

# CCF 252 - Organização de Computadores I

## O Processador

Prof. José Augusto Nacif – jnacif@ufv.br

# Introdução

Referência: Capítulo 4, Seção 4.1.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.



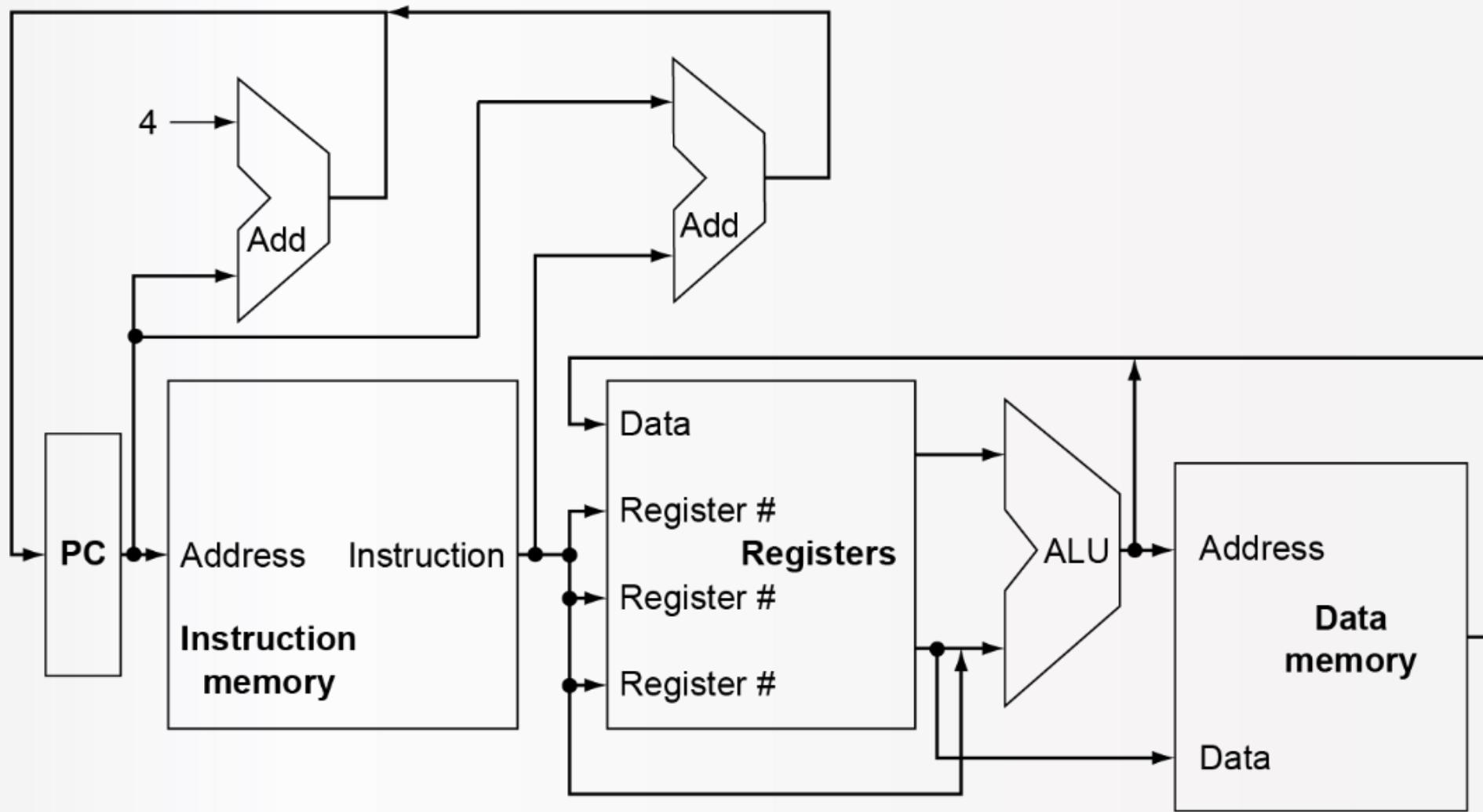
# Introdução

- Fatores de desempenho da CPU
  - Contagem de instruções: determinado pelo ISA e pelo compilador
  - CPI e tempo de ciclo: determinado pelo hardware da CPU
- Vamos estudar duas implementações RISC-V
  - Uma versão simplificada
  - Uma versão mais realista com pipeline
- Subconjunto simples, mostra a maioria dos aspectos
  - Acesso à memória: lw, sw
  - Aritmético / lógico: add, sub, and, or
  - Desvio: beq

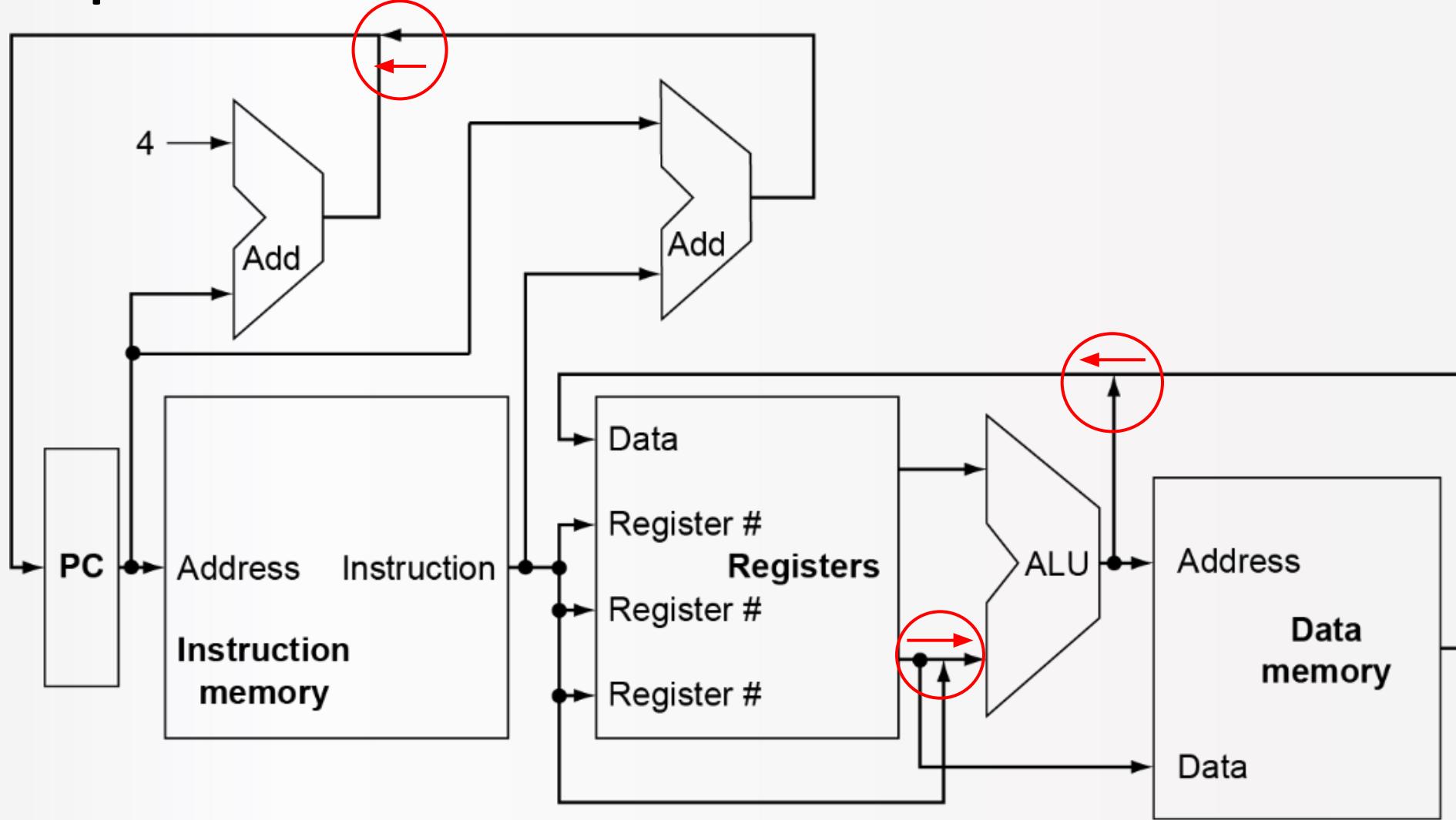
# Execução de instruções

- Busca de instruções: PC = memória de instrução
- Banco de registradores: Ler de acordo com o número
- Dependendo da classe de instrução
  - Utilizar ALU para calcular
    - Resultado da operação aritmética
    - Endereço de memória para carregar / armazenar
    - Comparação de desvios
  - Acesso à memória de dados para carregar / armazenar
  - PC = endereço de destino ou PC + 4

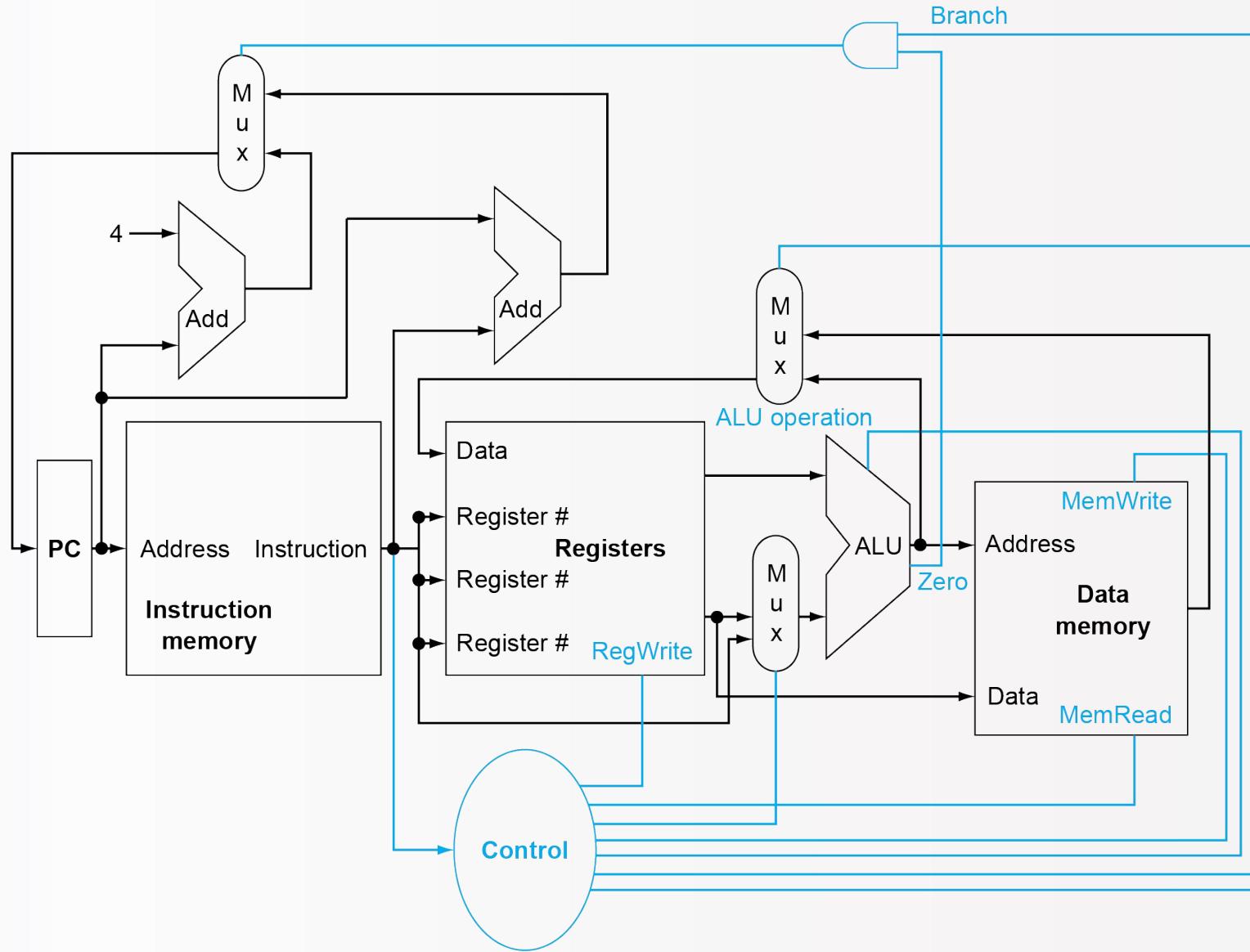
# Visão geral do caminho de dados da CPU



# Multiplexadores



# Controle



# Projeto Lógico

Referência: Capítulo 4, Seção 4.2.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

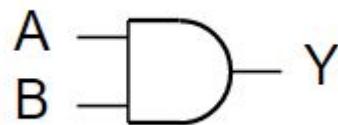


# Noções básicas de projeto lógico

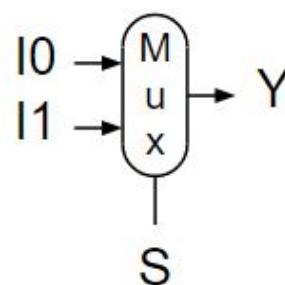
- Informação codificada em binário
  - 0V = 0, 5V = 1
  - Um fio por bit
  - Dados múltiplos em barramentos com vários fios
- Elemento combinacional
  - A saída é uma função da entrada
- Elementos de estado (sequencial)
  - Guardar informação
  - Elemento mais simples é o Flip-Flop D

# Elementos combinacionais

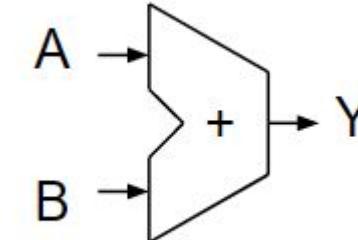
- Porta AND
  - $Y = A \& B$



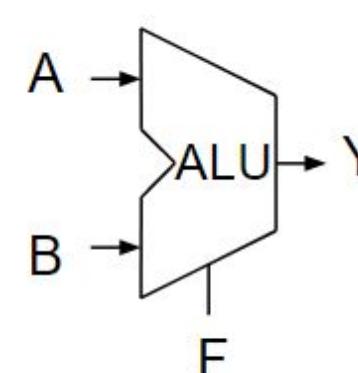
- Multiplexador
  - $Y = S ? I_1 : I_0$



- Somador
  - $Y = A + B$

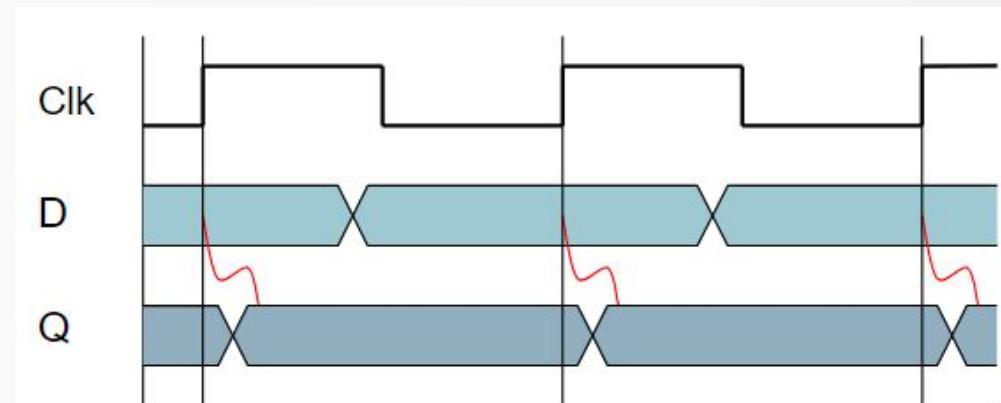
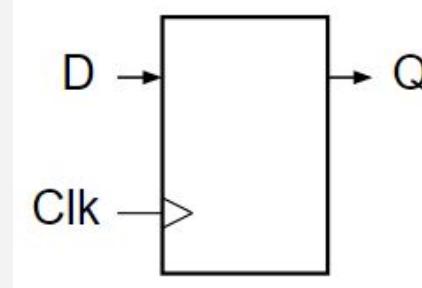


- Unidade de lógica aritmética
  - $Y = F(A, B)$



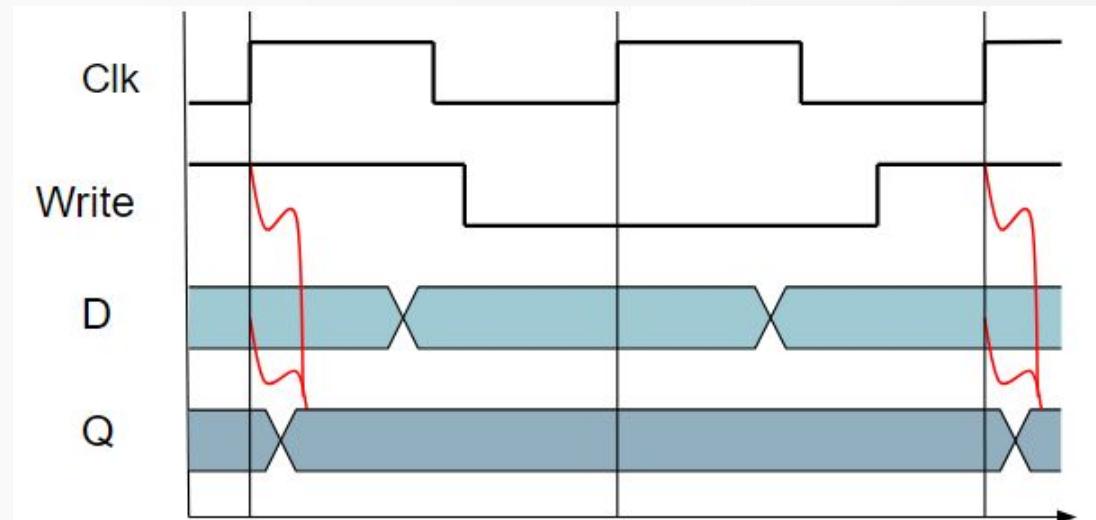
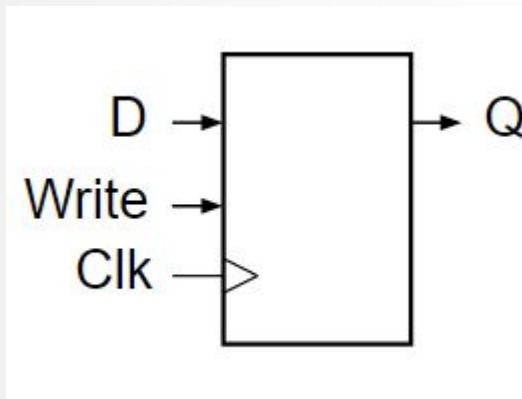
# Elementos sequenciais

- Registrador: armazena dados em um circuito
  - Usa um sinal de clock para determinar quando atualizar o valor armazenado
  - Acionado por borda: atualização quando Clk muda de 0 para 1



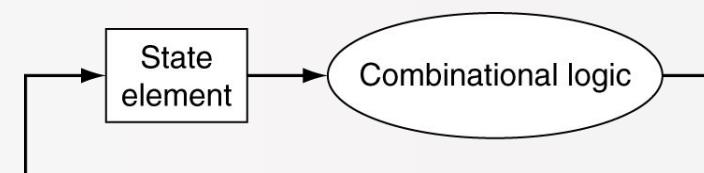
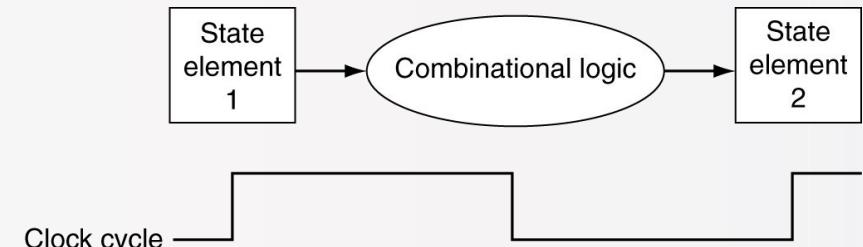
# Elementos sequenciais

- Registrador com controle de gravação
  - Somente atualizações na transição do clock quando a entrada de controle de gravação é 1
  - Usado quando o valor armazenado é necessário posteriormente



# Metodologia de clock

- A lógica combinacional transforma os dados durante os ciclos do clock
  - Entre as bordas do clock
    - Entrada dos elementos de estado
    - Saída para os elementos de estado
  - O maior atraso determina o período do clock
- Exemplo genérico de circuito que pode ser lido e escrito no mesmo ciclo de clock



# Construindo um Caminho de Dados

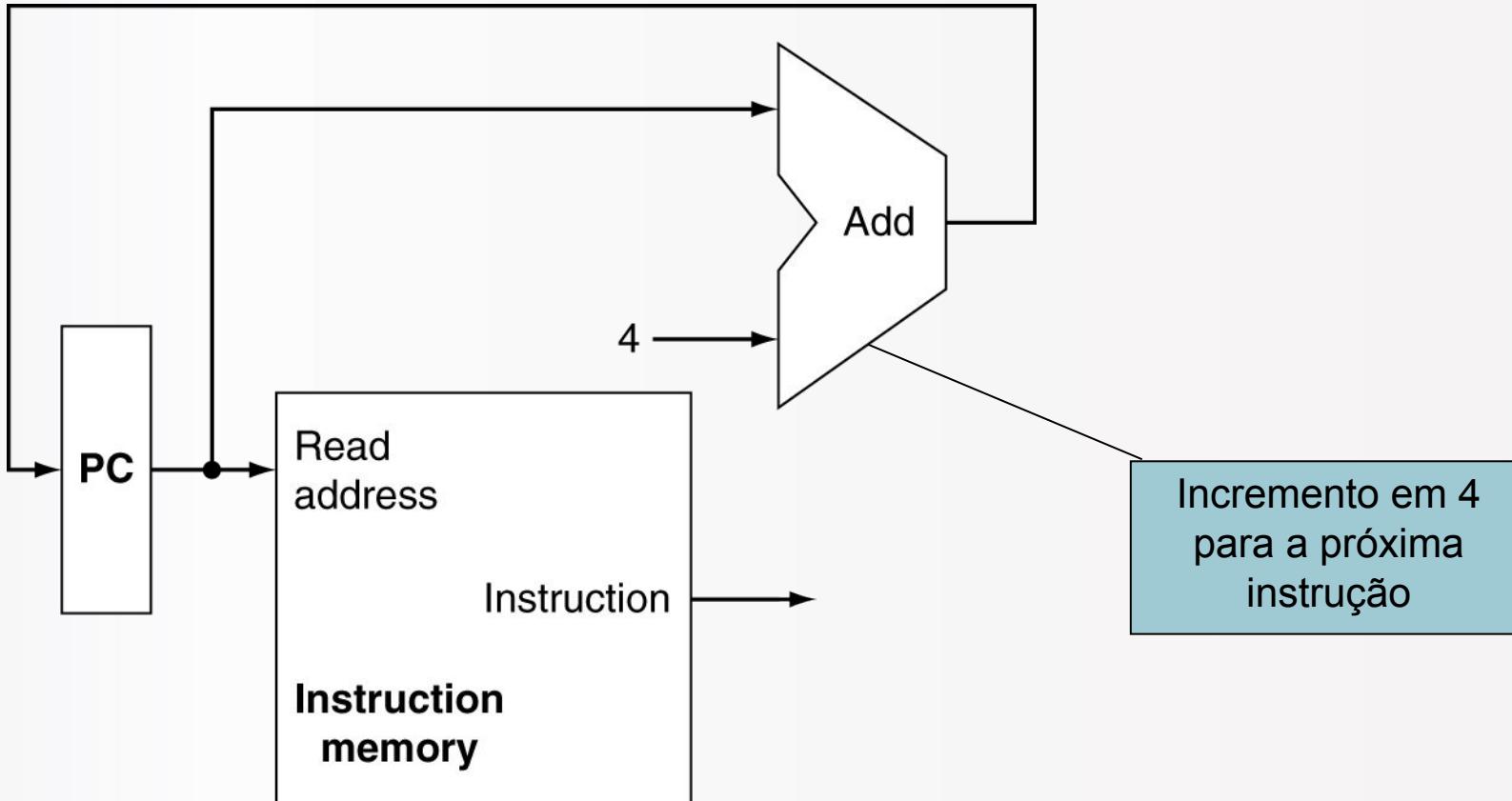
Referência: Capítulo 4, Seção 4.3.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Construindo um caminho de dados

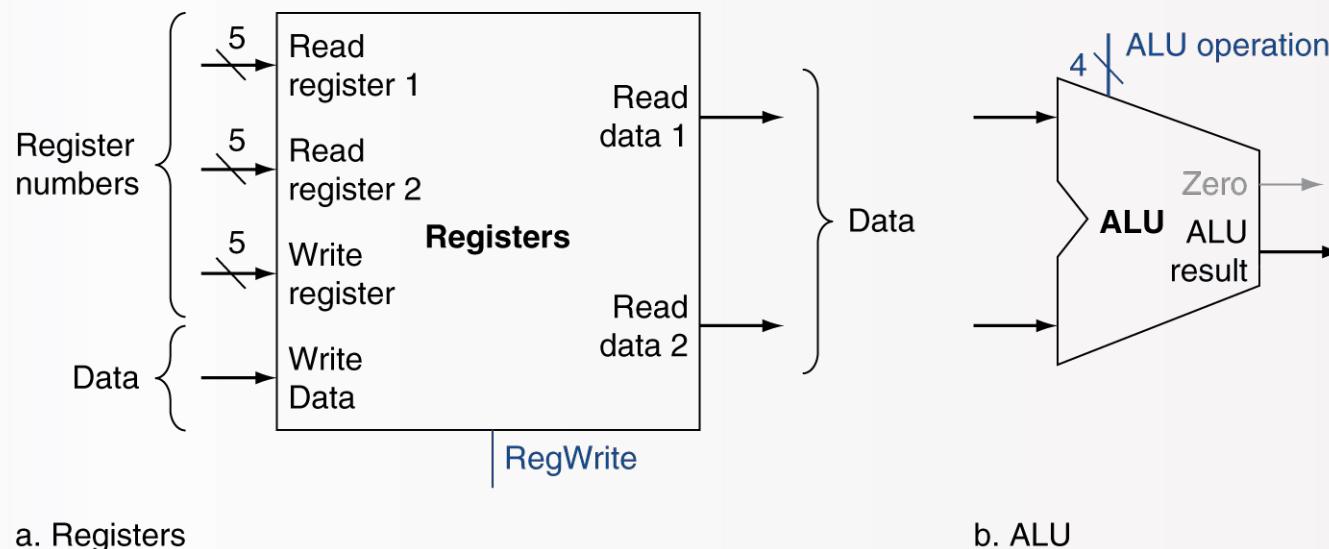
- Caminho de dados
  - Elementos que processam dados e endereços na CPU
    - Registradores, ALUs, multiplexadores, memórias, ...
- Vamos construir um caminho de dados RISC-V incrementalmente
  - Refinando o projeto da visão geral

# Busca de instrução



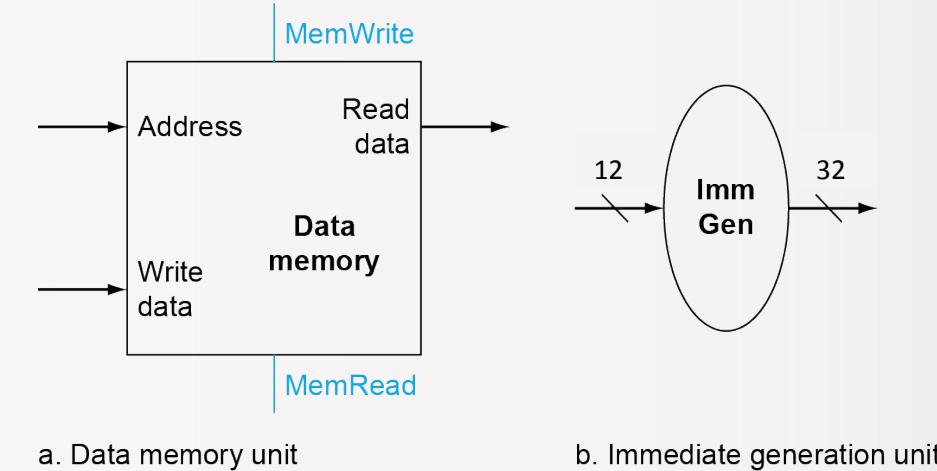
# Instruções de formato R

- Lê dois operandos de registrador
- Executa operação aritmética / lógica
- Grava resultado no registrador



# Carregar / armazenar instruções

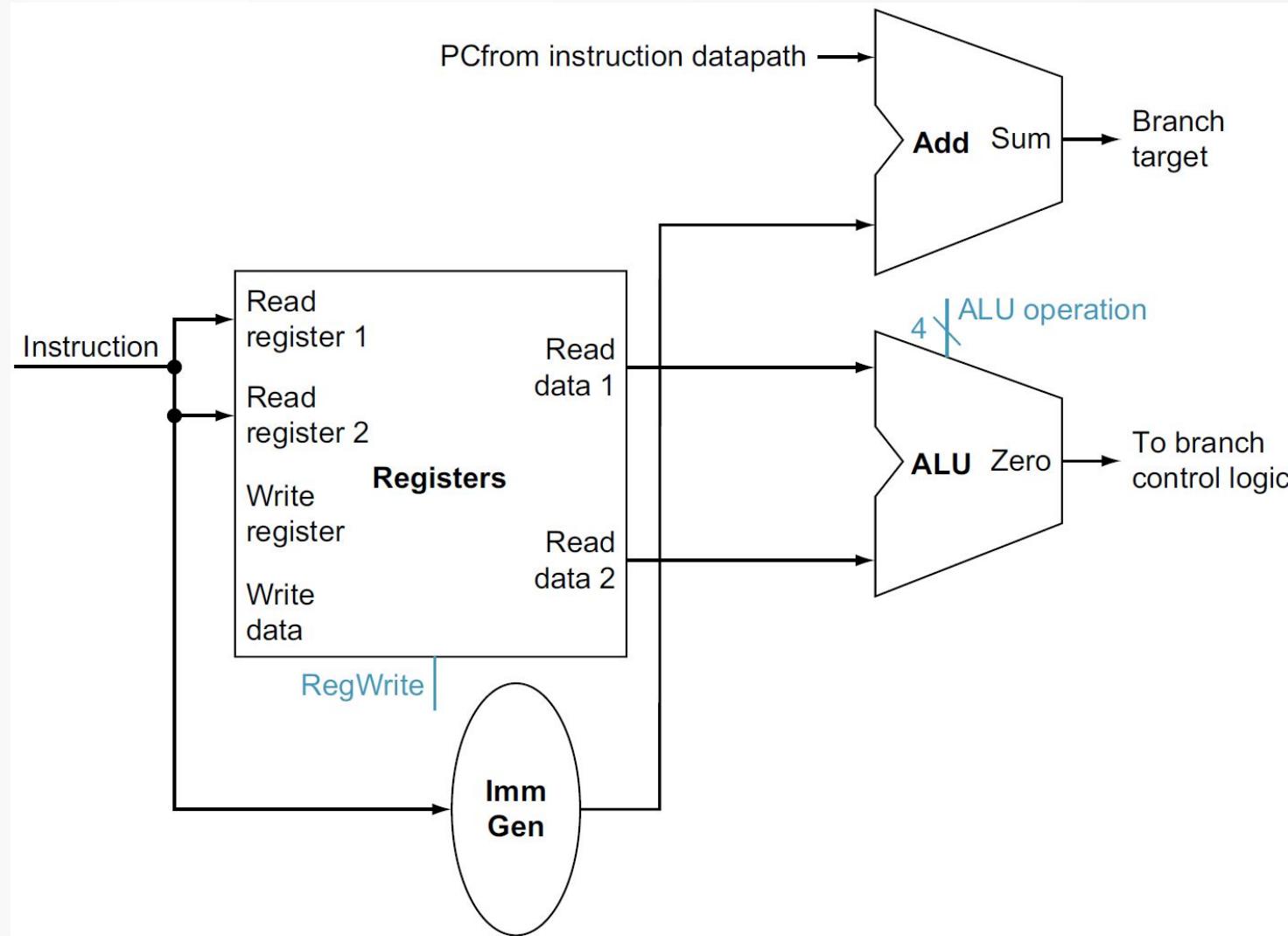
- Ler operandos de registrador
- Calcular o endereço somando o valor do registrador base o com o imediato de 12 bits da instrução
  - Soma realizada na ALU, com extensão de sinal para o imediato
- Carregar
  - Ler a memória e atualizar o registrador
- Armazenar
  - Grava o valor do registrador na memória



# Instruções de desvio

- Ler operandos do registrador
- Comparar operandos
  - Usar ALU
    - Subtrair e verificar a saída Zero
- Calcular endereço de destino
  - Extensão de sinal no deslocamento (imediato)
  - Deslocar 1 posição para a esquerda
  - Adicionar ao valor do PC

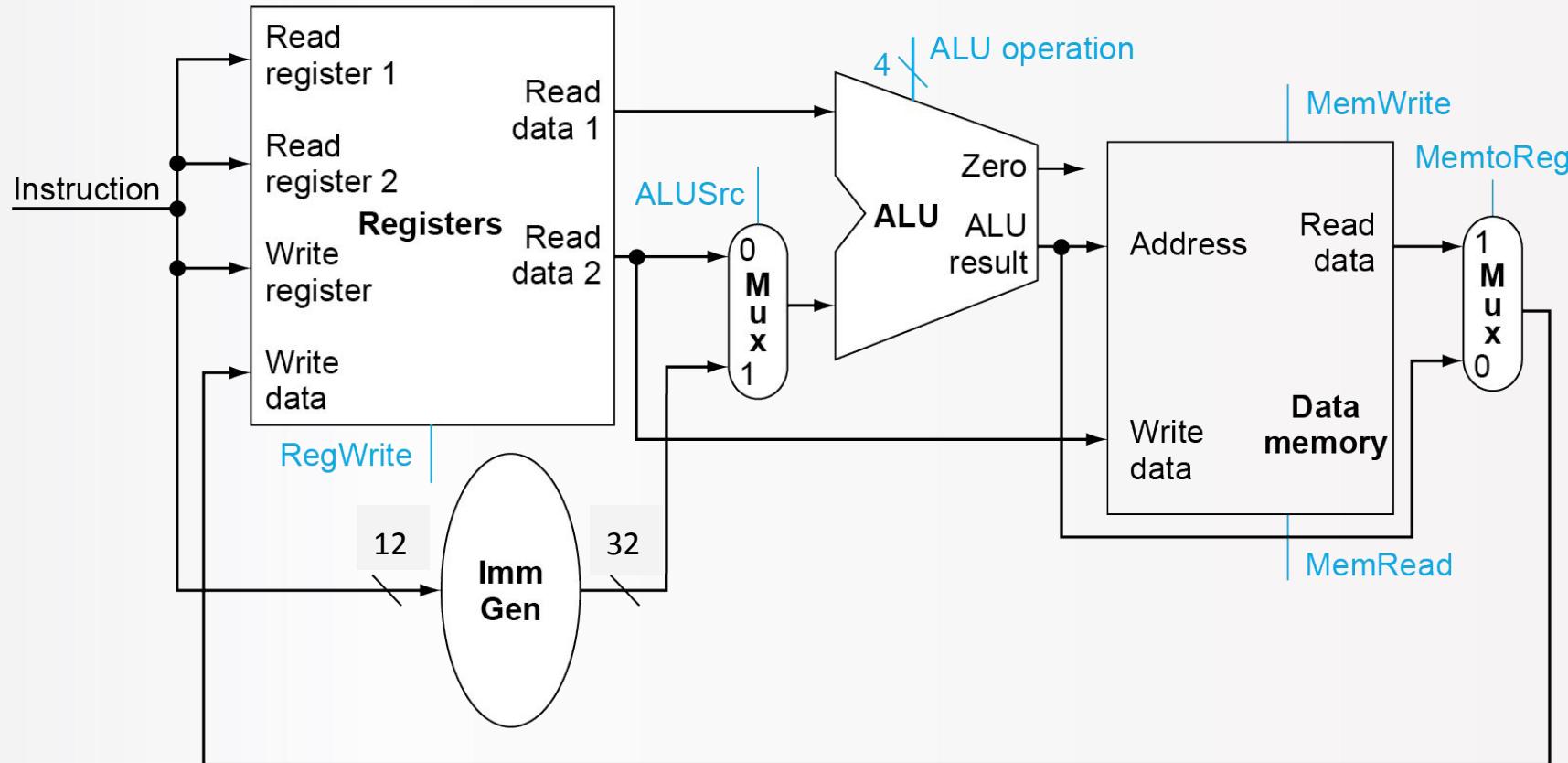
# Instruções de desvio



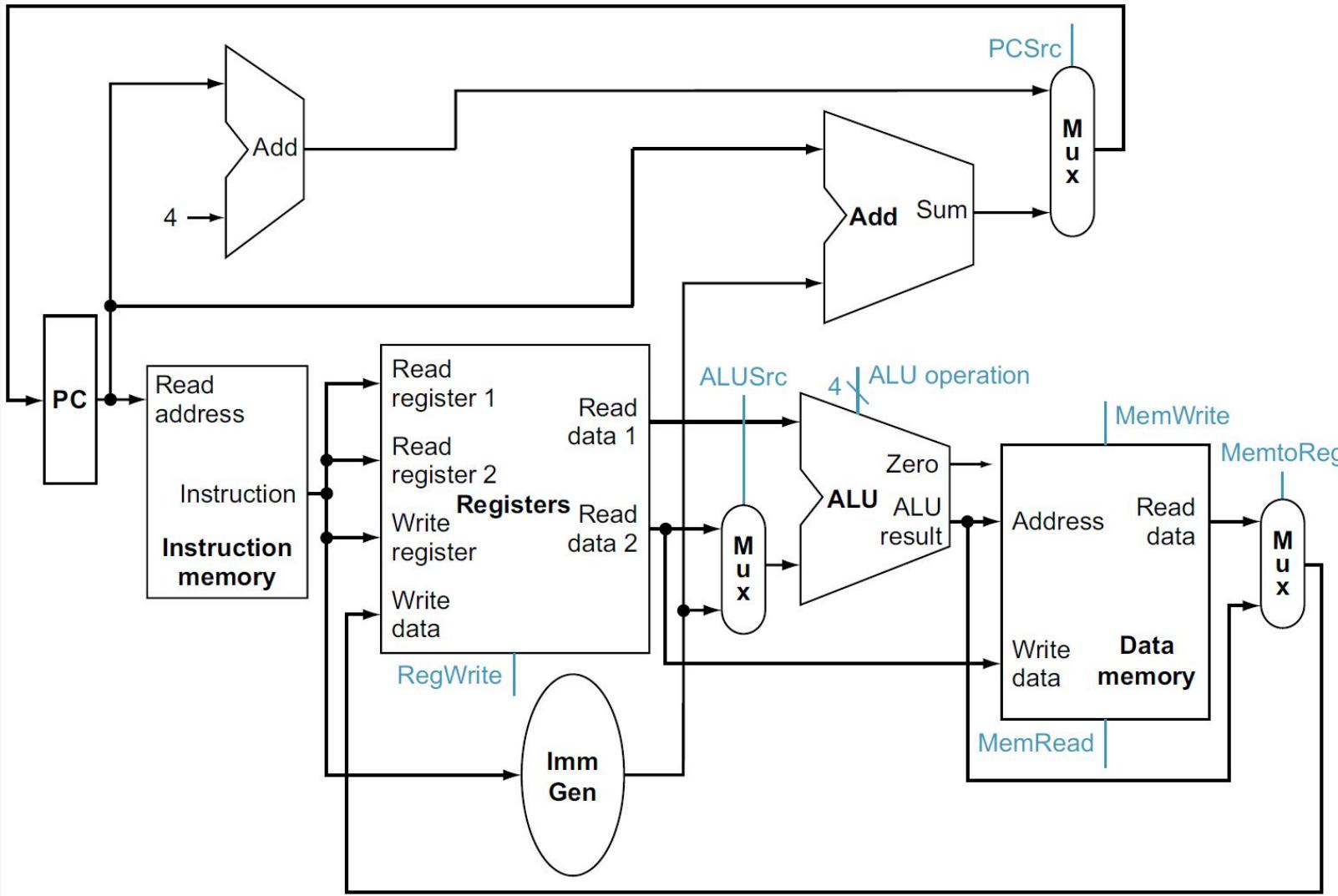
# Compondo os Elementos

- O caminho de dados da primeira versão faz uma instrução em um ciclo de clock
  - Cada elemento do caminho de dados só pode fazer uma função por vez
  - Portanto, precisamos de instruções separadas e memórias de dados
- Utilização de multiplexadores nos pontos nos quais há duas opções de origem dos dados em instruções diferentes

# Caminho de dados - Tipo R/Load/Store



# Caminho de dados completo



# Um Esquema de Implementação Simples

Referência: Capítulo 4, Seção 4.4.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Controle da ALU

- ALU usado para
  - Load / Store
    - Função = Soma
  - Desvio
    - F = Subtração
  - Tipo R
    - F depende do opcode

ALU control	Função
0000	AND
0001	OR
0010	Soma
0110	Subtração

# Controle da ALU

- Suponha ALUOp de 2 bits derivado do opcode
  - A lógica combinacional deriva o controle ALU

opcode	ALUOp	Operação	Opcode field	Função da ALU	ALU control
ld	00	Carrega registrador	XXXXXXXXXXXX	soma	0010
sd	00	Armazena registrador	XXXXXXXXXXXX	soma	0010
beq	01	Desvia se igual	XXXXXXXXXXXX	subtração	0110
R-type	10	add	100000	soma	0010
		sub	100010	subtração	0110
		and	100100	AND	0000
		or	100101	OR	0001

# A Unidade de Controle Principal

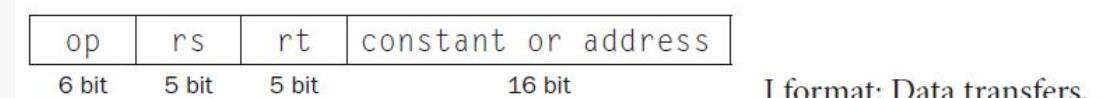
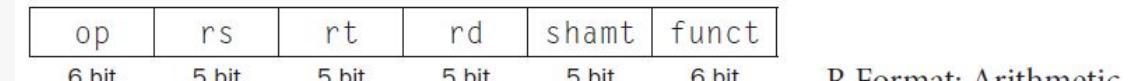
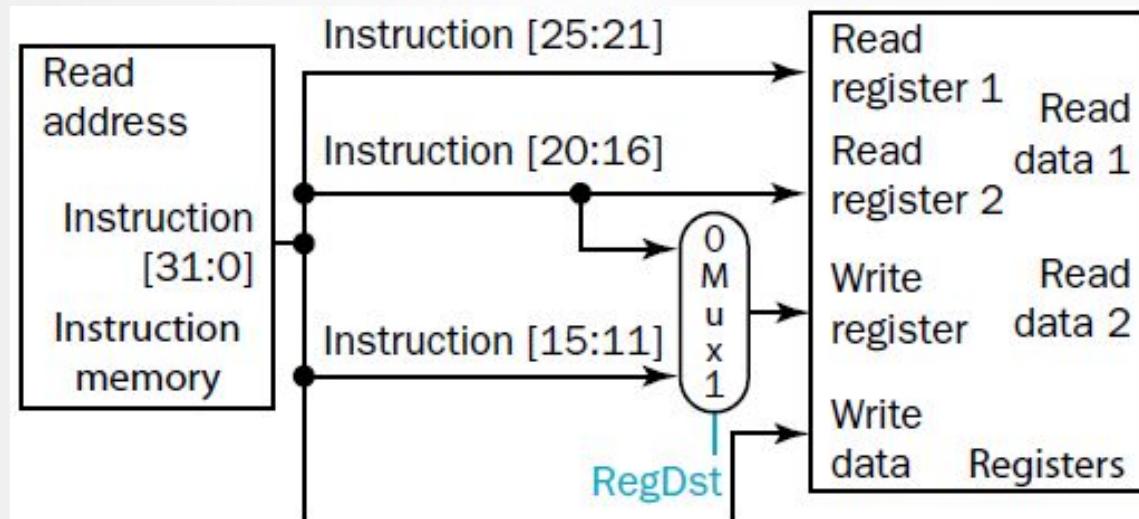
- Sinais de controle derivados da instrução

	Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type		funct7	rs2	rs1	funct3	rd	opcode
(b) I-type		immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type		immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type		immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp	Funct7 field												Funct3 field	Operation
	ALUOpI	ALUOpO	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]		
0	0	X	X	X	X	X	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	X	X	X	X	X	0110	
1	X	0	0	0	0	0	0	0	0	0	0	0	0010	
1	X	0	1	0	0	0	0	0	0	0	0	0	0110	
1	X	0	0	0	0	0	0	0	0	1	1	1	0000	
1	X	0	0	0	0	0	0	0	0	1	1	0	0001	

# Comparação RISC-V com MIPS

- Algumas alterações foram feitas para simplificar o hardware
  - Manter o Rd sempre nos mesmos campos economiza um mux 2:1



# Comparação RISC-V com MIPS

- Variações UJ e e SB

Name (Field size)	7 bits	5 bits	Field	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3		rd	opcode	Arithmetic instruction format
I-type		immediate[11:0]	rs1	funct3		rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]		opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]		opcode	Conditional branch format
UJ-type			immediate[20,10:1,11,19:12]			rd	opcode	Unconditional jump format
U-type			immediate[31:12]			rd	opcode	Upper immediate format

# Comparação RISC-V com MIPS

- Variações UJ e e SB
  - Redução de mux 3:1 para 2:1 (imm 19-12) e mux 4:1 para 2:1 (imm 10-01)
  - Simplificação total de 18 mux de 1 bit

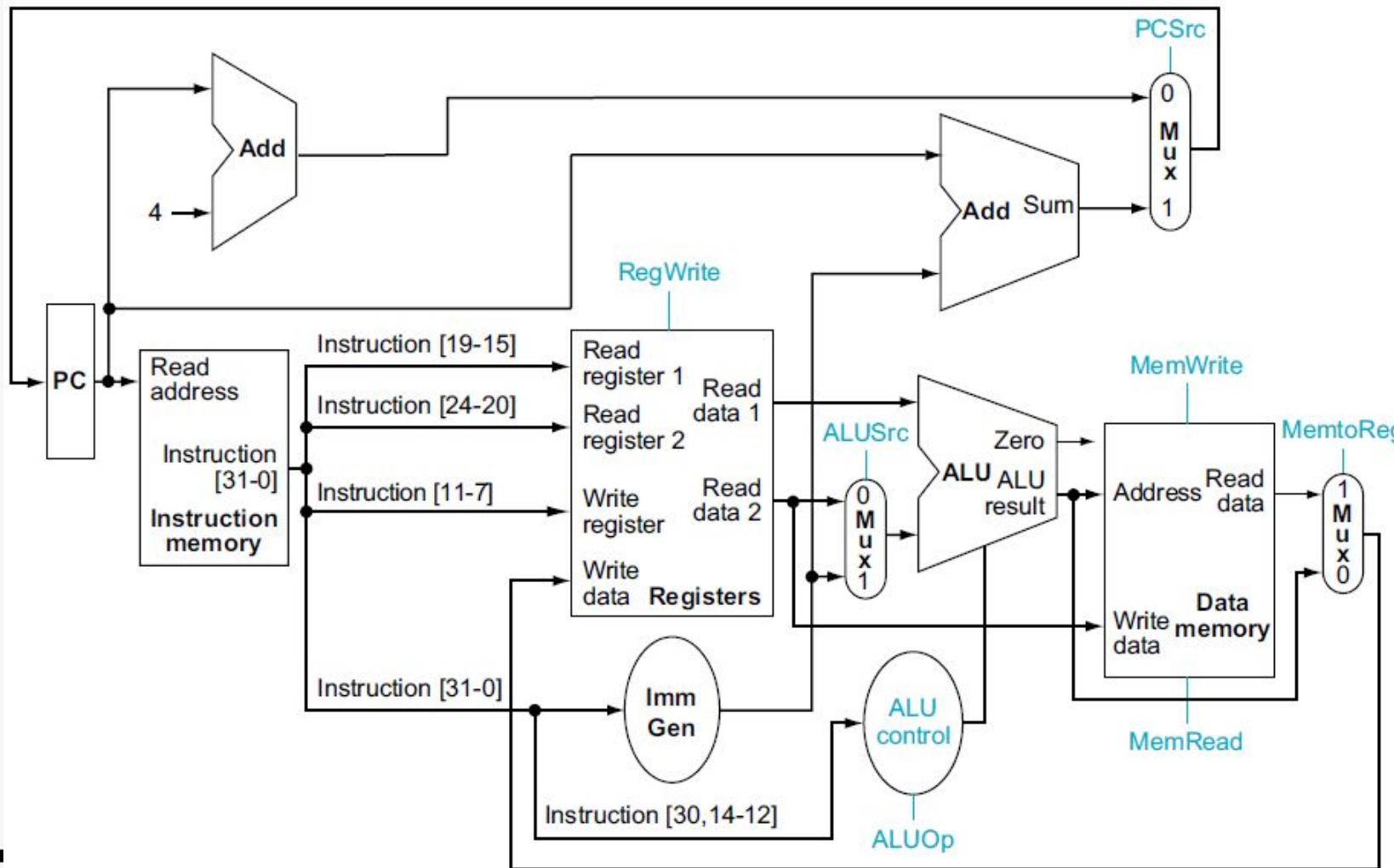
		Immediate Output Bit by Bit																																	
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Instruction	Format	Immediate Input Bit by Bit																																	
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20		
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	"	"	"	0
Uncond. Jump	U	"	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	"	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	0	0	0	0	0	0	0	0	0	0	0	"		
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	3		

		Immediate Output Bit by Bit																																		
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Instruction	Format	Immediate Input Bit by Bit																																		
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20			
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7	
Cond. Branch	SB	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	0	"
Uncond. Jump	UJ	"	"	"	"	"	"	"	"	"	"	"	"	i19	i18	i17	i16	i15	i14	i13	i12	i20	"	"	"	"	"	"	"	"	"	"	"	"		
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	"	"	"	"	"	"	"	"	"	0	0	0	0	0	0	0	0	0	0	0	"		
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	2	2	2	2	3	3	3	3	3	3			

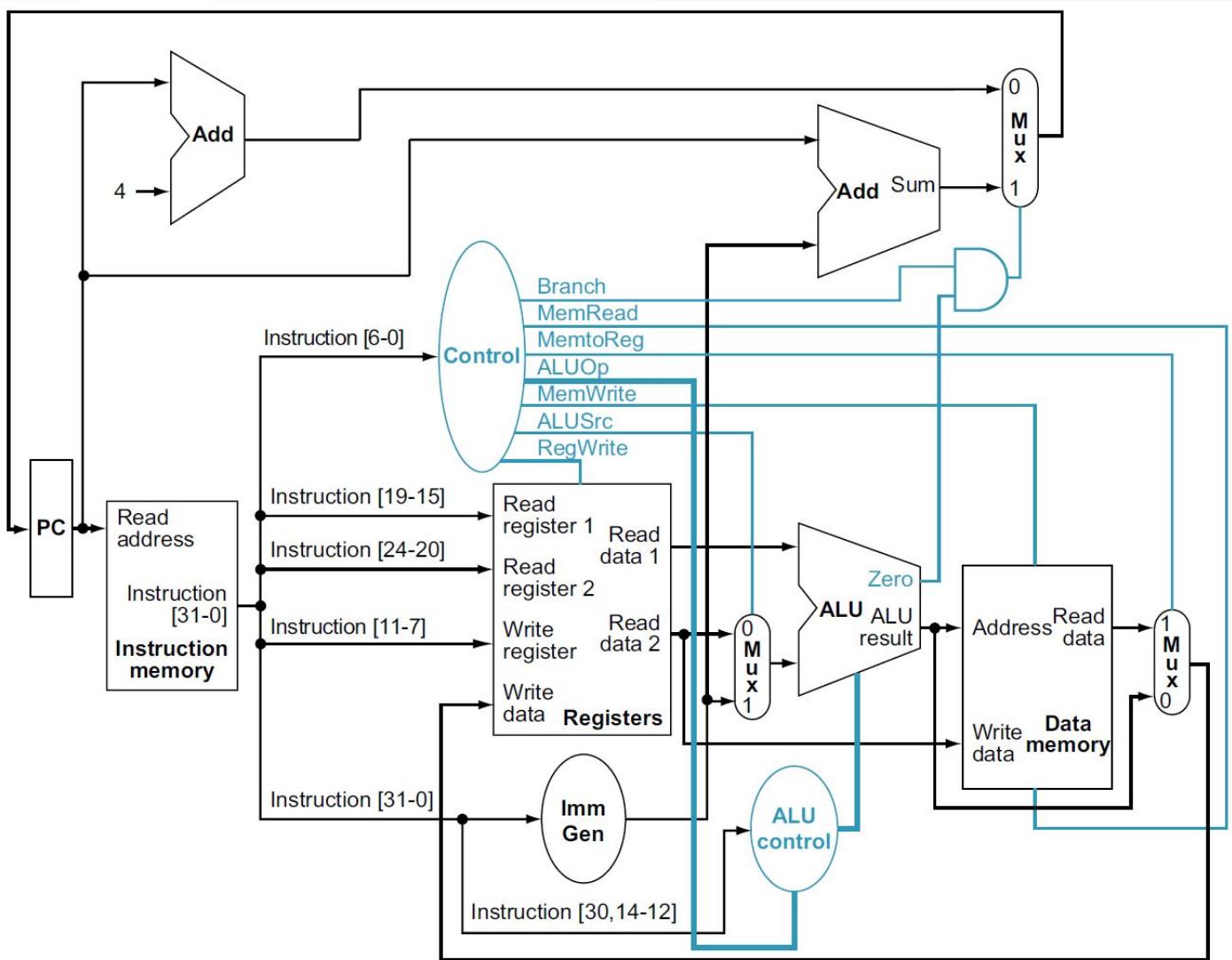
Entradas para imediato no caso o RISC-V utilizar os formatos U e S

Entradas para imediato utilizando os formatos UJ e SB no RISC-V

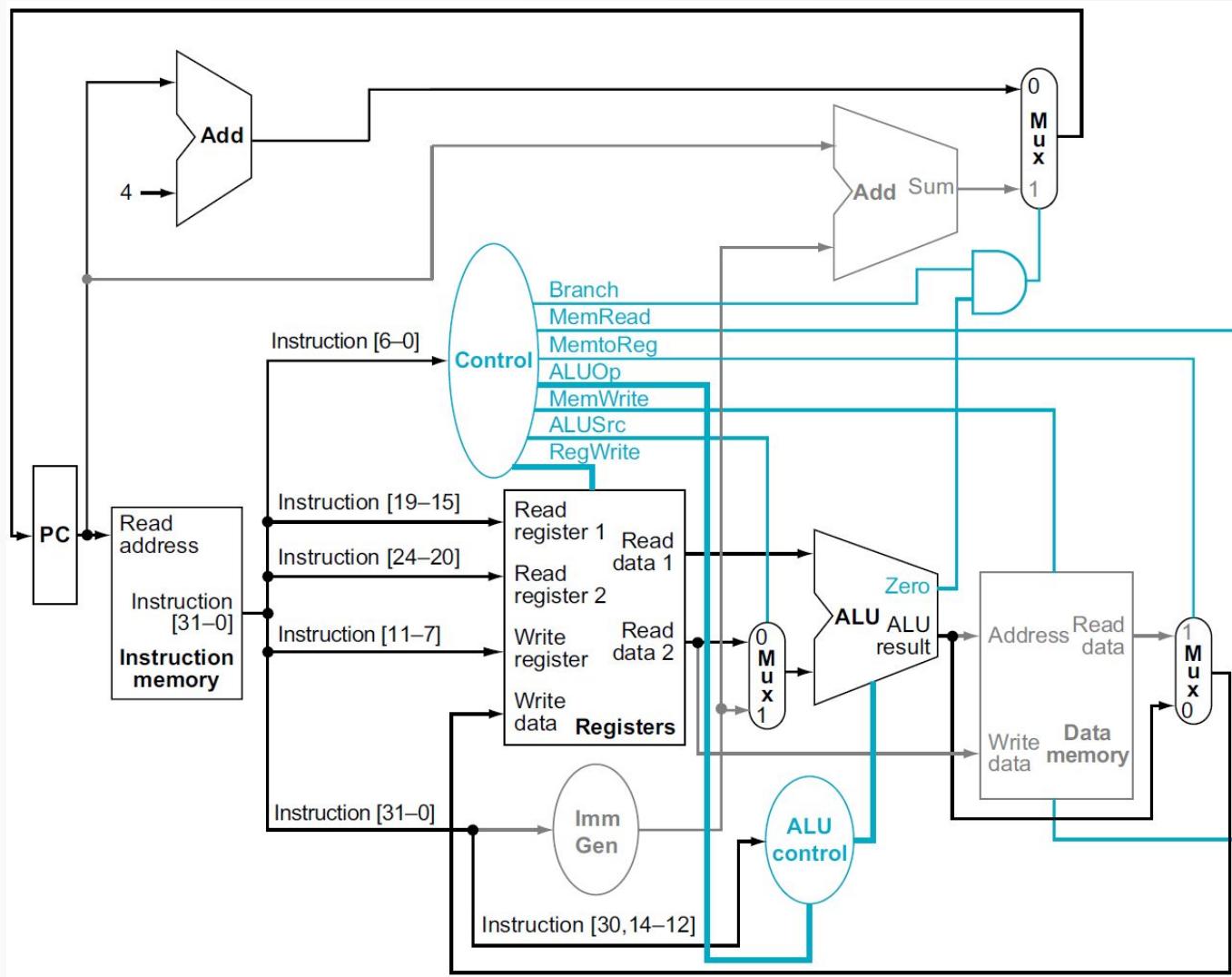
# Caminho de dados com sinais e mux identificados



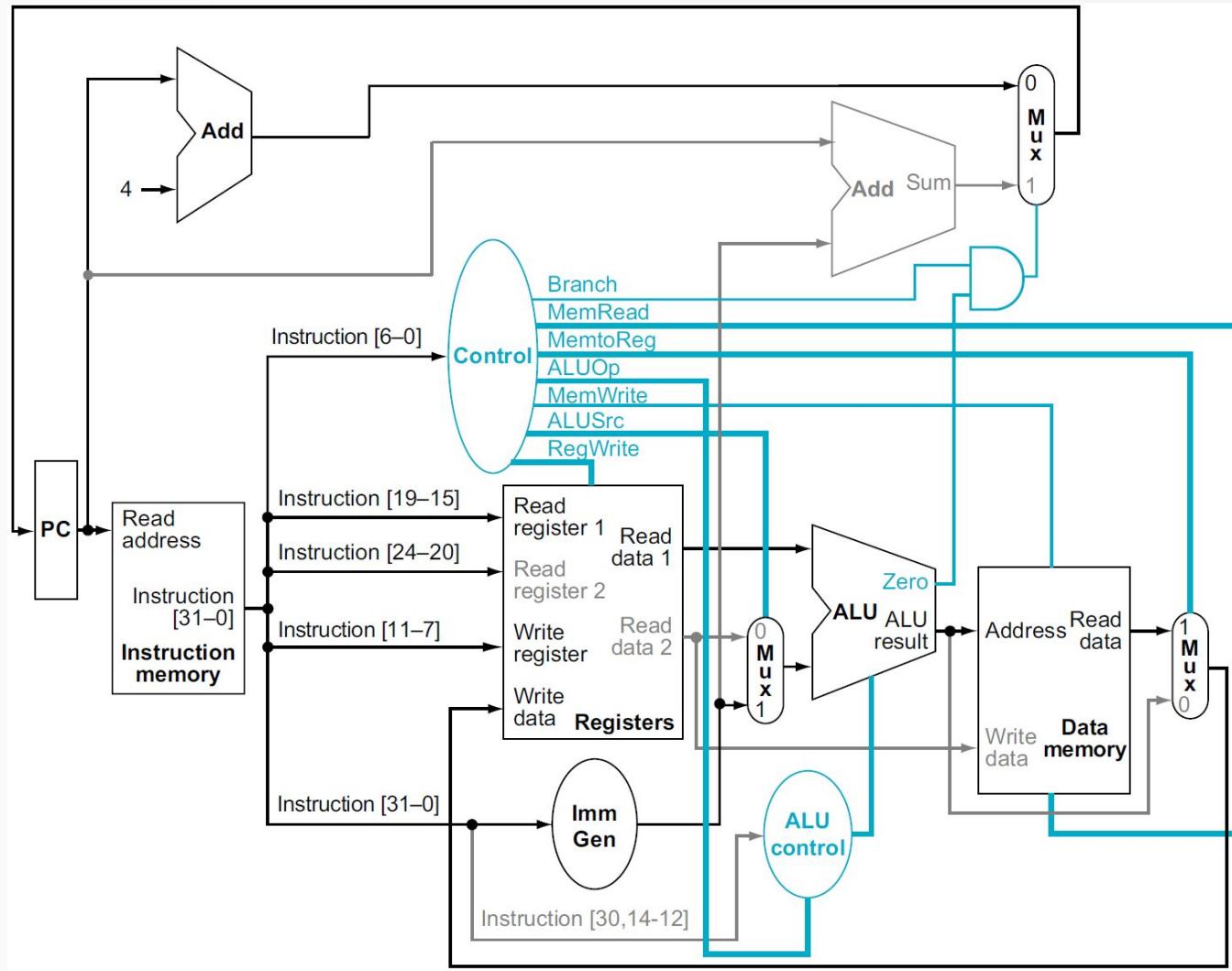
# Caminho de dados com controle



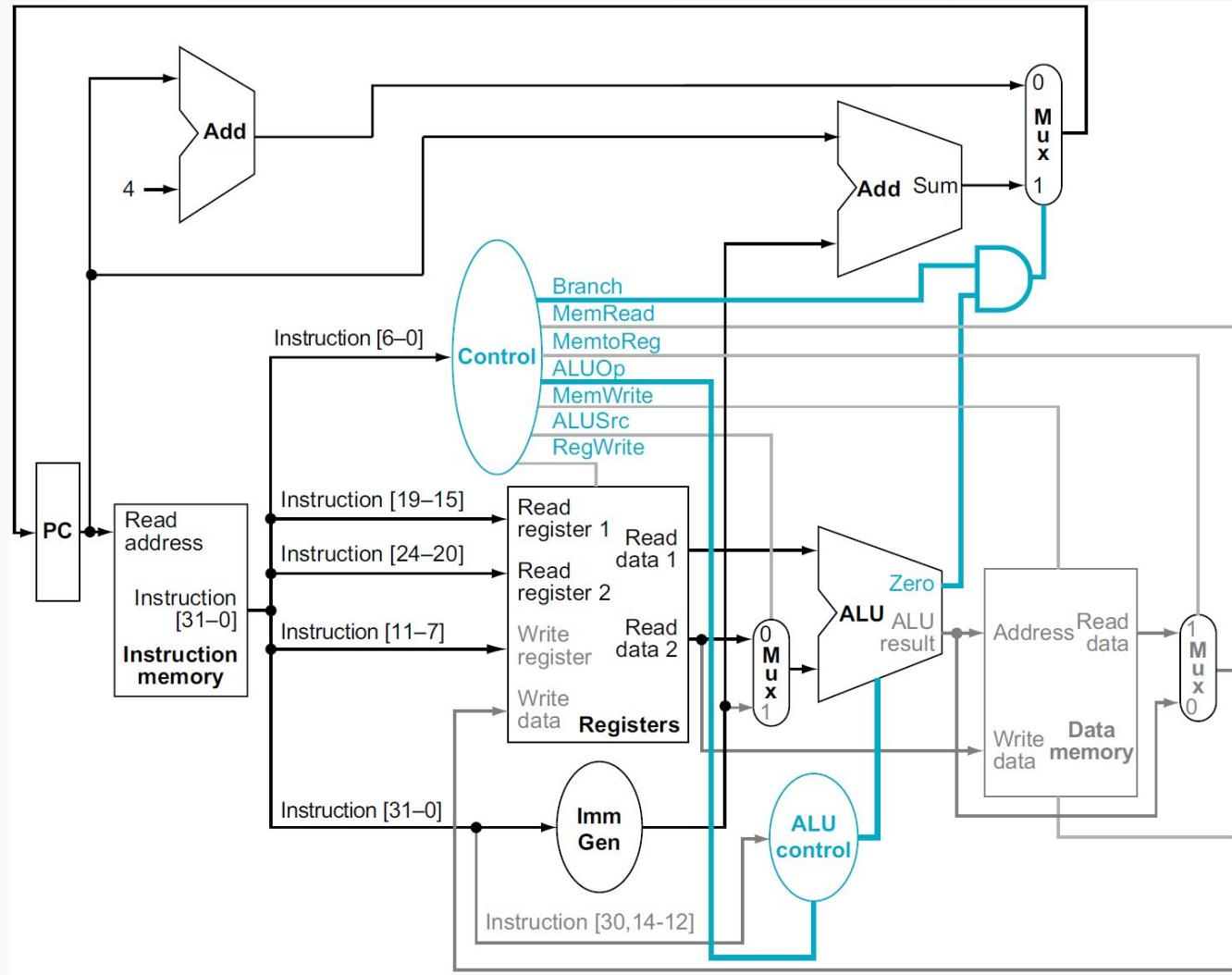
# Instruções Tipo R



# A instrução load



# A instrução BEQ



# Problemas de desempenho

- O maior atraso determina o período de clock
  - Caminho crítico: instrução load
  - (1) Memória de instrução → (2) banco de registradores → (3) ALU → (4) memória de dados → (5) banco de registradores
- Não é viável variar o período para instruções diferentes
- Viola o princípio de projeto
  - Tornando o caso comum rápido
- Vamos melhorar o desempenho através do pipelining

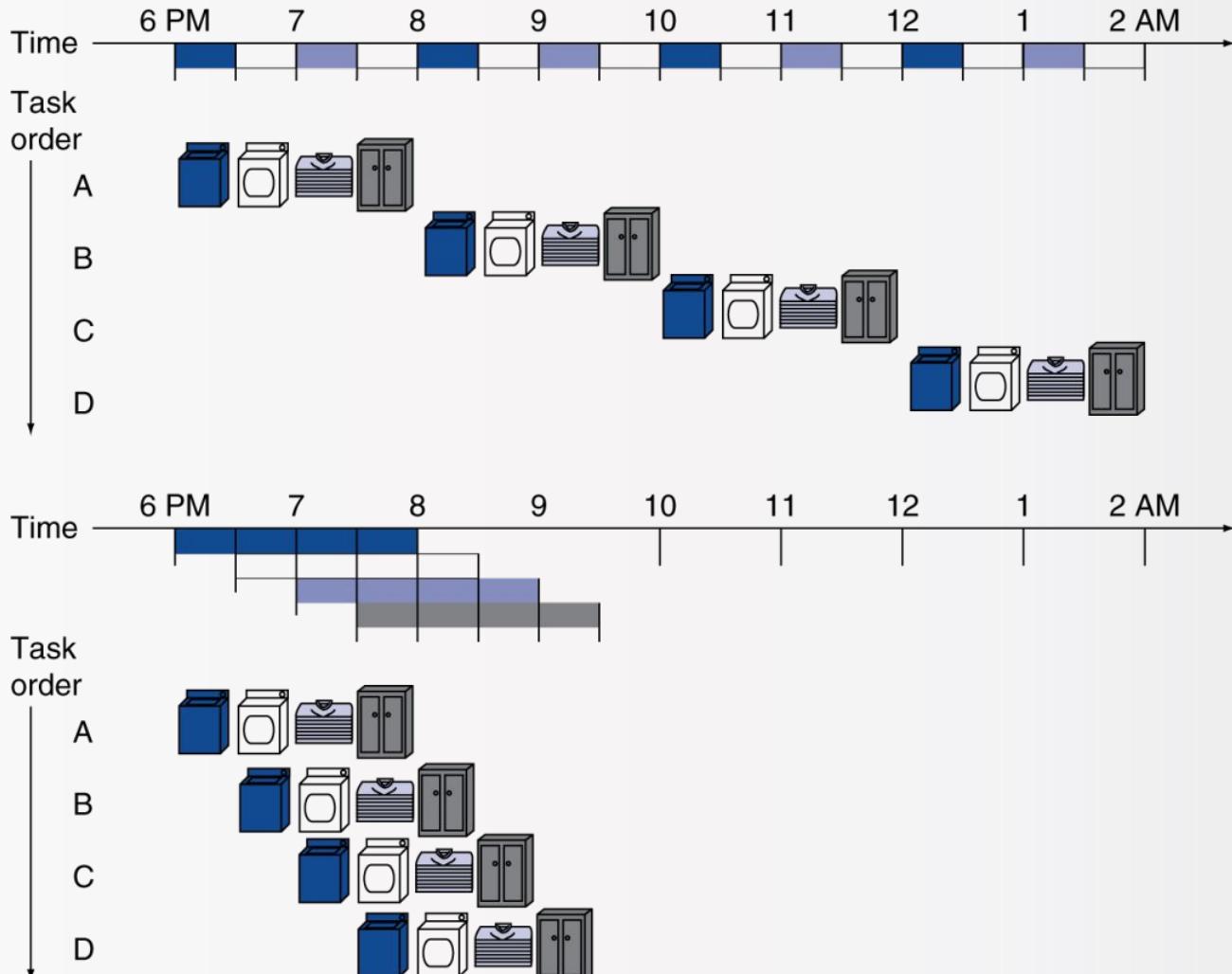
# Visão Geral de Pipeline

Referência: Capítulo 4, Seção 4.6.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Analogia

- Lavandaria com pipeline
  - Paralelismo
  - Melhor desempenho
- Tempo sequencial
  - 2 horas por lavagem
- Em paralelo
  - 4 lavagens
    - 3,5 horas
    - Speedup =  $8 / 3,5 = 2,3$



# Pipeline do RISC-V

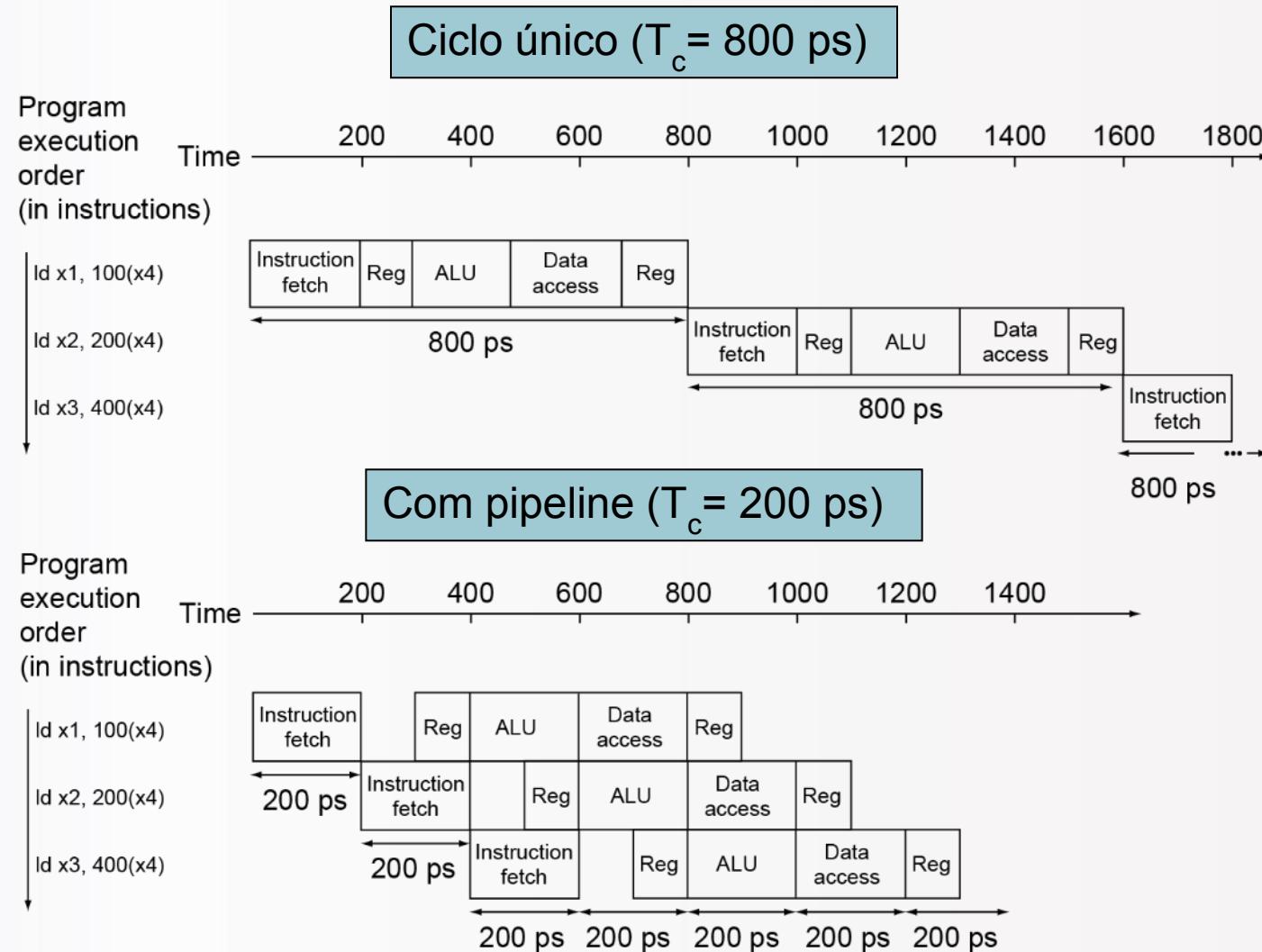
- Cinco etapas
  1. *Instruction Fetch (IF)*
    - Busca da instrução da memória
  2. *Instruction Decode (ID)*
    - Decodificação da instrução e leitura de registradores
  3. *Execution (EX)*
    - Execução da operação ou cálculo do endereço
  4. *Memory Access (MEM)*
    - Acesso à memória de dados
  5. *Write Back (WB)*
    - Gravação do resultado no registrador

# Desempenho do pipeline

- Suponha que o tempo para os estágios seja
  - 100 ps para leitura/escrita de registradores
  - 200 ps para outras etapas

Instrução	Busca (IF)	Leitura de registradores	Operação de ALU	Acesso à memória	Escrita de registradores	Total
ld	200 ps	100 ps	200 ps	200 ps	100 ps	800ps
sd	200 ps	100 ps	200 ps	200 ps		700ps
Formato R	200 ps	100 ps	200 ps		100 ps	600ps
beq	200 ps	100 ps	200 ps			500ps

# Desempenho do pipeline



# Speedup do pipeline

- Se todos os estágios estiverem equilibrados

$$Tempo\ entre\ instruções_{com\ pipeline} = \frac{Tempo\ entre\ instruções_{sem\ pipeline}}{Número\ de\ estágios}$$

- Se não estiver equilibrado
  - A aceleração é menor
- Aceleração acontece devido ao aumento de vazão
  - Latência (tempo para cada instrução) não diminui

# Pipeline e projeto do ISA

- RISC-V ISA projetado para pipeline
  - Todas as instruções são de 32 bits
    - Mais fácil de buscar e decodificar em um ciclo
    - x86
      - Instruções de 1 a 17 bytes
  - Poucos formatos de instrução regulares
    - Pode decodificar e ler registros em um único estágio
  - Endereçamento do Load/store
    - Pode calcular o endereço no 3º estágio
    - Acesso à memória no 4º estágio

# Dependências

- Situações que impedem o início da próxima instrução no próximo ciclo
- Dependências estruturais
  - Um recurso necessário está ocupado
- Dependências de dados
  - Precisa esperar pela instrução anterior para completar sua leitura / gravação de dados
- Dependências de controle
  - A decisão sobre a ação de controle depende da instrução anterior

# Dependências estruturais

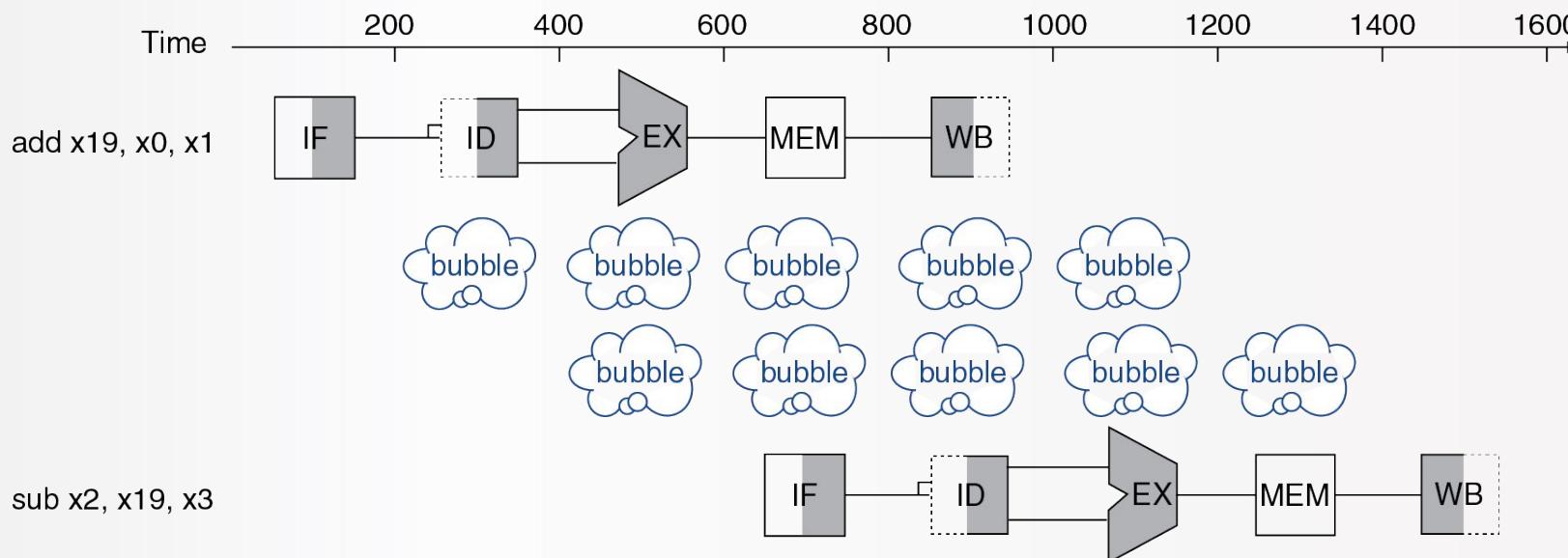
- Conflito para uso de um recurso
- No pipeline RISC-V com memória unificada para dados e instruções
  - As instruções load/store acessam memória de dados
  - A busca de instrução acessa a memória de instruções
    - Impossível fazer em paralelo, causando uma “bolha” no pipeline
- Para evitar dependências estruturais
  - Caminhos de dados com memórias separadas
    - Dados e instruções
  - Ou caches de dados e instruções separados

# Dependências de dados

- Uma instrução depende da escrita do resultado de uma instrução anterior

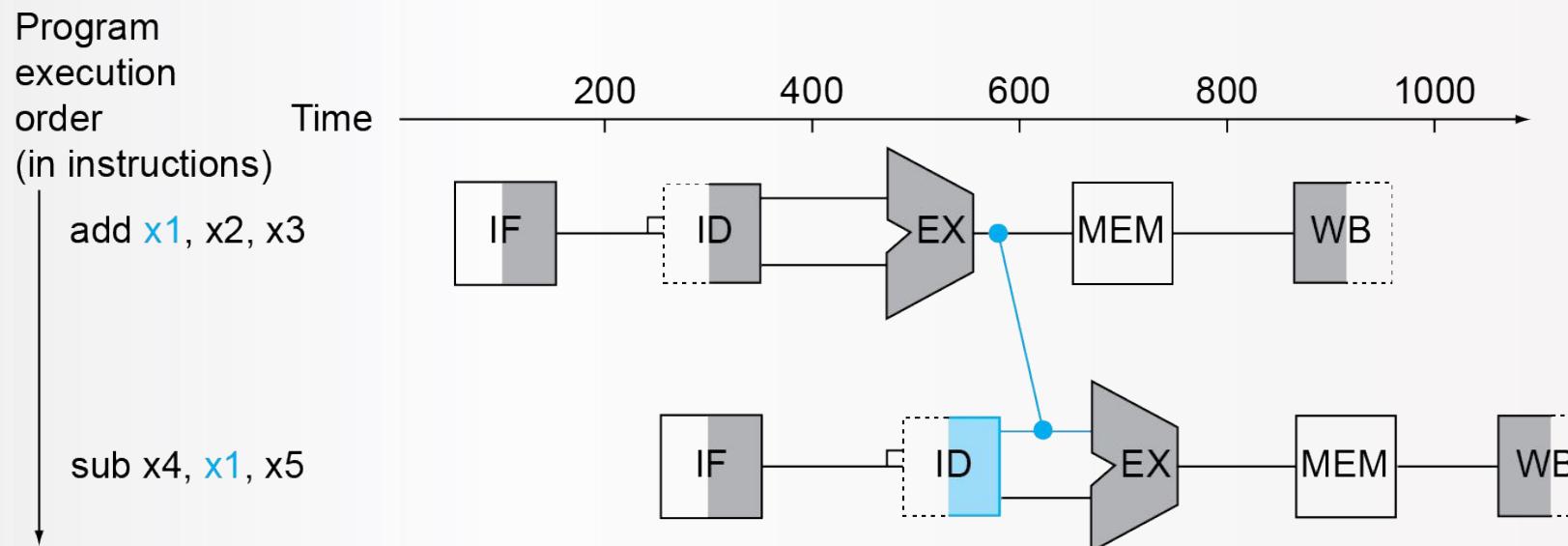
add x19, x0, x1

sub x2, x19, x3



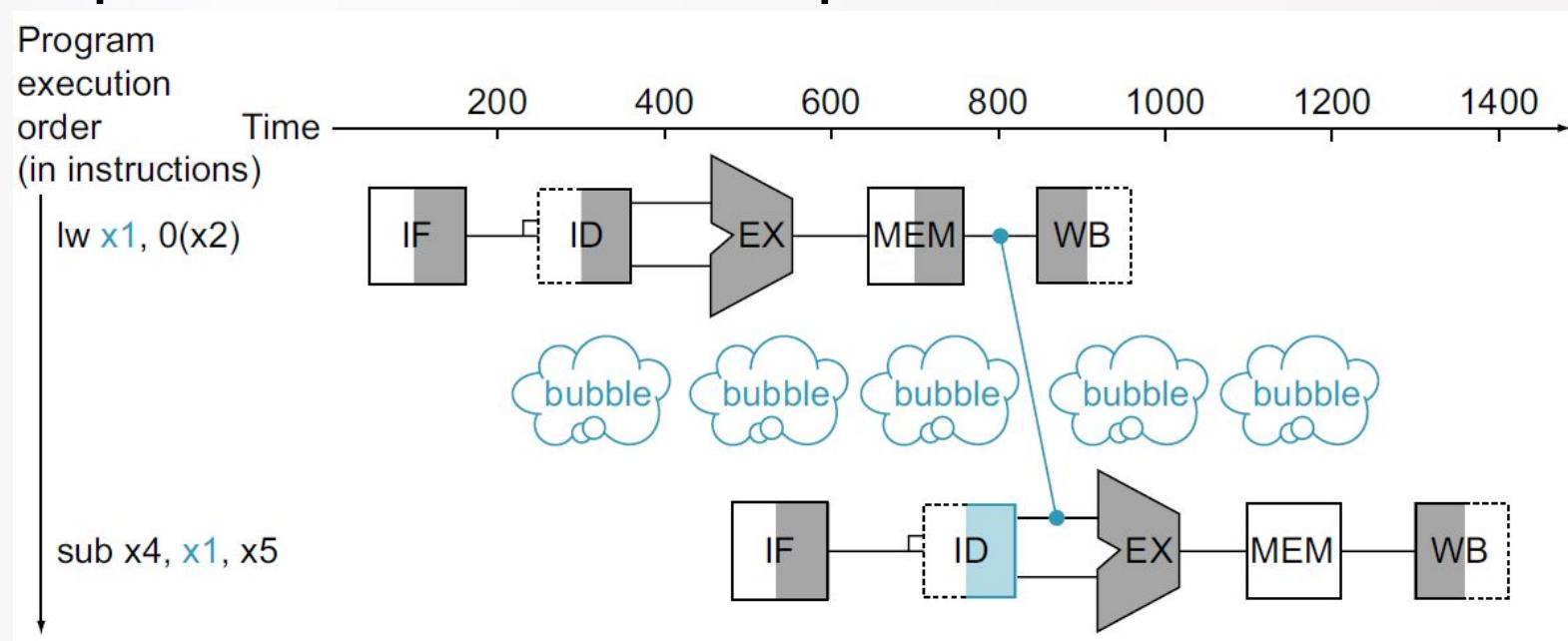
# Encaminhamento (*Forwarding* ou *Bypassing*)

- Utilizar o resultado assim que estiver disponível
  - Não esperar que seja armazenado em um registrador
  - Inclusão de “atalhos” no caminho de dados



# Dependências de dados com load

- Nem sempre é possível evitar dependências utilizando encaminhamento
  - Se o valor não for calculado quando necessário
  - Não é possível voltar no tempo!



# Como evitar paradas no pipeline

- Reordenar o código para evitar o uso do resultado de um load na próxima instrução
- Código em C:  $a = b + e; c = b + f;$

'parada

'parada

```
lw      x1, 0(x31)
lw      x2, 8(x31)
addx3, x1, x2
sw      x3, 24(x31)
lw      x4, 16(x31)
addx5, x1, x4
sw      x5, 32(x31)
```

13 ciclos

```
lw      x1, 0(x31)
lw      x2, 8(x31)
lw      x4, 16(x31)
addx3, x1, x2
sw      x3, 24(x31)
addx5, x1, x4
sw      x5, 32(x31)
```

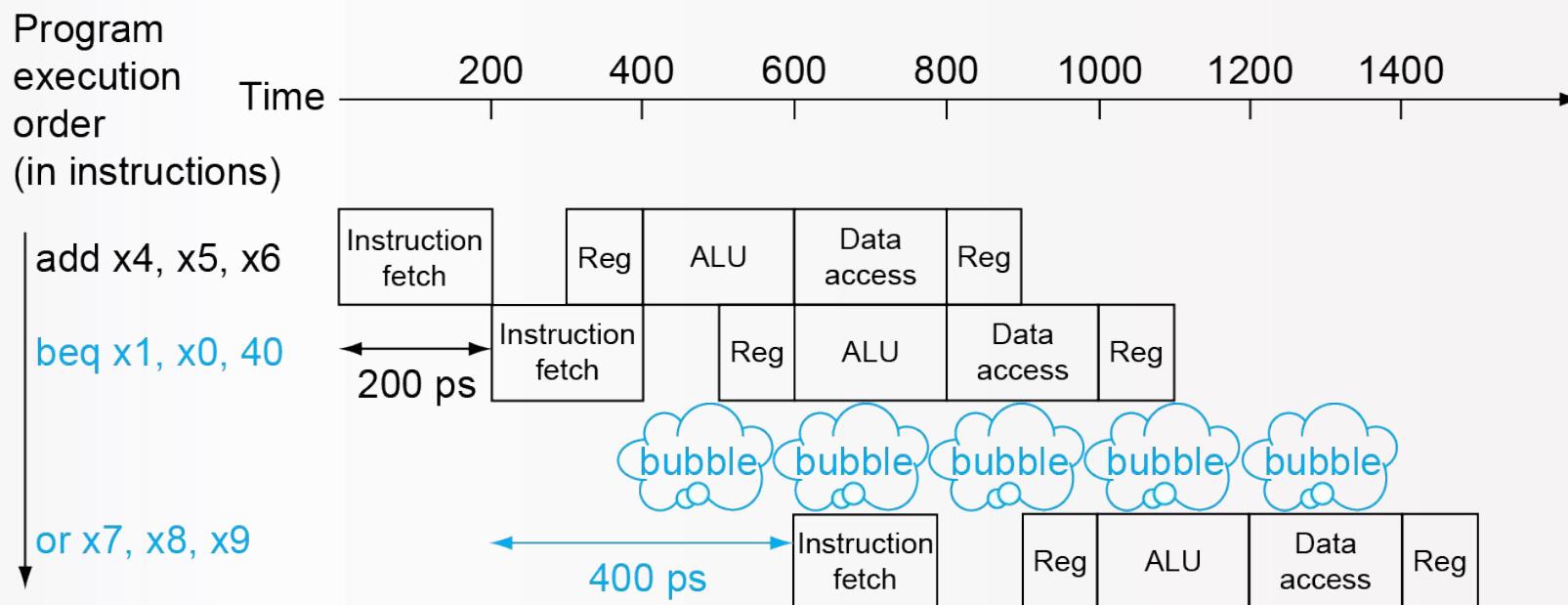
11 ciclos

# Dependências de controle

- Desvio condicional determina o fluxo de controle
  - Buscar a próxima instrução depende do resultado do desvio condicional
  - O pipeline nem sempre consegue obter a instrução correta
    - Ainda trabalhando no estágio ID do desvio condicional
- No pipeline RISC-V
  - Precisa comparar os registradores e calcular o destino no início do pipeline
  - Adicionar hardware para fazer isso no estágio ID

# Parada nos desvios condicionais

- Esperar até que o resultado do desvio condicional seja determinado antes de buscar a próxima instrução



# Predição de desvios

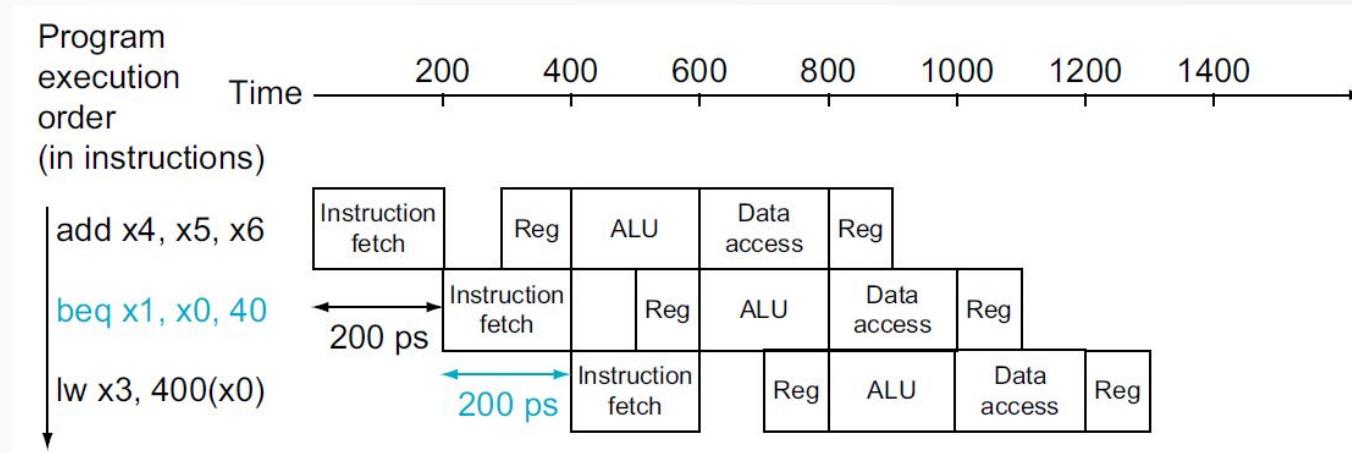
- Pipelines mais profundos não podem determinar o resultado do desvio condicional nos estágios iniciais
  - Parar o pipeline tem grande impacto de desempenho
- Previsão do resultado do desvio
  - Só para se a previsão estiver errada
- No pipeline RISC-V
  - Pode prever desvios não tomados
  - Busca a instrução após o desvio, sem atraso

# Predição de desvios mais realista

- Predição de desvio estática
  - Utiliza comportamento típico de condicionais e repetições
  - Exemplo: loop e ramificações da instrução if
    - Predição de desvios para endereços anteriores como tomados
    - Predição de desvios para endereços posteriores como não tomados
- Predição de desvio dinâmica
  - Hardware mede o comportamento real do desvio
    - Registrar o histórico recente de cada desvio
  - Suposição que o comportamento futuro será semelhante
    - Caso a predição esteja errada
      - Parar, fazer busca novamente e atualizar o histórico

# Predição de desvios

- Exemplo de predição de desvio não tomado no RISC-V



# Resumo de pipeline

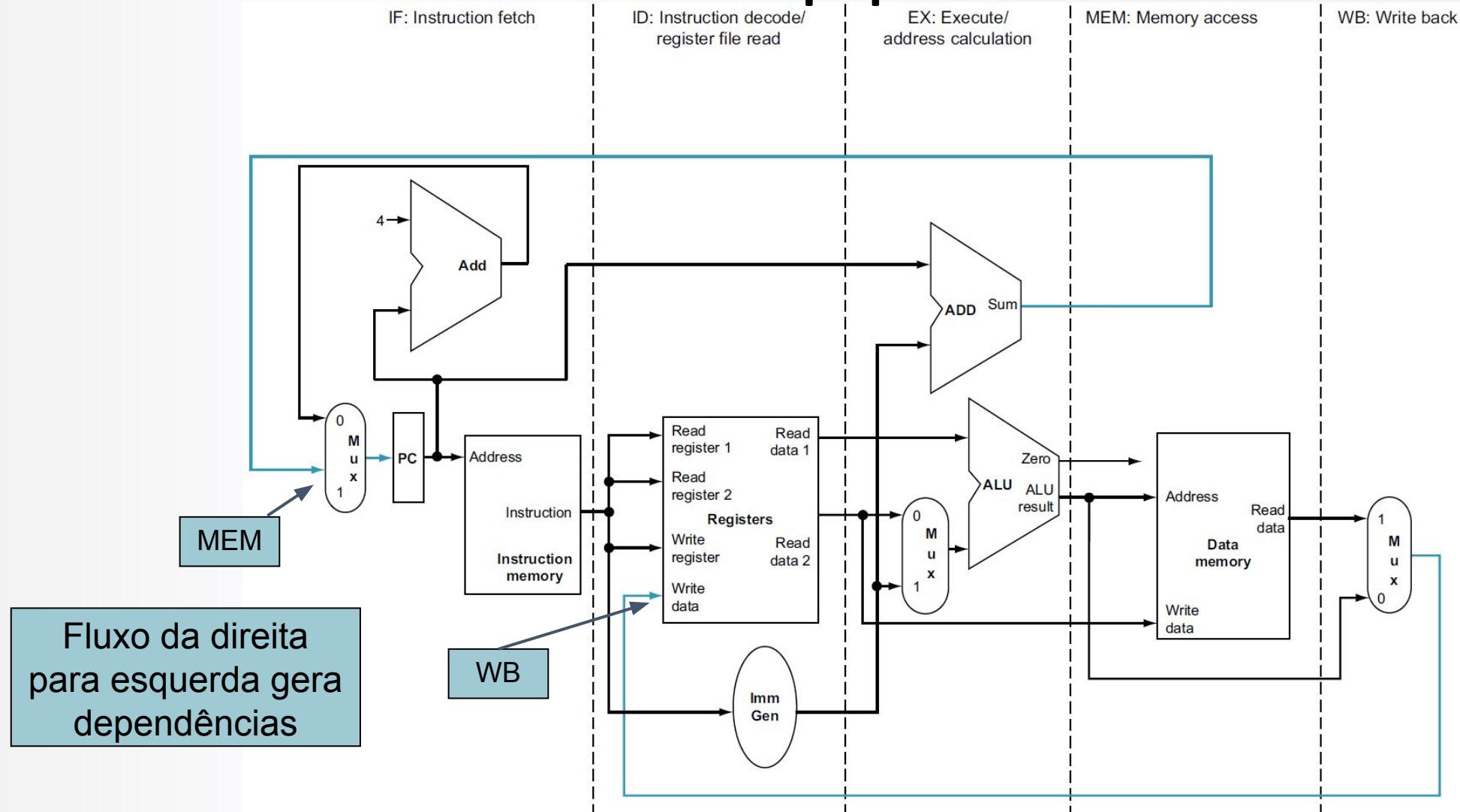
- O pipeline melhora o desempenho, aumentando a vazão de instruções
  - Executa várias instruções em paralelo
  - Cada instrução tem a mesma latência
- Sujeito a dependências
  - Estrutural, dados e controle
- O projeto do conjunto de instruções afeta a complexidade da implementação do pipeline

# Caminho de Dados com Pipeline e Controle

Referência: Capítulo 4, Seção 4.7.

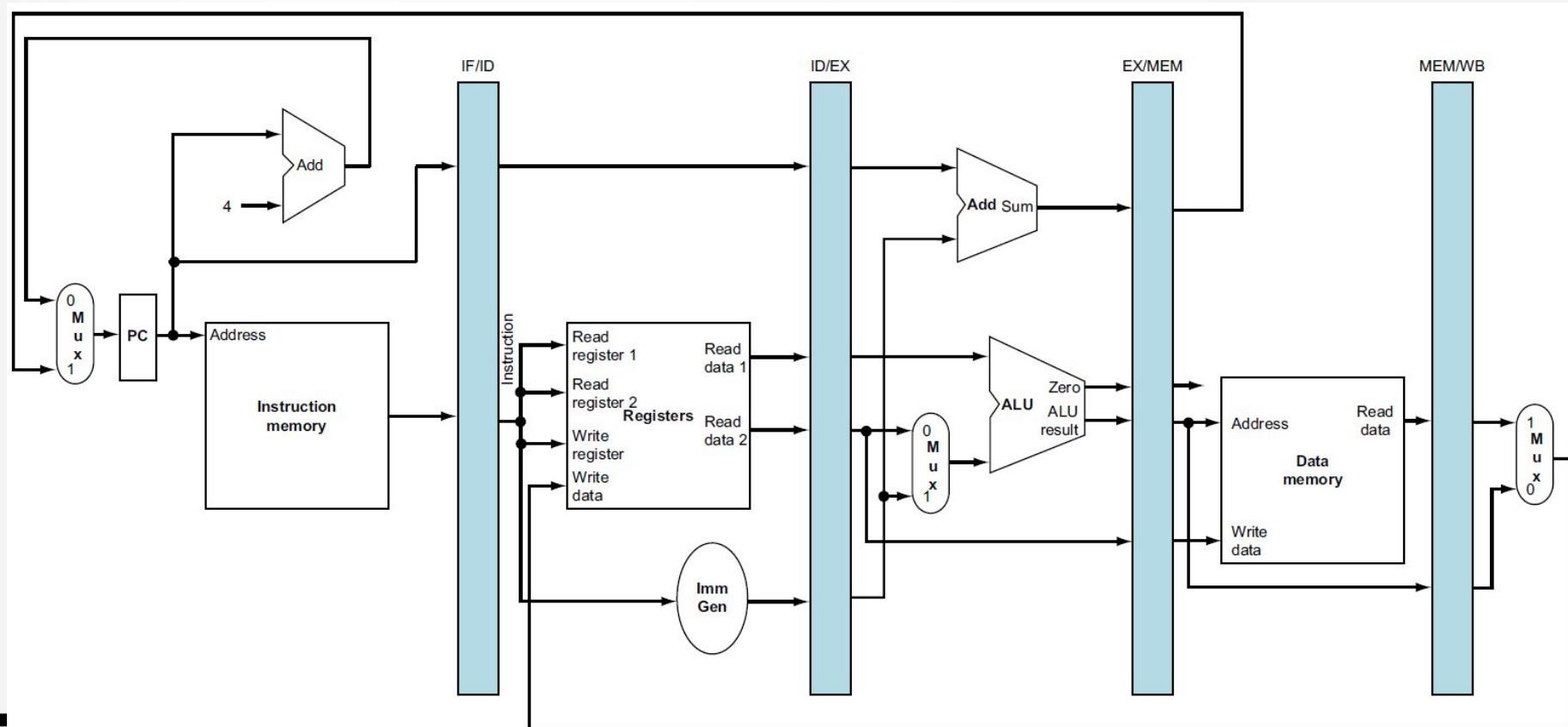
Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Caminho de dados com pipeline RISC-V



# Registradores de pipeline

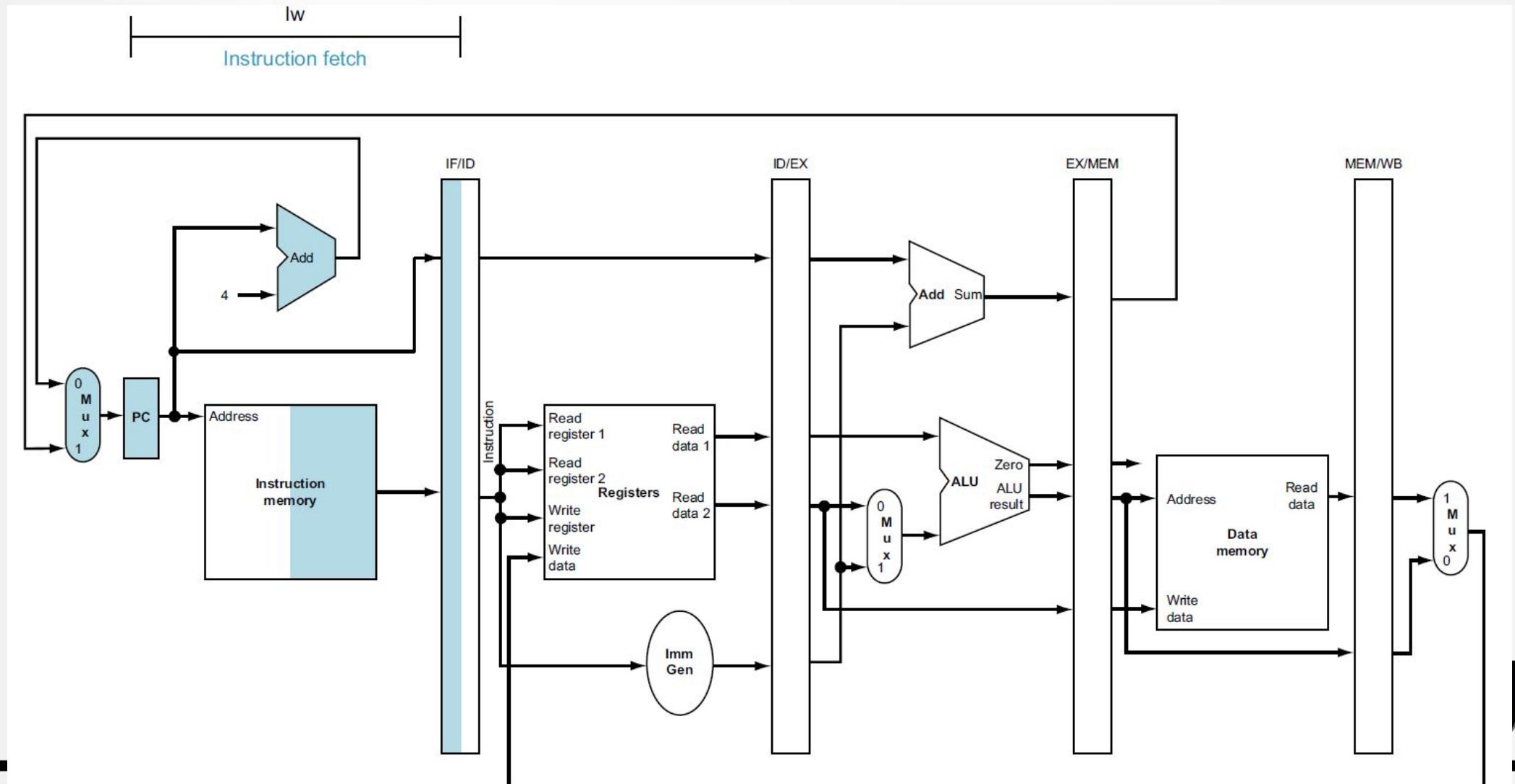
- Registradores entre os estágios são necessários
  - Para guardar informações produzidas no ciclo anterior



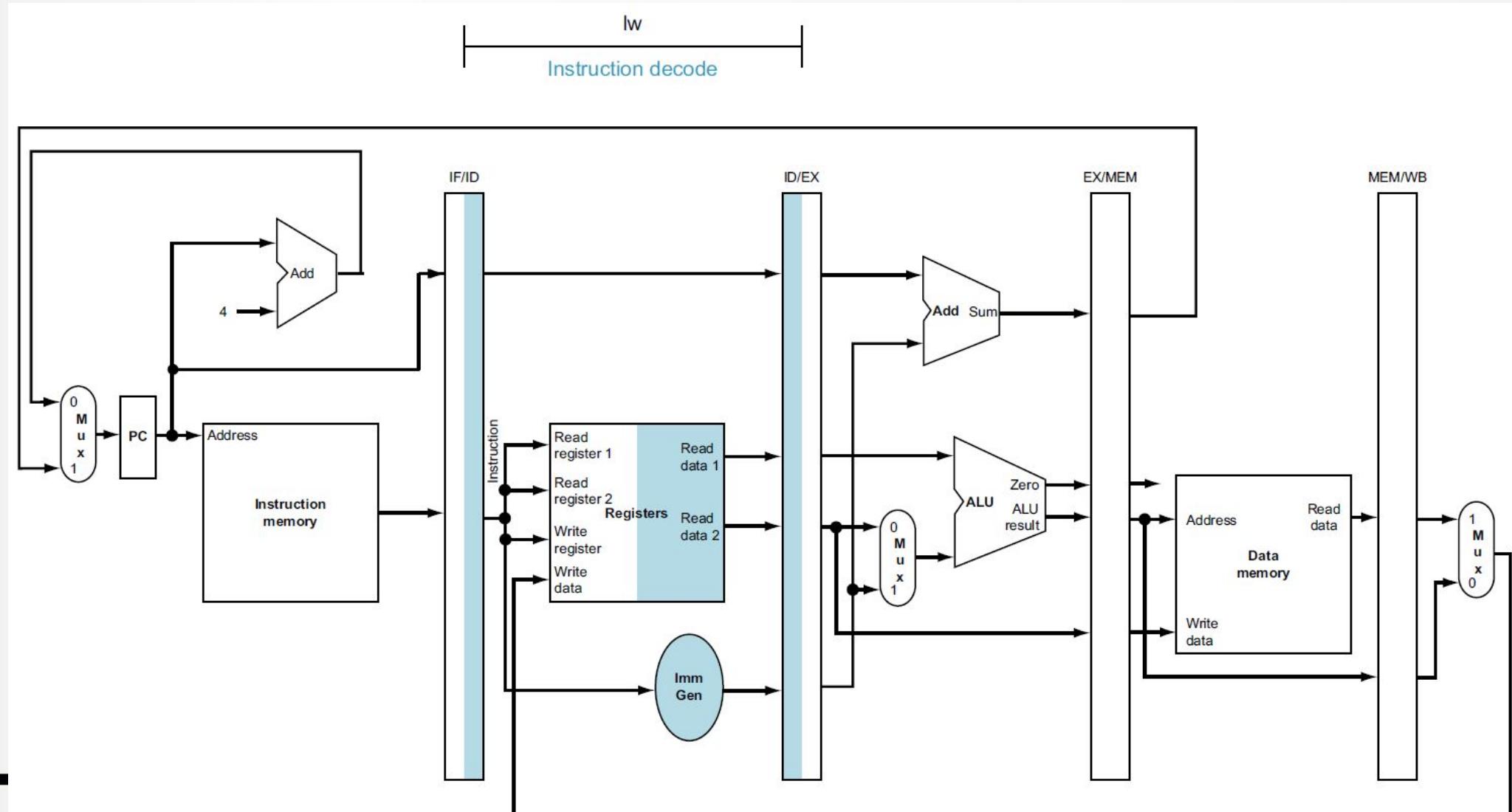
# Funcionamento do pipeline

- Fluxo de instruções ciclo a ciclo através do caminho de dados com pipeline
  - Diagrama de pipeline com ciclo de clock único
    - Mostra o uso do pipeline em um único ciclo
    - Destaca recursos usados
  - Diagrama de pipeline com múltiplos ciclos de clock
    - Gráfico de operação ao longo do tempo

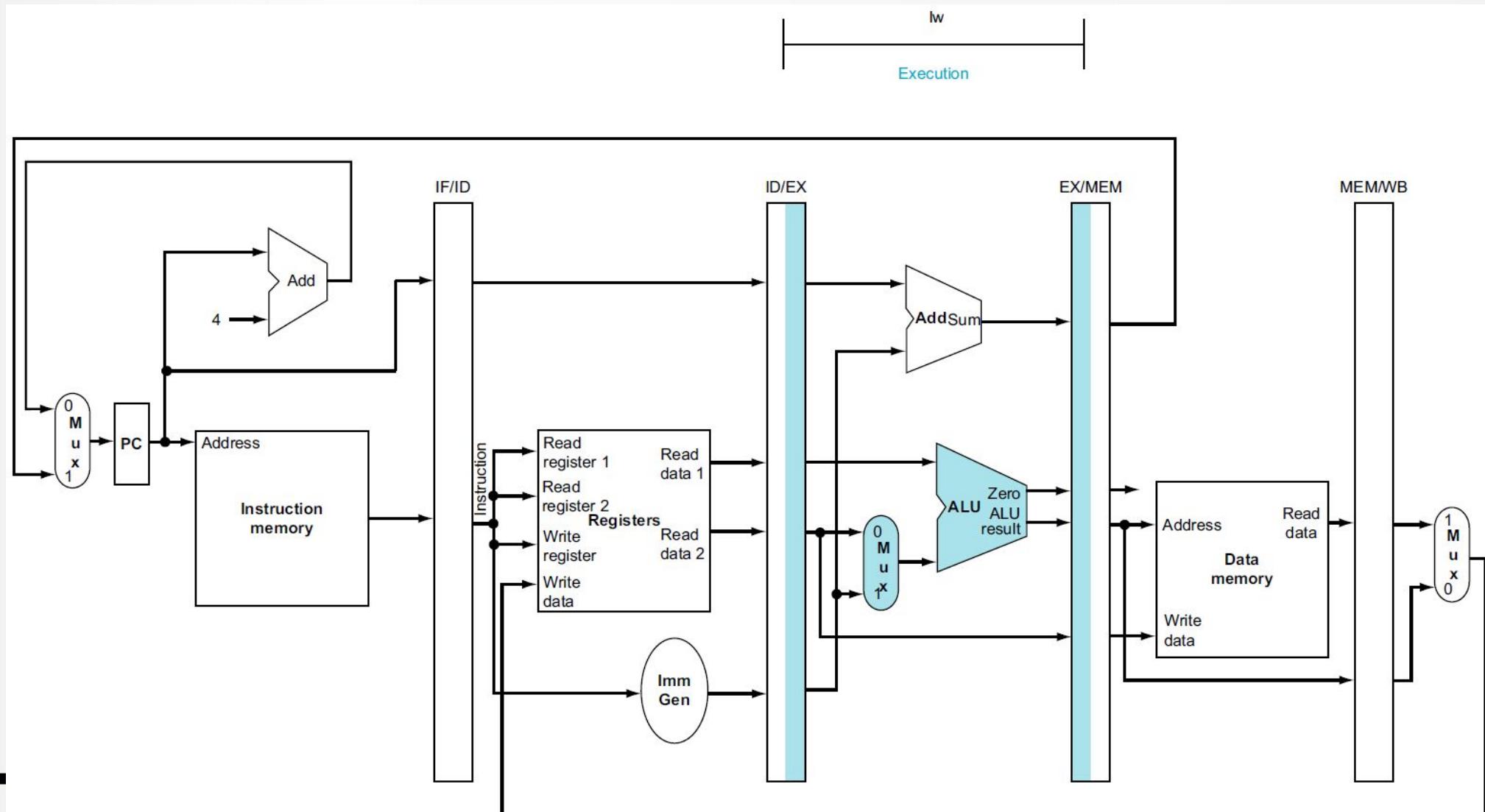
# Busca de instrução (IF) – Load/Store



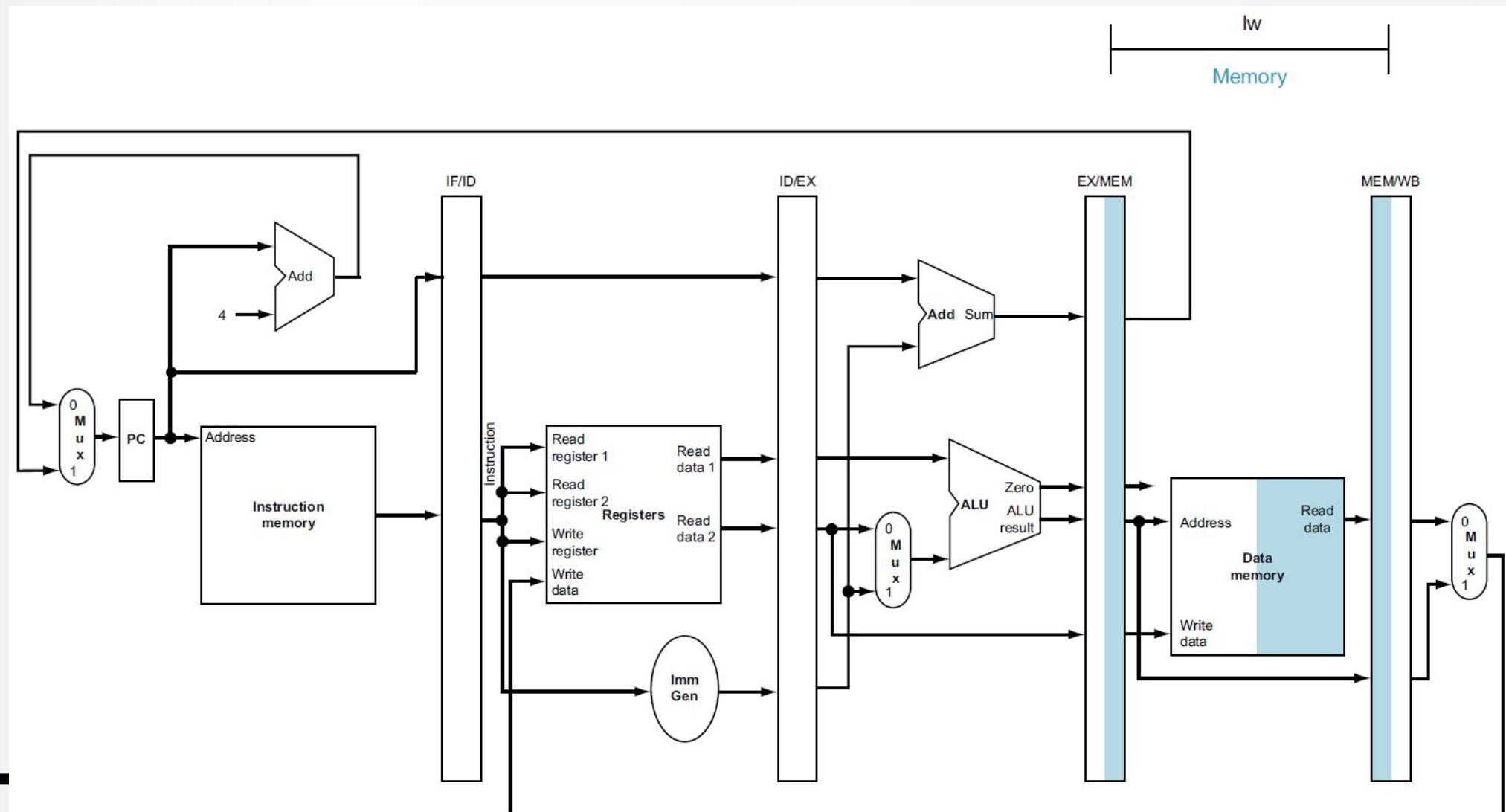
# Decodificação de instrução (ID) – Load/Store



# Execução (EX) – Load

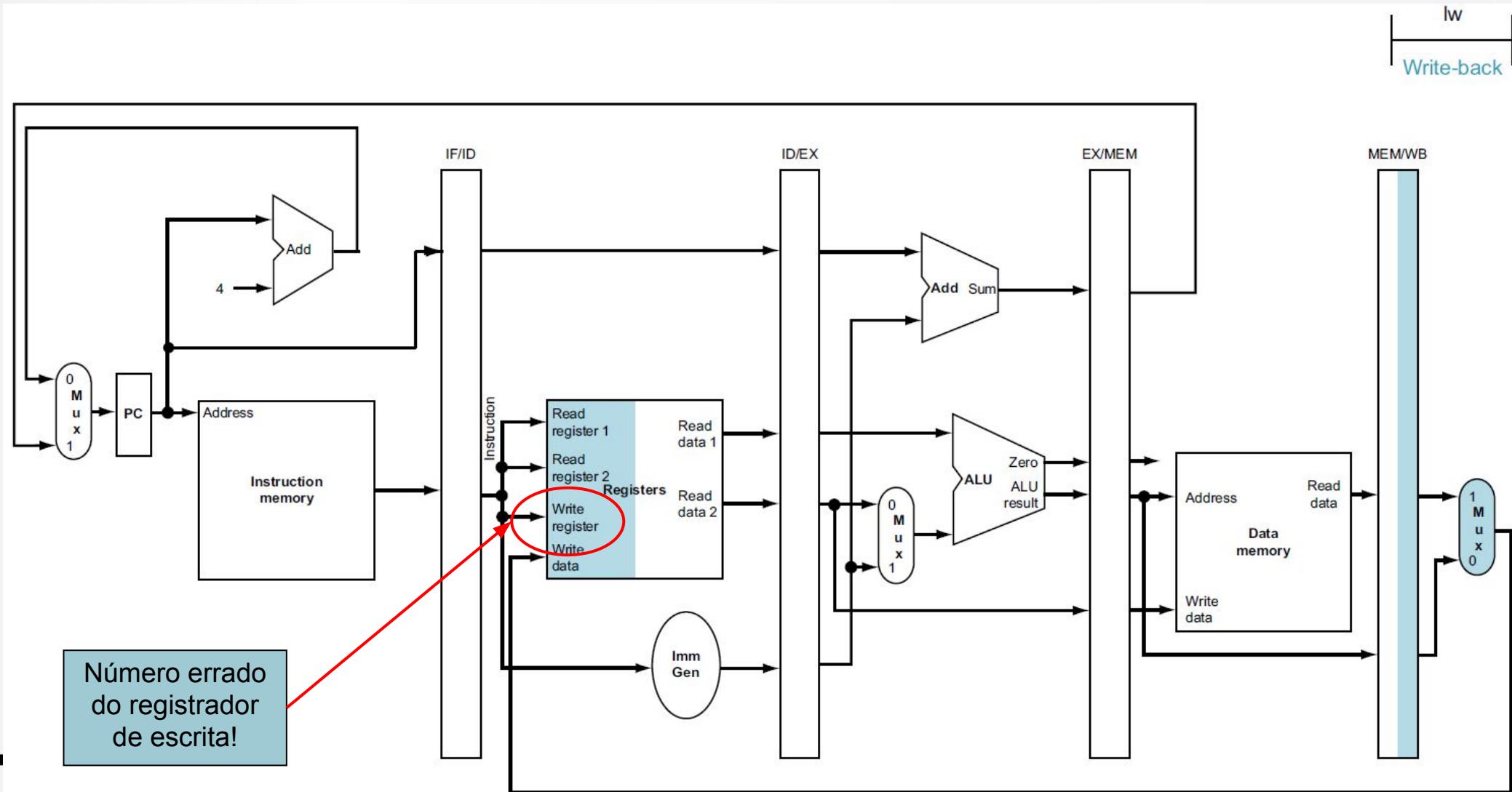


# Acesso à memória (MEM) – Load

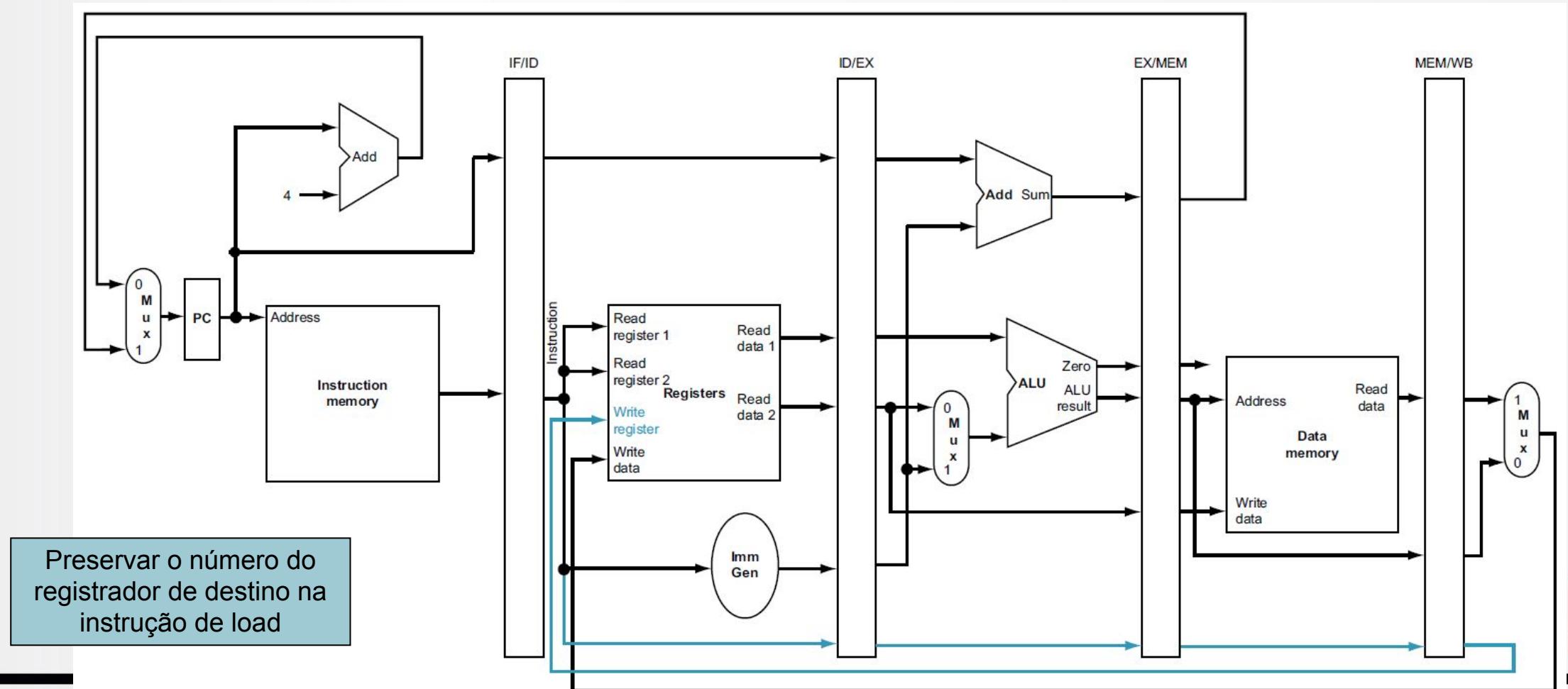


# Escrita de registradores (WB) – Load

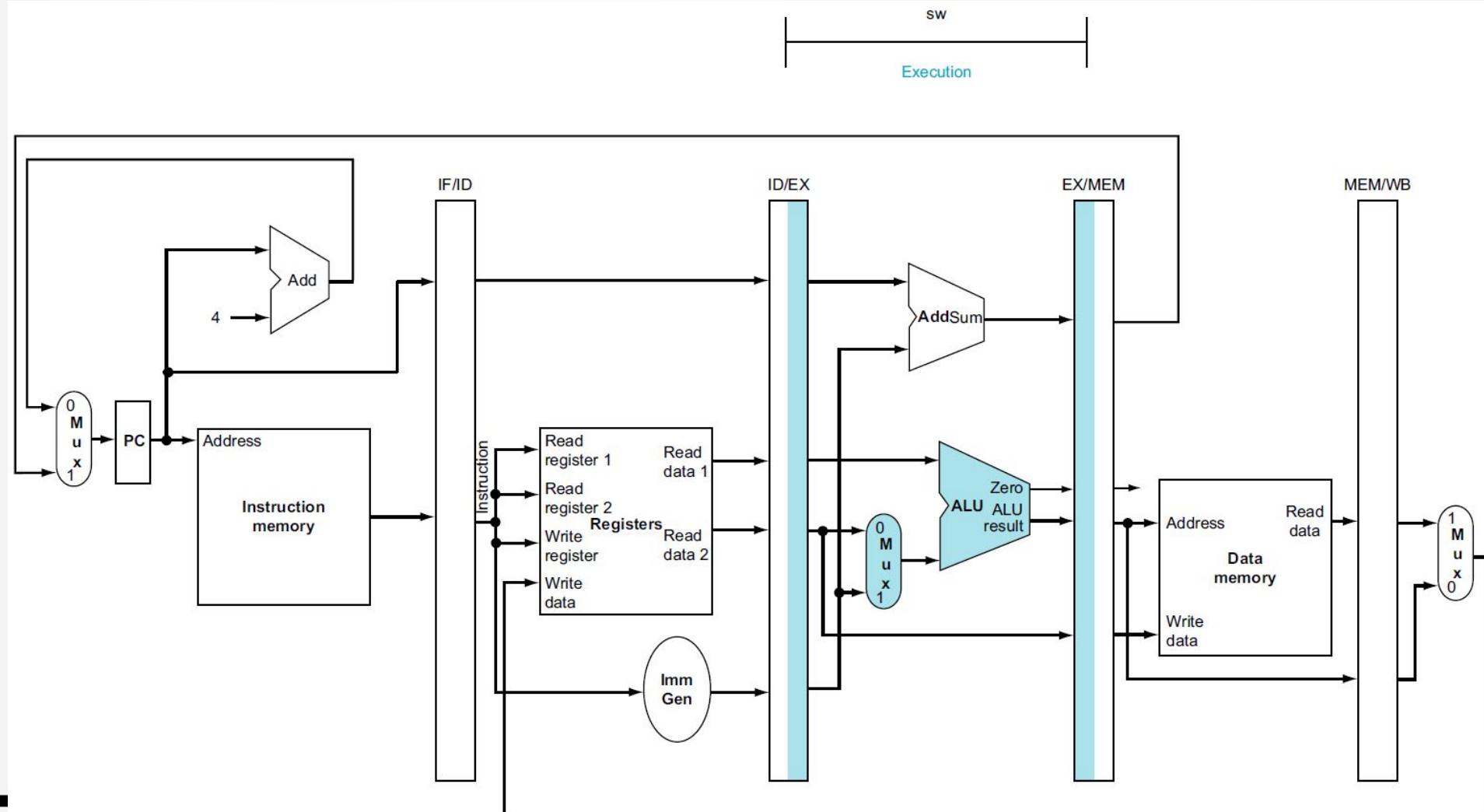
Iw  
Write-back



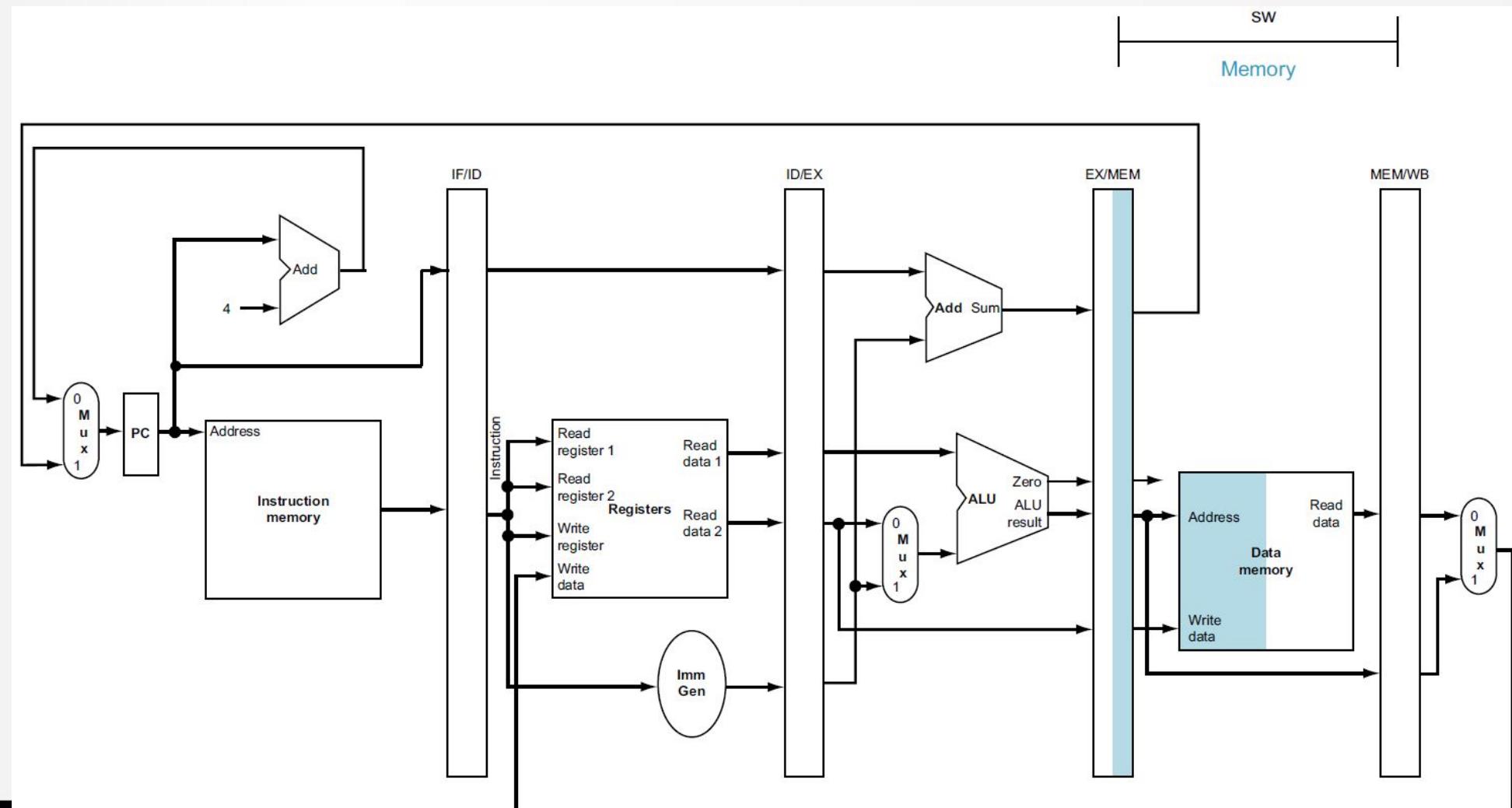
# Caminho de dados corrigido para o load



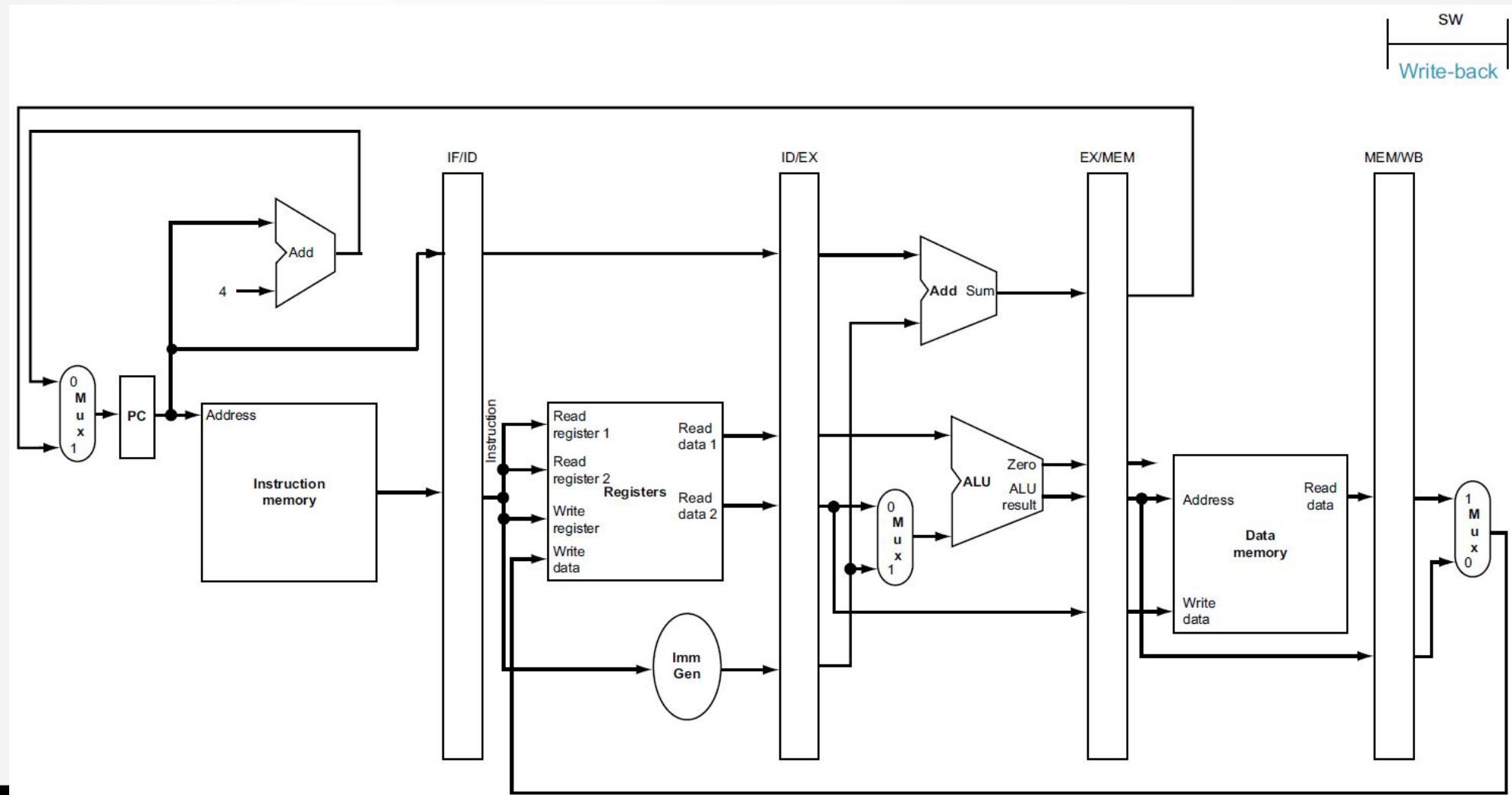
# Execução (EX) – Store



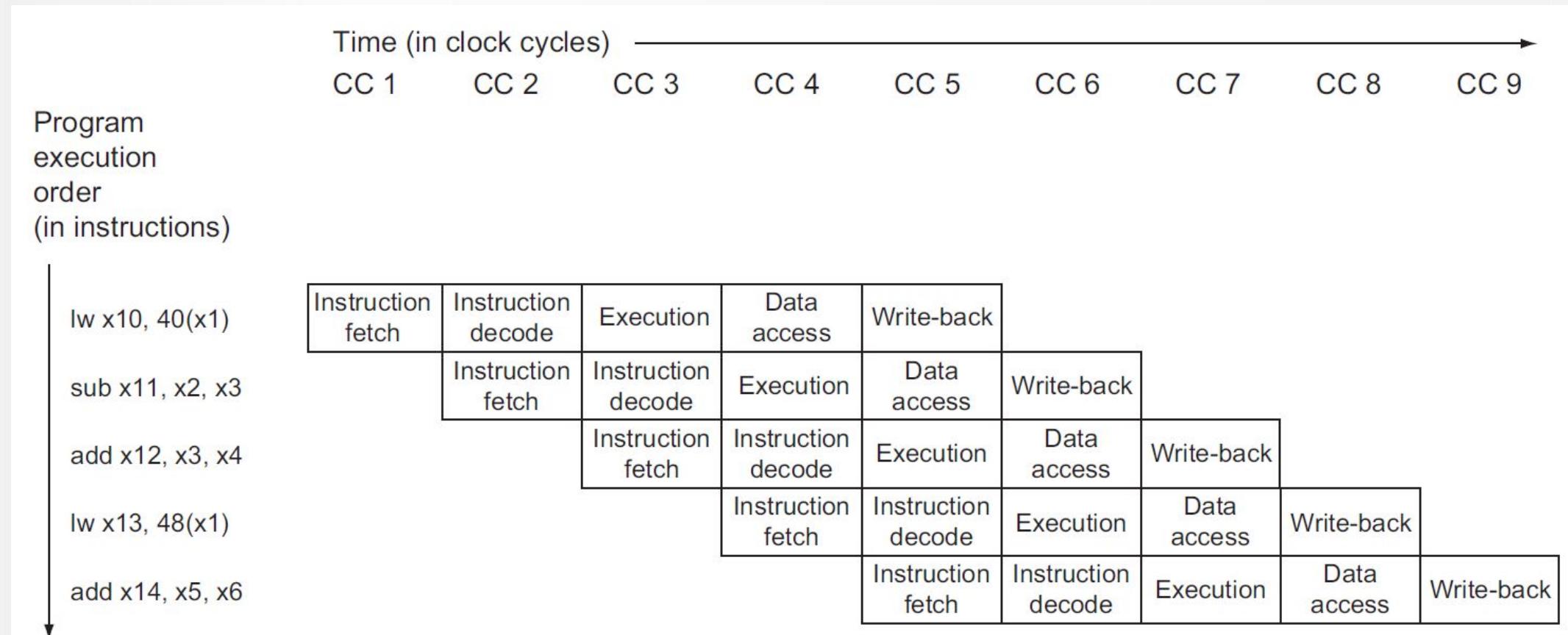
# Acesso à memória (MEM) – Store



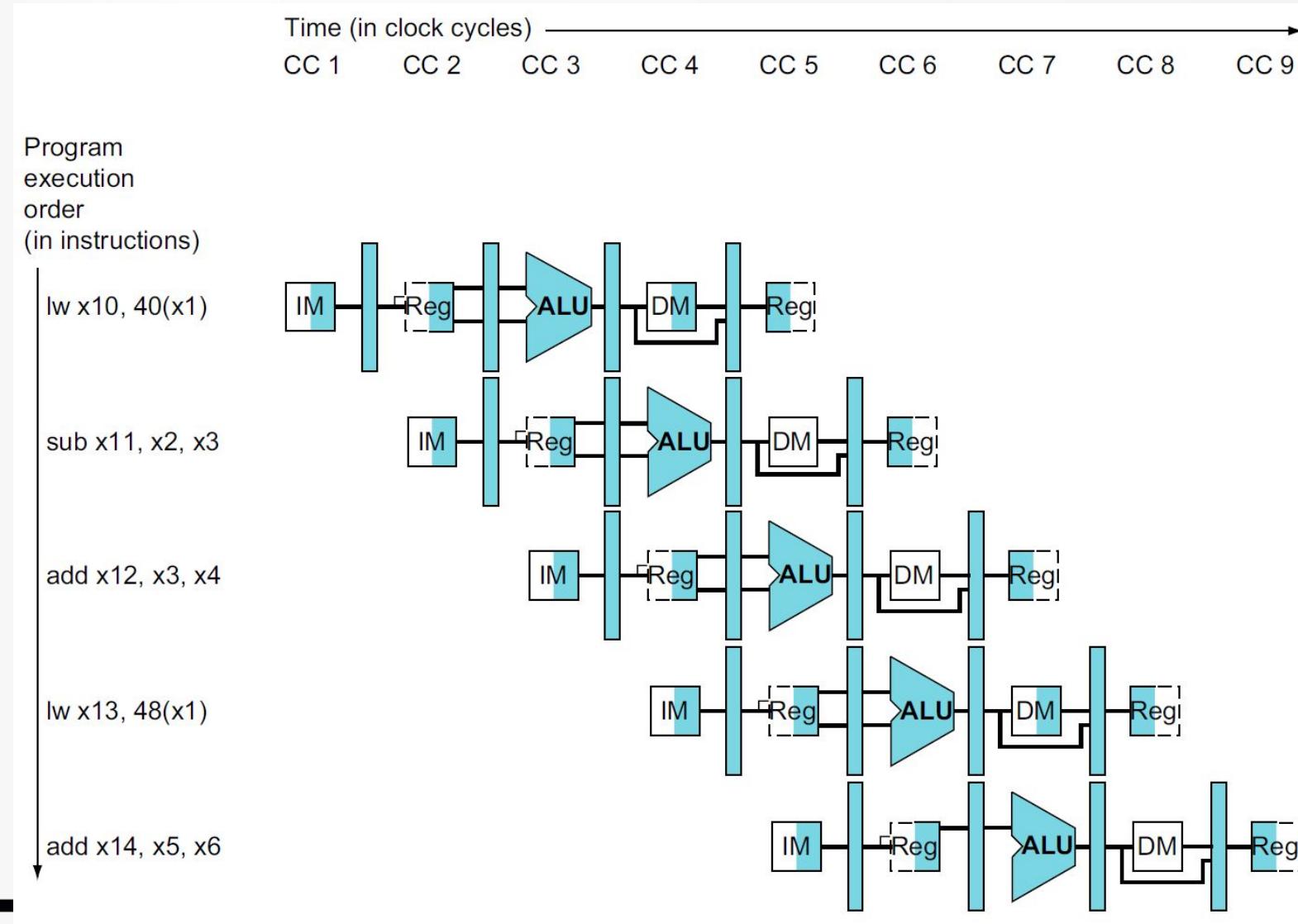
# Escrita de registradores (WB) – Store



# Diagrama de pipeline com múltiplos ciclos

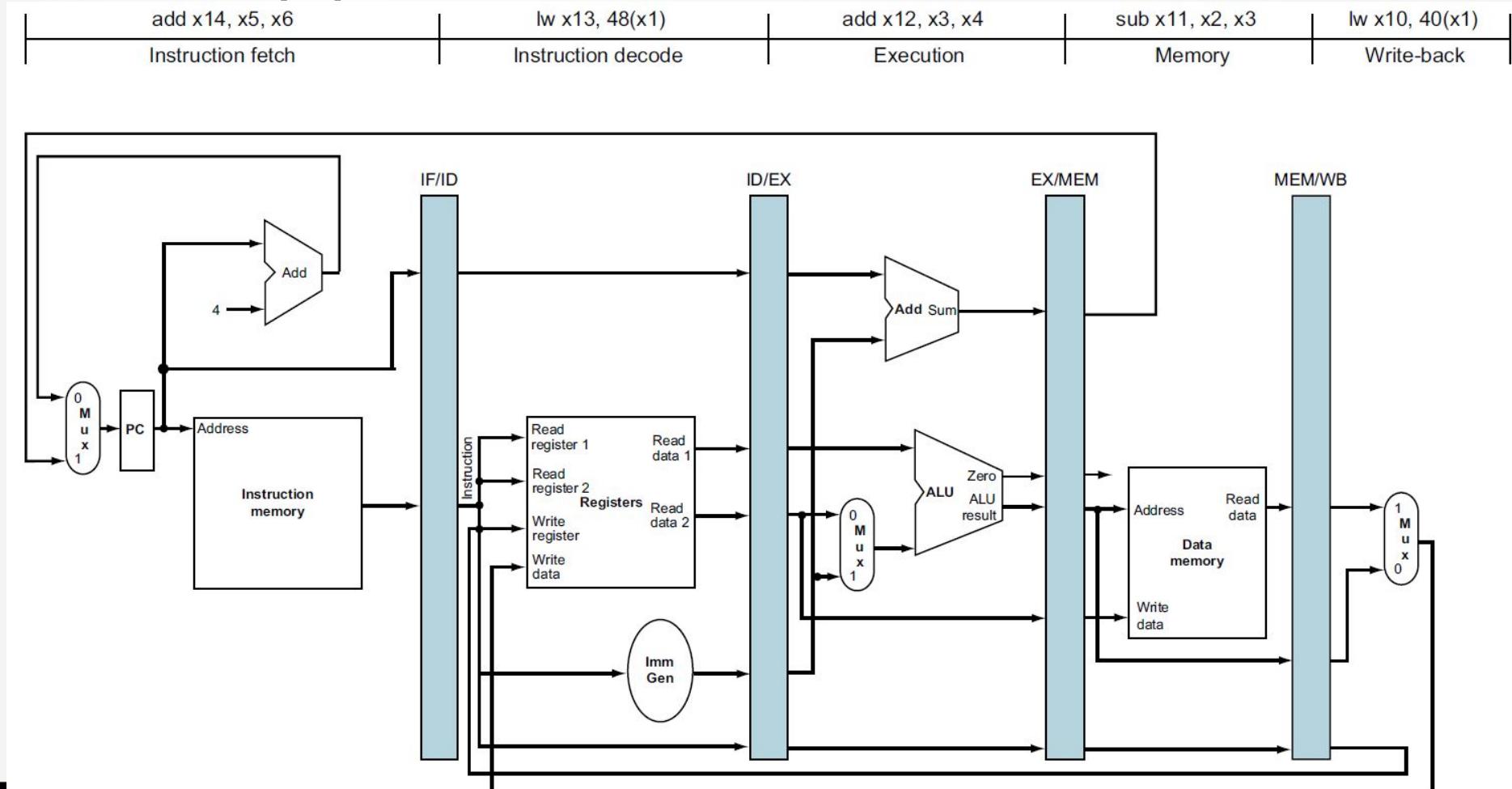


# Diagrama de pipeline com múltiplos ciclos

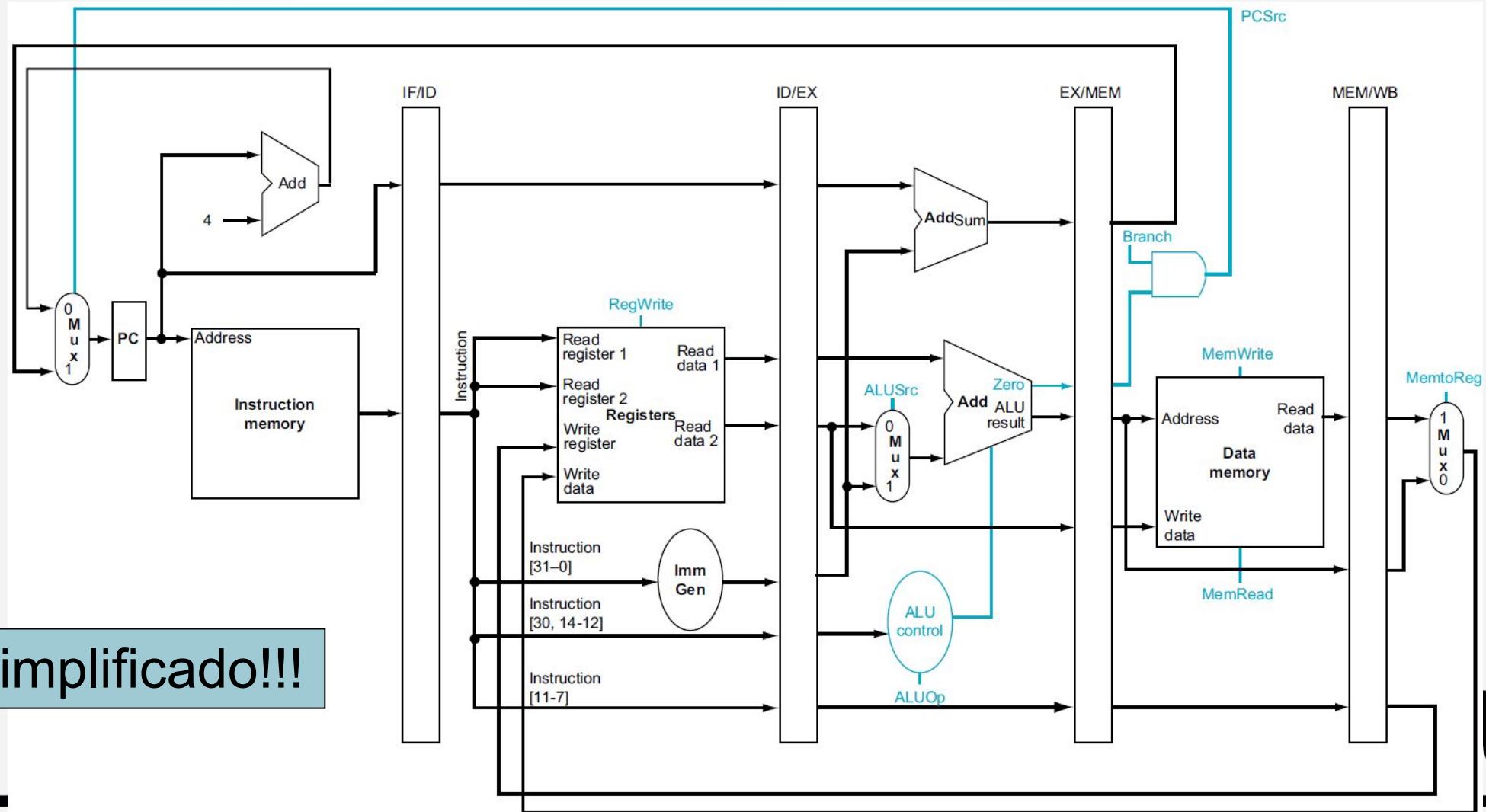


# Diagrama de pipeline com ciclo único

- Estado do pipeline em um determinado ciclo

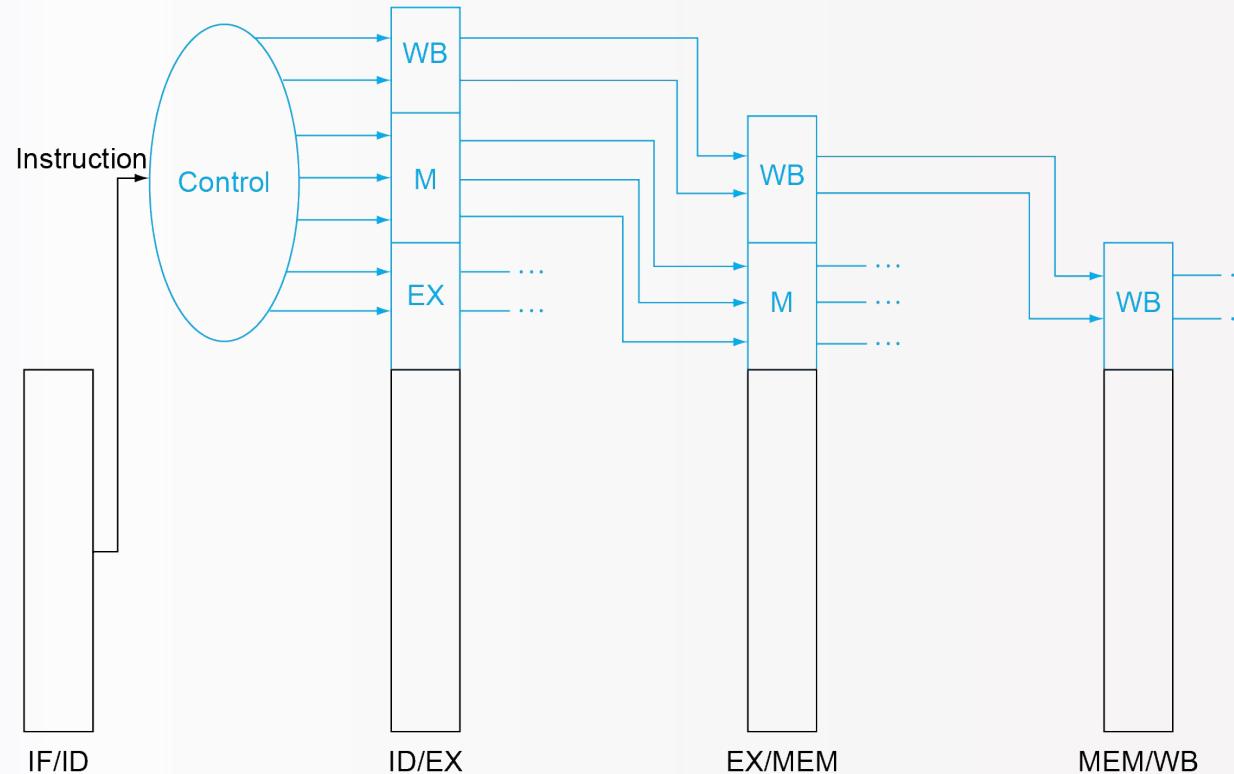


# Controle do caminho de dados com pipeline

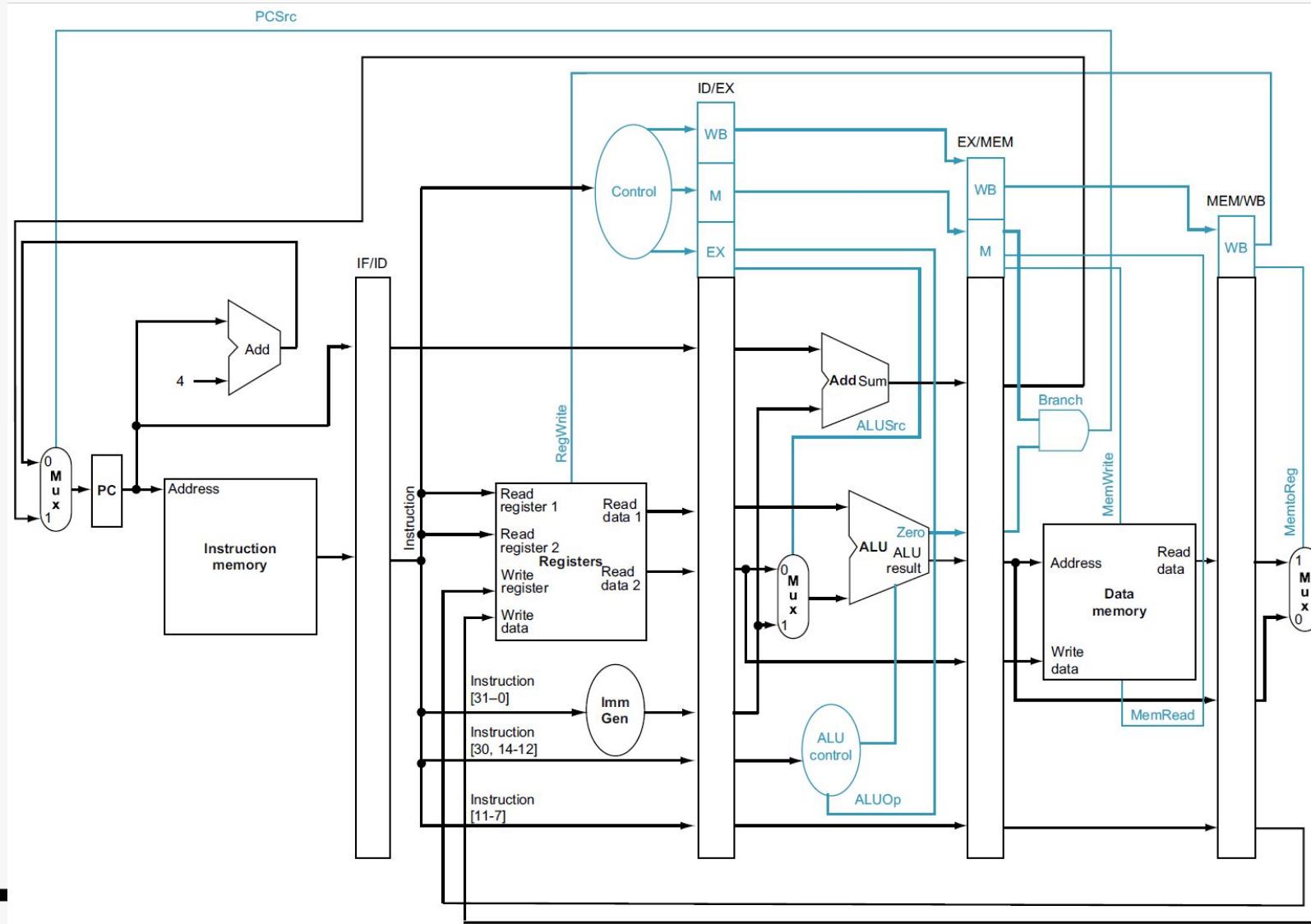


# Controle do caminho de dados com pipeline

- Sinais de controle derivados da instrução
  - Como na implementação de ciclo único



# Controle do caminho de dados com pipeline



# Dependências de Dados Encaminhamento e Paradas

Referência: Capítulo 4, Seção 4.8.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

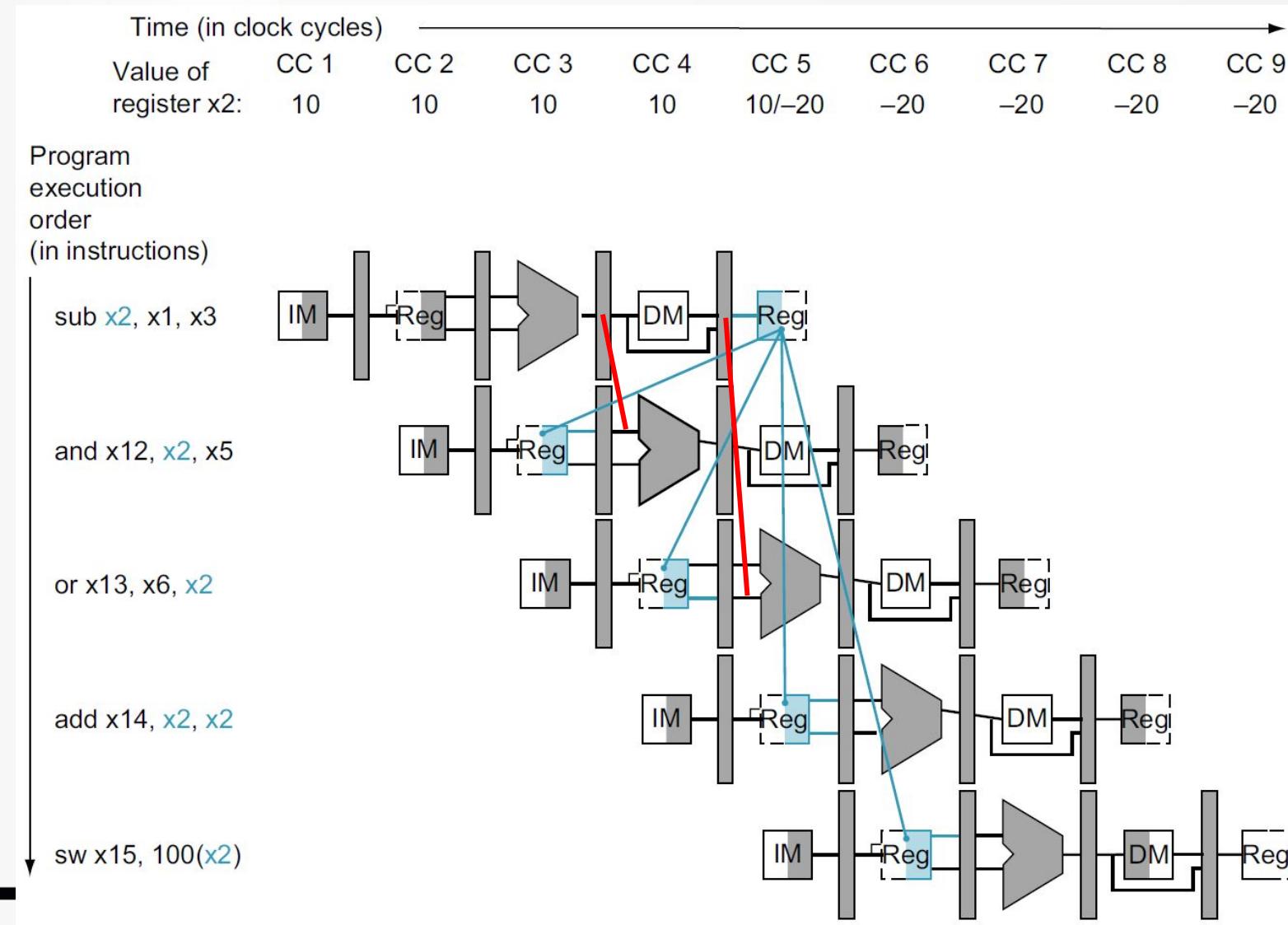
# Dependências de dados em instruções da ALU

- Considere esta sequência

```
sub  x2, x1, x3  
and  x12, x2, x5  
or   x13, x6, x2  
add  x14, x2, x2  
sw   x15, 100(x2)
```

- Podemos resolver dependências com encaminhamento
  - Como detectamos quando fazer o encaminhamento?

# Dependências e Encaminhamento



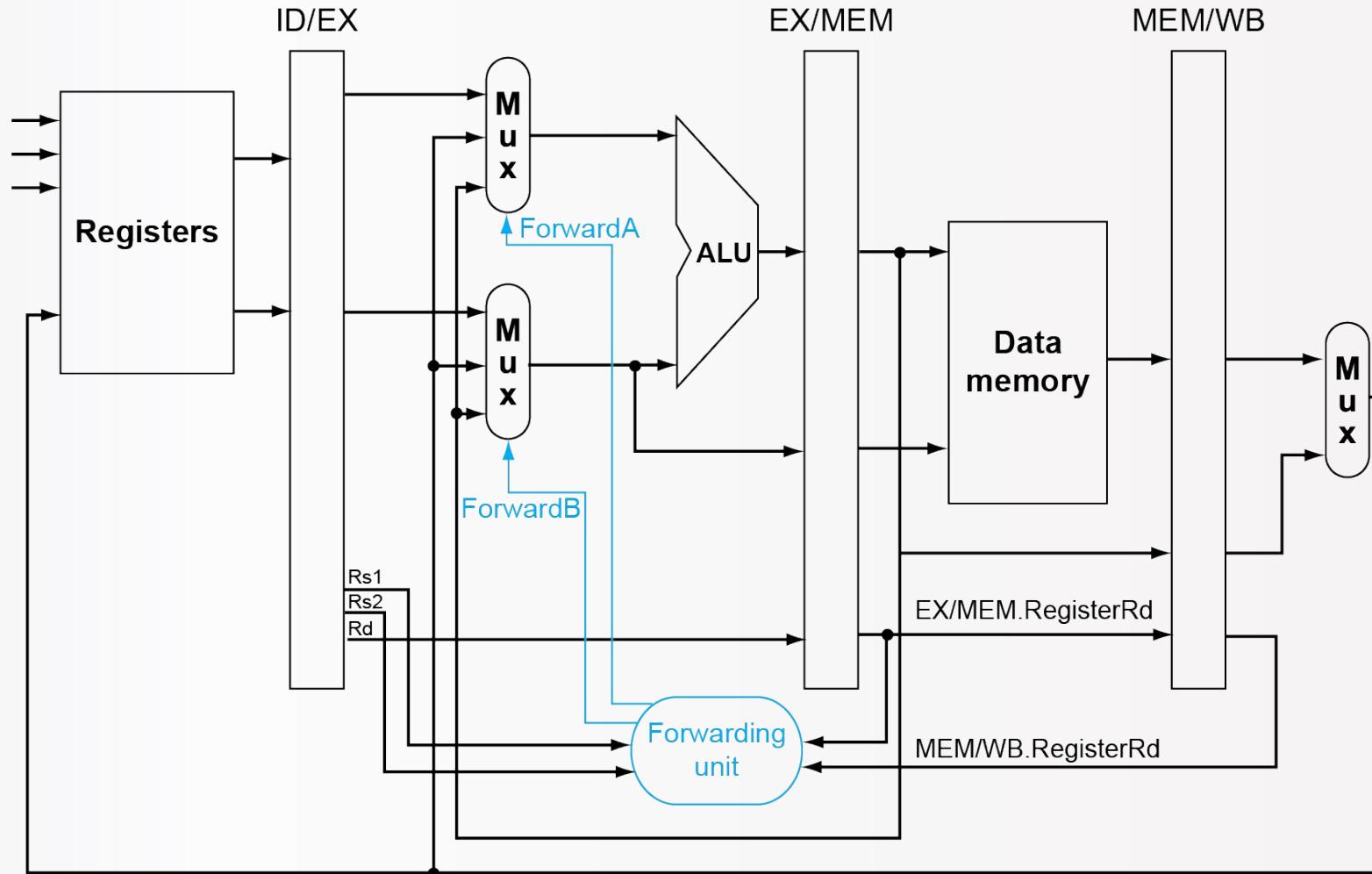
# Detectando a necessidade de encaminhamento

- Passe valores dos registradores ao longo do pipeline
    - e.g., ID/EX.RegisterRs1 = valor registrador em Rs1 no registrador de pipeline ID/EX
  - O valor dos registrador do operando ALU no estágio EX é dados por
    - ID/EX.RegisterRs1, ID/EX.RegisterRs2
  - Dependência de dados quando
    - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
    - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
    - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
    - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2
- 
- Encaminhamento dos registradores de pipeline EX/MEM
- Encaminhamento dos registradores de pipeline MEM/WB

# Detectando a necessidade de encaminhamento

- Mas somente se a instrução que gera o encaminhamento escrever em um registrador!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- E somente se Rd para essa instrução não for x0
  - EX/MEM.RegisterRd ≠ 0,  
MEM/WB.RegisterRd ≠ 0

# Conexões de encaminhamento



# Condições de encaminhamento

Controle do Mux	Origem	Explicação
ForwardA = 00	ID/EX	O primeiro operando ALU vem do banco de registradores.
ForwardA = 10	EX/MEM	O primeiro operando da ALU é encaminhado do resultado anterior da ALU.
ForwardA = 01	MEM/WB	O primeiro operando da ALU é encaminhado da memória de dados ou de um resultado anterior da ALU.
ForwardB = 00	ID/EX	O segundo operando da ALU vem do banco de registradores.
ForwardB = 10	EX/MEM	O segundo operando da ALU é encaminhado do resultado anterior da ALU.
ForwardB = 01	MEM/WB	O segundo operando da ALU é encaminhado da memória de dados ou de um resultado anterior da ALU.



# Dependência dupla

- Considere a sequência

add  $x_1, x_1, x_2$

add  $x_1, x_1, x_3$

add  $x_1, x_1, x_4$

- Neste caso o resultado deve ser encaminhado do estágio MEM

- Possuiu o resultado mais recente

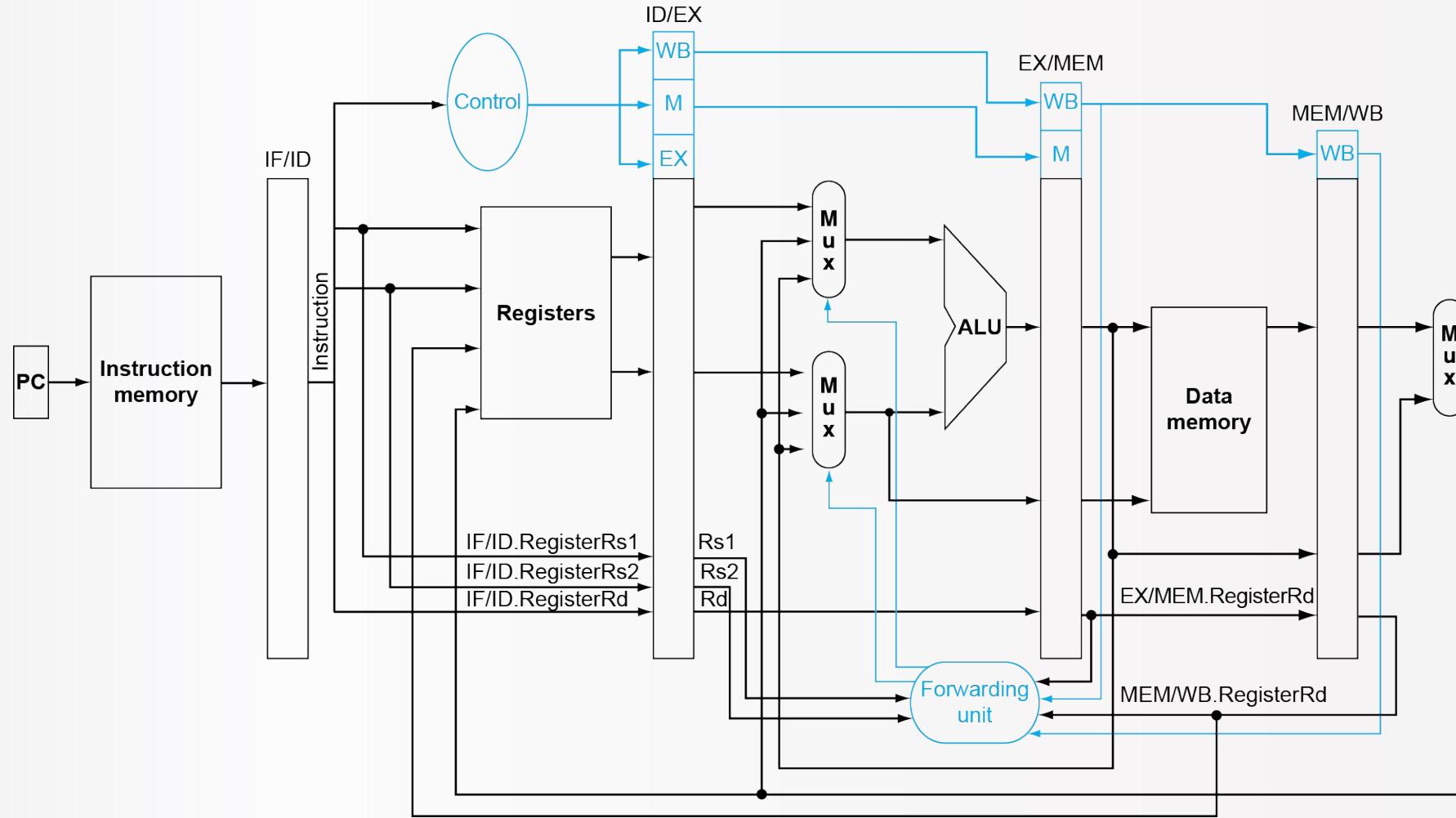
- Rever condição de dependência MEM

- Somente encaminhamento se a condição de dependência EX não for verdadeira

# Condição de encaminhamento revisada

- Dependência MEM
  - 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.RegisterRs1))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) 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.RegisterRs2))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

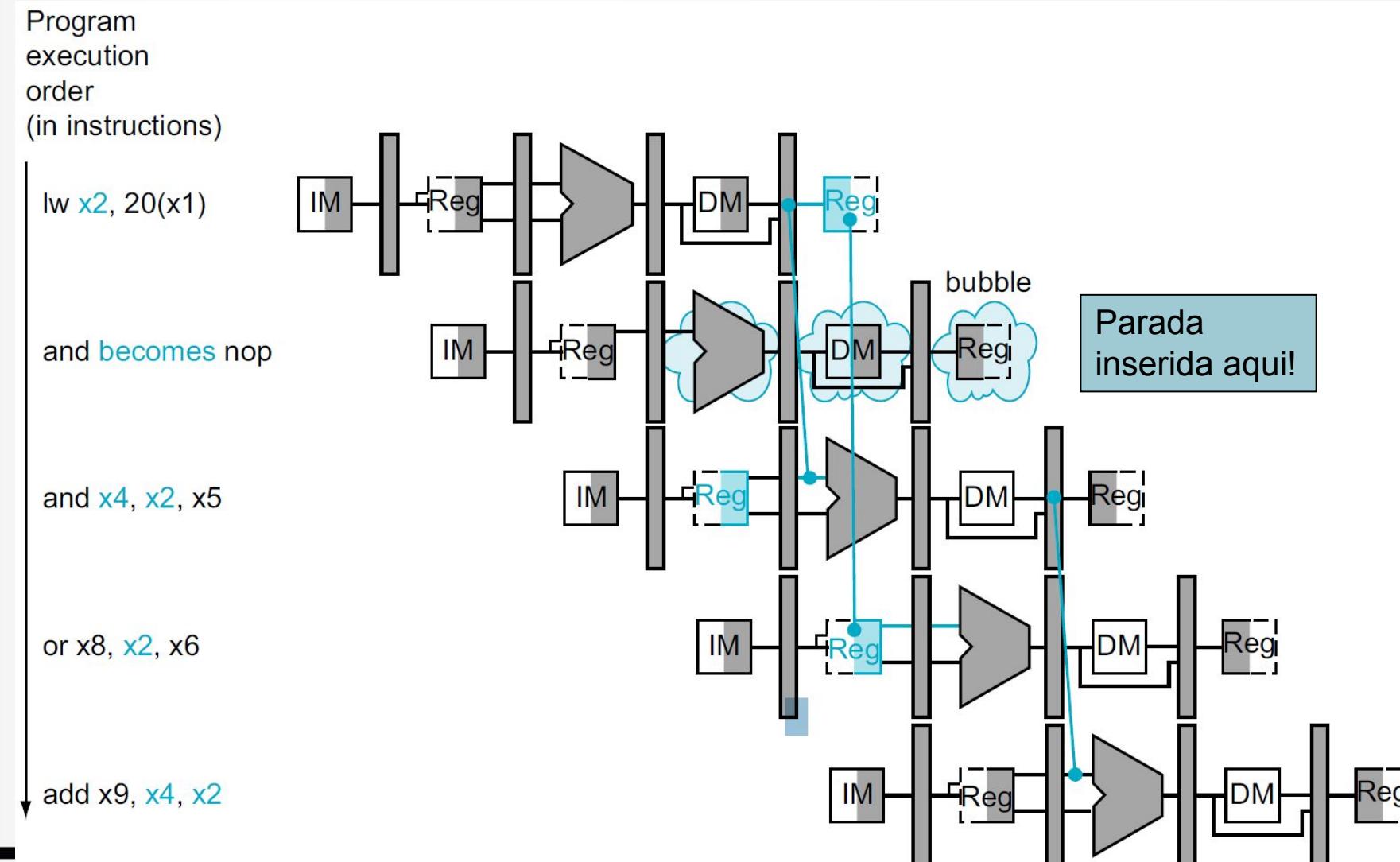
# Caminho de dados com encaminhamento



# Unidade de detecção de dependência (load)

- Verifique o estágio ID
- Os números do registrador do operando ALU no estágio de ID são fornecidos por
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Dependência de load quando
  - ID/EX.MemRead e ((ID/EX.RegisterRd = IF / ID.RegisterRs1) ou (ID/EX.RegisterRd = IF / ID.RegisterRs2))
- Se detectado, pare e insira a bolha

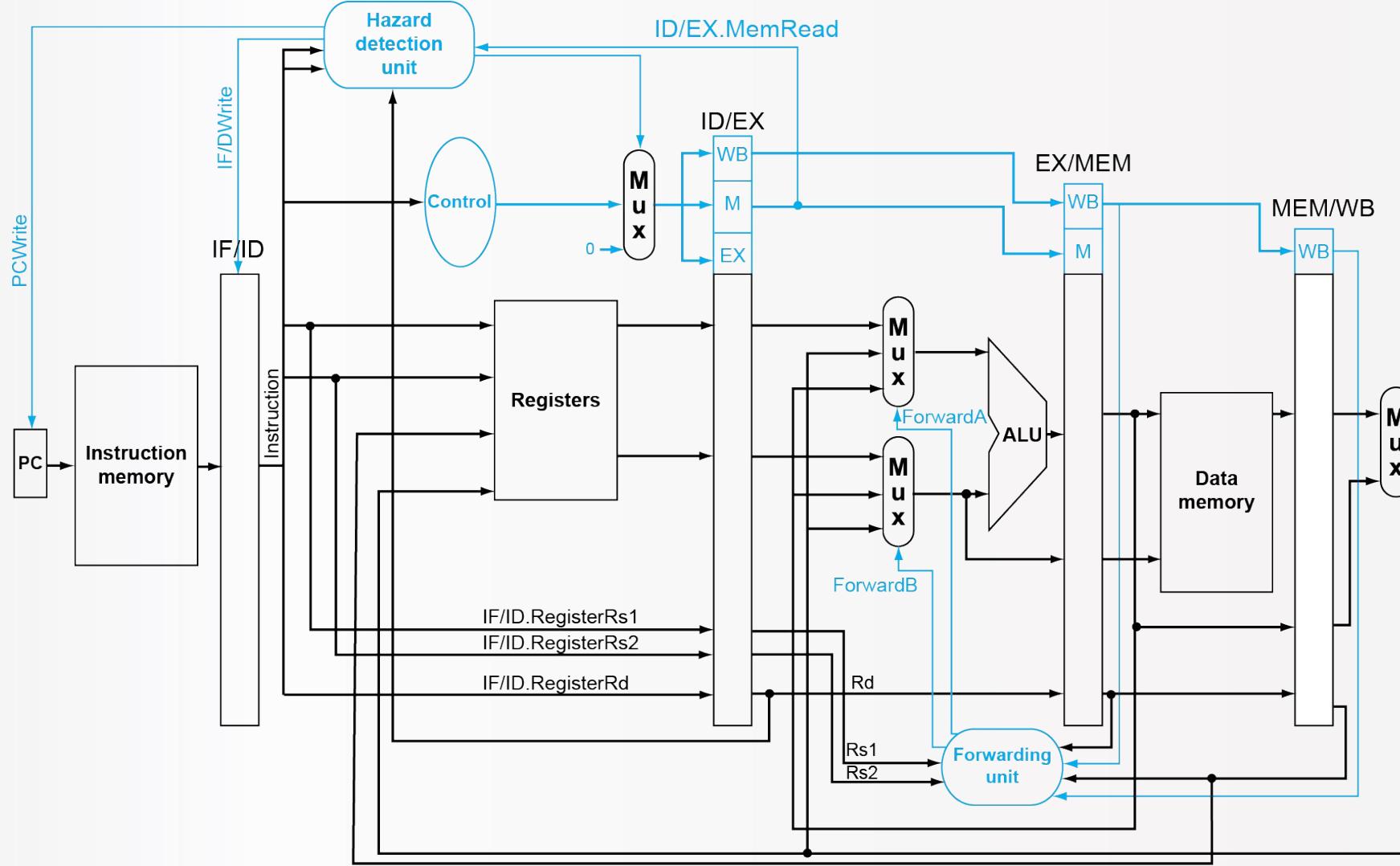
# Unidade de detecção de dependência (load)



# Como bloquear o pipeline

- Forçar valores de controle no registrador ID/EX para 0
  - EX, MEM e WB executam nop (sem operação)
- Impedir a atualização do PC e do registrador IF/ID
  - Instrução atual é decodificada novamente
  - A instrução a seguir é buscada novamente
  - A paralisação de 1 ciclo permite que o MEM leia dados para o load
  - Posteriormente, pode encaminhar para o estágio EX

# Unidade de detecção de dependência (load)



# Paradas e Desempenho

- Paradas reduzem o desempenho
  - Mas são necessárias para obter resultados corretos
- O compilador pode organizar o código para evitar dependências e paradas (bolhas) no pipeline
  - Requer conhecimento da estrutura do pipeline

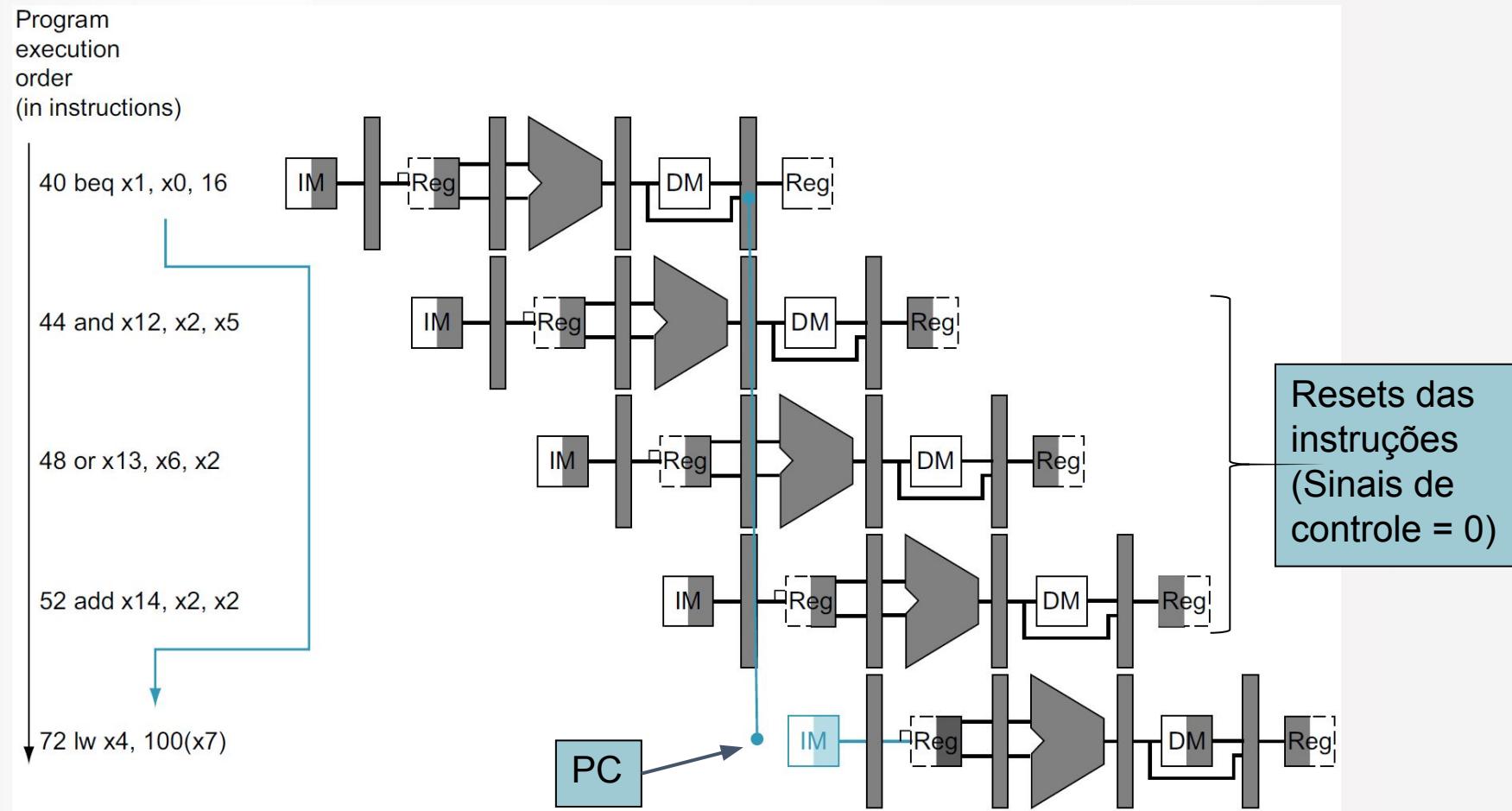
# Dependências de Desvio

Referência: Capítulo 4, Seção 4.9.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Dependências de desvio

- Se o resultado do desvio for determinado no MEM

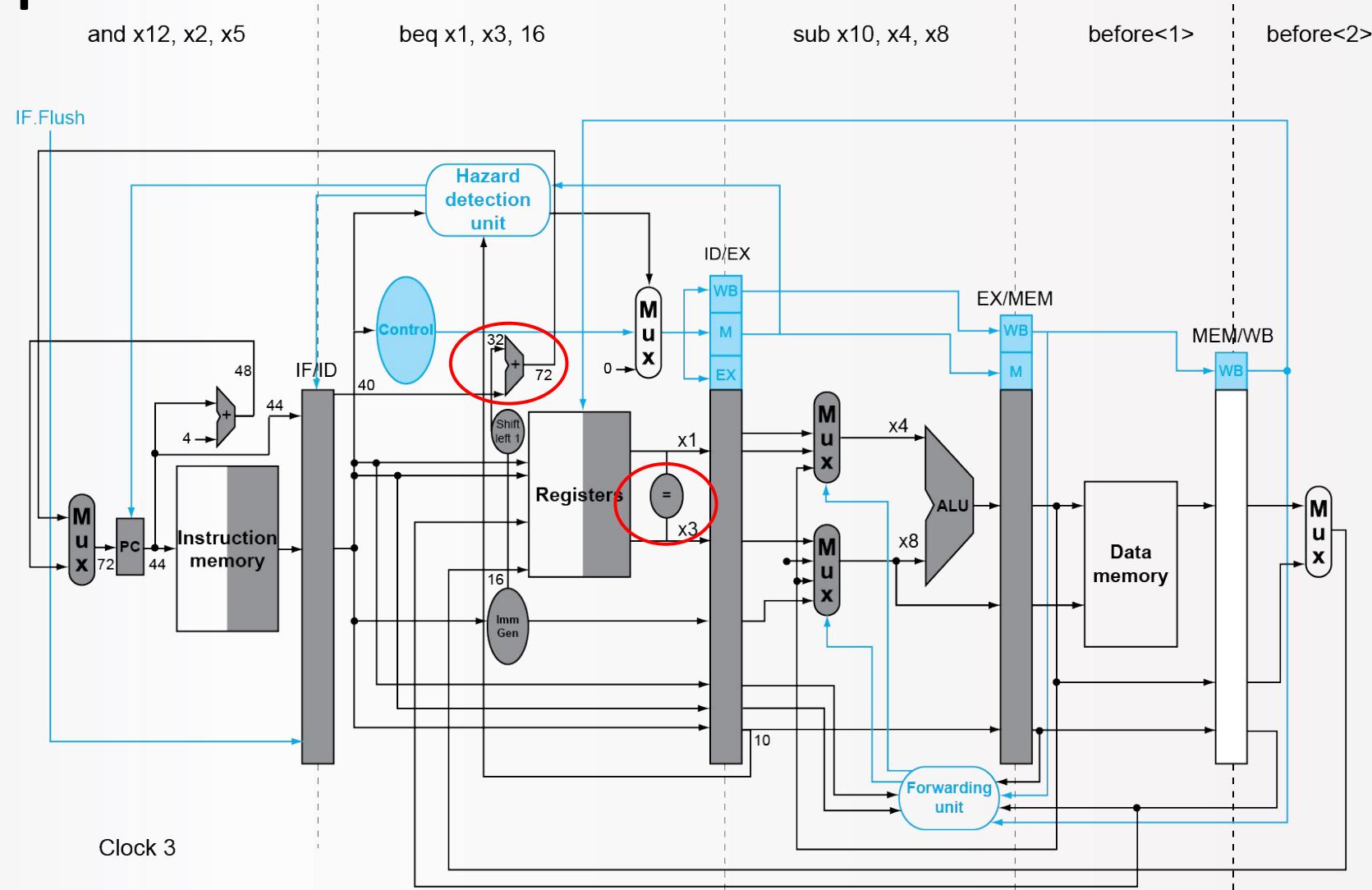


# Reduzindo o atraso do desvio

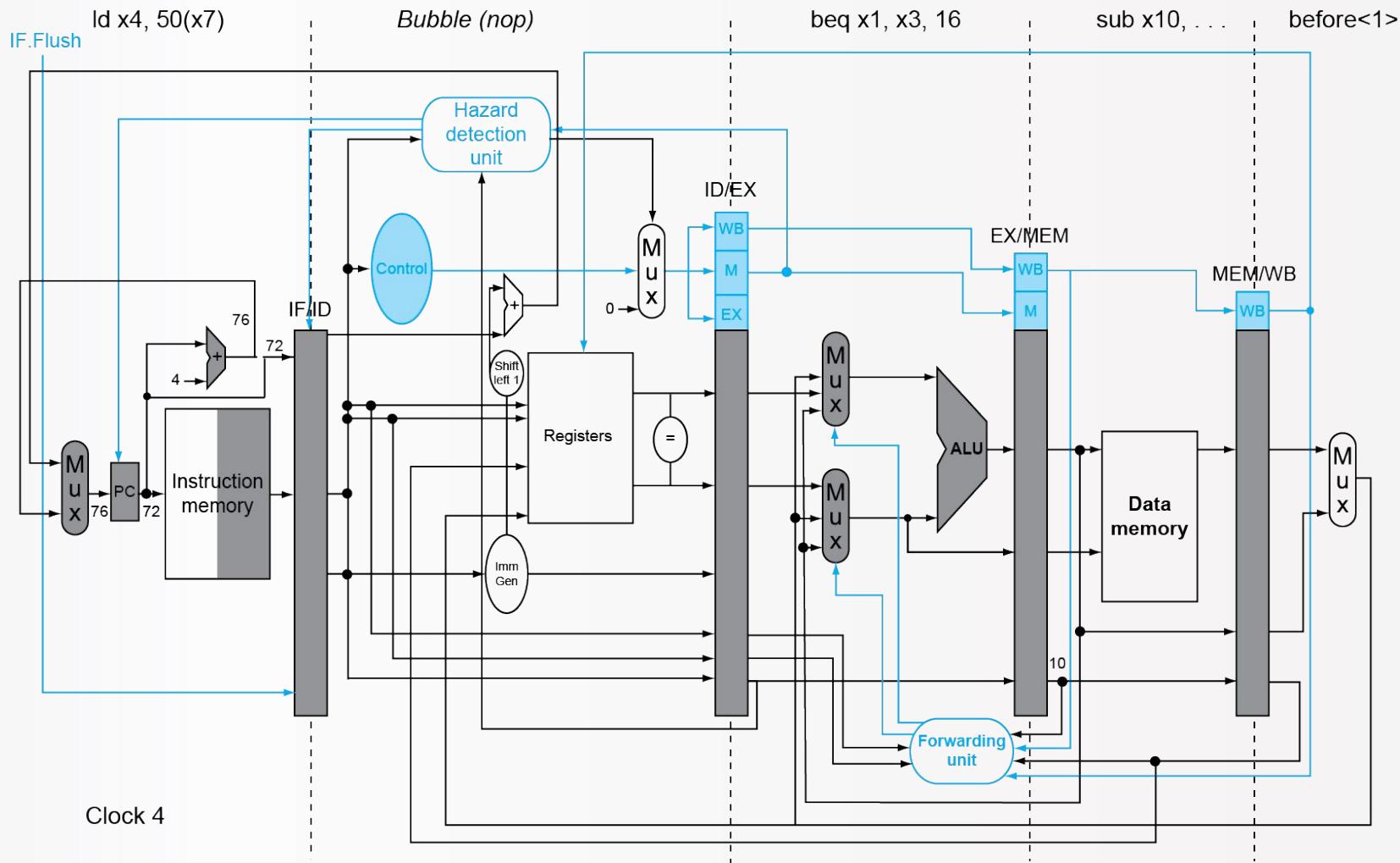
- Mover para ID o hardware de cálculo do resultado
  - Somador para endereço de destino
  - Comparador de registradores
- Exemplo: desvio tomado

```
36: sub x10, x4, x8
40: beq x1, x3, 16 // Desvio relativo ao PC para
                  // to 40+16*2=72
44: and x12, x2, x5
48: or x13, x2, x6
52: add x14, x4, x2
56: sub x15, x6, x7
...
72: lw x4, 50(x7)
```

# Exemplo: Desvio tomado



# Exemplo: Desvio tomado

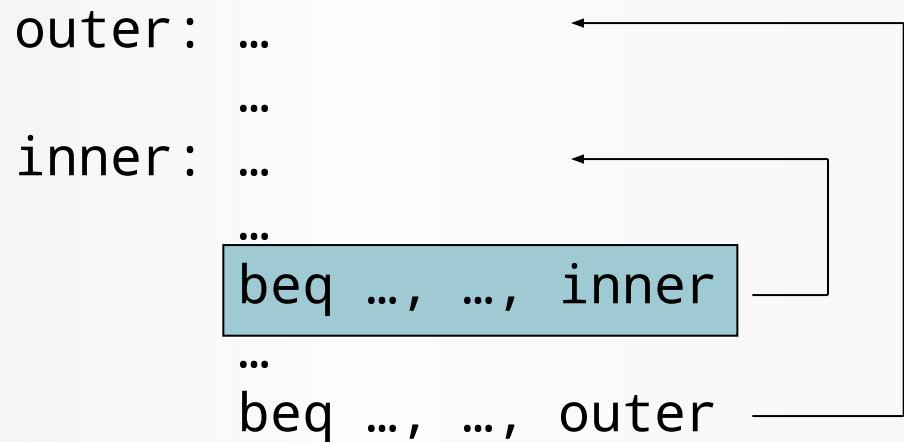


# Previsão dinâmica de desvios

- Em pipelines profundos e superescalares, o custo de parada por dependências de desvio é significativa
- Previsão dinâmica
  - Buffer de previsão de desvio (tabela de histórico de desvios)
  - Endereços de desvios recentes
  - Armazena resultado (tomado/não tomado)
  - Na execução de um desvio
    - Verificar a tabela, assumindo o mesmo resultado
    - Iniciar busca da próxima instrução
    - Se estiver errado, esvazie o pipeline e inverta a previsão

# Predictor de 1 bit

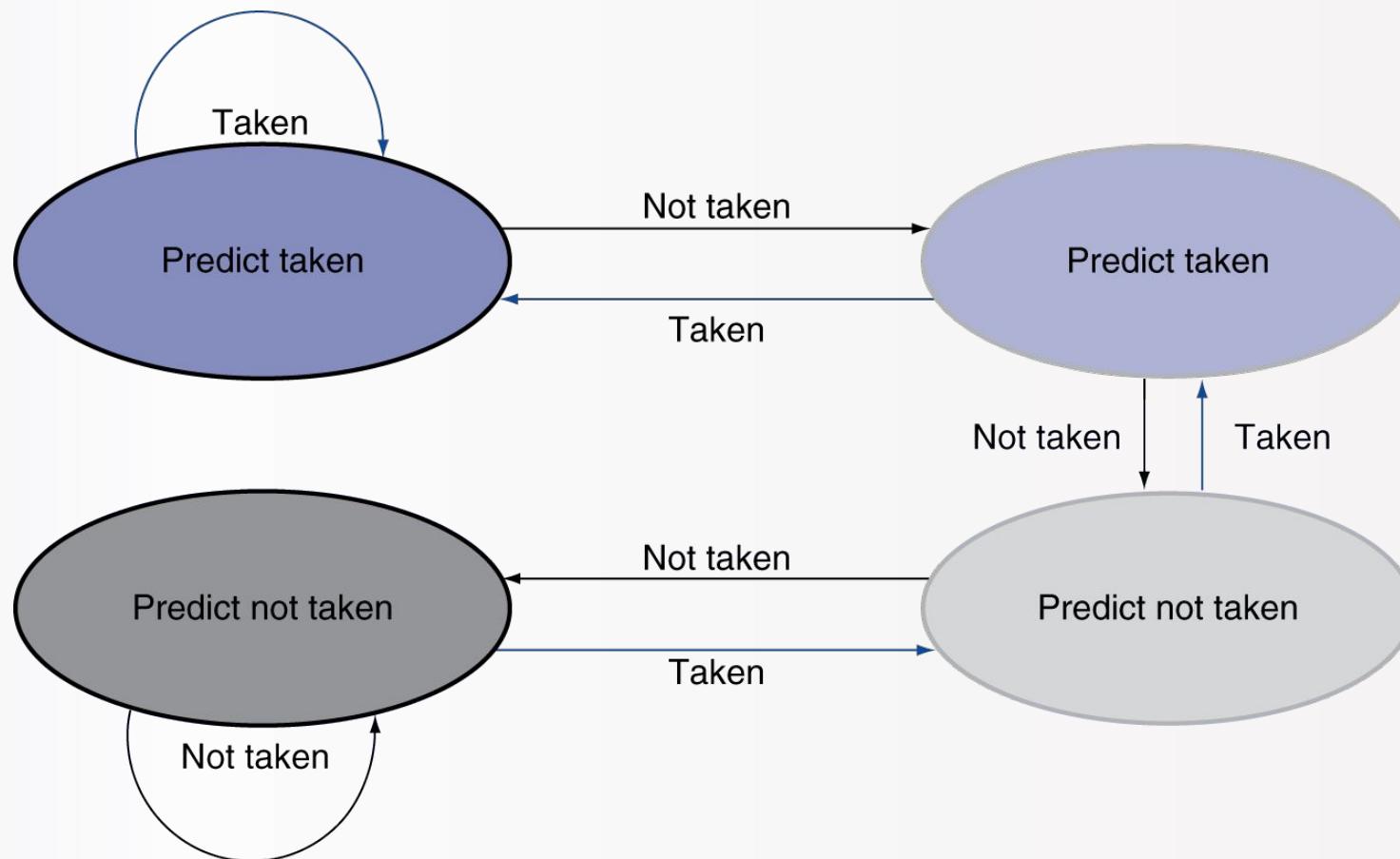
- Desvio de repetição interna errada duas vezes!



- Erro na previsão da última iteração da repetição interna
- Em seguida, erro na primeira iteração da repetição interna

# Predictor de 2 bits

- Muda a previsão após dois erros seguidos



# Calculando o endereço do desvio

- Mesmo com preditor, ainda é preciso calcular o endereço de destino
  - Custo de 1 ciclo para um desvio tomado
- Buffer de endereços de destino (*Branch target buffer*)
  - Cache de endereços de destino
  - Indexado pelo PC quando a instrução é buscada
    - Se acertar e a instrução é um desvio com previsão de tomado, podem realizar a busca da próxima instrução imediatamente

# Exceções

Referência: Capítulo 4, Seção 4.10.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.



# Exceções e interrupções

- Eventos que geram mudança no fluxo de controle
  - Diferentes ISAs usam os termos de maneiras diferentes
- Exceção
  - É gerada na própria CPU
    - Por exemplo, *opcode* indefinido, chamada de sistema, ...
- Interrupção
  - Gerada por um controlador externo de E/S
- Lidar com exceções e interrupções sem perder desempenho é difícil

# Tratamento de exceções

- Salvar o PC da instrução que está sendo executada
  - No RISC-V: *Supervisor Exception Program Counter* (SEPC)
- Salvar a origem do problema
  - No RISC-V: *Supervisor Exception Cause Register* (SCAUSE)
    - 64 bits, mas a maioria dos bits não utilizados
      - Campo de código de exceção: 2 bits para opcode indefinido, 12 para mau funcionamento de hardware, ...
- Desviar para o tratador (*handler*)
  - Assumir o endereço
    - 0000 0000 1C09 0000<sub>hex</sub>

# Mecanismo alternativo

- Interrupções vetorizadas
  - Endereço do tratador (*handler*) determinado pela causa
- Endereço do vetor de exceção somado a um registrador base. Exemplo:
  - Opcode indefinido: 00 0100 0000<sub>2</sub>
  - Mau funcionamento de hardware: 01 1000 0000<sub>2</sub>
- Instruções
  - Tratam a interrupção, ou
  - Desviam para o tratador real

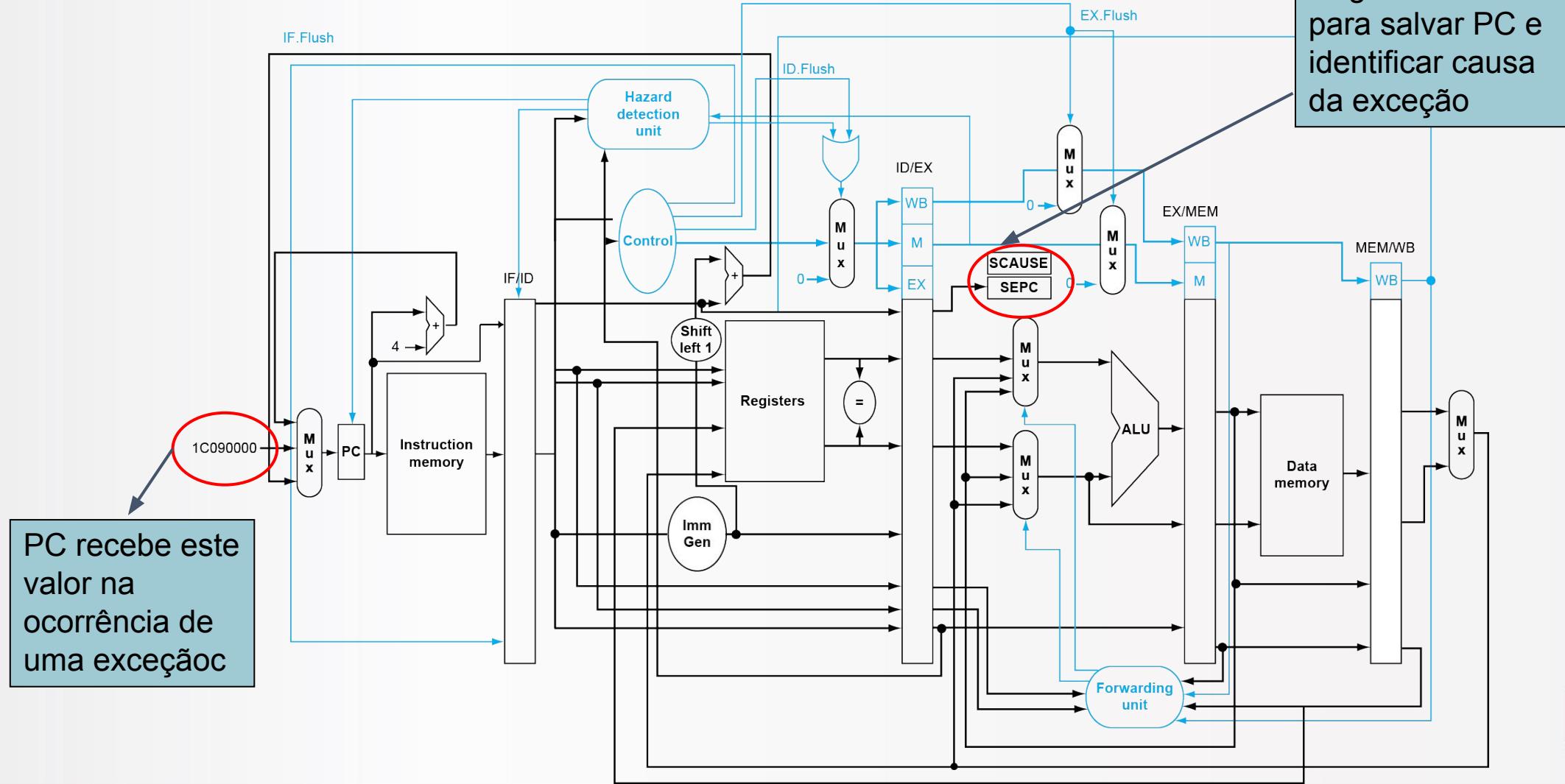
# Ações do tratador

- Identificação da causa e desvio para a rotina de tratamento correspondente
- Determinar a ação necessária
- Se reiniciável
  - Tomar uma ação corretiva
  - Utilizar o registrador SEPC para retornar ao programa
- Caso contrário a forma
  - Finalizar o programa
  - Reportar erro usando SEPC, SCAUSE, ...

# Pipeline com exceções

- Outra forma de dependência de controle
- Considere um mau funcionamento em uma instrução add no estágio EX: add x1 , x2 , x1
  - Impedir que x1 seja escrito no estágio WB
  - Completar as instruções anteriores
  - Resetar a execução do add e as instruções seguintes
  - Definir valores dos registradores SEPC e SCAUSE
  - Transferir o controle para o tratador
- Semelhante a um erro de predição de desvio
  - Utiliza boa parte do hardware já existente

# Pipeline com exceções



# Propriedades das exceções

- Exceções reinicializáveis
  - O pipeline pode resetar a instrução
  - Tratador é executado e, em seguida, retorna à instrução
    - Instrução é buscada novamente e executado do zero
- PC salvo no registrador SEPC
  - Identifica a instrução que causou a exceção

# Exemplo de exceção

- Exceção em add

```
40  sub   x11,  x2,  x4  
44  and   x12,  x2,  x5  
48  orr   x13,  x2,  x6  
4C  add   x1,   x2,  x1  
50  sub   x15,  x6,  x7  
54  lw    x16,  100(x7)
```

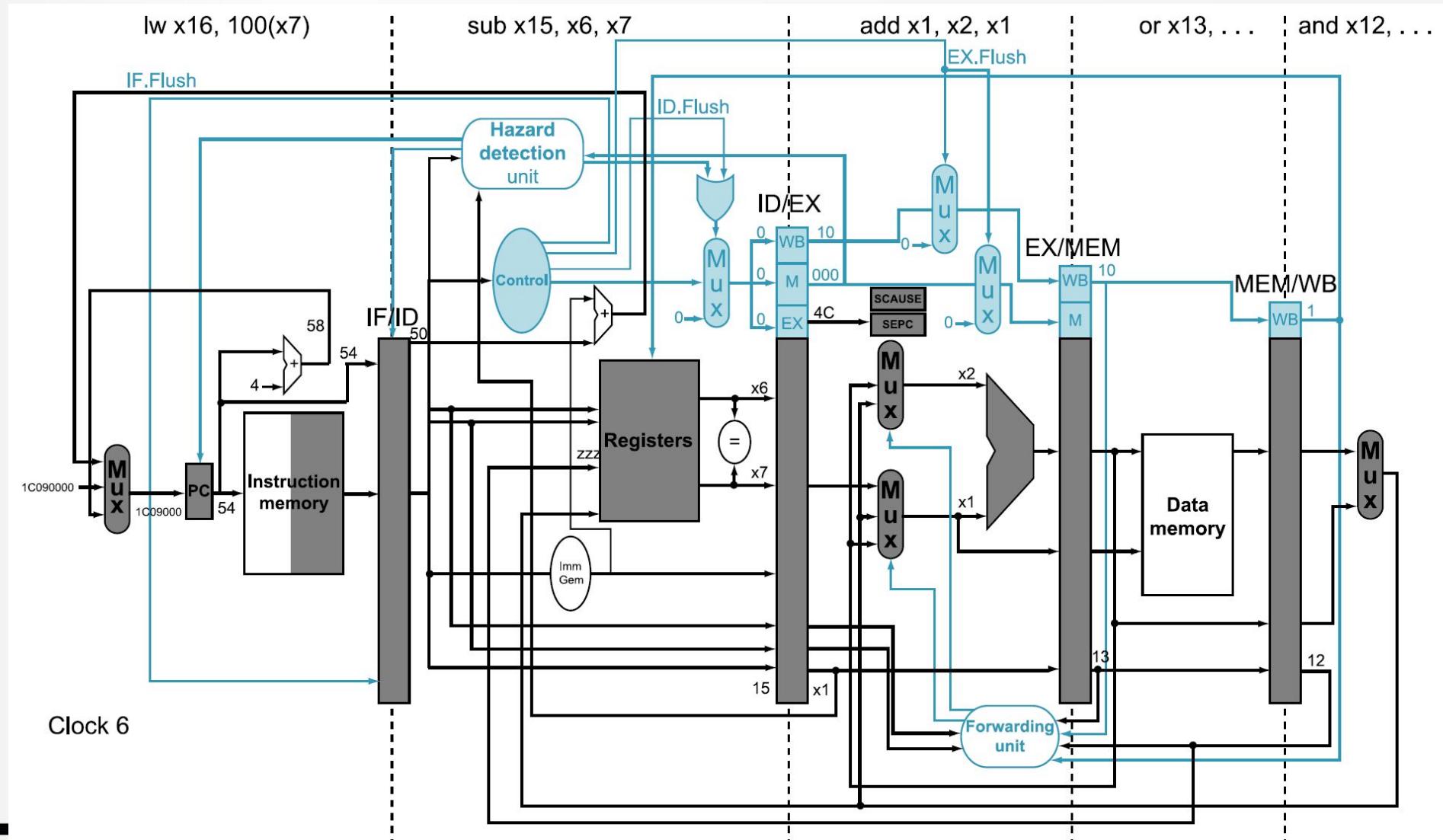
...

- Tratador da exceção

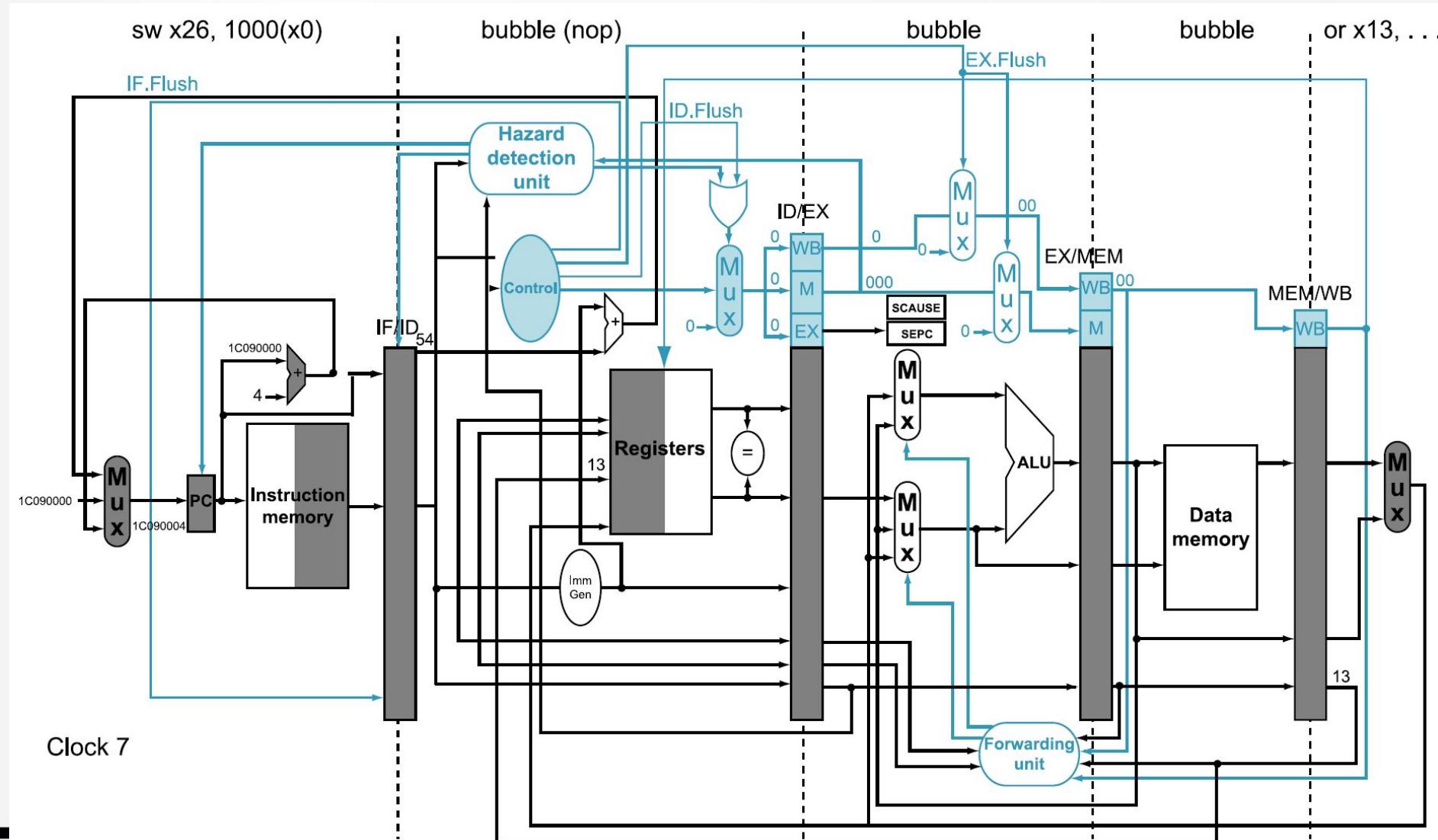
```
1C090000      sw    x26, 1000(x10)  
1c090004      sw    x27, 1008(x10)
```

...

# Exemplo de exceção



# Exemplo de exceção



# Múltiplas exceções

- O pipeline se sobrepõe a várias instruções
  - Várias exceções simultaneamente
- Abordagem simples
  - Lidar com a exceção da instrução inicial
  - Resetar as instruções seguintes
  - Exceções “precisas”
- Em pipelines complexos
  - Várias instruções buscadas por ciclo
  - Execução fora de ordem
  - Manter exceções precisas é difícil!

# Exceções imprecisas

- Basta parar o pipeline e salvar o estado
  - Incluindo a(s) causa(s) da exceção
- Tratador da exceção
  - Quais instruções tiveram exceções
  - Qual para completar ou liberar
- Simplifica o hardware transferindo complexidade para o software
- Inviável em pipelines *multiple issue* com execução fora de ordem

# Paralelismo utilizando Instruções

Referência: Capítulo 4, Seção 4.11.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Paralelismo em nível de instrução

- Pipelining: execução de várias instruções em paralelo
- Para aumentar o ILP (*Instruction-level parallelism*)
  - Pipeline mais profundo
    - Menos trabalho por estágio ⇒ ciclo de clock mais curto
    - No exemplo da lavanderia
      - Fazer nova divisão de estádios como, por exemplo
        - Lavar
        - Enxaguar
        - Centrifugar
      - Pipeline agora com 6 estágios
      - Balanceamento de estágios de forma que durem o mesmo tempo é importante

# Paralelismo em nível de instrução

- Para aumentar o ILP (*Instruction-level parallelism*)
  - Disparo de múltiplas instruções
    - Estágios de pipeline replicados ⇒ vários pipelines
    - Iniciar várias instruções por ciclo de clock
    - Por exemplo, 4 GHz e 4 vias para disparo múltiplo
      - Pico: CPI = 0,25 e IPC (Instruções por Ciclo) = 4
    - Processadores modernos com IPC entre 3 e 6
    - Mas as dependências reduzem isso na prática
    - No exemplo da lavadora
      - 3x o número de lavadoras, secadoras e pessoas
      - Desafio é manter todos ocupados

# Disparo múltiplo de instruções

- Estático
  - O compilador agrupa as instruções a serem disparadas simultaneamente
  - Agrupamento em slots
  - O compilador detecta e evita dependências
- Dinâmico
  - A CPU examina o fluxo de instruções e escolhe as instruções para disparar a cada ciclo
  - O compilador pode ajudar reordenando as instruções
  - CPU resolve dependências em tempo de execução

# Especulação

- “Adivinhar” o que fazer com uma instrução
  - Começar a operação o mais rápido possível
  - Verificar se a suposição estava certa
    - Se sim, concluir a operação
    - Se não, retroceder e corrigir
- Utilizada tanto em disparos múltiplos estáticos quanto dinâmicos
- Exemplos
  - Especular sobre o resultado do desvio
    - Reverter se não corresponder ao resultado do desvio
  - Especular no carregamento
    - Reverter se a localização for atualizada

# Especulação no compilador ou no hardware

- O compilador pode reordenar as instruções
  - Por exemplo, mover um load para antes de um desvio
  - Pode incluir instruções de “correção” para recuperar de especulações incorretas
- O hardware pode esperar instruções para executar
  - Guardar os resultados em um buffer até determinar se eles são realmente necessários
  - Limpar o buffer em caso de especulação incorreta

# Especulação e exceções

- E se ocorrer uma exceção em uma instrução executada especulativamente?
  - Por exemplo, um load especulativo antes de uma verificação de ponteiro nulo
- Especulação estática
  - Pode adicionar suporte no ISA para adiar tratamento de exceções
- Especulação dinâmica
  - Pode armazenar exceções até a conclusão da instrução

# Disparo múltiplo de instruções estático

- O compilador agrupa as instruções em pacotes de disparo (*issue packets*)
  - Pacote de instruções que podem ser disparadas em um ciclo
  - Determinado pelos recursos de pipeline necessários
- Pode-se pensar em um pacote de disparo como uma instrução muito longa
  - Especifica várias operações concorrentes simultâneas
  - Palavra de instrução muito longa (VLIW)

# Agendamento de vários disparos estáticos

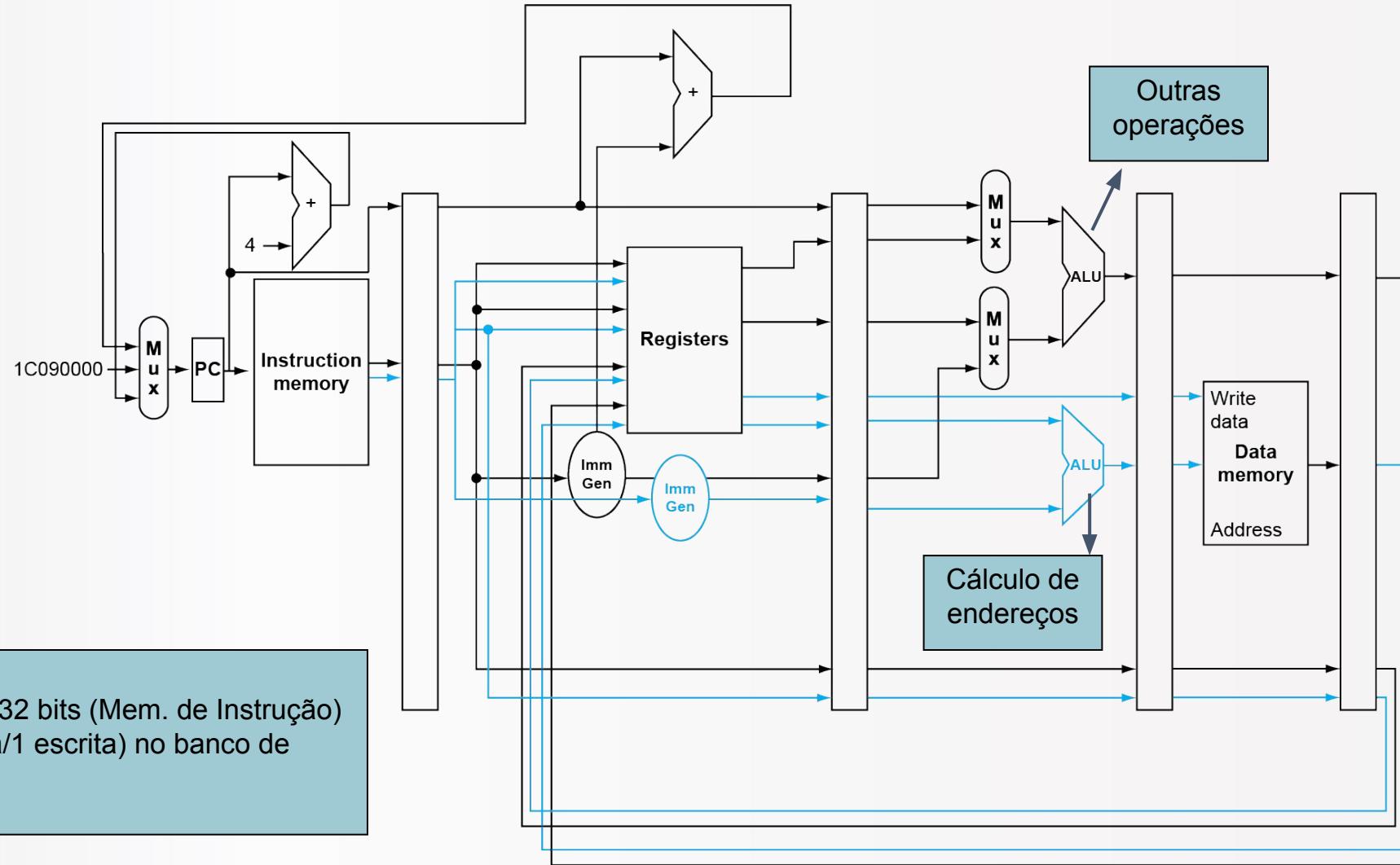
- O compilador deve remover as dependências
  - Reordenar as instruções em pacotes de disparo
    - Sem dependências no pacote
  - Possivelmente algumas dependências entre pacotes
    - Varia entre os ISAs, o compilador deve saber!
  - Preencher com nop se necessário

# RISC-V com disparo estático de 2 instruções

- Pacotes de 2 instruções
  - ALU/Desvio
  - Load/store
  - Alinhado em 64 bits
    - ALU/Desvio, em seguida, load/store
    - Instrução não utilizada preenchida com nop

Endereço	Tipo de instrução	Estágios do pipeline						
n	ALU/Desvio	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/Desvio		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/Desvio			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

# RISC-V com disparo estático de 2 instruções



# Dependências ao executar 2 instruções

- Mais instruções executando em paralelo
- Dependência de dados EX
  - Não é mais possível utilizar resultado da ALU em um load/store no mesmo pacote
    - add `x10`, `x0`, `x1`
    - lw `x2`, 0 (`x10`)
    - Dividir em dois pacotes, uma parada é inevitável
- Dependência de carregamento
  - Um ciclo de latência, mas agora 2 instruções
  - É necessário um escalonamento mais agressivo

# Exemplo de escalonamento

- RISC-V com disparo estático de 2 instruções

```
Loop: lw    x31,0(x20)      // x31=elemento do vetor
       add   x31,x31,x21    // soma escalar em x21
       sw    x31,0(x20)      // armazena resultado
       addi  x20,x20,-8      // decrementa ponteiro
       blt  x22,x20,Loop     // desvia se x22 < x20
```

	ALU/Desvio	Load/store	cycle
Loop:	nop	lw x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sw x31,8(x20)	4

- Instruções por ciclo =  $5/4 = 1,25$  (Máx = 2)

# Desenrolando laços (*loop unrolling*)

- Replicar o corpo do laço para melhorar o paralelismo
  - Reduz a sobrecarga de controle do laço
- Utiliza registradores diferentes para cada replicação
  - “Renomeação de registrador”
  - Evite dependências comuns em laços
    - Store seguido de um load no mesmo registrador

# Exemplo: Desenrolando laços

	ALU/Desvio	Load/store	Ciclo
Loop:	addi x20,x20,-32	lw x28, 0(x20)	1
	nop	lw x29, 24(x20)	2
	add x28,x28,x21	lw x30, 16(x20)	3
	add x29,x29,x21	lw x31, 8(x20)	4
	add x30,x30,x21	sw x28, 32(x20)	5
	add x31,x31,x21	sw x29, 24(x20)	6
	nop	sw x30, 16(x20)	7
	blt x22,x20,Loop	sw x31, 8(x20)	8

- Instruções por ciclo =  $14/8 = 1,75$ 
  - Mais perto de 2, mas ao custo de mais registradores e mais instruções no executável

# Disparo múltiplo dinâmico

- Processadores superescalares
- CPU decide quantas instruções deve disparar a cada ciclo
  - Evitando dependências estruturais e de dados
- Evita a necessidade de escalonamento pelo compilador
  - Embora ainda possa ajudar
  - Execução correta de código garantida pela CPU

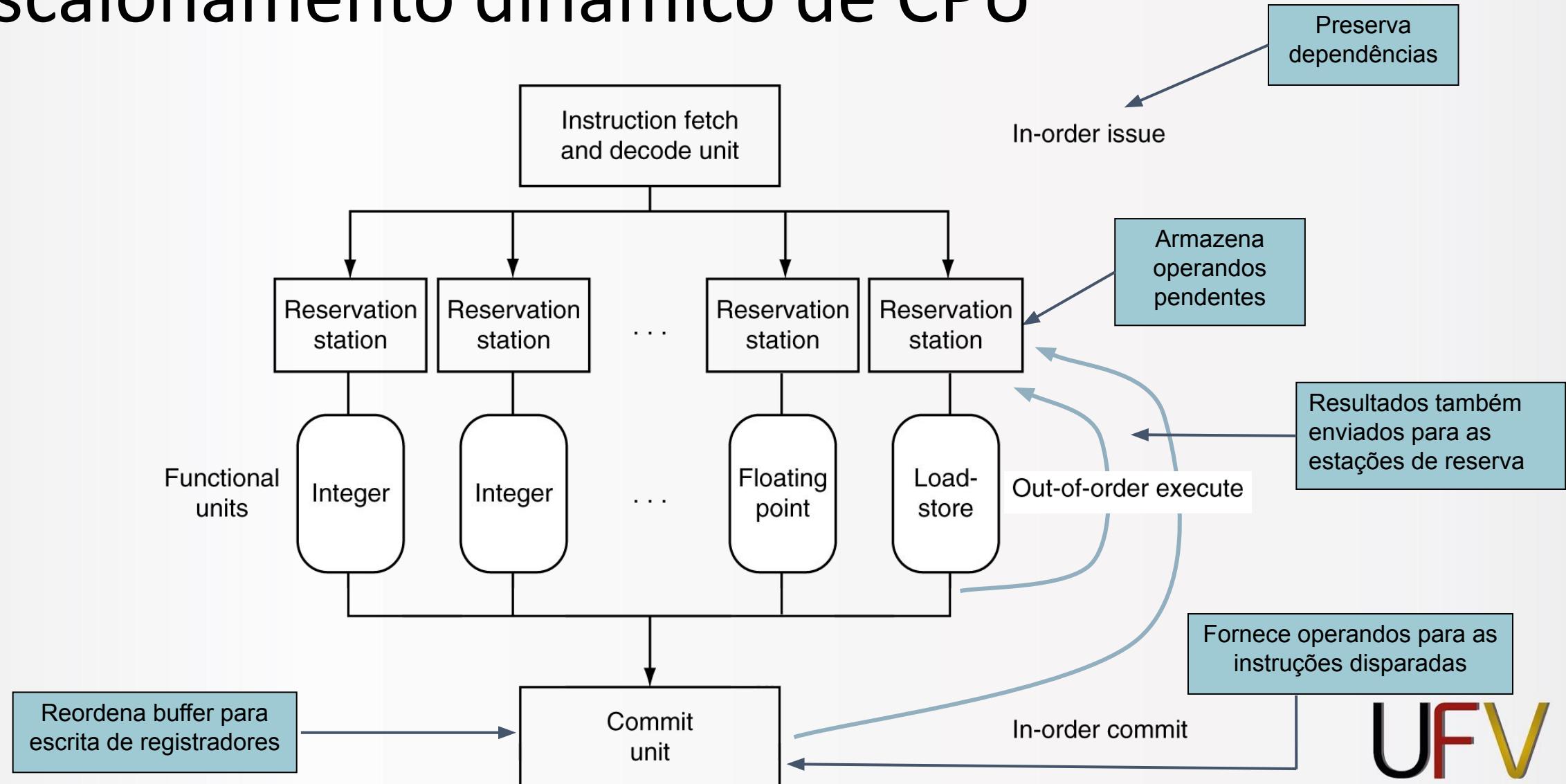
# Escalonamento dinâmico de pipeline

- CPU executa instruções fora de ordem para evitar paradas no pipeline
  - Escrita dos resultados nos registradores em ordem
- Exemplo

```
lw      x31, 20(x21)
add    x1,  x31, x2
sub    x23, x23, x3
andi   x5,  x23, 20
```

- Pode iniciar sub enquanto add está esperando por lw

# Escalonamento dinâmico de CPU



# Renomeação de registradores

- Estações de reserva e buffer de reordenação implementam renomeação de registradores
- Quando uma instrução é disparada para uma estação de reserva
  - Se o operando estiver disponível no banco de registradores ou no buffer de reordenação
    - Copiado para a estação de reserva
    - Não é mais necessário no registrador, pode ser sobreescrito
  - Se o operando ainda não estiver disponível
    - Será fornecido à estação de reserva por uma unidade funcional
    - A atualização do registrador pode não ser necessária

# Especulação

- Prever desvios e continuar disparando instruções
  - Não escrever no banco de registradores até que o resultado do desvio seja determinado
- Especulação de load
  - Evitar tanto atraso no load quanto perda na cache
    - Prever o endereço
    - Prever valor carregado
    - Load antes de completar stores pendentes
    - Copiar valores armazenados para a unidade de load
  - Não finalizar o load até que as especulações sejam esclarecidas

# Por que fazer o escalonamento dinâmico?

- Por que não deixar apenas o compilador escalar o código?
- Nem todas as paradas são previsíveis
  - Por exemplo, falhas de cache
- Nem sempre é possível escalar perto dos desvios
  - O resultado do desvio é determinado dinamicamente
- Diferentes implementações de um ISA têm diferentes latências e dependências

# Disparo de múltiplas instruções funciona?

- Sim, mas nem sempre o desempenho é tão bom
- Programas têm dependências que limitam o paralelismo em nível de instruções
- Algumas dependências são difíceis de eliminar
  - Alias de ponteiro (2 ponteiros apontando para o mesmo endereço)
- Determinados tipos de paralelismo são difíceis de explorar
  - Janela de tempo limitada durante disparo de instruções
- Atrasos de memória e largura de banda limitada
  - Difícil manter o pipeline cheio
- Especulação pode ajudar, se implementado corretamente

# Eficiência energética

- A complexidade do escalonamento dinâmico e especulações aumentam o consumo
- Vários núcleos mais simples pode ser melhor

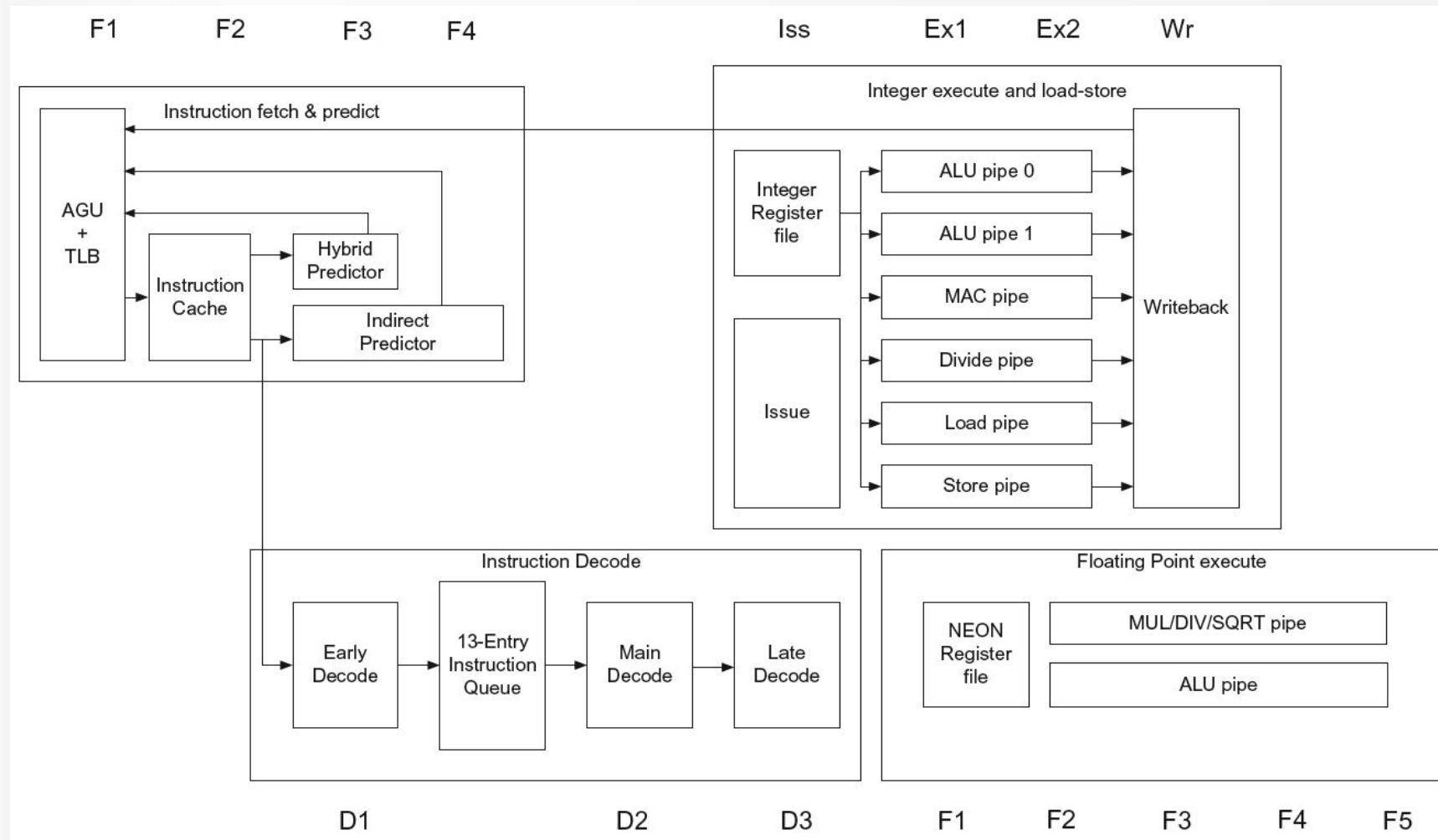
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/Speculation	Cores/Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

# Pipelines do ARM Cortex-A53 e do Intel Core i7

Referência: Capítulo 4, Seção 4.12.

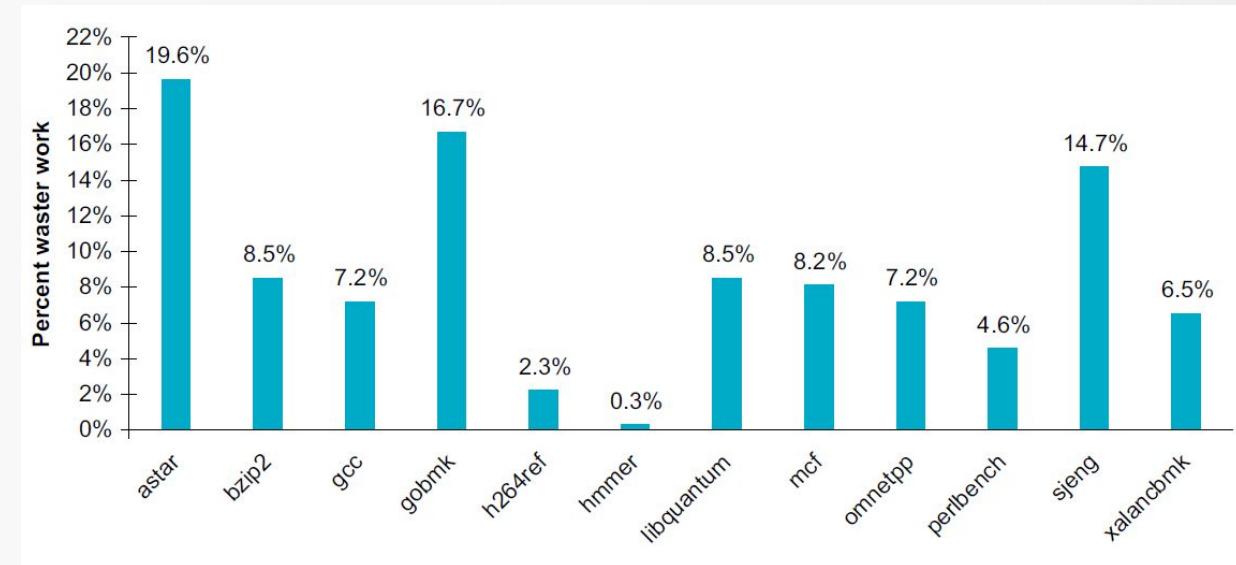
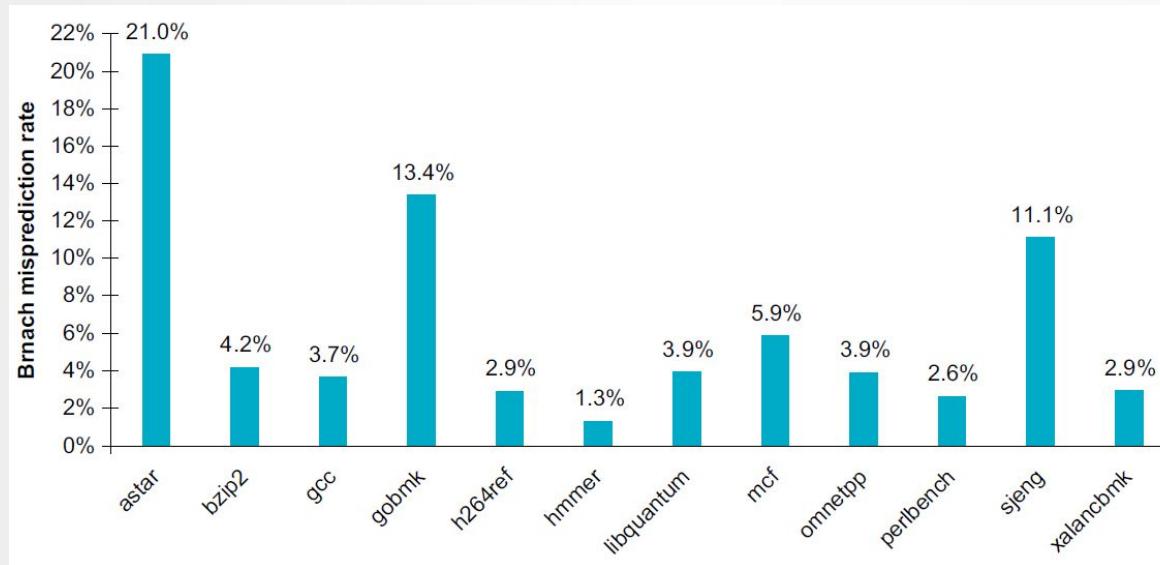
Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Pipeline do ARM Cortex-A53

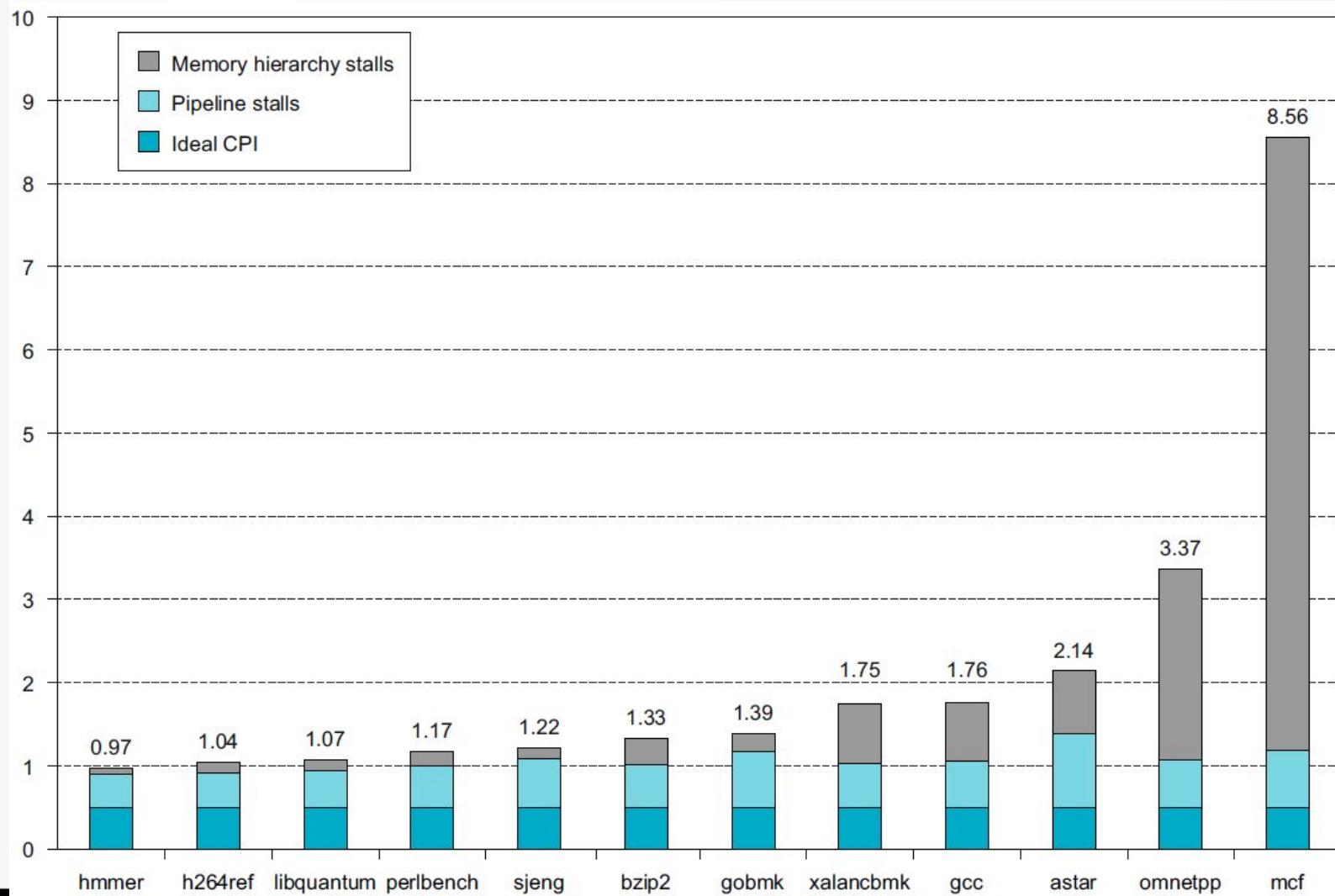


# Desempenho do ARM Cortex-A53

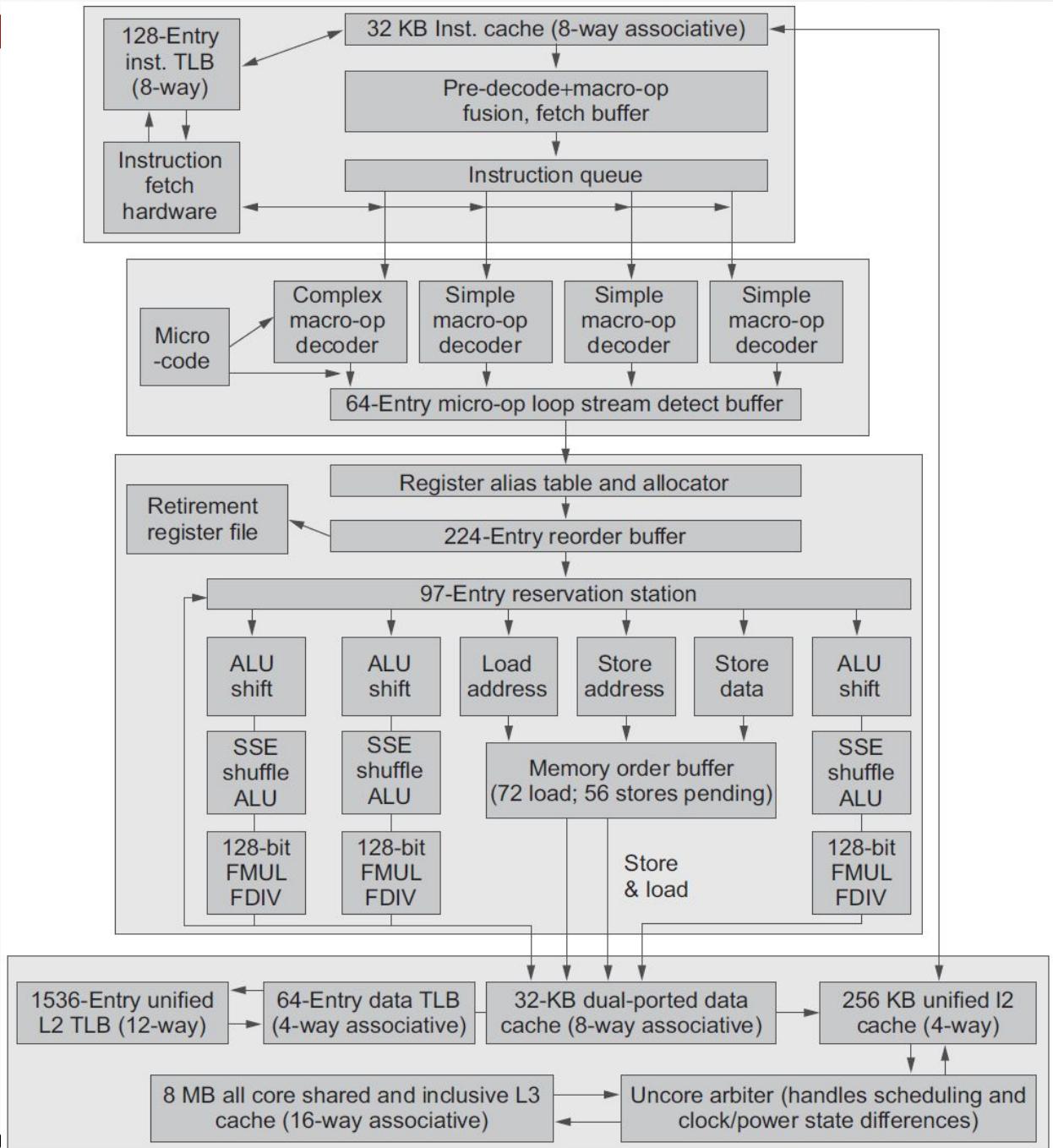
- Taxa de erros do preditor de desvios
- Trabalho desperdiçado



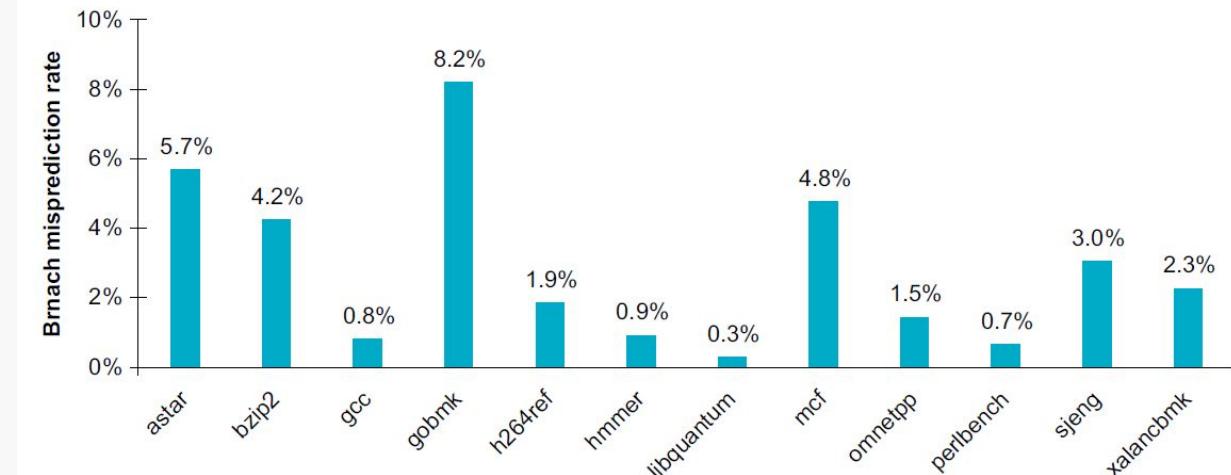
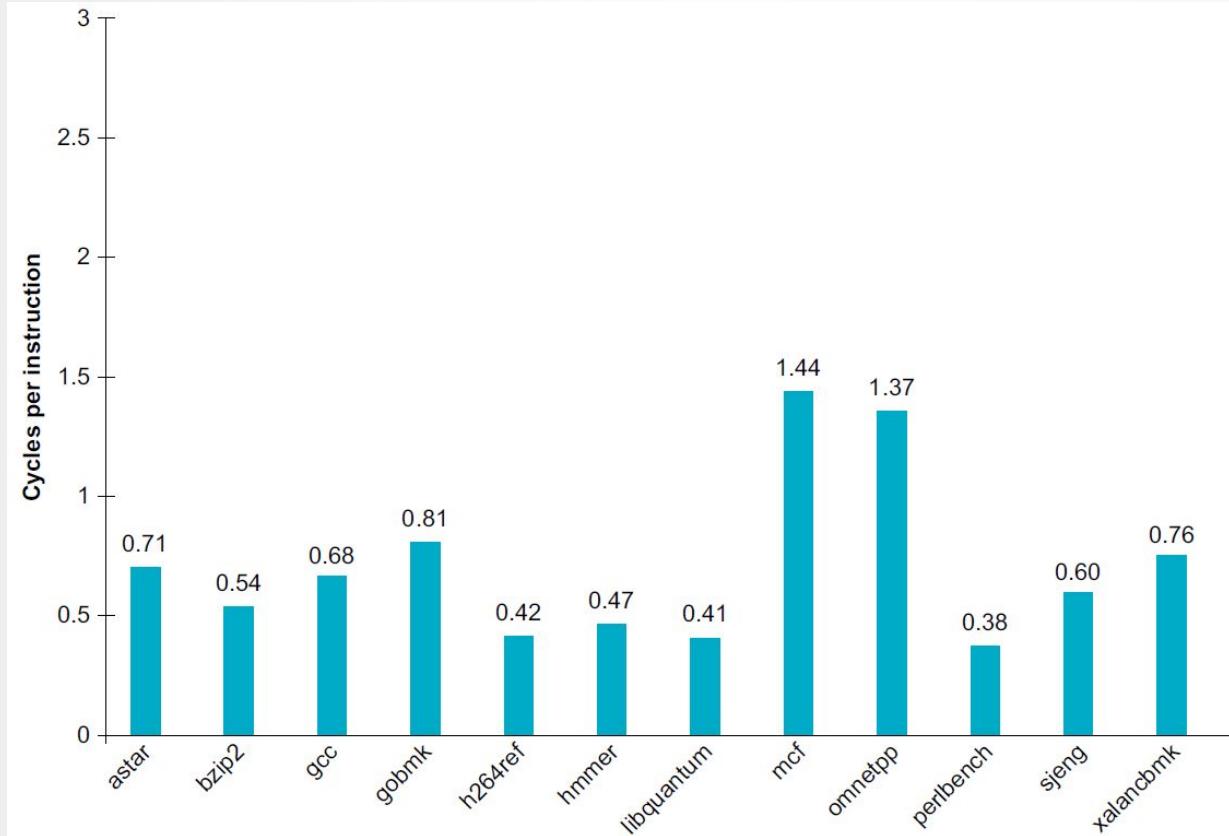
# Desempenho do ARM Cortex-A53



# Pipeline do Core i7



# Desempenho do Core i7



# Paralelismo em Nível de Instrução e Multiplicação de Matrizes

Referência: Capítulo 4, Seção 4.13.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Multiplicação de matrizes

- Código C desenrolado e utilizando instruções AVX

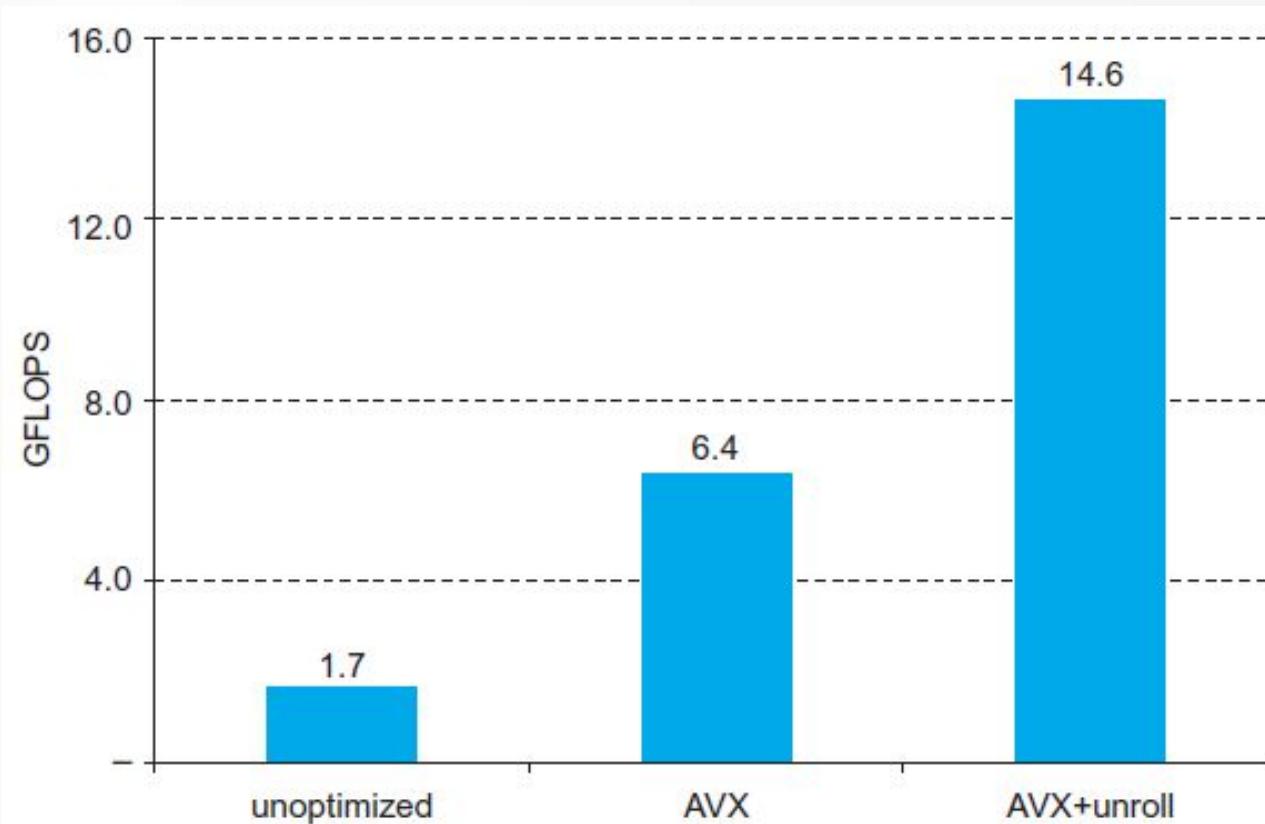
```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm ( int n, double* A, double* B, double* C)
5 {
6   for ( int i = 0; i < n; i+=UNROLL*4 )
7     for ( int j = 0; j < n; j++ ) {
8       __m256d c[4];
9       for ( int x = 0; x < UNROLL; x++ )
10         c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12     for( int k = 0; k < n; k++ )
13     {
14       __m256d b = _mm256_broadcast_sd(B+k+j*n);
15       for (int x = 0; x < UNROLL; x++)
16         c[x] = _mm256_add_pd(c[x],
17                               _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18     }
19
20     for ( int x = 0; x < UNROLL; x++ )
21       _mm256_store_pd(C+i+x*4+j*n, c[x]);
22   }
23 }
```

# Multiplicação de matrizes

- Código assembly do x86 com instruções AVX

```
1 vmovapd (%r11),%ymm4          # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                # register %rax = %rbx
3 xor %ecx,%ecx                # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3      # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2      # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1      # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0 # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5    # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4     # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3     # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                # register %rax = %rax + %r8
16 cmp %r10,%rcx                # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2     # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1     # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>         # jump if not %r8 != %rax
20 add $0x1,%esi                # register %esi = %esi + 1
21 vmovapd %ymm4,(%r11)          # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)      # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)      # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)      # Store %ymm1 into 4 C elements
```

# Impacto de desempenho



# Falácia e Armadilhas

Referência: Capítulo 4, Seção 4.15.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.



# Falácias

- O pipelining é fácil
  - A ideia básica é fácil
  - A dificuldade está nos detalhes
    - Por exemplo, detectar dependências de dados
- O pipelining é independente da tecnologia
  - Por que pipelining é uma técnica tão usada?
    - Mais transistores tornam técnicas mais avançadas viáveis
  - O projeto ISA relacionado ao pipeline deve levar em consideração as tendências de tecnologia
    - Por exemplo, instruções predicadas
      - Só executam se o predicado é verdadeiro

# Armadilhas

- Um projeto ISA ruim pode tornar o pipelining mais difícil
  - Exemplos
    - Conjuntos de instruções complexos (VAX, IA-32)
      - Sobrecarga significativa para fazer o pipelining funcionar
      - Abordagem micro-op IA-32
    - Modos de endereçamento complexos
      - Efeitos colaterais de atualização de registradores, indireção de memória
    - Resolução de desvios nos estágios finais do pipeline
      - Pipelines avançados têm longos slots de atraso

# Considerações Finais

Referência: Capítulo 4, Seção 4.15.

Computer Organization and Design RISC-V Edition: The Hardware Software Interface, David A. Patterson, John L. Hennessy. Editora Morgan Kaufmann, 2<sup>a</sup> Edição, 2020.

# Observações Finais

- ISA influencia o projeto do caminho de dados e controle
- O caminho de dados e o controle influenciam o projeto do ISA
- Pipelining melhora o rendimento da instrução usando paralelismo
  - Mais instruções concluídas por segundo
  - Latência para cada instrução não é reduzida
- Dependências: estruturais, dados e controle
- Disparo de múltiplas instruções e escalonamento dinâmico
  - Dependências limitam o paralelismo
  - A complexidade limitada pelo consumo energético