Aulas 22 a 26

- *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

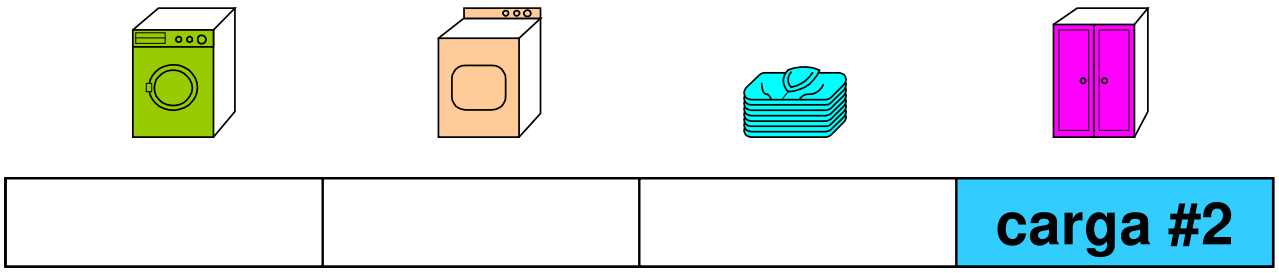
- Numa versão não *pipelined*, o processamento de N cargas de roupa seria, para cada carga:(Lavar > Secar > Engomar > Arrrumar)

carga #1

carga #2

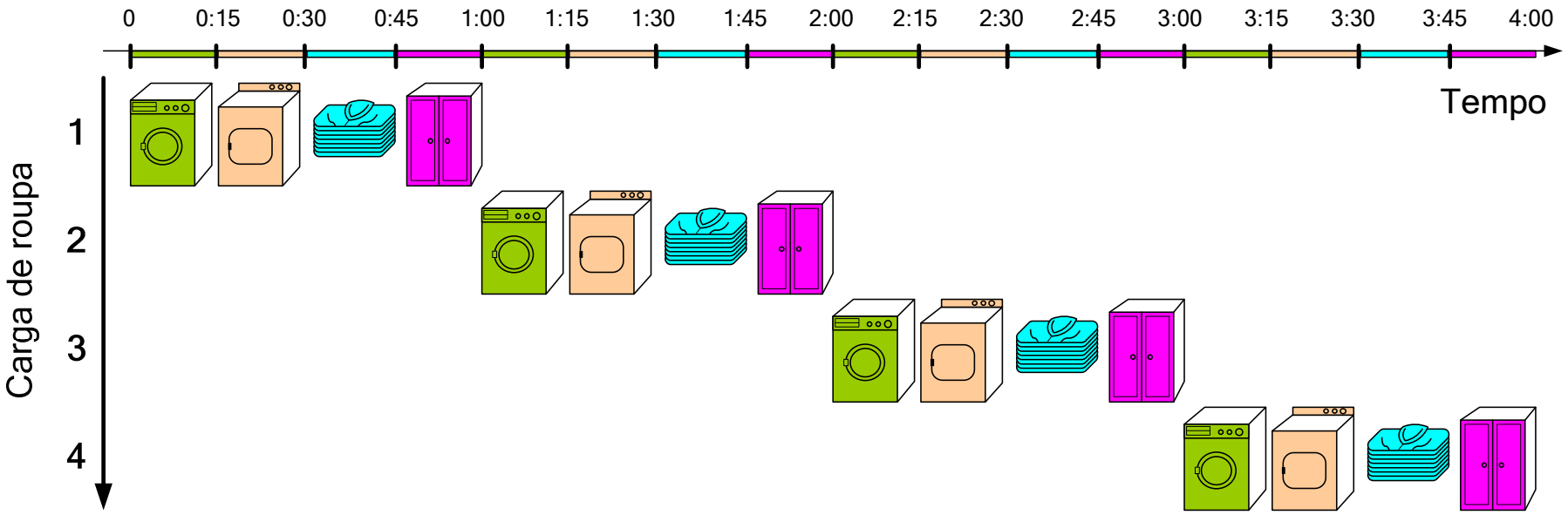
(…)

carga #n



Pipelining - exemplo por analogia

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



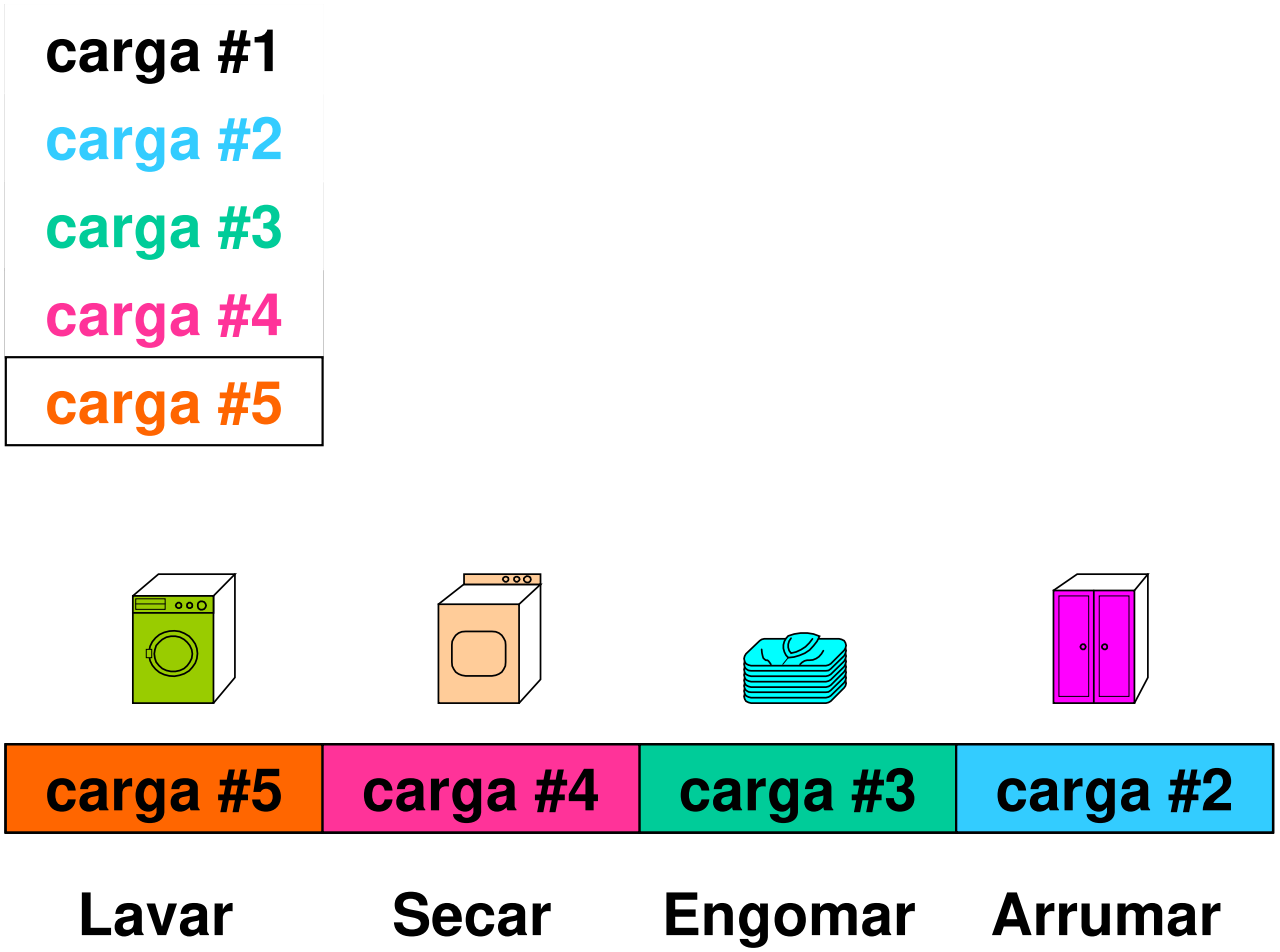
Se o tempo para tratar uma carga de roupa for uma hora, tratar quatro cargas demorará **quatro horas.**

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

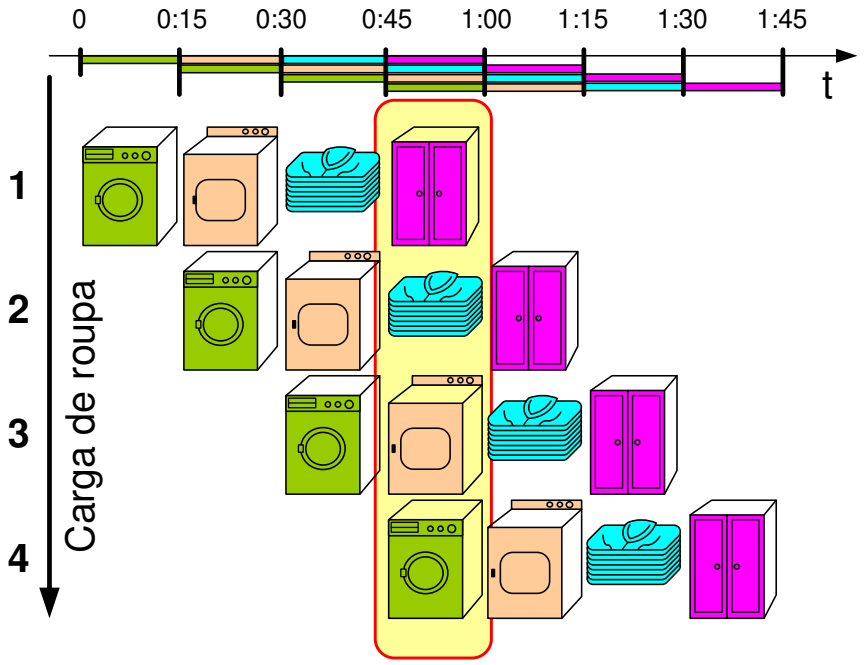
- Na versão *pipelined*, o processamento das cargas de roupa seria (admitindo tempo nulo entre a comutação de tarefas):



Pipelining - exemplo por analogia

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

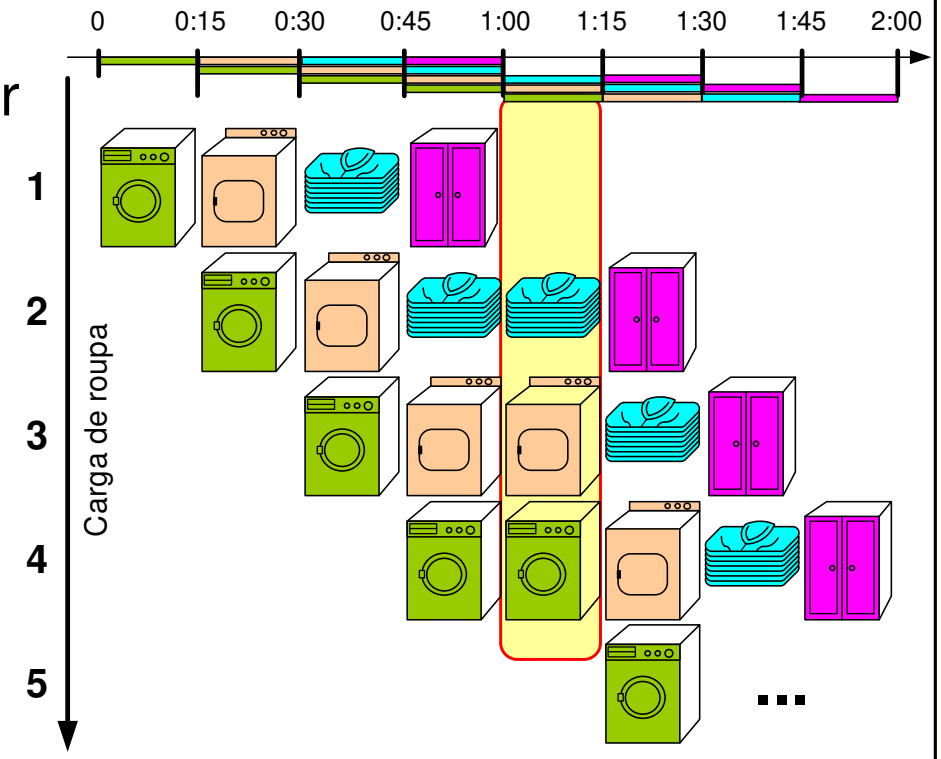
- Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45 (ou seja 135 minutos menos (240 – 105).



- O que acontece se, por exemplo, a carga 2 não precisar de ser engomada?

Pipelining - exemplo por analogia

- O que acontece se a carga 2 tiver roupa que, por alguma razão, demora mais tempo a engomar?
  - É necessária uma segunda "slot" de 15 min para completar a engomagem da carga 2
  - A carga 3 não pode avançar para a engomagem e permanece na máquina de secar
  - A carga 4 não pode avançar para a máquina de secar e permanece na máquina de lavar



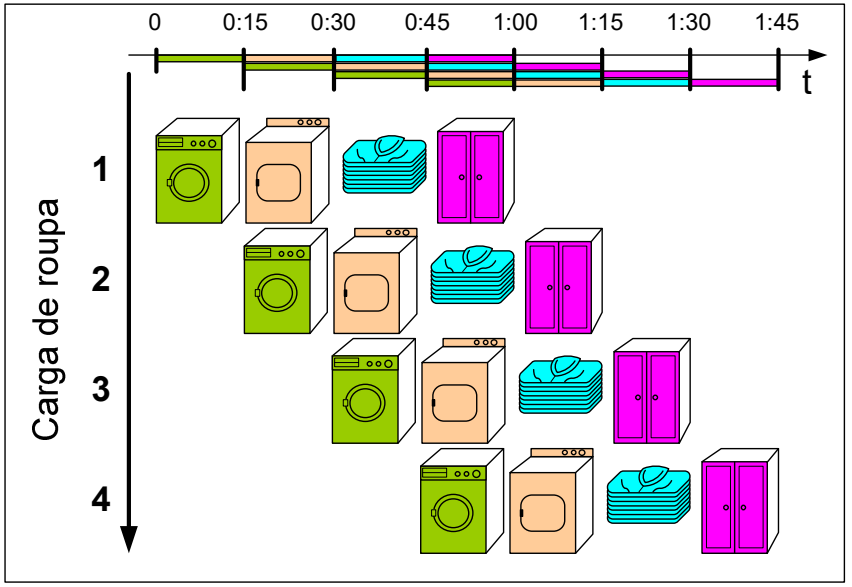
  - A carga 5 só é colocada na máquina de lavar no minuto 75

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

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

Ganho de desempenho obtido com a solução *pipelined*:

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

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

Pipelining – ganho de desempenho

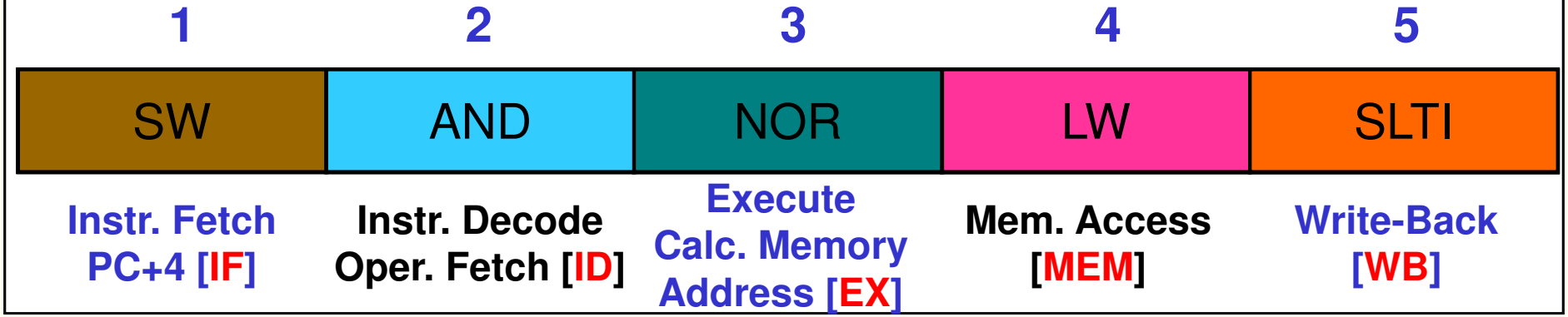
- No limite, para um número de cargas de roupa muito elevado, o ganho de desempenho (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e noutro modelo) é da ordem do **número de tarefas realizadas em paralelo** (isto é, igual ao número de fases do processo)
- Genericamente, poderíamos afirmar que o ganho em velocidade de execução é igual ao número de estágios do *pipeline* (F)
- No exemplo observado, o ganho 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 da instrução mais longa (LW) pode ser dividida genericamente em **cinco fases**
- 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:
  1. **Instruction fetch [IF]** - ler a instrução da memória, incremento do PC
  2. **Operand fetch [ID]** - ler os registos e descodificar a instrução (os formatos das instrução do MIPS permitem que estas duas tarefas possam ser executadas em paralelo)
  3. **Execute [EX]** - executar a operação ou calcular um endereço
  4. **Memory access [MEM]** - aceder à memória de dados para leitura ou escrita
  5. **Write-Back [WB]** - escrever o resultado no registo destino

Pipelining no MIPS

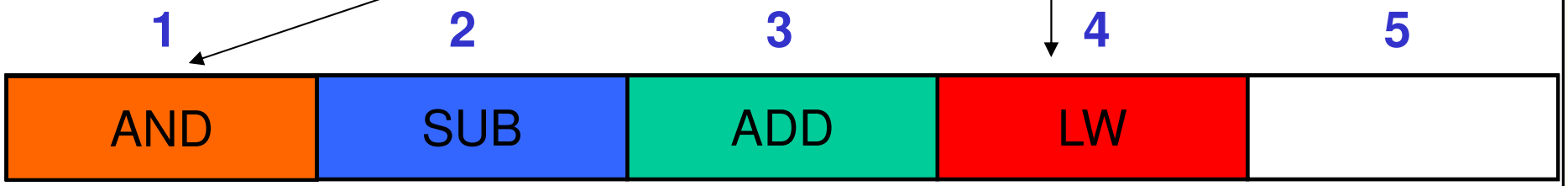
```
                or    $6,$3,$9
                sub   $2,$9,$3
                add   $4,$4,$3
5 (WB)          slti  $5,$9,0x01
4 (MEM)         lw    $3,0($3)
3 (EX)          nor   $7,$2,$6
2 (ID)          and   $8,$6,$9
1 (IF)          sw    $4,4($6)
                (…)
```



Pipelining – Problemas (exemplo 1)

```
lw    $1,0($9)
add   $2,$3,$4
sub   $3,$4,$5
and   $4,$5,$6
lw    $5,16($9)
```

- 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 ao mesmo tempo

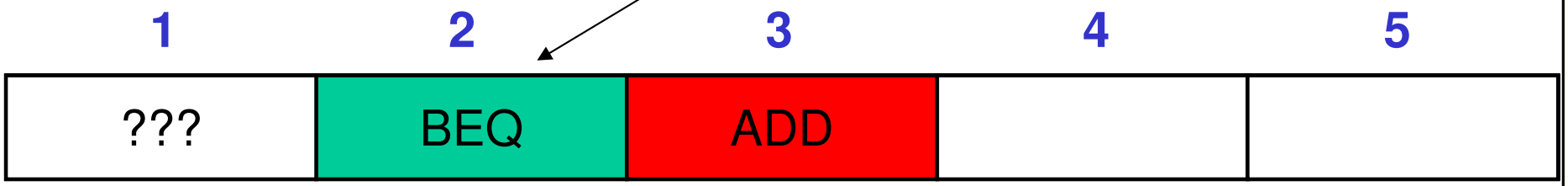


Instr. Fetch PC+4 [IF]	Instr. Decode Oper. Fetch [ID]	Execute Calc. Memory Address [EX]	Mem. Access [MEM]	Write-Back [WB]

Pipelining – Problemas (exemplo 2)

```
      add   $4,$5,$6
      beq   $2,$3,Z1
      lw    $3,300($6)
      (…)
Z1:   or    $7,$8,$9
```

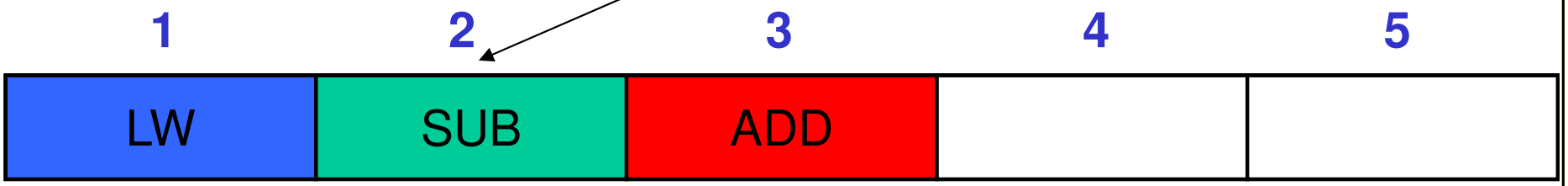
- Qual a próxima instrução a ler da memória (**LW** / **OR**) assumindo que a decisão do *branch* só é tomada no 3º estágio do *pipeline*?
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio, a unidade de controlo terá sempre que esperar pela execução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



Pipelining – Problemas (exemplo 3)

```
add   $3,$4,$5
sub   $2,$3,$6
lw    $4,0($5)
```

A instrução de subtração (SUB) não pode avançar para o estágio seguinte (EX) uma vez que o valor de um dos seus operandos ainda não foi calculado e armazenado no registo destino, o **$3**, pela instrução anterior.



Instr. Fetch PC+4 [IF]	Instr. Decode Oper. Fetch [ID]	Execute Calc. Memory Address [EX]	Mem. Access [MEM]	Write-Back [WB]

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:

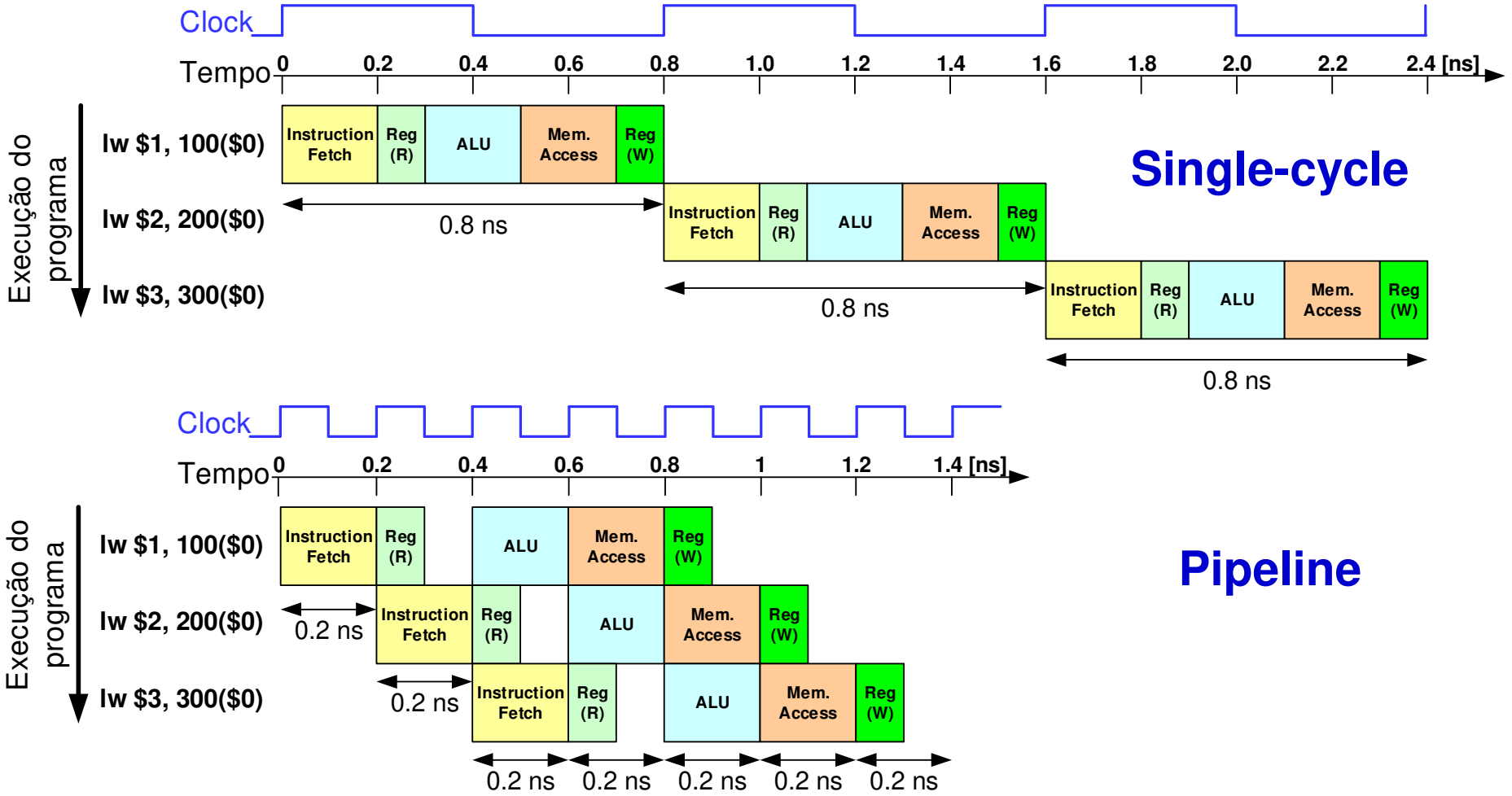
Instruction	Instruction Fetch	Register Read	ALU Operation	Memory Access	Register Write	Tempo total
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-Type (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps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

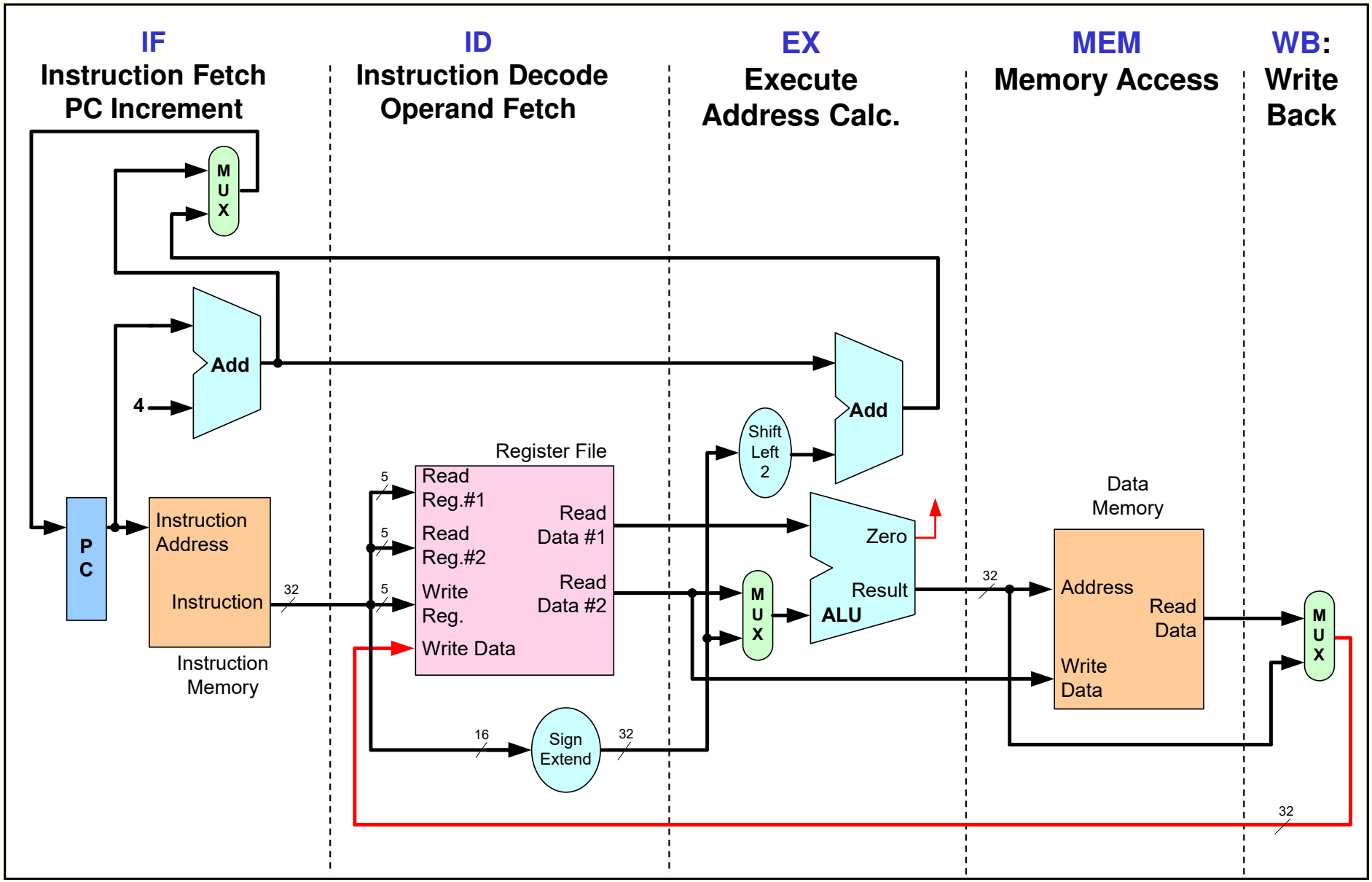
- O *instruction set* do **MIPS** (***M****icroprocessor without* ***I****nterlocked* ***P****ipeline* ***S****tages*) foi concebido para uma implementação em *pipeline*. Os aspetos fundamentais a considerar são:
  - **Instruções de comprimento fixo**: *Instruction Fetch* e *Instruction Decode* podem 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 nos mesmos campos (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 calcular o resultado da operação na ALU 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

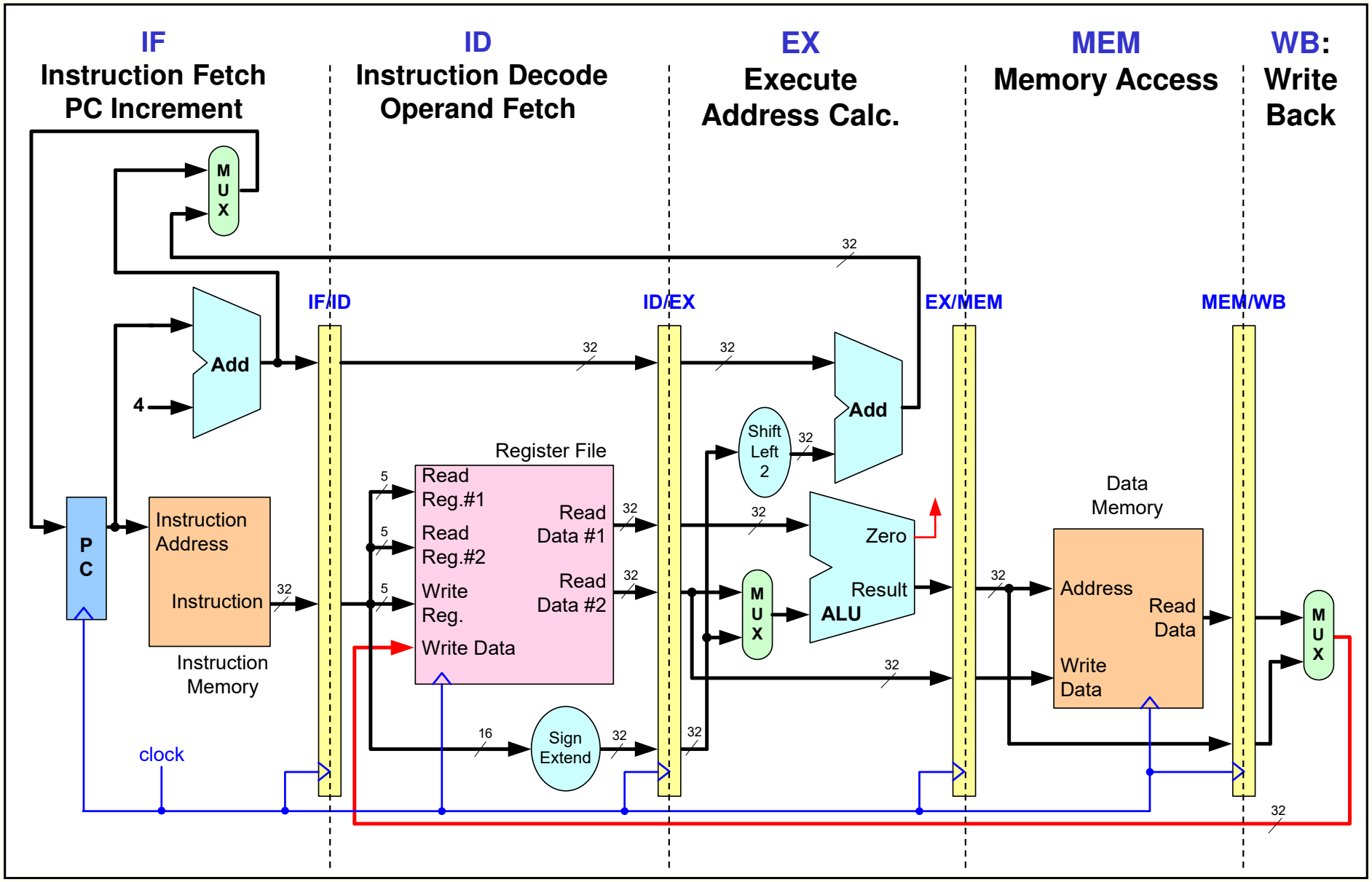
- O *pipeline* implementa as cinco fases sequenciais em que são decomponíveis as instruções:
  1. (**IF**) - *Instruction fetch* (ler a instrução da memória), incremento do PC
  2. (**ID**) - *Operand fetch* (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas 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)

- A solução *pipelined* para o MIPS parte do modelo do *datapath single-cycle*
- Na solução apresentada no slide seguinte não são identificados os sinais de controlo nem a respetiva unidade de controlo

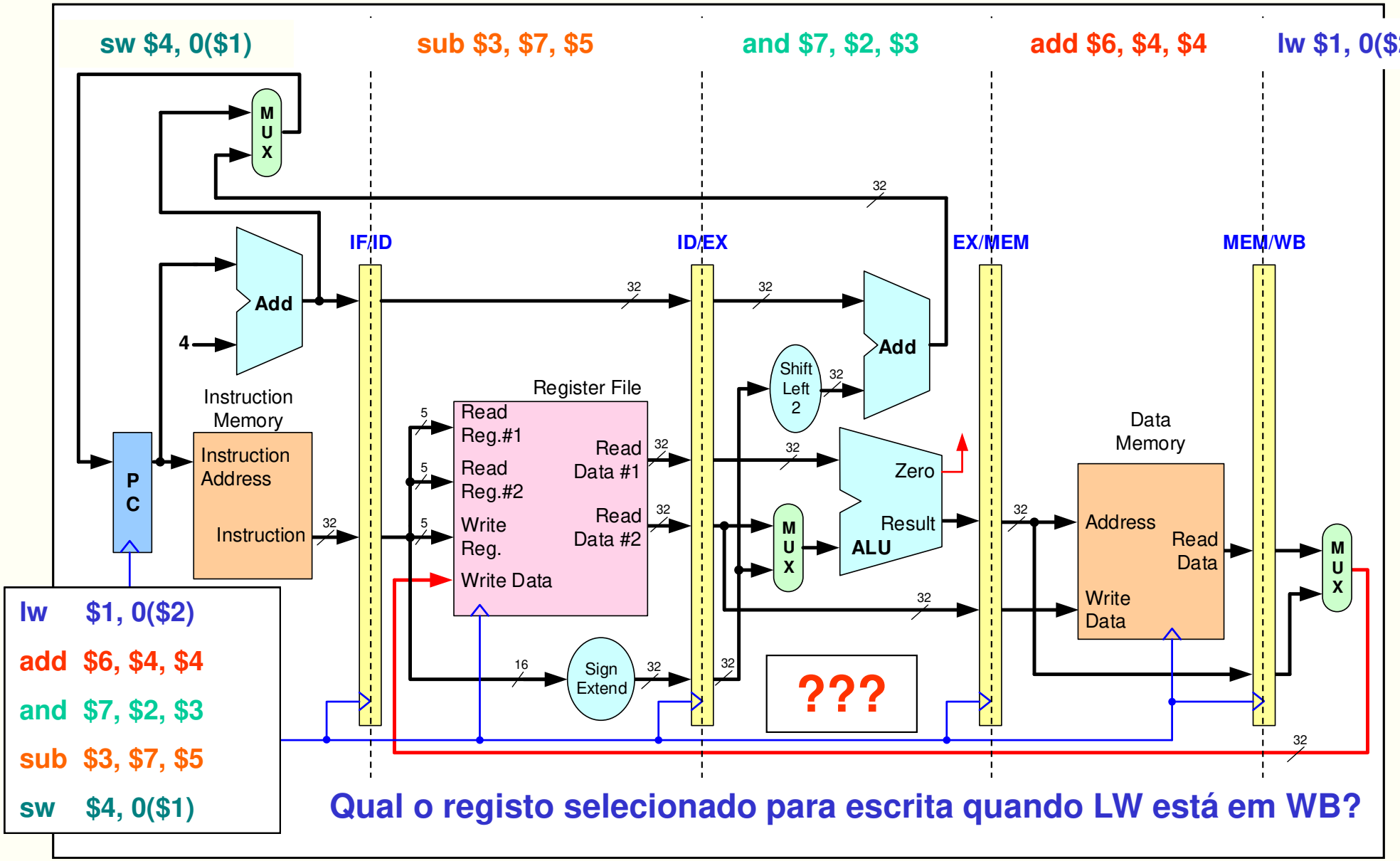
Divisão em fases de execução



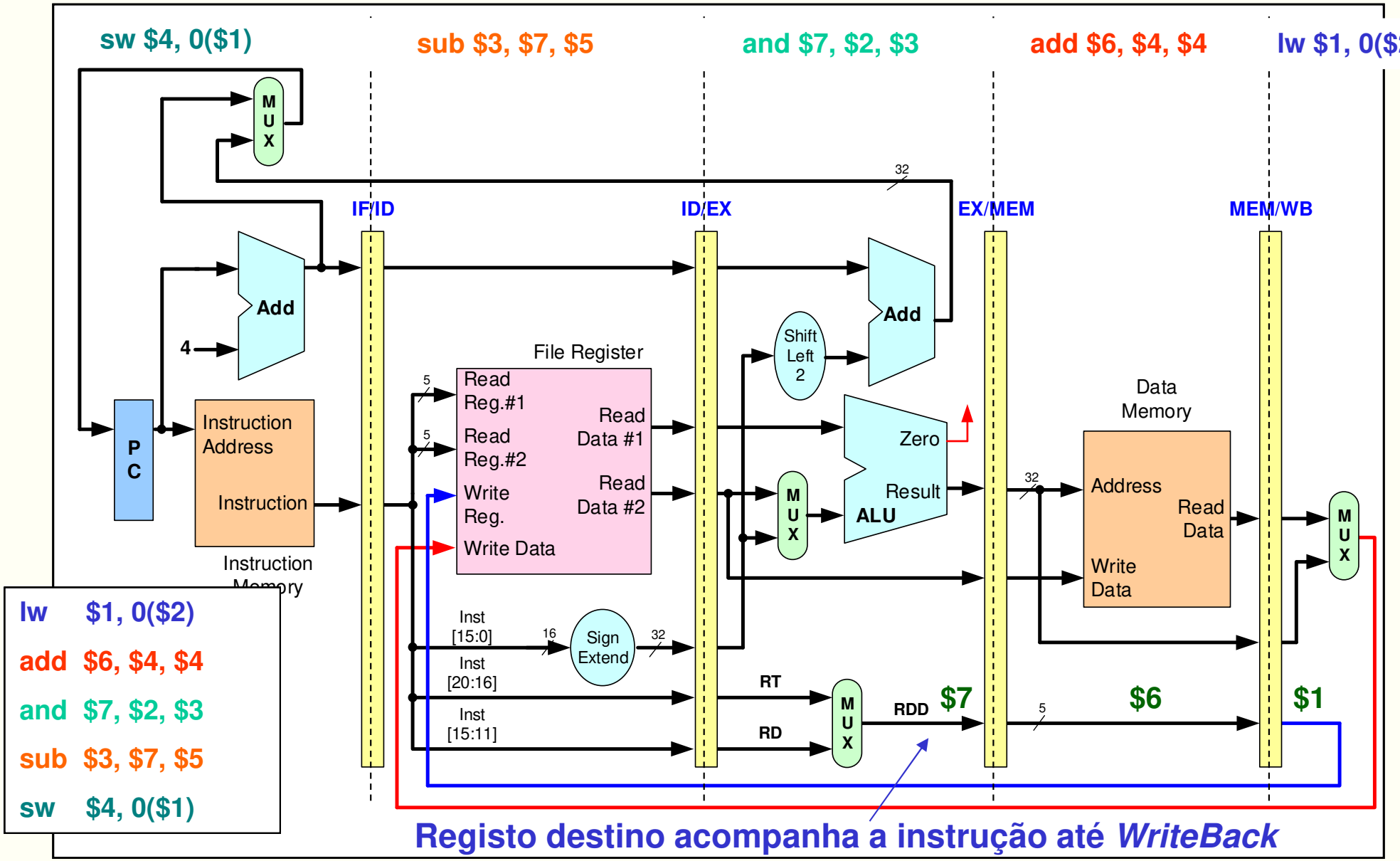
Divisão em fases de execução – registos de *pipeline*



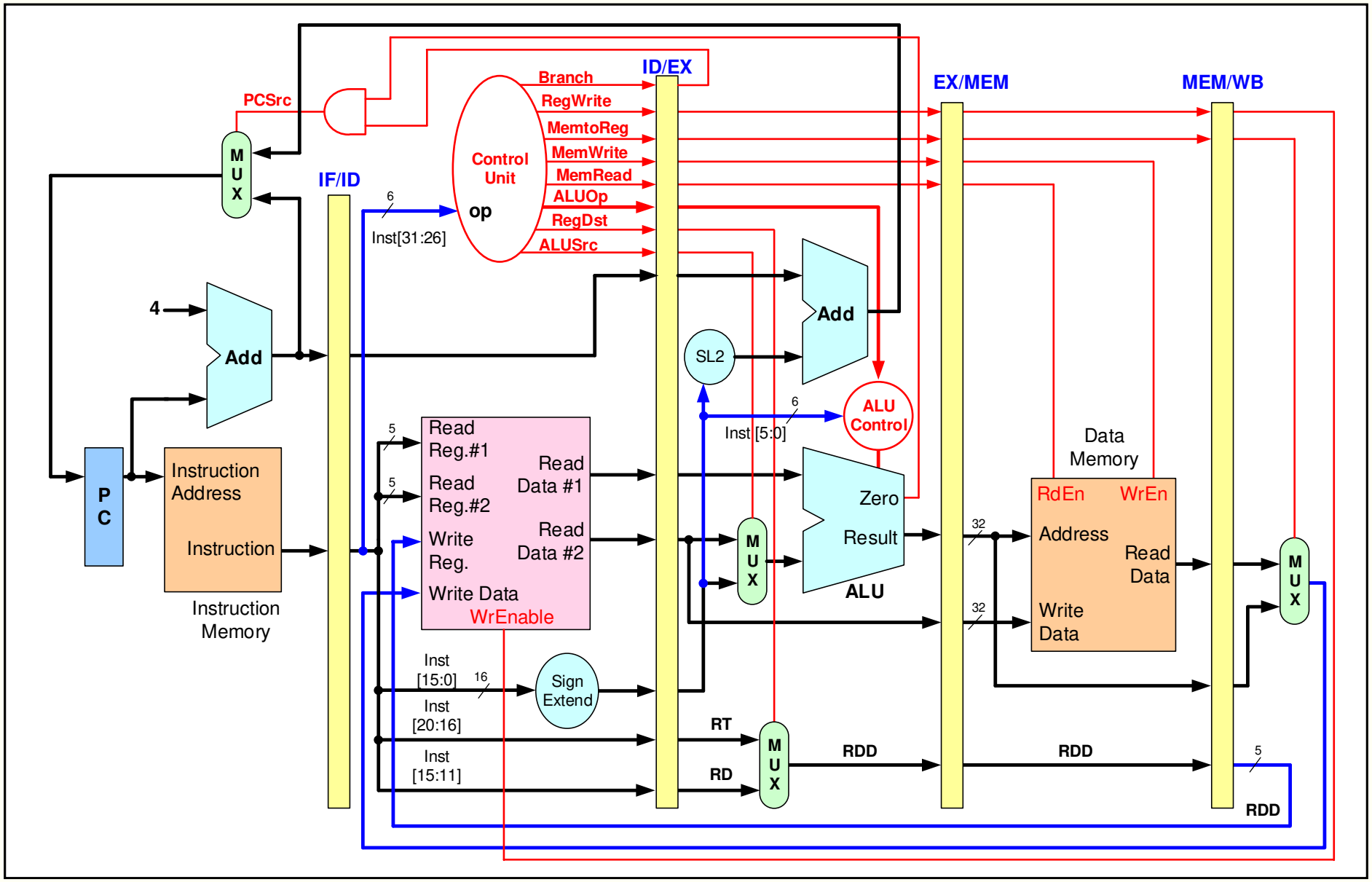
Execução de instruções



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
  - Os sinais **MemRead** e **MemWrite** são propagados até à fase MEM, onde controlam o acesso à memória
  - 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)

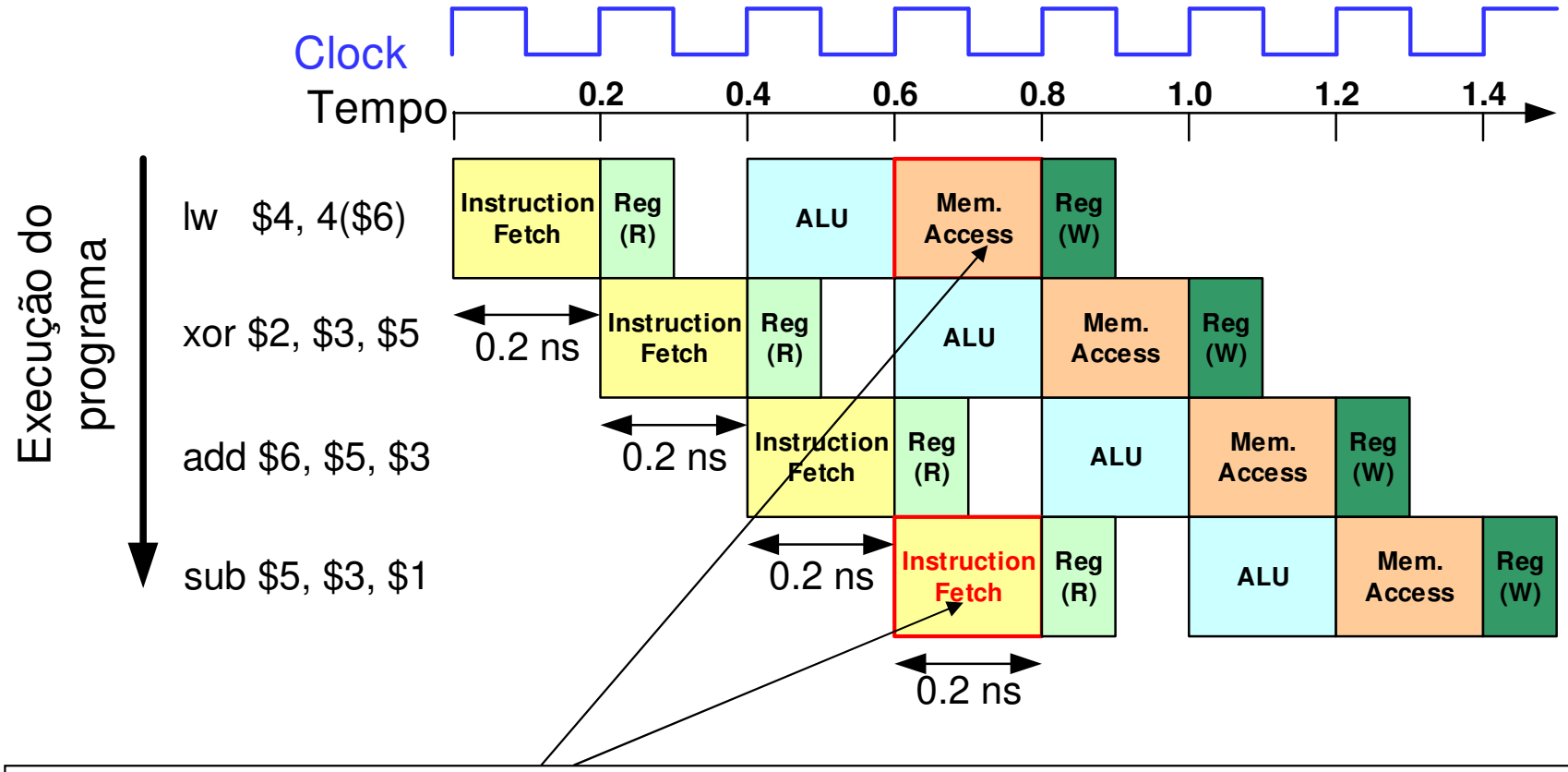
Pipeline Hazards

- 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 implementação da arquitetura *pipelined* do MIPS

Hazards Estruturais

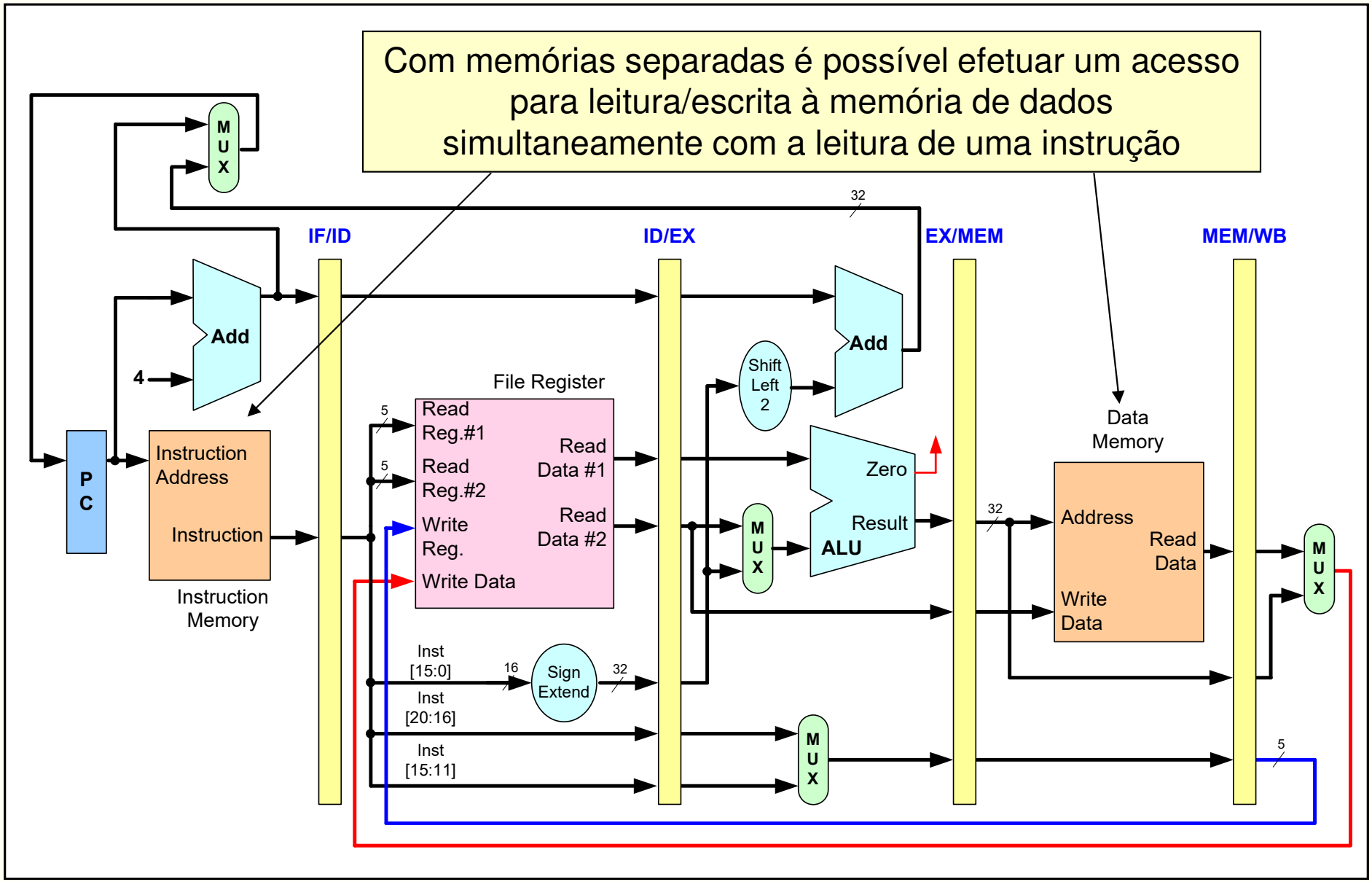
- Um ***hazard* estrutural** ocorre quando mais do que uma instrução necessita de aceder ao mesmo hardware
- Ocorre quando: 1) apenas existe uma memória ou 2) há instruções no *pipeline* com diferentes tempos de execução
- No primeiro caso o *hazard* estrutural é evitado duplicando a memória, i.e., uma memória de instruções 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, usando outro elemento operativo diferente da ALU

Hazards Estruturais



- 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*
- **A não existência de memórias separadas determinaria, neste caso, a ocorrência de um *hazard* estrutural**

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
- Exemplo:

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

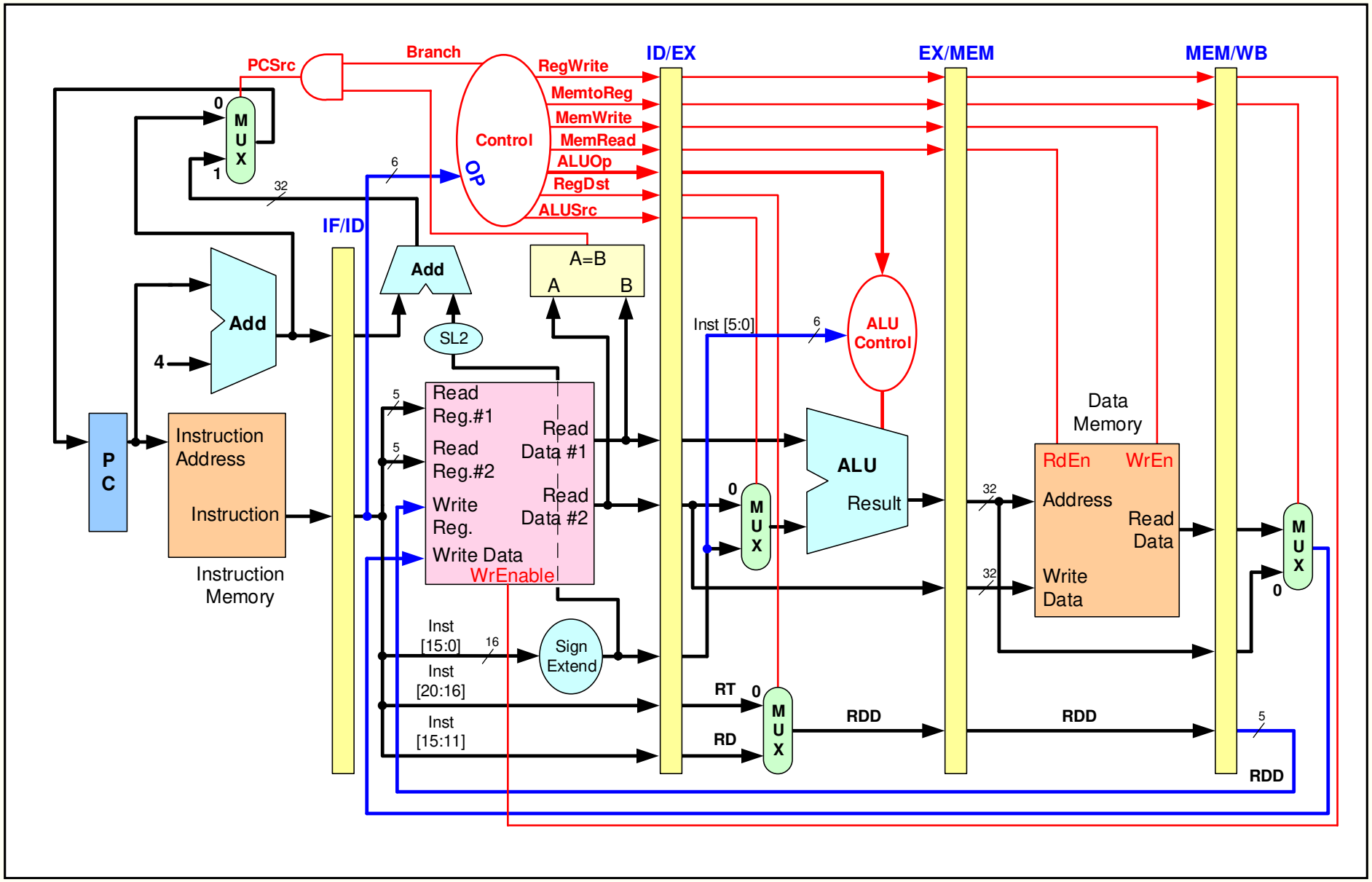
Qual a instrução que deve entrar no *pipeline* a seguir à instrução "beq"?

- No caso do MIPS, as situações de *hazard* de controlo surgem com as instruções de salto, (*jumps* e *branches*: j, jal, jalr, jr, beq, …)

Hazards de Controlo

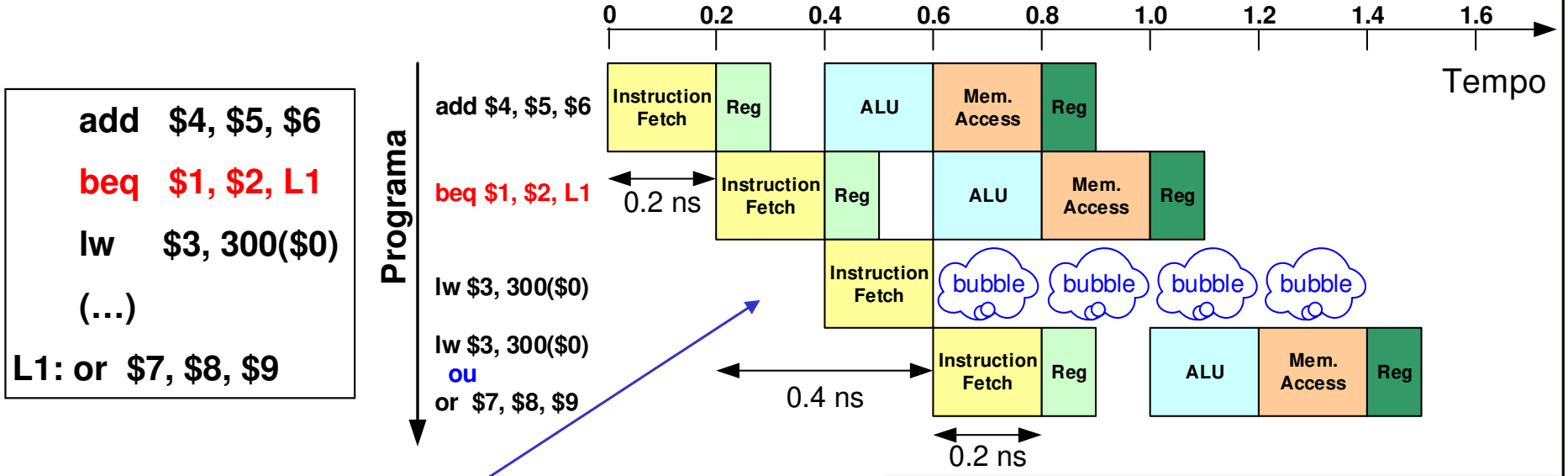
- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de instruções
- A resolução dos *branches* em ID minimiza o problema, e por isso a **comparação dos operandos passa a ser 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



Pipeline ***stall***.
Também conhecido por *bubble*

Bubble: A instrução é convertida em **NOP** (*no operation*. Exemplo: **sll $0,$0,0**), Código **0x00000000**

- É uma solução conservativa que tem um preço em termos de tempo de execução

Exercício

- Se 15% das instruções de um dado programa forem *branches* e *jumps*, qual o efeito da solução de *stalling* no desempenho da arquitetura, admitindo que essas instruções são resolvidas em ID?

  Sem *stalls*: CPI = 1

  Com *stalls*: CPI = 0,85 * 1 + 0,15 * 2 = 1,15

  Relação de desempenho = 1 / 1,15 = 0,87

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

  Relação de desempenho = 1 / (0,85 * 1 + 0,15 * 3) = 0,77

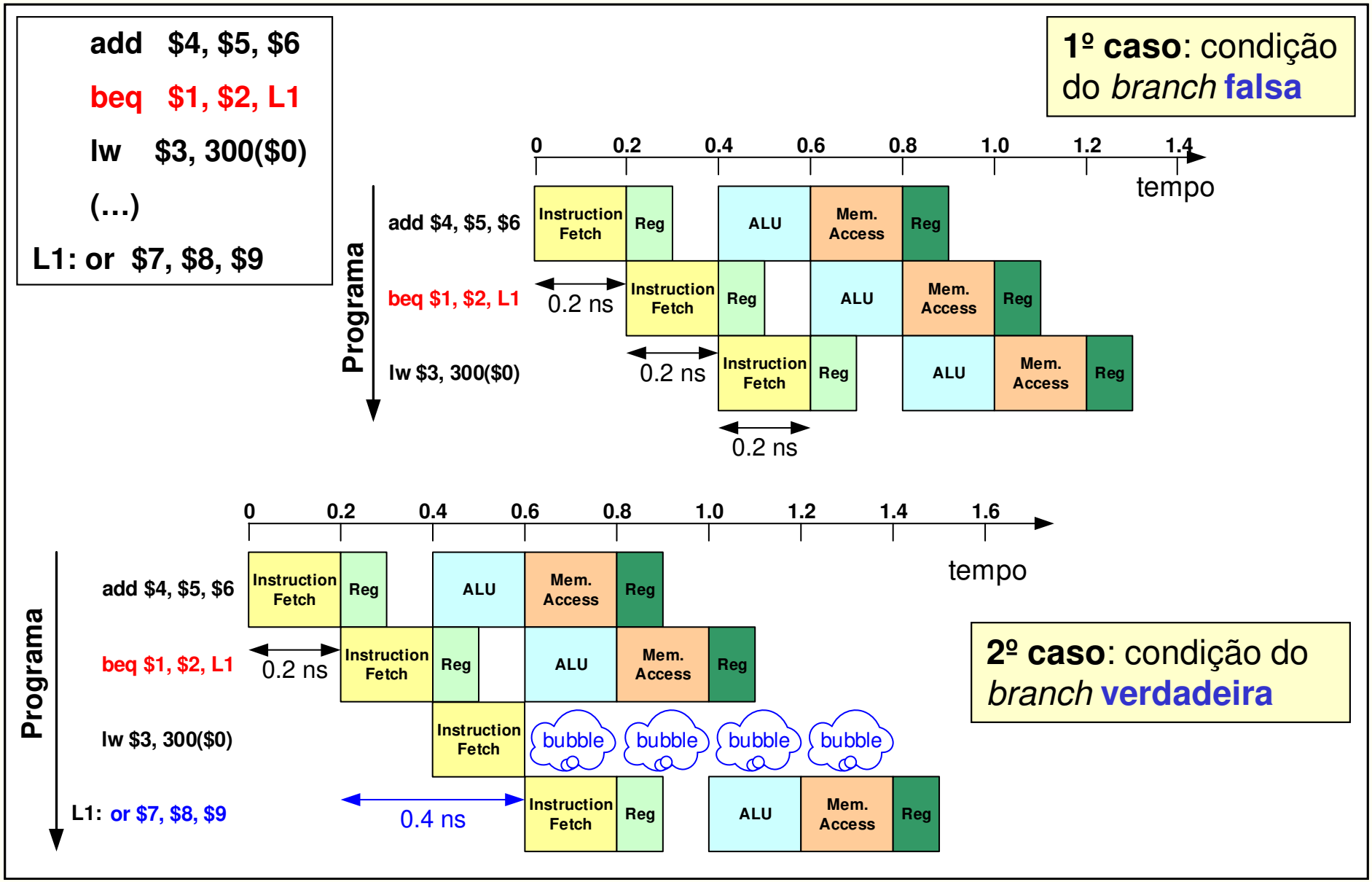
Hazards de controlo

- Uma solução alternativa ao *pipeline stalling* é designada por **previsão** (*prediction*):
  - Prevê-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 anulada (convertida em **nop**), continuando o *instruction fetch* na instrução correta
- Se a previsão estiver certa, esta estratégia permite poupar tempo
- Para o exemplo do slide anterior, se a previsão for correta 50% das vezes, a relação de desempenho passa a ser:

  CPI = 1 + 1 * 0,15 / 2 = 1,075

  Relação de desempenho = 1 / 1,075 = 0,93

Hazards de controlo – previsão estática *not taken*



Hazards de controlo – previsão

- Os previsores usados nas arquiteturas atuais são mais elaborados
- **Previsores estáticos**: o resultado da previsão não depende do histórico da execução das instruções de *branch* / *jump*:
  - **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 solução, o processador **executa sempre a instrução que se segue ao *branch*** (ou *jump*), independentemente de a condição ser verdadeira ou falsa
- Esta técnica é implementada com a ajuda do **compilador/assembler** que:
  - reorganiza 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 1ªs instruções

Código reordenado

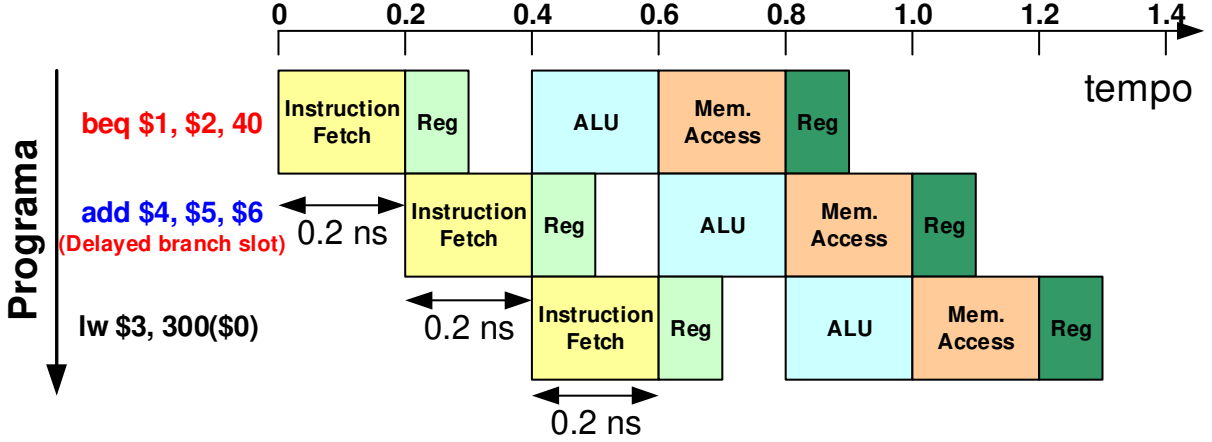
```
      beq   $1, $2, L1
      add   $4, $5, $6
      lw    $3, 300($0)
      (…)
L1: or    $7, $8, $9
```

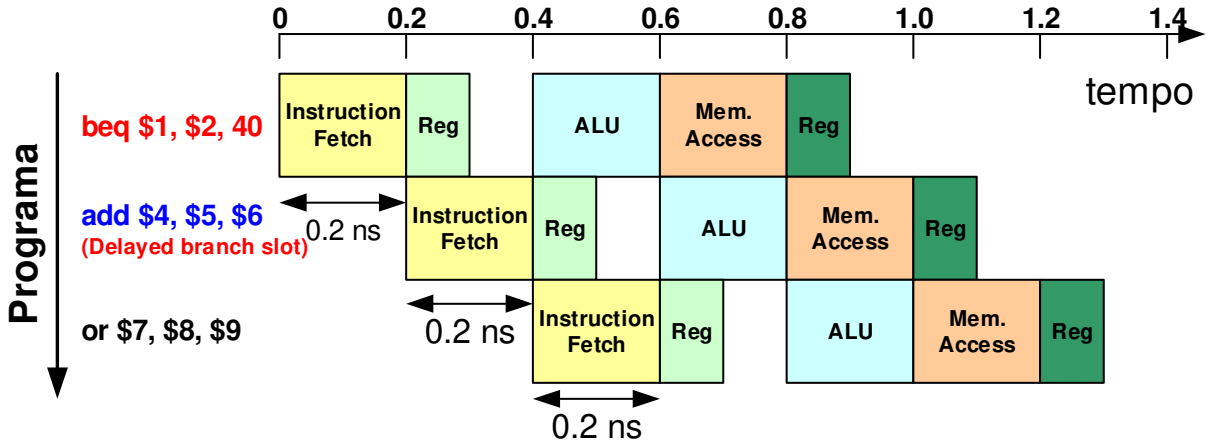
- Neste exemplo a instrução "**beq**" não depende do resultado produzido pela instrução "**add**", logo a troca das duas não altera o resultado final do programa
- No código reordenado (que é o executado no processador) a instrução "**add**" é sempre executada, independentemente do resultado *taken/not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

Hazards de controlo – *delayed branch*

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

1º caso: condição do *branch* **falsa**



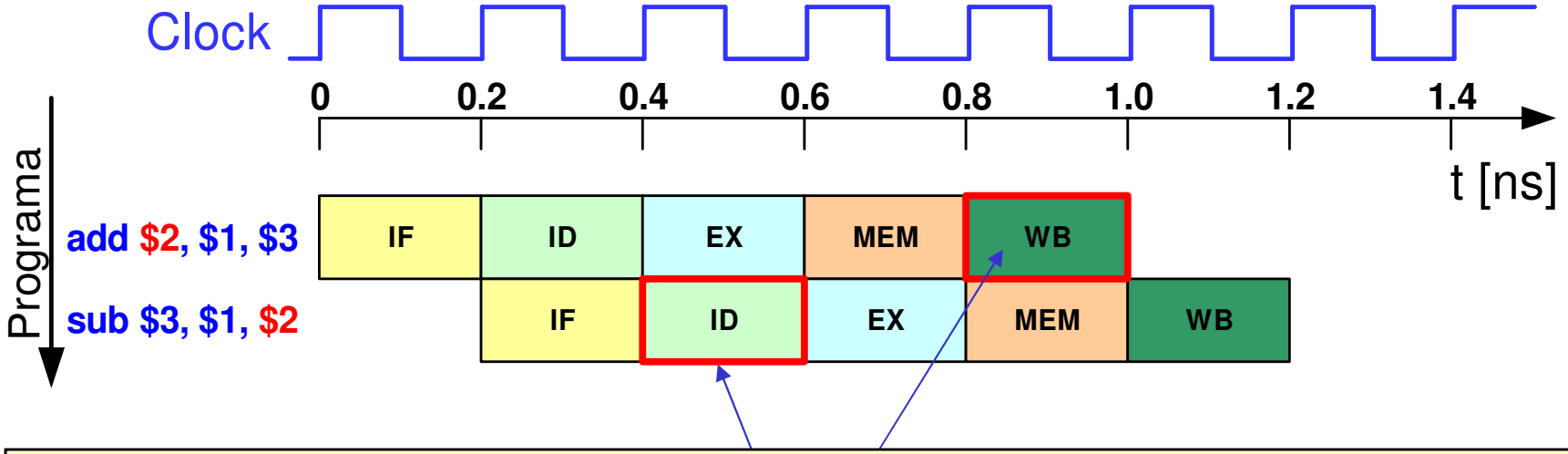


2º caso: condição do *branch* **verdadeira**

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 avançar no *pipeline* antes de o valor de **$2** ser calculado e armazenado pela instrução anterior (o valor é necessário em **t = 0.4**, mas só vai ser escrito no registo destino em **t = 1**)

Hazards de dados

- Se o resultado 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**)
- No exemplo anterior, a instrução SUB só poderia prosseguir para a fase EX em t=1.2
- O problema pode ser minorado, se a **escrita no banco de registos for feita a meio do ciclo de relógio** (i.e., na transição descendente)
  - a instrução que está na fase ID e que necessita do valor, poderá prosseguir na transição de relógio seguinte, já com o valor do registo atualizado
  - poupa-se 1 ciclo de relógio (no exemplo anterior, a instrução "**sub $3,$1,$2**" poderá prosseguir para a fase EX em t=1.0)

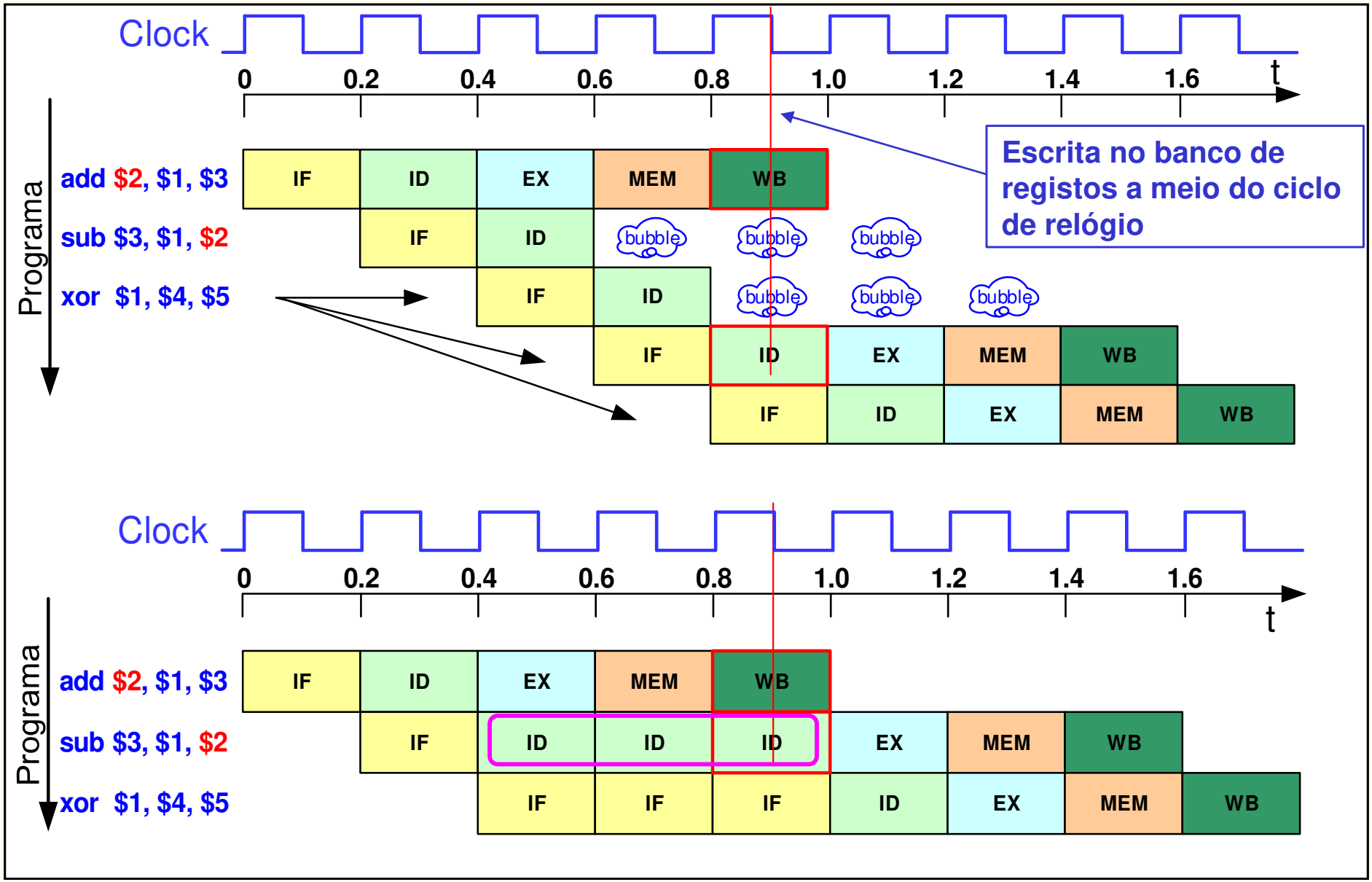
Hazards de dados – *pipeline stalling*

- Para o exemplo anterior, parar a progressão da instrução SUB no estágio ID durante 2 ciclos de relógio é equivalente a introduzir dois NOP entre as duas instruções

```
add   $2,$1,$3   # WB
nop              # MEM
nop              # EX
sub   $3,$1,$2   # ID
```

- Com a escrita no banco de registos feita a meio do ciclo de relógio, a instrução SUB lê o valor atualizado de $2, quando a instrução ADD está na fase WB
- Primeira solução – ***stall* do *pipeline***:
  - **parar a progressão no *pipeline* (*stall)*** da instrução que necessita do valor (e das anteriores), no estágio **ID**, até que a instrução que produz o resultado chegue ao estágio **WB**

Hazards de dados – *pipeline stalling*



Hazards de dados resolvidos com *stalling* (1)

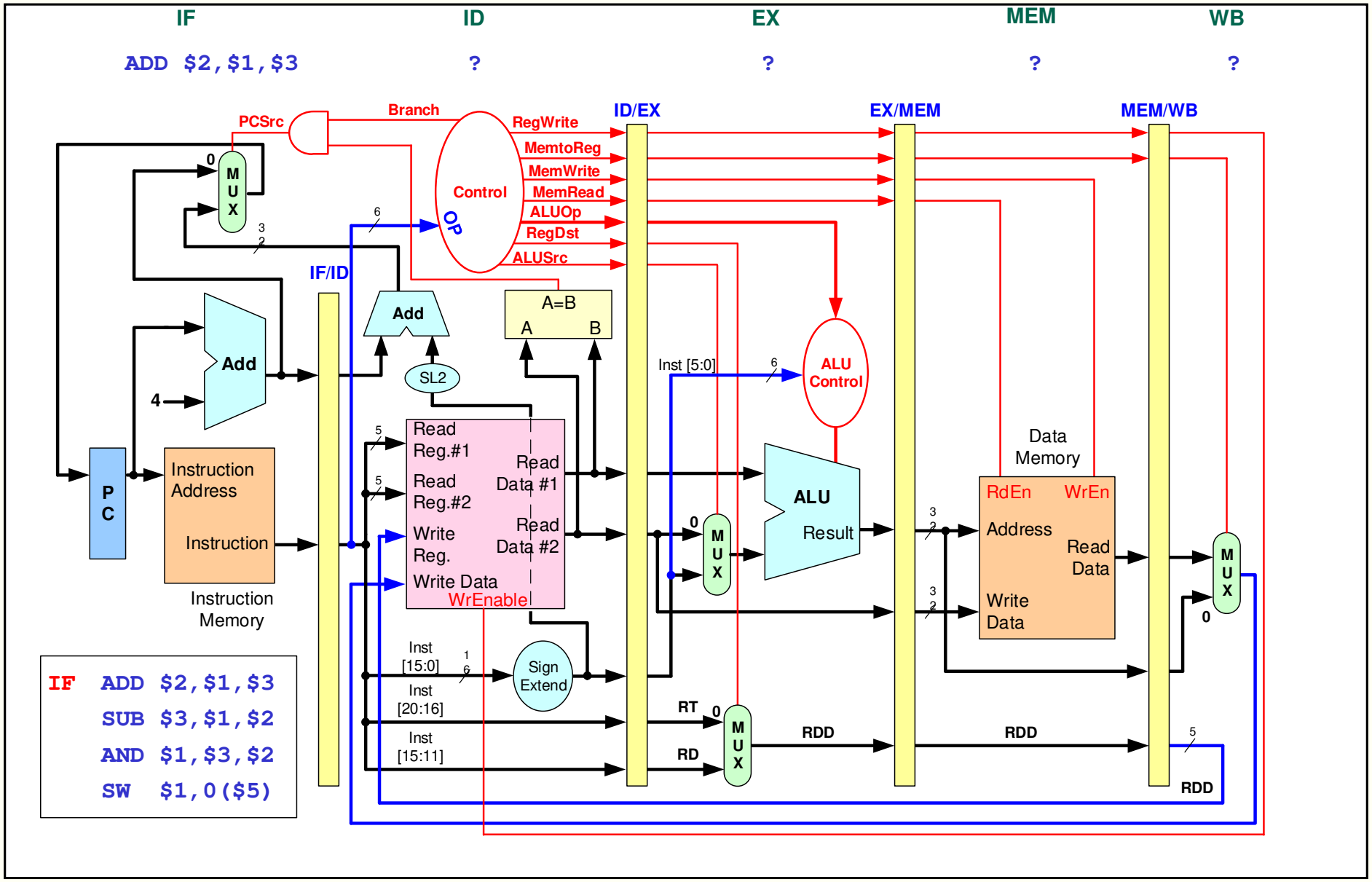
- Exemplo:

```
ADD $2,$1,$3   #
SUB $3,$1,$2   #
AND $1,$3,$2   #
SW  $1,0($5)   #
```

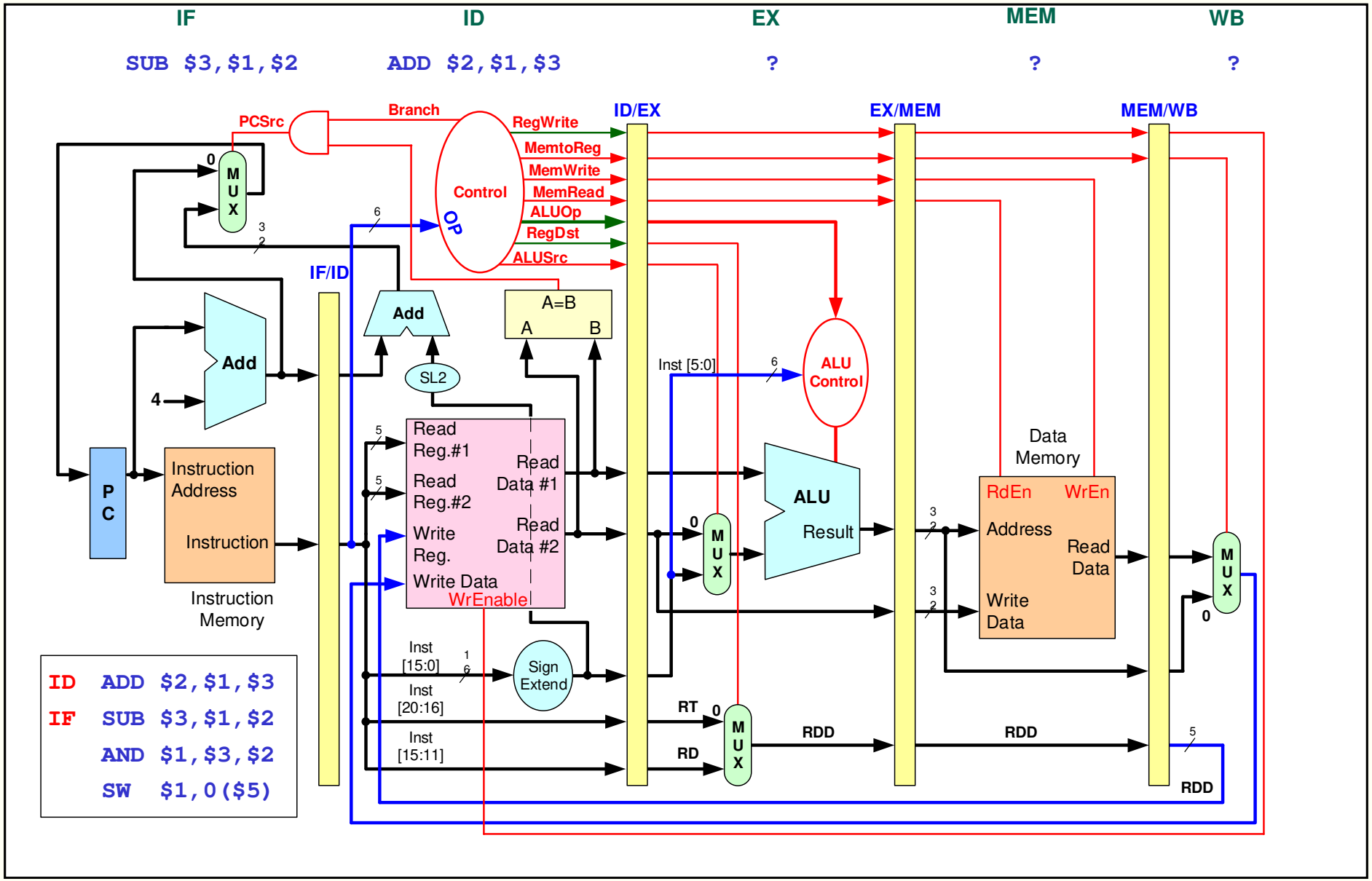
- A sequência de instruções apresenta 3 *hazards* de dados:
  - Na instrução SUB, resultante da dependência do registo **$2**
  - Na instrução AND, resultante da dependência do registo **$3**
  - Na instrução SW, resultante da dependência do registo **$1**

- Cada uma destas situações de *hazard* de dados obriga a fazer o *stall* do *pipeline* durante 2 ciclos de relógio

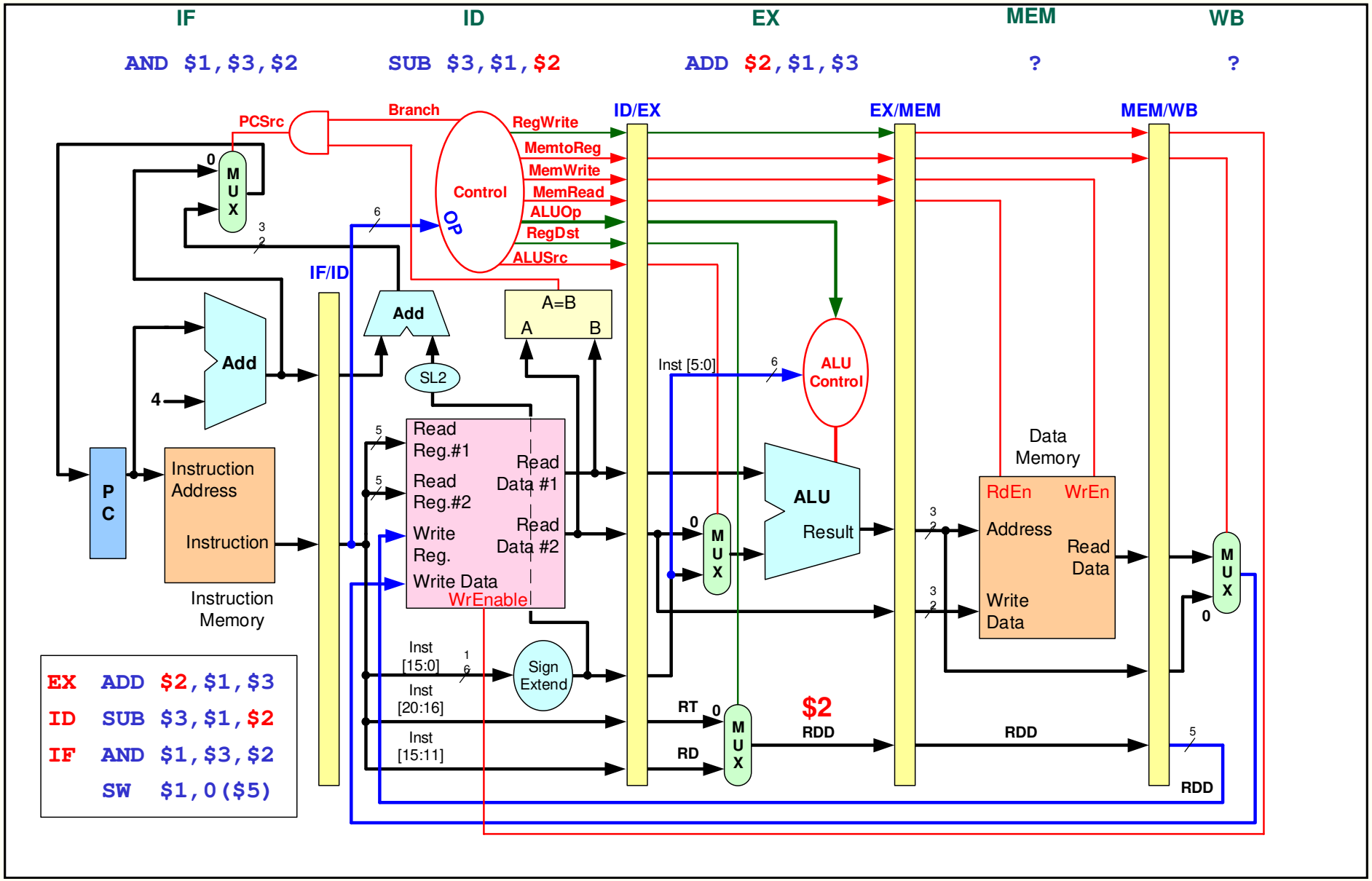
Hazards de dados resolvidos com *stalling* (2)



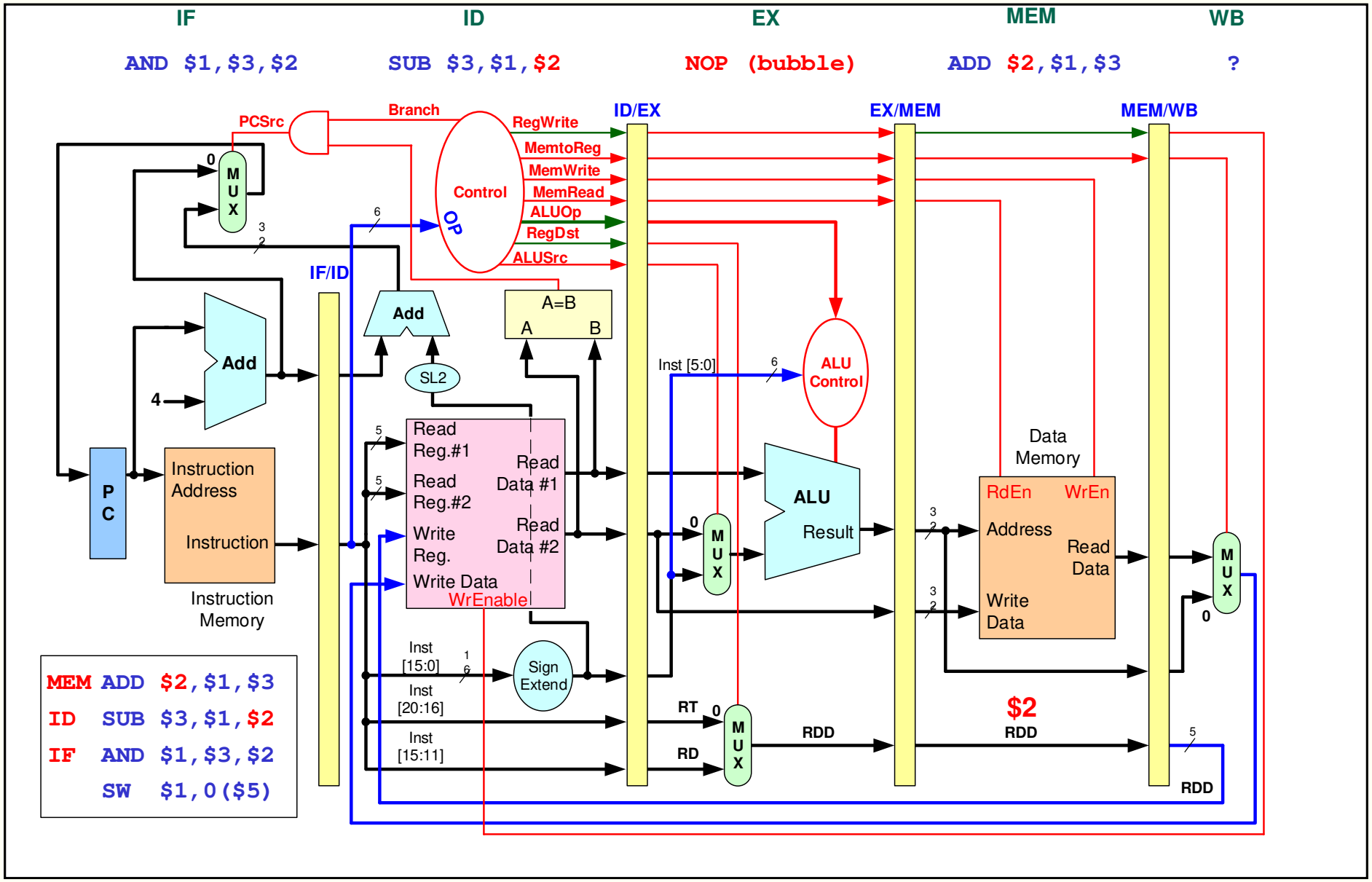
Hazards de dados resolvidos com *stalling* (3)



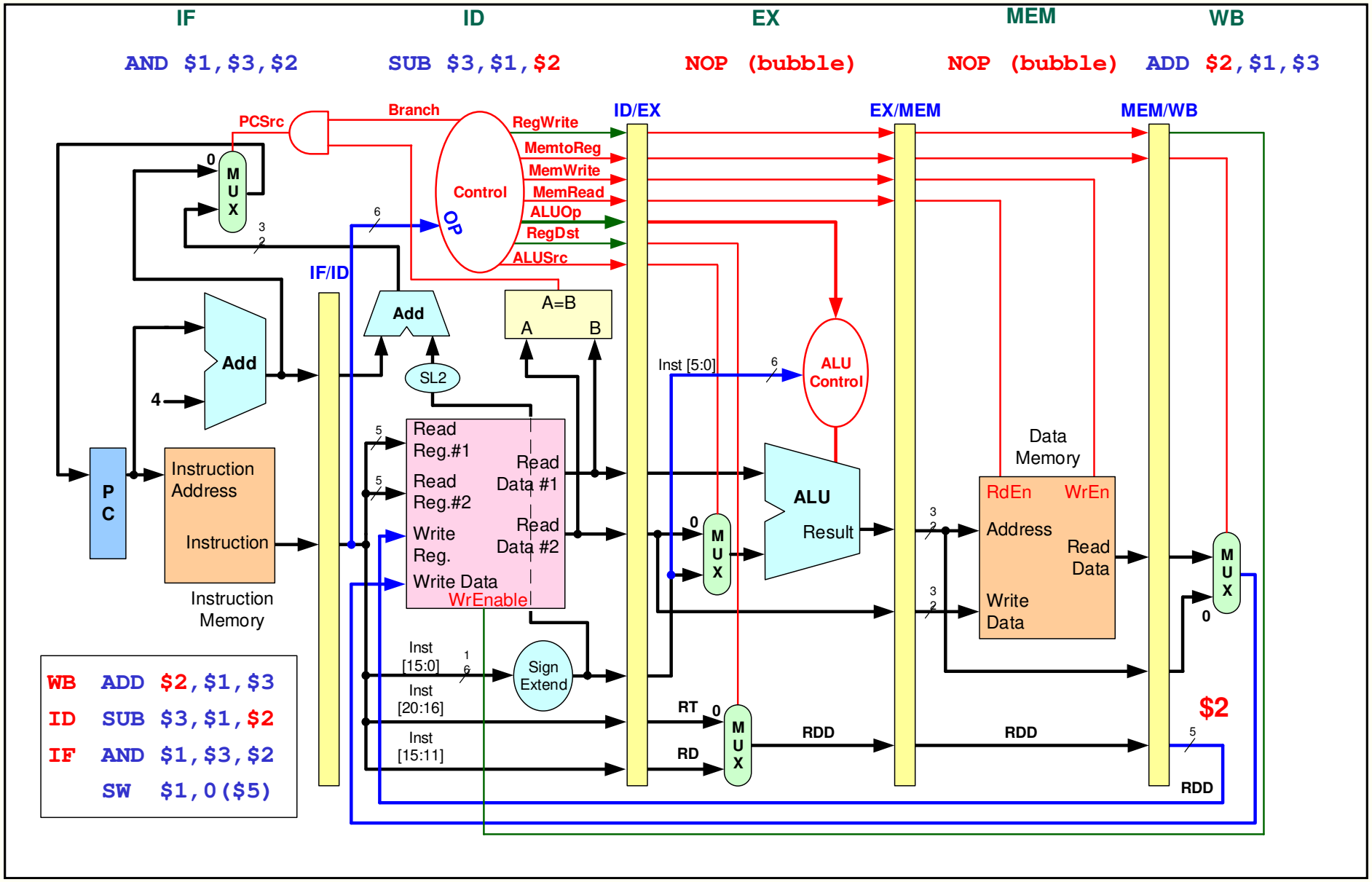
Hazards de dados resolvidos com *stalling* (4)



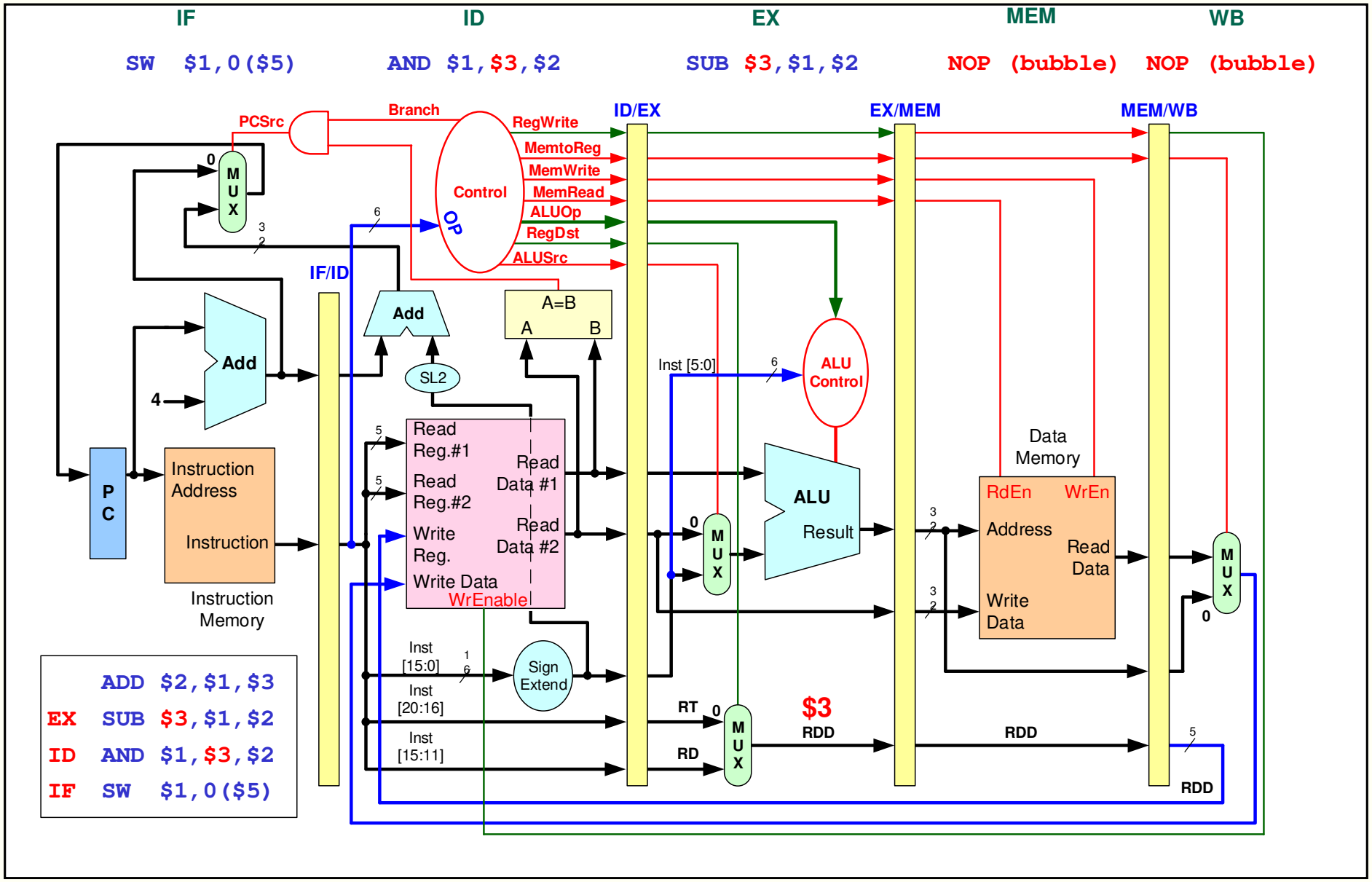
Hazards de dados resolvidos com *stalling* (5)



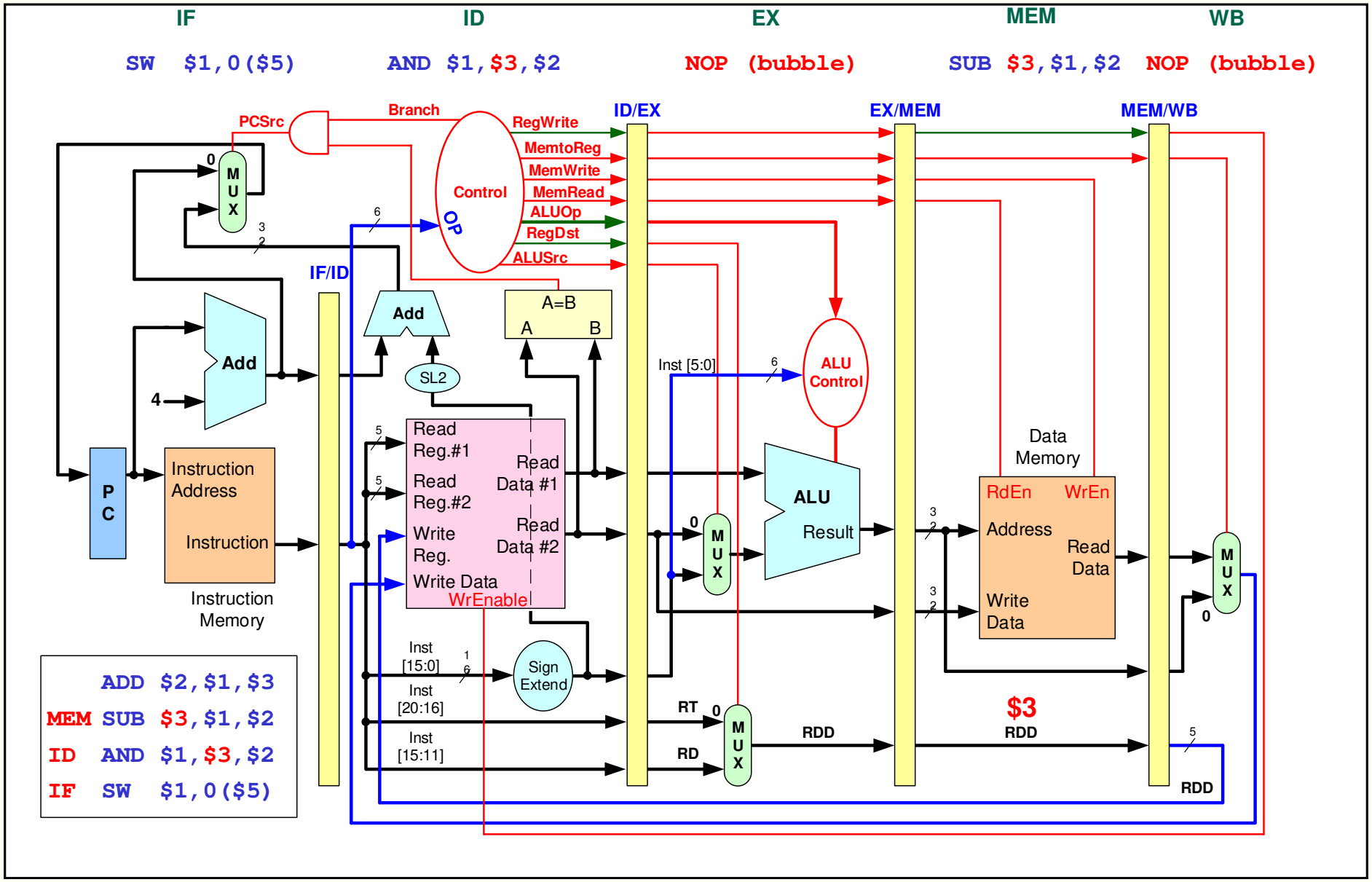
Hazards de dados resolvidos com *stalling* (6)



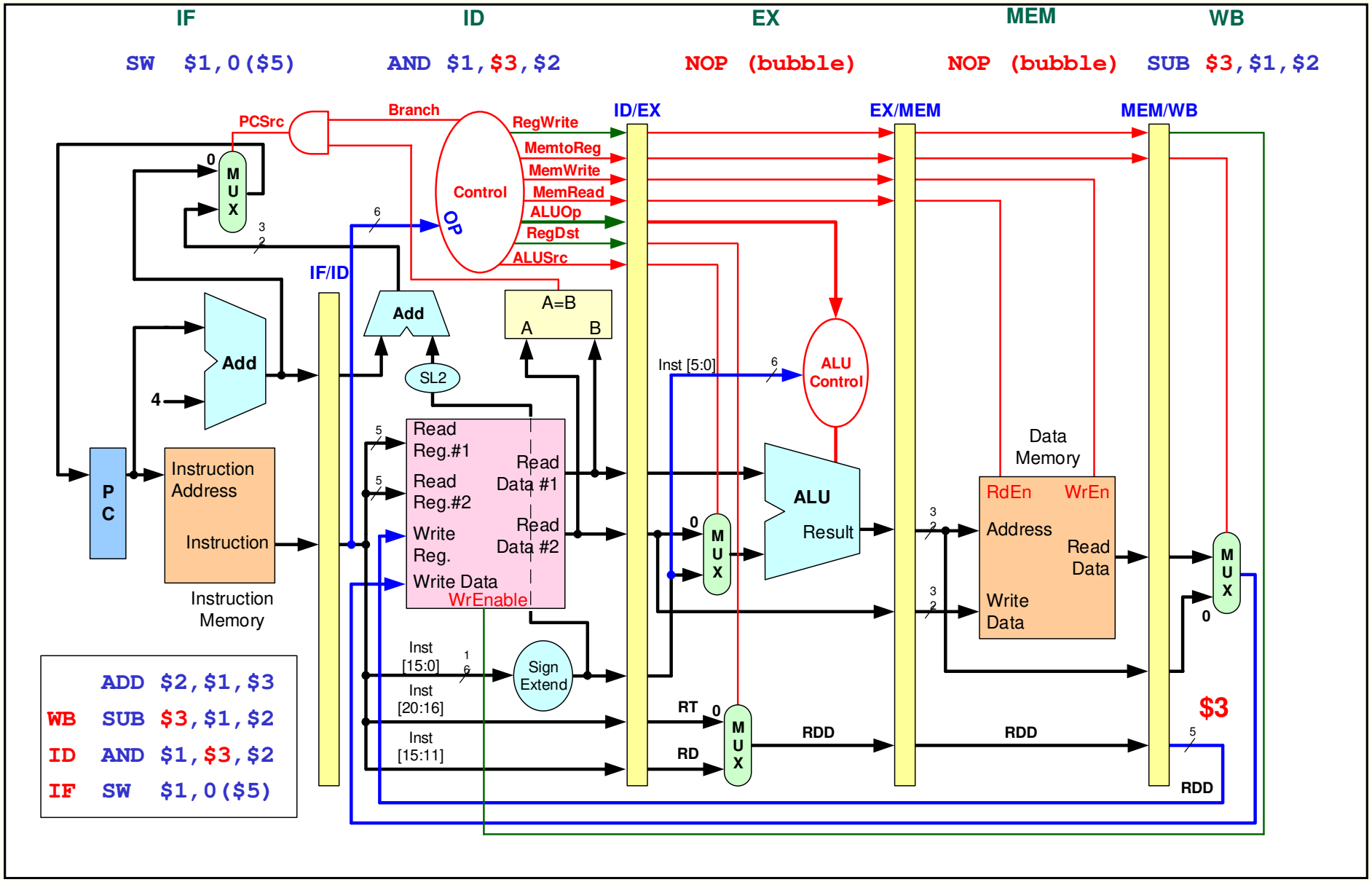
Hazards de dados resolvidos com *stalling* (7)



Hazards de dados resolvidos com *stalling* (8)



Hazards de dados resolvidos com *stalling* (9)



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, até, 2 ciclos de relógio (2 T); para o exemplo anterior, 6 T no total (3 *hazards* de dados):

  Texec_sem_stalls = F + (N-1) = 5 + (4-1) = 8 T

  Texec = F + (N-1) + Nr_stalls = 5 + (4-1) + 6 = 14 T

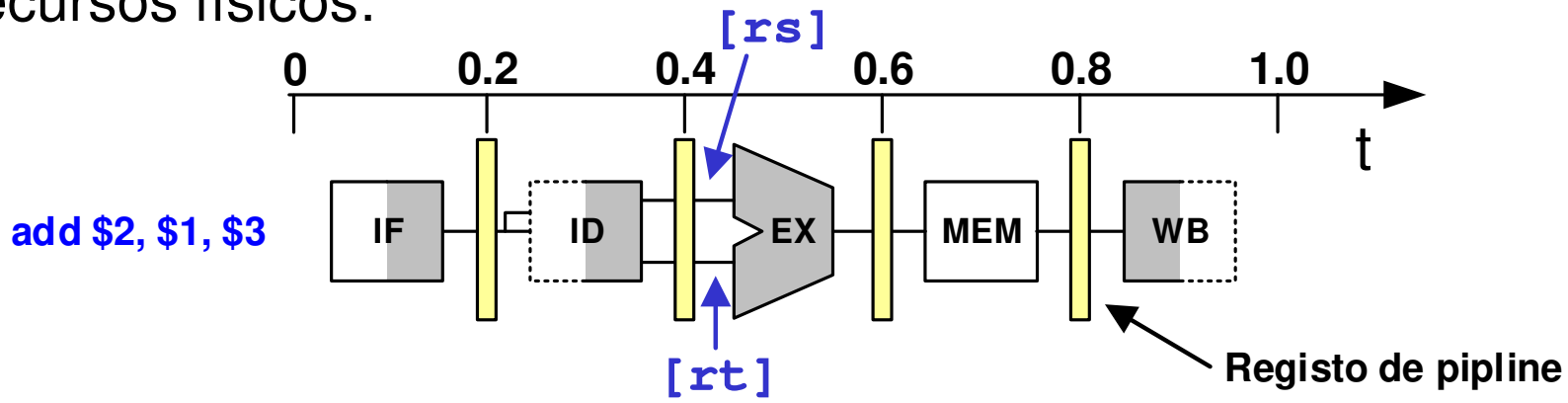
- Qual será então a solução?

Hazards de dados

- 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 instrução que produz o resultado para resolver o *hazard*
- Por exemplo, para as instruções do tipo R, logo que a operação da instrução que vai à frente seja realizada na ALU, (**EX**, 3º estágio), o resultado pode ser disponibilizado para a instrução seguinte
- Esta técnica de disponibilizar um resultado de uma instrução que ainda não terminou para uma instrução que vem a seguir na cadeia de *pipelining*, é designada por ***forwarding*** (ou *bypassing*)

Representação gráfica da cadeia de *pipelining*

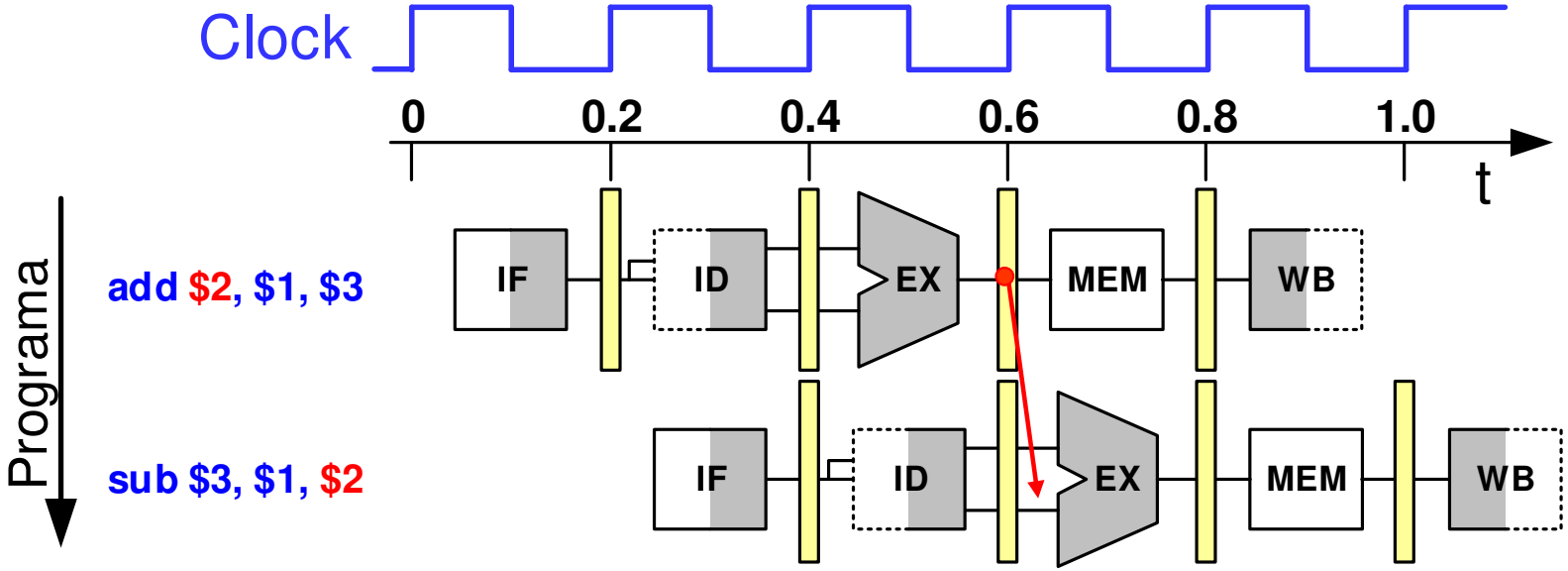
- Nesta representação gráfica usamos símbolos para representar os recursos físicos:



  - IF (memória de instruções), ID (banco de registos), MEM (memória de dados), WB (banco de registos)
  - Metade cinzenta à direita indica uma operação de leitura do elemento de estado
  - Metade cinzenta à esquerda indica uma operação de escrita no elemento de estado
  - Um quadrado branco indica que o elemento de estado não está envolvido na execução da instrução

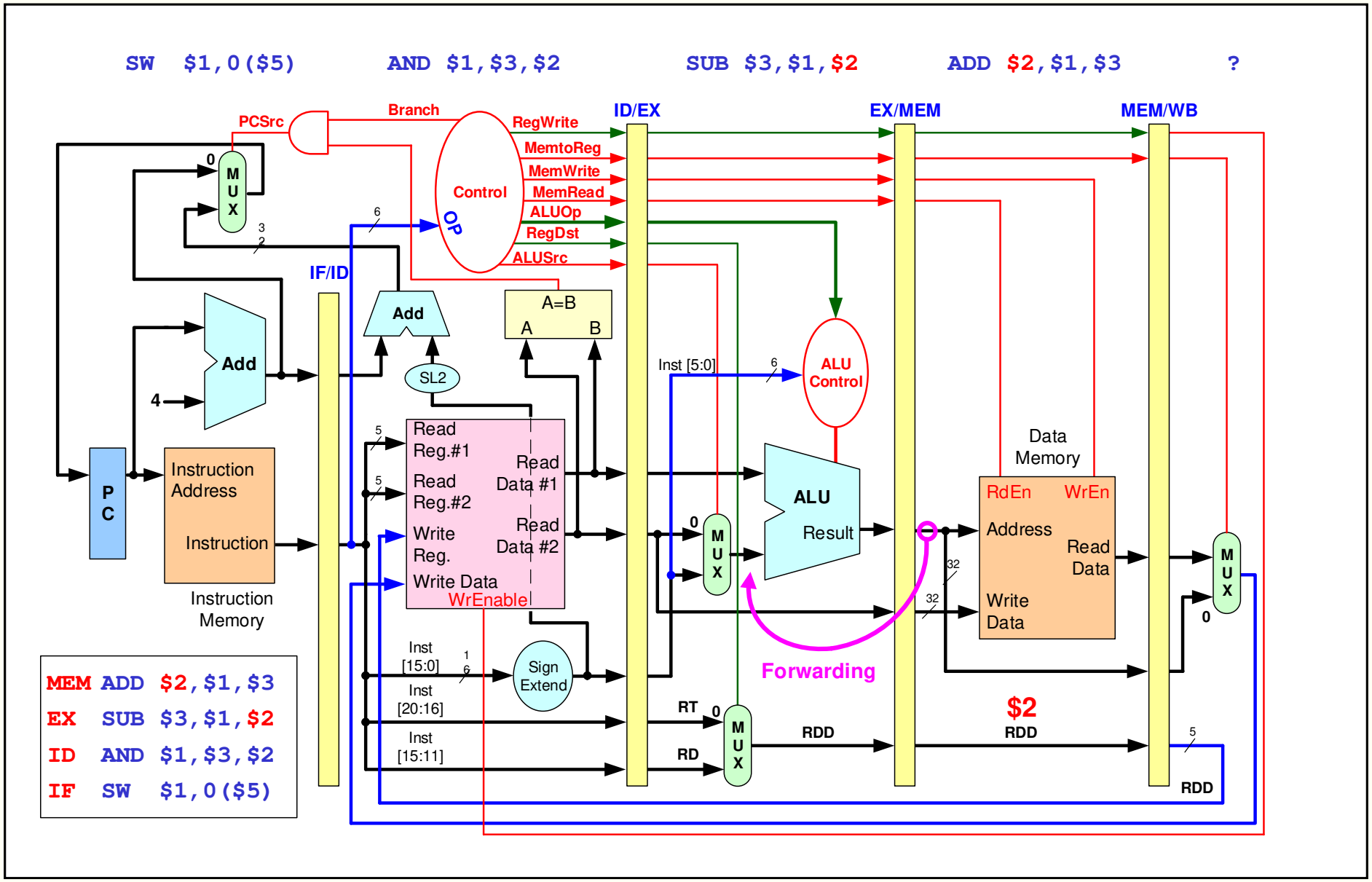
Hazards de dados

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



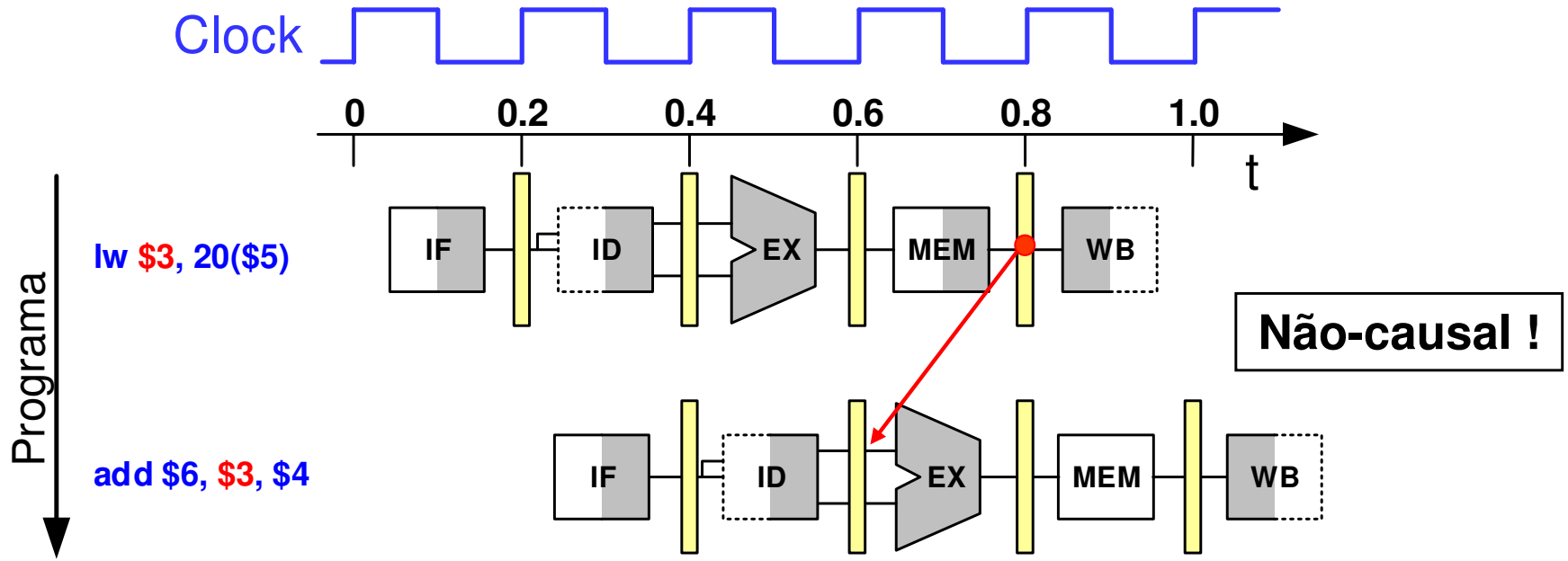
- O ***forwarding*** do valor presente no registo **`EX/MEM`** (resultado da instrução ADD) para a segunda entrada da ALU (estágio **`EX`**, instrução SUB) resolve o *hazard* de dados
  - Designamos esta operação por *"forwarding* de **`EX/MEM`** para **`EX`**" (para a segunda entrada da ALU)

Hazards de dados - *forwarding*



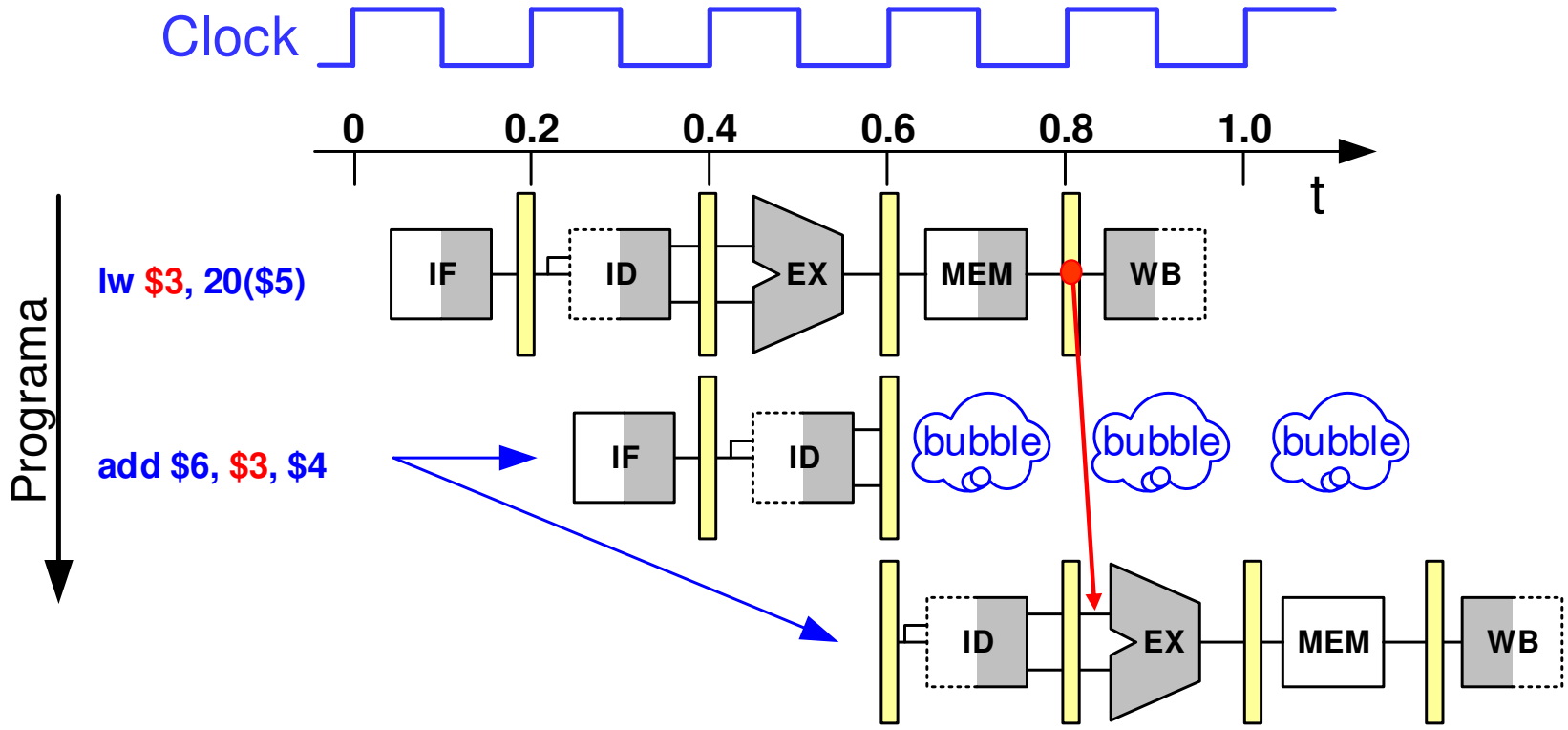
Hazards de dados

- 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 a situação do slide anterior, é necessário:
  1. Fazer o ***stall*** do *pipeline* durante um ciclo de relógio



  2. Fazer o ***forwarding*** do registo **`MEM/WB`** para o estágio **`EX`**, para a primeira entrada da ALU

Hazards de dados – reordenação de instruções

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

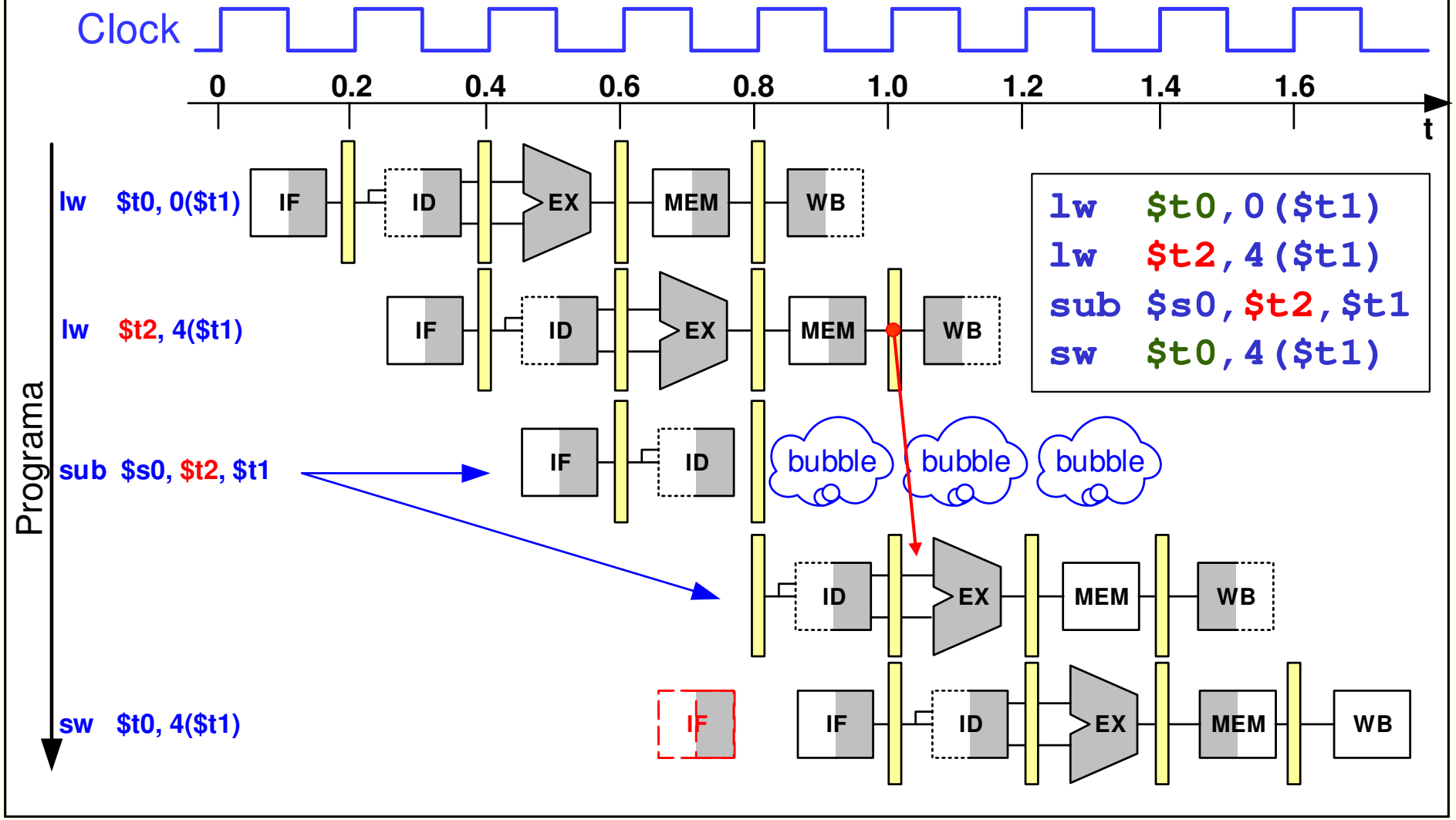
```
lw   $t0,0($t1)
lw   $t2,4($t1)
sub  $s0,$t2,$t1 # Stalling por hazard de dados (1T)
sw   $t0,4($t1)
```

- 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)
```

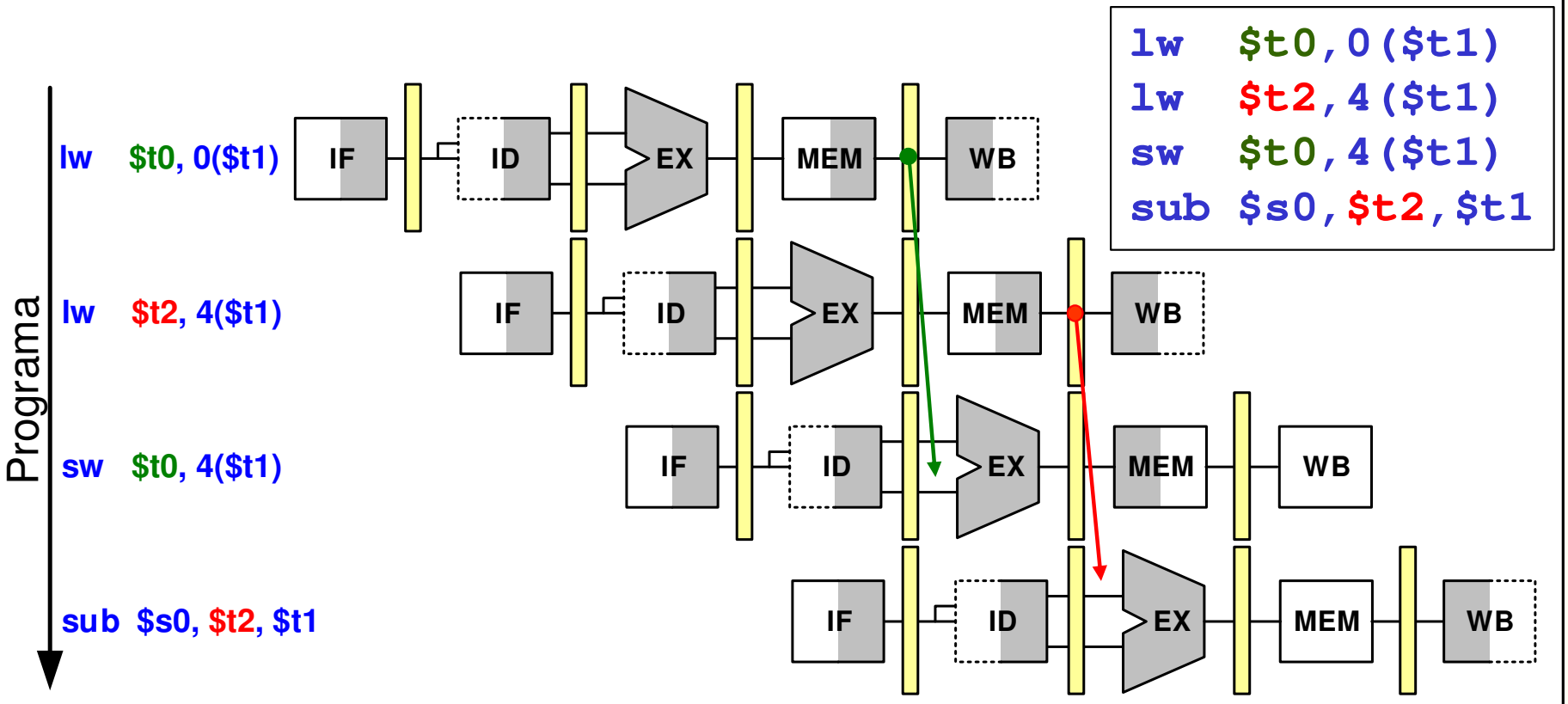
Hazards de dados – exemplo que gera *stalling*

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



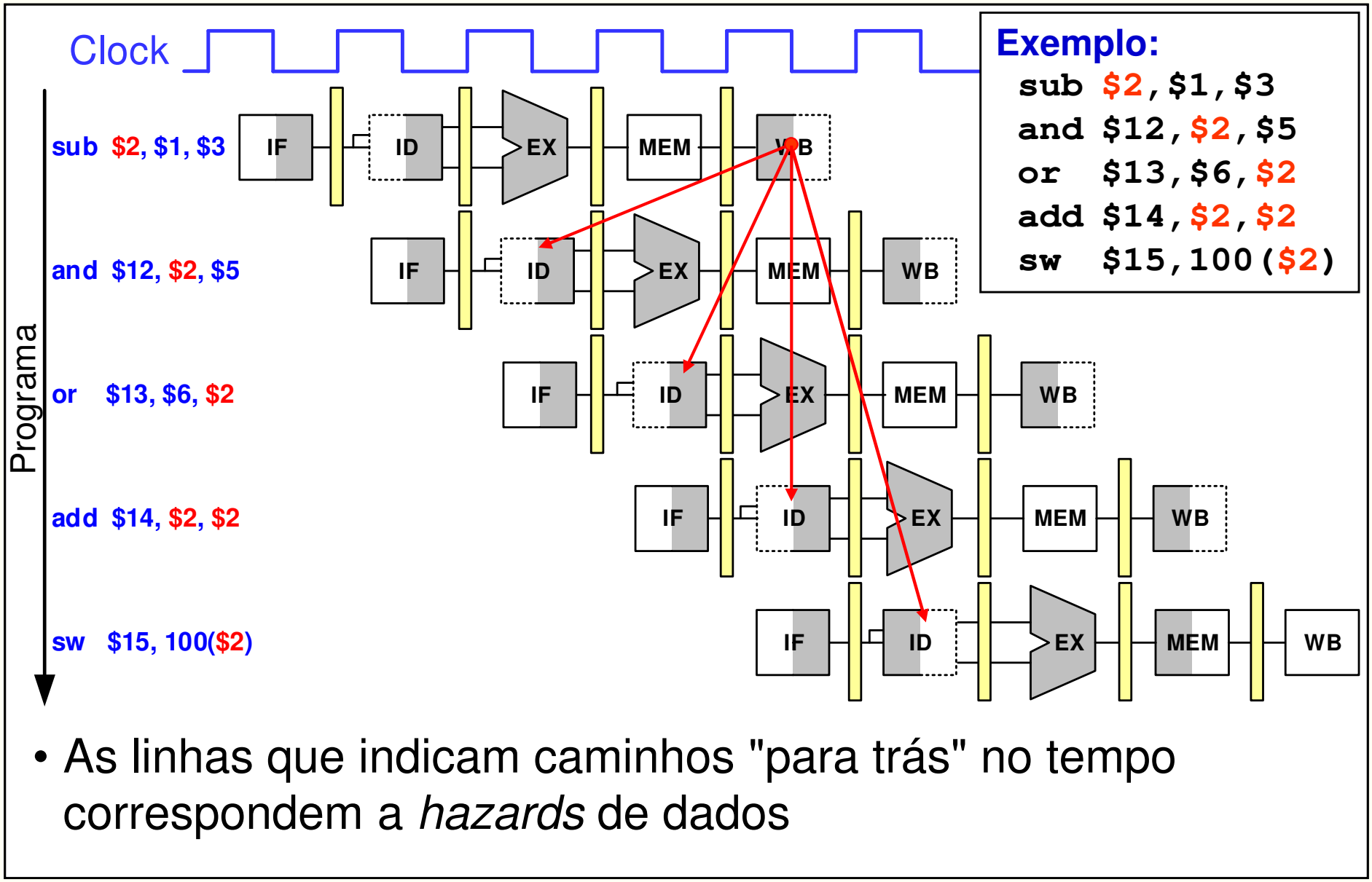
Hazards de dados

- A situação de *stalling* foi evitada pelo compilador/*assembler* através de reordenação. A reordenação gera um segundo *hazard* de dados que também é resolvido por *forwarding*



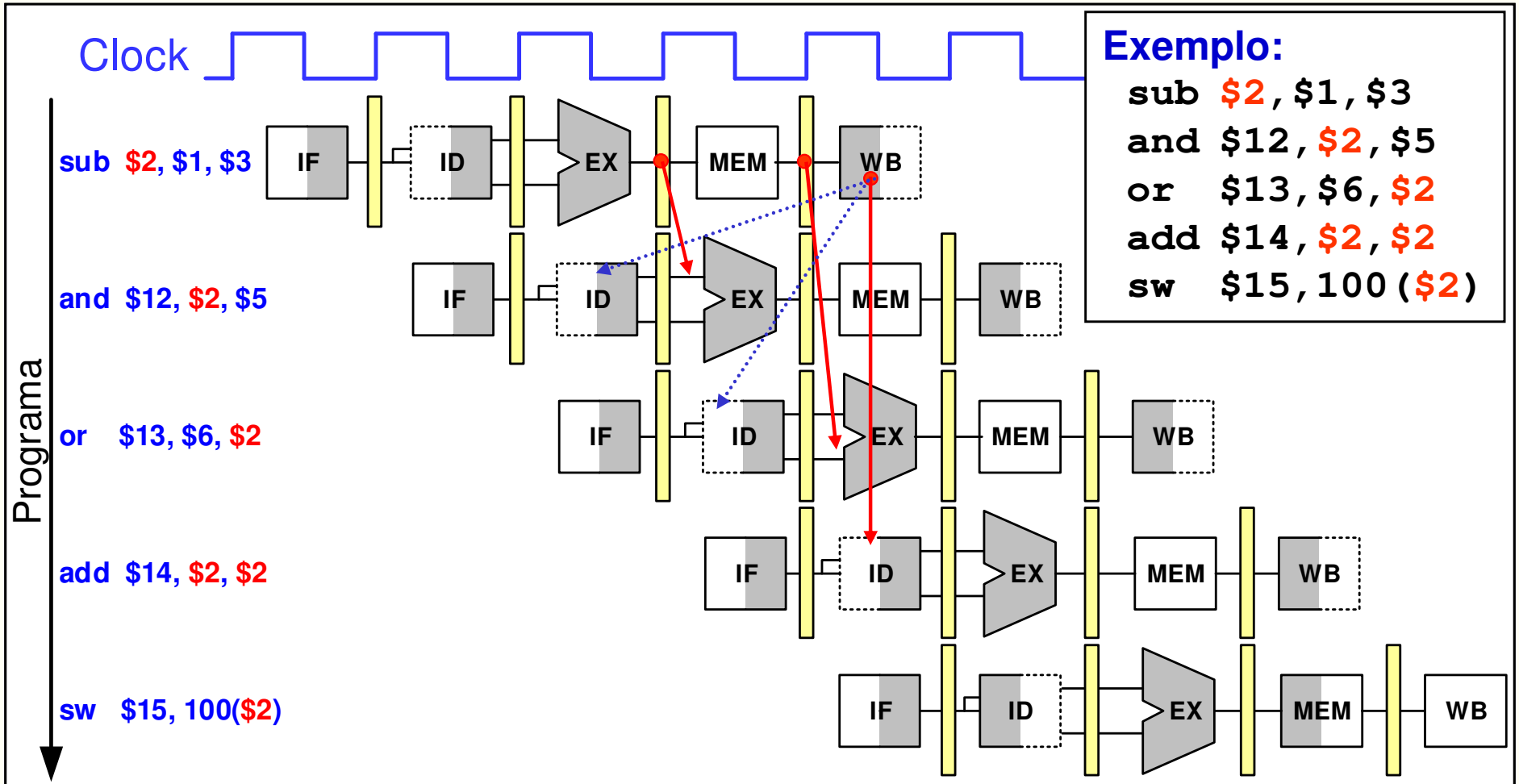
- A sequência reordenada executa em menos 1 ciclo de relógio

Hazards de dados – exemplo



- As linhas que indicam caminhos "para trás" no tempo correspondem a *hazards* de dados

Hazards de dados – exemplo



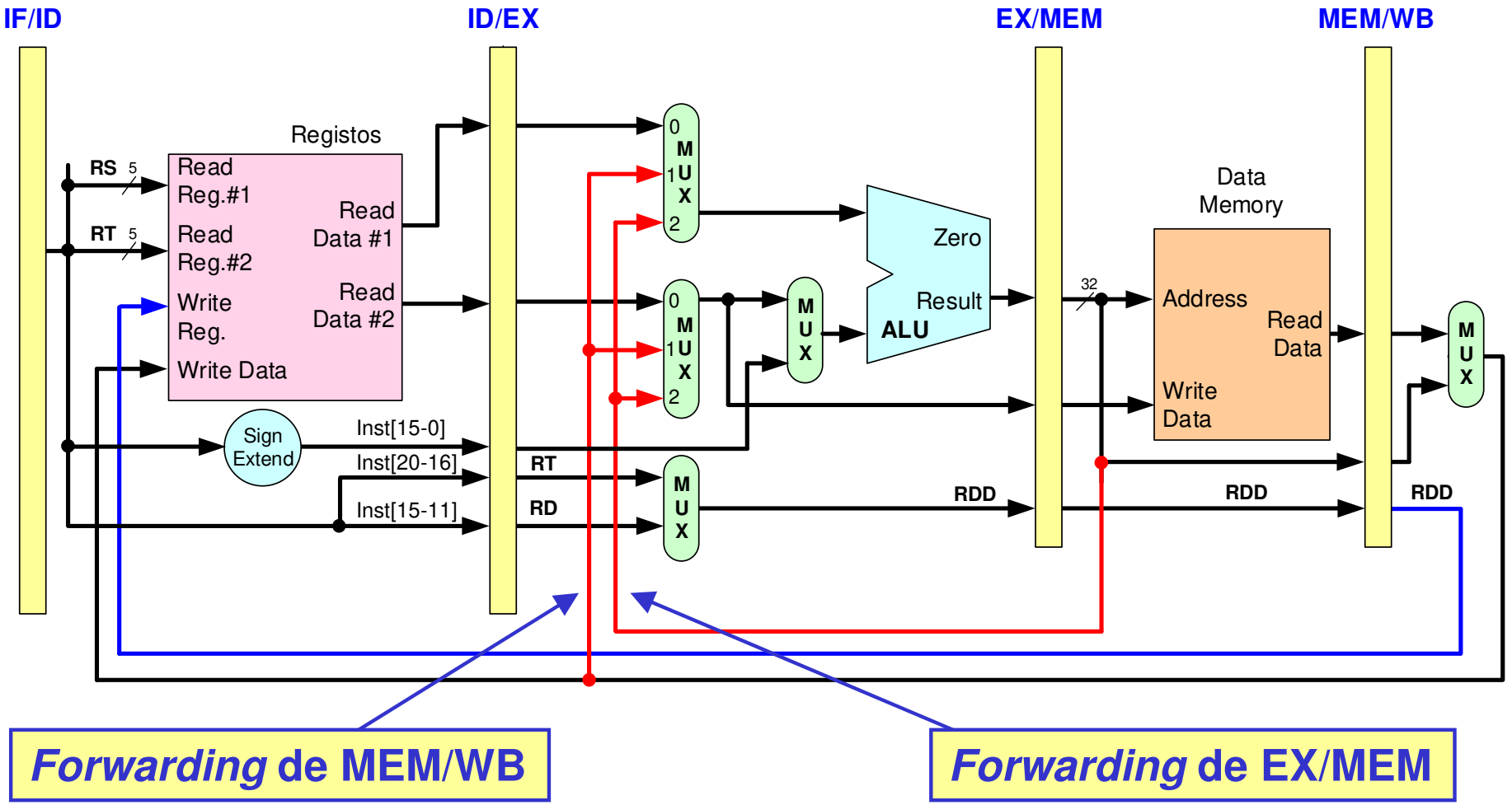
- A dependência existente entre as etapas WB e ID é resolvida fazendo a escrita do *Register File* a meio do ciclo de relógio, permitindo que a leitura seja realizada na 2ª metade do ciclo

Hazards de dados – implementação do *forwarding*

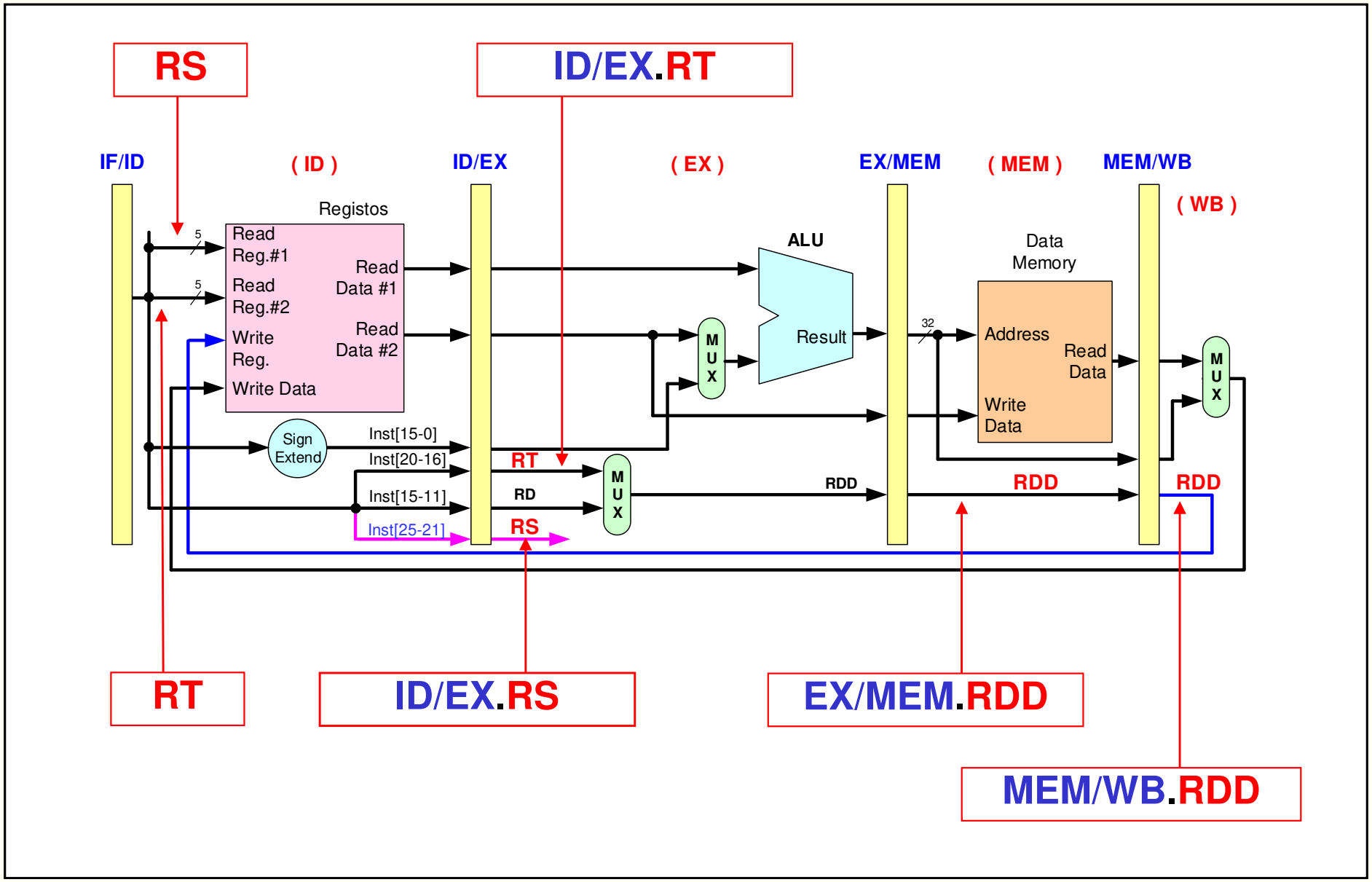
- Para resolver um *hazard* de dados através de ***forwarding*** é necessário:
  - **Detetar** a situação de *hazard*
  - **Encaminhar** o valor que se encontra num estágio mais avançado do *pipeline* para onde ele é necessário
- A resolução de uma parte significativa dos *hazards* de dados é feita através do encaminhando de valores para o estágio **`EX`**:
  - *forwarding* de **`EX/MEM`** para **`EX`** e de **`MEM/WB`** para **`EX`**
- As instruções de *branch* necessitam dos valores corretos dos registos no estágio **`ID`**
  - *forwarding* de **`EX/MEM`** para **`ID`**

Hazards de dados – encaminhamento

- *Forwarding* de **EX/MEM** para **EX** e de **MEM/WB** para **EX**



Hazards de dados – deteção



Hazards de dados – deteção

- As situações de *hazard* de dados, em que há necessidade de encaminhar valores para o estágio **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:

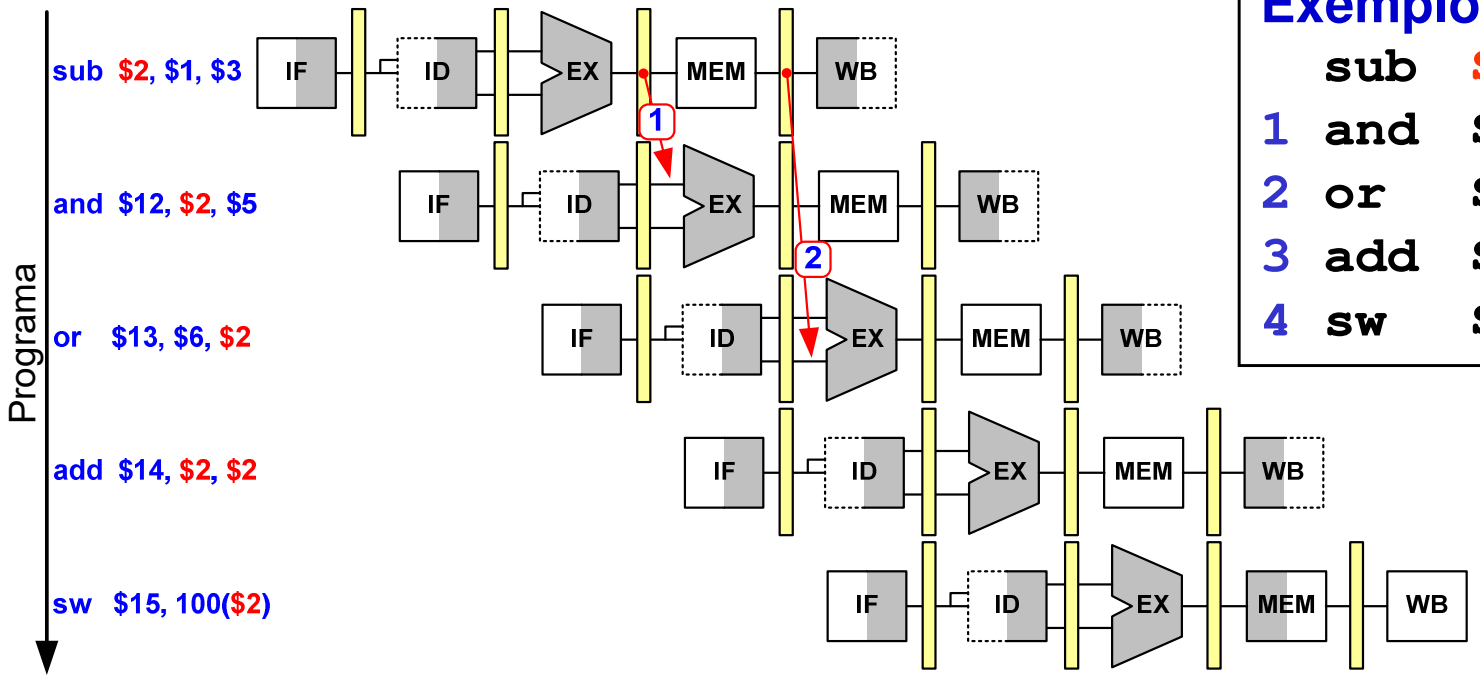
    MEM/WB.RDD == **ID/EX.RS**, e/ou

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

```
WB  add $1,$2,$3
M   add $6,$2,$3
EX  sub $4,$5,$1
```

Hazards de dados – deteção

- A situação 3 é resolvida sem *forwarding* (não se considera *hazard*)
- A situação 4 não corresponde a um *hazard* de dados



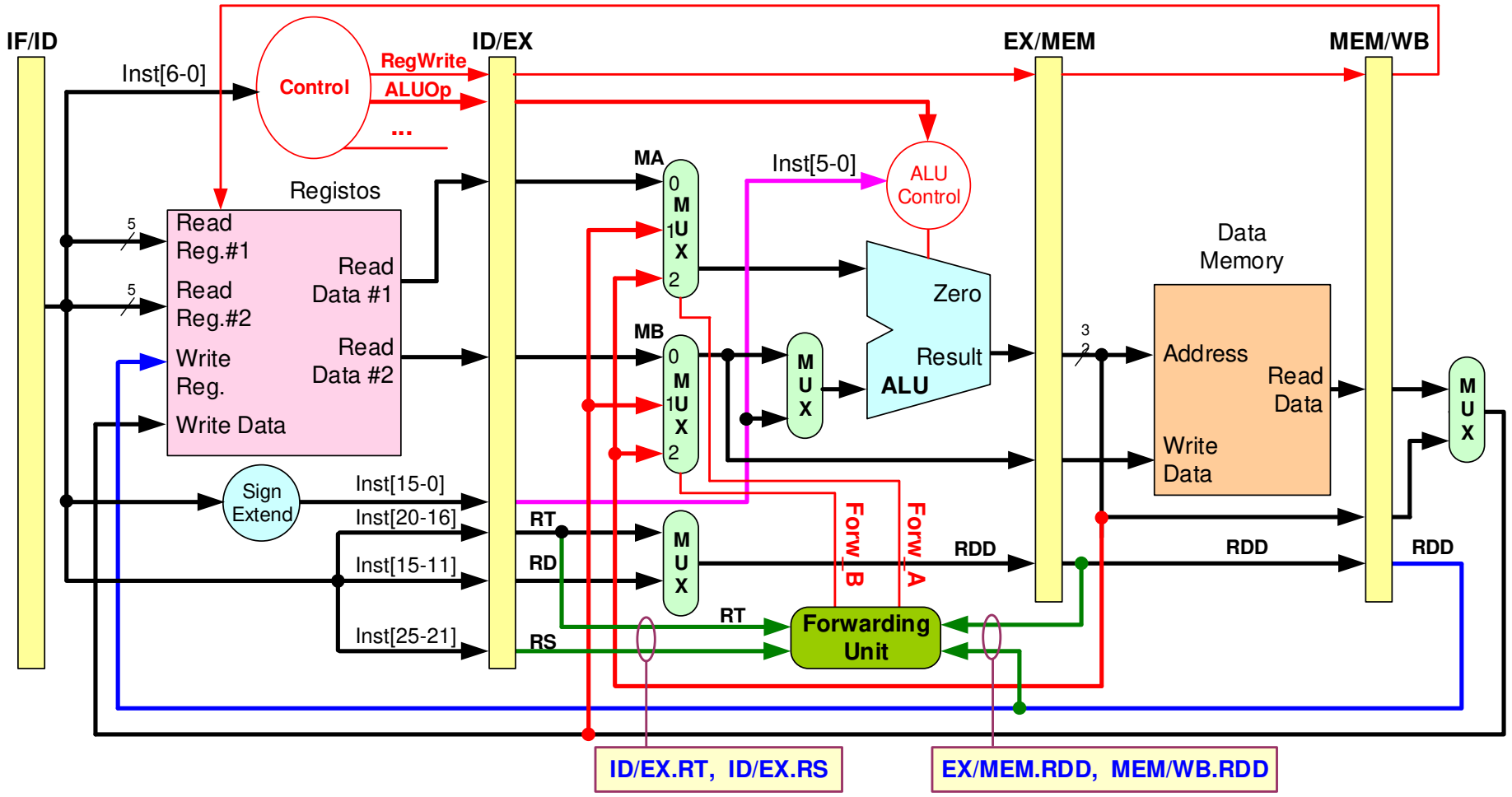
Exemplo:

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

- As situações de *hazard* de dados 1 e 2 podem ser detetadas por:
  1. **EX/MEM.RDD == ID/EX.RS** (**EX/MEM.RDD = $2, ID/EX.RS = $2**)
  2. **MEM/WB.RDD == ID/EX.RT** (**MEM/WB.RDD = $2, ID/EX.RT = $2**)

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

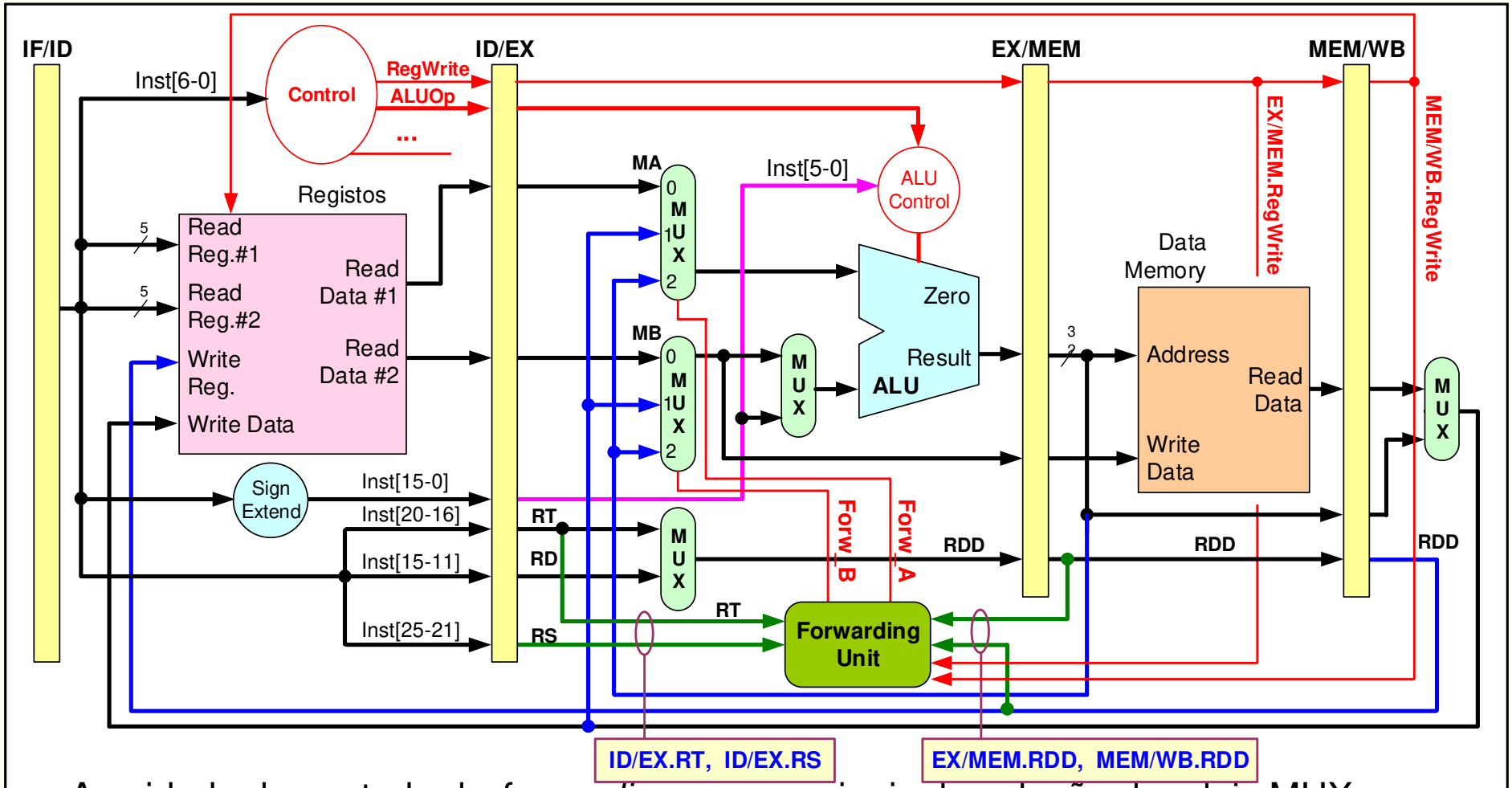
    (EX/MEM.RegWrite == 1) and (EX/MEM.RDD == ID/EX.RT)
  - Instrução na fase **WB** que escreve o resultado num registo (**`RegWrite='1'`**) igual ao registo operando de uma instrução que se encontra na fase **EX**:

    (MEM/WB.RegWrite == 1) and (MEM/WB.RDD == ID/EX.RS)

    e/ou

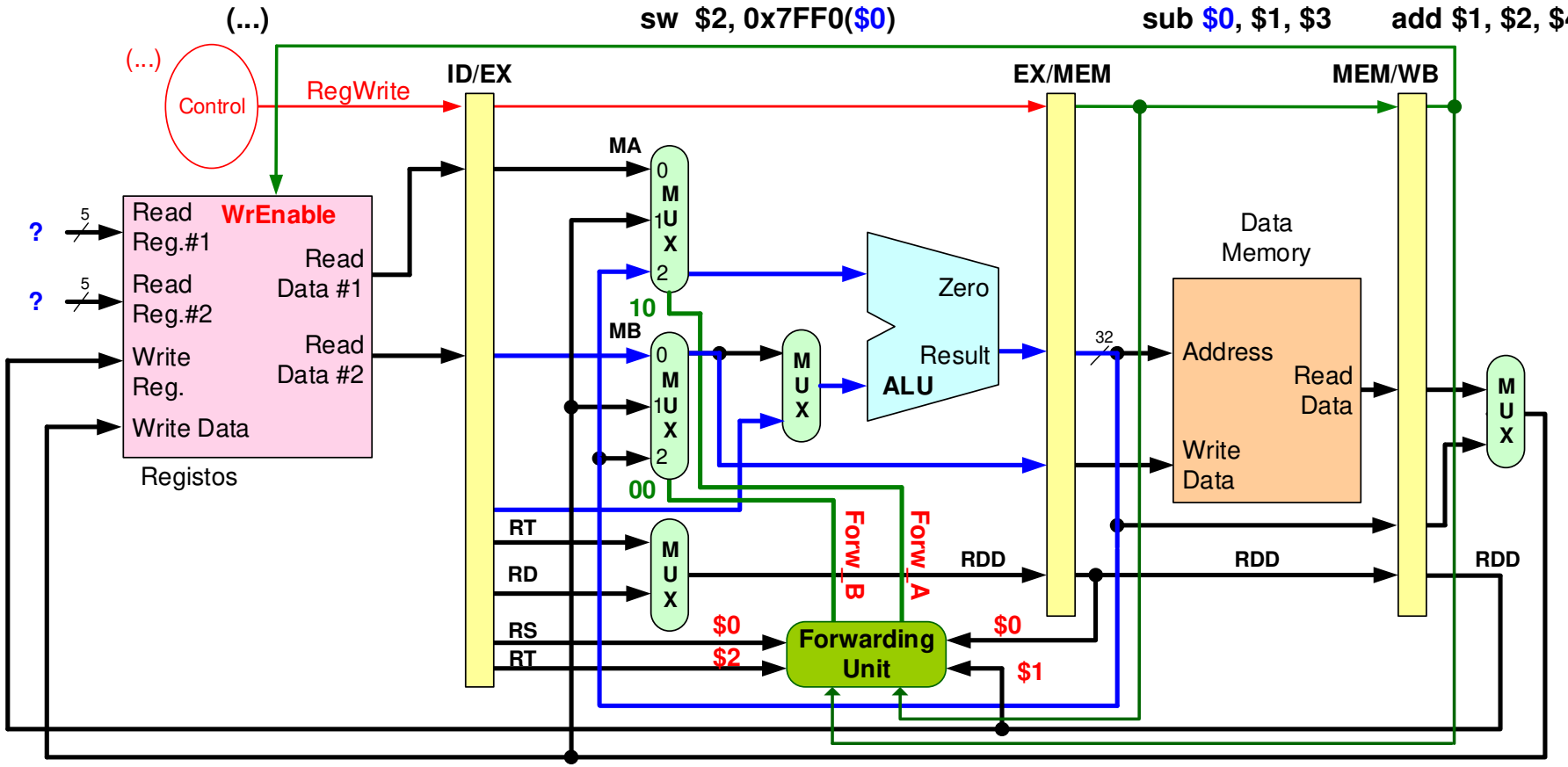
    (MEM/WB.RegWrite == 1) and (MEM/WB.RDD == ID/EX.RT)

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 r

```
architecture Behavioral of ForwardingUnit is
begin
  process(all)
  begin
    Forw_A <= "00"; -- no hazard
    Forw_B <= "00"; -- no hazard
    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;
```

```
1  add   $2,$3,$5
2  add   $2,$2,$6
3  sub   $4,$2,$7
```

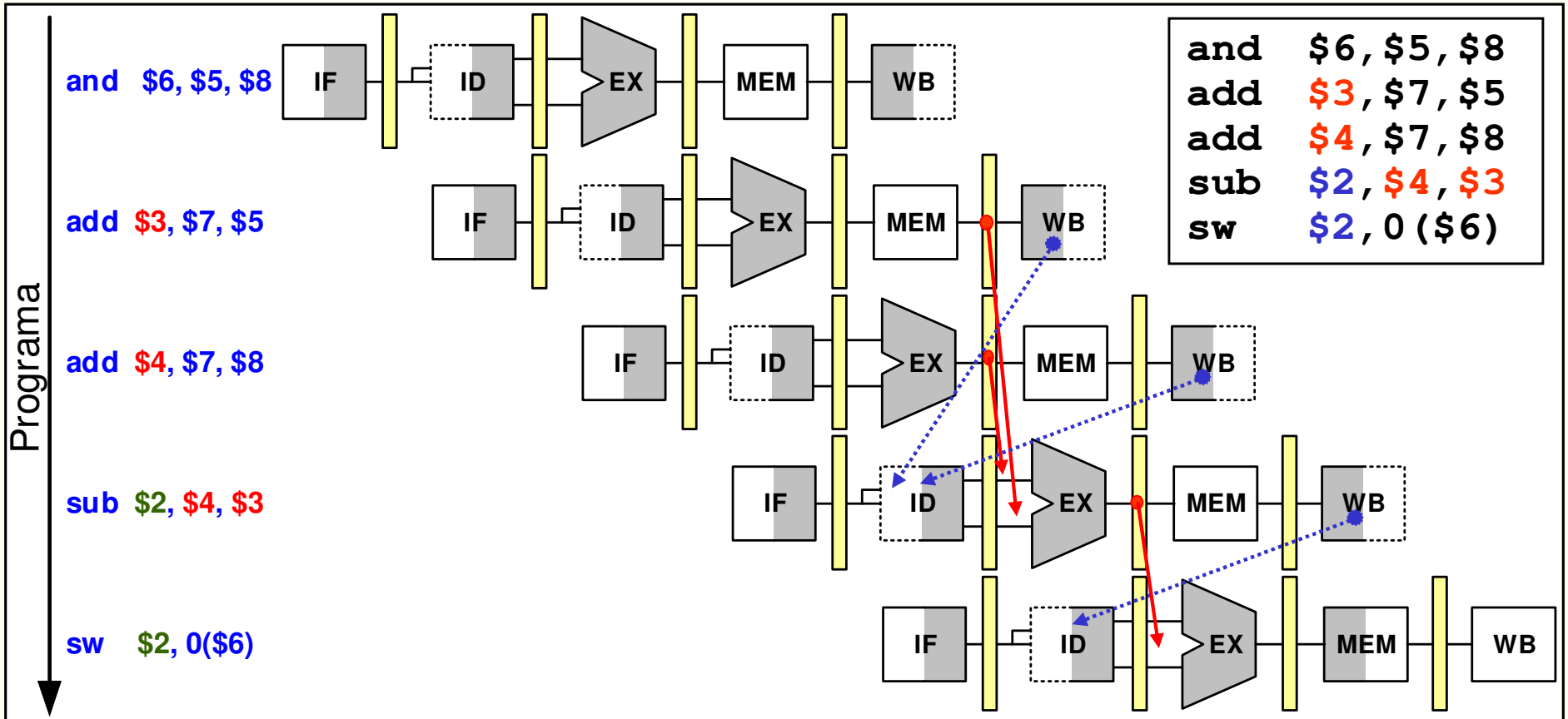
00 – encaminhar valor lido do banco de registos
01 – encaminhar o valor proveniente do registo **MEM/WB**
10 – encaminhar o valor proveniente do registo **EX/MEM**

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($6)   # Hazard de dados: $2
```

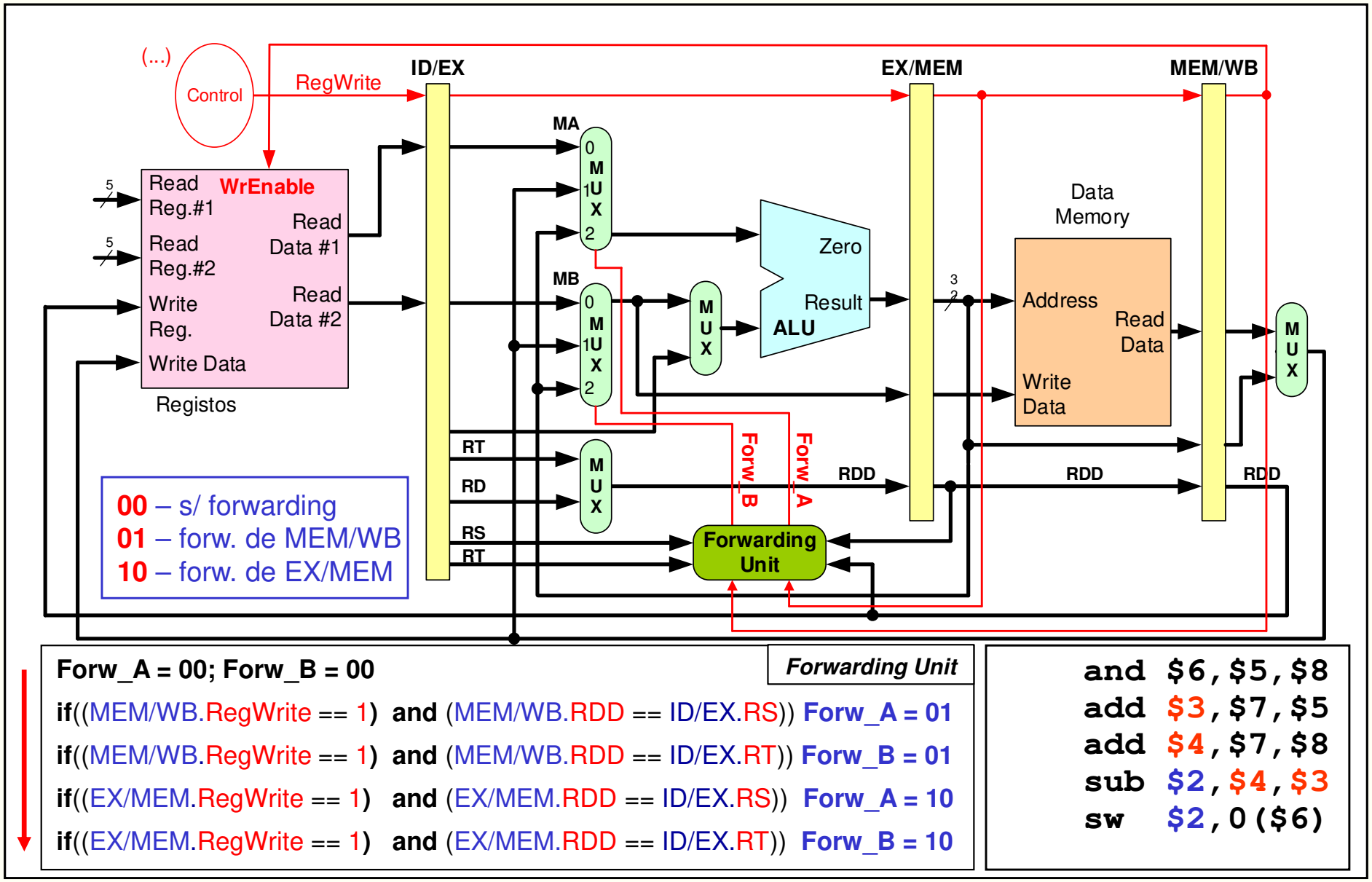
- A instrução "`sub $2,$4,$3`" apresenta duas situações de *hazards* de dados:
  - dependência do registo `$4` (`add $4,$7,$8`)
  - dependência do registo `$3` (`add $3,$7,$5`)
- A instrução "`sw $2,0($6)`" apresenta igualmente uma situação de *hazard* de dados (dependência em `$2`, `sub $2,$4,$3`)

Exemplo de *forwarding*

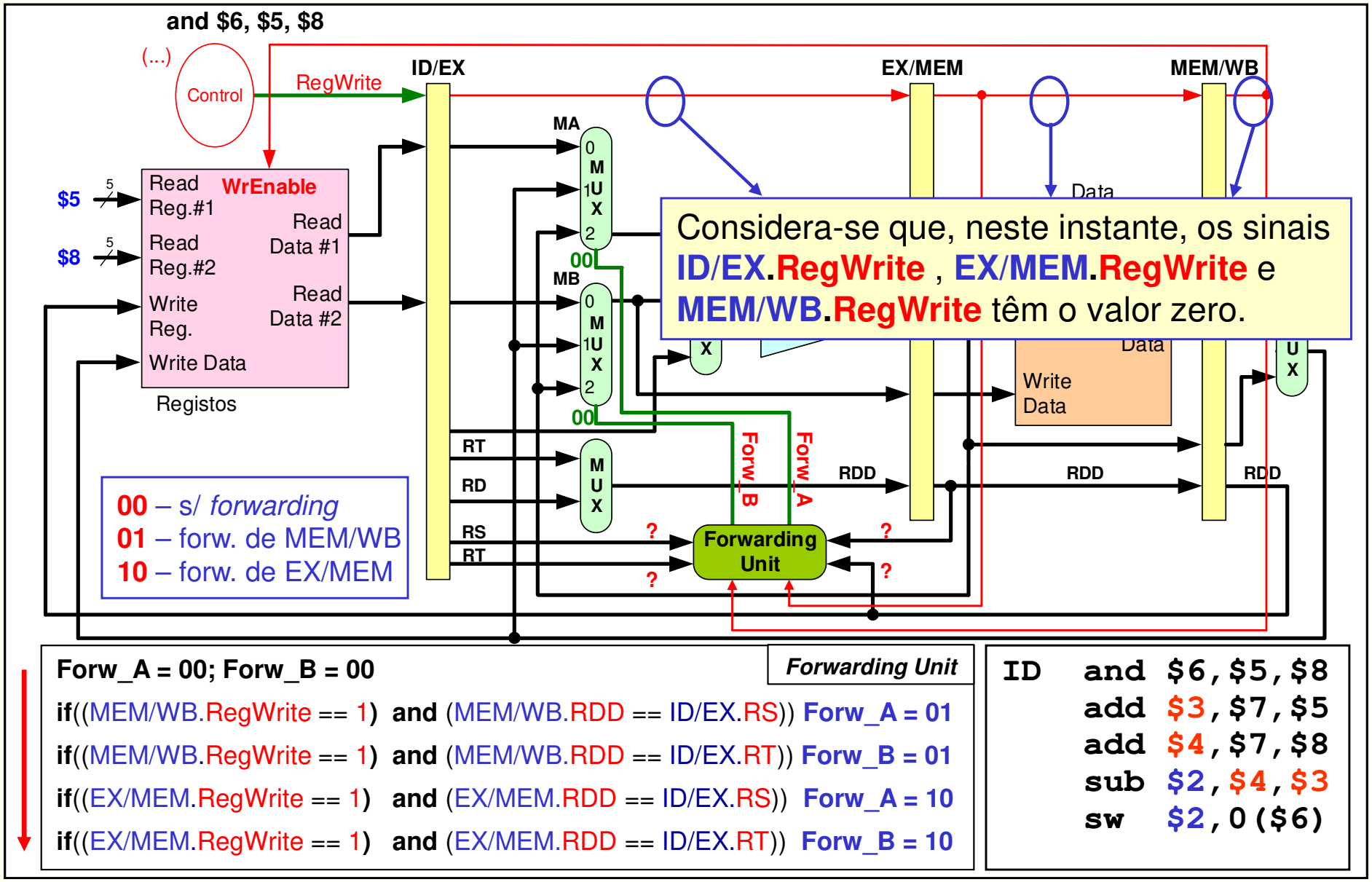


- **Dependência de $3**: *forwarding* de **MEM/WB** para a 2ª entrada da ALU (registo codificado no campo **RT**)
- **Dependência de $4**: *forwarding* de **EX/MEM** para a 1ª entrada da ALU (registo codificado no campo **RS**)
- **Dependência de $2**: *forwarding* de **EX/MEM** para o caminho correspondente à 2ª entrada da ALU (registo codificado no campo **RT**)

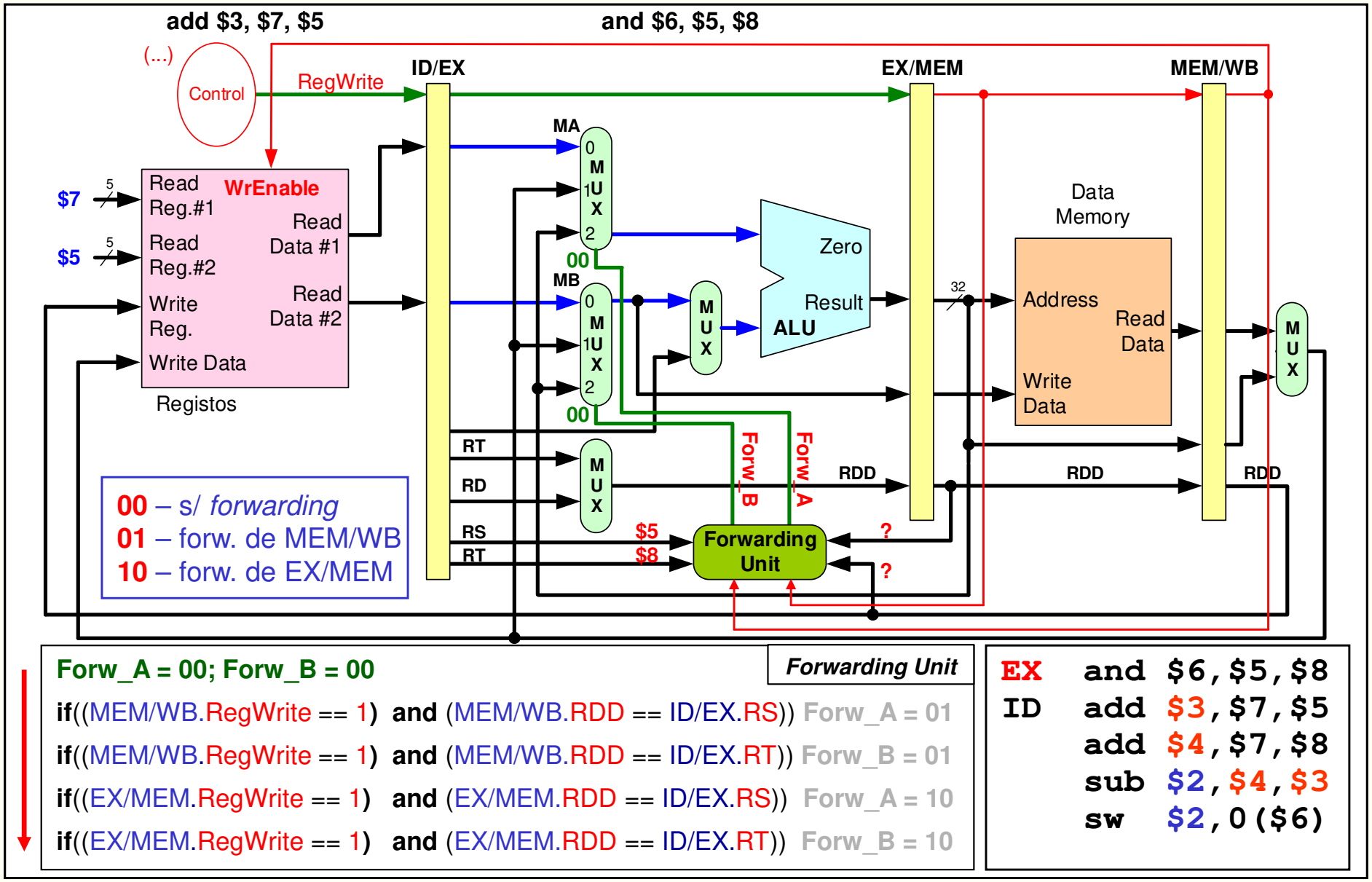
Exemplo de *forwarding*



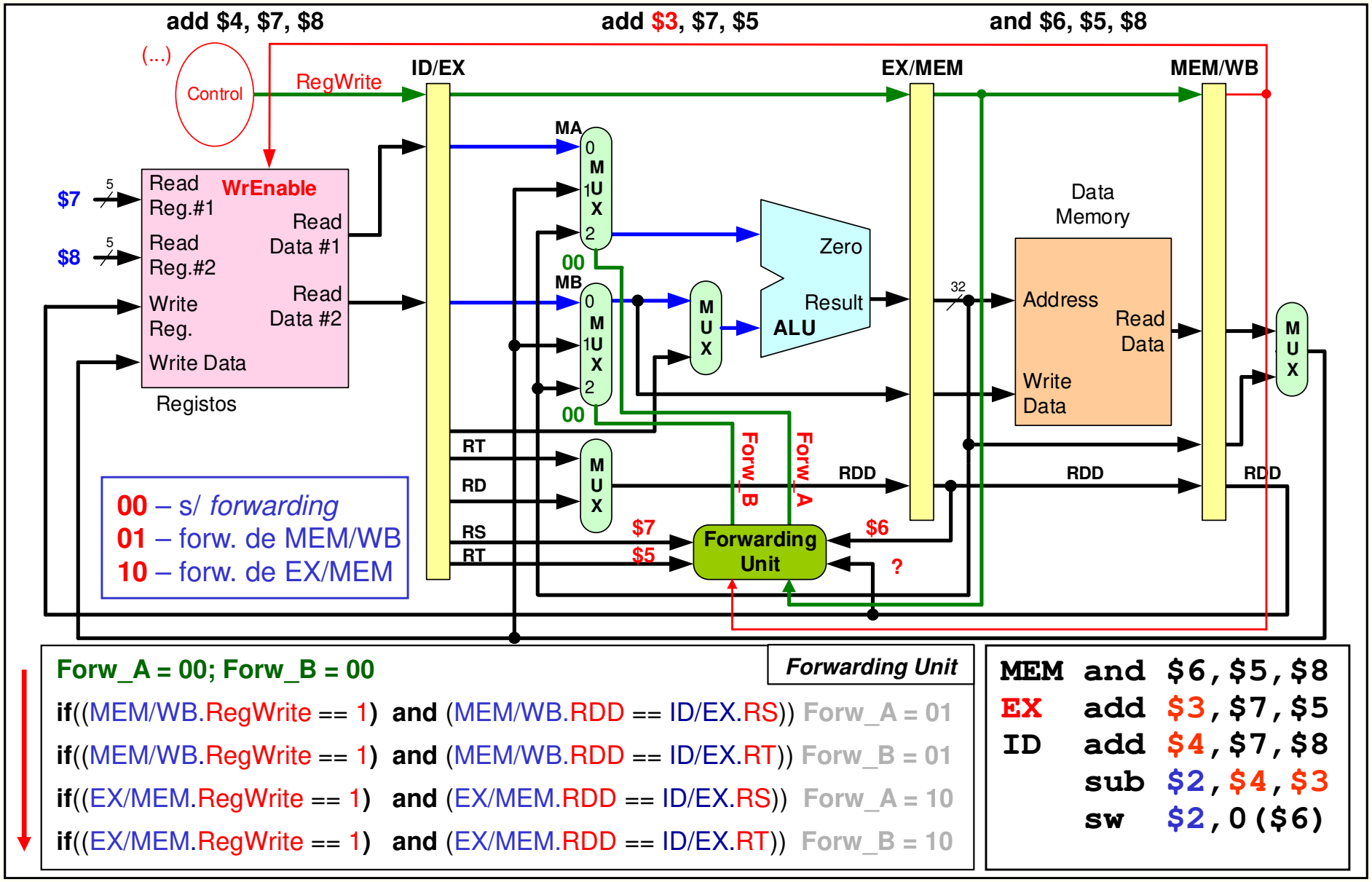
Exemplo de *forwarding*



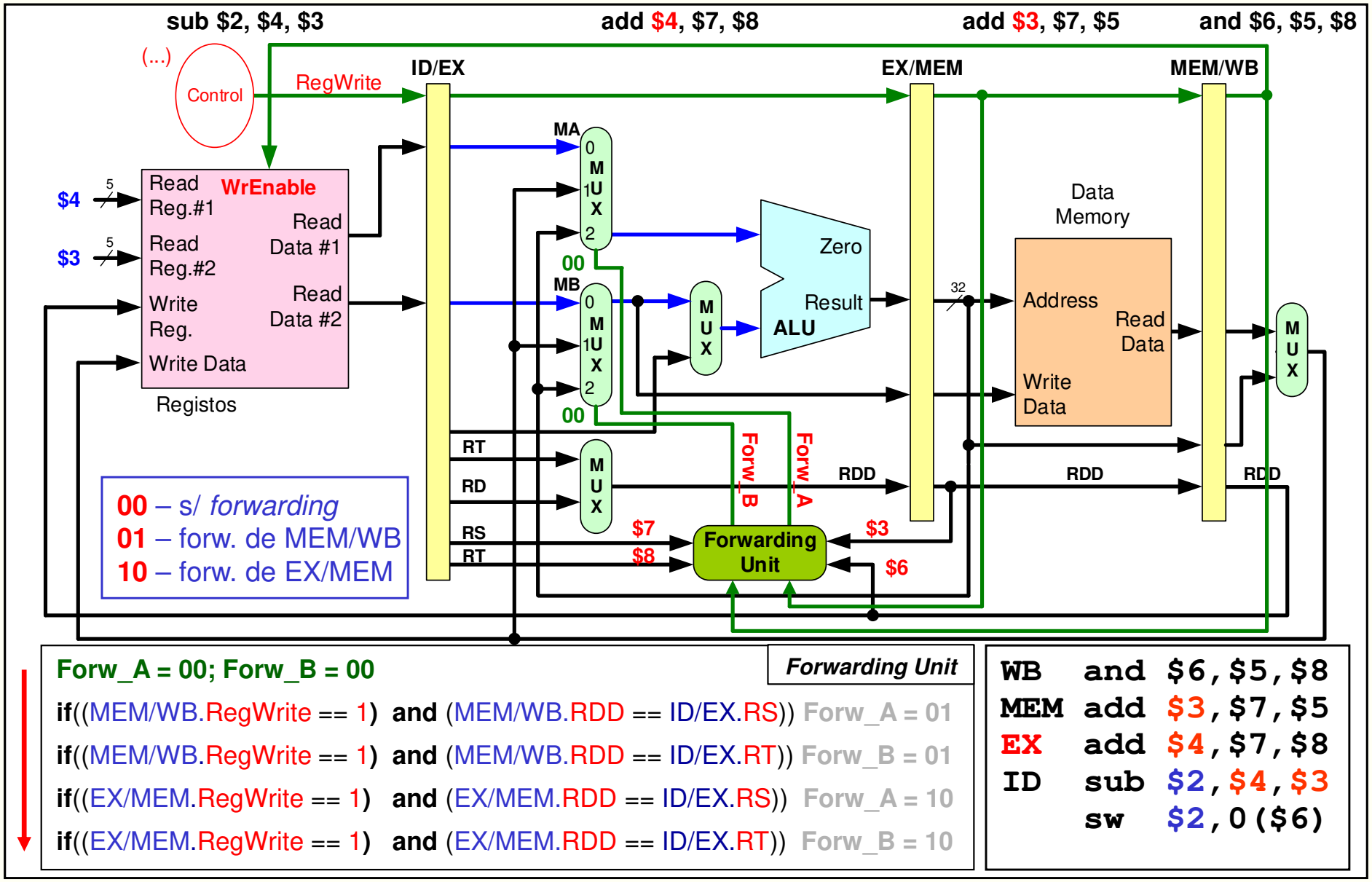
Exemplo de *forwarding*



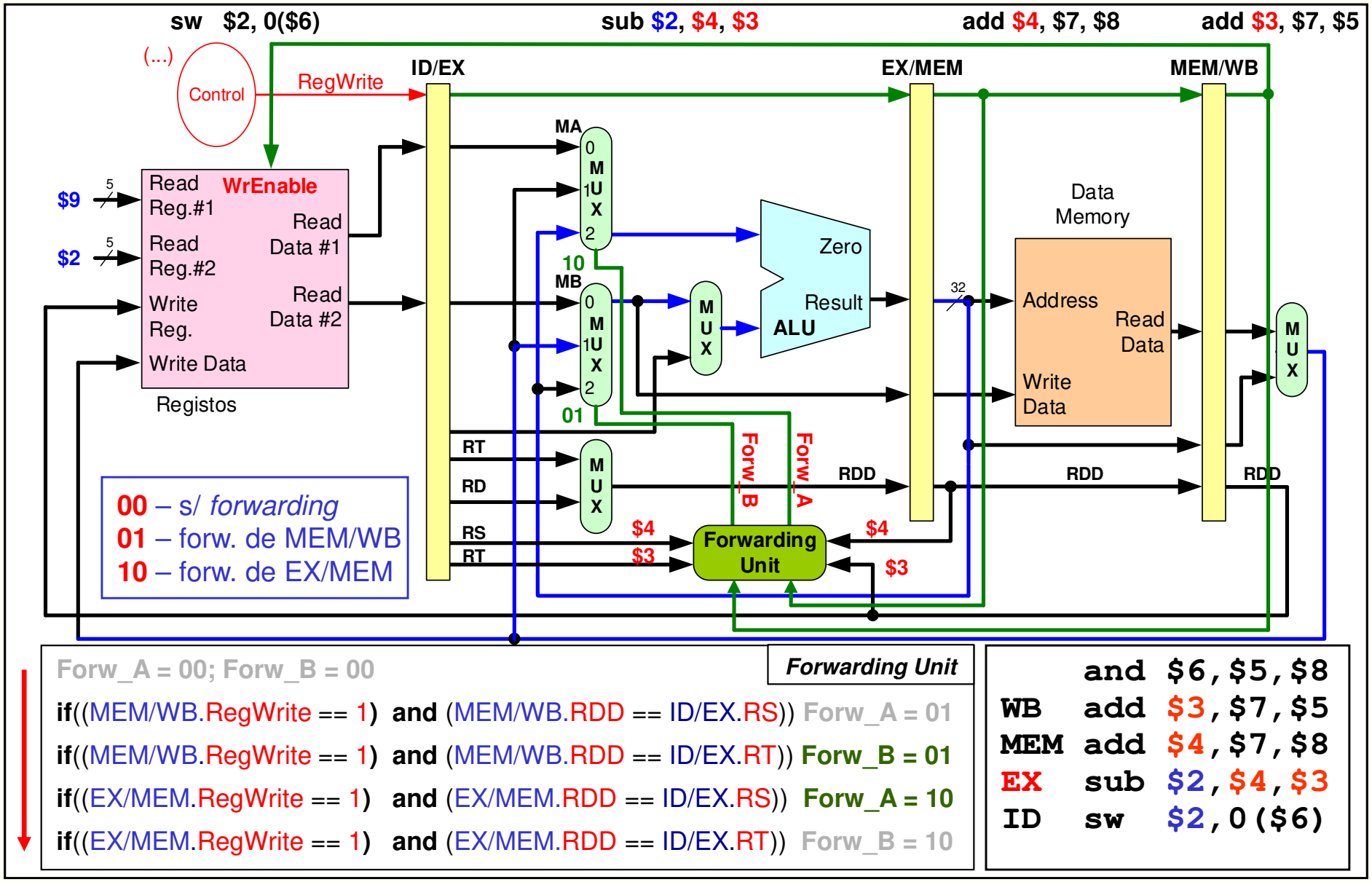
Exemplo de *forwarding*



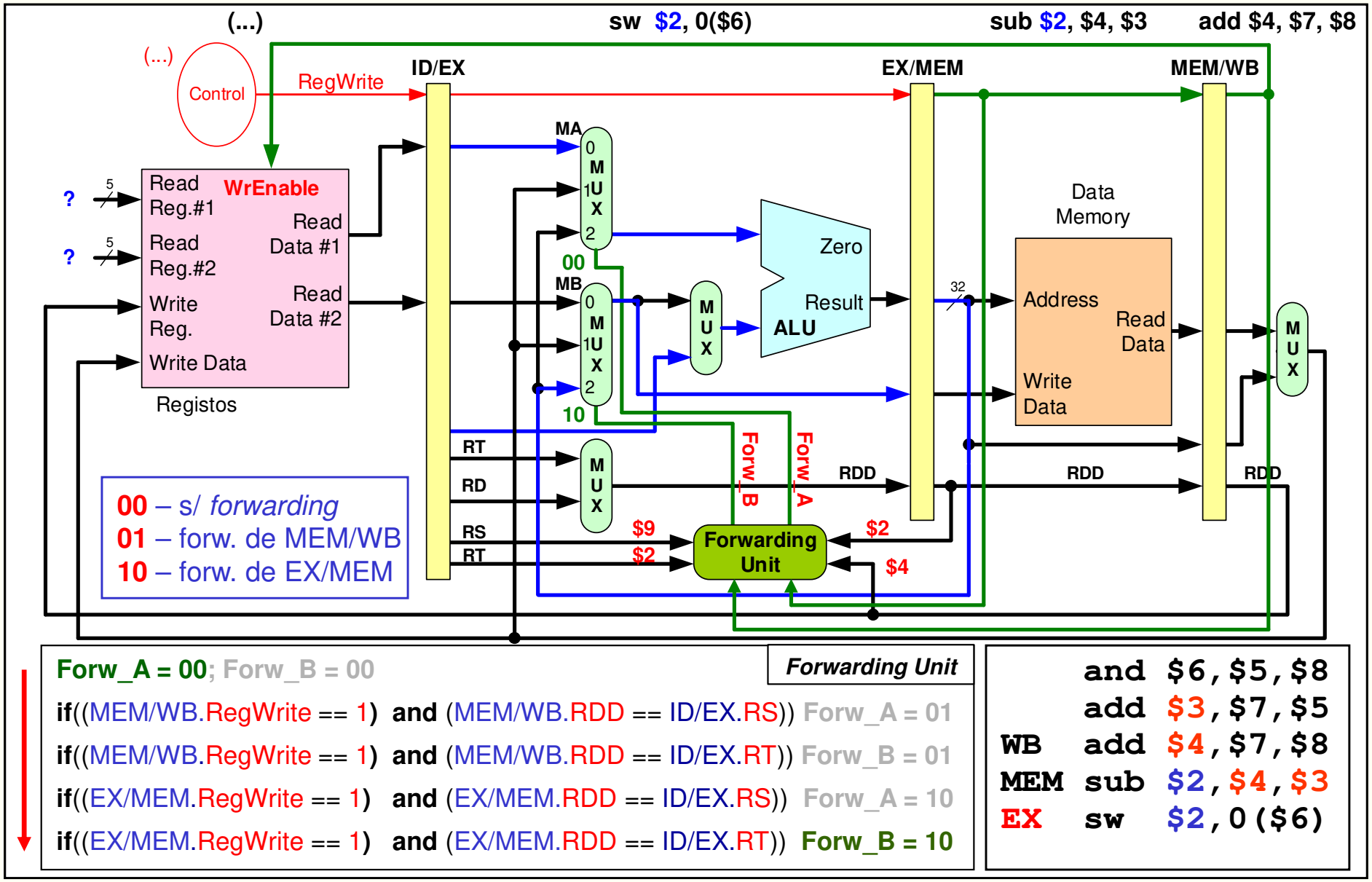
Exemplo de *forwarding*



Exemplo de *forwarding*



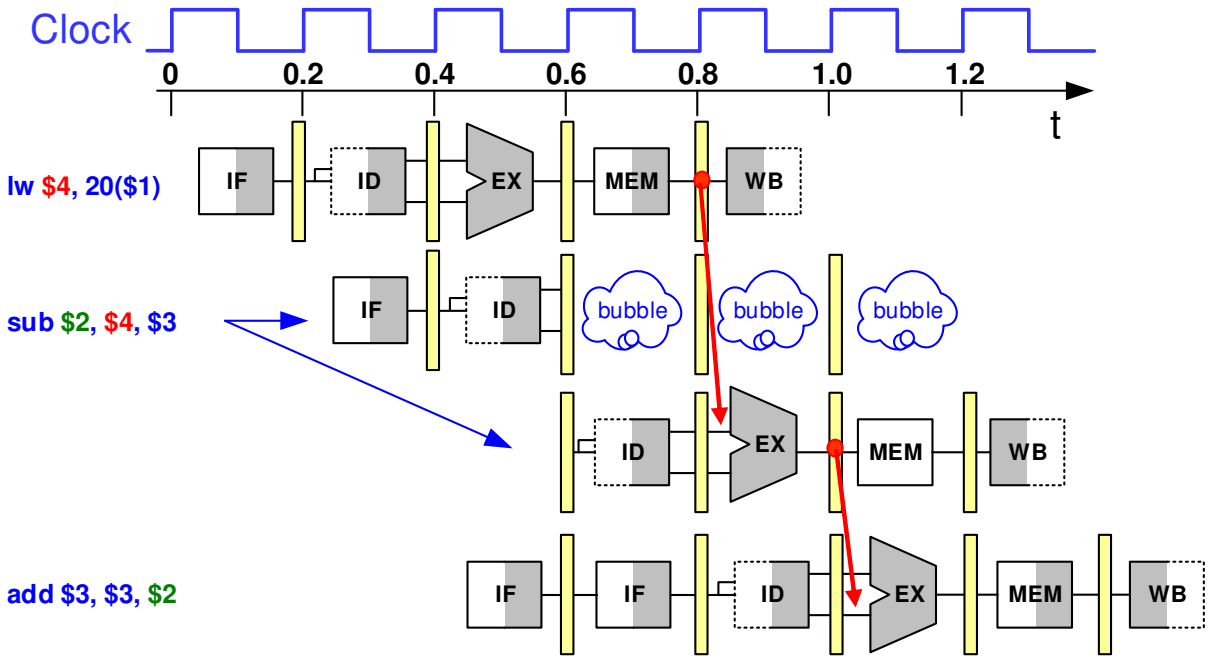
Exemplo de *forwarding*



Dependência que obriga a *stalling*

- Como já observado anteriormente, uma situação de dependência que obriga a *stalling* é a que resulta de uma instrução aritmética ou lógica executada a seguir e na dependência de uma instrução LW:

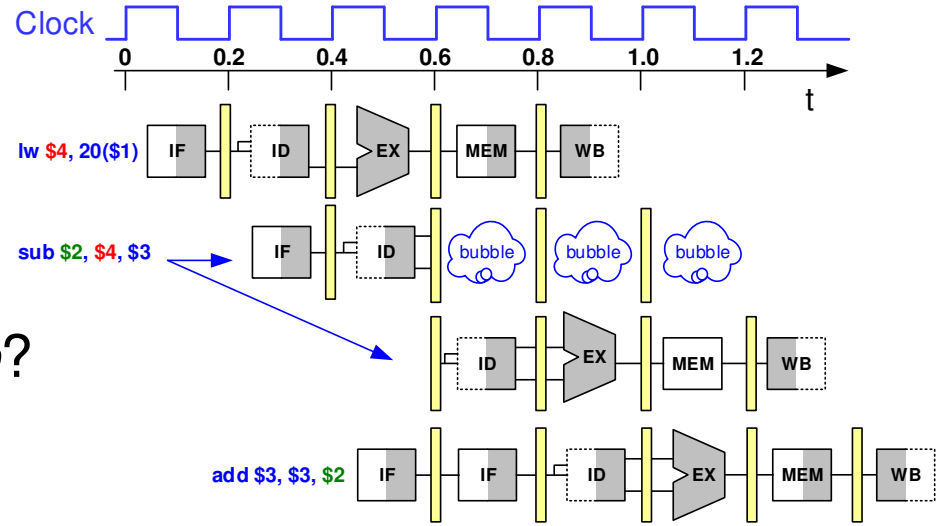
```
lw   $4, 20($1)   # valor disponível em WB
sub  $2, $4, $3   # Stall 1T, Forw. MEM/WB > EX
add  $3, $3, $2   # Forw. EX/MEM > EX
```



Dependência que obriga a *stalling*

- A situação de *stalling* tem que ser detetada quando a instrução tipo R está na sua fase ID (e o LW na fase EX). Como detetar?
- De forma simplificada:

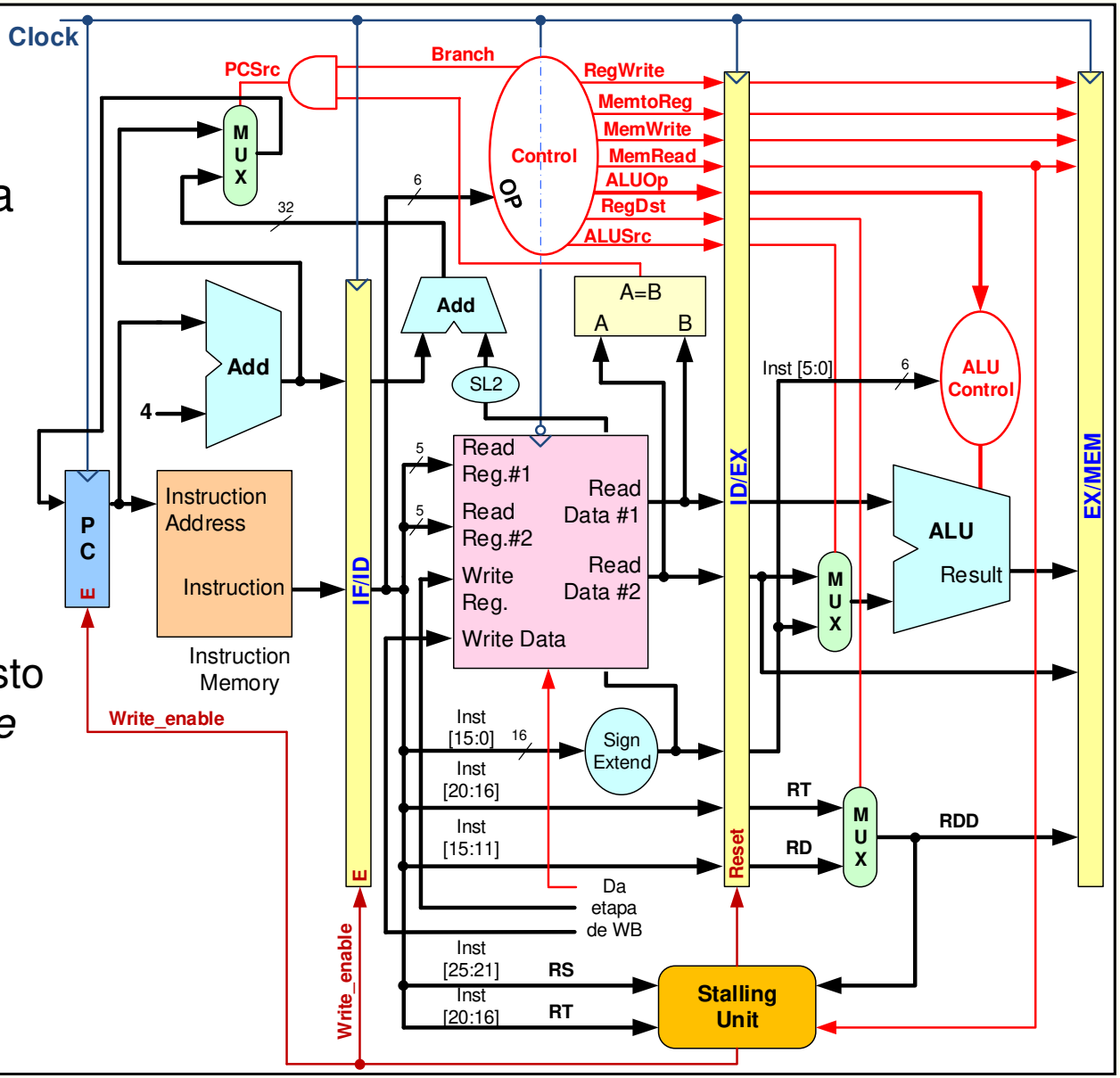
  (ID/EX.MemRead == 1) and (ID/EX.RDD == RS or ID/EX.RDD == RT)



- Como fazer o *stall* do *pipeline*?
  - **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 *stalling*

- **Inserir *bubble*** na etapa EX:
  - Ativar o *Reset* (síncrono) no registo ID/EX
- **Congelar, durante 1 ciclo de relógio, as etapas IF e ID**
  - Impedir a escrita no registo IF/ID - desativar o *Enable* do registo IF/ID
  - Impedir que seja feita a atualização do PC - desativar o *Enable* do registo PC



Unidade de controlo de *stalling* – VHDL

- Unidade de controlo de *stalling* simplificada, que contempla apenas a situação de dependência entre uma instrução LW e uma instrução tipo R
  - Instrução do tipo R: **(ALUOp == "10") and (RegWrite == 1)**

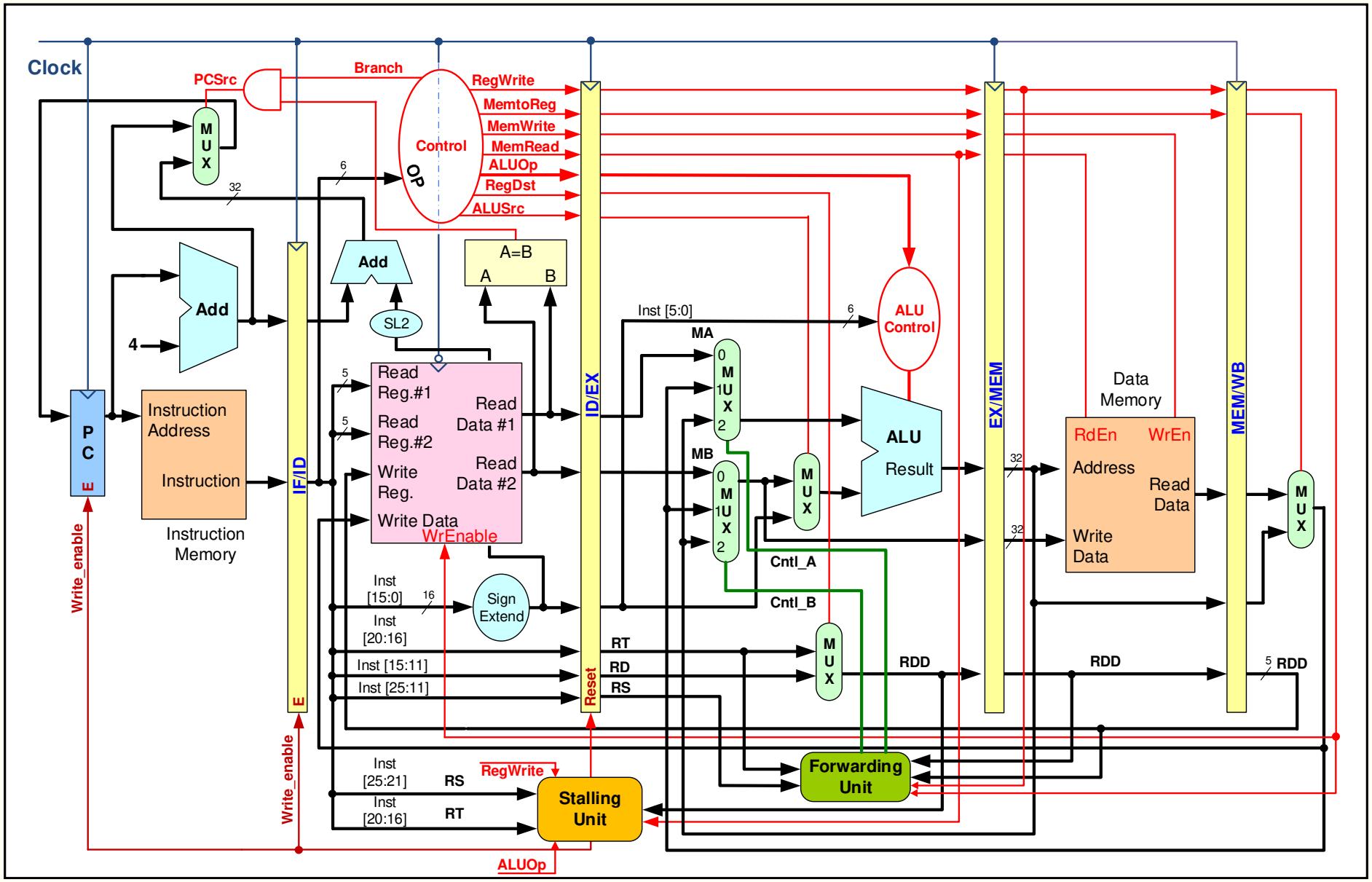
```
library ieee;
use ieee.std_logic_1164.all;

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);
        Ex_RDD        : in  std_logic_vector(4 downto 0);
        IdEx_MemRead  : in  std_logic;
        Reset_IdEx    : out std_logic;
        Enable_IfId   : out std_logic;
        Enable_PC     : out std_logic);
end StallingUnit;
```

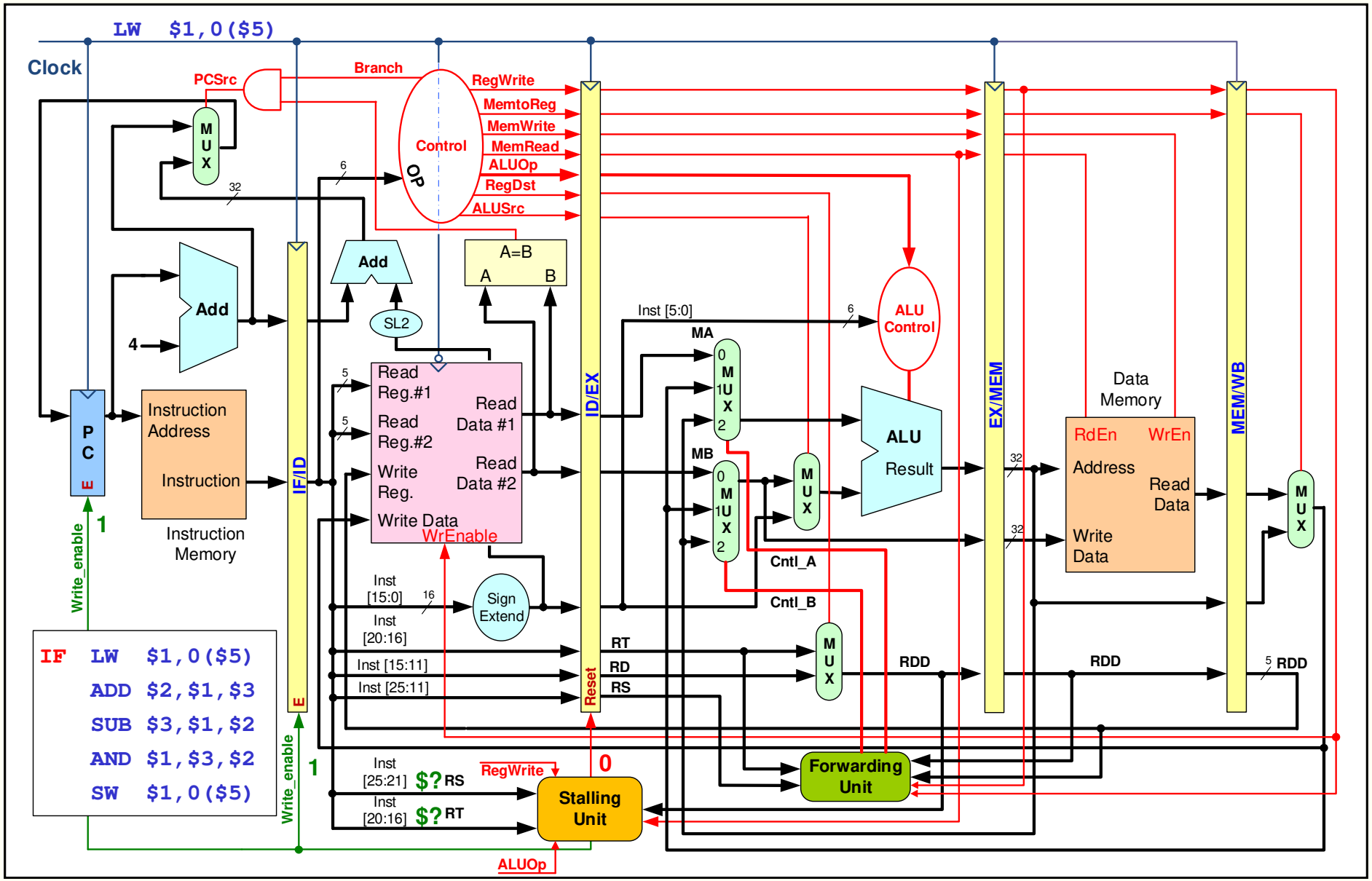
Unidade de controlo de *stalling* – VHDL

```
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 Ex_RDD /= "00000") then
        if(RegWrite = '1' and ALUOp = "10") then
           if(Ex_RDD = RS or Ex_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;
```

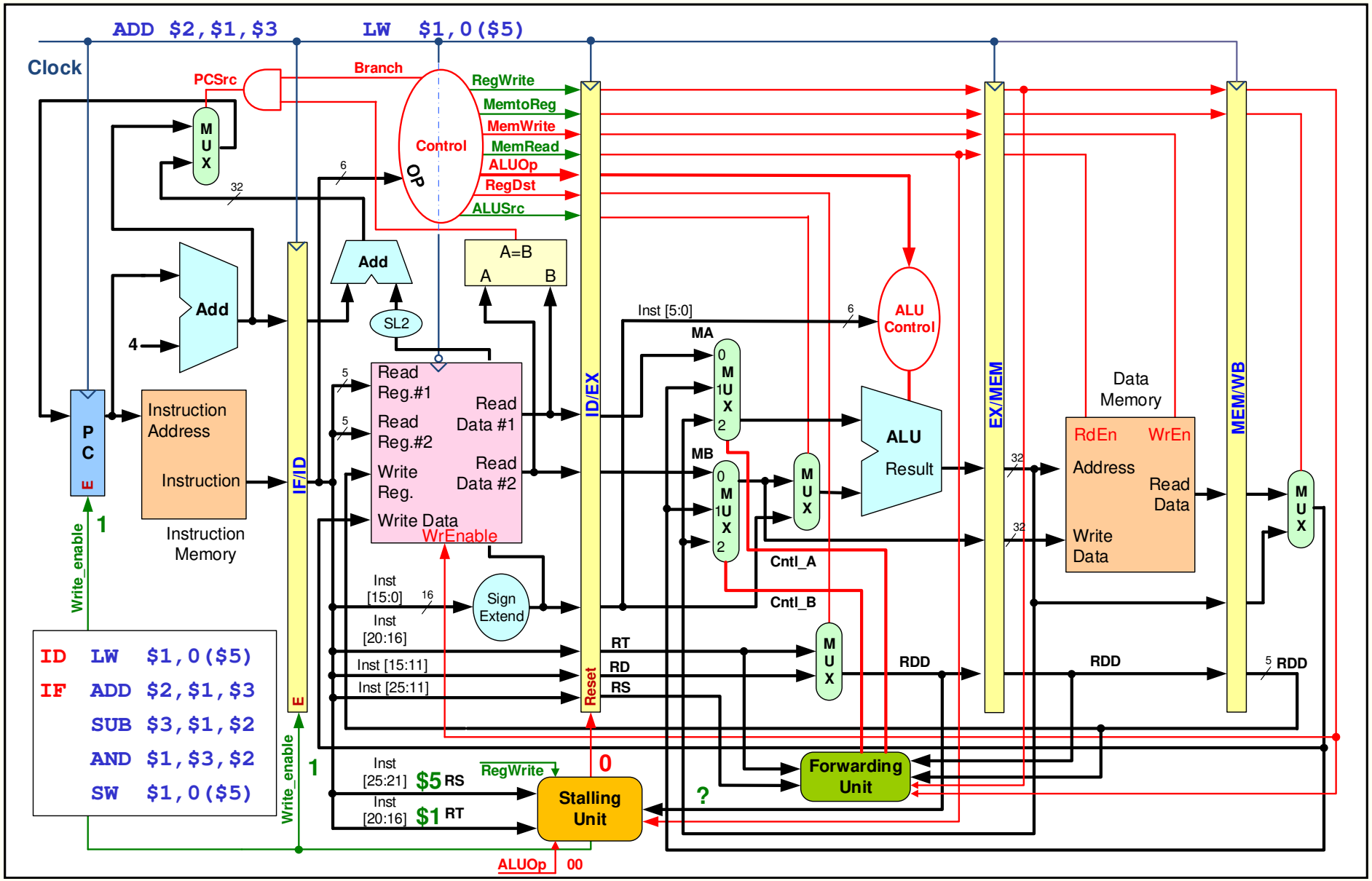
Datapath pipelining completo (com *forwarding* para EX, sem j)



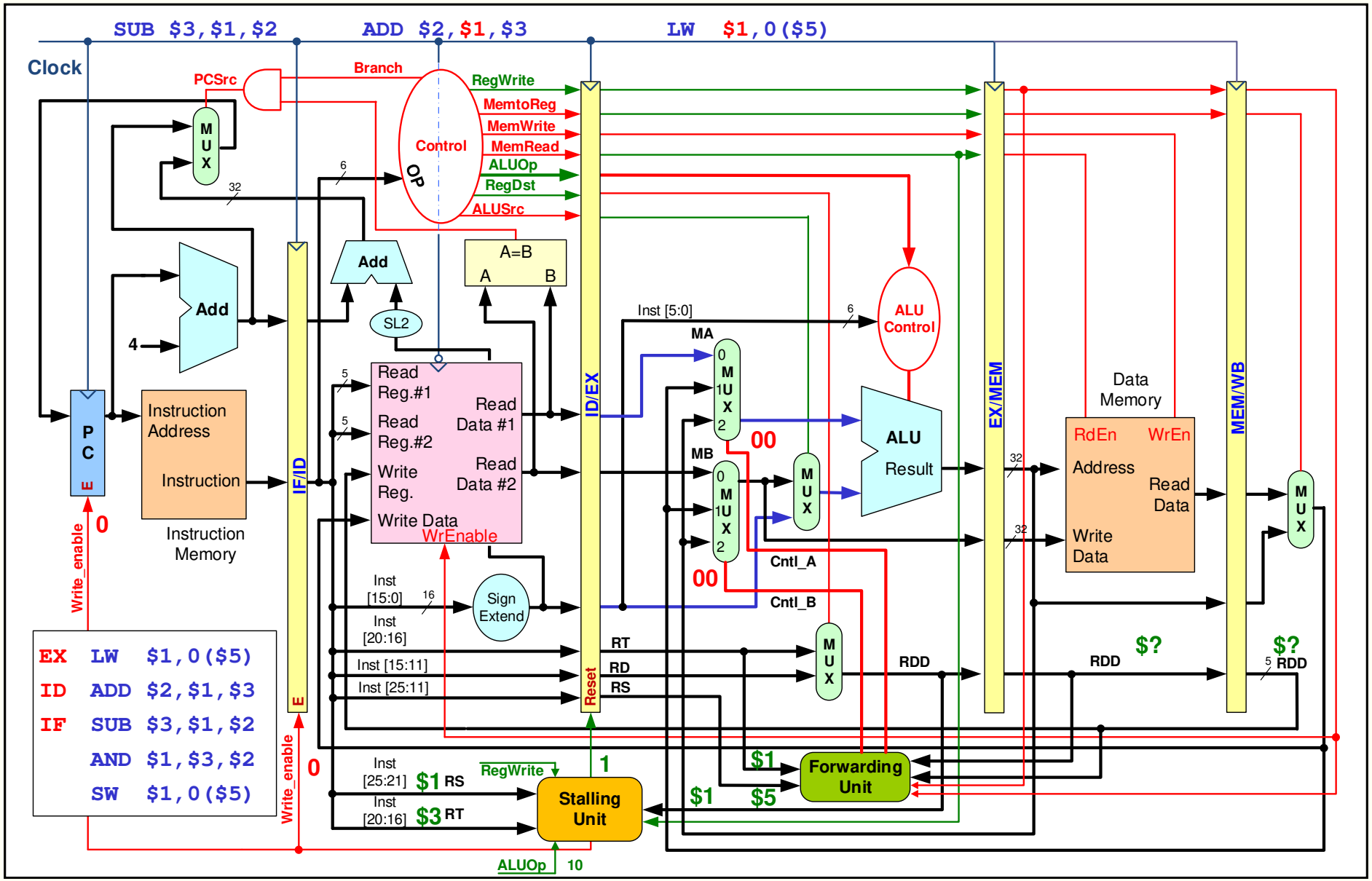
Datapath pipelining completo – exemplo de execução (1)



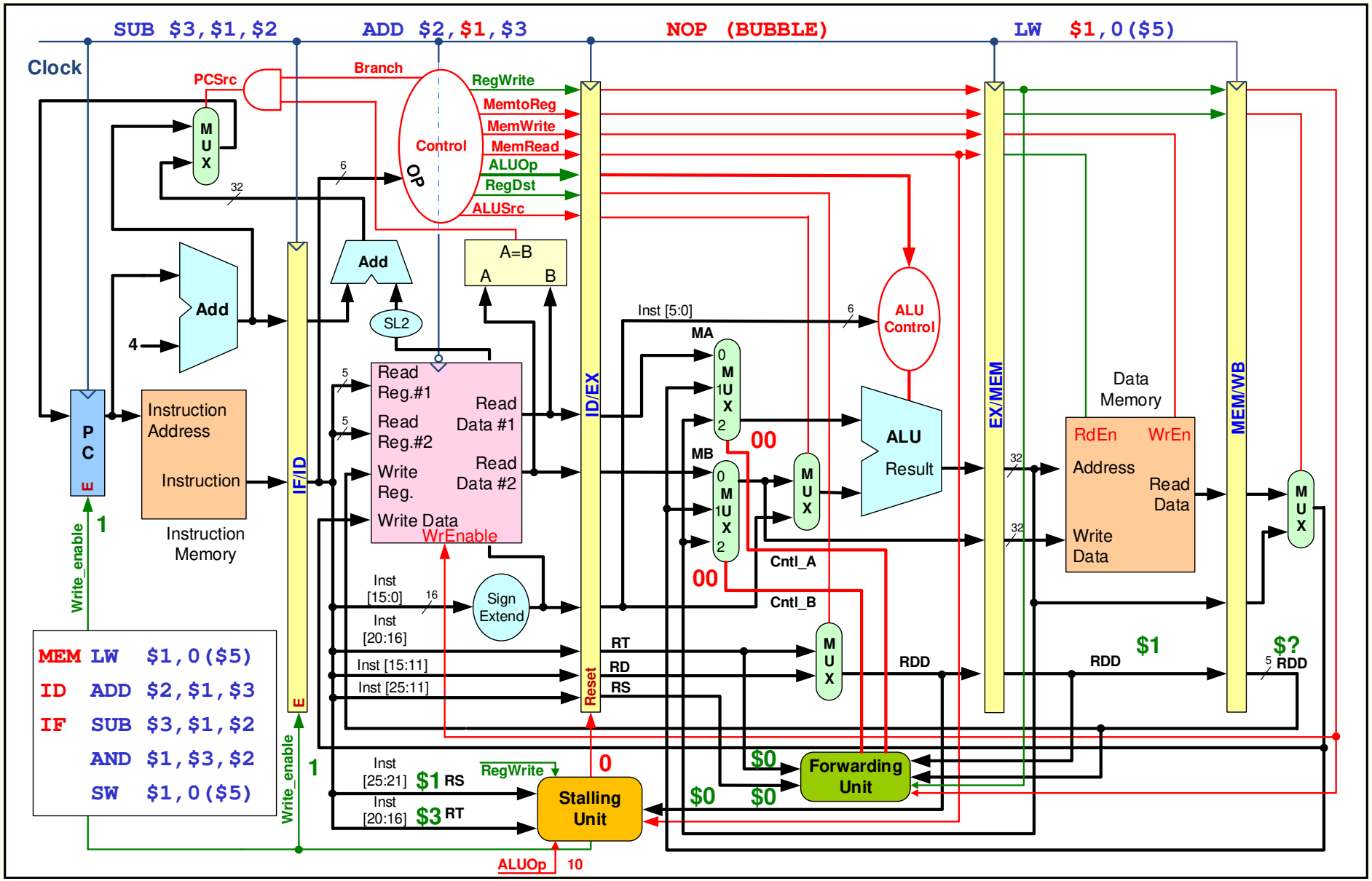
Exemplo de execução (2) (Normal Flow)



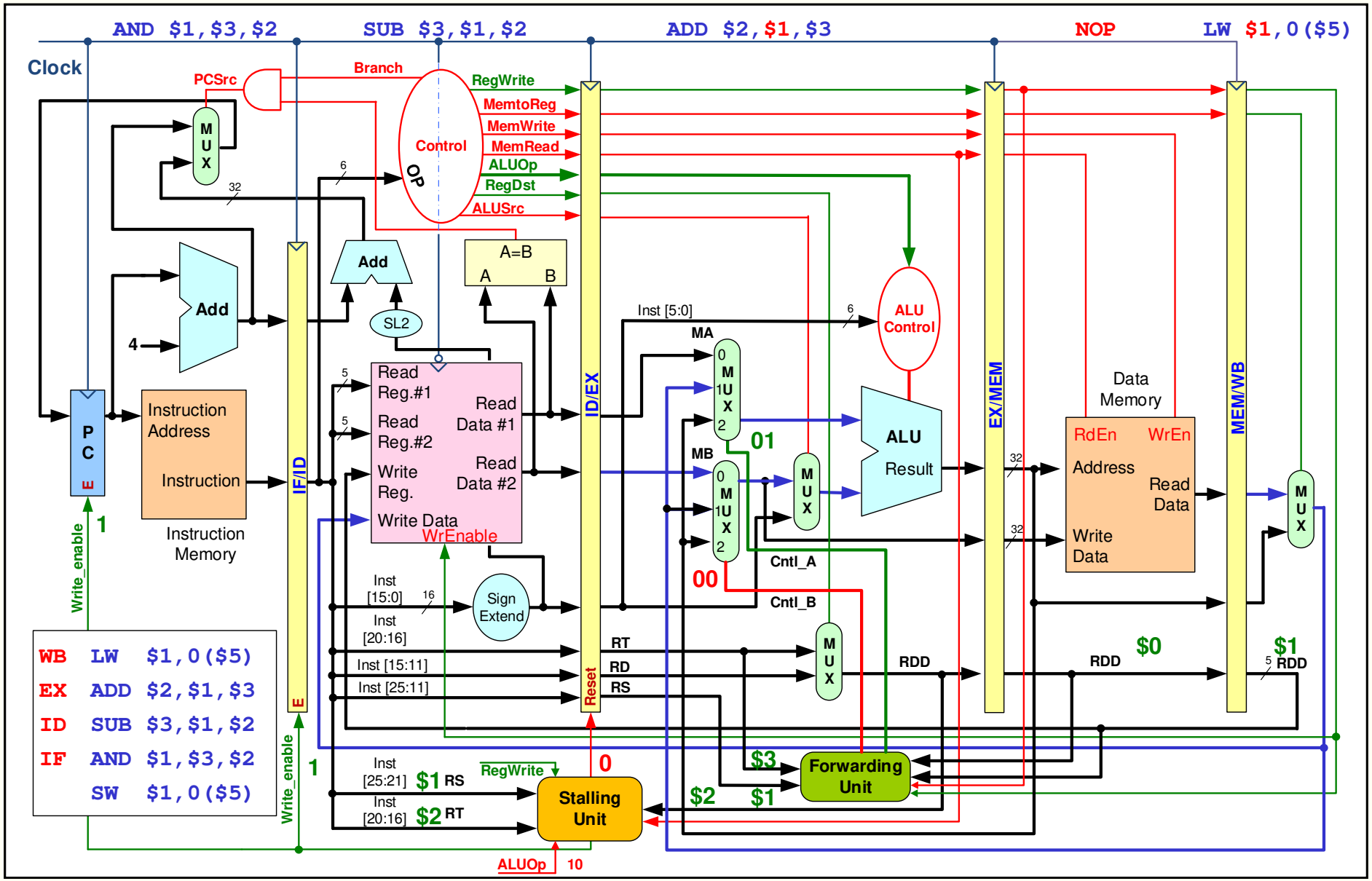
Exemplo de execução (3) (Normal Flow)



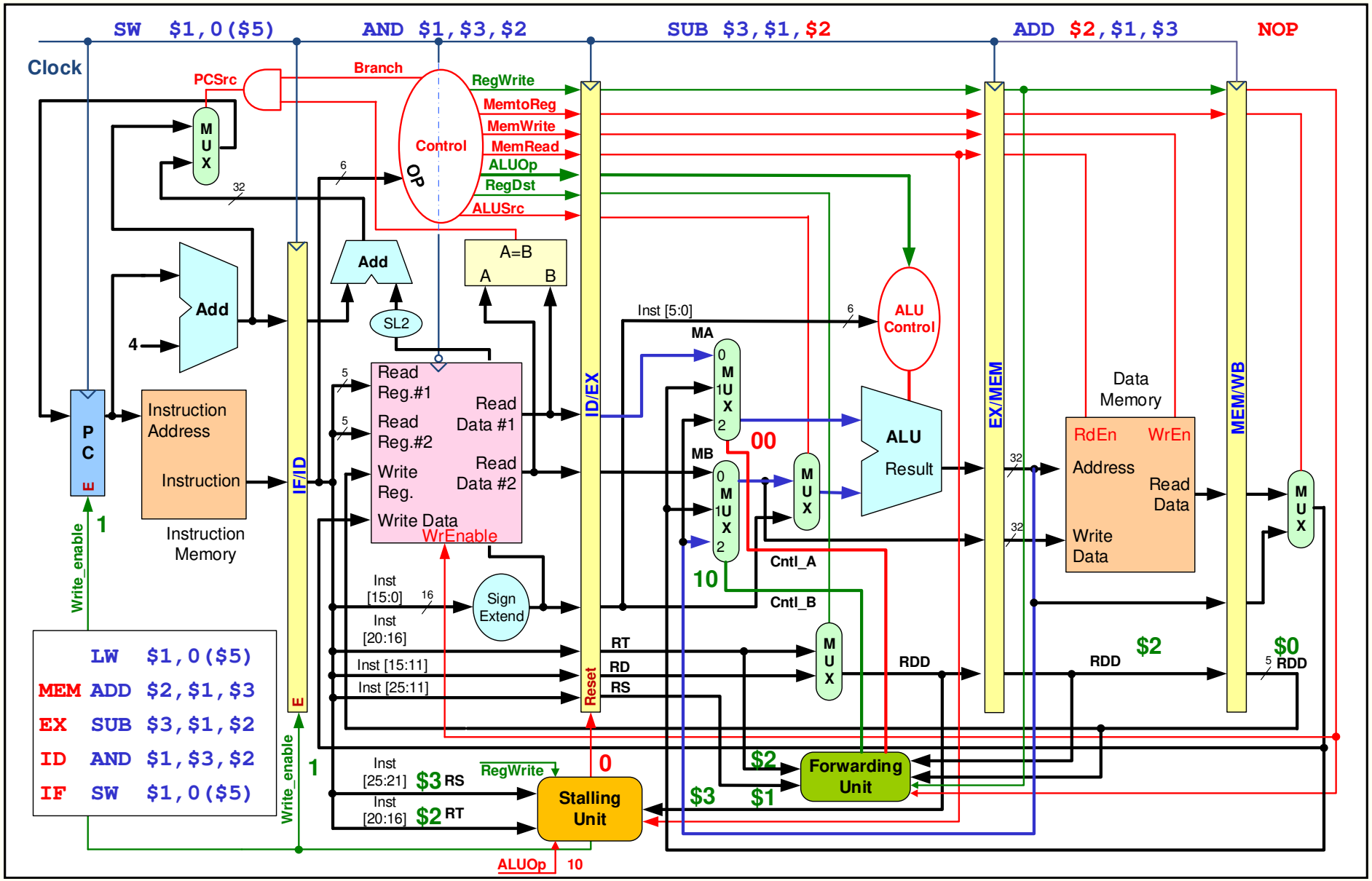
Exemplo de execução (4) (*STALL*)



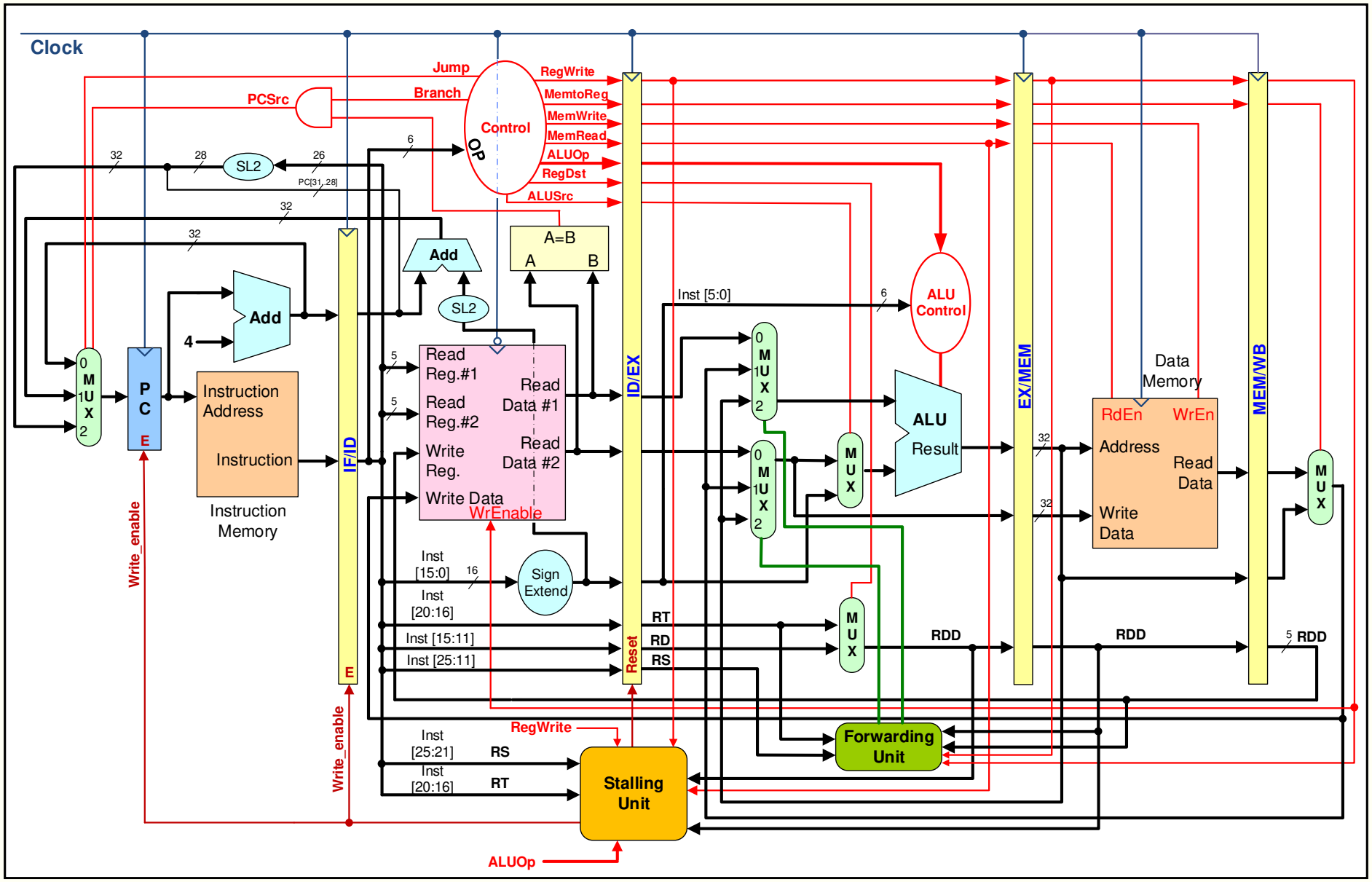
Exemplo de execução (5) (Fwd: **MEM/WB > EX, rs**)



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



Datapath pipelining completo, com *Jump*



Stalling seguido de *forwarding* para **EX**

- Para além da sequência descrita anteriormente, há outras situações que também são resolvidas com *forwarding* para EX, e que obrigam a *stall* do *pipeline*. Exemplos:

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

```
lw      $2,0($3)    #
lw      $4,8($2)    # Stall 1T, FW MEM/WB > EX (RS)
```

```
lw      $3,0($6)    #
addi    $4,$3,0x12  # Stall 1T, FW MEM/WB > EX (RS)
```

- Se a arquitetura apenas implementar *forwarding* para **EX**:

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

Forwarding para ID (EX/MEM para ID)

- Os *hazards* de dados que ocorrem em instruções de *branch* determinam que a arquitetura deva também implementar *forwarding* para ID
- Exemplos:

```
(MEM)     add   $1,$2,$3    #
(EX)      sub   $2,$4,$6    #
(ID)      beq   $1,$5,lab   # FW EX/MEM > ID (RS)
```

```
(MEM)     addi  $1,$3,0x25 #
(EX)      sub   $2,$1,$4,   # FW EX/MEM > EX (RS)
(ID)      beq   $5,$1,lab   # FW EX/MEM > ID (RT)
```

Stalling seguido de *forwarding* para **ID**

- Mesmo supondo que a arquitetura implementa *forwarding* para **ID** (**EX/MEM** para **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
```

Forwarding para **MEM** (**MEM/WB** para **MEM**)

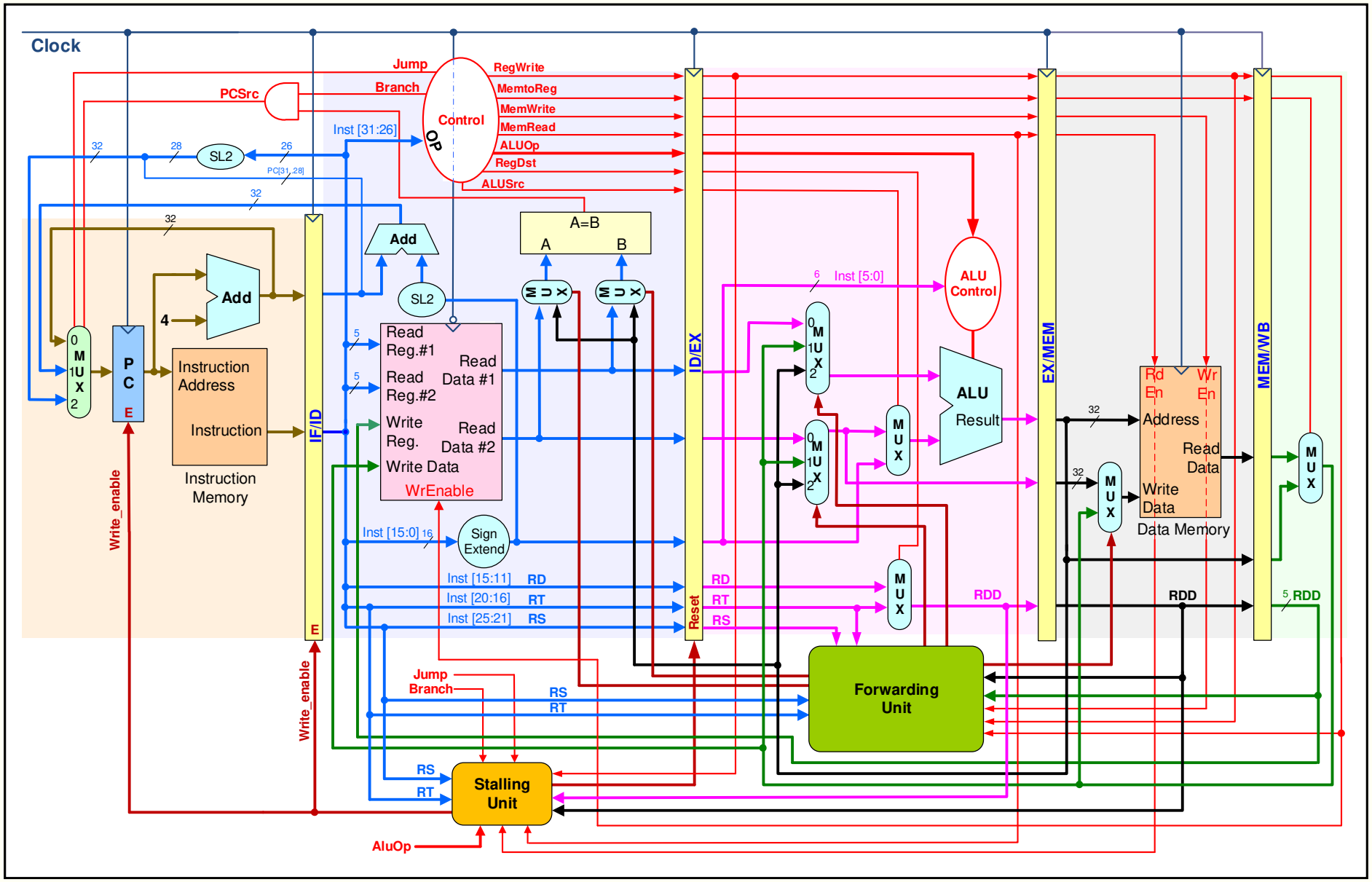
- Uma dependência originada por uma sequência do tipo:

```
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 (rt)**

- 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*

Datapath pipelining completo, com *forwarding* para MEM, EX e ID



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
```

Nr_Cycles = F + (Number_of_executed_instructions – 1)

= 5 + (6 – 1) = 10 T

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
0x0000000	0x10
0x0000004	0x20

- O ciclo é executado 4 vezes: $1∈[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

Nr_cycles = F + (Nr_instructions - 1) + Nr_Cycle_Stalls

= 5 + (28 + 4 - 1) + 32 = 68 T

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)   #
```

Nr_cycles = F + (Nr_instructions-1) + Nr_Cycle_Stalls

= 5 + (28 + 4 - 1) + 8 = 44 T (nr of *cycle stalls* = 4 * 2 = 8T)

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          #
```