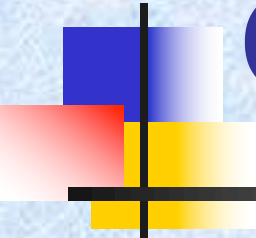
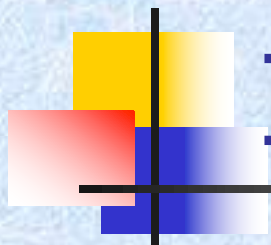


# Fundamentos de Sistemas Computacionais



---

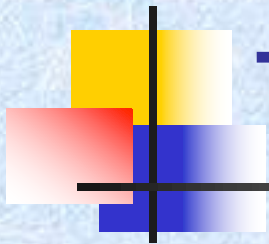
Programa de Pós-Graduação em Informática  
01/2022



# Informações Gerais

---

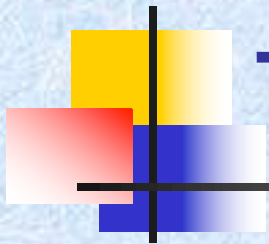
- Prof. Dra Alba Cristina M. A. Melo
  - Graduação em PD – UNB - 1986
  - Mestre em Ciência da Computação - UFRGS - 1991
  - PhD em Informática, sub-área sistemas operacionais paralelos - INPG - Grenoble, France – 1996
  - Pós-Doutorado na University of Ottawa, Canada, 2008
  - Visiting Scientist na Université Paris-Sud, France, 2011
  - Estágio Senior no Barcelona Supercomputing Center, Espanha, 2013
  - Visiting Scientist no INPG, France, 2018.
  - Pesquisadora CNPq/PQ 1C



# Tópicos

---

- Organização de Computadores
  - Revisão: componentes do computador, ciclo de instruções, pipelining, barramentos, caches, memória RAM
  - Arquiteturas RISC: características, RISC x CISC
  - Instruction Level Parallelism: Execução fora de ordem, Previsão de Desvios, Execução superescalar, Execução especulativa, VLIW



# Tópicos

---

- Organização de Computadores (cont)
  - Multiprocessadores simétricos (SMP): organização, coerência de caches
- Bibliografia básica:
  - Computer Organization & Architecture, W. Stallings.
  - Computer Architecture: A Quantitative Approach, Hennessy and Patterson.





# Tópicos

---

- Sistemas Operacionais
  - Revisão: estruturas, gerência de processos, memória, arquivos e E/S.
  - Micro-Kernels
  - Exo-Kernels
- Bibliografia básica:
  - Modern Operating Systems, A. Tanenbaum.
  - Operating System Concepts, A. Silberchatz, J. Peterson, P. Galvin.

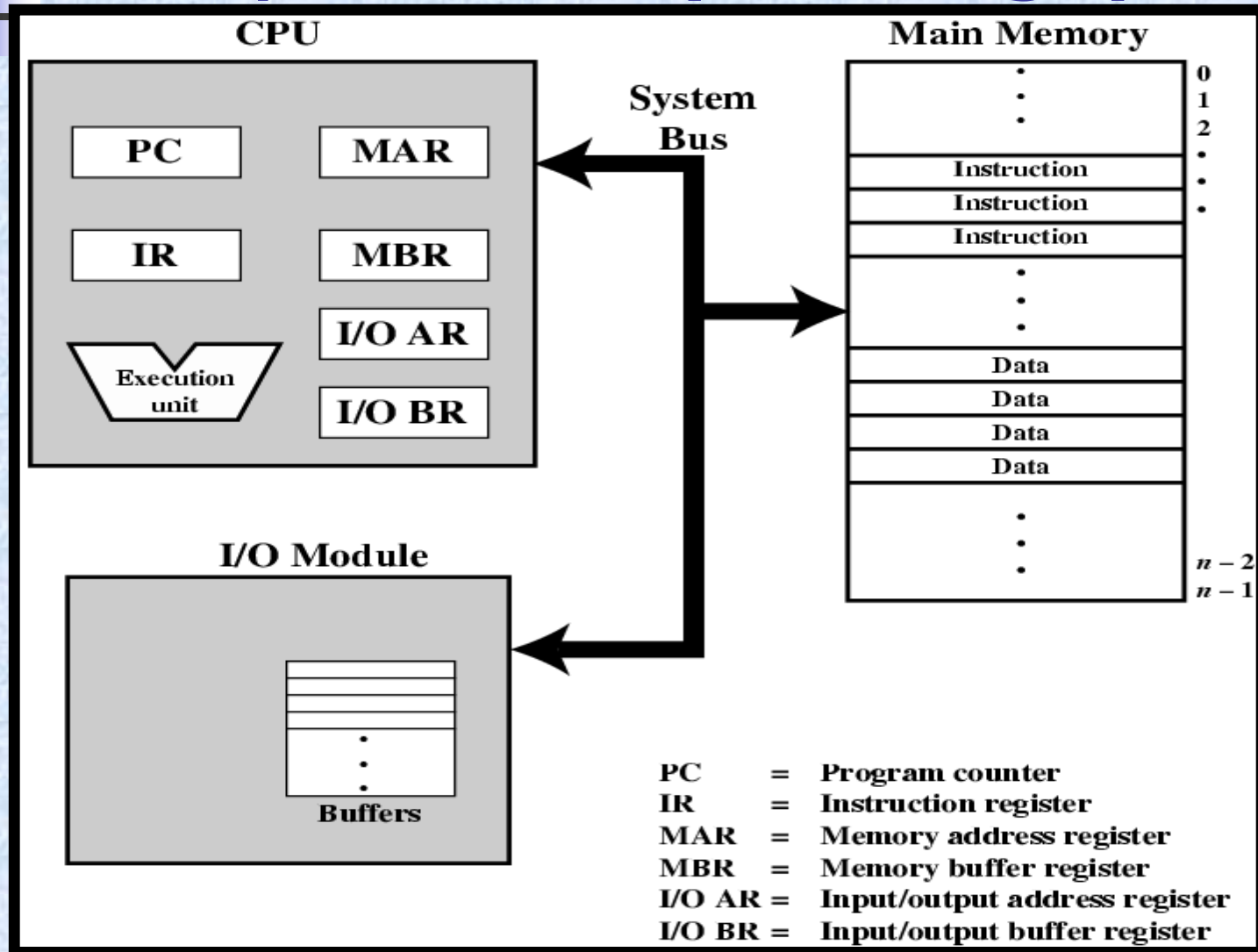
# Arquitetura de Computadores x Organização de Computadores

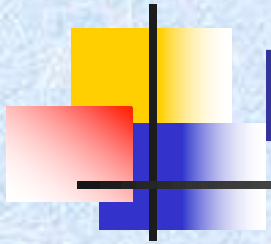


---

- Arquitetura de Computadores
  - Refere-se a atributos do sistema que são visíveis ao programador
  - Ex: conjunto de instruções, endereçamento de memória
- Organização de Computadores
  - Refere-se às unidades operacionais do computador e às interconexões entre as mesmas
  - Ex: sinais de controle, tecnologias de memória
- Ênfase em Organização de Computadores

# Componentes de um Computador (Stallings)



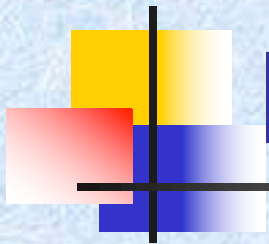


# Execução de Programas

---

1. No início do ciclo de instrução, o processador busca da memória a instrução cujo endereço está no PC (program counter).
2. A instrução é carregada no IR (instruction register).
3. O processador decodifica a instrução.



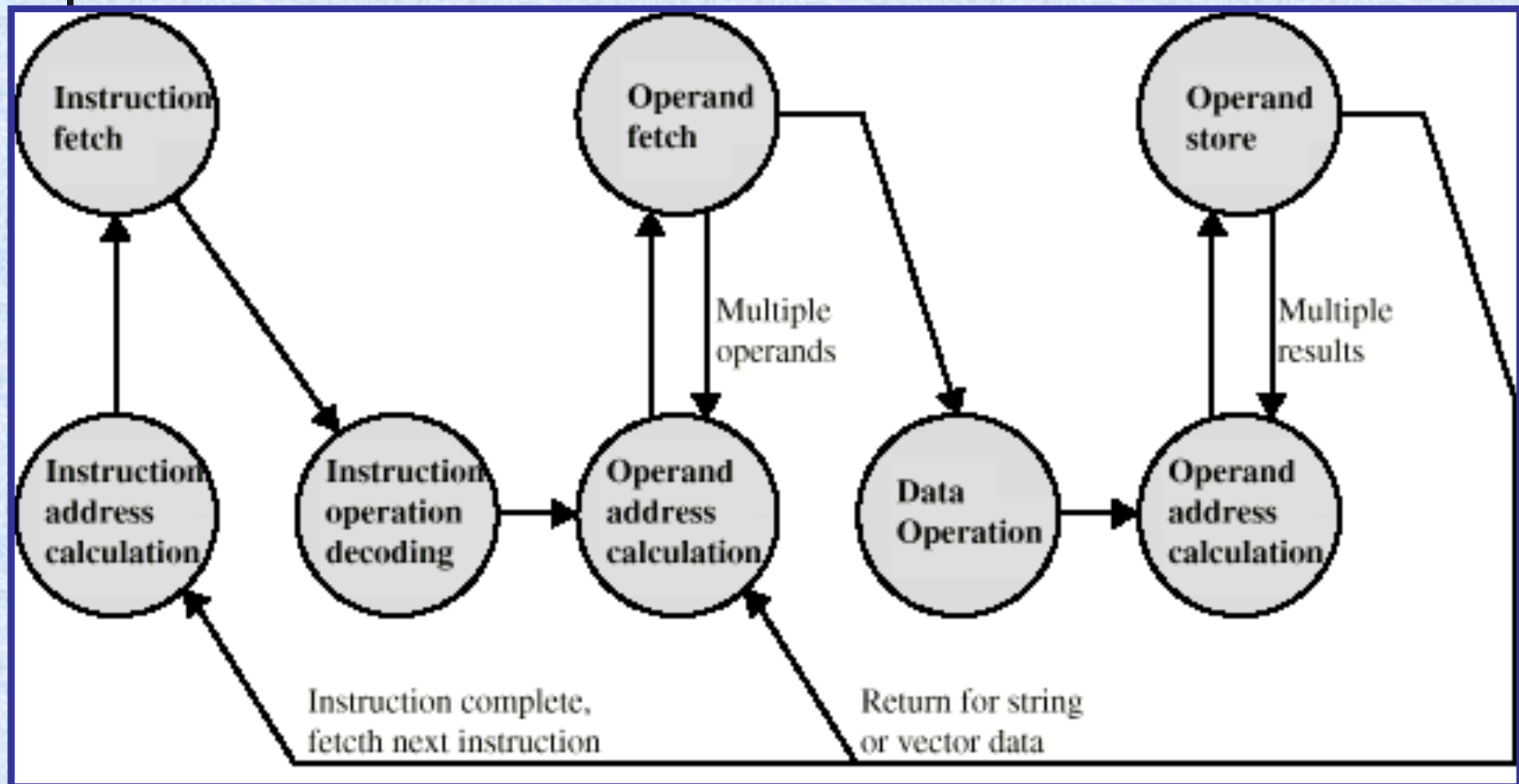


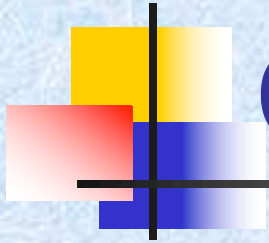
# Execução de Programas

---

4. A instrução decodificada é executada. Pode ser:
  - a) Transferência de dados entre processador e memória
  - b) Transferência de dados entre periféricos e processador
  - c) Execução de operações lógicas e aritméticas sobre os dados
  - d) Alterações na sequência de execução das instruções

# Ciclo de Instruções (Stallings)

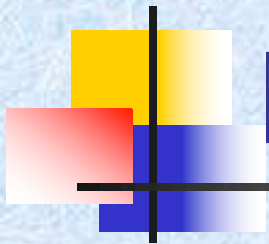




# Ciclo de Instruções (Stallings)

---

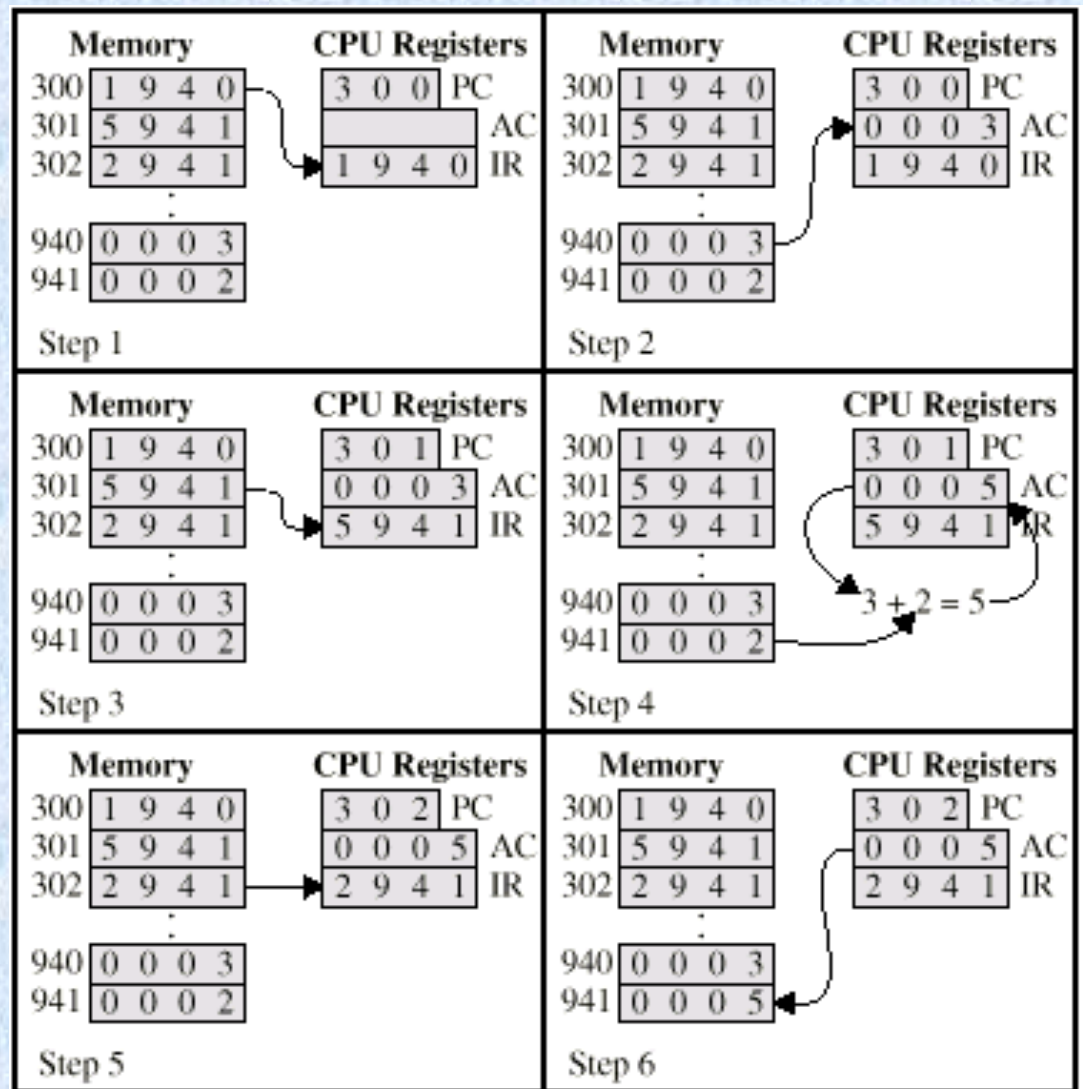
- IAC: determina o endereço da próxima instrução
- IF: carrega a instrução no processador
- ID: determina o tipo de instrução e os operandos necessários
- OF: busca o operando
- DO: executa a operação
- OS: escreve o resultado



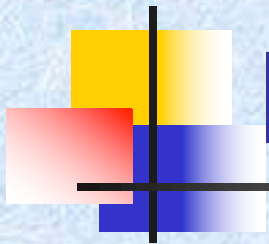
# Exemplo de Execução

## Programa:

LOAD AC,#(940)  
ADD AC,#(941)  
STORE AC,#(941)



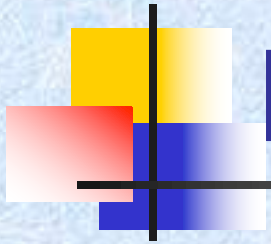




# Desvios

---

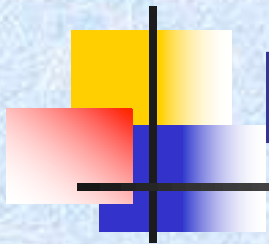
- No caso de execução sequencial, o endereço da próxima instrução é obtido através de uma operação de soma (add) no PC.
- Infelizmente, em geral, a execução não é estritamente sequencial.
- Em alguns pontos do código, decisões são tomadas que fazem com que caminhos alternativos sejam seguidos.
- Ifs e Loops são exemplos destas decisões.



# Desvios

---

- Os ifs e loops (for, while, do), que constam nas linguagens de programação de alto nível são traduzidos pelo compilador para instruções de máquina.
- As instruções de máquina que implementam os desvios são branches ou jumps.



# Desvios - Exemplo

---

LOAD AC2, 943 /\* carrega o valor 943 em AC2

LOAD AC1, 940 /\* carrega o valor 940 em AC1

L1: LOAD AC0,#(AC1) /\* carrega o conteúdo de AC1 em AC0

ADDC AC1,1 /\* AC1 = AC1 +1

ADD AC0,#(AC1) /\* AC0 = AC0 +(AC1)

STORE AC0,#(AC1) /\* armazena o valor de AC0 em AC1

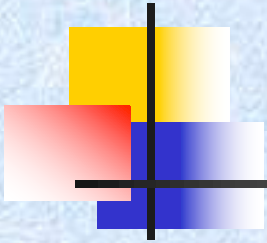
SUB AC3, AC2, AC1 /\*AC3 = AC2 - AC1

BNEZ L1 /\*se o valor de AC3 for  $\neq 0$ , desvia para L1

....

940	1
941	2
942	3
943	4

# Desvios - Exemplo



LOAD AC2, 943 /\* carrega o valor 943 em AC2

LOAD AC1, 940 /\* carrega o valor 940 em AC1

L1: LOAD AC0, #(AC1) /\* carrega o conteúdo de AC1 em AC0

ADDC AC1, 1 /\* AC1 = AC1 + 1

ADD AC0, #(AC1) /\* AC0 = AC0 + #(AC1)

STORE AC0, #(AC1) /\* armazena o valor de AC0 em AC1

SUB AC3, AC2, AC1 /\* AC3 = AC2 - AC1

PC →

BNEZ L1 /\*se o valor de AC3 for  $\neq 0$ , desvia para L1

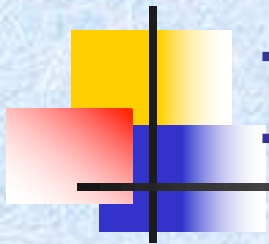
940	1	AC0	AC1	AC2	AC3
941	3	10	943	943	0
942	6				
943	10				





# Pipelining

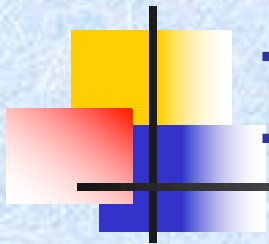
---



# Introdução

---

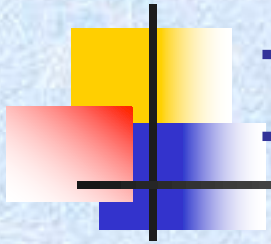
- O pipelining permite que a execução de múltiplas instruções seja sobreposta
- A execução de uma instrução é dividida em estágios
- Uma instrução não se executa mais rápido, porém o programa como um todo sim: maior throughput
- A grande maioria dos processadores atuais utiliza pipelining



# Introdução

---

- Decomposição da execução em estágios (exemplo DLX)
  - IF: Busca a instrução da memória e carrega em IR
  - ID: Decodifica a instrução e acessa registradores
  - EX: Executa operações
  - Mem: Executa loads, stores e branches
  - WB: escreve os valores nos registradores ou na memória



# IF (Instruction Fetch)

---

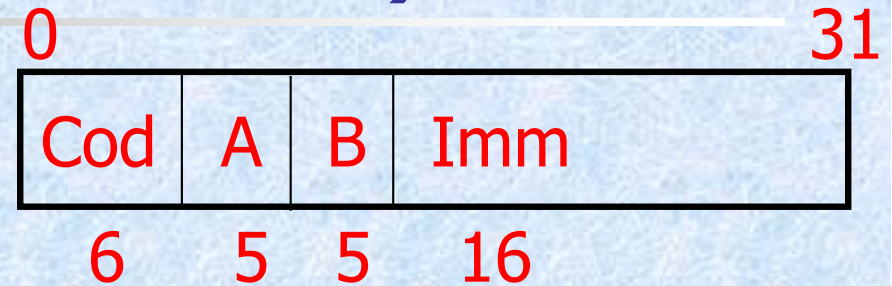
- $IR \leftarrow Mem[PC];$
- $NPC \leftarrow PC + 4;$
- Busca a instrução em memória, na posição determinada pelo PC e a carrega no registrador de instruções
- Incrementa o PC de 4, determinando o endereço da próxima instrução, e coloca este valor em NPC

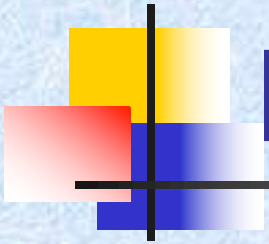




# ID (Instruction Decode)

- $A \leftarrow \text{Regs [IR6...10]}$
- $B \leftarrow \text{Regs [IR11...15]}$
- $\text{Imm} \leftarrow ((\text{IR16})...)$
- Decodifica a instrução e acessa o conjunto de registradores, que são lidos para 2 registradores temporários (A e B).
- Os 16 bits menos significativos de IR são armazenados em Imm
- A decodificação e busca de operandos é feita em paralelo (decodificação de campo fixo).

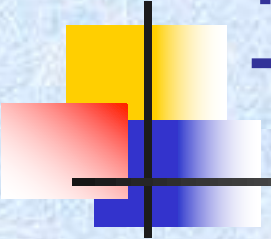




# EX (Execution)

---

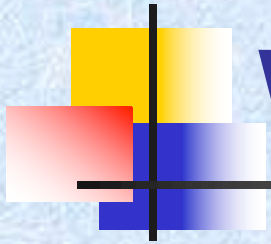
- A operação a ser executada depende da instrução:
  - Referência à memória
    - $ALUOutput \leftarrow A + Imm$
    - O endereço efetivo é calculado
  - Instrução ALU registrador-registrador
    - $ALUOutput \leftarrow A \text{ func } B$
  - Instrução ALU registrador-imediato
    - $ALUOutput \leftarrow A \text{ op } Imm$
  - Branch
    - $ALUOutput \leftarrow NPC + Imm$
    - $Cond \leftarrow (A \text{ op } 0)$



# MEM (Memory Access / Branch Termination Cycle)

---

- Referência à Memória
  - $LMD \leftarrow Mem[ALUOutput]$  ou
  - $Mem[ALUOutput] \leftarrow B$
- Se a instrução for um load, o dado é lido da memória e colocado no registrador LMD.
- Se for um store, os dados contidos no registrador B são escritos em memória
- Branch
  - if (cond)  $PC \leftarrow ALUOutput$  else  $PC \leftarrow NPC$ ;
- Se não for branch:
  - $PC \leftarrow NPC$

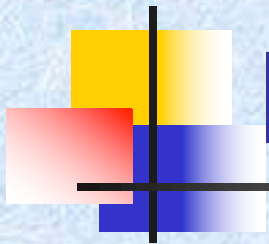


## WB (Write Back)

---

- Instrução ALU registrador-registrador
  - $\text{Regs}[\text{IR16} \dots \text{IR20}] \leftarrow \text{ALUOutput}$
- Instrução ALU registrador-immediate
  - $\text{Regs}[\text{IR11} \dots \text{IR15}] \leftarrow \text{ALUOutput}$
- Load
  - $\text{Regs}[\text{IR11} \dots \text{IR15}] \leftarrow \text{LMD}$





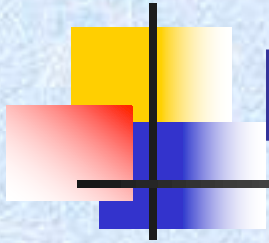
# Exemplo

- Mostre o que acontece em cada estágio da execução da seguinte fração de código:

2004	10
2000	5
	...
1008	STORE R2, 4(R3)
1004	ADDI R2, R1, 3
1000	LOAD R1, 0(R3)
	...

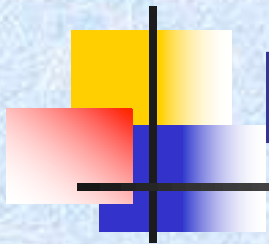
R1	R2	R3	PC	IR
20	10	2000	1000	

# Execução sem Pipeline x Execução com Pipeline



	1	2	3	4	5	6	7	8	9	10
Instrução 1	IF	ID	EX	Mem	<b>WB</b>					
Instrução 2						IF	ID	EX	Mem	<b>WB</b>

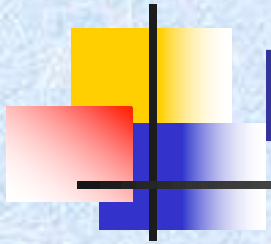
	1	2	3	4	5	6	7	8	9	10
Instrução 1	IF	ID	EX	Mem	<b>WB</b>					
Instrução 2		IF	ID	EX	Mem	<b>WB</b>				
Instrução 3			IF	ID	EX	Mem	<b>WB</b>			
Instrução 4				IF	ID	EX	Mem	<b>WB</b>		
Instrução 5					IF	ID	EX	Mem	<b>WB</b>	
Instrução 6						IF	ID	EX	Mem	<b>WB</b>



# Pipelining

---

- Todo estágio do pipeline deve se completar em um ciclo de clock.
- Qualquer combinação de estágios deve poder ocorrer ao mesmo tempo.
- Desbalanceamento entre os estágios do pipeline: o estágio mais lento do pipeline limita o seu desempenho.
- Estágios EX que incluem operações de ponto flutuante não se completam em um ciclo de clock

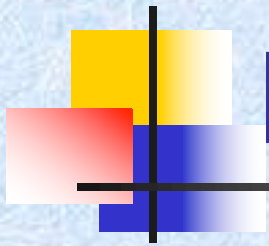


# Pipelining

---

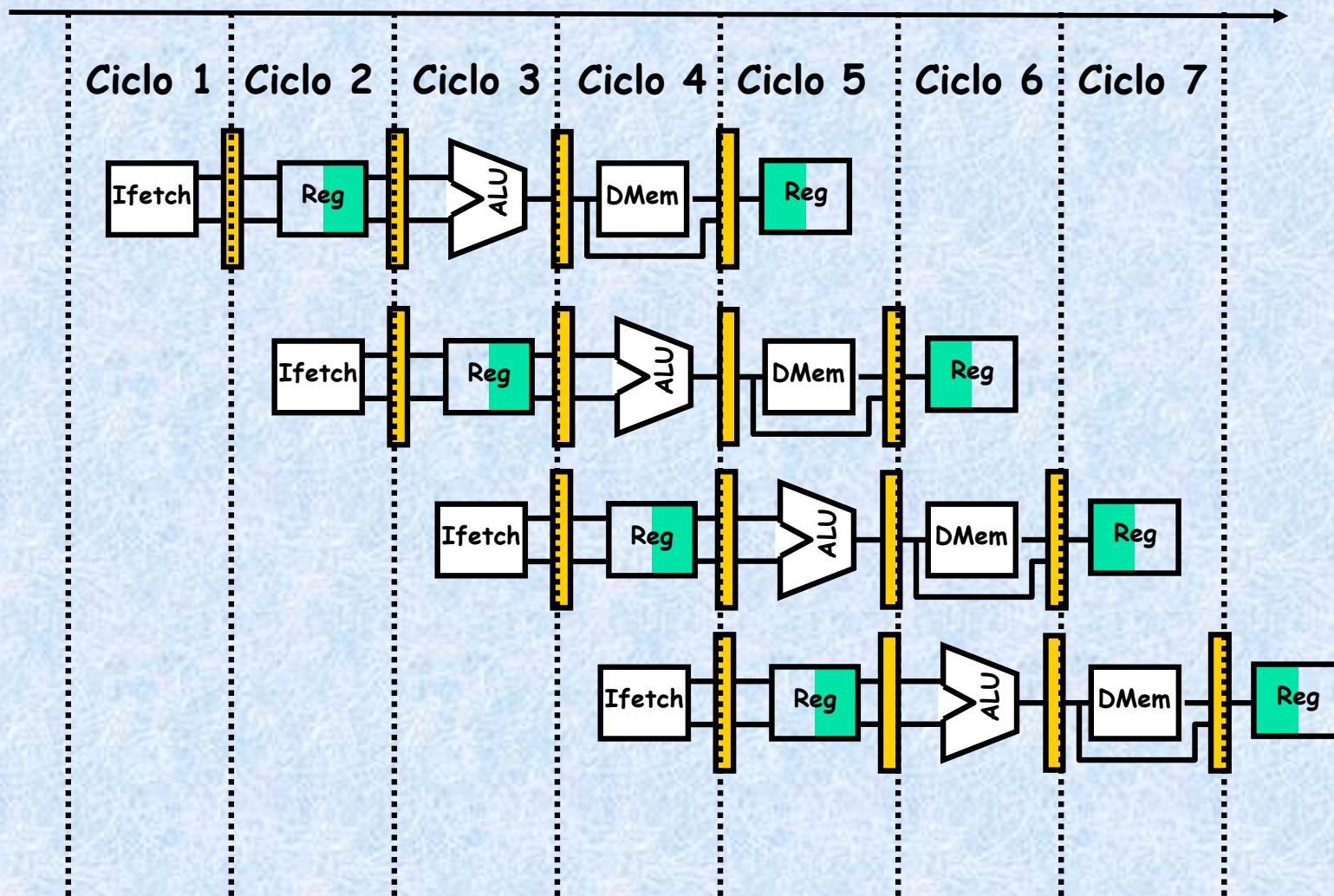
- Para a implementação do pipeline, overhead é adicionado por registradores de pipeline (latches)
- Os dados e controles necessários são passados de um estágio para outro através de latches.

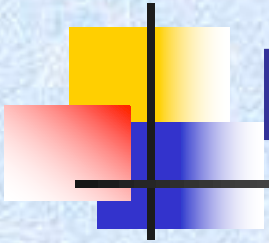




# Pipelining (Patterson)

*I  
n  
s  
t  
r.  
  
O  
r  
d  
e  
r*

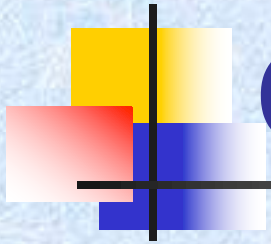




# Hazards

---

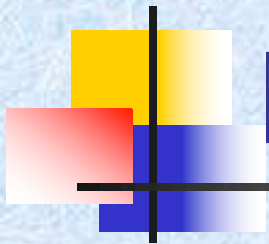
- Os hazards são a maior fonte de limitação do pipeline
- São situações que impedem que o próxima estágio da instrução seja executada no próximo ciclo de clock.
- Os hazards fazem com que o pipeline seja atrasado (stall).



# Classes de Hazards

---

- Estrutural: o hardware não é capaz de executar este passo de maneira sobreposta com os outros passos que já estão no pipeline
- Dados: uma instrução depende do resultado de uma instrução anterior, que ainda não foi produzido
- Controle: causados por desvios

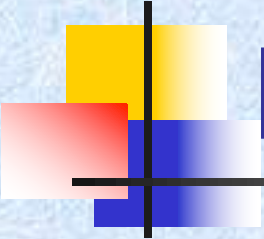


# Hazards Estruturais

---

- A execução sobreposta de instruções exige que as unidades funcionais sejam pipelined e que haja duplicação de recursos de modo a permitir qualquer combinação de instruções no pipeline.
- Quando isso não acontece, ocorre um hazard estrutural e o pipeline é atrasado.
- Este atraso é feito pela inserção de ciclos de stall (pipeline bubbles)

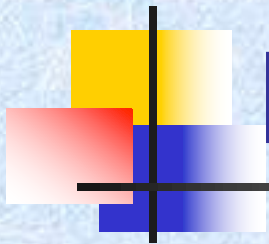




# Razões para Hazards Estruturais

---

- Algumas unidades funcionais não são totalmente pipelined
- Recursos não foram replicados de maneira suficiente.
  - Neste caso, ocorre conflito pela utilização do recurso
  - Exemplo de recursos: registradores, paths, portas para acesso à memória



# Exemplo de Hazard Estrutural

- Admita uma máquina com somente uma porta de memória e acessos à memória no estágio Mem, se a instrução for Load

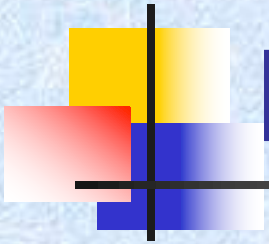
	1	2	3	4	5	6	7	8	9
Load	<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>				
Instrução 2		<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>			
Instrução 3			<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>		
Instrução 4				<b>stall</b>	<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>



# Por que permitir um hazard estrutural?

---

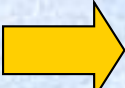
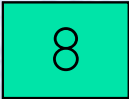

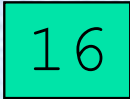
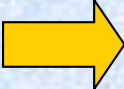
- Se a situação não ocorre com frequência, talvez o overhead e o custo introduzidos não sejam compensados
- Ex: MIPS R2010: escolha por menor latência na FPU ao invés do full pipelining



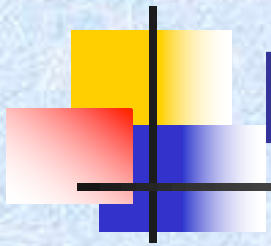
# Hazard de Dados

---

- Ocorre porque o pipelining pode modificar a ordem dos acessos aos operandos.
- Exemplo (resultado esperado):

		R1	R2	R3
	ADD R1, R2			
	MUL R3, R1			

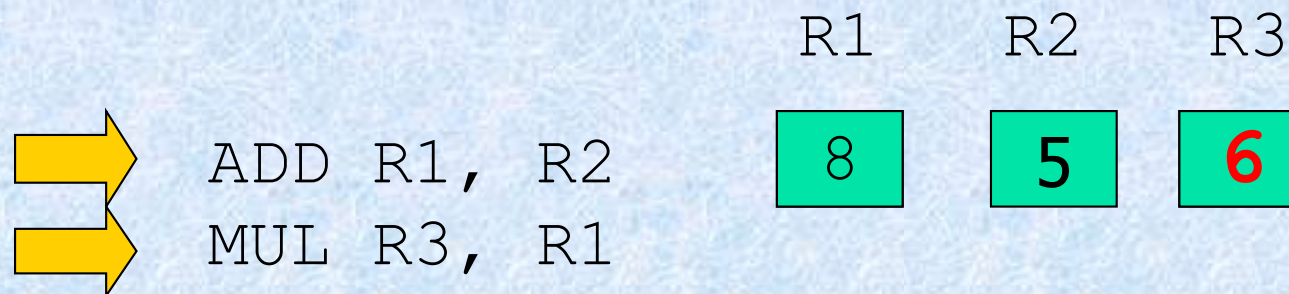


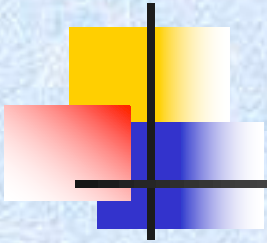


# Hazard de Dados

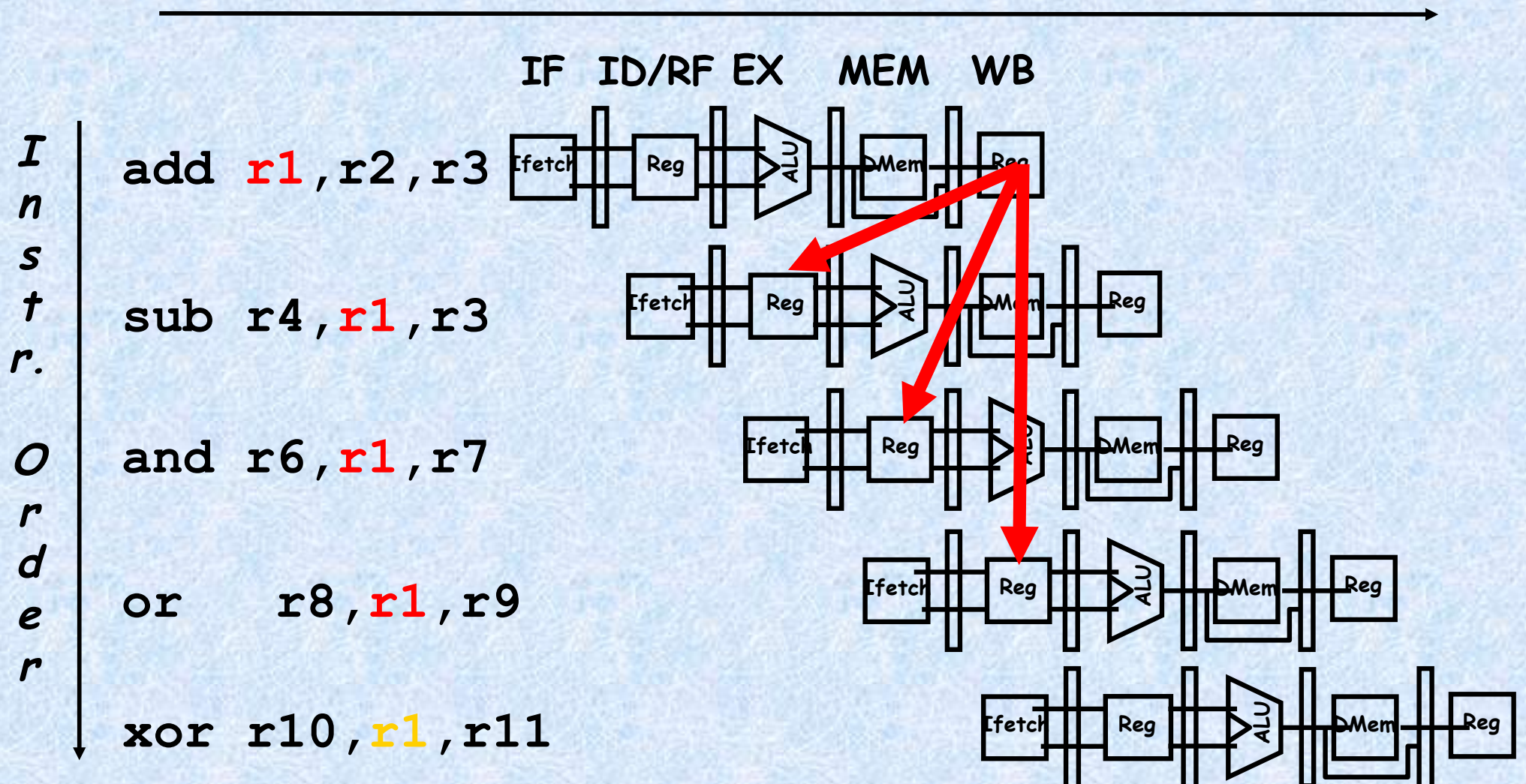
- Admita que o registrador só é escrito no estágio WB e é acessado no estágio ID

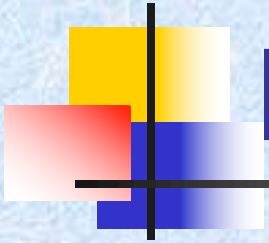
	1	2	3	4	5	6
ADD R1,R2	IF	ID	EX	Mem	<b>WB</b>	
MUL R3,R1		IF	<b>ID</b>	EX	Mem	WB





# Hazard de Dados (Patterson)

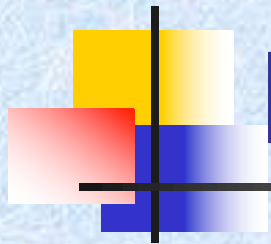




# Hazard de Dados

---

- Ocorrem quando há dependência entre as instruções.
- Tipos de dependências:
  - RAW
  - WAW
  - WAR



# Read After Write (RAW)

---

- Exemplo:

```
LOAD R1, # (10)
```

```
ADD R2, R1
```

```
STORE R2, # (14)
```

- Neste caso, a instrução LOAD escreve o conteúdo da posição de memória 10 no registrador R1.
- Na instrução ADD, o valor de R1 é lido, somado a R2 e colocado em R2 ( $R2 = R2 + R1$ ).
- Logo, há uma dependência RAW entre estas duas instruções, já que o ADD só pode começar quando o LOAD já houver carregado o valor em R1





# Write After Write (WAW)

---

- Exemplo:

```
LOAD R1, # (10)
```

```
ADD R1, R2
```

- Neste caso, a instrução LOAD escreve o conteúdo da posição de memória 10 no registrador R1.
- Na instrução ADD, o valor de R2 é lido, somado a R1 e colocado em R1 ( $R1 = R1 + R2$ ).
- Logo, há uma dependência WAW entre estas duas instruções, já que o valor final do registrador R1 deverá ser o produzido pela instrução ADD



# Write After Read (WAR)

---

- Exemplo:

```
ADD R1, R2
```

```
LOAD R2, # (10)
```

- Neste caso, a instrução ADD lê o registrador R2 e a instrução LOAD escreve o conteúdo da posição de memória 10 em R2.
- Logo, há uma dependência WAR entre estas duas instruções, já que o valor final do registrador R2 deverá ser o produzido pela instrução LOAD e o valor adicionado a R1 deve ser o valor anterior.

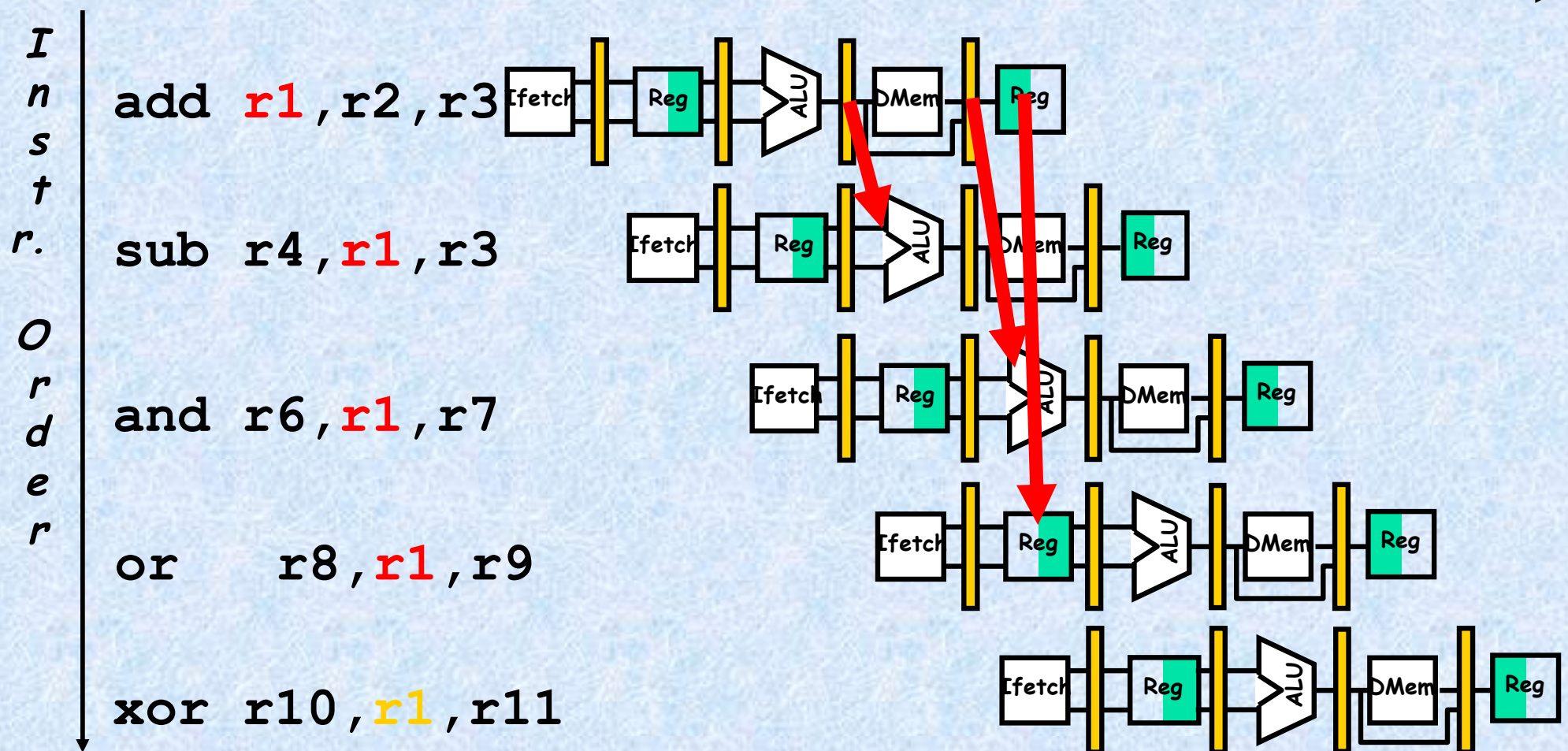


# Hazards de Dados - Forwarding

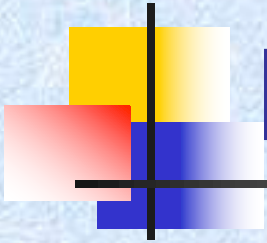
---

- O Forwarding (bypassing ou shortcutting) é uma técnica de hardware para solucionar alguns hazards de dados.
- Efetua a transferência direta para a unidade funcional que requer o dado.

# Hazards de Datos – Forwarding (Patterson)





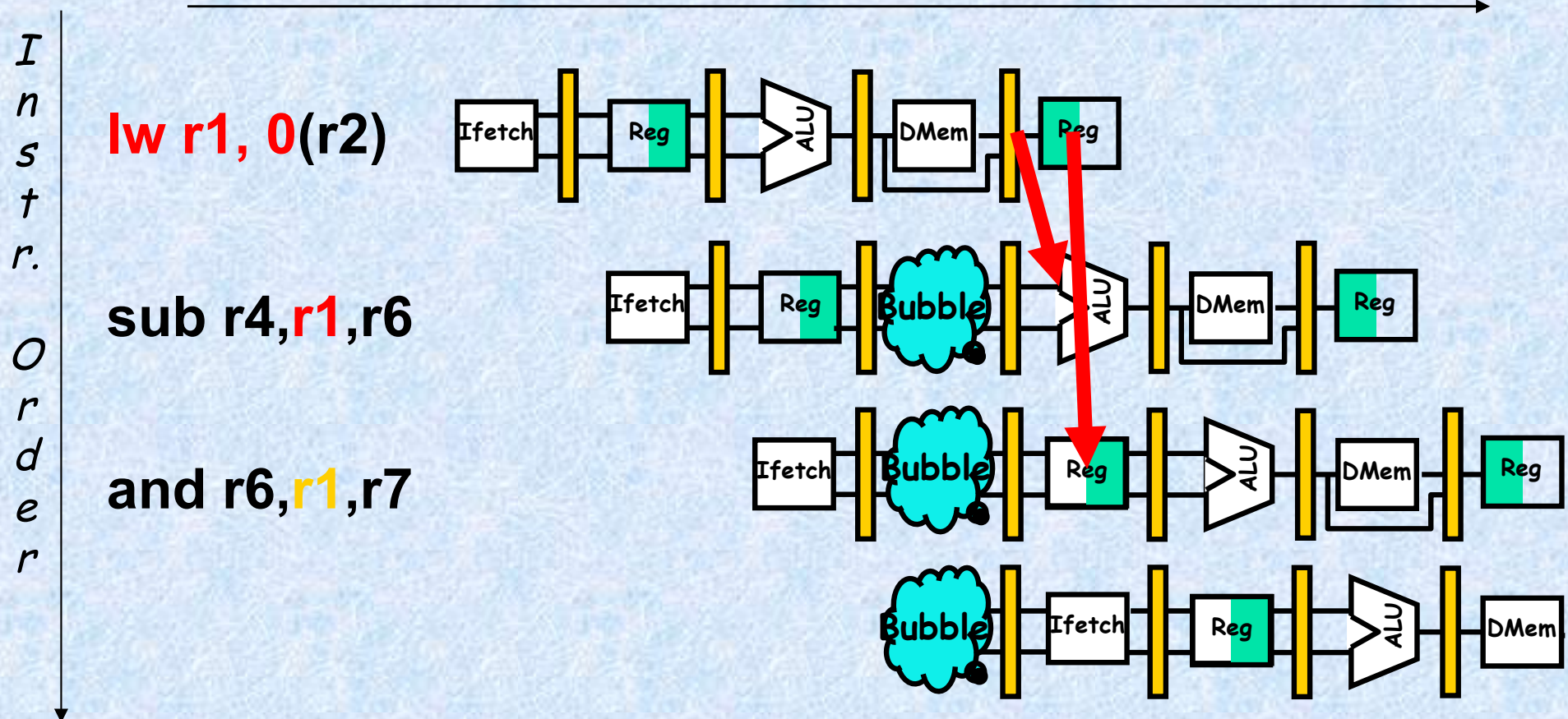


# Hazards de Dados com Stall

---

- Em alguns casos, o forwarding não é suficiente, pois o dado pode ser necessário em um estágio da instrução  $i+1$  quando ainda não foi produzido pela instrução  $i$ .
- Os loads são geralmente causadores deste tipo de hazard.
- Pipeline interlock: hardware que detecta o hazard e atrasa o pipeline, através da introdução de bubbles, até que a execução possa continuar.

# Hazards de Datos – Pipeline Interlock





# Harzards de Dados – Auxílio do Compilador

---

- Instruções simples e frequentes como “A=B+C” levam o pipeline a ser atrasado (stall).

```
LOAD R1, B
```

```
LOAD R2, C
```

```
ADD R3, R2, R1
```

```
STORE R3, A
```



# Harzards de Dados – Auxílio do Compilador

---

- O compilador pode, ao gerar código, tentar evitar certos padrões, e.g., evitar colocar um load em um registrador muito próximo da instrução que usa este registrador.
- Pipeline scheduling (instruction scheduling): técnica usada pelo compilador para reduzir o número de stalls.



# Harzards de Dados – Auxílio do Compilador

A=B+C  
D=E+F

- Código Básico

LOAD R1,B

LOAD R2,C

ADD R3,R1,R2



STORE A,R3

**LOAD R4,E**

**LOAD R5,F**

ADD R6,R4,R5



STORE D,R6

- Código Modificado

LOAD R1,B

LOAD R2,C

**LOAD R4,E**

ADD R3,R1,R2

**LOAD R5,F**

STORE A,R3

ADD R6,R4,R5

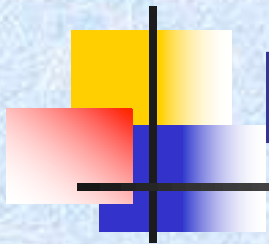
STORE D,R6



# Harzards de Dados – Blocos Básicos

---

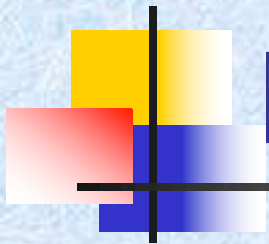
- Um bloco básico é uma sequência de instruções onde não há desvios e nem transferências de I/O.
- Em um bloco básico, todas as instruções são executadas, caso a primeira o seja.
- Uma técnica simples consiste em dividir o programa em blocos básicos e escalonar as instruções dentro de cada bloco



# Hazards de Controle

---

- Mudanças no fluxo de execução das instruções podem ocorrer por:
  - Branches: desvios condicionais (ifs e loops)
  - Jumps: desvios incondicionais (gotos)
  - Procedure calls
  - Procedure returns



# Hazards de Controle

---

- Quando uma instrução de desvio (branch) é executada, o PC pode assumir um valor que não seja o da próxima instrução.

- Exemplo:

10: LOAD RB, # (FF0)

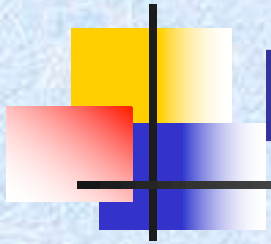
14: BZ 600 → branch condicional

18: LOAD R1, # (100)

...

600: ADD R1, 10 → alvo do branch

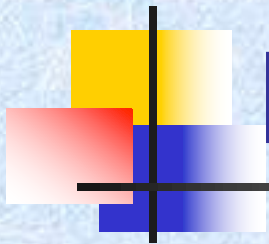




# Hazards de Controle: Desvios

---

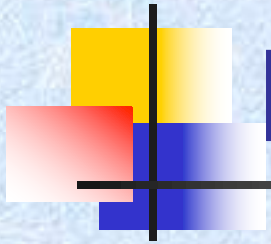
- Quando uma instrução de desvio muda o PC, diz-se que o desvio foi tomado (branch taken).
- Se a instrução  $i$  for um desvio tomado, o novo endereço do PC somente será conhecido no final do estágio Mem.



# Hazards de Controle: Desvios

	1	2	3	4	5	6	7	8	9
Branch	<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>				
Sucessor do Branch		<b>IF</b>	stall	stall	<b>IF</b>	ID	EX	Mem	WB
Sucessor do Branch+1						<b>IF</b>	ID	EX	<b>Mem</b>
Sucessor do Branch+2							<b>IF</b>	ID	EX

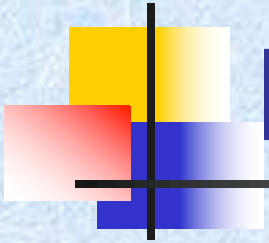
- Neste caso, atrasar o pipeline de 3 ciclos (dois stalls e um IF repetido) é uma perda muito grande em desempenho.



# Hazards de Controle: Desvios

---

- O número de ciclos de clock atrasados por causa de um branch pode ser reduzido com duas técnicas combinadas:
  - Descobrir se o branch é tomado ou não mais cedo
  - Calcular o novo PC antes



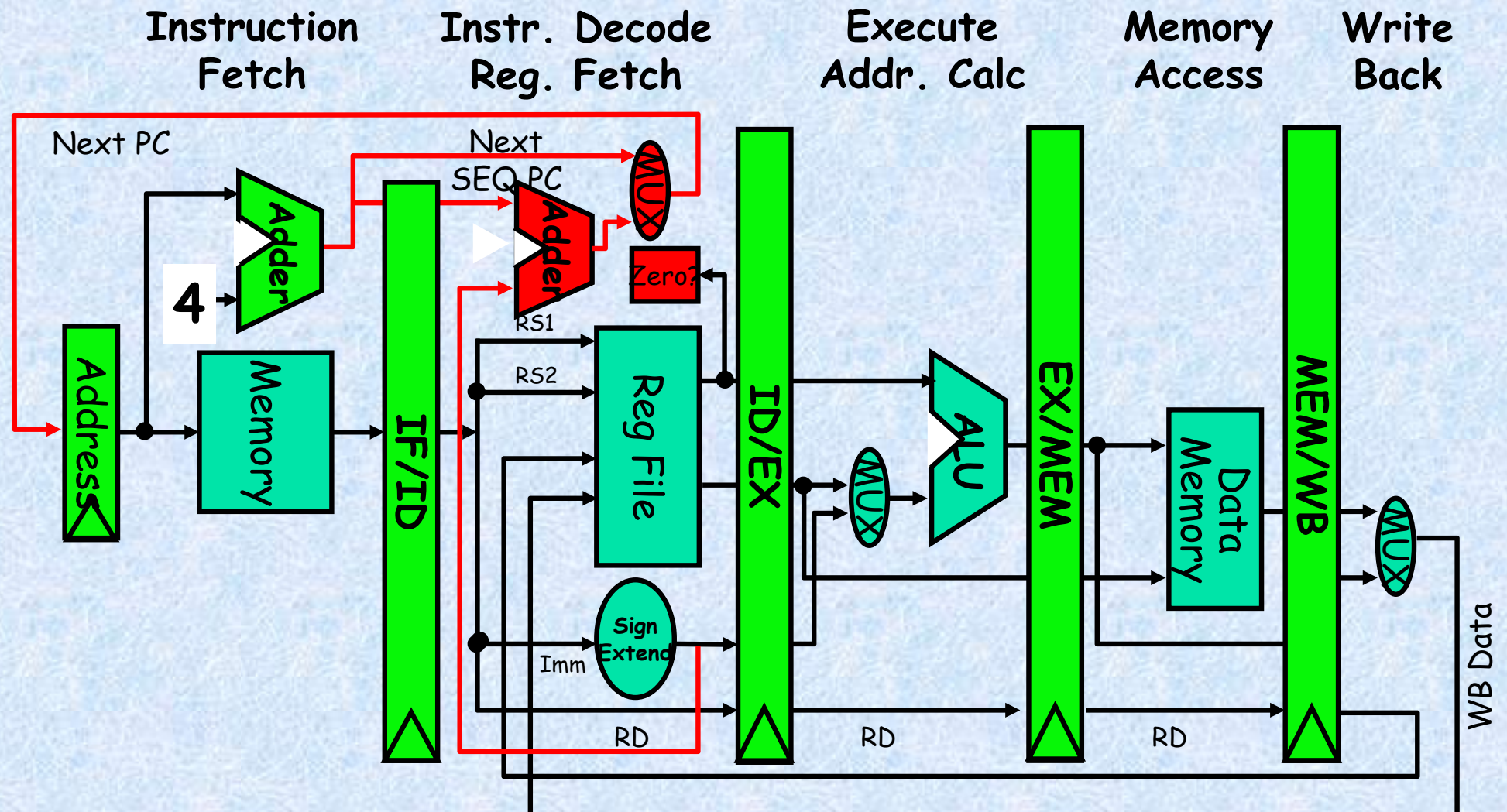
# Hazards de Controle: Desvios

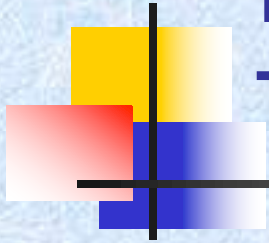
---

- Geralmente, os branches são decididos através de uma comparação de um registrador com o valor zero.
- Pode-se, então, mover este teste para o estágio ID através da inclusão neste estágio de um adicionador e do teste de branch.
- Neste caso, temos somente um ciclo de atraso (stall).



# Inclusão do Adicionador e do teste no estágio ID-Patterson





# Redução dos atrasos nos desvios:

## Técnica 1 – Previsão “not taken”

---

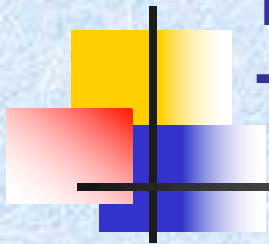
- Todos os branches são tratados como “not taken” e o pipeline é carregado com as instruções que imediatamente seguem o branch.
- O estado da máquina não pode ser alterado até que o resultado do desvio seja conhecido e, caso o desvio seja tomado, as eventuais mudanças devem ser desfeitas (back out).

# Redução dos atrasos nos desvios:

## Técnica 1 – Previsão “not taken”

	1	2	3	4	5	6	7	8
Branch not taken	<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>			
Instrução i+1		<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>		
Instrução i+2			<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>	
Instrução i+3				<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>

	1	2	3	4	5	6	7	8
Branch taken	<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>			
Instrução i+1		<b>IF</b>	idle	idle	idle	idle		
Branch target			<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>	
Branch target+1				<b>IF</b>	ID	EX	<b>Mem</b>	<b>WB</b>

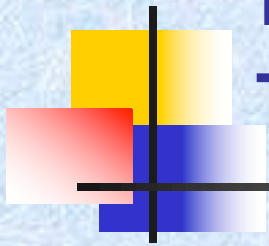


## Redução dos atrasos nos desvios: Técnica 2 – Previsão “taken”

---

- Logo que a instrução de branch for decodificada e o alvo do branch for conhecido, começa-se a buscar instruções a partir do alvo.
- Em muitas arquiteturas, não se conhece o alvo do branch antes de se executar o branch. Para estas arquiteturas, este esquema não é efetivo.





# Redução dos atrasos nos desvios: Técnica 3 – Delayed Branch

---

- Um ciclo de execução com branch delay  $n$  é definido como:

instrução\_do\_branch

sucessor\_1

sucessor\_2

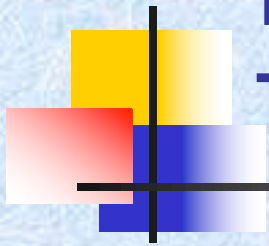
....

sucessor\_n

alvo\_do\_branch



Branch delay slots



# Redução dos atrasos nos desvios:

## Técnica 3 – Delayed Branch

---

- As instruções contidas no branch delay slot são executadas independente do resultado do branch (taken ou not taken).
- A grande maioria das máquinas que usam esta técnica possuem delay slot de tamanho 1.
- O compilador tem um papel fundamental nesta técnica, pois faz o escalonamento de instruções



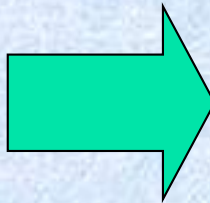
## Técnica 3 – Delayed Branch Escalonamento “from before”

---

ADD R1,R2,R3

If R2=0 then  
[delay slot]

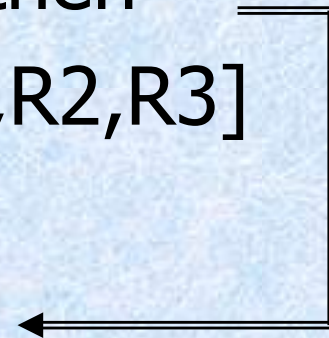
...



If R2=0 then

[ADD R1,R2,R3]

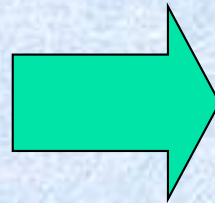
...



- O branch não deve depender da instrução escalonada
- Sempre melhora o desempenho
- Deve ser utilizada sempre que possível

## Técnica 3 – Delayed Branch Escalonamento “from target”

SUB R4,R5,R6  
...  
ADD R1,R2,R3  
If R1=0 then  
[delay slot]



SUB R4,R5,R6  
...  
ADD R1,R2,R3  
If R1=0 then  
[SUB R4,R5,R6]

- Usado quando a probabilidade de “taken” é grande
- No caso de “not taken”, trabalho é desperdiçado porém a execução deve continuar correta





## Técnica 3 – Delayed Branch Escalonamento “from fall through”

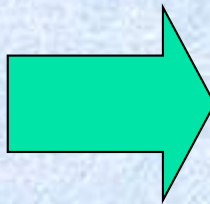
---

ADD R1,R2,R3

If R1=0 then  
[delay slot]

...

SUB R4,R5,R6



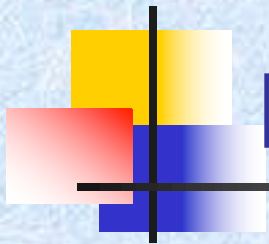
ADD R1,R2,R3

If R1=0 then  
[SUB R4,R5,R6]

...



- Usado quando a probabilidade de “not taken” é grande
- No caso de “taken”, a execução deve continuar correta



## Técnica 3 – Delayed Branch

### Limitações

---

- Muitas vezes, esta técnica não pode ser aplicada devido a:
  - Restrições das próprias instruções que deverão ser escalonadas no delay slots (e.g., dependências, branches)
  - Incapacidade de prever, em tempo de compilação, se um desvio será executado ou não.



## Técnica 3 – Delayed Branch Cancelling Branches

---

- Um cancelling branch contém a previsão daquele desvio.
- Se a previsão for correta, o delay slot é executado.
- Caso a previsão seja incorreta, o delay slot é transformado em no-op.
- Elimina restrições sobre a instrução a ser colocada no delay slot.

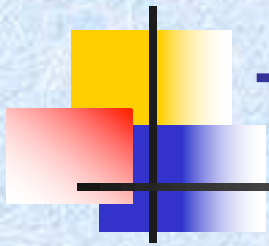


## Técnica 3 – Delayed Branch Cancelling Branches

---

- A maior parte das máquinas que oferece cancelling branches, oferece também a forma tradicional de branch.
- Os cancelling branches são normalmente cancelados se “not taken”.





## Técnica 3 – Delayed Branch Tendências

---

- A técnica de delayed branch foi muito usada nas primeiras máquinas RISC.
- Atualmente, hardware mais poderoso para previsão de desvios é usado, o que torna esta técnica desnecessária.



# Pipelining – Medidas Básicas de Desempenho

---

- Tempo Médio de Execução de Instrução
  - $AIT = \text{ciclo de clock} * CPI \text{ média}$
- Speedup do pipeline
  - $S = (AIT \text{ sem pipeline}) / (AIT \text{ com pipeline})$
- Tempo total de execução de n instruções com um pipeline de k estágios
  - $T_k = k + [(n-1) * \text{máximo atraso do estágio}]$

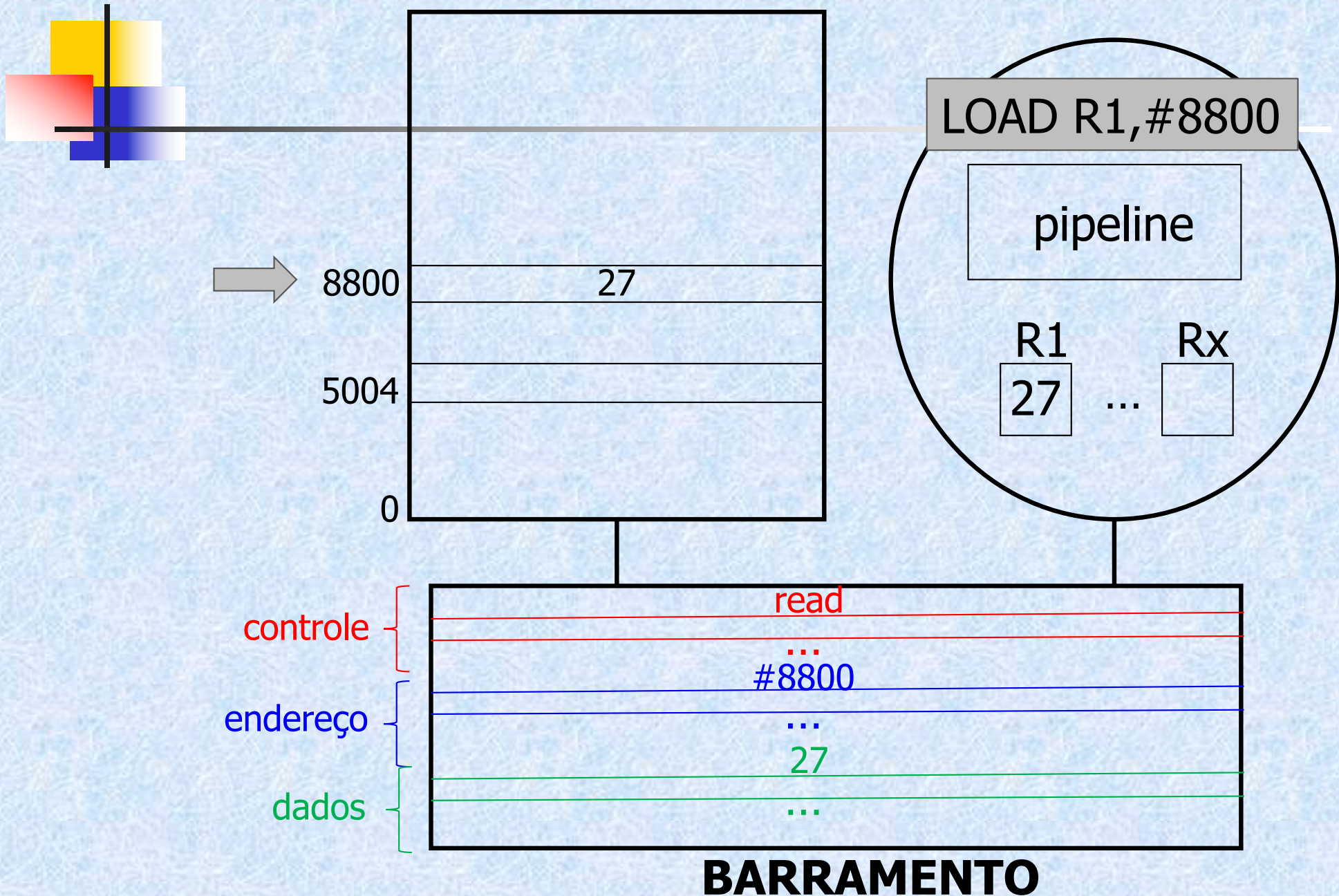


# Barramentos

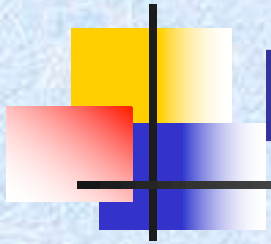
---

# MEMORIA

# CPU



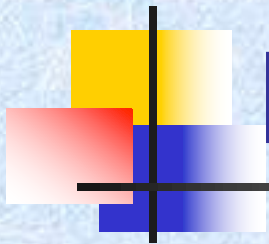




# Barramento

---

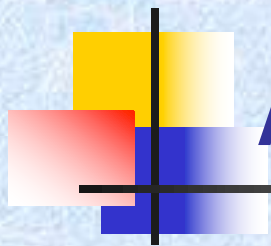
- É um meio de transmissão compartilhado que conecta dispositivos.
- Possui várias linhas:
  - Linhas de dados: cada linha transporta 1 bit por vez
  - Linhas de endereço: nelas é colocado o endereço da palavra a ser acessada
  - Linhas de controle: controlam o uso das linhas



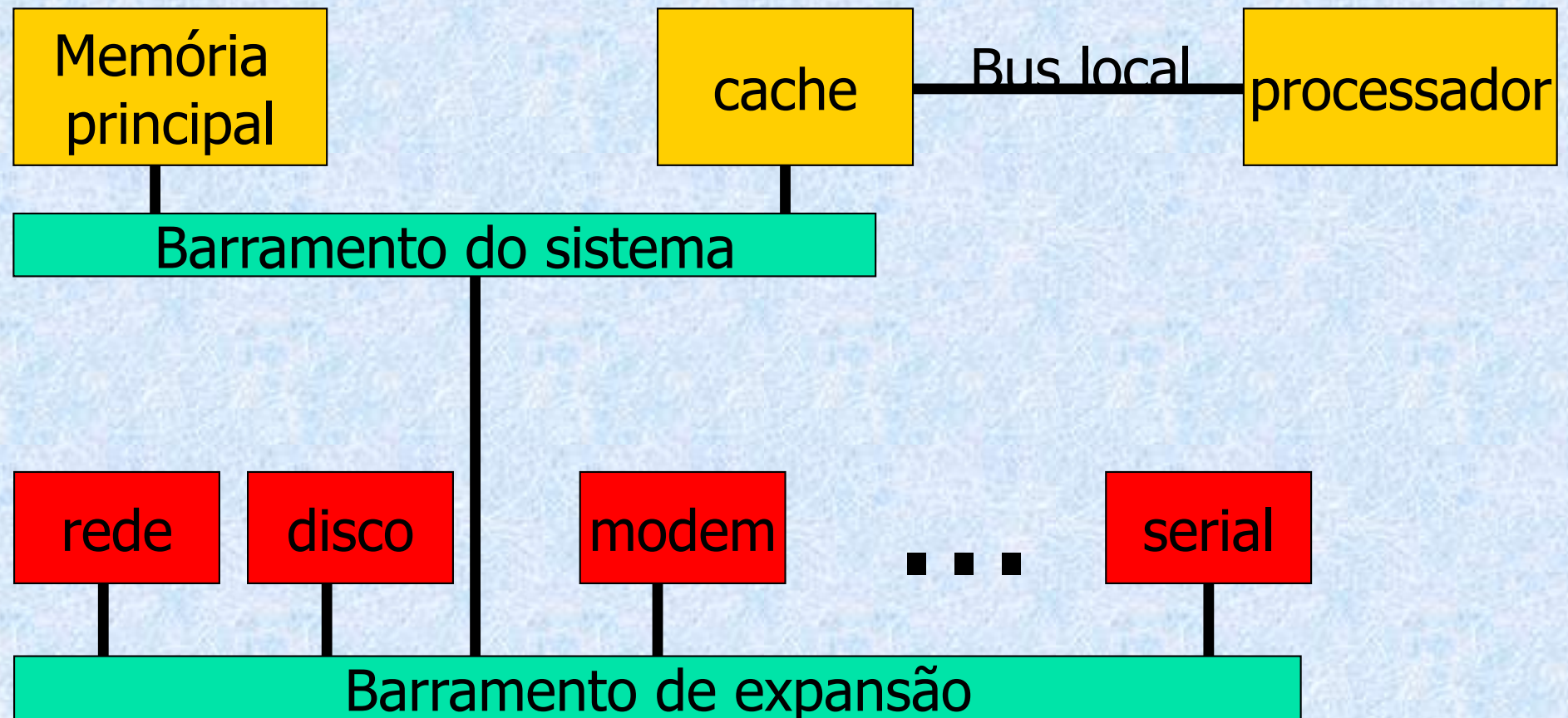
# Barramento

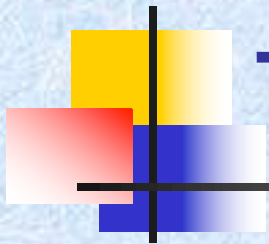
---

- Algumas linhas de controle
  - Memory write
  - Memory read
  - Transfer ACK
  - Bus request: pede o controle do barramento
  - Bus grant: obtém o controle do barramento
  - Clock
  - Reset



# Arquiteturas de Barramento



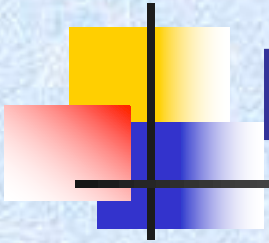


# Tipos de Barramento

---

- A velocidade máxima do barramento é limitada pelo:
  - Tamanho do barramento e
  - Número de dispositivos conectados
- Barramentos CPU-Memória
  - São pequenos e de alta velocidade
- Barramentos de I/O
  - São maiores e devem acomodar diversos tipos de dispositivos, seguindo um padrão

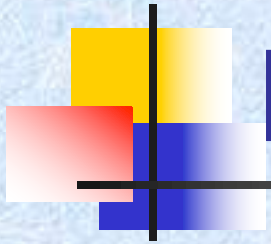




# Leitura de Dados da Memória

---

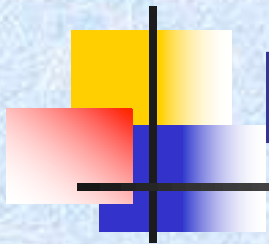
- O endereço e os sinais de controle indicando uma leitura são enviados no barramento (read e wait)
- A memória coloca o dado no barramento e retira (deasserts) o sinal de wait



# Escrita de Dados em Memória

---

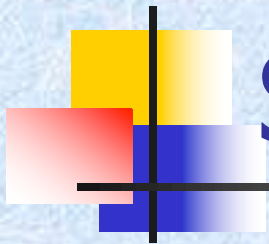
- A CPU coloca o endereço e o dado no barramento.
- A memória retira o dado e seu endereço do barramento e faz a atualização.
- Geralmente, a CPU não espera confirmação



# Bus Masters

---

- Bus masters: são os dispositivos que podem iniciar transações no barramento
- A CPU sempre é bus master.
- Sistemas com vários bus masters:
  - Várias CPUs, dispositivos de I/O específicos
  - Necessitam de um esquema de arbitragem para resolver conflitos, geralmente com prioridade fixa ou randômica

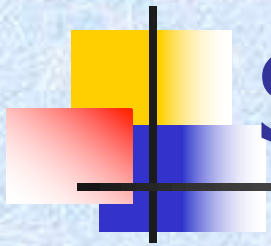


# Split Transaction Buses

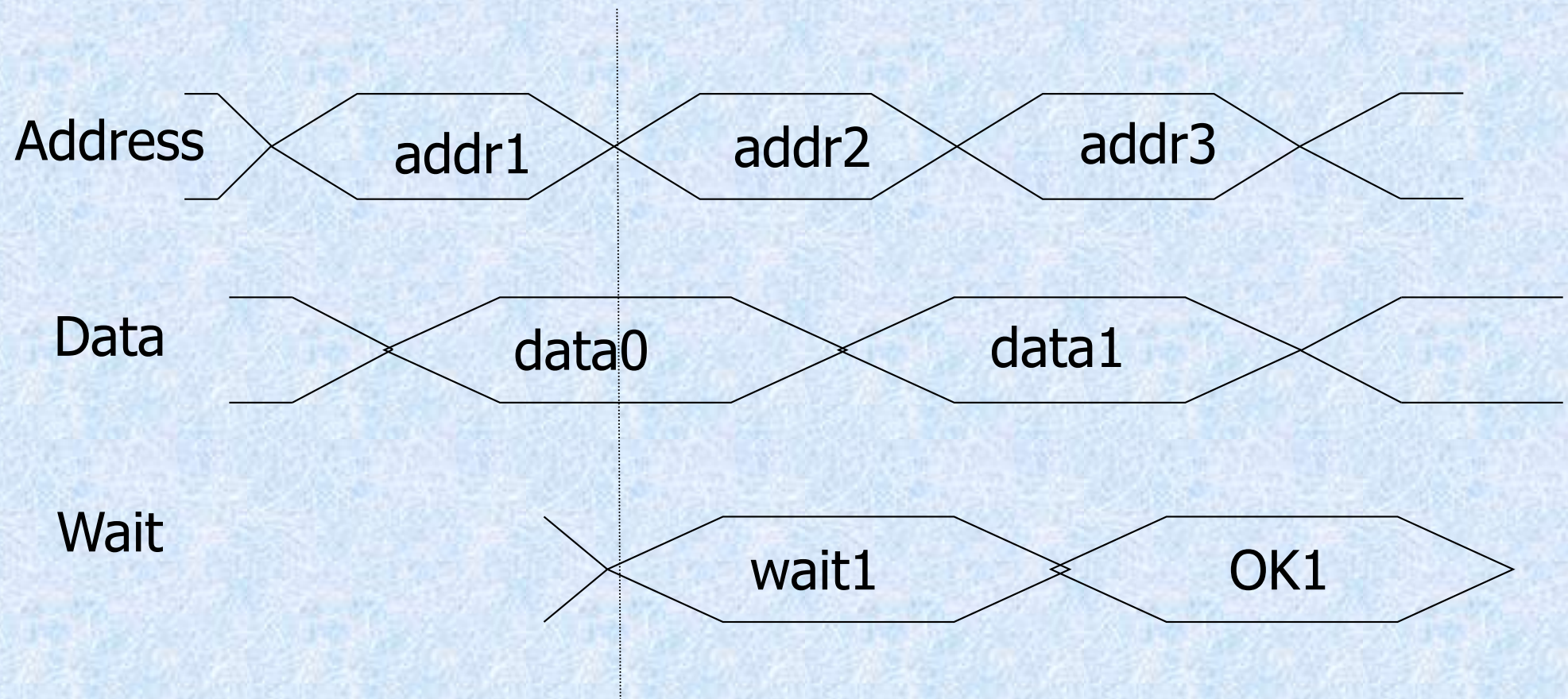
---

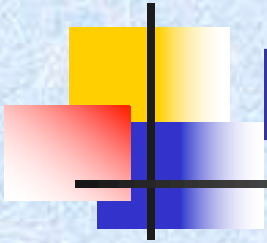
- Pipelined buses ou packet-switched buses
- As transações no barramento são divididas em etapas, não bloqueando o mesmo durante toda a transação.
- Exemplo: uma transação de read pode ser dividida em:
  - Read-request
  - Memory-reply
- A memória necessita participar na arbitragem
- As transações possuem tags, de modo a serem identificadas





# Split Transaction Buses

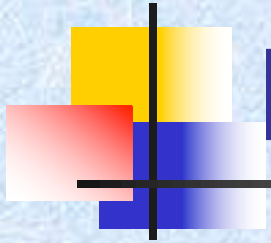




# Barramentos Síncronos

---

- Neste caso, uma das linhas de controle é um clock.
- Os protocolos para endereço e dados são fixos e baseados neste clock.
- São rápidos e baratos porém, devido a distorções do clock (clock skew), não podem ser longos.
- Os barramentos CPU-Memória são geralmente síncronos



# Barramentos Assíncronos

---

- Usam protocolos de handshaking entre o emissor e o receptor.
- Adicionam um overhead para sincronizar cada transação sendo, por isso, mais lentos.
- Permitem que mais tipos de dispositivos sejam conectados.
- São geralmente usados em I/O buses.