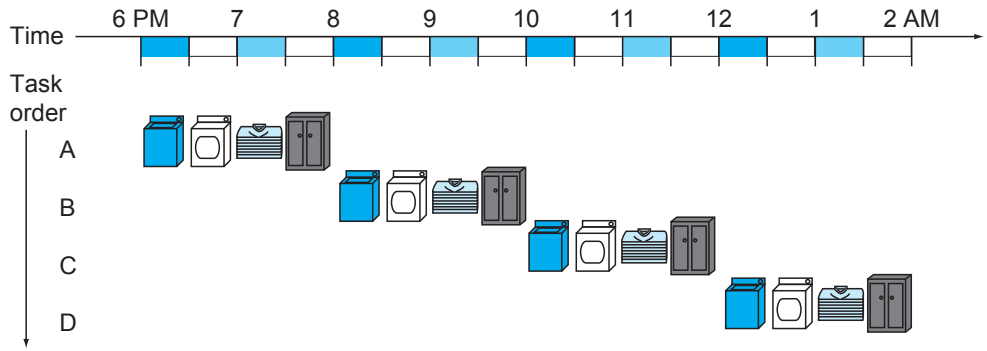
Lavagem de roupa

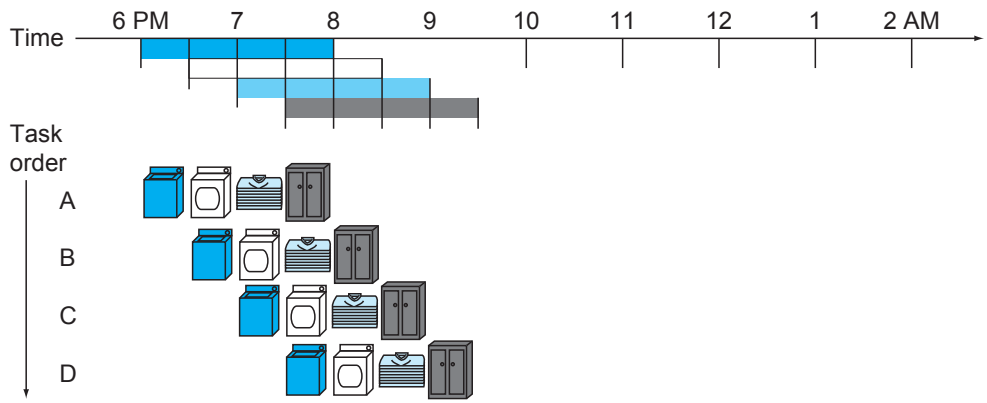
4 passos: lavar, secar, passar, arrumar

Processamento sequencial



Uma carga de roupa leva 2 horas, quatro cargas levam 8 horas

Princípio da linha de montagem



Uma carga de roupa leva 2 horas, quatro cargas levam 3.5 horas

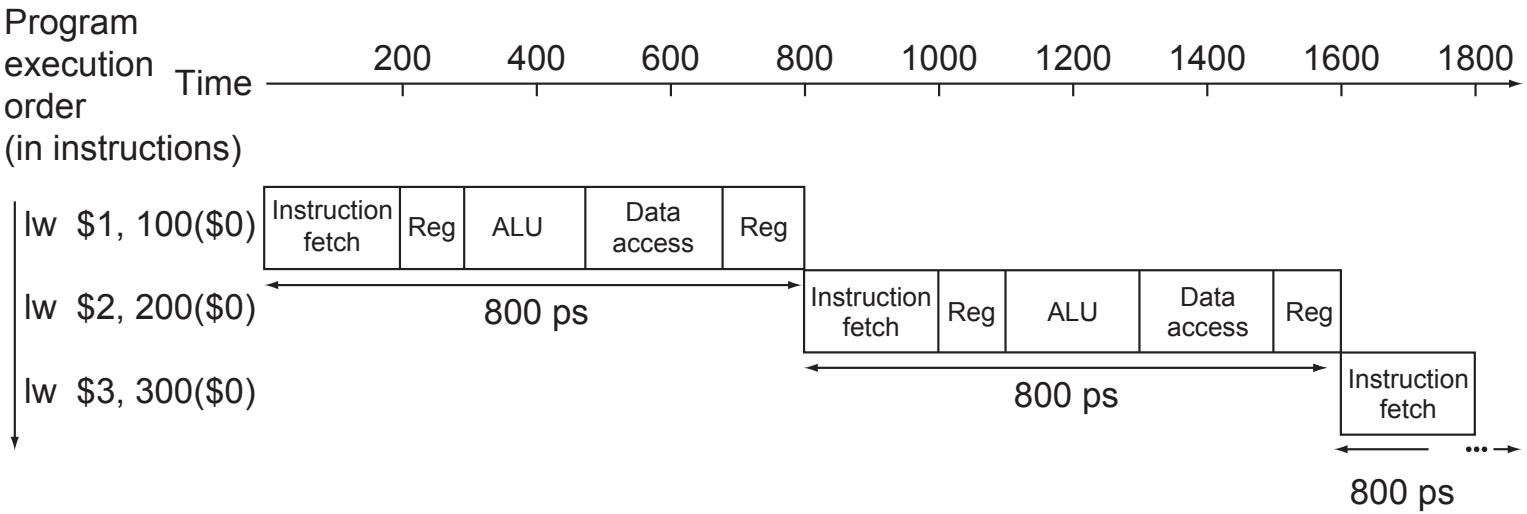
Comparação

	Processamento sequencial	Linha de montagem	
Cargas de roupa	*Tempo necessário*		*Speedup*
1	2 horas	2 horas	1.00
2	4 horas	2.5 horas	1.60
3	6 horas	3 horas	2.00
10	20 horas	6.5 horas	3.08
100	200 horas	51.5 horas	3.88
1000	2000 horas	501.5 horas	3.99
Tempo entre o fim de 2 cargas	2 horas	0.5 horas	

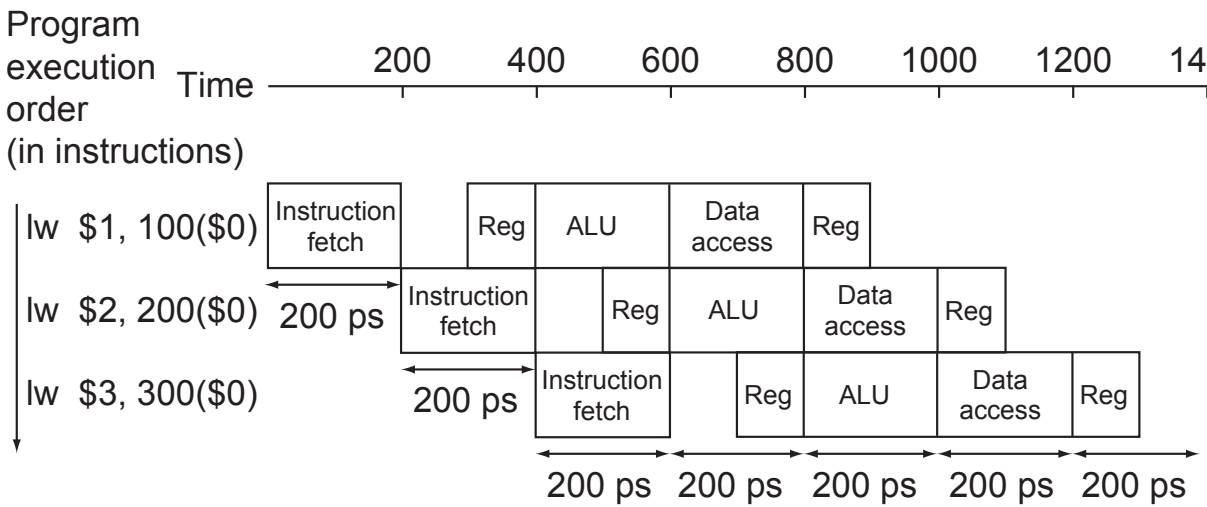
O que melhorou?

Execução de instruções

Sequencial (monociclo)



Em "linha de montagem"



Execução sequencial *vs* "linha de montagem"

	Execução sequencial	Execução linha de montagem
Duração de 1 instrução	800 ps	1000 ps
Tempo entre 2 instruções	800 ps	200 ps

Instruções executadas	*Tempo*		*Speedup*
1	800 ps	1000 ps	0.80
2	1600 ps	1200 ps	1.33
3	2400 ps	1400 ps	1.71
10	8000 ps	2800 ps	2.86
100	80000 ps	20800 ps	3.85
1000	800000 ps	200800 ps	3.98
10^6	800 μs	$\approx$ 200μs	$\approx$ 4.00

Fases de execução no MIPS revisitadas

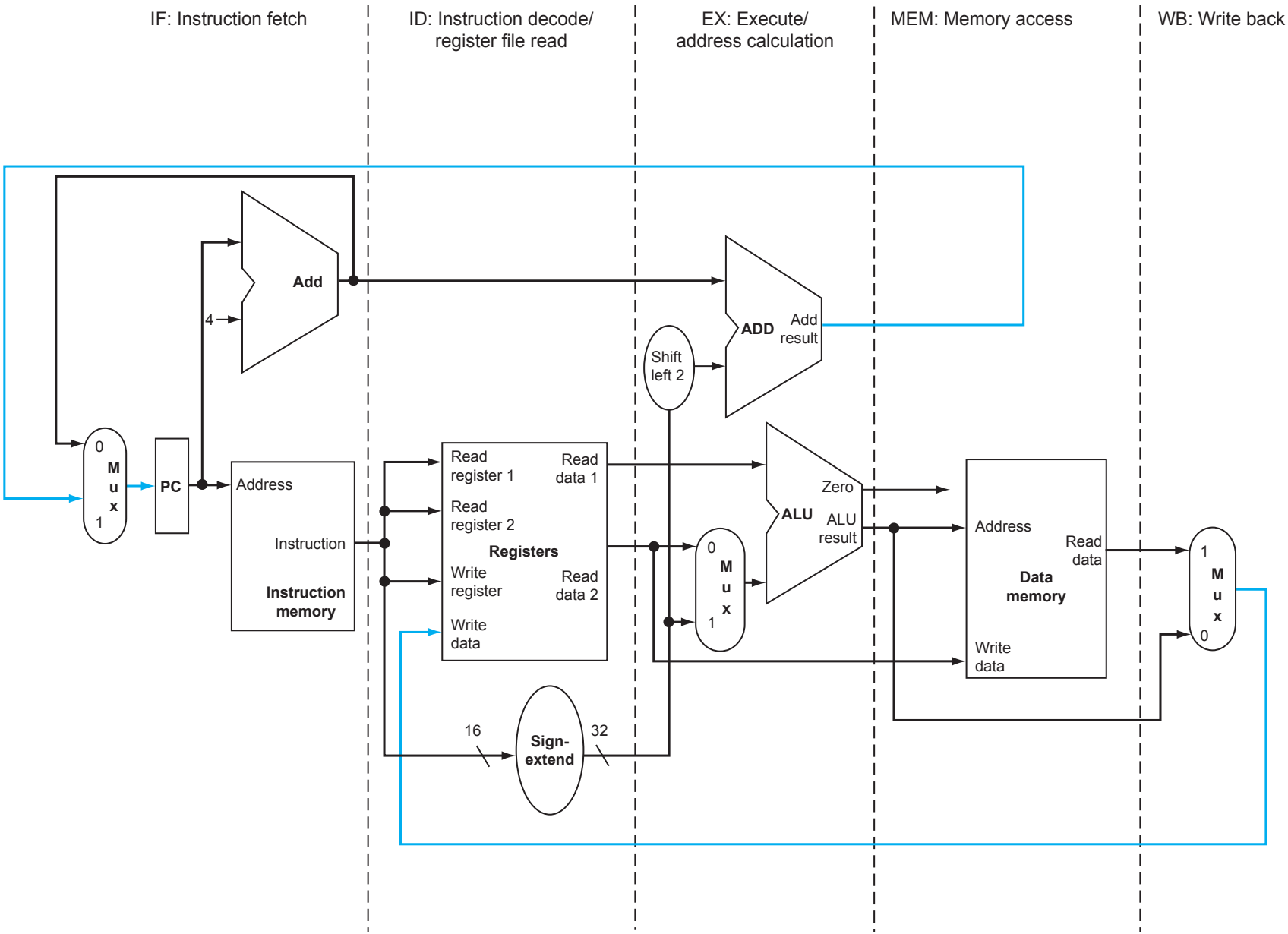
Instruções `add`, `lw` e `beq`

	`add rd, rs, rt`	`lw rt, imm(rs)`	`beq rs, rt, imm`	
IF	Leitura da instrução			
ID	Lê `rs` e `rt`	Lê `rs`	Lê `rs` e `rt`	Descod.
EX	Soma `rs` e `rt`	Soma `rs` e `imm`	Compara `rs` e `rt`	
MEM		Acede à memória		
WB	Escreve `rd`	Escreve `rt`		

Cada fase vai corresponder a um andar do *pipeline*

Andar	**Função**	**Unidade funcional principal**
IF	*Instruction fetch*	Memória de instruções
ID	*Instruction decode*	Banco de registos (leitura)
EX	*Execute*	ALU
MEM	*Memory access*	Memória de dados
WB	*Write back*	Banco de registos (escrita)

Andares do *pipeline* MIPS



Arquitectura MIPS e *pipelines*

Desenhada para ser implementada sobre um *pipeline*

Todas as instruções têm o mesmo tamanho e podem ser lidas no primeiro ciclo no *pipeline* e descodificadas no segundo

Poucos formatos diferentes de instrução e com os registos sempre nas mesmas posições, pelo que é possível iniciar a sua leitura em simultâneo com a descodificação

Só *loads* e *stores* lidam com endereços e não é necessário usar a ALU para cálculo de endereços e para outra operação na mesma instrução

Valores alinhados em memória permitem que um único acesso seja suficiente para os transferir

Instruções só produzem um valor que é escrito no último andar do *pipeline*

Execução *pipelined*

Execução em "linha de montagem"

Idealmente

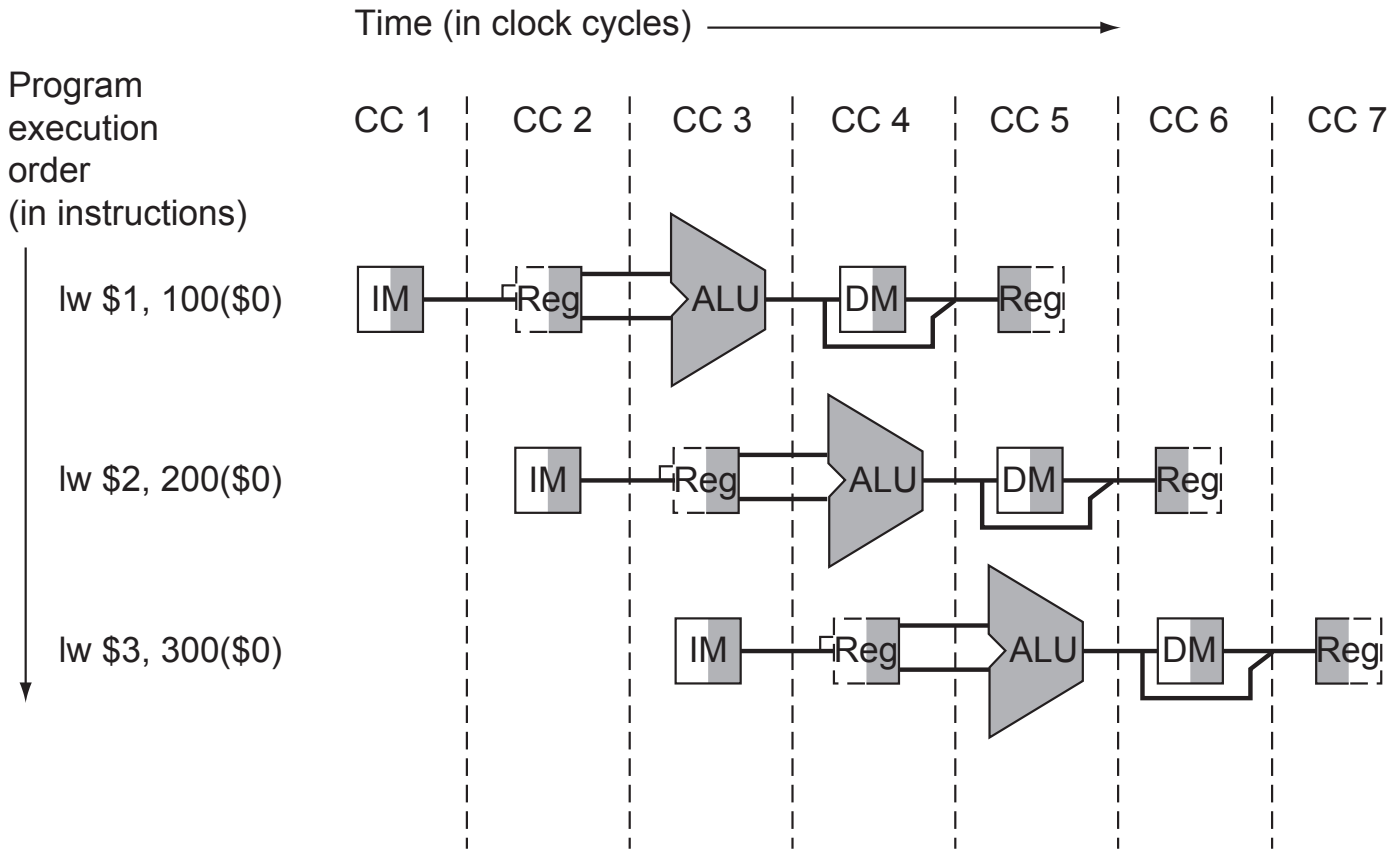
$$\text{Tempo entre instruções}_{\textit{pipelined}} = \frac{\text{Tempo entre instruções}_{\text{não } \textit{pipelined}}}{\text{Número de andares do } \textit{pipeline}}$$

Execução *pipelined* *vs* execução monociclo

- Duração do ciclo de relógio diminui
- Tempo para executar uma instrução não diminui
- Tende a aumentar, sobretudo se os andares do *pipeline* não são perfeitamente equilibrados
- Todas as instruções levam o mesmo tempo
- Aumenta o número de instruções executadas por unidade de tempo (*throughput*)

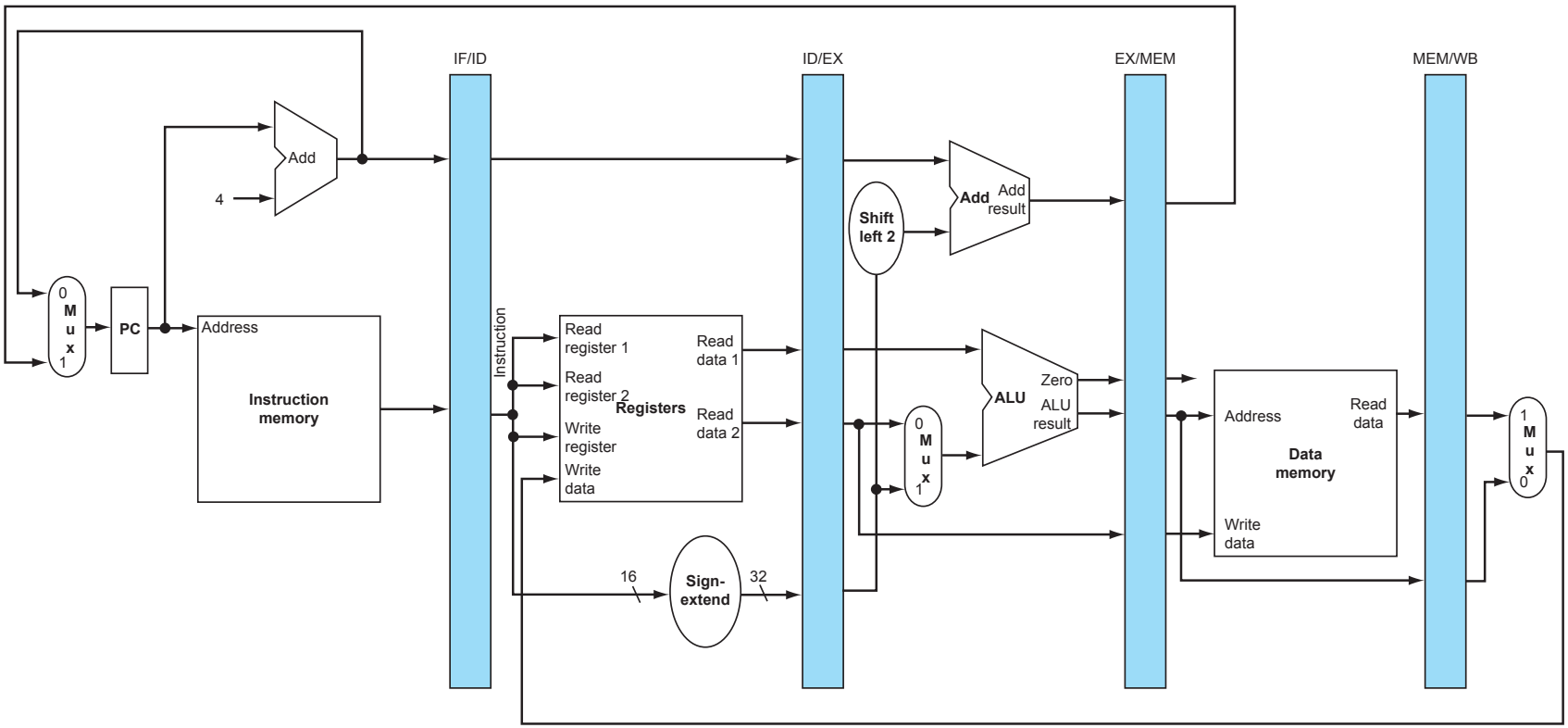
Implementação de um *pipeline*

3 instruções no *pipeline*



Como ter os valores correctos em cada andar do *pipeline*?

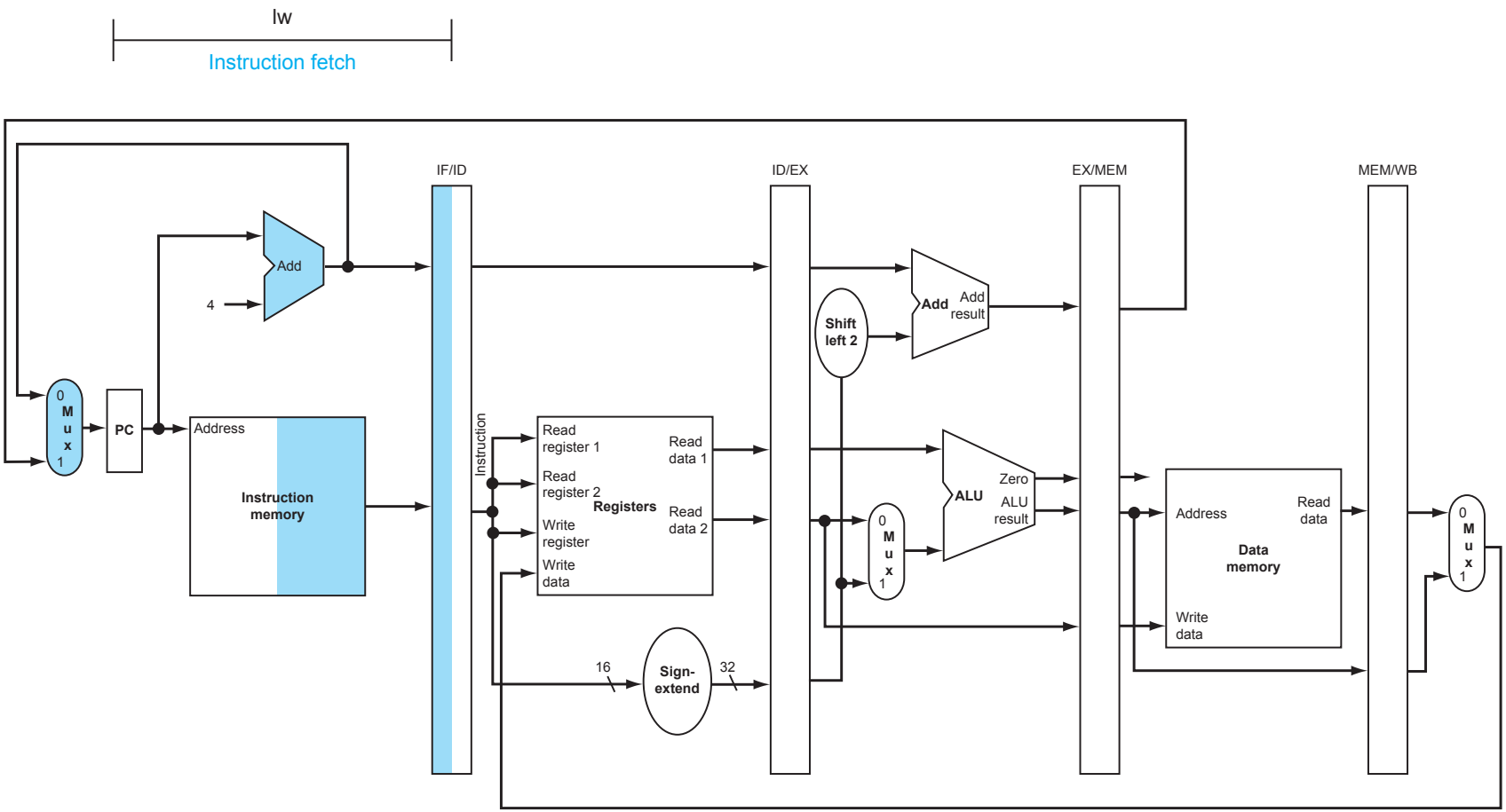
Registos do *pipeline*



Os registos do *pipeline* isolam os andares do *pipeline* e guardam a informação necessária sobre a instrução a executar em cada andar: instrução, conteúdo do(s) registos(s), resultado da ALU, valor lido da memória, ...

`lw` no *pipeline* (1º ciclo)

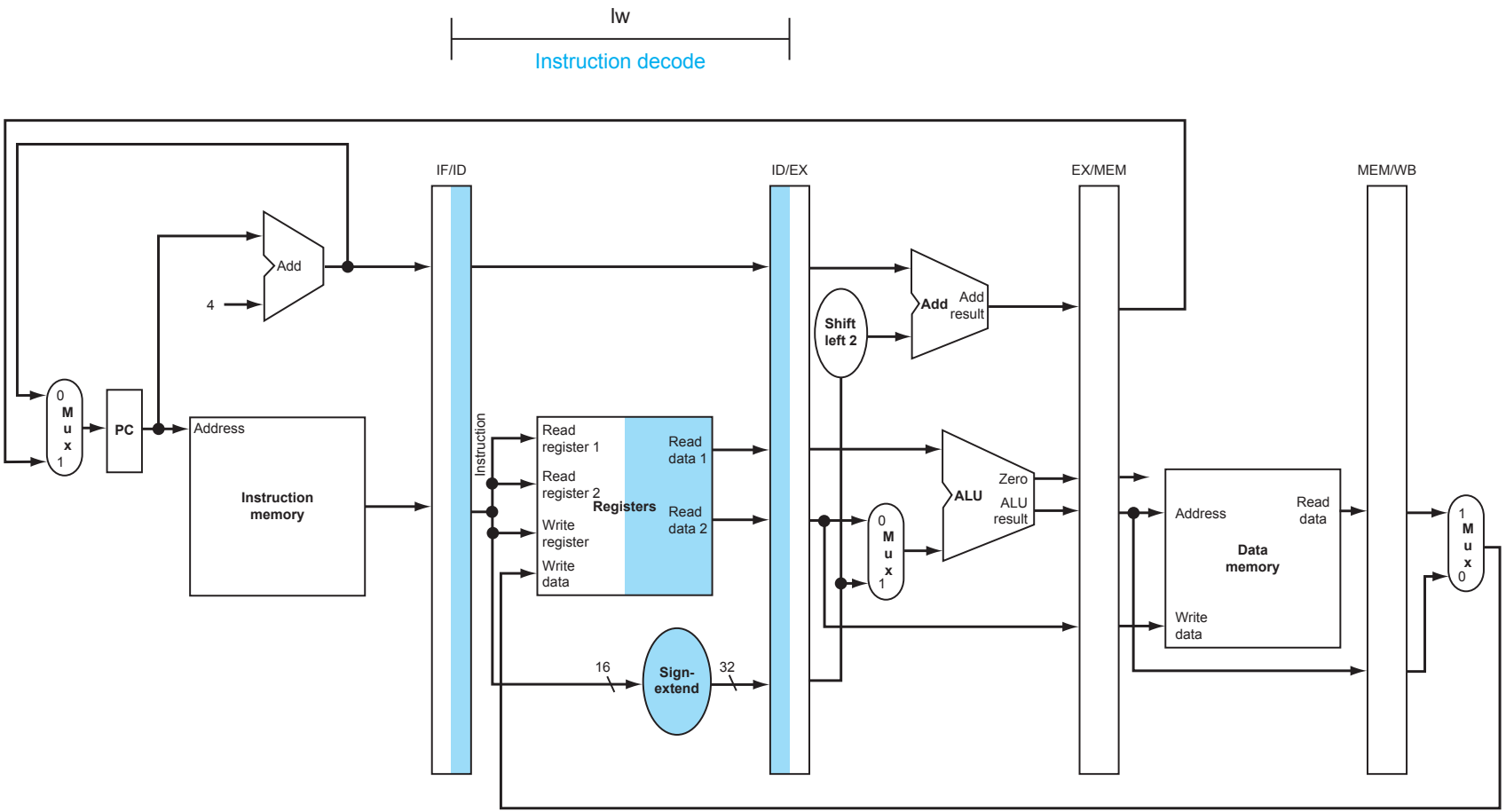
Instruction fetch (IF)



Unidades funcionais activas: mux(PCSrc), PC, somador (PC+4), memória de instruções (leitura), registo IF/ID (escrita)

`lw` no *pipeline* (2º ciclo)

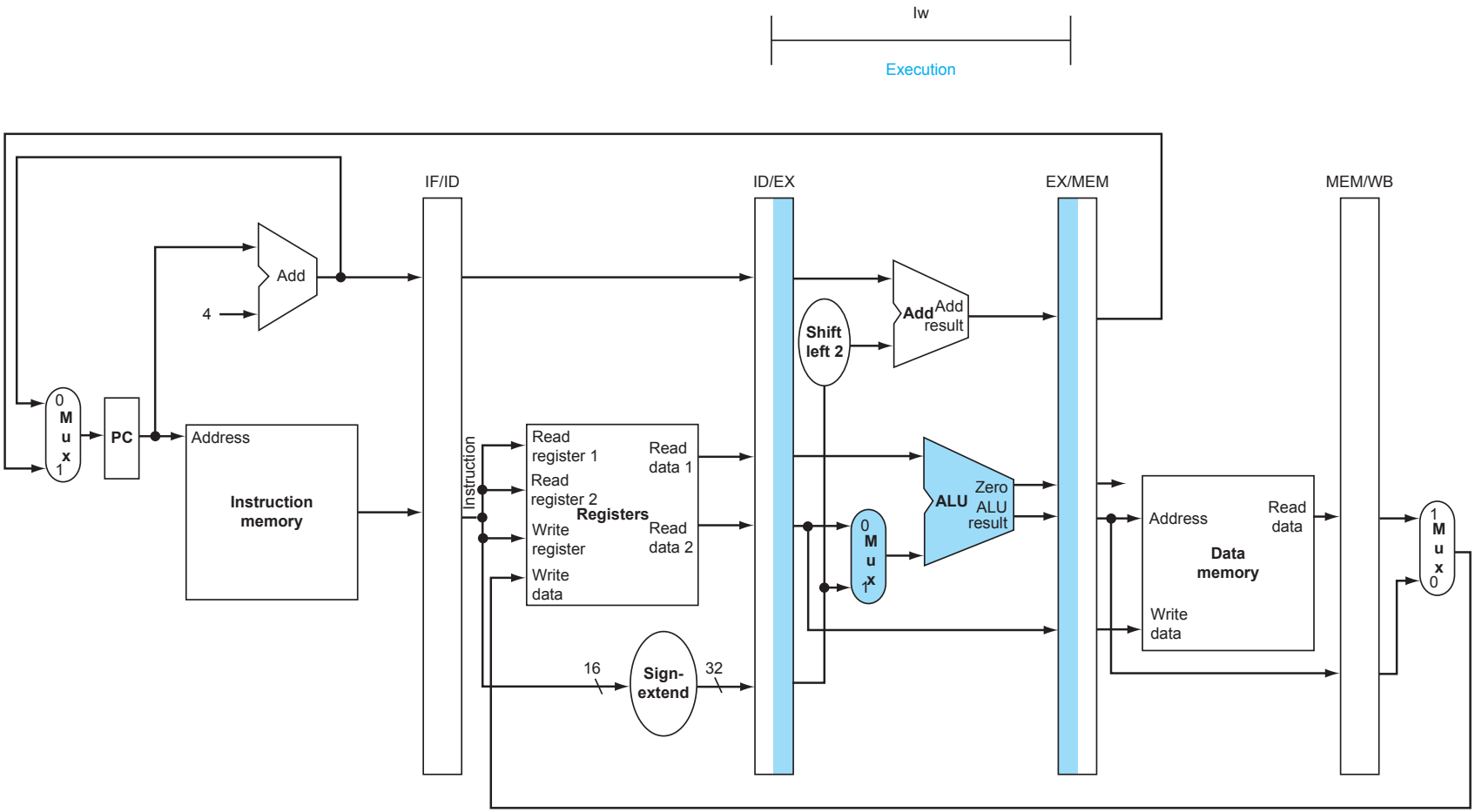
Instruction decode (ID)



Unidades funcionais activas: registo IF/ID (leitura), banco de registos (leitura), extensão com sinal, registo ID/EX (escrita)

`lw` no *pipeline* (3º ciclo)

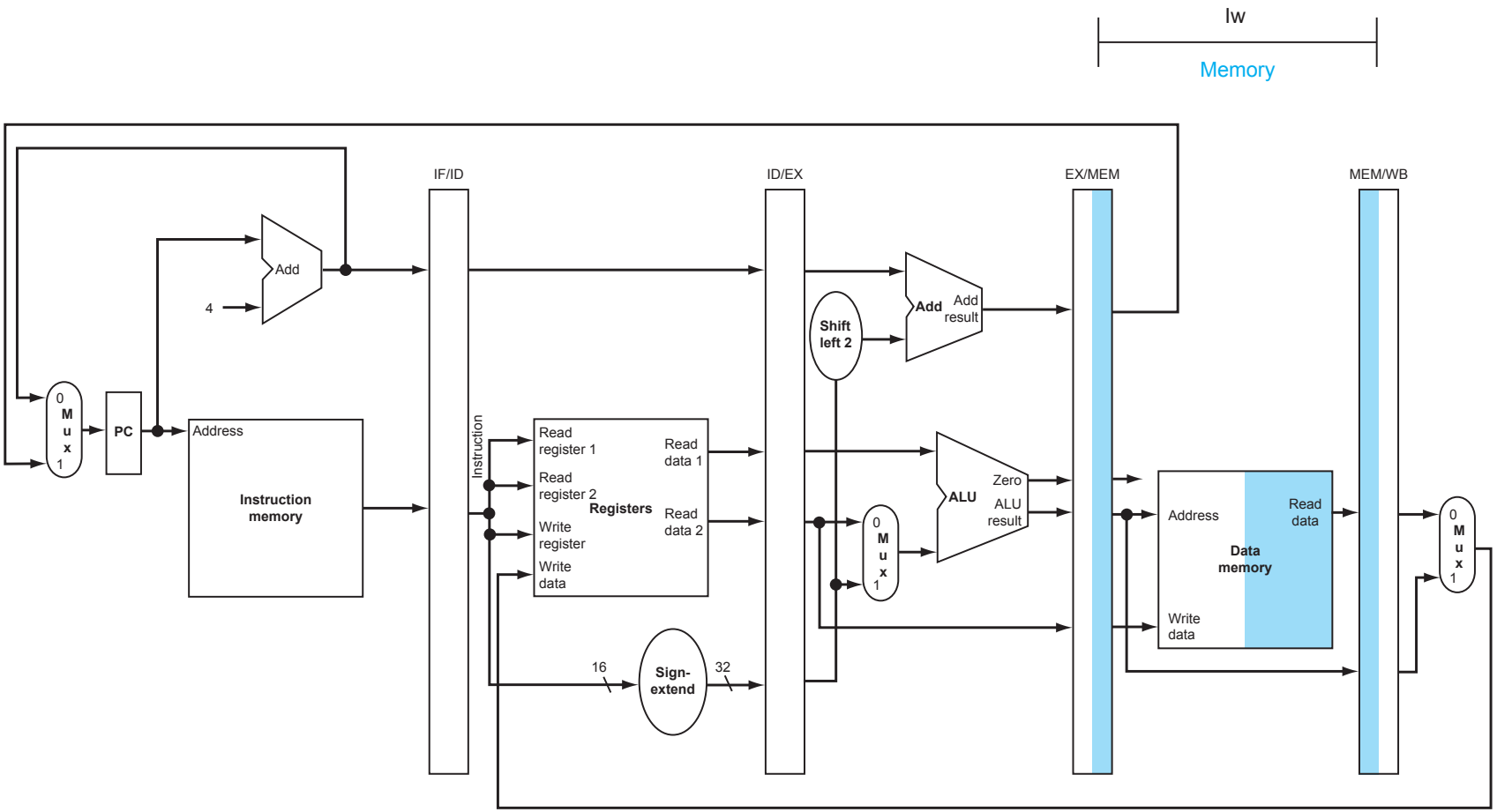
Execute (EX)



Unidades funcionais activas: registo ID/EX (leitura), mux(ALUSrc), ALU, registo EX/MEM (escrita)

`lw` no *pipeline* (4º ciclo)

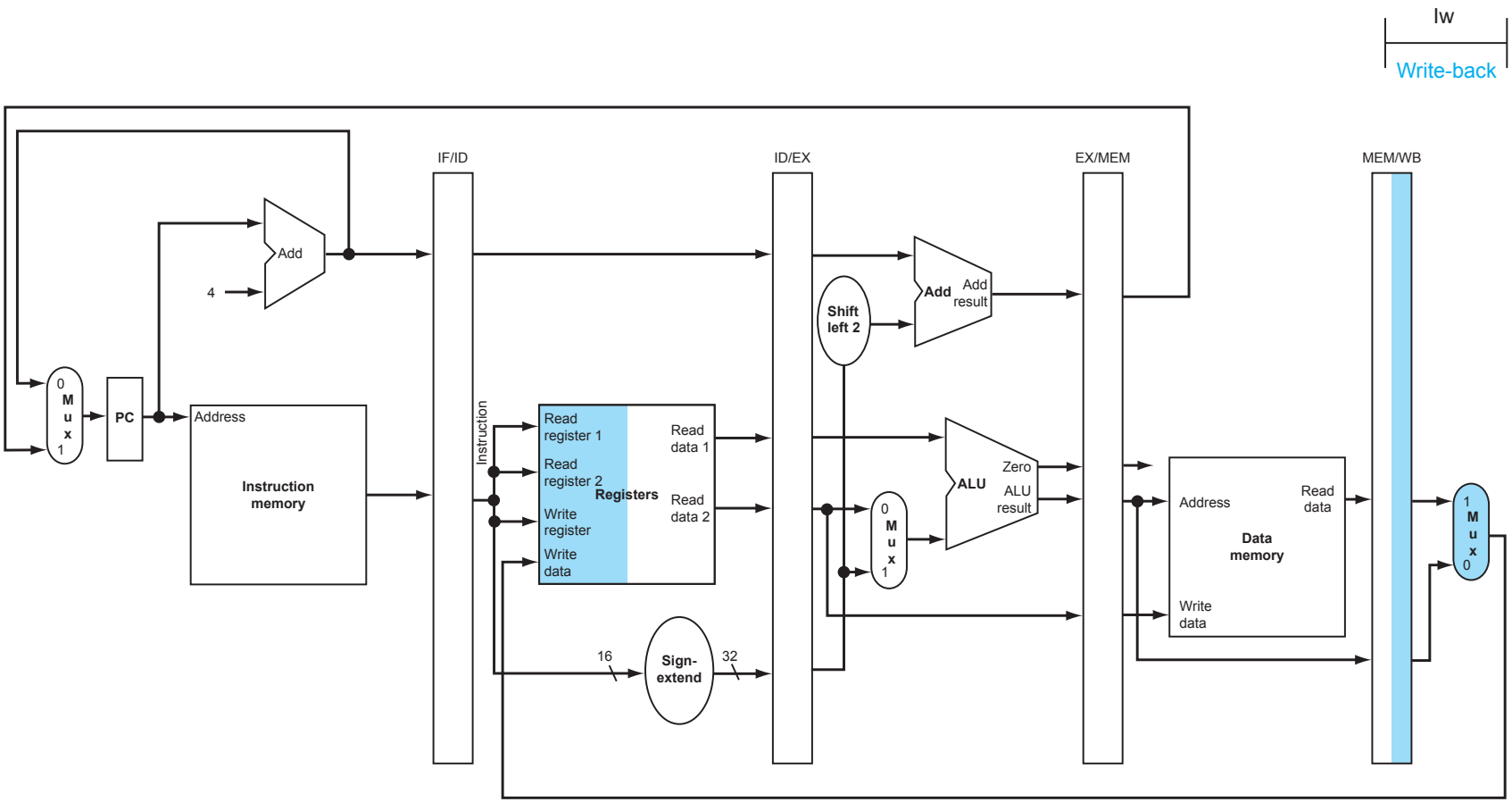
Memory access (MEM)



Unidades funcionais activas: registo EX/MEM (leitura), memória de dados (leitura), registo MEM/WB (escrita)

`lw` no *pipeline* (5º ciclo)

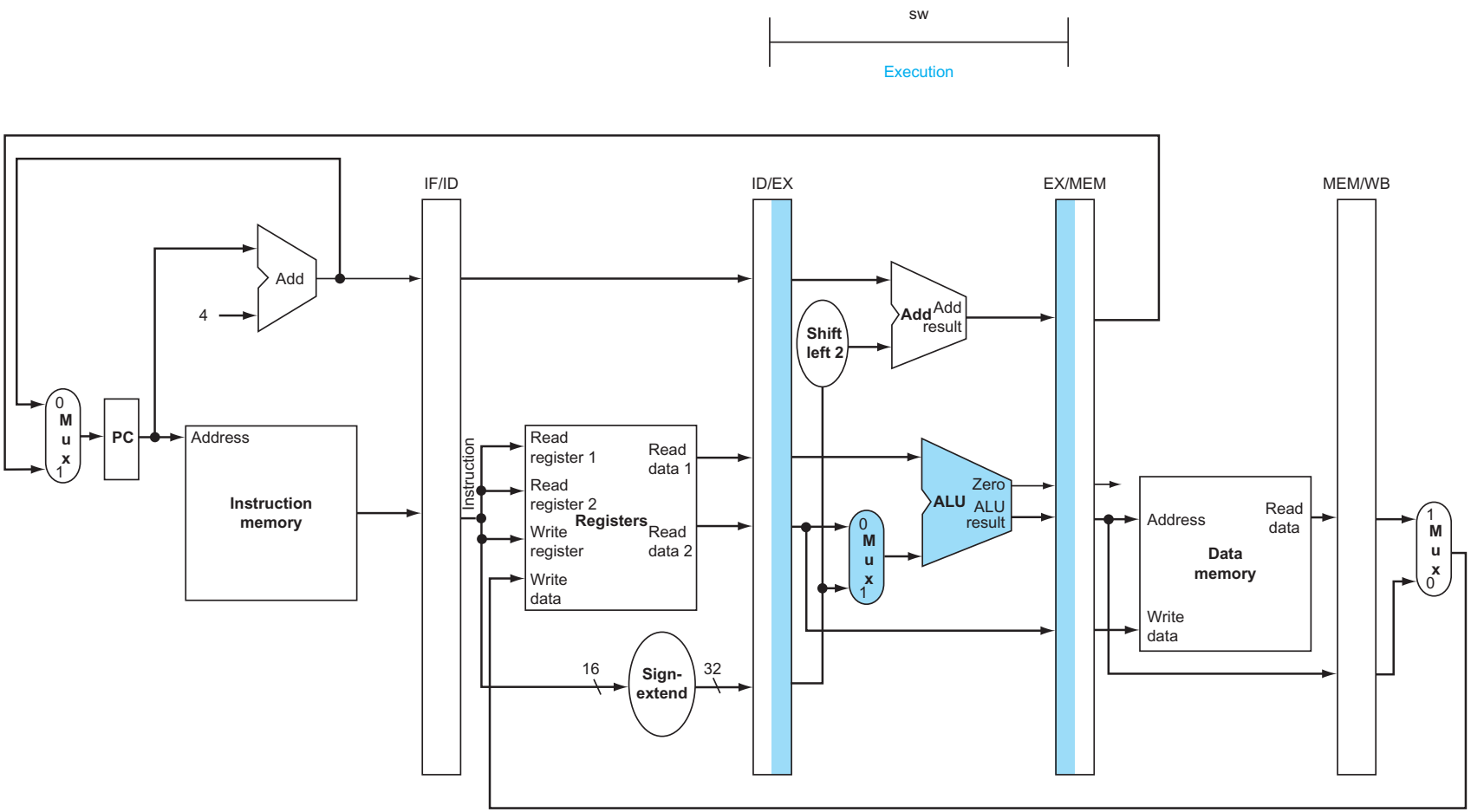
Write back (WB)



Unidades funcionais activas: registo MEM/WB (leitura), mux(MemtoReg), banco de registos (escrita)

`sw` no *pipeline* (3º ciclo)

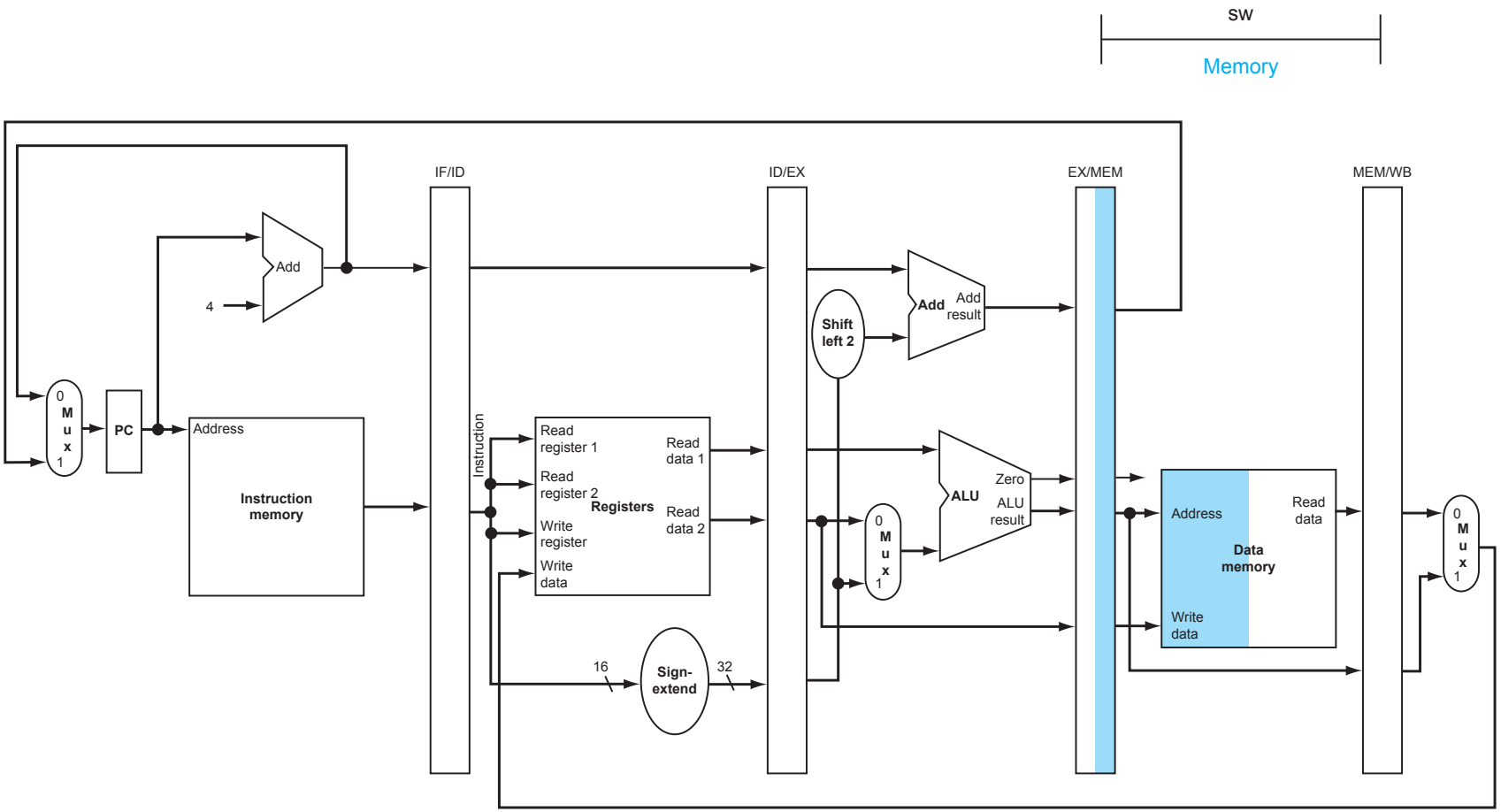
Execute (EX)



Unidades funcionais activas: registo ID/EX (leitura), mux(ALUSrc), ALU, registo EX/MEM (escrita, inclui conteúdo de `rt`)

`sw` no *pipeline* (4º ciclo)

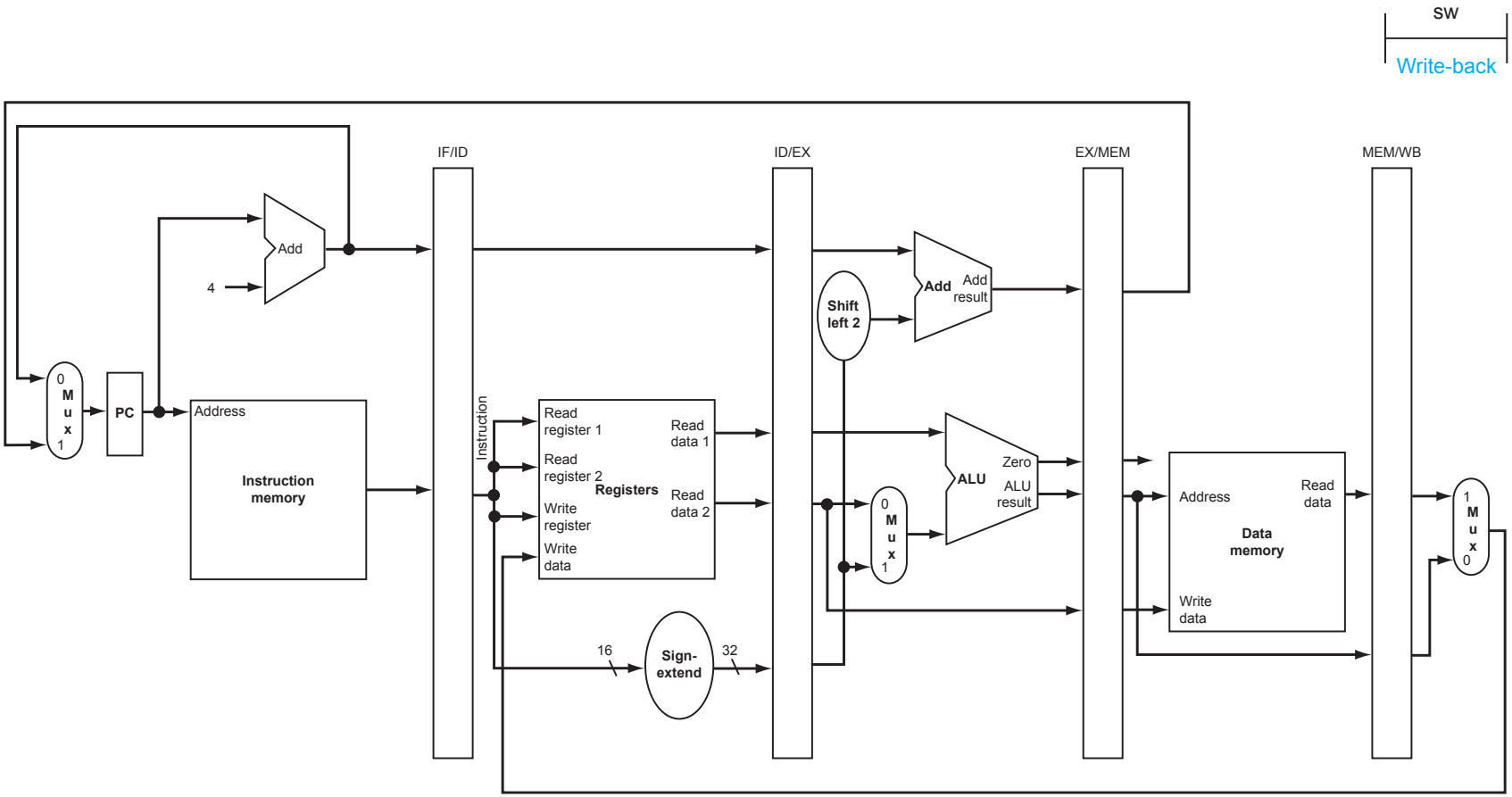
Memory access (MEM)



Unidades funcionais activas: registo EX/MEM (leitura), memória de dados (escrita)

`sw` no *pipeline* (5º ciclo)

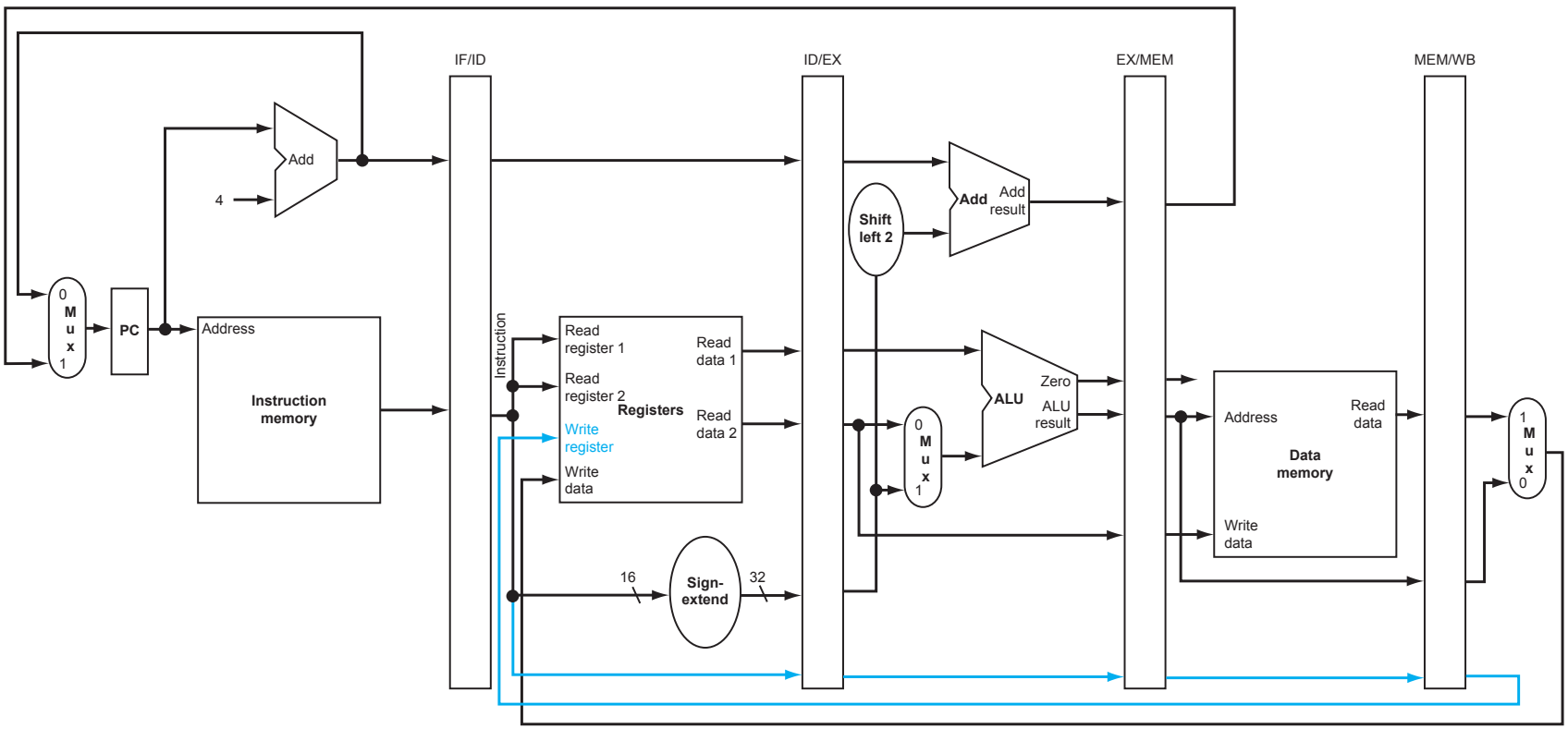
Write back (WB)



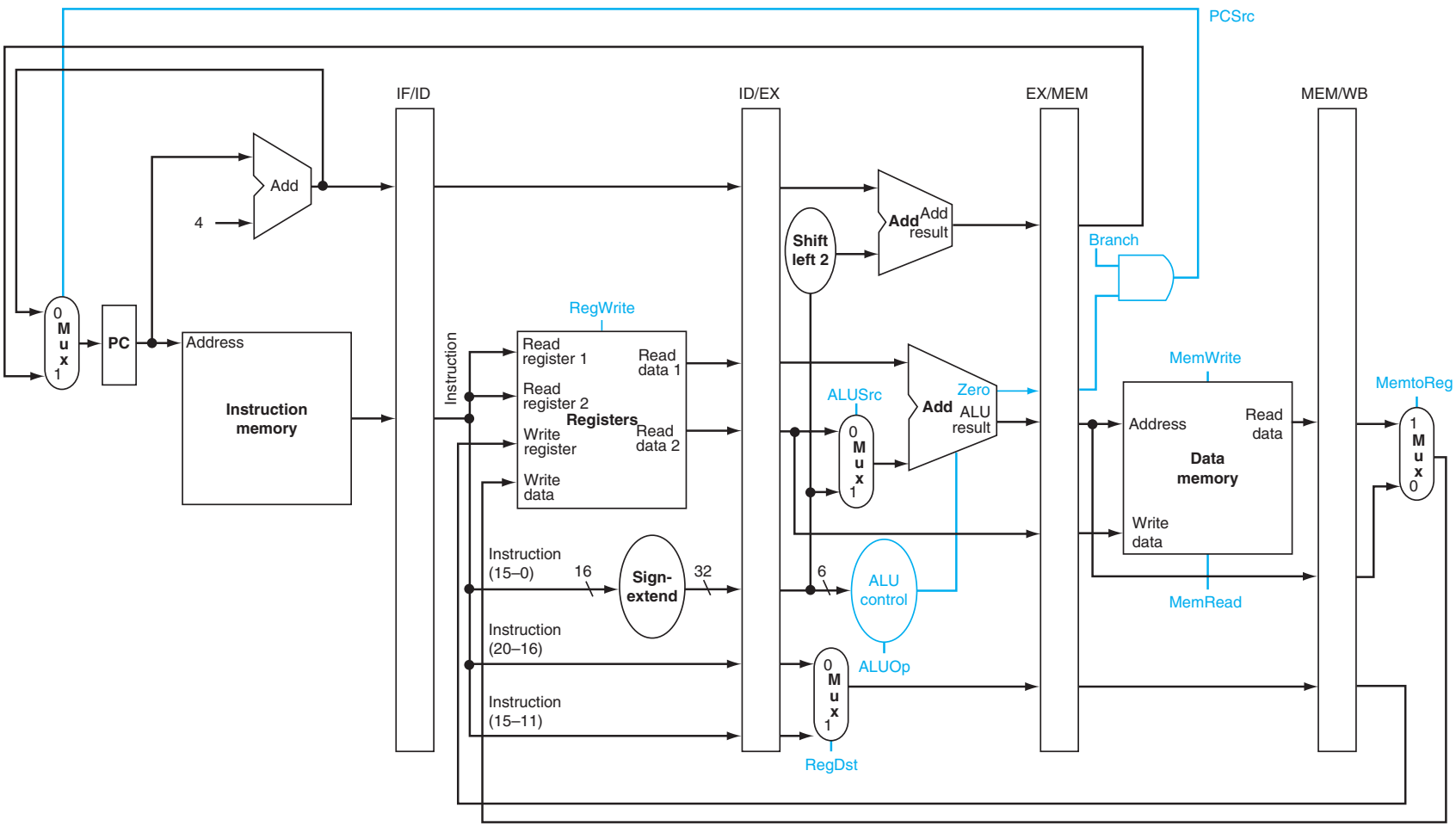
Unidades funcionais activas: *nenhuma*

Correcção do andar WB

O registo a escrever é o da instrução que está no andar WB



Sinais de controlo no *pipeline* (1)



Sinais de controlo no *pipeline* (2)

Organização dos sinais pelo andar em que são usados

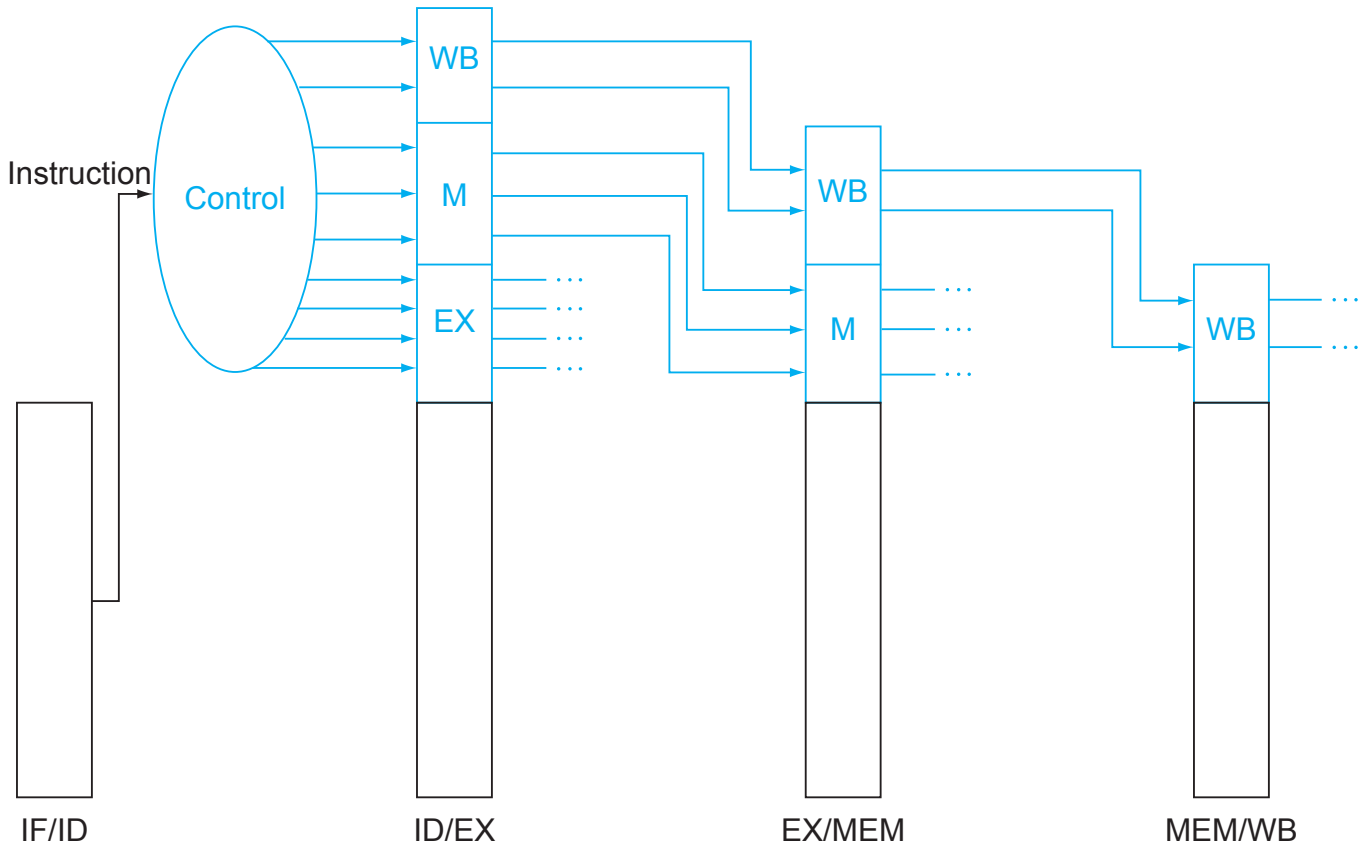
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0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

Em que andar são gerados?

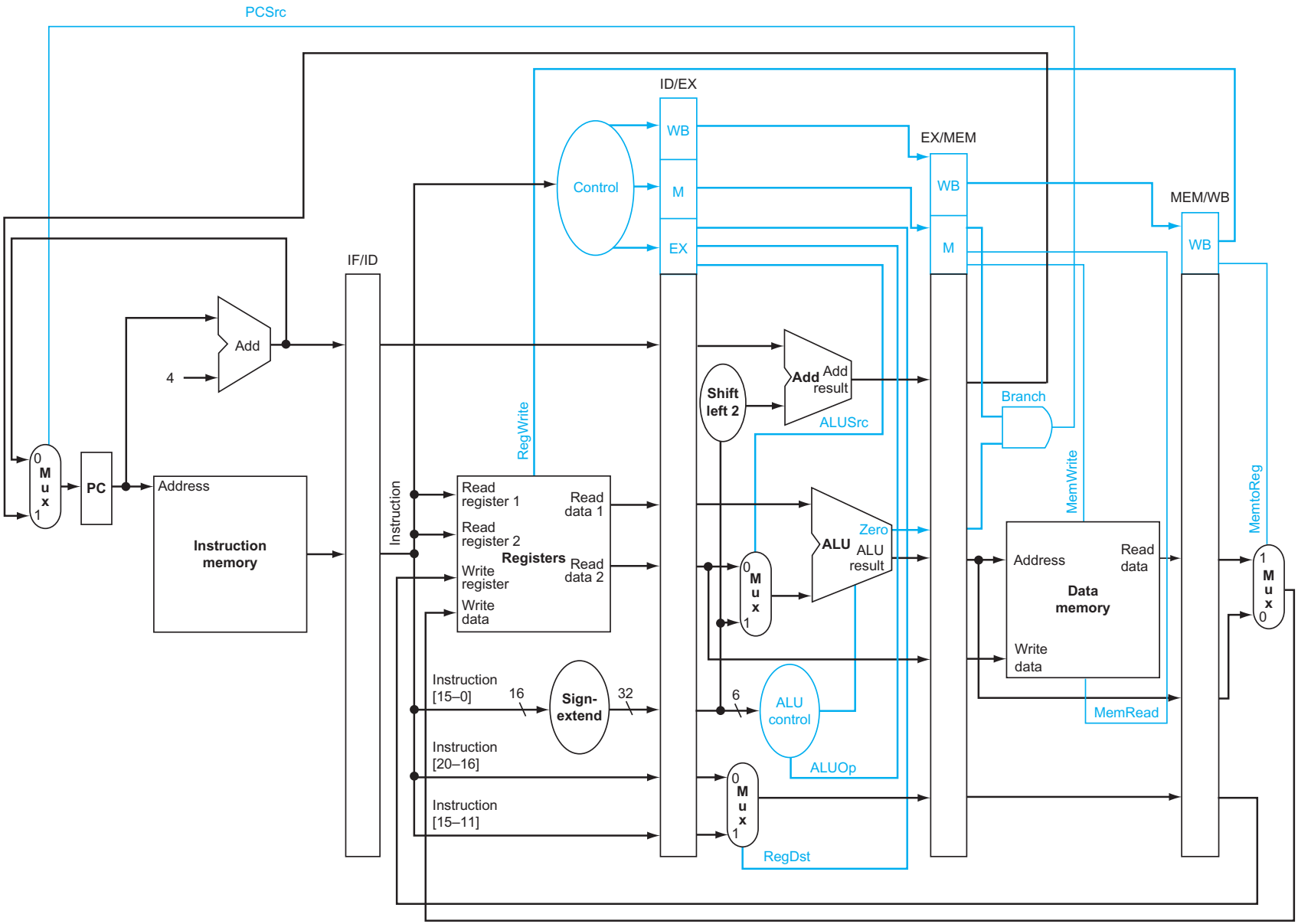
Como se propagam até ao andar onde são necessários?

Propagação do controlo no *pipeline*

Os registos dos andares do *pipeline* também guardam os sinais de controlo da instrução a executar

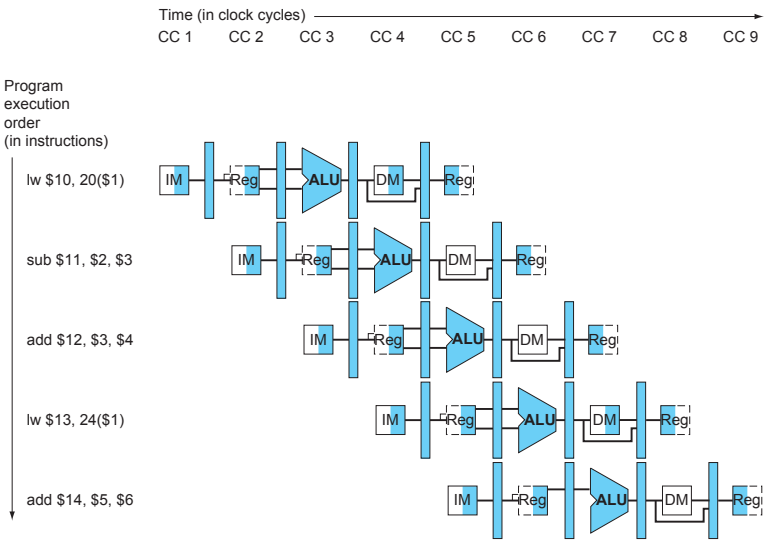


Pipeline com o controlo

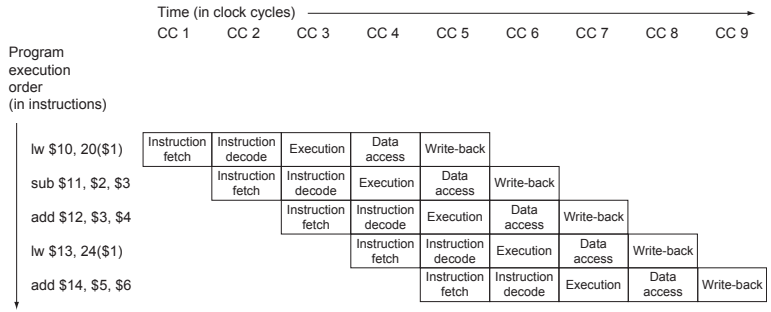


Representações da execução *pipelined*

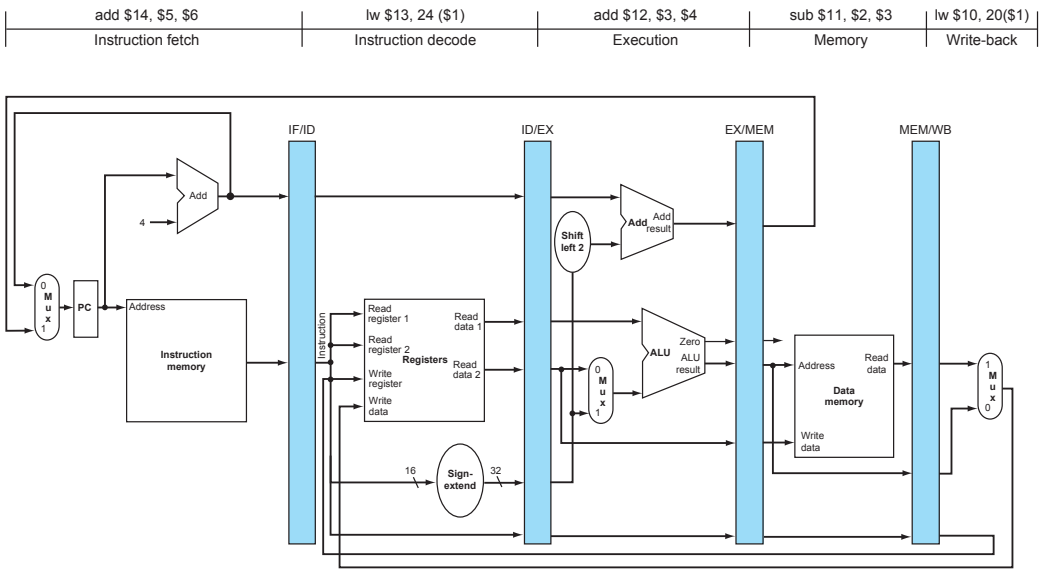
Evol. temporal com unid. funcionais



Evolução temporal tradicional



Estado do *pipeline* num dado ciclo de relógio



Problemas inerentes aos *pipelines*

Conflitos estruturais (*structural hazards*)

Quando não é possível executar uma combinação de instruções no mesmo ciclo

É a razão para o MIPS ter memórias de instruções e de dados

Conflitos de dados (*data hazards*)

Quando uma instrução necessita de um valor produzido por uma instrução anterior e ainda não disponível

Conflitos de controlo (*control* ou *branch hazards*)

É necessário saber o destino de um salto condicional para poder executar a próxima instrução

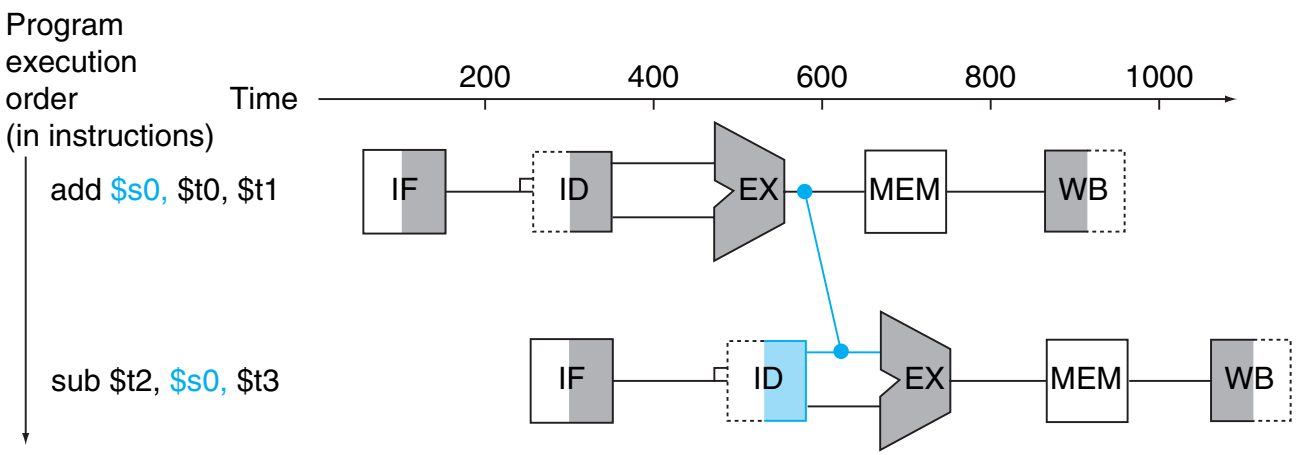
Conflitos de dados (1)

Resolução por *forwarding* (ou *bypassing*)

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

`sub` lê $s0 no mesmo ciclo em que `add` calcula o novo valor

Forwarding de `add` para `sub`



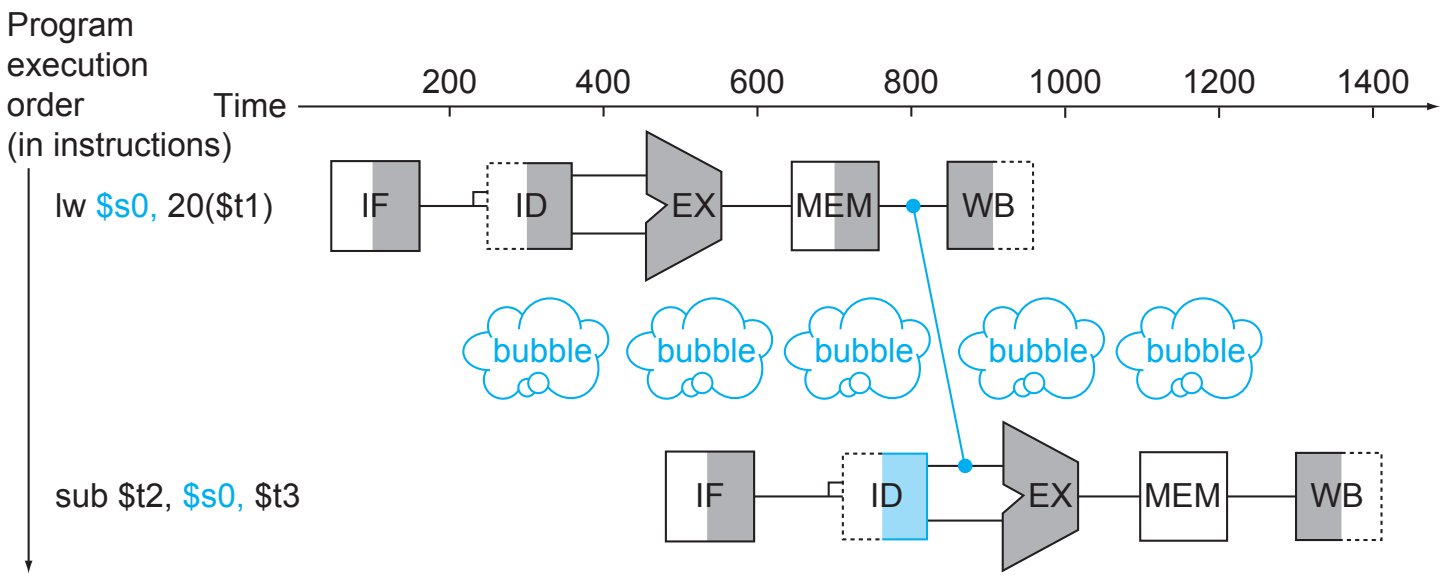
Conflitos de dados (2)

Resolução por atraso (ou *stalling*) do *pipeline*

```
lw  $s0, 20($t1)
sub $t2, $s0, $t3
```

`sub` lê `$s0` antes de `lw` ler o valor da memória

`sub` é atrasada no *pipeline*, através de um *pipeline stall* (ou bolha)



É um exemplo de um conflito de dados *load-use*

Conflitos de dados (3)

Resolução por reordenação das instruções

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
5  lw  $t4, 8($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

Reordenando as instruções, não é necessário atrasar o *pipeline*

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
5  lw  $t4, 8($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```