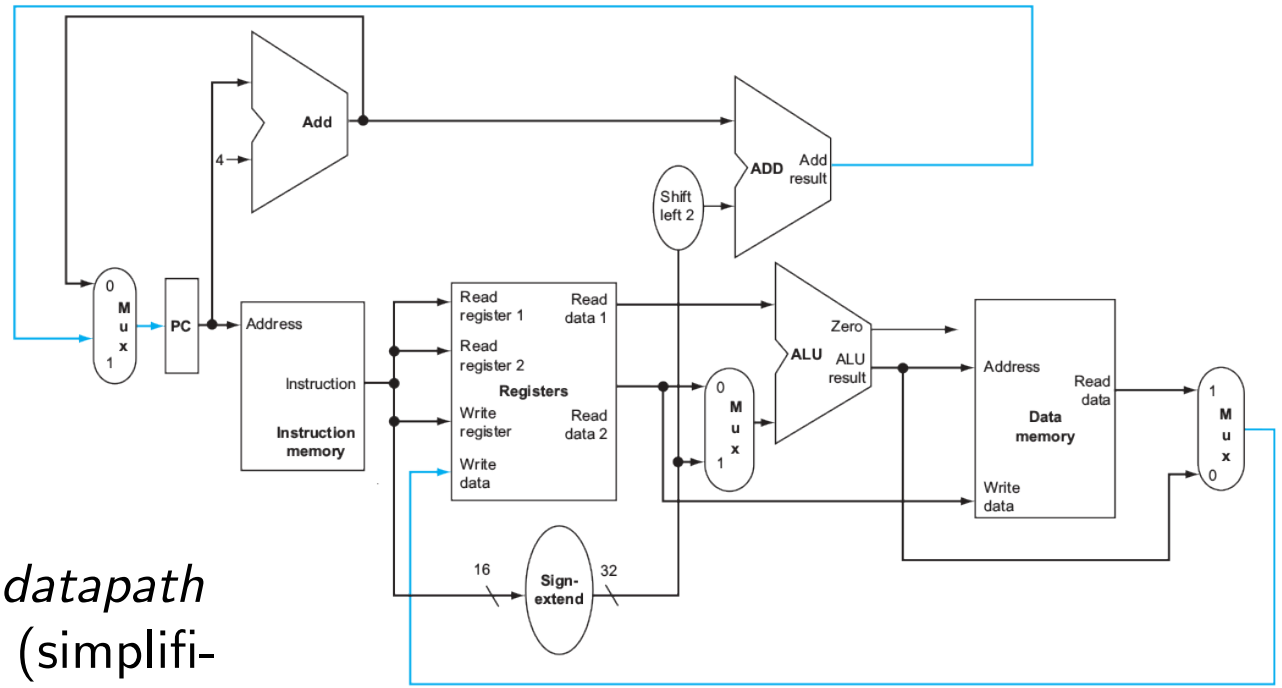
Arquitetura de Computadores
ACH2055
Aula 07 – *Pipeline* e seus Conflitos

Norton Trevisan Roman
(norton@usp.br)

15 de outubro de 2019

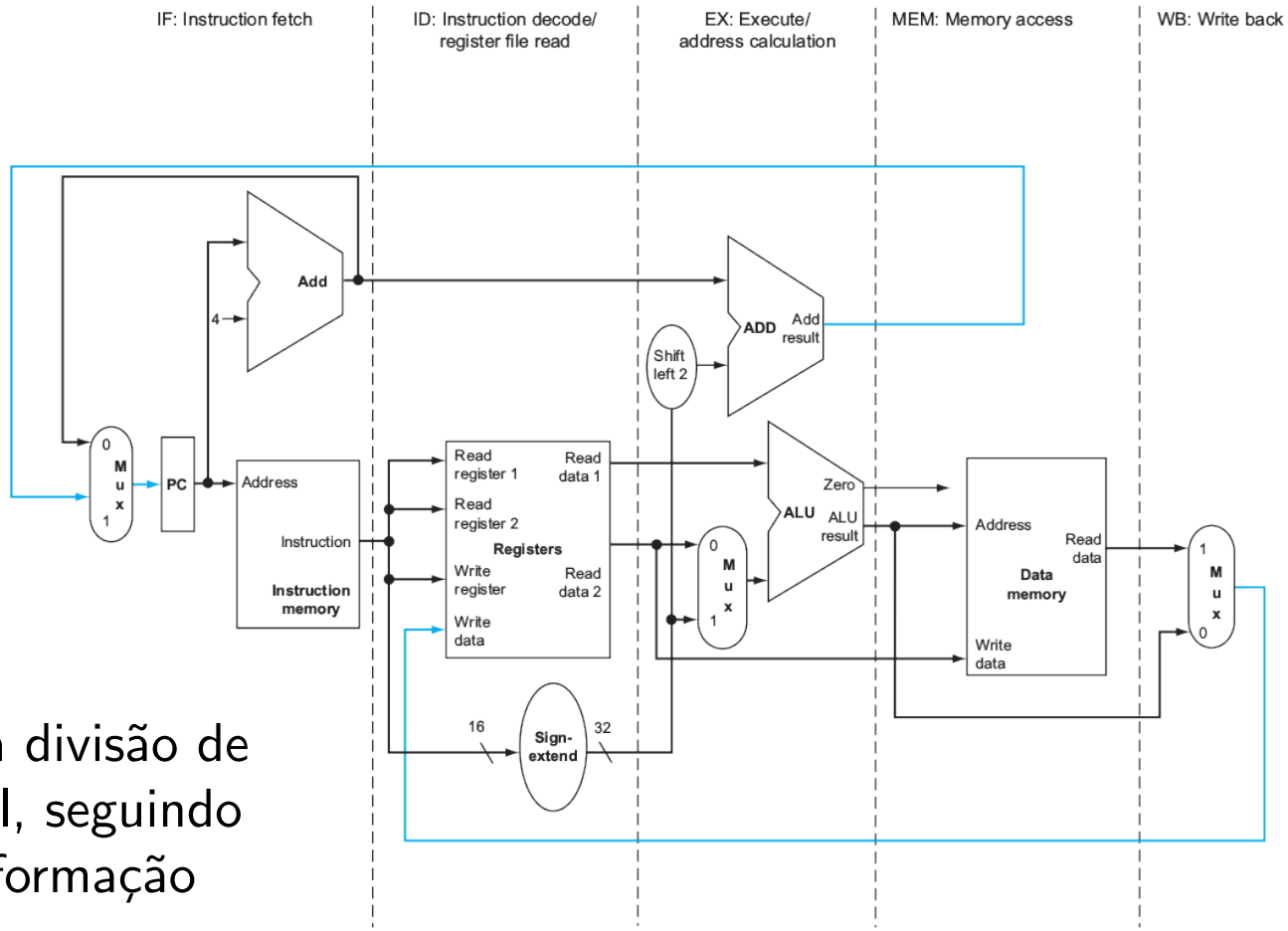
Organizando o Modelo de Ciclo Único



Voltemos ao *datapath* de ciclo único (simplificado e sem `j`) já visto

Fonte: Adaptado de [1]

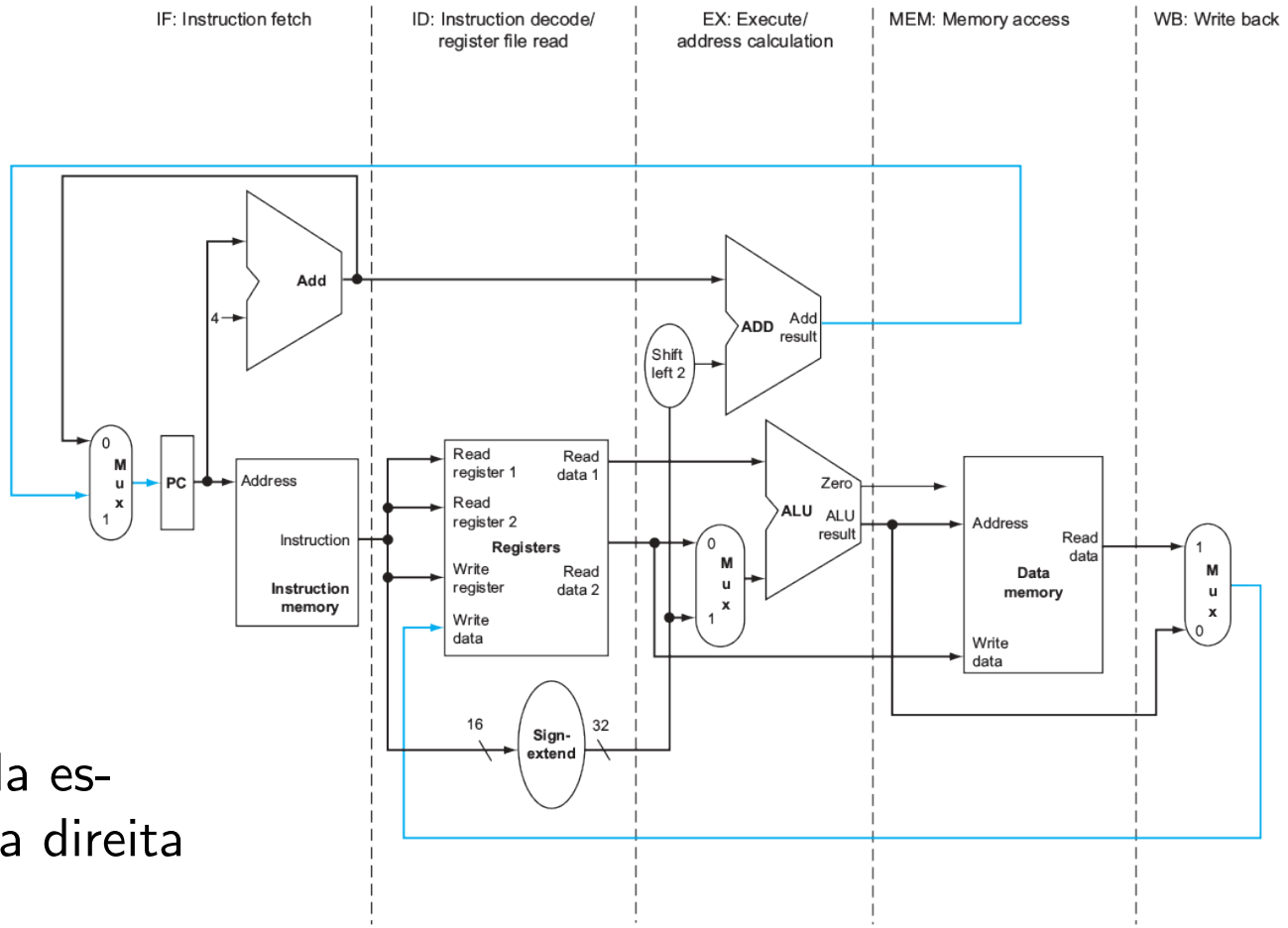
Organizando o Modelo de Ciclo Único



Existe nele uma divisão de estágios natural, seguindo o fluxo da informação

Fonte: Adaptado de [1]

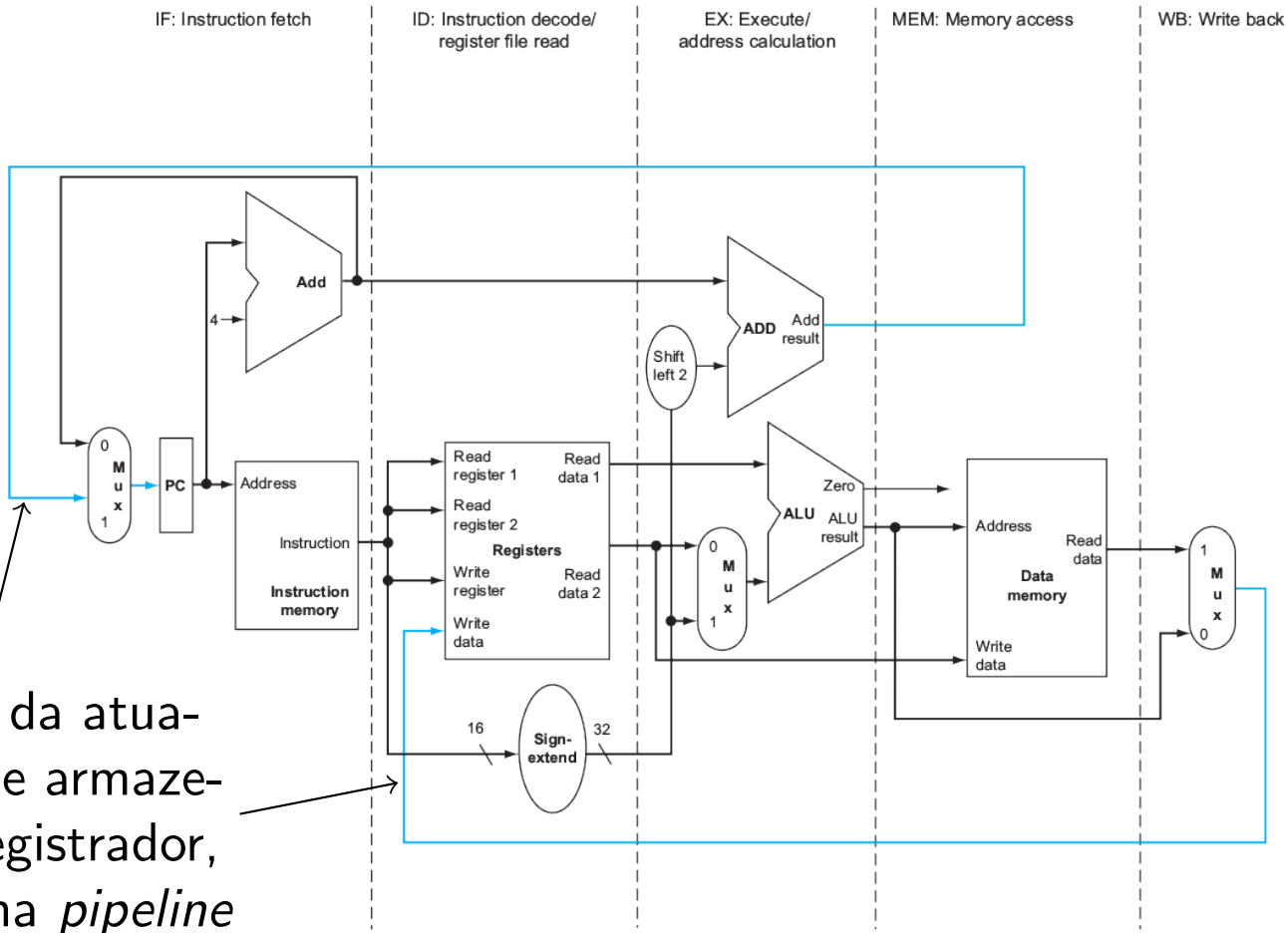
Organizando o Modelo de Ciclo Único



Que vai da esquerda para a direita

Fonte: Adaptado de [1]

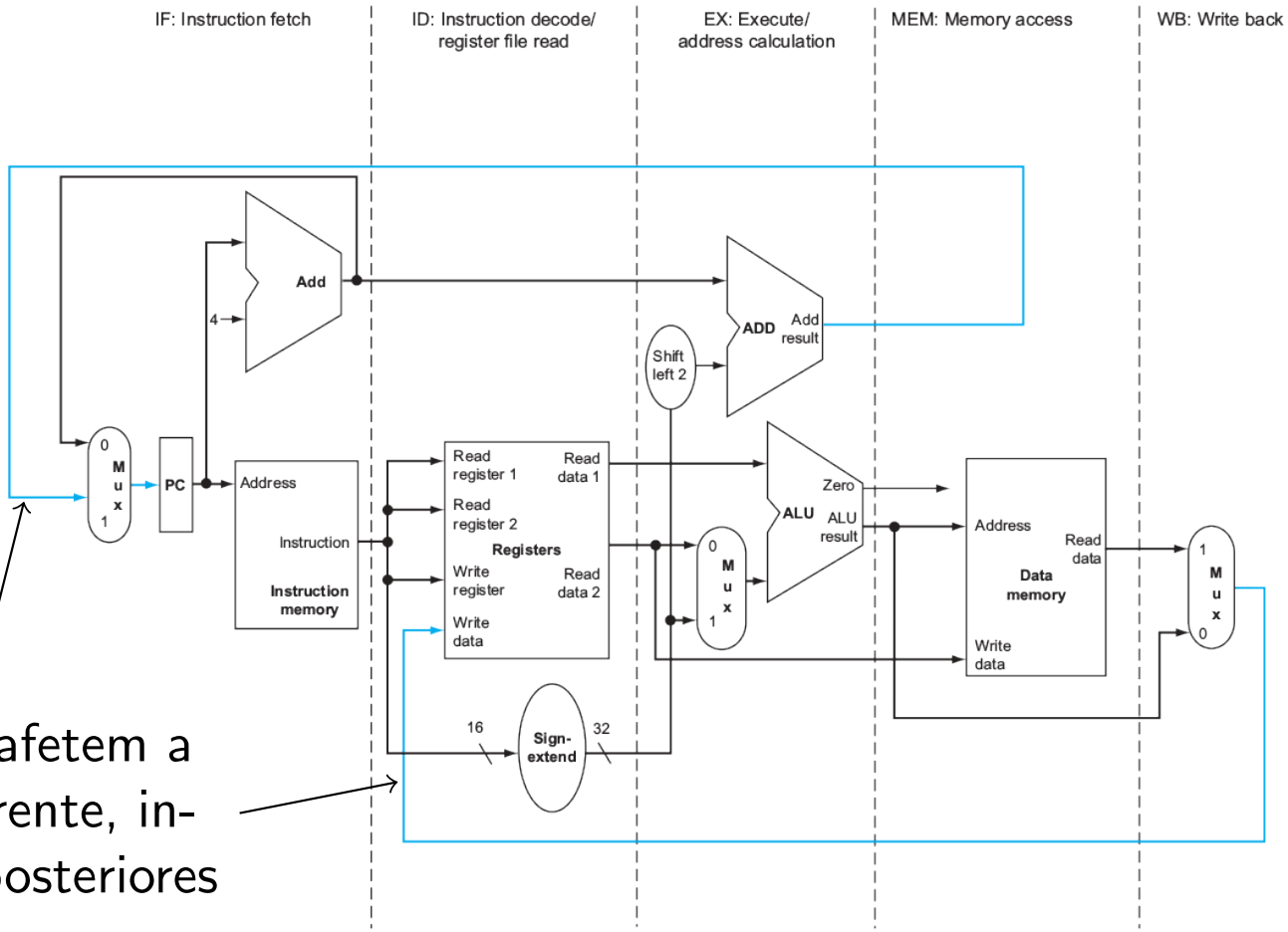
Organizando o Modelo de Ciclo Único



Com exceção da atualização do PC e armazenamento no registrador, que "voltam" na *pipeline*

Fonte: Adaptado de [1]

Organizando o Modelo de Ciclo Único

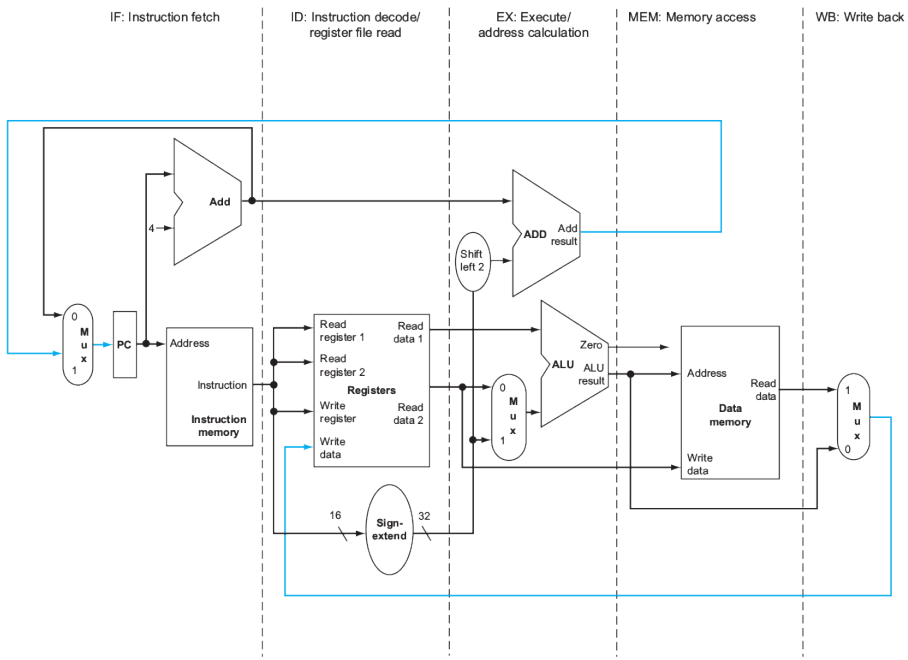


Embora não afetem a instrução corrente, influenciam as posteriores

Fonte: Adaptado de [1]

Organizando o Modelo de Ciclo Único

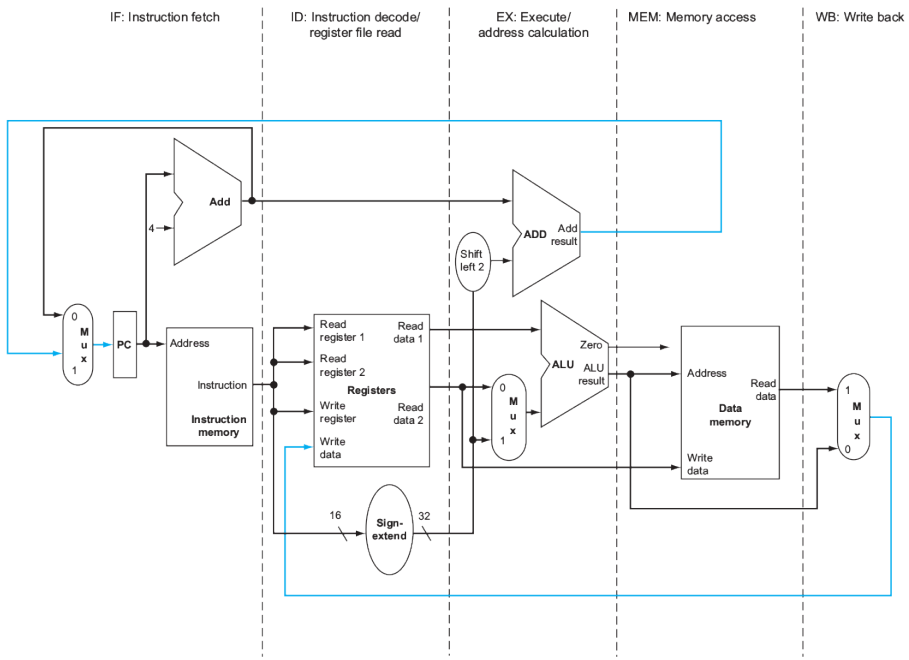
- Separar em estágios, contudo, não é simples



Fonte: [1]

Organizando o Modelo de Ciclo Único

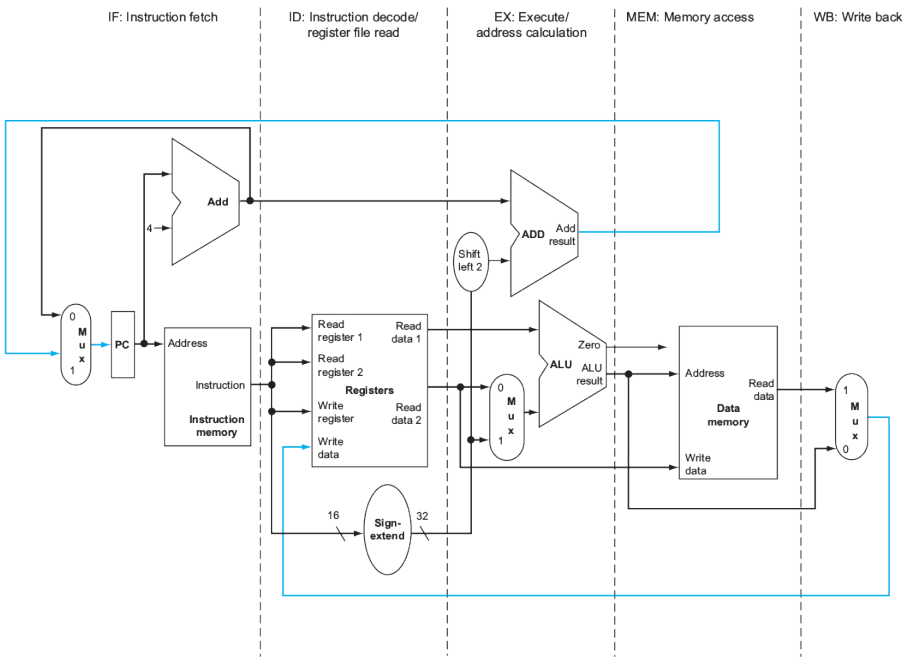
- Separar em estágios, contudo, não é simples
  - Rodaremos instruções diferentes em cada estágio



Fonte: [1]

Organizando o Modelo de Ciclo Único

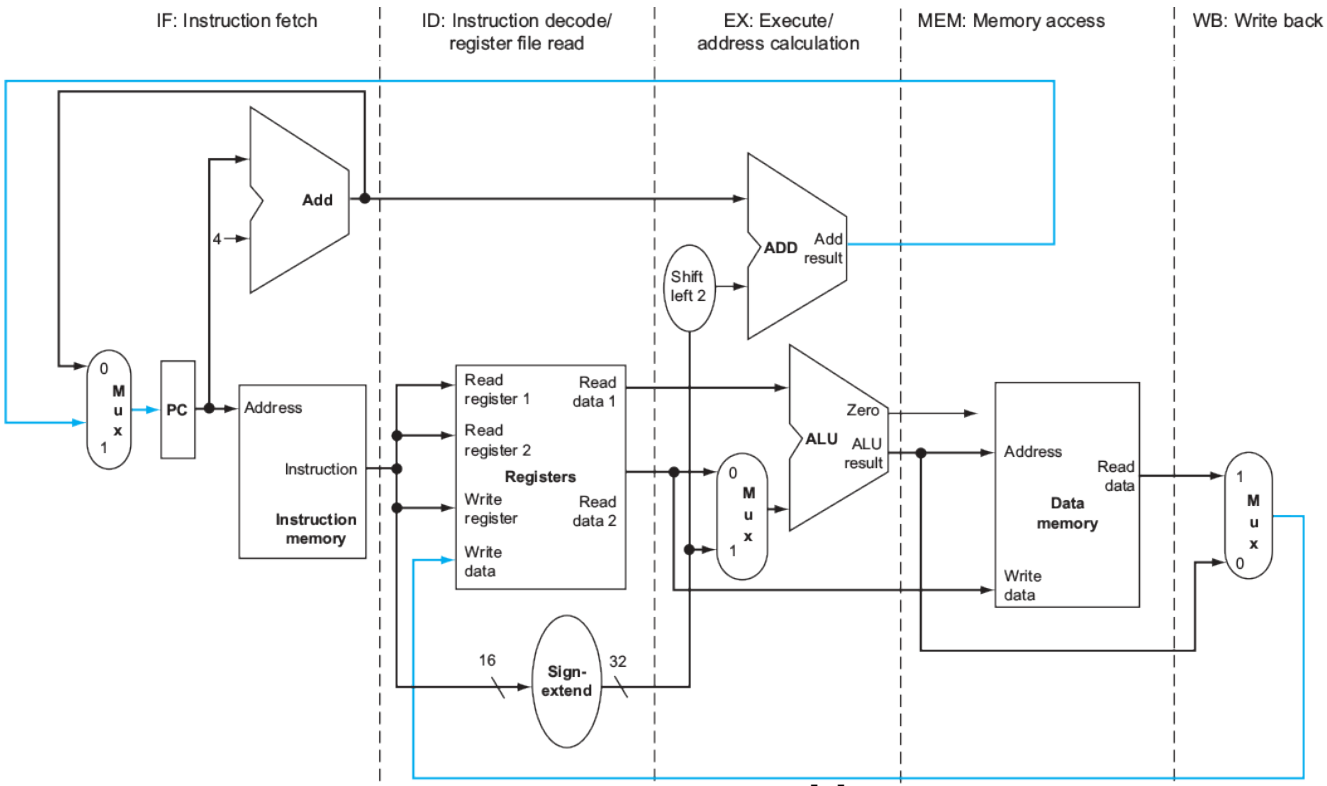
- Separar em estágios, contudo, não é simples
  - Rodaremos instruções diferentes em cada estágio
  - Precisamos então reter os valores parciais de cada instrução, para que possam ser usados nos estágios seguintes



Fonte: [1]

Organizando o Modelo de Ciclo Único

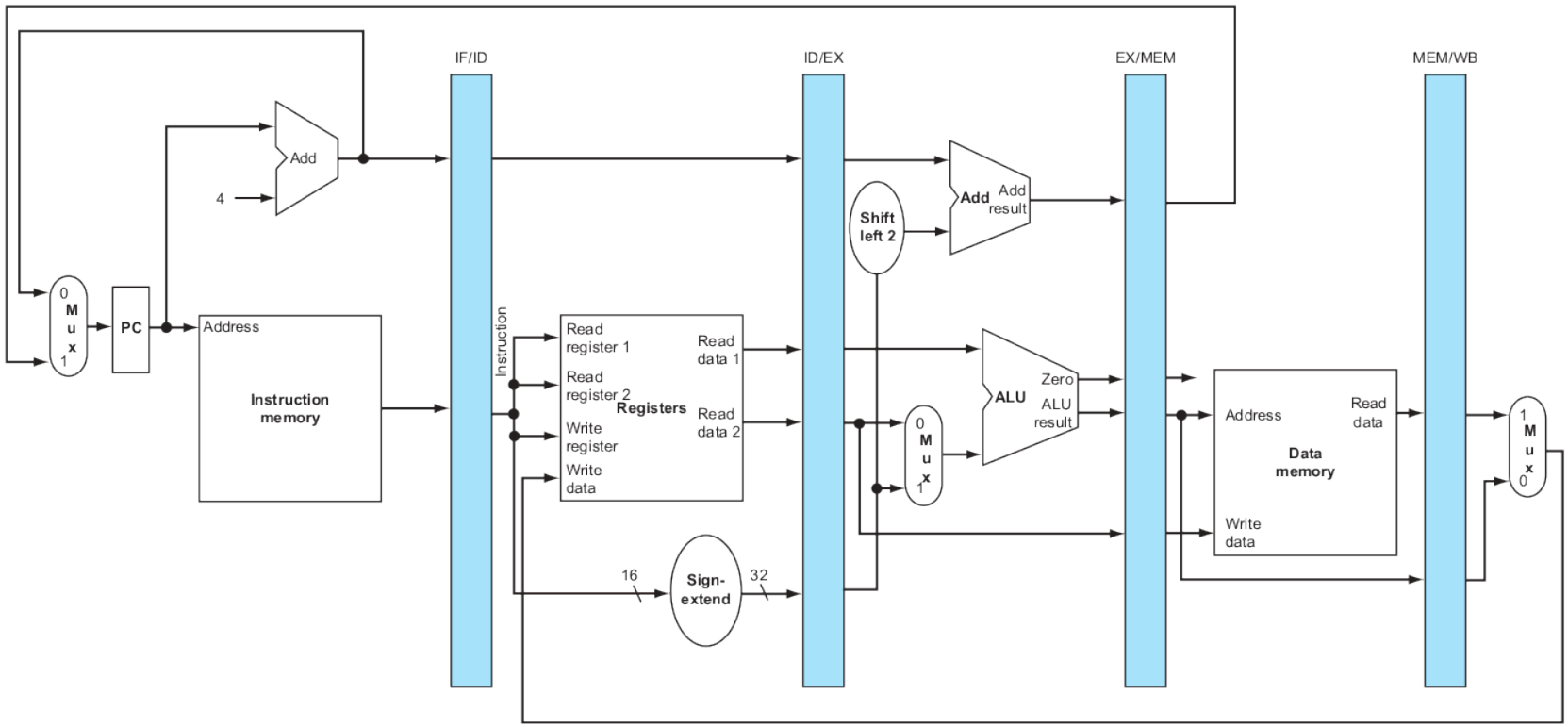
- Precisamos salvar esses valores em registradores



Fonte: Adaptado de [1]

Organizando o Modelo de Ciclo Único

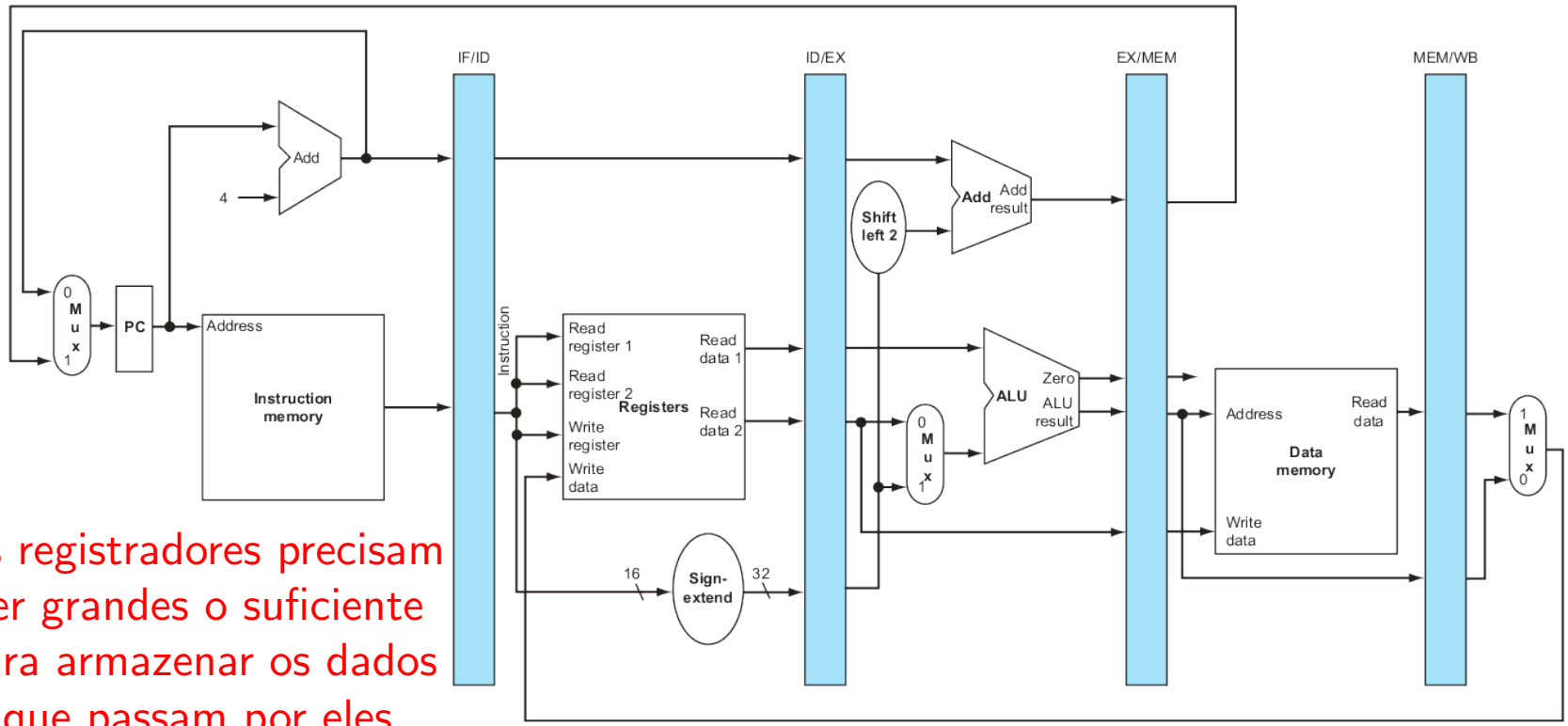
- Precisamos salvar esses valores em registradores



Fonte: [1]

Organizando o Modelo de Ciclo Único

- Precisamos salvar esses valores em registradores

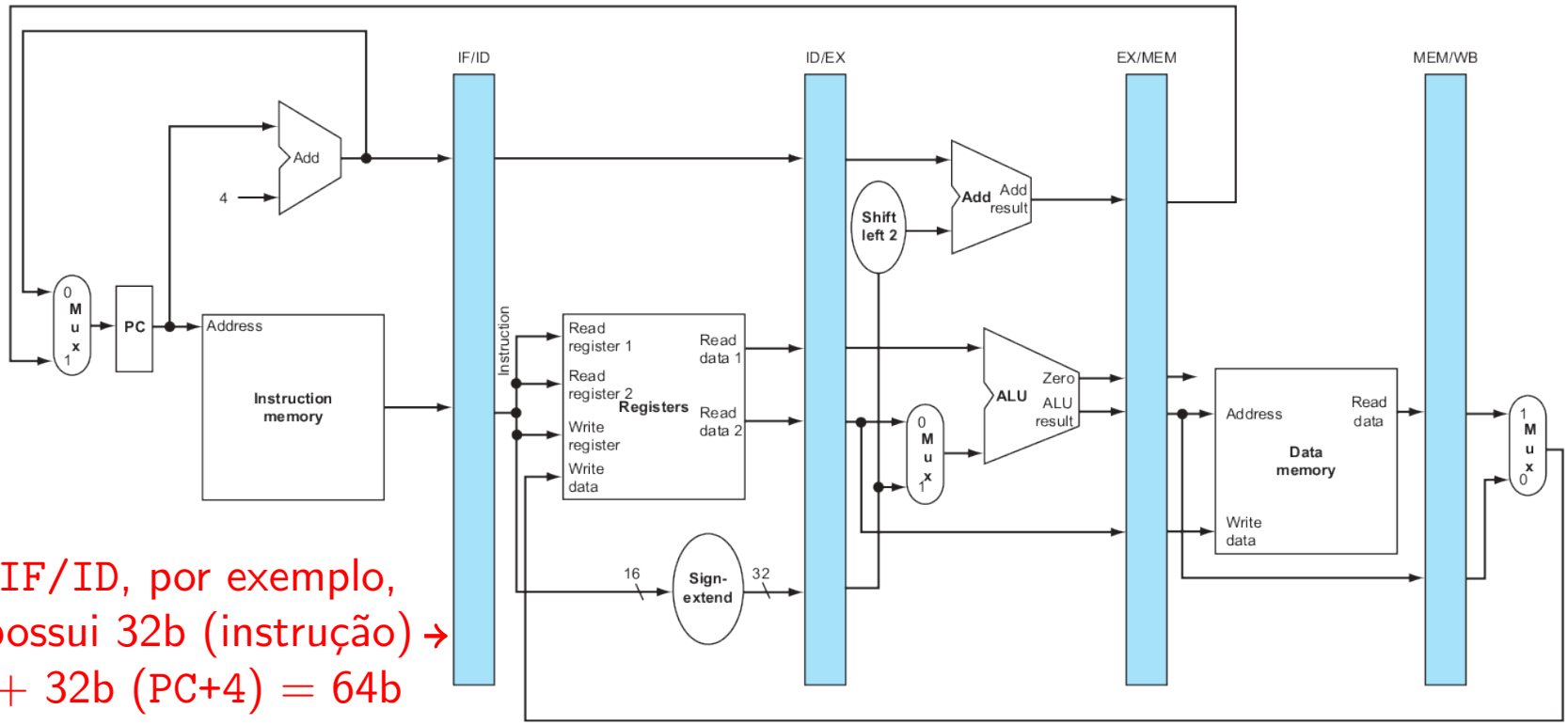


Os registradores precisam ser grandes o suficiente para armazenar os dados que passam por eles

Fonte: [1]

Organizando o Modelo de Ciclo Único

- Precisamos salvar esses valores em registradores

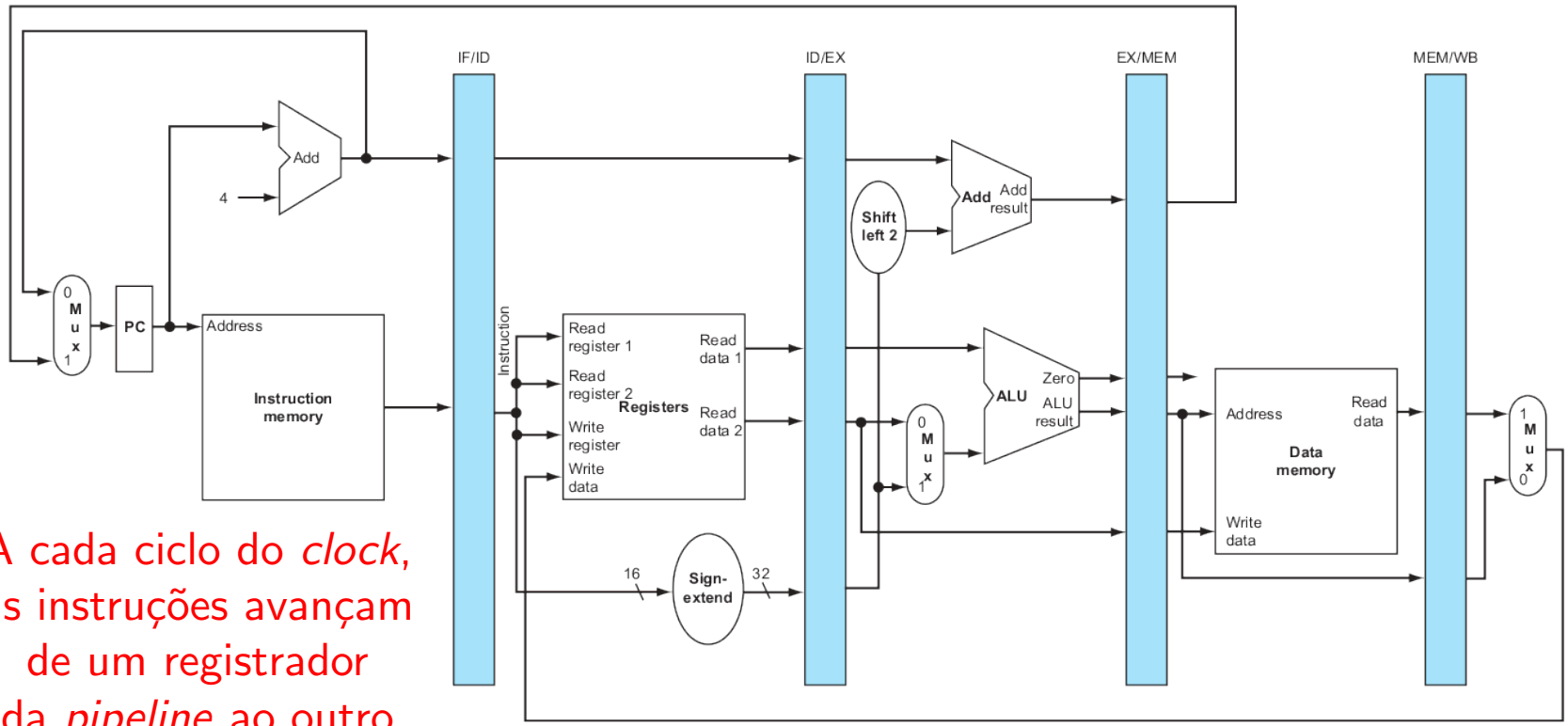


IF/ID, por exemplo, possui 32b (instrução) → + 32b (PC+4) = 64b

Fonte: [1]

Organizando o Modelo de Ciclo Único

- Precisamos salvar esses valores em registradores

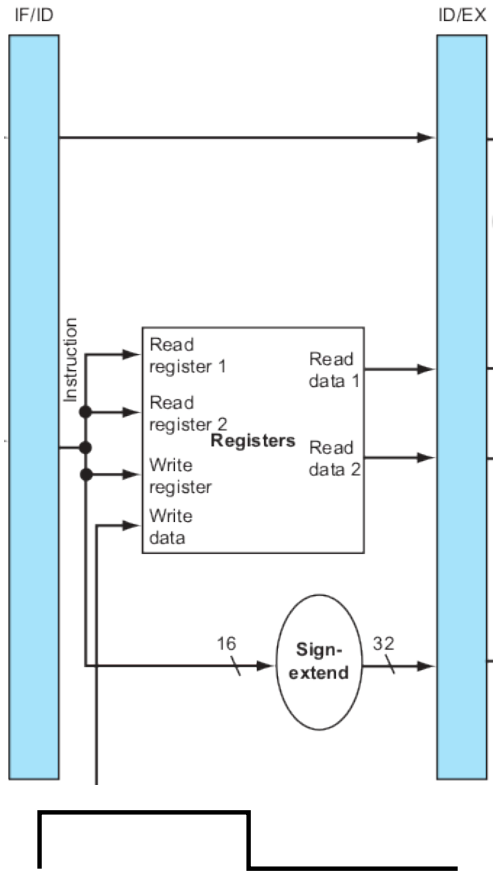


A cada ciclo do *clock*, as instruções avançam de um registrador da *pipeline* ao outro

Fonte: [1]

Organizando o Modelo de Ciclo Único

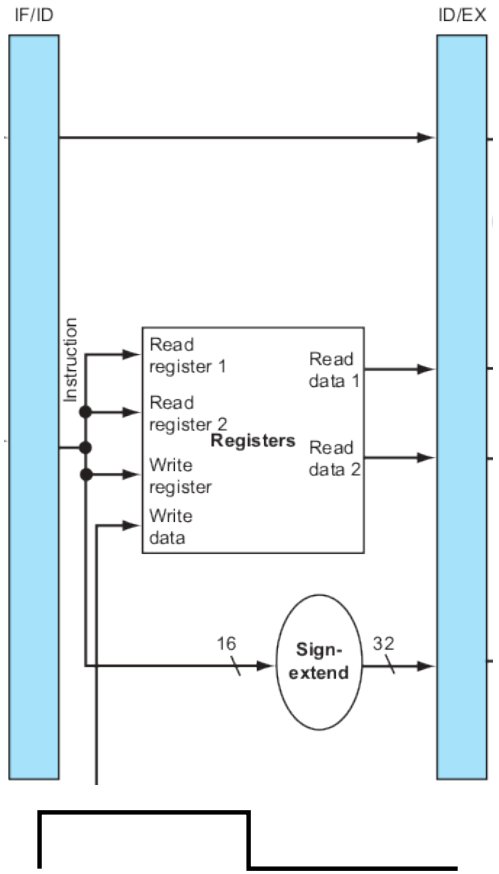
- Mas se cada ciclo comanda a saída e entrada de registradores da *pipeline*, quando o arquivo de registradores será atualizado, já que está dentro de um estágio?



Fonte: Adaptado de [1]

Organizando o Modelo de Ciclo Único

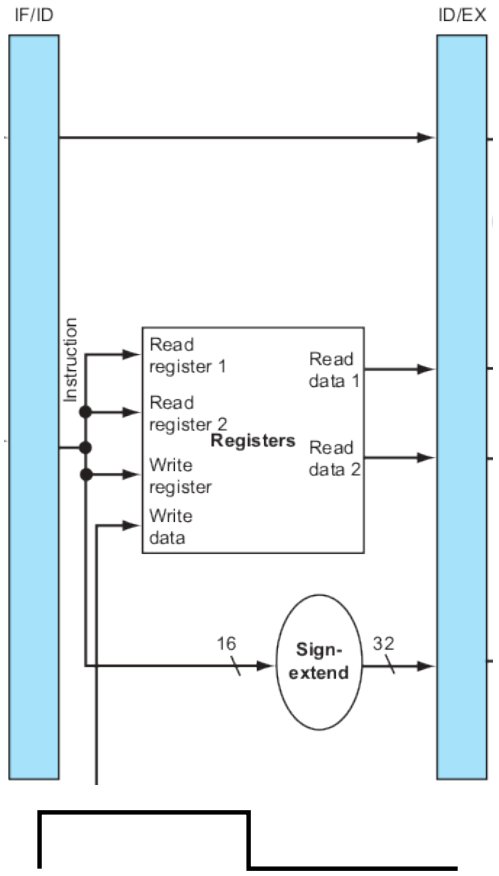
- Mas se cada ciclo comanda a saída e entrada de registradores da *pipeline*, quando o arquivo de registradores será atualizado, já que está dentro de um estágio?
  - Podemos esperar a próxima borda do ascendente *clock*, mas isso pode gerar algum atraso (como veremos mais adiante)



Fonte: Adaptado de [1]

Organizando o Modelo de Ciclo Único

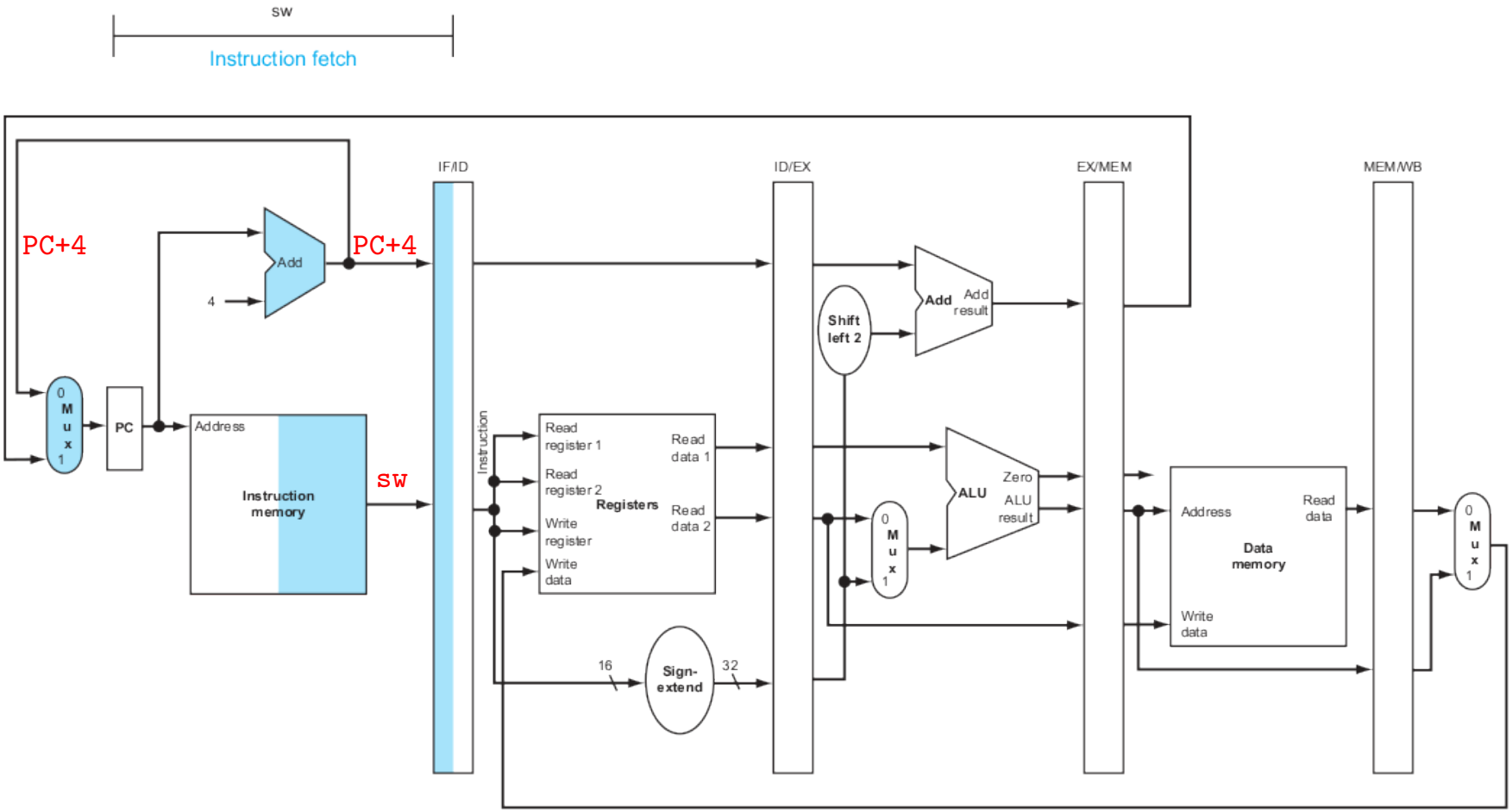
- Alternativamente, podemos mudar o projeto
  - Permitindo que o arquivo seja escrito e lido no mesmo ciclo de *clock*
  - A escrita ocorrendo na primeira metade, e a leitura na segunda
  - Assim, a leitura retorna o que foi escrito
  - Basta para isso usar a borda descendente no arquivo de registradores



Fonte: Adaptado de [1]

Pipeline

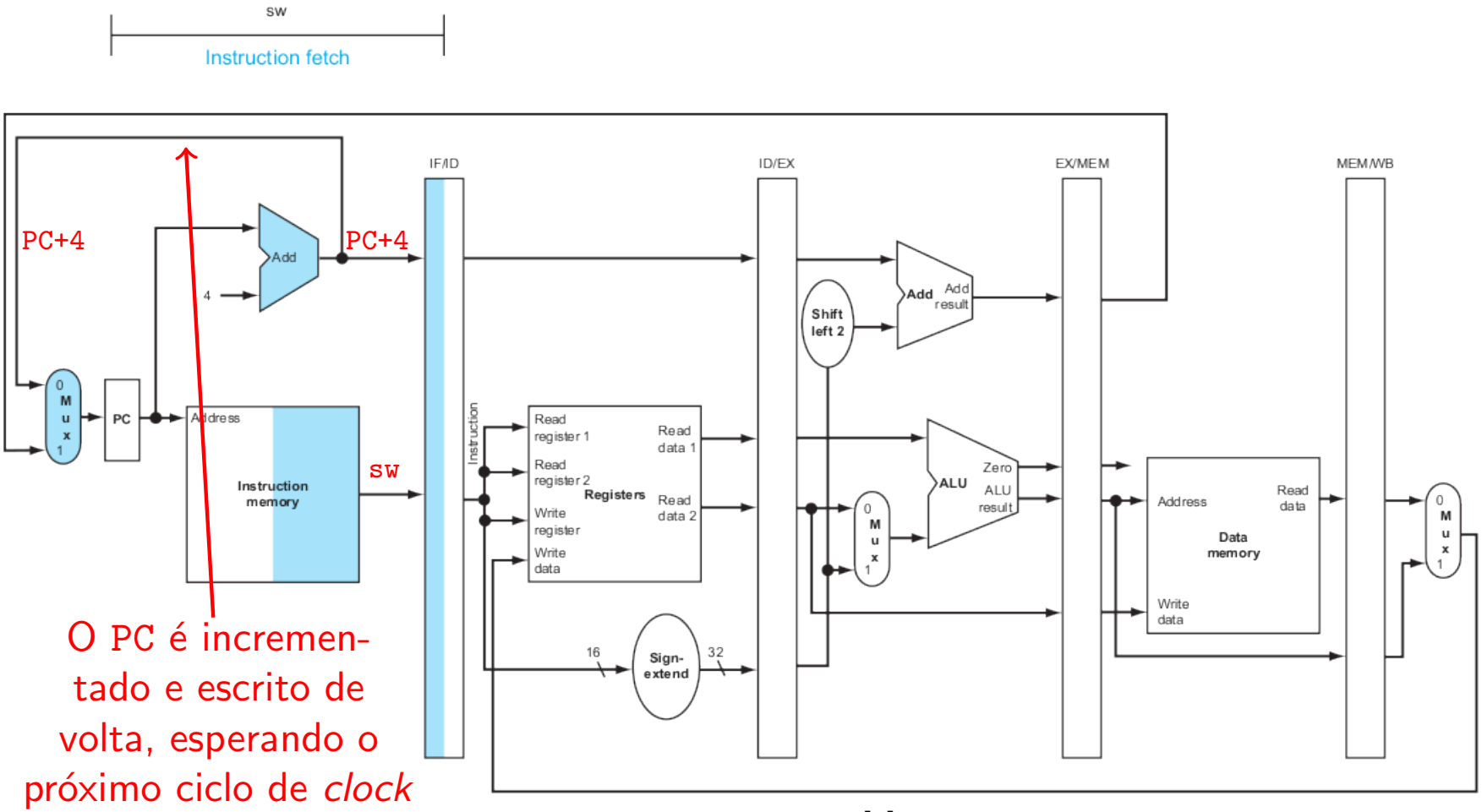
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

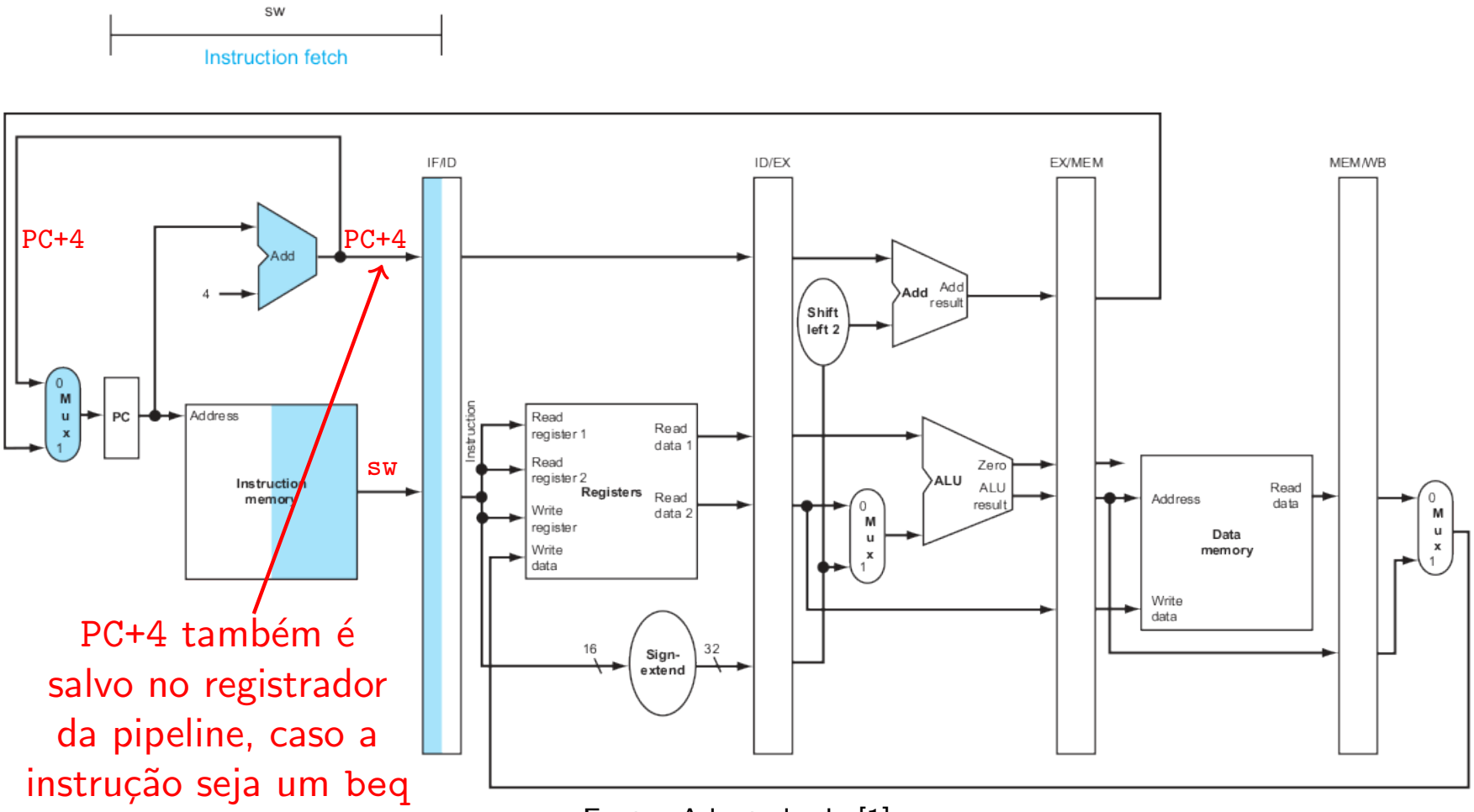


O `PC` é incrementado e escrito de volta, esperando o próximo ciclo de *clock*

Fonte: Adaptado de [1]

Pipeline

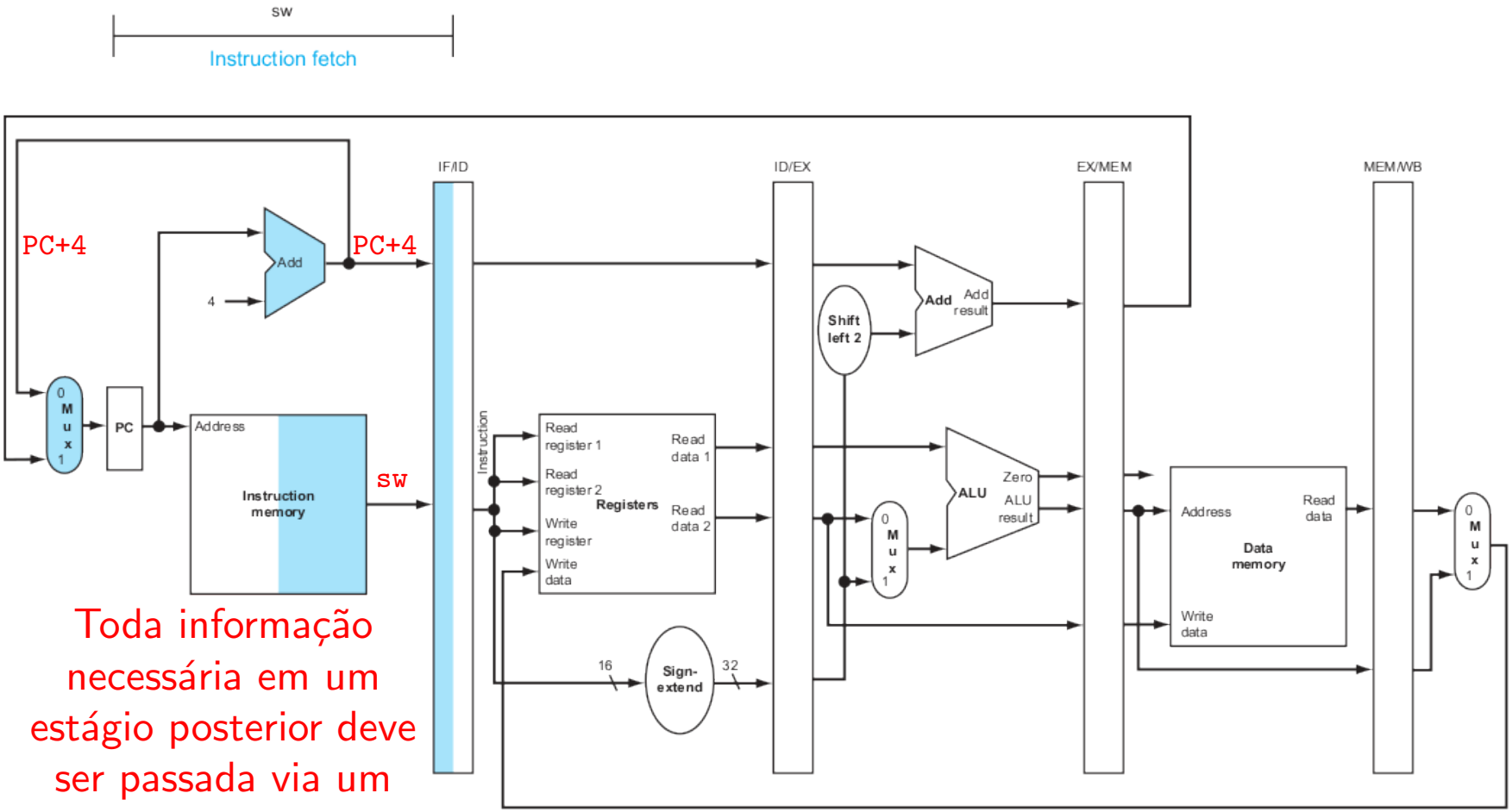
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

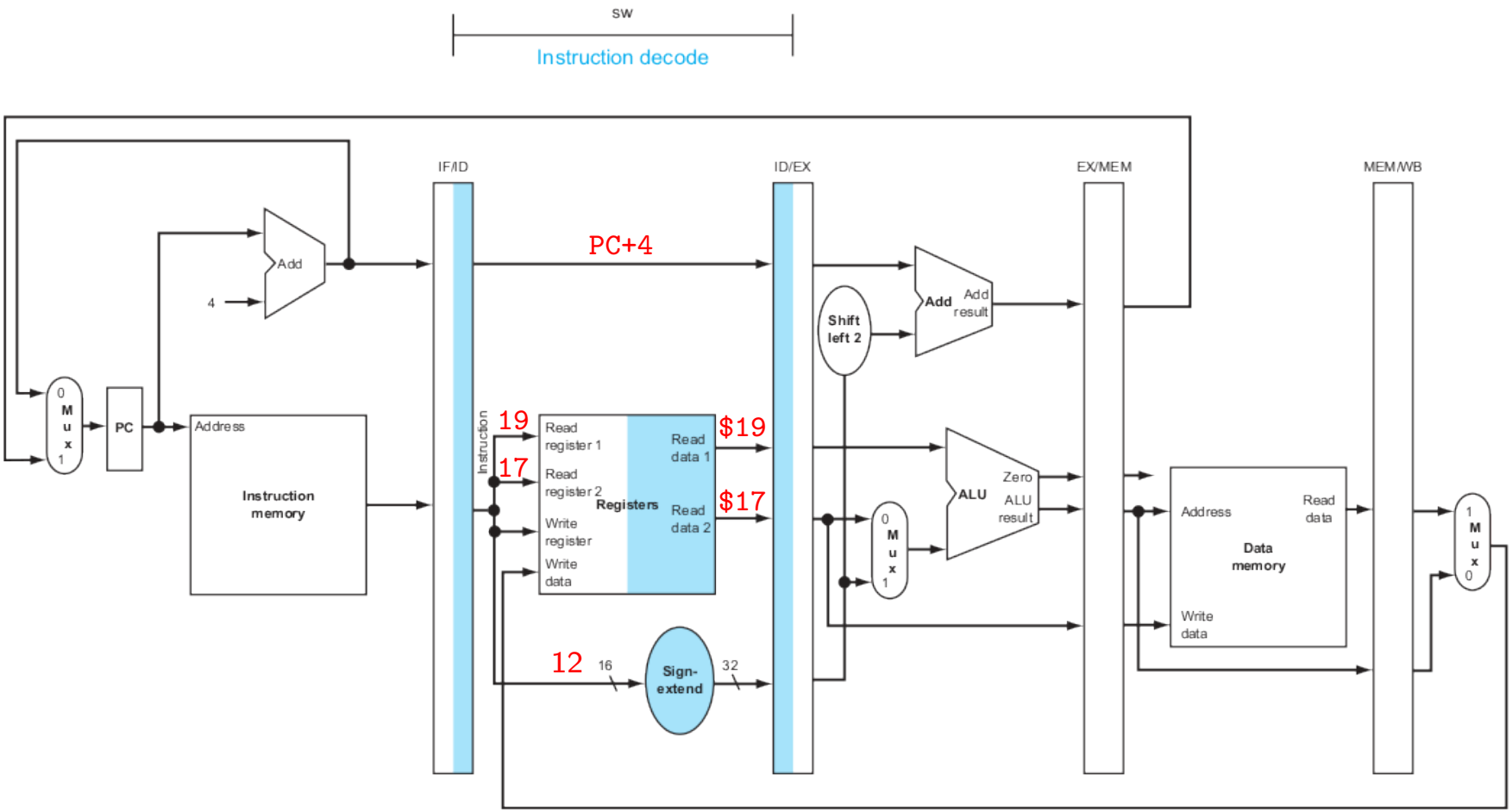


Toda informação necessária em um estágio posterior deve ser passada via um registrador da pipeline

Fonte: Adaptado de [1]

Pipeline

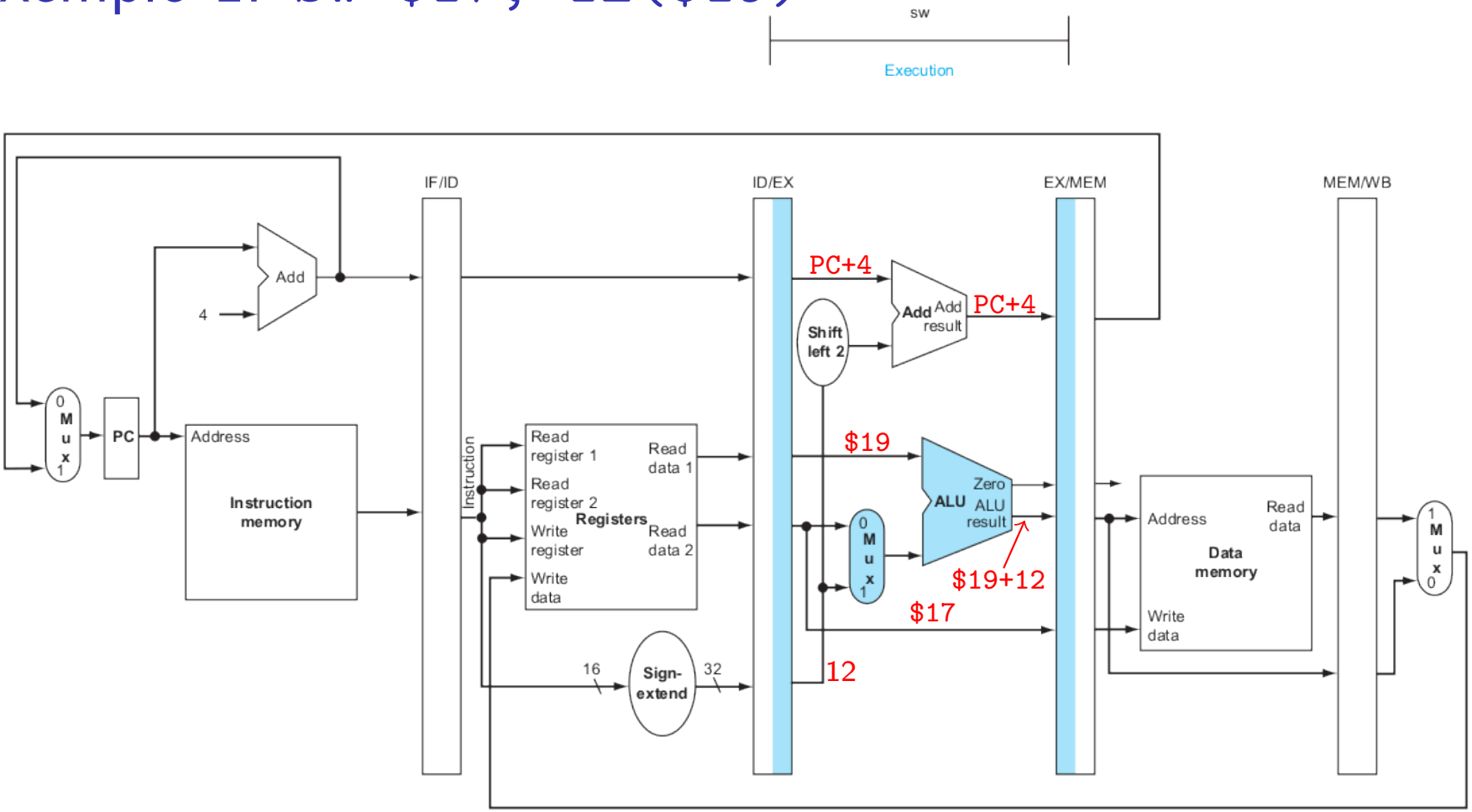
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

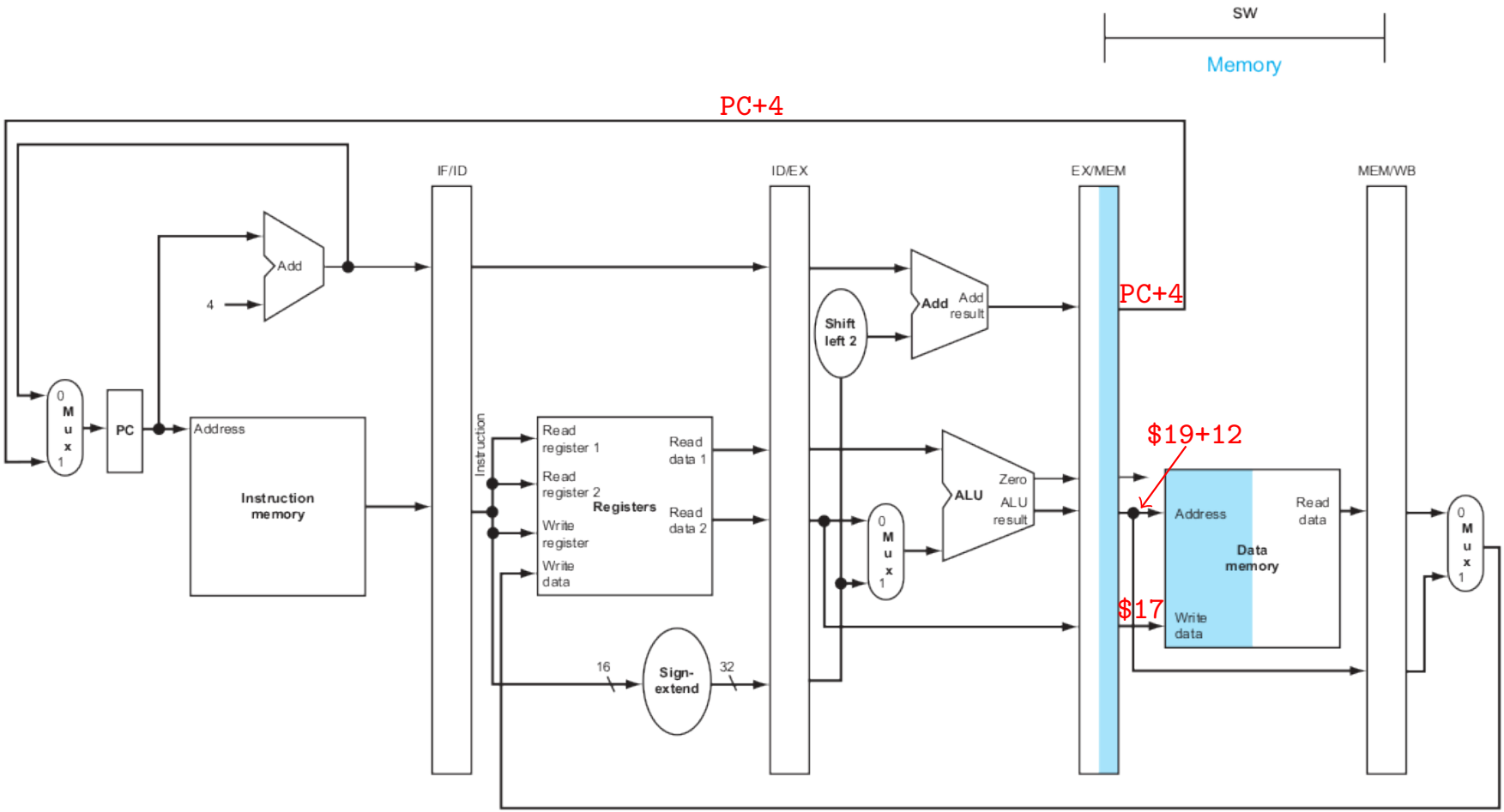
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

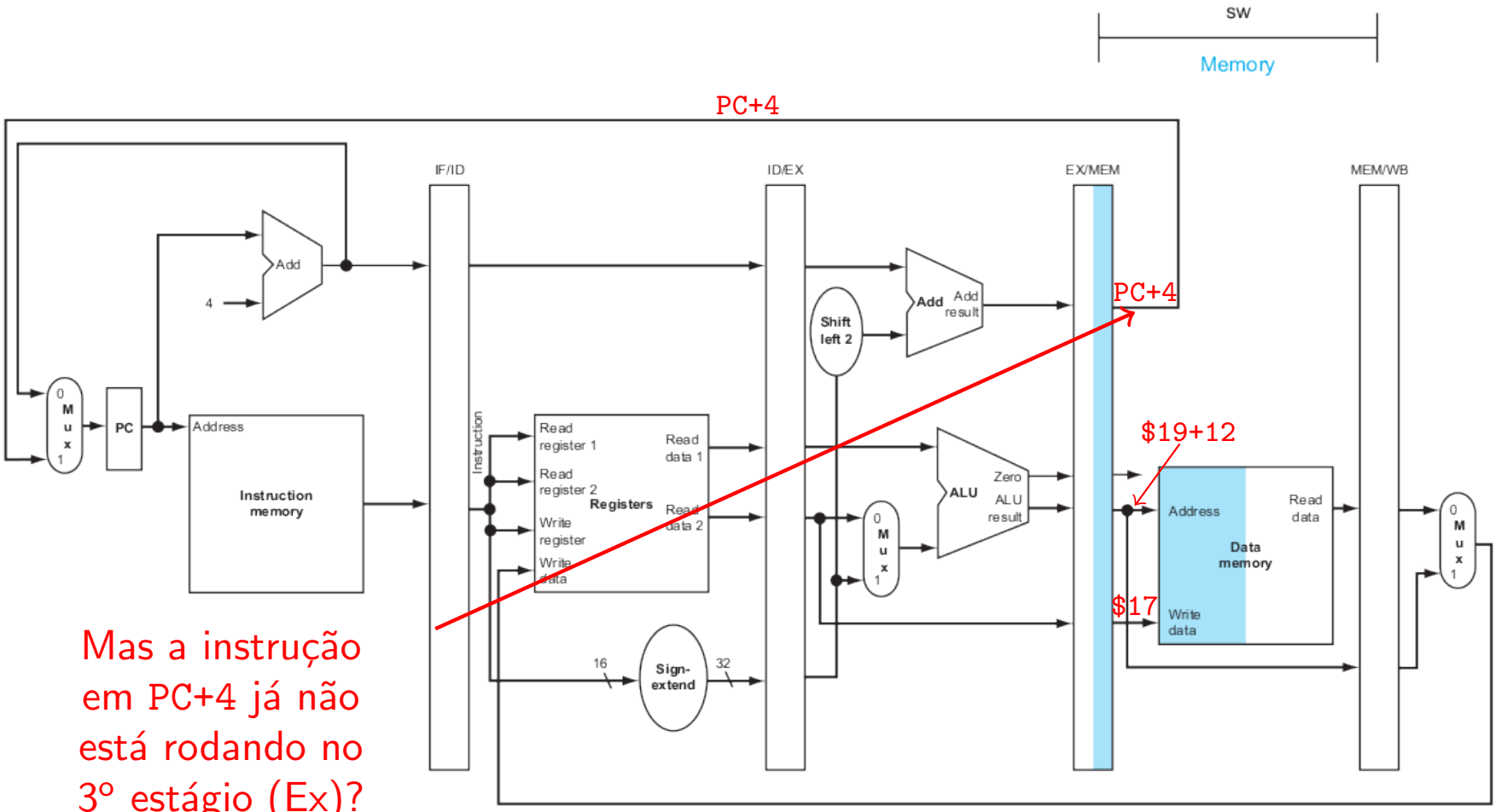
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

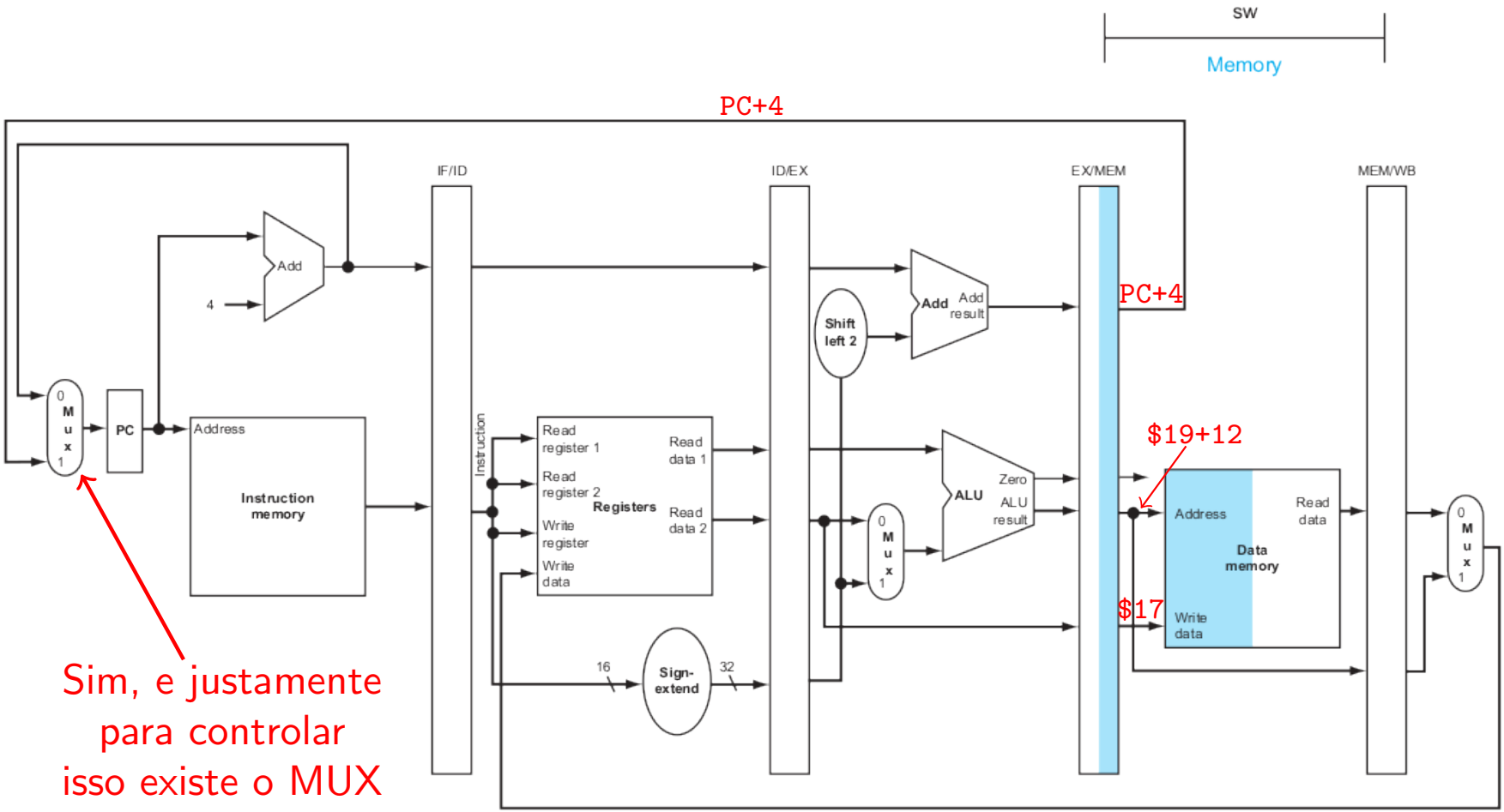


Mas a instrução em PC+4 já não está rodando no 3º estágio (Ex)?

Fonte: Adaptado de [1]

Pipeline

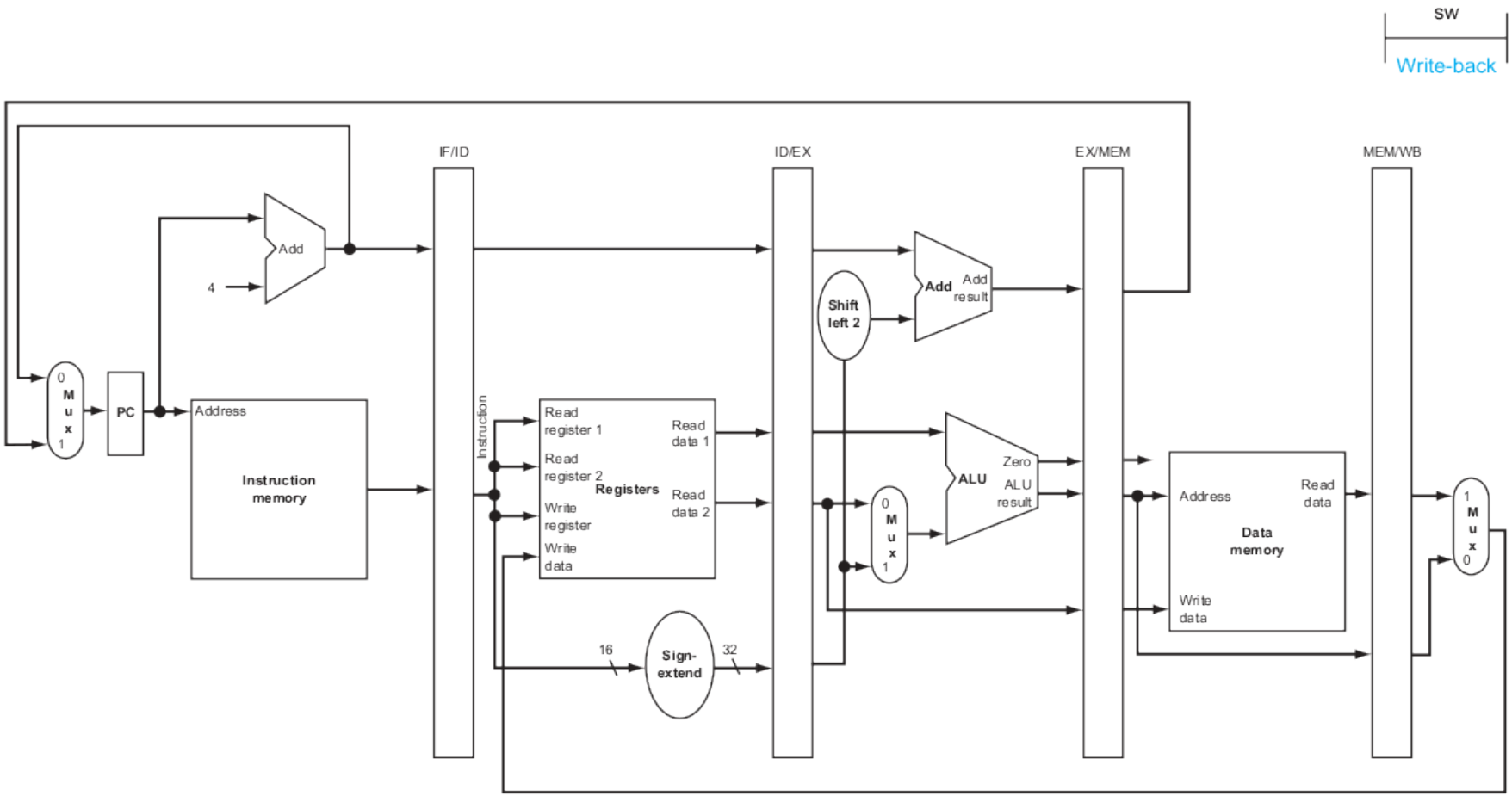
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

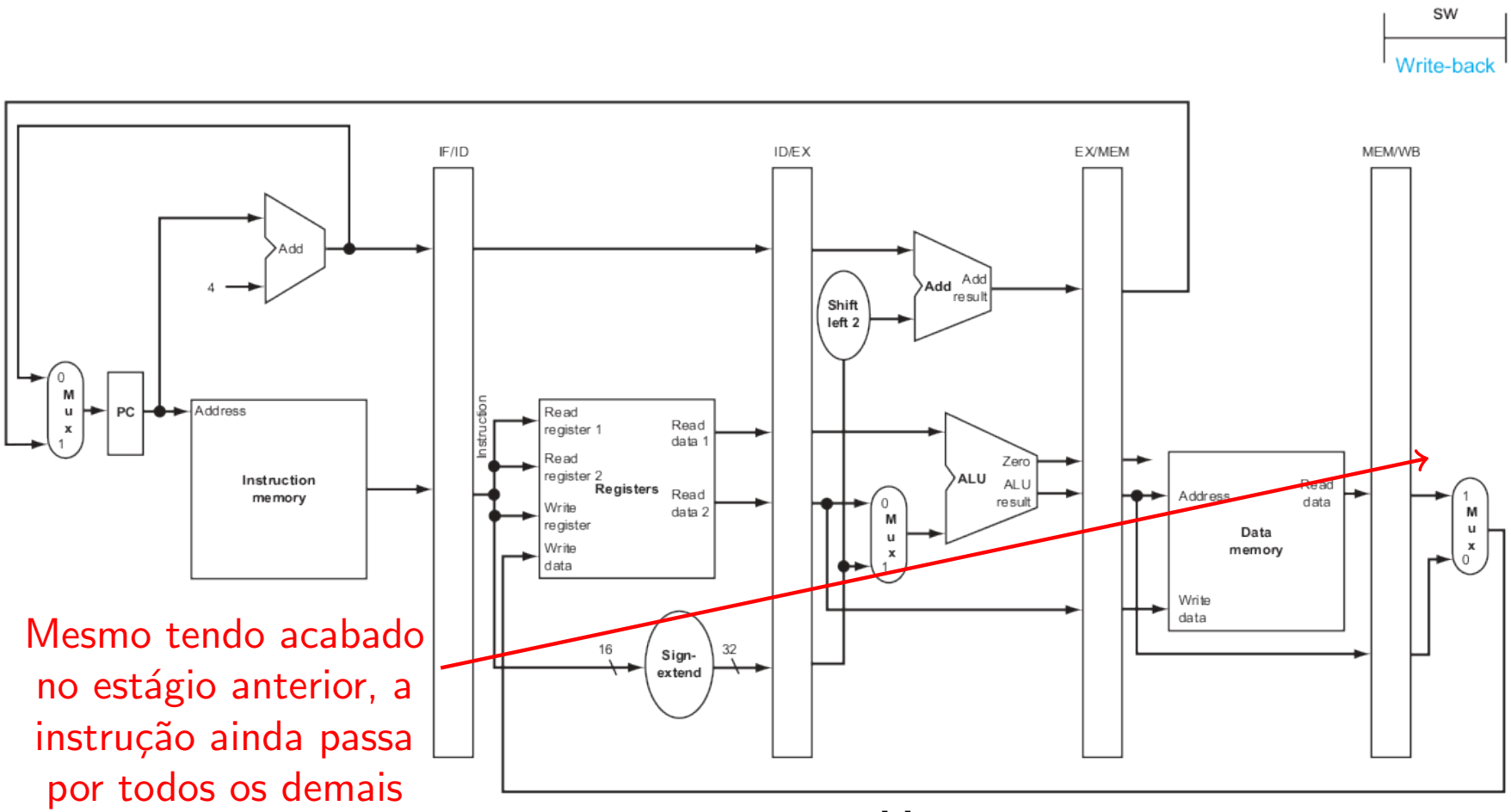
Exemplo 1: `sw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

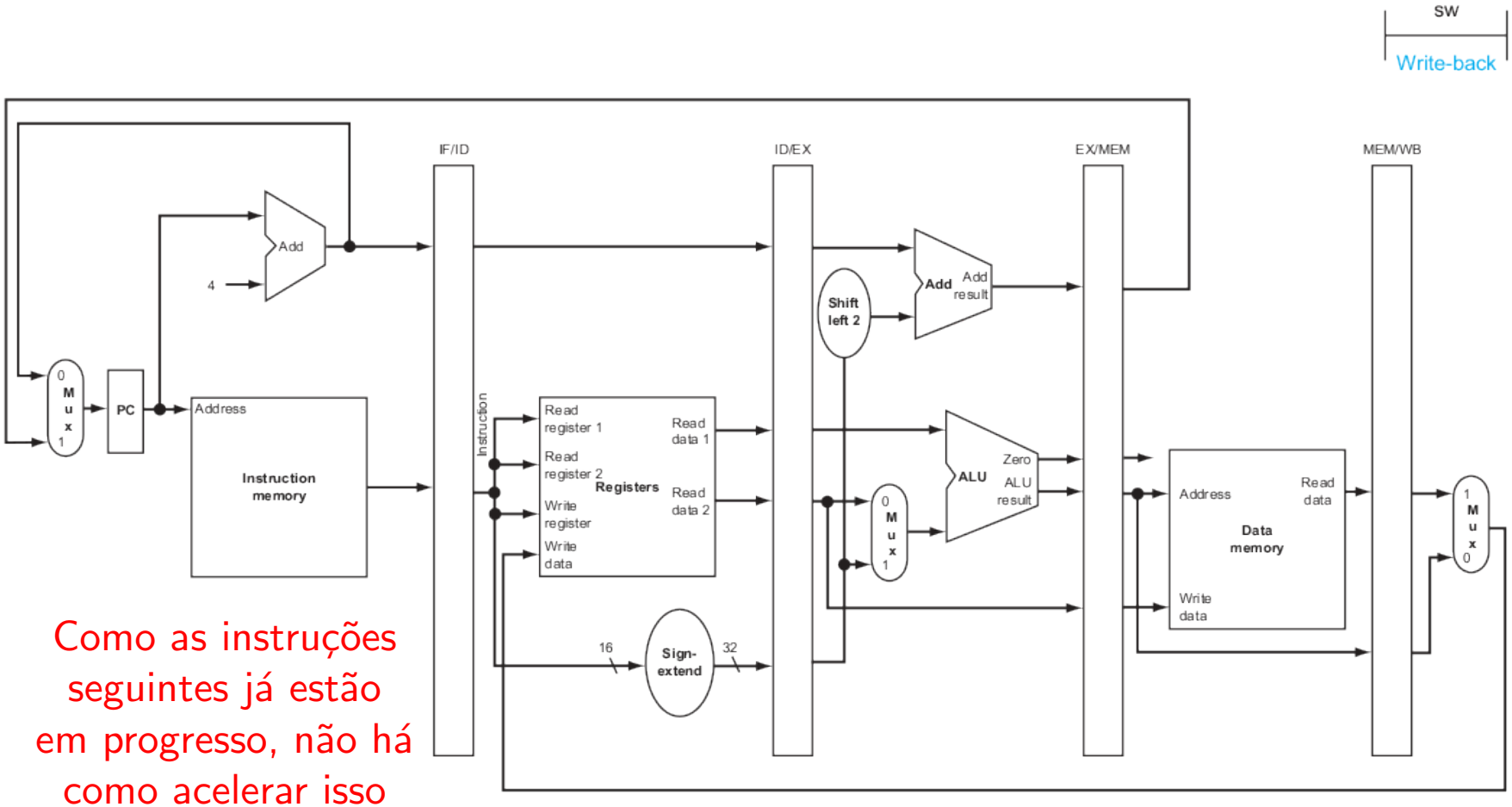


Mesmo tendo acabado no estágio anterior, a instrução ainda passa por todos os demais

Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

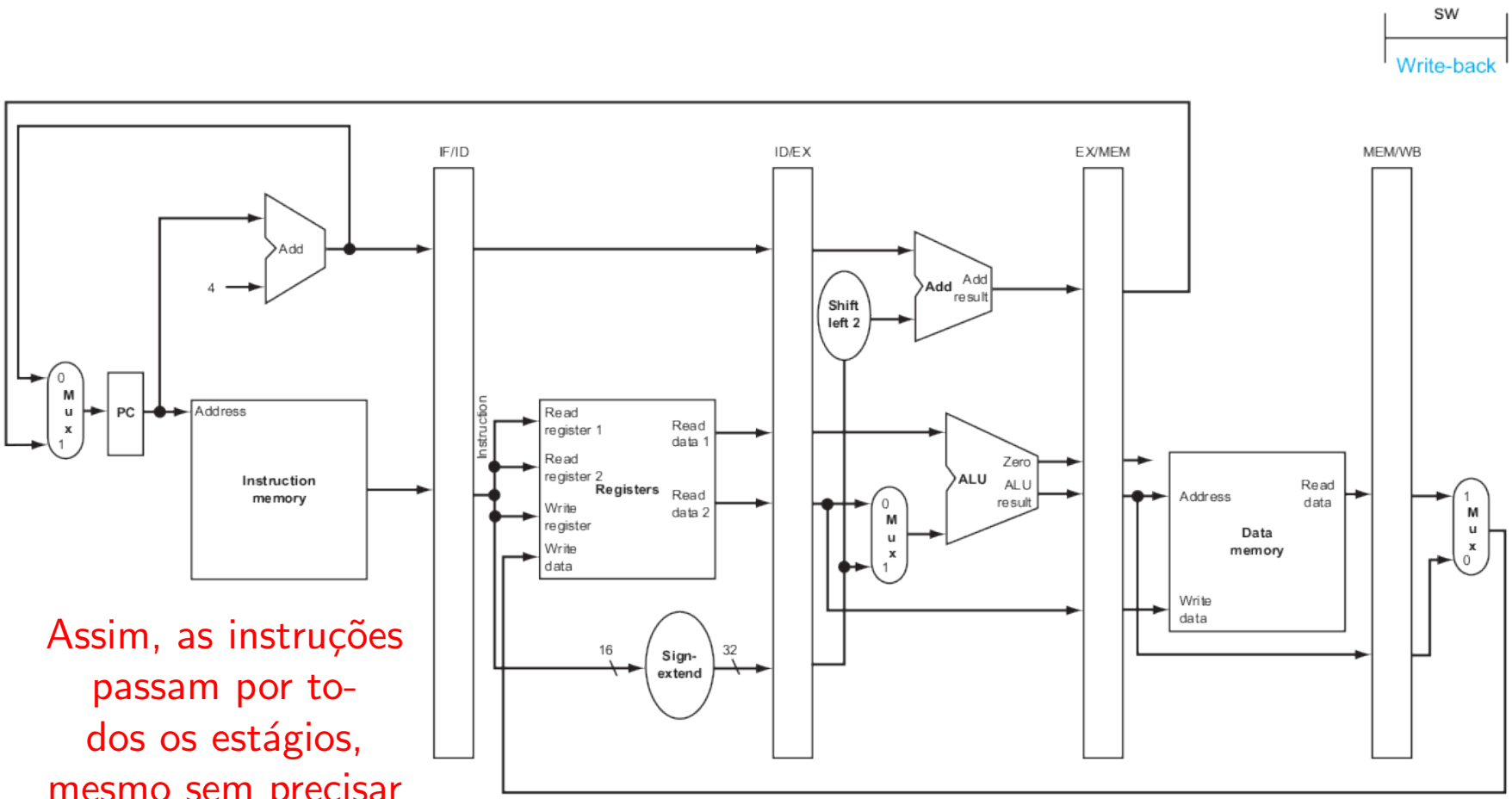


Como as instruções seguintes já estão em progresso, não há como acelerar isso

Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

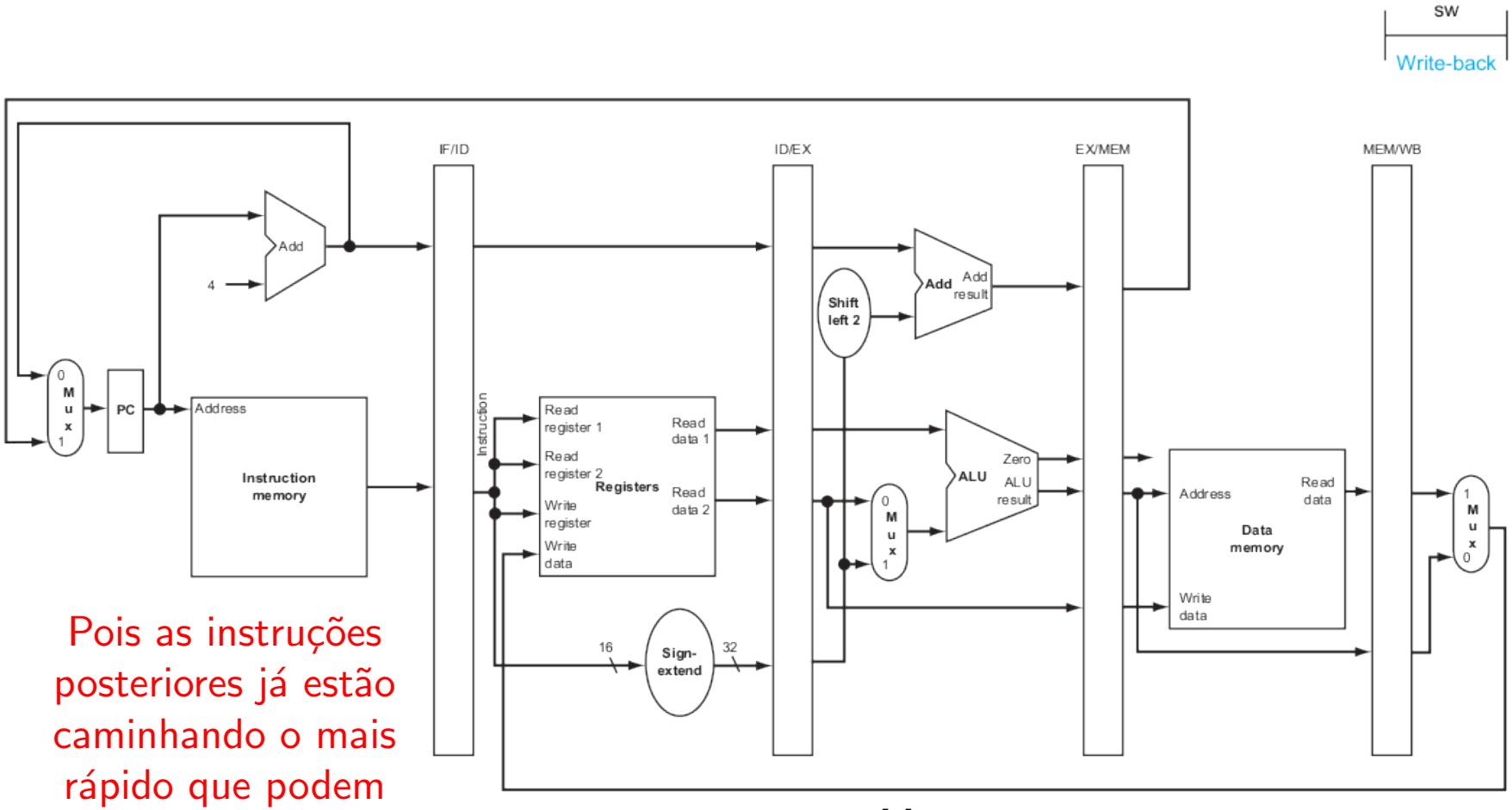


Assim, as instruções passam por todos os estágios, mesmo sem precisar

Fonte: Adaptado de [1]

Pipeline

Exemplo 1: `sw $17, 12($19)`

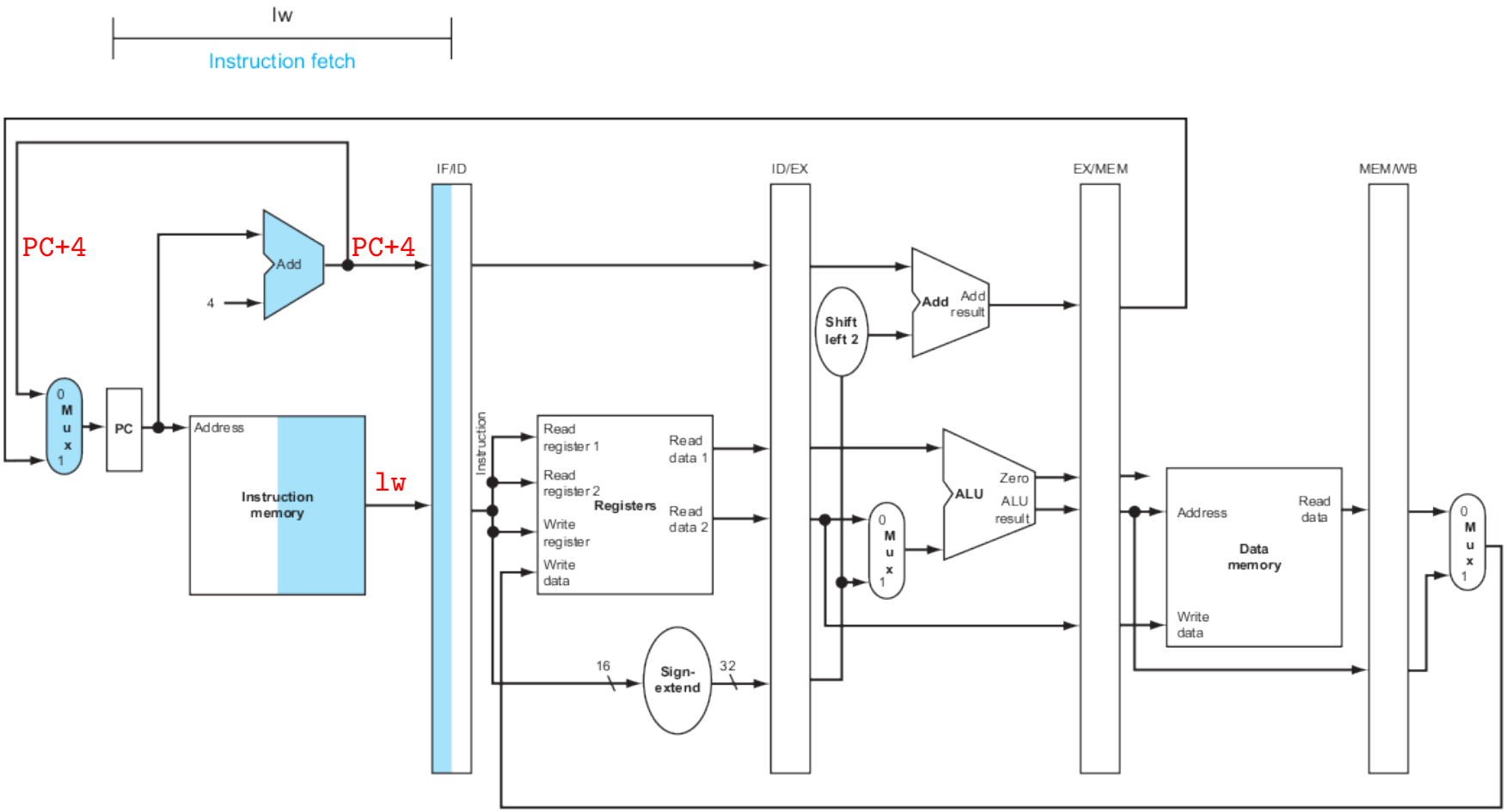


Pois as instruções posteriores já estão caminhando o mais rápido que podem

Fonte: Adaptado de [1]

Pipeline

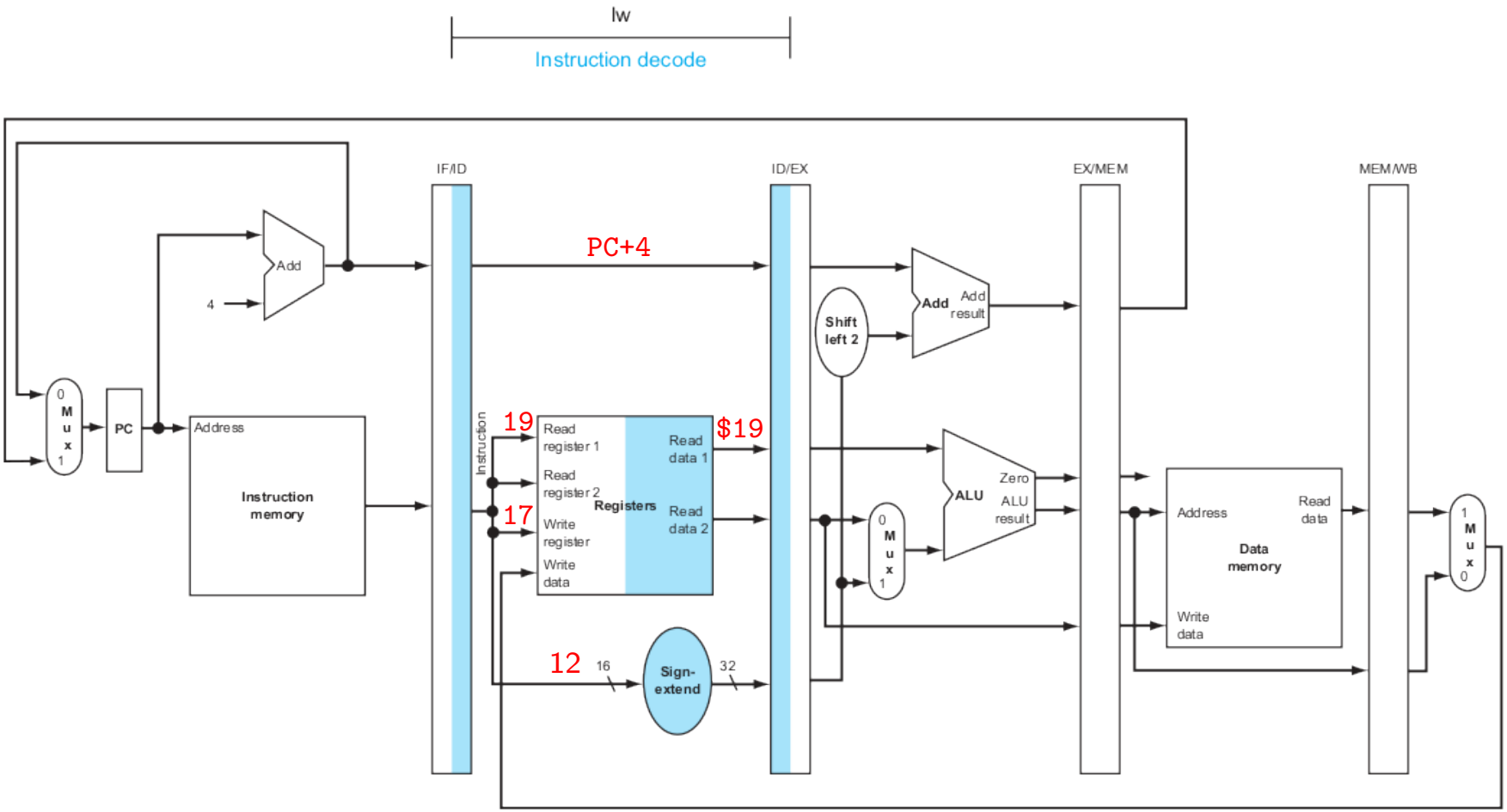
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

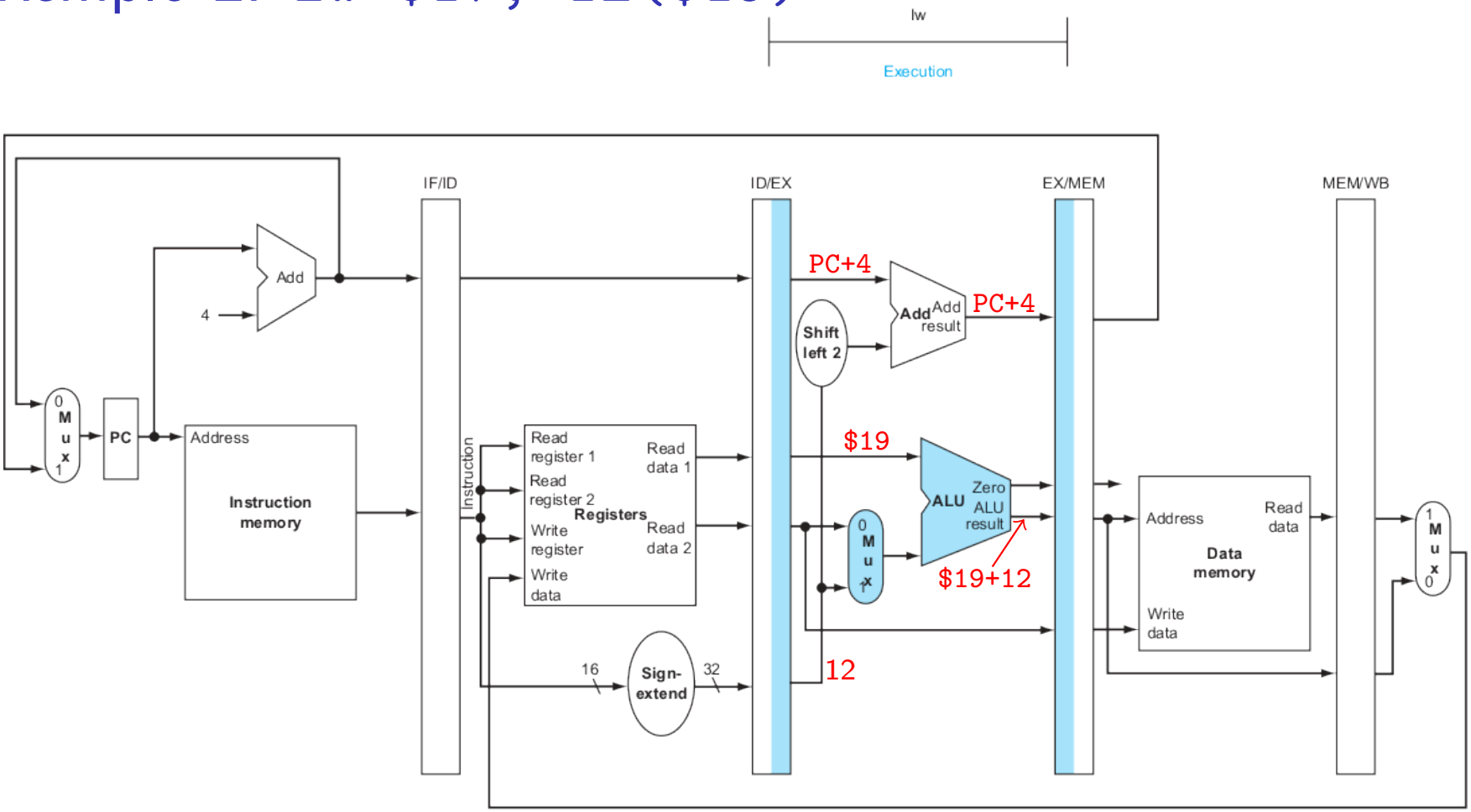
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

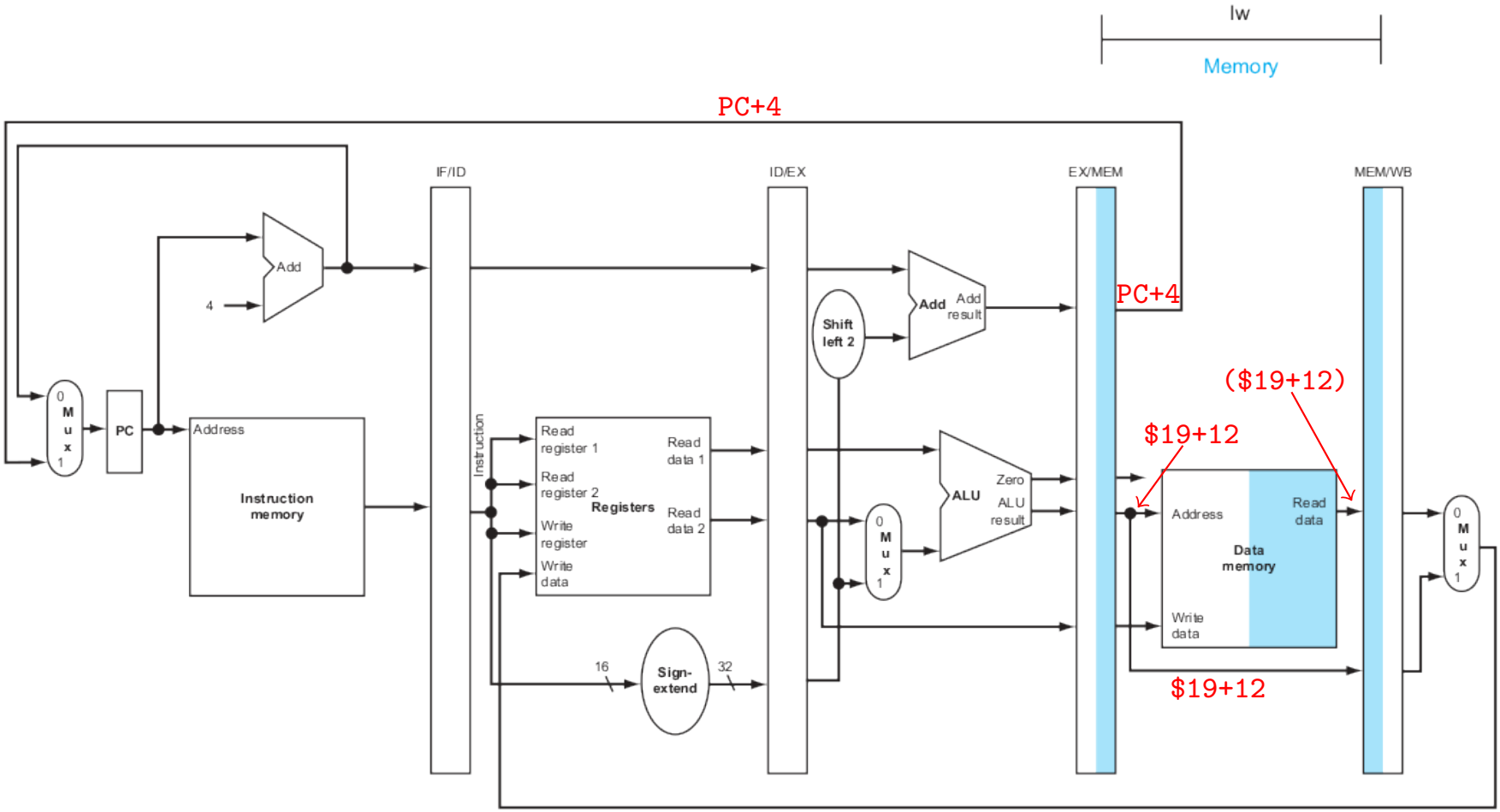
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

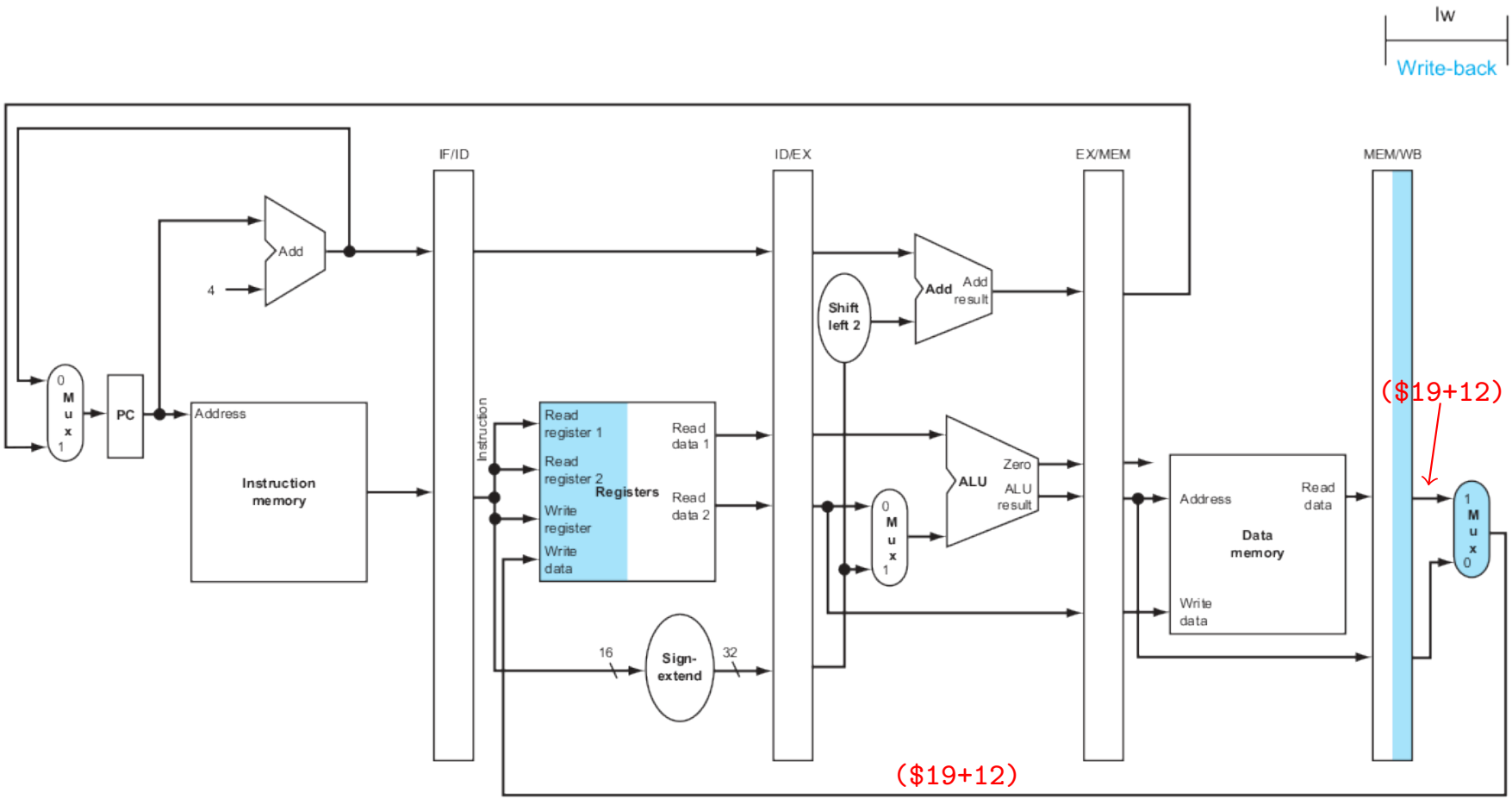
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

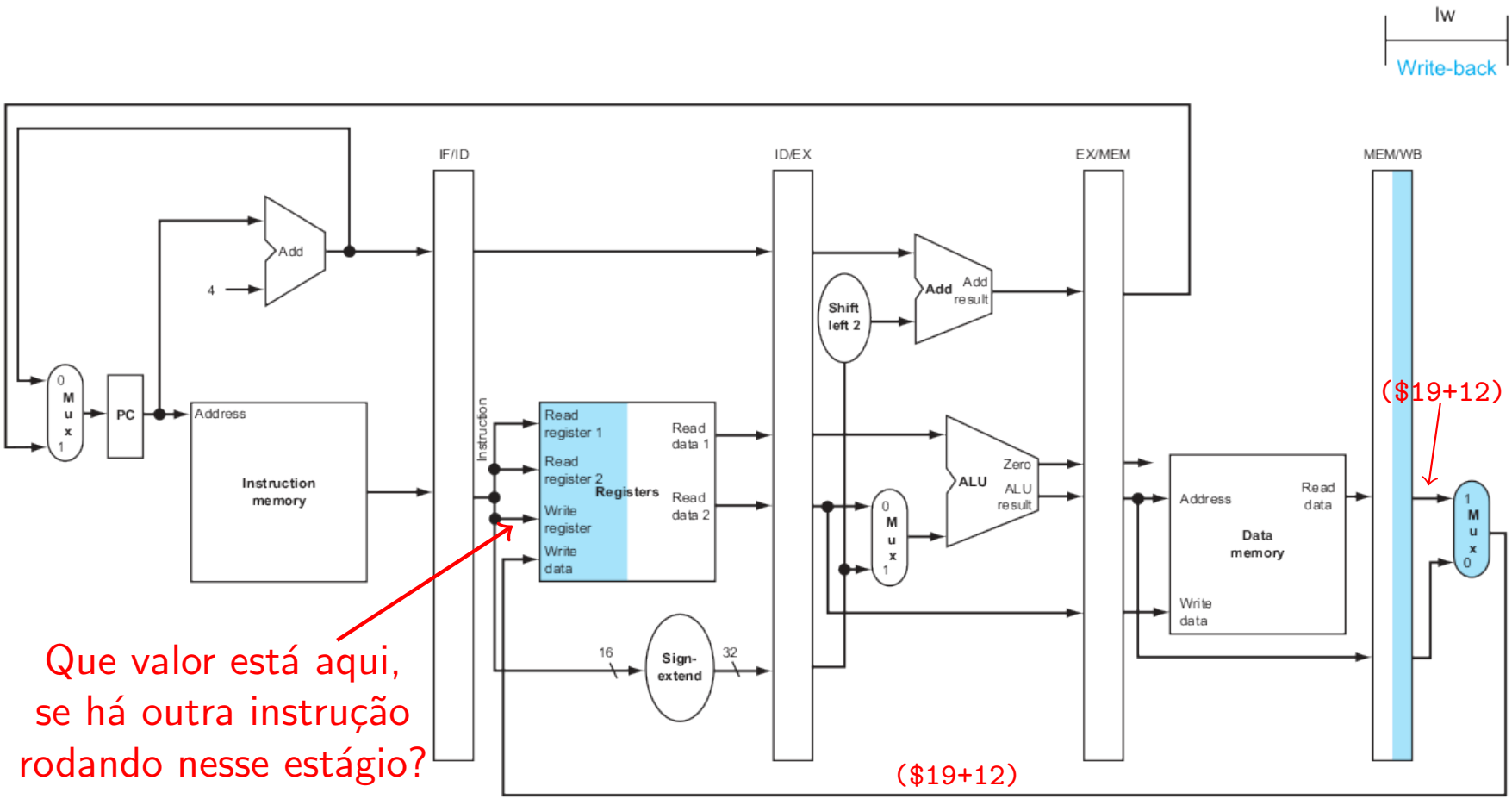
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

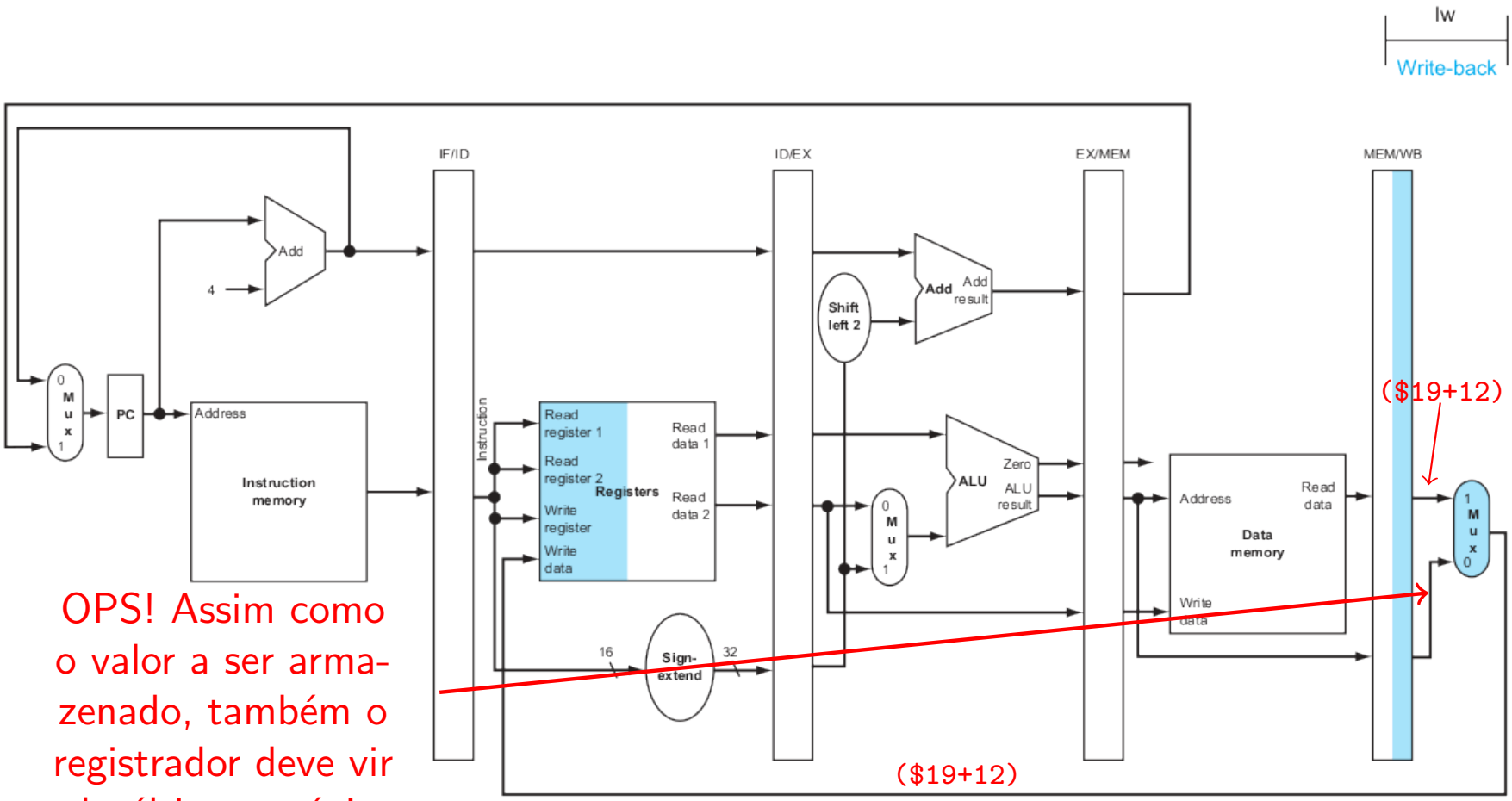
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

Exemplo 2: `lw $17, 12($19)`

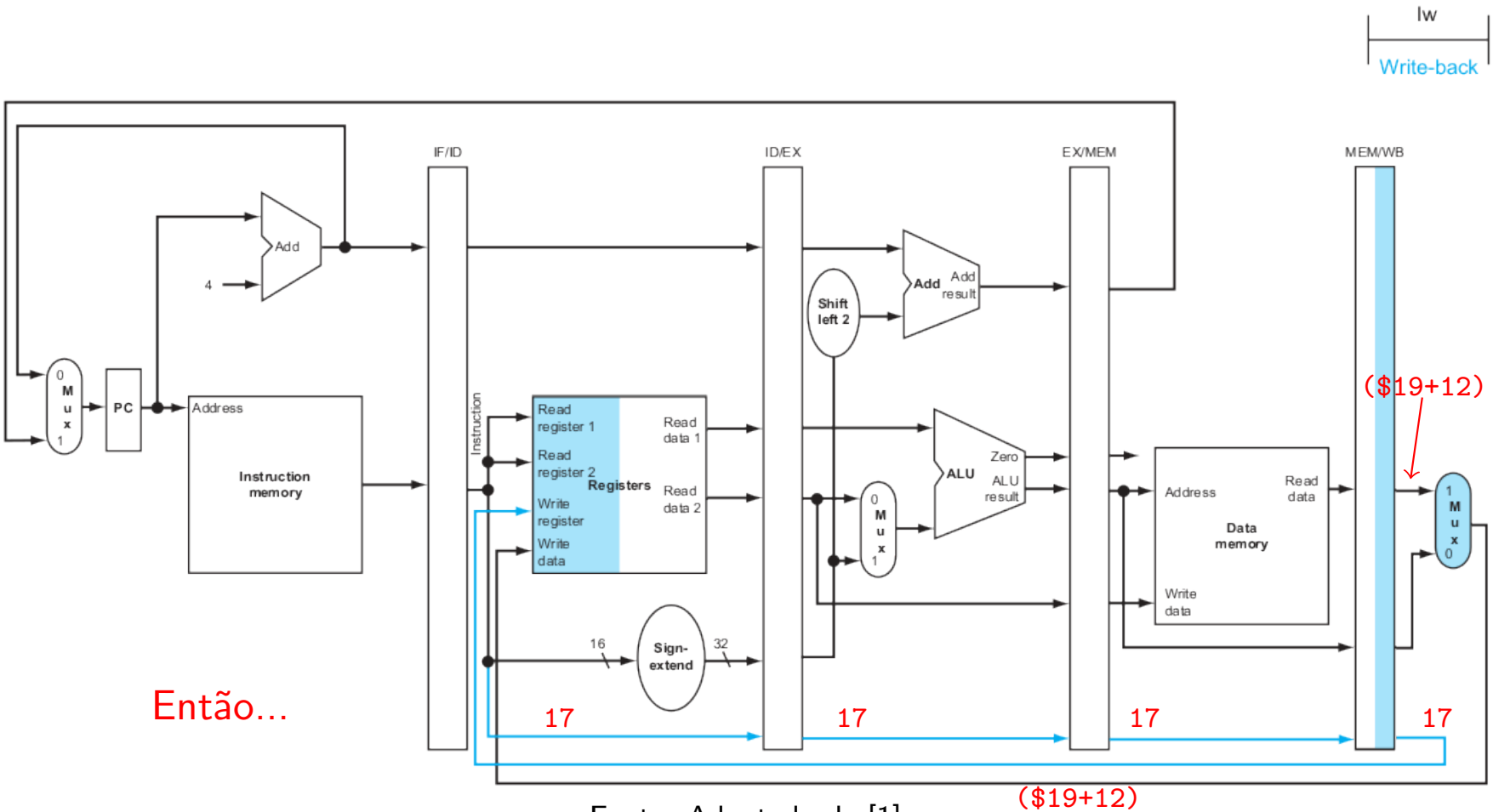


OPS! Assim como o valor a ser armazenado, também o registrador deve vir do último estágio

Fonte: Adaptado de [1]

Pipeline

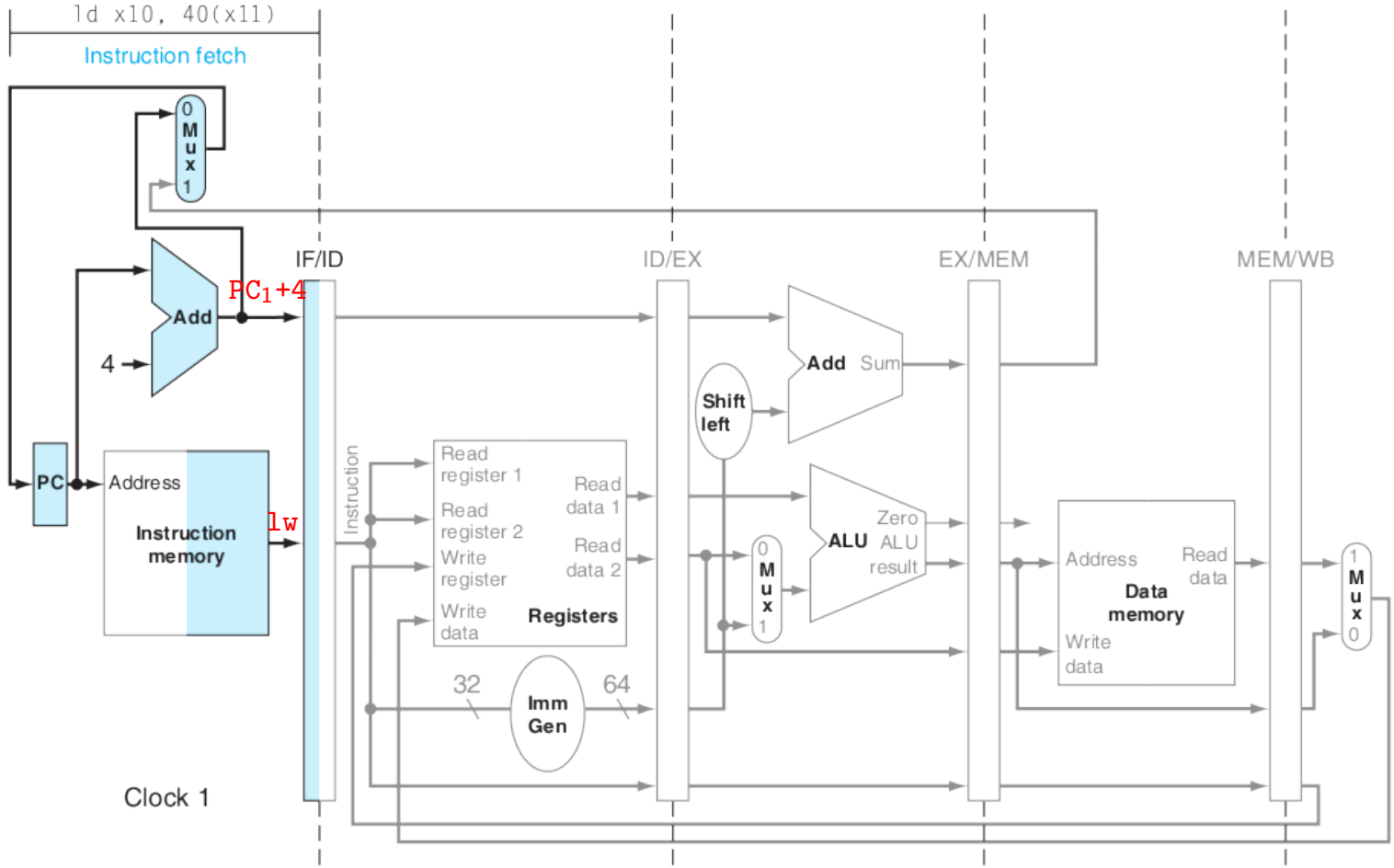
Exemplo 2: `lw $17, 12($19)`



Fonte: Adaptado de [1]

Pipeline

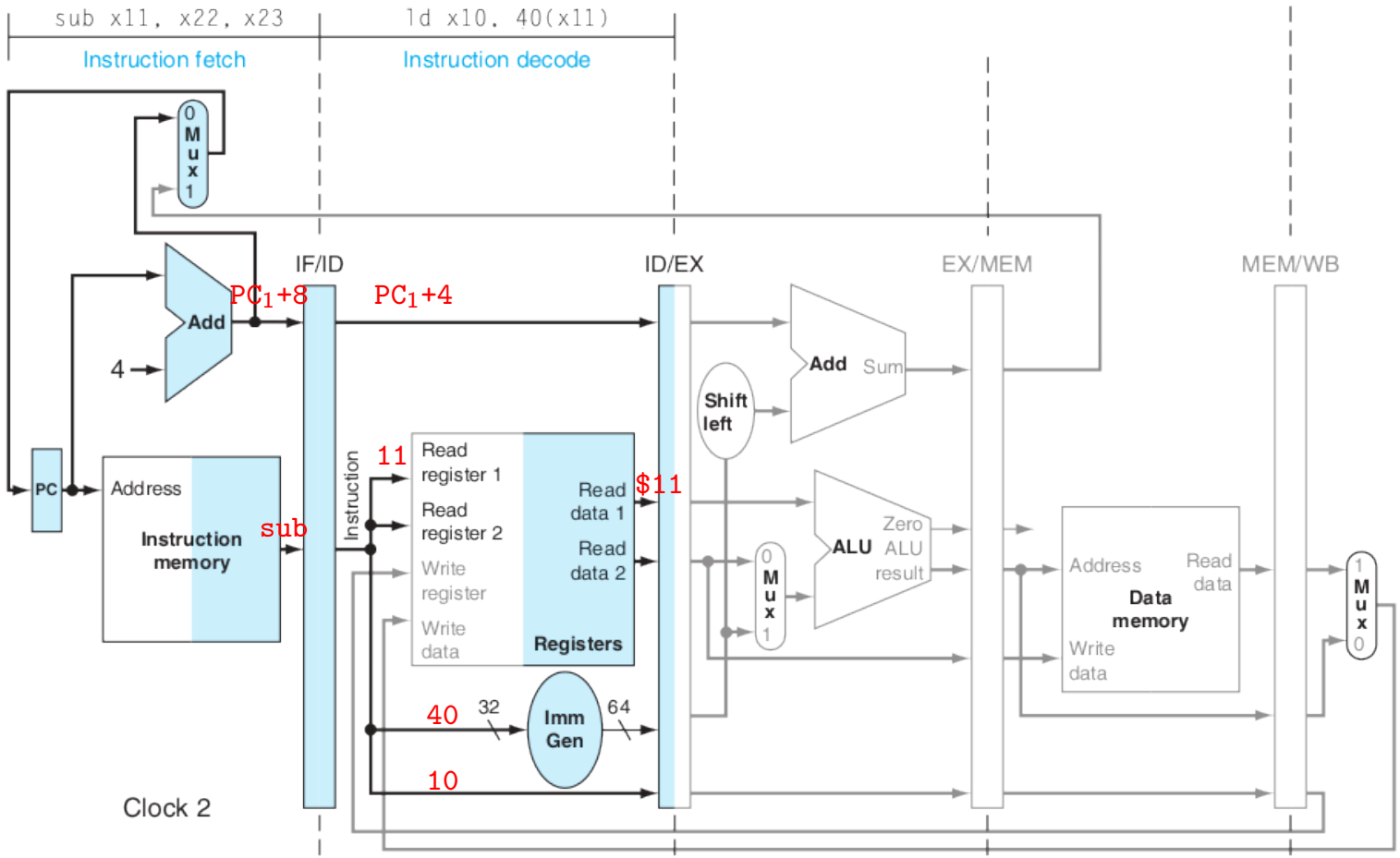
Exemplo 3: `lw $10,40($11); sub $11,$22,$23`



Fonte: Adaptado de [1]

Pipeline

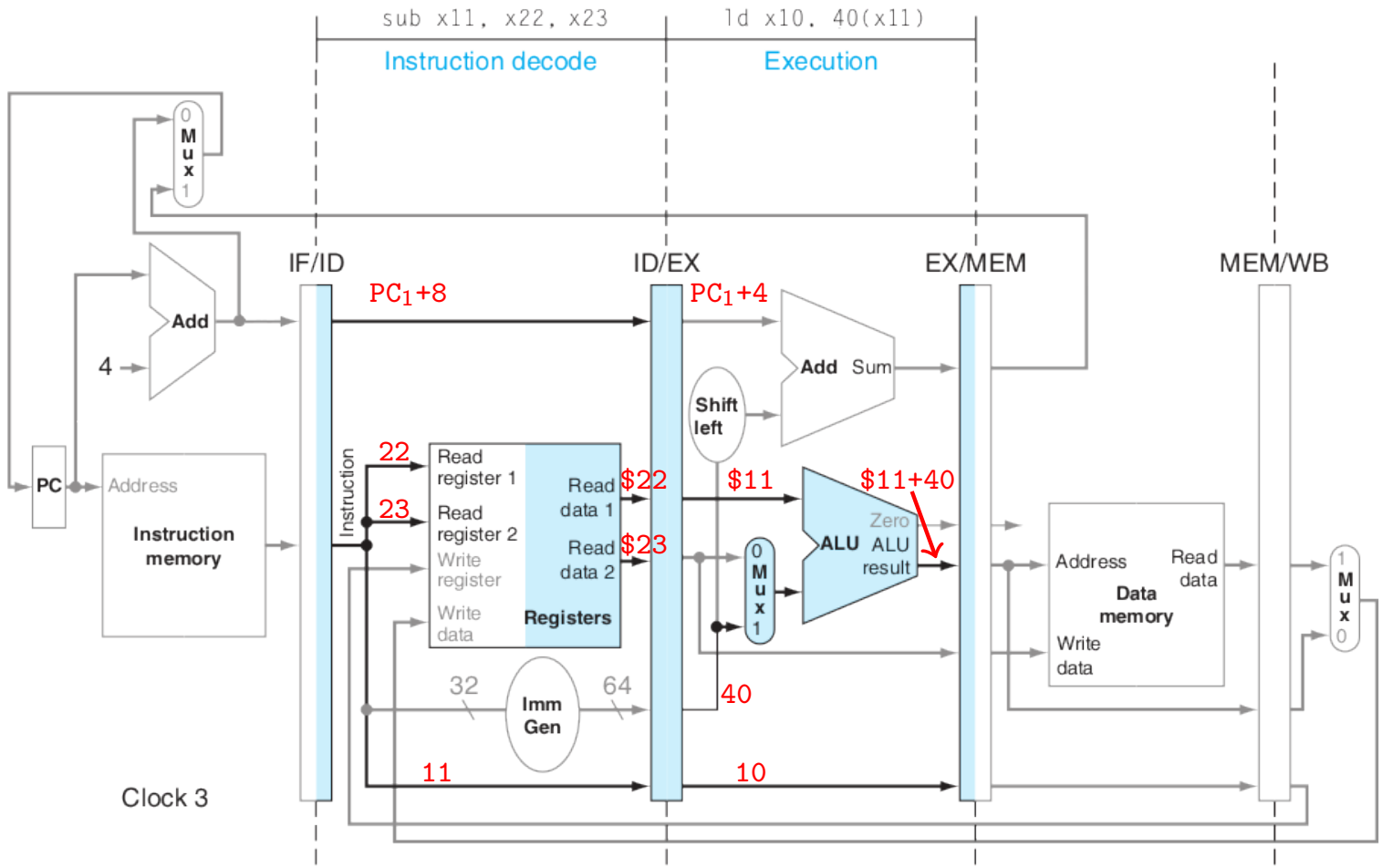
Exemplo 3: `lw $10,40($11); sub $11,$22,$23`



Fonte: Adaptado de [1]

Pipeline

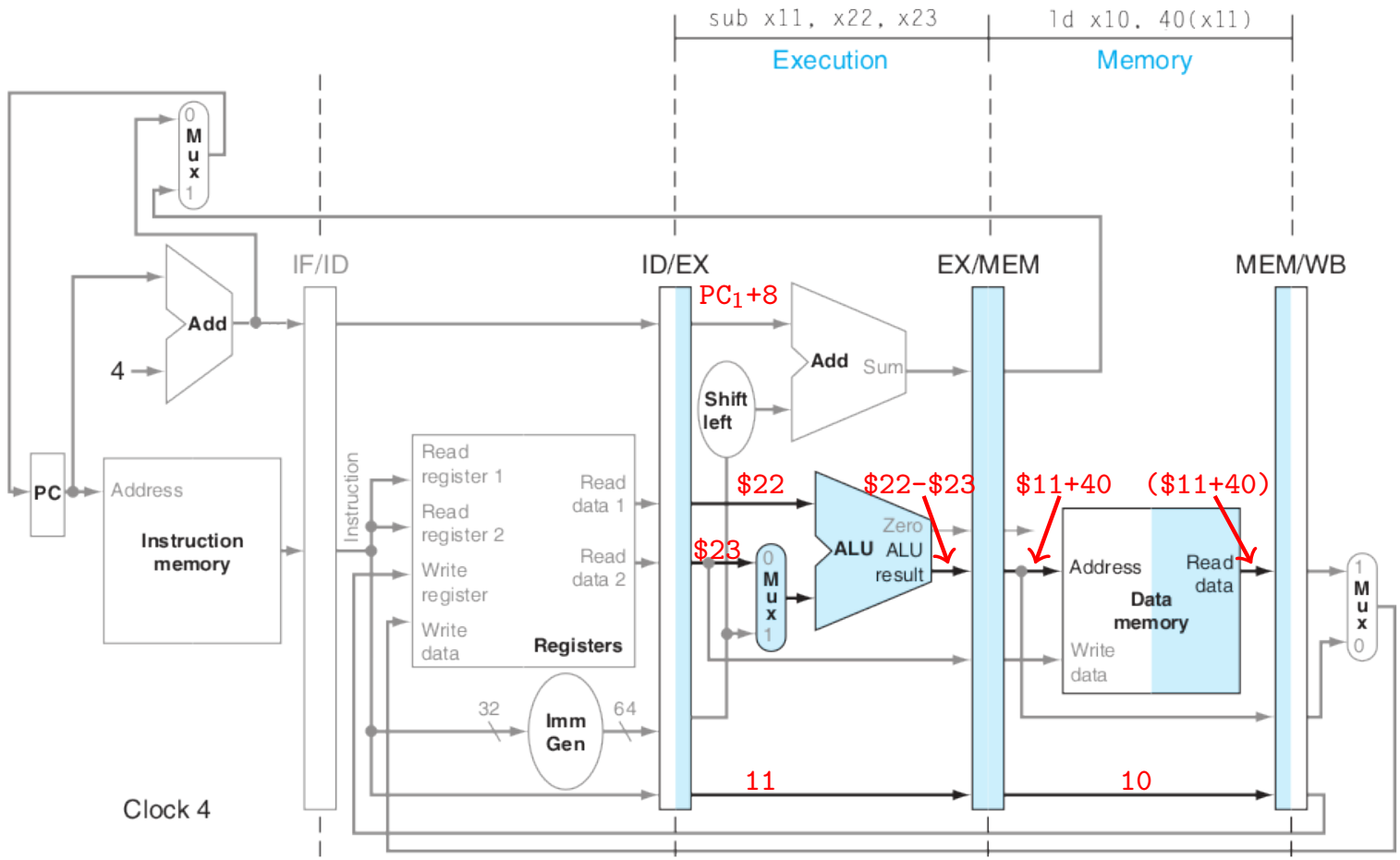
Exemplo 3: `lw $10,40($11); sub $11,$22,$23`



Fonte: Adaptado de [1]

Pipeline

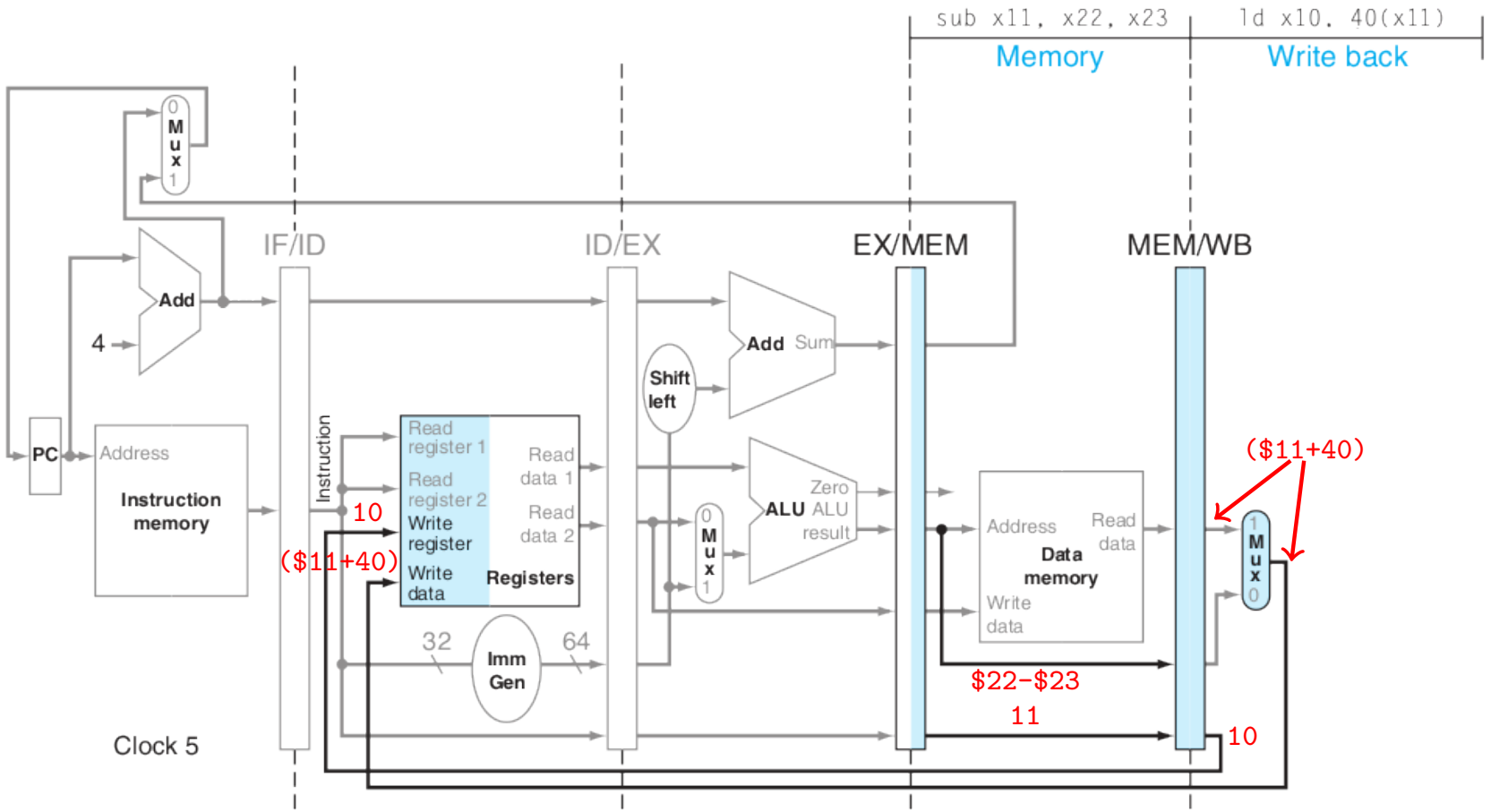
Exemplo 3: `lw $10,40($11); sub $11,$22,$23`



Fonte: Adaptado de [1]

Pipeline

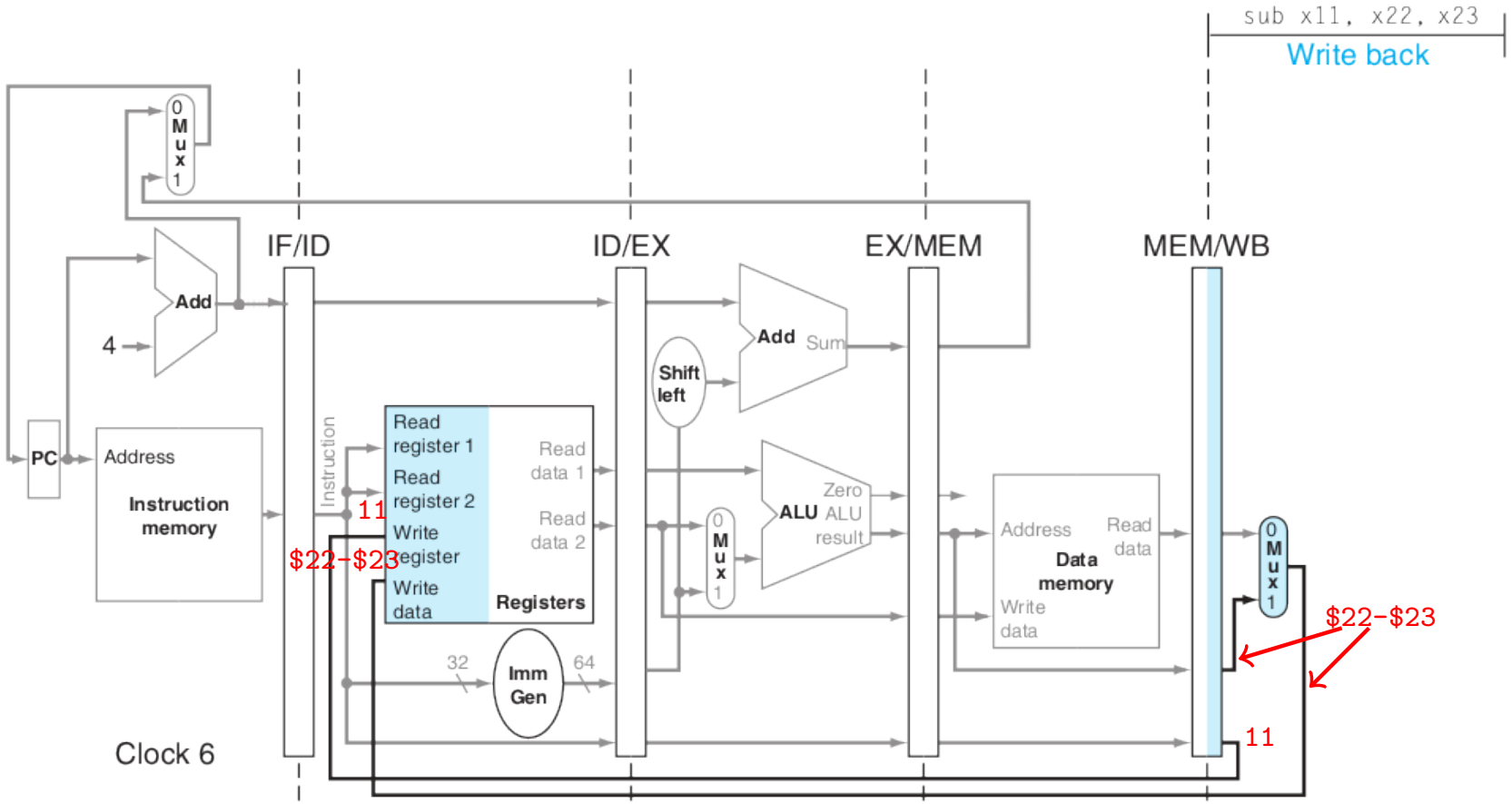
Exemplo 3: `lw $10,40($11); sub $11,$22,$23`



Fonte: Adaptado de [1]

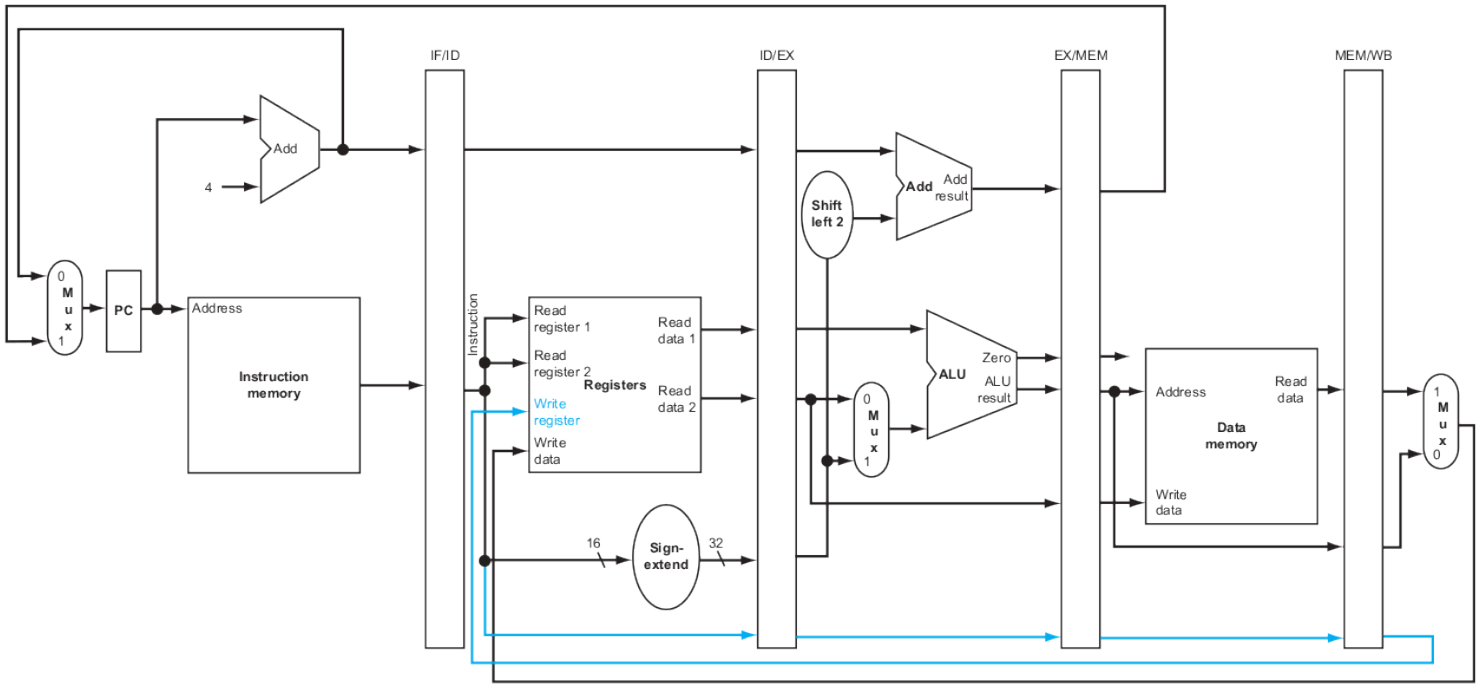
Pipeline

Exemplo 3: `lw $10,40($11); sub $11,$22,$23`



Fonte: Adaptado de [1]

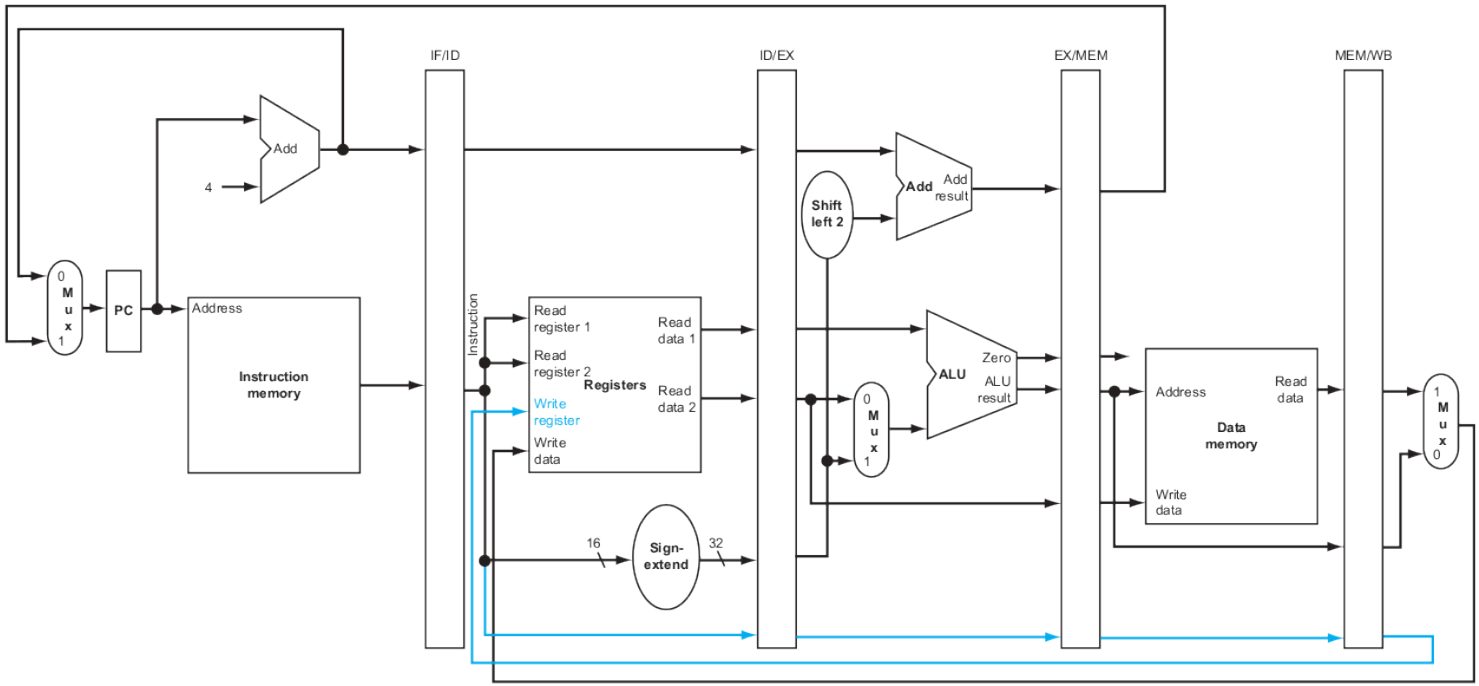
Controlando a *pipeline*



Fonte: [1]

- Agora que temos a *pipeline*, que mais falta?

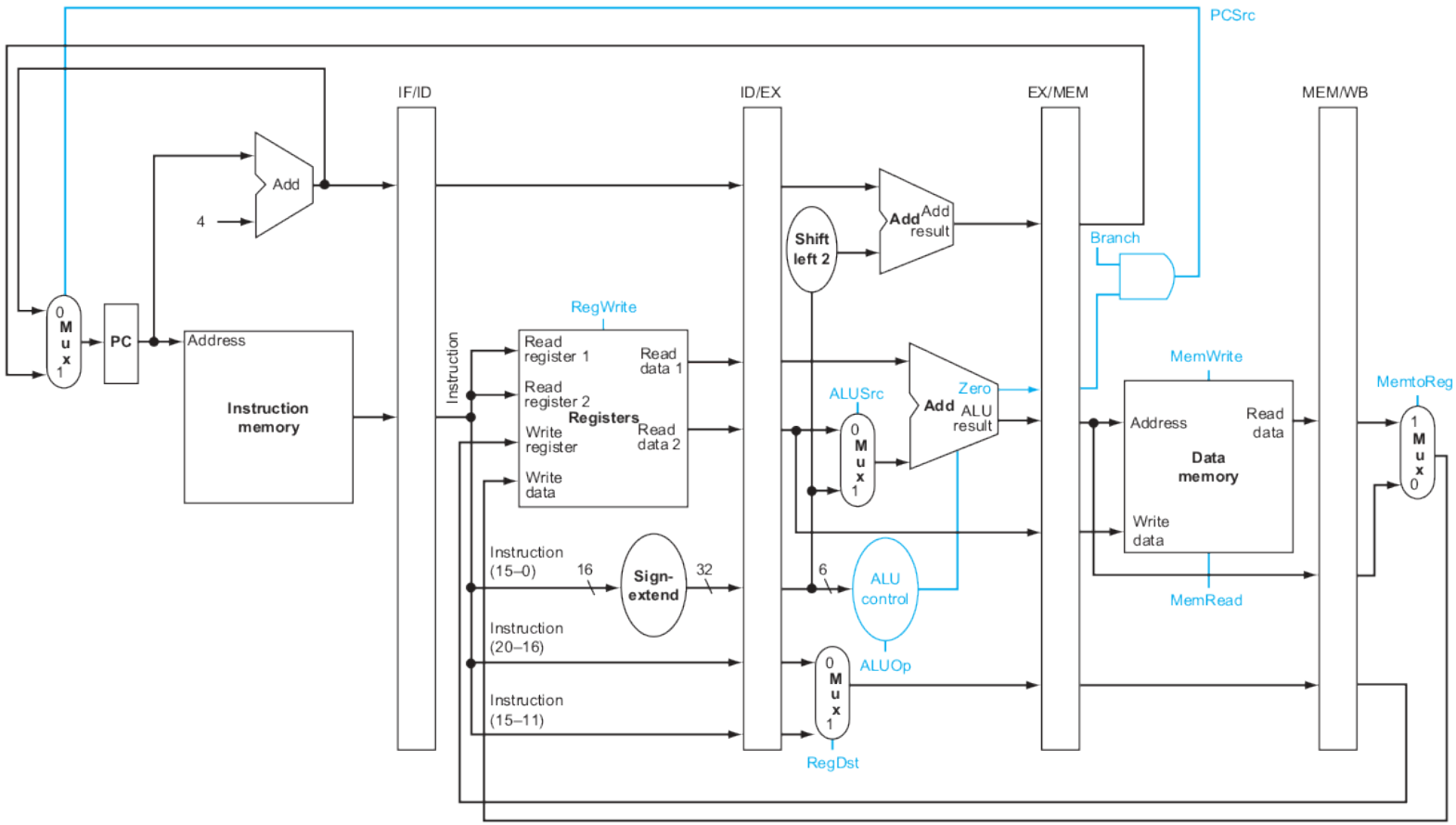
Controlando a *pipeline*



Fonte: [1]

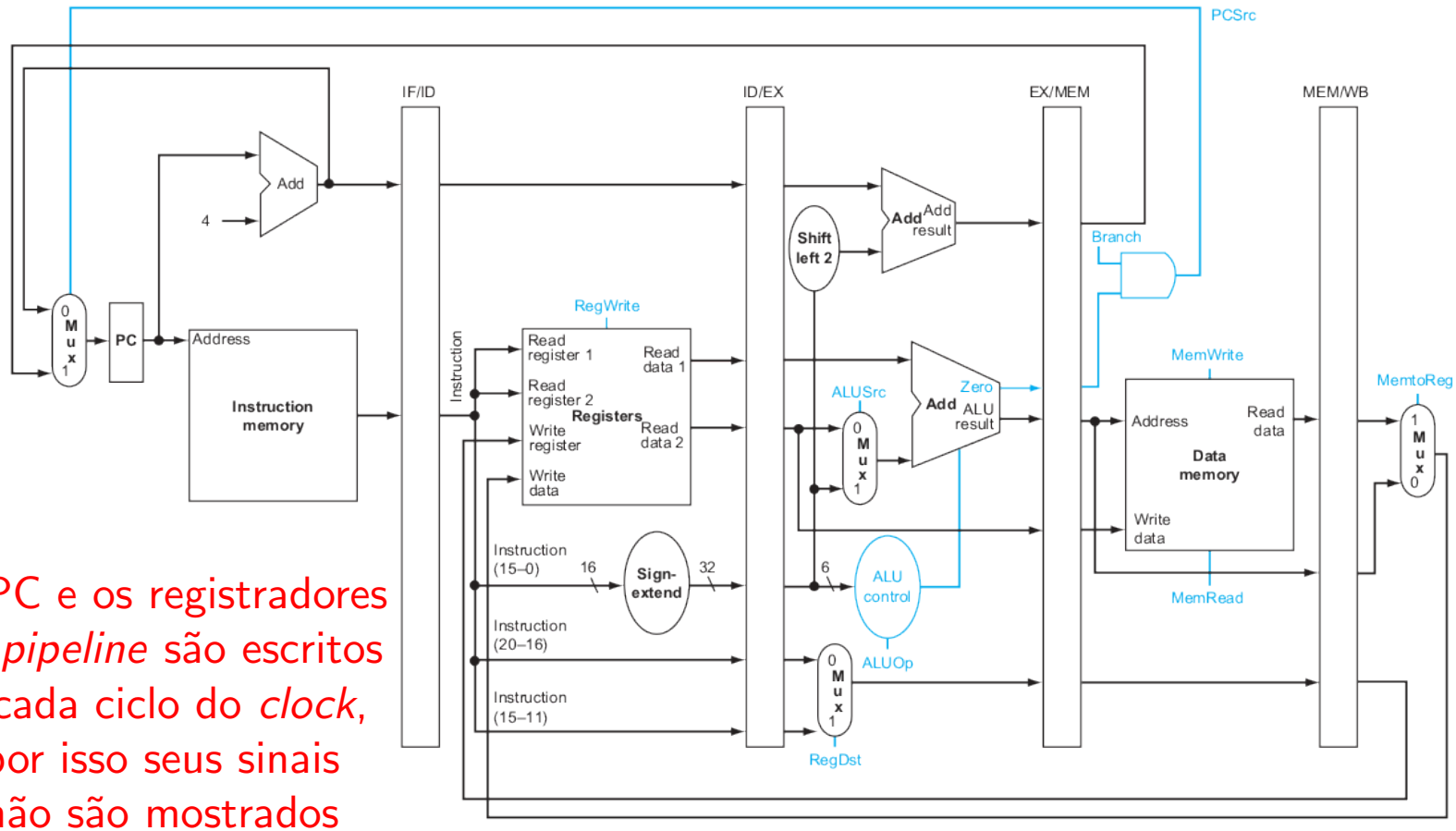
- Agora que temos a *pipeline*, que mais falta?
  - Adicionarmos os sinais de controle

Controlando a *pipeline*



Fonte: [1]

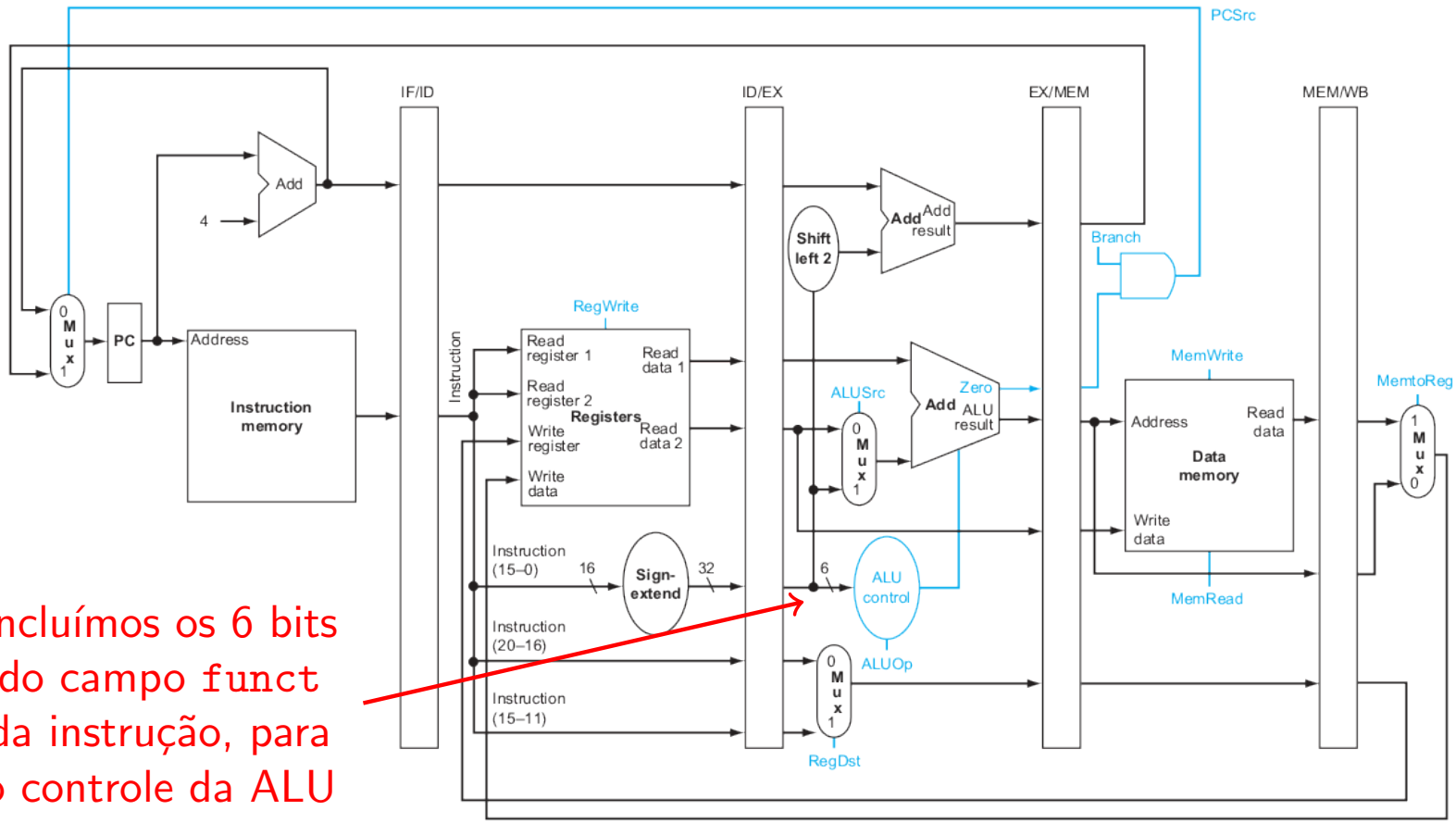
Controlando a *pipeline*



O PC e os registradores da *pipeline* são escritos a cada ciclo do *clock*, por isso seus sinais não são mostrados

Fonte: [1]

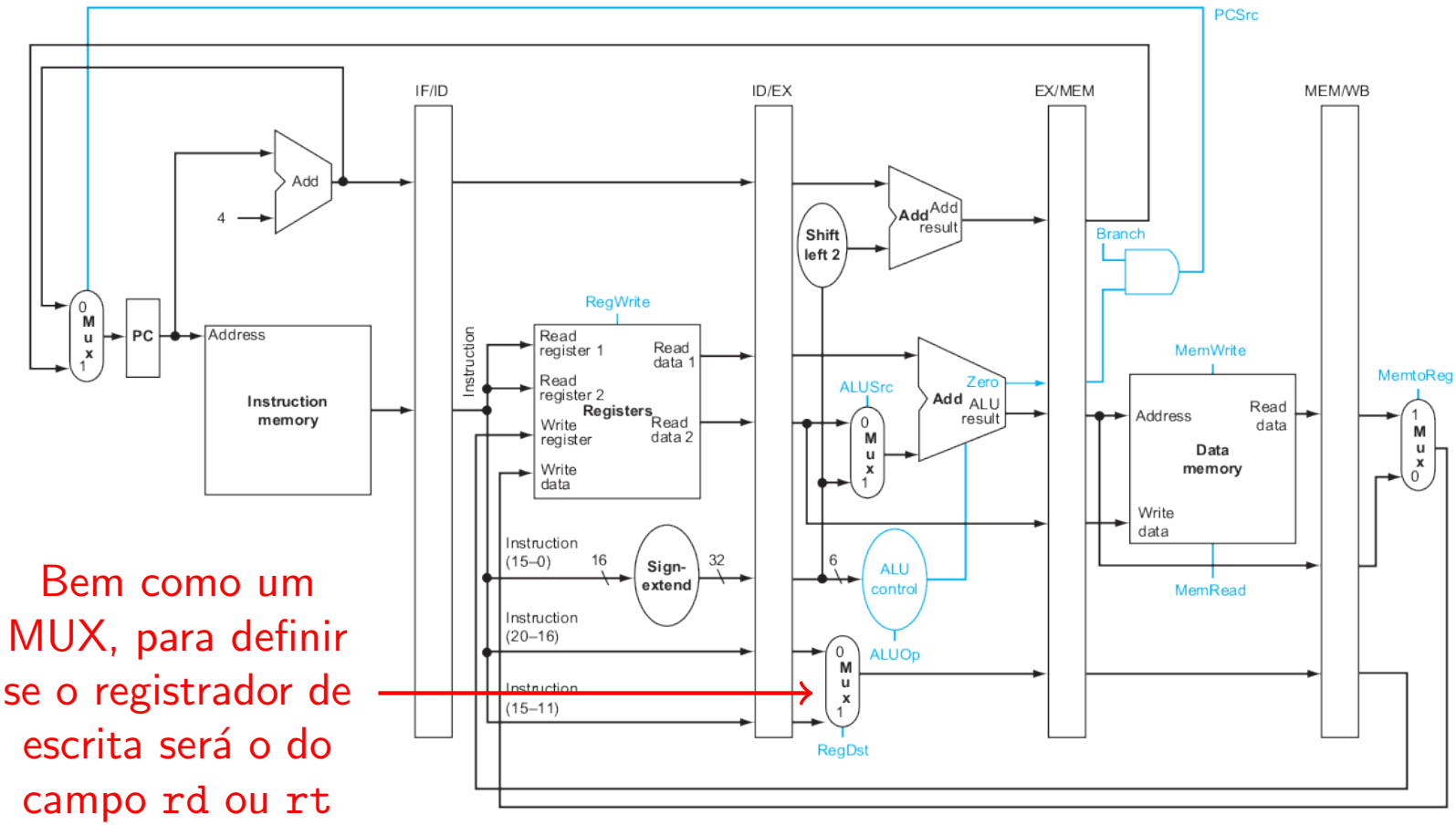
Controlando a *pipeline*



Incluímos os 6 bits do campo `funct` da instrução, para o controle da ALU

Fonte: [1]

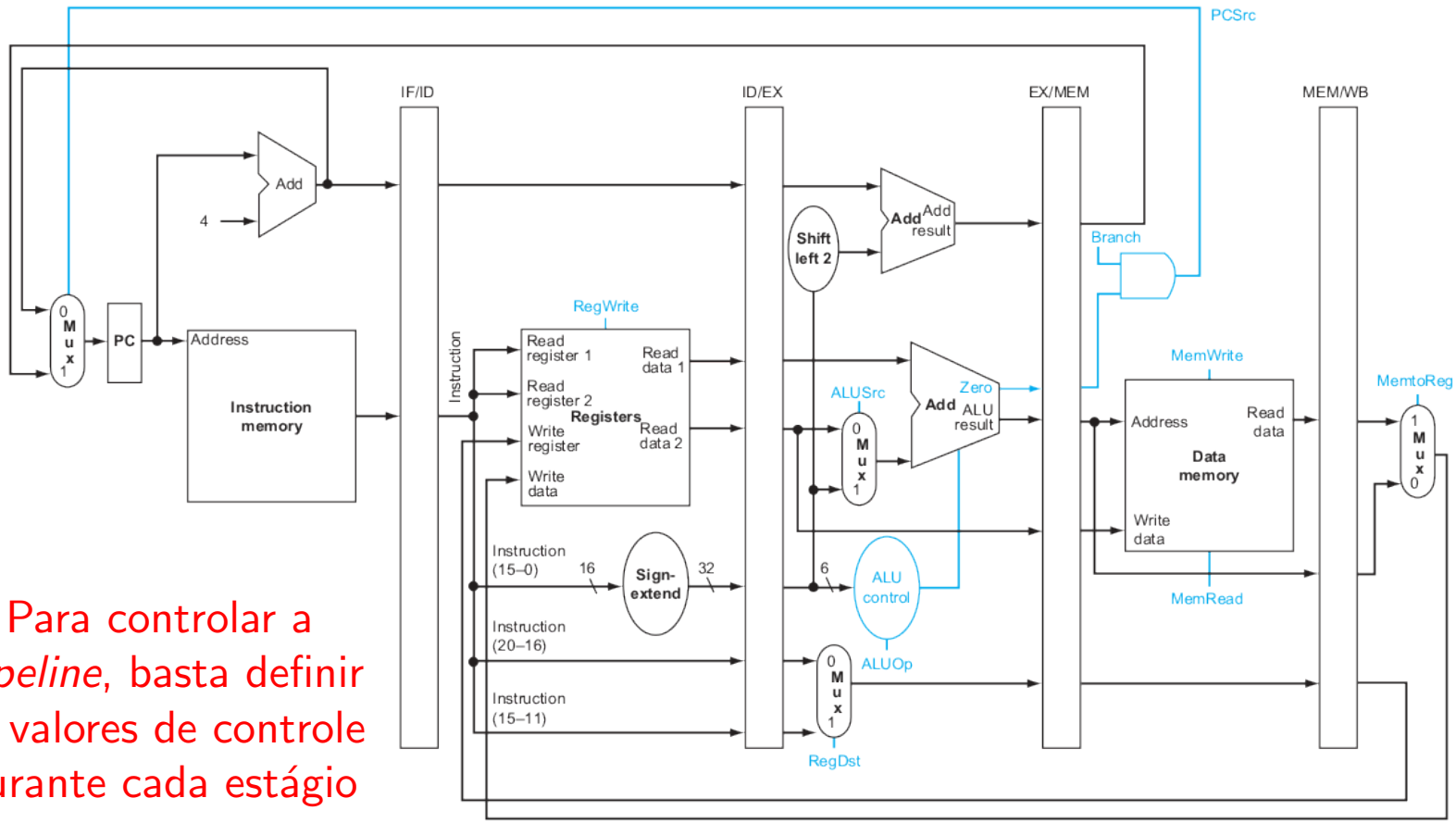
Controlando a *pipeline*



Bem como um MUX, para definir se o registrador de escrita será o do campo `rd` ou `rt`

Fonte: [1]

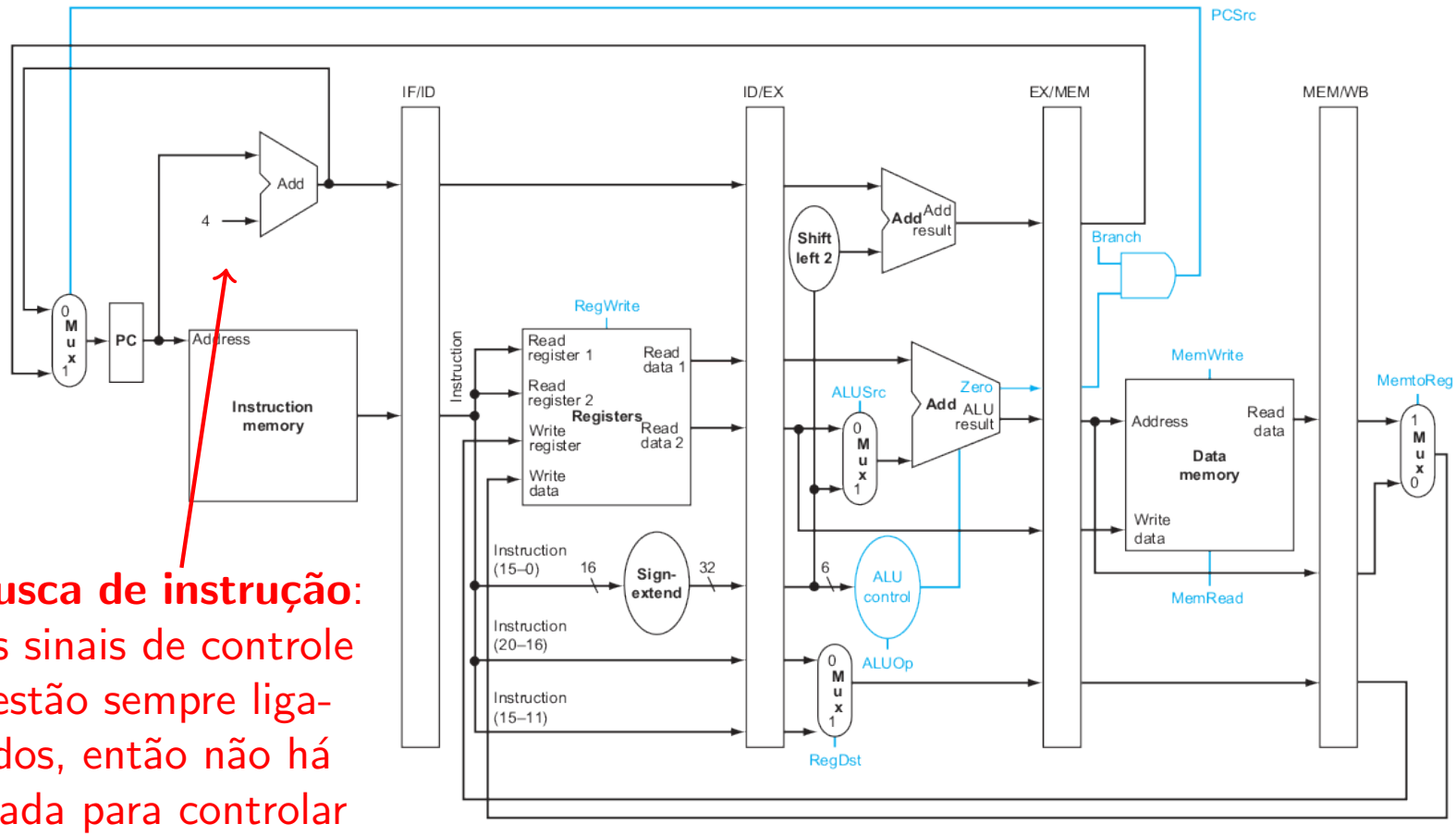
Controlando a *pipeline*



Para controlar a *pipeline*, basta definir os valores de controle durante cada estágio

Fonte: [1]

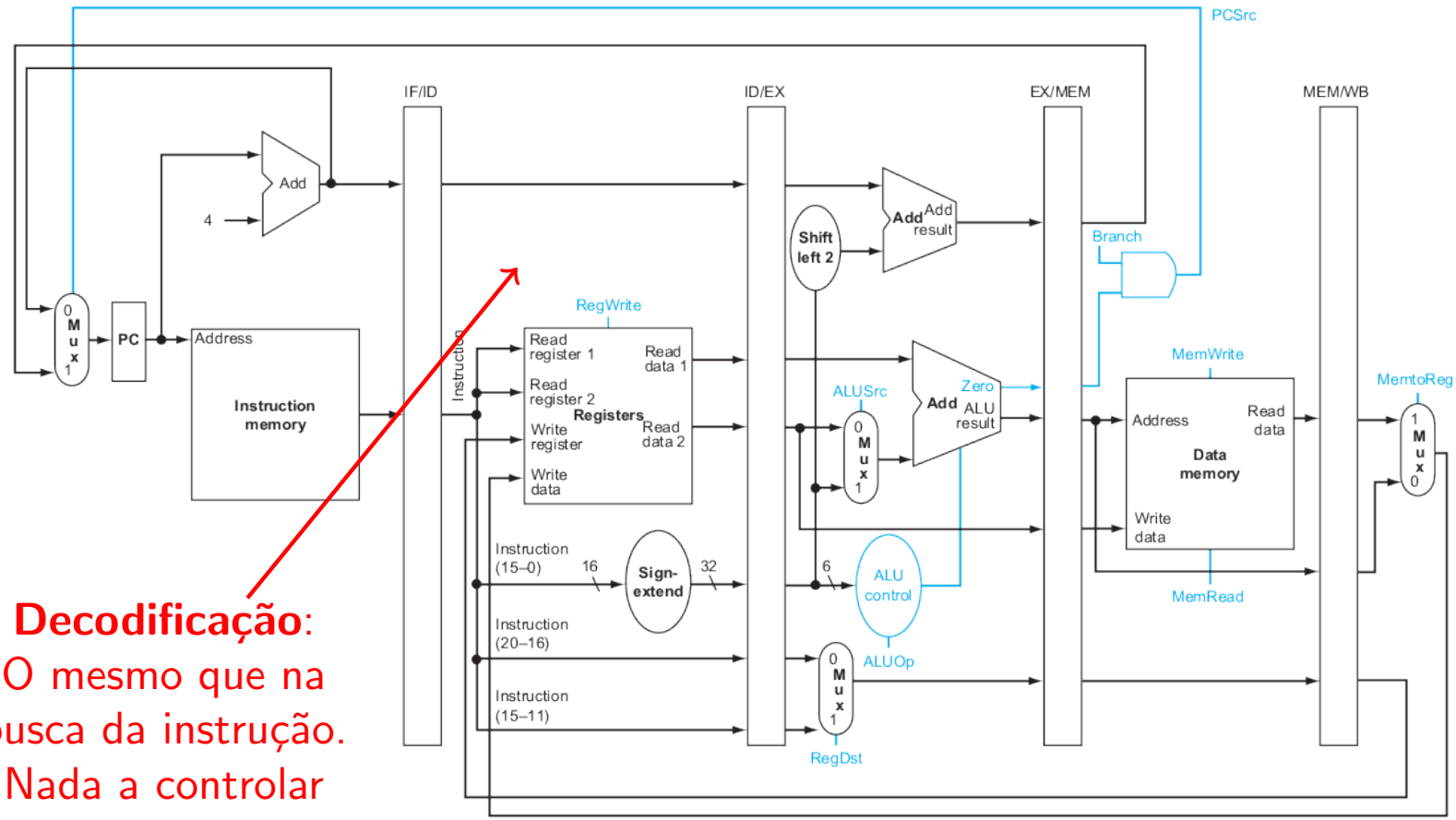
Controlando a *pipeline*



Busca de instrução: os sinais de controle estão sempre ligados, então não há nada para controlar

Fonte: [1]

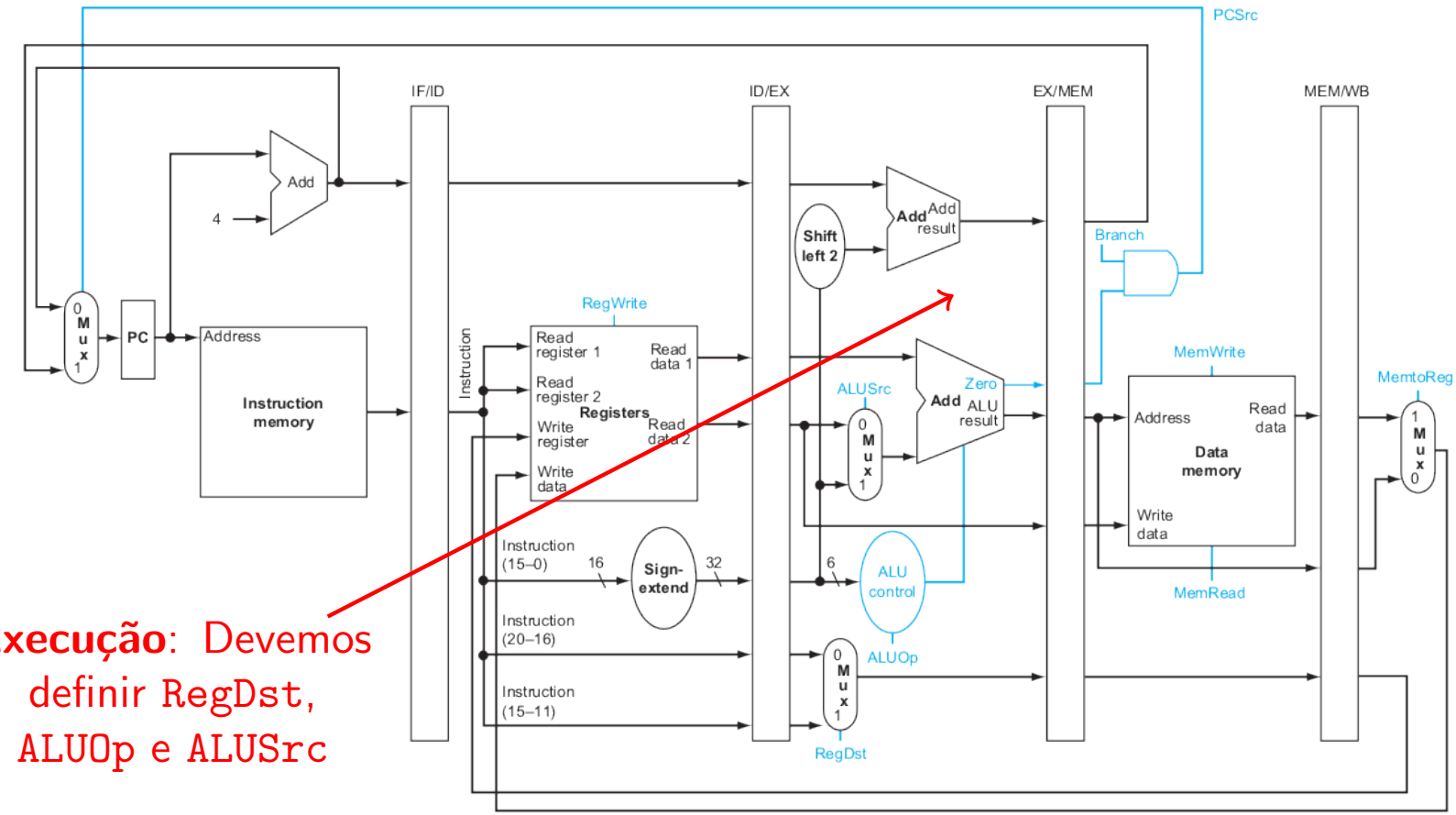
Controlando a *pipeline*



Decodificação:
O mesmo que na busca da instrução. Nada a controlar

Fonte: [1]

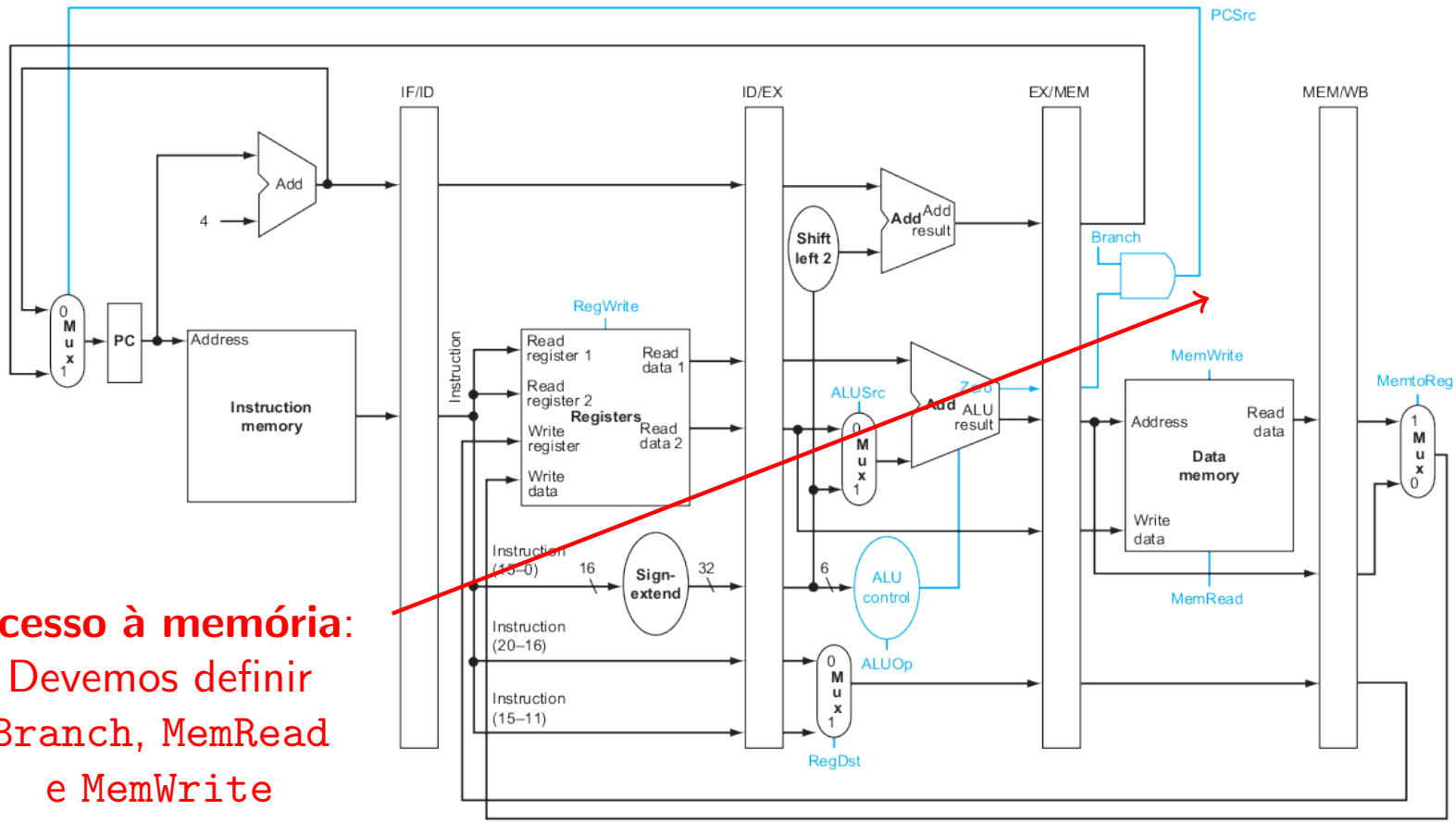
Controlando a *pipeline*



Execução: Devemos definir `RegDst`, `ALUOp` e `ALUSrc`

Fonte: [1]

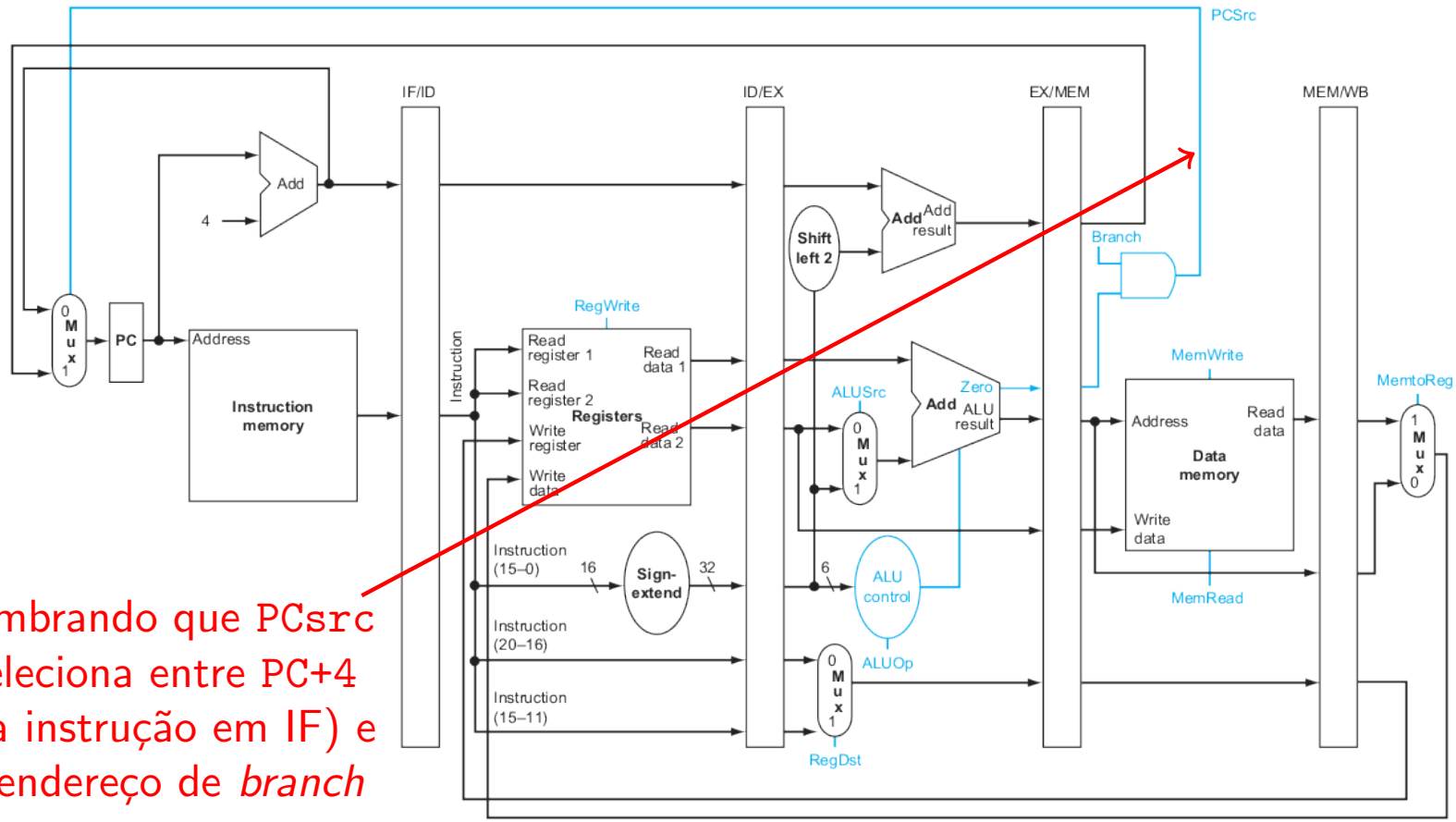
Controlando a *pipeline*



Acesso à memória:
Devemos definir
`Branch`, `MemRead`
e `MemWrite`

Fonte: [1]

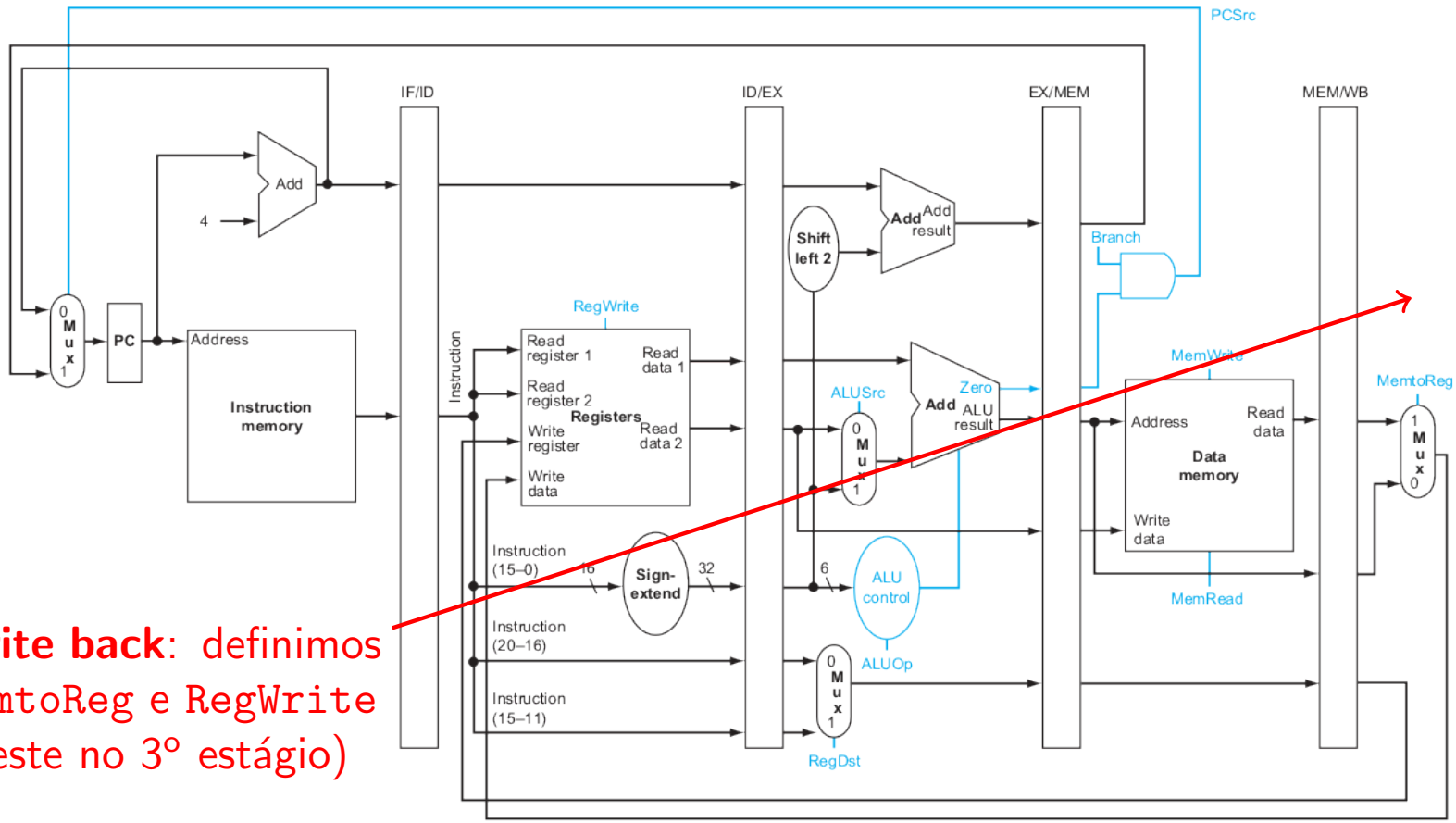
Controlando a *pipeline*



Lembrando que `PCsrc` seleciona entre PC+4 (da instrução em IF) e o endereço de *branch*

Fonte: [1]

Controlando a *pipeline*



Write back: definimos `MemtoReg` e `RegWrite` (este no 3º estágio)

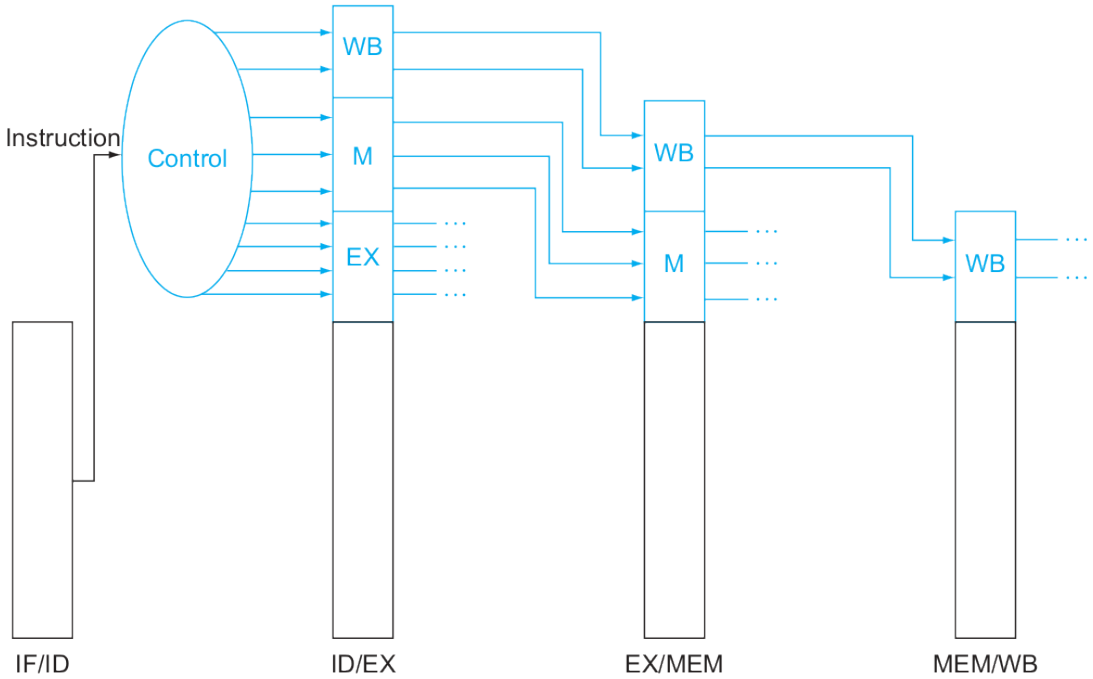
Fonte: [1]

Controlando a *pipeline*

- E como fica o controle então?

Controlando a *pipeline*

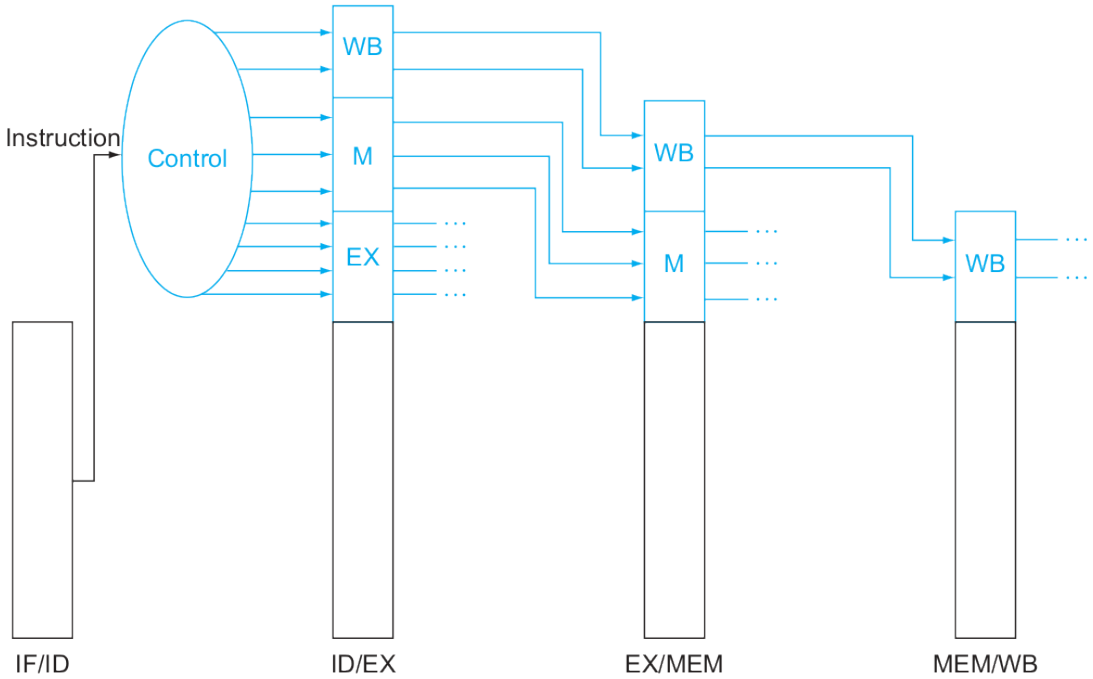
- E como fica o controle então?



Fonte: [1]

Controlando a *pipeline*

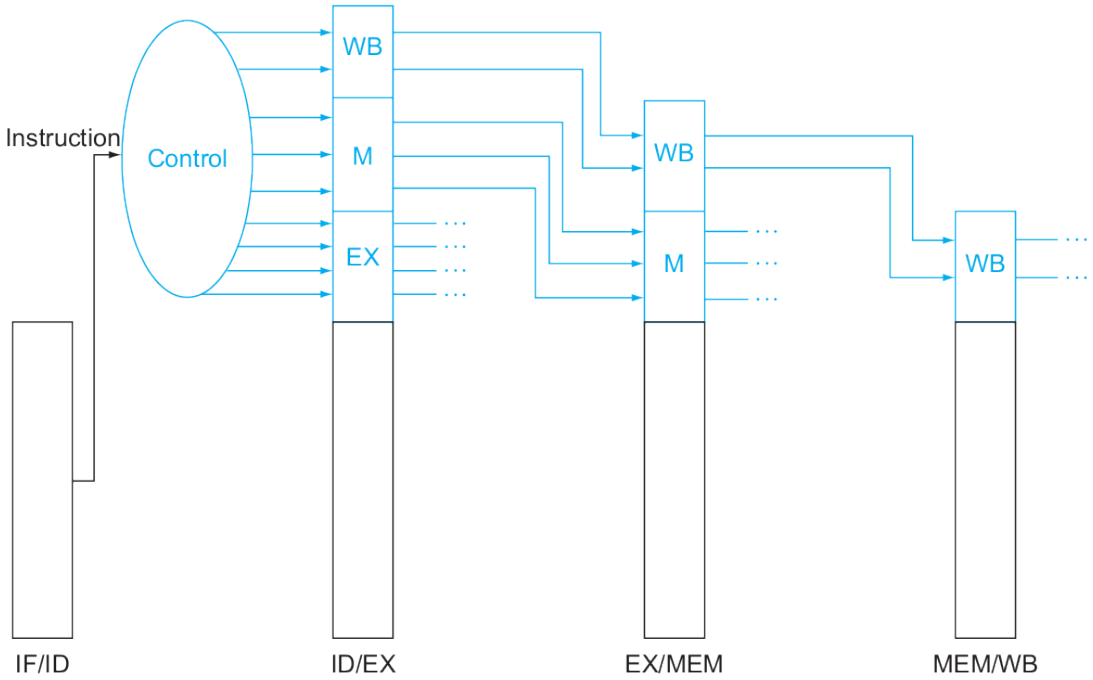
- E como fica o controle então?
  - Cada instrução carrega consigo os sinais necessários em cada estágio



Fonte: [1]

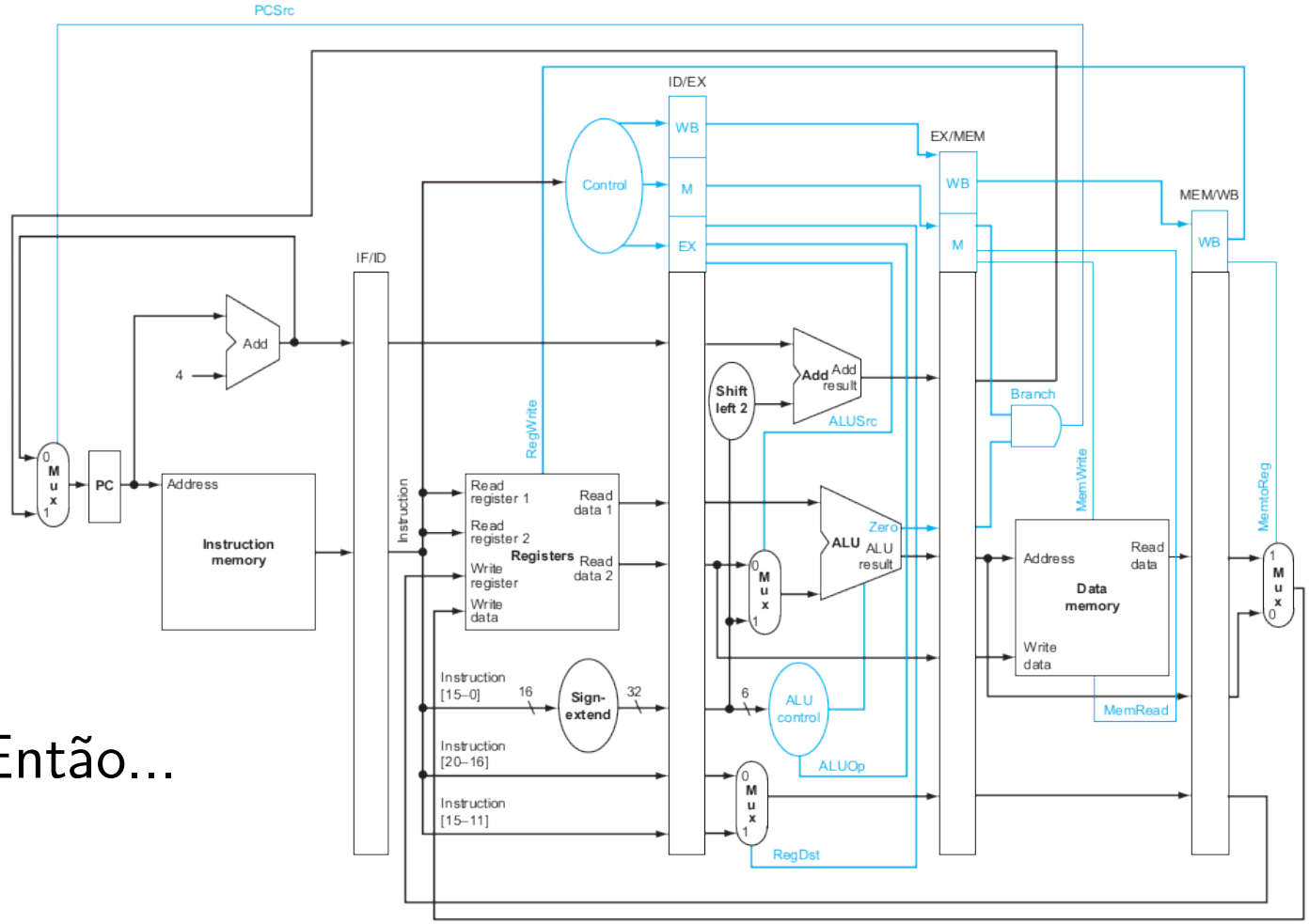
Controlando a *pipeline*

- E como fica o controle então?
  - Cada instrução carrega consigo os sinais necessários em cada estágio
  - Uma vez que as linhas começam no estágio de execução (EX), criamos a informação de controle durante a decodificação



Fonte: [1]

Controlando a *pipeline*



Então...

Fonte: [1]

Pipeline

Conflitos de *Pipeline*

- Considere as seguintes instruções

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

Pipeline

Conflitos de *Pipeline*

- Considere as seguintes instruções

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

Para completá-las
todas serão necessários
9 ciclos de *clock*

Conflitos de *Pipeline*

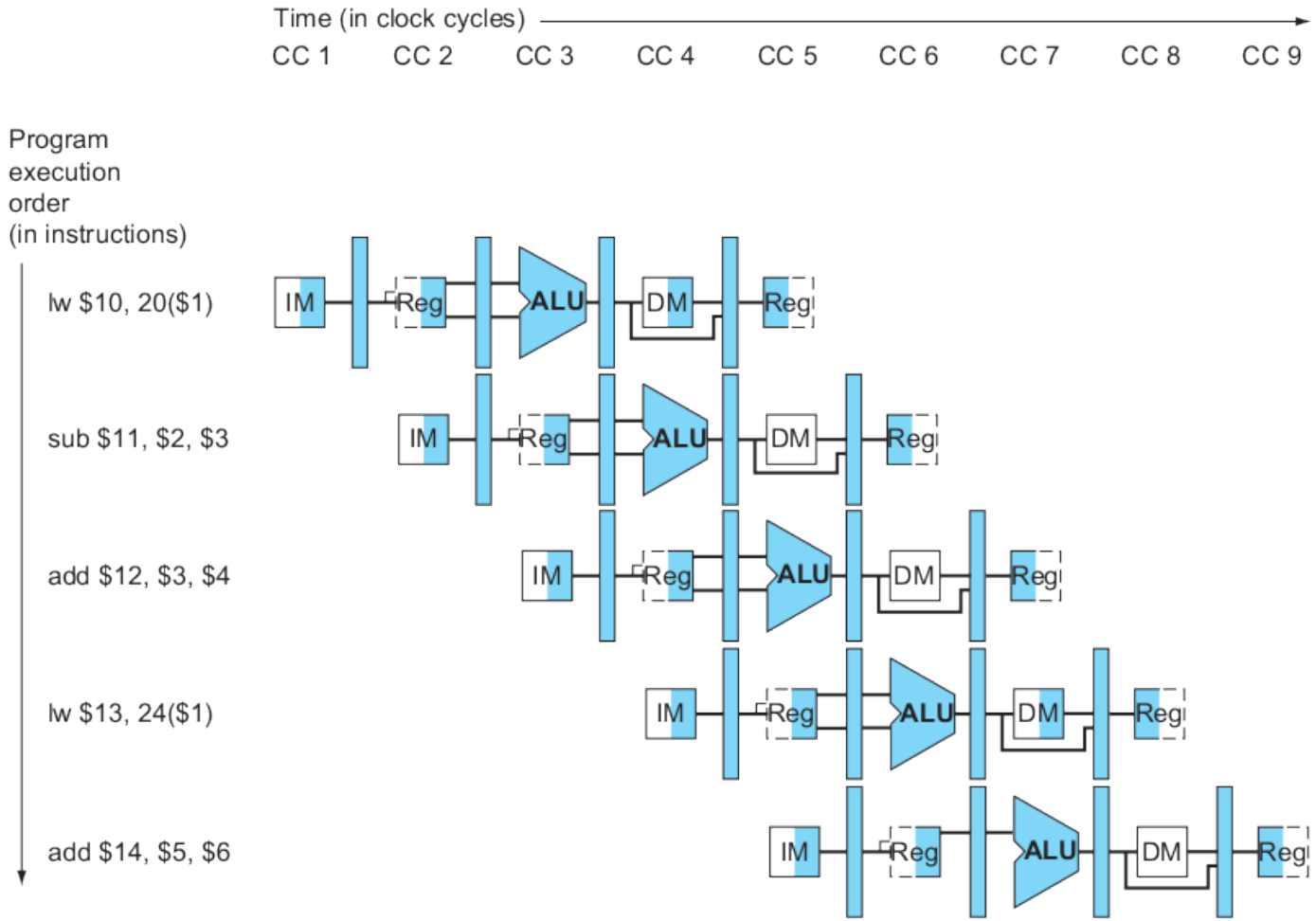
- Considere as seguintes instruções

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

Para completá-las
todas serão necessários
9 ciclos de *clock*

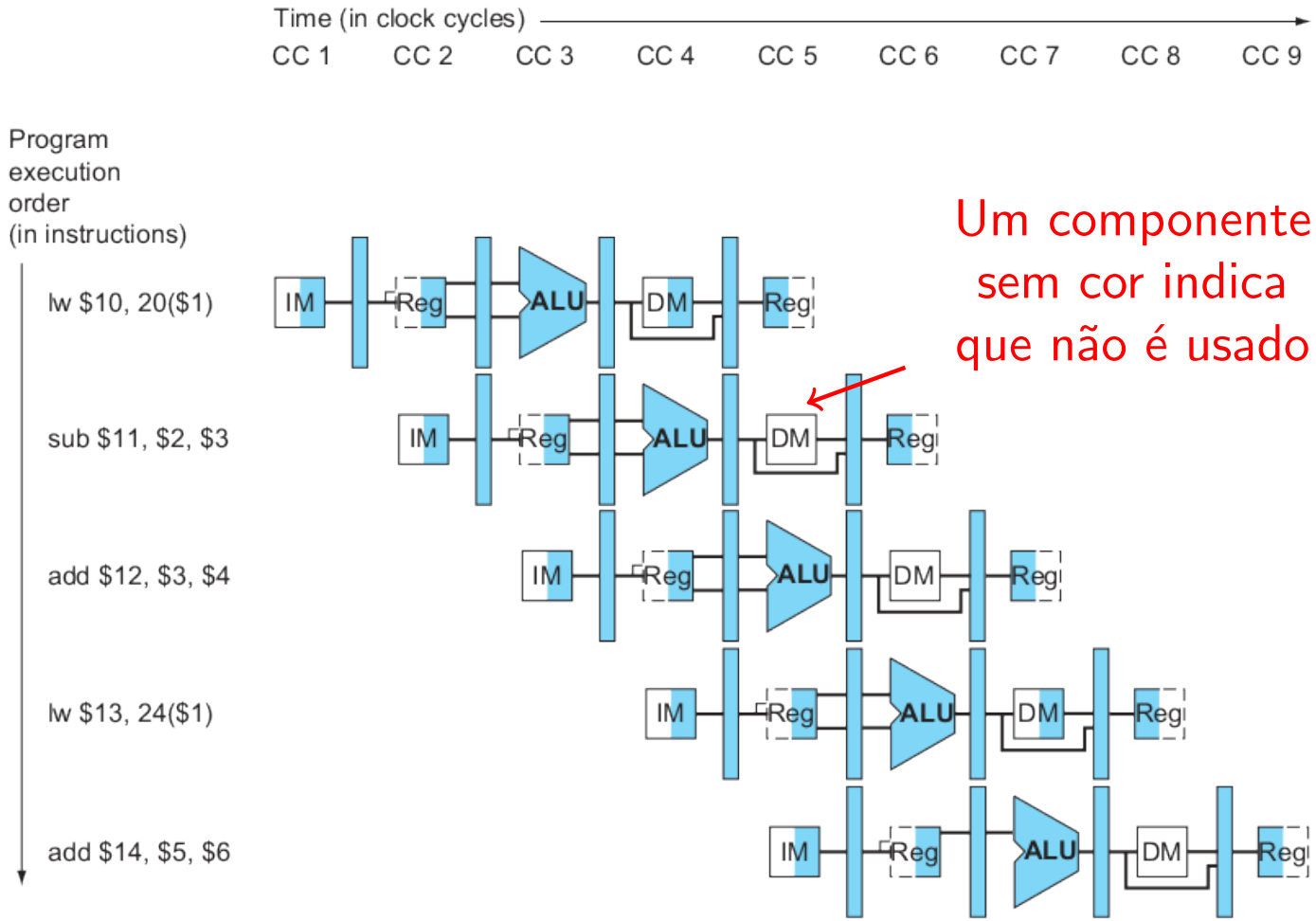
- Há algum problema com elas?

Conflitos de *Pipeline*



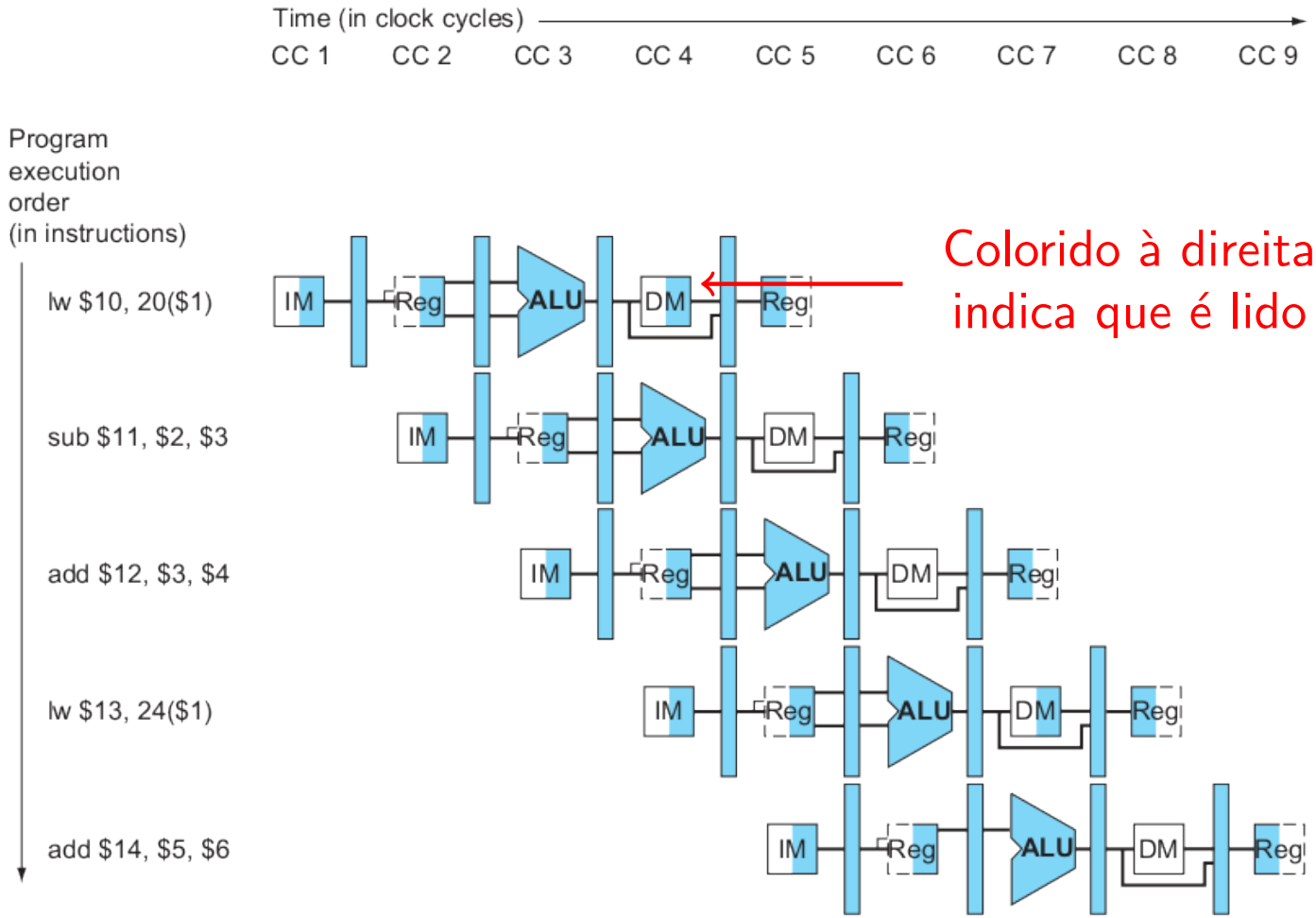
Fonte: [1]

Conflitos de *Pipeline*



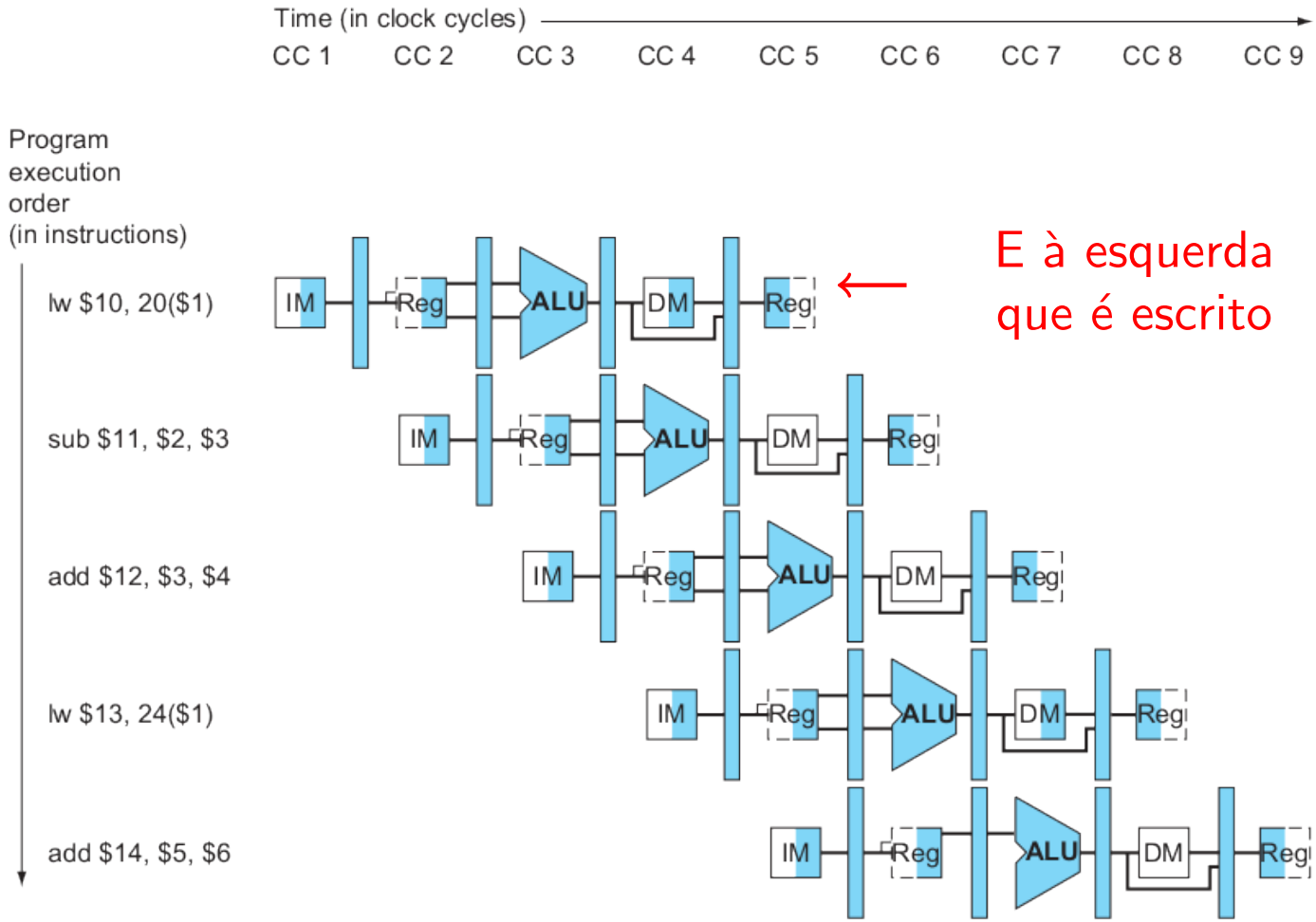
Fonte: [1]

Conflitos de *Pipeline*



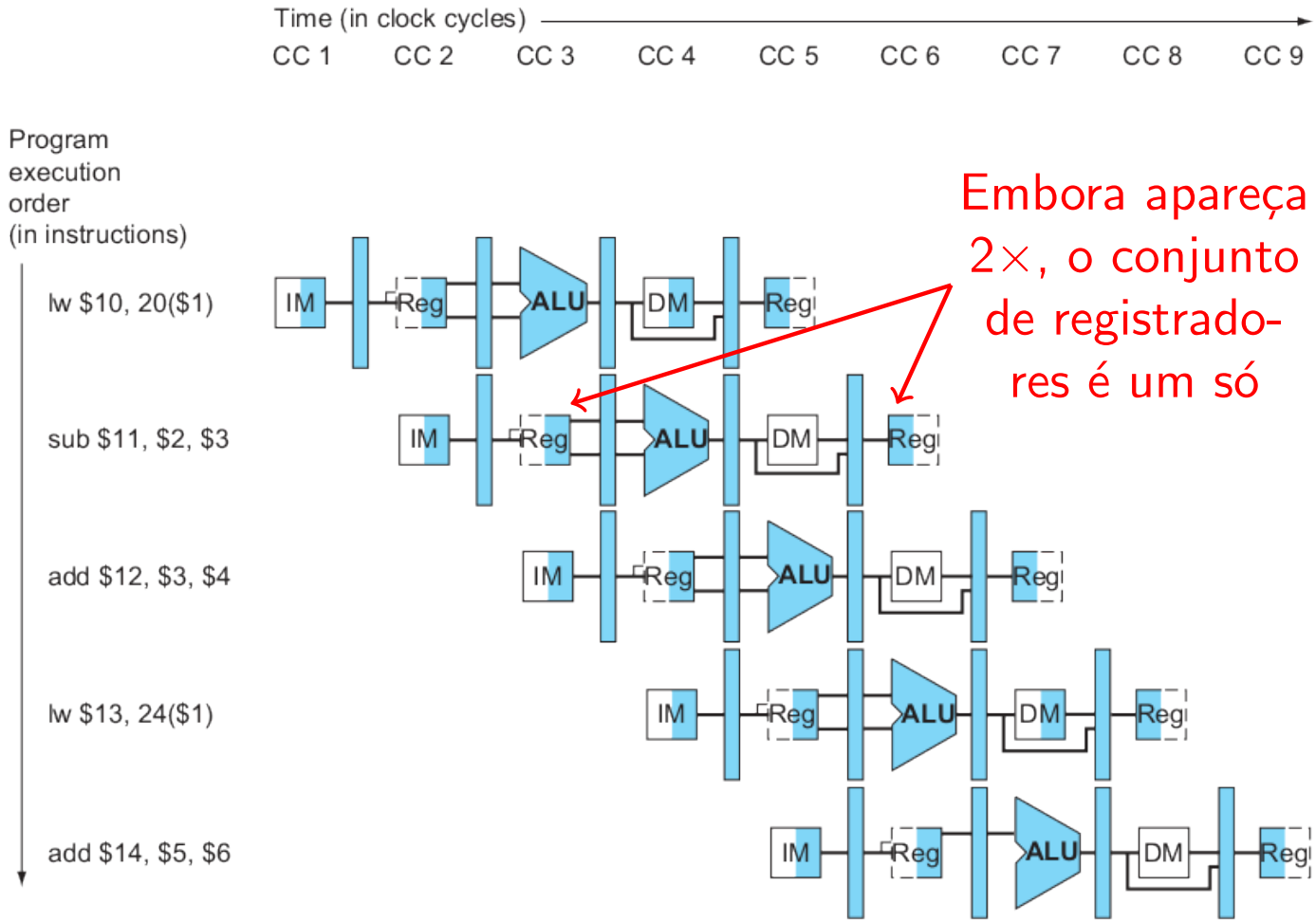
Fonte: [1]

Conflitos de *Pipeline*



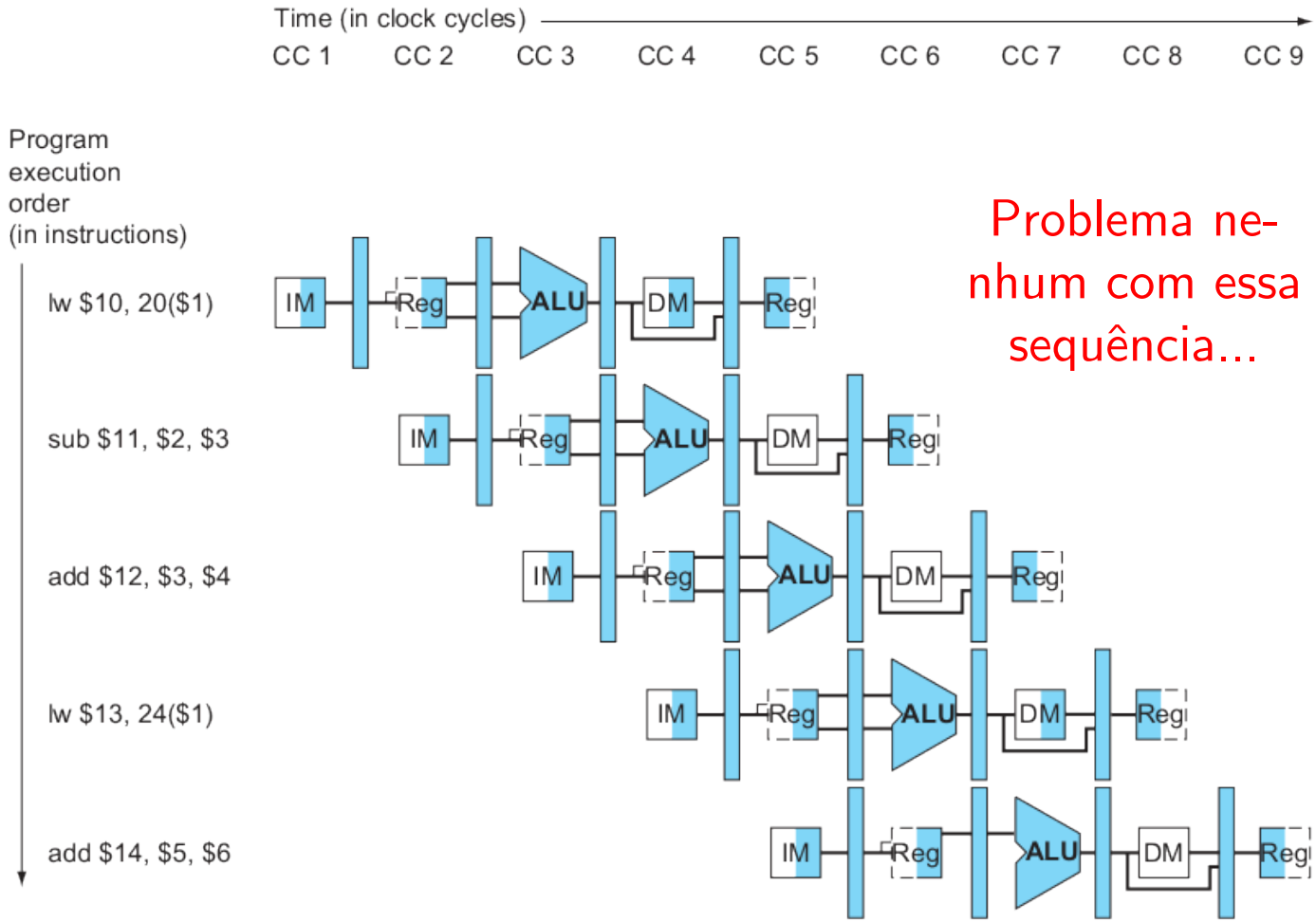
Fonte: [1]

Conflitos de *Pipeline*



Fonte: [1]

Conflitos de *Pipeline*



Problema nenhum com essa sequência...

Fonte: [1]

Pipeline

Conflitos de *Pipeline*

- E agora?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Pipeline

Conflitos de *Pipeline*

- E agora?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- Há uma grande dependência do valor de $2 definido em sub
  - Valor este definido no 5º estágio dessa instrução

Pipeline

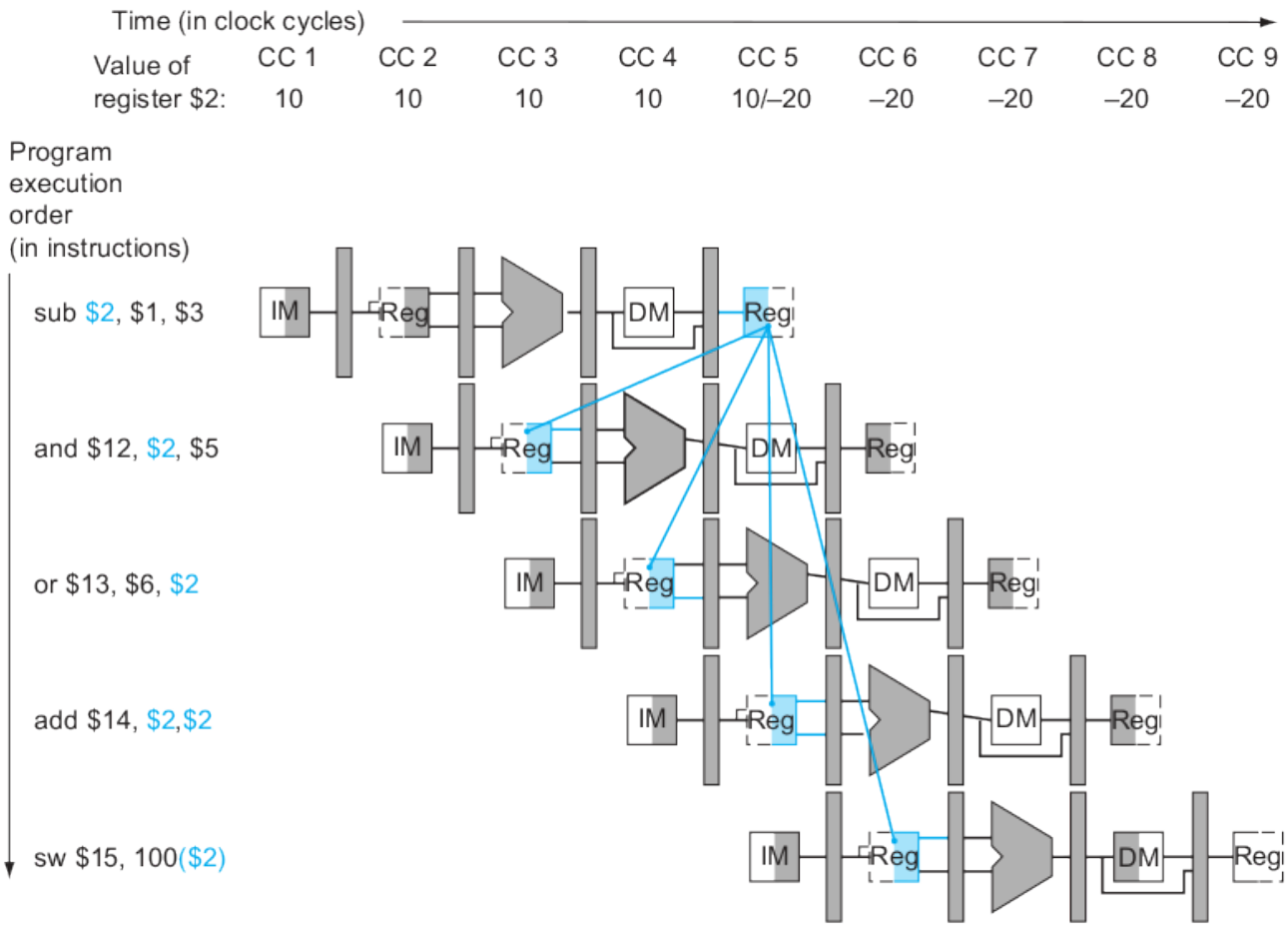
Conflitos de *Pipeline*

- E agora?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

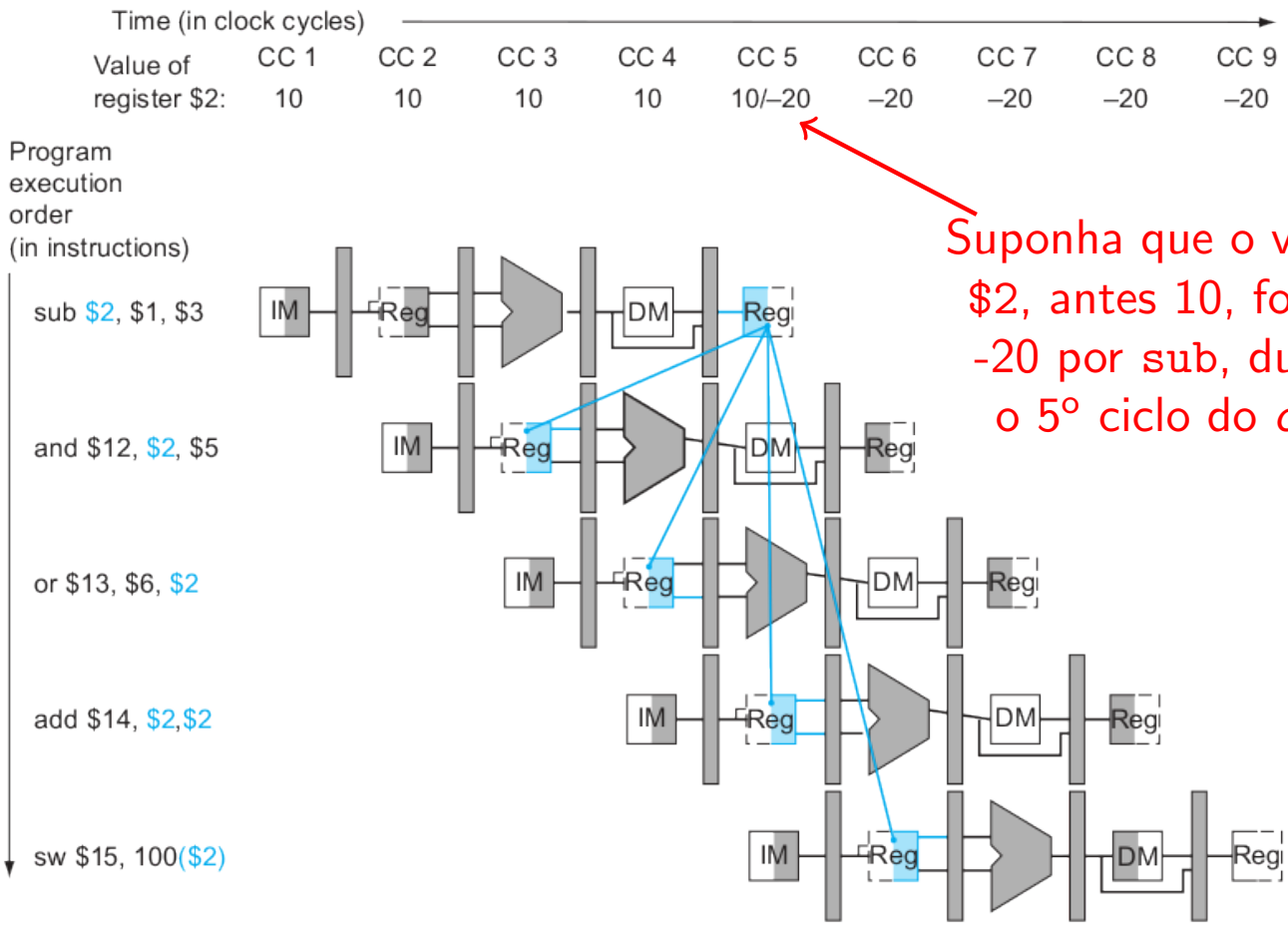
- Há uma grande dependência do valor de $2 definido em sub
  - Valor este definido no 5º estágio dessa instrução
- Mas quão grave é esse problema, e que instruções serão afetadas?

Conflitos de *Pipeline*



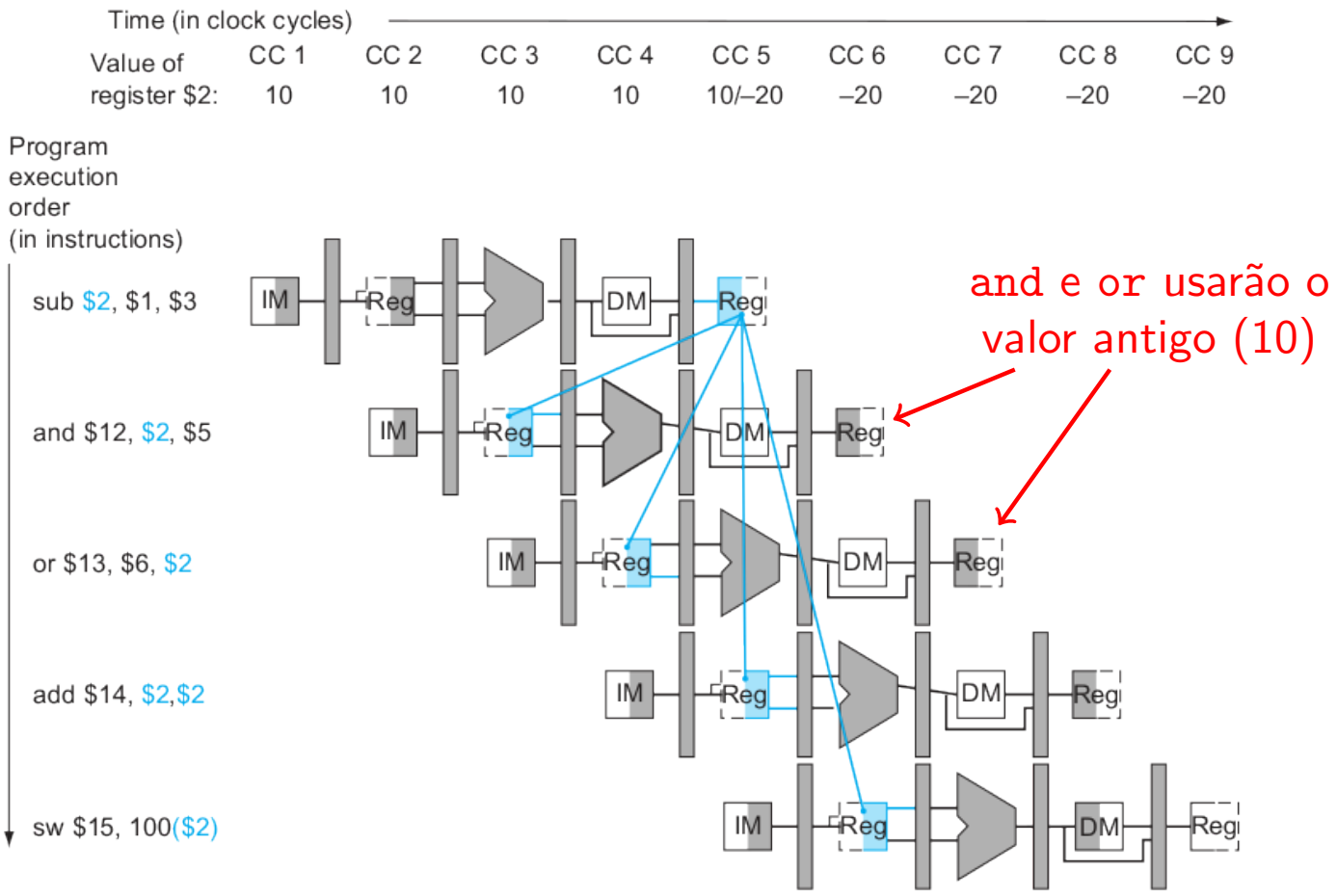
Fonte: [1]

Conflitos de *Pipeline*



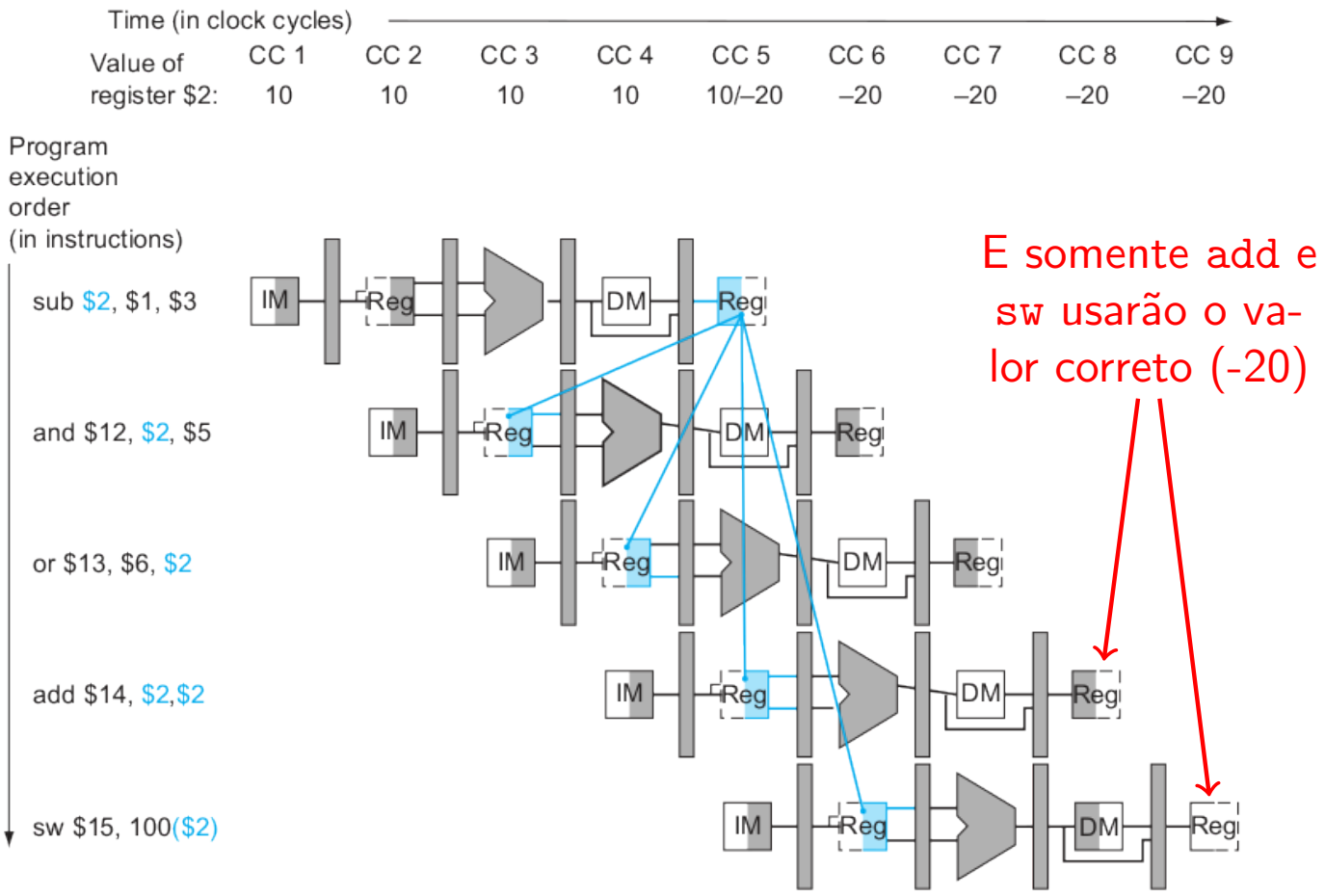
Fonte: [1]

Conflitos de *Pipeline*



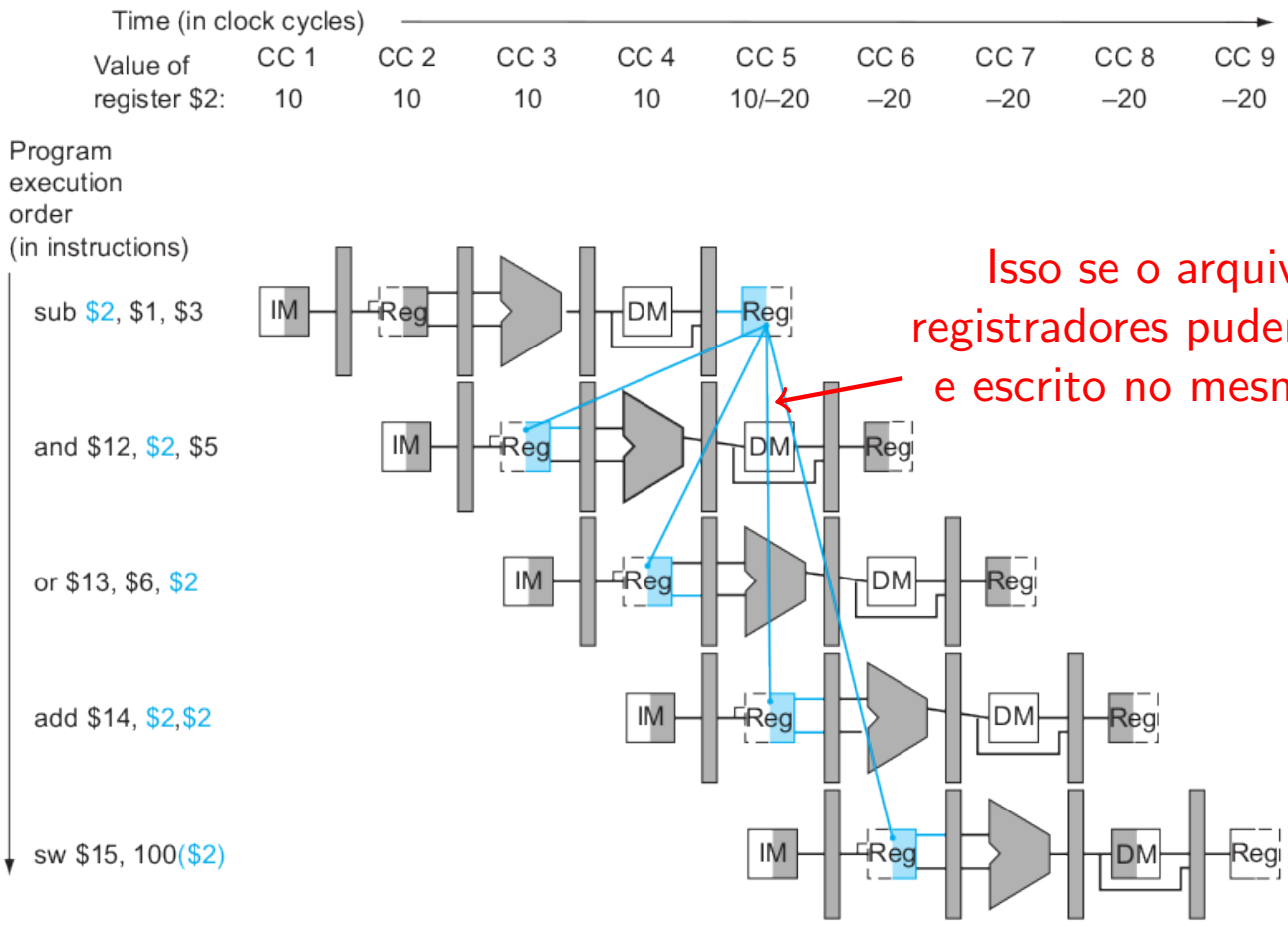
Fonte: [1]

Conflitos de *Pipeline*



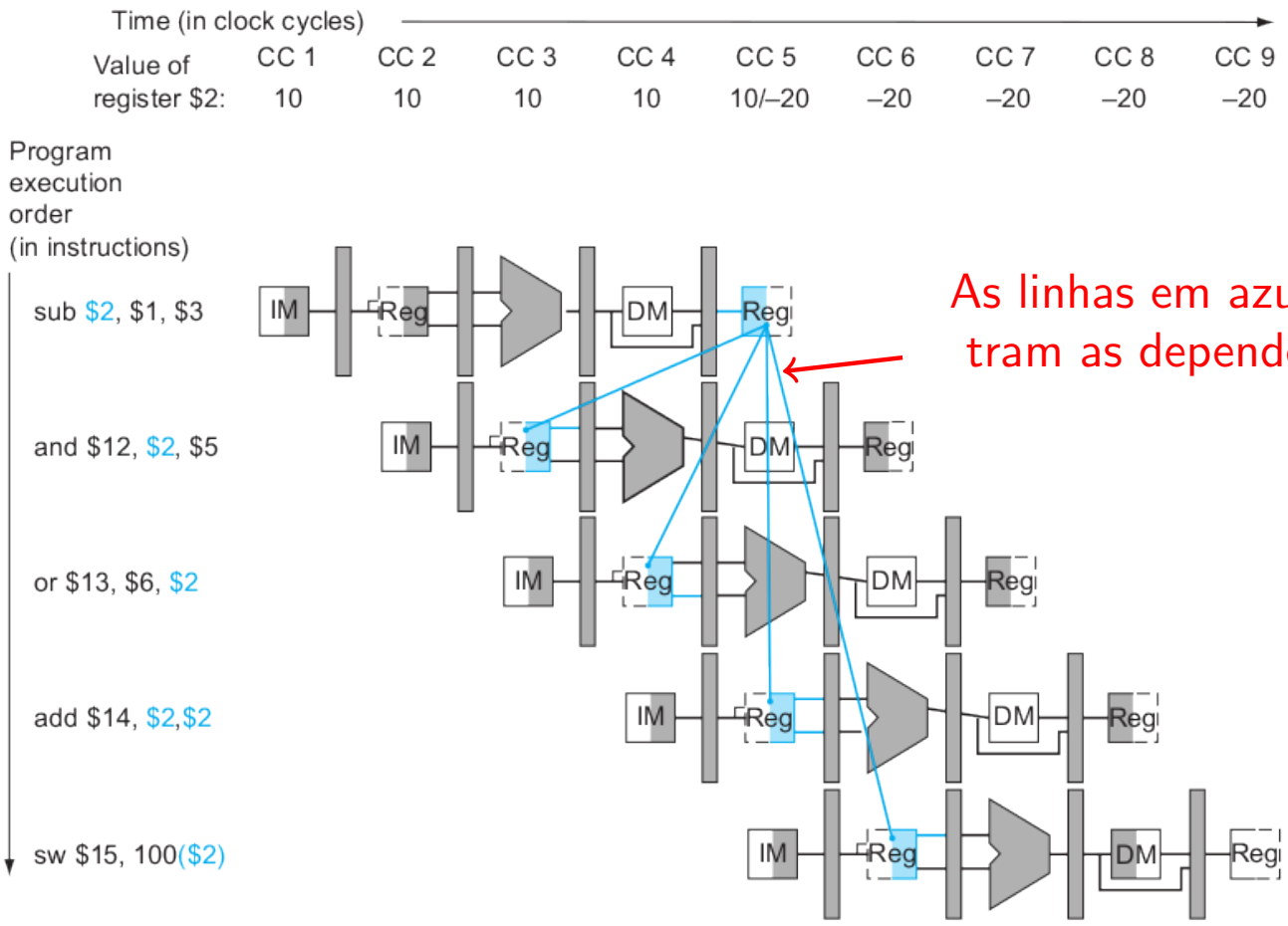
Fonte: [1]

Conflitos de *Pipeline*



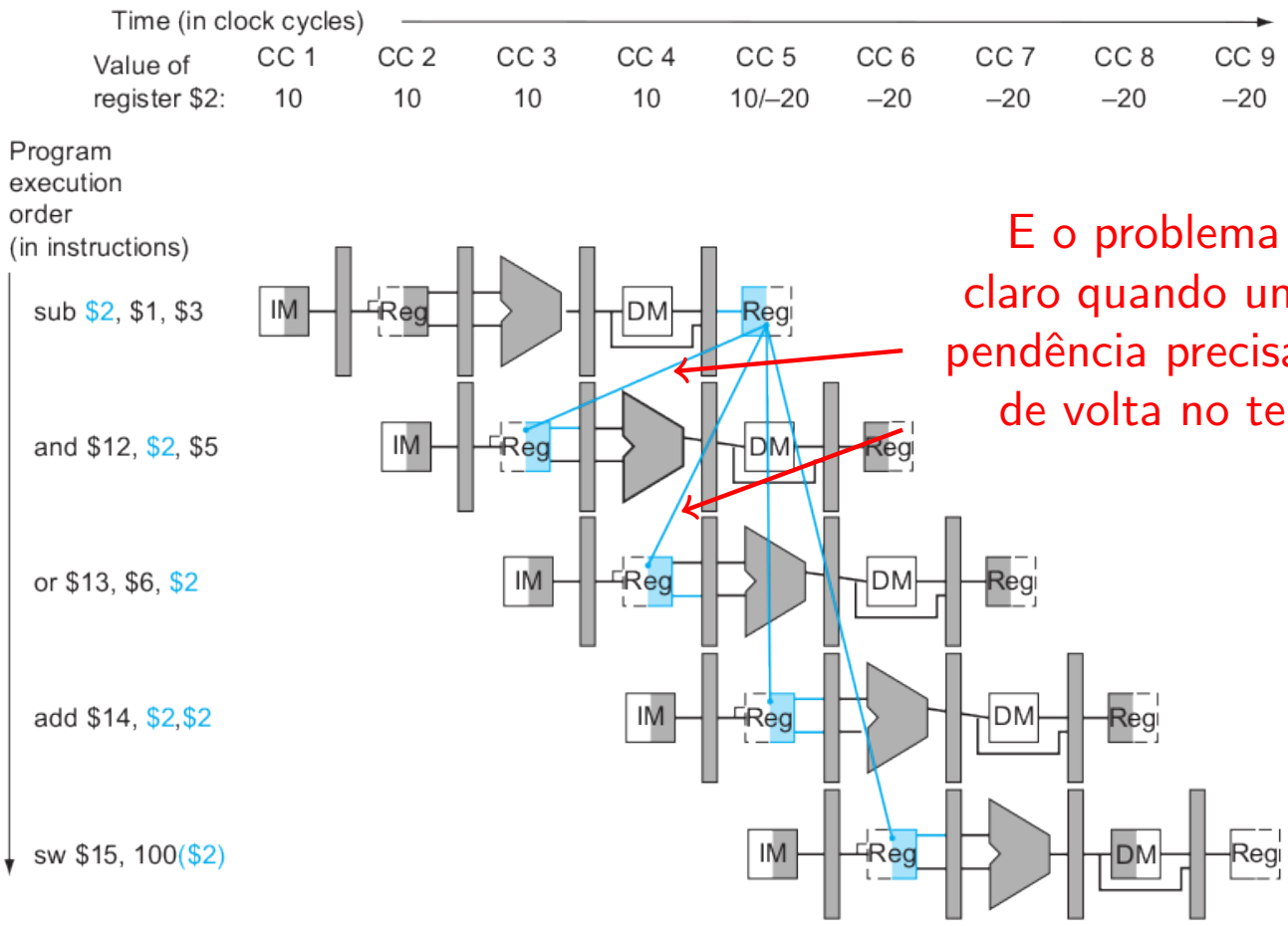
Fonte: [1]

Conflitos de *Pipeline*



Fonte: [1]

Conflitos de *Pipeline*



E o problema fica claro quando uma dependência precisa viajar de volta no tempo

Fonte: [1]

Conflitos de *Pipeline*

- Temos aqui um caso de **conflito de pipeline** (*pipeline hazard*)
  - Uma situação em que a instrução seguinte não pode executar no próximo ciclo de *clock*

Conflitos de *Pipeline*

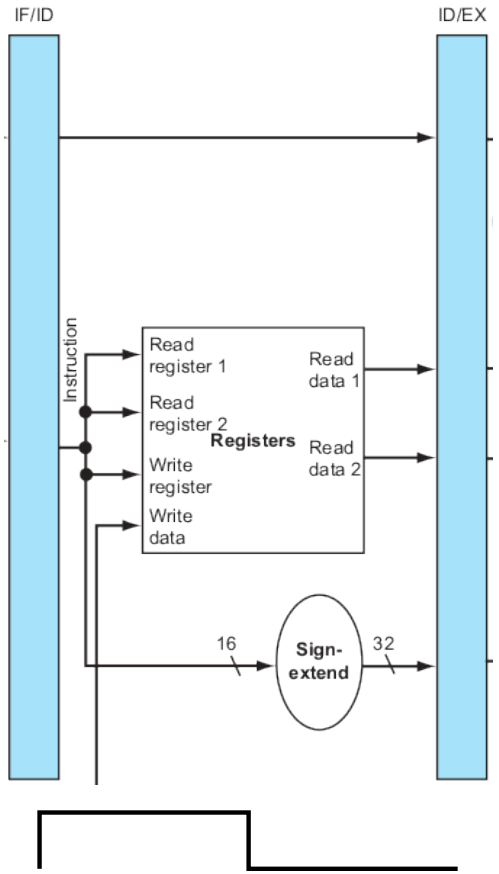
- Temos aqui um caso de **conflito de pipeline** (*pipeline hazard*)
  - Uma situação em que a instrução seguinte não pode executar no próximo ciclo de *clock*
- Há 3 tipos diferentes de conflitos:
  - Estruturais
  - De dados
  - De controle

Conflitos de *Pipeline*

- Conflitos estruturais
  - Quando o hardware não permite que uma determinada combinação de instruções seja executada no mesmo ciclo de *clock*

Conflitos de *Pipeline*

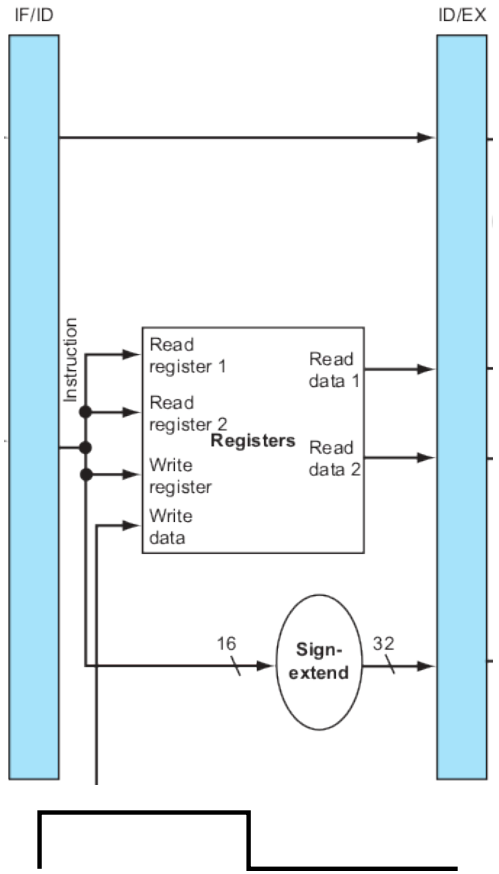
- Conflitos estruturais
  - Quando o hardware não permite que uma determinada combinação de instruções seja executada no mesmo ciclo de *clock*
  - Ex: se o arquivo de registradores for ativado na mesma borda do *clock* que os registradores da *pipeline*



Fonte: Adaptado de [1]

Conflitos de *Pipeline*

- Ou se tivéssemos uma única memória
  - Nesse caso, Teríamos um conflito toda vez que buscássemos uma nova instrução no mesmo ciclo em que outra acessa dados da memória



Fonte: Adaptado de [1]

Conflitos de *Pipeline*

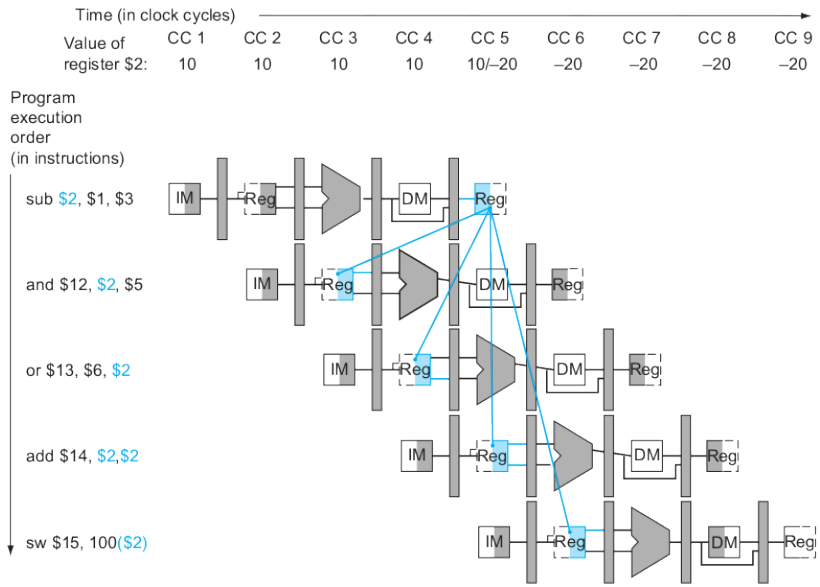
- Conflitos de Dados
  - Quando uma instrução não pode executar porque o dado de que necessita não está disponível

Conflitos de *Pipeline*

- Conflitos de Dados
  - Quando uma instrução não pode executar porque o dado de que necessita não está disponível
  - Surgem da dependência de uma instrução com relação a outra anterior que ainda está na *pipeline*

Conflitos de *Pipeline*

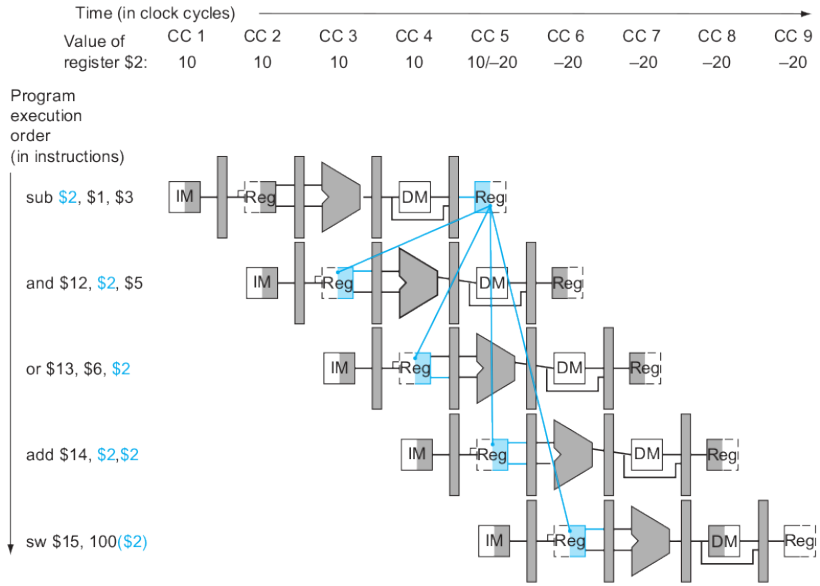
- Conflitos de Dados
  - Quando uma instrução não pode executar porque o dado de que necessita não está disponível
  - Surgem da dependência de uma instrução com relação a outra anterior que ainda está na *pipeline*
  - É o tipo visto em nosso exemplo inicial



Fonte: [1]

Conflitos de *Pipeline*

- Conflitos de Dados
  - Precisamos então parar (*stall*) a *pipeline* até a 1ª instrução se completar

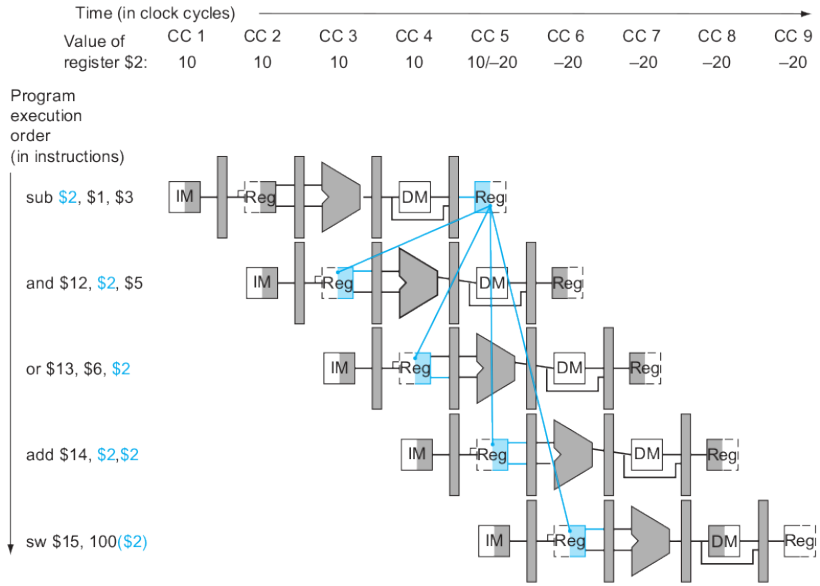


Fonte: [1]

Conflitos de *Pipeline*

- Conflitos de Dados
  - Precisamos então parar (*stall*) a *pipeline* até a 1ª instrução se completar
  - Suponha agora que temos as seguintes instruções:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```



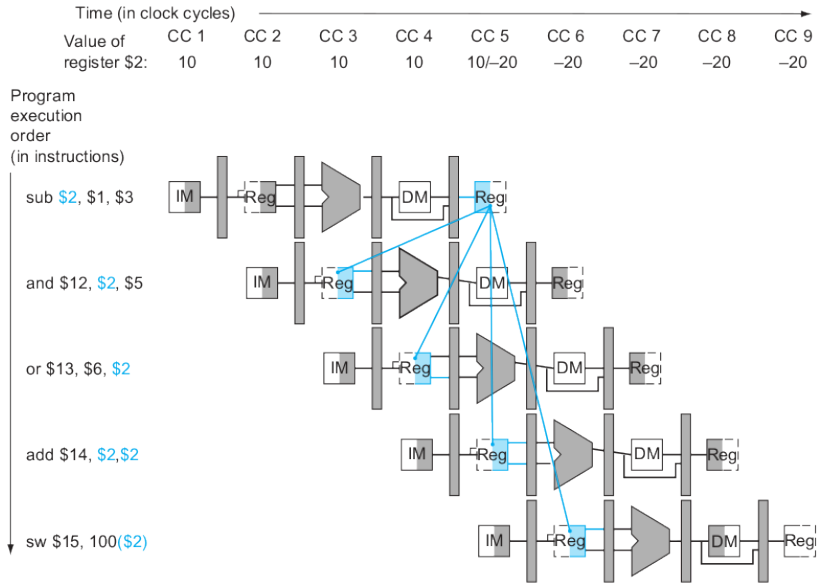
Fonte: [1]

Conflitos de *Pipeline*

- Conflitos de Dados
  - Precisamos então parar (*stall*) a *pipeline* até a 1ª instrução se completar
  - Suponha agora que temos as seguintes instruções:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

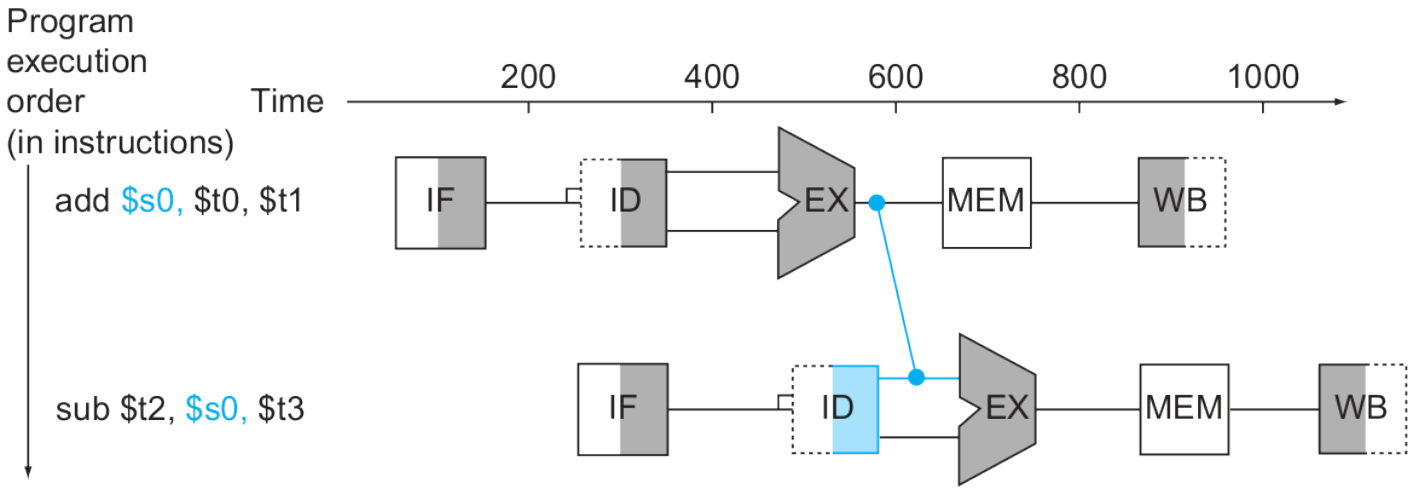
  - Temos um conflito



Fonte: [1]

Conflitos de *Pipeline*: Conflitos de Dados

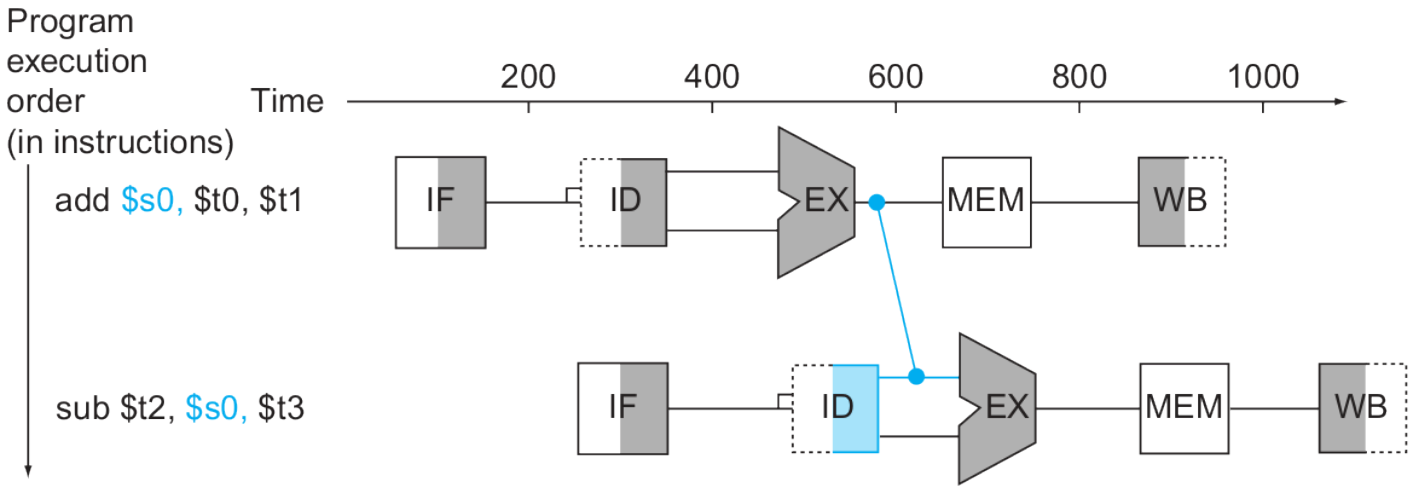
- A solução parte do princípio de que não precisamos esperar que a instrução se complete:
  - Assim que a soma é calculada, podemos enviá-la a `sub`



Fonte: [1]

Conflitos de *Pipeline*: Conflitos de Dados

- Processo denominado *forwarding* ou *bypassing*
  - Requer, naturalmente, a adição de *hardware*
  - Além de ser válido somente se o estágio de destino estiver mais no futuro que o atual



Fonte: [1]

Conflitos de Dados: *Forwarding*

- *Forwarding*, contudo, não previne todo *stall*

Pipeline

Conflitos de Dados: *Forwarding*

- *Forwarding*, contudo, não previne todo *stall*
- Suponha que as instruções fossem:

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```

Conflitos de Dados: *Forwarding*

- *Forwarding*, contudo, não previne todo *stall*
- Suponha que as instruções fossem:

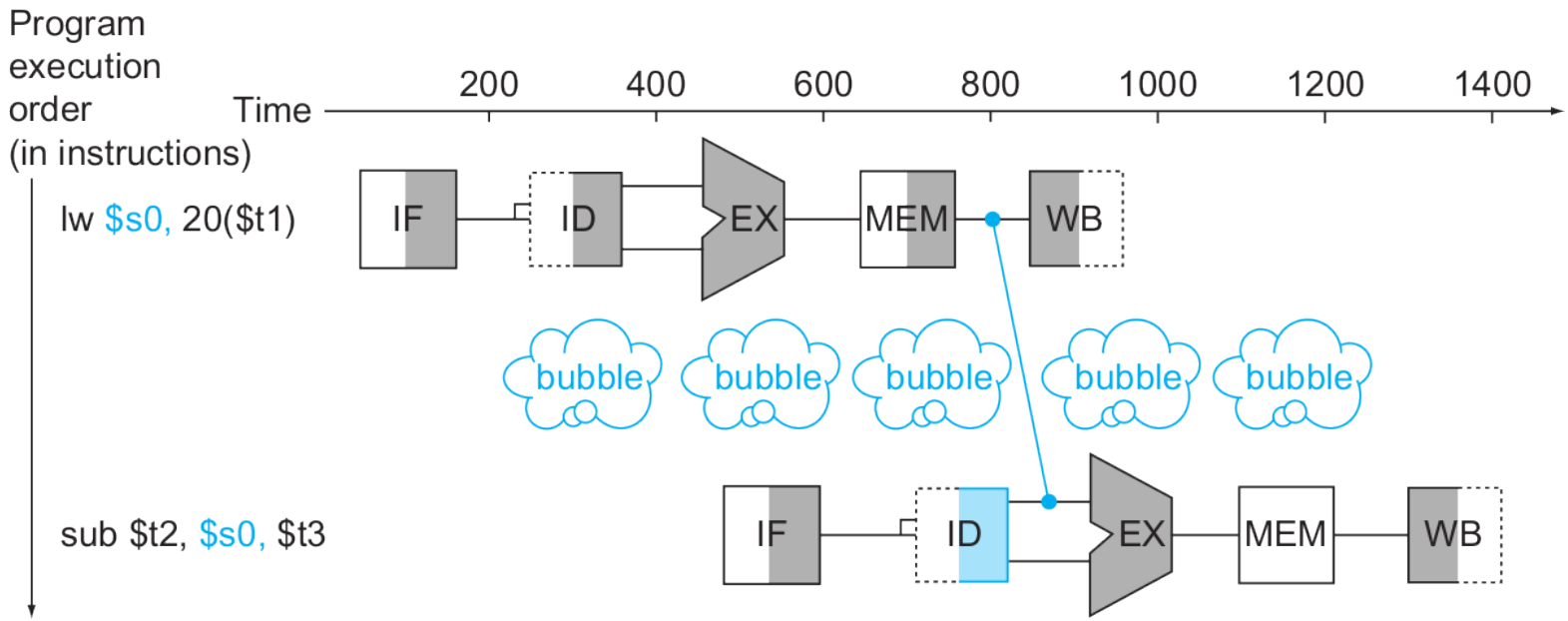
```
lw $s0, 20($t1)

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

- O dado estaria disponível somente após o 4º estágio de `lw`
  - Tarde demais para a entrada de `sub` no 3º estágio

Conflitos de Dados: *Forwarding*

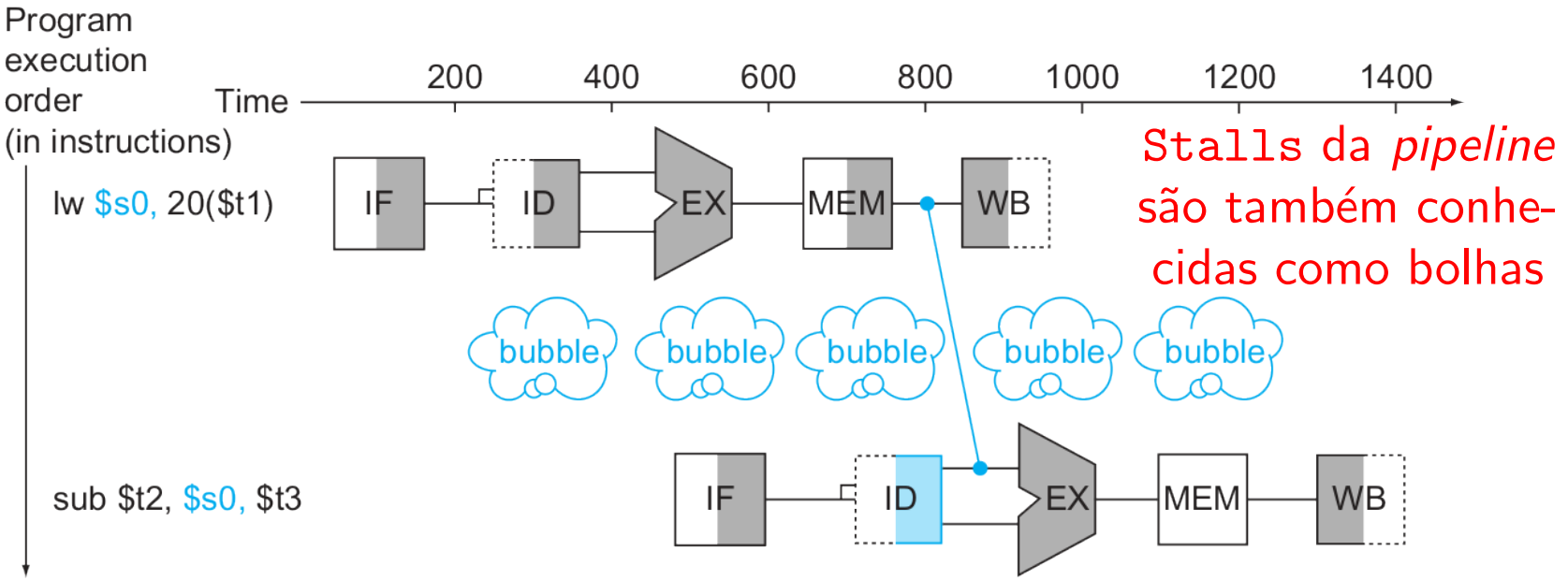
- Mesmo com *forwarding*, ainda precisaríamos parar a *pipeline* por um estágio



Fonte: [1]

Conflitos de Dados: *Forwarding*

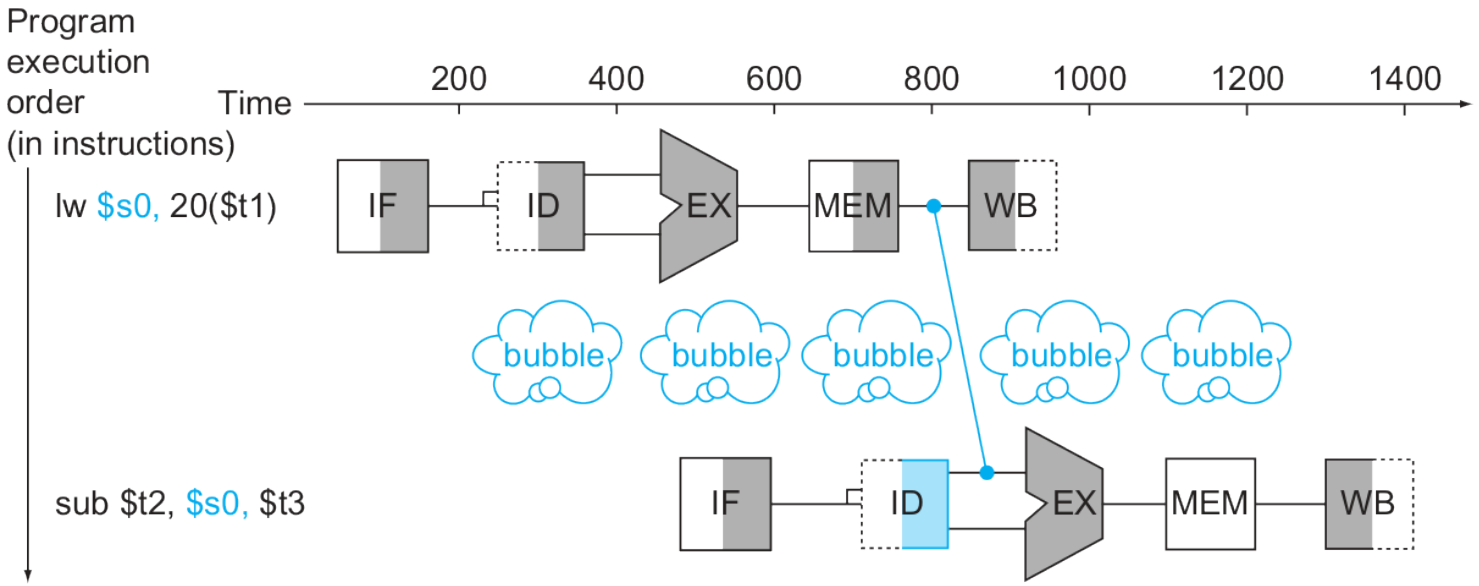
- Mesmo com *forwarding*, ainda precisaríamos parar a *pipeline* por um estágio



Fonte: [1]

Conflitos de Dados: *Forwarding*

- Uma maneira de lidar com isso é reordenar o código, quando possível
  - Procedimento adotado por compiladores e montadores



Fonte: [1]

Conflitos de *Pipeline*

- Conflitos de Controle (ou de *Branch*)
  - Quando precisamos tomar uma decisão de controle baseada no resultado de uma instrução enquanto outras executam
    - O atraso em determinar qual instrução rodar é chamado de **conflito de controle**

Conflitos de *Pipeline*

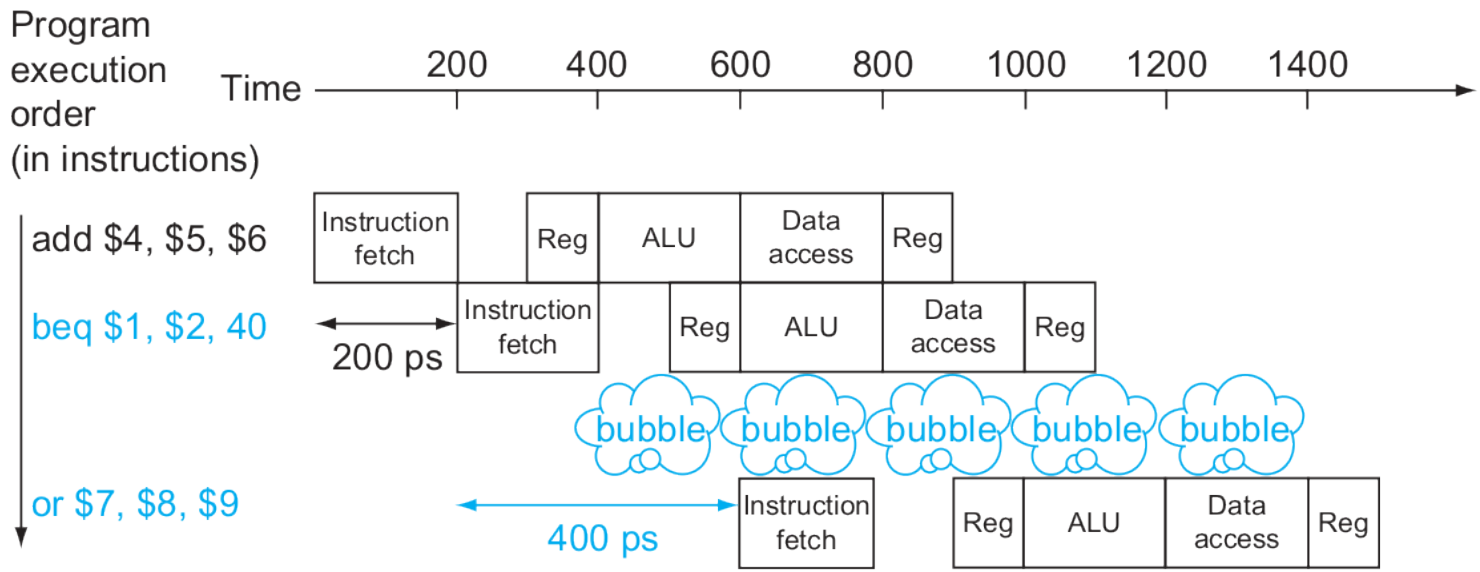
- Conflitos de Controle (ou de *Branch*)
  - Quando precisamos tomar uma decisão de controle baseada no resultado de uma instrução enquanto outras executam
    - O atraso em determinar qual instrução rodar é chamado de **conflito de controle**
  - Como, quando em um `beq`, a instrução que está na *pipeline* não é a que deveria ser rodada

Conflitos de *Pipeline*

- Conflitos de Controle (ou de *Branch*)
  - Quando precisamos tomar uma decisão de controle baseada no resultado de uma instrução enquanto outras executam
    - O atraso em determinar qual instrução rodar é chamado de **conflito de controle**
  - Como, quando em um `beq`, a instrução que está na *pipeline* não é a que deveria ser rodada
  - Solução 1: *stall*
    - Pare a *pipeline* imediatamente após buscar um *branch* da memória
    - Aguarde até que a *pipeline* possa determinar o resultado do *branch*

Conflitos de *Pipeline*: Conflitos de Controle

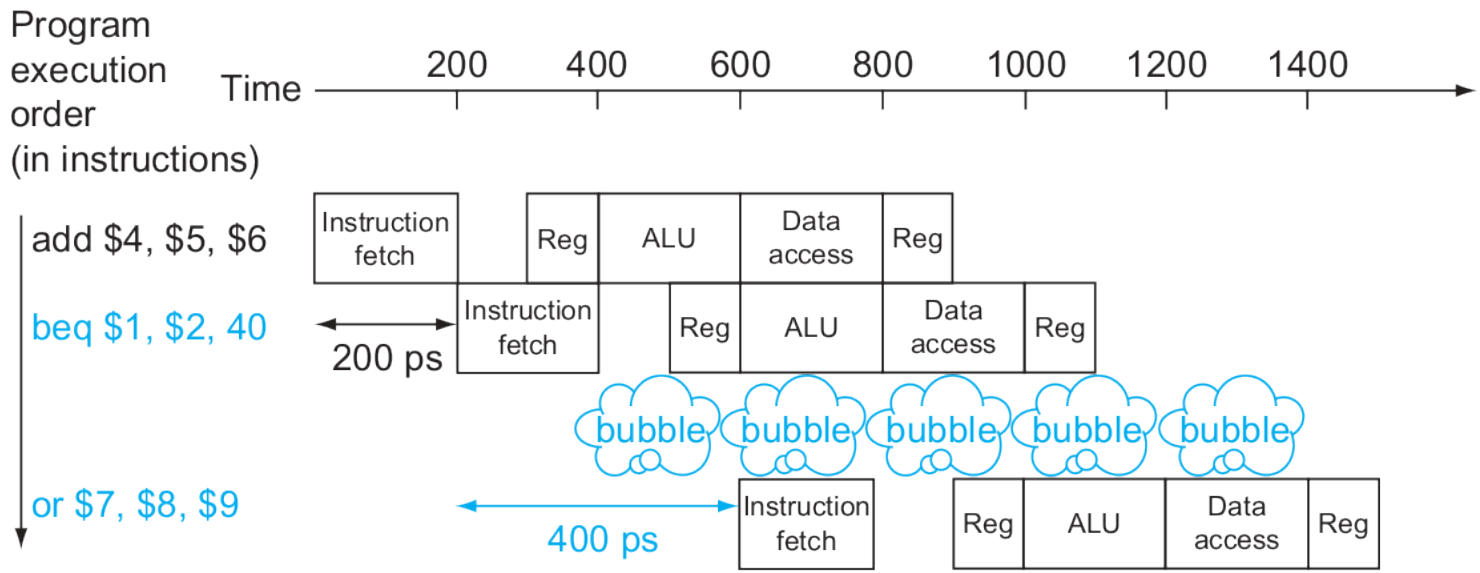
- Mesmo com adição de hardware para todo o cálculo do *branch* no 2º estágio, ainda teríamos paradas
  - Pois a nova instrução só poderia ser buscada após isso



Fonte: [1]

Conflitos de *Pipeline*: Conflitos de Controle

- Então o custo dessa opção é alto
  - Pois teríamos que parar a cada *branch*



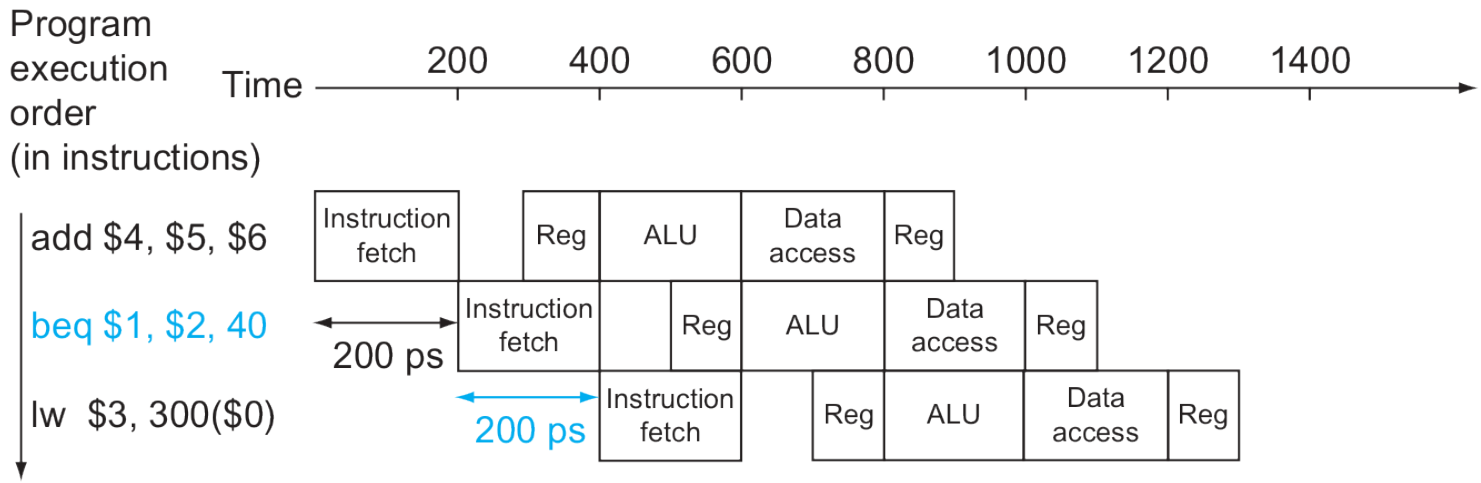
Fonte: [1]

Conflitos de *Pipeline*: Conflitos de Controle

- Solução 2: Predição
  - Assuma (prediga) que o *branch* sempre irá falhar

Conflitos de *Pipeline*: Conflitos de Controle

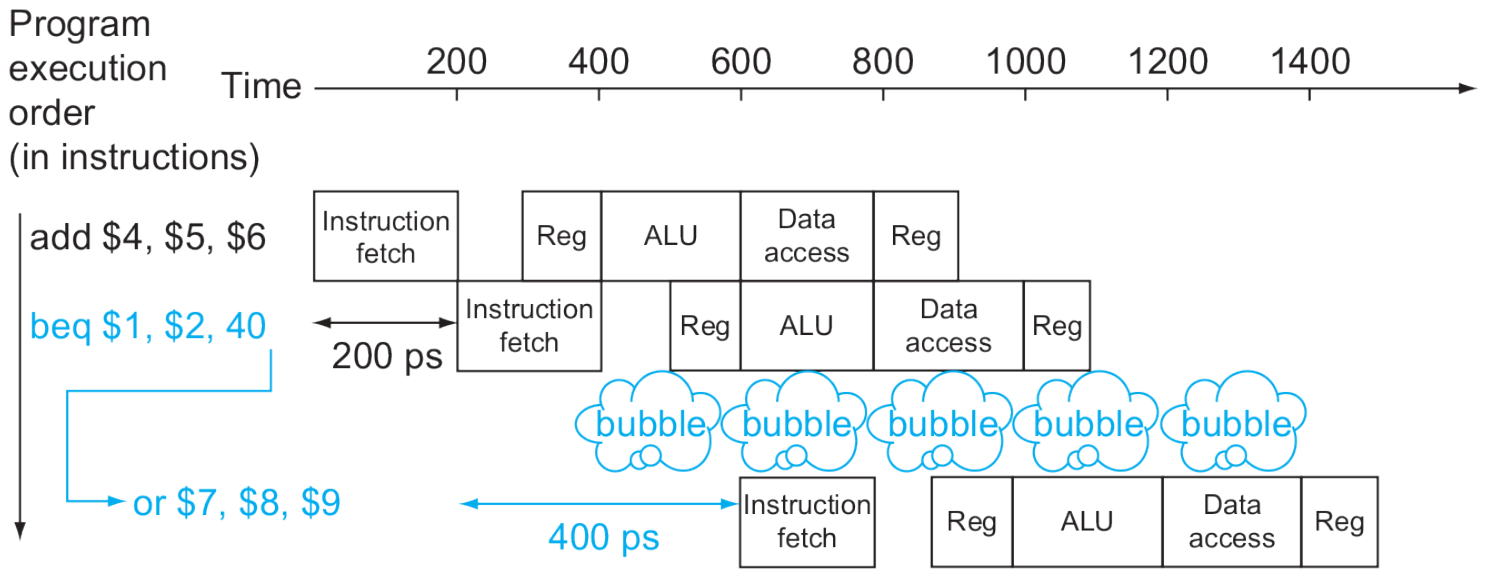
- Solução 2: Predição
  - Assuma (prediga) que o *branch* sempre irá falhar
  - Se estivermos corretos, não haverá atraso



Fonte: [1]

Conflitos de *Pipeline*: Conflitos de Controle

- Se estivermos errados, haverá o atraso
- Além do controle ter que "limpar" a pipeline das instruções incorretamente em andamento



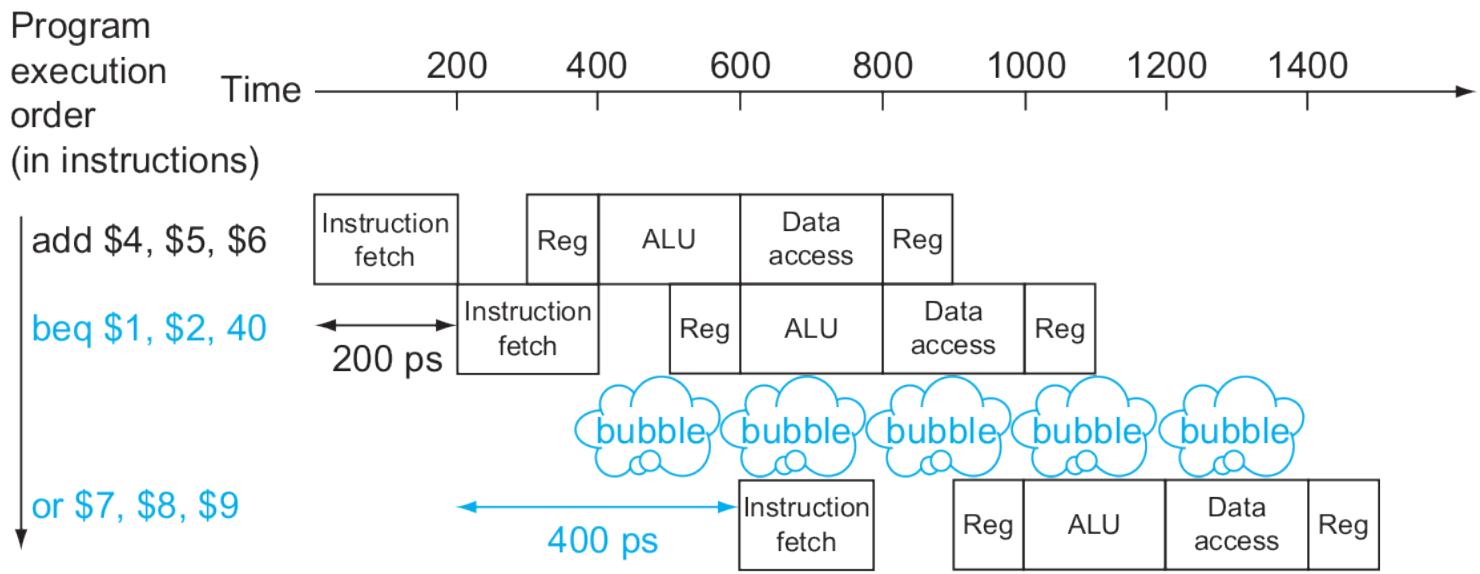
Fonte: [1]

Conflitos de *Pipeline*: Conflitos de Controle

- Solução 3: Decisão Adiada
  - Imediatamente após o *branch*, posicione uma instrução que não seja afetada por ele
  - Assim, o *branch* muda o endereço somente da instrução seguinte a esta instrução
  - Solução automaticamente implementada pelo montador

Conflitos de Controle: Decisão Adiada

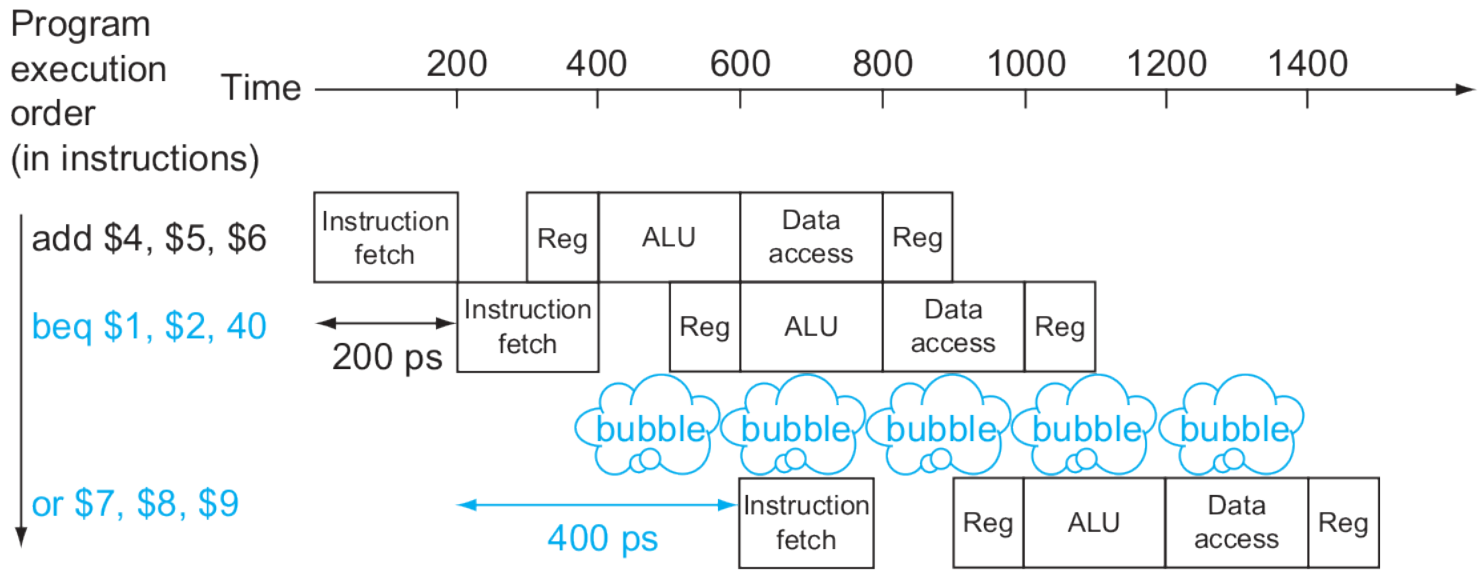
- No exemplo abaixo, o add não afeta o branch



Fonte: Adaptado de [1]

Conflitos de Controle: Decisão Adiada

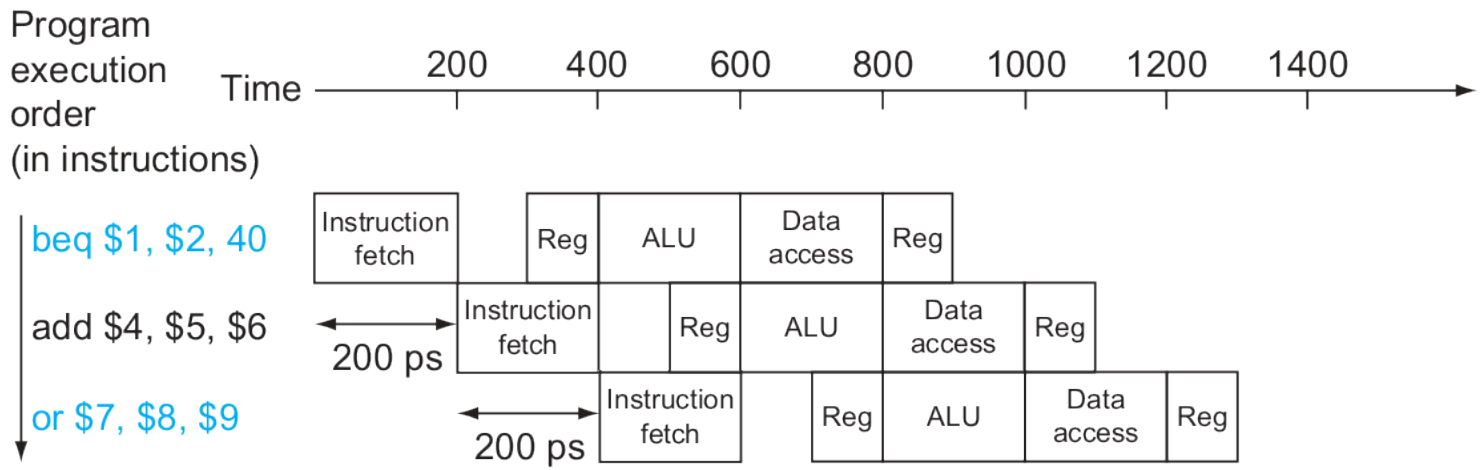
- No exemplo abaixo, o add não afeta o branch
  - Podemos então movê-lo para depois do *branch*, escondendo o atraso



Fonte: Adaptado de [1]

Conflitos de Controle: Decisão Adiada

- No exemplo abaixo, o add não afeta o branch
  - Podemos então movê-lo para depois do *branch*, escondendo o atraso



Fonte: Adaptado de [1]

Referências

1. Patterson, D.A.; Hennessy, J.L. (2013): Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann. 5ª ed.
   - Para detalhes sobre as partes do circuito consulte também o Apêndice B e a seção avançada 4.13 (*online*)
2. `http://bellerofonte.dii.unisi.it/index.asp`
   - Simulador de uma arquitetura de pipeline (MIPS)