Encadeamento de Instruções

Arquitetura de Computadores

Licenciatura em Engenharia Informática

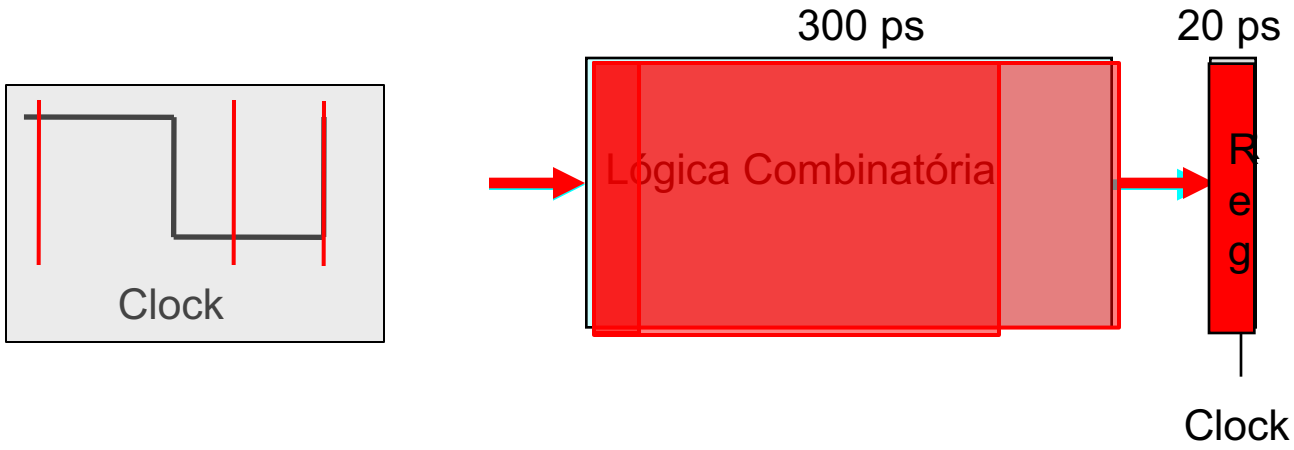
Luís Paulo Santos

Material de Apoio

- "*Computer Organization and Design: The Hardware / Software Interface*" David A. Patterson, John L. Hennessy; 5th Edition, 2013
  - Secção 4.5 (pags. 272 .. 286) – An overview of pipeline
  - Secção 4.6 (pags. 286 .. 300) – Pipeline datapath (~~and control~~)
  - Secção 4.7 (pags. 303 .. 316) – Data hazards
  - Secção 4.8 (pags. 316 .. 325) – Control hazards

- "*Computer Systems: a Programmer's Perspective*"; Randal E. Bryant, David R. O'Hallaron--Pearson (2nd ed., 2011)
  - Secção 4.4 (pags. 391 .. 398) – General principles of pipelining
  - Secção 4.5 (pags. 400 .. 446) – Pipelined Y86 implementations

Exemplo Sequencial



- Toda a computação feita num único ciclo:
  300 ps para gerar os resultados + 20 ps para os armazenar
- Ciclo do relógio >= 320 ps

Tempo de execução de uma instrução = 320 ps

Frequência relógio = ciclo^{-1} <= 1 / 320E-12 = 3.12 GHz

Execução de Instruções: Fases

- Execução de uma instrução
  (exemplo de decomposição em diferentes estágios)
  1. Leitura (*Fetch)*
  2. Descodificação / Leitura de Operandos
  3. Execução (ALU)
  4. Escrita de Resultados

- Estas fases podem ser agrupadas ou reordenadas para permitir a execução das instruções em vários estágios encadeados -> PIPELINE

Encadeamento na Vida Real

Sequencial

Paralelo

Encadeado (Pipeline)

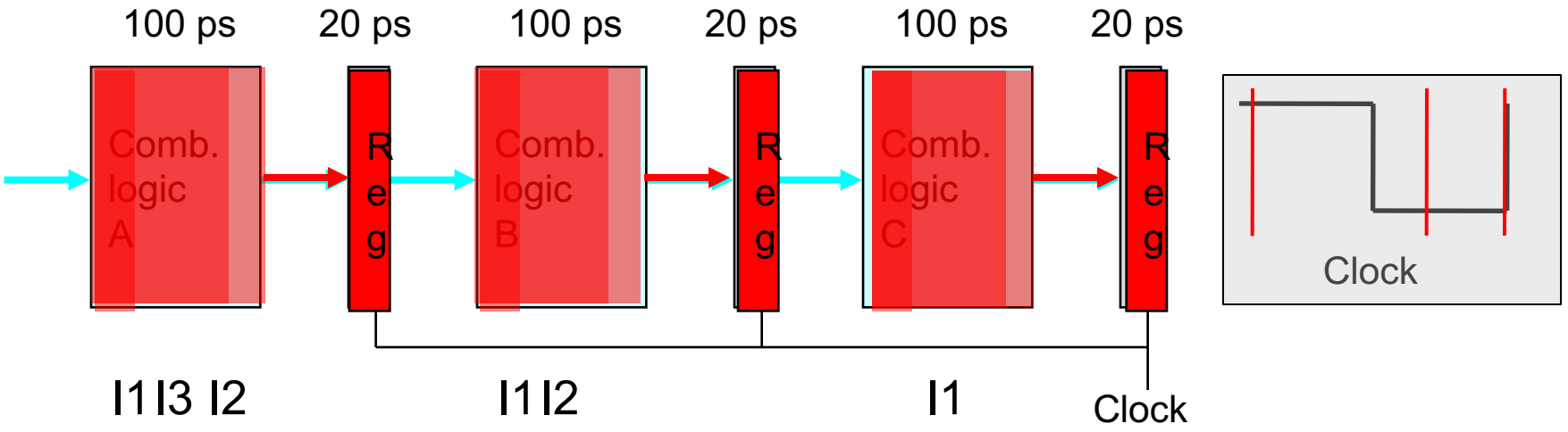
- Ideia
  - Dividir processo em estágios independentes
  - Objectos movem-se através dos estágios em sequência
  - Em cada instante, múltiplos objectos são processados simultaneamente

Ford’s assembly line: pipeline

The flywheel magneto, the first manufactured part to be made on a moving assembly line, passing by workers at Ford's Highland Park, Michigan, U.S., plant.

(https://www.britannica.com/technology/Model-T)

Encadeamento: Exemplo



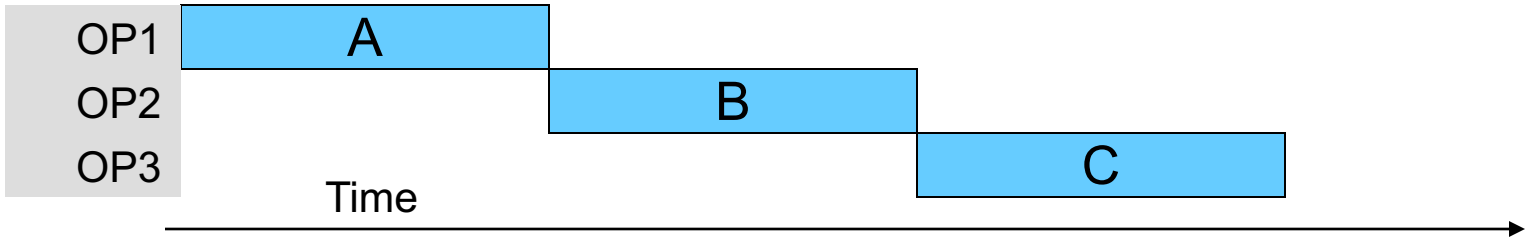
- Dividir lógica combinatória em 3 estágios de 100 ps cada
- Nova instrução começa logo que a anterior termina o primeiro estágio: Ciclo >= 120 ps

T_{exec} uma instrução = nº estágios * ciclo = 3 * 120 = 360 ps

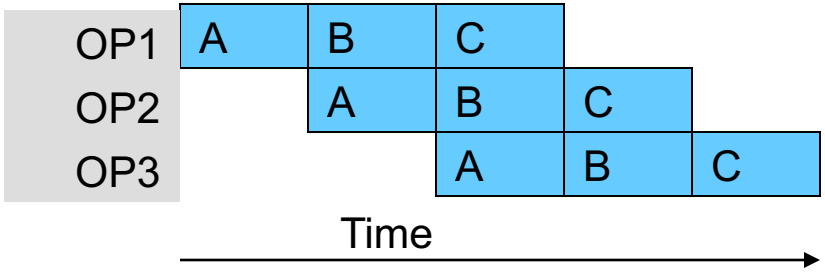
Frequência <= 1/120E-12 = 8.33 GHz

Encadeamento: Diagramas

- Sequencial



  - Só começa uma nova instrução quando a anterior termina
- Encadeada com três estágios



  - 3 instruções simultâneas
  - Tempo de execução: uma instrução necessita de 3 ciclos

Desempenho

- Arquitectura sequencial simples (e.g. RISC - 1 único estágio) -> CPI = 1
- Numa arquitectura com pipeline, assumindo:
  - #I >> #estágios, todas independentes umas das outras;
  - que não há penalizações no acesso à memória:

CPI = 1 (em cada ciclo termina 1 instrução)

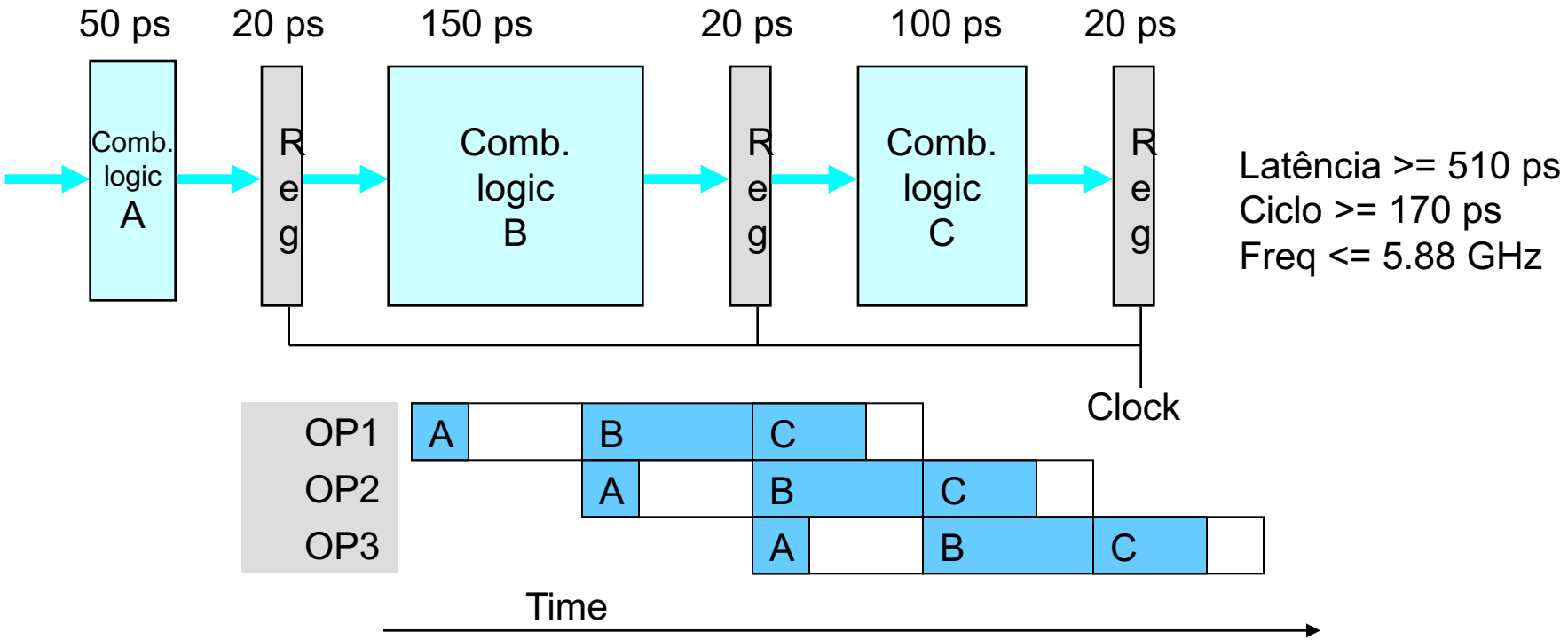
$$T_{exec} = CPI * \#I * T_{cc}$$

- Onde se ganha com o pipeline?

➢ O período do relógio (Tcc) pode ser menor do que na arquitectura sequencial (ciclo único), logo a frequência do relógio aumenta

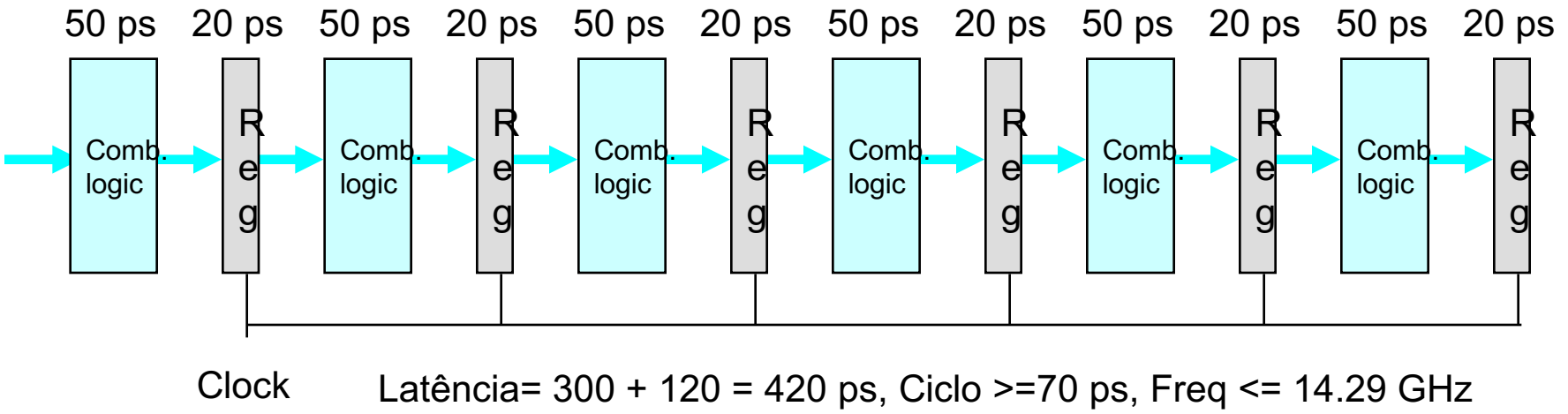
- Veremos posteriormente que as arquitecturas encadeadas implicam um aumento do CPI

Limitações: Latências não uniformes



- Período do relógio limitado pelo estágio mais lento
- Outros estágios ficam inactivos durante parte do tempo
- Desafio: decompor um sistema em estágios balanceados

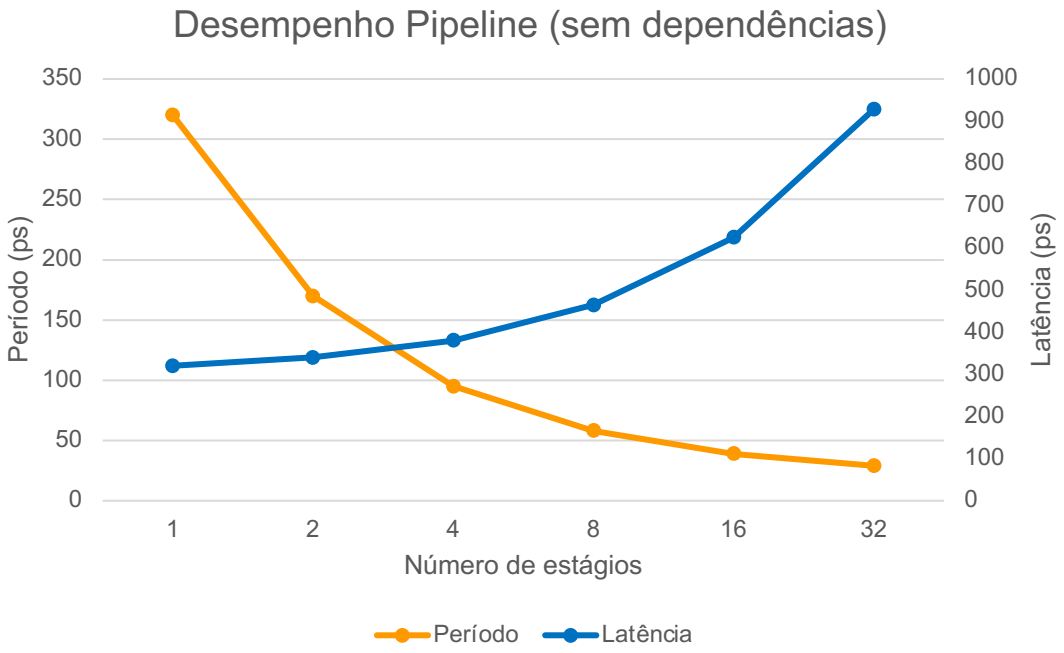
Limitações: custo do registo



- Pipelines mais profundos têm maiores custos associados aos registos
- Percentagem de tempo devido aos registos por instrução:
  - 1-stage pipeline: 6.25% (020 em 320 ps)
  - 3-stage pipeline: 16.67% (060 em 360 ps)
  - 6-stage pipeline: 28.57% (120 em 420 ps)

Limitações: custo do registo

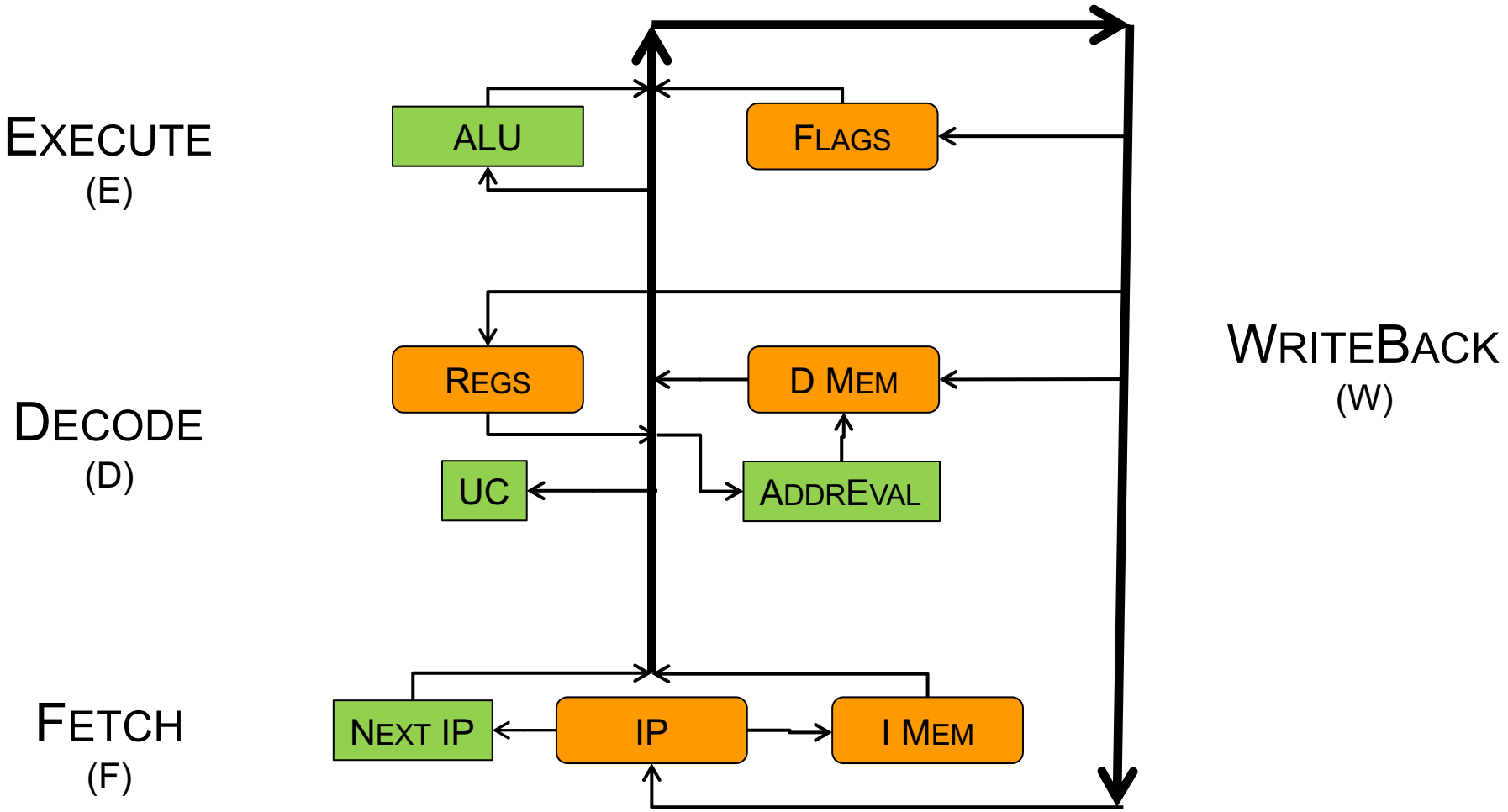
- Considerando a lógica combinatória com uma duração de **300 ps** e o registo com uma duração de **20 ps**:
  - Um número infinito de estágios implica ciclo do relógio = 20 os, mas latência = infinito



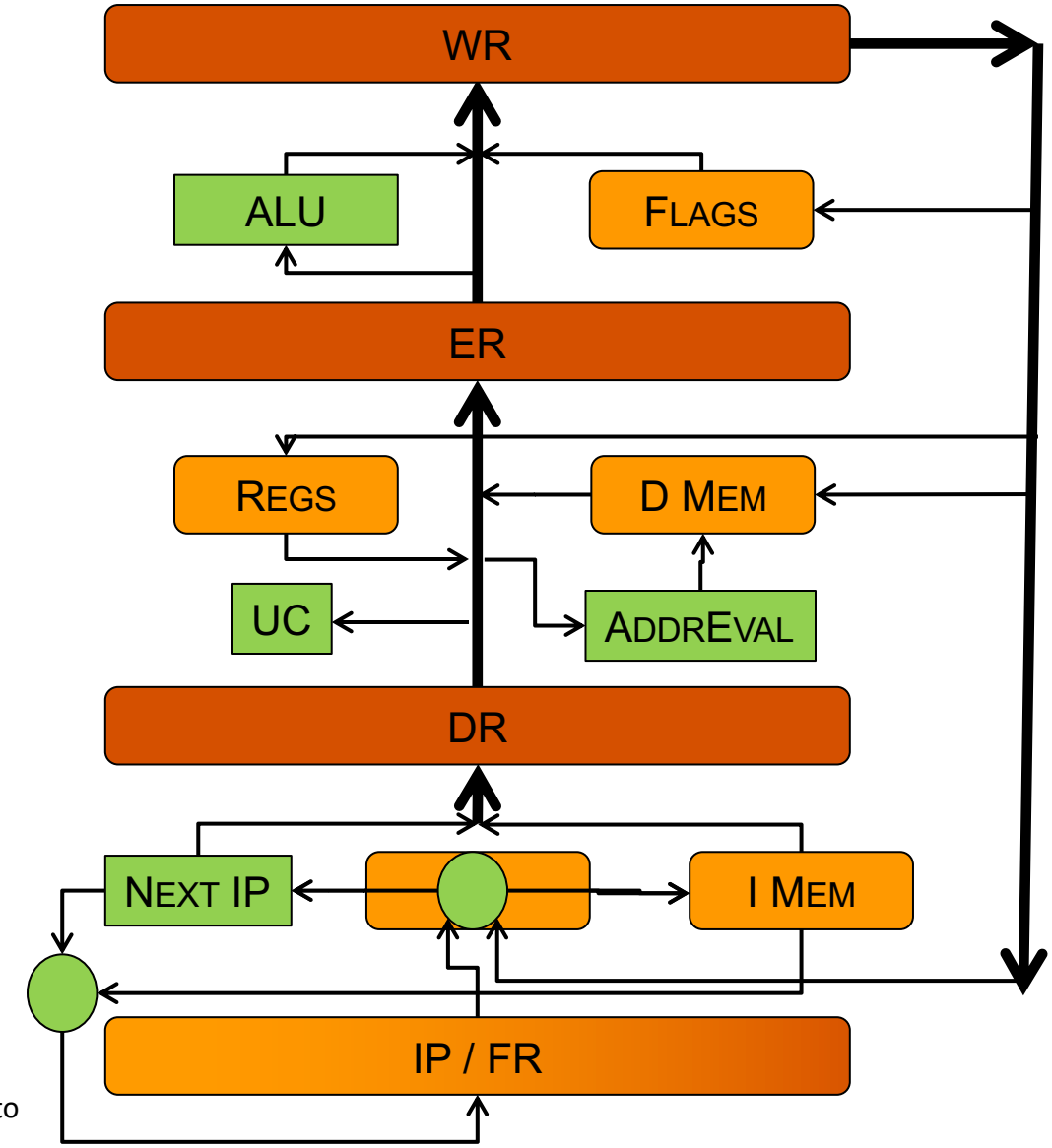
Instruction Level Parallelism

- As arquitecturas em *pipeline* exploram o paralelismo ao nível das instruções
- Uma vez que cada instrução se encontra num estágio diferente de execução, o *pipeline* permite reduzir o período do relógio
- O CPI com *pipeline* é normalmente superior a 1, devido a dependências entre instruções (a ver adiante)
- Outras formas de paralelismo, mesmo ao nível das instruções diminuem o CPI, como forma de aumentar o desempenho (tempo de resposta ou débito).

Arquitectura sequencial simples



Arquitectura encadeada simples



EXECUTE
(E)

DECODE
(D)

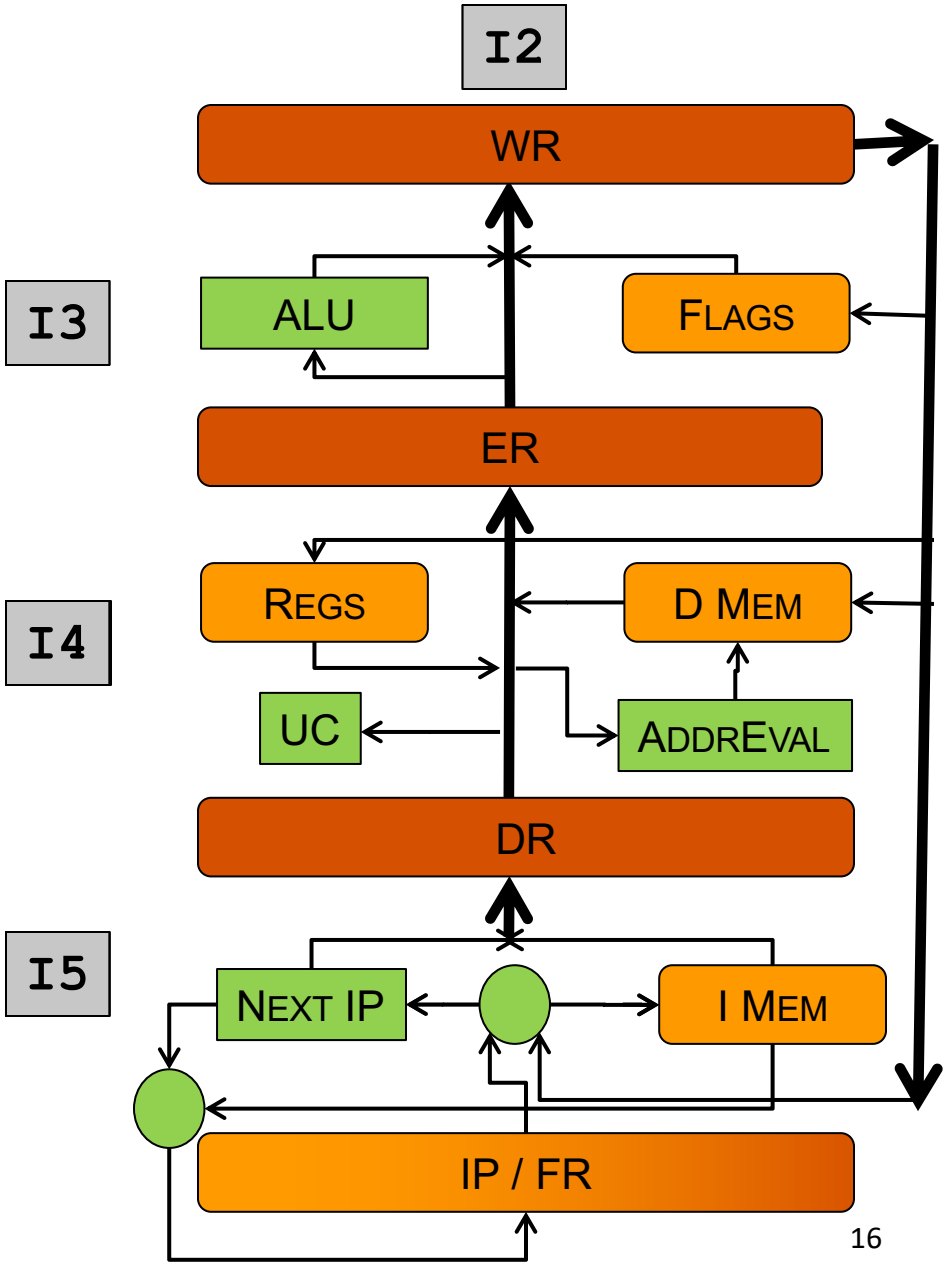
FETCH
(F)

WRITEBACK
(W)

Pipeline : Execução

```
I1: movl $10, %eax
I2: movl 30(%ebx), %ecx
I3: addl %esi, %edi
I4: subl %esi, %ebx
I5: jmp MAIN
I6: …
```

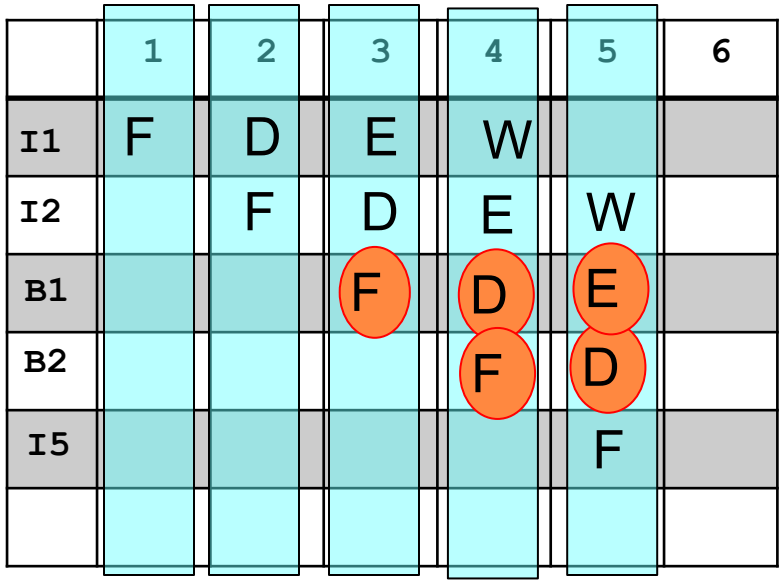
	1	2	3	4	5
I1	F	D	E	W	
I2		F	D	E	W
I3			F	D	E
I4				F	D
I5					F
I6					

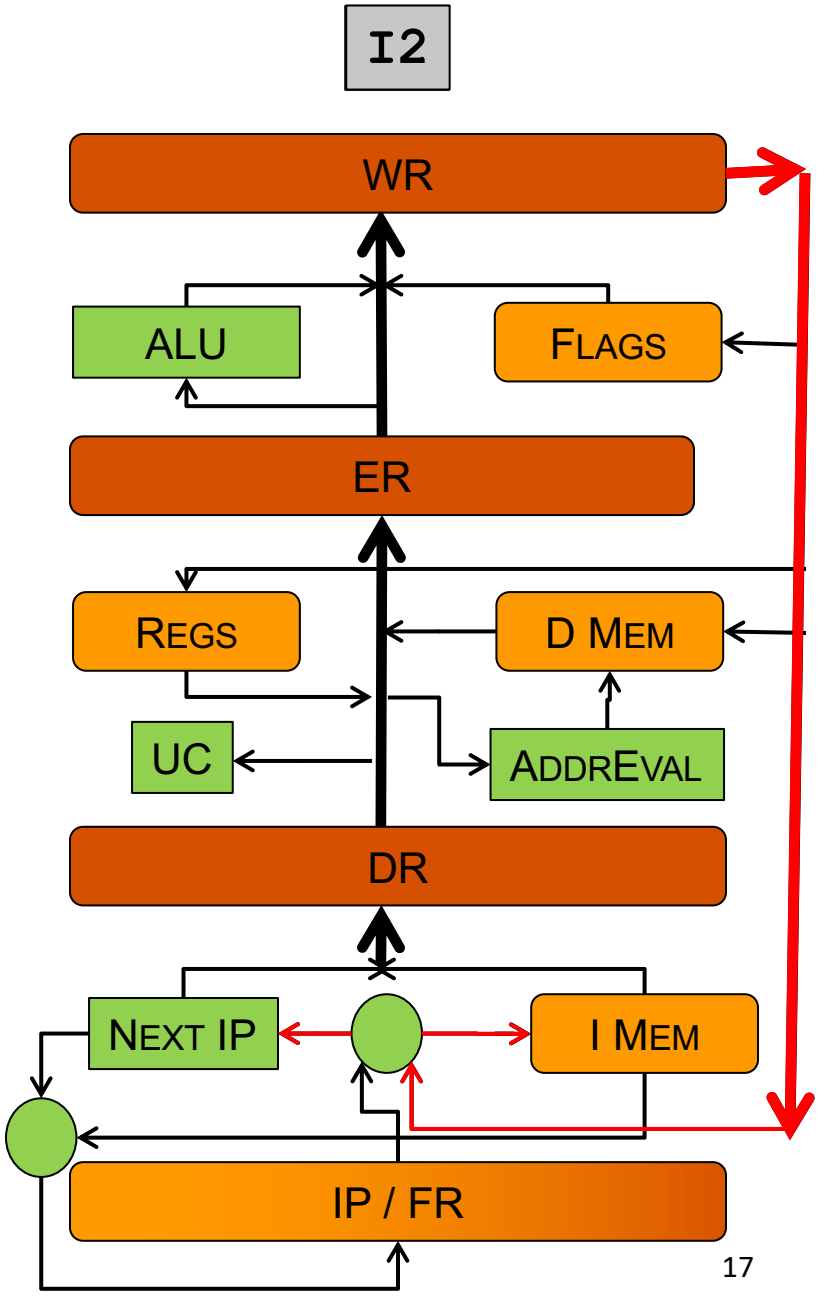


Dependências de Controlo - jXX

```
I1: subl %eax, %eax
I2: jz I5
I3: addl %esi, %edi
I4: subl %esi, %ebx
I5: addl %ecx, %edx
```

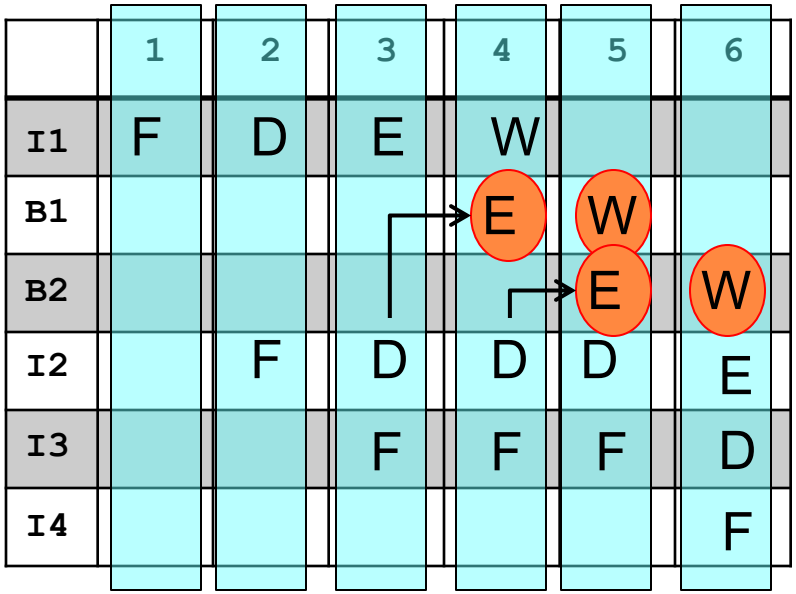
? *stalling*

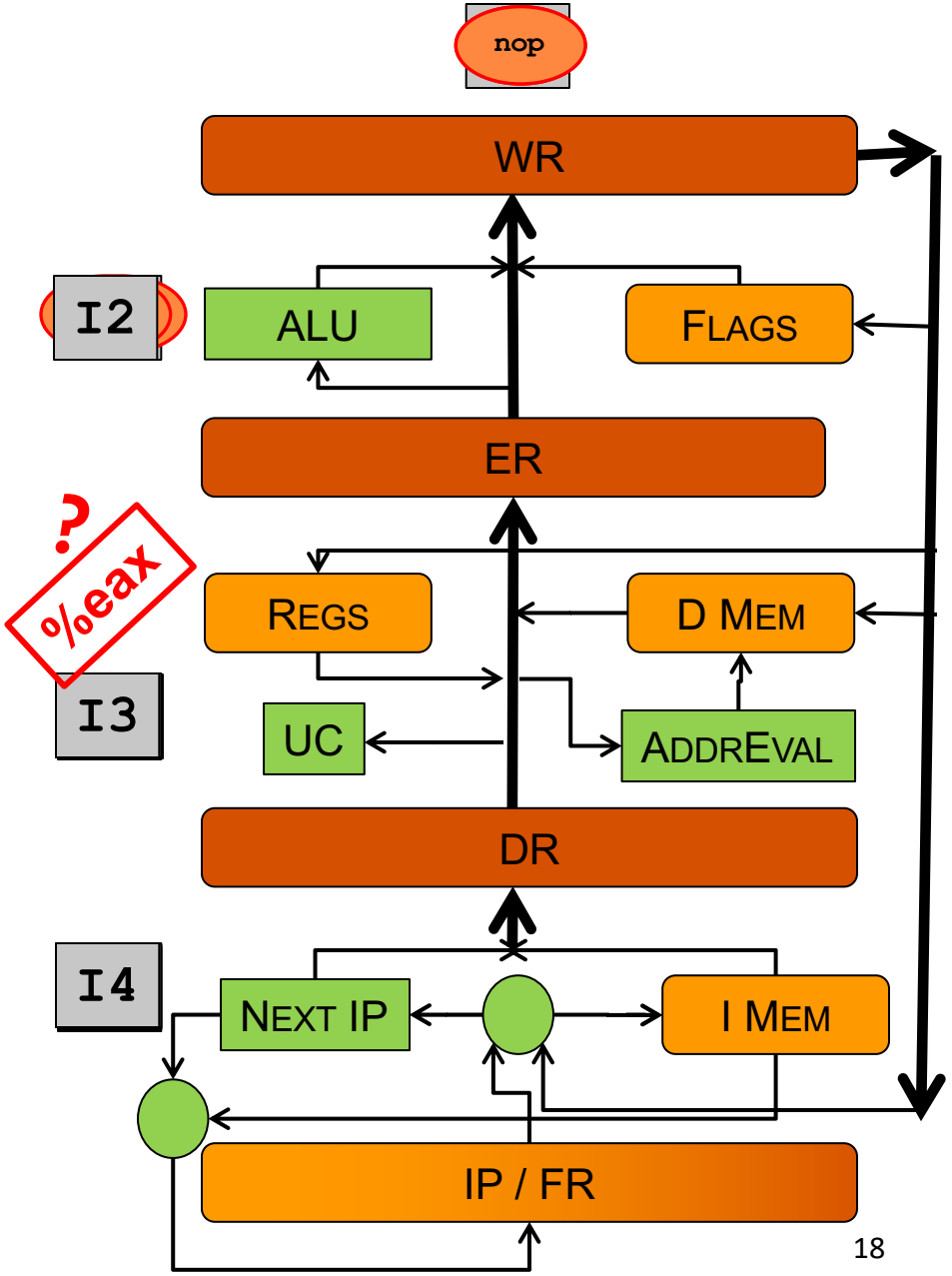




Dependências de dados

```
I1: movl $10, %eax
I2: addl %ebx, %eax
I3: movl $20, %ecx
I4: movl $20, %edx
```

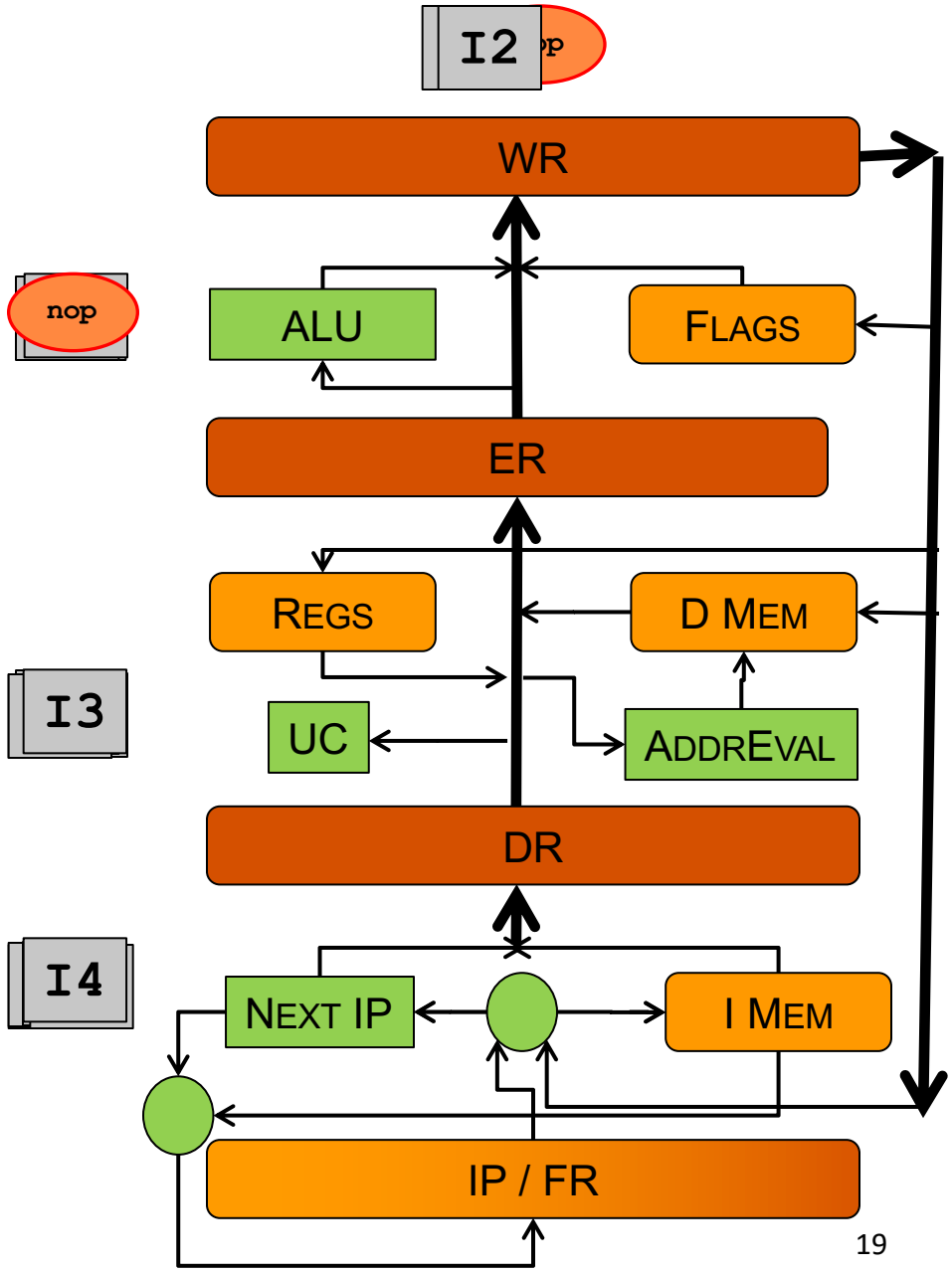




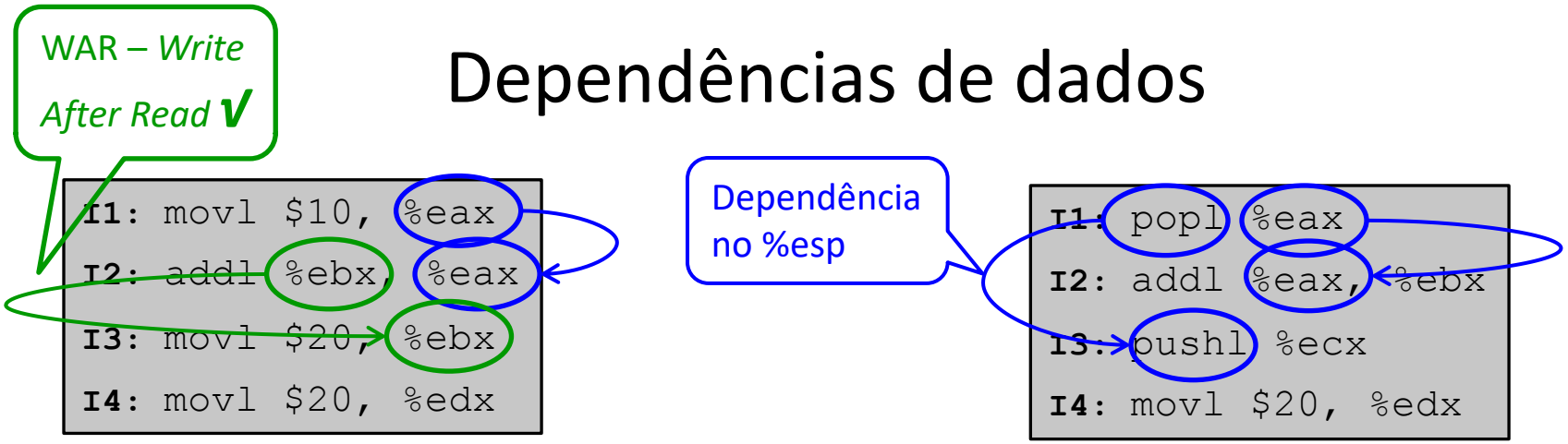
Dependências de dados

```
I1: movl $10, %eax
I2: movl $20, %ecx
I3: addl %ebx, %eax
I4: movl $20, %edx
```

	1	2	3	4	5	6
I1	F	D	E	W		
I2		F	D	E	W	
B1					E	W
I3			F	D	D	E
I4				F	F	D
…						



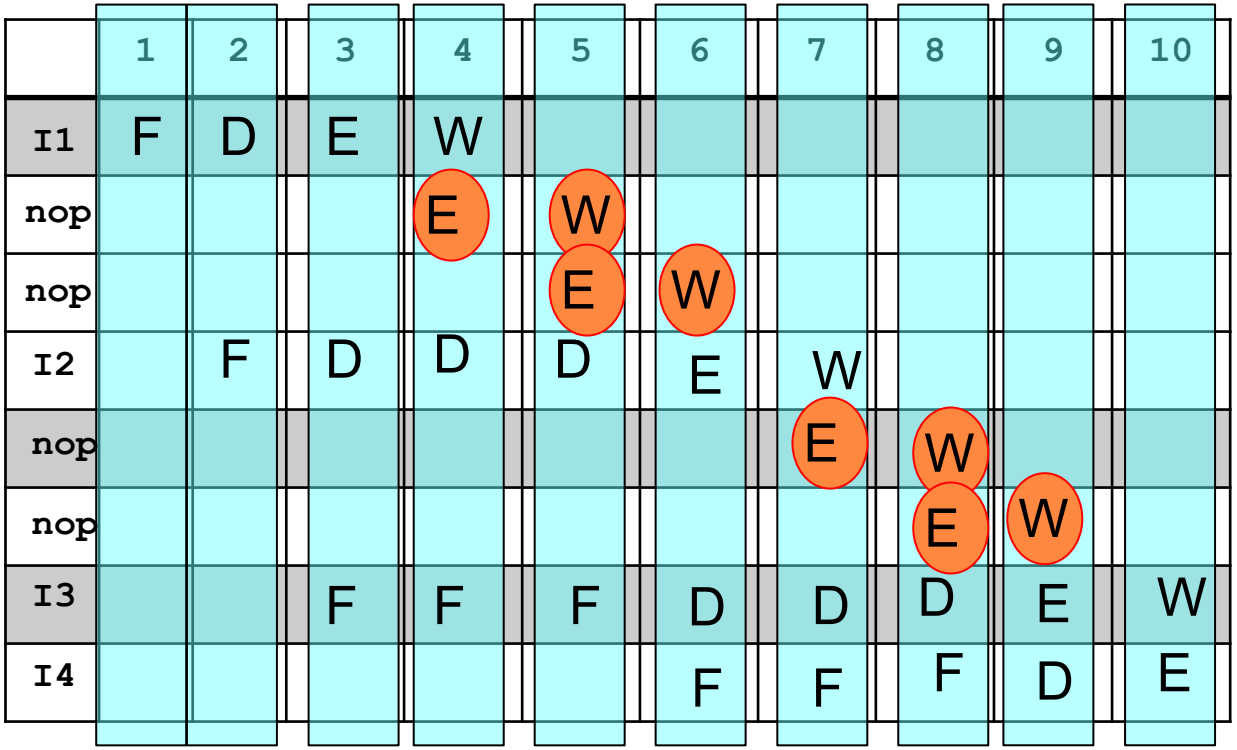
Dependências de dados



- Os registos são escritos apenas no estágio de WRITEBACK
- Se uma instrução tenta **ler** um registo **antes da escrita** estar terminada é necessário **resolver a dependência RAW** (Read After Write)
- Injectando "bolhas" (NOPs) no estágio de execução a leitura é adiada até ao ciclo imediatamente a seguir à escrita

Exercício: Dependências de dados

```
I1: movl $10, %eax
I2: pushl %eax
I3: addl %eax, %esp
I4: movl $20, %edx
```



Dependências e *stalling*

- Se uma instrução depende da execução de uma instrução anterior
- Se essa instrução anterior não terminou ainda
- Então o processador aguarda (*stalls)* os ciclos necessários, injectando NOPs no *pipeline*

- O CPI aumenta e esse aumento é proporcional ao número de estágios

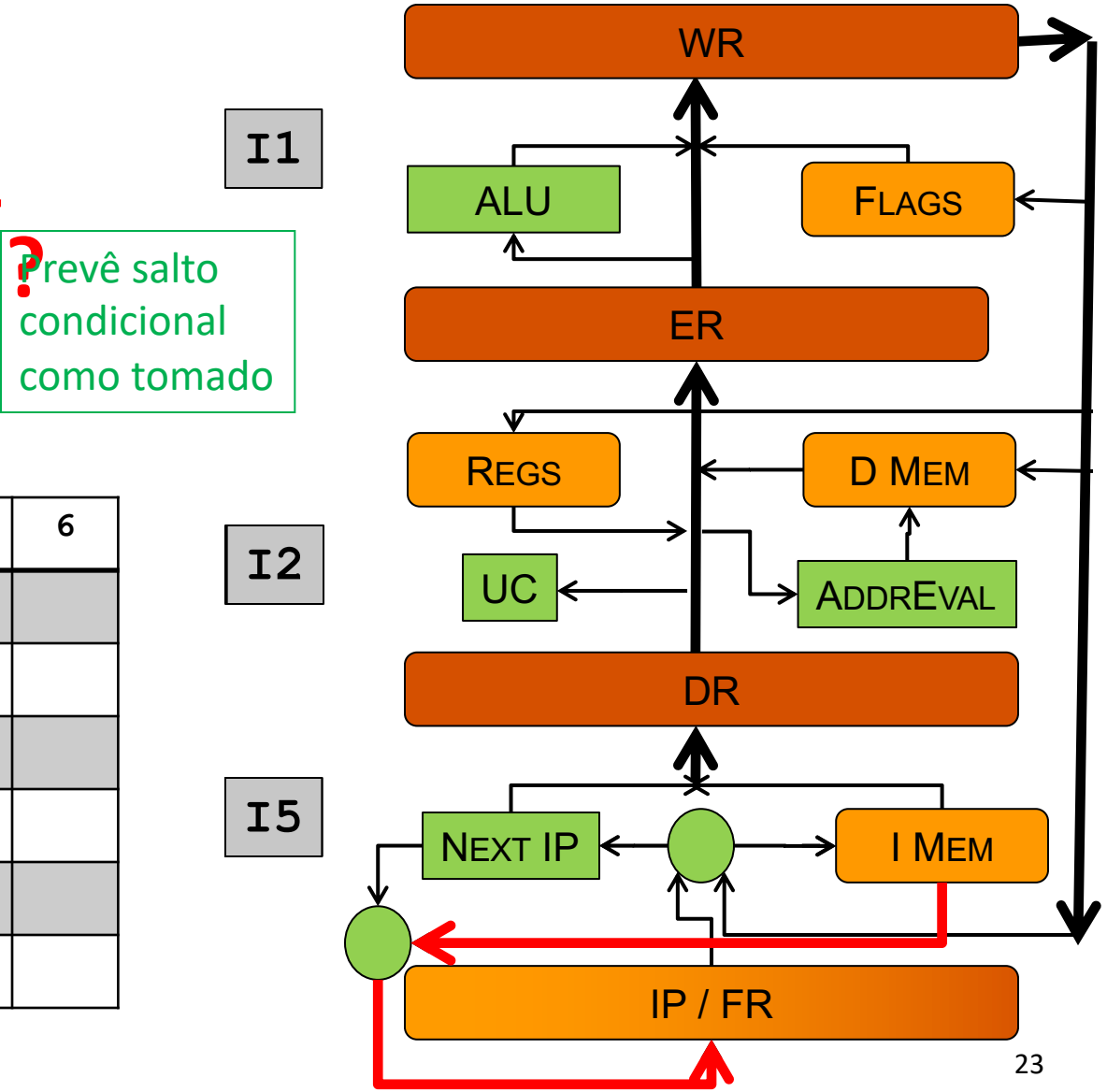
- Os processadores modernos só recorrem ao *stalling* quando não existe alternativa possível.

Dependências de Controlo - jXX

```
I1: subl %eax, %eax
I2: jz I5
I3: addl %esi, %edi
I4: subl %esi, %ebx
I5: addl %ecx, %edx
```

? Prevê salto condicional como tomado

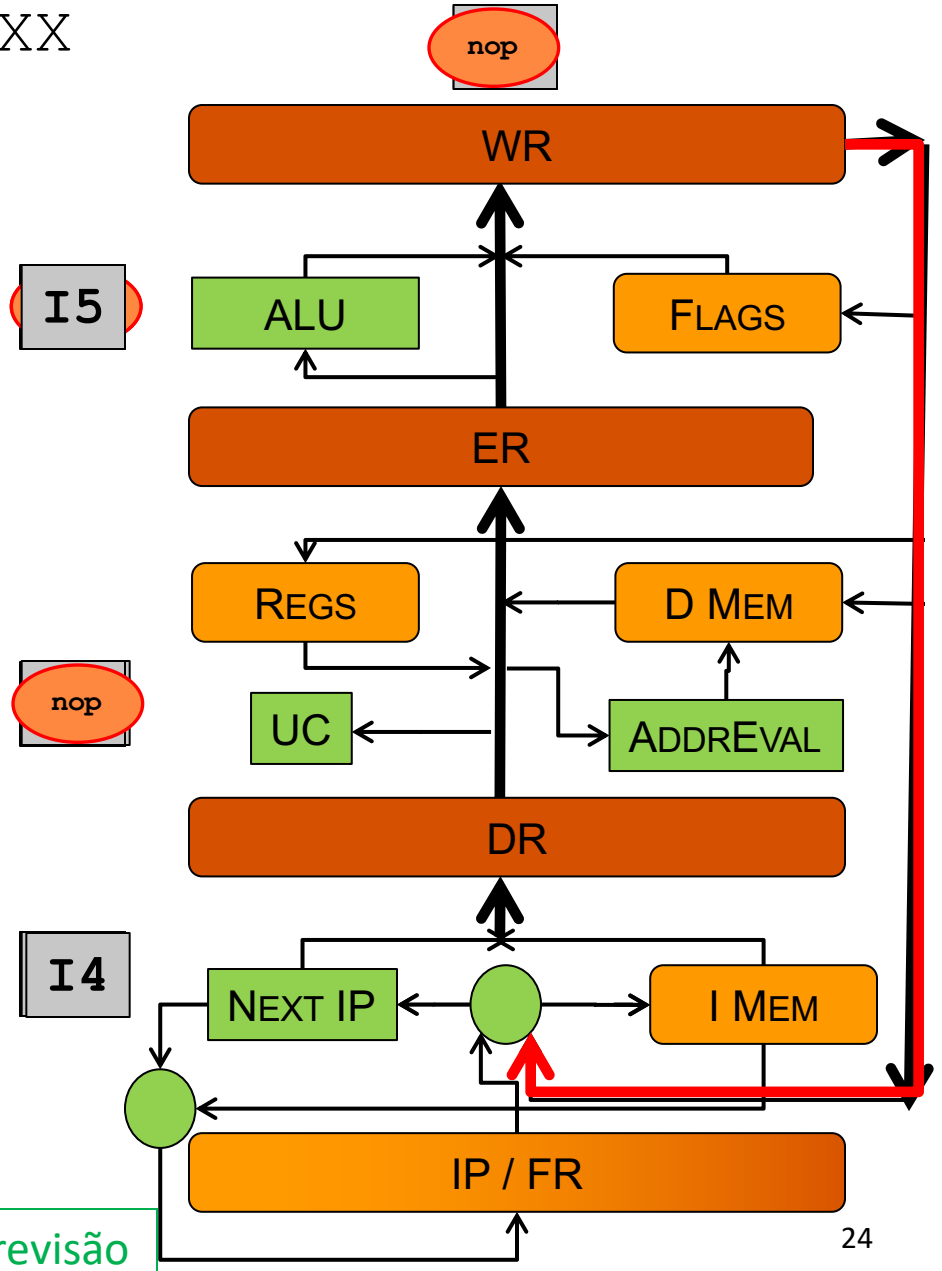
	1	2	3	4	5	6
I1	F	D	E			
I2		F	D			
I5			F			



Dependências de Controlo - jXX

```
I1: subl %eax, %eax
I2: jnz I5
I3: addl %esi, %edi
I4: subl %esi, %ebx
I5: addl %ecx, %edx
I6: movl $10, %eax
I7: movl $20, %esi
```

	1	2	3	4	5	6
I1	F	D	E	W		
I2		F	D	E	W	
I5			F	D	E	W
I6				F	D	E
I3					F	D
I4						F



Branch Ta
Corrige previsão

Dependências de Controlo - `jXX`

- Prevê-se que o salto é sempre tomado
- A correcção da previsão é determinada posteriormente, quando a instrução de salto termina o estágio de execução
- Se a previsão estiver errada as instruções que entretanto foram lidas para o *pipeline* são convertidas em `nops`:
  - Injecção de "bolhas"
- Isto é possível porque estas instruções ainda não tiveram hipótese de alterar o estado da máquina
  - Escritas que alteram o estado acontecem apenas no final do estágio de "WRITEBACK" (Registos)
- *stall* do pipeline (injecção de "bolhas"): resulta num desperdício de um número de ciclos igual ao número de bolhas injectadas

Dependências de Controlo - `jXX`

Previsão estática dos saltos

- Análises estatísticas: saltos condicionais são tomados em 60% dos casos. Prever que o salto é tomado (*Taken*) acerta mais do que metade das vezes
- Alternativas: NT – *Not Taken*
  BTFNT – *Backward Taken, Forward Not Taken*

Previsão dinâmica dos saltos

- A previsão é feita em tempo de execução baseada no historial recente
- *Branch prediction buffer* – tabela que guarda para cada instrução de salto do programa 1 *bit* indicando se o salto foi ou não tomado na última execução

  De facto este *buffer* tem um número limitado de entradas e guarda informação apenas sobre as últimas instruções de salto

Intel Core i7 920 - Desempenho

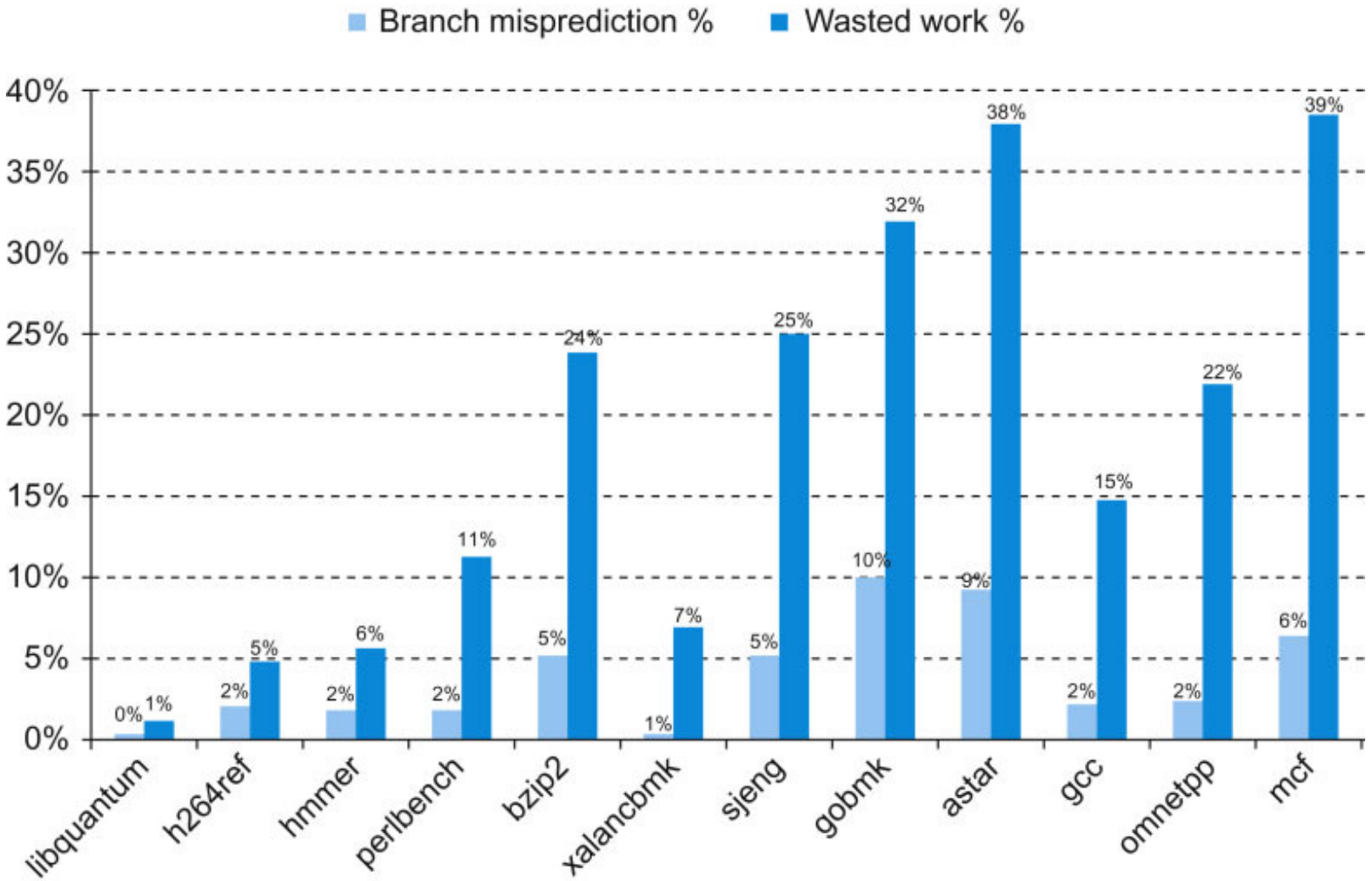


FIGURE 4.79 Percentage of branch mispredictions and wasted work due to unfruitful speculation of Intel Core i7 920 running SPEC2006 integer benchmarks.

Data forwarding: Motivação

- As dependências de dados são demasiado comuns
- Resolvê-las recorrendo à injecção de "bolhas" resulta no desperdício de um elevado número de ciclos, comprometendo o desempenho do *pipeline*
- A realimentação de dados (*data forwarding*) propõe-se resolver estas dependências de dados, diminuindo o número de bolhas injectadas (logo o número de ciclos desperdiçados)

- As dependências de controlo não sofrem qualquer alteração.

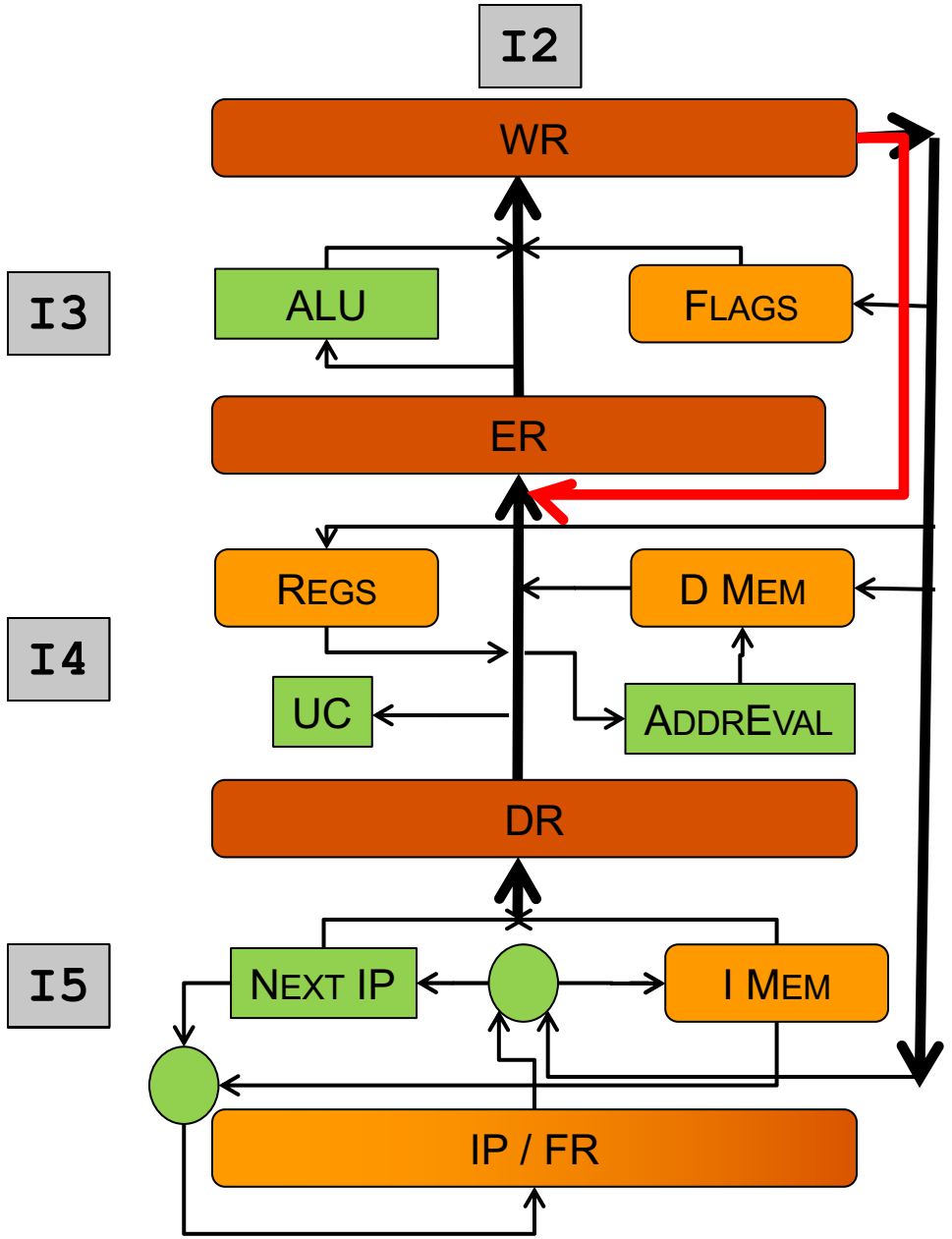
Data Forwarding

- Problema
  - Um registo é lido na fase de DECODE
  - A escrita só ocorre na fase de WRITEBACK
- Observação
  - O valor a escrever no registo existe dentro do *pipeline* desde a fase de execução
- Resolução do problema
  - Passar o valor necessário directamente do estágio onde está disponível (E ou W) para o estágio de DECODE

Exemplo de *Forwarding* (1)

```
I1: movl $10, %eax
I2: movl 30(%ebx), %ecx
I3: addl %esi, %eax
I4: …
```

	1	2	3	4	5
I1	F	D	E	W	
I2		F	D	E	W
I3			F	D	E
I4				F	D
I5					F
I6					

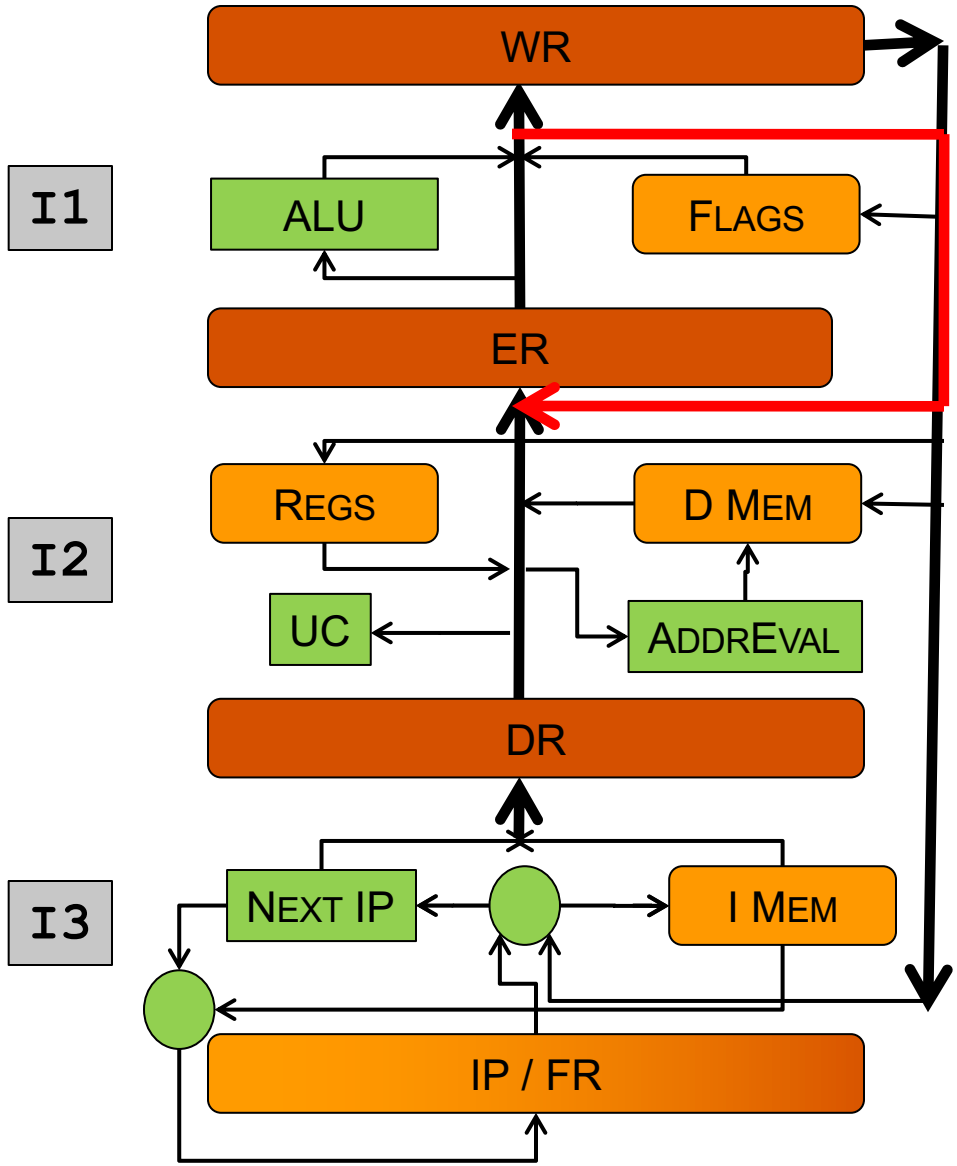


Realimenta de WR para ER

Exemplo de *Forwarding* (2)

```
I1: addl $10, %eax
I2: addl %esi, %eax
I3: …
```

	1	2	3	4	5
I1	F	D	E	W	
I2		F	D	E	W
I3			F	D	E
I4				F	D
I5					F
I6					



Realimenta de E para ER

Pipeline: Resumo

- Execução de n instruções simultaneamente em diferentes estágios
- Permite aumentar a frequência do relógio
- Dependências de Dados
  - *stalling :* injecção de bolhas (NOPs)
  - realimentação: elimina penalizações
- Dependências de Controlo
  - Saltos condicionais implicam execução especulativa (previsão do salto)
  - previsão errada implica *stalling* do *pipeline*