

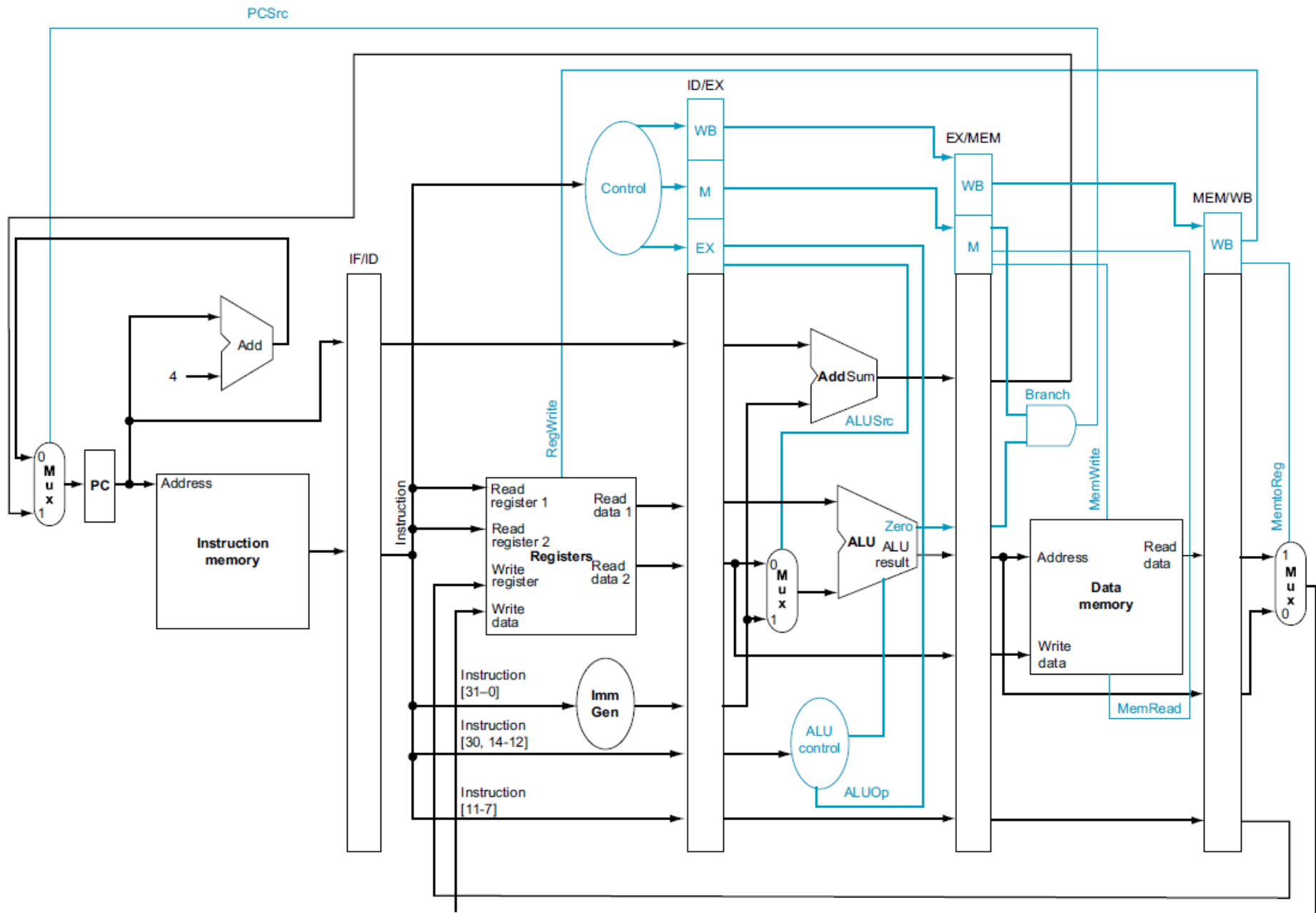


ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

Arquitetura RISC-V Pipelined

Profª. Fabiana F F Peres

Apoio: Camile Bordini



Conflitos no pipeline

- **Conflito estrutural:** Situação onde o hardware não foi projetado adequadamente para atender a combinação de instruções que se deseja executar num mesmo ciclo de clock
- **Conflito de Dados:** A execução de uma instrução depende do resultado de outra que ainda está em execução
- **Conflito de Controle:** Necessidade de tomar a decisão de qual será a próxima instrução a ser buscada com base nos resultados de uma instrução ainda em execução

Pipeline

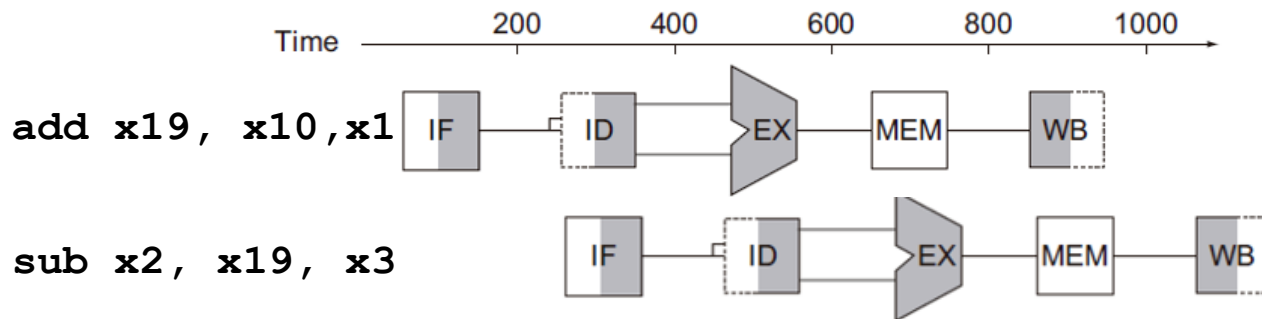
Banco de Registradores
antes da execução das
duas instruções

X0	0
X1	4
X2	2
X3	8
...	
X10	3
...	
X19	14
...	

Banco de Registradores
(**desejado**) após a execução
das duas instruções

X0	0
X1	4
X2	-1
X3	8
...	
X10	3
...	
X19	7
...	

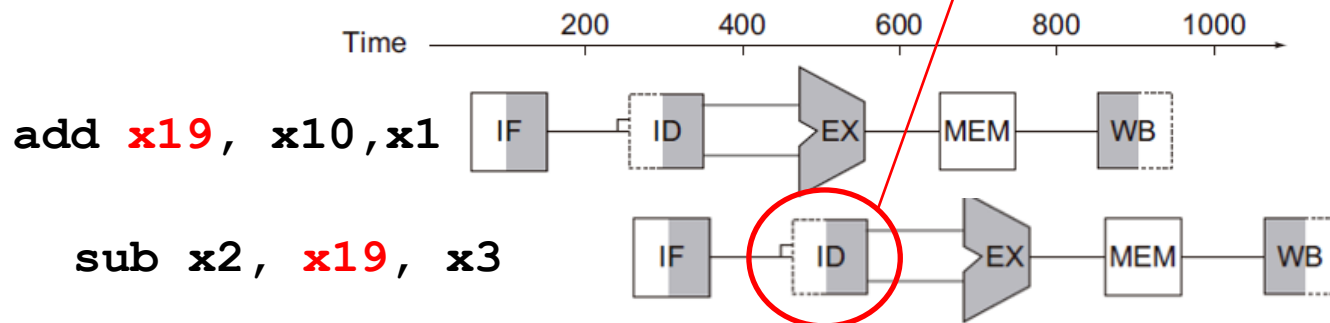
Gostaríamos que
estivesse dessa
forma
Mas, será que vai
estar?



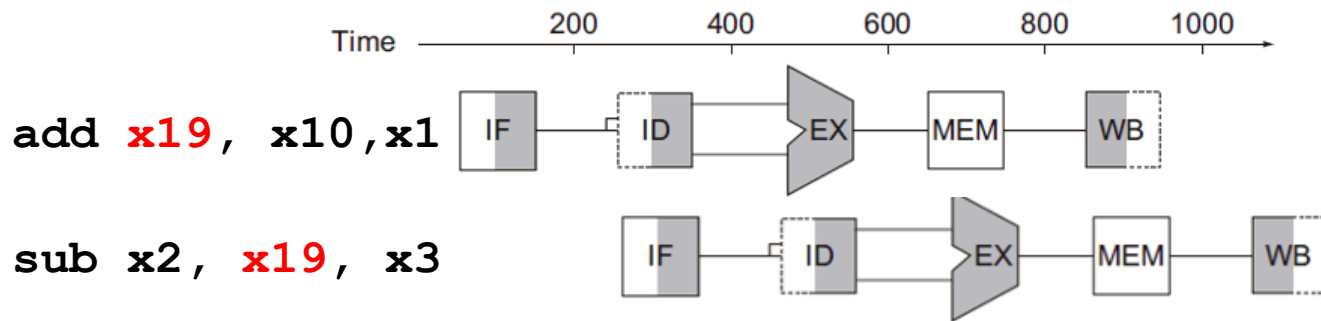
Conflito de dados

X0	0
X1	4
X2	2
X3	8
...	
X10	3
...	
X19	14
...	

No momento que a sub está lendo os registradores há o valor **14** ainda em x19 (e não o valor **7** desejado). Pois a add só escreverá seu resultado após o 5º estágio (WB)



Pipeline



X0	0
X1	4
X2	2
X3	8
...	
X10	3
...	
X19	7
...	

Banco de Registradores

Conflito de Dados

- A execução de uma instrução depende do resultado de outra que ainda está em execução
- Como resolver?

Soluções

(Conflito de dados)

1. Deixar que o compilador resolva (reorganizando a sequência de instruções)

- Mas não resolve totalmente pois ocorre com muita frequência:
 - Após a geração dos binários é necessário identificar as dependências e adiantar as instruções que são independentes
 - Nem sempre é possível, pois programas possuem muitas dependências de dados

Soluções

(Conflito de dados)

2. Inserir bolhas

- Há situações em que as soluções anteriores não resolvem, portanto, insere-se um “nop” entre instruções dependentes
- Acarreta retardo

Devido às bolhas: *“Em 80% dos ciclos de clock entrega-se uma instrução pronta no pipeline”*

3. Adiatamento de resultados

- Ou seja, alterar a via de dados
- Detectar o conflito e então adiantar o resultado


Conflitos

(de dados)

Solução 1: Deixar que o compilador resolva (reorganizando a sequencia de instruções)

a = b + e;

c = b + f;




```
lw      x1, 0(x31)    // Load b
lw      x2, 4(x31)    // Load e
add     x3, x1, x2     // b + e
sw      x3, 24(x31)   // Store a
lw      x4, 8(x31)    // Load f
add     x5, x1, x4     // b + f
sw      x5, 32(x31)   // Store c
```

Conflitos


(de dados)

Solução 1: Deixar que o compilador resolva (reorganizando a sequencia de instruções)

a = b + e;
c = b + f;



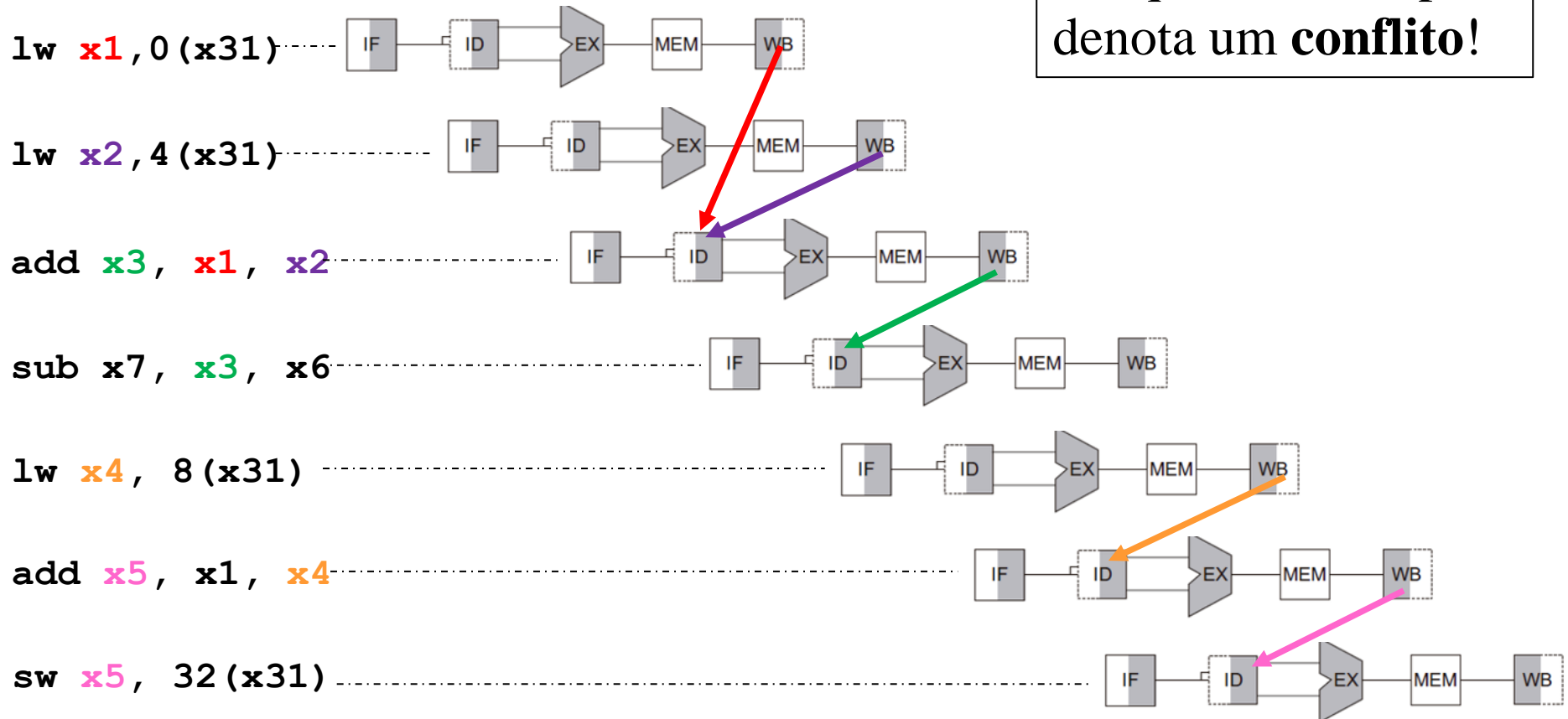
```
lw    x1, 0(x31)    // Load b
lw    x2, 4(x31)    // Load e
add   x3, x1, x2     // b + e
sw    x3, 24(x31)    // Store a
lw    x4, 8(x31)     // Load f
add   x5, x1, x4     // b + f
sw    x5, 32(x31)    // Store c
```



```
lw    x1, 0(x31)
lw    x2, 4(x31)
lw    x4, 8(x31)
add   x3, x1, x2
sw    x3, 12(x31)
add   x5, x1, x4
sw    x5, 16(x31)
```

Conflitos

Setas da “direita para a esquerda”: sempre denota um **conflito**!

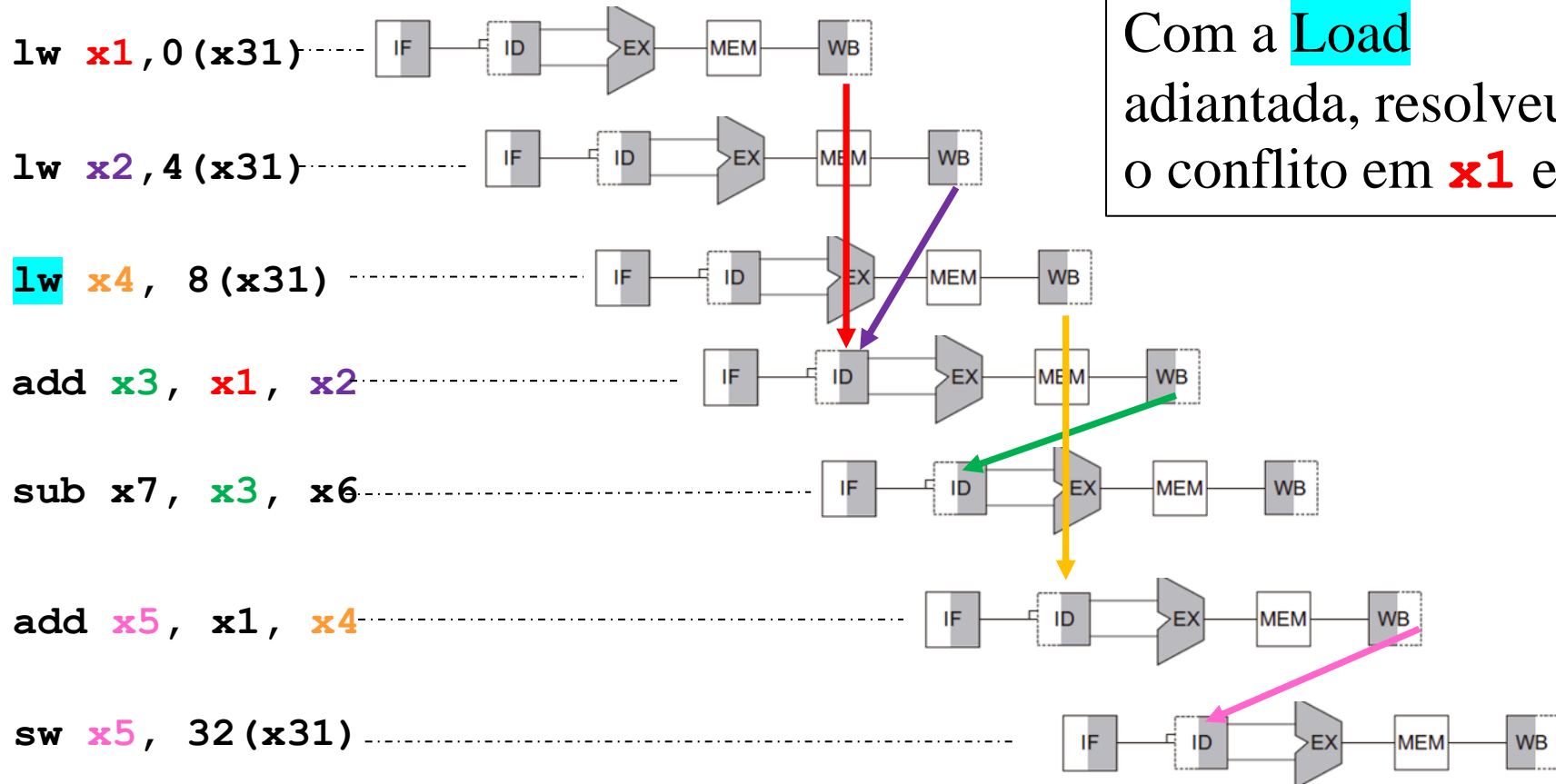


Solução 1

Solução 1: Reorganizar as instruções

Setas da “retas” ou “da esquerda para a direita”: **conflito resolvido!**

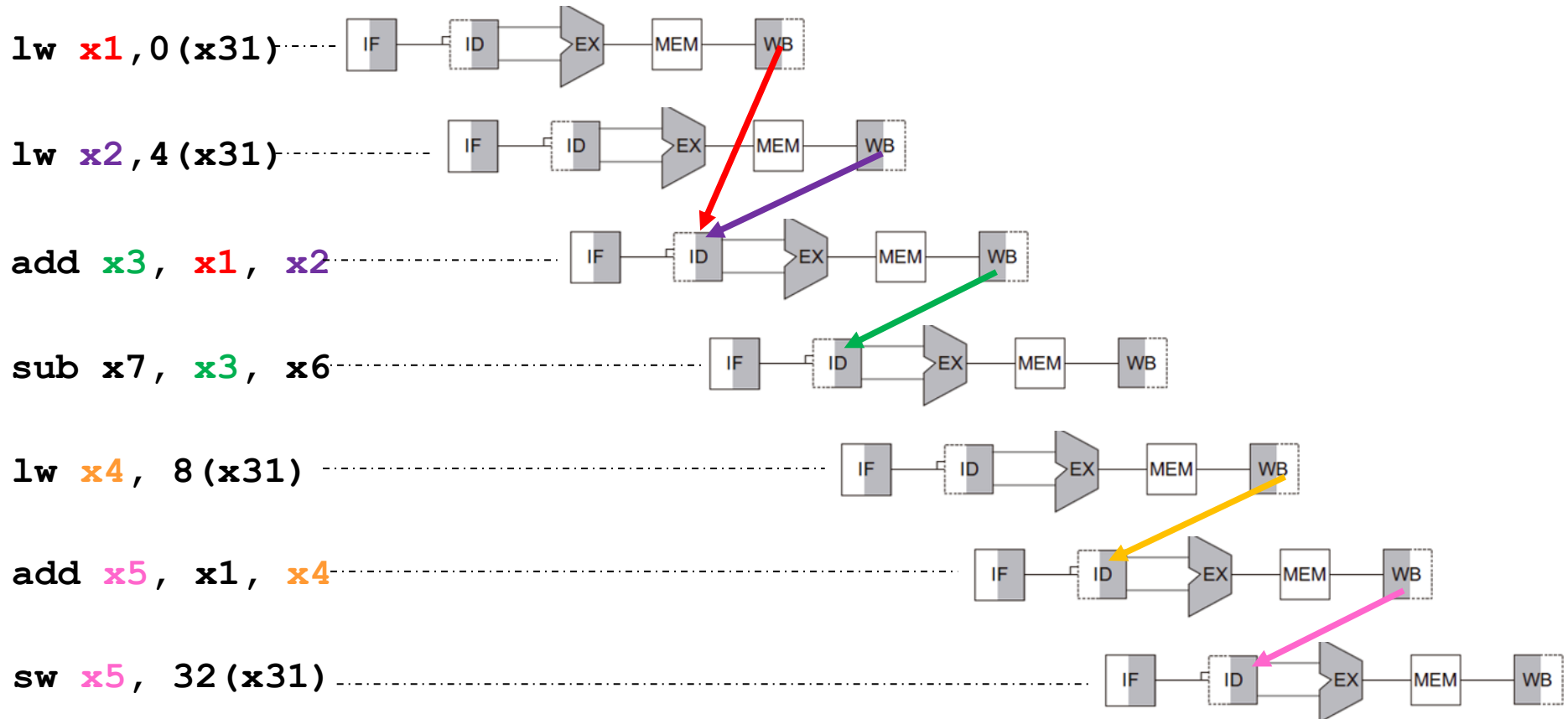
Com a **Load** adiantada, resolveu-se o conflito em **x1** e **x4**



Solução 1

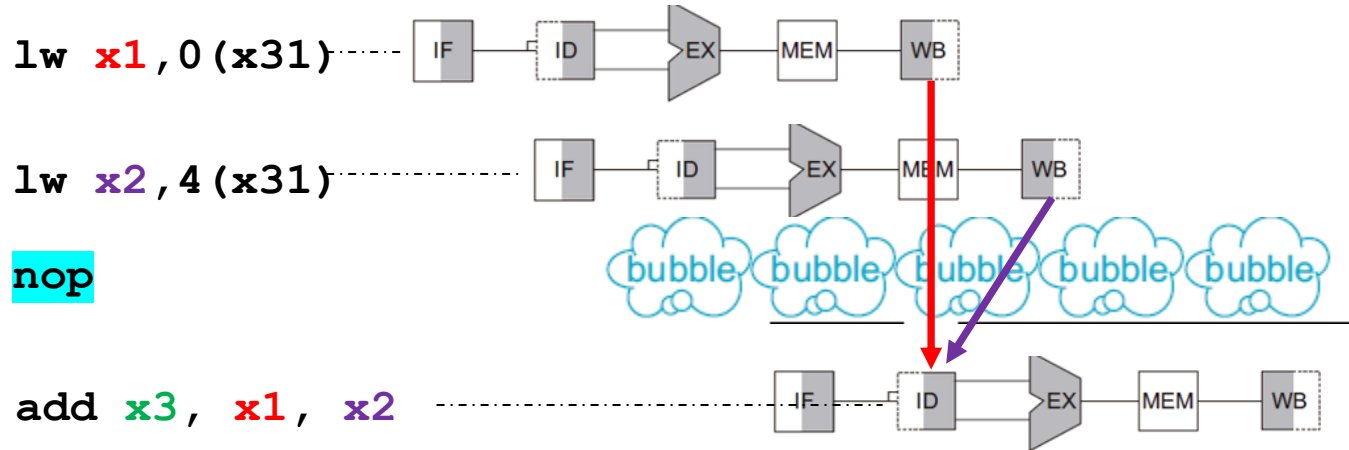
“Embora o compilador geralmente dependa do hardware para resolver conflitos e, assim, garantir a execução correta, o compilador deve compreender o *pipeline* para obter o melhor desempenho. Caso contrário, paralisações inesperadas reduzirão o desempenho do código compilado”

Conflitos



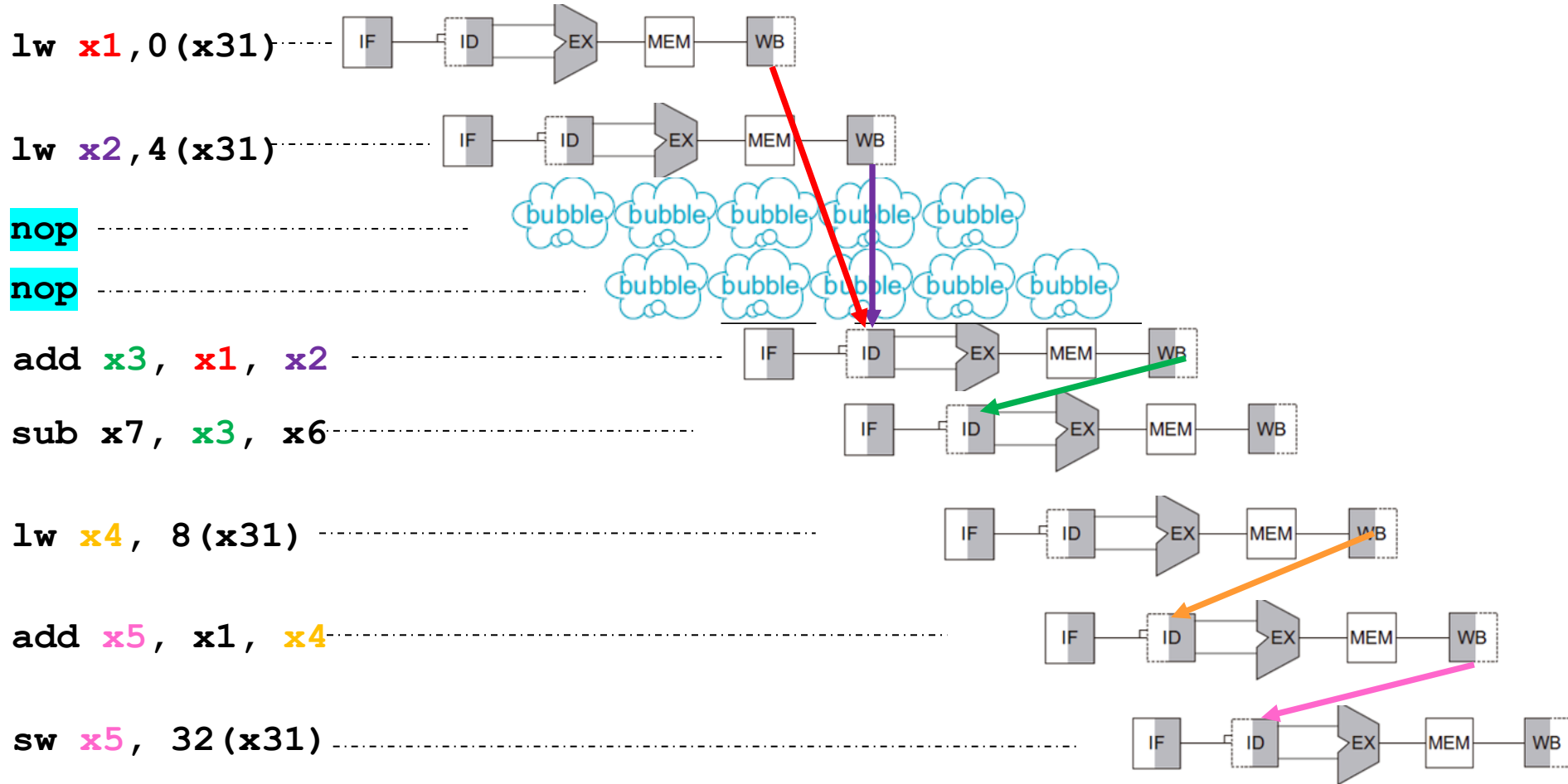
Solução 2

Solução 2: Inserir bolhas



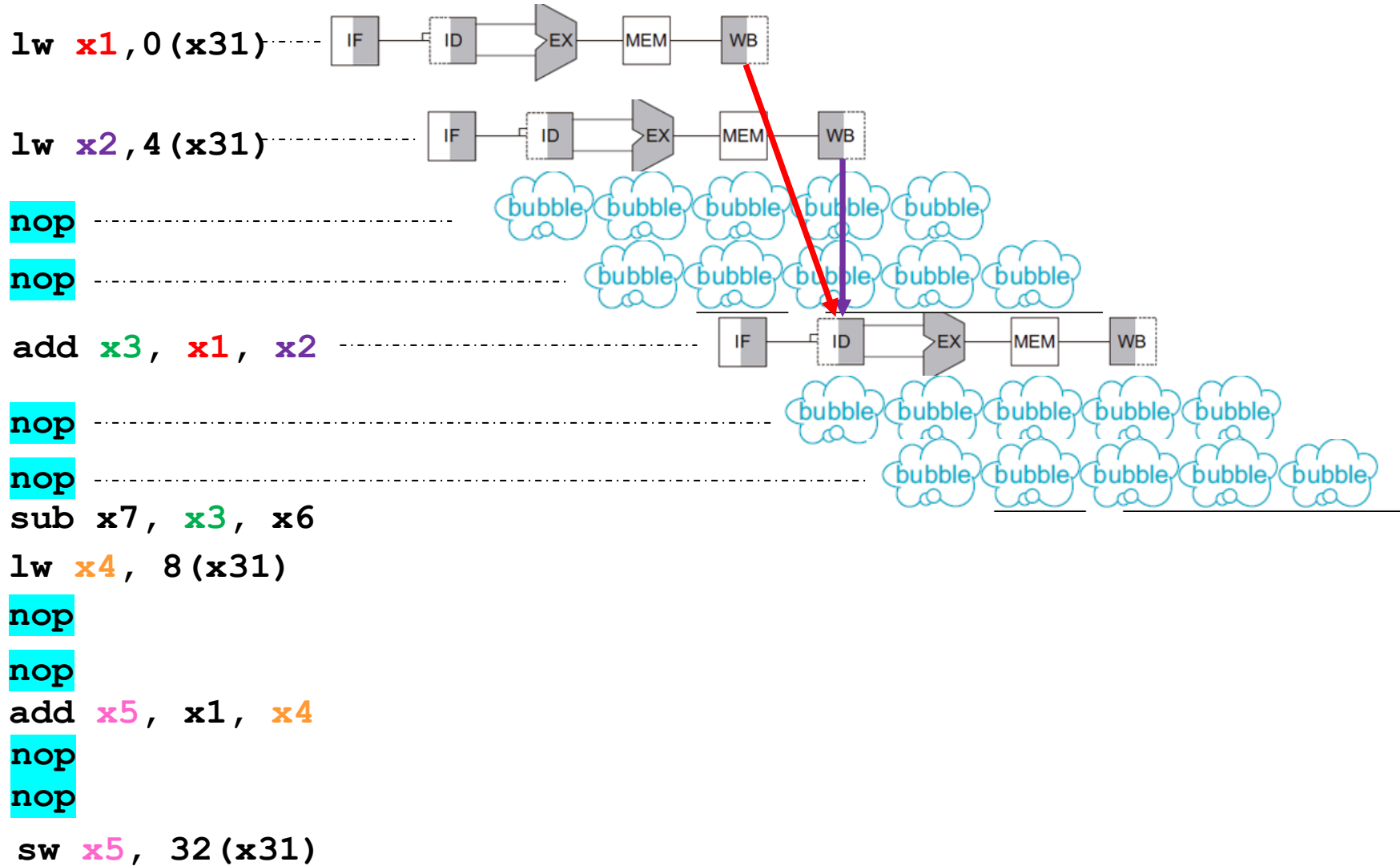
Solução 2

Solução 2: Inserir bolhas



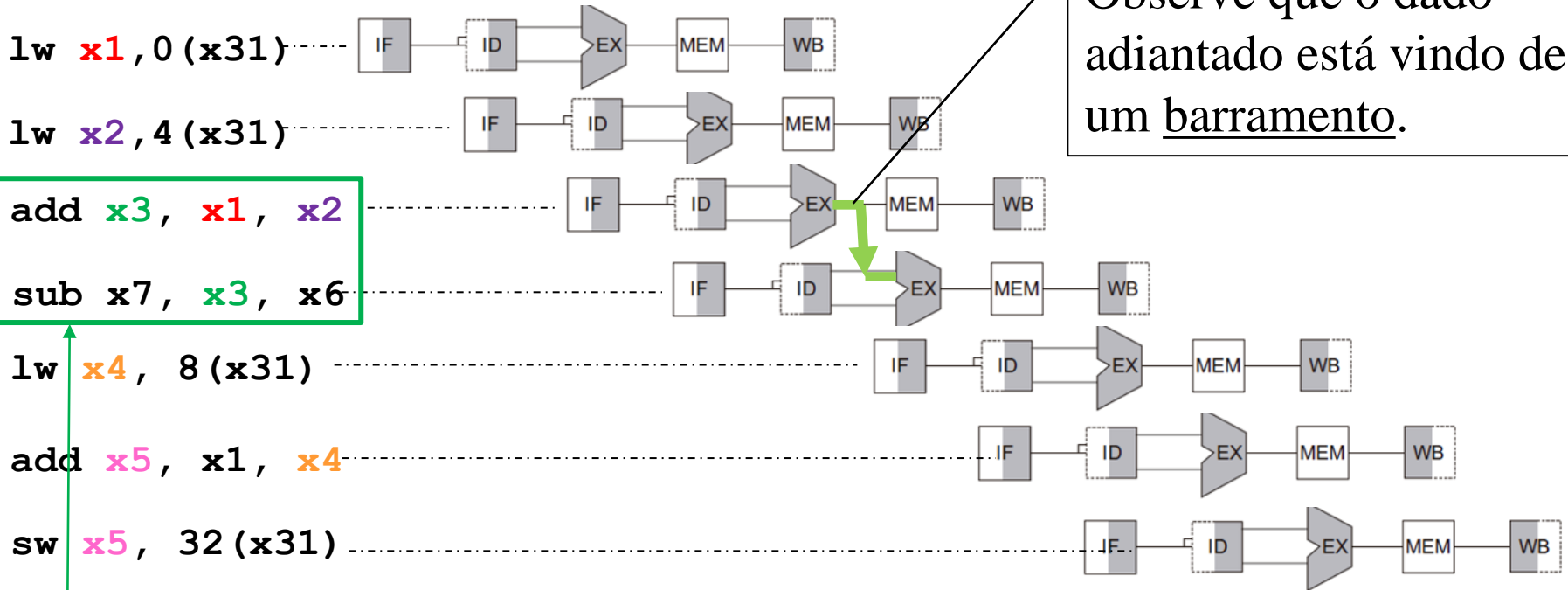
Solução 2

Solução 2: Inserir bolhas



Solução 3

Solução 3: Adiantamento de resultados



Saída da ULA sendo adiantada para a entrada da ULA.

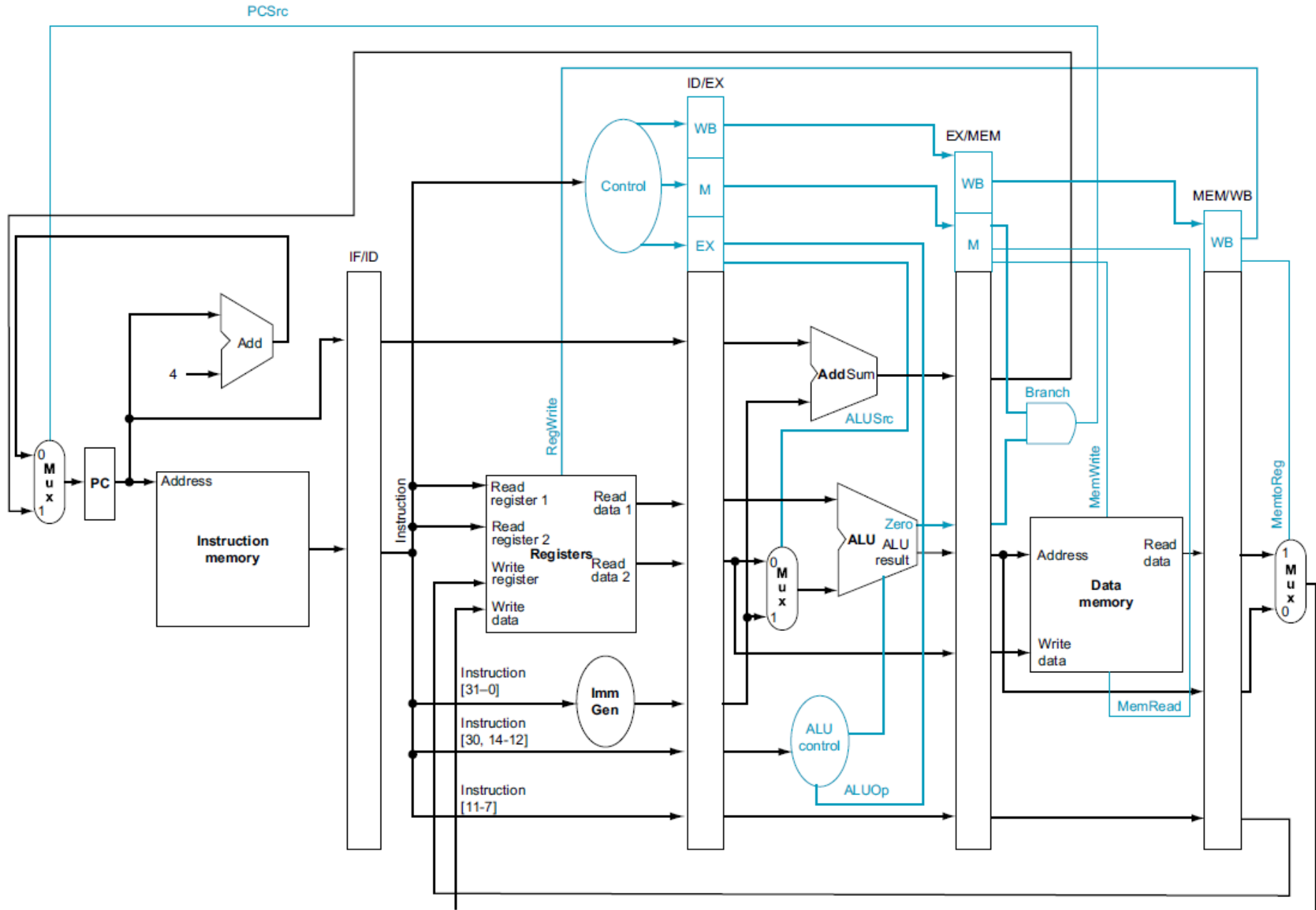
Observe que o dado adiantado está vindo de um barramento.

Vamos analisar esta situação

Como esse adiantamento de dados
ocorre em hardware?

Devem ser inseridos **unidades de
adiantamento**

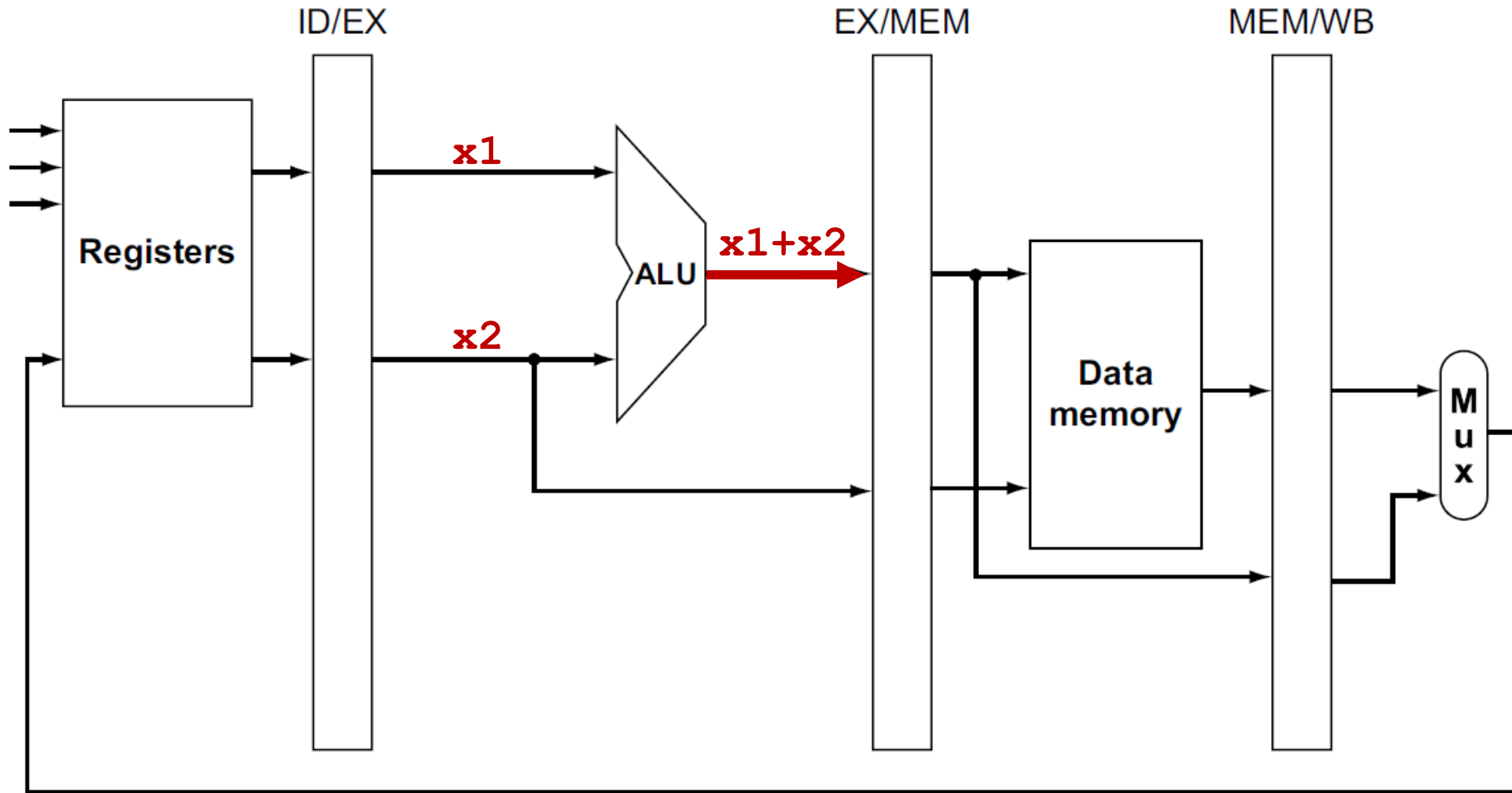
Pipeline sem unidades de adiamento



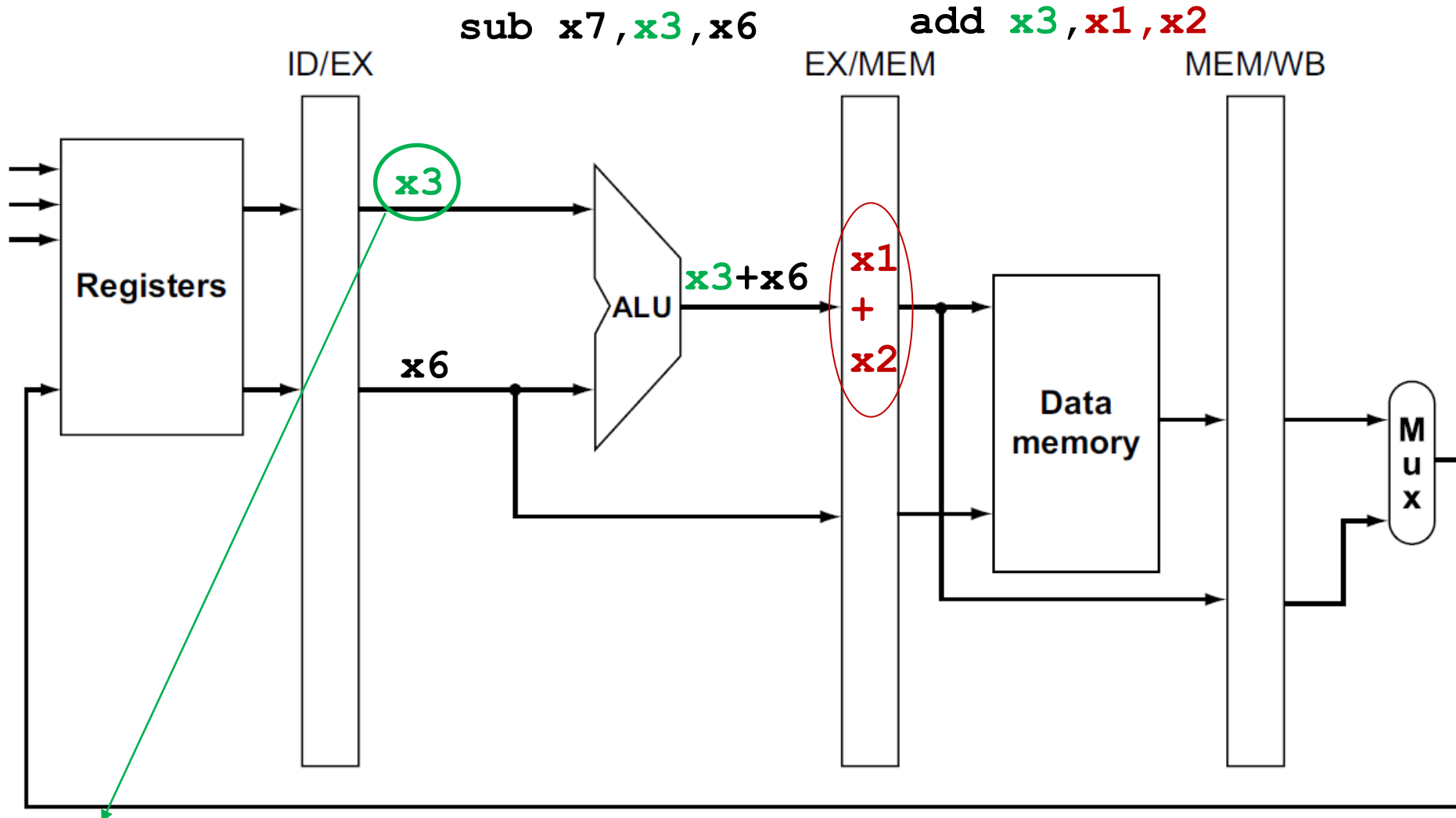
Na saída da ALU já se encontra a informação desejada (**x1+x2**)

sub x7, **x3**, x6

add **x3**, **x1**, **x2**

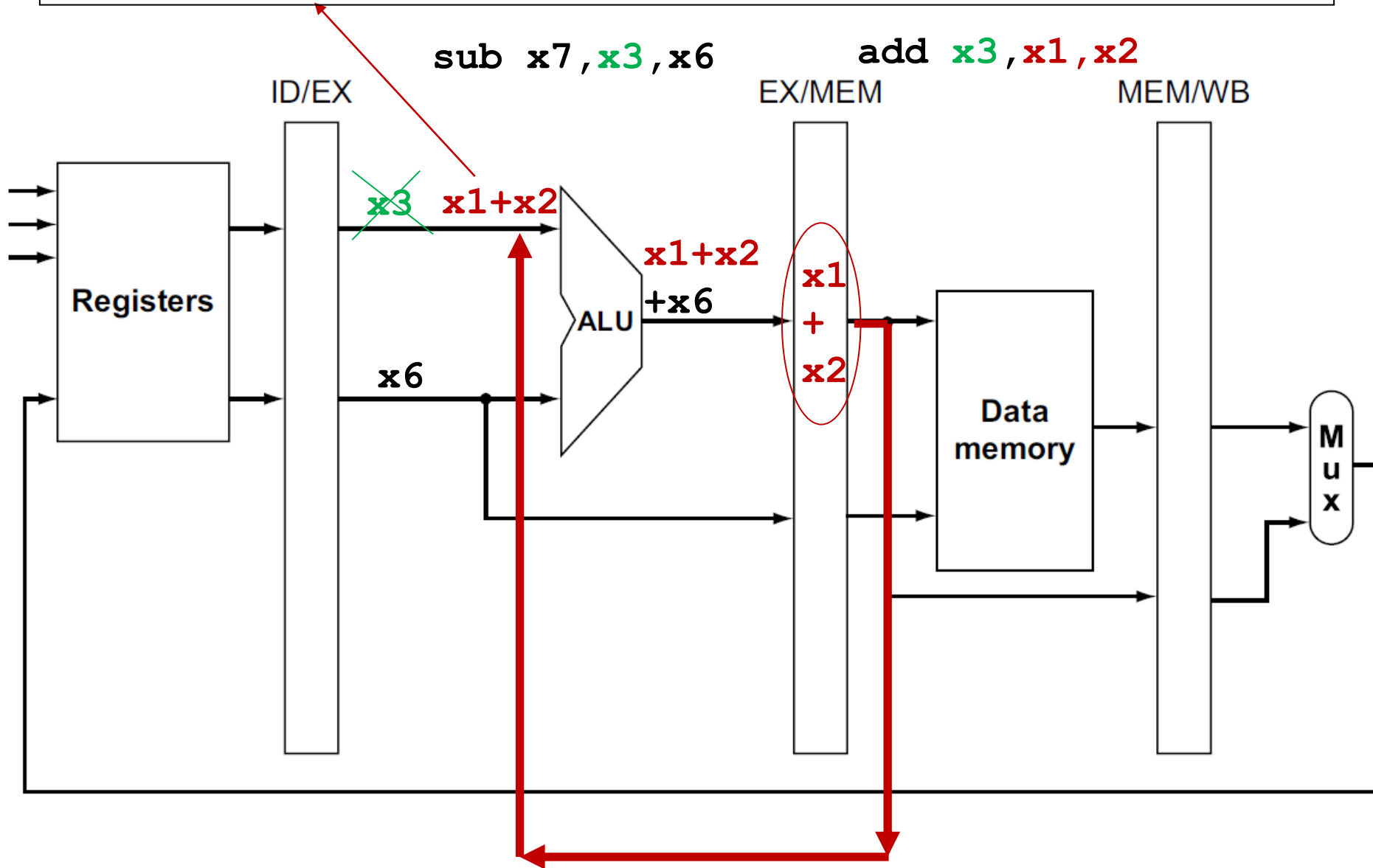


Portanto, ao final do ciclo, em EX/MEM se encontra a informação a ser inserida em **x3**, ou seja: **x1+x2**



No entanto, o valor de **x3** que **sub** está utilizando ainda não é o resultado de **x1+x2** produzido pela **add**, e sim um valor antigo vindo do banco de registradores!

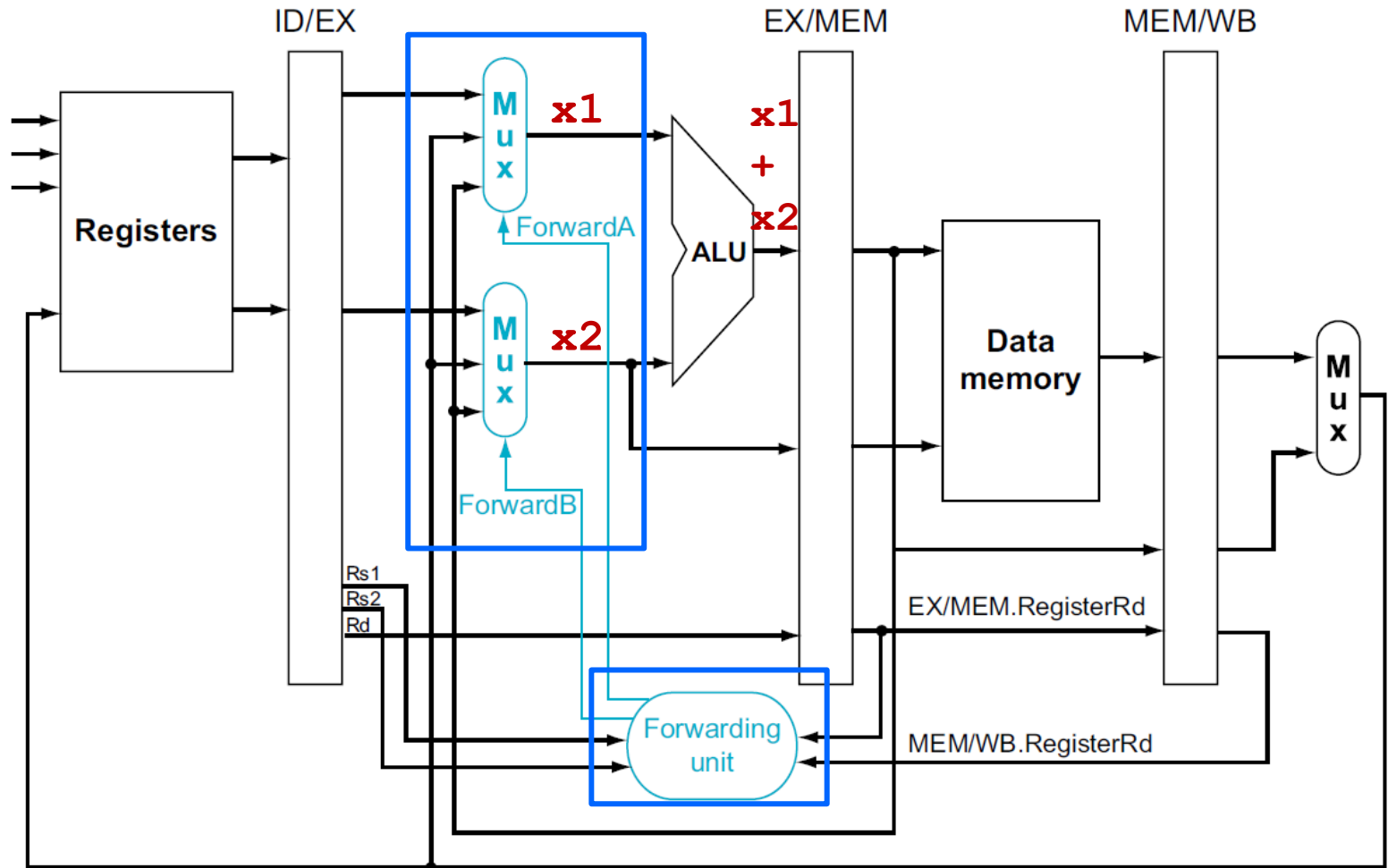
Mas, ao invés disso, faremos com que o resultado de $x1+x2$ é que seja o 1º operando da ULA. **Para isso, será necessário adiantar essa informação.**



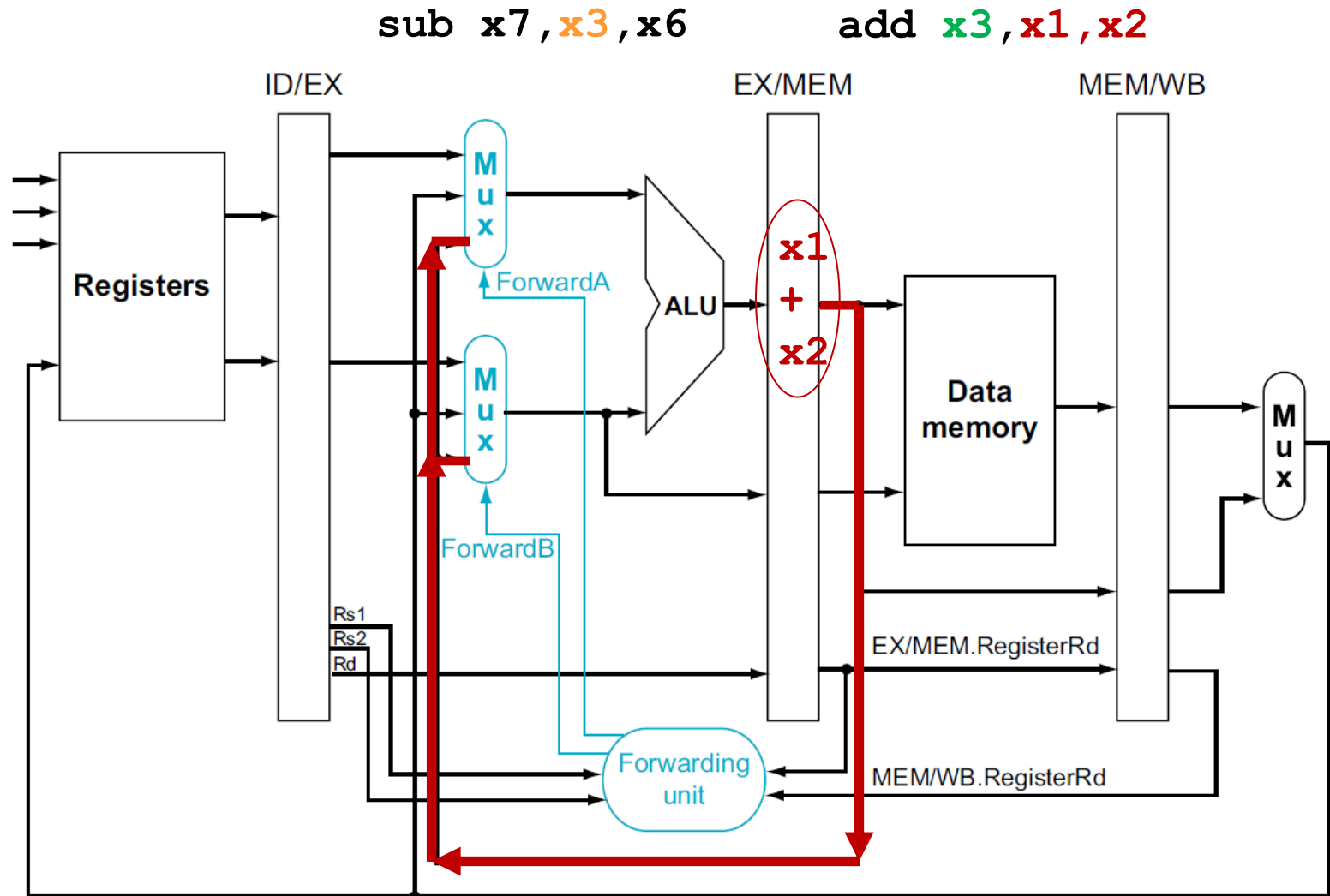
Para isso ser possível é necessário inserir uma **Unidade de Adiantamento** no 3º estágio, que recebe algumas entradas e gera **sinais de controle** como saída.

sub x7, x3, x6

add x3, x1, x2



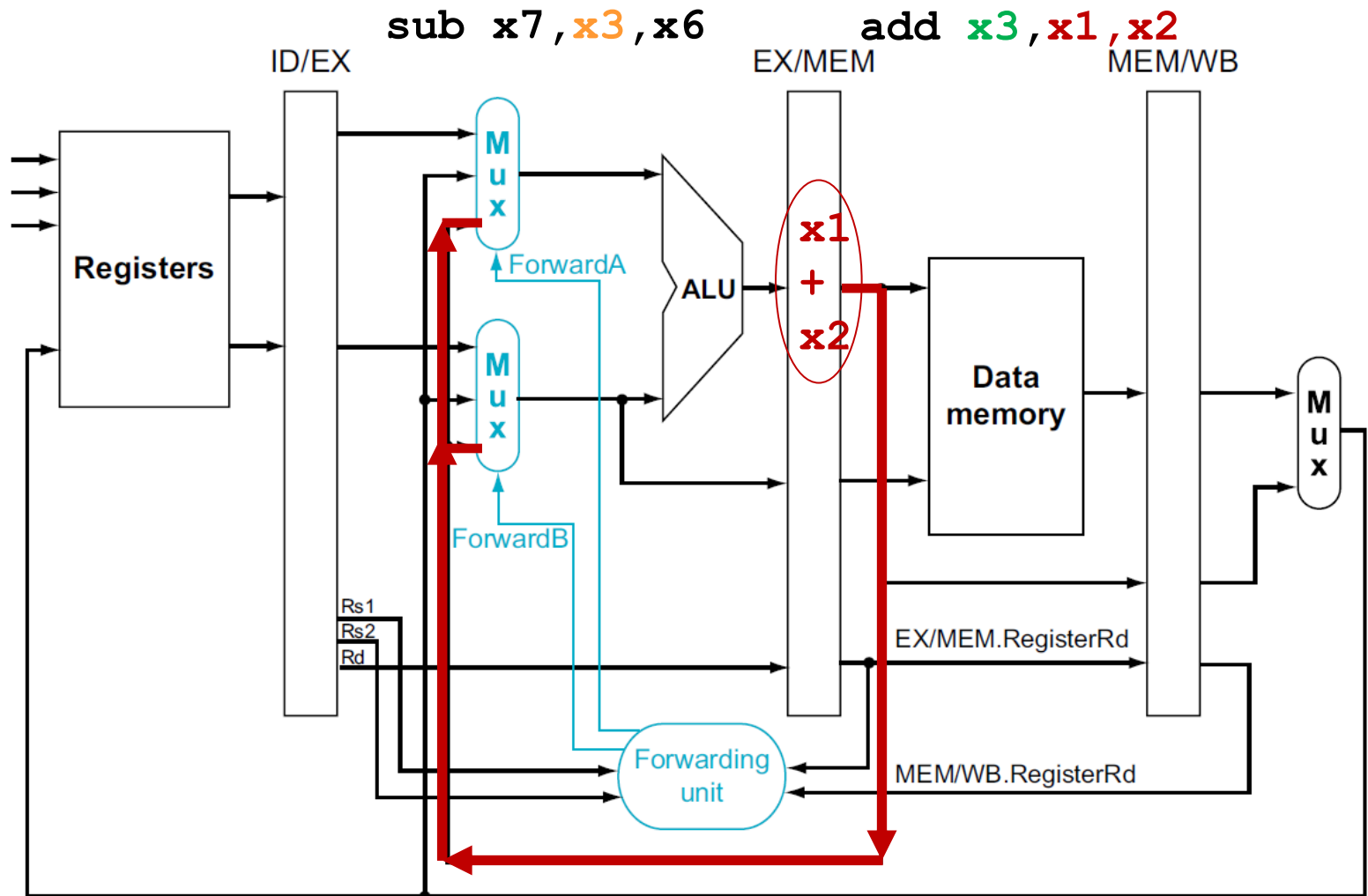
Como esta Unidade está implementada de forma que seja possível realizar o adiantamento sinalizado abaixo?



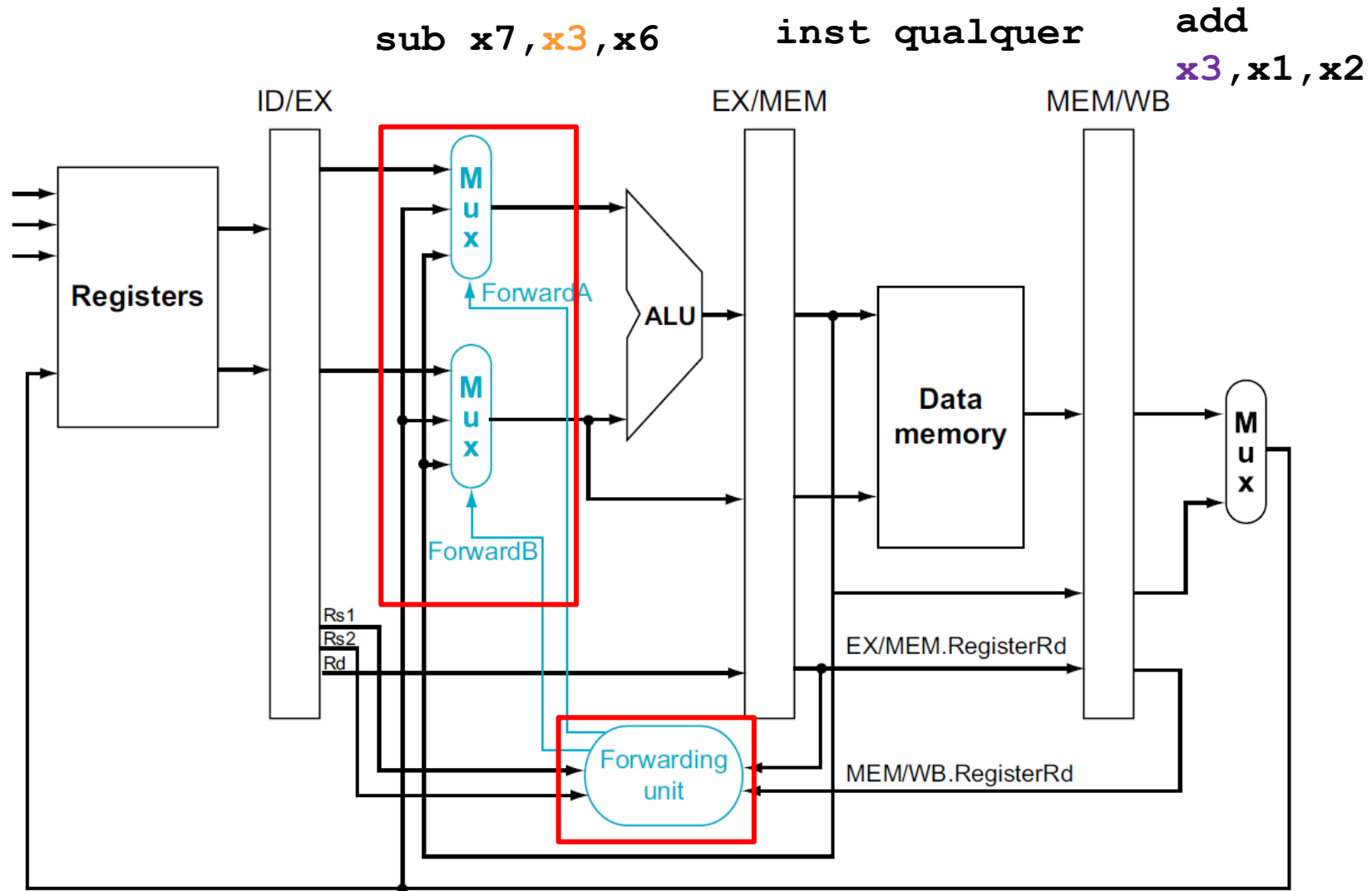
```

if ( EX/MEM.RegWrite
    and (EX/MEM.RegisterRd ≠ 0)
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRs1) )   ForwardA = 10
if ( EX/MEM.RegWrite
    and (EX/MEM.RegisterRd ≠ 0)
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRs2) )   ForwardB = 10

```



E se a dependência entre as instruções fosse a uma distância de 2?



Somente os `if`'s anteriores não adiantariam!

Forwarding Unit

- Conflito no estágio **EX**:

```
if ( (EX/MEM.RegWrite)
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1) ) ForwardA = 10
```

```
if ( (EX/MEM.RegWrite)
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2) ) ForwardA = 10
```

- Conflito no estágio **MEM**:

```
if( (MEM/WB.RegWrite)
    and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

```
if( (MEM/WB.RegWrite)
    and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```



Forwarding Unit

- E se tivéssemos a seguinte sequência de instruções abaixo, o que ocorreria na unidade de adiantamento conforme a lógica anterior?

```
add x1, x2, x3
add x1, x1, x2
add x1, x3, x1
```

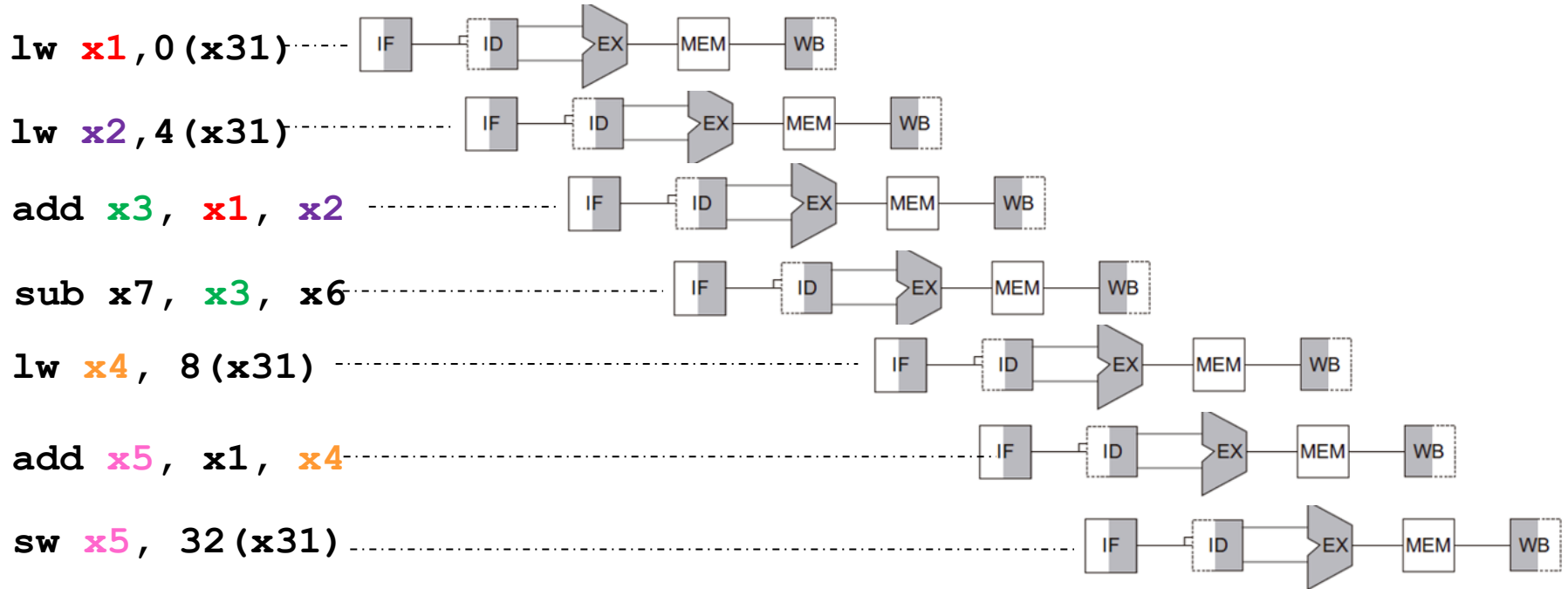
```
if( (MEM/WB.RegWrite)
    and (MEM/WB.RegisterRd ≠ 0)
    and not ( EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
              and (EX/MEM.RegisterRd = ID/EX.RegisterRs1) )
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

```
if( (MEM/WB.RegWrite)
    and (MEM/WB.RegisterRd ≠ 0)
    and not ( EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
              and (EX/MEM.RegisterRd = ID/EX.RegisterRs2) )
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Vamos voltar ao exemplo...

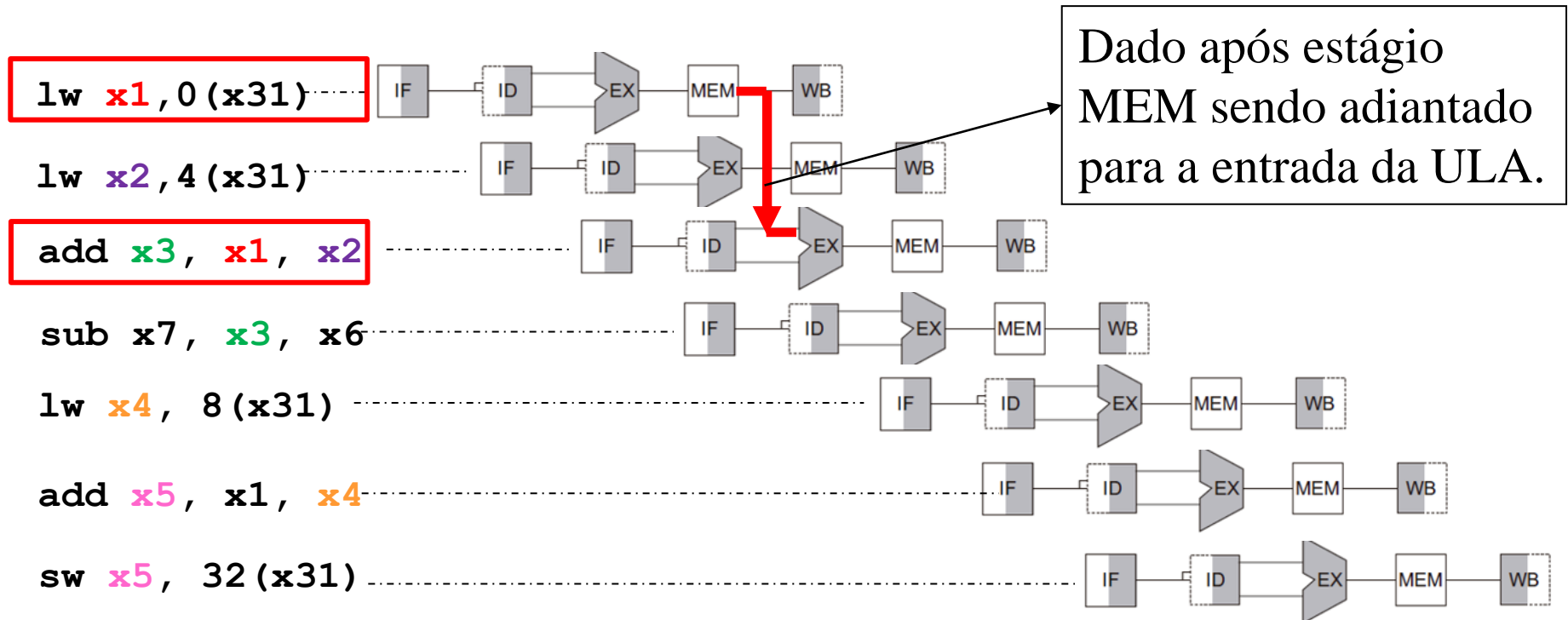
Solução 3

- Mais algum conflito resolvido com a unidade de adiantamento?



Solução 3

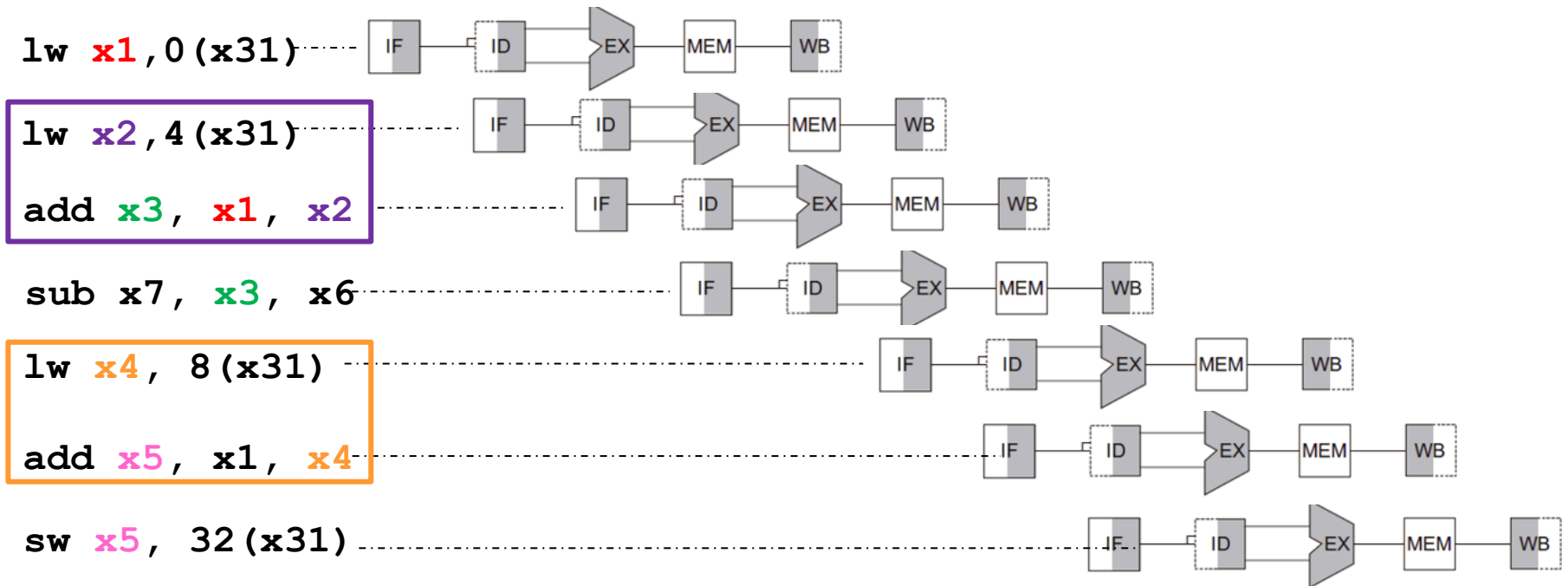
- Mais algum conflito resolvido com a unidade de adiantamento?



Observe que apesar de ser entre uma **load** e **tipo-R**, o comportamento da unidade de adiantamento será o mesmo que o exemplo anterior (entre duas **tipo-R**). Portanto, não importa se o dado adiantado veio da ULA ou da memória.

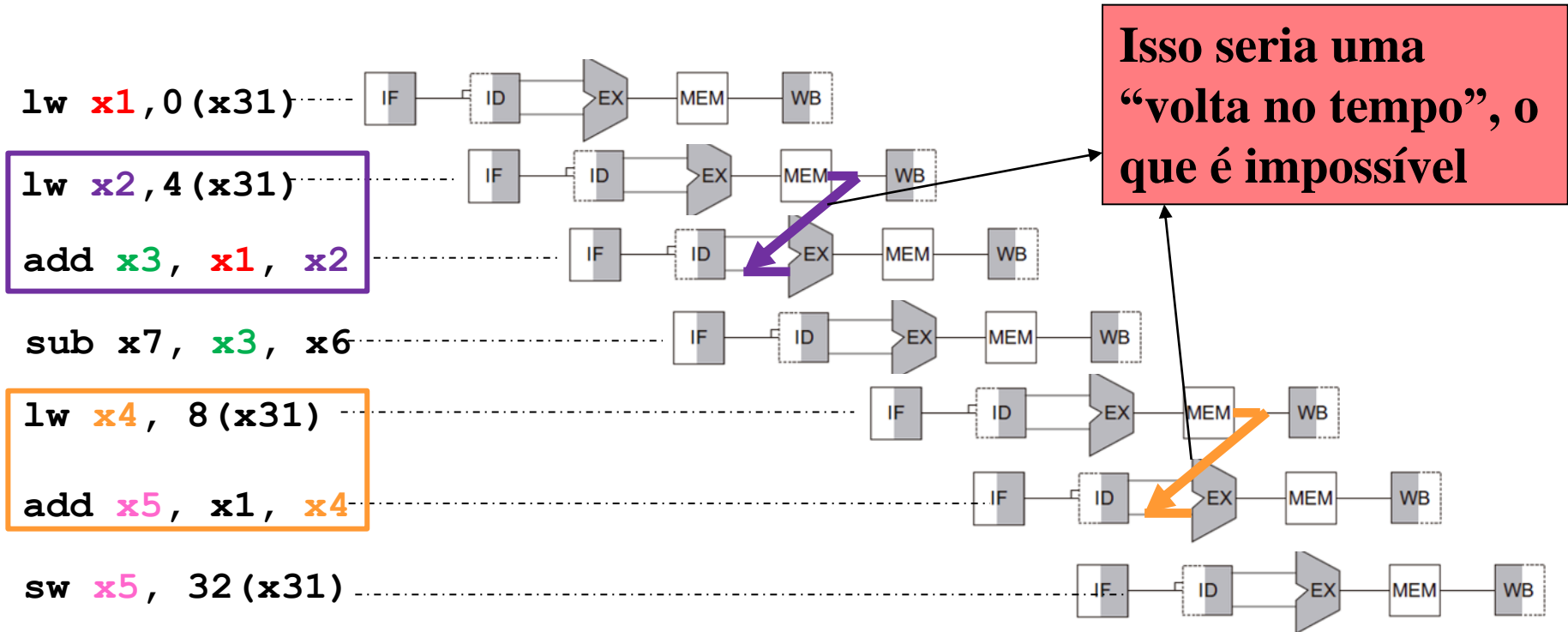
Solução 3

- Será que o mesmo ocorre entre estas instruções?



Solução 3

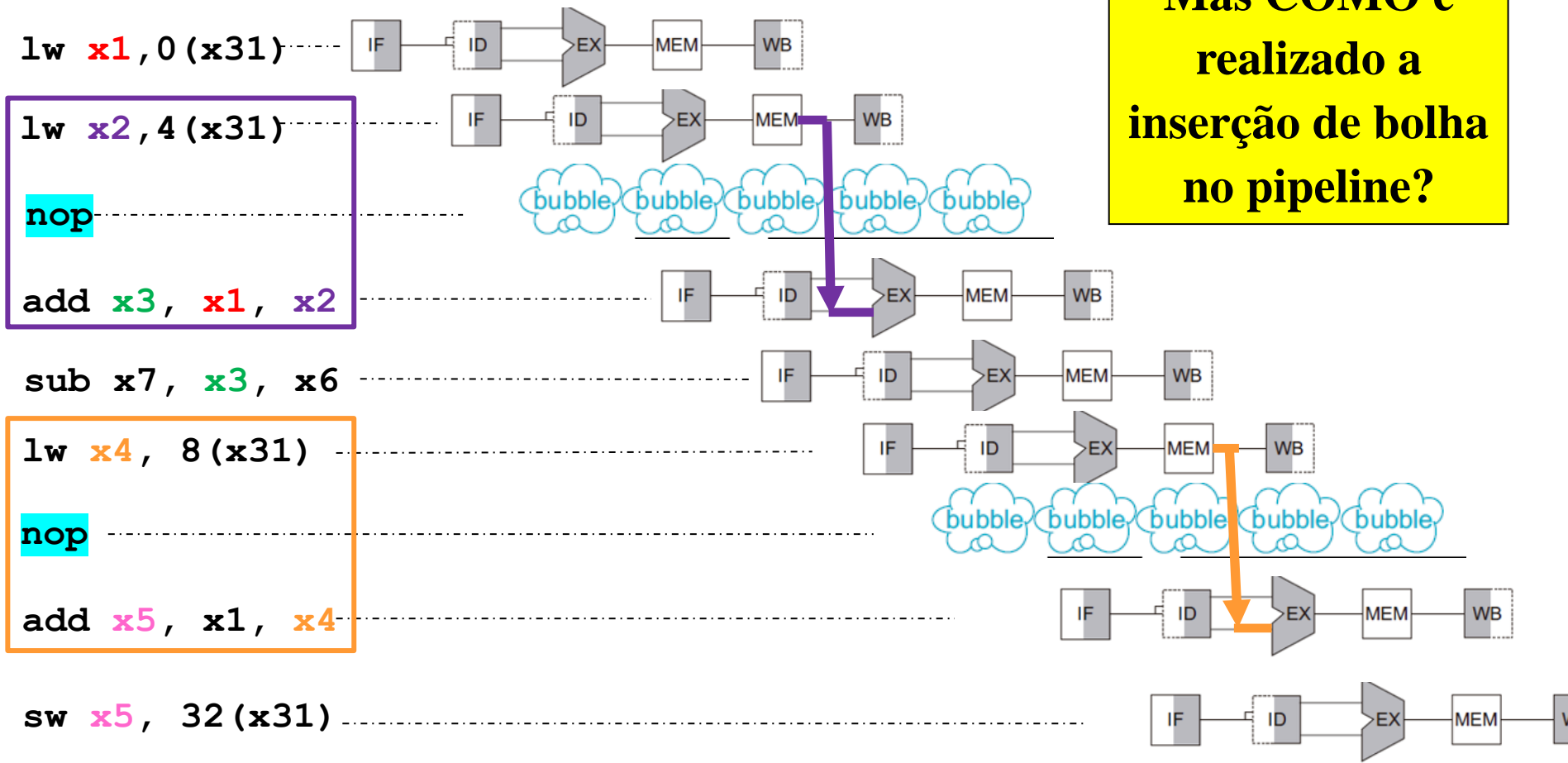
- Será que o mesmo ocorre entre estas instruções?



Como resolver?

Solução 3

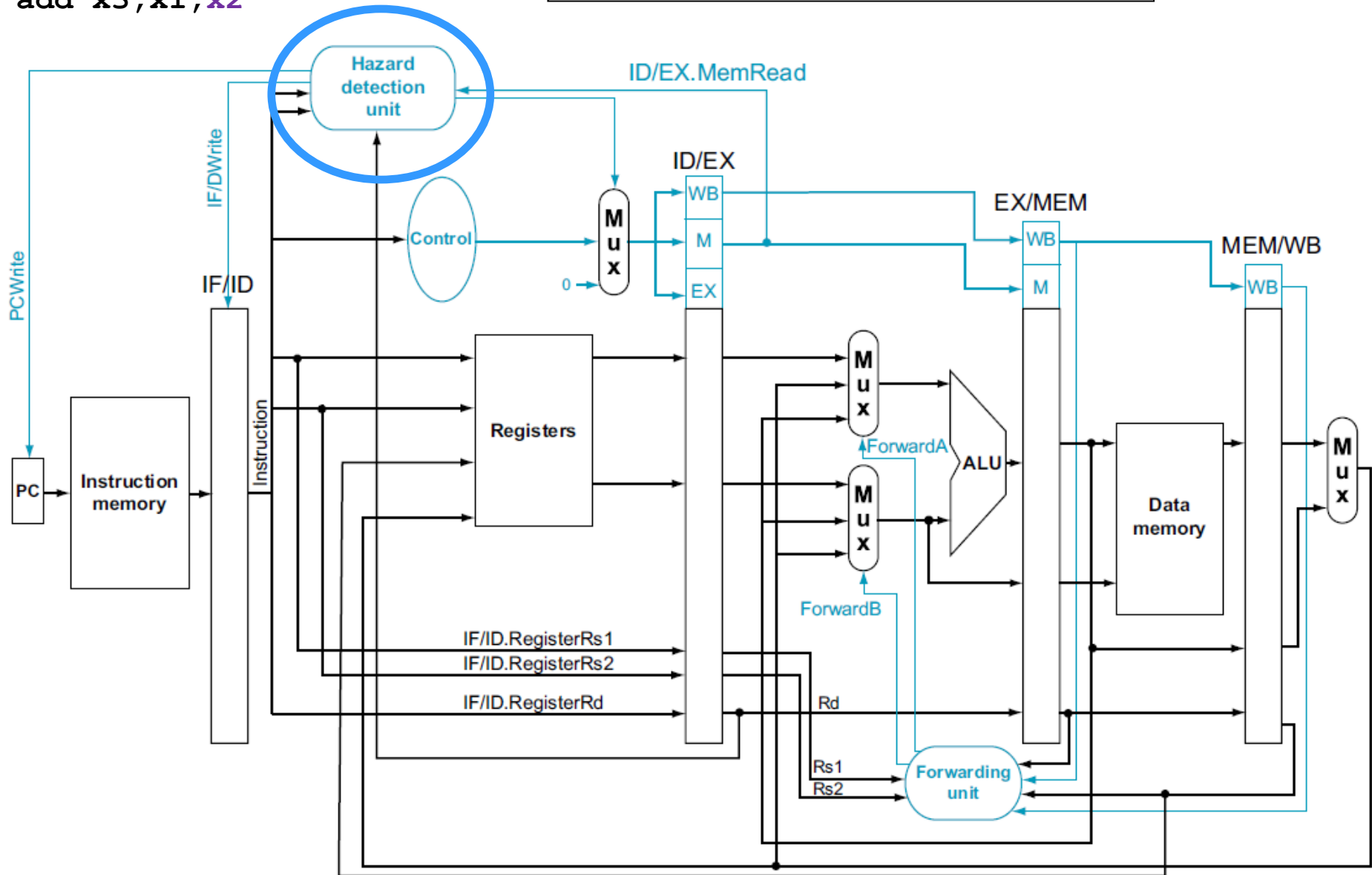
- Necessário a **inserção de bolha** conjuntamente com o **adiantamento de dados!**



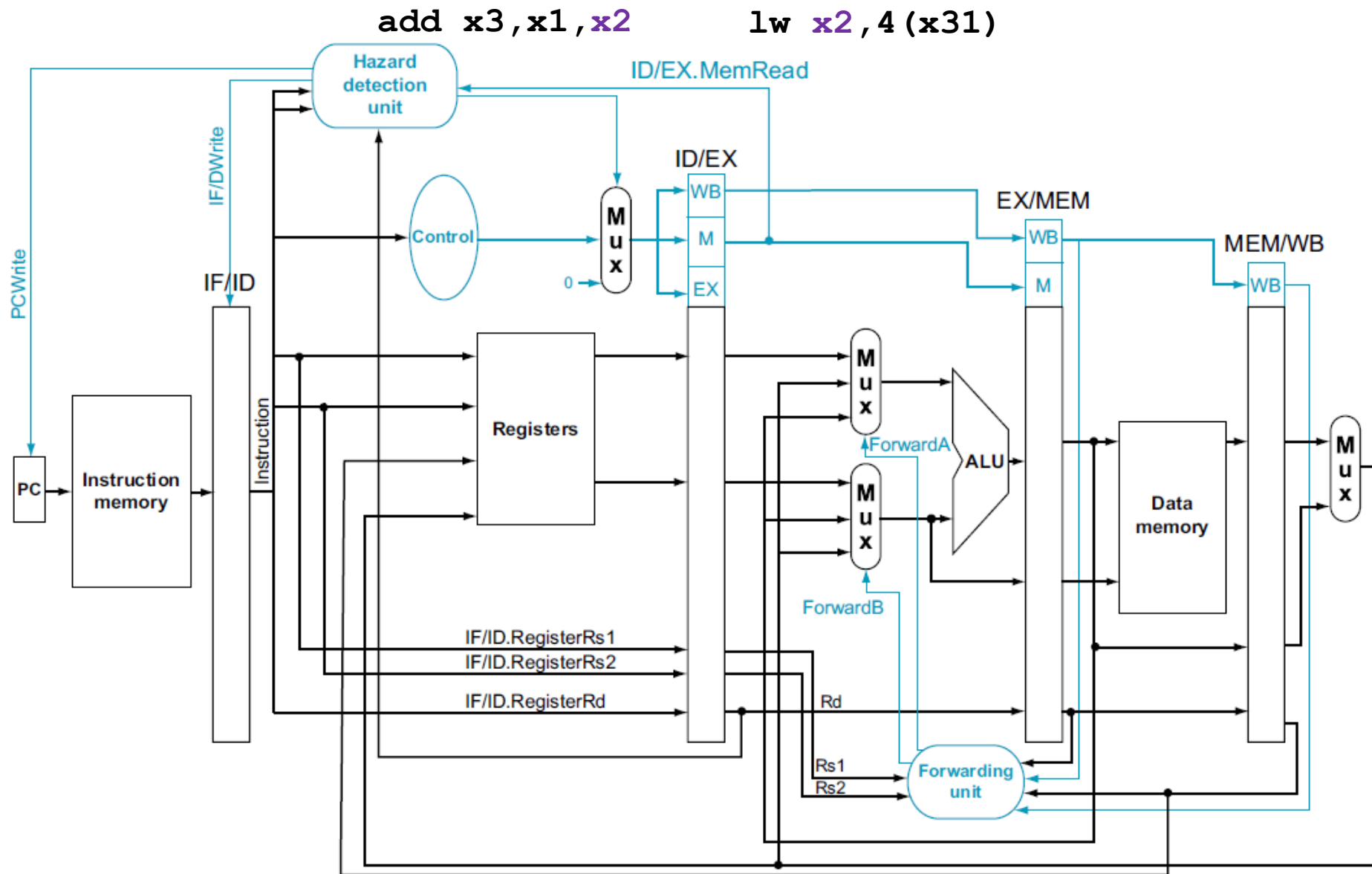
lw x2, 4(x31)

add x3, x1, x2

Necessário uma **Hazard Detection Unit** no 2º estágio



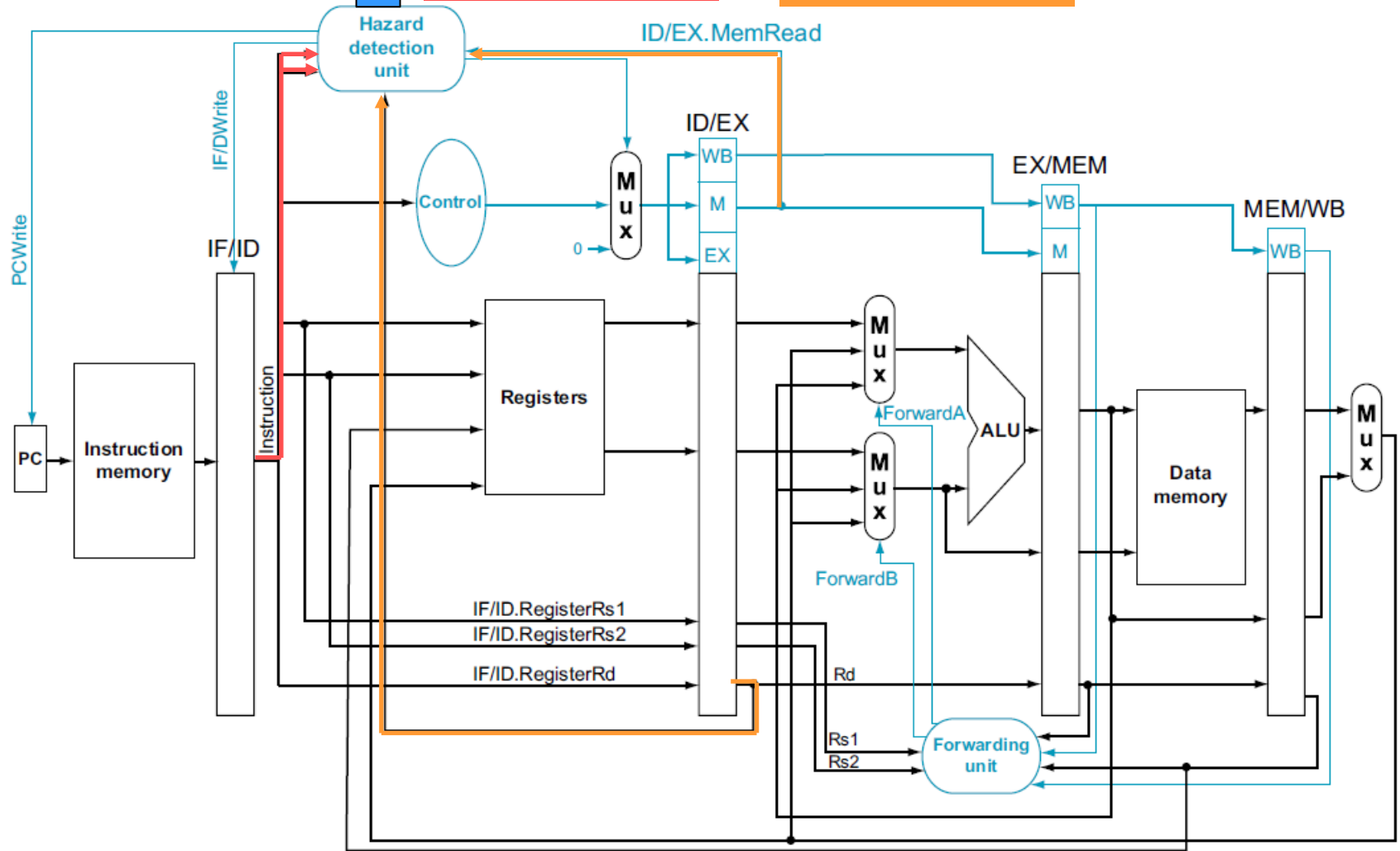
Se a instrução que está em EX é uma **load** e seu **registrador de destino** coincide com algum de **origem** da instrução no estágio anterior: **STALL!**



if(ID/EX.MemRead and
 ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
 (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
 stall the pipeline

add x3,x1,x2

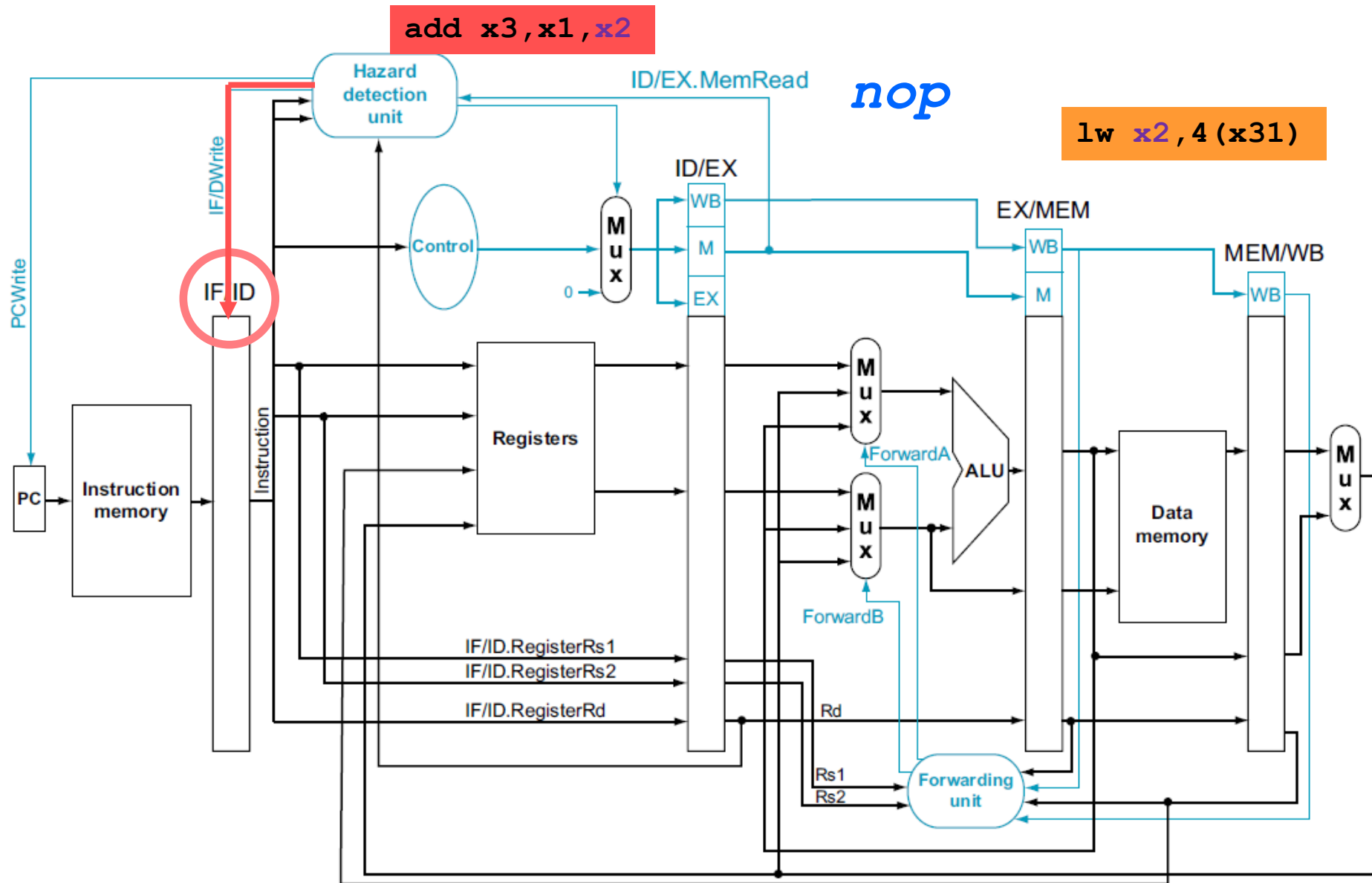
lw x2,4(x31)



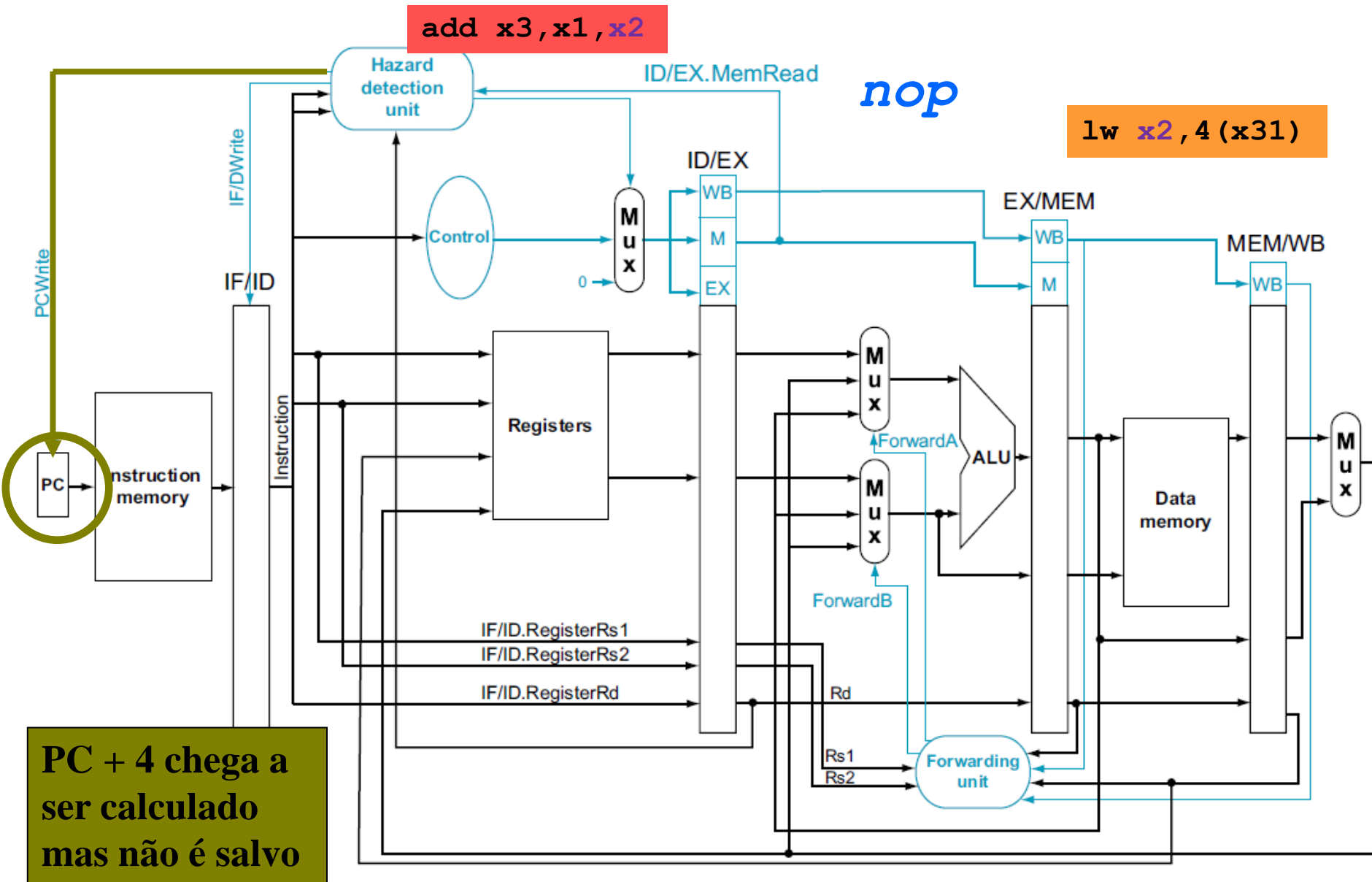
Stall

- O que o hardware faz para paralizar o pipeline por 1 ciclo?
 1. Não permite a alteração do registrador de pipe **IF/ID** (desabilita a escrita) – isso fará com que o estágio **ID** se repita, isto é, um *stall*
 2. Com isso, a instrução anterior (no estágio **IF**) precisa também de um *stall* para que o **hardware não altere o PC** (desabilita a escrita). **Estágio IF** será repetido.
 3. Alterações nos campos de controle em: “**EX**”, “**MEM**” e “**WB**” que se encontram no registrador de pipe **ID/EX** para **0**, para que a instrução que vem após a **load** se torne um **nop** – “*a bolha foi inserida no pipe*”

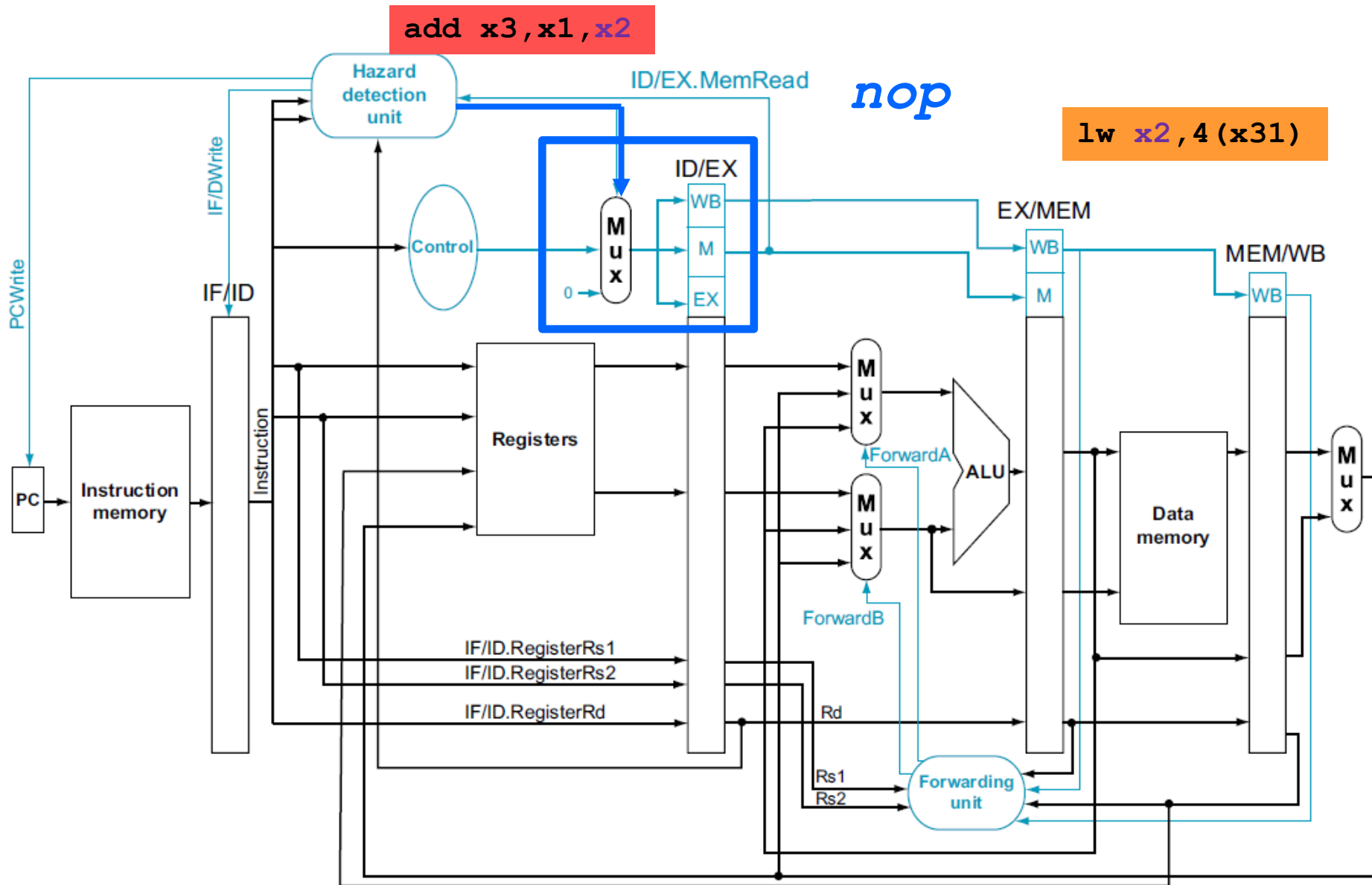
1. Travar **IF/ID** (**IF/IDWrite = 0**)



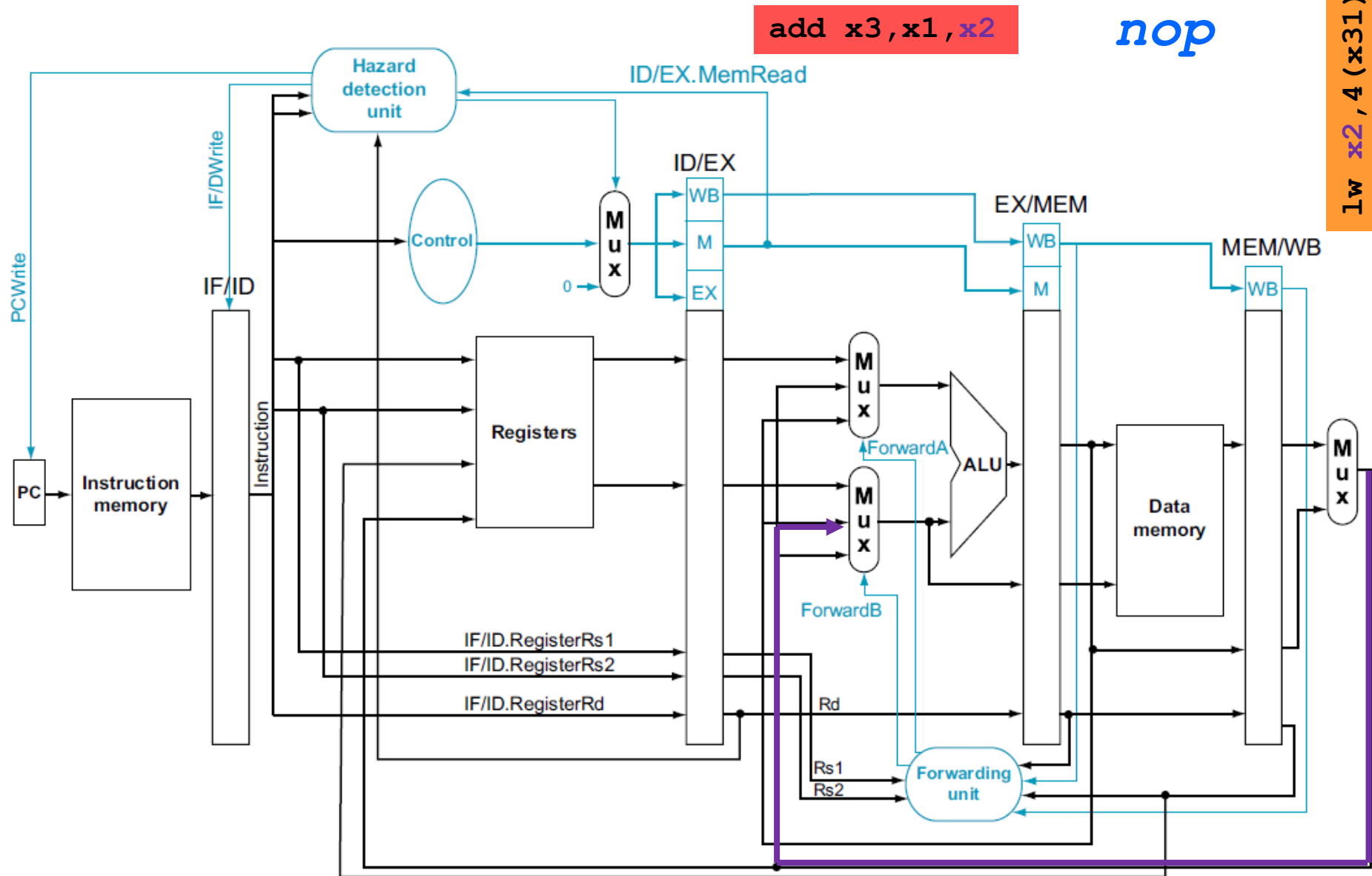
2. Travar **PC** (**PCWrite = 0**)



3. Emitir sinais de controle 0 no estágio ID



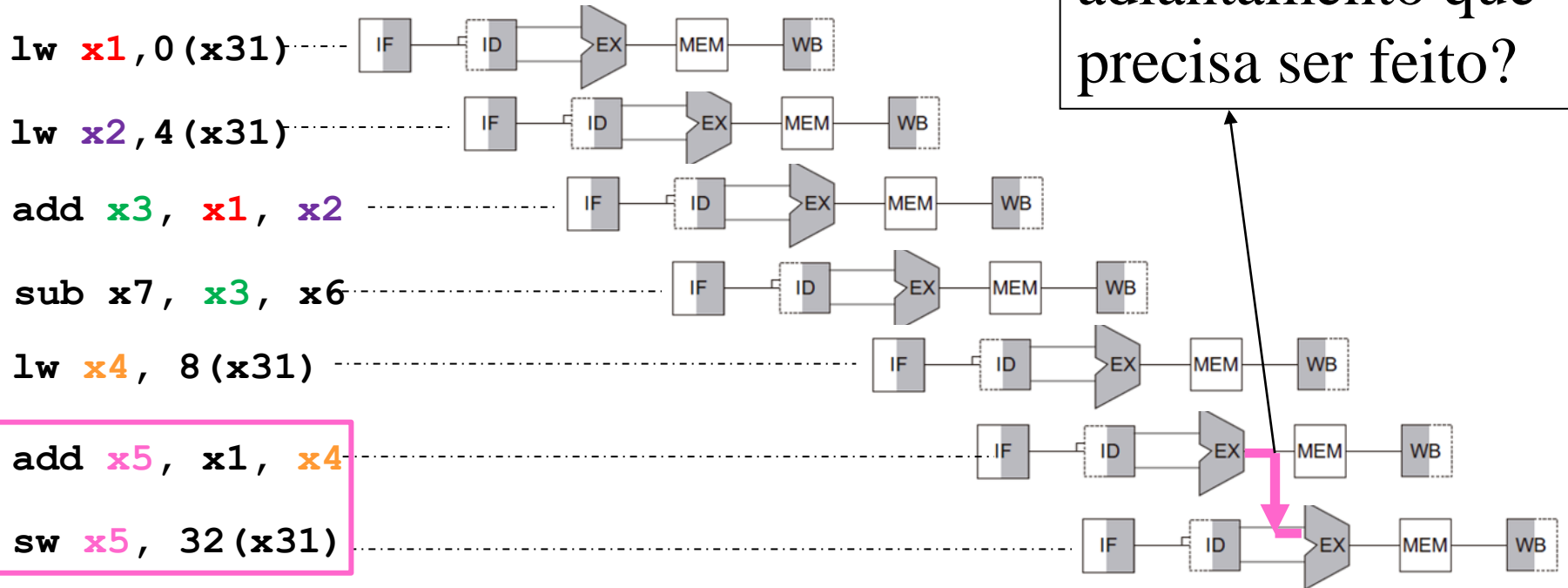
E neste ponto, quando a **load** estiver na etapa 5 e **add** na etapa 3: a **unidade de forwarding** fará o adiantamento de **x2**, conforme vimos anteriormente.



Voltando mais uma vez ao
exemplo...

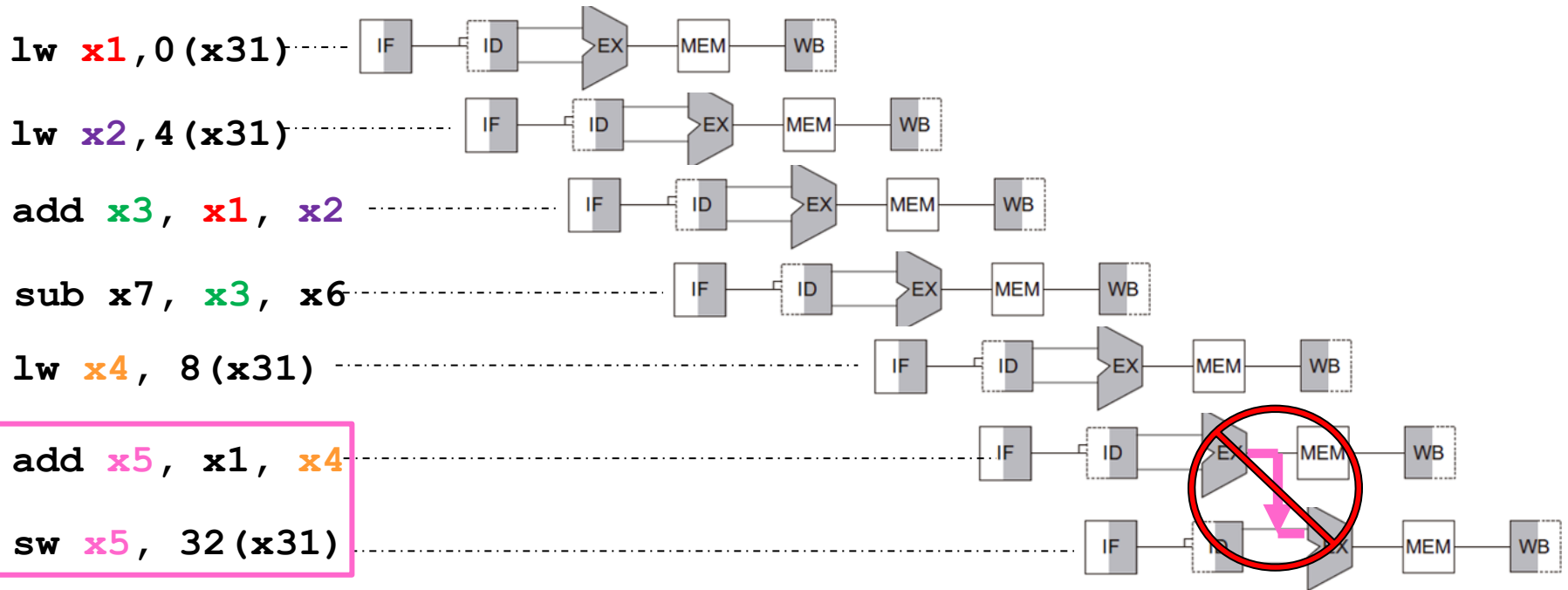
Solução 3

- E este conflito com a **store**?



Solução 3

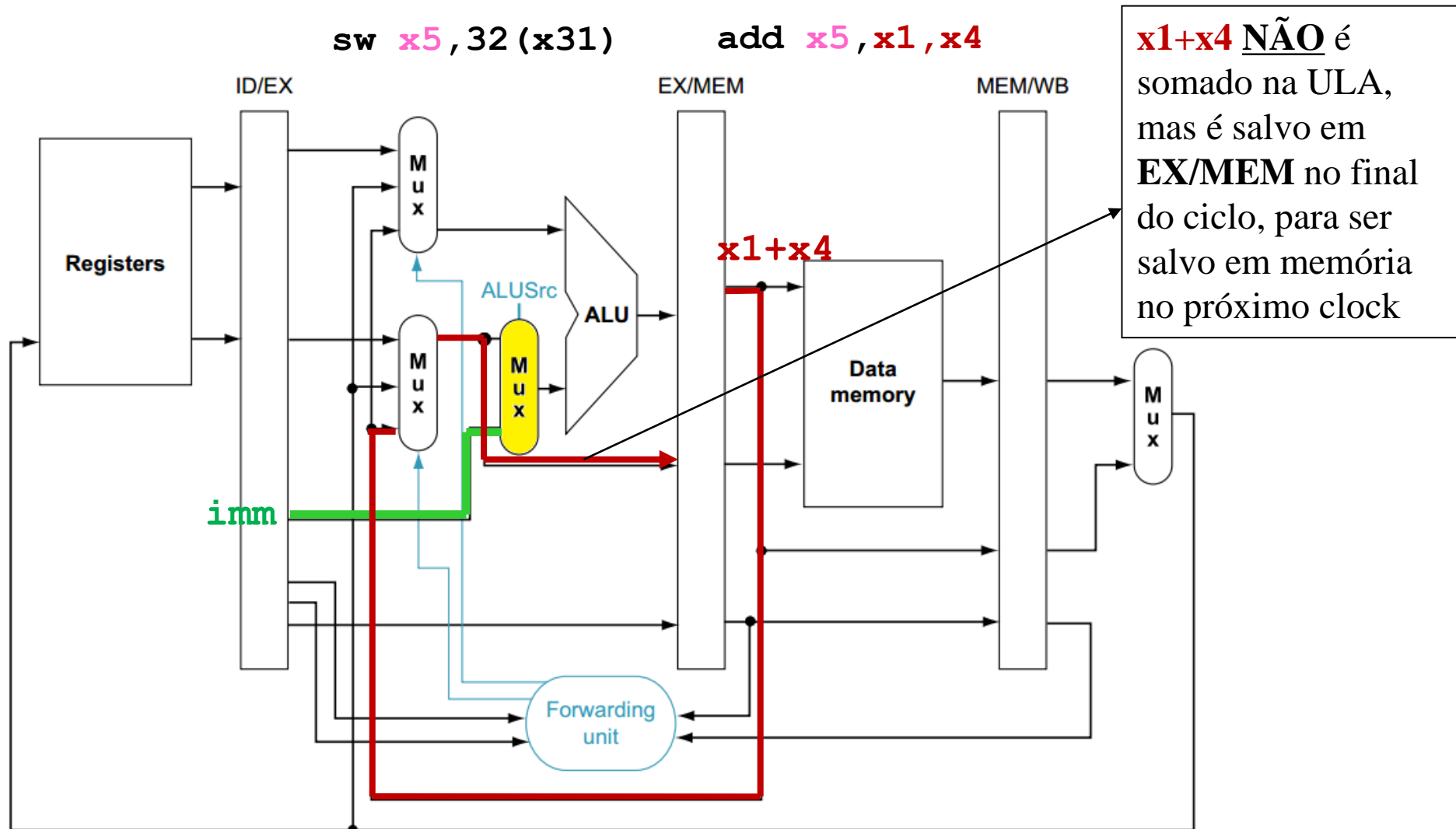
- E este conflito com a **store**?



Observe que diferentemente dos casos anteriores, o resultado da **add** NÃO deveria ser passado como um dos operandos da ULA para a **store**. Pois no 3º estágio, a **store** realiza a soma de **x31+32** (e não com o resultado da **add**). No entanto, queremos que o valor produzido pela **add** seja salvo na memória pela **store**!

Na figura anterior está faltando a valor do imediato (utilizado por **stores** e **loads**) ser enviado à ULA!

- Solução mais fácil: adicionar um **multiplexador 2:1** que escolhe entre a saída do **mux ForwardB** e o valor do **imediato**.



Referências

- PATTERSON, David A; HENNESSY, John L; Computer Organization and Design – The hardware/software interface RISC-V edition; Elsevier – Morgan Kaufmann/Amsterdam.
- PATTERSON, David; Waterman, Andrew; The RISC-V reader: an open architecture atlas; First edition. Berkeley, California: Strawberry Canyon LLC, 2017.