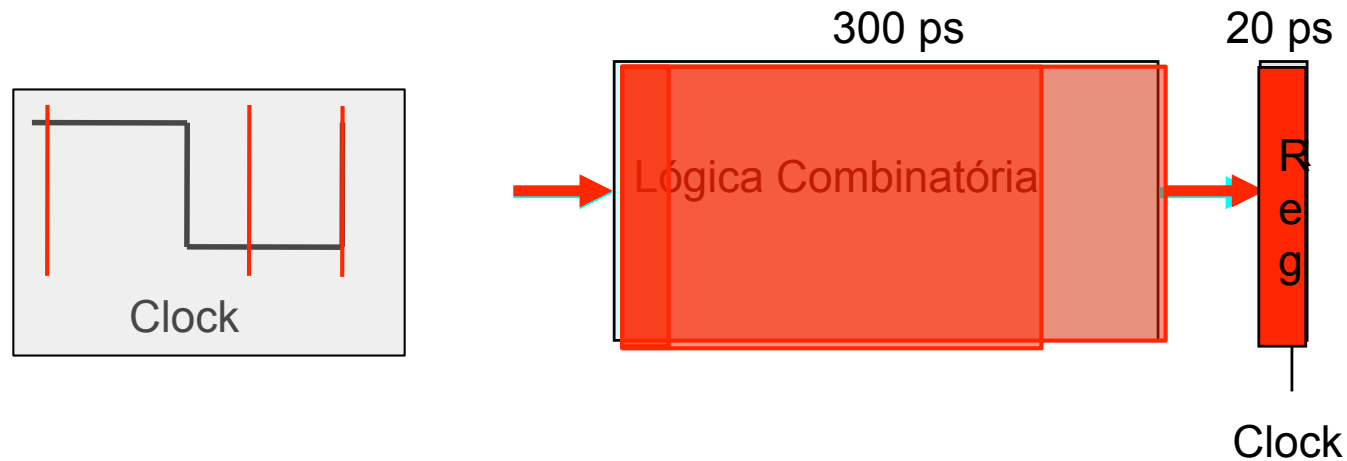


# *Encadeamento de Instruções*

Arquitetura de Computadores  
Mestrado Integrado em  
Engenharia Informática

# Exemplo Sequencial



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

Tempo de execução de uma instrução  $\rightarrow$  Latência = 320 ps

Taxa de execução de instruções  $\rightarrow$  Débito =  $\text{ciclo}^{-1} = 1 / 320\text{E-}12 = 3.12 \text{ GOPS}$

# Execução de Instruções: Fases

- Execução de uma instrução
  1. Leitura (*Fetch*)
  2. Descodificação
  3. Leitura de Operandos
  4. Execução (ALU)
  5. 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

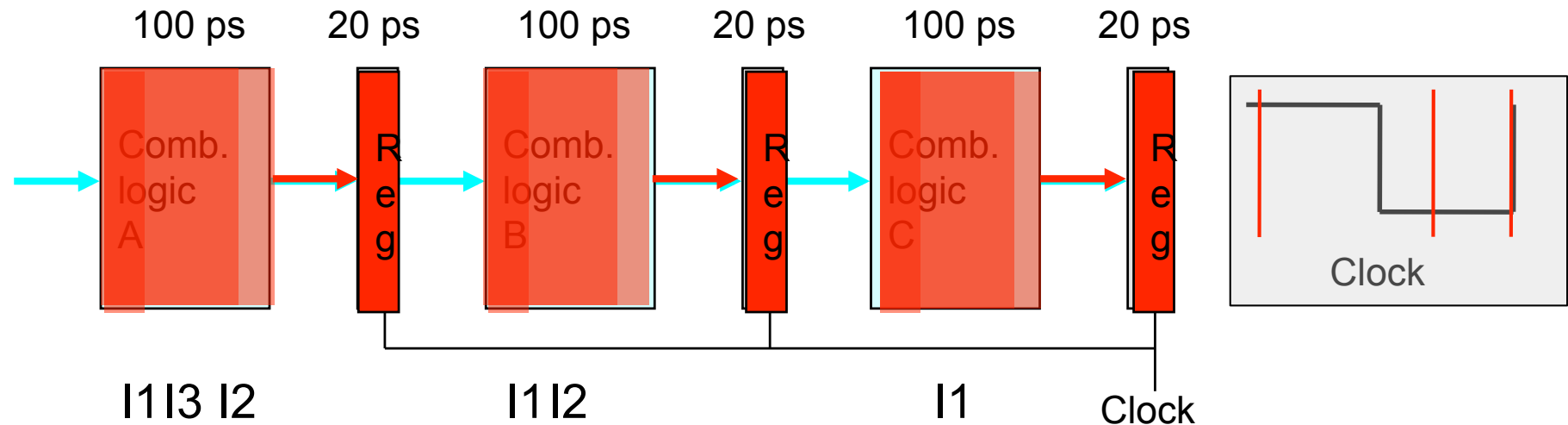


Encadeado (Pipeline)



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

# Encadeamento: Exemplo



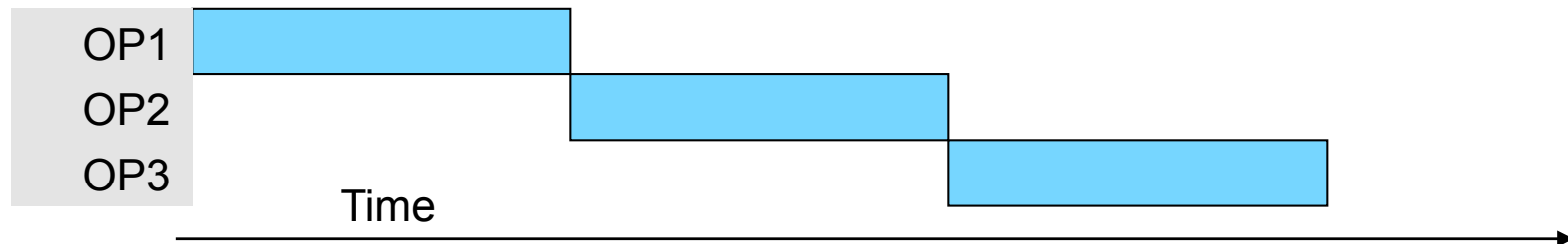
- Dividir lógica combinatória em 3 estágios de 100 ps cada
- Nova instrução começa logo que a anterior termina o primeiro estágio: Ciclo  $\geq 120$  ps

Latência = nº estágios \* ciclo = 3 \* 120 = 360 ps

Débito =  $1/120 \times 10^{-12} = 8.33$  GOPS

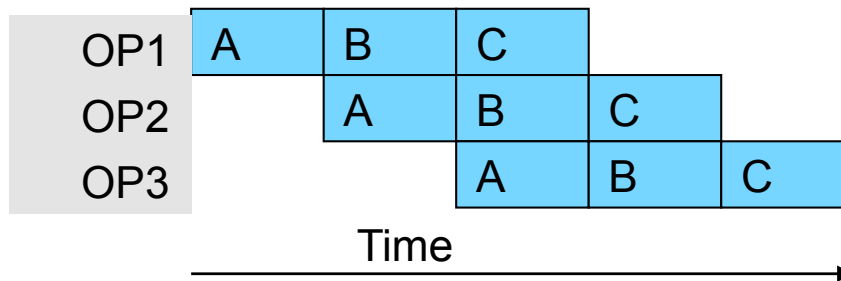
# Encadeamento: Diagramas

- Sequencial



- Só começa uma nova operação quando a anterior termina

- Encadeada com três estágios



- Débito: 3 operações simultâneas (ganho máximo de 3)
- Latência: cada instrução necessita sempre de 3 ciclos

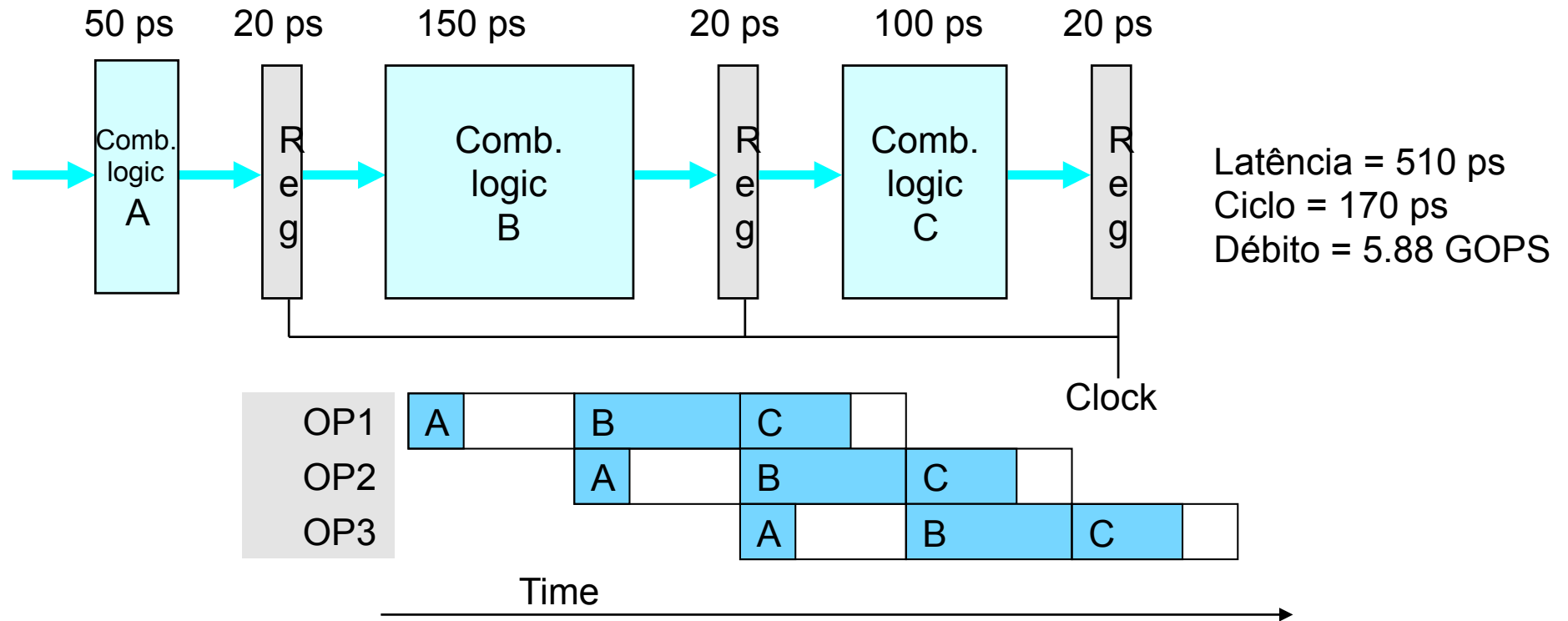
# Desempenho

- Arquitectura sequencial (1 único estágio) -> CPI = 1
- Numa arquitectura com pipeline (e assumindo por enquanto um programa com um elevado número de instruções ( $\#I > \#estágios$ ), todas independentes umas das outras), qual o CPI?
  - CPI = 1 (em cada ciclo termina 1 instrução)

$$T_{exec} = \frac{CPI * \#I}{f}$$

- Onde se ganha com o pipeline?
  - O período do relógio ( $T_{cc}$ ) 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 ( $>1$ , para máquinas com um único *pipeline*)

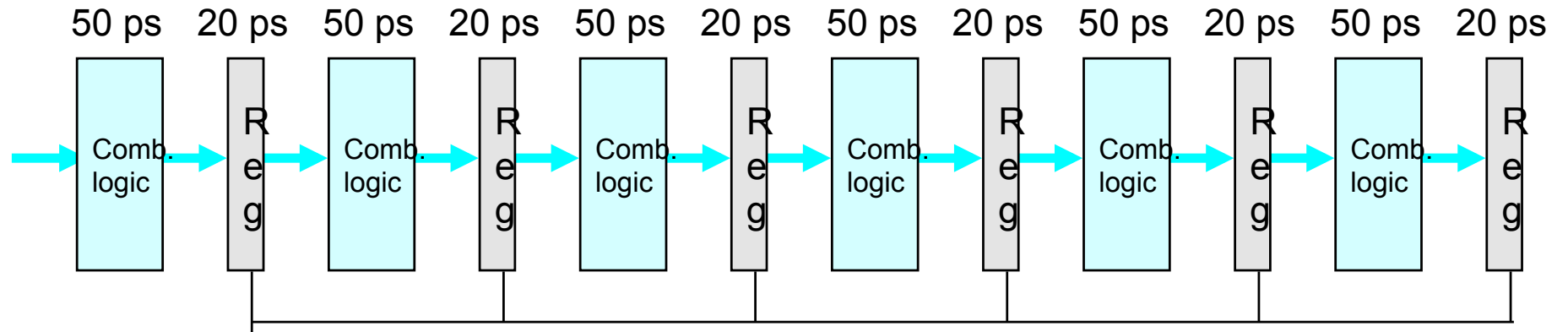
# Limitações: Latências não uniformes



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



# Limitações: custo do registo



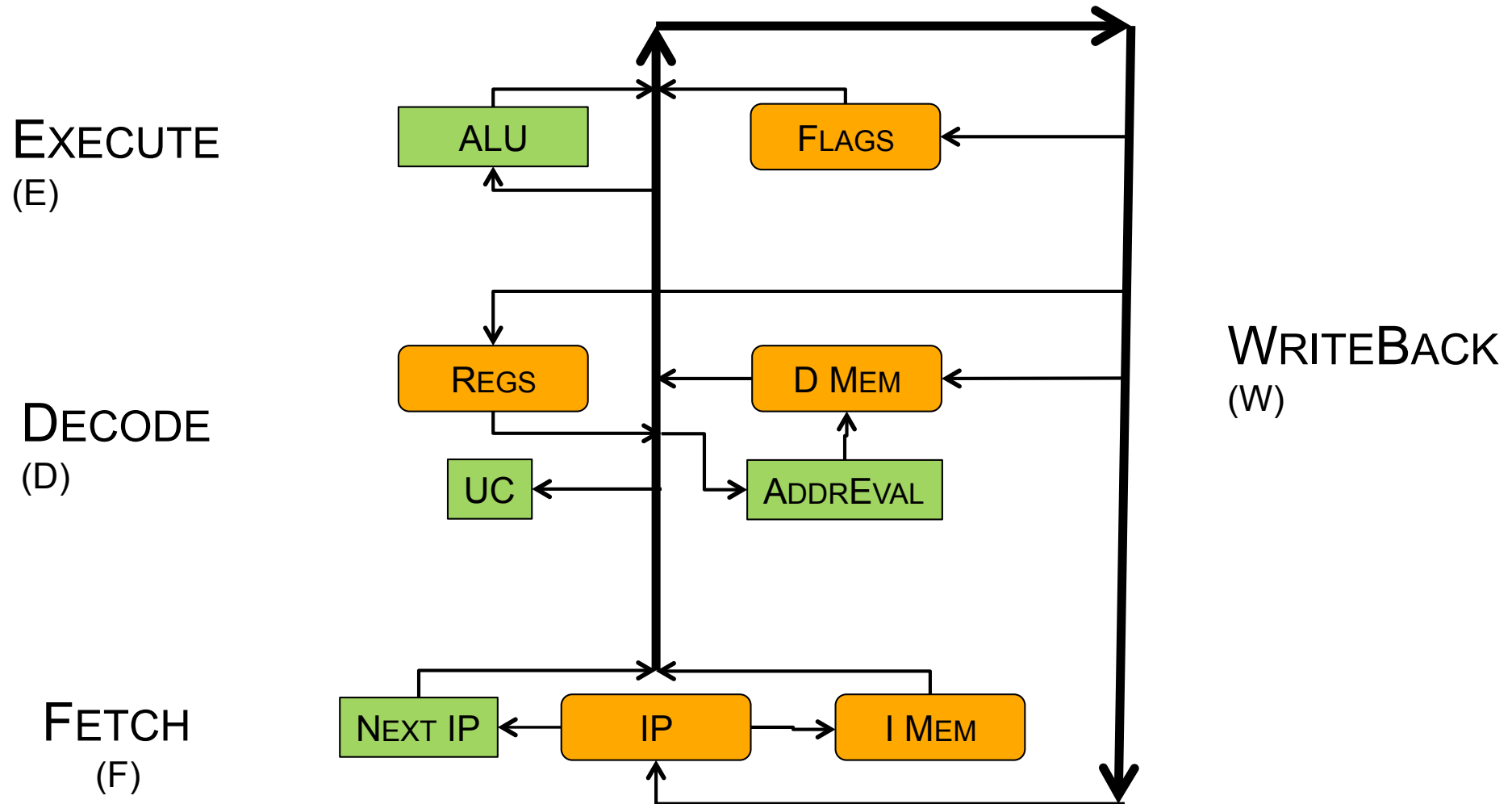
Clock      Latência= 300 + 120 = 420 ps, Ciclo=70 ps, Throughput = 14.29 GOPS

- 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)

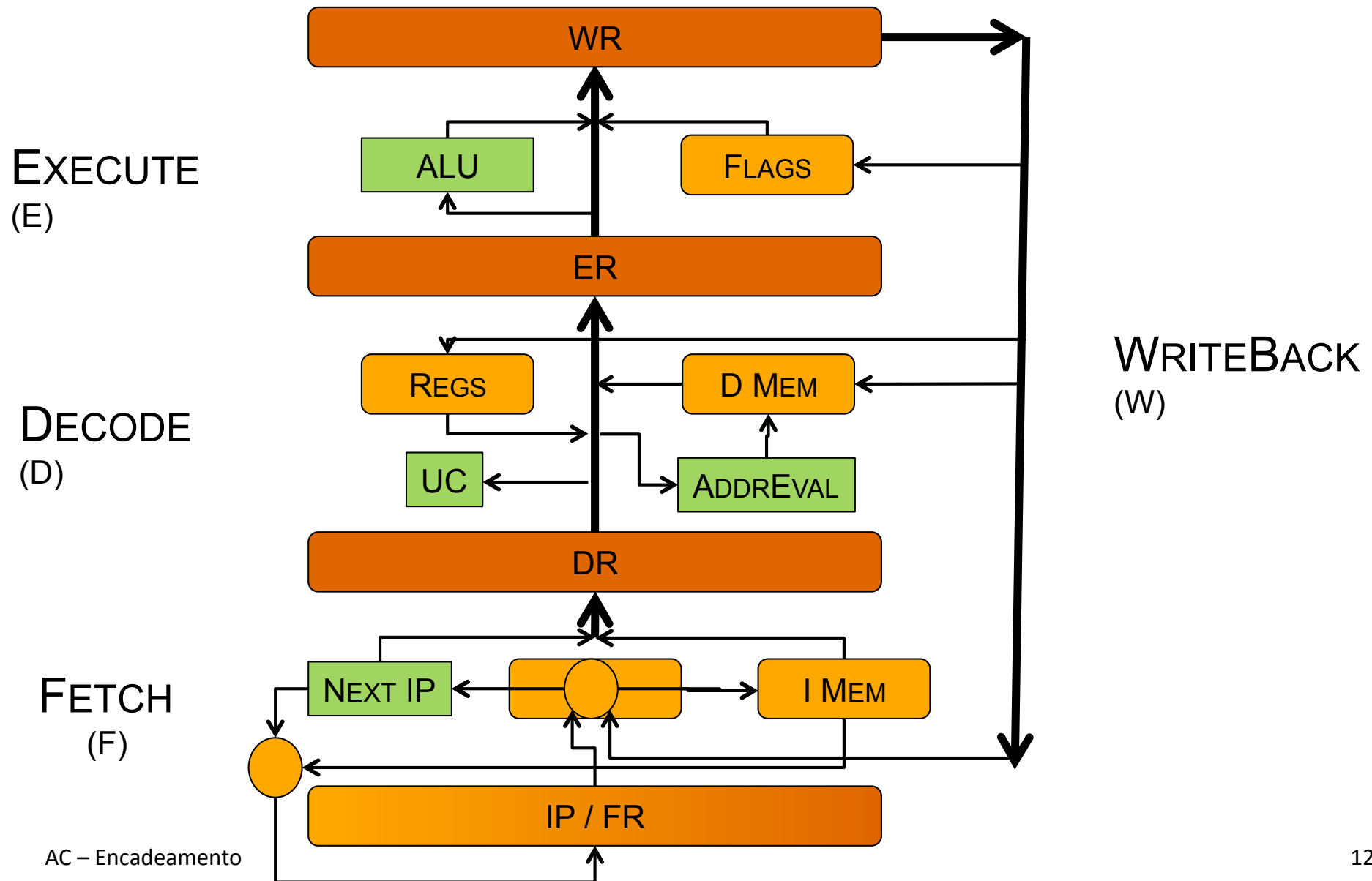
## *Instruction Level Parallelism*

- As arquiteturas 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).

# Arquitetura sequencial simples



# Arquitetura encadeada simples



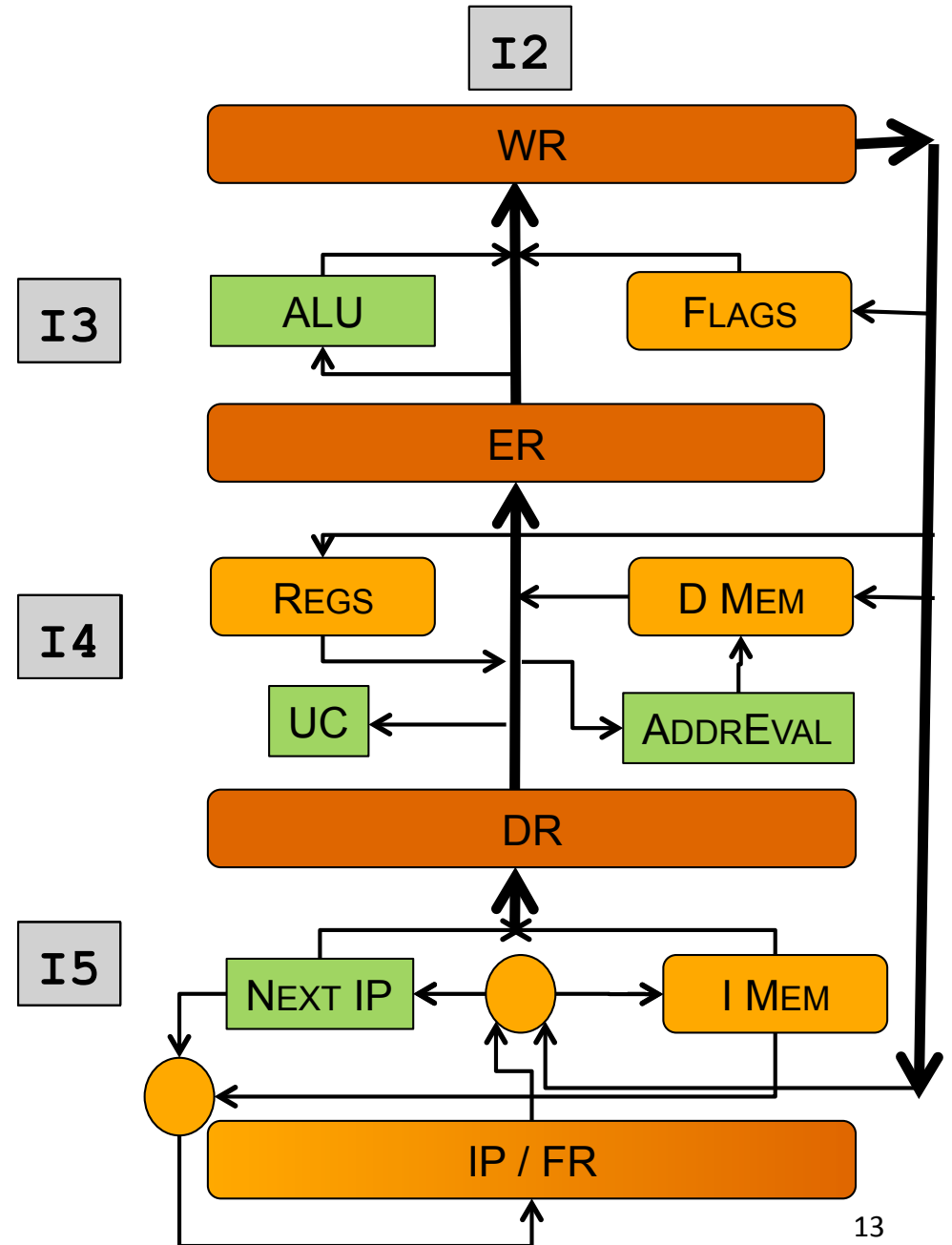
# Pipeline : Execução

```

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					

AC – Encadeamento



## Dependências de Controle - 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

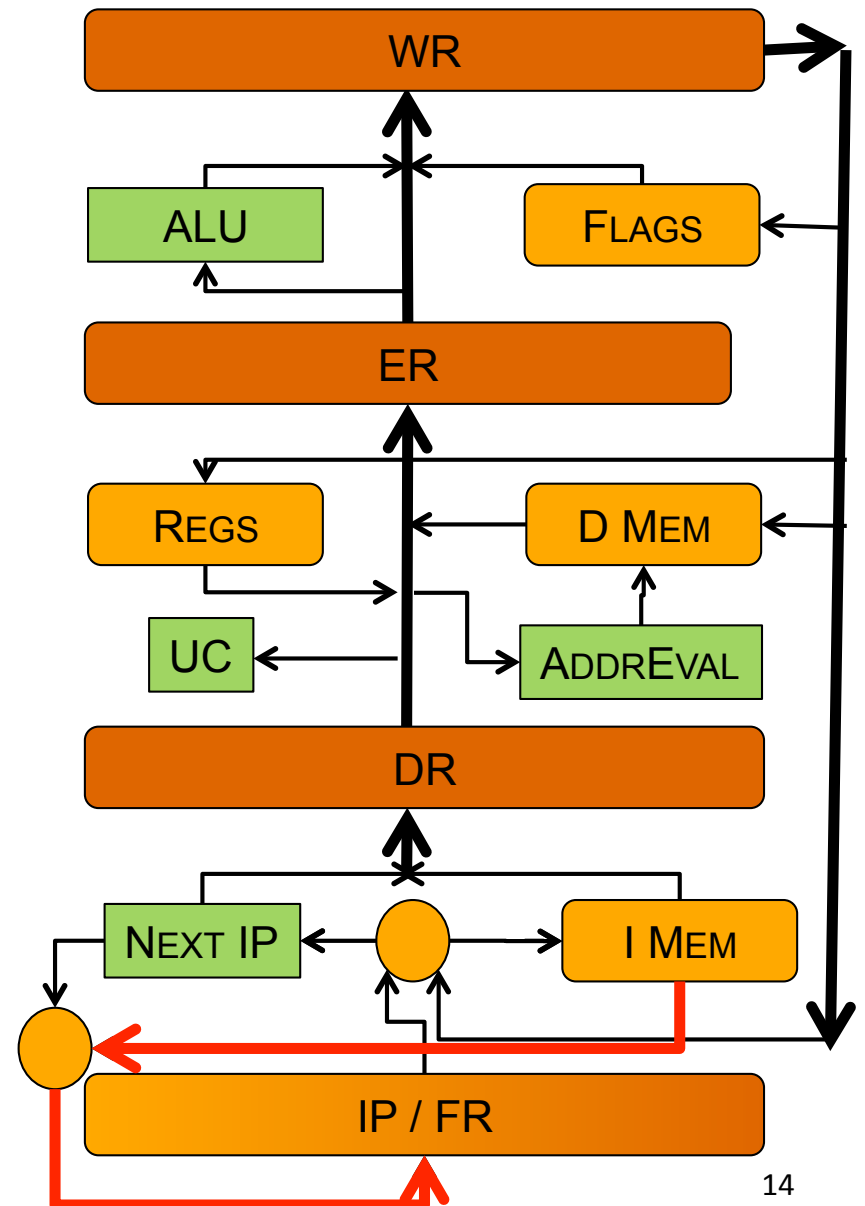
	1	2	3	4	5	6
I1	F	D	E			
I2		F	D			
I5			F			

AC – Encadeamento

I1

I2

I5



## Dependências de Controle - jXX

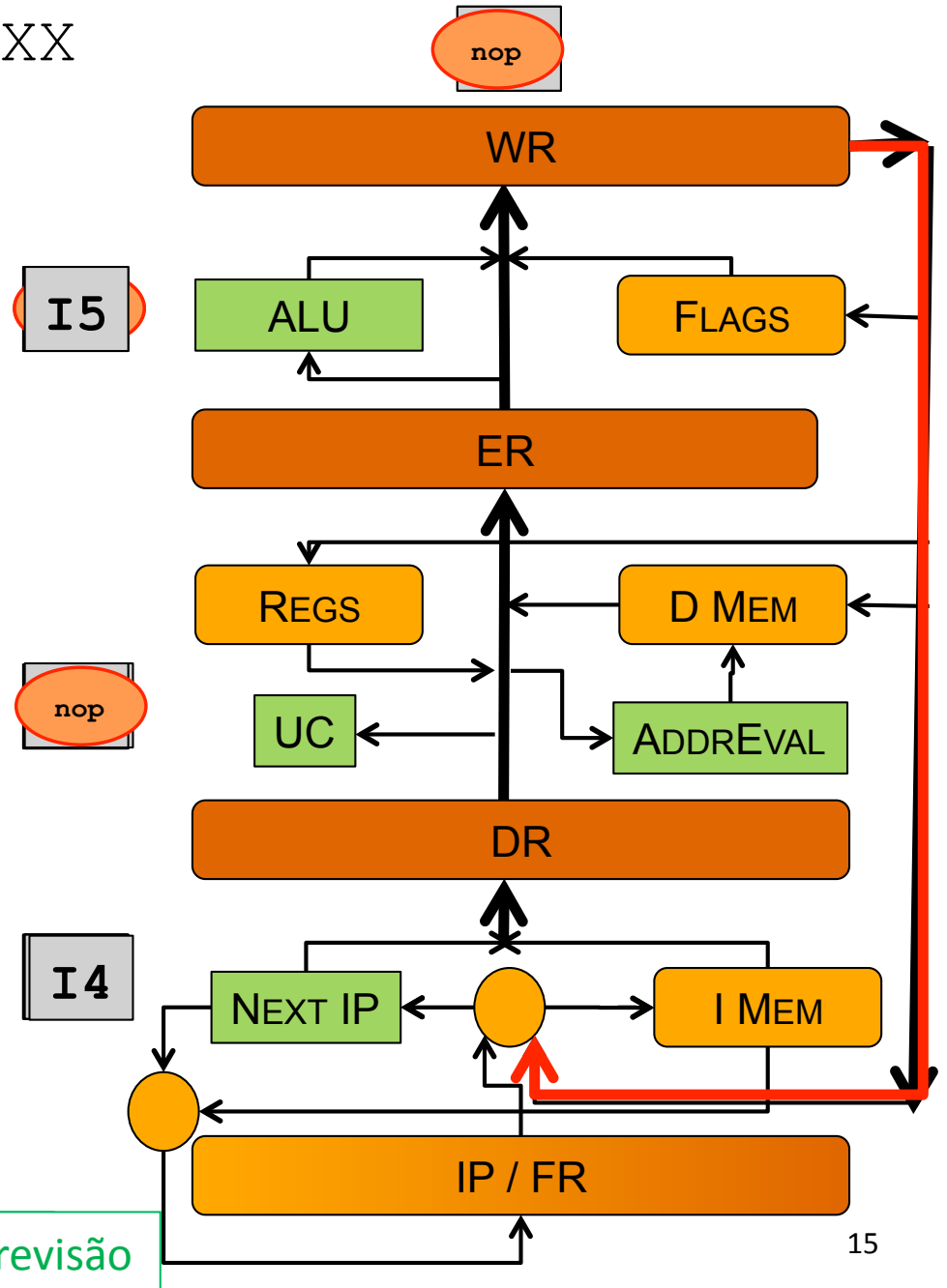
```

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

AC – Encadeamento

Branch To Corriga 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 nos estágios de “EXECUTE” (CC) e “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

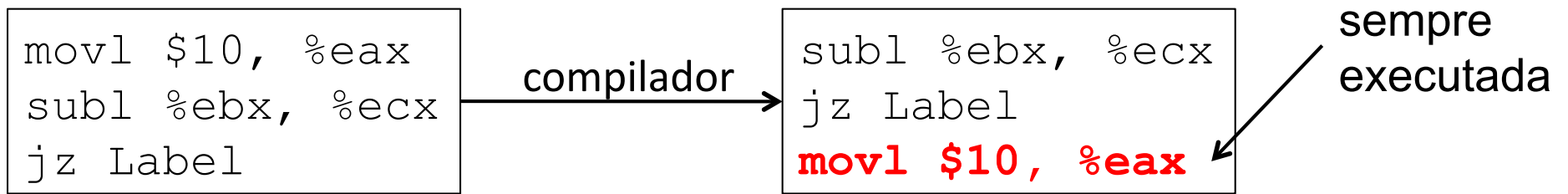
- A previsão é feita em tempo de execução baseada no historial recente
- *Branch prediction buffer* – tabela que guarda para cada instrução de salto do programa 1 *bit* indicando se o salto foi ou não tomado na última execução

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

# Dependências de Controlo - jXX

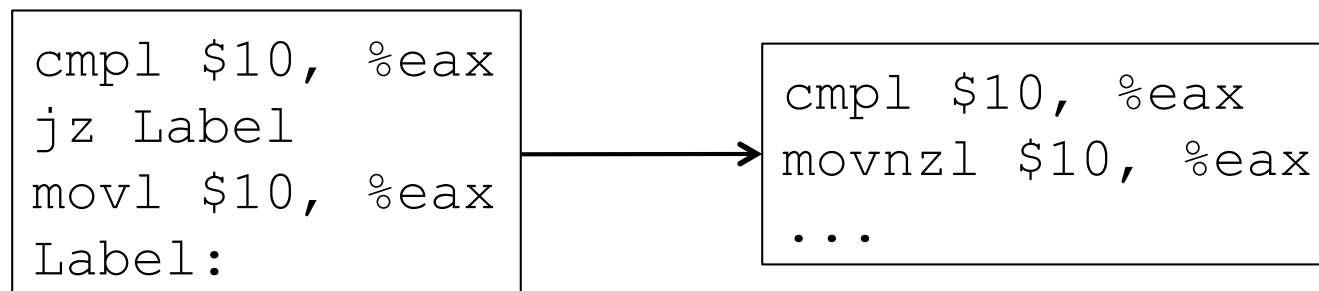
## Branch delay slot

- A instrução a seguir ao salto é sempre executada. Compete ao compilador colocar a seguir ao salto uma instrução executável (ou um `nop`).
- Técnica caiu em desuso devido à profundidade dos *pipelines* actuais



## Instruções condicionais

- instrução é executada dependendo das *flags*, reduzindo saltos condicionais.
- IA32 tem `moves` condicionais, o ARMv7 tem um campo condicional para quase todas as instruções.



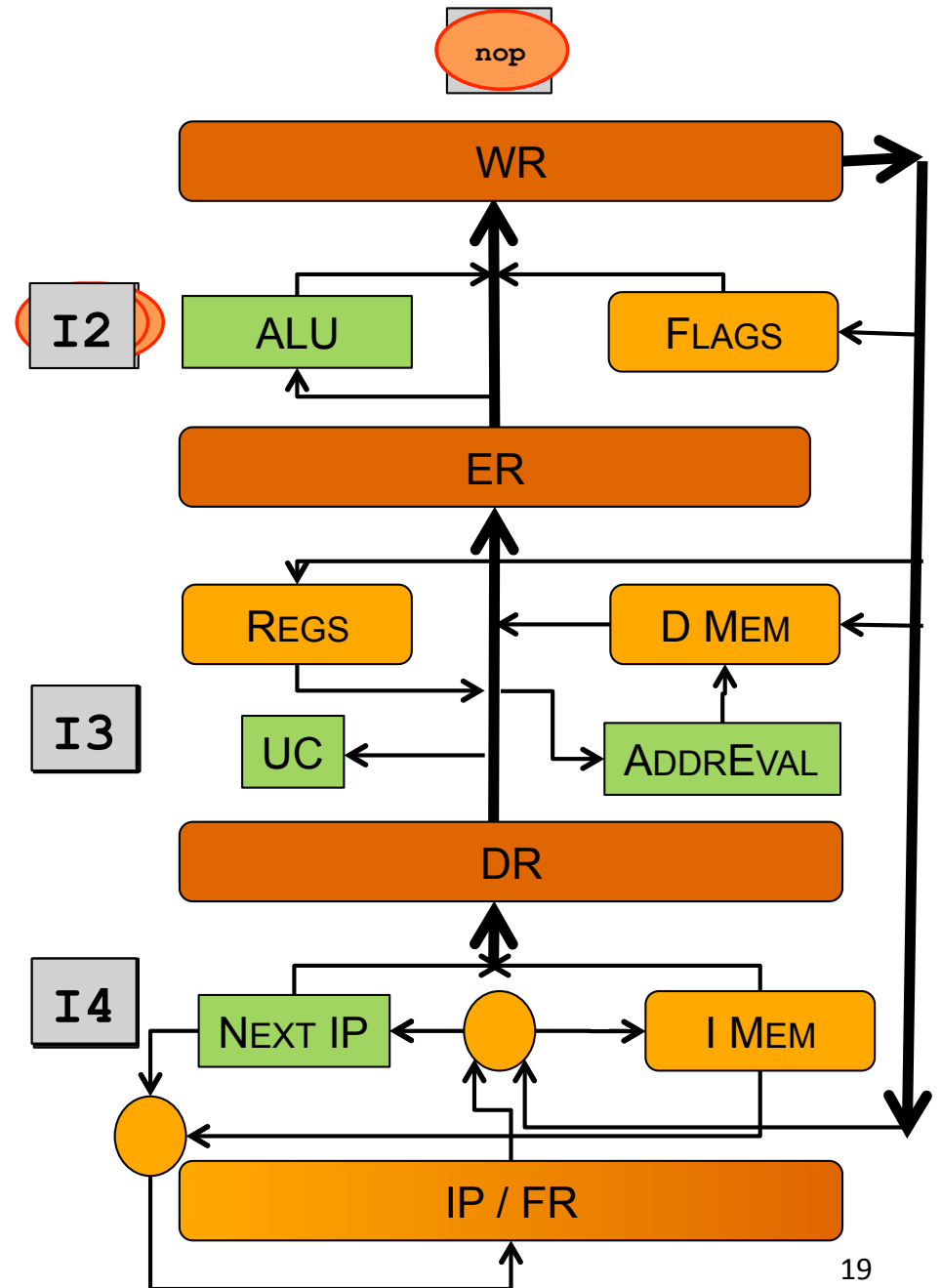
## Dependências de dados

```

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

	1	2	3	4	5	6
I1	F	D	E	W		
B1				E	W	
B2				E		W
I2		F	D	D	D	E
I3			F	F	F	D
I4						F

AC – Encadeamento



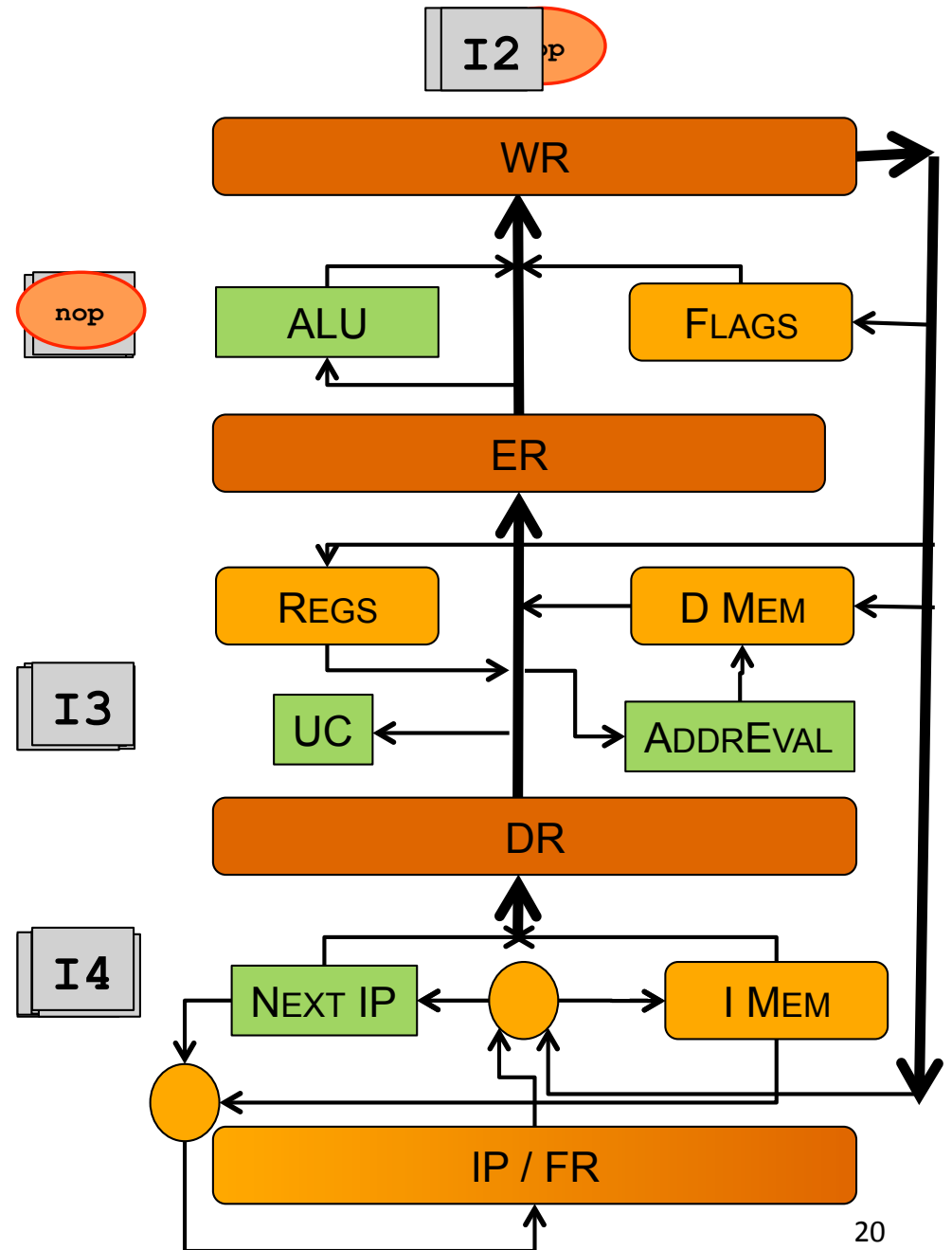
# Dependências de dados

```

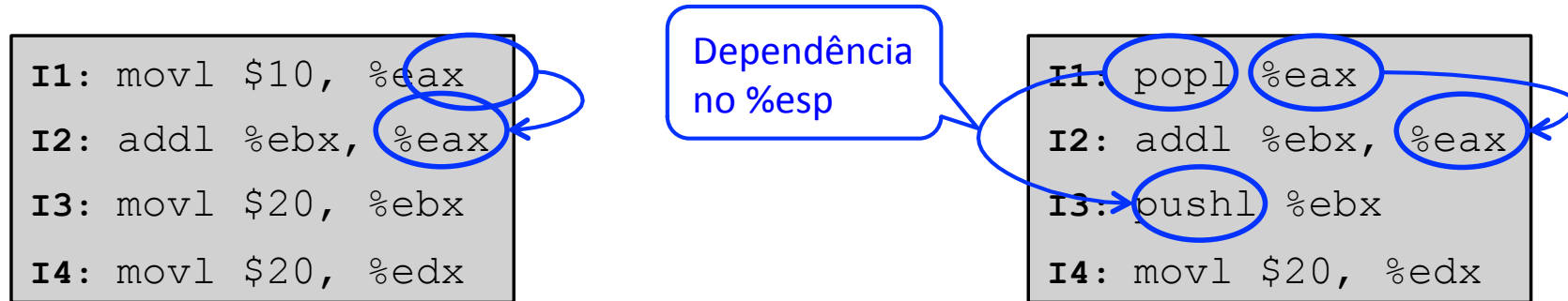
I1: movl $10, %eax
I2: movl $20, %ecx
I3: addl %ebx, %eax
I4: movl $20, %edx
    
```

	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
...						

AC – Encadeamento



# 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**
- Injectando “bolhas” (NOPs) no estágio de execução a leitura pode ser adiada até ao ciclo imediatamente a seguir à escrita

## *Data forwarding*: Motivação

- As dependências de dados são demasiado comuns
- Resolvê-las recorrendo à injeção 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)

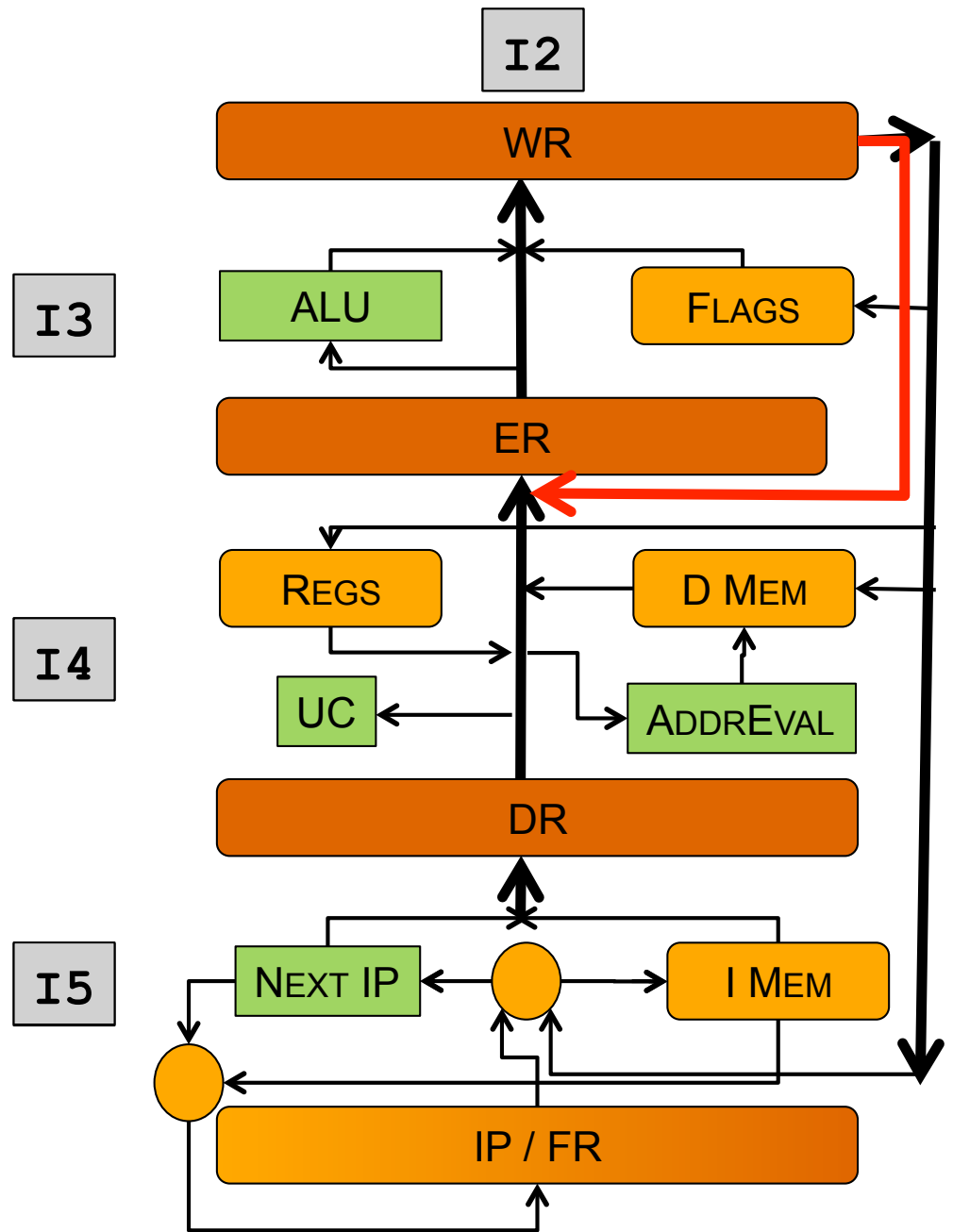
```

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					

AC – Encadeamento

Realimenta de WR para ER





## Exemplo de *Forwarding* (2)

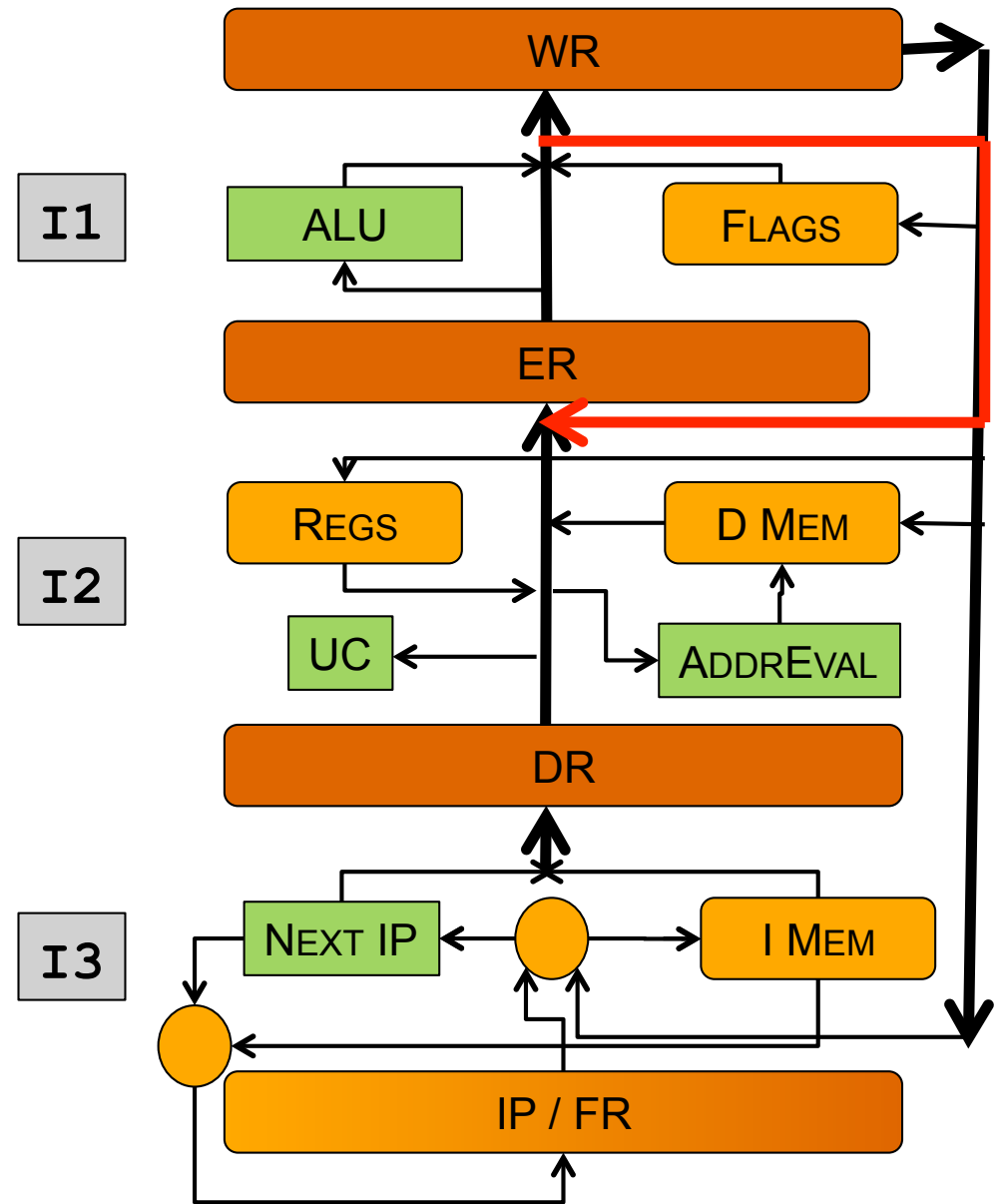
```

I1: irmovl $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					

AC – Encadeamento

Realimenta de E para ER



# ARM Cortex A8

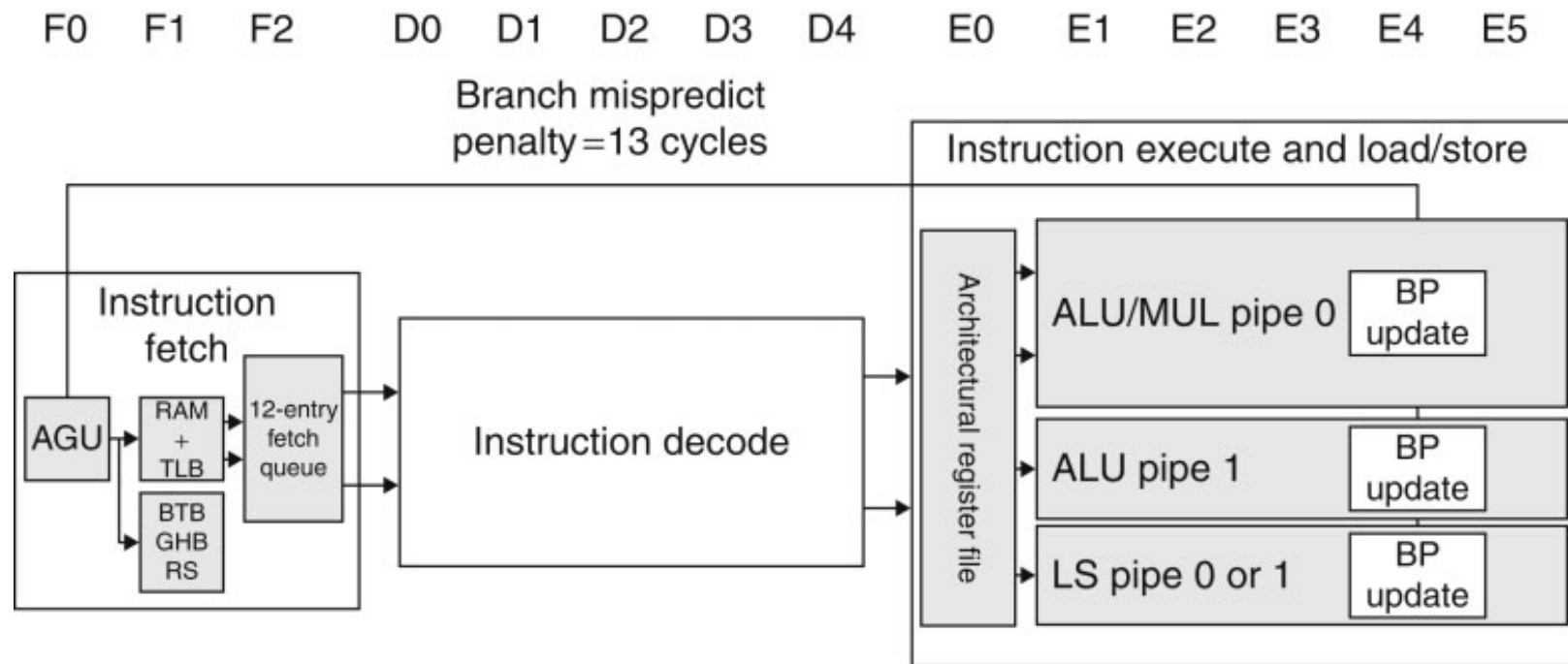


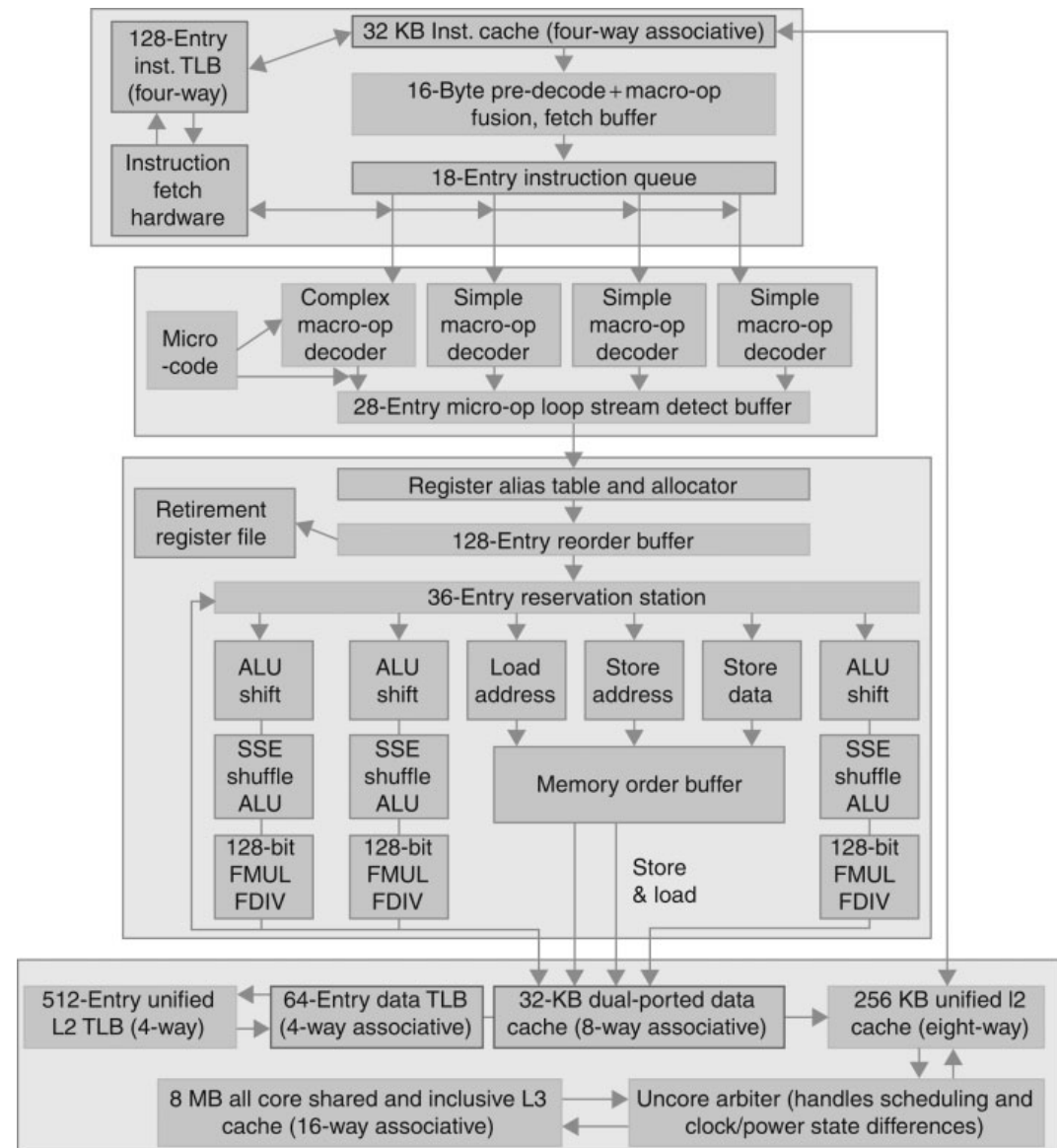
FIGURE 4.75 – Patterson & Hennessy; Computer Organization & Design – 5<sup>th</sup> Edition, Elsevier, 2013

The A8 pipeline (14 stages):

- . The first three stages fetch instructions into a 12-entry instruction fetch buffer. The *Address Generation Unit* (AGU) uses a *Branch Target Buffer* (BTB), *Global History Buffer* (GHB), and a *Return Stack* (RS) to predict branches to try to keep the fetch queue full;
- . Instruction decode is five stages;
- . instruction execution is six stages.
- . 2 execution pipelines: minimum CPI = 0.5 (median of 2.0 with SPEC2000 benchmarks)

# Intel Core i7 920

- instruções da arquitectura versus micro-ops
- registos arquitecturais versus registos físicos
- 6 unidades funcionais
- CPI mínimo = 0.25 (apenas 4 instruções podem ser convertidas em micro-ops simultaneamente),  
mediana da SPEC2000 benchmark = 0.76



# Pipeline: Resumo

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