

Arquitectura de Computadores II



3º Ano

Pipelining (execução encadeada de instruções)

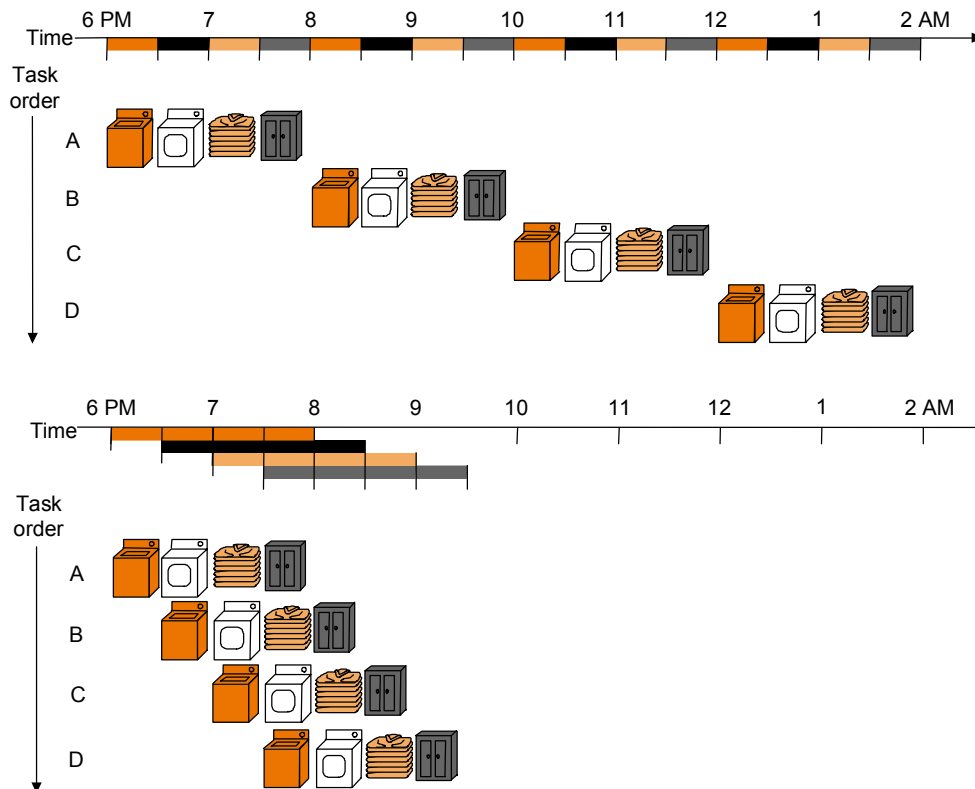
João Luís Ferreira Sobral
Departamento de Informática
Universidade do Minho



Abril 2005

Visão global de *Pipelining*

- Técnica que permite sobrepor a execução de várias instruções

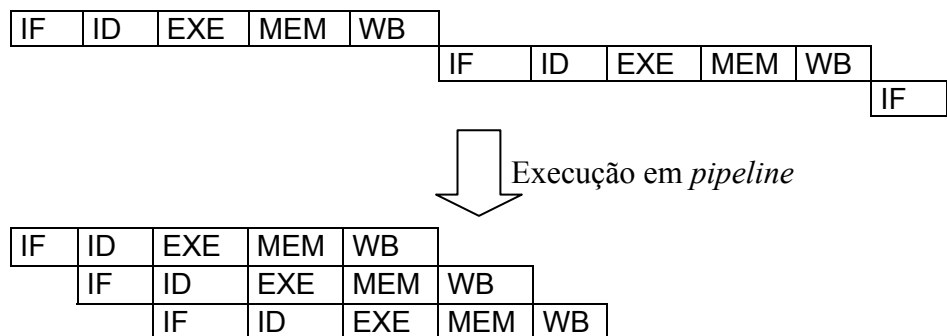


- Cada fase (estágio) da *pipeline* executa de forma concorrente, como numa linha de montagem

Pipelining em MIPS

- Fases de execução das instruções em MIPS:

- Busca da instrução (IF)
- Leitura dos registos e decodificação das instruções (ID)
- Execução da operação ou cálculo de endereço (EXE)
- Acesso ao operando em memória (MEM)
- Escrita do resultado em registo (WB)



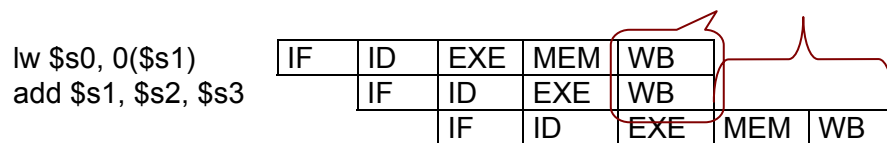
Visão global de *Pipelining*

● Desempenho do processador com *pipelining*

- Duração de cada ciclo de execução em *pipeline* é igual à duração do estágio mais longo, adicionando a sobrecarga da *pipeline* (passagem de informação entre estágios)

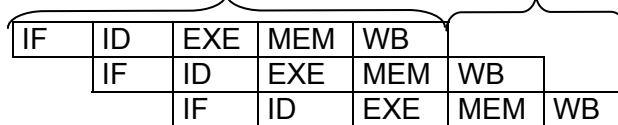
$$T_{cc\ pipeline} = \max [\text{estágio}_i] + \text{sobrecarga de } pipeline$$

- **A execução em *pipeline* requer que as várias fases estejam balanceadas (i.e., demorem o mesmo tempo a executar):**



- O tempo (ideal) de execução com *pipelining*:

$$T_{exec\ pl} = [\#estágios + (\#instruções - 1)] \times T_{cc}$$



- ***Pipelining* aumenta o débito (instruções realizadas por unidade de tempo) mas não diminui o tempo necessário para cada instrução!**

- O ganho é **potencialmente** igual ao número de estágios da *pipeline*

$$\text{ganho} = \frac{T_{exec\ Sc}}{T_{exec\ Pl}} = \frac{\#instruções \times \#estágios \times T_{cc}}{[\#estágios + (\#instruções - 1)] \times T_{cc}} \underset{\#instruções \rightarrow \infty}{=} \#estágios$$

- O ganho pode ser interpretado de forma diferente, consoante a base de comparação seja uma implementação *single-cycle* ou multi-ciclo

base *single-cycle*

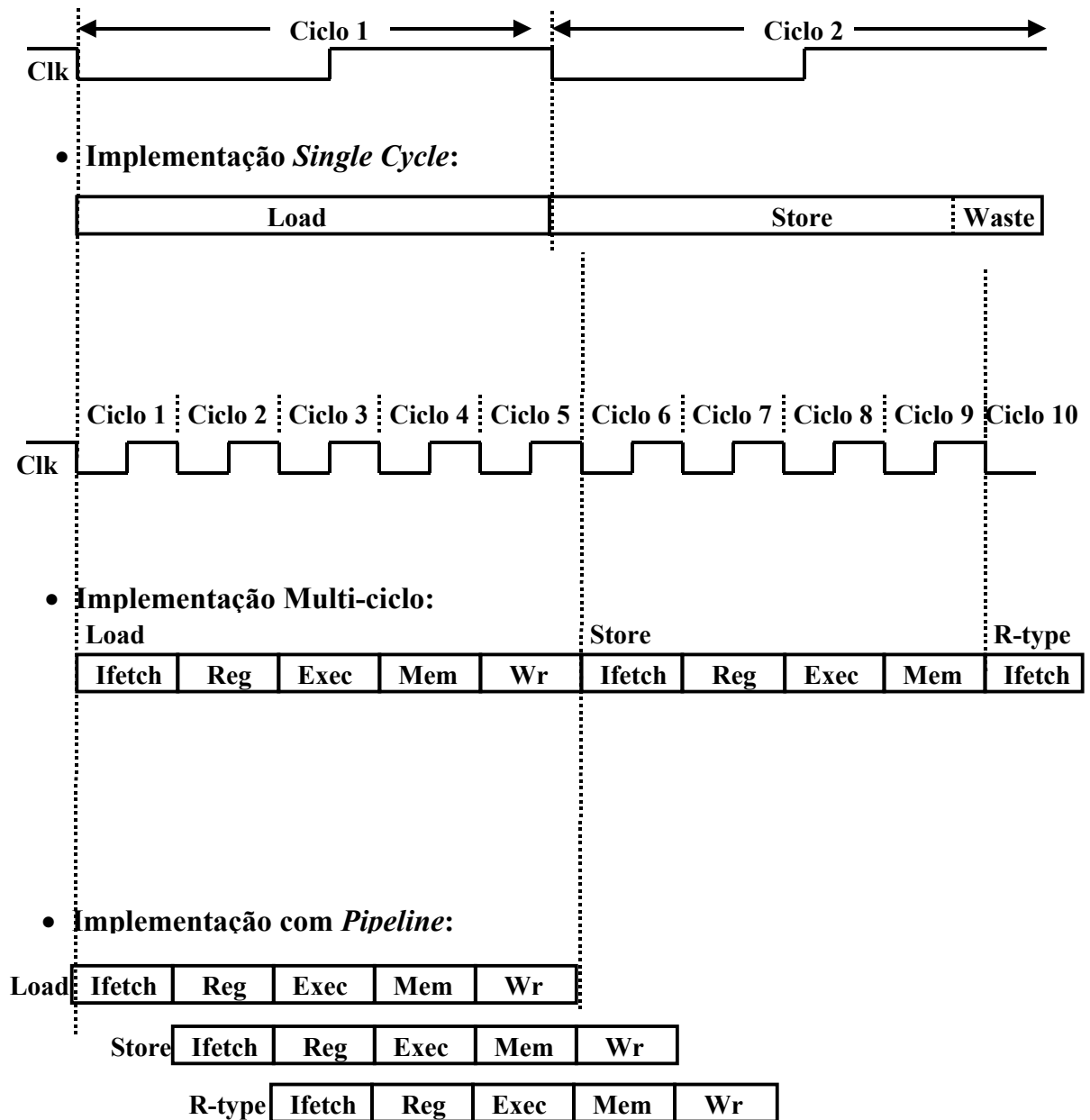
$$T_{cc\ pipeline} = T_{cc\ single-cycle} / \# \text{estágios}$$

base multi-ciclo

$$CPI_{pipeline} = CPI_{multi-ciclo} / \# \text{estágios da } pipeline$$

Visão global de *Pipelining*

- *Single-cycle versus Multi-ciclo versus Pipelining*



Visão global de *Pipelining*

● Exemplo de desempenho do processador MIPS com *pipelining*

- Tempos acesso às unidades funcionais:

Memória (Leitura ou escrita)	ALU	Banco de registos
2 nanosegundos	2 nanosegundos	1 nanosegundo

- Qual a diferença de desempenho entre uma implementação com *pipelining* e uma implementação *single-cycle* com ciclo de duração fixa?

- Duração de cada instrução:

Tipo de Instrução	Busca da instrução	Leitura de registos	Operação na ALU	Acesso à Memória	Escrita em registo	Total
Tipo R	2	1	2		1	6 ns
<i>Load</i>	2	1	2	2	1	8 ns
<i>Store</i>	2	1	2	2		7 ns
<i>Branch</i>	2	1	2			5 ns

- $CPI_{single-cycle} = CPI_{pipeline} = 1$.

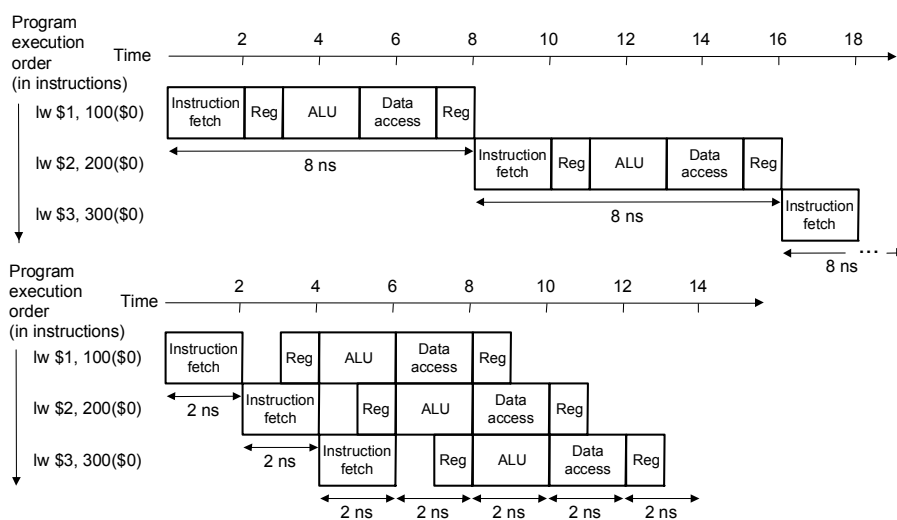
- Duração do ciclo na versão *single-cycle* com o ciclo de relógio fixo:

T_{csc} = duração da instrução mais longa = *load* = 8 ns

- Duração do período de relógio com *pipeline*, assumindo 5 estágios:

T_{cppe} = duração do estágio mais longo = 2 ns

- Ganho (máx.) = $T_{excsc} / T_{excpipe} = T_{csc} / T_{cppe} = 8 \text{ ns} / 2 \text{ ns} = 4,0x$



- Ganho na execução de 3 instruções = $24 / 14 = 1,74x$

- Ganho na execução de 1000 instruções = $8000 \text{ ns} / 2008 \text{ ns} = 3,98x$

Visão global de *Pipelining*

● Projectar o conjunto de instruções para execução em *pipelining*

- **Instruções de dimensão fixa** facilitam a busca e a descodificação de instruções
- **Poucos formatos de instruções** com os registos fonte especificados sempre nos mesmos campos permitem o paralelismo entre a descodificação e a leitura do valor dos registos
- **Arquitectura do tipo *load-store***, onde os operandos em memória só aparecem em *load* e *store* facilitam o balanceamento dos estágios da *pipeline*
- **Operandos alinhados em memória** tornam os acessos à memória mais rápidos

● Anomalias na execução em *pipelining* (*hazards*)

- **Impedem que a *pipeline* prossiga a execução normal, originando um aumento do CPI médio (para valores superiores a 1).**
- **Estruturais** – o hardware não suporta a combinação de instruções em execução (ex. são necessárias duas unidades para acesso à memória, uma para a busca das instruções outra para os acessos a dados em memória)

Solução => duplicar unidades funcionais ou portas de acesso

- **Controlo** – a execução da instrução seguinte depende de uma decisão baseada num resultado anterior (ex. saltos condicionais)

Soluções => 1. empatar a *pipeline*

2. prever o salto

3. retardar o salto

- **Dados** – a execução da instrução seguinte depende de dados produzidos por instruções anteriores.

Solução => 1. delegar no compilador

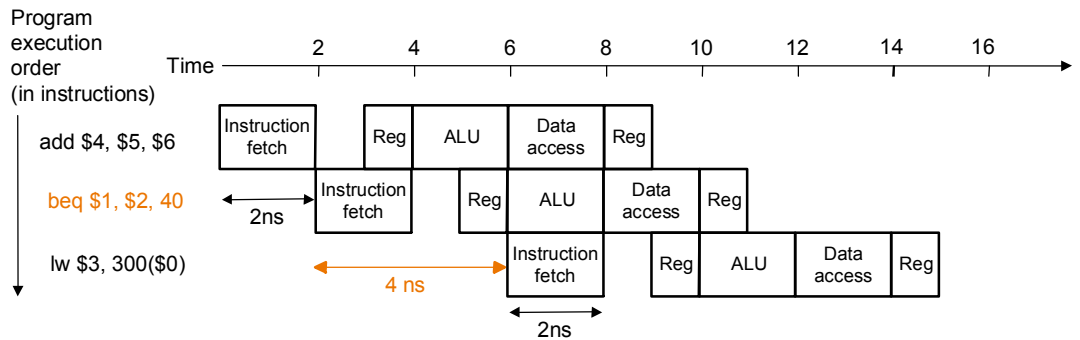
2. encaminhar os dados entre unidades

3. empatar a *pipeline*

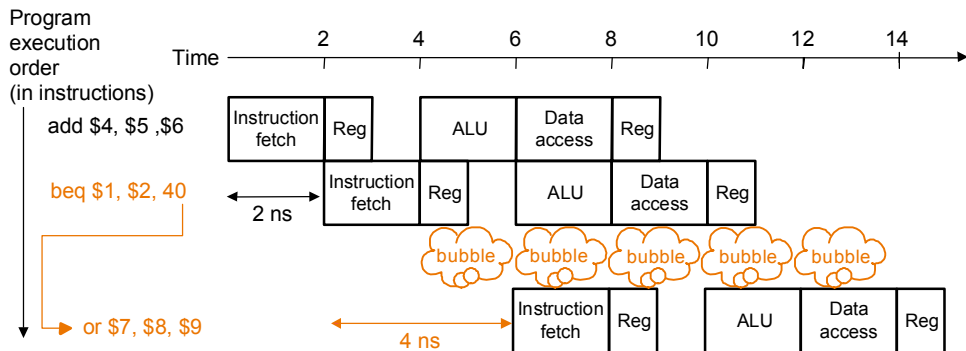
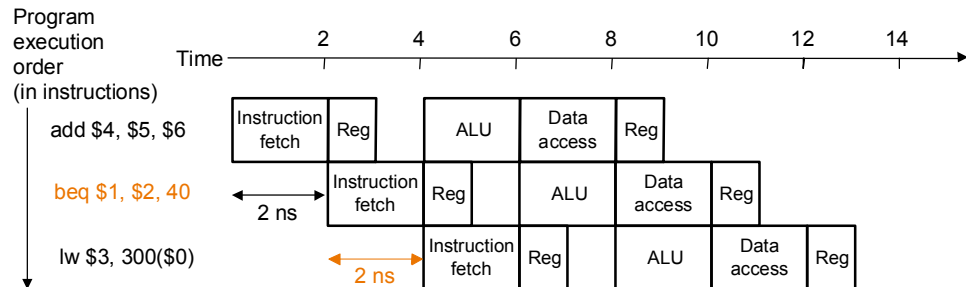
Visão global de *Pipelining*

Exemplos de anomalias de controlo na execução em *pipeline*

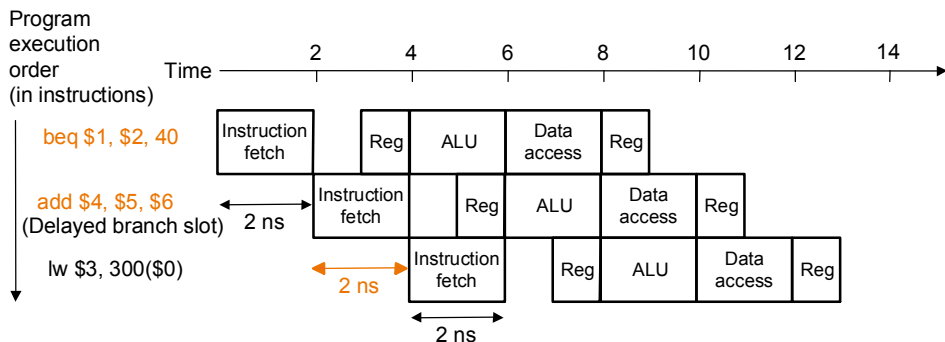
- Empatar a execução de um salto



- Prever o salto como não tomado



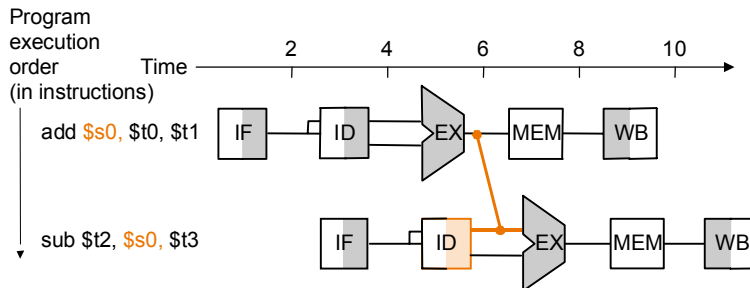
- Execução retardada do salto (actualmente menos utilizada)



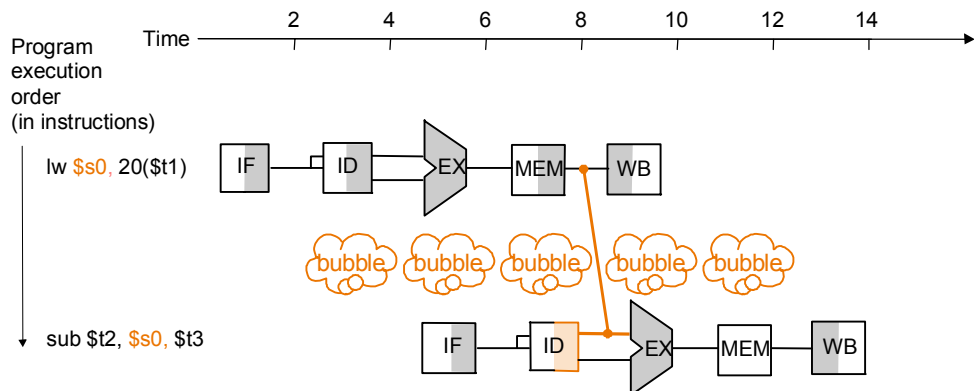
Visão global de *Pipelining*

Exemplos de anomalias de dados na execução em *pipeline*

- Encaminhar os dados para resolver a dependência



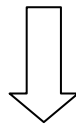
- Limitação no encaminhamento dos dados



- Reordenação das instruções para resolver a dependência de dados

```

lw $t0, 0($t1)      # reg $t1 contém &v[k]
                    # $t0 (temp) = v[k]
lw $t2, 4($t1)     # $t2 = v[k+1]
sw $t2, 0($t1)     # v[k] = $t2
sw $t0, 4($t1)     # v[k+1] = $t0
    
```



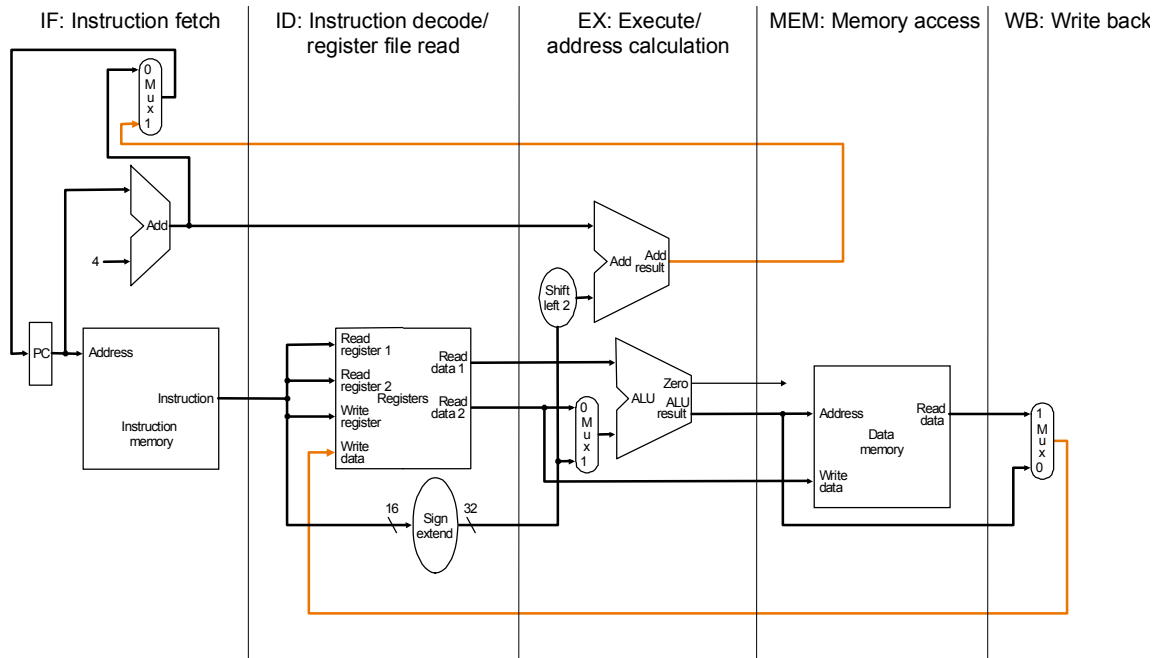
```

lw $t0, 0($t1)     # $t0 (temp) = v[k]
lw $t2, 4($t1)     # $t2 = v[k+1]
sw $t0, 4($t1)     # v[k+1] = $t0
sw $t2, 0($t1)     # v[k] = $t2
    
```

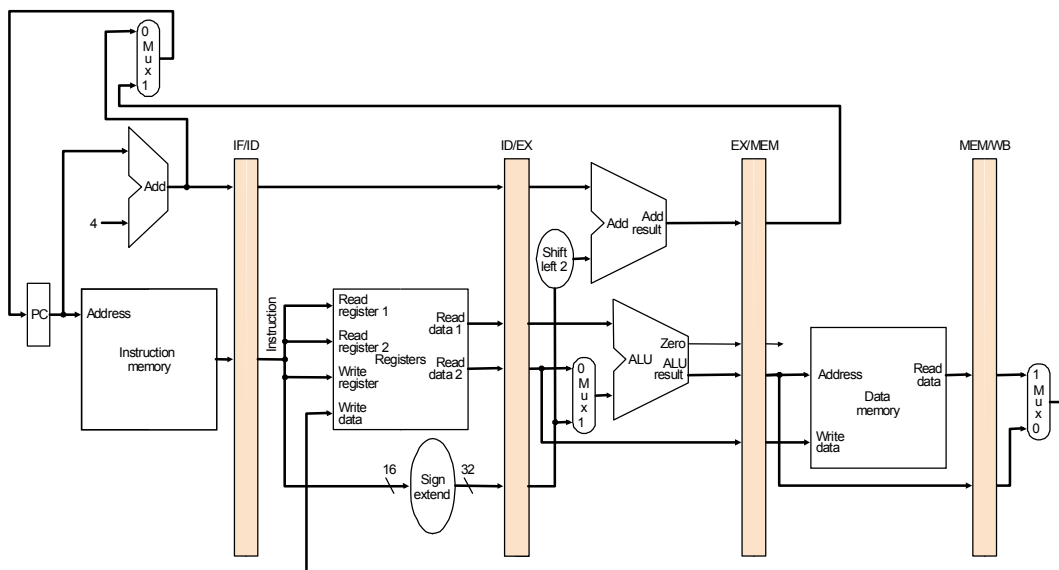

Pipelining

• Datapath para suporte à execução em pipeline

- 5 fases de execução => 5 estágios da *pipeline*



- Entre cada fase da *pipeline* é necessário introduzir um registo para conter **toda** a informação necessária à execução dessa instrução
- O nome de cada registo corresponde às fases que separa (ex. IF/ID)
- O valor de cada registo é copiado para o registo seguinte a cada ciclo de relógio.

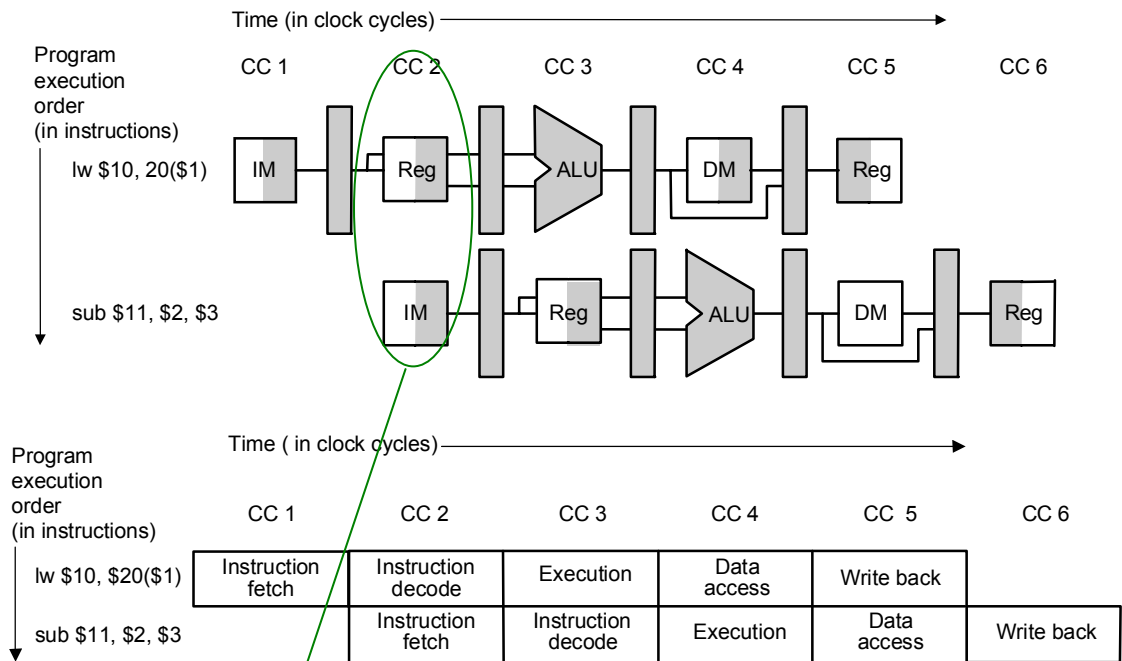


Pipelining

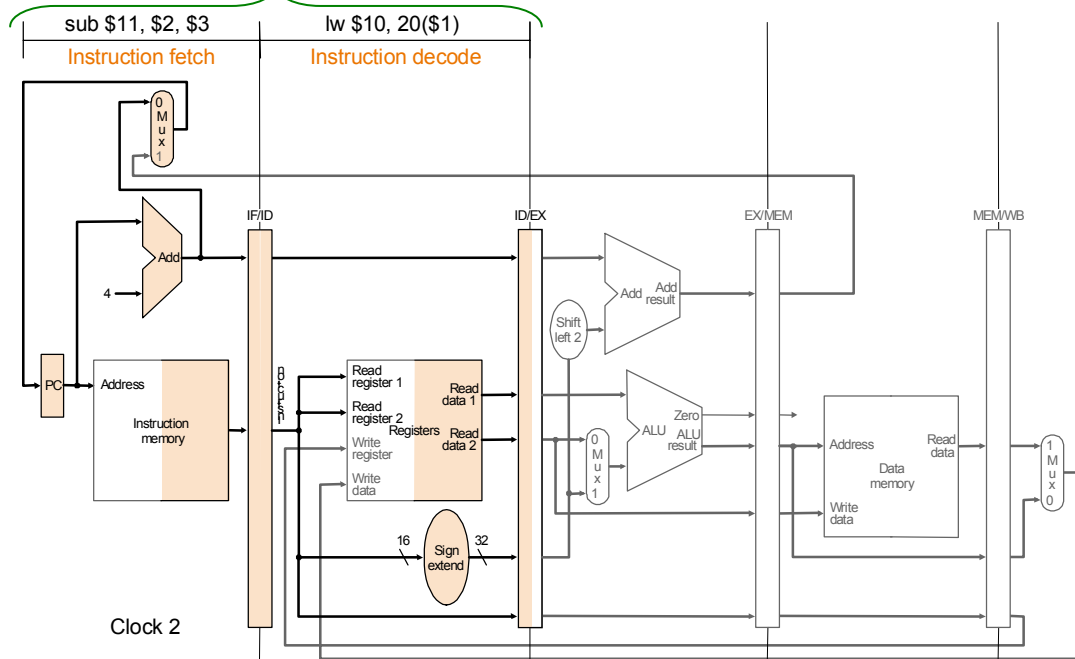
- Representação gráfica da execução em *pipelining*

```
lw $10, 20($1)
sub $11, $2, $3
```

- Representação com um diagrama multi-ciclo

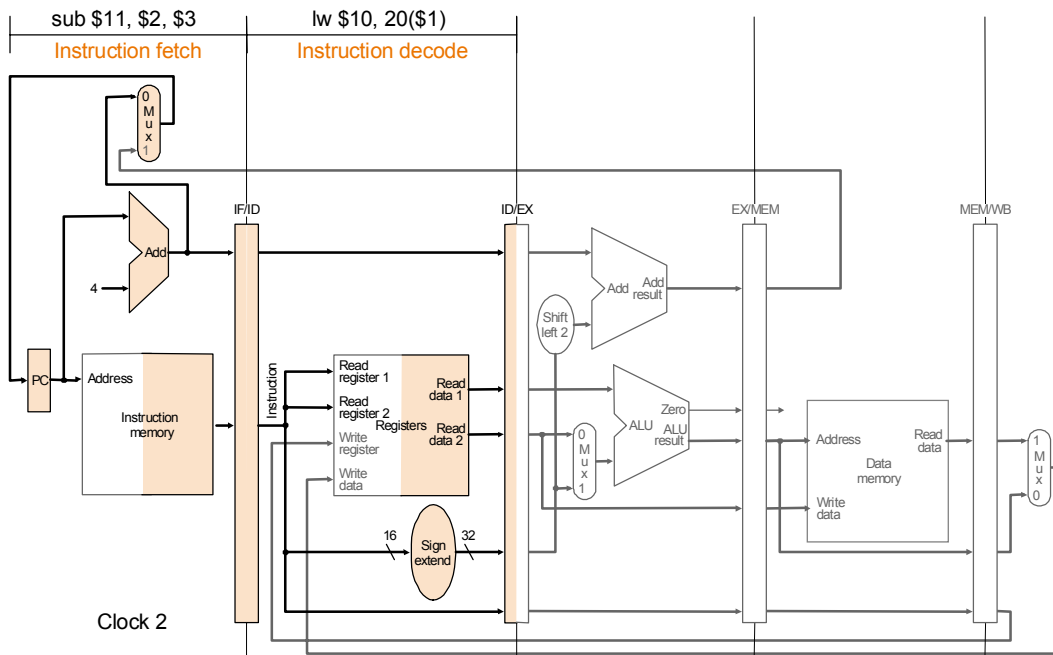
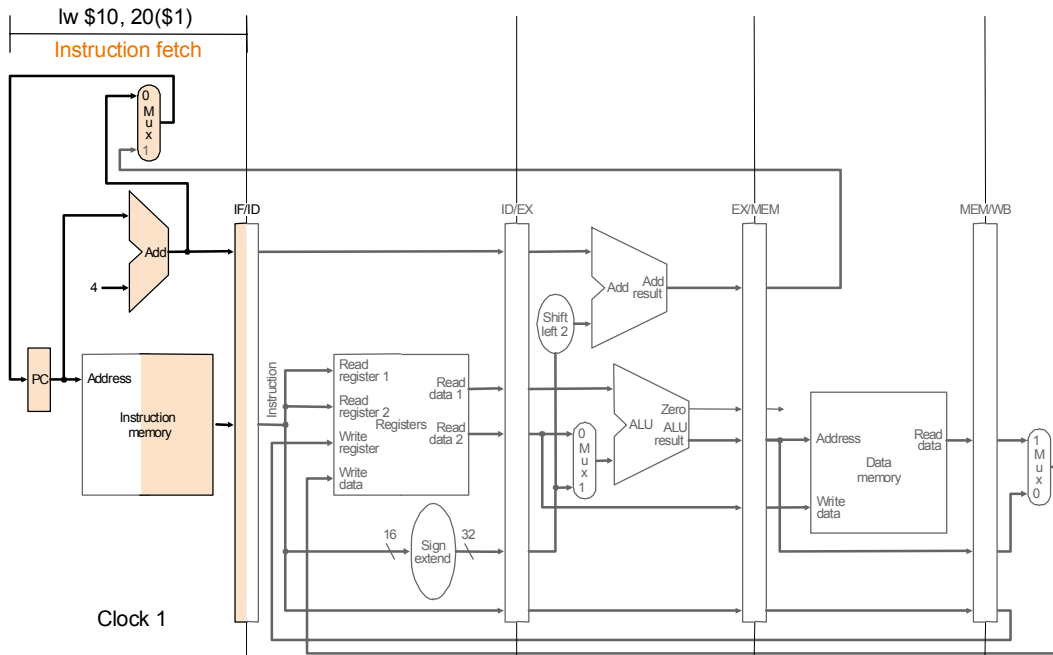


- Representação de um ciclo com um diagrama *single-cycle* (ciclo 2)



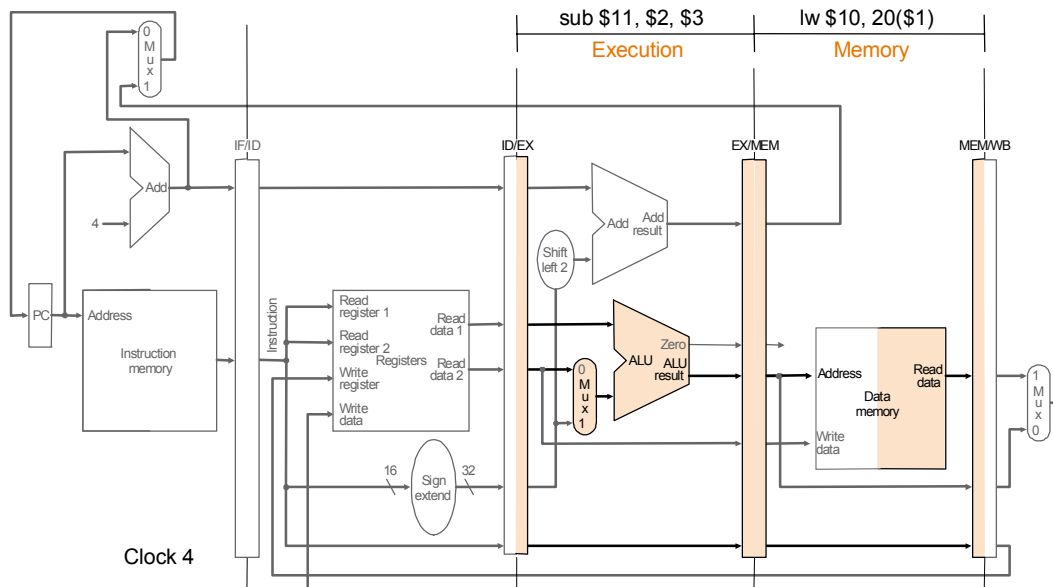
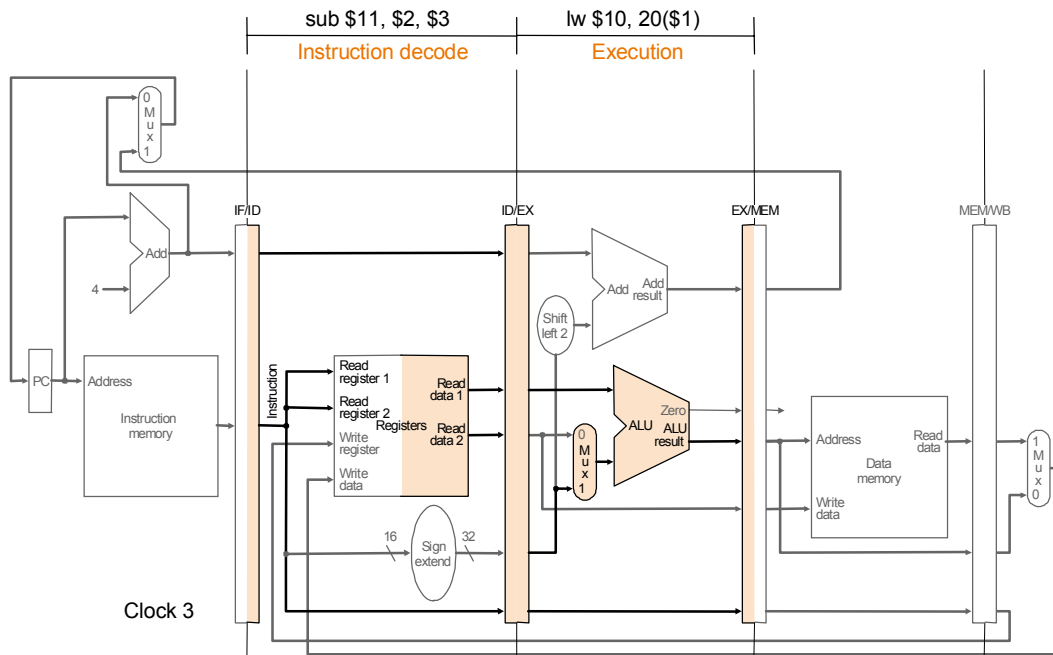
Pipelining

Exemplo de execução encadeada de instruções



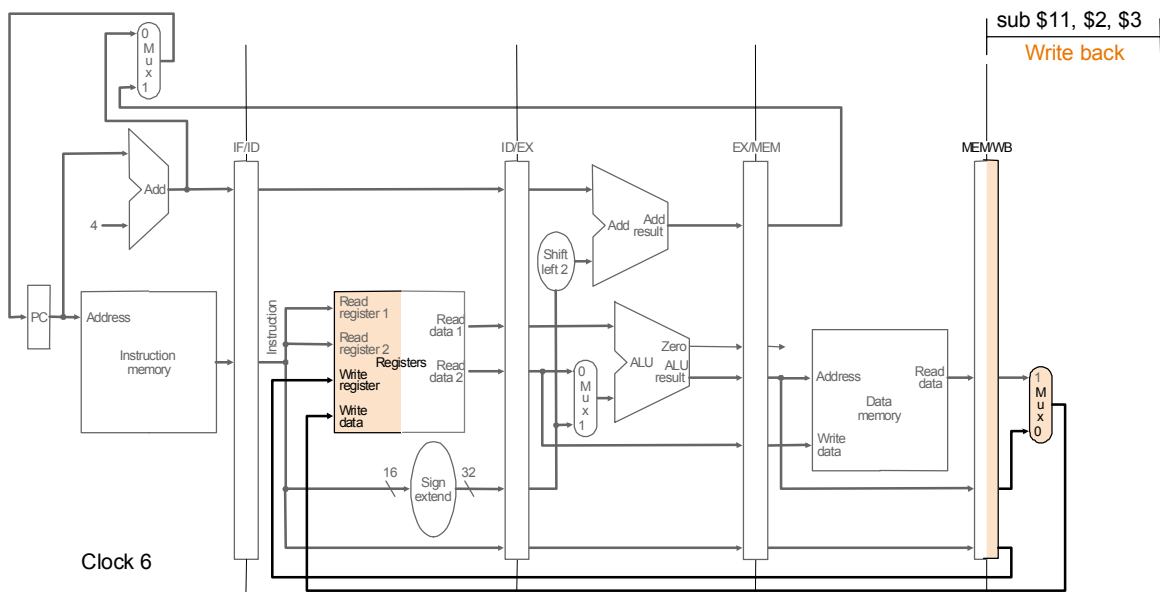
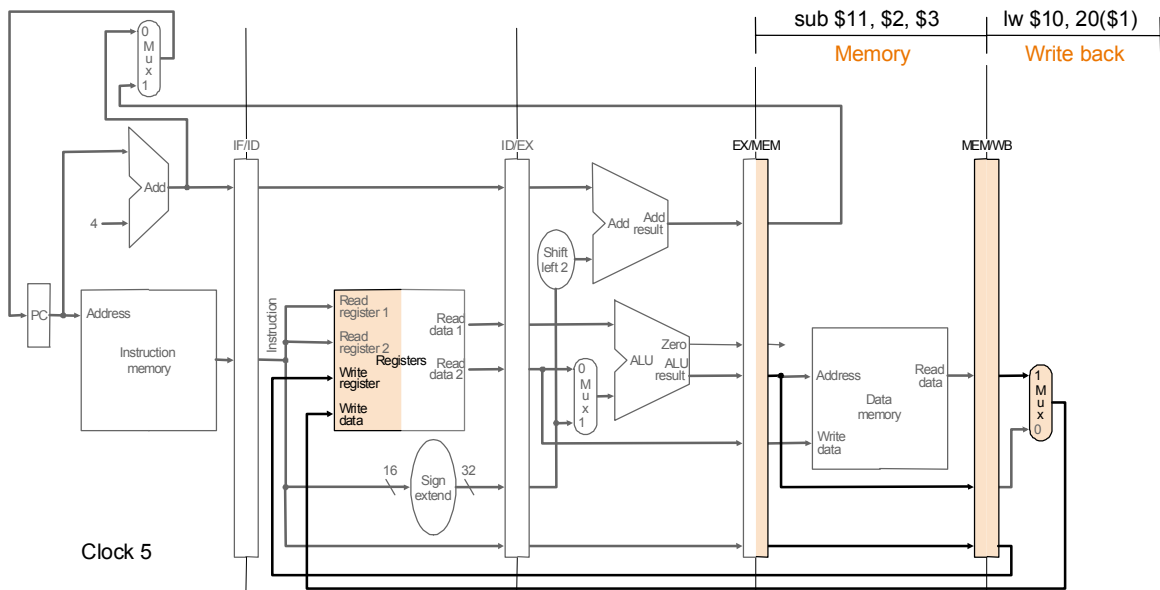
Pipelining

Exemplo de execução encadeada de instruções (cont.)



Pipelining

Exemplo de execução encadeada de instruções (cont.)



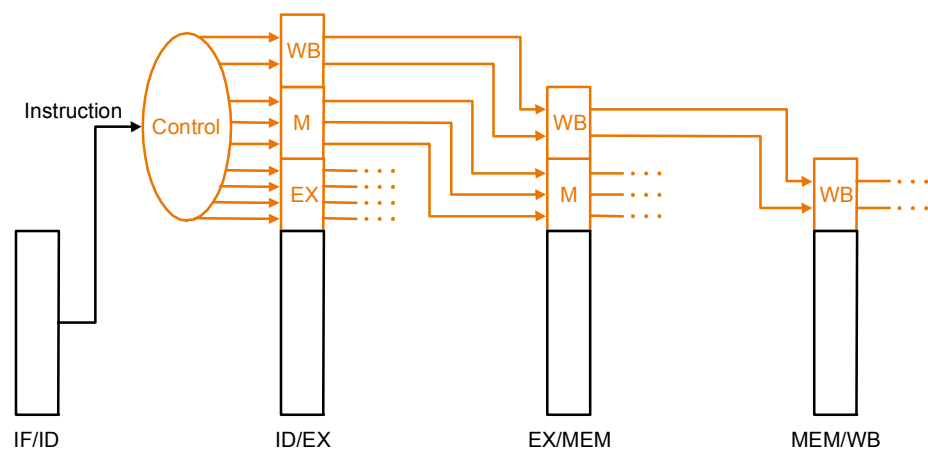
Pipelining

● Unidade de controlo para suporte à execução em *pipeline*

- Os sinais de controlo utilizados na versão do MIPS com *pipeline* são idênticos aos da versão *single-cycle*
- O registo PC e os registos da *pipeline* são escritos com novos valores a cada ciclo do relógio.
- A principal diferença consiste no facto de agora os sinais estarem associados à fase em que são necessários. As primeiras duas fases (IF e ID) são iguais para todas as instruções.

Instrução	EXE			MEM			WB	
	RegDst	ALU op	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
Tipo R	1	10	0	0	0	0	1	0
lw	0	00	1	0	1	0	1	1
sw	X	00	1	0	0	1	0	X
beq	X	01	0	1	0	0	0	X

- A informação de controlo é adicionada aos registos de *pipeline*, sendo gerada na fase de descodificação e acompanhando a execução da instrução ao longo da *pipeline*.



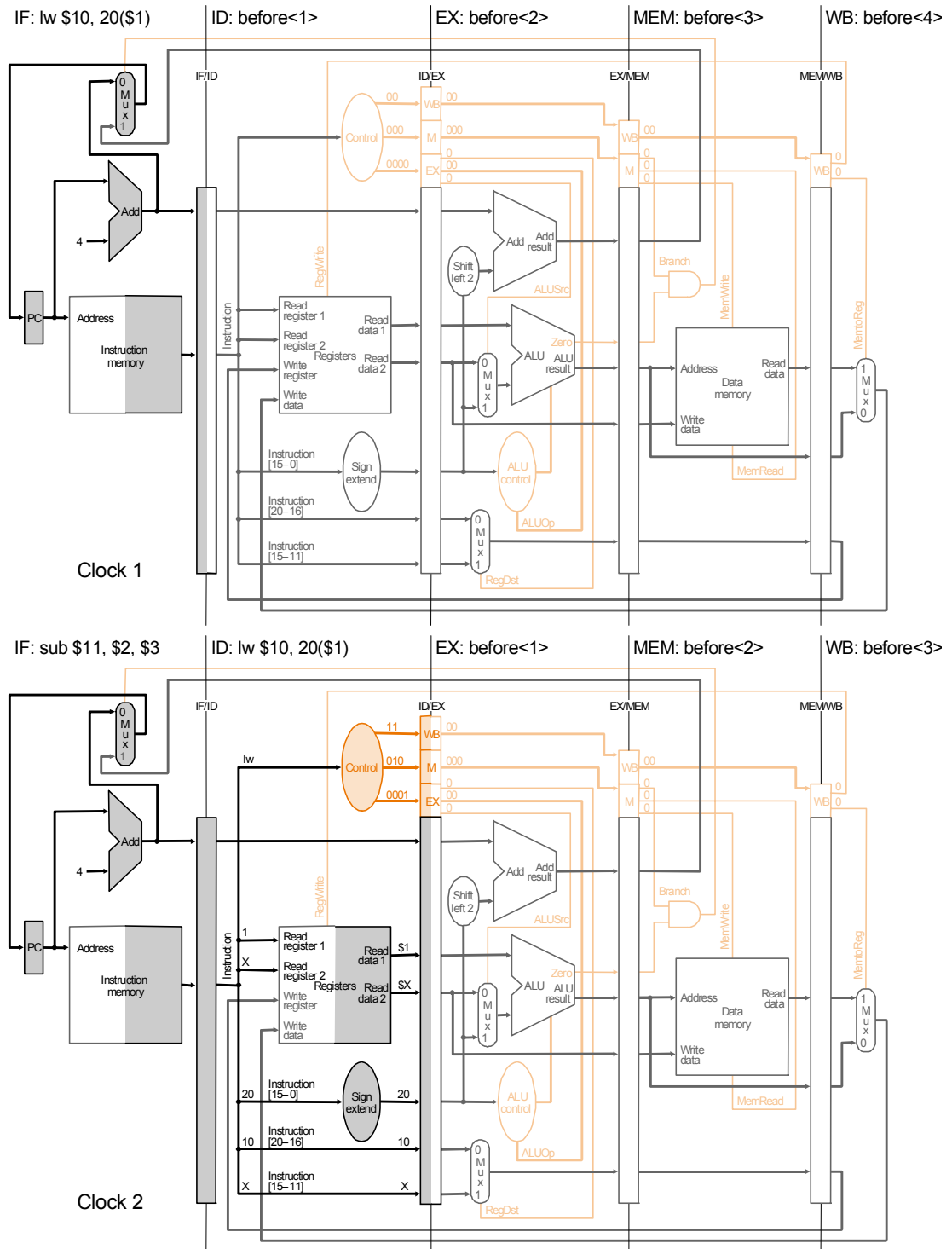
- Cada fase da *pipeline* “consome” os sinais de controlo presentes nos registos da *pipeline* que são destinados a essa fase.

- Exemplo:

```
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
```

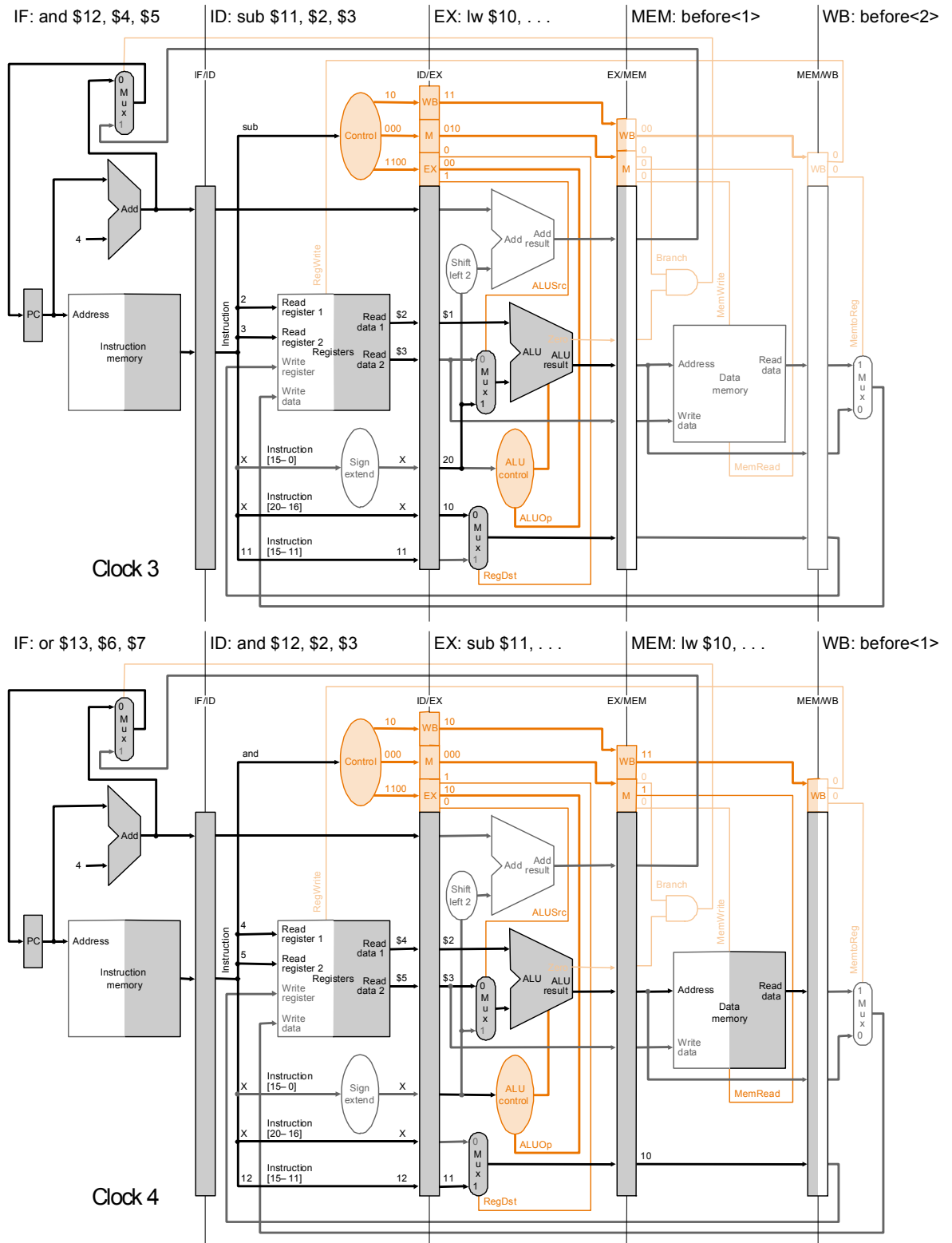
Pipelining

Exemplo 2 de execução encadeada de instruções (c/sinais de controlo)



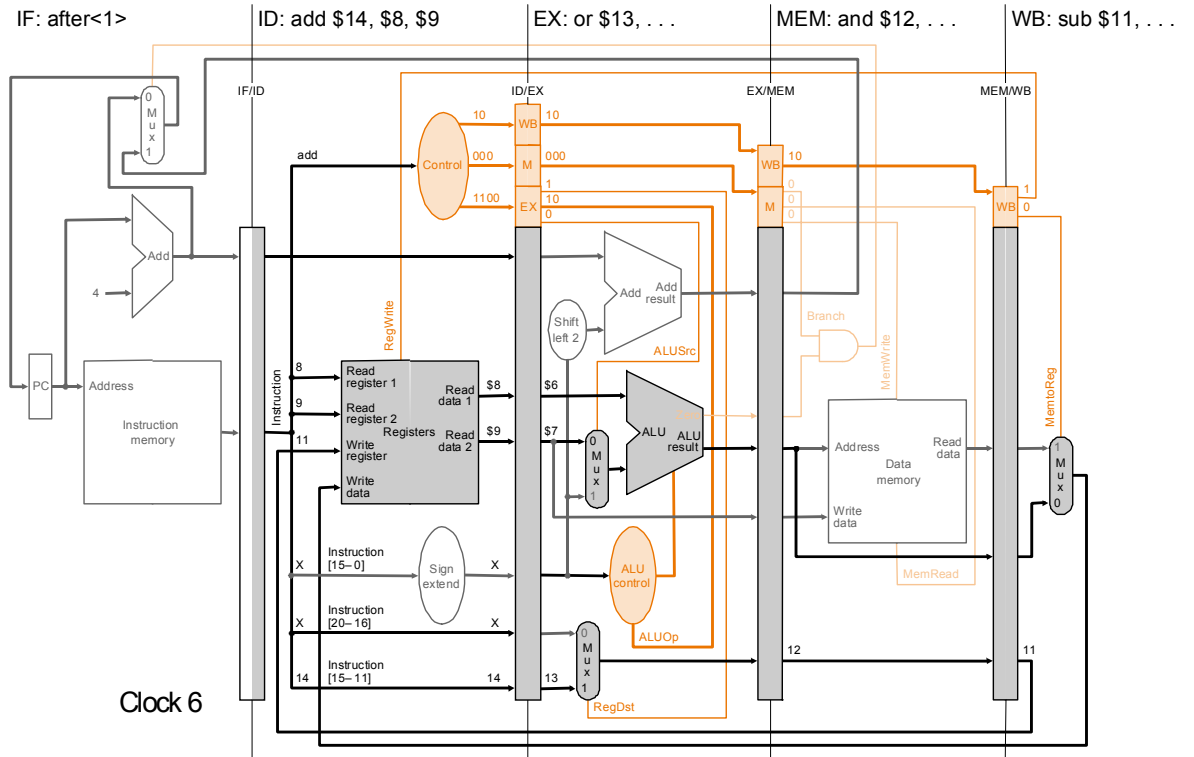
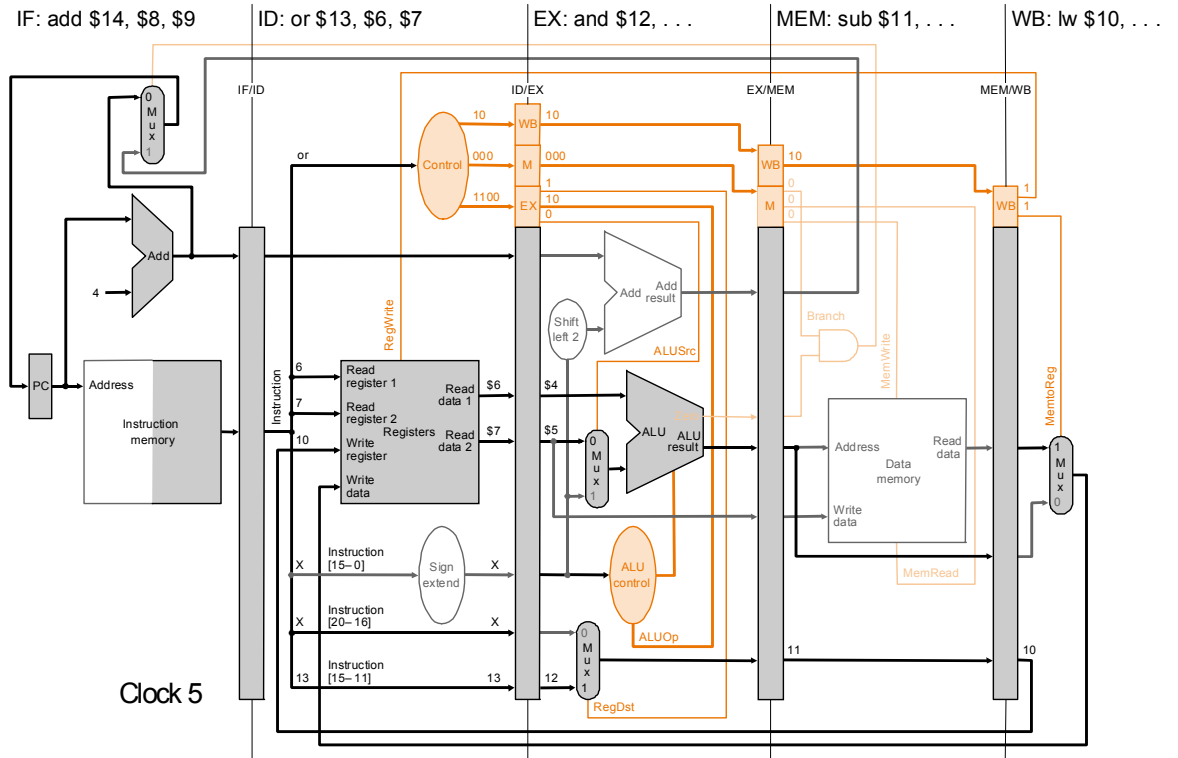
Pipelining

Exemplo 2 de execução encadeada de instruções (cont.)



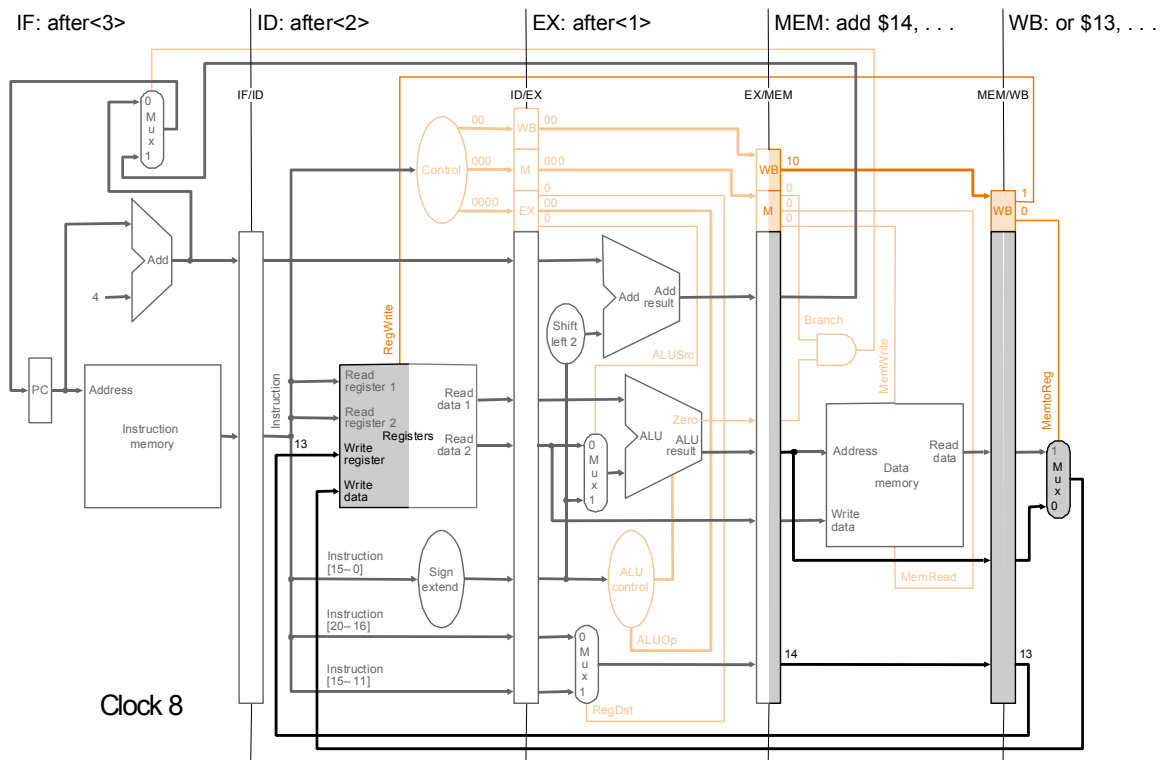
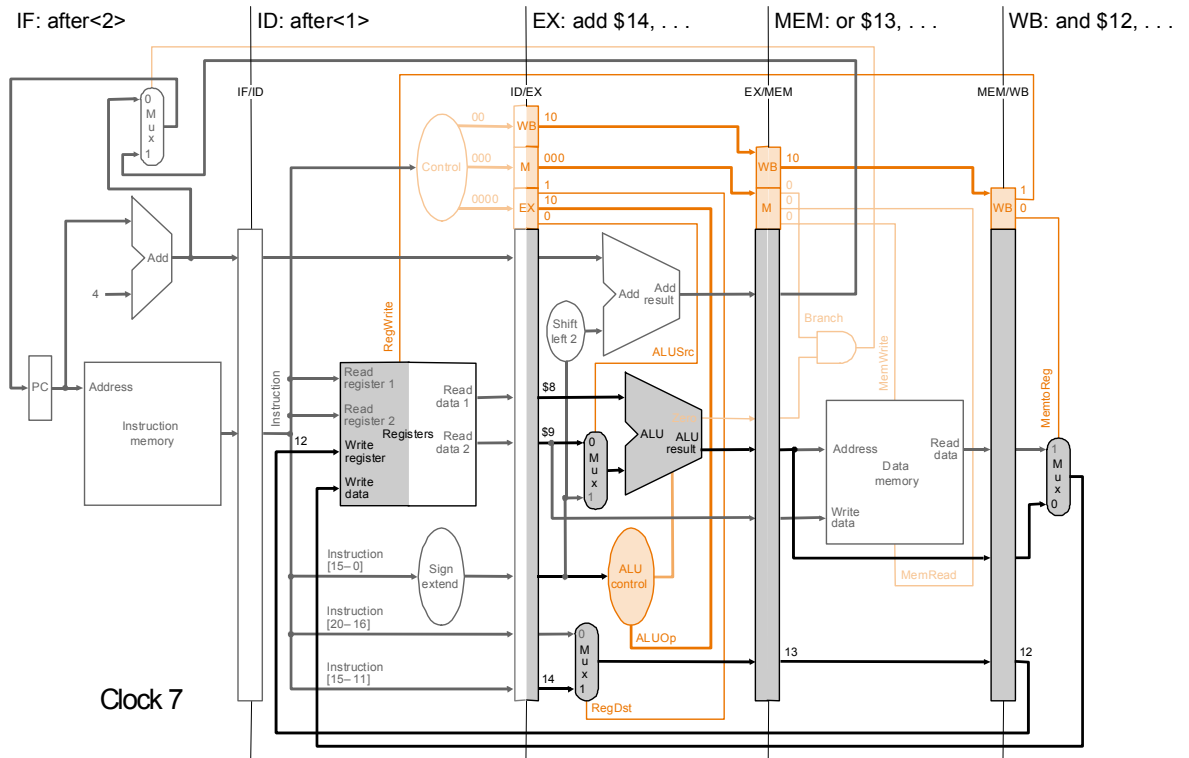
Pipelining

Exemplo 2 de execução encadeada de instruções (cont.)



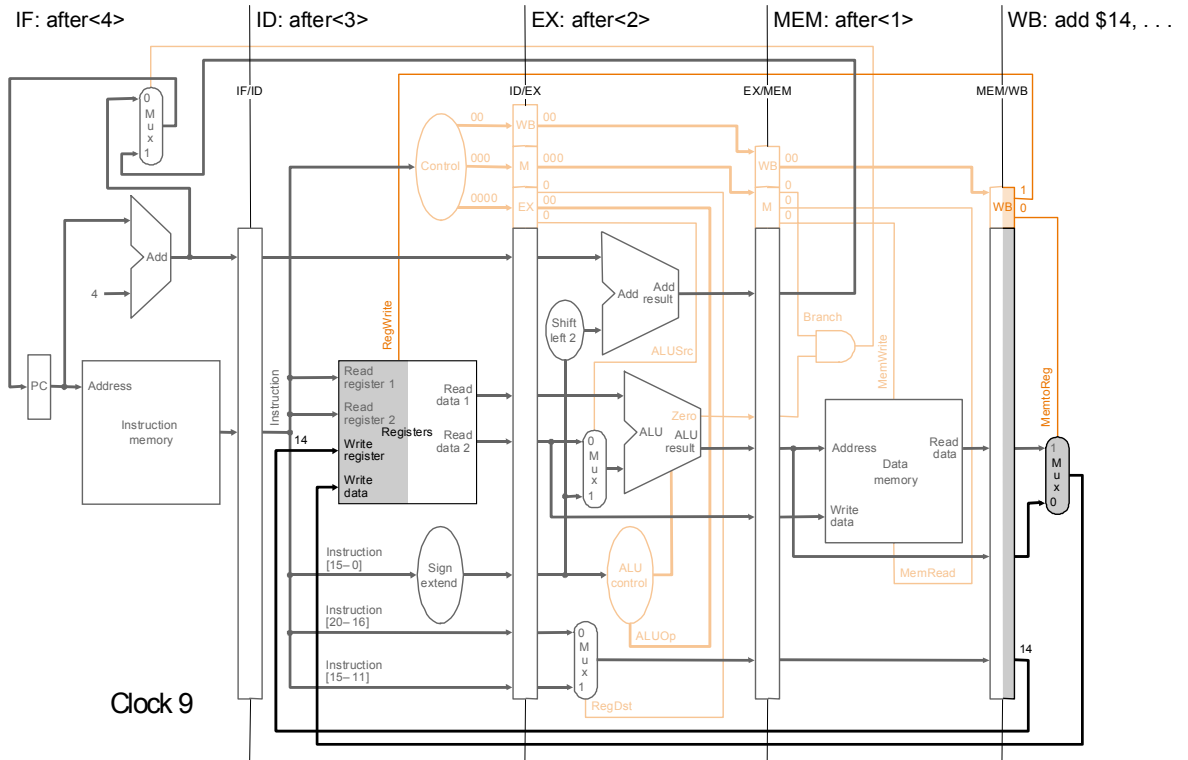
Pipelining

Exemplo 2 de execução encadeada de instruções (cont.)



Pipelining

Exemplo 2 de execução encadeada de instruções (cont.)



Pipelining

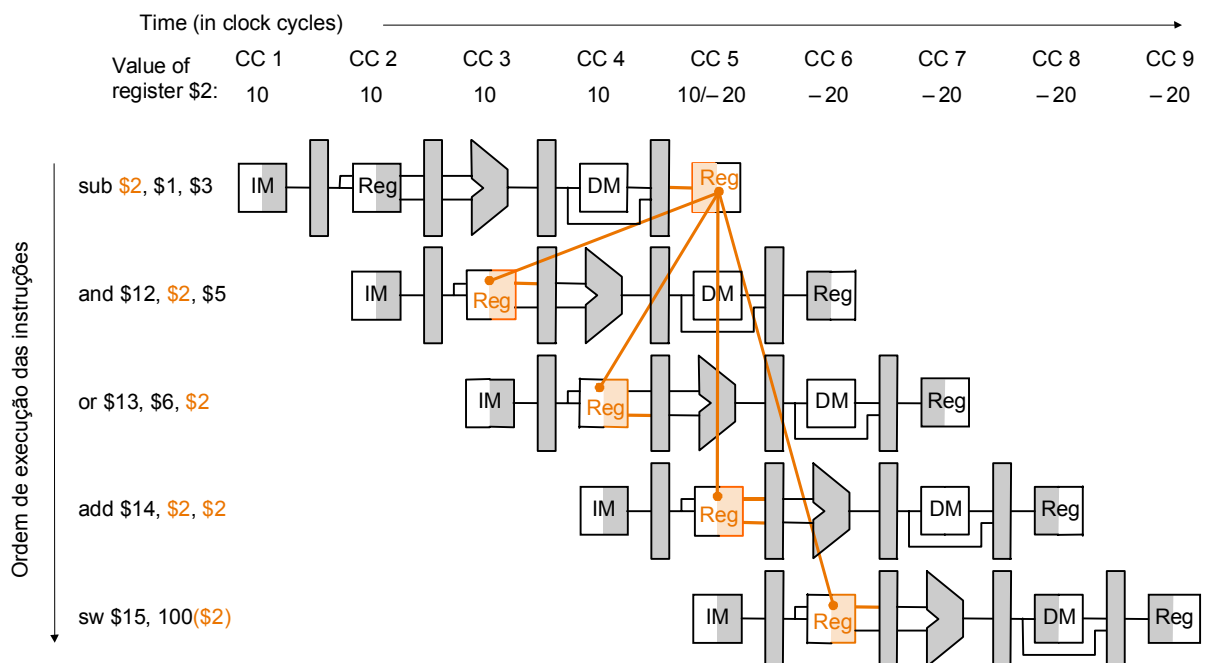
Dependências de dados (anomalias)

- A execução em *pipelining* gera problemas adicionais quando as instruções dependem de resultados produzidos por instruções anteriores:

```

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

- As anomalias resultam do facto das instruções seguintes efectuarem a leitura do banco de registos antes da instrução escrever no registo:



- As anomalias geradas pela leitura e escrita de um registo no mesmo ciclo (ex. `add $14, $2, $2` no ciclo 5) podem ser resolvidas ao nível do hardware se as escritas em registos forem efectuadas na primeira metade do ciclo e as leituras na segunda metade do ciclo.
- Uma solução (pouco eficiente) para os restantes casos seria obrigar o compilador a introduzir instruções de *nop* para eliminar as dependências.

```

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

Pipelining

Resolução de dependências de dados (anomalias)

- Detecção das dependências de dados** – detecção dos casos em que o registo destino numa instrução na fase de EX ou WB é utilizado como uma das entradas da ALU (fase EX):

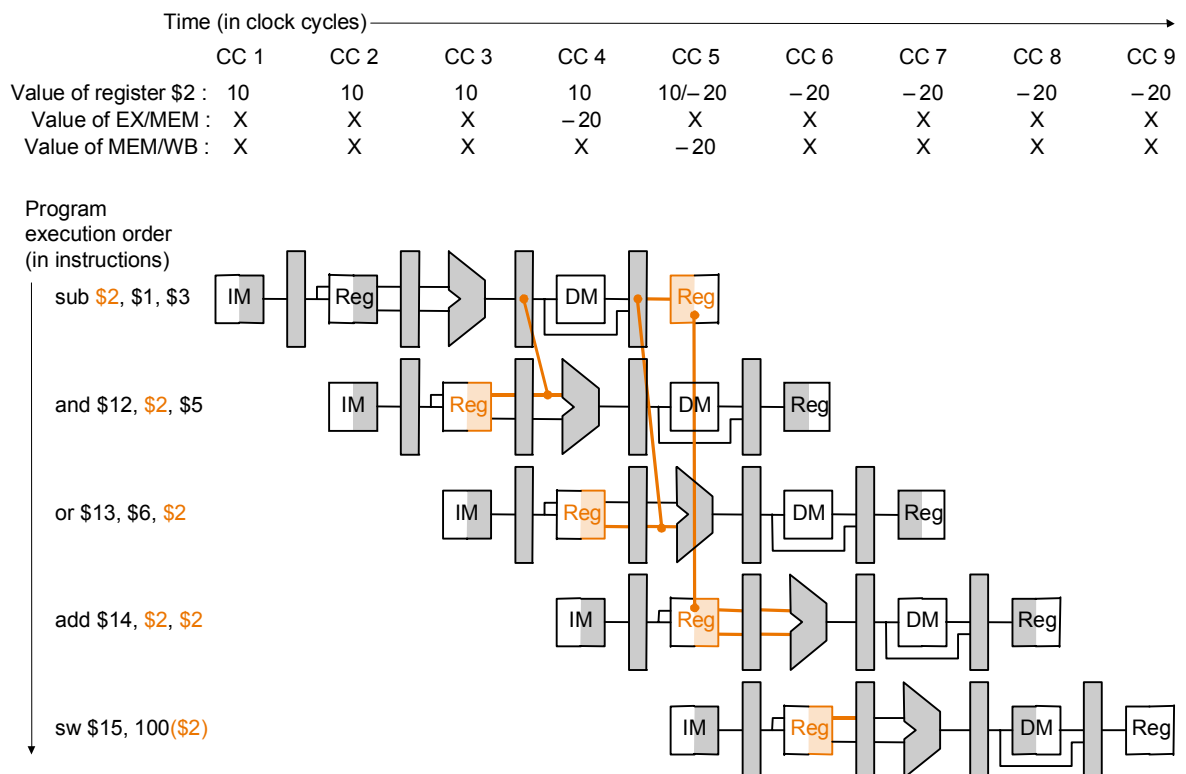
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

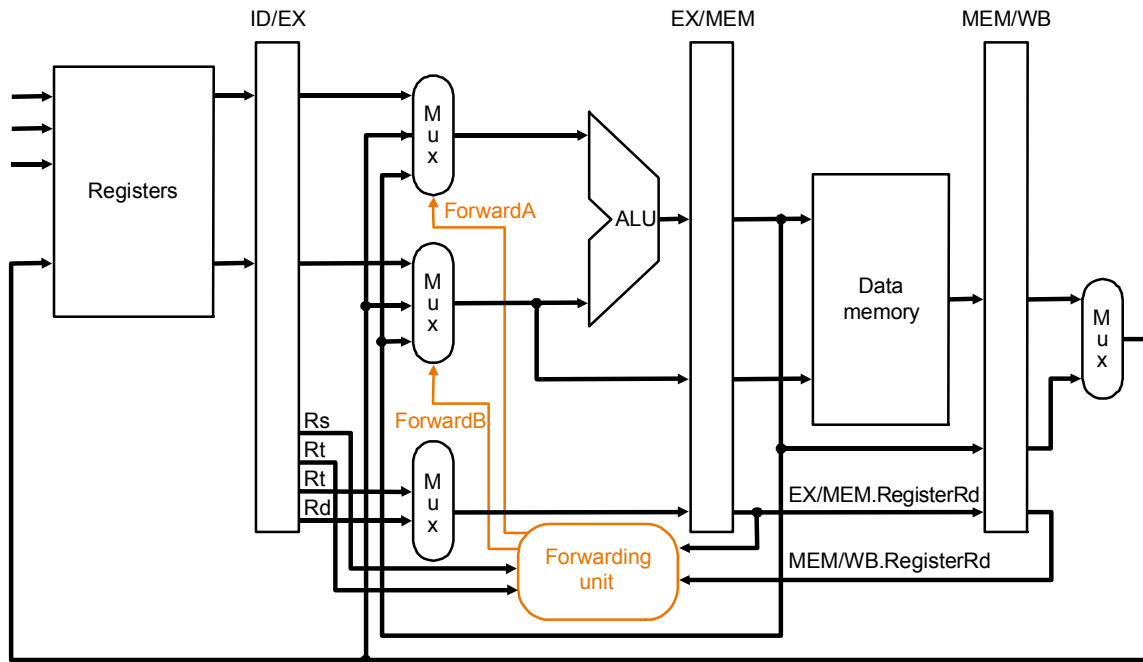
- Encaminhamento de dados** – Na maior parte das dependências de dados o resultado já se encontra calculado (em EX/MEM ou MEM/WB), mas ainda não foi escrito no banco de registos. Assim é possível criar atalhos no DP para encaminhar os dados e resolver as dependências.



Pipelining

Resolução de dependências de dados com encaminhamento de dados

- O DP para resolução de dependências de dados deve ter a capacidade de encaminhar para as entradas da ALU os valores em EX/MEM e MEM/WB. Tal é conseguido através da introdução de multiplexers na em cada entrada da ALU



- As condições anteriormente indicadas não são suficientes porque algumas instruções não escrevem nos registos ou escrevem em \$0:

Só efectua o encaminhamento quando a instrução escreve num registo e esse registo não é \$0

```
if (EX/MEM.RegWrite)
  and (EX/MEM.RegisterRd≠0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs) ForwardA = 10
```

```
if (EX/MEM.RegWrite)
  and (EX/MEM.RegisterRd≠0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt) ForwardB = 10
```

```
if (MEM/WB.RegWrite)
  and (MEM/WB.RegisterRd≠0)
  and (EX/MEM.registerRd≠ID/EX.RegisterRs)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs) ForwardA = 01
```

```
if (MEM/WB.RegWrite)
  and (MEM/WB.RegisterRd≠0)
  and (EX/MEM.registerRd≠ID/EX.RegisterRt)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt) ForwardB = 01
```

Quando EX/MEM.registerRd = MEM/WB.registerRd prefere o valor mais recente (em EX/MEM). exemplo:

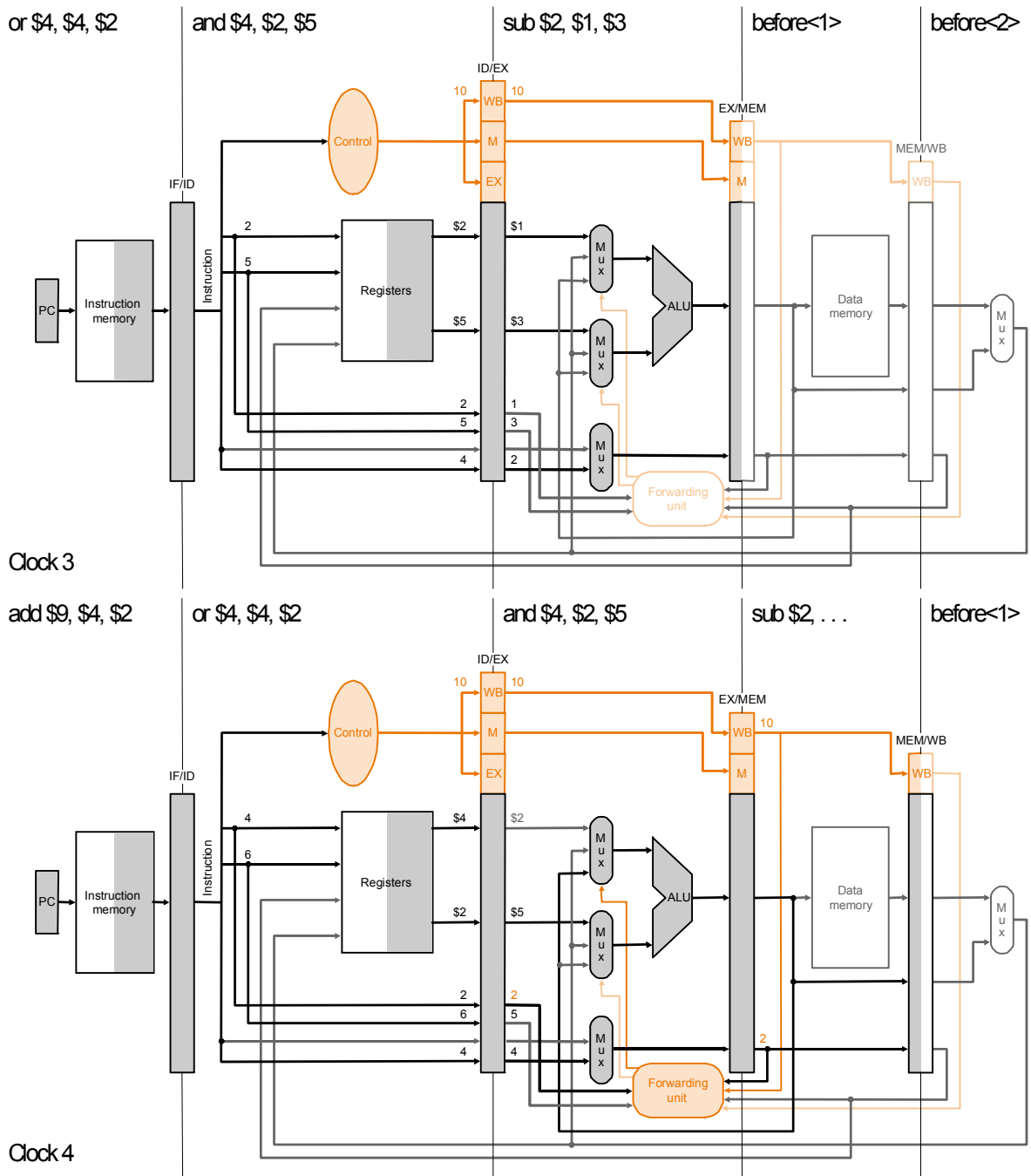
```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

Pipelining

Exemplo de encaminhamento de dados

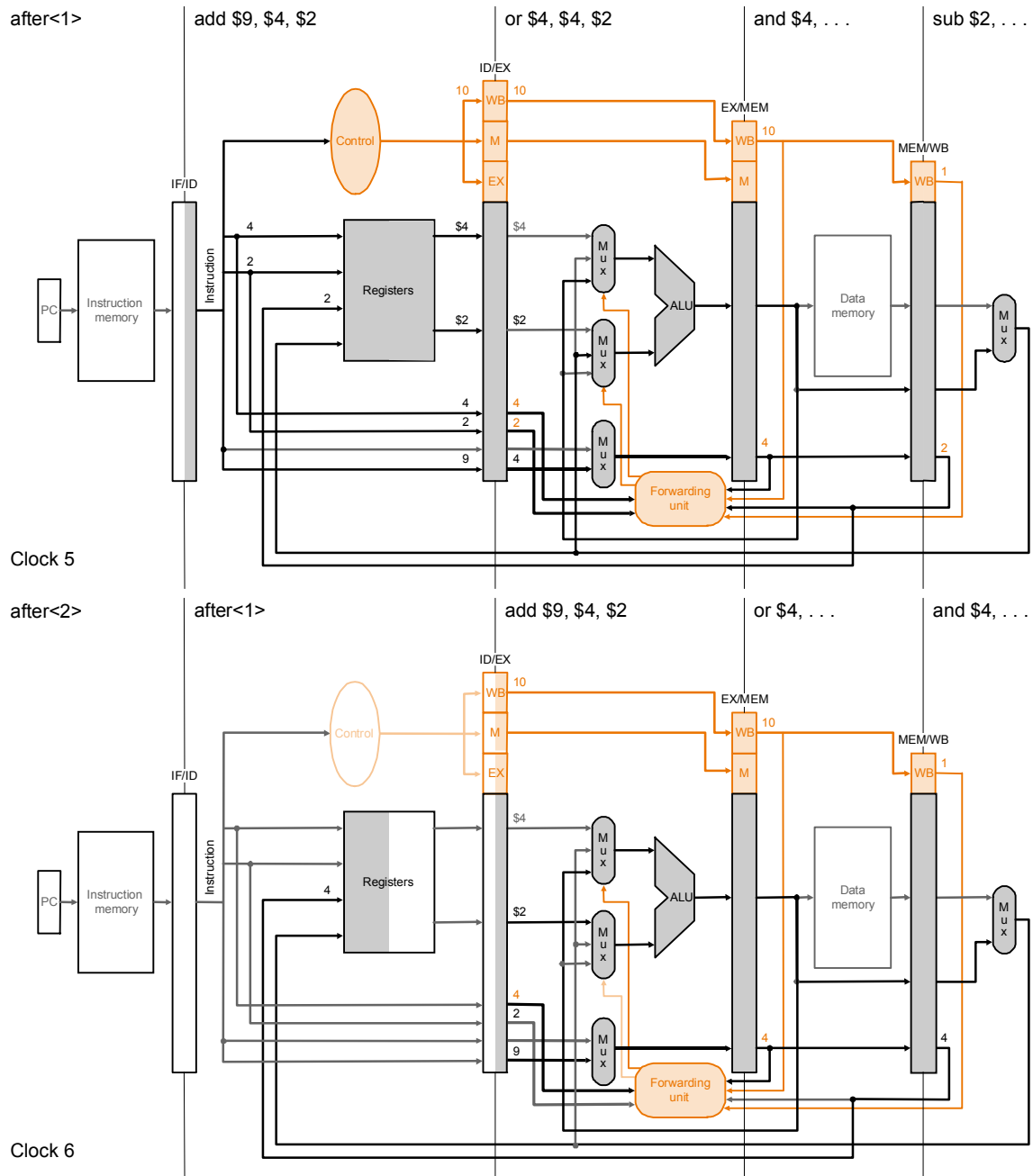
```

sub  $2, $1, $3
and  $5, $2, $5
or   $4, $4, $2
add  $9, $4, $2
    
```



Pipelining

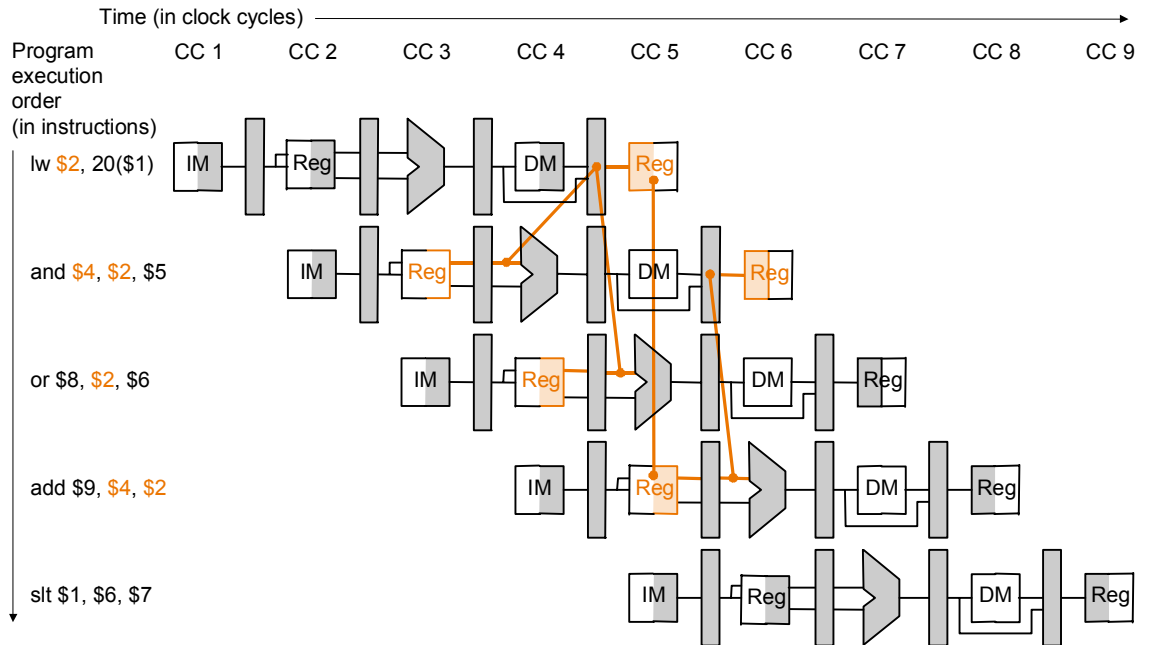
Exemplo de encaminhamento de dados (cont.)



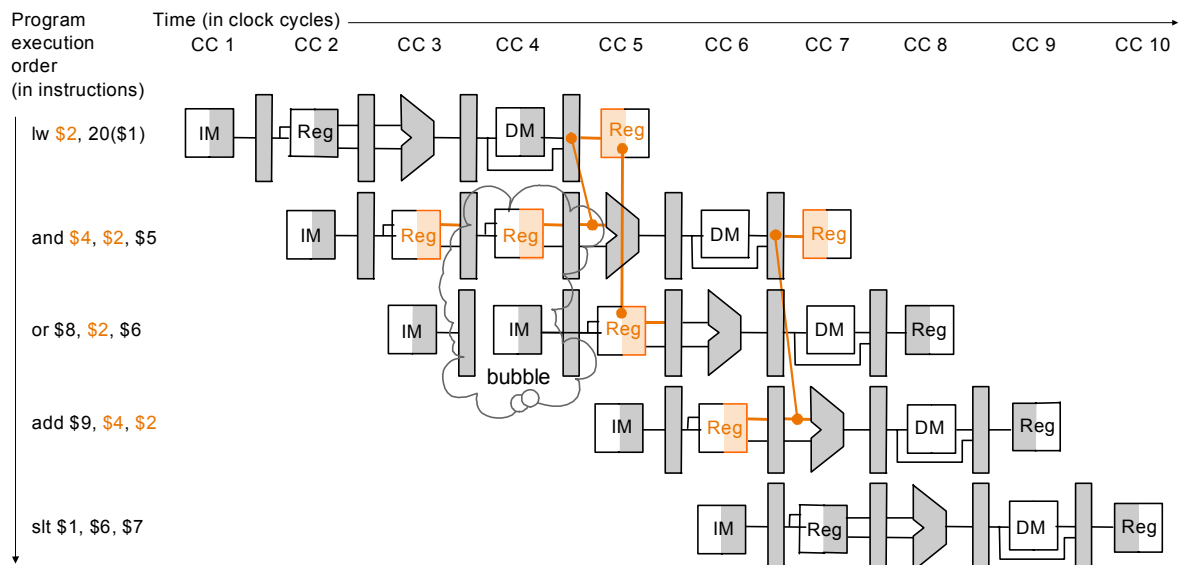
Pipelining

Resolução de dependências de dados: “empatar” o pipeline

- A utilização do valor de um *lw* na instrução seguinte introduz uma anomalia que não é resolvida com atalhos.



- Nos caso em que as anomalias não são resolvidas é necessário empatar o pipeline até que o valor esteja disponível:



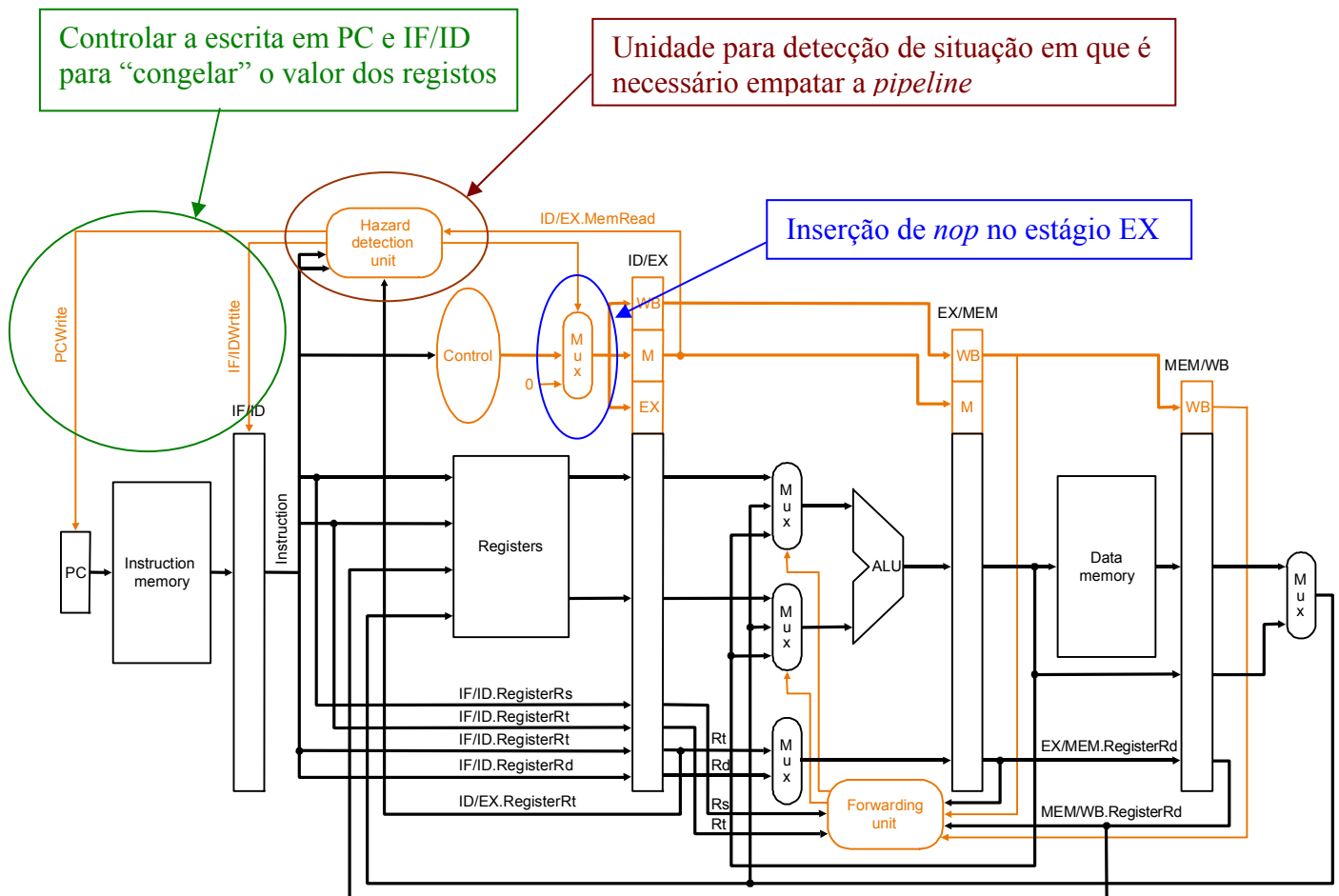
Pipelining

Resolução de dependências de dados: “empatar” o pipeline (cont.)

- A condição em que é necessário empatar o pipeline é dada por:

$$\text{if (ID/EX.MemRead) and} \\ \text{((ID/EX.RegisterRt=IF/ID.RegisterRs)} \\ \text{or (ID/EX.RegisterRt = IF/ID.RegisterRt))}$$

- Esta condição é implementada por uma nova unidade, designada por unidade de detecção de anomalias (*Hazard detection unit*).
- Empatar o pipeline é implementado através de sinais de não permitem a escrita nos registos PC, IF/ID e da inserção de um *nop* no estágio EX (nos sinais de controlo).

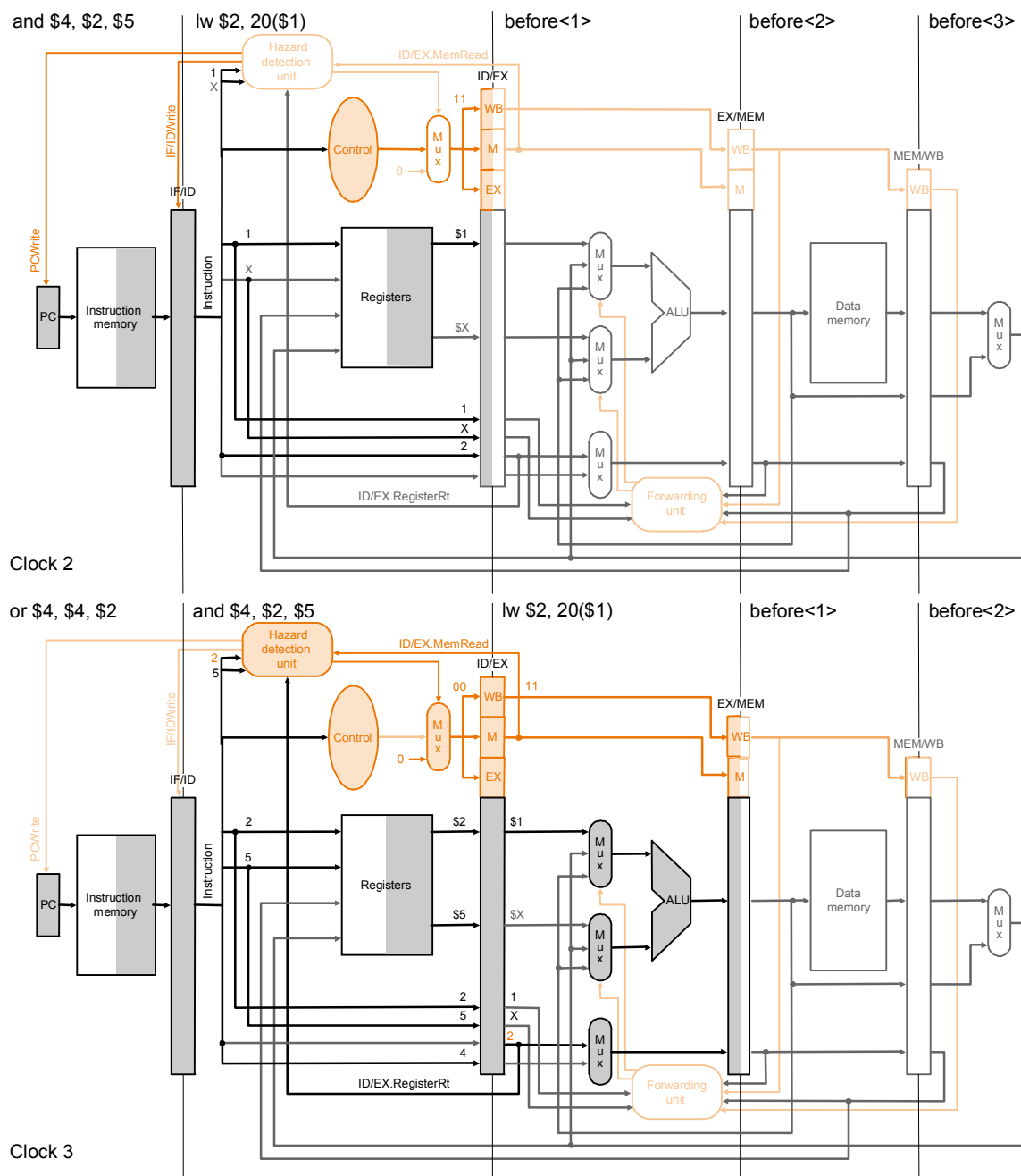


Pipelining

- Exemplo de *stall* da cadeia de execução (c/encaminhamento de dados)

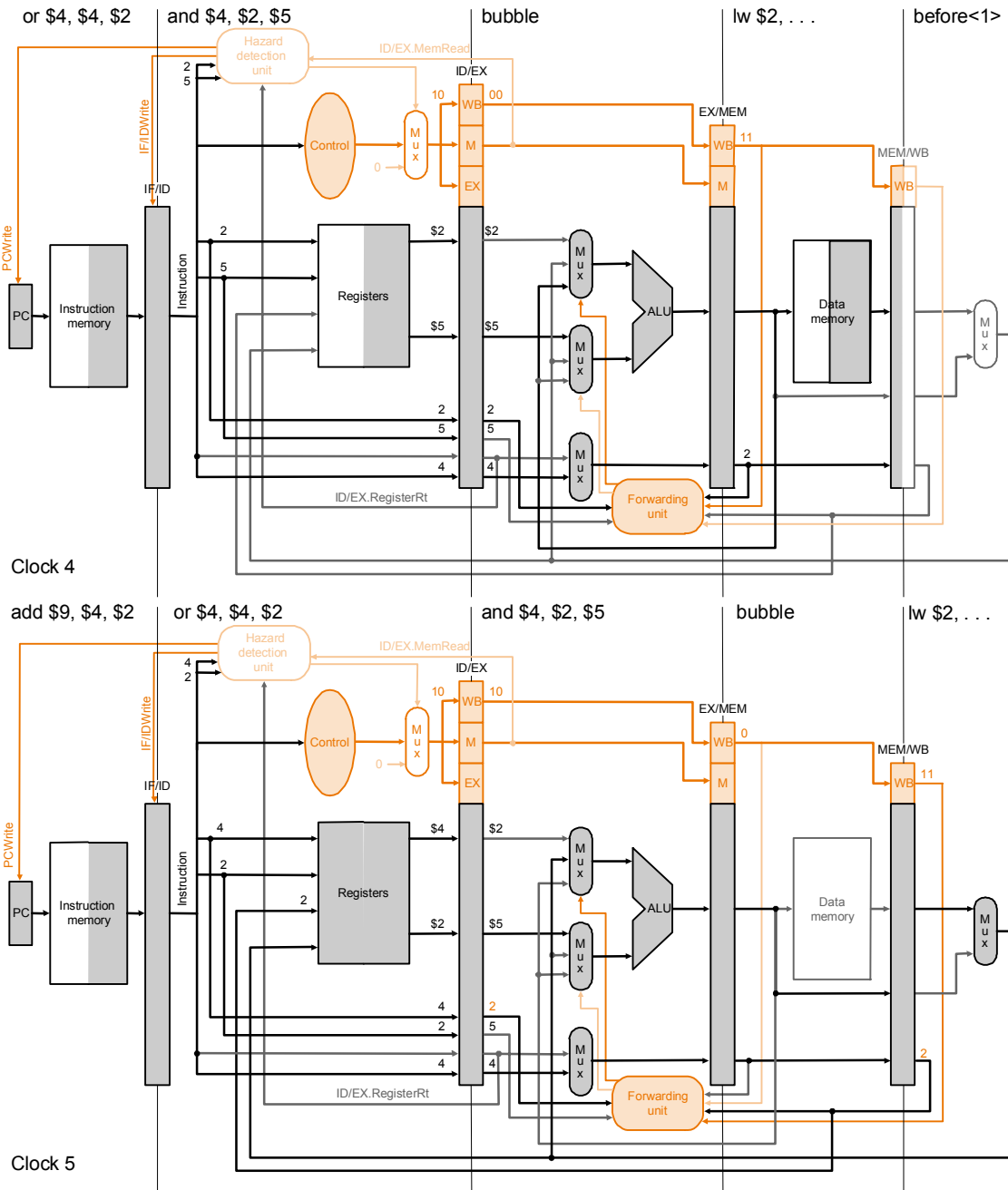
```

lw    $2, 20($1)
and   $5, $2, $5
or    $4, $4, $2
add   $9, $4, $2
    
```



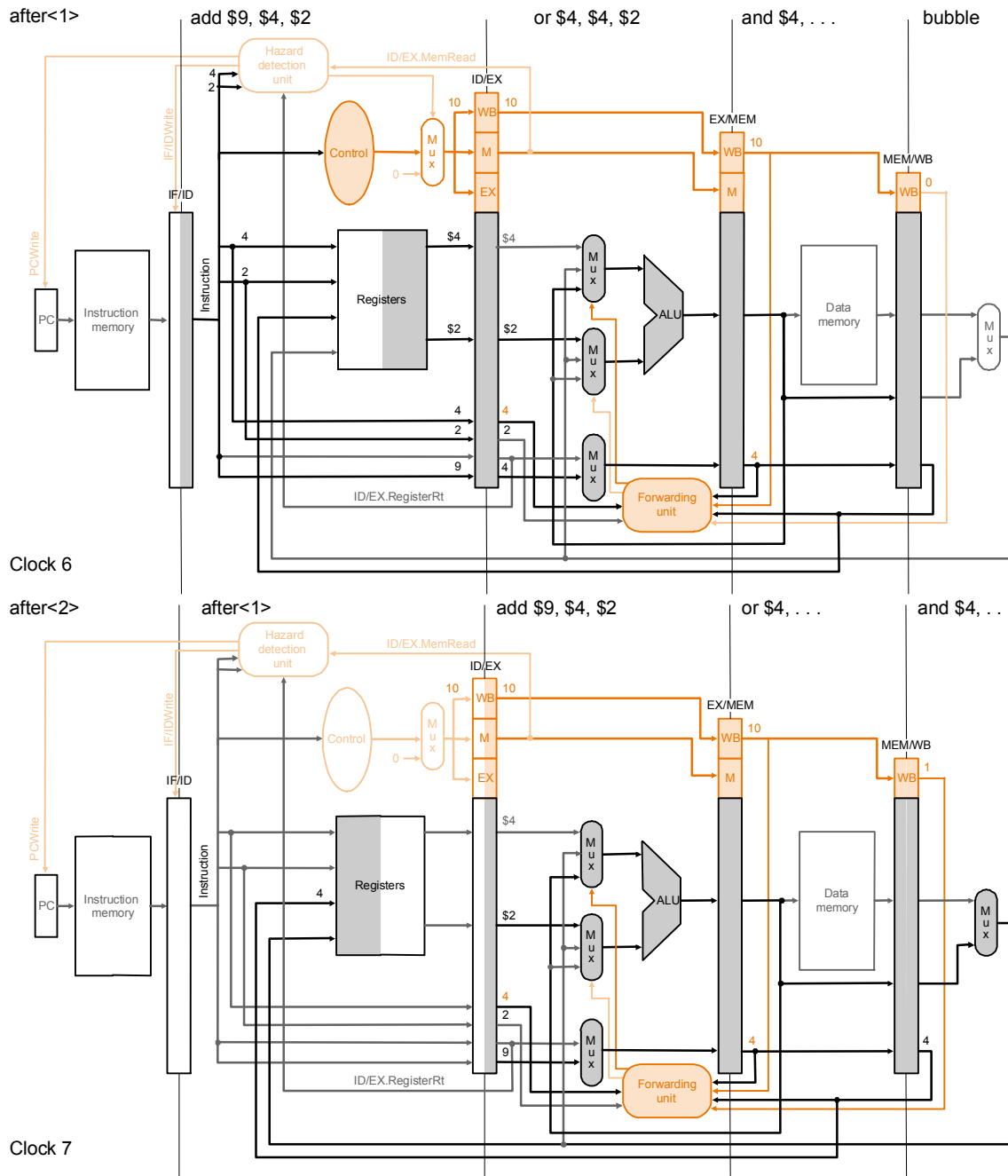
Pipelining

Exemplo de *stall* da cadeia de execução (cont.)



Pipelining

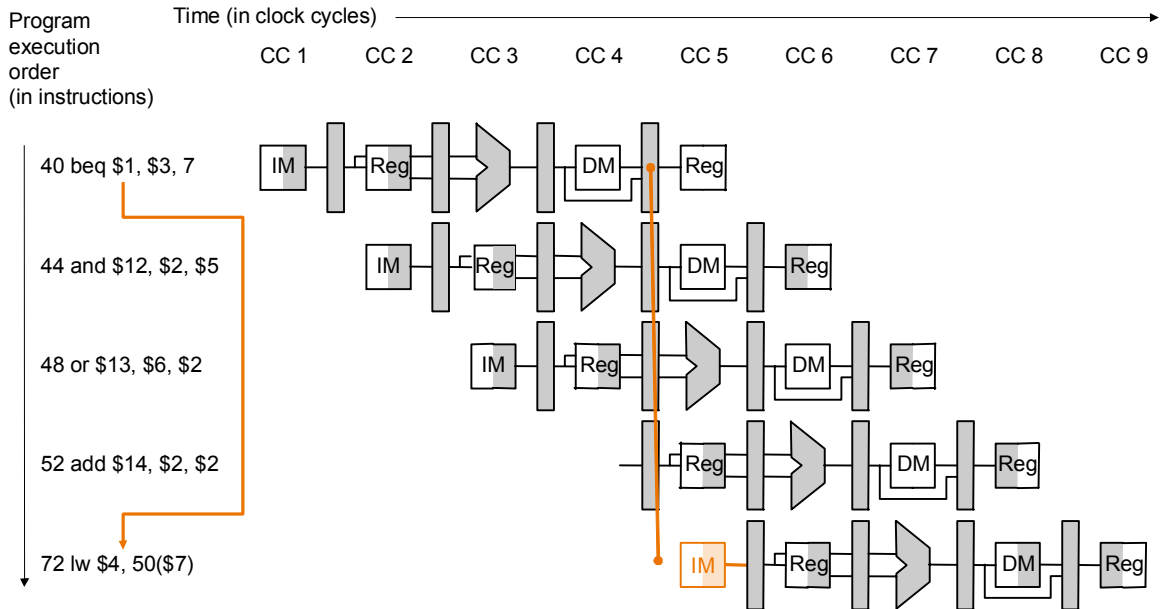
Exemplo de *stall* da cadeia de execução (cont.)



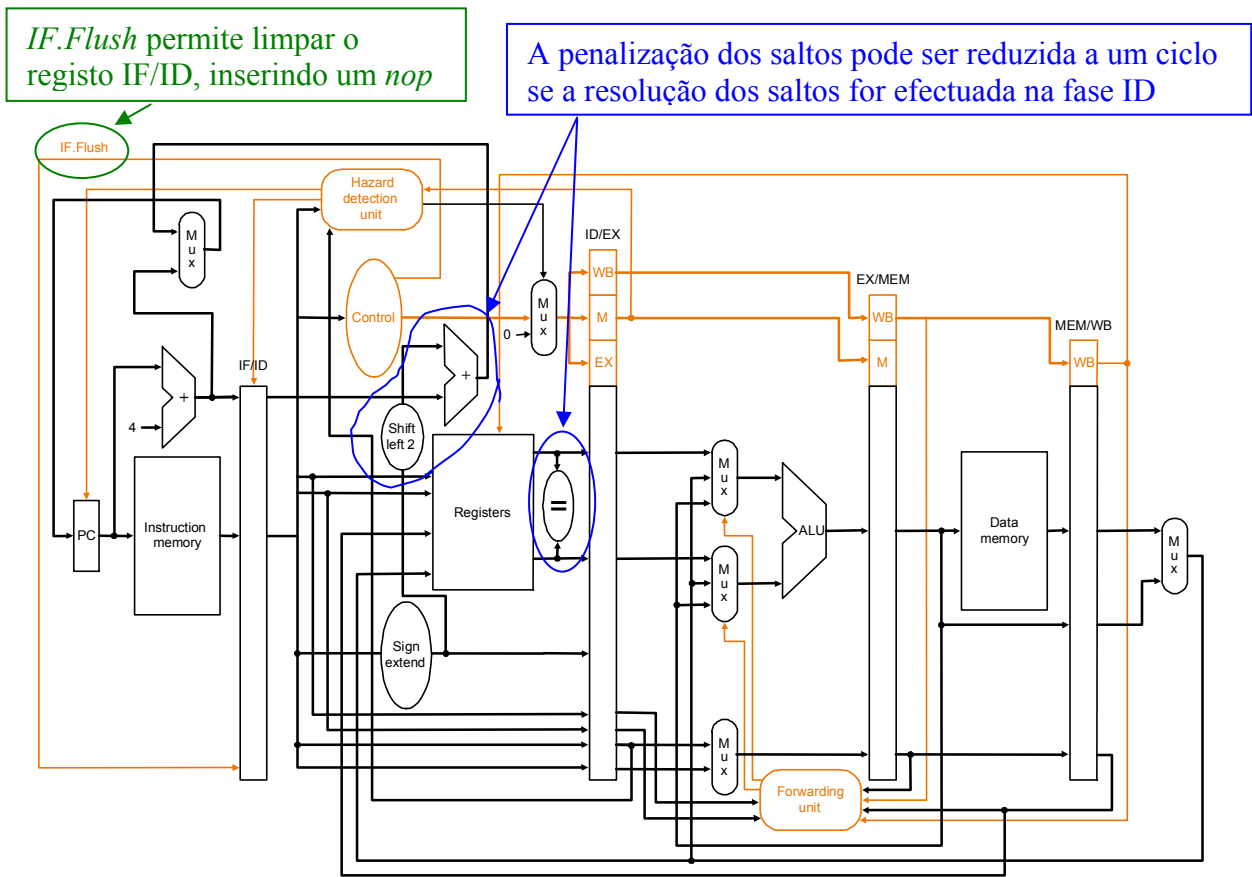
Pipelining

Resolução de dependências controlo

- Os saltos condicionais implicam uma penalização de 3 ciclos, uma vez que o endereço de salto apenas é resolvido na fase MEM:



- Solução 1 : resolver os saltos mais cedo (exemplo para fase ID)



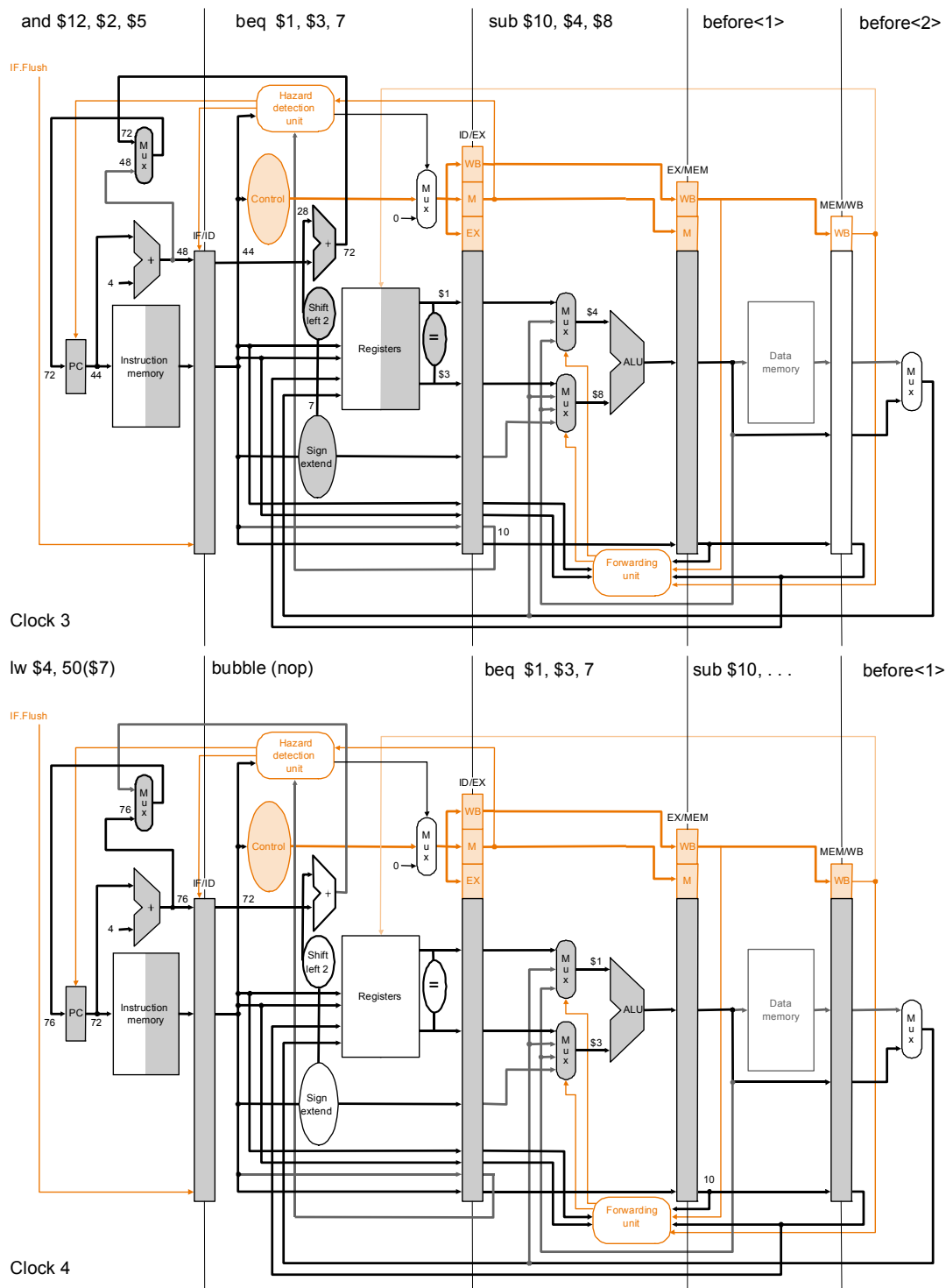
Pipelining

Exemplo de redução da penalização dos saltos

```

38 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
...
72 lw $4, 50($7)

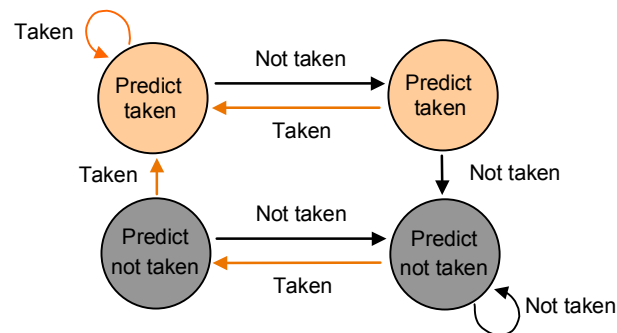
```



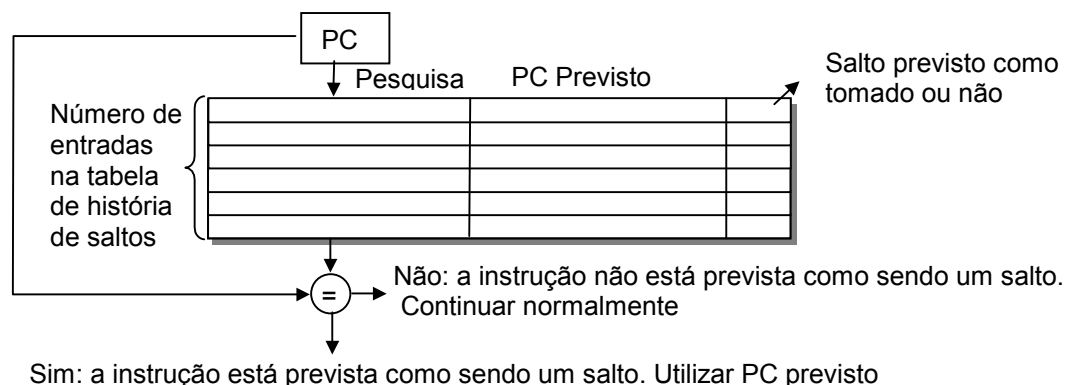
Pipelining

Resolução de dependências controlo

- **Previsão estática de saltos**, por exemplo, prevendo que o salto não é tomado, pode ser implementada apenas adicionado lógica para eliminar as instruções em IF, ID, EX quando a previsão falha ($\approx 50\%$ de previsões acertadas).
- **Previsão dinâmica de saltos baseada numa tabela com a história dos saltos anteriores ($\approx 90\%$ de previsões acertadas).**
 - *Um bit por endereço de salto* - pouco eficiente, uma vez que nos ciclos *for* falha sempre duas vezes: na entrada (resultante da última execução) e na saída.
 - *dois bits* - altera a previsão quando erra duas vezes seguidas, o que, em geral, melhora a percentagem de previsões acertadas:



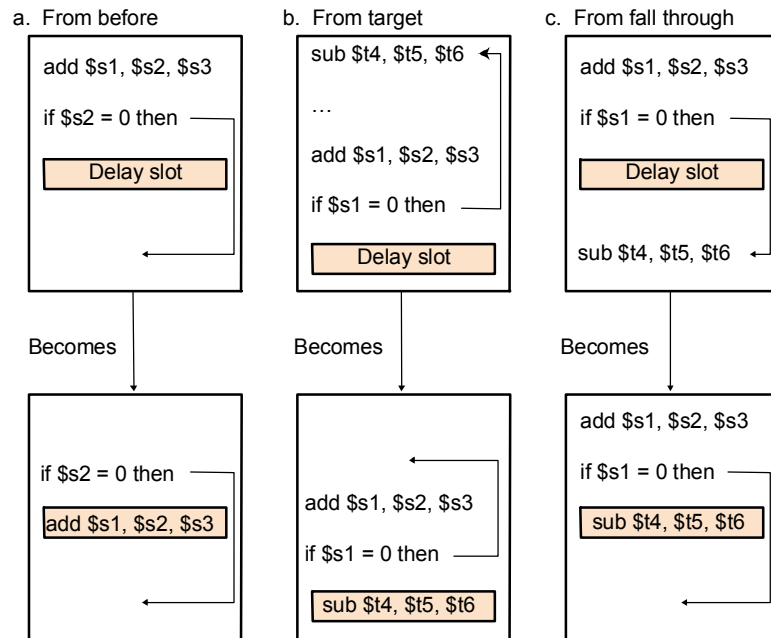
- A **tabela com a história dos saltos** contém o endereço da instrução de salto e o(s) bit(s) correspondentes à informação sobre os últimos saltos realizados
- A previsão de saltos também requer uma **tabela com os endereços de salto tomados** anteriormente, uma vez que o endereço de salto só é calculado em EXE.
- As duas tabelas têm uma implementação idêntica às *caches* e podem-se juntar numa só:



Pipelining

Resolução de dependências controlo

- Os saltos retardados, executam sempre a instrução seguinte ao salto.



- Remove a penalização do salto sempre é encontrada uma instrução para preencher o slot ($\approx 50\%$ dos casos).
- Os saltos retardados são uma solução limitada porque a sua eficiência diminui com a profundidade do *pipeline* e com a super-escalaridade.

Desempenho do *pipeline* com *stalls*

$CPI_{pipeline} = CPI_{ideal} + \text{ciclos de } stalls \text{ por instrução (CPI } stalls)$

$$Ganho = \frac{Texec_{sem\ pipeline}}{Texec_{com\ pipeline}} = \frac{\#estágios \times Tcc_{sem\ pipeline}}{(CPI_{ideal} + CPI_{stalls}) \times Tcc_{pipeline}} = \frac{\#estágios}{1 + CPI_{stalls}}$$

(CPI ideal = 1 e assumindo $Tcc_{sem\ pipeline} = Tcc_{pipeline}$)

- Exemplo: impacto da utilização de uma só porta para a memória

Máquina A – Memória *dual port*

Máquina B – Memória *single port*, relógio 1,05 vezes superior

40 % Loads

$$Ganho_B = \frac{Texec_A}{Texec_B} = \frac{(1+0) \times Tcc_A}{(1+0,4 \times 1) \times Tcc_B} = \frac{1 \times Tcc_A \times 1,05}{1,4 \times Tcc_A} = 0,75$$

A Máquina B tem pior desempenho!

Pipelining

Desempenho das variantes do processador MIPS

- Tempos acesso às unidades funcionais:

Memória (Leitura ou escrita)	ALU	Banco de registos
2 nanosegundos	2 nanosegundos	1 nanosegundo

- Mistura de instruções:

49% Tipo R	22% Load	11% Store	16% Branch	2% Jump
------------	----------	-----------	------------	---------

- Qual a diferença de desempenho entre uma implementação com *pipelining*, uma implementação multi-ciclo e uma implementação *single-cycle* com ciclo de duração fixa?
- Na versão com *pipelining* assume-se que 50% dos *loads* são utilizados na instrução seguinte, que 75% dos saltos são acertadamente previstos e que os *jump* tem sempre uma penalização de um ciclo..

Tipo de Instrução	Busca da instrução	Leitura de registos	Operação na ALU	Acesso à Memória	Escrita em registo	Total
Tipo R	2	1	2		1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2		7 ns
Branch	2	1	2			5 ns
Jump	2					

- **Single-cycle**

$$\text{CPI}=1$$

$$\text{duração de cada ciclo (Tcc)} = 8 \text{ ns}$$

- **Multi-ciclo**

$$\text{CPI} = 0,49 \times 4 + 0,22 \times 5 + 0,11 \times 4 + 0,16 \times 3 + 0,02 \times 2 = 4,02$$

$$\text{Tcc} = 2 \text{ ns}$$

- **Pipelining**

$$\text{CPI load} = 0,5 \times 1 + 0,5 \times 2 = 1,5 \text{ (50\% de stalls)}$$

$$\text{CPI branch} = 0,75 \times 1 + 0,25 \times 2 = 1,25 \text{ (25\% de previsões erradas)}$$

$$\text{CPI} = 0,49 \times 1 + 0,22 \times 1,5 + 0,11 \times 1 + 0,16 \times 1,25 + 0,02 \times 2 = 1,17$$

$$\text{Tcc} = 2 \text{ ns}$$

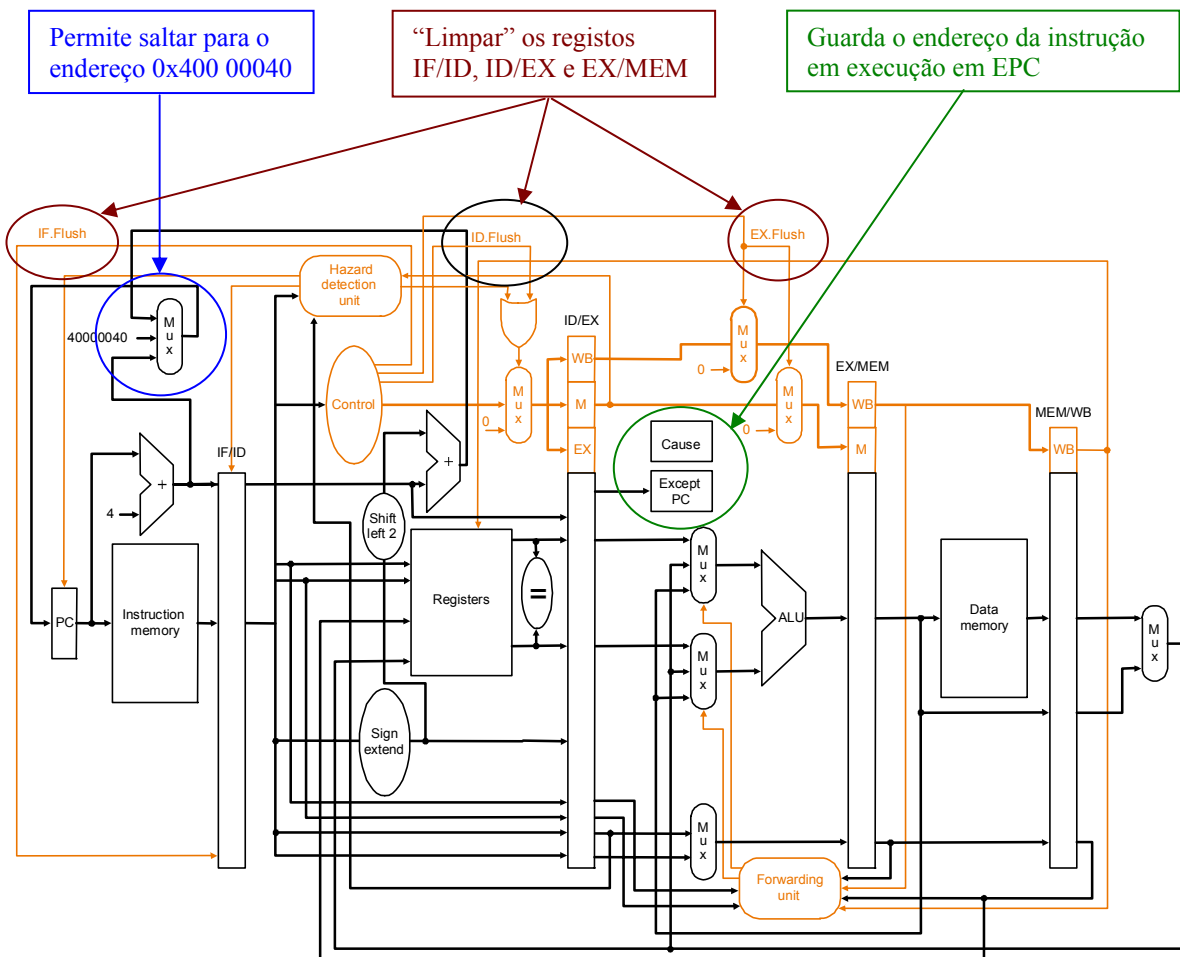
$$\text{Ganho Single-cycle/pipeline} = \#I \times 1 \times 8 / \#I \times 1,17 \times 2 = 3,42x$$

$$\text{Ganho Multi-ciclo/pipeline} = 4,02 / 1,17 = 3,44x$$

Pipelining

● Suporte a exceções

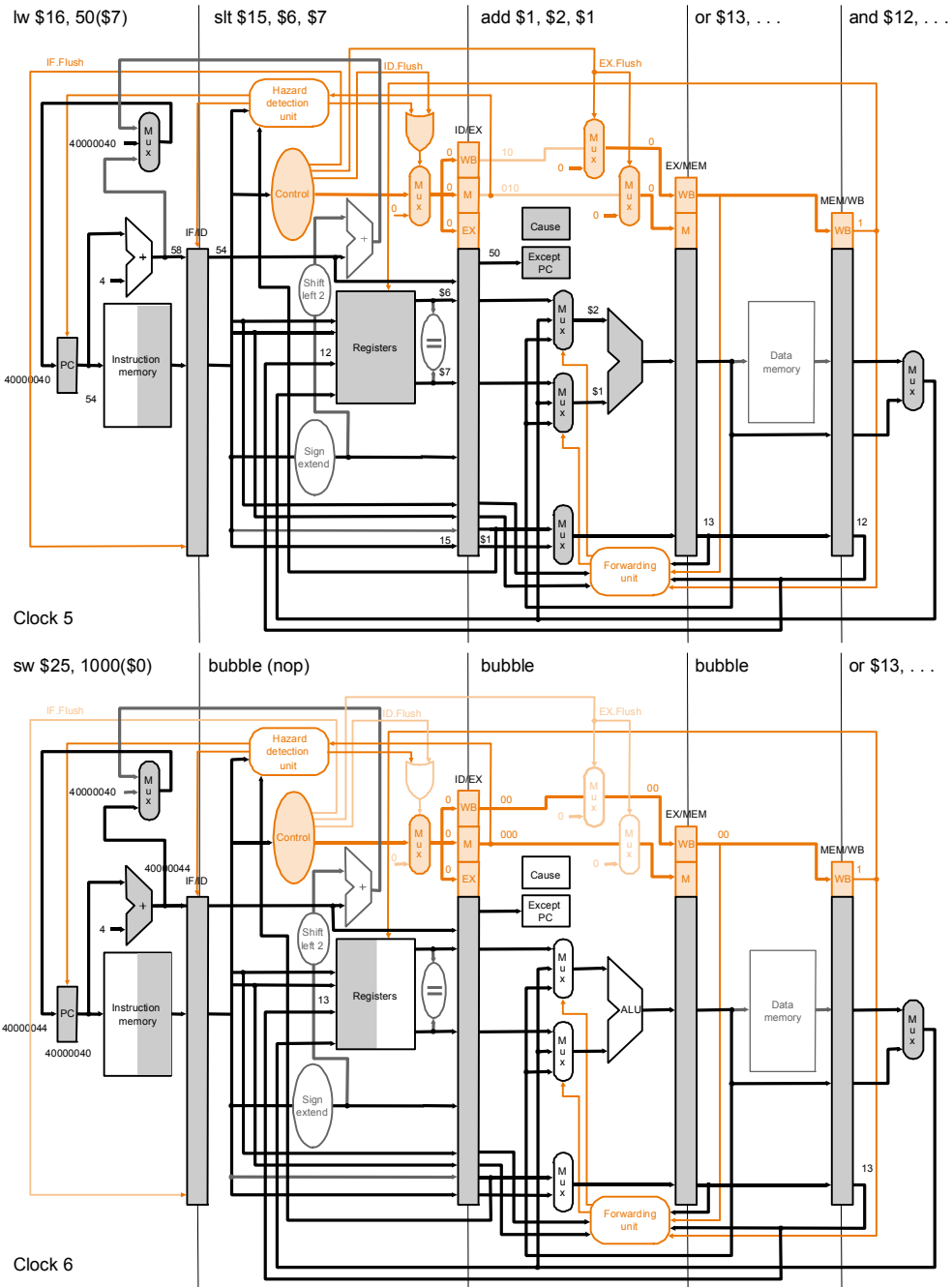
- Quando ocorre uma exceção o controlo é transferido para o programa no endereço de memória 0x4000040
- É necessário “limpar” os registos do *pipeline* que contêm instruções parcialmente executadas, posteriores à instrução onde ocorre a excepção.
- O valor actual do PC deve ser guardado em EPC
- DP para suporte a uma excepção aritmética (por exemplo em *add*):



- O tratamento de exceções é complexo, uma vez que estas podem ocorrer em vários estágios simultaneamente, obrigando a um esquema de prioridades
- Algumas arquitecturas implementam exceções imprecisas, onde o EPC pode não indicar exactamente a instrução que provocou a excepção, deixando essa responsabilidade ao sistema operativo

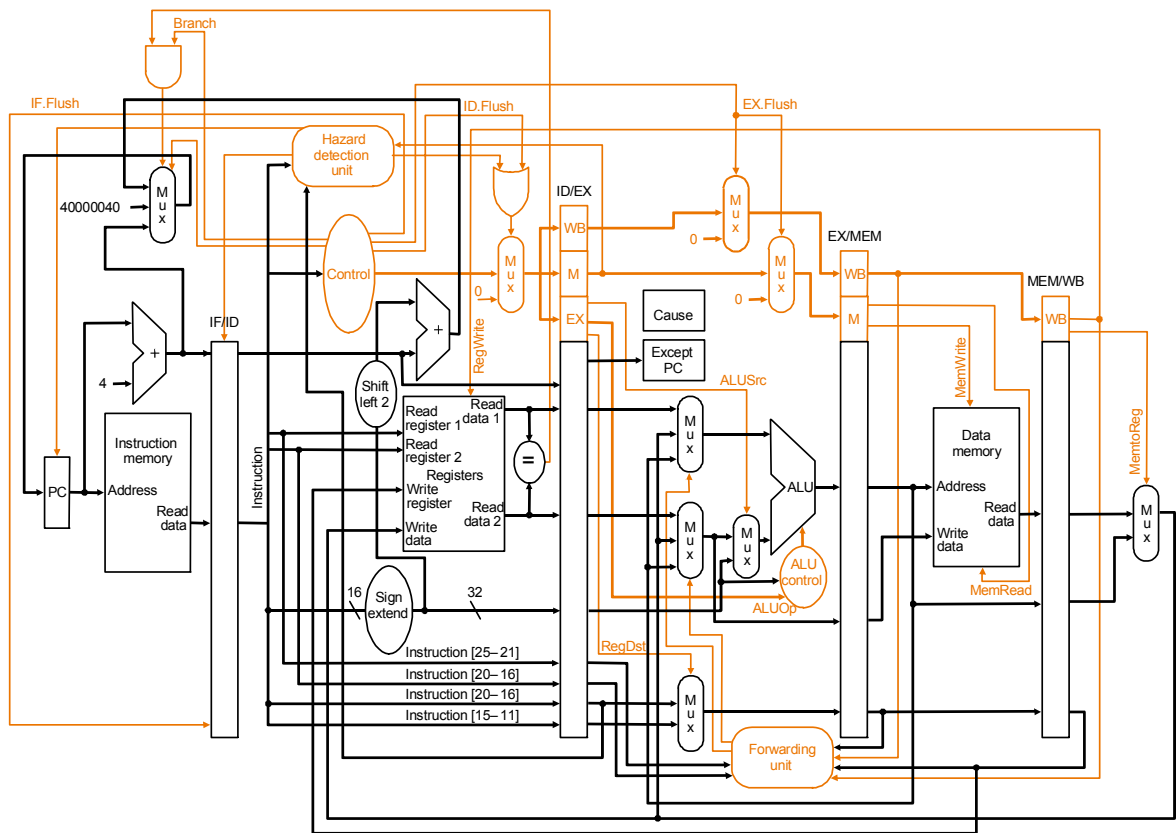
Pipelining

Exemplo de processamento de interrupções



Pipelining

- Figura completa do *datapath*



Bibliografia

Matéria	Bibliografia
Introdução a <i>pipelining</i>	COD2e 6.1
<i>Datapath</i> com <i>pipelining</i> e respectiva unidade de controlo	COD2e 6.2, 6.3
Anomalias de dados e de controlo e sua resolução	COD2e 6.4, 6.5, 6.6
Tratamento de exceções	COD2e 6.7

COD2e *Computer Organization and Design: the hardware/software interface*, D. Patterson and J. Hennessy, Morgan Kaufmann, 2ª edição, 1998.

CA3e *Computer Architecture: A Quantitative Approach*, J. Hennessy and D. Patterson, Morgan Kaufmann, 3ª edição, 2002.