Aula 5

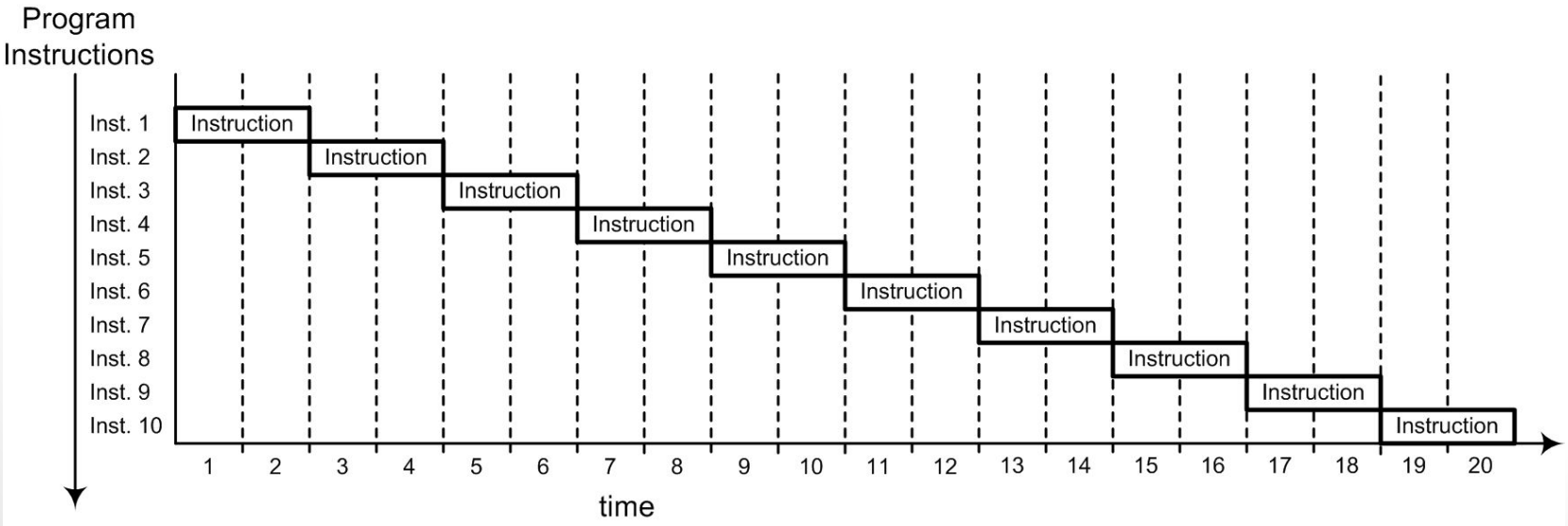
Organização de Computadores

Implementação do Pipeline

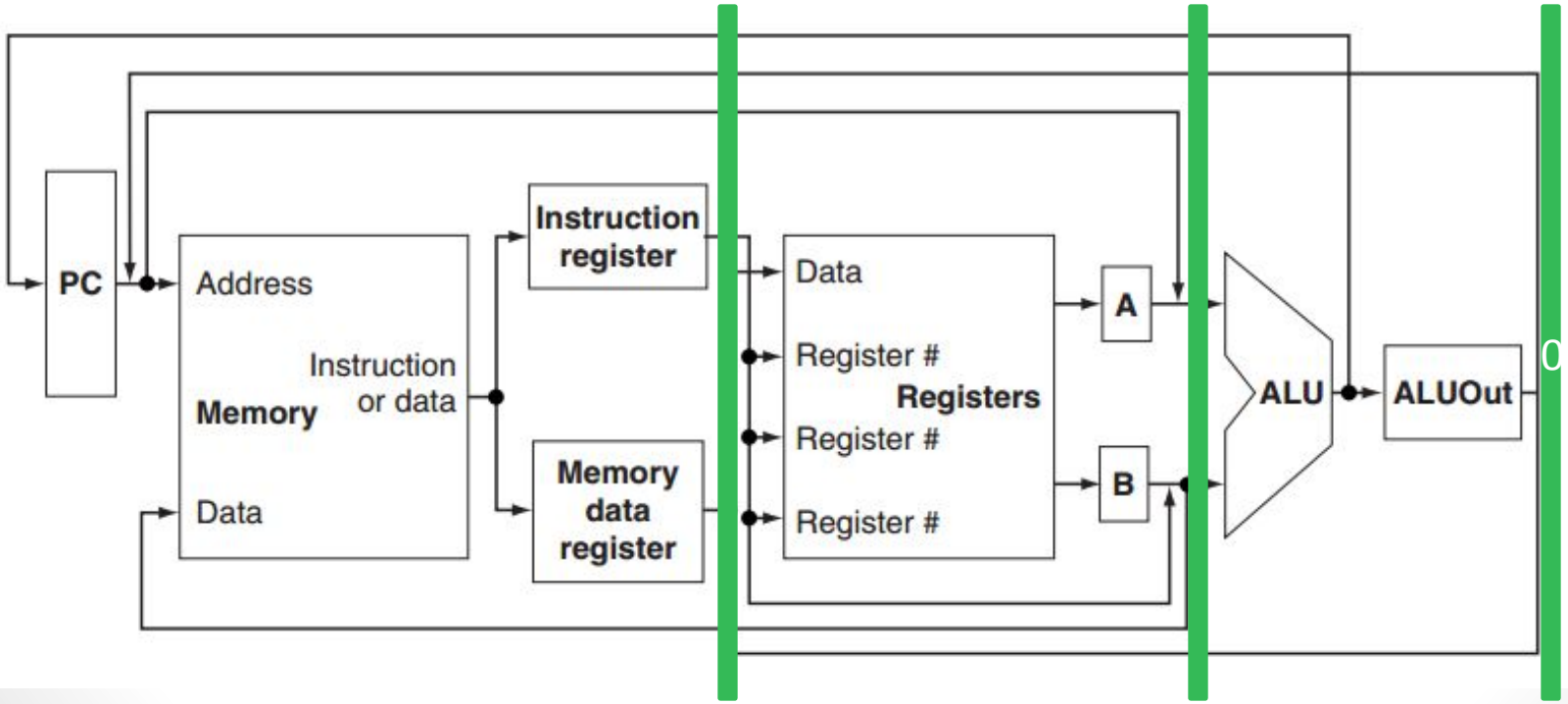
Profa. Débora Matos

Arquiteturas sem pipeline

Até aqui, todas as arquiteturas executam a próxima instrução somente após a conclusão da instrução anterior.



Como funciona o pipeline mesmo??



Visão geral do Pipeline

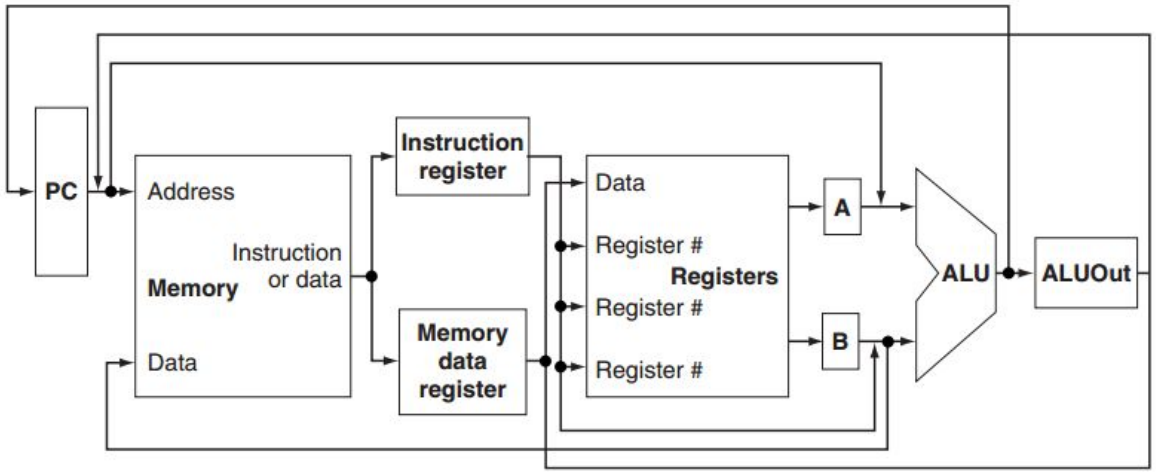
Pipeline é uma técnica de implementação em que várias instruções podem ser executadas ocupando estágios diferentes da organização.

É uma técnica que explora o paralelismo entre as instruções em fluxo de instruções sequenciais.

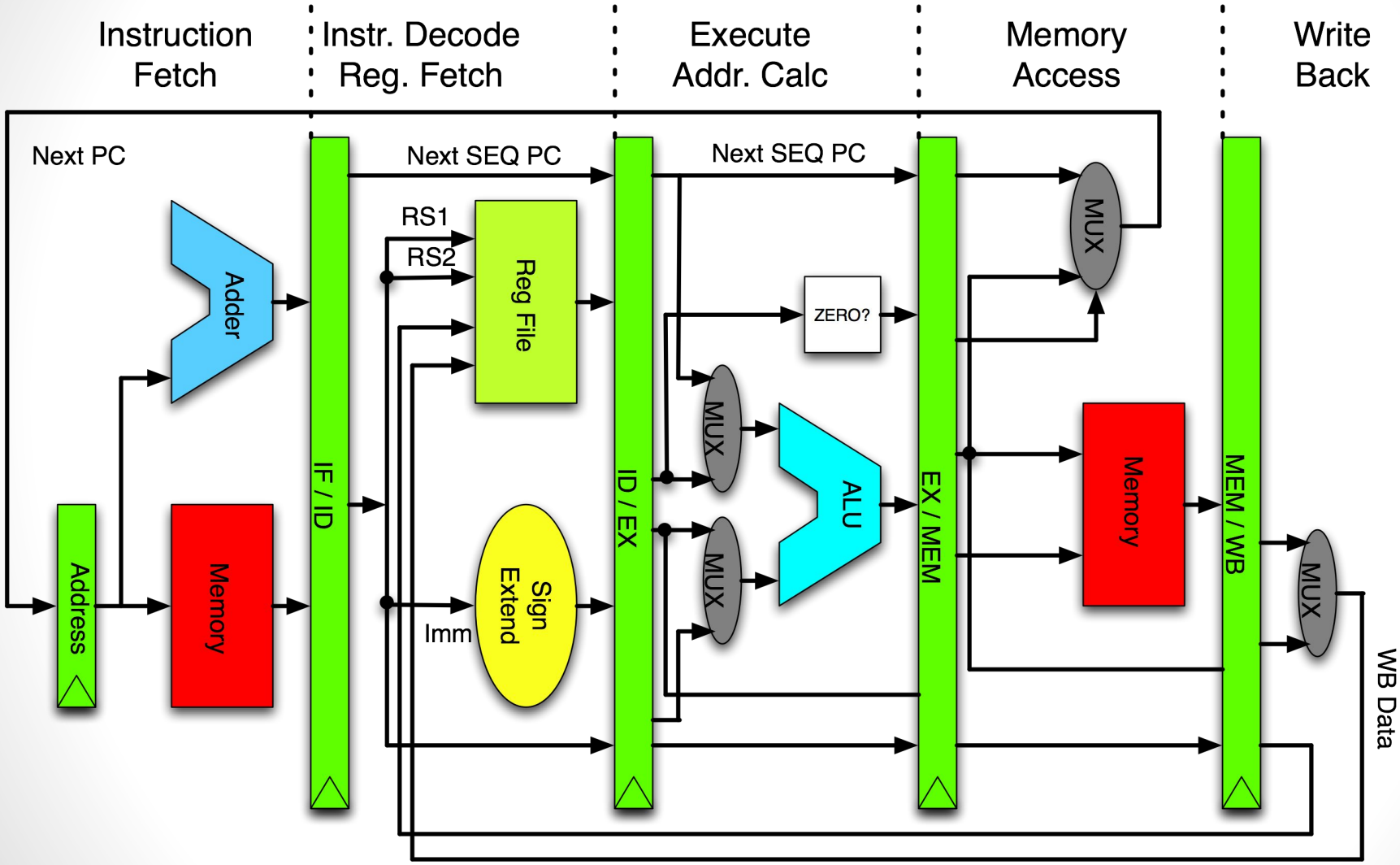
Pipeline

As instruções MIPS normalmente apresentam 5 etapas:

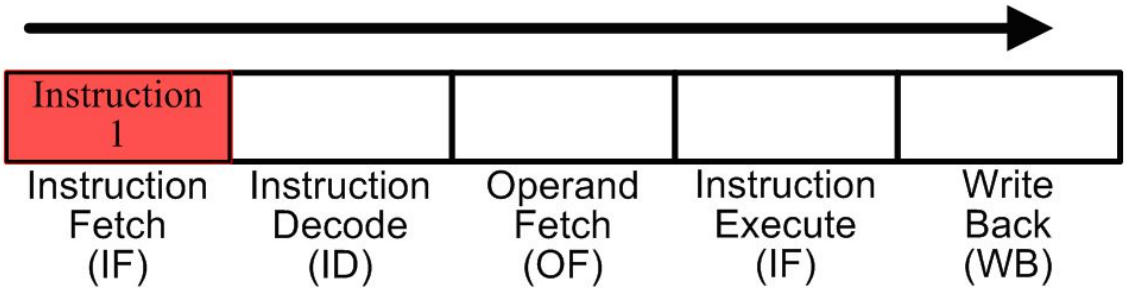
1. Buscar instrução na memória;
2. Ler registradores enquanto a instrução é decodificada;
3. Executar a operação ou calcular um endereço;
4. Acessar um operando na memória de dados;
5. Escrever o resultado em um registrador;



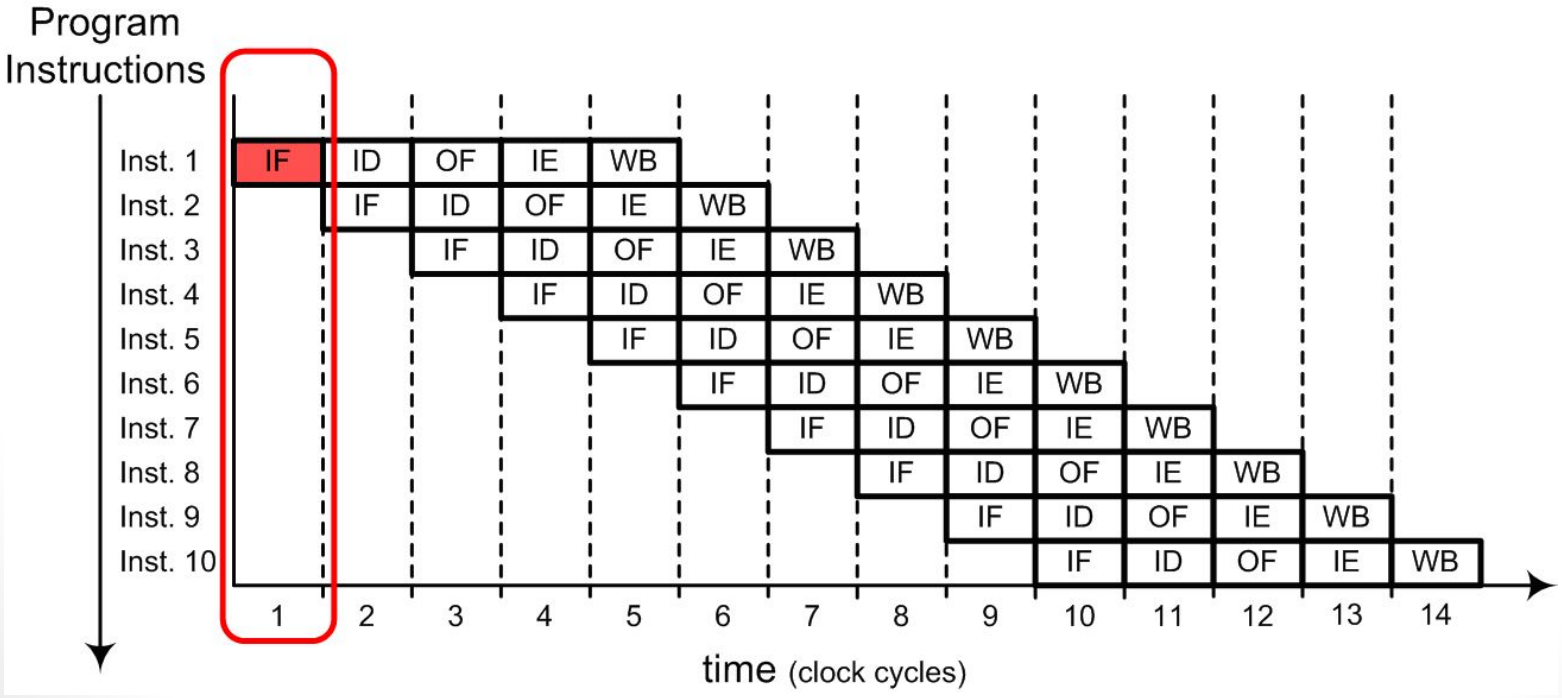
Pipeline


Instruction Fetch
Instr. Decode Reg. Fetch
Execute Addr. Calc
Memory Access
Write Back
Next PC
Next SEQ PC
Next SEQ PC
RS1
RS2
Reg File
Adder
IF / ID
ID / EX
EX / MEM
MEM / WB
MUX
ZERO?
MUX
ALU
Memory
MUX
Address
Memory
Sign Extend
Imm
WB Data

Pipeline



5 Stage Instruction Pipeline

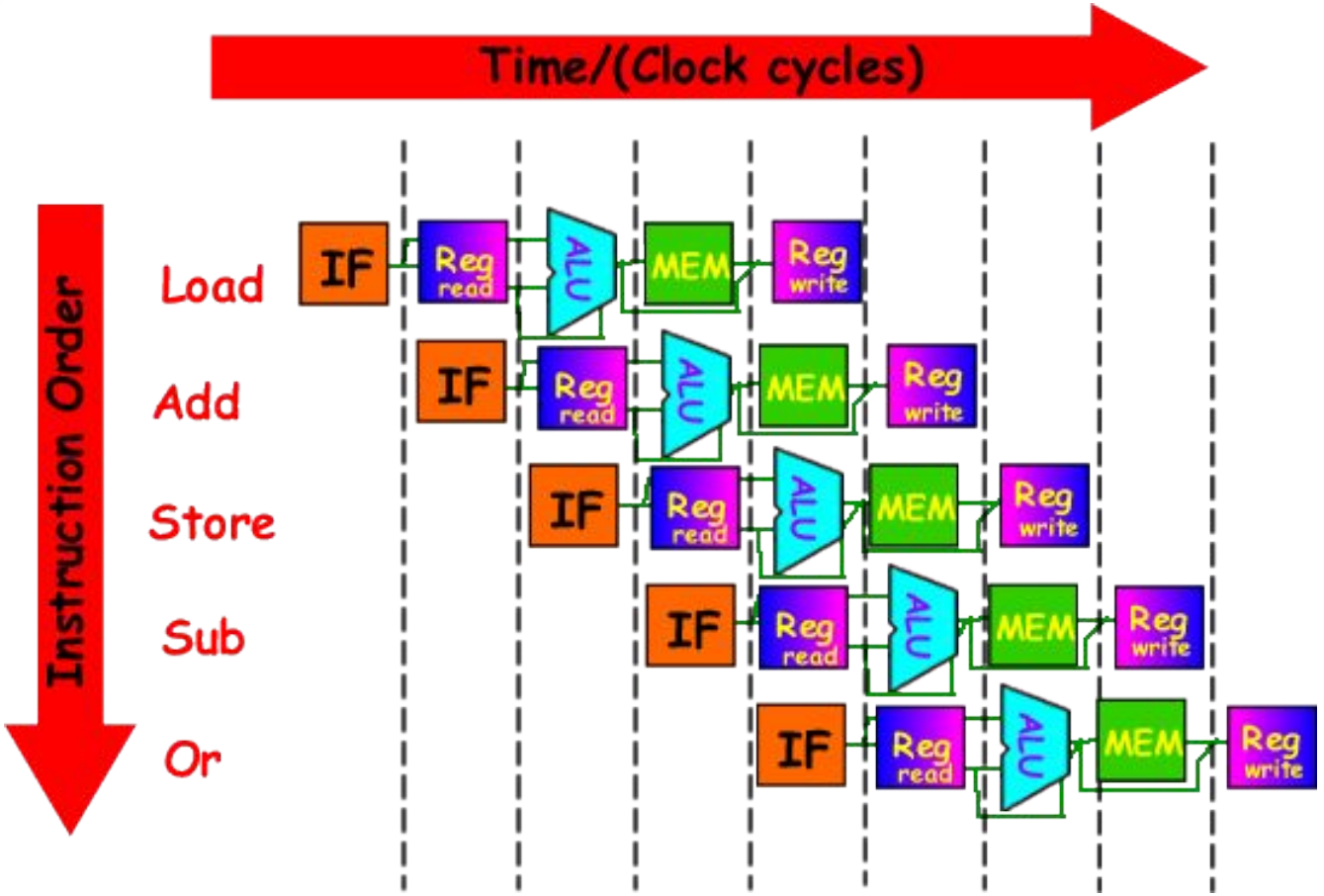


Pipeline

O pipeline utiliza técnicas de paralelismo.

1) **Sendo assim, as instruções são executadas mais rapidamente?**
2) **O número de ciclos de clock para executar cada instrução é o mesmo?**
3) **O que muda com relação a latência e a vazão na execução das instruções de um programa utilizando a técnica de pipeline?**

Pipeline



Se o tempo para os estágios são os seguintes:
100ps para leitura e escrita nos registradores
200ps para os demais estágios.
Qual é a frequência de operação?

Pipeline

Analisando o desempenho com pipeline.

Compare o tempo médio entre as instruções de uma implementação de ciclo único com uma implementação de pipeline. O tempo de operação de cada unidade funcional é:

- 200ps para operação na ALU
- 200ps para acesso à memória
- 100ps para leitura/escrita de registradores

Instruction	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps
1 ps = one picosecond = 1 x 10^{-12}s - see S.I. prefixes						

Pipeline

Analisando o desempenho com pipeline.

- No projeto de ciclo único, o ciclo de clock é definido pela instrução mais lenta.
- No projeto multiciclo e com pipeline, o ciclo de clock é definido pelo estágio mais lento.
- Na melhor hipótese, o ganho de velocidade com a técnica de pipeline é aproximadamente igual ao número de estágios de pipeline.

Instruction	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps
1 ps = one picosecond = 1 x 10^{-12}s - see S.I. prefixes						

Pipeline

Projetando um microprocessador para pipeline

- A implementação do pipeline é facilitada no MIPS já que todas as instruções apresentam o mesmo tamanho;
- O MIPS tem poucos formatos de instruções com campos semelhantes. Essa simetria facilita a implementação do pipeline;
- As instruções de acesso à memória utilizam o mesmo estágio de acesso a ULA para o cálculo do endereço.

Pipeline

Hazards (ou conflitos) de pipeline

- Ocorrem quando a próxima instrução não pode ser executada no ciclo de clock seguinte.

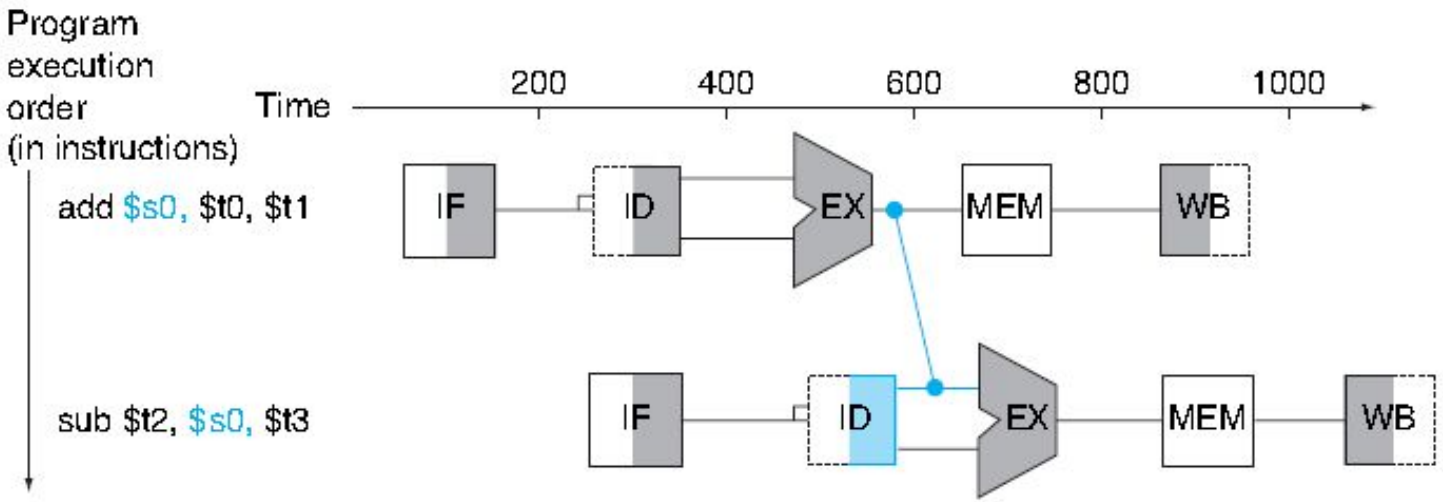
Exemplos?

- Hazards estruturais – conflito de acesso aos **mesmos recursos** (ex. memória – ler instrução e executar lw)
- Hazards de dados – quando ocorre **dependência** de dados. Exemplos?
- Hazards de controle – necessidade de tomar uma **decisão** com base em resultados anteriores.

Hazards de dados

Hazards (ou conflitos) de dados

- Como tais dependências acontecem com muita frequência, o ideal é não ter que esperar a instrução anterior ser executada.
- Para isso, utiliza-se técnicas de ***forwarding*** ou bypassing.
- O ***forwarding*** é empregado quando o estágio de dependência estiver mais adiante da obtenção do valor requerido.

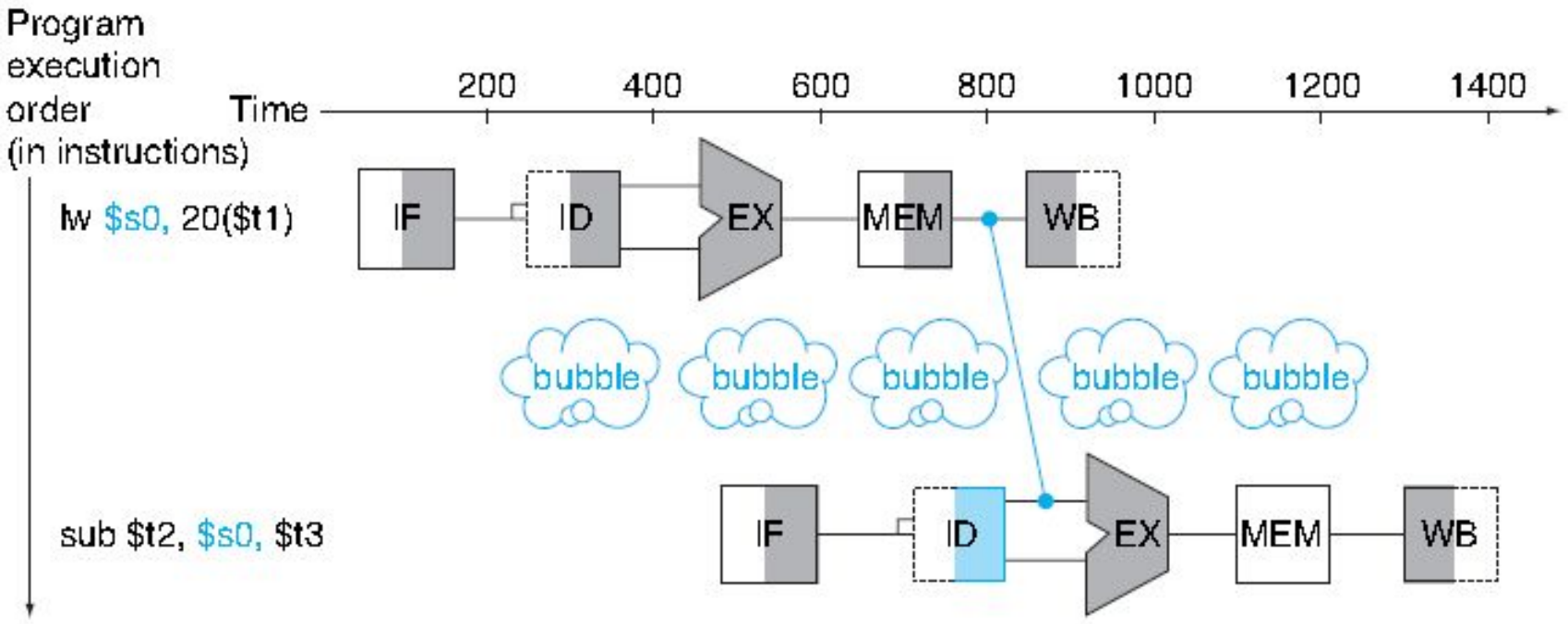


Hazards de dados

Hazards (ou conflitos) de dados

- Se o dado desejado não estiver disponível no estágio a seguir é necessário utilizar um *pipeline stall* (bolha).

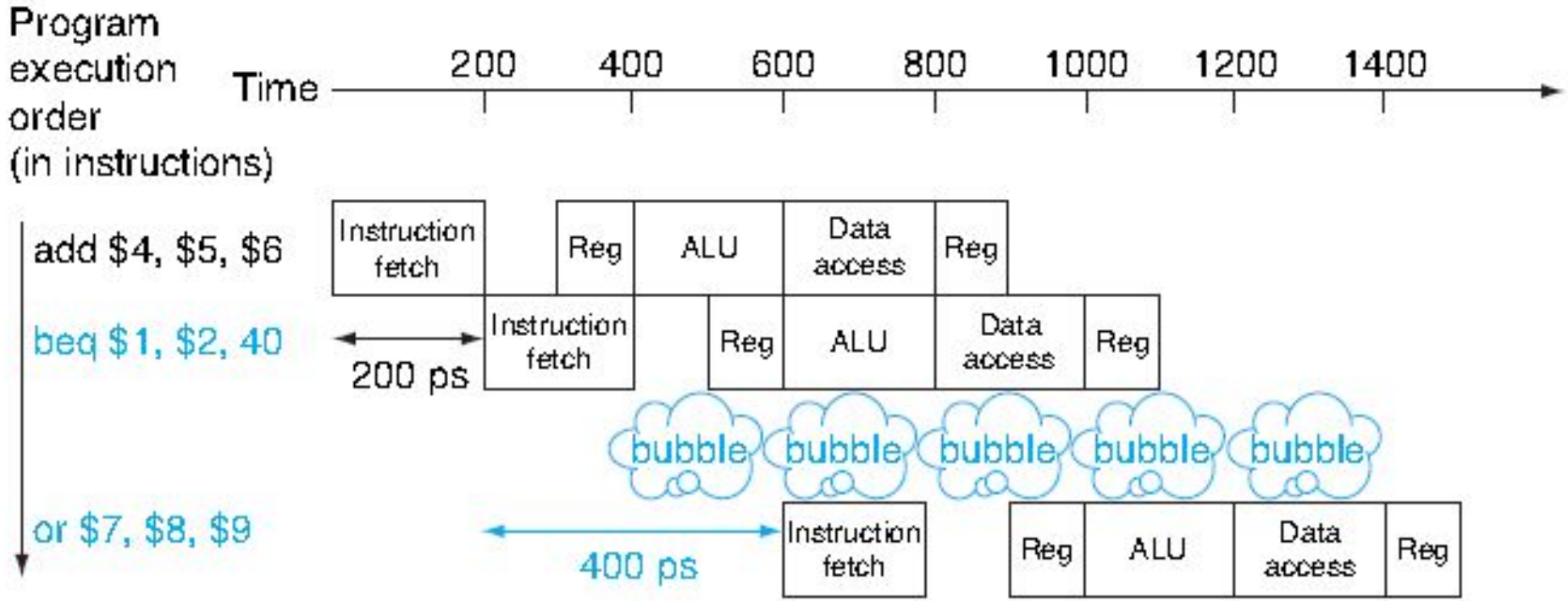
  Exemplo: instrução de *load* e após instrução aritmética.



Hazards de controle

Hazards de controle – ocorrem quando as instruções subsequentes dependem de uma decisão anterior.

Exemplo utilizando HW extra para testar registradores.

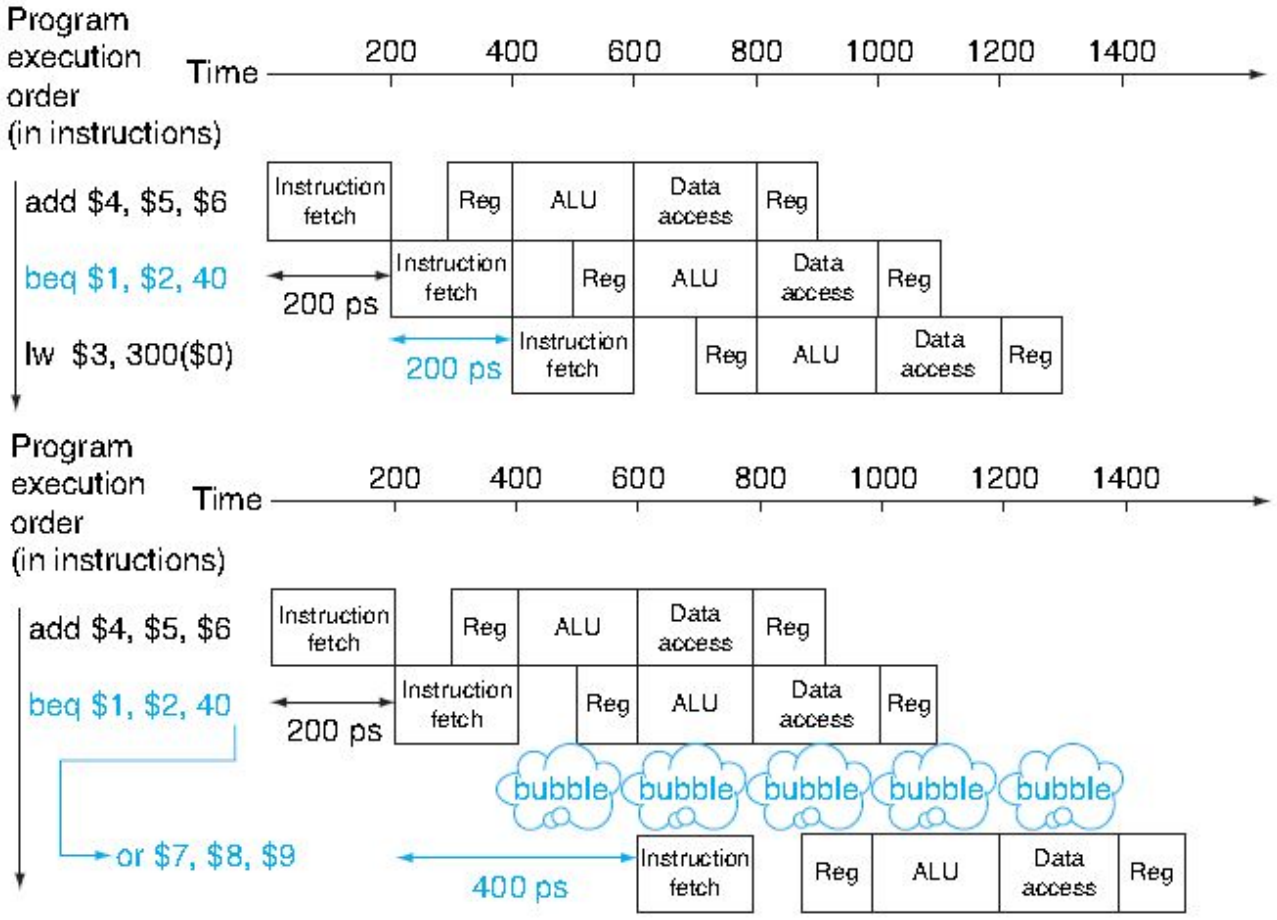


Hazards de controle

Hazards de controle – 2° solução: previsão de desvio.

Se a previsão for correta, não há atrasos no pipeline, porém se a decisão do desvio não ocorrer, então haverá o *stall.*

Uma possível solução é prever que os desvios **não** ocorrerão.



Hazards de controle

Hazards de controle – 2° solução: previsão de desvio.

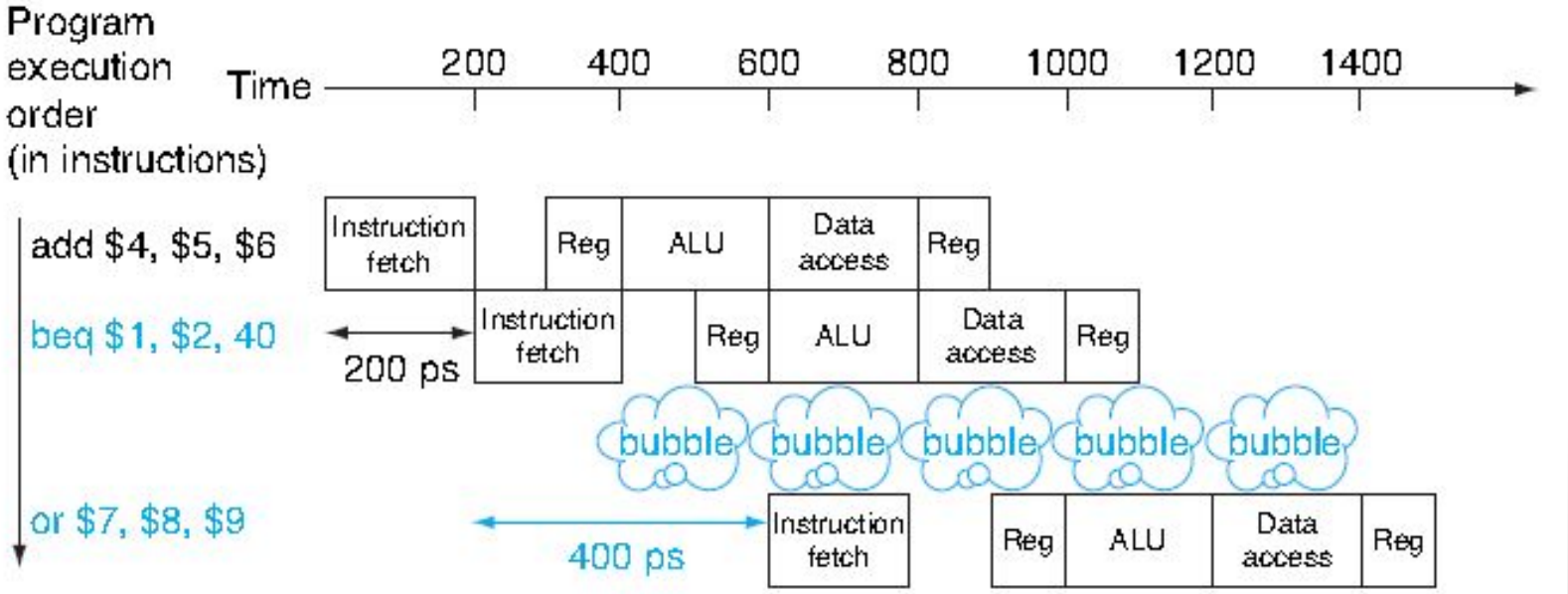
- Previsores de hardware modernos tomam decisões de que alguns desvios são tomados e outros não.
- **Previsores dinâmicos** fazem as escolhas dependendo do comportamento de cada desvio.
- Uma técnica comum é manter o histórico de cada desvio.
- Previsores de desvios modernos conseguem uma precisão de 90% dos desvios tomados.

Quais são os impactos (vantagens/desvantagens) quando são utilizados pipelines mais longos?

Hazards de controle

Hazards de controle – 3° solução: decisão adiada (*delayed branch*).

- É a decisão utilizada pelo MIPS. O software do MIPS sempre colocará uma instrução imediatamente após a instrução de *delayed branch* que não é afetada pelo desvio.
- A decisão do desvio mudará o endereço da instrução que vem após essa instrução segura. Como é o caso da instrução *add* no exemplo abaixo.



Exercícios pipelining

1) Analise cada sequência de código a seguir e indique se ocorrerá *stall*, se eles podem ser evitados usando a técnica de *forwarding* ou pode ser executada sem *stall* e *fowarding*:

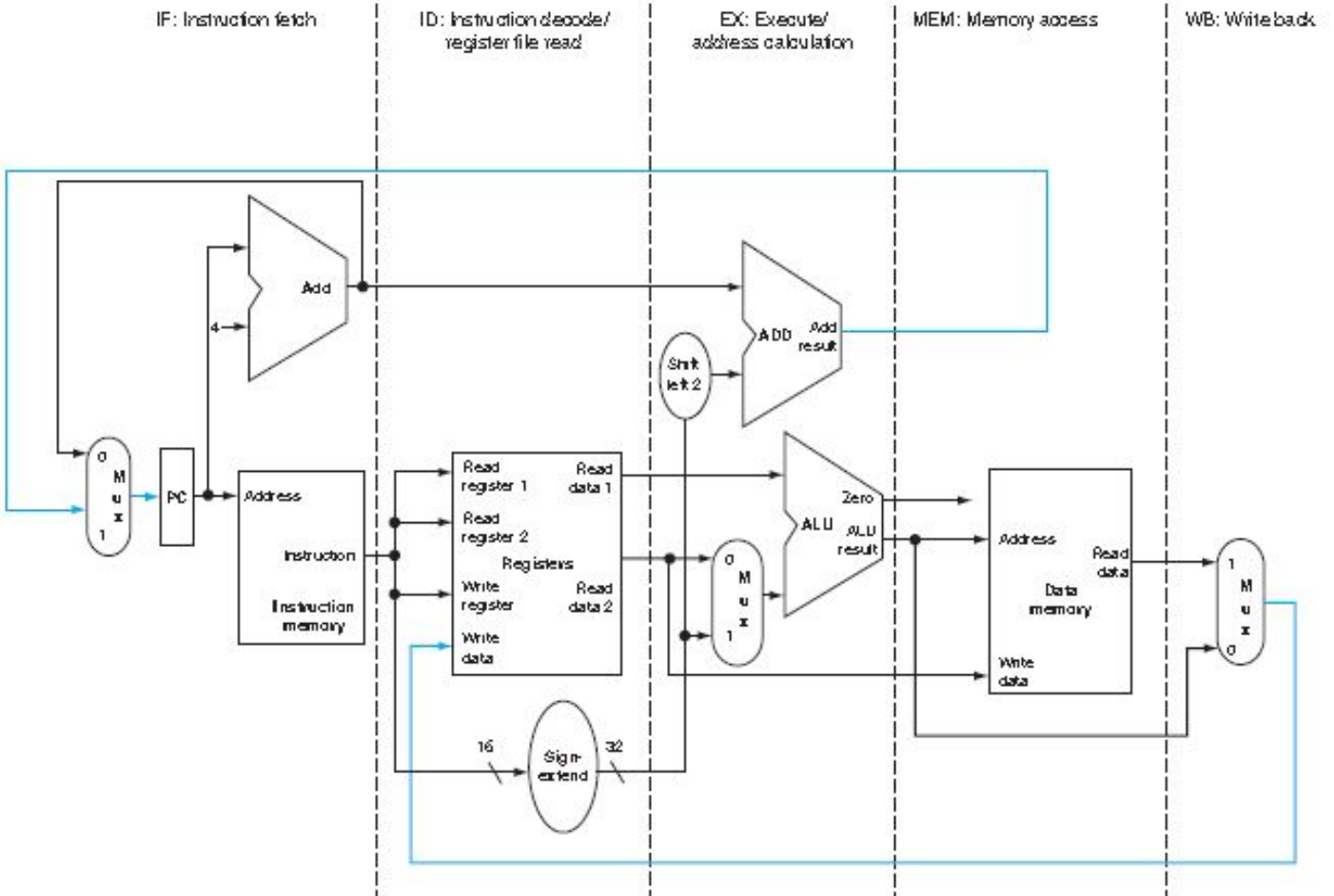
Sequence 1	Sequence 2

lw $t0,0($t0)
add $t1,$t0,$t0
	add $t1,$t0,$t0
	addi $t2,$t0,#5
	addi $t4,$t1,#5

Apresente 2 situações em uma implementação do MIPS com pipeline que é útil utilizar *forwarding:*

Caminho de dados usando pipeline

- Pipeline de **5 estágios**: divisão da organização do MIPS em 5 etapas.
- Permite que até 5 instruções sejam executadas em um mesmo ciclo.



Caminho de dados usando pipeline

- Observações:

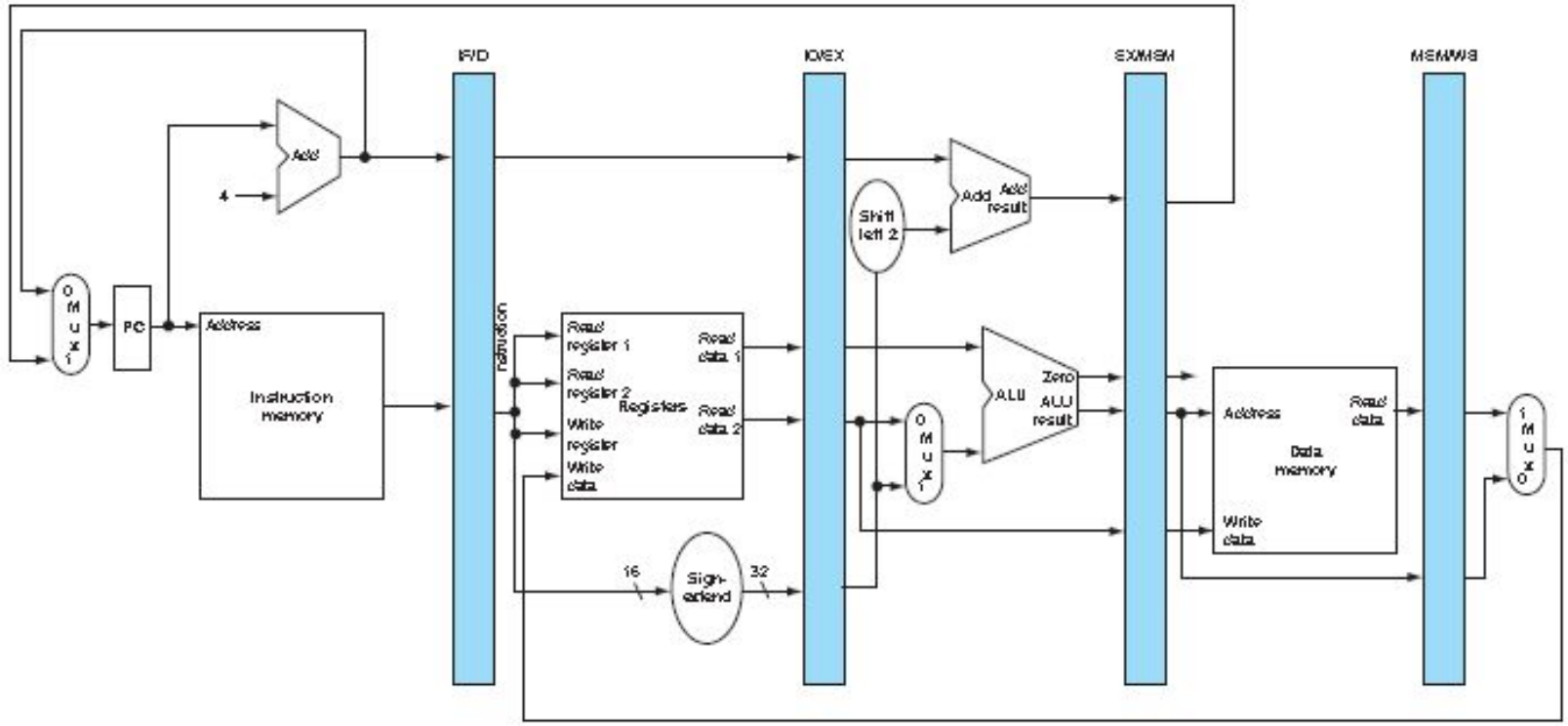
1. A memória de instruções é usada durante 1 dos 5 estágios, permitindo que seja compartilhada por outras instruções durante os outros 4 estágios.

2. A fim de reter o valor de uma instrução para os outros 4 estágios, o valor lido da memória de instrução precisa ser salvo em um registrador.

3. Etapas semelhantes se aplicam aos outros módulos de modo de que precisamos colocar registradores ao fim de cada estágio.

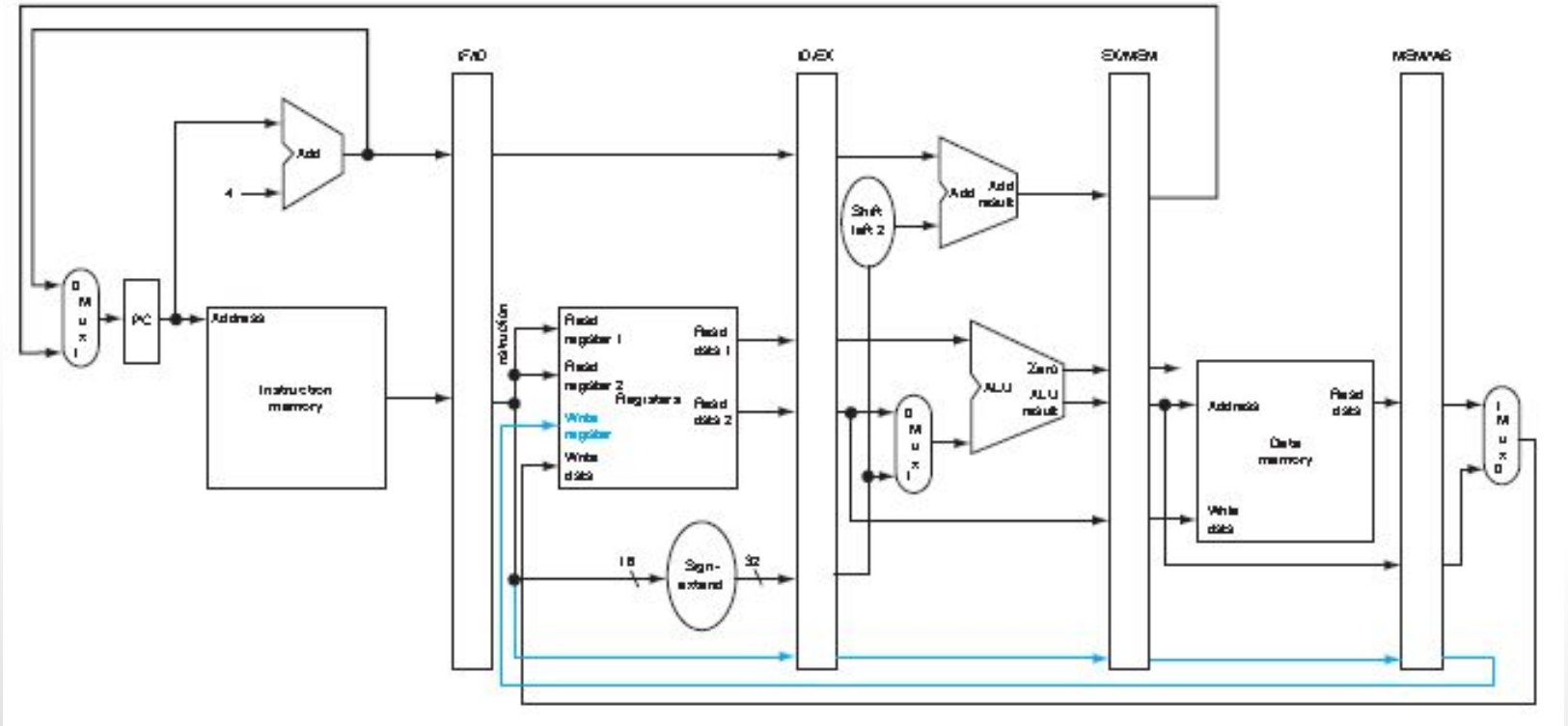
Caminho de dados usando pipeline

- Barreiras temporais e o uso de registradores em cada uma delas. Porque?
- Algumas informações precisam ser preservadas ao longo de toda a execução. Ex. Registrador de destino.



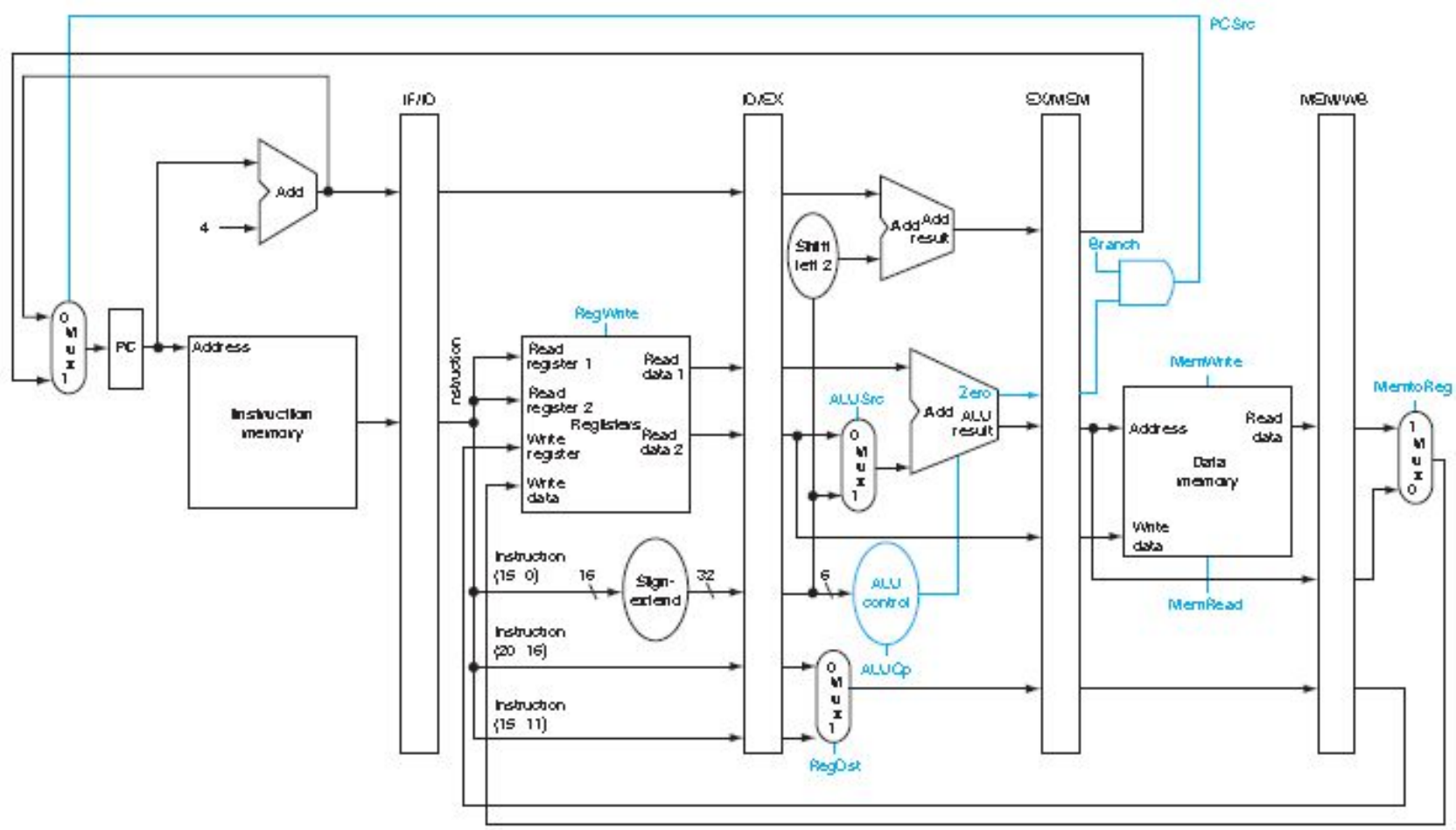
Caminho de dados usando pipeline

- A instrução lida em um determinado estágio precisa ser preservada de modo que cada registrador de pipeline contenha a parte da instrução necessária para o estágio atual e para os estágios seguintes.
- Exemplo: instrução de load.



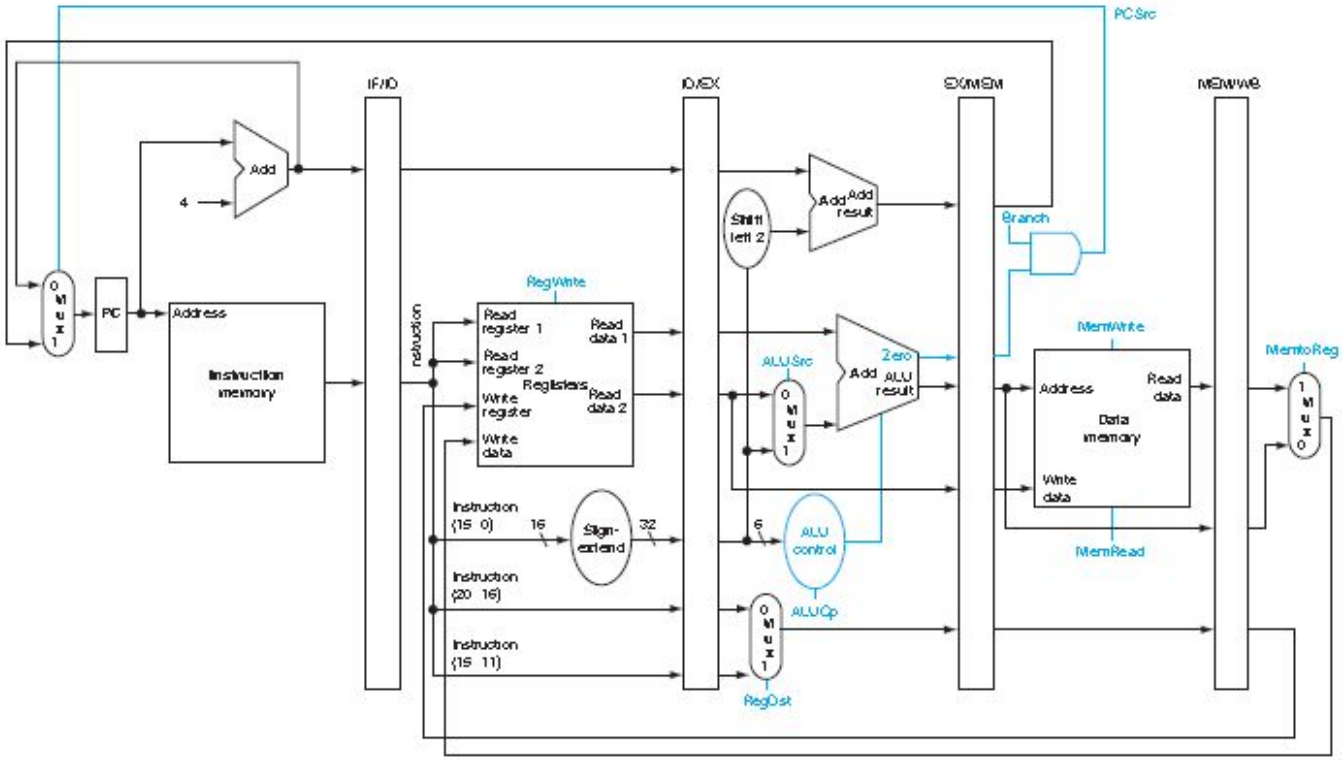
Controle do pipeline

- Sinais de controle identificados em cada estágio de pipeline



Controle do pipeline

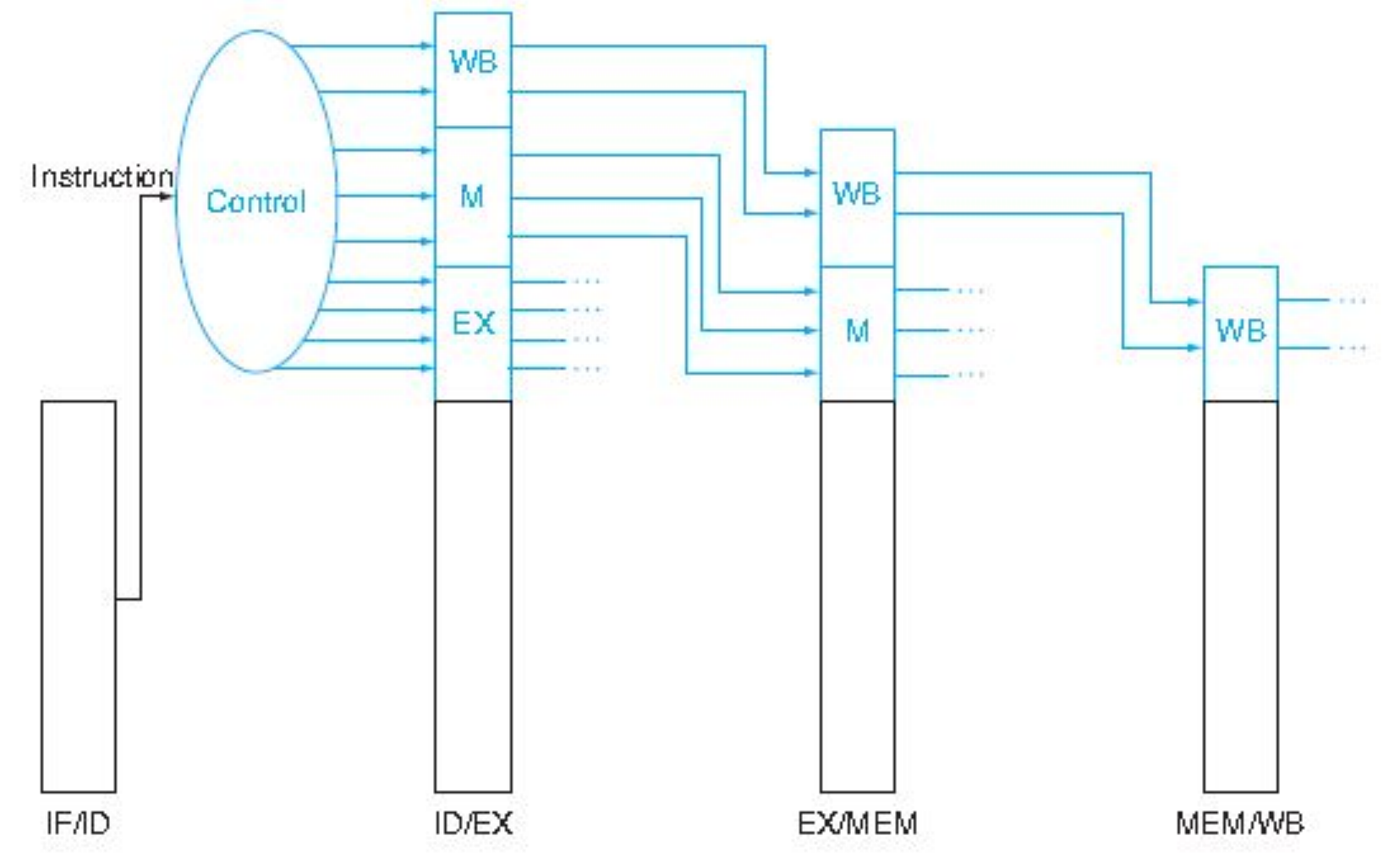
- Sinais de controle identificados em cada estágio de pipeline



Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

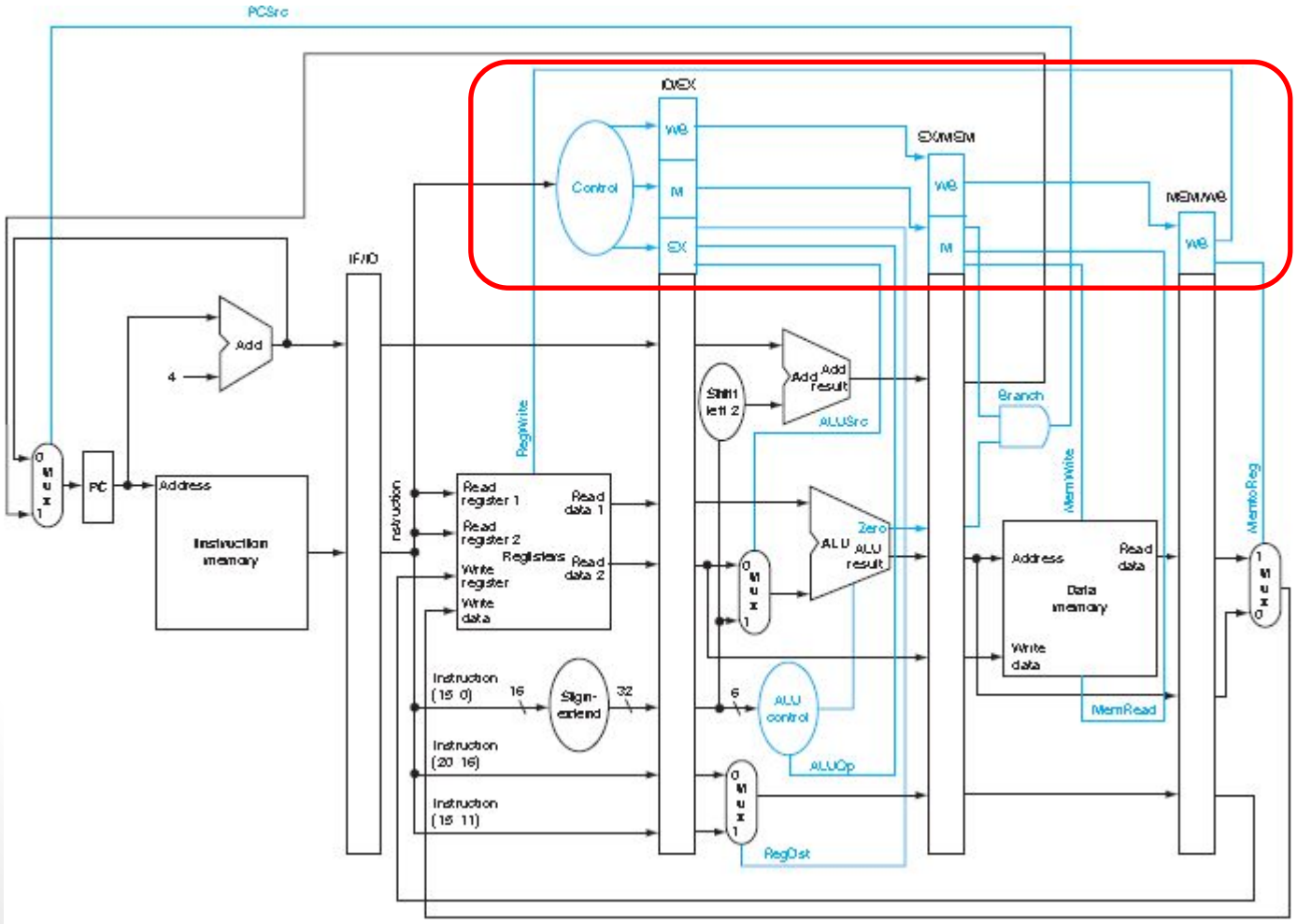
Controle do pipeline

- Linhas de controle utilizadas em cada estágio do pipeline:



Controle do pipeline

- Caminho de dados do pipeline com sinais de controle conectados às partes de controle dos registradores de pipeline.



Implementação Neander em VHDL

Implementação Neander

- Trata-se de um computador hipotético;
- Características:
  - ✓ Largura de dados e endereços de 8 bits;
  - ✓ Dados representados em complemento de 2;
  - ✓ Possui 1 acumulador de 8 bits;
  - ✓ Possui 1 apontador de programas de 8 bits (PC);
  - ✓ 1 registrador de estados com 2 códigos de condição: negativo (N) e zero (Z);

Implementação Neander

- O Neander só possui um modo de endereçamento, o modo direto (o campo de endereço contém o endereço do operando);

- O endereço passado na instrução corresponde o endereço de memória do operando;

- Nas instruções de desvio, o endereço contido na instrução corresponde à posição de memória onde está a instrução a ser executada.

Códigos de Condição do Neander

N (negativo):

1 – resultado é negativo

0 – resultado é positivo (zero é considerado positivo)

Z (zero)

1 – resultado é igual a 0;

0 – resultado é diferente de 0;

As instruções que alteram os códigos de condição são as instruções lógicas e aritméticas: ADD, NOT, AND, OR e a instrução LDA.

Conjunto de instruções do Neander

Código	Instrução	Comentário
0000	NOP	Nenhuma operação
0001	STA end	Armazena acumulador – (store)
0010	LDA end	Carrega acumulador – (load)
0011	ADD end	Soma
0100	OR end	“ou” lógico
0101	AND end	“e” lógico
0110	NOT	Inverte (complementa) acumulador
1000	JMP end	Desvio incondicional (jump)
1001	JN end	Desvio condicional (jump on negative)
1010	JZ end	Desvio condicional (jump on zero)
1111	HLT	Término da execução (halt)

Endereçamento do Neander

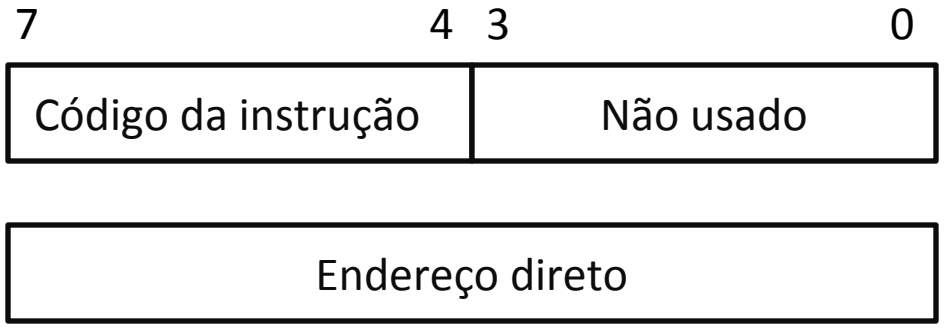
- *end* – endereço direto
- Nas instruções STA, LDA, ADD, OR e AND, *end* corresponde ao endereço do operando.
- Nas instruções JMP, JN e JZ, *end* corresponde ao endereço de desvio.

Endereçamento do Neander

Instrução	Comentário
NOP	Nenhuma operação
STA end	MEM(end) <- AC
LDA end	AC <- MEM(end)
ADD end	AC <- MEM(end) + AC
OR end	AC <- MEM(end) or AC
AND end	AC <- MEM(end) and AC
NOT	AC <- NOT AC
JMP end	PC <- end
JN end	IF N=1 then PC <- end
JZ end	IF Z=1 then PC <- end
HLT	Término da execução (halt)

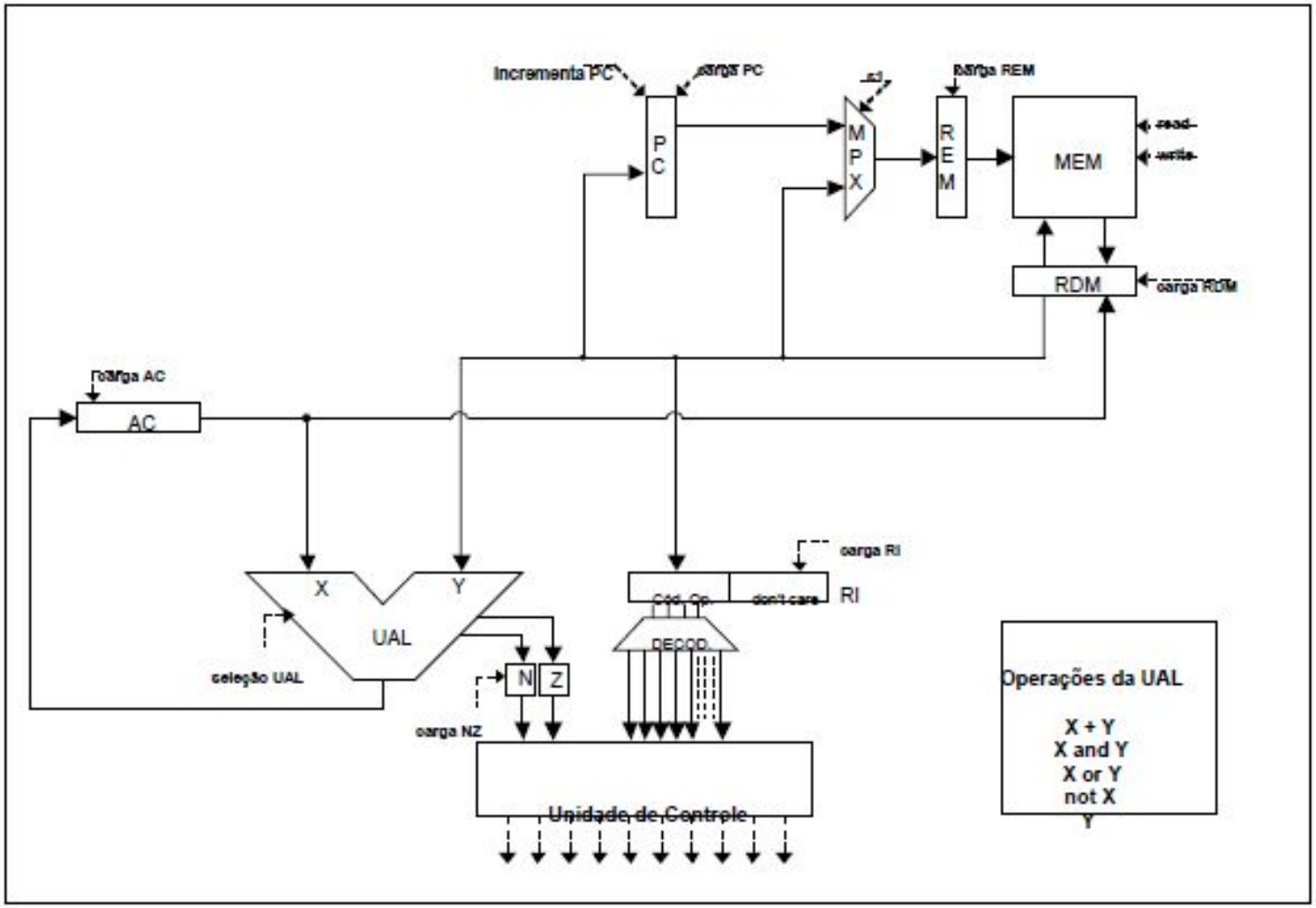
Formato das instruções do Neander

- As instruções são formadas por um ou dois bytes, ou seja, ocupam 1 ou 2 posições da memória;



- No Neander as instruções de 2 bytes são aquelas que fazem referência à memória.

Organização do Neander



Sinais de controle do Neander

Transferência	Sinais de controle
REM ← PC	sel=0, carga REM
PC ← PC + 1	incrementa PC
RI ← RDM	carga RI
REM ← RDM	sel =1, carga REM
RDM ← AC	carga RDM
AC ← RDM; Atualiza N e Z	UAL(Y), carga AC, carga NZ
AC ← AC + RDM; Atualiza N e Z	UAL(ADD), carga AC, carga NZ
AC ← AC or RDM; Atualiza N e Z	UAL(OR), carga AC, carga NZ
AC ← AC and RDM; Atualiza N e Z	UAL(AND), carga AC, carga NZ
AC ← not(AC); Atualiza N e Z	UAL(NOT), carga AC, carga NZ
PC ← RDM	carga PC

Sinais de controle do Neander

tempo	STA	LDA	ADD	OR	AND	NOT
t0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
t1	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC
t2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
t3	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	UAL(NOT), carga AC, carga NZ, goto t0
t4	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	
t5	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	
t6	carga RDM	Read	Read	Read	Read	
t7	Write, goto t0	UAL(Y), carga AC, carga NZ, goto t0	UAL(ADD), carga AC, carga NZ, goto t0	UAL(OR), carga AC, carga NZ, goto t0	UAL(AND, carga AC, carga NZ, goto t0	

Sinais de controle do Neander

tempo	JMP	JN, N=1	JN, N=0	JZ, Z=1	JZ, Z=0	NOP	HLT
t0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
t1	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC
t2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
t3	sel=0, carga REM	sel=0, carga REM	incrementa PC, goto t0	sel=0, carga REM	incrementa PC, goto t0	goto t0	Halt
t4	Read	Read		Read			
t5	carga PC, goto t0	carga PC, goto t0		carga PC, goto t0			
t6							
t7							

Tabela 10.4- Sinais de controle para JMP, JN, JZ, NOP e HLT

VHDL Neander