

Aulas 20 a 24

- *Pipelining*
 - Definição - exemplo prático por analogia
 - Adaptação do conceito ao caso do MIPS
 - Problemas da solução *pipelined*
- Construção de um *datapath* com *pipelining*
 - Divisão em fases de execução
 - Execução das instruções
- *Pipelining hazards*
 - *Hazards* estruturais: replicação de recursos
 - *Hazards* de controlo: *stalling*, previsão, *delayed branch*
 - *Hazards* de dados: *stalling*, *forwarding*
- *Datapath* para o MIPS com unidades simplificadas de *forwarding* e *stalling*

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Introdução

- **Pipelining** é uma técnica de implementação de arquiteturas do set de instruções (ISA), através da qual múltiplas instruções são executadas com algum grau de **sobreposição temporal**
- O objetivo é aproveitar, de forma o mais eficiente possível, os recursos disponibilizados pelo *datapath*, por forma a **maximizar a eficiência global do processador**

Pipelining - exemplo por analogia

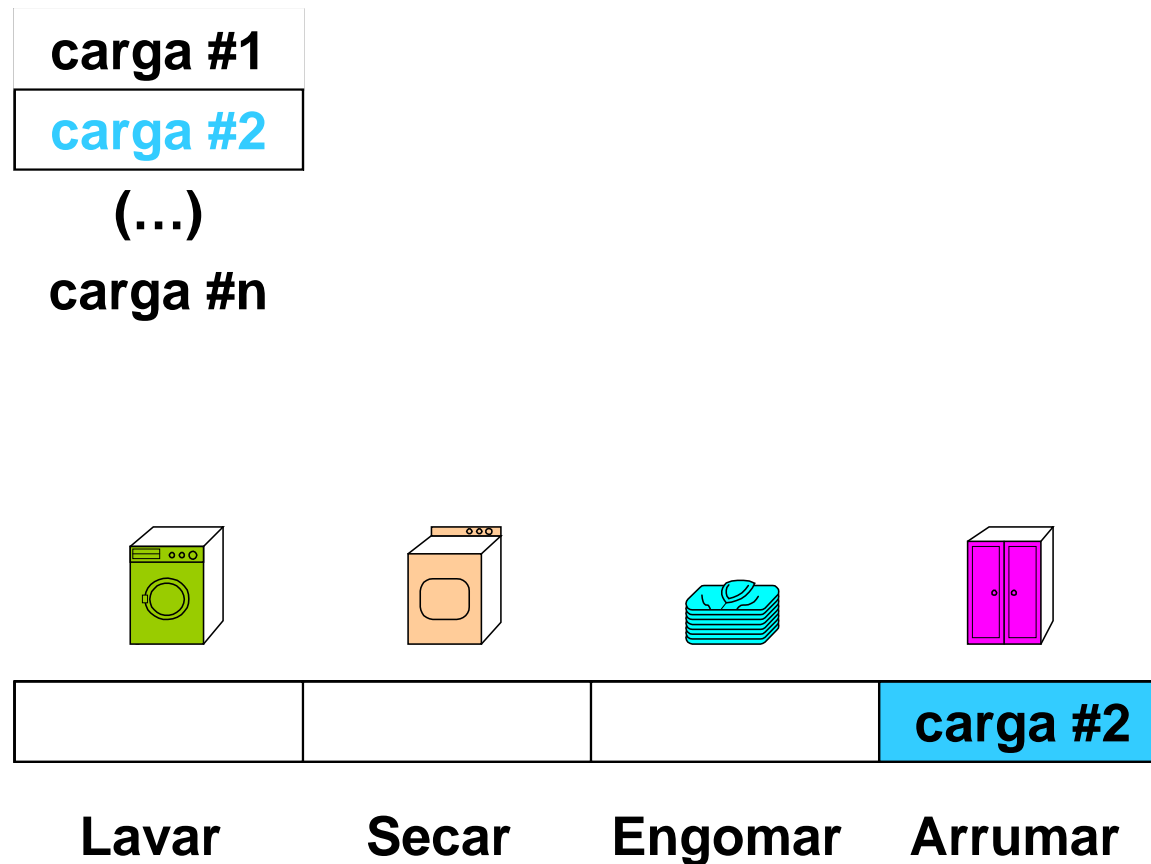
- O exemplo de *pipelining* que iremos observar de seguida apoia-se num conjunto de tarefas simples e intuitivas: o processo de tratamento da roupa suja 😊



- Neste exemplo, o tratamento da roupa suja desencadeia-se nas seguintes quatro fases:
 1. Lavar uma carga de roupa na máquina respetiva
 2. Secar a roupa lavada na máquina de secar
 3. Passar a ferro e dobrar a roupa
 4. Arrumar a roupa dobrada no guarda roupa respetivo

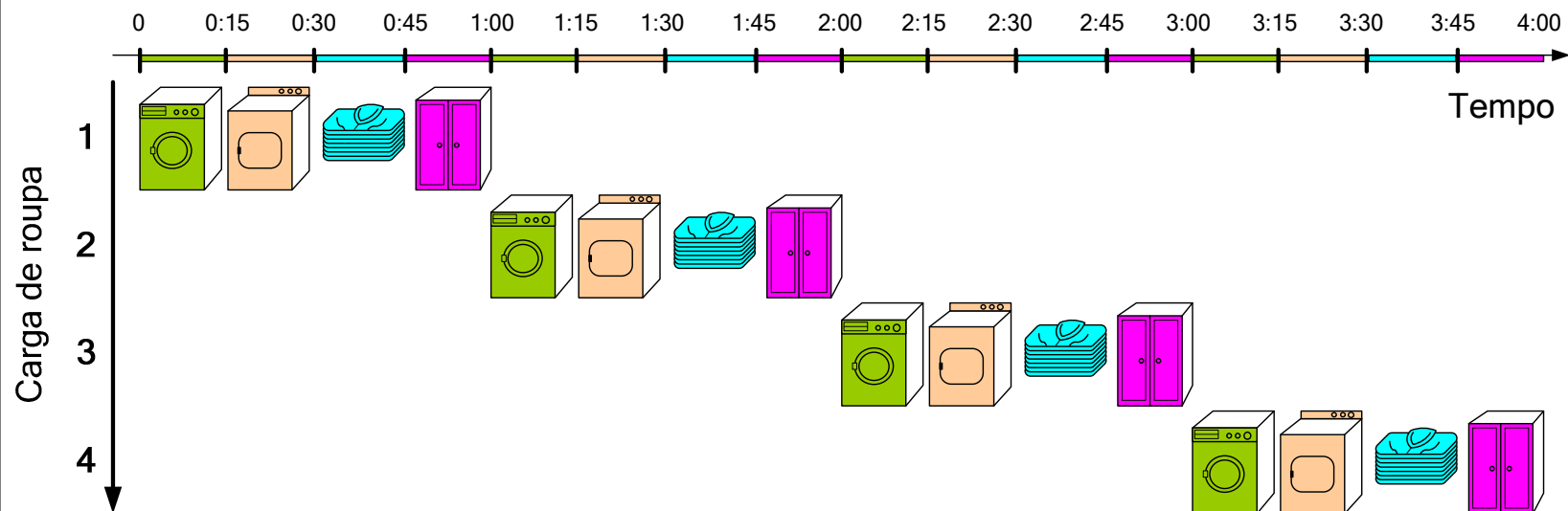
Pipelining - exemplo por analogia

- Numa versão não *pipelined*, o processamento de N cargas de roupa seria:



Pipelining - exemplo por analogia

- Este processo pode então ser descrito temporalmente do seguinte modo:



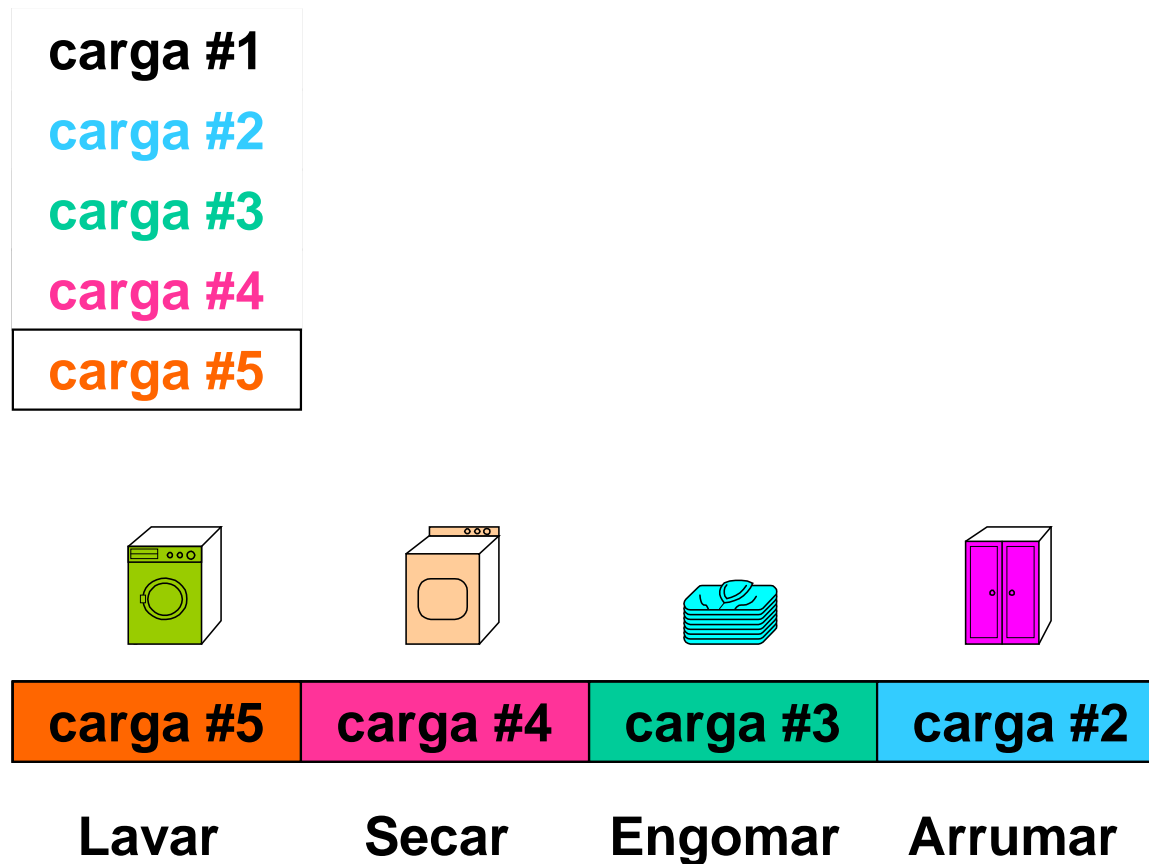
Se o tempo para tratar uma carga de roupa for uma hora, tratar quatro cargas demorará **quatro horas**.

Pipelining - exemplo por analogia

- Na versão *pipelined*, aproveita-se para carregar uma nova carga de roupa na máquina de lavar mal esteja concluída a lavagem da primeira carga
- O mesmo princípio se aplica a cada uma das restantes três tarefas
- Quando se inicia a arrumação da primeira carga, todos os passos (chamados **estágios** ou **fases** em *pipelining*) estão a funcionar em paralelo
- Maximiza-se assim a utilização dos recursos disponíveis

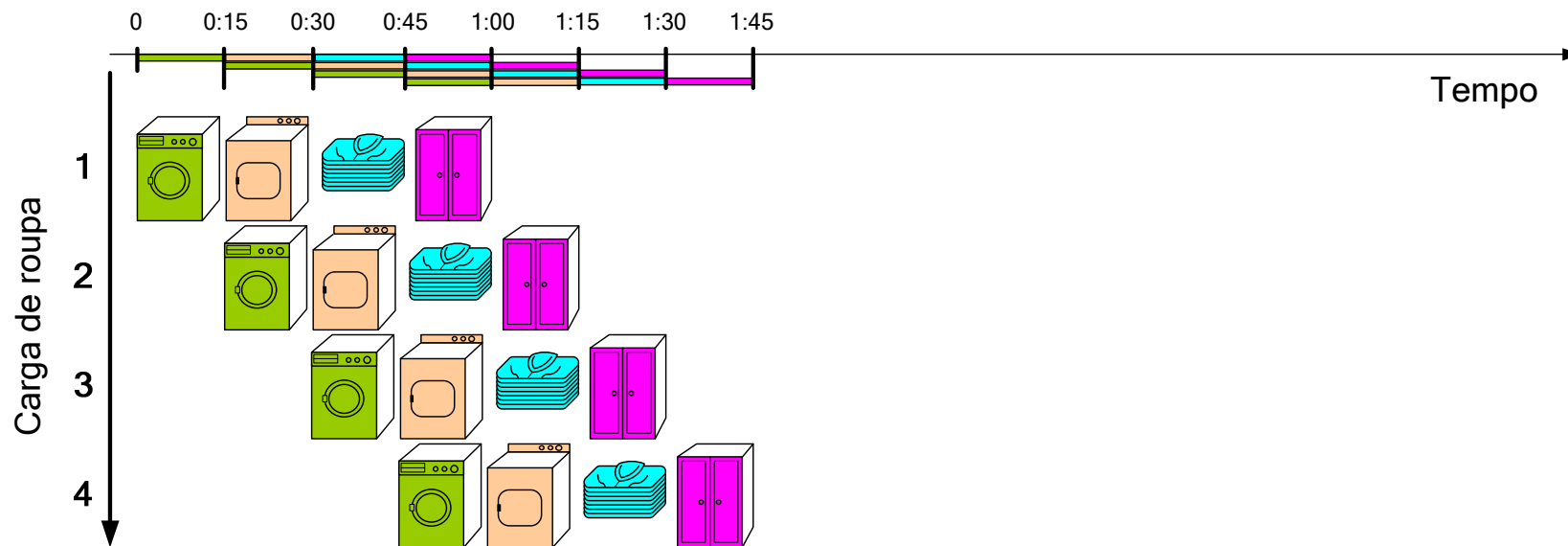
Pipelining - exemplo por analogia

- Na versão *pipelined*, o processamento das cargas de roupa seria (admitindo tempo nulo entre a comutação de tarefas):



Pipelining - exemplo por analogia

- O processo de tratamento da versão *pipelined* pode então ser descrito temporalmente do seguinte modo:



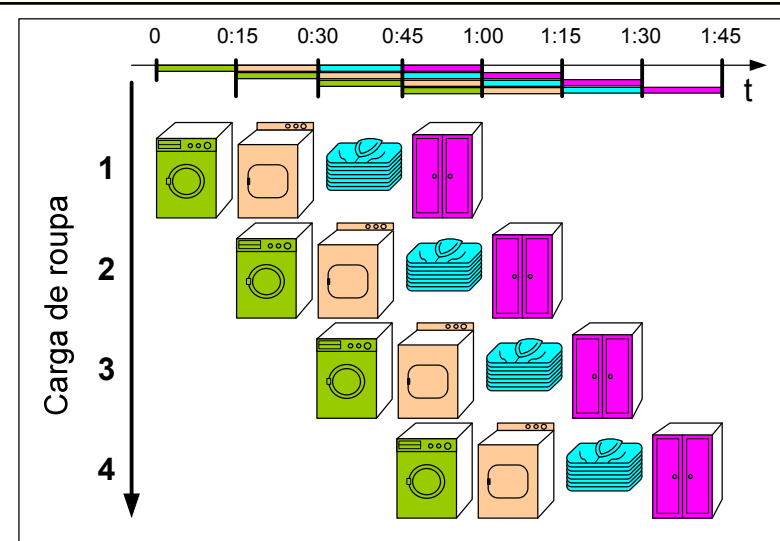
Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45. Ou seja 135 minutos menos ($240 - 105$).

Pipelining - exemplo por analogia

- O paradoxo aparente da solução *pipelined* é que o tempo necessário para o processamento completo de uma carga de roupa não difere do tempo de execução da solução não *pipelined*
- A eficiência da solução com *pipelining* decorre do facto de, para um número grande de cargas de roupa, todos os passos intermédios estarem a executar em paralelo
- O resultado é o aumento do número total de cargas de roupa processadas por unidade de tempo (**throughput**)
- Qual o **ganho de desempenho** que se obtém com o sistema *pipelined* relativamente ao sistema normal?

Pipelining – ganho de desempenho

- O tratamento de N cargas de roupa num sistema com F fases demorará idealmente (admitindo que cada fase demora 1 unidade de tempo):



Sistema não *pipelined*: $T_{\text{NON-PIPELINE}} = N \times F$

Sistema *pipelined*: $T_{\text{PIPELINE}} = F + (N - 1) = (F - 1) + N$

Ganho de desempenho
obtido com a solução
pipelined:

$$\frac{\text{Desempenho}_{\text{PIPELINE}}}{\text{Desempenho}_{\text{NON-PIPELINE}}} = \frac{T_{\text{NON-PIPELINE}}}{T_{\text{PIPELINE}}} = \frac{N \times F}{(F - 1) + N}$$

Se $N \gg (F - 1)$, então: $\text{Ganho} \approx \frac{N \times F}{N} = F$

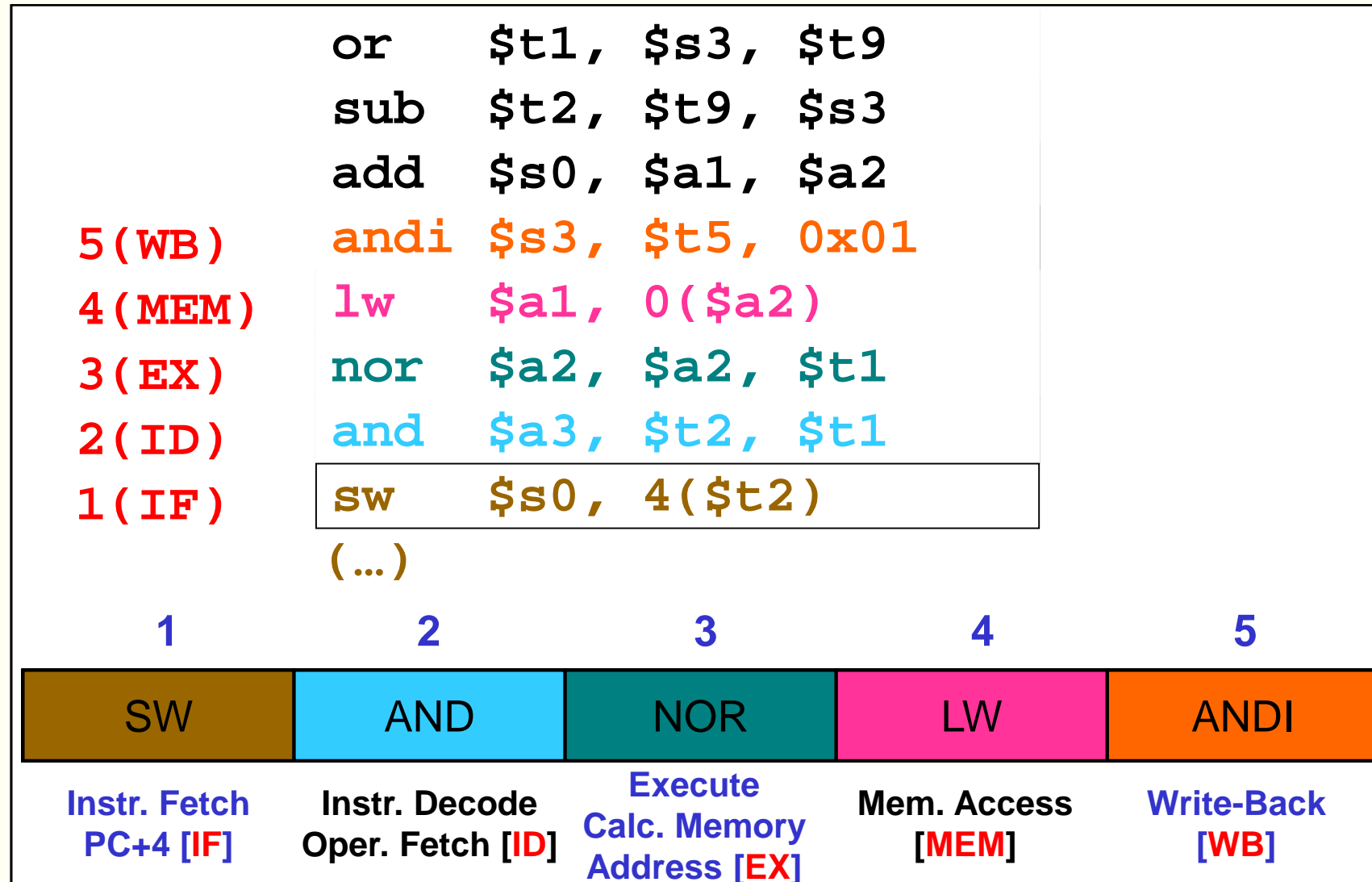
Pipelining – ganho de desempenho

- No limite, para um número de cargas de roupa muito elevado, o ganho de desempenho (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e noutro modelo) é da ordem do **número de tarefas realizadas em paralelo** (isto é, igual ao número de fases do processo)
- Genericamente, poderíamos afirmar que o ganho em velocidade de execução é igual ao número de estágios do *pipeline* (F)
- No exemplo observado, o limite teórico estabelece que a solução *pipelined* é quatro vezes mais rápida do que a solução não *pipelined*
- A adoção de *pipelines* muito longos (com muitos estágios) pode, contudo, limitar drasticamente a eficiência global

Pipelining no MIPS

- Os mesmos princípios observados no exemplo do tratamento da roupa, podem igualmente ser aplicados aos processadores
- Para o MIPS, como já analisado, a execução das instruções pode ser dividida genericamente em **cinco fases** (estágios, etapas):
 1. **Instruction fetch** (ler a instrução da memória), incremento do PC
 2. **Operand fetch** (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
 3. **Execute** (executar a operação ou calcular um endereço)
 4. **Memory access** (aceder à memória de dados para leitura ou escrita)
 5. **Write-Back** (escrever o resultado no registo destino)
- Parece assim razoável admitir a construção de uma solução *pipelined* do *datapath* do MIPS que implemente cinco estágios distintos, um para cada fase da execução das instruções

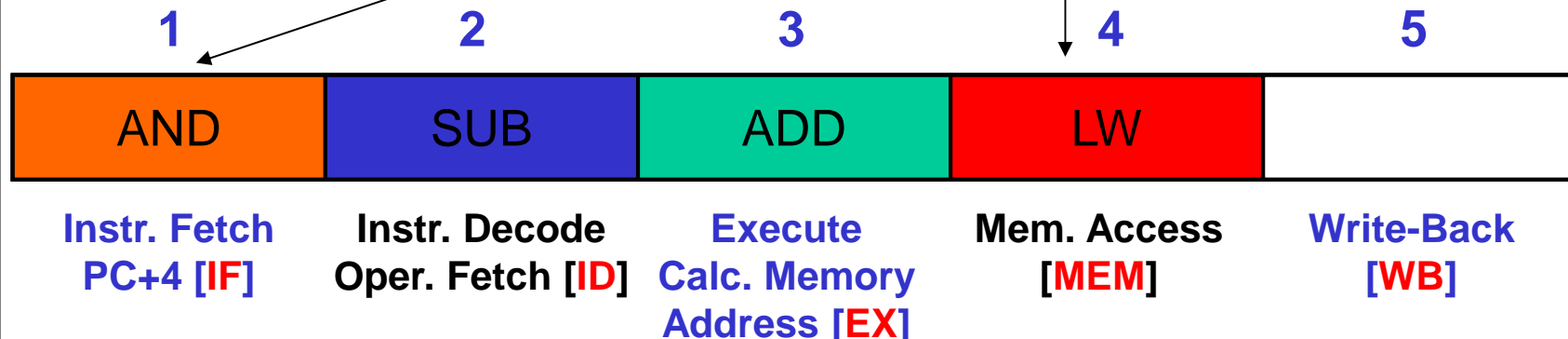
Pipelining no MIPS



Pipelining – Problemas (exemplo 1)

```
lw    $t1, 0($t9)
add   $t2, $t3, $t4
sub   $t3, $t4, $t5
and   $t4, $t5, $t6
lw    $t5, 16($t9)
```

- No quarto estágio da primeira instrução e no primeiro da quarta instrução é necessário efetuar, simultaneamente, um acesso à memória para **leitura de dados** e para o **instruction fetch**
- Se existir apenas uma memória para dados e código, as duas operações não podem ser executadas em paralelo

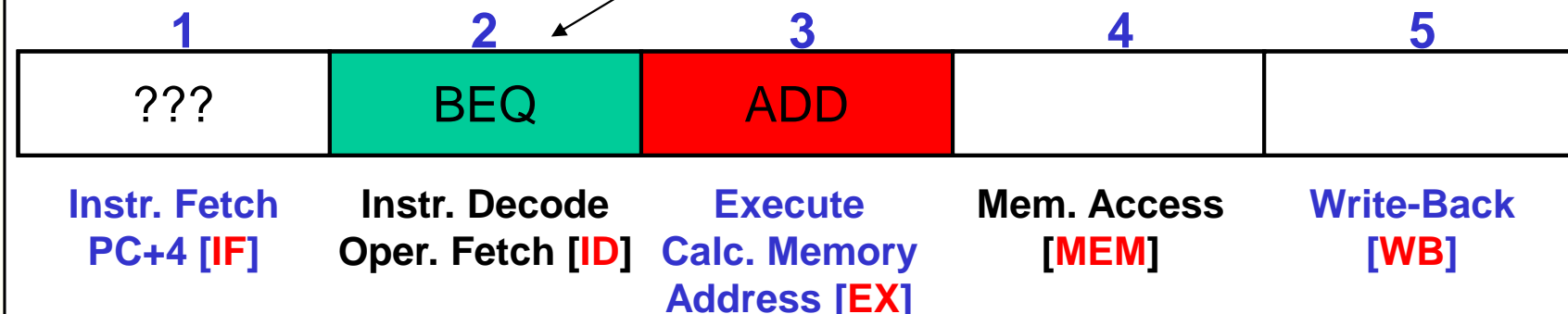


Pipelining – Problemas (exemplo 2)

```
add    $t4, $t5, $t6
beq     $t1, $t2, z1
lw      $s3, 300($a0)
(...)
```

```
z1: or   $t7, $t8, $t9
```

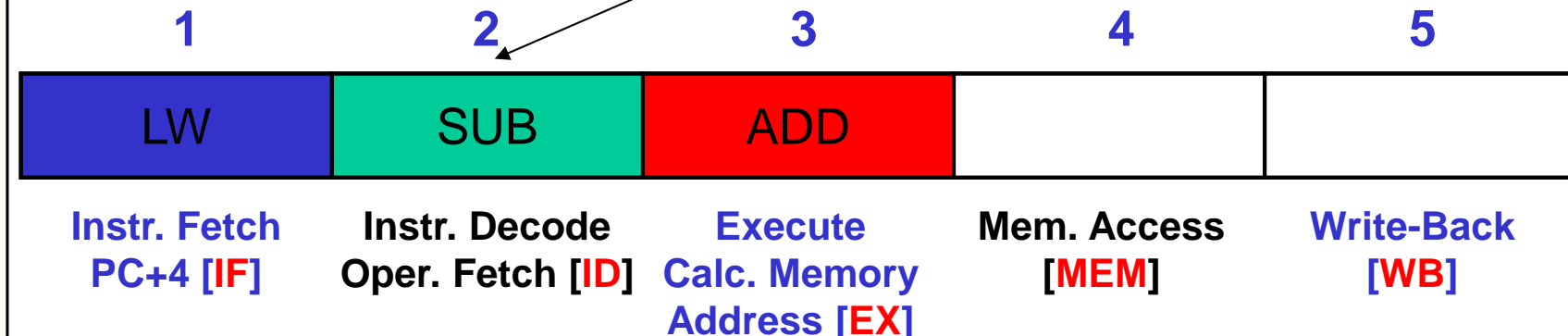
- Qual a próxima instrução a ler da memória (**LW** / **OR**) assumindo que a decisão do *branch* só é tomada no 3º estágio do *pipeline*?
- Mesmo admitindo que existe h/w dedicado para avaliar a condição do *branch* logo no 2º estágio, a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória.



Pipelining – Problemas (exemplo 3)

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
lw   $t4, 0($s2)
```

A instrução de subtração (SUB) não pode avançar para o estágio seguinte (EX) uma vez que o seu operando, **\$s0**, ainda não foi calculado e armazenado no registo destino pela instrução anterior.



Datapath pipelined para o MIPS

- Nos slides seguintes vamos construir e analisar um *datapath pipeline* que suporte a execução das instruções do MIPS que já considerámos anteriormente, isto é:
 - Acesso à memória: **lw** (load word) e **sw** (store word)
 - Tipo R: **add**, **sub**, **and**, **or** e **slt**
 - Imediatas: **addi** e **slti**
 - Alteração do fluxo de execução: **beq** e **j**
- Na comparação dos tempos de execução destas instruções num *DP single cycle* e num *DP pipelined*, tomamos como referência os seguintes tempos de execução de cada uma das fases:

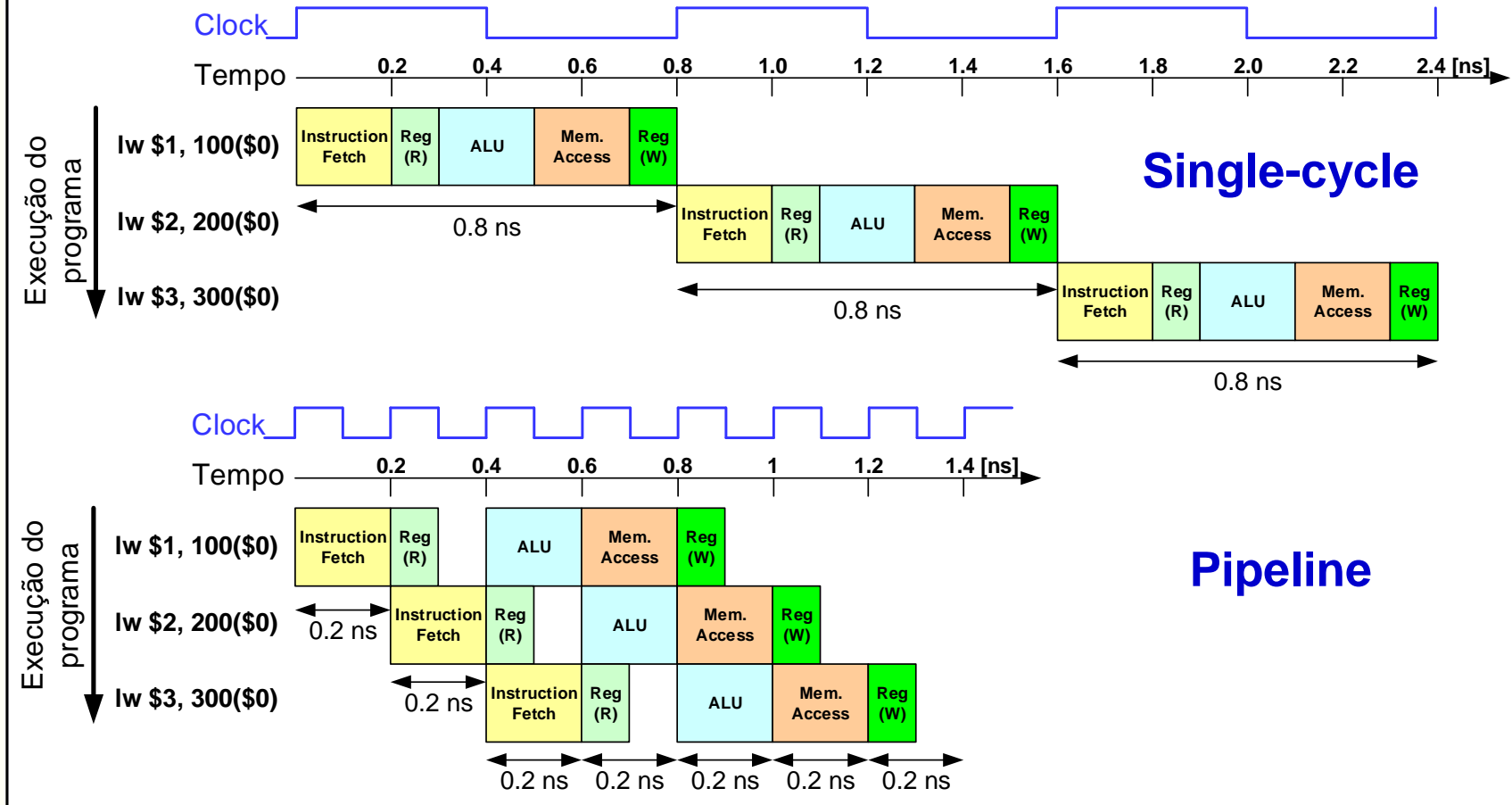
| Instruction | Instruction Fetch | Register Read | ALU Operation | Memory Access | Register Write | Tempo total |
|---------------------------------|-------------------|---------------|---------------|---------------|----------------|-------------|
| Load word (lw) | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| Store word (sw) | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-Type (add, sub, and, or, slt) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| Branch (beq) | 200 ps | 100 ps | 200 ps | | | 500 ps |
| Immediate (addi, slti) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |

Datapath pipelined para o MIPS

- Num *datapath single cycle* o período do sinal de relógio terá que ser ajustado de modo a permitir a execução da instrução mais lenta (lw)
- Na solução *single cycle* o período de relógio deve então ser, no mínimo, 800 ps, ou seja, todas as instruções, independentemente do tempo mínimo que poderiam durar, serão executadas num tempo de 800 ps
- Num *datapath pipelined*, embora alguns estágios pudessem executar em menos tempo, o período do relógio tem que ser ajustado para o atraso de propagação do elemento operativo mais lento, 200 ps no exemplo

Datapath pipelined para o MIPS

- Exemplo de execução de 3 instruções LW nos *datapaths* *single-cycle* e *pipelined*



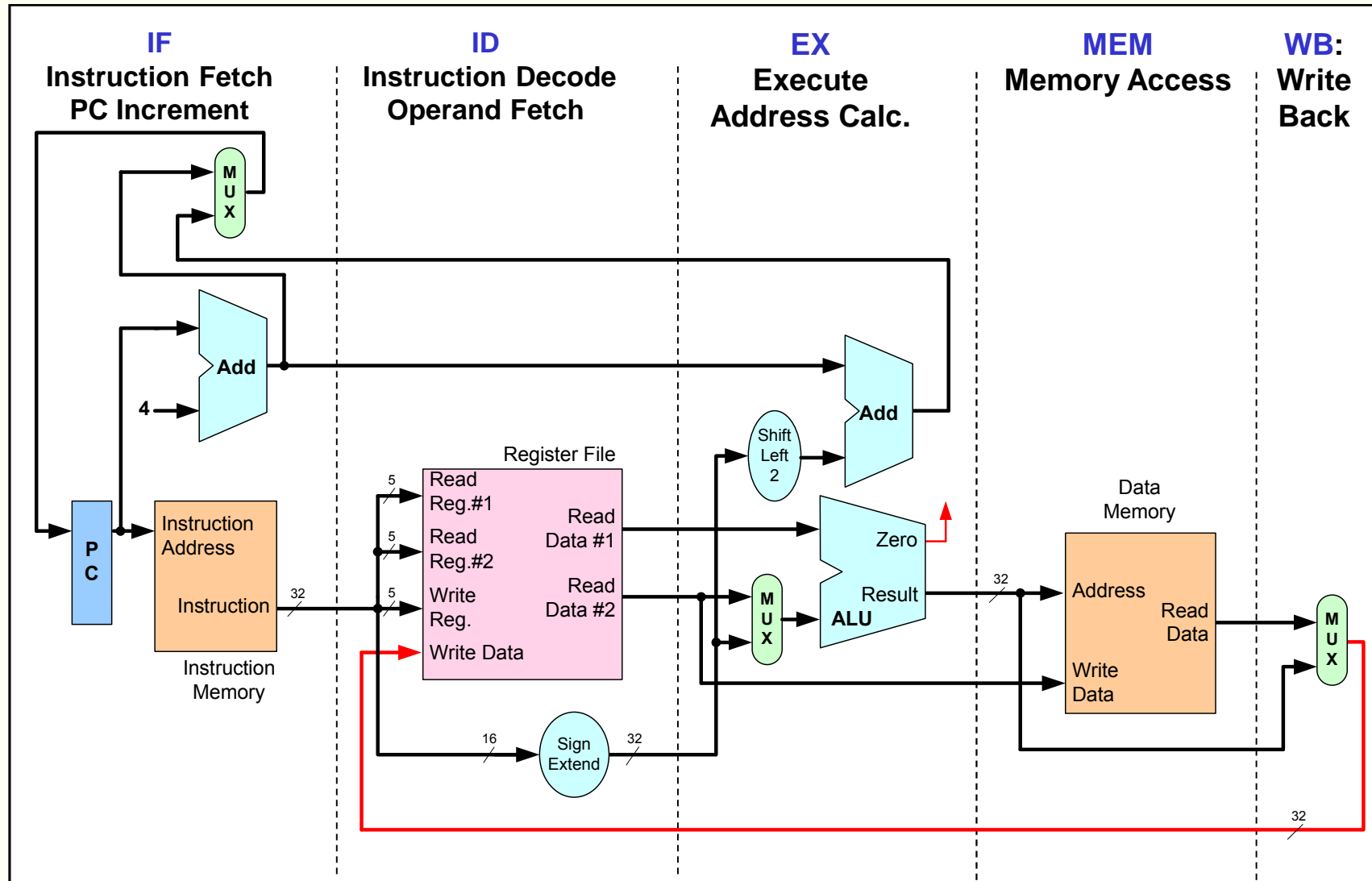
Datapath pipelined para o MIPS

- O *instruction set* do **MIPS** (*Microprocessor without Interlocked Pipeline Stages*) foi concebido para uma implementação em *pipeline*. Os aspetos fundamentais a considerar são:
 - **Instruções de comprimento fixo**: *Instruction Fetch* e *Instruction Decode* podem ser feitos em estágios sucessivos (a unidade de controlo não tem que ter em consideração a dimensão da instrução descodificada)
 - **Poucos formatos de instrução**, com a referência aos registos a ler sempre no mesmo campo (isso permite que os registos sejam lidos no segundo estágio ao mesmo tempo que a instrução é descodificada pela unidade de controlo)
 - **Referências à memória só aparecem em instruções de load/store**: o terceiro estágio pode ser usado para executar a instrução ou para calcular o endereço de memória, permitindo o acesso à memória no estágio seguinte
 - Os **operandos em memória têm que estar alinhados**: qualquer operação de leitura/escrita da memória pode ser feita num único estágio

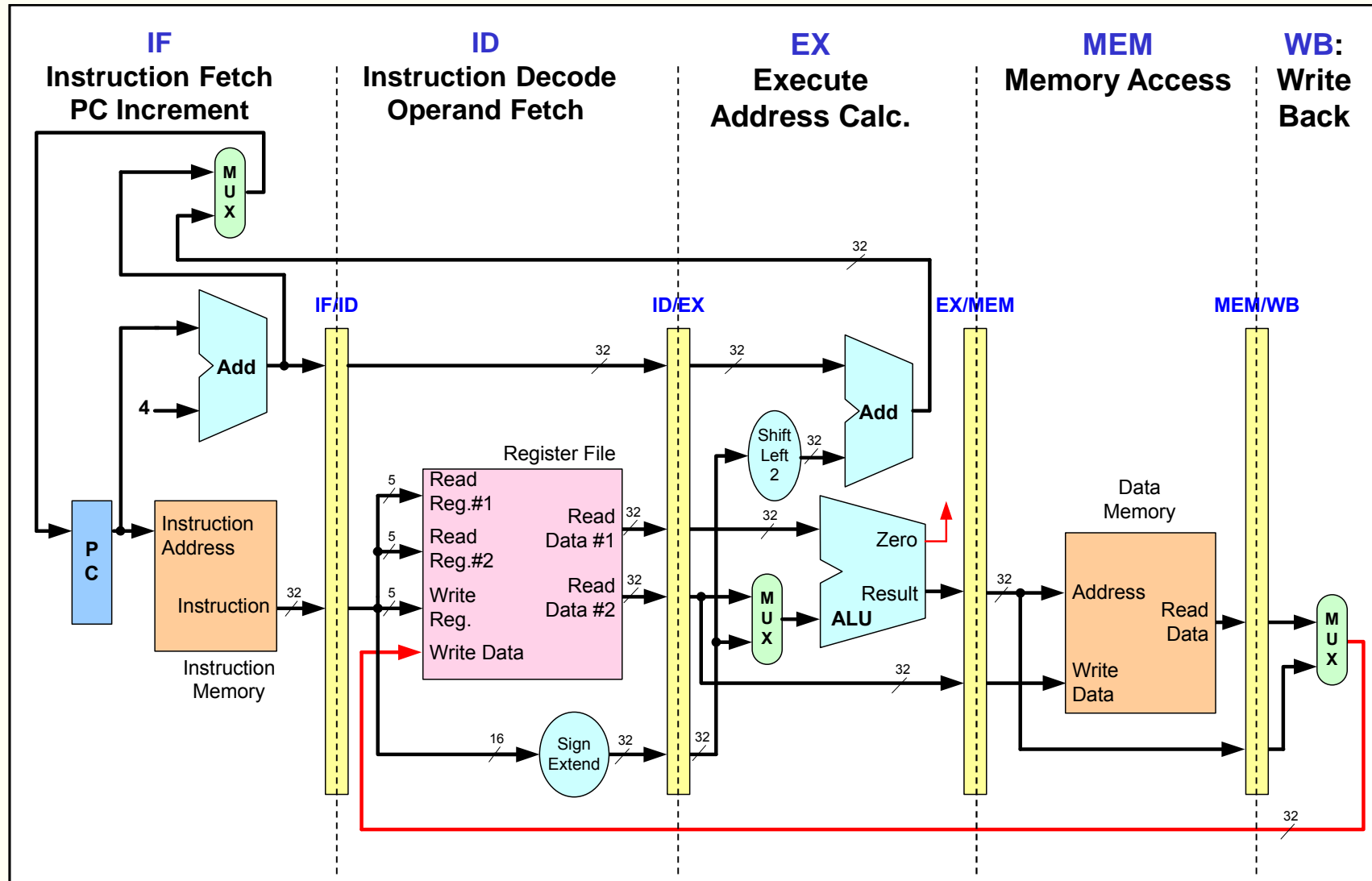
Datapath pipelined para o MIPS

- A solução *pipelined* para o MIPS parte do modelo do *datapath single-cycle*
- A organização implementa as cinco fases sequenciais em que são decomponíveis as instruções:
 1. (**IF**) - *Instruction fetch* (ler a instrução da memória), incremento do PC
 2. (**ID**) - *Operand fetch* (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
 3. (**EX**) - Executar a operação ou calcular um endereço
 4. (**MEM**) - *Memory access* (aceder à memória de dados para leitura ou escrita)
 5. (**WB**) - *Write-back* (escrever o resultado no registo destino)
- Na solução apresentada no slide seguinte não são identificados os sinais de controlo nem a respetiva unidade de controlo

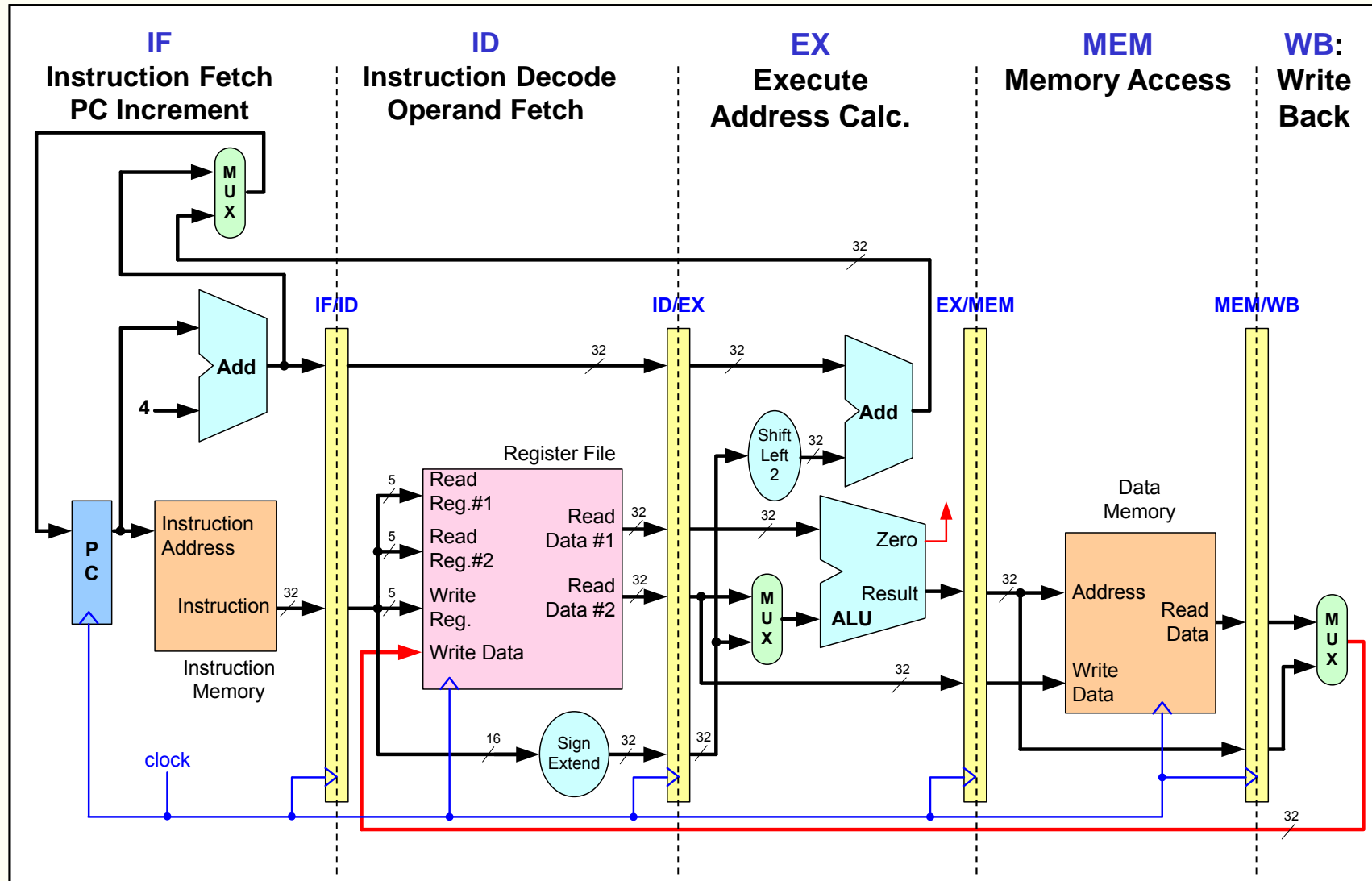
Divisão em fases de execução



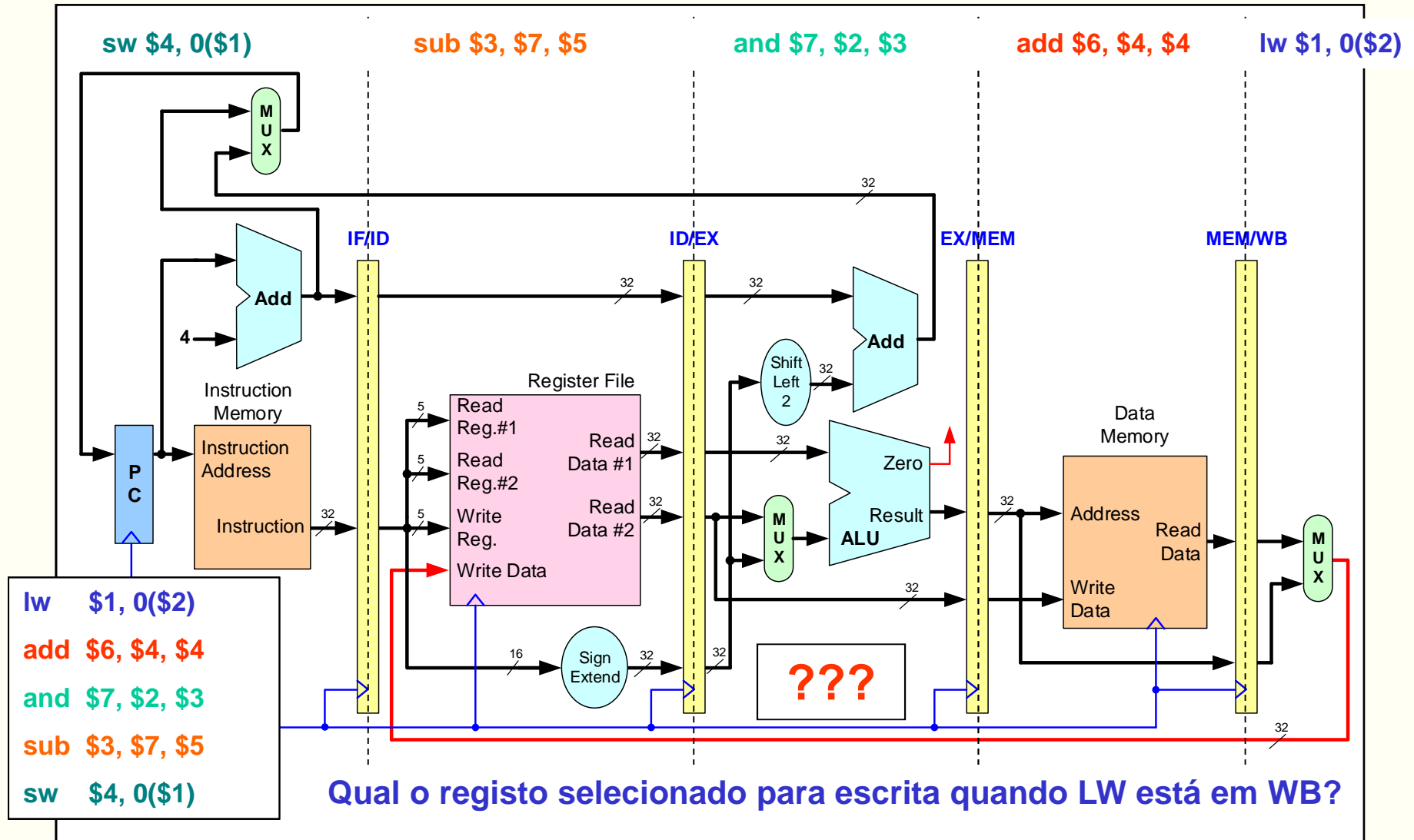
Divisão em fases de execução



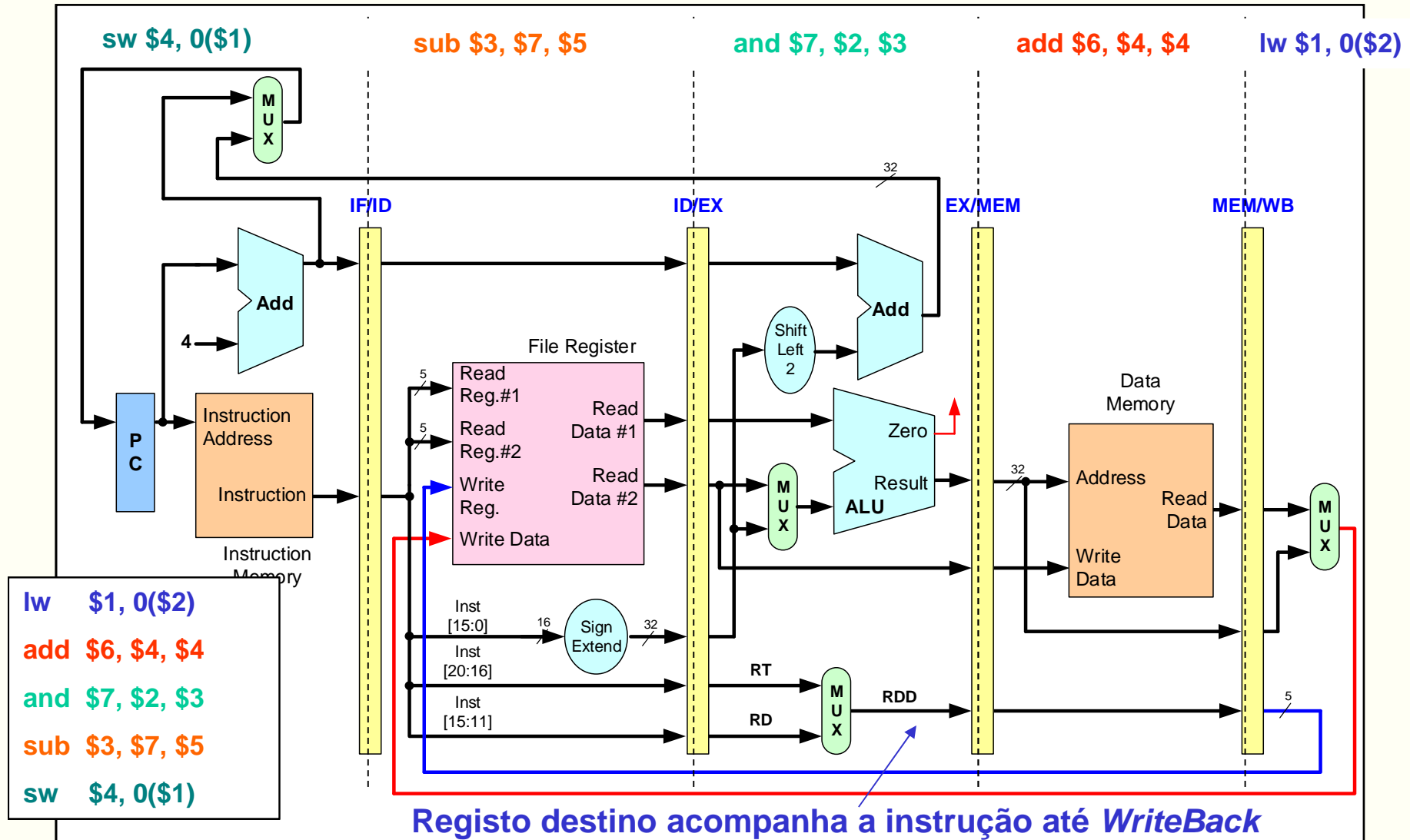
Divisão em fases de execução (com o sinal de relógio)



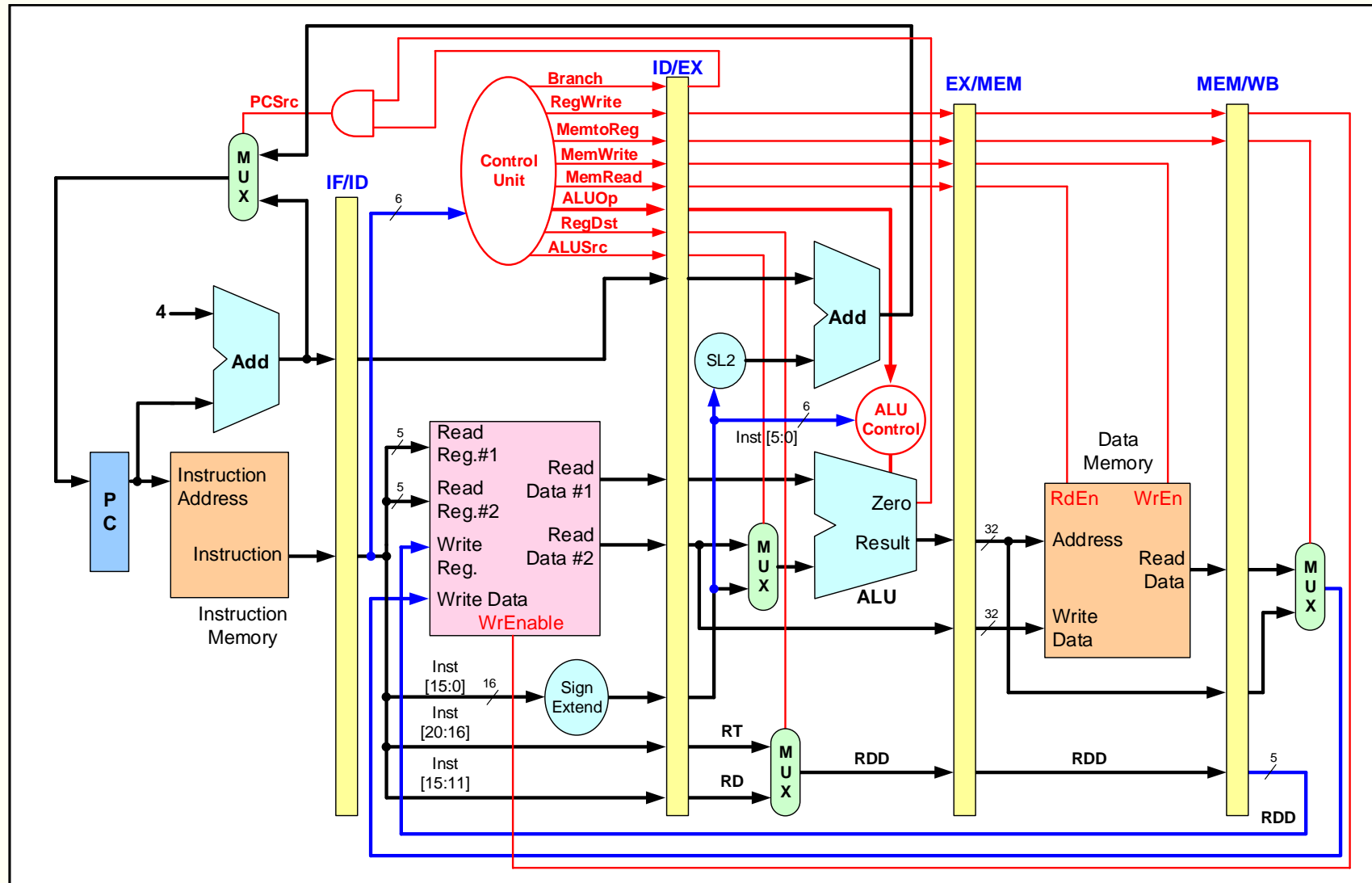
Execução de instruções



Datapath pipelined – 1ª versão



Datapath pipelined com unidade de controlo



Unidade de controlo

- A implementação *pipeline* do MIPS usa os mesmos sinais de controlo da versão *single-cycle*
- A unidade de controlo é, assim, uma **unidade combinatória** que gera os sinais de controlo em função do código da instrução (6 bits mais significativos da instrução, i.e., *opcode*) presente na fase ID
- Os sinais de controlo relevantes avançam no *pipeline* a cada ciclo de relógio (assim como os dados) estando, portanto, sincronizados com a instrução
- O sinal **RegWrite** é propagado até *WriteBack* e daí controla a escrita no *Register File* (fase ID)
- O sinal **Branch** é propagado até à fase EX (nesta versão o *branch* é resolvido nessa fase)

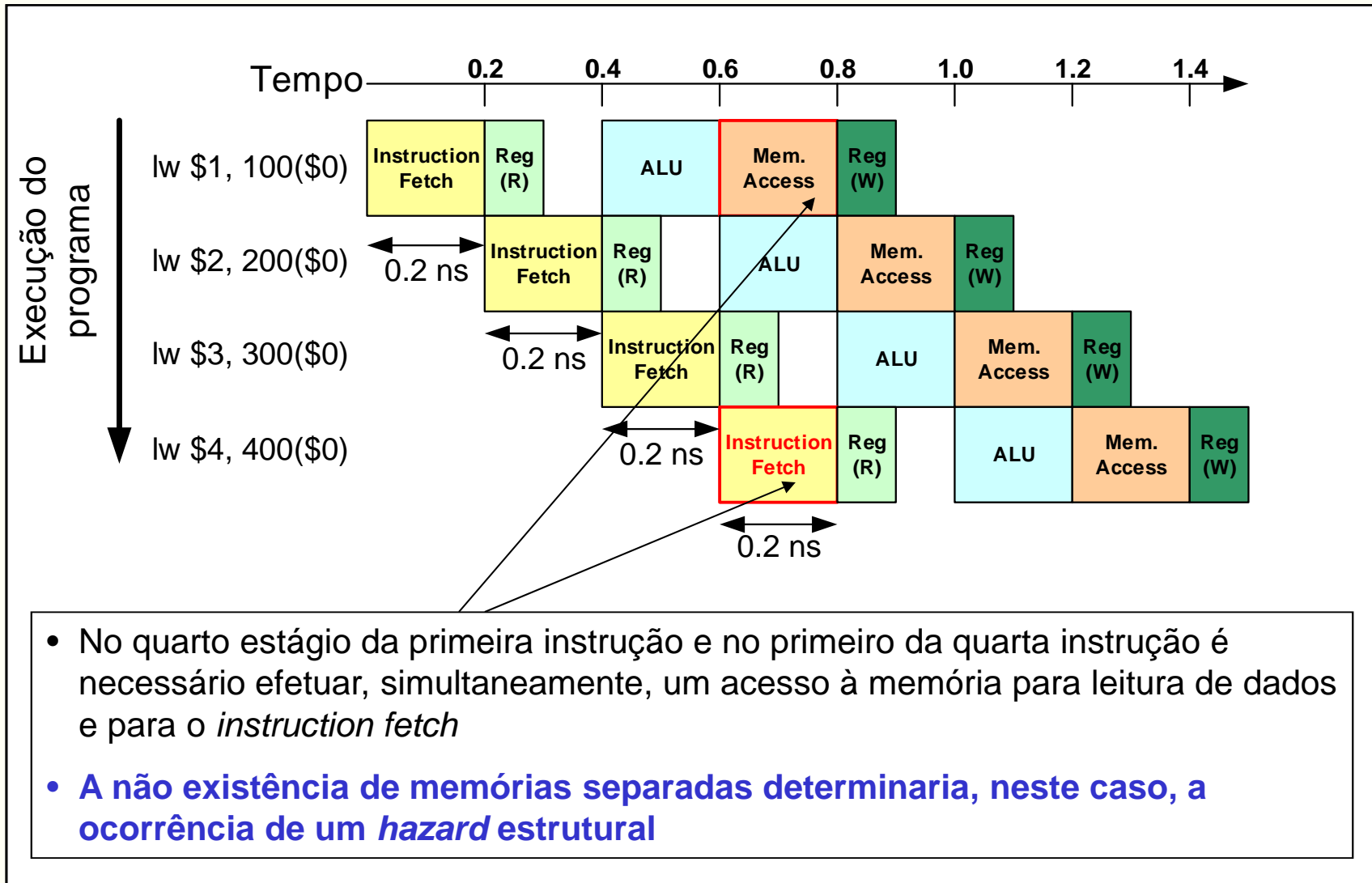
Pipeline Hazards

- Existe um conjunto de situações particulares que podem condicionar a progressão das instruções no *pipeline* no próximo ciclo de relógio
- Estas situações são designadas genericamente por **hazards**, e podem ser agrupadas em três classes distintas:
 - **Hazards estruturais**
 - **Hazards de controlo**
 - **Hazards de dados**
- Nos próximos slides serão discutidas, para cada tipo de *hazard*, as origens e as consequências, mapeando depois esses aspetos ao nível da arquitetura *pipelined* do MIPS

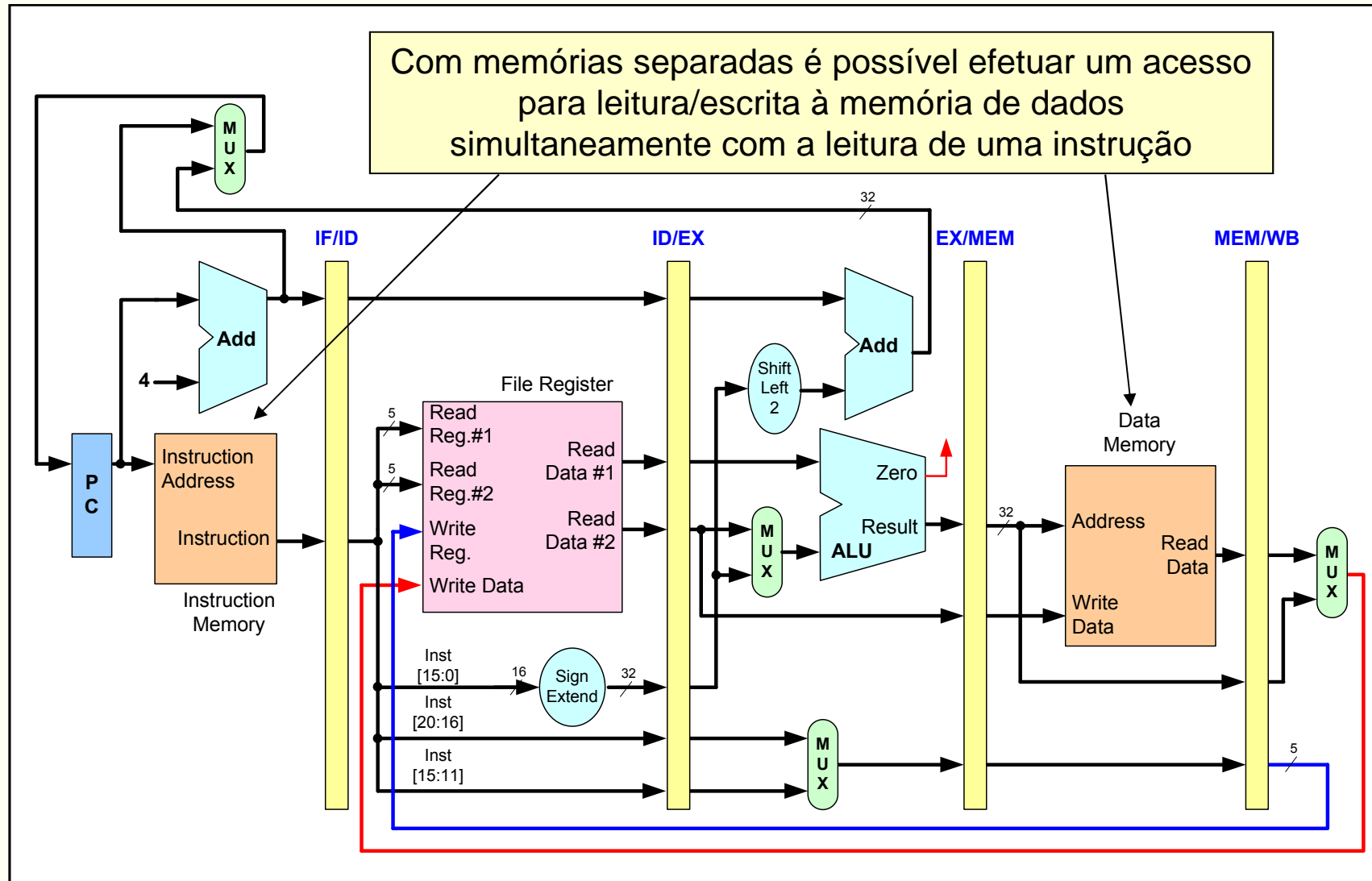
Hazards Estruturais

- Um **hazard estrutural** ocorre quando mais do que uma instrução necessita de aceder ao mesmo hardware
- Ocorre quando: 1) apenas existe uma memória ou 2) há instruções no *pipeline* com diferentes tempos de execução
- No primeiro caso o *hazard* estrutural é evitado duplicando a memória, i.e., uma memória de programa e uma memória de dados (acesso em IF não conflitua com possível acesso em MEM)
- O segundo caso está fora da análise feita nestes slides; como exemplo pode pensar-se na implementação de uma instrução mais complexa que demore 2 ciclos de relógio na fase EX

Hazards Estruturais



Hazards Estruturais



Hazards de Controlo

- Um *hazard* de controlo ocorre quando é necessário fazer o *instruction fetch* de uma nova instrução e existe numa etapa mais avançada do *pipeline* uma instrução que pode alterar o fluxo de execução e que ainda não terminou
- No caso do MIPS, as situações de *hazard* de controlo surgem com as instruções de salto, (*jumps* e *branches*)
- Exemplo:

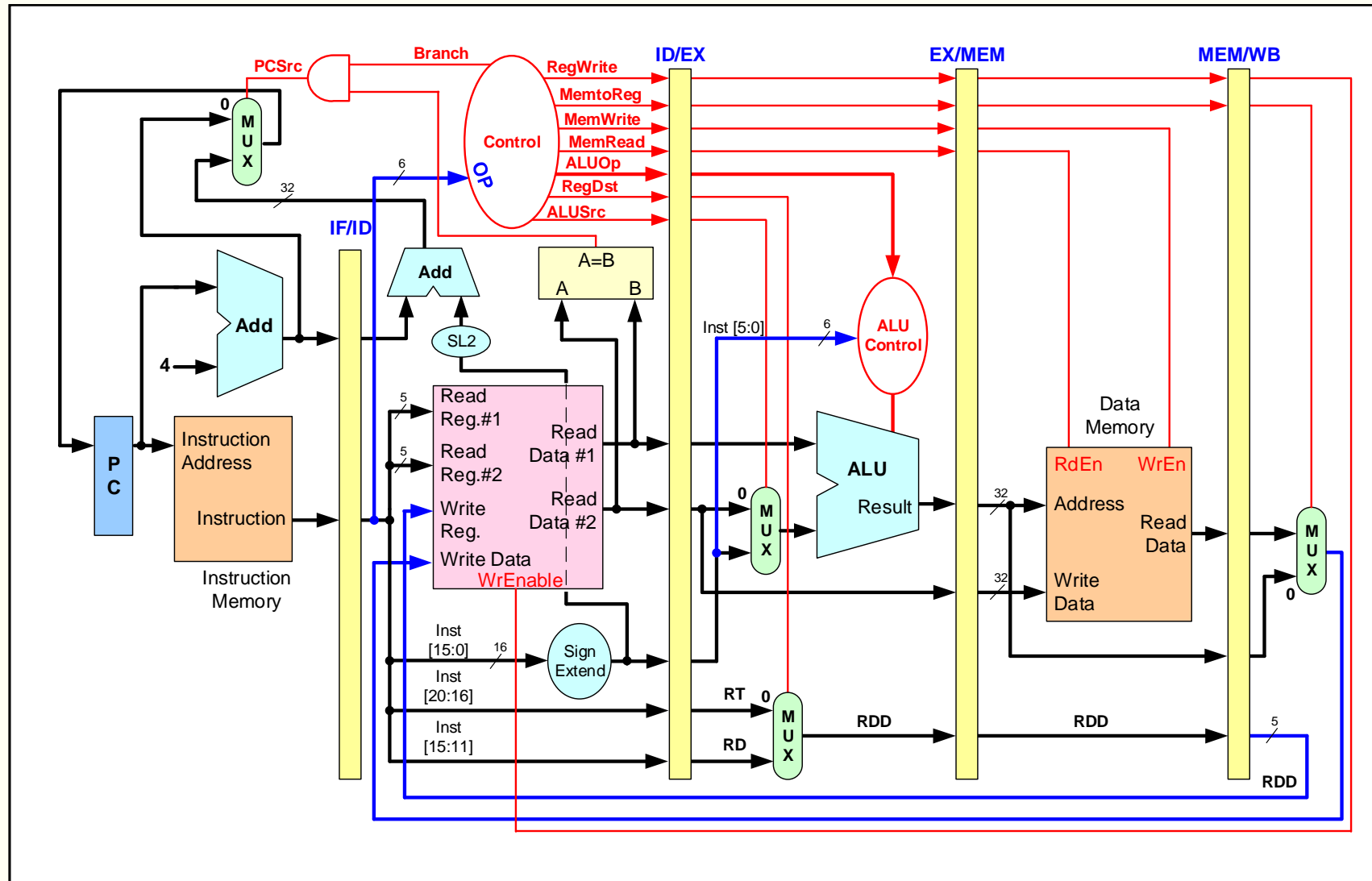
```
      beq $5,$6,next
      add $2,$3,$4
      ...
next:  lw  $3,0($4)
      ...
```

Qual a instrução que deve entrar no *pipeline* a seguir à instrução "beq"?

Hazards de Controlo

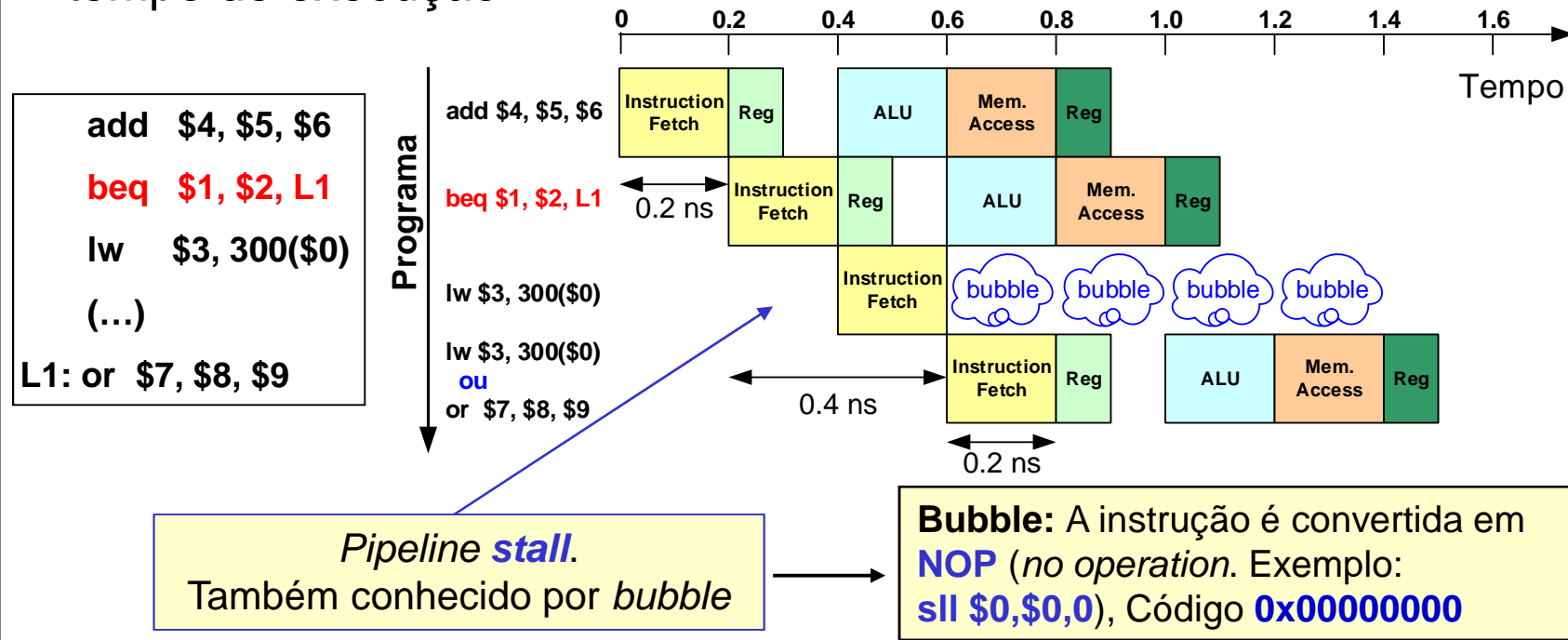
- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de código
- Na análise que se segue supõe-se que a **comparação dos operandos é efetuada no 2º estágio (ID)**, através de hardware adicional
- Do mesmo modo, **o cálculo do *Branch Target Address* passa também a ser efetuado em ID**

Datapath com branches resolvidos em ID



Hazards de controlo

- Há mais do que uma solução para lidar com os *hazards* de controlo. A primeira que vamos analisar é designada por **stalling** ("parar o progresso de...")
- Nesta estratégia a unidade de controlo atrasa a entrada no *pipeline* da próxima instrução até saber o resultado do *branch* condicional
- É uma solução conservativa que tem um preço em termos de tempo de execução



Exercício 2

- Se 15% das instruções de um dado programa forem *branches*, qual o efeito desta estratégia no desempenho da arquitetura, admitindo que os *branches* são resolvidos em ID?

Sem *stalls*: $CPI = 1$

Com *stalls*: $CPI = 1 + 1 * 0,15 = 1,15$

Relação de desempenho = ???

- A degradação do desempenho é tanto maior quanto mais tarde for resolvida a instrução de *branch*. Na mesma situação, se o *branch* for resolvido em EX, a relação passa a ser:

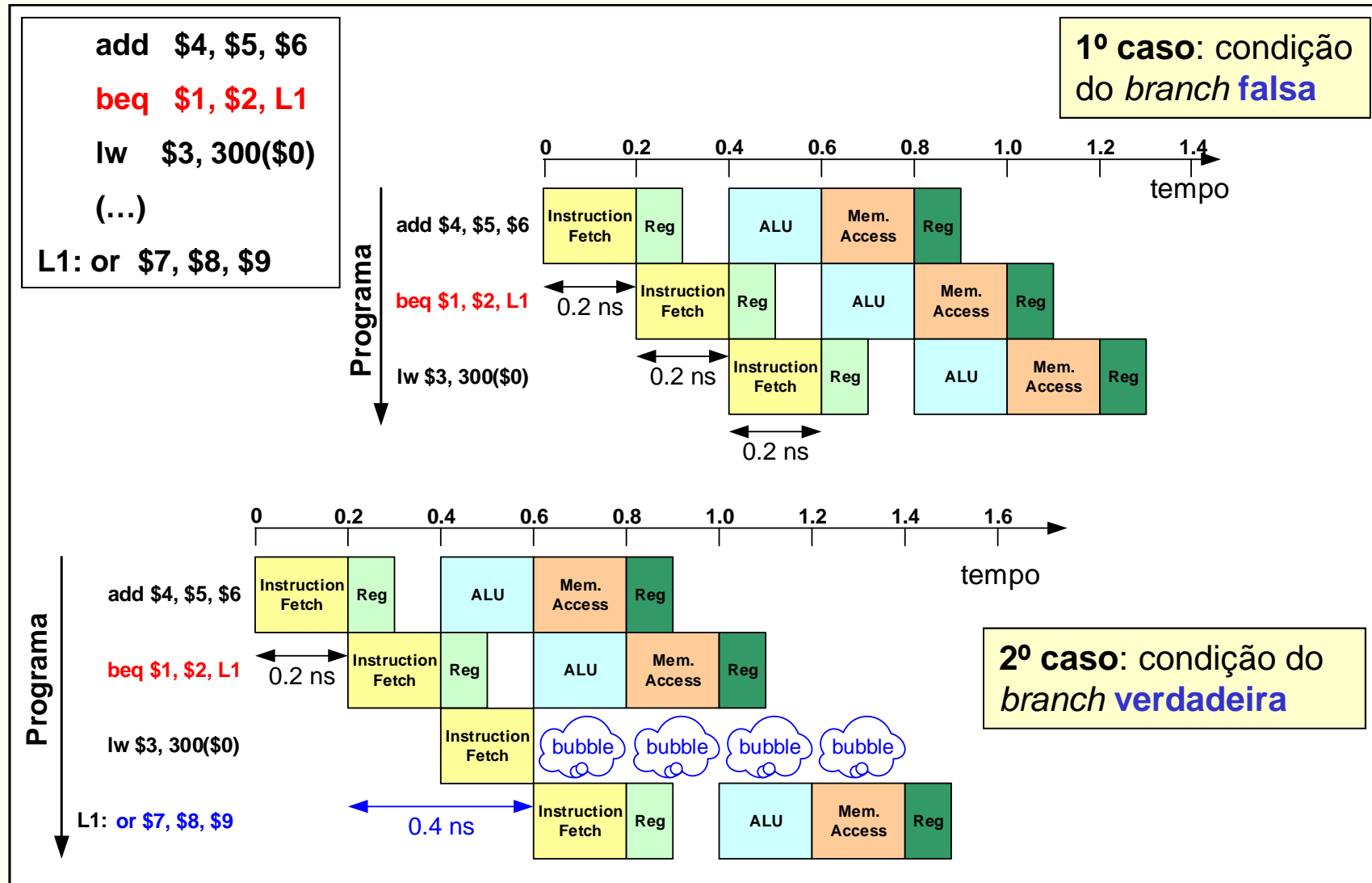
Relação de desempenho = ???

Hazards de controlo

- Uma solução alternativa ao *pipeline stalling* é designada por **previsão** (*prediction*):
 - Assume-se que a condição do *branch* é falsa (*branch not taken*), pelo que a próxima instrução a ser executada será a que estiver em PC+4 – estratégia designada por **previsão estática not taken**
 - Se a previsão falhar, a instrução entretanto lida (a seguir ao *branch*) é descartada (convertida em **nop**), continuando o *instruction fetch* na instrução correta
- Se a previsão estiver certa esta estratégia permite poupar tempo; para o exemplo anterior, se a previsão for correta 50% das vezes, a relação de desempenho passa a ser:
$$\text{CPI} = 1 + 1 * 0,15 / 2$$

Relação de desempenho = ???

Hazards de controlo – previsão *not taken*



Hazards de controlo – previsão

- Os previsores usados nas arquiteturas atuais são mais elaborados
- **Previsores estáticos**: o resultado da previsão não é dependente do resultado da execução das instruções:
 - **Previsor *Not taken***
 - **Previsor *Taken***
 - **Previsor *Backward taken, Forward not taken*** (BTFNT)
- **Previsores dinâmicos**: o resultado da previsão depende da história de *branches* anteriores:
 - Guardam informação do resultado *taken/not taken* de *branches* anteriores e do *target address*
 - A previsão é feita com base na informação guardada

Hazards de controlo – a solução do MIPS

- Uma outra alternativa para resolver os *hazards* de controlo, adotada no MIPS, é designada por **delayed branch**
- Nesta abordagem, o processador **executa sempre a instrução que se segue ao branch**, independentemente de a condição ser verdadeira ou falsa
- Esta técnica é implementada com a ajuda do **compilador/assembler** que:
 - organiza as instruções do programa por forma a trocar a ordem do *branch* com uma instrução anterior (desde que não haja dependência entre as duas), ou
 - não sendo possível efetuar a troca de instruções introduz um **NOP** ("no operation"; ex.: **sll, \$0, \$0, 0**) a seguir ao *branch*
- Não é uma técnica comum nos processadores modernos

Hazards de controlo – *delayed branch*

- Esta técnica **é escondida do programador** pelo compilador/assembler:

Código original

```
add  $4, $5, $6
beq  $1, $2, L1
lw   $3, 300($0)
(...)
L1: or $7, $8, $9
```

Assembler troca a
ordem das duas 1^{as}
instruções

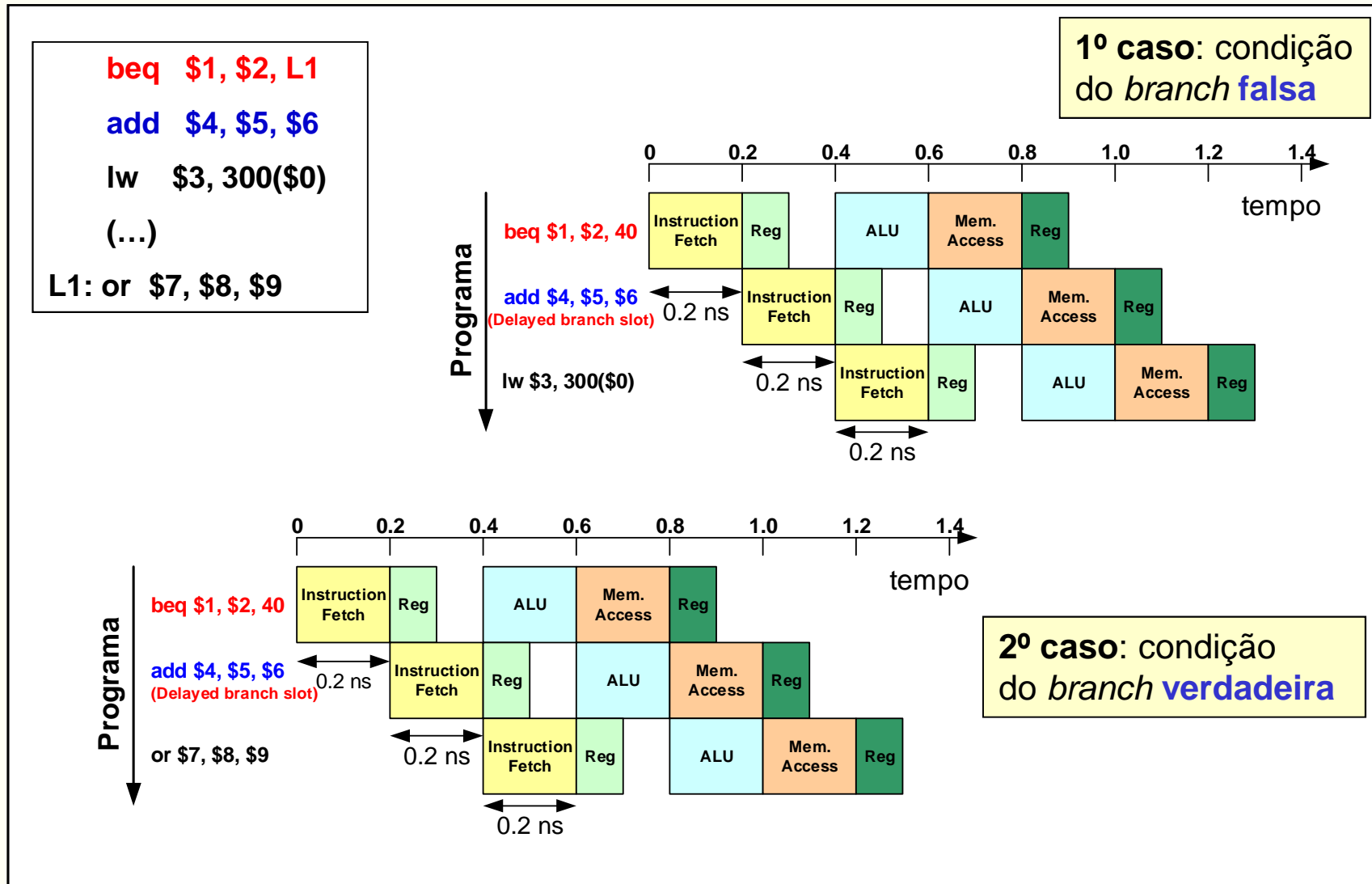


Código reordenado

```
beq  $1, $2, L1
add  $4, $5, $6
lw   $3, 300($0)
(...)
L1: or $7, $8, $9
```

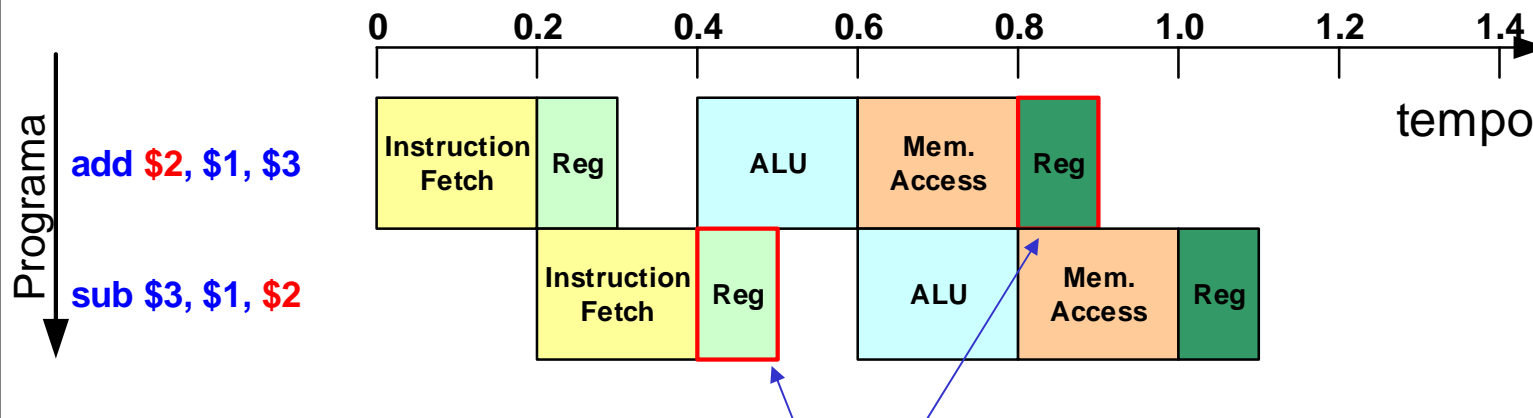
- Neste exemplo a instrução "**beq**" não depende do resultado produzido pela instrução "**add**", logo a troca das duas não altera o resultado final do programa
- No código reordenado (que é o executado no processador) a instrução "**add**" é sempre executada, independentemente do resultado *taken/not taken* do "**beq**"

Hazards de controlo – *delayed branch*



Hazards de dados

- O terceiro tipo de *hazards* resulta da **dependência** existente entre o resultado calculado por uma instrução e o operando usado por outra que segue mais atrás no *pipeline* (i.e., mais recente)
- Exemplo:
add **\$2**, \$1, \$3
sub \$3, \$1, **\$2**



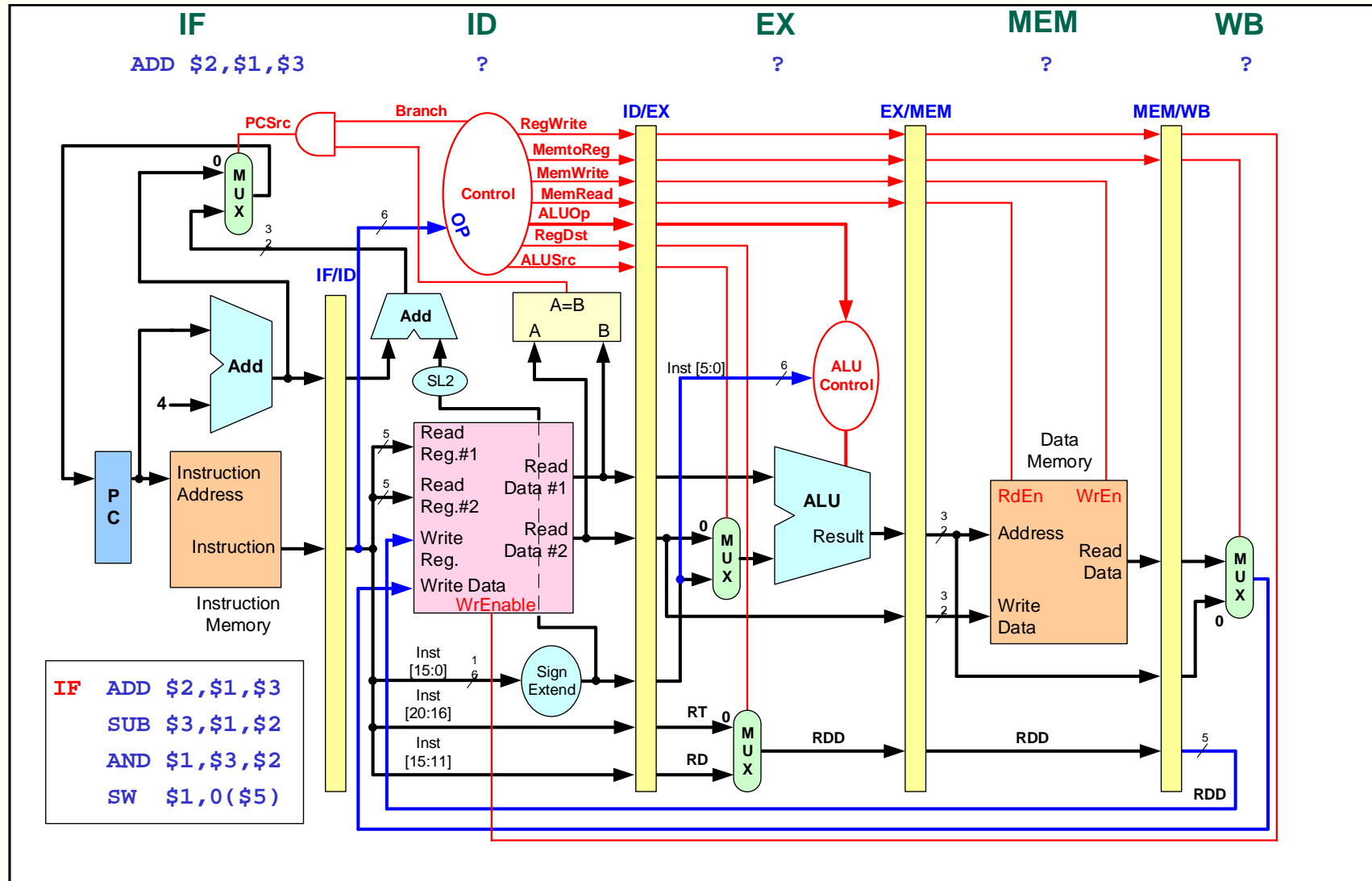
A instrução "**sub \$3,\$1,\$2**" não pode ser executada antes de o valor de **\$2** ser calculado e armazenado pela instrução anterior (o valor é necessário em **t = 4**, mas só vai ser escrito no registo destino em **t = 10**)

Hazards de dados

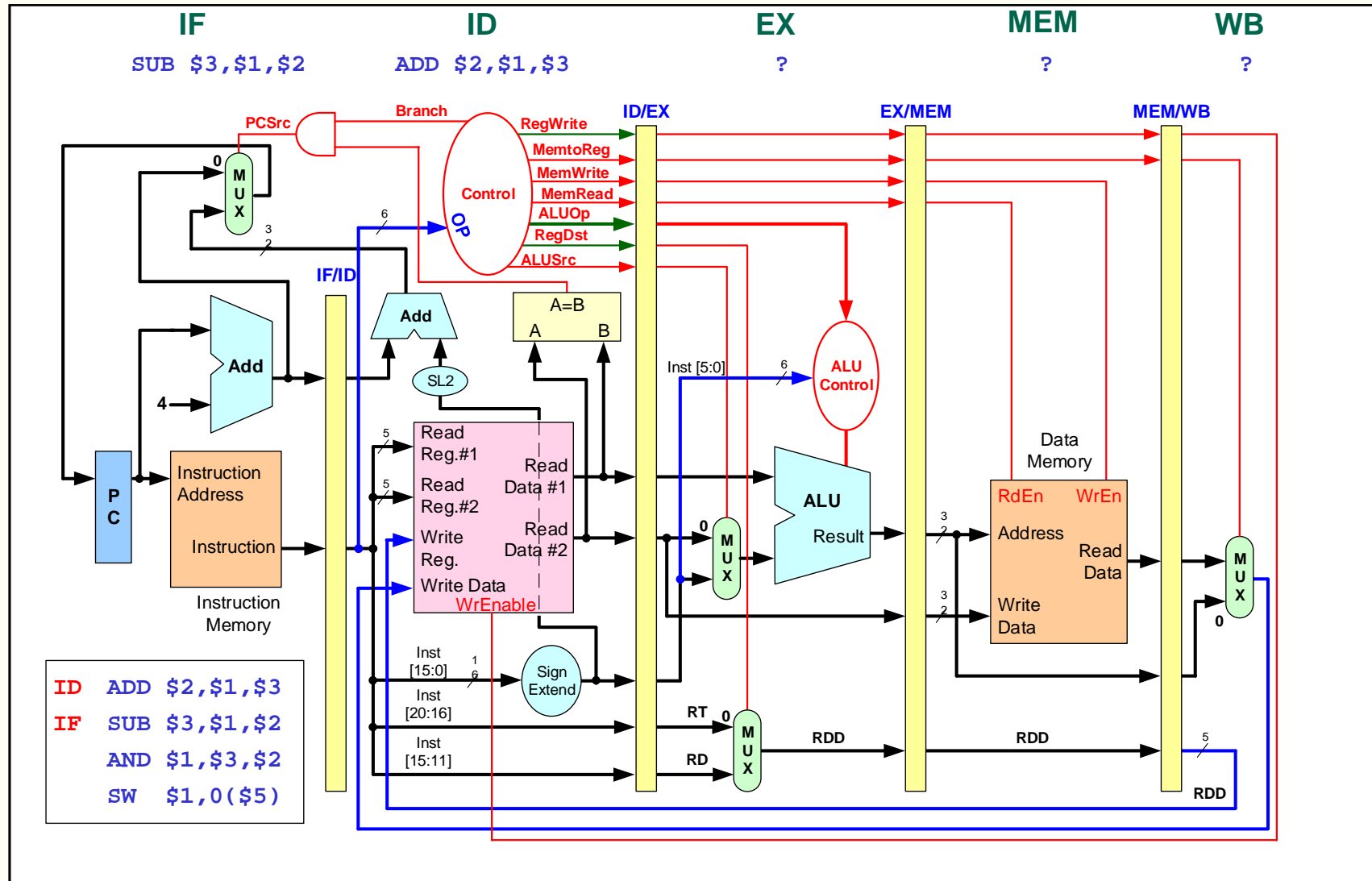
- Se o resultado que vai ser necessário para a instrução mais recente ainda não tiver sido armazenado, então essa instrução não poderá prosseguir porque irá tomar como operando um valor incorreto (**a escrita no registo só é feita quando a instrução chega a WB**)
- Exemplo:

| | | | | | | | |
|-----|-----|---|-----|---|-----|---|-------------------------|
| ADD | \$2 | , | \$1 | , | \$3 | | # |
| SUB | \$3 | , | \$1 | , | \$2 | | # Hazard de dados (\$2) |
| AND | \$1 | , | \$3 | , | \$2 | | # Hazard de dados (\$3) |
| SW | \$1 | , | 0 | (| \$5 |) | # Hazard de dados (\$1) |
- Primeira solução: **stall do pipeline**
 - parar a progressão da instrução que necessita do valor (e das anteriores, no pipeline), na etapa **ID**, até que a instrução que produz o resultado chegue à etapa **WB**
 - se a **escrita no banco de registos for feita a meio do ciclo de relógio**, então a instrução que necessita do valor poderá prosseguir na transição de relógio seguinte, já com o valor do registo atualizado

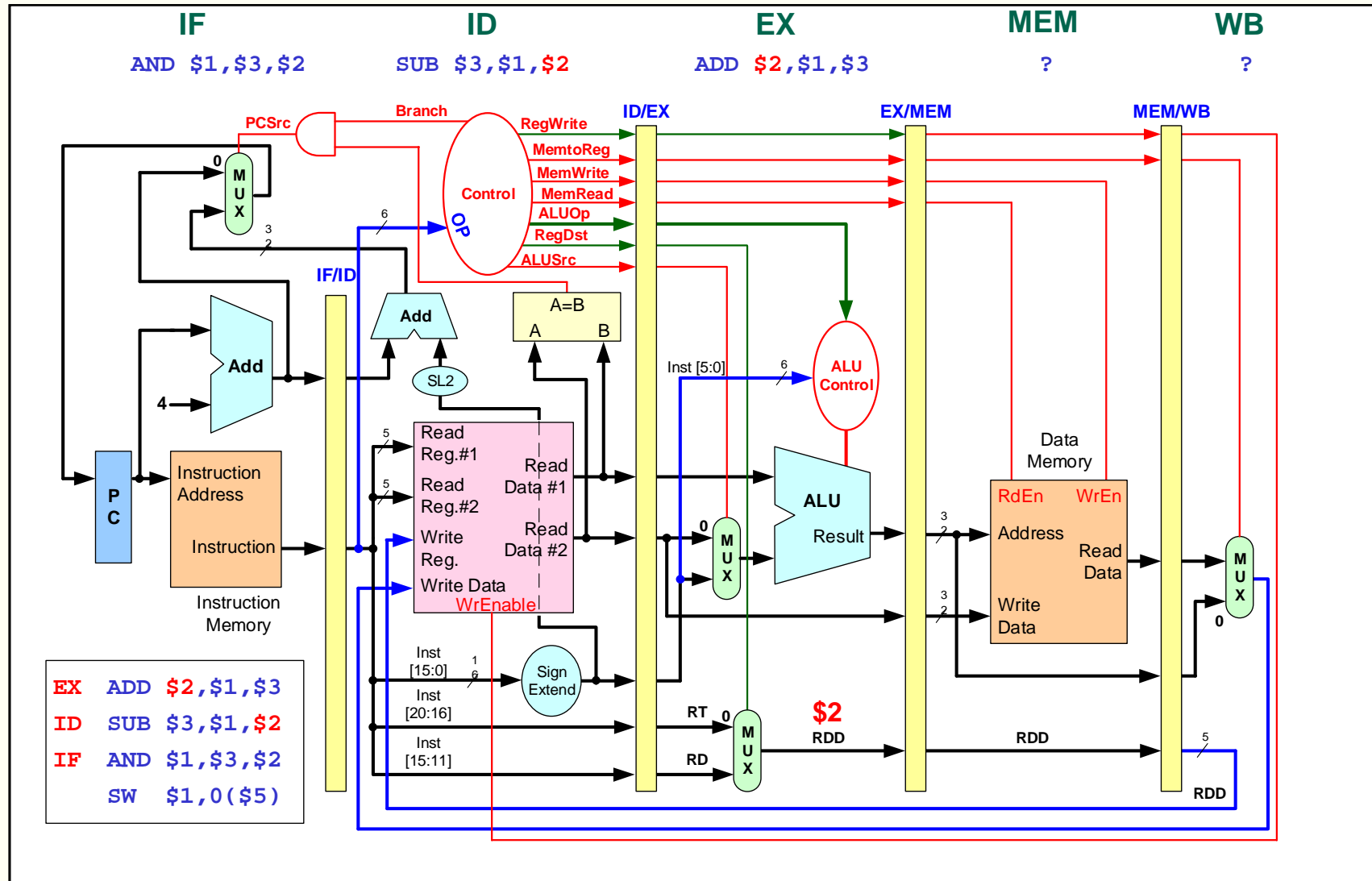
Hazards de dados resolvidos com *stalling* (1)



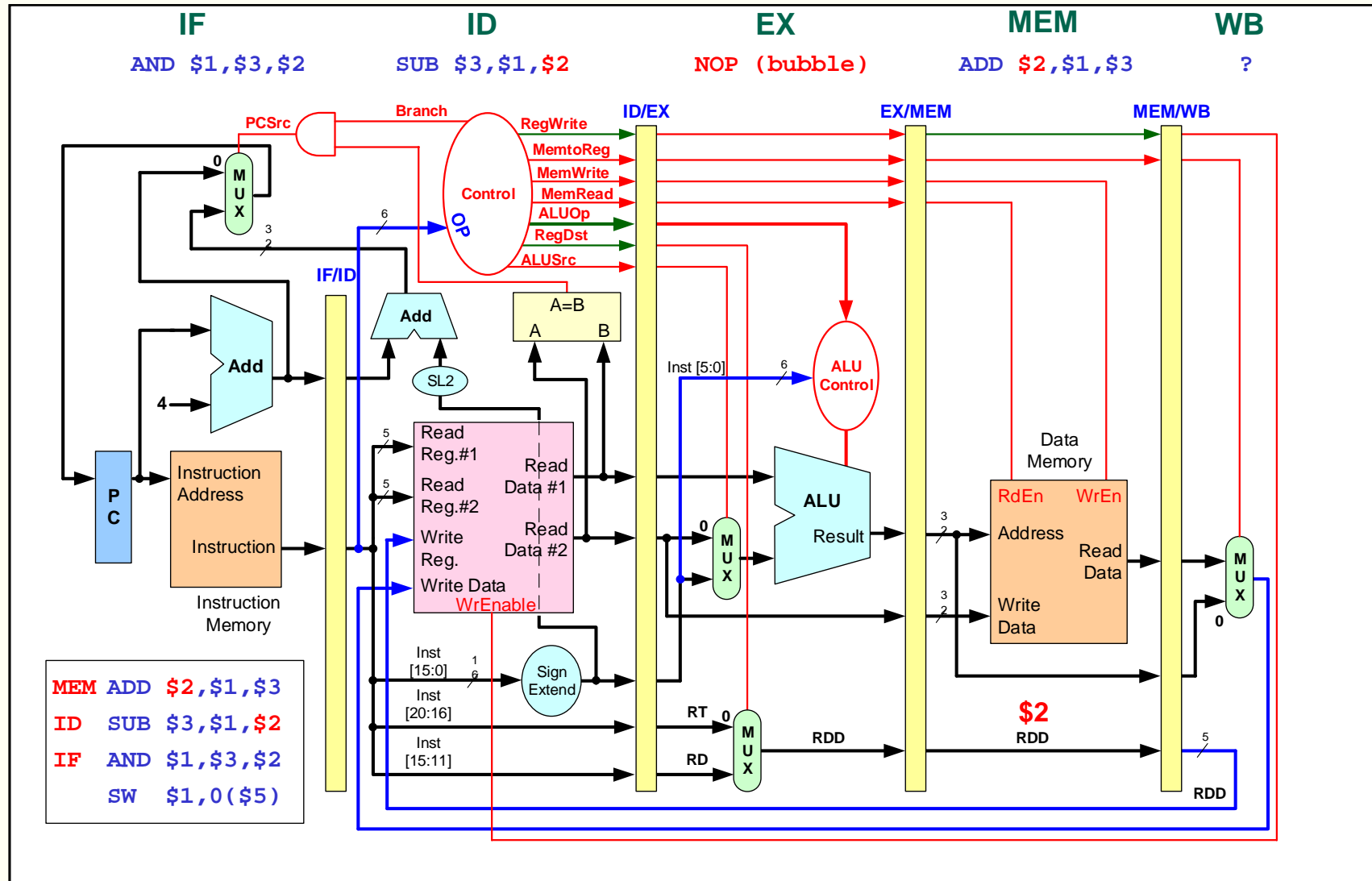
Hazards de dados resolvidos com *stalling* (2)



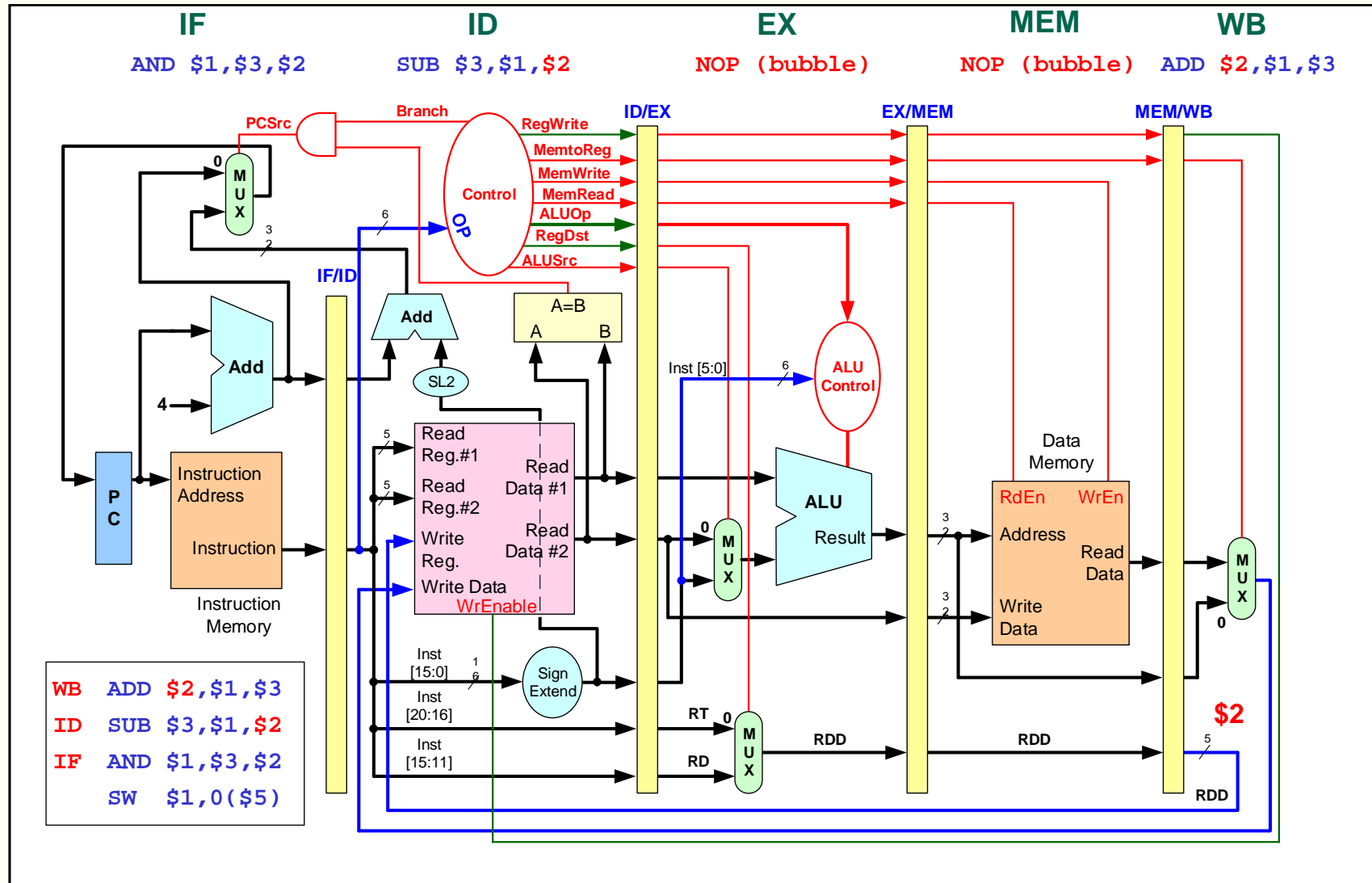
Hazards de dados resolvidos com stalling (3)



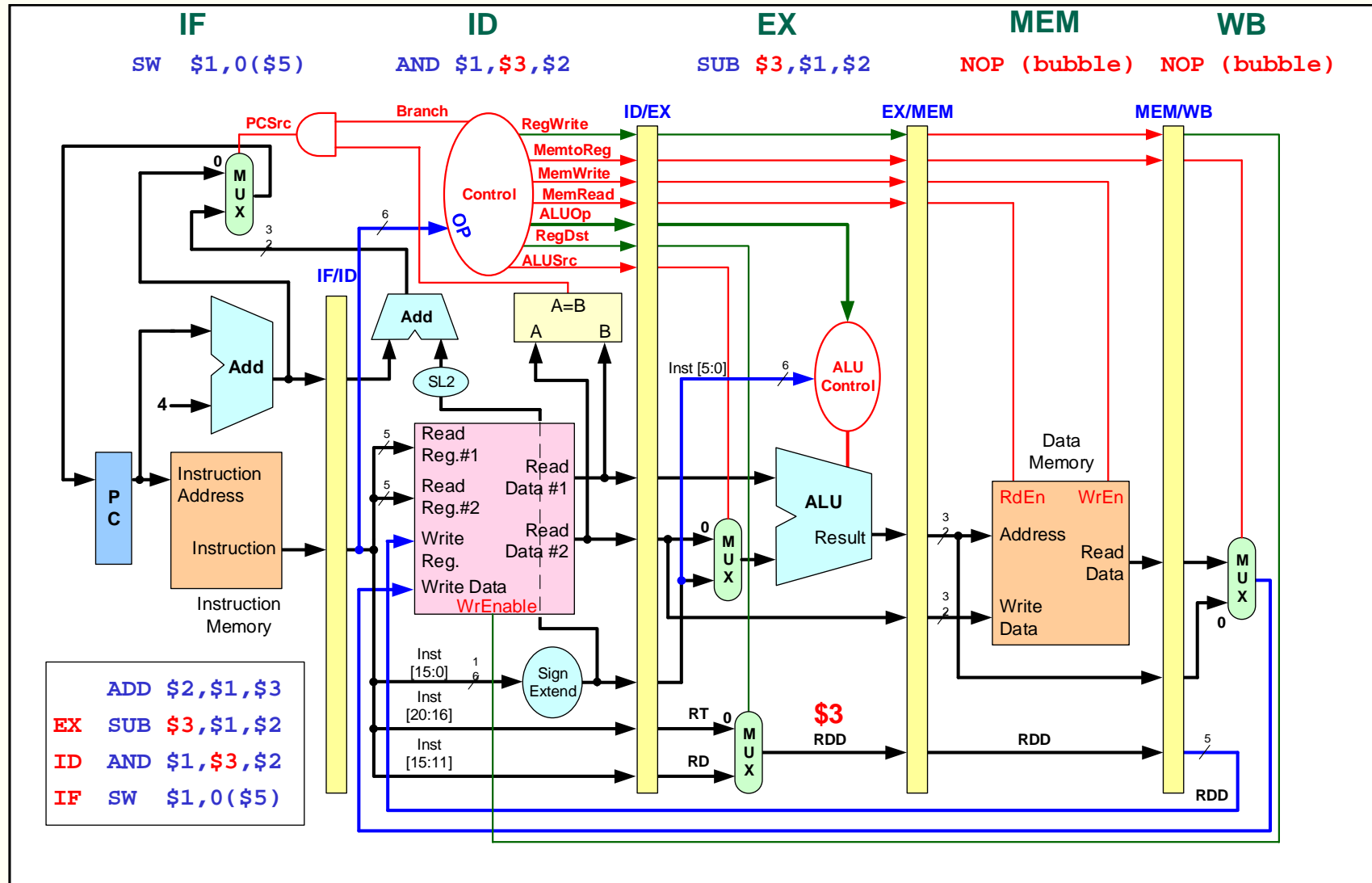
Hazards de dados resolvidos com *stalling* (4)



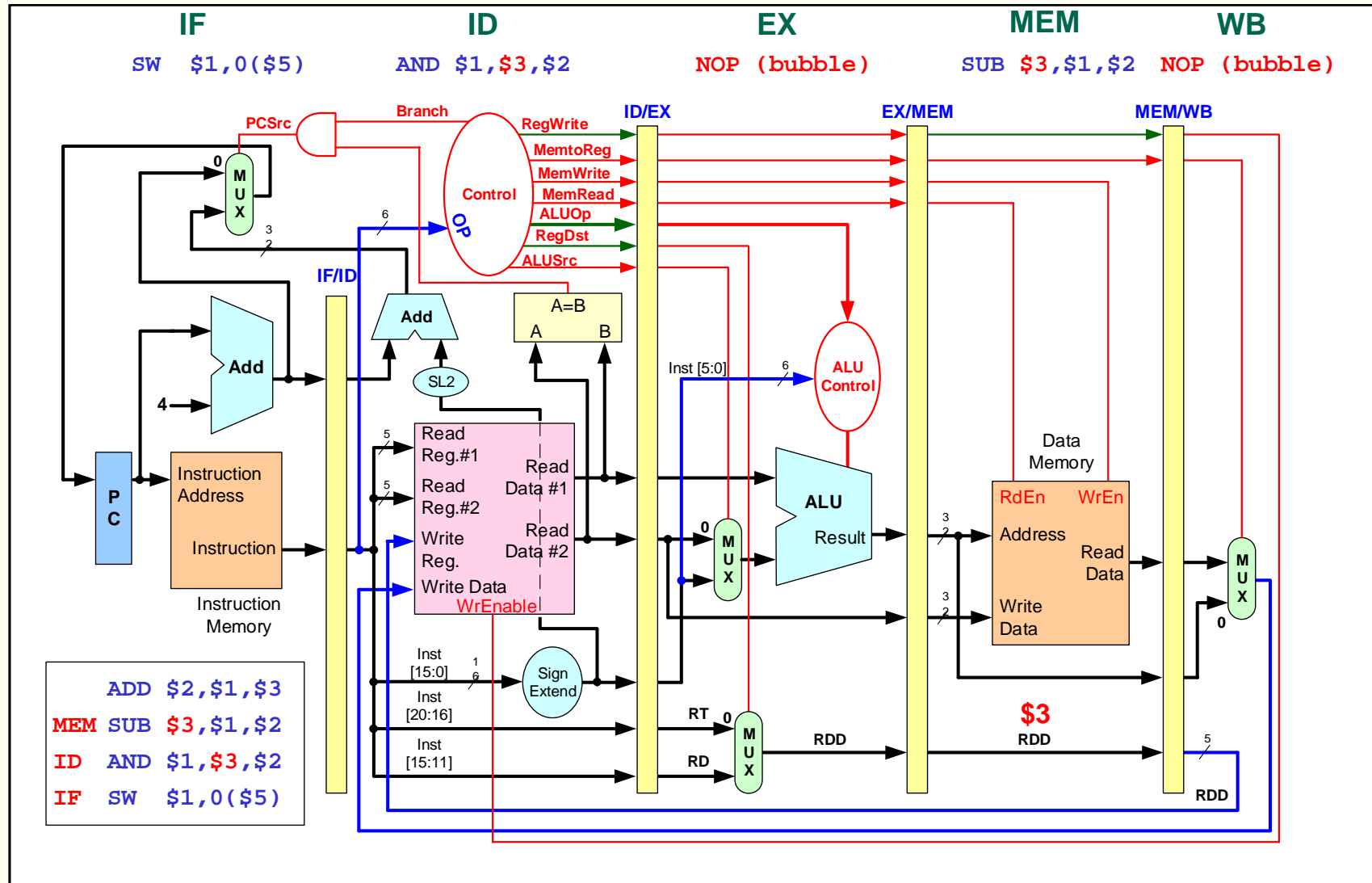
Hazards de dados resolvidos com stalling (5)



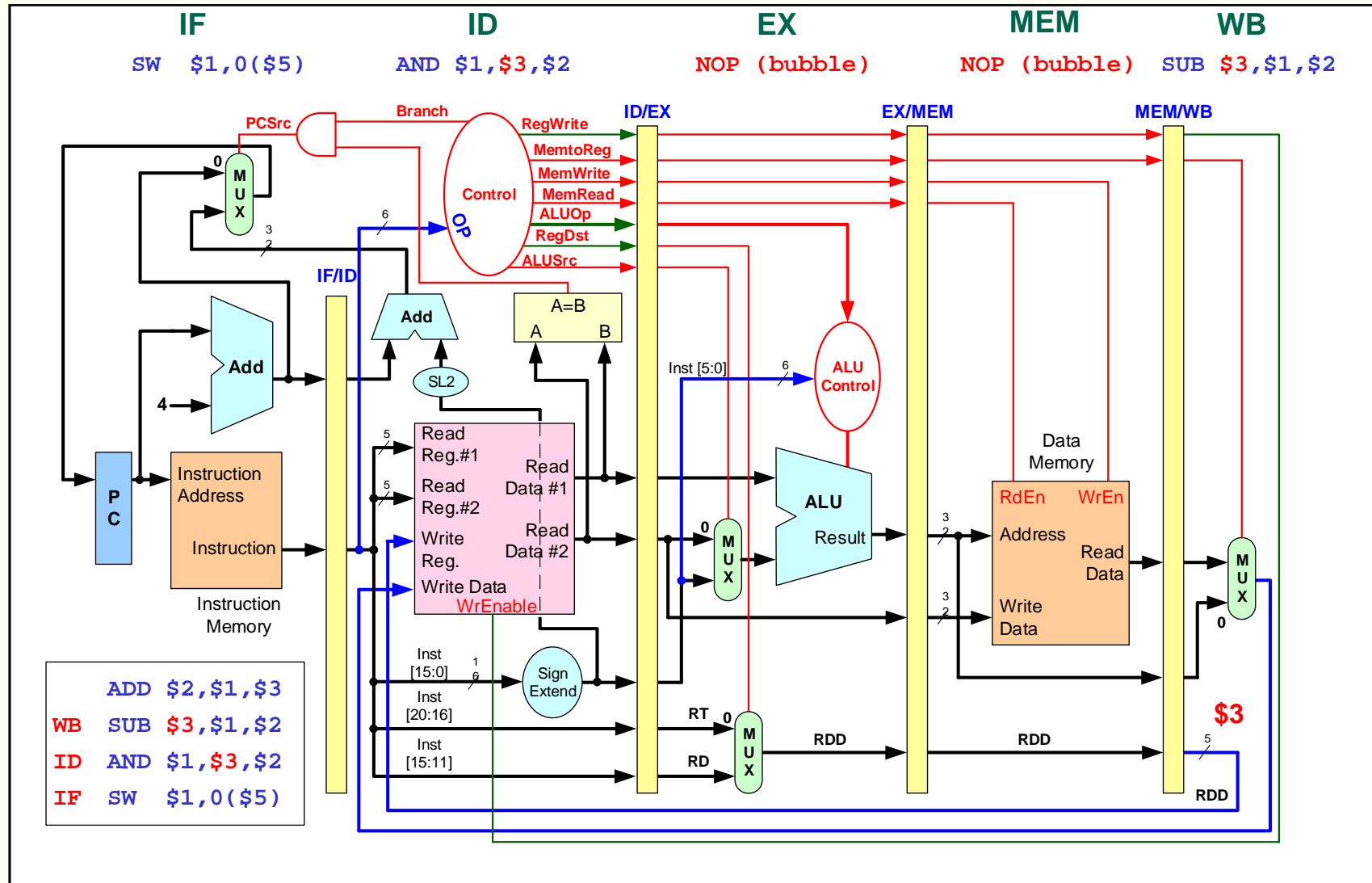
Hazards de dados resolvidos com stalling (6)



Hazards de dados resolvidos com stalling (7)



Hazards de dados resolvidos com *stalling* (8)



Hazards de dados

- Esperar pela conclusão da instrução que produz o resultado (através de *stalling*) tem um impacto elevado no desempenho...
- Cada instrução com dependência atrasa a progressão do *pipeline* em 2 ciclos de relógio (2 T); para o trecho de código do exemplo anterior, 6 T no total (3 *hazards* de dados):

$$\text{Texec_sem_stalls} = F + (N-1) = 5 + (4-1) = 8 \text{ T}$$

$$\text{Texec} = F + (N-1) + \text{Nr_stalls} = 5 + (4-1) + 6 = 14 \text{ T}$$

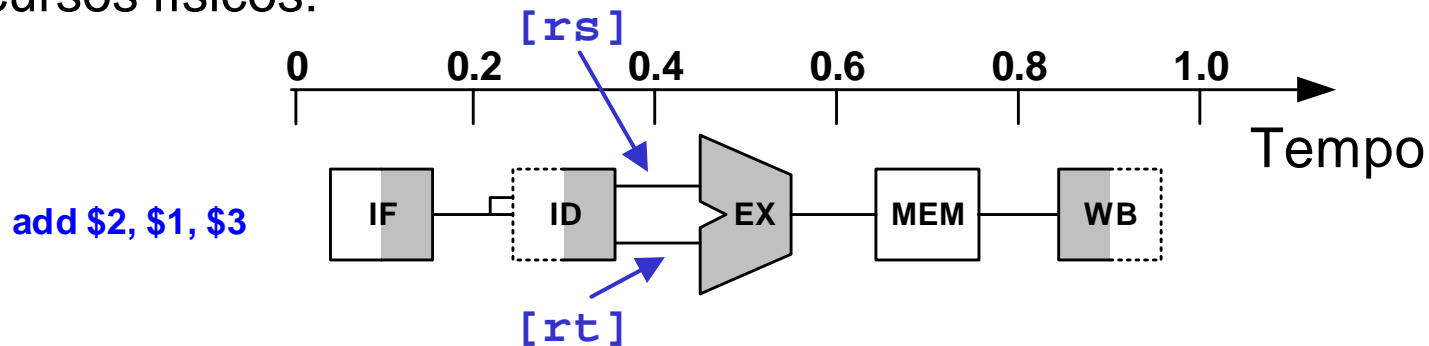
- Qual será então a solução?
- A principal solução para a resolução de situações de *hazards* de dados resulta da observação de que não é necessário, na maioria dos casos, esperar pela conclusão da primeira instrução para resolver o *hazard*

Hazards de dados

- Por exemplo, para as instruções do tipo R, logo que a operação seja realizada na ALU, (**EX**, 3º estágio), o resultado pode ser disponibilizado para a instrução seguinte
- Esta técnica de disponibilizar um resultado para uma instrução subsequente, mais cedo na cadeia de *pipelining*, é conhecida por **forwarding** ou **bypassing**

Representação gráfica da cadeia de *pipelining*

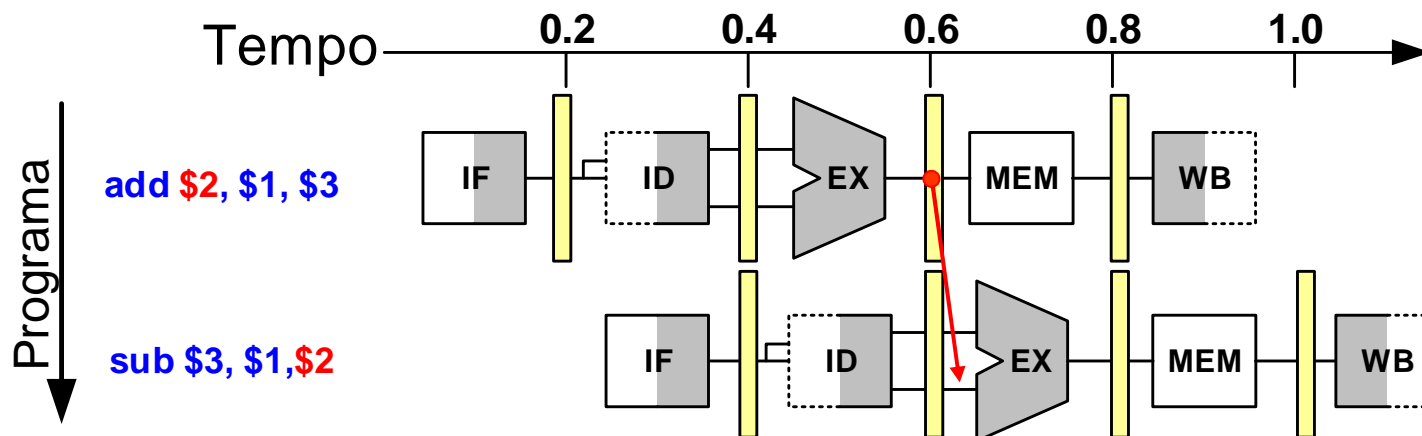
- Nesta representação gráfica usamos símbolos para representar os recursos físicos:



- IF corresponde ao estágio de *instruction fetch*, representando o quadrado a memória de instruções
- A metade cinza à direita tipifica uma operação de leitura
- Um quadrado branco (MEM) indica que esse elemento de estado não está envolvido na execução da instrução
- Quando a metade cinza está à esquerda, isso indica uma operação de escrita no elemento de estado respetivo (WB)

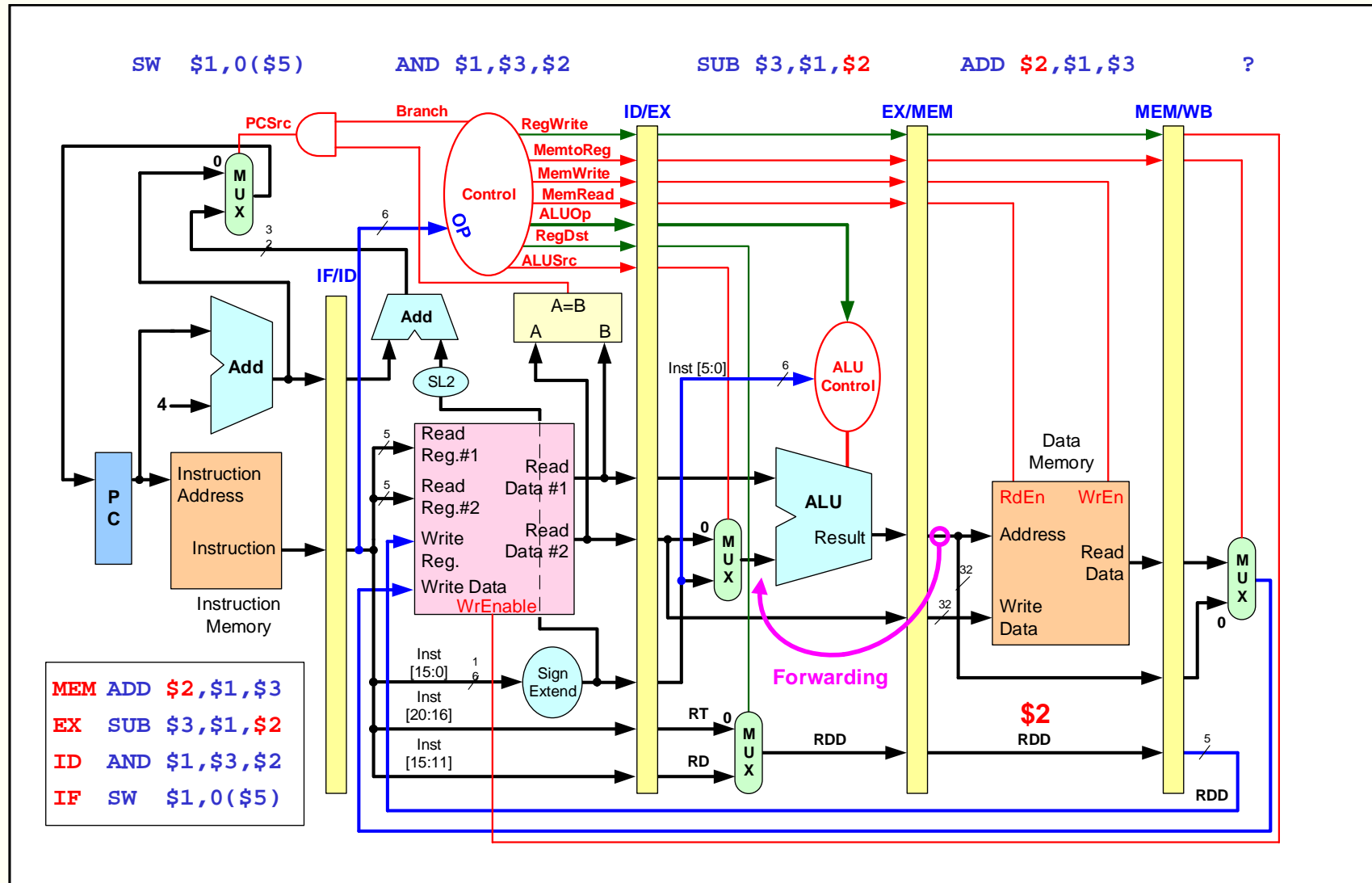
Hazards de dados

- O exemplo anterior, em que se observou a existência de um *hazard* de dados, pode então ser representado graficamente por:



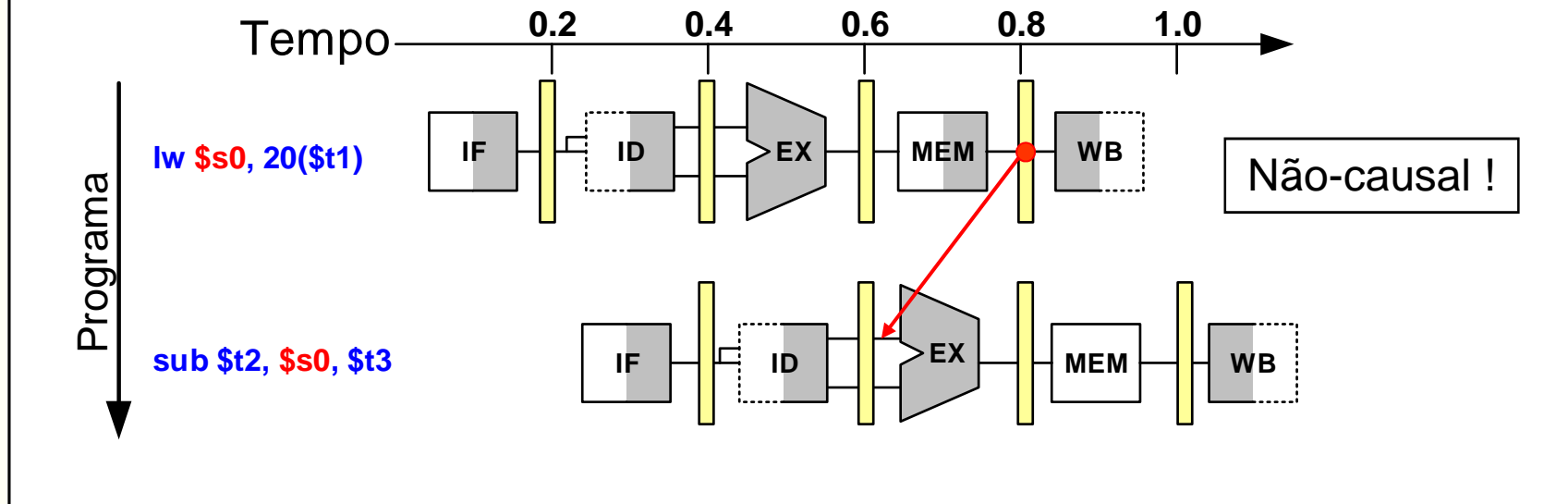
- O **forwarding** do valor presente no registo **EX/MEM** (resultado da instrução ADD) para a segunda entrada da ALU (estágio **EX**, instrução SUB) resolve o *hazard* de dados

Hazards de dados - forwarding



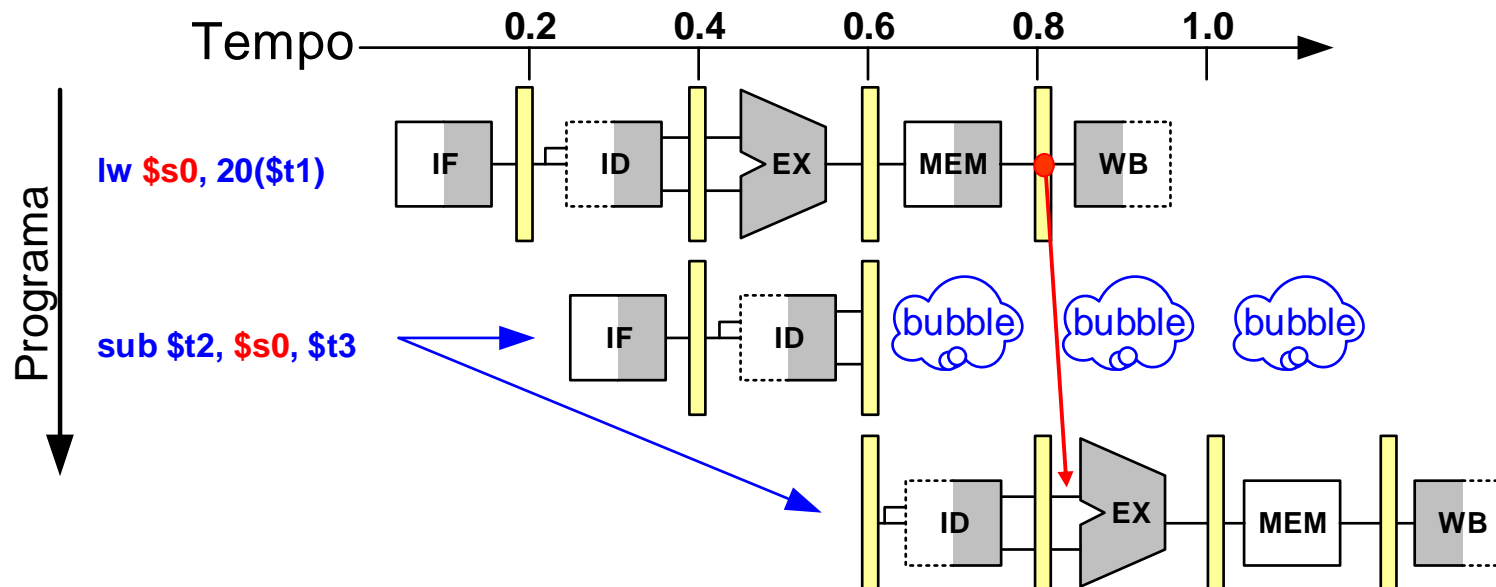
Hazards de dados

- Há situações em que o *forwarding*, por si só, não resolve o *hazard* de dados
- Um exemplo é o que ocorre quando uma instrução aritmética/lógica depende do resultado de uma instrução de acesso à memória (LW) que ainda não terminou



Hazards de dados – stalling

- Para resolver essa situação, é necessário:
 - Fazer o **stall** do *pipeline* durante um ciclo de relógio
 - Fazer o **forwarding** do registro **MEM/WB** para o estágio **EX**, para a entrada da ALU



Hazards de dados – reordenação de instruções

- Parte das situações de *hazards* de dados podem ser atenuadas ou **resolvidas pelo compilador**, através da reordenação de instruções
- A reordenação não pode comprometer o resultado final
- Código original (exemplo):

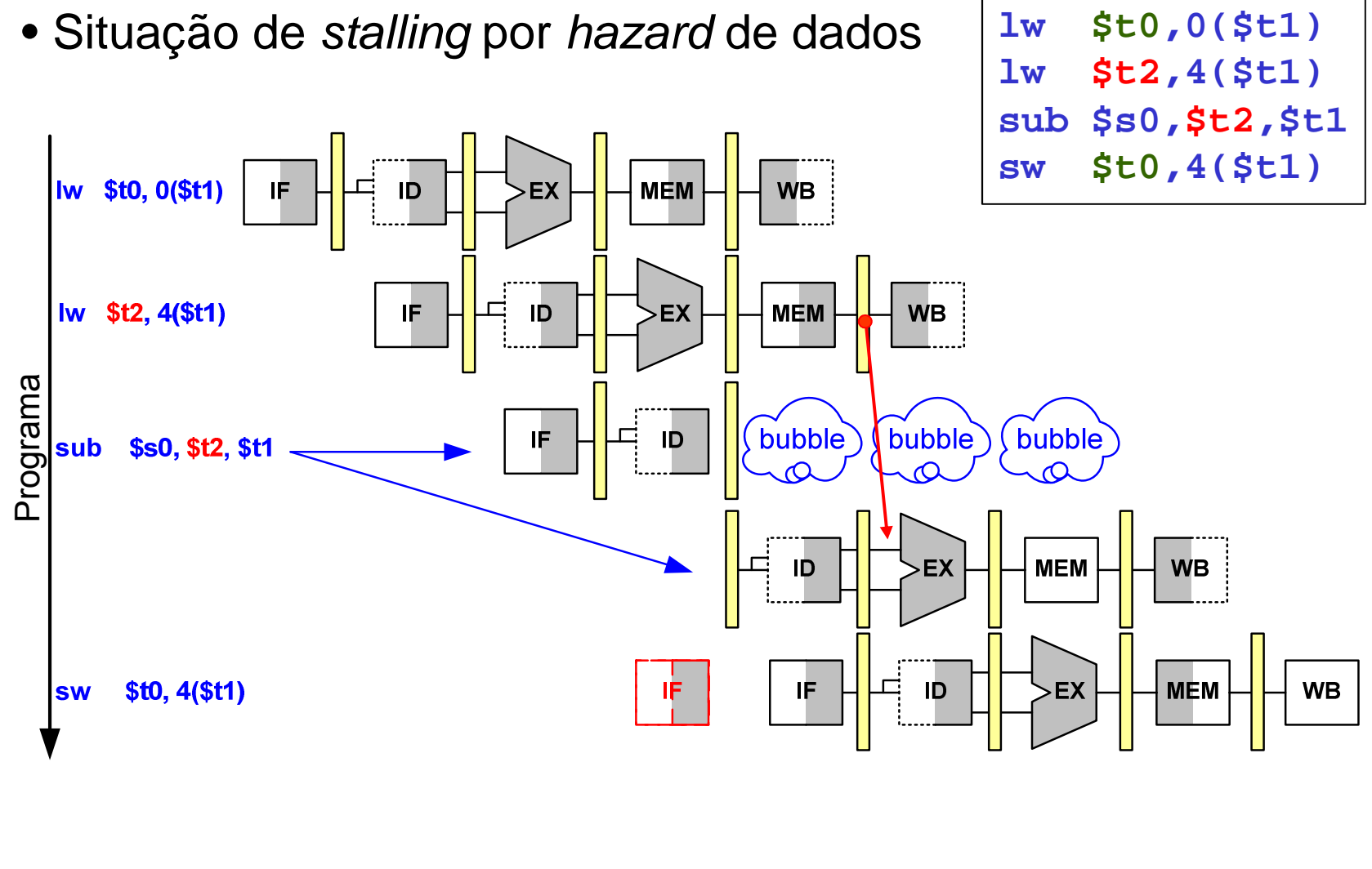
```
lw    $t0,0($t1)
lw    $t2,4($t1)
sub   $s0,$t2,$t1 # stalling por hazard de dados
sw    $t0,4($t1)
```

- Código reordenado pelo compilador/assembler:

```
lw    $t0,0($t1)
lw    $t2,4($t1)
sw    $t0,4($t1) # FW: MEM/WB > EX (rt)
sub   $s0,$t2,$t1 # stalling resolvido por reordenação
                        # FW: MEM/WB > EX (rs)
```

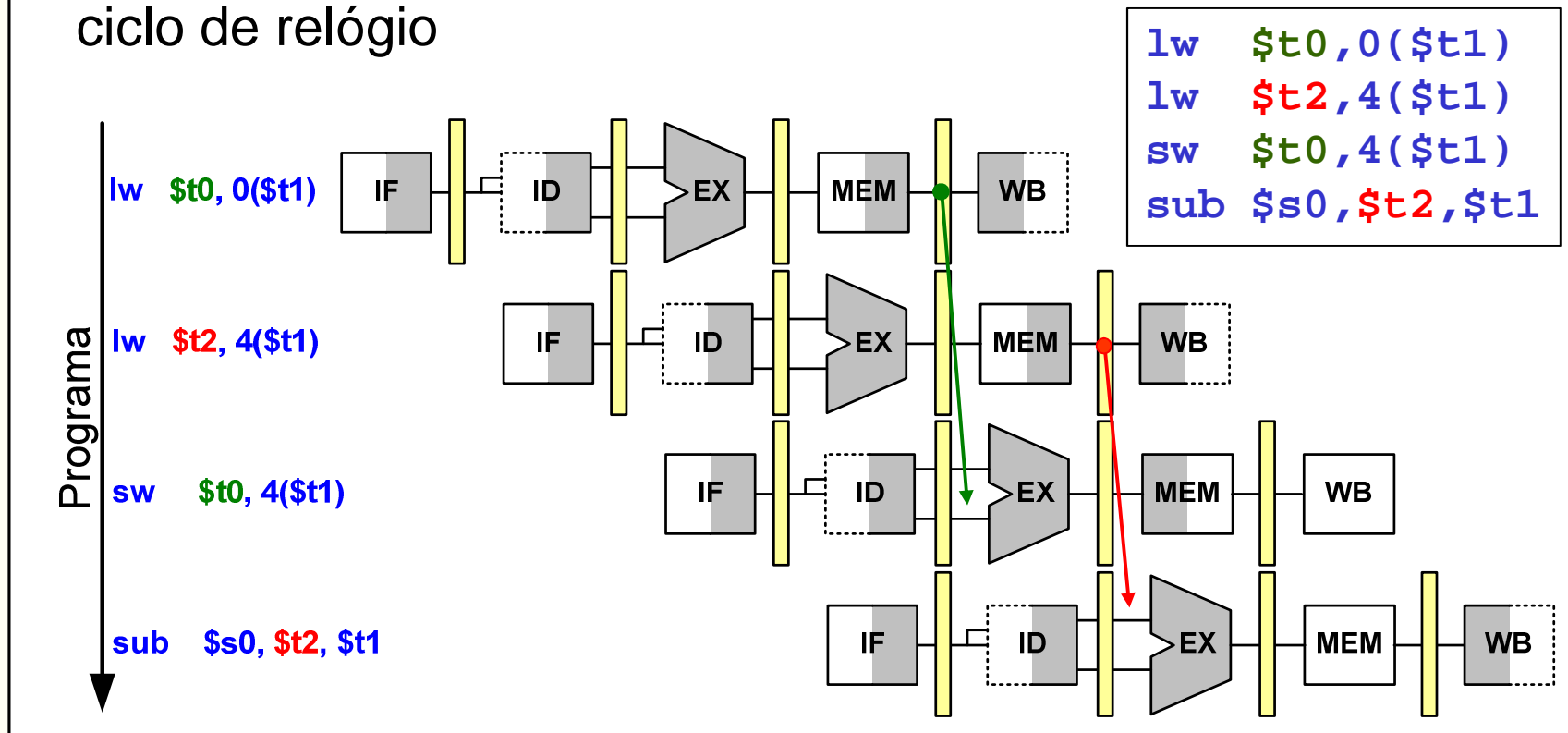
Hazards de dados – exemplo que gera *stalling*

- Situação de *stalling* por *hazard* de dados

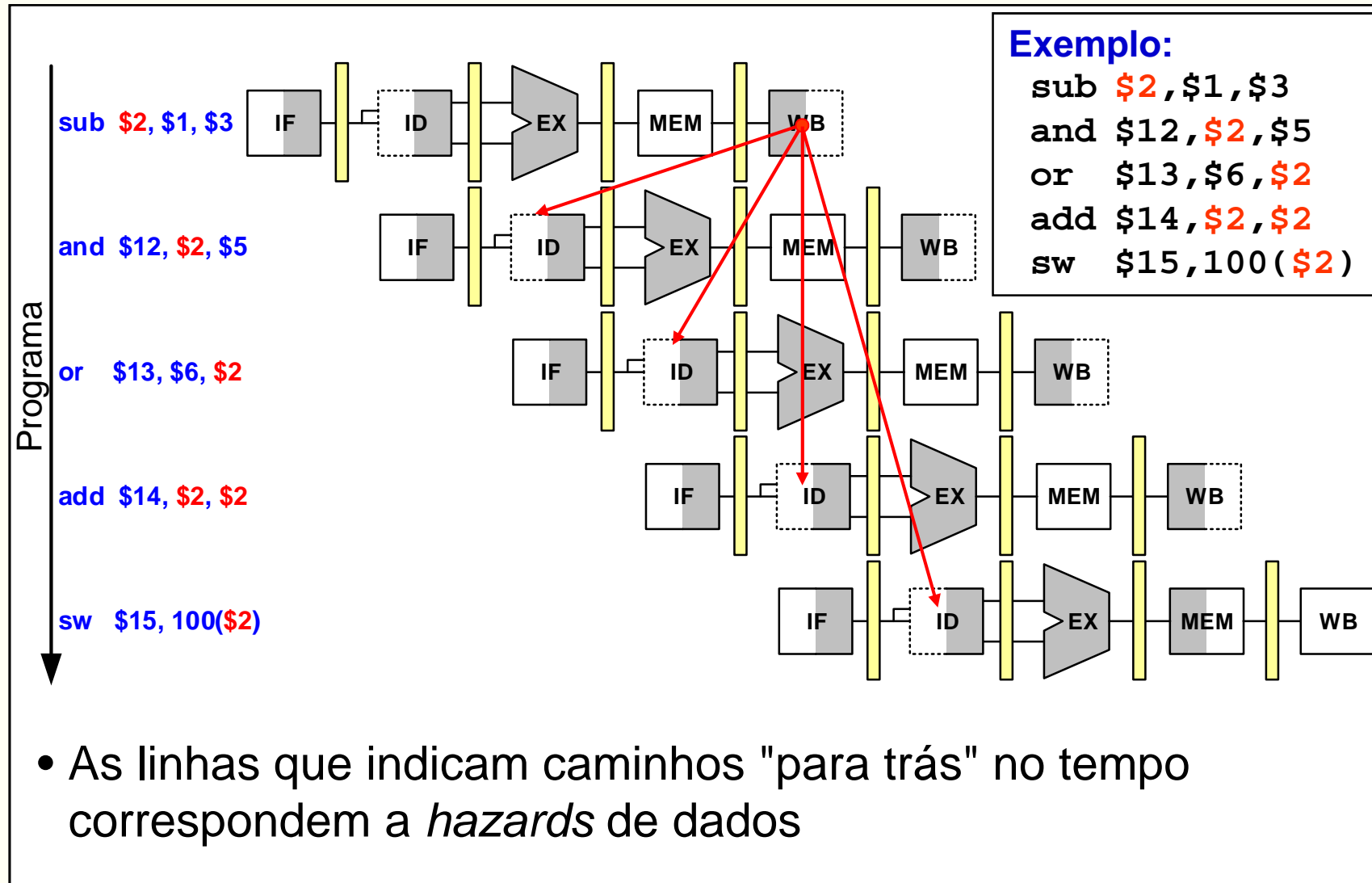


Hazards de dados

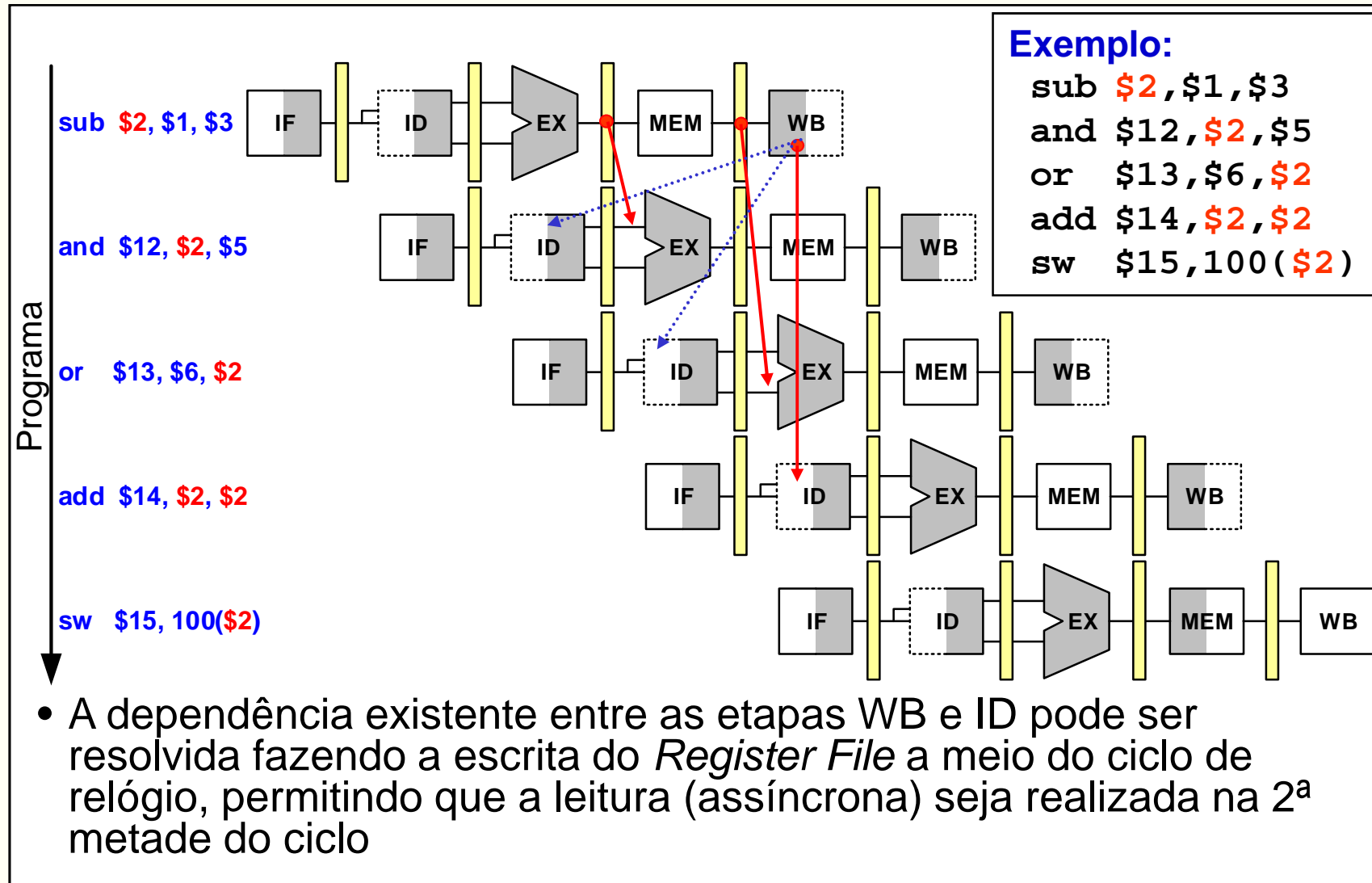
- A situação de *stalling* foi evitada pelo compilador/assembler através de reordenação. A reordenação gera um novo *hazard* de dados que é resolvido por *forwarding*
- A sequência de instruções reordenada executa em menos 1 ciclo de relógio



Hazards de dados – exemplo



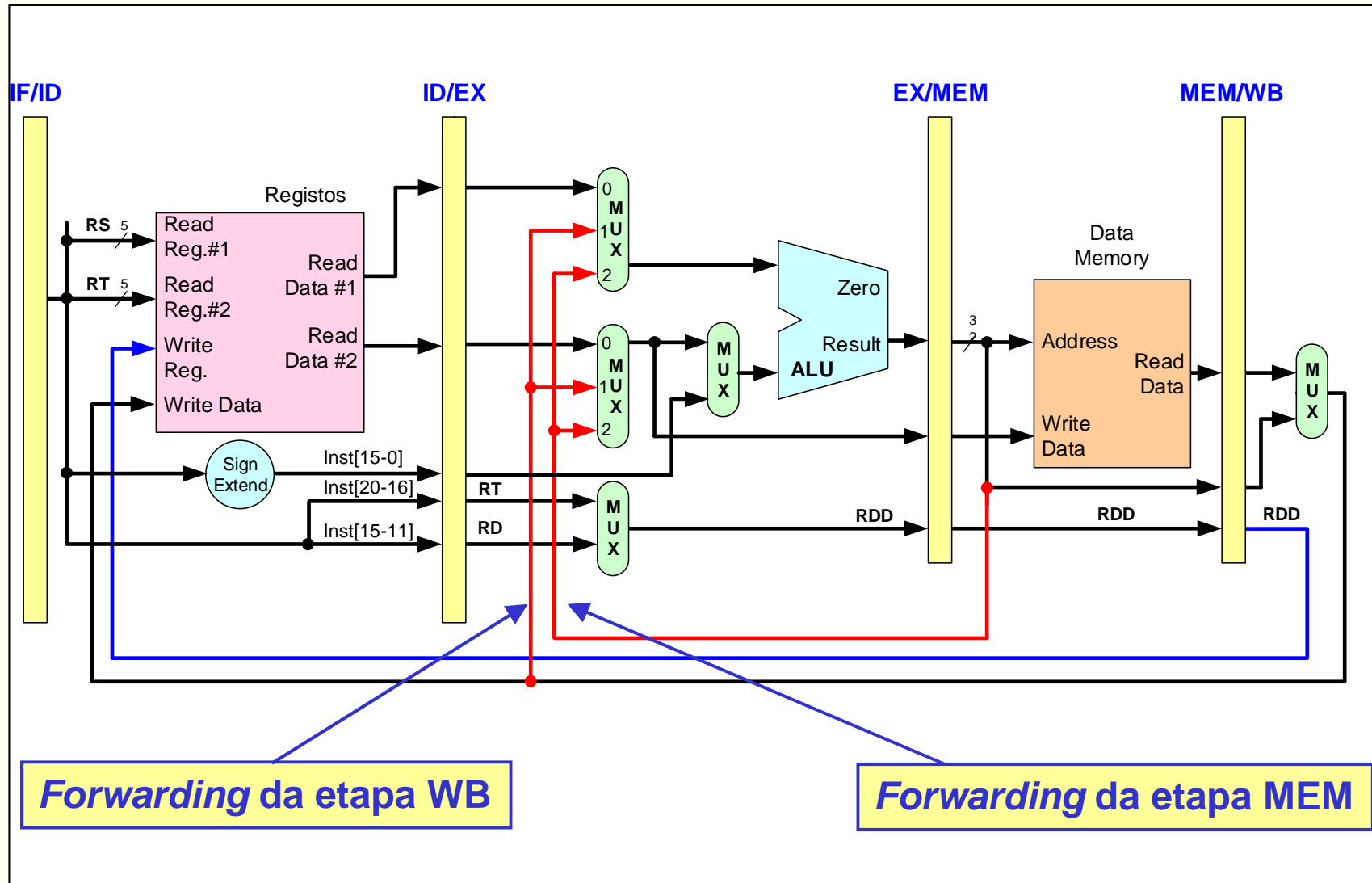
Hazards de dados – exemplo



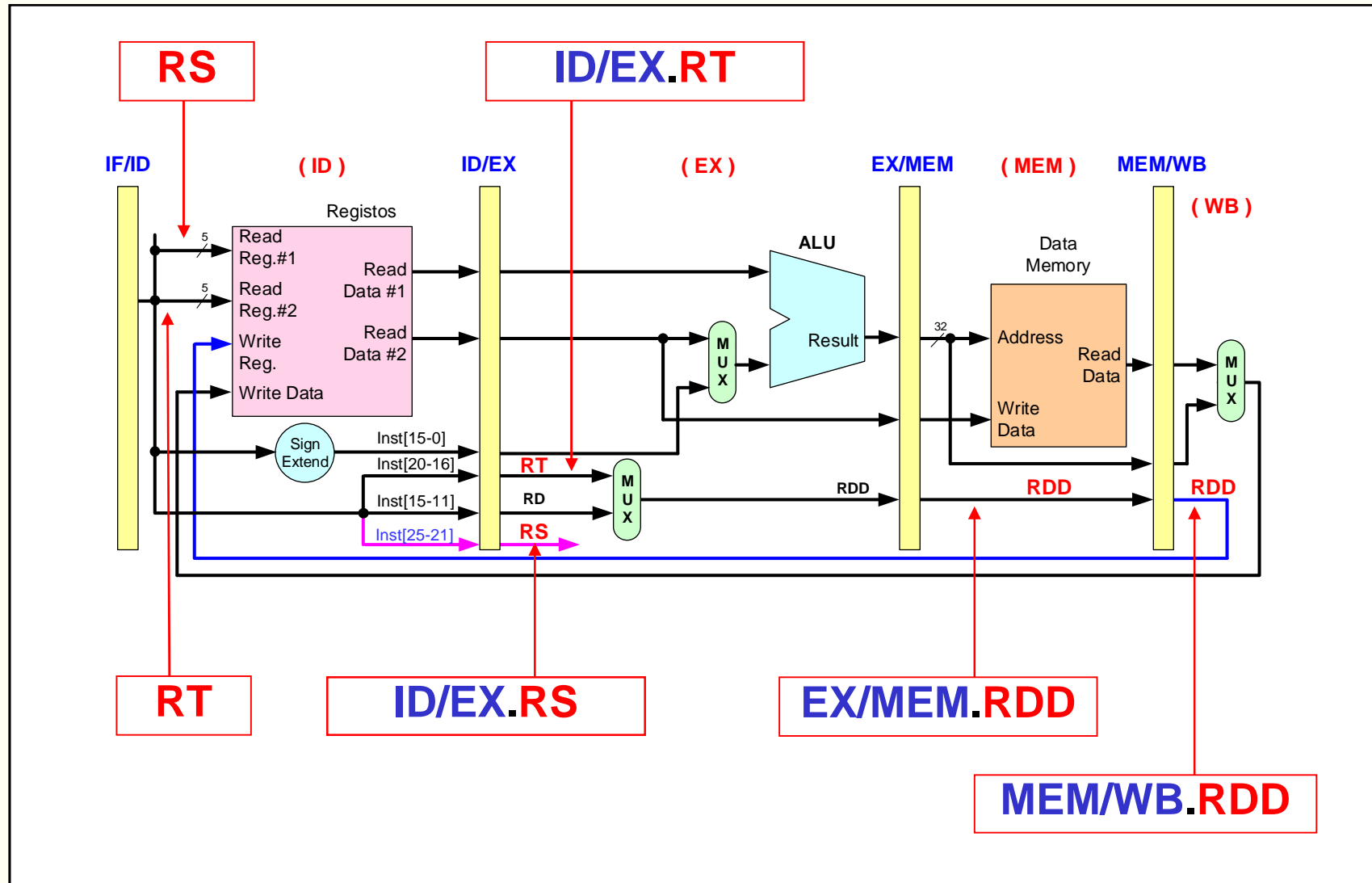
Hazards de dados – implementação do forwarding

- Para resolver um *hazard* de dados através de **forwarding** é necessário:
 - **Detetar** a situação de *hazard*
 - **Encaminhar** o valor ou os valores que se encontram em fases mais avançadas do *pipeline* (que ainda não foram escritos no registo destino) para onde eles são necessários
- À exceção das instruções de *branch*, a generalidade das outras instruções necessitam dos valores corretos dos registos na fase de execução (**EX**)
- Assim, a resolução de uma parte significativa dos *hazards* de dados resolve-se encaminhando os valores que se encontram em fases mais avançadas do *pipeline* para as entradas da ALU (fase **EX**)

Hazards de dados – encaminhamento



Hazards de dados – deteção



Hazards de dados – deteção

- As situações, correspondentes a *hazard* de dados, em que há necessidade de encaminhar valores para a fase **EX** são:

- Instrução na fase **MEM** cujo registo destino é um registo operando de uma instrução que se encontra na fase **EX**; de forma simplificada:

EX/MEM.RDD == **ID/EX.RS**, e/ou

EX/MEM.RDD == **ID/EX.RT**

M add **\$1**, \$2, \$3

EX sub \$4, **\$1**, \$5

- Instrução na fase **WB** cujo registo destino é um registo operando de uma instrução que se encontra na fase **EX**; de forma simplificada:

MEM/WB.RDD == **ID/EX.RS**, e/ou

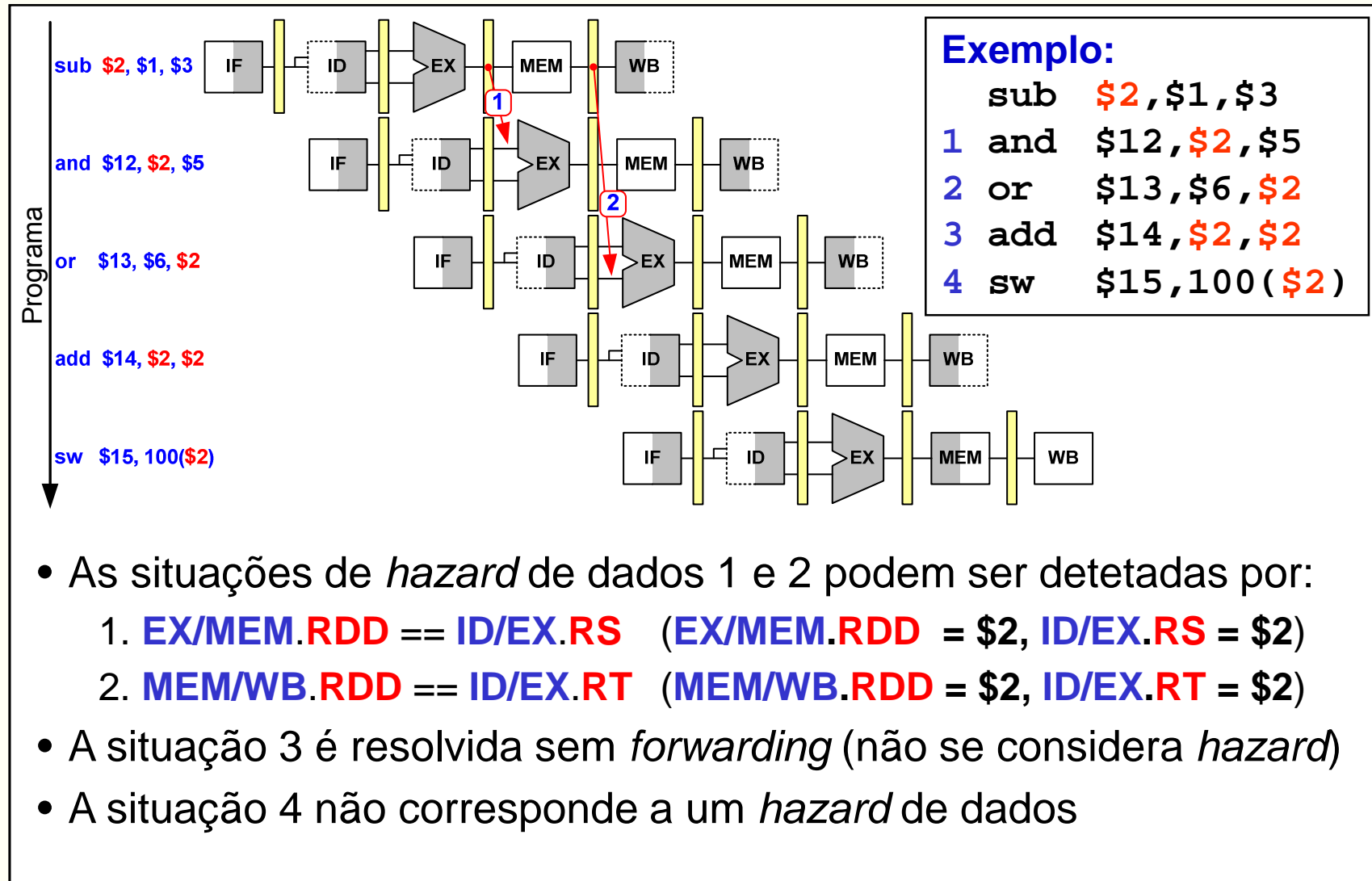
MEM/WB.RDD == **ID/EX.RT**

WB add **\$1**, \$2, \$3

M add \$6, \$2, \$3

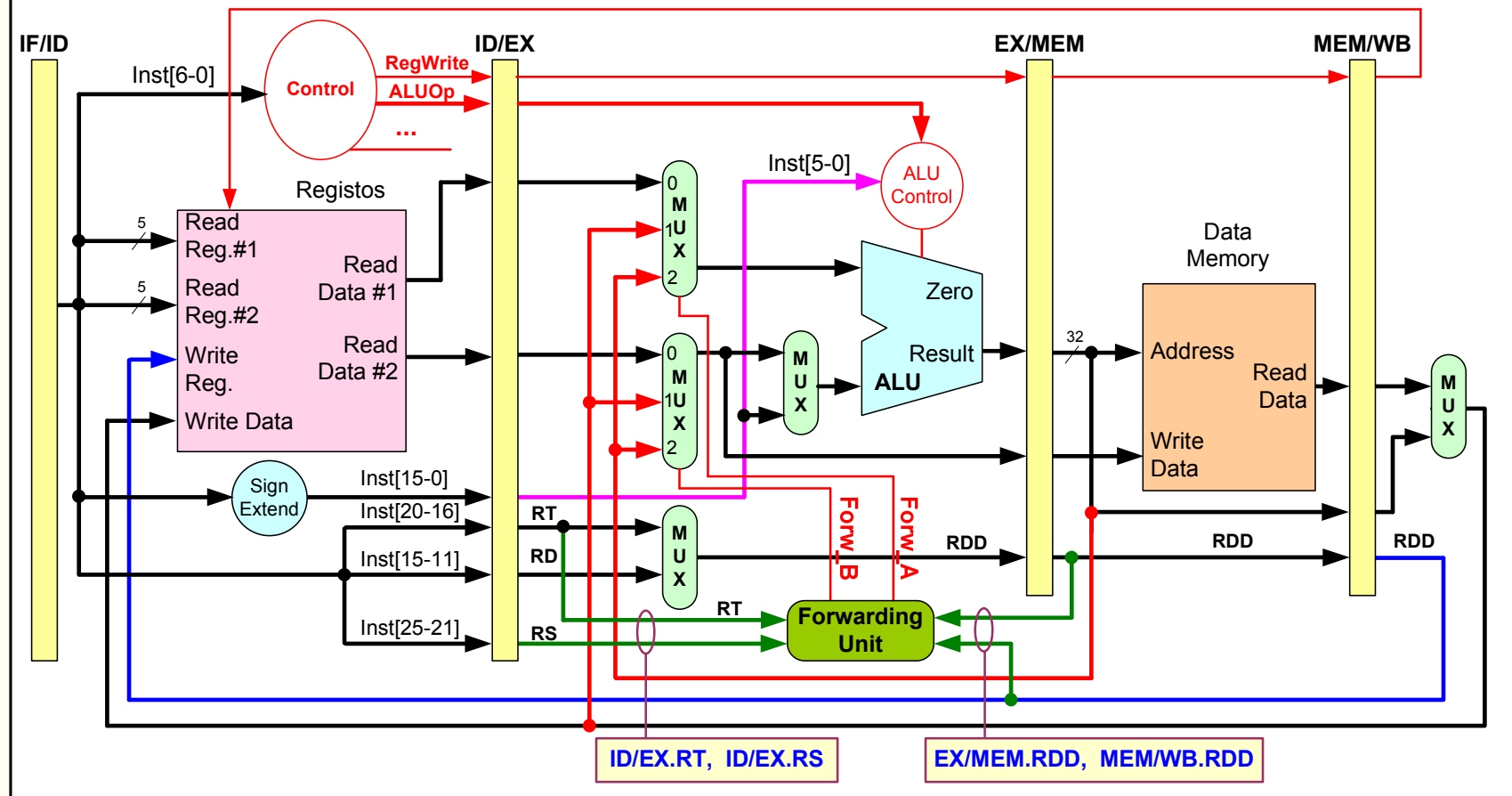
EX sub \$4, \$5, **\$1**

Hazards de dados – detecção



Hazards de dados – unidade de controlo de *forwarding*

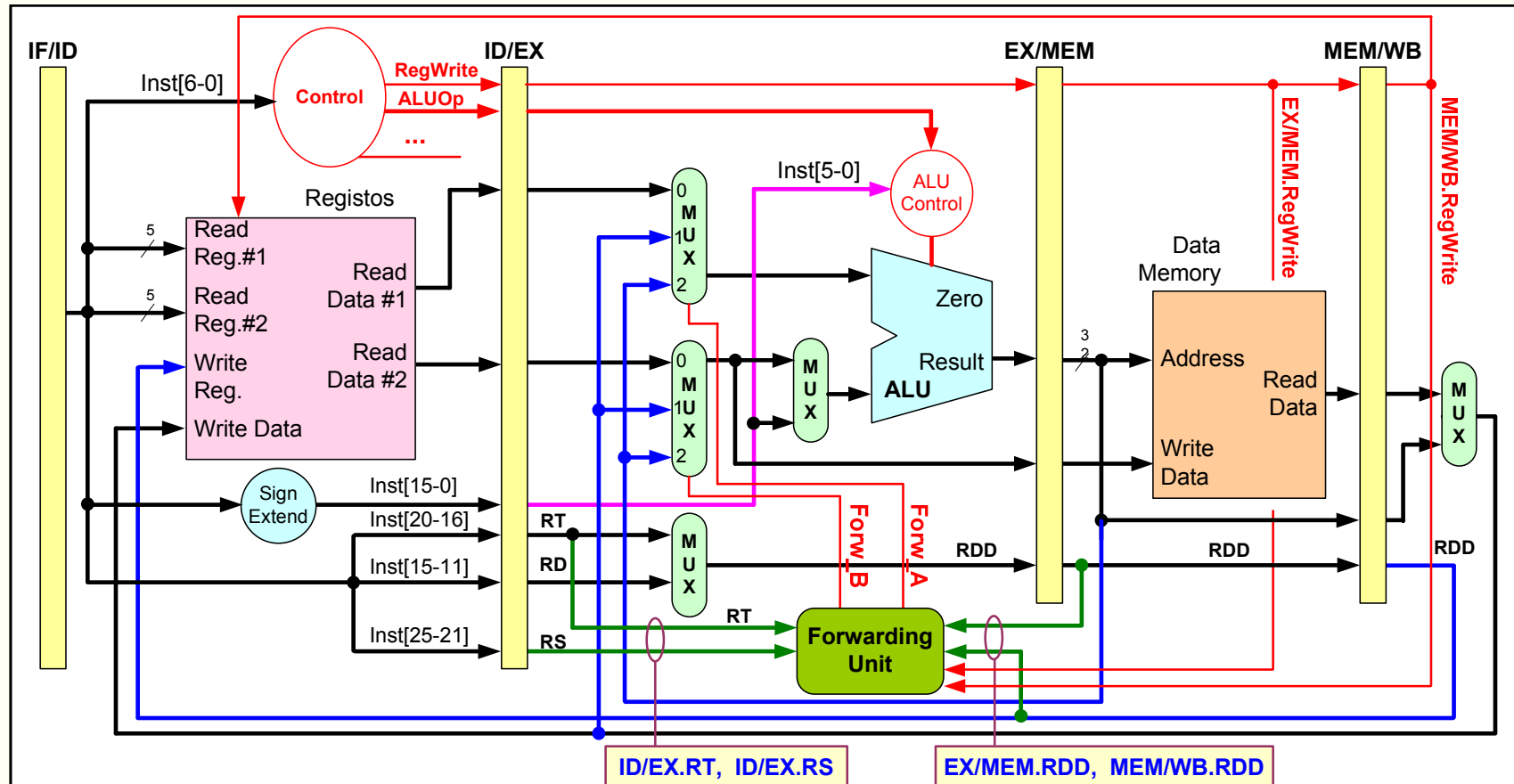
- Unidade de controlo de *forwarding*, simplificada



Hazards de dados – unidade de controlo de forwarding

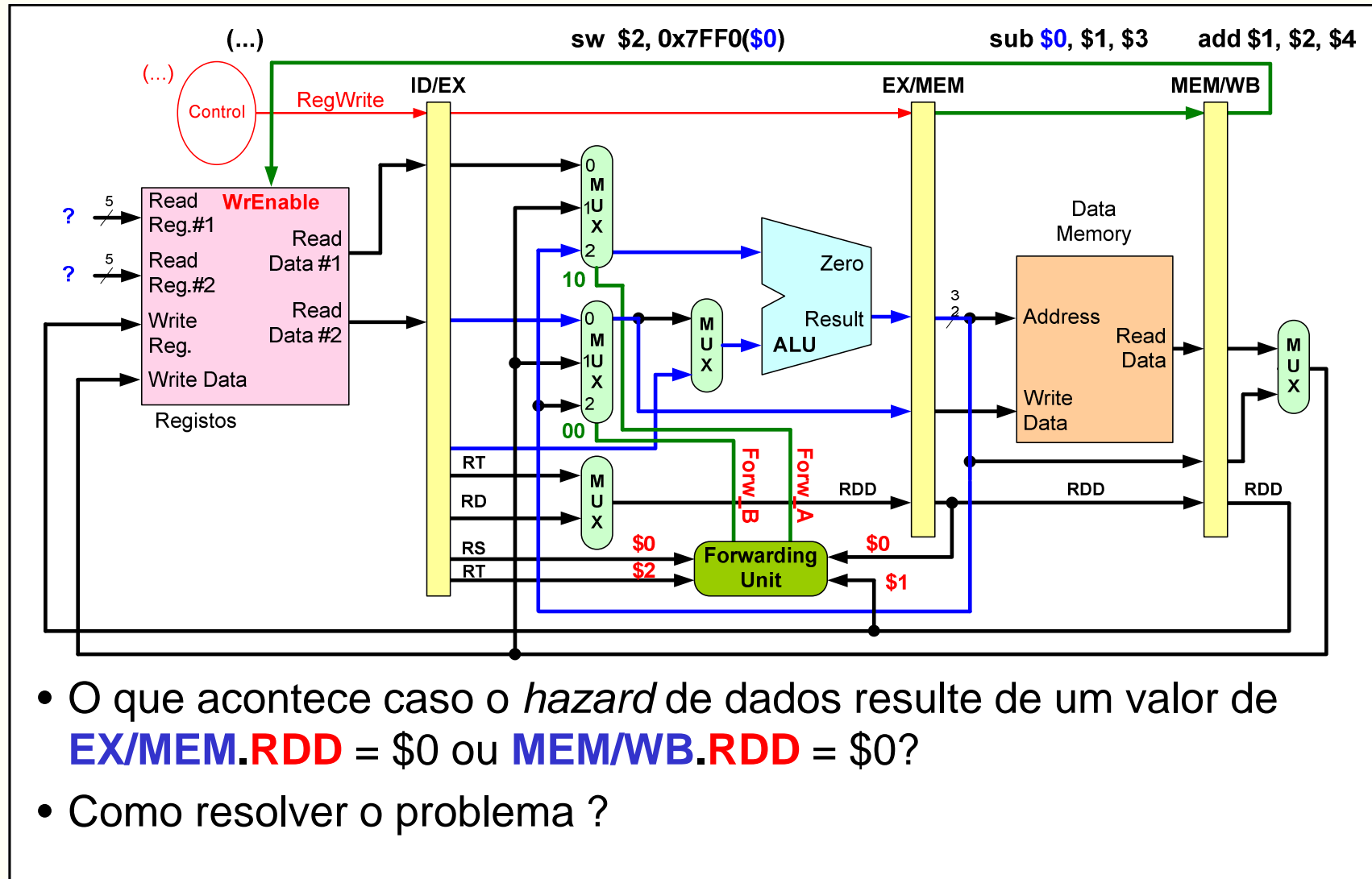
- A simples comparação dos registos não é suficiente para a correta deteção das situações de *hazard* de dados
- O sinal de controlo que permite a escrita no banco de registos (RegWrite) tem igualmente que ser avaliado:
 - Instrução na fase **MEM** que escreve o resultado num registo (**RegWrite='1'**) igual ao registo operando de uma instrução que se encontra na fase EX:
(**EX/MEM.RegWrite == 1**) and (**EX/MEM.RDD == ID/EX.RS**)
e/ou
(**EX/MEM.RegWrite == 1**) and (**EX/MEM.RDD == ID/EX.RT**)
 - Instrução na fase **WB** que escreve o resultado num registo (**RegWrite='1'**) igual ao registo operando de uma instrução que se encontra na fase EX:
(**MEM/WB.RegWrite == 1**) and (**MEM/WB.RDD == ID/EX.RS**)
e/ou
(**MEM/WB.RegWrite == 1**) and (**MEM/WB.RDD == ID/EX.RT**)

Hazards de dados – unidade de controlo de forwarding



- A unidade de controlo de *forwarding* gera os sinais de seleção dos dois MUX:
 - 00 – encaminhar valor lido do banco de registos
 - 01 – encaminhar o valor proveniente do registo **MEM/WB** (de uma instrução em WB)
 - 10 – encaminhar o valor proveniente do registo **EX/MEM** (de uma instrução em MEM)

Hazards de dados – unidade de controlo de *forwarding*



- O que acontece caso o *hazard* de dados resulte de um valor de **EX/MEM.RDD = \$0** ou **MEM/WB.RDD = \$0**?
- Como resolver o problema ?

Unidade de controlo de *forwarding* (para EX) – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ForwardingUnit is
  port( ExMem_RegWrite : in std_logic;
        MemWb_RegWrite : in std_logic;
        IdEx_RS       : in std_logic_vector(4 downto 0);
        IdEx_RT       : in std_logic_vector(4 downto 0);
        ExMem_RDD      : in std_logic_vector(4 downto 0);
        MemWb_RDD      : in std_logic_vector(4 downto 0);
        Forw_A         : out std_logic_vector(1 downto 0);
        Forw_B         : out std_logic_vector(1 downto 0));
end ForwardingUnit;
```

Unidade de controlo de *forwarding* (para EX) – VHDL

```
architecture Behavioral of ForwardingUnit is
begin
  process(all)
  begin
    Forw_A <= "00"; -- Op1 comes from Register File
    Forw_B <= "00"; -- Op2 comes from Register File

    if(MemWb_RegWrite = '1' and MemWb_RDD /= "00000") then
      if(MemWb_RDD = IdEx_RS) then Forw_A <= "01";end if;
      if(MemWb_RDD = IdEx_RT) then Forw_B <= "01";end if;
    end if;

    if(ExMem_RegWrite = '1' and ExMem_RDD /= "00000") then
      if(ExMem_RDD = IdEx_RS) then Forw_A <= "10";end if;
      if(ExMem_RDD = IdEx_RT) then Forw_B <= "10";end if;
    end if;
  end process;
end Behavioral;
```

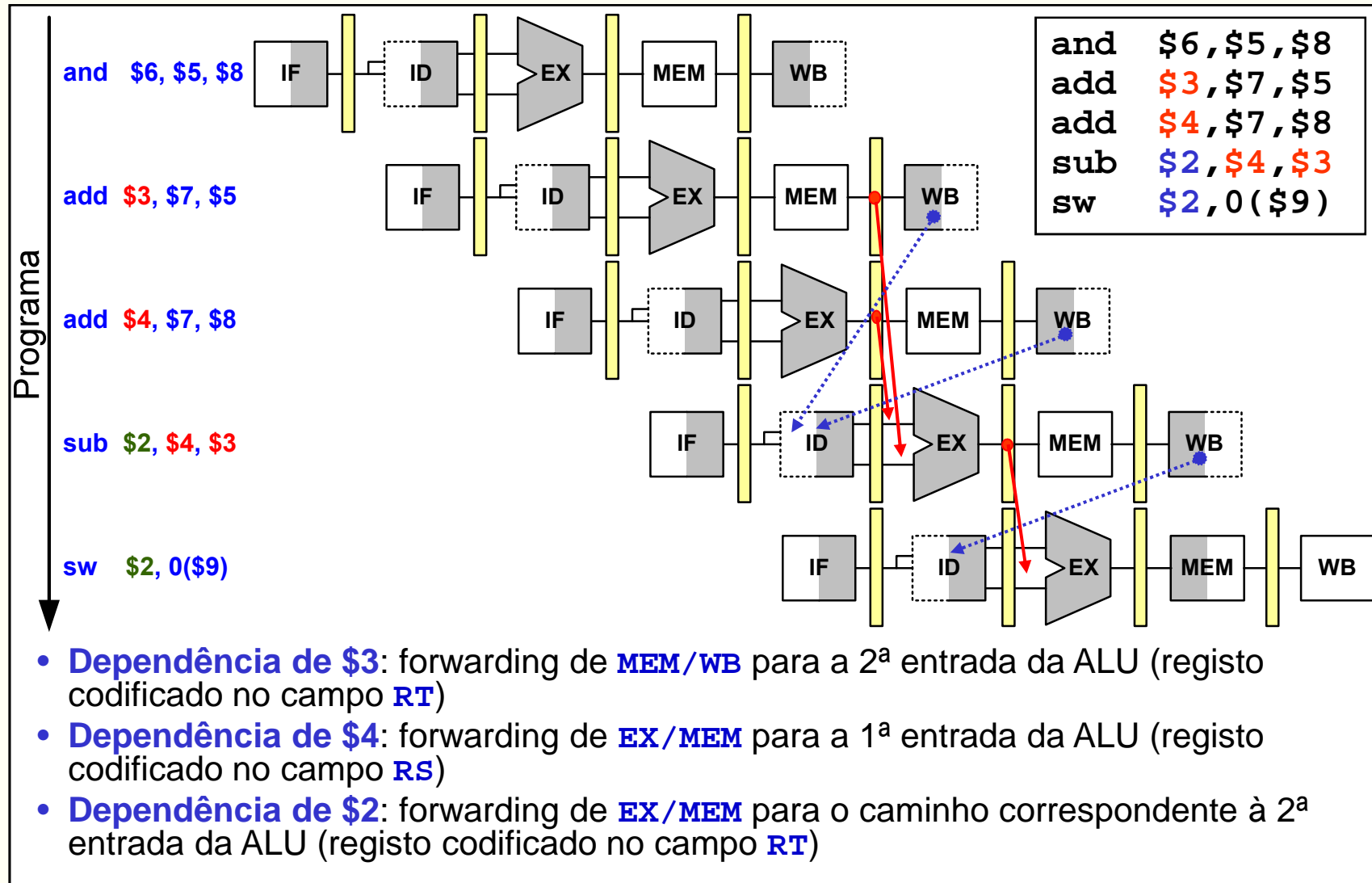
| | |
|----|--|
| 00 | – encaminhar valor lido do banco de registos |
| 01 | – encaminhar o valor proveniente do registo MEM/WB |
| 10 | – encaminhar o valor proveniente do registo EX/MEM |

Exemplo de *forwarding*

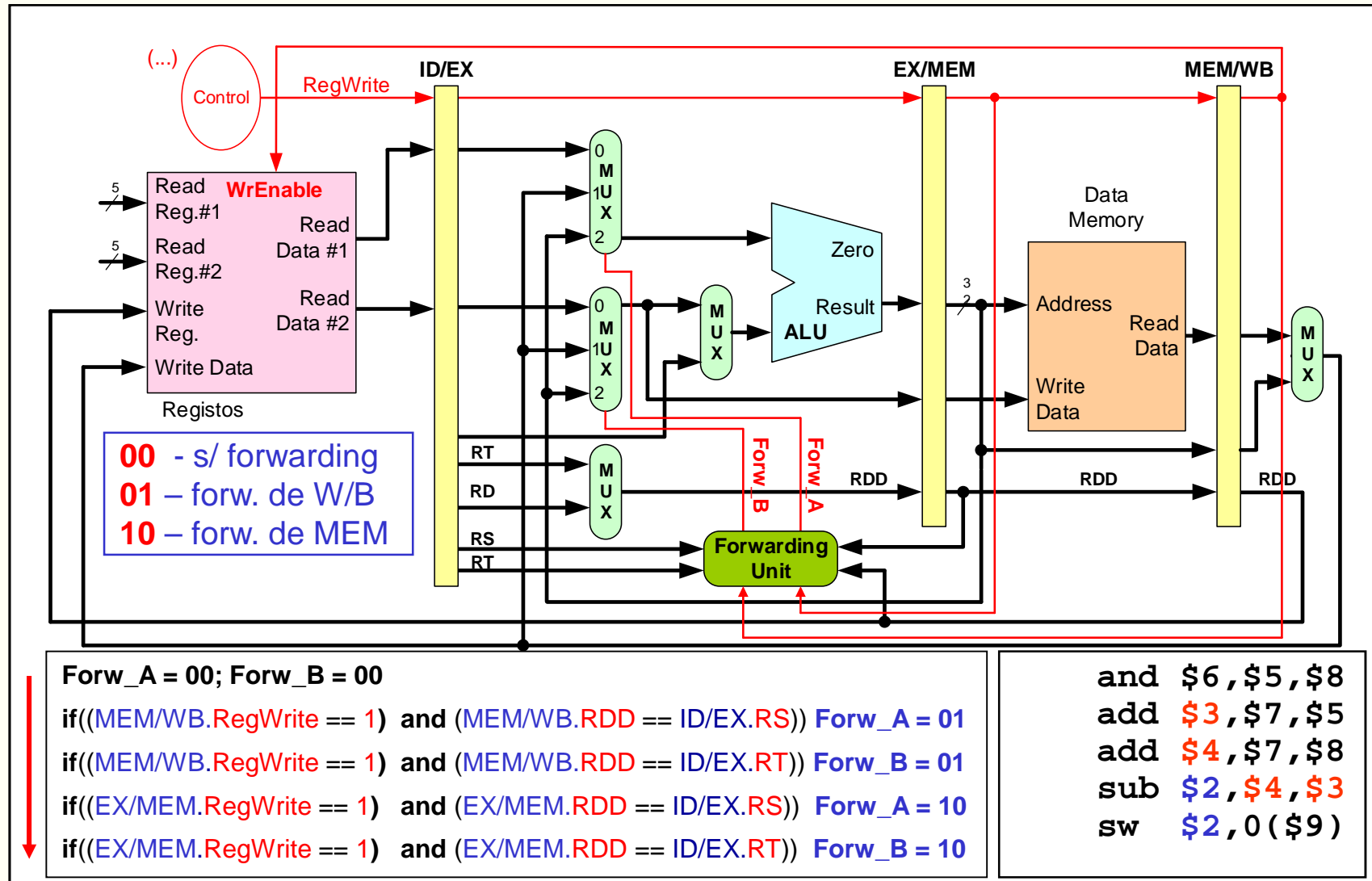
```
and  $6,$5,$8
add  $3,$7,$5
add  $4,$7,$8
sub  $2,$4,$3    # Hazard de dados: $3, $4
sw   $2,0($9)    # Hazard de dados: $2
```

- A instrução "**sub** \$2,\$4,\$3" apresenta duas situações de *hazards* de dados:
 - dependência do valor de \$4 (**add** \$4,\$7,\$8)
 - dependência de valor de \$3 (**add** \$3,\$7,\$5)
- A instrução "**sw** \$2,0(\$9)" apresenta igualmente uma situação de *hazard* de dados (dependência em \$2, **sub** \$2,\$4,\$3)

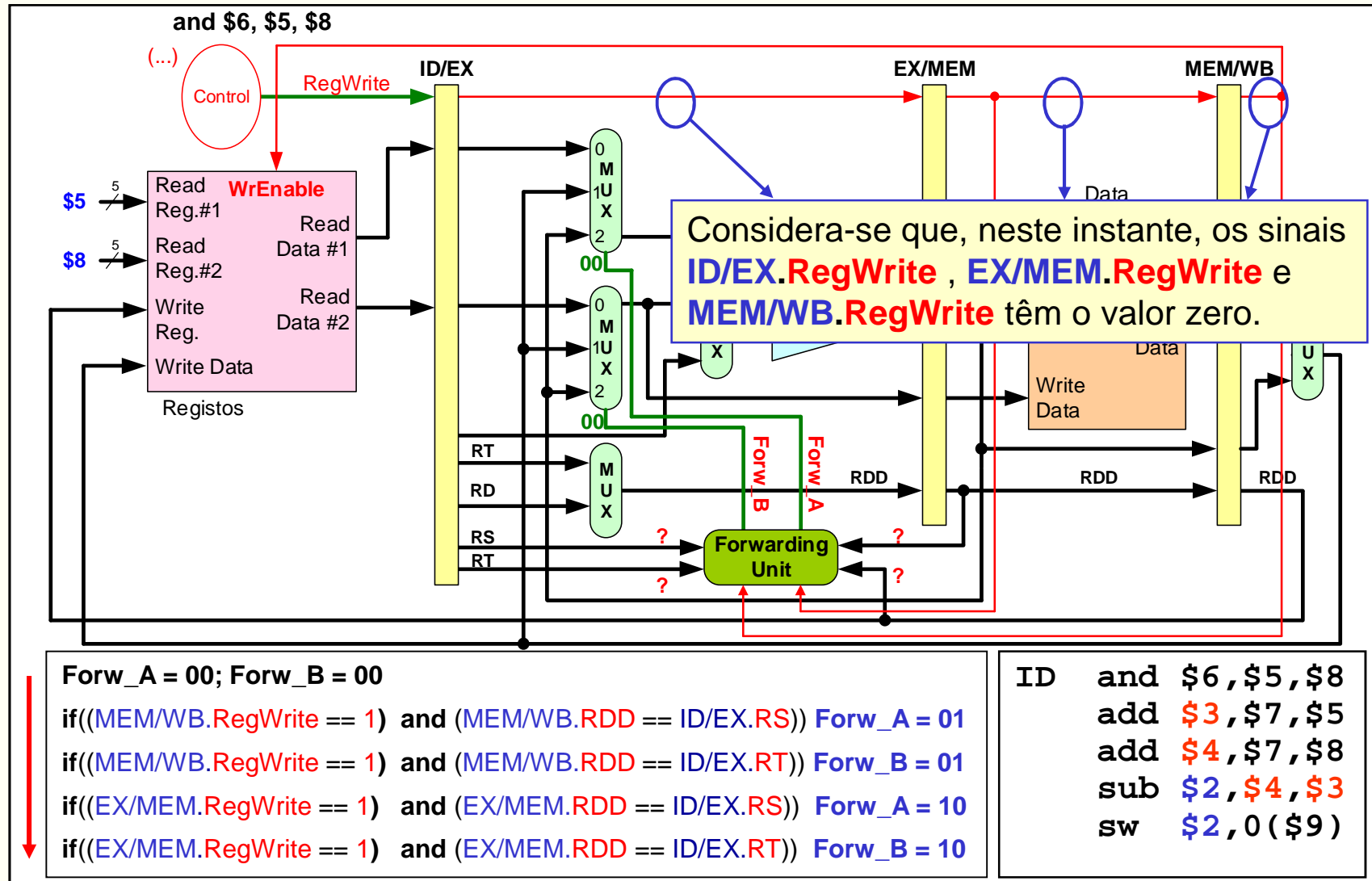
Exemplo de forwarding



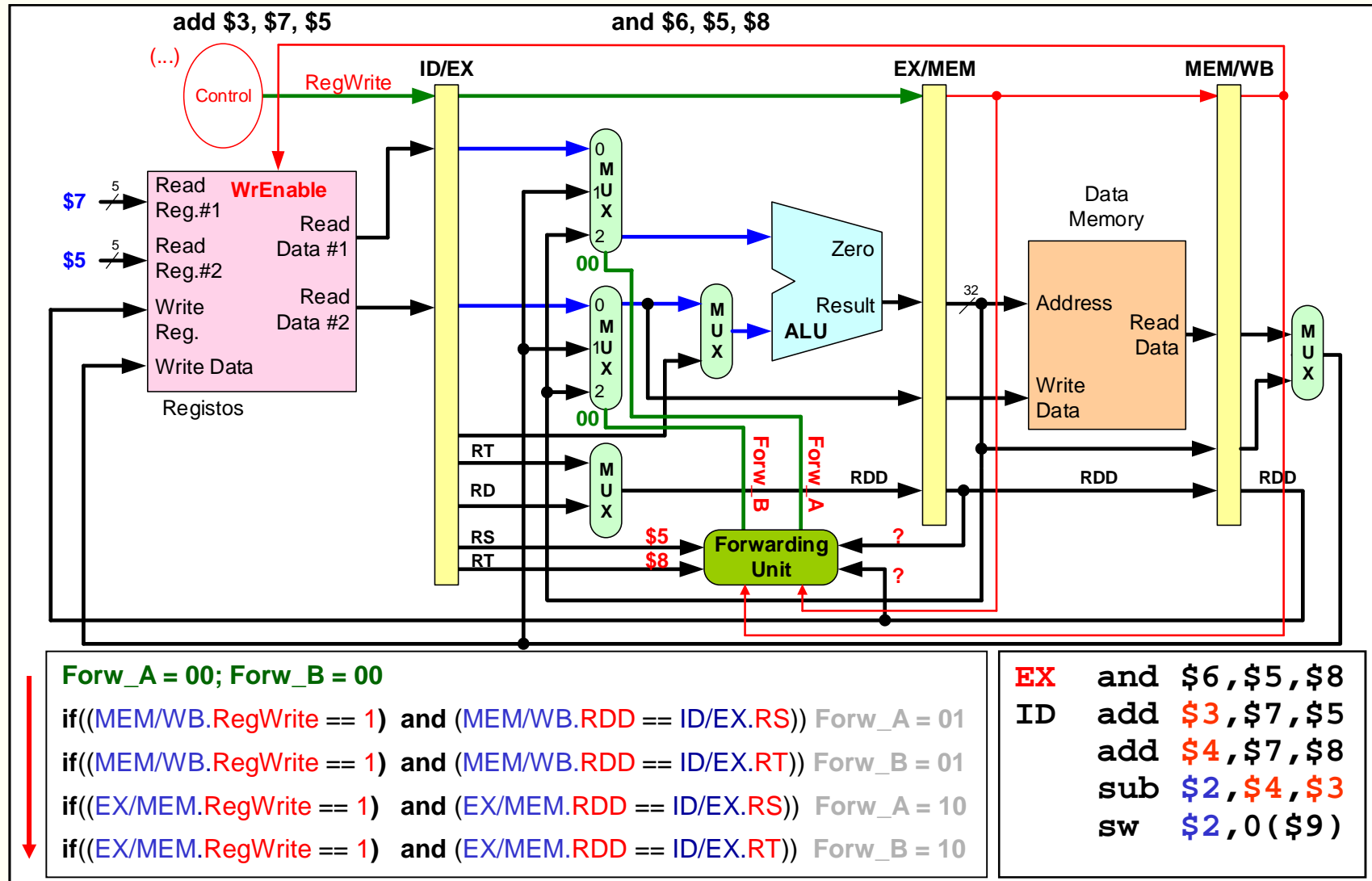
Exemplo de *forwarding*



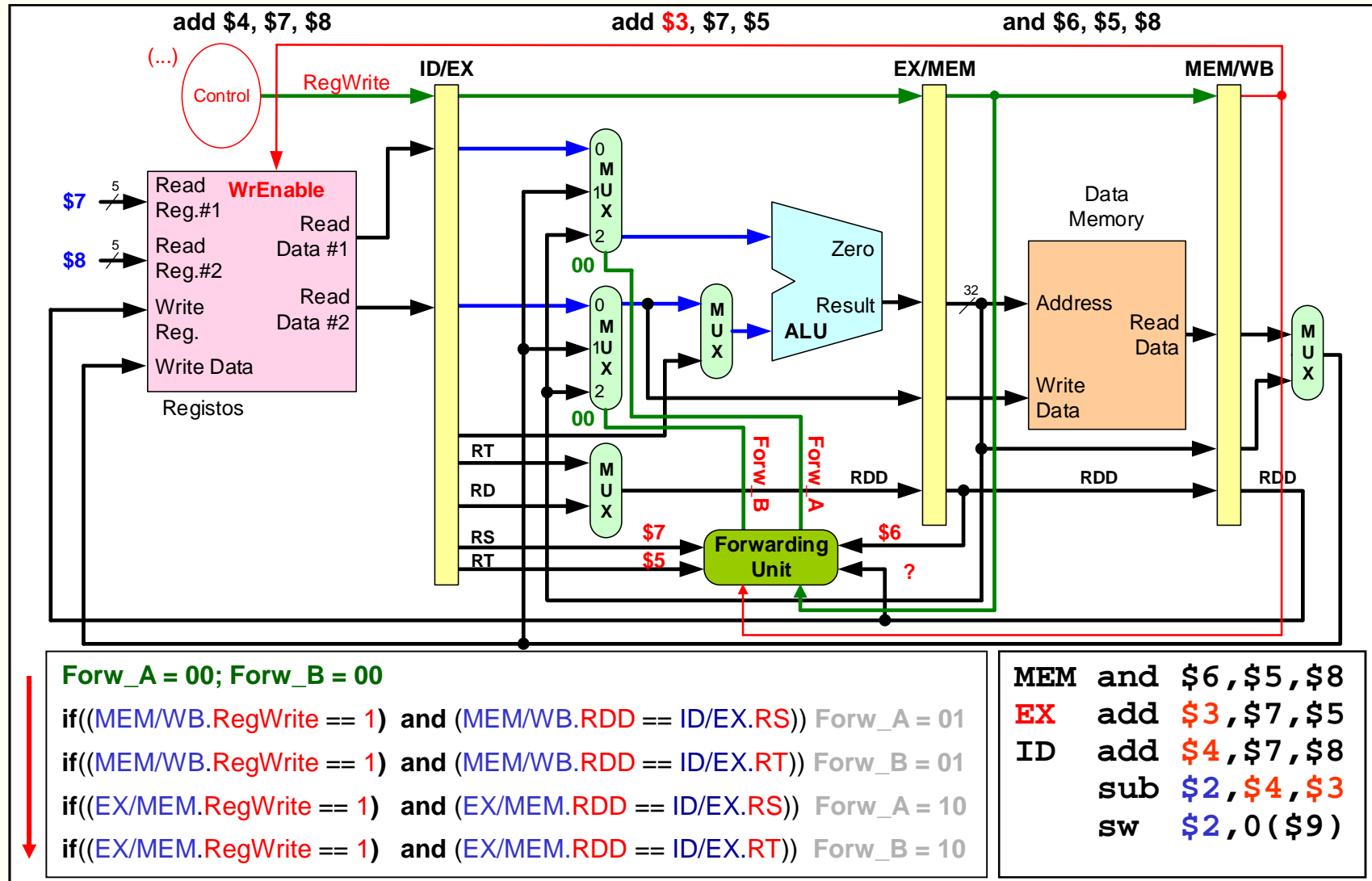
Exemplo de forwarding



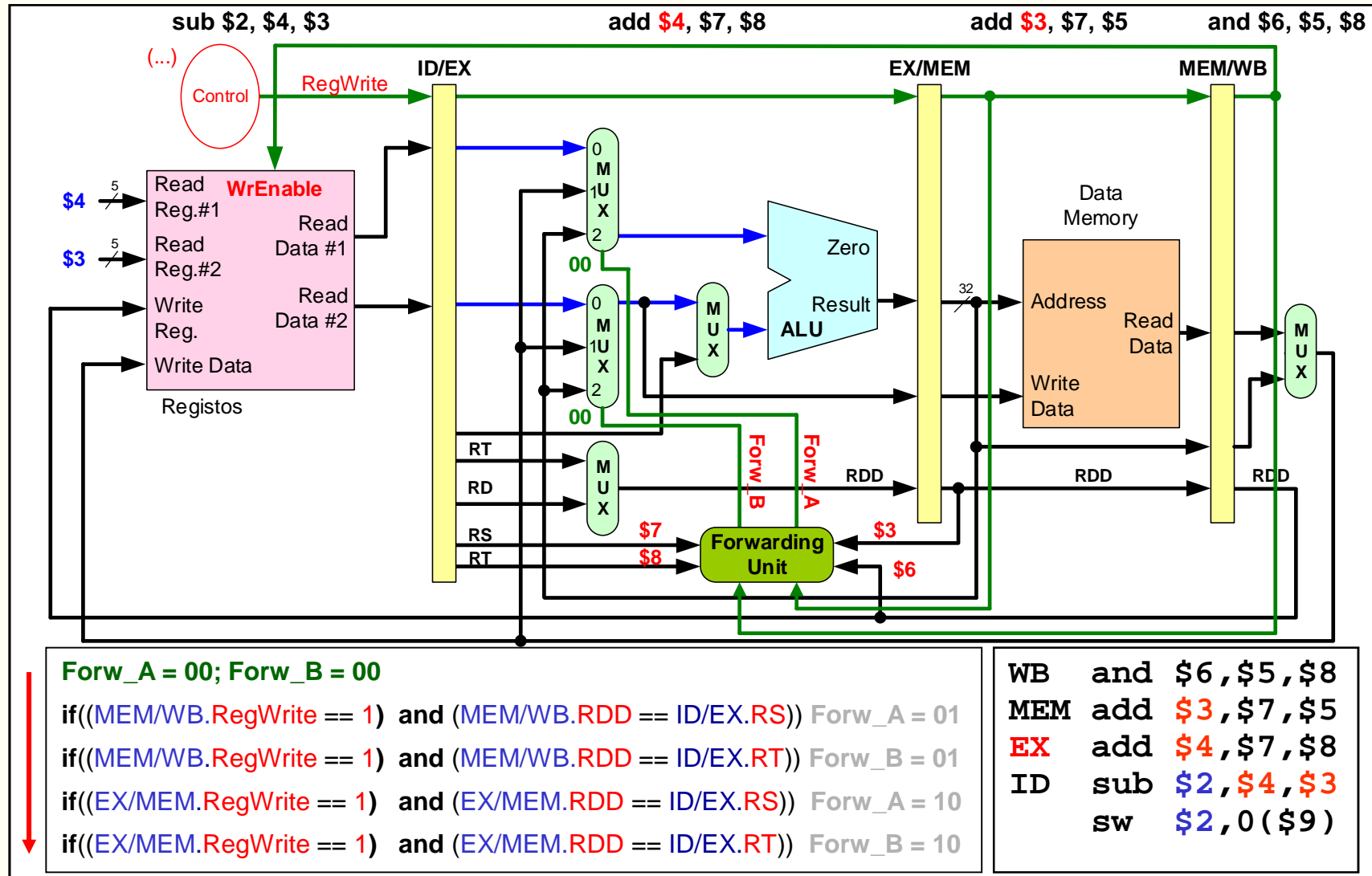
Exemplo de forwarding



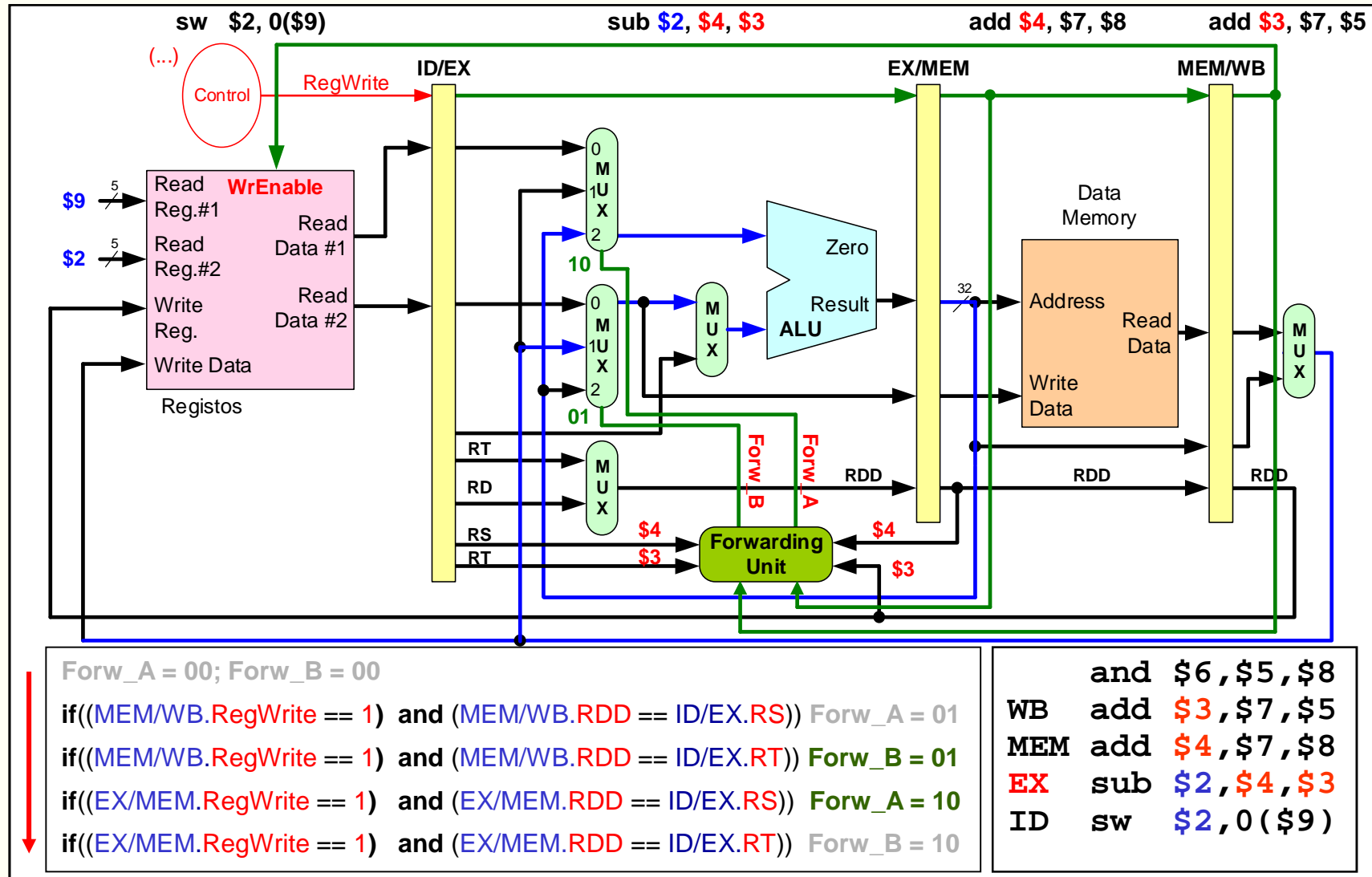
Exemplo de forwarding



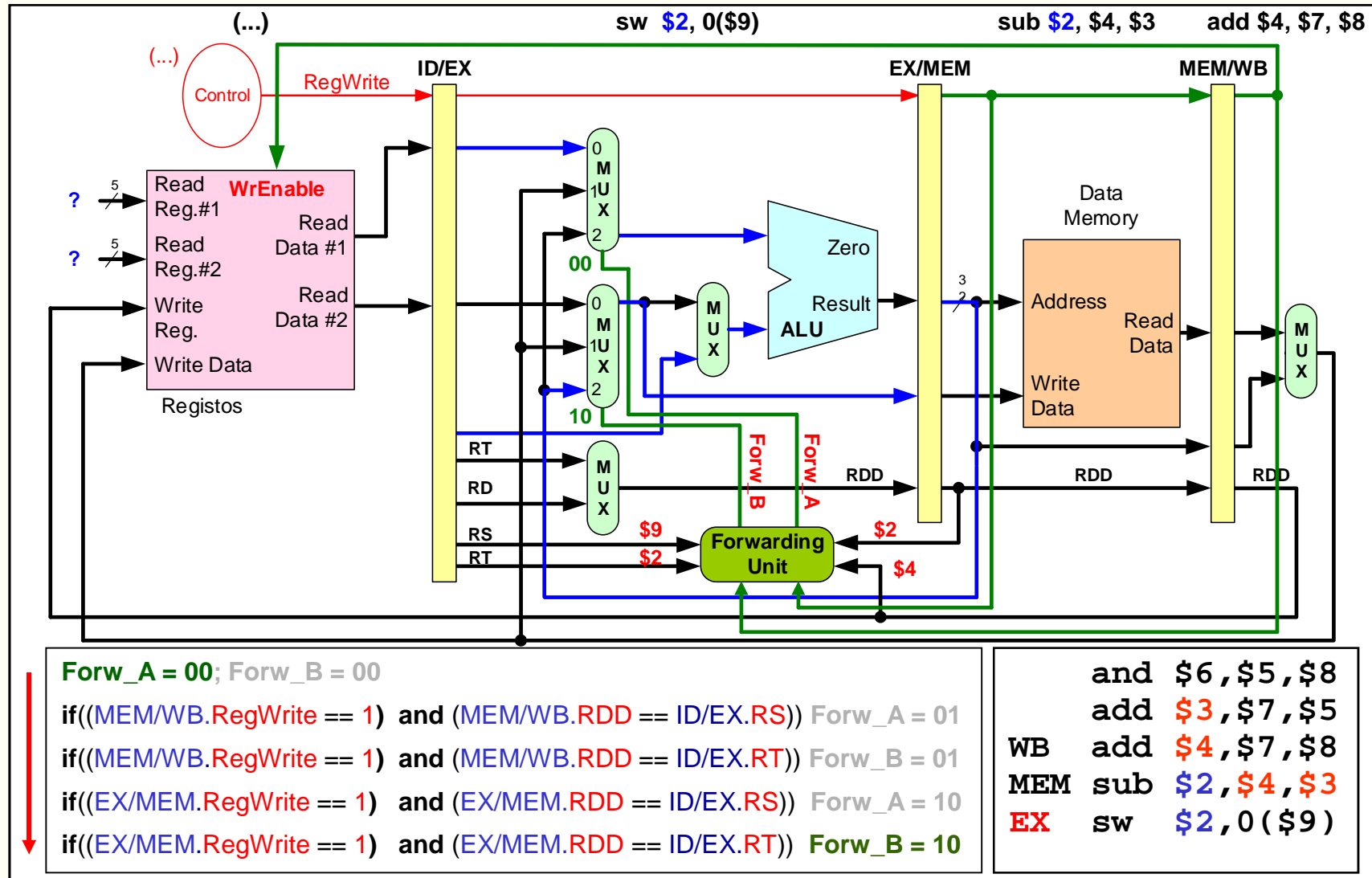
Exemplo de forwarding



Exemplo de forwarding



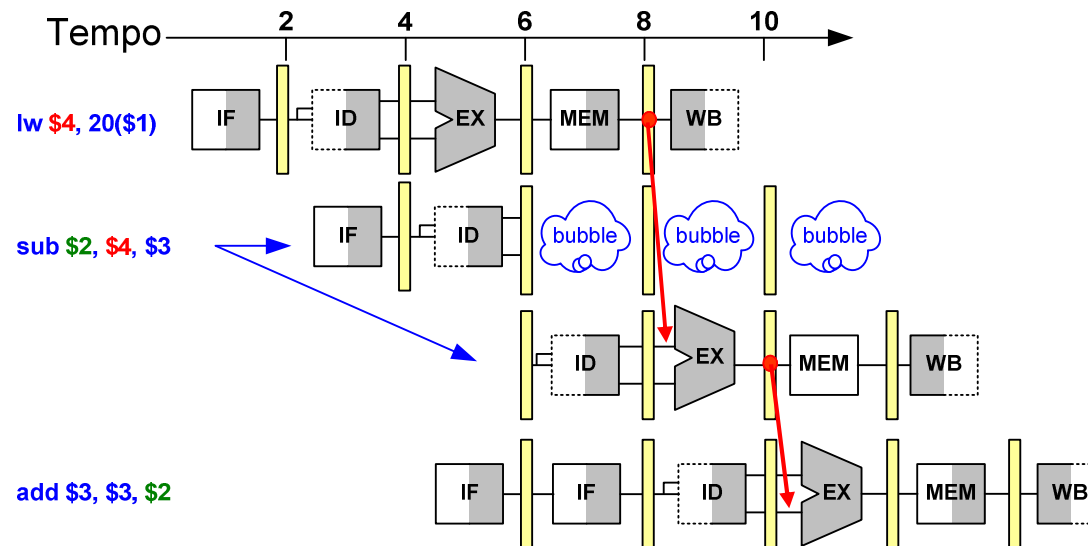
Exemplo de forwarding



Dependência que obriga a *stalling*

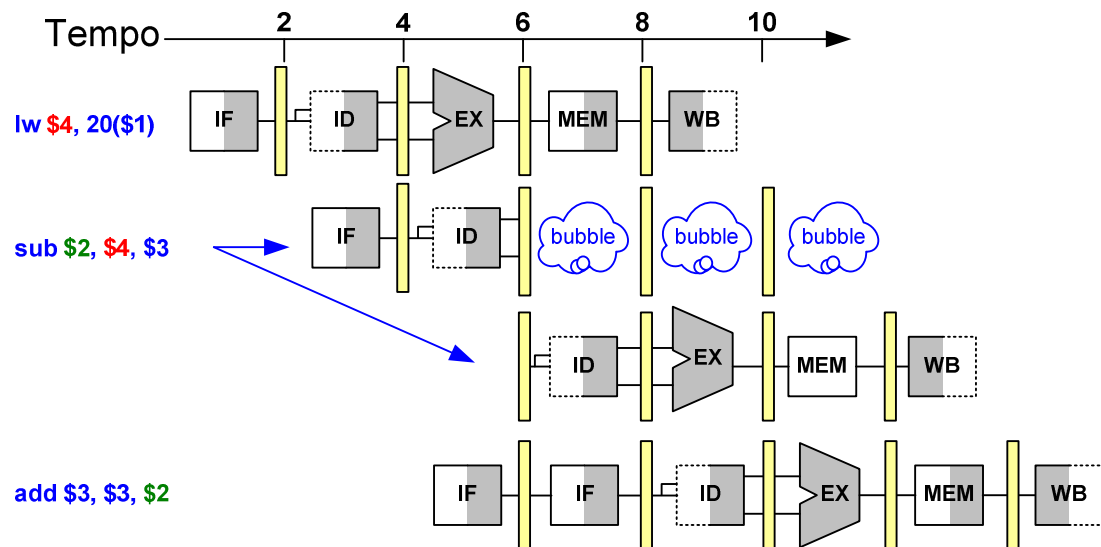
- Como já observado anteriormente, uma situação em que o *forwarding* não impede a ocorrência de *stalling* é a que resulta de uma instrução aritmética ou lógica executada a seguir e na dependência de uma instrução de load:

```
lw    $4, 20($1)  # valor disponível em WB
sub    $2, $4, $3  # Stall 1T, Forw. MEM/WB > EX
add    $3, $3, $2  # Forw. EX/MEM > EX
```



Dependência que obriga a *stalling*

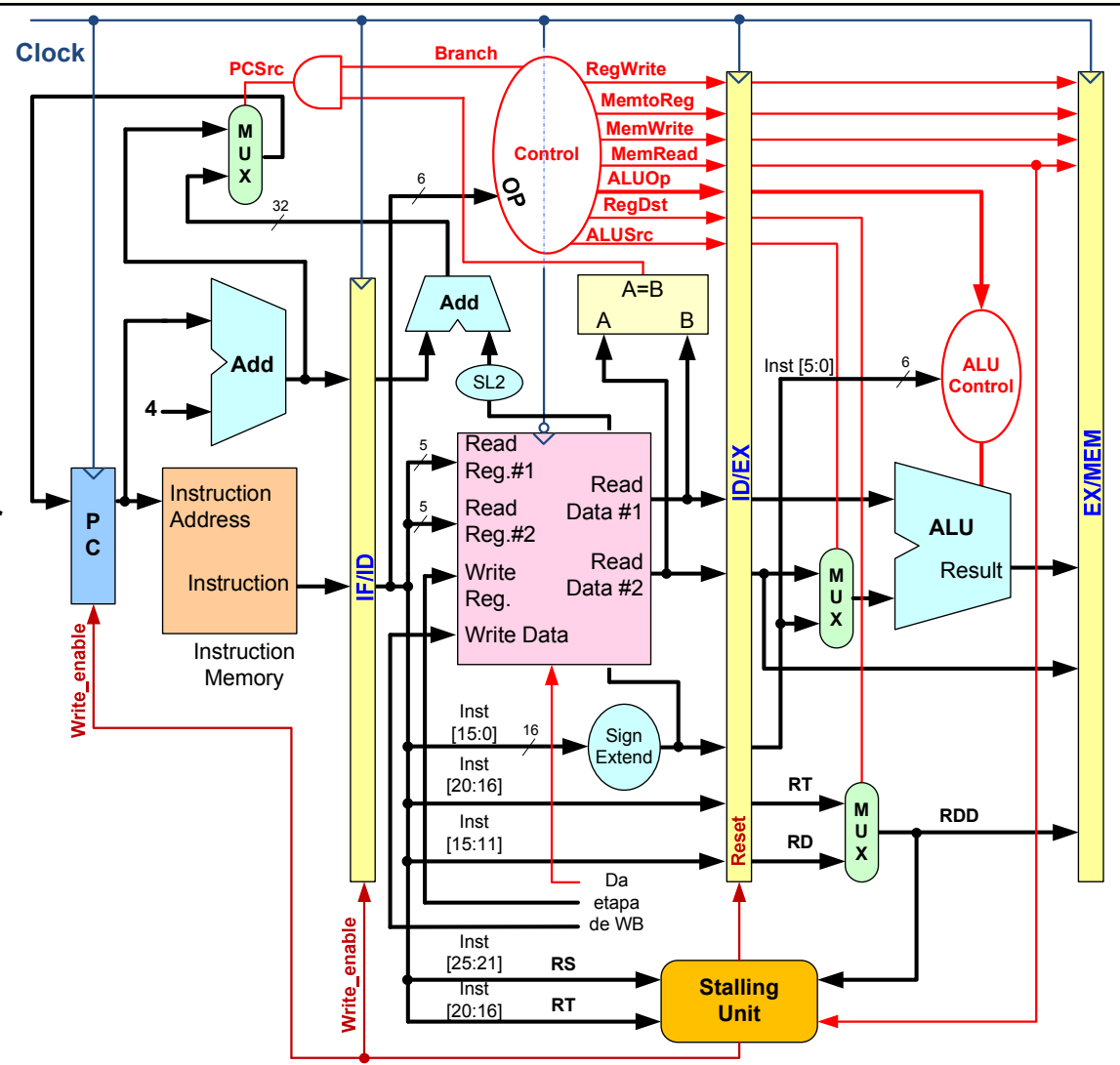
- A situação de *stalling* tem que ser desencadeada quando a instrução tipo R está na sua fase ID. Como fazer?
 - Inserir ***bubble*** na etapa **EX**: fazer o *reset* síncrono do registo **ID/EX**
 - Congelar, durante 1 ciclo de relógio, as etapas **IF** e **ID** (i.e. impedir a escrita no registo **IF/ID** e impedir que seja feita a atualização do PC)
- Como detetar?



(ID/EX.MemRead == 1) and (ID/EX.RDD == RS or ID/EX.RDD == RT)

Unidade de *stalling*

- Inserir *bubble* na etapa EX: fazer o reset síncrono do registo ID/EX
- Congelar, durante 1 ciclo de relógio, as etapas IF e ID (i.e. impedir a escrita no registo IF/ID e impedir que seja feita a atualização do PC)



Unidade de controlo de *stalling* – VHDL

```
library ieee;
use ieee.std_logic_1164.all;

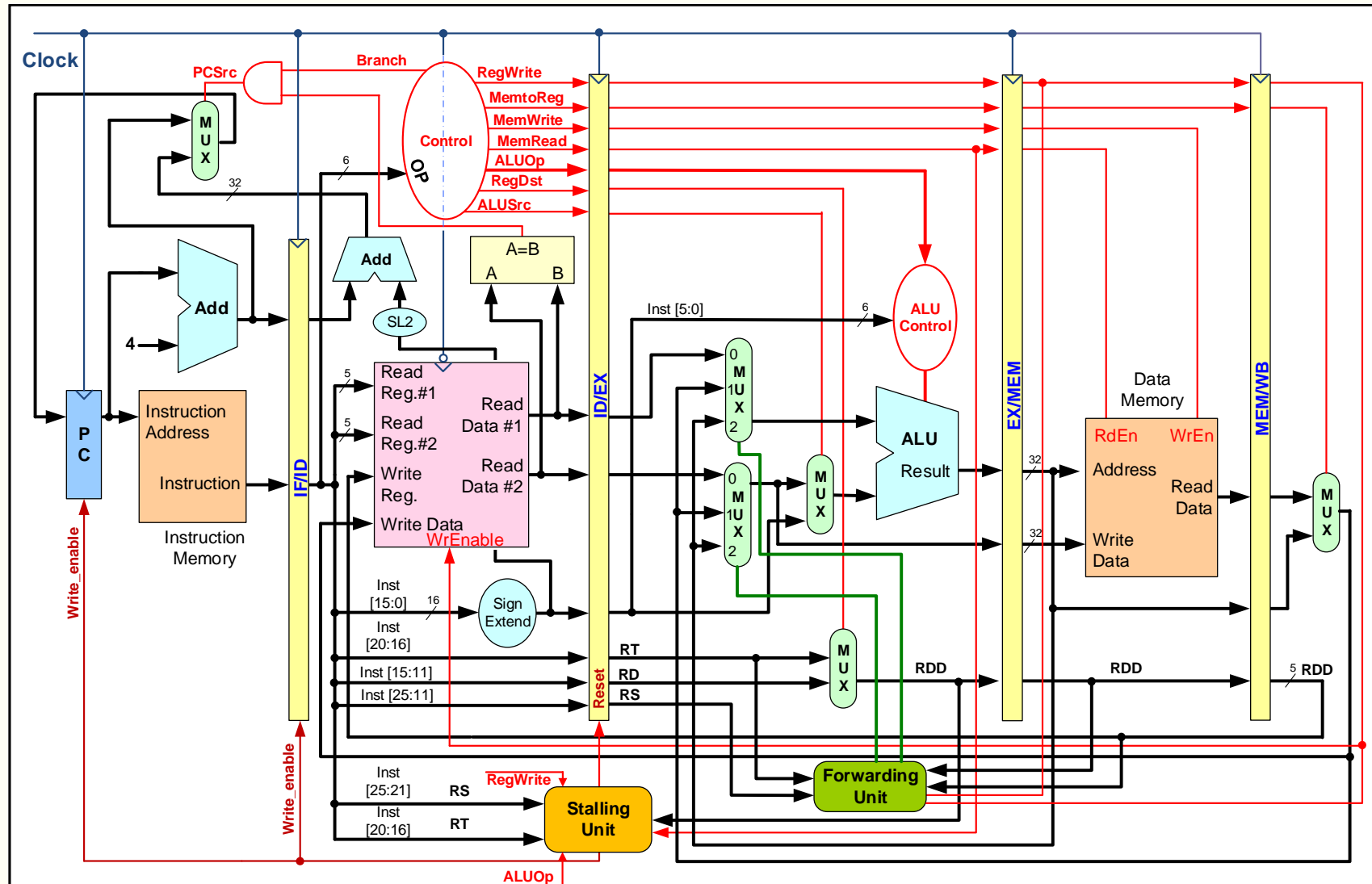
entity StallingUnit is
    port( RS          : in  std_logic_vector(4 downto 0);
          RT          : in  std_logic_vector(4 downto 0);
          RegWrite    : in  std_logic;
          ALUOp       : in  std_logic_vector(1 downto 0);
          IdEx_RDD     : in  std_logic_vector(4 downto 0);
          IdEx_MemRead : in  std_logic;
          Reset_IdEx   : out std_logic;
          Enable_PC    : out std_logic;
          Enable_IfId  : out std_logic);
end StallingUnit;
```

- Unidade de controlo de *stalling* simplificada, que contempla apenas a situação de dependência entre uma instrução LW e uma instrução tipo R (semelhante ao exemplo apresentado anteriormente)

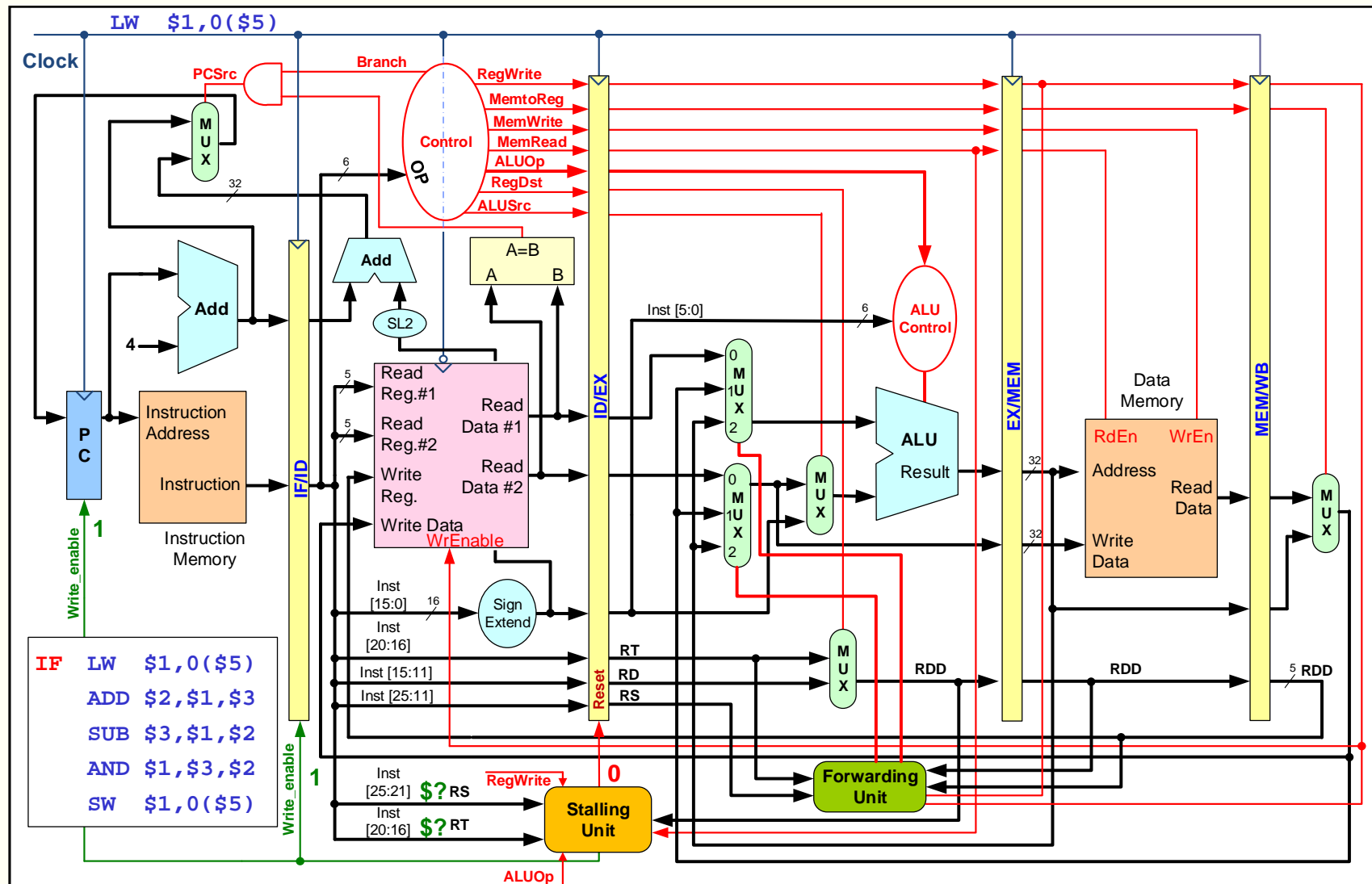
Unidade de controlo de *stalling* – VHDL

```
architecture Behavioral of StallingUnit is
begin
  process(all)
  begin
    Enable_PC      <= '1';  -- Normal flow
    Enable_IfId    <= '1';
    Reset_IdEx     <= '0';
    if(IdEx_MemRead = '1' and IdEx_RDD /= "00000") then
      if(RegWrite = '1' and ALUOp = "10") then
        if(IdEx_RDD = RS or IdEx_RDD = RT) then
          Enable_PC      <= '0';  -- Stall PC
          Enable_IfId    <= '0';  -- Stall IF/ID
          Reset_IdEx     <= '1';  -- Bubble in ID/EX
        end if;
      end if;
    end if;
  end process;
end Behavioral;
```

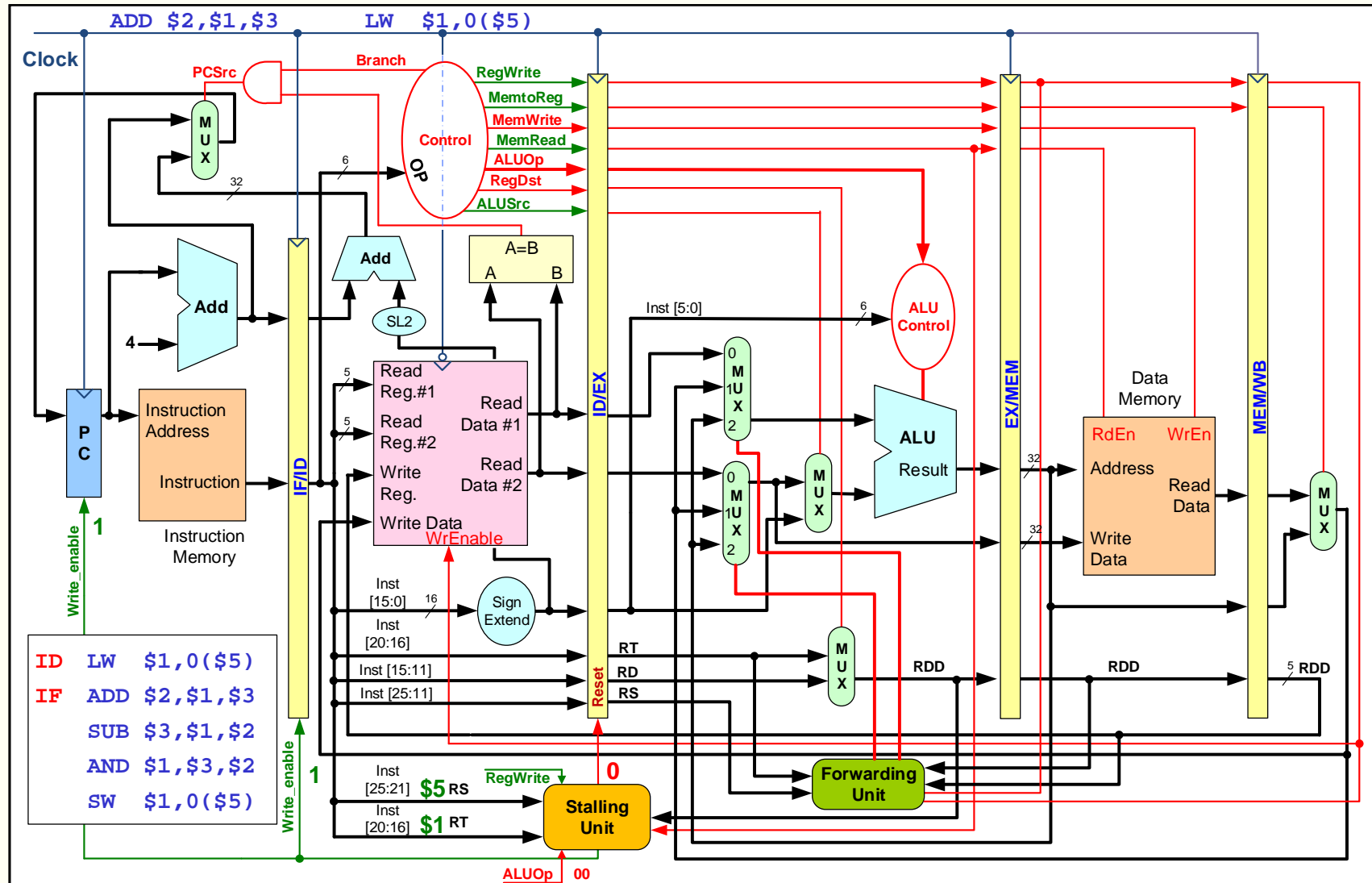
Datapath pipelining completo (com forwarding para EX, sem j)



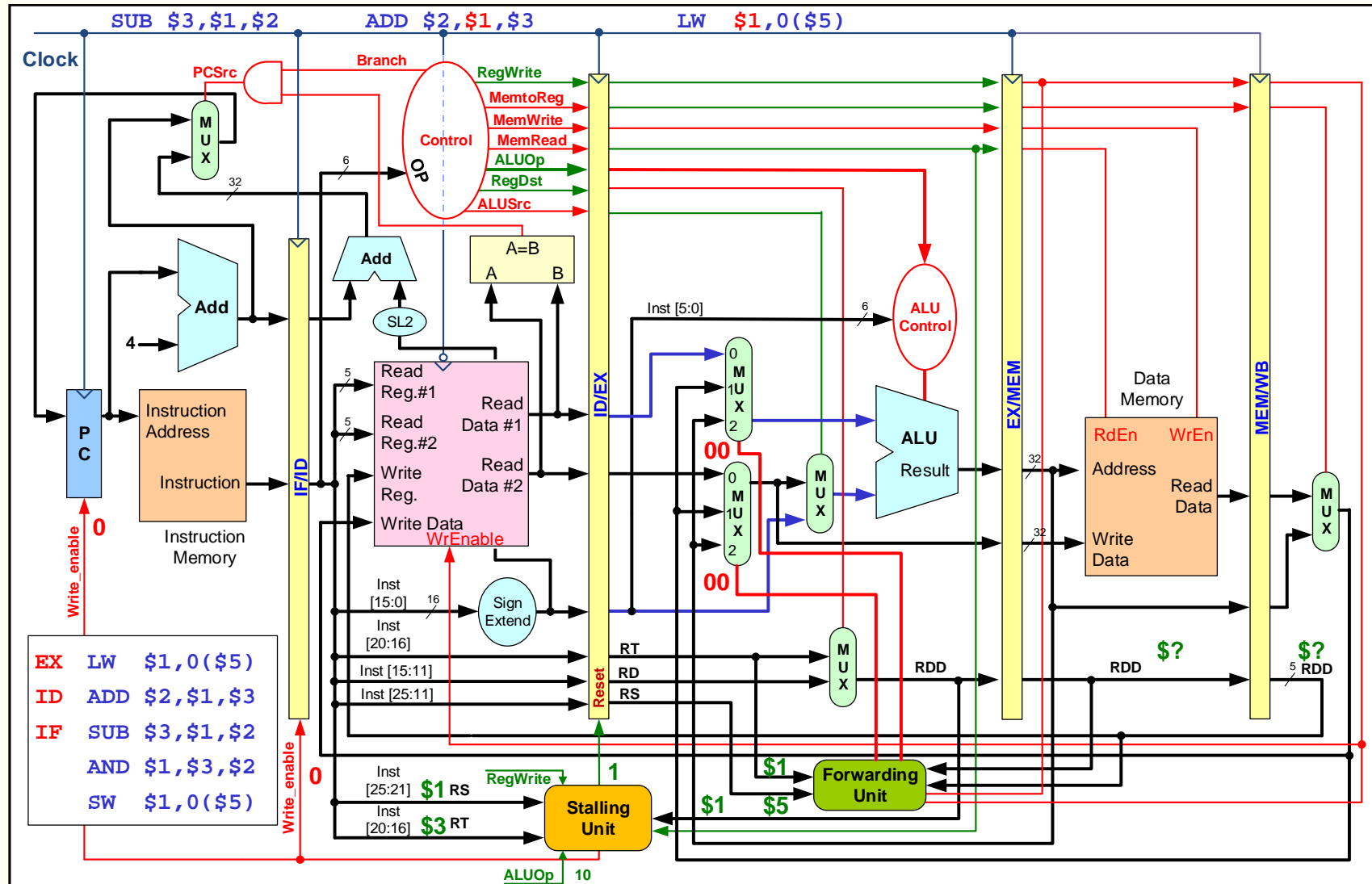
Datapath pipelining completo – exemplo de execução (1)



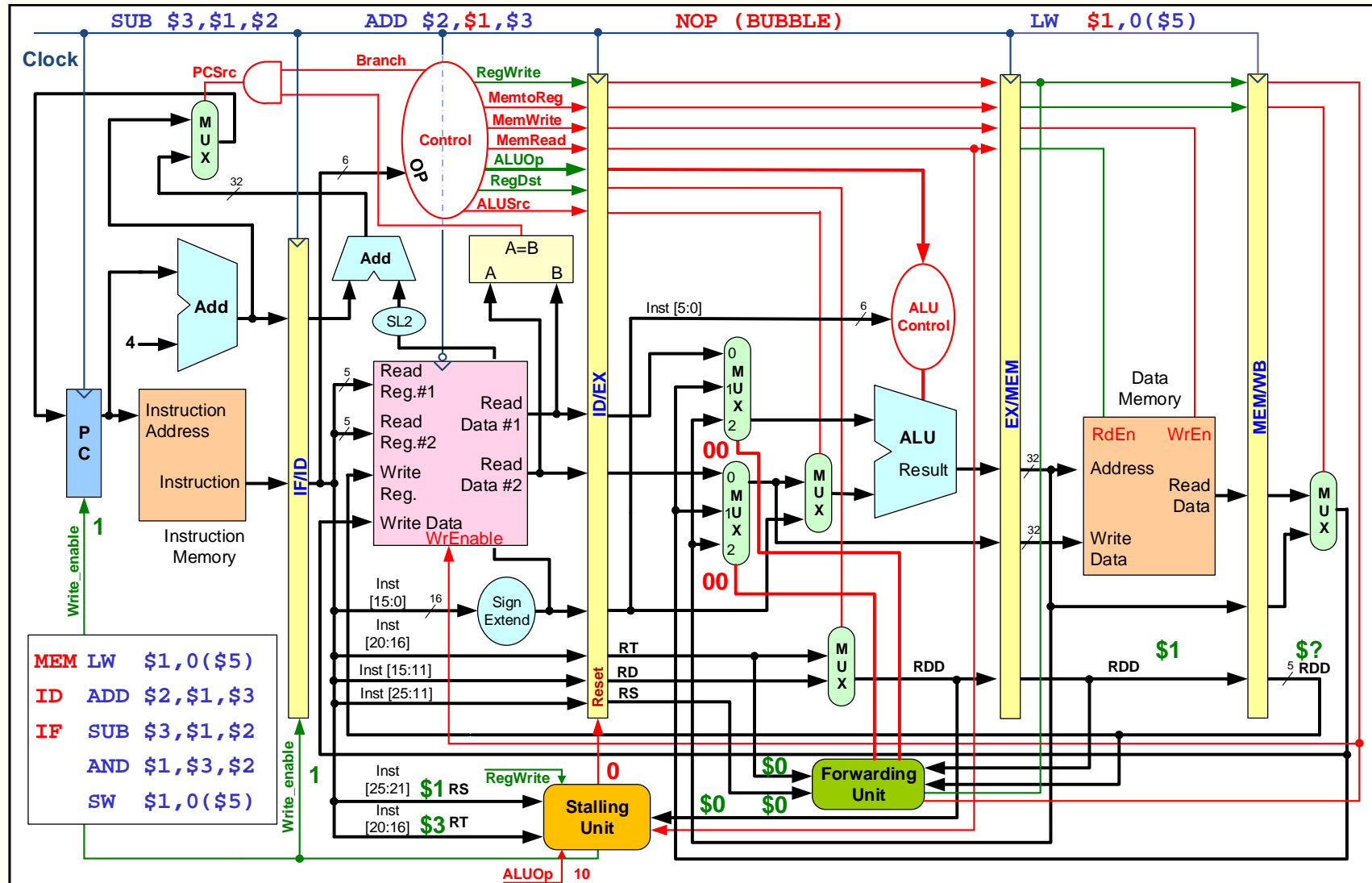
Exemplo de execução (2) (Normal Flow)



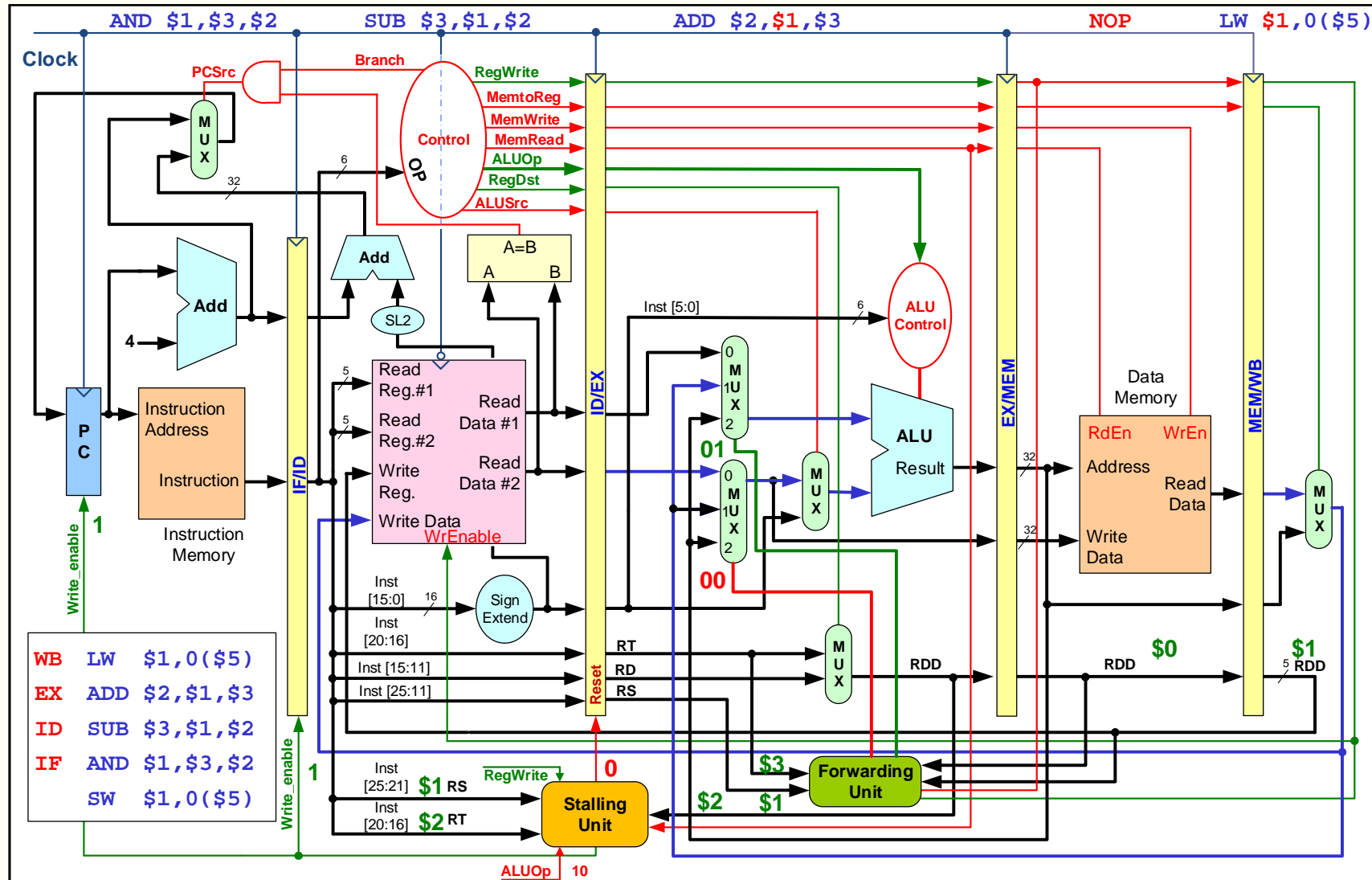
Exemplo de execução (3) (Normal Flow)



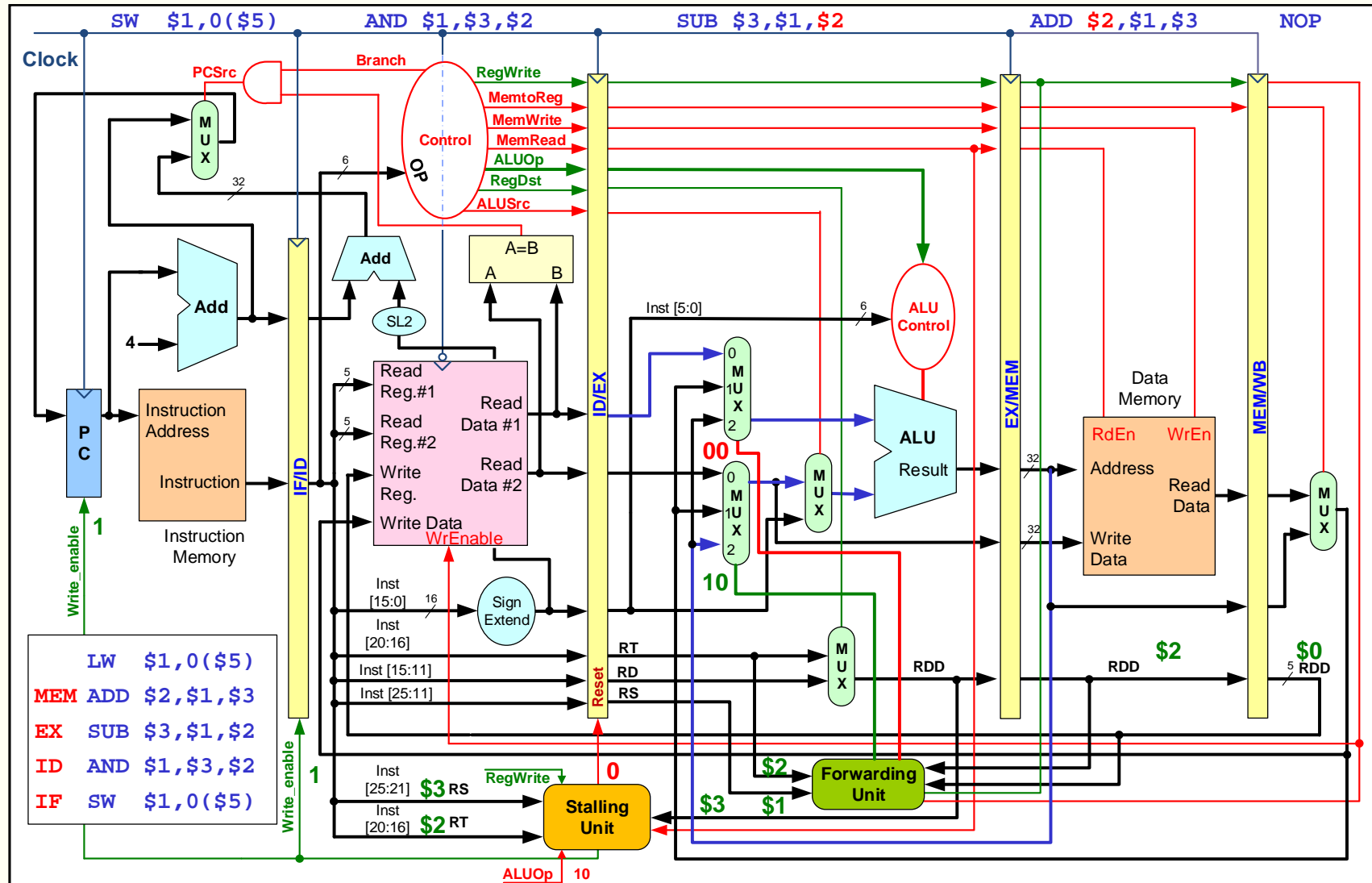
Exemplo de execução (4) (STALL)



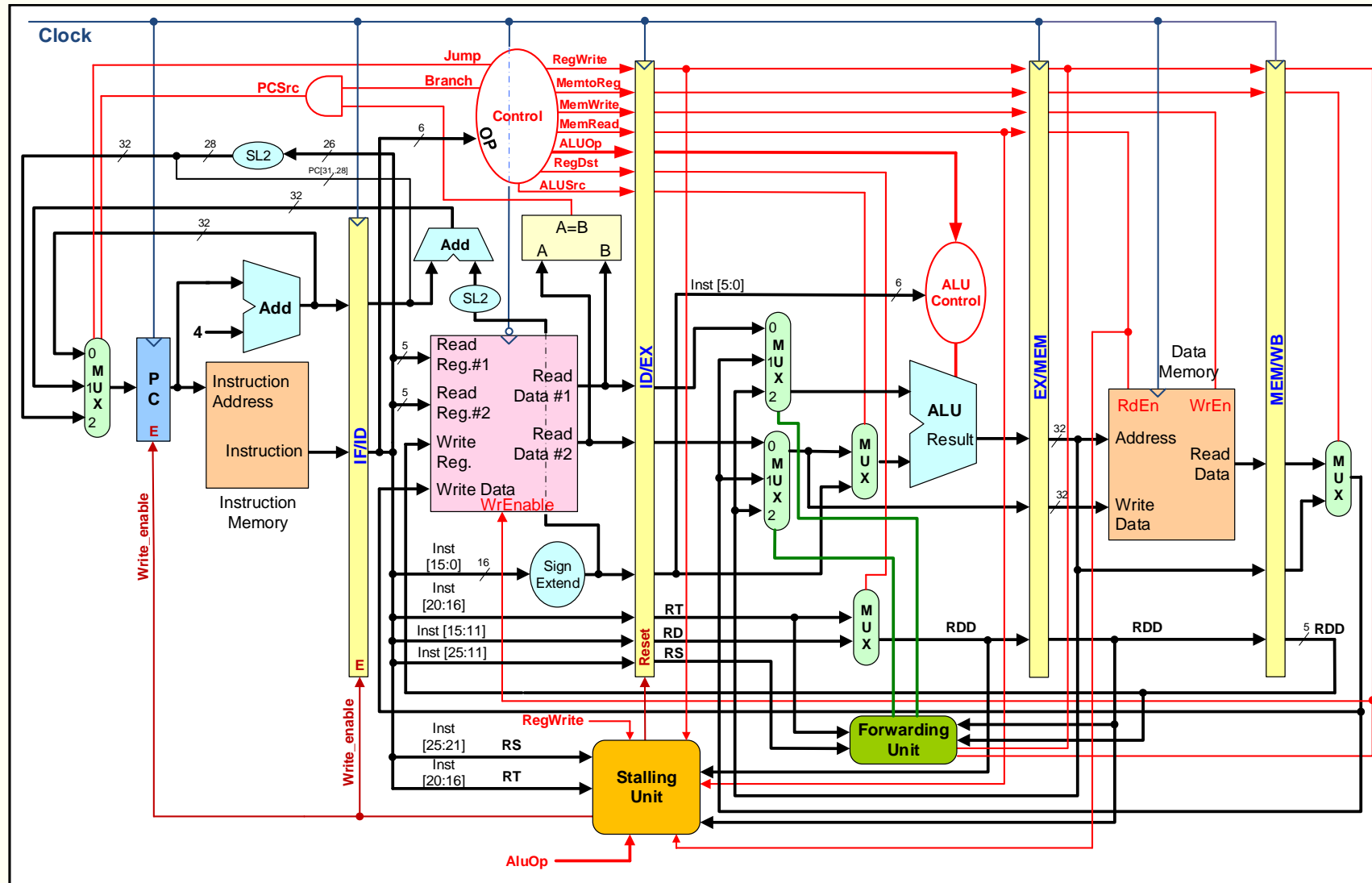
Exemplo de execução (5) (Fwd: MEM/WB > EX, rs)



Exemplo de execução (6) (Fwd: EX/MEM > EX, rt)



Datapath pipelining completo, com Jump



Forwarding de MEM/WB para MEM

- Uma dependência originada por uma sequência do tipo:

```
lw      $1, 0($5) #
```

```
sw      $1, 4($4) # Stall 1T, FW MEM/WB > EX (RT)
```

Pode ser resolvida com *stall* durante 1 ciclo de relógio seguido de *forwarding* de MEM/WB para EX

- Será mesmo necessário fazer o *stall* do pipeline?
- A instrução SW só necessita do valor de \$1 no estágio MEM (\$4 é necessário em EX), situação em que a instrução LW já se encontra em WB
- Esta situação particular pode então ser resolvida com *forwarding* de MEM/WB para MEM, evitando-se o *stall* do pipeline

Unidade de controlo de *stalling*

- Para além da sequência descrita anteriormente, há outras situações que obrigam a *stall* do *pipeline*. Exemplos (supondo que a arquitetura apenas implementa *forwarding* para **EX**):

| | | |
|----|------------|---------------------------------|
| lw | \$1,0(\$5) | # |
| sw | \$1,4(\$4) | # Stall 1T, FW MEM/WB > EX (RT) |

| | | |
|----|------------|---------------------------------|
| lw | \$1,0(\$3) | # |
| sw | \$4,8(\$1) | # Stall 1T, FW MEM/WB > EX (RS) |

| | | |
|------|--------------|---------------------------------|
| lw | \$1,0(\$6) | # |
| addi | \$4,\$1,0x12 | # Stall 1T, FW MEM/WB > EX (RS) |

| | | |
|----|------------|---------------------------------|
| lw | \$1,0(\$3) | # |
| lw | \$4,8(\$1) | # Stall 1T, FW MEM/WB > EX (RS) |

Unidade de controlo de *stalling*

- Mesmo supondo que a arquitetura implementa *forwarding* para **ID** (**EX/MEM** > **ID**) persistem situações em que há necessidade de fazer *stall* ao *pipeline*.
- Exemplos:

| | | |
|-----|-------------|---------------------------------|
| add | \$1,\$2,\$3 | # |
| beq | \$1,\$5,lab | # Stall 1T, FW EX/MEM > ID (RS) |

| | | |
|------|--------------|---------------------------------|
| addi | \$1,\$3,0x25 | # |
| beq | \$5,\$1,lab | # Stall 1T, FW EX/MEM > ID (RT) |

| | | |
|-----|-------------|------------|
| lw | \$1,0(\$5) | # |
| beq | \$1,\$2,lab | # Stall 2T |

Exercício 1

- Determine o número de ciclos de relógio que o trecho de código seguinte demora a executar num *pipeline* de 5 fases, desde o instante em que é feito o *Instruction Fetch* da 1ª instrução, até à conclusão da última:

```
add    $1,$2,$3
lw     $2,0($4)
sub     $3,$4,$3
addi    $4,$4,4
and     $5,$1,$5    #"and" em ID, "add" já terminou
sw      $2,0($1)    #"sw" em ID, "add" e "lw"
                        # já terminaram
```

$$\begin{aligned}\text{Nr_Cycles} &= F + (\text{Number_of_executed_instructions} - 1) \\ &= 5 + (6 - 1) = 10 \text{ T}\end{aligned}$$

Num *datapath single-cycle* o mesmo código demoraria 6 ciclos de relógio a executar. Porque razão é a execução no *datapath pipelined* mais rápida? Quantos ciclos de relógio demora a execução num *datapath multi-cycle*?

Exercício 2a

- Para o trecho de código seguinte identifique todas as situações de *hazard* de dados e de controlo que ocorrem na execução num *pipeline* de 5 fases, com *branches* resolvidos em ID:

```
main: lw      $1,0($0)    #
      add     $4,$0,$0    #
      lw      $2,4($0)    #
loop: lw      $3,0($1)    #
      add     $4,$4,$3    # hazard de dados ($3)
      sw      $4,36($1)   # hazard de dados ($4)
      addi    $1,$1,4     #
      slt     $5,$1,$2    # hazard de dados ($1)
      bne     $5,$0,loop  # haz. dados ($5) / haz. controlo
      sw      $4,8($0)    #
      lw      $1,12($0)   #
```

Exercício 2b

- Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o **pipeline não implementa forwarding**:

```
main: lw      $1,0($0)    #
      add     $4,$0,$0    #
      lw      $2,4($0)    #
loop: lw      $3,0($1)    #
      add     $4,$4,$3    # Stall 2T
      sw      $4,36($1)   # Stall 2T
      addi    $1,$1,4     #
      slt     $5,$1,$2    # Stall 2T
      bne     $5,$0,loop  # Stall 2T
      sw      $4,8($0)    #
      lw      $1,12($0)   #
```


Exercício 2c

- Calcule o número de ciclos de relógio que o programa anterior demora a executar num *pipeline* de 5 fases, **sem forwarding, com branches resolvidos em ID e delayed branch**, desde o IF da 1ª instrução até à conclusão da última instrução

```
main: lw      $1,0($0)    # $1=0x10
      add     $4,$0,$0    # $4=0
      lw      $2,4($0)    # $2=0x20
loop: lw      $3,0($1)    #
      add     $4,$4,$3    # stall 2T
      sw      $4,36($1)   # stall 2T
      addi    $1,$1,4     #
      slt     $5,$1,$2    # stall 2T
      bne     $5,$0,loop  # stall 2T
      sw      $4,8($0)    #
      lw      $1,12($0)   #
```

Memória de dados

| Addr | Value |
|------------|-------|
| 0x00000000 | 0x10 |
| 0x00000004 | 0x20 |

- O ciclo é executado 4 vezes: $\$1 \in [0x10, 0x20[$
- Nr de instruções executadas no ciclo: $4 * 7 = 28$
- Nr de instruções executadas fora do ciclo: $3 + 1 = 4$
- Nr de *cycle stalls* = $4 * 8 = 32$

$$\begin{aligned}\text{Nr_cycles} &= F + (\text{Nr_instructions} - 1) + \text{Nr_Cycle_Stalls} \\ &= 5 + (28 + 4 - 1) + 32 = 68 \text{ T}\end{aligned}$$

Exercício 2d

- Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o **pipeline implementa forwarding para EX e para ID**:

```
main: lw      $1,0($0)    #
      add     $4,$0,$0    #
      lw      $2,4($0)    #
loop: lw      $3,0($1)    #
      add     $4,$4,$3    # Stall 1T, FW MEM/WB > EX (RT)
      sw      $4,36($1)   # FW EX/MEM > EX (RT)
      addi    $1,$1,4     #
      slt     $5,$1,$2    # FW EX/MEM > EX (RS)
      bne     $5,$0,loop  # Stall 1T, FW EX/MEM > ID (RS)
      sw      $4,8($0)    #
      lw      $1,12($0)   #
```

Exercício 2e

- Calcule o número de ciclos de relógio que o programa anterior demora a executar num *pipeline* de 5 fases, **com forwarding para EX e para ID, com branches resolvidos em ID e delayed branch**, desde o IF da 1ª instrução até à conclusão da última instrução

```
main: lw      $1,0($0)    #
      add     $4,$0,$0    #
      lw      $2,4($0)    #
loop: lw      $3,0($1)    #
      add     $4,$4,$3    # Stall 1T, FW MEM/WB > EX (RT)
      sw      $4,36($1)   # FW EX/MEM > EX (RT)
      addi    $1,$1,4     #
      slt     $5,$1,$2    # FW EX/MEM > EX (RS)
      bne     $5,$0,loop  # Stall 1T, FW EX/MEM > ID (RS)
      sw      $4,8($0)    #
      lw      $1,12($0)   #
```

$$\begin{aligned}\text{Nr_cycles} &= F + (\text{Nr_instructions} - 1) + \text{Nr_Cycle_Stalls} \\ &= 5 + (28 + 4 - 1) + 8 = 44 \text{ T} \quad (\text{nr of cycle stalls} = 4 * 2 = 8\text{T})\end{aligned}$$

Exercício 3

Relativamente ao programa da página seguinte a executar num *pipeline* de 5 fases, com *branches* resolvidos em ID e *delayed branch slot*:

1. Identifique todas as situações de *hazard* de dados e de controlo
2. Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o *pipeline* não implementa *forwarding*
3. Calcule o número de ciclos de relógio que o programa demora a executar (na situação da questão anterior), desde o IF da 1ª instrução até à conclusão da última instrução
4. Apresente o modo de resolução das situações de *hazard* de dados, admitindo que o *pipeline* implementa *forwarding* para ID, EX e MEM
5. Calcule o número de ciclos de relógio que o programa demora a executar (na situação da questão anterior), desde o IF da 1ª instrução até à conclusão da última instrução

Para o mesmo programa, apresente os valores presentes nos registos \$1, \$2, \$3, \$4 e \$5 no final da execução do mesmo.

Exercício 3 (programa)

```
        .data                                # Segmento de dados: 0x00000000
A1:      .word  0x41,0x43,0x31,0x2D,0x32,0x30,0x31,0x38
A2:      .space  48                          #
        .text                                #
        .globl  main                        #
main:    addi    $5,$0,0                     #
        addi    $4,$0,0x20                  #
        addi    $2,$0,7                     #
        add     $2,$2,$2                     #
        add     $2,$2,$2                     #
        add     $2,$2,$4                     #
C1:      lw      $3,0($5)                    #
        sw      $3,0($2)                    #
        addi    $2,$2,-4                     #
        slt     $1,$2,$4                     #
        beq     $1,$0,C1                     #
        addi    $5,$5,4                     #
C2:      addi    $0,$0,0                     #
```