10 \times 10$ . Qual o ganho com 10 cores e com 100 cores? Considere que uma soma leva um tempo t.

## Solução:

Se só houvesse um core o tempo total seria  $100t$  (soma de duas matrizes  $10 \times 10$ ) +  $10t$  (soma de 10 escalares) =  $110t$

Soma de 10 escalares deve ser sequencial, portanto leva  $10t$  ou seja  $T_{sm} = 10t$

Soma de 2 matrizes pode ser feita em paralelo.

Para 100 cores:

$$T_m = T_a/Q_c + T_{sm} = 100t/100 + 10t = 11t$$

$$\text{Ganho} = 110t/11t = 10$$

# Analisando Resultados

- Com 10 cores, esperava-se ter uma execução 10 vezes mais rápida, ou seja  $110t/10 = 11t$ , porém só foi possível  $20t$ 
  - Apenas **55%** ( $11/20$ ) do ganho esperado foi atingido
- Com 100 cores, esperava-se execução 100 vezes mais rápida ou seja  $110t/100 = 1,1t$ , porém só foi possível  $11t$ 
  - Apenas **10%** ( $1,1/11$ ) do ganho esperado foi atingido

# Exemplo: Aumentando o Problema

## Problema:

Suponha que queiramos fazer dois tipos de soma: somar 10 variáveis escalares em sequencia e somar duas matrizes  $100 \times 100$ . Qual o ganho com 10 cores e com 100 cores? Considere que uma soma leva um tempo t.

## Solução:

Se só houvesse um core o tempo total seria  $10000t$  (soma de duas matrizes  $100 \times 100$ ) +  $10t$  (soma de 10 escalares) =  $10010t$

Soma de 10 escalares deve ser sequencial, portanto leva  $10t$  ou seja  $T_{sm} = 10t$

Soma de 2 matrizes pode ser feita em paralelo.

Para 10 cores:

$$T_m = T_a/Q_c + T_{sm} = 10000t/10 + 10t = 1010t$$

$$\text{Ganho} = 10010t/1010t = 9,9$$

# Exemplo: Aumentando o Problema

## Problema:

Suponha que queiramos fazer dois tipos de soma: somar 10 variáveis escalares em sequencia e somar duas matrizes  $100 \times 100$ . Qual o ganho com 10 cores e com 100 cores? Considere que uma soma leva um tempo t.

## Solução:

Se só houvesse um core o tempo total seria  $10000t$  (soma de duas matrizes  $100 \times 1000$ ) +  $10t$  (soma de 10 escalares) =  $10010t$

Soma de 10 escalares deve ser sequencial, portanto leva  $10t$  ou seja  $T_{sm} = 10t$

Soma de 2 matrizes pode ser feita em paralelo.

Para 100 cores:

$$T_m = T_a/Q_c + T_{sm} = 10000t/100 + 10t = 110t$$

$$\text{Ganho} = 10010t/110t = 91$$

# Analizando Resultados com o Aumento do Problema

- Com 10 cores, esperava-se ter uma execução 10 vezes mais rápida, ou seja  $10010t/10 = 1001t$ , foi possível 1010t
  - **99%** ( $1001/1010$ ) do ganho esperado foi atingido
- Com 100 cores, esperava-se execução 100 vezes mais rápida ou seja  $10010t/100 = 100,1t$ , foi possível 110t
  - **91%** ( $100,1/110$ ) do ganho esperado foi atingido
- Aumento do tamanho do problema resultou em um ganho de desempenho quase proporcional à quantidade de cores
  - Balanceamento de carga entre cores também é importante

**Difícil é melhorar desempenho com tamanho de problema fixo!**

# Escalabilidade

- Aumentar o speedup em multiprocessadores mantendo o tamanho do problema fixo é difícil
  - **Escalabilidade forte** – quando speedup pode ser alcançado sem aumentar o tamanho do problema
  - **Escalabilidade fraca** – quando speedup pode ser alcançado apenas aumentando o tamanho do problema proporcionalmente ao aumento da quantidade de processadores
- Balanceamento de carga é um importante componente
  - Um único processador com o dobro da carga em relação aos outros processadores diminui em quase metade o speedup

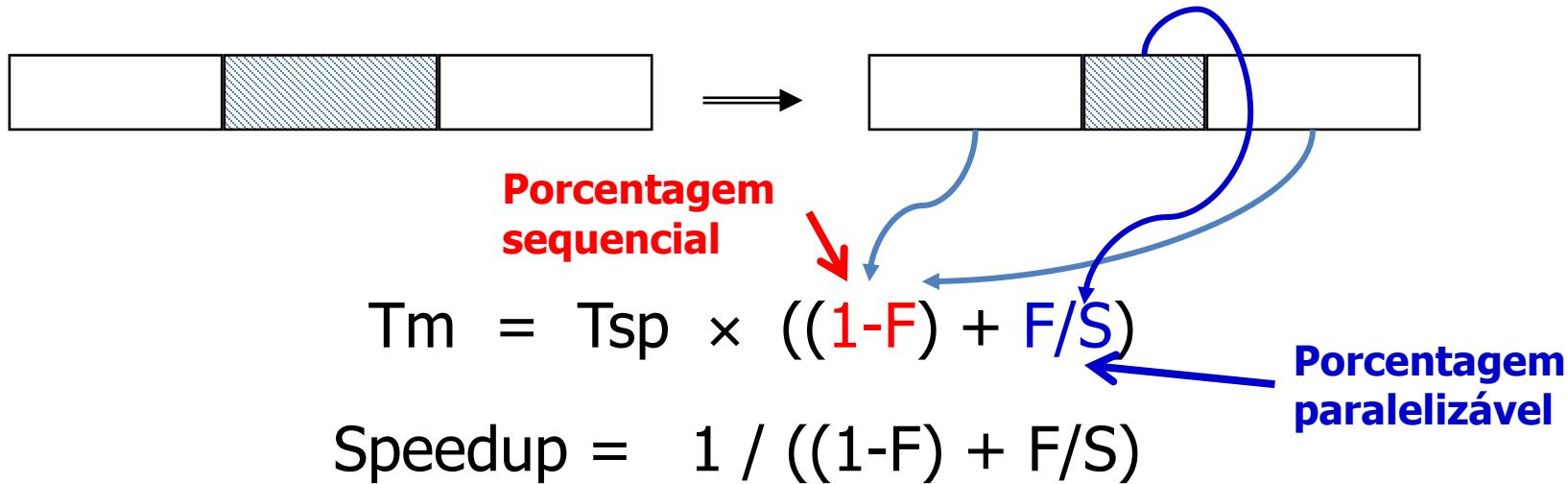
# Calculando Speedup com Lei de Amdahl

- Speedup

$$\text{Speedup} = \frac{T_{sp}}{T_m}$$

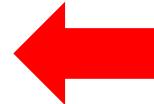
**T<sub>sp</sub>** – tempo total se não houvesse paralelização

- Suponha que a paralelização acelere uma fração  $F$  ( $F < 1$ ) da tarefa por um fator  $S$  ( $S > 1$ ) e o resto da tarefa permanece inalterado



# Desafio do Processamento Paralelo

- Suponha que se queira um speedup de 80X para 100 processadores. Qual a fração do programa que deve ser sequencial?
  - 10%
  - 5%
  - 1%
  - <1%



0,25%

# Resposta Usando Lei de Amdahl

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times ((1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

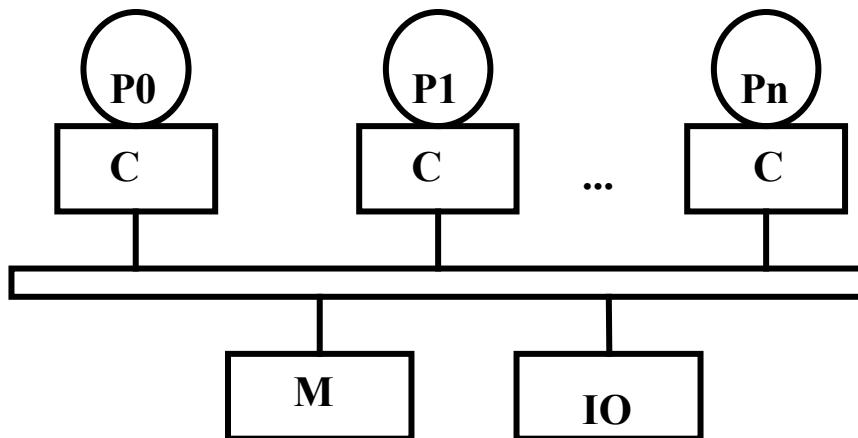
# Classificação de Multiprocessadores

- Arquitetura Paralela = Arquitetura do Computador + Arquitetura da Comunicação
- Classificação por memória (Arquitetura de Computador):
  - Multiprocessador de Memória Centralizada (Symmetric)
    - Típico para sistemas pequenos → demanda de largura de banda de memória e rede de comunicação.
  - Multiprocessador de Memória Fisicamente Distribuída
    - Escala melhor → demanda de largura de banda para rede de comunicação

# Classificação por Memória

## ■ Multiprocessadores de Memória Centralizada

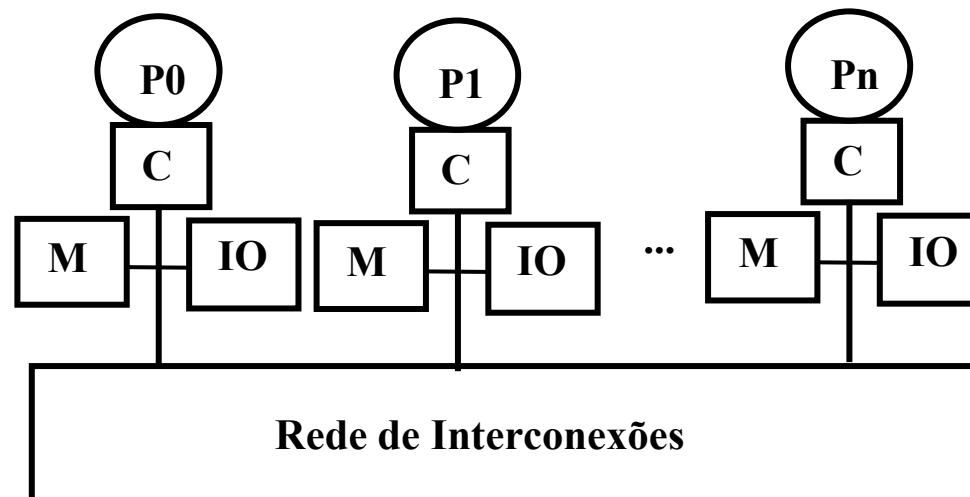
- Poucos processadores ( poucas dezenas chips ou cores) em 2006
- Memória única e centralizada



**Memória Centralizada**

# Classificação por Memória

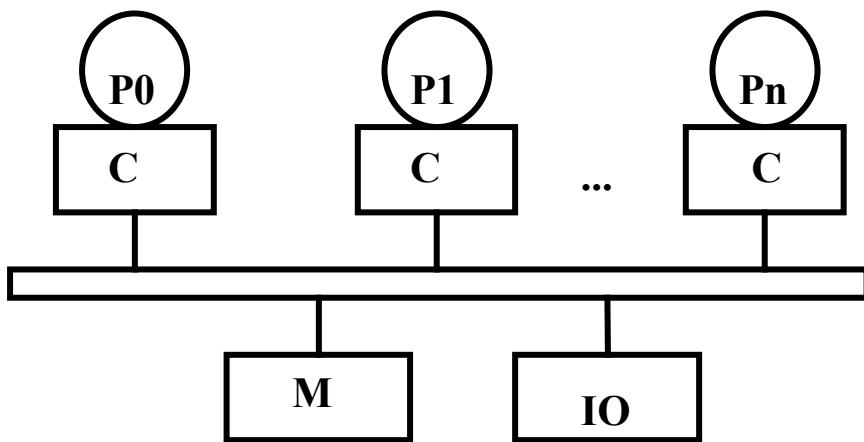
- Multiprocessadores de Memória Fisicamente Distribuída
  - Maior número de processadores (centenas de chips ou cores)
  - Memória distribuída entre processadores



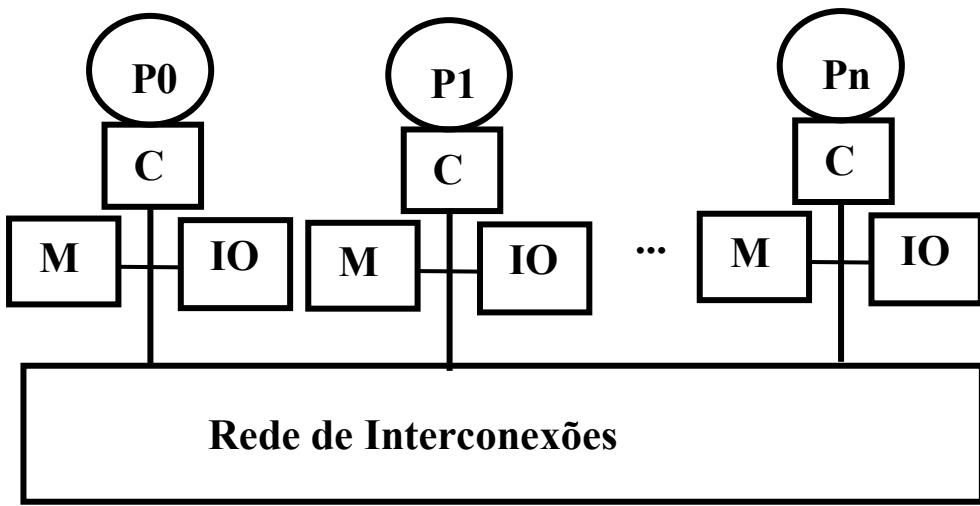
**Memória Distribuída**

# Centralizada vs. Distribuída

Escalabilidade maior



Memória Centralizada



Memória Distribuída

# Infraestrutura de Hardware

Aula de Revisão da 1<sup>a</sup> prova

Prof. Adriano Sarmento

# Exercício 1

**(ENADE 2014)** A seguinte sequência de instruções será executada por um processador em pipeline de 5 estágios (IF, ID, EX, MEM, WB). A sequência, no entanto, apresenta conflito de dados.

```
and R5, R4,R3
or  R6, R4, R2
add R1, R2, R2
mul R3, R2, R1
sub R1, R1, R4
```

O pipeline foi implementado sem HW adicional para a resolução de conflitos de dados, mas os valores dos registradores podem ser escritos na primeira metade do ciclo e lidos na segunda metade. Sabendo-se que o primeiro operando das instruções é o registrador destino, avalie as afirmações a seguir:

- I. A troca de posição entre as instruções **or** e **add** soluciona o conflito
- II. A troca de posição entre as instruções **add** e **and** soluciona o conflito
- III. A inserção de uma operação **nop** entre **add** e **mul** soluciona o conflito

# Resposta do Exercício 1

(ENADE 2014)

and R5, R4,R3  
or R6, R4, R2  
add R1, R2, R2  
mul R3, R2, R1  
sub R1, R1, R4

- I. A troca de posição entre as instruções **or** e **add** soluciona o conflito
- II. A troca de posição entre as instruções **add** e **and** soluciona o conflito
- III. A inserção de uma operação **nop** entre **add** e **mul** soluciona o conflito

É correto o que se afirma em:

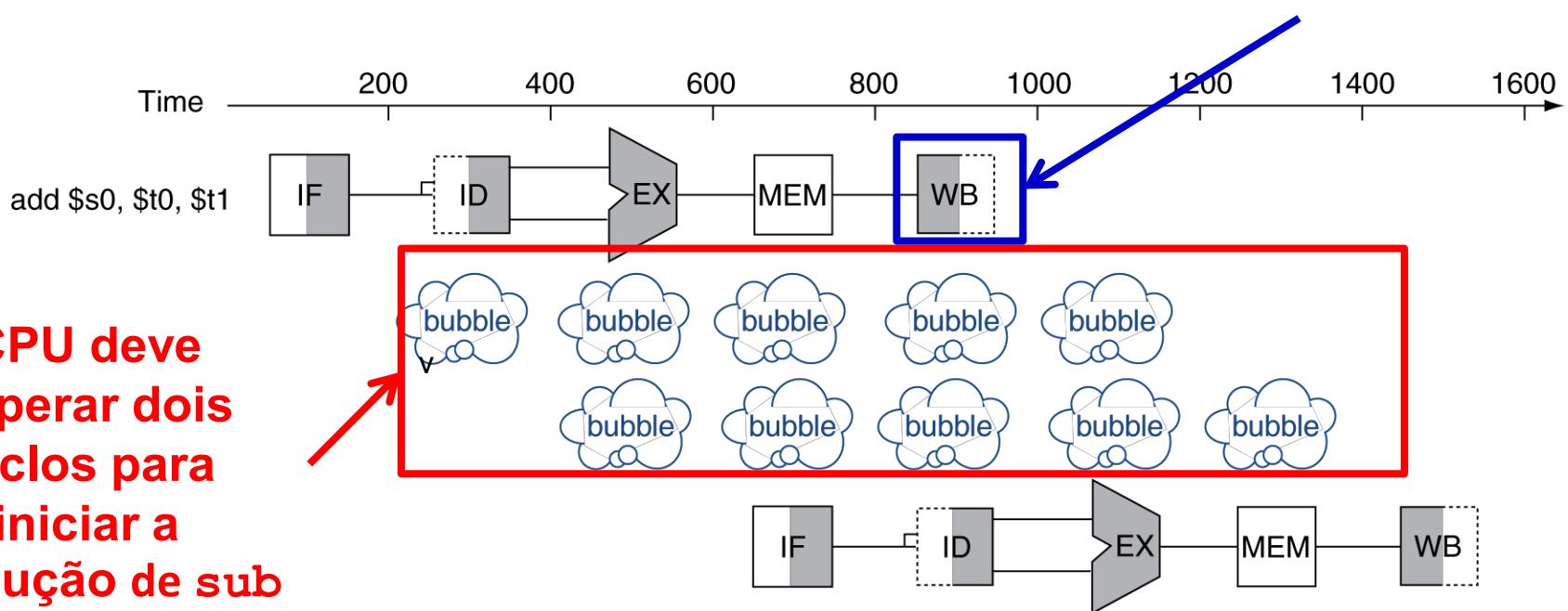
- (a) I, apenas
- (b) II, apenas
- (c) I e III, apenas
- (d) II e III apenas
- (e) I,II e III

← Letra b

# Conflito de Dados

- Uma instrução para ser executada depende de um dado gerado por uma instrução anterior
  - **add      \$s0 , \$t0 , \$t1**
  - sub      \$t2 , \$s0 , \$t3**

**Resultado da soma  
só será escrito em  
\$s0 neste ciclo**



## Exercício 2

(POSCOMP 2005 - 21) Considere uma CPU usando uma estrutura pipeline com 5 estágios (IF, ID, EX, MEM, WB) e com memórias de dados e de instruções separadas, sem mecanismo de data forwarding, escrita no banco de registradores na borda de subida do relógio e leitura na borda de descida do relógio e o conjunto de instruções a seguir:

I1: **lw \$2, 100(\$5)**

I2: **add \$1, \$2, \$3**

I3: **sub \$3, \$2, \$1**

I4: **sw \$2, 50(\$1)**

I5: **add \$2, \$3, \$3**

I6: **sub \$2, \$2, \$4**

- Quantos ciclos de relógio são gastos para a execução deste código?

# Resposta do Exercício 2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I1	IF	ID	EX	ME	WB												
I2		IF	ID	X	X	EX	ME	WB									
I3			IF	X	X	ID	X	X	EX	ME	WB						
I4						IF	X	X	ID	EX	ME	WB					
I5									IF	ID	X	EX	ME	WB			
I6										IF	X	ID	X	X	EX	ME	WB
I7																	

**I1: lw \$2, 100(\$5)**  
**I2: add \$1, \$2, \$3**  
**I3: sub \$3, \$2, \$1**  
**I4: sw \$2, 50(\$1)**  
**I5: add \$2, \$3, \$3**  
**I6: sub \$2, \$2, \$4**

**17 ciclos**

## Resposta do Exercício 2 (Com Forward)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I1	IF	ID	EX	ME	WB												
I2		IF	ID	X	EX	ME	WB										
I3			IF	X	ID	EX	ME	WB									
I4					IF	ID	EX	ME	WB								
I5						IF	ID	EX	ME	WB							
I6							IF	ID	EX	ME	WB						
I7																	

**I1: lw \$2, 100(\$5)**  
**I2: add \$1, \$2, \$3**  
**I3: sub \$3, \$2, \$1**  
**I4: sw \$2, 50(\$1)**  
**I5: add \$2, \$3, \$3**  
**I6: sub \$2, \$2, \$4**

**11 ciclos**

## Exercício 3

Considere uma CPU usando uma estrutura pipeline com 5 estágios (IF, ID, EX, MEM, WB) e com memórias de dados e de instruções separadas, com mecanismo de data forwarding, com previsão estática de que o desvio não se confirmará e o conjunto de instruções a seguir:

- I1: addi \$2, \$zero, 1
- I2: bne \$2,\$zero, I7
- I3: sub \$3, \$2, \$1
- I4: sw \$2, 50(\$1)
- I5: add \$2, \$3, \$3
- I6: sub \$2, \$2, \$4
- I7: lw \$4, 0 (\$1)

- Quantos ciclos de relógio são gastos para a execução deste código?

# Resposta do Exercício 3

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I1	IF	ID	EX	ME	WB												
I2		IF	ID	EX	ME	WB											
I3			IF	ID	EX	F	F										
I4				IF	ID	F	F	F									
I5					IF	F	F	F	F								
I7						IF	ID	EX	ME	WB							
I8																	

I1: addi \$2, \$zero, 1

I2: bne \$2,\$zero, I7

I3: sub \$3, \$2, \$1

I4: sw \$2, 50(\$1)

I5: add \$2, \$3, \$3

I6: sub \$2, \$2, \$4

I7: lw \$4, 0 (\$1)

10 ciclos

# Resposta do Exercício 3 (Com adiantamento de avaliação de desvio)

## I1: addi \$2, \$zero, 1

**I2: bne \$2,\$zero, I7**

### I3: sub \$3, \$2, \$1

I4: sw \$2, 50(\$1)

I5: add \$2, \$3, \$3

## I6: sub \$2, \$2, \$4

I7: Iw \$4, 0 (\$1)

# 10 ciclos

## Exercício 4

Considere a mesma CPU do exercício anterior (sem adiantamento de avaliação de desvio), mas com um slot para desvio retardado e o conjunto de instruções a seguir:

```
I1: add $s0, $zero,$zero
I2: lw $t0, 0($s0)
I3: sub $t0,$t0,$s2
I4: sw $t0, 0($s0)
I5: lw $t1,4($s0)
I6: addi $s1, $t1,1
I7: sw $s1, 4($s3)
I8: addi $s0,$s0,4
I9: slti $s2,$s0,8
I10: bne $s2,$zero, I2
I11: sw $t2,0($s3)
I12: add $s0, $zero, $zero
```

- Modifique este código para evitar muitos retardos e calcule os ciclos de relógio para a execução da primeira iteração do laço?

# Possível Solução

Rearrumando o código:

```
I1: add $s0, $zero,$zero
I2: lw $t0, 0($s0)
I3: sub $t0,$t0,$s2
I4: sw $t0, 0($s0)
I5: lw $t1,4($s0)
I6: addi $s1, $t1,1
I7: sw $s1, 4($s3)
I8: addi $s0,$s0,4
I9: slti $s2,$s0,8
I10: bne $s2,$zero, I2
I11: sw $t2,0($s3)
I12: add $s0, $zero, $zero
```



```
I1: add $s0, $zero,$zero
I2: lw $t0, 0($s0)
I3: lw $t1,4($s0)
I4: sub $t0,$t0,$s2
I5: sw $t0, 0($s0)
I6: addi $s1, $t1,1
I7: addi $s0,$s0,4
I8: slti $s2,$s0,8
I9: bne $s2,$zero, I2
I10:sw $s1, 4($s3) #slot
I11: sw $t2,0($s3)
I12: add $s0, $zero, $zero
```

# Possível Resposta do Exercício 4

## Exercício 5

(POSCOMP 2008 – 32 Modificada) Analise as seguintes afirmativas e indique quais são falsas.

- I. Uma arquitetura multithreading executa simultaneamente o código de diversos fluxos de instruções (threads).
- II. Em uma arquitetura VLIW, o controle da execução das várias instruções por ciclo de máquina é feito pelo compilador.
- III. Uma arquitetura superescalar depende de uma boa taxa de acerto do mecanismo de predição de desvio para obter um bom desempenho.
- IV. Um processador dual-core tem eficiência equivalente em termos de consumo de energia do que dois processadores *single-core* de mesma tecnologia.

Falsa

# Exercício 6

Considere as seguintes organizações de CPU:

1. CPU – FG : Processador superescalar de grau 2, que utiliza política de Fine-Grain para escalonar threads
2. CPU – SMT: Processador superescalar de grau 2, que utiliza política Simultaneous Multithreading para escalonar threads

Assuma que temos as threads A e B que contém as seguintes instruções:

Thread A	Thread B
A1 – leva 2 ciclos para executar	B1 – nenhuma dependencia
A2 – depende do resultado de A1	B2 – conflito estrutural com B1
A3 – conflito estrutural com A2	B3 – nenhuma dependencia
A4 – depende do resultado de A2	B4 – depende do resultado de B2

- Quantos ciclos levará para executar as 2 threads para cada configuração de CPU?

# Possível Resposta do Exercício 6

- CPU-FG: 7 ciclos

FU1	FU2
A1	A3
A1	
B1	B3
A2	
B2	
A4	
B4	

# Possível Resposta do Exercício 6

- CPU-SMT: 5 ciclos

FU1	FUI2
A1	B1
A1	B2
A2	B3
A3	B4
A4	

# Exercício 7

Considere um processador multicore com cores heterogêneos : A, B, C e D, onde B é 2 vezes mais rápido que A, C é 3 vezes mais rápido que A, e D e A tem velocidades iguais de processamento. Suponha que uma aplicação precisa computar o quadrado de cada elemento de um array de 256 elementos. Considere a seguinte divisão de trabalho:

Core	Quantidade de Elementos
A	32
B	128
C	64
D	32

- Qual o tempo de execução desta aplicação, considerando que o processamento de cada elemento em A leva 1 ns?

# Resposta do Exercício 7

- Tempo de execução = 64 ns

Tempo total de execução =  $\max(32/1, 128/2, 64/3, 32/1) = 64 \text{ ns}$

# Infraestrutura de Hardware

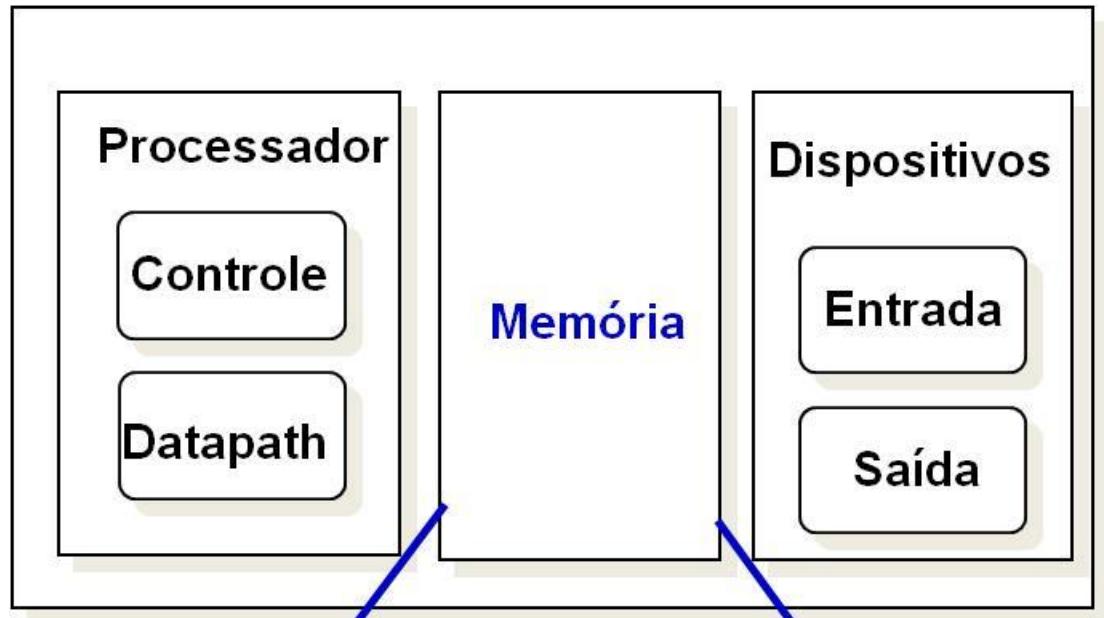
## Explorando a Hierarquia de Memória

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Componentes de um Computador

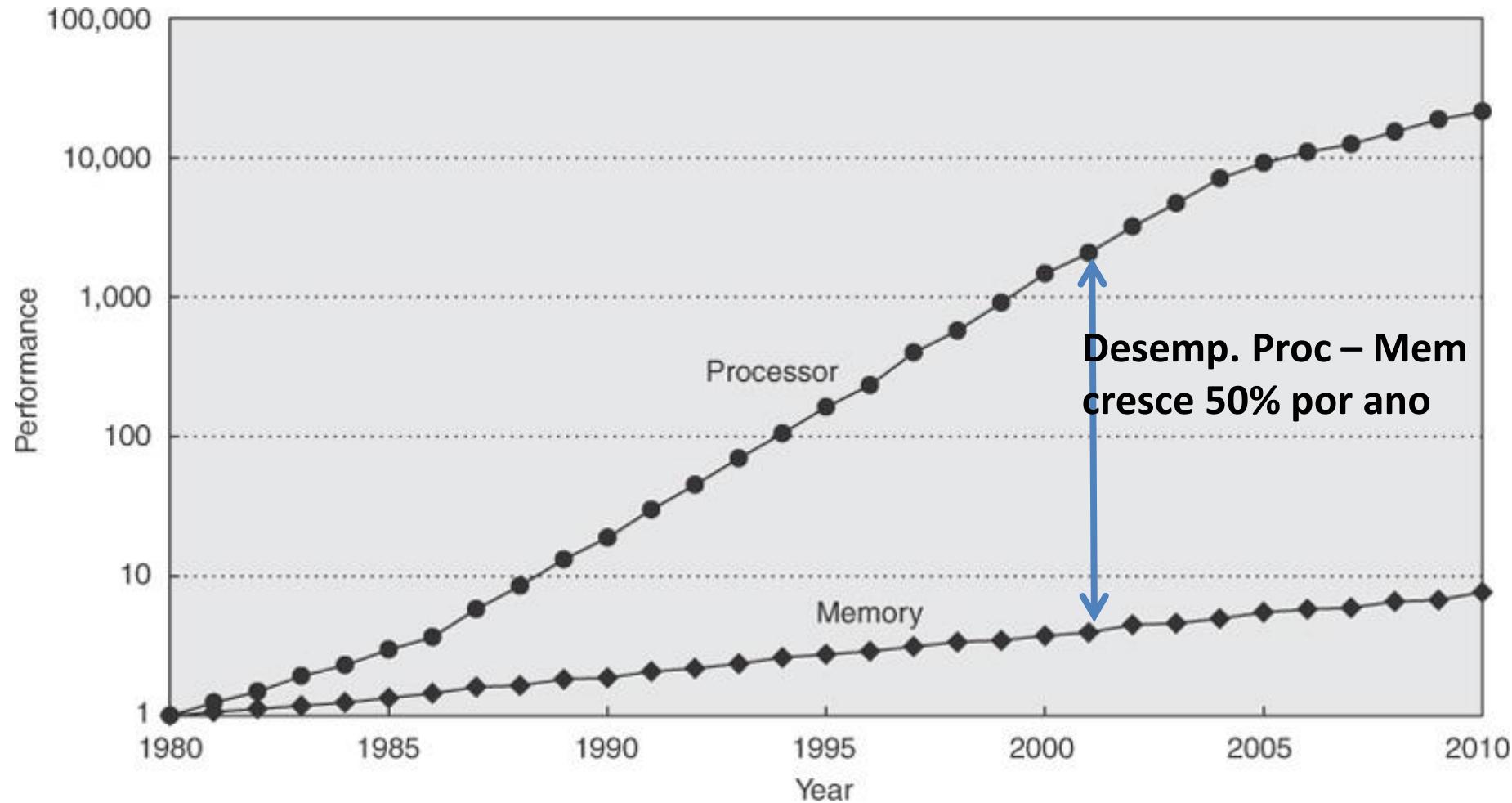


Sistema de memória de  
um computador é  
composto por vários  
tipos de memória

# Importância da Memória no Desempenho

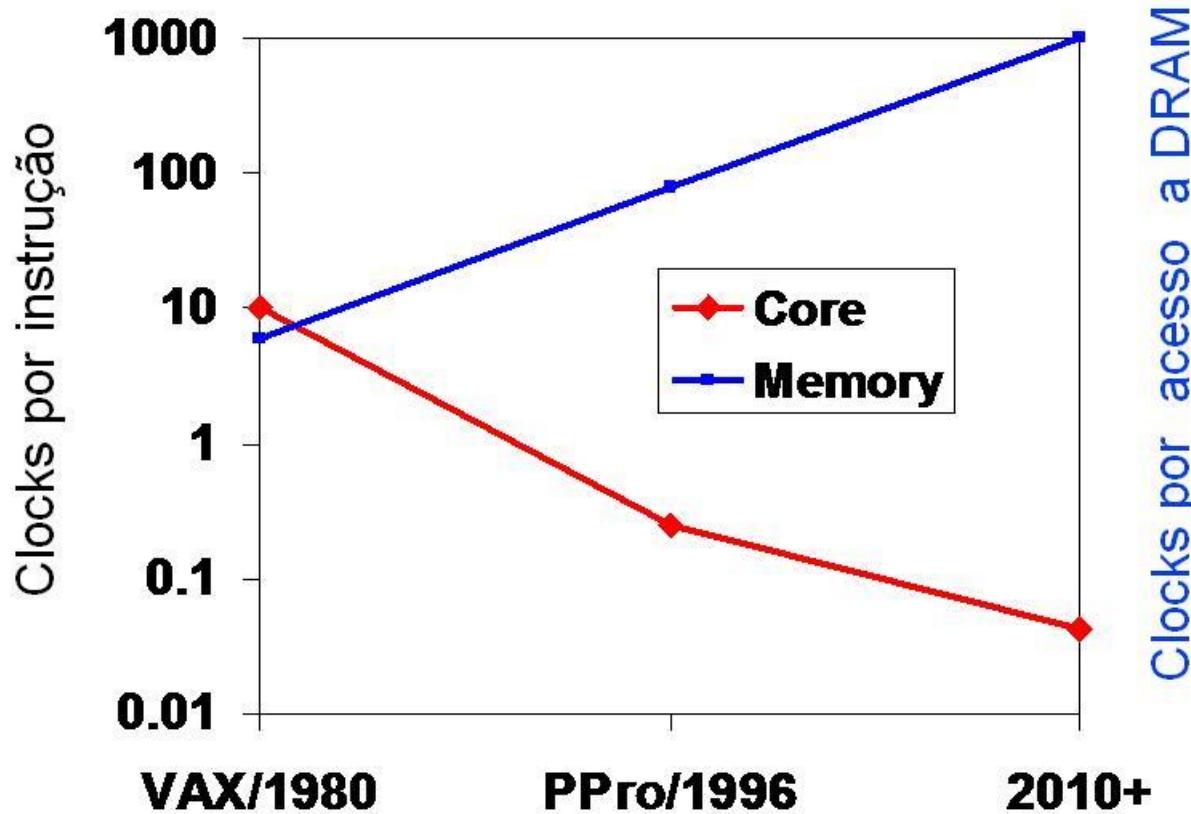
- Uma forma de aumentar o desempenho de um computador é melhorar o desempenho da CPU
  - Algumas técnicas vistas:
    - Pipeline, Superescalar, Multithread, Multicore
- Comumente, programas contém muitas instruções que acessam a memória
- Portanto, tempo de acesso a memória deve também ser otimizado
- Tempo de acesso a memória representa, na maioria das vezes, o gargalo da execução de um programa

# Processador x Memória: Desempenho

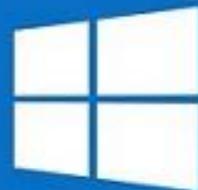


# “Memory Wall”

- Desempenho do sistema limitado pela memória
- Bom projeto de sistema de memória é fundamental para melhorar desempenho



# Requisitos Mínimos do Windows 10



## Windows 10 System Requirements

1 Gigahertz (GHz) processor

1 Gigabyte (GB) RAM for 32-bit version

2 Gigabytes (GB) RAM for 64-bit version

16 GB available storage for 32-bit version

20 GB available storage for 64-bit version

Graphics card with DirectX 9 Compatibility

Display with at least 800x600 resolution

**Requisitos do  
sistema de memória**

# Tecnologias de Memória

Tecnologia	Ano	Tempo de acesso	U\$/Gbyte
Static RAM (SRAM)	2021	0,5 – 2,5ns	500 a 1000
Dynamic RAM (DRAM)	2021	50 - 70ns	< 10
Disco	2021	5-20 ms ou 5.000.000 a 20.000.000 ns	< 0,05

- Memória ideal: velocidade de SRAM , tamanho e custo de disco!

# RAM Dinâmica vs. Estática

- DRAM (Dynamic Random Access Memory)
  - Maior tempo de acesso (50 – 70 ns)
  - Perda de informação após algum tempo
    - Necessidade de refreshing (~32 - 64ms)
  - Grande capacidade de integração (baixo custo por bit)
  
- SRAM (Static Random Access Memory)
  - Pequeno tempo de acesso (0,5 – 2,5 ns)
  - Não existe necessidade de refreshing
  - Alto custo por bit (alta integração)

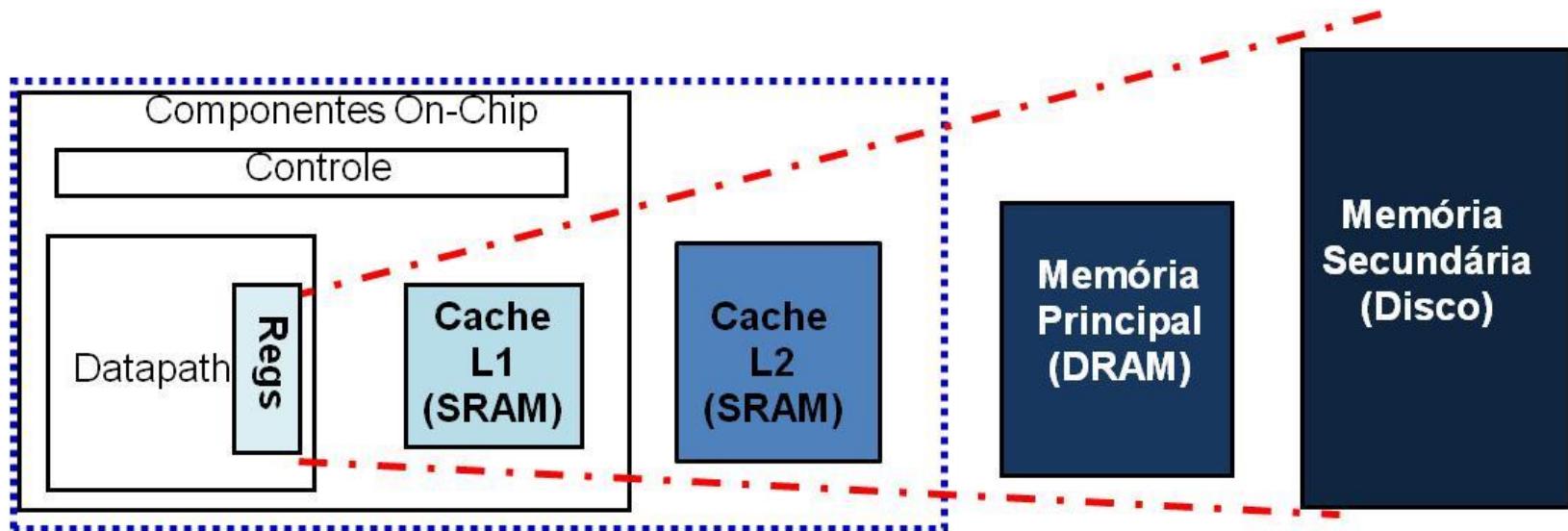
# Como Minimizar Tempo de Acesso a Memória?

- Programador deseja ter memória maior e mais veloz
- Existem vários tipos de memória construídos com tecnologia diferente
  - SRAM, DRAM, Magnético, etc
- Diferença de velocidade e de custo
  - Fato: Mais velozes → mais caras
    - Maiores são mais lentas, menores são mais rápidas
- Deve-se criar ilusão de oferecer memória maior, mais barata e mais rápida(na maioria das vezes)

**Hierarquia de Memória!**

# Hierarquia de Memória

- Aproveitar **princípio da localidade** para oferecer o máximo de memória de baixo custo com uma velocidade de acesso oferecida pelo tipo de memória mais rápida



**Tempo (%ciclos):**  $\frac{1}{2}$ 's      1's      10's      100's      10000's

**Tamanho(bytes):** 100's      10K's      M's      G's

# O Que é Princípio da Localidade?

## ■ Localidade Temporal

- Programa tende a referenciar as instruções e dados referenciados recentemente
  - **Mantenha itens mais recentemente referenciados junto ao processador**
  - Ex: instruções de um loop

## ■ Localidade Espacial

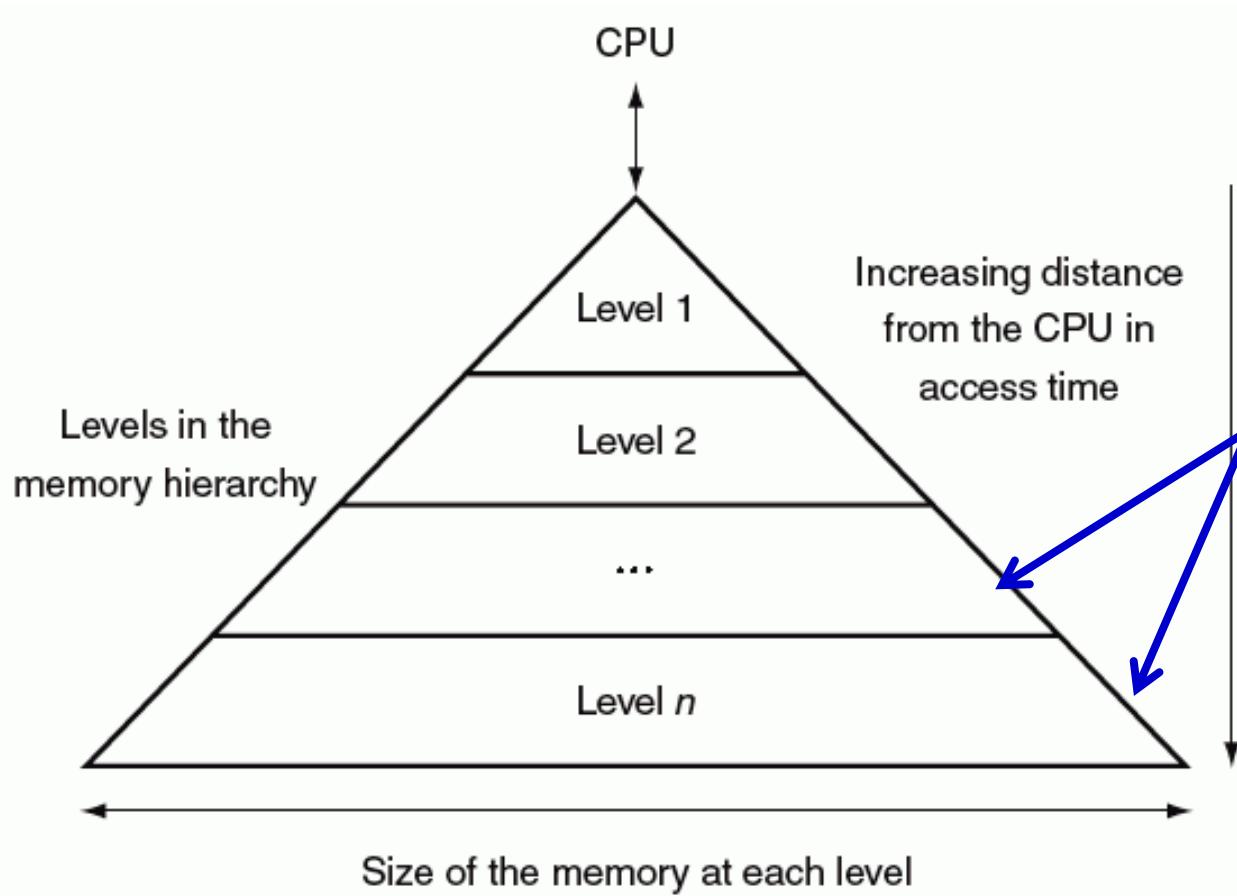
- Programa tende a referenciar as instruções e dados que tenham endereços próximos das últimas referências
  - **Mova blocos de dados de palavras contíguas para junto do processador**
  - Ex: dados de um array

# Aproveitando o Princípio da Localidade

- Armazene tudo em disco
  - Ilusão de que a memória é muito grande
- Copie itens acessados **recentemente** e itens em endereços **próximos** do disco para uma memória DRAM que é menor
  - Memória principal
- Copie itens acessados **mais recentemente** e itens em endereços **mais próximos** da DRAM para uma SRAM menor
  - Memória cache on-chip
  - Ilusão de que a memória é muito rápida

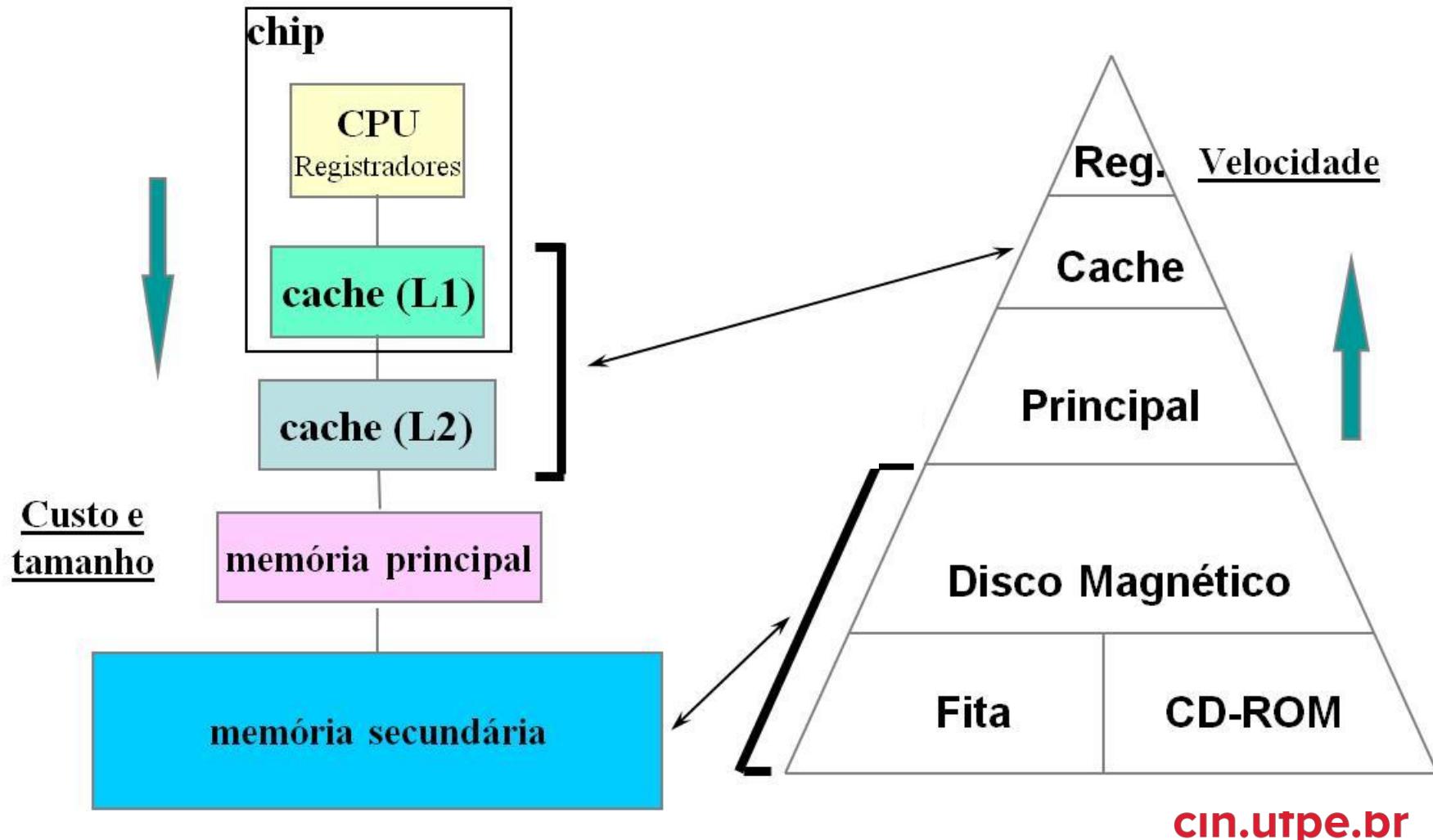
# Idéia Geral de Hierarquia de Memória

- Níveis mais altos de memória são mais rápidas
- Níveis mais baixos são maiores



Cada nível mais alto possui um subconjunto de dados/instruções do nível mais abaixo

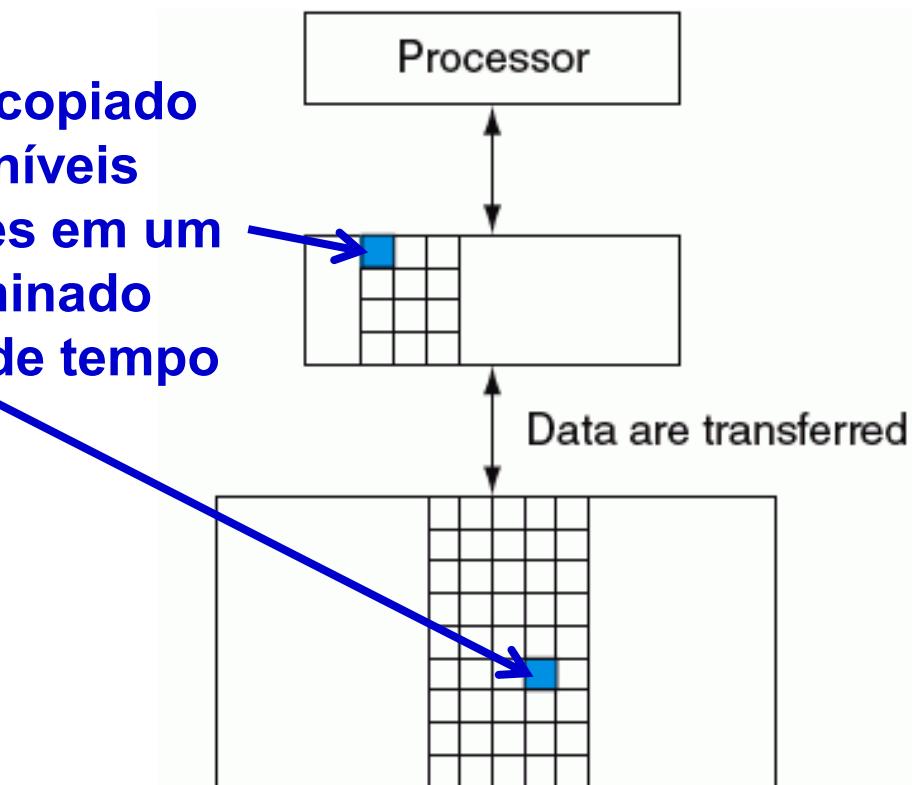
# Sistema Hierárquico de Memória



# Hierarquia de Memória: Terminologias

- **Bloco** (linha) é a unidade mínima de informação que pode estar presente ou não em uma cache
  - Geralmente é a unidade que é copiada entre um nível e outro

**Bloco é copiado  
entre níveis  
adjacentes em um  
determinado  
instante de tempo**



# Hierarquia de Memória: Terminologias

- Hit (acerto) ocorre quando os dados requisitados pela CPU estão em algum bloco do nível de memória desejado
  - Hit time – Tempo para acessar dado no nível desejado
  - Hit rate – Hits/acessos
- Miss (falta) ocorre quando os dados requisitado pela CPU não estão em algum bloco do nível de memória desejado
  - Miss rate – Misses/acessos ou  $(1 - \text{hit rate})$
- Miss penalty é o tempo que leva para buscar o bloco de um nível mais abaixo para o nível mais acima e enviá-lo para a CPU
  - Tempo de acesso do bloco no nível mais abaixo + tempo de transmissão do bloco para nível mais acima + tempo de escrita do bloco no nível mais acima + tempo de transmissão para a CPU

# Infraestrutura de Hardware

## Cache

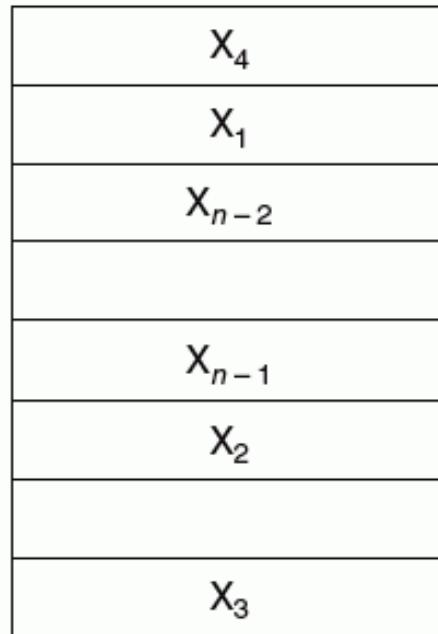
Prof. Adriano Sarmento

# Cache

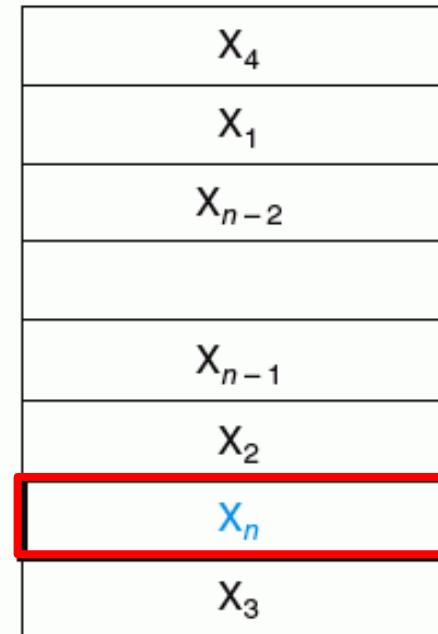
- Memória SRAM menor que memória principal (DRAM) localizada perto da CPU
  - Acesso rápido
    - Tecnologia mais rápida
    - Nível de memória mais perto da CPU, não utiliza barramento comum aos outros dispositivos
- Utiliza-se do princípio da localidade para armazenar dados acessados mais recentemente e de endereços mais próximos

# Exemplo de Cache Simples

- Suponha cache onde bloco é igual a uma palavra
- Dados os acessos as palavras  $X_1, \dots, X_{n-1}$ , o acesso a  $X_n$  geraria uma falta (miss), depois o elemento seria buscado e colocado na cache



Cache antes do acesso a  $X_n$



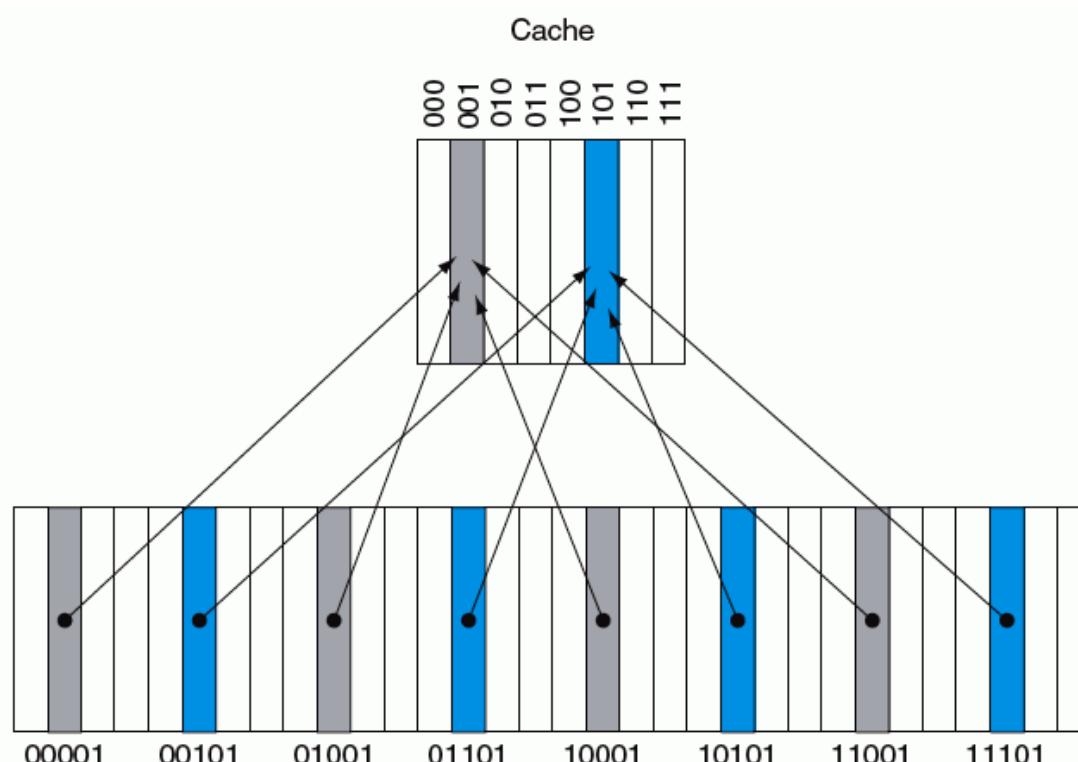
Cache depois do acesso a  $X_n$

# Acessando Dados da Cache

- Pergunta 1: Como saber se um dado está na cache?
- Pergunta 2: Se dado estiver, como localizamos ele na cache?

# Mapeamento Direto

- Forma mais simples de determinar a localização de um bloco na cache é pelo endereço do bloco na memória
- Cache com mapeamento direto faz com que cada bloco da memória seja sempre mapeado para o mesmo bloco da cache



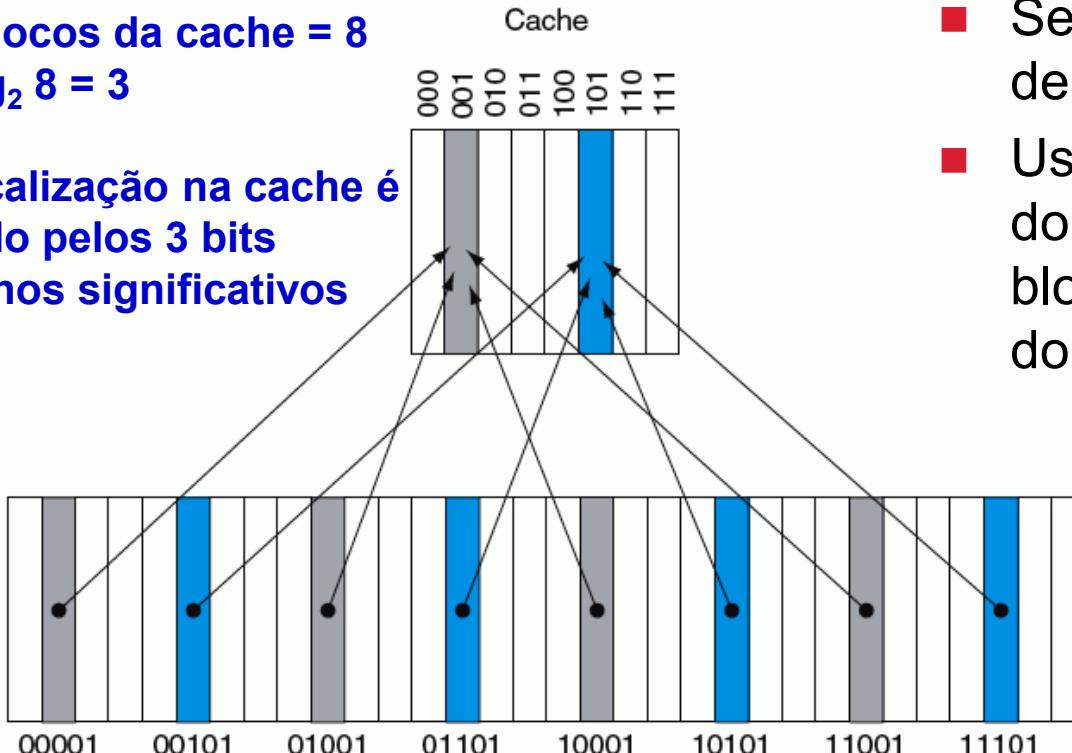
# Localizando Blocos com Mapeamento Direto

- Localização do bloco de memória na cache (resposta da pergunta 2):
  - (endereço do bloco) modulo (#blocos na cache)
- Muitos blocos de memória compartilham o mesmo bloco da cache em instantes diferentes

# blocos da cache = 8

$$\log_2 8 = 3$$

Localização na cache é dado pelos 3 bits menos significativos



- Se # blocos da cache, potência de 2
- Usa-se bits menos significativos do endereço da memória do bloco para determinar o número do bloco da cache

$\log_2 \# \text{blocos}$  bits menos significativos

# Verificando se Bloco Está na Cache

- Cache armazena não só os dados de um bloco de memória, mas também parte do endereço deste bloco
  - Tag
- Uma **tag** é formada pelos bits mais significativos do endereço do bloco
  - Todos os bits do endereço do bloco, menos os usados para identificar um bloco da cache
- Uso de tags permite saber se bloco de memória está ou não na cache
  - Resposta para a pergunta 1
- Cache ainda armazena um bit de validade (valid bit) para saber se o conteúdo do bloco ainda é valido
  - Para os casos onde computador é iniciado e ou quando a cache ainda possuir algum lixo por causa do fim de execução de programa

# Tag

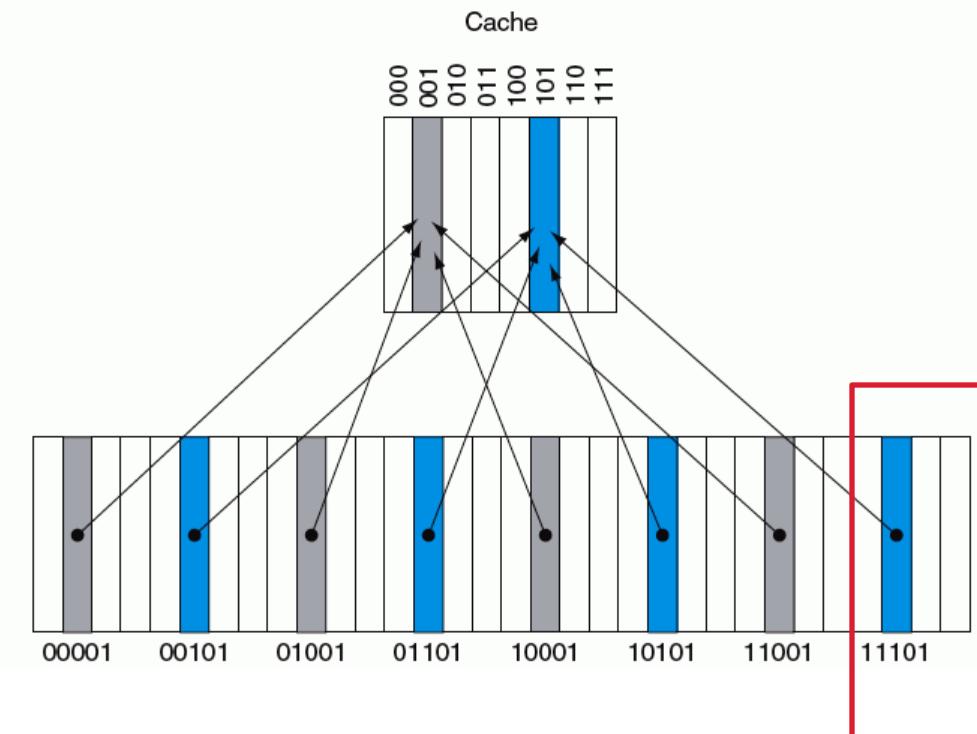
Validade	Tag	Dados
000		
001		
010		
011		
100		
101	1	11
110		
111		

**# blocos da cache = 8**  
 $\log_2 8 = 3$

**Localização na cache é  
dado pelos 3 bits  
menos significativos**

**Tag = bits mais significativos**

**Este bloco de memória  
está na cache**



# Endereçando a Cache

## ■ Composição de um endereço de memória

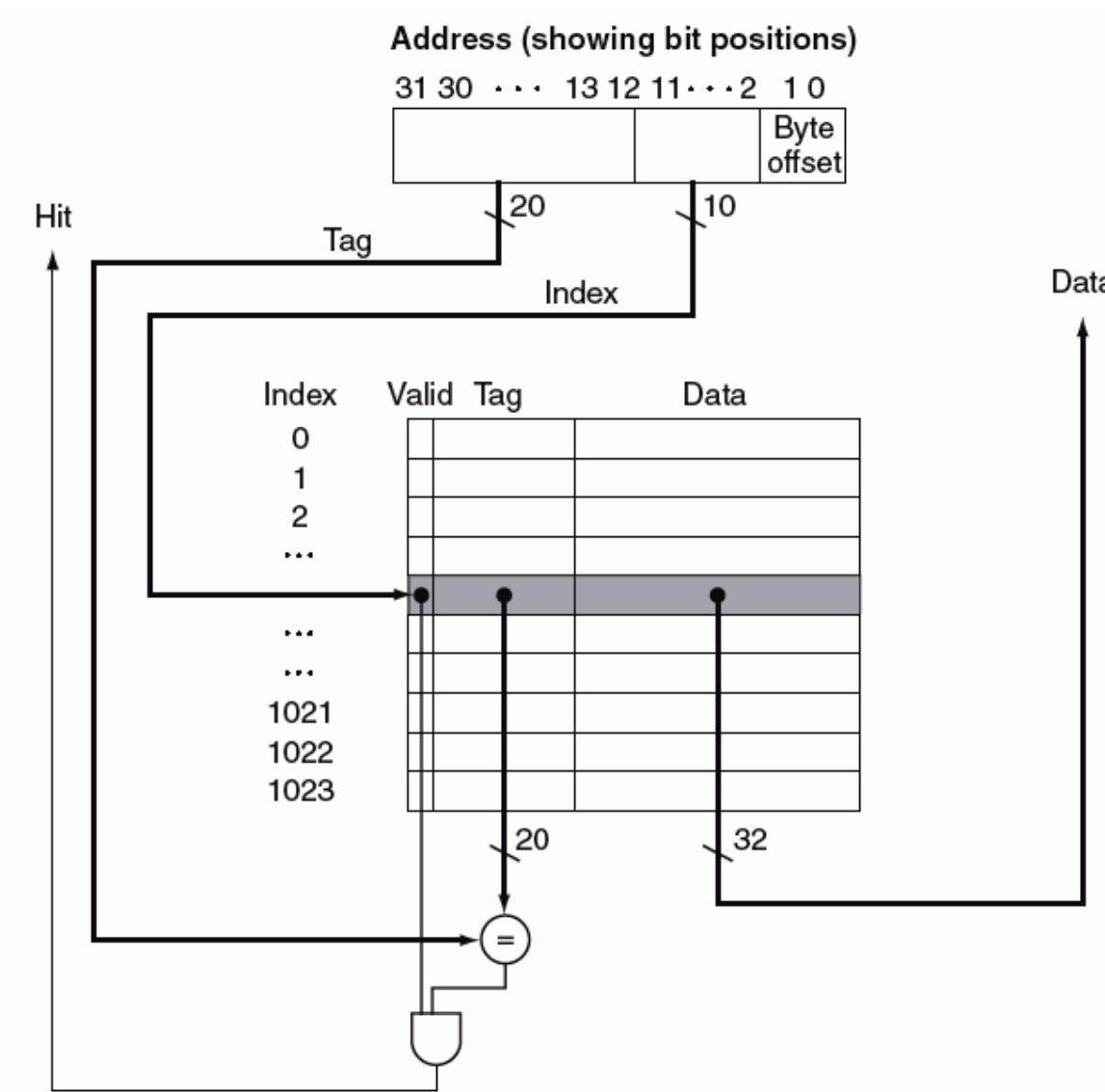
- Tag
  - Índice do bloco
  - Posição de byte dentro do bloco
- } **Endereço do bloco**

## ■ Exemplo:

- Memória: endereço de 32 bits, cada bloco é uma palavra, acesso por palavra(32 bits), endereçamento por byte
- Cache: capacidade para armazenar 64 palavras

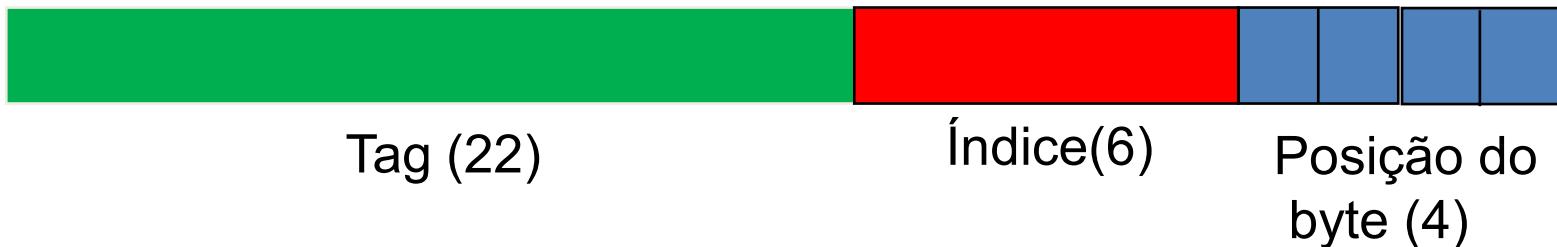


# Mapeando Endereços de Memória na Cache



# Tamanho do Bloco da Cache

- Cada bloco de uma cache pode conter mais de uma palavra
  - Aproveita a localidade espacial
- Exemplo:
  - Memória: endereço de 32 bits, acesso por palavra(32 bits), endereçamento por byte
  - Cache: capacidade para armazenar 64 blocos de **4 palavras cada** (tamanho bloco:16 Bytes)

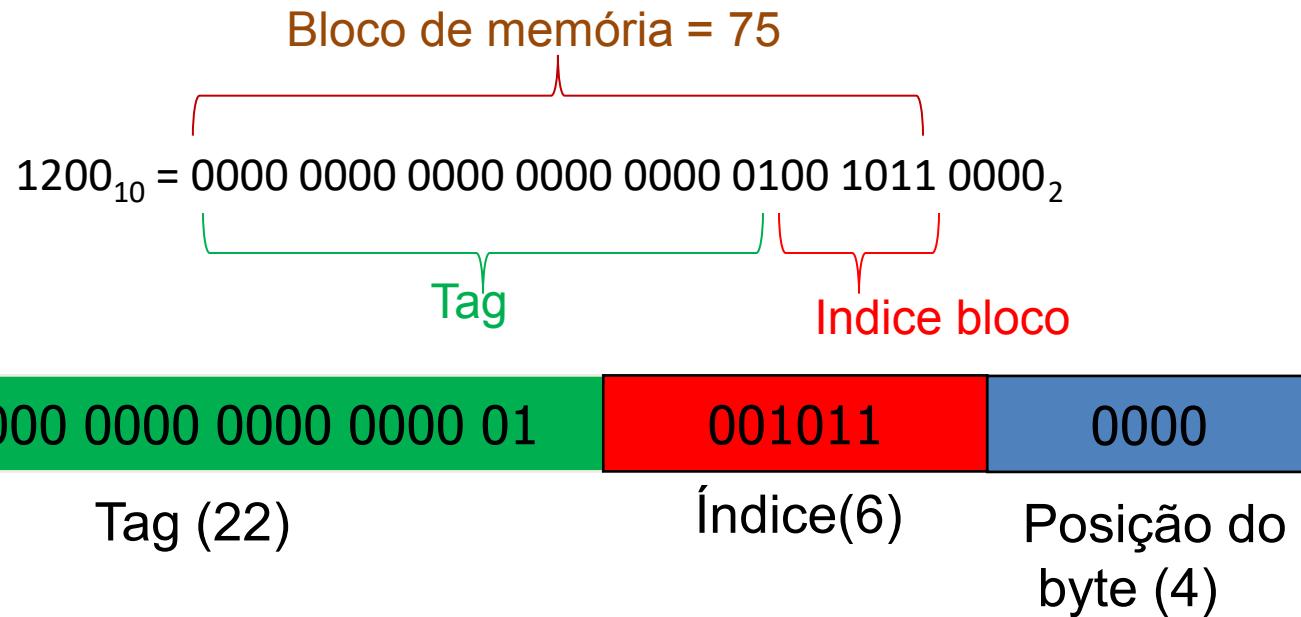


# Exemplo: Mapeando Endereço em Bloco da Cache

- 64 blocos, 16 bytes/bloco
  - `addi $s0, $zero, 1200`
  - `lb $t0, 0 ($s0)`
  - Endereço do byte 1200 é mapeado para que bloco da cache?
- Endereço do bloco na memória =  $[1200/16] = 75$
- Número do bloco da cache =  $75 \text{ modulo } 64 = 11$

# Exemplo: Mapeando Endereço em Bloco da Cache

- 64 blocos, 16 bytes/bloco
  - Endereço do byte 1200 é mapeado para que bloco da cache?
- Endereço do bloco na memória =  $[1200/16] = 75$
- Número do bloco da cache =  $75 \text{ modulo } 64 = 11$



# Exemplo de Utilização de Cache com Mapeamento Direto

- 8 blocos, 1 palavra/bloco, mapeamento direto

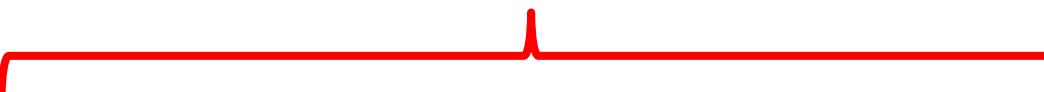
**Estado Inicial**

<u>Índice</u>	<u>V</u>	<u>Tag</u>	<u>Dado</u>
<u>000</u>	<u>N</u>		
<u>001</u>	<u>N</u>		
<u>010</u>	<u>N</u>		
<u>011</u>	<u>N</u>		
<u>100</u>	<u>N</u>		
<u>101</u>	<u>N</u>		
<u>110</u>	<u>N</u>		
<u>111</u>	<u>N</u>		

# Exemplo de Utilização de Cache com Mapeamento Direto

End. decimal	End. Binário	Hit/miss	Bloco da cache
22	10 110	Miss	110

Apenas essa parte é armazenada na cache



Índice	V	Tag	Dado
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
<b>110</b>	<b>Y</b>	<b>10</b>	<b>Mem[10110]</b>
111	N		

# Exemplo de Utilização de Cache com Mapeamento Direto

End. decimal	End. Binário	Hit/miss	Bloco da cache
26	11 010	Miss	010

Índice	V	Tag	Dado
000	N		
001	N		
<b>010</b>	<b>Y</b>	<b>11</b>	<b>Mem[11010]</b>
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Exemplo de Utilização de Cache com Mapeamento Direto

End. decimal	End. binário	Hit/miss	Bloco da cache
22	10 110	Hit	110
26	11 010	Hit	010

Índice	V	Tag	Dado
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Exemplo de Utilização de Cache com Mapeamento Direto

End. decimal	End. binário	Hit/miss	Bloco da cache
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Índice	V	Tag	Dado
<b>000</b>	<b>Y</b>	<b>10</b>	<b>Mem[10000]</b>
001	N		
010	Y	11	Mem[11010]
<b>011</b>	<b>Y</b>	<b>00</b>	<b>Mem[00011]</b>
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Exemplo de Utilização de Cache com Mapeamento Direto

End. decimal	End. binário	Hit/miss	Bloco da Cache
18	10 010	Miss	010

Índice	V	Tag	Dado
000	Y	10	Mem[10000]
001	N		
<b>010</b>	<b>Y</b>	<b>10</b>	<b>Mem[10010]</b> ← <span style="color:red">Bloco de memória sobrescreve o antigo</span>
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Faltas (Misses) na Cache

- Ocorrendo uma falta na cache
  - Congela o pipeline
  - Busca bloco na memória
    - Com ajuda de controlador de HW que acessa memória e preenche cache
    - Se o bloco da cache estiver preenchido com outro bloco, a informação é sobre-escrita
  - Caso a falta seja referente a uma instrução
    - Reinicia a busca da instrução
  - Caso a falta seja referente a um dado
    - Completa o acesso ao dado
- Desempenho é penalizado

# Tipos de Acesso à Cache

- Leitura
  - Simples de implementar
- Escrita
  - Lento e complicado
  - Dado e tag são atualizados na cache
  - Inconsistencia entre memória principal e cache!!
    - se um bloco da cache foi alterado pela CPU, não pode ser descartado da cache sem garantir que foi copiado para a memória principal
  - **Como resolver?**

# Políticas de Escrita e Consistência

- Caches do tipo Write through
  - Cache e memória são atualizadas simultaneamente
  - Cache e memória sempre consistentes
  - Penaliza desempenho, CPU deve esperar acesso a memória
  
- Caches do tipo Write back
  - Memória principal somente é atualizada quando bloco é substituído da cache
  - Cache e memória momentaneamente inconsistentes
  - Usa dirty bit para marcar linhas alteradas na cache
  - Reduz quantidade de acessos a memória

# Analisando as Diferentes Políticas de Escrita

<b><i>Write through</i></b>	<b><i>Write back</i></b>
<b><i>facilidade de implementação</i></b>	<b><i>redução de acessos à memória</i></b>
<b><i>consistência da memória principal</i></b>	

- Dois métodos requerem que a CPU espere a escrita
- **Solução: Write buffers**
  - Bloco é escrito no buffer e CPU pode continuar execução

# Infraestrutura de Hardware

Cache em Sistemas Multiprocessados  
- Coerência

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Multicores - Coerência das Caches

- Suponha que 2 cores compartilham um mesmo espaço de memória
  - Caches Write-through

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

# Definindo Coerência

- Informalmente: Leitura retorna valor escrito mais recentemente
- Formalmente:
  - P escreve em X; P lê valor em X (nenhuma escrita entre as duas operações)  
⇒ leitura retorna valor escrito
  - $P_1$  escreve em X;  $P_2$  lê em X (intervalo de tempo suficiente)  
⇒ leitura retorna valor escrito
  - $P_1$  escreve em X,  $P_2$  escreve em X  
⇒ todos os processadores enxergam escritas na mesma ordem
    - No final todos terão o mesmo valor de X

# Supporte a Compartilhamento com Coerência

- Migração de dados para caches locais
  - Valor mais atualizado da variável é movido entre a cache que tem o valor mais atual e a cache que precisa do dado
  - Reduz tempo de acesso e largura de banda da memória compartilhada
- Replicação dos dados compartilhados para leitura
  - Cada cache possui cópias de dados compartilhados que estão usando no momento
  - Reduz tempo de acesso e contenção para acesso

# Protocolos de Coerência de Cache

- Para garantir coerência é necessário mecanismos implementados em hardware
  - **Protocolos Snooping**
    - Cada cache monitora leituras/escritas no barramento
    - Cada bloco da cache mantém também o estado de compartilhamento do dado
  - **Protocolos baseados em Diretórios**
    - Caches e memória mantêm status de compartilhamento de blocos em um diretório
    - Diretório pode ser centralizado ou distribuído

# Protocolos Snooping

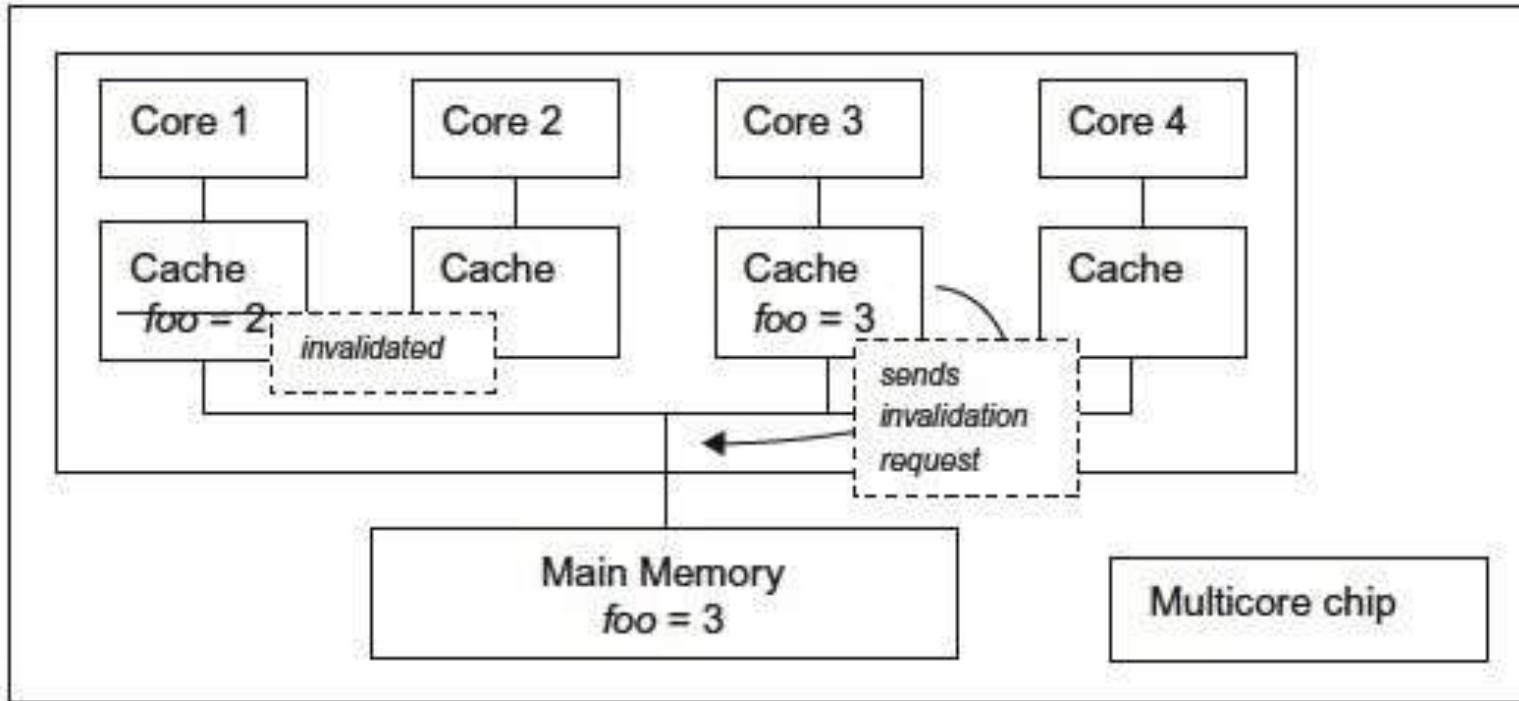
## ■ Write Invalidate

- Escrita em uma cache de um dado compartilhado invalida todas as cópias das outras caches
- Protocolo Snooping mais utilizado hoje em dia por multiprocessadores

## ■ Write Update

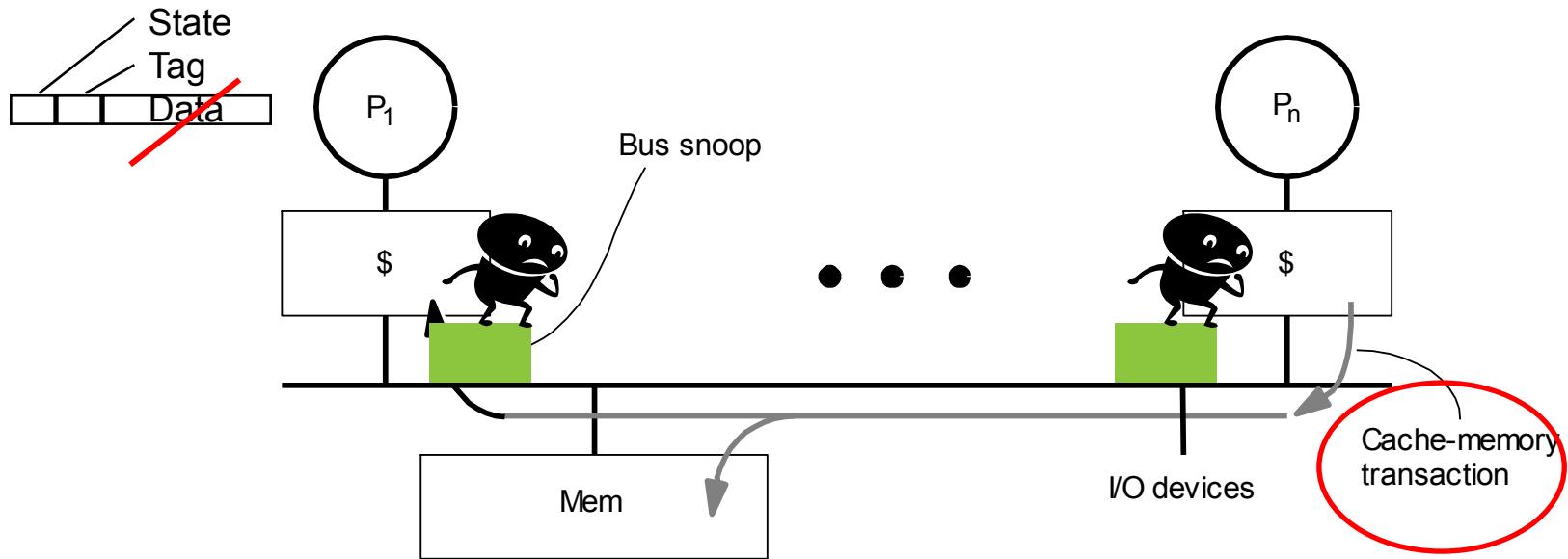
- Escrita em uma cache de um dado compartilhado deve vir acompanhada de atualização de todas as cópias das outras caches
- Requer maior utilização do barramento

# Snooping – Write Invalidate



- Core que escreve manda uma mensagem para invalidar outras cópias presentes em outra caches
  - Se cache write-back, qualquer leitura por outro core, será fornecida pela cache atualizada

# Write Invalidate : Controladores de Cache



- Controladores de Cache “snoop” (monitoram) todas as transações no barramento
  - Realizam ações para garantir coerência
    - **invalidam**, atualizam, ou fornecem valor
  - Atualizam estados de compartilhamento dos blocos das caches

# Mais Sobre Snooping – Write Invalidate

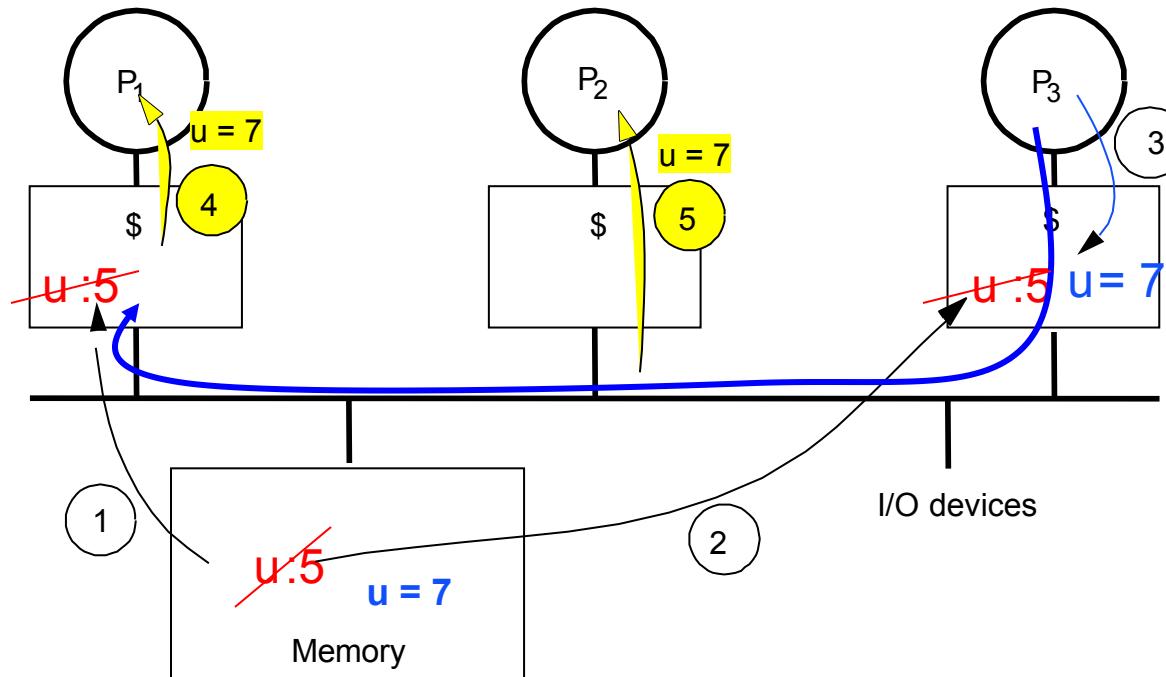
- Cache tem acesso exclusivo ao bloco quando vai escrever
  - Broadcasts uma mensagem de invalidação no barramento
  - Escrita só é completada quando cache obtém acesso ao barramento
  - Leituras subsequentes de outras caches causam um miss
    - Cache com cópia atualizada fornece o resultado

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

# Localização de Cópia Mais Atualizada para Leitura

- Depende da política de escrita da cache
  - **Write-through**
    - Sempre se busca na memória
    - Resulta em uma maior quantidade de acessos a memória
  - **Write-back**
    - Cache com cópia mais atualizada atende às requisições
    - Reduz acessos à memória
    - Maioria dos multiprocessadores utilizam write-back

# Exemplo: Write-thru Invalidate



- Processador P3 deve invalidar todas as cópias antes da escrita
- Dado é atualizado primeiro na memória
- Demais caches buscam dado na memória

# Localização de Cópia Mais Atualizada em Caches Write-Back

- Cache com cópia atualizada responde requisições do bloco de memória
  - Controladores das caches monitoram o barramento para saber se possuem a cópia mais atualizada
  - Cancela a requisição à memória principal
- Blocos da cache precisam armazenar também estado do bloco
  - **shared**: bloco pode ser lido, pois a cópia está atualizada
  - **modified/exclusive**: cache possui a única cópia válida que pode ser escrita (marcada também como dirty)
  - **invalid**: bloco inválido
- Somente escritas em blocos “shared” precisam mandar a requisição de invalidação para as demais caches

# Leitura em Caches Write-Back com Write Invalidate

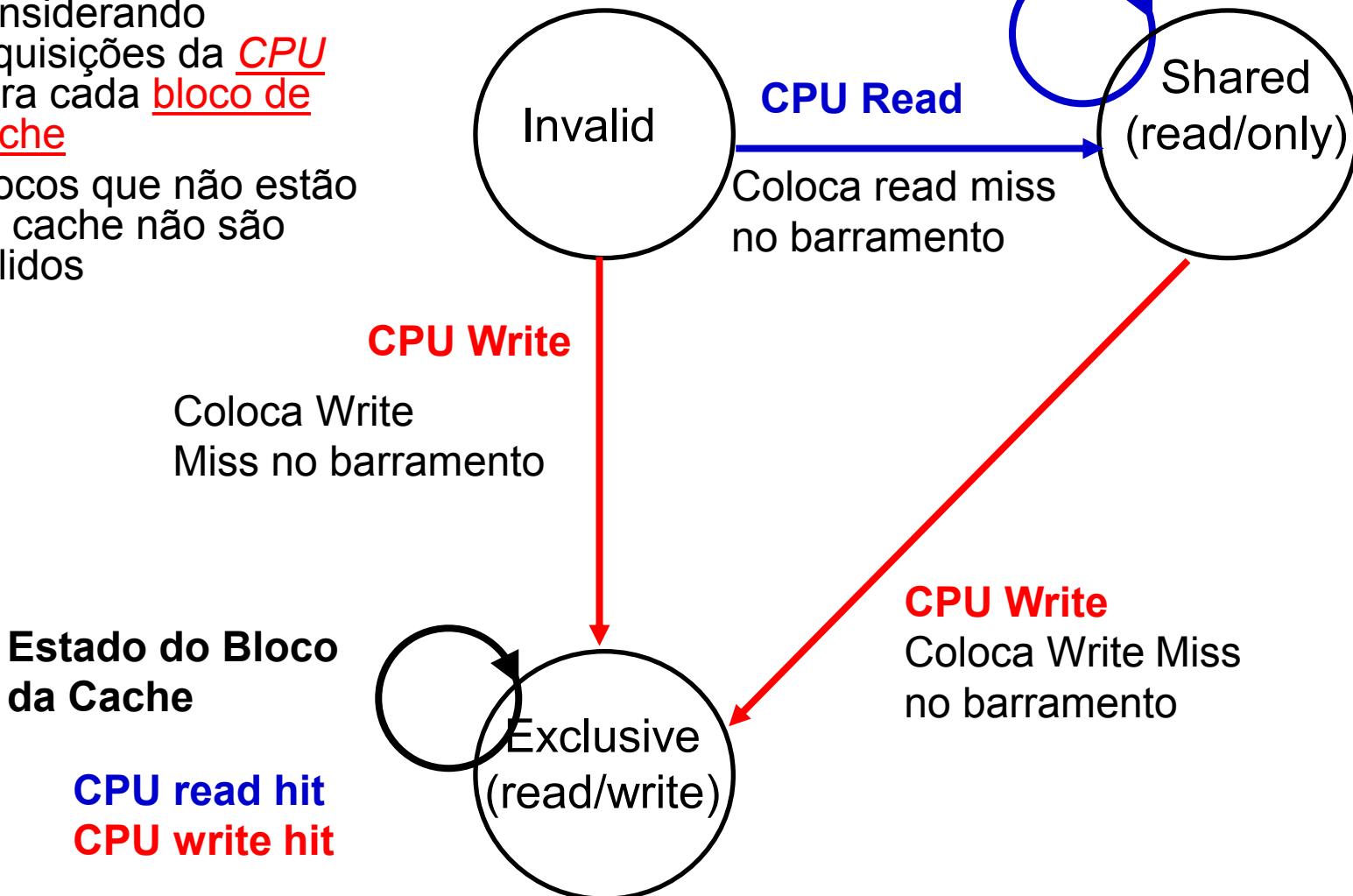
1. CPU faz requisição de leitura
2. Se cache local não possui o bloco, coloca-se um read miss no barramento
3. Controladores das outras caches dão um “snoop” no barramento
4. Cache que possui o bloco marcado como **exclusive** responde a requisição cancelando acesso à memória

# Escrita em Caches Write-Back com Write Invalidate

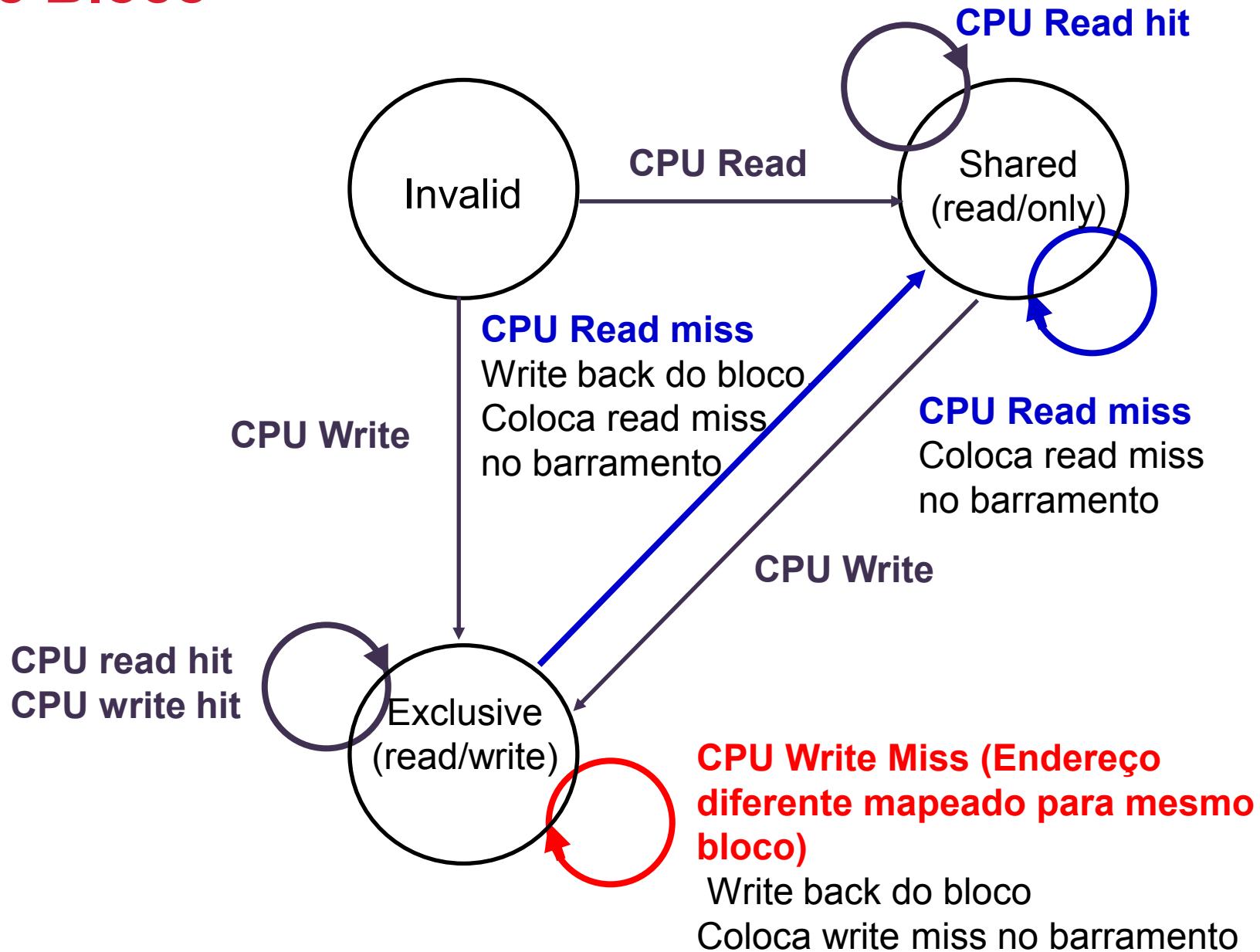
1. CPU faz requisição de escrita
2. Se cache local não possui o bloco, coloca-se um write miss no barramento
3. Controladores das outras caches dão um “snoop” no barramento
4. Cache que possui o bloco marcado como **exclusive** dá um “write-back” (atualização) na memória e invalida a sua cópia
5. Cache que possui bloco marcado como **shared** invalida a sua cópia

# Snooping: Write-Back - CPU

- Mudança de estado considerando requisições da CPU para cada bloco de cache
- Blocos que não estão na cache não são validos

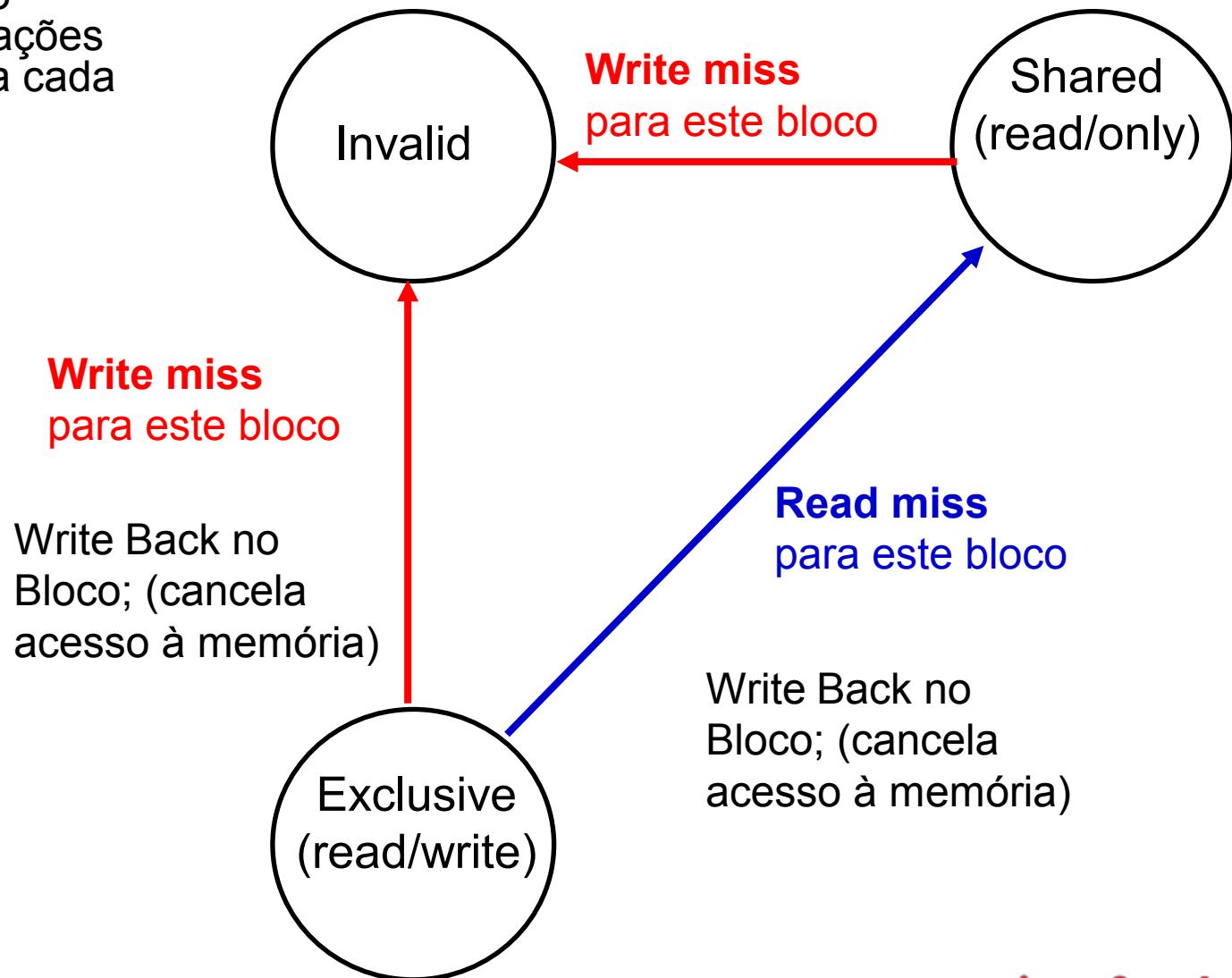


# Snooping: Write-Back - Substituição de Bloco



# Snooping: Write-Back - Barramento

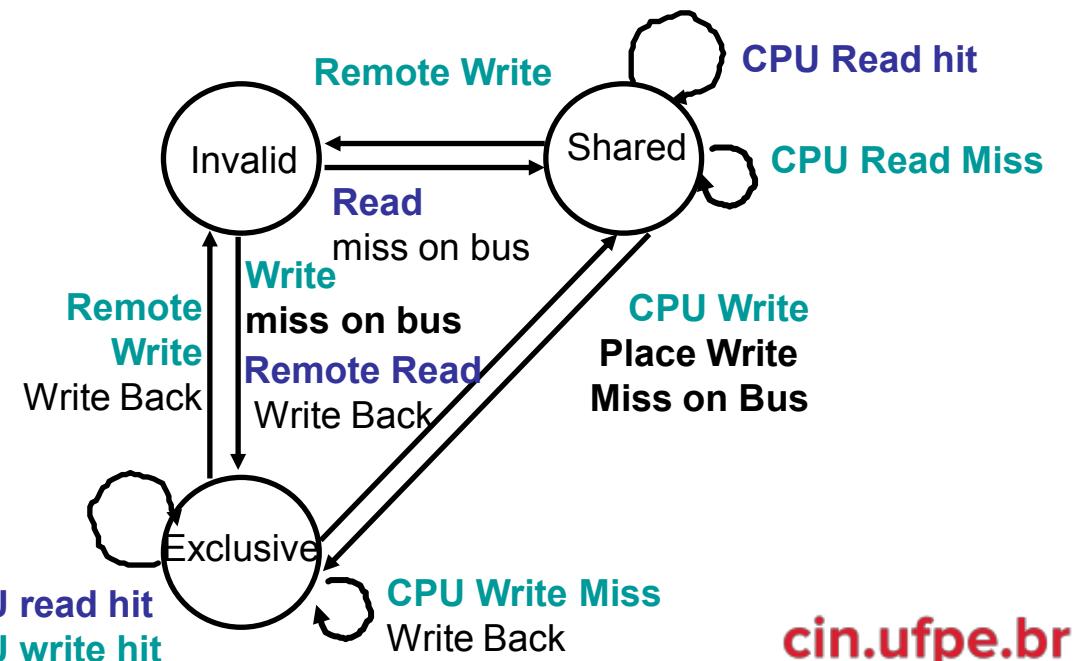
Mudança de estado  
considerando operações  
do barramento para cada  
bloco de cache



# Exemplo:

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

- Assuma que estado inicial da cache é “não válido”
- A1 e A2 mapeiam para o mesmo slot de cache mas A1  $\neq$  A2

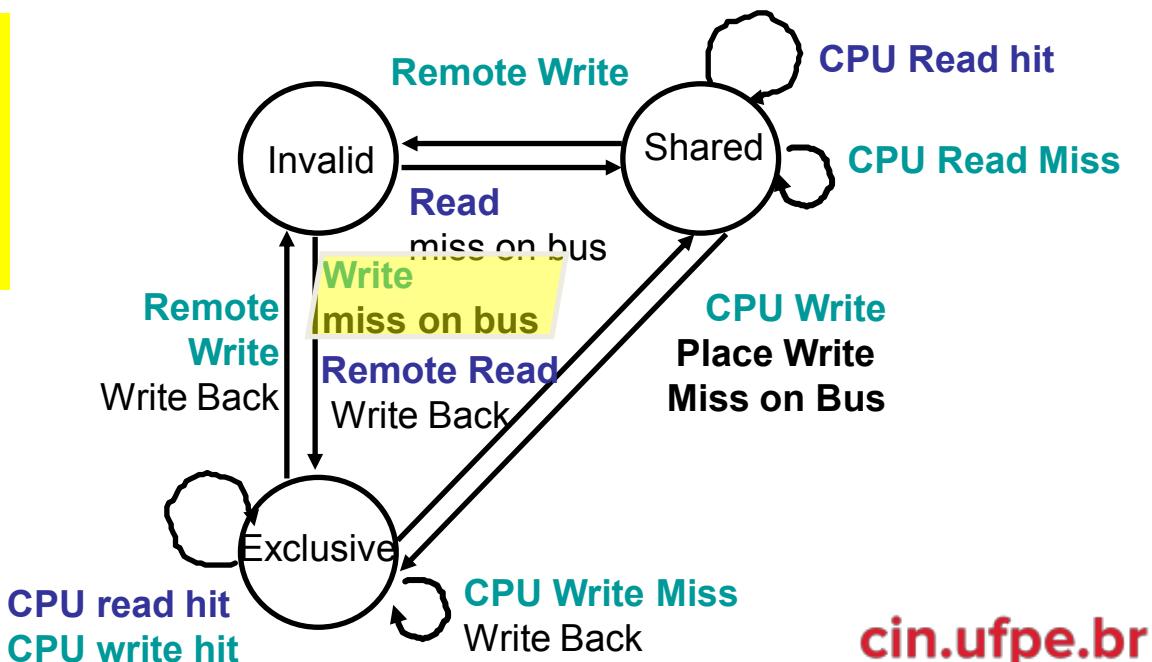


# Exemplo: Passo 1

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

- Assuma que estado inicial da cache é “não válido”
- A1 e A2 mapeiam para o mesmo slot de cache mas A1  $\neq$  A2

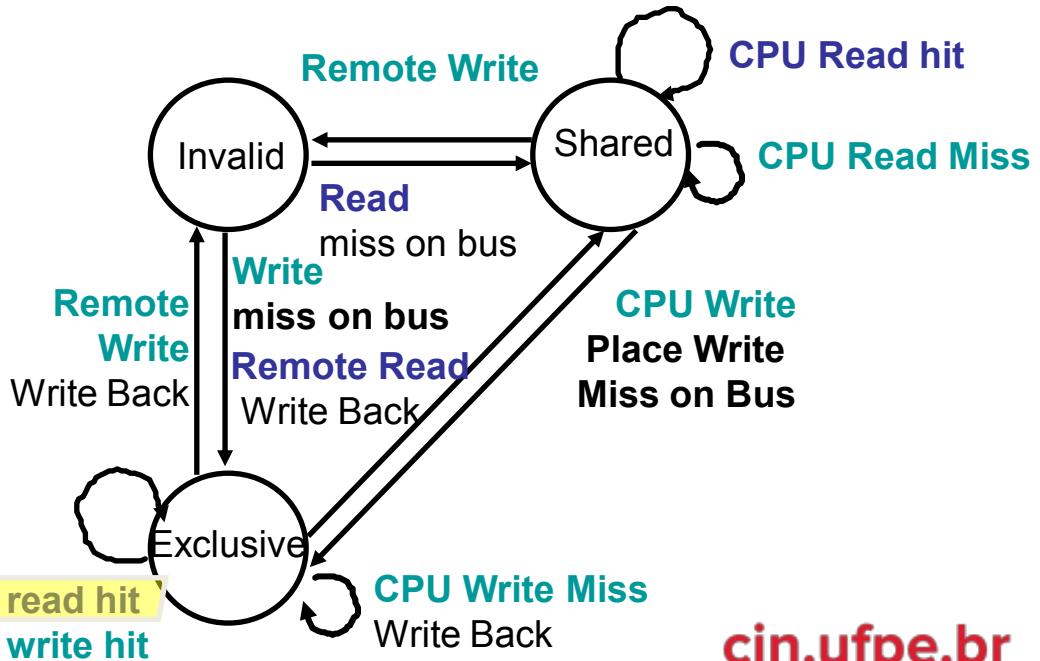
Estado ativo:



# Exemplo: Passo 2

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

- Assuma que estado inicial da cache é “não válido”
- A1 e A2 mapeiam para o mesmo slot de cache mas  $A1 \neq A2$



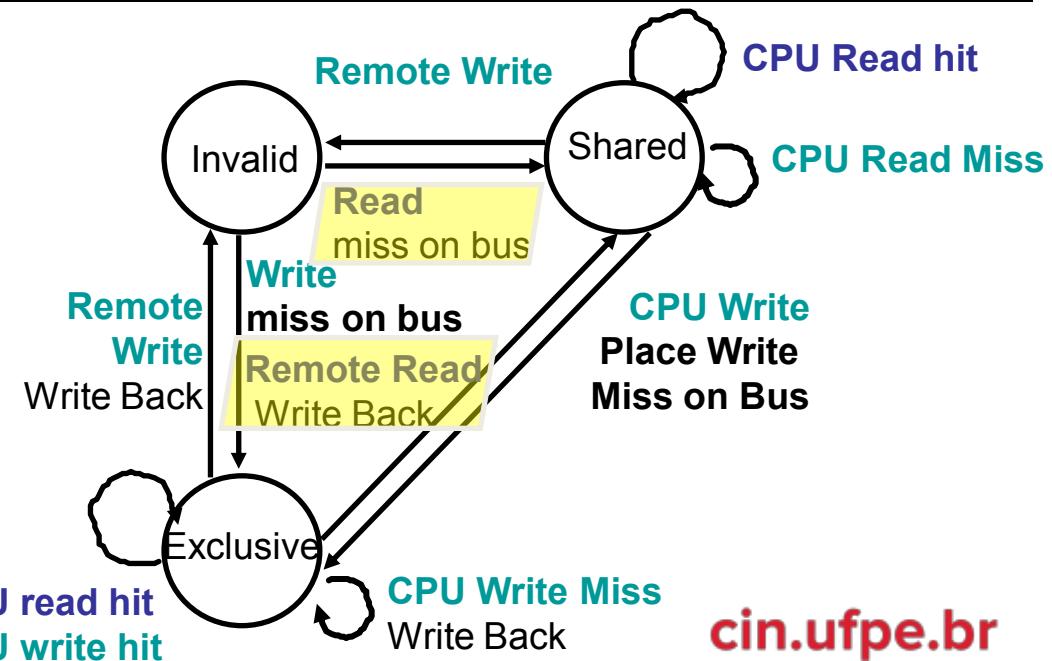
Estado ativo:

CPU read hit  
CPU write hit

# Exemplo: Passo 3

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1		A1	
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10		10
P2: Write 20 to A1												10
P2: Write 40 to A2												10
												10

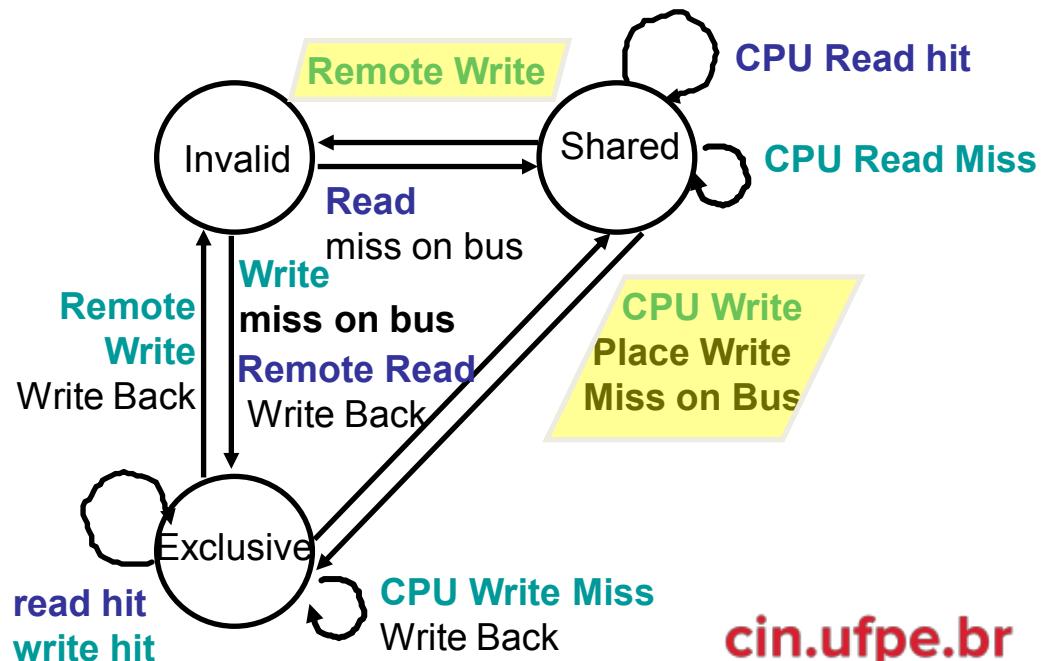
- Assuma que estado inicial da cache é “não válido”
- A1 e A2 mapeiam para o mesmo slot de cache mas  $A1 \neq A2$



# Exemplo: Passo 4

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1			10
P2: Write 40 to A2												10
												10

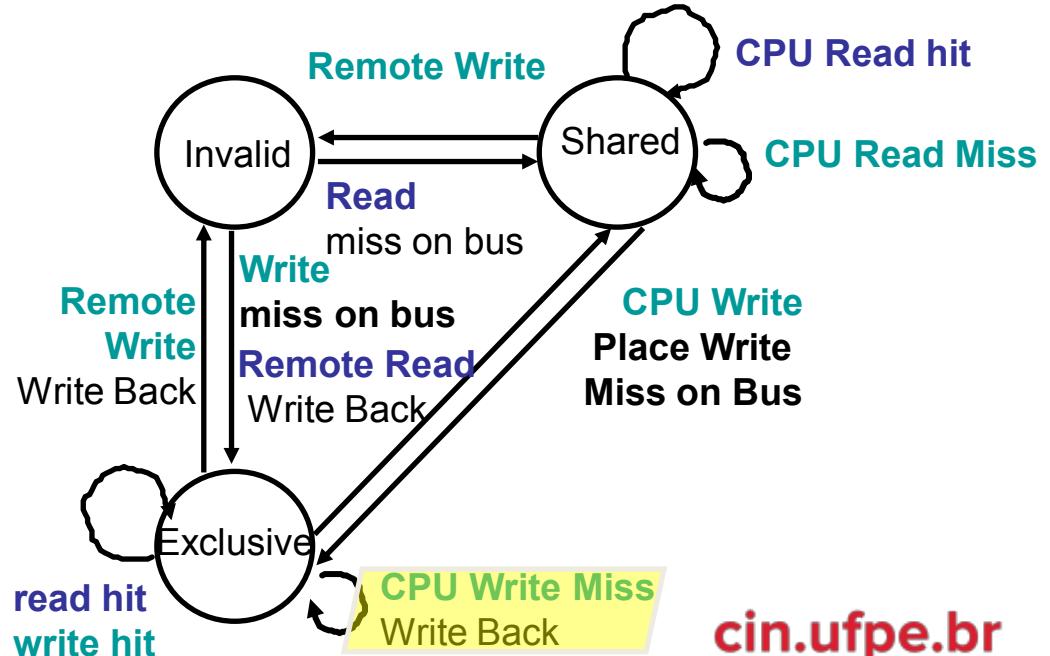
- Assuma que estado inicial da cache é “não válido”
- A1 e A2 mapeiam para o mesmo slot de cache mas  $A1 \neq A2$



# Exemplo: Passo 5

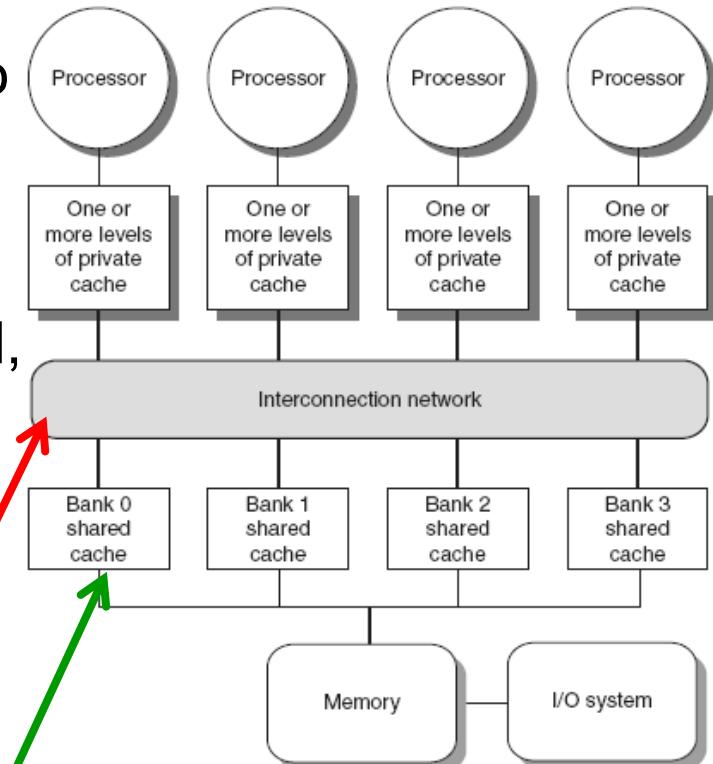
	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									A1
P2: Read A1				<u>Shar.</u>	A1		RdMs	P2	A1			A1
	<u>Shar.</u>	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	<u>Inv.</u>			<u>Excl.</u>	A1	20	WrMs	P2	A1			A1
P2: Write 40 to A2							WrMs	P2	A2			10
				Excl.	A2	40	WrBk	P2	A1	20		20

- Assuma que estado inicial da cache é “não válido”
- A1 e A2 mapeiam para o mesmo slot de cache mas  $A1 \neq A2$



# Limitações de Protocolos Snooping

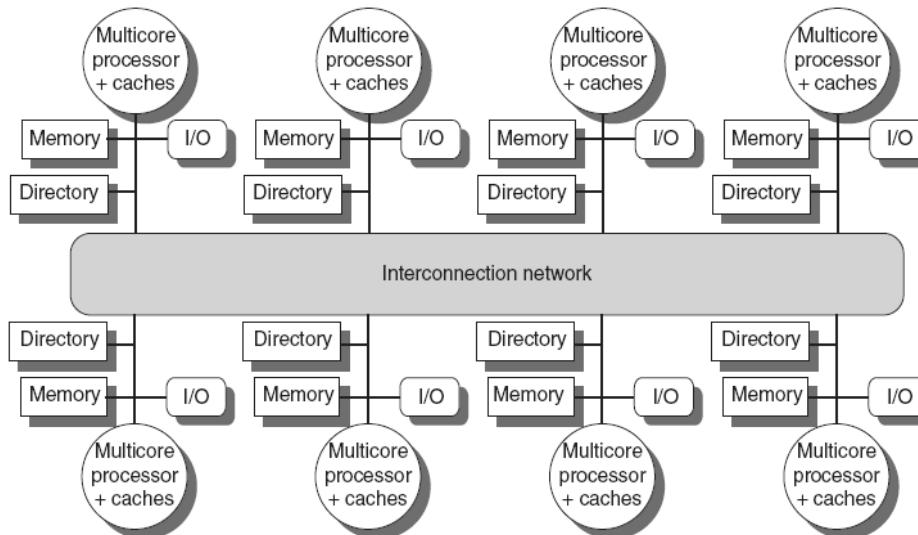
- Aumento no número de processadores provoca crescimento na demanda por memória
  - Se memória única, cria-se um gargalo
- Para multiprocessadores bus-based, a quantidade de acessos devido a transações de coerência aumenta
  - Limitando a largura de banda e também se tornando um gargalo
- Soluções:
  - Múltiplos barramentos e **redes de interconexão**
  - Memória configurada em **bancos de memória**
  - OU:



**Diretórios**

# Diretório

- Diretório mantém informações sobre cada bloco
  - Quais caches tem quais blocos
  - Estado do bloco
  - Em caso de miss, encontra entrada no diretório, analisa o mesmo e comunica somente com os nós que possuem cópia
- Pode ser centralizado
  - Implementado por exemplo numa cache compartilhada L3
- Ou ainda implementada de forma distribuída
  - Comunicação se dá através de rede de interconexões



# Mais Sobre Diretórios

- Para cada bloco, mantém estado:
  - **Shared**
    - Uma ou mais caches possuem o bloco e o valor atualizado está na memória
    - Lista de IDs das caches que possuem o bloco
  - **Uncached**
    - Está apenas na memória
  - **Exclusive/Modified**
    - Apenas uma cache possui o bloco e o valor na memória está desatualizado
    - ID da cache que possui o bloco atualizado
- Diretório mantém os estados dos blocos e envia mensagens de invalidação

# Protocolo Baseado em Diretório (1)

- Para um bloco que NÃO está na cache (uncached):
  - Read miss
    1. Nó que fez a requisição recebe o bloco
    2. Nó marcado no diretório como o único nó que possui o bloco **compartilhado (shared)**
  - Write miss
    - Nó que fez a requisição recebe o bloco
    - Nó marcado no diretório como o único nó que possui o bloco e este bloco é **exclusivo**
- Para bloco **compartilhado**:
  - Read miss
    - Nó que fez a requisição recebe o bloco
    - Nó entra na lista de nós que possuem o bloco **compartilhado**
  - Write miss
    - Nó que fez a requisição recebe o bloco
    - Todos os demais nós recebem uma mensagem de invalidação,
    - Lista de nós que possuem o bloco é composta somente por este nó, e o bloco agora é marcado como **exclusivo**

# Protocolo Baseado em Diretório (2)

## ■ Para bloco exclusivo:

- Read miss

- Nó dono do bloco recebe uma requisição

- Bloco se torna compartilhado,

- Bloco é enviado para a memória (diretório)

- Lista de nós que compartilham o bloco possui o antigo dono e o nó que fez a requisição

- Write back no caso de uma substituição

- Bloco é enviado para a memória

- Bloco se torna “uncached”

- Lista de nós que compartilham o bloco fica vazia

- Write miss

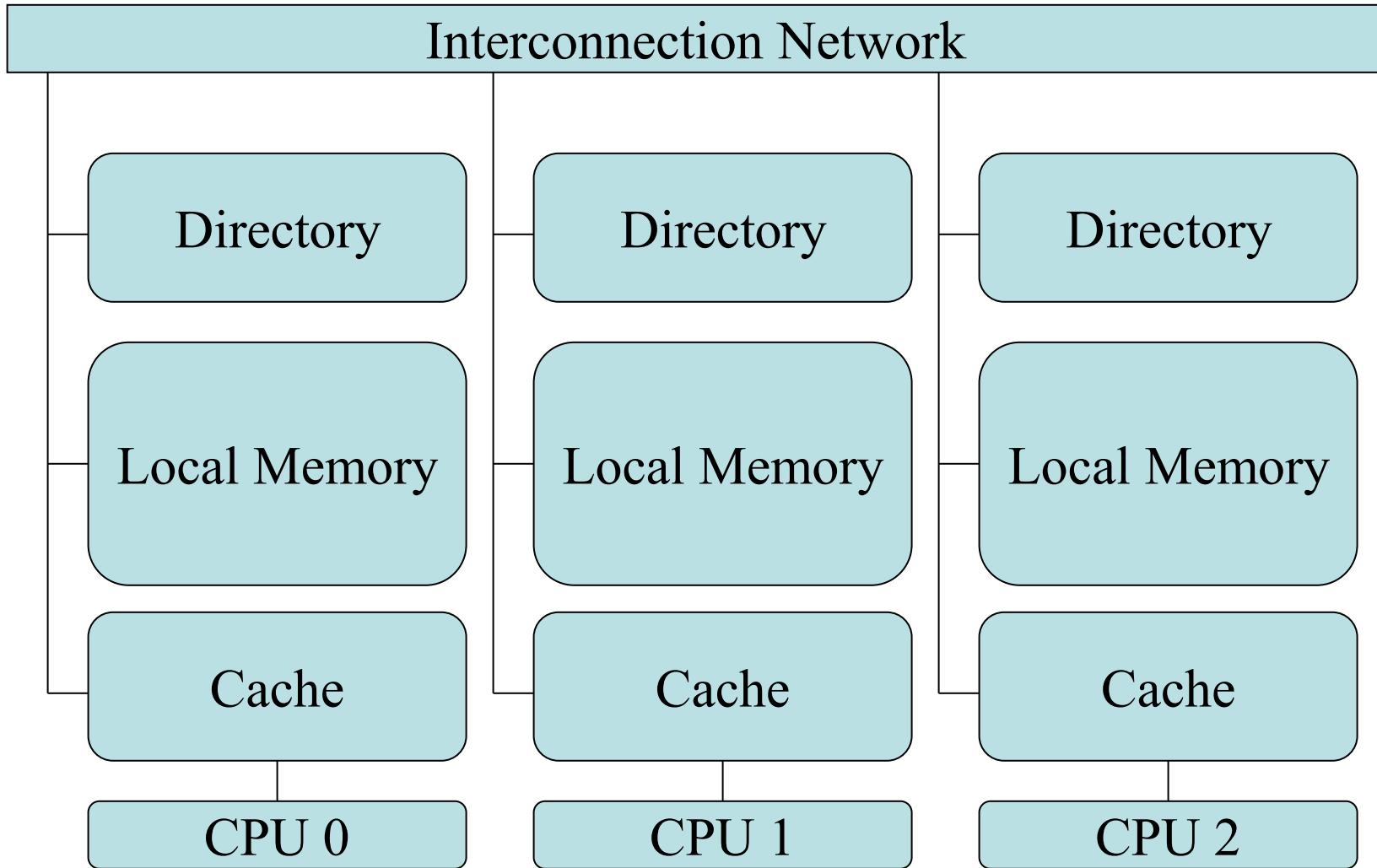
- Requisição é enviada para antigo dono

- Antigo dono manda bloco para a memória (diretório)

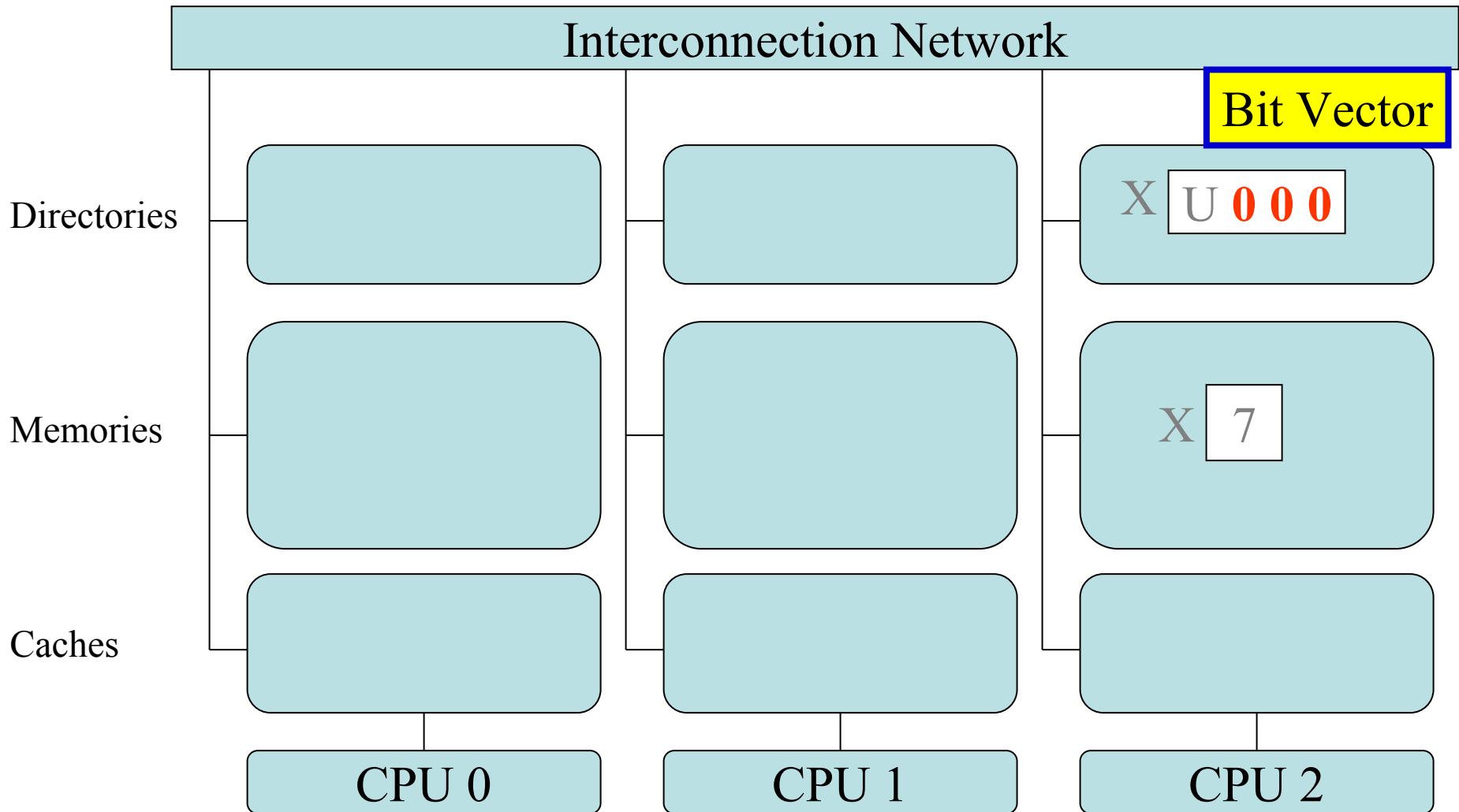
- Bloco no antigo dono é invalidado

- Nó que fez requisição se torna o novo dono e bloco continua sendo exclusivo

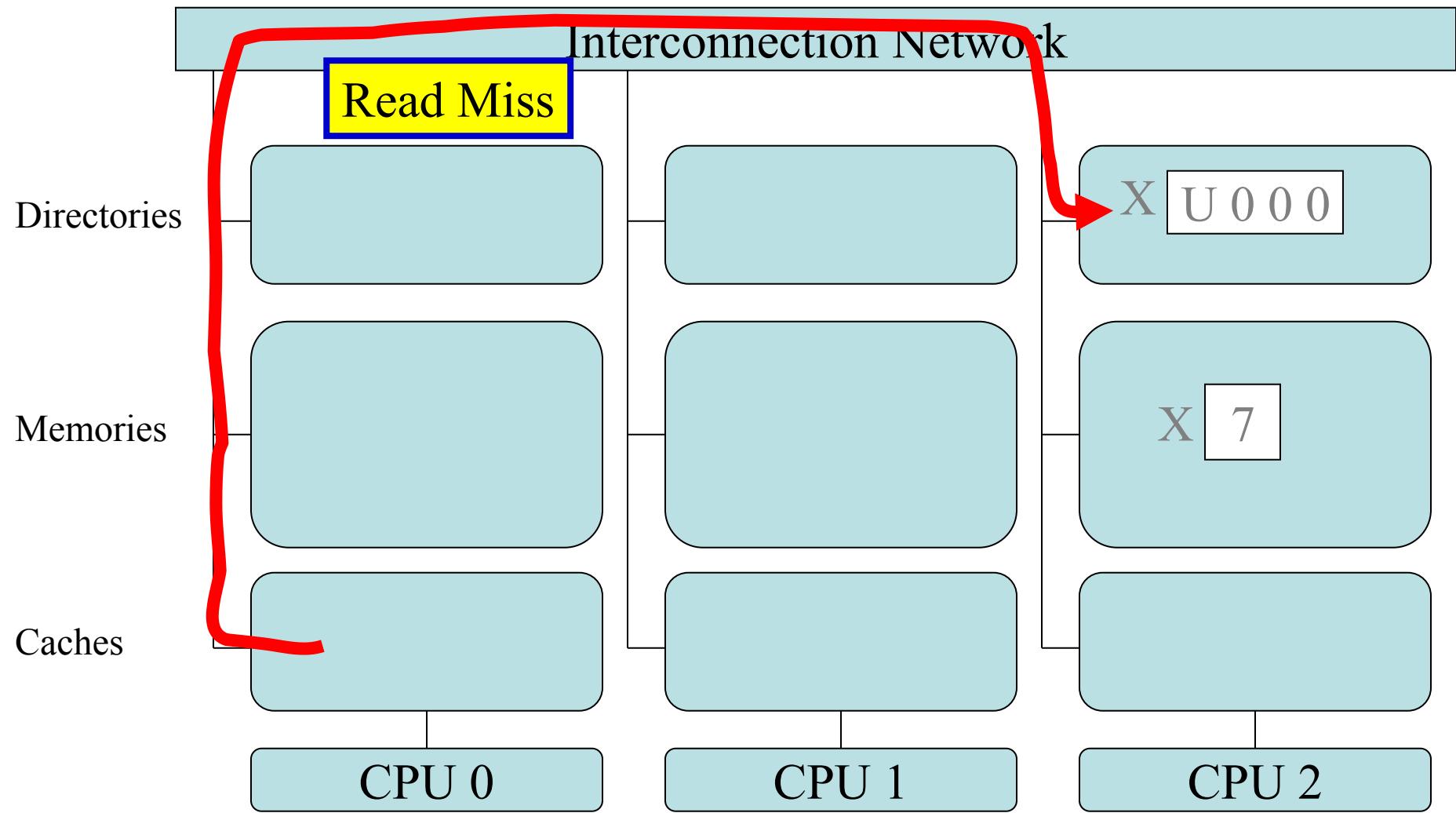
# Exemplo de Protocolo Baseado em Diretório



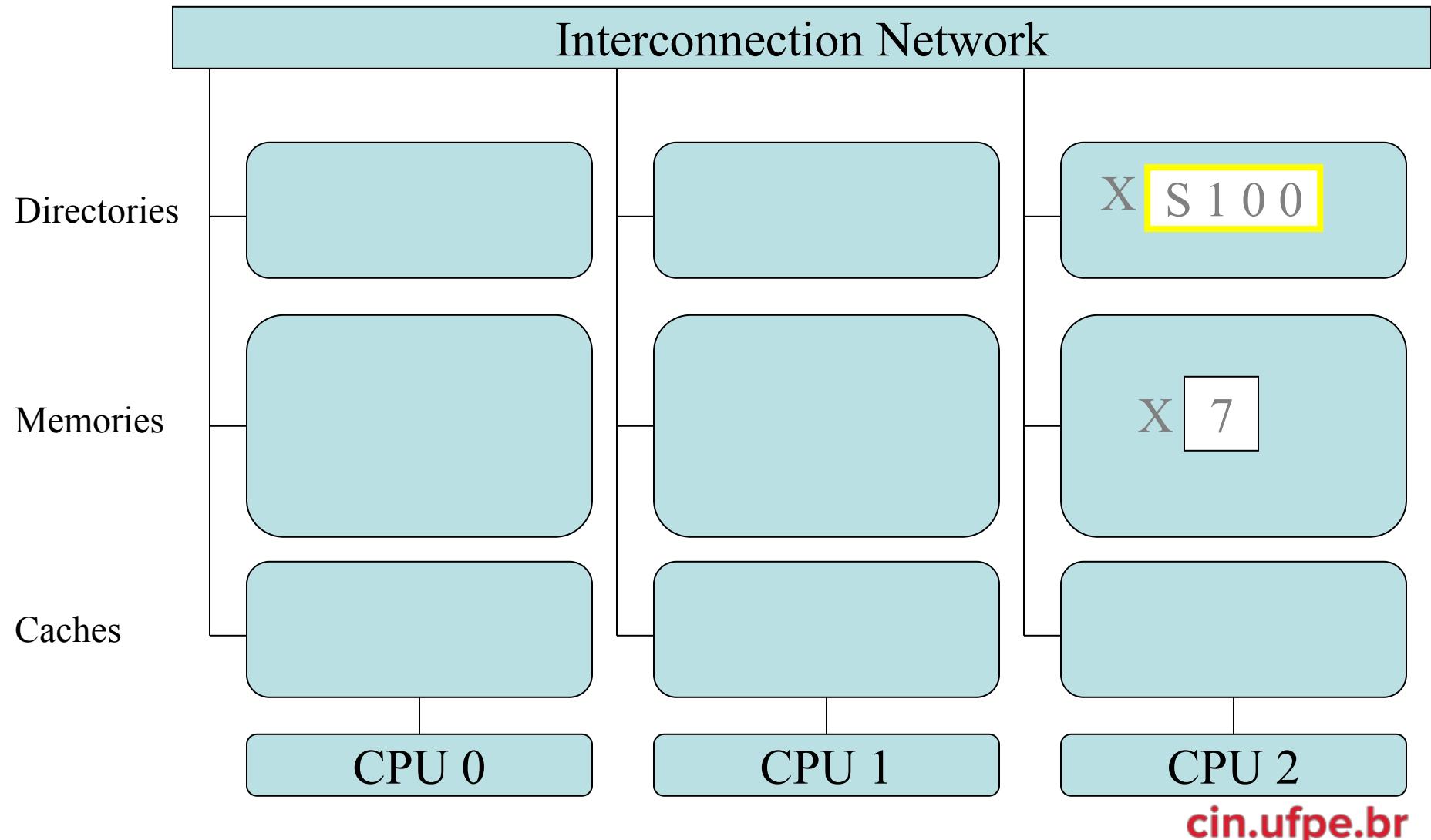
# Exemplo de Protocolo Baseado em Diretório



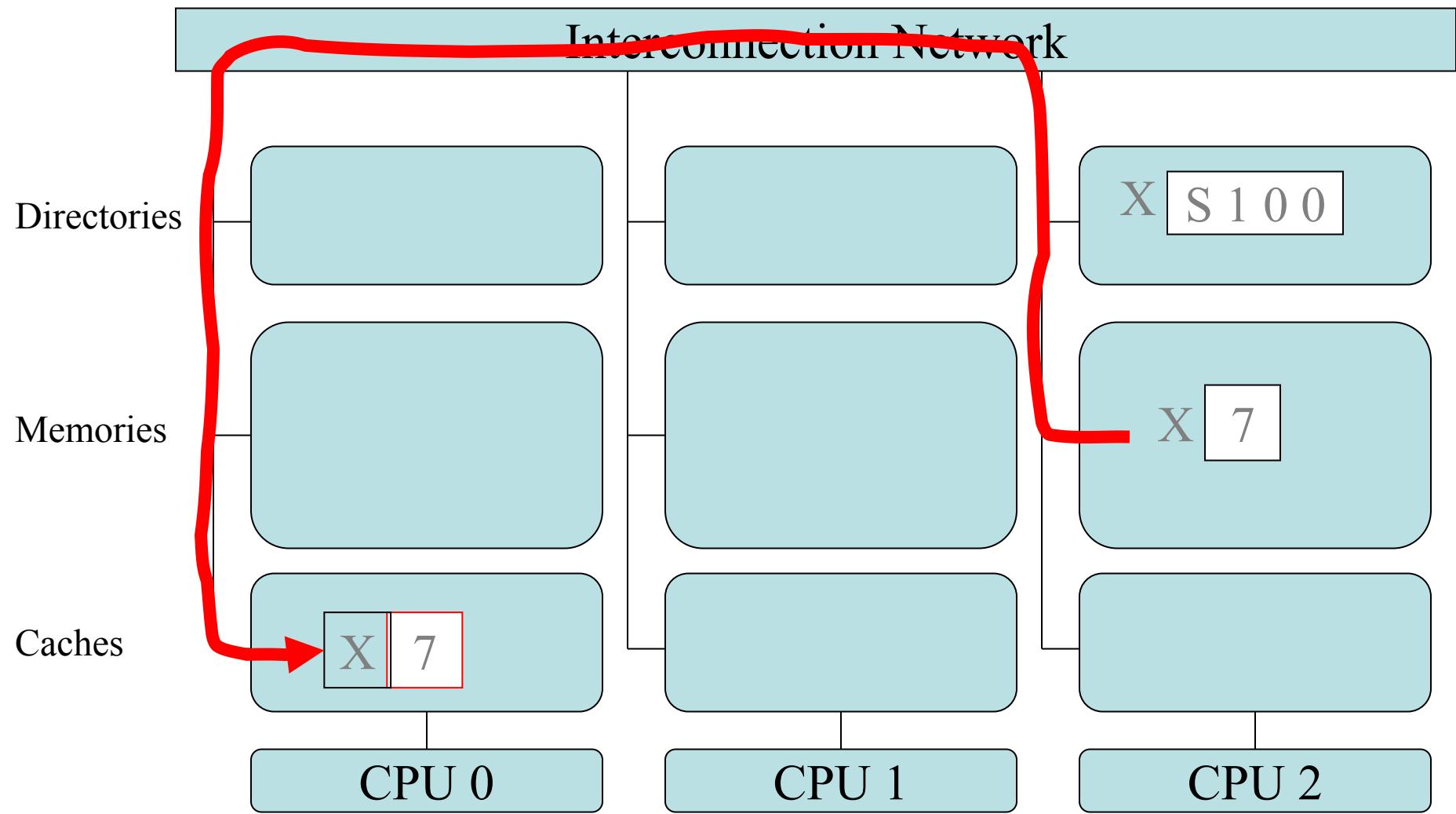
# CPU 0 lê X



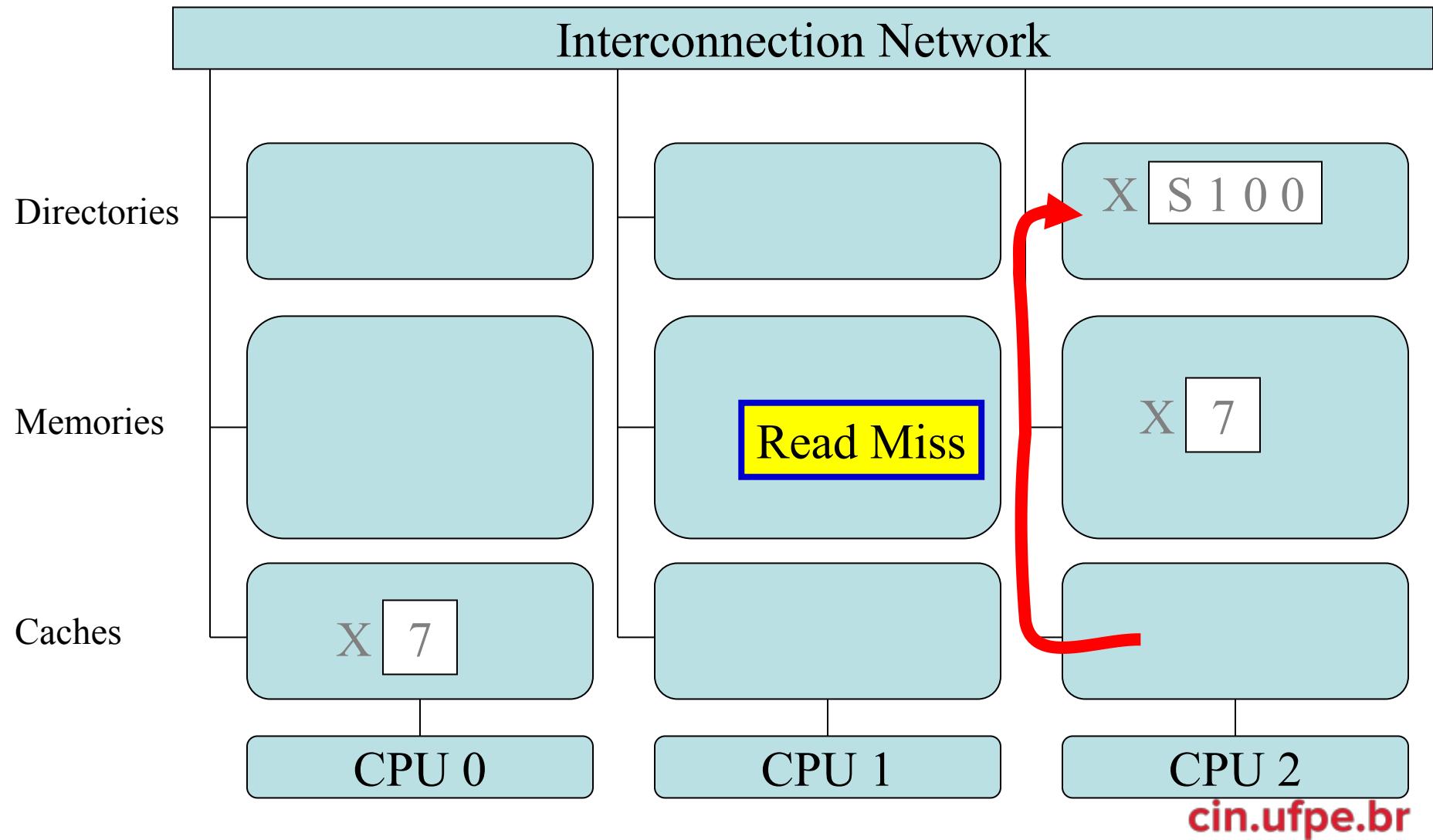
# CPU 0 lê X



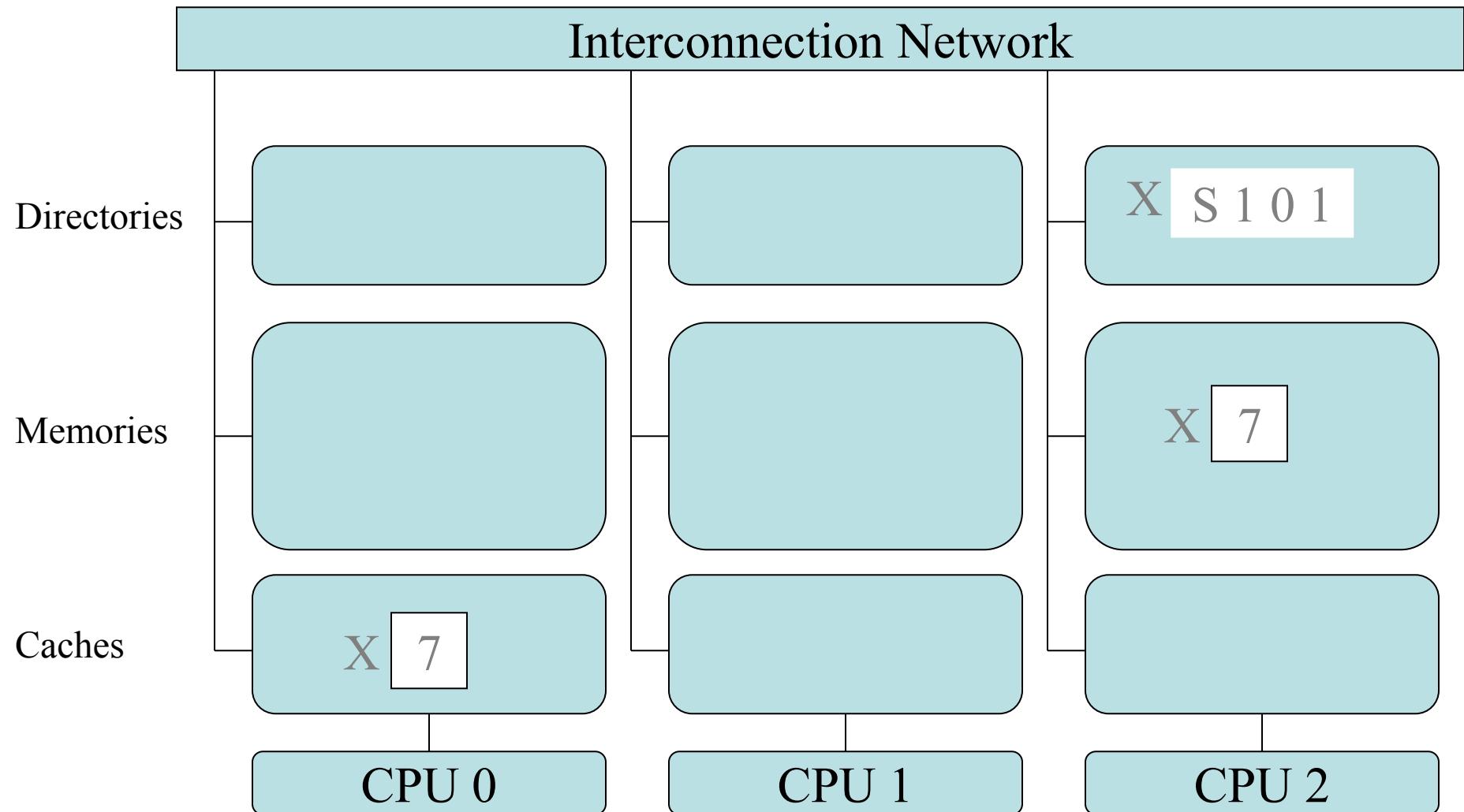
# CPU 0 lê X



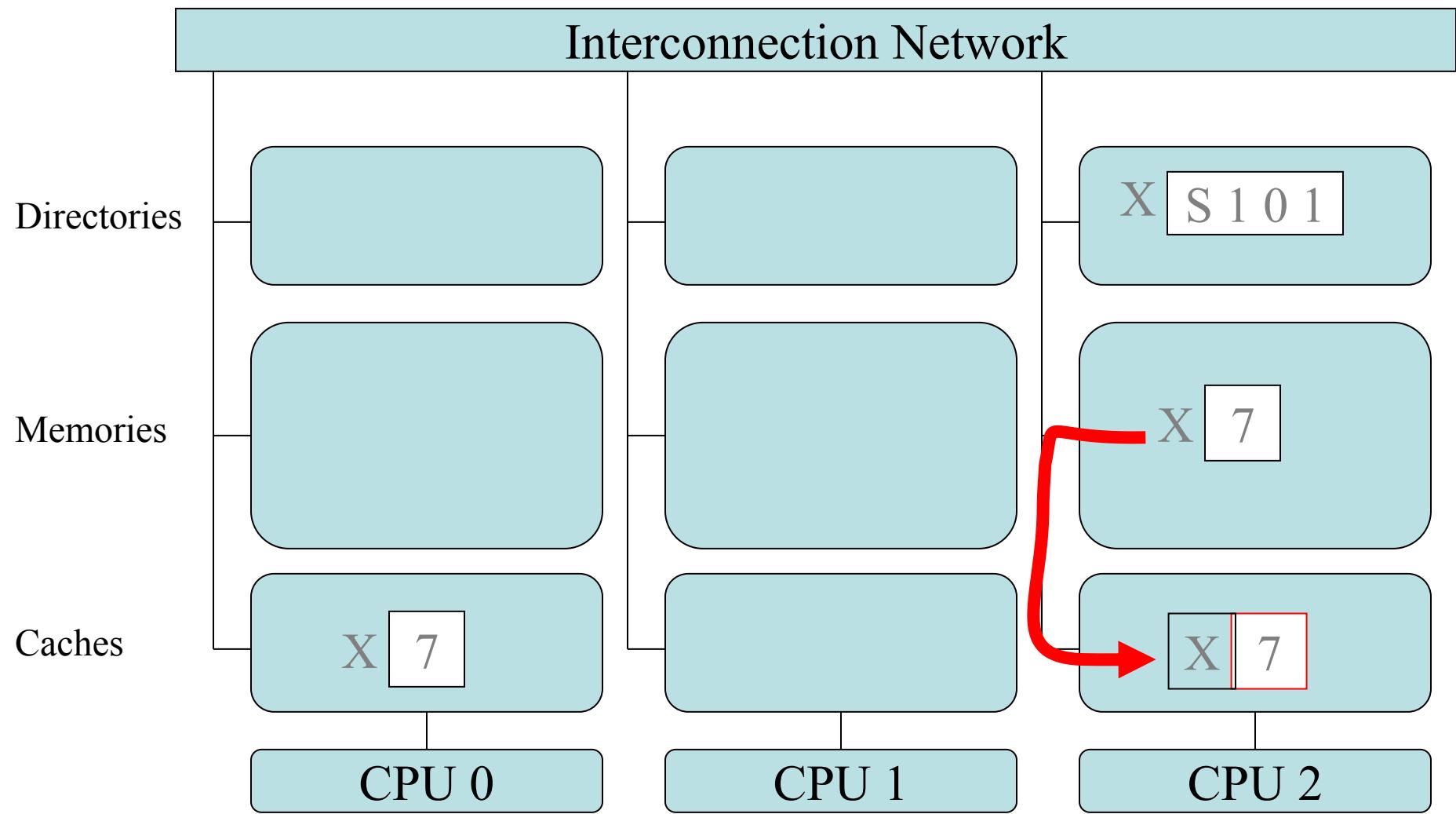
# CPU 2 lê X



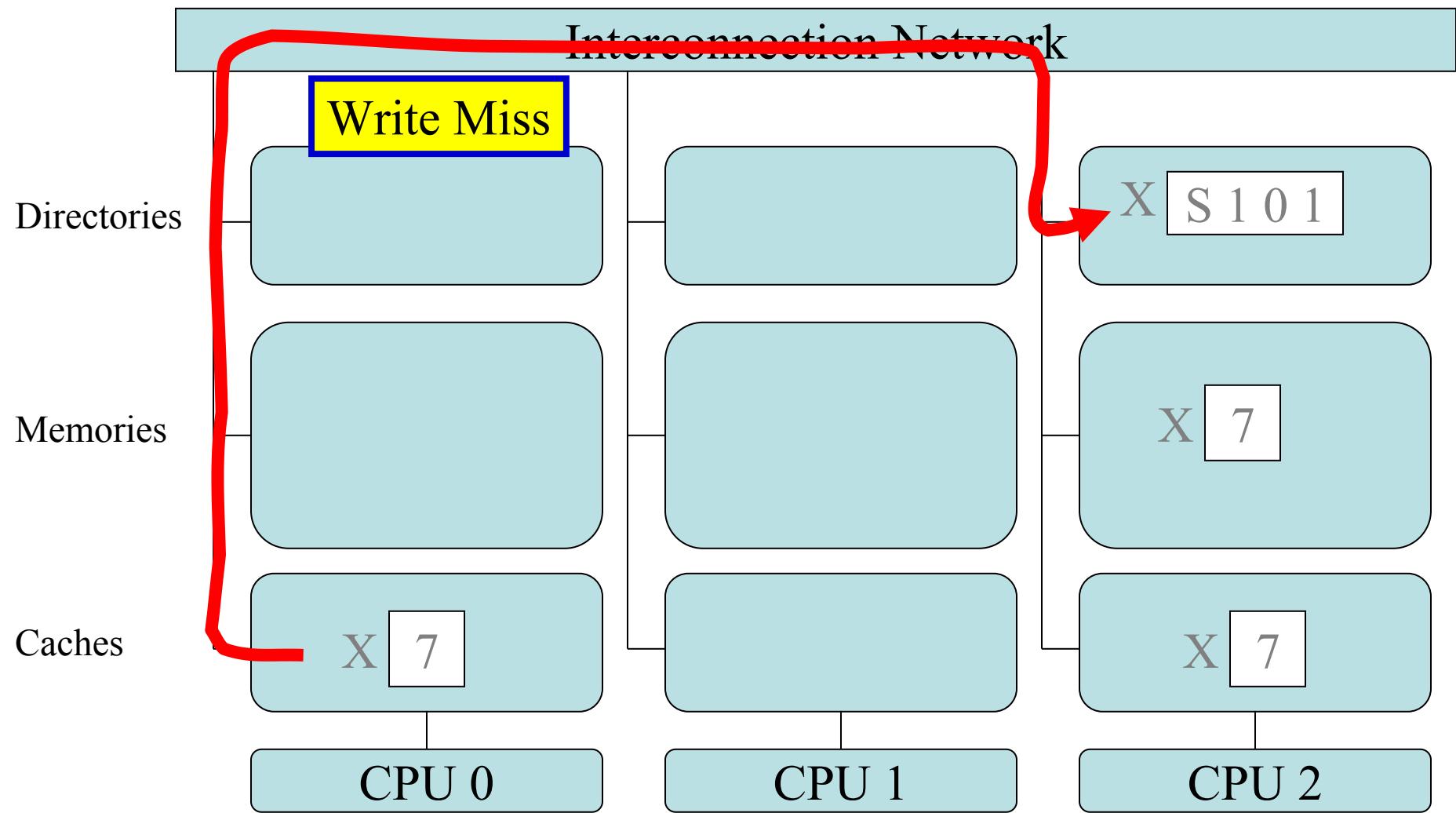
# CPU 2 lê X



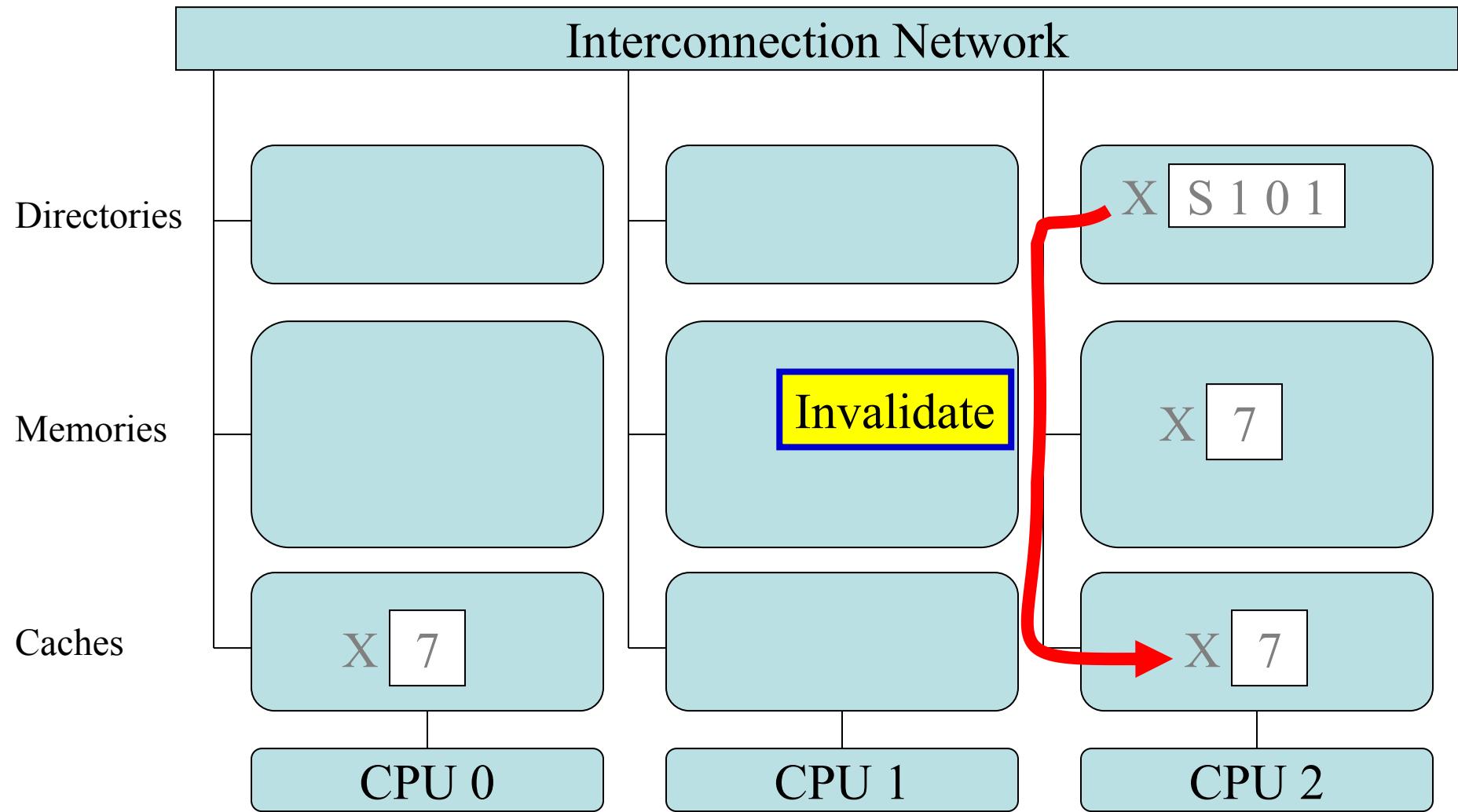
# CPU 2 lê X



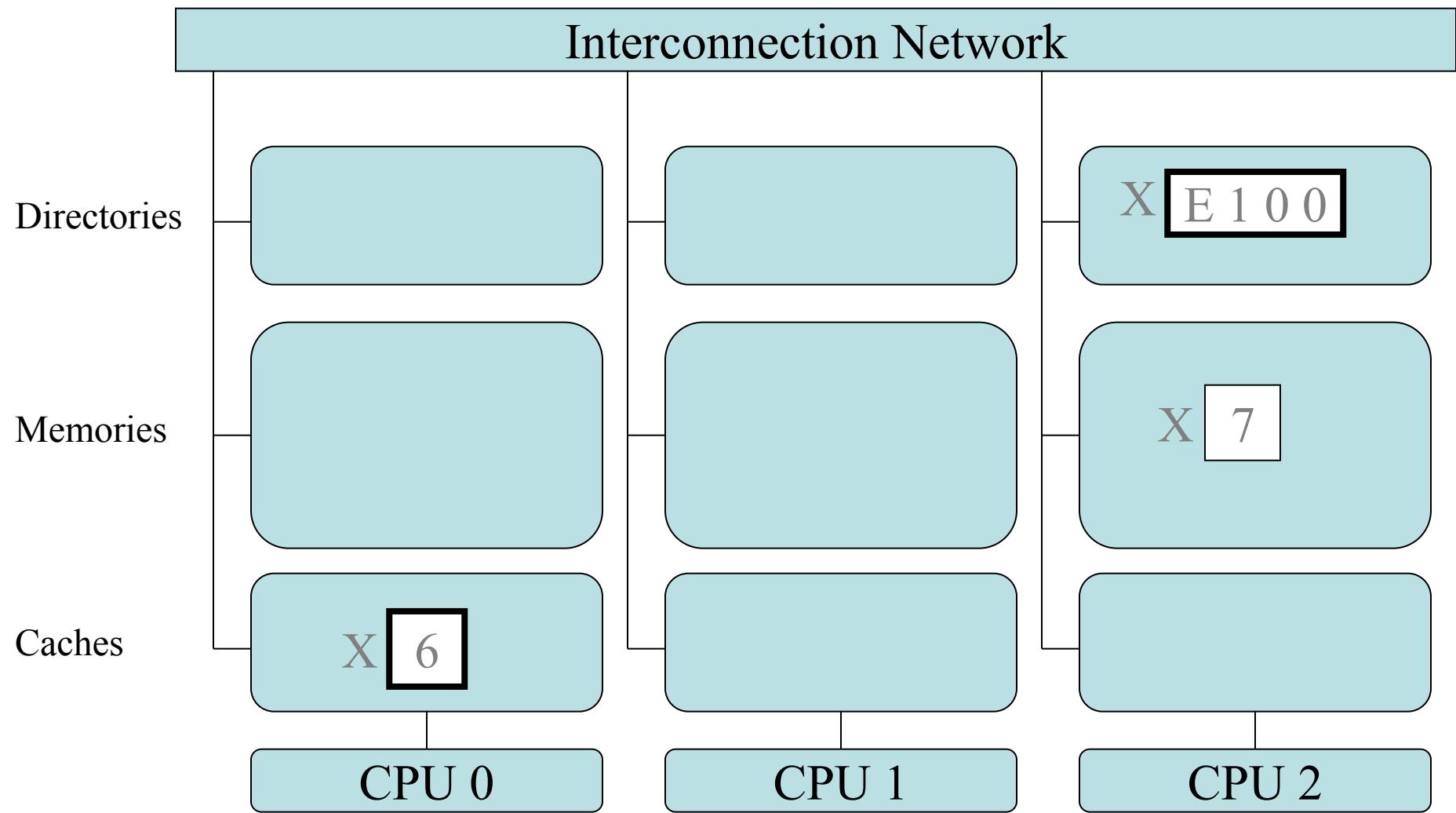
# CPU 0 escreve 6 em X



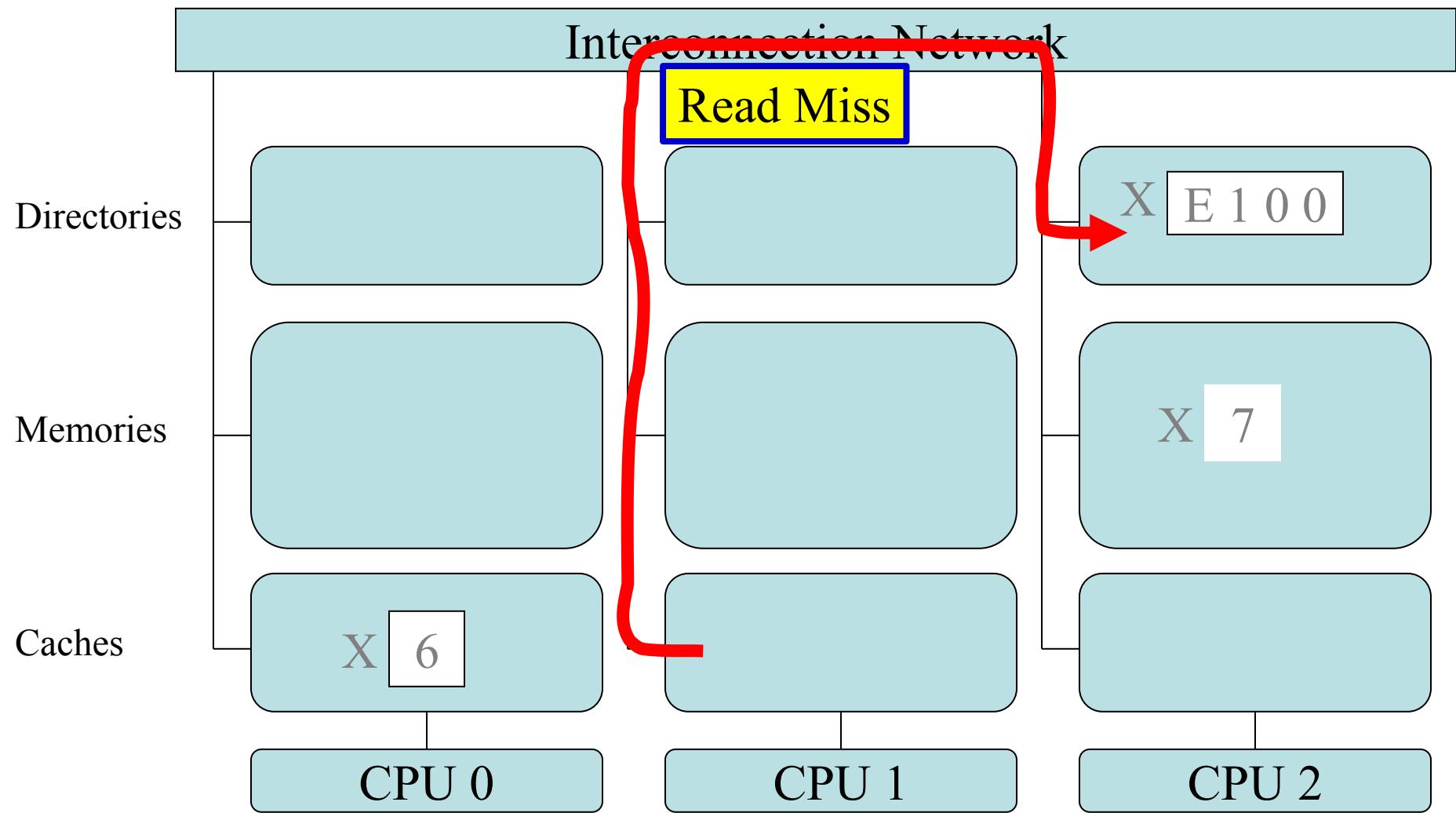
# CPU 0 escreve 6 em X



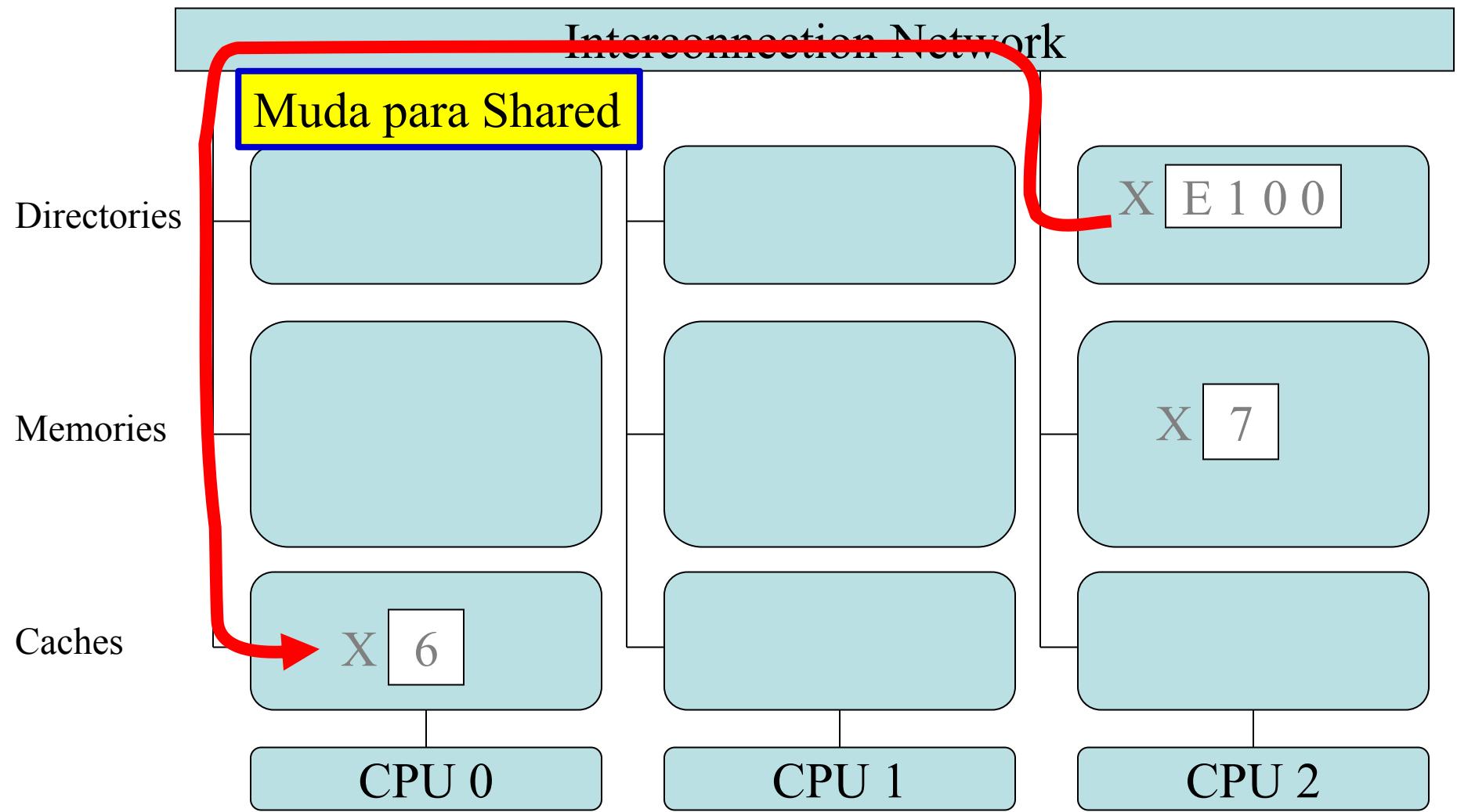
# CPU 0 escreve 6 em X



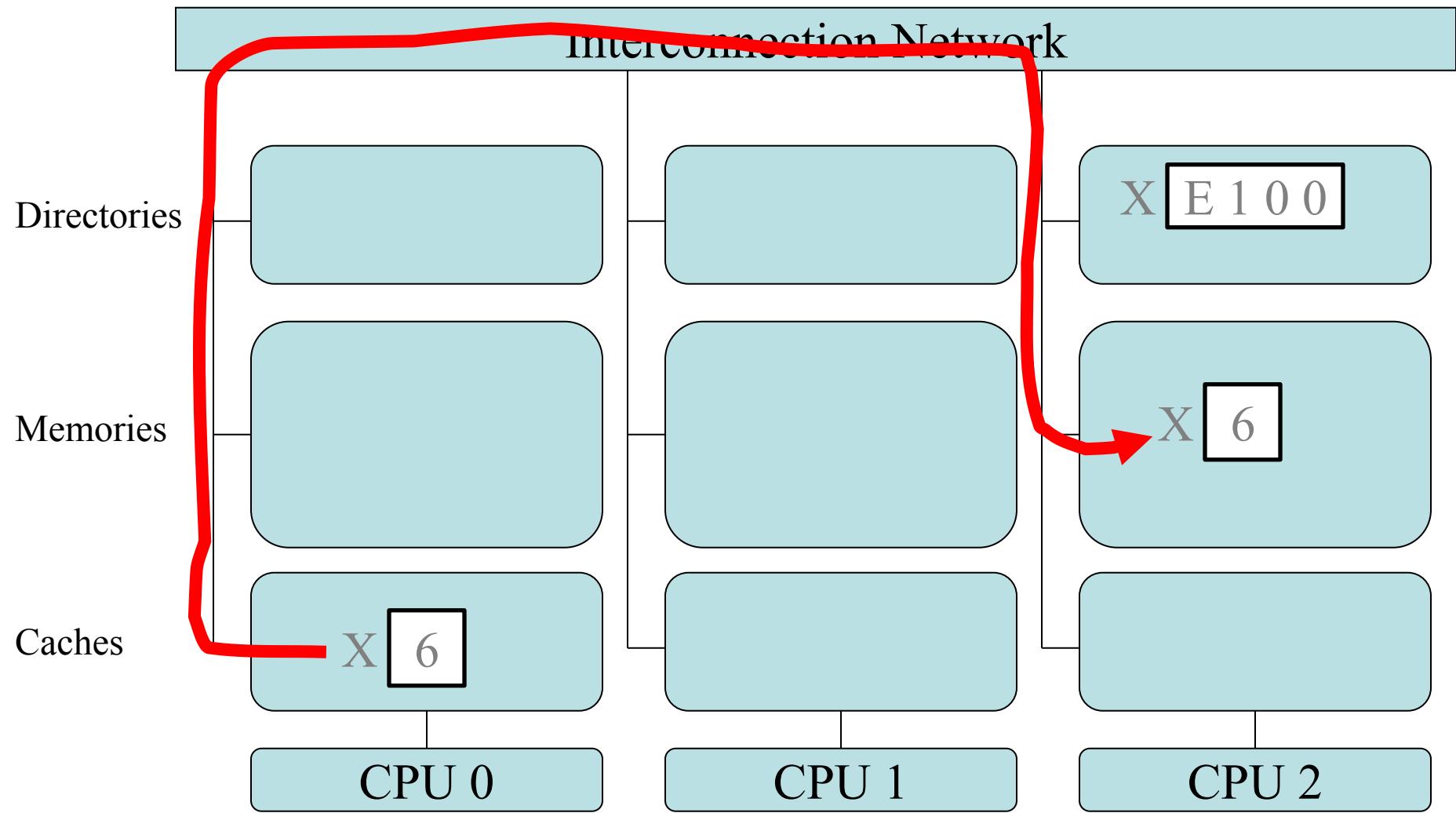
# CPU 1 lê X



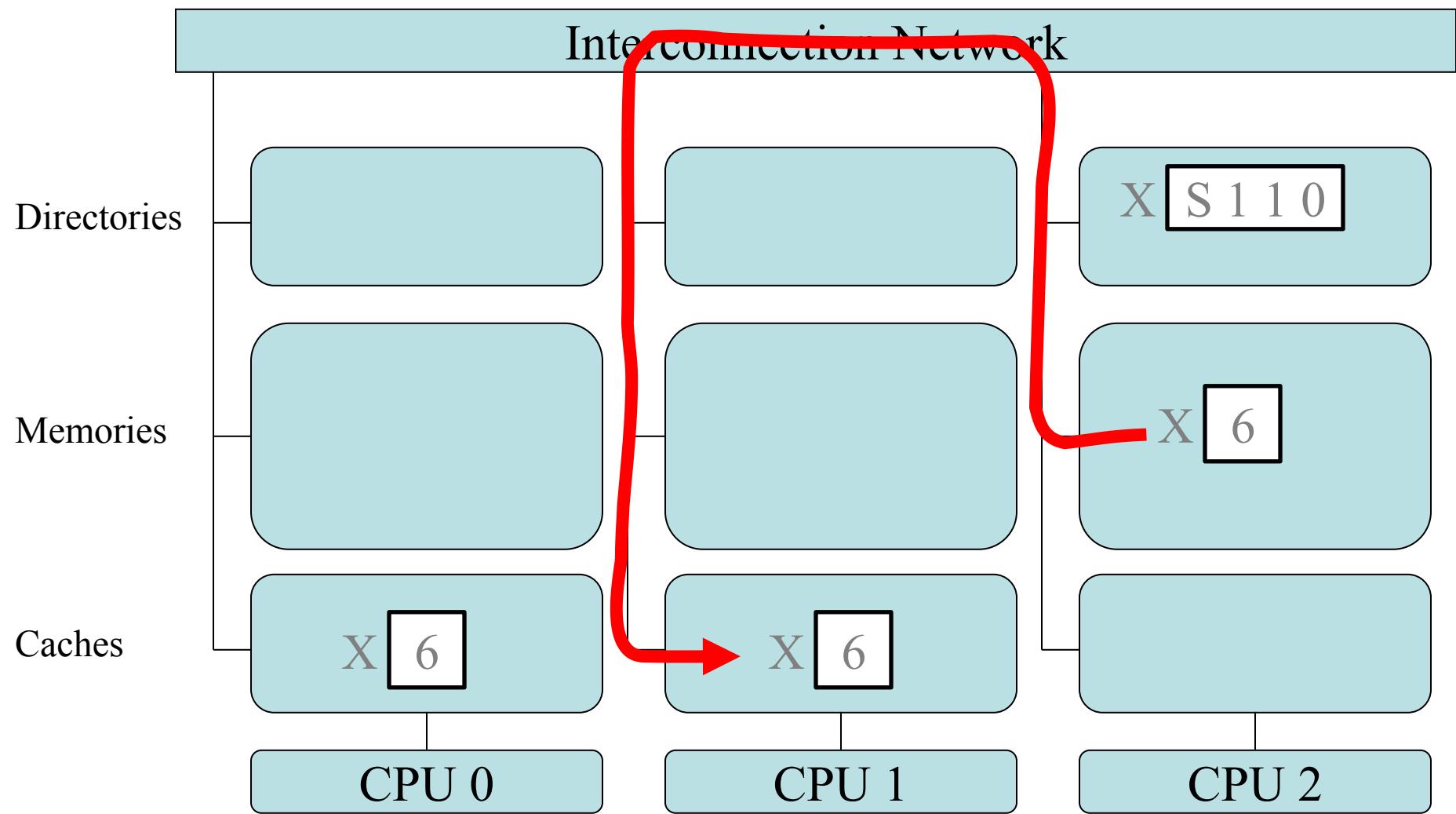
# CPU 1 lê X



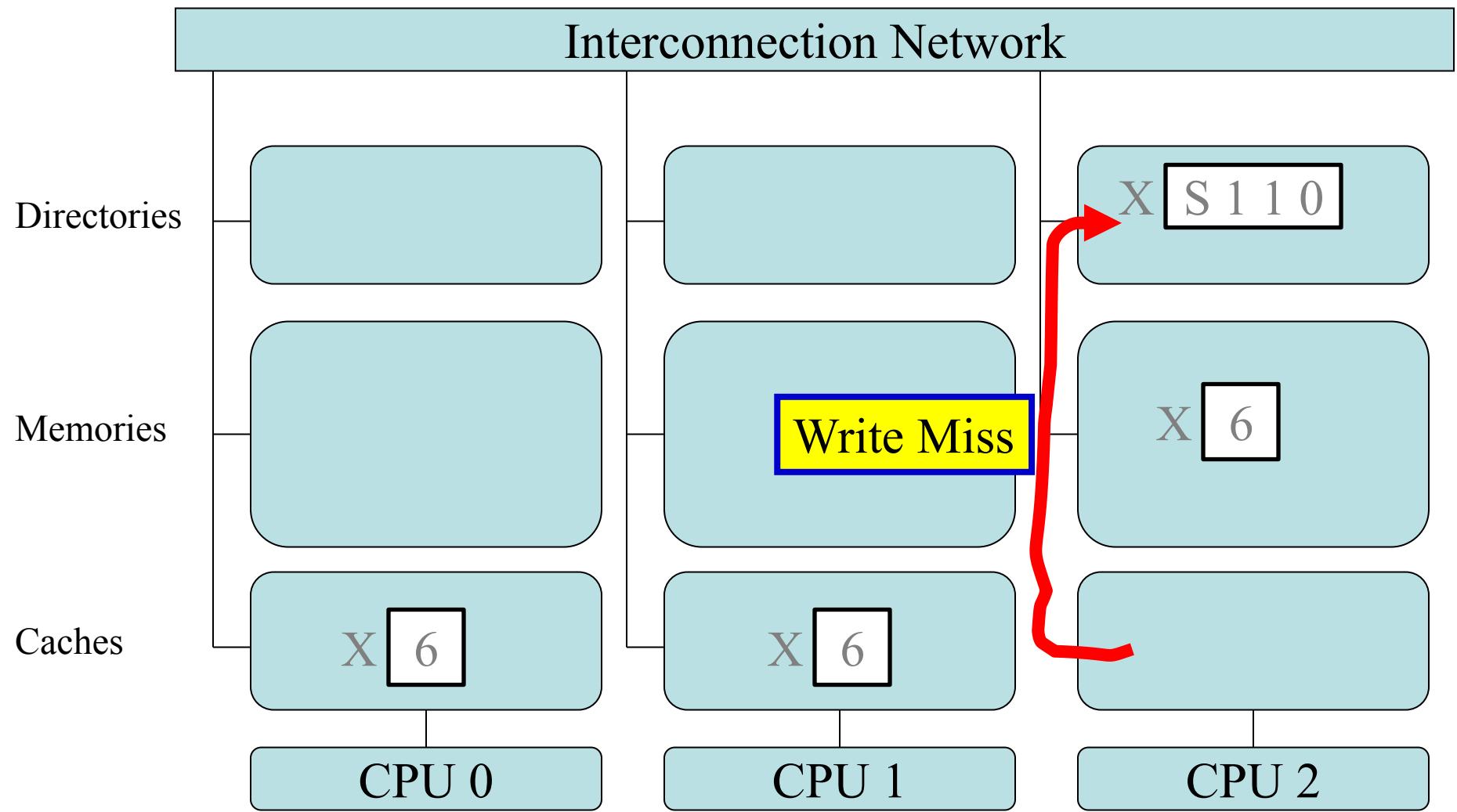
# CPU 1 lê X



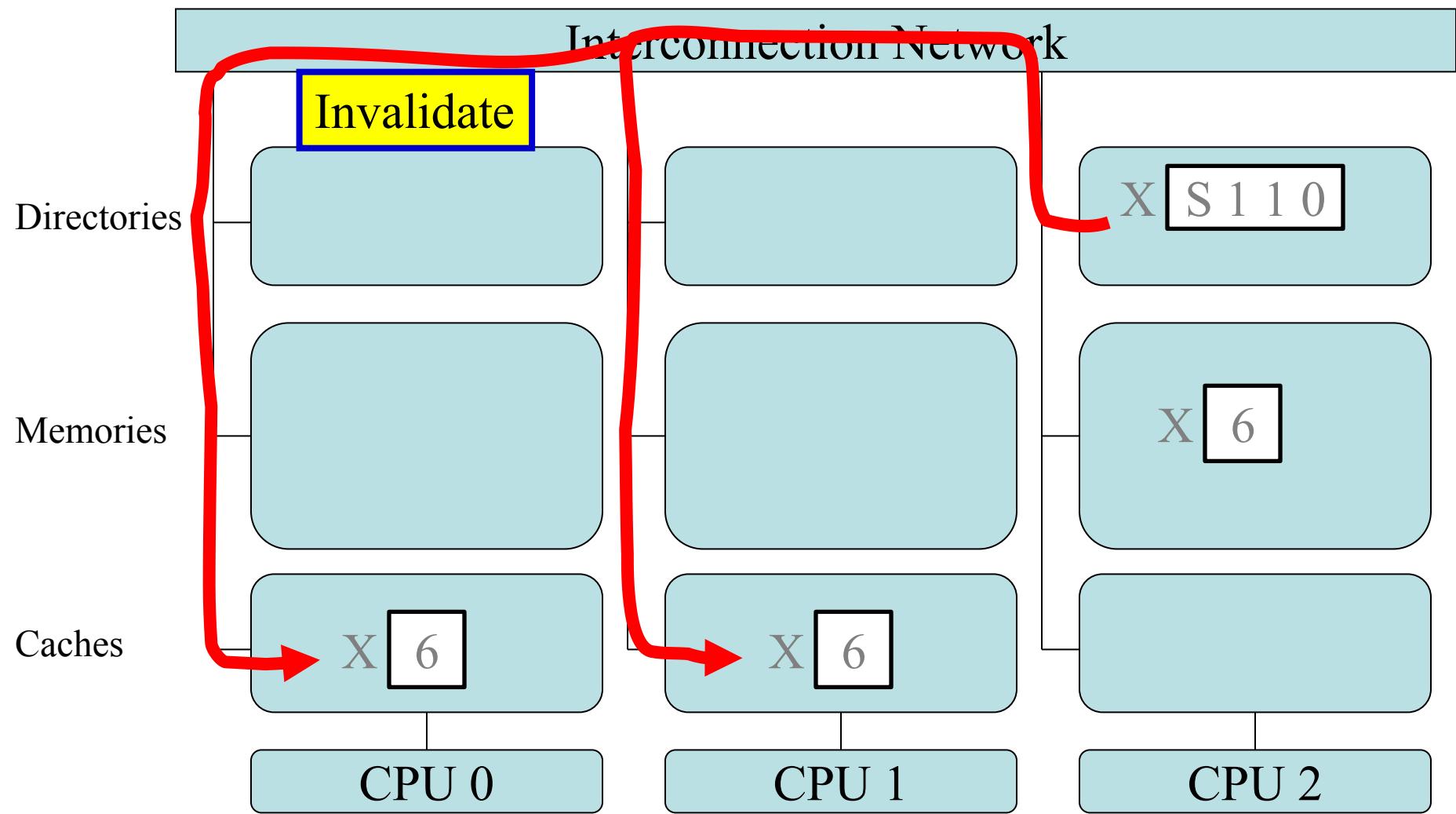
# CPU 1 lê X



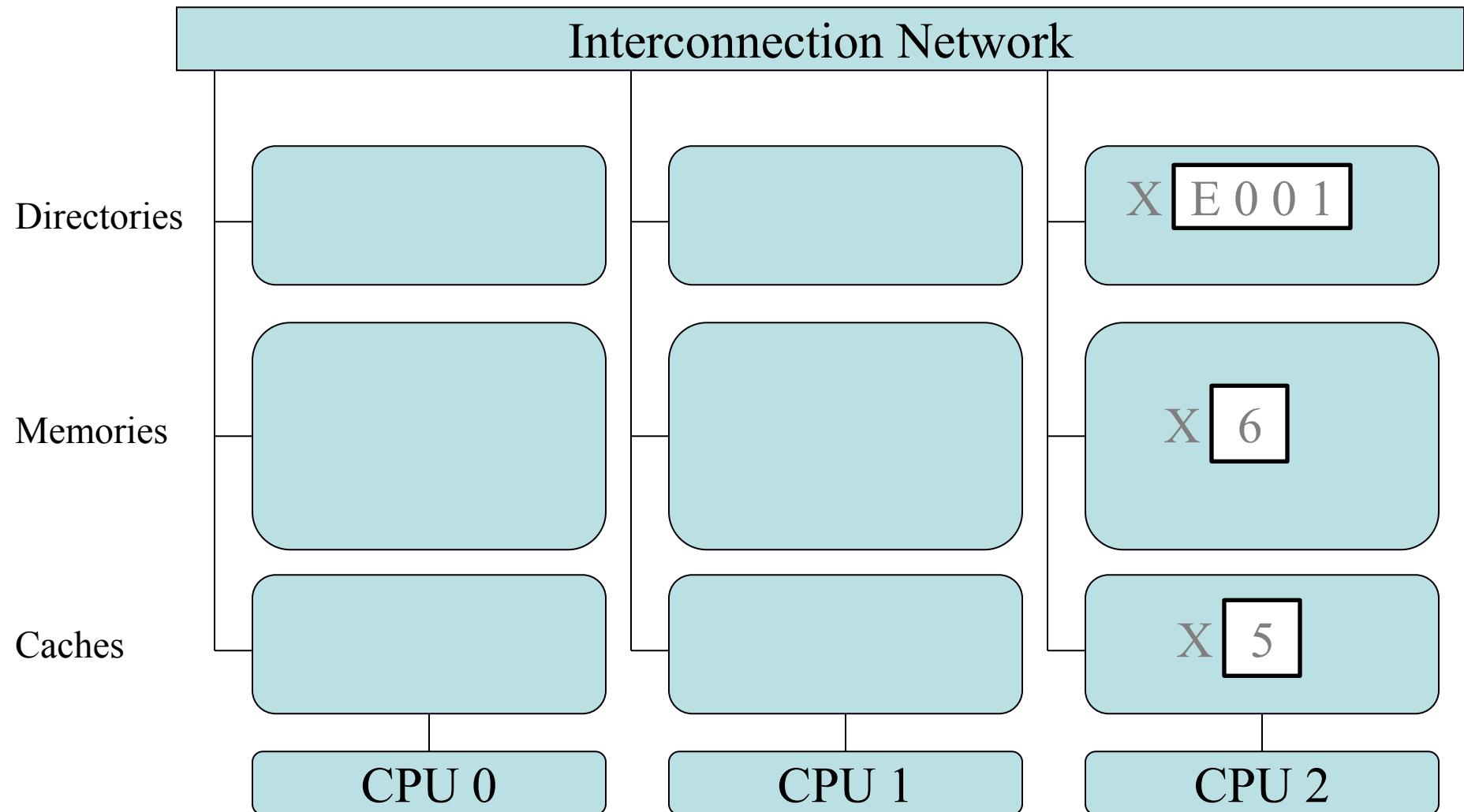
# CPU 2 escreve 5 em X



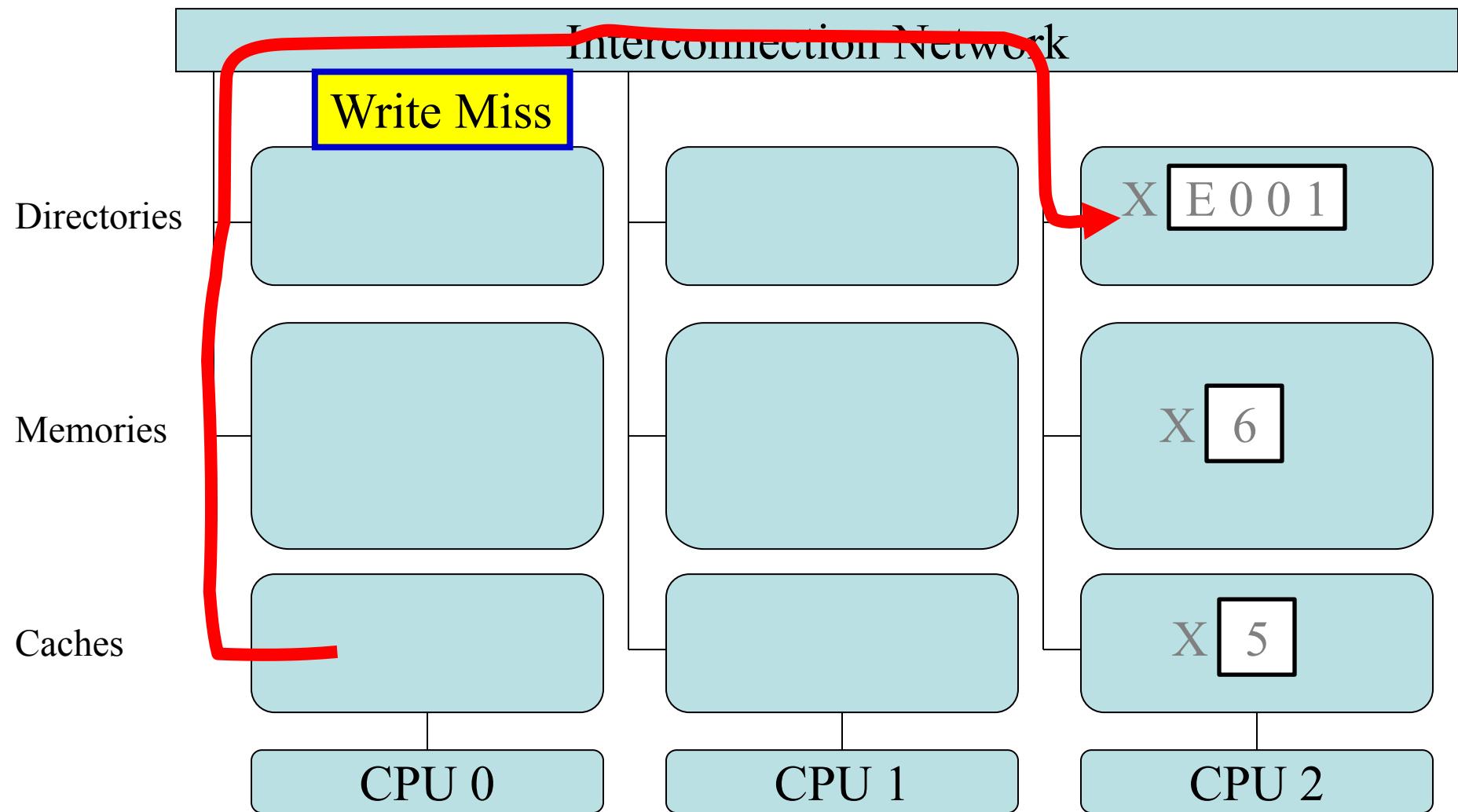
# CPU 2 escreve 5 em X



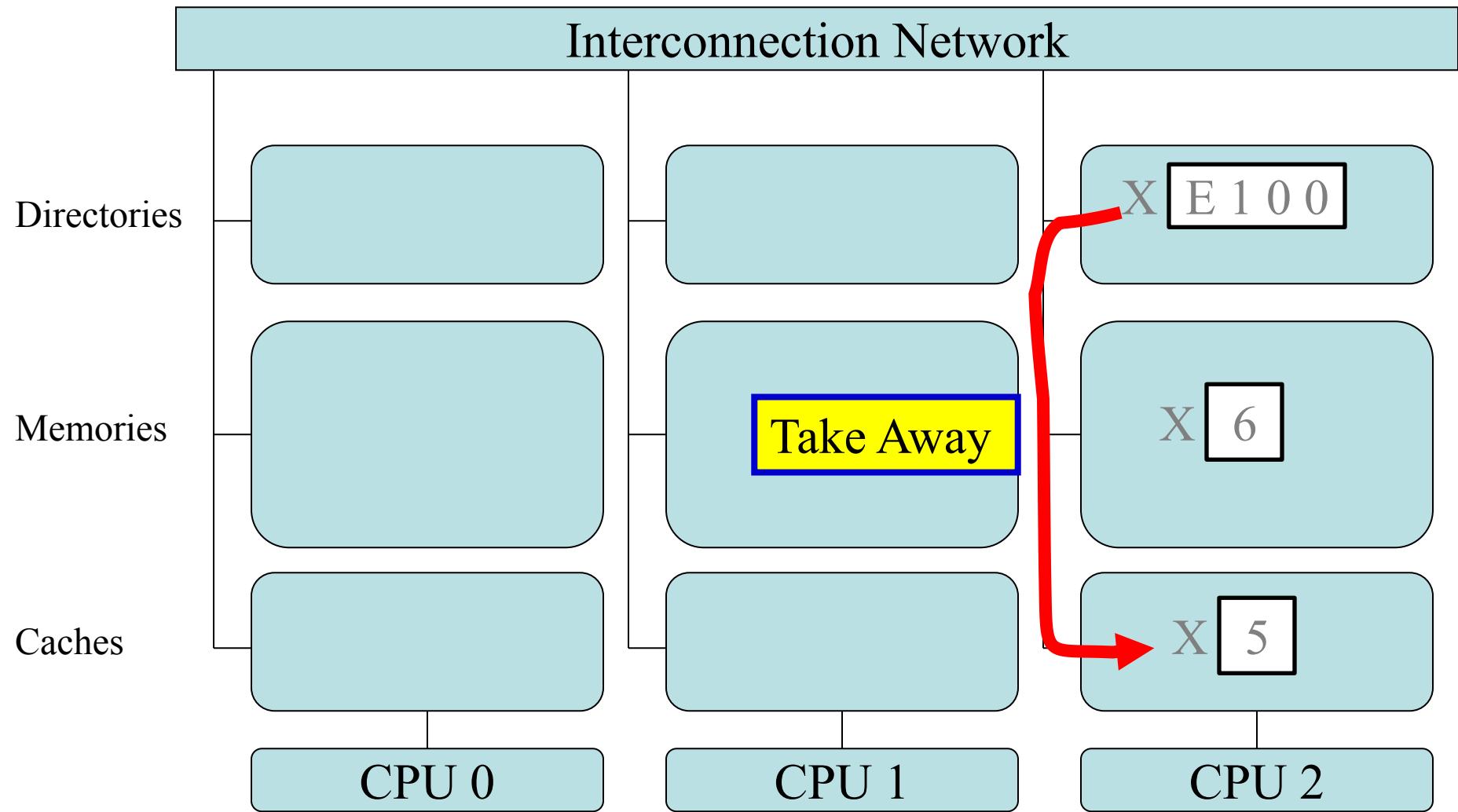
# CPU 2 escreve 5 em X (Write back)



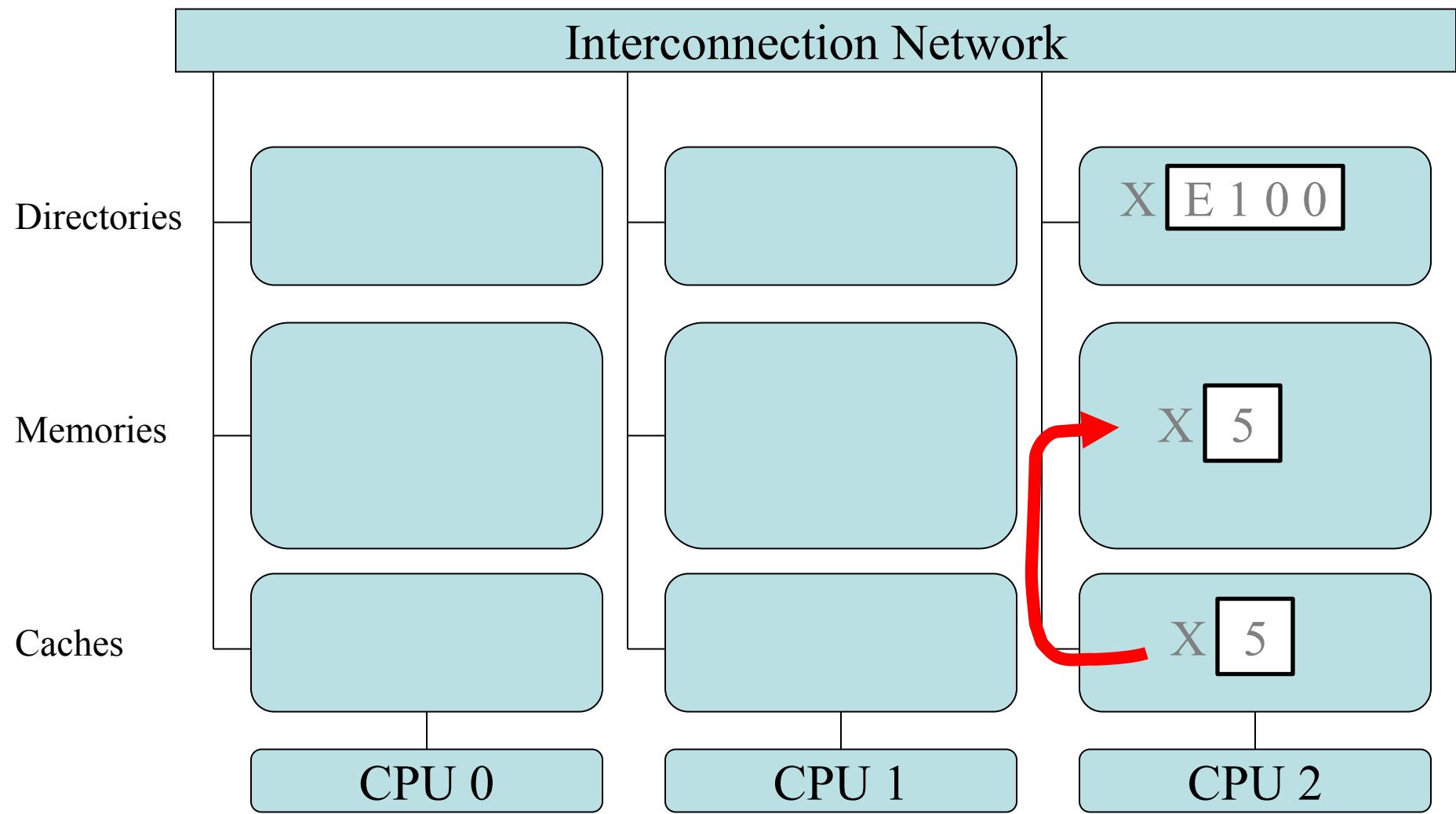
# CPU 0 escreve 4 em X



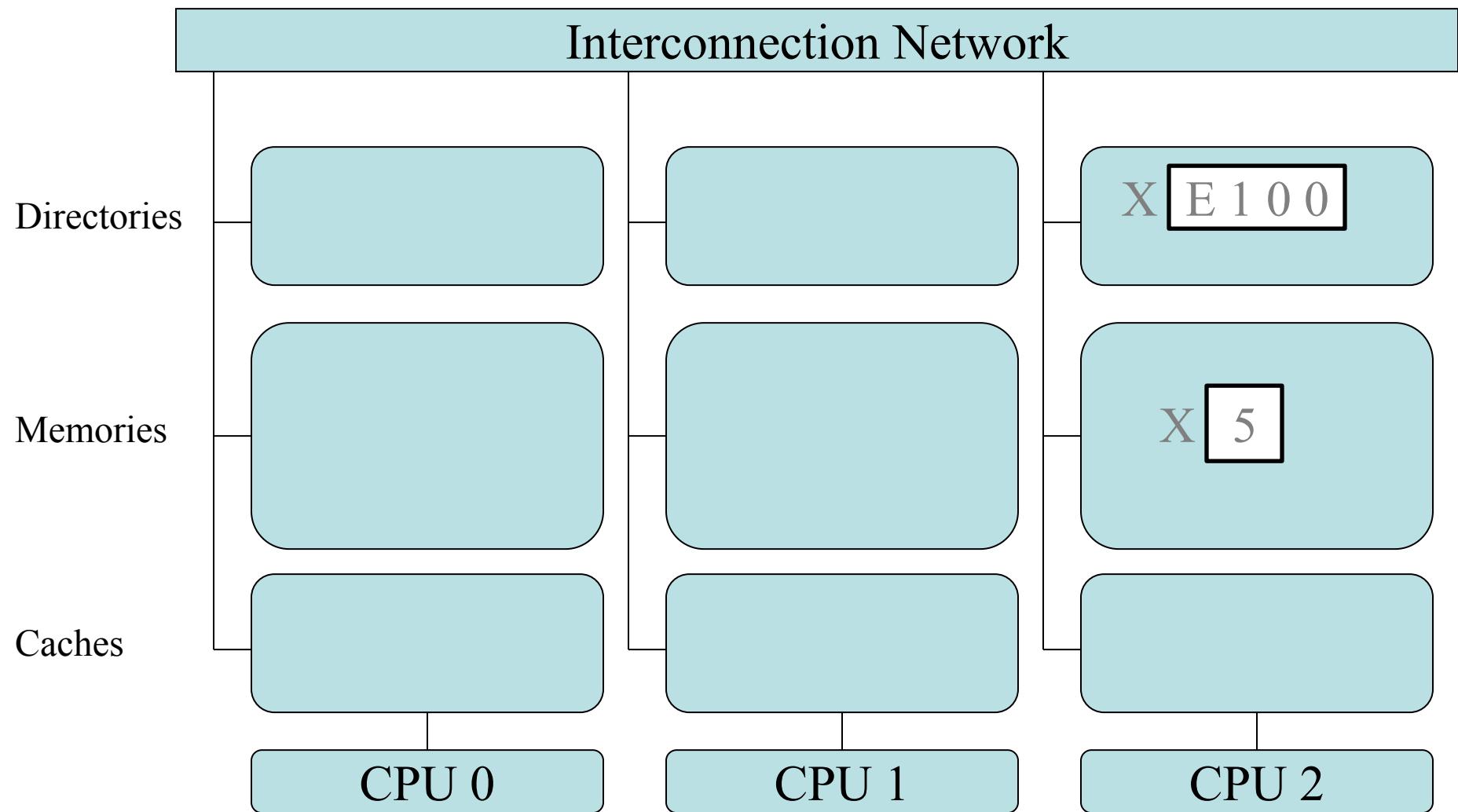
# CPU 0 escreve 4 em X



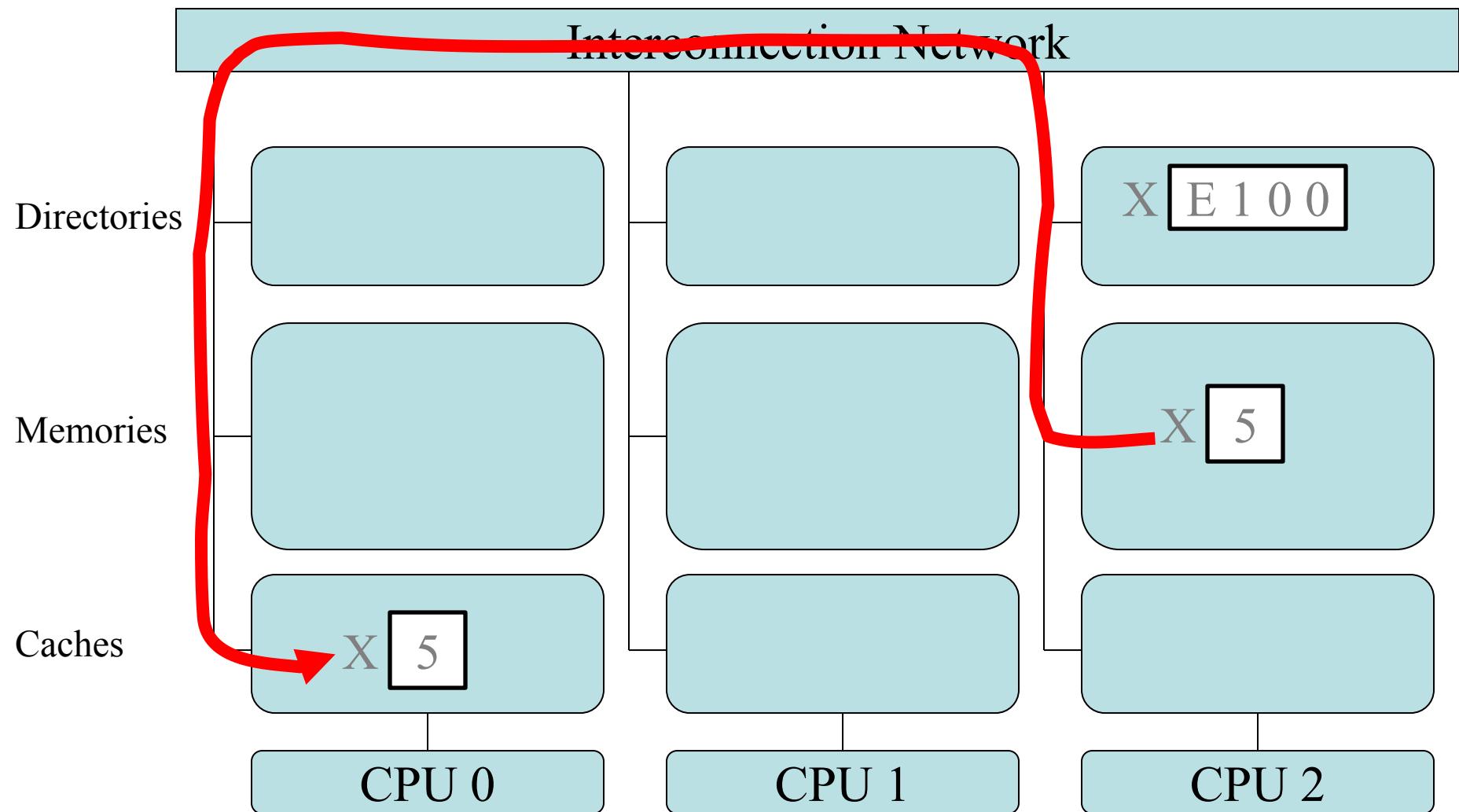
# CPU 0 escreve 4 em X



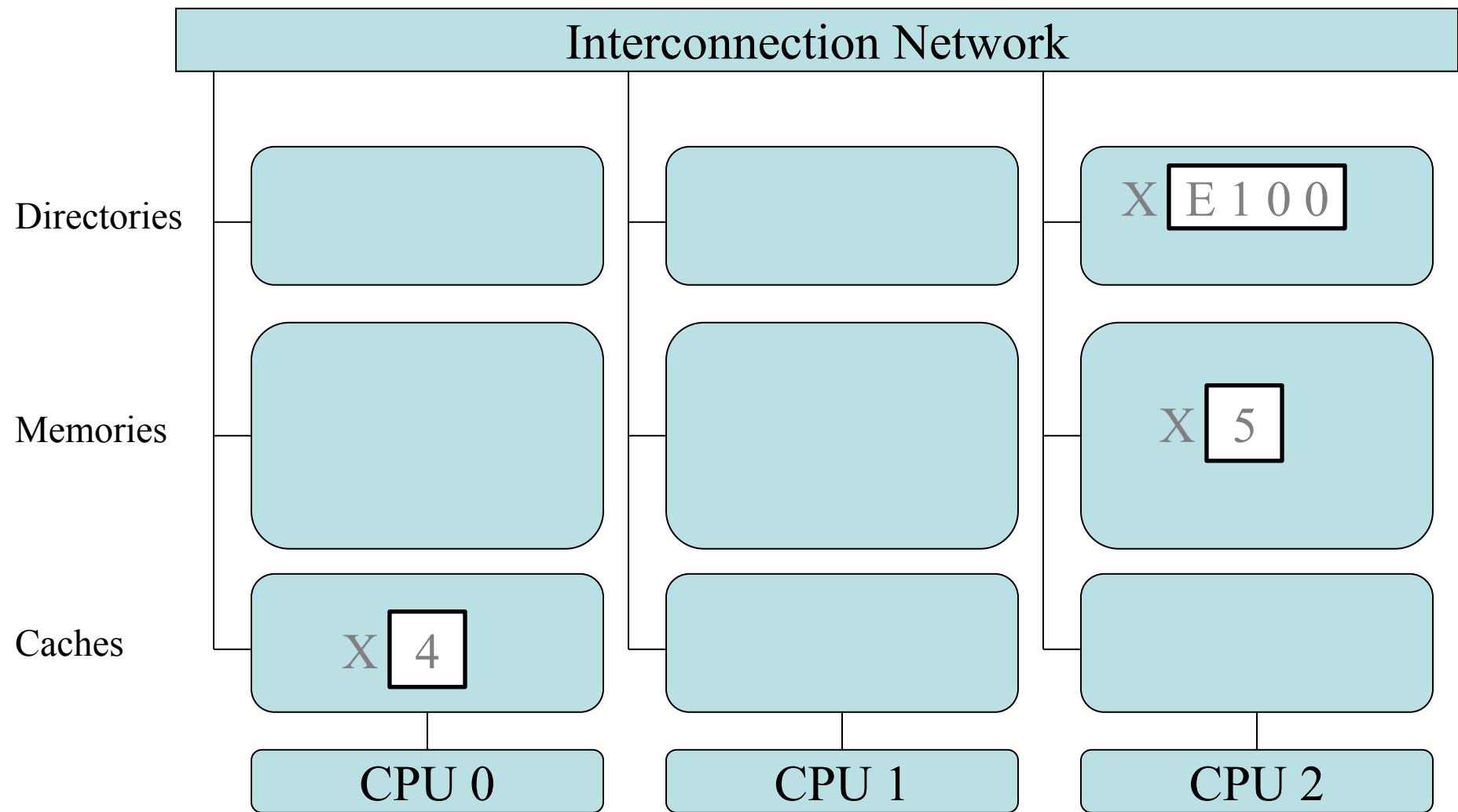
# CPU 0 escreve 4 em X



# CPU 0 escreve 4 em X



# CPU 0 escreve 4 em X



# Infraestrutura de Hardware

Melhorando o Desempenho na Hierarquia de Memória – Reduzindo o Miss Rate de Caches

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Tempo de CPU Incluindo Cache

**Tempo de CPU = Ciclos de Clock × Período de Clock**

- Ciclos de clock devem incluir:
  - Ciclos utilizados pela CPU para executar instruções
  - Ciclos gastos na espera da memória
- Espera da memória → Falta na cache
- Rescrevendo a expressão:

**Tempo de CPU = (CPU<sub>ciclos</sub> + Memoria<sub>ciclos</sub>) × Período**

- Ciclos de espera comumente é o componente mais significativo do tempo de CPU
  - Necessidade de técnicas de redução de espera

# Tempo de Espera de Memória

$$\text{Memória ciclos} = \text{Leitura ciclos} + \text{Escrita ciclos}$$

- Ciclos gastos em leitura depende da quantidade de leituras, taxa de faltas e a penalidade

$$\text{Leitura ciclos} = \#\text{Leituras} \times \text{Miss rate} \times \text{Miss penalty}$$

- Na escrita, por simplicidade, podemos negligenciar tempo de espera de write buffers (se houver)

$$\text{Escrita ciclos} = \#\text{Escritas} \times \text{Miss rate} \times \text{Miss penalty}$$

# Exemplo: Calculando Desempenho de CPU com Cache

## Problema:

CPU com  $CPI_{base} = 2$ , caso não haja nenhuma falta. Calcule a nova CPI caso:

Miss rate instrução = 2%

Miss rate dado = 4%

Miss penalty = 100 ciclos

Frequencia load/store = 36%

## Solução:

Seja  $i = \#instrucoes$

$Miss_{instrucao} \text{ ciclos} = i \times 0,02 \times 100 = 2 \times i$

$Miss_{dado} \text{ ciclos} = i \times 0,36 \times 0,04 \times 100 = 1,44 \times i$

$Memoria \text{ ciclos} = 2 \times i + 1,44 \times i = 3,44 \times i$

$CPI_{memoria} = CPI_{base} + Memoria \text{ ciclos} / i$

$CPI_{memoria} = 2 + 3,44 = 5,44$

# Exemplo: Impacto da Cache no Desempenho

- E se reduzíssemos CPI sem faltas do exemplo para 1 (através de um pipeline por exemplo)?
- Qual seria o impacto de faltas da cache no desempenho?

$$\text{CPI}_{\text{antigo}} = 2$$

$$\text{CPI}_{\text{memória}} = 2 + 3,44 = 5,44$$

$$\% \text{ de tempo de execução} = 3,44/5,44 = 63\%$$

$$\text{CPI}_{\text{novo}} = 1$$

$$\text{CPI}_{\text{memória}} = 1 + 3,44 = 4,44$$

$$\% \text{ de tempo de execução} = 3,44/4,44 = 77\%$$

# Hit Time e Desempenho

- Hit time também pode influenciar o desempenho
  - Hit time = tempo de acesso ao dado + tempo de verificação se dado está ou não na cache
  - Diferentes fatores podem alterar hit time
    - Exs: tamanho de cache, associatividade, etc
- Tempo de acesso a memória deve levar em conta tempo de misses e hits

$$\text{Tempo\_acesso}_{\text{memória}} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

# Exemplo: Calculando Tempo Médio de Acesso à Memória

## Problema:

CPU com clock de 1ns. Calcule o tempo médio de acesso à memória caso:

Miss rate <sub>instrução</sub> = 5%

Miss penalty = 20 ciclos

Hit time = 1 ciclo

## Solução:

$$\text{Tempo\_acesso}_{\text{medio}} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

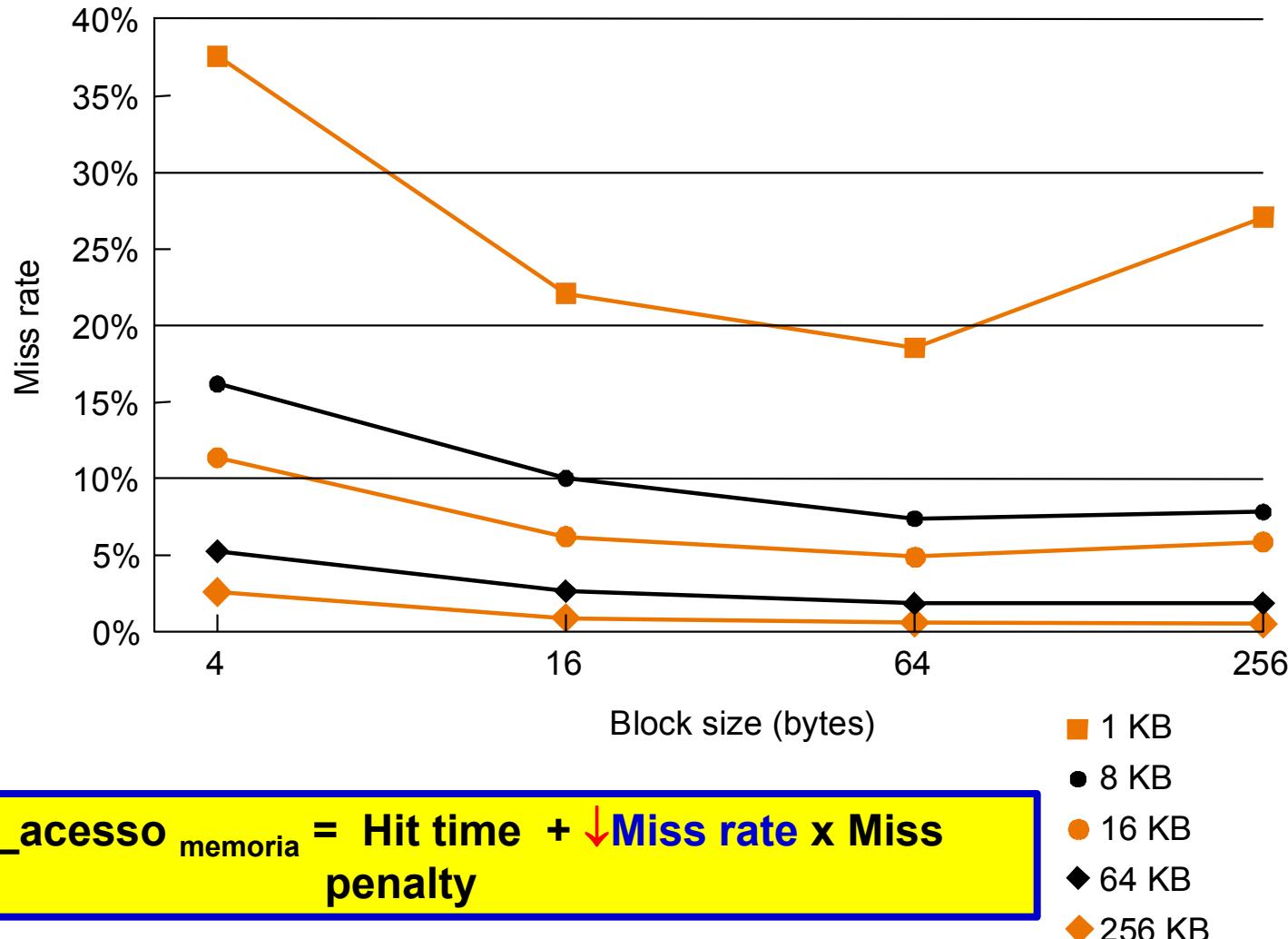
$$\text{Tempo\_acesso}_{\text{medio}} = 1 + 0,05 \times 20 = 2 \text{ ciclos ou } 2\text{ns}$$

# Melhorando Desempenho da Cache

$$\text{Tempo\_acesso}_{\text{memória}} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- Melhorar desempenho: Diferentes estratégias para atacar os diferentes fatores que influenciam tempo médio de acesso
  - Redução do Miss rate
  - Redução do Miss penalty
  - Redução do Hit time

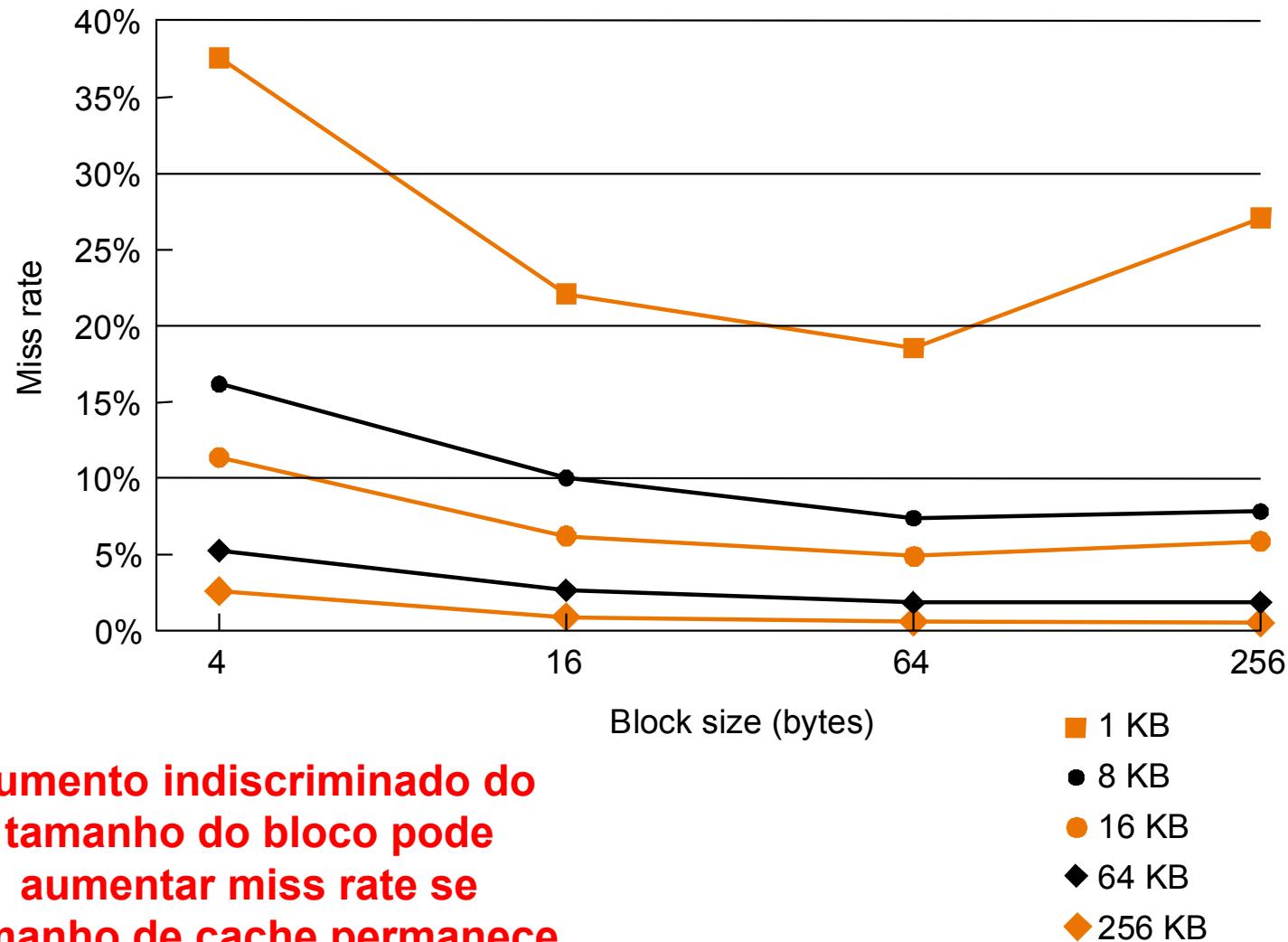
# Reducindo Miss Rate: Aumentar Tamanho do Bloco



$\text{Tempo\_acesso}_{\text{memória}} = \text{Hit time} + \downarrow \text{Miss rate} \times \text{Miss penalty}$

- 1 KB
- 8 KB
- 16 KB
- ◆ 64 KB
- ◆ 256 KB

# Tamanho do Bloco x Miss Rate



**Aumento indiscriminado do  
tamanho do bloco pode  
aumentar miss rate se  
tamanho de cache permanece  
constante**

# Considerações sobre o Aumento do Tamanho do Bloco

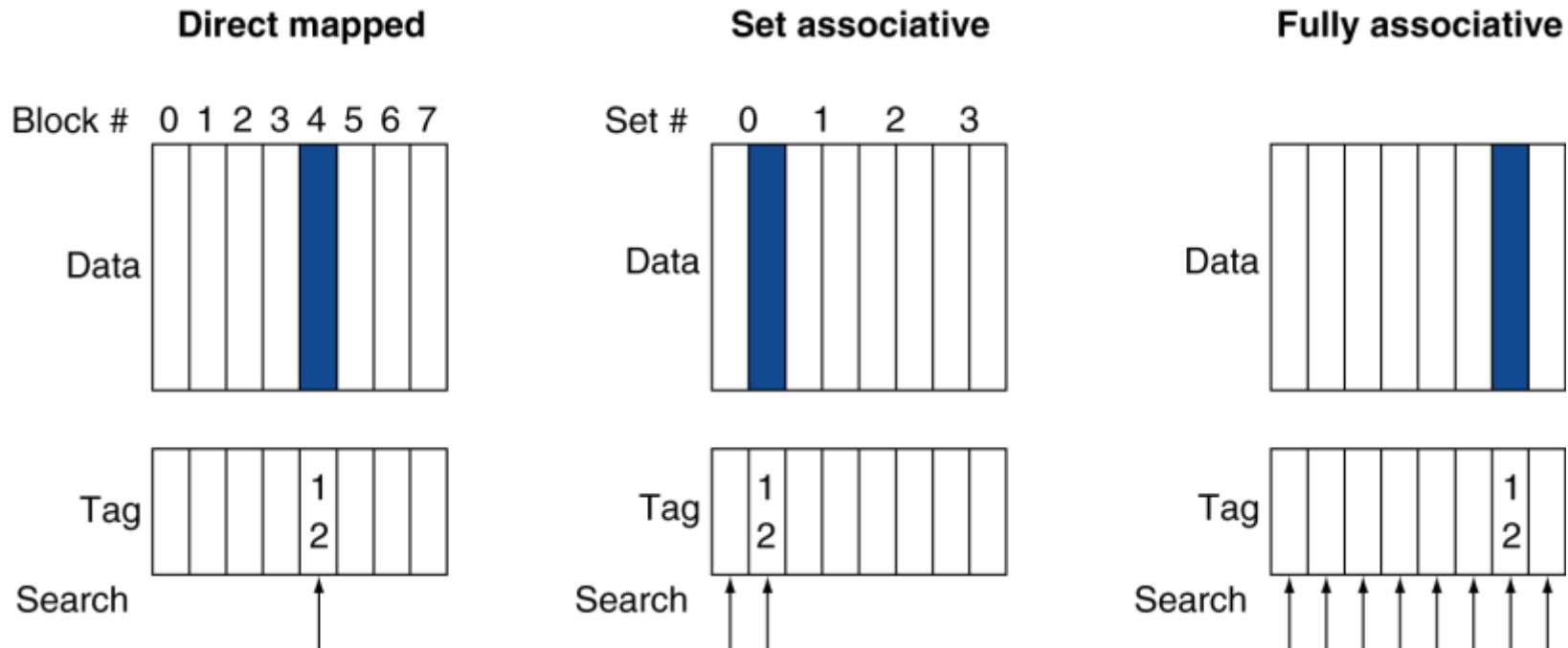
- Blocos maiores podem reduzir miss rate
  - Por causa da localidade espacial
- Contudo, em cache de tamanho fixo
  - Blocos maiores  $\Rightarrow$  menos blocos
    - Mais competição  $\Rightarrow$  aumenta miss rate
- Aumenta o miss penalty
  - Tempo de transmissão de bloco é maior

$$\text{Tempo\_acesso}_{\text{memória}} = \text{Hit time} + \downarrow \text{Miss rate} \times \uparrow \text{Miss penalty}$$

# Reduzindo Miss Rate – Associatividade

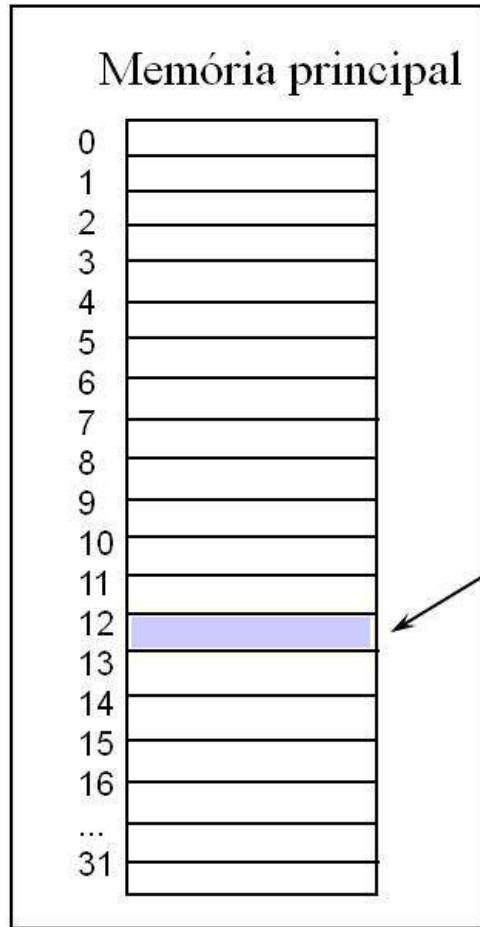
- Estratégias para posicionamento dos blocos:
  - Mapeamento direto: cada bloco de memória possui posição única na cache
  - Associativa por conjunto: cada bloco de memória pode ser colocado em algumas posições na cache
  - Completamente Associativa: cada bloco de memória pode ser colocado em qualquer posição da cache

# Reduzindo Miss Rate: Aumentar Associatividade

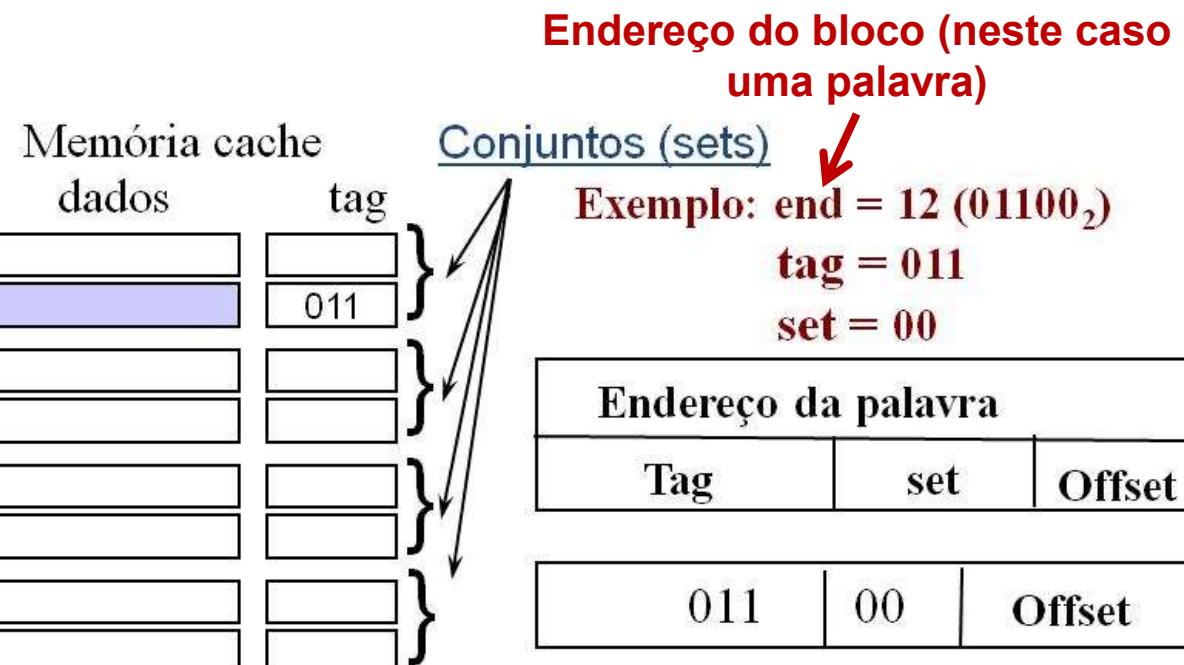


$$\text{Tempo\_acesso}_{\text{memória}} = \text{Hit time} + \downarrow \text{Miss rate} \times \text{Miss penalty}$$

# Mapeamento Associativo por Conjunto



Um bloco na memória principal pode ocupar qualquer posição dentro de um conjunto definido de blocos da cache

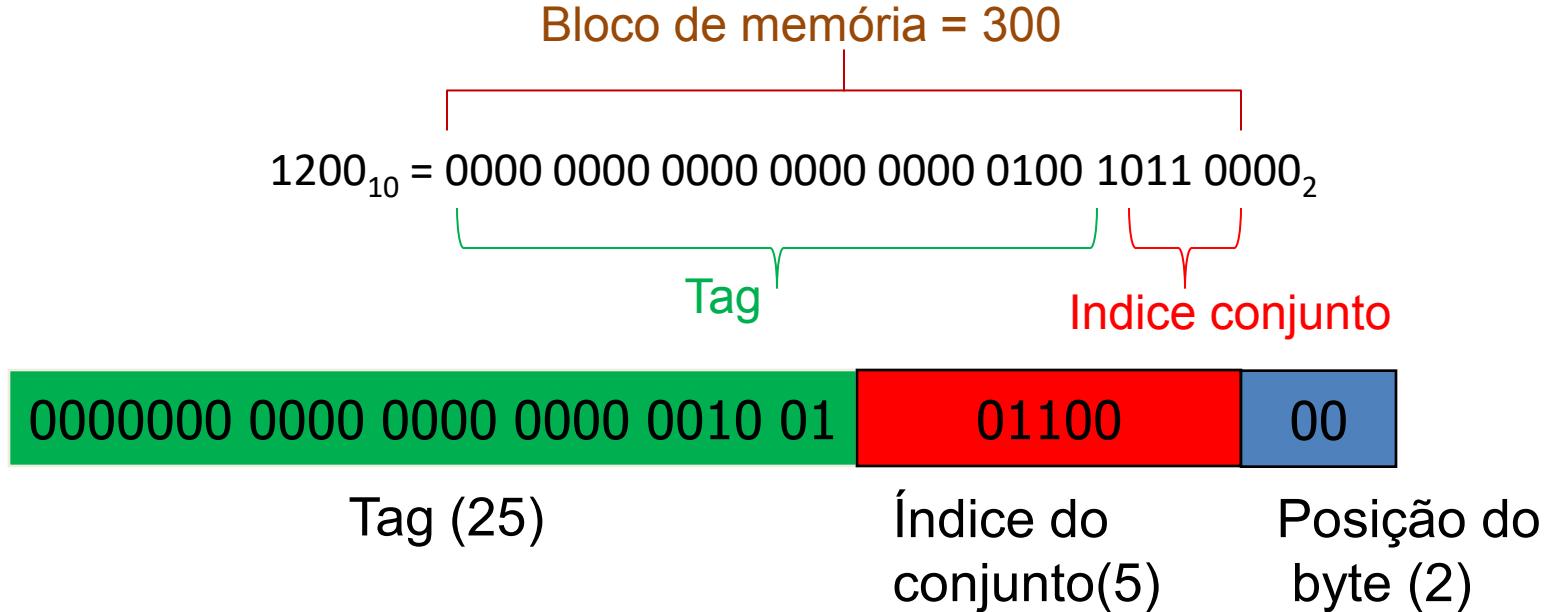


# Exemplo: Mapeando Endereço em Conjunto da Cache

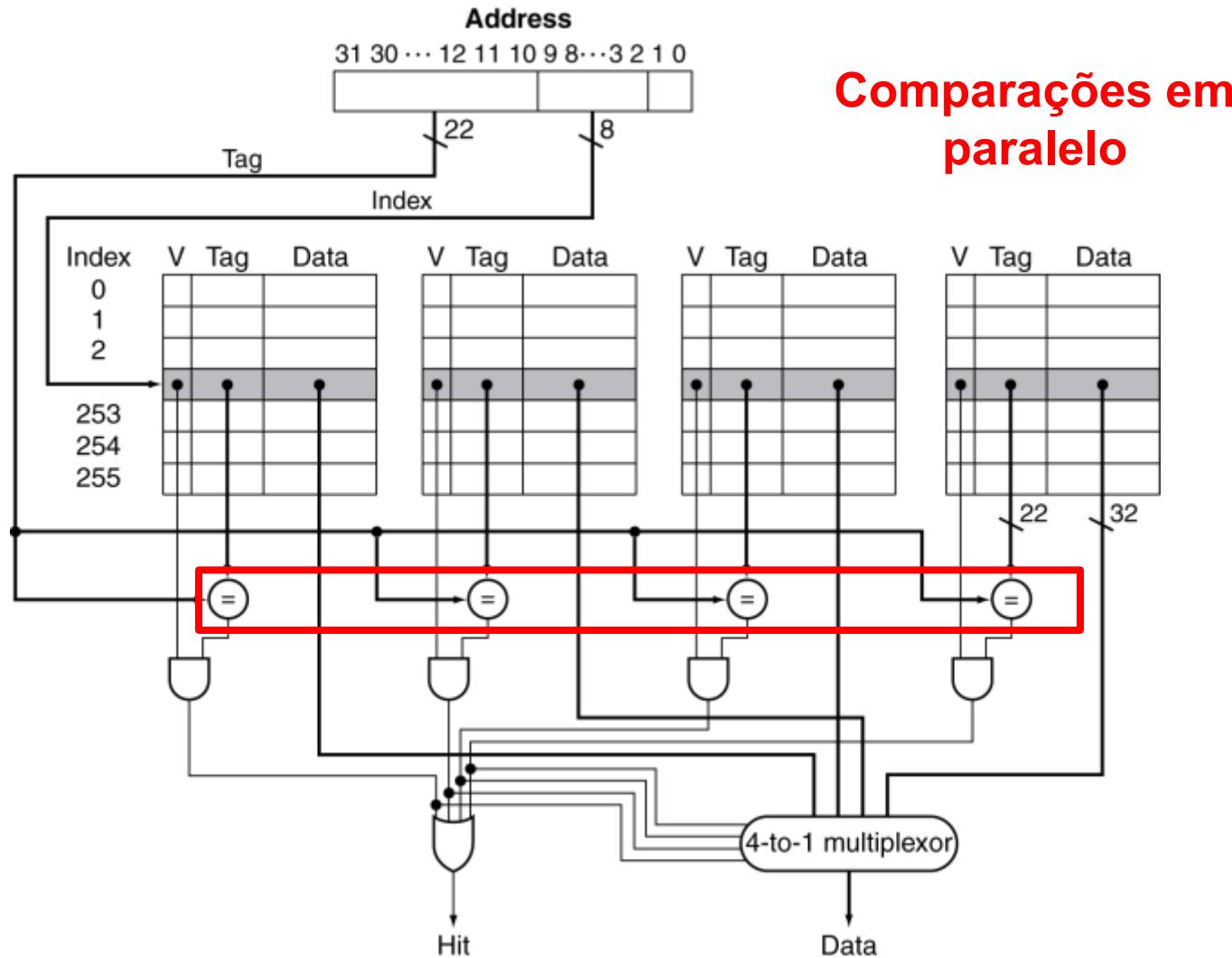
- 64 blocos, 4 bytes/bloco, associativa de grau 2
  - `addi $s0, $zero, 1200`
  - `lb $t0, 0 ($s0)`
  - Endereço do byte 1200 é mapeado para que conjunto da cache?
- Endereço do bloco na memória =  $[1200/4] = 300$
- Quantidade de conjuntos da cache =  $[64/2] = 32$
- Número do conjunto da cache =  $300 \text{ modulo } 32 = 12$

# Exemplo: Mapeando Endereço em Bloco da Cache

- 64 blocos, 4 bytes/bloco
- Endereço do bloco na memória =  $[1200/4] = 300$
- Número do conjunto da cache =  $300 \text{ modulo } 32 = 12$



# Organização de uma Cache Associativa (Grau 4)



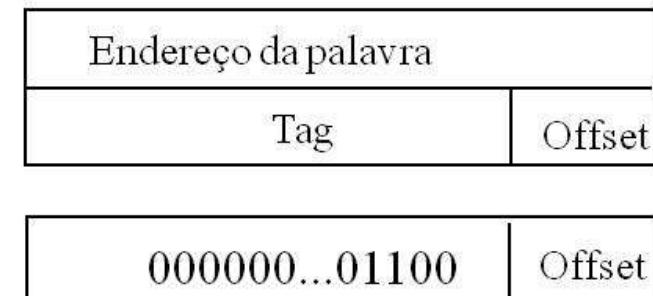
# Mapeamento Completamente Associativo

Um bloco na memória principal pode ocupar qualquer posição na cache



Tag armazena na cache o endereço do bloco na memória

Exemplo: tag = 12 ( $01100_2$ )



# Grau de Associatividade

- Para cache com 8 entradas

**One-way set associative  
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

**Two-way set associative**

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

**Four-way set associative**

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

**Eight-way set associative (fully associative)**

Tag	Data														

# Exemplo de Associatividade

- Comparativo entre caches de 4 blocos
  - Mapeamento Direto, associativa grau 2, completamente associativa
  - Sequencia de acesso aos blocos: 0, 8, 0, 6, 8
- Mapeamento Direto

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

# Exemplo de Associatividade

- Associativa de grau 2

Block address	Cache index	Hit/miss	Cache content after access		
			Set 0		Set 1
0	0	miss	Mem[0]		
8	0	miss	Mem[0]	Mem[8]	
0	0	hit	Mem[0]	Mem[8]	
6	0	Miss	Mem[0]	Mem[6]	
8	0	miss	Mem[8]	Mem[6]	

- Completamente associativa

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

# Comparação de Métodos de Mapeamento

- Mapeamento direto
  - Simples e Barata
  - Mais faltas
- Associativa
  - Menos Faltas
  - Nem sempre significativa
  - Cara (comparação do endereço em paralelo)
- Exemplo: Simulação de um sistema com 64KB D-cache, 16-palavras/bloco, SPEC2000 (Miss Rate)
  - 1-way (Mapeamento direto): 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

# Considerações sobre Aumento da Associatividade

- Blocos de memória podem ser mapeados para posições diferentes na cache
  - Diminui quantidade de conflitos entre blocos de memória mapeados para um mesmo bloco da cache → redução de miss rate
- Delay gerado por comparações
  - Aumento do hit time
  - Vários comparadores em paralelo
    - HW fica mais caro

$$\text{Tempo\_acesso}_{\text{memória}} = \uparrow \text{Hit time} + \downarrow \text{Miss rate} \times \text{Miss penalty}$$

# Reduzindo Miss Rate: Políticas de Substituição

- Mapeamento direto: sem escolha
- Associativo por conjunto
  - Entrada inválida, se existir uma
- Least-recently used (LRU)
  - Escolhe a menos usada recentemente
    - Simples para associatividade de grau 2, gerenciável para grau 4, difícil para grau além destes
- Randômico
  - Aproximadamente mesmo desempenho de LRU para alta associatividade

# Infraestrutura de Hardware

Melhorando o Desempenho na Hierarquia de Memória – Reduzindo o Miss Penalty e Hit Time de Caches e Configurando a Memória

Prof. Adriano Sarmento

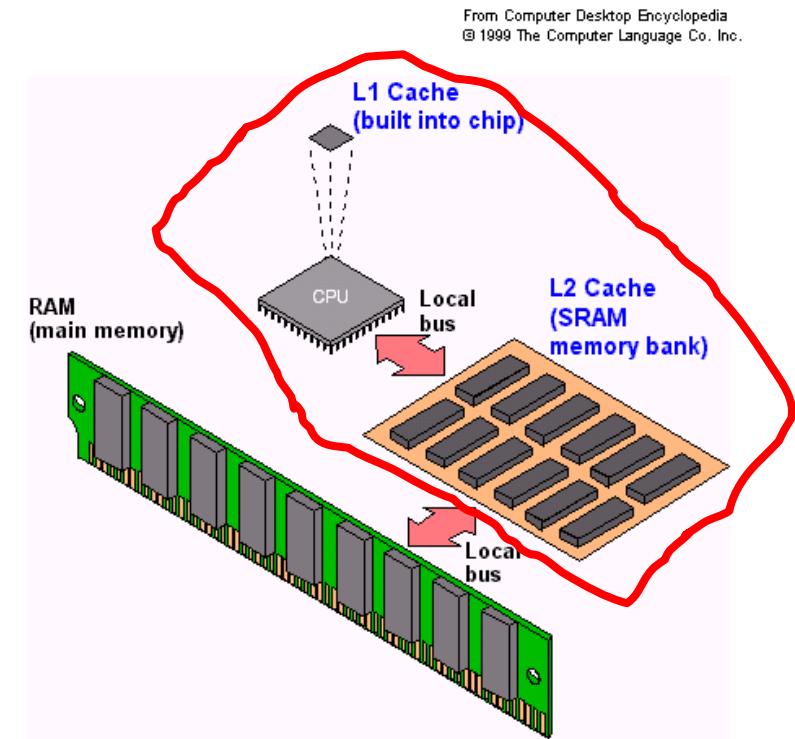
# Reduzindo Miss Penalty: Diminuição de Espera da CPU

- Blocos de memória buscados durante um miss causa a interrupção da execução da CPU
  - CPU recomeça quando bloco é colocado na cache
- Early restart
  - Assim que a palavra procurada dentro do bloco for carregada na cache esta é enviada para a CPU
- Critical Word First
  - Requisita palavra procurada primeiro e a envia para a CPU assim que a mesma foi carregada.
- Técnicas aplicáveis para grandes blocos

$$\text{Tempo\_acesso}_{\text{memoria}} = \text{Hit time} + \text{Miss rate} \times \downarrow \text{Miss penalty}$$

# Reducindo Miss Penalty: Multi-níveis de Cache

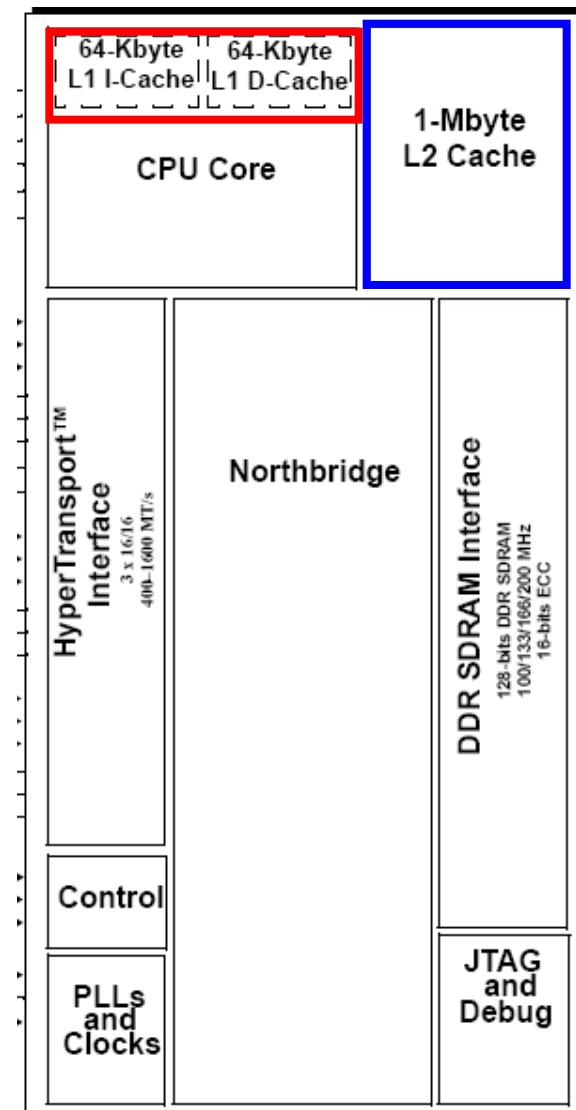
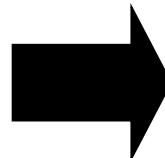
- Muitos processadores vem com pelo menos dois níveis de cache
  - 1º nível:
    - Menor tempo de acesso (hit time)
    - Menor capacidade → blocos menores
    - Objetivo: Reduzir miss penalty e hit time
  - 2º nível:
    - Maior tempo de acesso
    - Maior capacidade → geralmente blocos maiores
    - Objetivo: Reduzir miss rate



**Integrado no  
microprocessador**

# Exemplo de Cache de 2 Níveis: AMD Athlon 64

Cache de 1º nível



Cache de  
2º nível

# Desempenho de Cache de 2 Níveis

## ■ Primeiro nível de cache:

Foco em minimizar hit time e miss penalty

$$\text{Tempo\_acesso}_{L1} = \downarrow \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \downarrow \text{Miss penalty}_{L1}$$

## ■ Segundo nível de cache

Foco em minimizar miss rate

$$\text{Tempo\_acesso}_{L2} = \text{Hit time}_{L2} + \downarrow \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

# Exemplo: Melhorando Desempenho com 2 Níveis de Cache

**Problema:**

CPU com 1 nível de cache,  $CPI_{base} = 1$  e Frequência do clock = 4GHz, Miss rate  $L_1 = 2\%$ ,  
Tempo de acesso memória = 100ns

Calcule o ganho de desempenho se adicionássemos:  
2º nível de cache com hit ou miss time = 5ns e que seja grande o suficiente para reduzir o miss rate global em relação à memória para 0,5%.

# Exemplo: Melhorando Desempenho com 2 Níveis de Cache

**Problema:**

$CPI_{base} = 1$ , Frequência do clock = 4GHz, Miss rate  $L_1 = 2\%$ ,  
Tempo de acesso memória = 100ns

**Calculando CPI com 1 nível de cache**

Miss Penalty  $\frac{\text{cycles}}{\text{miss}}$  = Tempo de Acesso a Memoria/Período

Miss Penalty  $\frac{\text{cycles}}{\text{miss}}$  =  $100\text{ns}/0,25\text{ns/ciclo} = 400 \text{ ciclos}$

Memory stall  $\frac{\text{cycles}}{\text{miss}} = \text{Miss Rate} \times \text{Miss Penalty} \frac{\text{cycles}}{\text{miss}}$   
 $= 0,02 \times 400\text{ciclos} = 8 \text{ ciclos}$

$CPI_{L_1} = CPI_{base} + \text{Memory stall} \frac{\text{cycles}}{\text{miss}} = 1 + 8 = 9$

# Exemplo: Melhorando Desempenho com 2 Níveis de Cache

## Problema:

$CPI_{base} = 1$ , Frequência do clock = 4GHz, Miss rate  $L_1 = 2\%$ ,  
Tempo de acesso memória = 100ns, Hit time  $L_2 = 5\text{ns}$  e  
Miss rate  $Global = 0,5\%$

## Calculando CPI com 2 níveis de cache

Hit time  $cycles = Hit\ time_{L_2} / Período = 5\text{ns}/0,25\text{ns/ciclo} = 20\ ciclos$

Primary stalls  $cycles = 2\% \times Hit\ time_{cycles} = 0,02 \times 20 = 0,4\ ciclos$

Secondary stalls  $cycles = 0,005 \times 100/0,25/ciclo = 2\ ciclos$

$CPI_{L_1L_2} = CPI_{base} + Primary\ stalls\ cycles + Secondary\ stalls\ cycles$   
 $= 1 + 0,4 + 2 = 3,4$

Ganho  $Desempenho = CPI_{L_1}/CPI_{L_1L_2} = 9,0/3,4 = 2,6$

# Considerações sobre Multi-níveis de Cache

- Cache L1
  - Foca em minimizar hit time e miss penalty → cache menor, com blocos menores e com menos associatividade
- Cache L-2
  - Foca em minimizar miss rate → cache maior e blocos grandes, mais associatividade
- Geralmente blocos de cache L1 são menores que blocos de L2
- E quanto a duplicação de dados nos dois níveis?
  - Os dados devem ser duplicados (consistência)

# Reduzindo Hit Time: Caches Separadas

- Cache de dados e cache de instruções
  - Vantagens:
    - Maior largura de banda
    - Evita hazard estrutural
  - Desvantagens:
    - maior taxa de falta

Programa	Miss rate (instr.)	Miss rate (dado)	Miss rate (sep.)	Miss rate (única)
Gcc	6.1%	2.1%	5.4%	4.8%

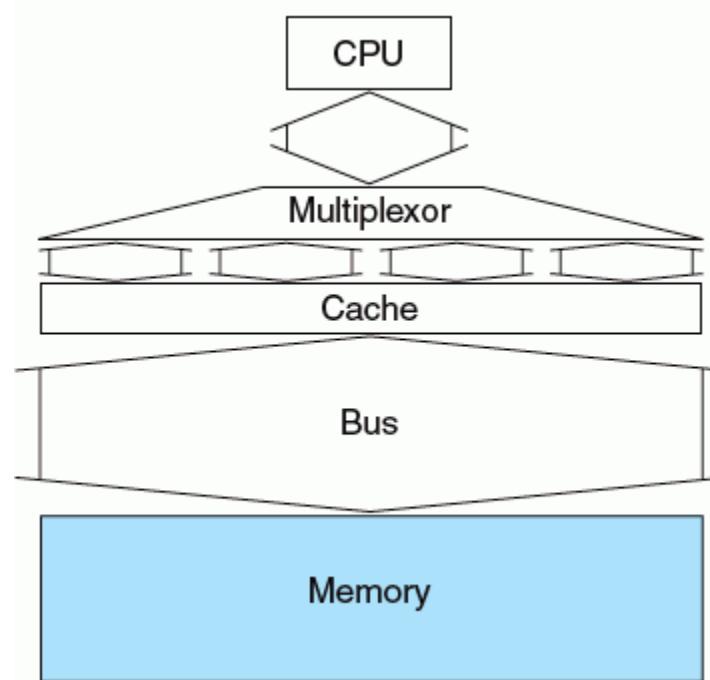
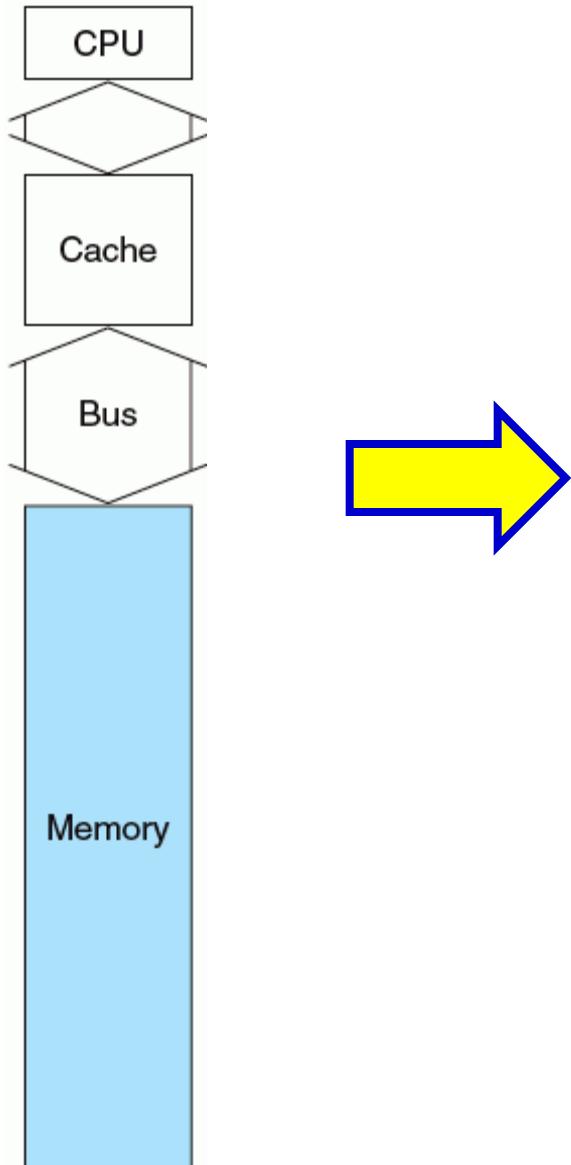
# Outras Formas de Reduzir Hit Time

- Diminuir tamanho da cache
  - Menos linhas examinadas
  - Pode ficar no mesmo chip da CPU
- Diminuir associatividade
  - Comparações em caches com associatividade alta provocam delays

# Projetando Memória para Dar Suporte a Caches

- Tempo de acesso(latência) a primeira palavra de bloco na memória é significativo
  - Decodificação do endereço e localização da palavra é demorada
  - Afeta miss penalty de uma cache
- Memória utiliza barramento para transmissão de dados
  - Velocidade do barramento < Velocidade da CPU
  - Afeta miss penalty de uma cache
- Solução: Aumentar largura de banda entre cache e memória

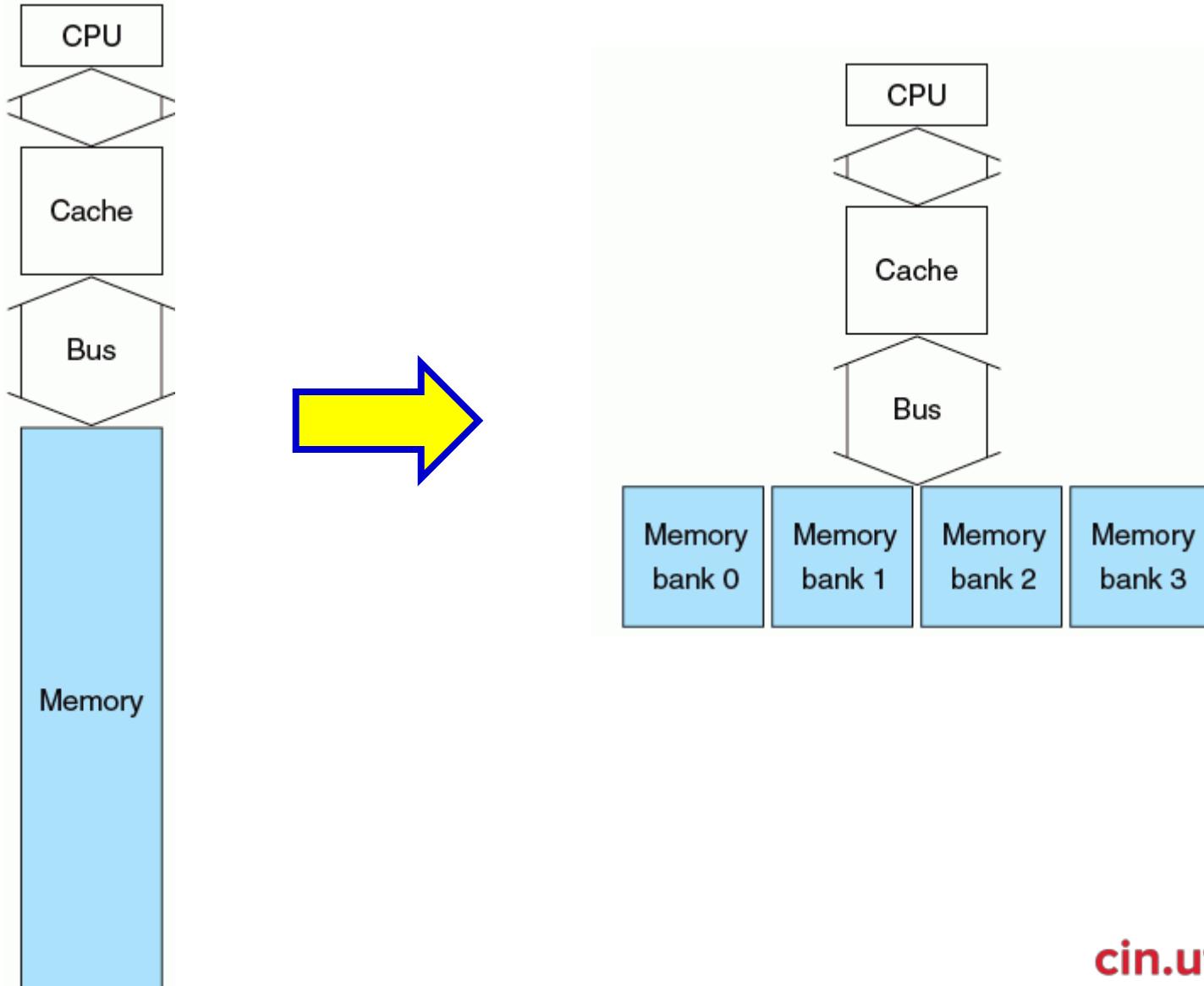
# Aumentando Largura de Banda: Memória Mais Larga



# Memórias Mais Largas

- Acesso paralelo a mais de uma palavra por bloco
  - Redução da penalidade de cache
- Necessidade de barramento mais largo
  - Expansão da largura da memória condicionada a barramento
- Necessidade de multiplexadores e lógica de controle

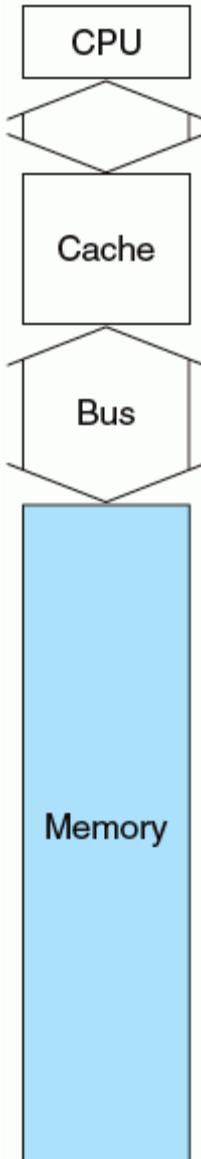
# Aumentando Largura de Banda: Memórias “Interleaved”



# Memória Interleaved

- Bancos de memória para escrita/leitura de múltiplas palavras
  - Reduz penalidade
- Endereço é enviado simultaneamente para os diferentes bancos de memória
- Largura do barramento não precisa ser aumentado
- Necessita pouco hardware adicional

# Exemplo: Tempo de Acesso de Memória Simples



## Problema:

Dada uma cache com blocos de 4 palavras e considerando que a largura do banco de memória é de 1 palavra. Calcule tempo de acesso de memória dados o número de ciclos do barramento de cada operação:

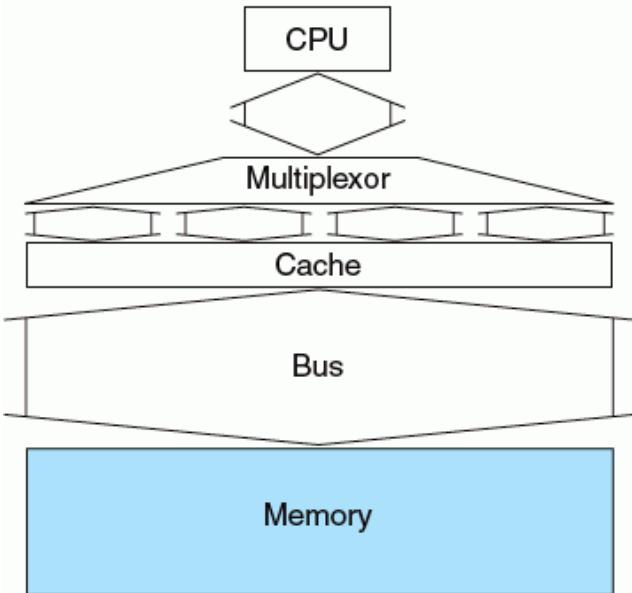
- Envio de endereço = 1 ciclo
- Acesso inicial a uma palavra = 15 ciclos
- Envio de uma palavra = 1 ciclo

## Solução:

Tempo total = Envio de endereço + acesso inicial  
+ envio de bloco

$$\text{Tempo total} = 1 + 4 \times 15 + 4 \times 1 = 65 \text{ ciclos}$$

# Exemplo: Tempo de Acesso de Memória Mais Larga



## Problema:

Dada uma cache com blocos de 4 palavras e considerando que a largura do banco de memória é de 2 palavras. Calcule tempo de acesso de memória dados o número de ciclos do barramento de cada operação:

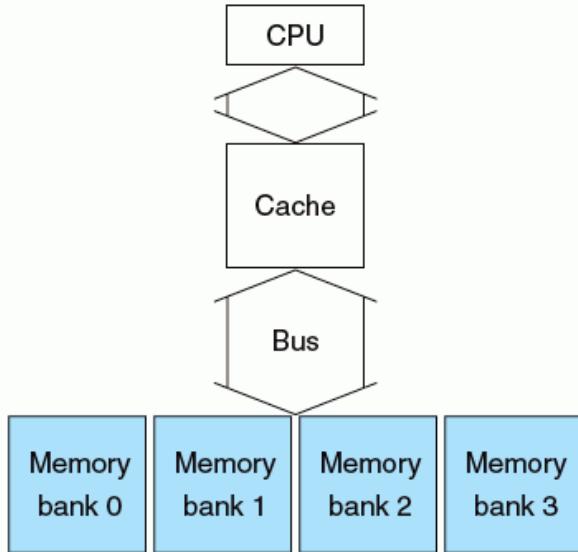
- Envio de endereço = 1 ciclo
- Acesso inicial a uma palavra = 15 ciclos
- Envio de uma palavra = 1 ciclo

## Solução:

**Tempo total = Envio de endereço + acesso inicial + envio de bloco**

$$\text{Tempo total} = 1 + 2 \times 15 + 2 \times 1 = 33 \text{ ciclos}$$

# Exemplo: Tempo de Acesso de Memória Interleaved



## Problema:

Dada uma cache com blocos de 4 palavras e considerando que existem 4 bancos de memória com largura de 1 palavra. Calcule tempo de acesso de memória dados o número de ciclos do barramento de cada operação:

- Envio de endereço = 1 ciclo
- Acesso inicial a uma palavra = 15 ciclos
- Envio de uma palavra = 1 ciclo

## Solução:

**Tempo total = Envio de endereço + acesso inicial + envio de bloco**

$$\text{Tempo total} = 1 + 1 \times 15 + 4 \times 1 = 20 \text{ ciclos}$$

# Considerações Finais sobre Desempenho de Caches

- Aumento do desempenho da CPU
  - Miss penalty se torna mais significativo
- Redução da CPI base
  - Maior proporção do tempo é gasto em ciclos de espera pela memória (memory stalls)
- Aumento da frequencia do clock
  - Memory stalls gastam mais ciclos
- Cache não pode ser negligenciado no cálculo do desempenho
  - Muitos parâmetros para conseguir configuração certa de cache
- Organização da memória ajuda a melhorar desempenho de cache

# Infraestrutura de Hardware

Memória Virtual

Prof. Adriano Sarmento

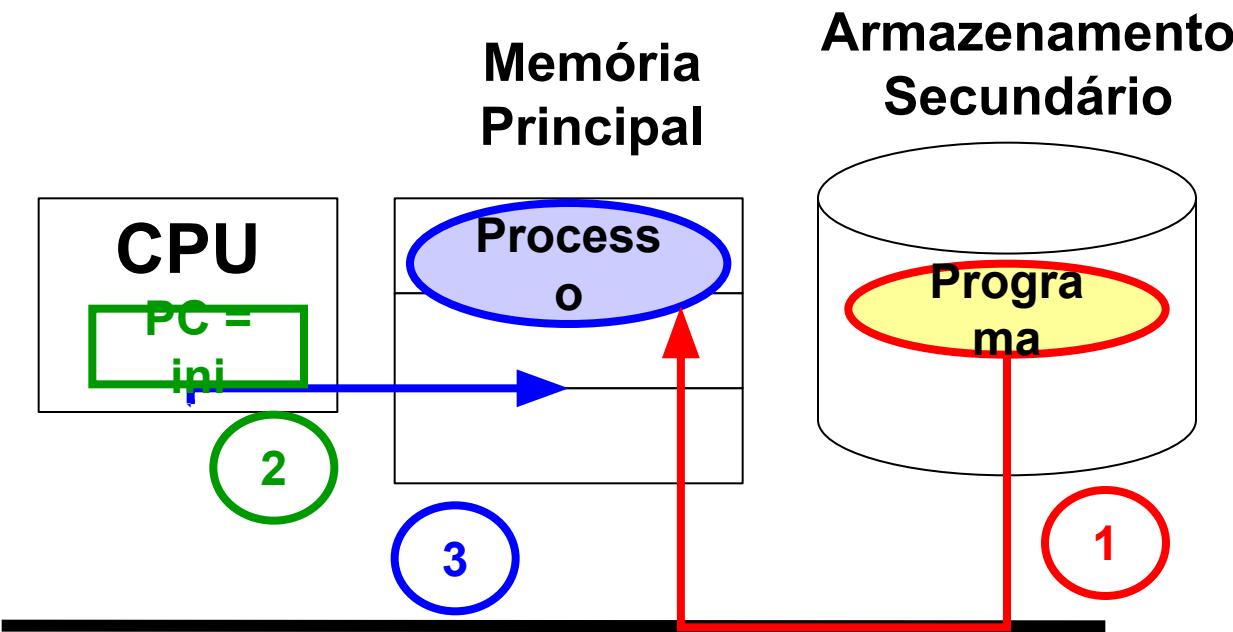
# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Memória

- Programas devem ser carregados na memória para poderem ser executados
- Memória é finita  não comporta todos os programas de um computador
- Hierarquia de memórias
  - Cache – rápida e cara
  - Memória principal – velocidade e custo médios
  - Disco – velocidade e custo baixo

# Execução de um Programa



1. Programa vira processo e seus dados e código objeto são carregados na memória
2. PC aponta para endereço de memória do início do programa
3. CPU acessa memória para executar cada instrução e carregar dados e/ou escrever dados

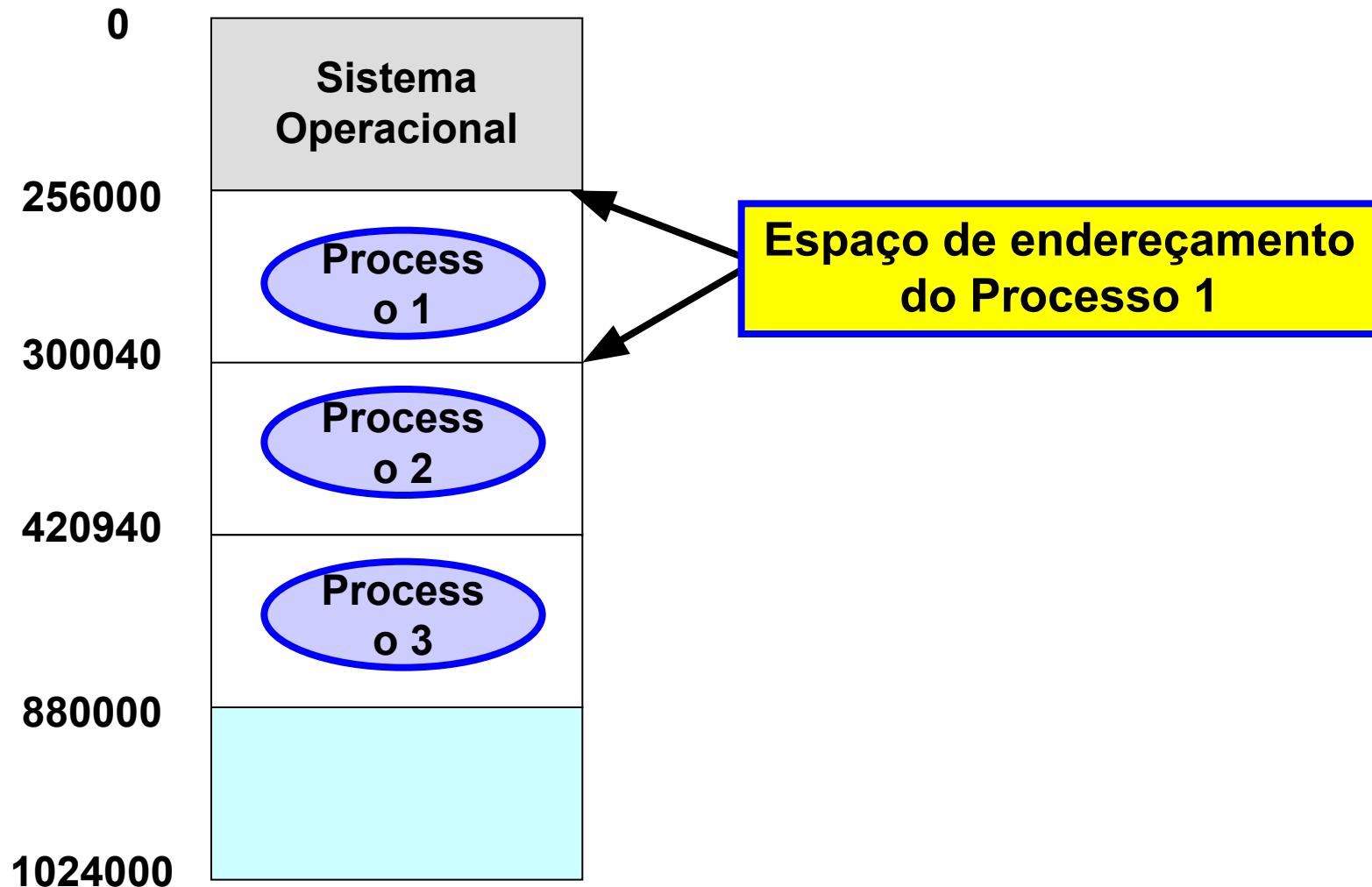
# Memória Virtual

- Técnica utilizada para criar ilusão de que memória é maior do que ela é
- Separação entre espaço de endereçamento visto pelo processo ( e CPU) e o endereçamento real da memória
  - **Memória Virtual (Lógica)**
    - Gerado pela CPU
    - Pode ser maior do que memória principal, neste caso considera outros recursos de armazenamento como se fosse uma única grande memória
  - **Memória Física**
    - Suportado pela memória principal

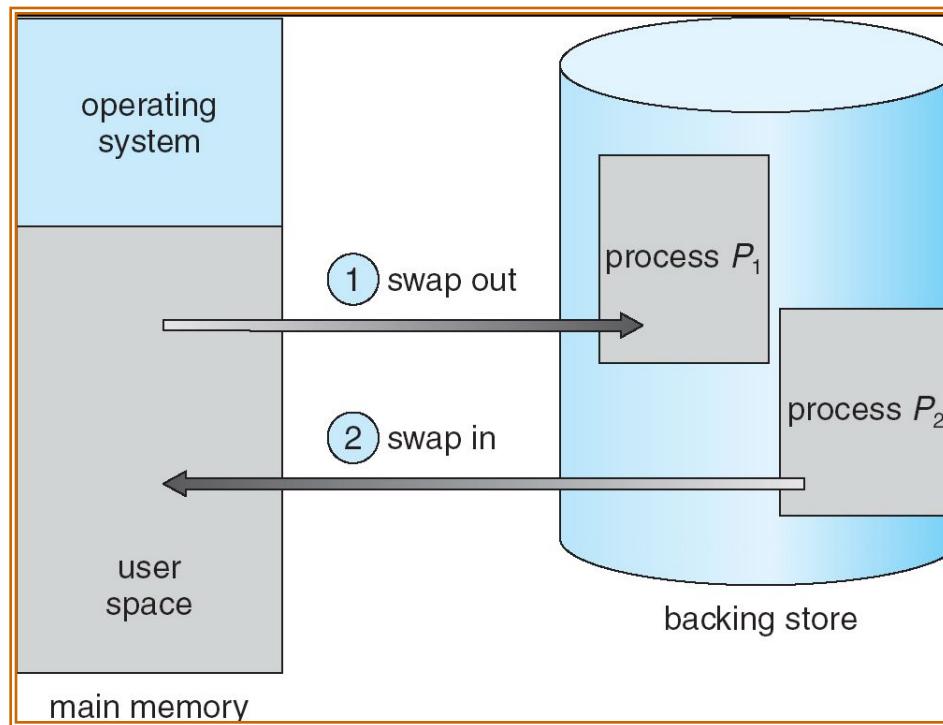
# Mais Sobre Memória Virtual...

- Memória principal funciona como uma “cache” para o disco
- Gerenciamento de memória virtual é feita pela CPU e Sistema Operacional (S.O)
  - Tradução de endereços virtuais em reais
  - Busca de dados do disco para a memória
  - Proteção
- Programas compartilham memória
  - Cada programa tem seu espaço de endereçamento
  - Protegidos de acesso por parte de outros programas

# Espaço de Endereçamento



# Swapping – Troca de Processos



- Processos residentes na memória podem ser movidos para disco e vice-versa
- Tempo de mudança de contexto é alto
  - **Maior parte gasto na transferência do disco!**

# Partições de Memória

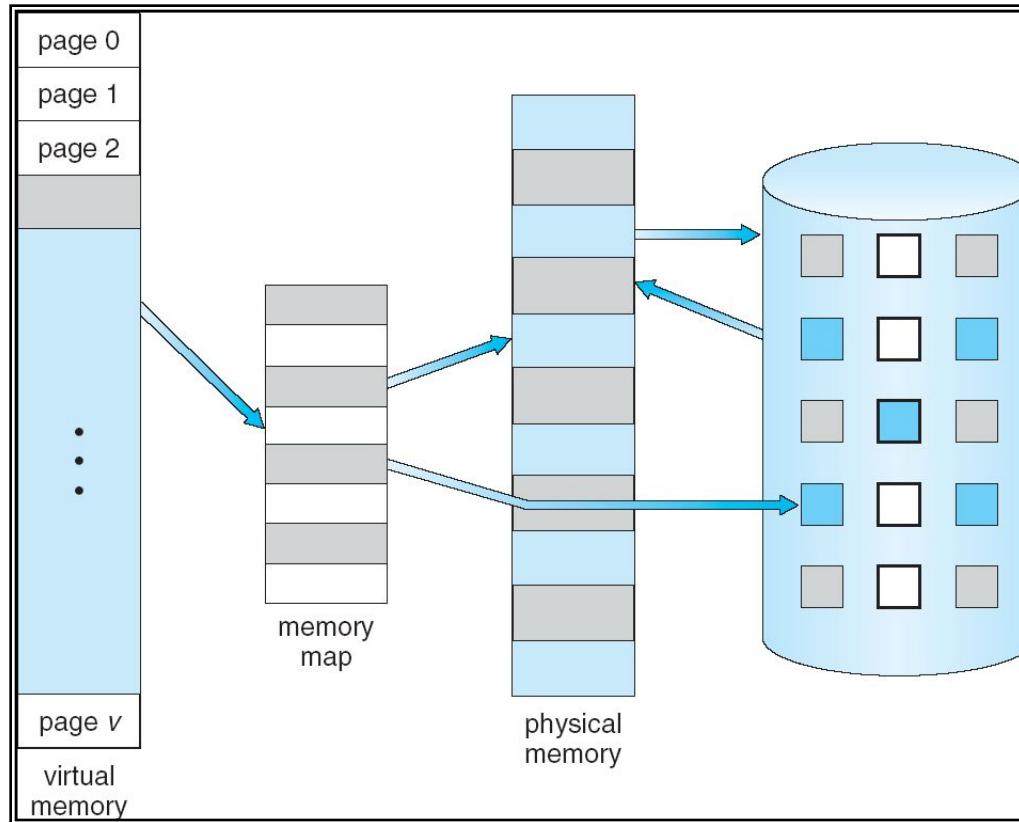
- Multiprogramação □ vários processos na memória
- Memória pode ser dividida em partições (blocos contíguos de memória)
- S.O aloca para cada processo um espaço contíguo de memória (partição)
- Nem sempre espaço contíguo existe

**Solução:**  
**Paginação**

# Paginação

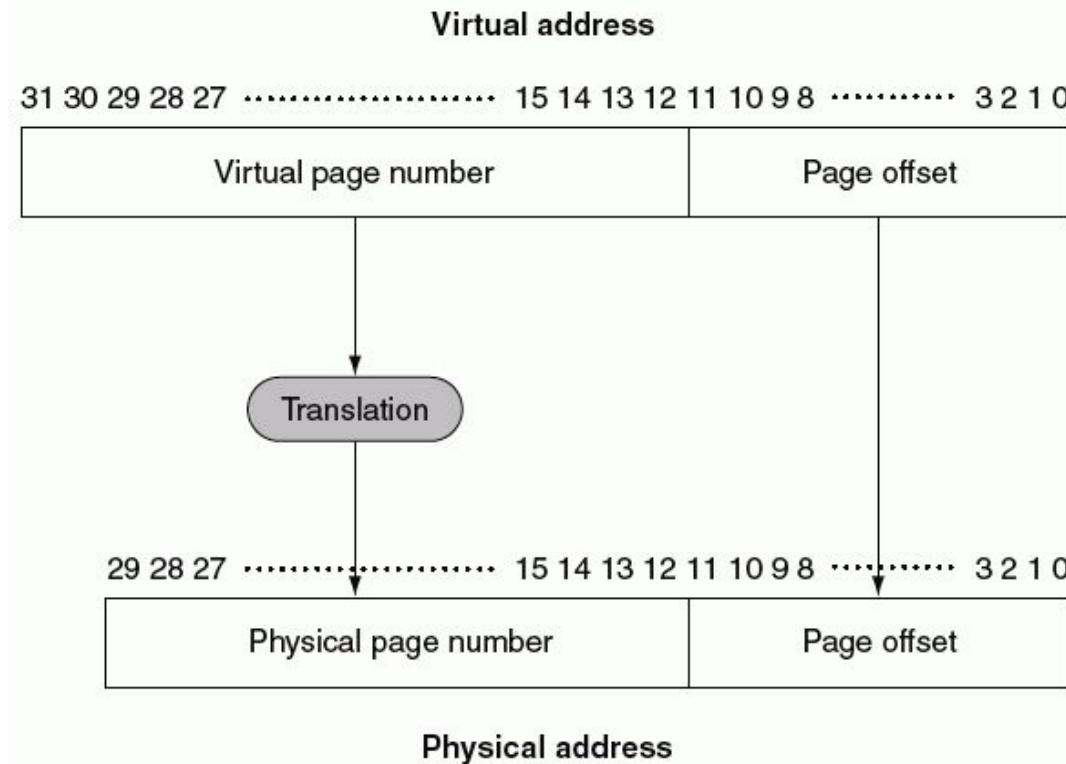
- Paginação é uma técnica de gerenciamento de memória que permite que espaço de endereçamento de processo seja não contíguo
  - Espaço de endereçamento é dividido em páginas
  - Cache: blocos, Memória Virtual: páginas
    - Página pode corresponder mais de um bloco
    - Tamanho típico: 4KB – 64KB
- Vantagens:
  - Diferentes partes de um processo podem estar espalhados na memória
  - Algumas partes não utilizadas podem estar em disco

# Paginação e Memória Virtual



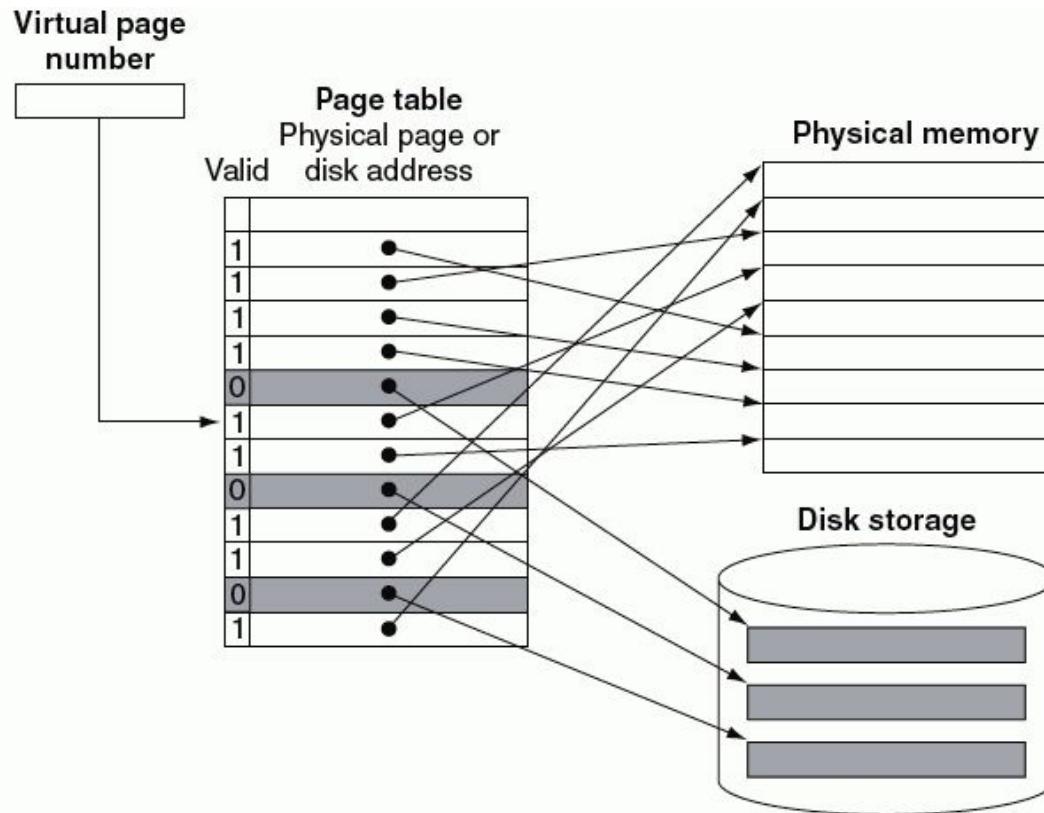
- Espaço de endereçamento virtual é maior do que o oferecido pela memória física  algumas páginas podem estar no disco

# Tradução de Endereços



- Endereços virtuais precisam ser traduzidos para endereços reais
  - Geralmente bits mais significativos indicam número de página

# Tabela de Páginas

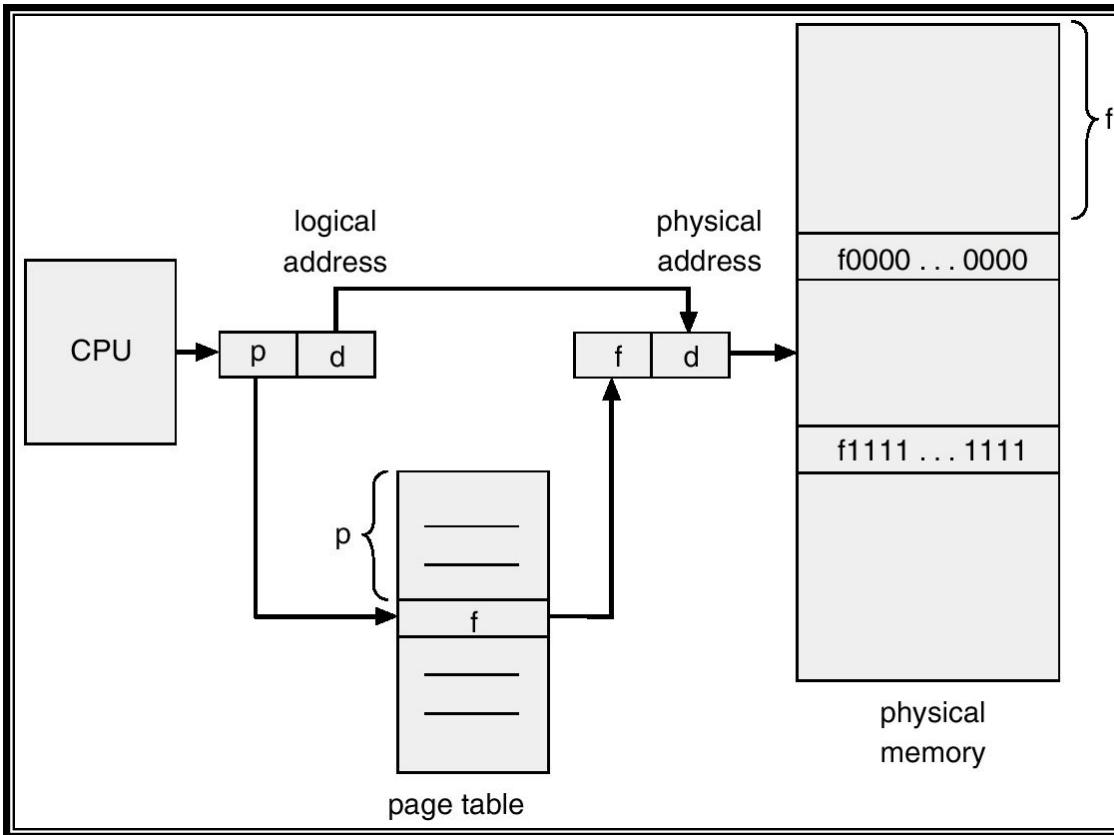


- Tabela de páginas faz a correspondência entre as páginas da memória virtual e os endereços das páginas da memória física

# Implementação de Tabela de Páginas

- Tabela de páginas é armazenada na memória
- Indexada pelo número de página virtual
- Geralmente para cada processo existe uma tabela de páginas associada
- Comumente a CPU possui registradores que armazenam o endereço da tabela de páginas
  - Registrador **Context** no MIPS

# Método de Tradução de Endereços Virtuais



- Se espaço de endereçamento virtual é  $2^m$  e tamanho de página é  $2^n$ , os  $m - n$  bits de mais alta ordem de um endereço virtual dão o número da página e os  $n$  bits restantes dão o deslocamento dentro da página

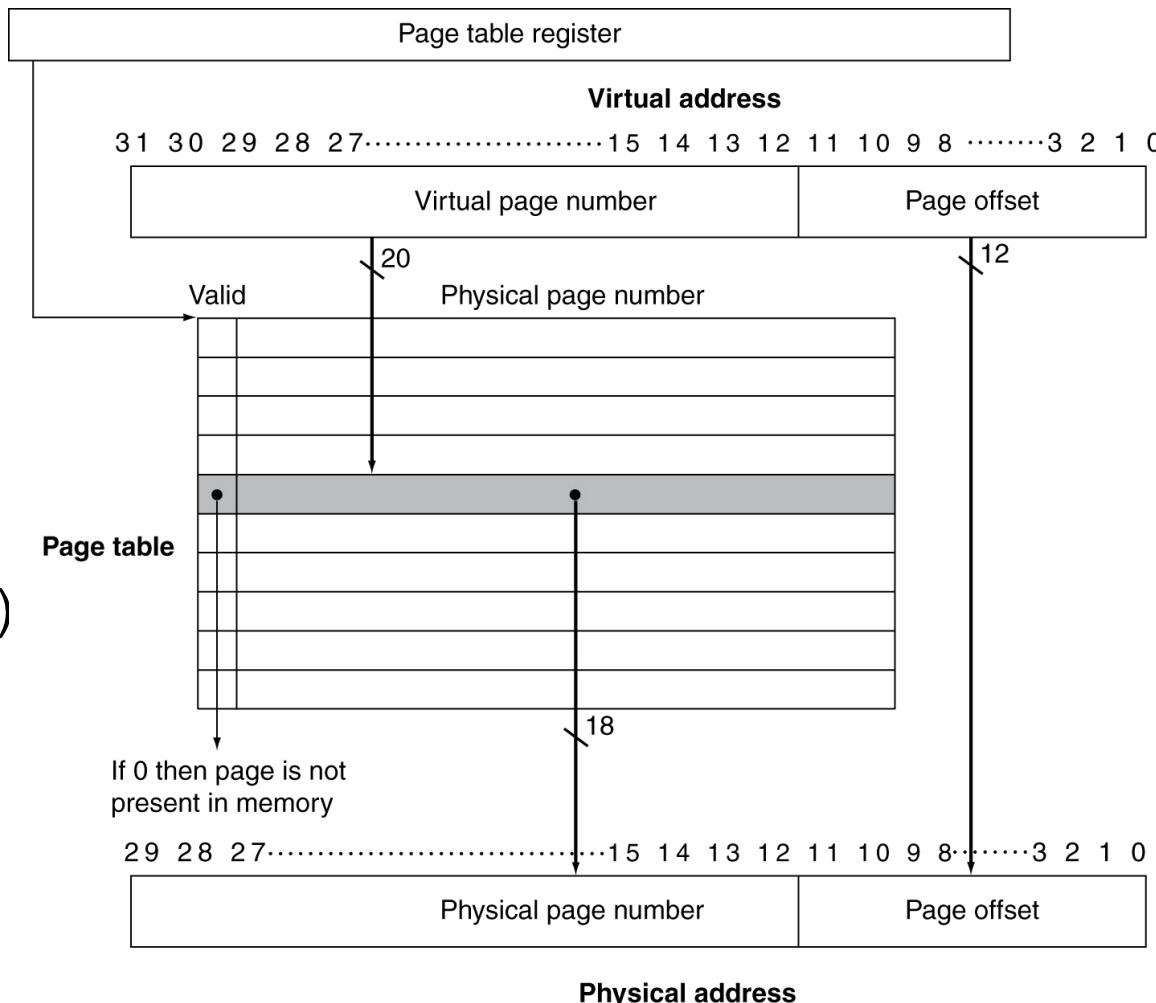
**p – número da página virtual**

**d - deslocamento dentro da página**

**f – página física**

# Exemplo de Tradução de Endereços Virtuais

- Endereços de 32 bits
- Tamanho de página: 4KB ( $2^{12}$  bytes)
- Número de entradas na tabela de páginas:  $2^{20}$
- Espaço de endereçamento virtual: 4GB ( $2^{32}$  bytes)
- Espaço de endereçamento físico: 1GB ( $2^{30}$  bytes)



# Informações de uma Tabela de Páginas

Endereço da página física

Modificada

Leitura ou Escrita/Leitura

--	--	--	--

Entrada de uma tabela de páginas

- Uma entrada em uma tabela de páginas pode conter outros dados além da localização da página:
  - Bit informando se a página é somente leitura ou leitura-escrita
  - Bit informando se a página é válida ou não (on - está na memória, off – está no disco)
  - Bit informando se a página foi modificada na memória (dirty bit)

# Minimizando Uso de Memória na Paginação

- Se espaço de endereçamento virtual for grande, tabela de páginas será grande

Espaço de endereçamento virtual =  $2^{32}$

Tamanho de página = 4KB ( $2^{12}$ )

**Tabela de página tem 1 milhão ( $2^{32} / 2^{12}$ ) entradas!**

- Alocar tabela de páginas contiguamente na memória é impraticável

Solução:

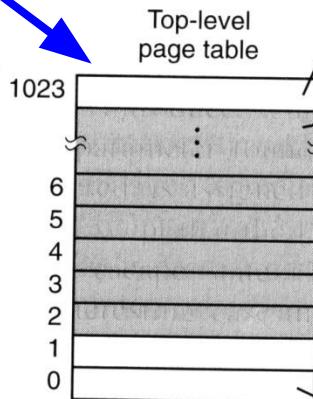
**Tabelas de Páginas Multi-Níveis!**

# Tabelas de Páginas Multi-Níveis

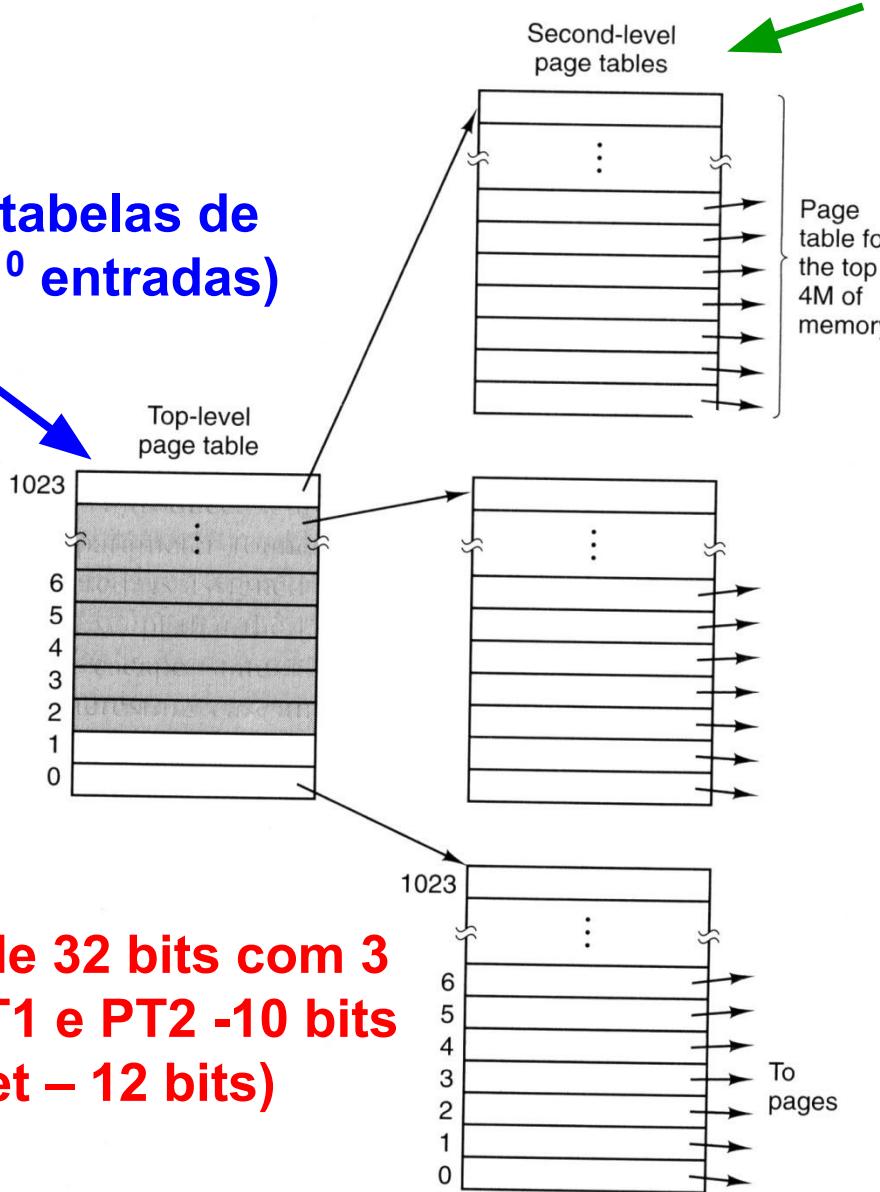
- Consiste em **paginar** a tabela de páginas
- Quebra-se espaço de endereçamento virtual em múltiplas tabelas de páginas
- Existe uma tabela de tabela de páginas (a de maior nível) que contém os índices (localização) para as demais tabelas de páginas
- Tabelas de páginas são carregadas na memória de acordo com a necessidade

# Tabela de Páginas de 2 Níveis

Tabela de tabelas de páginas ( $2^{10}$  entradas)



Endereço de 32 bits com 3 campos (PT1 e PT2 -10 bits e Offset – 12 bits)



PT1 indica a tabela de páginas desejada

Uma vez carregada a tabela de páginas desejada, PT2 indica qual é a página que deve ser carregada

# Falta de Página (Page Fault)

- Execução de programa não implica necessariamente no carregamento de todo ele do disco para a memória
- Geralmente se carrega algumas páginas do programa e quando necessário outras páginas são trazidas
- Quando a execução do programa requisita uma página que não está na memória, dizemos que ocorreu uma falta de página ou page fault

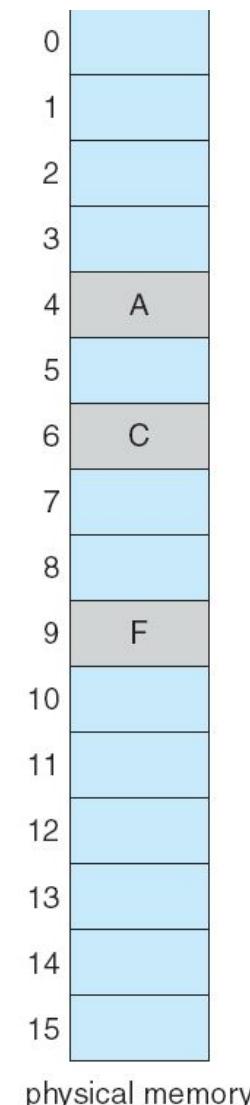
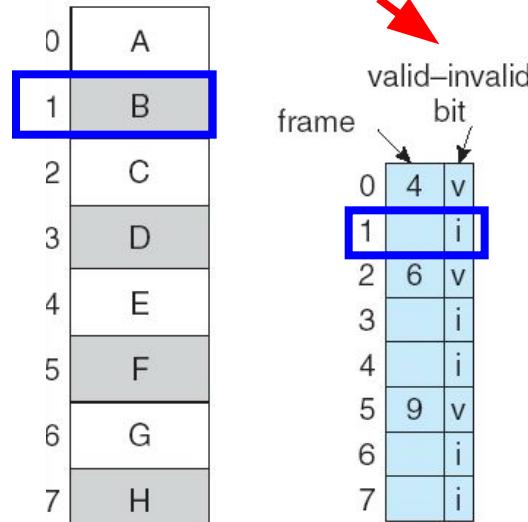
# Preço de uma Falta de Página

- Preço da penalidade

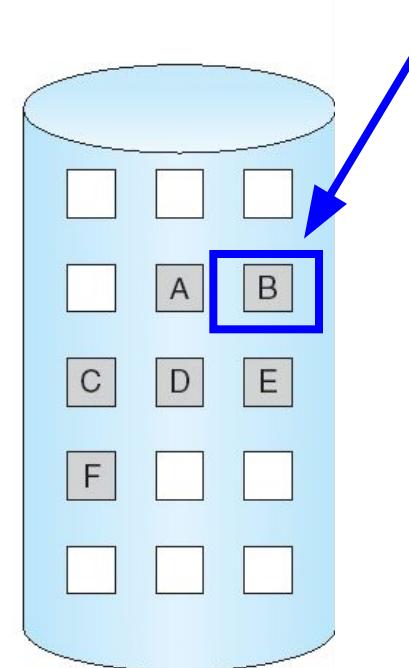
Tecnologia	Tempo de acesso
SRAM	0,5-2,5 ns
DRAM	60-120 ns
Disco	10-20 million ns

# Tabela de Páginas e Falta de Página

Bit válido/inválido informa se página está na memória

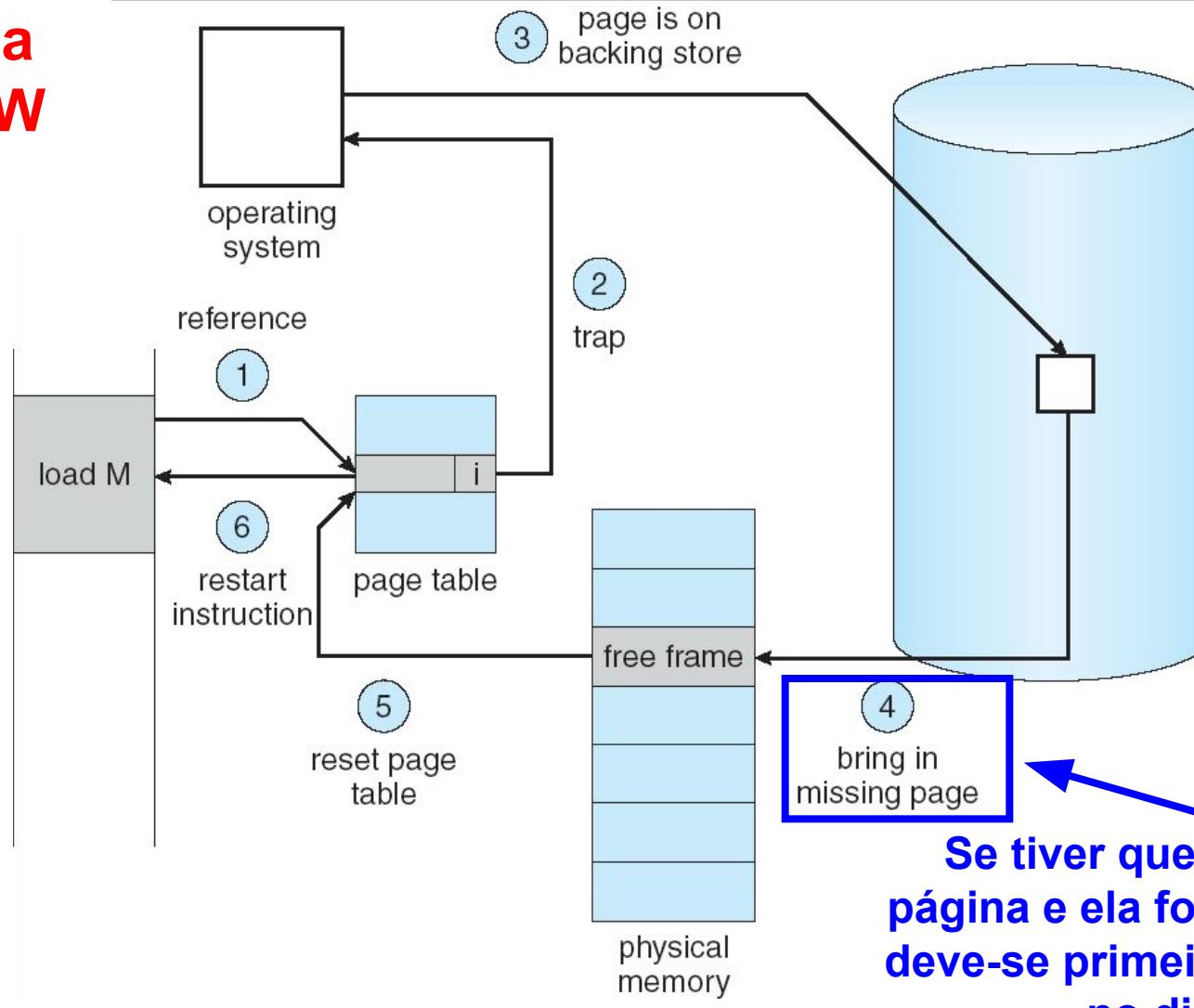


Ocorre falta de página  
-Página 1 está no disco



# Tratando a Falta de Página

Tratada  
pelo SW  
(S.O)



Se tiver que substituir  
página e ela foi modificada,  
deve-se primeiro escrevê-la  
no disco

# MMU e Memória Virtual

- Memory Management Unit (MMU) é um dispositivo de HW que mapeia endereços virtuais em endereços físicos
  - Requisição de CPU passa pela MMU
  - MMU decodifica endereço e utiliza tabela de páginas
  - Avisa falta de páginas a CPU
- MMU auxilia na proteção de processos, detecção de falhas, realocação de endereços, etc (*veremos em breve*)

# Acelerando a Paginação

- Mapeamento de endereços virtuais para endereços físicos deve ser rápido
- Tabela de páginas na memória ☐ acesso ao espaço de endereçamento do processo requer no mínimo dois acessos à memória
  - Um para acessar a tabela
  - Outro para acessar a página

**Solução:**

**Transaction Look-aside Buffer (TLB)!**

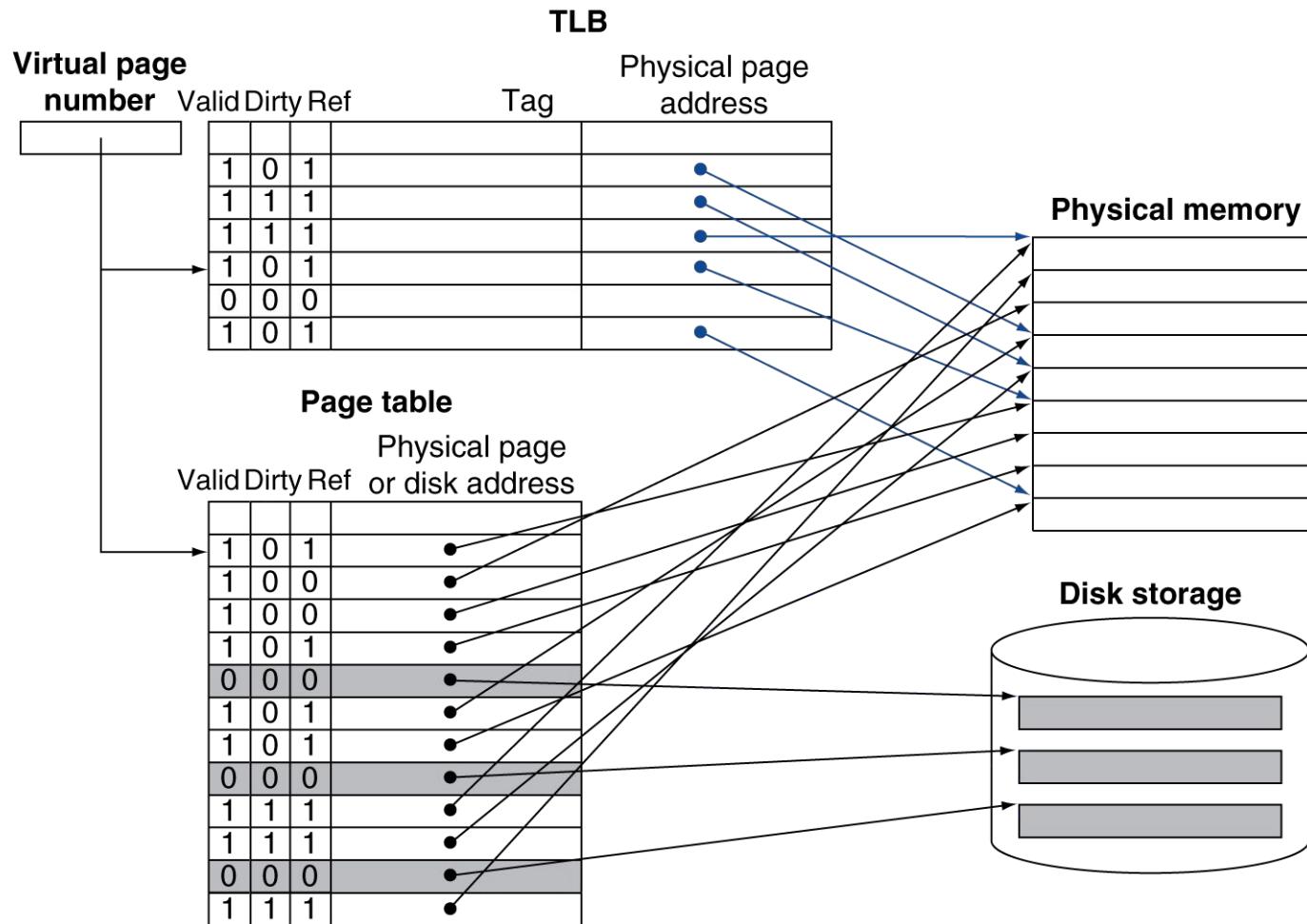
# Transaction Look-aside Buffer (TLB)

- Memória (cache) associativa pequena e rápida
  - Geralmente localizada na MMU
  - Política de substituição de entradas: Randômica
- Contém algumas entradas da tabela de páginas
  - Entradas da tabela usadas mais recentemente
    - Aproveita-se de localidade temporal
  - Funciona como uma cache
- CPU gera um n° de página virtual, que é comparado simultaneamente com todas as entradas da TLB

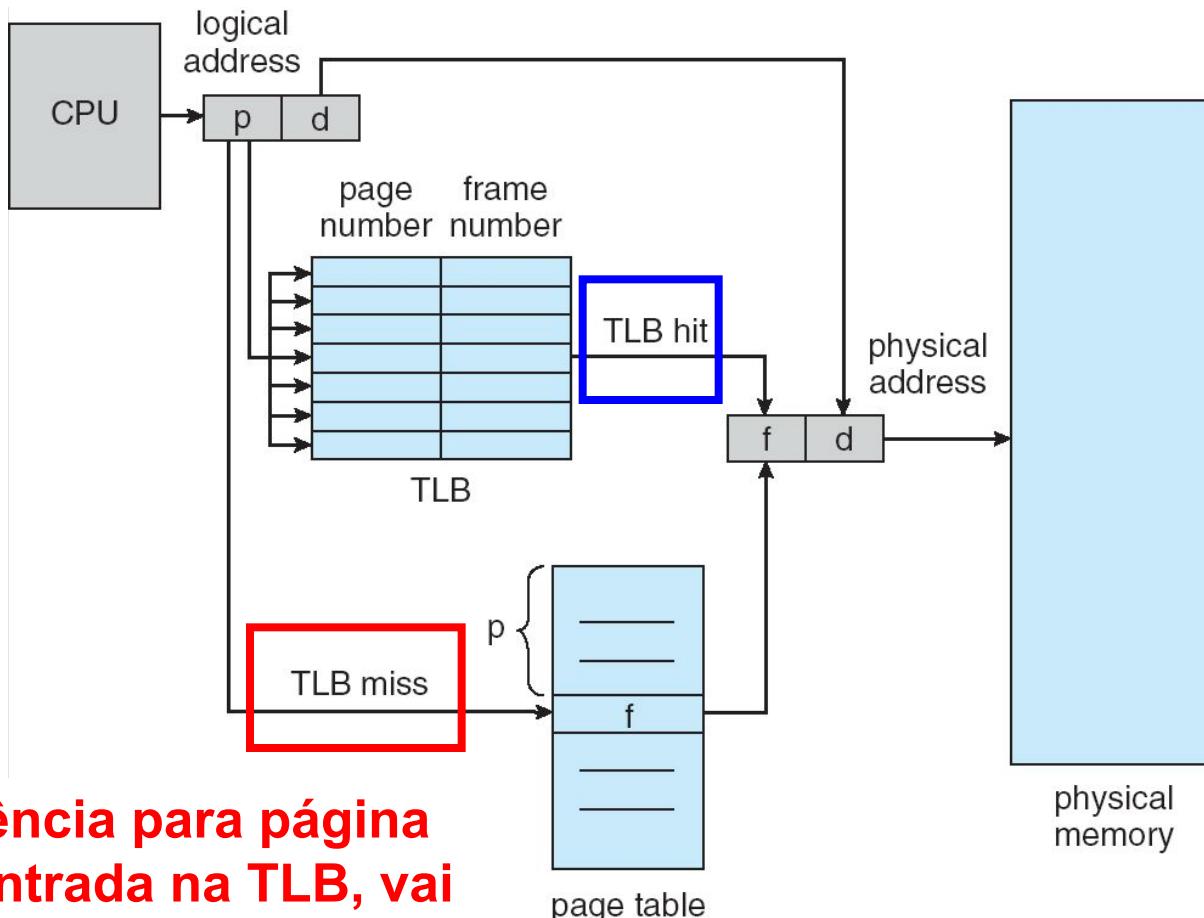
# Valores Típicos de TLBs

- Tamanho : 16 – 512 entradas
- Tamanho do bloco: 1-2 entradas da tabela de páginas
  - Entrada da tabela de páginas: 4- 8 bytes
- Hit time : 0,5 – 1 ciclo
- Miss penalty: 10 – 100 ciclos
- Miss rate: 0,01% - 1%

# Tradução Rápida Usando uma TLB



# Miss (Falta) em TLB



**Se referência para página  
não encontrada na TLB, vai  
para a tabela de páginas**

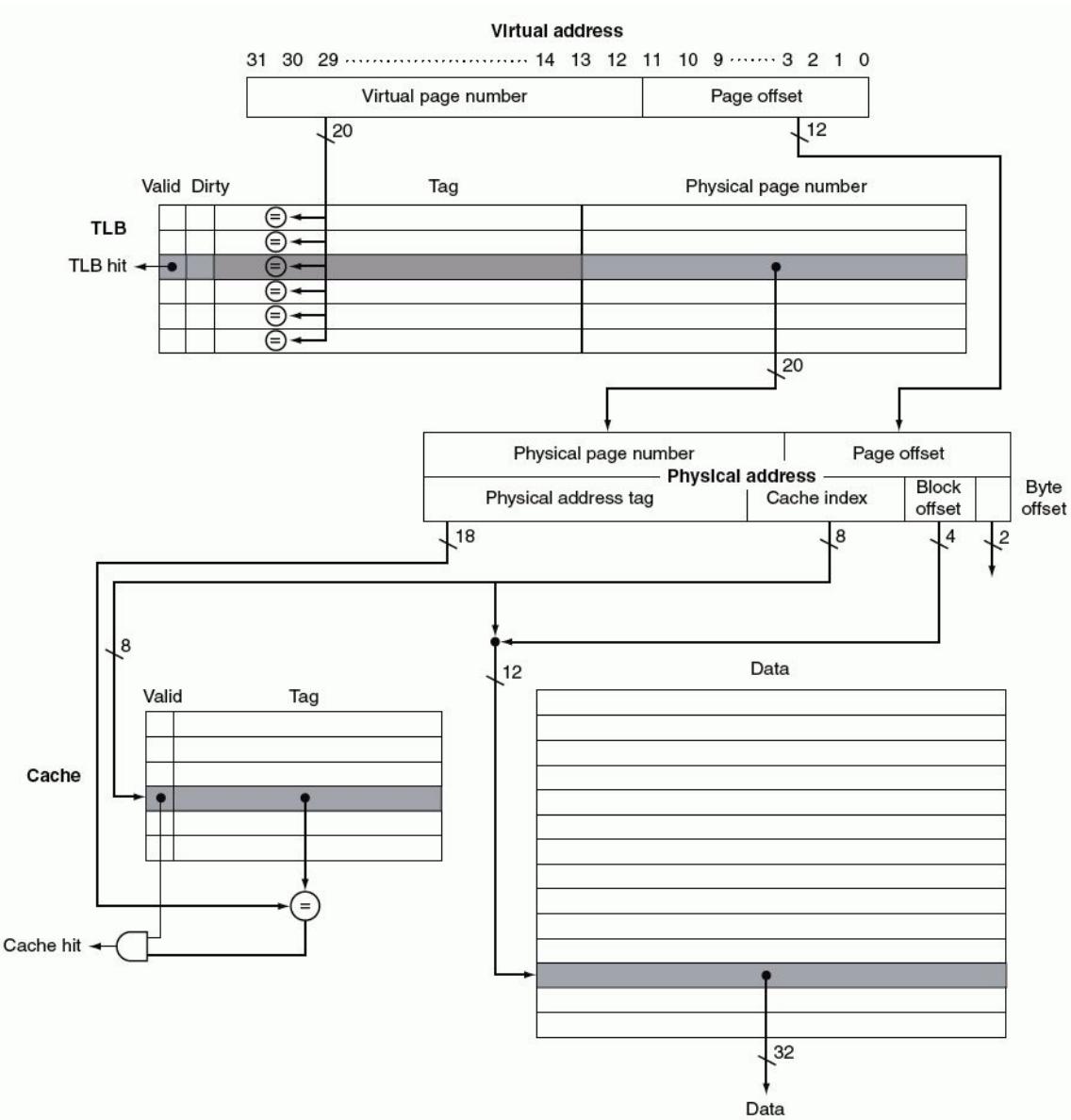
**Se referência para página  
encontrada na TLB, acessa  
página  
[cin.ufpe.br](http://cin.ufpe.br)**

# Tratando um Miss em TLB

- Se página estiver na memória
  - Carrega a entrada da tabela de páginas da memória e tenta novamente
  - Pode ser feita pelo HW (MMU)
  - Ou em SW
    - Levanta exceção, S.O pode tratá-la
- Se página não estiver na memória (page fault)
  - S.O trata, buscando a página e atualizando a tabela de páginas
  - Então reinicia a instrução que causou a falta

# Interação TLB e Cache

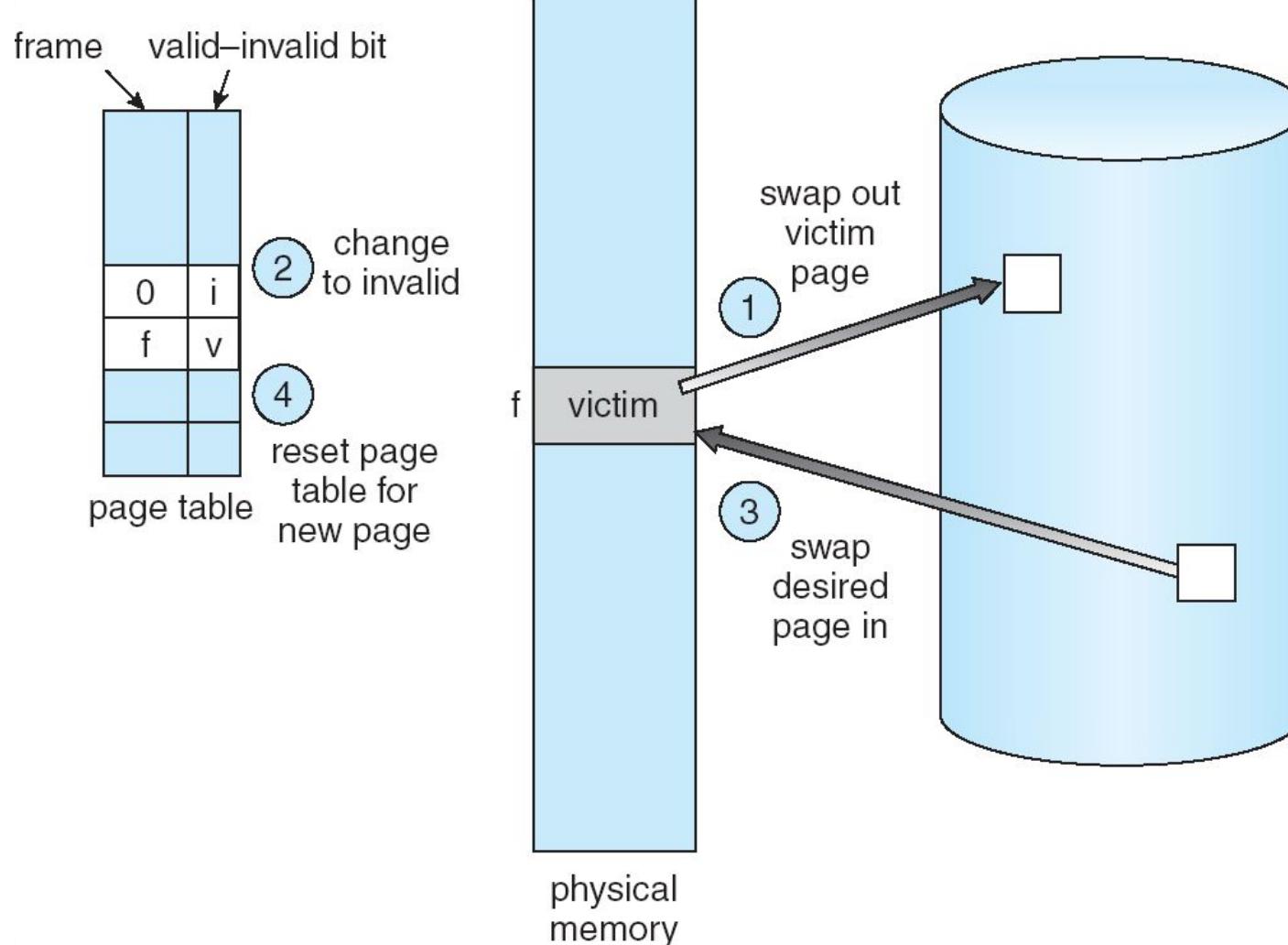
- Se tag da cache utiliza endereço físico
  - Necessidade de traduzir endereço antes de procurar na cache



# Substituição de Página

- Quando ocorre falta de página e não há espaço disponível para carregar página, uma página na memória é removida e substituída por página requisitada
- Página removida, se alterada, é salva em disco
- Escolha das páginas a serem substituídas deve ser feita com cuidado pelo S.O para não provocar uma repetidas faltas de páginas
  - Grande impacto no desempenho do sistema

# Esquema Geral de Substituição de Página



# Algoritmos de Substituição de Página

- S.O se encarrega de substituição de páginas
- Diferentes algoritmos podem ser utilizados para a substituição de páginas:
  - First In First Out (**FIFO**)
  - Least Recently Used (**LRU**) – **Muito Utilizado**
  - Segunda Chance
- Objetivo é reduzir ao máximo o número médio de falta de páginas

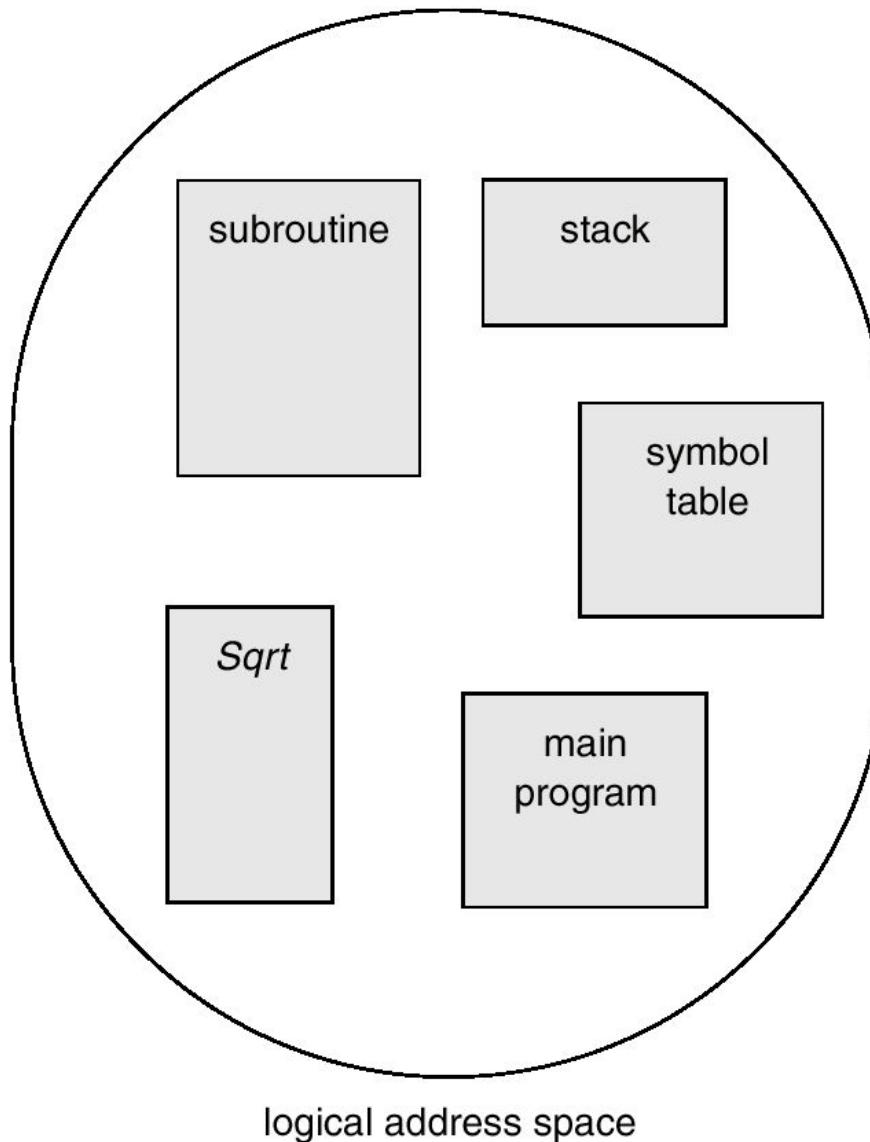
# Resumindo Paginação...

- Localização da página
  - Associativa
- Política de escrita
  - Write-back
- Falta de páginas
  - tratamento por software

# Segmentação

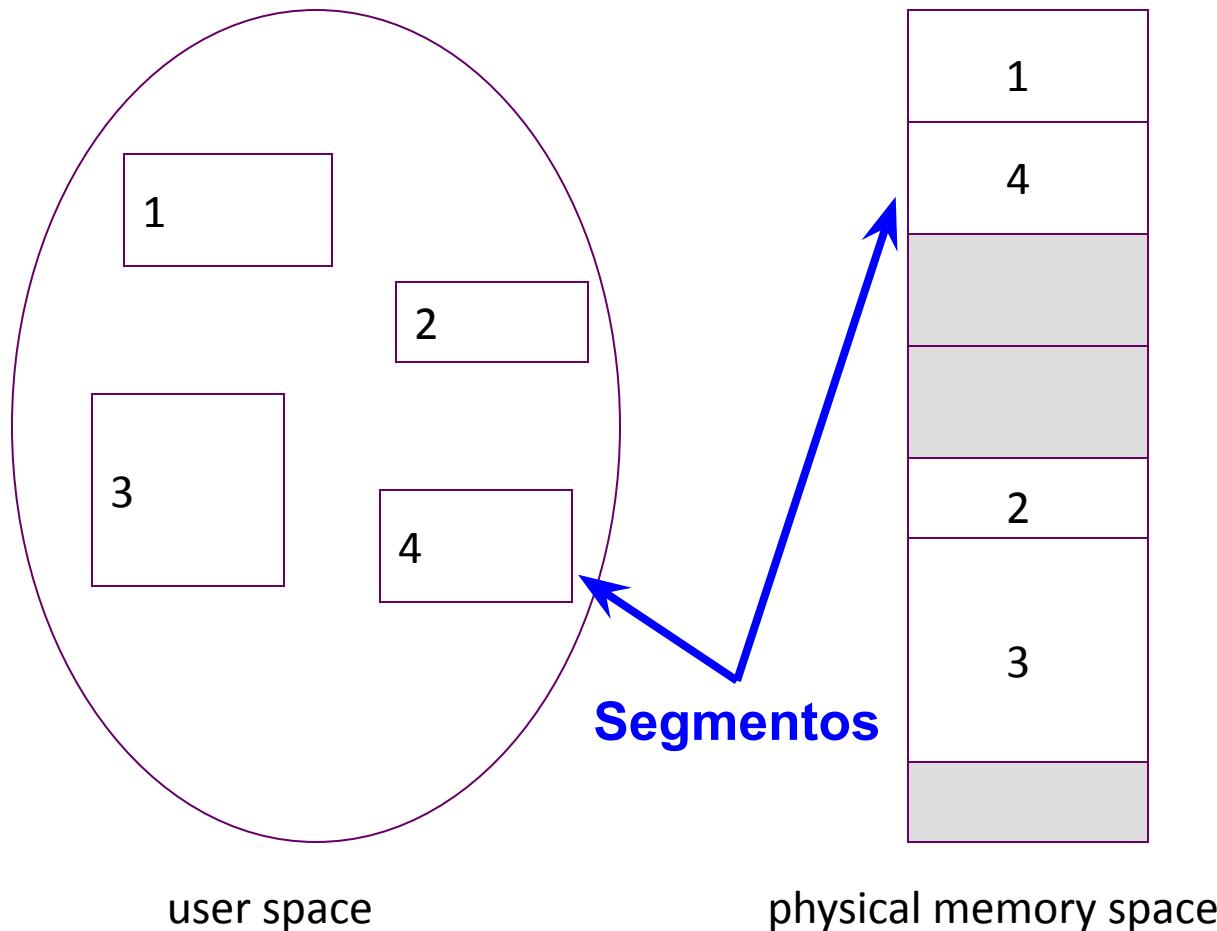
- Segmentação é uma técnica de gerenciamento de memória que permite que programa seja dividido em blocos de tamanho variável chamados de segmentos
  - Paginação → blocos de tamanho fixo (páginas)
  - Segmentação → blocos de tamanho variável (segmentos)
- Idéia é dividir programas em segmentos lógicos
  - Programa principal, funções (métodos), variáveis globais, variáveis locais, classes, etc
- Divisão é feita baseada em como o programador enxerga um programa
- Utilizado em processadores Intel Pentium

# Visão do Programa pelo Usuário



logical address space

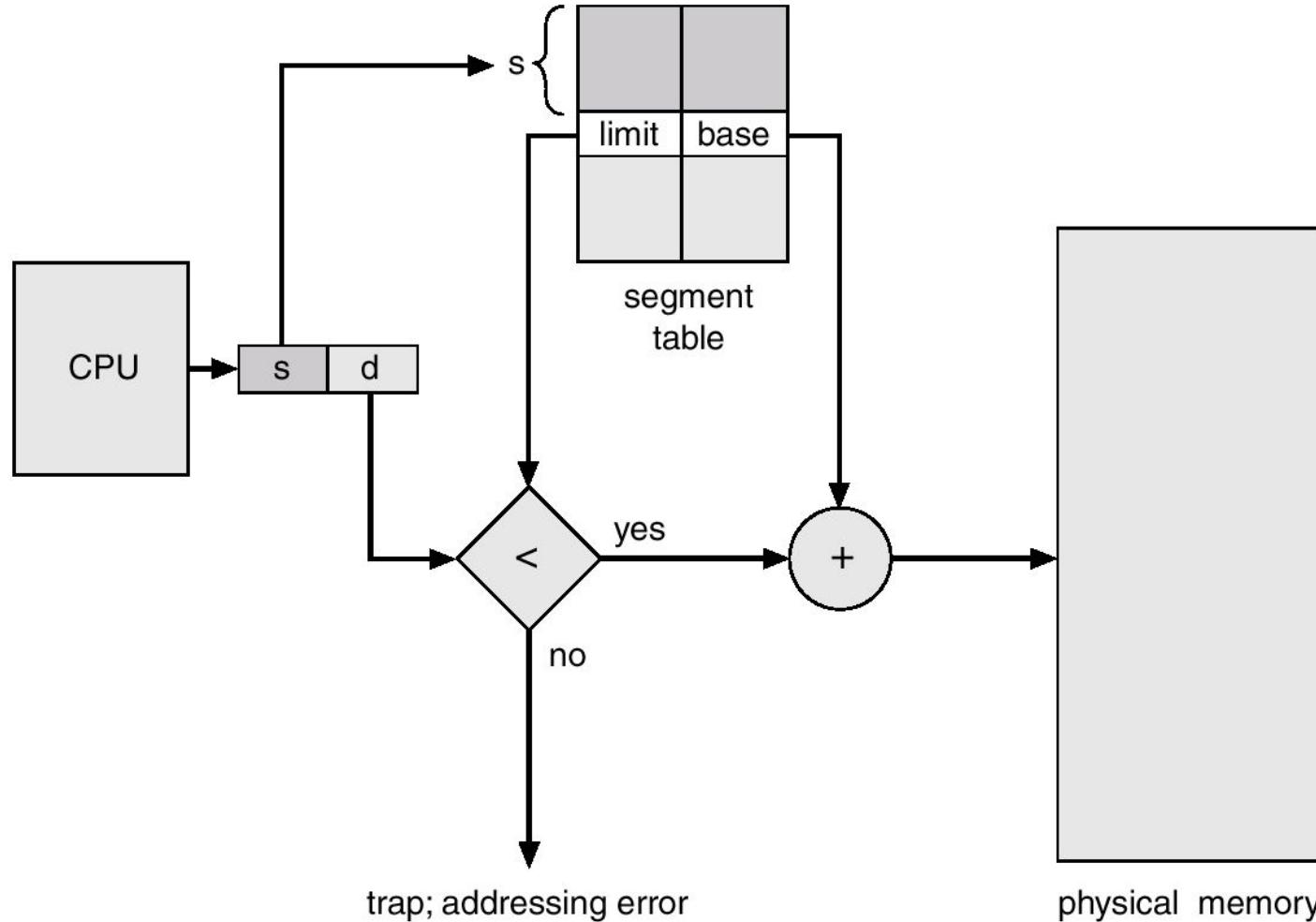
# Idéia Geral de Segmentação



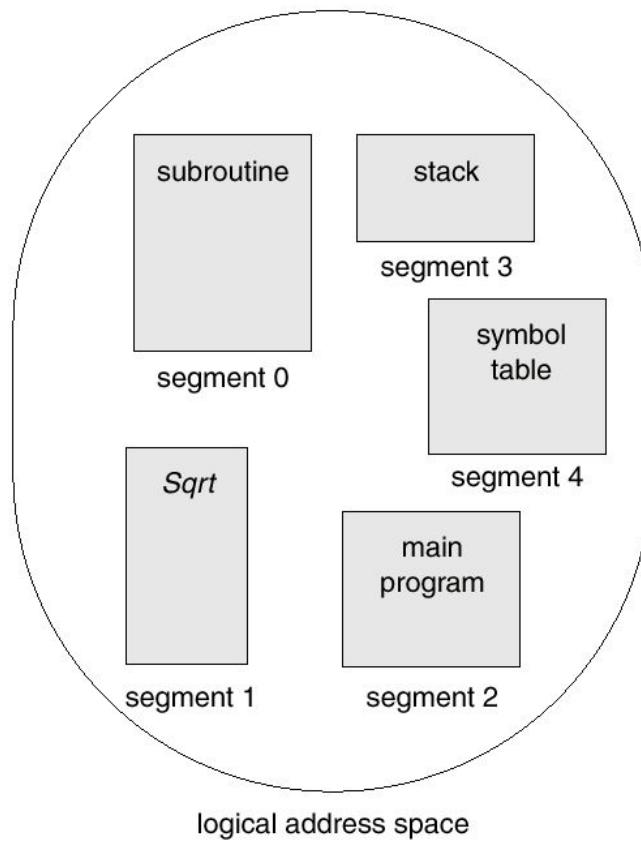
# Implementando Segmentação

- Endereços virtuais consistem de um número de segmento (bits mais significativos) e deslocamento dentro do segmento (bits menos significativos)
- Tabela de segmentos mapeia endereços virtuais em físicos
  - Indexada pelo número do segmento
  - Cada entrada possui endereço físico do começo do segmento e o limite do segmento
  - Entradas possuem bits de controle
    - Leitura/escrita
    - Válido
    - Modificado (dirty)

# Implementação de Segmentação

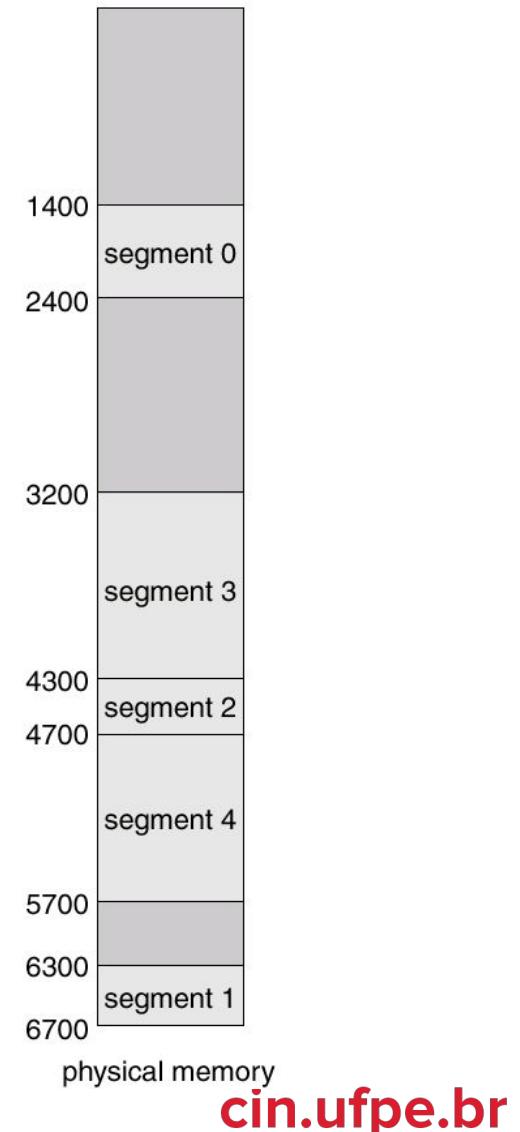


# Exemplo de Segmentação



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



# Proteção de Memória

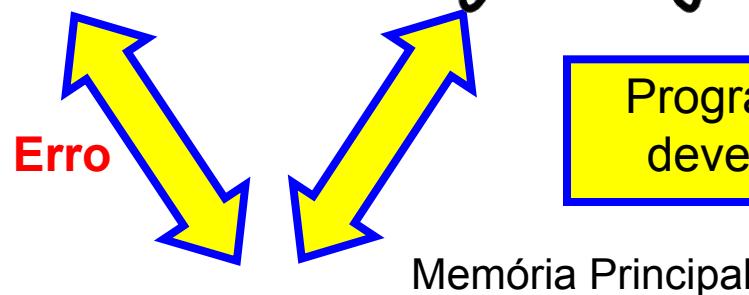
- Processo pode ser carregado em praticamente qualquer endereço de memória
- Diferentes processos podem ter espaços de endereçamento diferentes
- Ou diferentes processos podem compartilhar espaços de endereçamento desde que autorizados
- HW pode verificar se processo tenta acessar espaços de endereçamento inválidos
  - **Proteção de Memória**

# Compartilhamento de Recursos

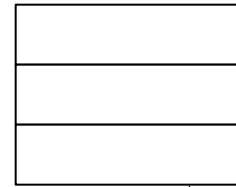


Erro

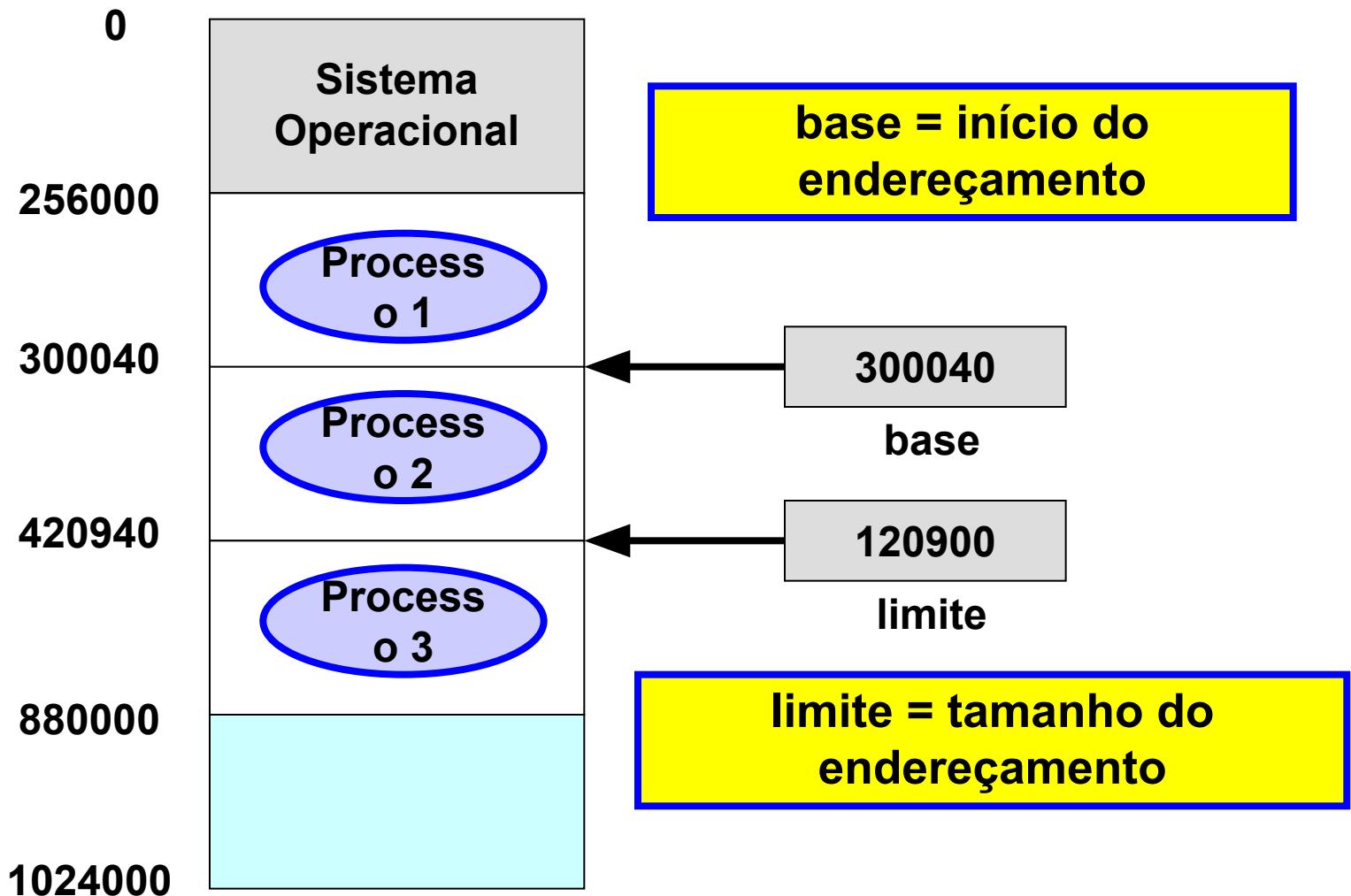
Programa defeituoso não  
deve interferir em outro



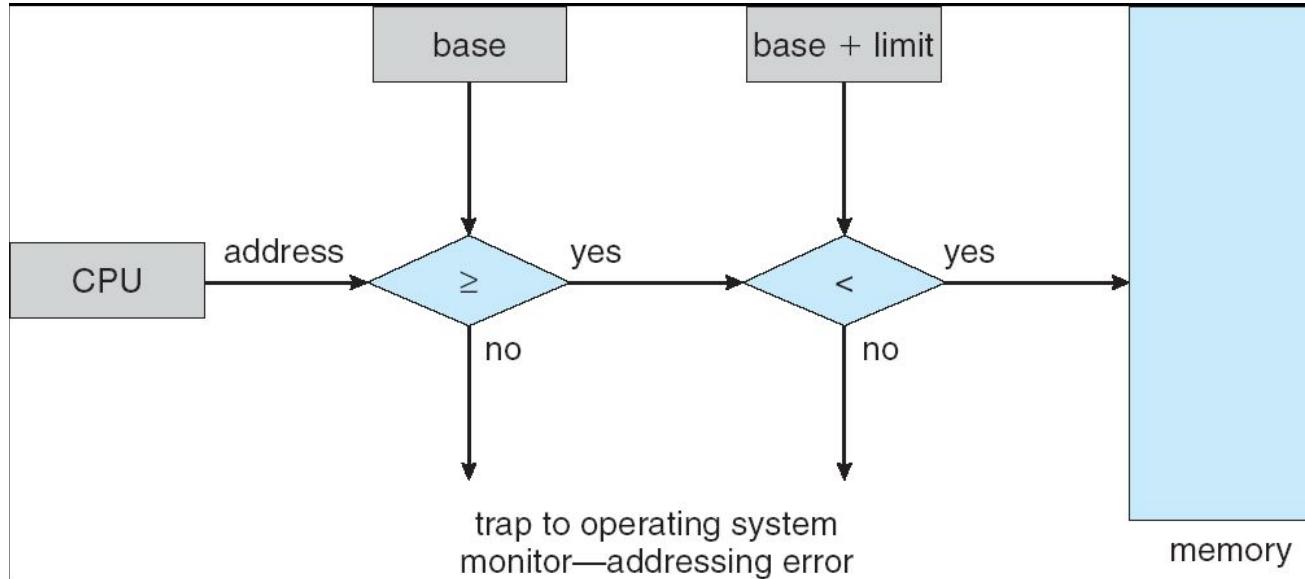
CPU



# Registradores de Base e Limite



# Proteção com Registradores de Base e Limite



- MMU compara endereço gerado por processo com registradores
  - **Se há violação, avisa ao sistema operacional**
- Somente S.O pode carregar os registradores

# Sistemas Operacionais e Operação em Dual-Mode

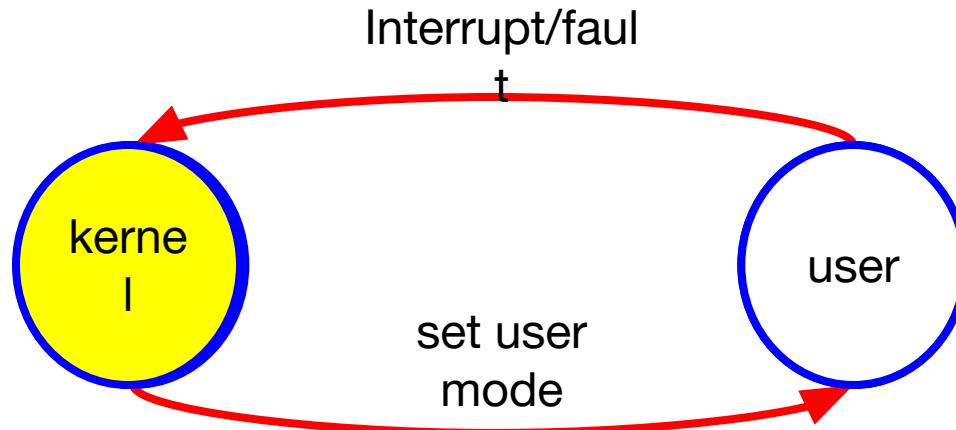
- Compartilhamento de recursos requer um SO para garantir que um programa defeituoso não cause erros em outros programas
- Computadores atuais dão suporte em hardware para executar instruções em dois modos de operação
  - User mode – execução de instrução relativa a um aplicativo do usuário
  - Kernel mode (ou system mode) – execução de uma instrução vinda de um SO

# Sistemas Operacionais e Operação em Dual-Mode

- Hardware possui um bit para indicar o modo em um determinado instante
  - (0) kernel mode
  - (1) user mode
- Instruções de aplicativos não podem ser executados quando o HW está em kernel mode
- Instruções que só podem ser executadas em kernel mode são chamadas de instruções privilegiadas (privileged instructions)
  - Ex: instruções de E/S, acesso a tabela de páginas
  - S.O usa instruções privilegiadas

# Sistemas Operacionais e Operação em Dual-Mode

- Quando um erro ou uma interrupção ocorre, hardware muda para kernel mode



# Sistema Hierárquico

- TLB, Memória e Cache

Cache	TLB	Tabela pag.	Possível? Como?
miss	hit	Hit	
hit	miss	Hit	
miss	miss	Hit	
miss	miss	Miss	
miss	hit	Miss	
hit	hit	Miss	
hit	miss	Miss	

# Infraestrutura de Hardware

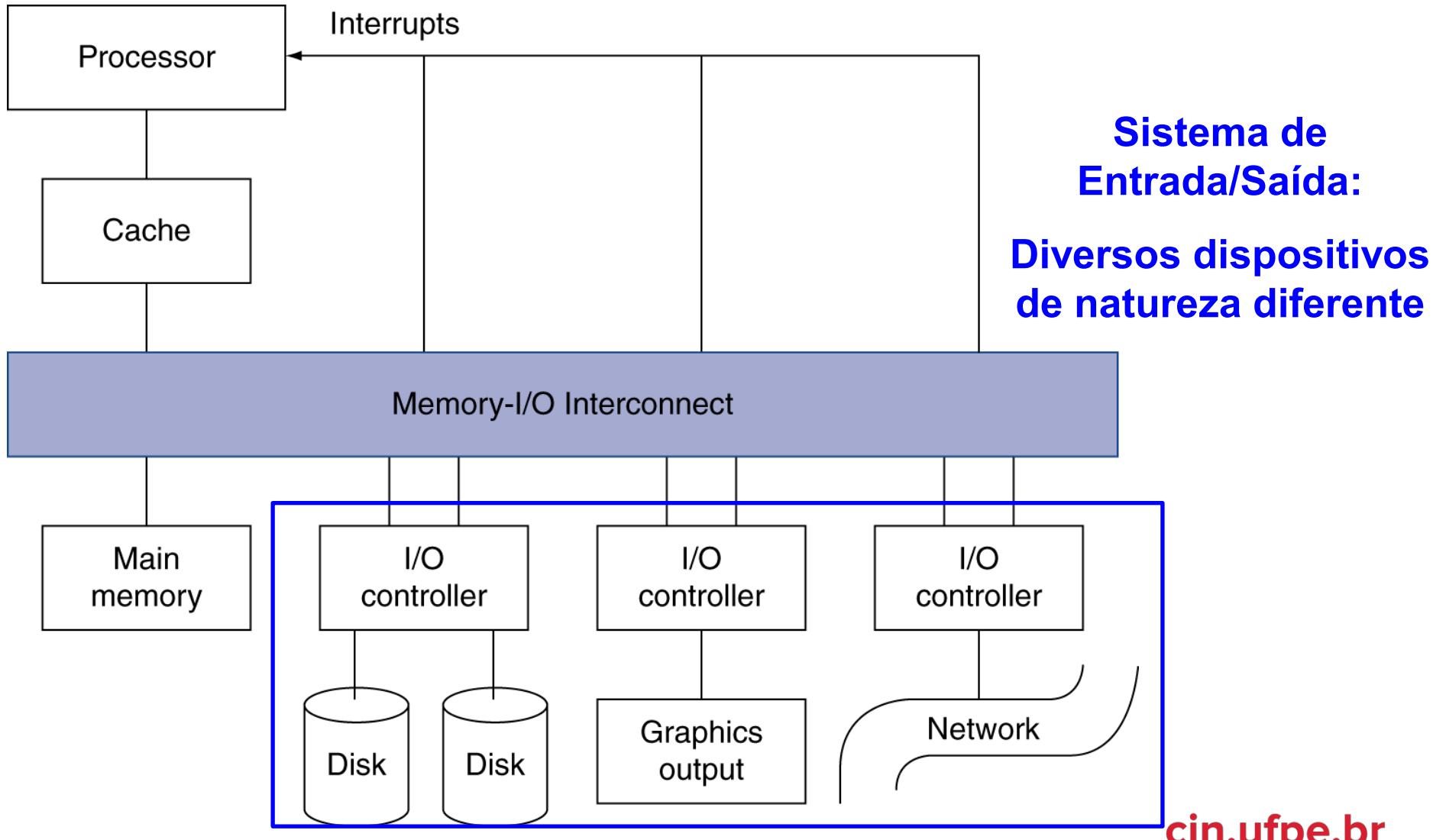
Entrada/Saída: Armazenamento

Prof. Adriano Sarmento

# Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

# Organização Típica de Sistemas Computacionais



# Sistema de E/S e Desempenho do Sistema

- Melhora de desempenho de CPU: 60% por ano
- Desempenho de Sistemas de E/S: Limitado por Delays Mecânicos (disco E/S)
  - Melhora de 10% por ano
- Lei de Amdahl: Speed-up Limitado pelo Sub-Sistema mais lento!
- E/S : Gargalo
  - Reduz a fração do tempo na CPU
  - Reduz o valor de CPUs mais rápidas

# Dispositivos de E/S

- Podem diferir em:
  - Comportamento: entrada, saída, armazenamento
  - Interação: humano ou máquina
  - Taxa de transferência: dados/seg, operações/seg

<b>Device</b>	<b>Behavior</b>	<b>Partner</b>	<b>Data rate (Mb/s)</b>
Keyboard	input	human	0.0001
Mouse	input	human	0.0038
Laser printer	output	human	3.2
Magnetic disk	storage	machine	800-3000
Graphics display	output	human	800-8000
Network/LAN	input or output	machine	100-10000

# Projeto de Sistema de E/S

- Considerações de projeto:
  - possibilidade de expandir o sistema
  - diversidade de dispositivos
  - comportamento no caso de falhas
  - custo
  - Desempenho
- Desktops & sistemas embarcados
  - Importante a diversidade de dispositivos
- Servidores
  - Importante a expansibilidade de dispositivos e tolerância a falhas

# Desempenho de Sistema de E/S

- Diferentes métricas, depende da aplicação
  - tempo de resposta(latência)
  - taxa de transferência (throughput)
    - Quantidade de dados transferidos
    - Quantidade de operações de E/S
- Desktops & sistemas embarcados
  - Importante a latência
- Servidores
  - Importante o throughput
- Difícil de medir, dependências:
  - características do dispositivo
  - conexão com o sistema
  - hierarquia de memória
  - sistema operacional (software de I/O)

# Dependabilidade (Dependability)

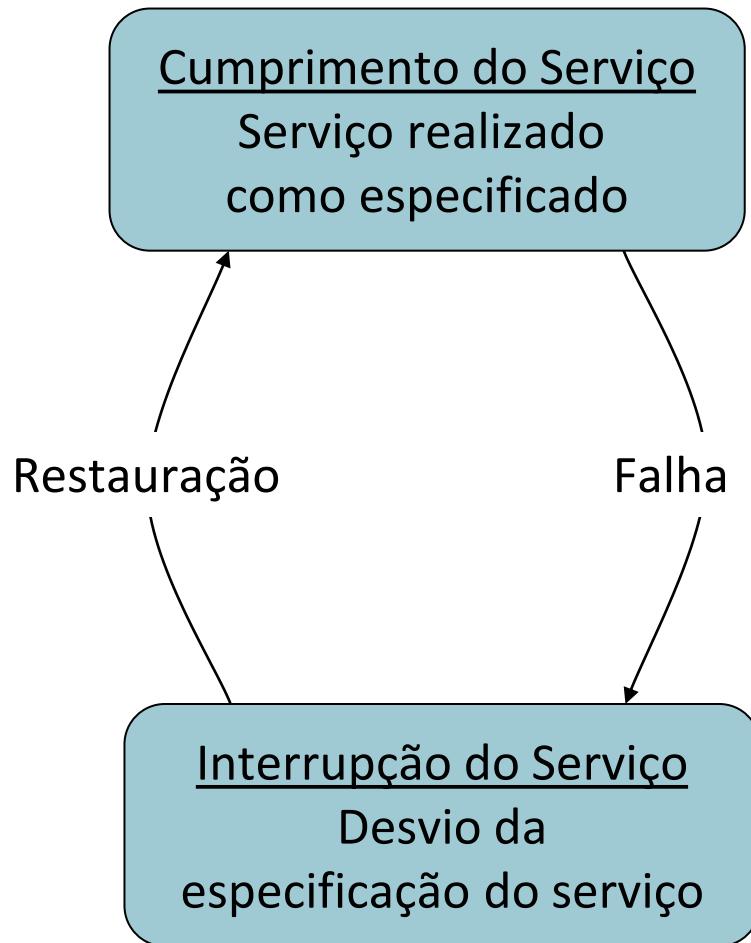
- Dependabilidade é a propriedade que define a capacidade dos sistemas computacionais de prestar um serviço que se pode justificadamente confiar
- Atributos importantes:
  - Confiabilidade (reliability)
  - Segurança
  - Disponibilidade (availability)
  - Mantenabilidade
- Dependabilidade é importante em um sistema
  - **Particularmente para dispositivos de armazenamento**

# Impacto da Dependabilidade em Um Sistema

Application	Cost of downtime per hour (thousands of \$)	Annual losses (millions of \$) with downtime of		
		1% (87.6 hrs/yr)	0.5% (43.8 hrs/yr)	0.1% (8.8 hrs/yr)
Brokerage operations	\$6450	\$565	\$283	\$56.5
Credit card authorization	\$2600	\$228	\$114	\$22.8
Package shipping services	\$150	\$13	\$6.6	\$1.3
Home shopping channel	\$113	\$9.9	\$4.9	\$1.0
Catalog sales center	\$90	\$7.9	\$3.9	\$0.8
Airline reservation center	\$89	\$7.9	\$3.9	\$0.8
Cellular service activation	\$41	\$3.6	\$1.8	\$0.4
Online network fees	\$25	\$2.2	\$1.1	\$0.2
ATM service fees	\$14	\$1.2	\$0.6	\$0.1

Figure 1.3 The cost of an unavailable system is shown by analyzing the cost of downtime (in terms of immediately lost revenue), assuming three different levels of availability, and that downtime is distributed uniformly. These data are from Kembel [2000] and were collected and analyzed by Contingency Planning Research.

# Dependabilidade e Falhas



- **Falha (Fault)** : componente não executa como especificado
  - Pode ser permanente ou transiente
  - Pode ou não provocar a falha do sistema

# Medindo a Dependabilidade

- Confiabilidade: Mean Time To Failure (MTTF).
- Interrupção de serviço : Mean Time To Repair (MTTR)
- Disponibilidade:  $MTTF/(MTTF + MTTR)$
  
- Melhorando a disponibilidade
  1. Aumentando MTTF
    - Fault avoidance
    - Fault tolerance
    - Fault forecasting
  - Diminuindo MTTR
    - Melhorando ferramentas e processos de diagnóstico e reparação

# Armazenamento Secundário

- Importante para garantir dependabilidade de um sistema
  - Memória e cache → voláteis
  - Armazenamento secundário → dados persistidos
- Tem impacto no desempenho do sistema
  - Faltas na cache e memória → acesso a disco
  - Gargalo no tempo de execução
- Desafios:
  - Como aumentar dependabilidade?
  - Como aumentar desempenho?

# Tipos de Armazenamento Secundário

## ■ Disco

- Dispositivo magnético
  - Partes gravadas são magnetizadas
- Possui componentes mecânicos



## ■ Flash

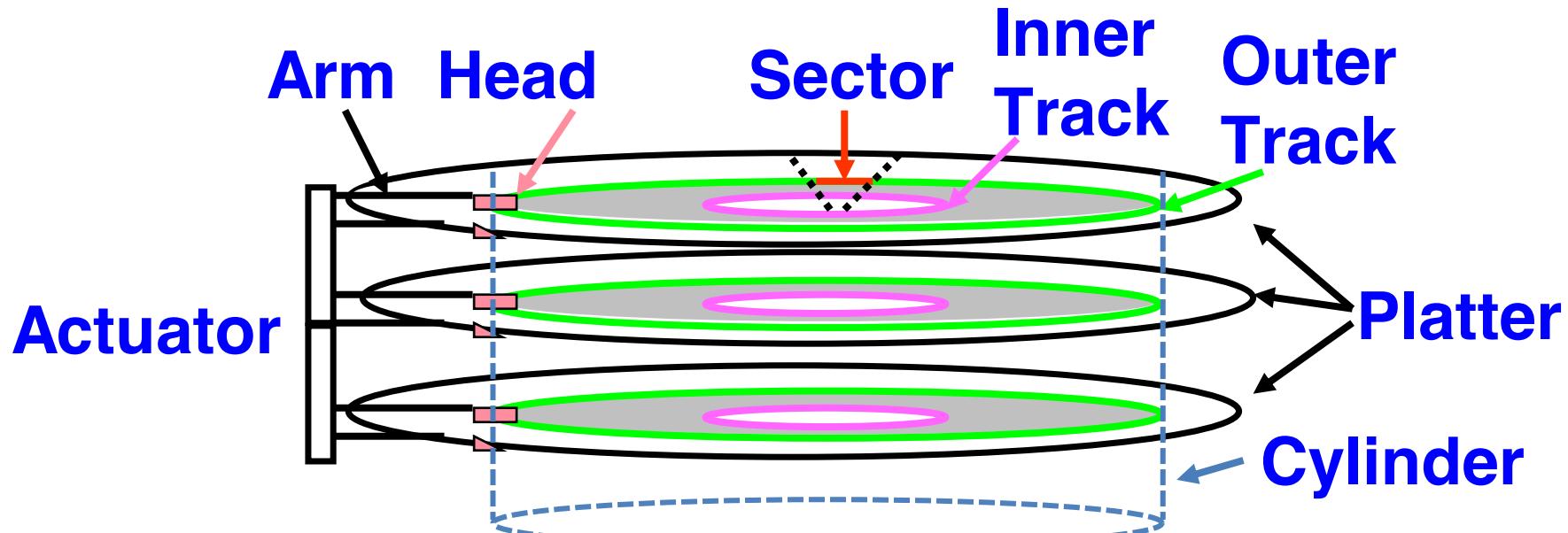
- Memória semicondutora (EEPROM)
- Wearable
  - Após 100000-1000000 de escritas pode perder capacidade de armazenamento
  - Wear leveling : Redistribui dados em áreas menos usadas



# Disco x Flash

- Velocidade de acesso
  - Flash 100-1000 mais rápida do que disco
- Preço
  - Disco até 2 a 3x mais barato (2021)
    - Mais popular em desktops e servidores
- Wearable
  - Disco sem limite para escrita
- Energia e Robustez
  - Flash mais eficiente e robusto
    - Mais popular em sistemas embarcados

# Disco Rígido: Terminologia



- Vários pratos, com a informação armazenada magneticamente em ambas superfícies (usual)
- Bits armazenados em trilhas, que por sua vez são divididas em setores (e.g., 512 Bytes)
- Atuador move a cabeça (fim do braço, 1/superfície) sobre a trilha ("seek"), seleciona a superfície, espera pelo setor passar sob a cabeça, então lê ou escreve
  - “Cilindro”: todas as trilhas sob as cabeças

# Setores do Disco e Acesso

- Cada setor armazena
  - ID do setor
  - Dados (512 bytes, 4096 bytes nos discos novos)
  - Error correcting code (ECC)
    - Usado para esconder defeitos e erros de leitura
  - Campos de sincronização e espaços
- Acesso a setores envolve
  - Delays devidos a espera se outros acessos foram feitos antes
  - Procura (Seek): mover as cabeças
  - Latência rotacional
  - Transferência de dados
  - Overhead do controlador

# Calculando o Desempenho de um Disco

**Disk Latency = Seek Time + Rotation Time + Transfer  
Time + Controller Overhead**

- Seek Time
  - Depende do no. de trilhas e velocidade de **seek** do disco
  - Fabricantes anunciam seek time “pessimistas”
    - Seek time real entre 25% - 33% do anunciado
- Rotation Time(Latency)
  - Tempo para o setor girar embaixo da cabeça
    - depende da velocidade de rotação do disco
    - Tempo médio =  $0,5 \text{ rotação/velocidade de rotação}$
- Transfer Time
  - depende do tamanho do setor, densidade dos bits por trilha, tamanho da requisição, taxa de transferência

# Exemplo : Cálculo de Acesso ao Disco

**Problema:**

**Tamanho do Setor = 512 bytes**

**Velocidade de rotação = 15000 rpm**

**Tempo médio de seek = 4ms**

**Taxa de transferência = 100MB/s**

**Overhead do controlador = 0,2ms**

**Considerando que o disco está desocupado e tempo real de seek é 25% do anunciado, calcule o tempo de acesso ao disco**

# Exemplo : Cálculo de Acesso ao Disco

**Problema:**

**Tamanho do Setor = 512 Bytes = 0,5 KB**

**Velocidade de rotação = 15000 rpm**

**Tempo médio de seek = 4ms**

**Taxa de transferência = 100MB/s**

**Overhead do controlador = 0,2ms**

**Calculando tempo, utilizando seek anunciado:**

**Disk Latency = Seek time + Rotation time + Transfer time + Controller Overhead**

**Disk Latency = 4ms + 0,5 rotacao/15000 rotacao/60 + 0,5KB/100MB/s + 0,2ms**

**Disk Latency = 4ms + 2ms + 0,005ms + 0,2ms = 6,205ms**

# Exemplo : Cálculo de Acesso ao Disco

**Problema:**

**Tamanho do Setor = 512 Bytes = 0,5 KB**

**Velocidade de rotação = 15000 rpm**

**Tempo médio de seek = 4ms**

**Taxa de transferência = 100MB/s**

**Overhead do controlador = 0,2ms**

**Calculando tempo, utilizando seek real:**

**Disk Latency = Seek time + Rotation time + Transfer time + Controller Overhead**

**Disk Latency =  $4\text{ms} \times 0,25 + 0,5 \text{ rotacao}/15000 \text{ rotacao}/60 + 0,5\text{KB}/100\text{MB/s} + 0,2\text{ms}$**

**Disk Latency =  $1\text{ms} + 2\text{ms} + 0,005\text{ms} + 0,2\text{ms} = 3,205\text{ms}$**

# Melhorando Desempenho: Localidade

- Fabricantes anunciam tempo (pior caso) de seek
  - Baseado em todos os seek possíveis
  - Localidade e escalonamento de acessos pelo S.O podem diminuir tempo de seek
    - Depende de aplicação e de algoritmo de escalonamento

# Melhorando Desempenho: Controladores Inteligentes

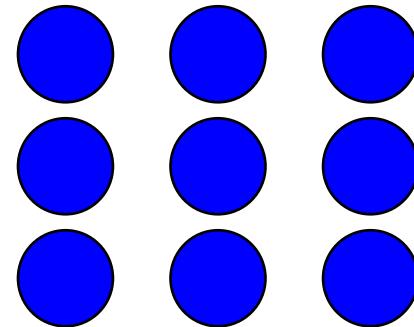
- Controladores com microprocessadores
  - Podem otimizar desempenho
  - Apresentam interface alto nível permitindo que S.O enxergue blocos lógicos e escalone acessos
  - Interfaces padrões (comandos, protocolos, interface elétrica) que permite conexão e transferências de dados entre computadores e periféricos
    - SCSI (Small Computer Systems Interface)
    - SATA (Serial Advanced Technology Attachment )

# Melhorando Desempenho: Controladores Inteligentes

- Controladores incluem caches
  - Mantém dados acessados recentemente
  - Controlador implementa algoritmos que fazem a busca antecipada de setores que tem probabilidade maior de serem buscados
  - Evita seek e latência rotacional

# Melhorando Desempenho : RAIDs:

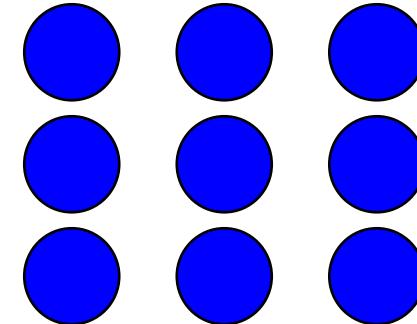
Redundant Array of  
Inexpensive Disks



- Arrays de discos pequenos e baratos
  - Paralelismo aumenta desempenho
    - Dado espalhado sobre múltiplos discos
    - Acessos múltiplos são feitos a vários discos simultaneamente

# Melhorando Dependabilidade : RAIDs:

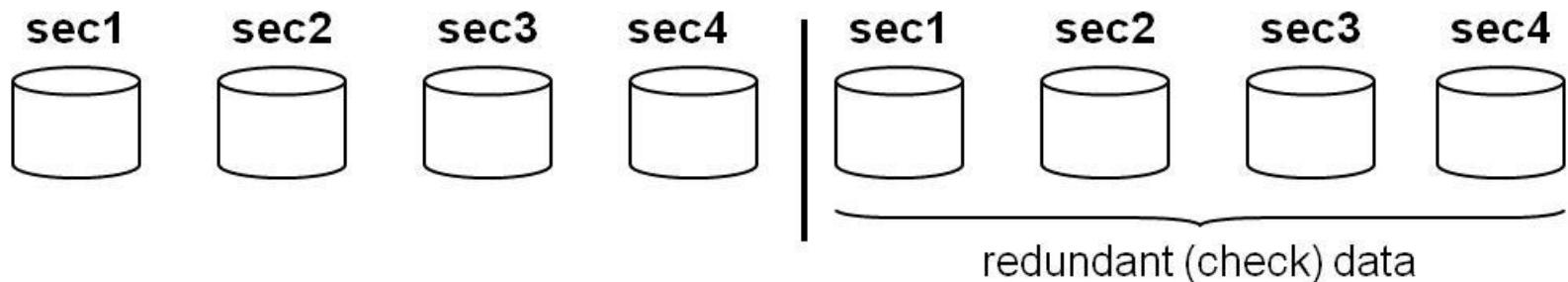
Redundant Array of  
Inexpensive Disks



- Confiabilidade < disco UNICO
- MAS disponibilidade pode ser melhorada pela adição de discos redundantes(RAID)
  - Informação perdida pode ser recuperada através da informação redundante → tolerância a falhas

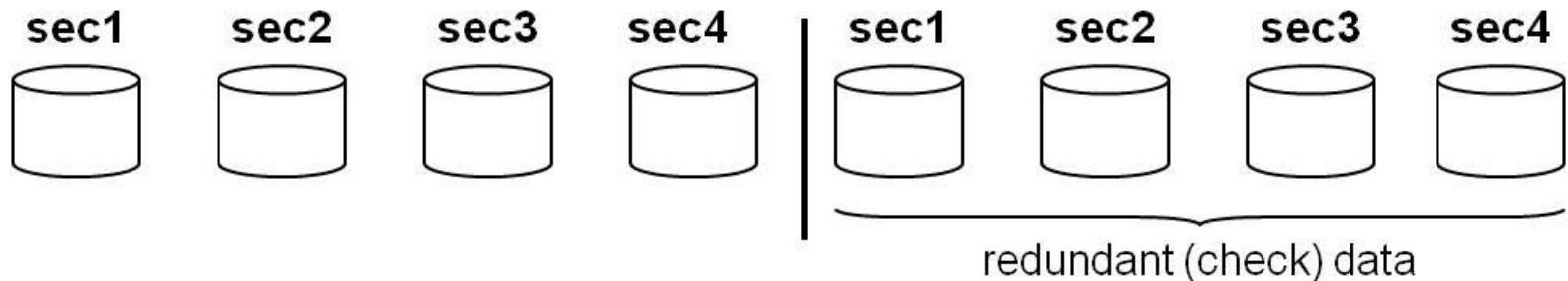
# RAID Nível 1

- RAID 1: Mirroring (Espelhamento)
  - $N + N$  discos → dados replicados
  - Escrita deve ser feita nos discos de dados e nos redundantes
    - Em caso de falha, leitura do espelho



# RAID Nível 10 ( 1 + 0)

- RAID 10: Mirroring (Espelhamento)
  - Distribuição de blocos sobre discos múltiplos— **striping**
    - Múltiplos blocos podem ser acessados em paralelo aumentando o desempenho
  - $N + N$  discos → dados replicados
  - Escrita deve ser feita nos discos de dados e nos redundantes
    - Em caso de falha, leitura do espelho



# RAID Nível 3: Bit-Interleaved Parity

## ■ N + 1 discos

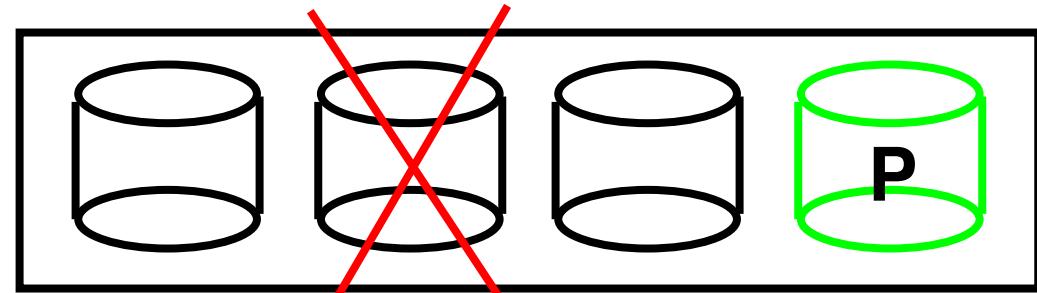
- Dados espalhados em N discos em nível de byte
- Disco redundante armazena paridade
  - Não é necessário armazenar todo o dado
- Leitura
  - Leitura em todos os discos
- Escrita
  - Gera nova paridade e atualiza todos os discos
- Em uma falha
  - Usa paridade para reconstruir dado

# RAID Nível 3: Disco de Paridade

10010011
11001101
10010011
...

registro lógico

Striped registros  
físicos



1	1	1	1
0	1	0	1
0	0	0	0
1	0	1	0
0	1	0	1
0	1	0	1
1	0	1	0
1	1	1	1

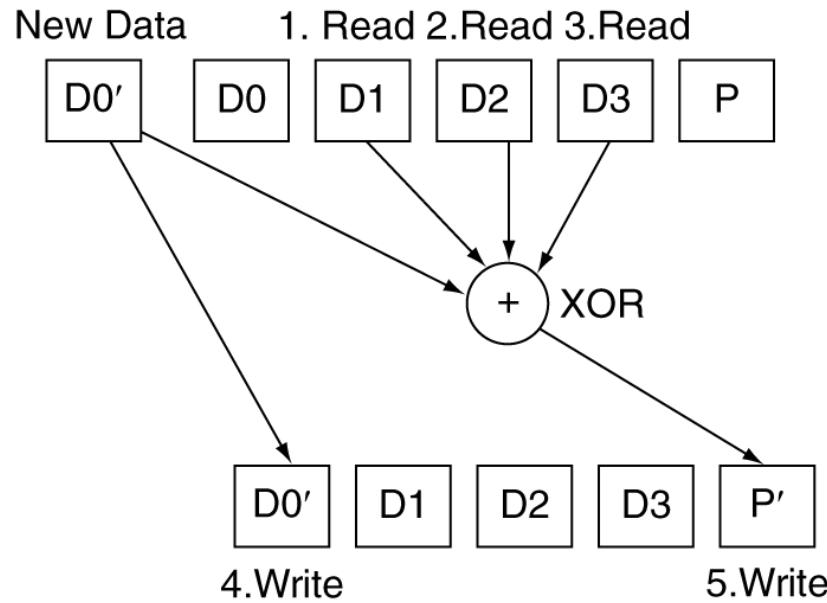
P contém a soma  
dos discos por stripe mod 2  
("parity").  
Se disco falha,  
basta subtrair P  
da soma dos outros  
discos para recuperar  
informação

# RAID Nível 4: Block-Interleaved Parity

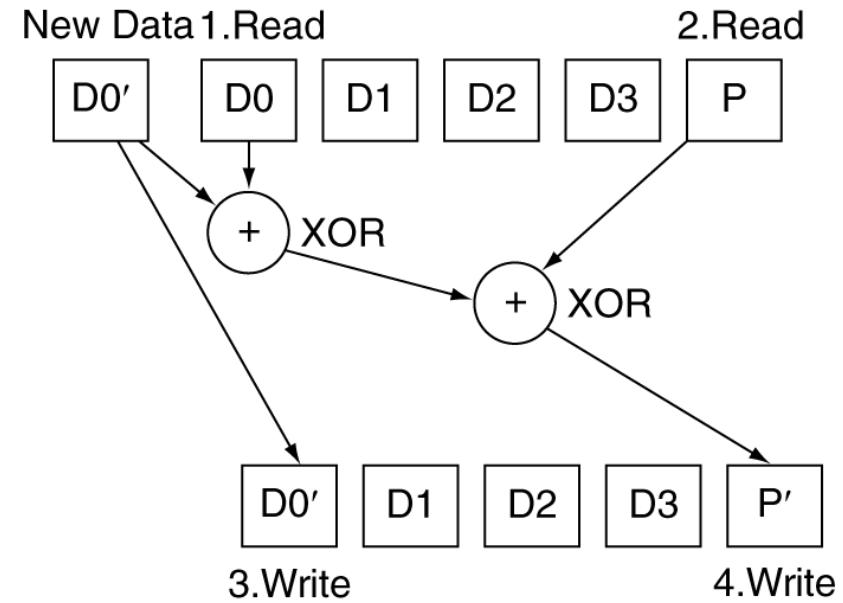
- **N + 1 discos**
  - Dados espalhados em N discos em nível de bloco
  - Disco redundante armazena paridade para um grupo de blocos
  - Leitura
    - Leitura apenas no disco contendo o bloco
  - Escrita (small writes)
    - Escrita no disco contendo bloco modificado e disco de paridade
    - Gera nova paridade e atualiza disco de dado e de paridade
  - Em uma falha
    - Usa paridade para reconstruir dado

# RAID Nível 3 x Raid Nível 4

## RAID 3



## RAID 4

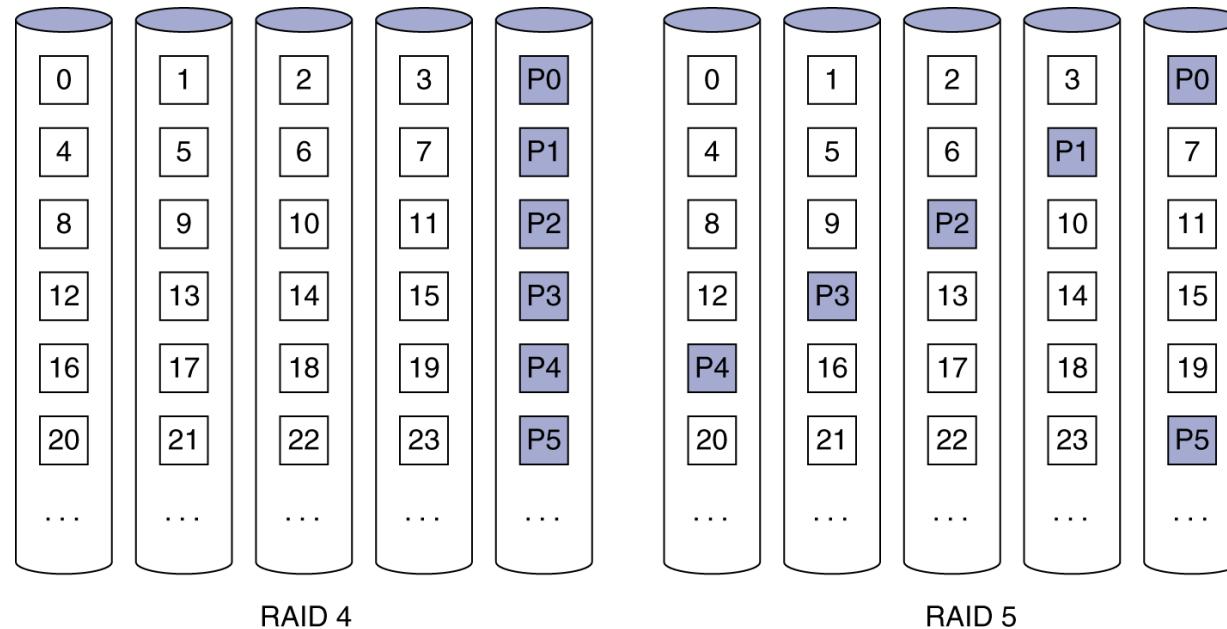


Como a escrita geralmente é feita pelo menos em um bloco (com múltiplos bytes) requer acesso em todos os discos

Escrita em um bloco requer acesso apenas no disco atualizado e o de paridade

# RAID Nível 5: Paridade Distribuída

- $N + 1$  discos
  - Parecido com RAID 4, mas blocos de paridade distribuídos pelos discos do RAID
    - Evita que disco de paridade seja gargalo
    - Permite escritas simultâneas a diferentes discos



# Resumindo RAID...

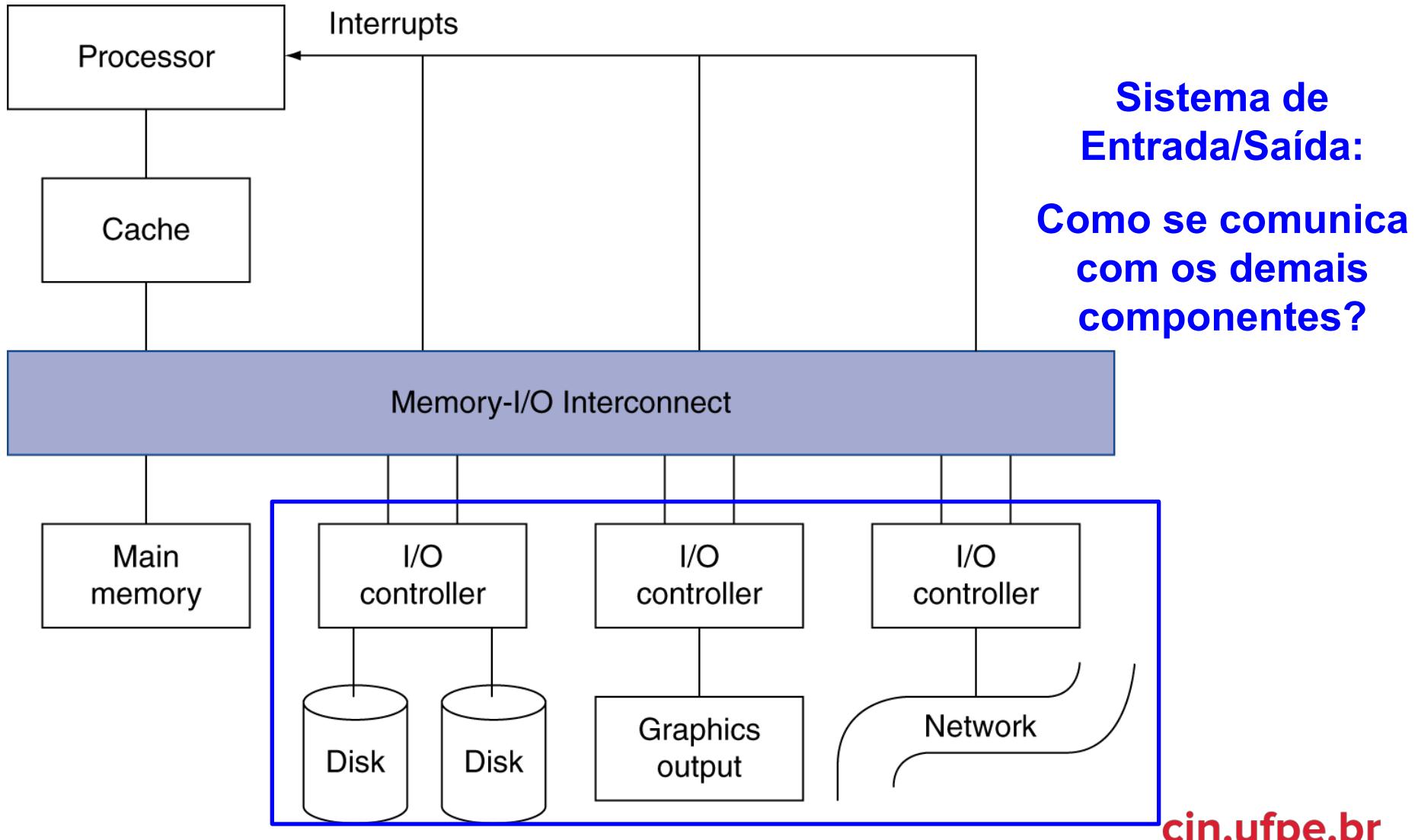
- RAID pode melhorar desempenho e dependabilidade
  - Acessos paralelos
  - Tolerância a falhas
- RAID 10 e RAID 5 muito utilizados
- RAID muito usado em servidores
  - Dependabilidade é essencial

# Infraestrutura de Hardware

Entrada/Saída: Comunicação  
Processador, Memória e E/S

Prof. Adriano Sarmento

# Organização Típica de Sistemas Computacionais



# Comunicação entre Componentes

- Barramento: canal de comunicação compartilhado, geralmente com uma interface padrão
  - Conjunto de fios paralelos para transferência de dados e sincronização desta transferência
- Redes de E/S: conexões seriais ponto-a-ponto de alta velocidade com switches
  - Alternativa mais recente
  - Sistemas Embarcados: NoC (Network on Chip)

# Barramento

## ■ Vantagens

- Flexibilidade
  - novos dispositivos podem ser adicionados facilmente e podem até ser movidos para outros computadores que utilizarem o mesmo padrão de barramento
- Baixo custo
  - um único conjunto de fios é compartilhado

## ■ Desvantagem

- Gargalo de comunicação
  - Largura de banda limita o throughput de E/S

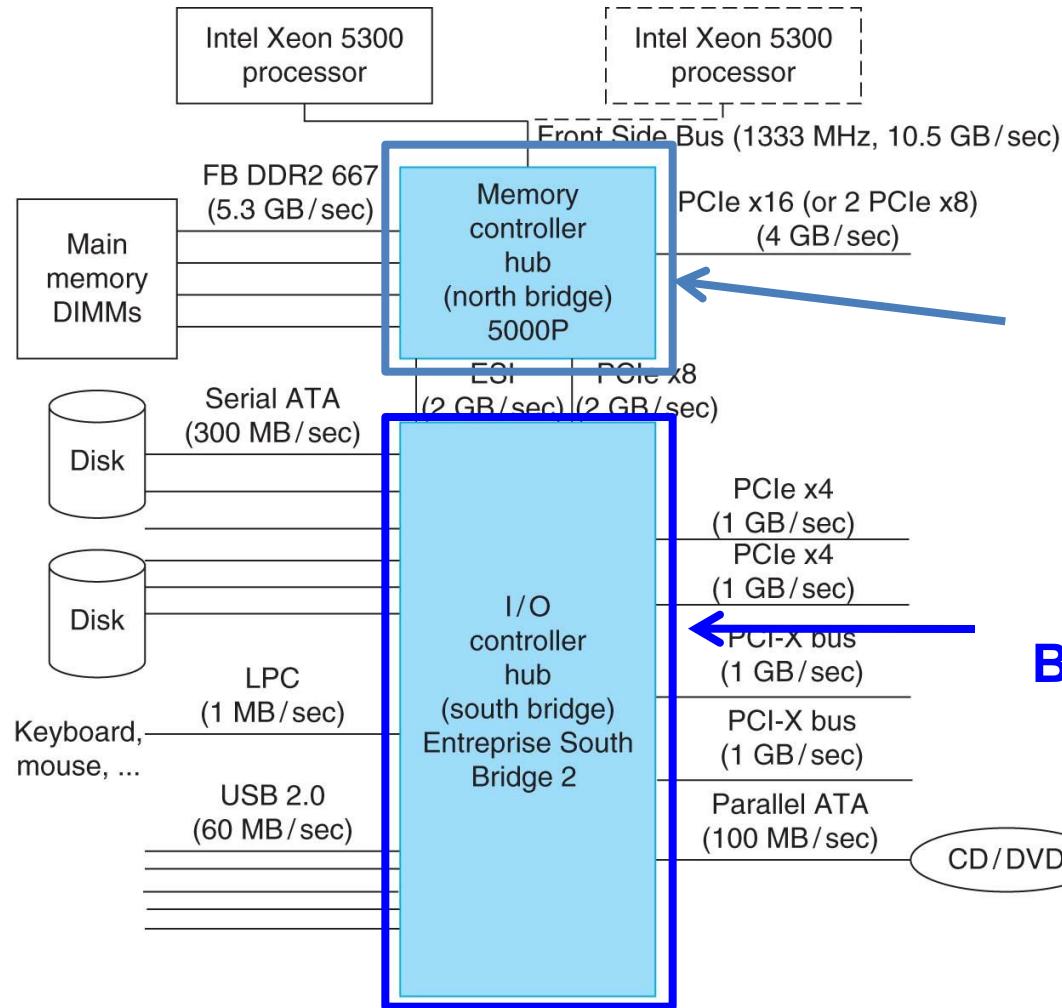
## ■ Velocidade máxima do barramento limitado por:

- Comprimento do barramento
- Número de dispositivos conectados

# Tipos de Barramentos

- Processor-memory bus
  - Curto e de alta velocidade
  - Adaptado à memória para maximizar largura de banda entre processador e memória
  - Otimizado para transferência de blocos de cache
- I/O bus
  - Mais comprido e lento
  - Acomoda uma extensa variedade de dispositivos de E/S
  - Conecta-se ao processor-memory bus através de um bridge (ponte) e/ou um backplane bus
  - Exs: SCSI, USB, Firewire
- Backplane bus
  - Usado como um barramento intermediário para conectar I/O buses a processor-memory bus
  - Ex: PClexpress

# Exemplo de Organização de E/S: Intel 5000P



**North Bridge:**  
Conecta processador à memória e à placa de vídeo

**South Bridge:**  
Conecta North Bridge aos vários I/O buses

# O Que Define um Barramento?

Transaction Protocol

Timing and Signaling Specification

Bunch of Wires

Electrical Specification

Physical / Mechanical Characteristics  
– the connectors

# Transação de E/S

- Sequencia de operações sobre um canal de comunicação que inclui um pedido e pode incluir uma resposta
  - Pedido ou resposta pode incluir a transmissão de dados
  - Iniciada por um pedido que pode ser realizado através de várias operações sobre um barramento
- Inclui tipicamente duas partes
  1. Enviar um endereço
  2. Receber ou enviar dados
- É definida pelo tipo de acesso à memória
  - Transação de **leitura** lê dados da memória e envia dados ao processador ou dispositivo E/S
  - Transação de **escrita** escreve dados na memória vindo do processador ou dispositivo de E/S

# Entrada e Saída

- Definição de entrada e saída leva em conta perspectiva do processador
- Entrada
  - Operação onde dispositivos escrevem dados na memória
  - Assim processador pode ler estes dados da memória
- Saída
  - Operação onde o processador escreve dados na memória
  - Assim os dispositivos podem ler estes dados

# Barramentos Síncronos e Assíncronos

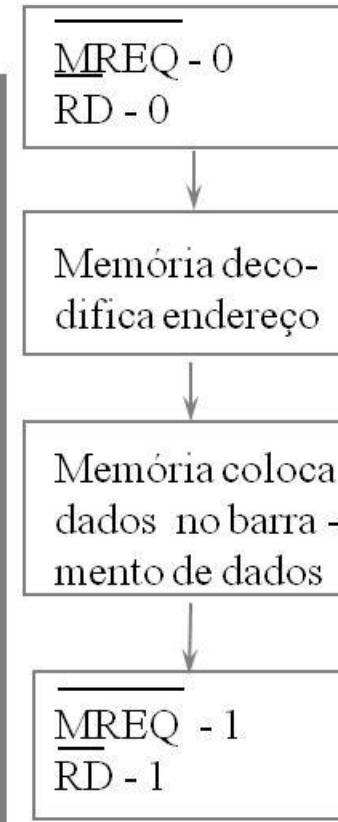
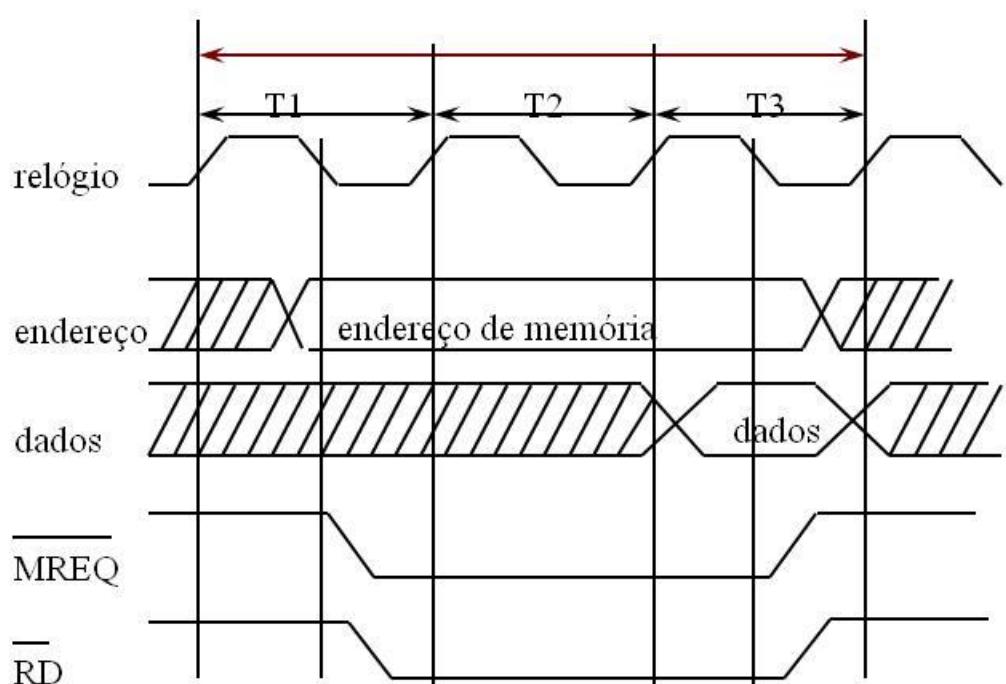
## ■ Barramento Síncrono

- Inclui um clock nas linhas de controle
- Protocolo de comunicação fixo baseado no clock
- Exemplo: Processor-Memory Bus

## ■ Barramento Assíncrono

- Não utiliza o sinal de clock
- Requer um protocolo de handshake e mais linhas de controle
- Exemplo: I/O Bus

# Barramento Síncrono



$T_1$  - CPU ativa sinais de controle e endereço

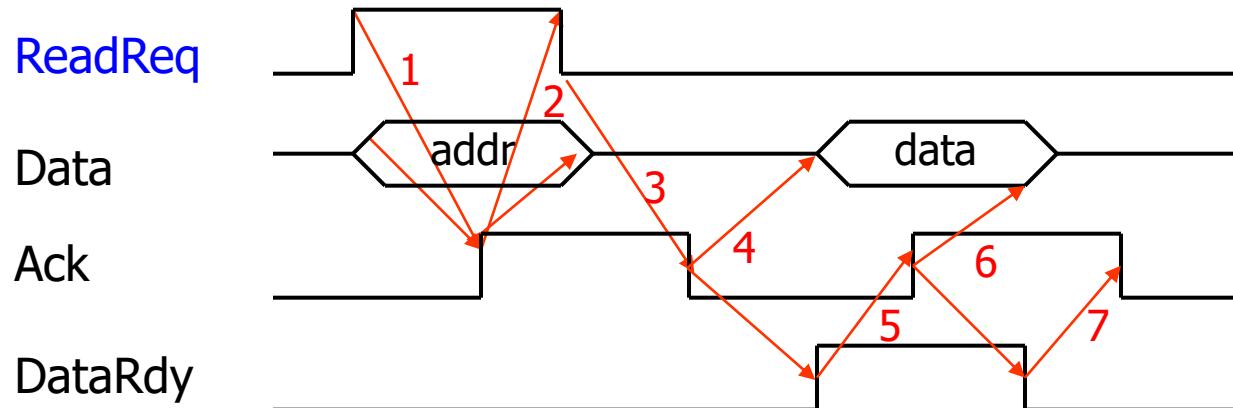
$T_2$  - Endereço estável no barramento

$T_3$  - Memória libera dados no barramento

-Dados são lidos pela CPU  
 -CPU desabilita controle

# Protocolo de Barramento Assíncrono

- Leitura da memória por um dispositivo E/S



Dispositivo E/S sinaliza requisição ativando ReadReq e colocando addr nas linhas de dados.

1. Memória vê **ReadReq**, lê addr da linha de dados e ativa Ack
2. Dispositivo E/S vê Ack e desativa **ReadReq** e linhas de dados
3. Memória vê **ReadReq** e desativa Ack
4. Quando dado da memória está pronto disponibiliza e ativa DataRdy
5. Dispositivo E/S vê DataRdy, lê dado e ativa Ack
6. Memória vê Ack, libera linhas de dados e desativa DataRdy
7. Dispositivo vê DataRdy desativado e desativa Ack

# Síncronos x Assíncronos

## ■ Barramento Síncrono

- Vantagens:
  - envolve muito menos lógica
  - pode operar em altas velocidades
- Desvantagens:
  - Todo dispositivo no barramento deve operar no mesmo clock rate
  - Não podem ser longos se são rápidos (clock skew)

## ■ Barramento Assíncrono

- Vantagens:
  - Acomoda grande variedade de dispositivos com diferentes velocidades
  - Podem ser longos sem se preocupar com clock skew ou com problemas de sincronização
- Desvantagem: Mais lento

# Acessando o Barramento

- Mestre do barramento
  - Inicia e controla todas as requisições ao barramento
- Quem pode ser mestre?
  - Mestre Único: Processador
    - Simplicidade
    - Processador coordena todas as transações do barramento  
→ degradação de desempenho
  - Múltiplos Mestres: Processador e Dispositivos de E/S
    - Necessidade de protocolo: request, grant e bus-release
    - Necessidade de esquema de arbitragem (Centralizada ou Distribuída)

# Barramento - Aspectos de projeto

- Considerações na implementação do barramento do sistema:

Opções	Alta Performance	Baixo custo
Largura do barramento	Endereços e dados separados	Multiplexação das linhas de endereço/dados
Largura de dados	16 bits, 32 bits, ... (Quanto maior mais rápido)	Menor, mais barato
Transações por pacotes	Múltiplas palavras menos overhead	Transferência por palavra é mais simples
Barramentos masters	Múltiplos mestres (requer arbitragem)	Apenas um mestre (sem arbitragem)
Relógio (Clocking)	Síncrono	Assíncrono

# Executando Operações de E/S

- Comunicação com os dispositivos de E/S:
  - envio de comandos aos dispositivos
  - transferência de dados de/para dispositivos
  - análise do status dos dispositivos e da transmissão
- Sistema Operacional : Interface entre usuário (programa) e dispositivo
  - Desenvolvimento de programas é facilitado pois detalhes de baixo nível dos dispositivos são abstraídos pelo S.O

# Suporte do S.O. a E/S

- Proteção
  - garante o acesso a dispositivos/dados para os quais se tenha permissão
- Abstração (dispositivo)
  - possui rotinas específicas com detalhes de cada dispositivo
- Gerenciamento
  - trata as interrupções causadas pelos dispositivos
- Escalonamento
  - controla a utilização de dispositivos compartilhados entre processos

# Controladores de E/S

- Dispositivos de E/S são gerenciados por controladores de HW
  - Transfere dados do/para o dispositivo
  - Sincroniza operações com software
- Controladores possuem diferentes registradores:
  - Registradores de comando
    - Faz o dispositivo começar alguma operação
  - Registradores de Status
    - Indica o que o dispositivo está fazendo e a ocorrência de erros
  - Registradores de Dados
    - Escrita: transfere dados para o dispositivo
    - Leitura: transfere dados do dispositivo

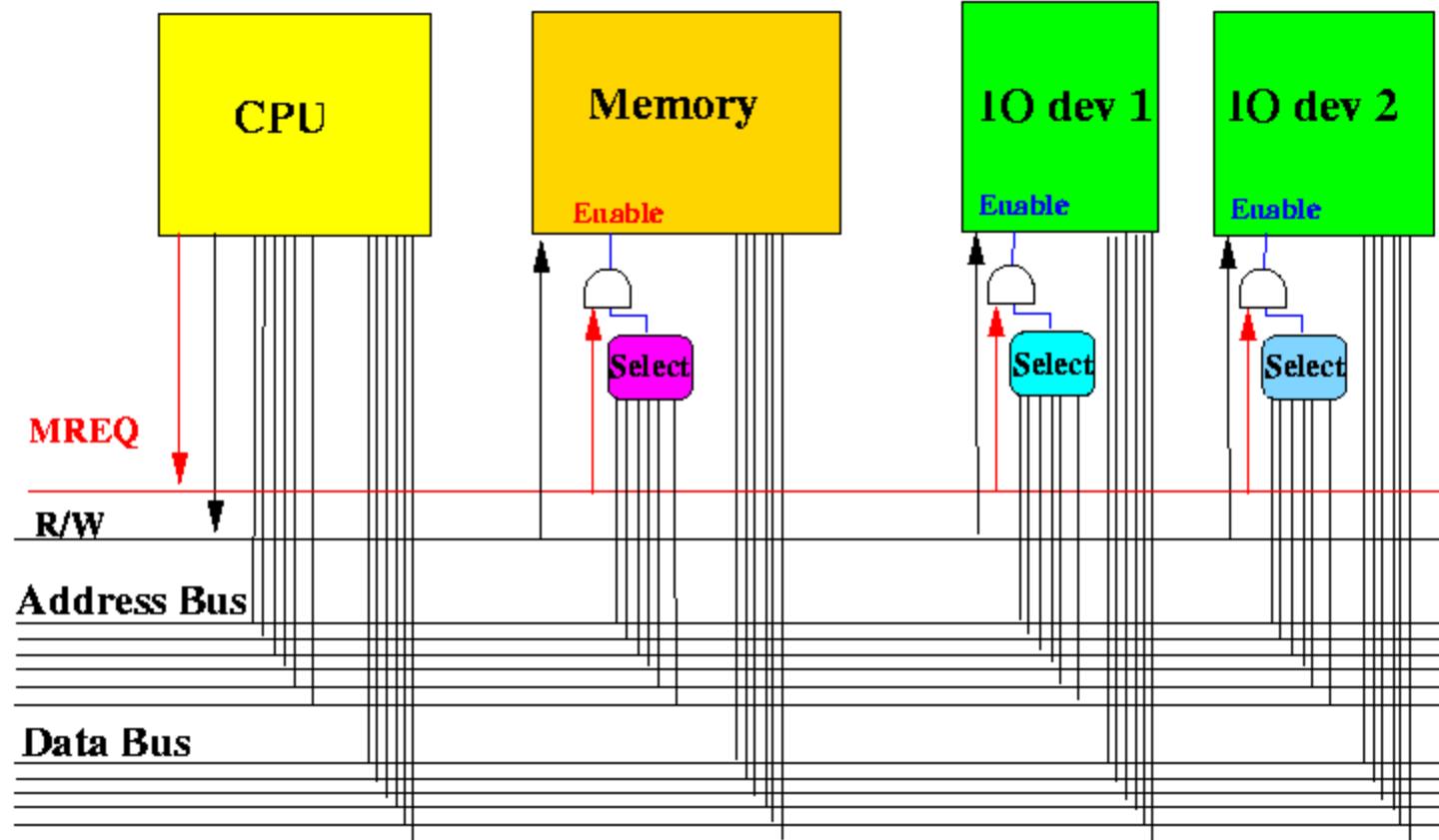
# Implementando Comando de E/S

- Para enviar um comando a um dispositivo de E/S, processador deve acessar registradores do controlador
- Processador deve endereçar dispositivo e enviar comandos
- Dois métodos de endereçamento do dispositivo
  - E/S mapeada em memória
  - Instruções específicas de E/S

# E/S Mapeada em Memória

- Parte do espaço de endereçamento é destinado a E/S
  - Instruções de Leitura/Escrita a endereços destinados aos dispositivo são interpretados como comandos
  - Memória ignora endereços, mas controlador de dispositivo examina endereço, grava dados nos seus registradores e manda sinal para dispositivo
- Instruções de escrita e leitura aos endereços de E/S só podem ser feitas através de S.O
- Vantagem:
  - Simplicidade de implementação
- Desvantagem:
  - Processadores que possuem espaço de endereçamento limitados são penalizados

# Implementação de E/S Mapeada em Memória



A memória só é ativada para um determinado intervalo de endereços

# Exemplo: E/S Mapeada em Memória no MIPS

**Problema:**

Suponha que um dispositivo E/S seja mapeado para endereço 0xFFFFFFF4

Escreva código para o MIPS que escreva o valor 7 para este dispositivo e depois leia a saída do dispositivo.

**Solução:**

```
addi $t0,$zero,7
sw $t0, 0xFFFF4($zero)
lw $s0, 0xFFFF4($zero)
```

# Instruções Específicas de E/S

- ISA do processador contém instruções específicas para acessar dispositivos de E/S
  - Linhas (sinais) de endereço diferentes para dispositivos e memória
- Instrução deve especificar o comando e o dispositivo
  - Só podem ser executadas através de S.O
- Utilizado em processadores Intel
  - Instruções **in** e **out**
  - Intel também utiliza E/S mapeada em memória
- Vantagem:
  - Permite que todo espaço de endereçamento seja utilizado para a memória
- Desvantagem:
  - ISA mais complexa de implementar

# Comunicação do Dispositivo E/S para o Processador

- Dois métodos comumente utilizados
  - Polling
    - Processador periodicamente checa o status do dispositivo para determinar se dispositivo precisa de algum serviço (ou que acabou uma operação)
  - Interrupção
    - Dispositivo sinaliza processador que precisa de algum serviço (ou que acabou uma operação)
- Ambos os métodos podem ser utilizados em um sistema para diferentes tipos de dispositivos

# Polling

- Dispositivo coloca informação no registrador do controlador e processador lê registrador
- Processador detém controle e faz todo o trabalho
- Utilizado em dispositivos como impressoras
- Vantagem:
  - Simplicidade de implementação
- Desvantagem:
  - CPU fica esperando pelo dispositivo, gastando ciclos de processamento

# Comunicação com Polling

*CPU*

*1- lê reg do status do disp.*

*3- Inspeciona status,  
se não está pronto va para 1*

*4- escreve no reg. dado*

*6- se existe mais dados vá  
para 1*

*DISPOSITIVO*

*2- envia status para reg.*

*5- aceita dado,  
status=ocupado até escrita  
terminada*

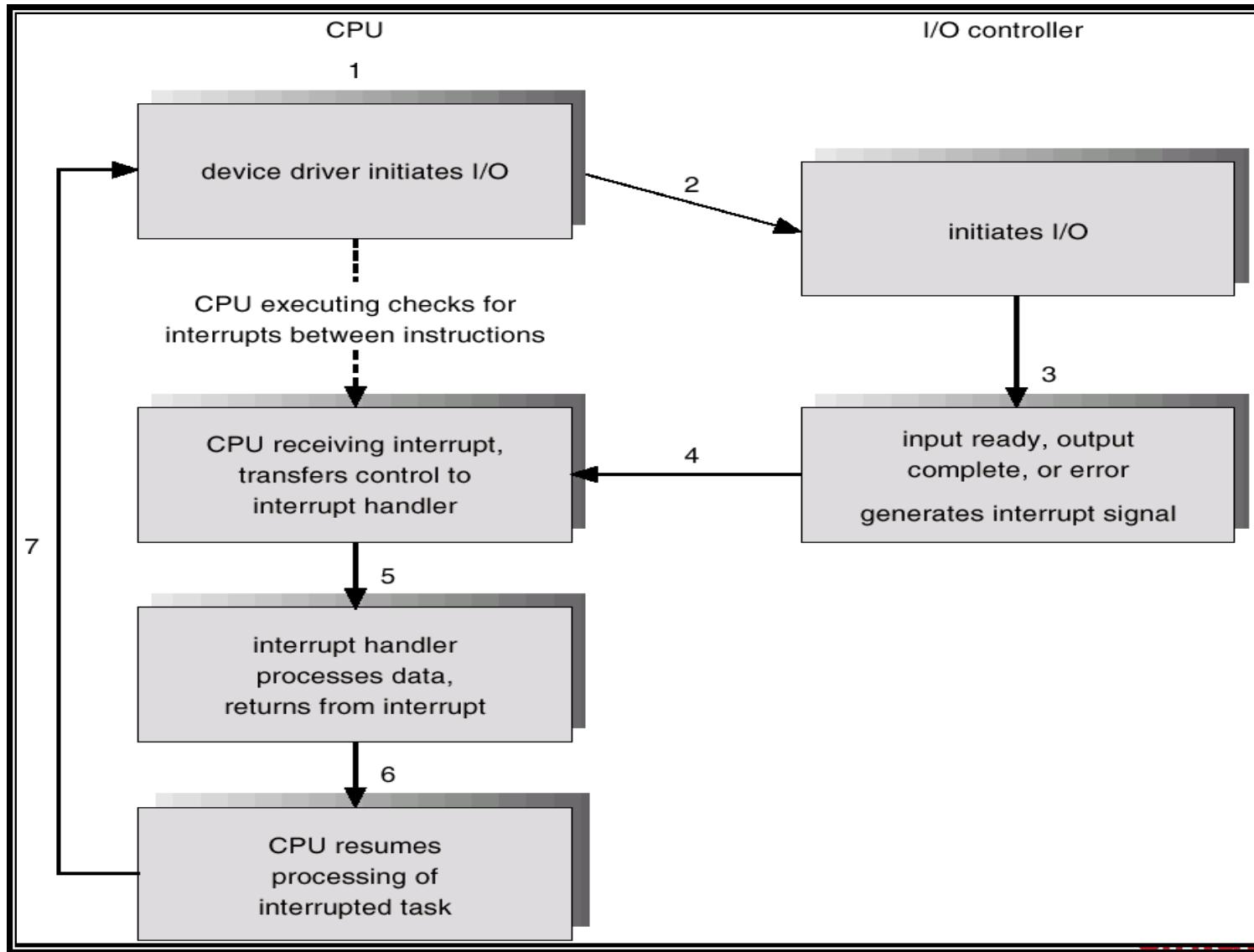
# Interrupções

- Uma interrupção é similar a uma exceção, porém:
  - É assíncrona em relação a execução de instruções
    - Não pára a execução da instrução
  - Pode ser tratada quando for conveniente
- Informação sobre a causa da interrupção frequentemente identifica dispositivo
- Interrupções podem ter diferentes urgências
  - Necessidade de mecanismo para atribuir prioridades

# E/S Com Interrupções

- CPU executa outras instruções enquanto espera E/S
- Sincronização: interrupção
  - disp. E/S está pronto para nova transferência
  - operação de E/S terminou
  - ocorreu erro
- Vantagem:
  - Melhor utilização da CPU
    - Não precisa ficar verificando o status do dispositivo
    - Programa só é suspenso quando E/S interromper
- Desvantagem:
  - HW especial é necessário
    - Para indicar dispositivo
    - Para salvar contexto antes de atender dispositivo

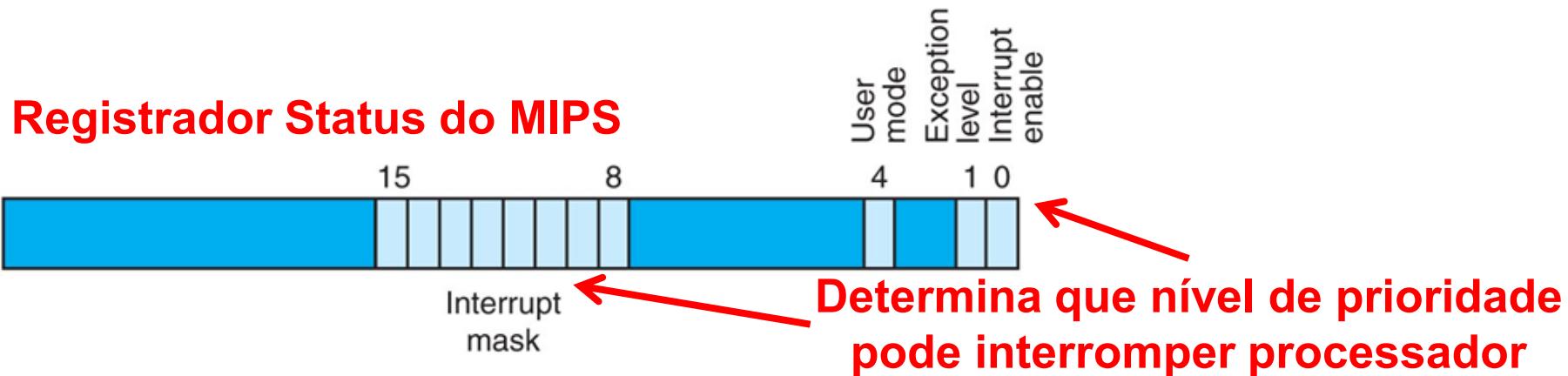
# Comunicação com Interrupção



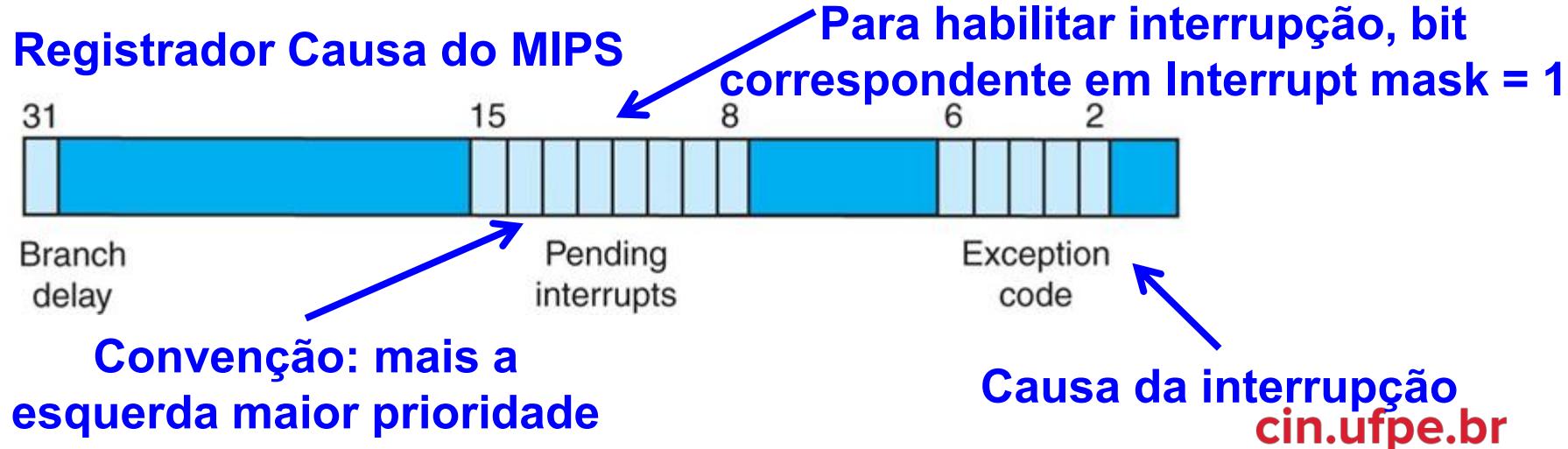
# Níveis de Prioridades de Interrupções

- Níveis de prioridade podem ser usados para direcionar a ordem em que o S.O atende as interrupções

**Registrador Status do MIPS**



**Registrador Causa do MIPS**



# Transferência entre Dispositivos E/S e Memória -Direct Memory Access (DMA)

- Polling e Interrupções requerem que processador seja encarregado de transferir dados entre dispositivos e a memória
  - Gasta muitos ciclos do processador
  - Bom para quantidade de dados pequena
- DMA é um mecanismo que permite a transferência de blocos de dados diretamente entre dispositivos e memória

# Funcionamento do DMA

- Processador fornece ao controlador de DMA:
  - Endereço do dispositivo, operação, endereço de memória e número de bytes
- Controlador de DMA gerencia a transferência
  - Quando controlador de DMA termina, interrompe o processador
- Processador pode ter que esperar enquanto memória está realizando uma transferência de DMA
  - Uso de caches evita acesso do processador à memória

# Interação DMA-Cache

- Se DMA escreve para um bloco de memória que está na cache
  - Cópia da cache se torna desatualizada (stale)
- Se cache com write-back tiver bloco “sujo” e DMA lê o bloco da memória
  - DMA lê dados desatualizados (stale)
- Necessário assegurar coerência da cache
  - Flushing
    - Invalidar blocos da cache em uma entrada de E/S
    - Forçar escrita do bloco da cache na memória em uma saída de E/S