Aulas 24, 25 & 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: *prediction*, *delayed branch*
  - *Hazards* de dados: *forwarding*
- *Datapath pipelined* para o MIPS com unidade de *forwarding*

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira, Tomás Oliveira e Silva

Pipelining

- *Pipelining* é uma técnica de implementação de arquitecturas do *set* de instruções, através da qual múltiplas instruções são executadas com algum grau de **sobreposição temporal**

- O objectivo é aproveitar, de forma o mais eficiente possível, os recursos disponibilizados pelo *datapath*, de forma a **maximizar a eficiência global do processador**

Pipelining

- 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 ☺

Para isso admite-se que o tratamento da roupa suja se desencadeia nas seguintes quatro fases:

1. Lavar uma carga de roupa na máquina respectiva
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 respectivo

Pipelining

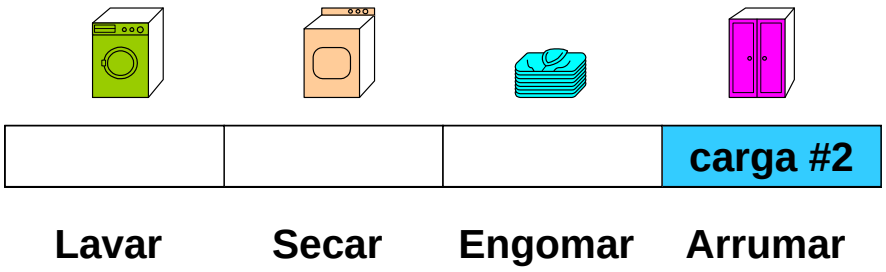
Numa versão *não pipelined*, o processamento de **n** cargas de roupa seria:

carga #1

carga #2

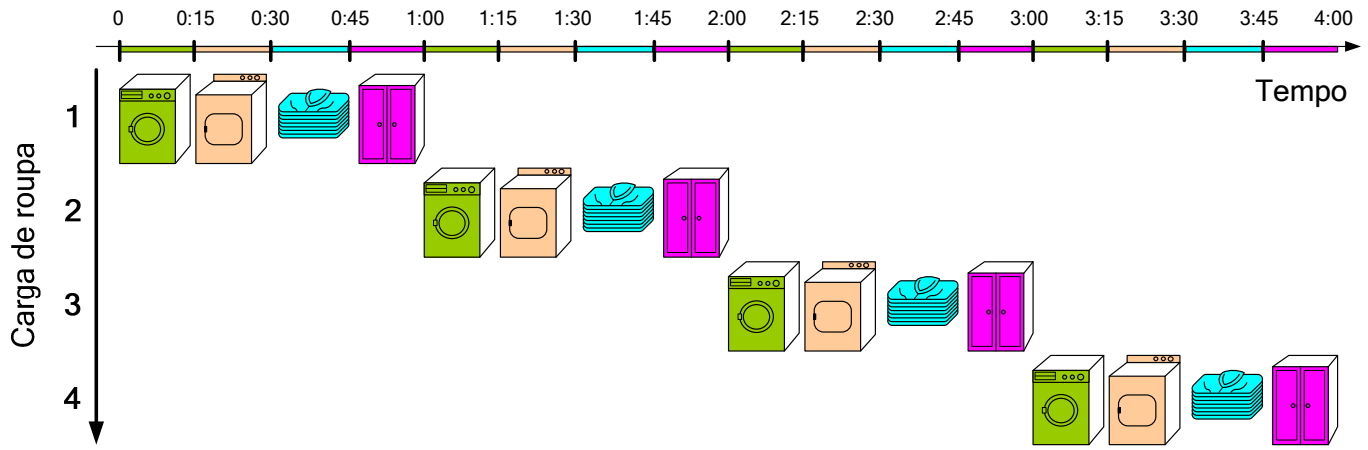
(…)

carga #n



Pipelining

Este processo pode então ser descrito temporalmente do seguinte modo:



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

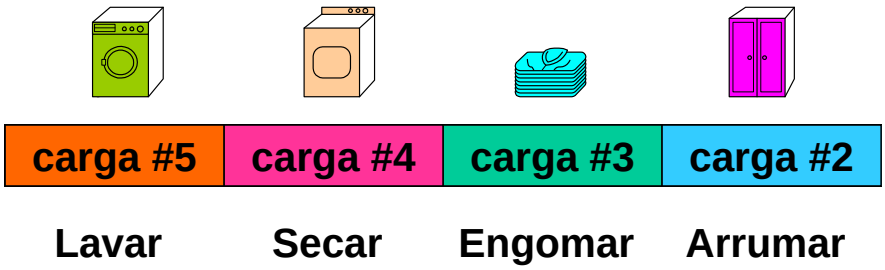
Pipelining

- 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

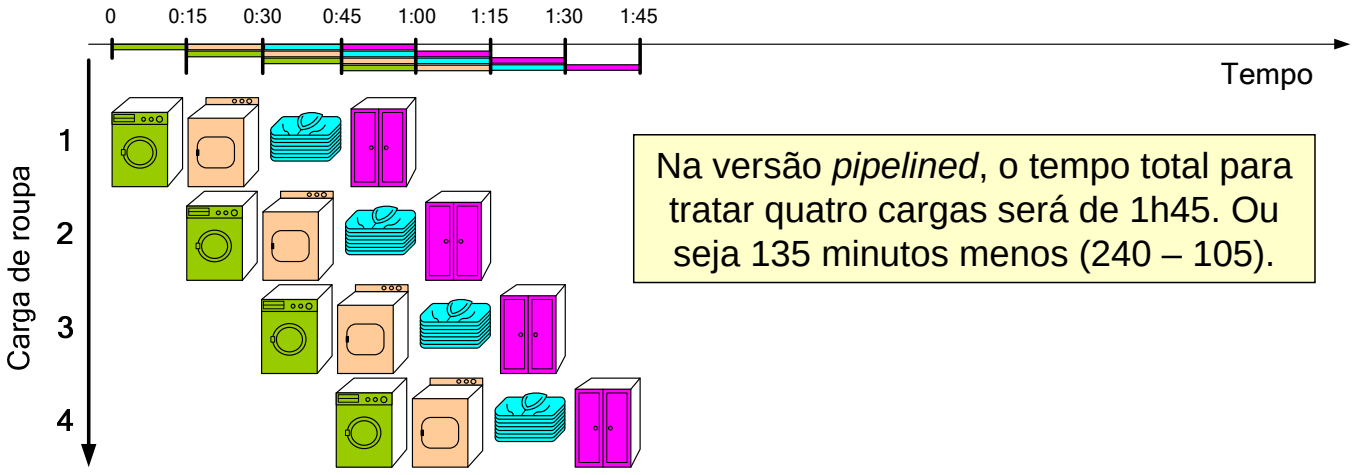
Na versão *pipelined*, o processamento das cargas de roupa seria (admitindo tempo nulo entre a comutação de tarefas):

carga #1
carga #2
carga #3
carga #4
carga #5



Pipelining

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

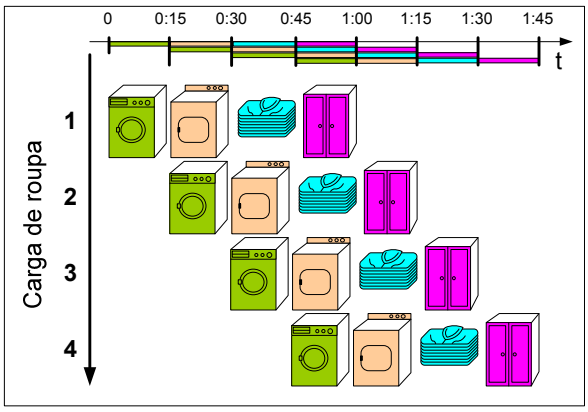


Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45. Ou seja 135 minutos menos (240 – 105).

Pipelining

- 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 muito 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*)

Pipelining



Questão: Qual o ganho de desempenho que se obtém com o sistema *pipelined* relativamente ao sistema normal?

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

Ganho 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: $$Ganho \approx \frac{N * F}{N} = F$$

Pipelining

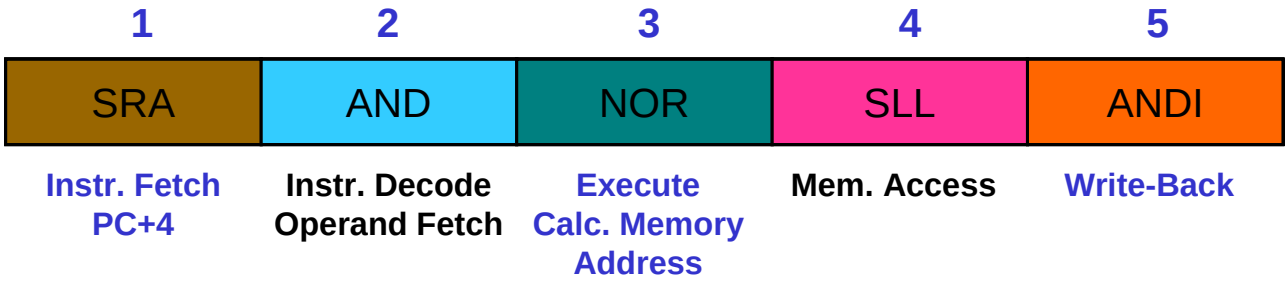
- No limite, para um número de cargas de roupa muito elevado, o **ganho de desempenho** (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e noutro modelo) é da ordem do **número de tarefas realizadas em paralelo** (isto é, igual ao número de fases do processo)
- Genericamente,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

Pipelining

- Os mesmos princípios que observámos para o caso do tratamento da roupa, podem igualmente ser aplicados aos processadores
- No caso do MIPS, como já sabemos, as instruções podem ser divididas 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

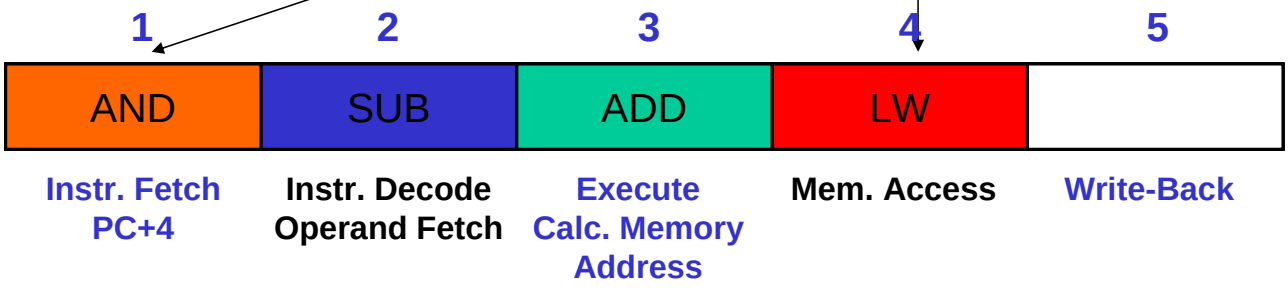
```
or   $t1, $s3, $t9
sub  $t2, $t9, $s3
add  $s0, $a1, $a2
andi $s3, $t5, 0x01
sll  $a1, $a2, 5
nor  $a2, $a2, $t1
and  $a3, $t2, $t1
sra  $s0, $t2, 1
(…)
```



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 efectuar, simultaneamente, um 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 (o **lw** ou o **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 de código.

1	2	3	4	5
Instr. Fetch PC+4	Instr. Decode Operand Fetch	Execute Calc. Memory Address	Mem. Access	Write-Back

Pipelining – problemas (Exemplo 3)

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

A instrução de subtracção não pode avançar para o estágio seguinte uma vez que o seu operando **$s0** ainda não foi calculado e armazenado no registo destino pela instrução anterior.

1	2	3	4	5
Instr. Fetch PC+4	Instr. Decode Operand Fetch	Execute Calc. Memory Address	Mem. Access	Write-Back

Pipelining

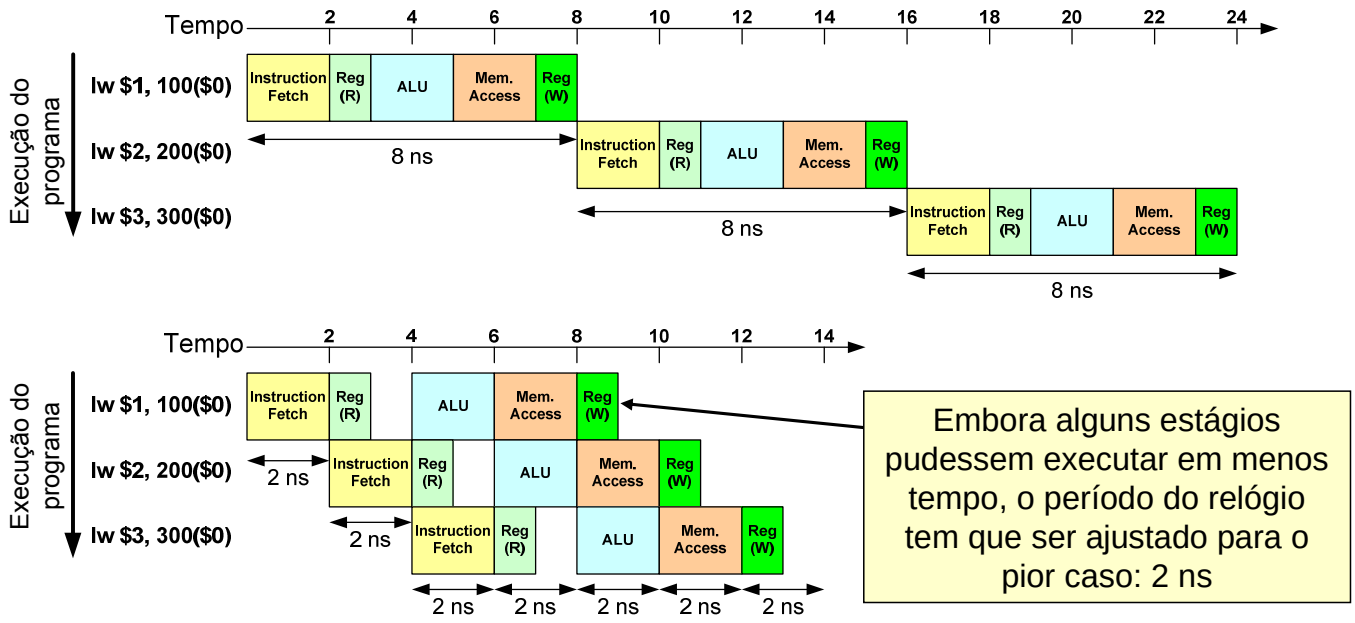
- Para tornar a discussão mais concreta, vamos construir um *datapath* que implemente um *pipeline,* que suporte as instruções que já considerámos anteriormente, isto é:
  - acesso à memória: load *word* (**lw**) e *store word* (**sw**)
  - instruções **tipo R: *add***, ***sub***, ***and***, ***or*** e ***slt***
  - Instruções imediatas: ***addi*** e ***slti***
  - *branch if equal* (**beq**)
- Comparemos os tempos necessários à execução destas instruções num *datapath single cycle* e num *datapath pipelined*
- Para o efeito, admitamos que o tempo necessário à execução de cada uma das fases da instrução é o indicado na tabela seguinte:

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Memory Access	Register Write	Tempo total
Load word (lw)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (sw)	2 ns	1 ns	2 ns	2 ns		7 ns
R-Format (add, sub, and, or, slt)	2 ns	1 ns	2 ns		1 ns	6 ns
Branch (beq)	2 ns	1 ns	2 ns			5 ns
addi, slti	2 ns	1 ns	2 ns		1 ns	6 ns

Pipelining

- De acordo com a tabela fornecida, e para a solução *single cycle*, teremos que ajustar o período do relógio ao tempo necessário para executar a instrução mais lenta (lw)
- Ou seja, na solução *single cycle* todas as instruções, independentemente do tempo mínimo que poderiam durar, serão executadas num tempo de 8ns
- Para verificarmos como comparar o tempo de execução dum trecho de código por cada uma das soluções (*pipelined* e não *pipelined*), observemos o exemplo do slide seguinte

Pipelining



Embora alguns estágios pudessem executar em menos tempo, o período do relógio tem que ser ajustado para o pior caso: 2 ns

Pipelining

O *instruction set* do MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) foi concebido desde o início para uma implementação em *pipeline*. Os aspectos fundamentais a considerar são:

- **Instruções de comprimento fixo**. *Instruction fetch* e *instruction decode* podem ser feitos em estágios sucessivos uma vez que a unidade de controlo não tem que se preocupar com a dimensão da instrução descodificada
- **Poucos formatos de instrução**, com a referência aos registos a ler sempre no mesmo campo. Isto 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 assim 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. Desta forma qualquer operação de leitura/escrita da memória pode ser feita num único estágio

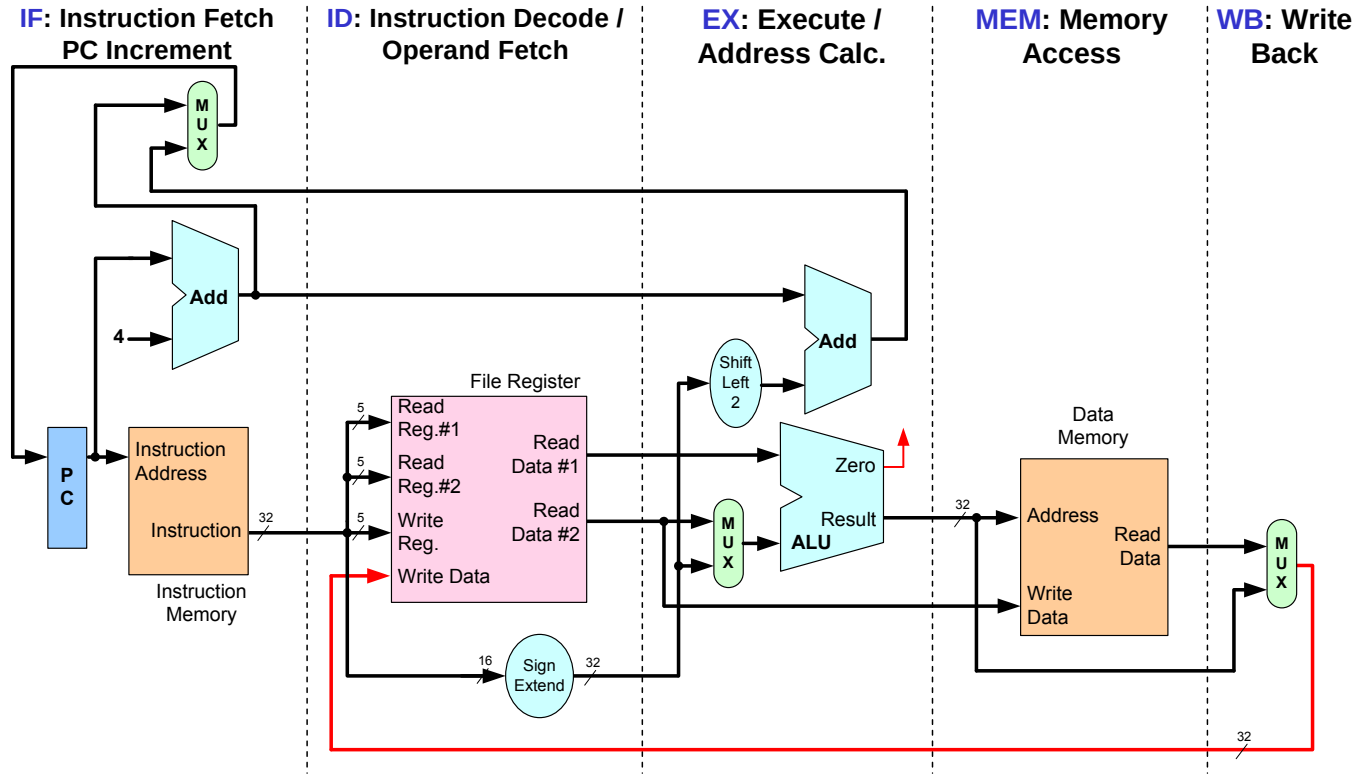
Pipelining

- Observemos então o aspecto de uma solução *pipelined* para o MIPS, partindo do modelo do *datapath single-cycle*
- A organização tenta retratar, como já referido, as cinco fases sequenciais em que são decomponíveis as instruções:

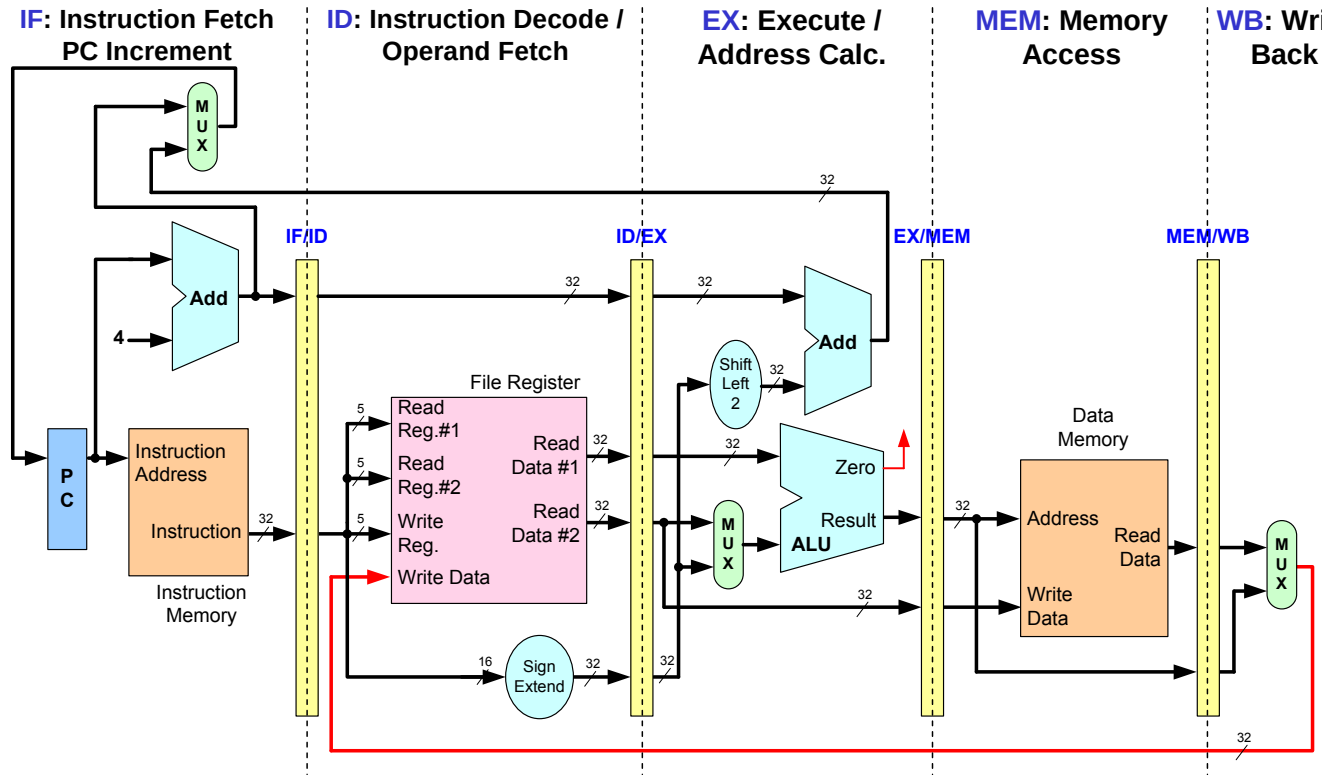
1. (**IF**) - *Instruction fetch* (ler a instrução da memória), incremento do PC
2. (**ID**)- *Operand fetch* (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
3. (**EX**) - Executar a operação ou calcular um 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 respectiva unidade de controlo

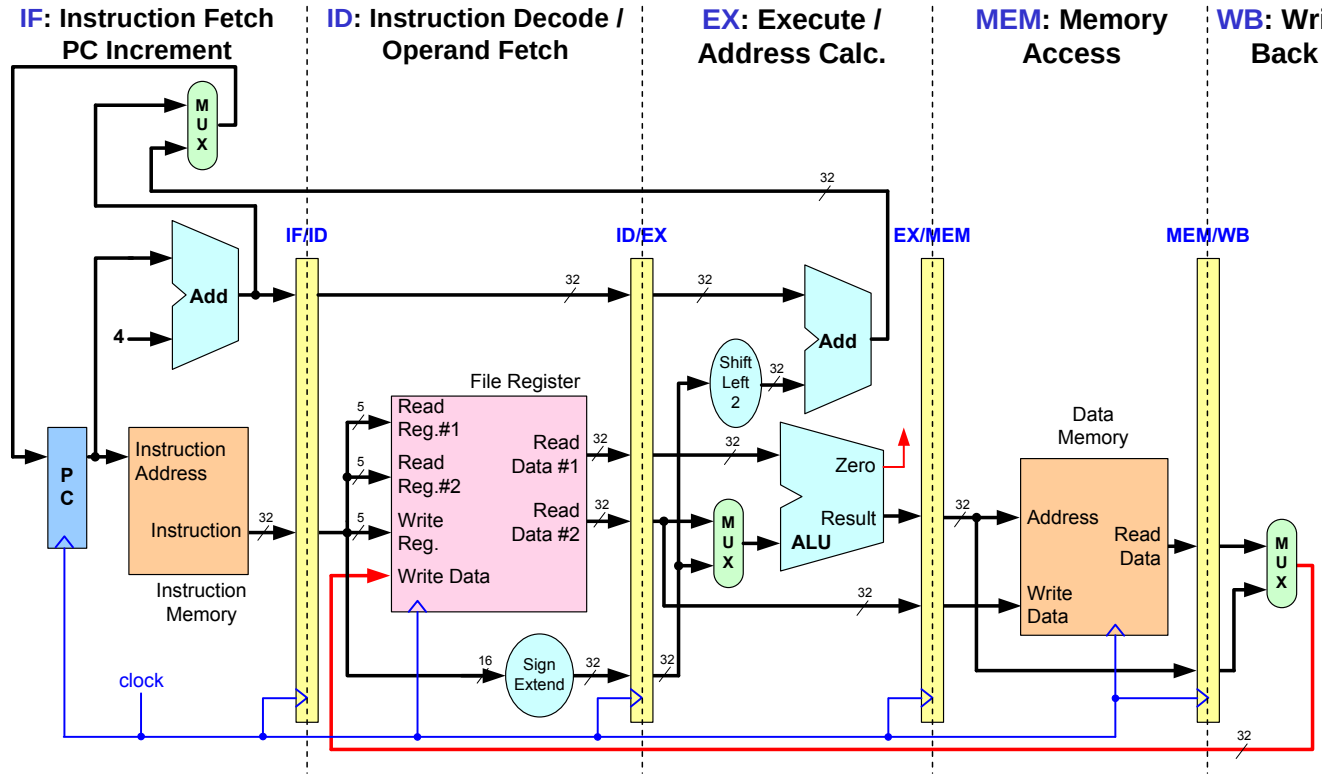
Pipelining: divisão em fases de execução



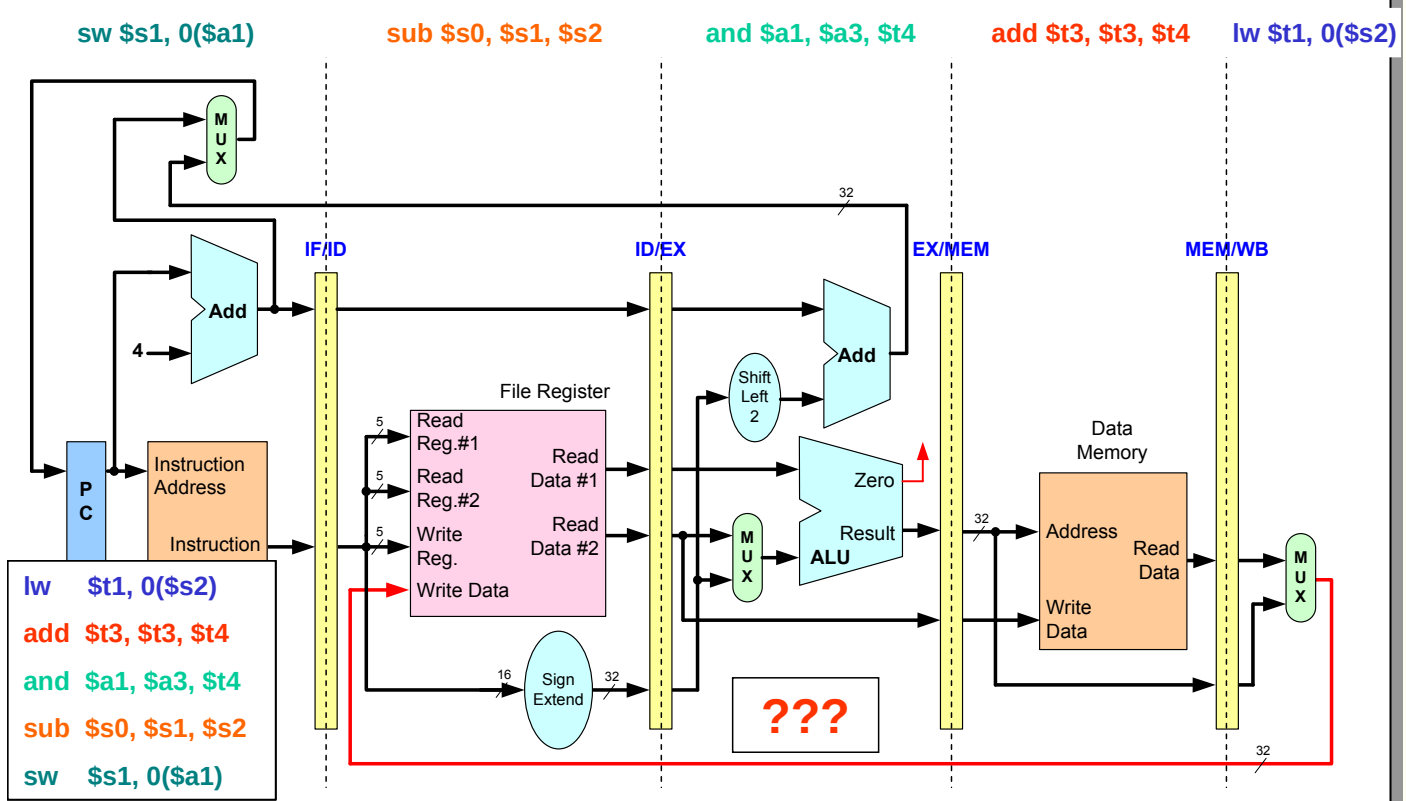
Pipelining: divisão em fases de execução



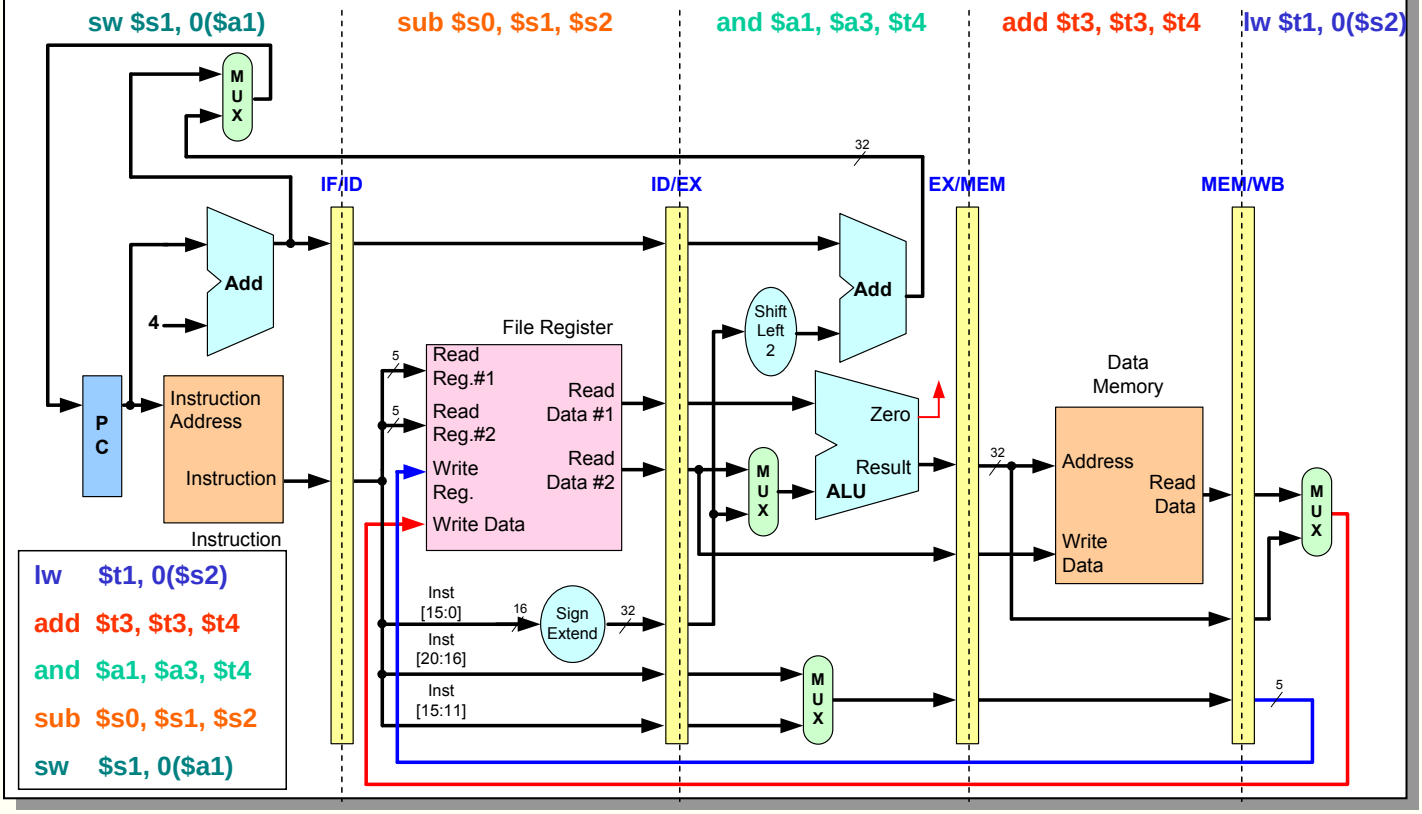
Pipelining: divisão em fases de execução



Pipelining: Execução de instruções



Pipelining: Execução de instruções



Pipelining Hazards

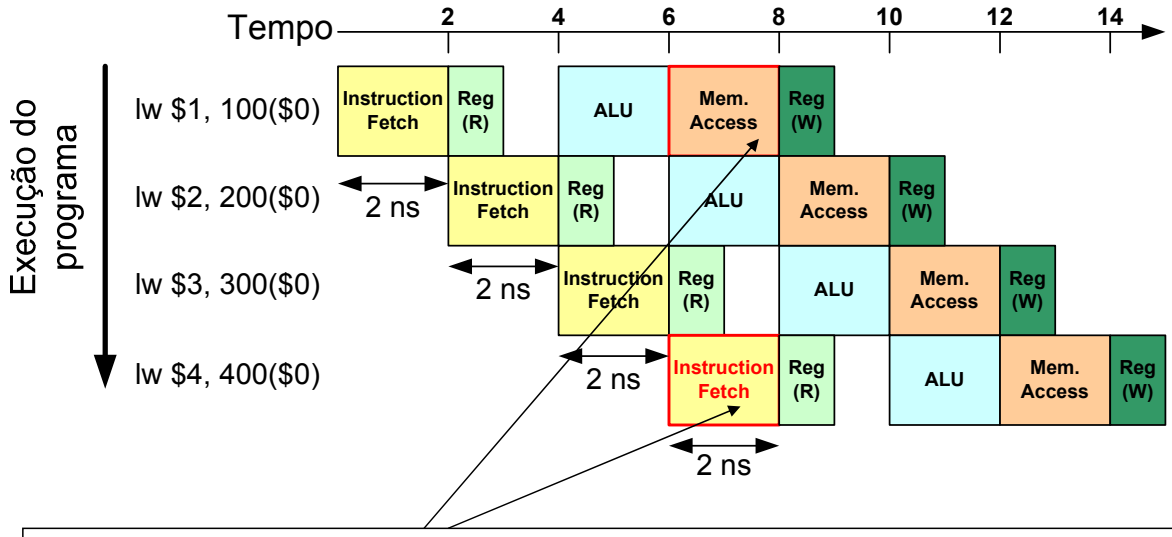
- Existe um conjunto de situações particulares que podem determinar que a próxima instrução não possa progredir no *pipeline* no próximo ciclo de relógio
- Estas situações são designadas genericamente por ***hazards***, e podem ser agrupadas em três classes distintas:
  - *Hazards* **estruturais**
  - *Hazards* de **controlo**
  - *Hazards* de **dados**
- Observemos, para cada tipo de *hazard*, as origens e as consequências, mapeando depois esses aspectos ao nível da arquitectura *pipelined* do MIPS

Pipelining: *Hazards* estruturais

- Um *hazard* estrutural ocorre quando o *hardware* não consegue suportar a execução simultânea de duas operações com base nos recursos de que dispõe
- No caso do MIPS, e como já vimos, o *set* de instruções foi pensado para suportar directamente uma solução *pipelined* minimizando o potencial de ocorrência de *hazards* estruturais
- Isso não evita, contudo, a necessidade de duplicar alguns recursos
- É o caso da memória que, tal como já acontecia na solução *single-cycle*, precisa de ser desdobrada em memória de programa e memória de dados

No exemplo da tarefa de tratar da roupa, um hazard estrutural resultaria, por exemplo, da utilização de uma máquina mista de lavar e secar a roupa. Nesse caso, não seria possível lavar e secar em paralelo duas cargas de roupa, muito embora continuasse a ser possível passar a ferro e arrumar. O início de uma nova carga de roupa estaria assim condicionado à libertação do recurso lavar/secar.

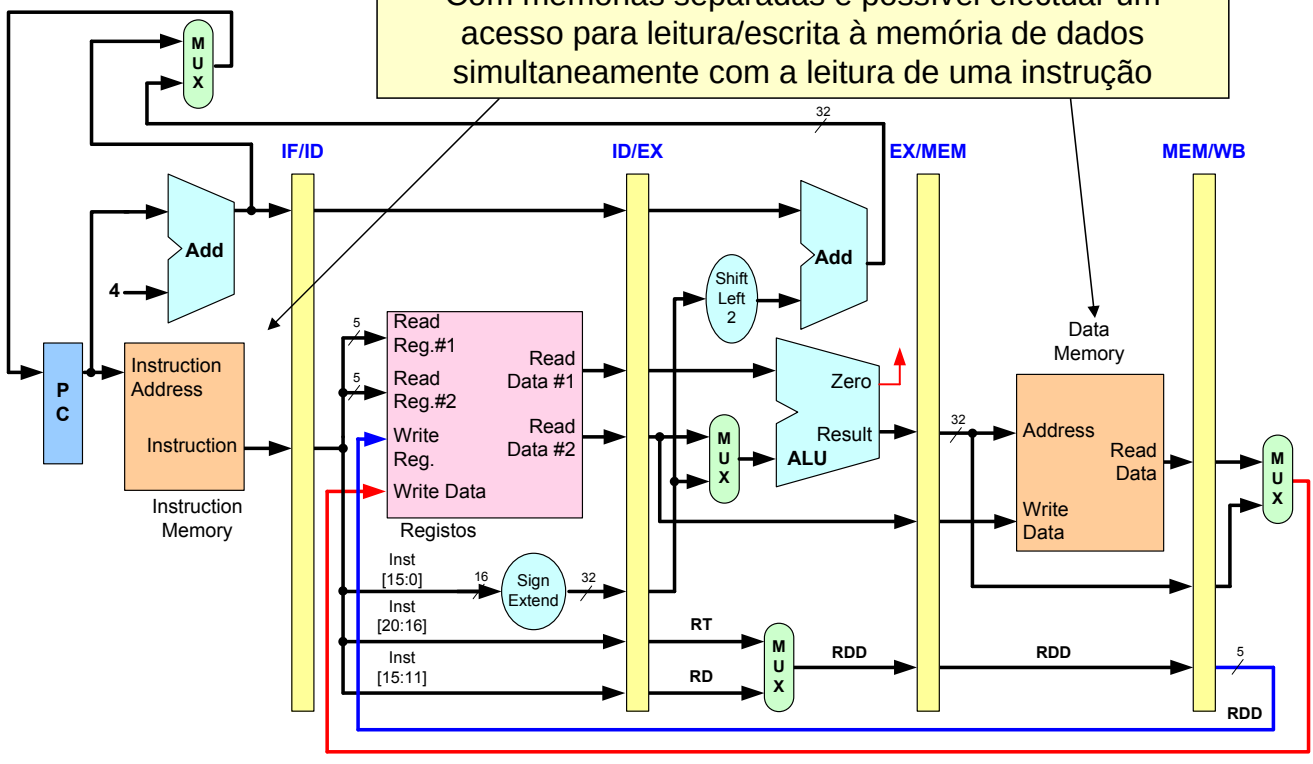
Pipelining: *Hazards* estruturais



- No quarto estágio da primeira instrução e no primeiro da quarta instrução é necessário efectuar, 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**

Pipelining: *Hazards* estruturais

Com memórias separadas é possível efectuar um acesso para leitura/escrita à memória de dados simultaneamente com a leitura de uma instrução

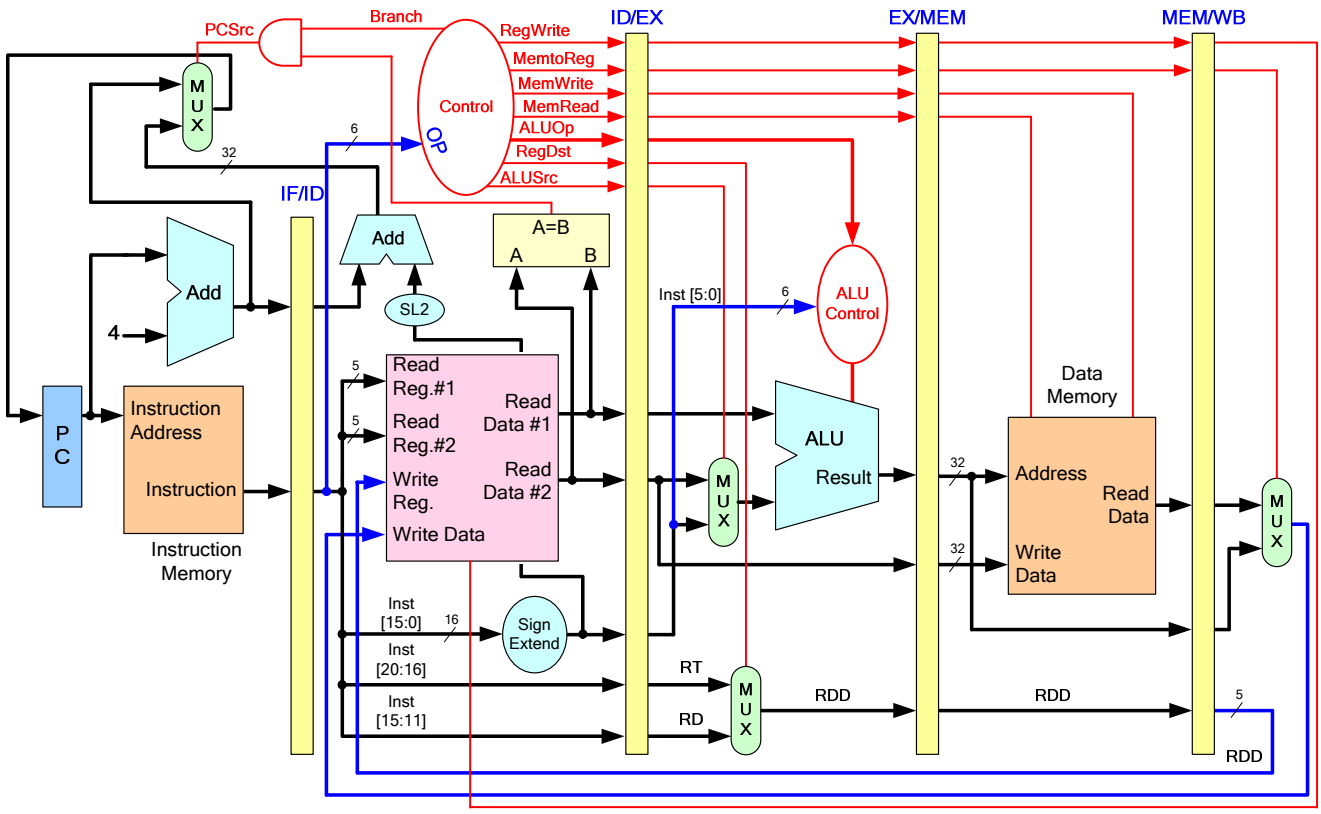


Pipelining: *Hazards* de controlo

- 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**, em particular com as de salto condicional *(branches):*
  - 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 de código
  - Assim, nos exemplos que se seguem, supõe-se que a **comparação dos operandos é efectuada no 2º estágio**, através de *hardware* adicional
  - Do mesmo modo, **o cálculo do *Branch Target Address* passa também a ser efectuado no 2º estágio**, e não no 3º

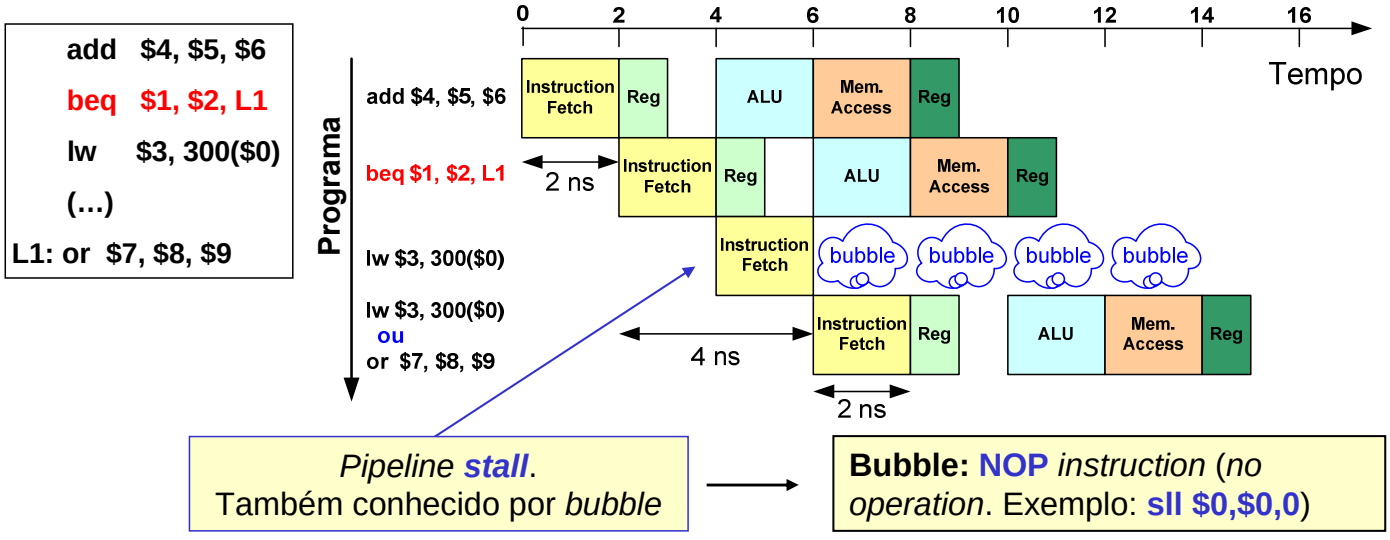
No exemplo da tarefa de tratar da roupa, um hazard de controlo resultaria, por exemplo da necessidade de avaliar se a quantidade de detergente e a temperatura da água são suficientes para a roupa ficar bem lavada. Só após a secagem da roupa da primeira carga é possível observar o resultado e determinar se é ou não necessário reajustar o detergente e a temperatura.

Pipelining: *Datapath* com unidade de controlo



Pipelining: *Hazards* de controlo

- Há mais do que uma solução para lidar com os *hazards de controlo*. A primeira que iremos observar é designada em Inglês por ***stalling*** ("empatar")
- Nesta estratégia a unidade de controlo atrasa a execução da próxima instrução até saber o resultado do *branch* condicional. É uma solução conservativa que tem um preço em termos de tempo de execução



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

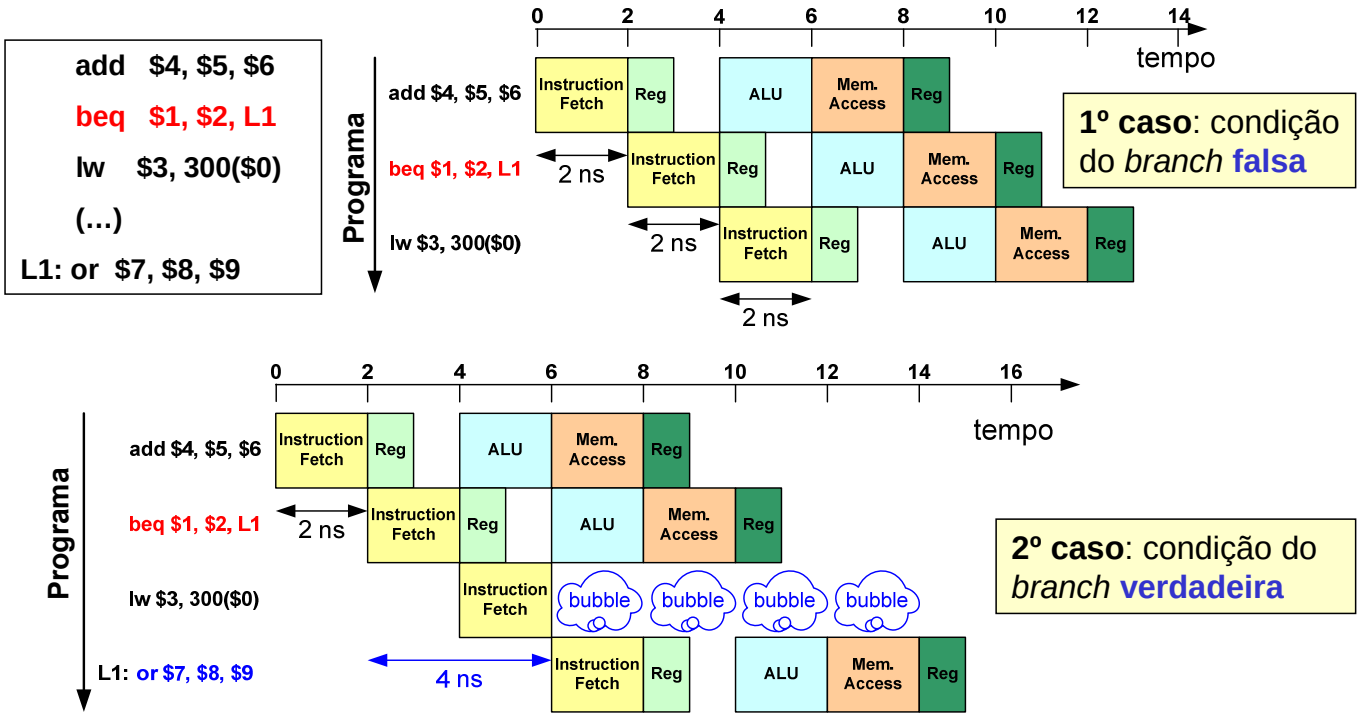
Bubble: NOP *instruction* (*no operation*. Exemplo: **sll $0,$0,0**)

Pipelining: *Hazards* de controlo

- Uma solução alternativa ao *pipeline stalling* é designada por *predição*:
  - Assume-se que a condição do *branch* falha sempre, pelo que a próxima instrução a ser executada será sempre a que estivar em PC+4
  - Nos casos em que a condição é verdadeira (i.e. o salto é para ser executado), a solução é usar a estratégia de *stalling*, abortando a execução da instrução que entretanto começara
- Esta estratégia permite poupar tempo se a previsão estiver certa
- Se a previsão falhar, a instrução entretanto lida tem de ser descartada, recomeçando a leitura na instrução correcta

No caso do tratamento da roupa corresponderia a dizer que, se estivermos bastante seguros de que a fórmula que usamos é adequada, podemos predizer que resultará bem, começando pois a lavar a segunda carga ainda antes de termos observado a primeira carga de roupa depois de seca. Se a nossa previsão falhar, por outro lado, teremos de repetir o processo para as cargas que entretanto estavam a ser processadas.

Pipelining: *Hazards* de controlo - *prediction*



Pipelining: *Hazards* de controlo - *prediction*

- As técnicas de predição usadas correntemente são na realidade mais elaboradas. Podem, por exemplo, adoptar estratégias baseadas em estereótipos de programação:
  - Se o *branch* for para um endereço anterior – caso dos *loops* – antecipar sempre que a condição é verdadeira, i.e., o branch é tomado. Isto tem como consequência que a instrução que entra no *pipeline* é a residente no endereço alvo do *branch*
  - Se o *branch* for para um endereço posterior, antecipar que a condição é sempre falsa
  - Se a previsão assumida não estiver certa, a instrução entretanto lida tem de ser descartada, recomeçando a leitura na instrução correcta
- As técnicas mais sofisticadas adoptam mecanismos de predição dinâmica baseados em análise estatística do comportamento de cada *branch* ao longo da execução do programa

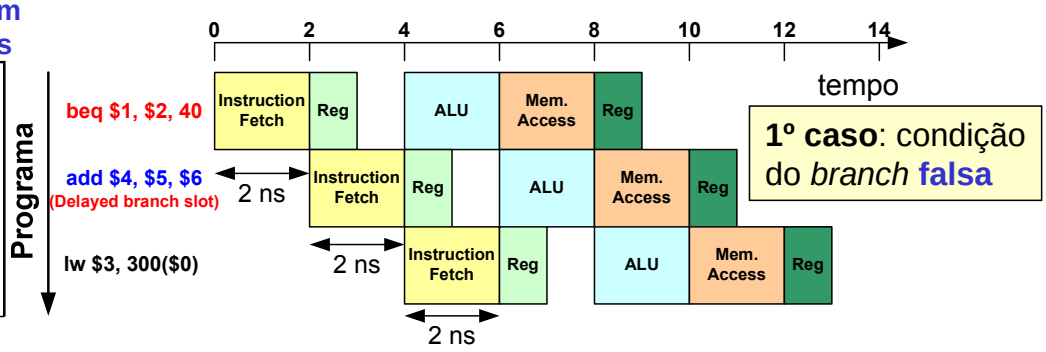
Pipelining: *Hazards* de controlo – *delayed branch*

- Uma terceira alternativa para resolver os *hazards* de controlo é designada por ***delayed branch***
- Nesta abordagem, o processador executa sempre a instrução que se segue ao *branch*
- Esta técnica é escondida do utilizador pelo *Assembler*
  - organiza as instruções por forma a trocar a ordem do *branch* com a instrução anterior, desde que esta não afecte ou seja afectada pelo *branch*
  - Quando não é possível efectuar a troca de instruções introduz um **NOP (no operation; ex.: sll, $0, $0, 0)**
- Esta é a solução adoptada pelo MIPS

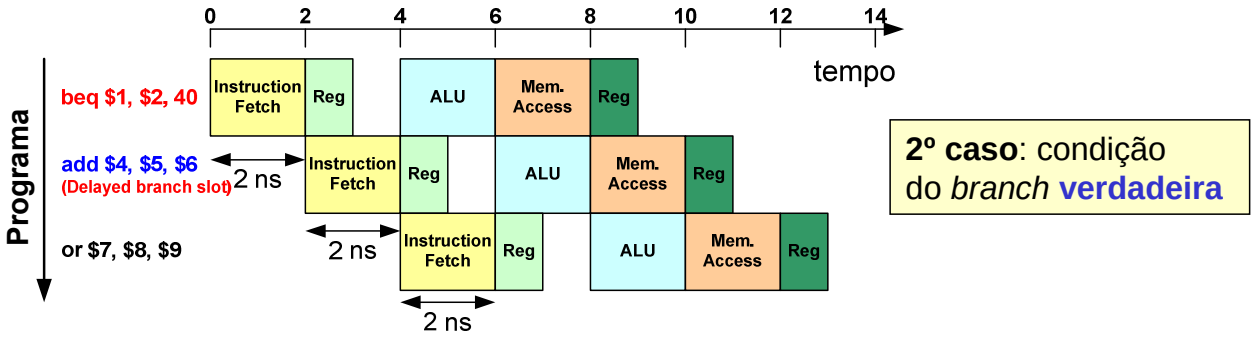
Pipelining: *Hazards* de controlo – *delayed branch*

Assembler troca a ordem das duas 1ªs instruções

```
      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**

Pipelining: *Hazards* de dados

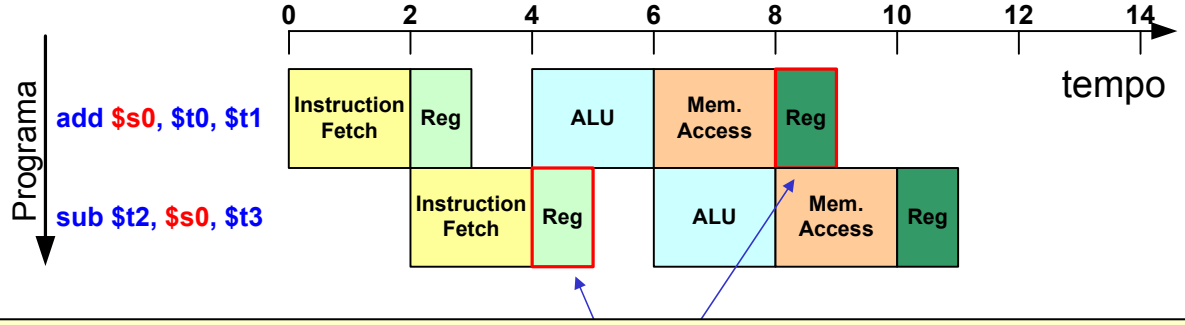
- O terceiro tipo de *hazards* resulta da **dependência** que possa existir entre o resultado calculado por uma instrução e o operando usado por outra que segue mais atrás no *pipeline*
- Se o resultado que vai ser necessário para a instrução que vem atrás no *pipeline* ainda não tiver sido armazenado, então essa instrução não poderá prosseguir porque irá tomar como operando um valor incorrecto (desactualizado)

No exemplo do tratamento da roupa, um hazard de dados corresponde a uma situação em que numa carga de roupa apenas estivesse presente uma peúga de cada par. Enquanto não forem lavadas as peúgas que emparelham com estas, não será possível arrumar o conjunto.

Pipelining: *Hazards* de dados

- Um *hazard de dados* resulta do aparecimento de uma instrução que manipula dados, sendo que esses dados dependem de uma instrução anterior que ainda não foi concluída. Por exemplo:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

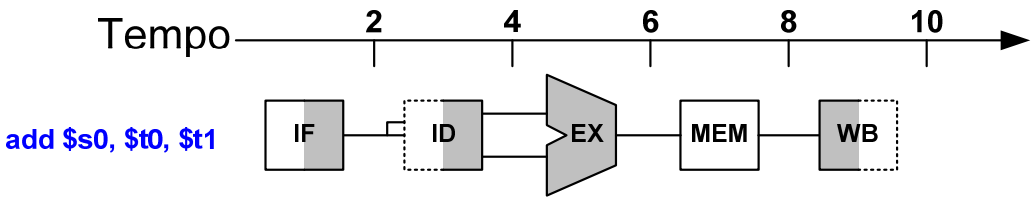


A instrução de subtracção não pode ser executada antes de o valor de **$s0** ser calculado e armazenado pela instrução anterior (o valor é necessário em **t = 4**, mas só vai ser escrito no registo destino em **t = 10**)

Pipelining: *Hazards* de dados

- A principal solução para a resolução de situações de *hazards de dados* resulta da observação de que não é necessário esperar pela conclusão da primeira instrução para tentar resolver o *hazard*
- A partir do momento em que a operação da primeira instrução seja executada (o que acontece na ALU – terceiro 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***
- Para exemplificar uma situação de *forwarding*, e tornar mais clara esta técnica, comecemos por apresentar uma versão gráfica simplificada da cadeia de *pipelining*

Pipelining: *Hazards* de dados

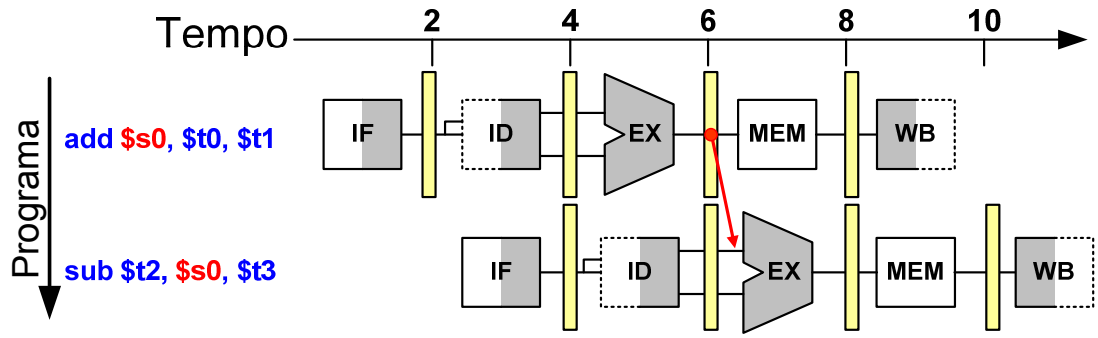


Nesta representação gráfica usamos símbolos para representar os recursos físicos:

- **IF** corresponde ao estágio de *instruction fetch,* representando o quadrado a memória de instrução
- A metade cinzenta 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 cinzenta está à esquerda, isso indica uma operação de **escrita** no elemento de estado respectivo (WB)

Pipelining: *Hazards* de dados

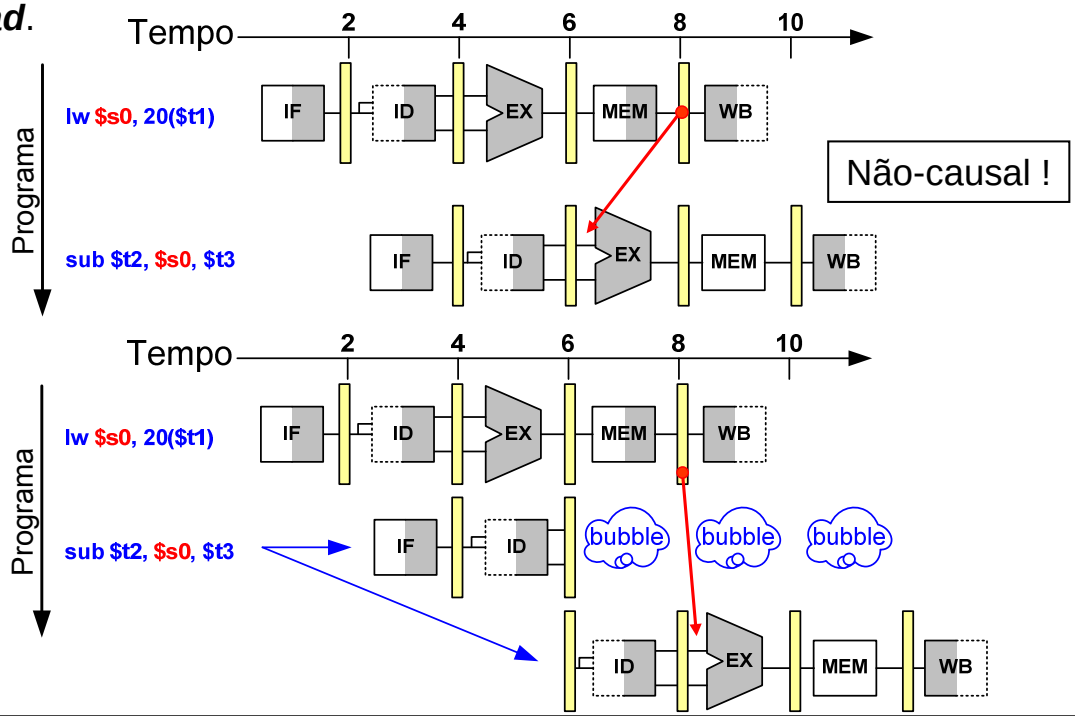
- Usando então a representação gráfica do *pipelining*, podemos voltar a observar a sequência de instruções que vimos ser responsável por uma situação de *hazard de dados*



- Como se pode observar, o ***forwarding*** do resultado à saída da ALU na instrução **add** para a entrada do estágio EX da instrução **sub,** resolve o *hazard* de dados
- Esta técnica só funciona, contudo, se o *forwarding* for efectuado para um estágio da instrução subsequente que ainda não tenha ocorrido (relação causal)

Pipelining: *Hazards* de dados

Um exemplo em que o *forwarding* não impede a ocorrência de *stalling* é o que decorre de uma instrução aritmética executada a seguir e na dependência de uma instrução de ***load***.



Pipelining: *Hazards* de dados

- Parte das situações de *hazards* de dados podem ainda ser atenuadas ou resolvidas pelo compilador, através da **reordenação de instruções**, desde que essa reordenação não comprometa o resultado

Exemplo:

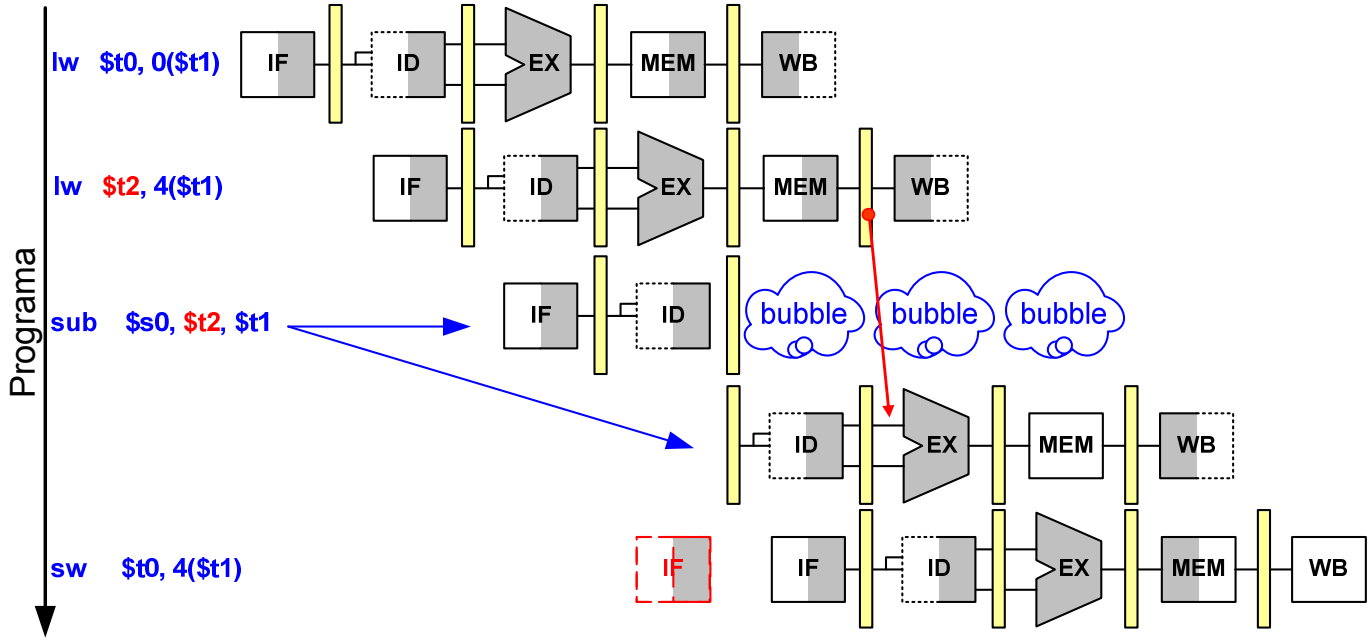
```
lw    $t0, 0($t1)
lw    $t2, 4($t1)
sub   $s0, $t2, $t1    # Situação de stalling por hazard de dados
sw    $t0, 4($t1)
```

Solução:

```
lw    $t0, 0($t1)
lw    $t2, 4($t1)
sw    $t0, 4($t1)
sub   $s0, $t2, $t1    # Situação de stalling resolvida por reordenação
```

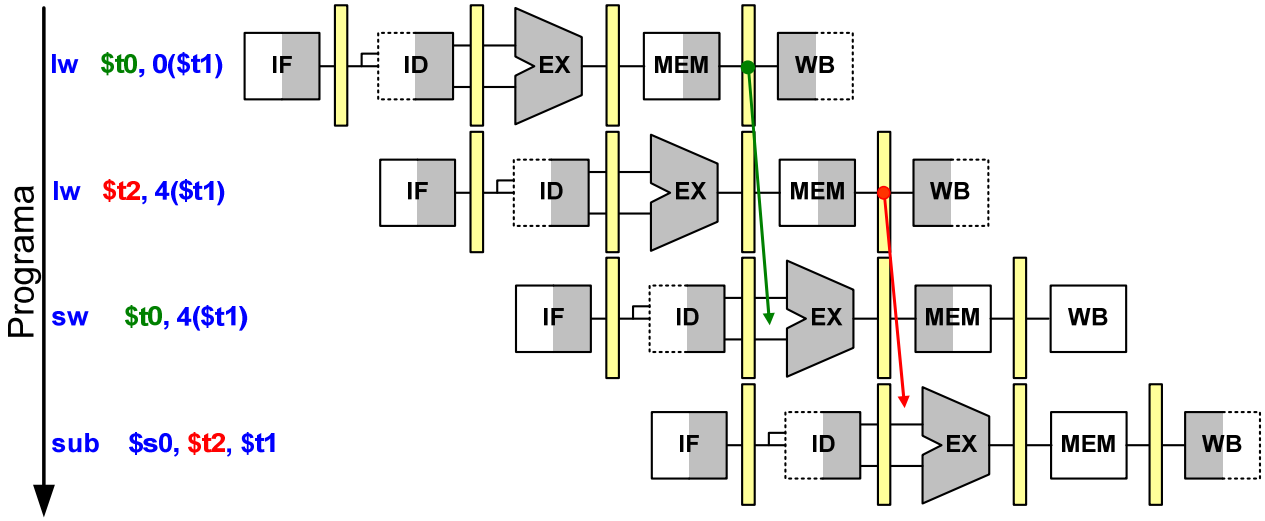

Pipelining: *Hazards* de dados

Situação de *stalling* por *hazard de dados*



Pipelining: *Hazards* de dados

Situação de *stalling* resolvido por reordenação. A reordenação gera um novo hazard de dados que também pode ser resolvido por forwarding

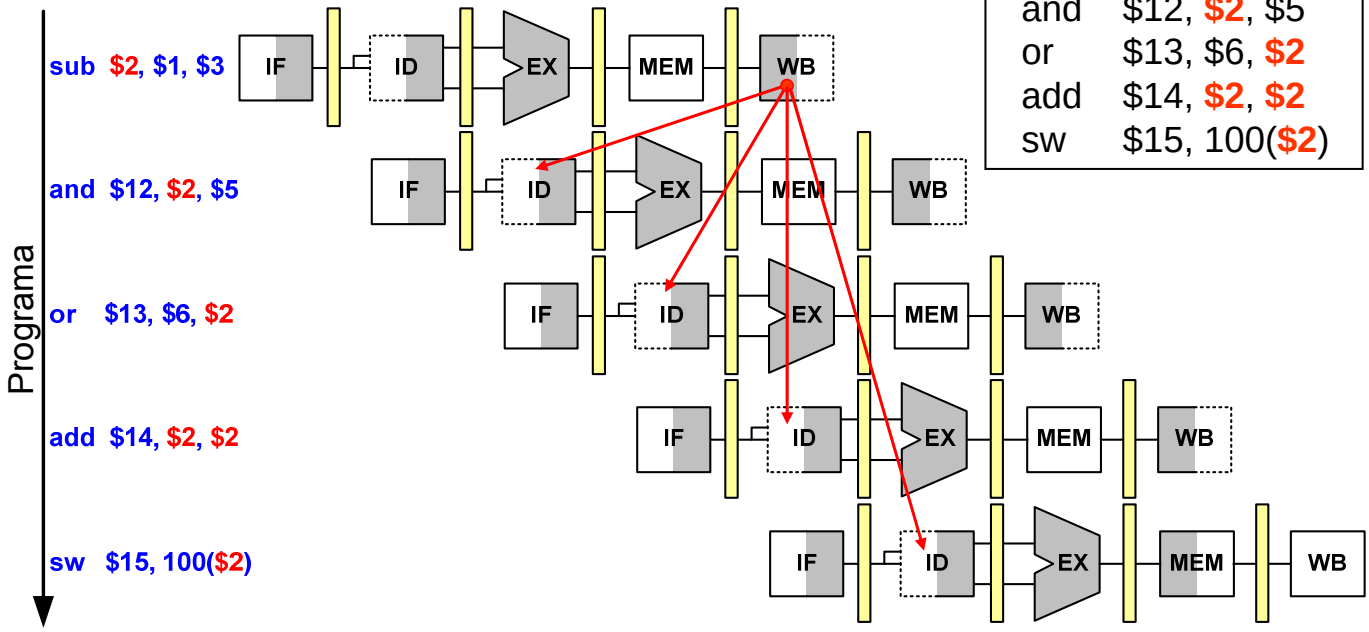


A sequência de instruções reordenada executa em menos 1 ciclo de relógio

Pipelining: *Hazards* de dados

Exemplo:

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

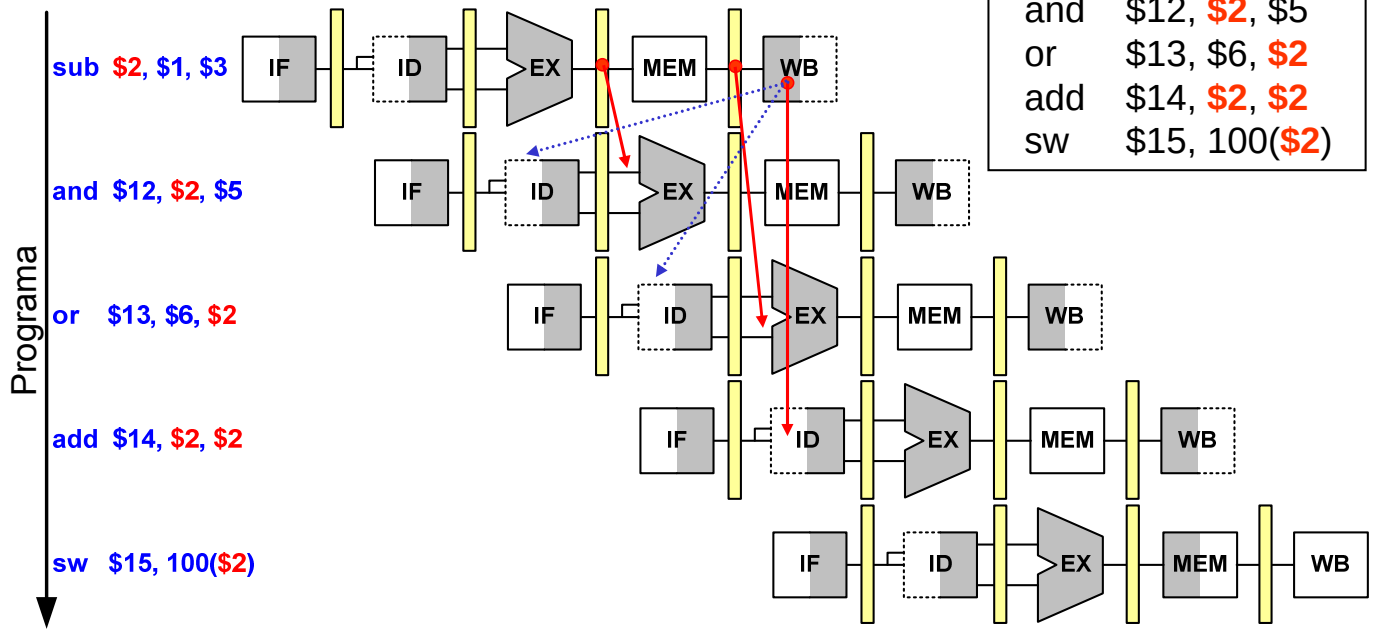


As linhas que indicam caminhos "para trás" no tempo correspondem a *hazards* de dados

Pipelining: *Hazards* de dados

Exemplo:

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



A dependência existente entre a etapa WB e a ID pode facilmente ser resolvida **desfasando** temporalmente a escrita e a leitura do "file register": a escrita é efectuada a meio do ciclo de relógio, enquanto que a leitura é realizada na 2ª metade do ciclo.

Pipelining: *Hazards* de dados

- Para resolver um *hazard* de dados através de ***forwarding*** é necessário:
  - **Detectar 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 excepção das instruções de *branch*, a generalidade das outras instruções necessitam dos valores correctos 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)**

Pipelining: *Hazards* de dados

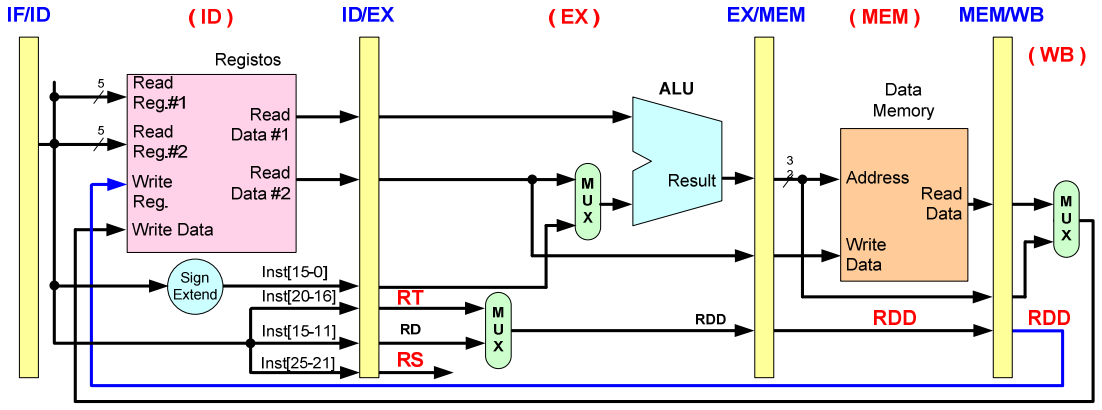
- As situações, correspondentes a *hazard* de dados, em que há necessidade de **encaminhar valores para a fase EX** são:

-1a) EX/MEM.**RDD** == ID/EX.**RS**
-1b) EX/MEM.**RDD** == ID/EX.**RT**

Instrução na fase **MEM** cujo registo destino é um dos registos operando de uma instrução que se encontra na fase EX

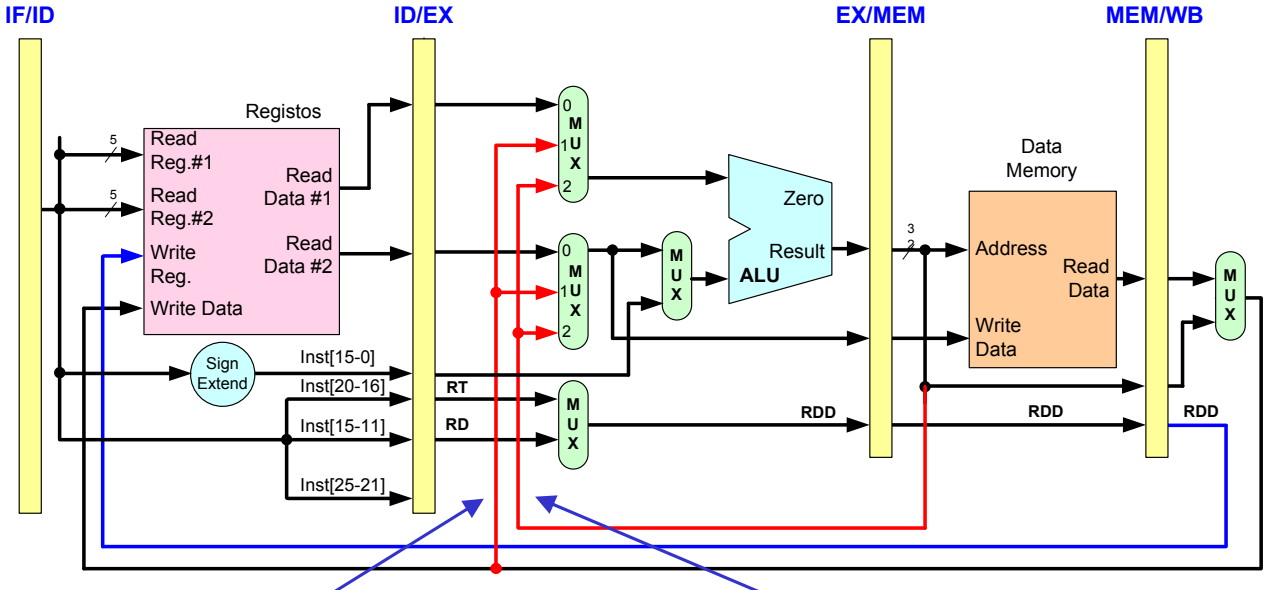
-2a) MEM/WB.**RDD** == ID/EX.**RS**
-2b) MEM/WB.**RDD** == ID/EX.**RT**

Instrução na fase **WB** cujo registo destino é um dos registos operando de uma instrução que se encontra na fase EX



Pipelining: *Hazards* de dados

Implementação da técnica de *forwarding*



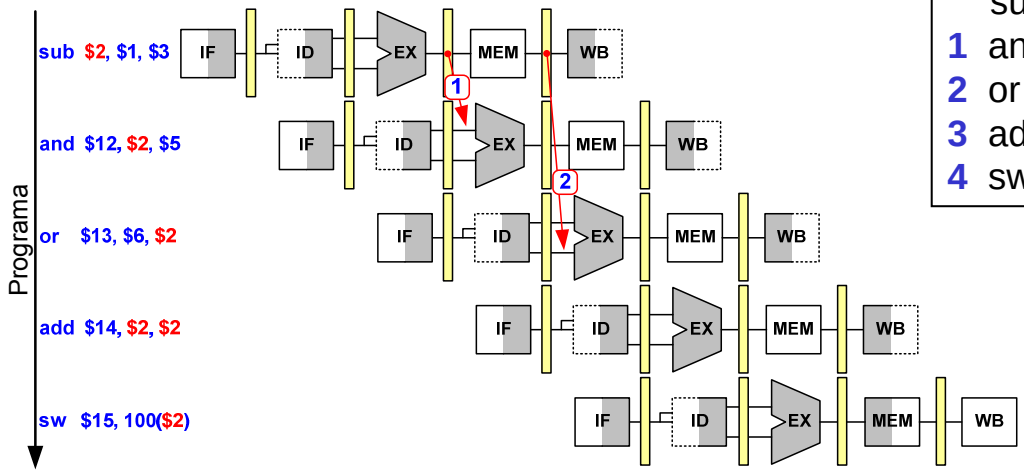
***Forwarding* da etapa WB**

***Forwarding* da etapa MEM**

Pipelining: *Hazards* de dados

Exemplo:

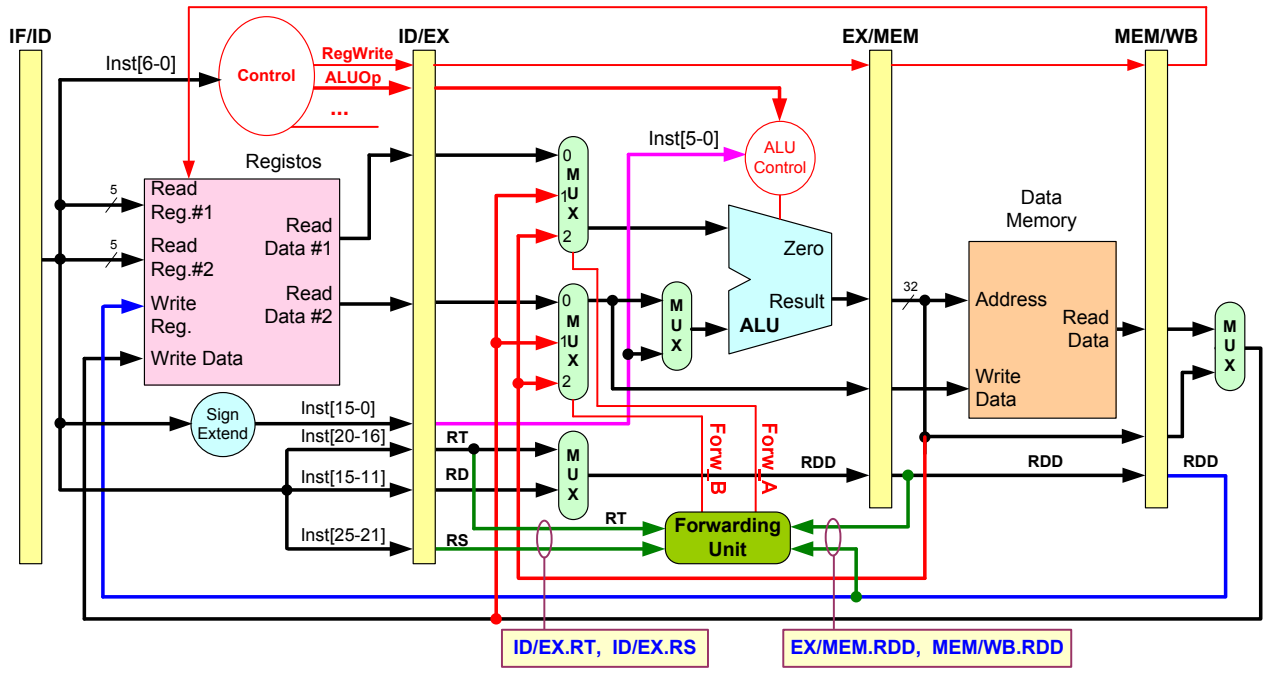
```
   sub  $2, $1, $3
1  and  $12, $2, $5
2  or   $13, $6, $2
3  add  $14, $2, $2
4  sw   $15, 100($2)
```



- No exemplo anterior, as situações de *hazard* de dados 1 e 2 podem ser detectadas através de:
  - 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)
- A situação 3 pode ser resolvida sem *forwarding* (não se considera *hazard*)
- A situação 4 também não corresponde a um *hazard* de dados

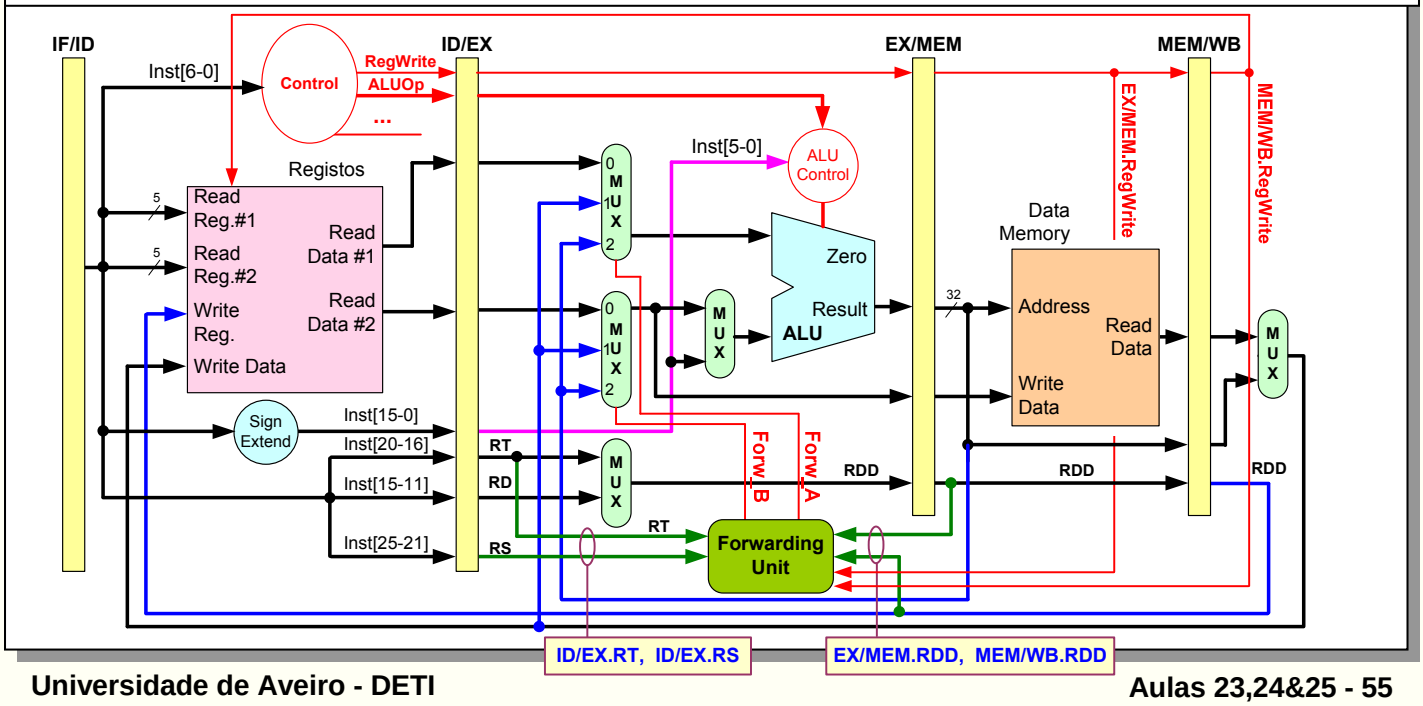
Pipelining: *Hazards* de dados

Parte do *datapath*, com unidade de controlo de *forwarding* (simplificada)

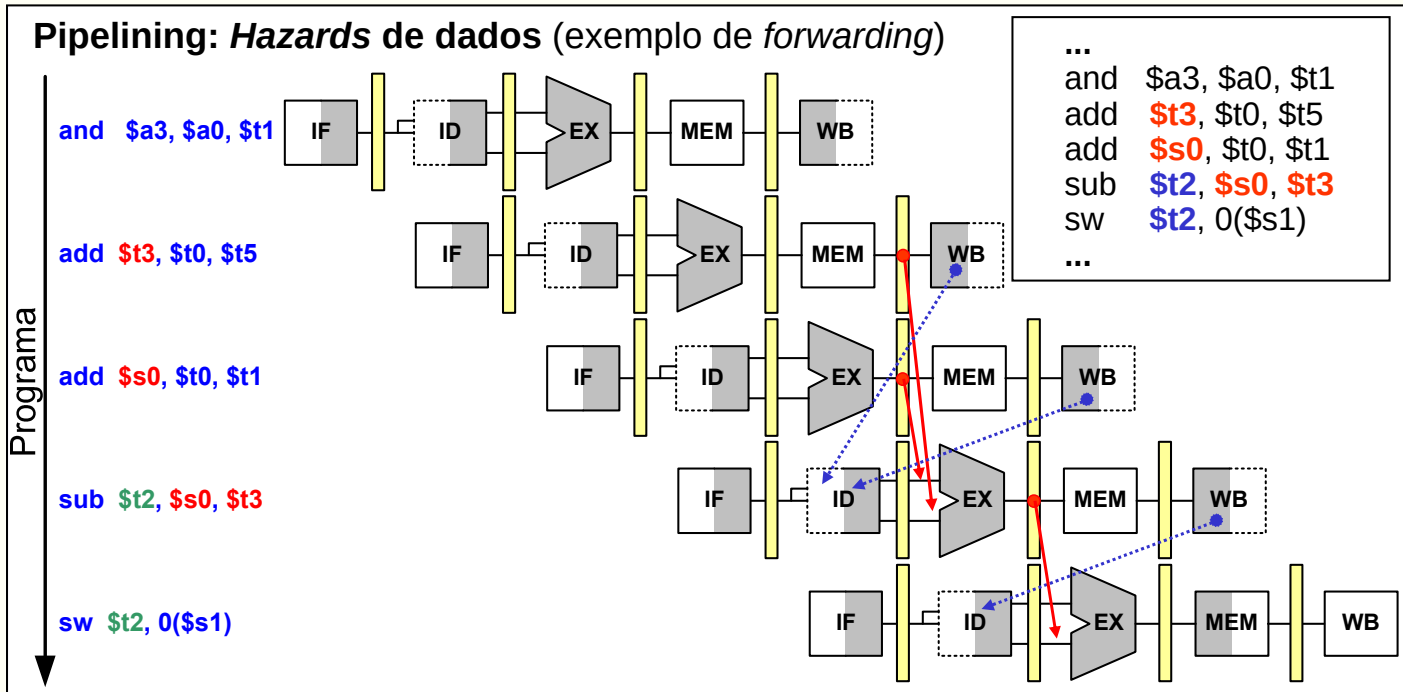


Instrução na **fase 5** que escreve num registo
- **if**((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RS)) **Forw_A = 01**
- **if**((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RT)) **Forw_B = 01**

Instrução na **fase 4** que escreve num registo
- **if**((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RS)) **Forw_A = 10**
- **if**((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RT)) **Forw_B = 10**



Pipelining: *Hazards* de dados (exemplo de *forwarding*)



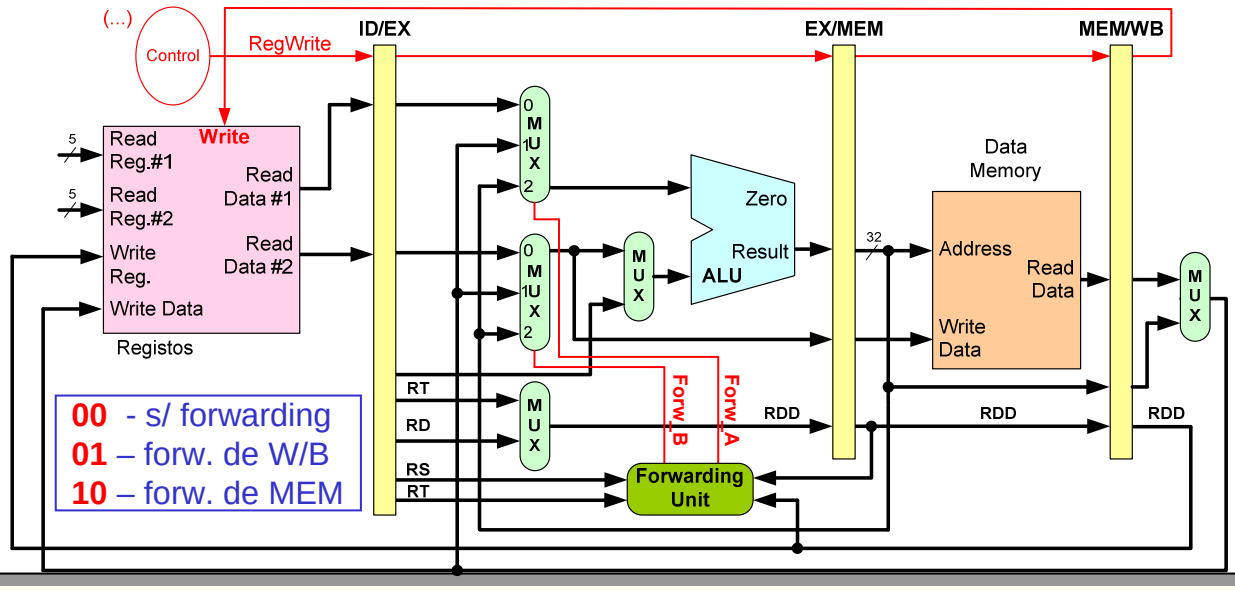
- A instrução "**sub $t2, $s0, $t3**" apresenta duas situações de hazards de dados: dependência dos valores de $t3 e de $s0 calculados pelas duas instruções que estão à frente no pipeline
- A instrução "**sw $t2, 0($s1)**" apresenta igualmente uma situação de hazard de dados

Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RS))	**Forw_A = 01**
if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RT))	**Forw_B = 01**
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RS))	**Forw_A = 10**
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RT))	**Forw_B = 10**

```
...
and  $a3, $a0, $t1
add  $t3, $t0, $t5
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
sw   $t2, 0($s1)
...
```



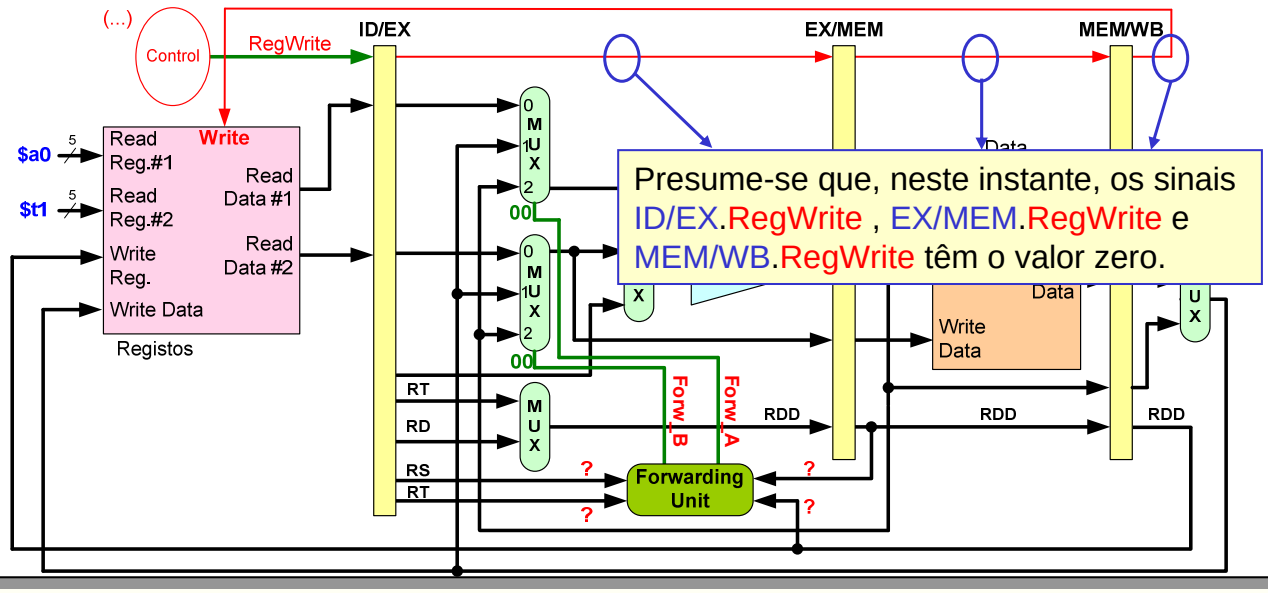
Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RS))	*Forw_A = 01*
if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RT))	*Forw_B = 01*
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RS))	*Forw_A = 10*
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RT))	*Forw_B = 10*

```
...
and  $a3, $a0, $t1
add  $t3, $t0, $t5
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
sw   $t2, 0($s1)
...
```

and $a3, $a0, $t1



Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

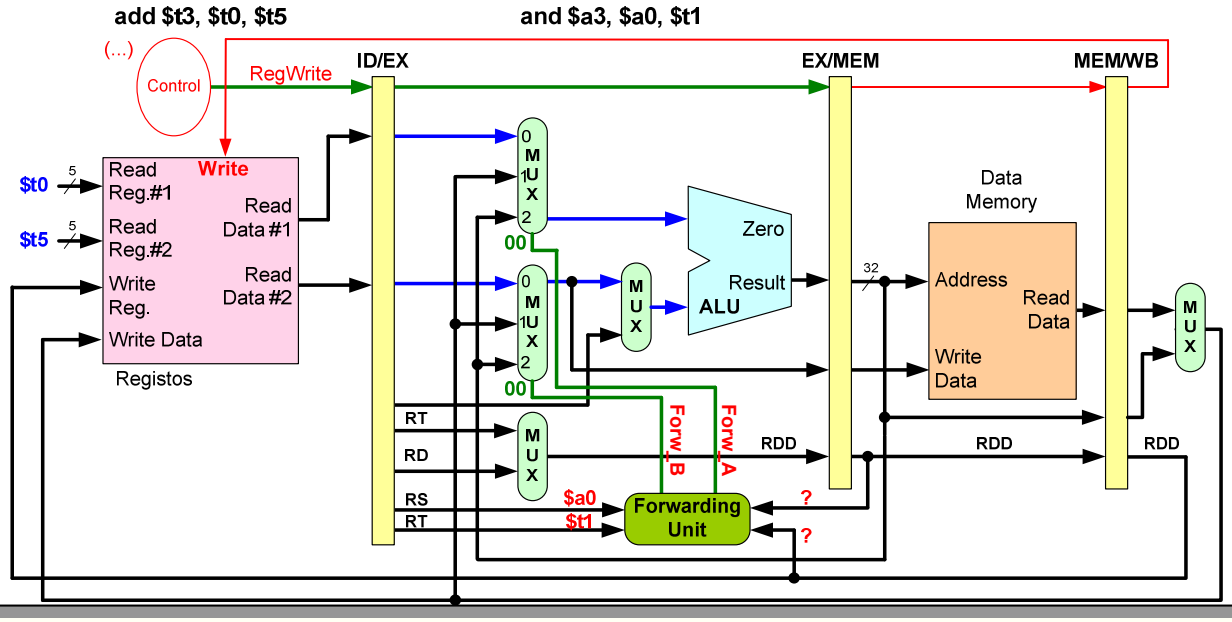
if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RS)) *Forw_A = 01*

if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RT)) *Forw_B = 01*

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RS)) *Forw_A = 10*

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RT)) *Forw_B = 10*

```
...
▶and  $a3, $a0, $t1
 add  $t3, $t0, $t5
 add  $s0, $t0, $t1
 sub  $t2, $s0, $t3
 sw   $t2, 0($s1)
...
```



Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

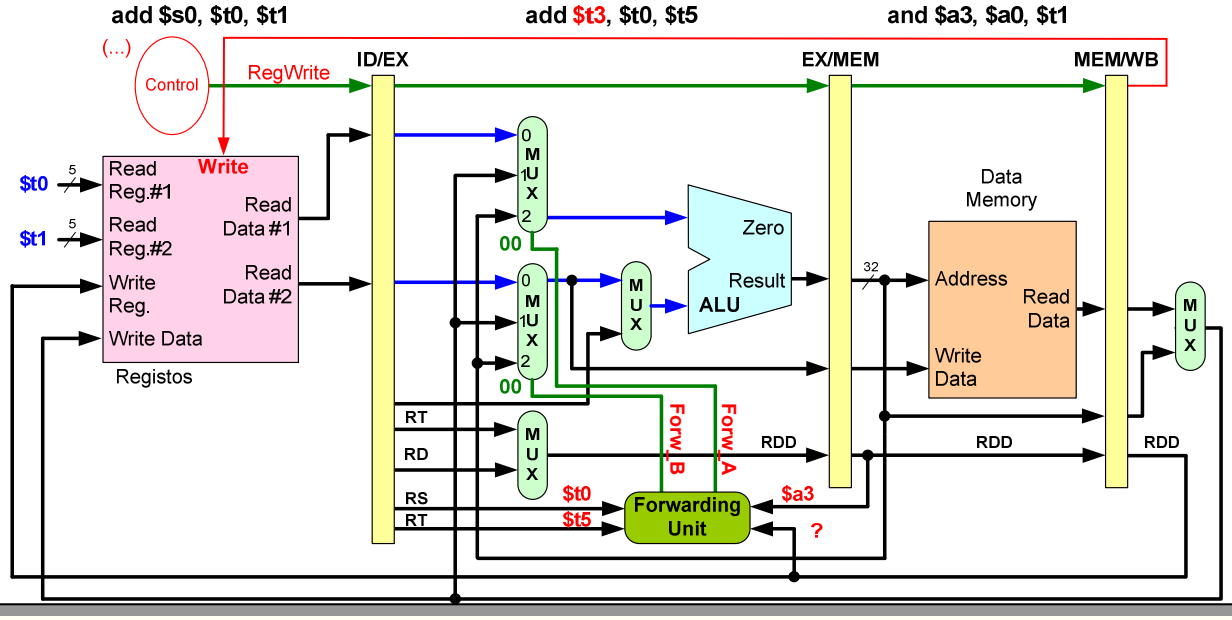
if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RS)) *Forw_A = 01*

if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RT)) *Forw_B = 01*

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RS)) *Forw_A = 10*

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RT)) *Forw_B = 10*

```
...
 and  $a3, $a0, $t1
▶add  $t3, $t0, $t5
 add  $s0, $t0, $t1
 sub  $t2, $s0, $t3
 sw   $t2, 0($s1)
...
```



Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

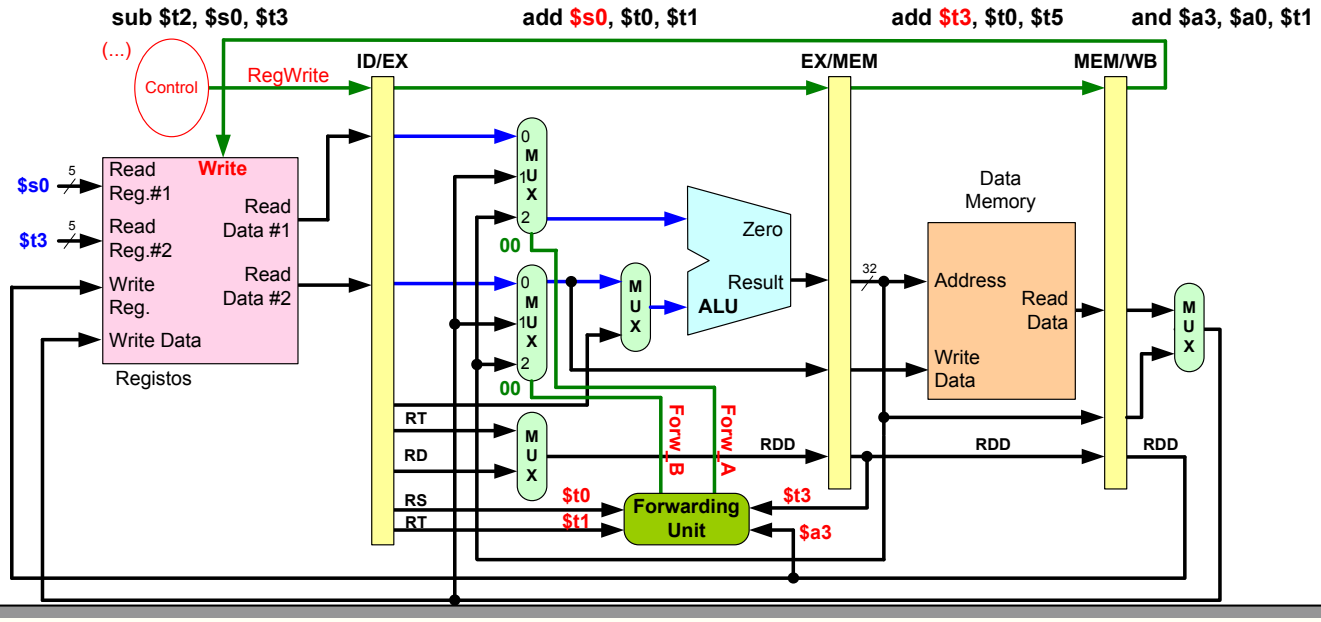
if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RS)) *Forw_A = 01*

if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RT)) *Forw_B = 01*

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RS)) *Forw_A = 10*

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RT)) *Forw_B = 10*

```
...
and  $a3, $a0, $t1
add  $t3, $t0, $t5
▶add  $s0, $t0, $t1
sub  $t2, $s0, $t3
sw   $t2, 0($s1)
...
```



Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

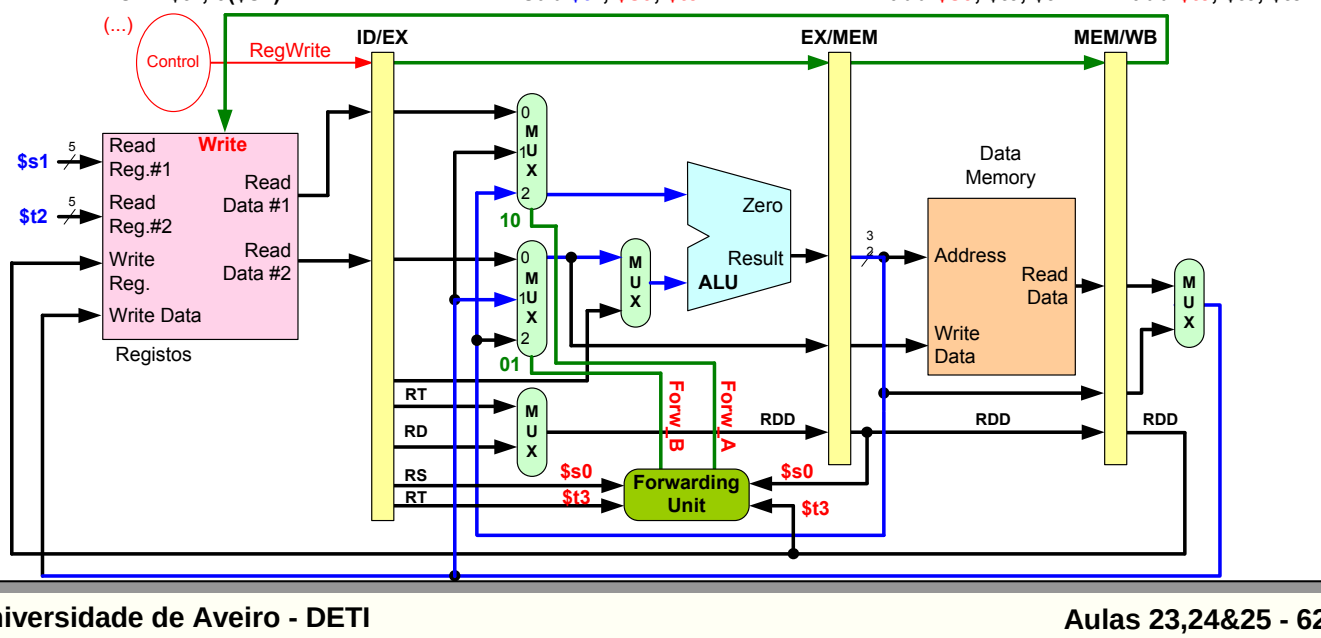
if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RS)) *Forw_A = 01*

if((MEM/WB.RegWrite == 1) **and** (MEM/WB.RDD == ID/EX.RT)) **Forw_B = 01**

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RS)) **Forw_A = 10**

if((EX/MEM.RegWrite == 1) **and** (EX/MEM.RDD == ID/EX.RT)) *Forw_B = 10*

```
...
and  $a3, $a0, $t1
add  $t3, $t0, $t5
add  $s0, $t0, $t1
▶sub  $t2, $s0, $t3
sw   $t2, 0($s1)
...
```

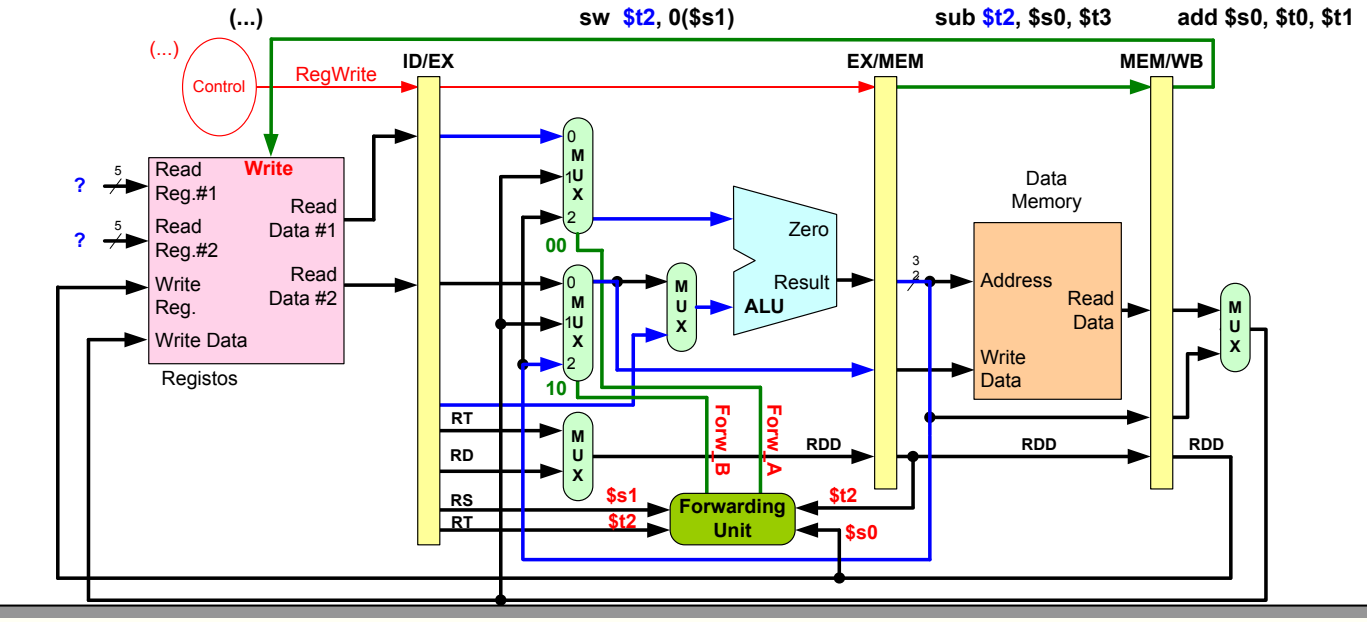


Pipelining: *Hazards* de dados (exemplo de *forwarding*)

Forw_A = 00; Forw_B = 00

if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RS))	*Forw_A = 01*
if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RT))	*Forw_B = 01*
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RS))	*Forw_A = 10*
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RT))	**Forw_B = 10**

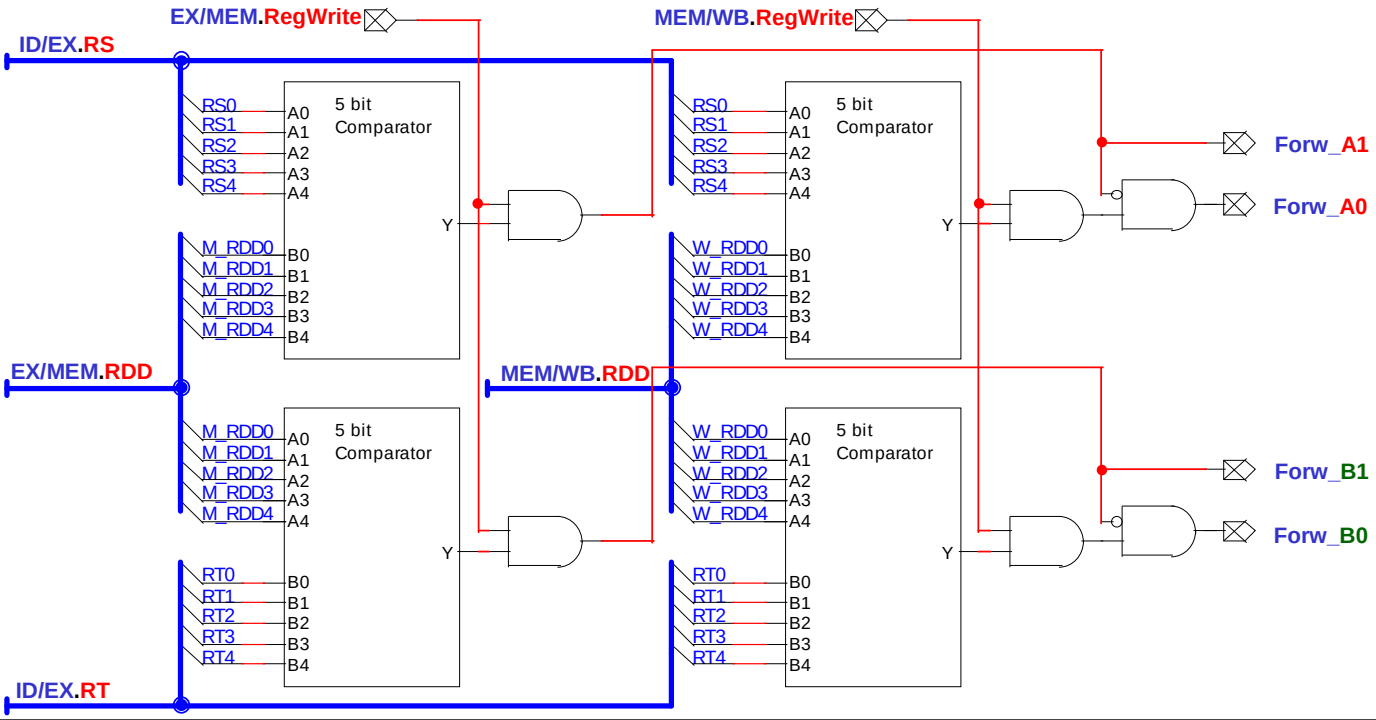
```
...
and  $a3, $a0, $t1
add  $t3, $t0, $t5
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
▶sw  $t2, 0($s1)
...
```



Pipelining: Estrutura interna da Unidade de *Forwarding*

Forw_A = 00; Forw_B = 00

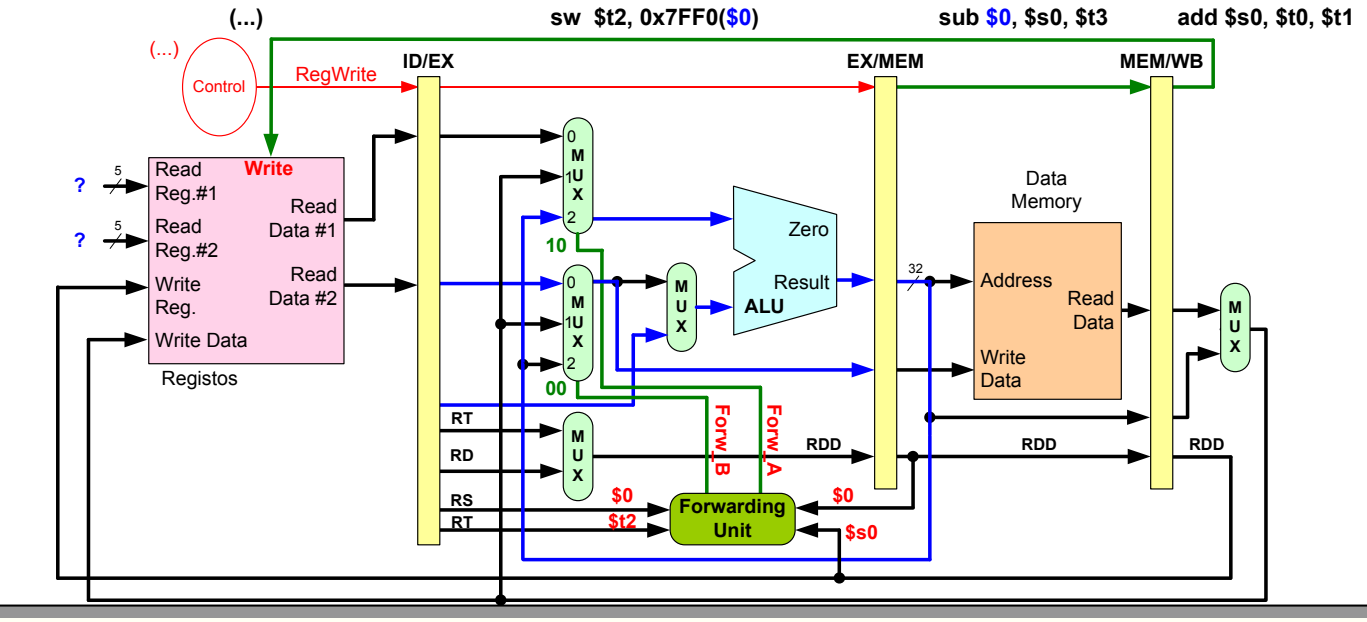
if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RS))	**Forw_A = 01**
if((MEM/WB.RegWrite == 1)	**and** (MEM/WB.RDD == ID/EX.RT))	**Forw_B = 01**
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RS))	**Forw_A = 10**
if((EX/MEM.RegWrite == 1)	**and** (EX/MEM.RDD == ID/EX.RT))	**Forw_B = 10**



Pipelining: *Hazards* de dados

Problema:

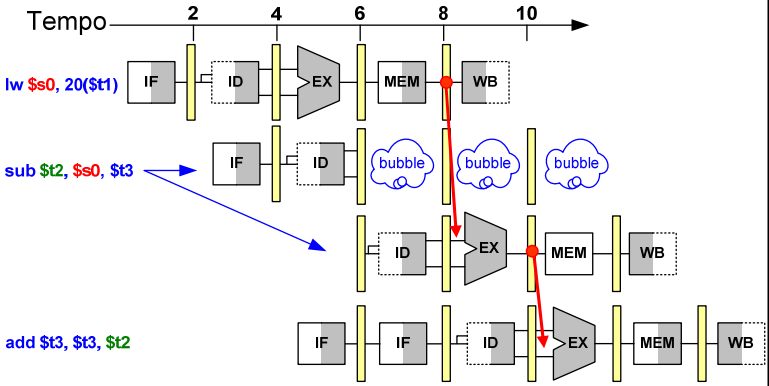
- O que acontece neste *datapath*, caso o *hazard* de dados resulte de um valor de EX/MEM.RDD = $0 ou MEM/WB.RDD = $0?
- Como resolver o problema ?



Pipelining: *Hazards* de dados

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

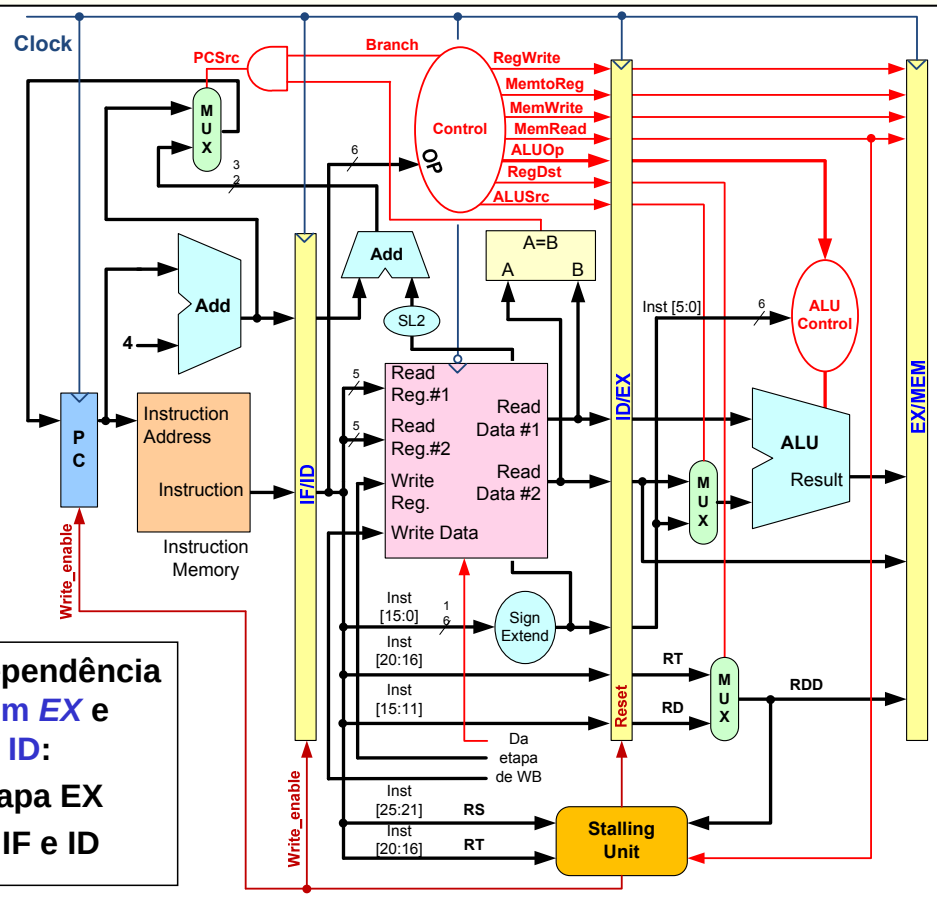
```
lw   $s0, 20($t1)
sub  $t2, $s0, $t3
add  $t3, $t3, $t2
```



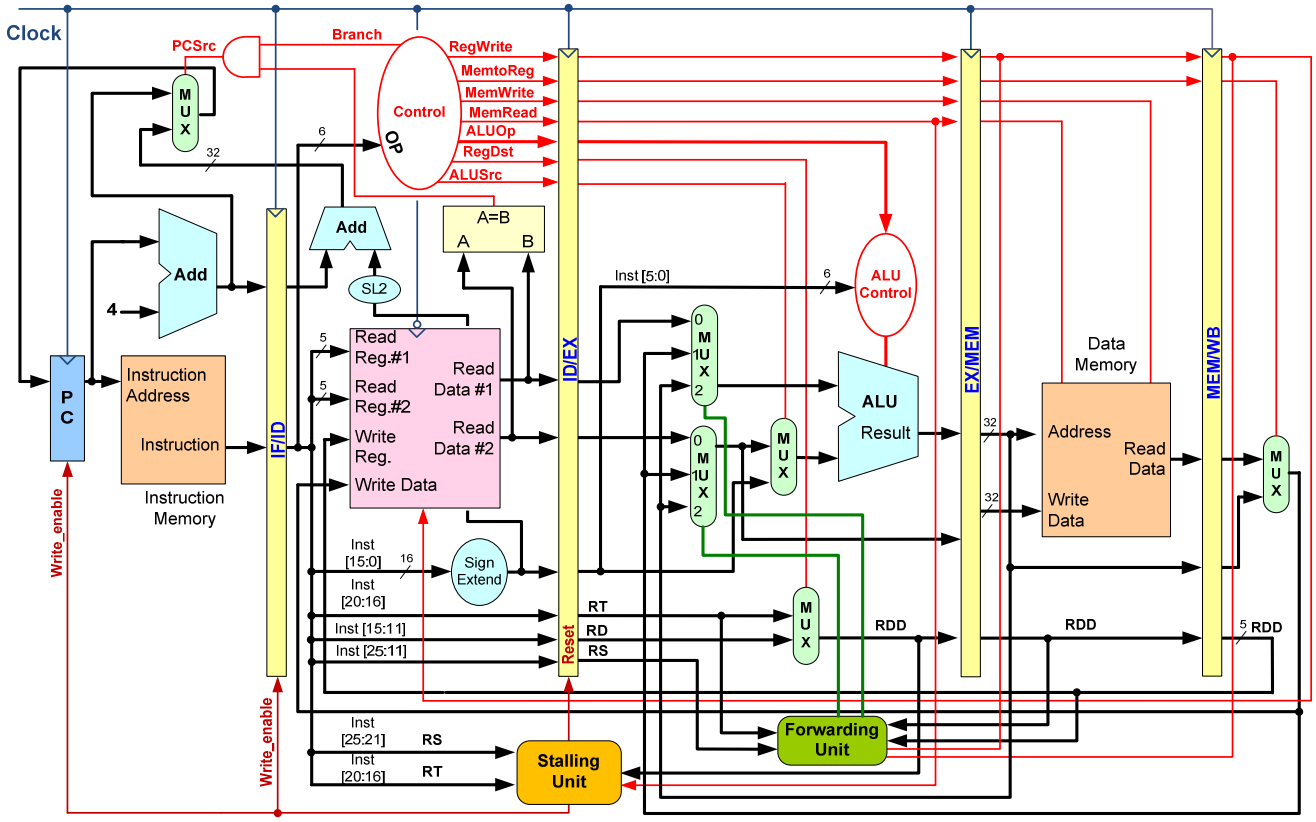
- A situação de *stall* tem que ser desencadeada quando a instrução tipo R está na sua fase ID. Como fazer?
  - **Inserir *bubble*** na etapa EX: fazer o **reset** do sinal **RegWrite** no registo ID/EX (de forma síncrona com o clock)
  - **Congelar, durante 1 ciclo de relógio, as etapas IF e ID** (i.e. impedir a escrita no registo IF/ID e impedir que seja feita a actualização do PC)
- Como detectar? (ID/EX.MemRead == 1) **and** (ID/EX.RDD == IF/ID.RS **or** ID/EX.RDD == IF/ID.RT)

- **Inserir *bubble*** na etapa EX: fazer o reset do sinal RegWrite no registo ID/EX (de forma síncrona com o clock)
- **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 actualização do PC)

- ***Stalling* devido a uma dependência entre uma instrução *lw* em *EX* e uma instrução tipo R em ID:**
  - **Inserir *bubble* na etapa EX**
  - **Congelar as etapas IF e ID**



Data path pipelining completo (sem *forwarding* nas instruções de *branch*)



E agora…

… il grand finale

Data path pipelining completo (sem *forwarding* nas instruções de *branch*)
mas com **Jump**