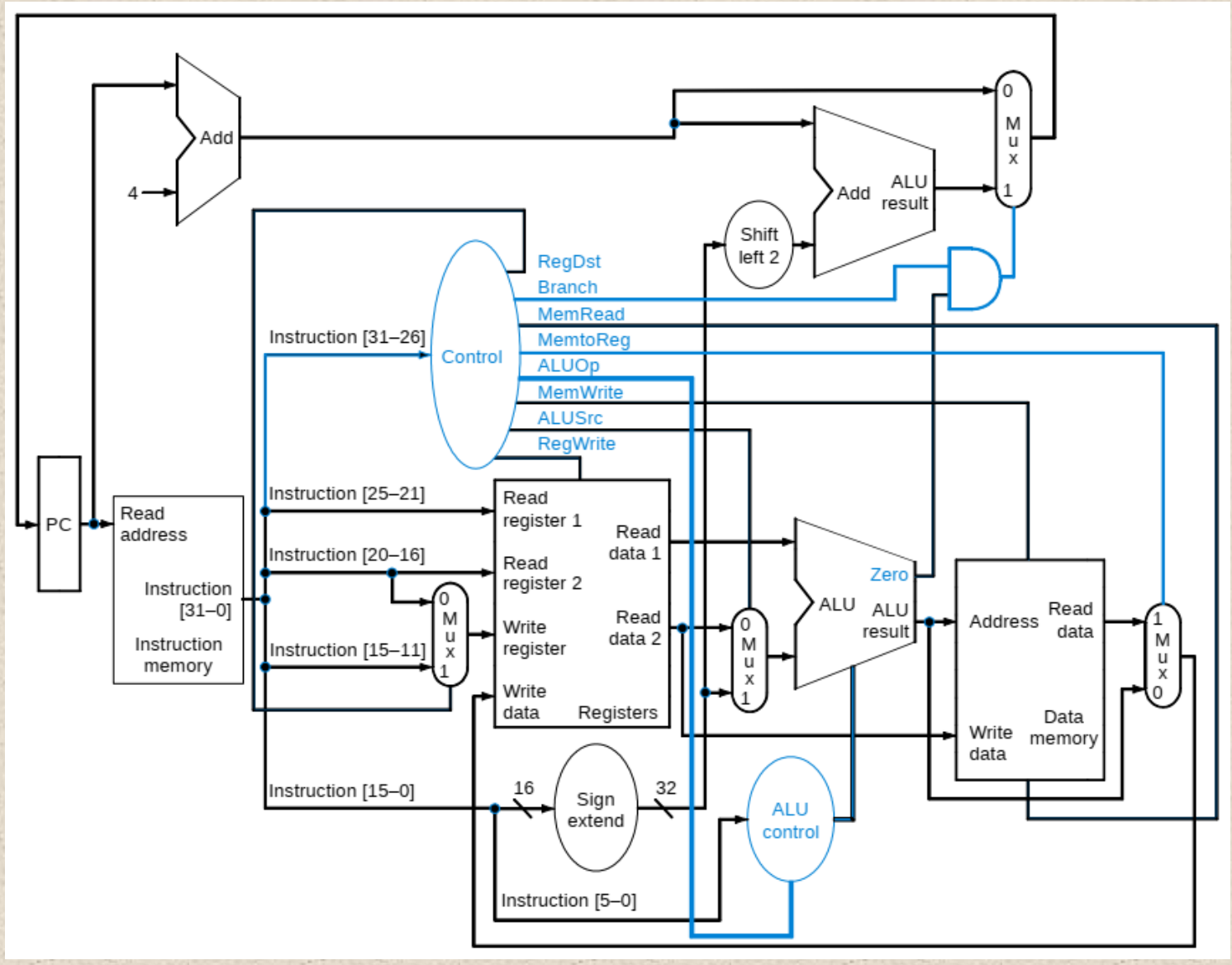
Pipeline MIPS

Revisão - Monociclo



Load é lento

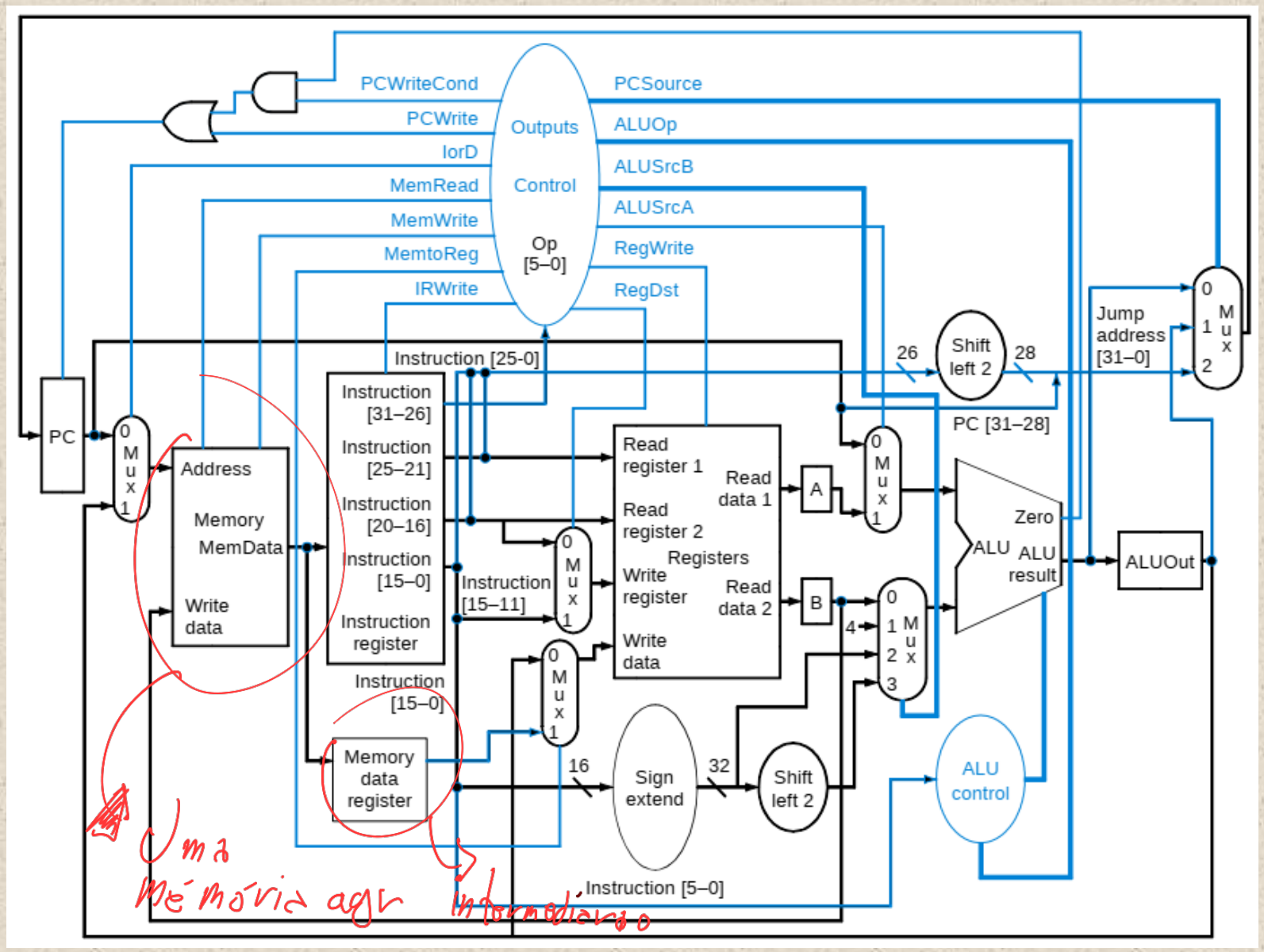
Revisão - Multiciclo

1. Busca de instruções na memória para registrador intermediário;
2. Decodificação de instruções e busca de registradores;
3. Execução, cálculo de endereço de memória ou desvio;
4. Acesso à memória ou conclusão de instruções do tipo R;
5. Etapa de *write-back.*

INSTRUÇÕES LEVAM DE 3 - 5 CICLOS!

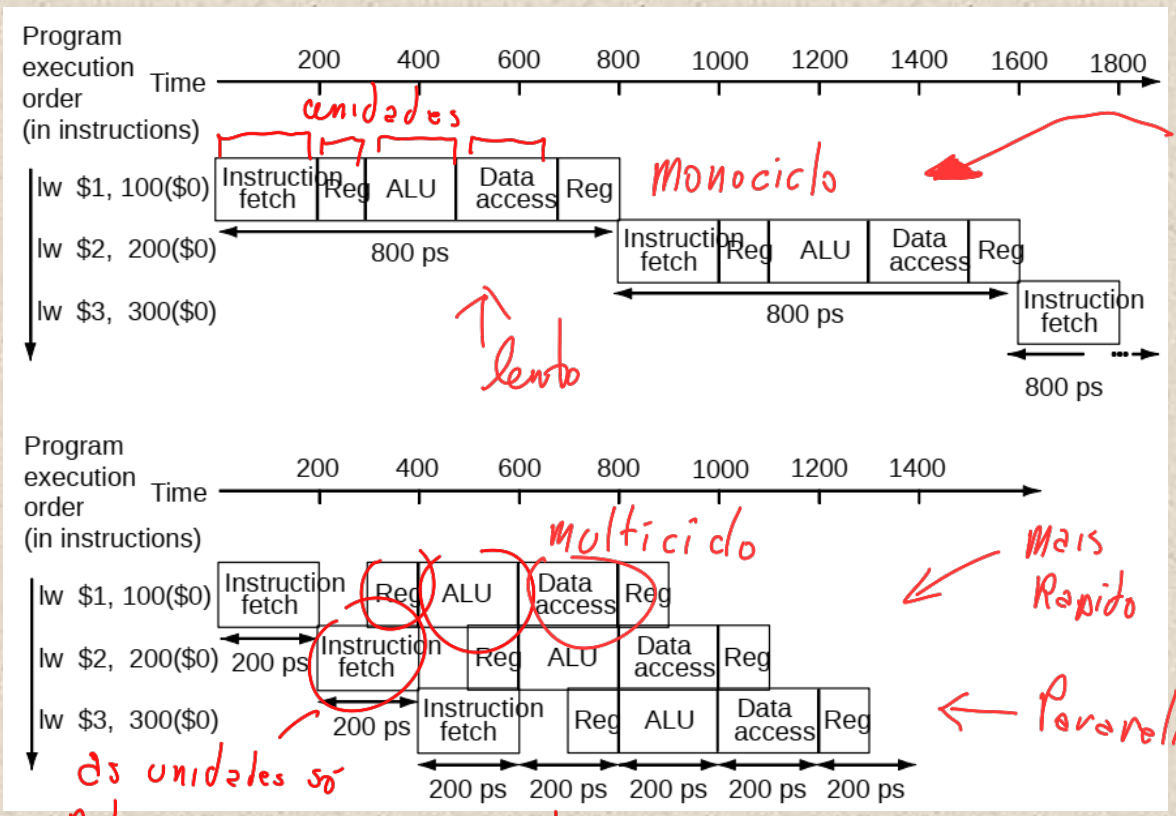
Revisão - Multiciclo

Mais enxuta



Pipelining

- Como melhorar o desempenho do processador: mais instruções em menos tempo



Cada estágio sem todas unidades sendo usadas (partes da CPU) sem uso

Nota: suposições de tempo alteradas para este exemplo

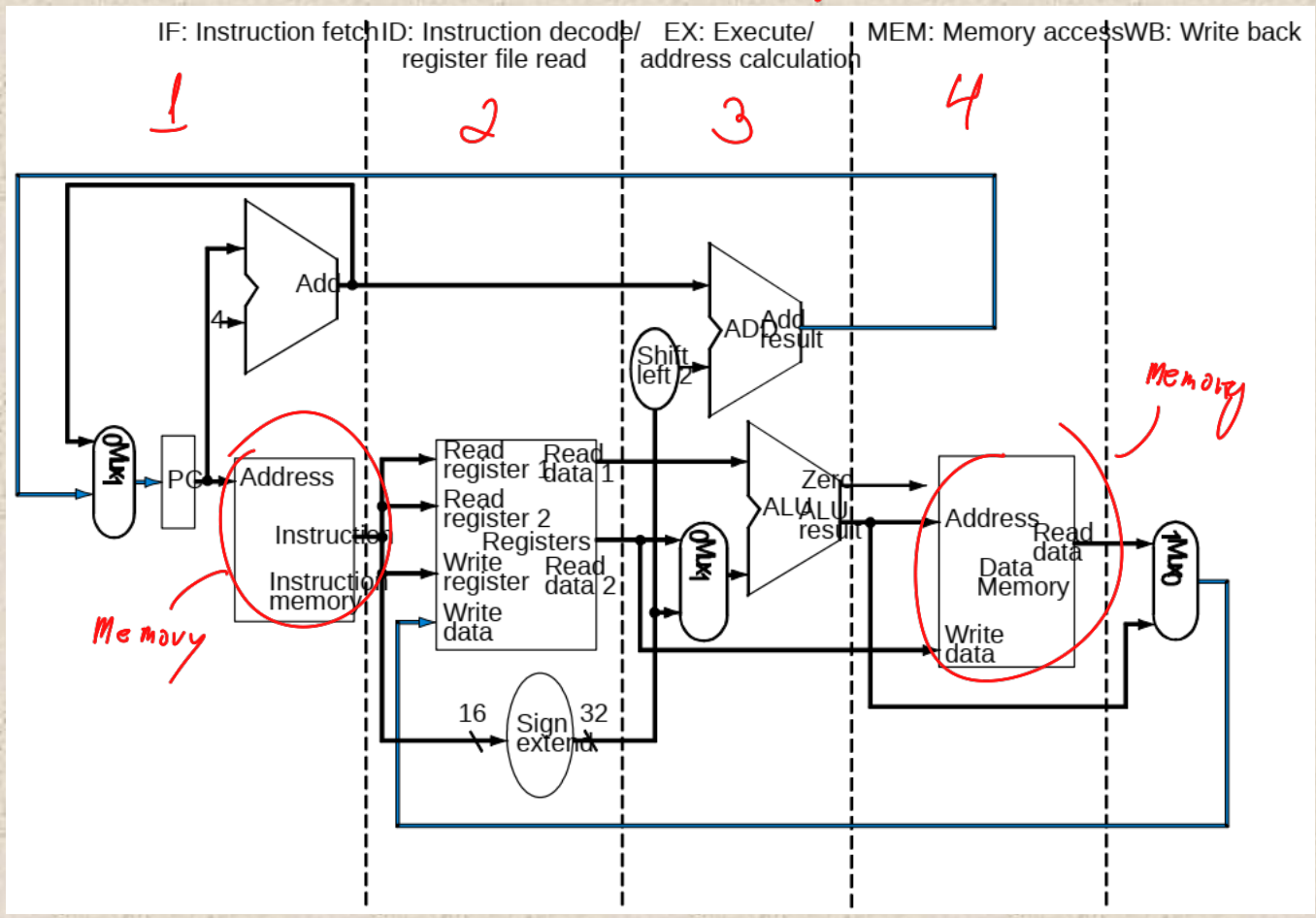
- O ganho de tempo ideal é o número de estágios no pipeline (5 estágios, 5 vezes menos tempo na execução). Conseguimos isso?

Pipelining

- Pontos positivos
  - Todas as instruções têm o mesmo tamanho (um único acesso à memória para busca e escrita);
  - Apenas alguns formatos de instrução;
  - Operandos de memória aparecem apenas em *load* e *store*.
- Pontos negativos (dificuldades não só do MIPS)
  - Conflitos estruturais: suponha que tivéssemos apenas uma memória;
  - Conflitos de controle: instruções de desvio;
  - Conflitos de dados: uma instrução depende de uma instrução anterior.
- Construiremos um *pipeline* simples e analisaremos esses conflitos;
- Falaremos sobre processadores modernos e suas dificuldades:
  - Manipulação de exceção;
  - Tentar melhorar o desempenho com execução fora de ordem (superescalar), etc.

Ideia básica

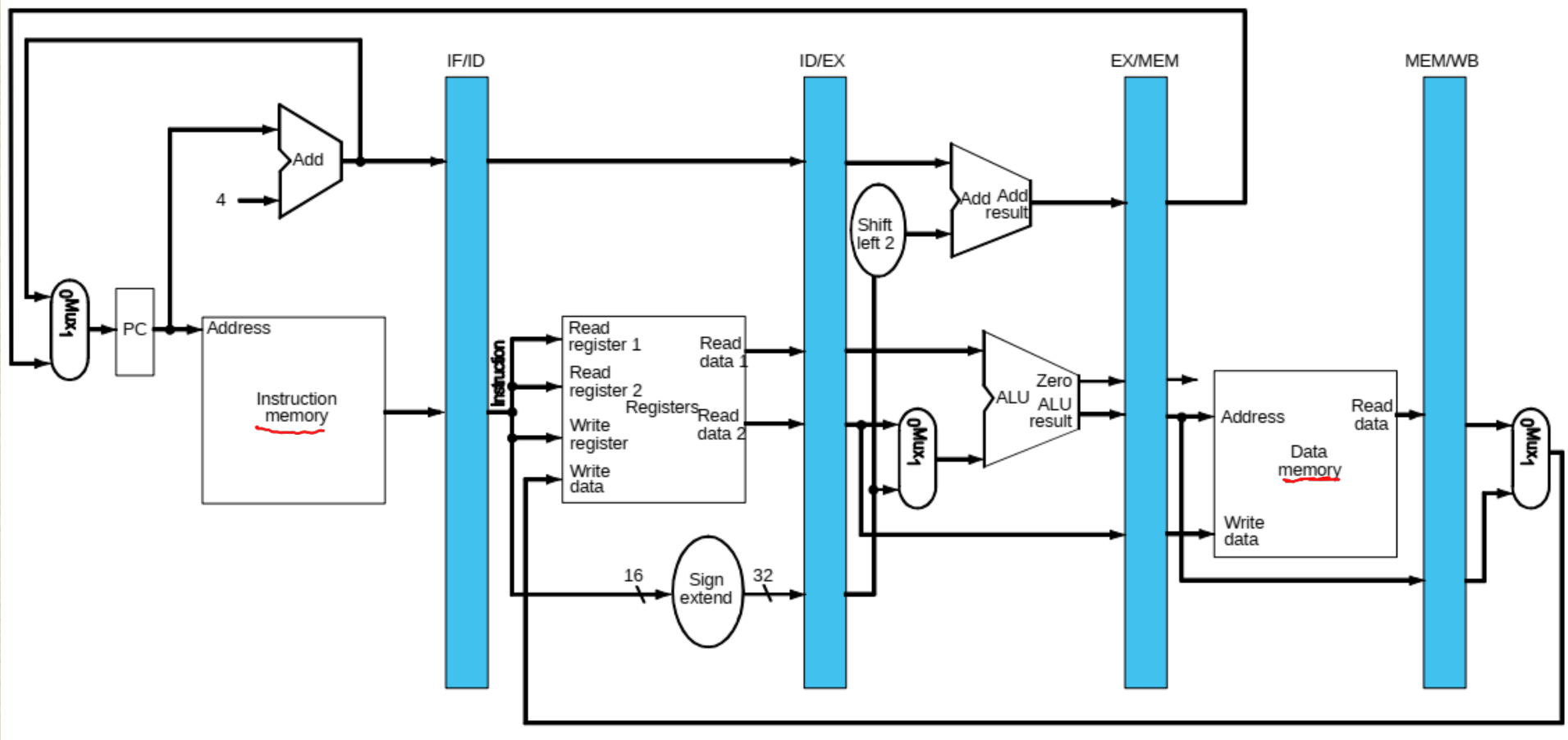
Mistura de Mono e Multi ciclo



De novo & memórias

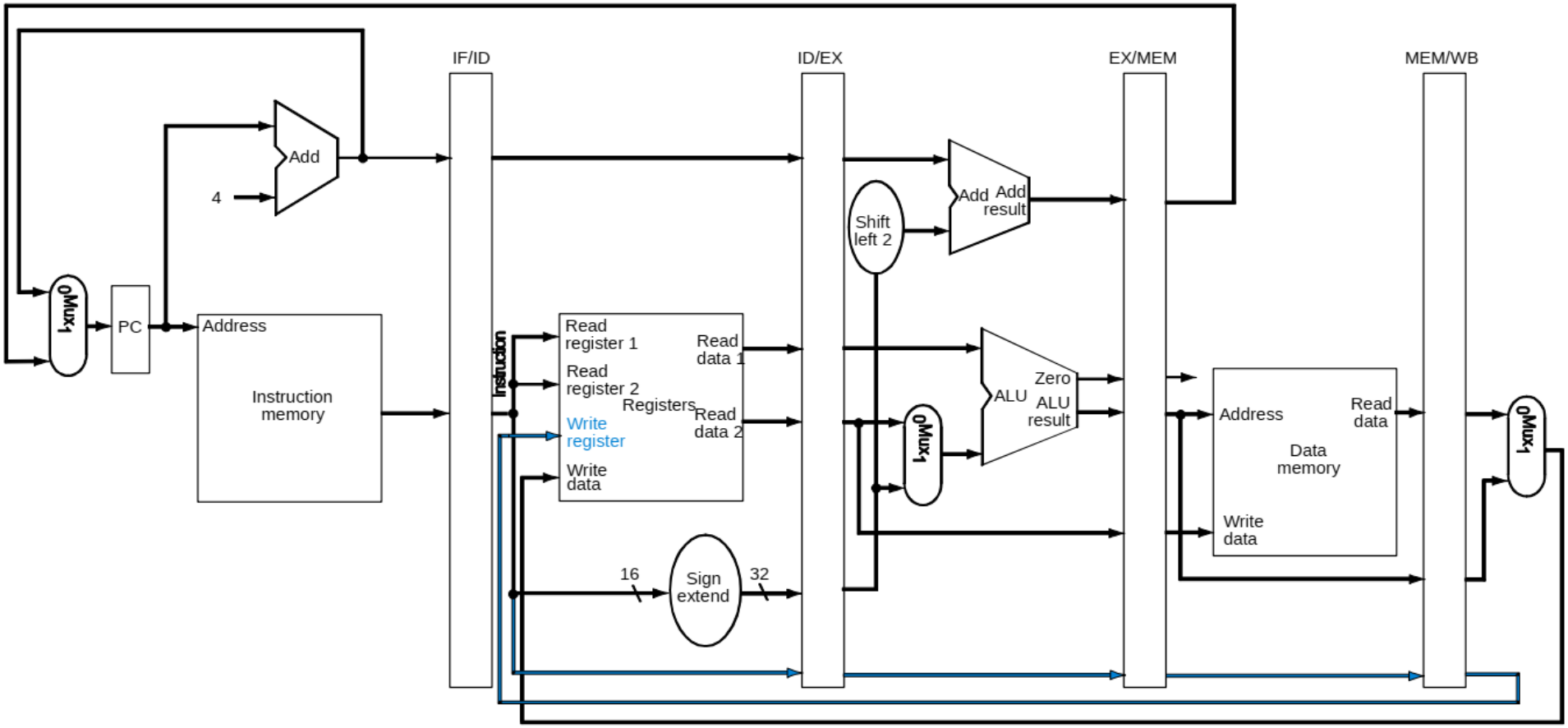
- O que precisamos adicionar para realmente dividir o caminho de dados em estágios?

Pipelined Datapath



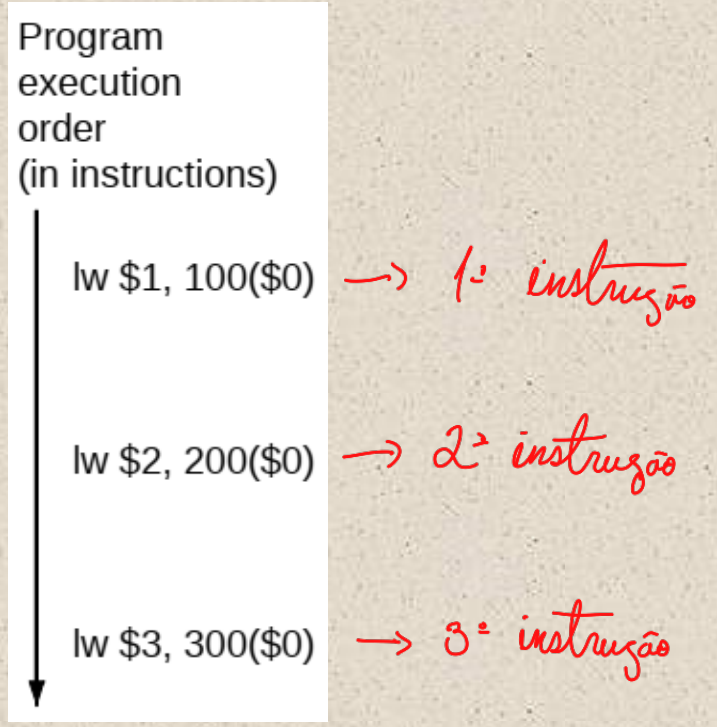
- Instrução *load* leva a problemas de dependência na busca pelo registrador de escrita!

Datapath correto



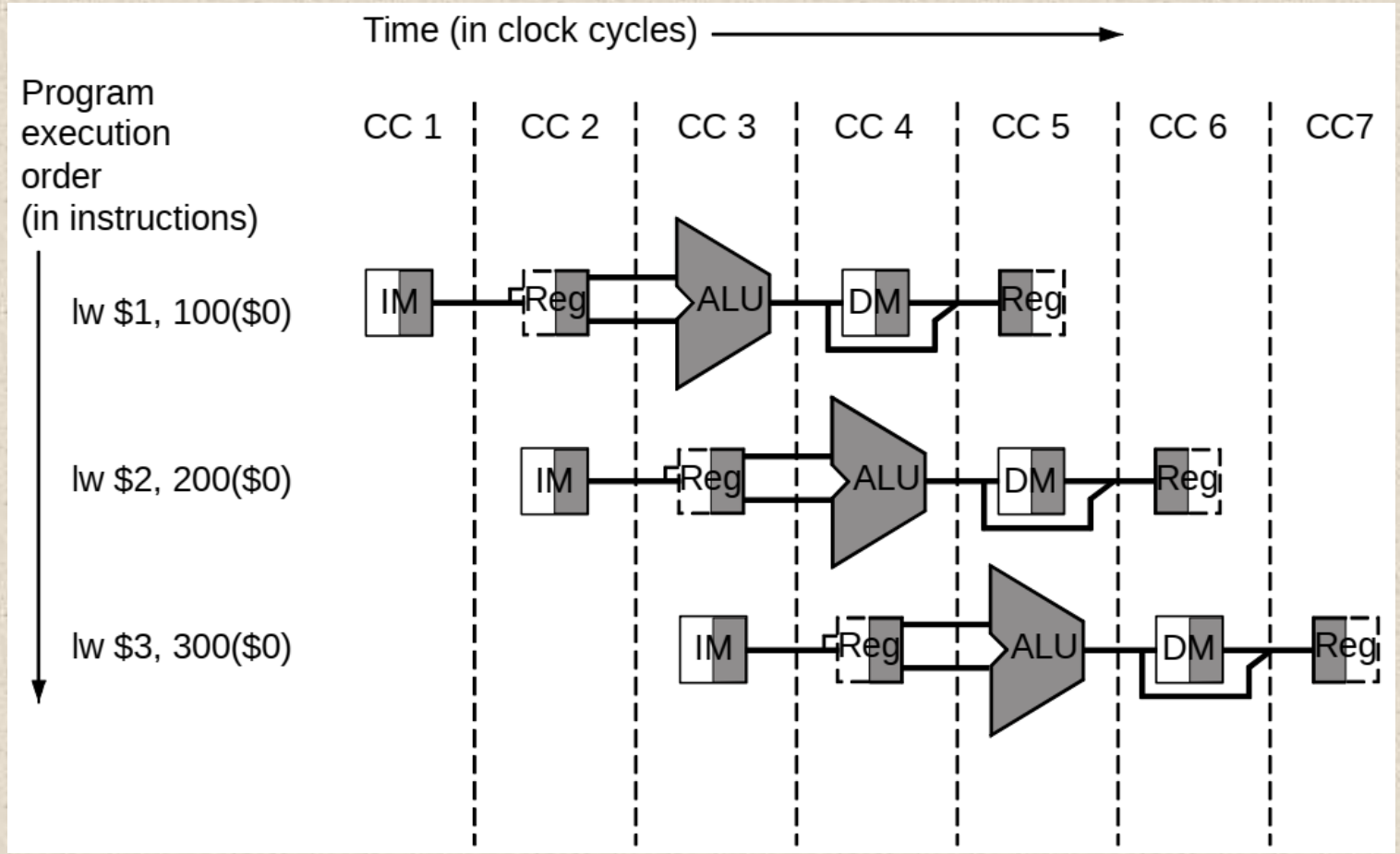
- Solução para *load*: armazenar também o registrador de escrita ao longo do caminho de dados (linha azul);
- O mesmo para instruções do tipo *R*.

Representando graficamente *pipelines*

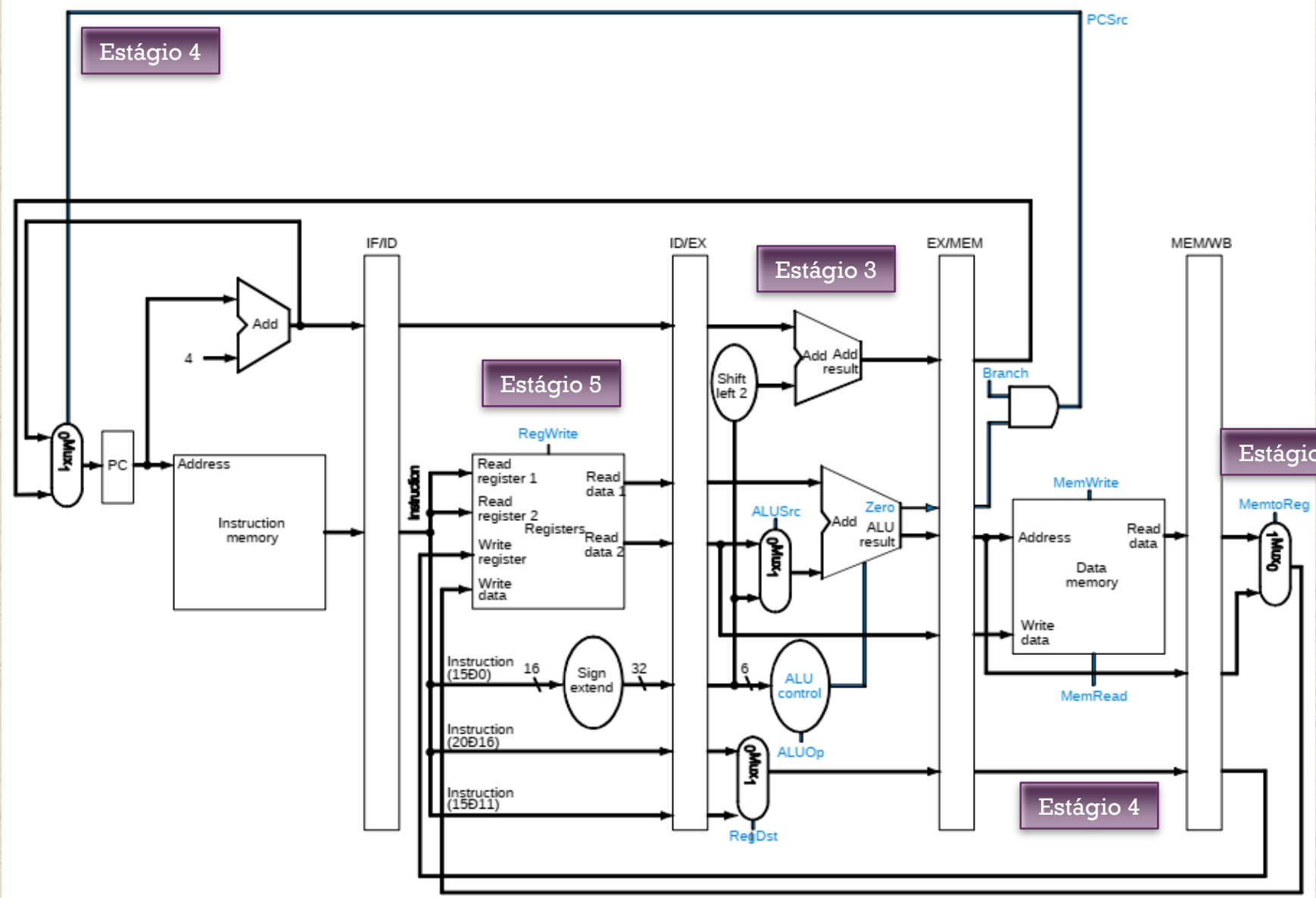


- Pode ajudar a responder perguntas como:
  - Quantos ciclos são necessários para executar este código?
  - O que a ULA está fazendo durante o ciclo 4?
  - Use esta representação para ajudar a entender os caminhos de dados

Representando graficamente *pipelines*



Controle do *pipeline*



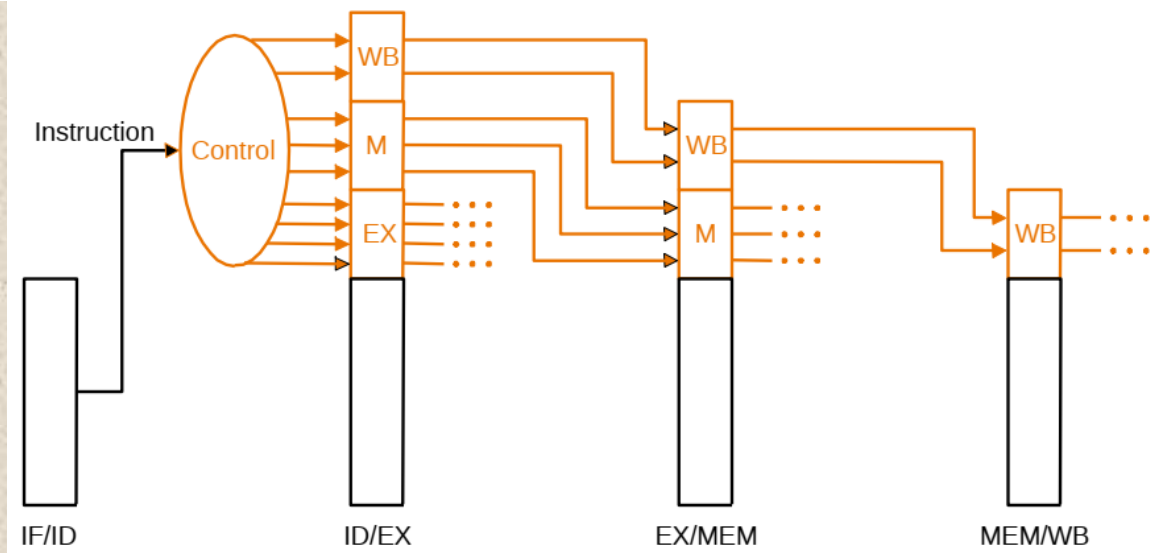
Controle do *pipeline*

- Temos 5 etapas. O que precisa ser controlado em cada etapa?
  - Busca de instruções e incremento de PC
  - Decodificação de instruções / busca de registro
  - Execução
  - Estágio da memória
  - Escrita de volta
- Como o controle seria feito em uma fábrica de automóveis?
  - Um centro de controle sofisticado dizendo a todos o que fazer? Não, definir controle pela instrução de cada etapa;
  - Devemos usar uma máquina de estados finitos? Complexo para todas as combinações possíveis de sequência de instruções.

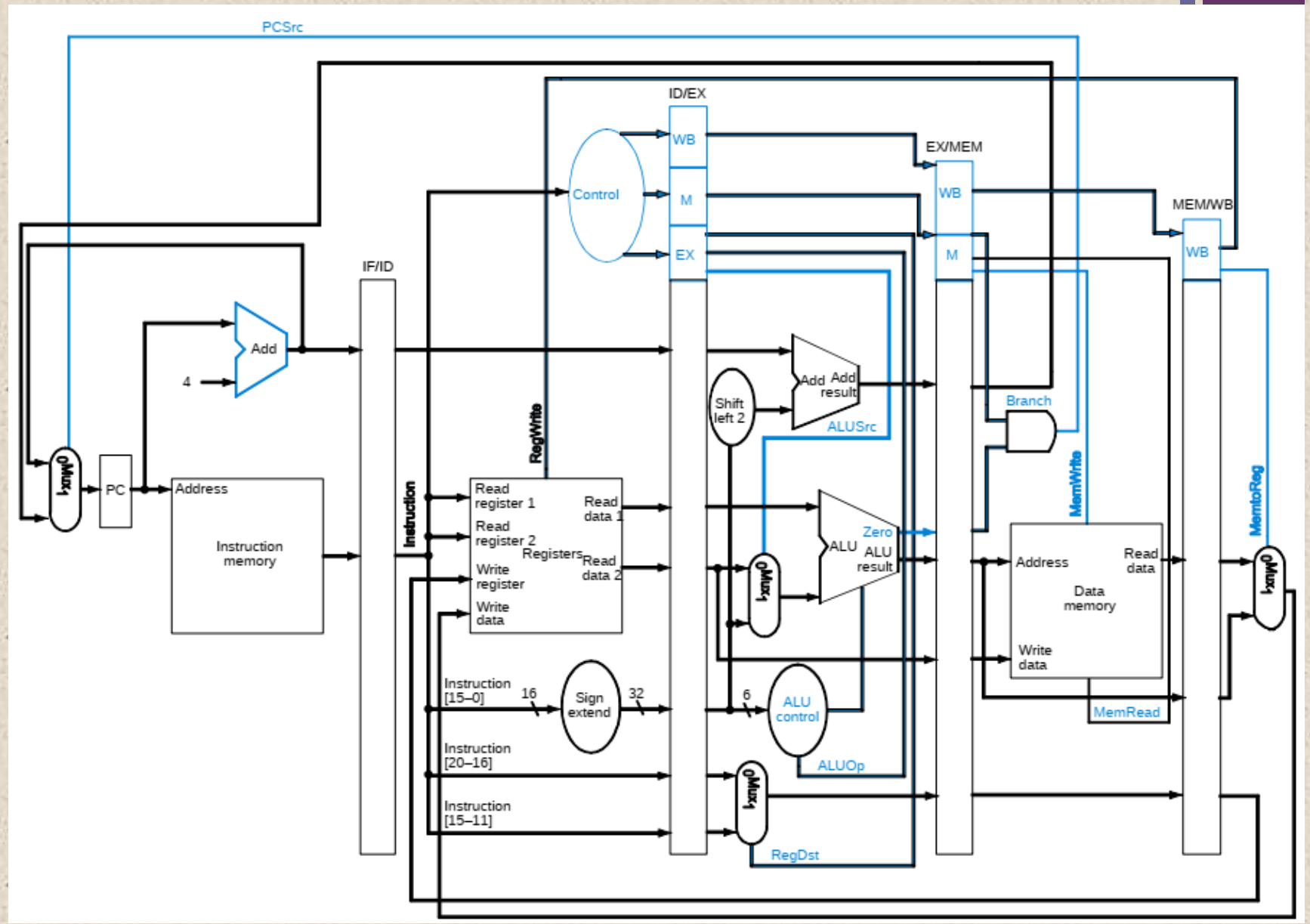
Controle do *pipeline*

- Passar os sinais de controle pelos registradores intermediários, assim como os dados.

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to 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

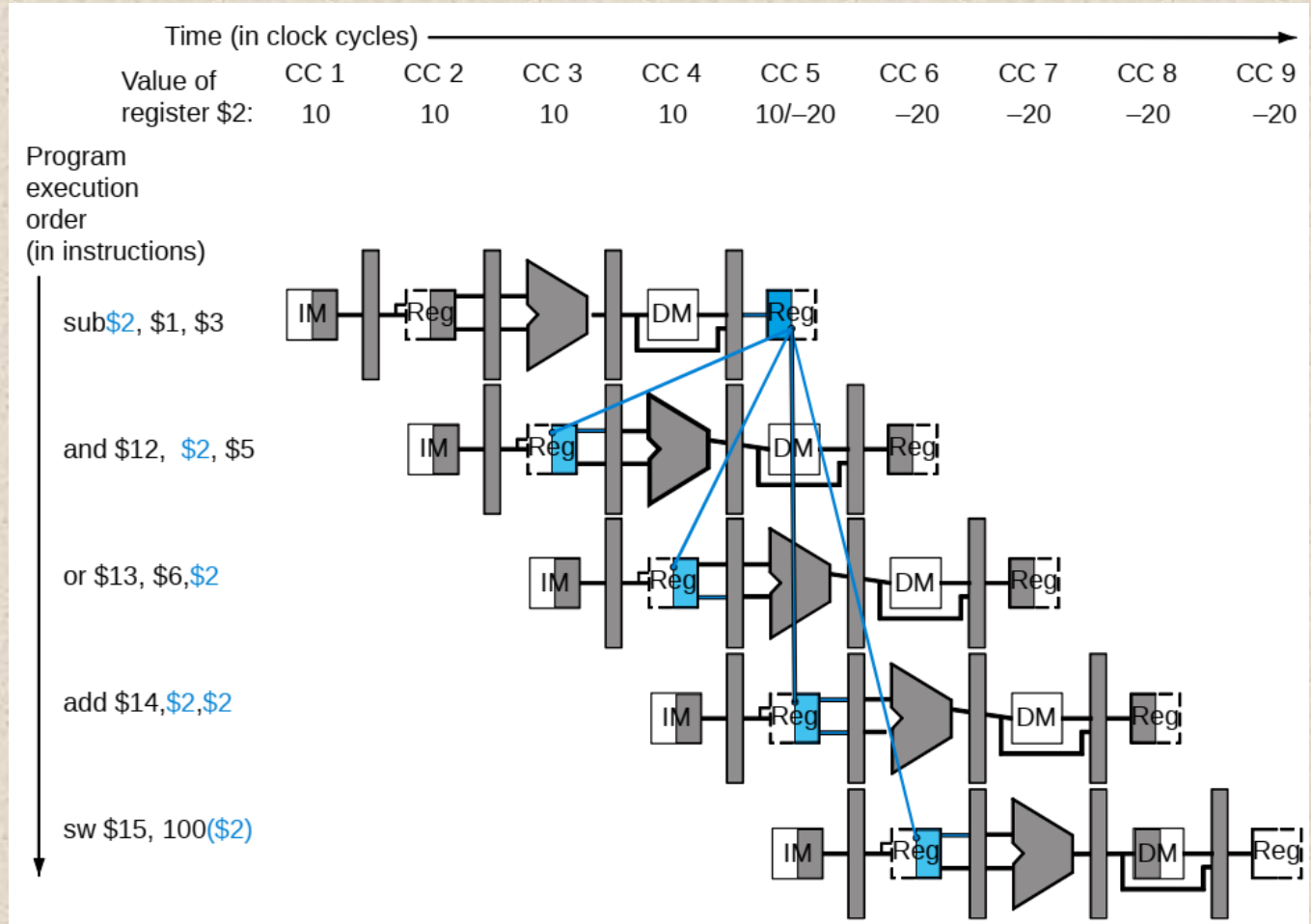


Datapath com controle



Conflitos de dados - dependências

- Problema ao iniciar a próxima instrução antes de terminar a primeira
  - dependências que "voltam no tempo" são conflitos de dados



Solução de software

- Tenha a garantia do compilador sem conflitos;
- Onde inserimos as instruções "nops"? Como usando o registrador $zero;

  sub $2, $1, $3

  and $12, $2, $5

  or $13, $6, $2

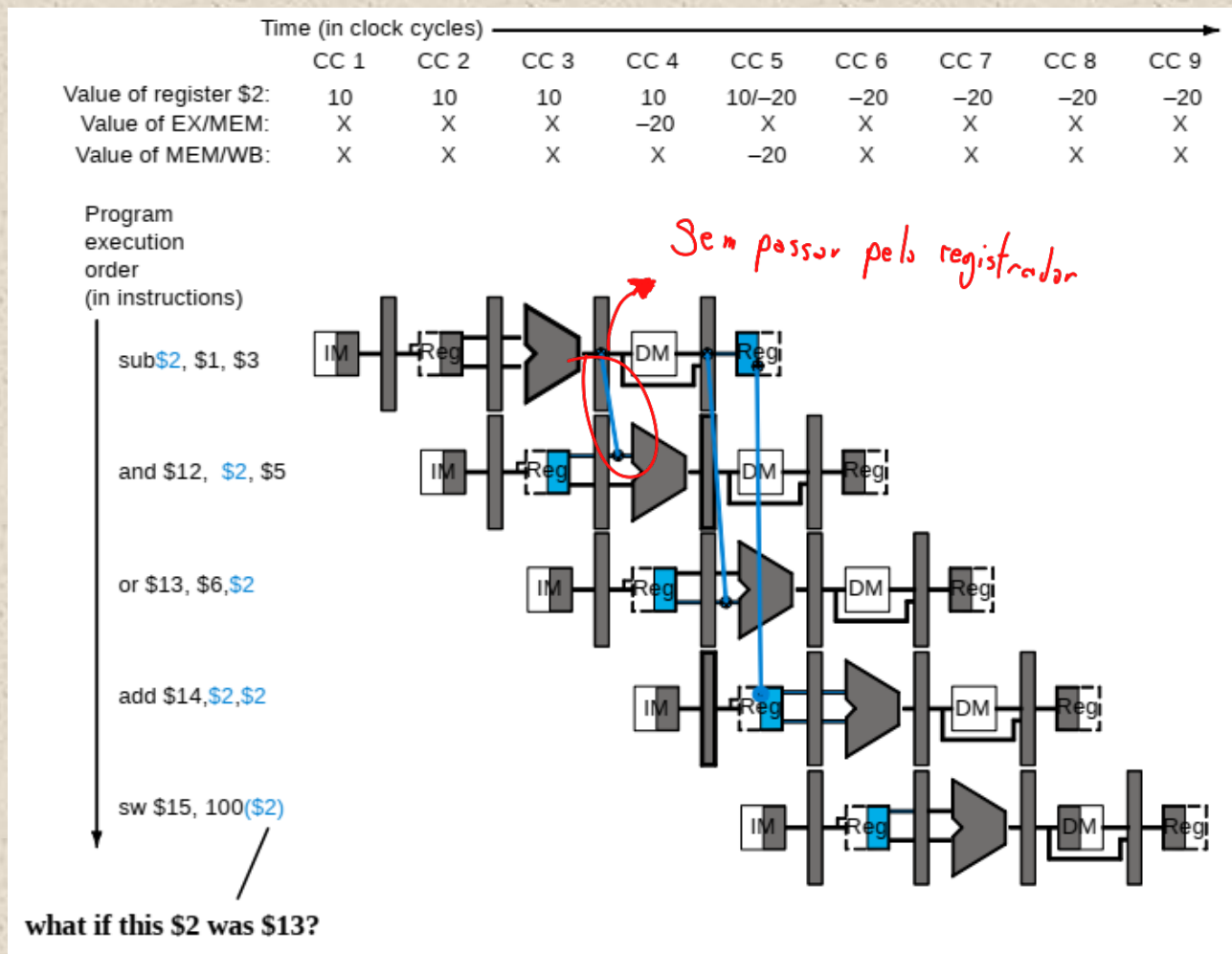
  add $14, $2, $2

  sw $15, 100 ($2)

- Problema: isso realmente atrasa a execução, mas é algo comum.

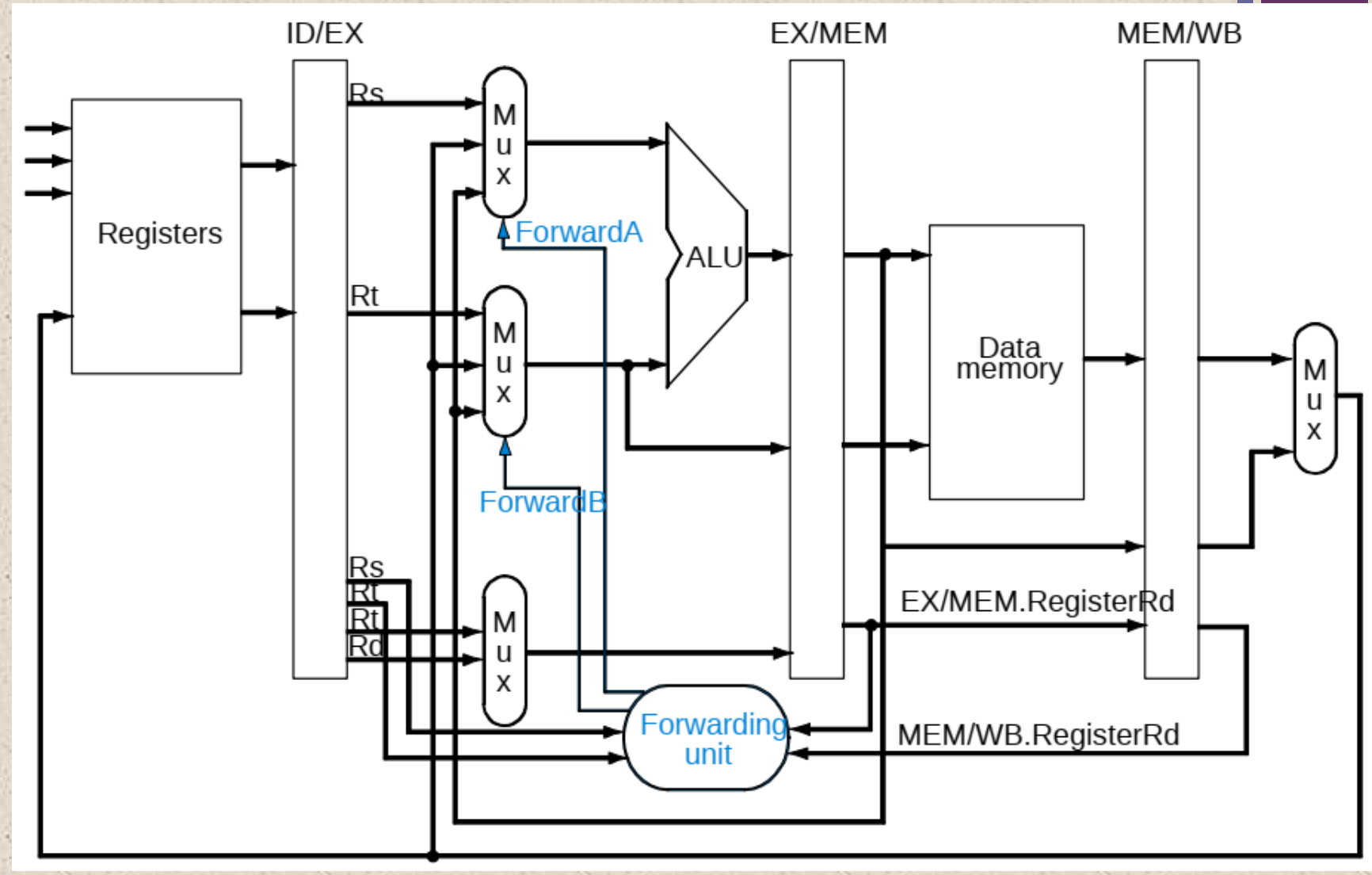
Forwarding

- Usar resultados temporários, e não esperar que eles sejam escritos nos registradores
  - *Forwarding* da ULA



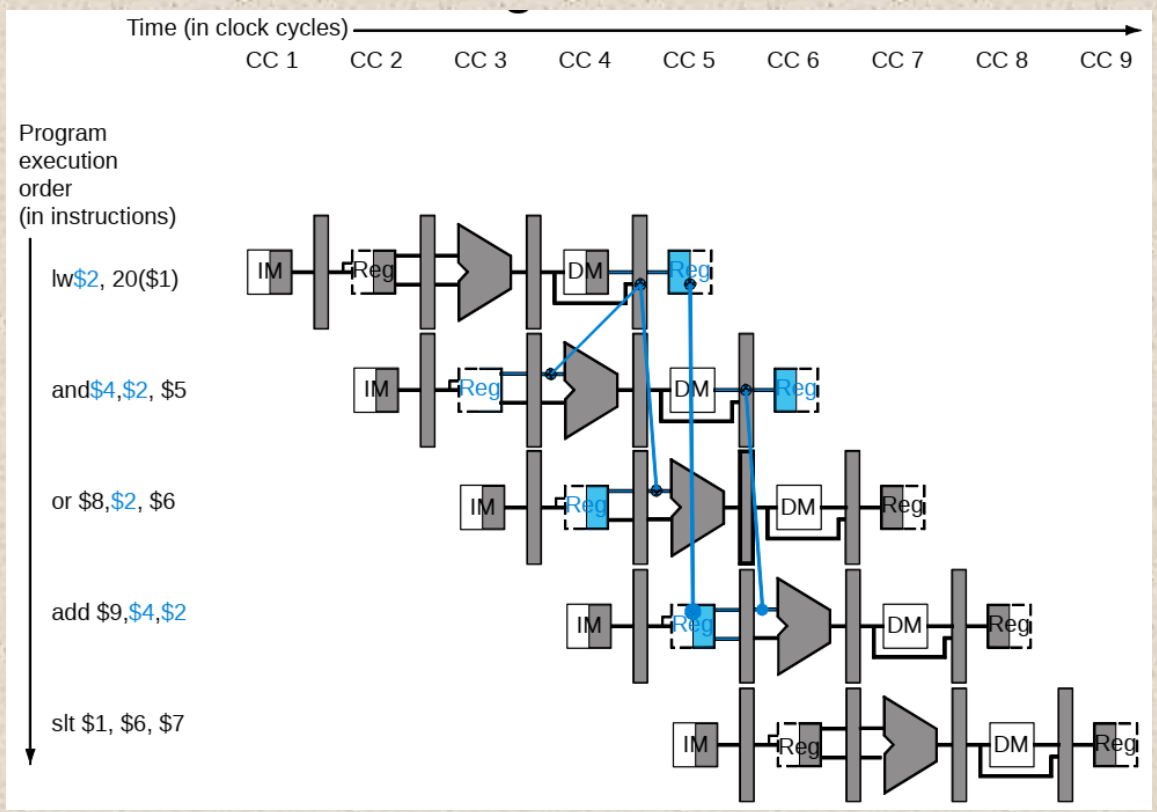
Forwarding

A ideia principal (alguns detalhes ocultos)



Nem sempre é possível fazer *forward*

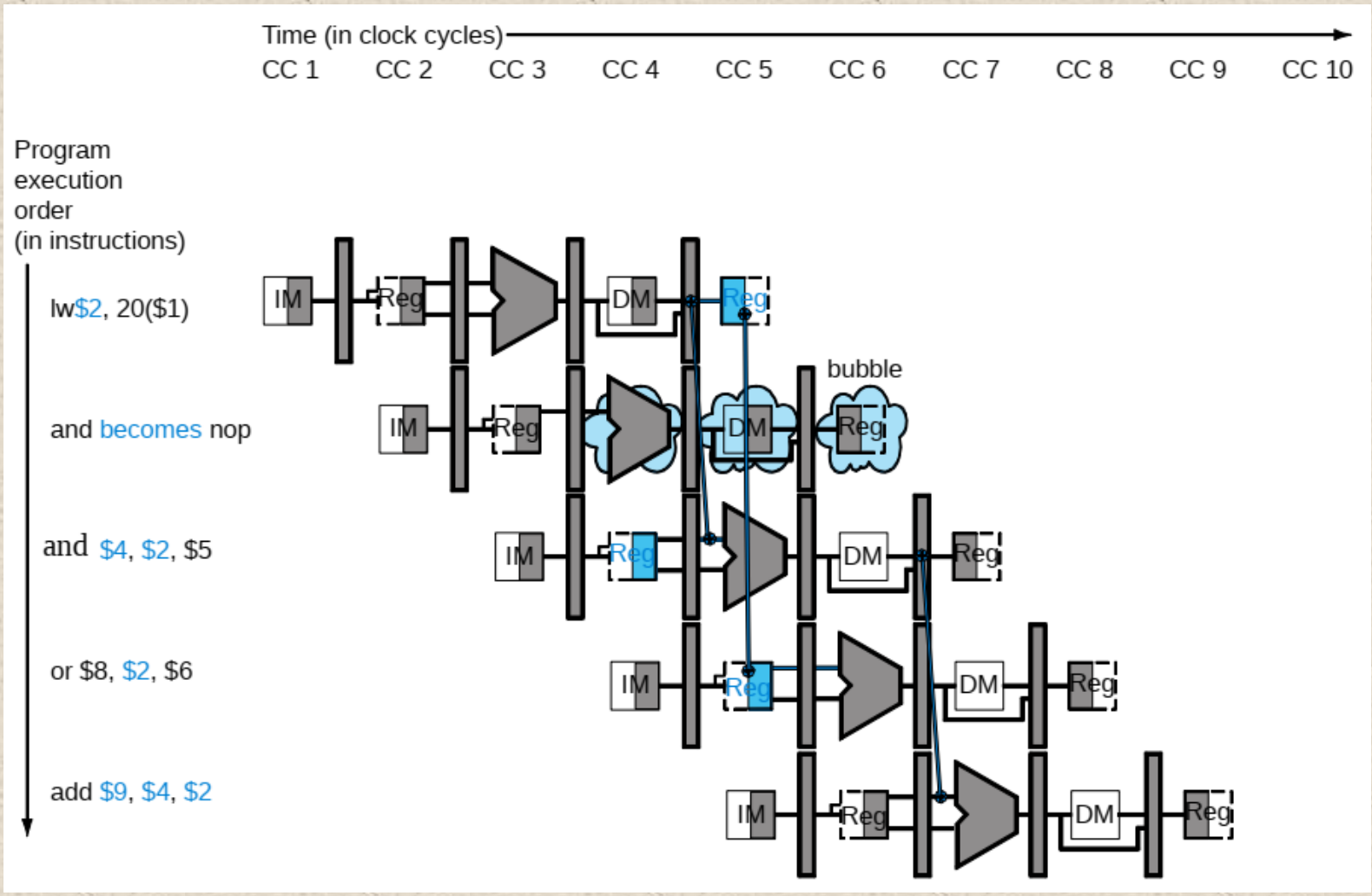
- A *load word* ainda pode causar um conflito:
  - uma instrução tenta ler um registrador seguindo uma instrução de *load* que escreve no mesmo registrador.



- Assim, precisamos de uma unidade de detecção de conflito.

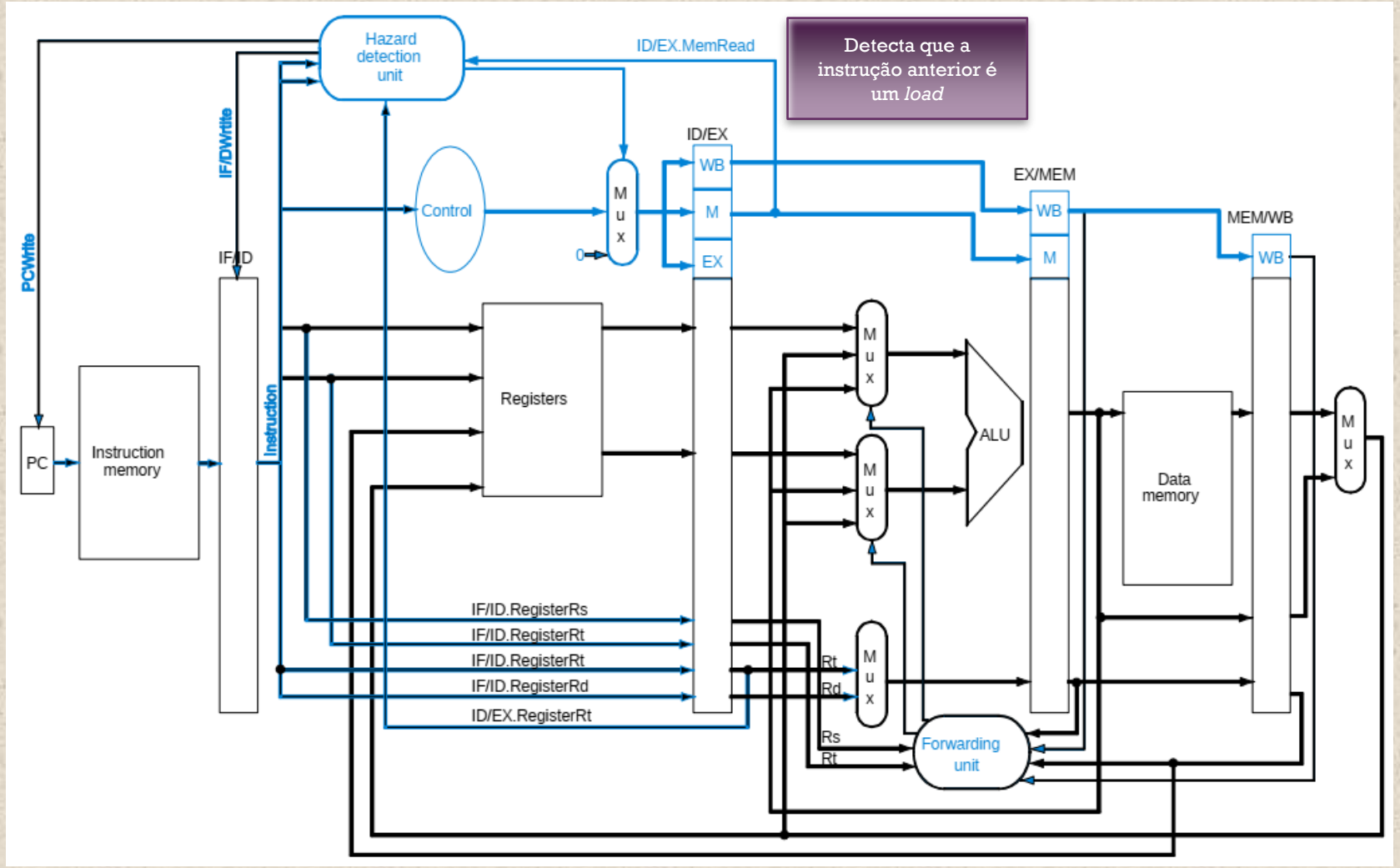
Parar o *pipeline*

- Podemos parar o *pipeline* mantendo uma instrução no mesmo estágio



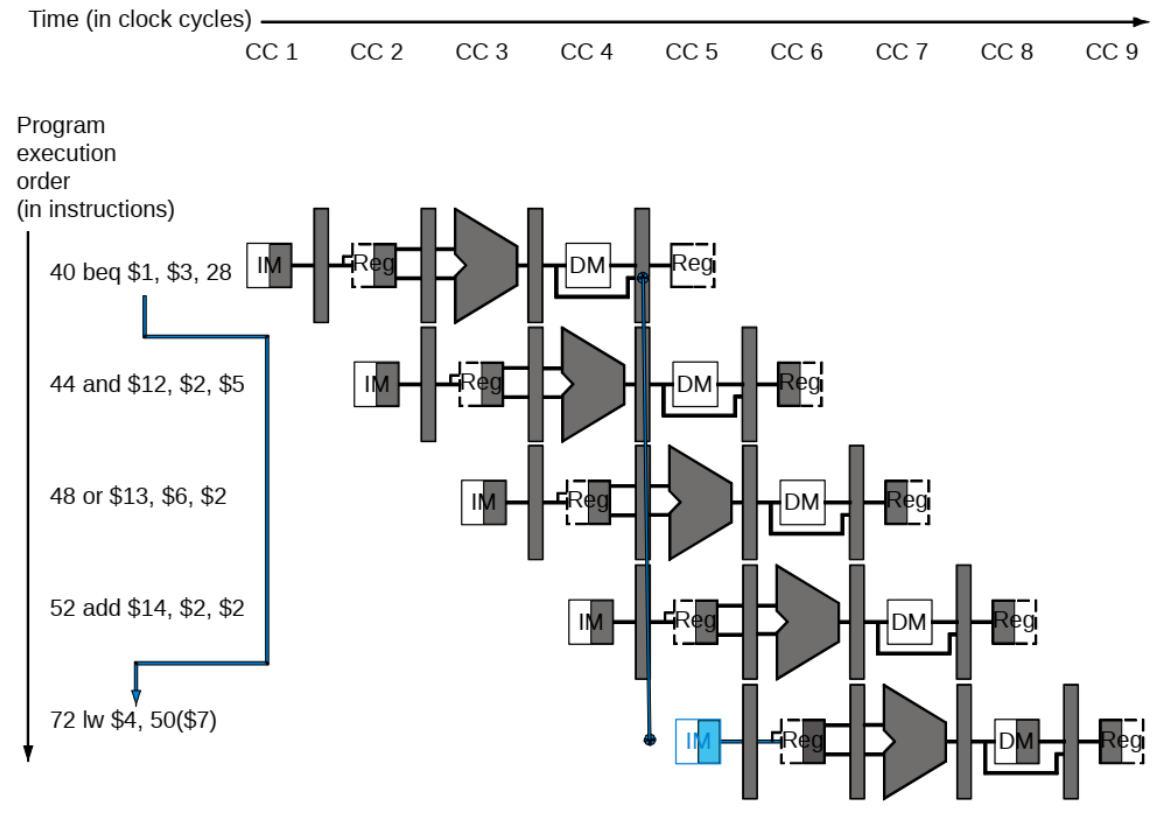
Unidade de detecção de conflito

- Parar deixando uma instrução que não escreverá nada seguir em frente



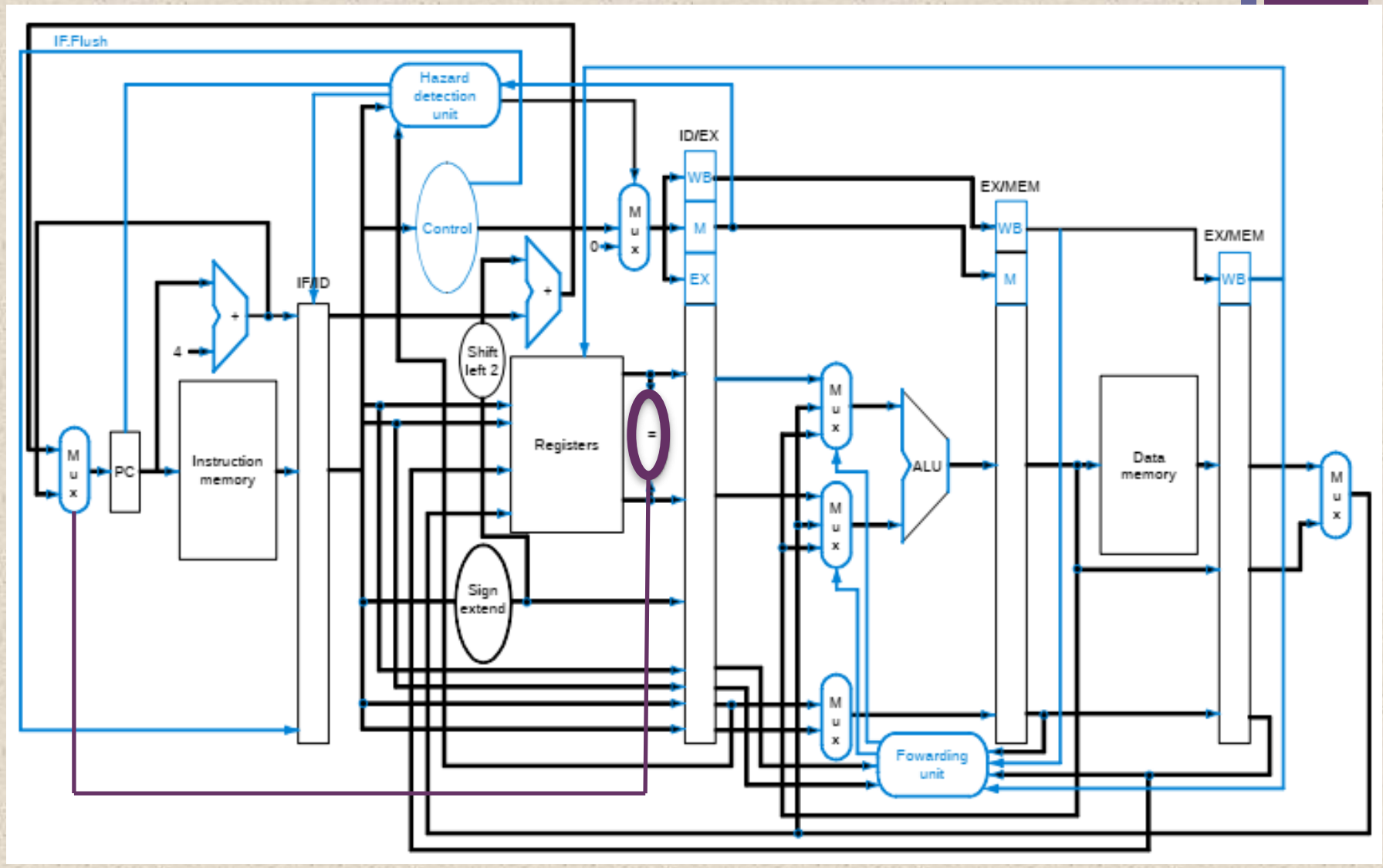
Conflitos de controle (desvios)

- Quando decidimos desviar, outras instruções estão em andamento!



- Estamos prevendo "desvio não executado"
  - precisa adicionar hardware para instruções *flushing* se estivermos errados

+ Instruções *flushing*



Observação: mudamos a decisão de desvio para o estágio 2 e adicionamos um nop na instrução seguinte.

Desvios

- Se o desvio for tomado, temos uma paralização por um ciclo de clock;
- Para nosso design simples, isso é razoável;
- Com *pipelines* mais profundos, aumentos de paralização e previsão de desvio estática prejudica drasticamente o desempenho;
  - Instruções para for
- Solução: previsão dinâmica de desvio → Veremos...
  - Processadores modernos preveem corretamente 95% das vezes!: princípio da localidade → loops
- Tente evitar parar o *pipeline*! Por exemplo, reordene estas instruções:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Pipeline MIPS

Agradeço a Prof. Dr. Fábio A. M. Cappabianco, pelos materiais disponibilizados.