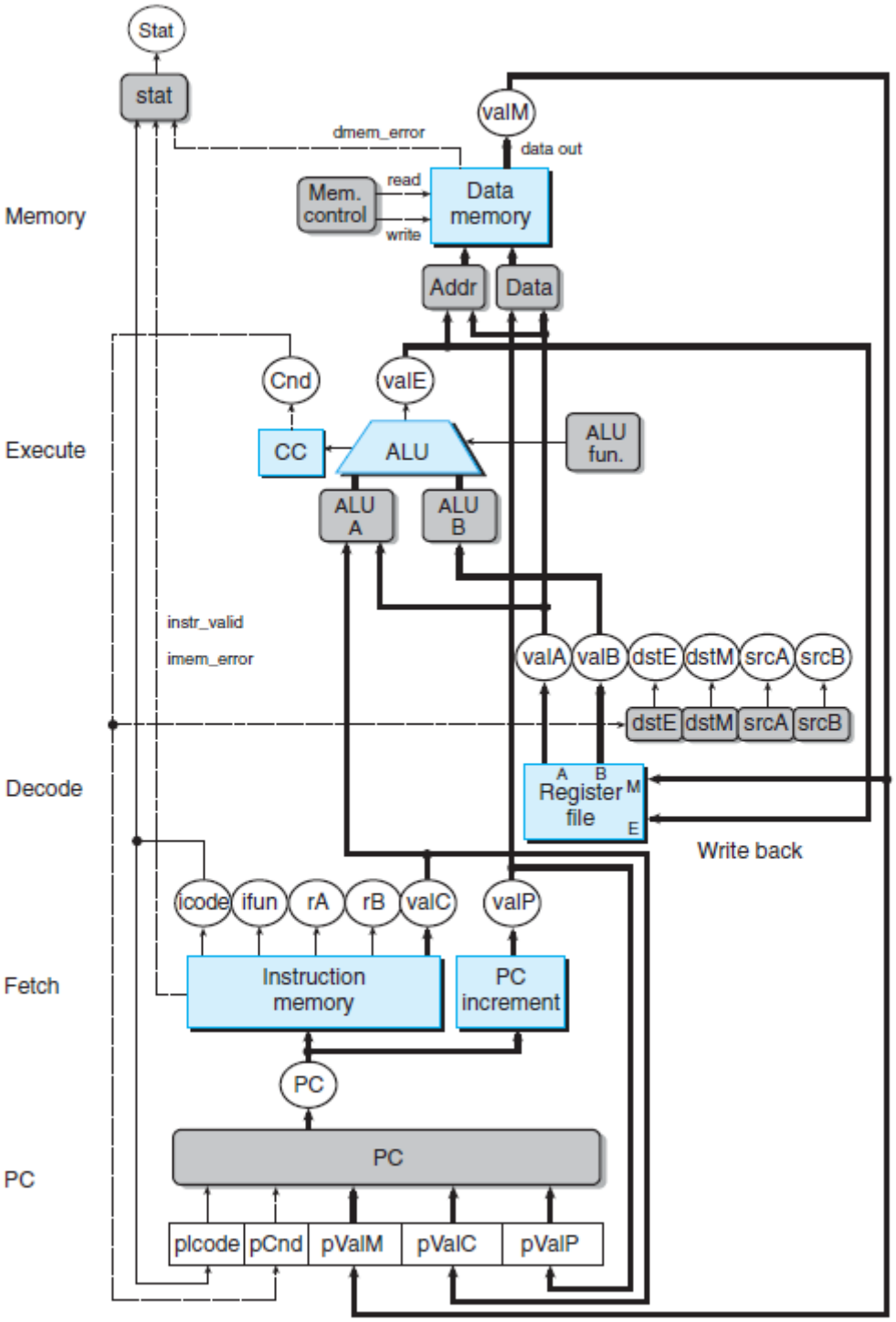


Figure 4.40 **SEQ+ hardware structure.** Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

Y86 PIPE- : Registos usado no encadeamento

F Guarda o **valor previsto para próximo *PC***.

D Está inserido entre os estágios de extração e decodificação. Guarda **informação sobre a última instrução extraída** e que vai ser processada pelo estágio de descodificação.

E Está inserido entre os estágios de decodificação e execução. Guarda **informação sobre a última instrução descodificada e valores lidos do banco de registos**, a qual vai ser processada pelo estágio de execução.

M Está inserido entre os estágios de execução e de memória. Guarda os **resultados da última instrução executada**, os quais vão ser processados pelo estágio de memória. Também guarda **informação sobre as condições de salto e sobre o alvo desses saltos**, que vai permitir processar os saltos condicionais.

W Está inserido entre os estágios de memória e os caminhos de *feedback.* Estes caminhos fornecem (i) os **resultados que vão ser escritos no banco de registos** e (ii) **o endereço de retorno a usar na lógica de seleção do PC**, no caso de estar a terminar o processamento duma instrução ***ret***.

Y86 PIPE- : Estágios do encadeamento

- ***Fetch***
  - Selecionar o PC
  - Ler a instrução
  - Calcular o próximo PC
- ***Decode***
  - Ler os registos genéricos
- ***Execute***
  - Efetuar cálculos com a ALU
- ***Memory***
  - Ler ou escrever na memória
- ***Write Back***
  - Escrever nos registos

Os valores que atravessam o encadeamento não podem saltar estágios.

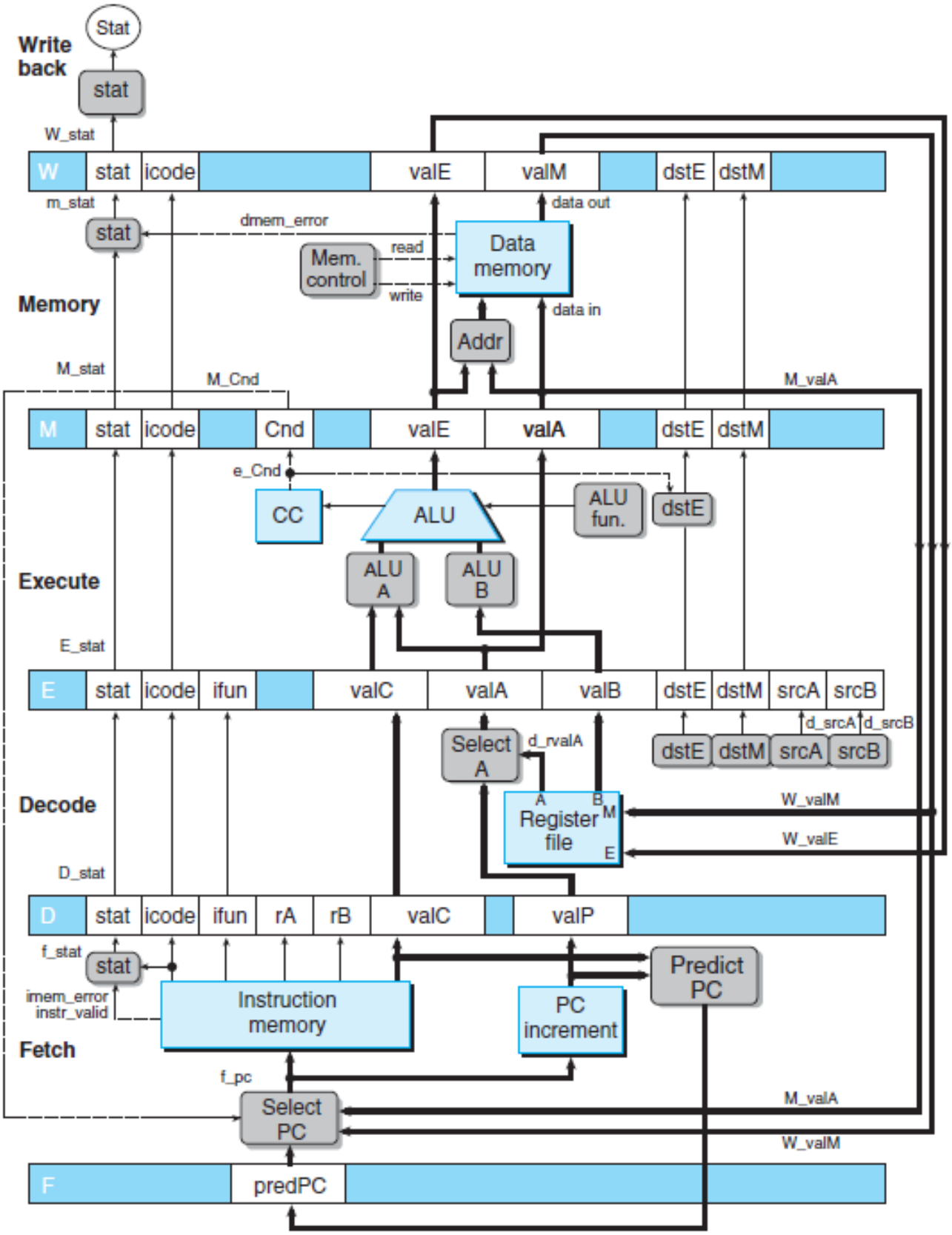
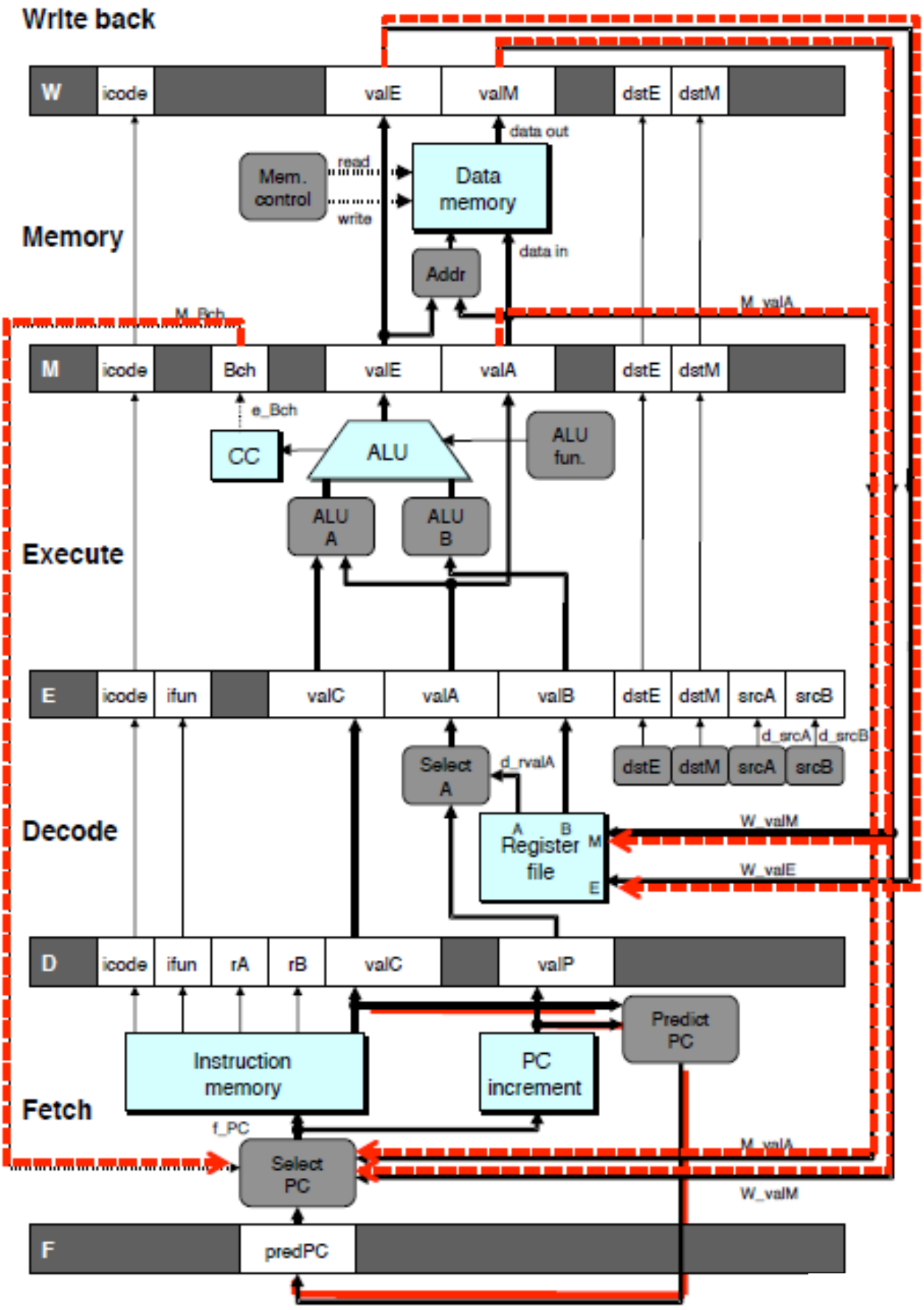


Figure 4.41 **Hardware structure of PIPE–, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

Y86 PIPE- : caminhos de realimentação (*feedback*)

[assinalados a vermelho na próxima figura]

- ***Predict PC***
  - Valor do próximo PC
    - **valP**: em instruções genéricas
    - **valC**: no ***jmp*** e ***call***
- **Salto condicional**
  - É tomado **ou** não é tomado
  - Salta para o alvo (valC) **ou** continua inst. seguinte (valP)
- **Endereço de retorno**
  - Valor lido da memória (**valM**)
- **Atualizar registos**



Caminhos de realimentação no Y86 PIPE-

Prever o próximo PC

- Objetivo do Y86 PIPE- → Iniciar (e concluir) uma instrução por ciclo.
- Para isso é preciso prever qual é o **endereço da próxima instrução a executar**, imediatamente após concluir a extração da instrução atual.
- Esta previsão não é possível nos:
  - **saltos condicionais** → só é possível após a fase de **E**xecução.
  - ***ret*** → só é possível após a fase de **M**emória.

No ***call*** e no ***jmp*** o endereço da próxima instrução a executar será ***valC*** (o valor imediato incluído na instrução).

Nas instruções "regulares" o endereço da próxima instrução a executar será ***valP*** (o endereço da instrução seguinte no código).

Nos **saltos condicionais** podemos optar por prever que

(i) os saltos serão sempre efetuados (***newPC***=***valC***) [Y86 PIPE-] **ou**

(ii) os saltos nunca serão efetuados (***newPC***=***valP***) **ou**

(iii) usar outra estratégia de previsão mais complexa (por ex., **BTFNT**).

Em qualquer dos 3 casos, temos de ter uma forma de **lidar com os casos em que a previsão estiver errada**.

O **Y86 PIPE-** não usa qualquer estratégia de **previsão** do valor do **endereço de retorno** nas instruções ***ret***, apenas **adia o processamento de mais instruções** até que a instrução ***ret*** termine o estágio de **Memória**.

Anomalias no encadeamento

- Devido a **dependências de dados** → quando os dados calculados numa instrução são utilizados numa instrução a seguir.
- Devido a **dependências de controlo** → quando a instrução atual (***jXX***, ***call***, ***ret***) determina a localização da instrução seguinte.
- Uma anomalia resultante de dependências de dados (***hazard* de dados**) pode surgir quando 1, ou os 2, operandos duma instrução são atualizados por qualquer uma das 3 instruções anteriores.
- ***Hazards*** podem ocorrer quando uma instrução altera o estado do programa, o qual será lido por uma instrução posterior.
- O **estado dum programa** inclui: os registos genéricos, o PC, a memória, o registo de *flags* (S|O|Z) e o registo de estado.
- Analisando a possibilidade de ocorrência de *hazards* associados a cada tipo de componente do estado dum programa, podemos concluir que só é preciso gerir:
  - os *hazards* de dados nos registos genéricos.
  - os *hazards* de controlo no PC.
  - as exceções assinaladas no registo de estado.

Resolver anomalias no encadeamento usando *stalling*

- ***Stalling*** ⇔ empatar o encadeamento das instruções.
- O **Y86 PIPE-** evita um *hazard* de dados mantendo a instrução que gera esse *hazard* no estágio de **D**escodificação até que as instruções, que produzem os operandos que ela precisa, tenham passado pelo estágio ***W**rite-back*.
- Ao executar um programa, se após descodificar uma instrução I_{dh} se deteta que tem uma dependência de dados de outra(s) instrução(ões) também em execução, a lógica de controlo do *stall* injeta uma **bolha** no estágio de **E**xecução e repete a **D**escodificação da instrução I_{dh} no(s) ciclo(s) seguinte(s). Na realidade, o **Y86 PIPE-** insere instrução(ões) ***nop*** dinamicamente no estágio de **E**xecução.
- Ao manter a instrução I_{dh} no estágio de **D**escodificação, devemos manter também a instrução seguinte no seu atual estágio de **E**xtração. Para conseguir isso basta manter o **PC** fixo.

Exercício 1

I1: irmovl $10, **%eax**
I2: rrmovl **%ecx**, %edx
I3: rmmovl **%eax**, $0(**%ecx**)

	1	2	3	4	5	6	7	8	9	10	11	12	13
I1	F	D	E	M	**W**								
I2		F	**D**	E	M	W							
Bolha					**E**	**M**	**W**						
Bolha						**E**	**M**	**W**					
I3			F	**D**	**D**	**D**	E	M	W				

Entre I1 e I3 → **existe uma dependência de dados** no **%eax** (RAW - leitura após escrita)
Entre I2 e i3 → **não** existe dependência de dados no %**ecx** (RAR - leitura após leitura)

Exercício 2

I1: rmmovl %eax, **$0(%ebx)**
I2: rrmovl %ecx, **%edx**
I3: mrmovl **$0(%ebx)**, **%edx**

	1	2	3	4	5	6	7	8	9	10	11	12	13
I1	F	**D**	E	**M**	W								
I2		F	D	E	M	**W**							
I3			F	**D**	E	**M**	**W**						

Entre I1 e I3 → **não** existe dependência no %**ebx** (RAR - leitura após leitura)
Entre I1 e I3 → existe uma dependência do tipo RAW no acesso à **memória** mas não causa *stall*
Entre I2 e I3 → **não** existe dependência no %**edx** (WAW - escrita após escrita não causa problemas)

Exercício 3

```
I1:   xorl %eax, %eax     # resultado=0 ➔ flag Z=1
I2:   jne I4              # salta se flag Z=0, logo não salta
I3:   subl %esi, %edx
I4:   rrmovl %edx, %esi
I5:   halt
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	F	D	**E (sF)**	M	W										
I2		F	D	**E (sB)**	M	W									
I4→bolha			~~F~~	~~D~~	~~E (aB)~~	M	W								
I5→bolha				~~F~~	~~D (aB)~~	E	M	W							
I3					F	D	E	M	W						
I4→bolha								E	M	W					
I4→bolha									E	M	W				
I4→bolha										E	M	W			
I4						F	D	D	D	D	E	M	W		
I5							F	F	F	F	D	E	M	W	

- **(sF)** - **I1** altera as *flags* na fase **E (Z=1)**.
- **(sB)** - **I2** gera o sinal ***Bch*** na fase **E (não salta)**.
- No ciclo **3** arranca **I4** pela regra de previsão de salto do Y86 PIPE-: saltar sempre.
- No ciclo **4** o resultado da execução de **I2** indica uma correção da previsão de salto: a próxima instrução a ser extraída é a instrução **I3**.
- **(aB)** - No ciclo **5** (quando a condição ***ne*** já é conhecida) a previsão é corrigida, iniciando a extração de **I3**.
- **Entre I3 e I4** → **existe uma dependência de dados** no **%edx** (RAW - leitura após escrita)
- **I5** é mantida no estágio de **extração** enquanto **I4** se mantém em descodificação (*stalled*)

Exercício 4

Este programa inclui o retorno de uma função. Tenha em atenção que, ao executar um *ret*, o endereço de retorno é lido da pilha e só fica disponível após o *ret* terminar o estágio de acesso à memória (**M**).

```
I1:   call I3
I2:   halt
I3:   ret
I4:   addl %eax, %eax   # I4 nunca é executada
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
I1 *(call)*	F	D rd **ESP**	**E** valE = ESP-4	M M[valE] = e.ret	**W** **ESP=valE** wr PC								
I3→**Bolha**		~~F~~	~~D~~ **rd ESP**	**E**	M	W							
I3→**Bolha**		~~F~~	~~D~~	~~D~~	**E**	M	W						
I3→**Bolha**		~~F~~	~~D~~	~~D~~	~~D~~	**E**	M	W					
I3 *(ret)*		F	**D** rd ESP	**D** rd ESP	**D** rd ESP	**D** rd ESP	**E** valE = ESP+4	**M** valM= M[]	W ESP= valE wr PC				
Bolha								**E**	M	W			
Bolha									**E**	M	W		
Bolha										**E**	M	W	
I2			stall	stall	stall	stall	stall	stall	**F**	D	**E**	M	W

- Este programa apresenta dependências de controlo.
- **Entre I1 e I3** existe uma dependência do tipo RAW no **ESP**.
- **I2** só pode prosseguir após **I3** (***ret***) terminar a fase **M** (ou seja, estiver na fase **W**).

- **Nota**: o livro do Bryant apresenta outra solução para este exercício (na página 432). Qual está correta?