Y86:
Encadeamento de Instruções (PIPE-)

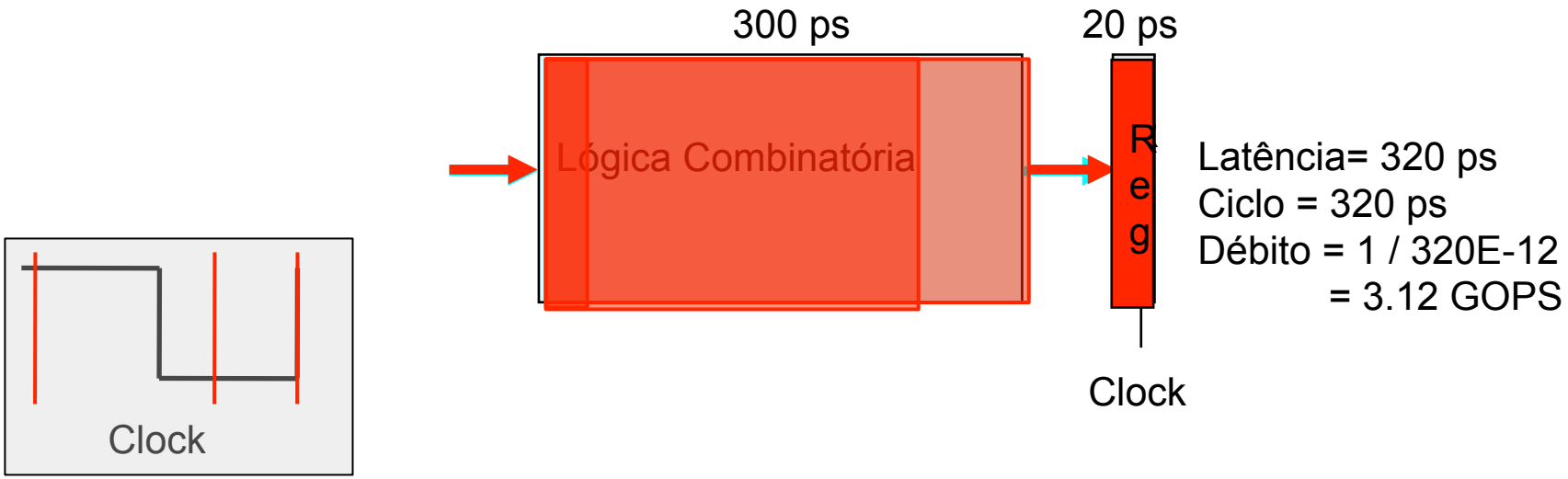
Arquitetura de Computadores

Lic. em Engenharia Informática

Y86: Encadeamento de instruções *(pipeline)*

Conteúdos	**3 – Organização do Processador**
	3.2 – *Datapath* encadeado (*pipeline*)
	3.3 – Dependências de Dados e Controlo
Resultados de Aprendizagem	R3.2 – Analisar e descrever organizações encadeadas de processadores elementares
	R3.3 – Caracterizar limitações inerentes a organizações encadeadas (dependências) e conceber potenciais soluções

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

Encadeamento na Vida Real

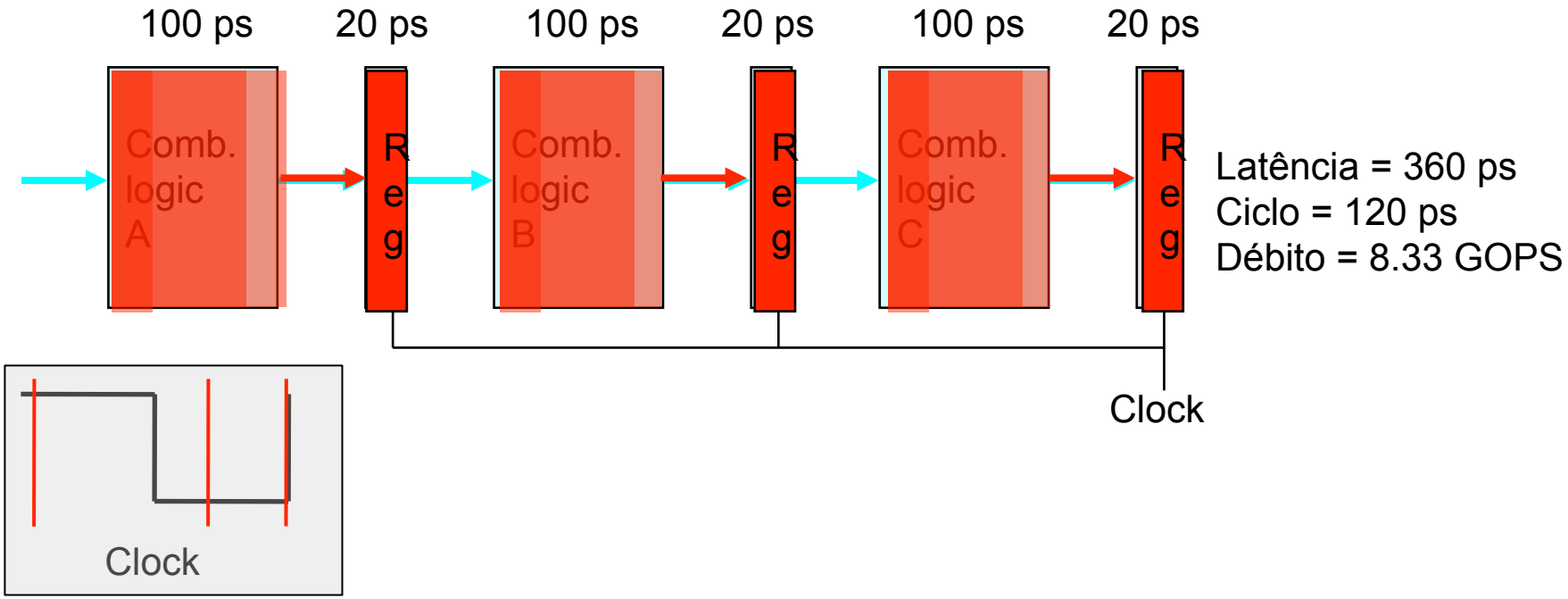
Sequencial

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

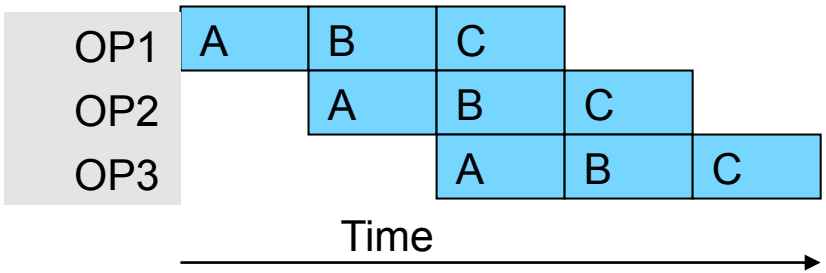


- Dividir lógica combinatória em 3 blocos de 100 ps cada
- Nova operação começa logo que uma termina o bloco A.
  - Ciclo >= 120 ps
- Latência aumenta (360 ps) mas débito também

Encadeamento: Diagramas

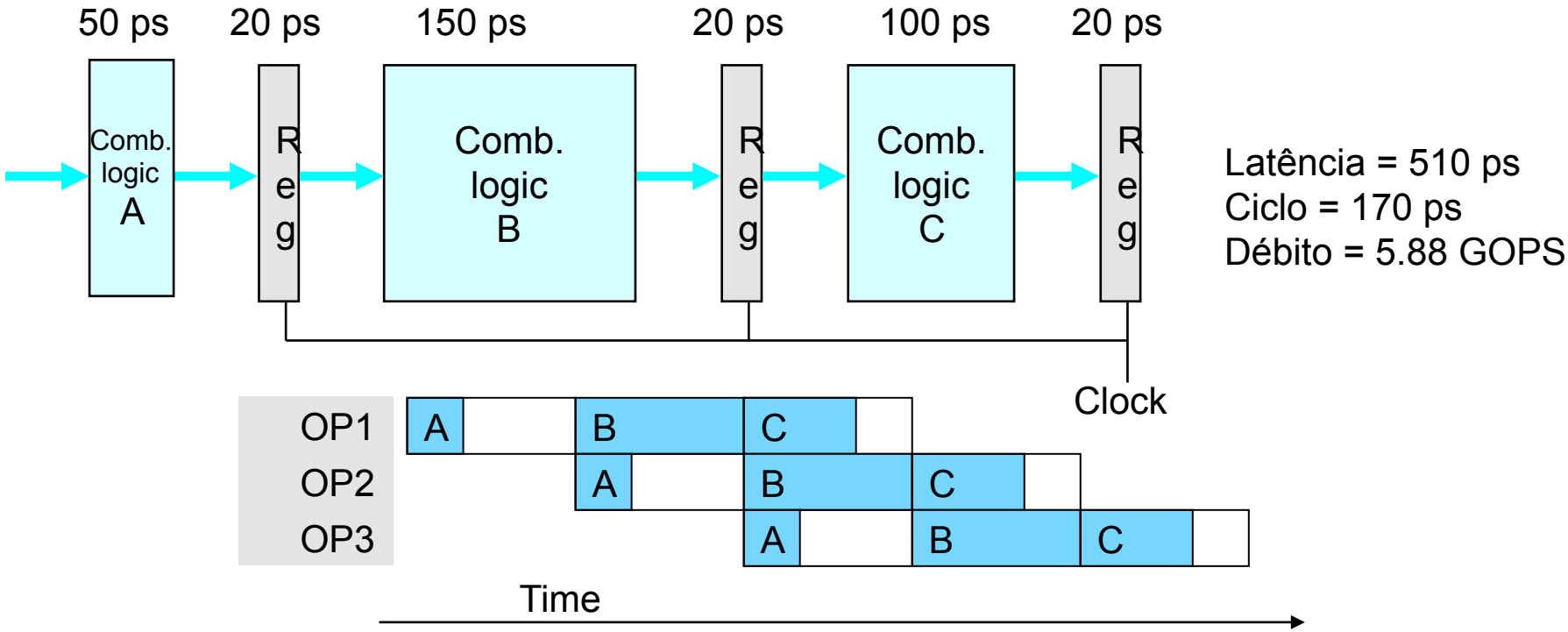
- Sequencial

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



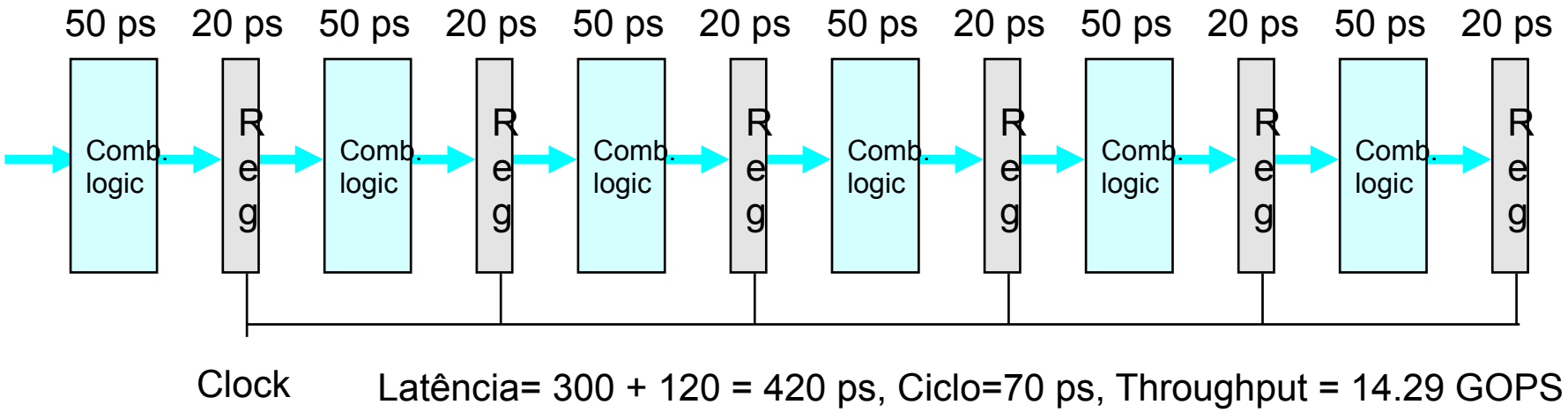
  - Débito: 3 operações simultâneas (ganho máximo de 3)
  - Latência: cada instrução necessita sempre de 3 ciclos

Limitações: Latências não uniformes



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

Limitações: custo do registo

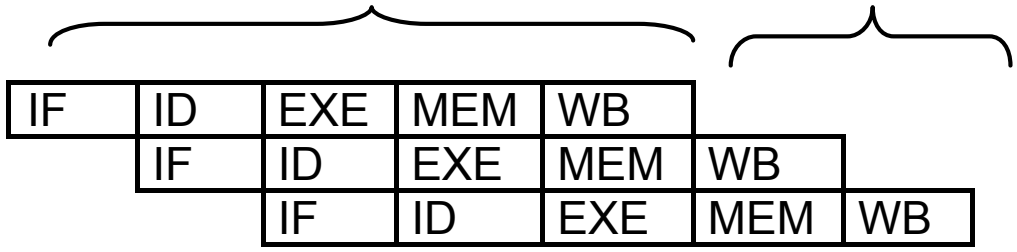


- Pipelines mais profundos têm maiores custos associados aos registos
- Percentagem de tempo devido aos registos por instrução:
  - 1-stage pipeline: 6.25% (020 em 320 ps)
  - 3-stage pipeline: 16.67% (060 em 360 ps)
  - 6-stage pipeline: 28.57% (120 em 420 ps)

Desempenho

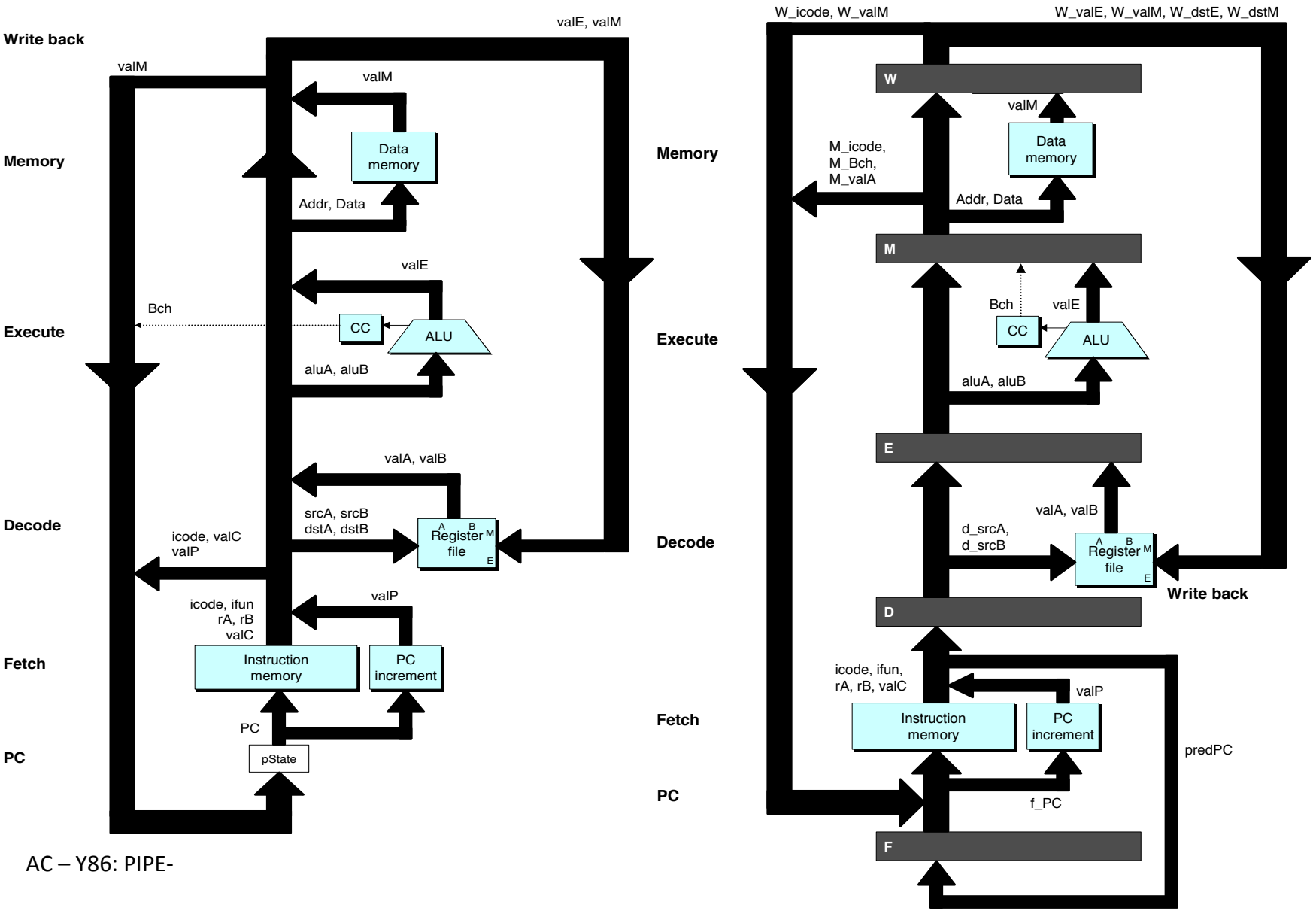
– Tempo (ideal) de execução (considerando CPI=1)

Texe*pipe* = [#estágios + (#instruções - 1)] x Tcc



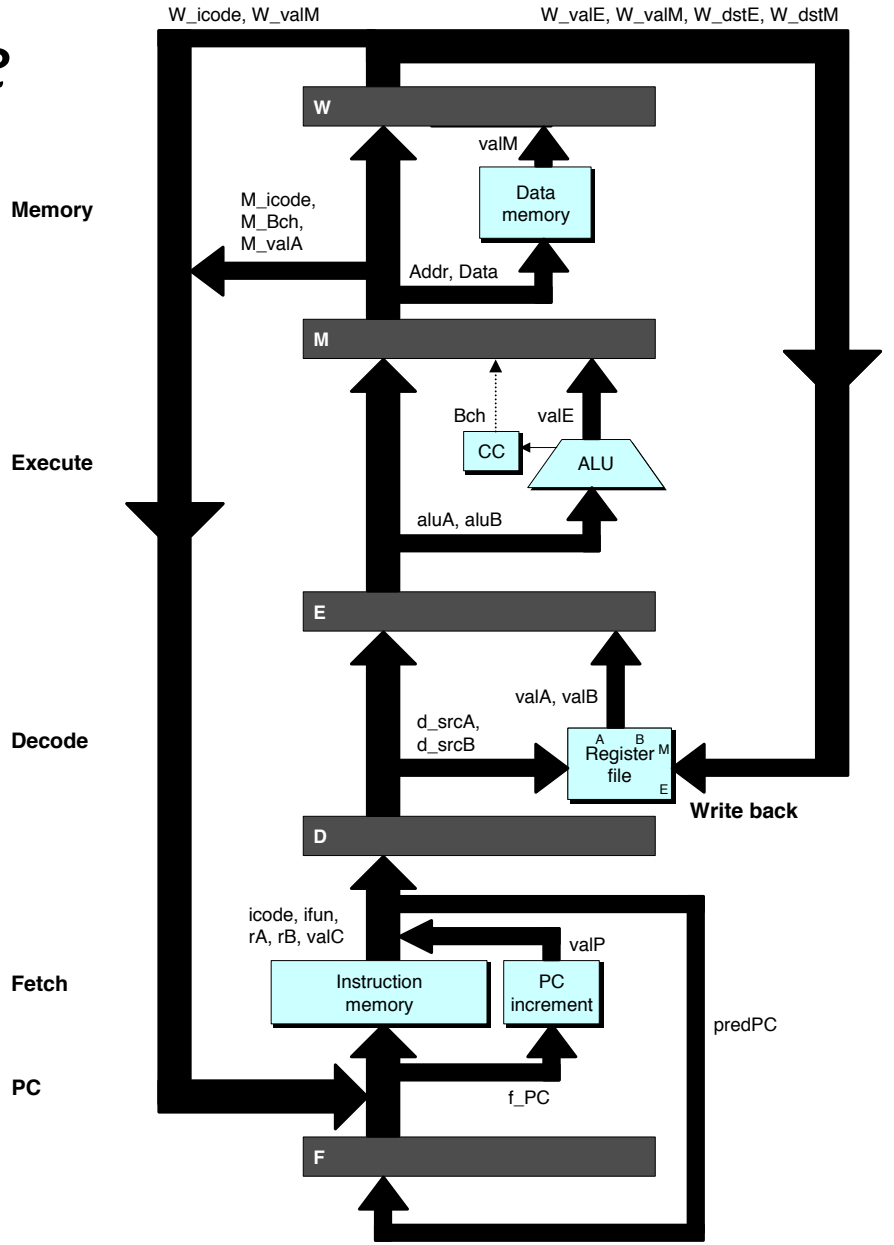
– Execução encadeada aumenta o débito (instruções realizadas por unidade de tempo) mas não diminui o tempo necessário para executar cada instrução
– O ganho relativo a arquiteturas sem encadeamento é potencialmente igual ao número de estágios
  – Tcc*pipe* = Tcc*seq* / #estágios

Y86: PIPE-


Write back
valE, valM
valM
Memory
valM
Data memory
Addr, Data
valE
Bch
Execute
CC
ALU
aluA, aluB
valA, valB
srcA, srcB
dstA, dstB
Decode
icode, valC
valP
A B M
Register file
E
valP
icode, ifun
rA, rB
valC
Fetch
Instruction memory
PC increment
PC
PC
pState
W_icode, W_valM
W_valE, W_valM, W_dstE, W_dstM
W
valM
Memory
M_icode,
M_Bch,
M_valA
Data memory
Addr, Data
M
Bch
valE
Execute
CC
ALU
aluA, aluB
E
valA, valB
d_srcA,
d_srcB
Decode
A B M
Register file
E
Write back
D
icode, ifun,
rA, rB, valC
valP
Fetch
Instruction memory
PC increment
predPC
PC
f_PC
F

Y86: Estágios do *pipeline*

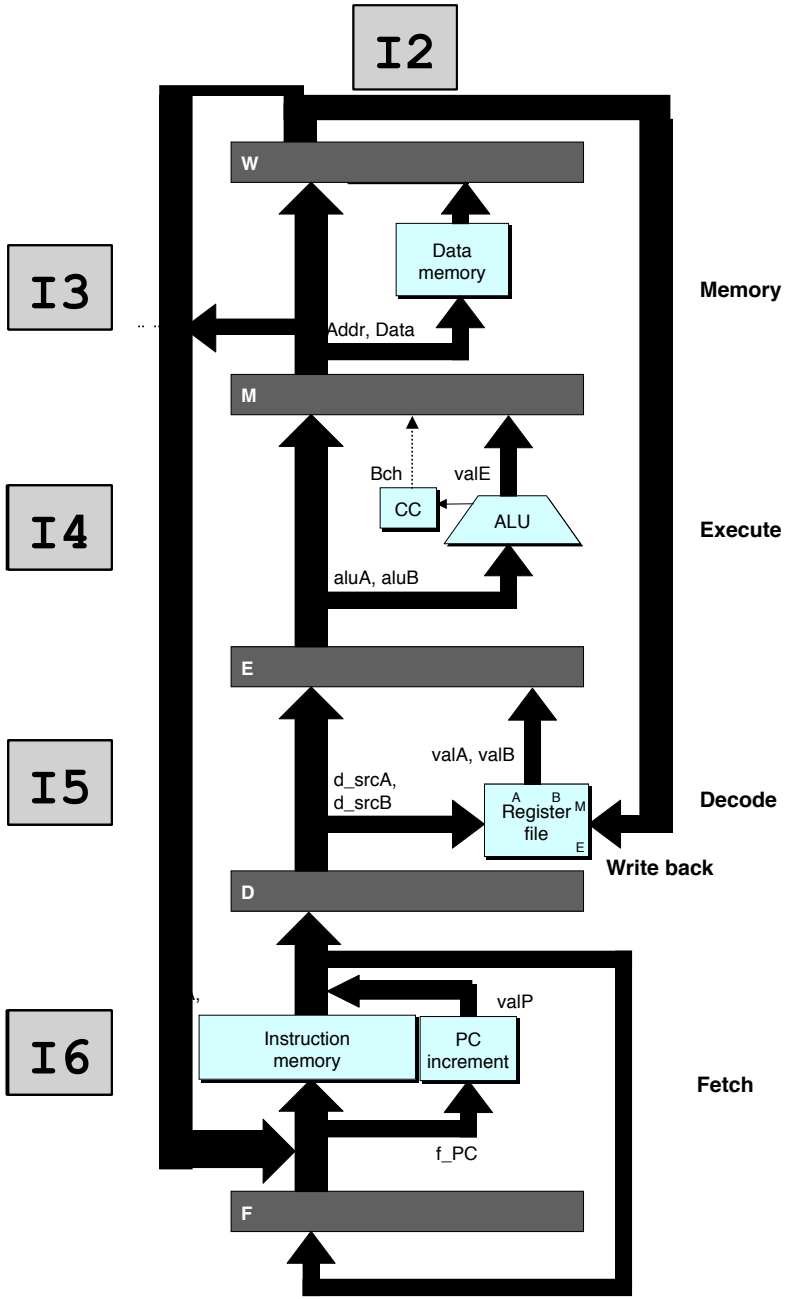
- Fetch
  - Seleccionar PC
  - Ler instrução
  - Calcular próximo PC
- Decode
  - Ler registos genéricos
- Execute
  - ALU
- Memory
  - Ler ou escrever na memória
- Write Back
  - Escrever nos registos



Y86 PIPE- : Execução

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

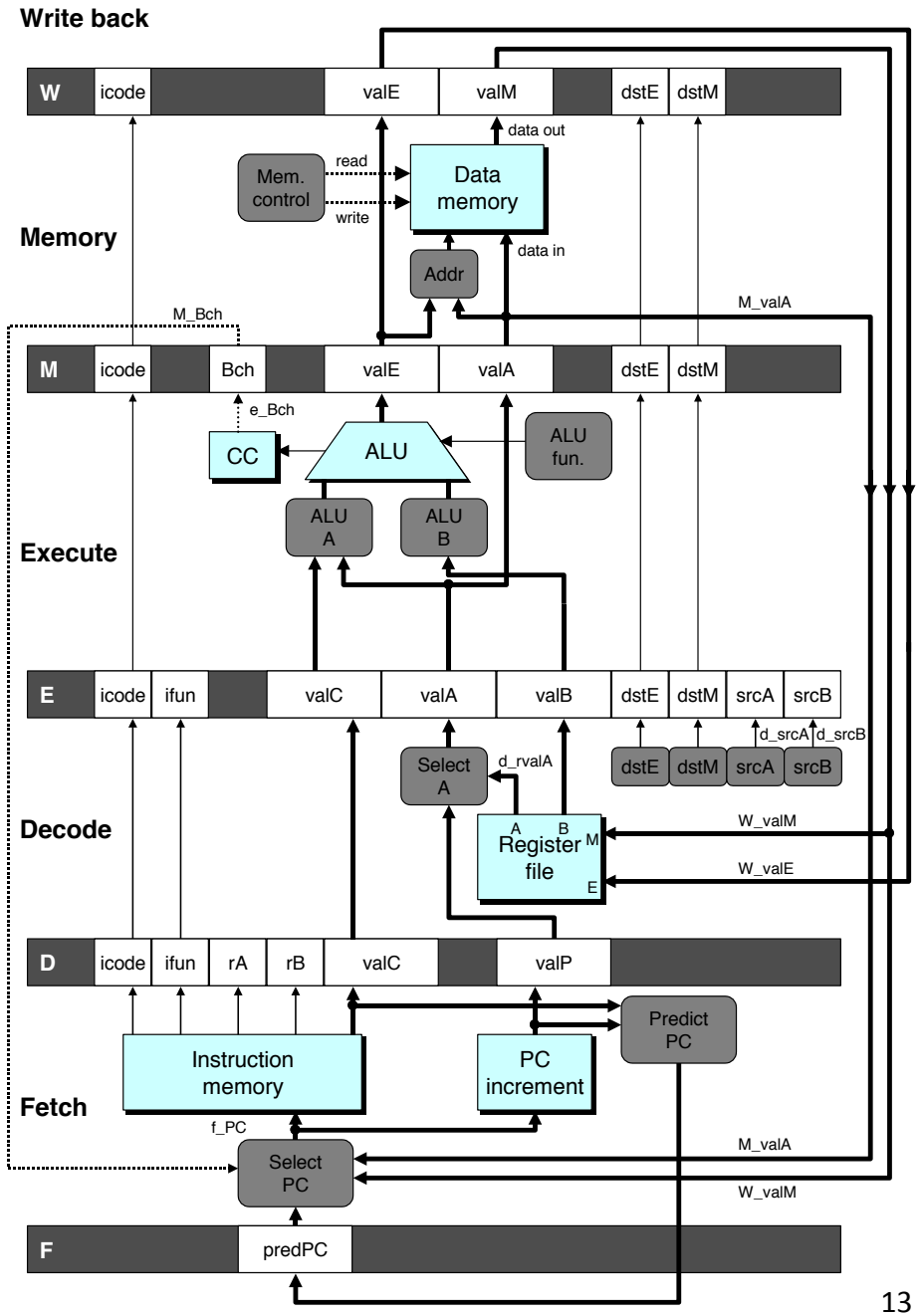
	1	2	3	4	5	6
I1	F	D	E	M	W	
I2		F	D	E	M	W
I3			F	D	E	M
I4				F	D	E
I5					F	D
I6						F



Y86 PIPE-

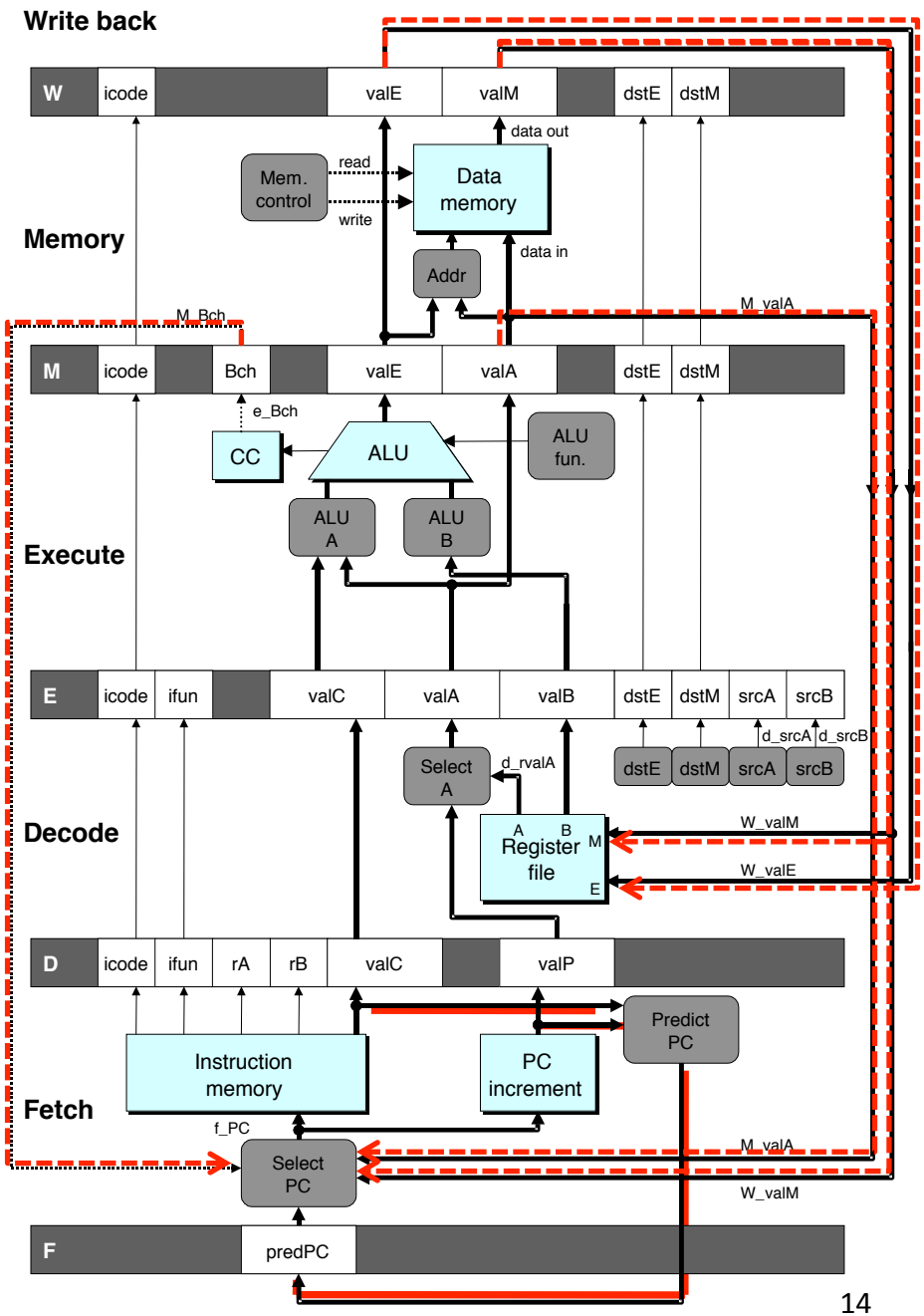
- Os registos do Pipeline contêm todos os valores intermédios necessários para executar a instrução

- Forward (Upward) Paths
  - Os valores passam de um estágio para o próximo
  - Não podem saltar estágios
    - ex., valC passa pelo estágio de Decode (apesar de não ser usado)



Y86 PIPE-: *feedback*

- Predict PC
  - Valor do próximo PC
    - `valP`: instruções genéricas
    - `valC`: `jmp, call`
- Saltos condicionais
  - tomado/não tomado
  - Alvo (`valC`) / Continua (`valP`)
- Endereço de retorno
  - Ler da memória
- Actualizar registos

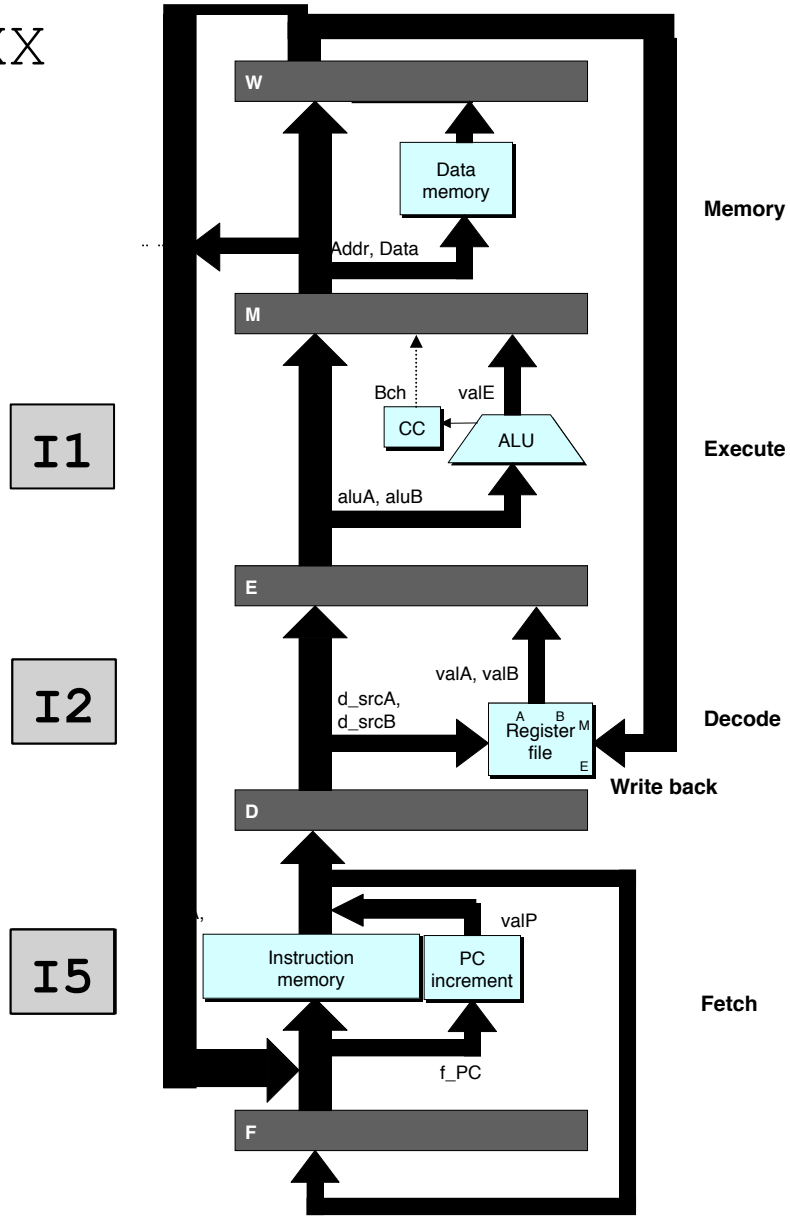


Y86: Dependências de Controlo - jXX

```
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			



Y86: Dependências de Controlo - `jXX`

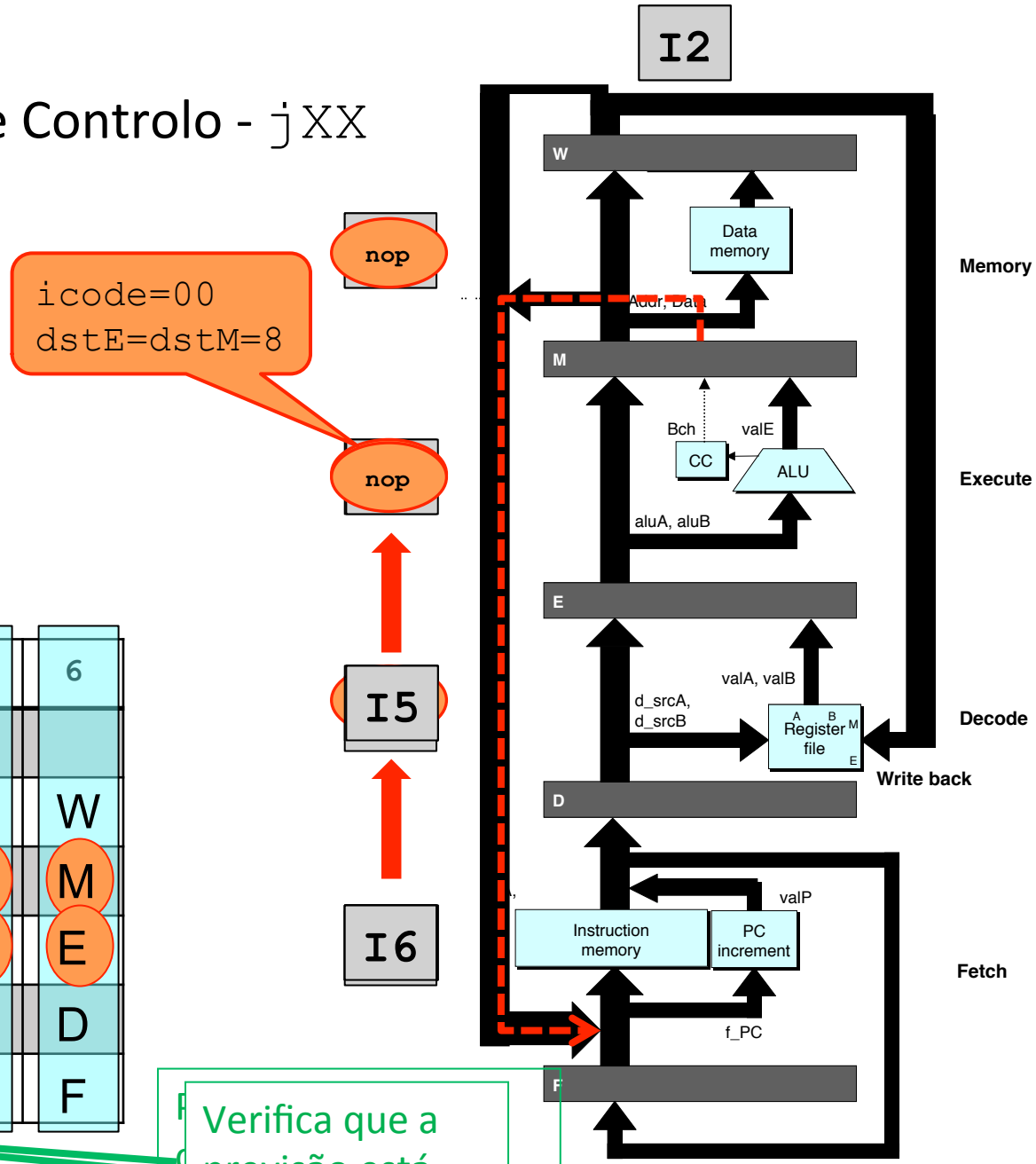
- Análises estatísticas mostram que os saltos condicionais são tomados em 60% dos casos
- Prevendo que o salto é tomado a decisão certa é tomada mais do que metade das vezes

- Alternativas:
  - NT – *Not Taken*
  - BTFNT – *Backward Taken, Forward Not Taken*

Y86: Dependências de Controlo - jXX

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

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



Verifica que a previsão está errada

Y86: Dependências de Controlo - `jXX`

- Prevê-se que o salto é sempre tomado
- A correcção da previsão é determinada 2 ciclos depois, quando a instrução de salto termina o estágio de execução
- Se a previsão estiver errada as 2 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 nas fases de "EXECUTE" (CC), "MEMORY" e "WRITEBACK" (Registos)
- *stall* do pipeline (injecção de "bolhas"): resulta num desperdício de um número de ciclos igual ao número de bolhas injectadas

Y86: Dependências de Controlo - `ret`

```
I1: call I3
I2: halt
I3: addl %ebx, %ebx
I4: addl %ecx, %ecx
I5: addl %ecx, %ecx
I6: ret
I7: addl %eax, %eax
```

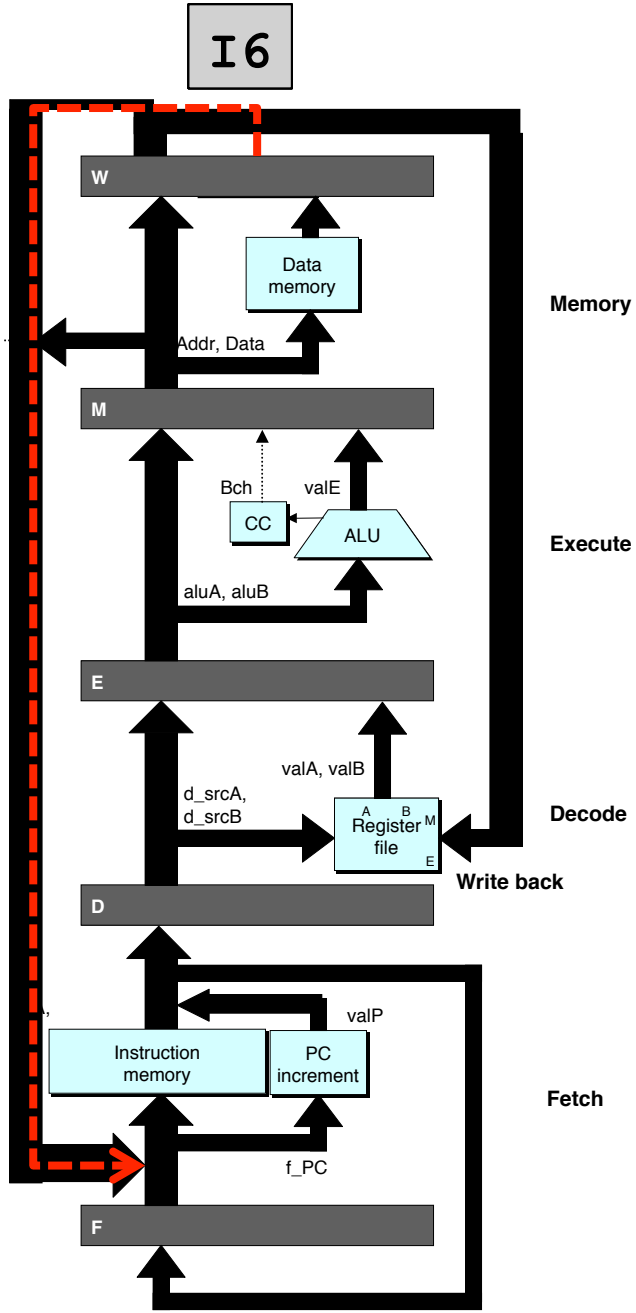
	1	2	3	4	5	6	7	8	9
I1	F	D	E	M	W				
I3		F	D	E	M	W			
I4			F	D	E	M	W		
I5				F	D	E	M	W	
I6					F	D	E	M	W
I7						F	D	E	M
I7							F	D	E
I7								F	D
I2									F

nop

nop

nop

I2

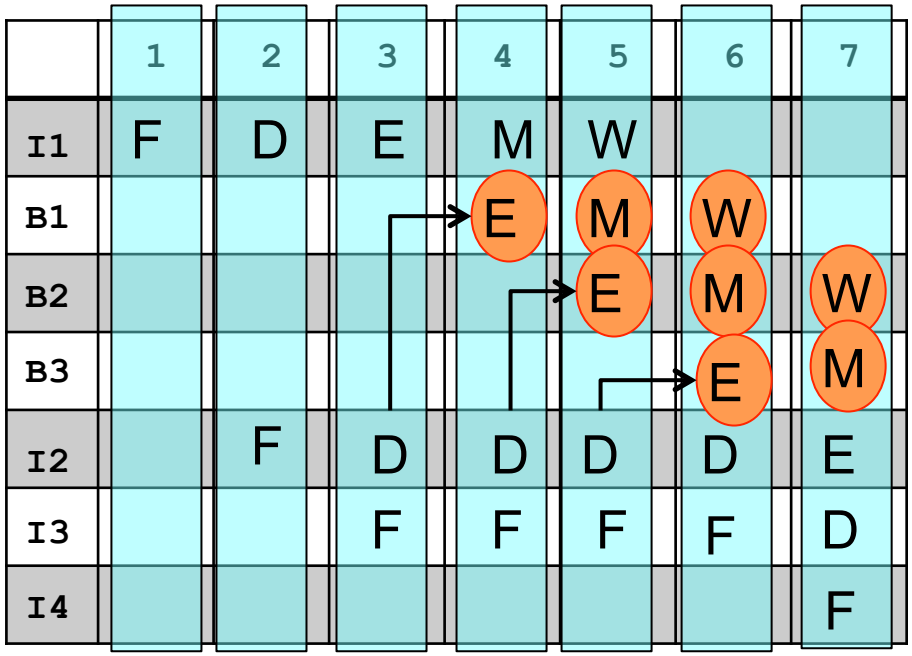


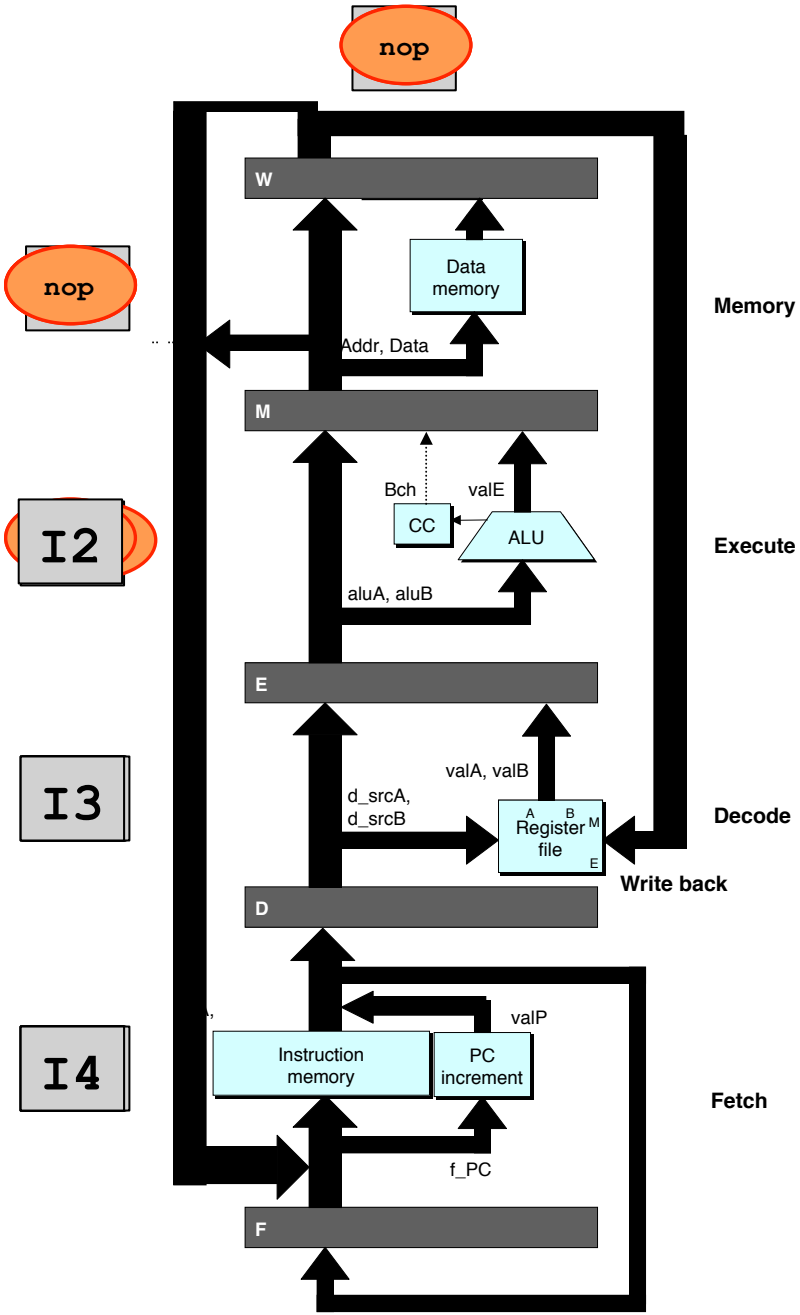
Y86: Dependências de Controlo - `ret`

- Não é possível prever o destino do salto, pois este endereço encontra-se no topo da pilha
- A memória é lida apenas durante o estágio "MEMORY"
- O endereço de retorno estará disponível apenas 4 ciclos depois do início da execução do `ret (W_valM)`
- Obriga a injectar 3 "bolhas"
- Alternativas:
  - *Stack* específica para endereços de retorno (`call / ret`)

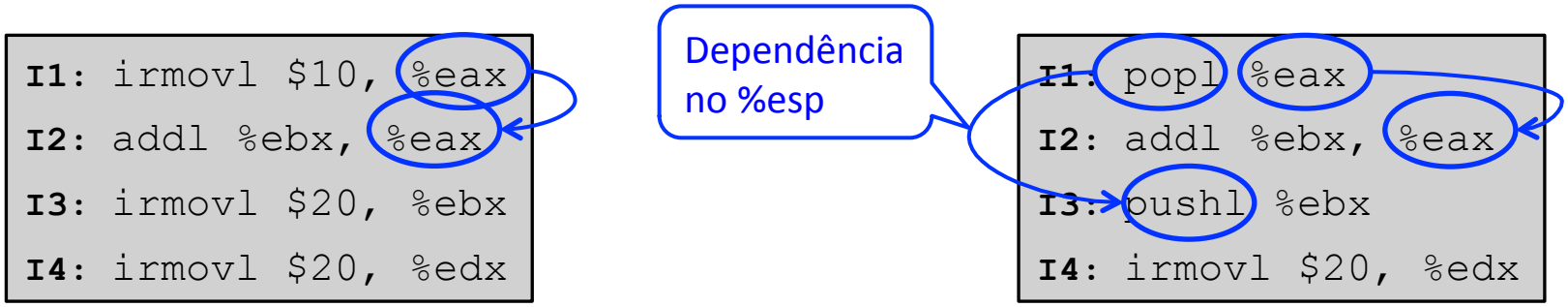
Y86: Dependências de dados

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





Y86 – 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**
- Na versão PIPE- a leitura tem que ser adiada até ao ciclo imediatamente a seguir à escrita
- Isto é conseguido injectando "bolhas" no estágio de execução

Y86 PIPE- : Sumário

- Conceito
  - Dividir execução das instruções em 5 estágios
- Limitações
  - Dependências entre instruções resolvidas injectando “bolhas”
    Pode resultar num número significativo de ciclos desperdiçados
  - Dependências de Dados
    - Uma instrução escreve um registo, outra lê-o mais tarde
    - Exigem a injecção de “bolhas”
  - Dependências de Controlo
    - Saltos condicionais: pode ser carregado um valor errado no PC (erro de previsão)
    - Return: não pode ser previsto qual o endereço de retorno
    - Exigem a injecção de “bolhas”