CAPÍTULO 4
PIPELINE

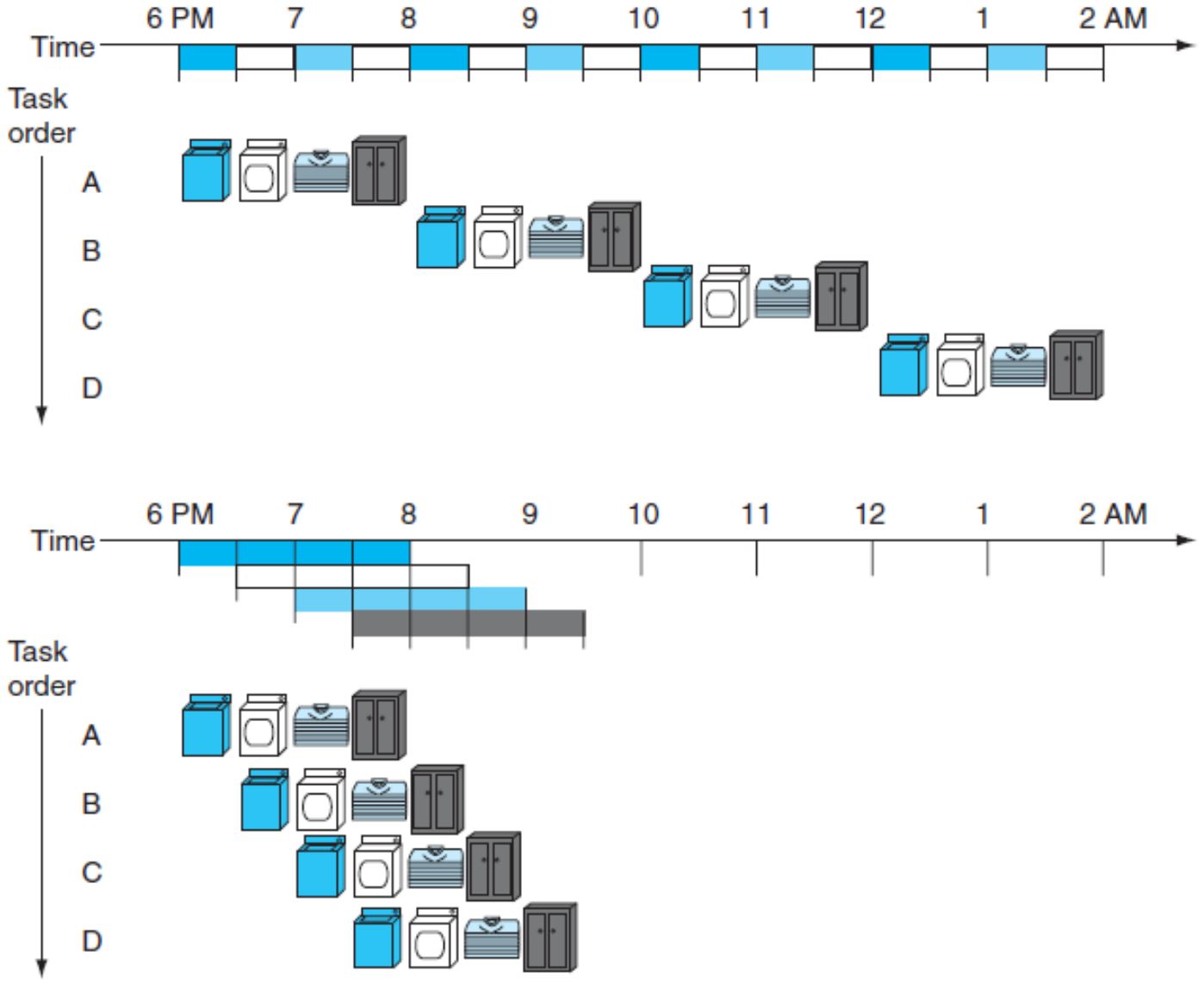
- Introdução
- Ciclo único versus pipeline
- Projetando o conjunto de instruções
- Pipeline *Hazards*
- Caminho de dados usando o pipeline
- Diagramas de Pipeline
- Controle do Pipeline
- Hazard de dados e forwarding
- Detecção do Hazard
- Hazard de dados e forwarding
- Hazard de dados e stalls
- Hazard de desvios
- Caminho de dados completo
- Exceções
- Exercícios

Introdução

- O Pipeline é uma técnica de paralelismo de instruções
- Consiste em permitir a execução de instruções sobrepostas
- É uma técnica fundamental para tornar os processadores mais rápidos
- A ideia fundamental é análoga à execução das tarefas em uma lavanderia de roupas, numeradas como etapas do processo:

1. Colocar a roupa suja na lavadora
2. Transferir a roupa lavada da lavadora para a secadora
3. Levar a roupa seca para a mesa de passar
4. Guardar a roupa no armário.

Introdução


6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D
6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

Introdução

- Os mesmos princípios se aplicam a processadores. No MIPS as instruções são divididas normalmente em cinco etapas:

1. Buscar instrução na memória
2. Identificar a instrução e ler registradores
3. Executar a operação ou calcular um endereço
4. Acessar um operando na memória de dados
5. Escrever o resultado em um registrador

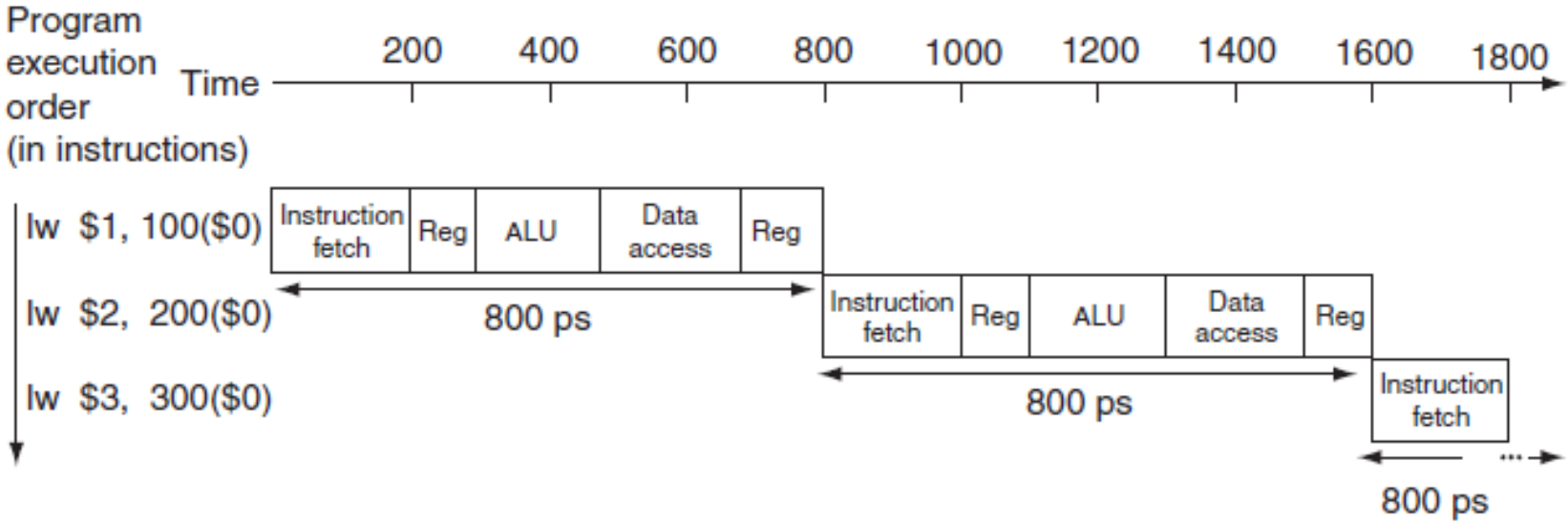
Ciclo único versus pipeline

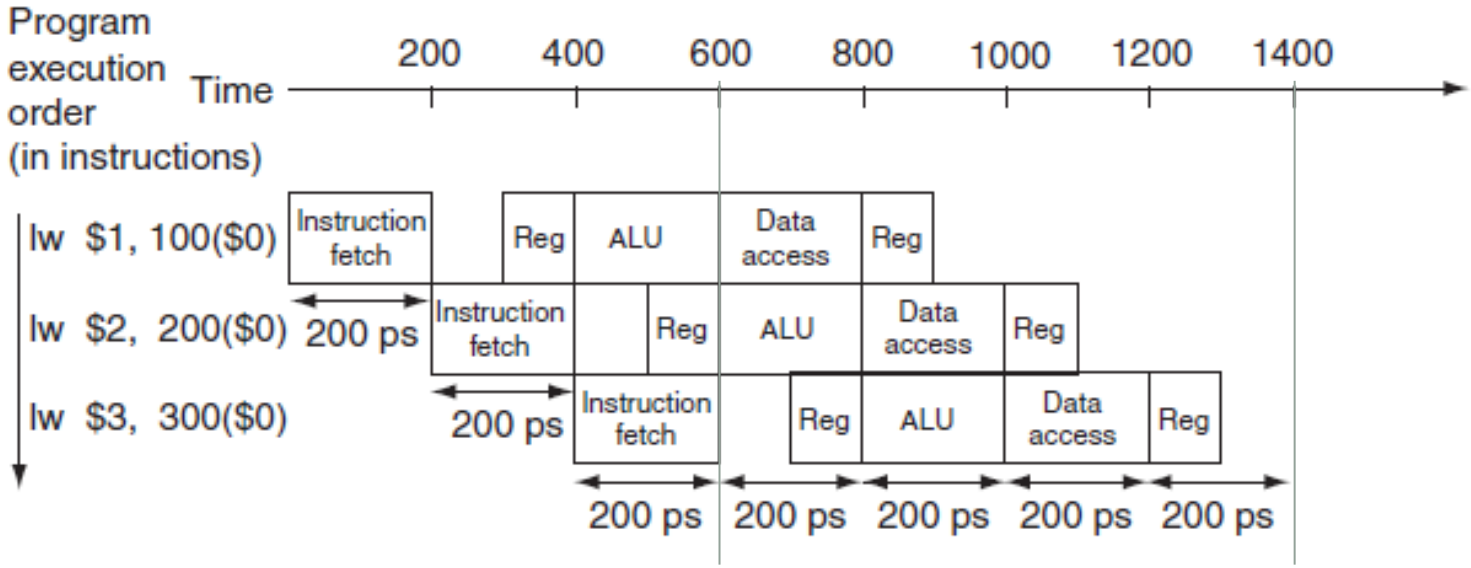
- Vamos comparar uma implementação de ciclo único com uma com pipeline para a execução de três instruções load, com base na tabela:

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

- Na implementação de ciclo único, cada ciclo deve ser de 800ps. Tempo total dos três lw: 2400ps
- Com pipeline, o ciclo de clock pode ser 200ps (da tabela) e o tempo de execução passa a ser de 1400ps

Ciclo único versus pipeline





Ciclo único versus pipeline

- O tempo entre a primeira e a quarta instrução foi de 600ps, 4 vezes menos que no caso ciclo único
- O tempo total não reduziu 4 vezes (de 2400 para 1400) porque o número de instruções é pequeno
- Vamos acrescentar 1.000.000 instruções ao exemplo.
  - No caso monociclo: tempo total = 1000000×800+2400 (ps)
  - No caso com pipeline: tempo total = 1000000×200+1400 (ps)

$$\frac{800002400ps}{200001400ps} \approx 4{,}00 = \frac{800us}{200us}$$

- No caso ótimo, em condições ideais, a melhoria é o número de estágios do pipeline

$$tempo\ entre\ instruções_{com\ pipeline} = \frac{tempo\ entre\ instruções_{sem\ pipeline}}{numero\ de\ estágios\ do\ pipeline}$$

Projetando o conjunto de instruções

- Alguns cuidados que o MIPS toma para implementar o pipeline com maior facilidade são:

1. Instruções de mesmo tamanho
2. Poucos formatos de instruções e simetria. Permite por exemplo ler os registradores antes de ter decodificado a instrução
3. Somente load ou store acessam a memória. Se um add acessasse a memória teriam que ser adicionados novos estágios
4. Transferências sempre em um único ciclo (dados alinhados)

Pipeline Hazards

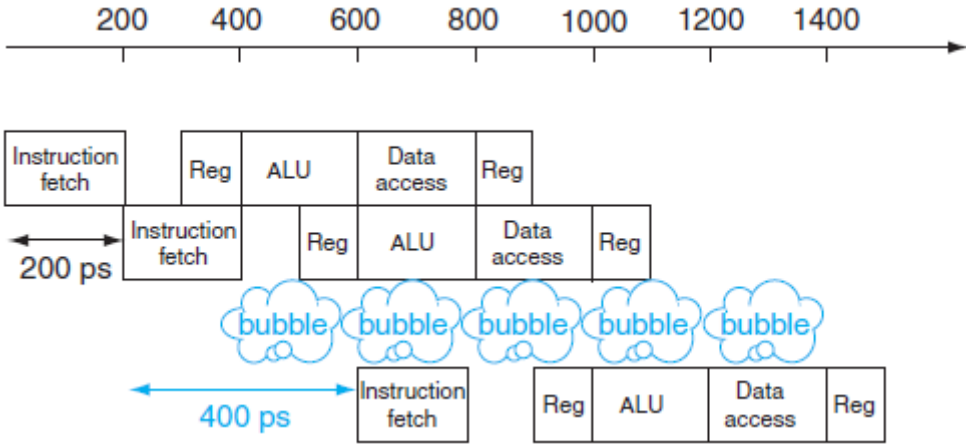
- Hazards são situações em que a próxima instrução não pode ser executada no ciclo de clock seguinte. Existem três tipos de hazards:
- **Hazards estruturais**
  - Conflito de unidades funcionais
- **Hazards de dados**
  - Ocorre quando uma etapa precisa esperar até que outra seja executada. Por exemplo:

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

  - É necessário uma intervenção para que ocorra o resultado correto da operação
- **Hazards de controle**
  - O que ocorre em desvios condicionais e incondicionais

Pipeline Hazards

- Uma solução é o chamado **“stall”** (ou bolha) onde se deixa o pipeline vazio até o conflito ser resolvido:

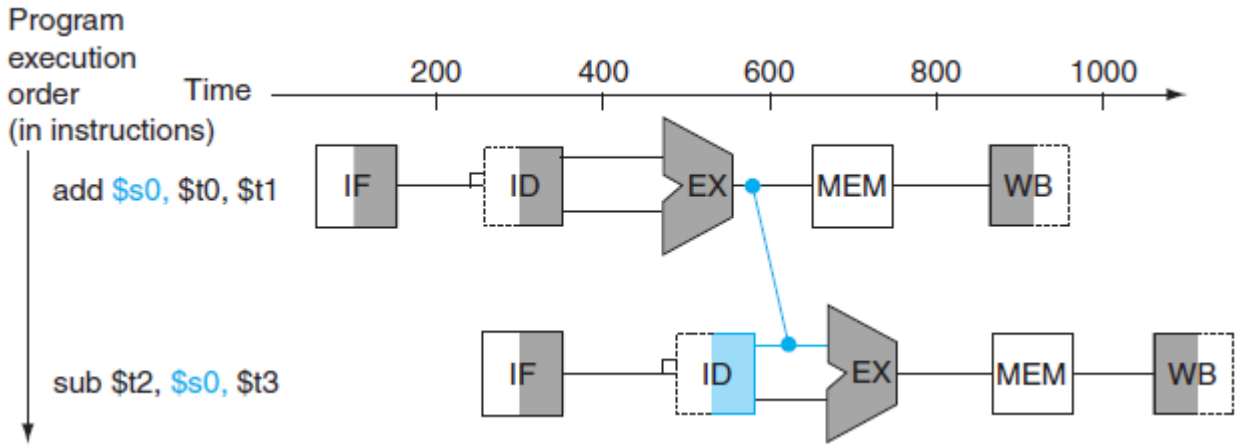


- Porém isso é feito em conjunto com a técnica de **forwarding** ou **bypassing**, que consiste em adicionar hardware extra para entregar um item para um estágio posterior, antecipadamente.

Pipeline Hazards

- No hazard de dados do exemplo anterior:

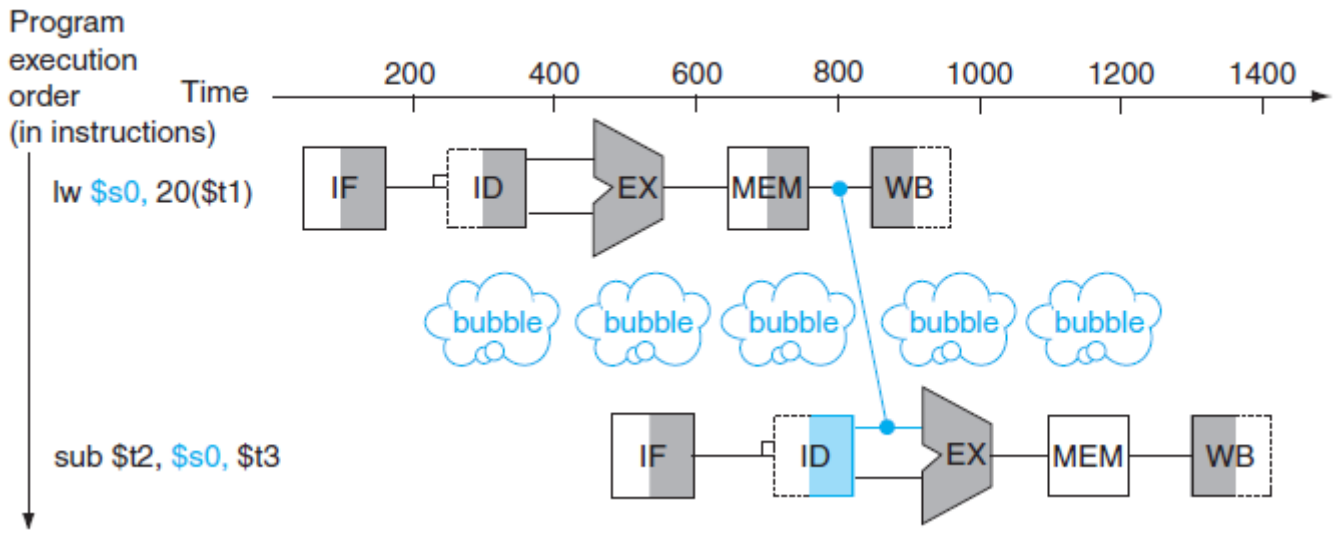
```
add $s0, $t0, $t1
add $t2, $s0, $t3
```



- A conexão mostra o caminho do **forwarding** desde a saída do estágio EX de add até a entrada do estágio EX para sub, substituindo o valor do registrador $s0 lido no segundo estágio de sub

Pipeline Hazards

- O hazard de dados é mais grave no caso do uso da instrução load
- No caso do load, o dado só está disponível após o 4º estágio.
- Neste caso o “stall” é inevitável



Pipeline Hazards

- Uma outra forma de combater os hazards é a reordenação do código.
- Considere o seguinte segmento de código em linguagem de alto nível:

```
A = B + E;
C = B + F;
```

- Código MIPS, assumindo acesso às variáveis via $t0:

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

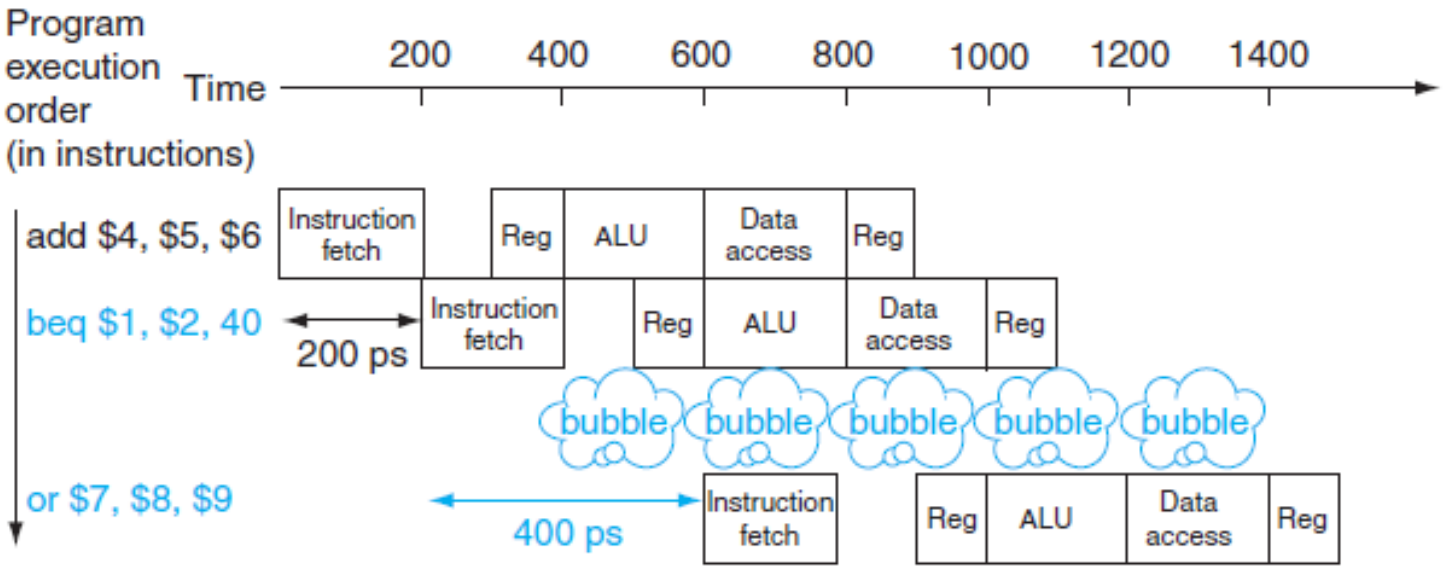
Pipeline Hazards

- Com a ordem de execução alterada, o “stall” após o uso do load pode ser evitado.
- Demais hazards são resolvidos por bypassing.
- Na sequência abaixo, são necessários dois ciclos a menos.

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
lw      $t4, 8($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

Pipeline Hazards

- **Hazards de controle**
  - Numa instrução de controle (desvios e jumps) qual será a próxima instrução a ser buscada?
  - No segundo estágio do pipeline, a decisão do *branch* ainda não é conhecida



Pipeline Hazards

Desempenho alterado pelos “stalls” no desvio

Problema:

Estime o impacto nos ciclos de clock por instrução (CPI) do stall nos desvios. Suponha que todas as outras instruções tem CPI=1

Resposta:

- Vamos supor que 13% das instruções são desvios (para o SPECint2000).
- Outras instruções: CPI=1
- Supondo 100 instruções, serão consumidos 113 ciclos de clock
- Portanto o CPI médio é 1,13
- O programa fica 1,13 vezes mais lento que o caso ideal

Pipeline Hazards

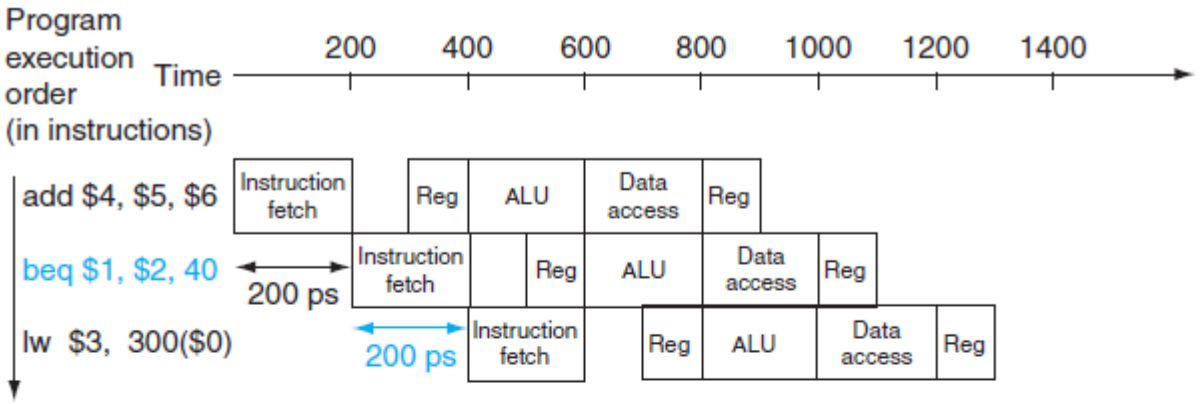
- Uma outra solução para os hazards de controle: **prever** se o desvio será tomado ou não tomado
- Três técnicas:

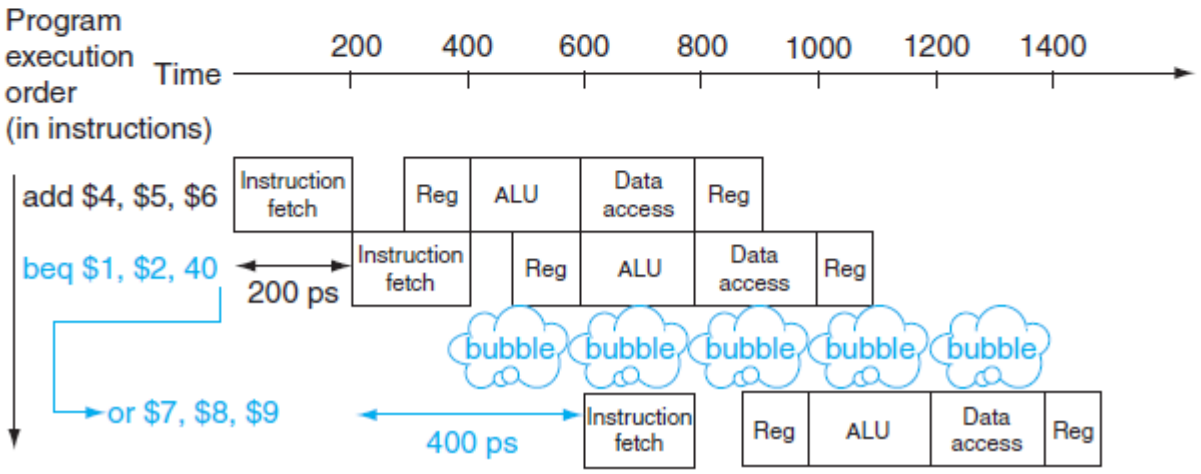
1. Assumir desvios tomados
2. Assumir desvios não tomados
3. Previsão dinâmica que muda a previsão de acordo com cada execução

- O “stall” ocorre quando a decisão tomada está errada.
- Neste caso o pipeline tem que garantir que a instrução errada não terá efeito

Pipeline Hazards

Assumindo desvios não tomados





Pipeline Hazards

- Foram abordadas duas técnicas: O “stall” após um desvio em todos os casos e a previsão de desvio (tomado, não tomado ou dinâmico)
- Existe ainda uma terceira técnica: **desvio adiado (ou delayed branch)** que é a adotada no MIPS
- Nesta técnica, uma instrução anterior ao desvio e que não afeta o desvio é colocada depois do desvio
- Nos exemplos anteriores, a instrução add não afeta o desvio e se colocada após a instrução de desvio, dará tempo suficiente de se decidir se o desvio será tomado ou não

Observações sobre Pipeline

- Pipeline é uma técnica de ILP invisível ao programador
- Aumenta a vazão de instruções, mas não reduz o tempo de execução de uma única instrução (latência)
- Por exemplo, no pipeline de 5 estágios, cada instrução ainda leva 5 ciclos de clock
- A ISA pode simplificar ou complicar a implementação eficiente de um pipeline

Exercício

Para cada sequência de código abaixo, indique se deverá haver “stall”, se o hazard pode ser resolvido somente com forwarding ou se não há hazard algum.

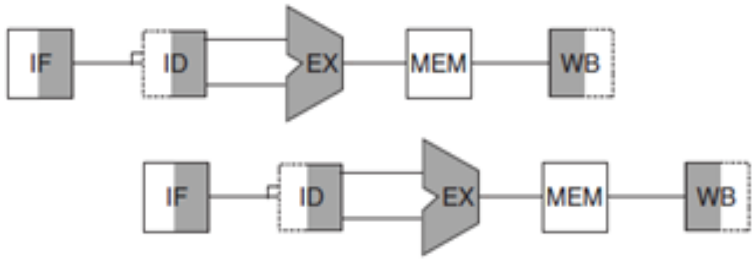
Sequence 1	Sequence 2	Sequence 3

`lw $t0,0($t0)`
`add $t1,$t0,$t0`
	`add $t1,$t0,$t0`
	`addi $t2,$t0,#5`
	`addi $t4,$t1,#5`
		`addi $t1,$t0,#1`
		`addi $t2,$t0,#2`
		`addi $t3,$t0,#2`
		`addi $t3,$t0,#4`
		`addi $t5,$t0,#5`

Exercícios

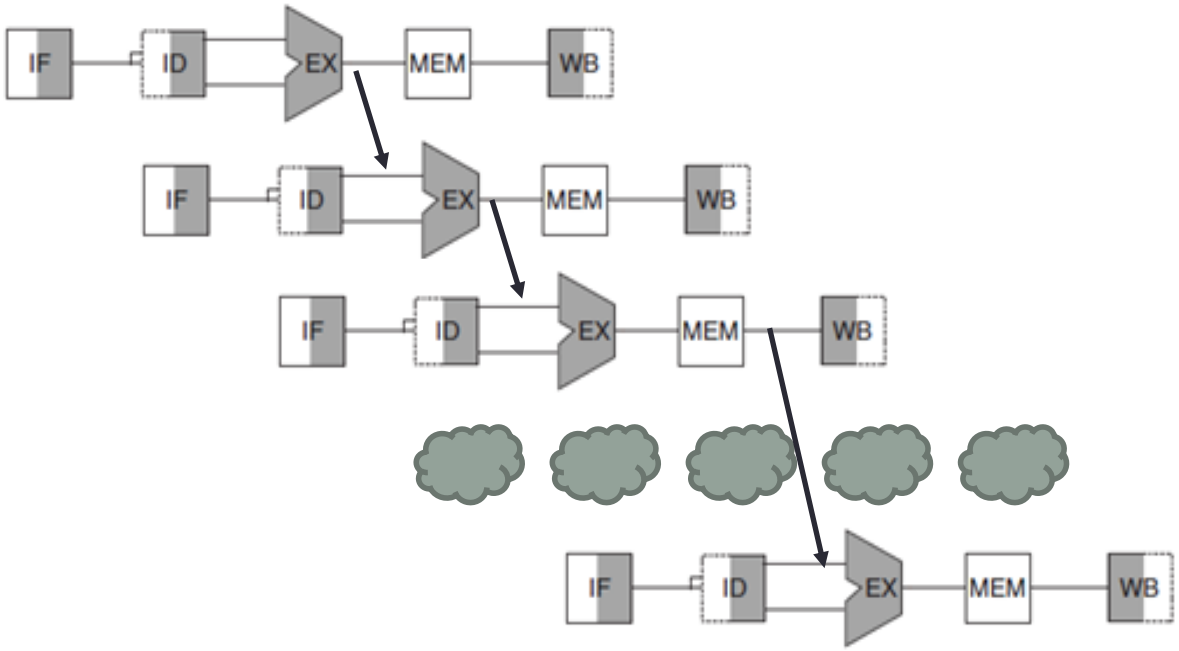
Usando um desenho semelhante ao da figura, mostre os stalls e caminhos de forwarding para executar as quatro instruções abaixo:

```
add $3, $4, $6
sub $5, $3, $2
lw $7, 100($5)
add $8, $7, $2
```



Exercícios

```
add $3, $4, $6
sub $5, $3, $2
lw $7, 100($5)


add $8, $7, $2
```

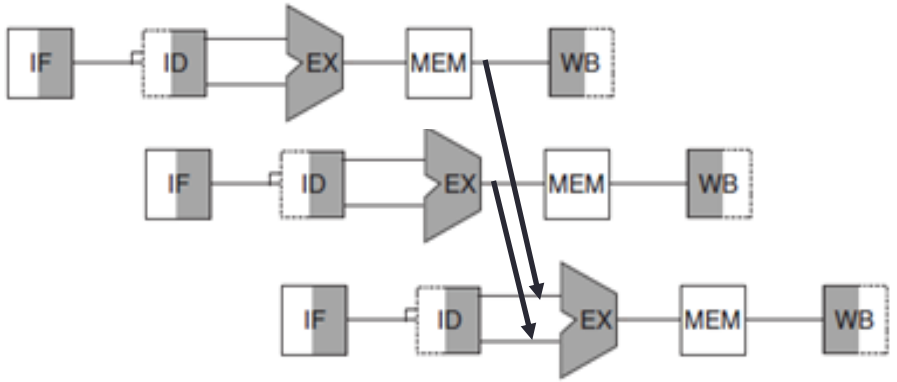


Exercícios

E se entre as instruções `lw $7,100($5)` e `add $8,$7,$2` colocássemos uma instrução que modifica o registrador $2, por exemplo `add $2,$4,$6` ?

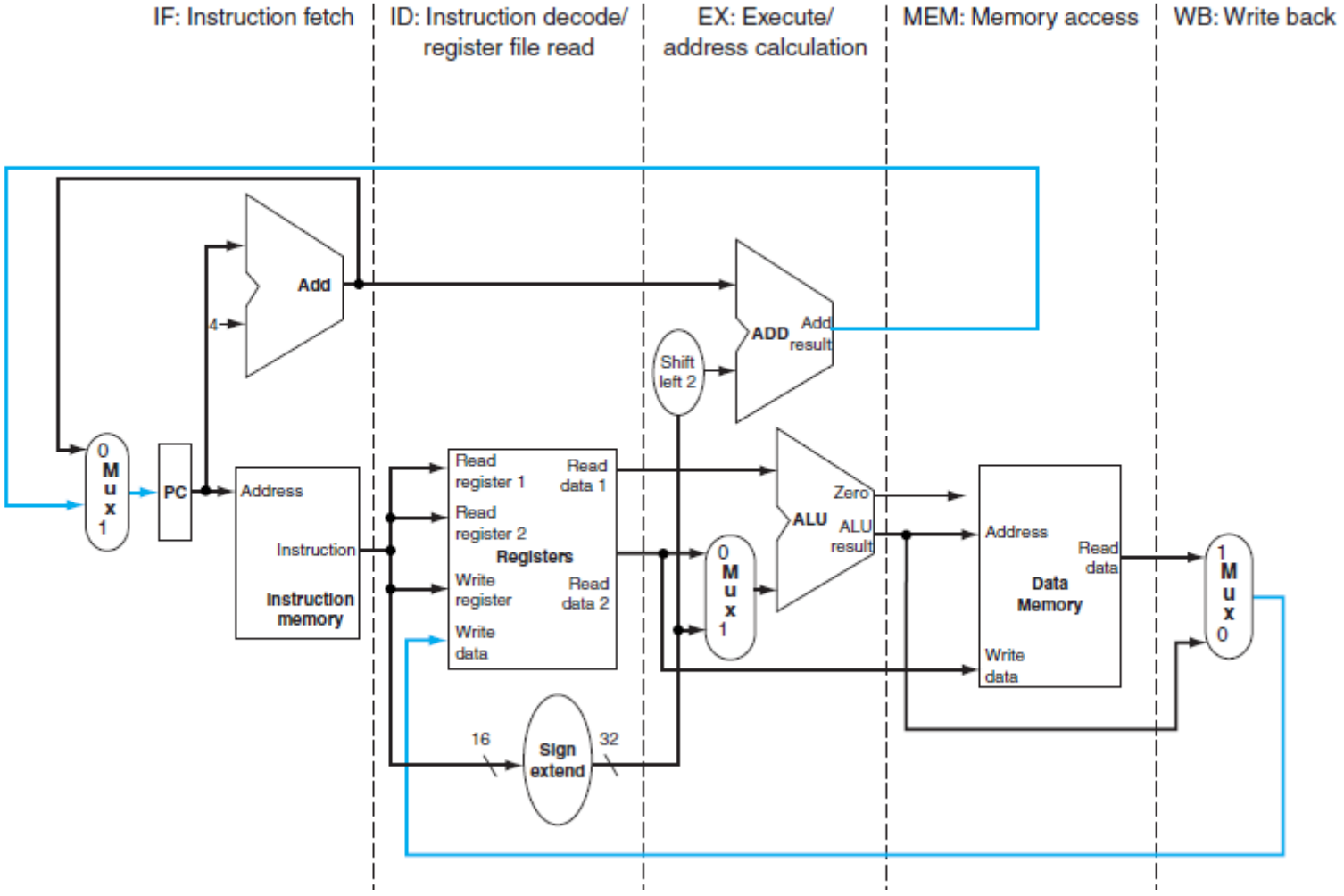
Teríamos dois *forwardings* na mesma instrução:

```
lw $7,100($5)
add $2,$4,$6
add $8,$7,$2
```



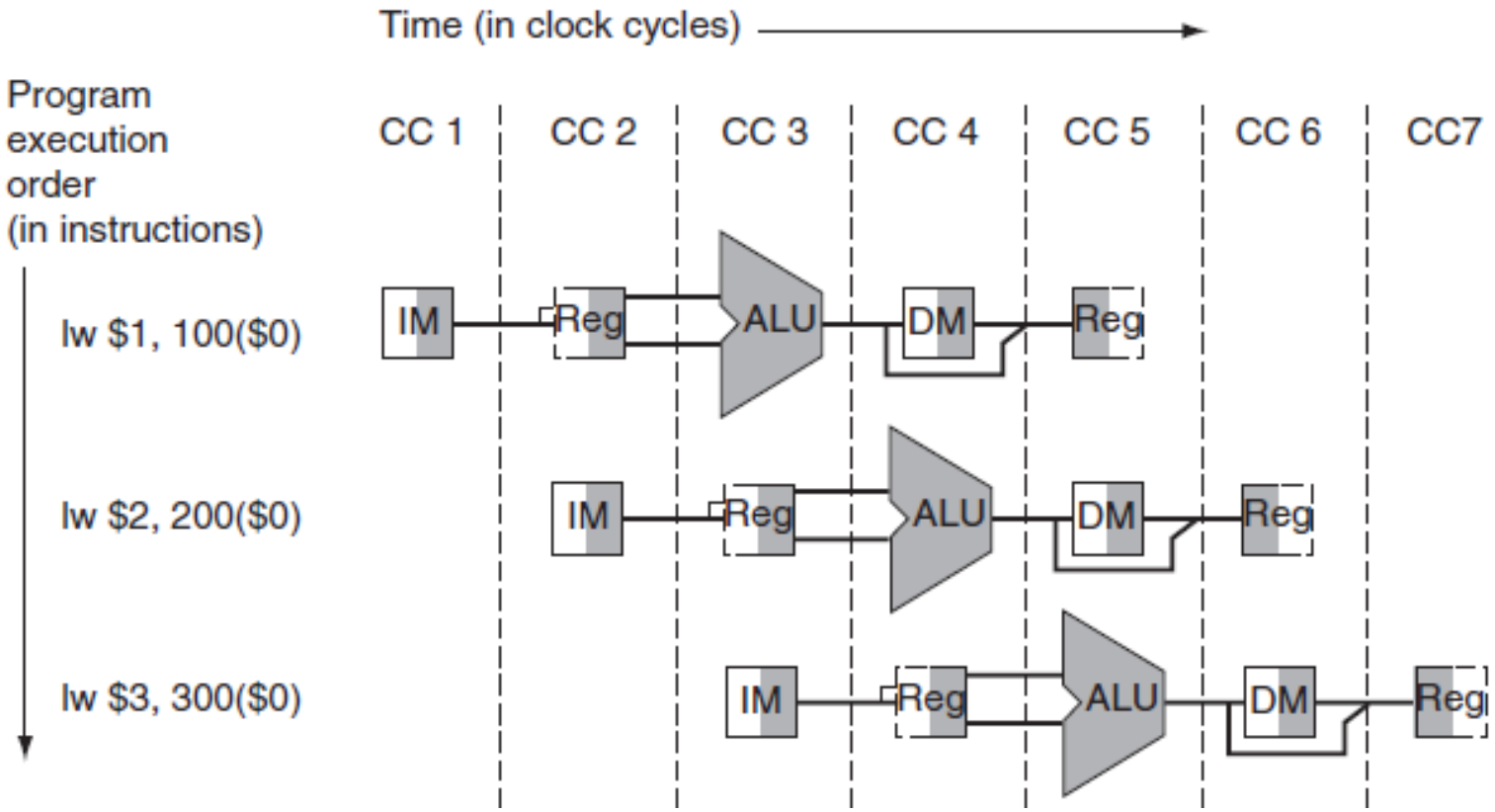
Caminho de dados usando o pipeline

- Para um pipeline de 5 estágios, devemos dividir o caminho de dados de ciclo único em cinco partes



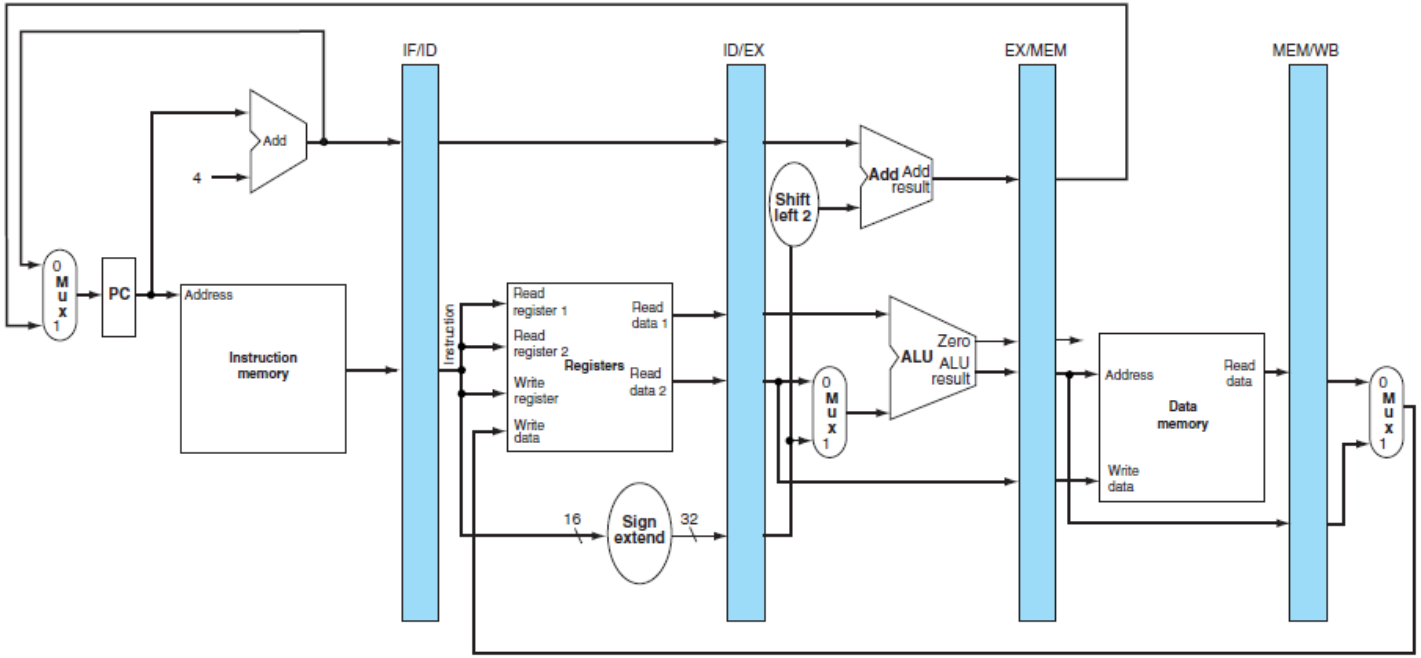
Caminho de dados usando o pipeline

- Para analisarmos o pipeline, consideramos que cada instrução tem um caminho de dados próprio:



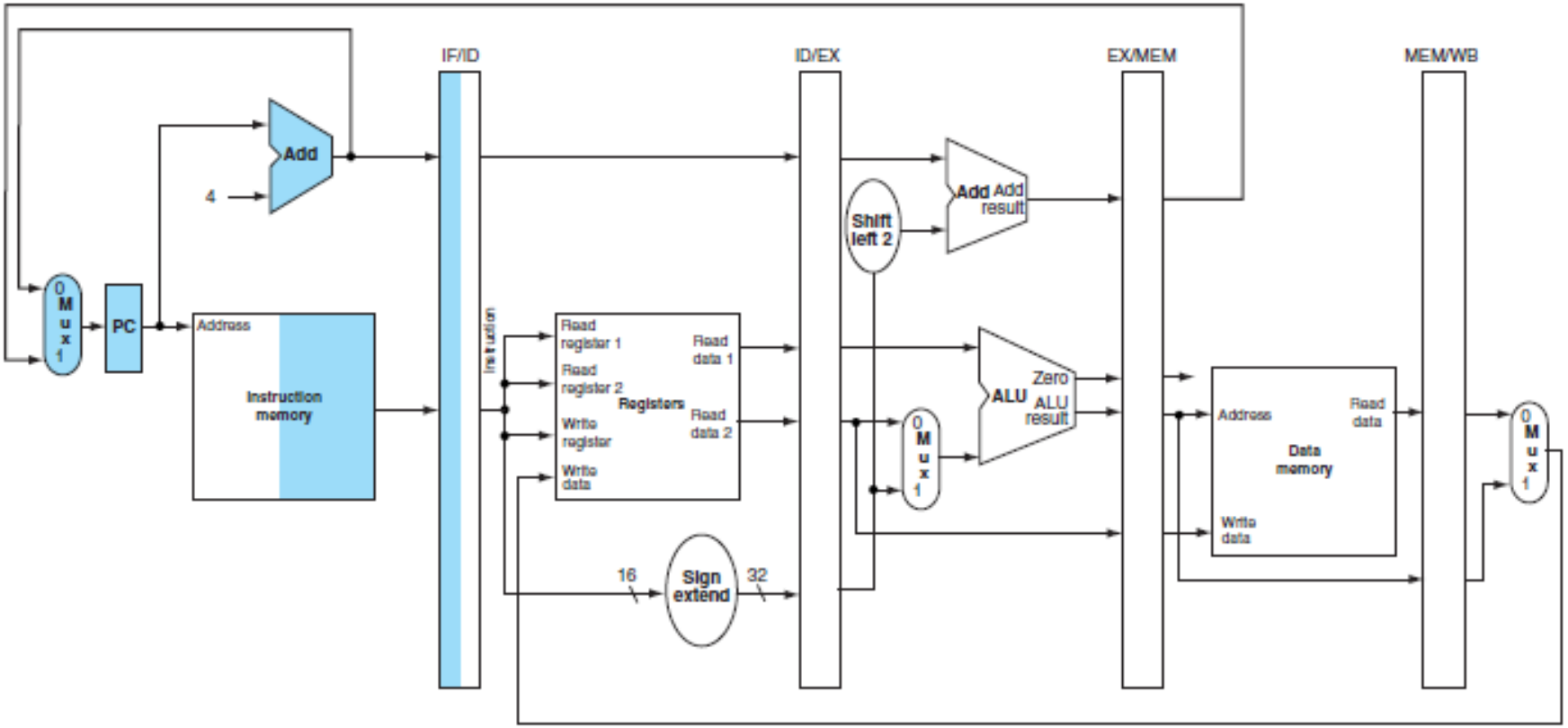
Caminho de dados usando o pipeline

- Cada divisão entre etapas deve conter registradores para reter os resultados de um ciclo enquanto processam o ciclo seguinte (similar à implementação multiciclo)
- Não é necessário um registrador após o estágio WB
- Os registradores são de 64, 128, 97 e 64 bits, para IF/ID, ID/EX, EX/MEM, MEM/WB respectivamente.



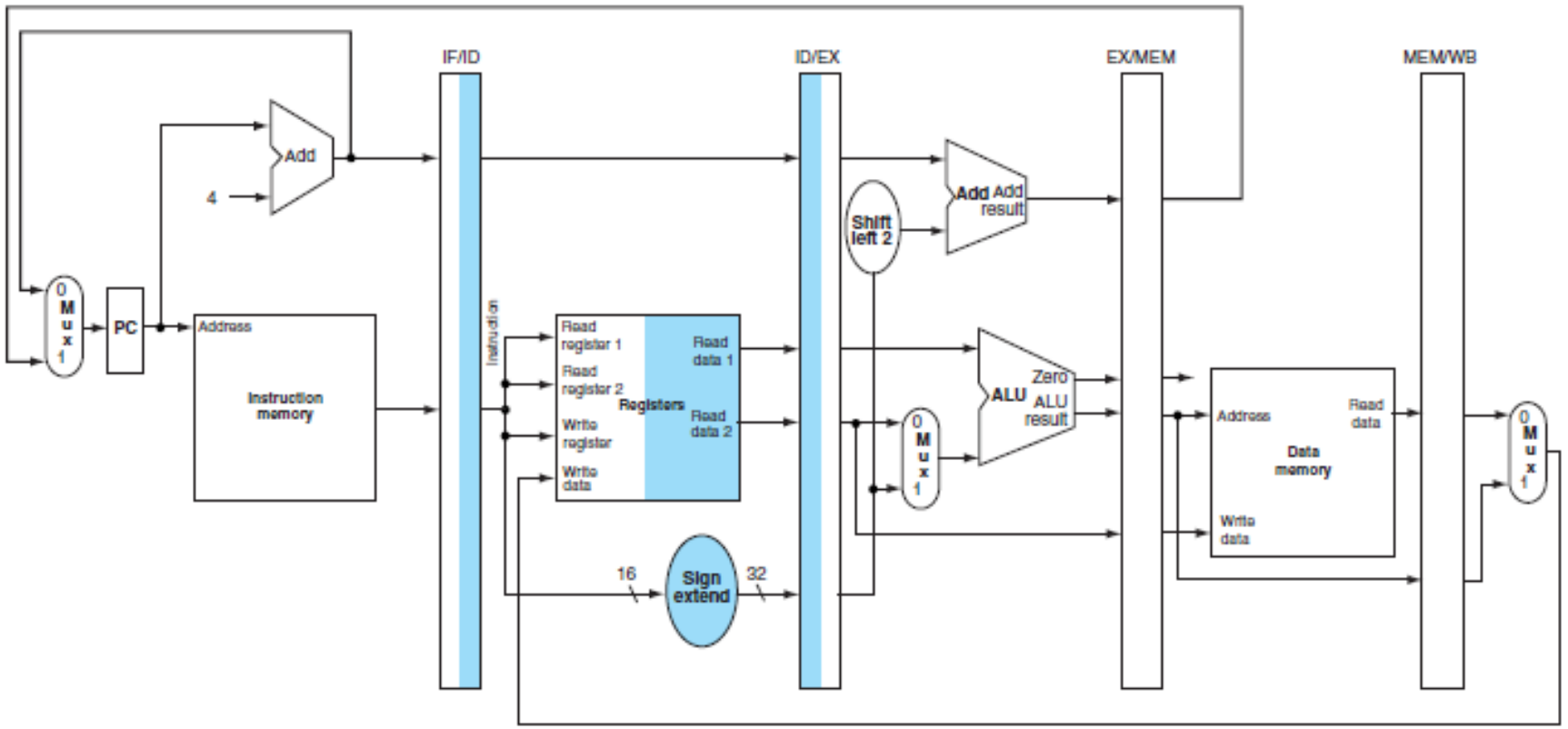
Caminho de dados usando o pipeline

- A sequência a seguir mostra as partes ativas a cada estágio para uma instrução load
- 1º estágio (IF)



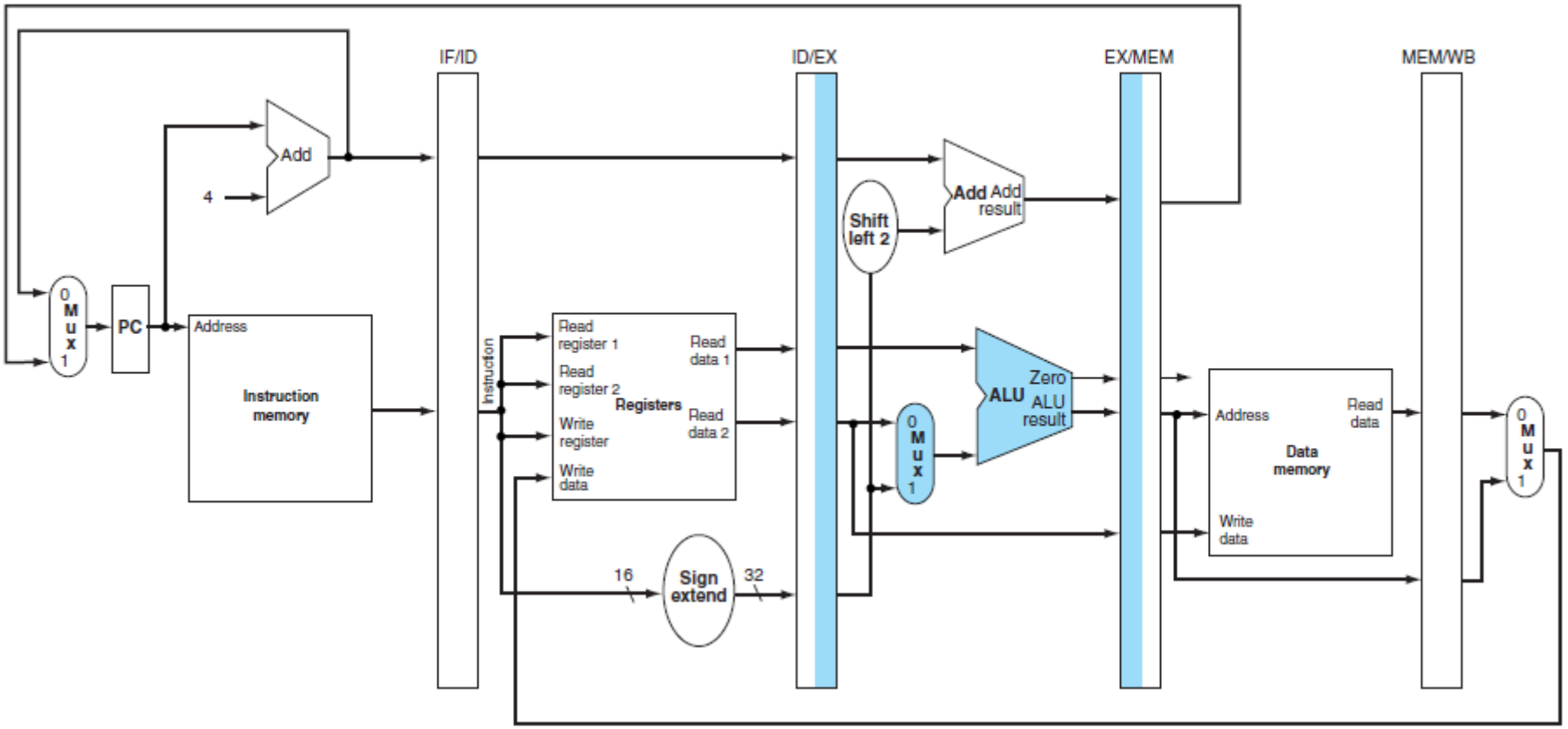
Caminho de dados usando o pipeline

- 2º estágio (ID) para o **load**:



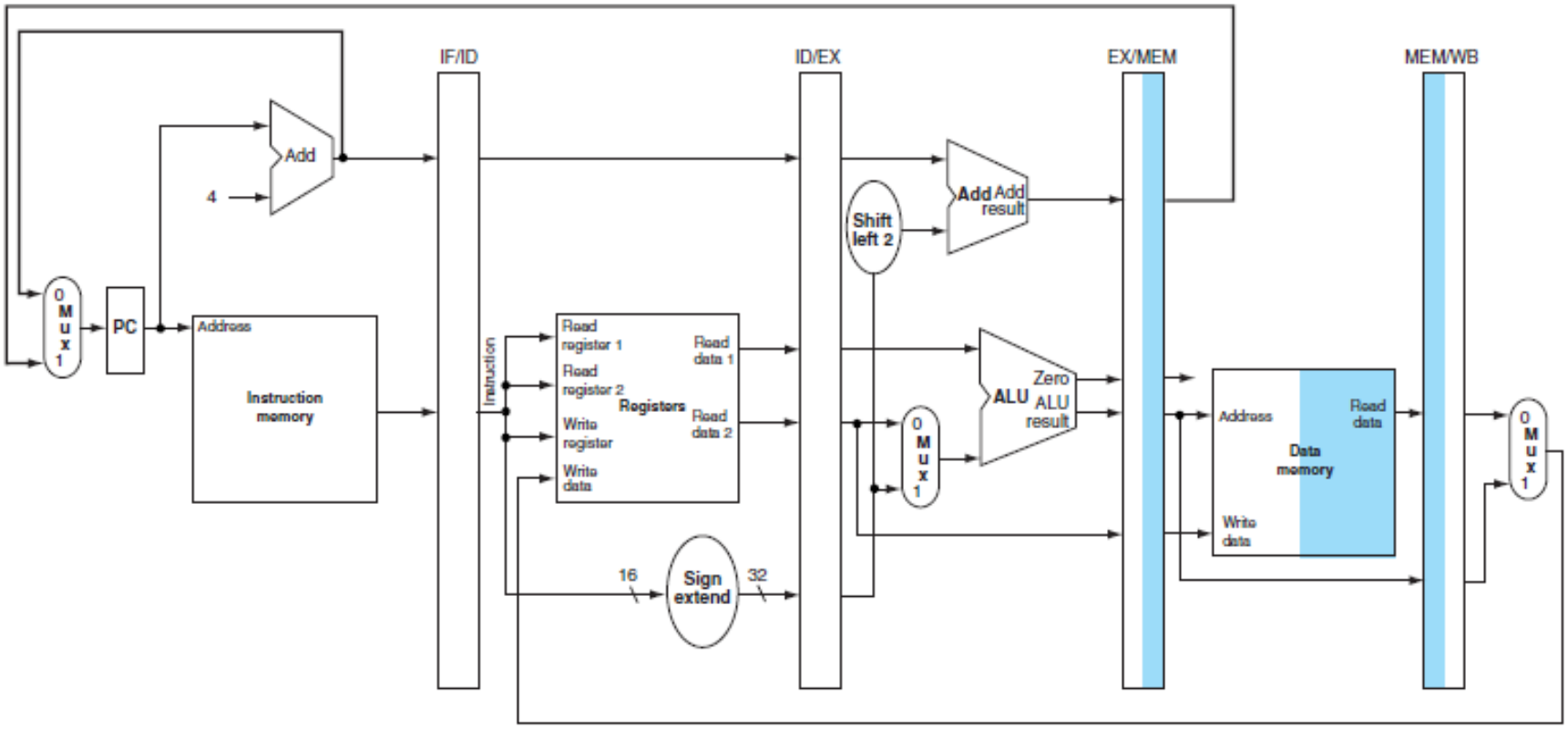
Caminho de dados usando o pipeline

- 3º estágio (EX) para o **load**:



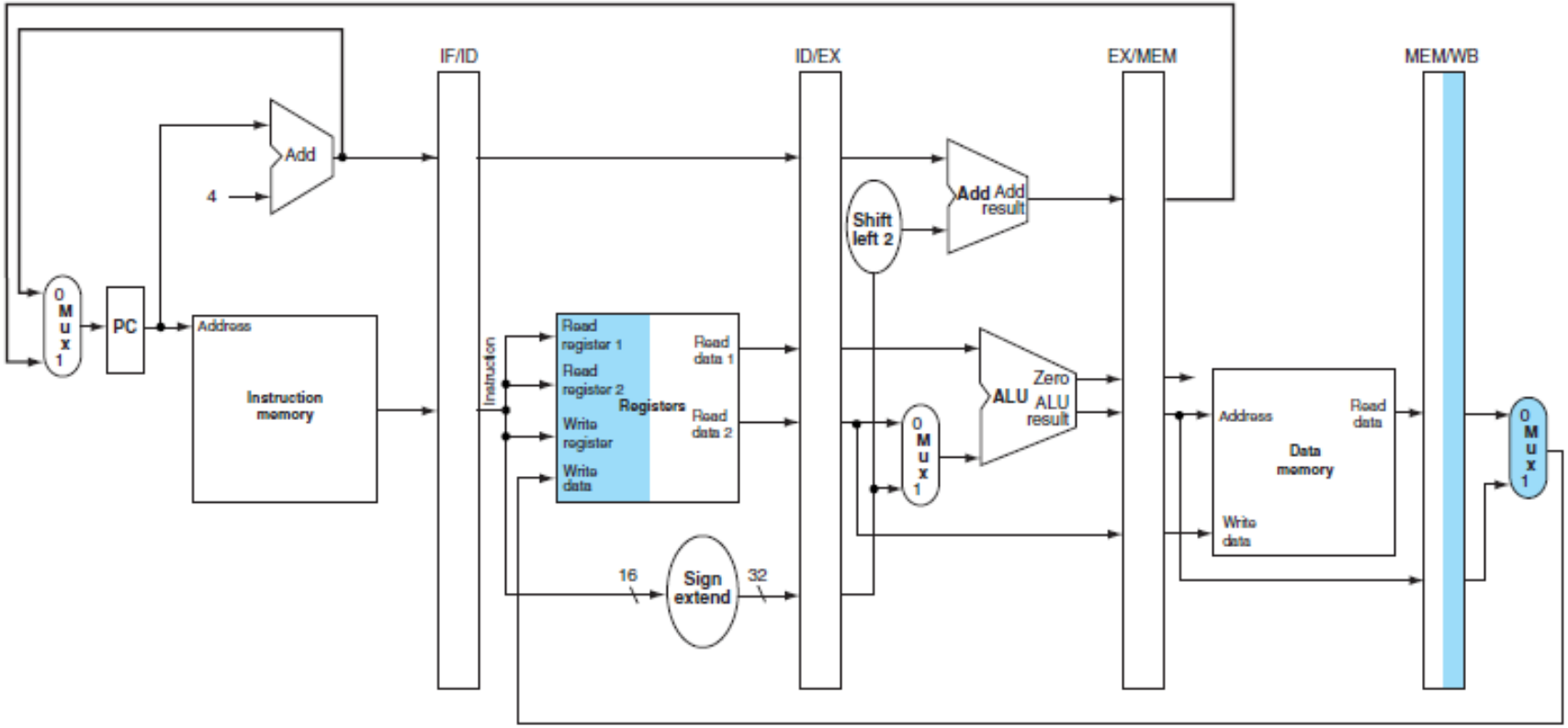
Caminho de dados usando o pipeline

- 4º estágio (MEM) para o **load**:



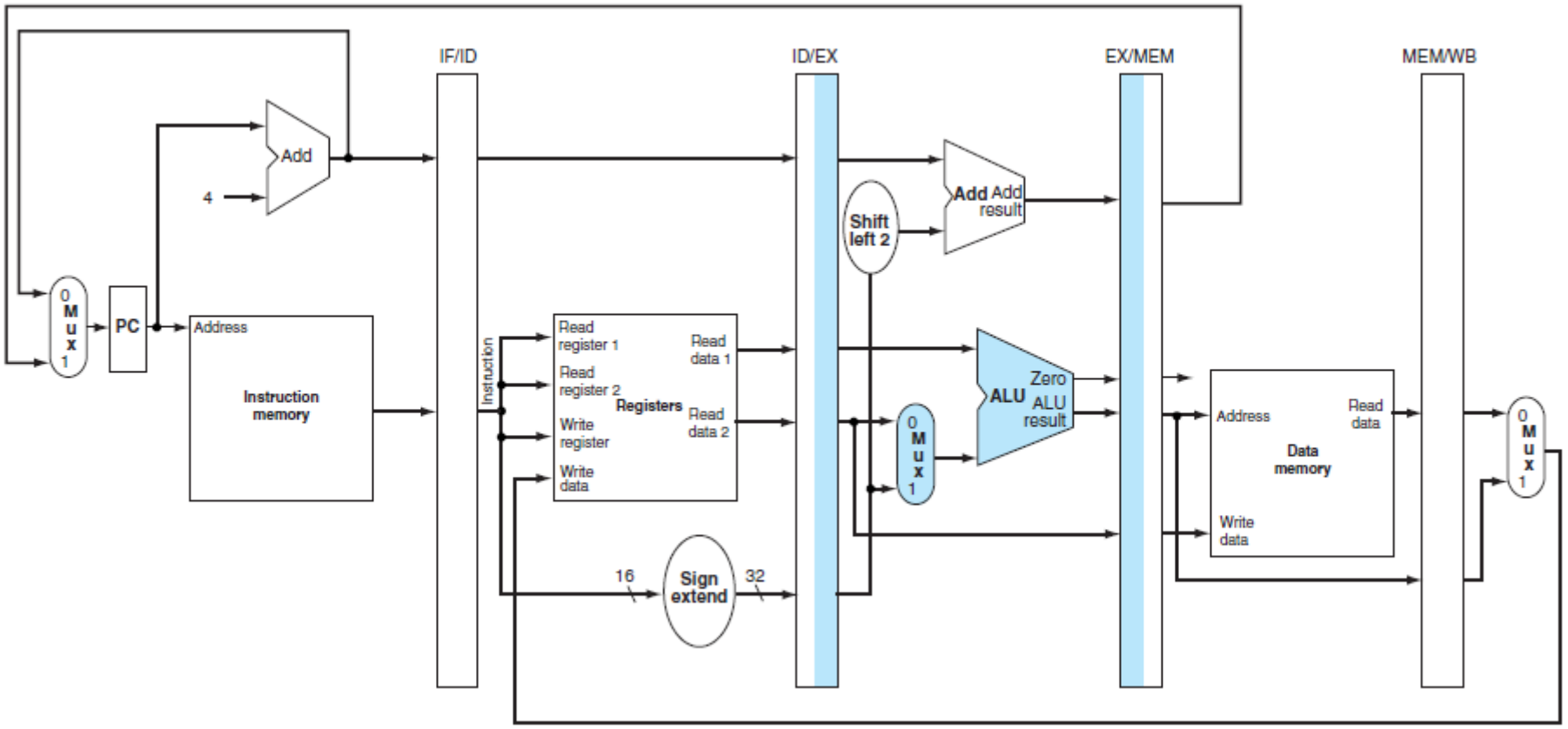
Caminho de dados usando o pipeline

- 5º estágio (WB) para o **load**:



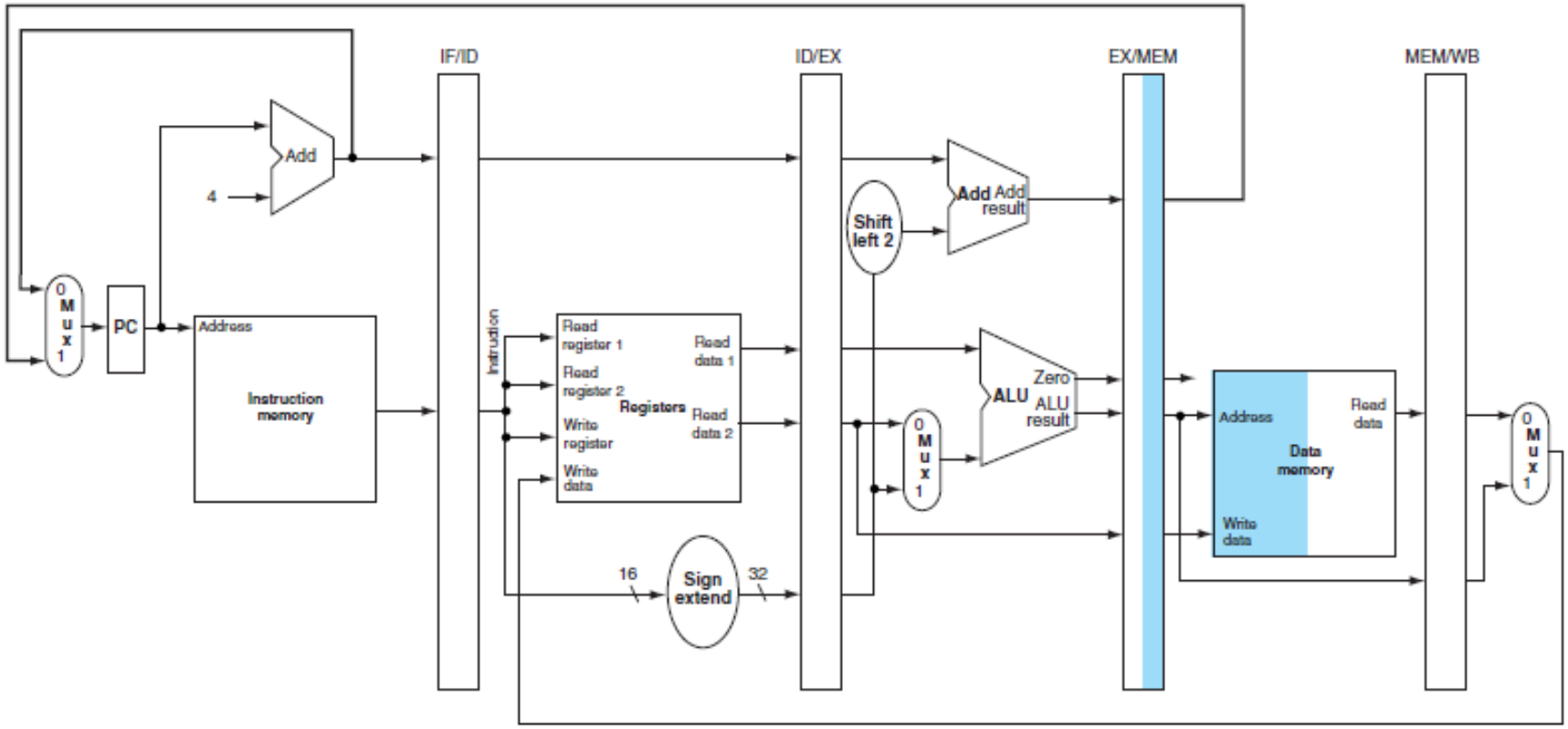
Caminho de dados usando o pipeline

- 3º estágio (EX) para o **store**:



Caminho de dados usando o pipeline

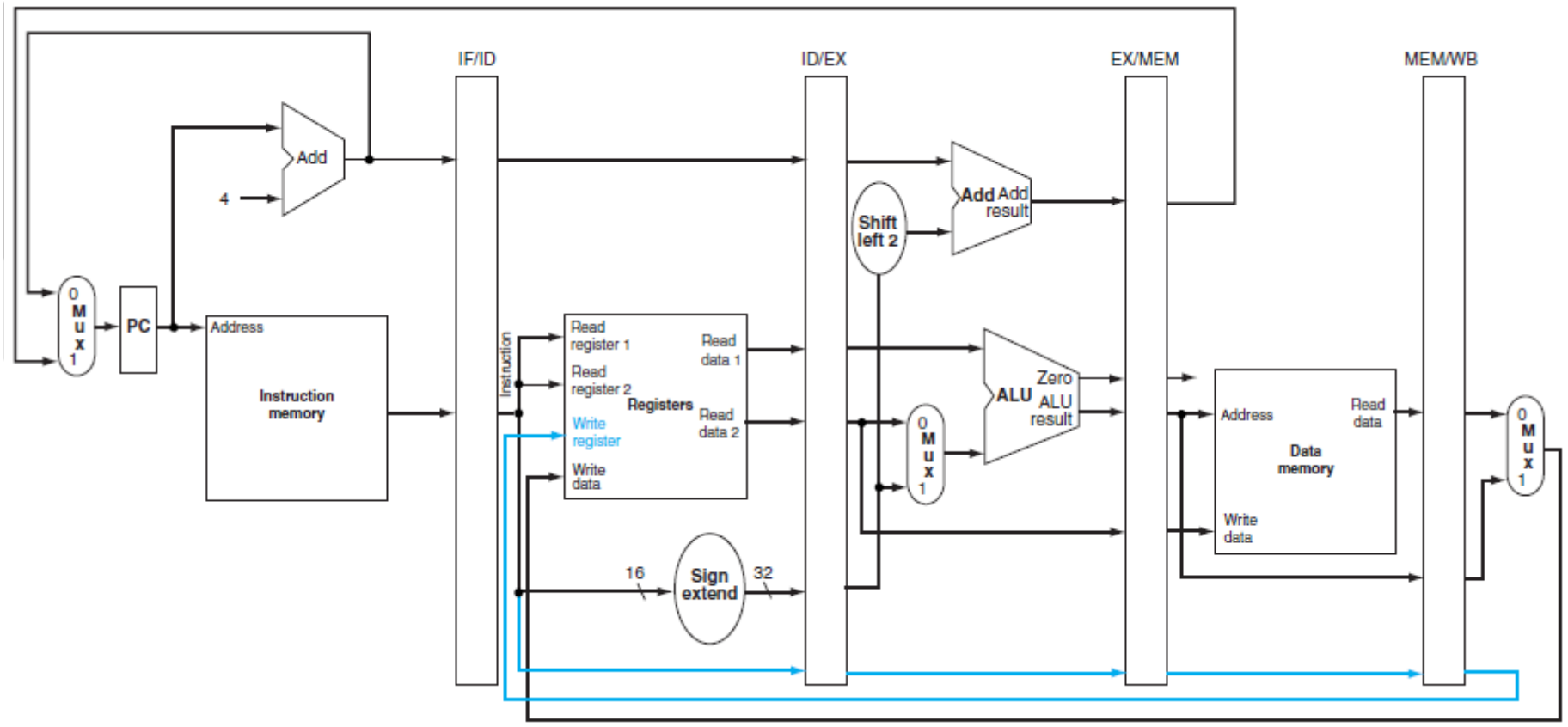
- 4º estágio (MEM) para o **store**:



- Nada acontece no 5º estágio para o store

Caminho de dados usando o pipeline

- Algo está faltando no caminho de dados da instrução load apresentado. O que está errado?

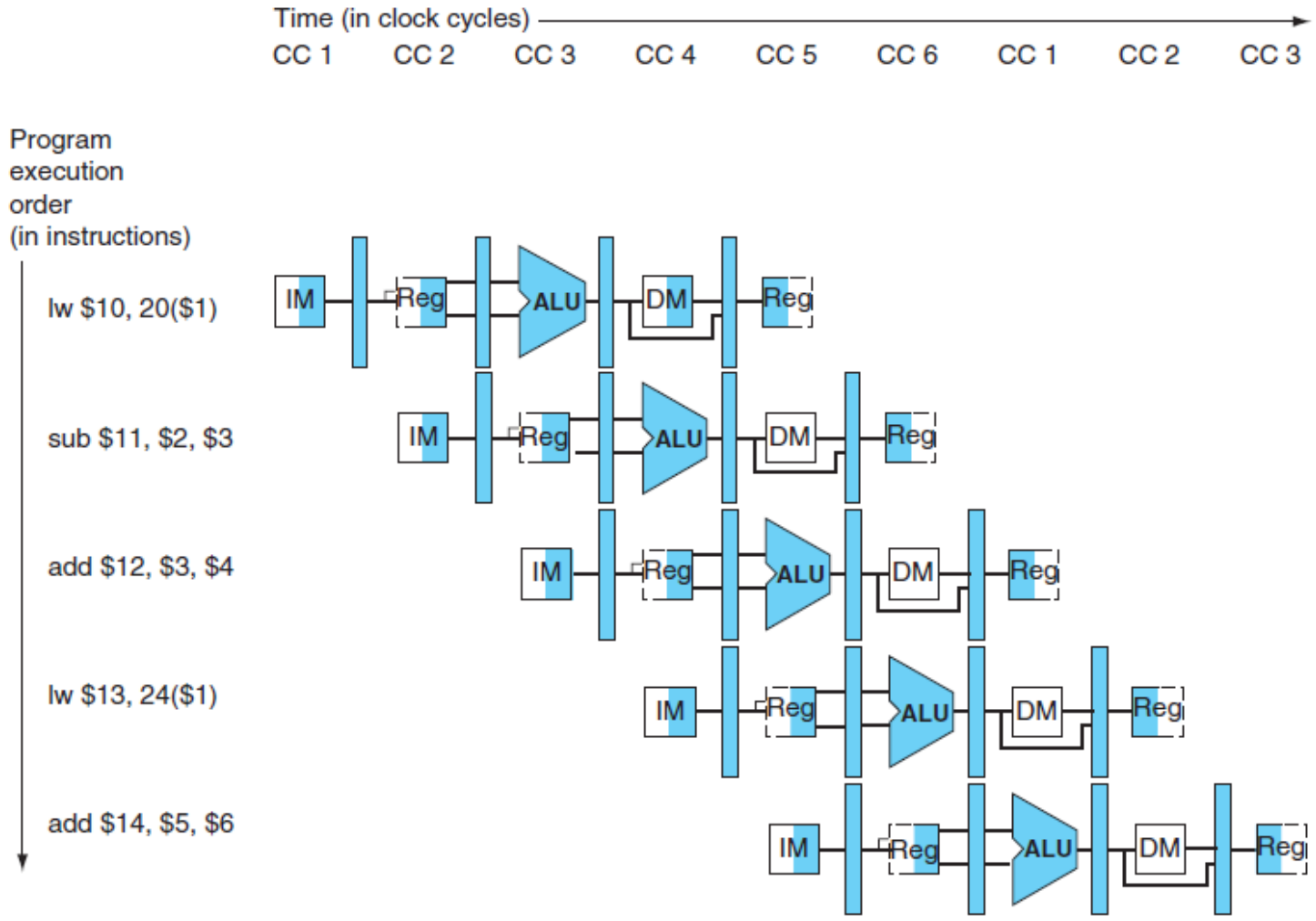


Diagramas de Pipeline

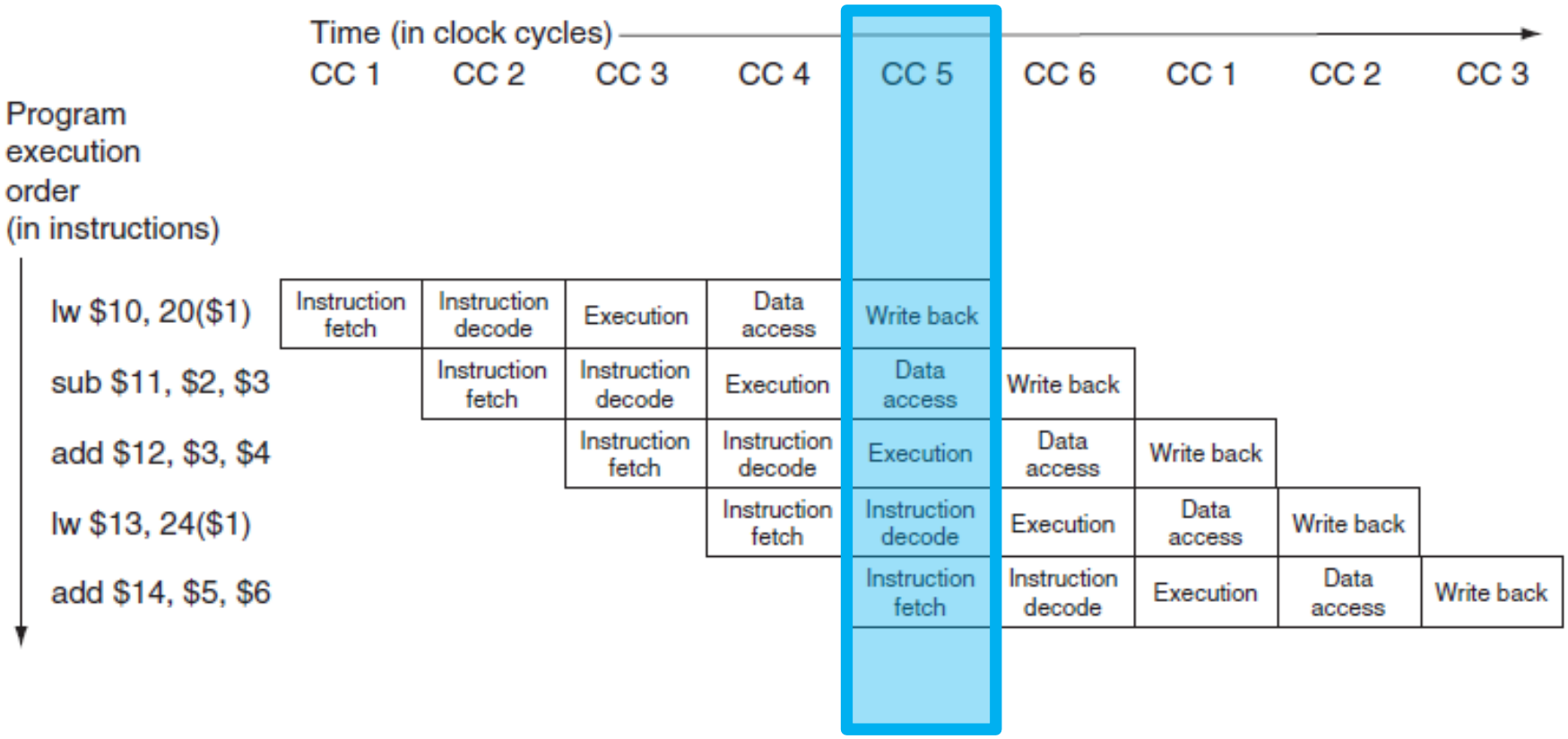
- São diagramas para representar o que está no pipeline em um dado momento
  - Diagramas com vários ciclos
  - Diagramas de um único ciclo
- Exemplo:
- Considere o código:

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

Diagramas de Pipeline



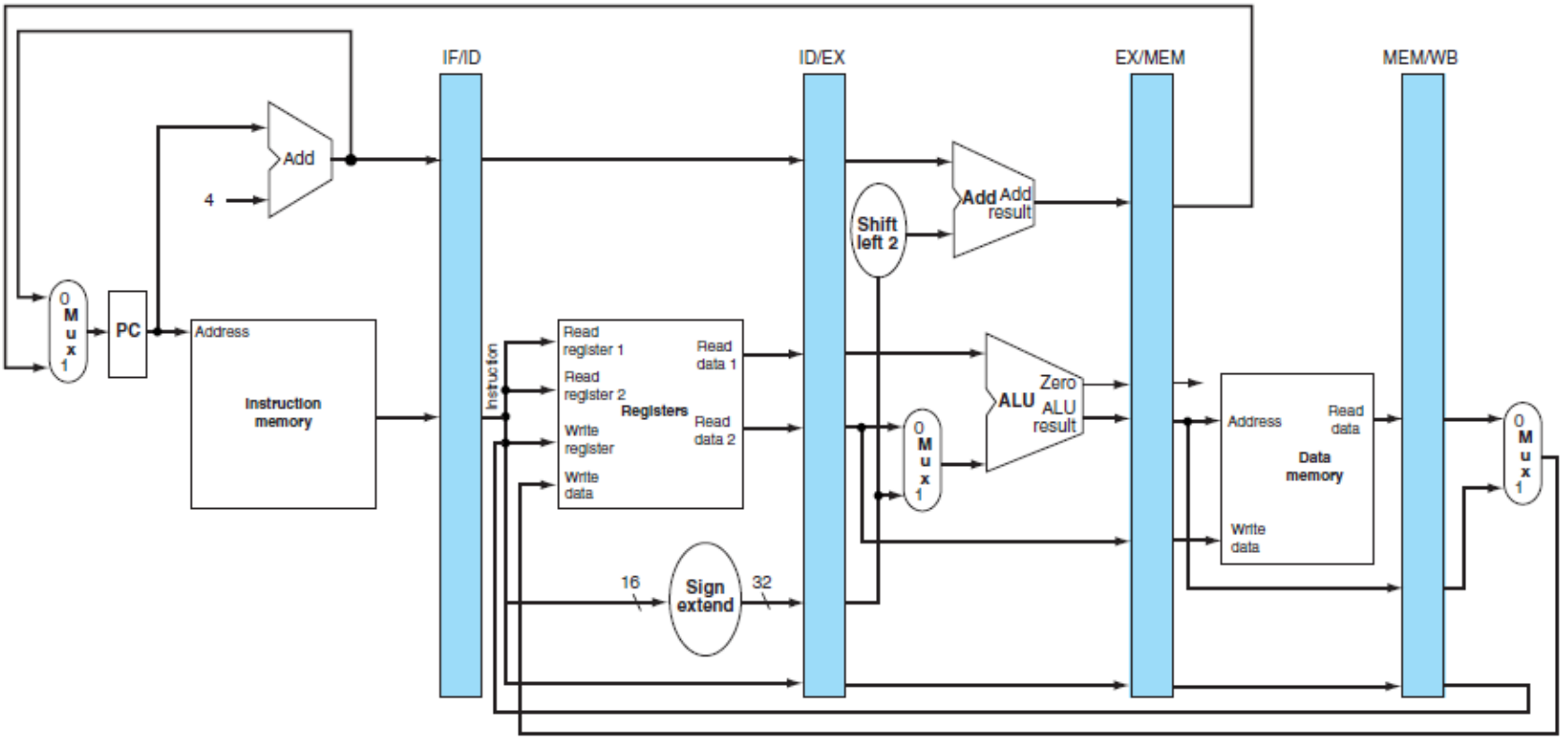
Diagramas de Pipeline



Diagramas de Pipeline

- Ciclo de clock 5 (CC 5):

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4, $11	sub $11, $2, $3	lw$10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write back



Controle do Pipeline

- A unidade de controle para o processador com pipeline muda o seu funcionamento, mas os sinais tem o mesmo significado
- A diferença é que precisamos identificar os sinais referentes a cada estágio e gerá-los no momento certo

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

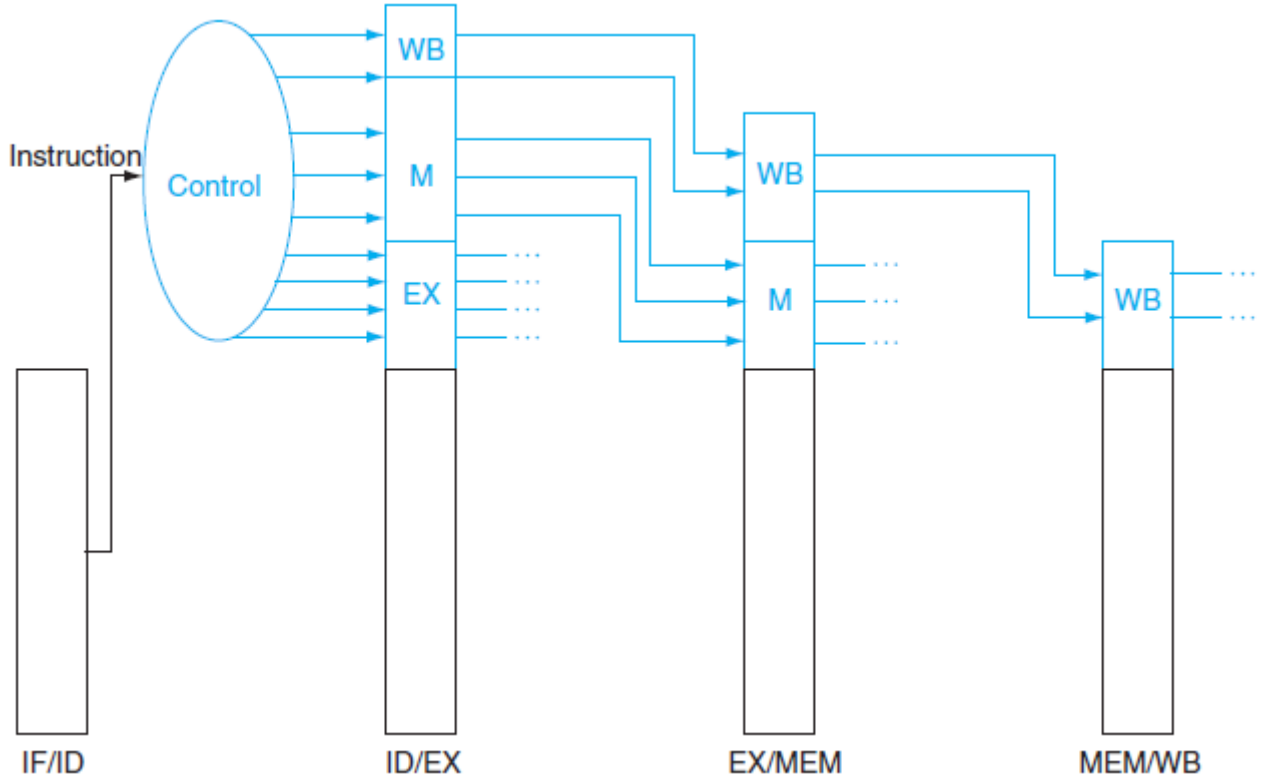
Controle do Pipeline

- Esta tabela também é igual à anterior.

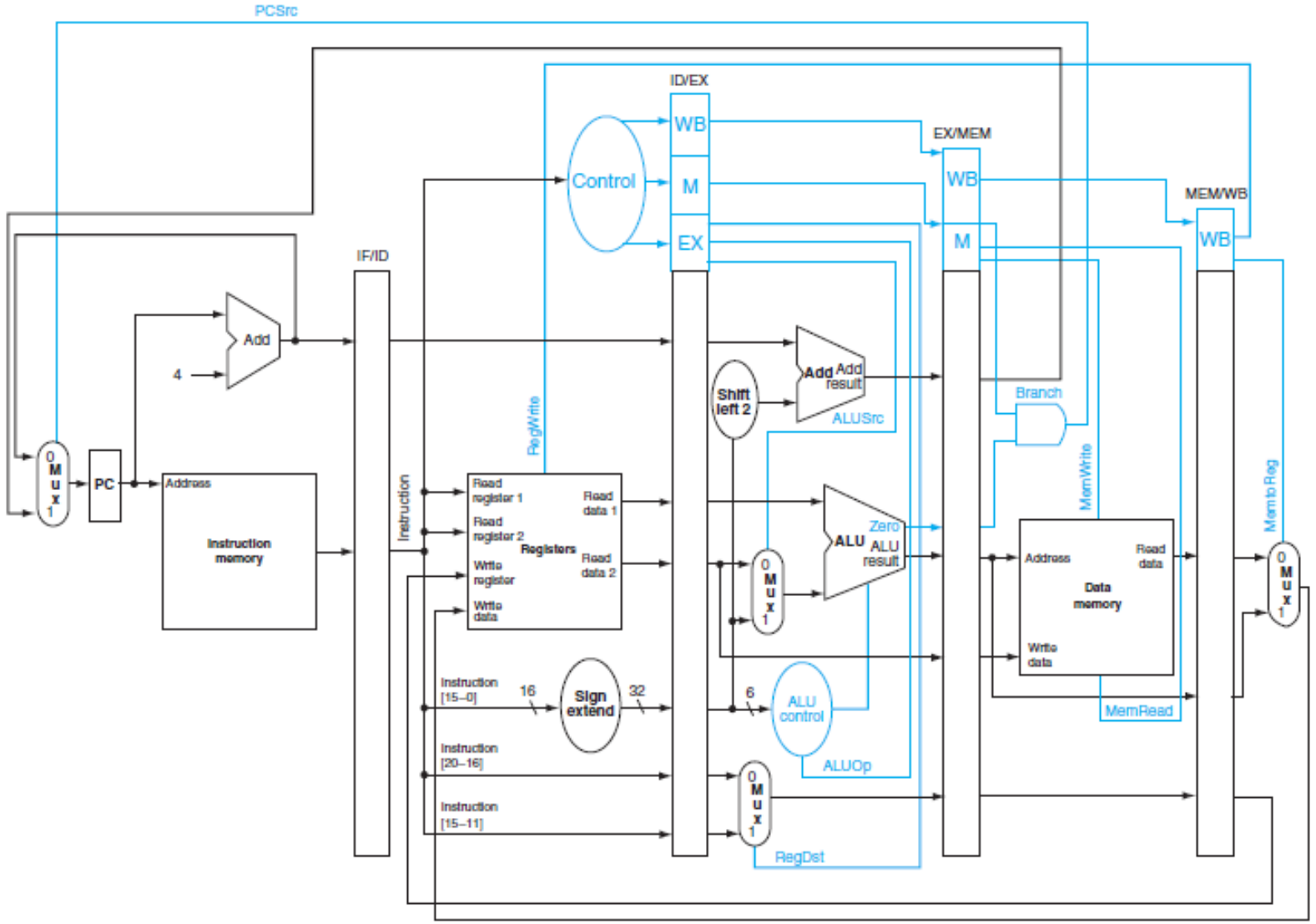
Sinal	Efeito quando =0	Efeito quando =1
RegDst	O número do reg. destino vem do campo 20:16	O número do reg. destino vem do campo 15:11
RegWrite	nenhum	O registrador na entrada Write register é escrito com o valor da entrada Write data
ALUSrc	O segundo operando da ALU vem da segunda saída da caixa de registradores (Read data 2)	O segundo operando da ALU vem da parte imediata da instrução estendida para 32 bits
PCSrc	PC recebe PC+4	PC recebe PC mais deslocamento da instrução de desvio (beq)
MemRead	Nenhum	O dado da memória no endereço Address é colocado na saída Read data
MemWrite	Nenhum	O dado na entrada Write data é escrito na memória no endereço Address
MemtoReg	Um resultado da ALU é enviado à caixa de registradores para ser gravado em um registrador	Um dado da memória é enviado à caixa de registradores para ser gravado em um registrador

Controle do Pipeline

- Os estágios de busca de instruções e decodificação são iguais para todas as instruções, portanto o controle fica como antes, a cada ciclo de clock.
- Nos demais, salvamos sinais de controle nos registradores de pipeline



Controle do Pipeline



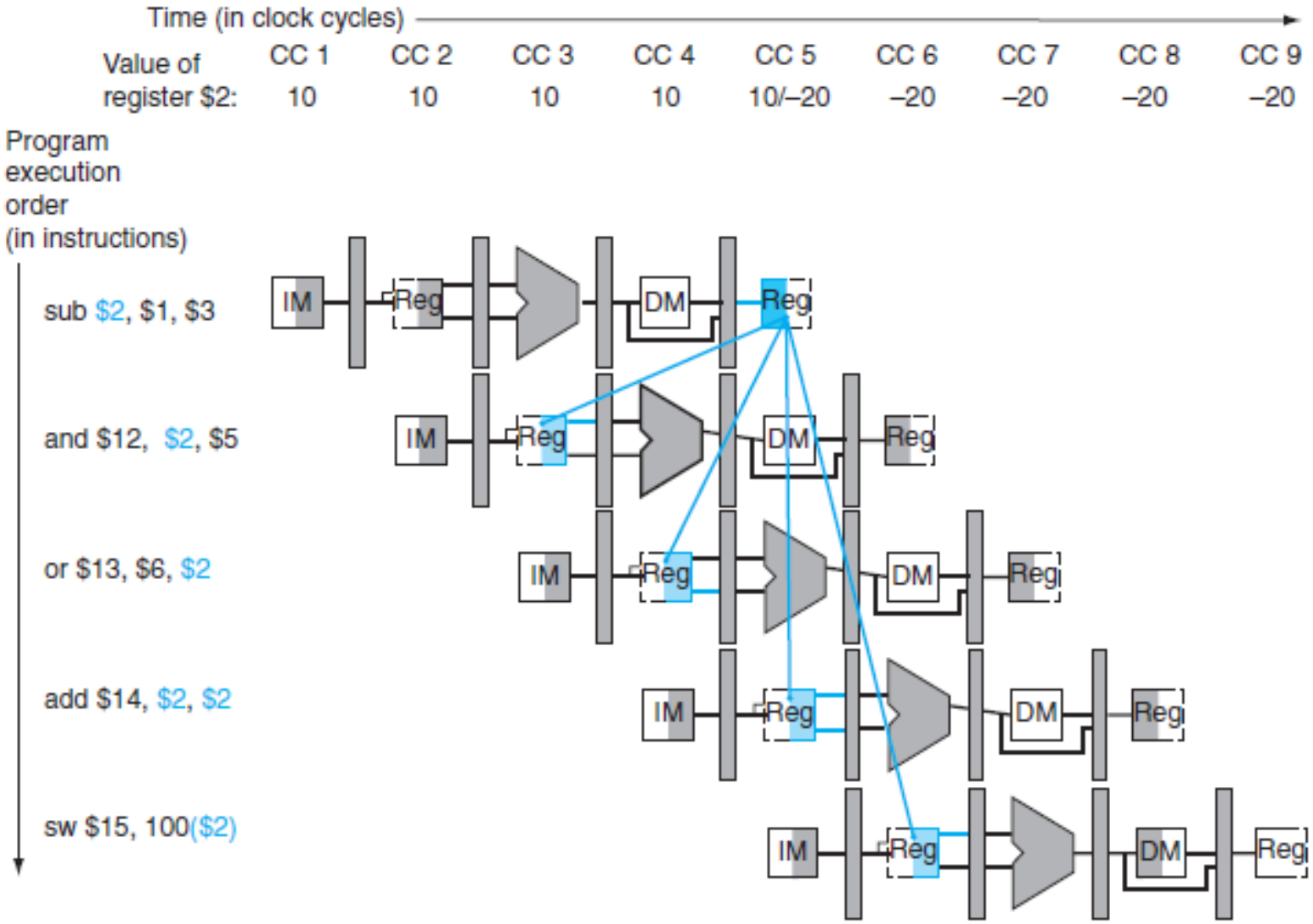
Hazard de dados e forwarding

- Vejamos agora maiores detalhes do funcionamento do pipeline
- Considere o código:

```
sub    $2, $1,$3      # Register $2 written by sub
and    $12,$2,$5      # 1st operand($2) depends on sub
or     $13,$6,$2      # 2nd operand($2) depends on sub
add    $14,$2,$2      # 1st($2) & 2nd($2) depend on sub
sw     $15,100($2)    # Base ($2) depends on sub
```

- As quatro últimas operações são todas dependentes do resultado da primeira instrução, que escreve no registrador $2

Hazard de dados e forwarding



Hazard de dados e forwarding

- Para o hardware identificar o hazard e o tipo, é preciso uma notação.
- Como foi visto, cada registrador de pipeline armazena informações de controle e dados passados da etapa anterior
- Algumas dessas informações são os números dos registradores envolvidos na instrução rs, rt e rd
- Vamos usar a seguinte notação, com o nome do registrador de pipeline e o'nome do registrador da instrução. Por exemplo:

  ID/EX.RegisterRd

Hazard de dados e forwarding

- Existem dois pares de condições de hazard baseado nessa notação:

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

- No código anterior, o hazard **sub-and** é do tipo 1a.

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

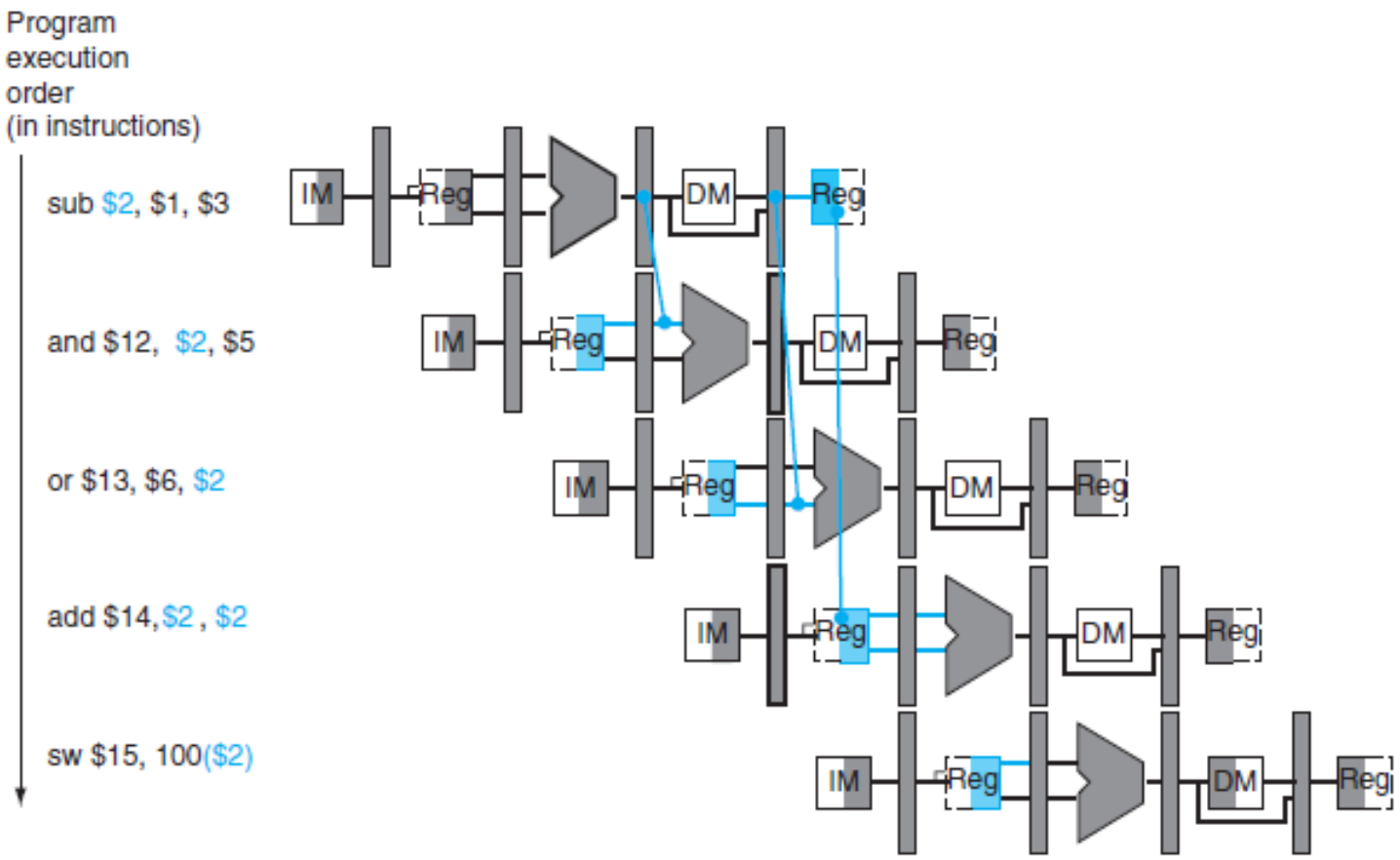
- Qual é o tipo do hazard **sub-or**?

Resposta: 2b

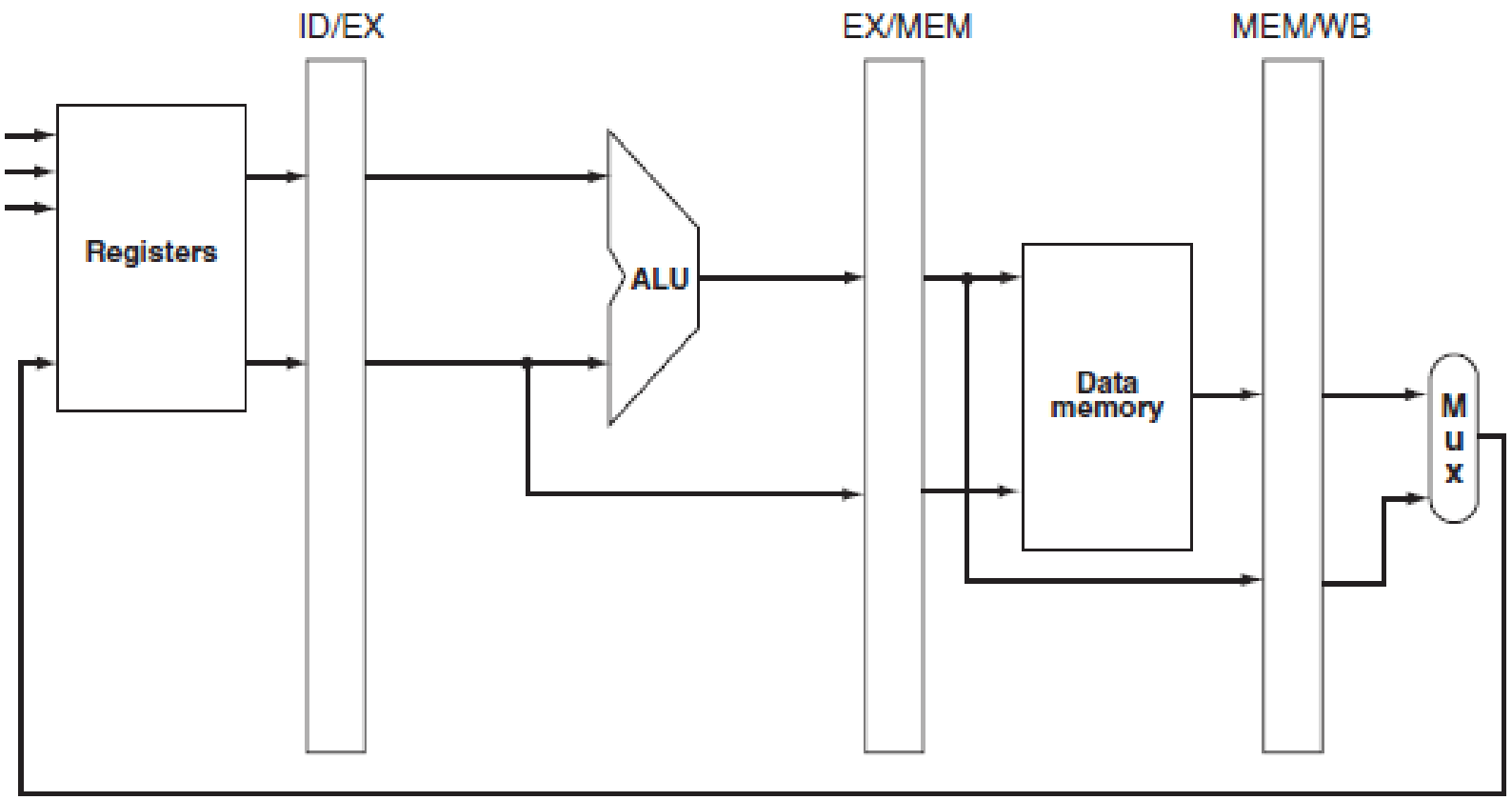
Hazard de dados e forwarding

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

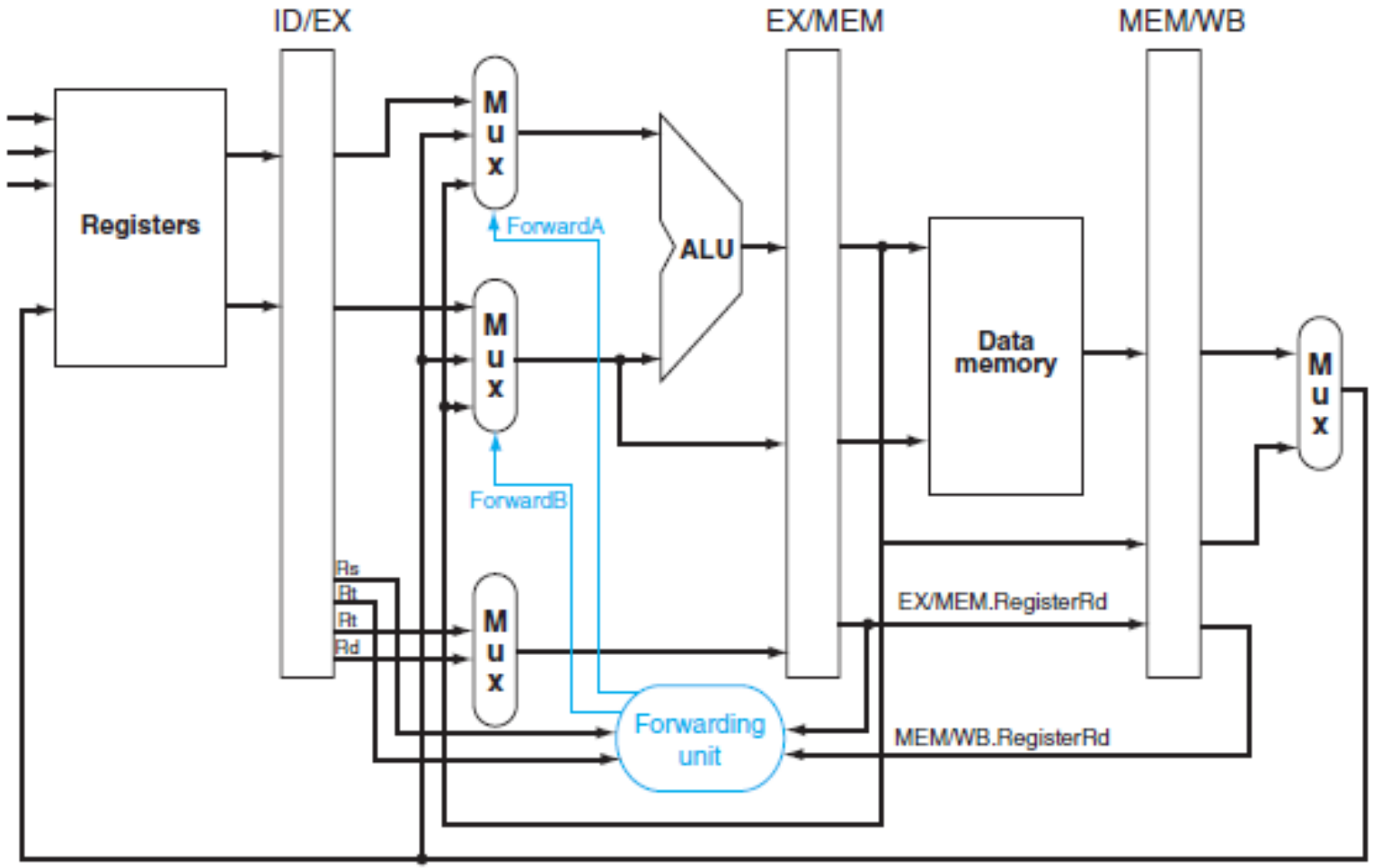


Hazard de dados e forwarding



a. No forwarding

Detecção do Hazard



b. With forwarding

Hazard de dados e forwarding

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

- Registrador $0
  - Pode surgir uma instrução com o $0 como destino
  - Neste caso, evitar o forwarding
  - Ex.: `sll $0,$1,$2`
  - Um valor errado pode ser passado adiante

Hazard de dados e forwarding

Condições para detecção:

1. Hazard EX:

```
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
```

- Testa-se o RegWrite para evitar forwarding desnecessário caso a instrução não escreva em registrador.

Hazard de dados e forwarding

Condições para detecção:

2. Hazard MEM:

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

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

Hazard de dados e forwarding

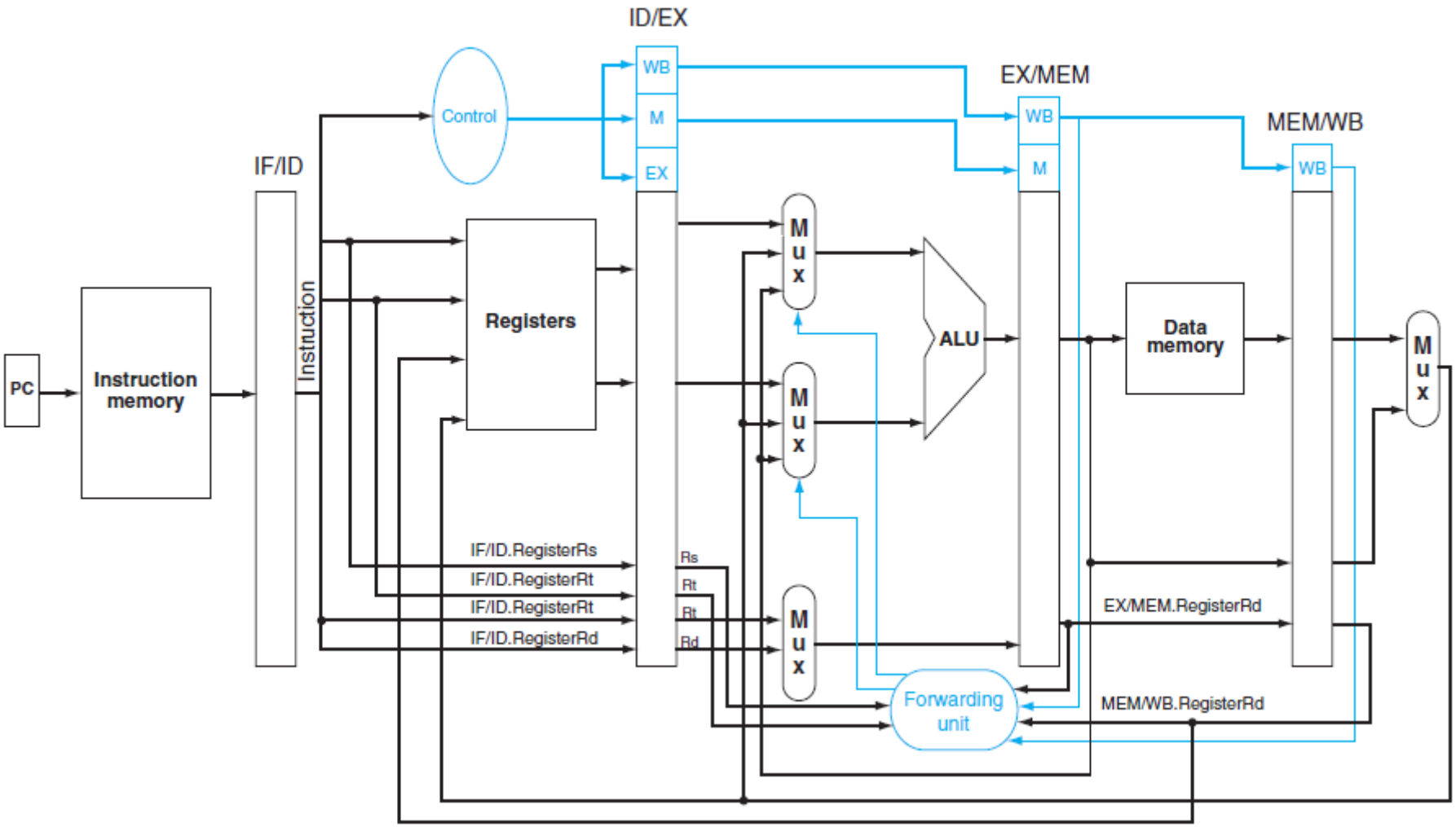
- Uma complicação ocorre neste exemplo:

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

- Neste caso ambas as condições ocorrem. O dado é selecionado do estágio MEM que é o mais recente:

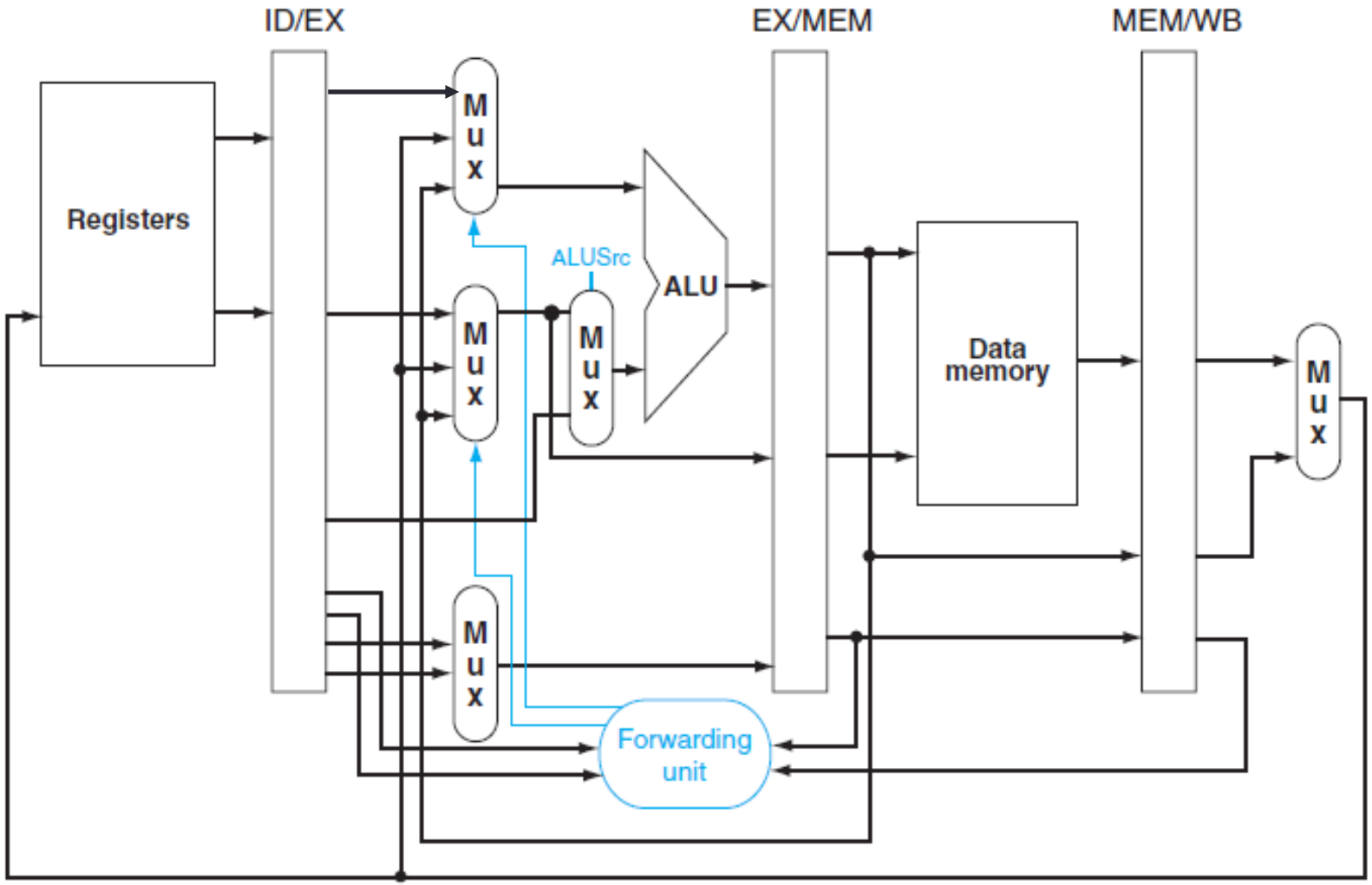
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

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
```

Hazard de dados e forwarding



Caminho de dados modificado para resolver hazards via forwarding

Hazard de dados e forwarding



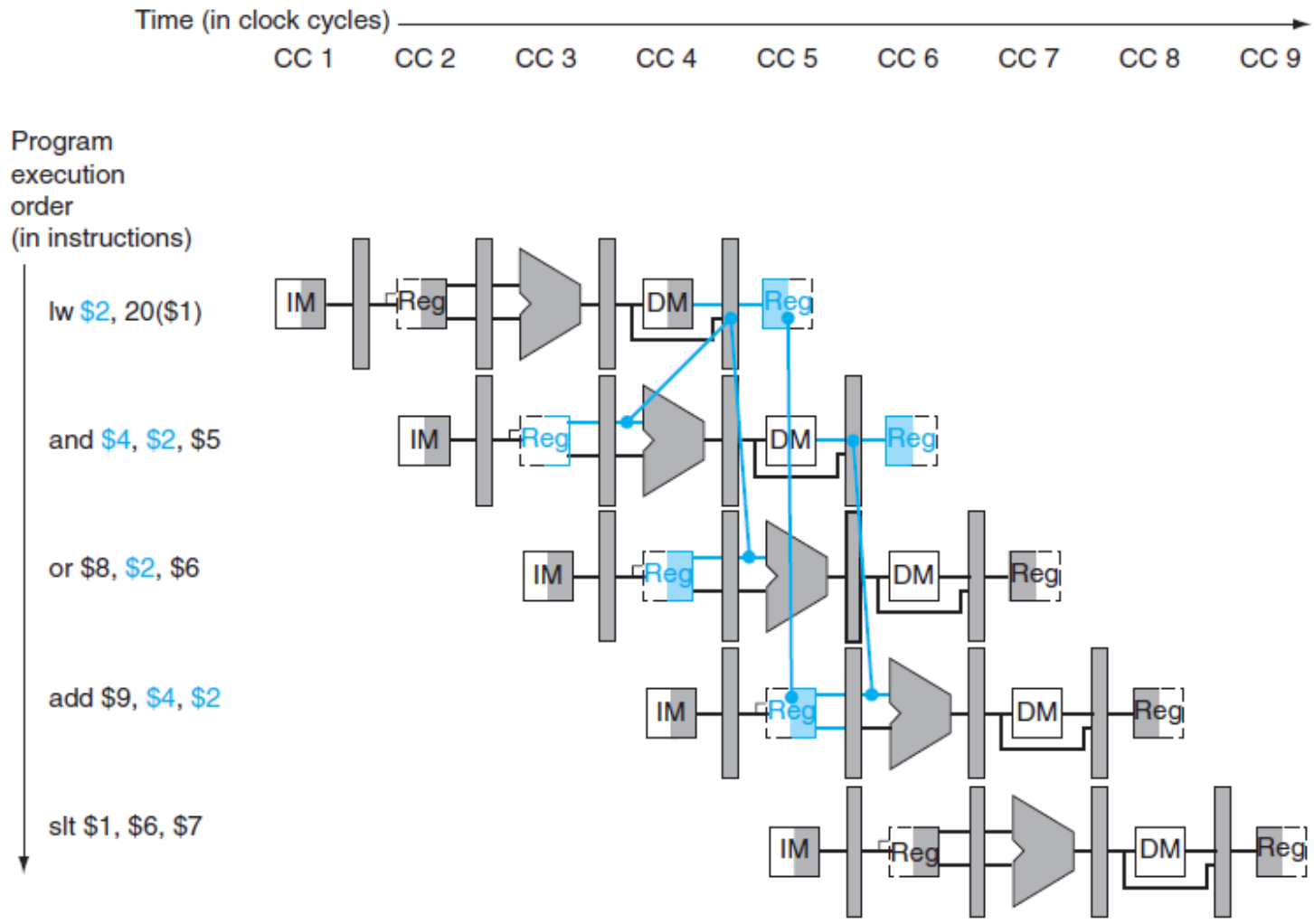
Mux para permitir a seleção do operando imediato

Hazard de dados e stalls

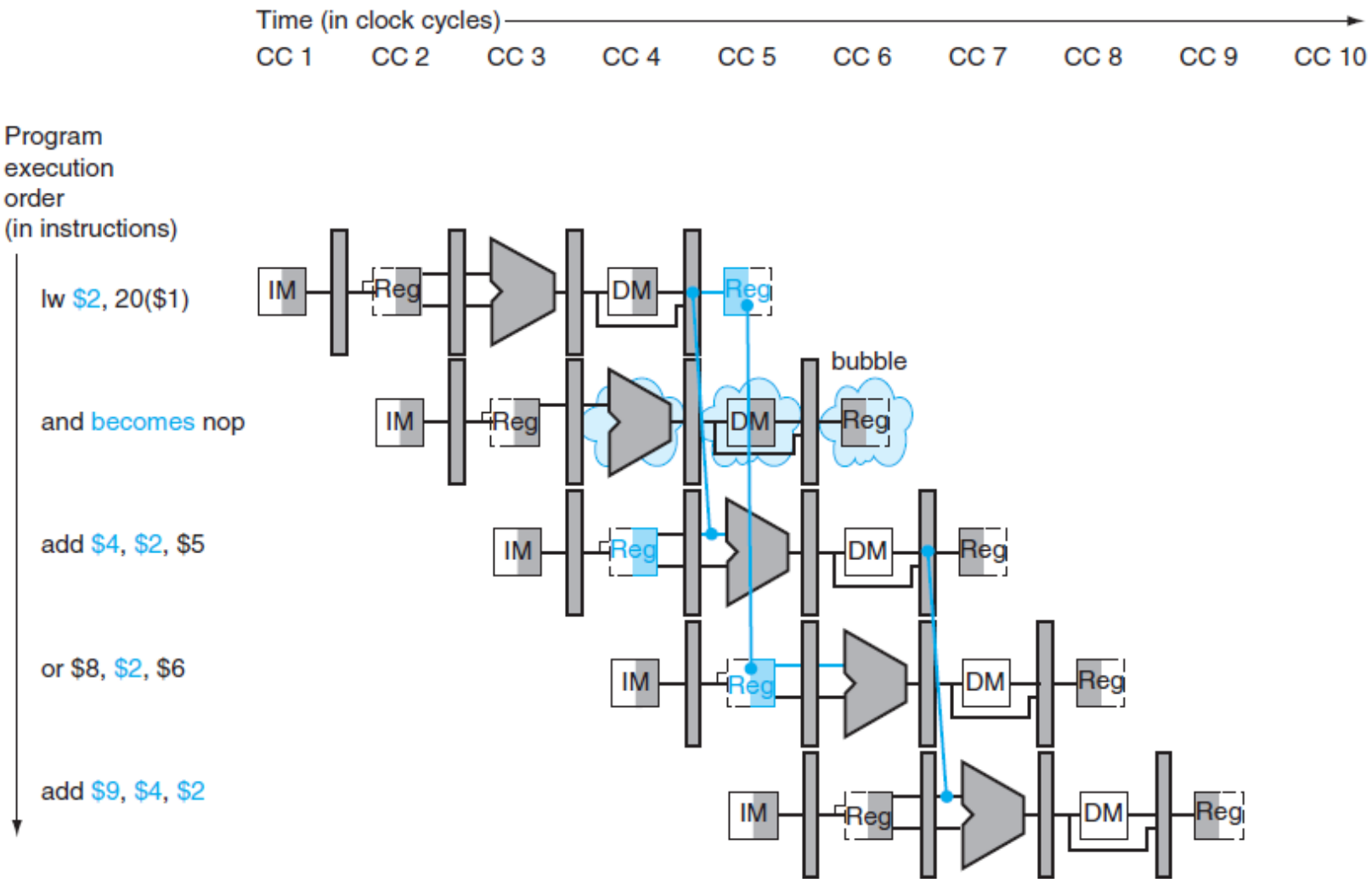
- Caso de stall inevitável por forwarding: após uma instrução load e acesso ao mesmo registrador.
- Necessário um detector desse tipo de hazard no estágio ID

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      stall the pipeline
```

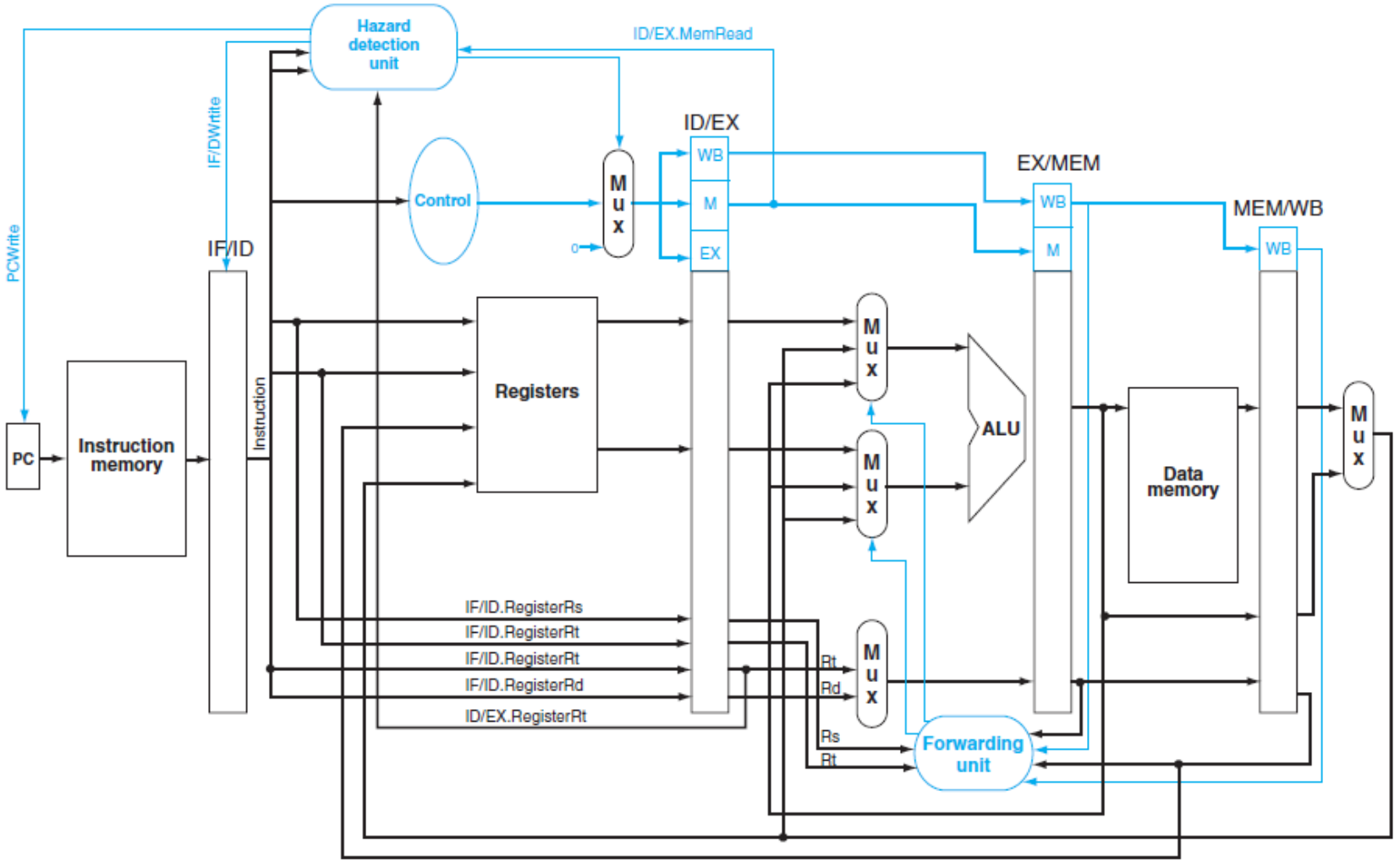
Hazard de dados e stalls



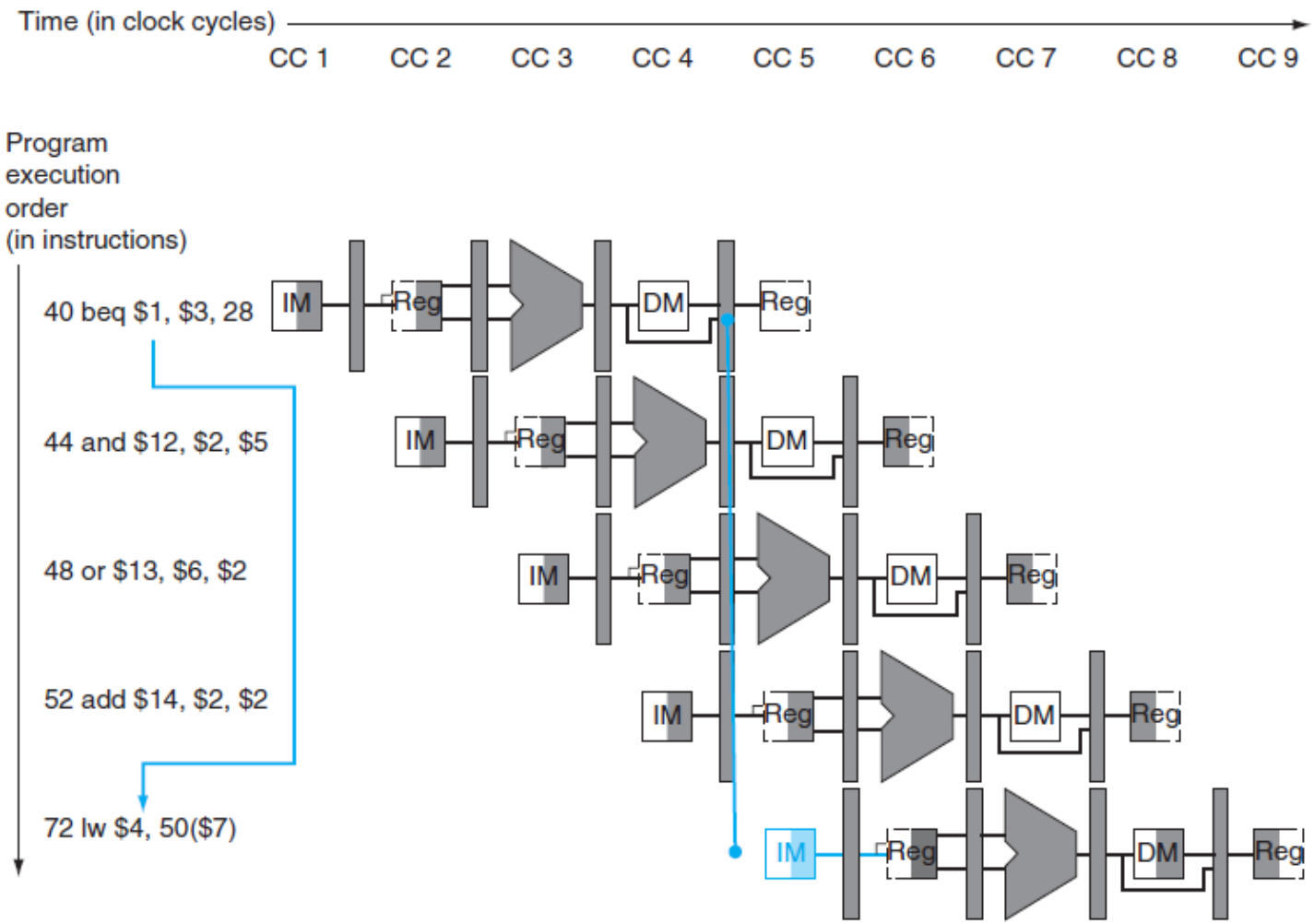
Hazard de dados e stalls



Hazard de dados e stalls



Hazard de desvios



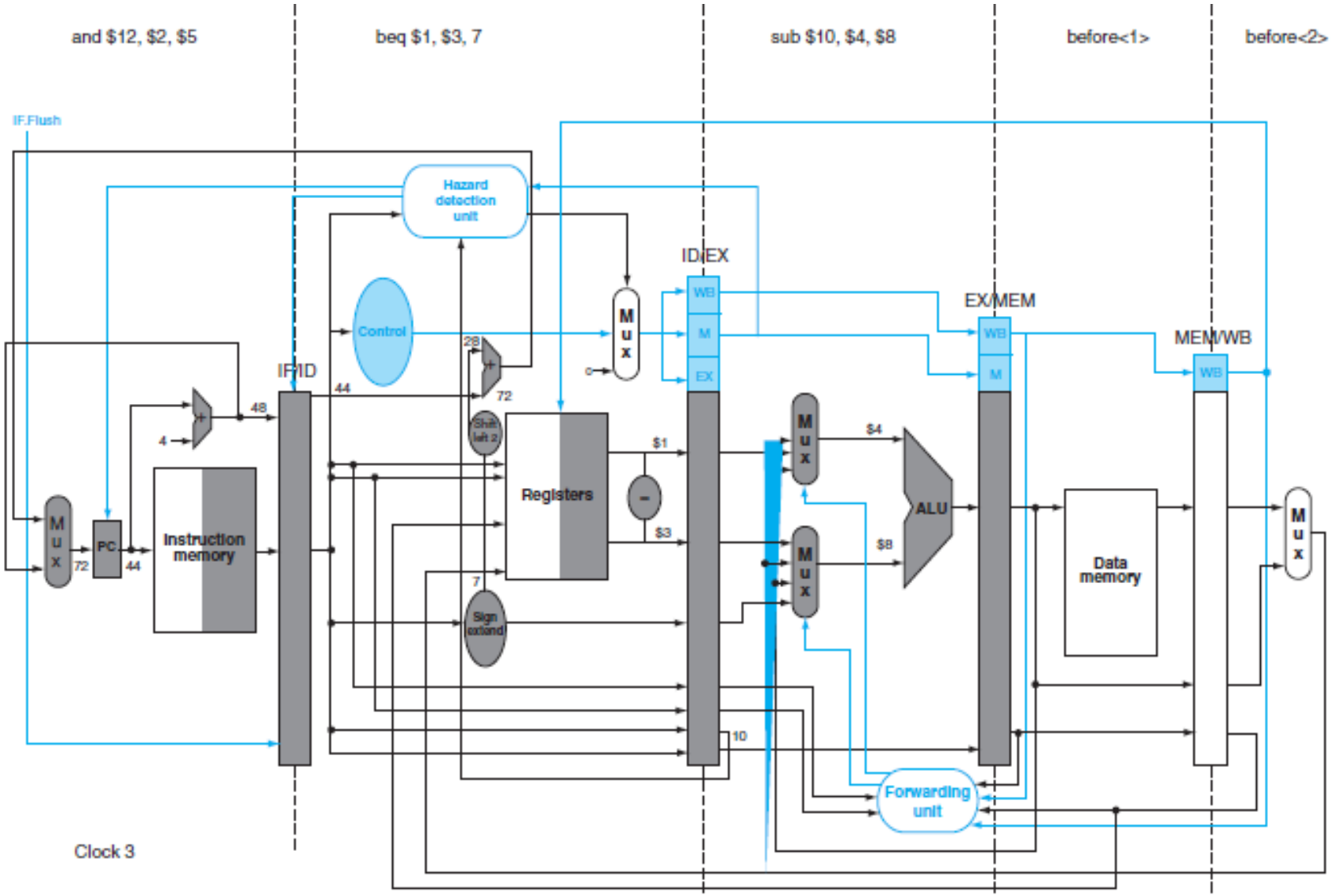
Hazard de desvios

- Exemplo: Mostre o que acontece quando o desvio é tomado na seguinte sequência de instruções.

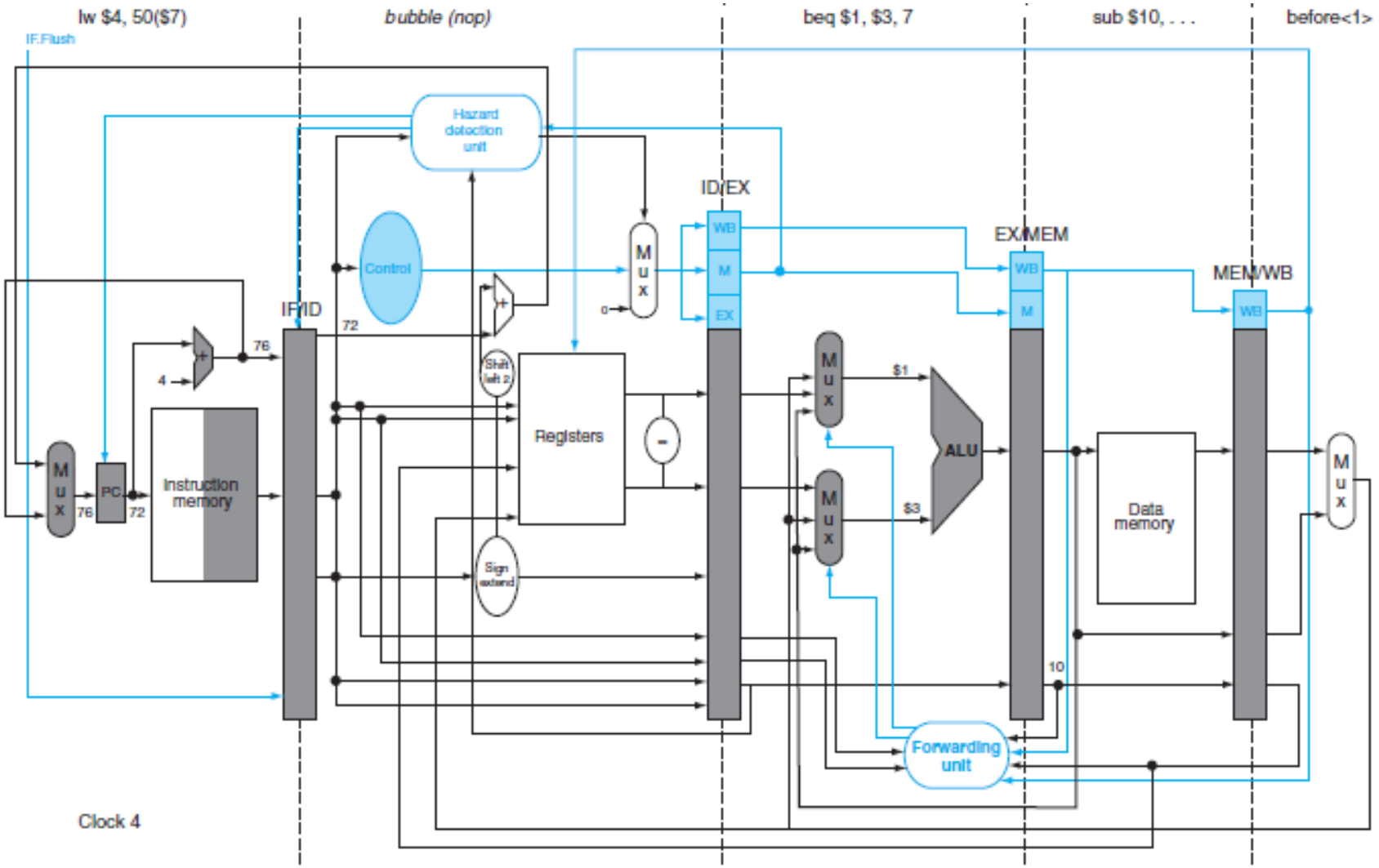
```
36 sub $10, $4, $8
40 beq  $1, $3, 7 # PC-relative branch to 40 + 4
+ 7 * 4 = 72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw  $4,  50($7)
```

- O pipeline está otimizado para desvios não tomados

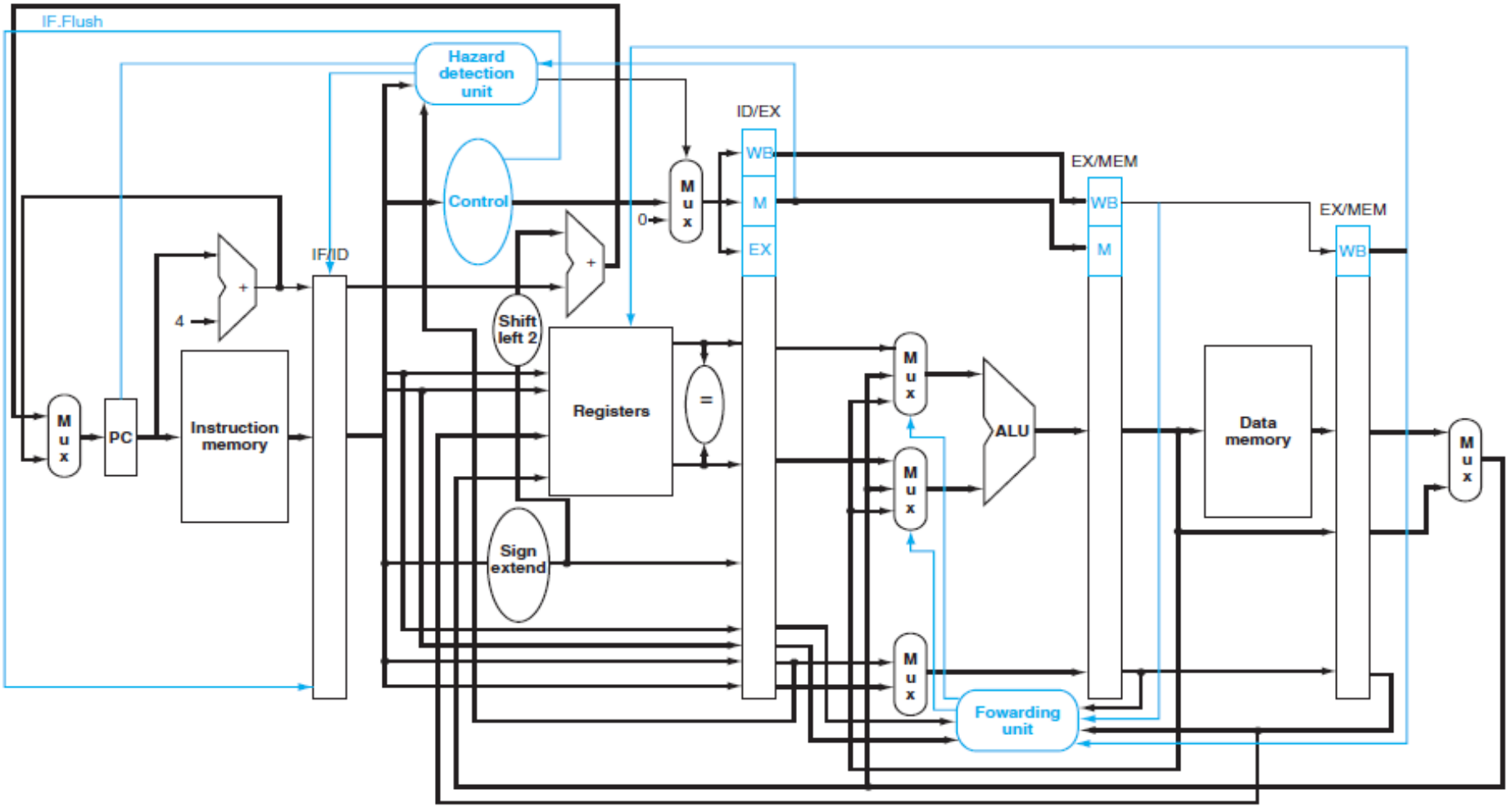
Hazard de desvios



Hazard de desvios



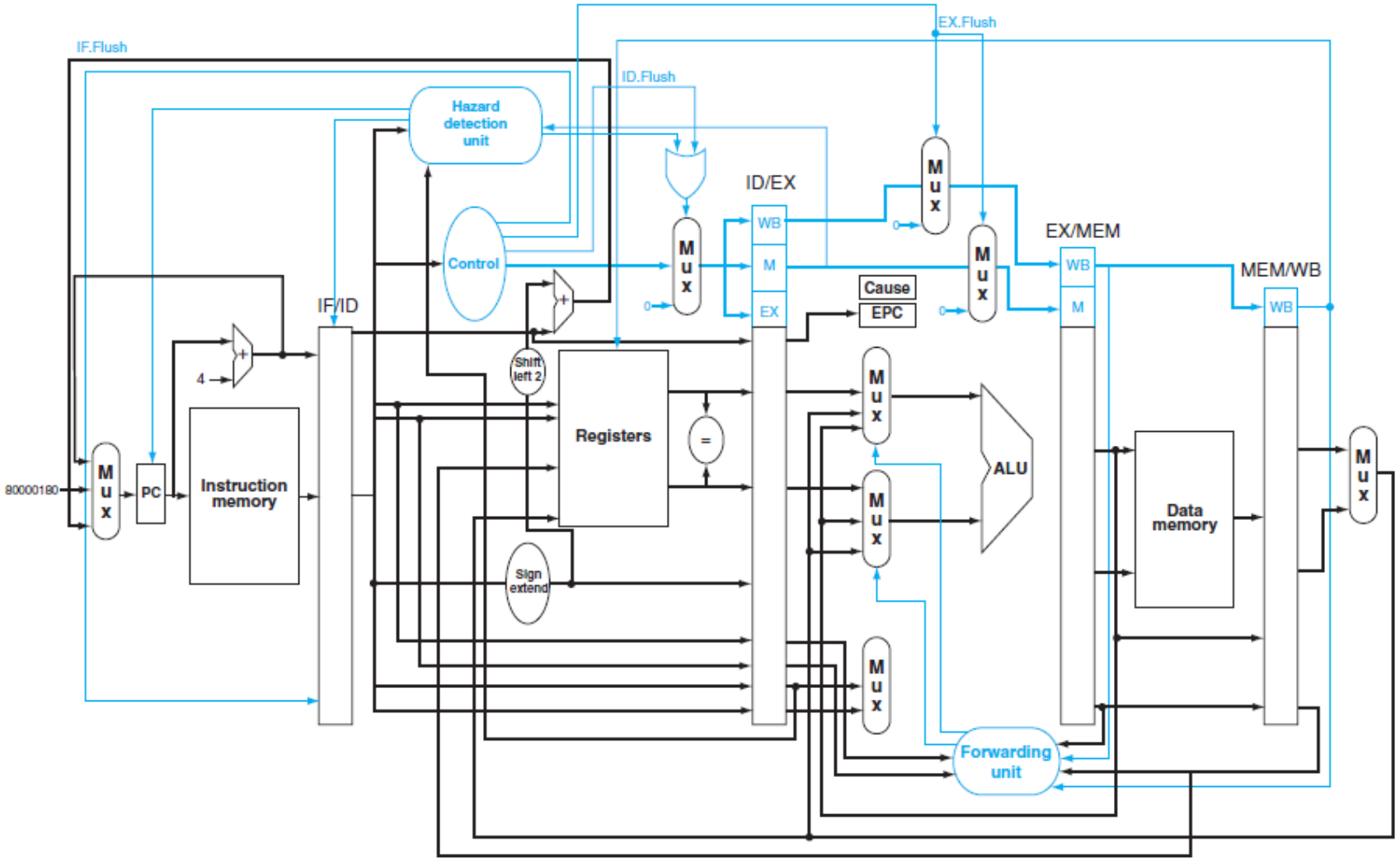
Caminho de dados completo



Exceções

- Exceções são semelhantes a chamadas de procedimento
- O controle deve ser desviado para uma rotina de tratamento
- Por exemplo, se uma instrução add $1,$2,$1 produzir overflow aritmético
- Após a instrução, o desvio será tomado e um flush no pipeline será causado nos estágios ID e EX (ao contrário dos branches que causam o flush no estágio IF)
- Sinais ID.Flush e EX.Flush
- Causa a desativação das linhas de controle
- Reexecução da instrução
- Salvamento do PC no EPC (Exception Program Counter)

Exceções


IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Cause
EPC
IF/ID
Shift left 2
Registers
=
Sign extend
Instruction memory
PC
80000180
4
ALU
Data memory
Forwarding unit
Mux

Exercícios

1. Na tabela abaixo, se o tempo para a operação da ALU puder ser reduzido em 25%, responda:

a) Isso afetará o ganho de velocidade obtido pela técnica de pipelining? Nesse caso, em quanto? Caso contrário, por quê?

b) E se a operação da ALU exigir a mais 25% do tempo?

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `and`, `or`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Exercícios

2. Mostre os caminhos de forwarding necessários para executar as quatro instruções a seguir:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

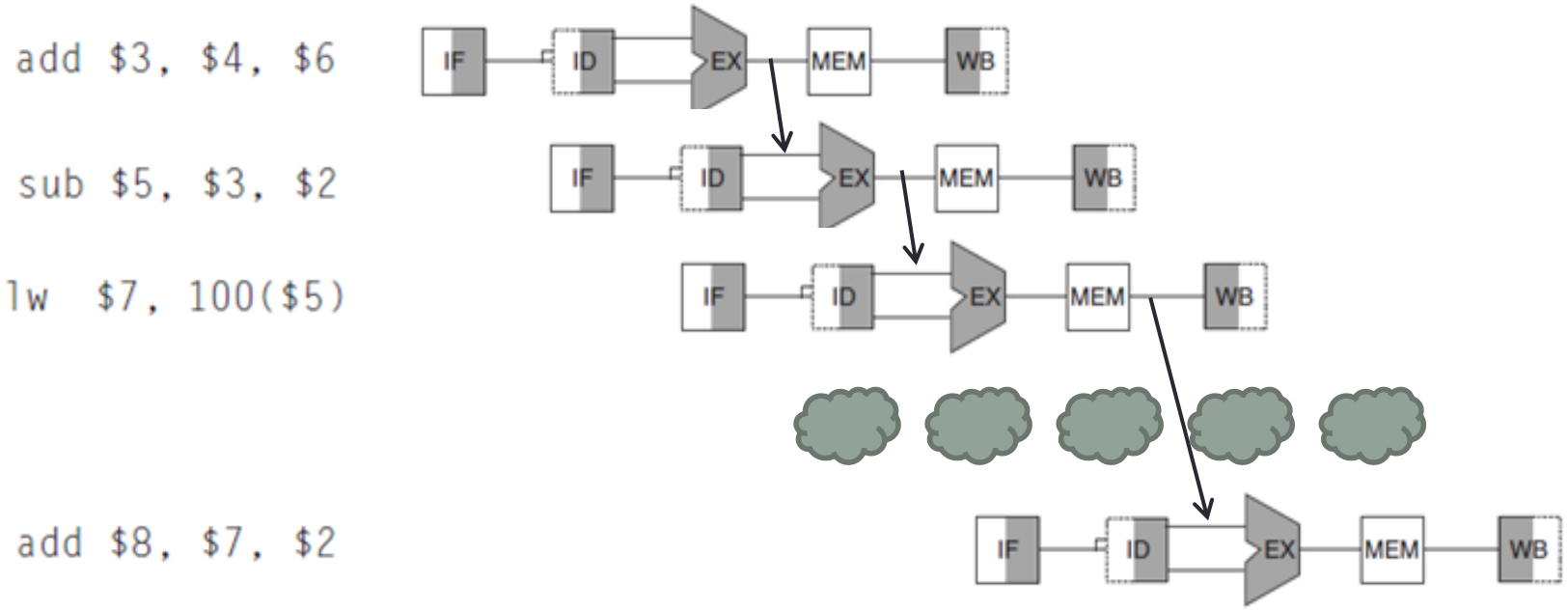
3. Encontre as dependências de dados no código abaixo. Quais são resolvidas por forwarding? Quais geram stall?

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

4. Quantos ciclos de clock totais serão gastos nos códigos das questões 2 e 3?

Solução Ex.2

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```



Exercícios

- 4.5 Mostre os caminhos de forwarding necessários para executar as instruções a seguir:

```
add $3,$4,$6
add $4,$5,$6
add $2,$3,$4
add $1,$2,$2
add $1,$2,$2
lw $2,100($1)
add $1,$2,$2
```

Exercícios

5. Considere a execução do seguinte código no MIPS com pipeline, forwarding e detecção de hazard:

```
lw   $4, 100($2)
sub  $6, $4, $3
add  $2, $3, $5
```

Quantos ciclos serão necessários para executar esse código? Desenhe um diagrama de vários ciclos mostrando as dependências. Redesenhe o diagrama mostrando a solução que será tomada. Como seria possível reduzir o tempo de execução desse trecho de código?

Exercícios

6. Considere a execução do código a seguir no processador MIPS com pipeline:

```
add    $2,  $3,  $1
sub    $4,  $3,  $5
add    $5,  $3,  $7
add    $7,  $6,  $1
add    $8,  $2,  $6
```

Ao final do 5º ciclo da execução, quais registradores estão sendo lidos e qual registrador será escrito?

Exercícios

7. Vimos um exemplo em que uma reordenação das instruções tornou o código mais eficiente porque evitou stalls. Reescreva o código a seguir causando o efeito contrário, isto é, minimizando o desempenho e fazendo-o utilizar mais ciclos de clock, mas sem que o resultado da execução seja modificado.

```
lw    $2, 100($6)
lw    $3, 200($7)
add   $4, $2, $3
add   $6, $3, $5
sub   $8, $4, $6
lw    $7, 300($8)
beq   $7, $8, Loop
```