

Aulas 24, 25 & 26

Pipelining:

Definição - exemplo prático por analogia

Adaptação do conceito ao caso do MIPS

Problemas da solução *datapath pipelined*

Construção do *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 e Silva

Pipelining

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

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

Pipelining

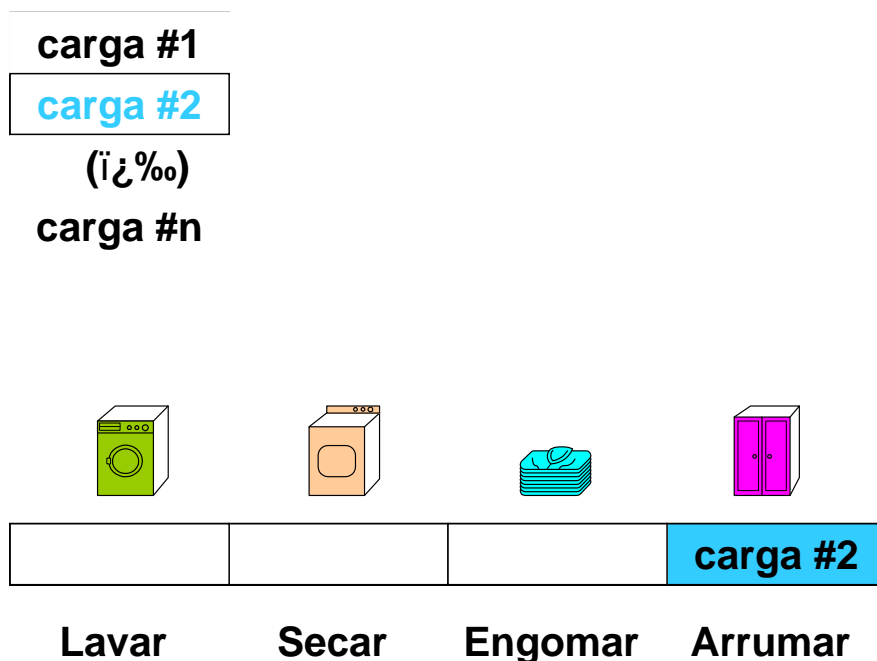
O exemplo *pipelining* que iremos observar de seguida apoia-se num conjunto de tarefas simples e intuitivas: o processo de tratamento da roupa suja

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

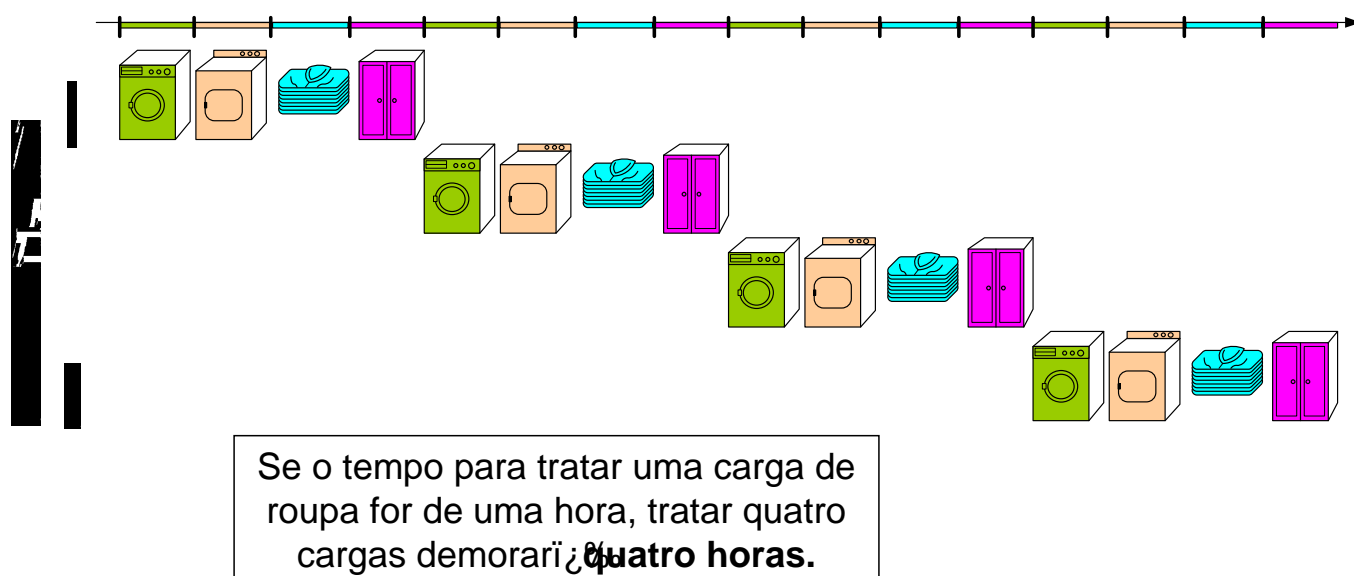
Pipelining

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



Pipelining

Este processo pode então ser descrito temporalmente da seguinte modo:

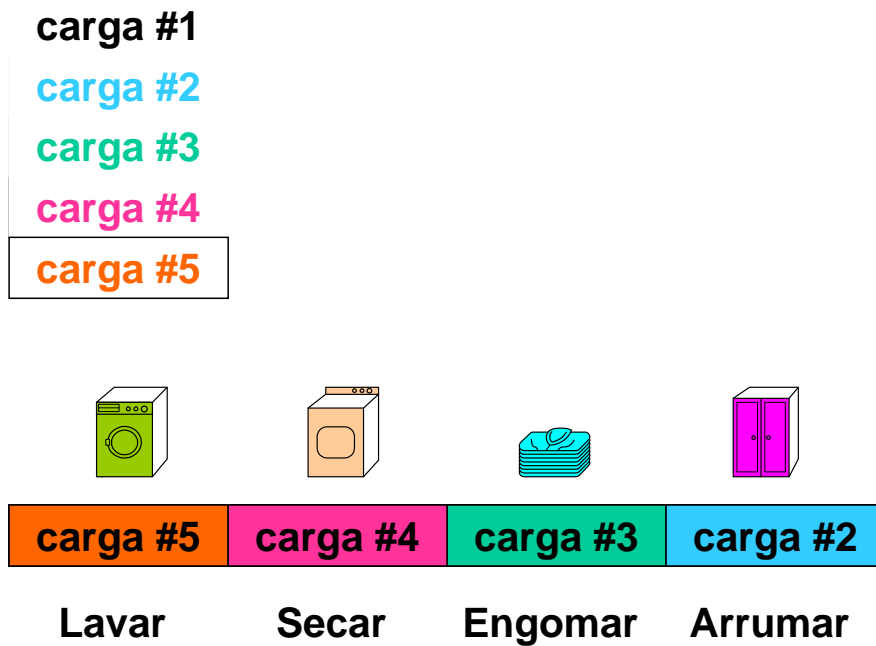


Pipelining

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

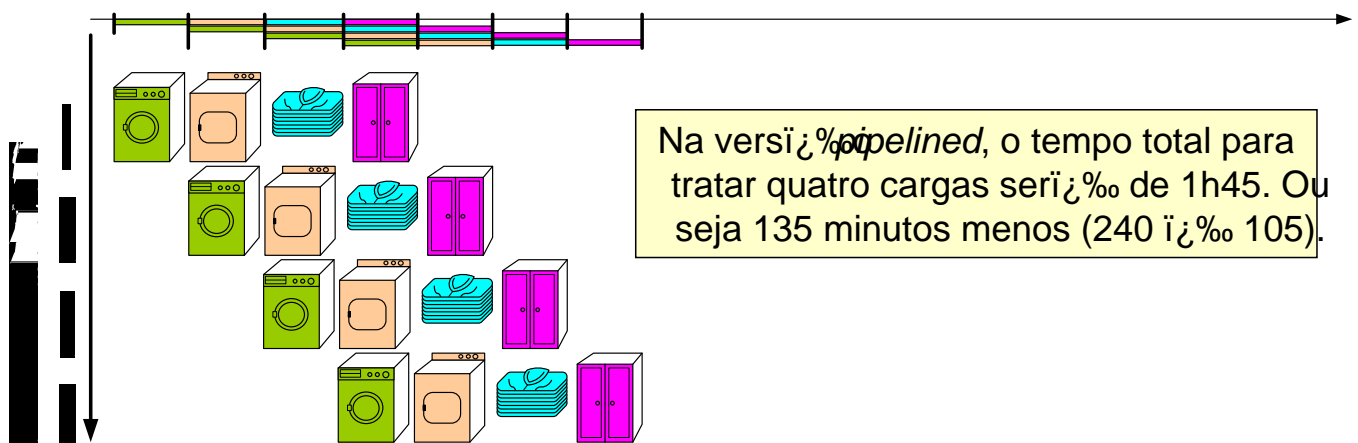
Pipelining

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



Pipelining

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



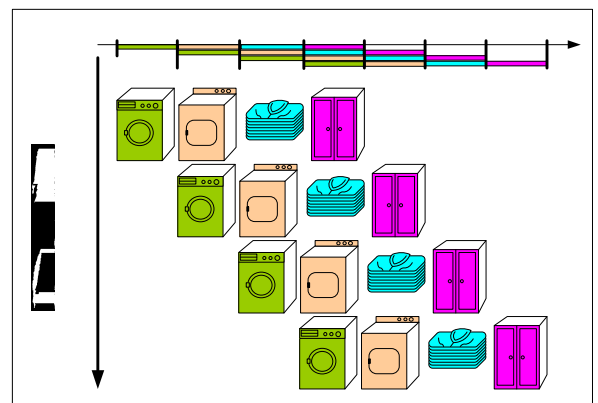
Pipelining

- 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 *pipelined*
- A eficiência da solução *pipelining* decorre do facto de, para um número muito grande de cargas de roupa, todos os ~~passos~~ *stages* intermédios estarem a executar em paralelo
- O resultado é o aumento do número total de cargas d 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*: $T_{\text{NON-PIPELINE}} = N \cdot F$

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

Ganho obtido com a solução *pipelined*: $\frac{\text{Desempenho}_{\text{PIPELINE}}}{\text{Desempenho}_{\text{NON-PIPELINE}}} = \frac{T_{\text{NON-PIPELINE}}}{T_{\text{PIPELINE}}} = \frac{N \cdot F}{(F - 1) + N}$

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

Pipelining

• No limite, para um número de cargas de roupa elevado, o **ganho de desempenho** (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e noutro modo) 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 de desempenho** é igual ao número de estágios do

• No exemplo observado, o limite teórico estabelecido pela solução *pipelined* é quatro vezes mais rápida do que a solução *non-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 tratamento da roupa, podem igualmente ser aplicados aos processadores

• No caso do MIPS, como já sabemos, as instruções são divididas genericamente em cinco fases (**estágios**):

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) **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

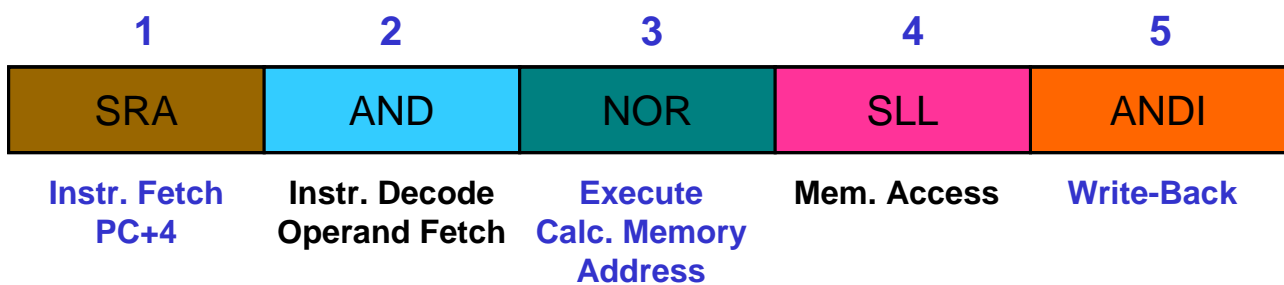
Pipelining

```

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

```

(i %)

**Pipelining Problemas (Exemplo 1)**

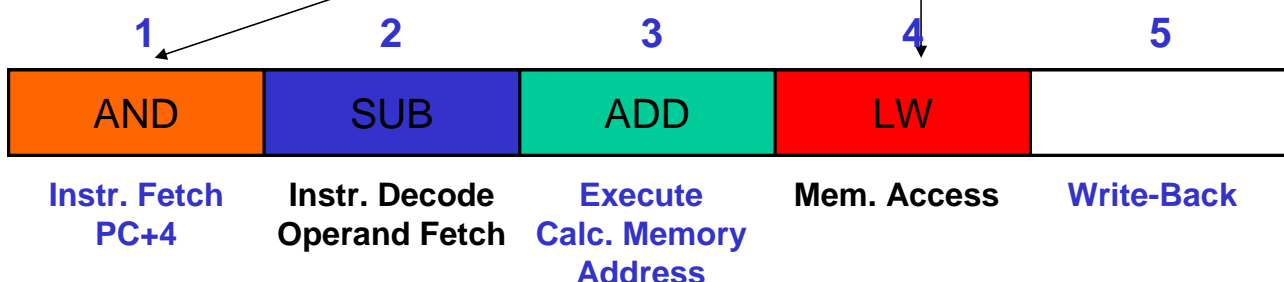
```

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

```

o 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*

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
```

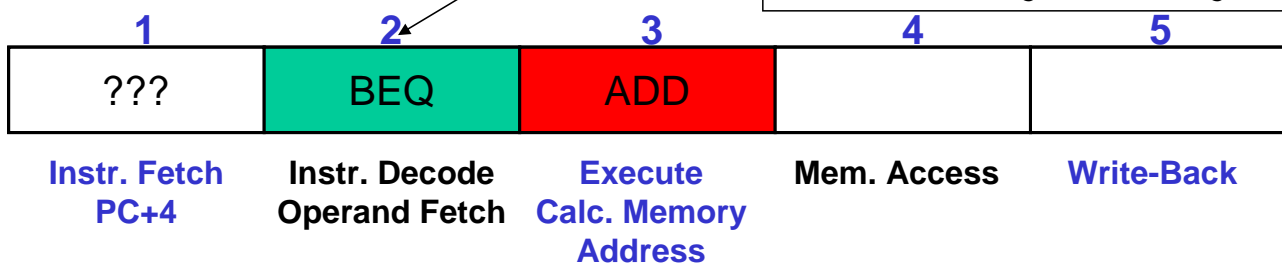
```
lw $s3, 300($a0)
```

(i.e.)

```
Z1: or $t7, $t8, $t9
```

Qual a próxima instrução a ler da memória (o ou o or) assumindo que a decisão do *branch* será tomada no 3º estágio pipeline?

Mesmo admitindo que existe h/w dedicado para avaliar a condição *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.



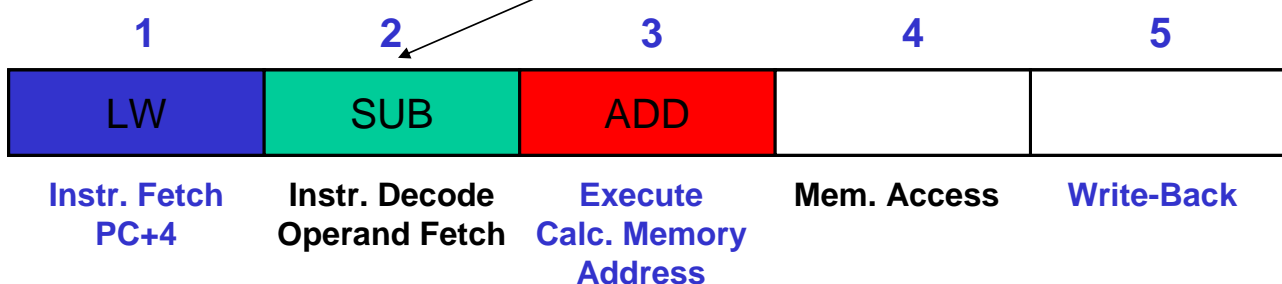
Pipelining Problemas (Exemplo 3)

```
add $s0, $t0, $t1
```

```
sub $t2, $s0, $t3
```

```
lw $t4, 0($s2)
```

A instrução de subtração 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.



Pipelining

Para tornar a discussão mais concreta, vamos construir *datapath* que implemente um *pipeline*, que suporte as instruções que já consideramos anteriormente, isto é:

acesso à memória (**lw**) e *store word* (**sw**)

instruções **R-Format**, **add, sub, and, or** e **slt**

Instruções **addi** e **slli**

Branch if equal (**beq**)

Comparemos os tempos necessários à execução das 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, slli	2 ns	1 ns	2 ns		1 ns	6 ns

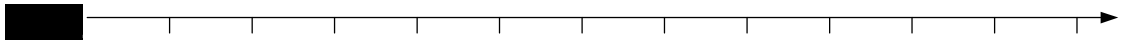
Pipelining

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 em 8 ns

Para verificarmos como comparar o tempo de execução de código por cada uma das soluções (*pipelined* e *single cycle*), observemos o exemplo do slide seguinte

Pipelining



Pipelining

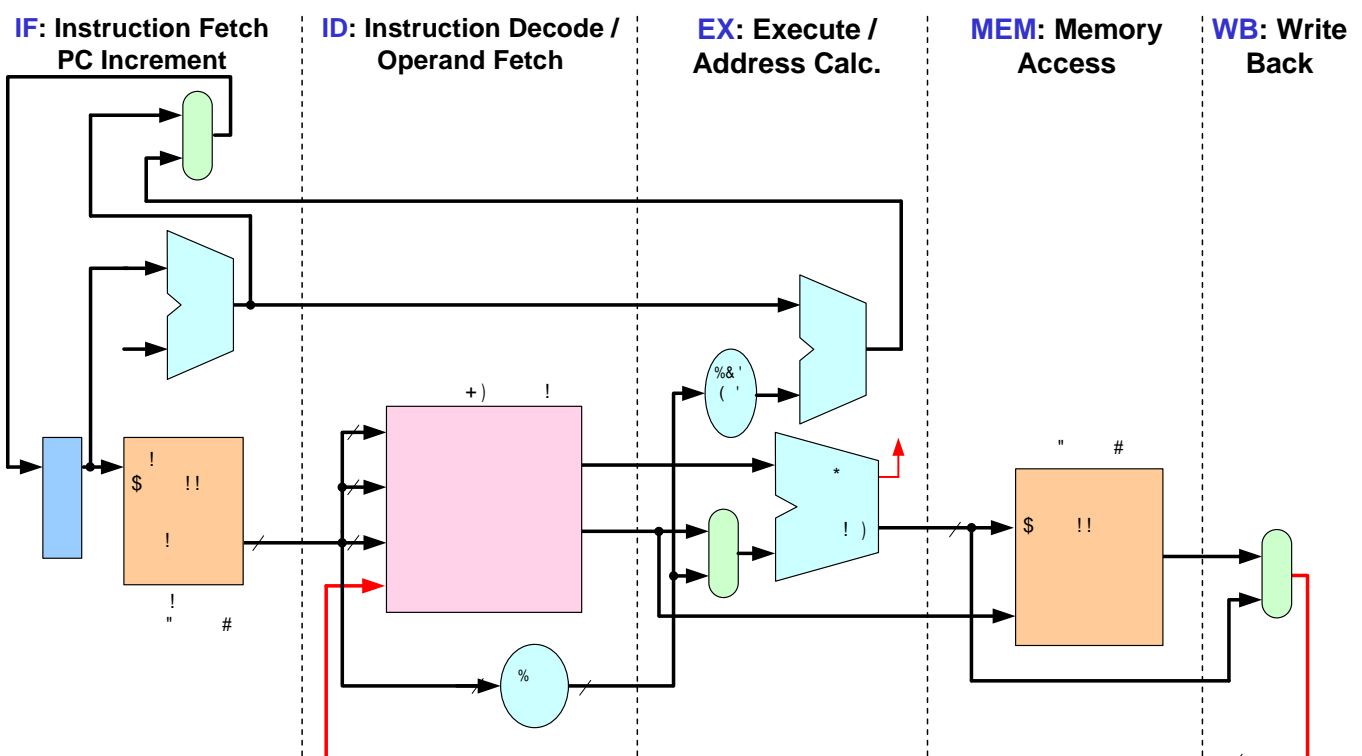
Observemos então o aspecto de uma solução para o MIPS, partindo do modelo do *datapath single-cycle*

A organização tenta retratar, como já vimos, as instruções consecutivas em que são decompostas as instruções:

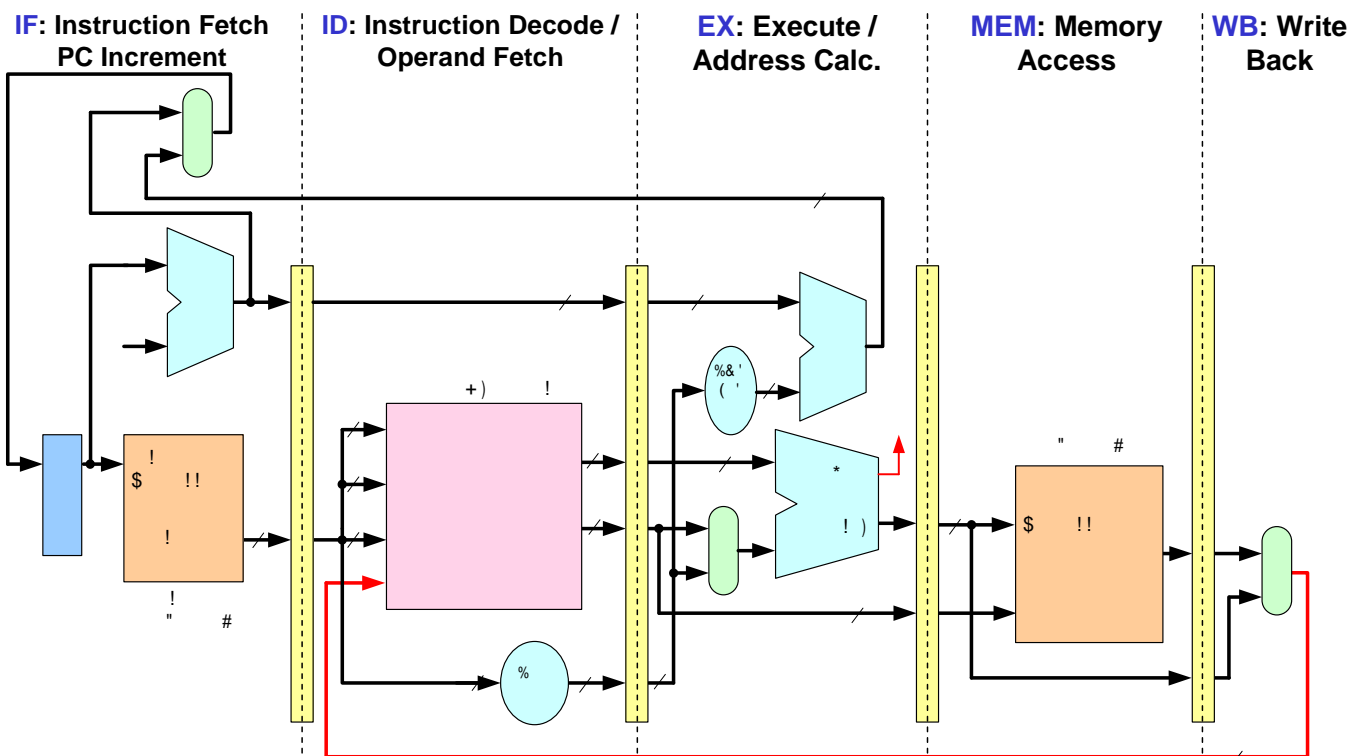
1. **(IF)** - *Instruction fetch* (ler a instrução da memória), incremento do PC
2. **(ID)** - *Operand fetch* (ler os registos) e decodificar a instrução (o *decoder* de instrução do MIPS permite que estas duas tarefas sejam executadas em paralelo)
3. **(EX)** - Executar a operação ou calcular um endereço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

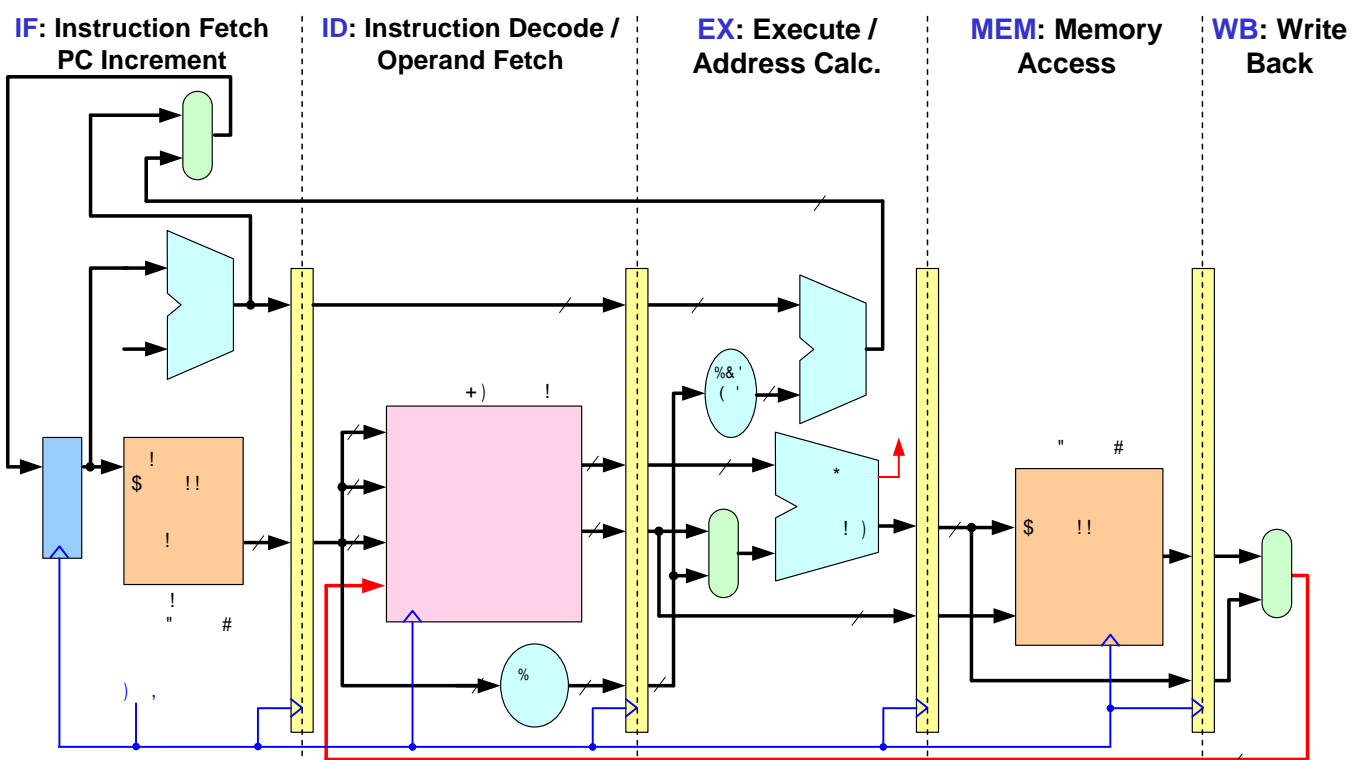
Pipelining: divisão em fases de execução



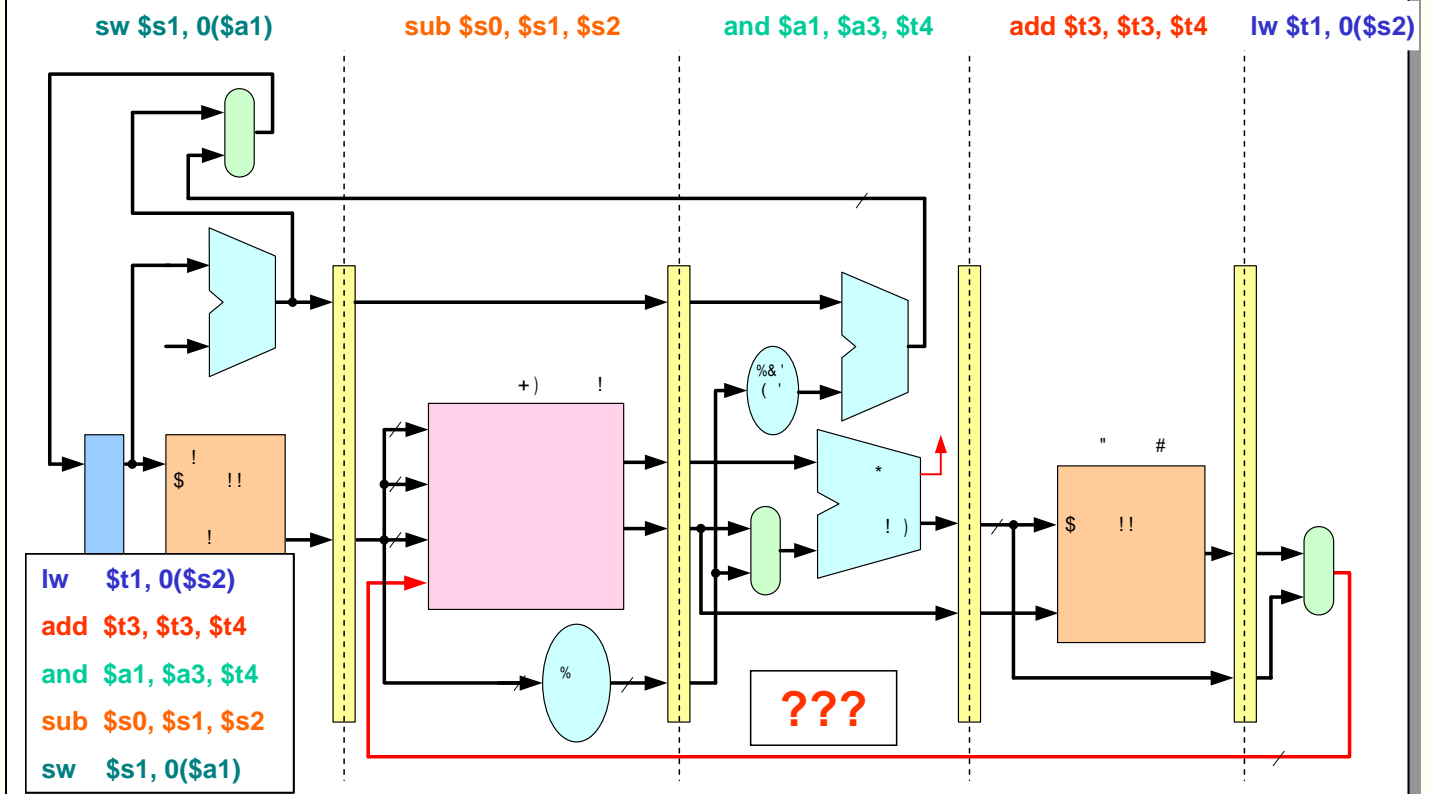
Pipelining: divisão em fases de execução



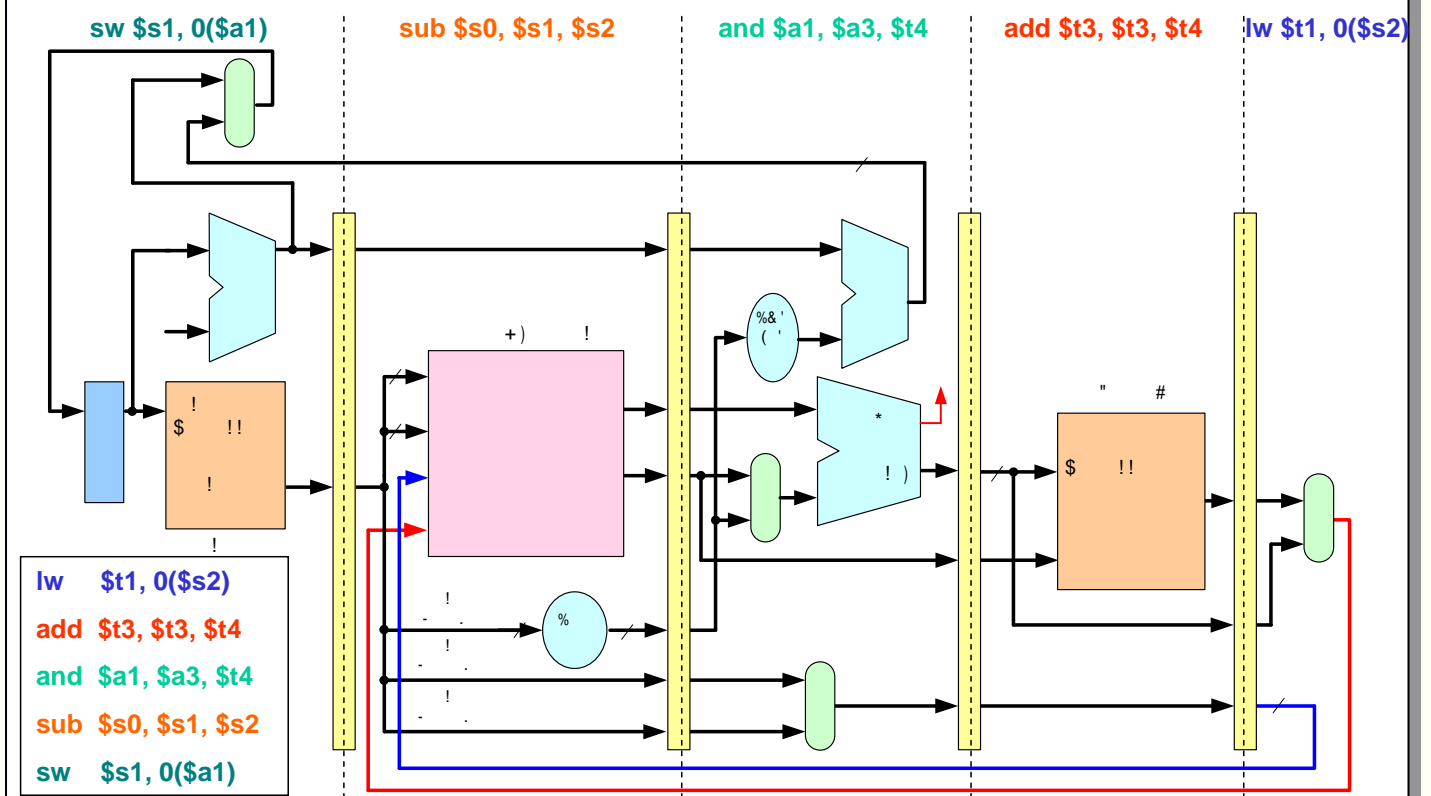
Pipelining: divisão em fases de execução



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 prosseguir 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 **escrever e ler** no mesmo registo que dispõe

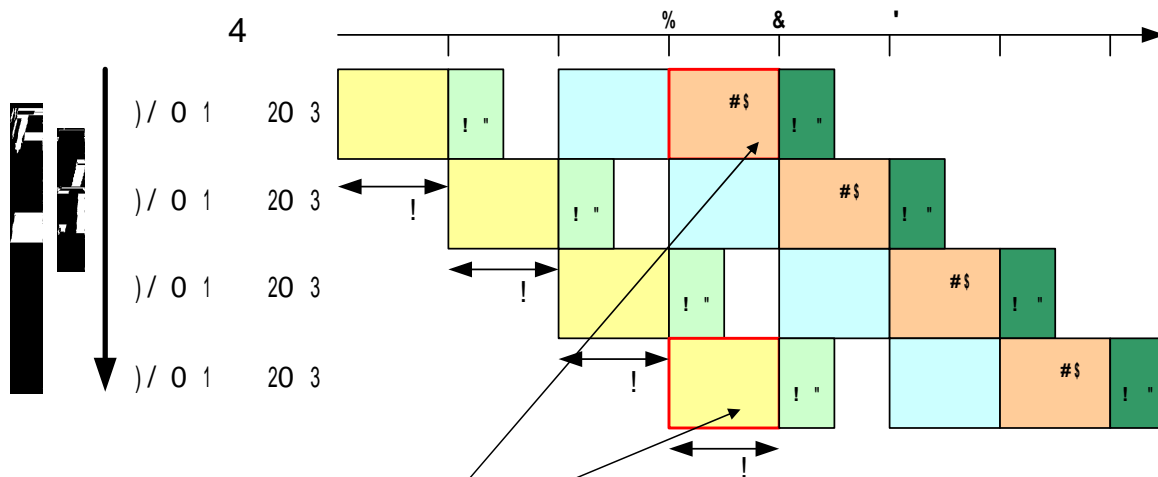
No caso do MIPS, e como já vimos, o *hardware* foi pensado para suportar directamente uma solução **pipelined** minimizando o potencial de ocorrência de **hazards** estruturais

Isso não evita, contudo, a necessidade de duplicar recursos

No caso da memória que, tal como já vimos, é **one-chip cache**, 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. 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.

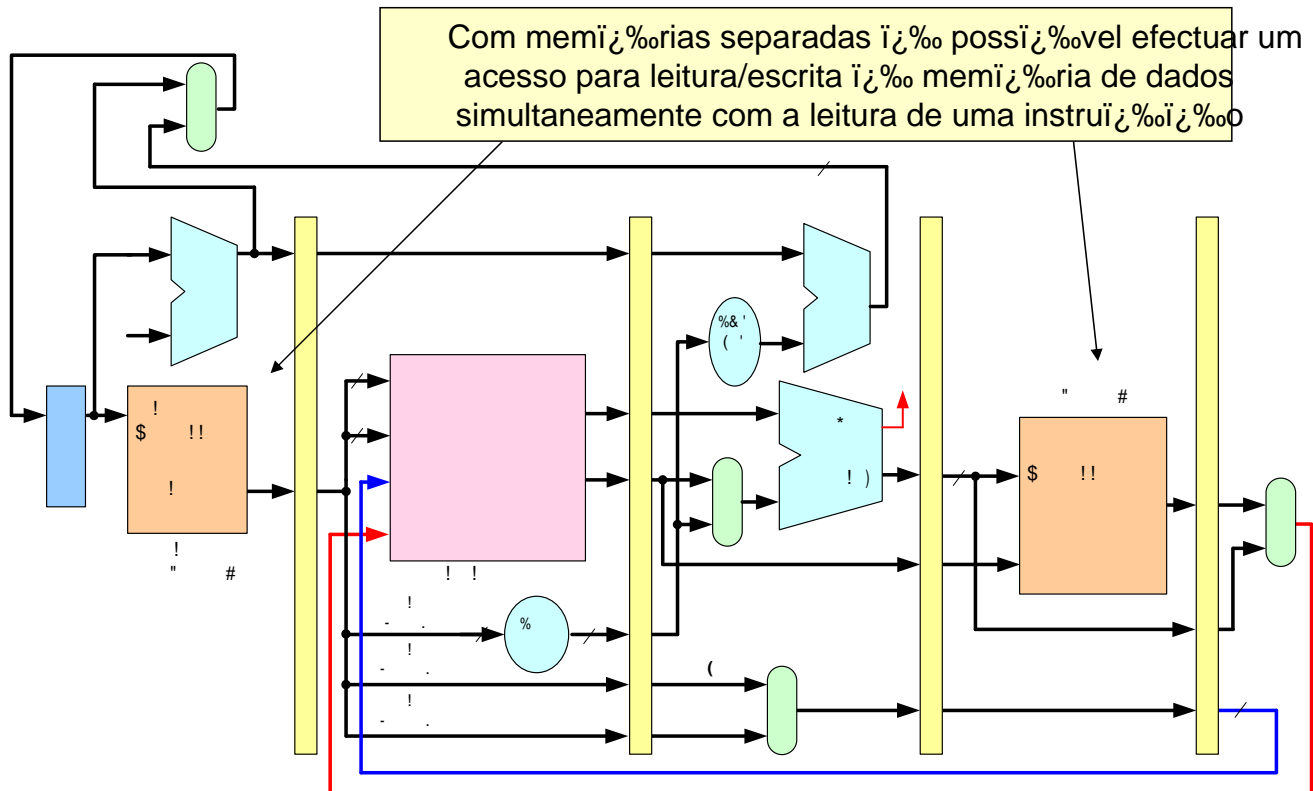
Pipelining: Hazards estruturais



No quarto estágio da primeira instrução, a quarta instrução necessita efectuar, simultaneamente, um acesso à memória para leitura de dados e para o *instruction fetch*

Como não existe a existência de memórias separadas determinando, neste caso, a ocorrência de **hazard estrutural**

Pipelining: Hazards estruturais



Pipelining: Hazards de controlo

Um **hazard** de controlo ocorre quando é necessário fazer **instruction fetch** de uma nova instrução e existe numa etapa mais avançada da **pipeline** uma instrução que pode alterar o fluxo de execução. Quando a instrução 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*):

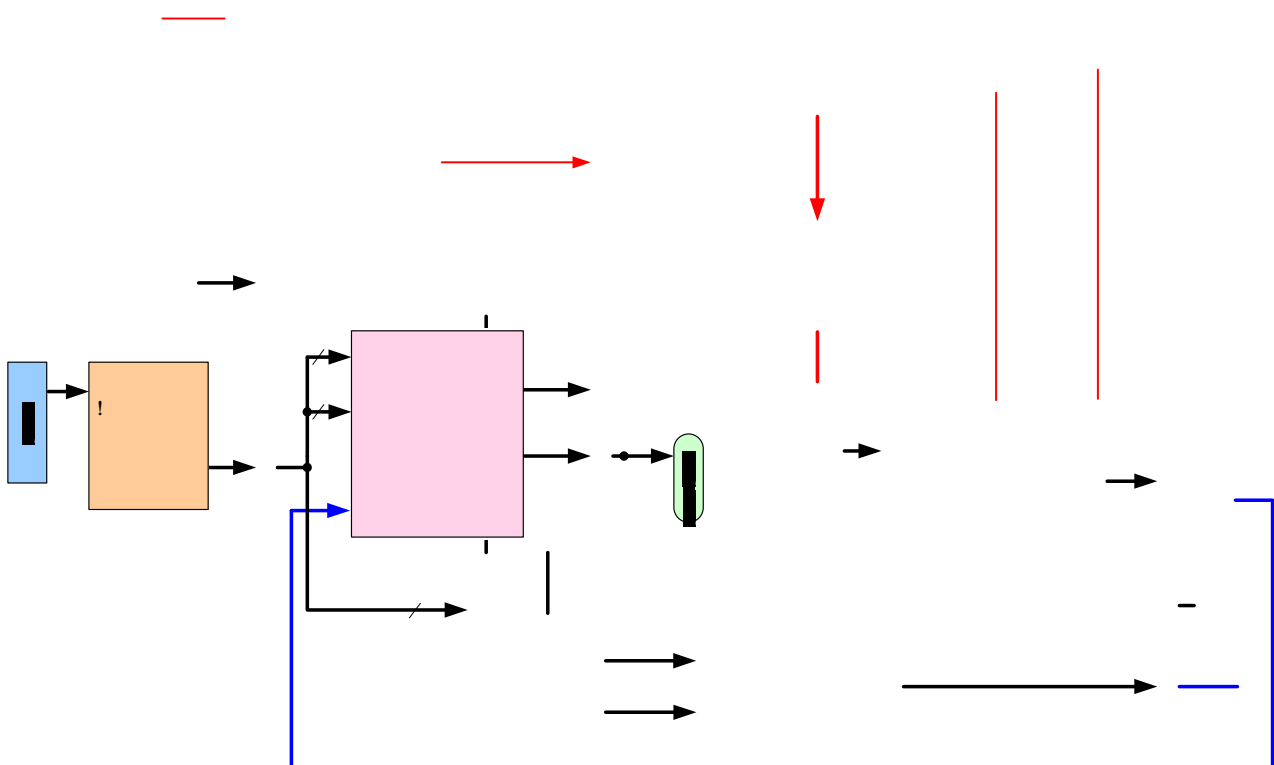
Mesmo admitindo que existe **hardware** dedicado para avaliar a condição do **branch** logo no 2º estágio, a unidade de controlo terá que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de código

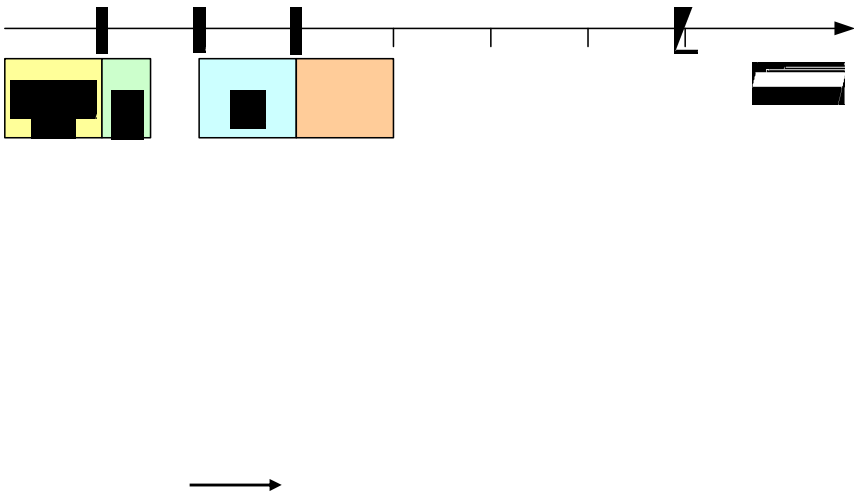
Assim, nos exemplos que se seguem, supõe-se que a **comparação dos operandos** é efectuada no 2º estágio através de **hardware** adicional

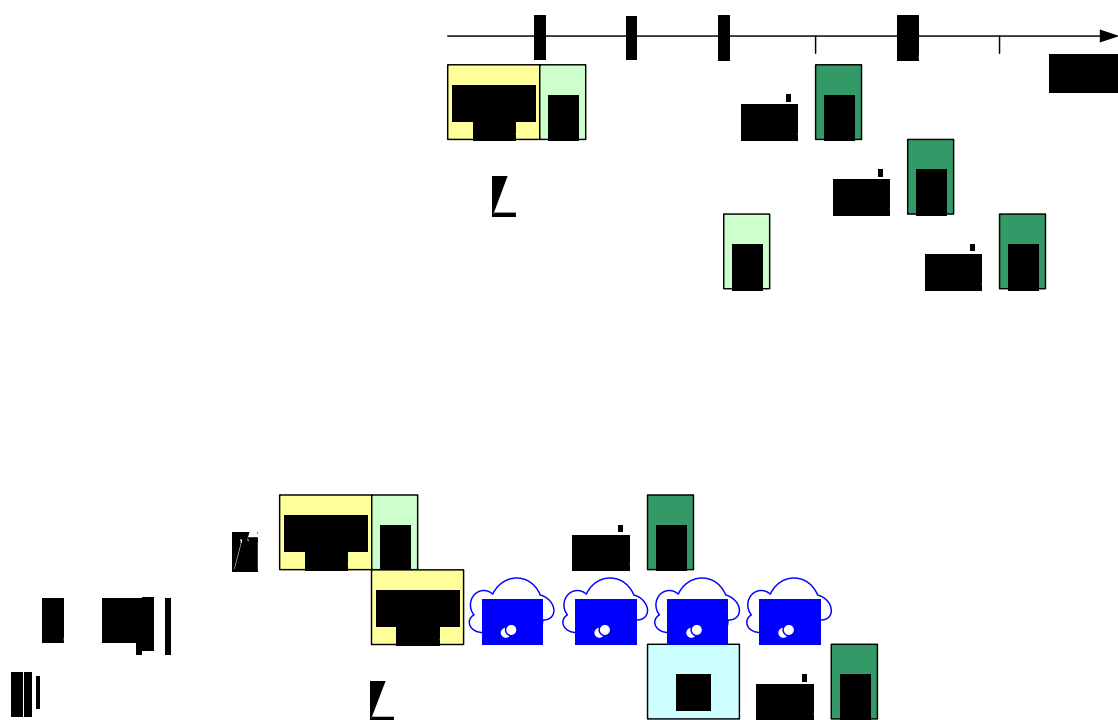
Do mesmo modo, o **calculo Branch Target Address** passa também a ser efectuada no 2º estágio no 3º estágio

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 lavagem da roupa da primeira carga é possível observar o resultado e determinar se é necessário reajustar o detergente e a temperatura.

Pipelining: Datapath com unidade de controlo







Pipelining: Hazards de controlo - *prediction*

As técnicas de predição usadas correntemente são mais elaboradas. Podem, por exemplo, adoptar estratégias baseadas em este tipo de programas:

Se a *branch* for para um endereço anterior ao caso em que se encontra a *branch*, antecipar sempre que a condição é verdadeira e a *branch* é tomada. Isto tem como consequência que a instrução que pipeline a residente no endereço alvo *branch*

Se a *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 da contagem de cada *branch* ao longo da execução do programa

Pipelining: Hazards de controlo **Delayed branch**

Uma terceira alternativa para resolver hazards de controlo designada por **delayed branch**

Nesta abordagem, o processador executa sempre a instrução que se segue ao *branch*

Esta técnica é escondida do utilizador

organiza as instruções por forma a trocar a instrução com a anterior, desde que esta não seja afectada pelo *branch*

Quando não é possível efectuar a troca, insere-se uma **NOE (no operation; ex.: `sll, $0, $0, 0`)**

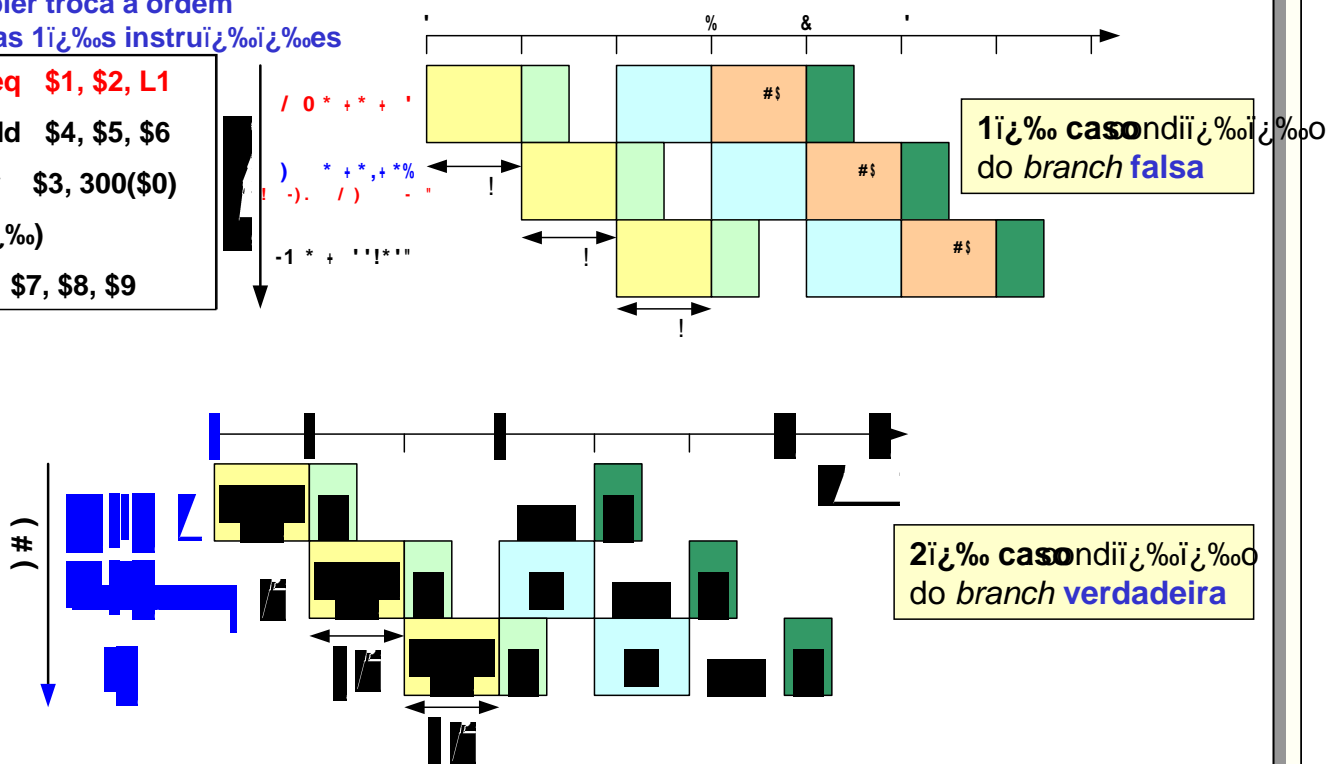
Esta é a solução adoptada pelo MIPS

Pipelining: Hazards de controlo **Delayed branch**

Assembler troca a ordem das duas primeiras instruções

```

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



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 na **pipeline**.

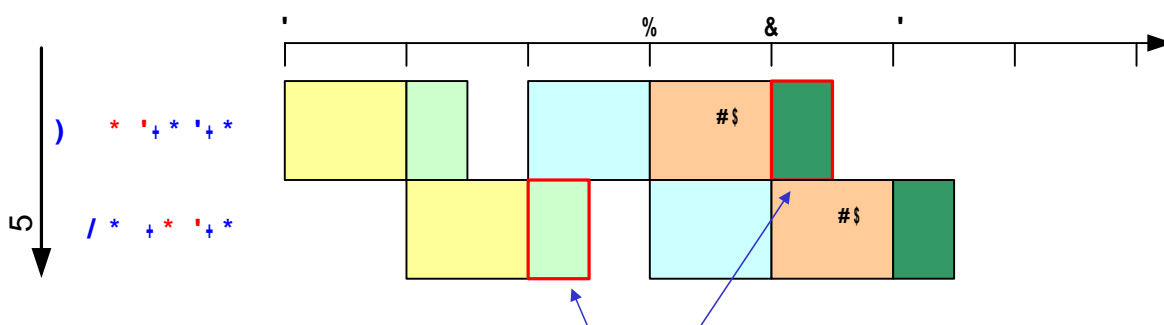
Se o resultado que vai ser necessário para a instrução que vem atrás no **pipeline** ainda não tiver sido armazenado, então essa instrução não poderá prosseguir porque irá tomar como operando um valor incorrecto (desactualizado).

No exemplo do tratamento da roupa, um hazard de dados corresponde a uma situação em que numa carga de roupa apenas estivesse presente uma peça de cada par. Enquanto não forem lavadas as peças que emparelham 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 subtração não pode ser executada se o conteúdo de **\$s0** ser calculado e armazenado pela instrução anterior, pois necessita do seu valor, mas só vai ser escrito no registo destino em **10**).

Pipelining: Hazards de dados

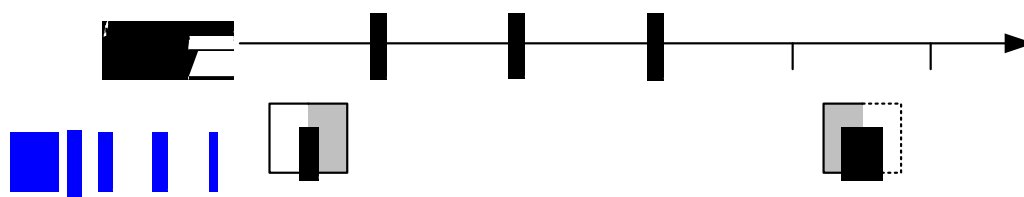
A principal solução para a resolução de **hazards de dados** resulta da observação de que não é necessário esperar pela primeira instrução para tentar resolver o **hazard**.

A partir do momento em que a operação da instrução seja executada (o que acontece na ALU no terceiro estágio), o **resultado** pode ser disponibilizado para a instrução seguinte.

Esta técnica de disponibilizar um resultado para a 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, começamos por apresentar uma versão gráfica simplificada da cadeia de **pipelining**.

Pipelining: Hazards de dados



Nesta representação gráfica usamos símbolos para representar recursos físicos:

• O **quadrado azul** corresponde ao estágio de **instruction fetch**, representando o quadrado a memória de instrução.

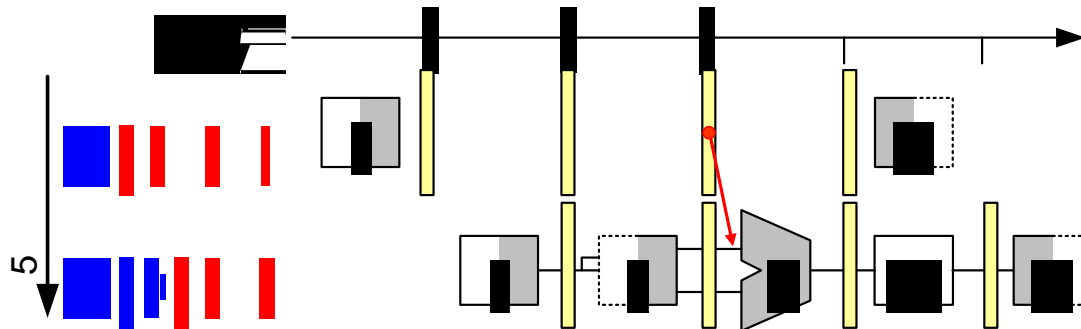
• A metade cinzenta à direita tipifica uma operação de **leitura**.

• 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 a operação de **escrita** no elemento de estado respectivo (WB).

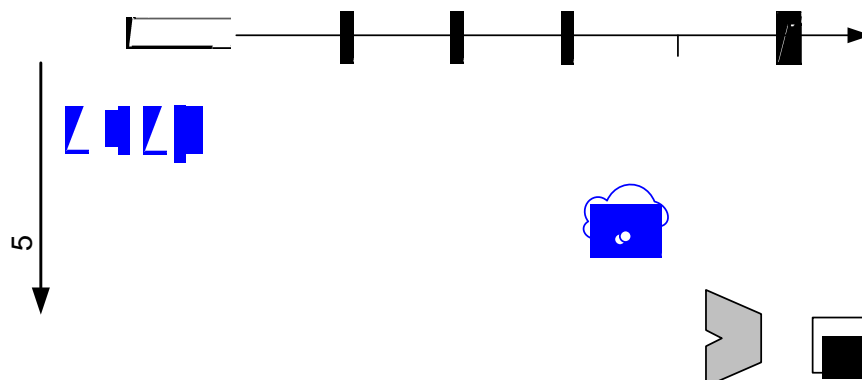
Pipelining: Hazards de dados

Usando esta representação de pipeline, podemos voltar a observar a sequência de instruções que vimos ser responsáveis pela situação de *hazard de dados*



Como se pode observar, **forwarding** do resultado sai da ALU na instrução *addl* para a entrada do estágio EX da instrução *subl*, resolve o *hazard de dados*

Esta técnica só funciona, contudo, se for efectuado para um estágio da instrução subsequente que ainda não coincide (relação causal)



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

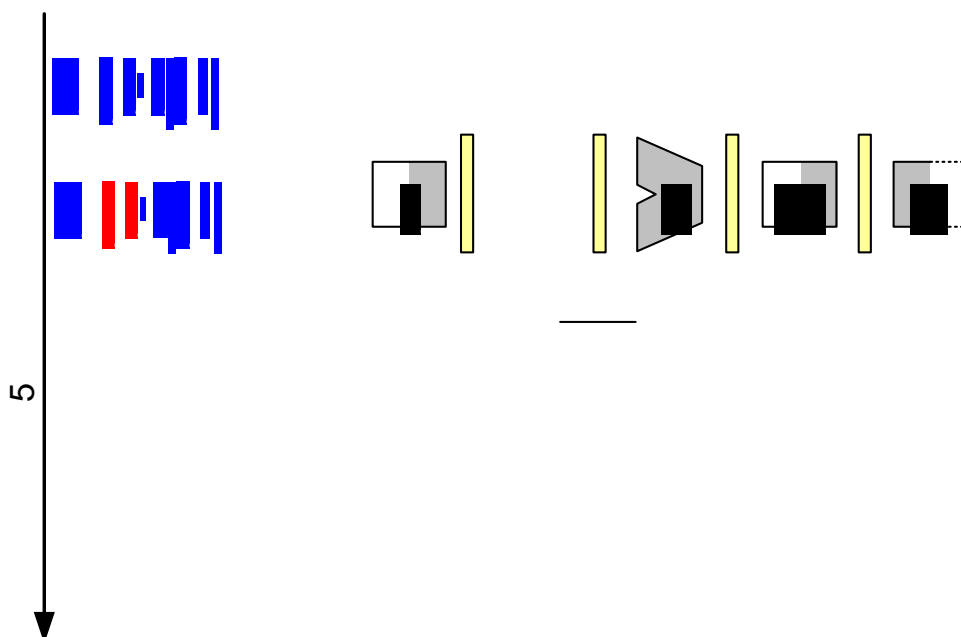
Exemplo:

```
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





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)

Pipelining: Hazards de dados

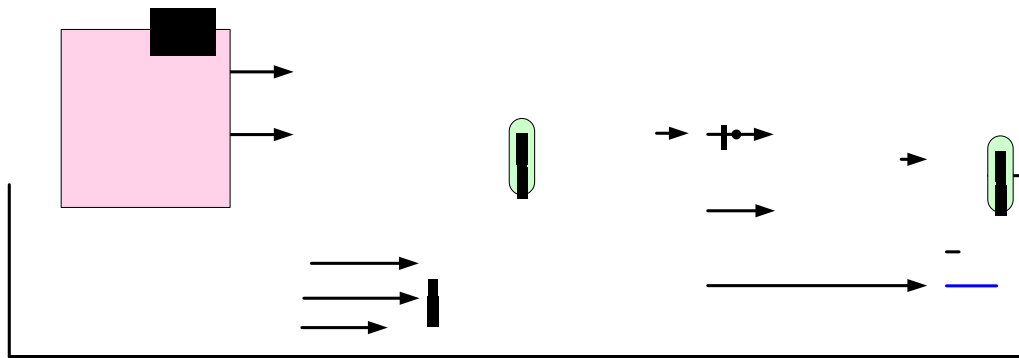
As situações, correspondentes a hazards de dados, em que há necessidade de encaminhar valores para a fase EX são:

- 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



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 hazards de dados 1 e 2 podem ser detectadas através de:

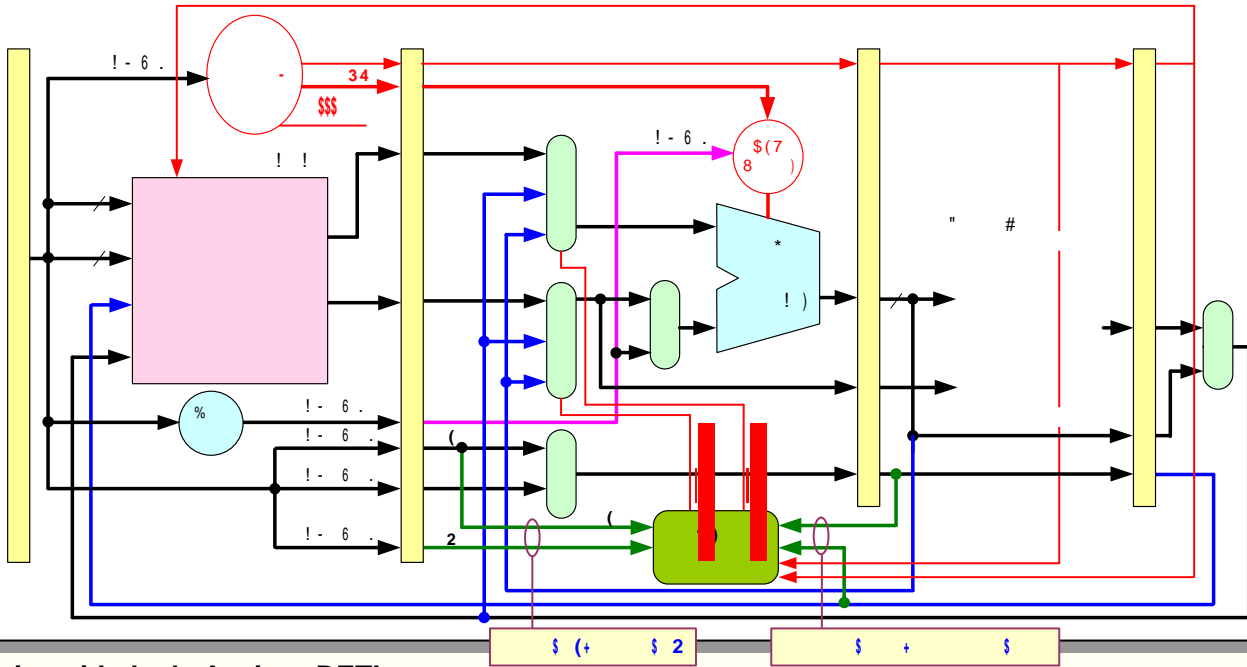
1) $EX/MEM.RDD == ID/EX.RS$ ($EX/MEM.RDD = \$2$, $ID/EX.RS = \$2$)

2) $MEM/WB.RDD == ID/EX.RT$ ($MEM/WB.RDD = \$2$, $ID/EX.RT = \$2$)

A situação 3 pode ser resolvida através de forwarding (não se considera hazard)

A situação 4 também não corrige os dados

Pipelining: *Hazards* 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

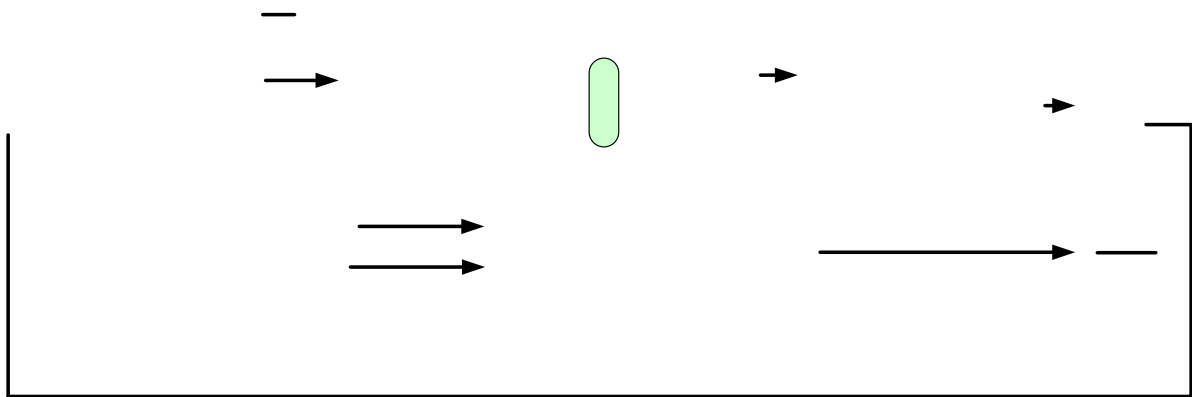
if((EX/MEM.RegWrite == 1) and (EX/MEM.RDD == ID/EX.RS)) Forw_A = 10

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

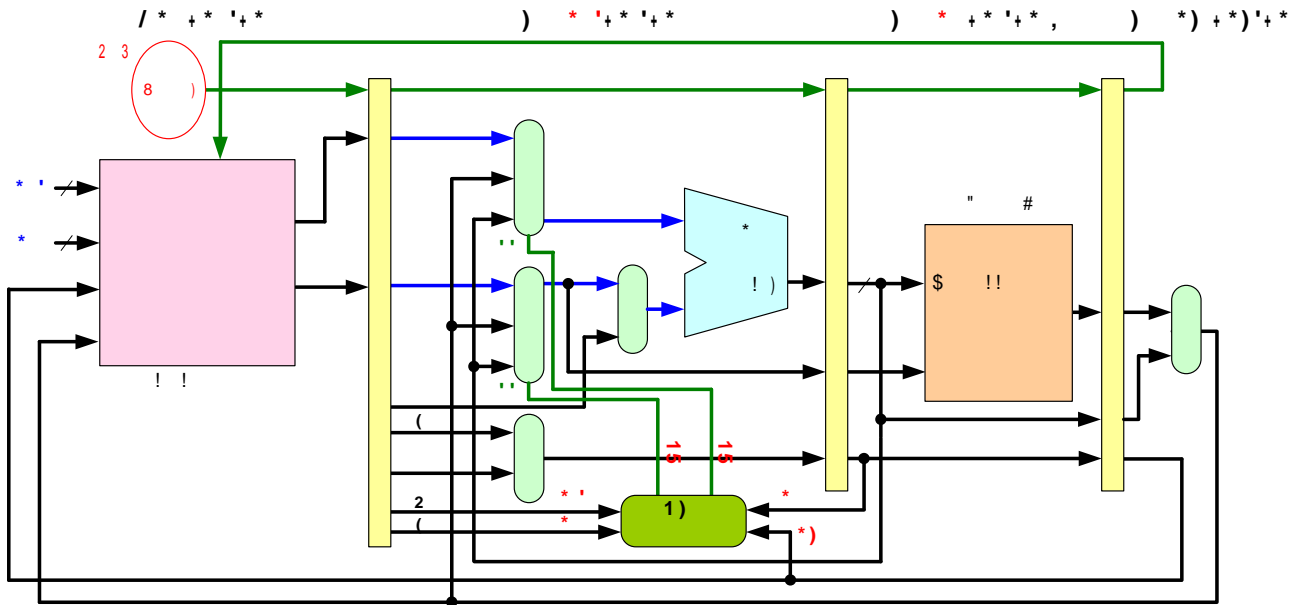
if((EX/MEM.RegWrite == 1) and (EX/MEM.RDD == ID/EX.RS)) Forw_A = 10

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)
...

```



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

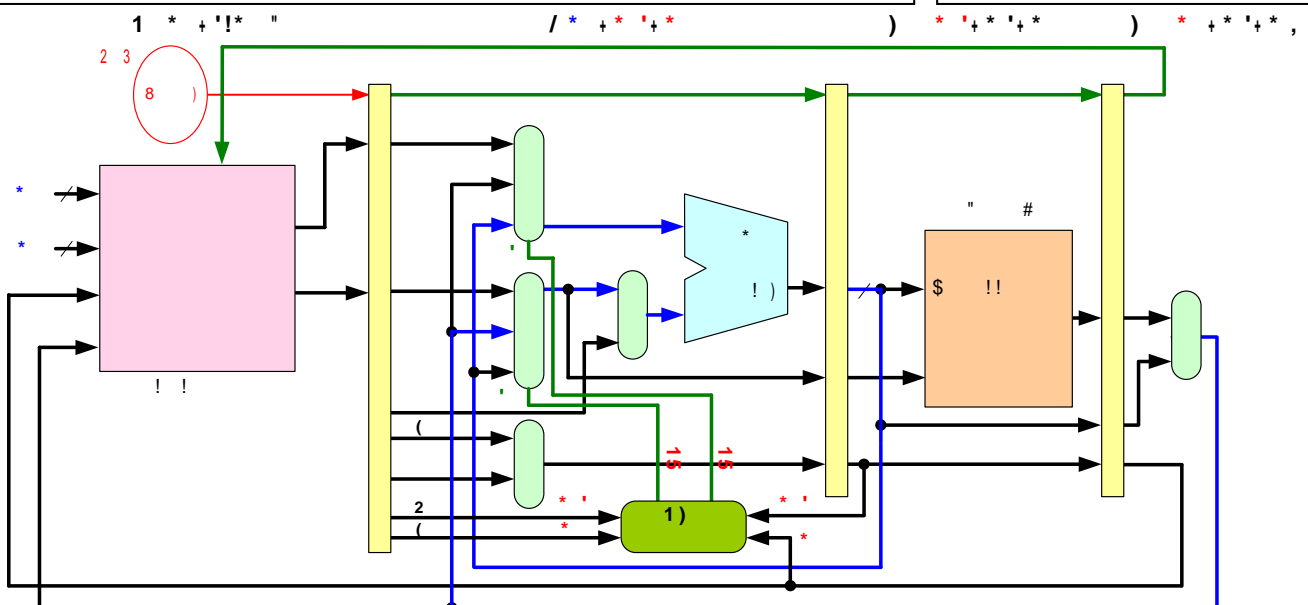
if((EX/MEM.RegWrite == 1) and (EX/MEM.RDD == ID/EX.RS)) Forw_A = 10

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)
...

```

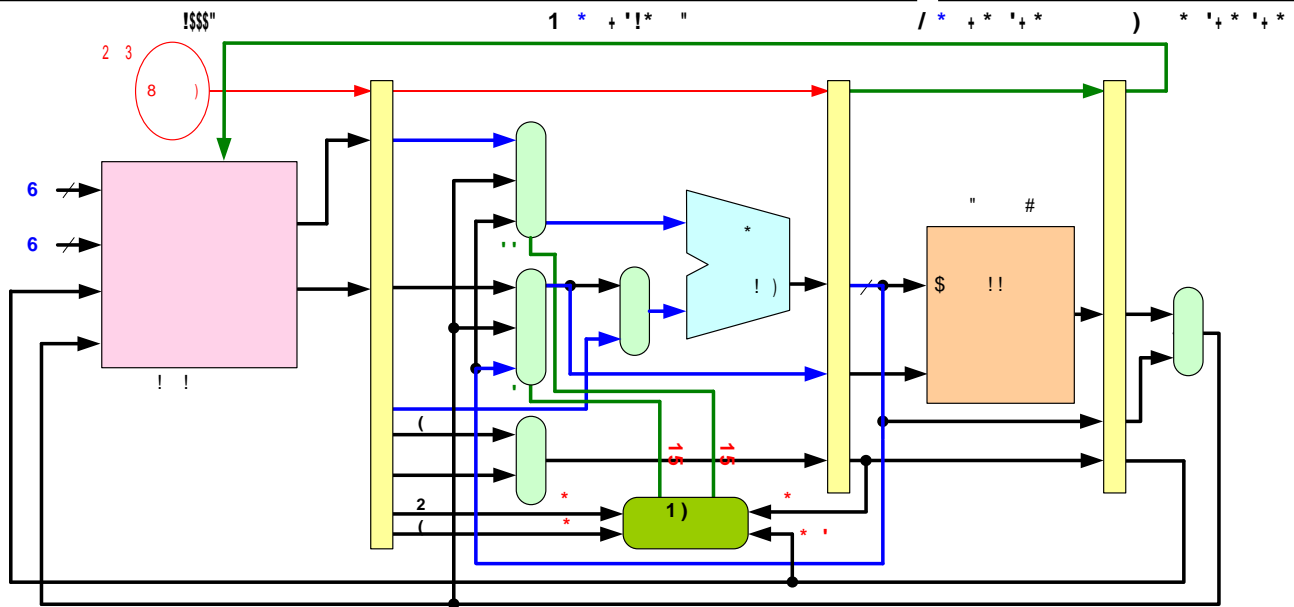


Pipelining: Hazards de dados (exemplo de forwarding)

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

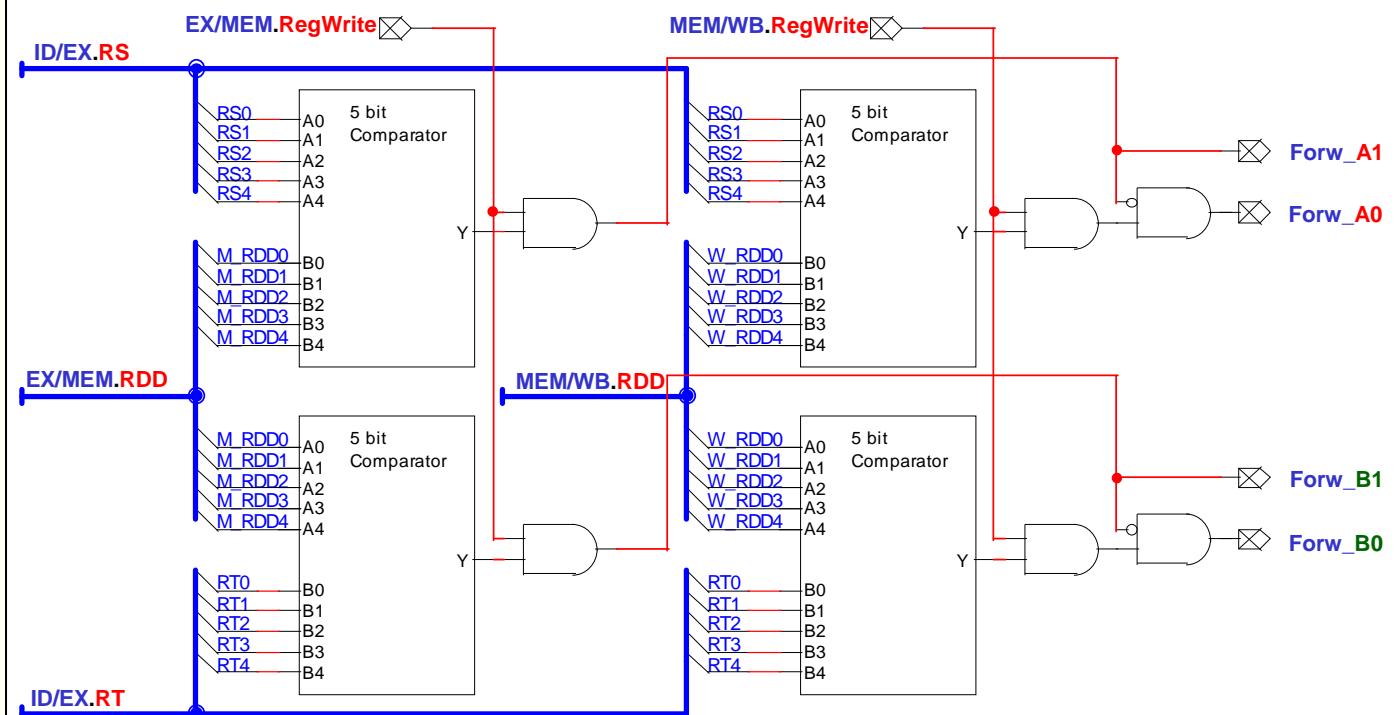
```
...
and $a3, $a0, $t1
add $t3, $t0, $t5
add $s0, $t0, $t1
sub $t2, $s0, $t3
sw $t2, 0($s1)
...
```



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

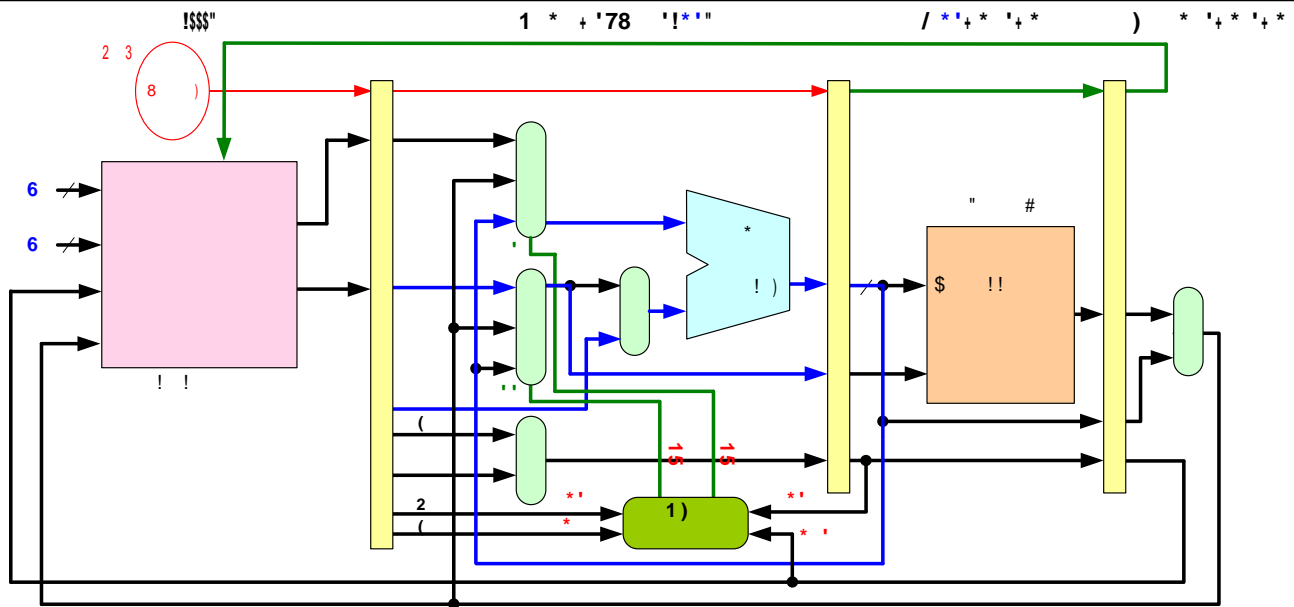


Pipelining: Hazards de dados

Problema:

¿O que acontece nestes *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

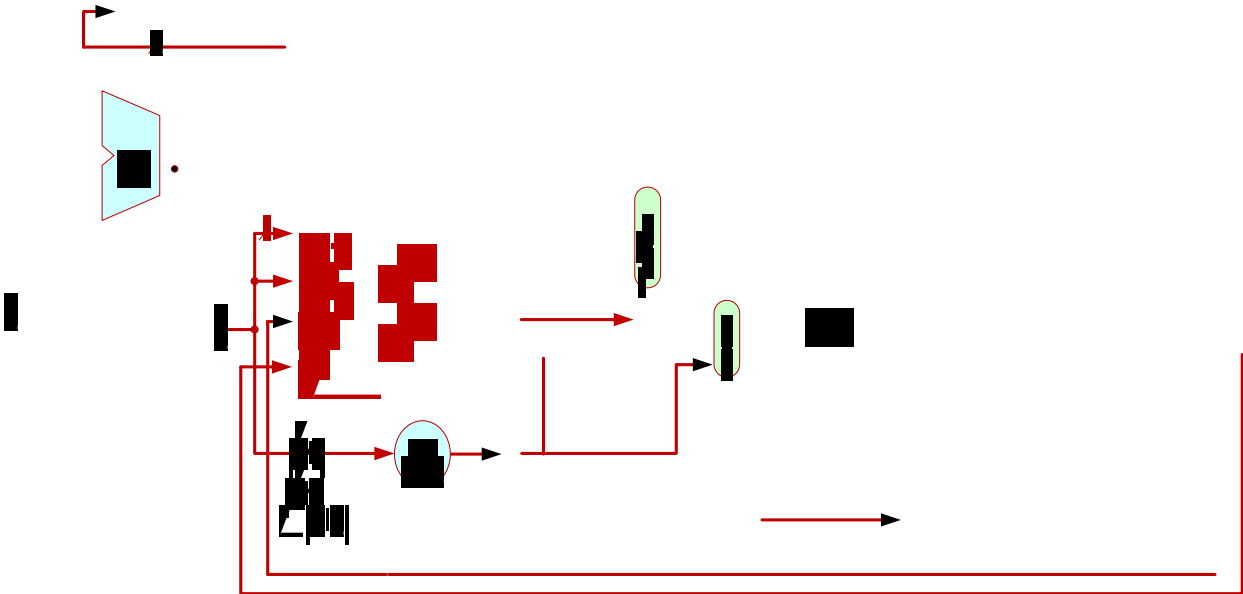
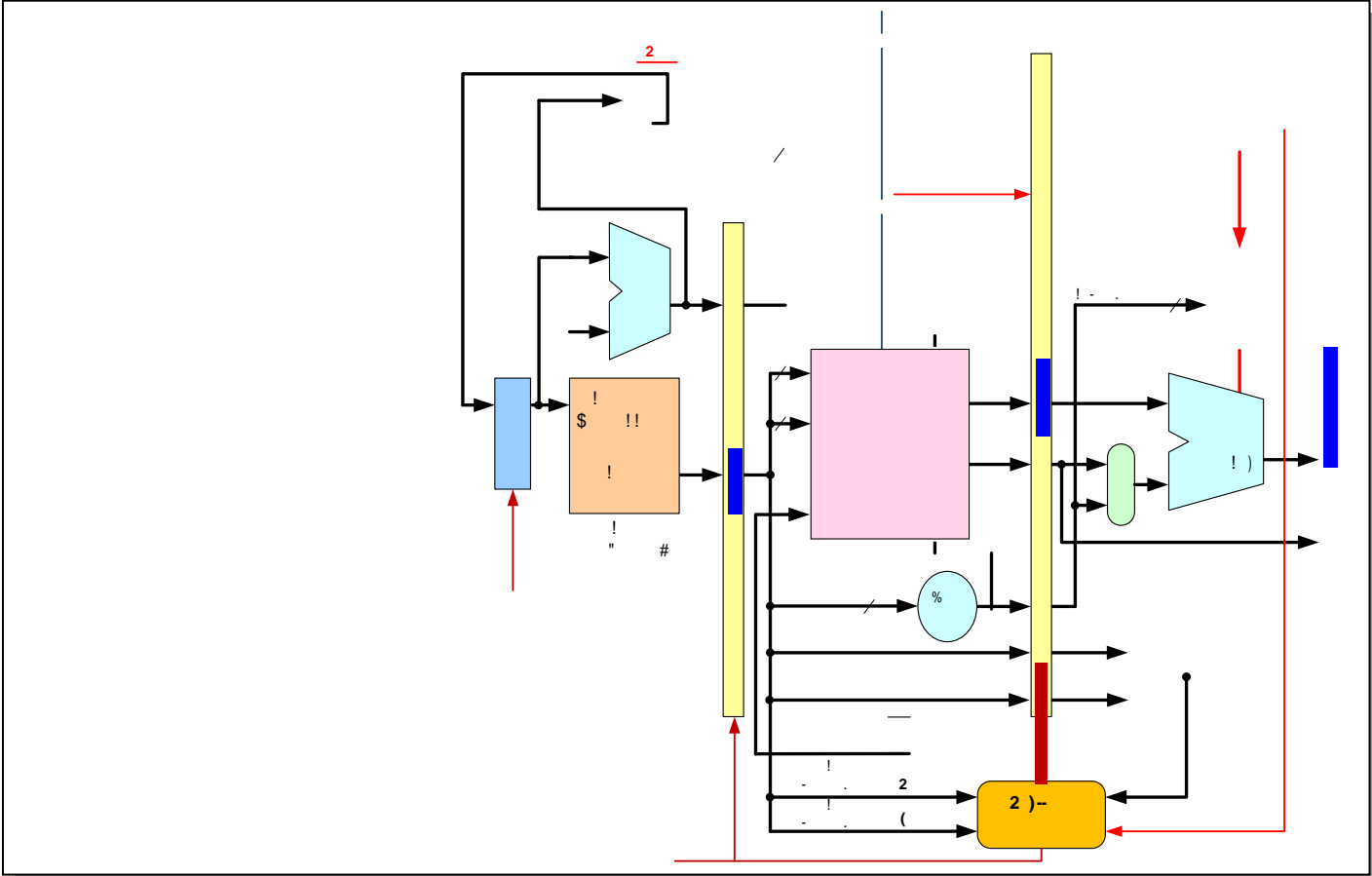
Como já observado anteriormente, um exemplo em que *forwarding* não impede a ocorrência de *stalling* é o que resulta de uma instrução aritmética a seguir e na dependência de uma instrução *load* de

lw \$s0, 20(\$t1)

sub \$t2, \$s0, \$t3

add \$t3, \$t3, \$t2





E agora?

il grand finale

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

