# Arquitetura de Computadores
# ACH2055
## Aula 10 – Processadores Superescalares

Norton Trevisan Roman
(norton@usp.br)

29 de novembro de 2019

# Paralelismo em Nível de Instrução

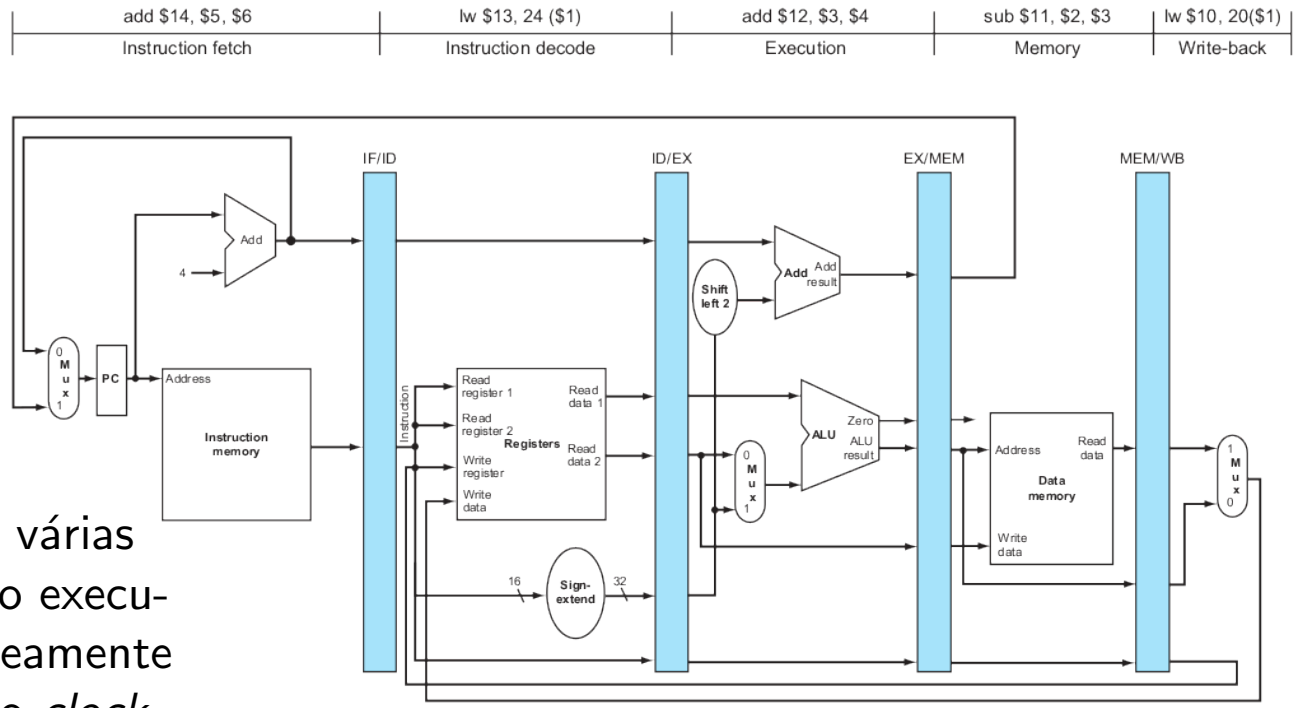
## *Pipeline*

- Vimos que o uso de *pipeline* corresponde a uma forma de paralelismo

# Paralelismo em Nível de Instrução

## *Pipeline*

- Vimos que o uso de *pipeline* corresponde a uma forma de paralelismo



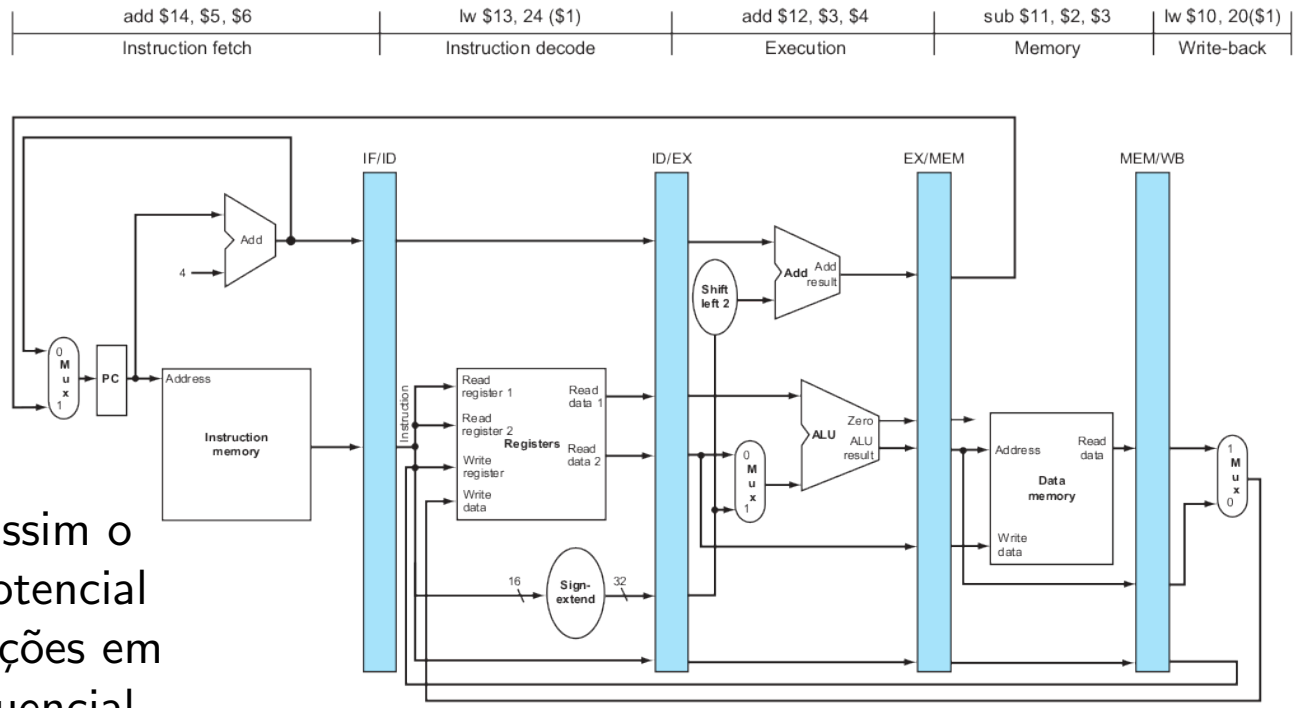

Uma vez que várias instruções estão executando simultaneamente a cada ciclo do *clock*

Fonte: [1]

# Paralelismo em Nível de Instrução

## *Pipeline*

- Vimos que o uso de *pipeline* corresponde a uma forma de paralelismo



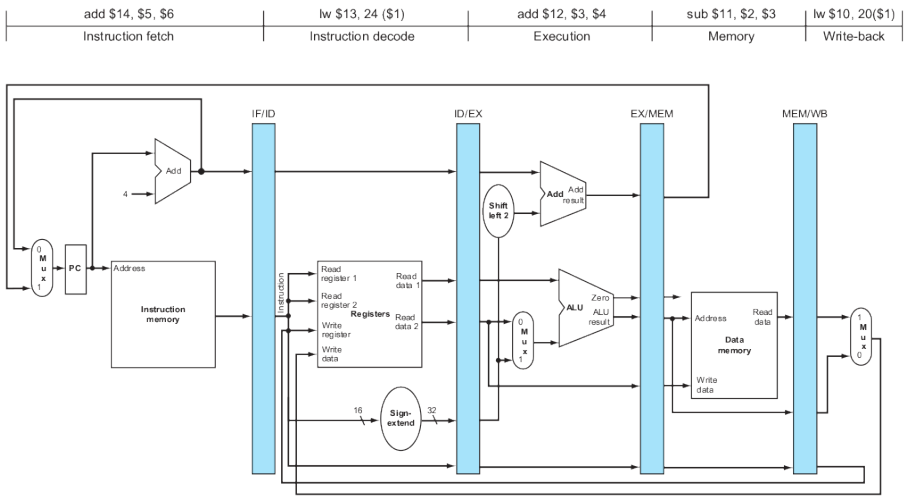

Explorando assim o paralelismo potencial entre as instruções em um fluxo sequencial

Fonte: [1]

# Paralelismo em Nível de Instrução

## *Pipeline*

- Esse tipo de paralelismo é chamado de **Paralelismo em Nível de Instrução**
  - *Instruction-level Paralelism* (ILP)
  - Referindo-se ao grau com que, em média, as instruções de um programa podem ser executadas em paralelo




Fonte: [1]

# Paralelismo em Nível de Instrução

Há 2 modos de se aumentar o paralelismo

# Paralelismo em Nível de Instrução

## Há 2 modos de se aumentar o paralelismo

- Ou aumentamos o comprimento da *pipeline*
  - De modo a sobrepor mais instruções
  - E arriscando *stalls* durarem muito

# Paralelismo em Nível de Instrução

## Há 2 modos de se aumentar o paralelismo

- Ou aumentamos o comprimento da *pipeline*
  - De modo a sobrepor mais instruções
  - E arriscando *stalls* durarem muito
- Ou replicamos os componentes internos do computador
  - De modo a rodar mais de uma instrução em cada estágio da *pipeline*
  - Técnica conhecida como **expedição múltipla** (*multiple issue*)

# Paralelismo em Nível de Instrução

## Expedição múltipla

- Com expedição múltipla podemos conseguir que $CPI < 1$
  - Ou, Alternativamente, $IPC > 1$ (**Instruções por Ciclo**)
  - $CPI = \frac{1}{ICP}$, então $CPI = 0.25 \Rightarrow IPC = 4$

# Paralelismo em Nível de Instrução

## Expedição múltipla

- Com expedição múltipla podemos conseguir que $CPI < 1$
  - Ou, Alternativamente, $IPC > 1$ (**Instruções por Ciclo**)
  - $CPI = \frac{1}{ICP}$, então $CPI = 0.25 \Rightarrow IPC = 4$
- Há contudo restrições quanto a que instruções podem rodar simultaneamente
  - Especialmente com relação a dependências

# Paralelismo em Nível de Instrução

## Expedição múltipla: Implementação

- Há dois tipos principais de implementação de expedição múltipla
  - A depender se as decisões serão feitas estatica ou dinamicamente

# Paralelismo em Nível de Instrução

## Expedição múltipla: Implementação

- Há dois tipos principais de implementação de expedição múltipla
  - A depender se as decisões serão feitas estatica ou dinamicamente
- **Expedição Múltipla Estática**
  - As decisões são tomadas em tempo de compilação

# Paralelismo em Nível de Instrução

## Expedição múltipla: Implementação

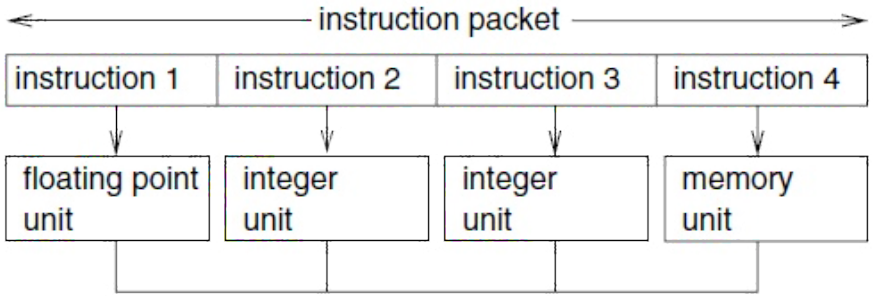
- Há dois tipos principais de implementação de expedição múltipla
  - A depender se as decisões serão feitas estatica ou dinamicamente
- **Expedição Múltipla Estática**
  - As decisões são tomadas em tempo de compilação
- **Expedição Múltipla Dinâmica**
  - As decisões são tomadas em tempo de execução

# Expedição Múltipla Estática

# Expedição Múltipla Estática

## Pacote de Expedição

- Necessitam de ajuda do compilador para determinar o **pacote de expedição**
  - O conjunto de instruções lançadas em um determinado ciclo de *clock*, como se fossem uma única longa instrução

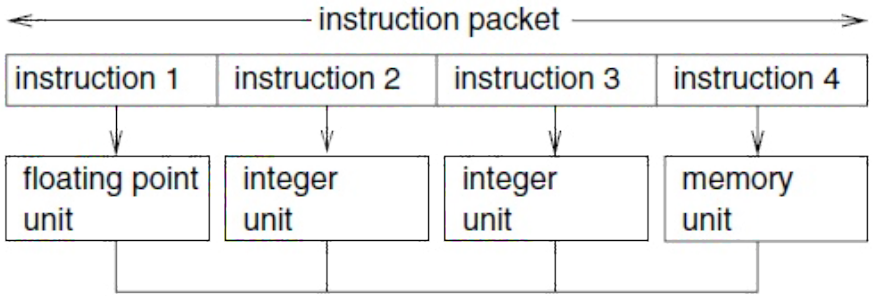



Fonte: `puyaa.ir/long-instruction-word-vliw-processors/`

# Expedição Múltipla Estática

## Pacote de Expedição

- Necessitam de ajuda do compilador para determinar o **pacote de expedição**
  - O conjunto de instruções lançadas em um determinado ciclo de *clock*, como se fossem uma única longa instrução

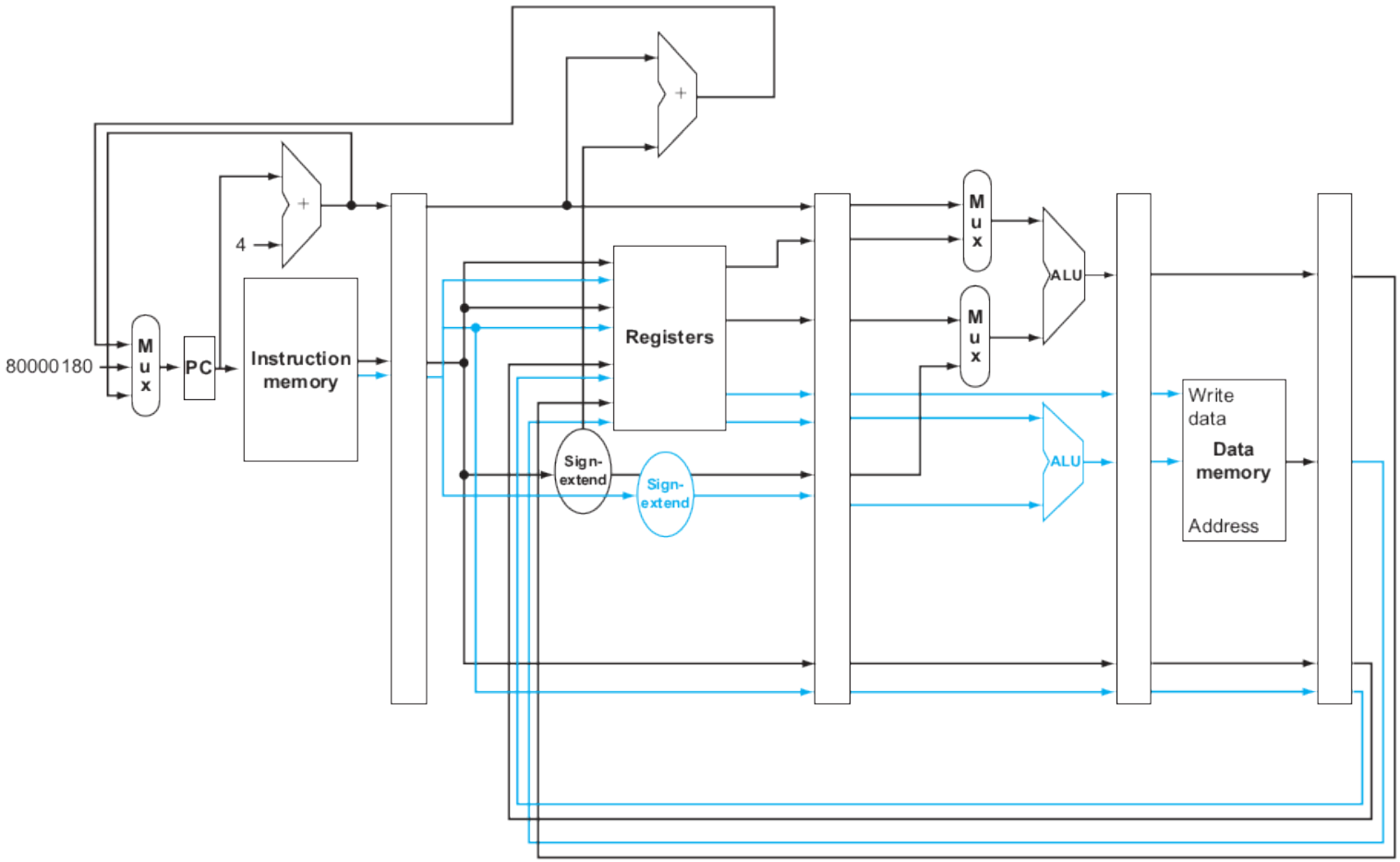



Fonte: `puyaa.ir/long-instruction-word-vliw-processors/`

- Essa interpretação rendeu seu nome original: *Very Long Instruction Word* (VLIW)

# Expedição Múltipla Estática

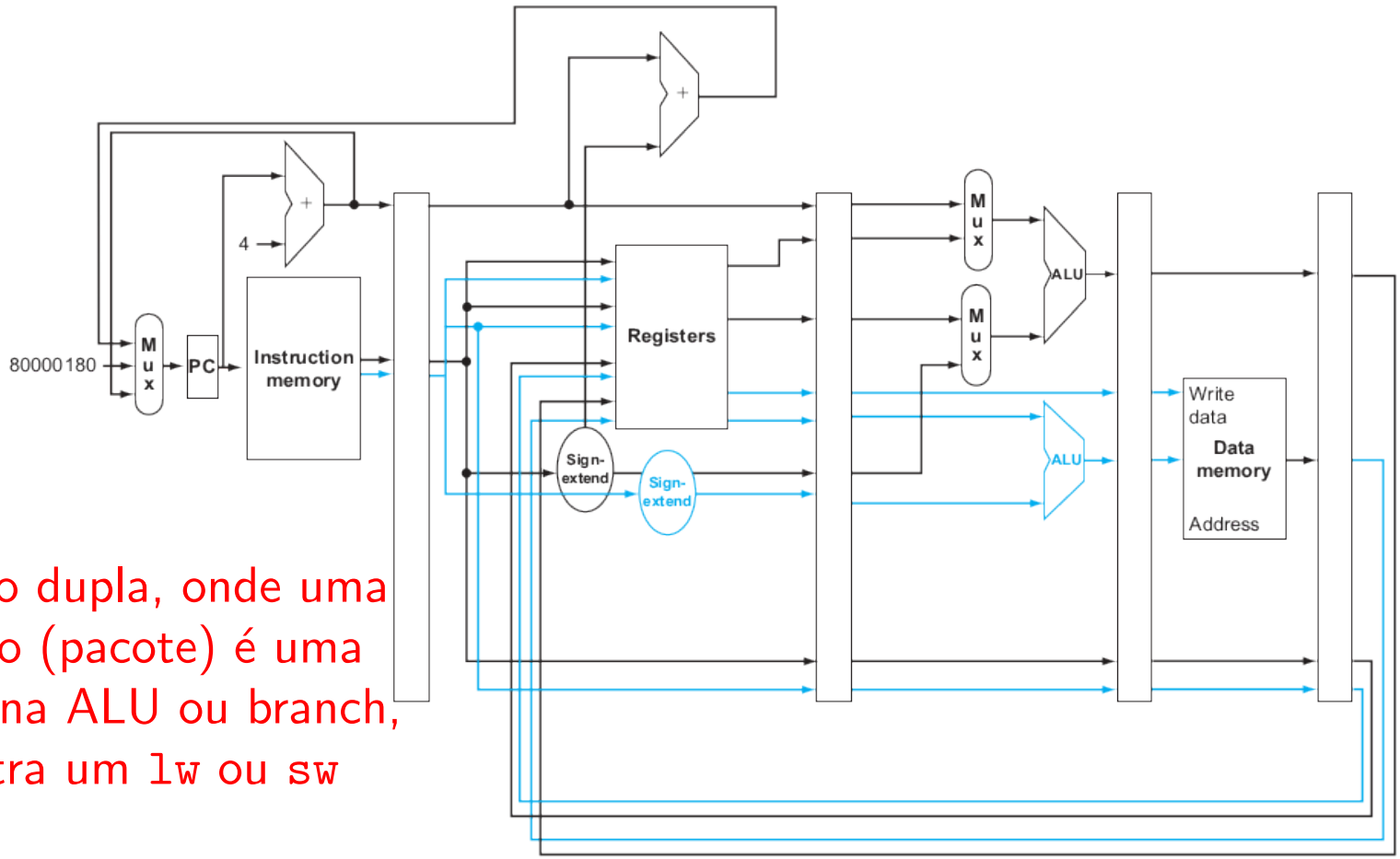
## Exemplo




Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

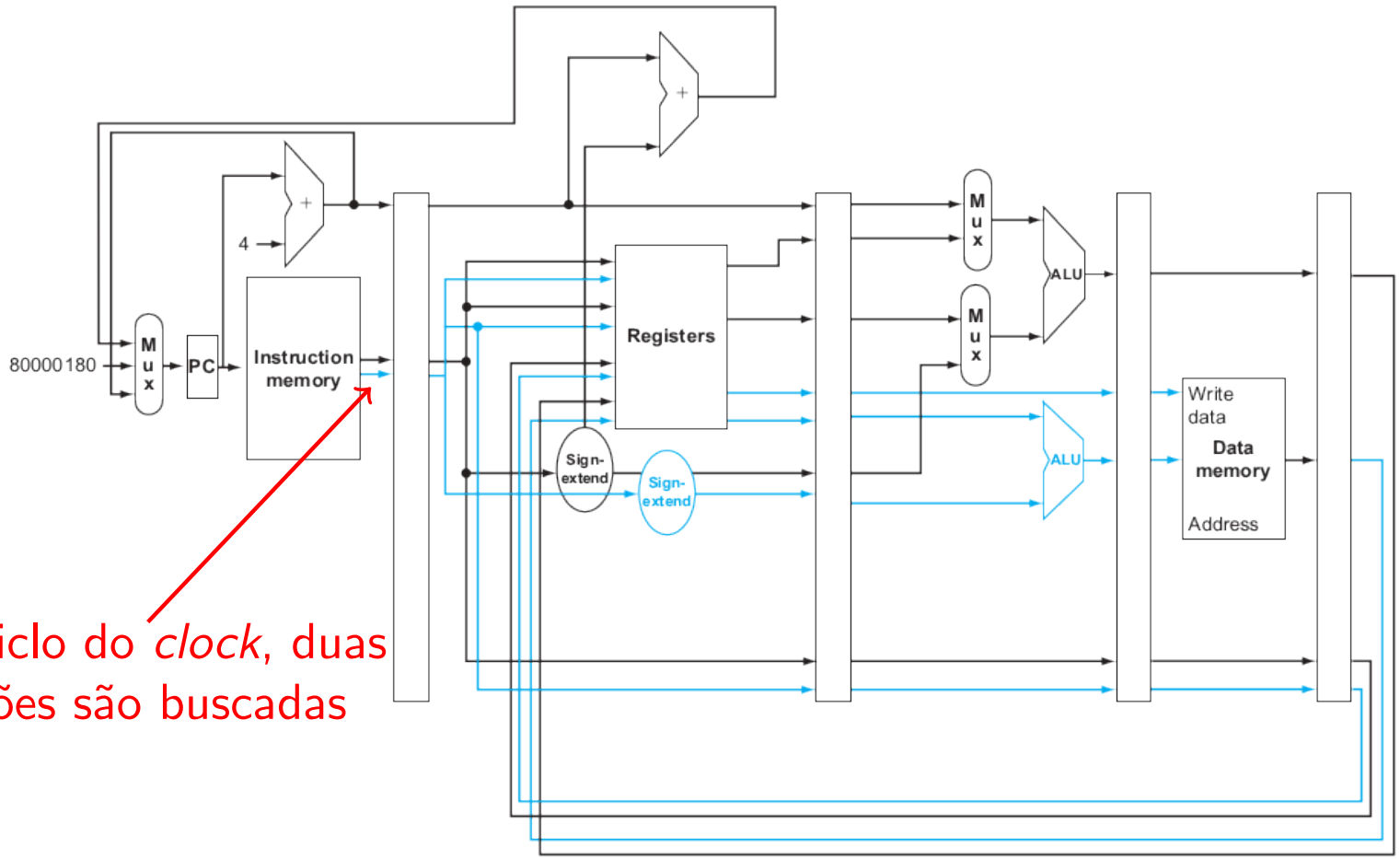



Expedição dupla, onde uma instrução (pacote) é uma operação na ALU ou branch, e a outra um `lw` ou `sw`

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

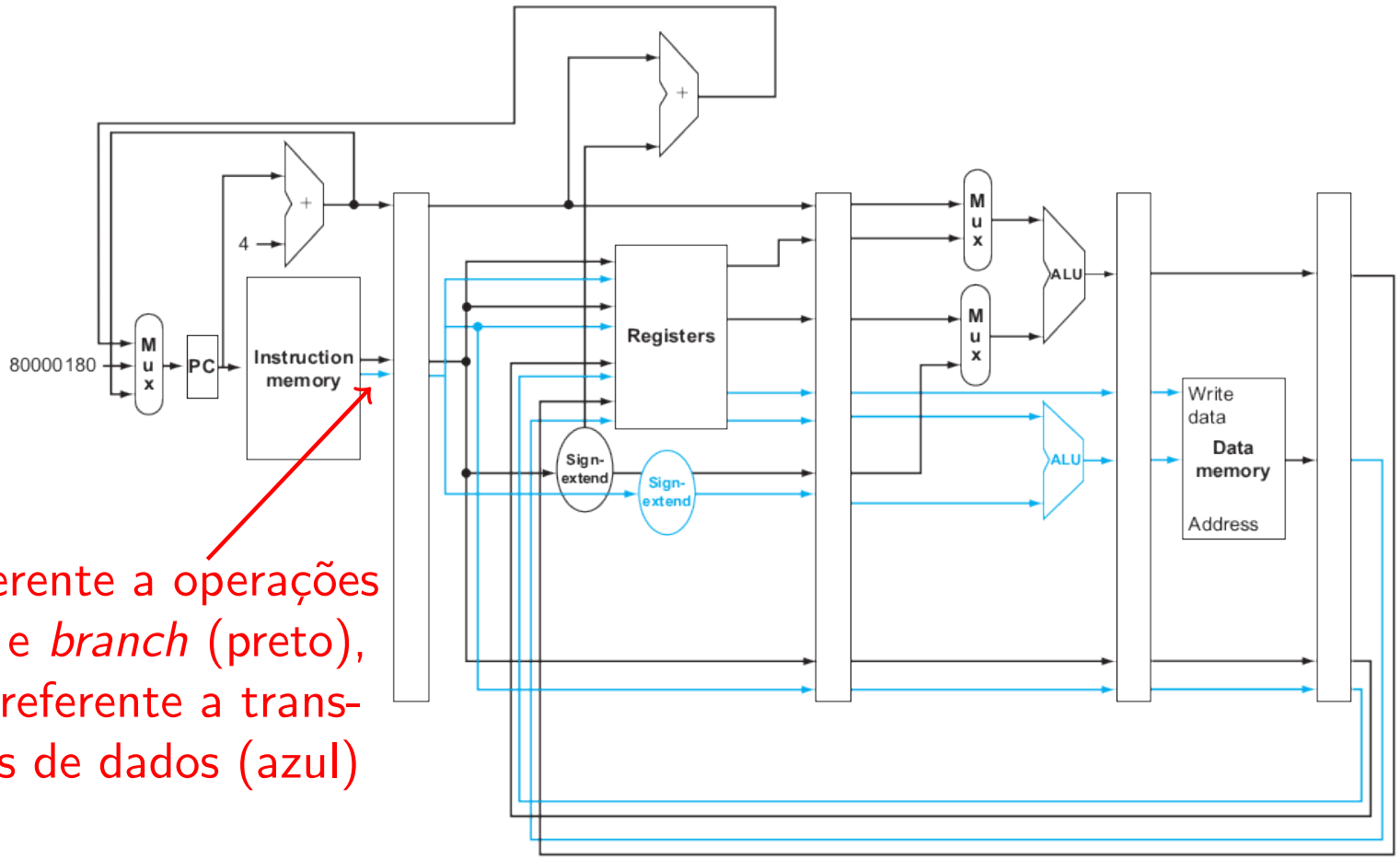



A cada ciclo do *clock*, duas instruções são buscadas

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

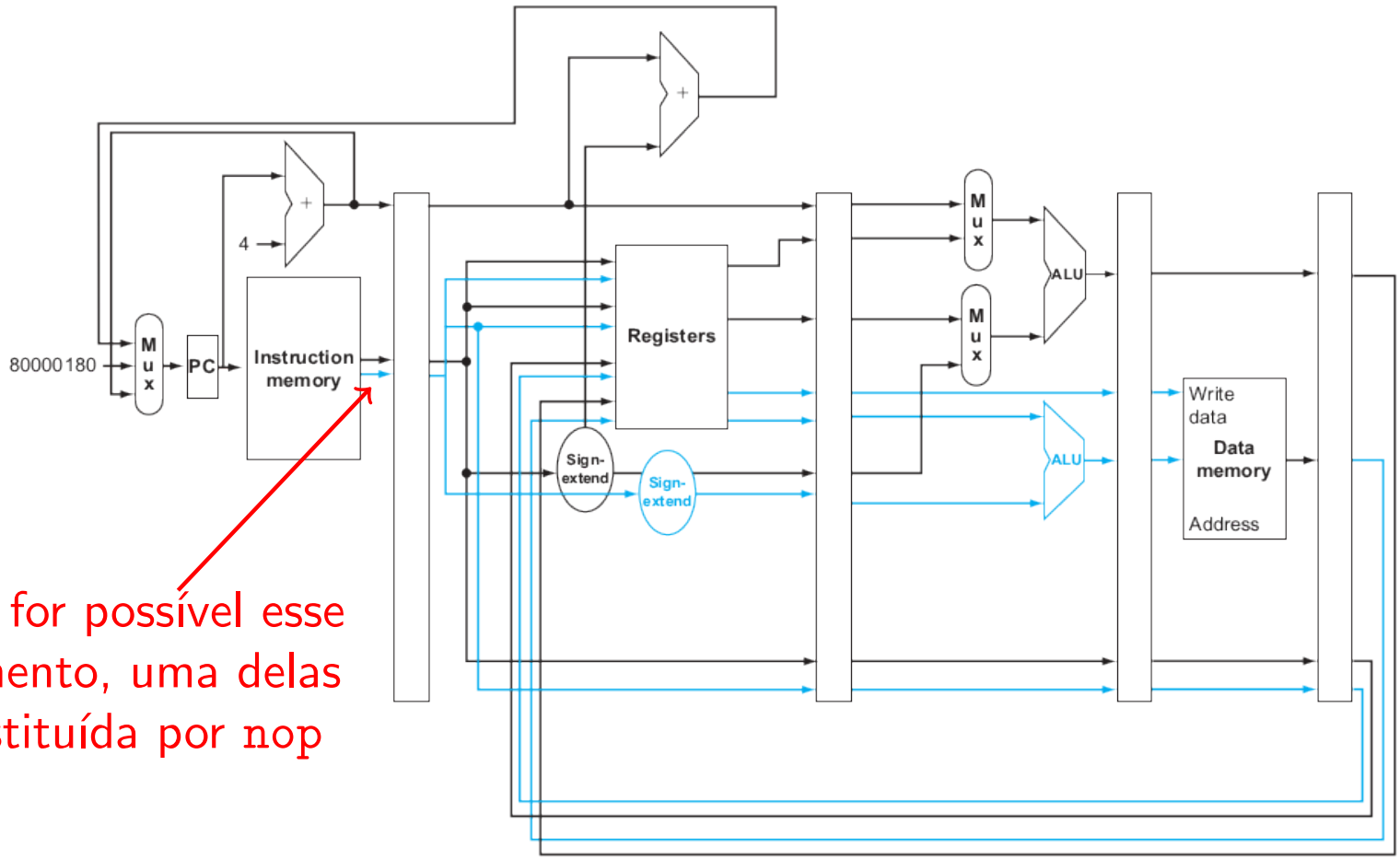



Uma referente a operações na ALU e *branch* (preto), e outra referente a transferências de dados (azul)

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

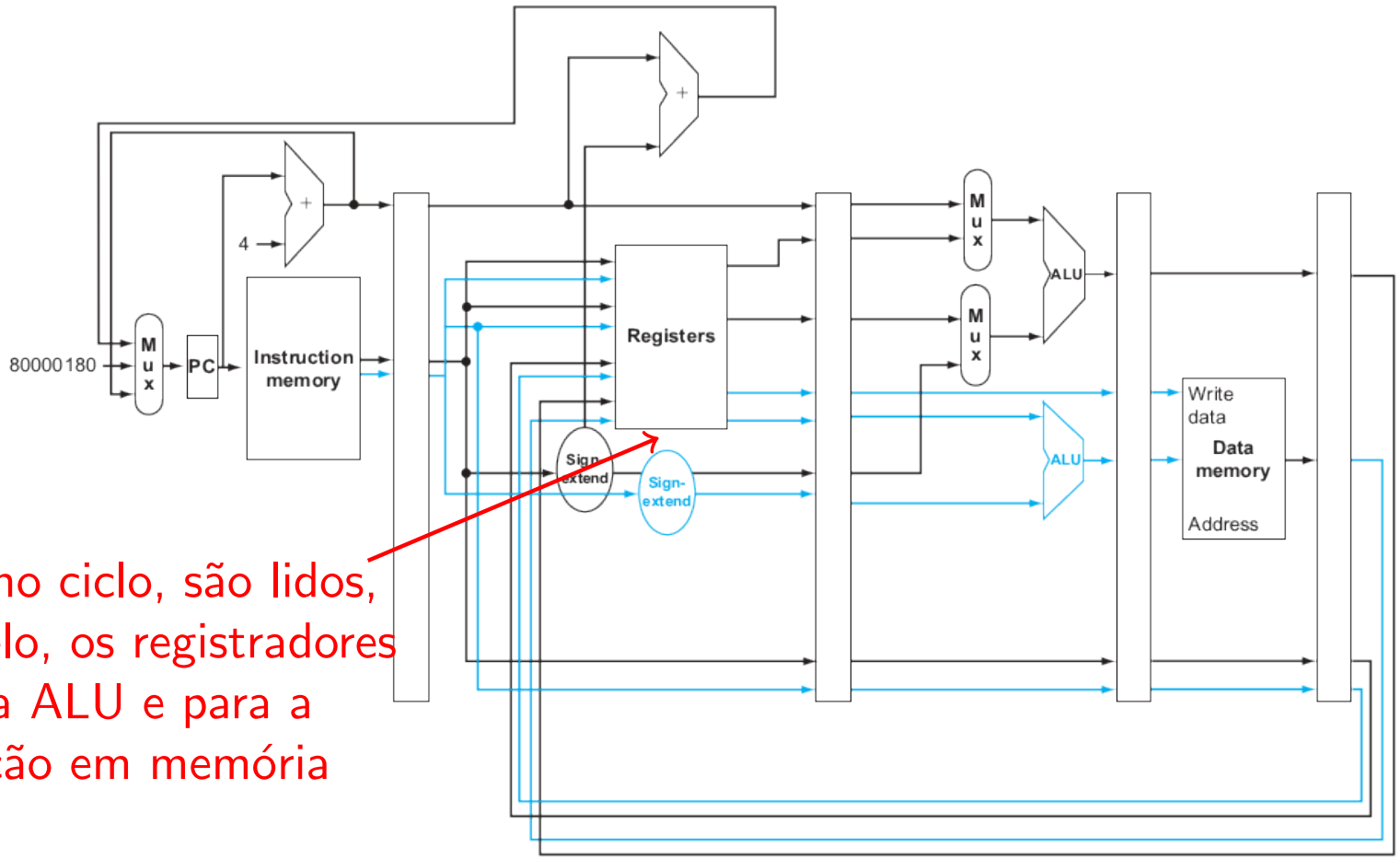



Se não for possível esse pareamento, uma delas é substituída por `nop`

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

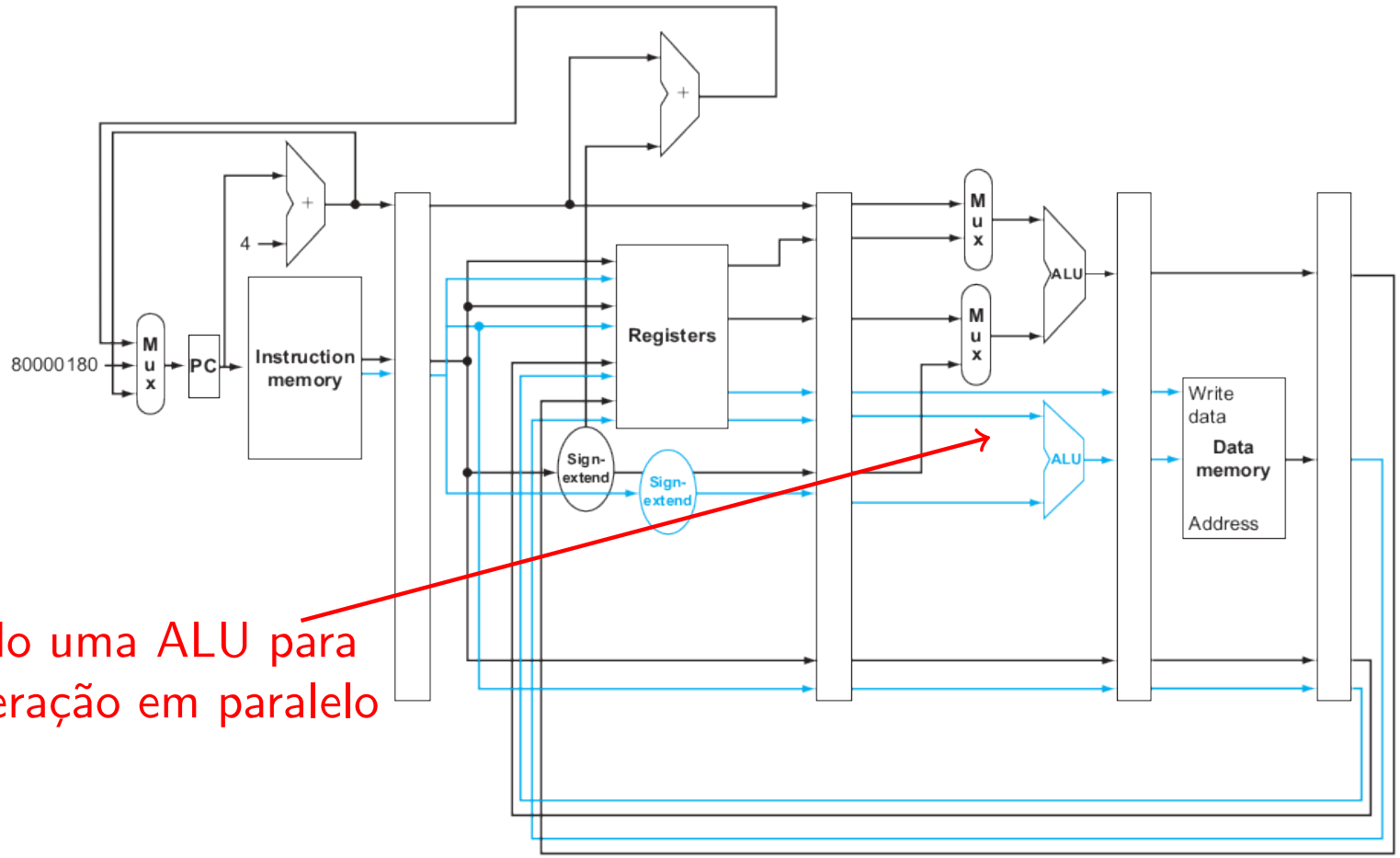



No mesmo ciclo, são lidos, em paralelo, os registradores para a ALU e para a operação em memória

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

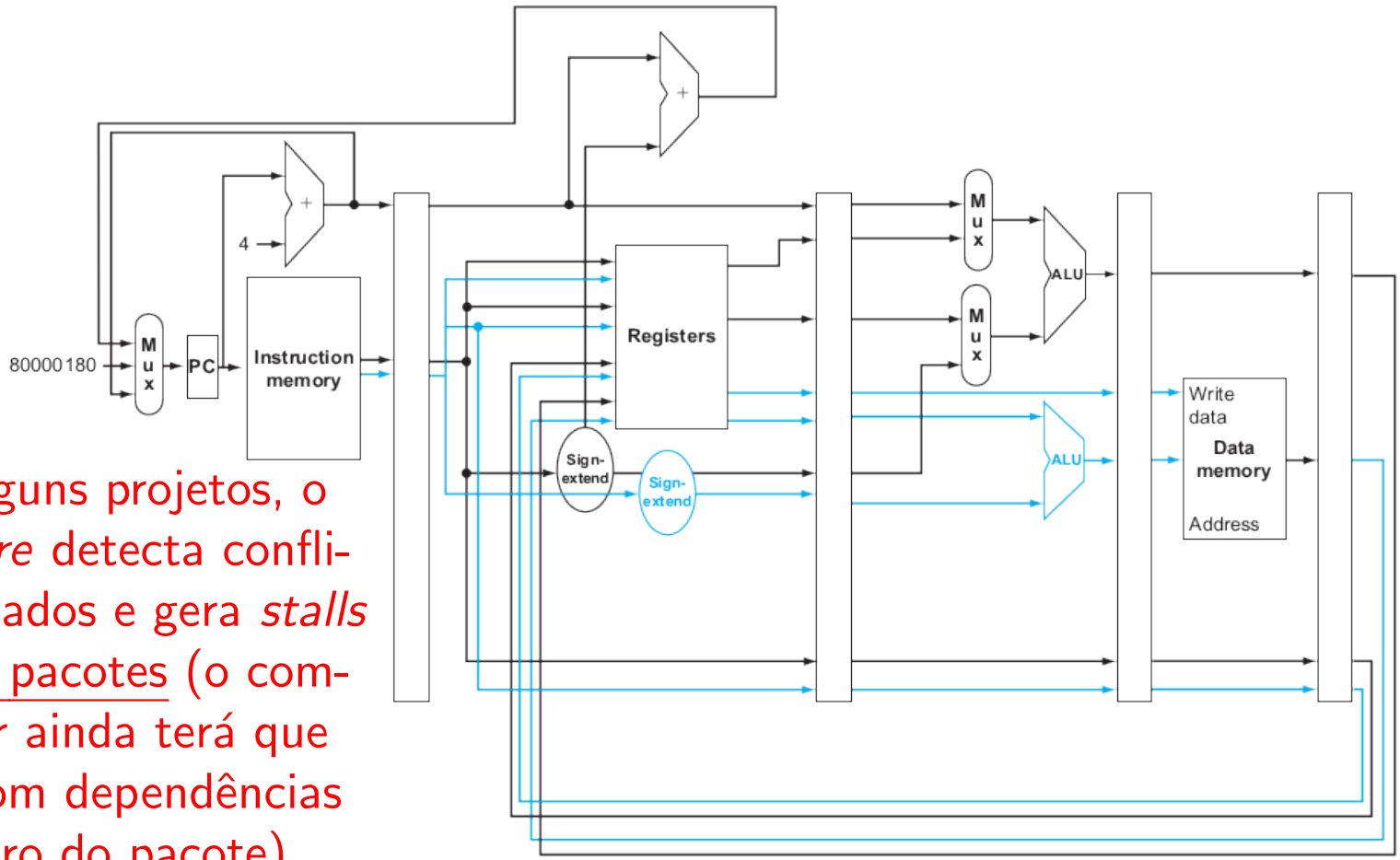



Existindo uma ALU para cada operação em paralelo

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

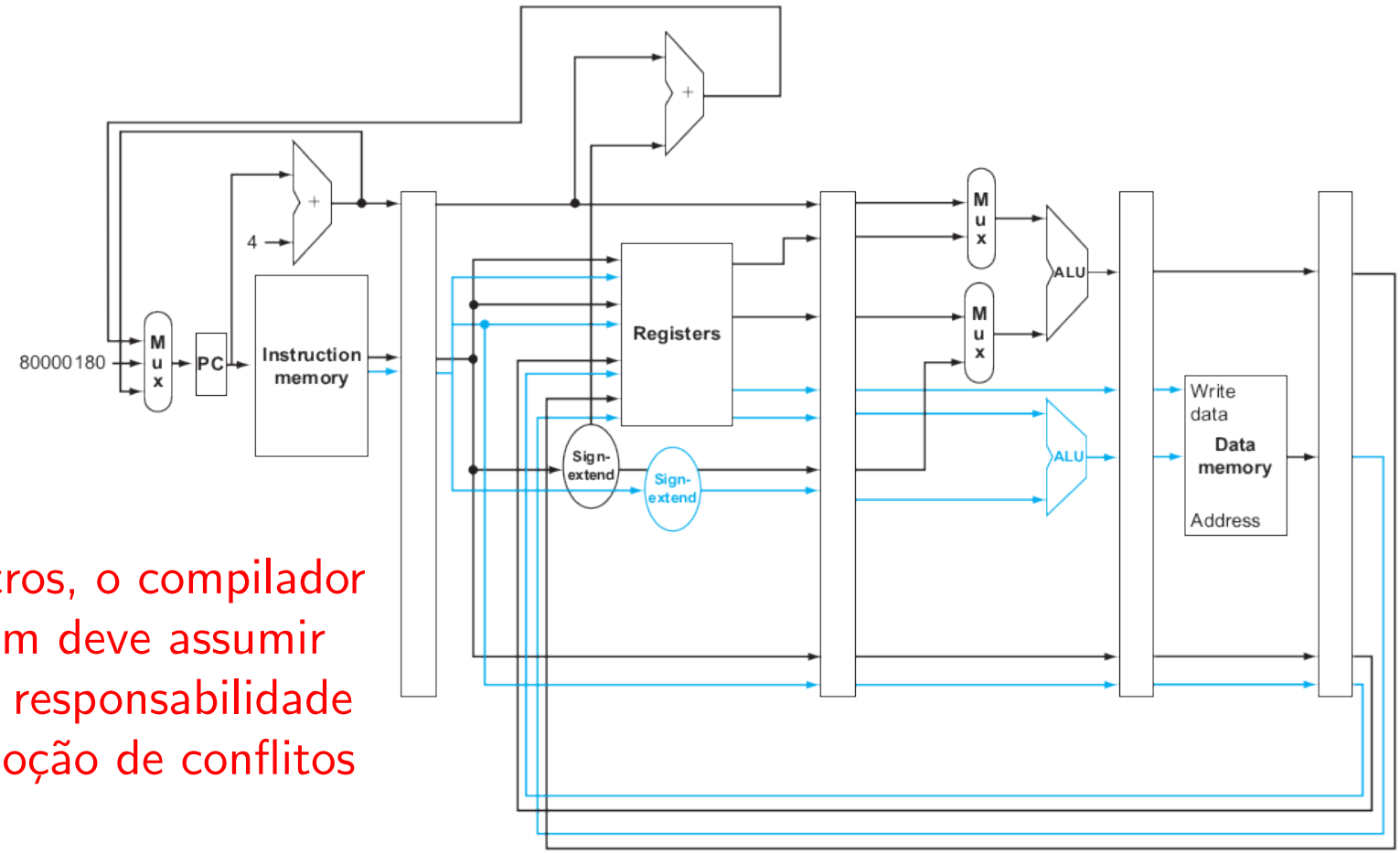



Em alguns projetos, o *hardware* detecta conflitos de dados e gera *stalls* entre 2 pacotes (o compilador ainda terá que lidar com dependências dentro do pacote)

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

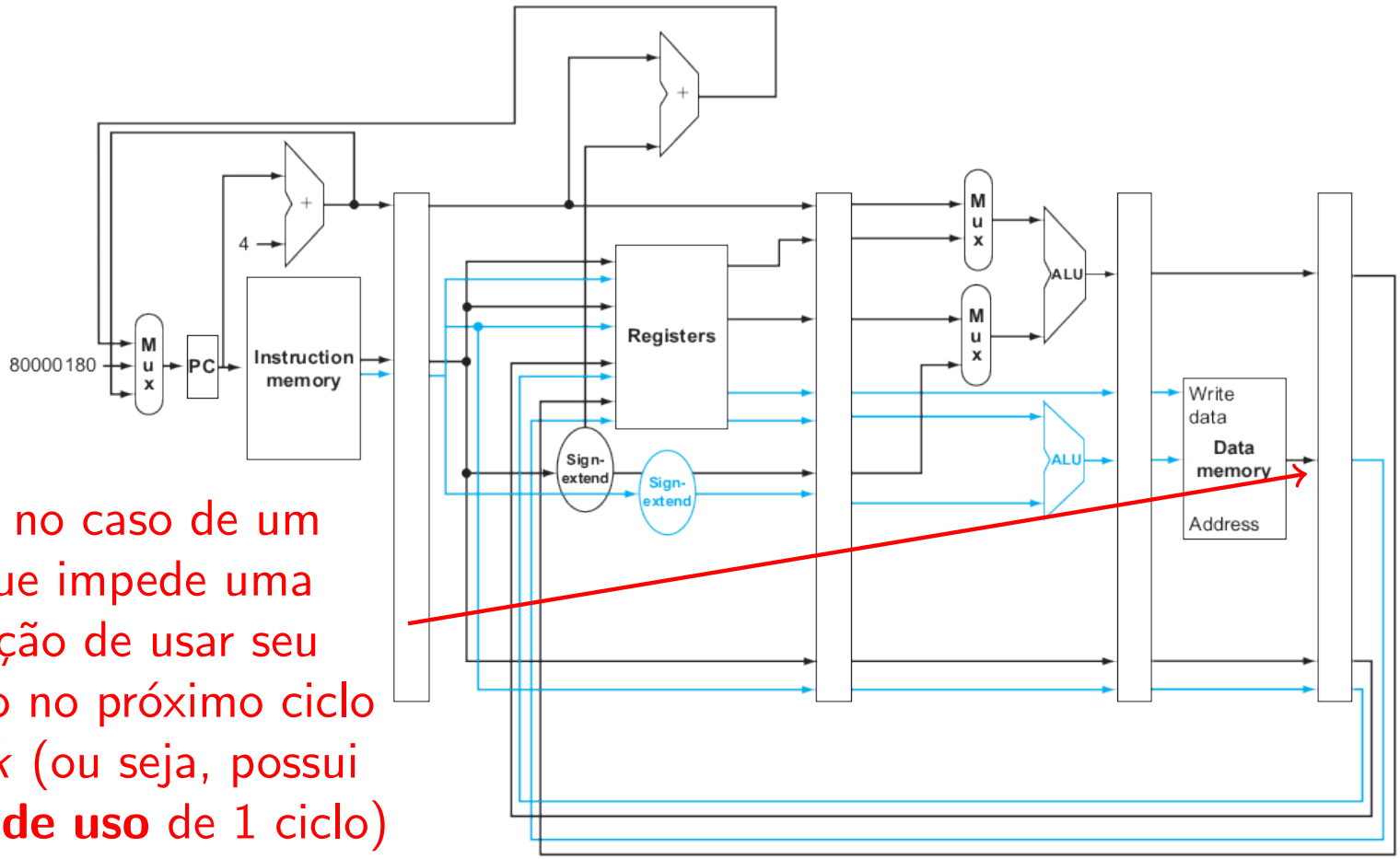



Em outros, o compilador é quem deve assumir toda a responsabilidade de remoção de conflitos

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

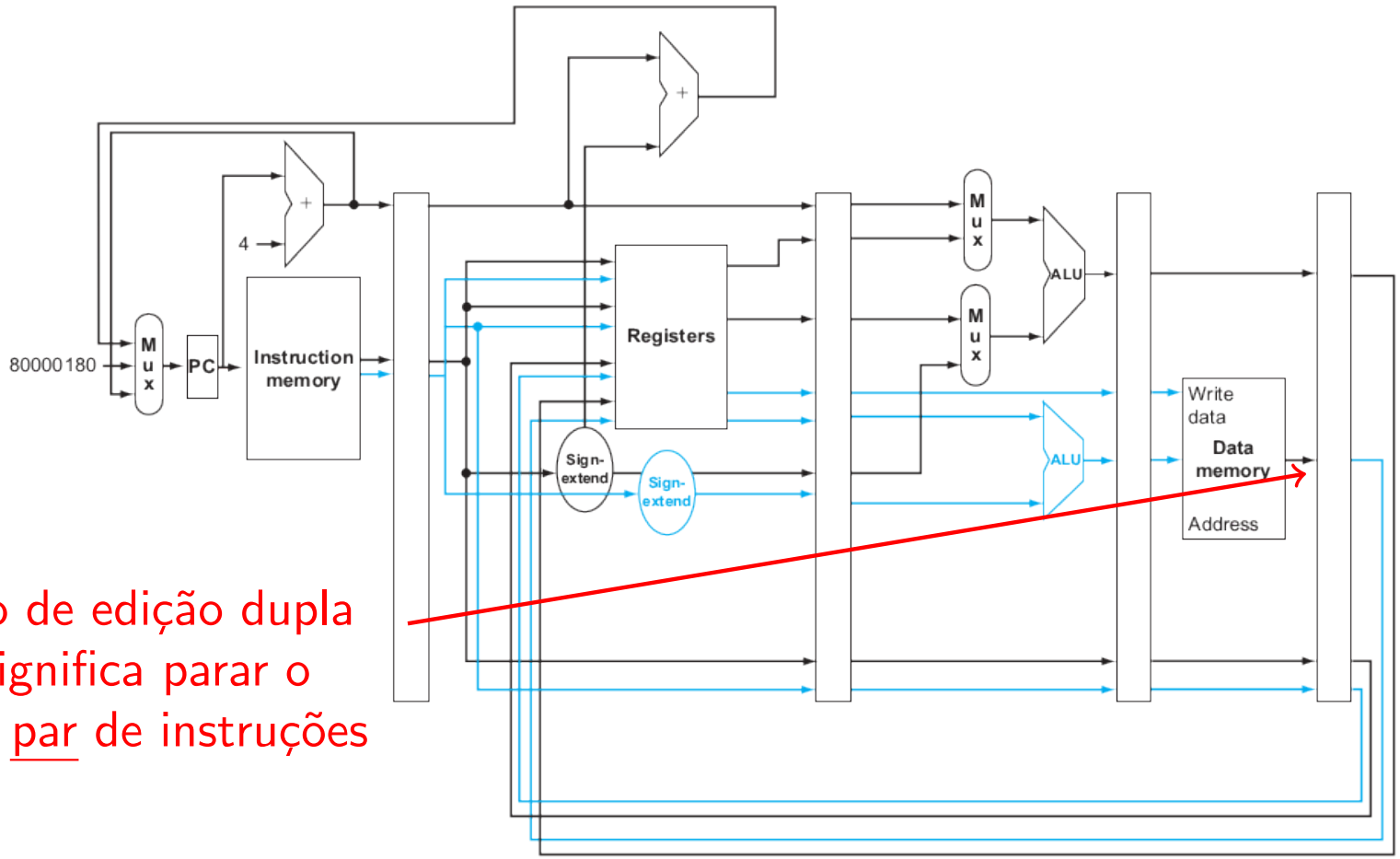



Como no caso de um `lw`, que impede uma instrução de usar seu resultado no próximo ciclo de *clock* (ou seja, possui **latência de uso** de 1 ciclo)

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

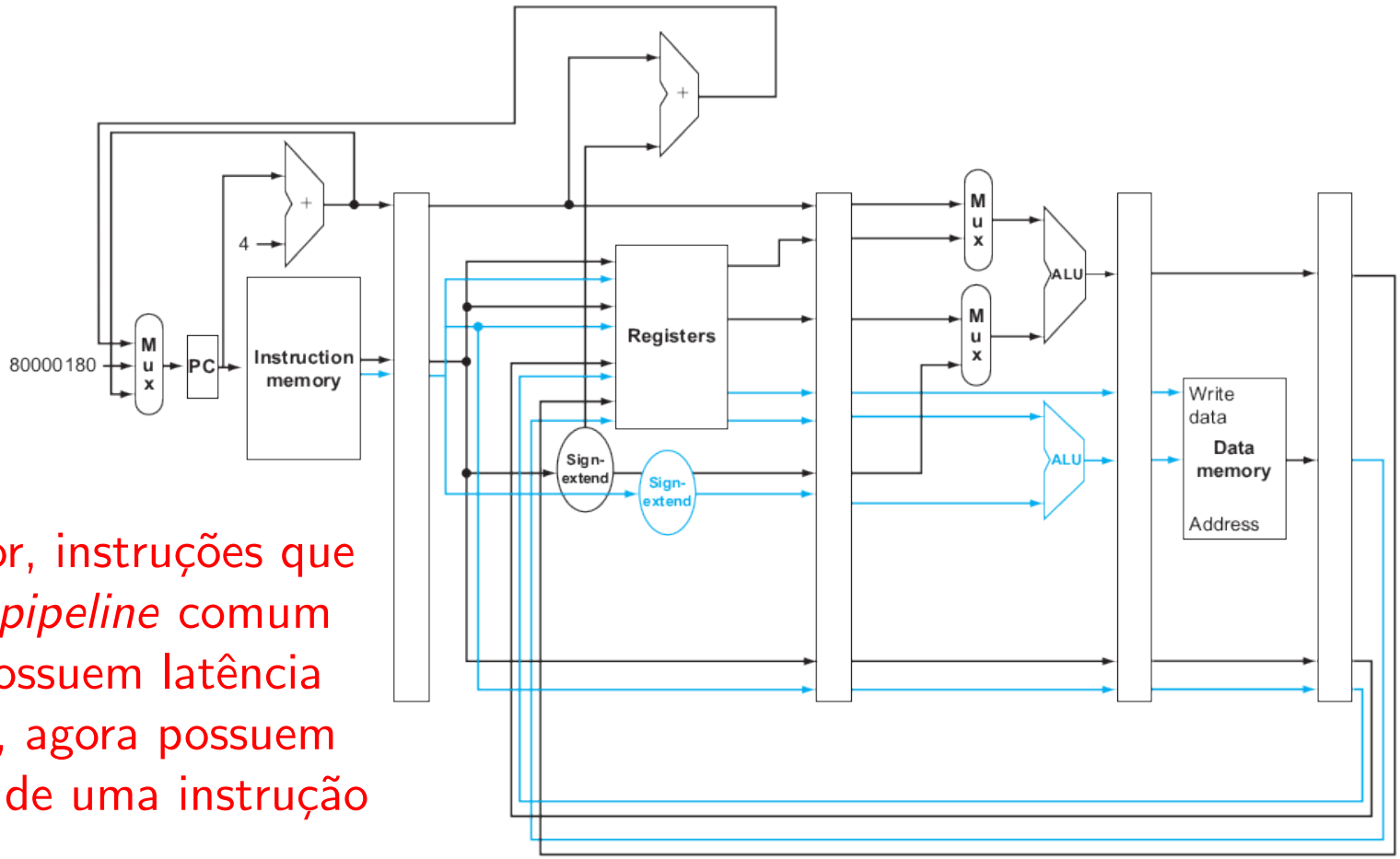



No caso de edição dupla isso significa parar o próximo par de instruções

Fonte: [1]

# Expedição Múltipla Estática

## Exemplo

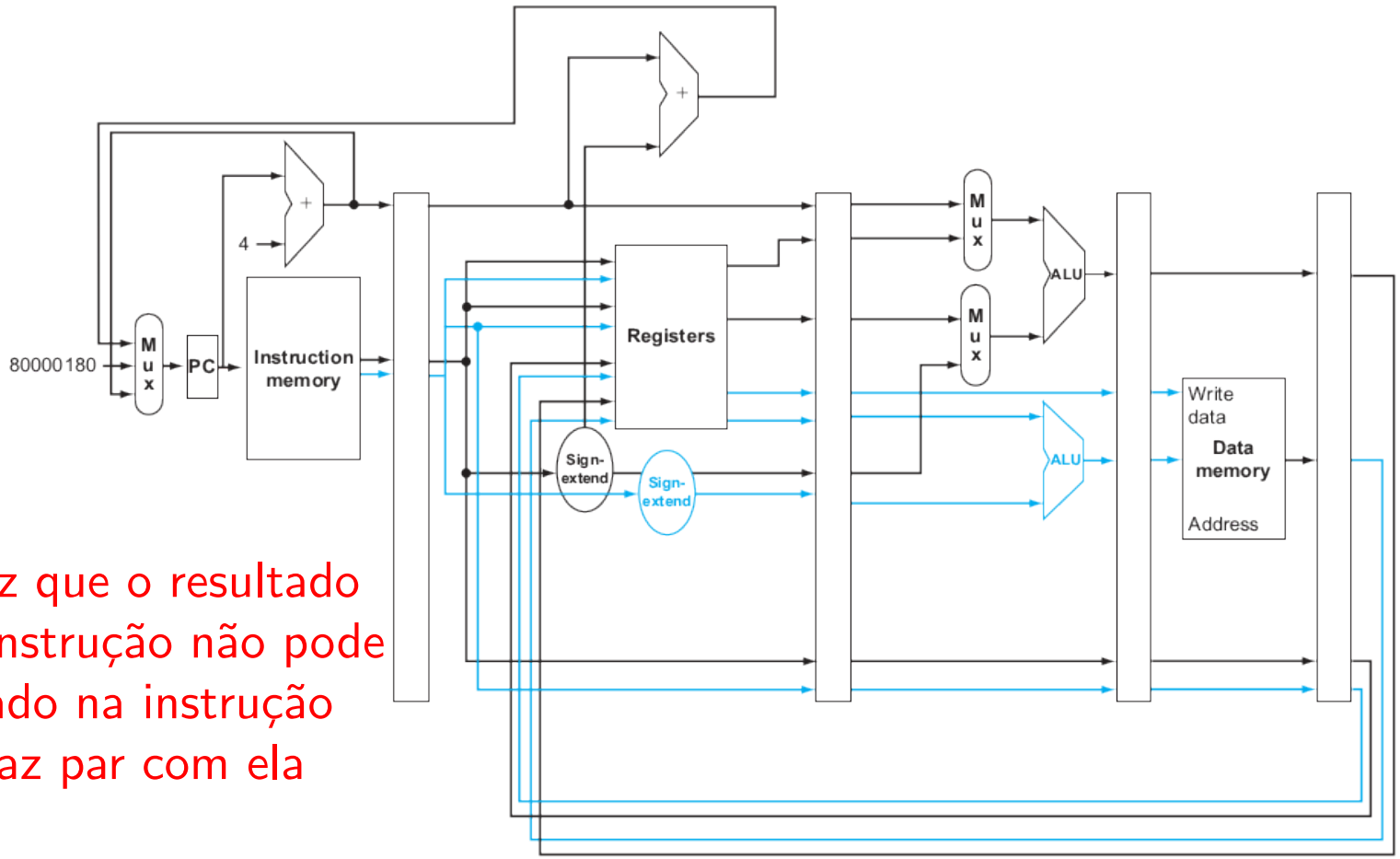



Fonte: [1]

Mas pior, instruções que numa *pipeline* comum não possuem latência de uso, agora possuem latência de uma instrução

# Expedição Múltipla Estática

## Exemplo




Uma vez que o resultado de uma instrução não pode ser usado na instrução que faz par com ela

Fonte: [1]

# Expedição Múltipla Dinâmica

# Expedição Múltipla Dinâmica

## Superescalar

- Processadores assim são também conhecidos como **superescalares**
  - Em contraste com a arquitetura **escalar** vista até agora, que processa um único dado por vez

# Expedição Múltipla Dinâmica

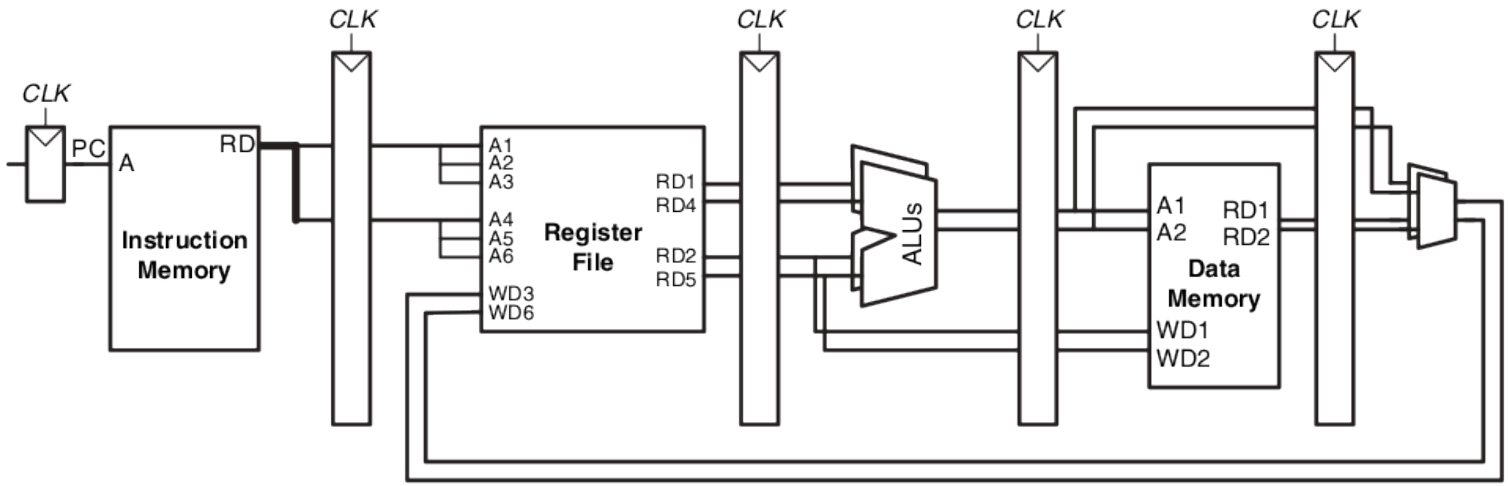
## Superescalar

- Processadores assim são também conhecidos como **superescalares**
  - Em contraste com a arquitetura **escalar** vista até agora, que processa um único dado por vez
  - Superescalar é então uma técnica de *pipeline* que permite que processadores executem mais de uma instrução por ciclo de *clock*, selecionando cada instrução durante a execução

# Expedição Múltipla Dinâmica

## Superescalar

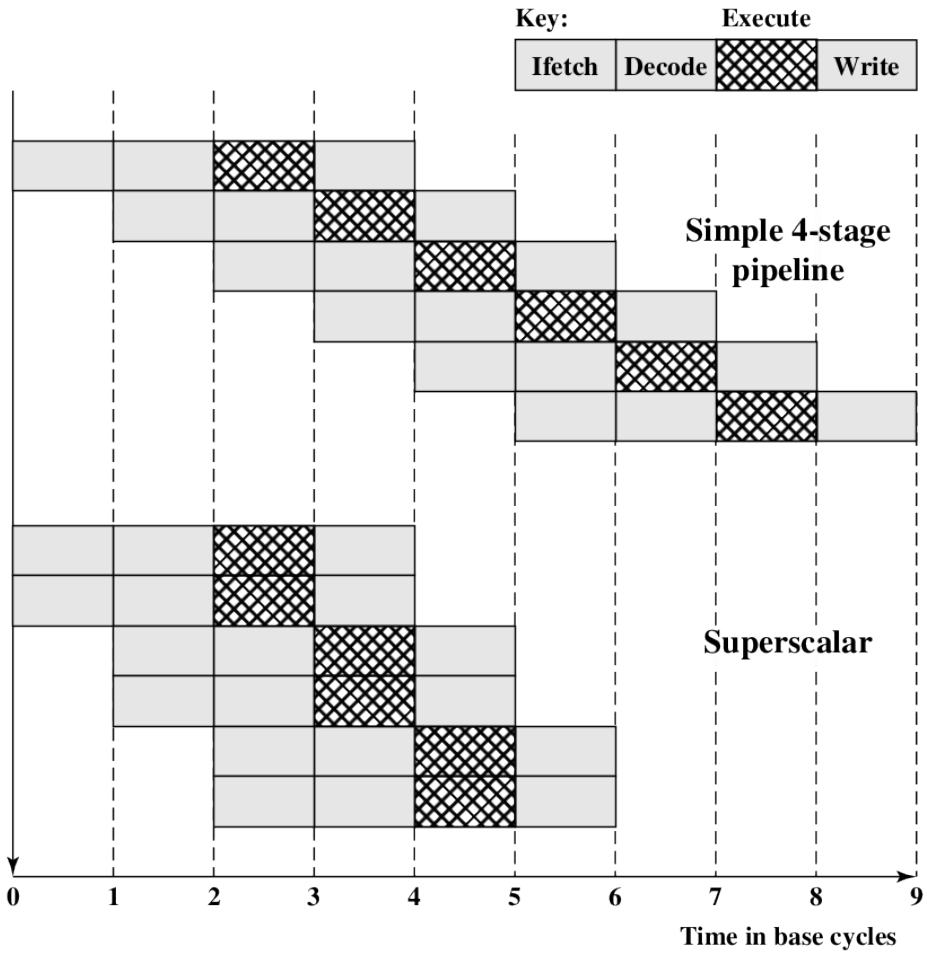
- Possuem múltiplas cópias do hardware para executar múltiplas instruções simultaneamente
  - Bastante semelhante à expedição estática




Fonte: [3]

# Expedição Múltipla Dinâmica

## Superescalar

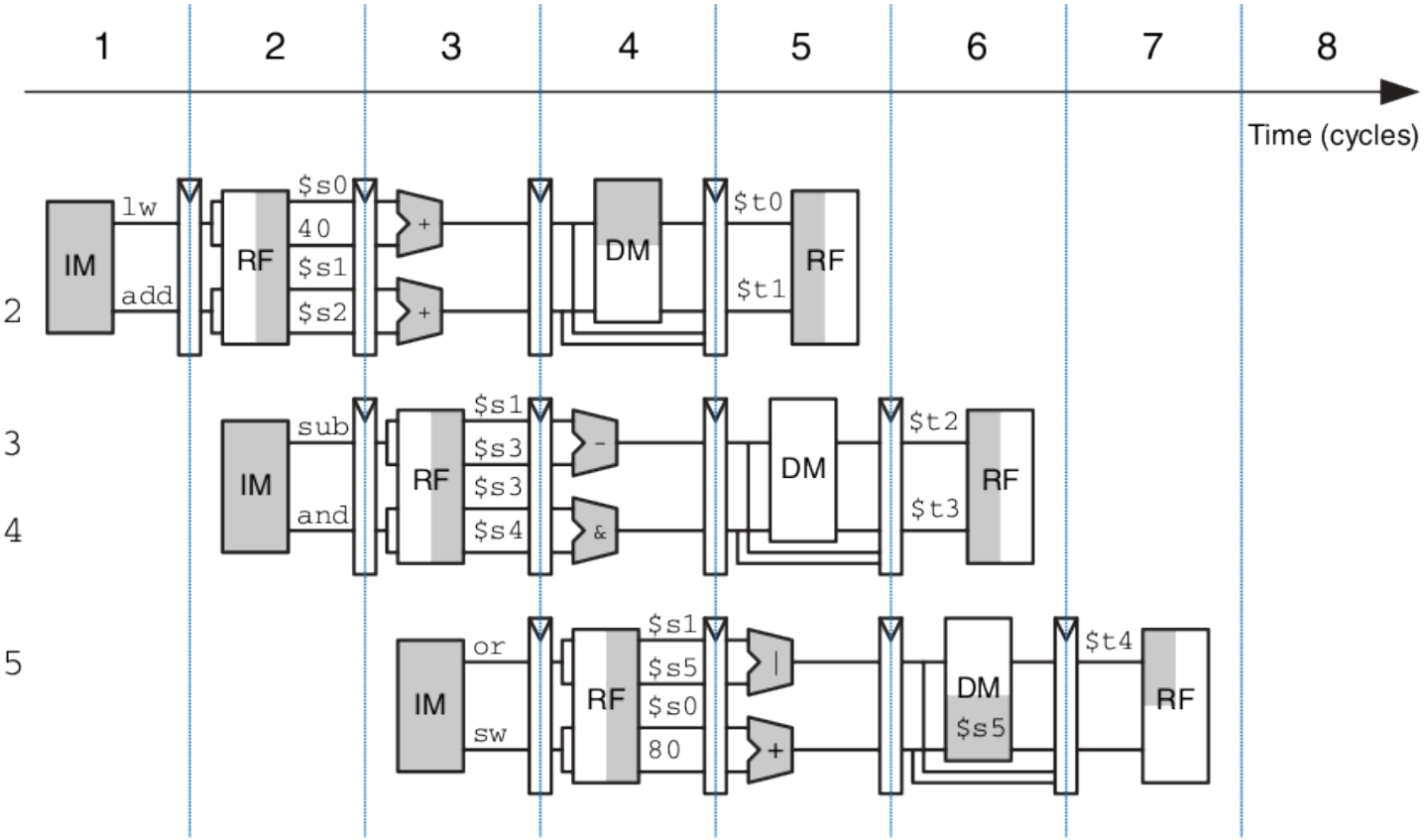



Fonte: Adaptado de [2]

# Expedição Múltipla Dinâmica

## Superescalar: Exemplo

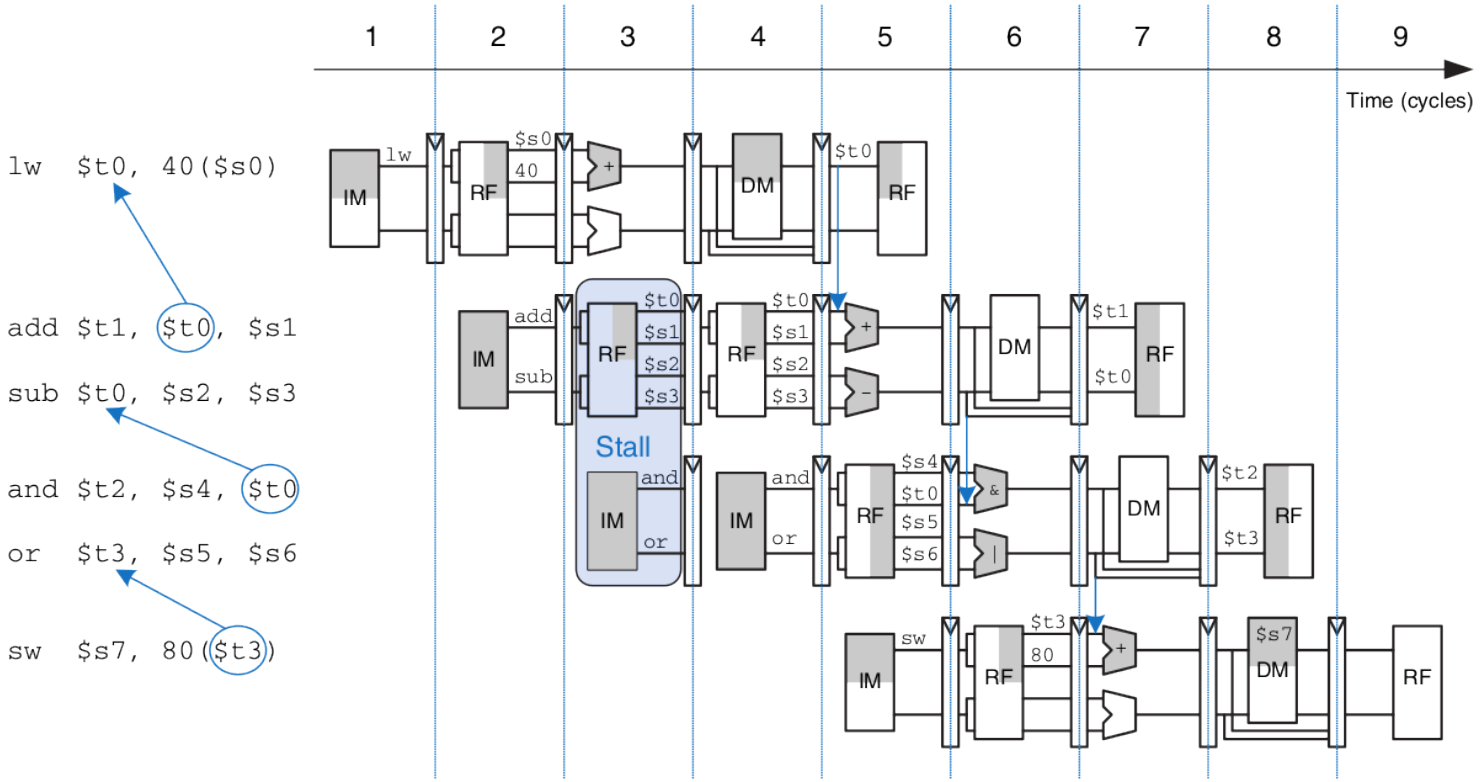
```
lw   $t0, 40($s0)
add  $t1, $s1, $s2

sub  $t2, $s1, $s3
and  $t3, $s3, $s4

or   $t4, $s1, $s5
sw   $s5, 80($s0)
```




Fonte: [3]

# Expedição Múltipla Dinâmica

## Superescalar: Conflitos

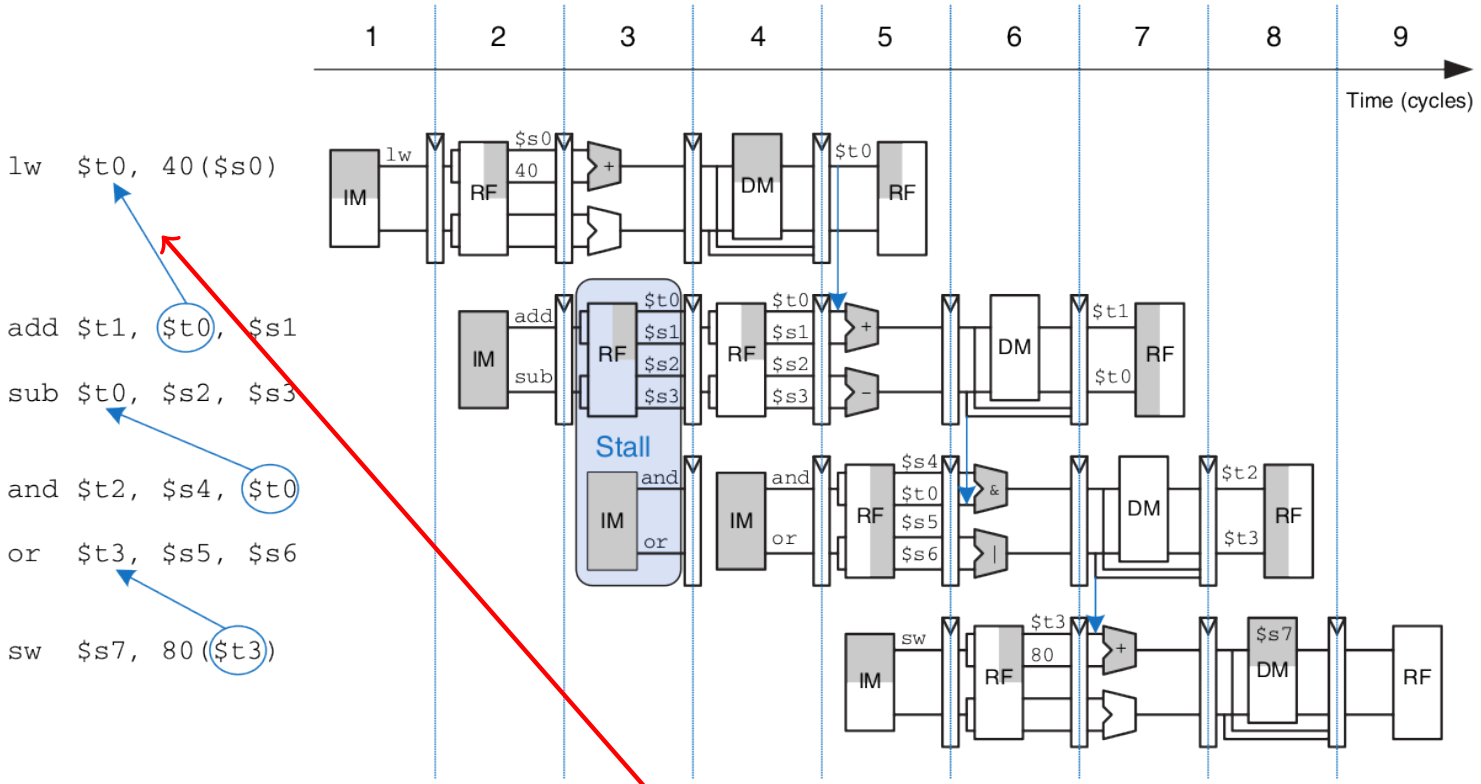



Fonte: [3]

Naturalmente, conflitos ocorrem

# Expedição Múltipla Dinâmica

## Superescalar: Conflitos

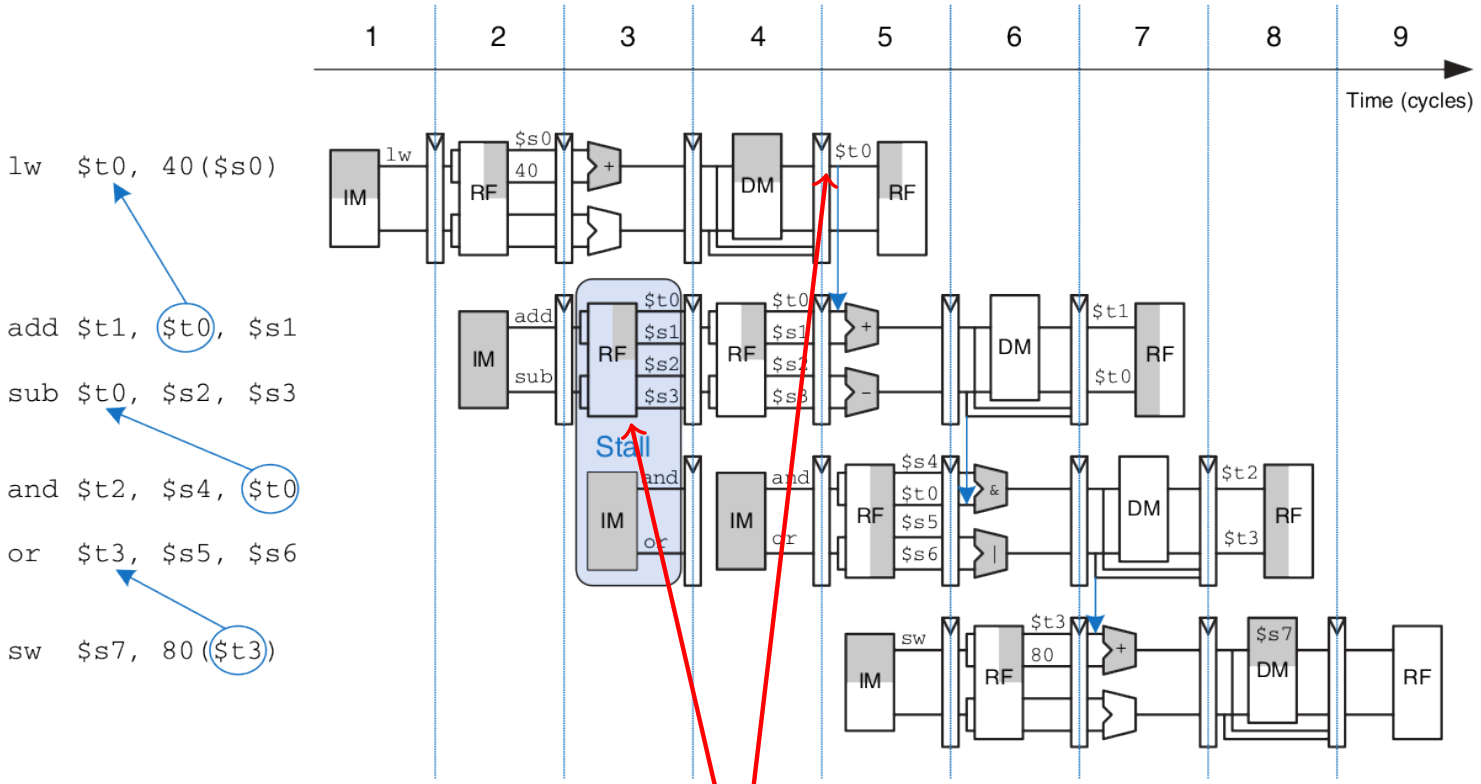



Fonte: [3]

A dependência entre `add` e `lw` não só impede sua expedição conjunta

# Expedição Múltipla Dinâmica

## Superescalar: Conflitos

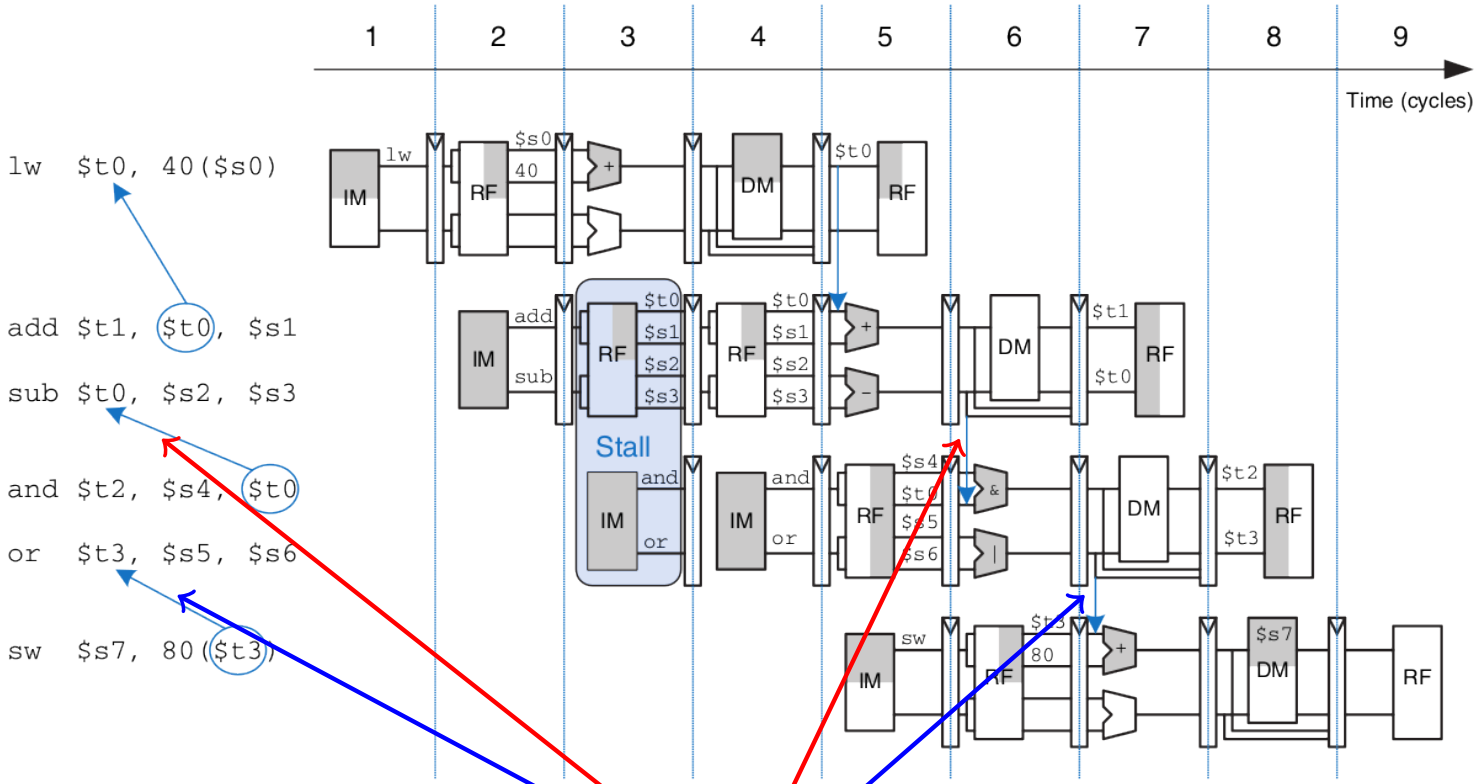



Fonte: [3]

Como para a *pipeline* por mais um ciclo, até `lw` poder fazer um *forward* de `$t0` a `add` no 5º ciclo

# Expedição Múltipla Dinâmica

## Superescalar: Conflitos




Fonte: [3]

As dependências entre `sub` e `and` e entre `or` e `sw` são resolvidas via *forwarding*

# Expedição Múltipla Dinâmica

## Superescalar

- Tipicamente buscam múltiplas instruções, tentando determinar dependências entre elas
  - Para que instruções independentes possam rodar em paralelo
  - Ainda assim precisam da remoção de dependência, pelo compilador, para obter um bom desempenho

# Expedição Múltipla Dinâmica

## Superescalar

- Tipicamente buscam múltiplas instruções, tentando determinar dependências entre elas
  - Para que instruções independentes possam rodar em paralelo
  - Ainda assim precisam da remoção de dependência, pelo compilador, para obter um bom desempenho
- Fazem uso do Escalonamento Dinâmico de Pipeline
  - *Dynamic pipeline scheduling*
  - Consiste de escolher que instruções executar em um dado ciclo de *clock*, buscando evitar conflitos e *stalls*

# Superescalar

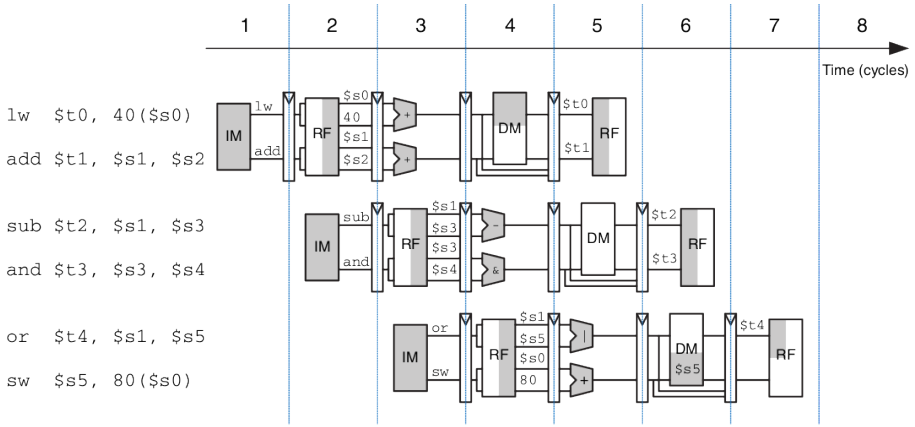
## Escalonamento Dinâmico de *Pipeline*

- Naturalmente, a ordem de execução das instruções não necessariamente bate com a ordem original

# Superescalar

## Escalonamento Dinâmico de *Pipeline*

- Naturalmente, a ordem de execução das instruções não necessariamente bate com a ordem original
- Em nosso exemplo, o processador podia rodar até 2 instruções por ciclo
  - Vindas de qualquer parte do código, desde que fossem observadas as dependências




Fonte: [3]

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

- Considere a sequência

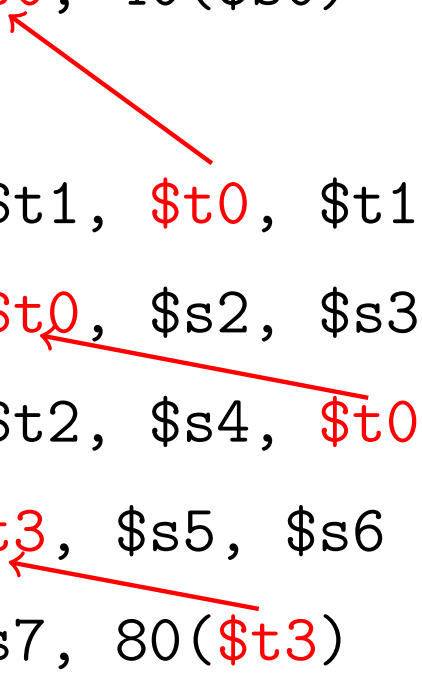
```
lw $t0, 40($s0)
nop
add $t1, $t0, $t1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

- Considere a sequência

```
lw $t0, 40($s0)
nop
add $t1, $t0, $t1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

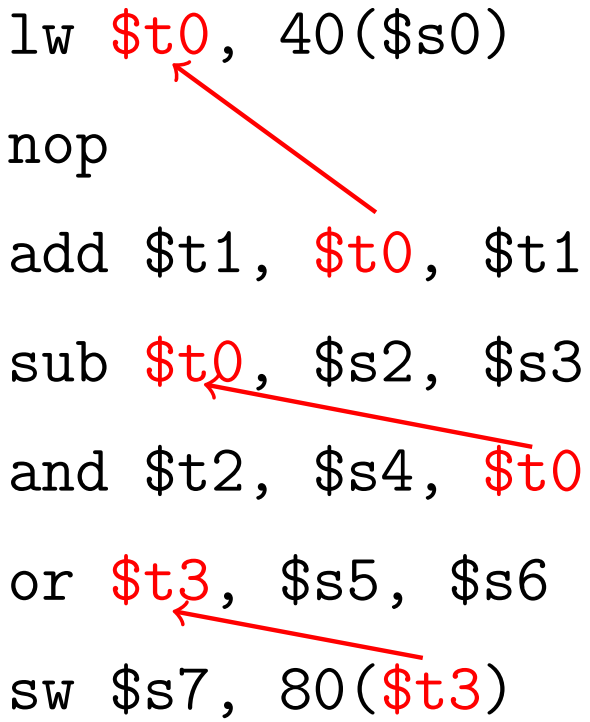


Há dependências que são velhas conhecidas nossas...

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

- Considere a sequência




Estas são conhecidas como RAW (*Read After Write*), porque tentamos ler algo que depende de escrita que pode não ter finalizado

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

- Considere a sequência

```
lw $t0, 40($s0)
nop
add $t1, $t0, $t1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

E há outras introduzidas pelo paralelismo superescalar, que impedem 2 instruções de serem expedidas em conjunto

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

- Considere a sequência

```
lw $t0, 40($s0)
nop
add $t1, $t0, $t1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

Como as WAR (*Write After Read*, ou **antidependência**), quando modificamos um valor sem conseguir garantir que leituras anteriores foram corretas

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

- Considere a sequência

```
lw $t0, 40($s0)
nop
add $t1, $t0, $t1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

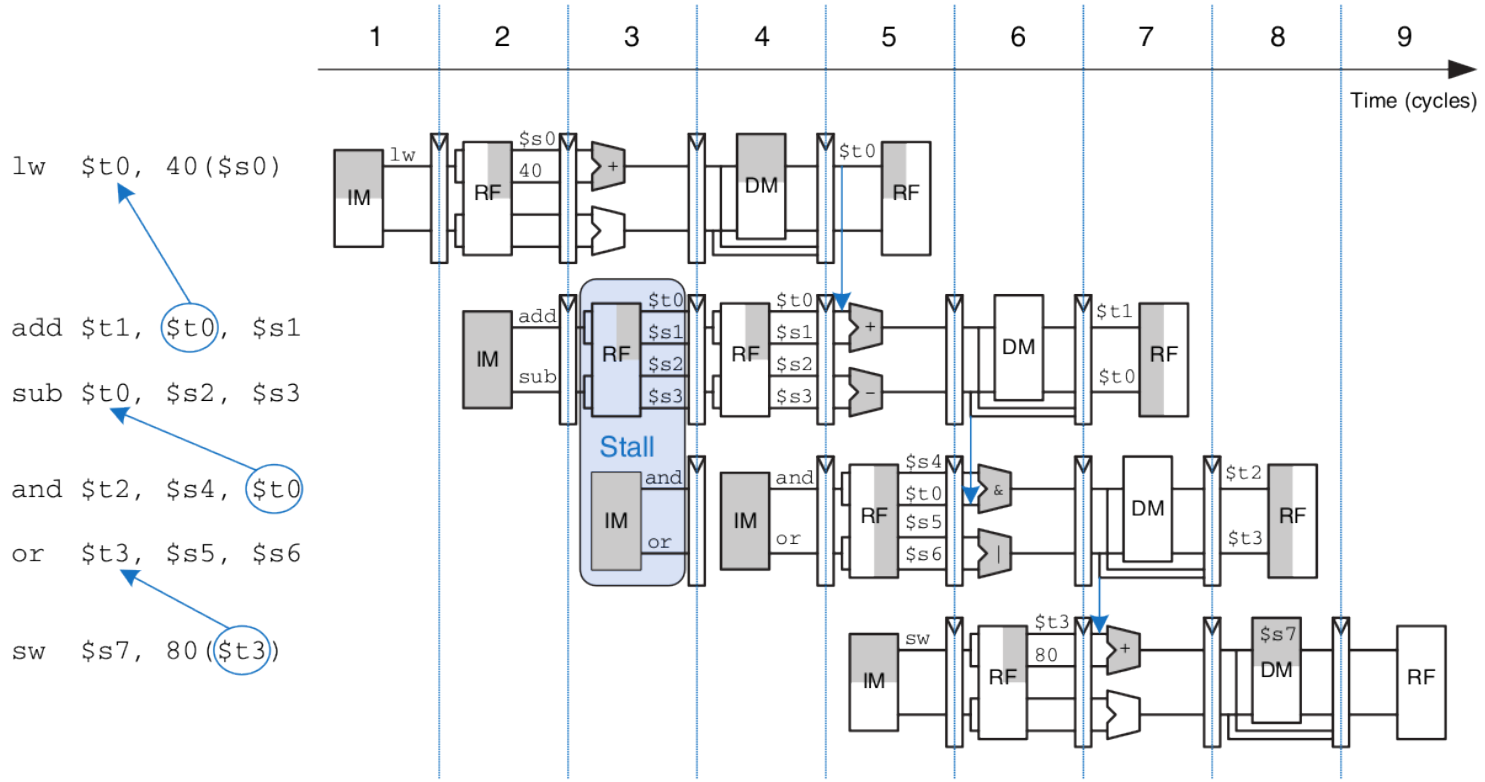
Ou as WAW (*Write After Write*), onde 2 instruções modificam um mesmo valor sem garantir que sua ordem seja preservada. Ex:

```
add $t0, $s1, $s2
sub $t0, $s3, $s4
```

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

Rodado nessa ordem, o código se comportaria como já visto:

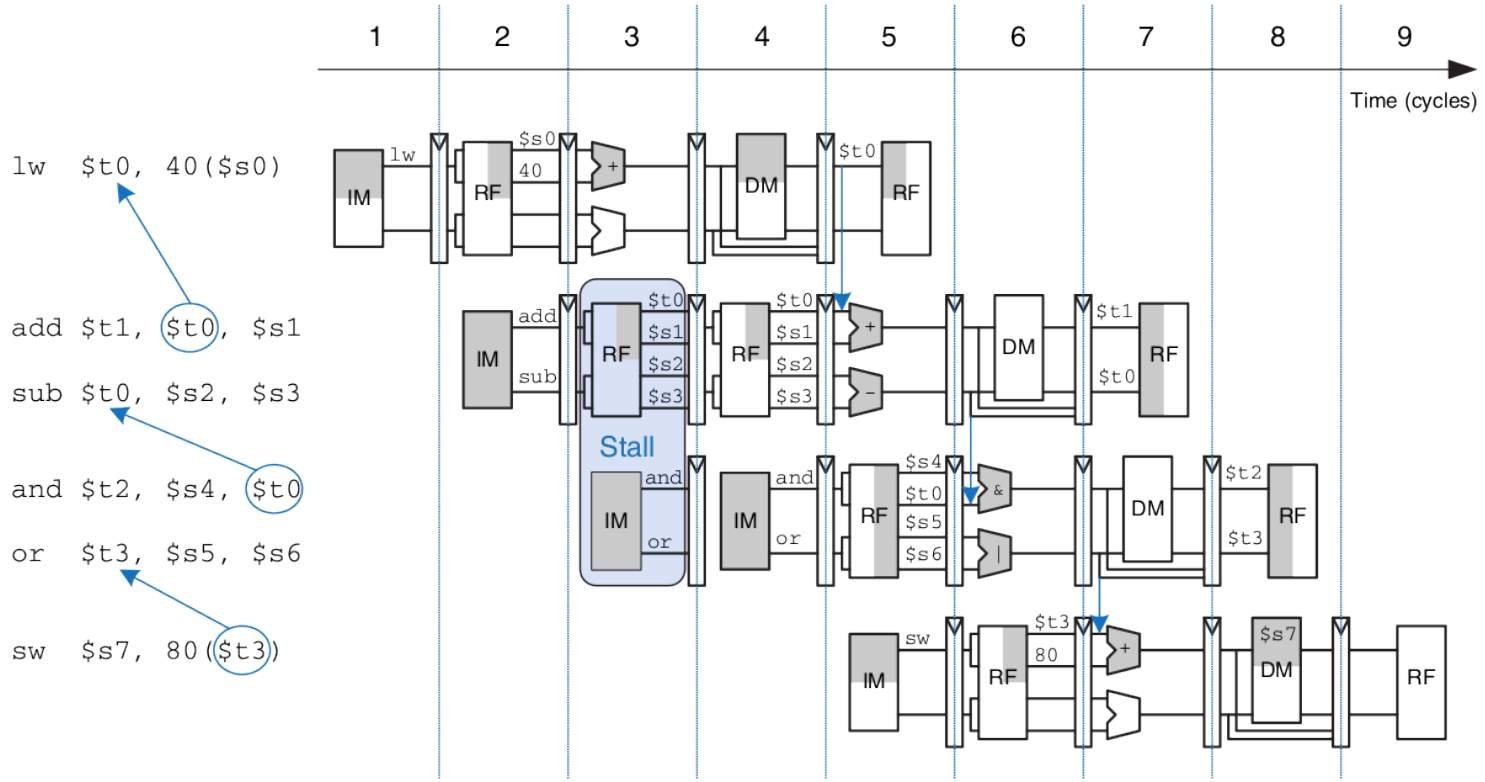



Fonte: [3]

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

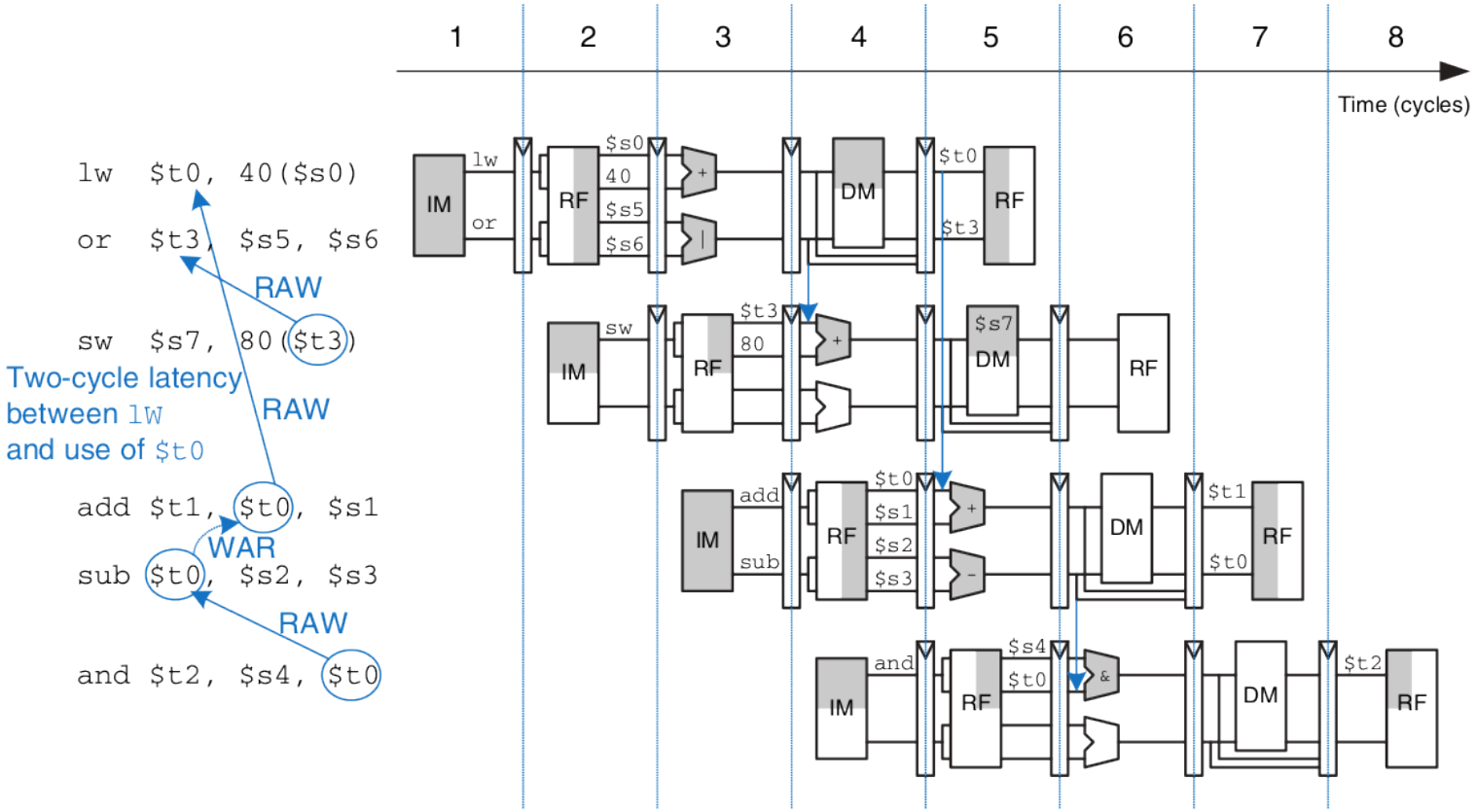
E se o processador conseguisse definir a ordem dinamicamente?




Fonte: [3]

# Superescalar

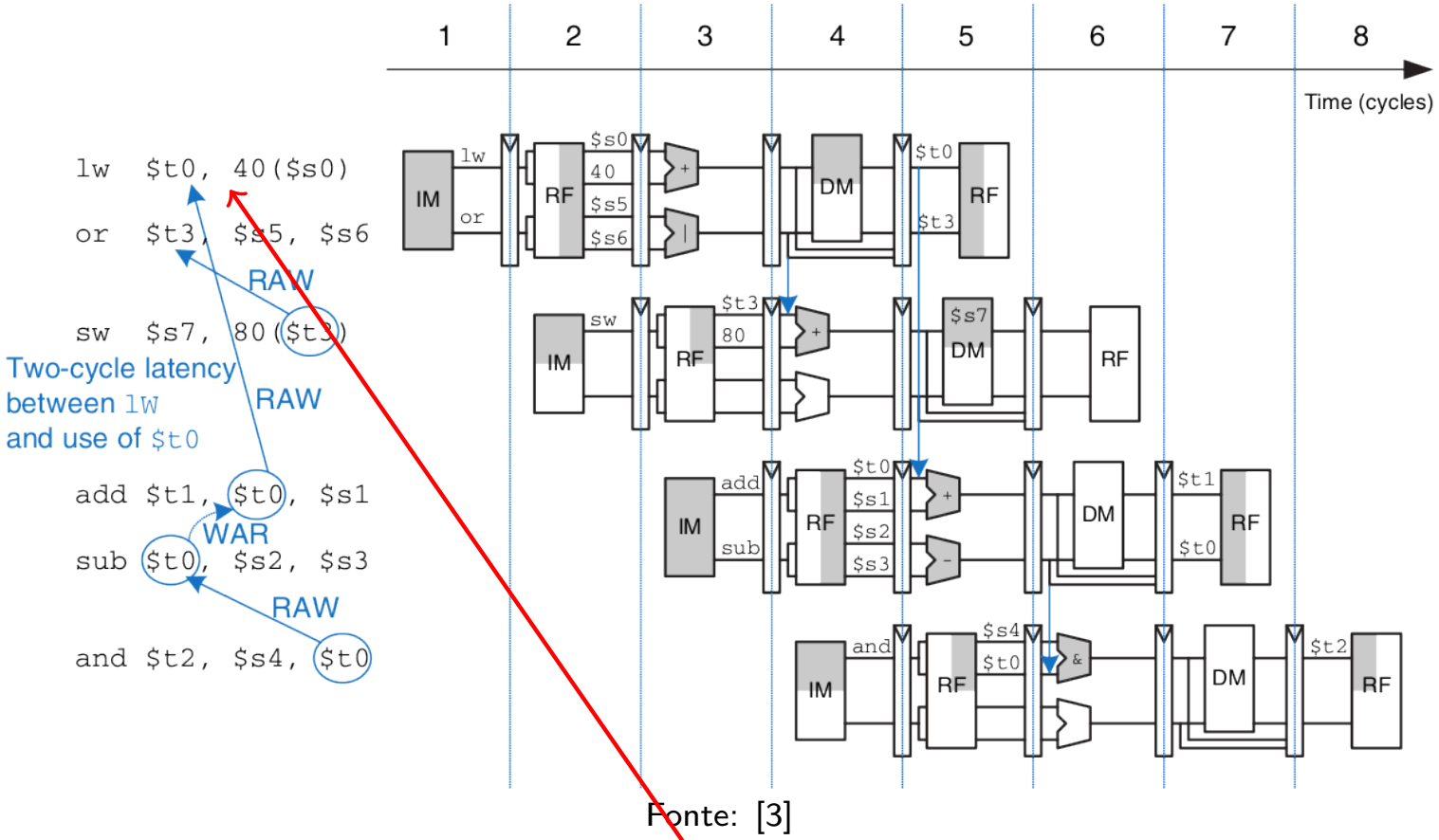
## Escalonamento Dinâmico de *Pipeline*: Exemplo




Fonte: [3]

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

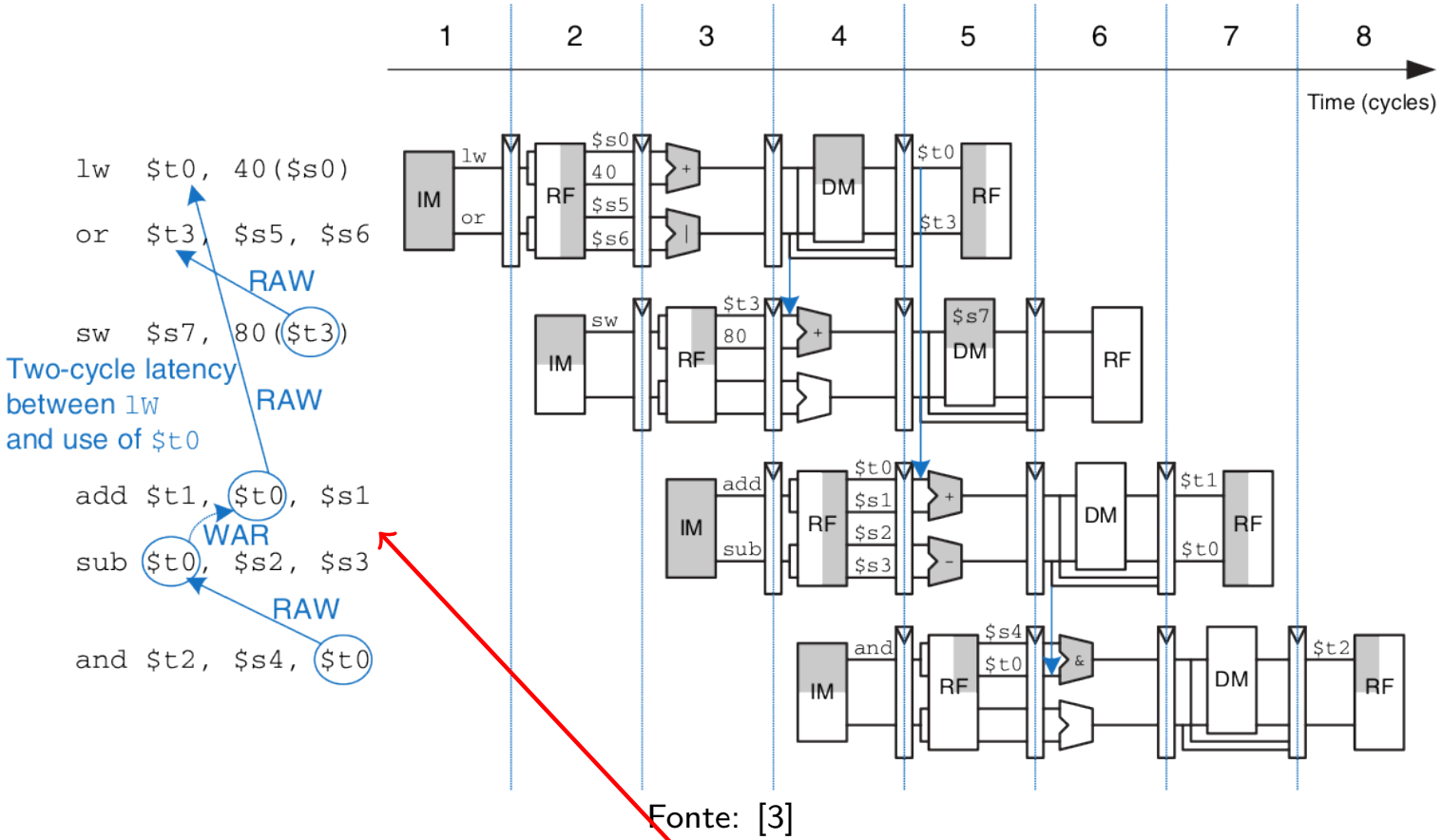



Fonte: [3]

**Ciclo 1**: `lw` é escolhido para rodar.

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

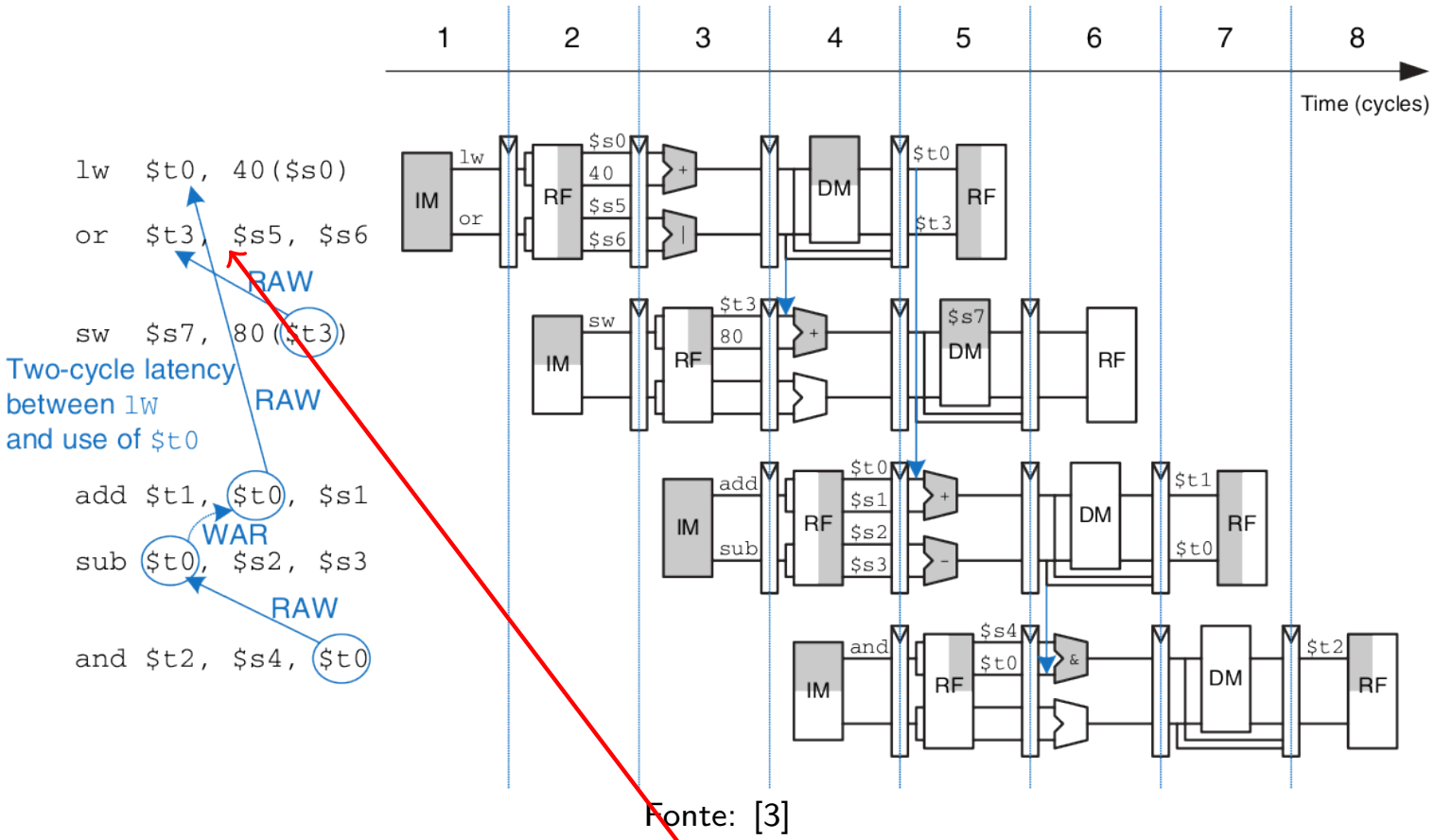



Fonte: [3]

Como `add` e `sub` dependem de `lw`, elas não podem rodar

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

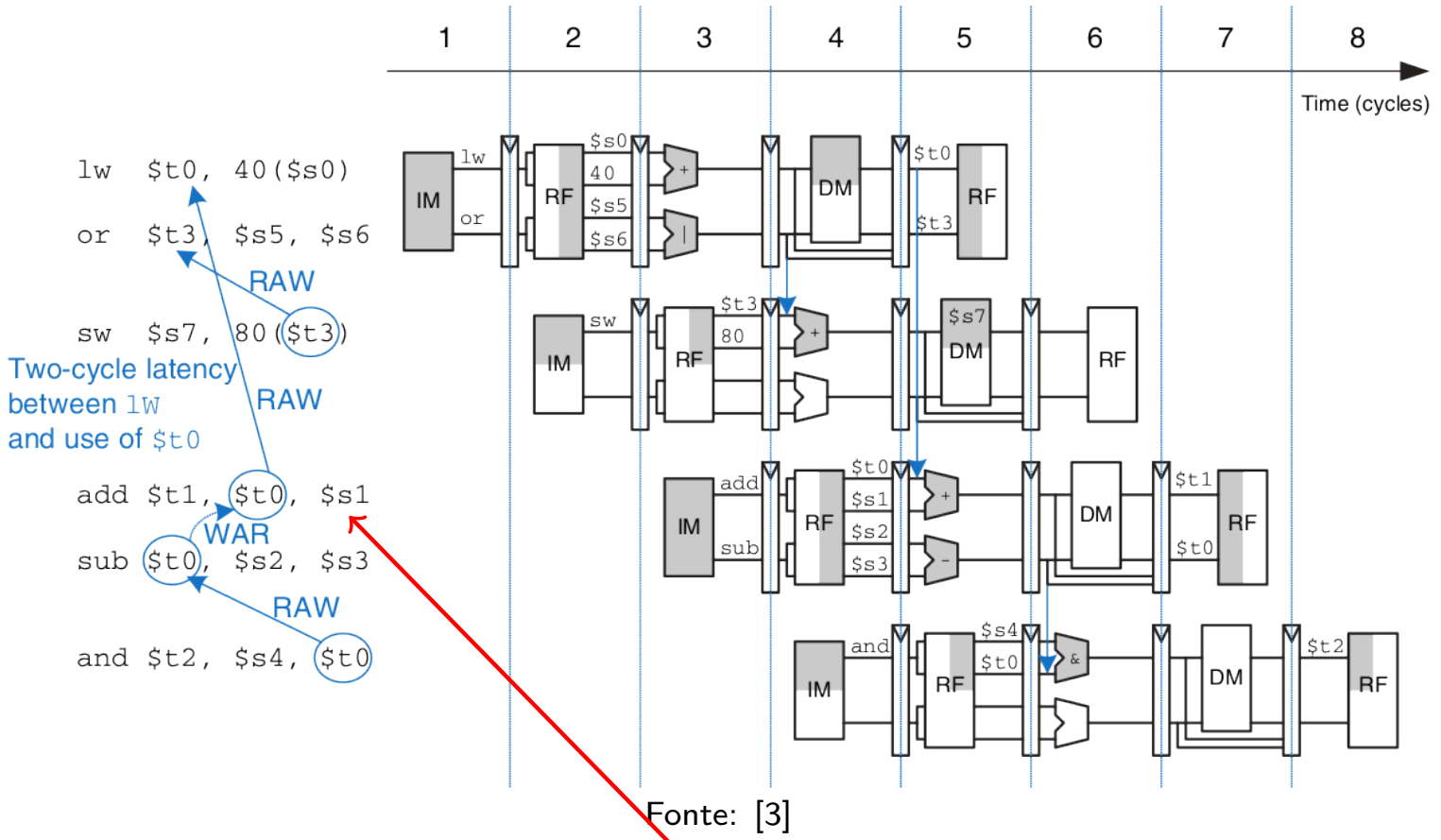



Fonte: [3]

Por ser independente dessas, `or` é escolhida para rodar com `lw`

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

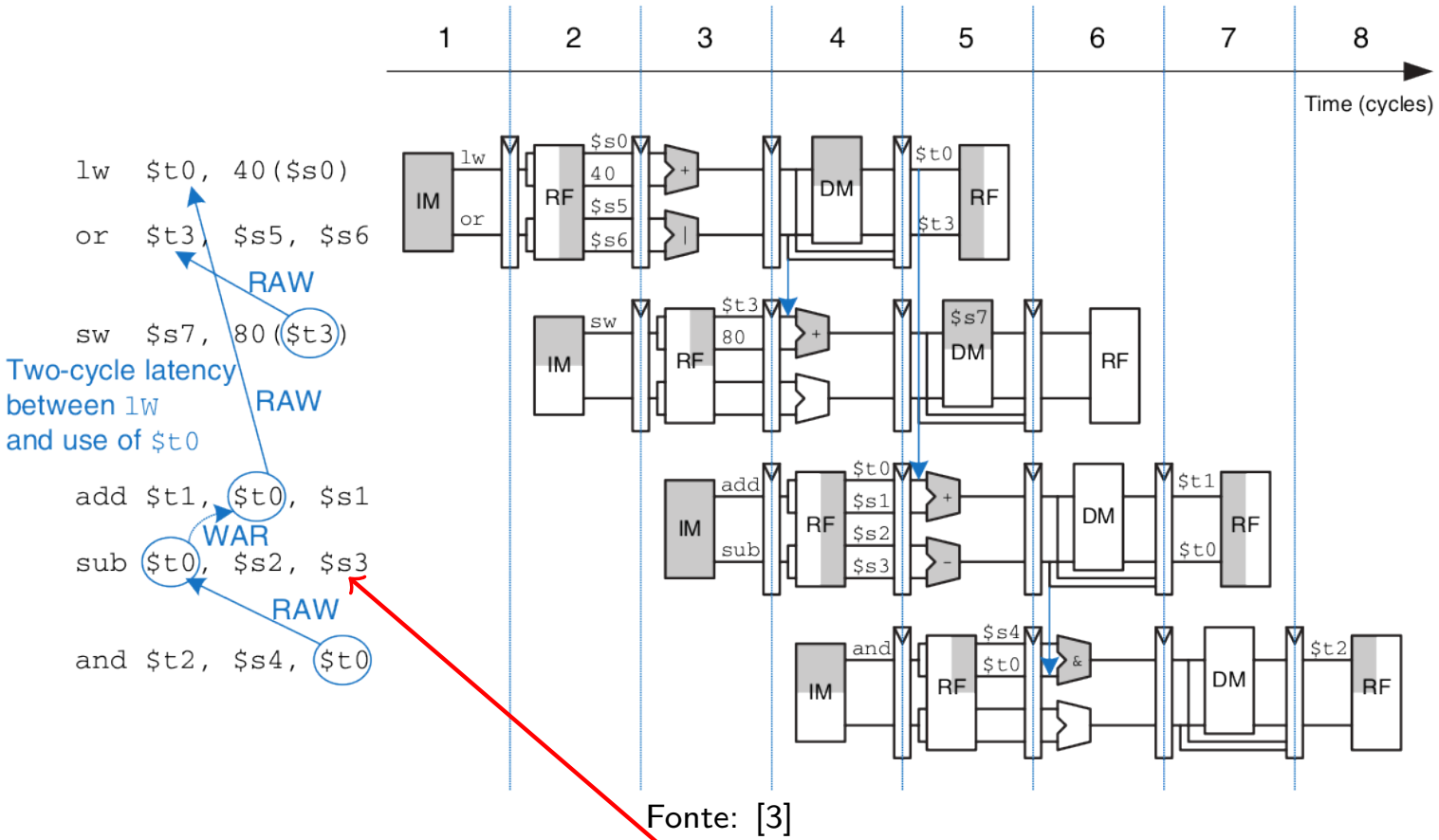



Fonte: [3]

**Ciclo 2**: Há uma latência de 2 ciclos para `lw`, então add tem que esperar

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

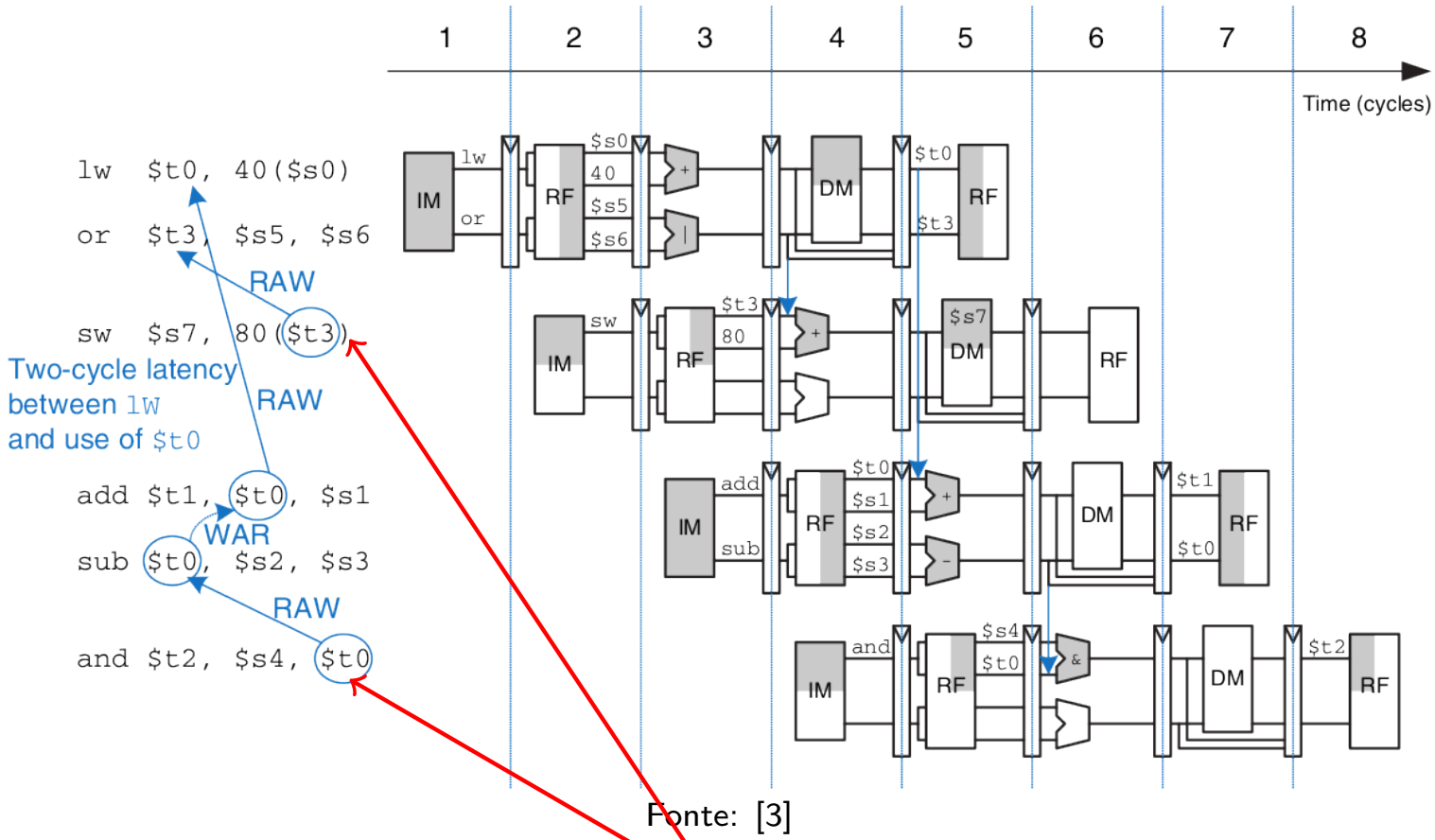



Fonte: [3]

Por escrever em `$t0`, `sub` não pode ir antes de `add` (WAR)

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

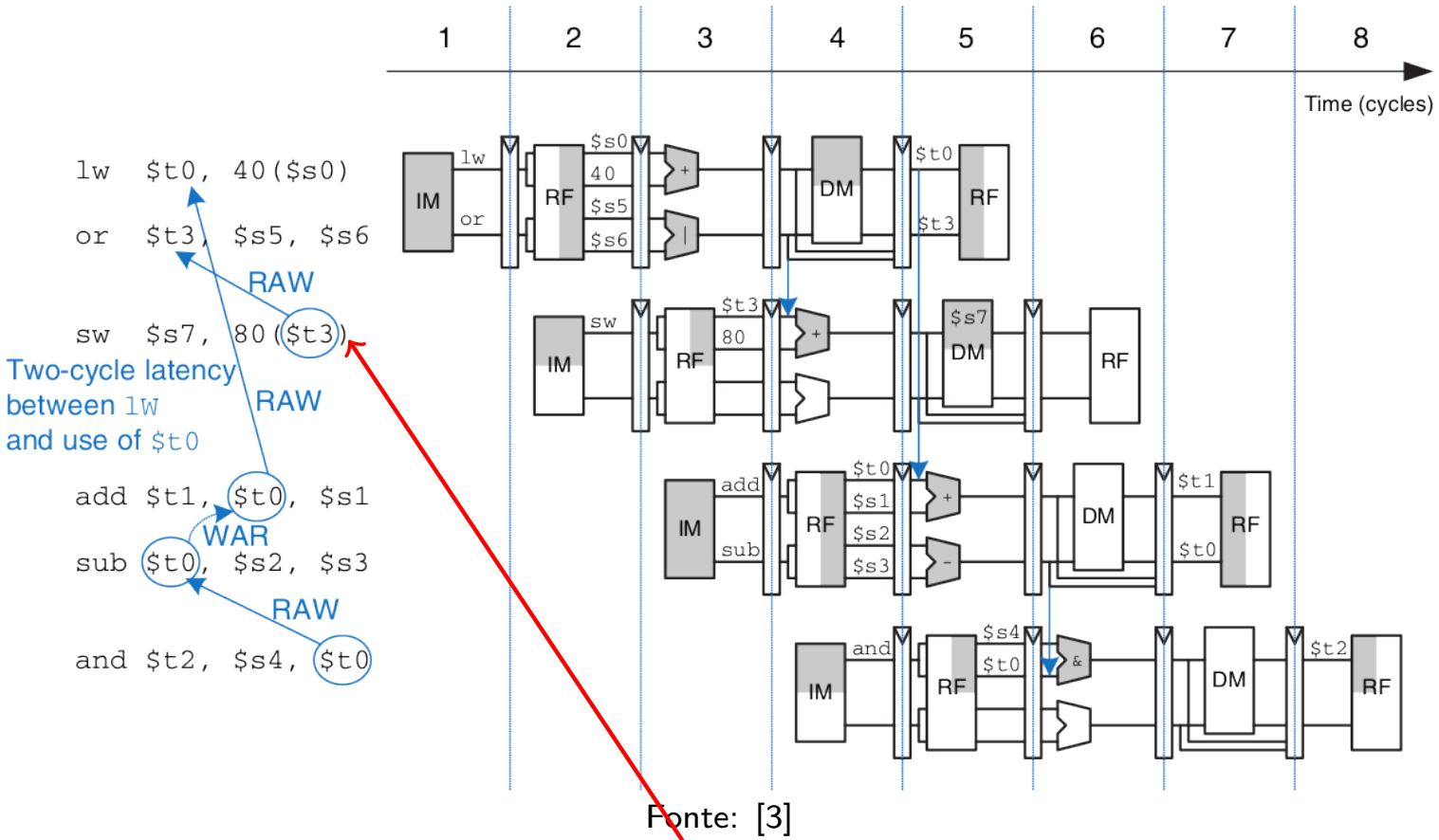



Fonte: [3]

Como and depende de sub, sobrou apenas sw para ser escolhida

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

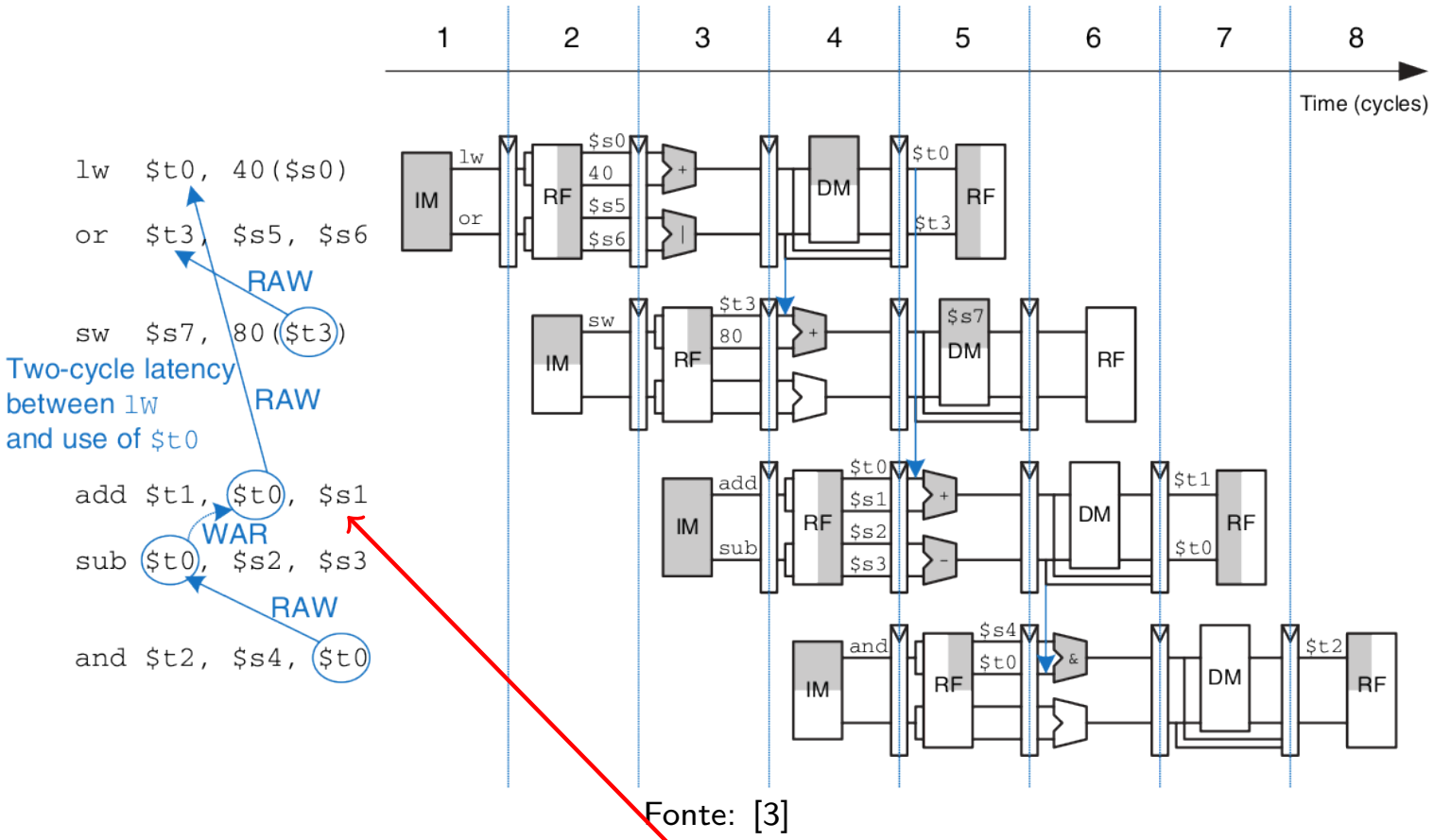



Fonte: [3]

E esta roda sozinha...

# Superescalar

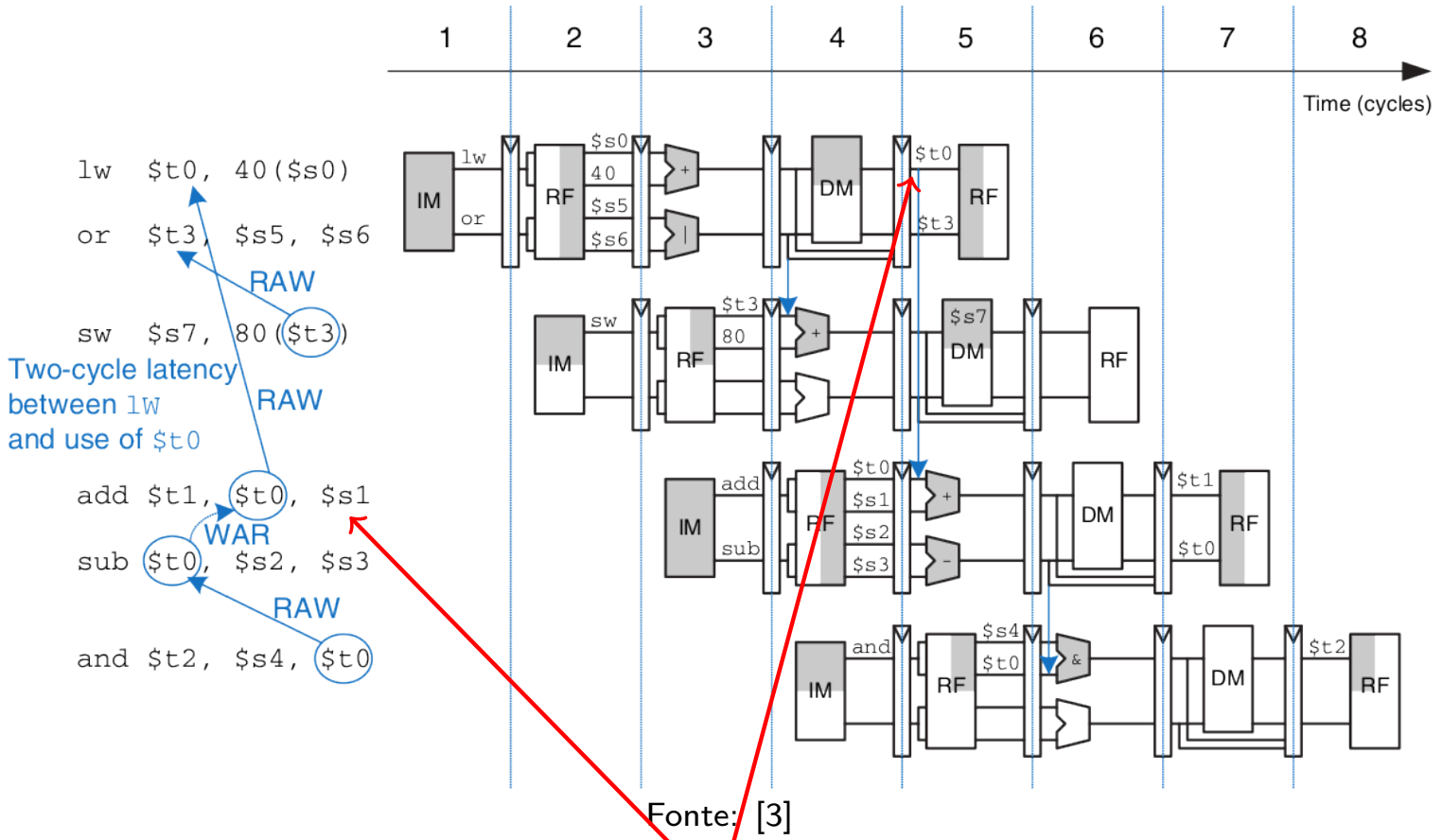
## Escalonamento Dinâmico de *Pipeline*: Exemplo




Fonte: [3]

**Ciclo 3**: `$t0` estará disponível no ciclo seguinte, então add pode rodar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

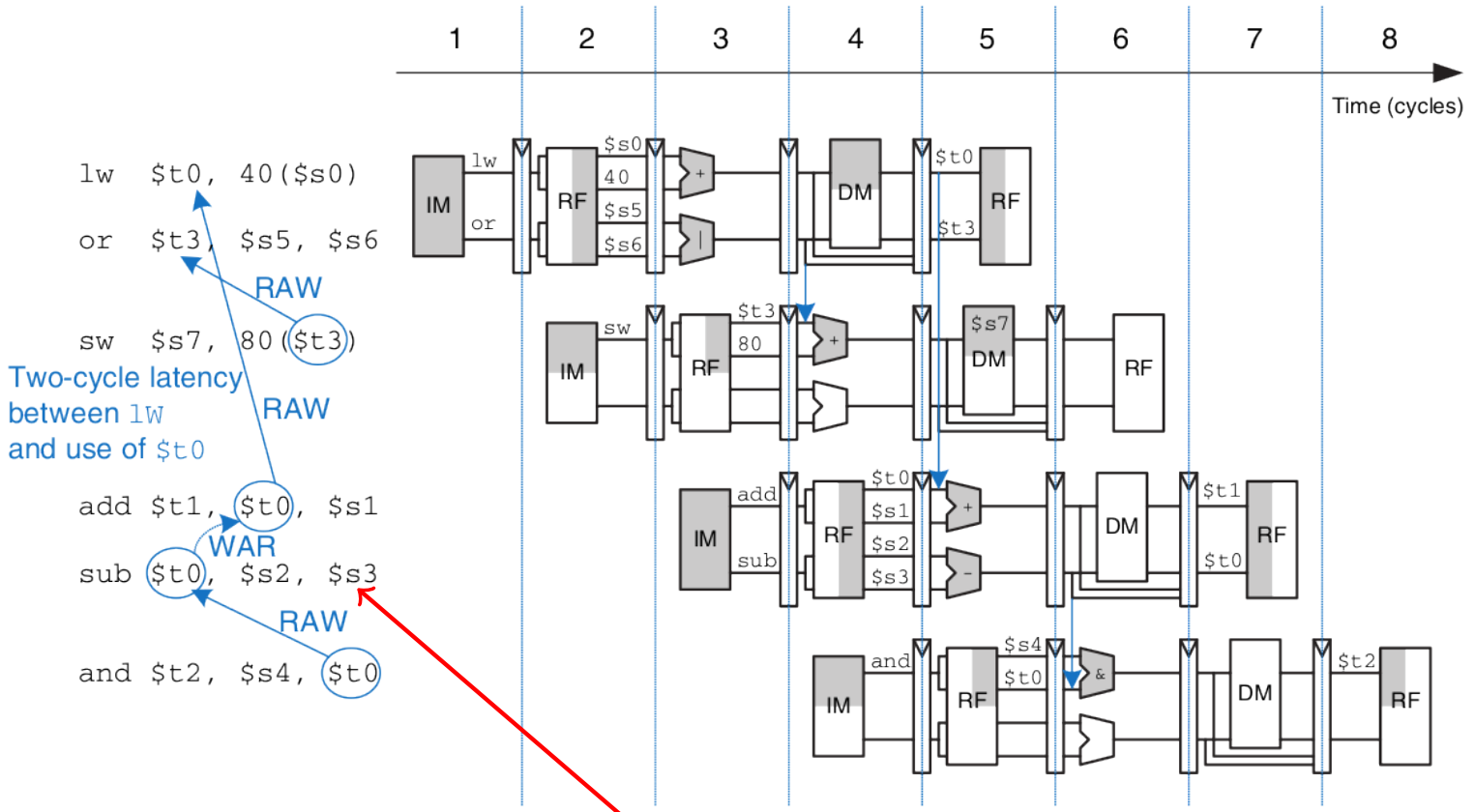



Fonte: [3]

Naturalmente, seu valor será passado via *forwarding*

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

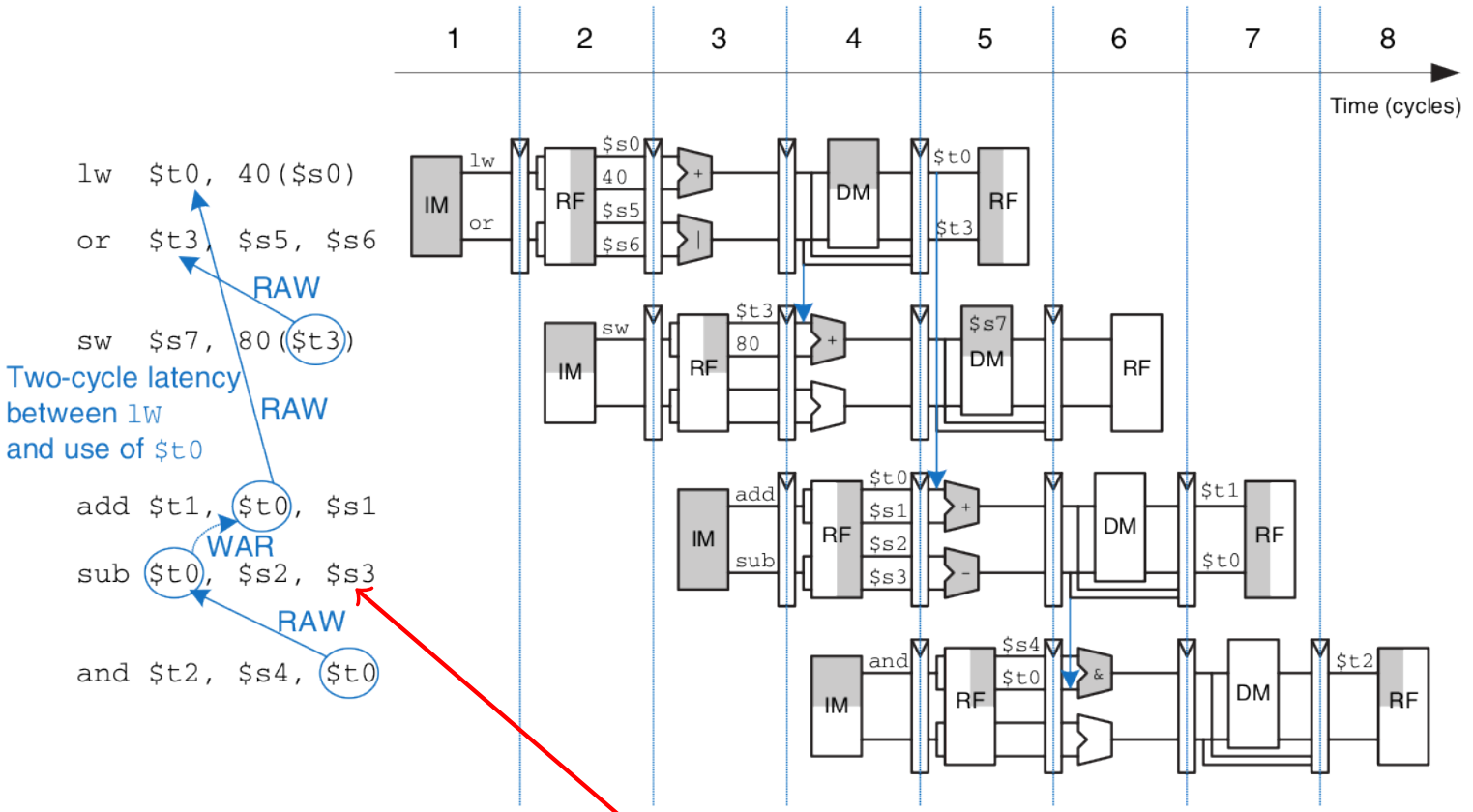



Fonte: [3]

sub também pode rodar, pois escreverá $t0 somente após add lê-lo

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

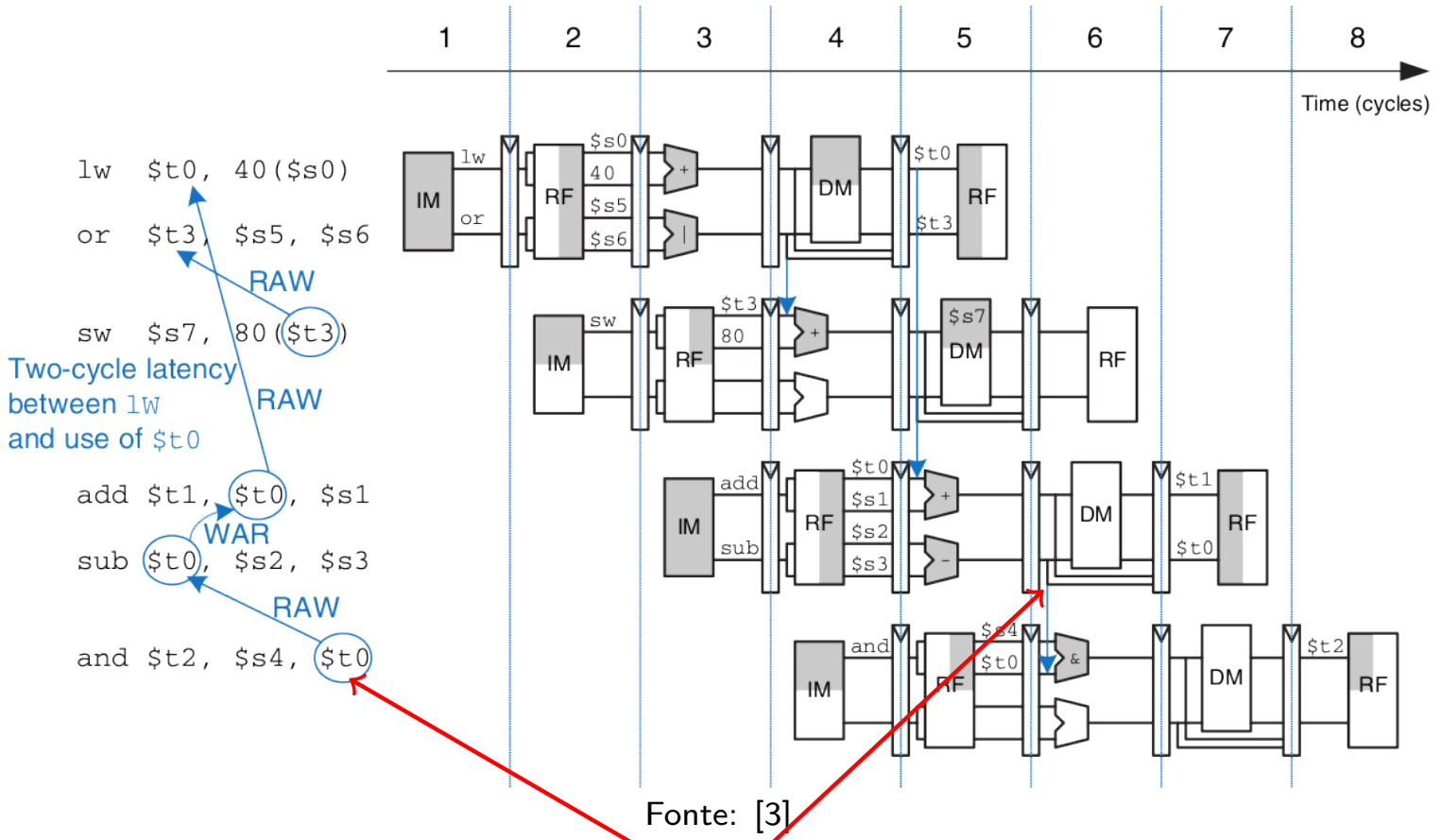



Fonte: [3]

Mas isso somente se fizermos o arquivo de registradores agir assim

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo

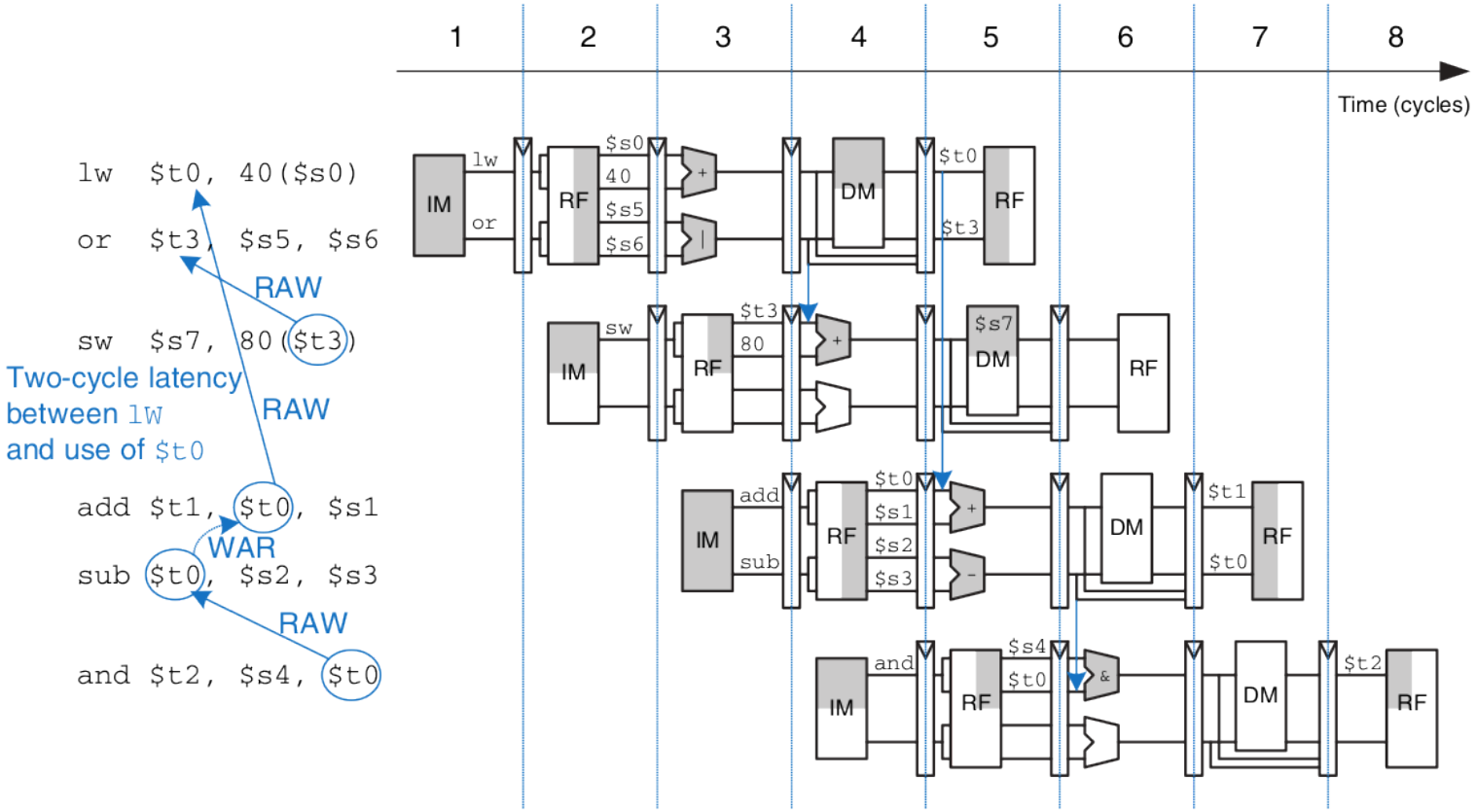



Fonte: [3]

`and` pode finalmente rodar, pois `$t0` é enviado a ela via `forwarding`

# Superescalar

## Escalonamento Dinâmico de *Pipeline*: Exemplo



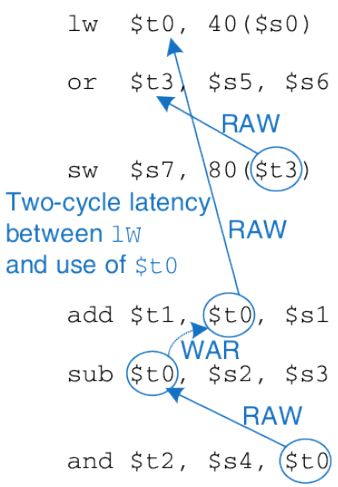

Fonte: [3]

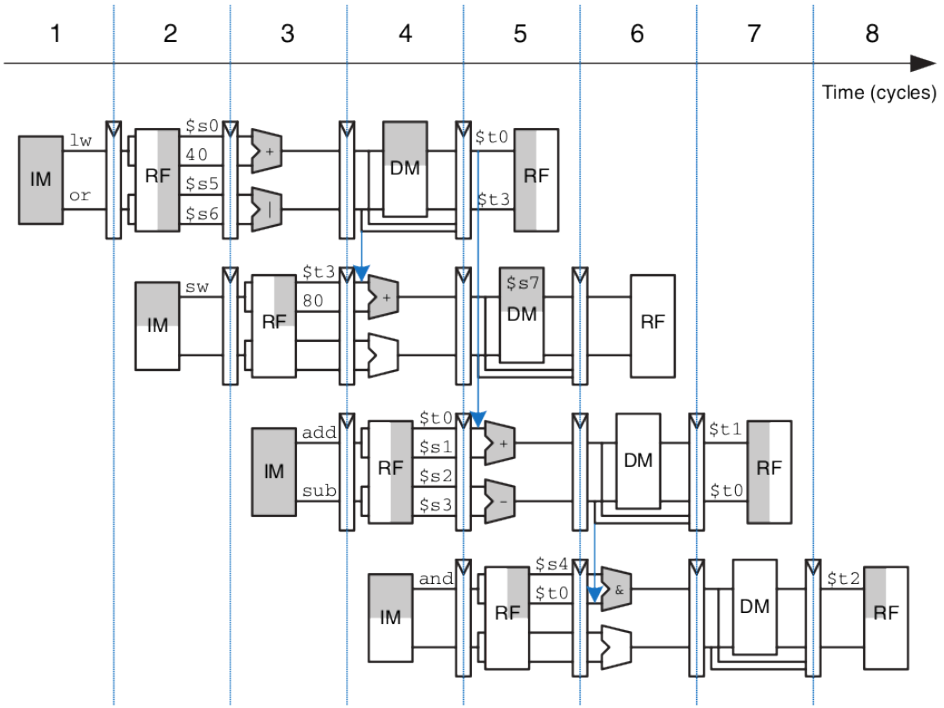
São então iniciadas 6 instruções em 4 ciclos → um IPC de 1.5

# Superescalar

## Escalonamento Dinâmico de *Pipeline*

- O processador preserva então o fluxo dos dados, em vez da ordem original no programa
  - Modelo conhecido como **execução fora de ordem**




Fonte: [3]

# Superescalar

## Escalonamento Dinâmico de *Pipeline*

- Como vimos, execução fora de ordem faz surgir novos problemas de dependência
  - Notoriamente WAR
  - Oriundos do fato de não conseguirmos determinar o valor dos registradores olhando apenas para a sequência de instruções

# Superescalar

## Escalonamento Dinâmico de *Pipeline*

- Como vimos, execução fora de ordem faz surgir novos problemas de dependência
  - Notoriamente WAR
  - Oriundos do fato de não conseguirmos determinar o valor dos registradores olhando apenas para a sequência de instruções
- Como então lidar com esse problema?
  - Renomeação de registradores

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores

- Consiste de adicionar registradores de renomeação ao processador
  - `$r0` a `$r19` em MIPS

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores

- Consiste de adicionar registradores de renomeação ao processador
  - `$r0` a `$r19` em MIPS
- E usá-los para substituir registradores em conflito
  - Estes não podem ser usados diretamente pelo programador
  - Apenas o processador os usa, para eliminar conflitos.

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- Considere novamente a sequência e seu WAR

```
lw $t0, 40($s0)
add $t1, $t0, $t1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- Considere novamente a sequência e seu WAR

```
lw $t0, 40($s0)
add $t1, $t0, $t1
sub $r0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

- O processador poderia, por exemplo, renomear `$t0` para `$r0` em `sub`, eliminando assim o WAR

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- Considere novamente a sequência e seu WAR

```
lw $t0, 40($s0)
add $t1, $t0, $t1
sub $r0, $s2, $s3
and $t2, $s4, $r0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

- O processador poderia, por exemplo, renomear `$t0` para `$r0` em `sub`, eliminando assim o WAR
- Naturalmente, usos posteriores de `$t0` teriam que ser substituídos também

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- Considere novamente a sequência e seu WAR

```
lw $t0, 40($s0)
add $t1, $t0, $t1
sub $r0, $s2, $s3
and $t2, $s4, $r0
or $t3, $s5, $s6
sw $s7, 80($t3)
```

- Para isso, o processador mantém uma tabela com os registradores que foram renomeados
- De modo a renomear de modo consistente os registradores em instruções subsequentes

# Superescalar: Escalonamento Dinâmico

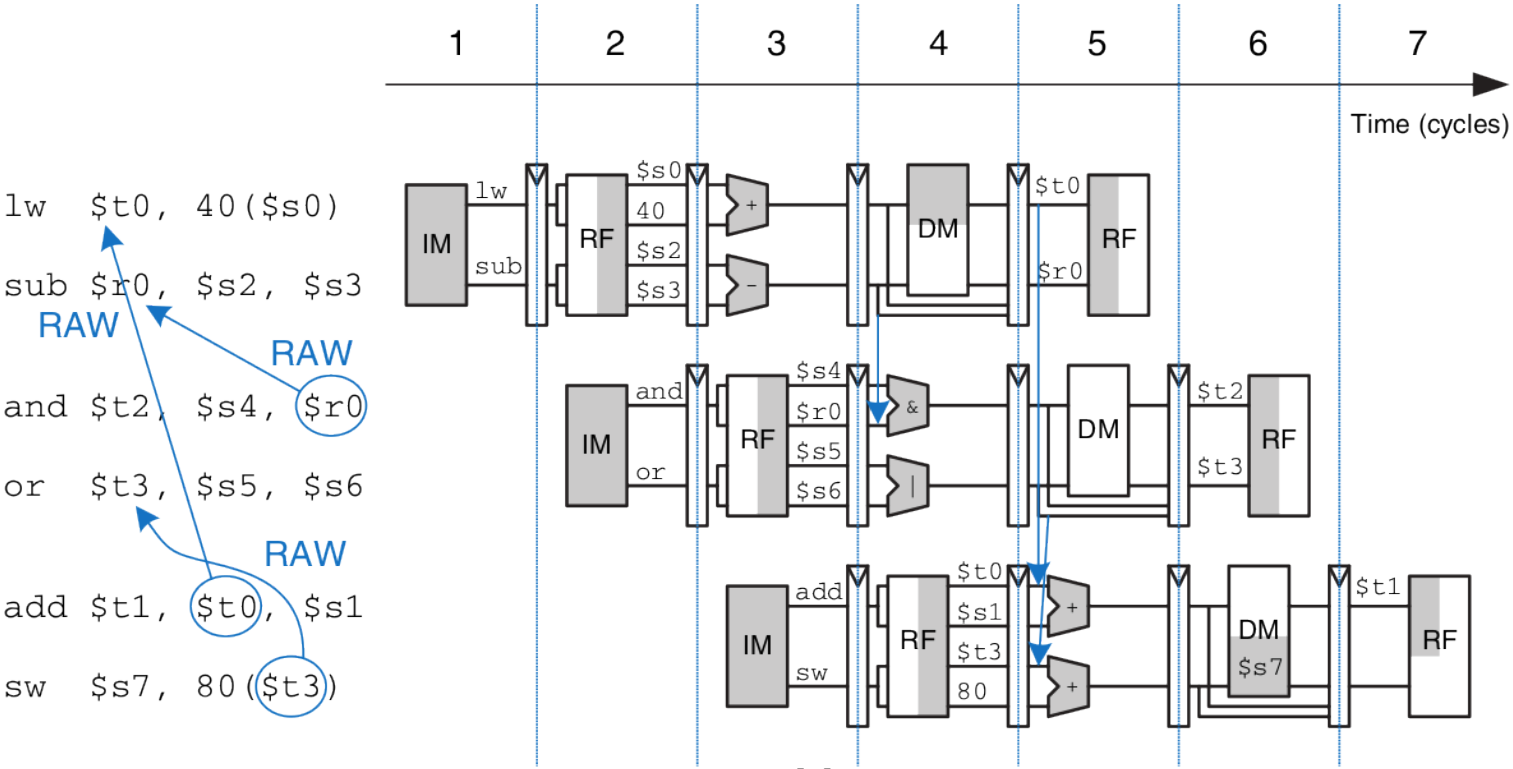
## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?

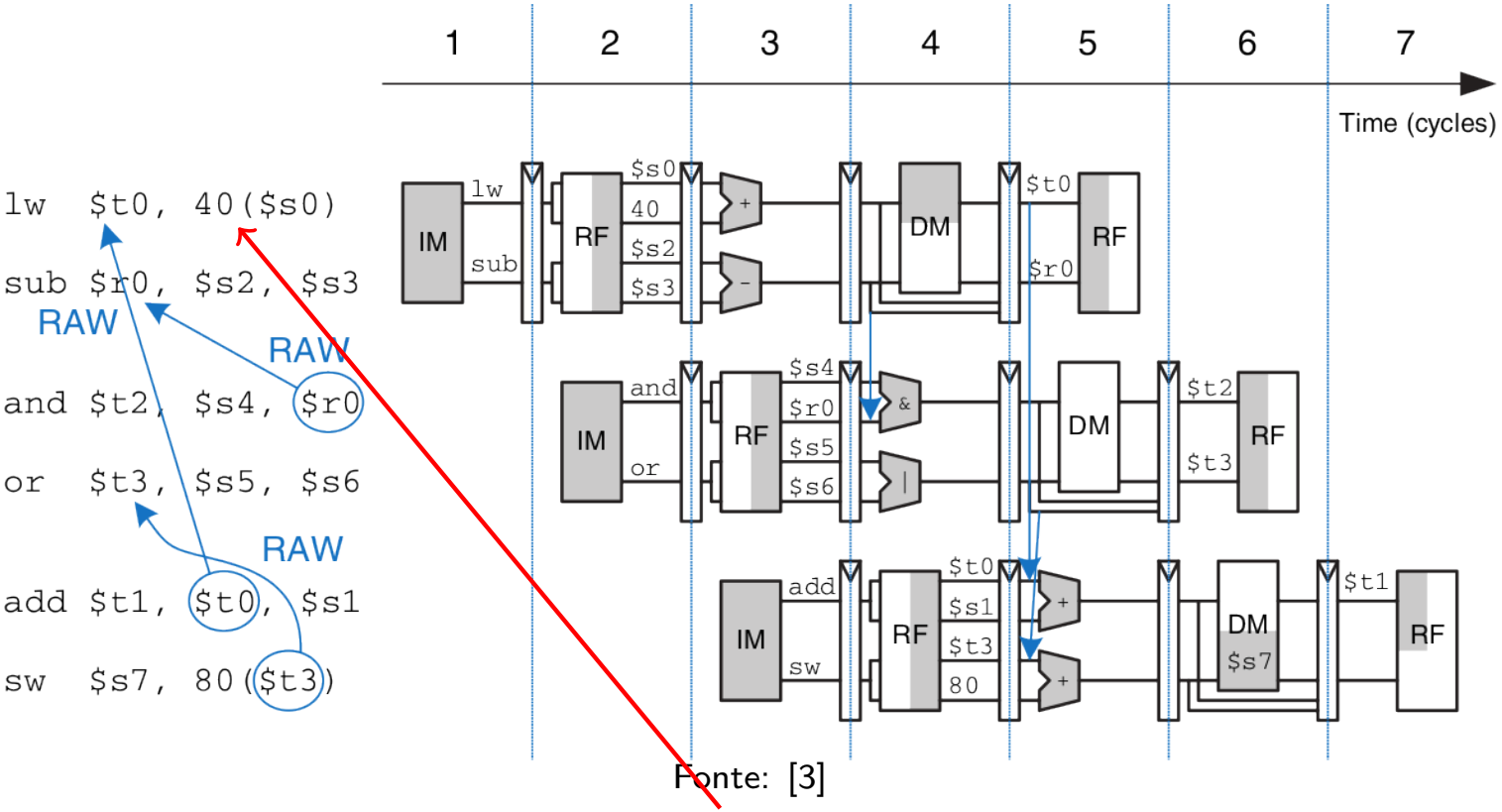



Fonte: [3]

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



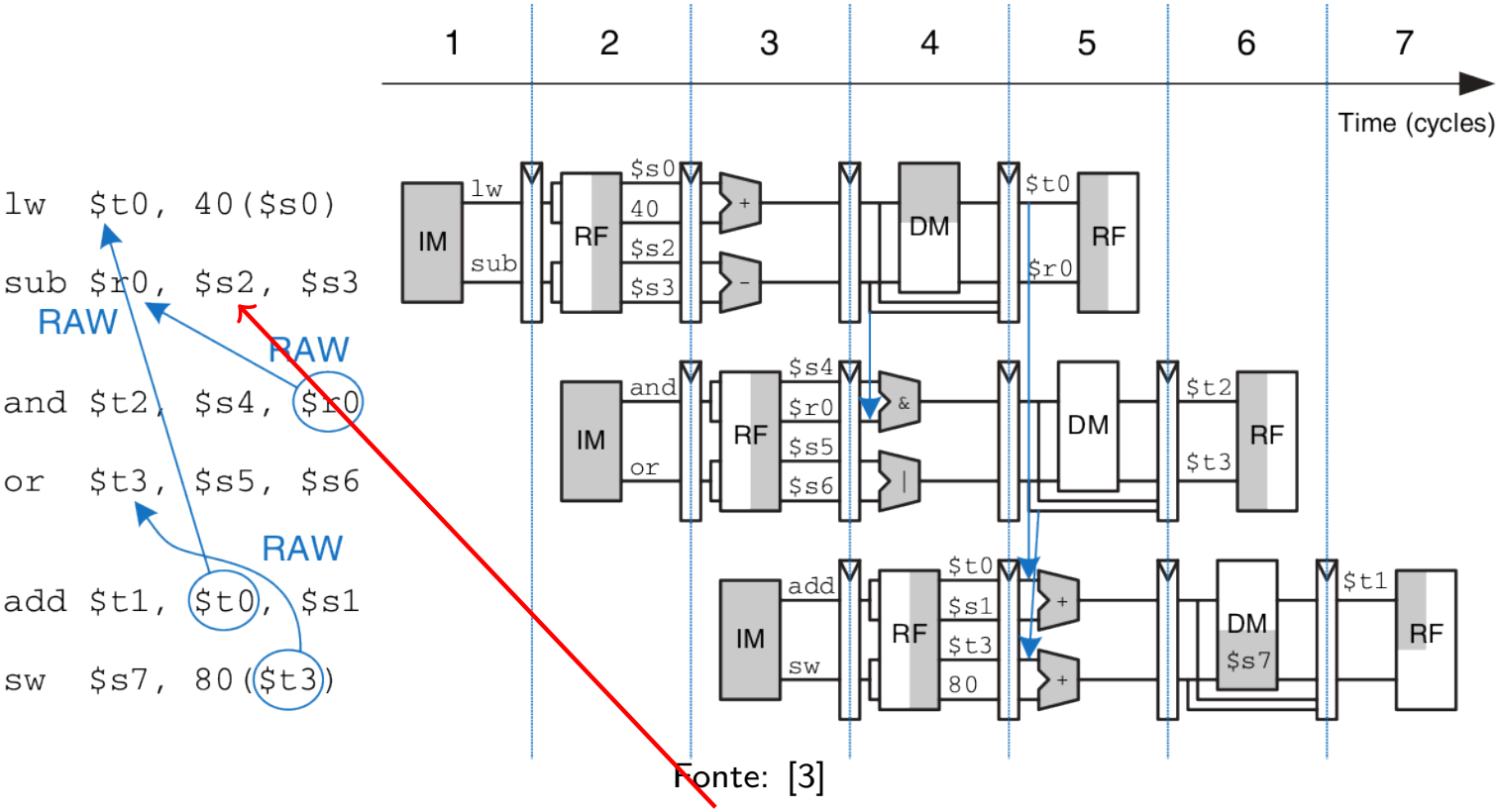

Fonte: [3]

**Ciclo 1**: `lw` é escolhido para rodar.

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



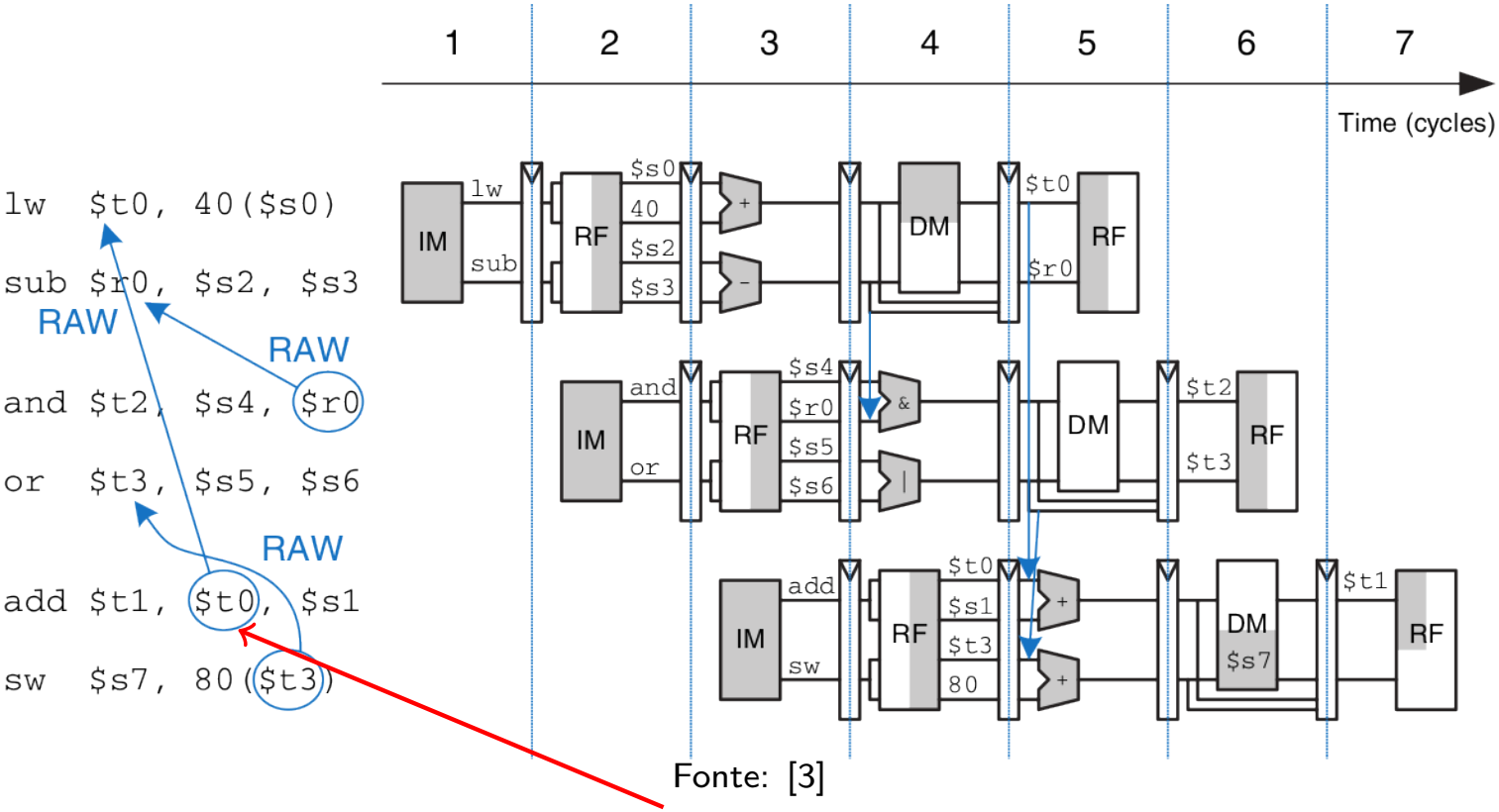

Fonte: [3]

Com `$r0`, `sub` ficou independente de `lw`, então é escolhida

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



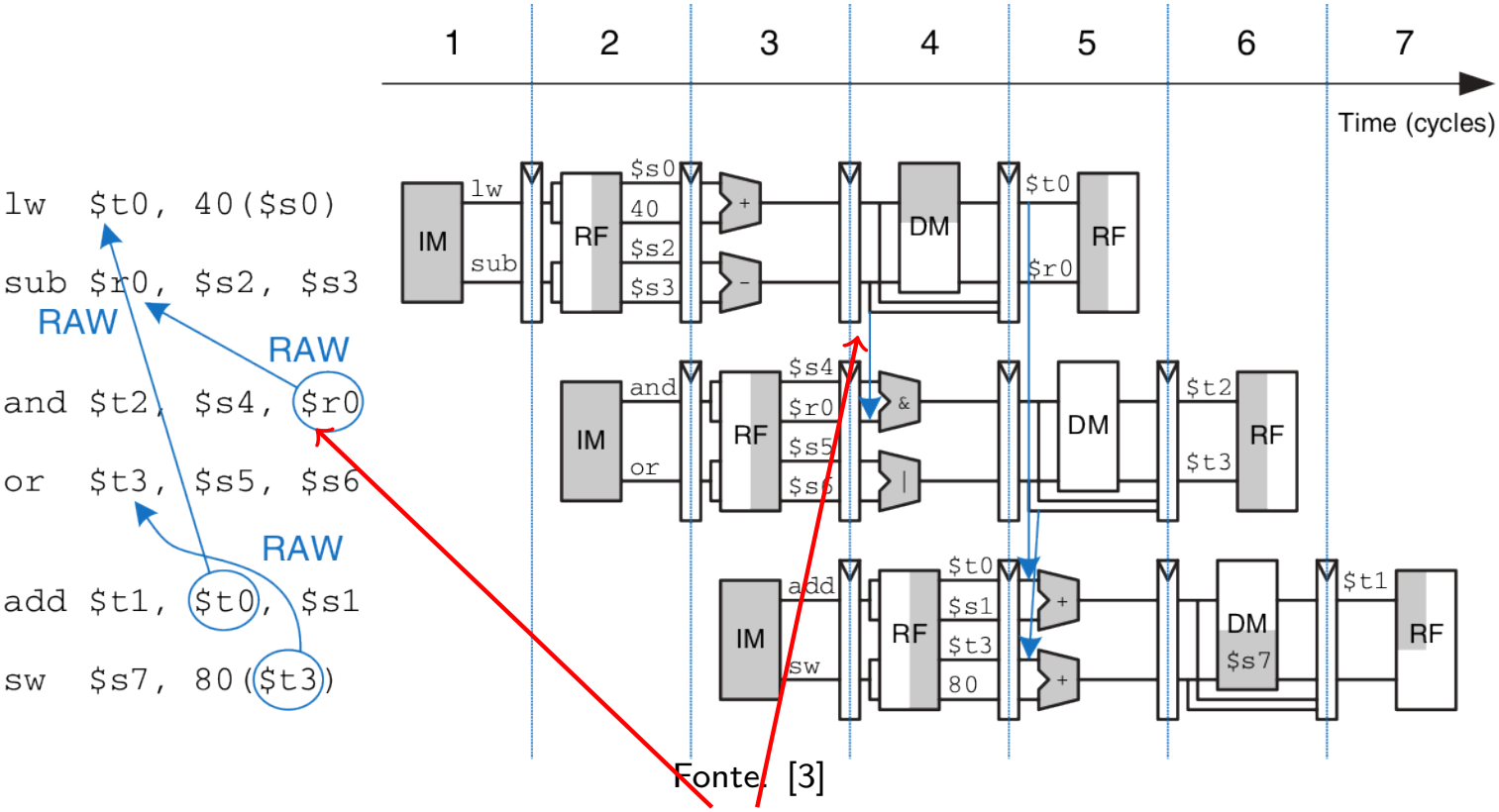

Fonte: [3]

**Ciclo 2**: `add` não pode rodar, pela latência de 2 ciclos em `lw`

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



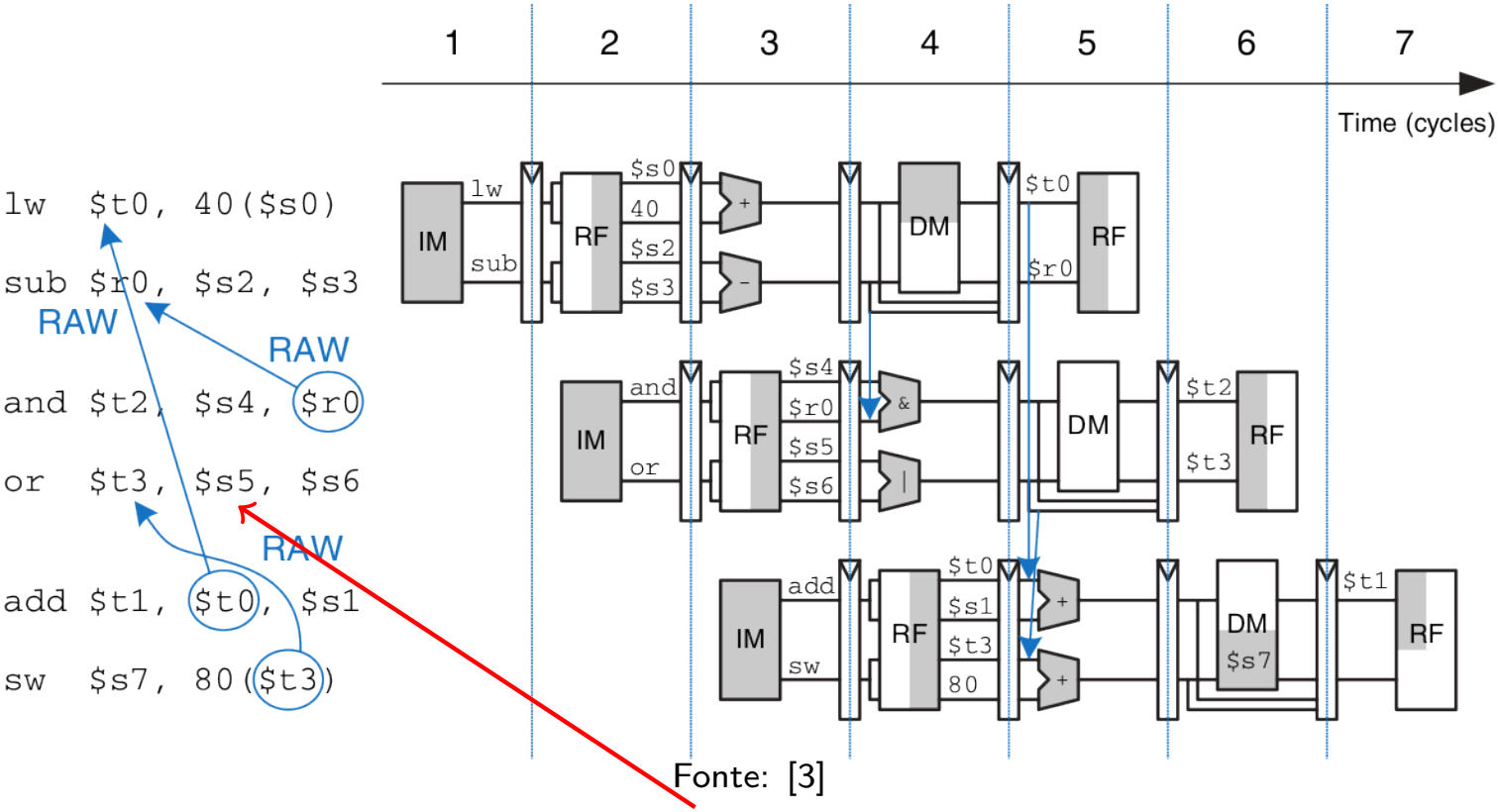

Fonte: [3]

`and` depende de `sub`, então podemos passar `$r0` por *forwarding*

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



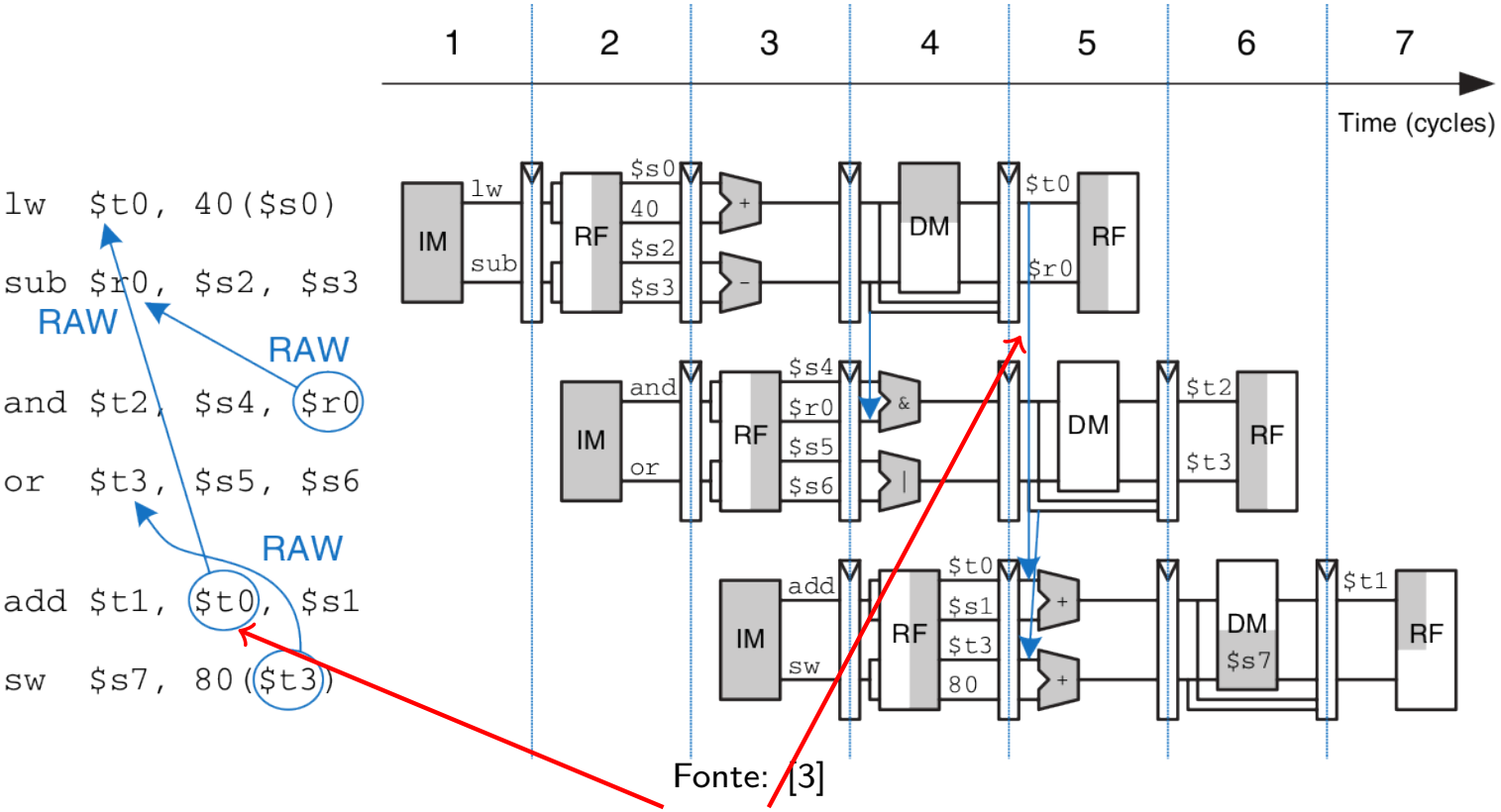

Fonte: [3]

Por ser independente, `or` também roda

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



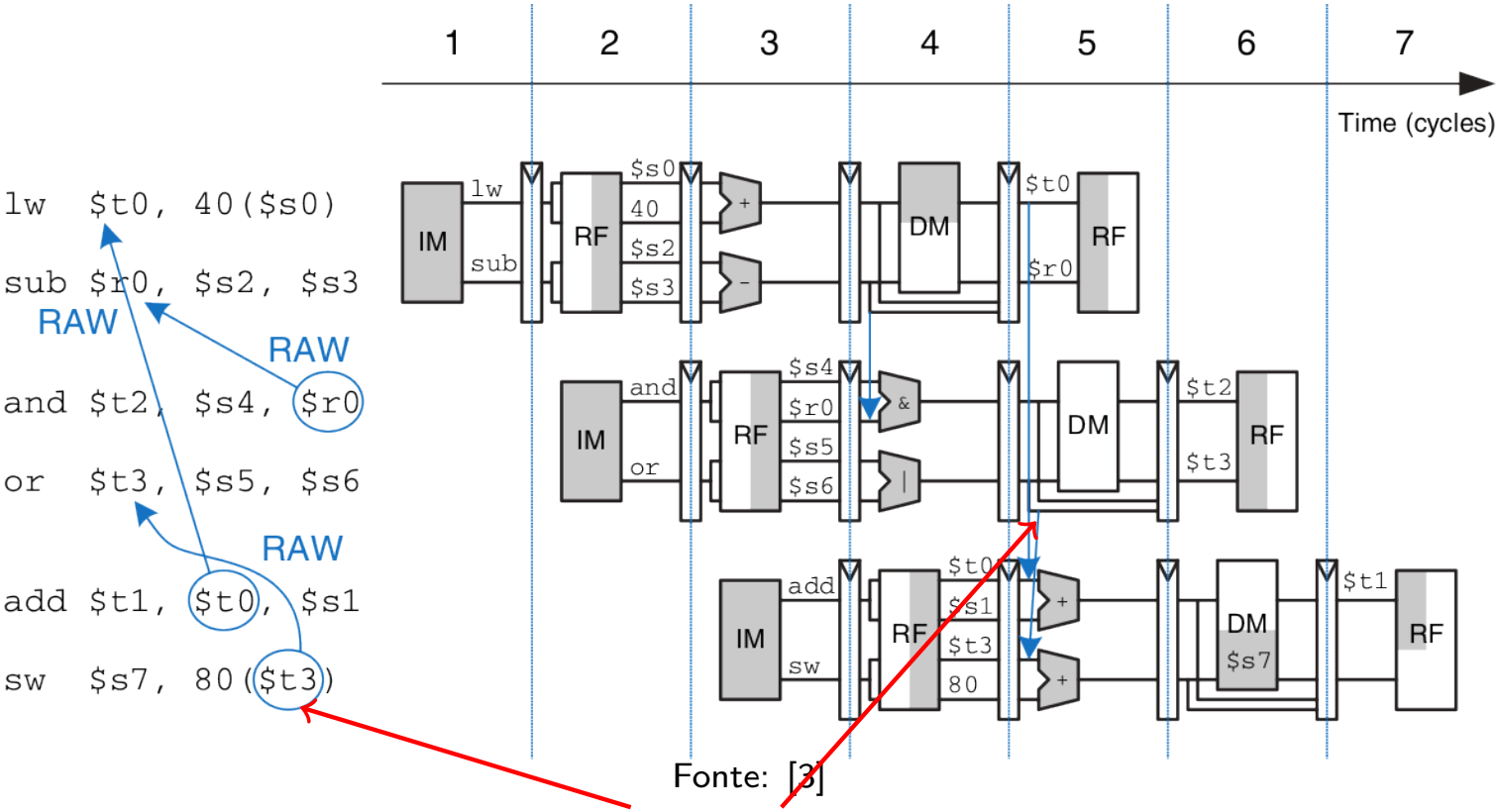

Fonte: [3]

**Ciclo 3**: `$t0` já está disponível (*forwarding*), então `add` pode rodar

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?



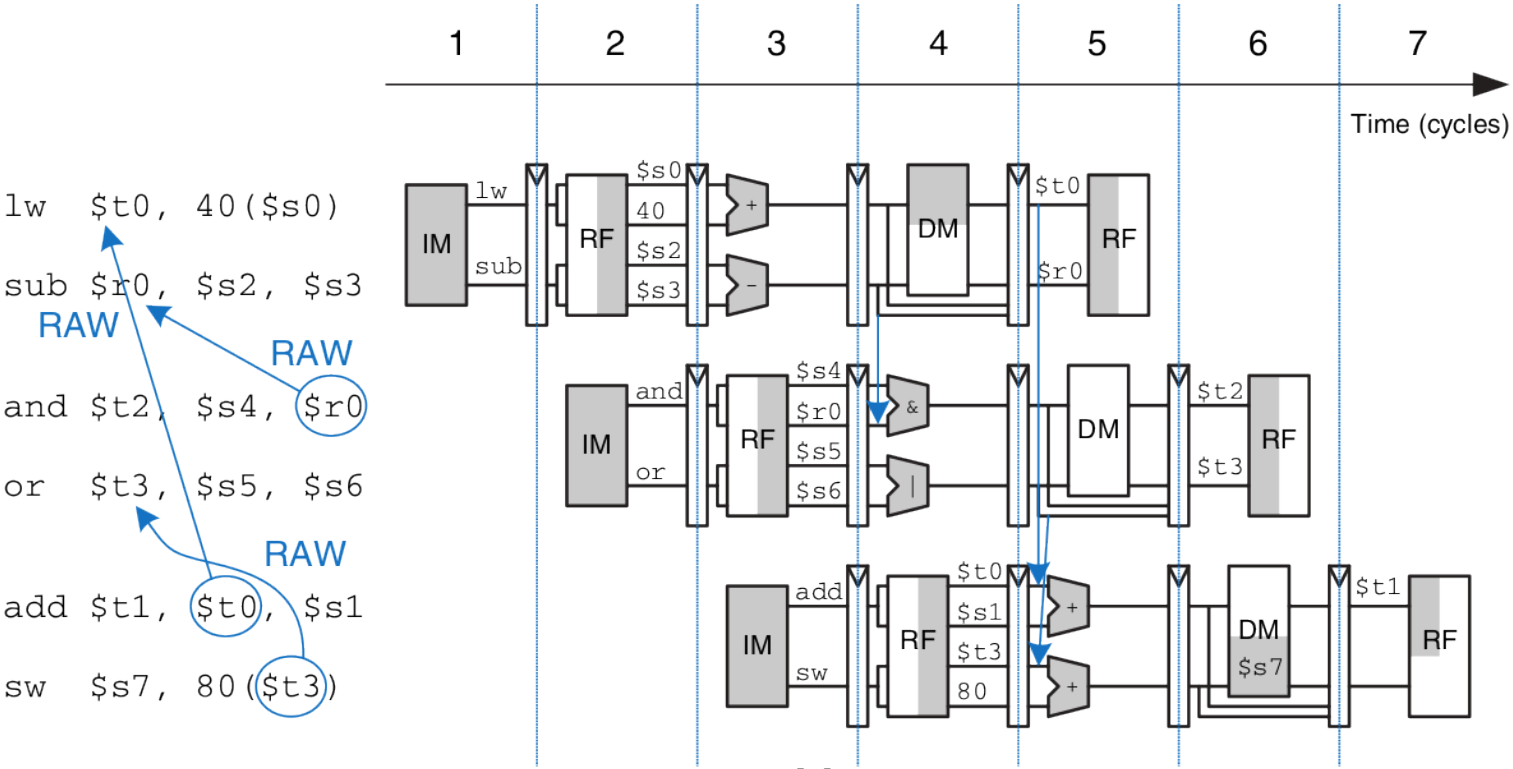

Fonte: [3]

$t3 também está disponível (*forwarding*), então `sw` roda

# Superescalar: Escalonamento Dinâmico

## Renomeação de Registradores: Exemplo

- E isso traz algum benefício?

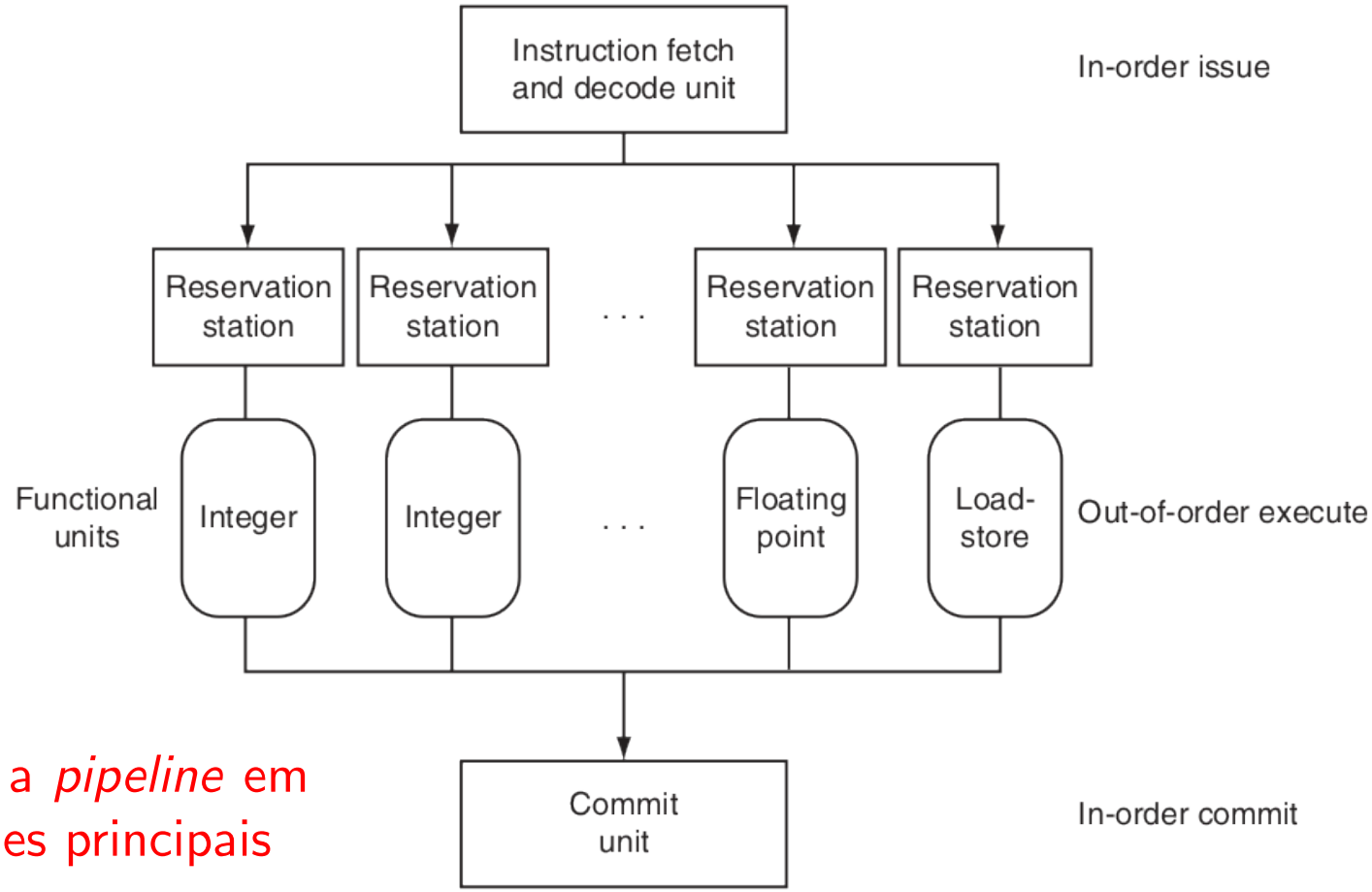



Fonte: [3]

E as mesmas 6 instruções agora são iniciadas em 3 ciclos, com $IPC = 2$

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

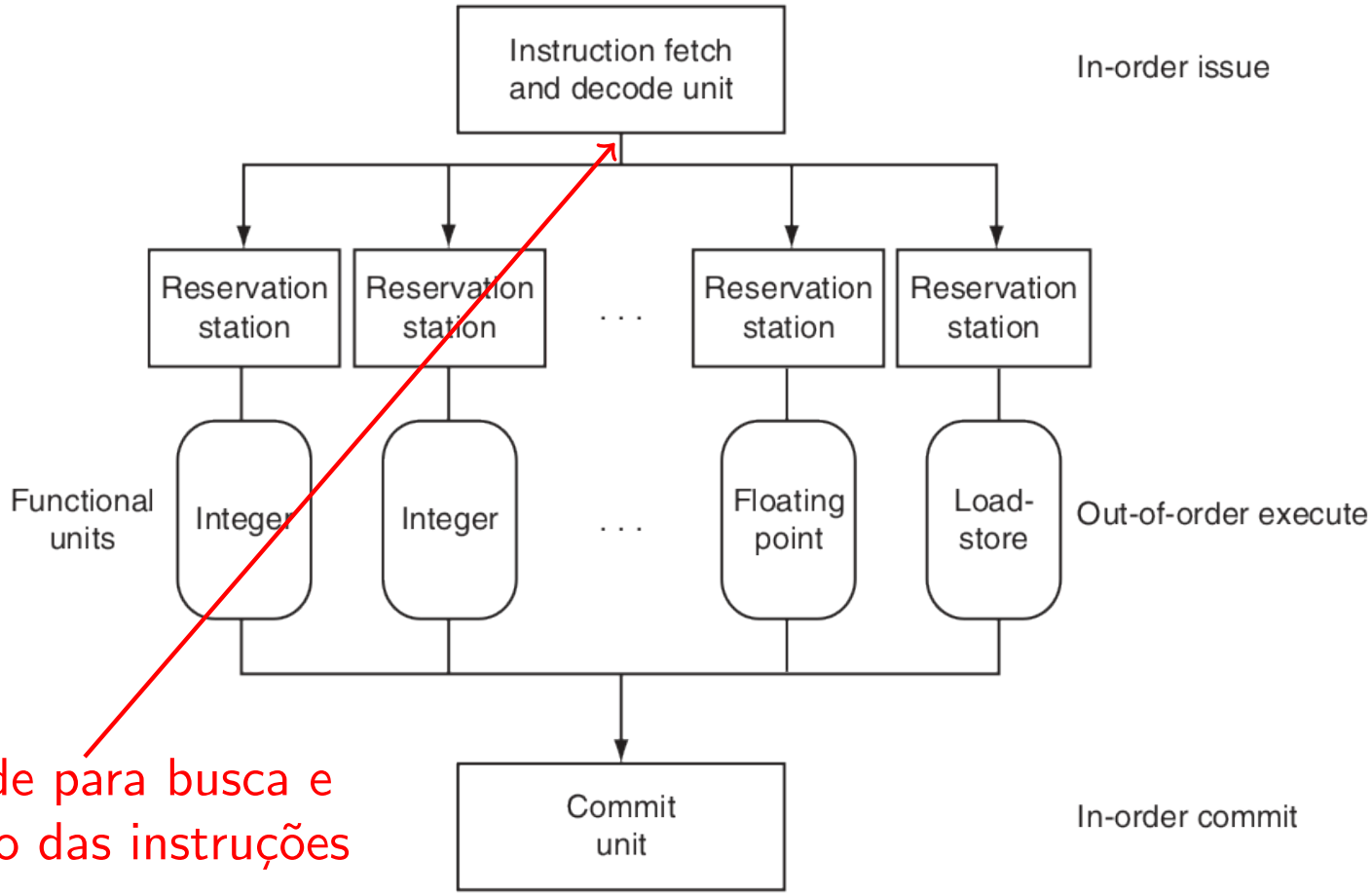



Fonte: [1]

Dividimos a *pipeline* em 3 unidades principais

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

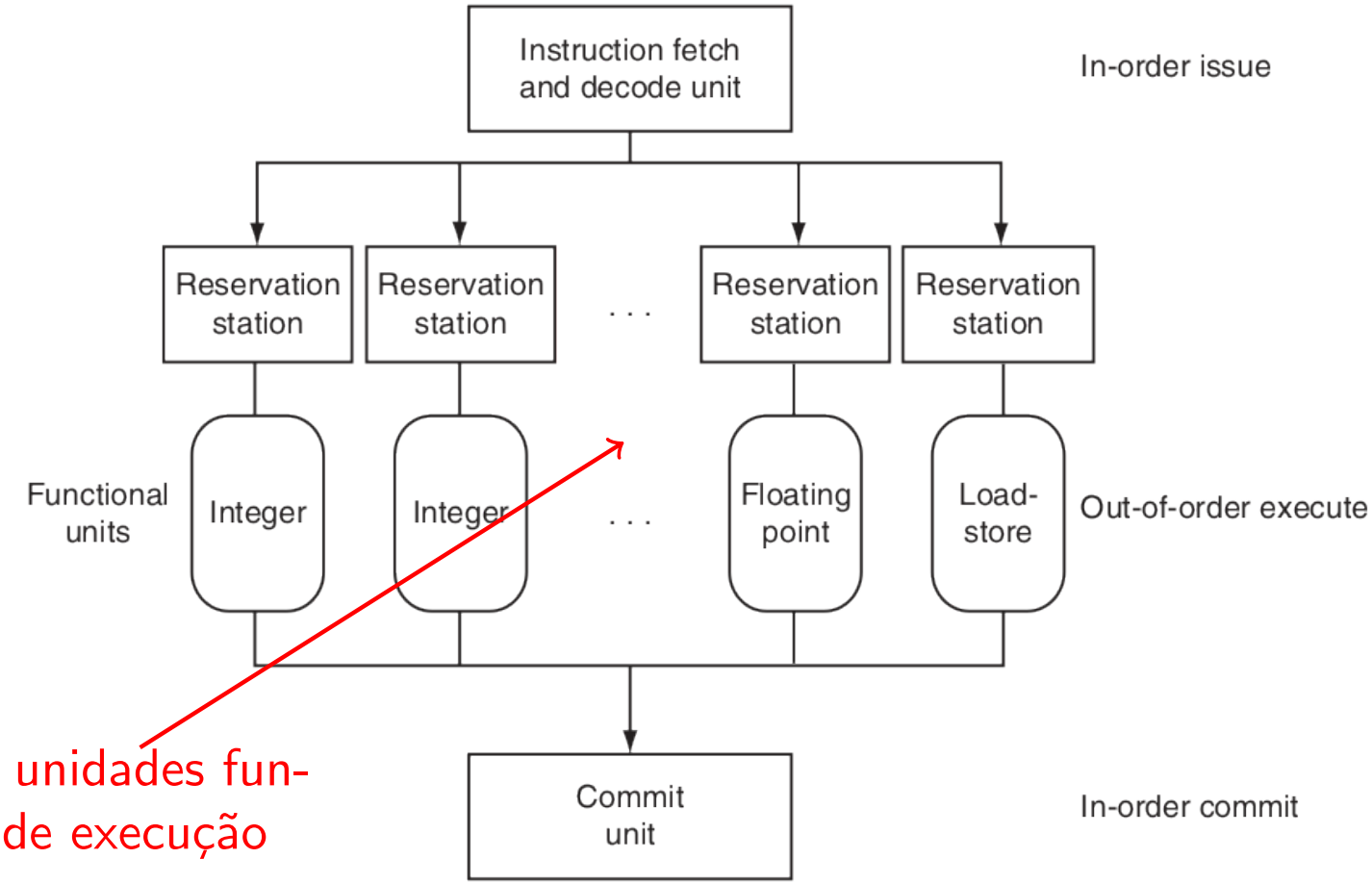



Uma unidade para busca e decodificação das instruções

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

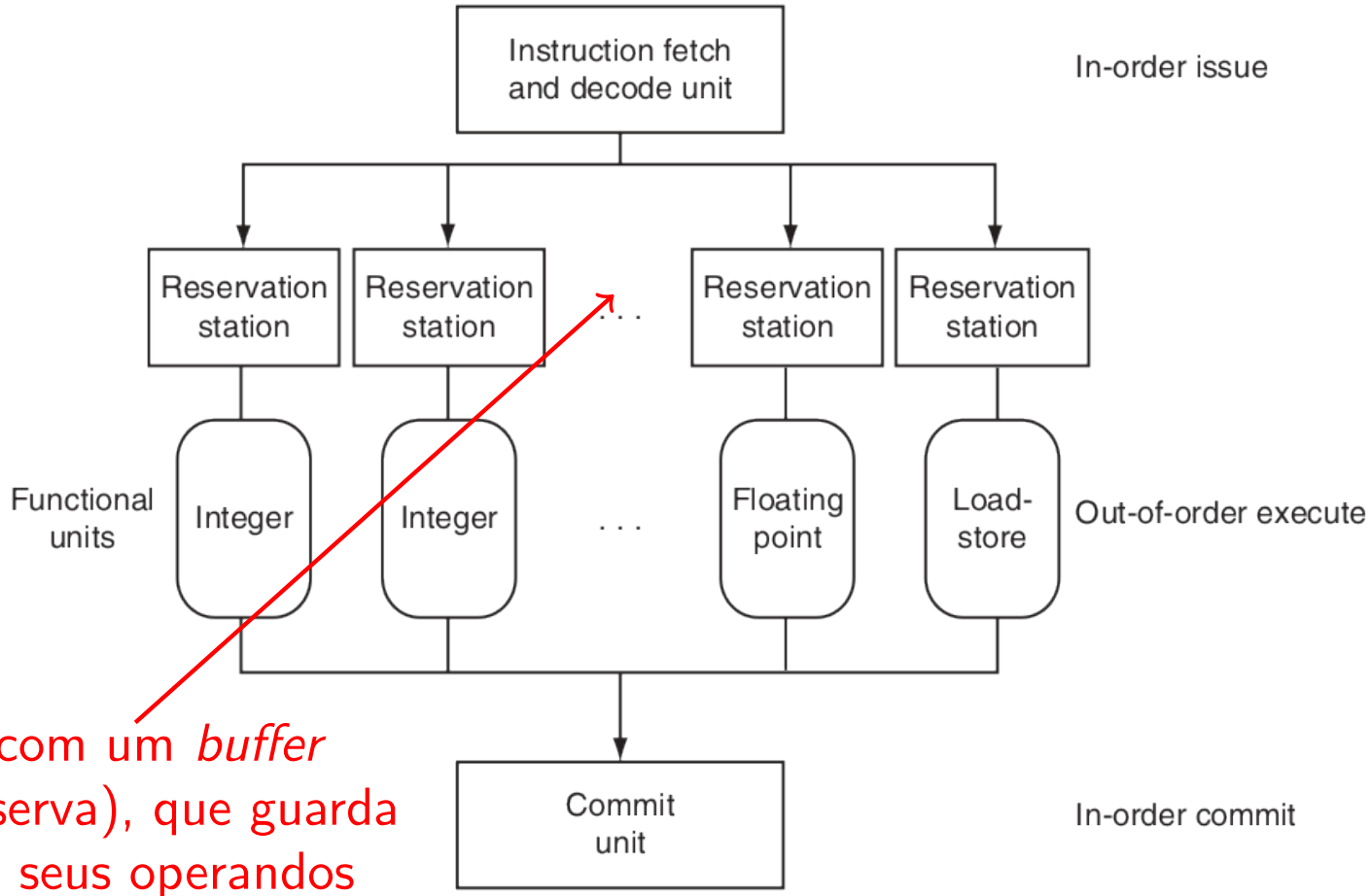
## E como é feito esse escalonamento dinâmico?




Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

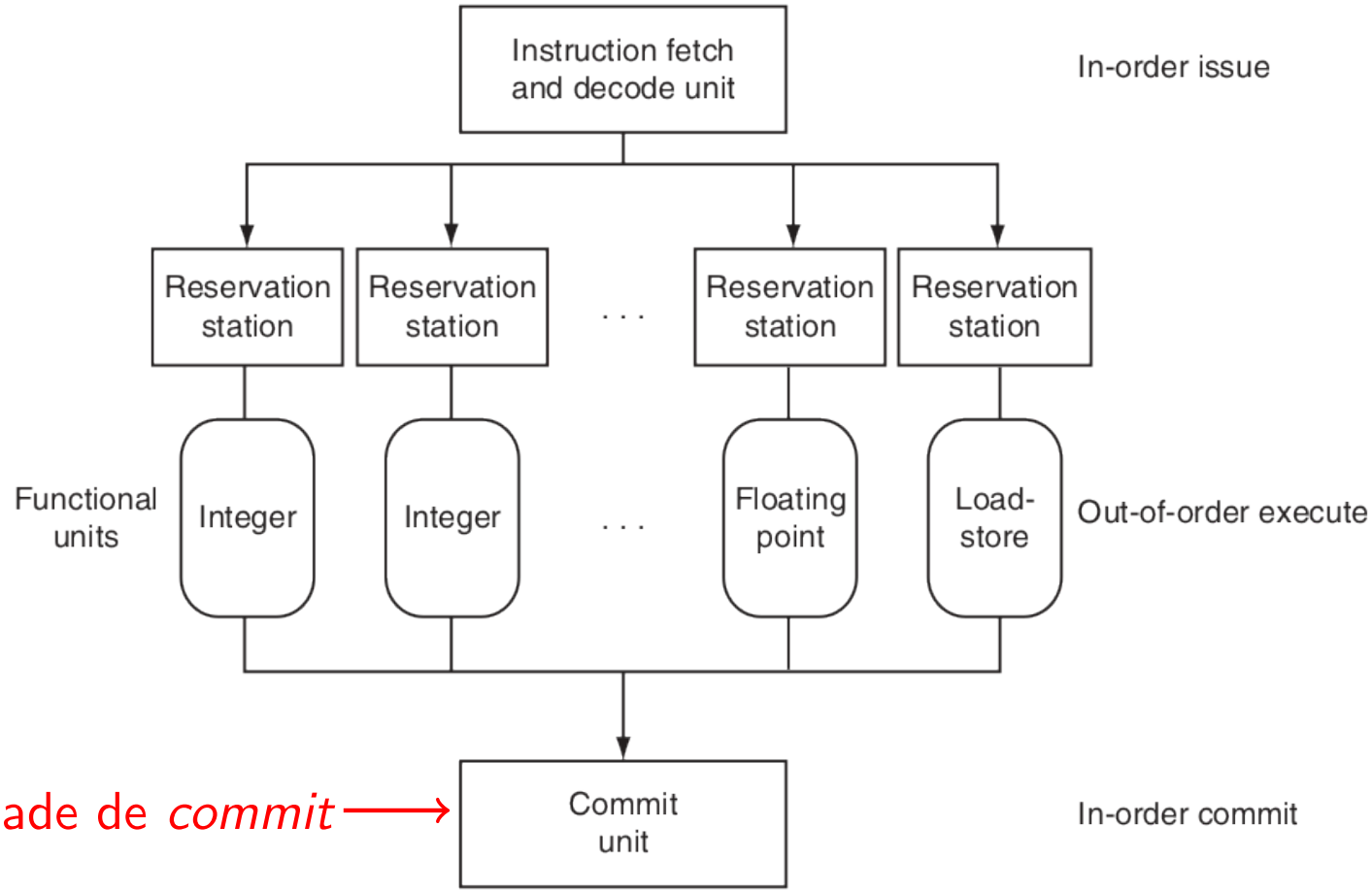



Cada uma com um *buffer* (estação de reserva), que guarda a operação e seus operandos

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

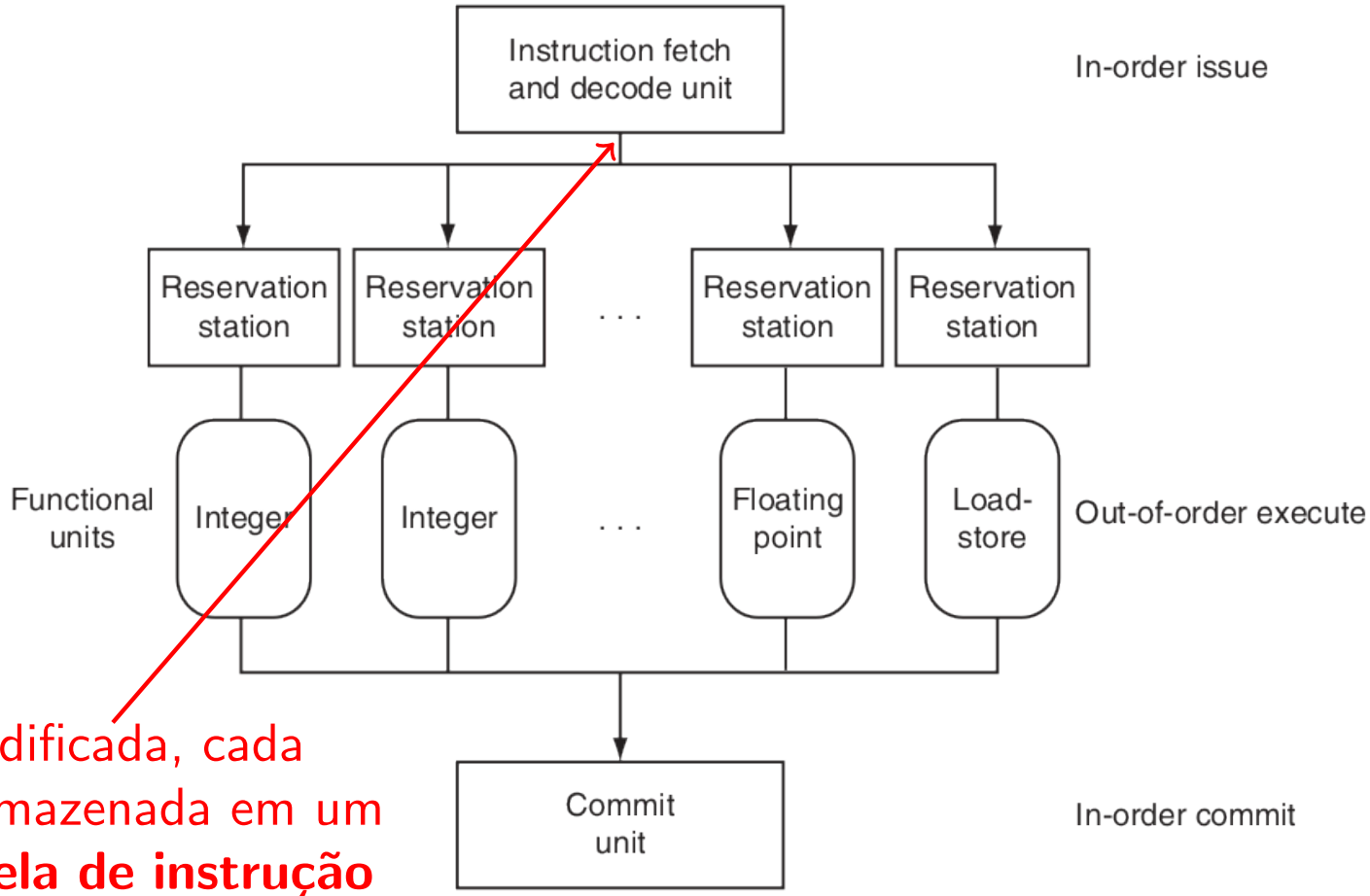



E uma unidade de *commit* ⟶

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

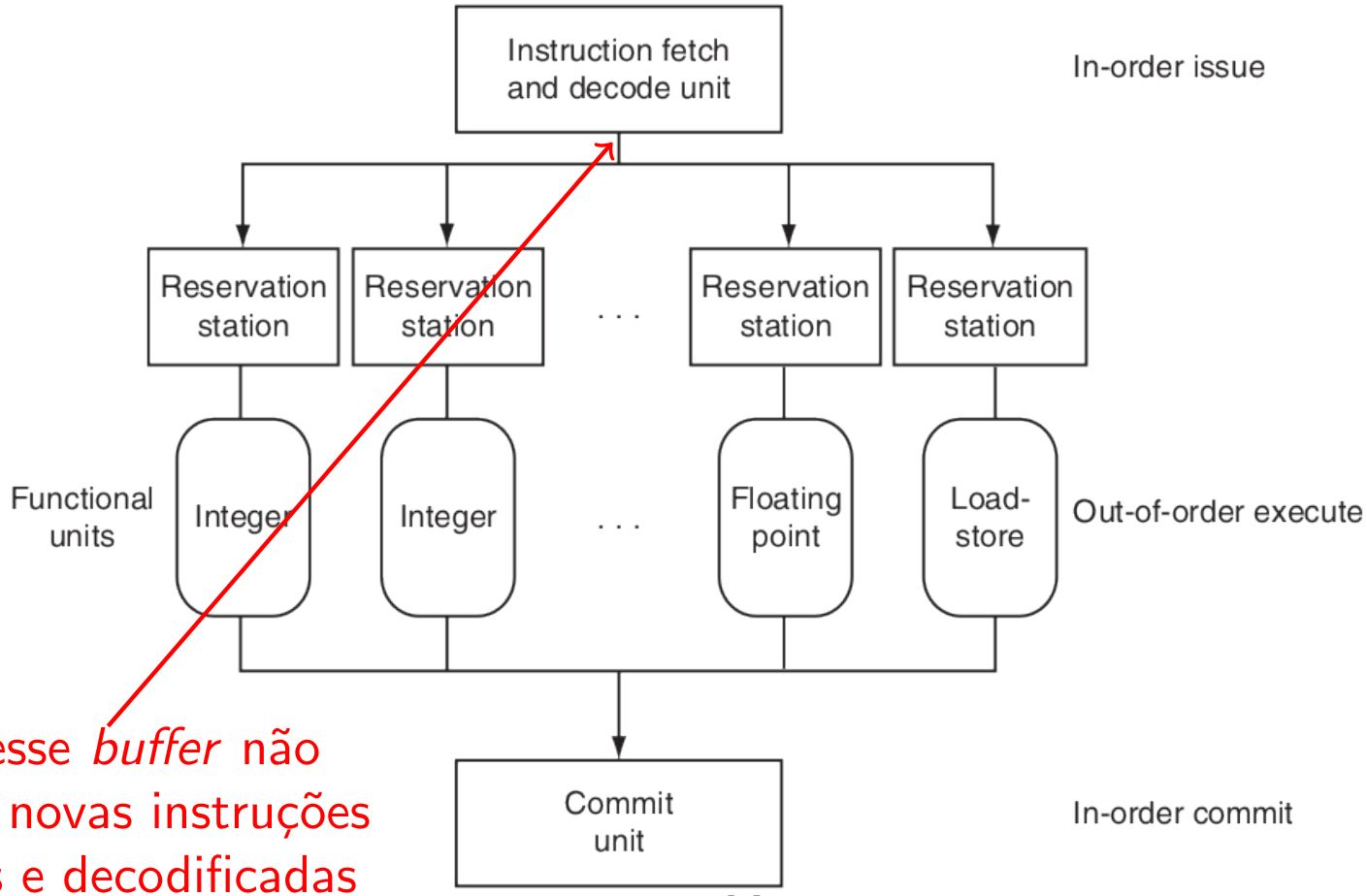



Após decodificada, cada instrução é armazenada em um *buffer*: a **janela de instrução**

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

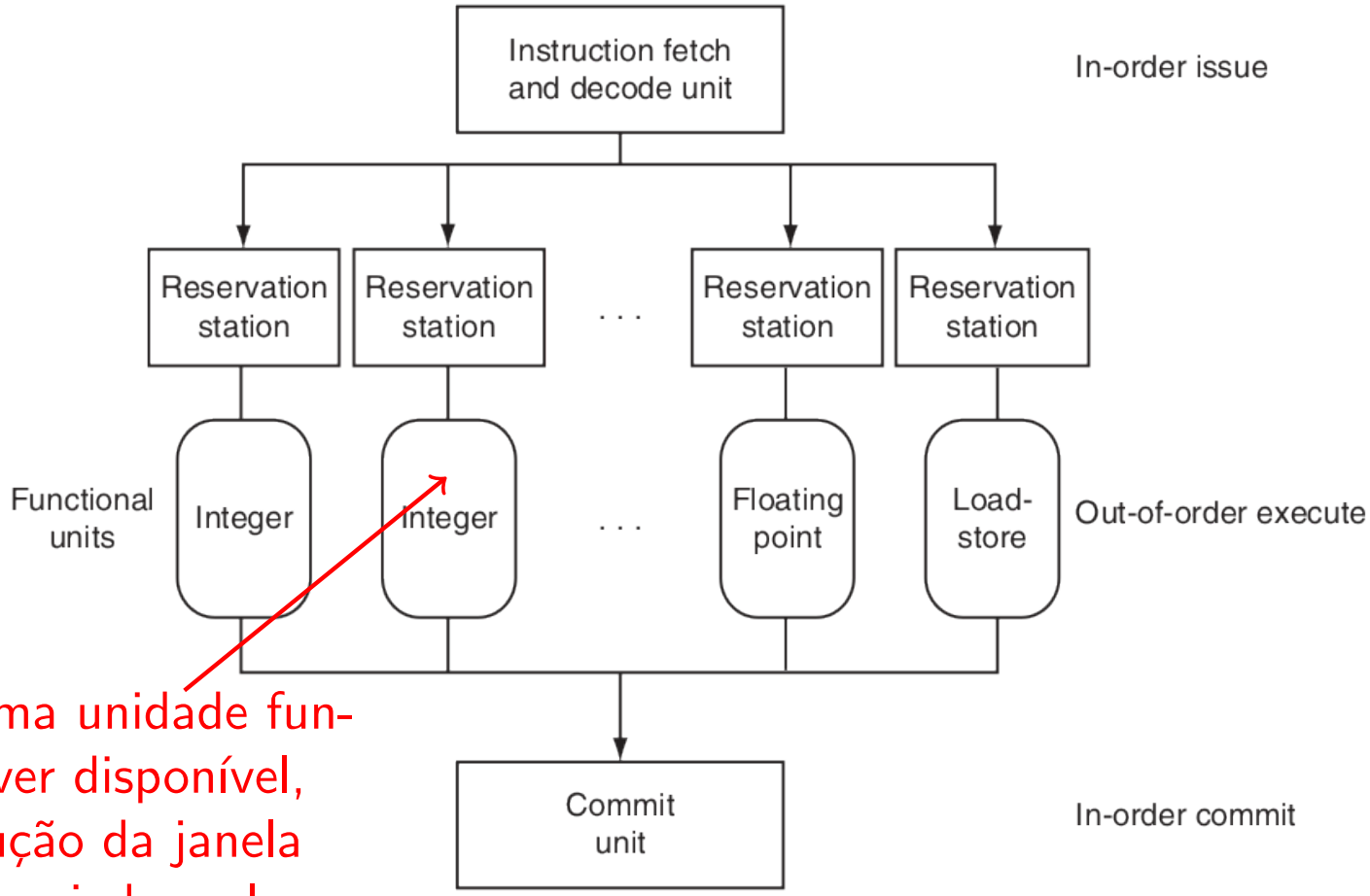



Enquanto esse *buffer* não estiver cheio, novas instruções são buscadas e decodificadas

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

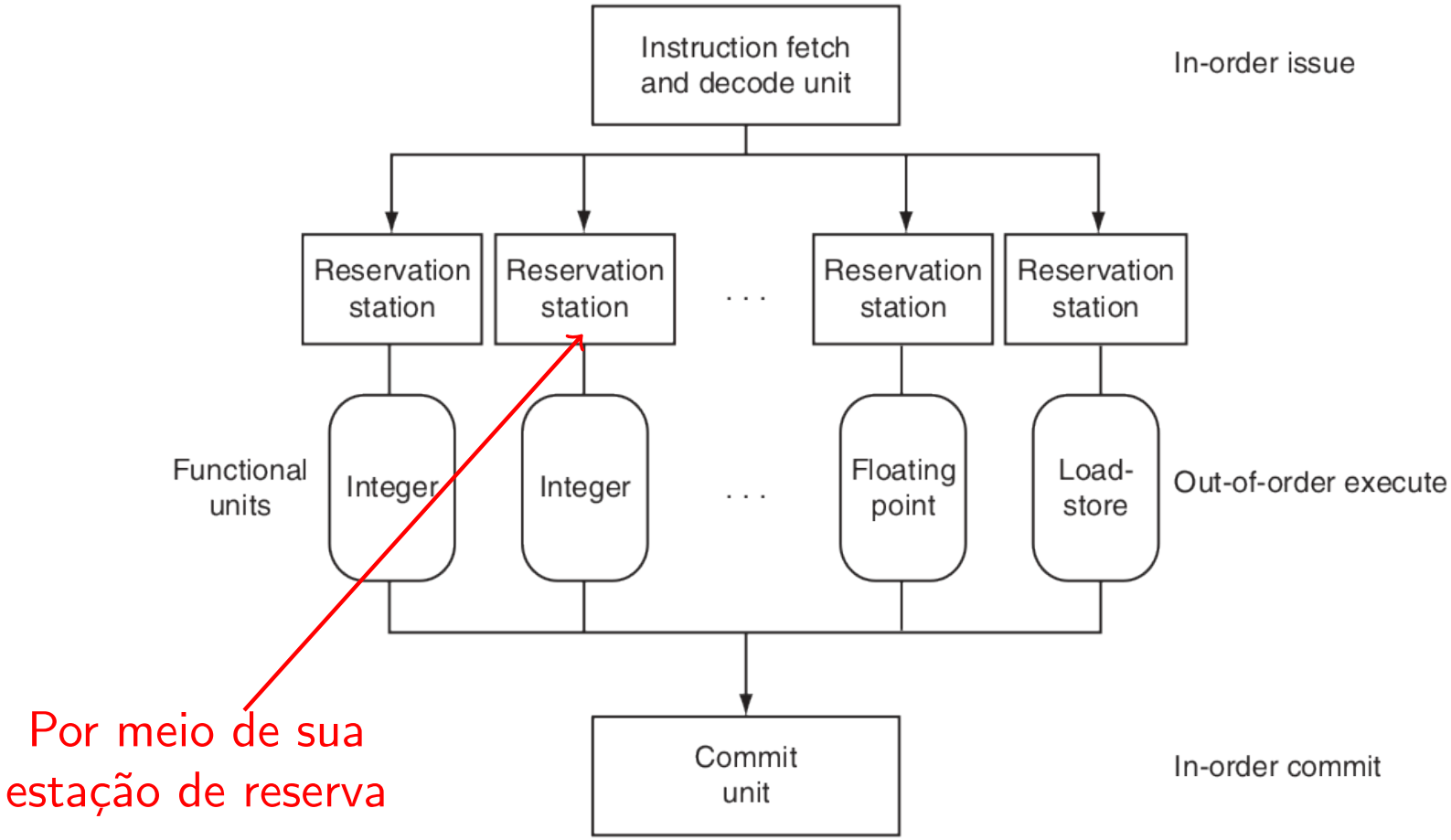



Assim que uma unidade funcional estiver disponível, uma instrução da janela pode ser enviada a ela

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

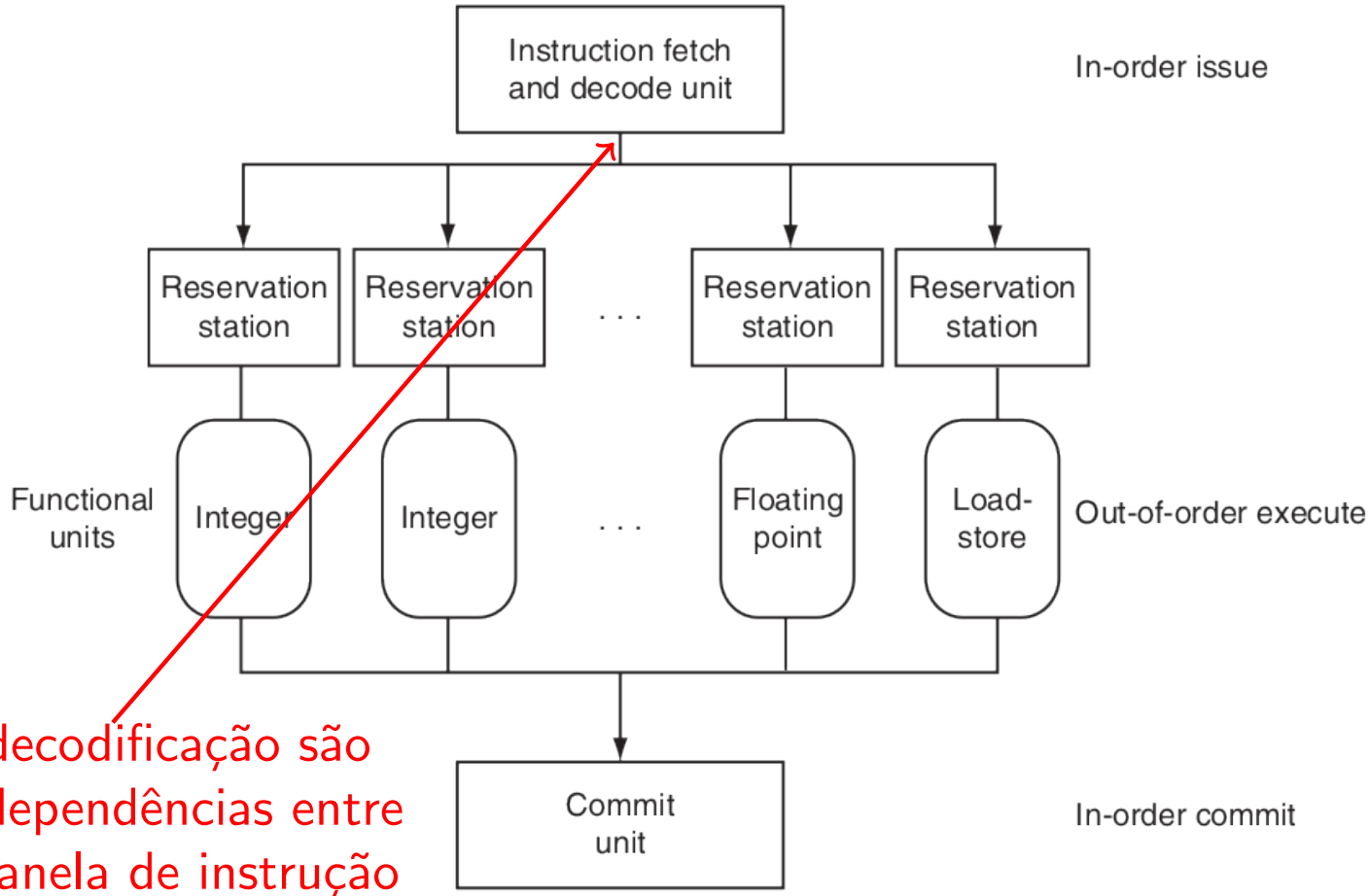
## E como é feito esse escalonamento dinâmico?




Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

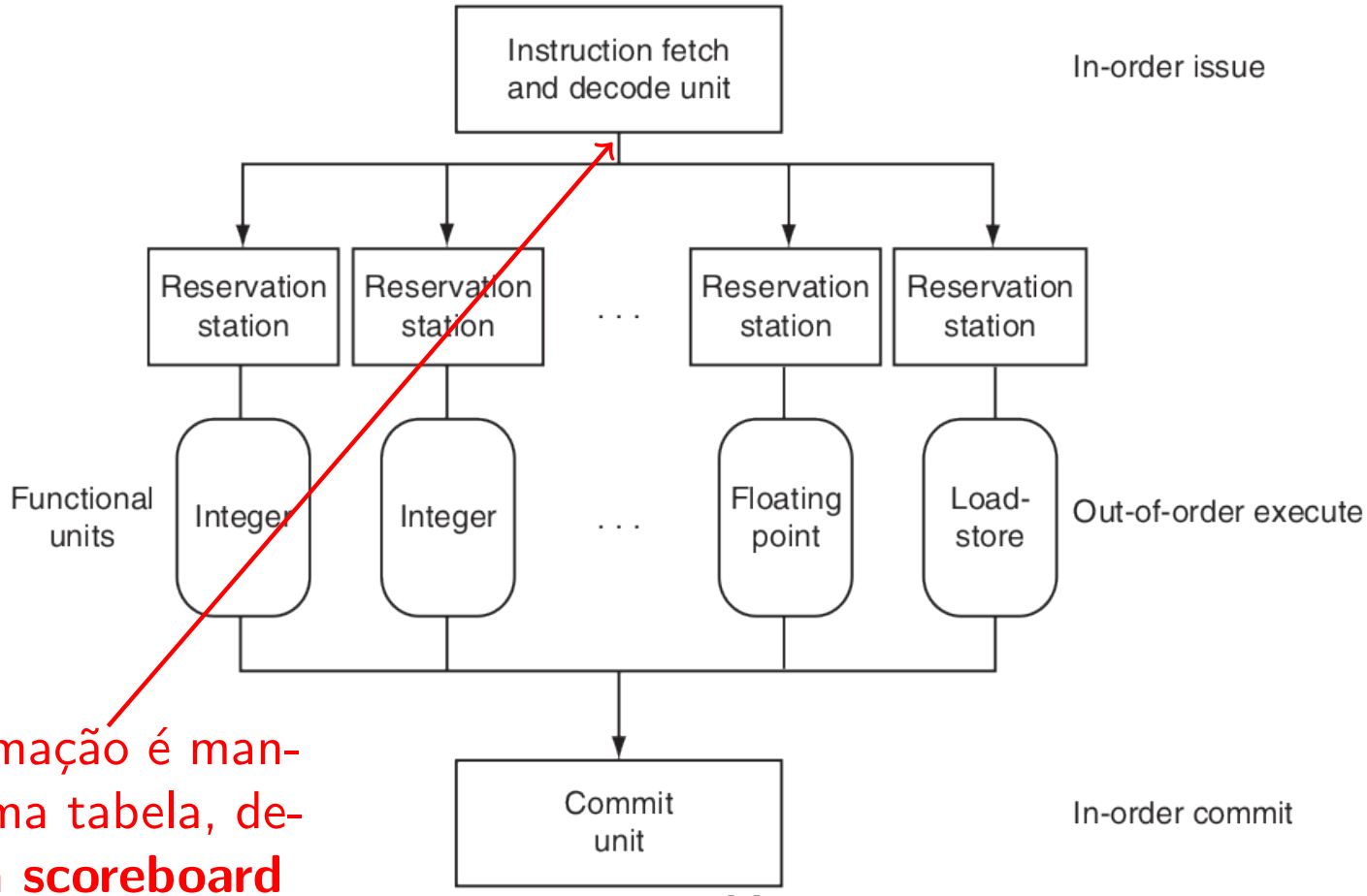



Também na decodificação são verificadas as dependências entre instruções na janela de instrução

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

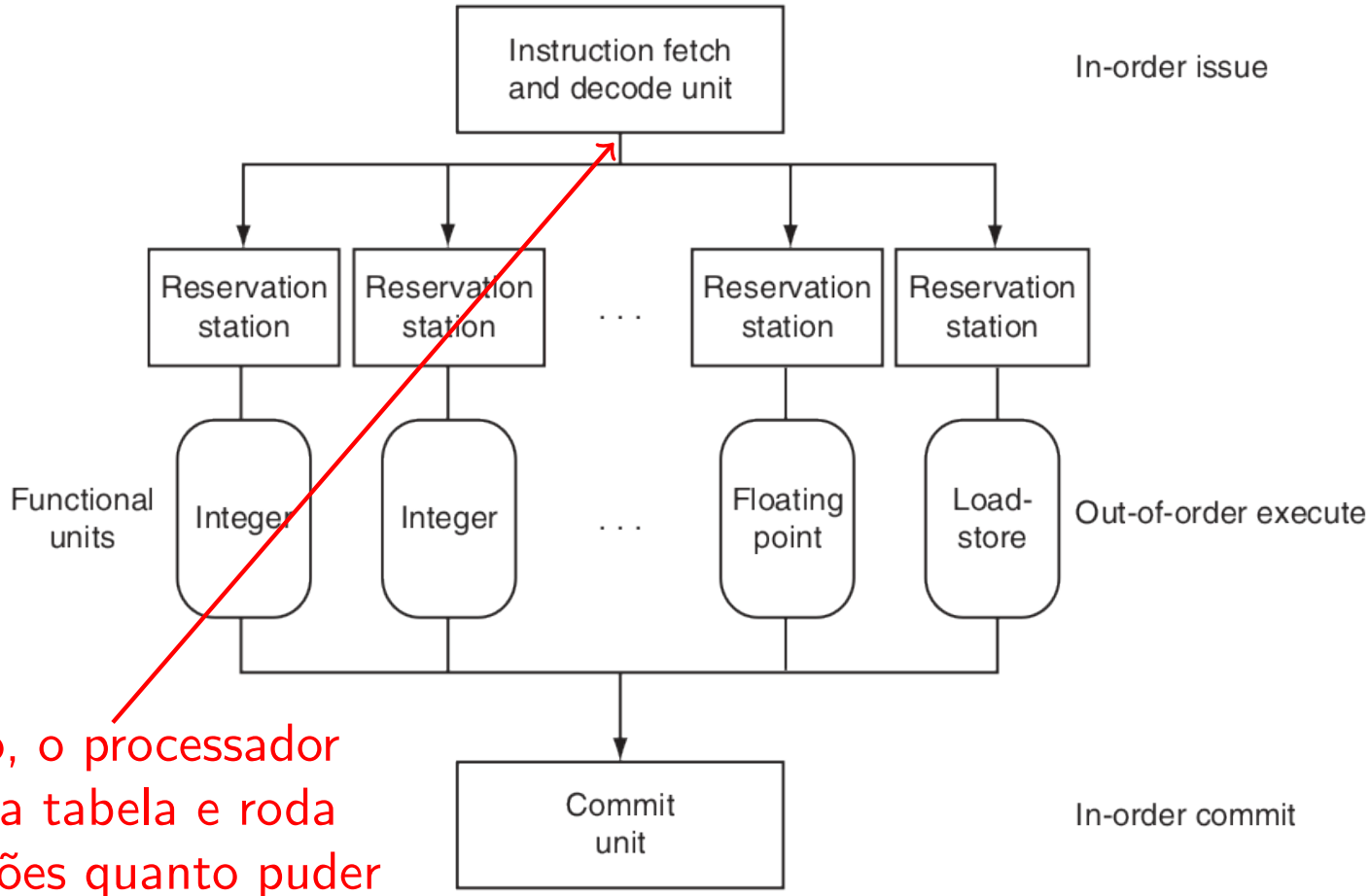



Essa informação é mantida em uma tabela, denominada **scoreboard**

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

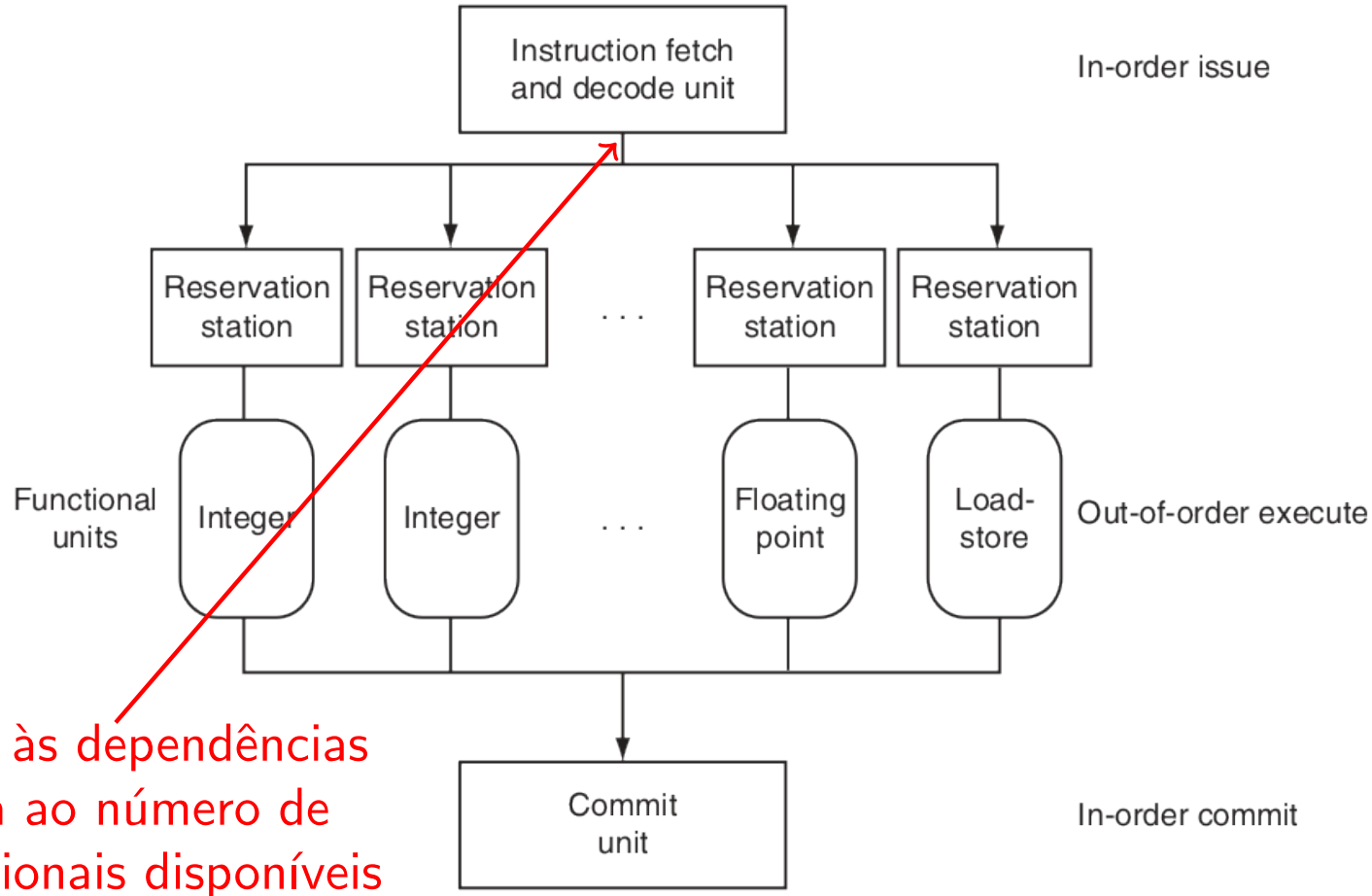



A cada ciclo, o processador examina essa tabela e roda tantas instruções quanto puder

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

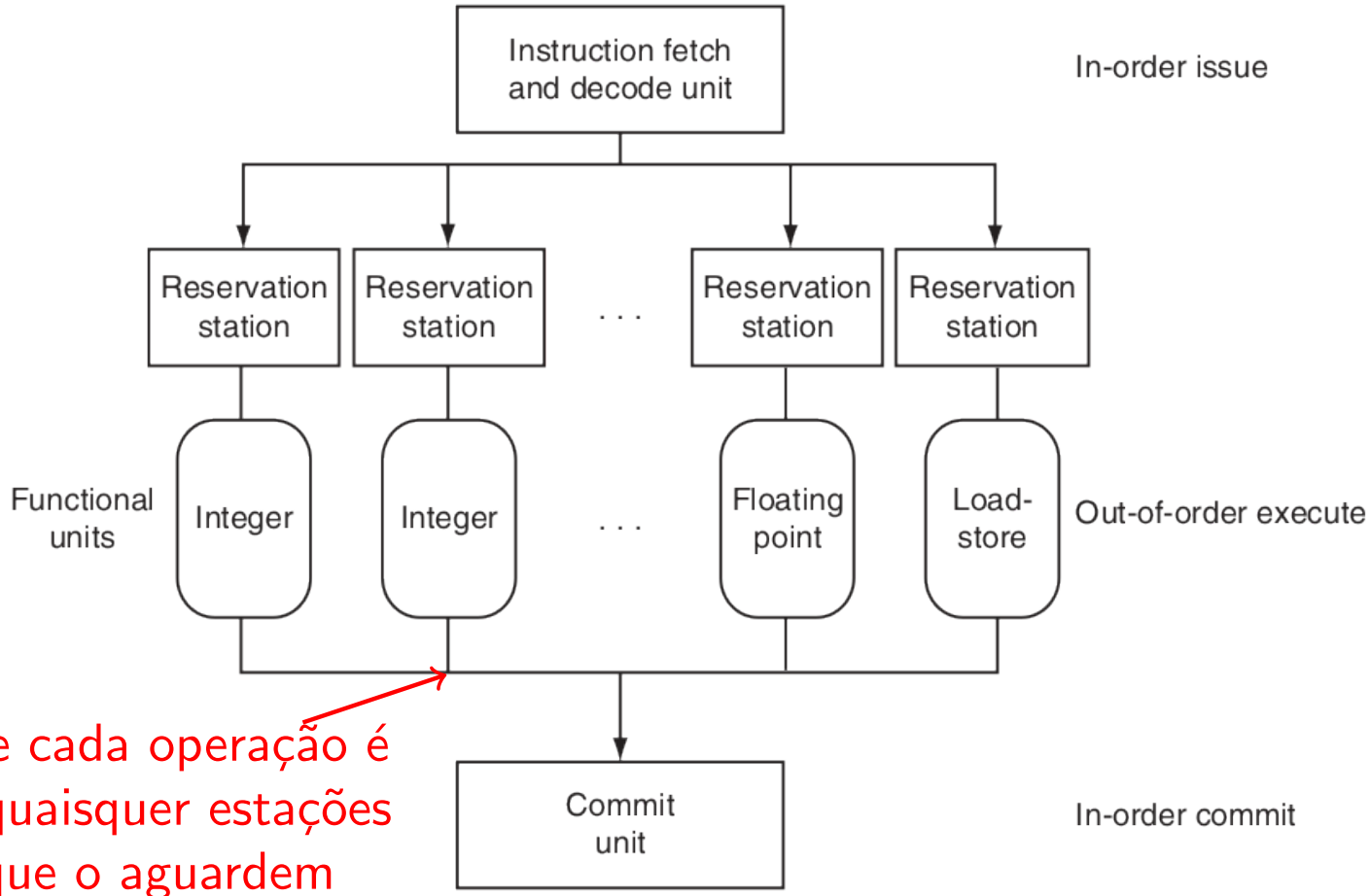



Limitando-se às dependências existentes a ao número de unidades funcionais disponíveis

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

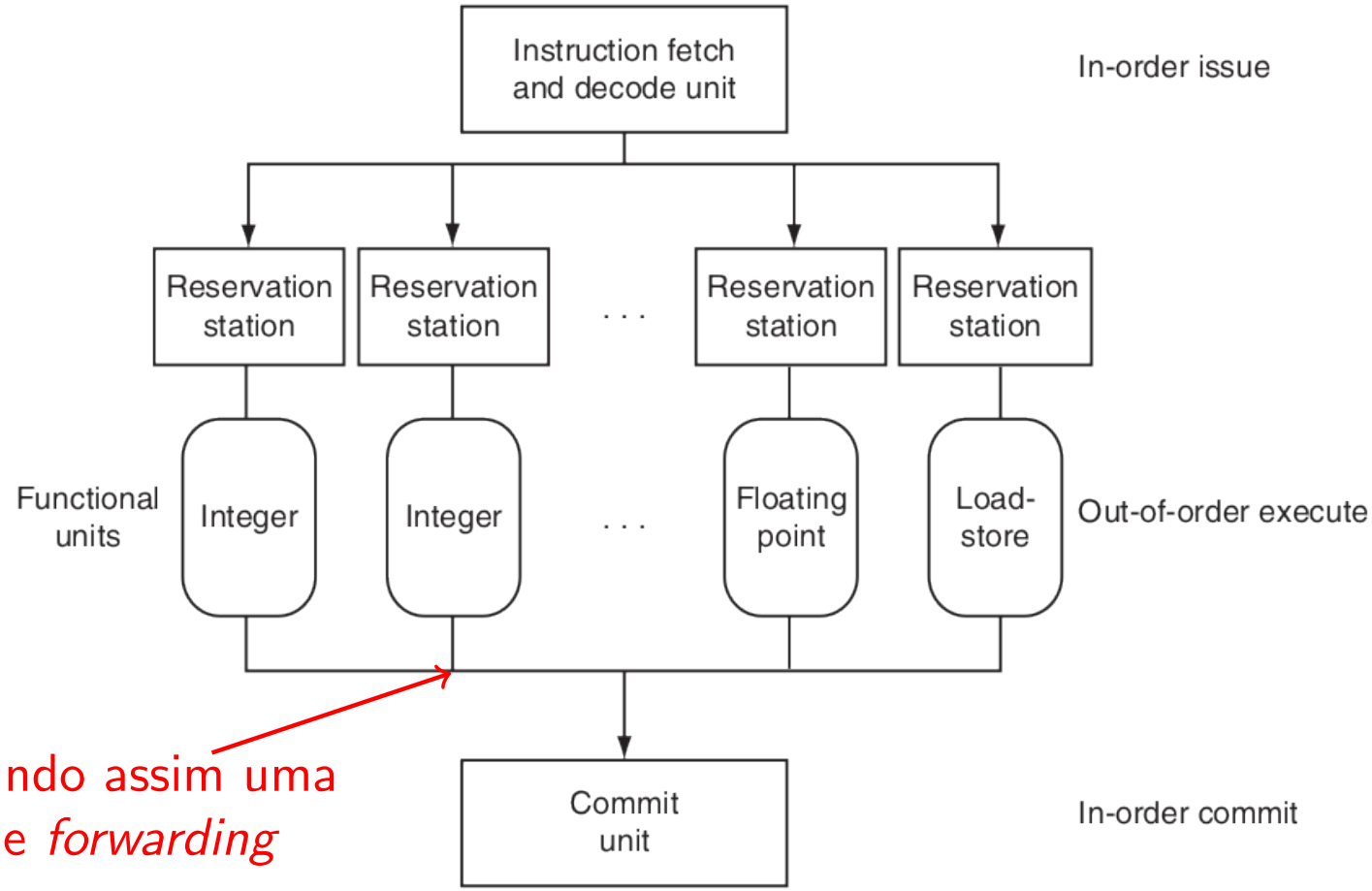



O resultado de cada operação é enviado para quaisquer estações de reserva que o aguardem

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

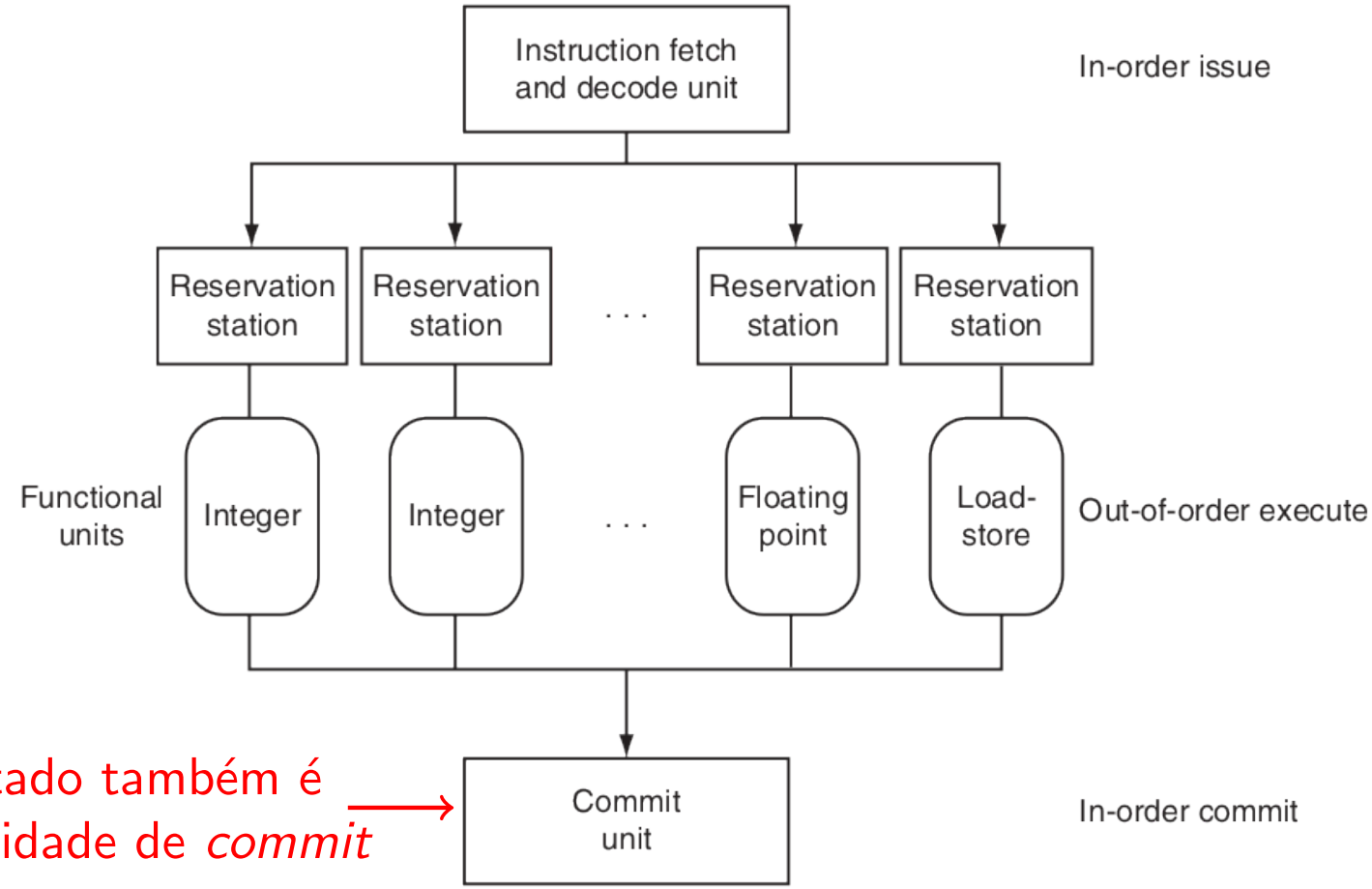



Implementando assim uma espécie de *forwarding*

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

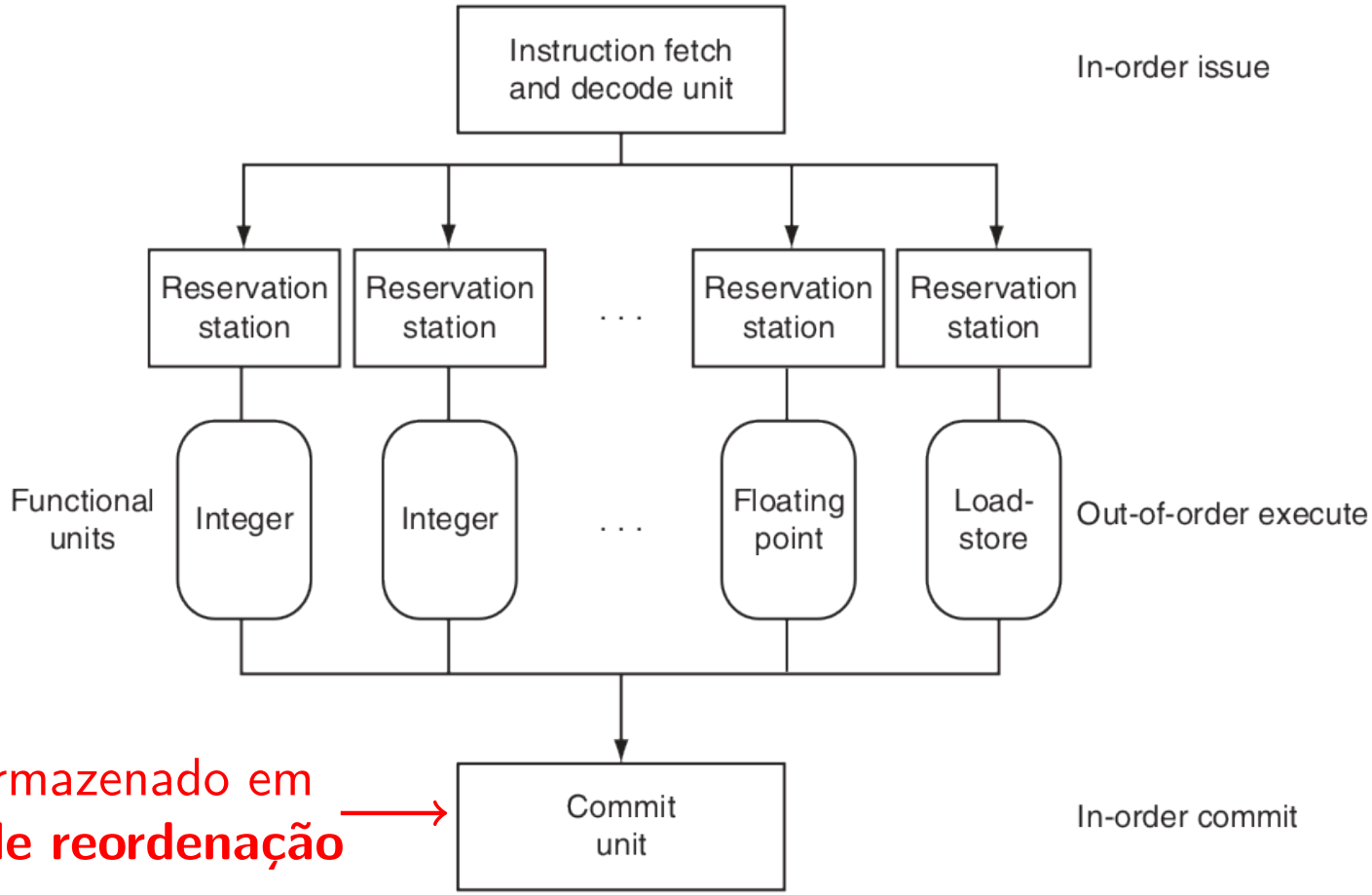



Esse resultado também é enviado à unidade de *commit*

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

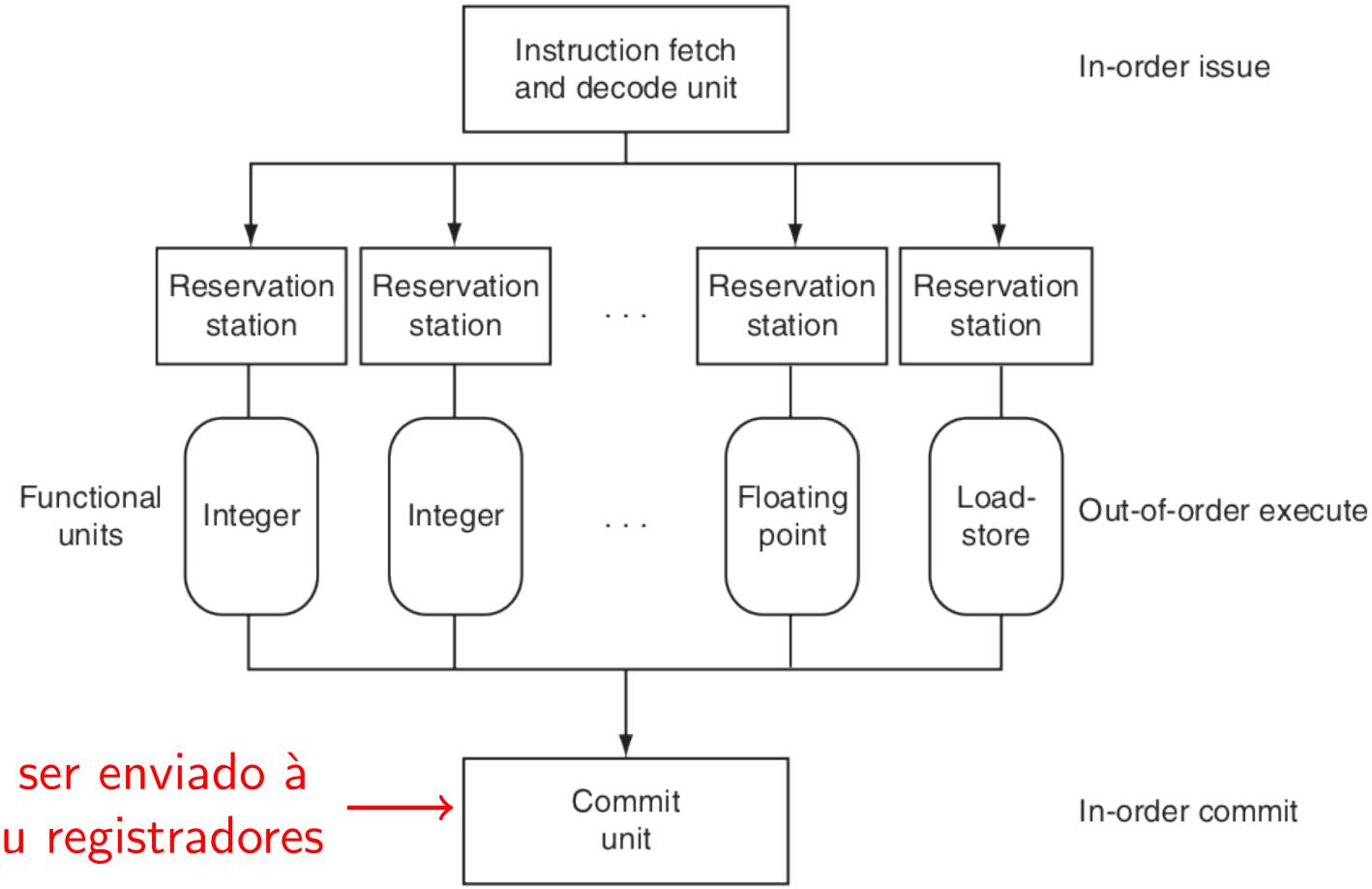



Lá, ele é armazenado em um **buffer de reordenação**

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

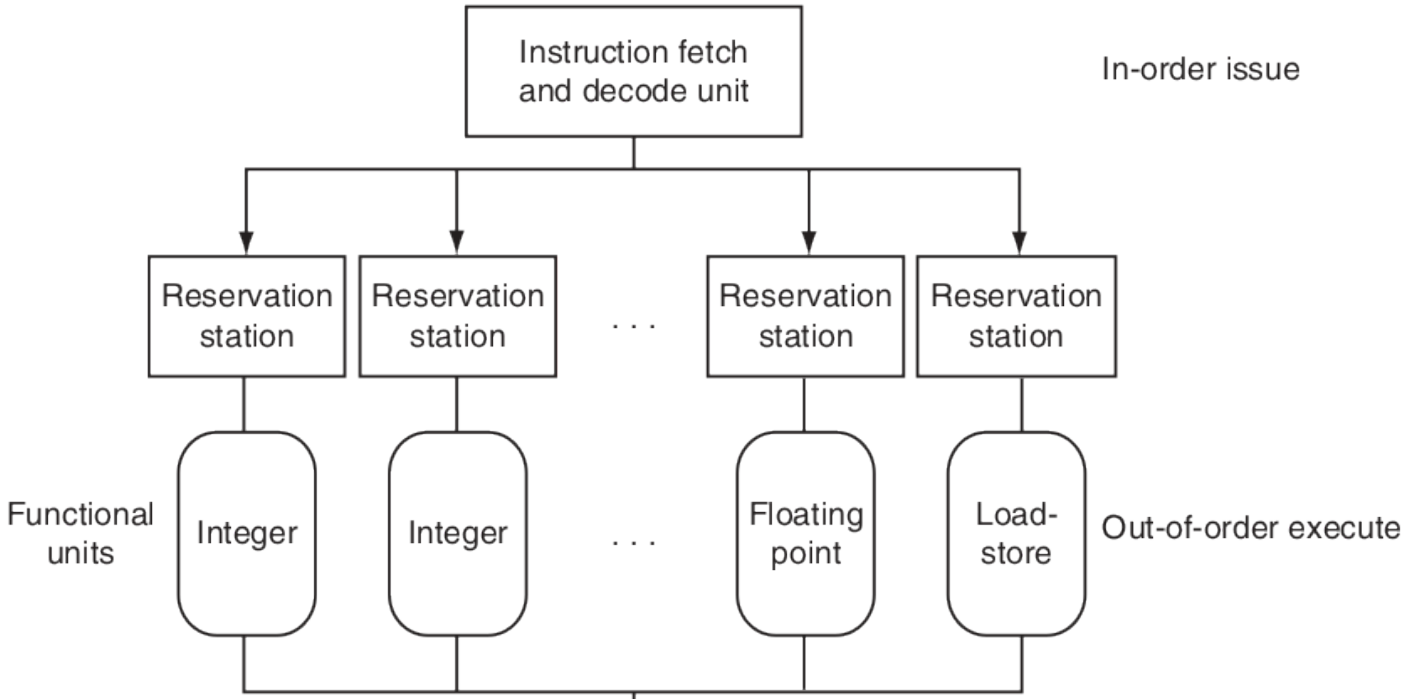



Até poder ser enviado à memória ou registradores →

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

## E como é feito esse escalonamento dinâmico?

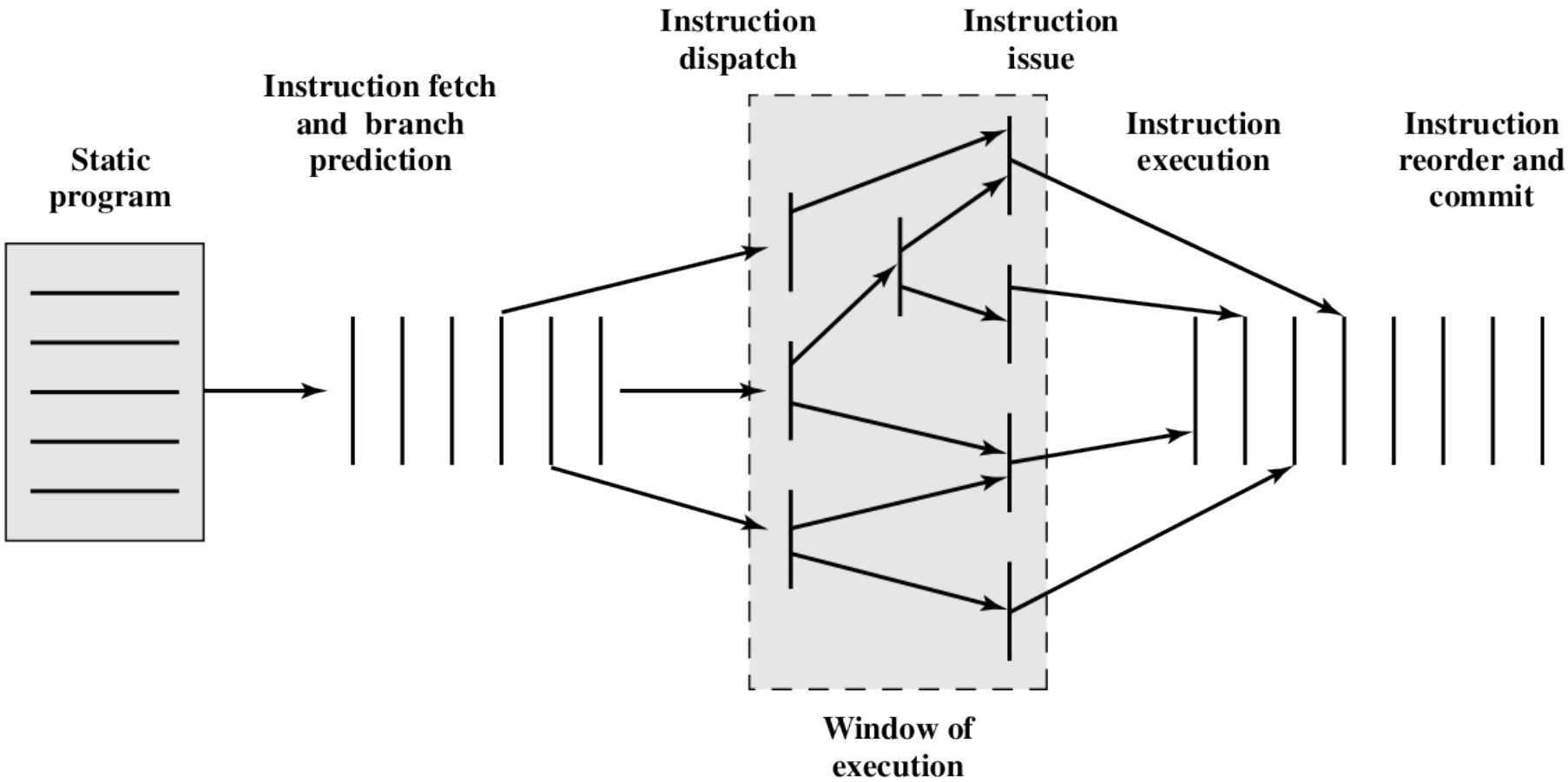



Caso o resultado pertença a um *branch* erroneamente predito, ele é descartado

Fonte: [1]

# Superescalar: Escalonamento Dinâmico

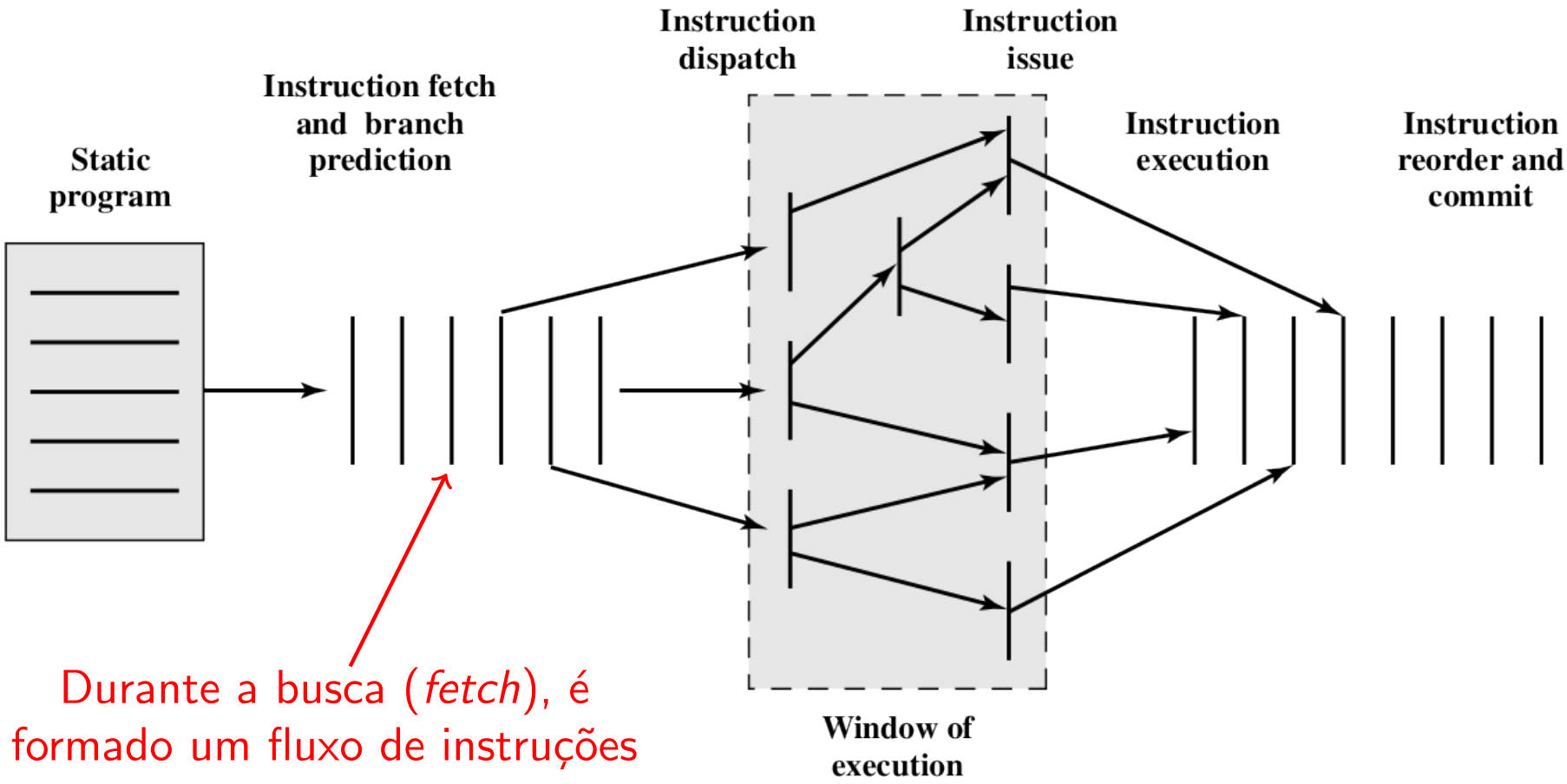
## Rodando um programa...




Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

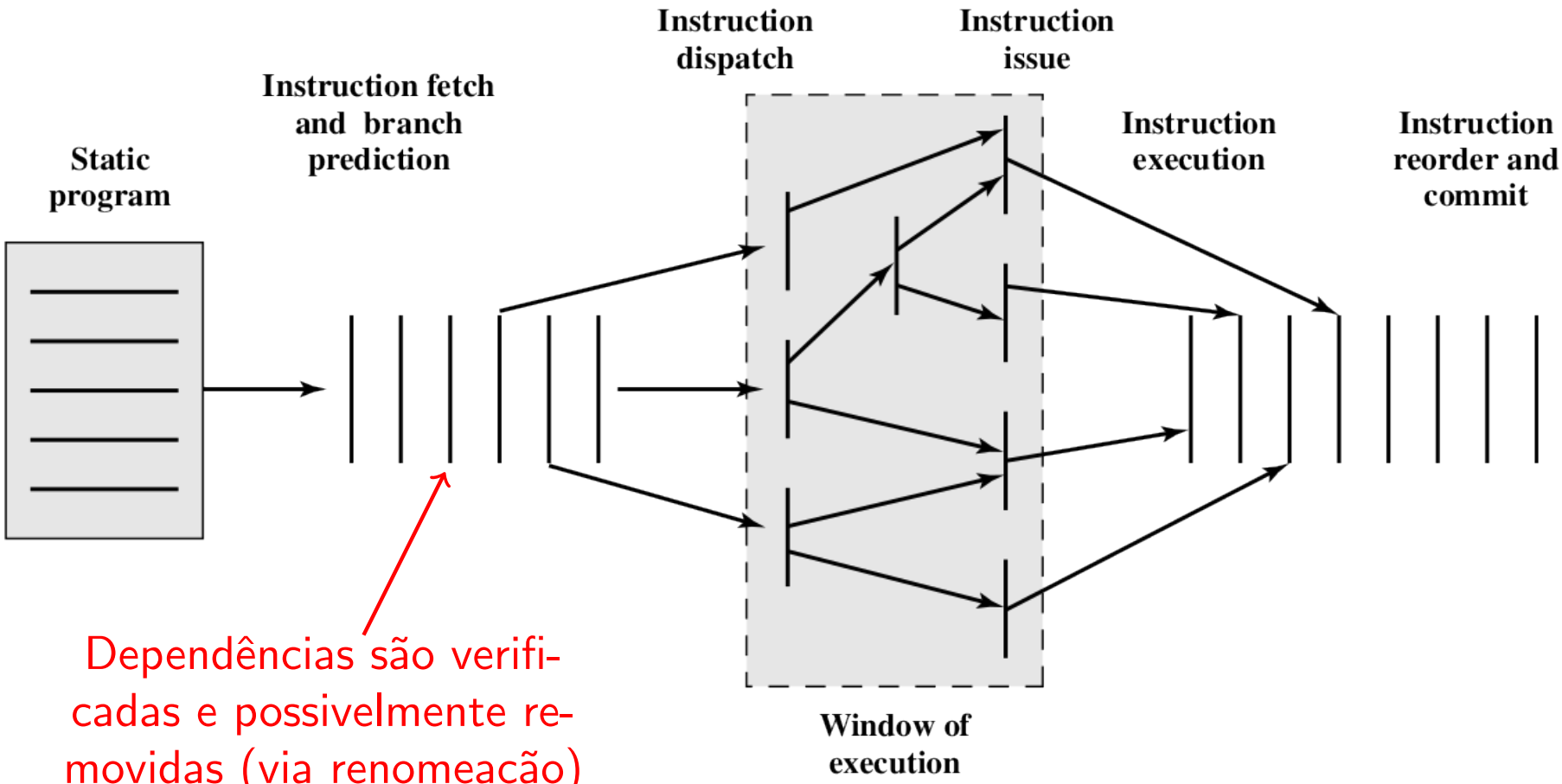



Durante a busca (*fetch*), é formado um fluxo de instruções

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

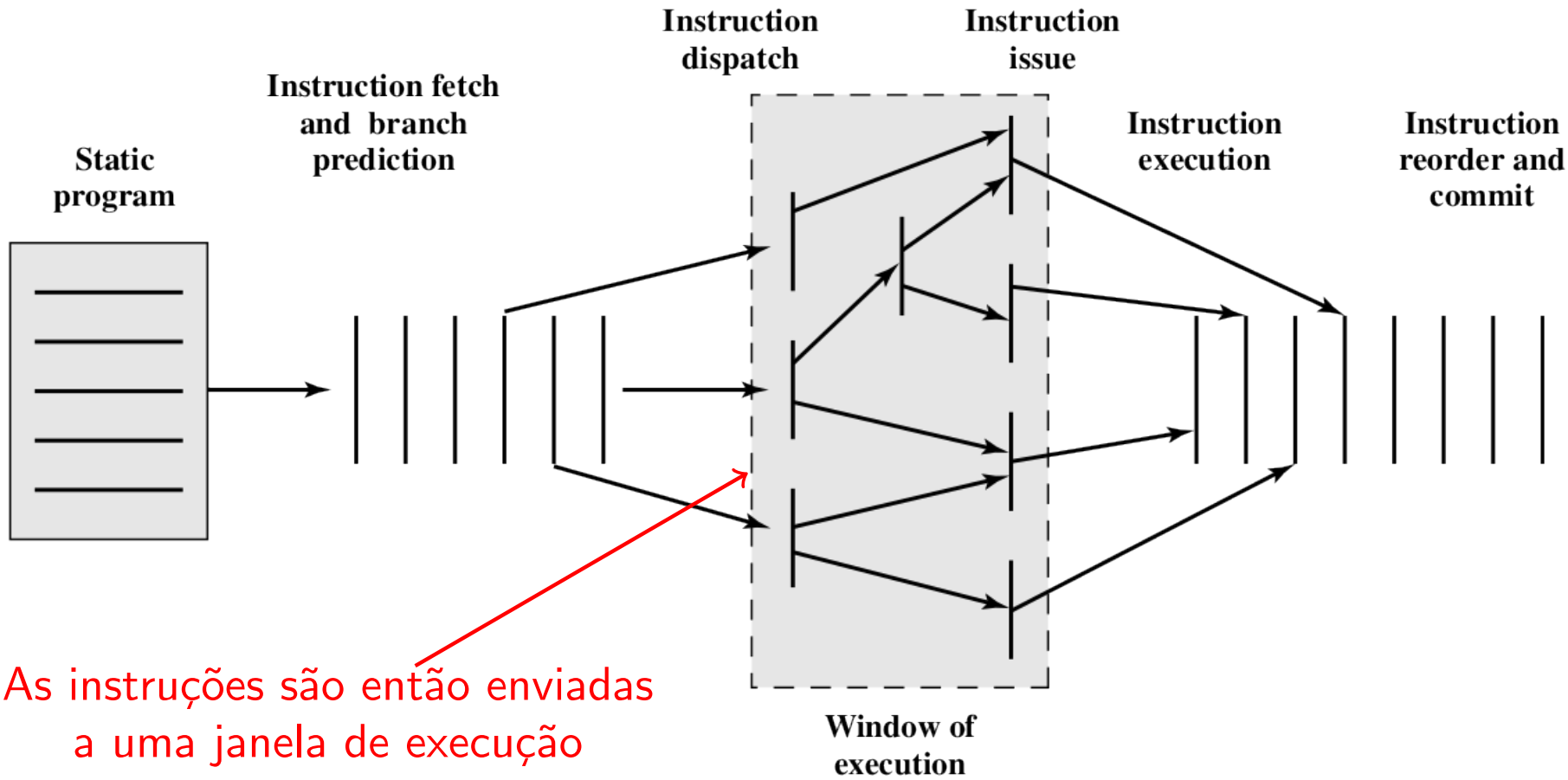
## Rodando um programa...




Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

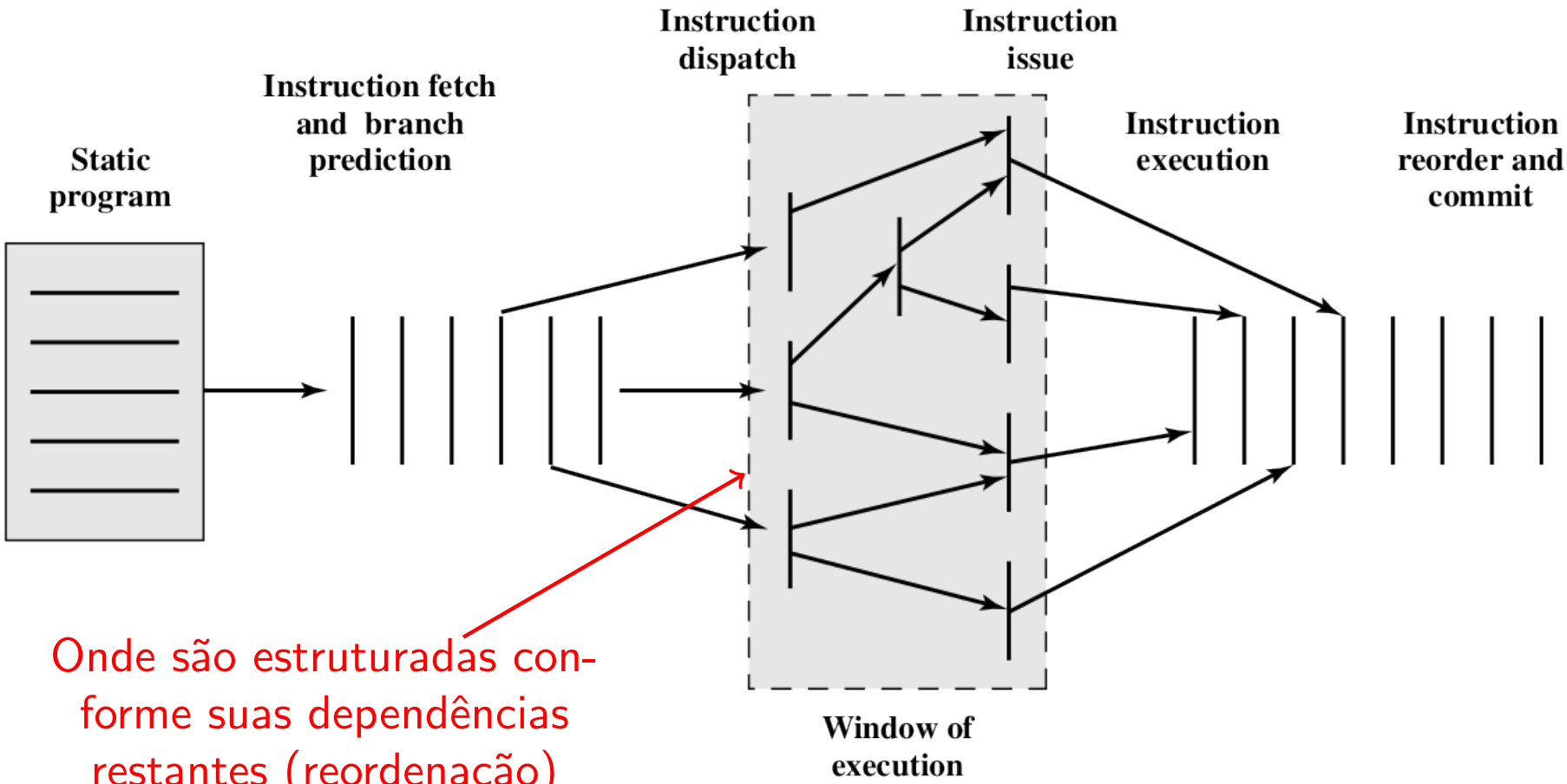



As instruções são então enviadas a uma janela de execução

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

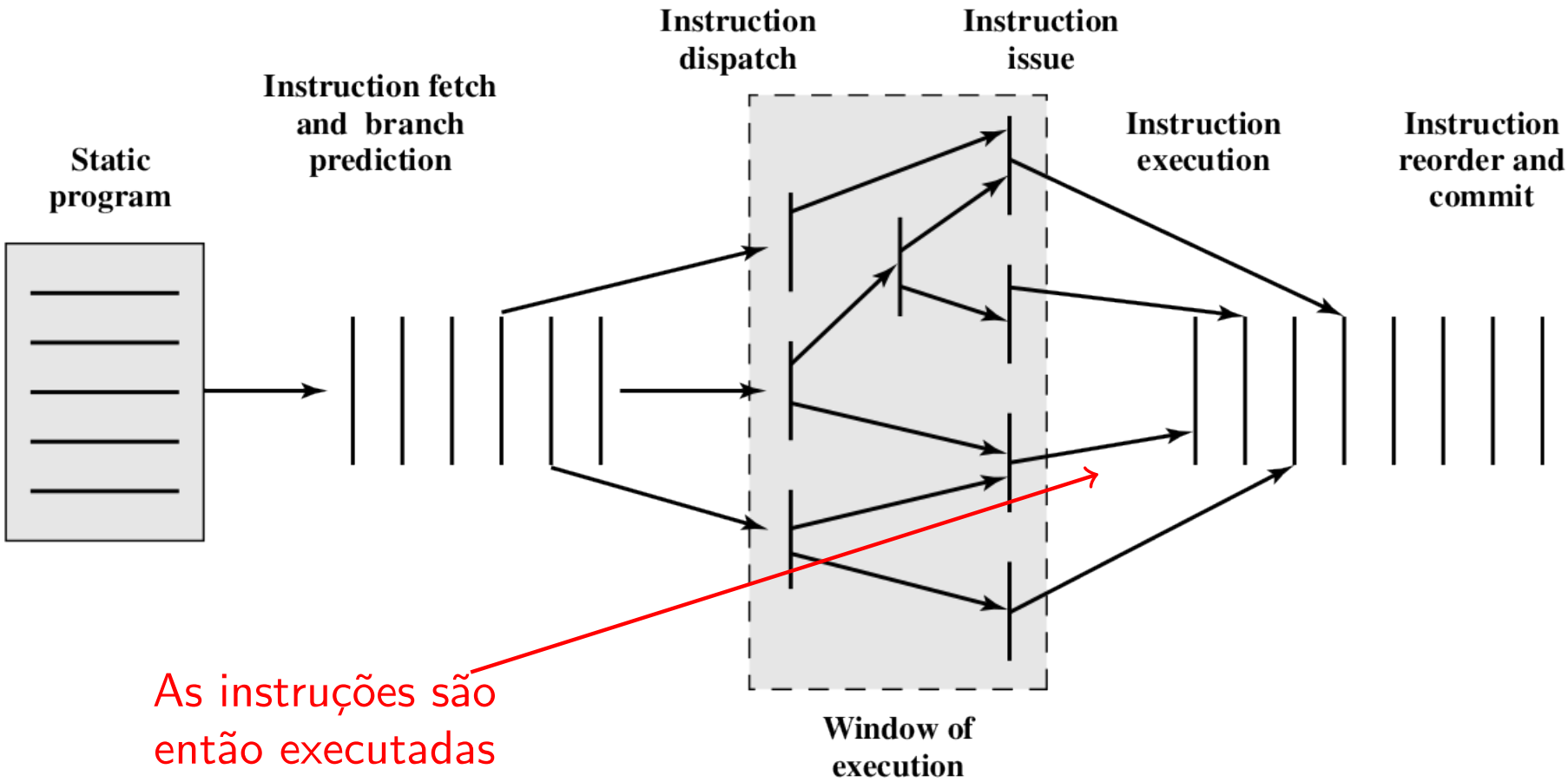



Onde são estruturadas conforme suas dependências restantes (reordenação)

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

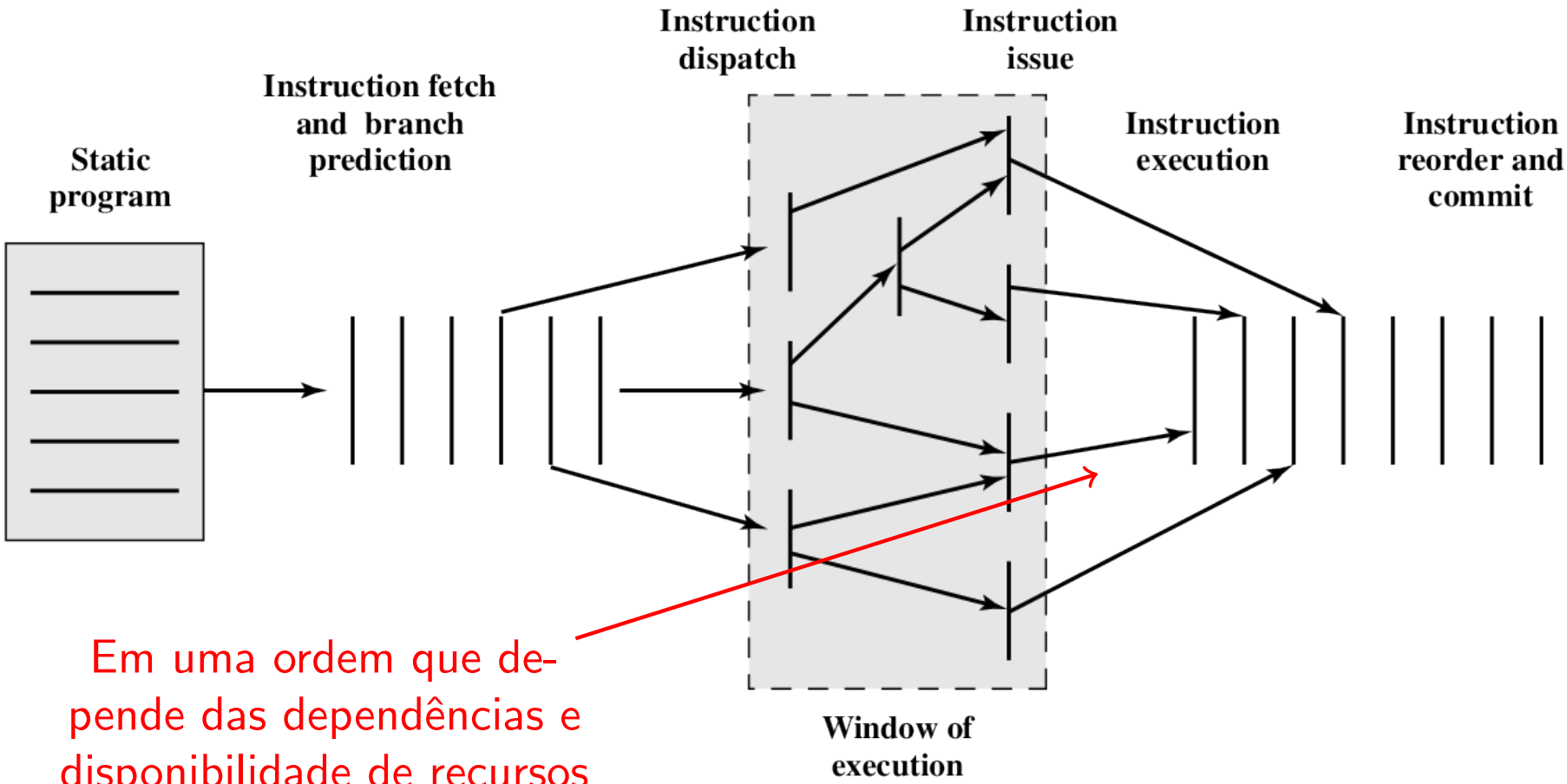
## Rodando um programa...




Fonte: [2]

# Superescalar: Escalonamento Dinâmico

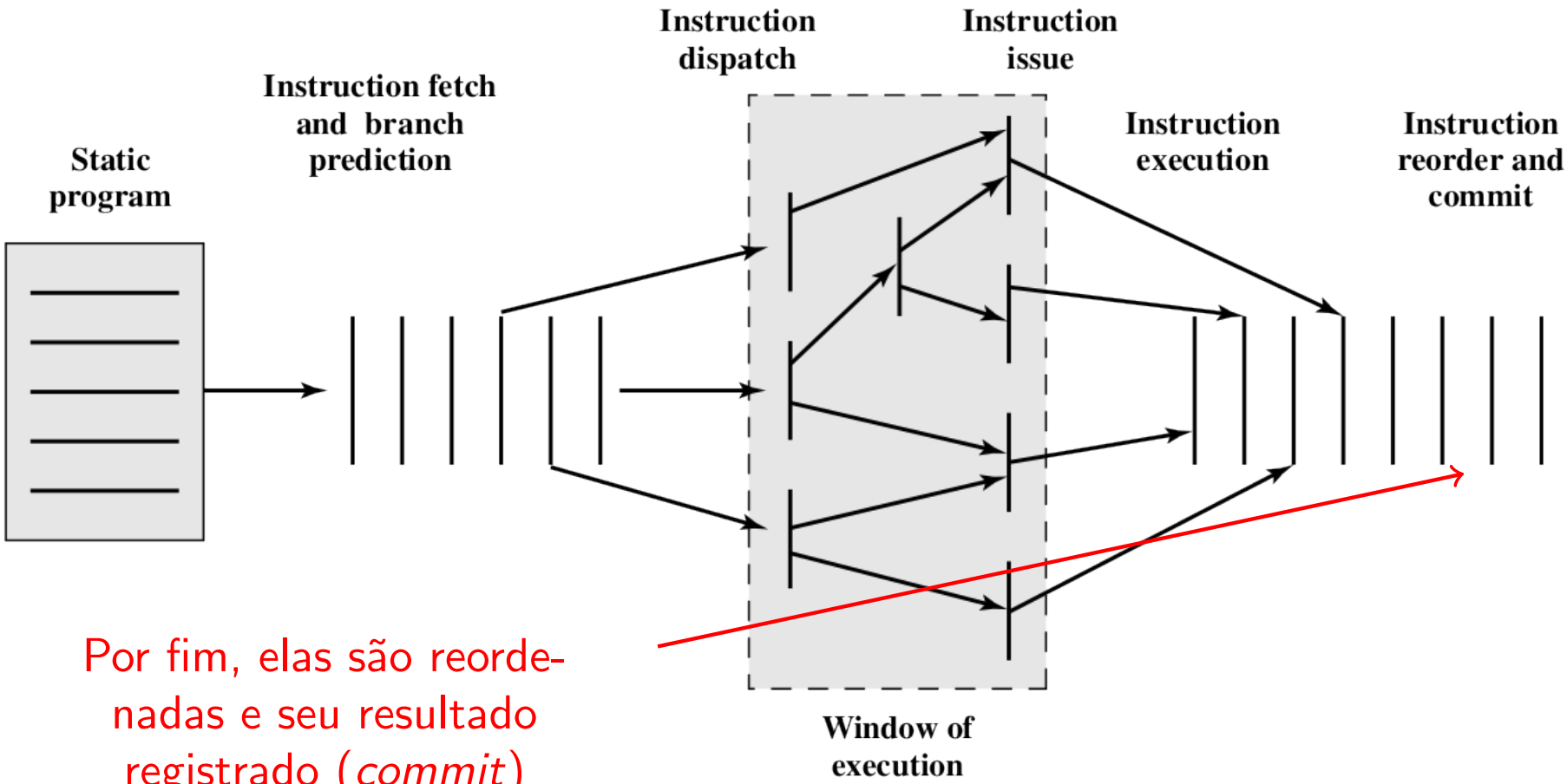
## Rodando um programa...




Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

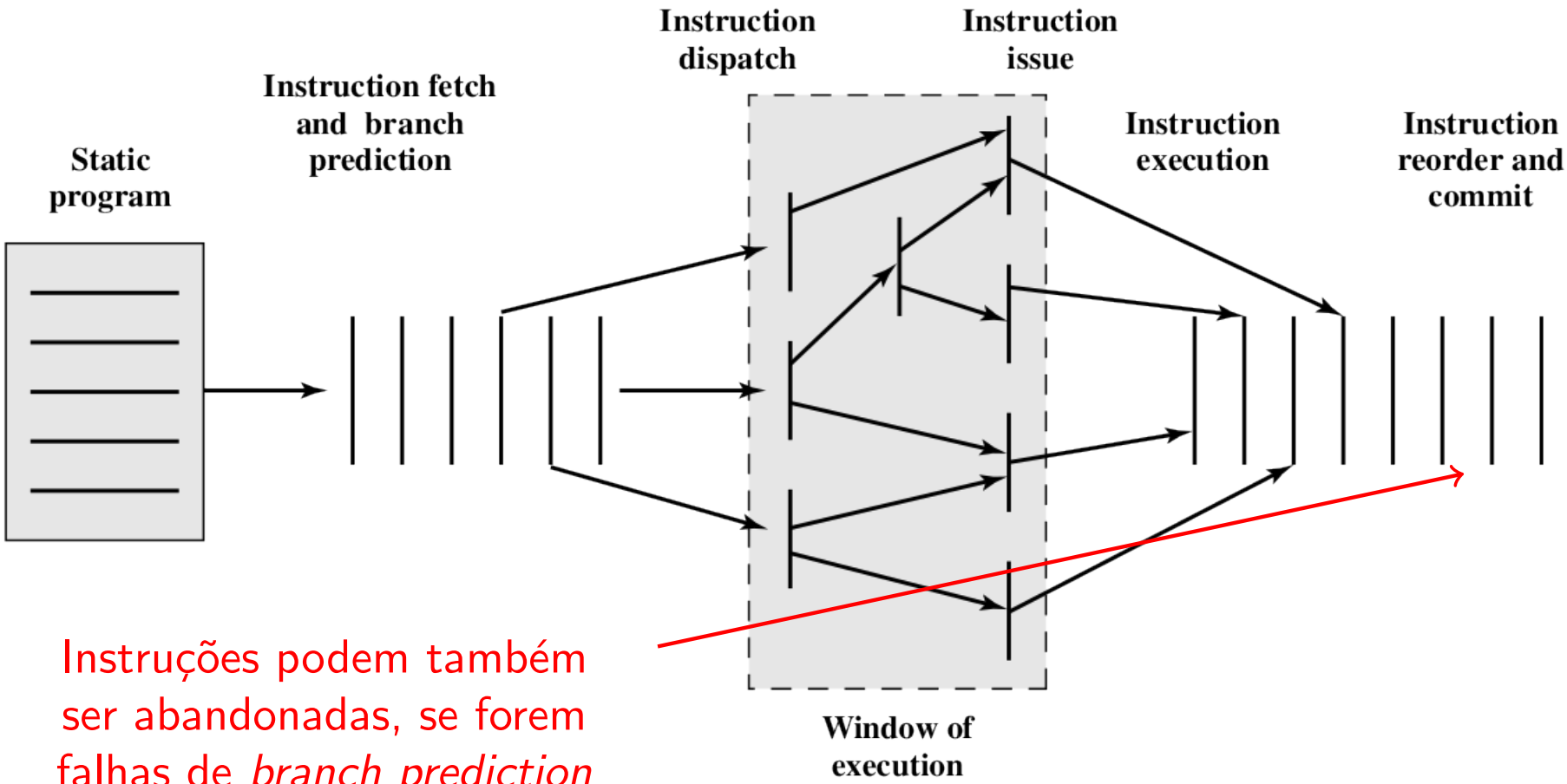



Por fim, elas são reordenadas e seu resultado registrado (*commit*)

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

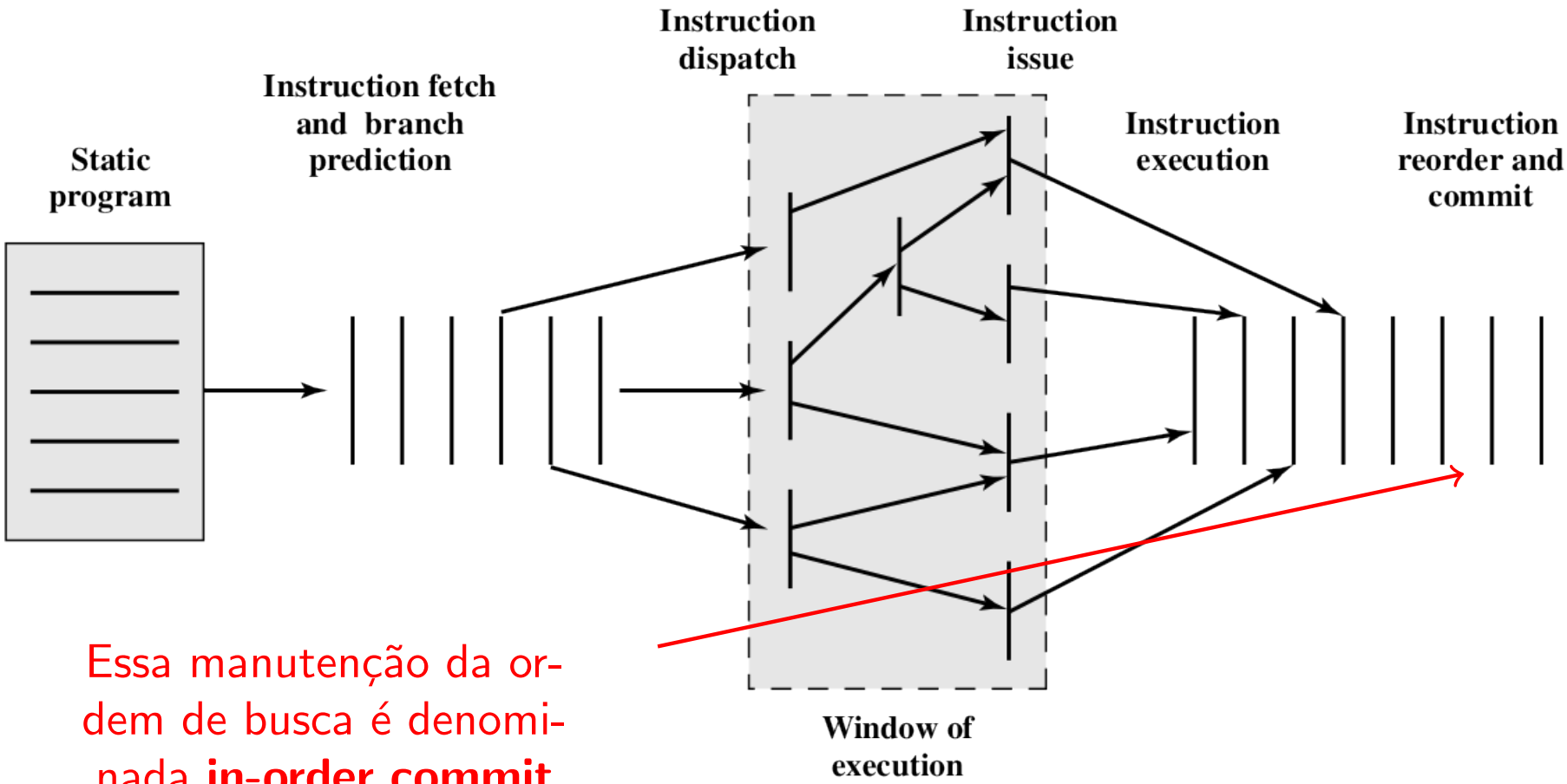



Instruções podem também ser abandonadas, se forem falhas de *branch prediction*

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

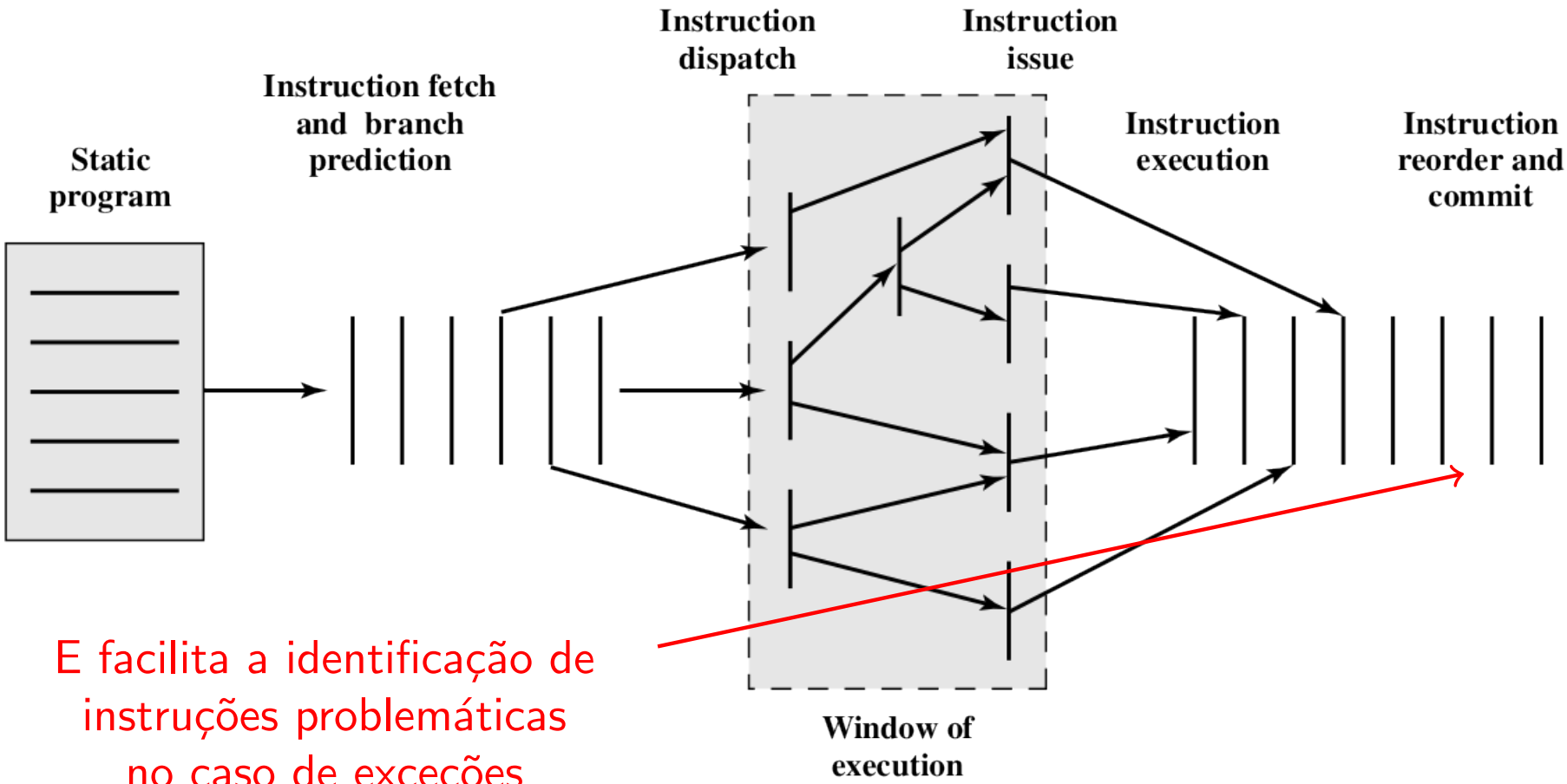



Essa manutenção da ordem de busca é denominada **in-order commit**

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

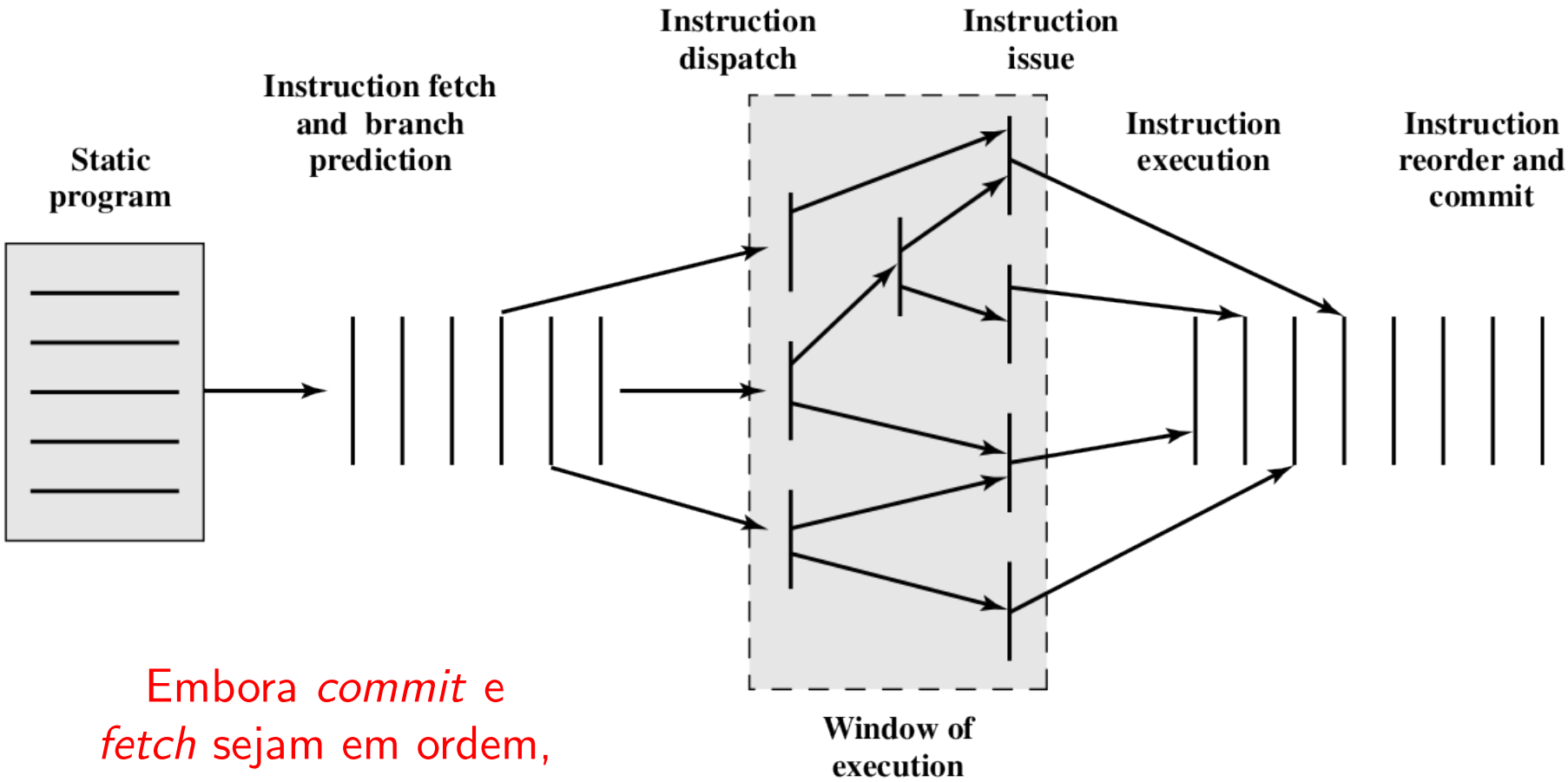



E facilita a identificação de instruções problemáticas no caso de exceções

Fonte: [2]

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...

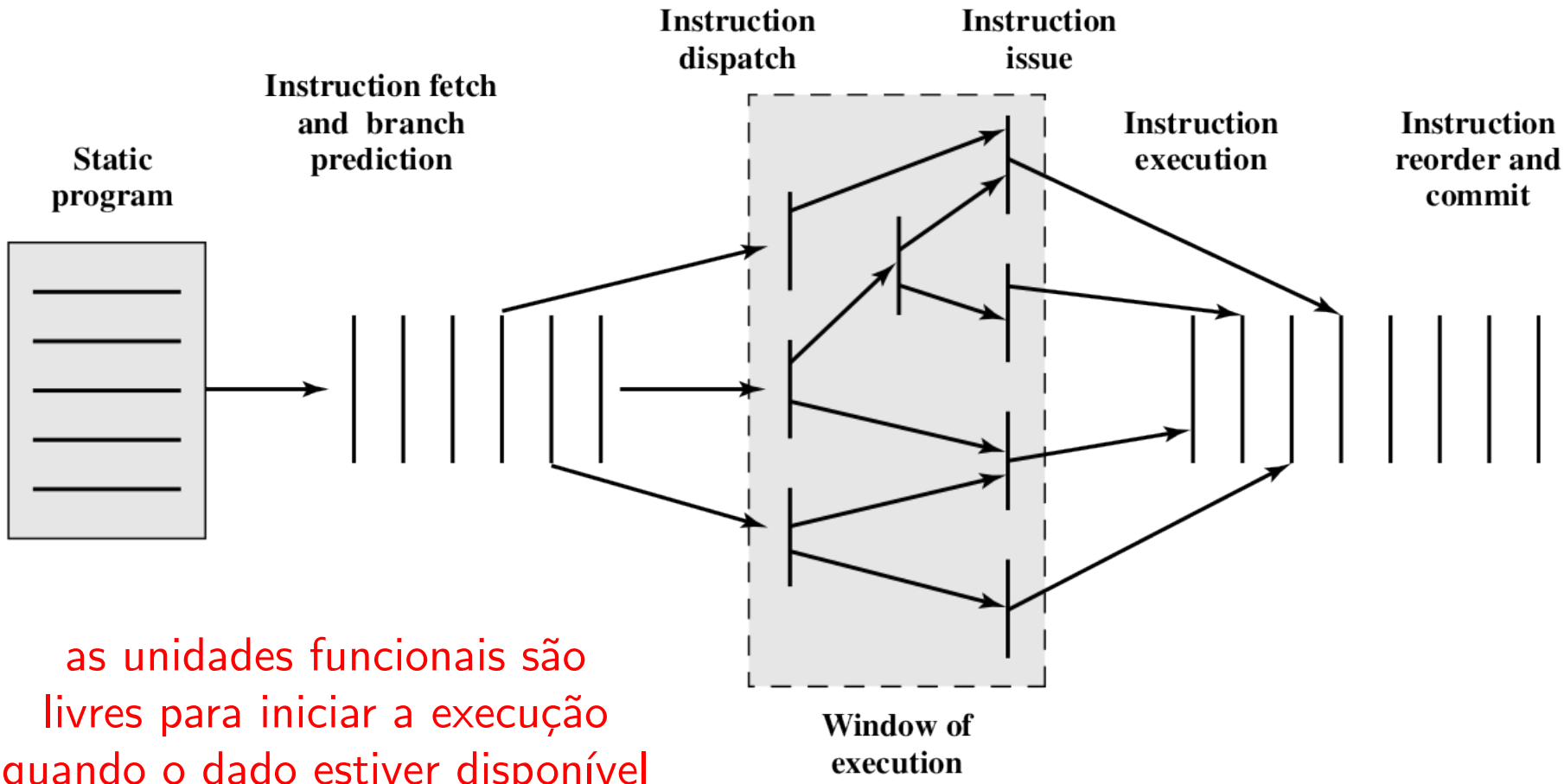



Fonte: [2]

Embora *commit* e *fetch* sejam em ordem,

# Superescalar: Escalonamento Dinâmico

## Rodando um programa...




as unidades funcionais são
livres para iniciar a execução
quando o dado estiver disponível

Fonte: [2]

# Paralelismo em Nível de Instrução

## Tempo × Espaço

- O paralelismo pode acontecer de forma temporal ou espacial

# Paralelismo em Nível de Instrução

## Tempo × Espaço

- O paralelismo pode acontecer de forma temporal ou espacial
  - *Pipelining* é um caso de paralelismo temporal
    - Não há realmente duplicação de recurso algum, mas apenas uma organização do fluxo no tempo

# Paralelismo em Nível de Instrução

## Tempo × Espaço

- O paralelismo pode acontecer de forma temporal ou espacial
  - *Pipelining* é um caso de paralelismo temporal
    - Não há realmente duplicação de recurso algum, mas apenas uma organização do fluxo no tempo
  - Múltiplas unidades de execução é um caso de paralelismo espacial

# Paralelismo em Nível de Instrução

## Tempo × Espaço

- O paralelismo pode acontecer de forma temporal ou espacial
  - *Pipelining* é um caso de paralelismo temporal
    - Não há realmente duplicação de recurso algum, mas apenas uma organização do fluxo no tempo
  - Múltiplas unidades de execução é um caso de paralelismo espacial
- Processadores superescalares exploram ambas as formas de paralelismo

# Referências


1. Patterson, D.A.; Hennessy, J.L. (2013): Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann. 5ª ed.
2. Stallings, W (2010): Computer Organization and Architecture: Designing for Performance. Prentice Hall. 8ª ed.
3. Harris, D.M.; Harris, S.L.: Digital Design and Computer Architecture. Morgan Kaufmann. 2ª ed.