



Organização e Arquitetura de Computadores

Prof. Lúcio Renê Prade

Arquitetura de um Computador

A arquitetura de um computador é a teoria por detrás do desenho de um computador.

Arquitetura de um Computador

Arquitetura refere-se aos atributos que são visíveis para o programador, ou seja, os atributos que tem impacto direto na execução do programa.

Atributos:

- ❑ Conjunto de instruções
- ❑ Número de bits
- ❑ Mecanismos de E/S

Arquitetura de um Computador

Especificar se um computador deve ou não ter uma instrução de multiplicação constitui uma decisão de projeto de **Arquitetura**

Definir se essa instrução será implementada por uma unidade específica de multiplicação ou por um mecanismo que utiliza repetidamente sua unidade de soma é uma decisão de **Organização**

Arquitetura de um Computador

Exemplo

- Todo processador Intel da família x86 compartilham a mesma arquitetura básica.
- No entanto, a organização difere de uma versão para outra.

Conclusão

- uma organização deve ser projetada para implementar uma especificação particular de arquitetura.

Instruções

Representação elementar que gera uma ação em um computador.

Determina o que o computador deve fazer naquele instante. Um programa é composto por muitas instruções, que são executadas de forma ordenada pelo processador.

Estrutura de uma instrução

OPCODE	OPERANDO1	OPERANDO2	OPERANDO3	OPERANDOn
--------	-----------	-----------	-----------	-----------

00000010001100100100000000100000
add \$t0, \$s1, \$s2

Instruções

Matemáticas e lógicas

- ❑ Soma, subtração, and, or...

Movimentação de dados

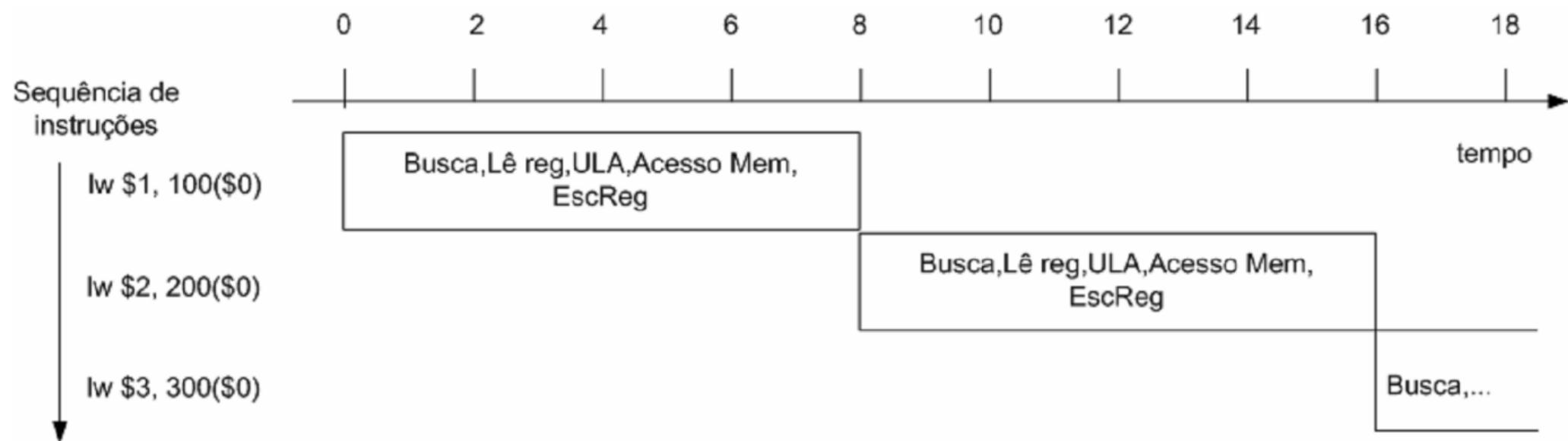
- ❑ registrador – registrador; registrador – memória;
memória – registrador.

Entrada/Saída

Controle

- ❑ Instruções de salto

Arquitetura Monociclo



Arquitetura Multiciclo

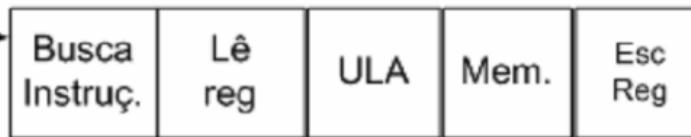


Sequência de instruções

Iw \$1, 100(\$0)



Iw \$2, 200(\$0)



Iw \$3, 300(\$0)



10ns

10ns

10ns

Paralelismo

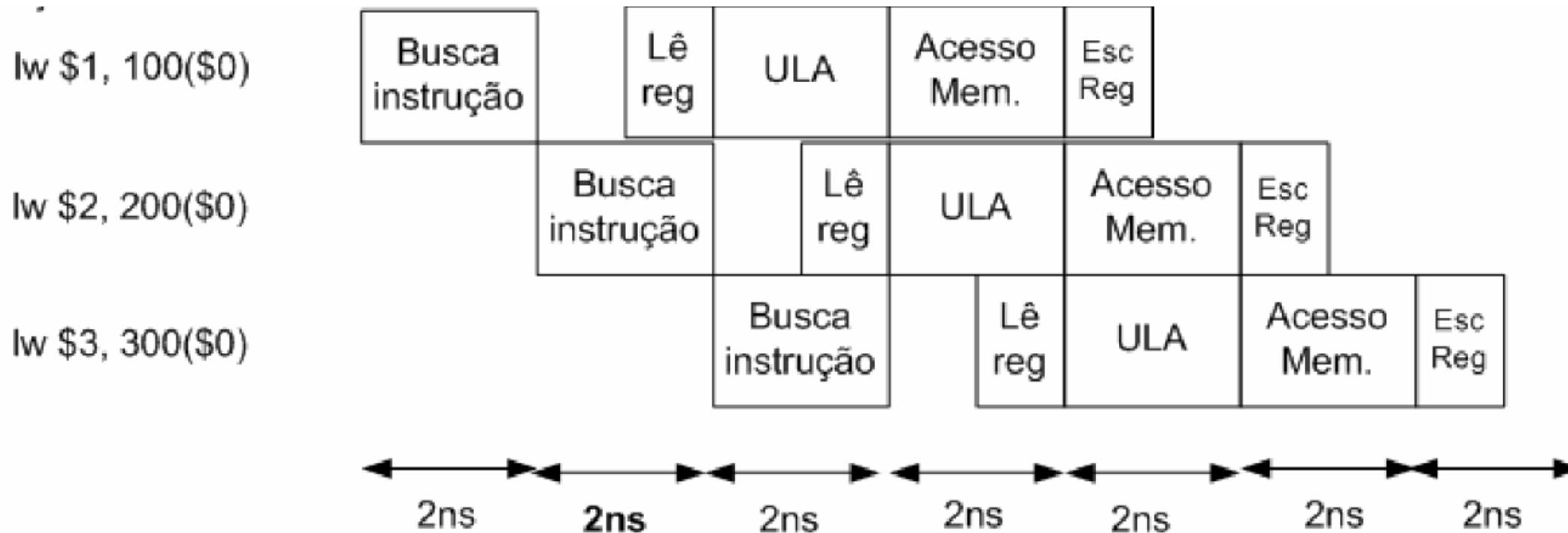
O processador perde muito tempo aguardando os dados da memória.

Para melhorar o desempenho do sistema, utiliza-se o paralelismo.

Pode ocorrer em dois níveis

- ❑ Nível de instrução – pipeline
- ❑ Nível de hardware – mais de um processador

Arquitetura Pipeline



Paralelismo de Hardware

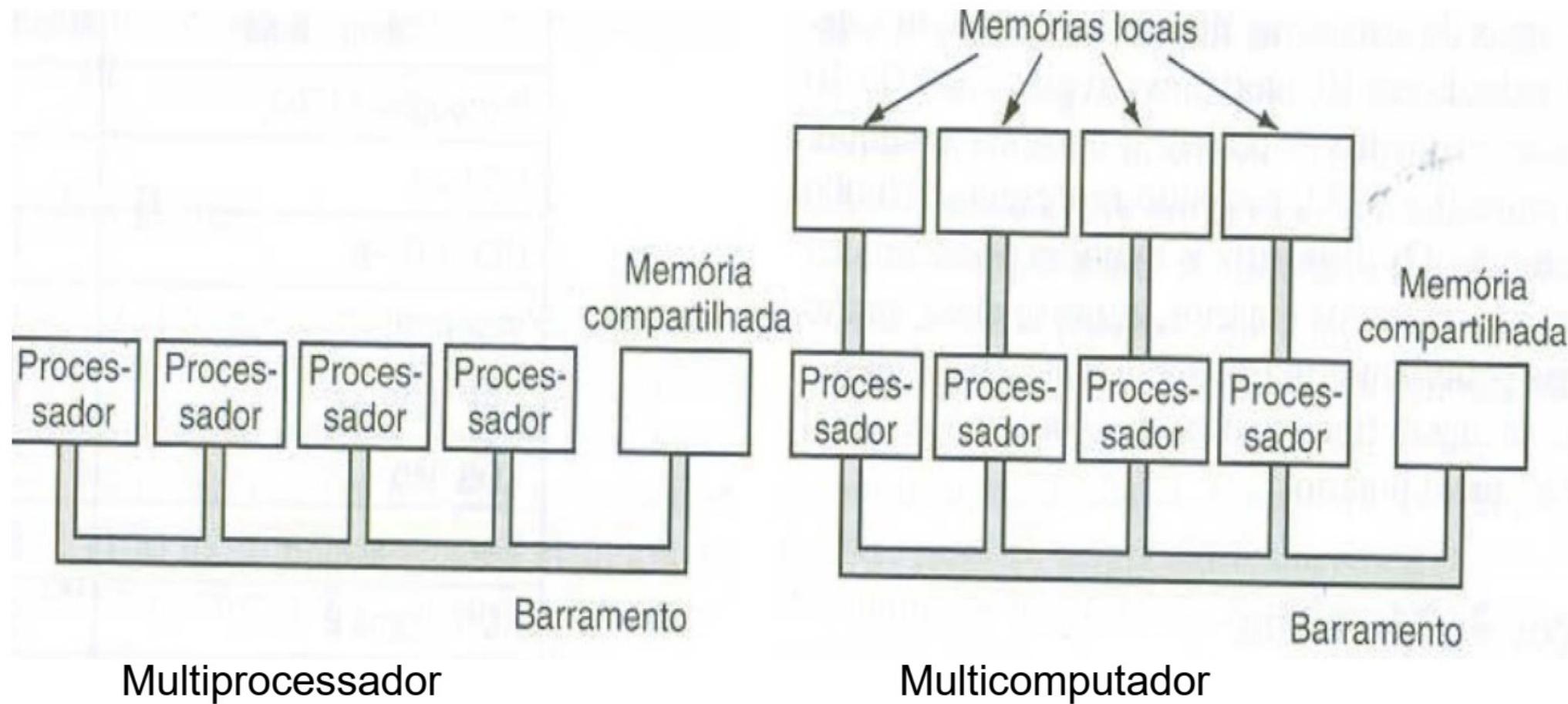
Multiprocessador

- Processadores interligados que executam instruções do mesmo programa e que compartilham a mesma memória.

Multicomputador

- O mesmo que o anterior, porém, além da memória compartilhada, possui também uma memória própria.
- Trabalha como se fosse um outro computador, portanto, melhorando o desempenho.

Paralelismo de Hardware



Arquitetura do Processador MIPS

Por simplicidade, faremos uma implementação de um subconjunto de instruções MIPS:

Instruções de referência à memória:

load word (lw) e *store word (sw)*

Instruções lógico-aritméticas:

add, *sub*, *and*, *or* e *slt*

Instrução de desvio condicional:

branch equal (beq)

Instrução de desvio incondicional:

jump (j)

Arquitetura do Processador MIPS

Dois passos são fundamentais e comuns a todas as instruções que podem ser executadas:

- 1) Busca (*fetch*) de instrução: envia o *program counter* (PC) para a memória que contém o código e recupera a instrução.
- 2) Leia um ou dois registradores, usando os campos da instrução para selecioná-los adequadamente:
 - Se for uma instrução *load*, um único registrador é lido.
 - A maior parte, porém, das instruções, lê dois registradores.

Arquitetura do Processador MIPS

As ações subsequentes dependem da classe à que pertence a instrução.

Vantagem da arquitetura MIPS: mesmo instruções pertencentes a classes distintas requerem ações relativamente semelhantes.

- A simplicidade e regularidade do conjunto de instruções MIPS favorece a implementação ao tornar parecida a execução das várias classes de instruções.

Arquitetura do Processador MIPS

Todas as instruções, exceto o *jump*, utilizam a unidade lógico-aritmética após a leitura dos registradores.

- Instruções de referência à memória: cálculo do endereço;
- Instruções lógico-aritméticas: operação especificada;
- Instruções de desvio: comparação.

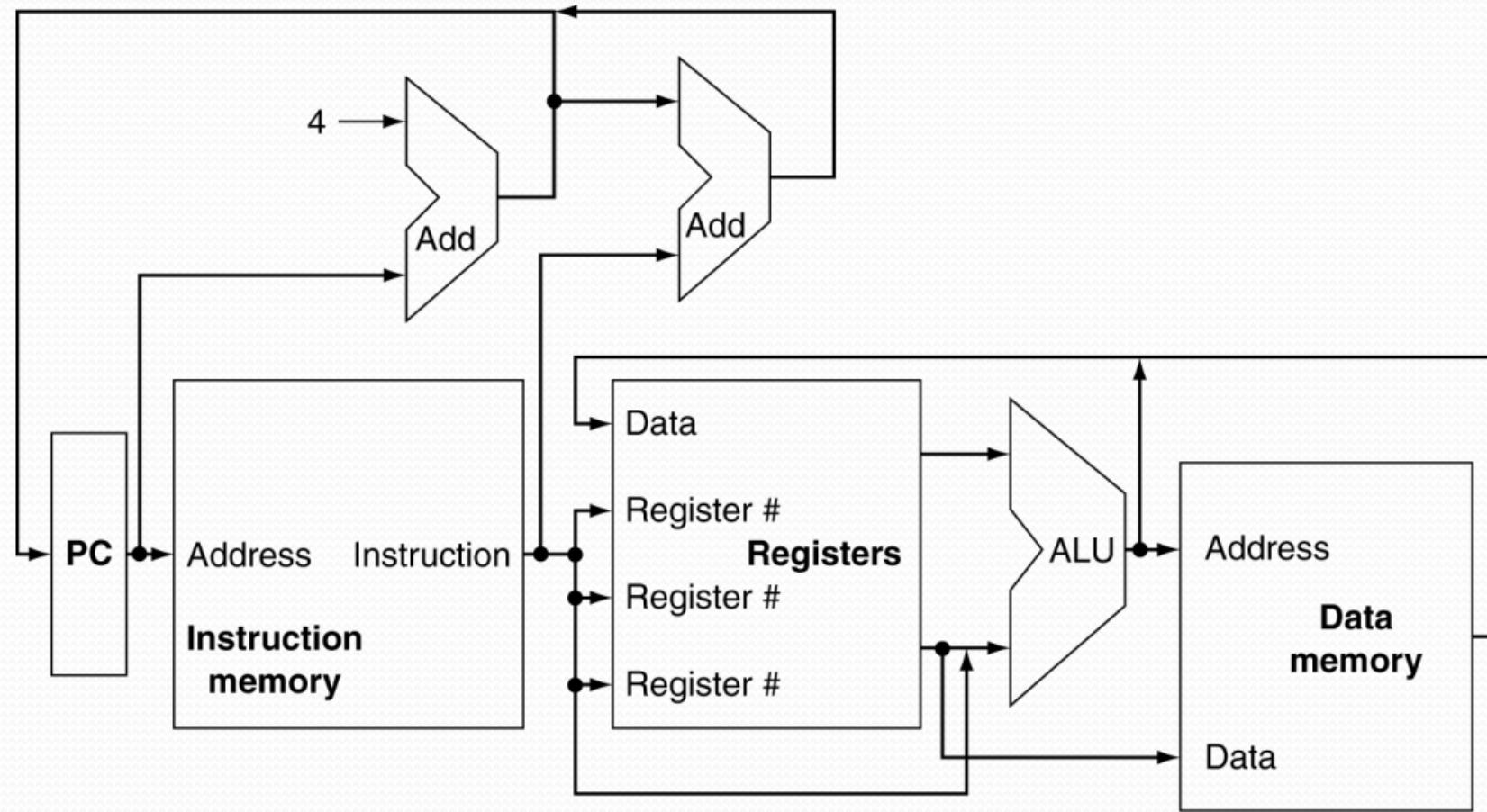
Arquitetura do Processador MIPS

Após o uso da ALU, as ações necessárias para a conclusão da instrução dependem da classe.

- As instruções de referência à memória precisam acessar a memória para ler ou escrever um dado.
- Instruções lógico-aritméticas, bem como a instrução de *load*, precisam escrever um dado (da ALU ou da memória) de volta em um registrador.
- Instruções de *branch* alteram o endereço da próxima instrução (i.e., o conteúdo de PC) com base na comparação.
- De modo geral (ou caso a comparação de um *branch* falhe), o valor de PC precisa ser incrementado por 4 para avançarmos à próxima instrução.

Organização do Processador MIPS

- Datapath básico:

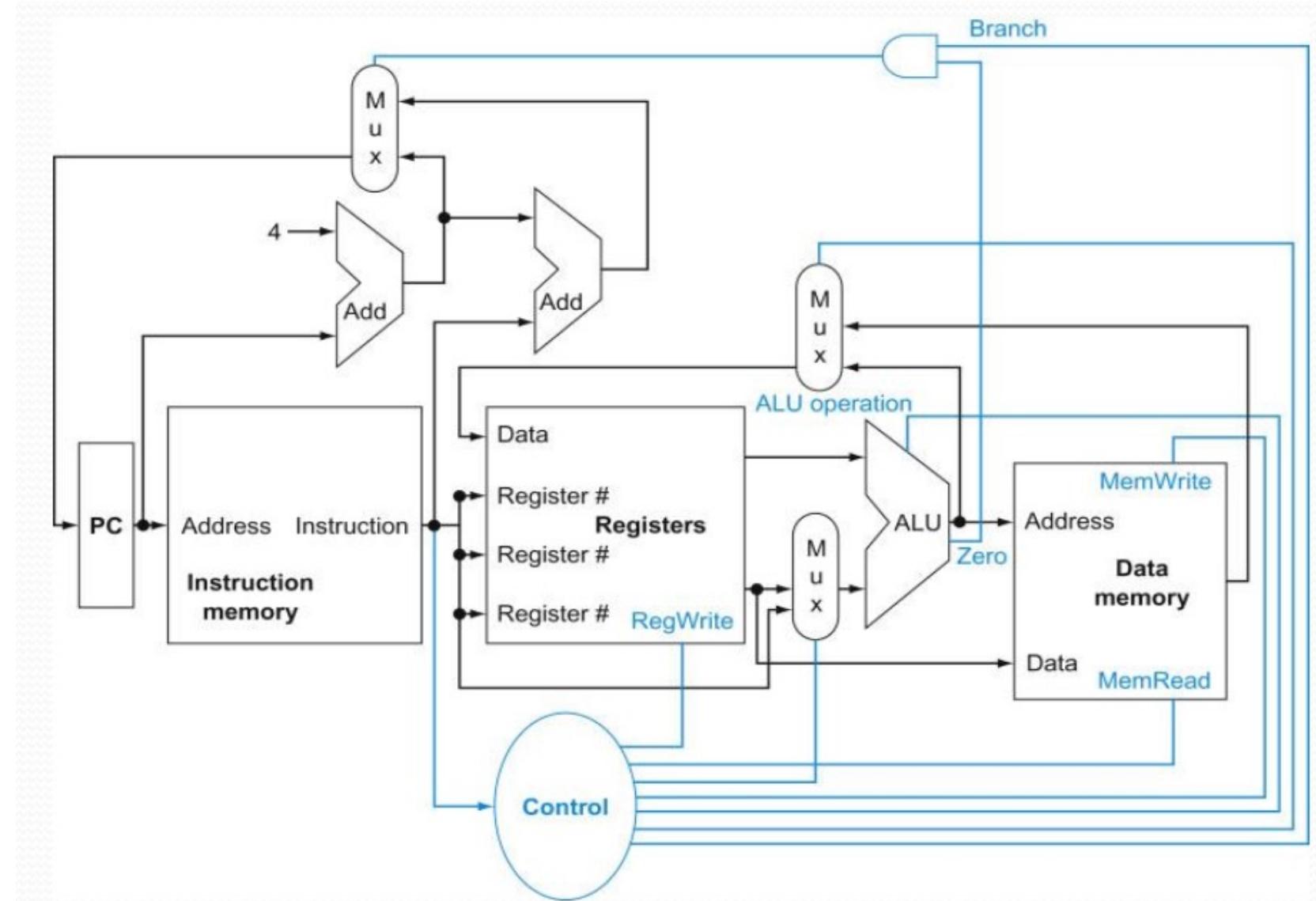


Organização do Processador MIPS

As linhas de dados não podem assumir diferentes valores gerados por diferentes fontes ao mesmo tempo – é preciso selecionar qual informação deve ser transmitida em cada momento.

Várias unidades, como, por exemplo, a ALU, precisam ser controladas (instruídas) para que executem a operação correta determinada pela instrução em curso.

Organização do Processador MIPS



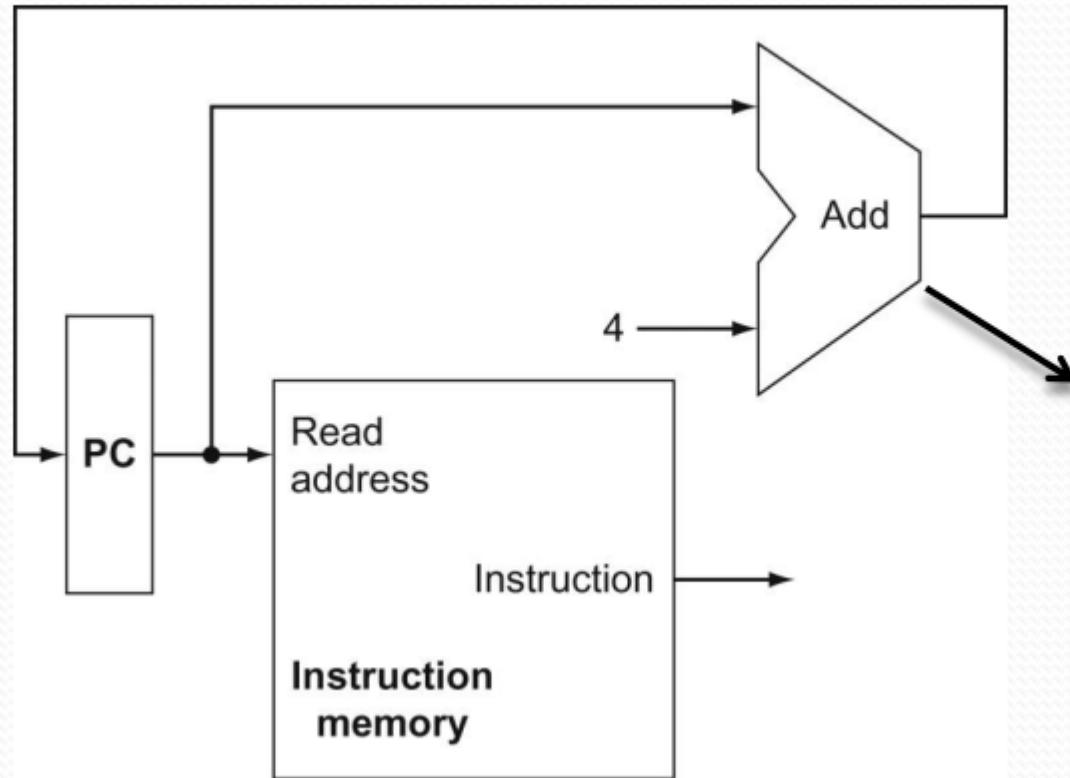
Organização do Processador MIPS

Opção: implementação uniciclo

- Toda instrução inicia sua execução em uma borda ativa do sinal de relógio e termina na próxima borda.
- **Vantagem:** simples de entender.
- **Desvantagem:** o ciclo de relógio precisa ser alongado o suficiente para acomodar a instrução mais longa.

Organização do Processador MIPS

1º passo: *fetch* de instrução

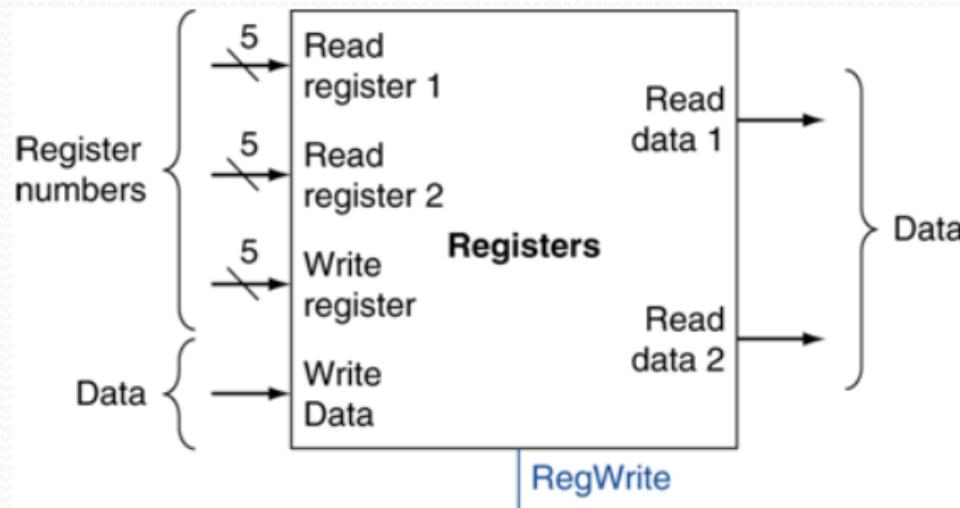


Círculo somador ou ALU fixada (em hardware) para sempre realizar uma operação de soma.

Organização do Processador MIPS

2º passo: Instruções do formato R (lógico-aritméticas)

- *add, sub, and, or e slt*



Arquivo de registradores: 32 registradores (rotulados como 0 – 31)

- **Entradas:** dois índices de registrador (5 bits) para leitura
um índice de registrador para escrita
uma entrada de dados (32 bits)
uma entrada de controle (RegWrite)
- **Saídas:** dois conteúdos de registradores (32 bits)

Organização do Processador MIPS

2º passo: Instruções do formato R (lógico-aritméticas)

- *add, sub, and, or e slt*



Unidade lógico-aritmética: oferece um conjunto de operações básicas que podem ser selecionadas por meio de uma entrada de controle.

Organização do Processador MIPS

Soma / Subtração

- Se há subtração, é preciso haver um mecanismo que compute o complemento de 2 de uma entrada.

'E' e 'Ou' lógico

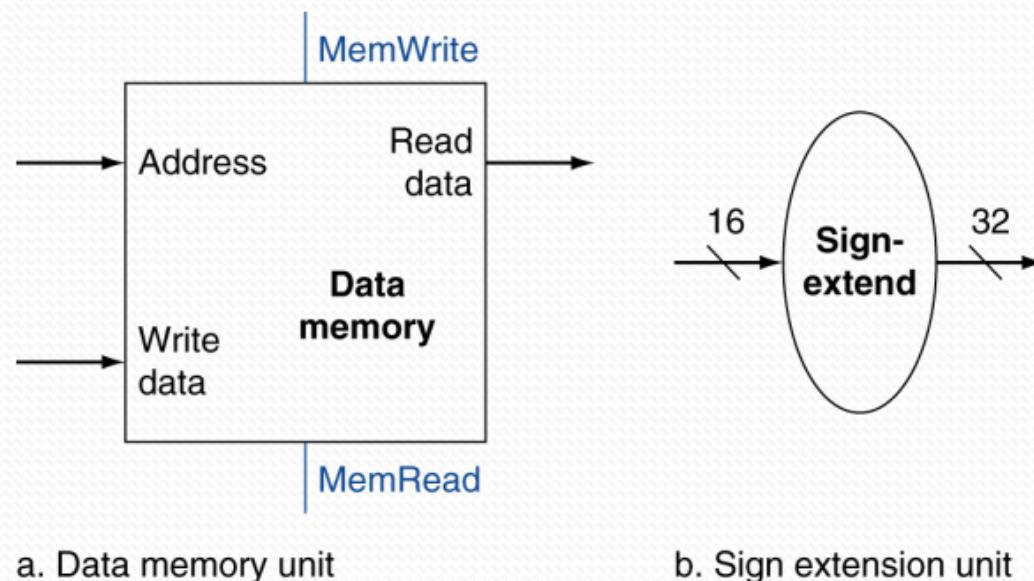
Detecção de zero / Comparação

- Para a instrução slt, é preciso checar se o resultado da subtração entre os operandos, e.g., $(a - b)$, foi negativo.
- Logo, o bit mais significativo do resultado, sendo um, indica que é preciso gerar na saída da ALU o valor 0000...1; caso contrário, o valor deve ser 0000...0.

Organização do Processador MIPS

3º passo: instruções de referência à memória

- $lw \$t1, offset(\$t2) / sw \$t1, offset(\$t2)$
 - Usa o arquivo de registradores e a ALU.
 - Requer uma memória de dados, que pode ser acessada tanto para leitura quanto para escrita.
 - Precisa de um módulo que faça a extensão de sinal do valor imediato contido nos 16 bits menos significativos da instrução.

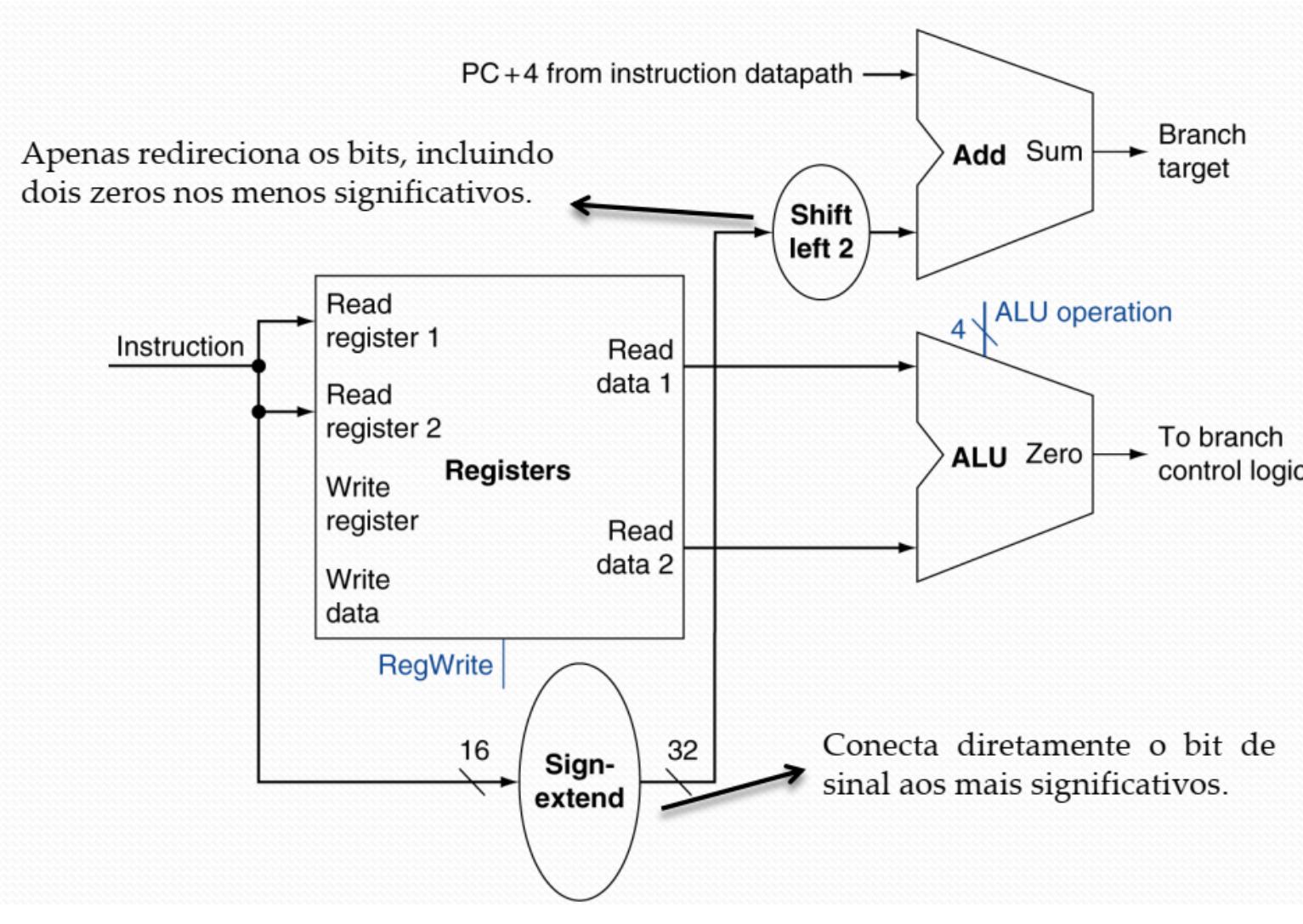


Organização do Processador MIPS

4º passo: instrução de desvio condicional

- **beq \$t1, \$t2, offset**
 - **Endereço alvo:** soma do valor de $(PC + 4)$ com o offset (após extensão de sinal).
 - O campo de *offset* precisa, antes, ser deslocado para a esquerda por dois dígitos (para referenciar *byte*).
 - Depois, é preciso comparar os valores armazenados nos registradores especificados para saber se o desvio será tomado ou não.

Organização do Processador MIPS



Organização do Processador MIPS

5º passo: instrução de desvio incondicional

- É preciso substituir os 28 bits menos significativos do PC pelos 26 bits contidos no campo da instrução deslocados dois dígitos à esquerda (i.e., concatenados com 00).

Organização do Processador MIPS

5º passo: instrução de desvio incondicional

- É preciso substituir os 28 bits menos significativos do PC pelos 26 bits contidos no campo da instrução deslocados dois dígitos à esquerda (i.e., concatenados com 00).

Organização do Processador MIPS

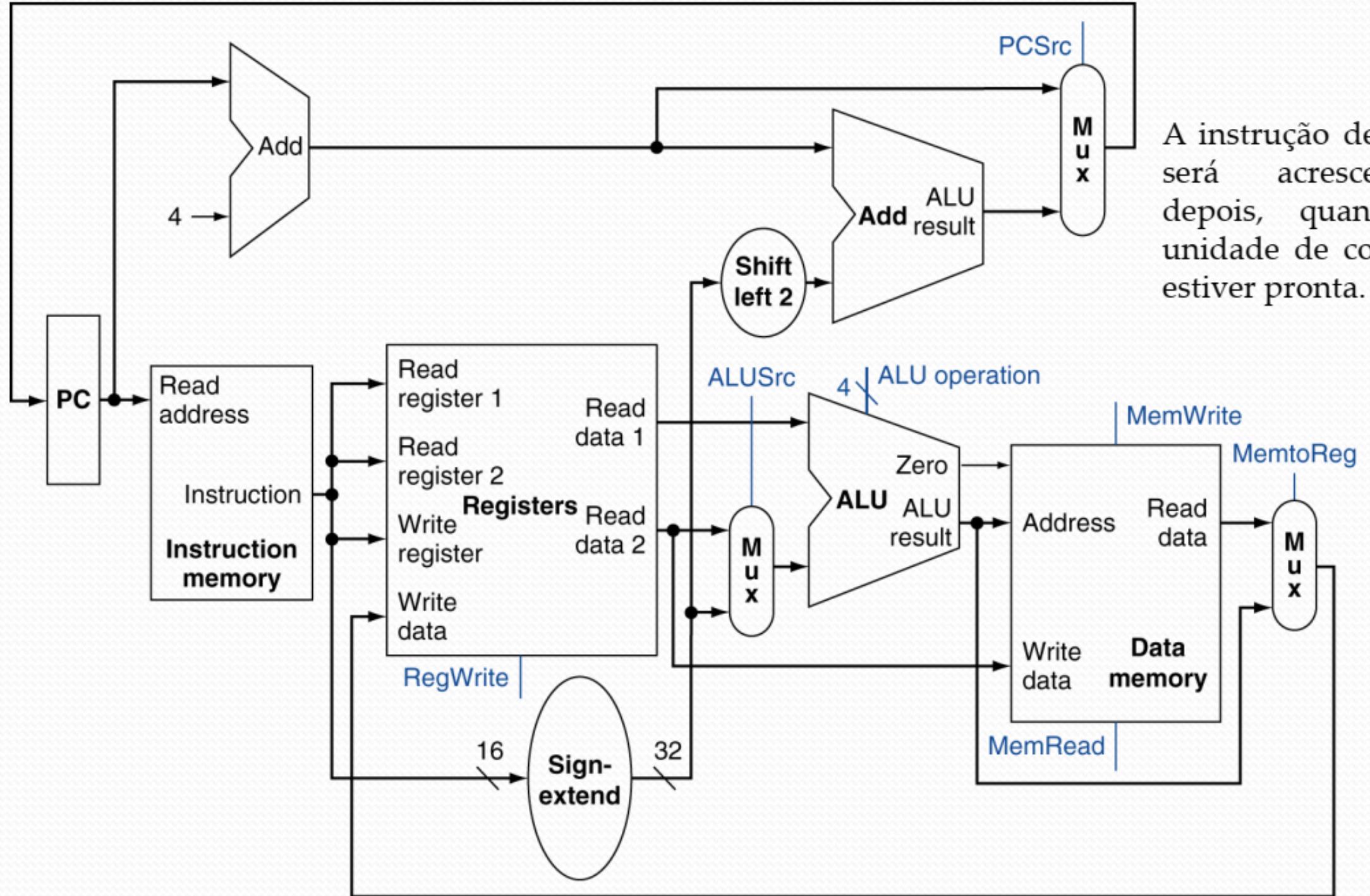
Nenhum recurso do *datapath* pode ser utilizado **mais do que uma vez** por instrução.

- Isto impõe, portanto, o uso de uma unidade de memória para instruções separada da memória de dados.

Logo, qualquer unidade que precisa ser utilizada mais vezes tem que ser replicada.

Ainda assim, várias unidades poderão ser compartilhadas pelas instruções.

- Para isto, teremos que selecionar as entradas apropriadas destas unidades (usando multiplexadores e sinais de controle).



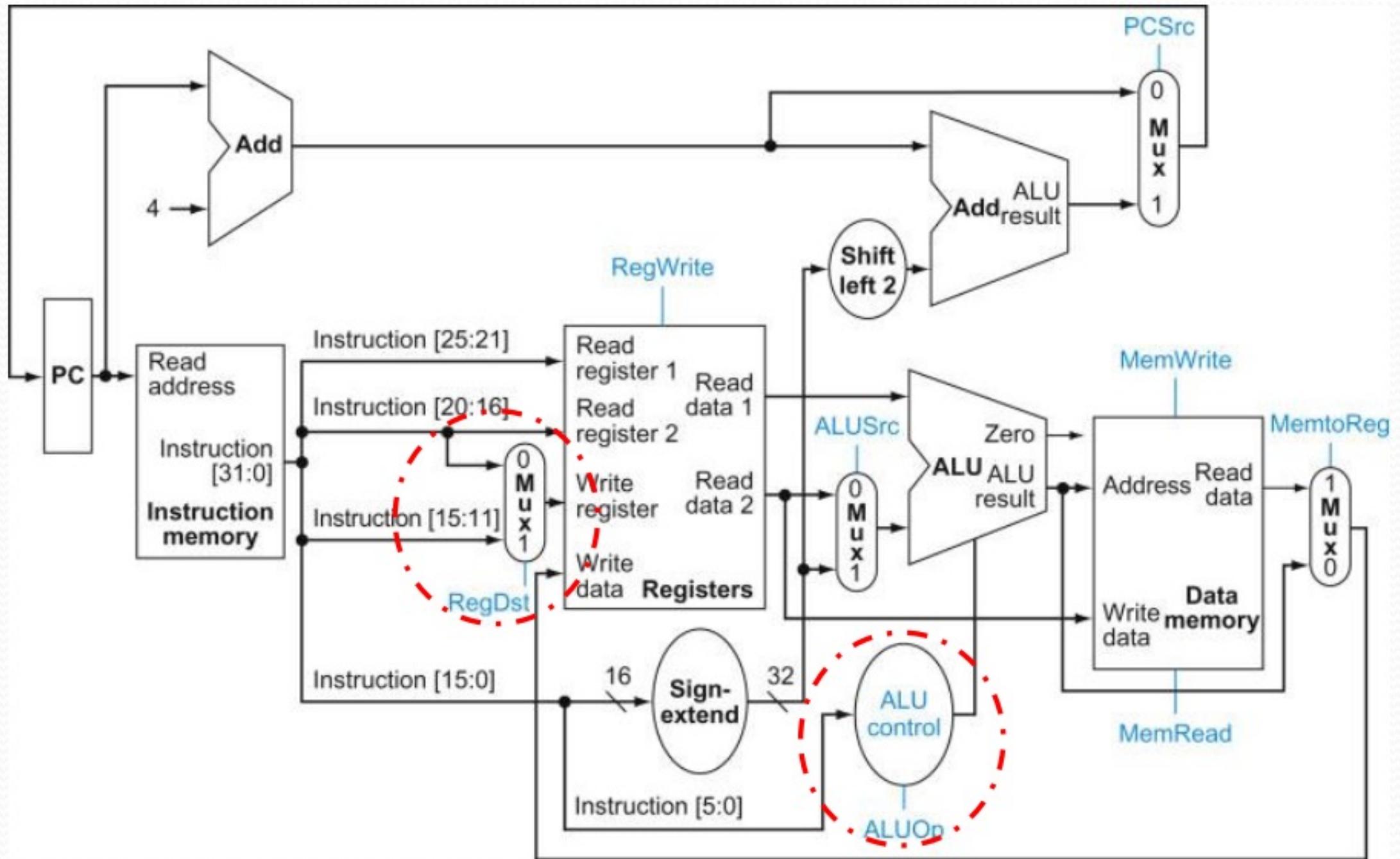
A instrução de *jump* será acrescentada depois, quando a unidade de controle estiver pronta.

Organização do Processador MIPS

Unidade central de controle:

➤ Vamos recordar os formatos de instrução:

Field	0	rs	rt	rd	shamt	funct		
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0		
a. R-type instruction								
Field	35 or 43	rs	rt	address				
Bit positions	31:26	25:21	20:16	15:0	Extensão de sinal			
b. Load or store instruction								
Field	4	rs	rt	address				
Bit positions	31:26	25:21	20:16	15:0	Extensão de sinal			
c. Branch instruction								
Opcode – Op[5:0]				Sempre é lido – como registrador base (<i>load/store</i>) ou como operando (<i>instituição R e branch</i>).				

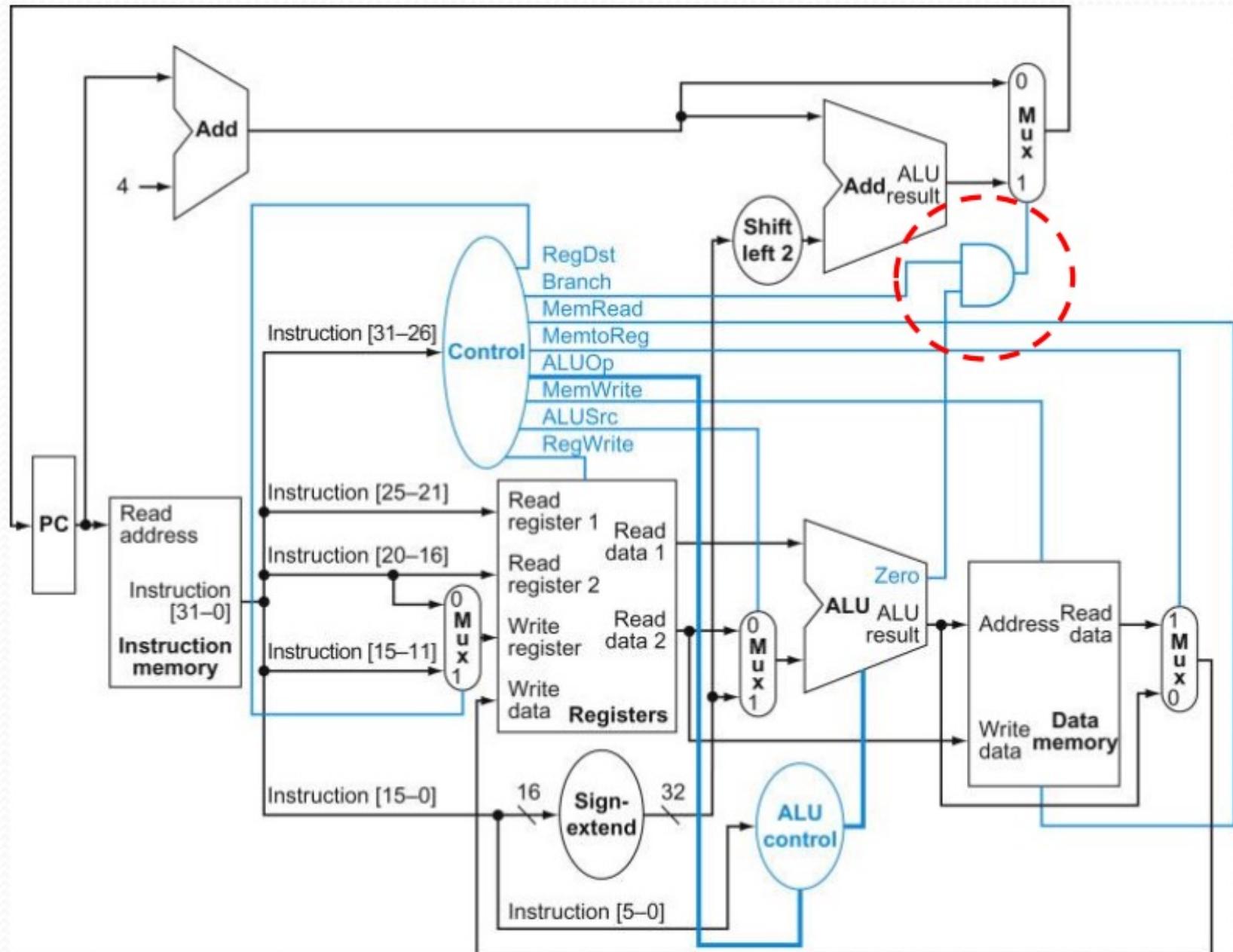


• Unidade central de controle:

➤ Gera todos os sinais de controle tendo como base apenas o *opcode* da instrução.

➤ Exceção: PCSrc

- Ele deve ser ativado quando a instrução for um *branch* E a saída zero da ALU for igual a 1.



Evolução do Processador

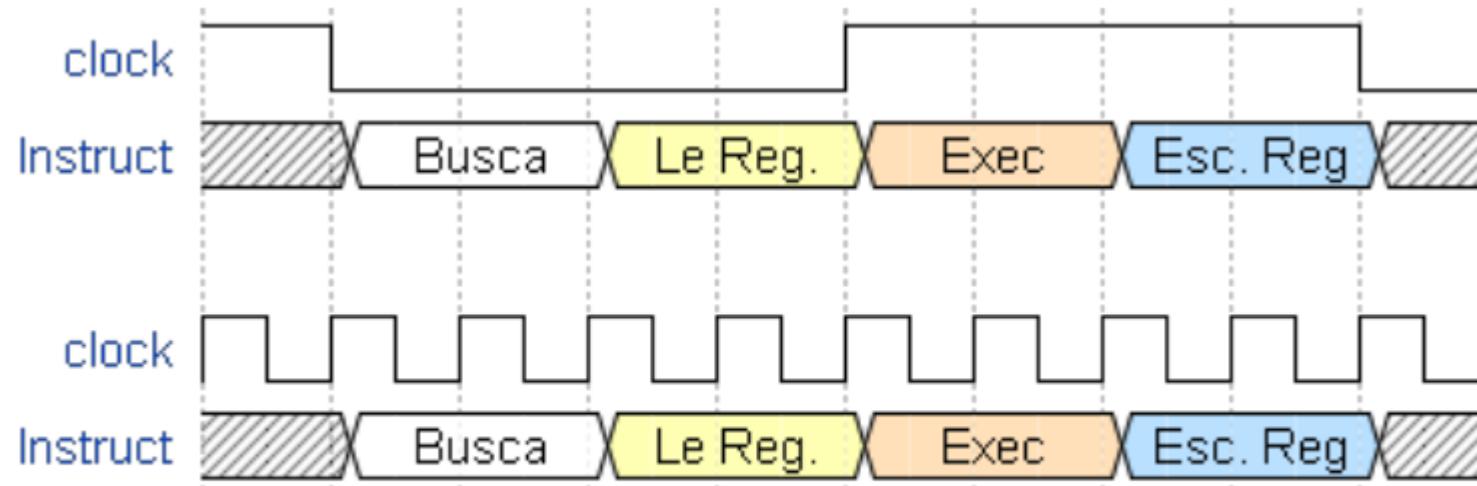


Figura: Monociclo x Multiciclo

Evolução do Processador

Vamos começar com uma visão simplificada da ideia de pipeline.

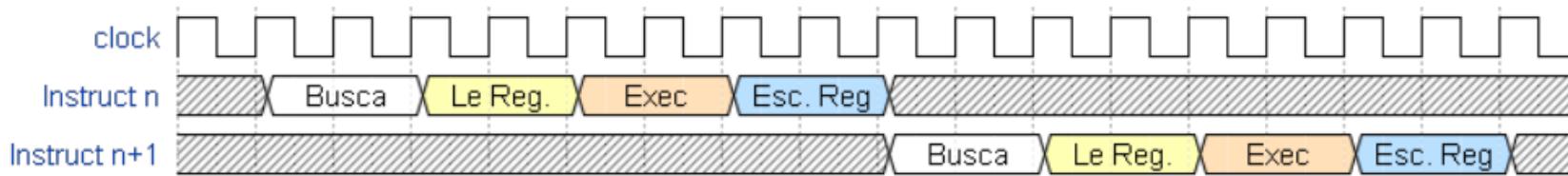


Figura: Execução sequencial das instruções

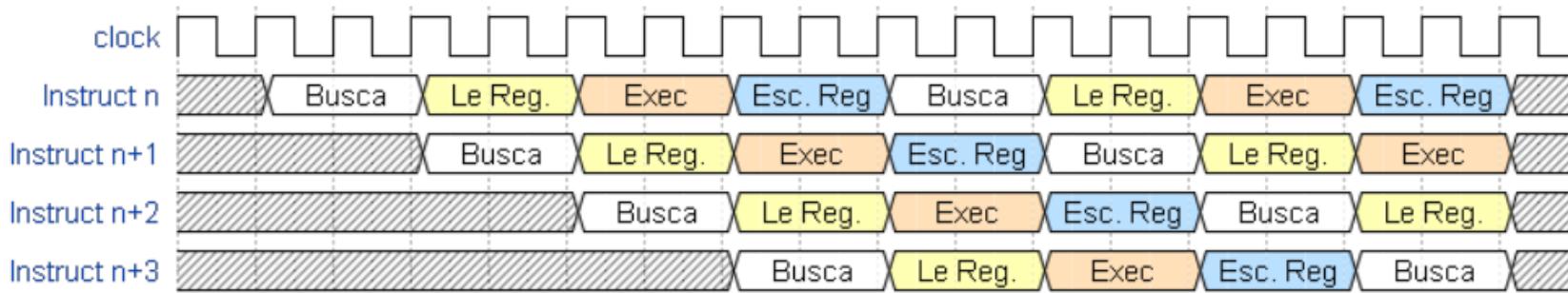


Figura: Execução paralela das instruções

Evolução do Processador

A execução das instruções do MIPS classicamente envolve **cinco** passos:

1. Busca de instrução (*Fetch*).
2. Leitura dos registradores (operандos) enquanto ocorre a decodificação da instrução.
 - Isto é possível graças ao formato regular das instruções MIPS.
3. Executa a operação ou calcula um endereço.
4. Acessa um operando na memória de dados.
5. Escreve o resultado em um registrador.

Logo, a *pipeline* que vamos projetar e analisar neste tópico terá cinco estágios.

Evolução do Processador

MIPS: 8 instruções – *lw, sw, add, sub, and, or, slt e beq.*

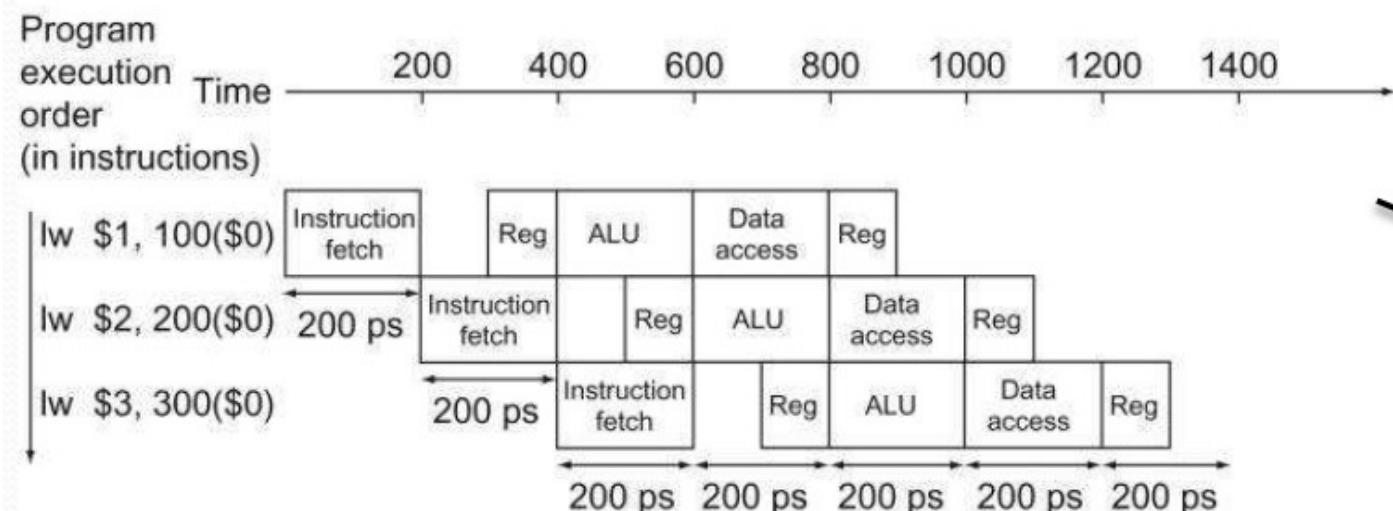
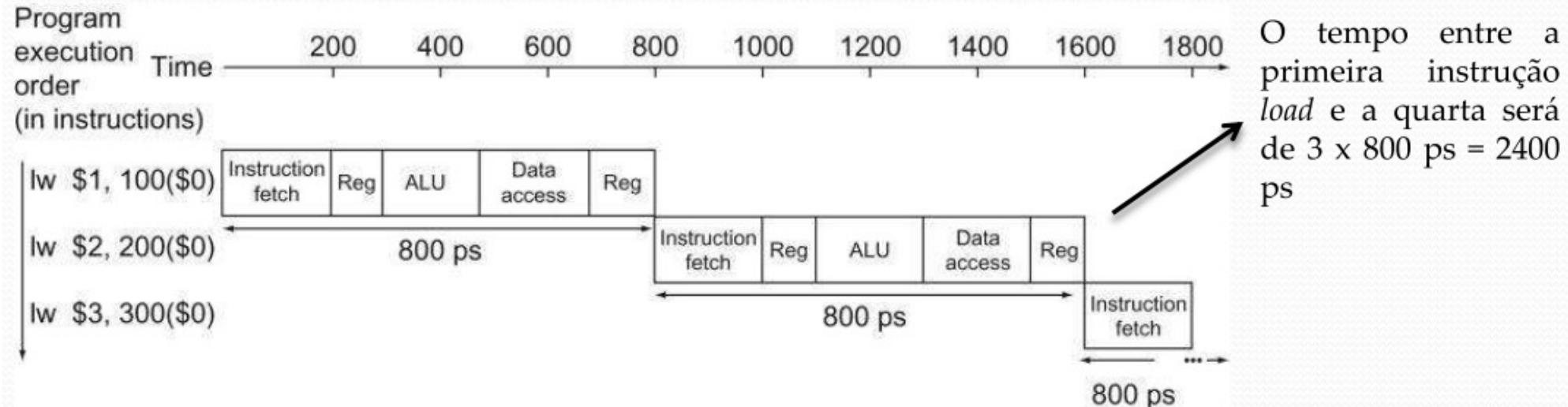
Tempo de operação:

- 200ps para acesso à memória;
- 200ps para operação da ALU;
- 100ps para leitura ou escrita no arquivo de registradores.

Tempo de execução das instruções:

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (<i>lw</i>)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (<i>sw</i>)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, <i>slt</i>)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (<i>beq</i>)	200 ps	100 ps	200 ps			500 ps

Evolução do Processador



O tempo entre a primeira instrução *load* e a quarta será de $3 \times 800 \text{ ps} = 2400 \text{ ps}$

Com *pipeline*, a quarta instrução começará a ser executada 600 ps após a primeira.

Evolução do Processador

Speed-up:

- Se os estágios da *pipeline* forem perfeitamente balanceados (i.e., consomem o mesmo tempo), então o tempo entre instruções em um processador com *pipeline*, assumindo condições ideais de operação, será:

$$T_{\text{pipeline}} = \frac{T_{\text{sem pipeline}}}{\text{Número de estágios da pipeline}}$$

- Sob condições ideais e para um número suficientemente elevado de instruções, o *speed-up* é aproximadamente igual ao número de estágios da *pipeline*.
- A expressão sugere que uma *pipeline* com 5 estágios deveria oferecer um aprimoramento por um fator de 5 em relação aos 800 ps da versão sem *pipeline*.
- No entanto, os estágios podem não estar perfeitamente平衡ados. Ademais, a estratégia de *pipeline* introduz algum *overhead*.
- Por isso, o ganho real observado é um pouco inferior ao máximo atingível.

Evolução do Processador

No exemplo anterior, o tempo de execução das três instruções caiu de 2400ps (uniciclo) para 1400ps (*pipeline*).

E se executássemos mais um milhão de instruções?

➤ *Pipeline:*

- Cada instrução acrescenta 200ps ao tempo total.
- Então, $1.000.000 \times 200\text{ps} + 1400\text{ps} = 200.001.400 \text{ ps}$.

Fator 4

➤ **Uniciclo:**

- Cada instrução acrescenta 800ps ao tempo total.
- $2400\text{ps} + 1.000.000 \times 800\text{ps} = 800.002.400 \text{ ps}$.

➤ **Ganho:** $800.002.400 / 200.001.400 \approx 4,0$

Evolução do Processador

A organização de um processador em *pipeline* não altera o tempo de execução de uma única instrução, mas sim a taxa ou vazão de instruções (*throughput*) que o processador consegue atingir.

Throughput: trata-se de uma métrica essencial, uma vez que os programas exigem, de uma maneira geral, a execução de milhões a bilhões de instruções.

Evolução do Processador

O conjunto de instruções MIPS foi planejado visando uma implementação “eficiente” com *pipeline*.

- **Todas as instruções possuem o mesmo tamanho (32 bits):**
 - Facilita a busca e a decodificação, que podem ser feitas em um ciclo cada (no 1º e 2º estágios da *pipeline*, respectivamente).
- **Contra-exemplo:** arquitetura x86 – instruções variam de 1 a 15 bytes.
 - **Curiosidade:** algumas implementações mais recentes desta arquitetura transformam as instruções x86 em operações mais simples – que se parecem com instruções MIPS – e, então, fazem a *pipeline* destas operações mais elementares.

Evolução do Processador

MIPS possui apenas alguns formatos de instrução com certa regularidade:

- Os campos dos registradores que fornecem operandos estão localizados no mesmo lugar em cada instrução.
- Esta simetria abre a possibilidade de o segundo estágio da *pipeline* iniciar a leitura do arquivo de registrador ao mesmo tempo em que o *hardware* está identificando a instrução recebida.

Evolução do Processador

Operandos em memória somente aparecem nas instruções *load* e *store*:

- Esta restrição de projeto possibilita que o 3º estágio (execução) seja utilizado para o cálculo do endereço de memória e a referida posição seja acessada no estágio seguinte.
- Se fosse permitida a especificação de operandos em memória, como acontece na arquitetura x86, os estágios 3 e 4 seriam expandidos em: (1) estágio de endereço, (2) estágio de acesso à memória e (3) estágio de execução.

Evolução do Processador

Alinhamento dos operandos em memória:

- No MIPS, as palavras sempre iniciam em endereços de memória que são múltiplos de 4.
- Por causa desta restrição (ou garantia), um acesso à memória pode ser feito em apenas um estágio.

Evolução do Processador

Há situações especiais em que a próxima instrução do programa não pode ser executada no ciclo seguinte. Estes eventos são chamados de **hazards**.

Veremos três tipos:

- A. **Estruturais**
- B. **Dados**
- C. **Controle**

Hazard Estrutural

Situação de conflito pelo uso (simultâneo) de um mesmo recurso de *hardware*.

Ocorre quando duas instruções precisam utilizar o mesmo componente de *hardware* – para fins distintos ou com dados diferentes – no mesmo ciclo de relógio.

A arquitetura MIPS foi pensada tendo em vista uma implementação em *pipeline*. Por isso, as escolhas feitas facilitam a tarefa dos projetistas em evitar os *hazards* estruturais.

Exemplo: caso não houvesse duas memórias distintas (dados e instrução), teríamos um *hazard* estrutural no momento em que uma instrução estivesse no 4º estágio da *pipeline* (para efetuar um acesso à memória) e uma nova instrução estivesse para ser buscada.

Hazard de Dados

Ocorrem quando uma instrução depende da conclusão de uma instrução prévia que ainda esteja na *pipeline* para realizar sua operação e/ou acessar um dado.

Exemplo:

```
add $s0, $t0, $t1
```

```
sub $t2, $s0, $t3
```

- A instrução *add* somente escreve seu resultado no final do 5º estágio da *pipeline*.
- Logo, teríamos que desperdiçar três ciclos de relógio aguardando até que o resultado correto (\$s0) pudesse ser lido pela instrução *sub*.

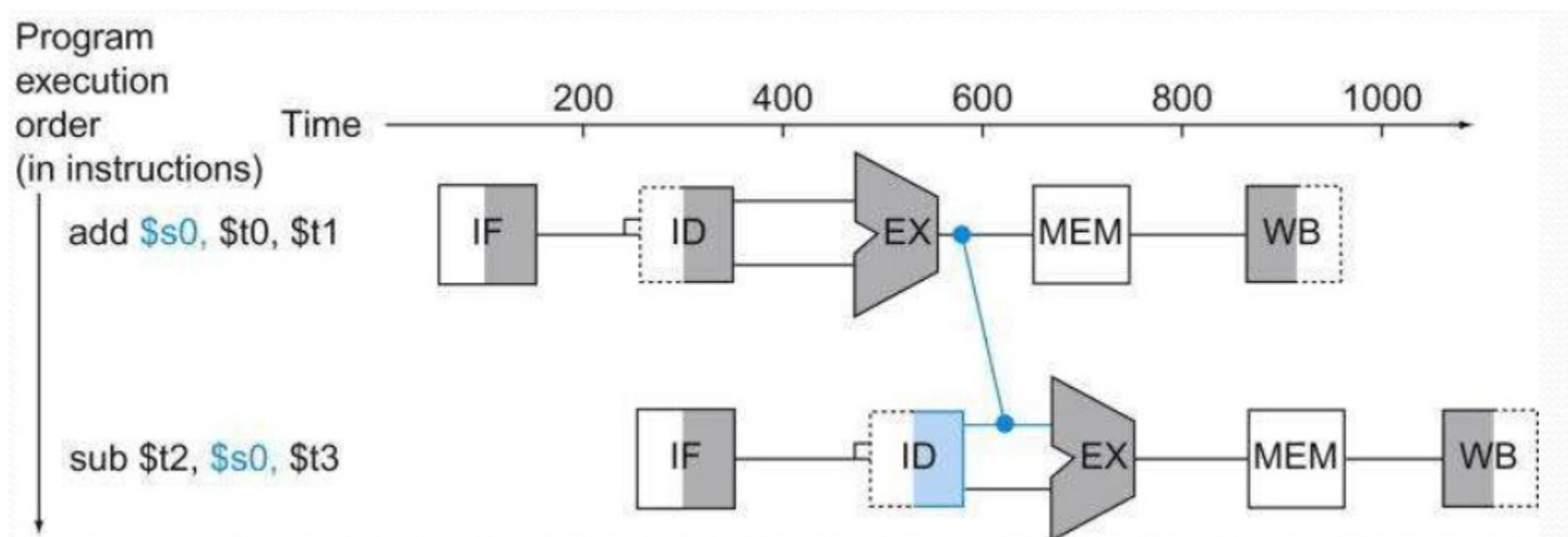
Hazard de Dados

Uma possível solução pode ser proposta a partir da observação de que não é necessário aguardar até que a instrução termine sua execução.

No exemplo anterior, assim que a ALU cria o resultado da soma prevista pela instrução *add*, este valor poderia ser passado como entrada para a subtração no ciclo seguinte.

A inserção de *hardware* extra para recuperar uma informação referente a uma instrução passada caracteriza a estratégia conhecida como *forwarding* ou *bypassing*.

Hazard de Dados



Hazard de Controle

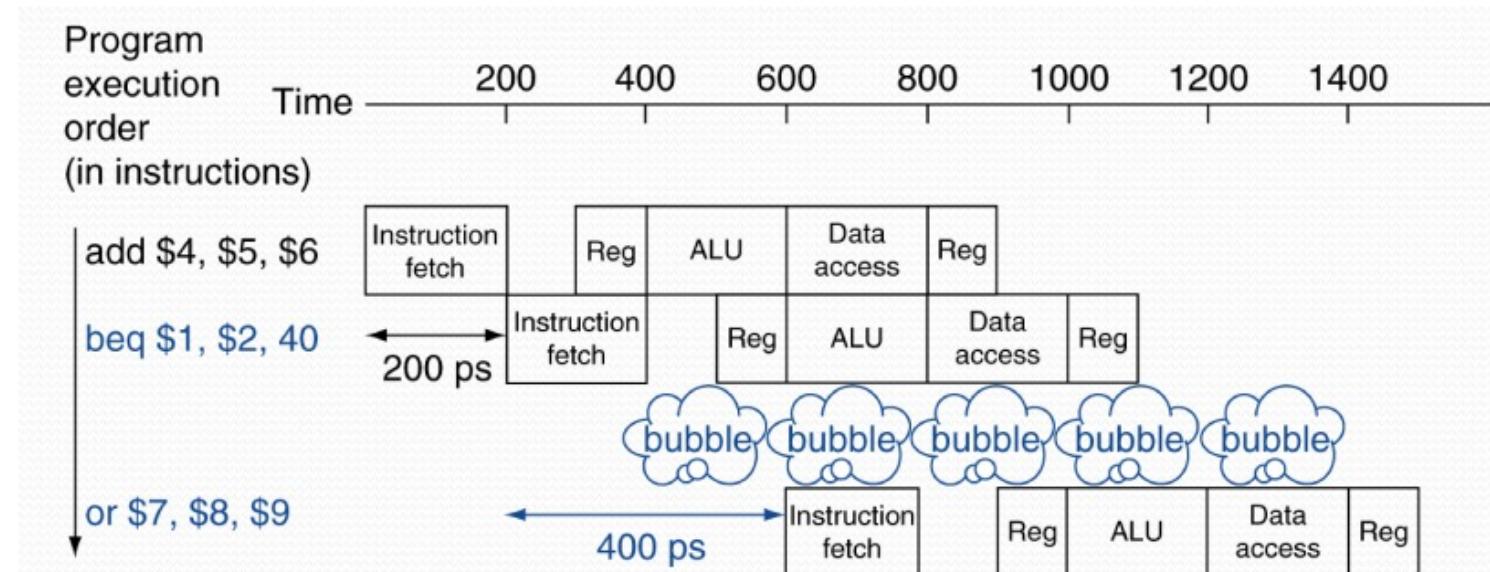
Surge por causa da necessidade de tomar uma decisão baseada em resultados de uma instrução enquanto outras estão em execução.

Ou seja, está ligado a instruções do tipo *branch*.

Problema:

- A *pipeline* inicia a busca da instrução subsequente ao *branch* no próximo ciclo de relógio.
- Porém, não há como a *pipeline* saber qual é a instrução correta a ser buscada, uma vez que acabou de receber o próprio *branch* da memória.

Hazard de Controle



Mesmo que o desvio não seja tomado, haverá um atraso de 200ps.

Este cenário fica mais dramático se não for possível resolver o desvio logo no segundo estágio da *pipeline*.

Para *pipelines* mais longas, o atraso devido aos *stalls* se torna inaceitável.