

EA960

Organização do Processador – Projeto de *Pipeline*



Faculdade de Engenharia Elétrica e de Computação (FEEC)
Universidade Estadual de Campinas (UNICAMP)

Prof. Levy Boccato

Introdução

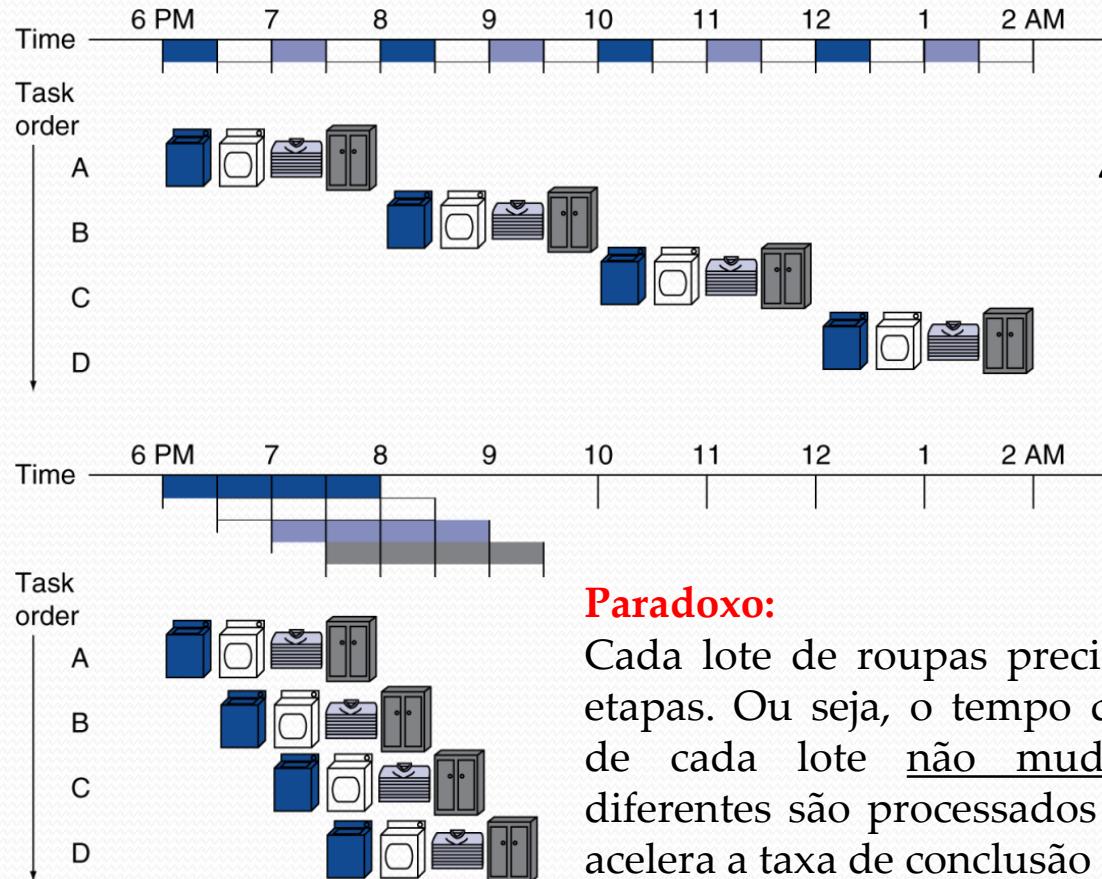
- A primeira estratégia de construção do processador baseado na arquitetura MIPS optou por fixar o número de ciclos de relógio consumidos por cada instrução em 1 (uniciclo).
- A mudança para uma abordagem multiciclo traz alguns impactos:
 - O número de ciclos por instrução (CPI, *clock cycles per instruction*) passa a ser variável.
 - A duração de um ciclo de relógio pode ser reduzida, pois precisa acomodar a etapa mais longa de uma instrução, em vez da instrução completa.

Introdução

- Outra opção de projeto do processador que busca acelerar a execução de um conjunto de instruções baseia-se na ideia de *pipelines*.
- *Pipeline*: estratégia de implementação do processador na qual múltiplas instruções são sobrepostas em execução.

Introdução

- Vamos começar com uma visão simplificada da ideia de *pipeline*.



4 etapas:

1. Lavar
2. Secar
3. Dobrar
4. Guardar

Paradoxo:

Cada lote de roupas precisa passar pelas 4 etapas. Ou seja, o tempo de processamento de cada lote não muda. Porém, lotes diferentes são processados em paralelo. Isto acelera a taxa de conclusão de lotes por hora.

Introdução

- A execução das instruções do MIPS classicamente envolve **cinco** passos:
 1. Busca de instrução (*Fetch*).
 2. Leitura dos registradores (operандos) enquanto ocorre a decodificação da instrução.
 - Isto é possível graças ao formato regular das instruções MIPS.
 3. Executa a operação ou calcula um endereço.
 4. Acessa um operando na memória de dados.
 5. Escreve o resultado em um registrador.
- Logo, a *pipeline* que vamos projetar e analisar neste tópico terá cinco estágios.

Introdução

- **Comparaçao preliminar:** uniciclo vs. *pipeline*
 - MIPS: 8 instruções – *lw*, *sw*, *add*, *sub*, *and*, *or*, *slt* e *beq*.
 - Tempo de operação:
 - 200ps para acesso à memória;
 - 200ps para operação da ALU;
 - 100ps para leitura ou escrita no arquivo de registradores.
 - Tempo de execução das instruções:

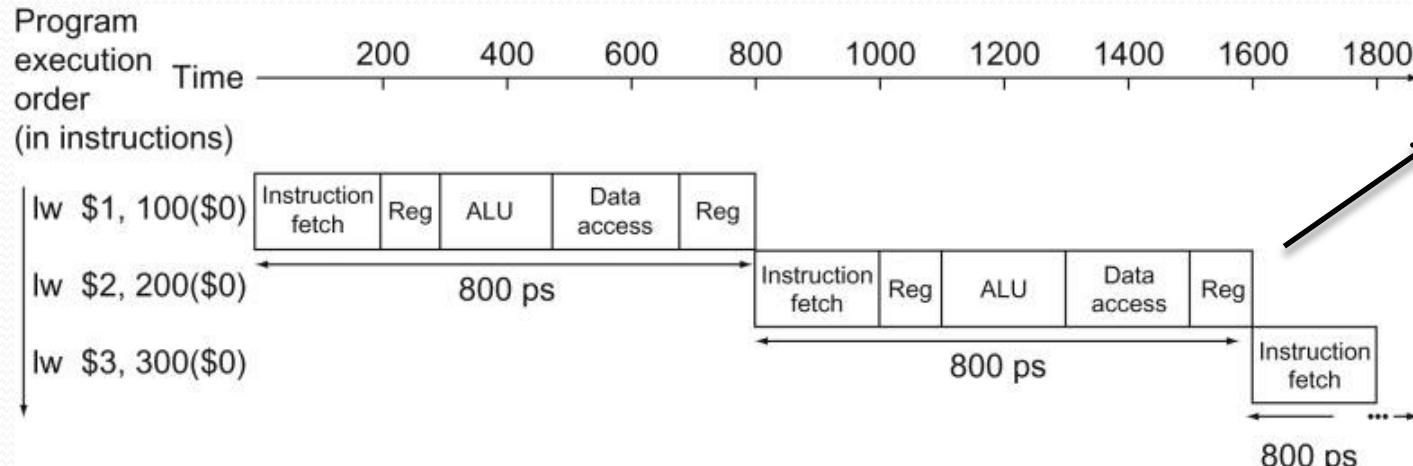
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (<i>lw</i>)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (<i>sw</i>)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (<i>add</i> , <i>sub</i> , AND, OR, <i>slt</i>)	200 ps	100 ps	200 ps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

Introdução

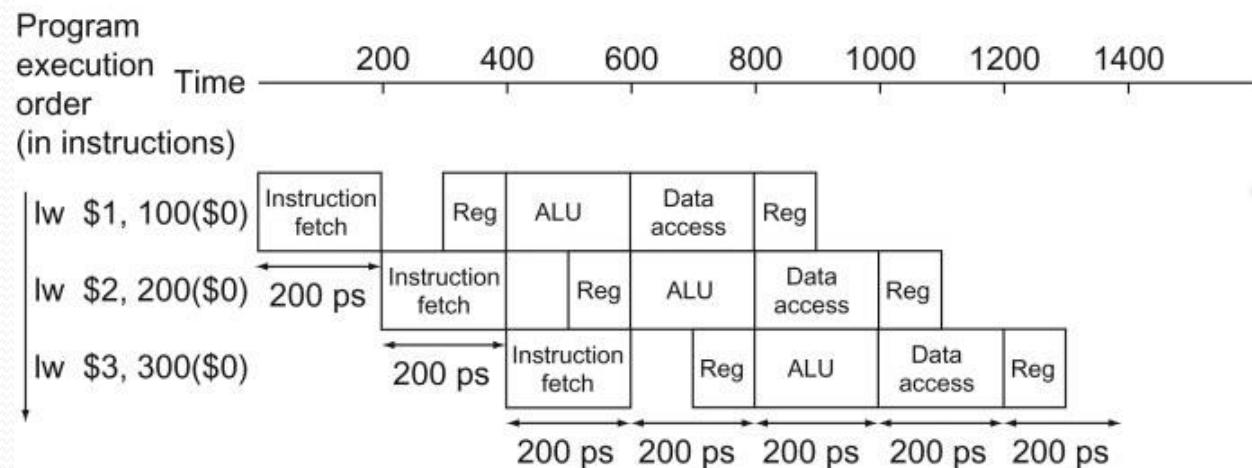
- **Comparaçao preliminar:** uniciclo vs. *pipeline*
 - **Uniciclo:** o mínimo ciclo de relógio será 800ps.
 - **Pipeline:** todos os estágios da *pipeline* consomem um ciclo de relógio.
 - Logo, o período do relógio deve ser longo o suficiente para acomodar a operação mais longa – 200ps.

Introdução

- Comparação preliminar: uniciclo vs. pipeline



O tempo entre a primeira instrução *load* e a quarta será de $3 \times 800 \text{ ps} = 2400 \text{ ps}$



Com *pipeline*, a quarta instrução começará a ser executada 600 ps após a primeira.

Introdução

- *Speed-up:*
 - Se os estágios da *pipeline* forem perfeitamente balanceados (i.e., consomem o mesmo tempo), então o tempo entre instruções em um processador com *pipeline*, assumindo condições ideais de operação, será:
$$T_{\text{pipeline}} = \frac{T_{\text{sem pipeline}}}{\text{Número de estágios da pipeline}}$$
 - Sob condições ideais e para um número suficientemente elevado de instruções, o *speed-up* é aproximadamente igual ao número de estágios da *pipeline*.
 - A expressão sugere que uma *pipeline* com 5 estágios deveria oferecer um aprimoramento por um fator de 5 em relação aos 800 ps da versão sem *pipeline*.
 - No entanto, os estágios podem não estar perfeitamente平衡ados. Ademais, a estratégia de *pipeline* introduz algum *overhead*.
 - Por isso, o ganho real observado é um pouco inferior ao máximo atingível.

Introdução

- No exemplo anterior, o tempo de execução das três instruções caiu de 2400ps (uniciclo) para 1400ps (*pipeline*).
- E se executássemos mais um milhão de instruções?

➤ *Pipeline:*

- Cada instrução acrescenta 200ps ao tempo total.
- Então, $1.000.000 \times 200\text{ps} + 1400\text{ps} = 200.001.400 \text{ ps}$.

Fator 4

➤ **Uniciclo:**

- Cada instrução acrescenta 800ps ao tempo total.
- $2400\text{ps} + 1.000.000 \times 800\text{ps} = 800.002.400 \text{ ps}$.

➤ **Ganho:** $800.002.400 / 200.001.400 \approx 4,0$

Introdução

- **Importante:**

- A organização de um processador em *pipeline* não altera o tempo de execução de uma única instrução, mas sim a taxa ou vazão de instruções (*throughput*) que o processador consegue atingir.
- *Throughput*: trata-se de uma métrica essencial, uma vez que os programas exigem, de uma maneira geral, a execução de milhões a bilhões de instruções.

Conjunto de Instruções e Pipeline

- O conjunto de instruções MIPS foi planejado visando uma implementação “eficiente” com *pipeline*.
 - **Todas as instruções possuem o mesmo tamanho (32 bits):**
 - Facilita a busca e a decodificação, que podem ser feitas em um ciclo cada (no 1º e 2º estágios da *pipeline*, respectivamente).
 - **Contra-exemplo:** arquitetura x86 – instruções variam de 1 a 15 bytes.
 - **Curiosidade:** algumas implementações mais recentes desta arquitetura transformam as instruções x86 em operações mais simples – que se parecem com instruções MIPS – e, então, fazem a *pipeline* destas operações mais elementares.

Conjunto de Instruções e Pipeline

- O conjunto de instruções MIPS foi planejado visando uma implementação “eficiente” com *pipeline*.
 - **MIPS possui apenas alguns formatos de instrução com certa regularidade:**
 - Os campos dos registradores que fornecem operandos estão localizados no mesmo lugar em cada instrução.
 - Esta simetria abre a possibilidade de o segundo estágio da *pipeline* iniciar a leitura do arquivo de registrador ao mesmo tempo em que o *hardware* está identificando a instrução recebida.

Conjunto de Instruções e Pipeline

- O conjunto de instruções MIPS foi planejado visando uma implementação “eficiente” com *pipeline*.
 - **Operandos em memória somente aparecem nas instruções *load* e *store*:**
 - Esta restrição de projeto possibilita que o 3º estágio (execução) seja utilizado para o cálculo do endereço de memória e a referida posição seja acessada no estágio seguinte.
 - Se fosse permitida a especificação de operandos em memória, como acontece na arquitetura x86, os estágios 3 e 4 seriam expandidos em: (1) estágio de endereço, (2) estágio de acesso à memória e (3) estágio de execução.

Conjunto de Instruções e Pipeline

- O conjunto de instruções MIPS foi planejado visando uma implementação “eficiente” com *pipeline*.

➤ Alinhamento dos operandos em memória:

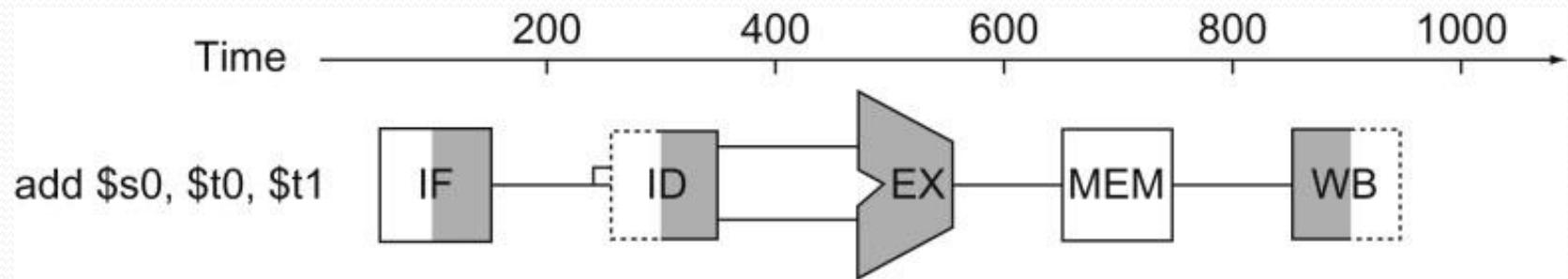
- No MIPS, as palavras sempre iniciam em endereços de memória que são múltiplos de 4.
- Por causa desta restrição (ou garantia), um acesso à memória pode ser feito em apenas um estágio.

Hazards

- Há situações especiais em que a próxima instrução do programa não pode ser executada no ciclo seguinte. Estes eventos são chamados de *hazards*.
- Veremos três tipos:
 - A. Estruturais
 - B. Dados
 - C. Controle

Hazards

- Antes, porém, vamos adotar a seguinte representação gráfica para a *pipeline* de instruções MIPS.



IF: busca de instrução.

ID: decodificação de instrução e leitura do arquivo de registradores.

EX: estágio de execução.

MEM: estágio de acesso à memória.

WB: estágio de escrita no arquivo de registradores.

OBS: fundo branco – componente não utilizado.

fundo escuro (lado direito) – componente utilizado para leitura.

fundo escuro (lado esquerdo) – componente utilizado para escrita.

Hazards

- ***Hazard estrutural:***

- Situação de conflito pelo uso (simultâneo) de um mesmo recurso de *hardware*.
- Ocorre quando duas instruções precisam utilizar o mesmo componente de *hardware* – para fins distintos ou com dados diferentes – no mesmo ciclo de relógio.
- A arquitetura MIPS foi pensada tendo em vista uma implementação em *pipeline*. Por isso, as escolhas feitas facilitam a tarefa dos projetistas em evitar os *hazards* estruturais.
- **Exemplo:** caso não houvesse duas memórias distintas (dados e instrução), teríamos um *hazard* estrutural no momento em que uma instrução estivesse no 4º estágio da *pipeline* (para efetuar um acesso à memória) e uma nova instrução estivesse para ser buscada.

Hazards

- ***Hazard de dados:***

- Ocorrem quando uma instrução depende da conclusão de uma instrução prévia que ainda esteja na *pipeline* para realizar sua operação e/ou acessar um dado.

- **Exemplo:**

- $\text{add } \$s0, \$t0, \$t1$

- $\text{sub } \$t2, \$s0, \$t3$

- A instrução *add* somente escreve seu resultado no final do 5º estágio da *pipeline*.
 - Logo, teríamos que desperdiçar três ciclos de relógio aguardando até que o resultado correto ($\$s0$) pudesse ser lido pela instrução *sub*.

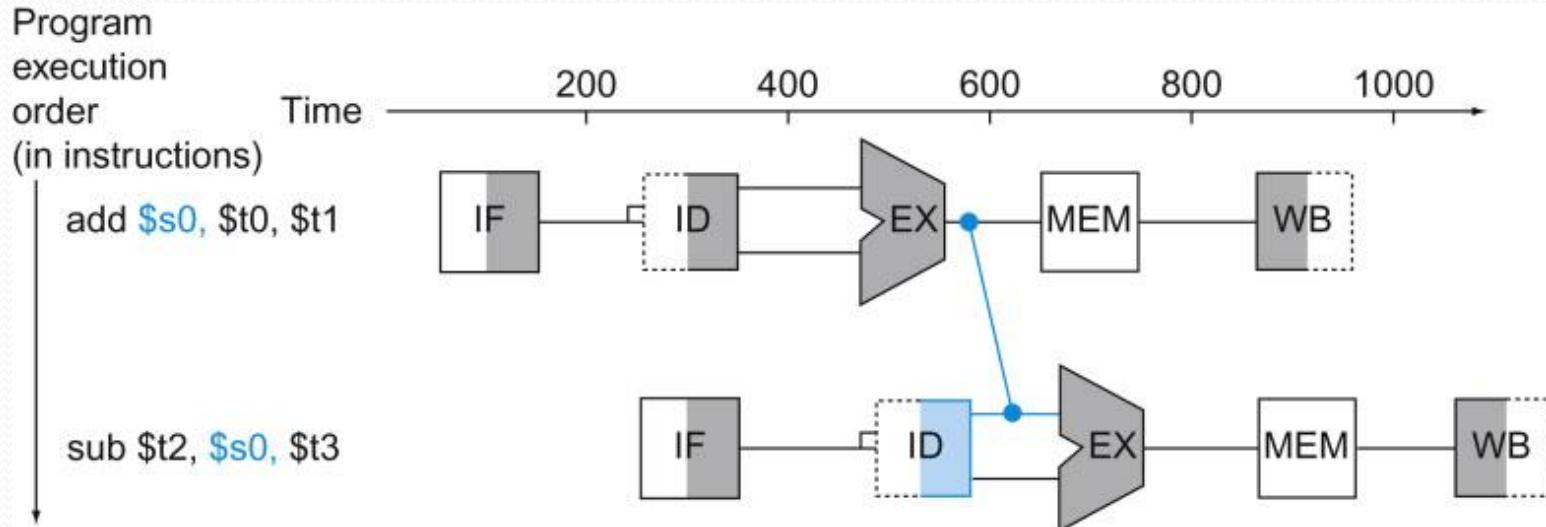
Hazards

- ***Hazard de dados:***

- Uma possível solução pode ser proposta a partir da observação de que não é necessário aguardar até que a instrução termine sua execução.
- No exemplo anterior, assim que a ALU cria o resultado da soma prevista pela instrução *add*, este valor poderia ser passado como entrada para a subtração no ciclo seguinte.
- A inserção de *hardware* extra para recuperar uma informação referente a uma instrução passada caracteriza a estratégia conhecida como *forwarding* ou *bypassing*.

Hazards

- *Hazard de dados:*

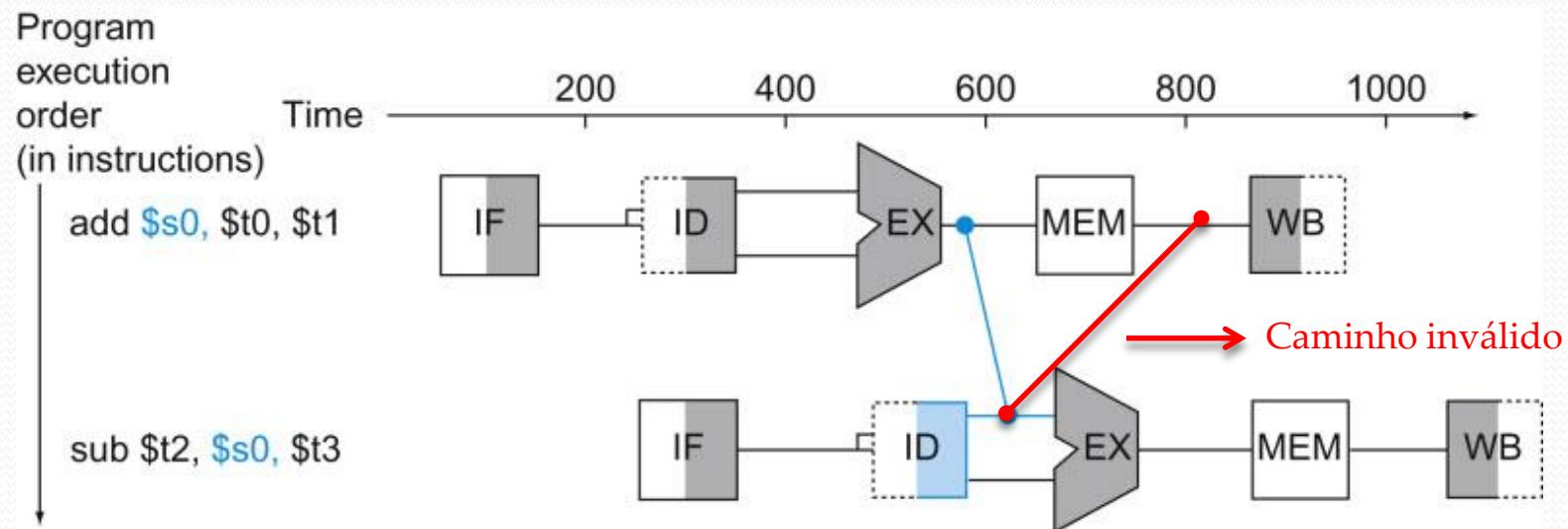


A conexão destacada mostra o caminho de *forwarding* da saída do estágio EX da instrução *add* para uma entrada do estágio EX da instrução *sub*, substituindo o valor do registrador *\$s0* lido no segundo estágio (ID).

Hazards

- *Hazard de dados:*

- Caminhos de *forwarding* são válidos somente se o estágio de destino estiver à frente no tempo que o estágio de origem da informação.



Hazards

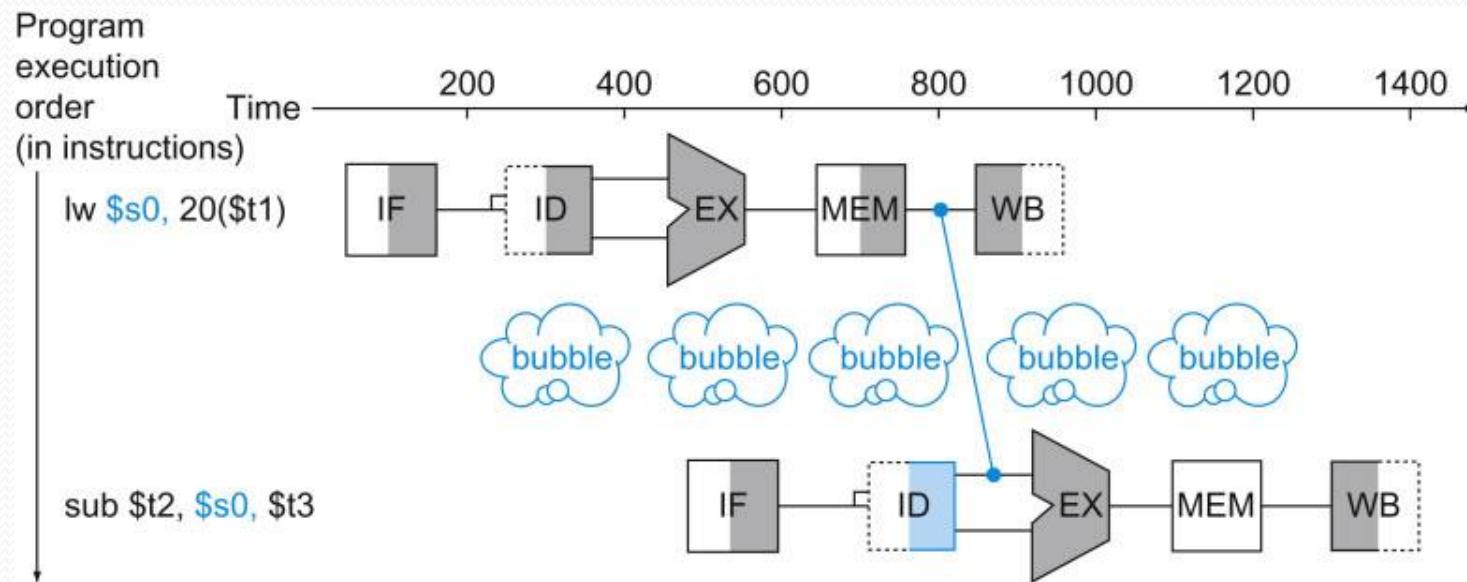
- **Hazard de dados:**

- A estratégia de *forwarding* se mostra bastante útil, mas não é capaz de evitar todos os *hazards* de dados.
- **Exemplo:**
`lw $s0, 20($t1)
sub $t2, $s0, $t3`
- O valor de correto de `$s0` só estará disponível após o 4º estágio de execução da instrução *lw*, o que é tarde demais para que, no 3º estágio, a instrução *sub* possa utilizar o valor correto.
- Logo, é necessário introduzir um atraso de um estágio (ciclo) na *pipeline* – *pipeline stall* ou *bubble*.
- Este tipo particular de *hazard* é conhecido como *load-use data hazard*.

Hazards

- *Hazard de dados:*

- A estratégia de *forwarding* se mostra bastante útil, mas não é capaz de evitar todos os *hazards* de dados.



Hazards

- *Hazard de dados:*

- **Exercício:** Identifique os *hazards* que podem ser resolvidos por meio da reordenação das instruções.

```
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)
```

13 ciclos

Após reordenação →

```
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)
```

11 ciclos

Observação: os demais *hazards* podem ser contornados via *forwarding*.

Hazards

- ***Hazard de dados:***

- Outra vantagem da arquitetura MIPS:
 - Cada instrução MIPS escreve, no máximo, um resultado e o faz apenas no último estágio da *pipeline*.
 - Nesta condição, é mais fácil aplicar a estratégia de *forwarding*.
 - Caso oposto:
 - ✓ Múltiplos resultados precisam ser adiantados via *forwarding* por instrução.
 - ✓ É necessário escrever um resultado antes do fim da execução de uma instrução (i.e., em um estágio anterior da *pipeline*)
- A estratégia de *forwarding* se torna mais complexa.

Hazards

- **Hazard de controle:**

- Surge por causa da necessidade de tomar uma decisão baseada em resultados de uma instrução enquanto outras estão em execução.
 - Ou seja, está ligado a instruções do tipo *branch*.
-
- **Problema:**
 - A *pipeline* inicia a busca da instrução subsequente ao *branch* no próximo ciclo de relógio.
 - Porém, não há como a *pipeline* saber qual é a instrução correta a ser buscada, uma vez que acabou de receber o próprio *branch* da memória.

Hazards

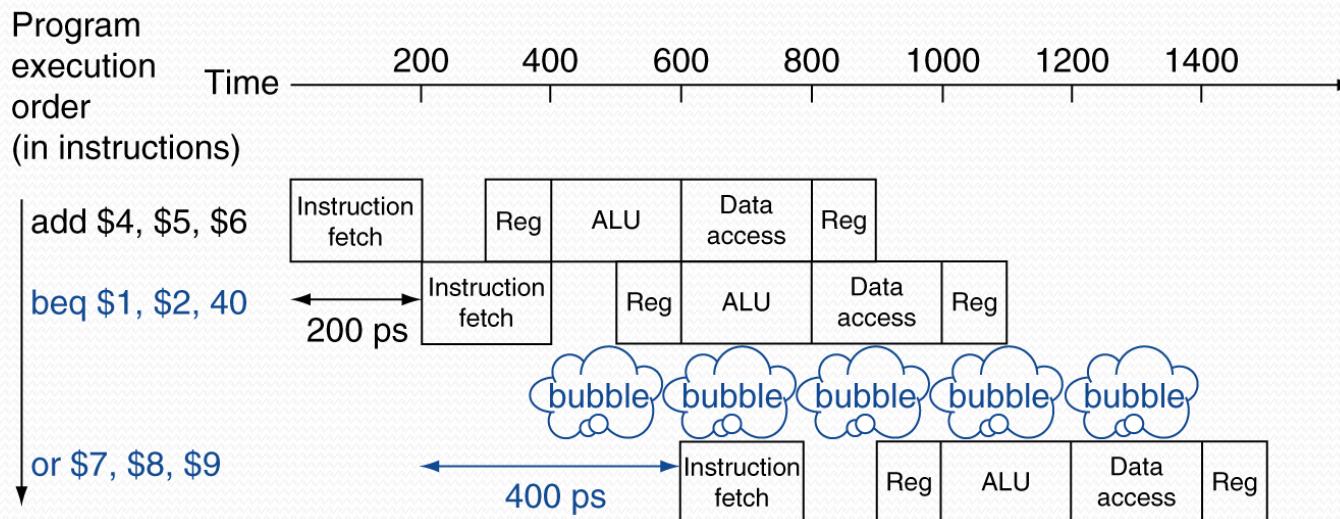
- *Hazard de controle:*

- **Solução conservadora:** esperar até que o *branch* tenha testado sua condição e o endereço da próxima instrução esteja determinado.
- **Suposição:** há hardware extra que possibilita (1) o teste dos registradores, (2) o cálculo do endereço de desvio e (3) a atualização de PC durante o segundo estágio da *pipeline*.

Hazards

- *Hazard de controle:*

- Se o desvio for tomado:



- Mesmo que o desvio não seja tomado, haverá um atraso de 200ps.
- Este cenário fica mais dramático se não for possível resolver o desvio logo no segundo estágio da *pipeline*.
- Para *pipelines* mais longas, o atraso devido aos *stalls* se torna inaceitável.

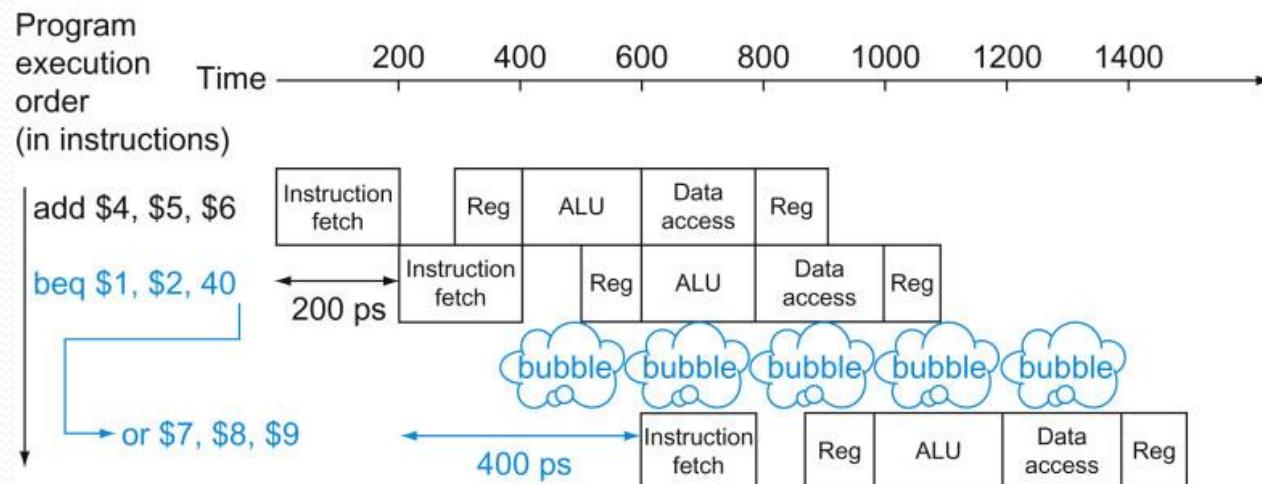
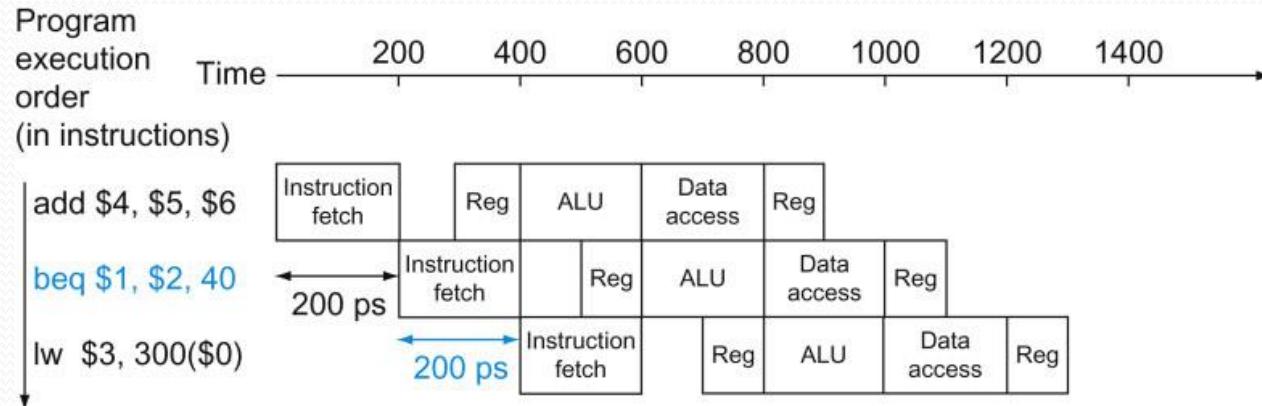
Hazards

- *Hazard de controle:*

- **Solução alternativa:** prever qual é a próxima instrução.
- **1^a possibilidade:** considerar que o desvio nunca será tomado.
 - Quando o palpite estiver correto, não haverá atrasos na *pipeline*.
 - Somente quando o desvio for tomado é que algum atraso será introduzido.

Hazards

- *Hazard de contrôle:*



Hazards

- **Hazard de controle:**

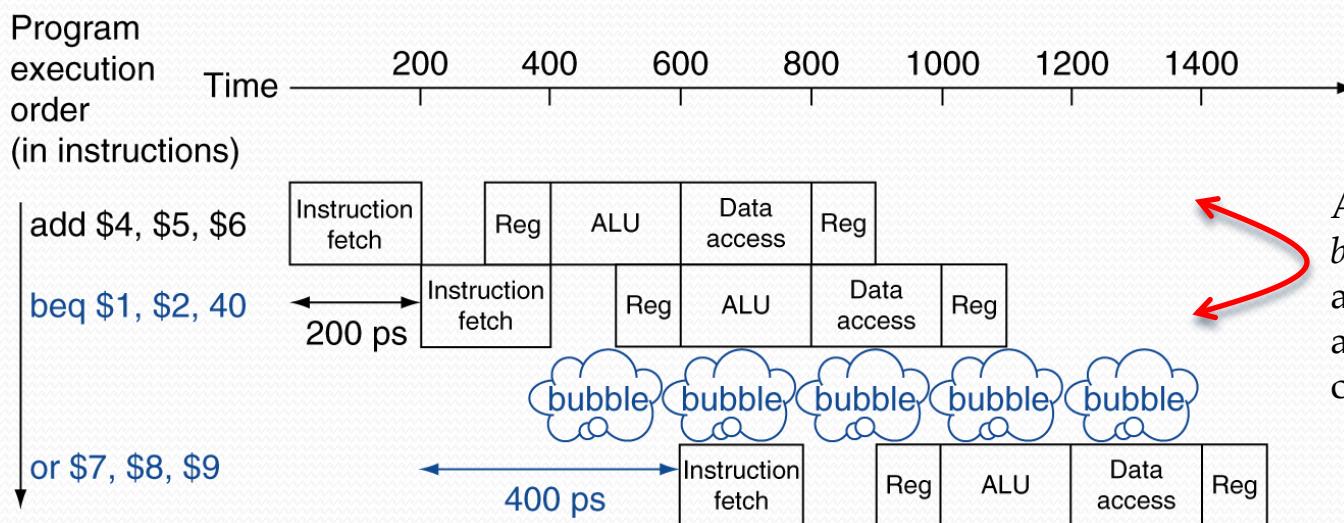
- **2^a possibilidade:** usar um mecanismo estático de previsão de desvios, baseado no comportamento típico (esperado) de alguns desvios.
 - **Exemplos:**
 - ✓ *Branches* no fim de estruturas como laços (para trás) – assumir que sempre será tomado.
 - ✓ *Branches* para frente – assumir que nunca será tomado.
- **3^a possibilidade:** previsão dinâmica
 - As previsões dependem do comportamento de cada *branch* durante a própria execução de um programa.
 - Um histórico do comportamento (tomado/não tomado) de cada *branch* é mantido e atualizado.
 - Sempre que a previsão for incorreta, é preciso atrasar a *pipeline* enquanto é realizada a busca da nova instrução.

Hazards

- *Hazard de controle:*

- 4^a possibilidade: desvio atrasado

- Solução adotada pelo MIPS.
- Uma instrução que não seja afetada pelo desvio é inserida imediatamente após o *branch* e sempre é executada.
- Caso o desvio seja tomado, ele altera o endereço da instrução posterior a esta que foi colocada após ele.

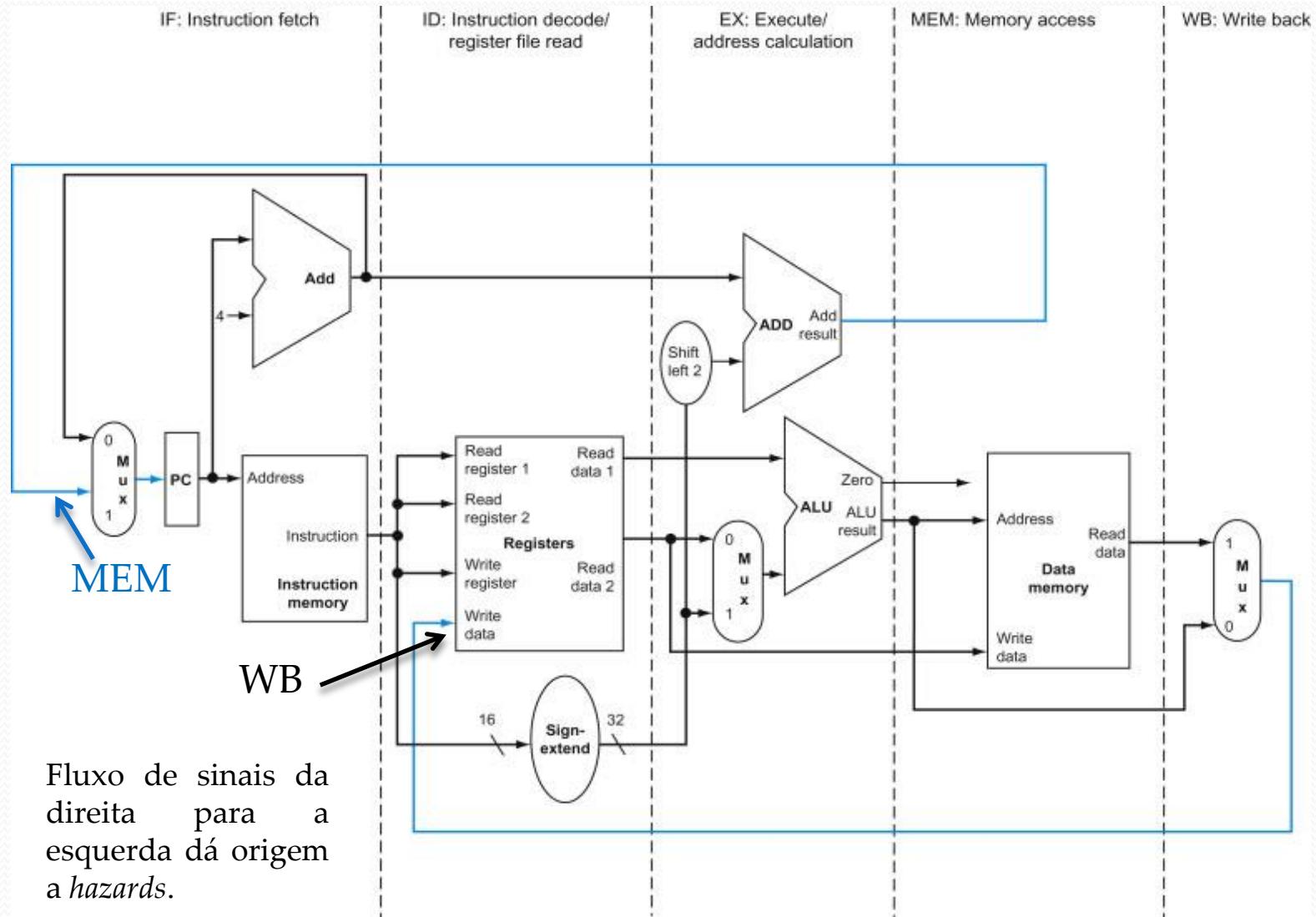


A instrução *add* não afeta o *branch* e pode ser movida para a posição imediatamente após a instrução *beq*, escondendo completamente o atraso.

Datapath e Controle com Pipeline

- Usaremos como base a implementação uniciclo de um subconjunto da arquitetura MIPS vista no tópico anterior.
- Vamos separar o *datapath* em cinco partes, uma para cada estágio da *pipeline*: IF, ID, EX, MEM e WB.
- Depois, vamos analisar possíveis ajustes tanto em termos de unidades de *hardware* quanto em relação aos sinais de controle para que a implementação em *pipeline* seja bem-sucedida.

Datapath e Controle com Pipeline

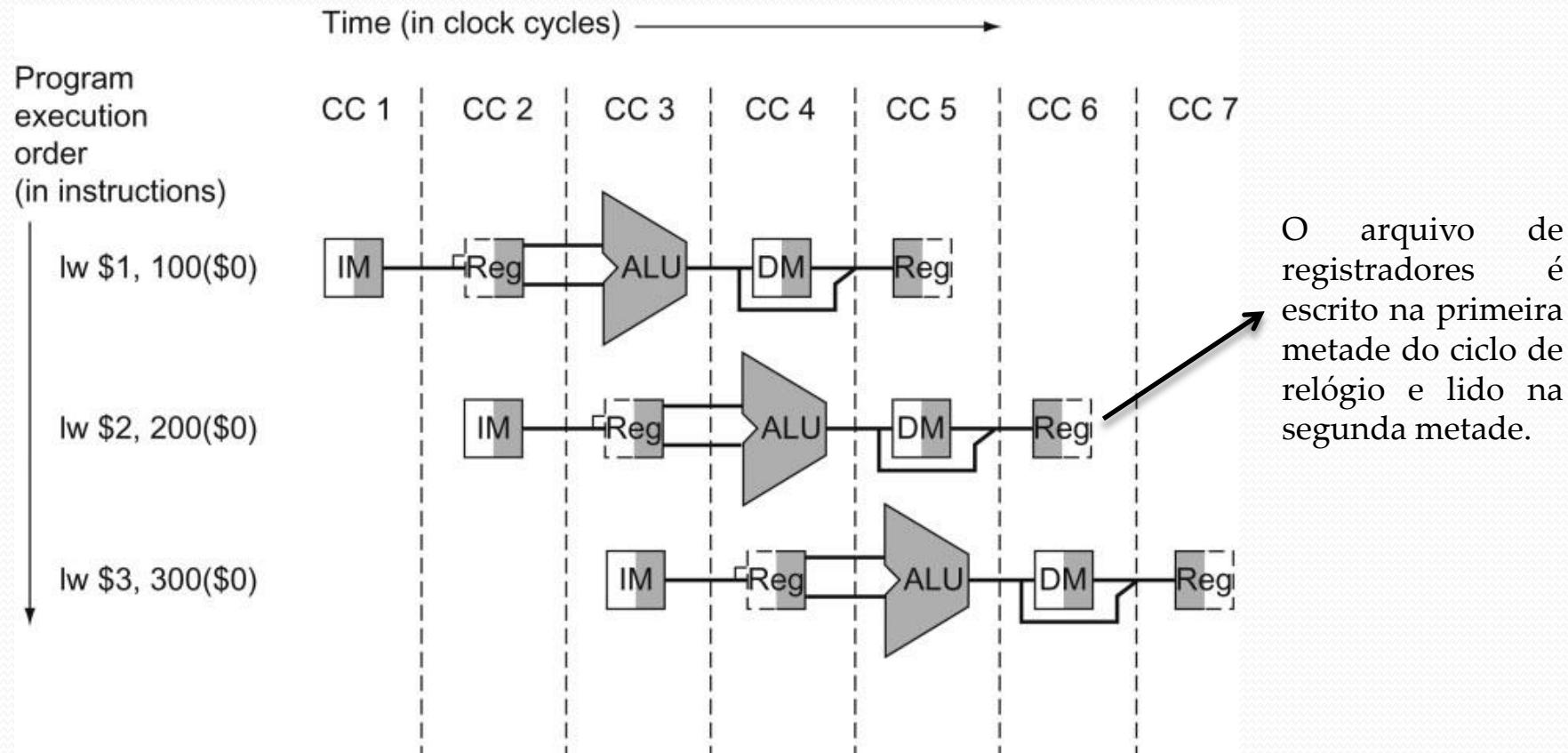


Datapath e Controle com Pipeline

- Usualmente, os dados fluem da esquerda para a direita, avançando nos estágios da *pipeline*.
- Exceções:
 - Estágio WB, que coloca o resultado de volta no arquivo de registradores (no meio do *datapath*).
 - A definição do próximo valor de PC: ou o PC incrementado ou o endereço de desvio gerado no estágio MEM.

Datapath e Controle com Pipeline

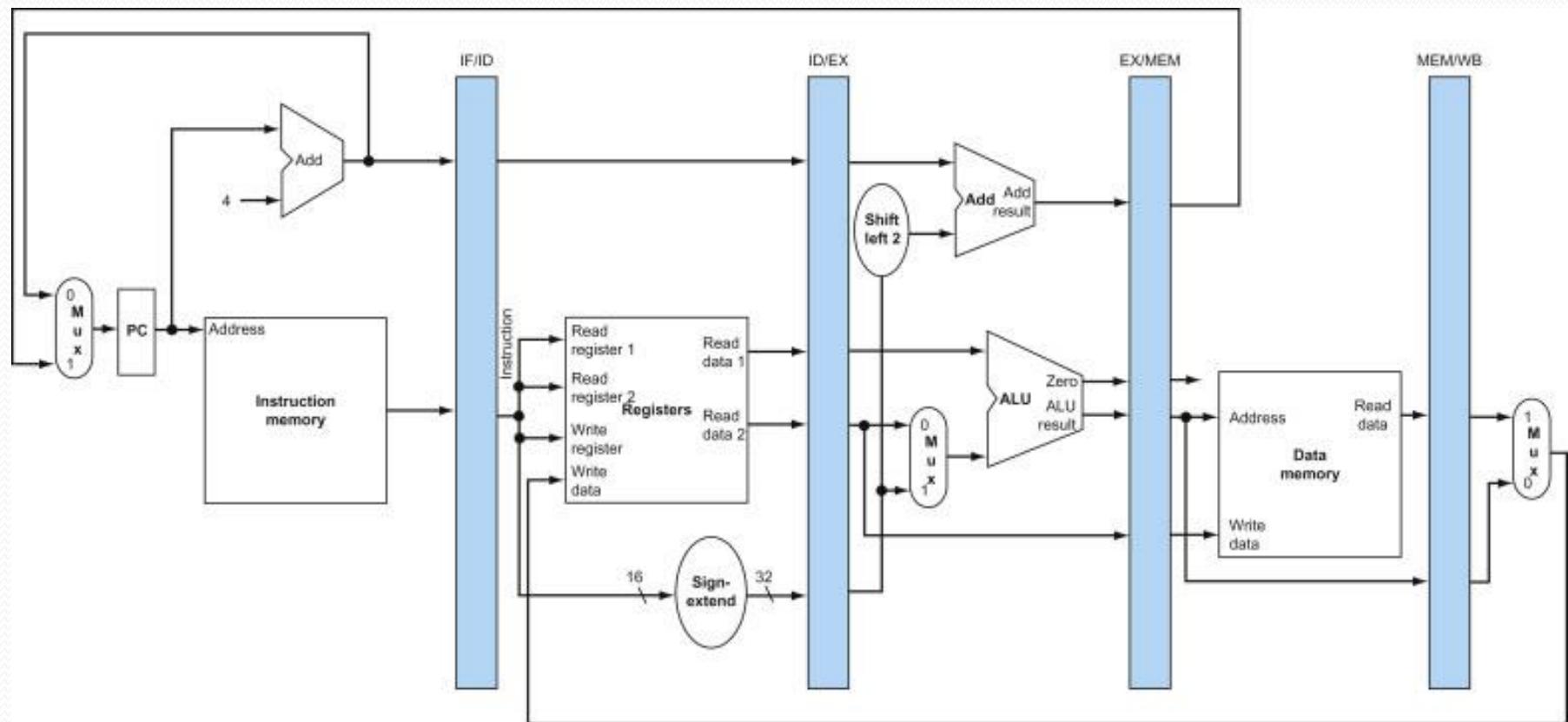
- Representação visual:



Fica claro que é preciso, de alguma forma, guardar e passar informações para os estágios seguintes da *pipeline* de acordo com a instrução lida.

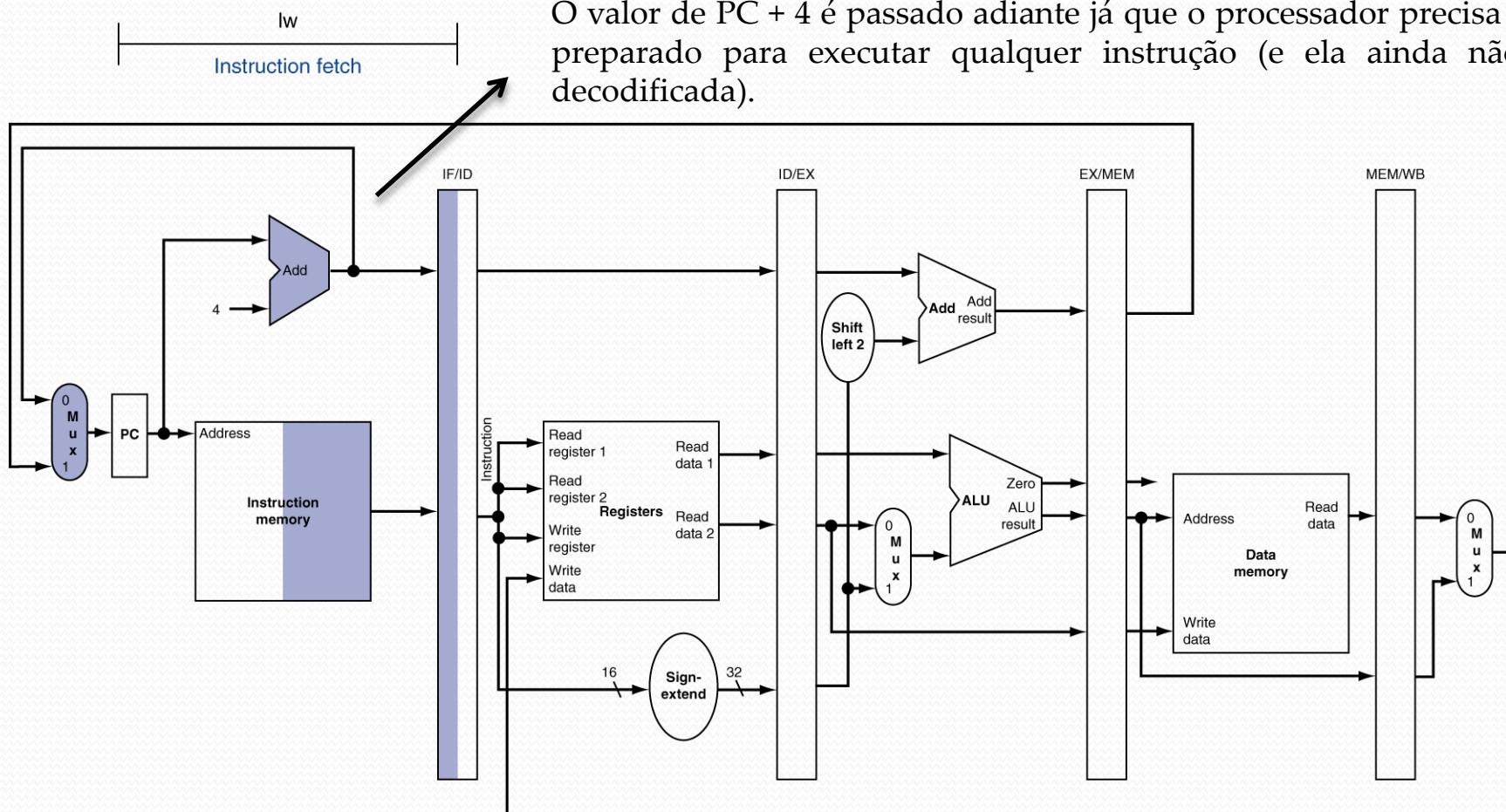
Datapath e Controle com Pipeline

- Solução: acrescentar registradores na fronteira entre estágios.



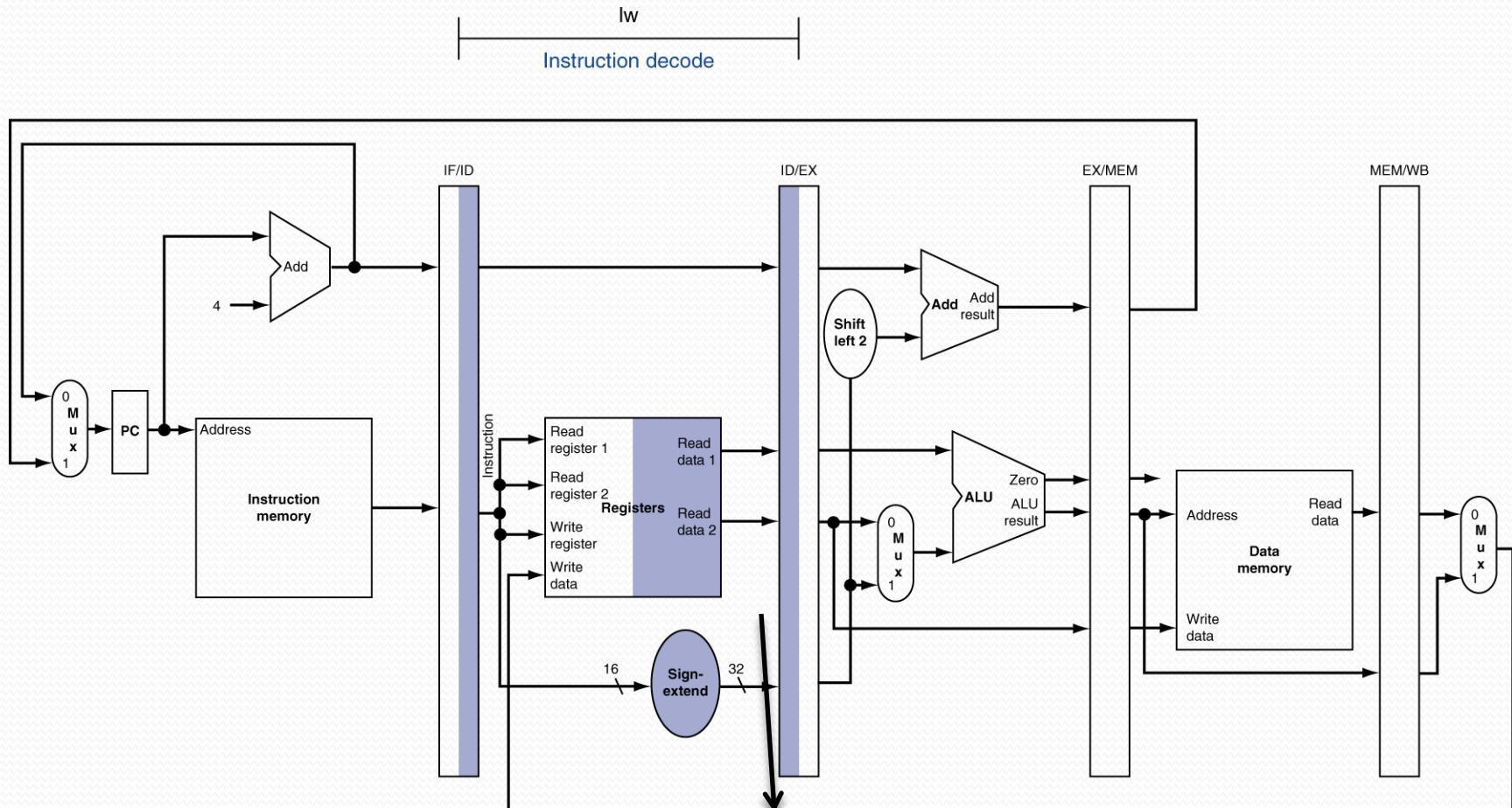
Datapath e Controle com Pipeline

- Exemplo: diagrama de único ciclo – instrução *load*.



Datapath e Controle com Pipeline

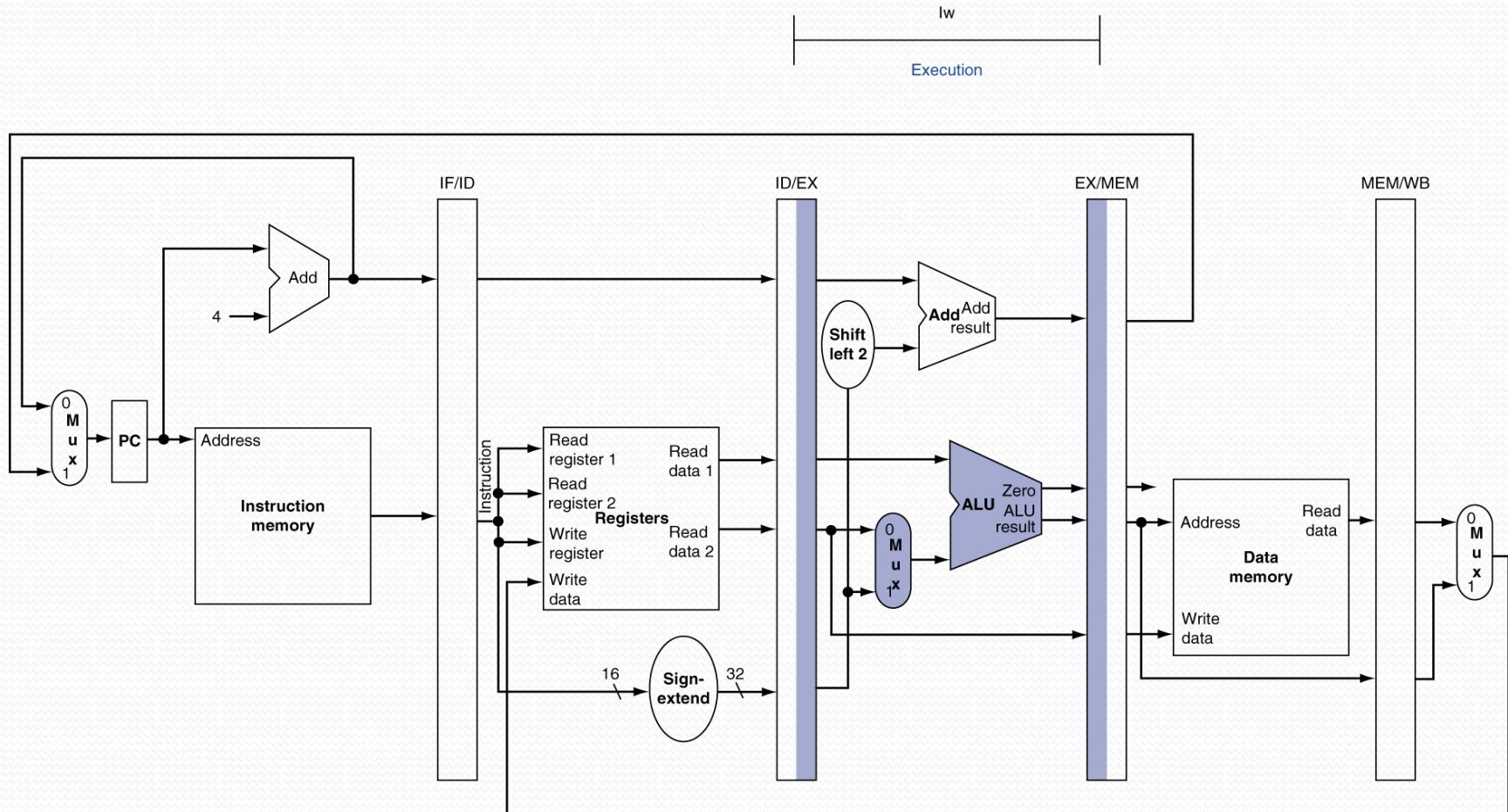
- Exemplo: diagrama de único ciclo – instrução *load*.



Dois registradores, o valor de PC + 4 e a extensão de sinal do campo imediato são colocados em ID/EX independentemente da instrução.

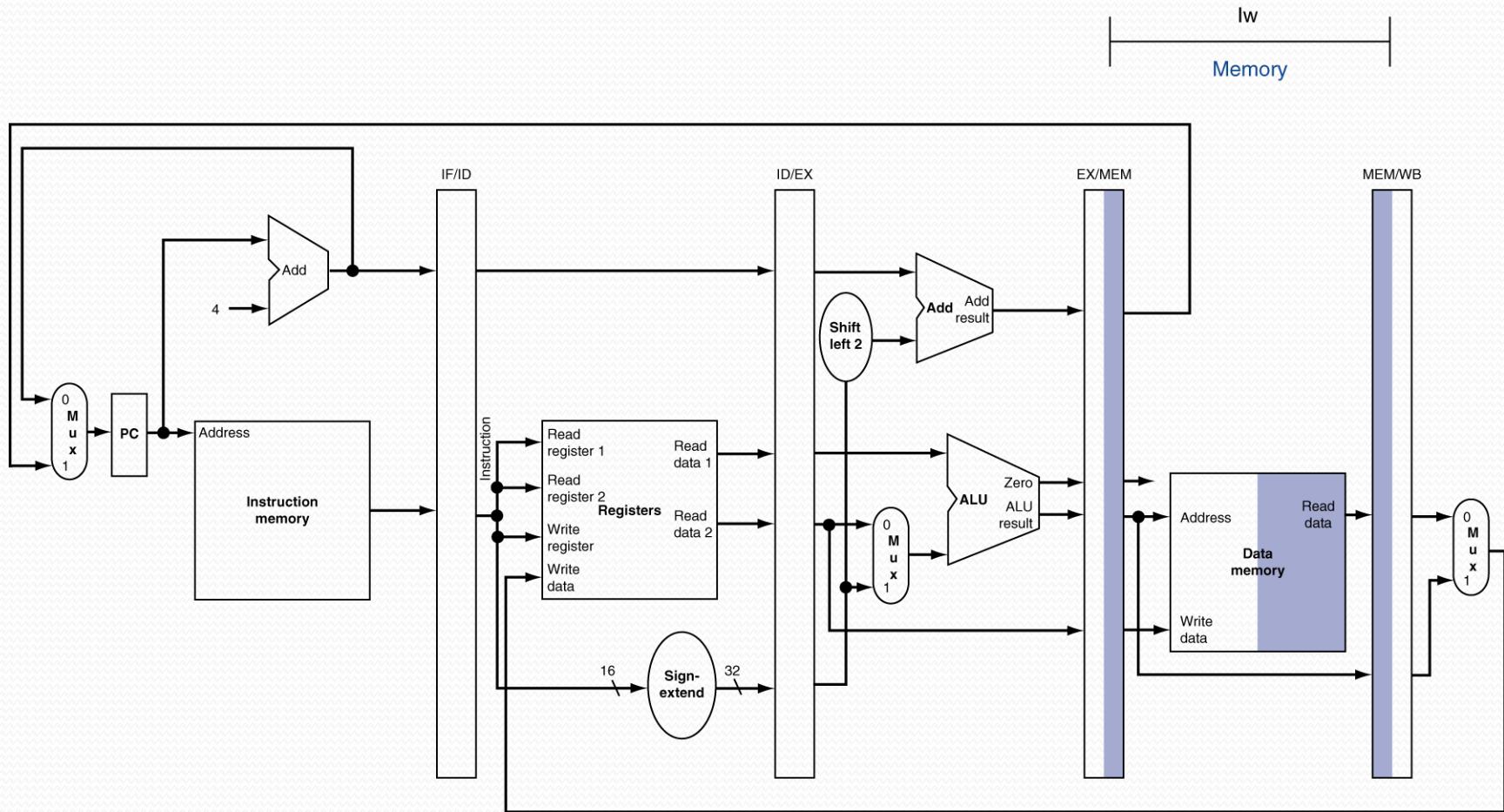
Datapath e Controle com Pipeline

- Exemplo: diagrama de único ciclo – instrução *load*.



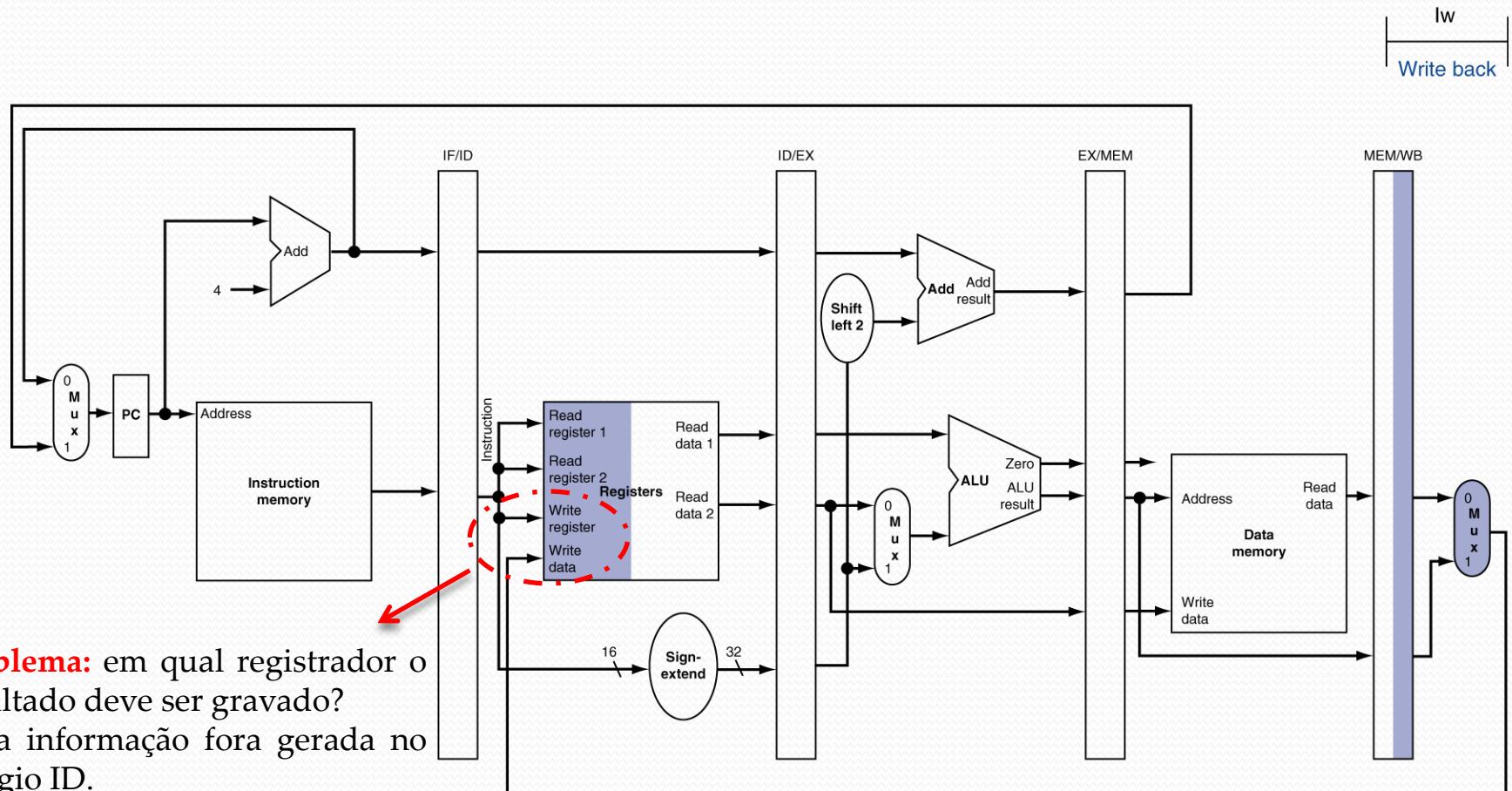
Datapath e Controle com Pipeline

- Exemplo: diagrama de único ciclo – instrução *load*.



Datapath e Controle com Pipeline

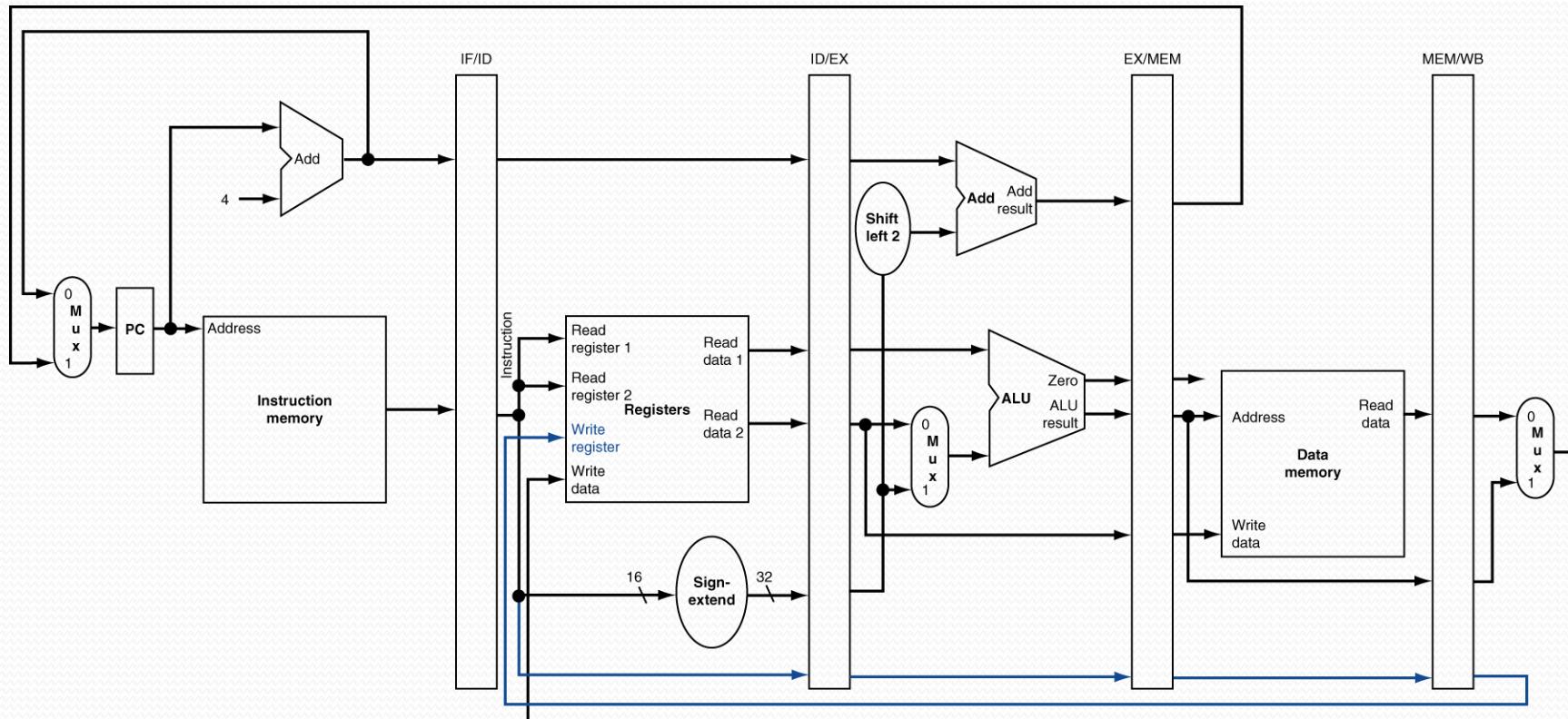
- Exemplo: diagrama de único ciclo – instrução *load*.



Problema: em qual registrador o resultado deve ser gravado?
- Esta informação fora gerada no estágio ID.

Datapath e Controle com Pipeline

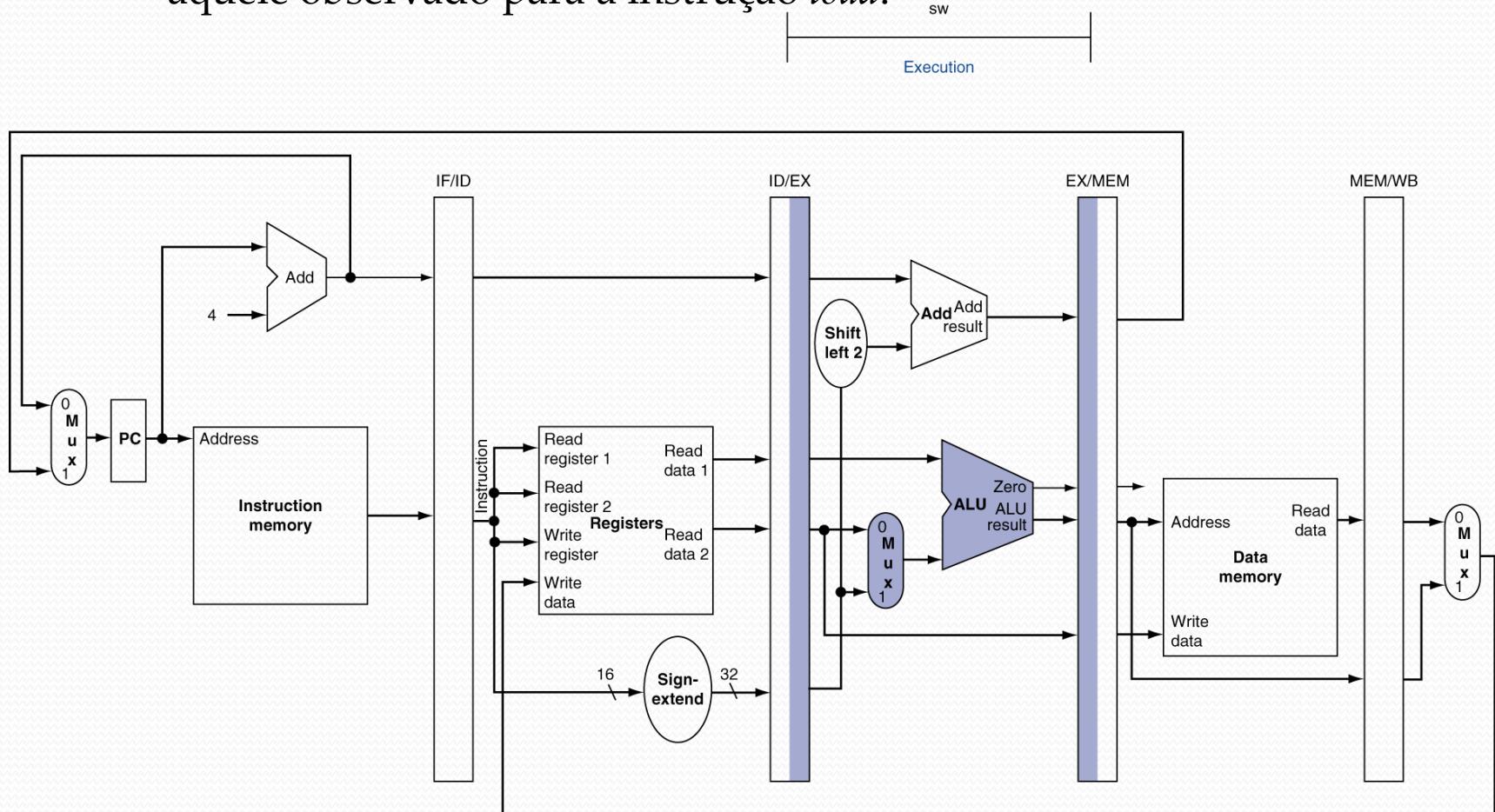
- Exemplo: diagrama de único ciclo – instrução *load*.



Solução: passar adiante (através dos registradores da *pipeline*) o índice do registrador de destino até chegar a MEM/WB.

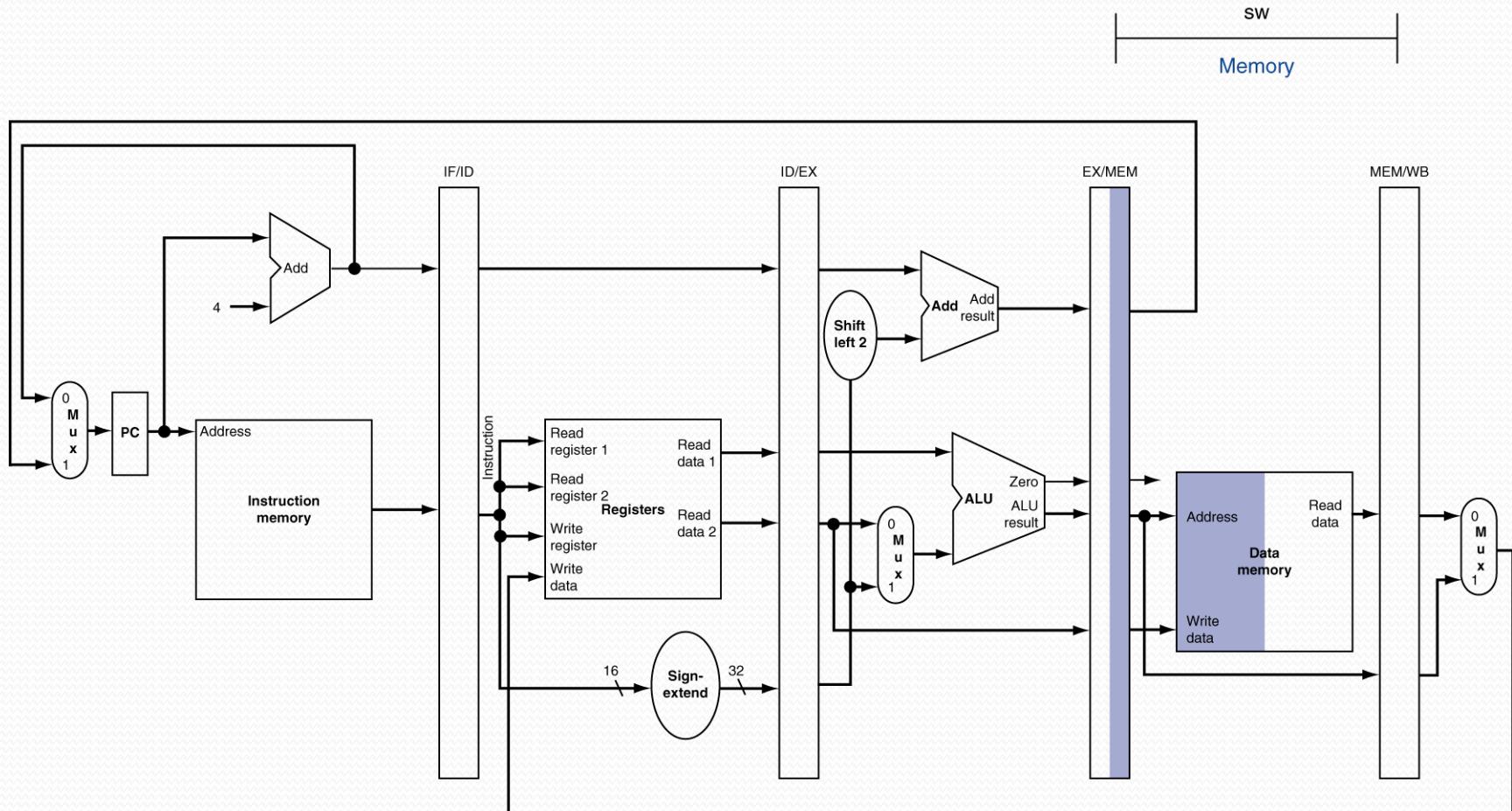
Datapath e Controle com Pipeline

- Exemplo: diagrama de único ciclo – instrução *store*.
 - O comportamento nos dois primeiros ciclos é exatamente o mesmo que aquele observado para a instrução *load*.



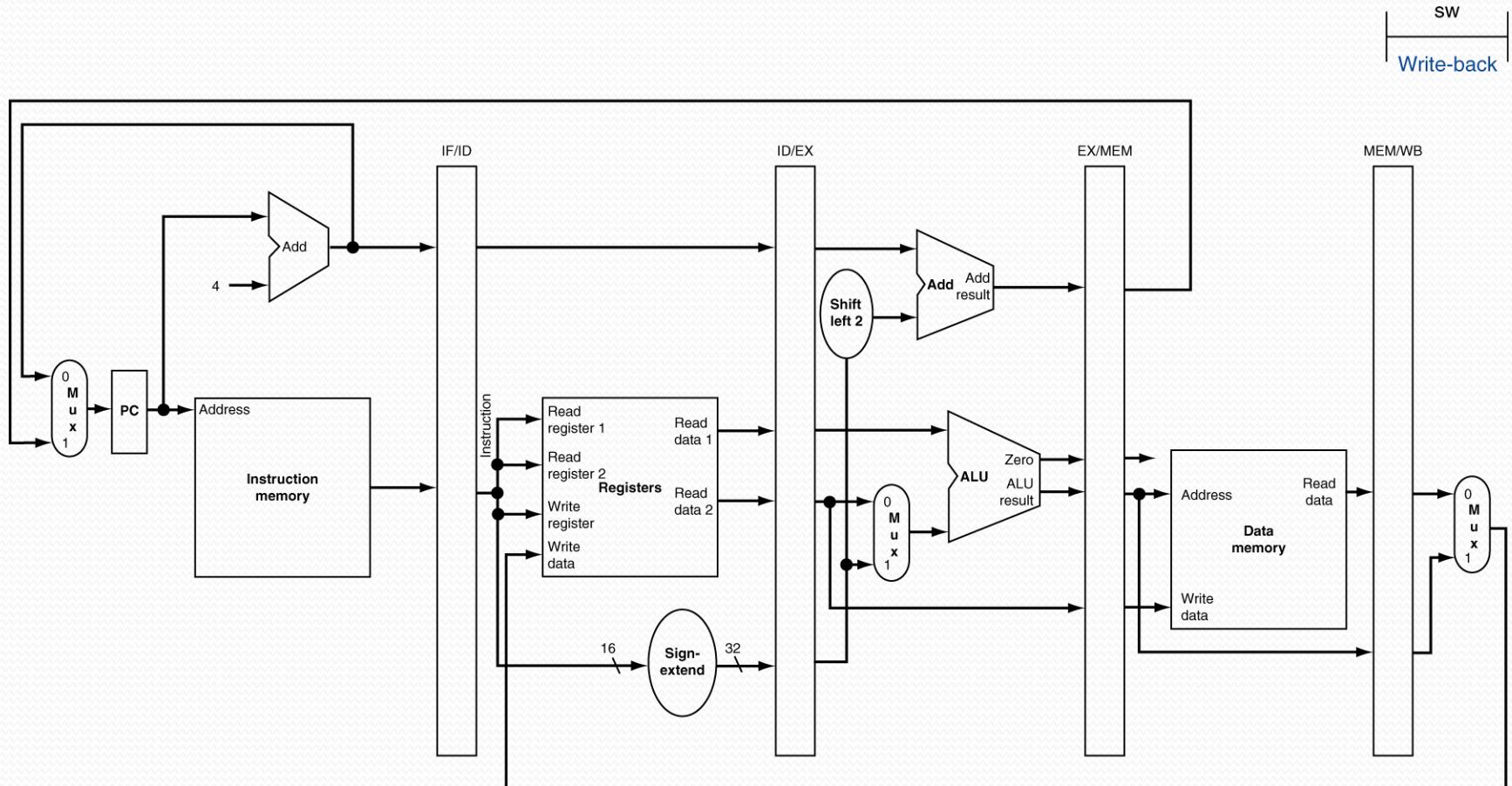
Datapath e Controle com Pipeline

- Exemplo: diagrama de único ciclo – instrução *store*.



Datapath e Controle com Pipeline

- Exemplo: diagrama de único ciclo – instrução *store*.

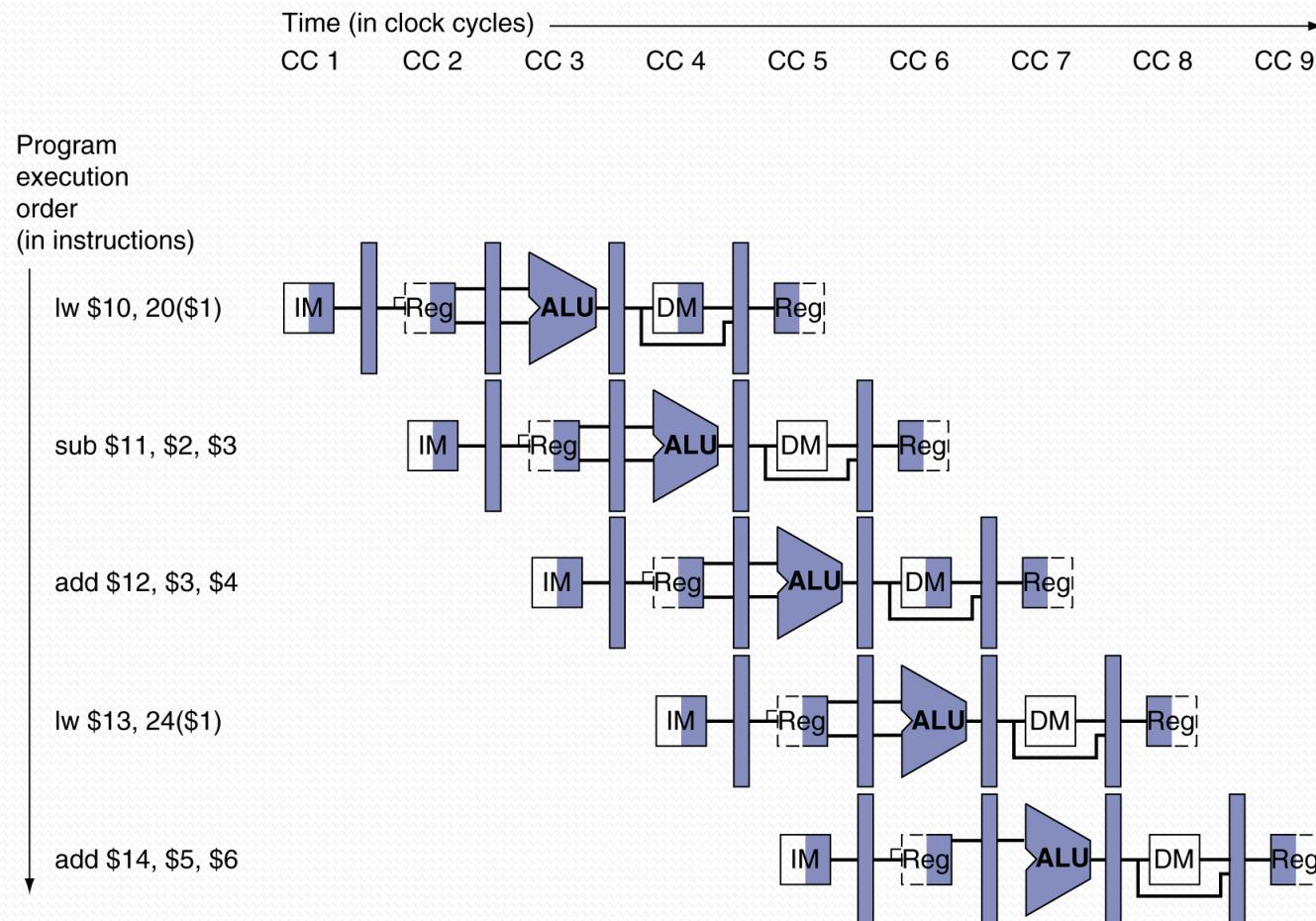


Datapath e Controle com Pipeline

- Através destes dois exemplos, vimos alguns pontos fundamentais para a implementação em *pipeline*:
 - Passagem de sinais e dados de estágios anteriores para estágios posteriores da *pipeline*.
 - Cada componente do *datapath* – memória de instrução, portas de leitura de registradores, ALU, memória de dados e porta de escrita de registrador – só pode ser utilizado dentro de um único estágio da pipeline.
Caso contrário, teríamos um **hazard estrutural**.

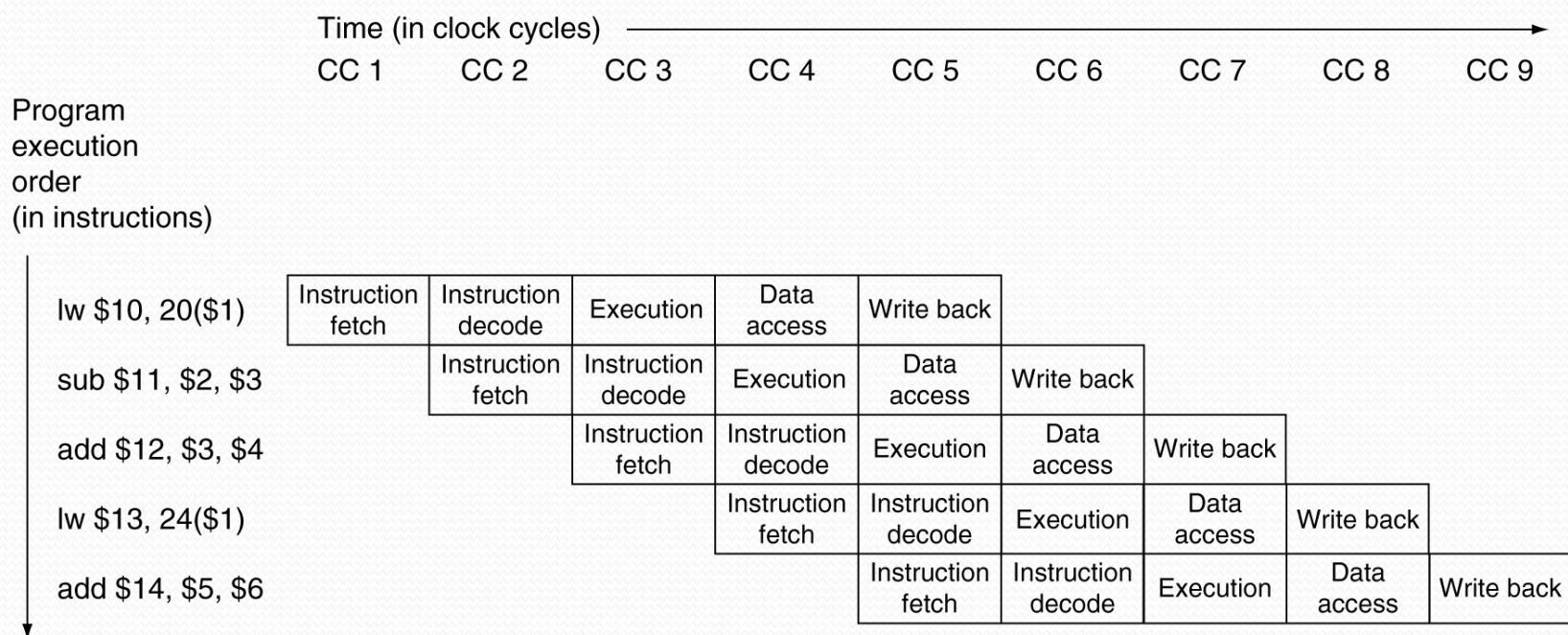
Datapath e Controle com Pipeline

- Duas formas de ilustrar o funcionamento de uma *pipeline*:
 - Um único diagrama com réplicas do *datapath* (ou dos estágios da *pipeline*) para vários ciclos de relógio.



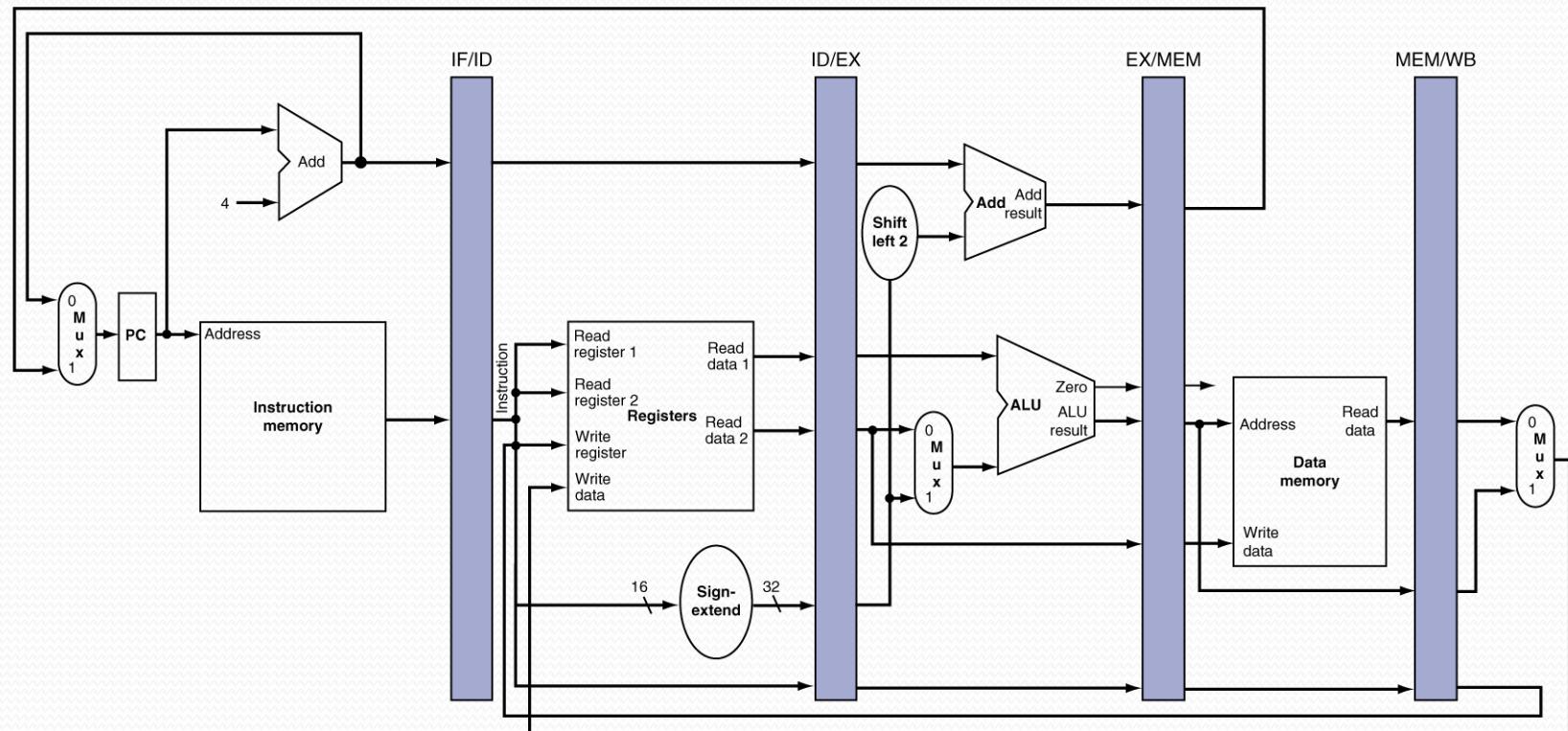
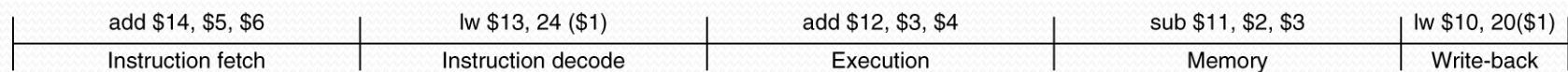
Datapath e Controle com Pipeline

- Duas formas de ilustrar o funcionamento de uma *pipeline*:
 - Um único diagrama com réplicas do *datapath* (ou dos estágios da *pipeline*) para vários ciclos de relógio.



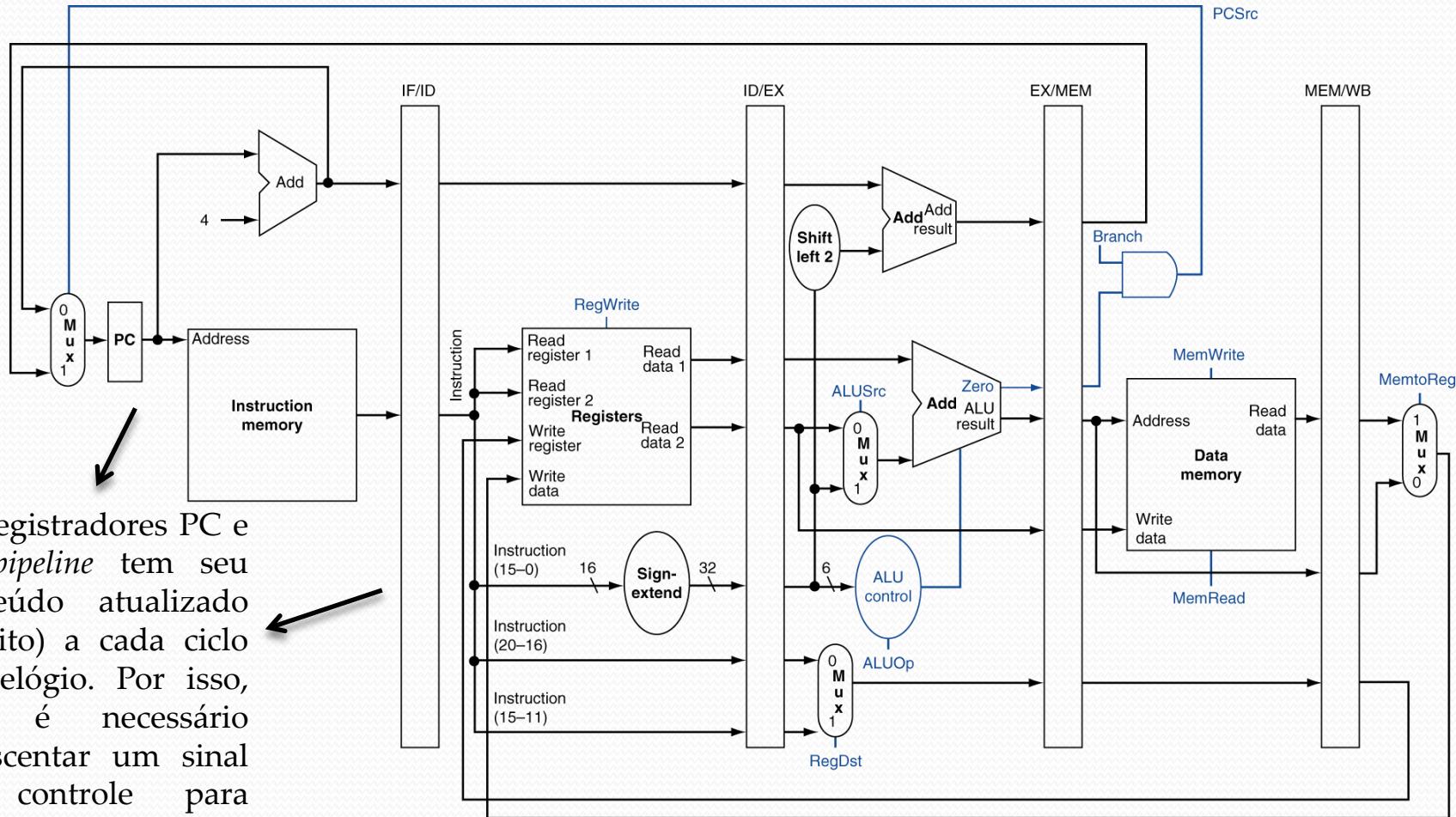
Datapath e Controle com Pipeline

- Duas formas de ilustrar o funcionamento de uma *pipeline*:
 - Vários diagramas do *datapath*, cada um para um ciclo particular de relógio.



Datapath e Controle com Pipeline

- Controle:



Datapath e Controle com Pipeline

- **Controle:**

- **IF:** os sinais de controle para ler a instrução e para escrever em PC estão sempre ativos.
- **ID:** não há linhas de controle opcionais, uma vez que a operação neste estágio sempre é a mesma, independentemente da instrução.
- **EX:** os sinais que precisam ser ajustados são RegDst, ALUOp e ALUSrc.
- **MEM:** as linhas de controle deste estágio são Branch, MemRead e MemWrite.
- **WB:** as linhas de controle deste estágio são MemtoReg, que decide entre enviar o resultado da ALU ou um valor de memória para o arquivo de registradores, e RegWrite, que escreve o valor selecionado.

Datapath e Controle com Pipeline

- **Controle:**

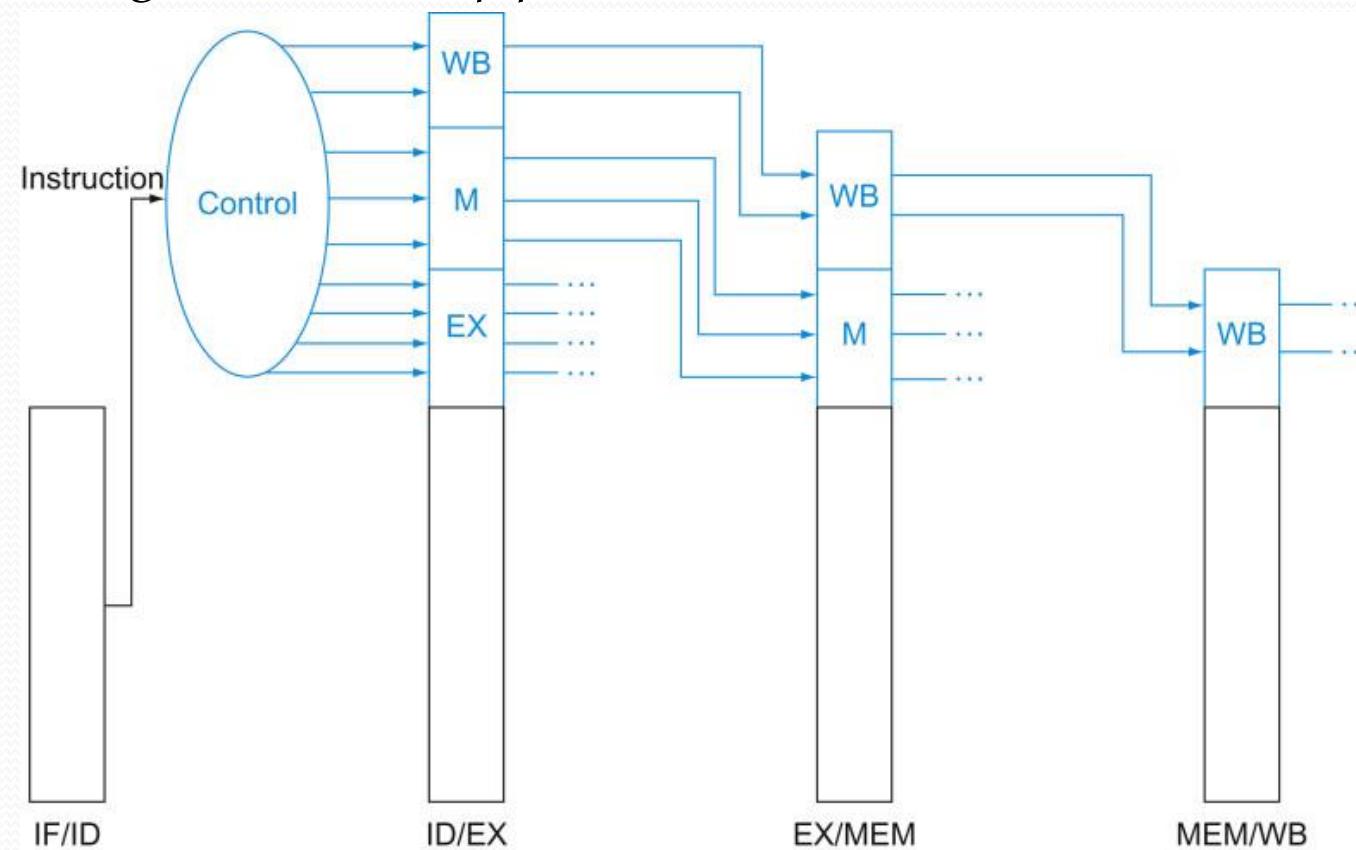
- Não houve alteração do significado dos sinais de controle em relação aos da implementação uniciclo.
- Convém, apenas, agrupá-los de acordo com o estágio da *pipeline* em que exercem algum efeito.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Datapath e Controle com Pipeline

- **Controle:**

- As informações de controle podem ser criadas durante o estágio de decodificação da instrução (ID) e passadas adiante através dos registradores de *pipeline*.



Hazard de Dados

- Agora que vimos as modificações no *datapath* necessárias para o funcionamento da *pipeline*, vamos analisar as complicações produzidas por *hazards* de dados e como amenizá-las via *forwarding* ou via *stalls*.
- Exemplo:

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

O valor de **\$2** é atualizado pela operação *sub*. O programador certamente deseja que todas as instruções seguintes utilizem o novo valor contido em **\$2**.

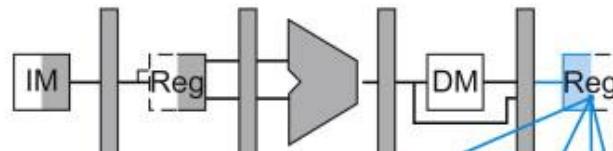
Hazard de Dados

Time (in clock cycles)

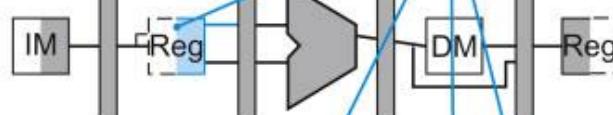
Value of register \$2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/-20	-20	-20	-20	-20

Program execution order (in instructions)

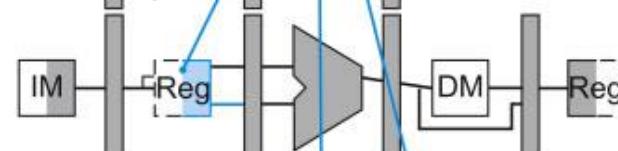
sub \$2, \$1, \$3



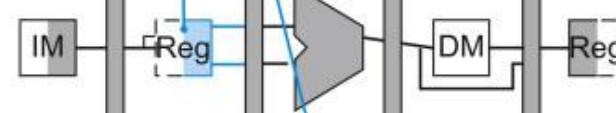
and \$12, \$2, \$5



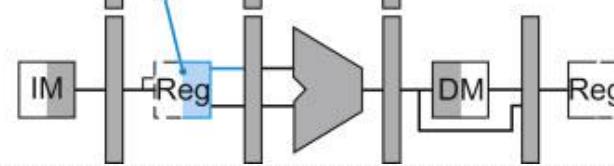
or \$13, \$6, \$2



add \$14, \$2,\$2



sw \$15, 100(\$2)



O valor correto em **\$2** só seria utilizado pelas instruções *add* (já que, por construção, o arquivo de registradores é escrito na primeira metade do ciclo e é lido na segunda metade) e *sw*.



Hazard de Dados

- Quando o valor de **\$2** está disponível?
 - No final do estágio EX (3º ciclo).
- Quando o valor de **\$2** é realmente necessário para as operações AND e OR?
 - No início do estágio EX (4º e 5º ciclos, respectivamente).
- Neste caso, é possível passar adiante (*forwarding*) o valor de **\$2** assim que ele estiver pronto para quaisquer unidades que precisem dele antes de ele ser armazenado no arquivo de registradores.

Hazard de Dados

- Como implementar a estratégia de *forwarding*?
 - Primeiro, é preciso escrever as condições de dependência de dados que caracterizam um *hazard*.
 - Por simplicidade, vamos considerar apenas o caso de *forwarding* para uma instrução que está no estágio EX.
 - **Notação:**
 - ID/EX.RegisterRs = número do registrador Rs que está armazenado no registrador de *pipeline* ID/EX.
 - Os operandos da ALU – estágio EX – podem ser identificados como ID/EX.RegisterRs e ID/Ex.RegisterRt, respectivamente.

Hazard de Dados

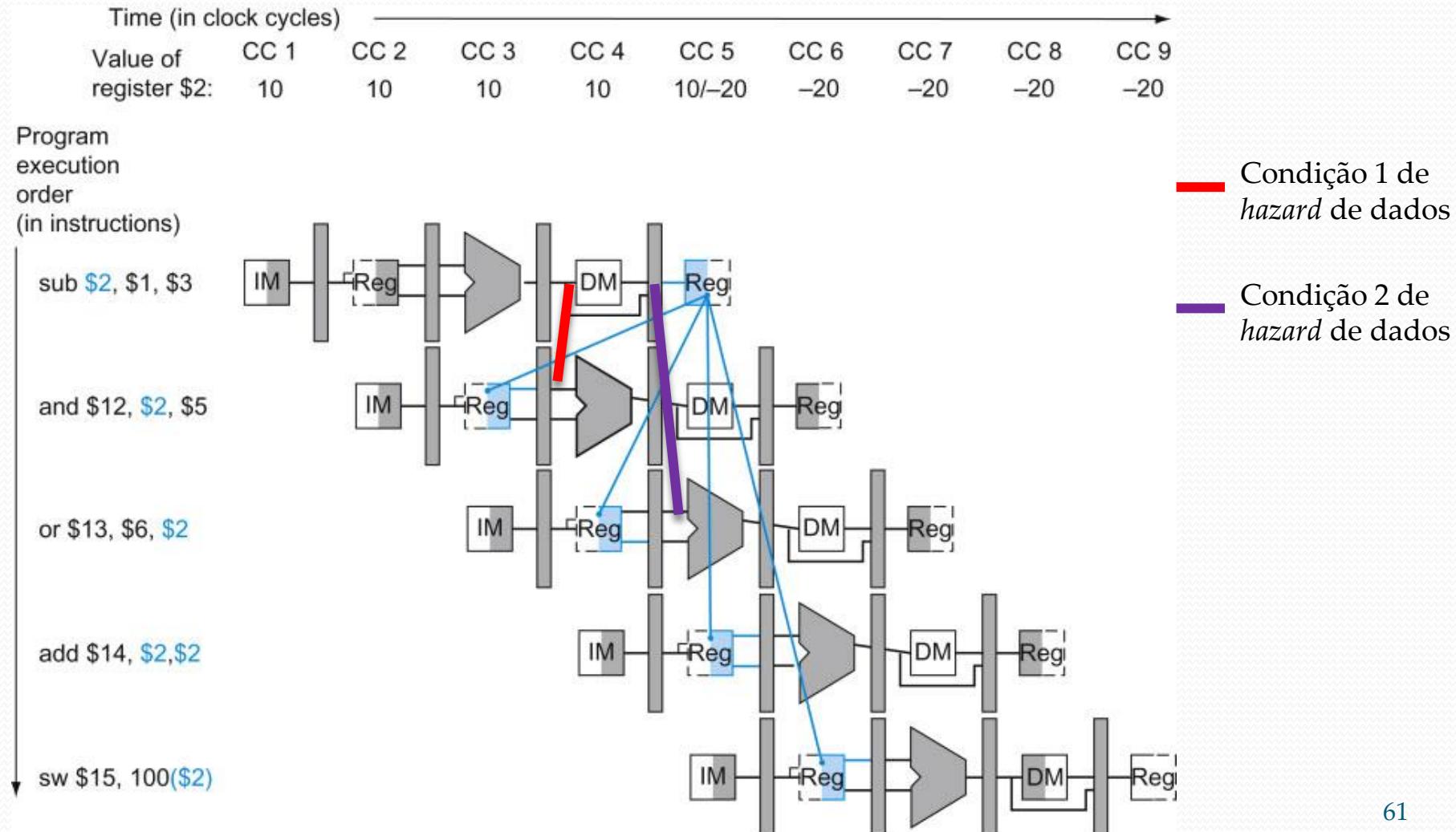
- Como implementar a estratégia de *forwarding*?
 - Usando a notação anterior, as seguintes condições descrevem a ocorrência dos *hazards* de dados:

Solução

- | | | |
|--|---|---|
| 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt | } | <i>Forwarding</i> a partir do registrador de pipeline EX/MEM. |
| 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt | | <i>Forwarding</i> a partir do registrador de pipeline MEM/WB. |

Hazard de Dados

- Como implementar a estratégia de *forwarding*?

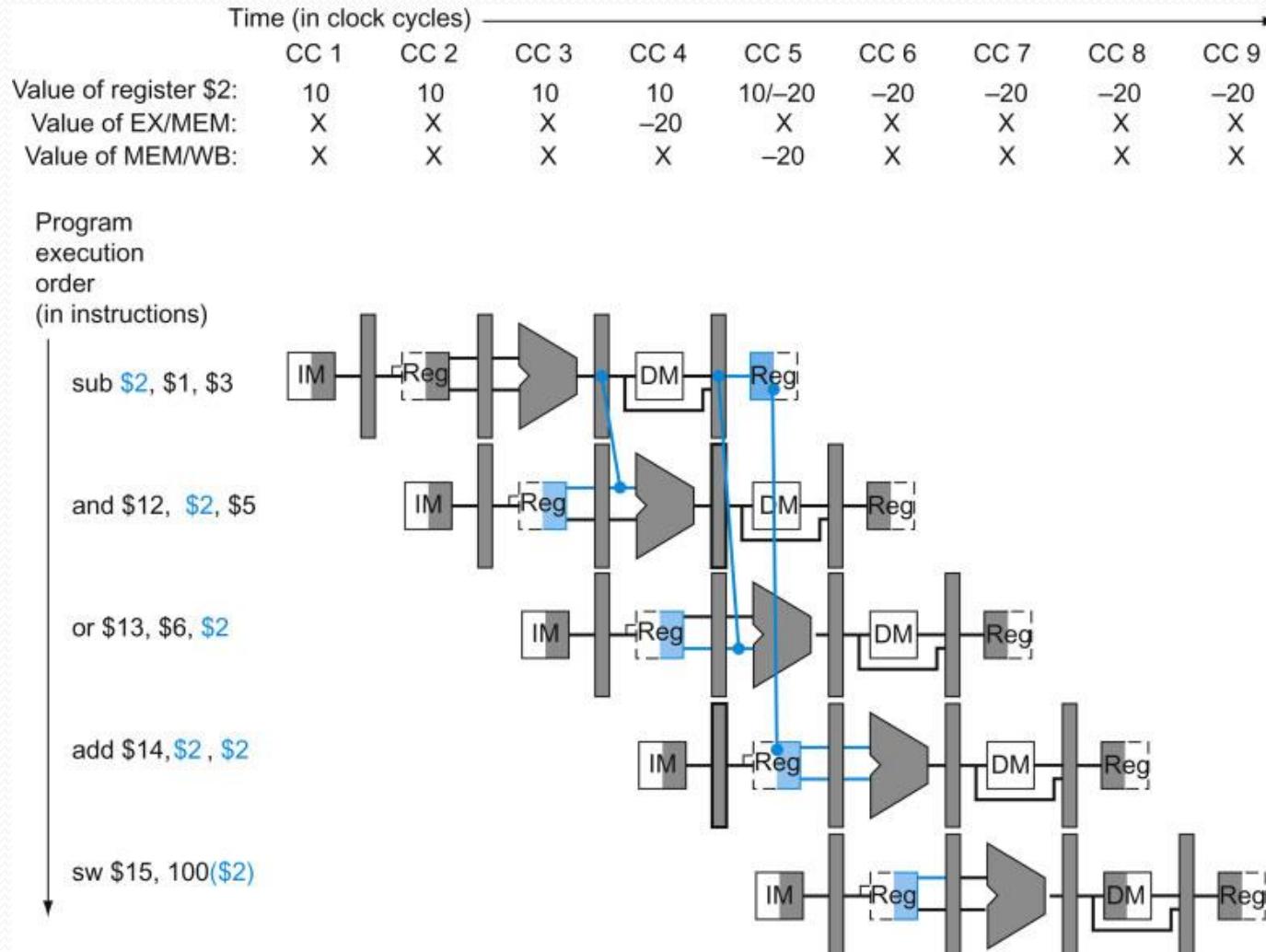


Hazard de Dados

- Como implementar a estratégia de *forwarding*?
 - As condições mostradas anteriormente estão parcialmente corretas.
 - O *forwarding* só deve acontecer se a instrução mais antiga (da qual as seguintes dependeriam) efetivamente escrever um valor em um registrador.
 - EX/MEM.RegWrite = 1, MEM/WB.RegWrite = 1.
 - Além disso, o registrador de destino não pode ser o \$0 (\$zero).
 - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0.

Hazard de Dados

- Como implementar a estratégia de *forwarding*?



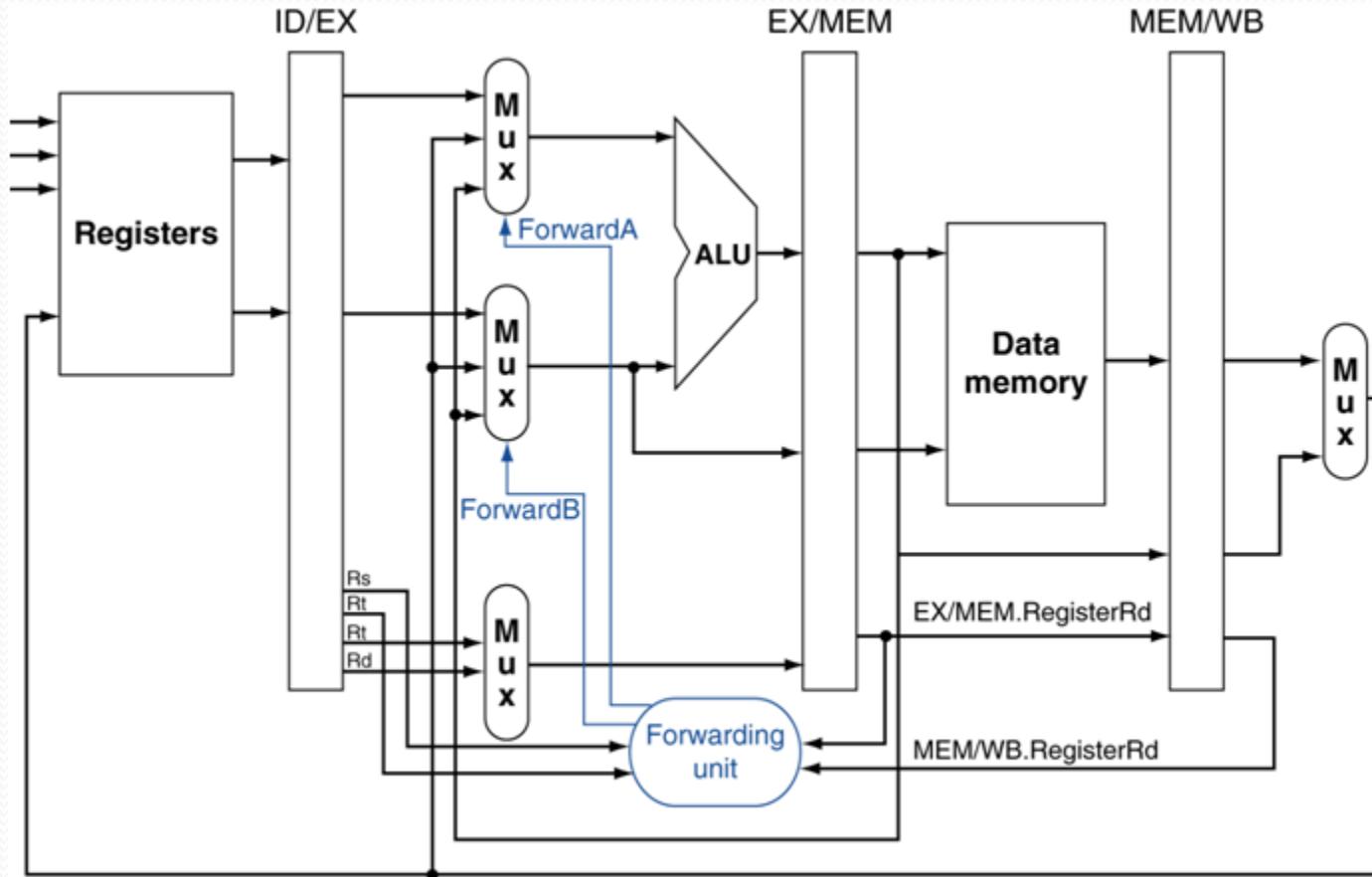
As dependências podem ser mostradas em função dos registradores da *pipeline*.

Este procedimento já indica os possíveis caminhos de *forwarding* para contornar os *hazards*.

Importante: se as entradas da ALU puderem vir de qualquer registrador de *pipeline*, e não somente do registrador ID/EX, então o *forwarding* acontecerá e a *pipeline* funcionará mesmo na presença destas dependências.

Hazard de Dados

- Como implementar a estratégia de *forwarding*?



Hazard de Dados

- Reescrevendo as condições:

- EX hazard

- 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

- MEM hazard

- 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

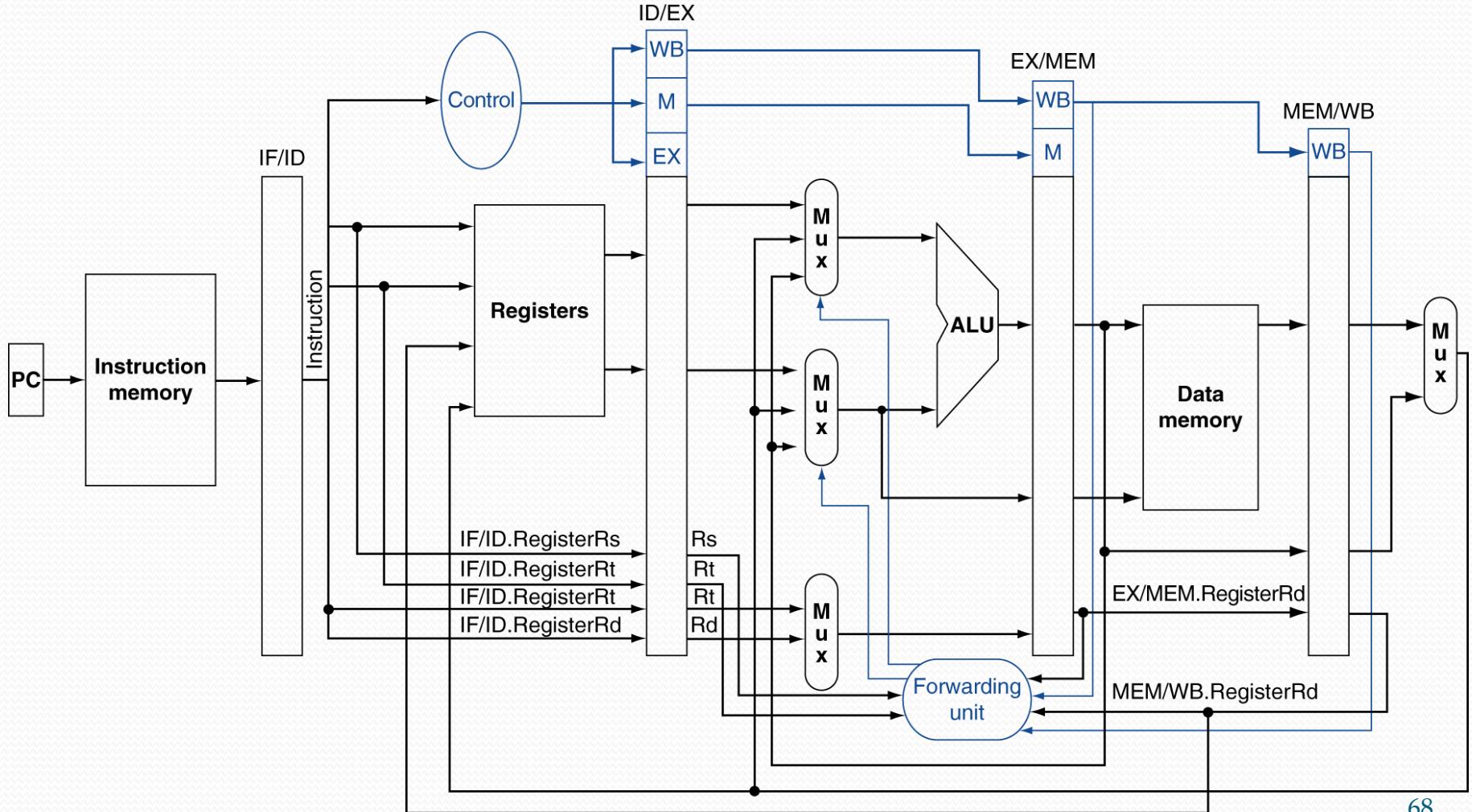
- Considere a seguinte sequência de instruções:
add \$1, \$1, \$2
add \$1, \$1, \$3
add \$1, \$1, \$4
- Neste caso, os dois tipos de *hazard* ocorrem – a opção é fazer *forwarding* a partir do estágio MEM (em vez do WB) uma vez que ele contém o resultado mais recente.
- É preciso revisar a condição do *hazard* MEM:
 - Apenas faz *forwarding* se a condição de *hazard* EX não for verdadeira.

Hazard de Dados

- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.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 not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

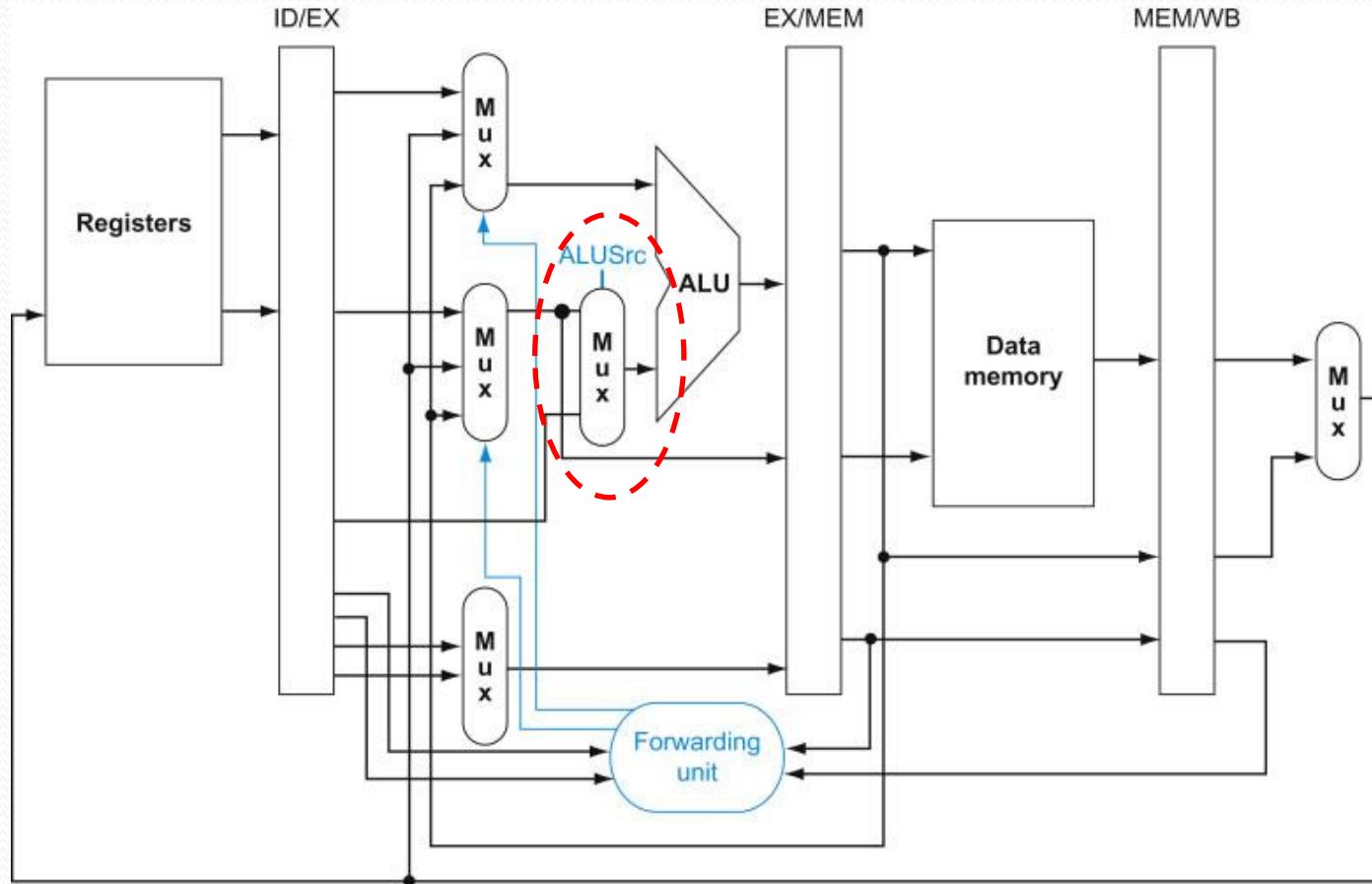
Hazard de Dados

- Datapath com forwarding:



Hazard de Dados

- Levando em conta o campo de imediato também:



Hazard de Dados

- **Para pensar:**

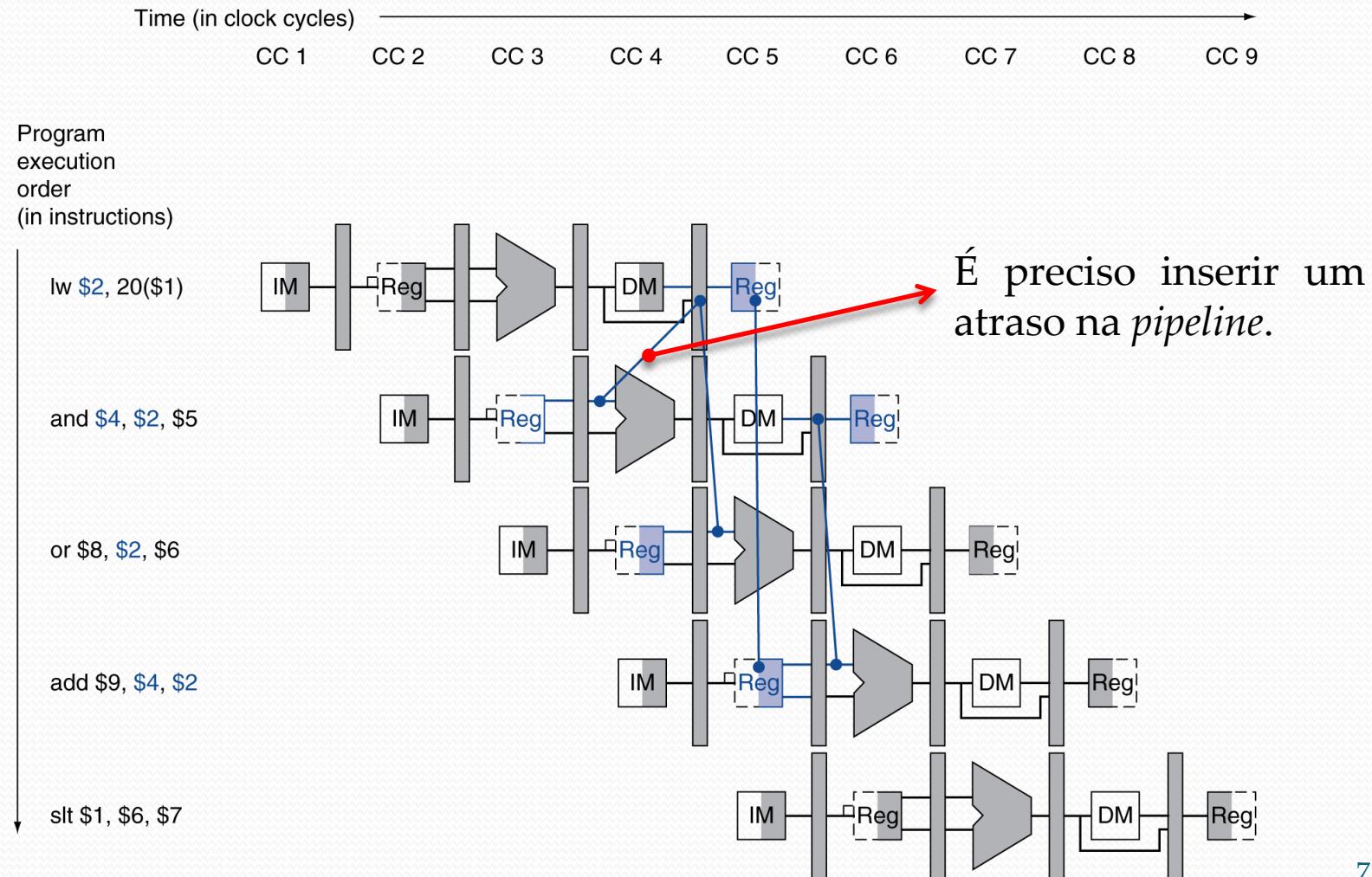
lw \$2, 0(\$3)

sw \$2,0(\$5)

- Neste caso, seria possível evitar um *stall* já que o valor de \$2, que será armazenado em seguida pela instrução *store*, pode ser recuperado de MEM/WB para seu uso no estágio MEM da instrução *store*.
- Para isto, teríamos que acrescentar uma unidade de *forwarding* no estágio de acesso à memória (MEM).

Hazard de Dados

- Load-use hazard:



Hazard de Dados

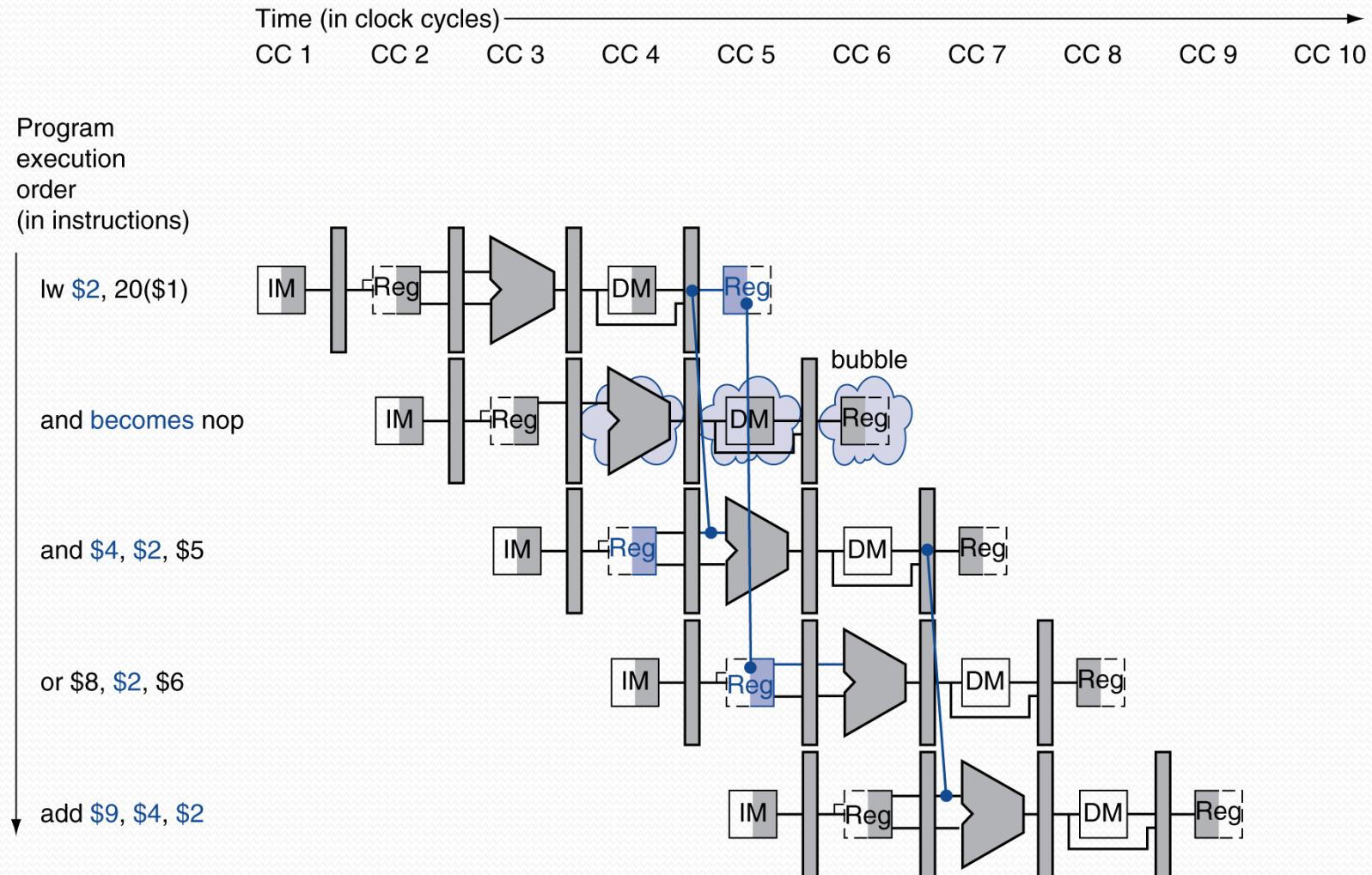
- Como identificar um *load-use hazard*?
 - **Momento:** no estágio ID da instrução posterior ao *load* é possível descobrir se há um *load-use hazard*. Ou, equivalentemente, quando a instrução *load* está no 3º estágio (EX).
 - **Condição:**
if (ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
insira um *stall* na *pipeline*

Hazard de Dados

- Como o *stall* é feito?
 - É preciso evitar que o conteúdo dos registradores PC e IF/ID seja modificado.
 - Para isto, basta forçar os sinais de controle contidos no registrador ID/EX para o valor 0.
 - Desta forma, os estágios EX, MEM e WB não terão qualquer efeito sobre a memória, o arquivo de registradores e o valor de PC.
 - Em essência, eles fazem um *nop* (*no-operation*).
 - Assim, a instrução em uso (i.e., a que foi lida após o *load*) será decodificada novamente e a instrução subsequente será buscada mais uma vez.
 - O atraso de 1 ciclo permite que o estágio MEM leia o dado referente à instrução *load*, o qual poderá ser passado via *forwarding* para a próxima instrução (que estará no estágio EX).

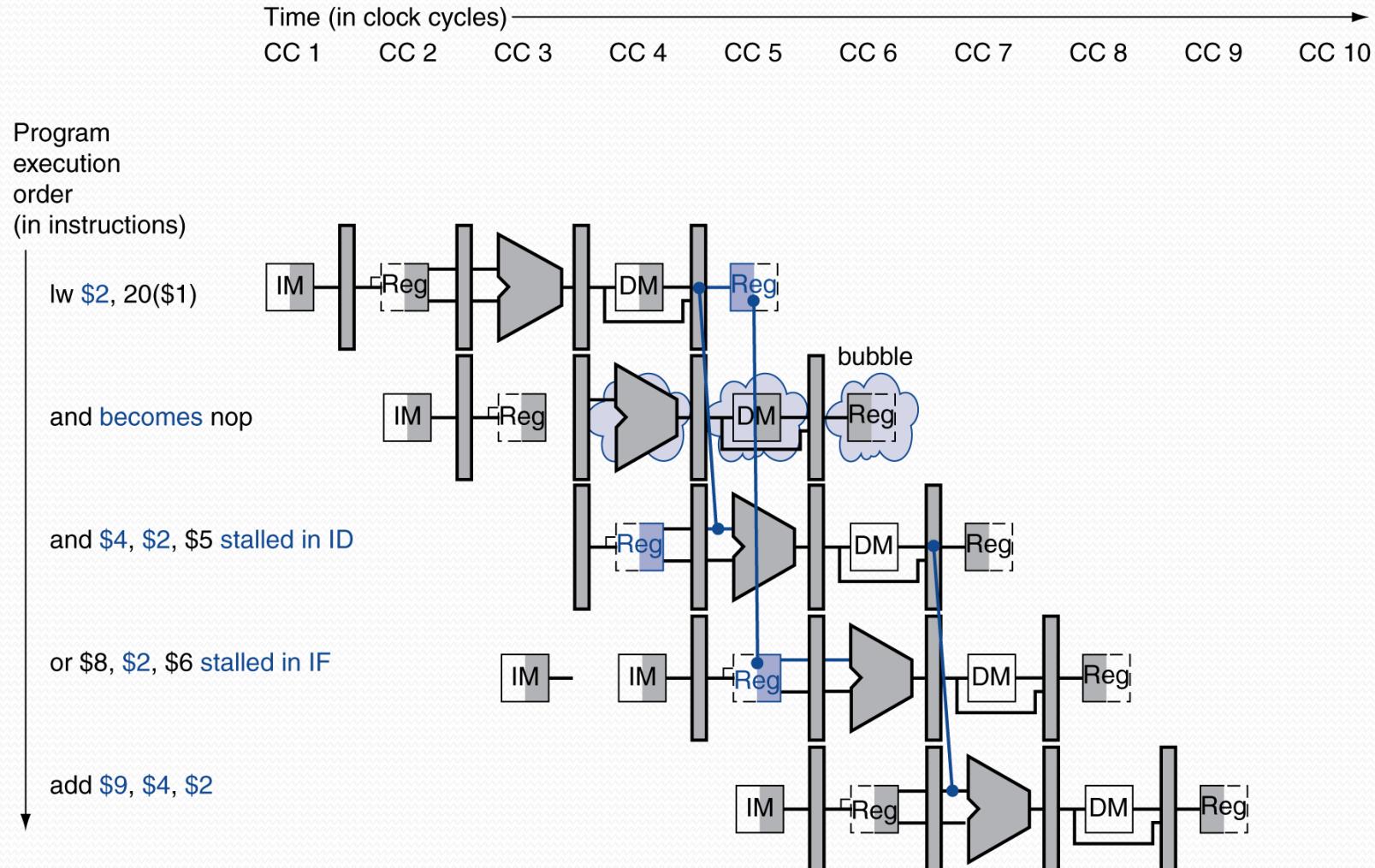
Hazard de Dados

- Como o *stall* é feito?



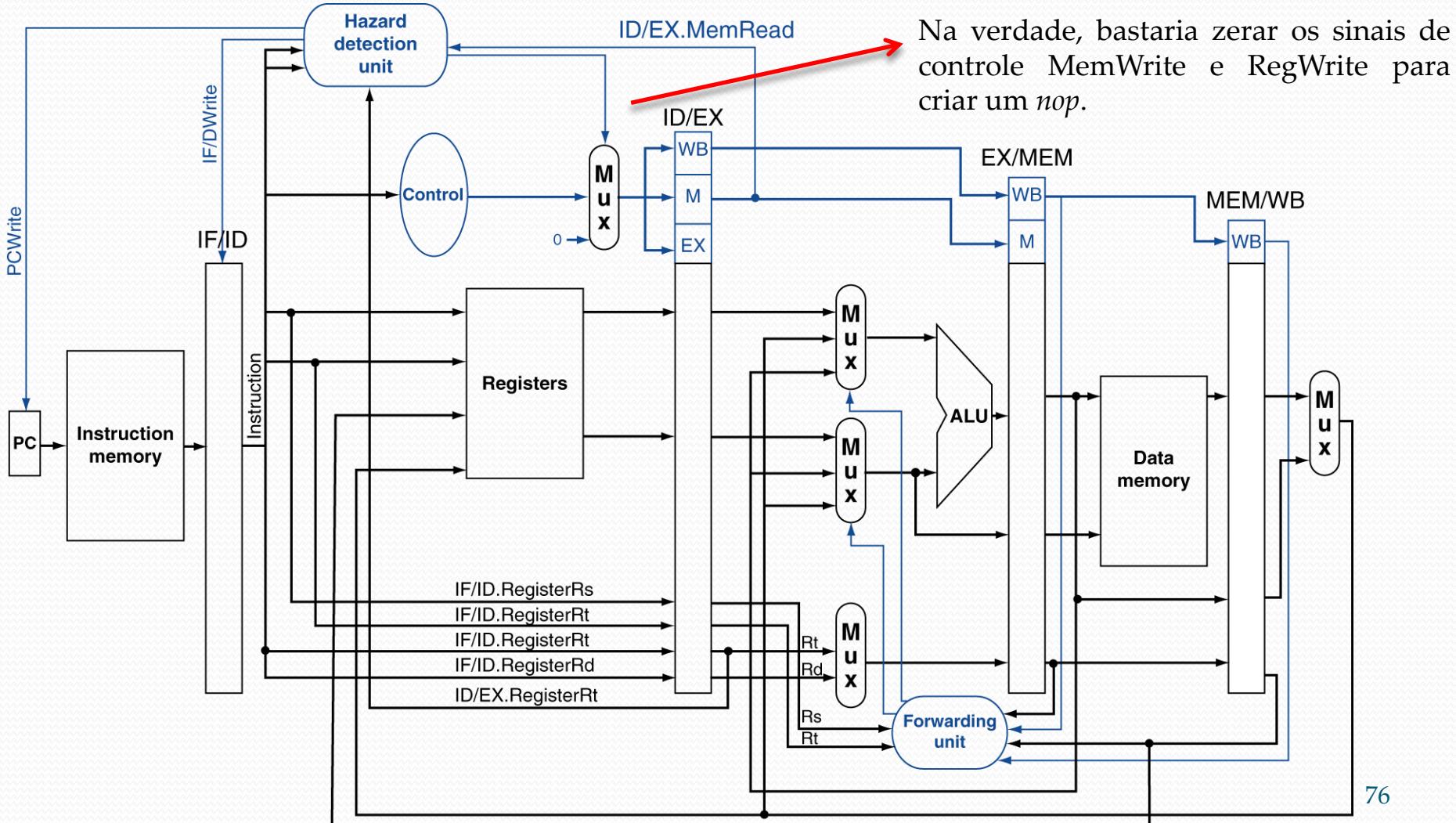
Hazard de Dados

- Sendo mais preciso,



Hazard de Dados

- Datapath com *forwarding* e detecção de *hazard*:



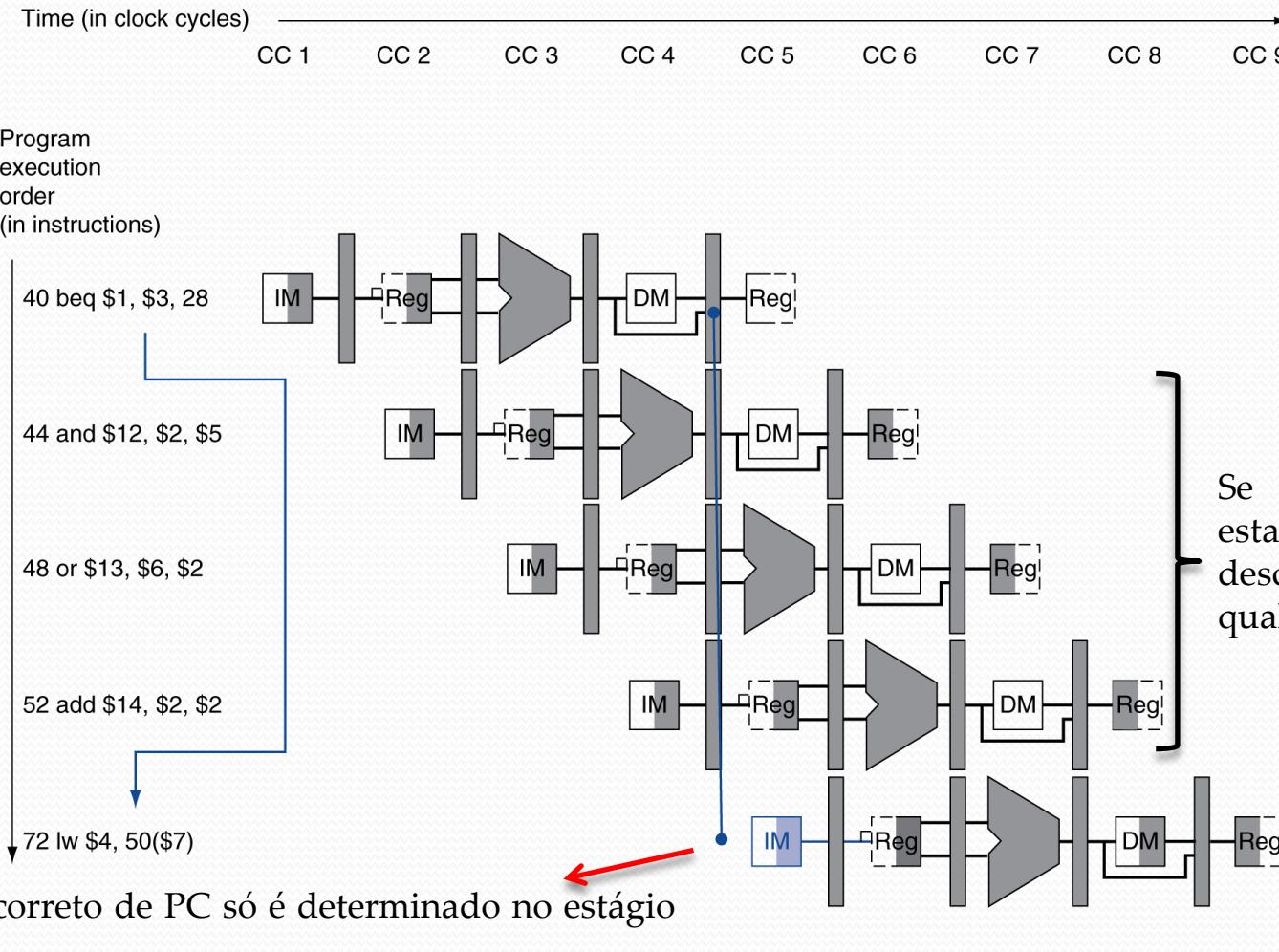
Hazard de Dados

- Discussão:

- *Stalls* reduzem o desempenho da *pipeline*.
 - Porém, são necessários para que se garanta o resultado correto.
- Compiladores podem rearranjar o código a fim de evitar a ocorrência de *hazards* e *stalls*.
 - Isto requer um conhecimento a respeito da estrutura da *pipeline*.

Hazard de Controle

- Exemplo:



Hazard de Controle

- Como já comentamos anteriormente, uma solução simples seria sempre imaginar que o desvio não será tomado.
- Se ele for tomado, as instruções sendo buscadas e decodificadas precisam ser descartadas.
- *Flush*: limpar a *pipeline* – é preciso zerar os sinais de controle para todas as instruções (nos estágios IF, ID e EX) quando o *branch* atinge o estágio MEM.

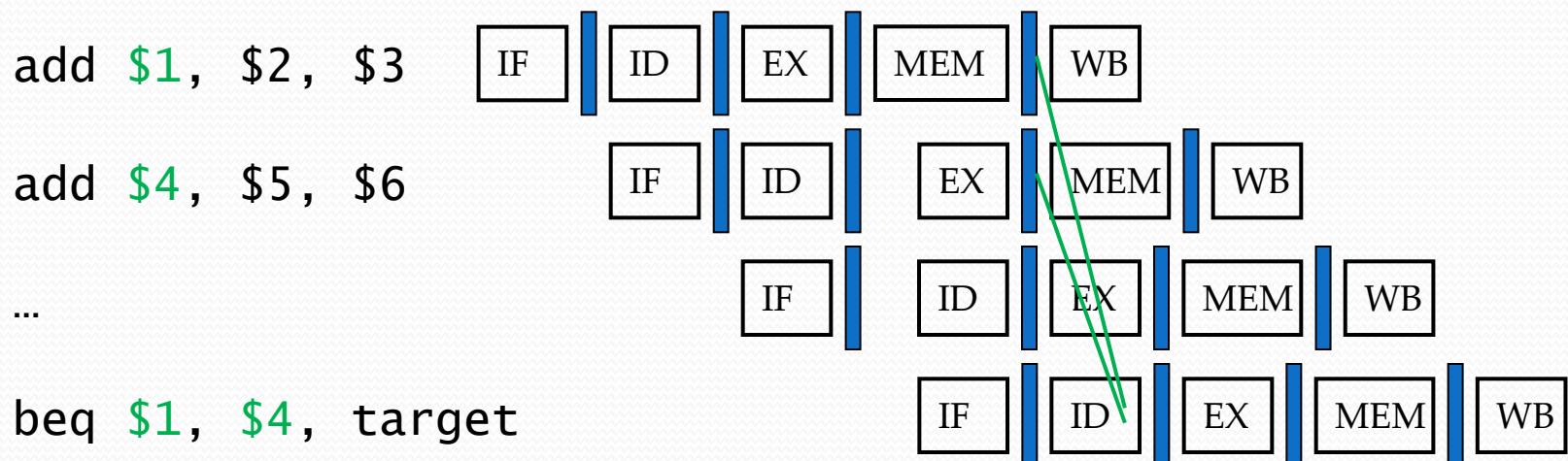
Hazard de Controle

- É possível reduzir o custo de um *branch* quando ele é tomado:
 - Somador extra é inserido no estágio ID para calcular o endereço alvo do desvio.
 - Comparador simples (para *beq*, por exemplo) para testar a condição assim que os valores dos registradores são lidos.
 - **Atenção:** o fato de a comparação ser feita no estágio ID abre a possibilidade para a ocorrência de *hazards* de dados.

Hazard de Controle

- Cuidados:

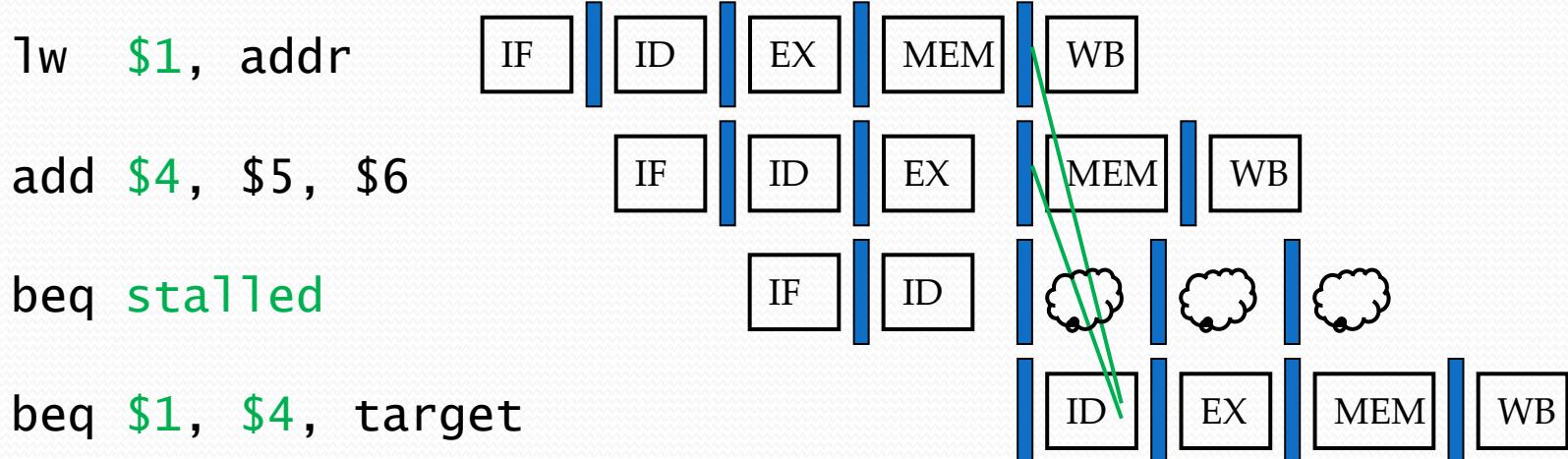
➤ Neste caso, *forwarding* pode resolver os *hazards*.



Hazard de Controle

- Cuidados:

➤ Neste caso, é necessário introduzir 1 ciclo de atraso.

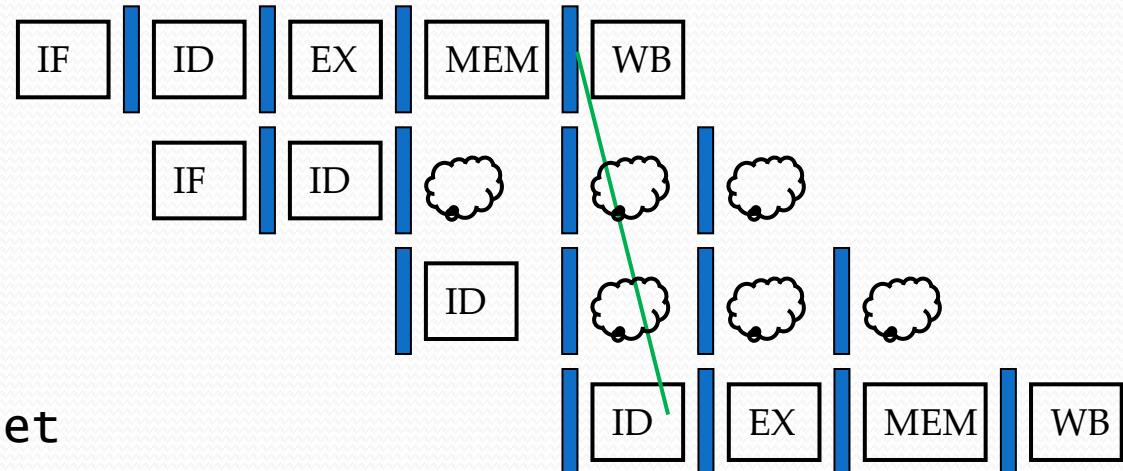


Hazard de Controle

- Cuidados:

➤ Neste caso, é necessário introduzir 2 ciclos de atraso.

lw \$1, addr



beq stalled

beq stalled

beq \$1, \$0, target

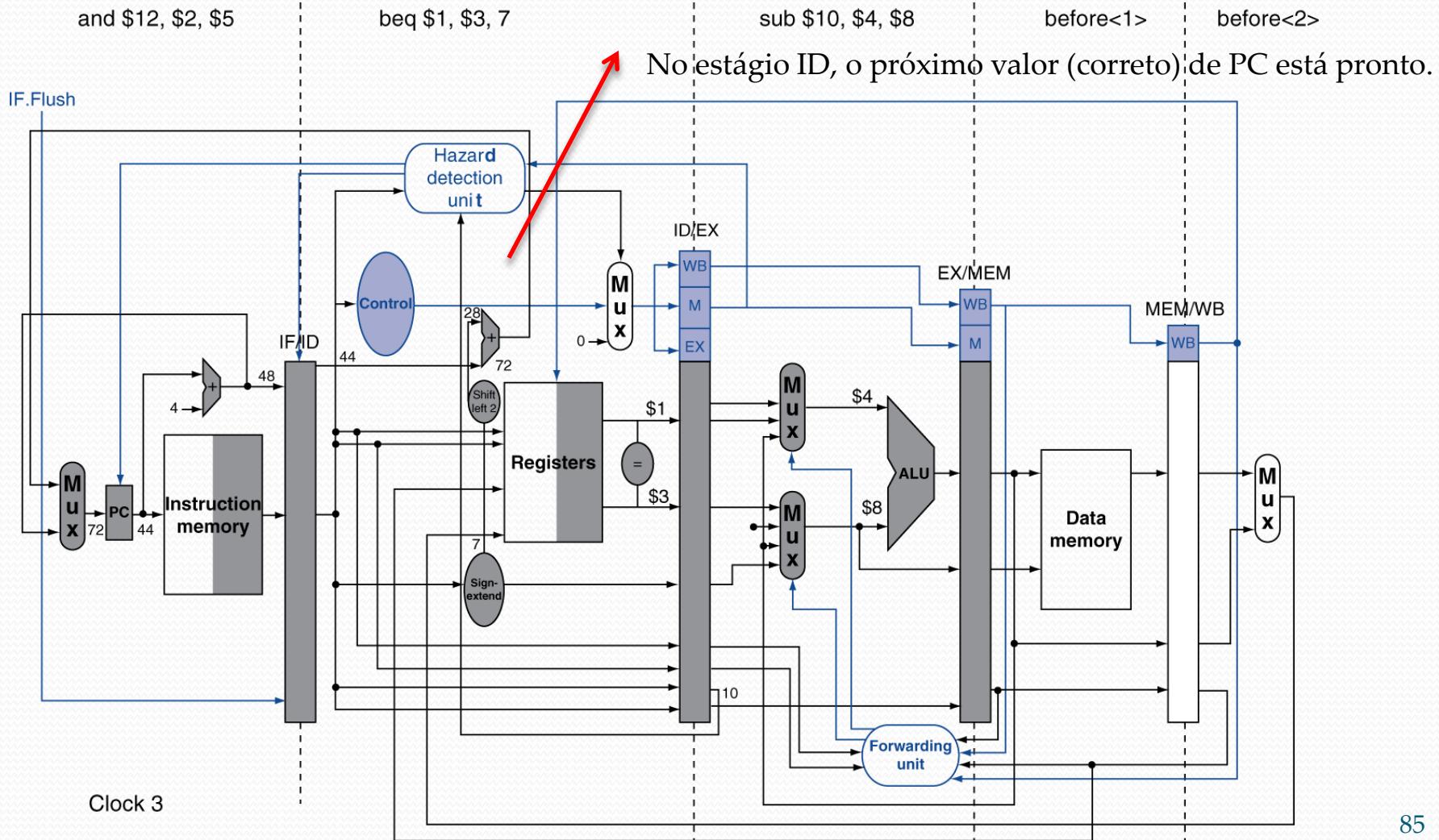
Hazard de Controle

- **Exemplo:** desvio tomado

```
36: sub $10, $4, $8  
40: beq $1, $3, 7  
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)
```

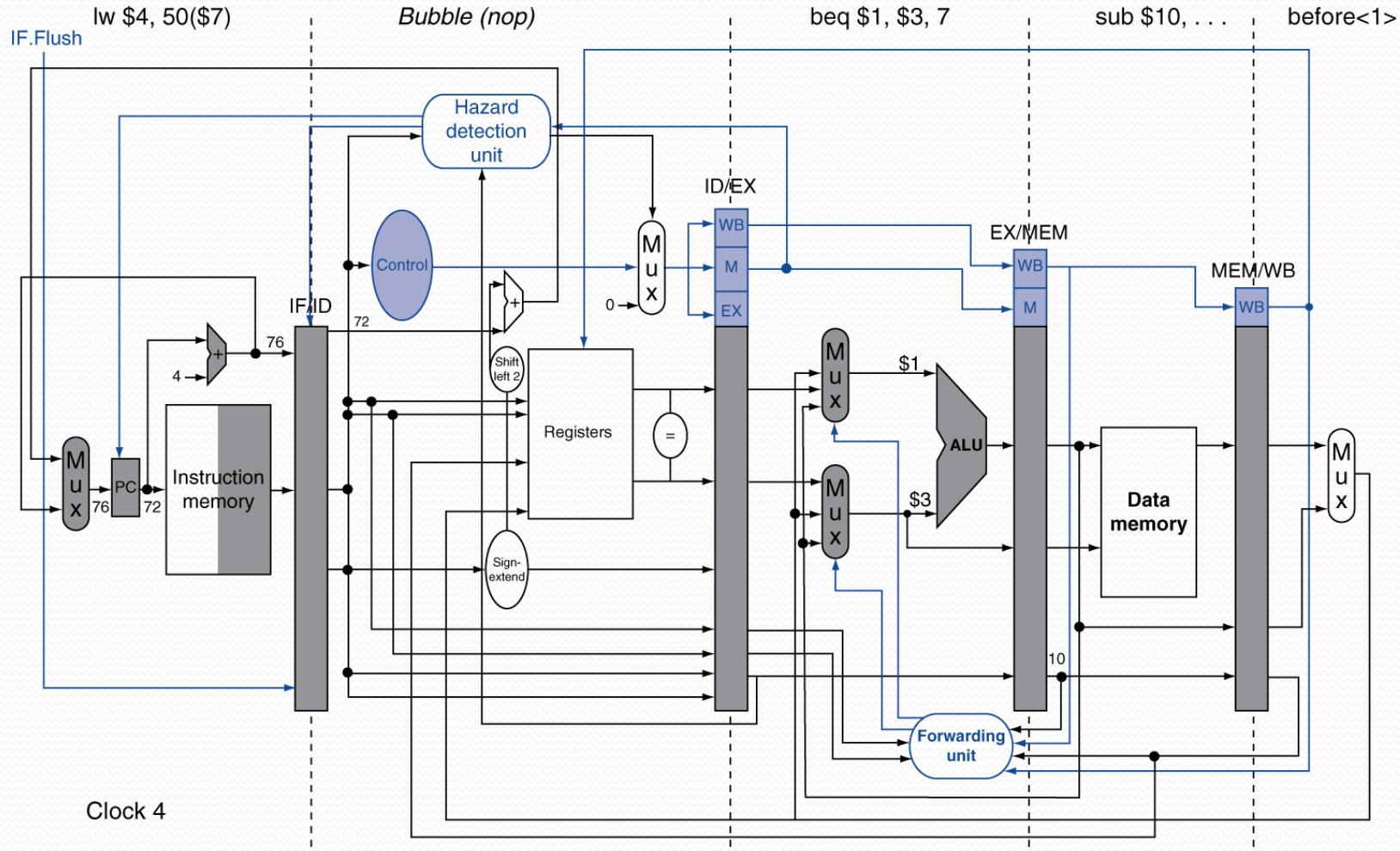
Hazard de Controle

- Exemplo: desvio tomado



Hazard de Controle

- Exemplo: desvio tomado



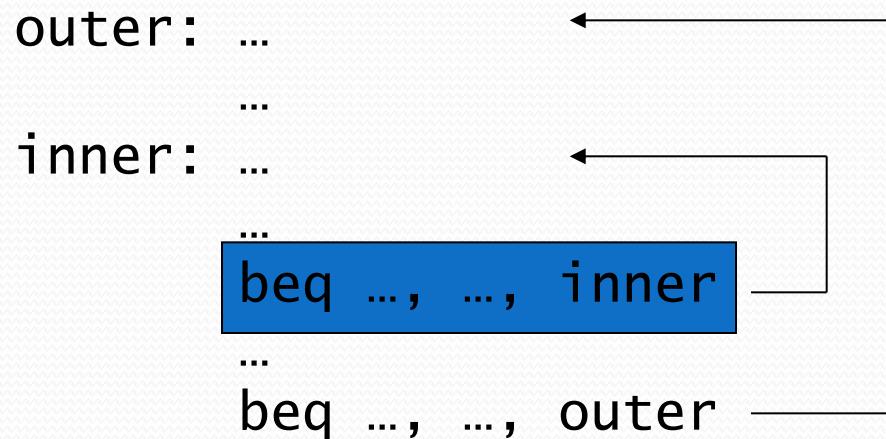
Hazard de Controle

- **Previsão dinâmica:**

- Tentativa de prever o comportamento de um desvio durante a execução do programa.
- **Possibilidade:** buffer de previsão (tabela de histórico de desvio)
 - Indexada pelos bits menos significativos do endereço da instrução de desvio.
 - Contém 1 bit que informa se o *branch* foi recentemente tomado ou não.
 - **Desvantagem:** mesmo se um desvio quase sempre for tomado, é possível cometer dois erros de previsão, em vez de apenas um.

Hazard de Controle

- Previsão dinâmica:

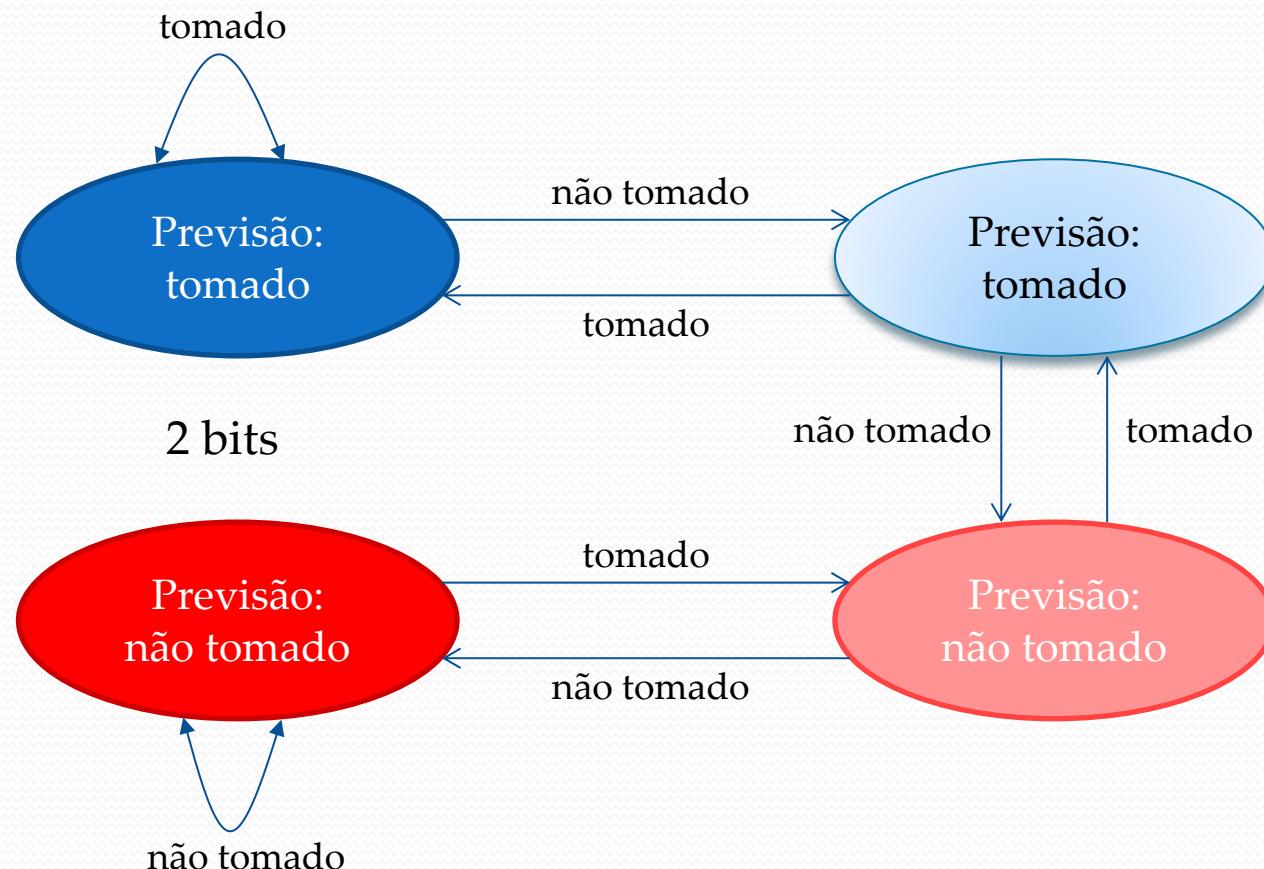


- Primeiro erro ocorre quando a previsão é de que ele será tomado na última iteração do loop interno.
- Segundo erro ocorre quando retornamos ao loop interno e prevemos que ele não será tomado.

Hazard de Controle

- **Previsão dinâmica:**

- Só altera a previsão após dois erros consecutivos.

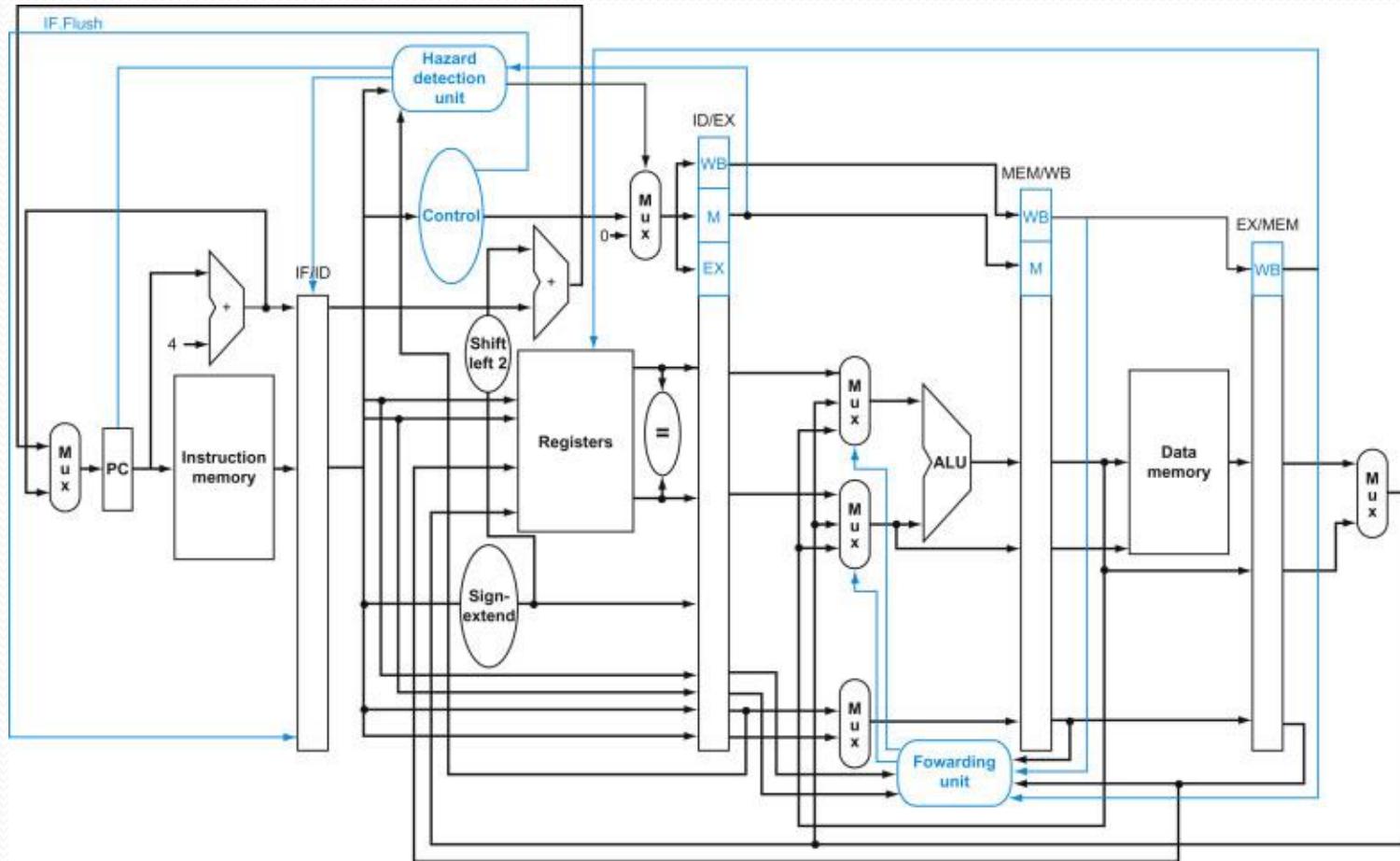


Hazard de Controle

- **Exercício:**

- Considere três esquemas de previsão de desvio: não tomado, tomado e previsão dinâmica.
 - A taxa de acerto da previsão dinâmica é de 90%.
 - Quando a previsão está correta, não há penalidades.
 - Quando há um erro, dois ciclos são desperdiçados como penalidade.
- Qual preditor representa a melhor opção para os seguintes casos?
 - a) Desvio que é tomado com 5% de frequência;
 - b) Desvio que é tomado com 95% de frequência;
 - c) Desvio que é tomado com 70% de frequência.

Resumo



Exceções

- Um dos aspectos mais desafiadores no projeto de um processador envolve o tratamento de eventos excepcionais que alteram o fluxo normal de execução de instruções.
- Os chamados mecanismos de tratamento de exceções (ou interrupções) foram criados inicialmente para lidar com eventos inesperados que fossem internos ao processador, como a ocorrência de um *overflow*. Posteriormente, foram estendidos para a comunicação de dispositivos de entrada/saída com o processador.
- **Importância:** a tarefa de detectar uma condição atípica e tomar uma ação apropriada comumente está relacionada ao caminho crítico de tempo de um processador.

Exceções

- **Arquitetura MIPS:**

- A implementação simplificada que temos construído possibilita a ocorrência de duas exceções: (1) execução de uma instrução indefinida e (2) *overflow*.
- **Ação básica:**
 - Salvar o endereço da instrução que causou a exceção no registrador EPC (*exception program counter*);
 - Transferir o controle para o sistema operacional em algum endereço específico. Então, o sistema operacional pode tomar uma ação apropriada, como, por exemplo, interromper a execução do programa e reportar um erro.
 - Para isto, ele precisa conhecer o motivo da exceção.

Exceções

- Duas possibilidades:

- Incluir um registrador de *status* (chamado de registrador de causa, *Cause*), que contém um campo para indicar o motivo ou tipo de exceção.
 - Opção empregada na arquitetura MIPS.
- Utilizar interrupção vetorizada – o endereço para o qual o controle é transferido é determinado pela causa da exceção.
 - Exemplo: Instrução indefinida $8000\ 0000_{hex}$
Overflow $8000\ 0180_{hex}$

Exceções

- **Proposta:**

- Para realizar o tratamento de exceções, vamos acrescentar alguns registradores e sinais de controle à implementação atual, bem como expandir a lógica de controle.
- Endereço: 8000 0180_{hex}
- Registradores adicionais:
 - EPC: um registrador de 32 bits usado para guardar o endereço da instrução que causou a exceção.
 - *Cause*: um registrador de 32 bits usado para especificar a causa da exceção. Vamos considerar que um campo de 5 bits codifica as duas possíveis fontes de exceção tratadas: 10 – instrução indefinida; 12 – estouro aritmético.

Exceções

- Exceções e *pipeline*:

- Da perspectiva da *pipeline*, uma exceção representa uma outra forma de *hazard* de controle.
- Suponha que um estouro aritmético tenha ocorrido durante a execução de uma instrução *add \$1, \$2, \$1*.
- De modo semelhante ao que ocorre com um desvio, devemos limpar (*flush*) as instruções que se seguem ao *add* e começar a buscar as instruções a partir do novo endereço.

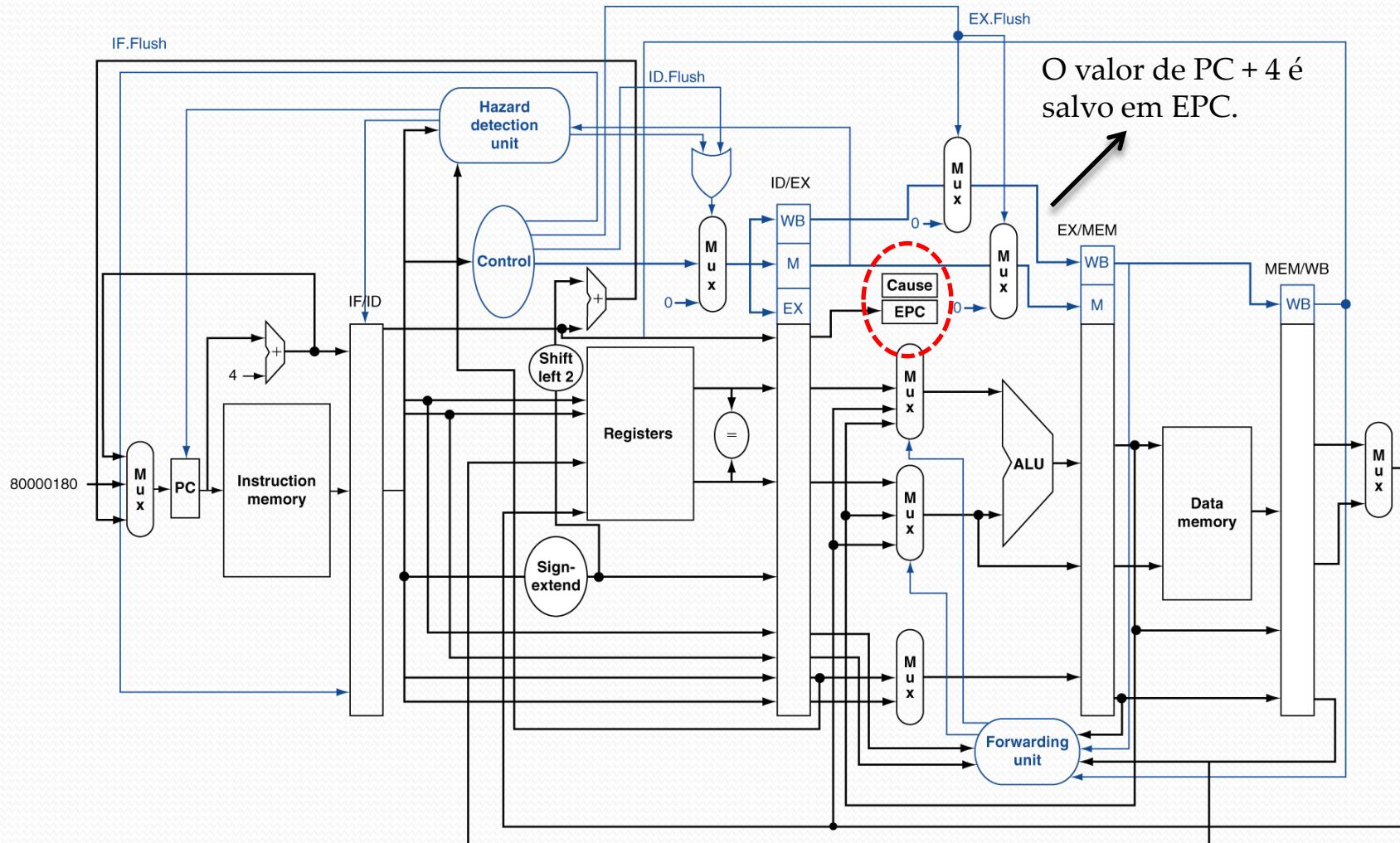
Exceções

- **Exceções e *pipeline*:**

- Para limpar instruções no estágio ID, usamos o multiplexador já presente no estágio ID a fim de zerar os sinais de controle, criando, assim, stalls.
 - Um novo sinal, chamado ID.Flush, é colocado em um OR com o sinal gerado pela unidade de detecção de *hazard*.
- Para limpar instruções no estágio EX, usamos um novo sinal EX.Flush para forçar zeros nos sinais de controle a partir de novos multiplexadores.
 - Isto evita a consumação da operação *add*. Ou seja, ela nunca chega a escrever o resultado da soma (com *overflow*) no registrador de destino (\$1).
- Para iniciar a busca a partir do endereço 8000 0180_{hex}, incluímos este valor como entrada adicional do multiplexador que determina o próximo endereço guardado em PC.

Exceções

- Exceções e *pipeline*:



Exceções

- Exemplo:

- Exceção na instrução **add**:

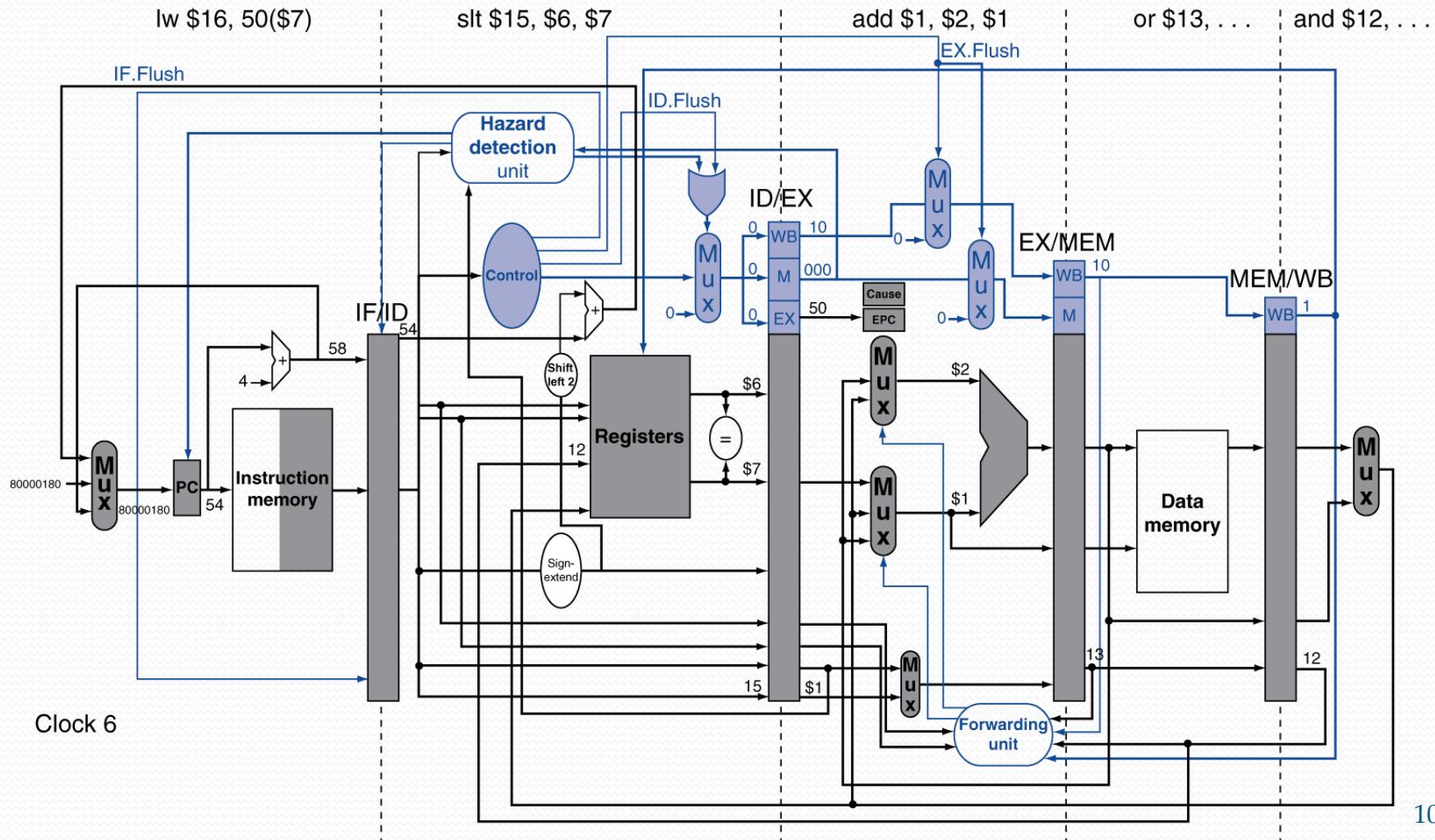
40 _{hex}	sub \$11, \$2, \$4
44 _{hex}	and \$12, \$2, \$5
48 _{hex}	or \$13, \$2, \$6
4C _{hex}	add \$1, \$2, \$1
50 _{hex}	slt \$15, \$6, \$7
54 _{hex}	lw \$16, 50(\$7)
...	

- Handler:

80000180 _{hex}	sw \$25, 1000(\$0)
80000184 _{hex}	sw \$26, 1004(\$0)

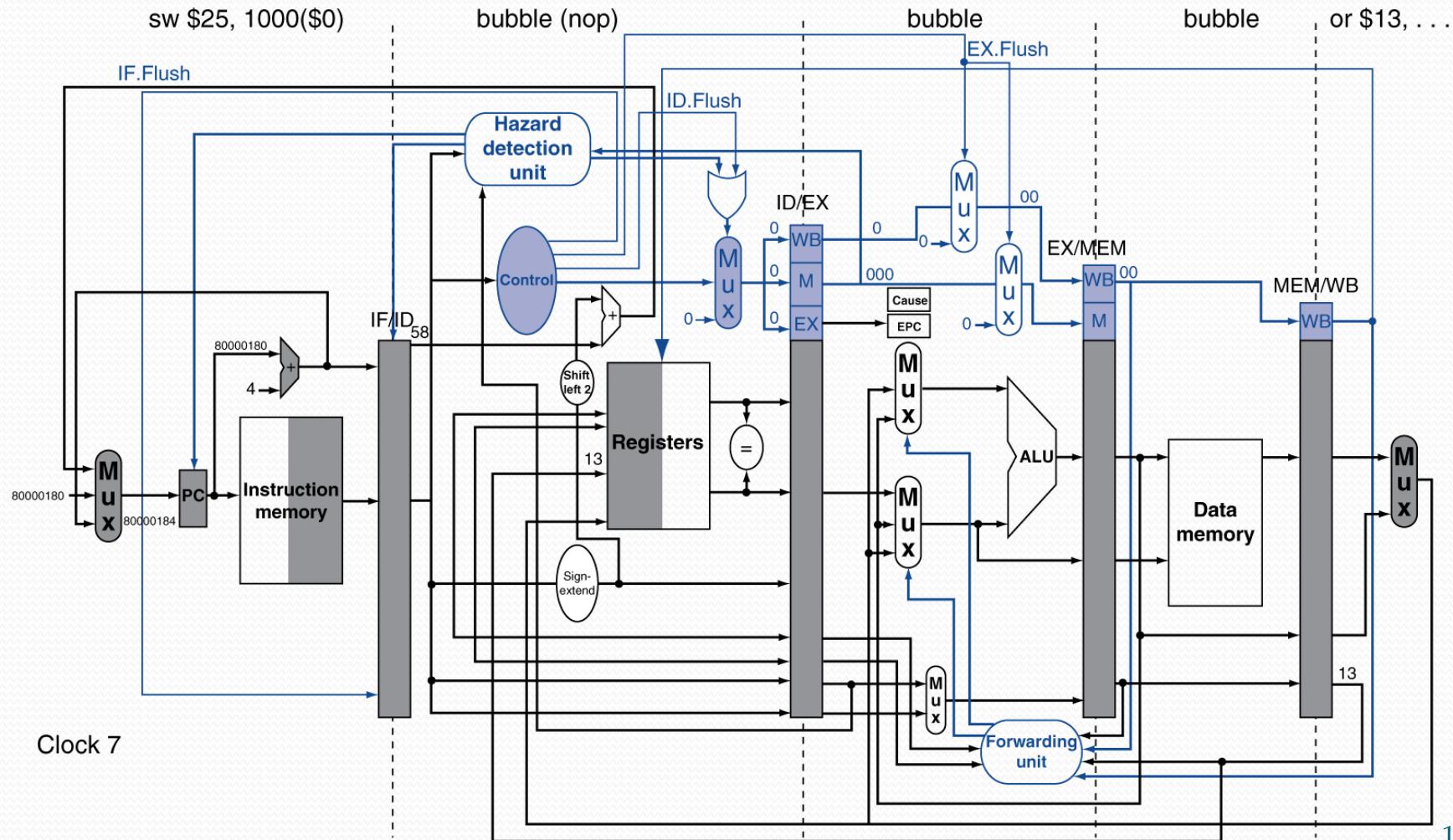
Exceções

- Exemplo:



Exceções

- Exemplo:



Exceções

- **Múltiplas exceções:**

- Com cinco instruções ativas em qualquer ciclo de relógio, é possível que múltiplas exceções sejam geradas no mesmo ciclo.
- Uma solução simples consiste em priorizar as exceções, dando preferência ao tratamento de alguns casos. No MIPS, a exceção associada à instrução mais antiga na *pipeline* é tratada, e as demais instruções são descartadas.
- **Exceções imprecisas:** o processador salva o endereço (atual) do PC, que pode não ser precisamente o endereço da instrução na *pipeline* que efetivamente deu origem à exceção. Caberá ao sistema operacional descobrir qual instrução causou o problema.
- Em *pipelines* mais complexas (múltiplos *issues*, término fora de ordem), manter as exceções na forma precisa é difícil.

Créditos

- Figuras extraídas de D. A. Patterson e J. L. Hennessy, “*Computer Organization and Design: The Hardware/Software Interface*”, Morgan Kauffman, 5^a edição, 2013.

http://textbooks.elsevier.com/web/product_details.aspx?isbn=9780124077263