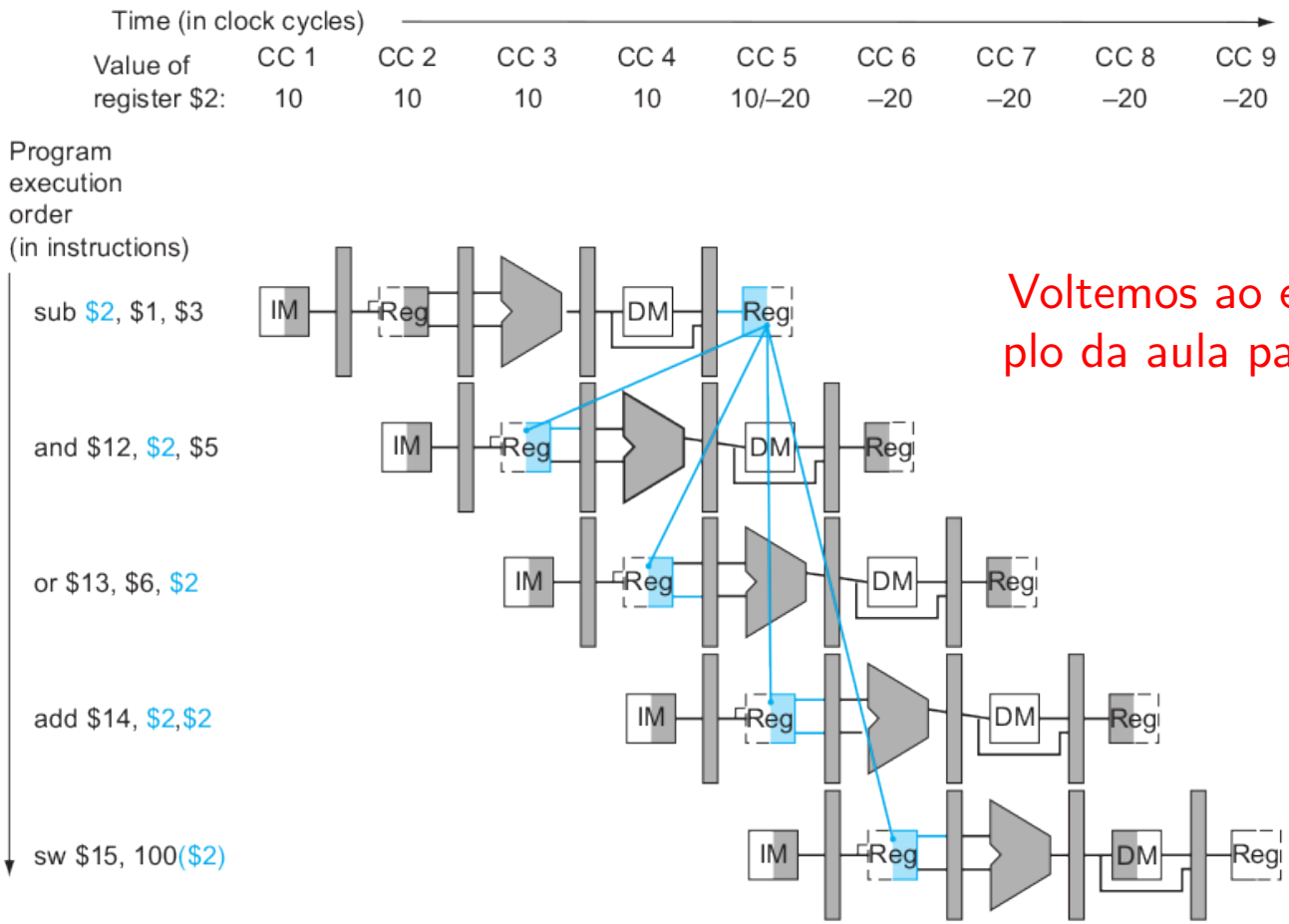
Arquitetura de Computadores
ACH2055
Aula 08 – Conflitos de *Pipeline* e Exceções

Norton Trevisan Roman
(norton@usp.br)

18 de outubro de 2019

Conflitos de Dados

Forwarding

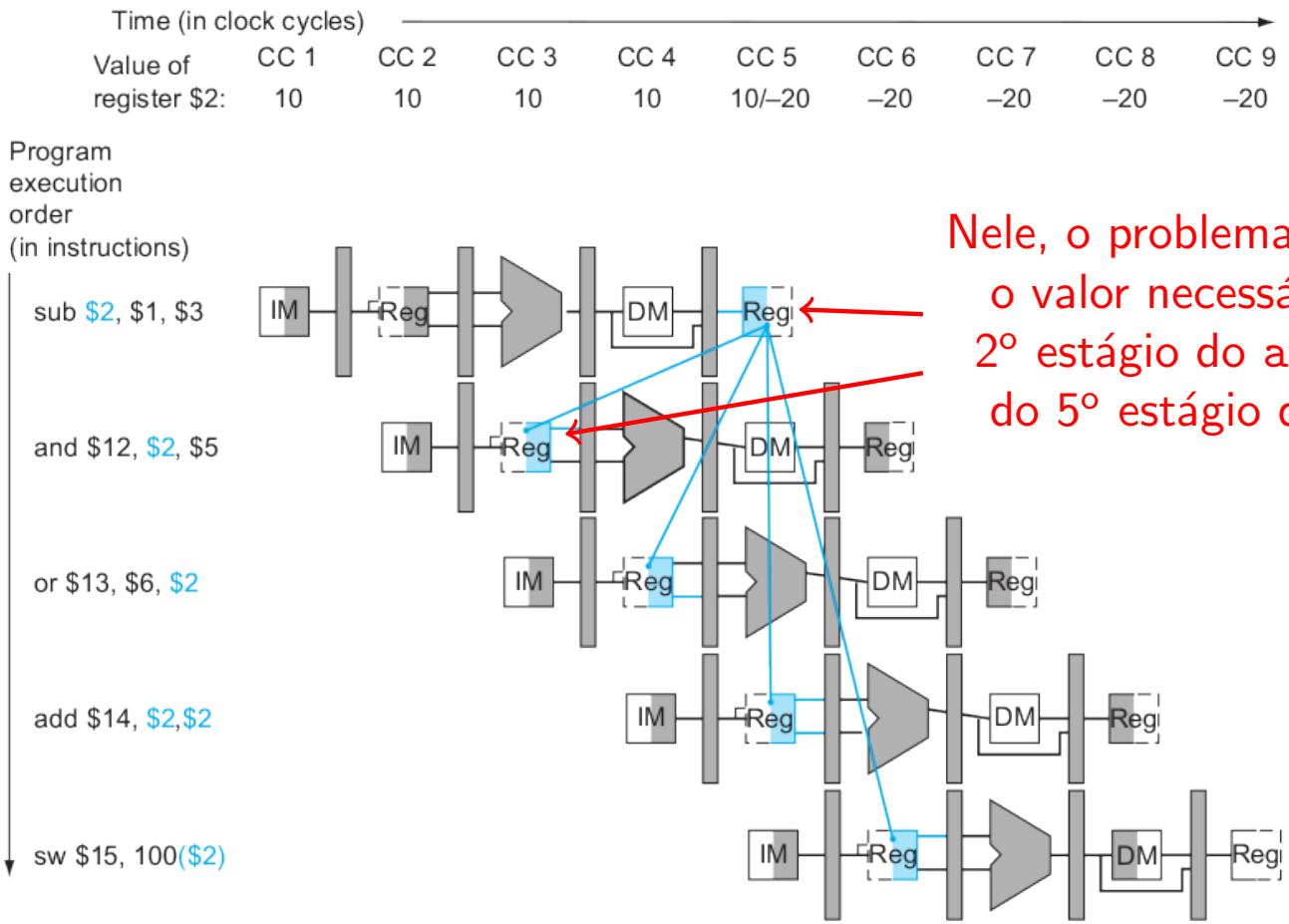


Voltemos ao exemplo da aula passada

Fonte: Adaptado de [1]

Conflitos de Dados

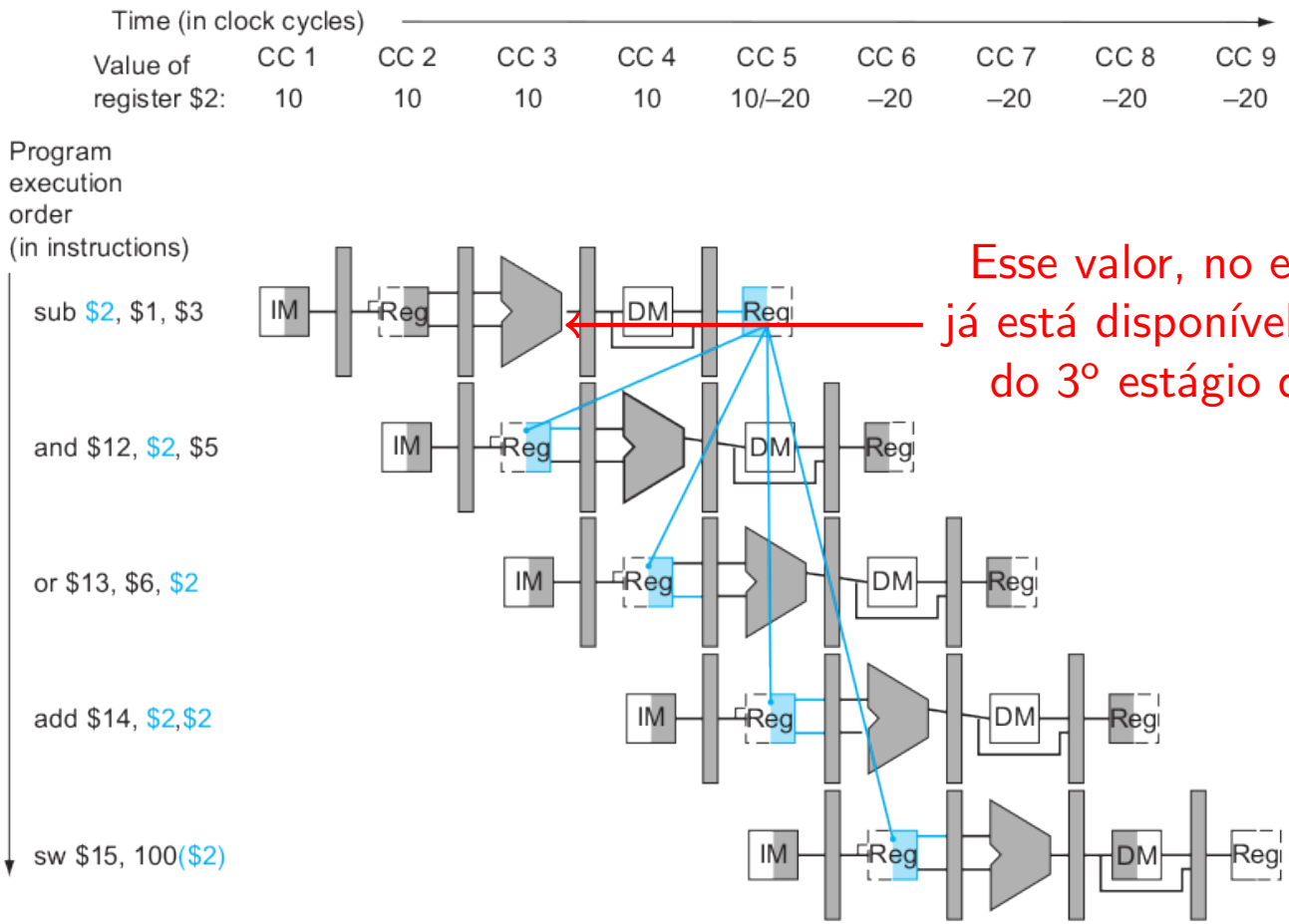
Forwarding



Fonte: Adaptado de [1]

Conflitos de Dados

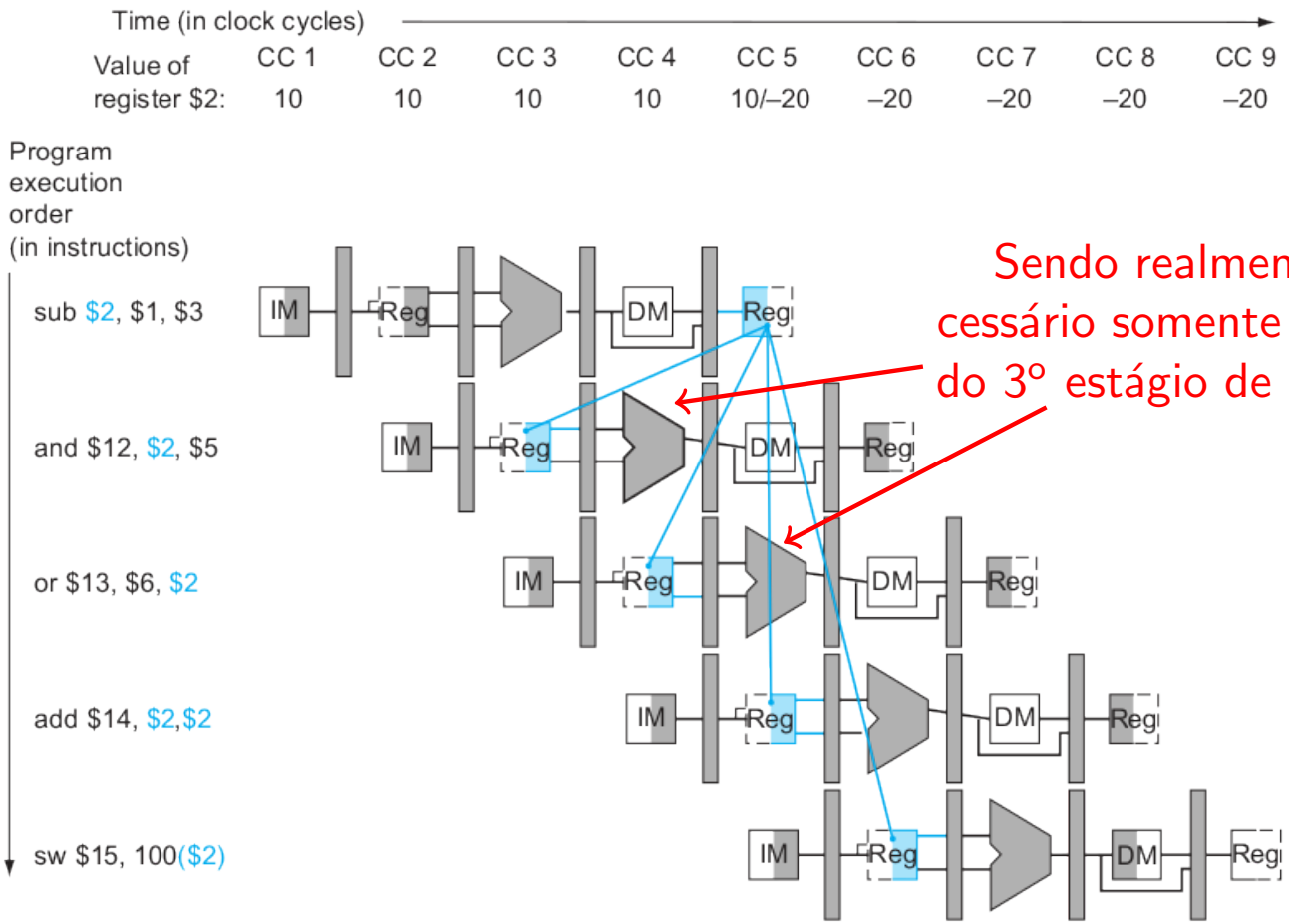
Forwarding



Fonte: Adaptado de [1]

Conflitos de Dados

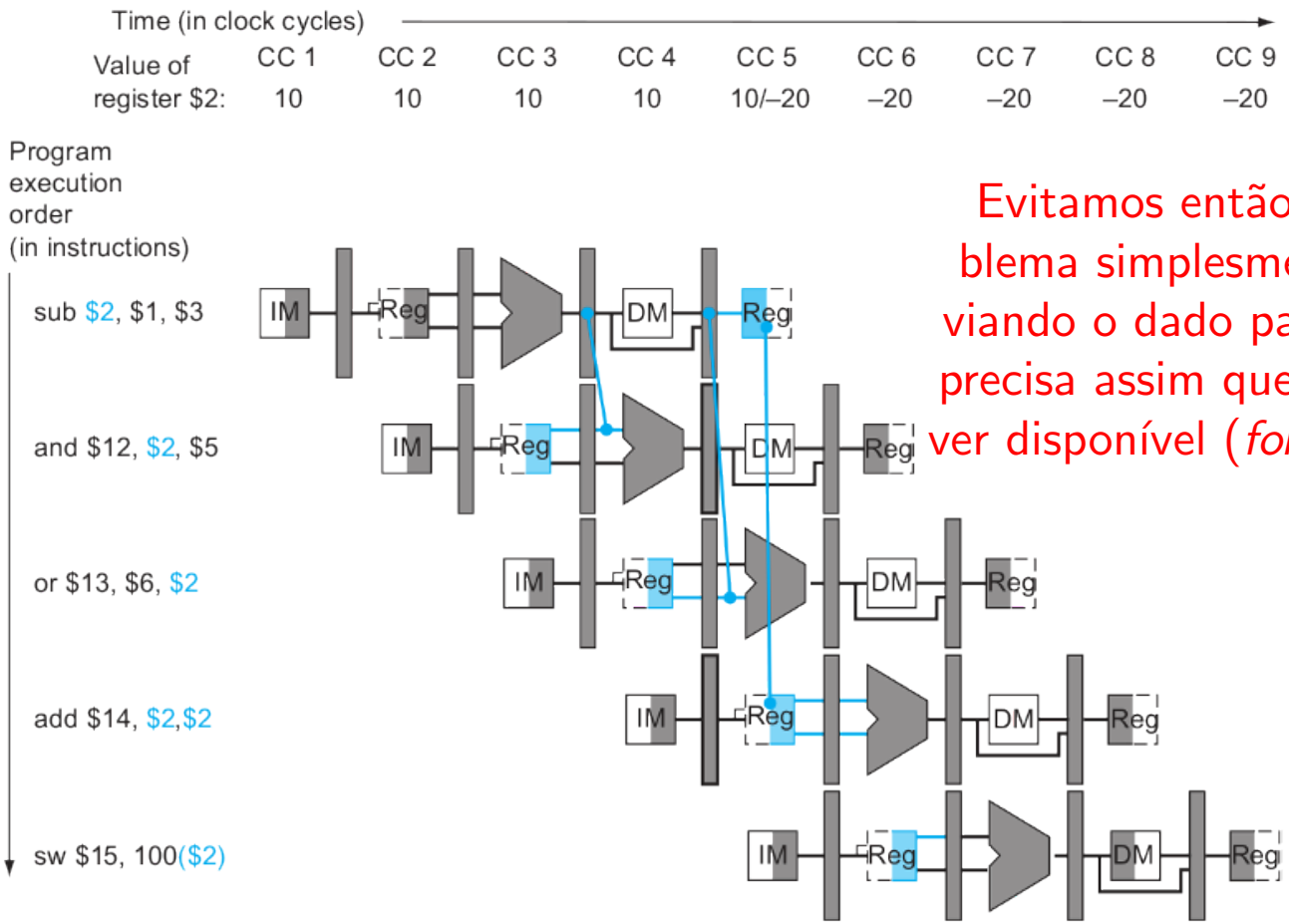
Forwarding



Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding

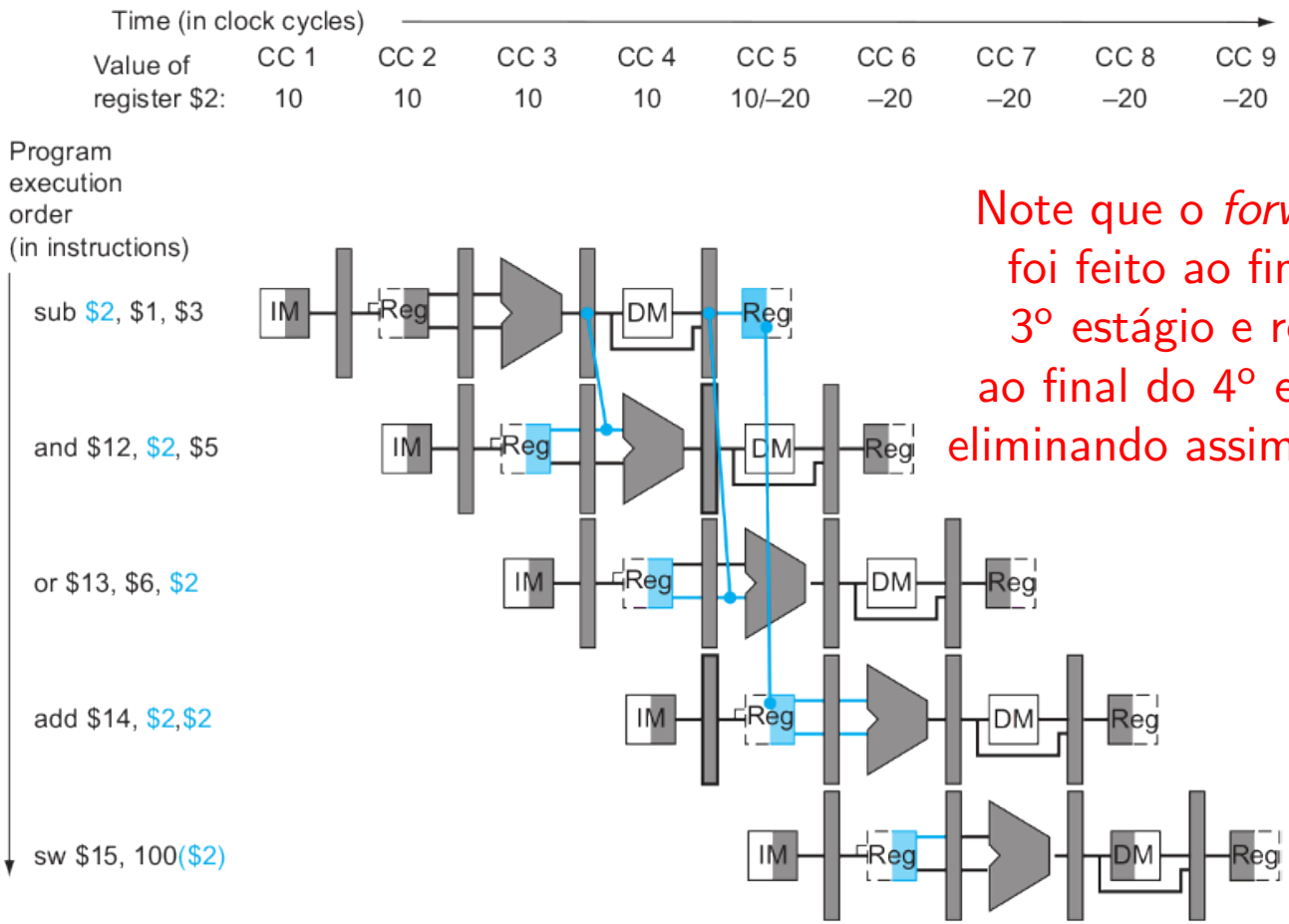


Evitamos então o problema simplesmente enviando o dado para quem precisa assim que ele estiver disponível (*forwarding*)

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding



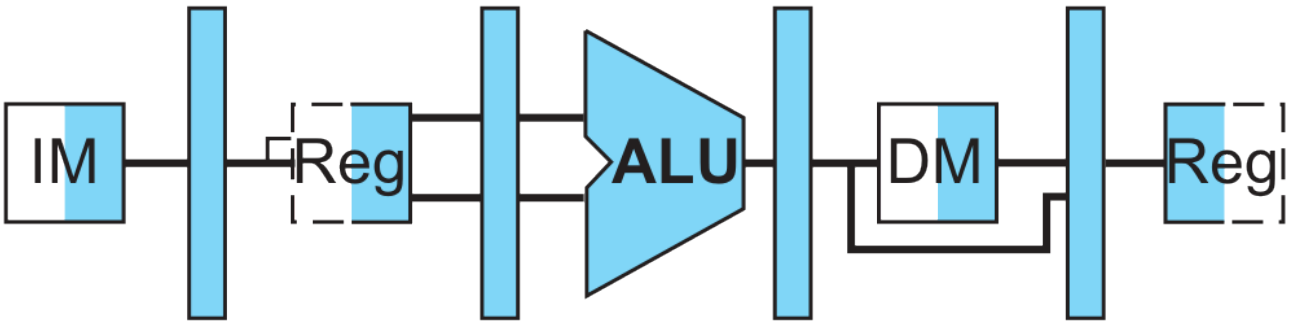
Note que o *forwarding* foi feito ao final do 3º estágio e refeito ao final do 4º estágio, eliminando assim 2 *stalls*

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?



Tipo-R:

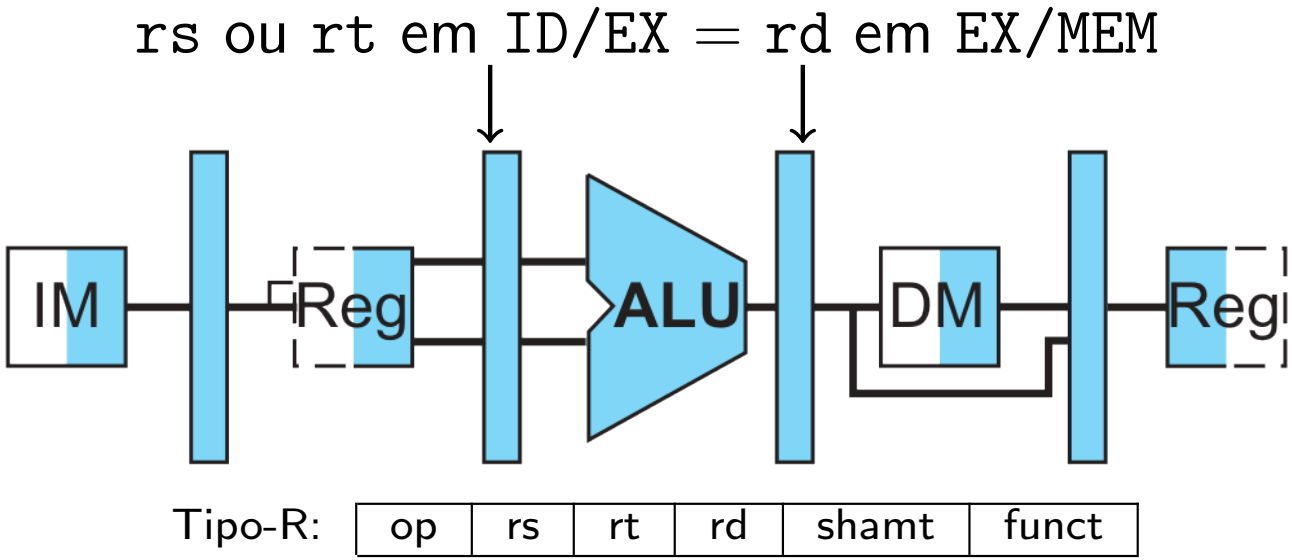
op	rs	rt	rd	shamt	funct

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?



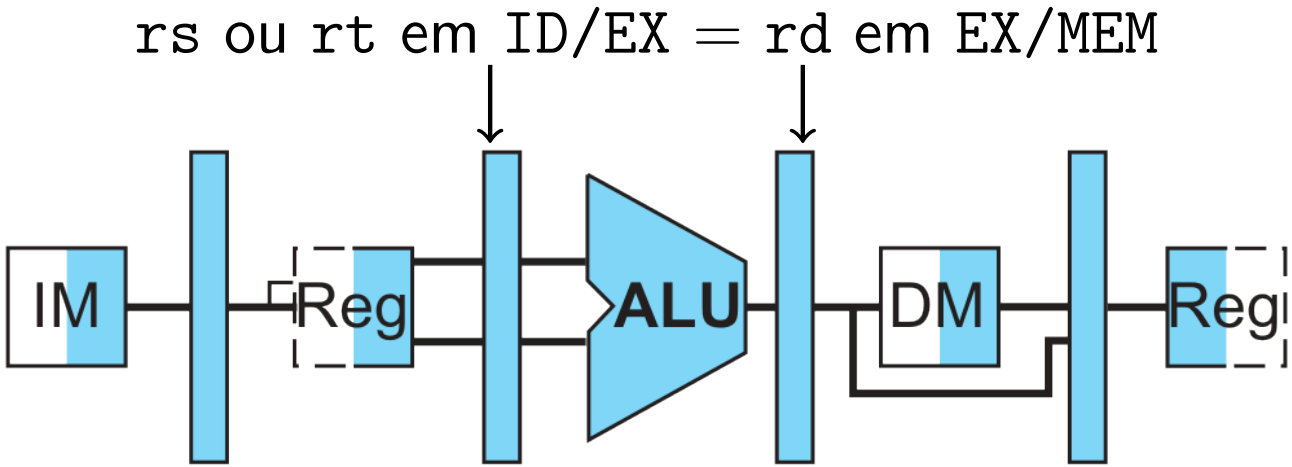
Fonte: Adaptado de [1]

A instrução que sai de EX irá armazenar o resultado da ALU em uma entrada da ALU da instrução seguinte

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?



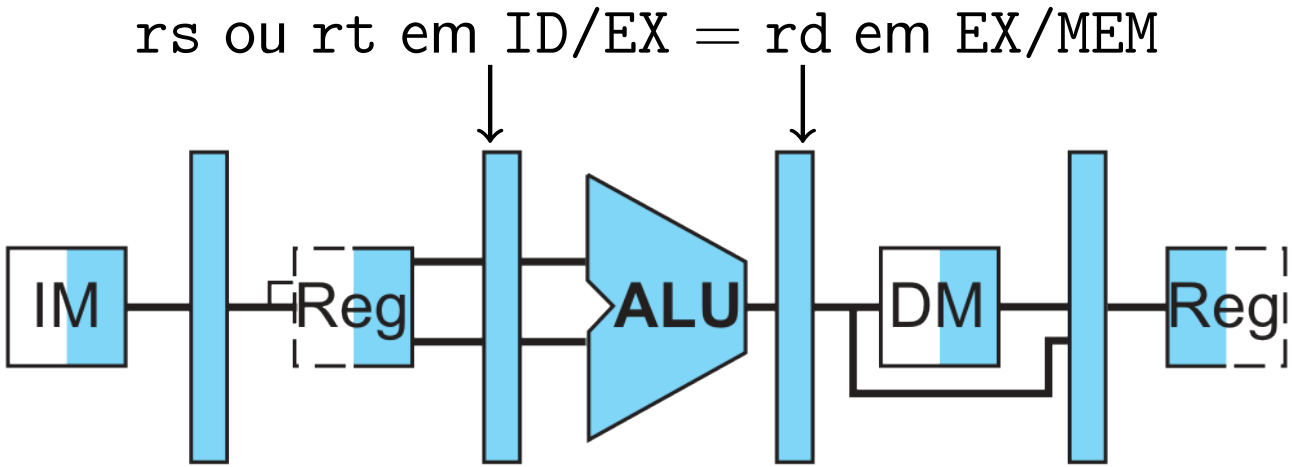
Fonte: Adaptado de [1]

Ou seja, o resultado de uma instrução é usado diretamente pela ALU da instrução seguinte

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?



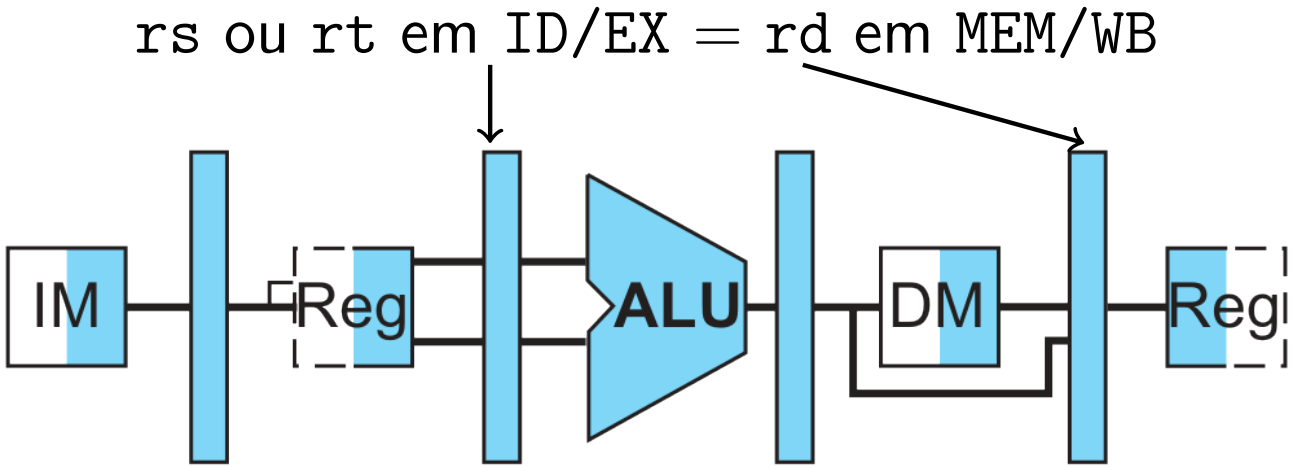
Fonte: Adaptado de [1]

Mas somente se a instrução em EX/MEM for realmente escrever no registrador (caso em que seu RegWrite = 1)

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?



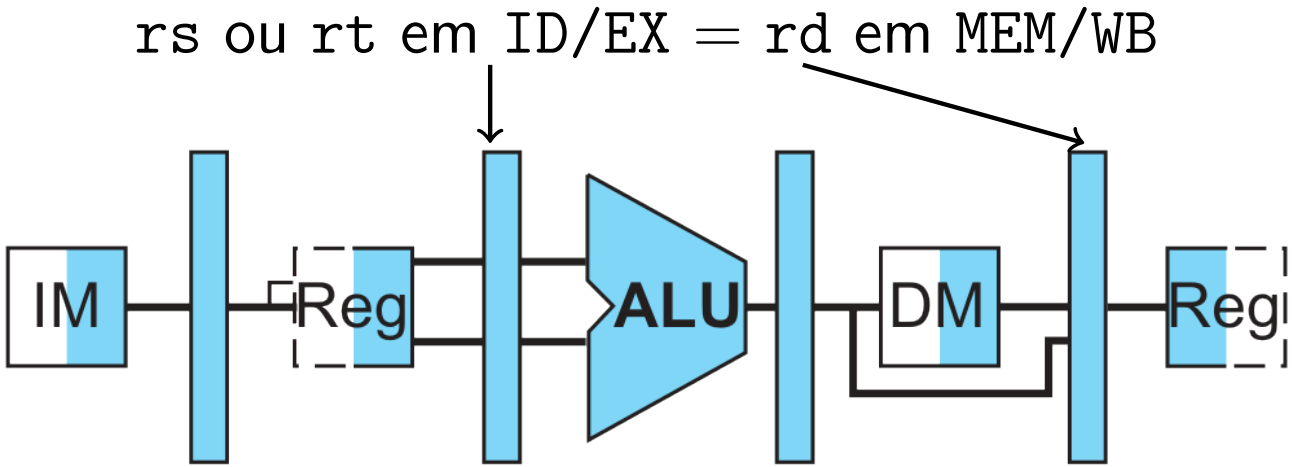
Fonte: Adaptado de [1]

A instrução que sai de MEM irá armazenar o valor lido em uma entrada da ALU da instrução que sai de ID

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?



Fonte: Adaptado de [1]

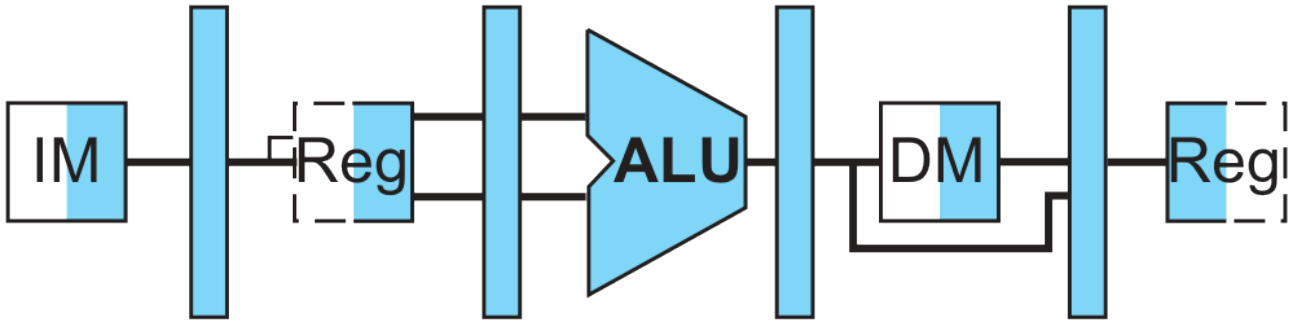
Novamente, somente se a instrução em `MEM/WB` for realmente escrever no registrador (seu `RegWrite = 1`)

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?

Isso, contudo, não impede *forwarding* no caso de tentativas de escrita em `$s0` (`rd` = `$s0`)



Tipo-R:

op	rs	rt	rd	shamt	funct

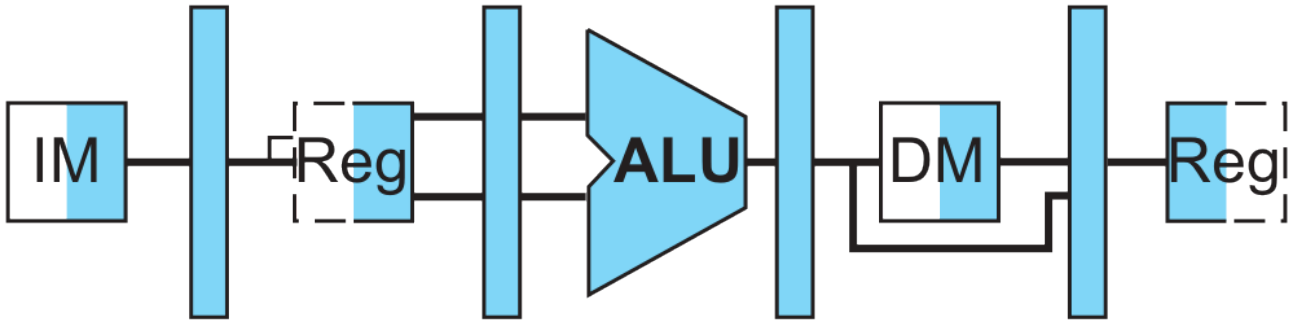
Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding

- Sob que condições *forwarding* deve ocorrer?

Isso, contudo, não impede *forwarding* no caso de tentativas de escrita em `$s0` (`rd` = `$s0`)



Tipo-R:

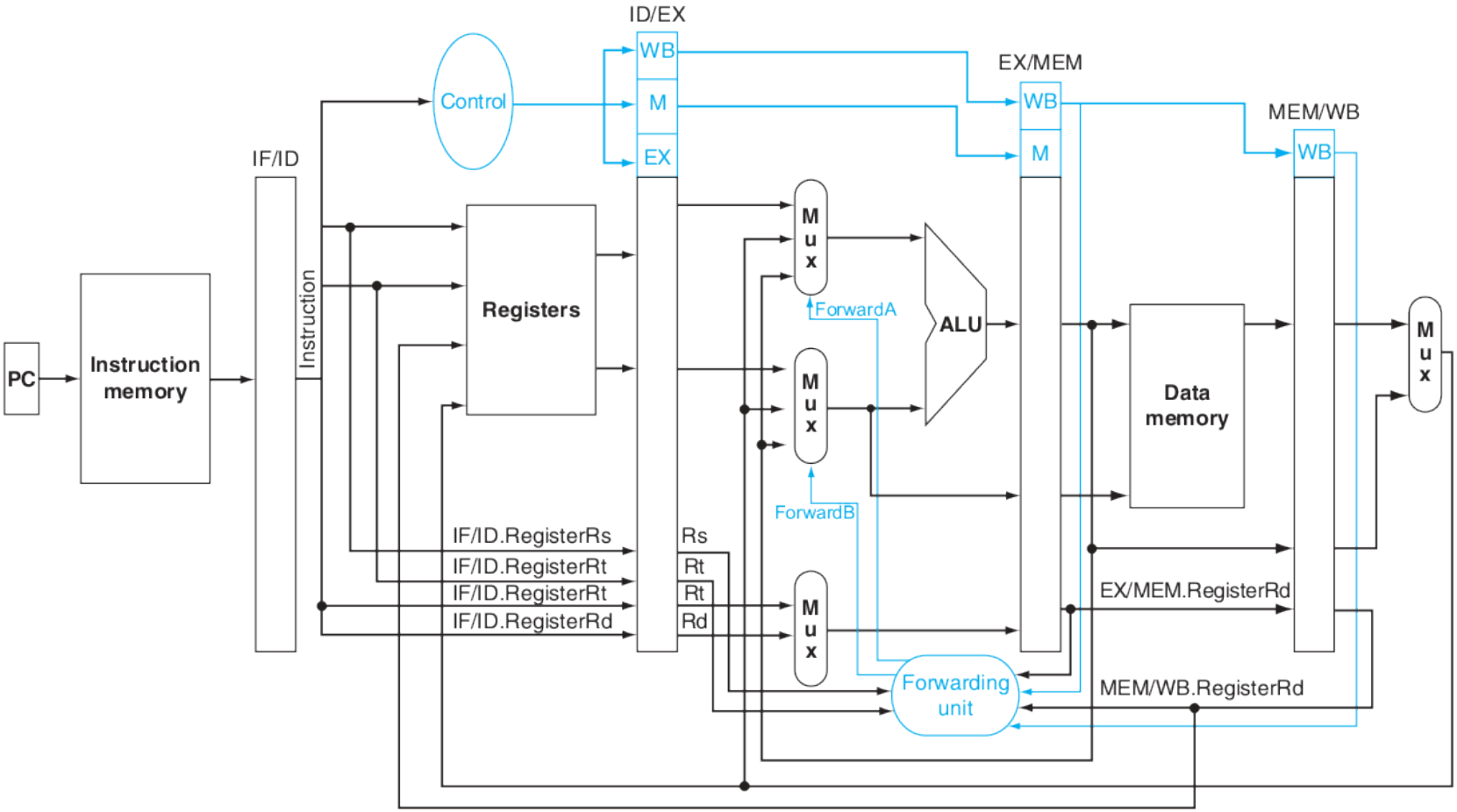
op	rs	rt	rd	shamt	funct

Fonte: Adaptado de [1]

Devemos então adicionar essa condição

Conflitos de Dados

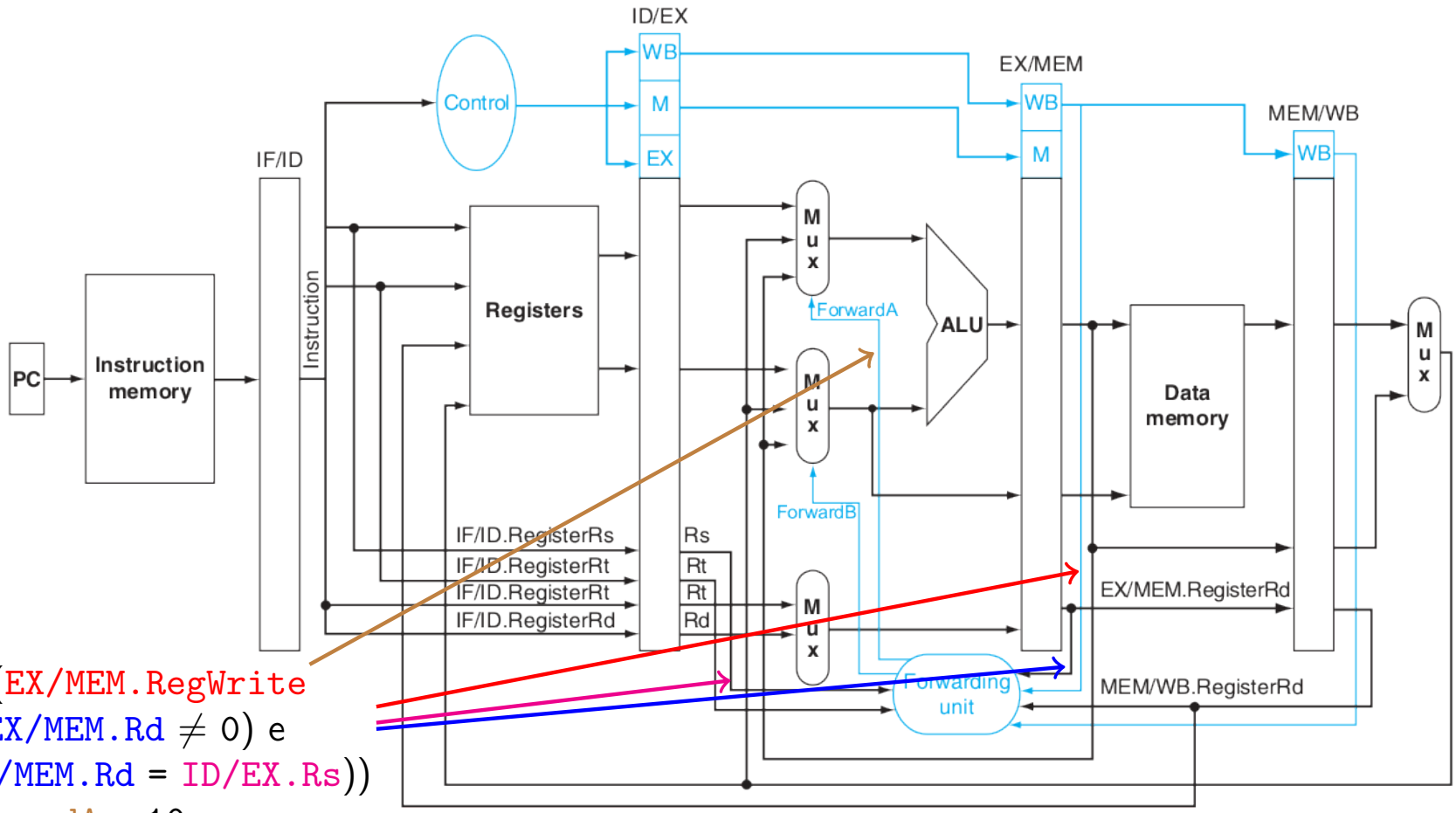
Forwarding Unit: Condições de Controle



Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

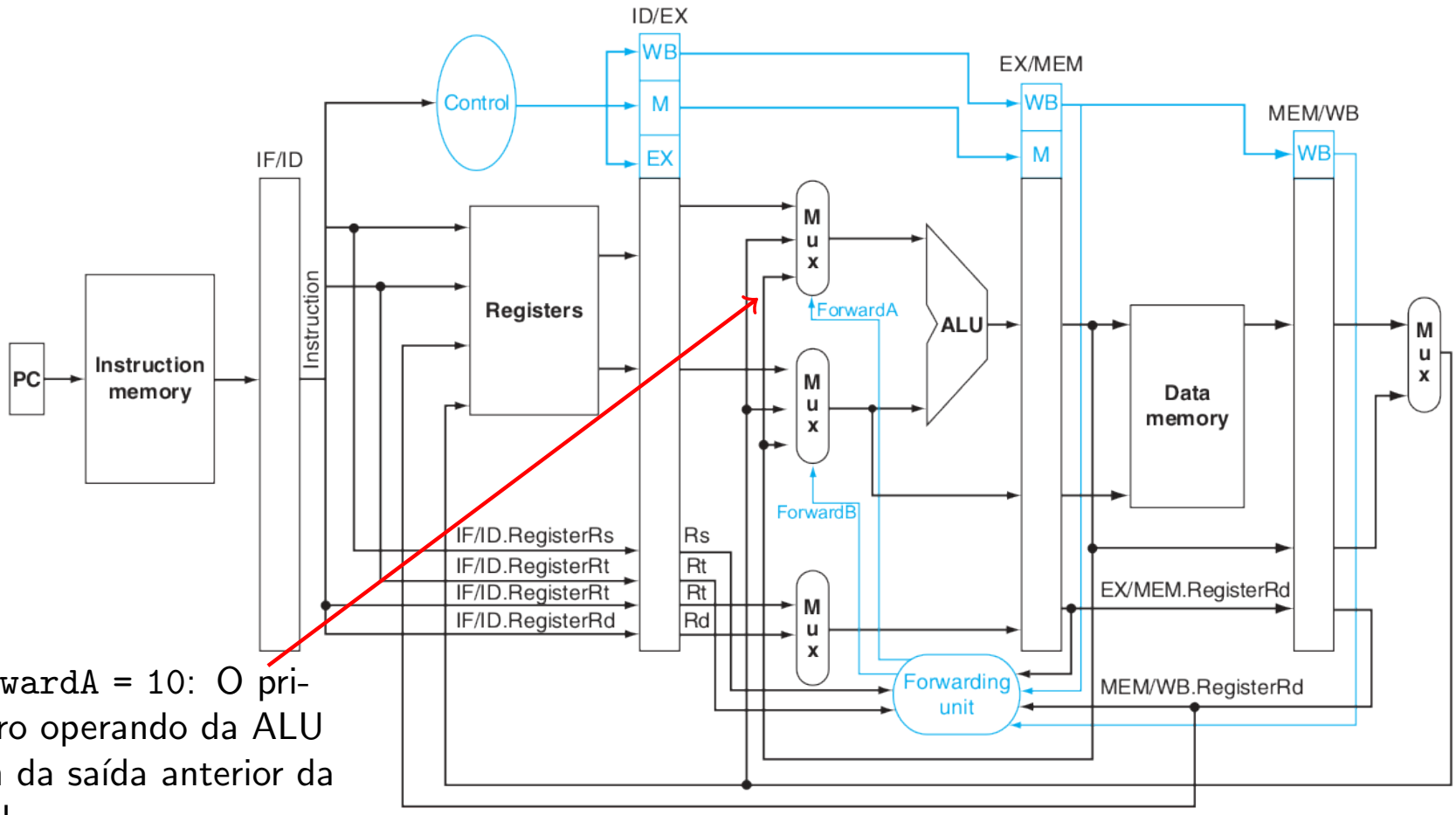


Se (EX/MEM.RegWrite
e (EX/MEM.Rd $\neq$ 0) e
(EX/MEM.Rd = ID/EX.Rs))
ForwardA = 10

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

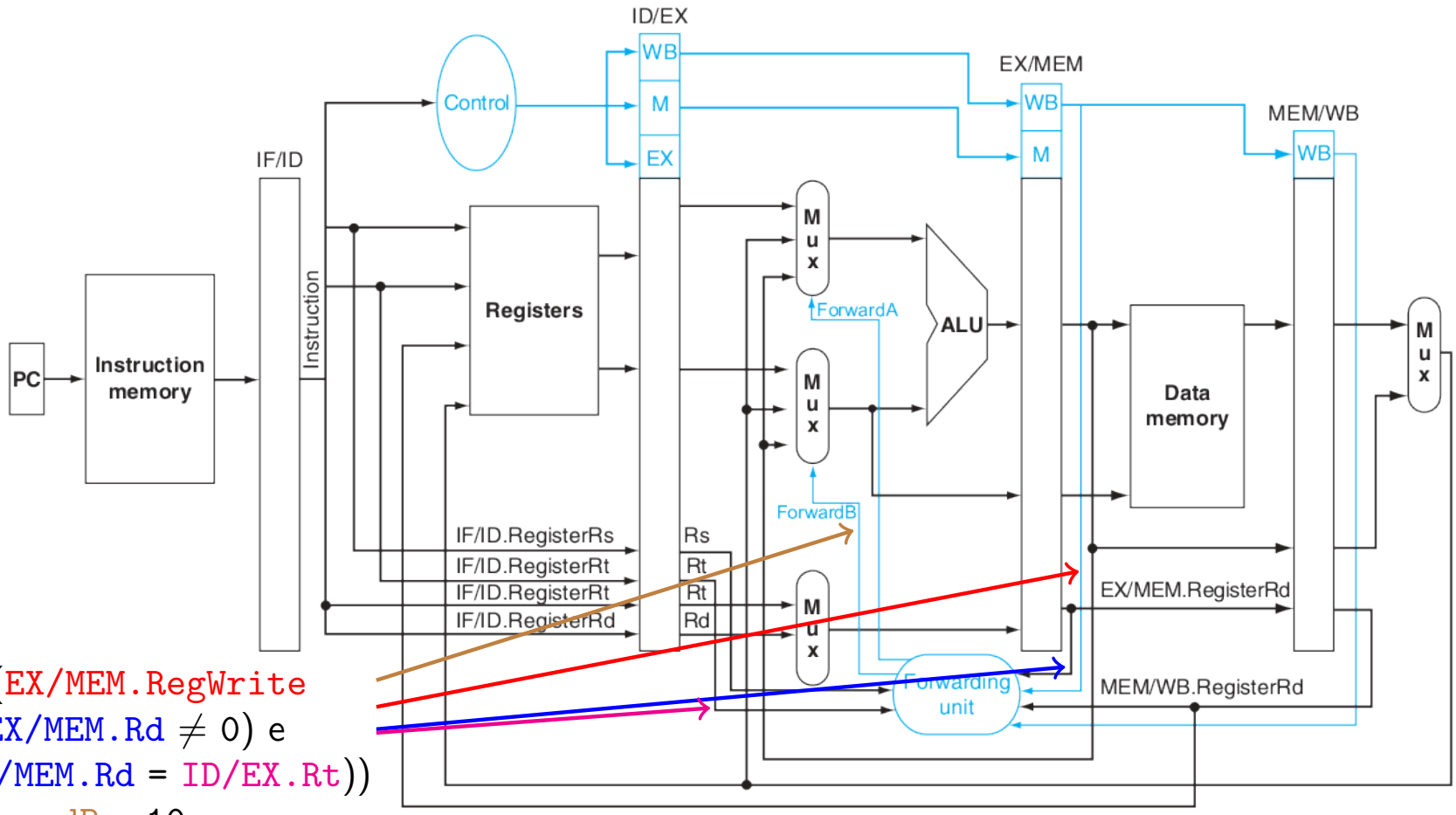


`ForwardA = 10`: O primeiro operando da ALU vem da saída anterior da ALU

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

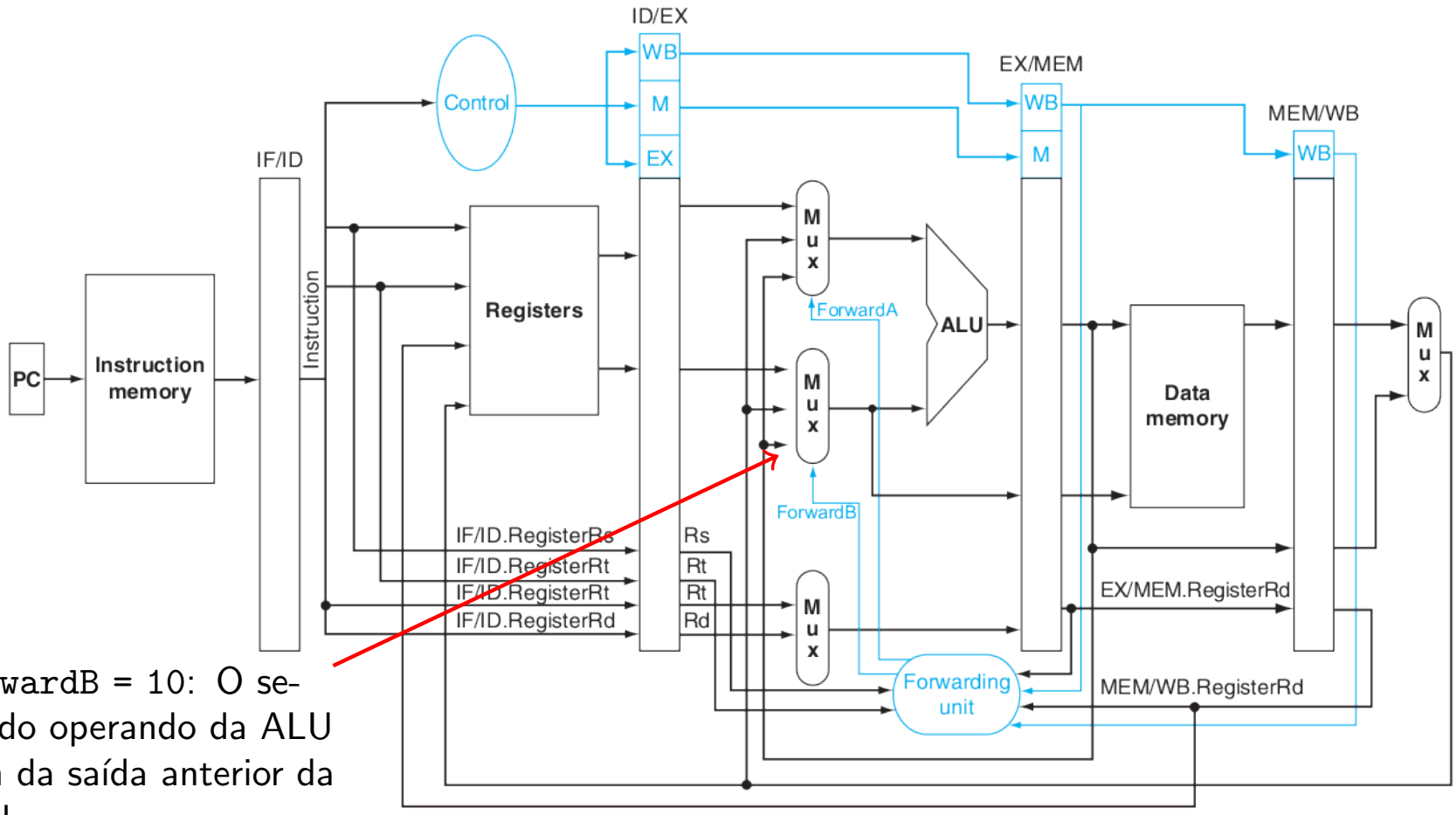


Se (EX/MEM.RegWrite
e (EX/MEM.Rd $\neq$ 0) e
(EX/MEM.Rd = ID/EX.Rt))
ForwardB = 10

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

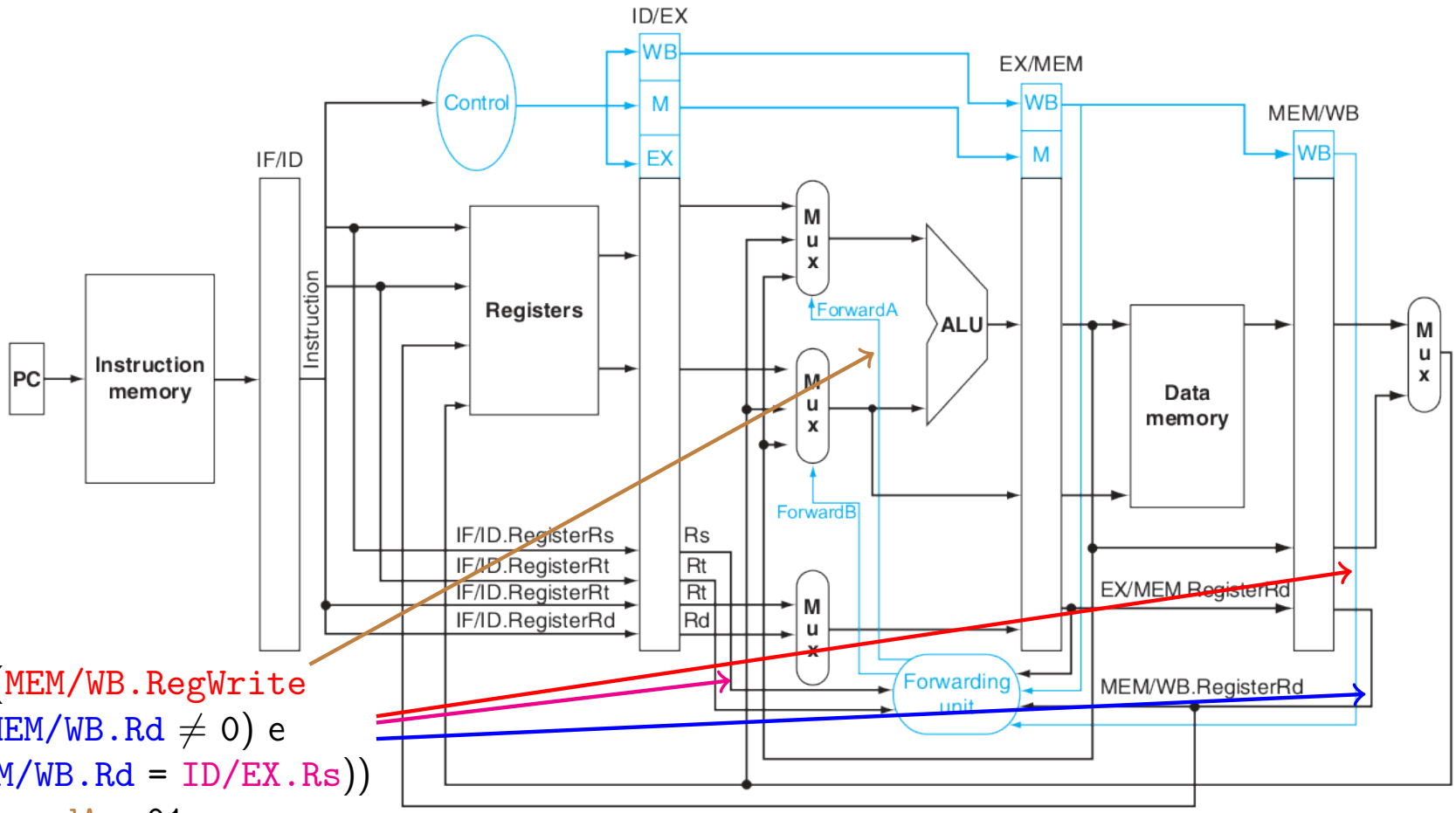


`ForwardB = 10`: O segundo operando da ALU vem da saída anterior da ALU

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

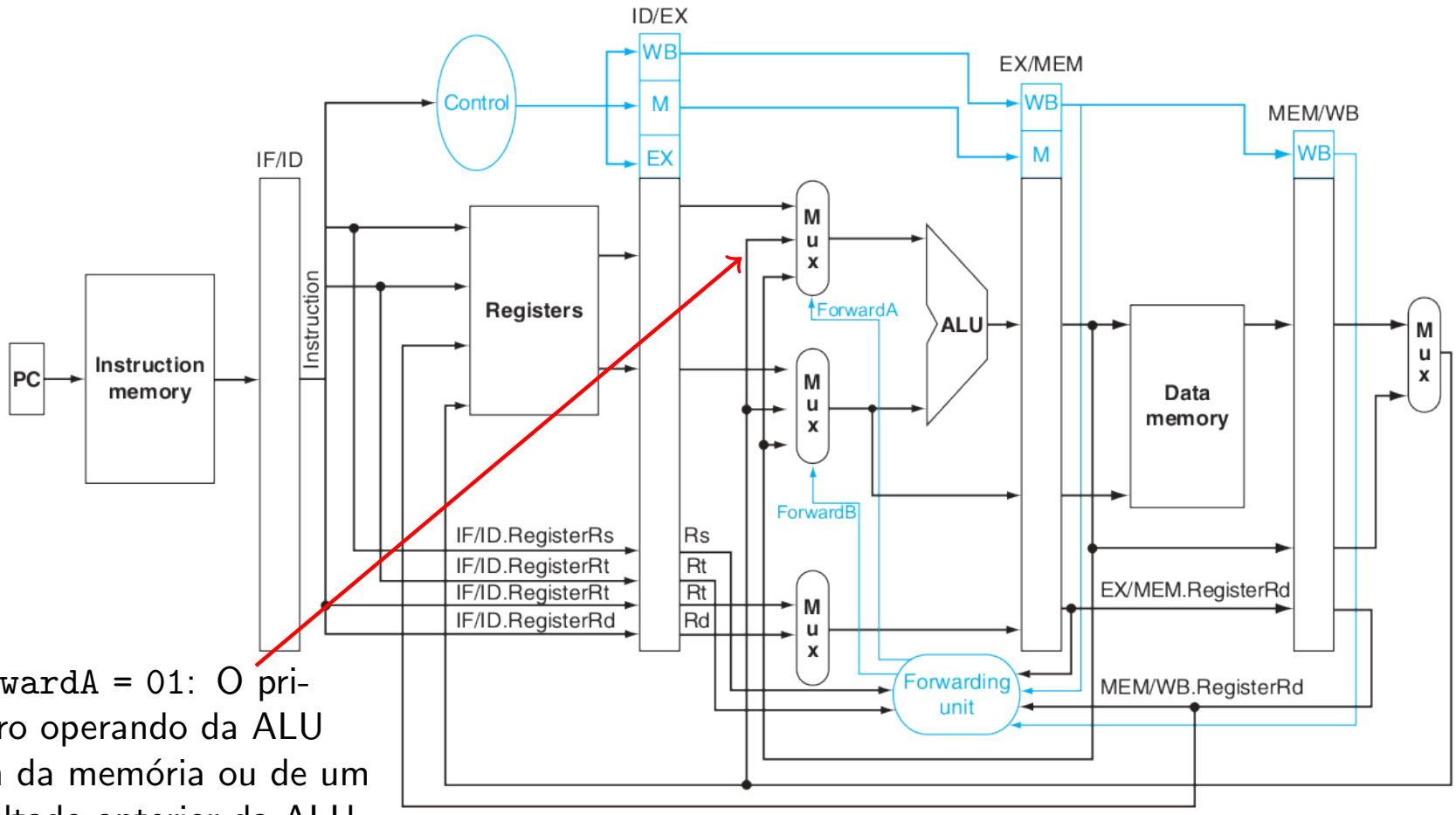


Se (MEM/WB.RegWrite
e (MEM/WB.Rd $\neq$ 0) e
(MEM/WB.Rd = ID/EX.Rs))
ForwardA = 01

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

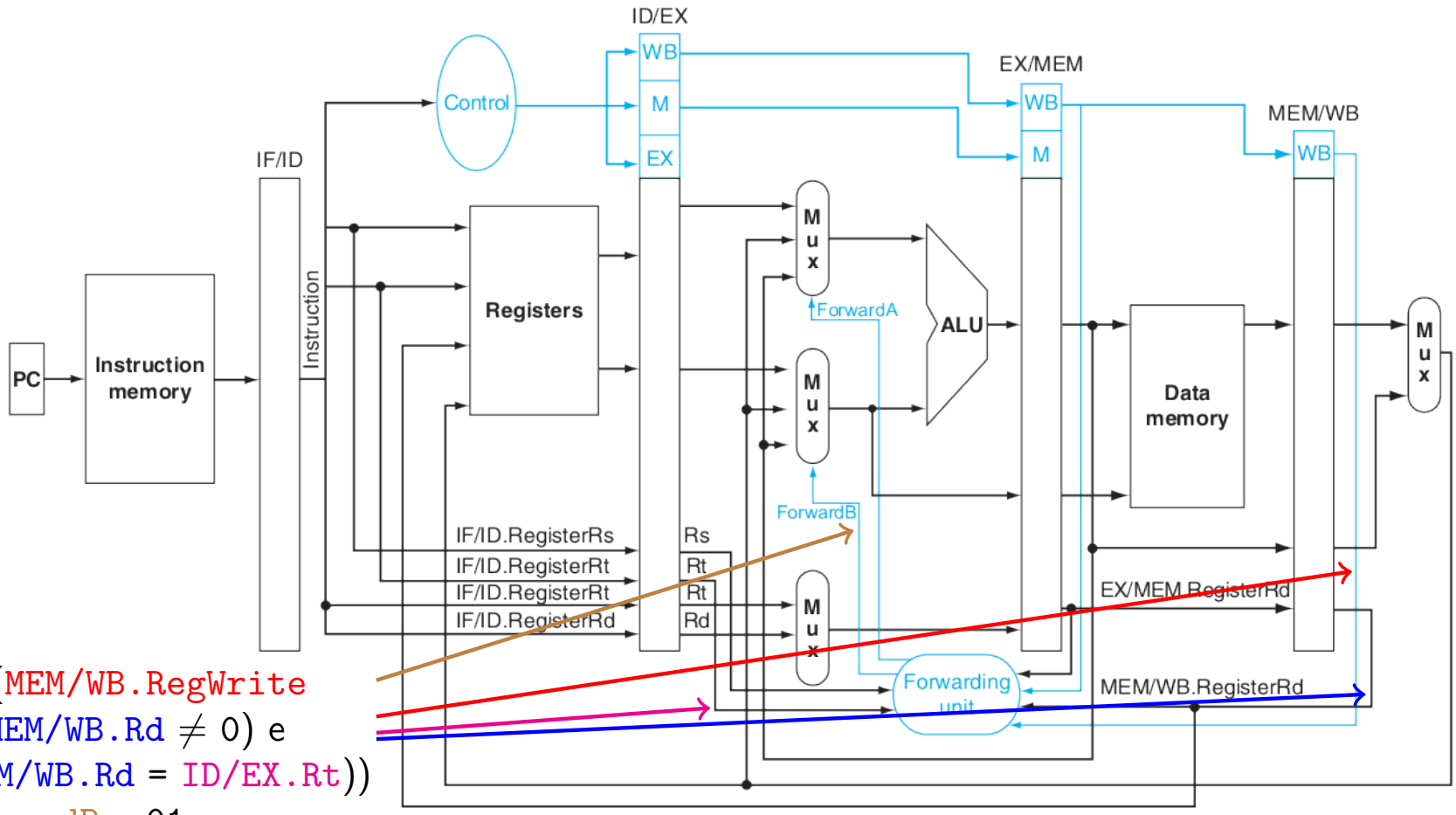


`ForwardA = 01`: O primeiro operando da ALU vem da memória ou de um resultado anterior da ALU

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

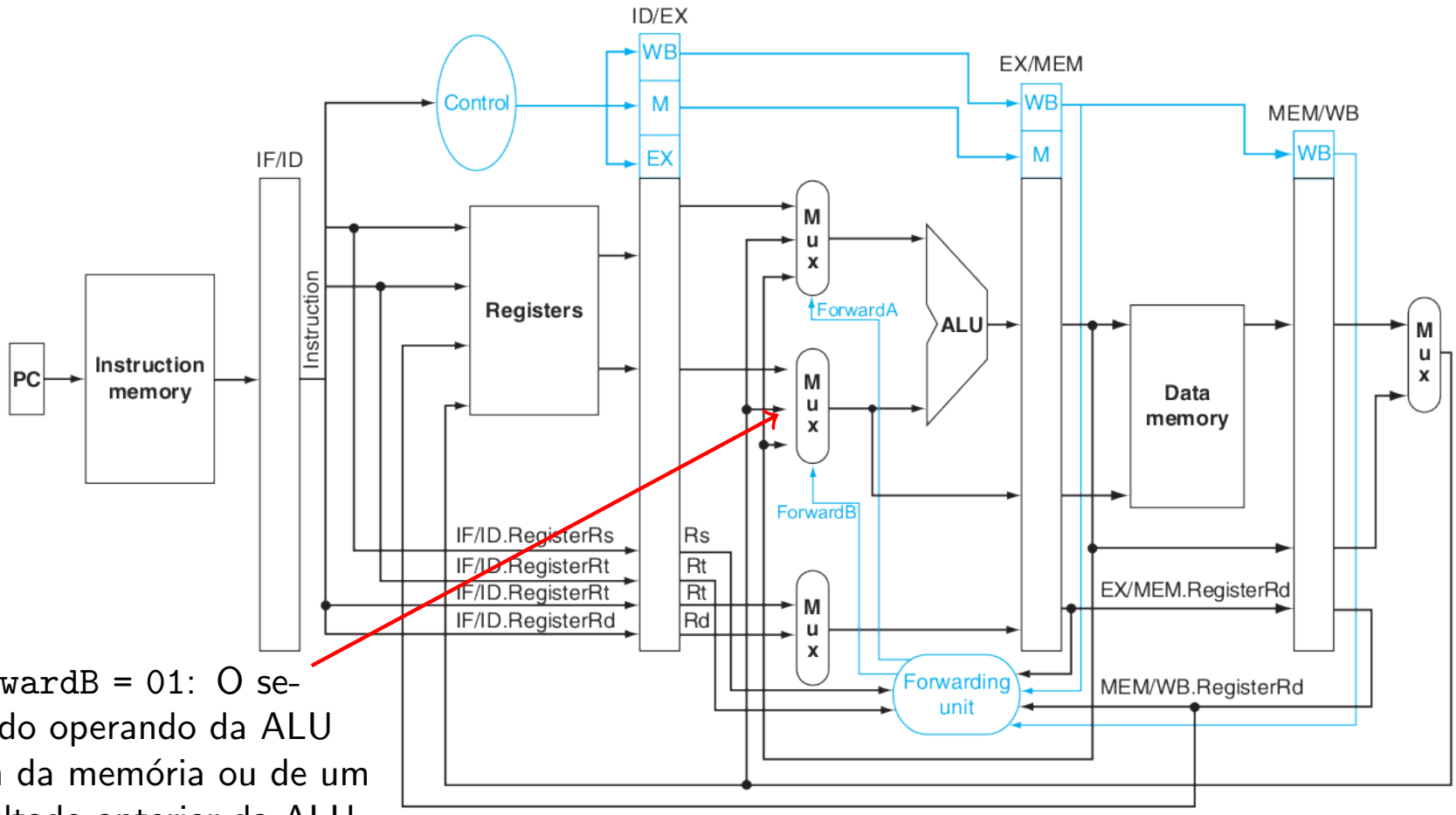


Se (MEM/WB.RegWrite
e (MEM/WB.Rd $\neq$ 0) e
(MEM/WB.Rd = ID/EX.Rt))
ForwardB = 01

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

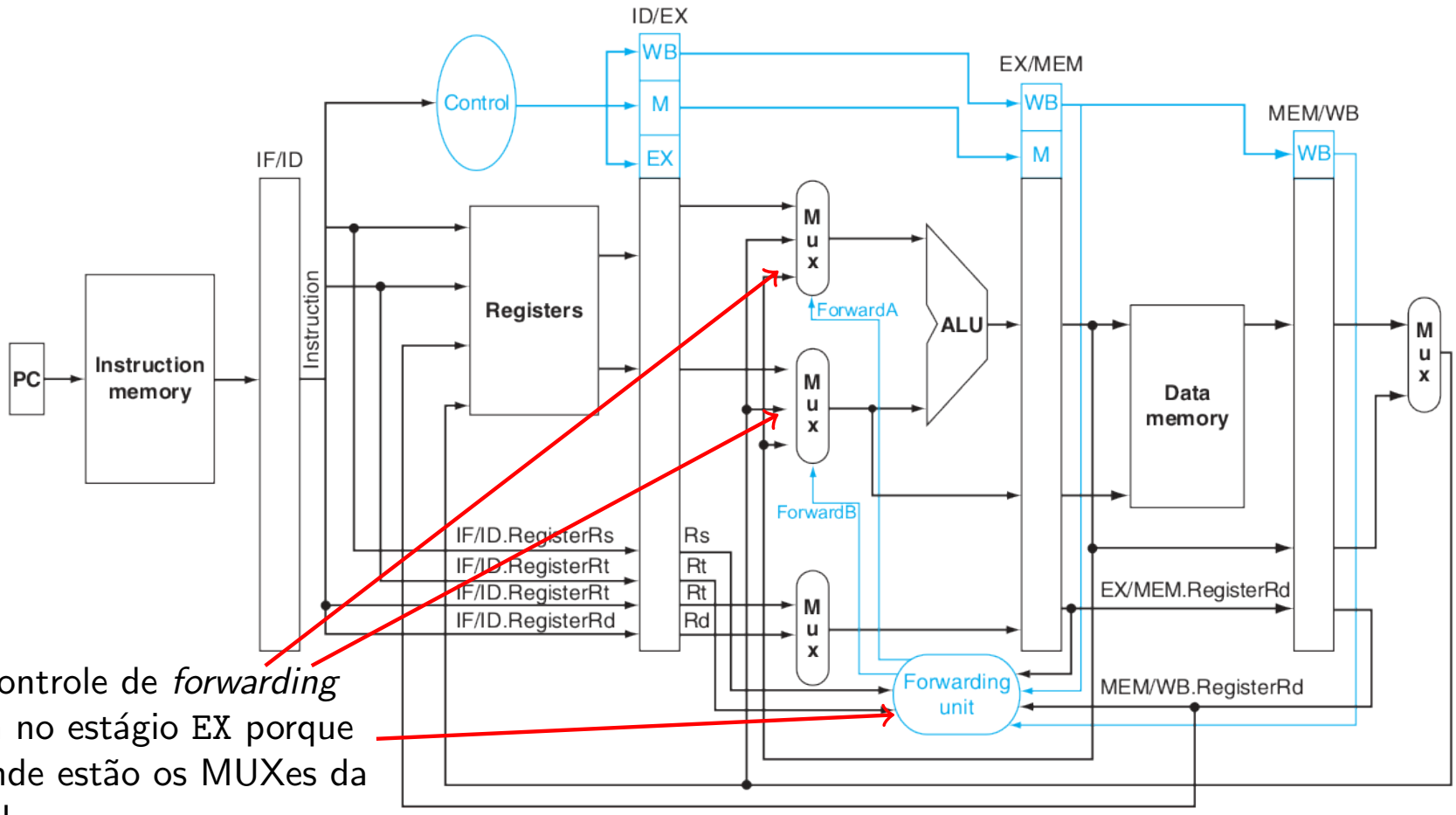


`ForwardB = 01`: O segundo operando da ALU vem da memória ou de um resultado anterior da ALU

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

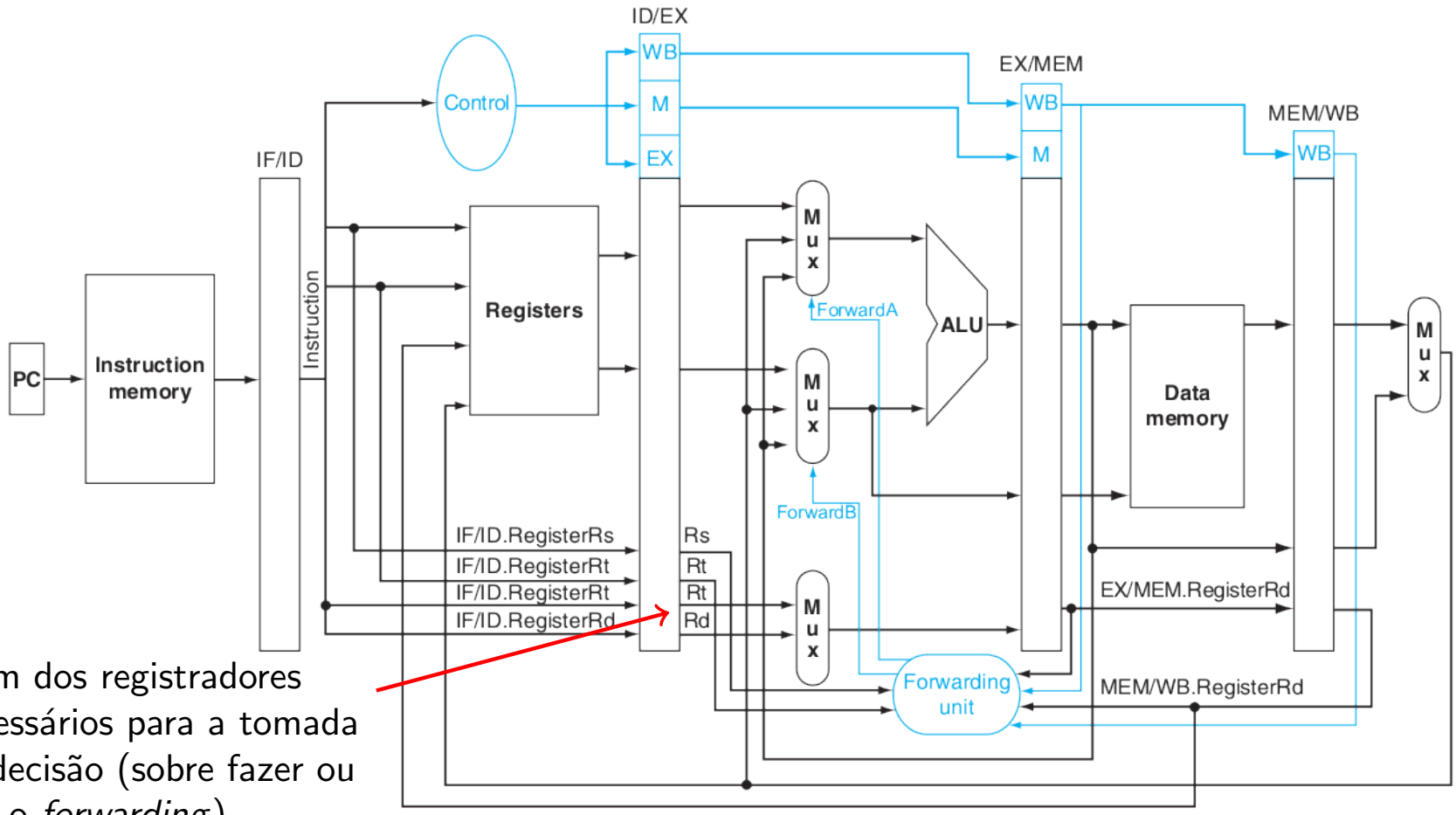


O controle de *forwarding* está no estágio `EX` porque é onde estão os MUXes da ALU

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

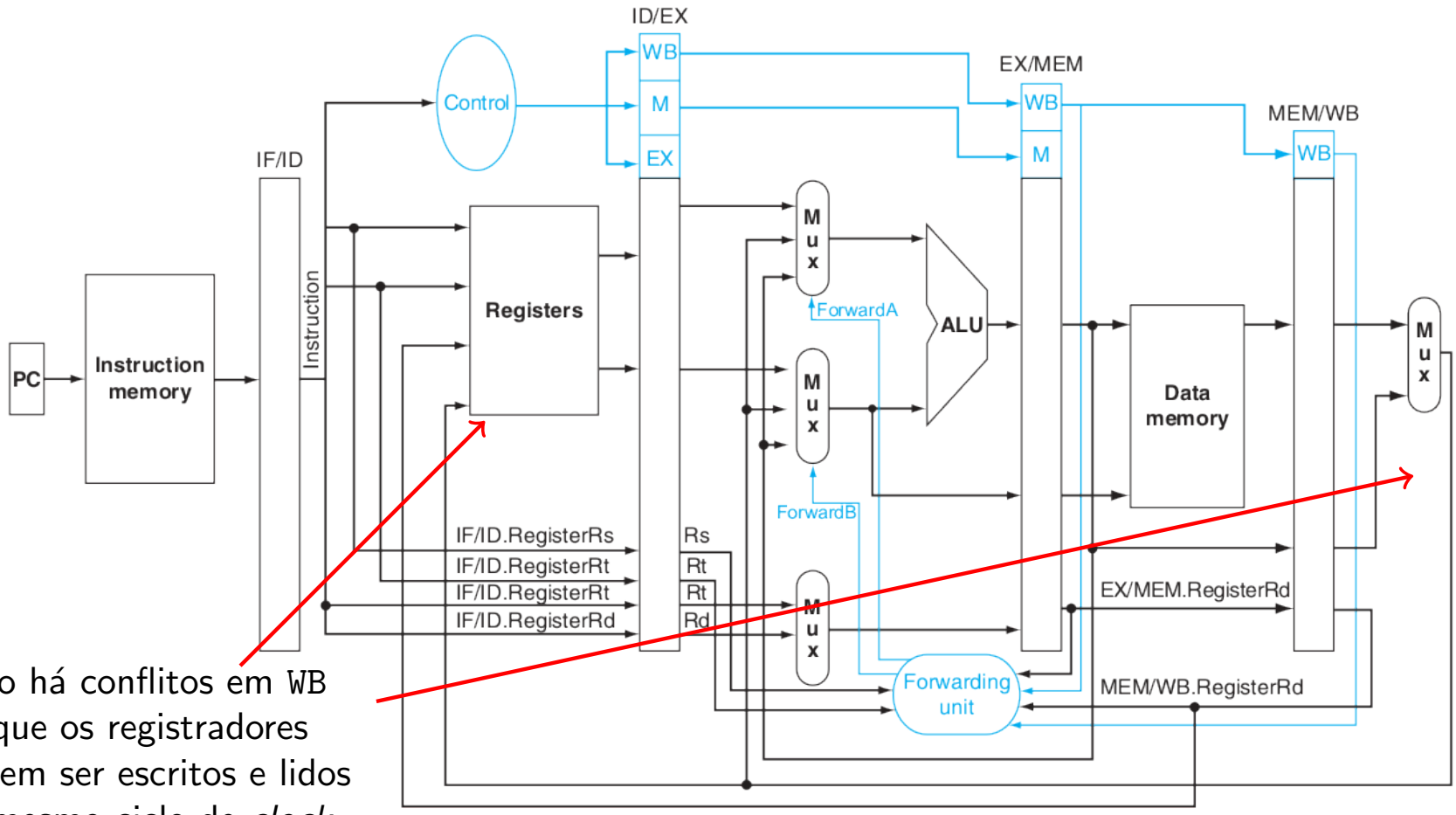


Além dos registradores necessários para a tomada de decisão (sobre fazer ou não o *forwarding*)

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle



Não há conflitos em `WB` porque os registradores podem ser escritos e lidos no mesmo ciclo de *clock*

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Estamos livres de problemas?

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Estamos livres de problemas?
- Considere o seguinte código:

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Estamos livres de problemas?
- Considere o seguinte código:

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

De onde virá esse valor?
(`add RD, RS, RT`)

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Estamos livres de problemas?
- Considere o seguinte código:

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

De onde virá esse valor?
(`add RD, RS, RT`)

Se (`MEM/WB.RegWrite` e (`MEM/WB.Rd` $\neq$ `0`)
e (`MEM/WB.Rd = ID/EX.Rs`)) `ForwardA = 01`

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Estamos livres de problemas?
- Considere o seguinte código:

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

De onde virá esse valor?
(add RD, RS, RT)

Se (MEM/WB.RegWrite e (MEM/WB.Rd $\neq$ 0)
e (MEM/WB.Rd = ID/EX.Rs)) ForwardA = 01

Se (EX/MEM.RegWrite e (EX/MEM.Rd $\neq$ 0)
e (EX/MEM.Rd = ID/EX.Rs)) ForwardA = 10

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Estamos livres de problemas?
- Considere o seguinte código:

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

OPS! Dos 2 estágios seguintes...

Se (MEM/WB.RegWrite e (MEM/WB.Rd $\neq$ 0)
e (MEM/WB.Rd = ID/EX.Rs)) ForwardA = 01

Se (EX/MEM.RegWrite e (EX/MEM.Rd $\neq$ 0)
e (EX/MEM.Rd = ID/EX.Rs)) ForwardA = 10

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Que fazer então?

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Que fazer então? Fazer com que valha o mais novo:

```
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.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) 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.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Fonte: Adaptado de [1]

Conflitos de Dados

Forwarding Unit: Condições de Controle

- Que fazer então? Fazer com que valha o mais novo:

```
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.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) 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.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Somente fará *forwarding* de `WB`
se este não for acontecer em `MEM`

Fonte: Adaptado de [1]

Conflitos de Dados

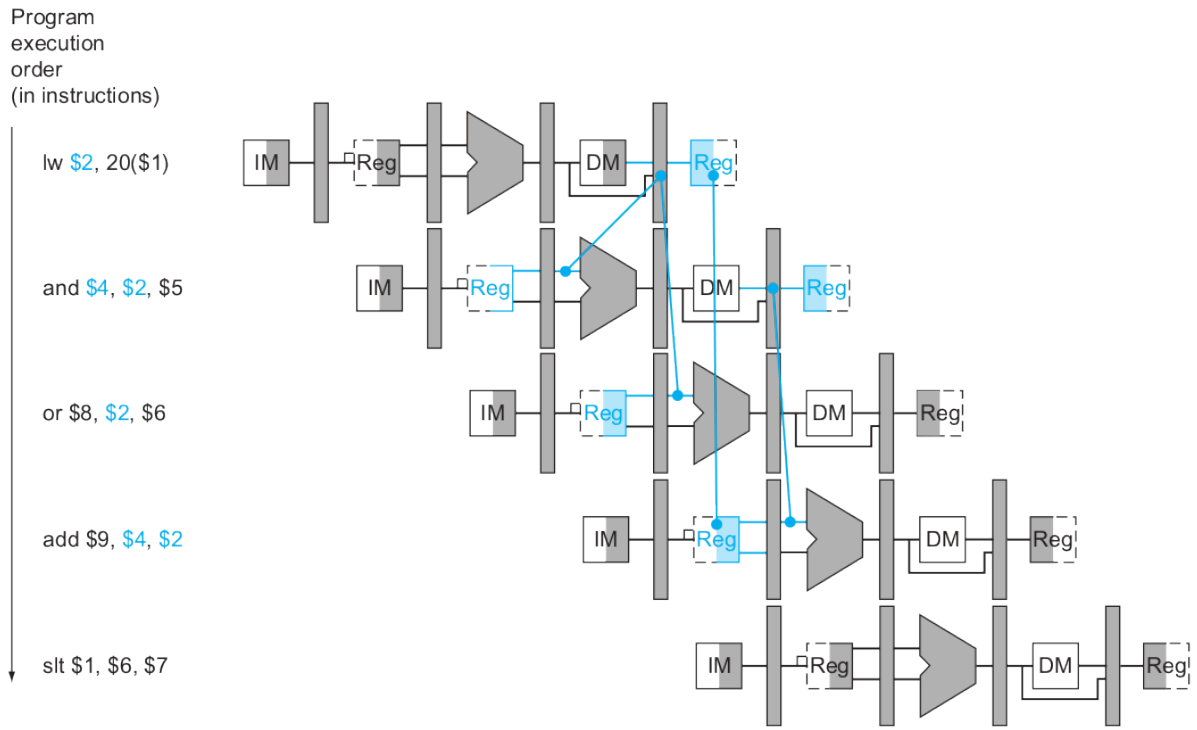
Pipeline Stalls

- Vimos na aula passada que *forwarding* não evita todo problema

Conflitos de Dados

Pipeline Stalls

- Vimos na aula passada que *forwarding* não evita todo problema
  - Como quando uma instrução lê um registrador em seguida a um `lw` para esse mesmo registrador

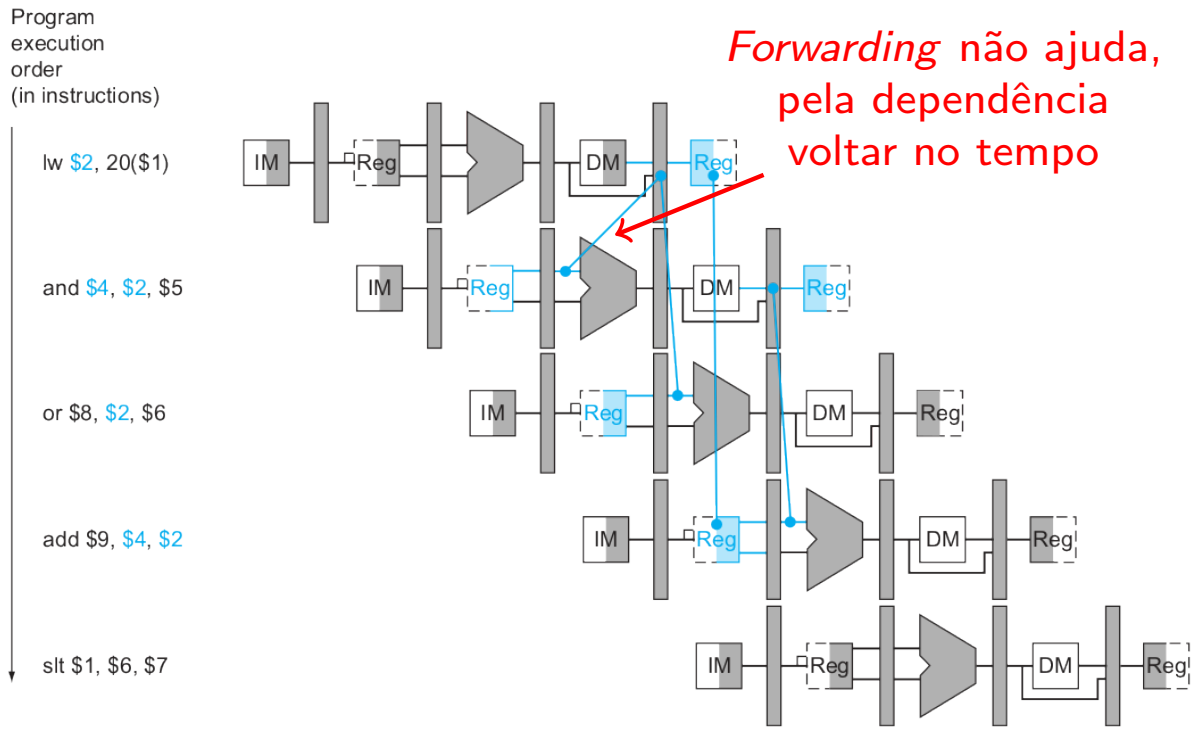


Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Vimos na aula passada que *forwarding* não evita todo problema
  - Como quando uma instrução lê um registrador em seguida a um `lw` para esse mesmo registrador



Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nesses casos nos resta apenas parar a *pipeline* até o conflito ser resolvido

Conflitos de Dados

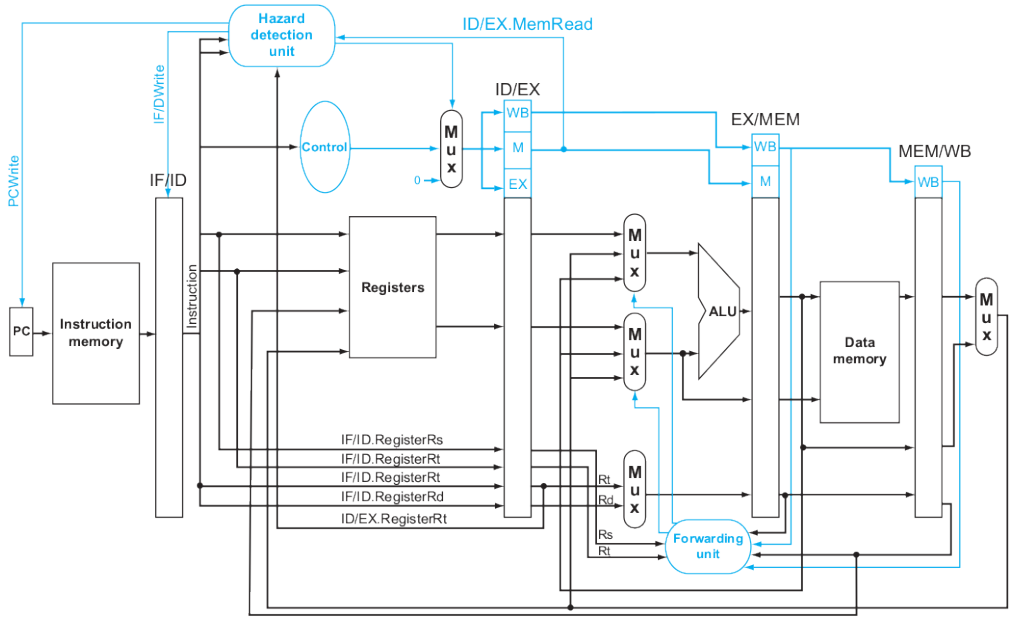
Pipeline Stalls

- Nesses casos nos resta apenas parar a *pipeline* até o conflito ser resolvido
  - E para isso, precisamos de uma **unidade de detecção de conflitos** (*Hazard Detection Unit*)

Conflitos de Dados

Pipeline Stalls

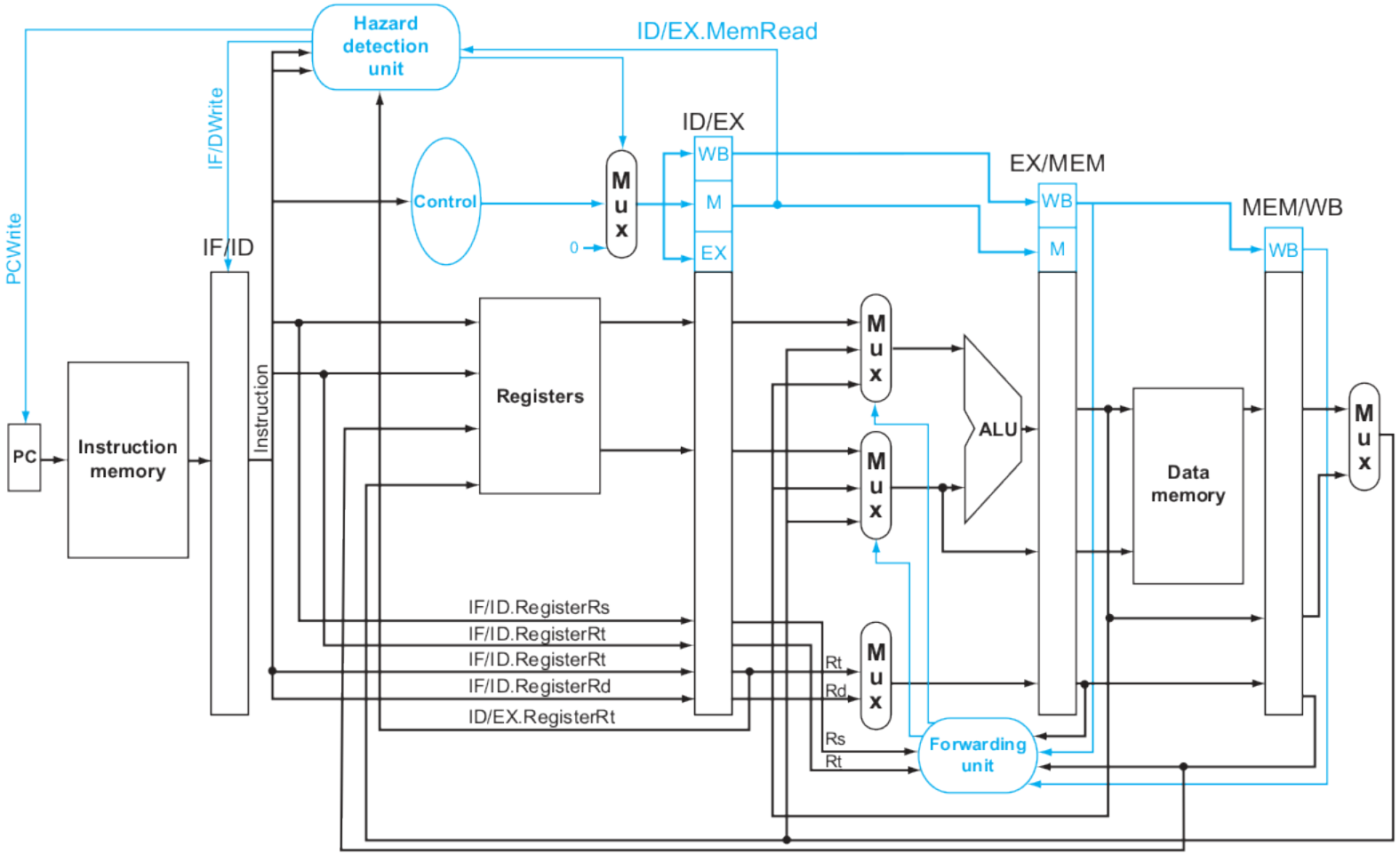
- Nesses casos nos resta apenas parar a *pipeline* até o conflito ser resolvido
  - E para isso, precisamos de uma **unidade de detecção de conflitos** (*Hazard Detection Unit*)
  - Posicionada em `ID`, para que ela possa introduzir a parada entre a carga e o uso de uma instrução



Fonte: [1]

Conflitos de Dados

Pipeline Stalls



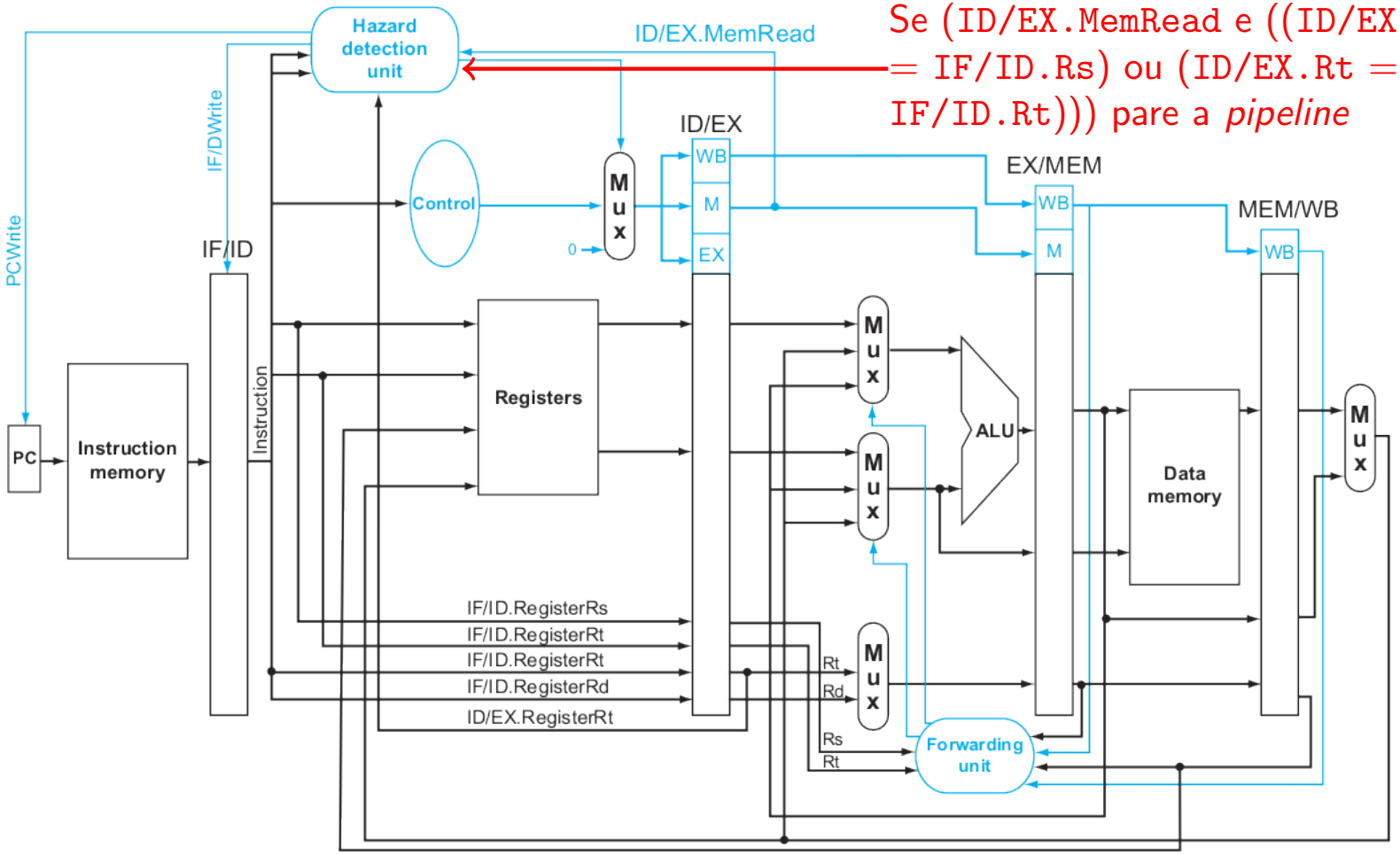
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



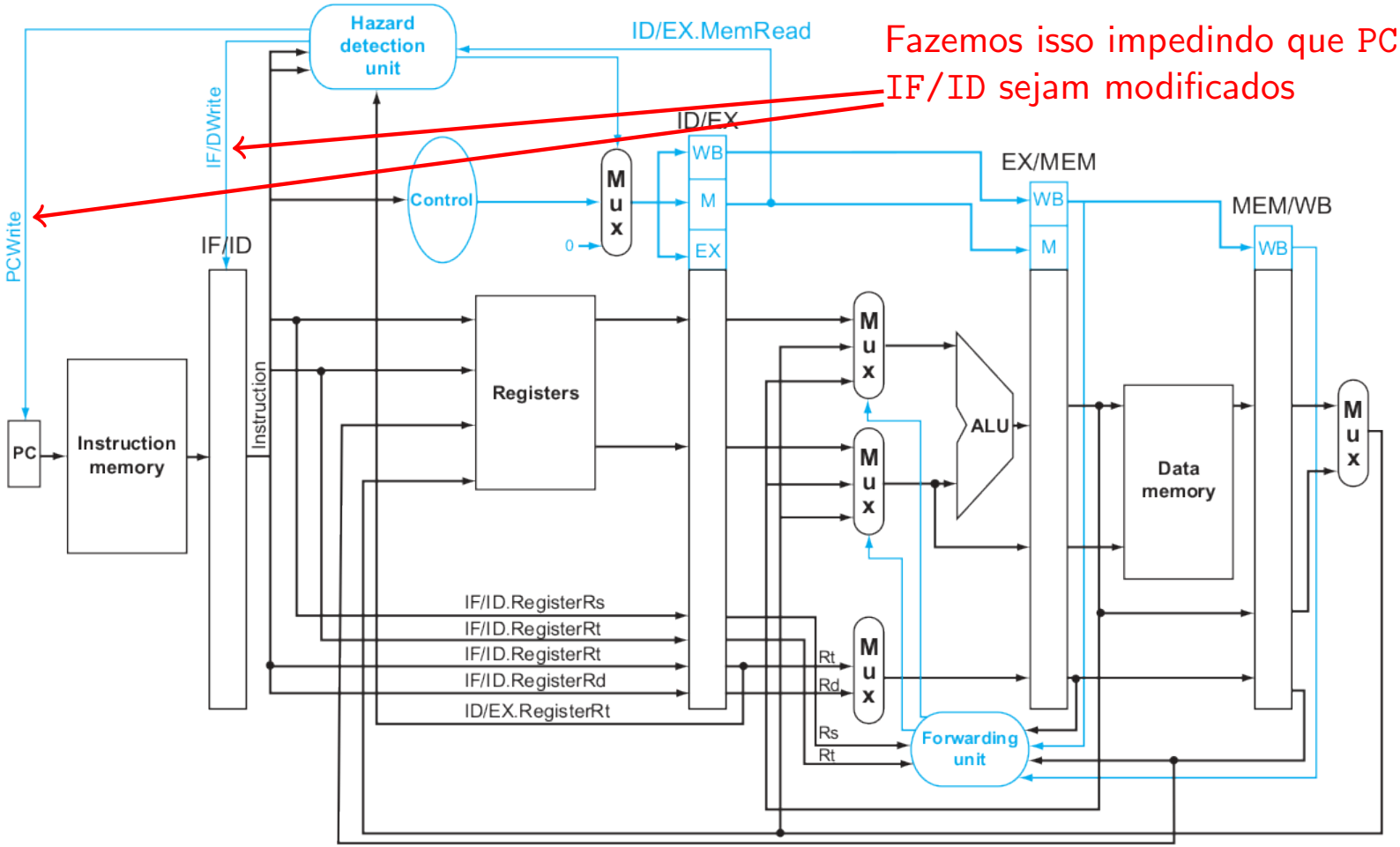
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



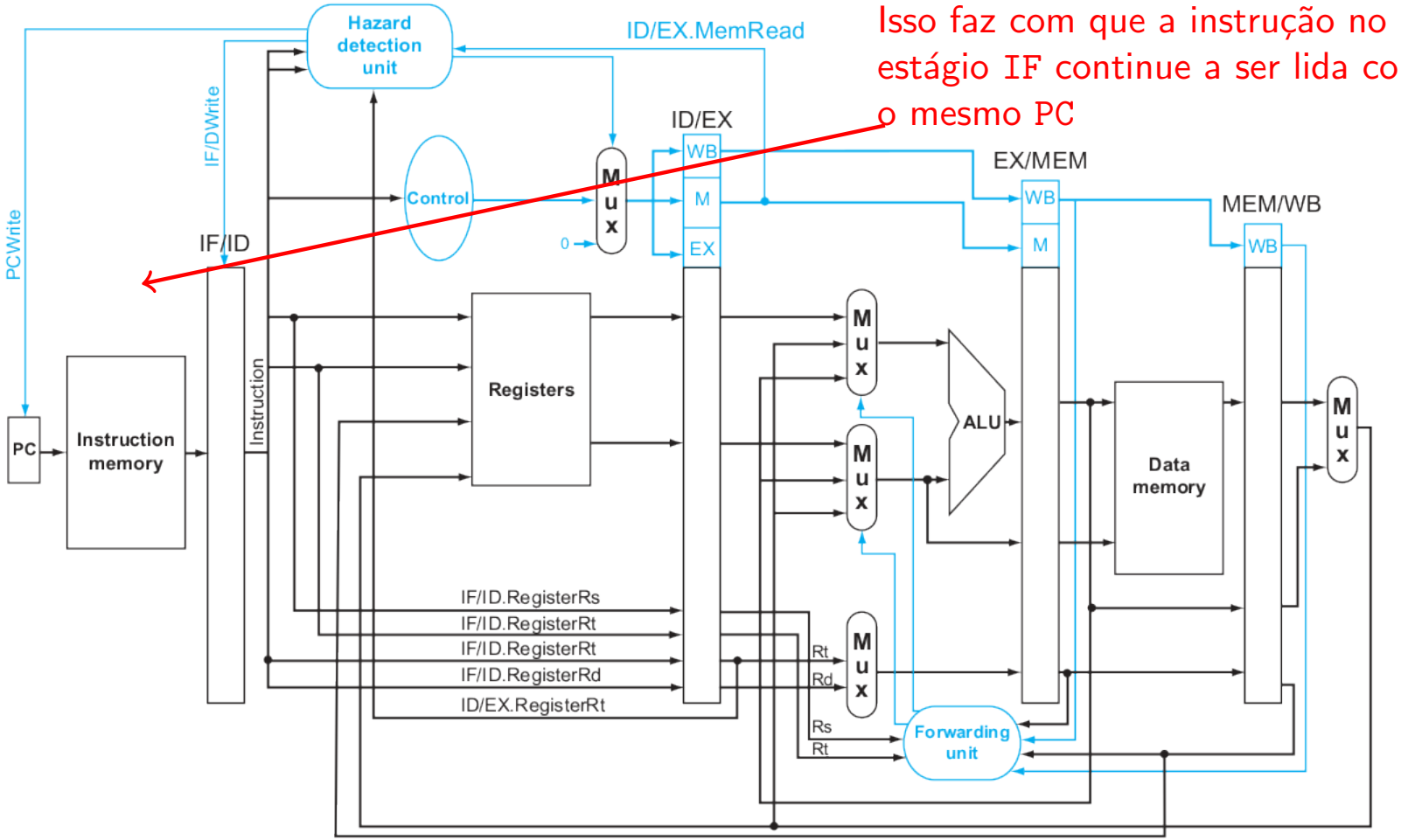
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



Isso faz com que a instrução no estágio `IF` continue a ser lida com o mesmo `PC`

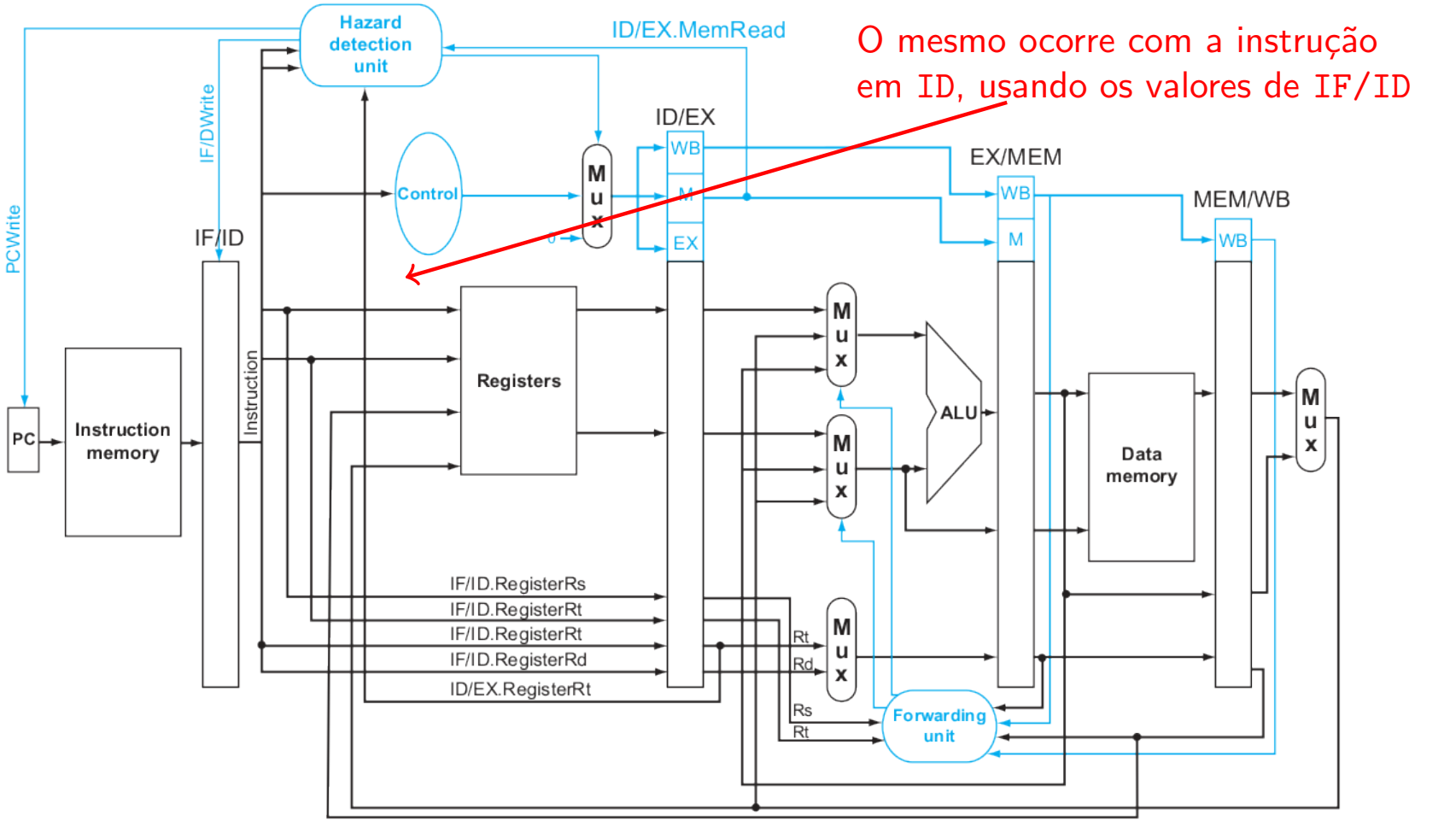
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



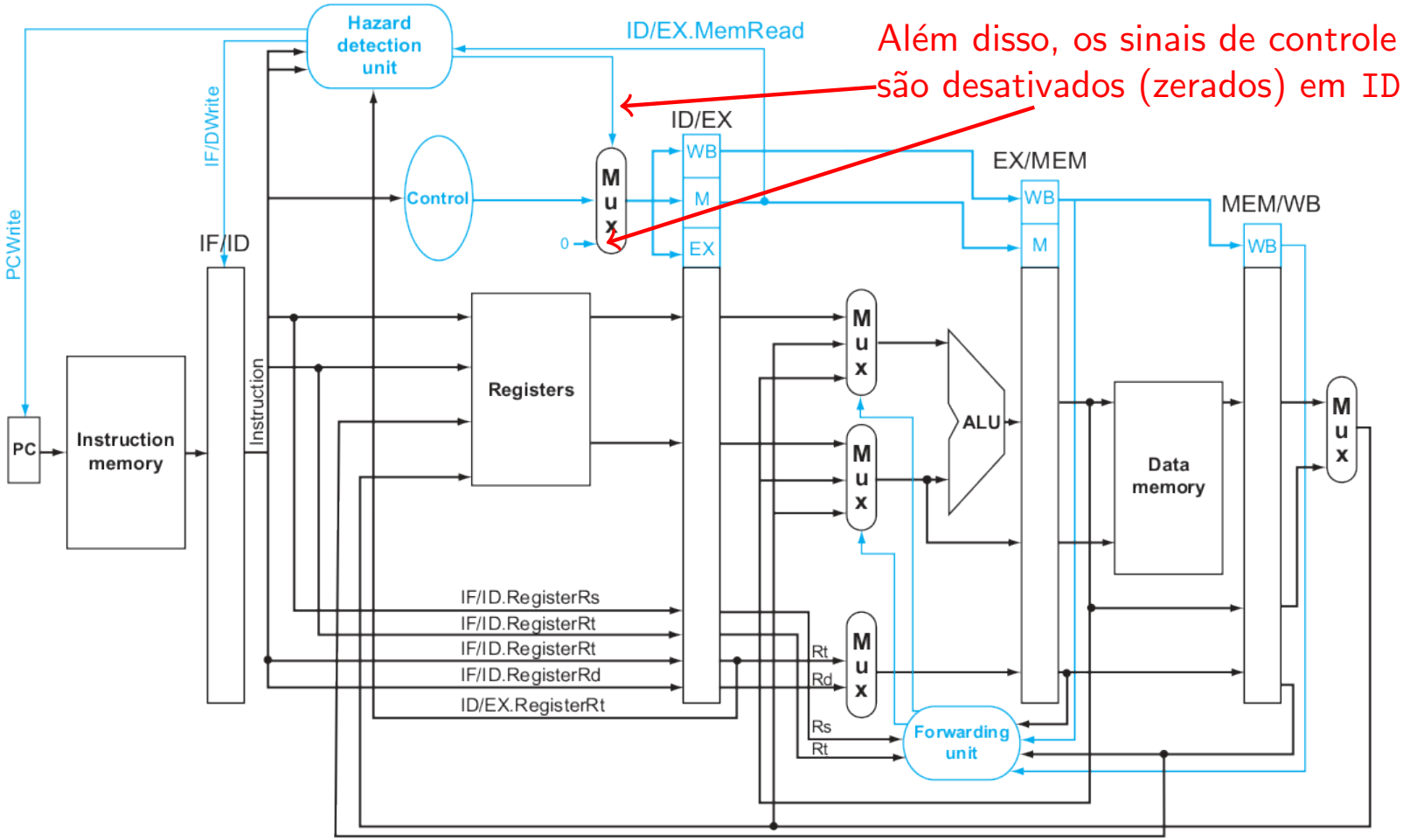
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



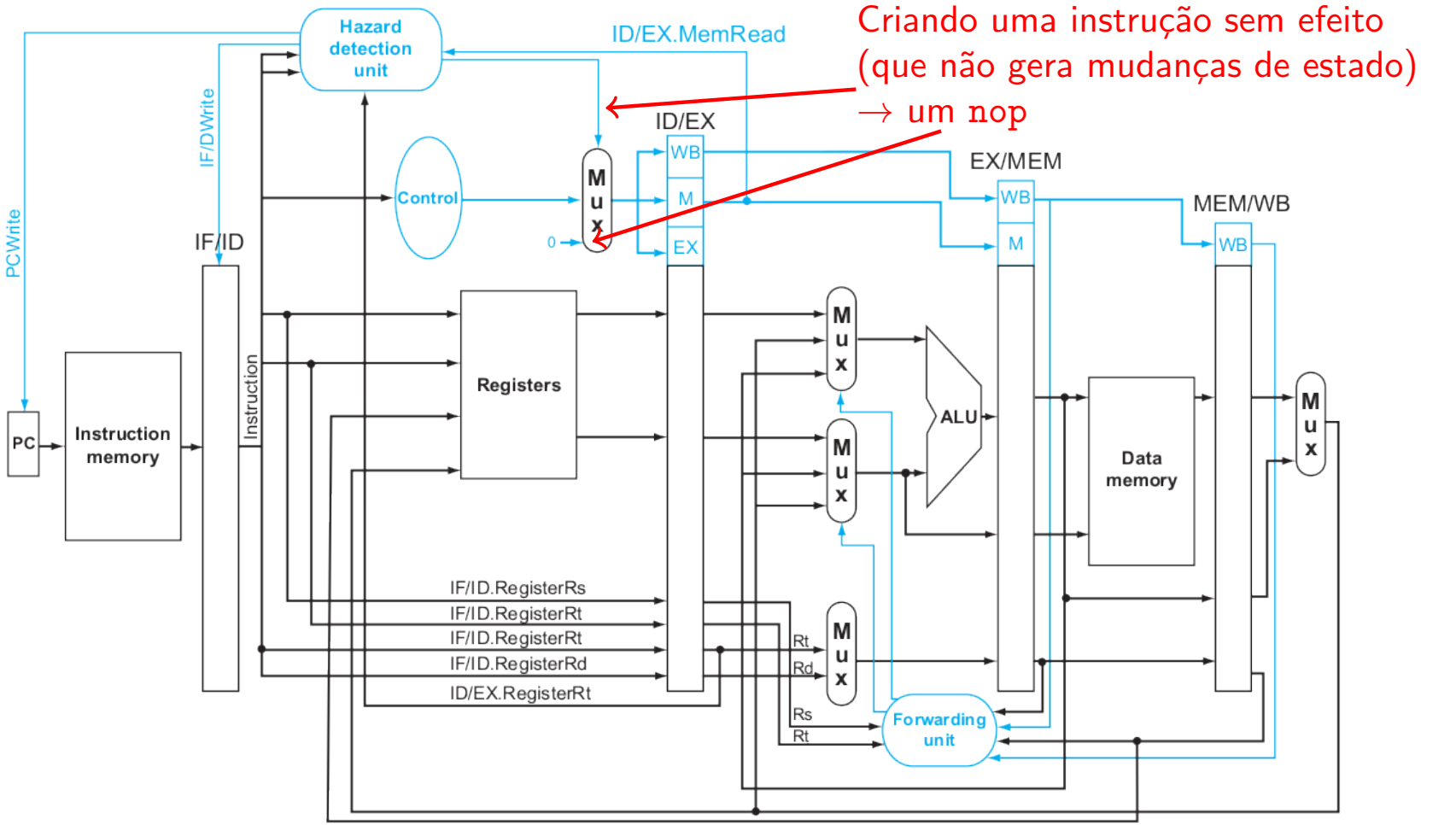
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



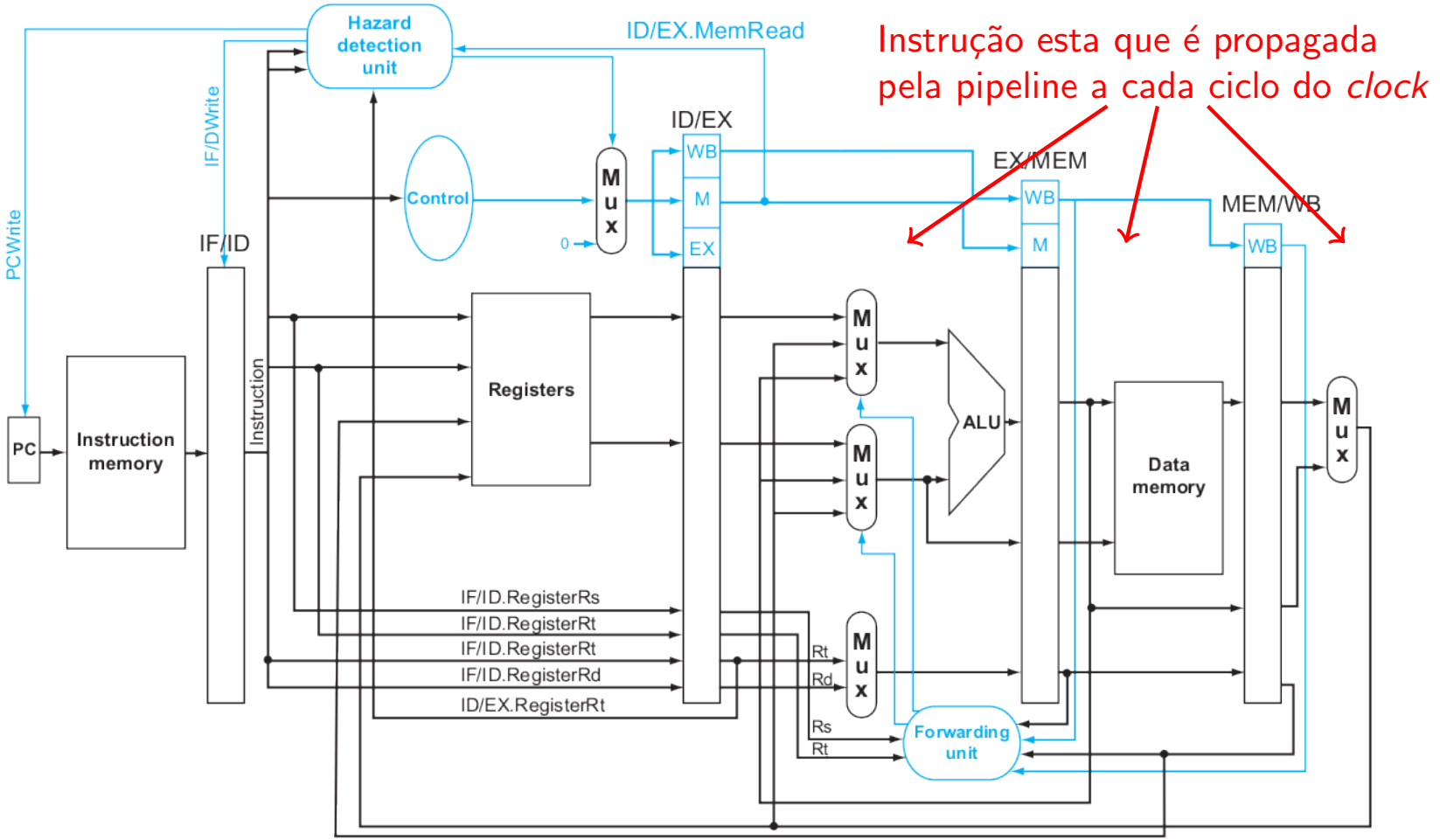
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



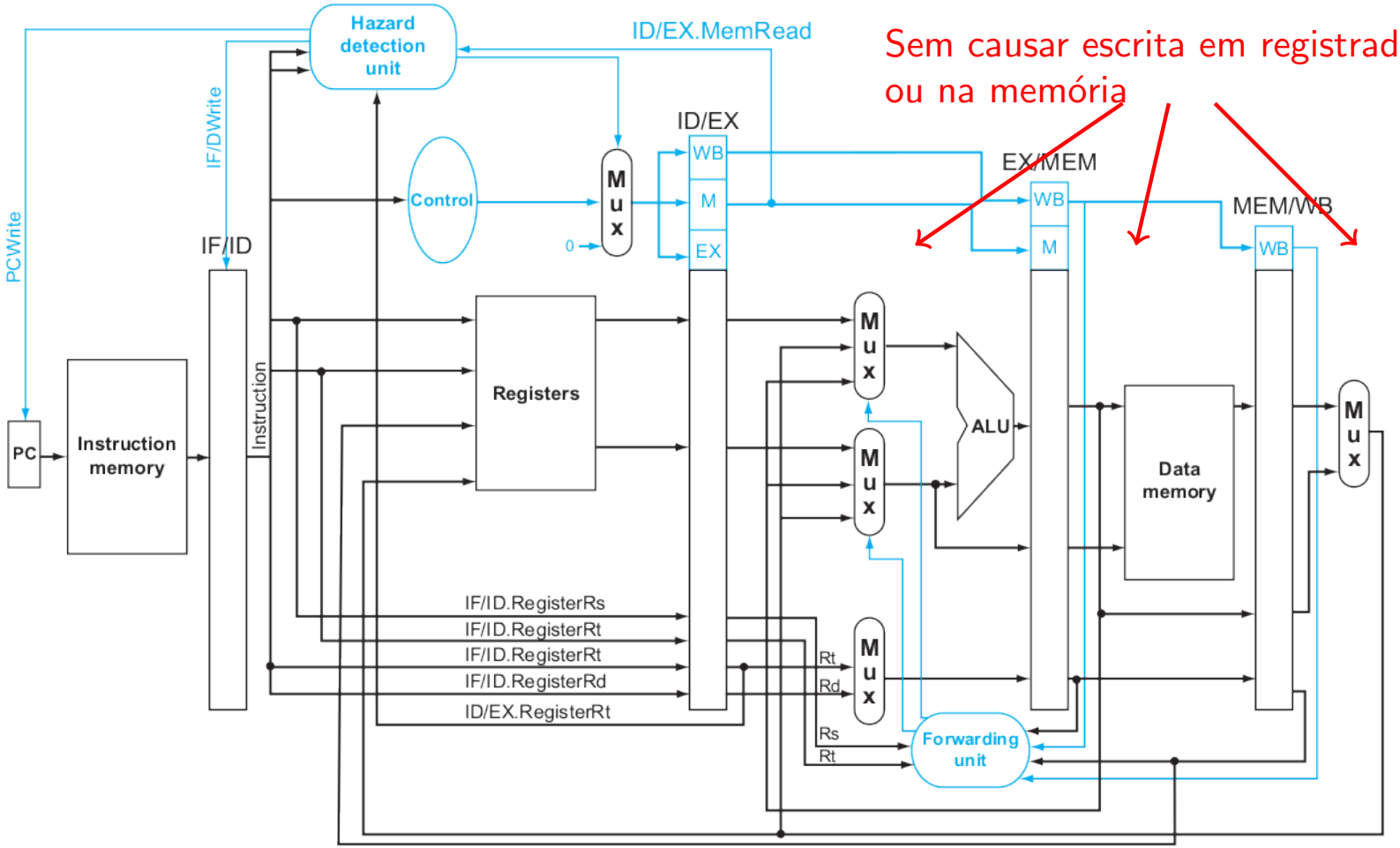
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



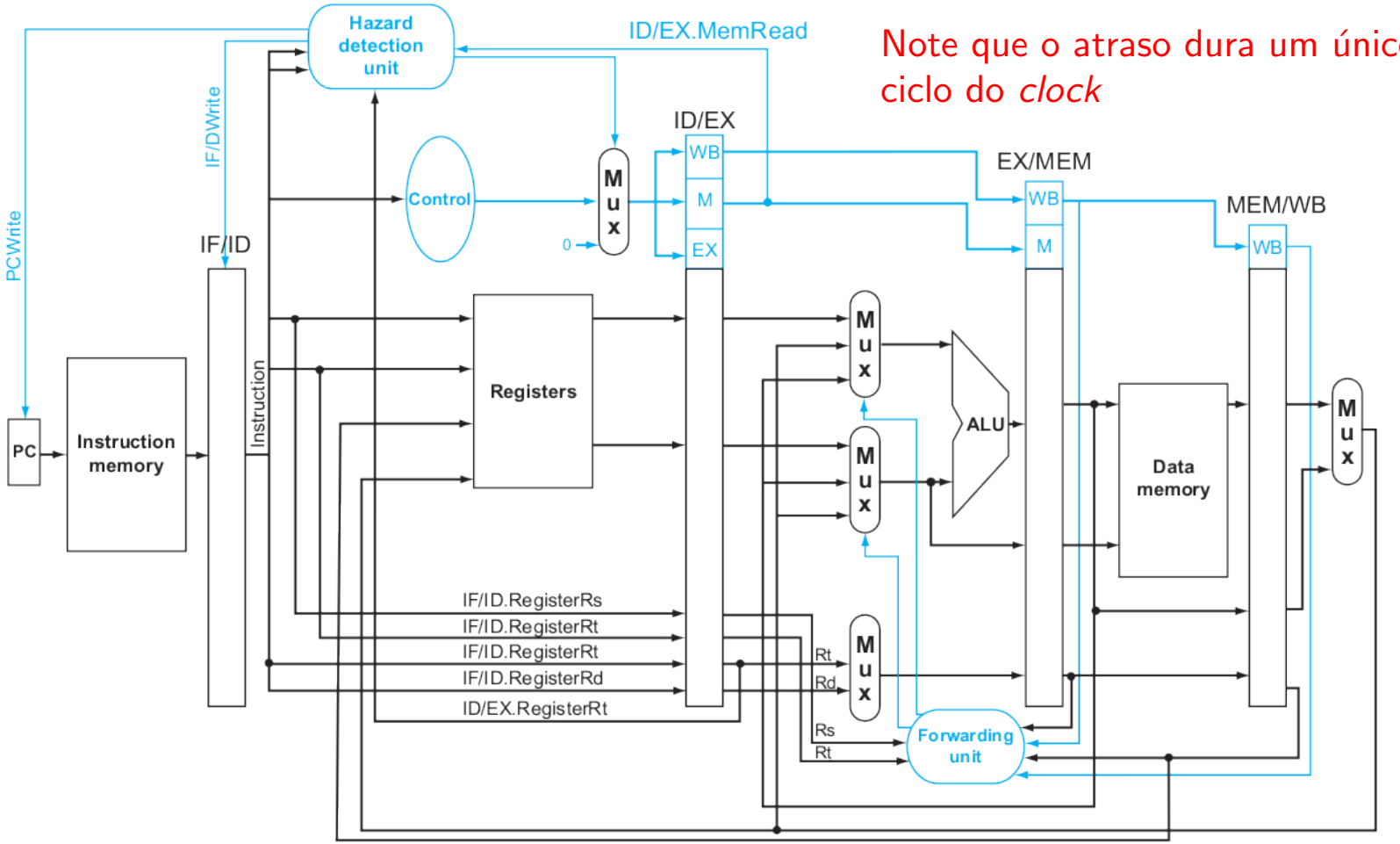
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



Note que o atraso dura um único ciclo do *clock*

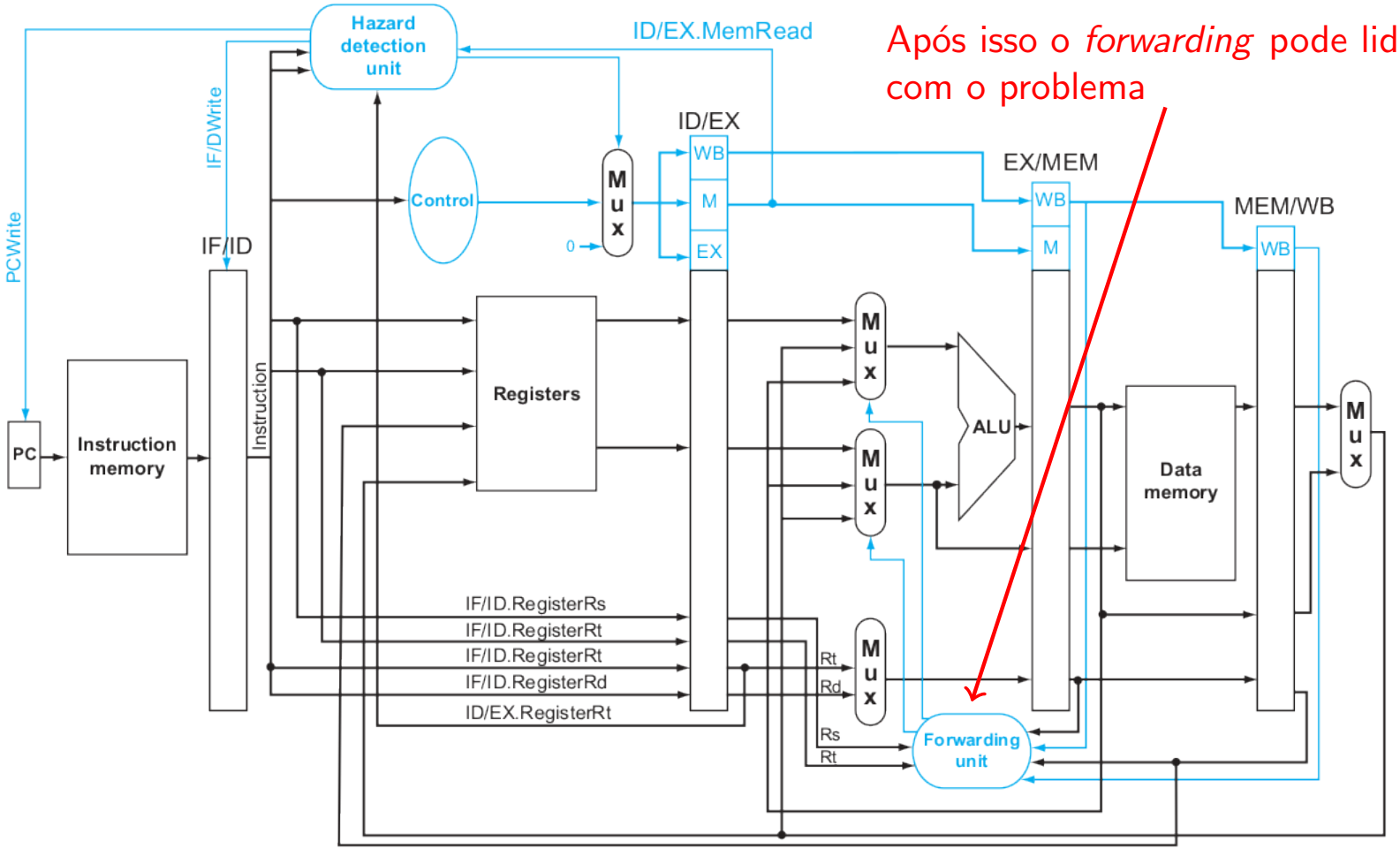
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



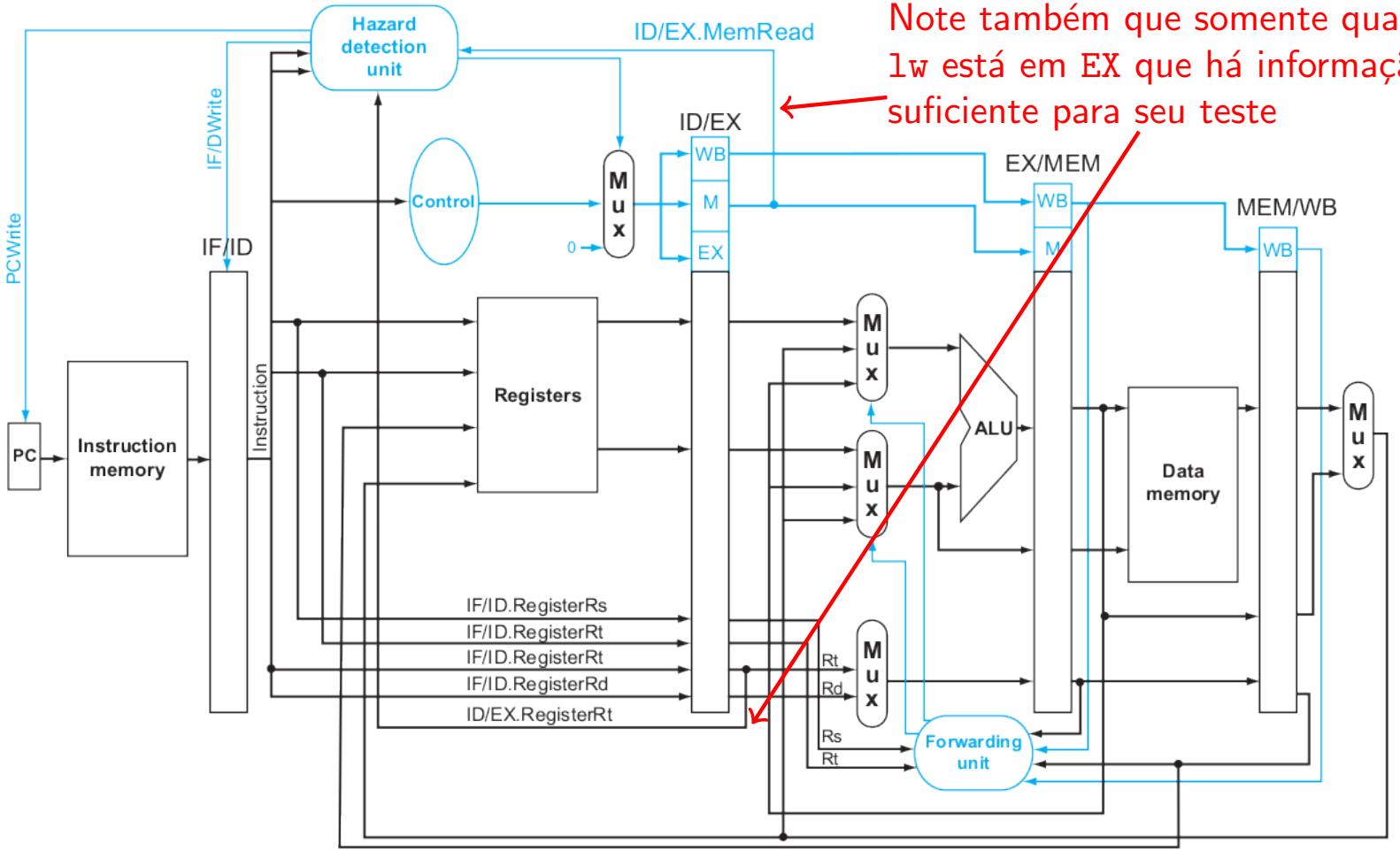
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



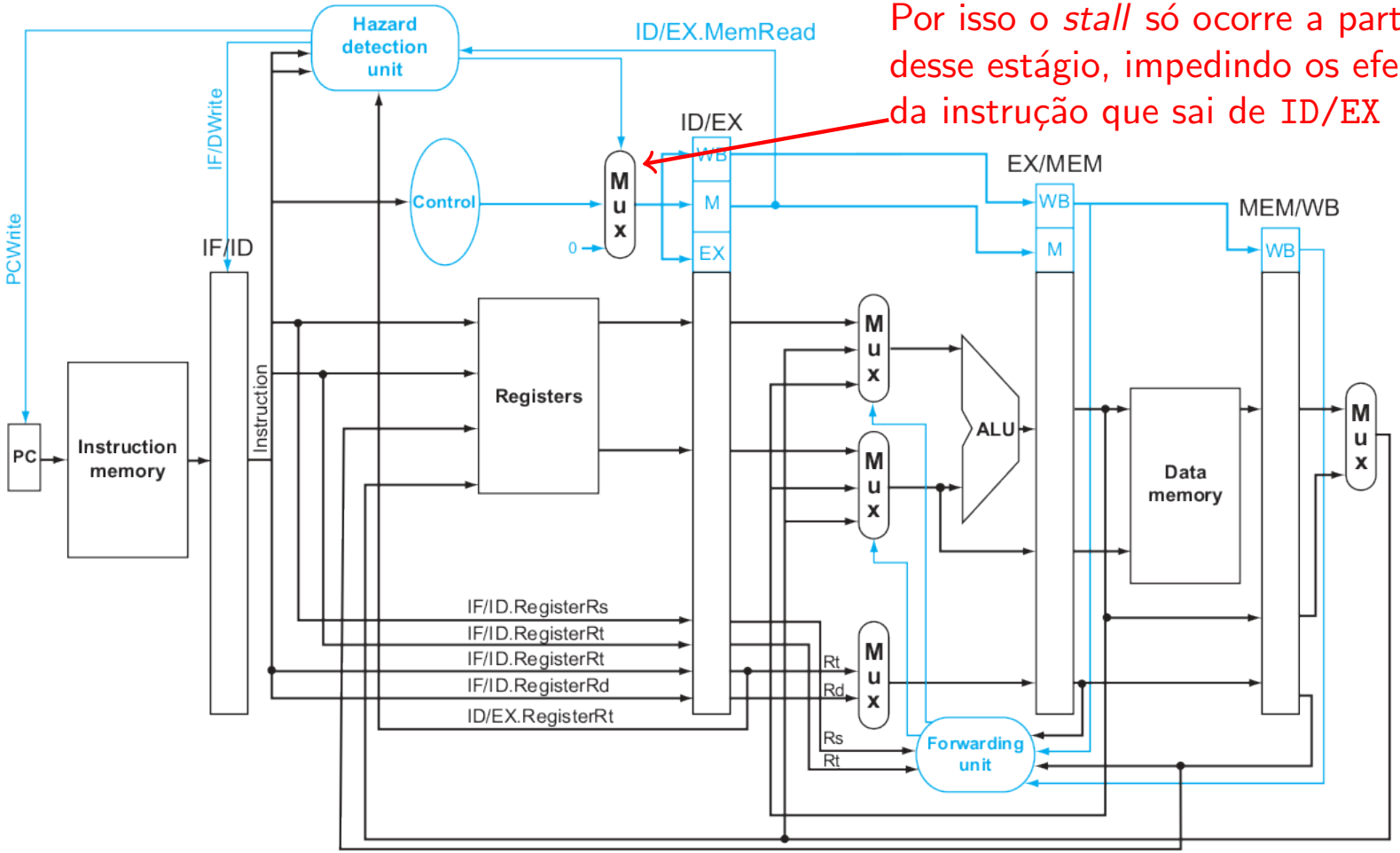
Fonte: Adaptado de [1]

Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls



Por isso o *stall* só ocorre a partir desse estágio, impedindo os efeitos da instrução que sai de `ID/EX`

Fonte: Adaptado de [1]

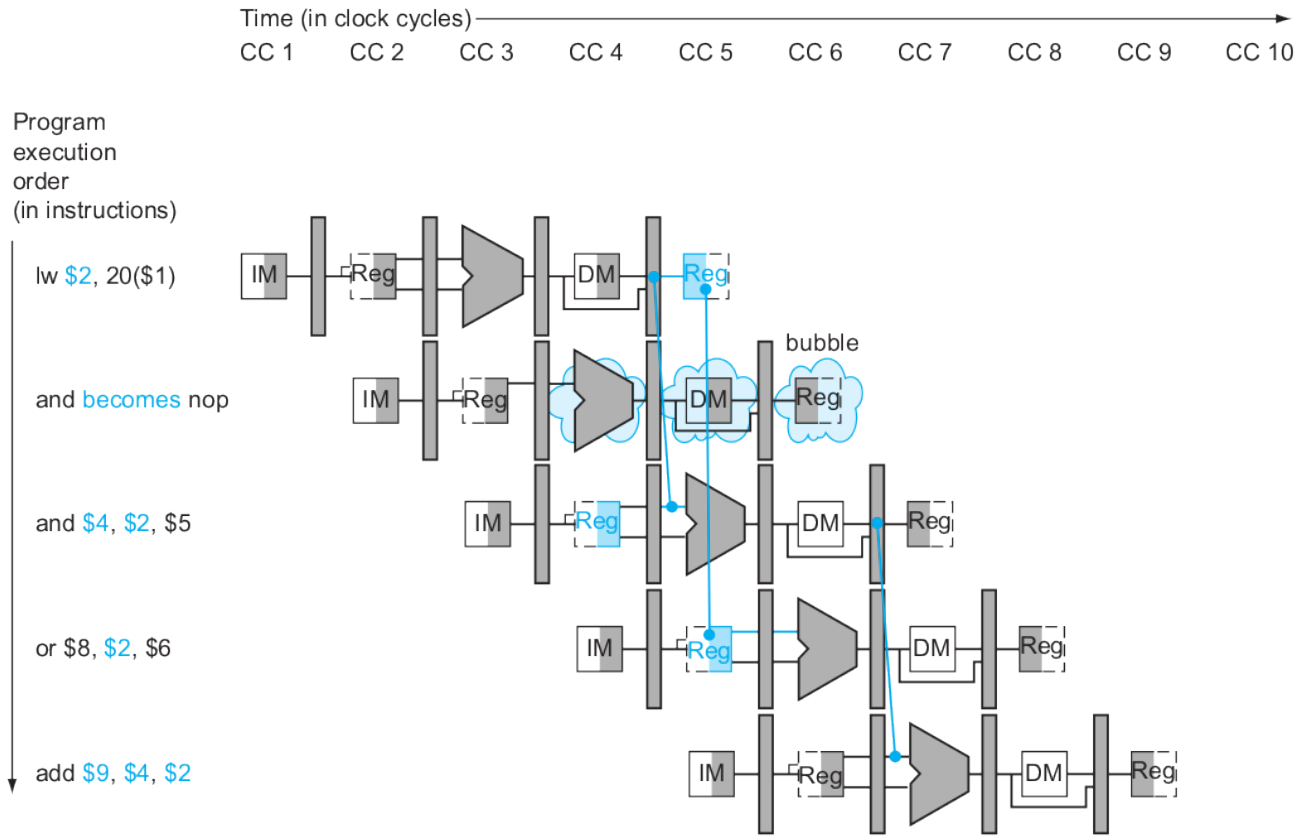
Tipo-I:

op	rs	rt	deslocamento

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica

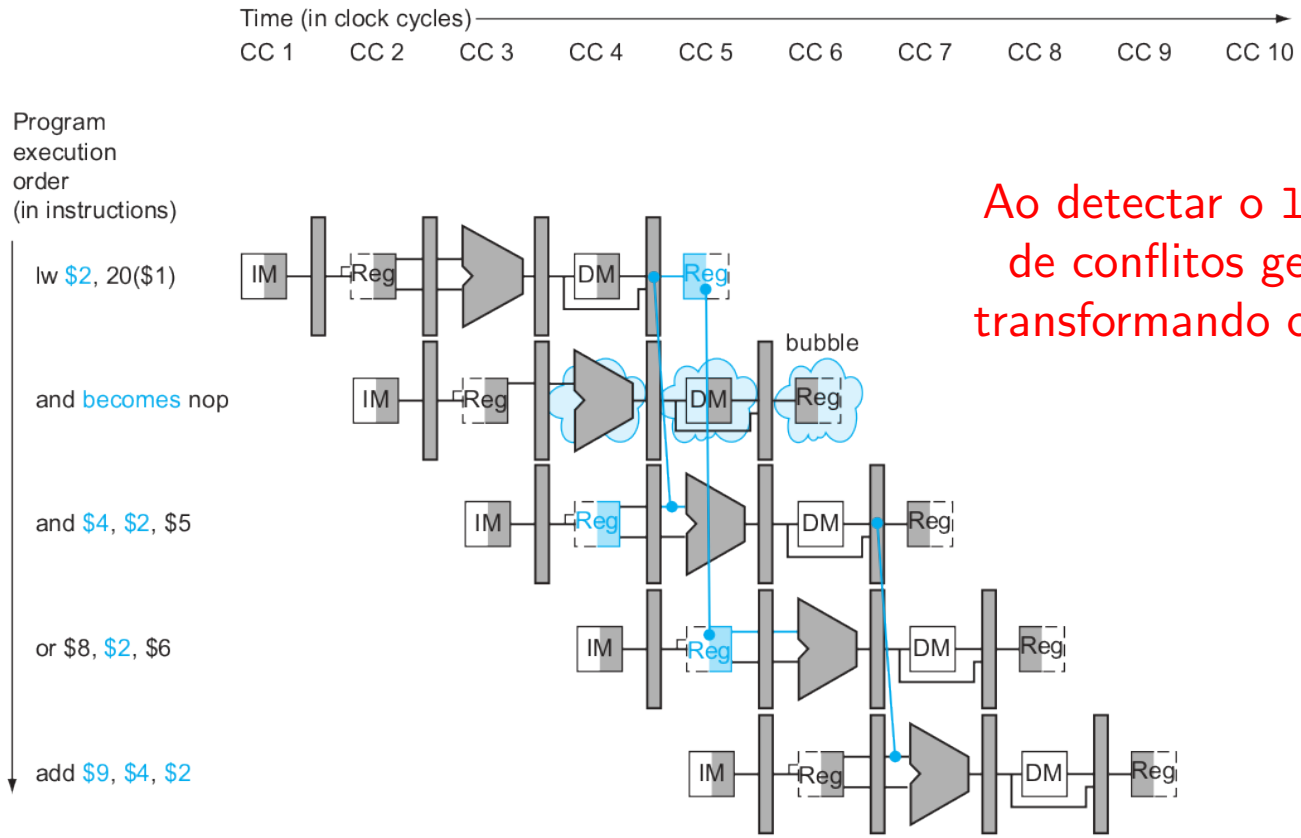


Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica



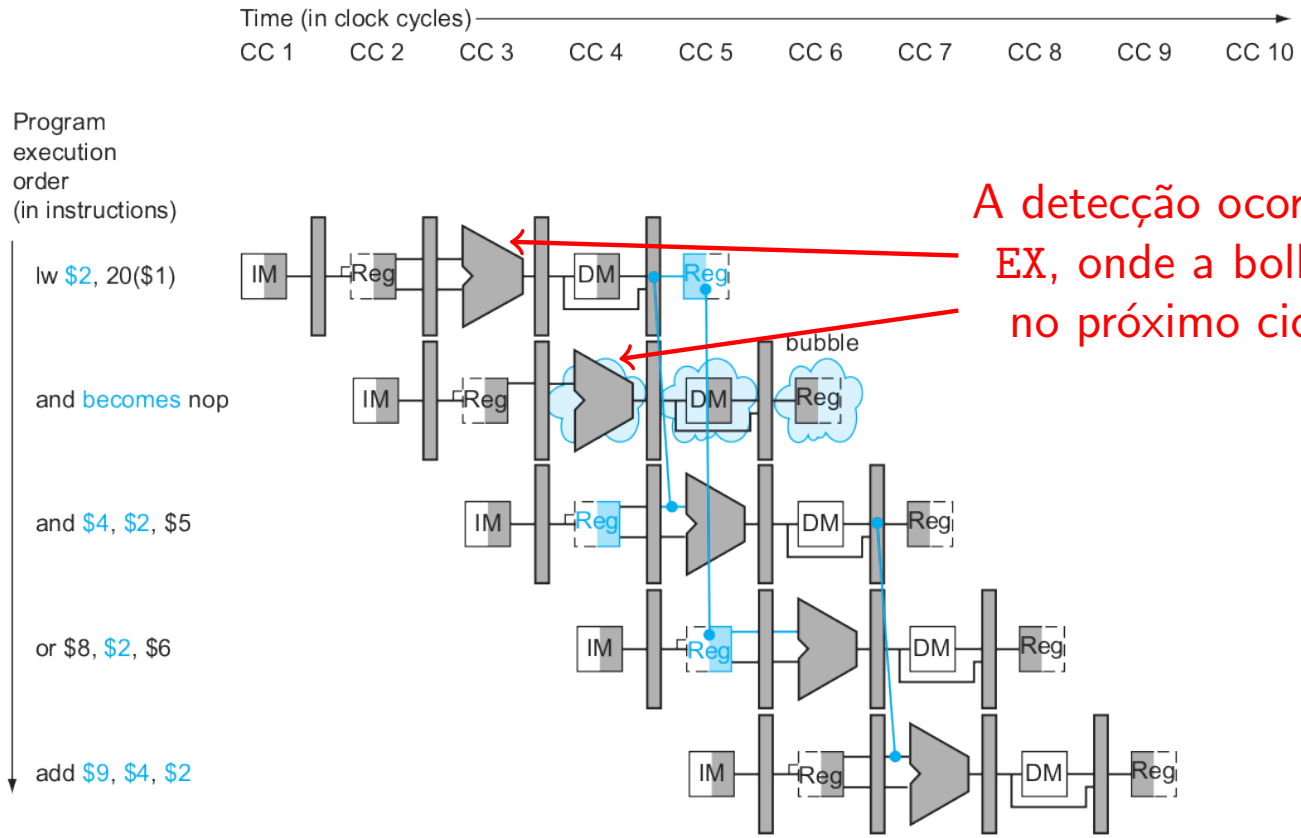
Ao detectar o `lw`, o controle de conflitos gera um *stall*, transformando o `and` em `nop`

Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica



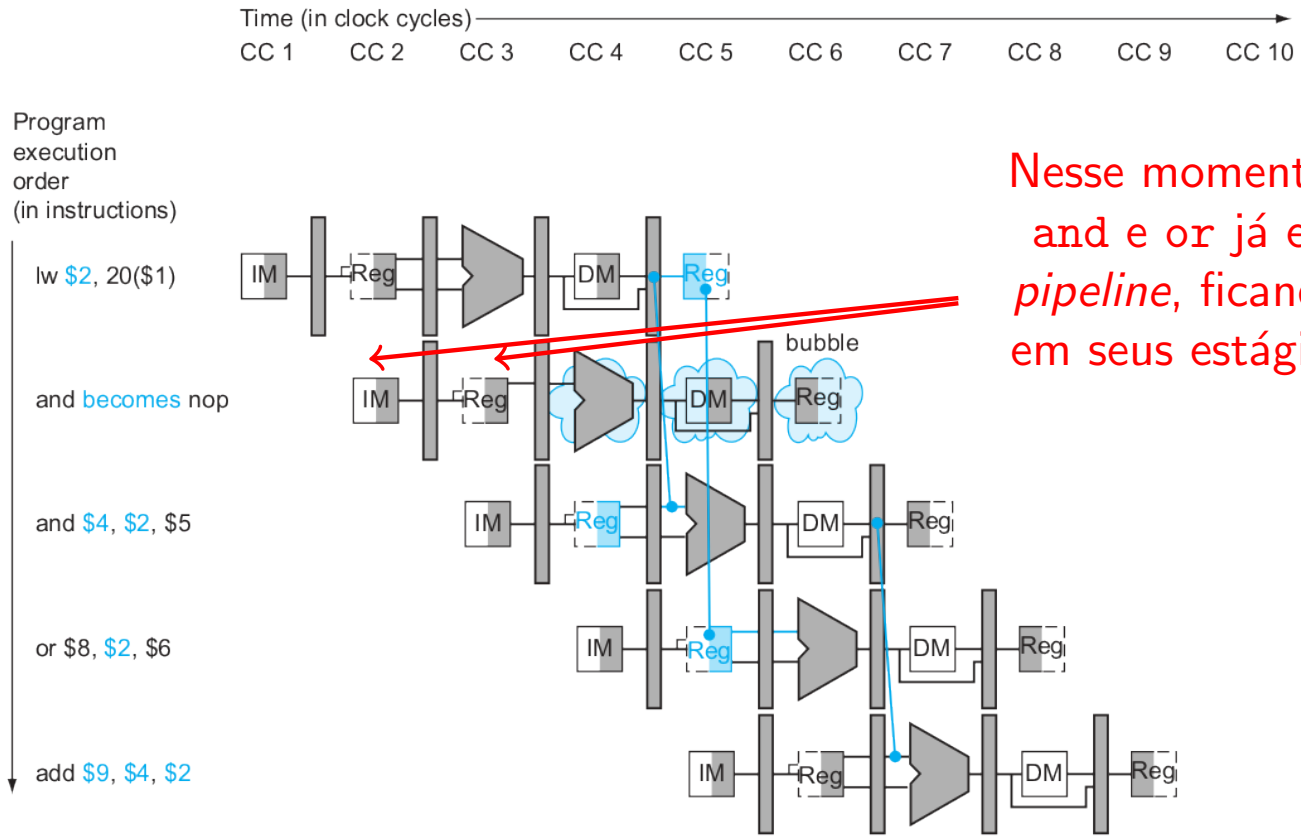
A detecção ocorre apenas em `EX`, onde a bolha é inserida no próximo ciclo do *clock*

Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica



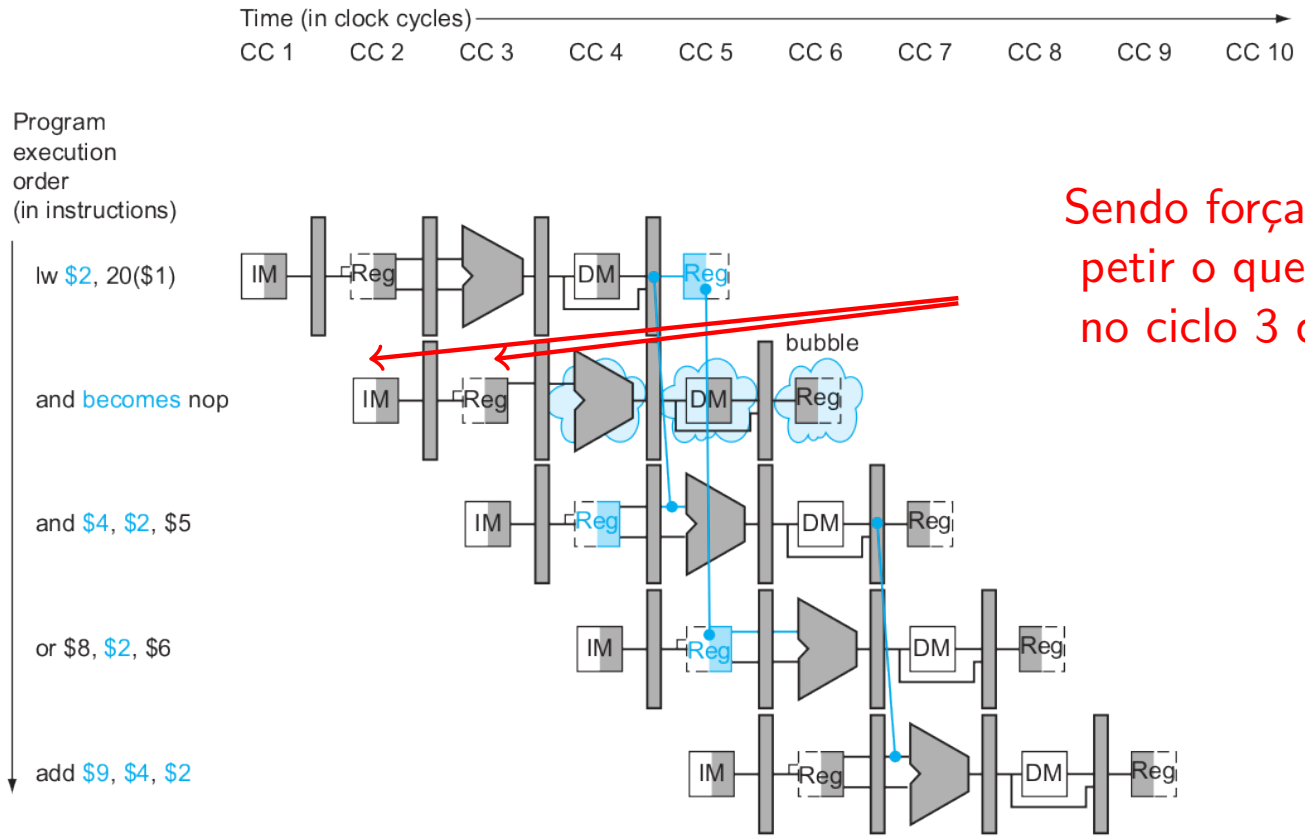
Nesse momento, contudo, `and` e `or` já estavam na *pipeline*, ficando "presos" em seus estágios (ciclo 4)

Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica



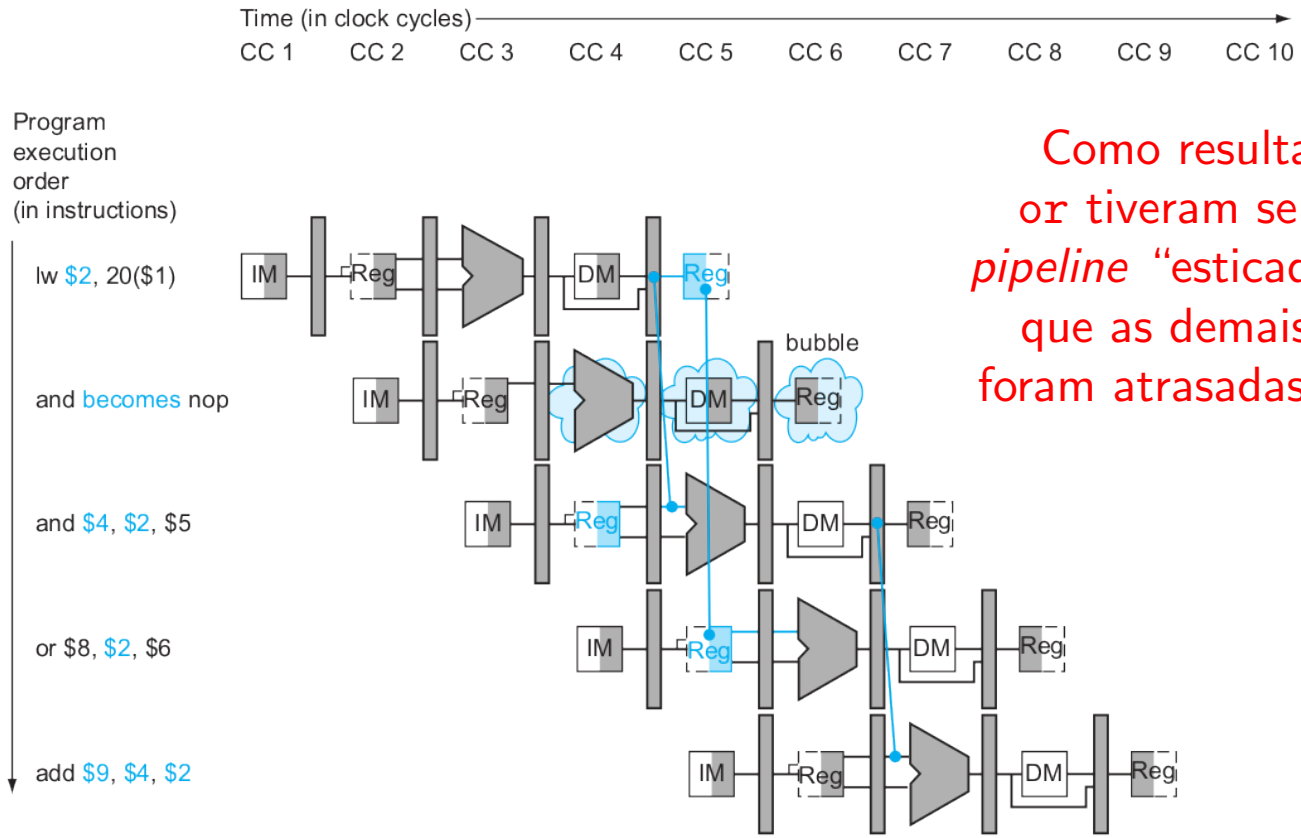
Sendo forçados a repetir o que fizeram no ciclo 3 do *clock*

Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica



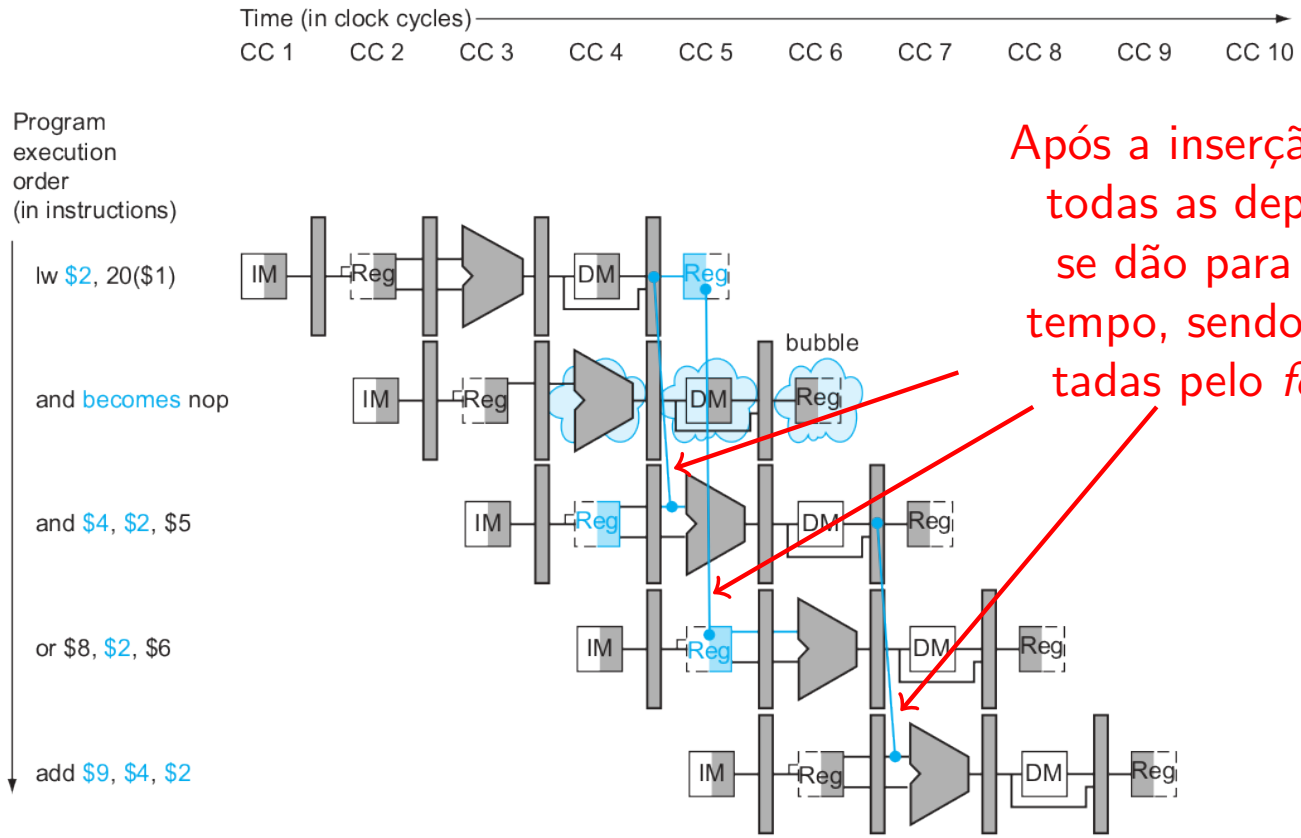
Como resultado, `and` e `or` tiveram seu tempo de *pipeline* "esticado", enquanto que as demais instruções foram atrasadas em um ciclo

Fonte: [1]

Conflitos de Dados

Pipeline Stalls

- Nosso exemplo então fica



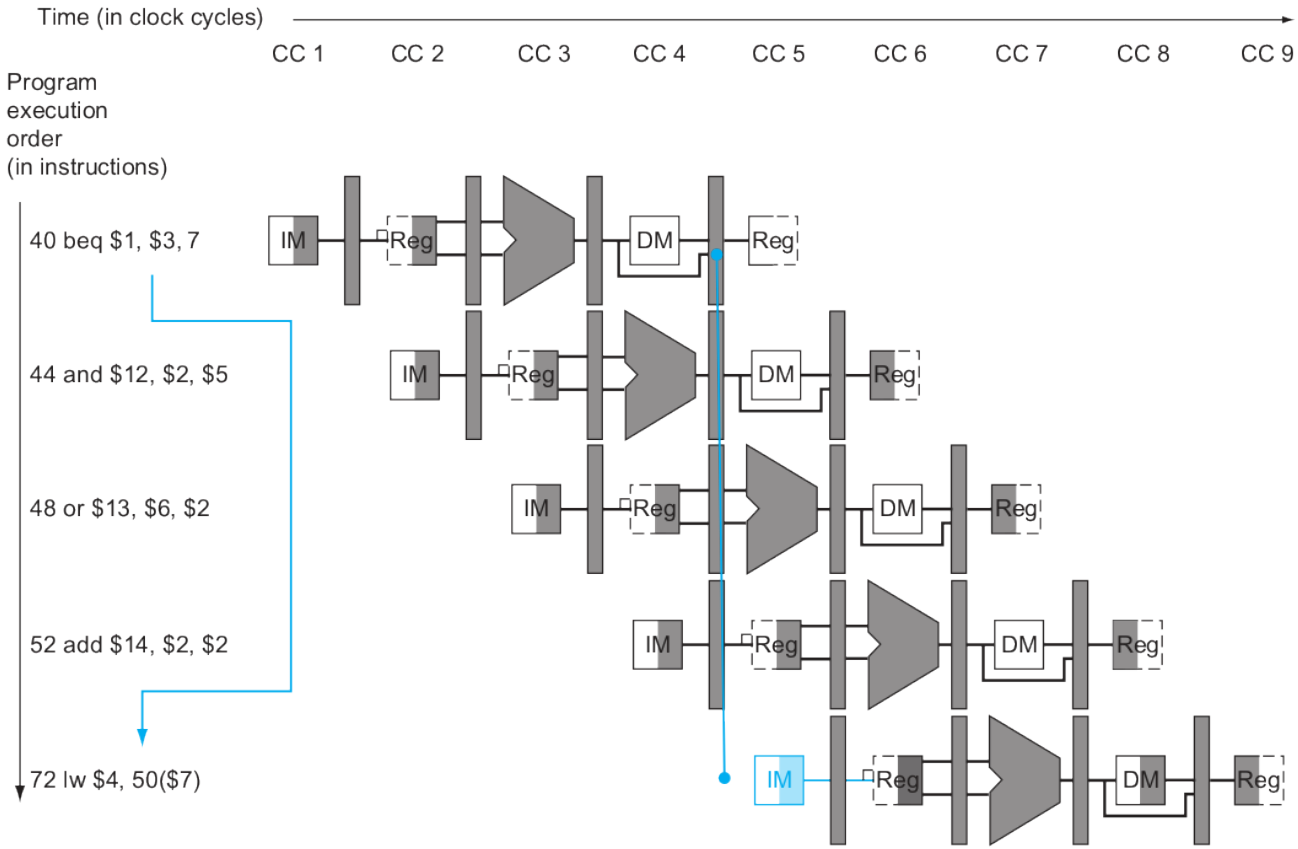
Após a inserção da bolha, todas as dependências se dão para frente no tempo, sendo então tratadas pelo *forwarding*

Fonte: [1]

Conflitos de Controle

Branches

- Considere o seguinte código na *pipeline*:

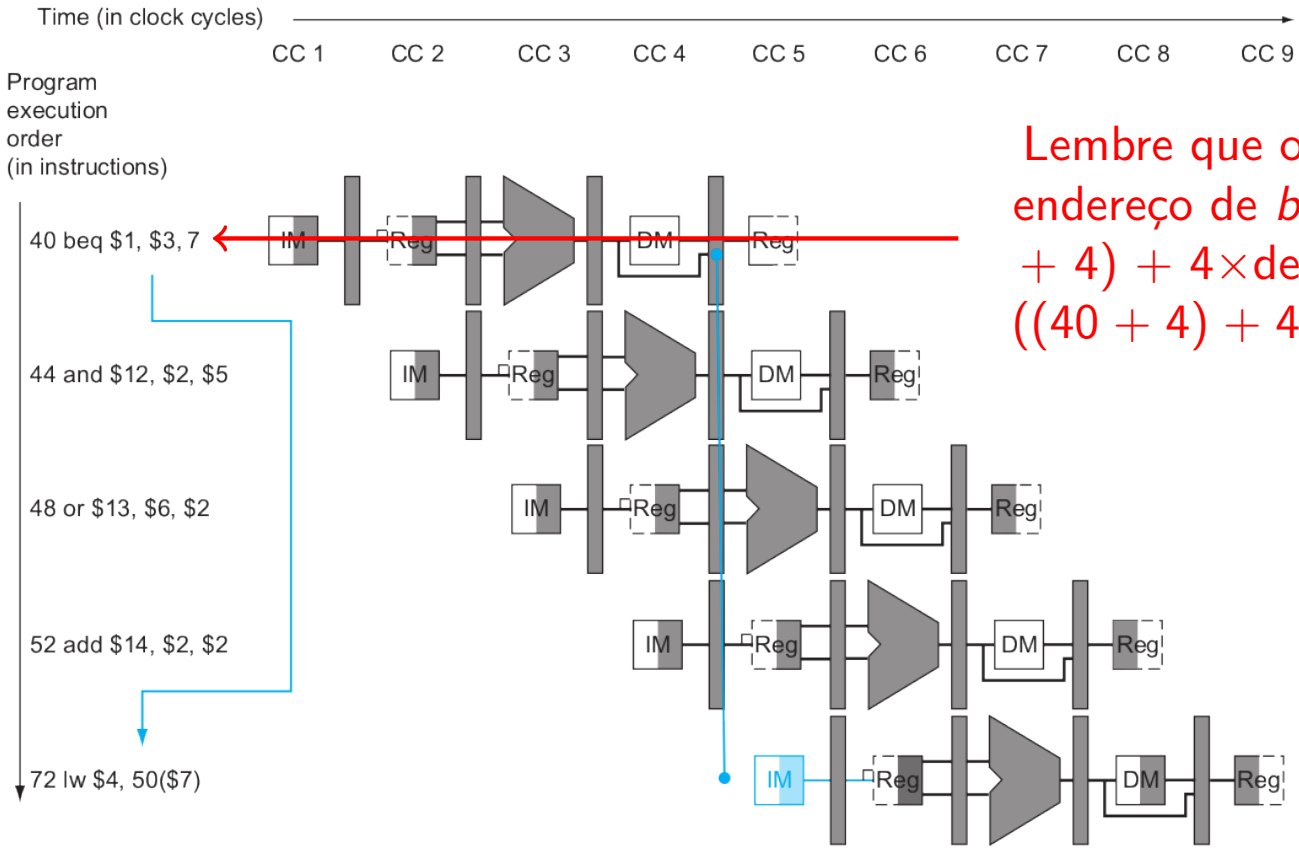


Fonte: Adaptado de [1]

Conflitos de Controle

Branches

- Considere o seguinte código na *pipeline*:



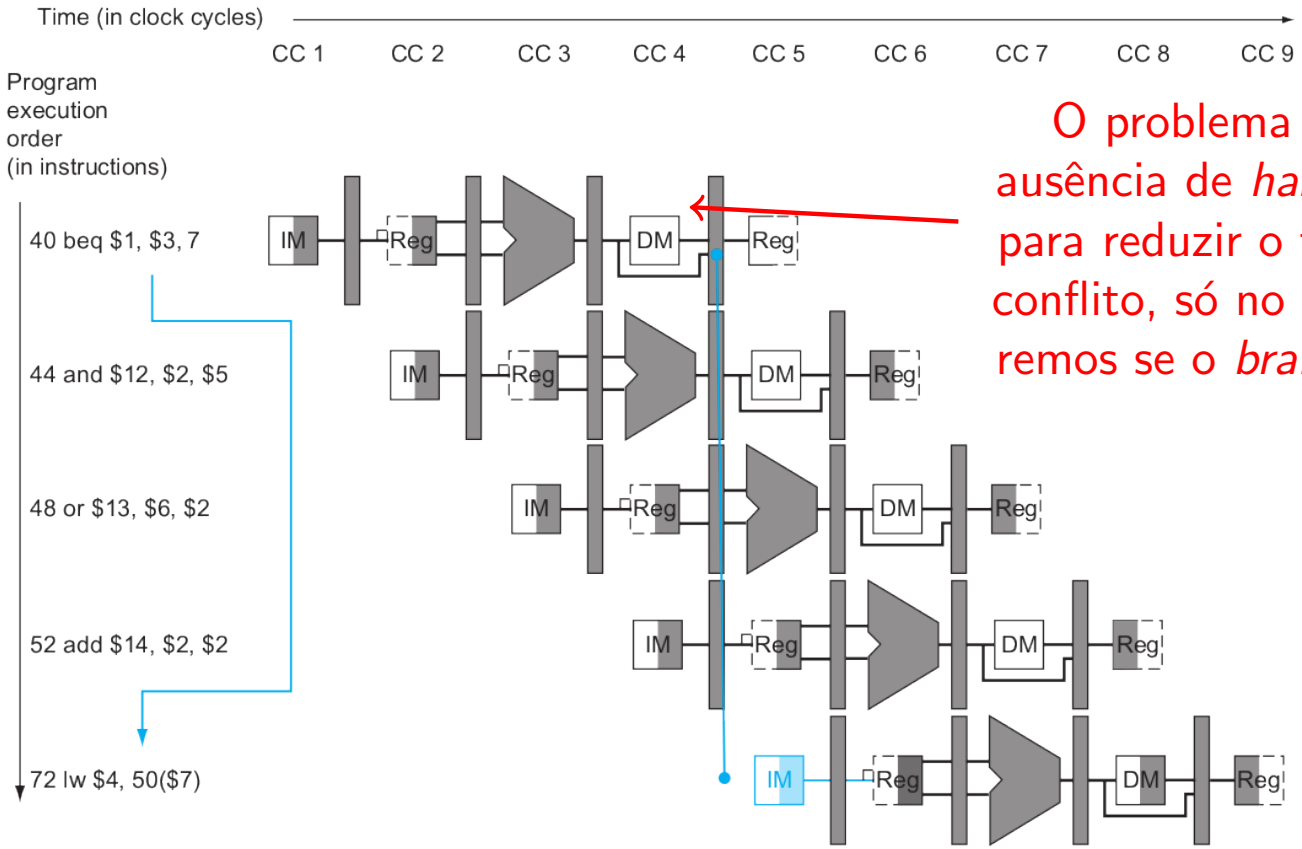
Lembre que o cálculo do endereço de *branch* é (PC + 4) + 4×deslocamento ((40 + 4) + 4 × 7 = 72)

Fonte: Adaptado de [1]

Conflitos de Controle

Branches

- Considere o seguinte código na *pipeline*:



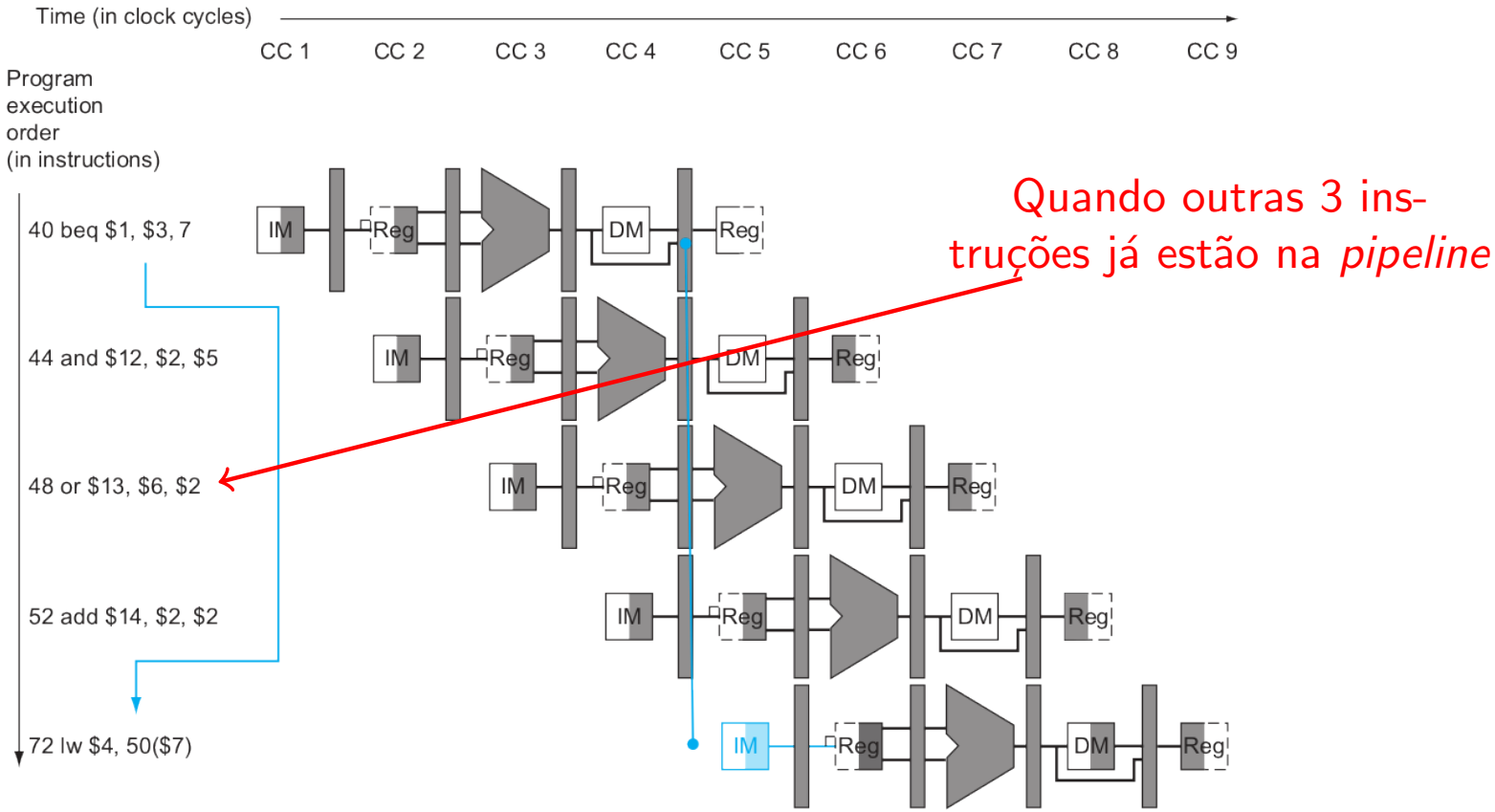
O problema é que, na ausência de *hardware* extra para reduzir o tempo desse conflito, só no ciclo 4 saberemos se o *branch* ocorrerá

Fonte: Adaptado de [1]

Conflitos de Controle

Branches

- Considere o seguinte código na *pipeline*:



Fonte: Adaptado de [1]

Conflitos de Controle

Branches

- Infelizmente, não há um equivalente ao *forwarding* para conflitos de controle

Conflitos de Controle

Branches

- Infelizmente, não há um equivalente ao *forwarding* para conflitos de controle
- Que fazer então?

Conflitos de Controle

Branches

- Infelizmente, não há um equivalente ao *forwarding* para conflitos de controle
- Que fazer então?
  - Assumir que o *branch* não ocorrerá
  - Reduzir o atraso de *branches*
  - Predição dinâmica de *branches*

Conflitos de Controle

Branches: Assumindo que o *branch* não ocorrerá

- Continuamos a execução, supondo que o *branch* não ocorrerá
  - Se ele ocorrer, precisamos descartar as instruções processadas indevidamente
  - A execução continua então no alvo do *branch*

Conflitos de Controle

Branches: Assumindo que o *branch* não ocorrerá

- Continuamos a execução, supondo que o *branch* não ocorrerá
  - Se ele ocorrer, precisamos descartar as instruções processadas indevidamente
  - A execução continua então no alvo do *branch*
- E como descartar uma instrução?
  - Mudando seus valores de controle para 0s (como num *stall*), nos estágios `IF`, `ID` e `EX`, quando o *branch* chegar a `MEM`
  - Processo conhecido como **descarga** (*flush*)

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- Movemos o tratamento de *branches* para antes de `MEM`
  - Assim menos instruções precisam ser removidas

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- Movemos o tratamento de *branches* para antes de `MEM`
  - Assim menos instruções precisam ser removidas
- Fazemos isso apenas para testes simples, que não necessitam de uma ALU completa
  - Como testes de igualdade e sinal, por exemplo
  - Bastante comuns em código

Conflitos de Controle

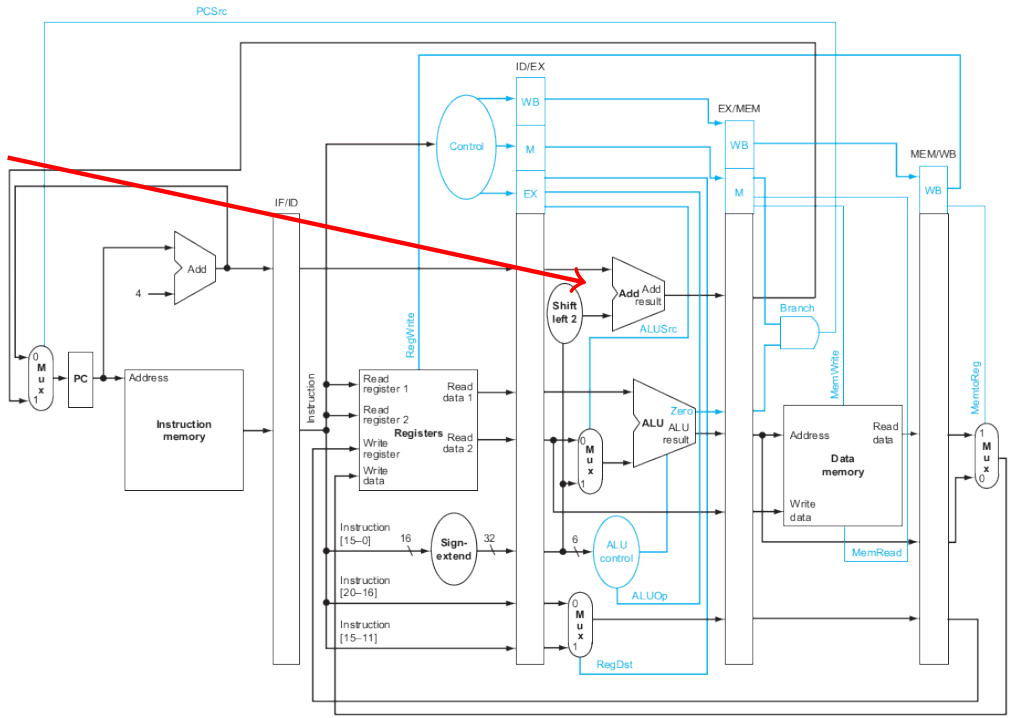
Branches: Reduzindo o atraso de *branches*

- Movemos o tratamento de *branches* para antes de `MEM`
  - Assim menos instruções precisam ser removidas
- Fazemos isso apenas para testes simples, que não necessitam de uma ALU completa
  - Como testes de igualdade e sinal, por exemplo
  - Bastante comuns em código
- Se testes mais complexos são necessários, usamos uma instrução separada, para uso da ALU

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- Para isso, adiantamos o cálculo o endereço-alvo do *branch*
  - Fácil, pois temos o `PC` e o campo imediato da instrução no registrador `IF/ID`
  - Então podemos mover o cálculo do endereço de `EX` para `ID`

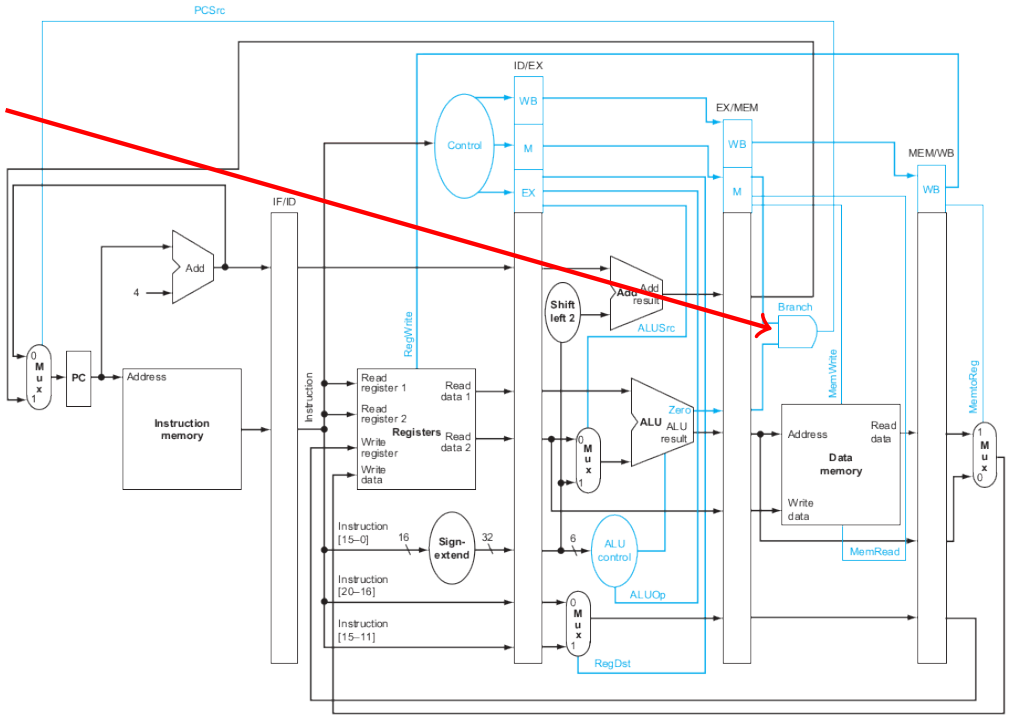


Fonte: [1]

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- O problema é definir se haverá ou não o *branch*

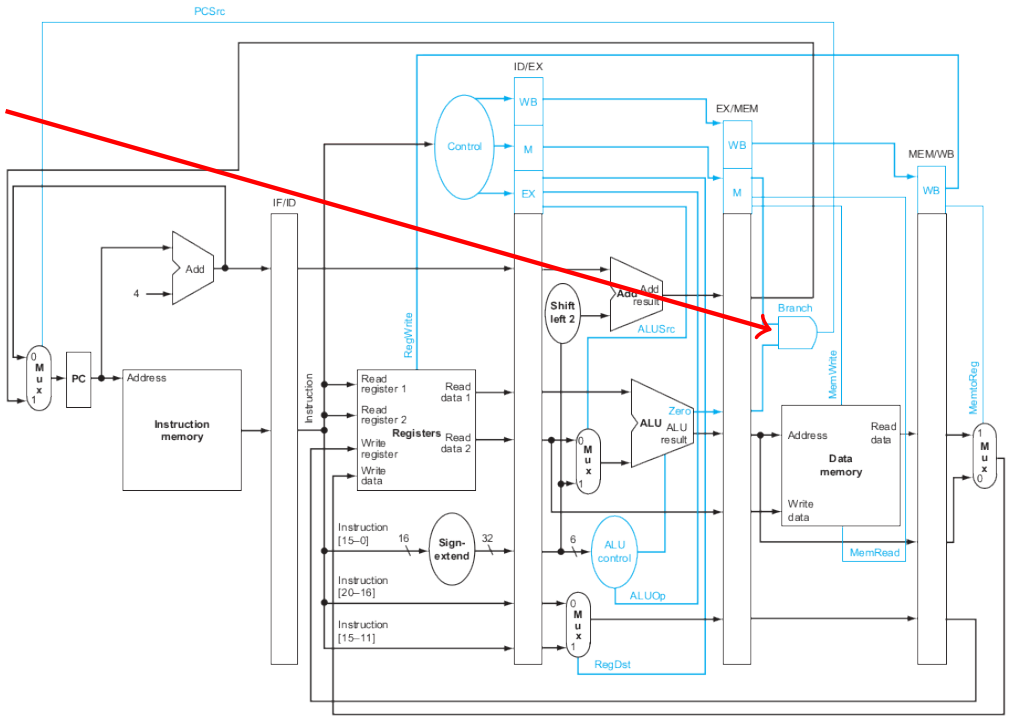


Fonte: [1]

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- O problema é definir se haverá ou não o *branch*
- Mover esse teste para `ID` implica mais hardware para detecção de conflitos e *forwarding*

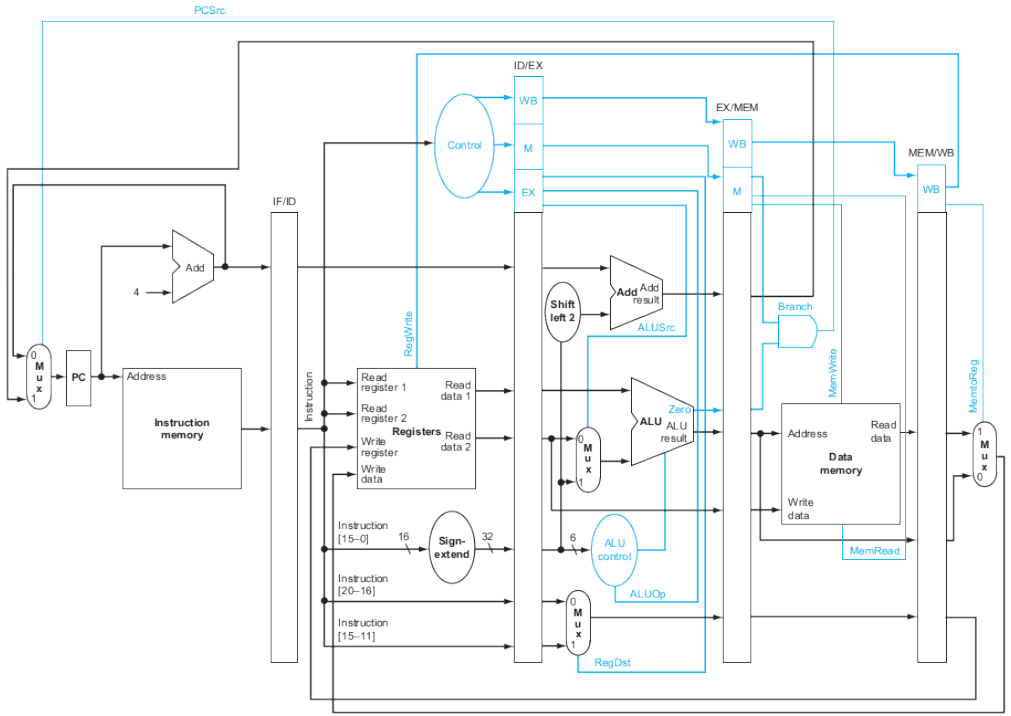


Fonte: [1]

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- Uma vez que o *branch* pode depender do resultado de uma instrução ainda na *pipeline*
  - Como no caso de conflitos de dados

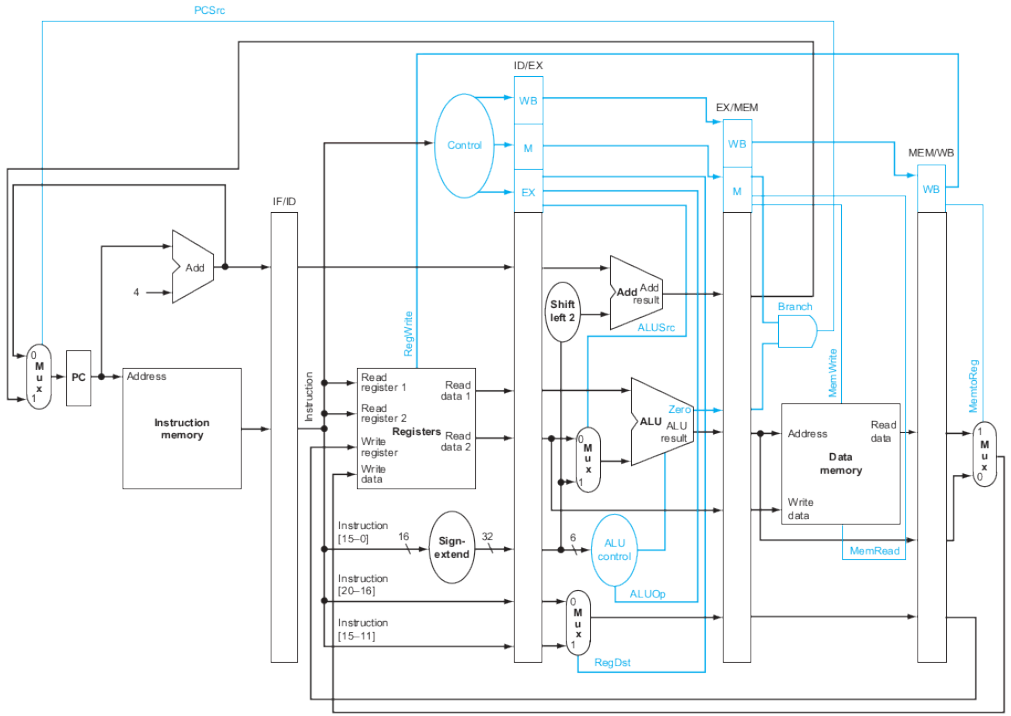


Fonte: [1]

Conflitos de Controle

Branches: Reduzindo o atraso de *branches*

- Uma vez que o *branch* pode depender do resultado de uma instrução ainda na *pipeline*
  - Como no caso de conflitos de dados

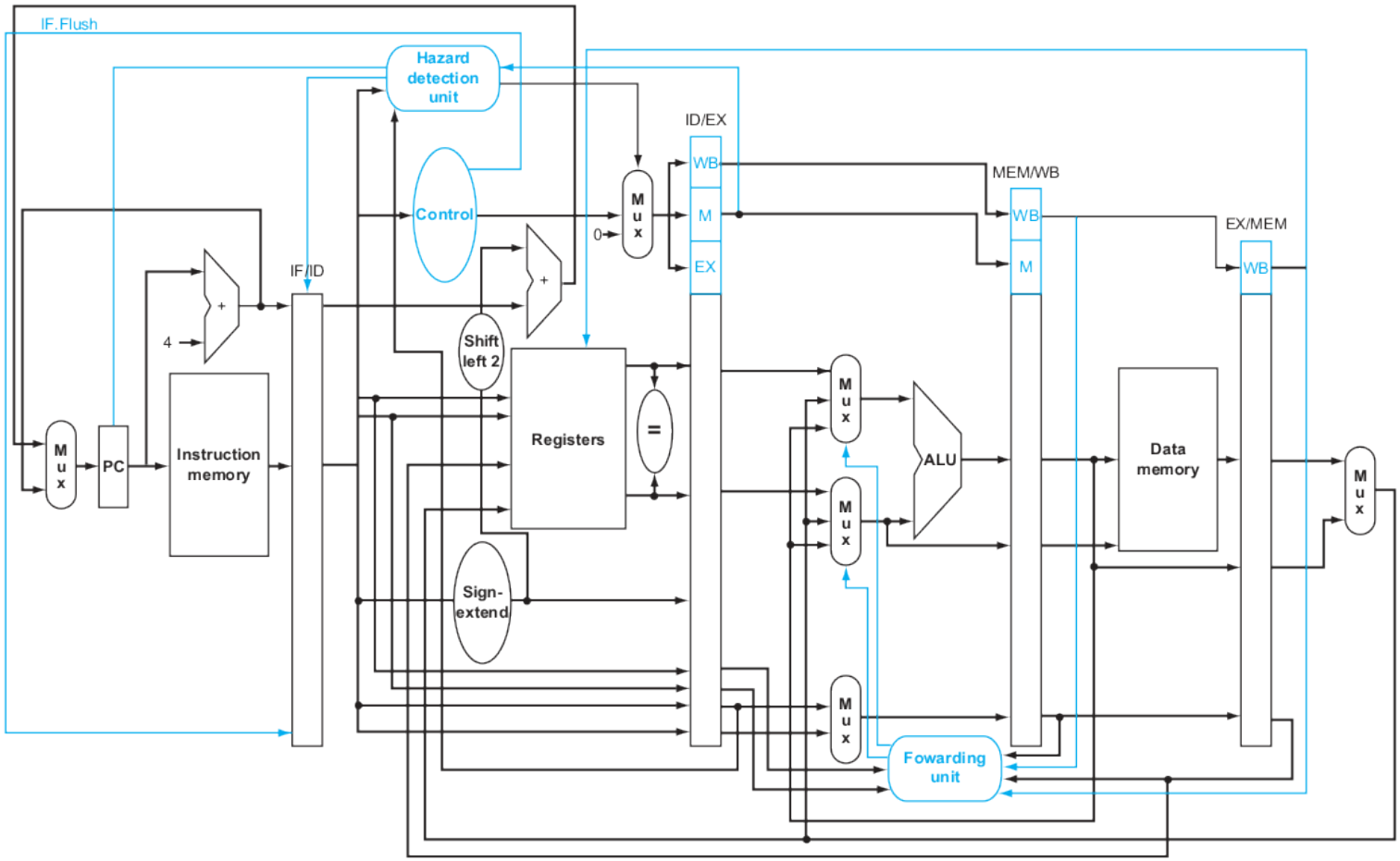


Fonte: [1]

- Ainda assim, vale
  - Pois reduz o custo de um *branch* para apenas 1 instrução, caso este ocorra

Conflitos de Controle

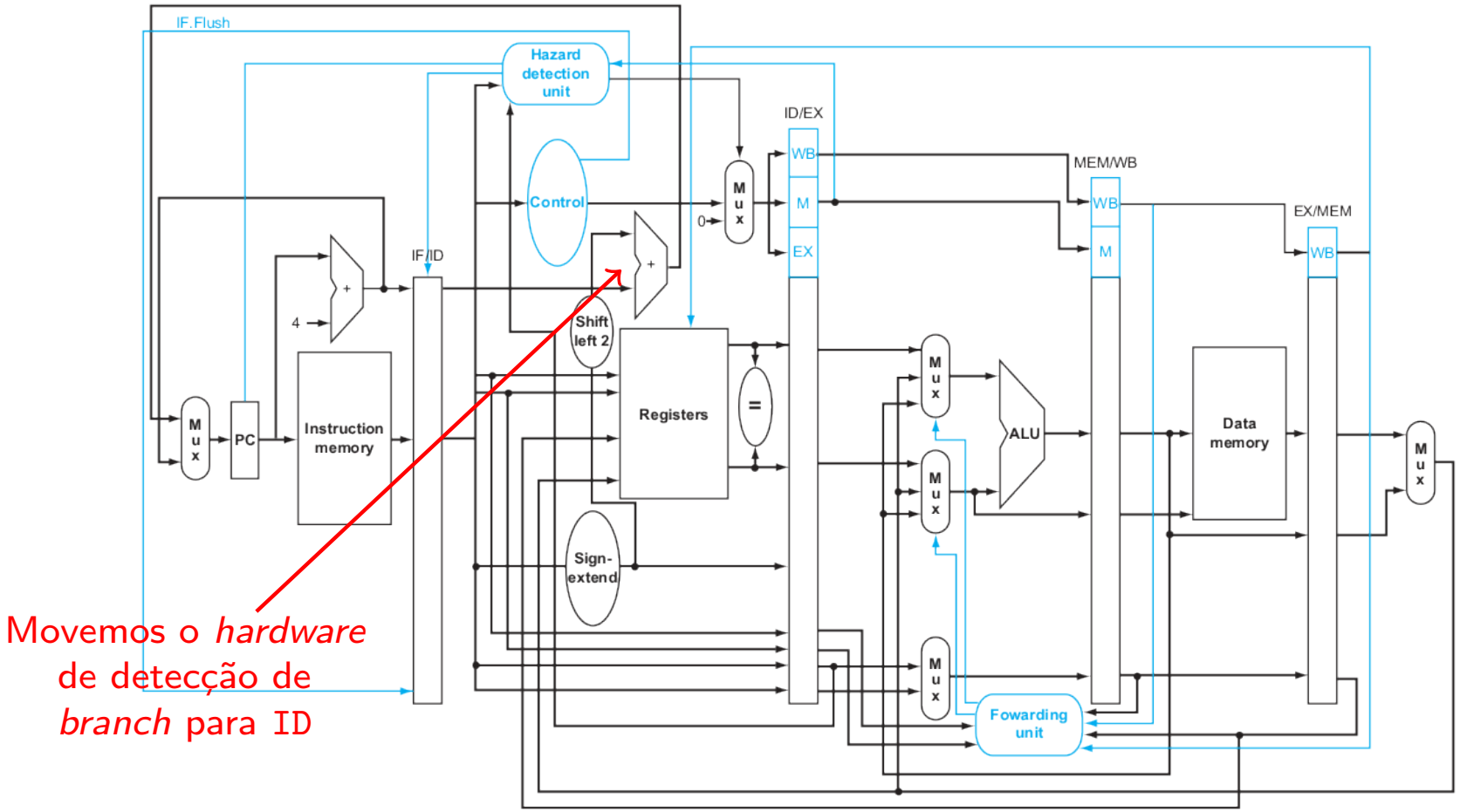
Branches: *Datapath*



Fonte: [1]

Conflitos de Controle

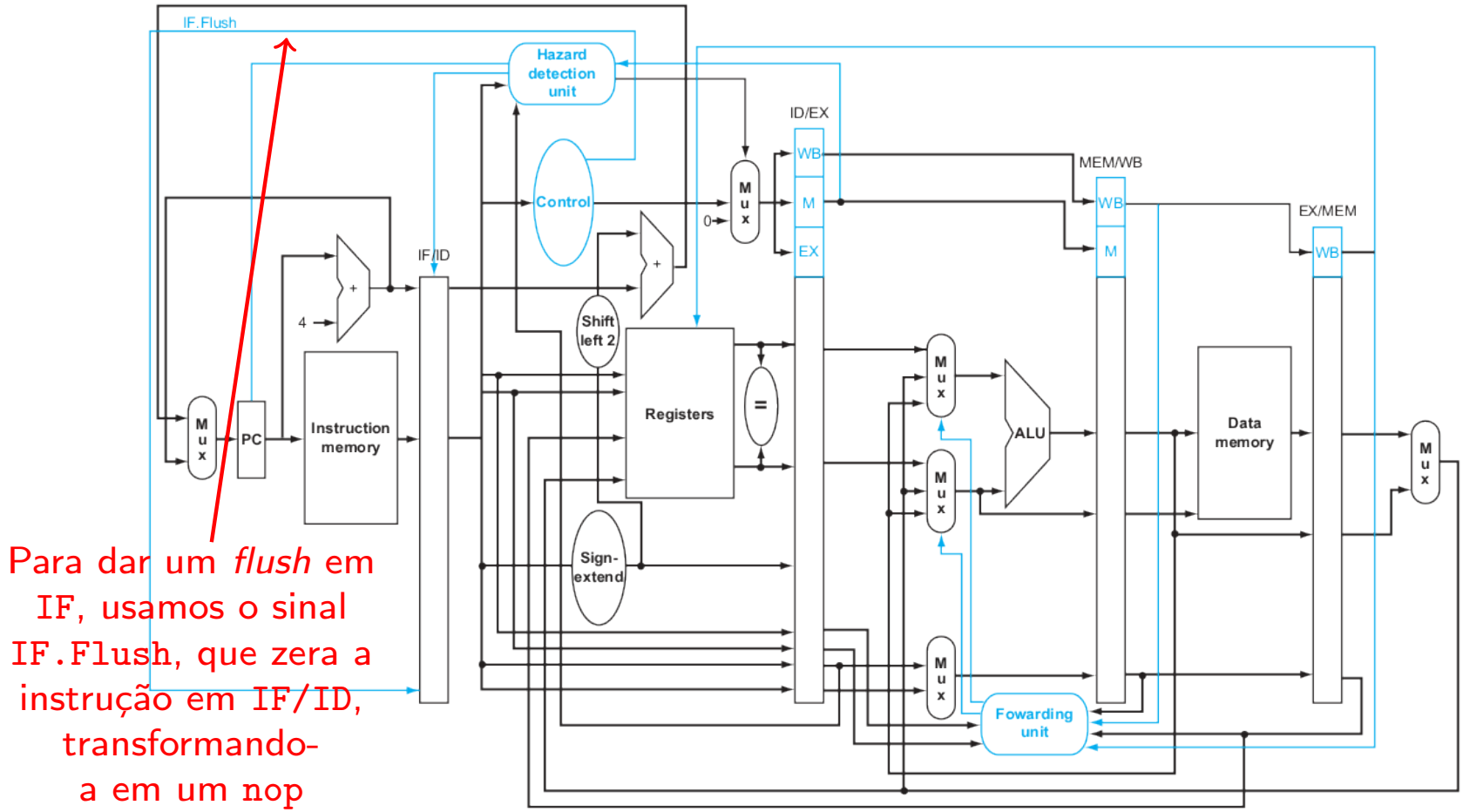
Branches: *Datapath*



Fonte: [1]

Conflitos de Controle

Branches: *Datapath*



Para dar um *flush* em `IF`, usamos o sinal `IF.Flush`, que zera a instrução em `IF/ID`, transformando-a em um `nop`

Fonte: [1]

Conflitos de Controle

Branches: Exemplo

- Considere a seguinte sequência de instruções:

```
36 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw $4, 50($7)
```

Conflitos de Controle

Branches: Exemplo

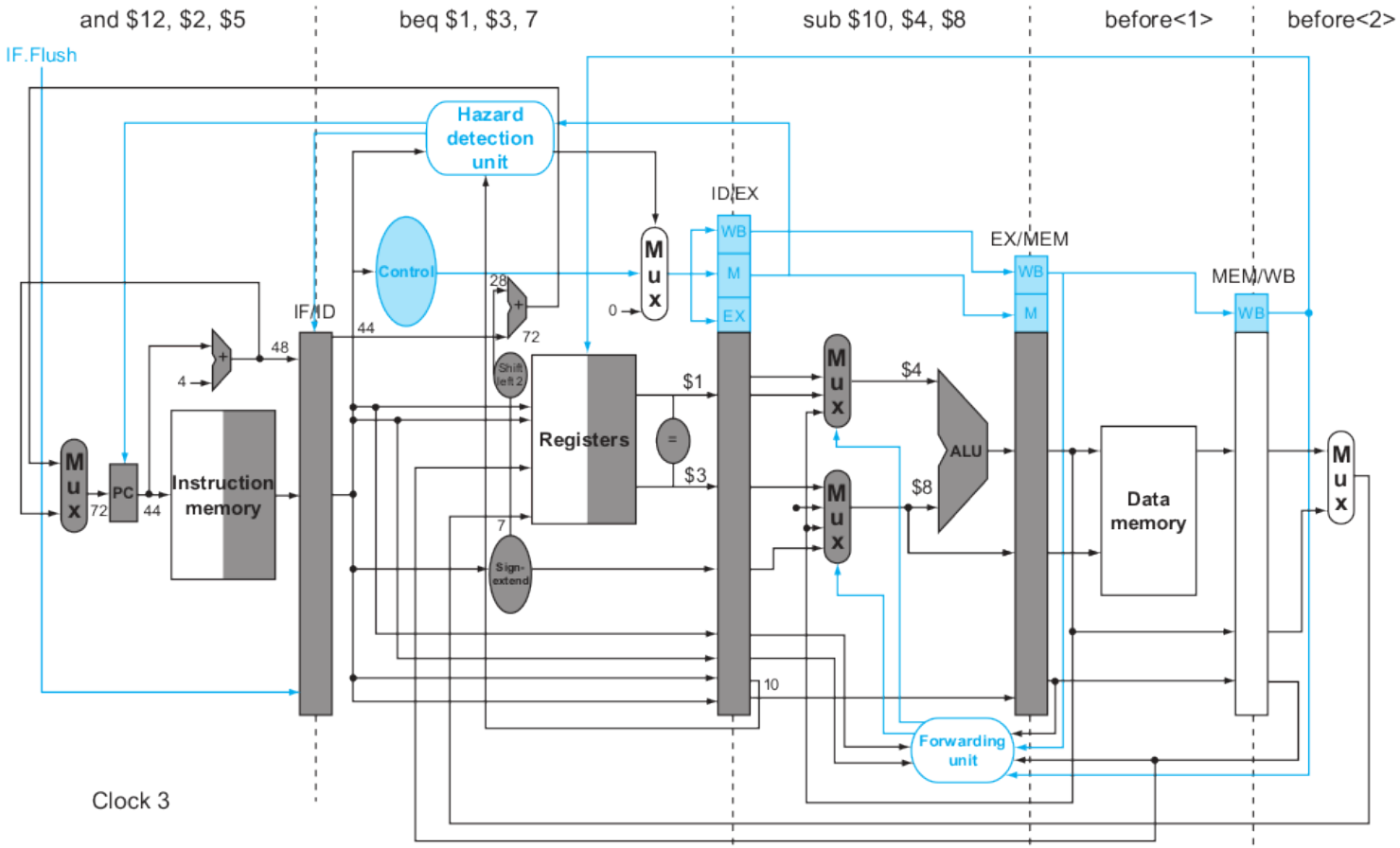
- Considere a seguinte sequência de instruções:

```
36 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw $4, 50($7)
```

O que acontece quando o *branch* é seguido?

Conflitos de Controle

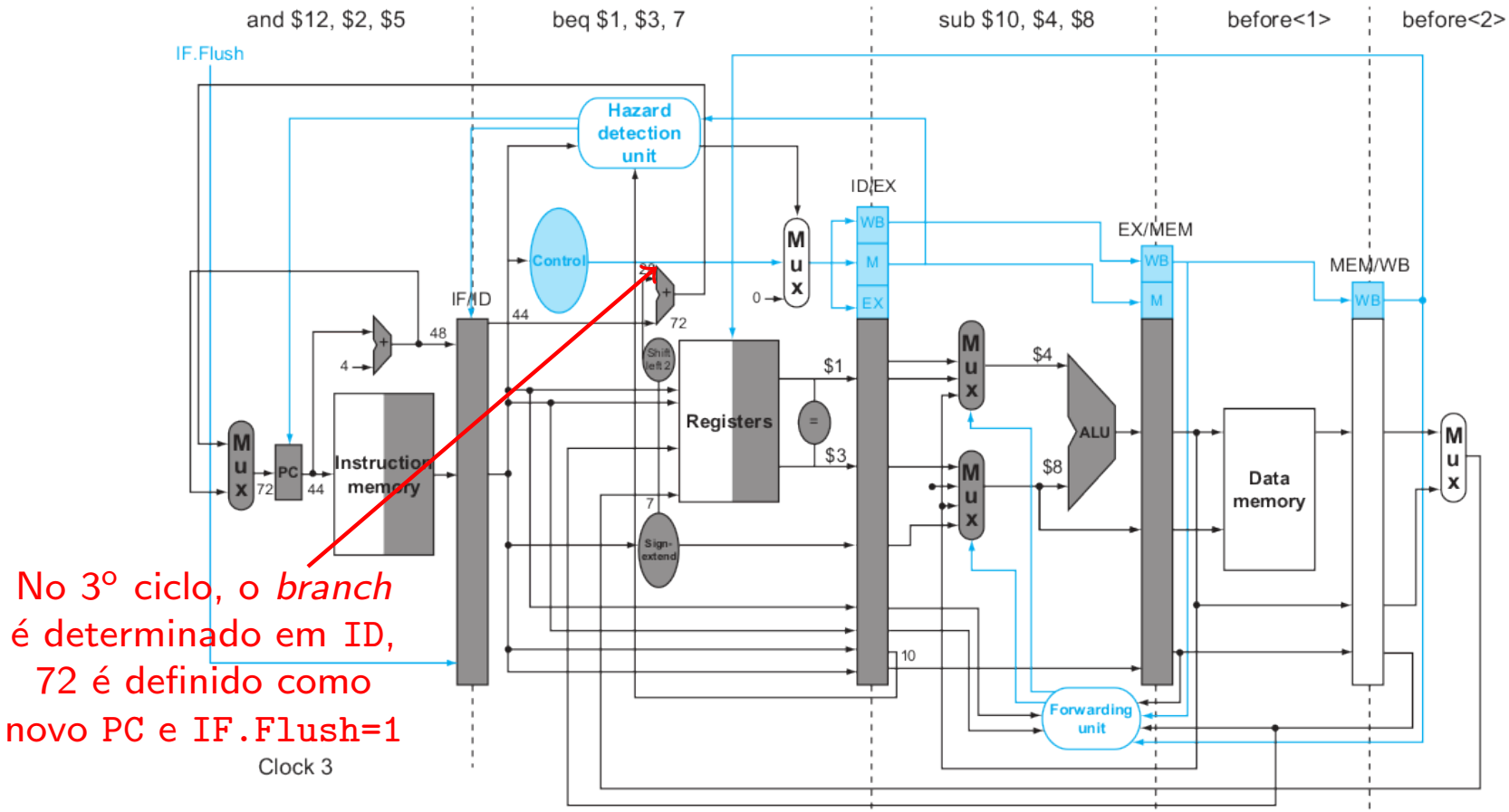
Branches: Exemplo



Fonte: [1]

Conflitos de Controle

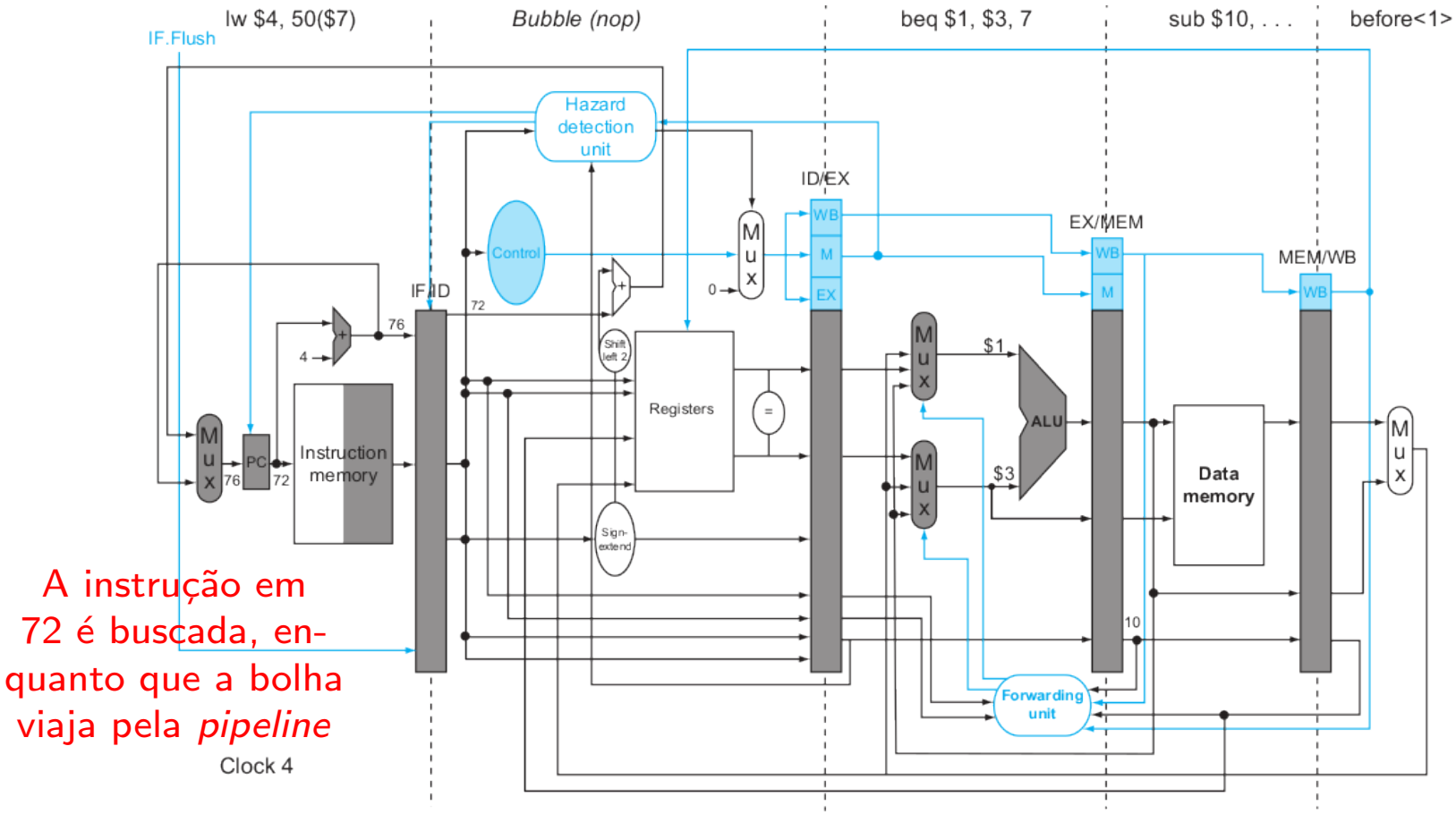
Branches: Exemplo



Fonte: [1]

Conflitos de Controle

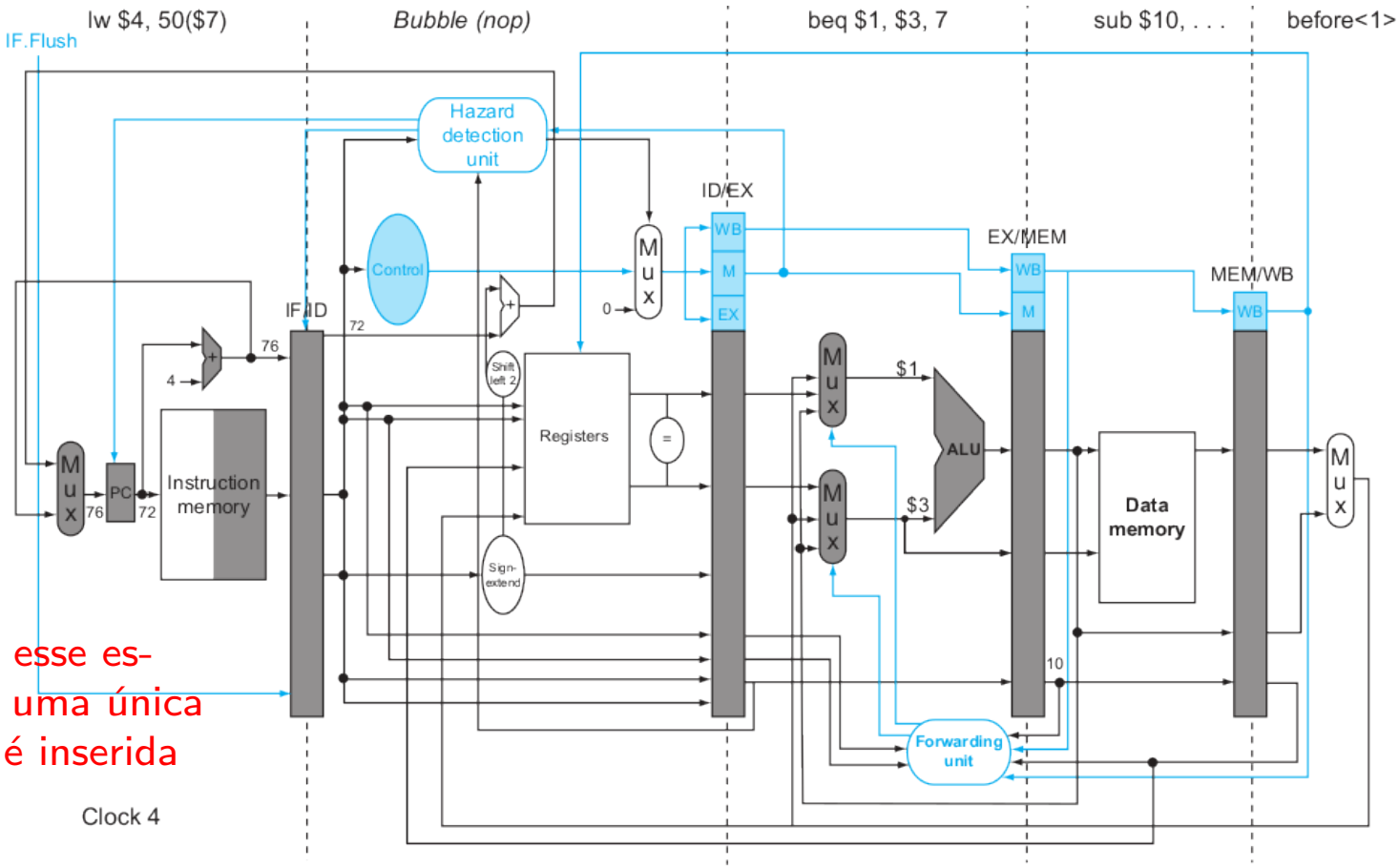
Branches: Exemplo



Fonte: [1]

Conflitos de Controle

Branches: Exemplo



Com esse esquema, uma única bolha é inserida

Fonte: [1]

Conflitos de Controle

Branches: Predizendo *branches* dinamicamente

- Com *hardware* adicional, podemos prever o comportamento de *branches* durante a execução
  - Podemos, por exemplo, verificar se da última vez que o endereço da instrução foi executado, ocorreu um *branch*
  - Se tiver ocorrido, podemos começar buscando instruções do mesmo lugar do último *branch*

Conflitos de Controle

Branches: Predizendo *branches* dinamicamente

- Com *hardware* adicional, podemos prever o comportamento de *branches* durante a execução
  - Podemos, por exemplo, verificar se da última vez que o endereço da instrução foi executado, ocorreu um *branch*
  - Se tiver ocorrido, podemos começar buscando instruções do mesmo lugar do último *branch*
- Usamos para isso um *buffer* de predição de *branch*
  - Uma memória, em `ID`, indexada por parte do endereço da instrução, contendo um bit dizendo se houve ou não *branch*

Conflitos de Controle

Branches: Predizendo *branches* dinamicamente

- Isso, naturalmente, é um "chute"
  - Nada garante que o comportamento ocorrerá
  - Se não ocorrer, basta dar um *flush* na instrução incorretamente carregada, e inverter o bit de predição no *buffer*

Conflitos de Controle

Branches: Predizendo *branches* dinamicamente

- Isso, naturalmente, é um "chute"
  - Nada garante que o comportamento ocorrerá
  - Se não ocorrer, basta dar um *flush* na instrução incorretamente carregada, e inverter o bit de predição no *buffer*
- Frequentemente são usados 2 bits em vez de 1
  - Exigindo 2 erros antes da mudança dos bits
  - E acomodando assim as variações de comportamento na entrada e saída de laços

Exceções

Interrupções e Exceções

Definição

- Exceção
  - Um evento não programado que interrompe a execução do programa (Ex: instrução inválida e *overflow* em operações aritméticas)
- Interrupção
  - Uma exceção vinda de fora do processador (Ex: E/S)

Interrupções e Exceções

Definição

- Exceção
  - Um evento não programado que interrompe a execução do programa (Ex: instrução inválida e *overflow* em operações aritméticas)
- Interrupção
  - Uma exceção vinda de fora do processador (Ex: E/S)
- Algumas arquiteturas adotam o nome "interrupção" para ambas

Interrupções e Exceções

Exceções em MIPS

- Suponha um *overflow* em `add $1, $2, $1`. O que fazer?

Interrupções e Exceções

Exceções em MIPS

- Suponha um *overflow* em `add $1, $2, $1`. O que fazer?
  - O processador salva o endereço da instrução problemática no `EPC` (*exception program counter*)
    - De fato, salva seu `PC+4`

Interrupções e Exceções

Exceções em MIPS

- Suponha um *overflow* em `add $1, $2, $1`. O que fazer?
  - O processador salva o endereço da instrução problemática no `EPC` (*exception program counter*)
    - De fato, salva seu `PC+4`
  - Ele então transfere o controle ao sistema operacional, em algum endereço específico

Interrupções e Exceções

Exceções em MIPS

- Suponha um *overflow* em `add $1, $2, $1`. O que fazer?
  - O processador salva o endereço da instrução problemática no `EPC` (*exception program counter*)
    - De fato, salva seu `PC+4`
  - Ele então transfere o controle ao sistema operacional, em algum endereço específico
  - O S.O. então executa a ação apropriada

Interrupções e Exceções

Exceções em MIPS

- Suponha um *overflow* em `add $1, $2, $1`. O que fazer?
  - O processador salva o endereço da instrução problemática no `EPC` (*exception program counter*)
    - De fato, salva seu `PC+4`
  - Ele então transfere o controle ao sistema operacional, em algum endereço específico
  - O S.O. então executa a ação apropriada
  - Se resolver não matar o programa, o S.O. pode usar o `EPC` para continuar sua execução

Exceções em MIPS

- Mas saber que instrução causou a exceção não é suficiente
  - O S.O. também precisa saber sua razão

Interrupções e Exceções

Exceções em MIPS

- Mas saber que instrução causou a exceção não é suficiente
  - O S.O. também precisa saber sua razão
- Algumas arquiteturas usam um arranjo de interrupções
  - Um arranjo, indexado pela causa, contendo o endereço em RAM de onde deve estar a rotina de tratamento:

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

- MIPS usa um único ponto de entrada para todas as exceções
  - Endereço de memória fixo (`0x8000 0180`), onde deve ser colocado (pelo S.O.) o código geral para o tratamento das exceções

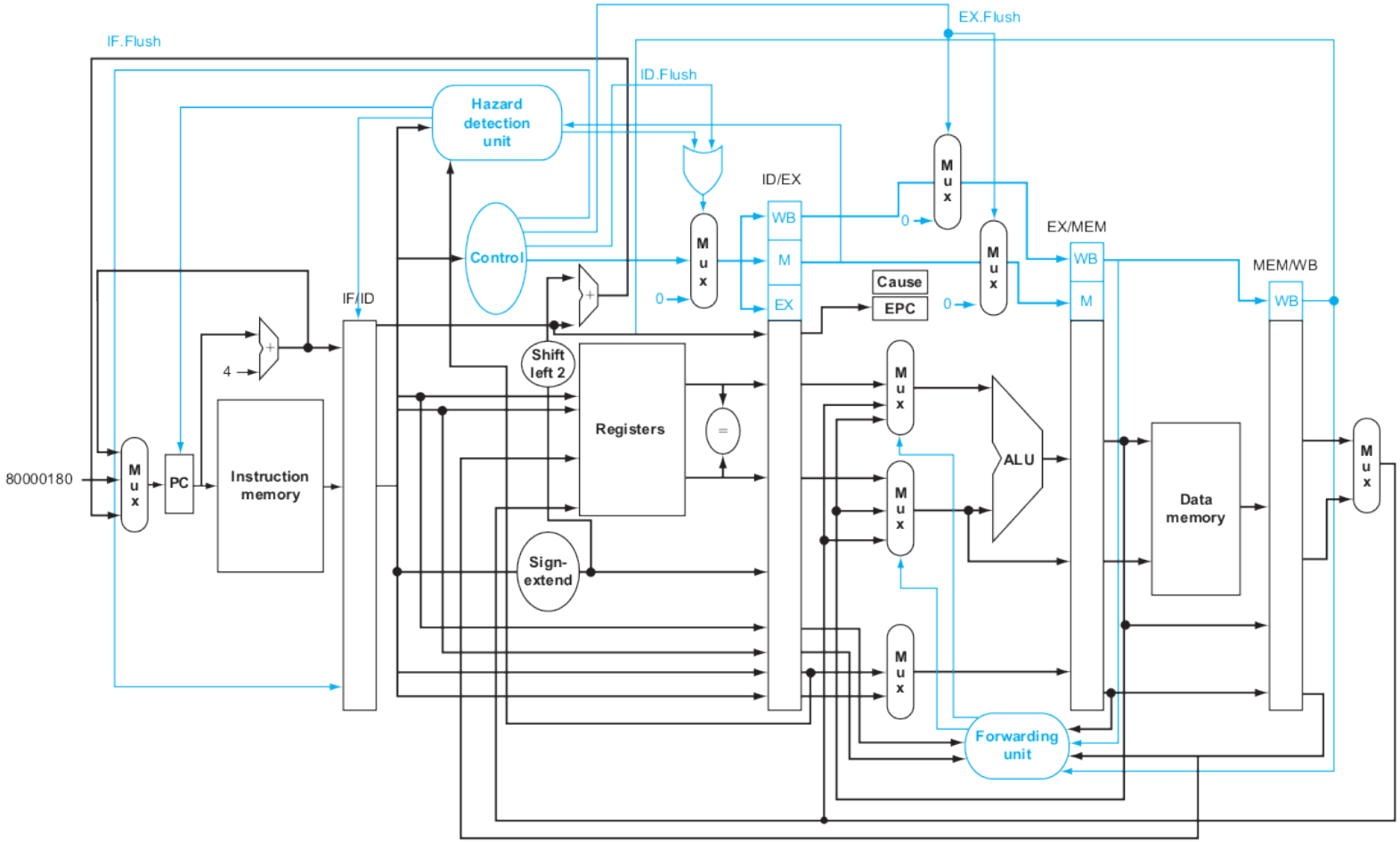
Interrupções e Exceções

Exceções em MIPS

- MIPS usa um único ponto de entrada para todas as exceções
  - Endereço de memória fixo (`0x8000 0180`), onde deve ser colocado (pelo S.O.) o código geral para o tratamento das exceções
  - A causa da exceção é armazenada em um registrador de status chamado **Registrador de Causa** (*Cause Register*)
    - Usado pelo S.O. para ativar a rotina de tratamento apropriada

Interrupções e Exceções

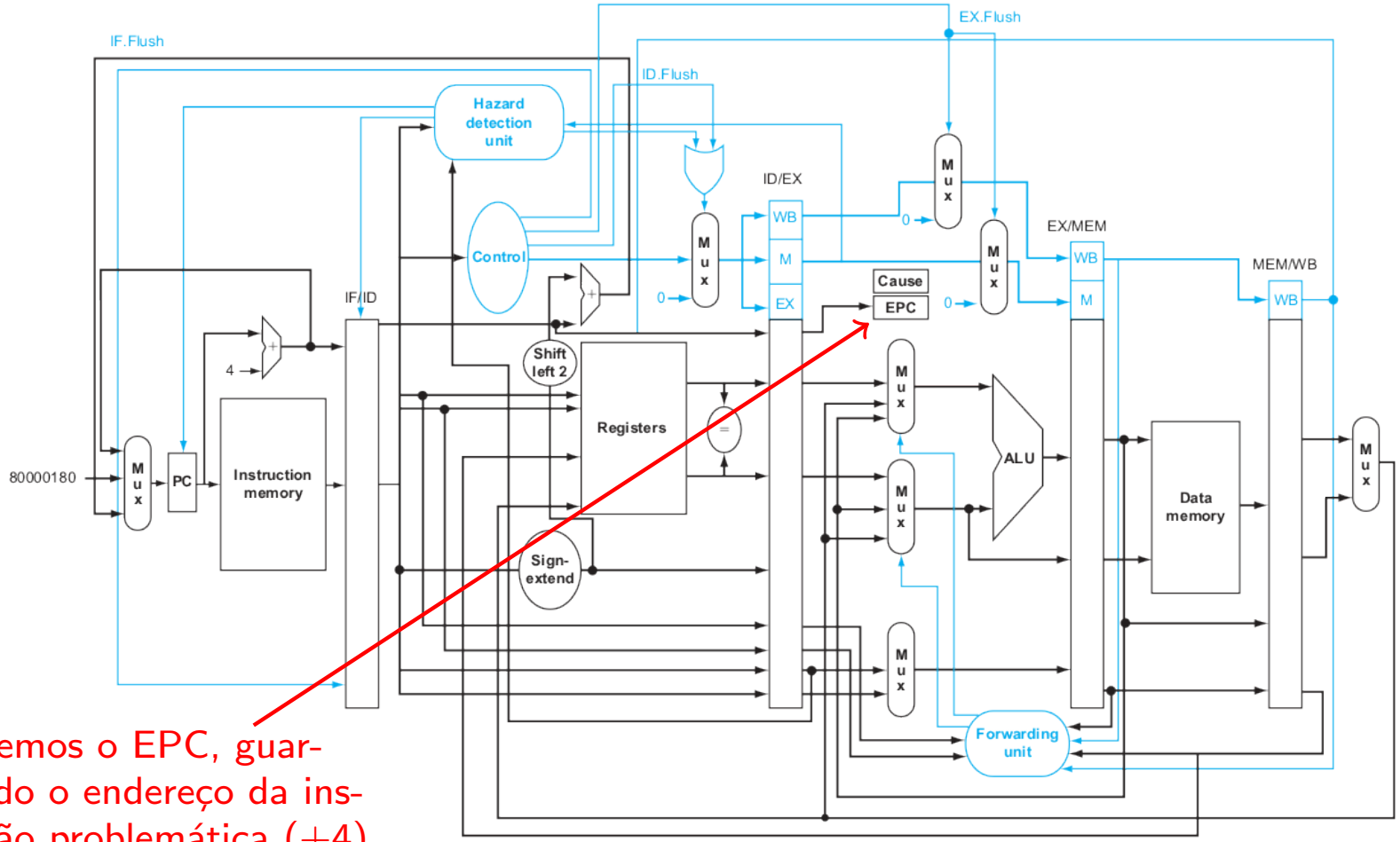
Exceções em MIPS



Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

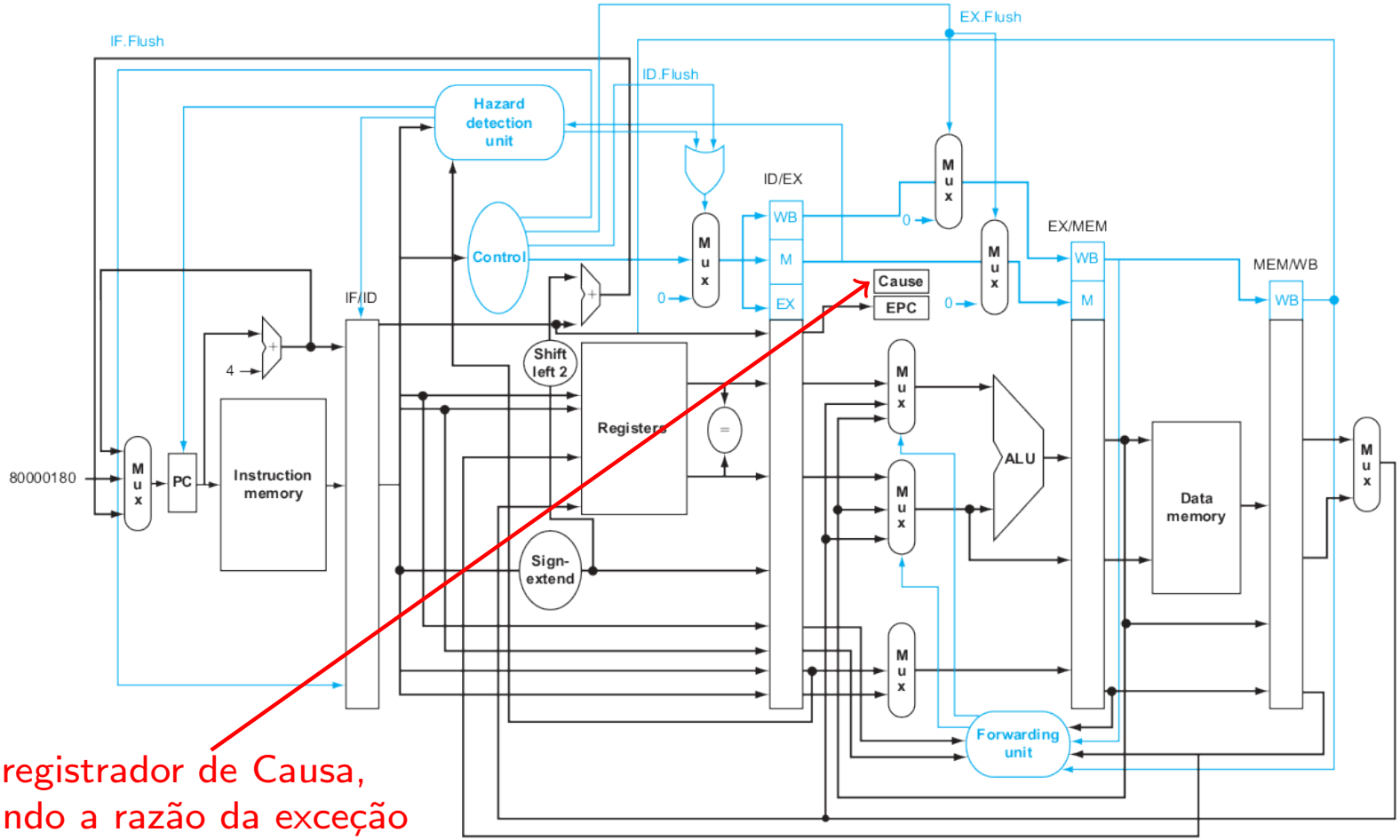


Temos o EPC, guardando o endereço da instrução problemática (+4)

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

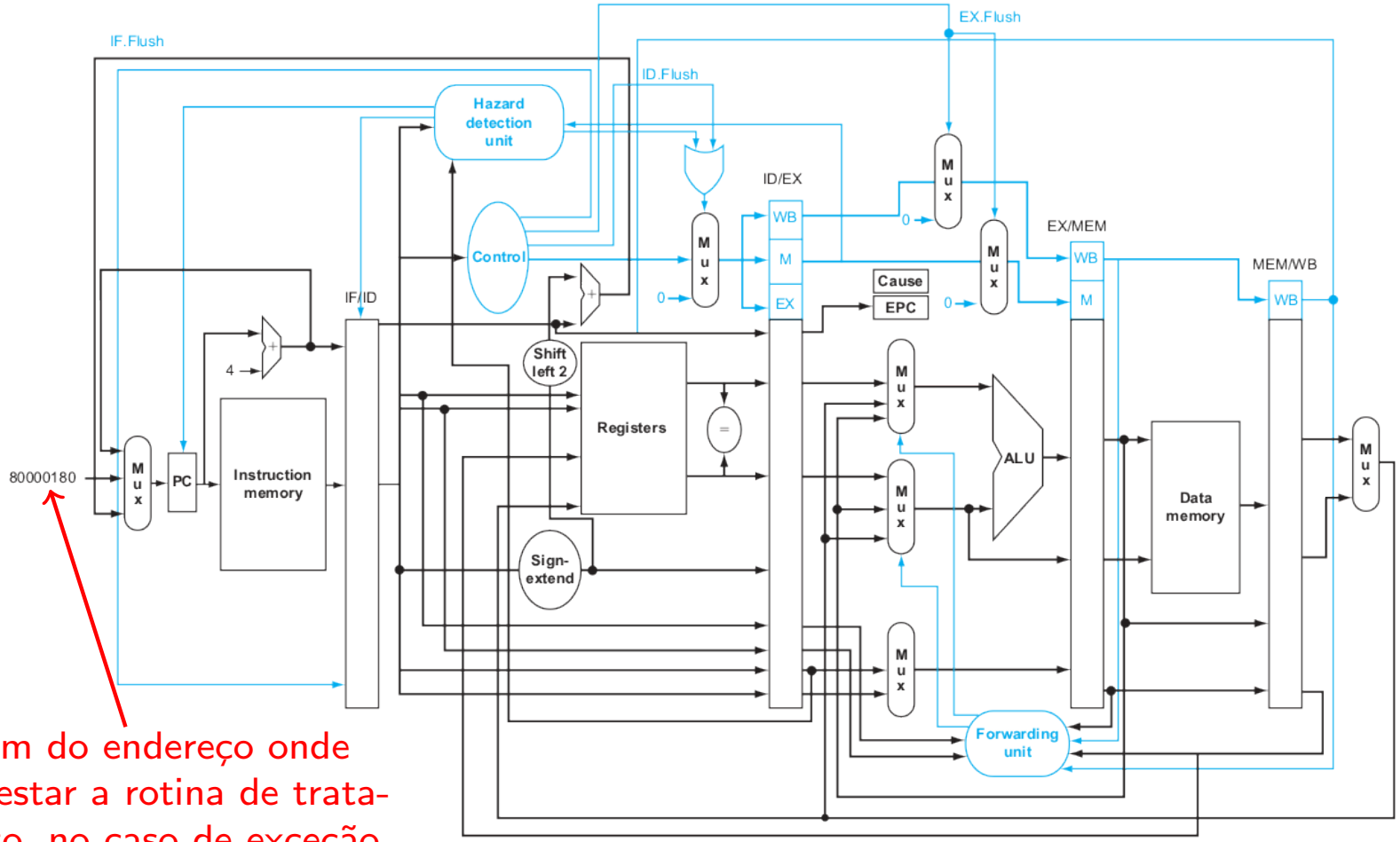


E o registrador de Causa, guardando a razão da exceção

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

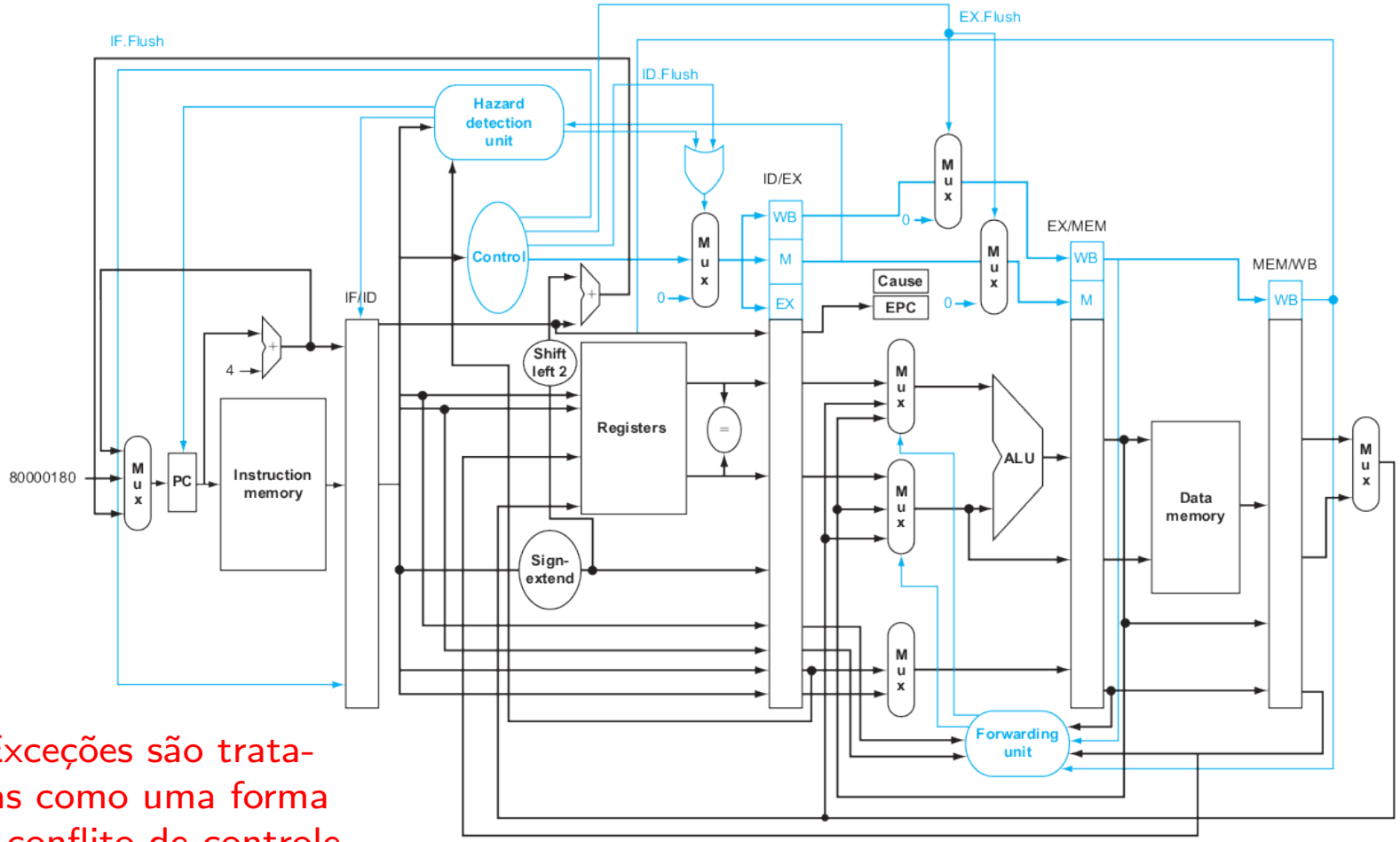


Além do endereço onde deve estar a rotina de tratamento, no caso de exceção

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

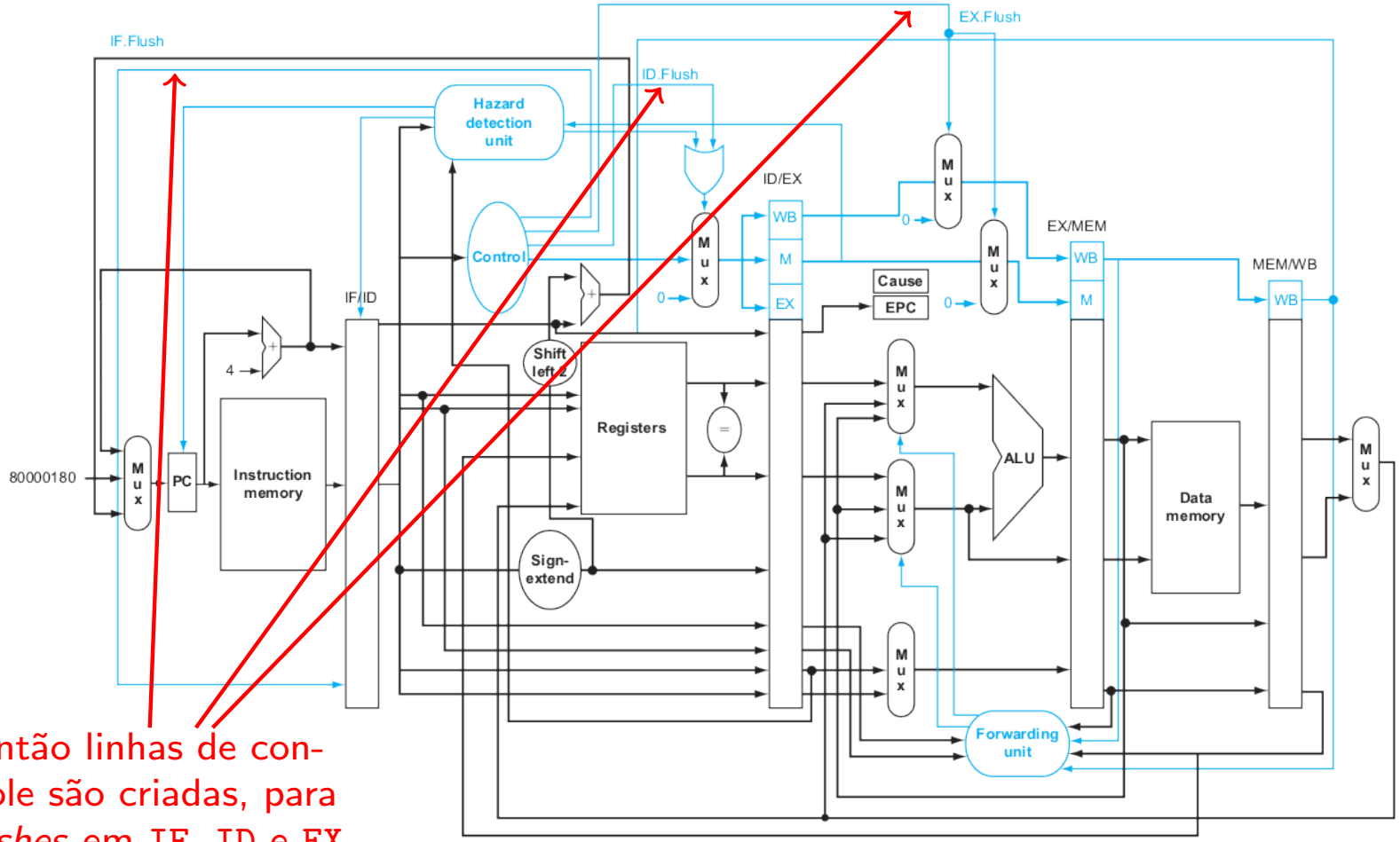


Exceções são tratadas como uma forma de conflito de controle

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

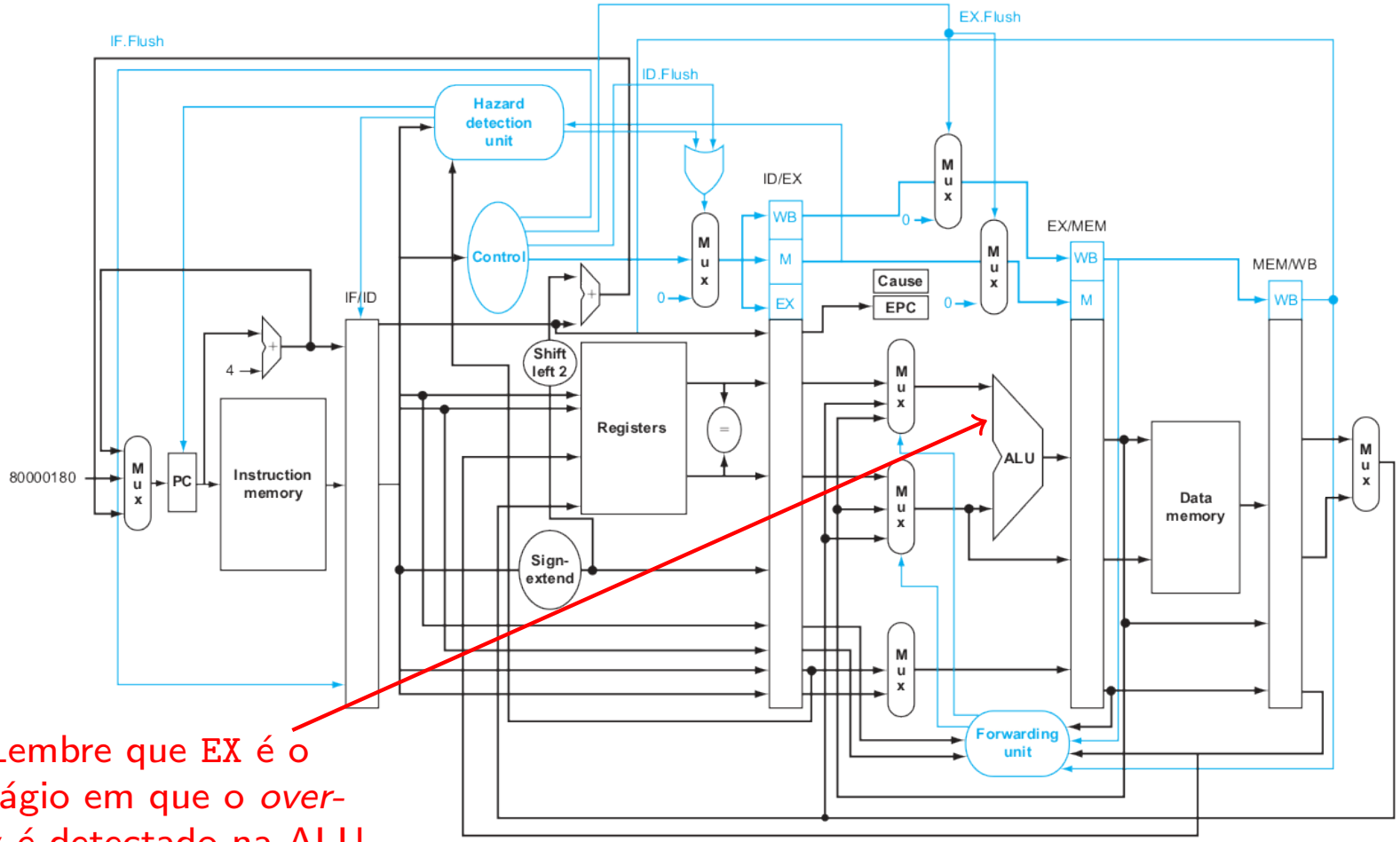


Então linhas de controle são criadas, para *flushes* em `IF`, `ID` e `EX`

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

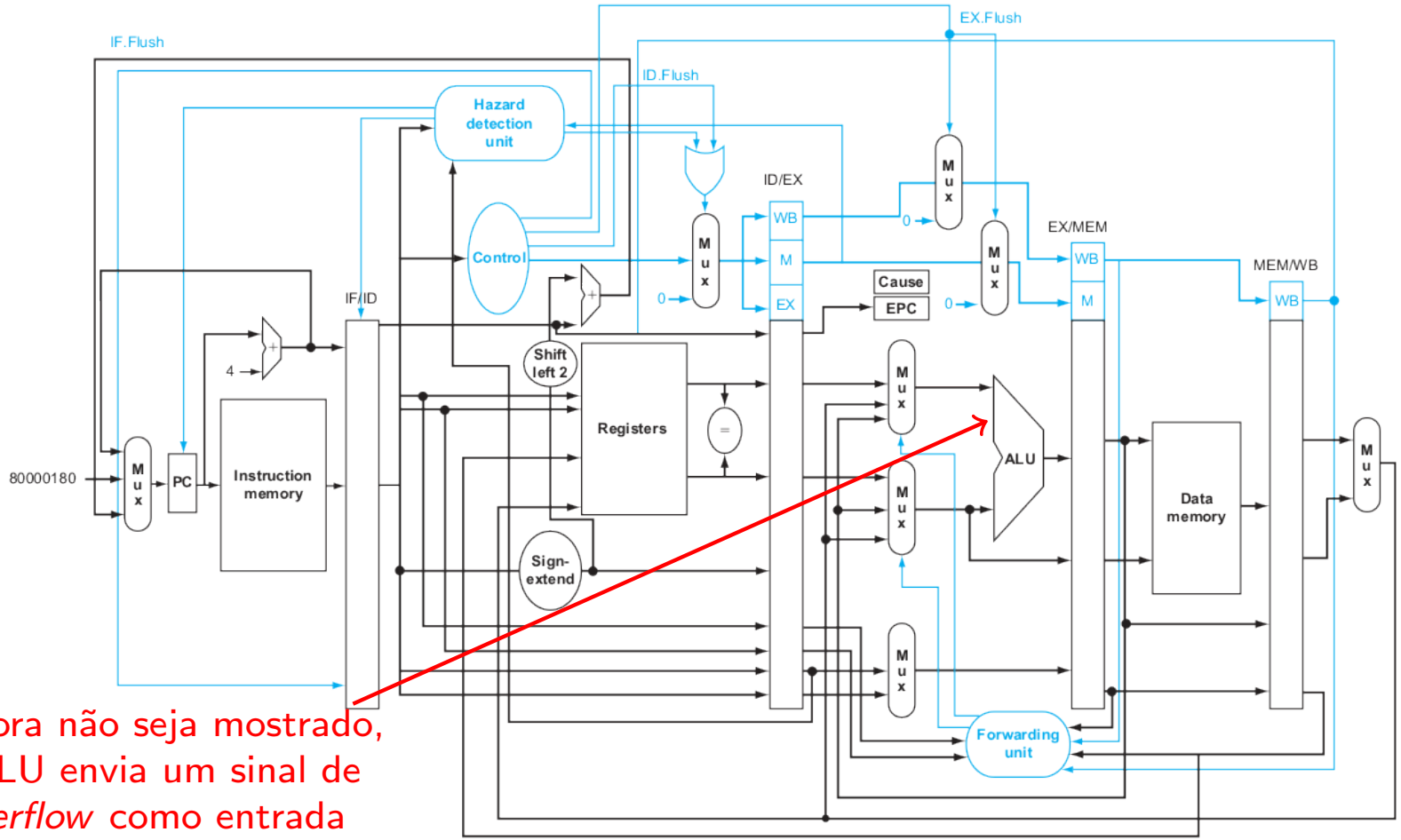


Lembre que `EX` é o estágio em que o *over-flow* é detectado na ALU

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

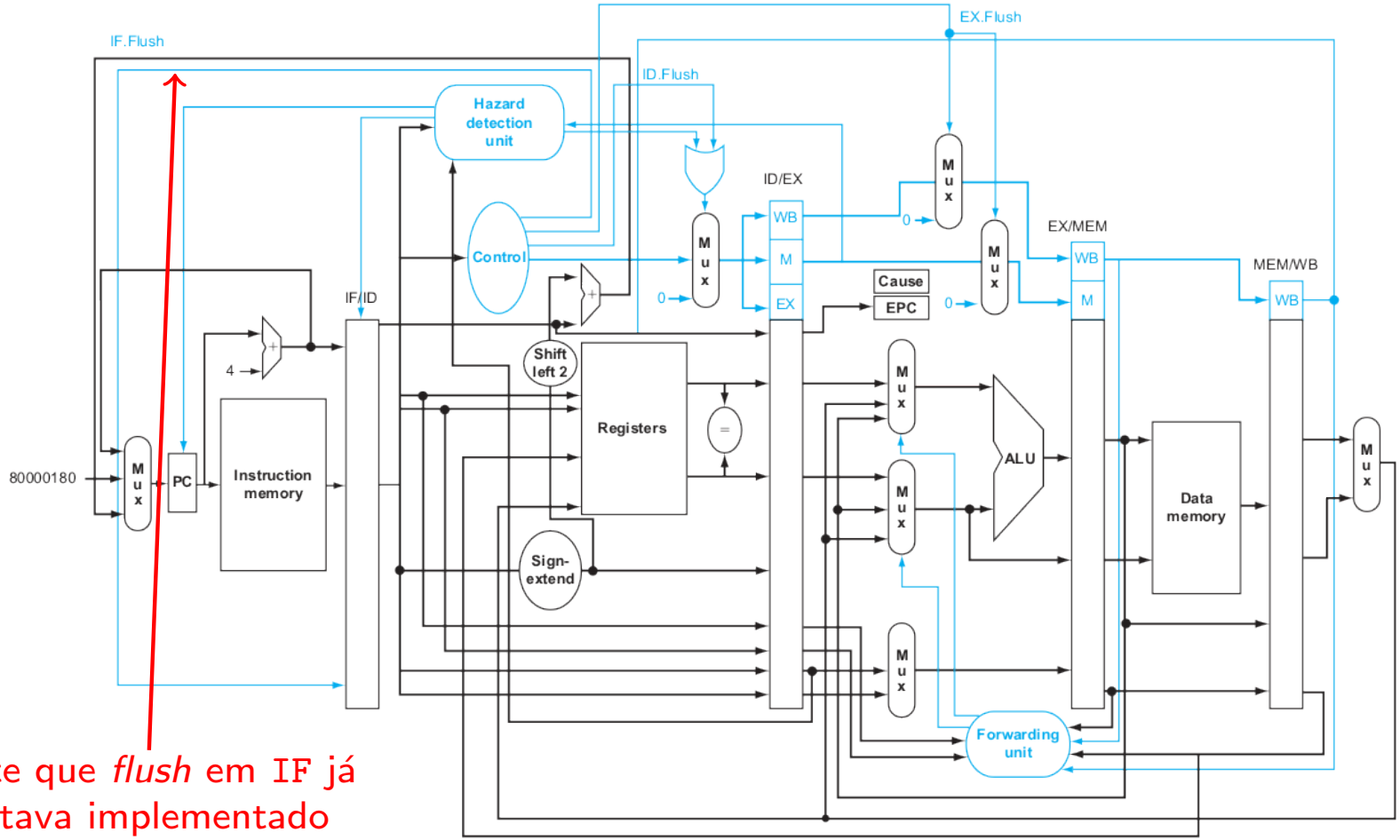


Embora não seja mostrado, a ALU envia um sinal de *overflow* como entrada para a unidade de controle

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

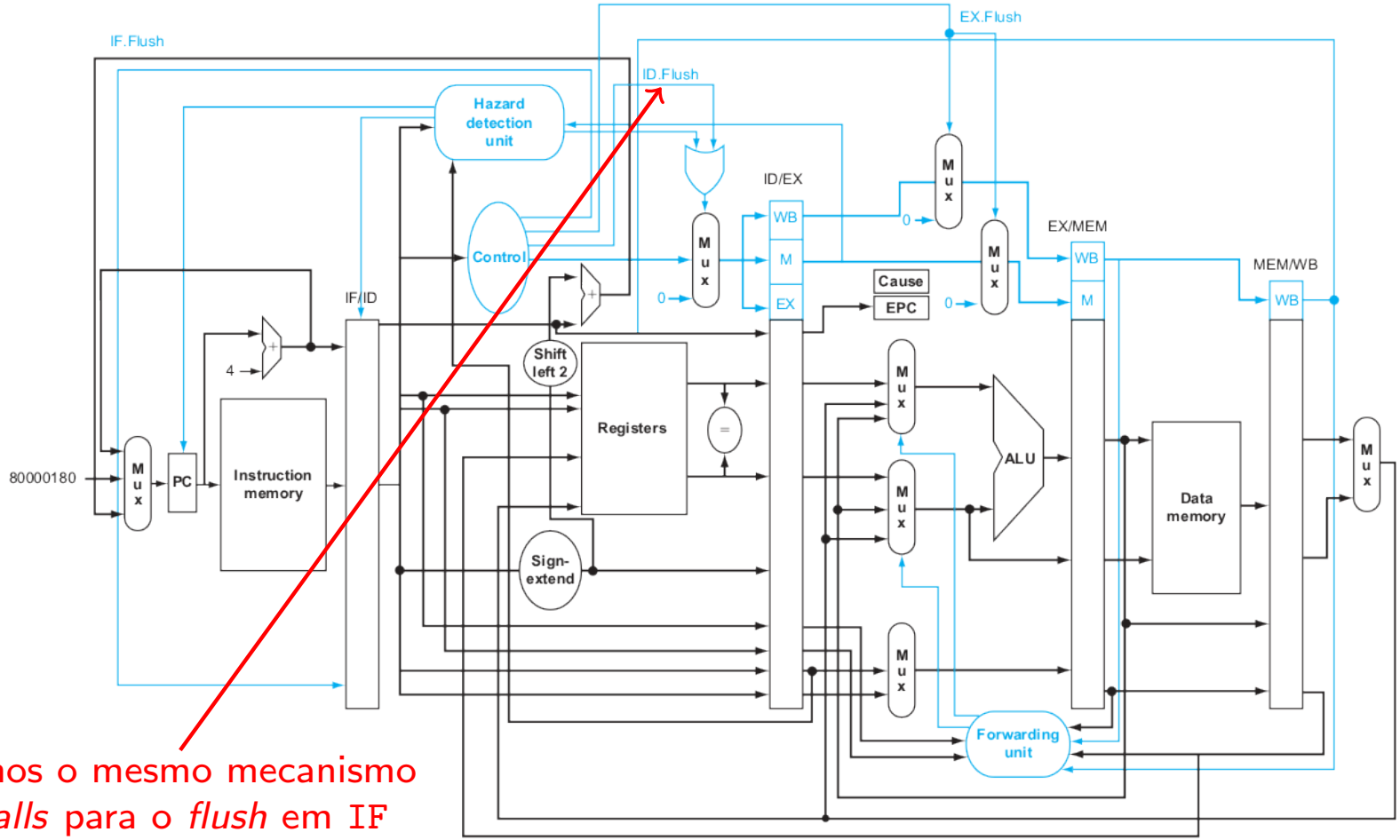


Note que *flush* em `IF` já estava implementado

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

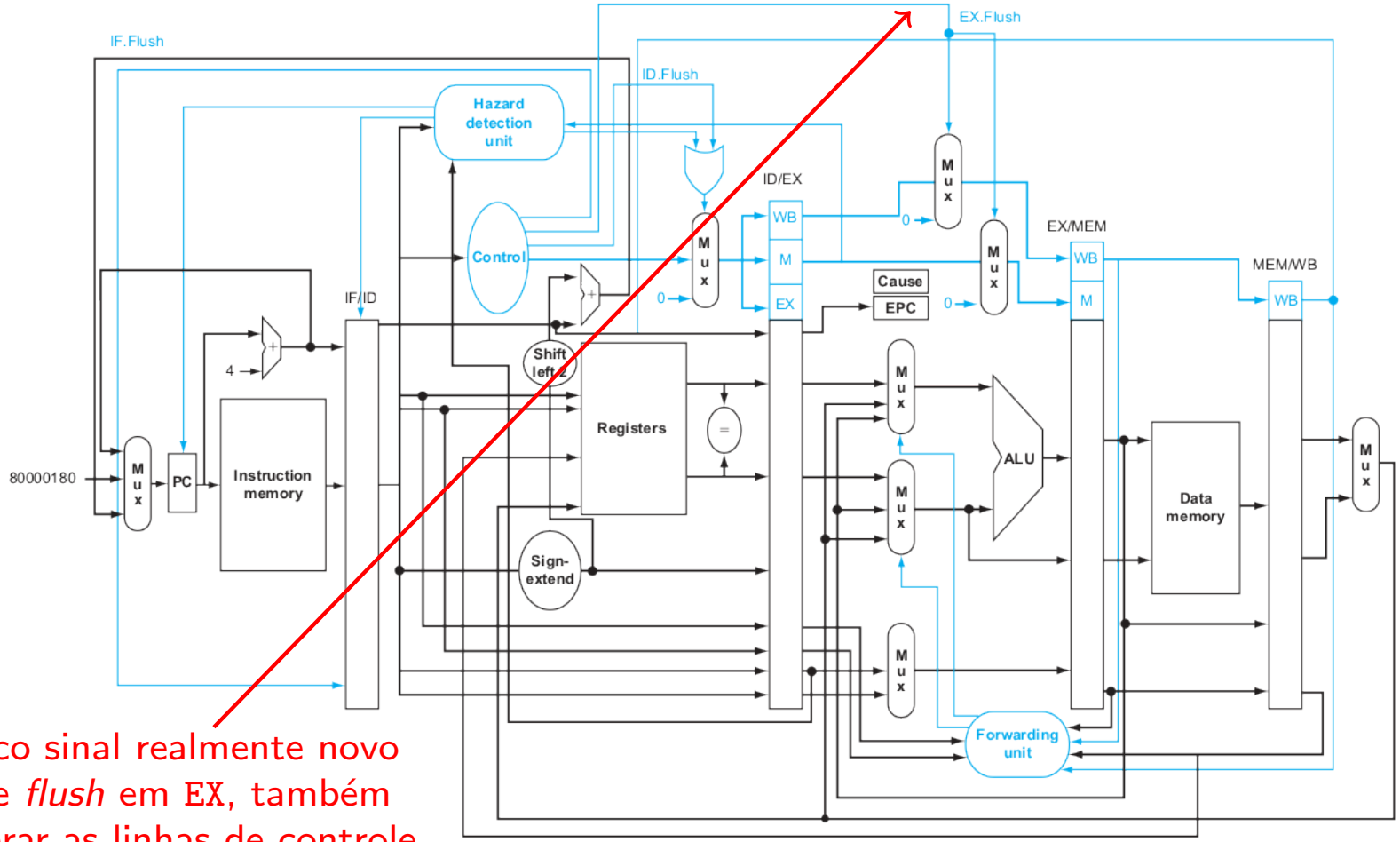


E usamos o mesmo mecanismo de *stalls* para o *flush* em `IF`

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

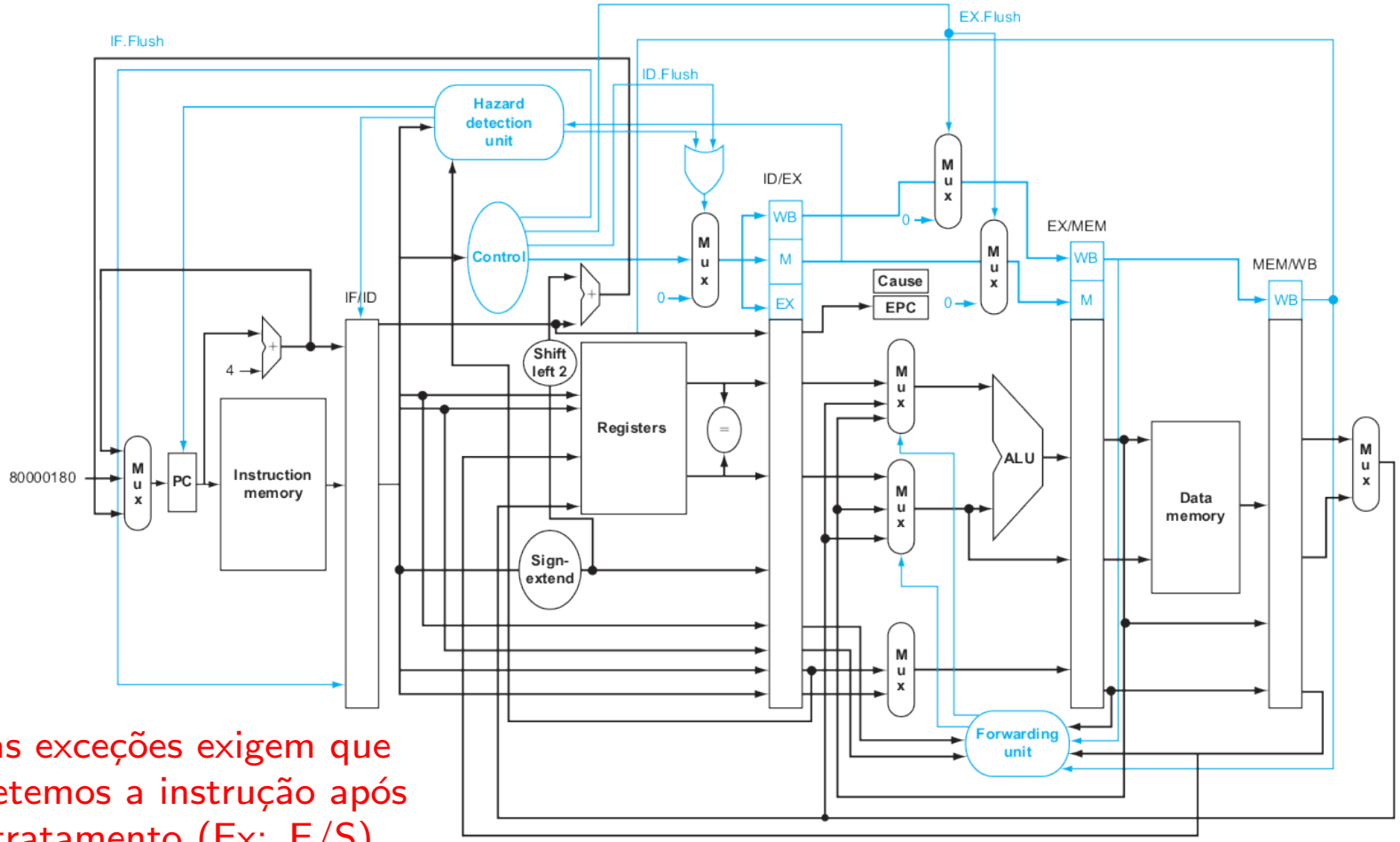


O único sinal realmente novo é o de *flush* em `EX`, também para zerar as linhas de controle

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS

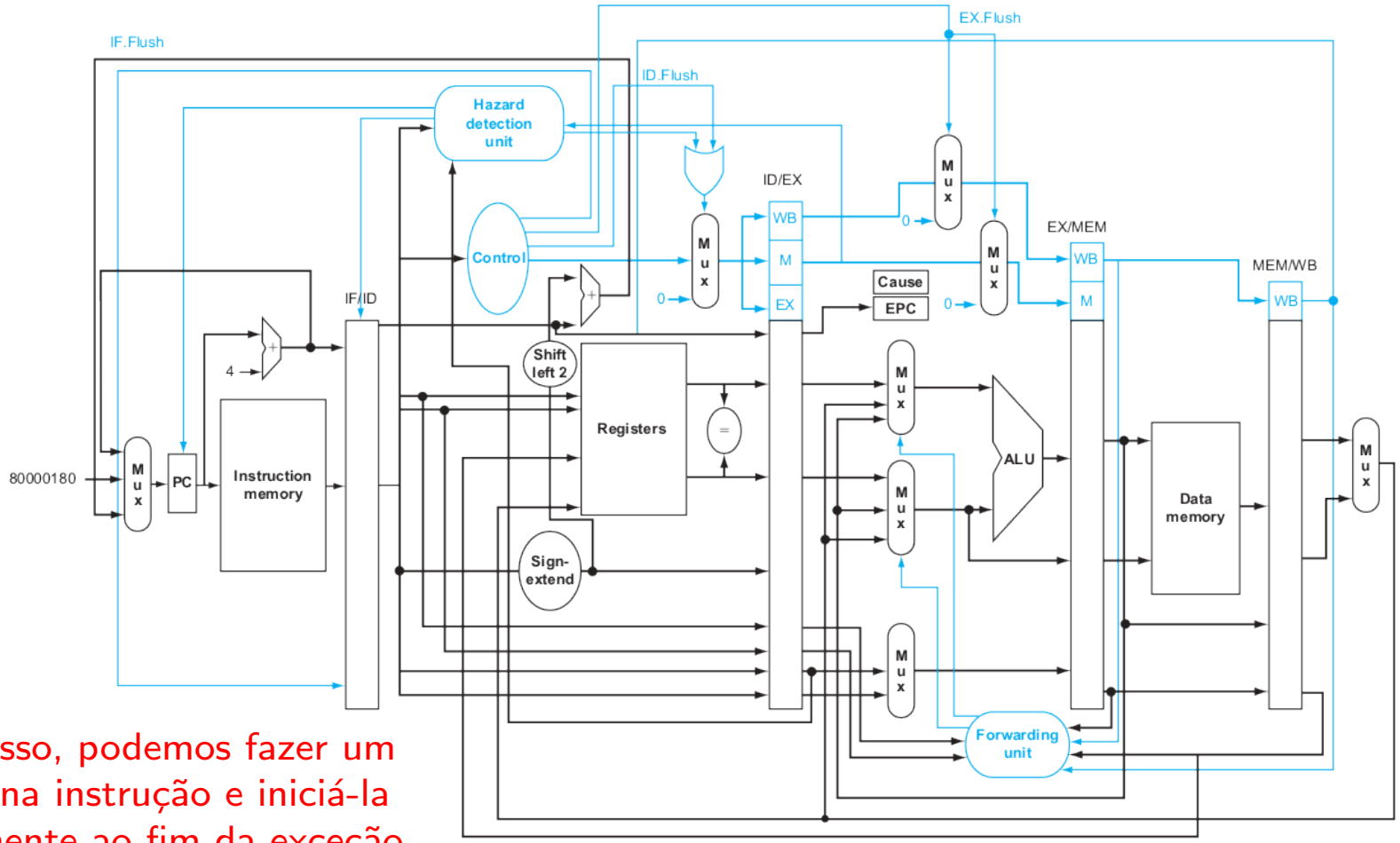


Muitas exceções exigem que completemos a instrução após seu tratamento (Ex: E/S)

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS



Para isso, podemos fazer um *flush* na instrução e iniciá-la novamente ao fim da exceção

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS: Exemplo

- Considere a sequência de instruções:

Endereço	*Instrução*
0x40	sub $11, $2, $4
0x44	and $12, $2, $5
0x48	or $13, $2, $6
0x4C	add $1, $2, $1
0x50	slt $15, $6, $7
0x54	lw $16, 50($7)
...	

Interrupções e Exceções

Exceções em MIPS: Exemplo

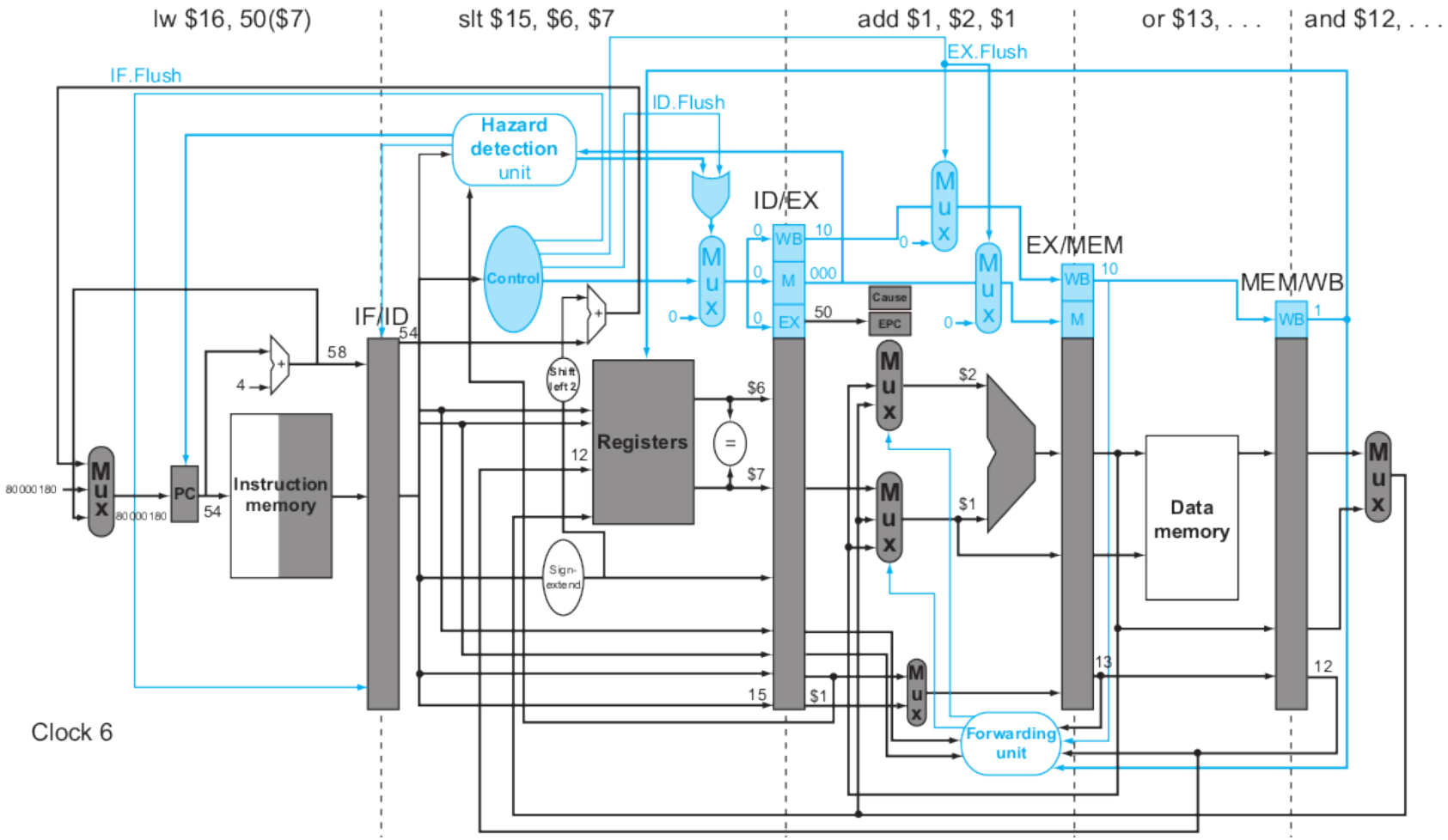
- Considere a sequência de instruções:

Endereço	*Instrução*			
0x40	sub	$11,	$2,	$4
0x44	and	$12,	$2,	$5
0x48	or	$13,	$2,	$6
0x4C	add	$1,	$2,	$1
0x50	slt	$15,	$6,	$7
0x54	lw	$16,	50($7)	
...				

O que acontece em caso de *overflow* em add?

Interrupções e Exceções

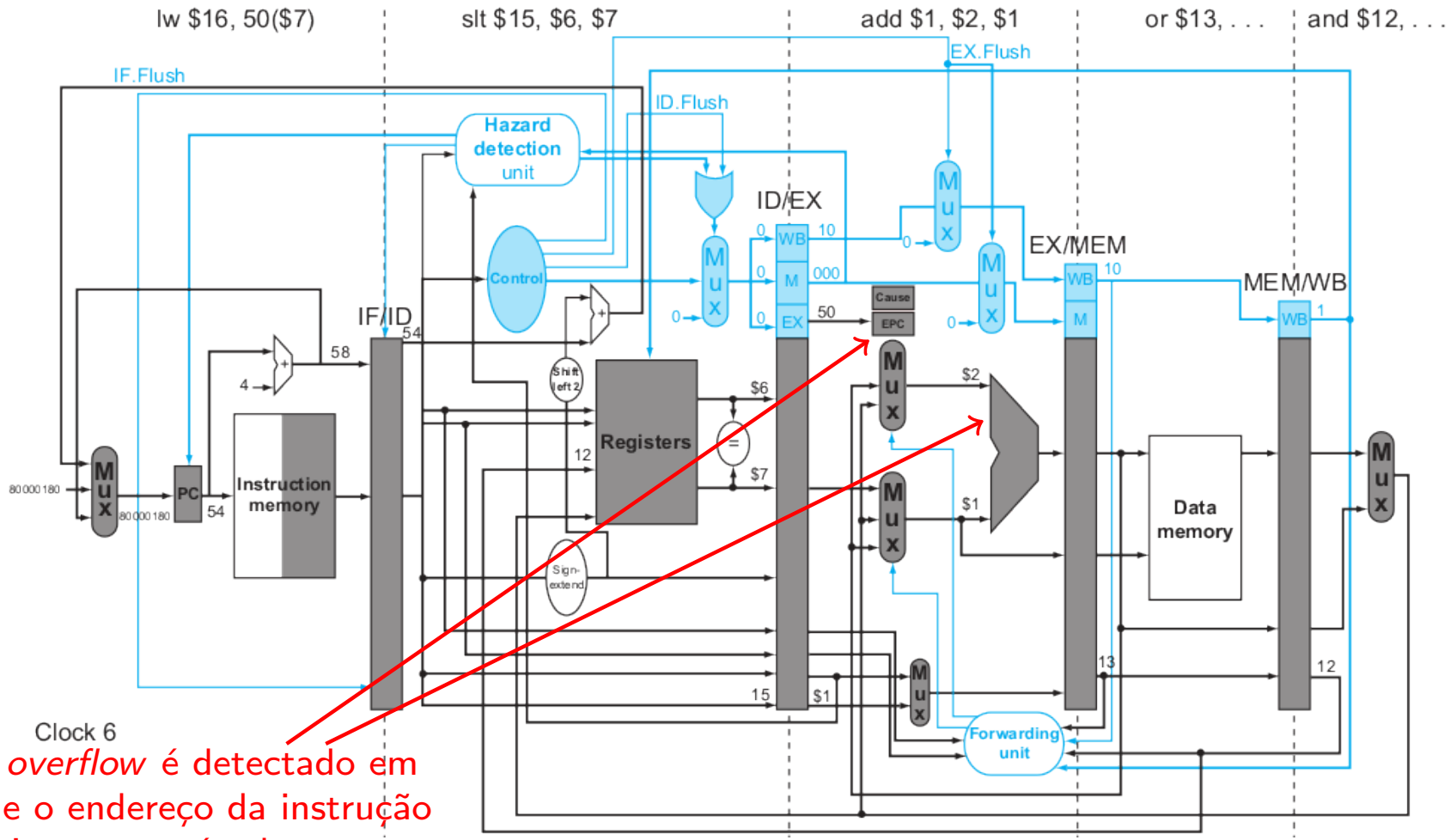
Exceções em MIPS: Exemplo



Fonte: [1]

Interrupções e Exceções

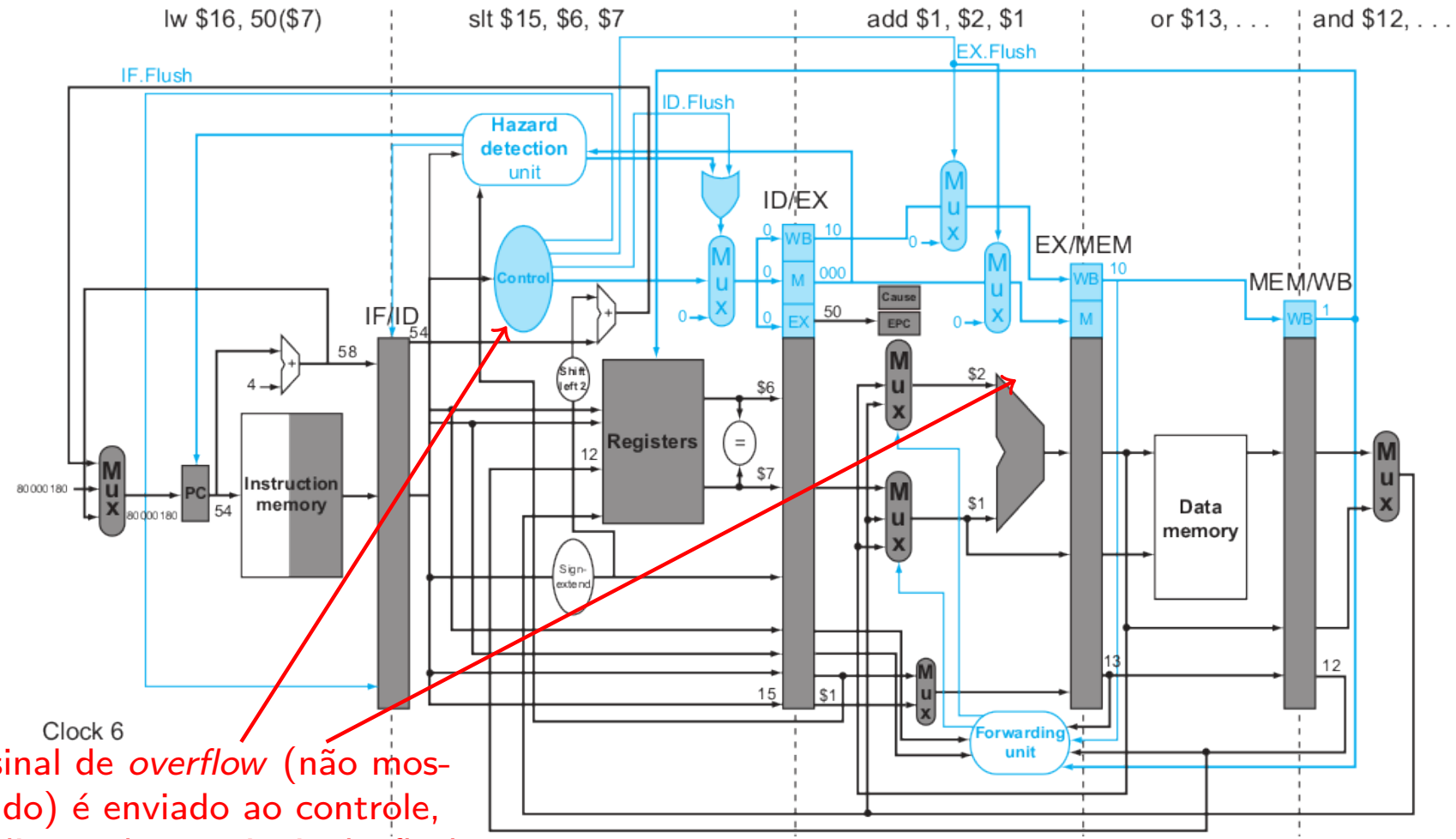
Exceções em MIPS: Exemplo



Fonte: [1]

Interrupções e Exceções

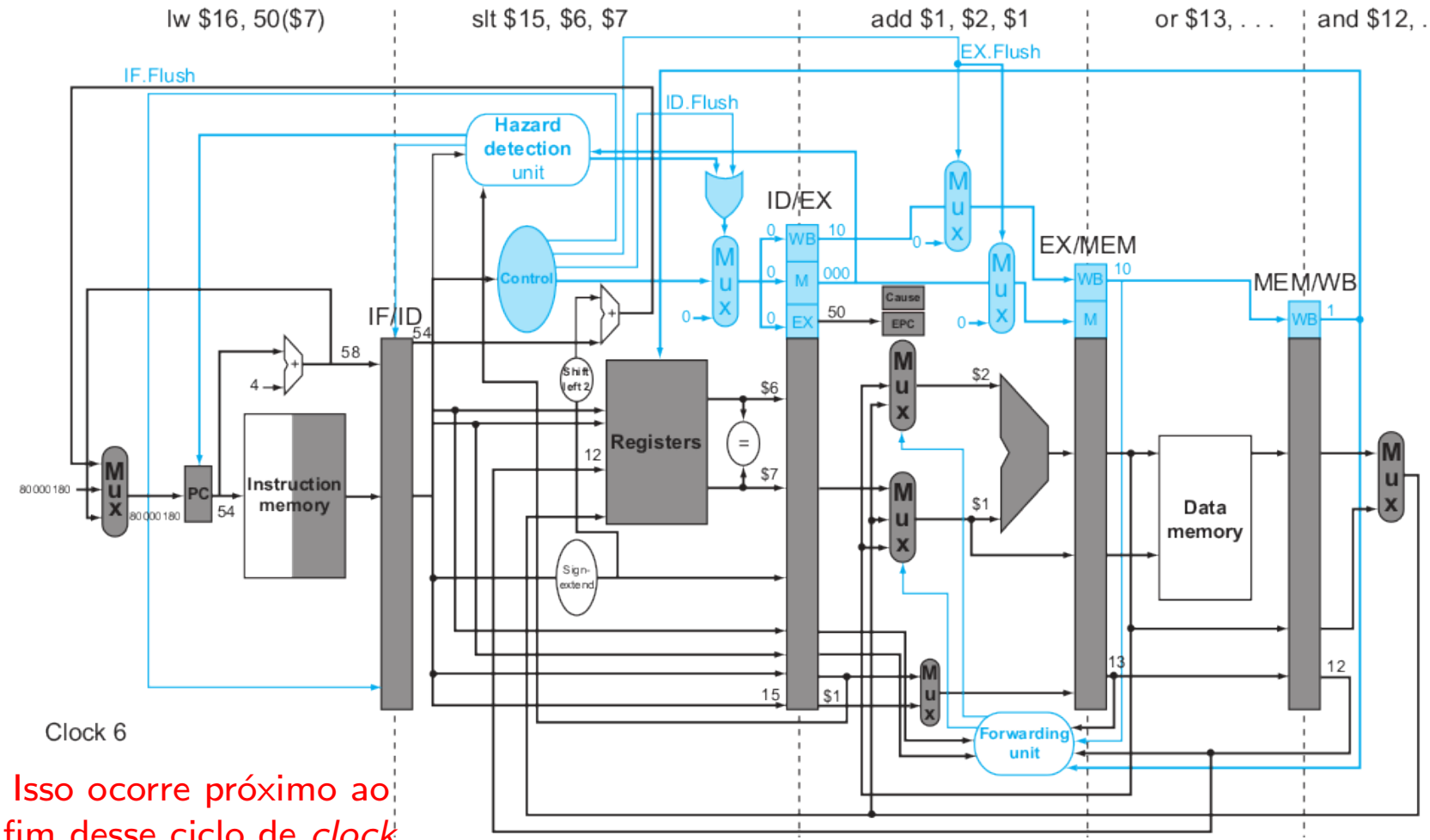
Exceções em MIPS: Exemplo



Fonte: [1]

Interrupções e Exceções

Exceções em MIPS: Exemplo

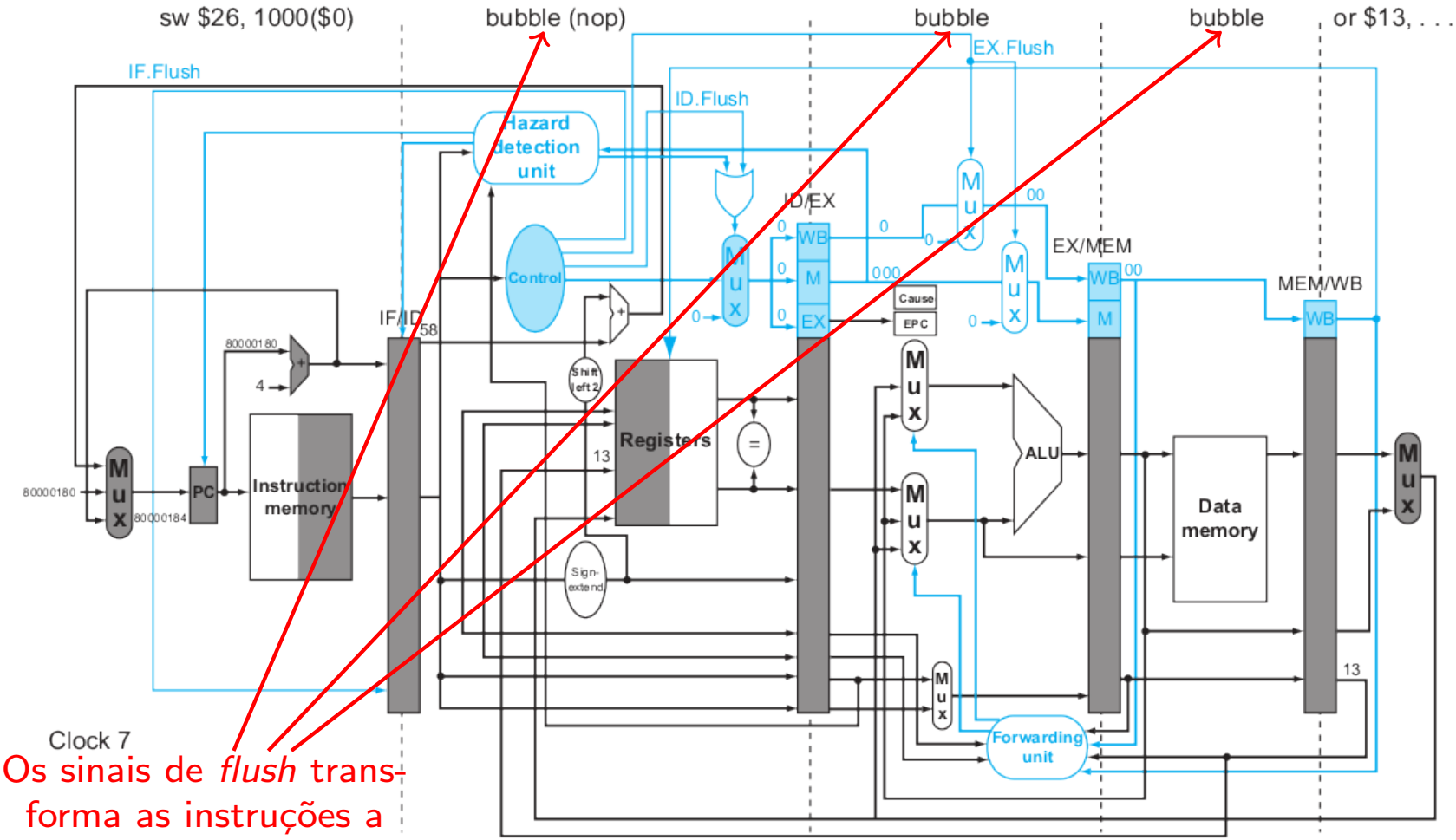


Isso ocorre próximo ao fim desse ciclo de *clock*

Fonte: [1]

Interrupções e Exceções

Exceções em MIPS: Exemplo

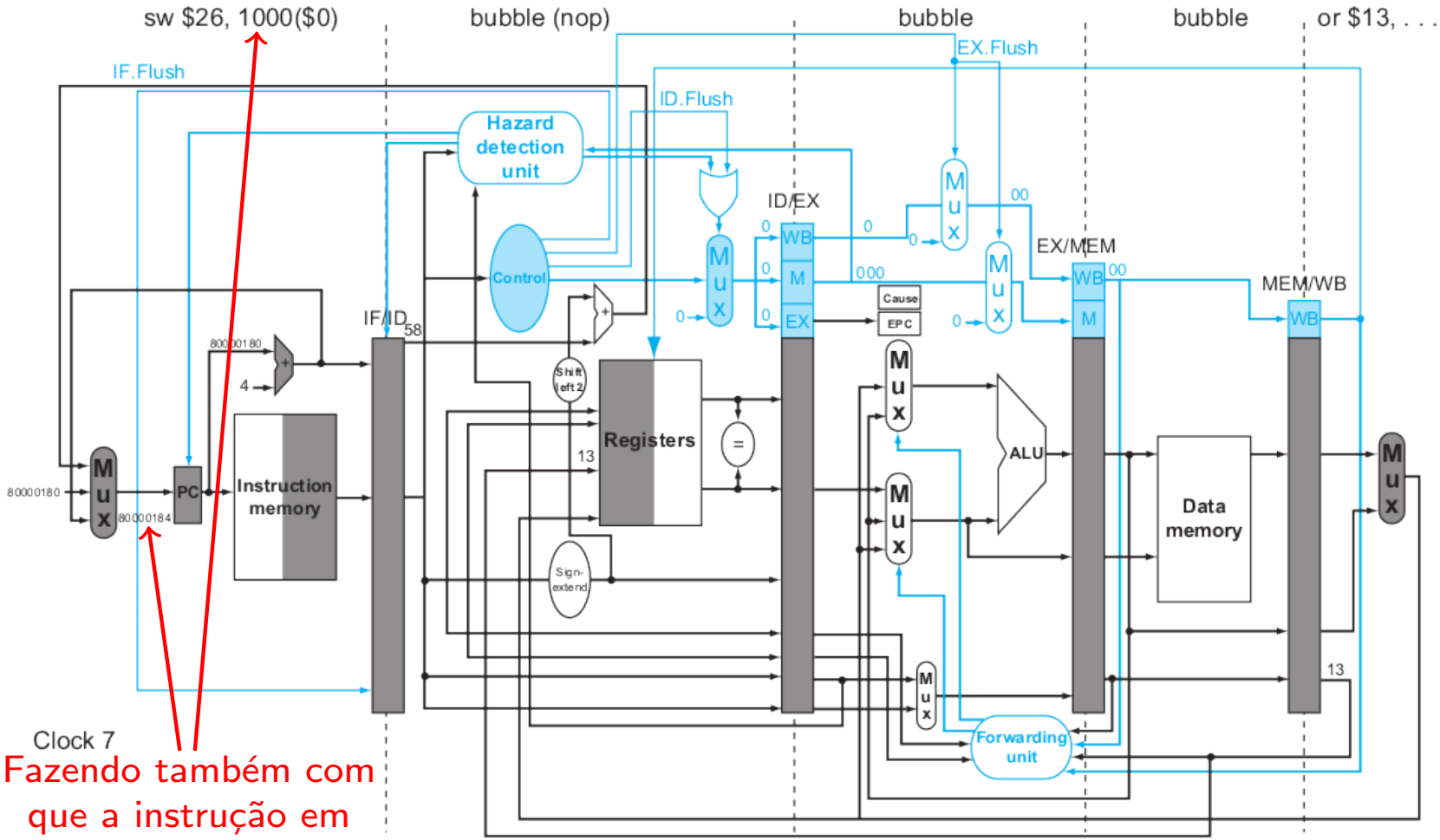


Os sinais de *flush* transforma as instruções a partir de add em bolhas

Fonte: [1]

Interrupções e Exceções

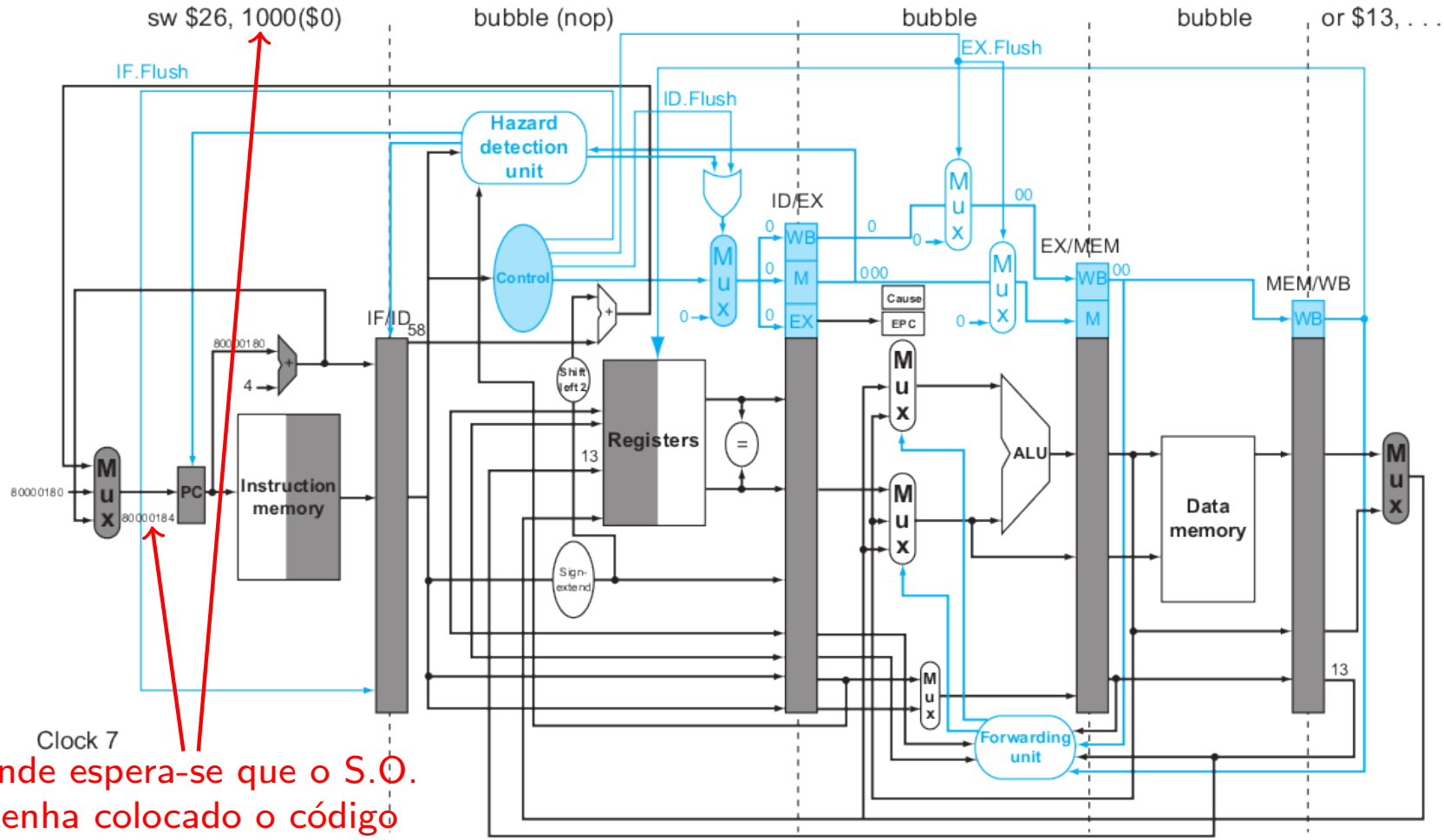
Exceções em MIPS: Exemplo



Fonte: [1]

Interrupções e Exceções

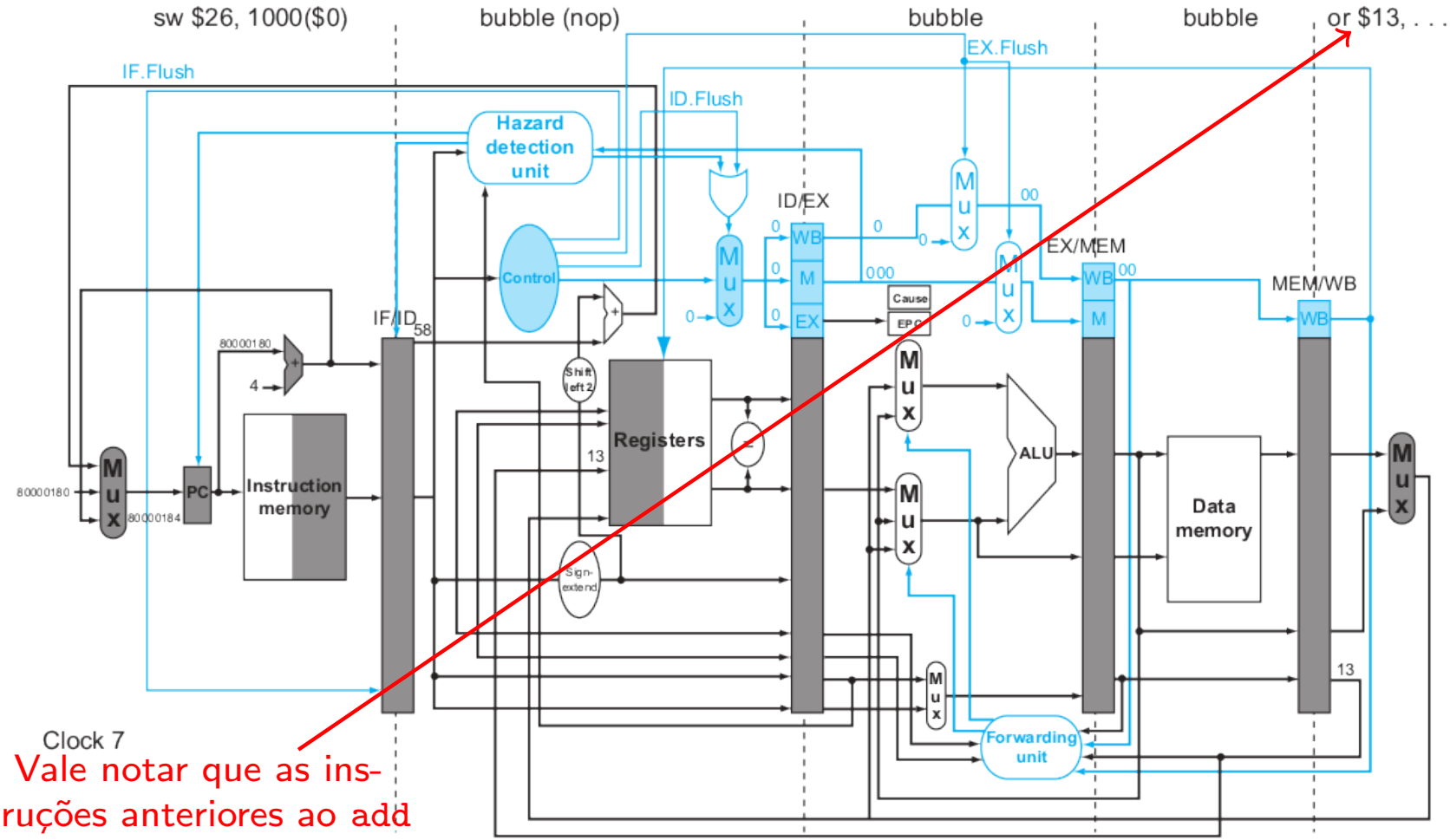
Exceções em MIPS: Exemplo



Fonte: [1]

Interrupções e Exceções

Exceções em MIPS: Exemplo



Fonte: [1]

Interrupções e Exceções

Exceções em MIPS: Múltiplas Exceções

- Múltiplas exceções podem ocorrer no mesmo ciclo de *clock*
  - Caso em que devem ser priorizadas
  - Em muitas implementações, são ordenadas de modo a interromper a instrução mais antina na *pipeline*

Interrupções e Exceções

Exceções em MIPS: Múltiplas Exceções

- Múltiplas exceções podem ocorrer no mesmo ciclo de *clock*
  - Caso em que devem ser priorizadas
  - Em muitas implementações, são ordenadas de modo a interromper a instrução mais antina na *pipeline*
- Interrupções (E/S) tem a vantagem de não serem associadas a instruções específicas
  - Temos mais flexibilidade com relação a quando interromper, e podemos usar os mesmos mecanismos das exceções

Interrupções e Exceções

Exceções em MIPS: Múltiplas Exceções

- No caso de múltiplas exceções, nada muda com o `EPC`
  - Ele ainda mantém o endereço da instrução seguinte à interrompida

Interrupções e Exceções

Exceções em MIPS: Múltiplas Exceções

- No caso de múltiplas exceções, nada muda com o `EPC`
  - Ele ainda mantém o endereço da instrução seguinte à interrompida
- O registrador de causa, contudo, registra todas as interrupções em um mesmo ciclo de *clock*
  - Fazendo com que a rotina de tratamento tenha que casar cada exceção à sua instrução correspondente
  - Para isso, ajuda saber em que estágio da *pipeline* cada exceção pode ocorrer

Interrupções e Exceções

Exceções em MIPS: Múltiplas Exceções

- As exceções são mantidas no registrador de causa em um campo de exceções pendentes
  - Assim o hardware pode interromper a pipeline com base nas exceções ainda não tratadas, quando terminar o tratamento de alguma

Interrupções e Exceções

Exceções em MIPS

- É importante que o hardware e o S.O. trabalhem em conjunto
  - O *hardware* deve parar a instrução problemática; completar as anteriores; dar um *flush* nas posteriores; carregar o registrador de causa com a causa da exceção; carregar o `EPC` com o endereço da instrução seguinte à problemática; e desviar para um endereço de memória pré-definido
  - O S.O. deve olhar a causa da exceção e agir apropriadamente, armazenando o código para seu tratamento no endereço de memória definido pelo *hardware*

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)