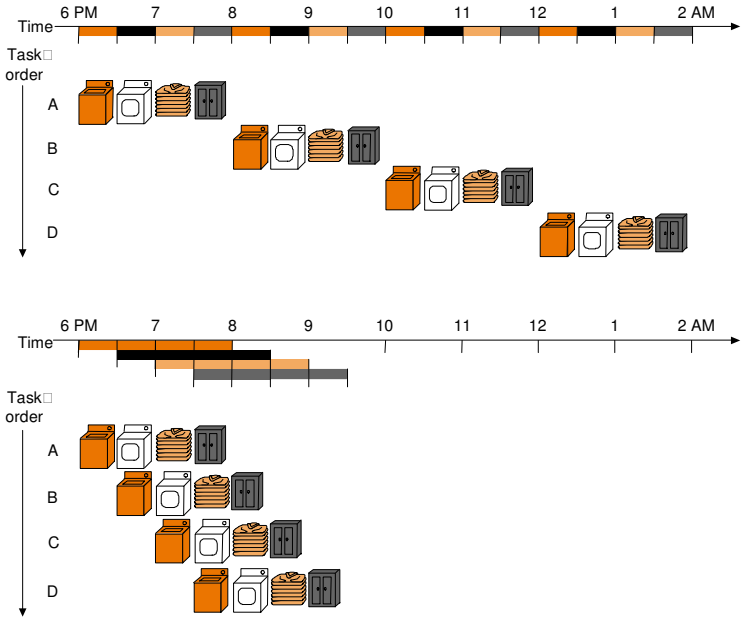
Pipeline

Lavanderia – analogia com o pipelining



Pipeline de instruções no MIPS

- **Fetch da instrução**
- **Leitura dos registradores e decodificação**
- **Execução da operação ou cálculo de endereço**
- **Acesso ao operando na memória**
- **Escrita do resultado em um registrador**

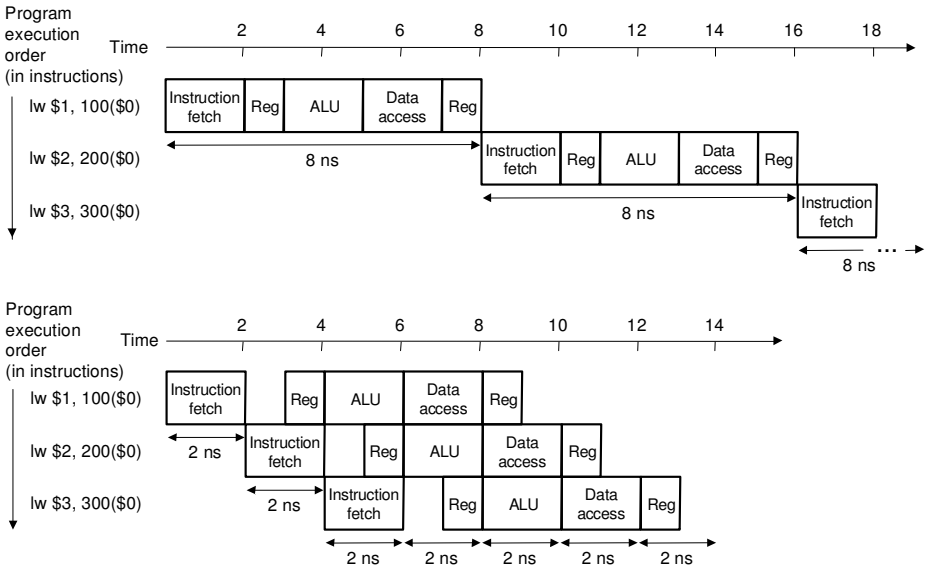
Exemplo

- Compare o tempo médio entre instruções da implementação em single-cycle (uma instrução por ciclo) com uma implementação com pipeline. Supor maior tempo de operação para acesso à memória = 2ns, operação da ULA = 2ns e acesso ao register file = 1ns. (Instrs lw, sw, add, sub, and, or slt e beq).
- Inicialmente suponha a execução de 3 instruções lw (tempo entre o início da 1ª instrução e o início da 4ª instrução)

Tempo total para as oito instruções calculado a partir do tempo de cada componente

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (sw)	2 ns	1 ns	2 ns	2 ns		7 ns
R-format (add, sub, and, or, slt)	2 ns	1 ns	2 ns		1 ns	6 ns
Branch (beq)	2 ns	1 ns	2 ns			5 ns

Execução não-pipeline X pipeline



OBS.:

- Sob condições ideais, com estágios balanceados, o speedup do pipeline é igual ao número de estágios do pipeline (5 estágios , 5 vezes mais rápido)
- Na realidade o tempo de execução de uma instrução é um pouco superior (overheads) ➔ speedup é menor que o número de estágios do pipeline

Desempenho do pipeline é devido ao aumento do throughput.

Suponha a execução de 1003 instruções

com pipeline ➔
1000 X 2ns + 14 = 2014 (para cada instrução adiciono 2ns)

sem pipeline ➔ 1000 X 8ns + 24 = 8024

spedup = 8024 / 2014 = 3.98 ~~ 8 / 2

Projeto de um conjunto de instruções para pipeline

- **O que torna a implementação mais fácil**
  - **Instruções de mesmo tamanho**
  - **Poucos formatos, com campos de registradores sempre dispostos no mesmo lugar (Simetria, no 2º estágio podemos ler registradores e decodificar ao mesmo tempo).**
  - **Acesso à memória apenas com as instruções lw e sw.**
  - **Operandos alinhados na memória: o dado pode ser transferido da memória para a CPU e CPU para a memória em um único estágio do pipeline.**

Projeto de um conjunto de instruções para pipeline

- **O que torna a implementação mais dificil**
  - **Hazard**
    - **Hazard Estrural**
    - **Hazard de Controle**
    - **Hazard de Dados**

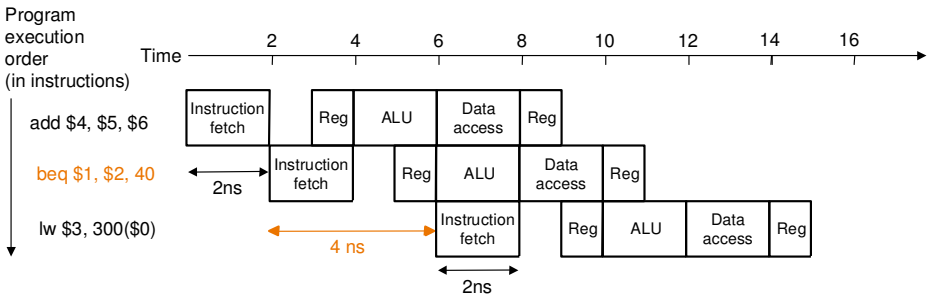
Pipeline Hazards

- **Hazard Estrutural**
  - **O hardware não suporta uma combinação de instruções que queremos executar em um único período de clock**
    - **Ex.: escrever e ler da memória em um mesmo ciclo**

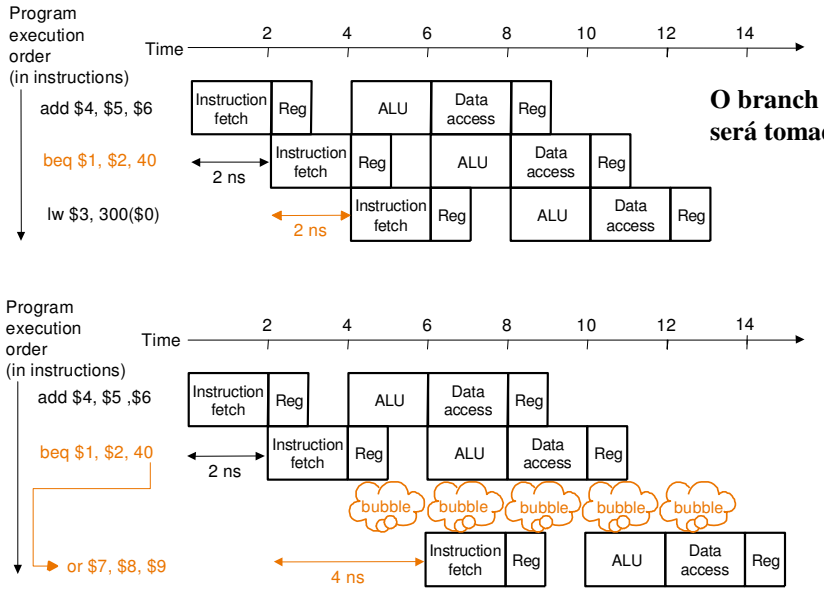
Pipeline Hazards

- **Hazard de Controle**
  - **Problemas devido à execução de instruções de desvio**
    - **Ex.: Quando um branch é tomado, como tratar a(s) instruções que seguem (fisicamente) o branch no programa e que já estão no pipeline**

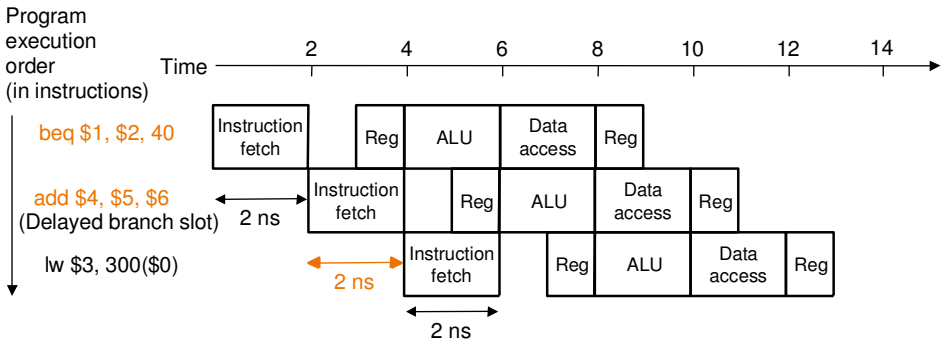
Pipelining stalling para instruções branch



Branch prediction: Tentar "adivinhar" qual dos caminhos do branch será tomado



Pipeline delayed branch



Hazard de Dados

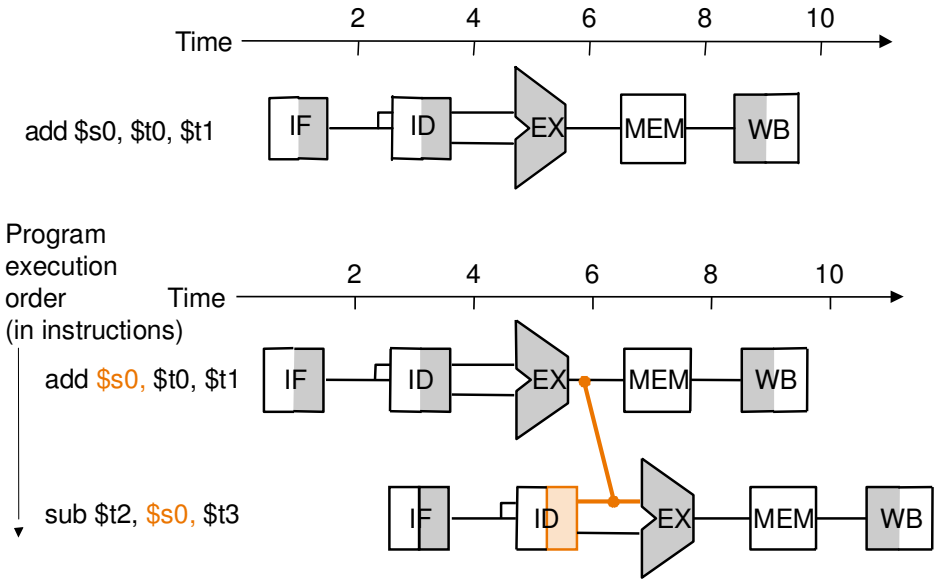
- **Quando uma instrução necessita de um dado que ainda não foi calculado**
  - **Ex.:** **add $s0,$t0,$t1**
    sub $t2,$s0,$t3

Soluções :

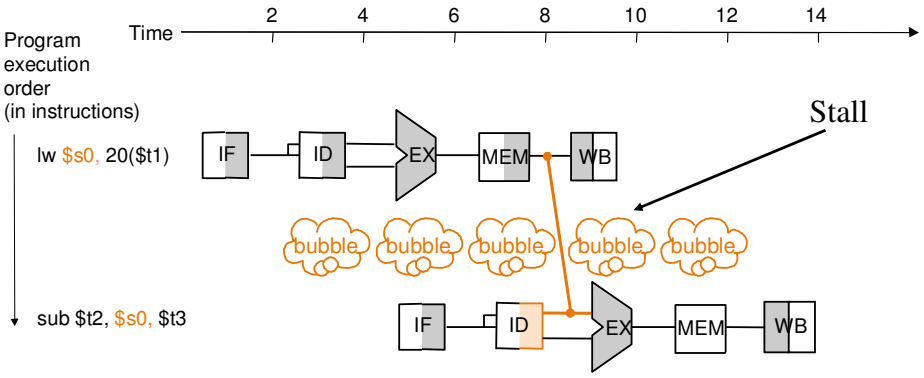
Compilador (programador) gera código livre de data hazard (introduzindo, por ex., instruções nop no código; alterando a ordem das instruções; ...)

Stall; Forwarding ou bypassing

Hazard de Dados



Hazard de Dados



Exemplo

- **Encontre o hazard no código abaixo e resolva-o:**

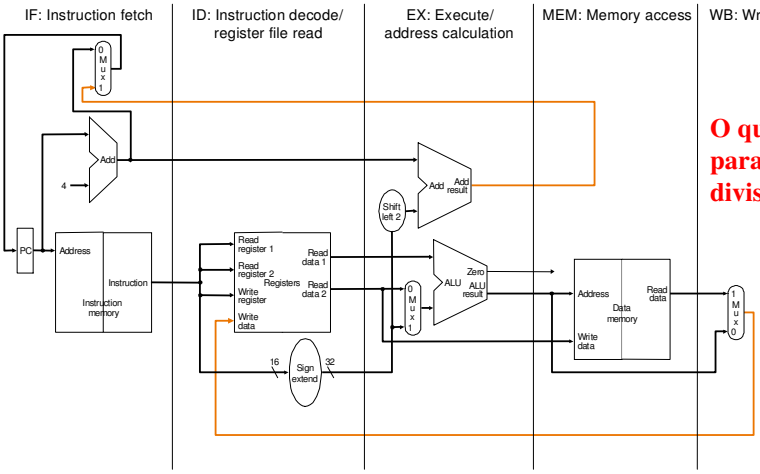
```
                                # $t1 tem o end. de v[k]
lw      $t0, 0($t1)             # $t0 = v[k]
lw      $t2,4($t1)              # $t2 = v[k+1]
sw      $t2, 0($t1)             # v[k] = $t2
sw      $t0, 4($t1)             # v[k+1] = $t0
```

Solução:

```
                                # $t1 tem o end. de v[k]
lw  $t0, 0($t1)                 # $t0 = v[k]
    lw      $t2,4($t1)          # $t2 = v[k+1]
    sw      $t0, 4($t1)         # v[k+1] = $t0
    sw      $t2, 0($t1)         # v[k] = $t2
```

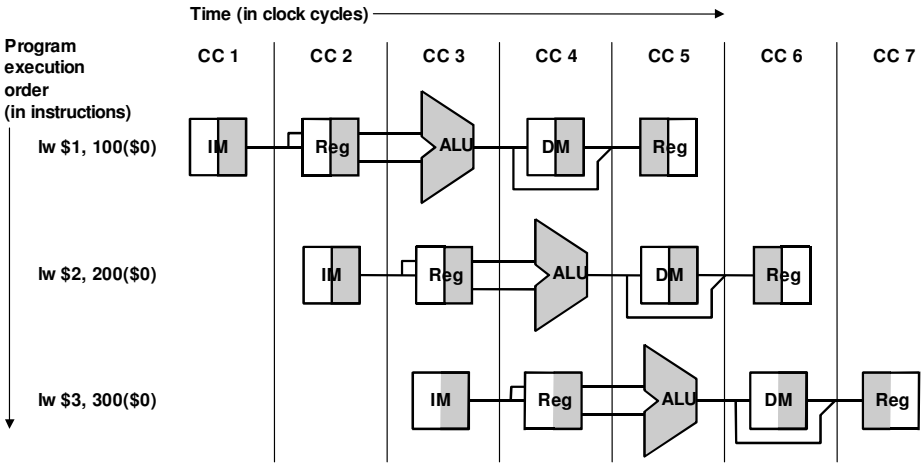
Pipeline: Idéia Básica

- **5 estágios: Fetch; Decodificação e leitura dos regs; execução ou cálculo de end. ; acesso à memória; escrita no reg. destino**

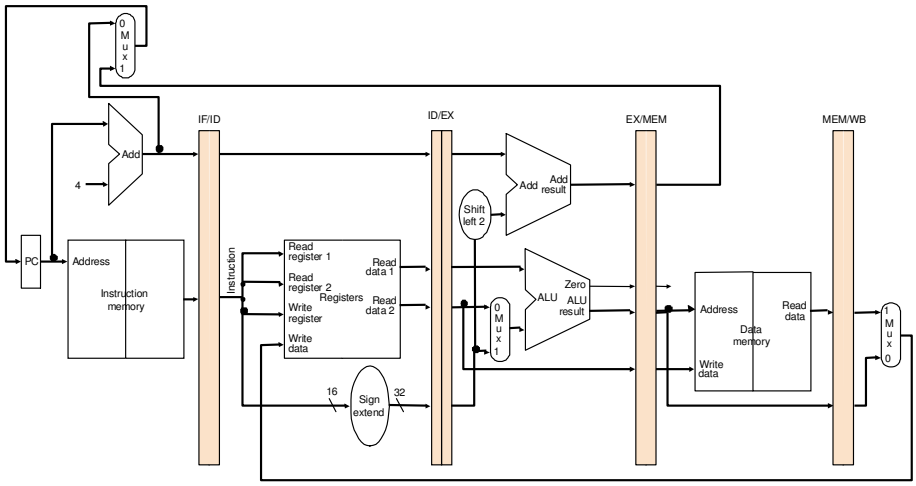


O que é necessário para tornar cada divisão em estágios?

Instruções sendo executadas pelo datapath



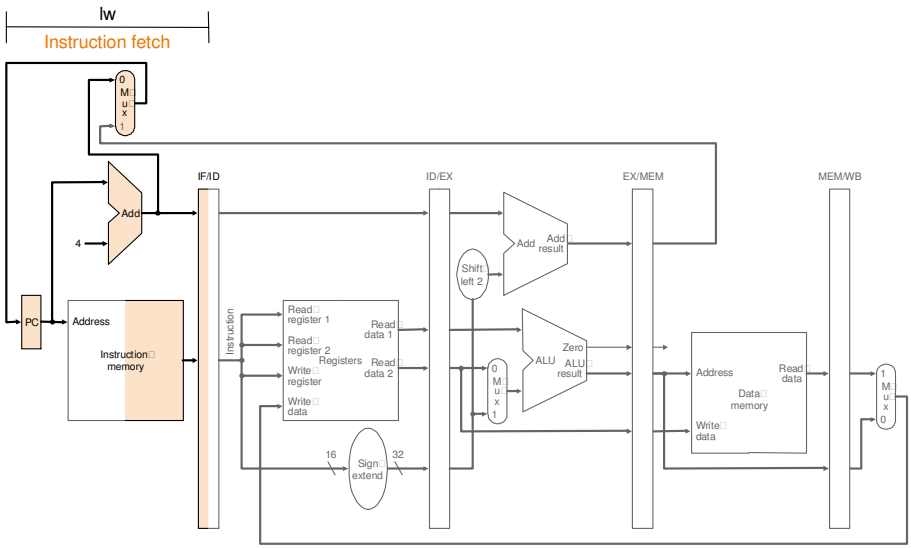
Pipelined Datapath



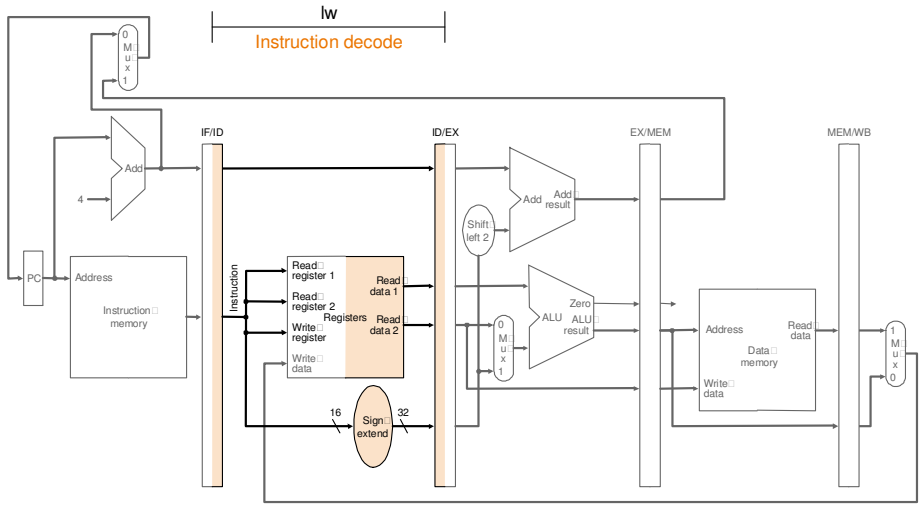
Pode acontecer algum problema nesta solução se não existir dependência de dados?

A execução de qual instrução causa o problema?

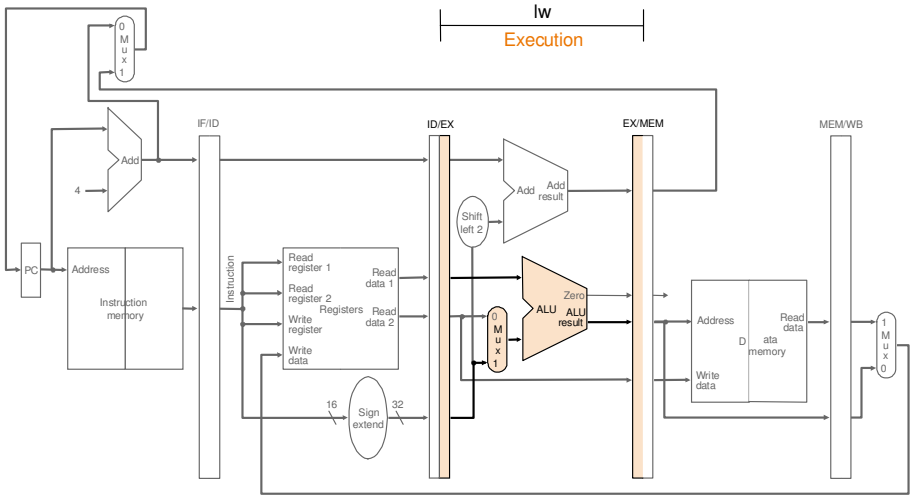
Pipelined Datapath



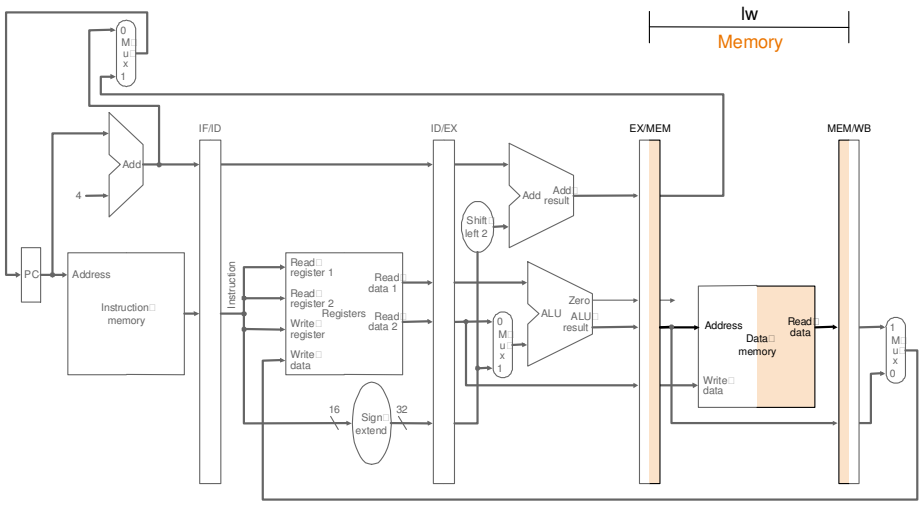
Pipelined Datapath



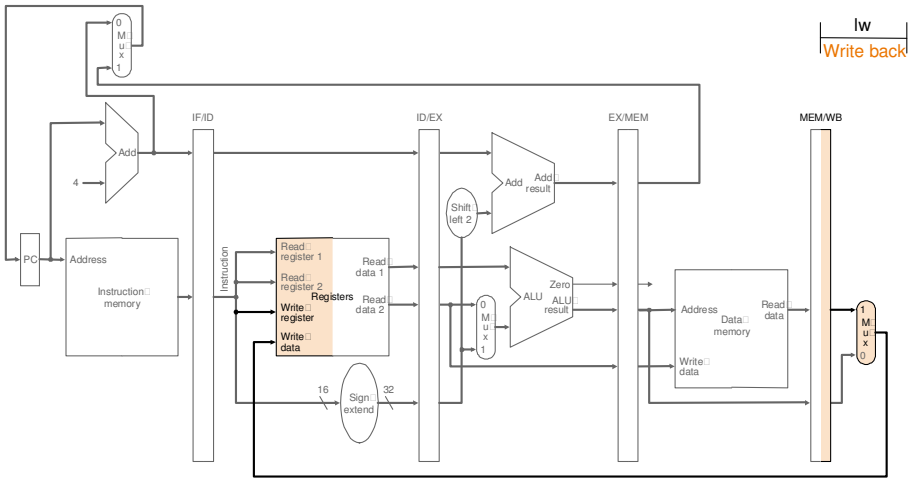
Pipelined Datapath



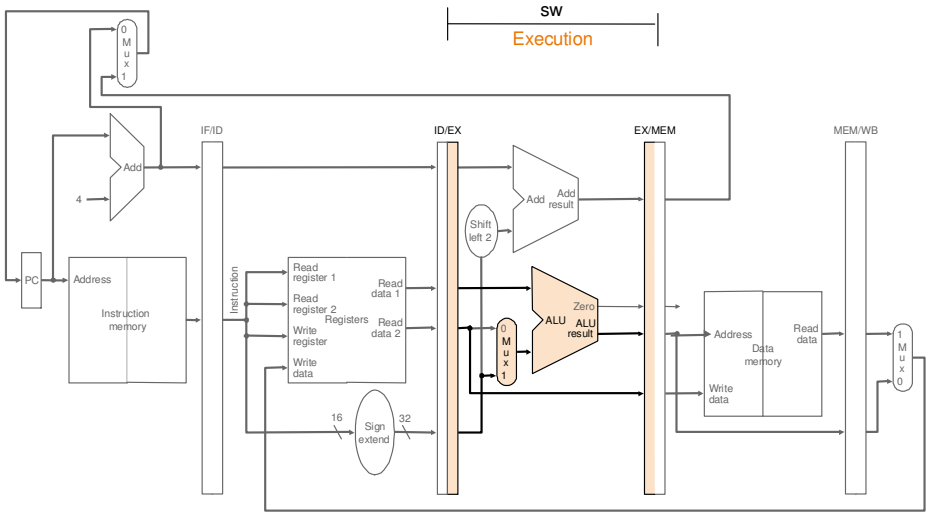
Pipelined Datapath



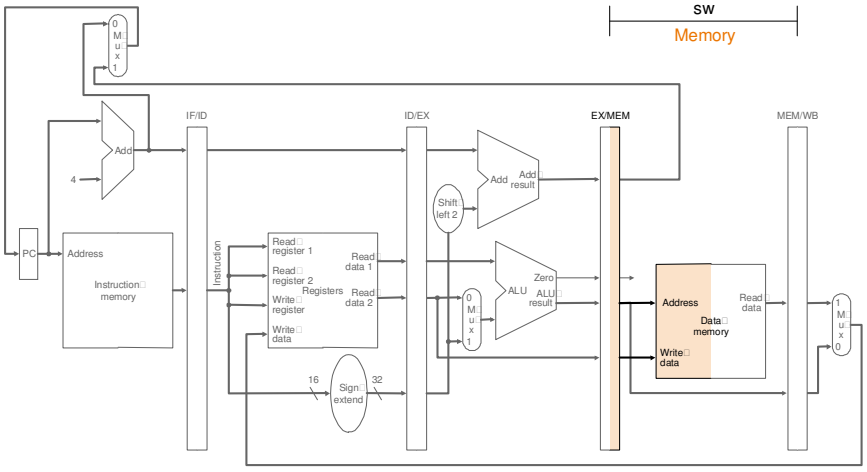
Pipelined Datapath



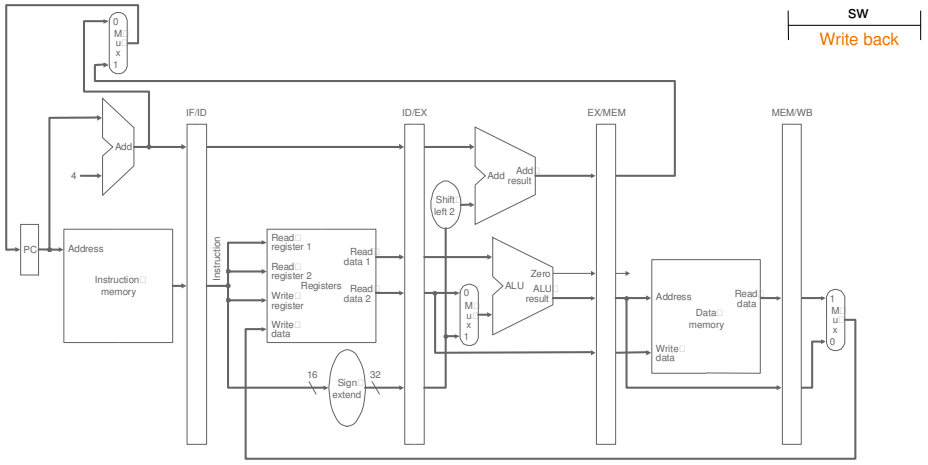
Pipelined Datapath



Pipelined Datapath

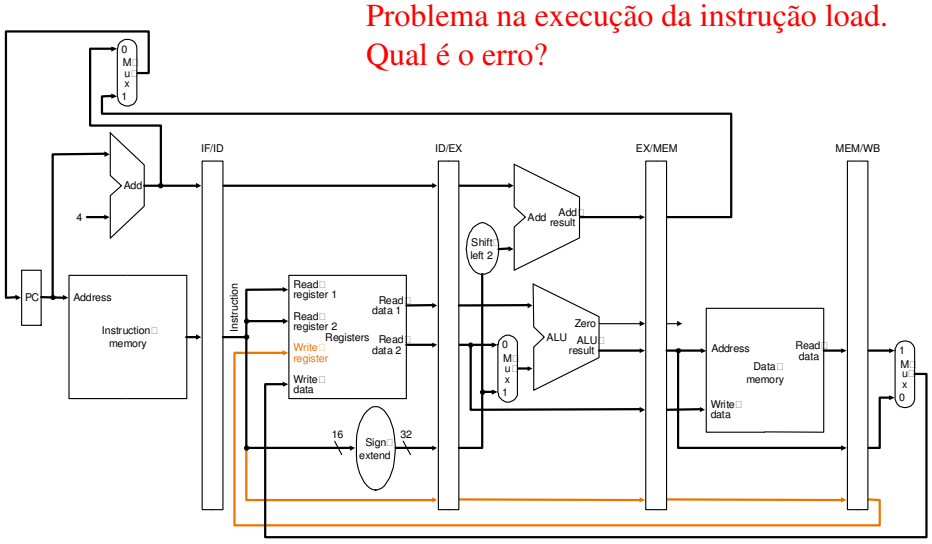


Pipelined Datapath

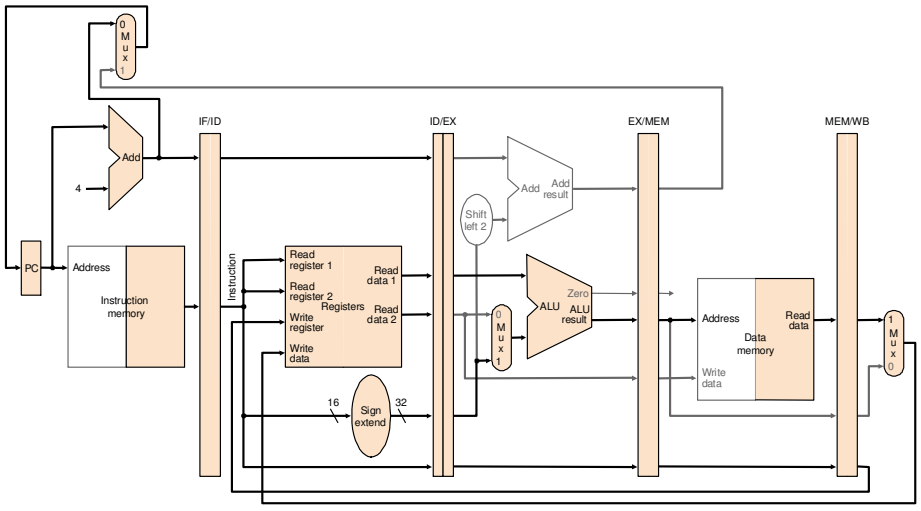


Datapath Correto

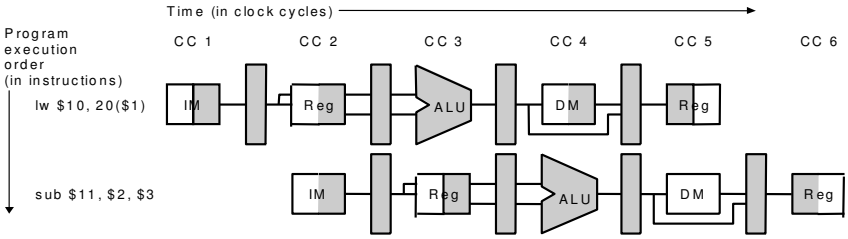
Problema na execução da instrução load.
Qual é o erro?



Datapath com os estágios usados para um instrução lw



Representações Gráfica do Pipeline



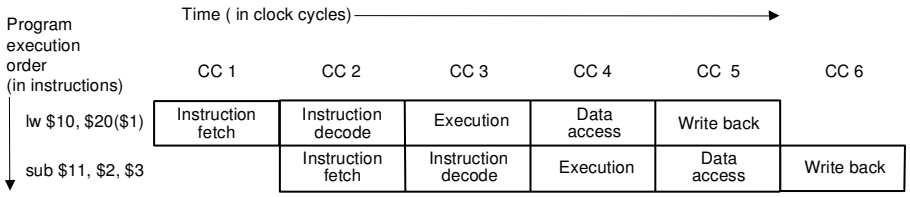
Ajuda a responder perguntas como:

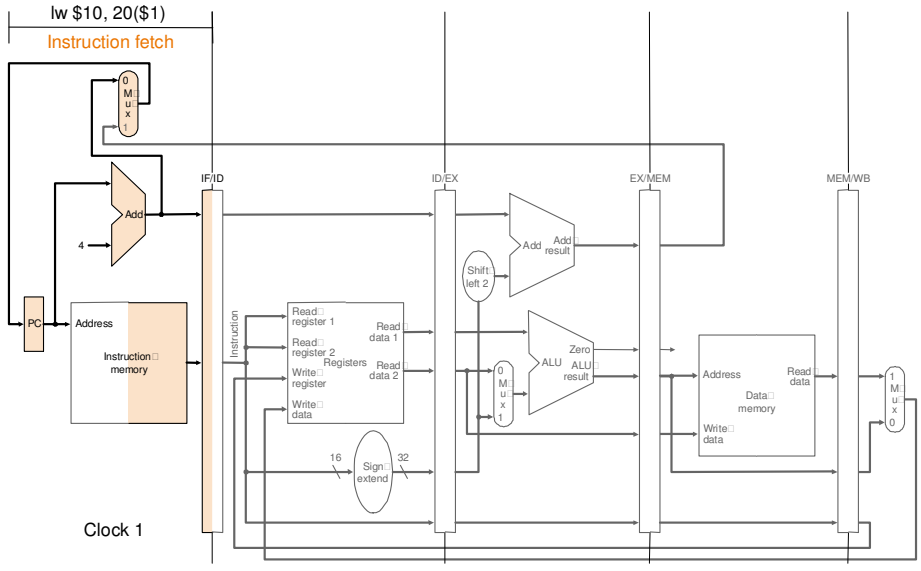
Quantos ciclos são gasto para executar este código?

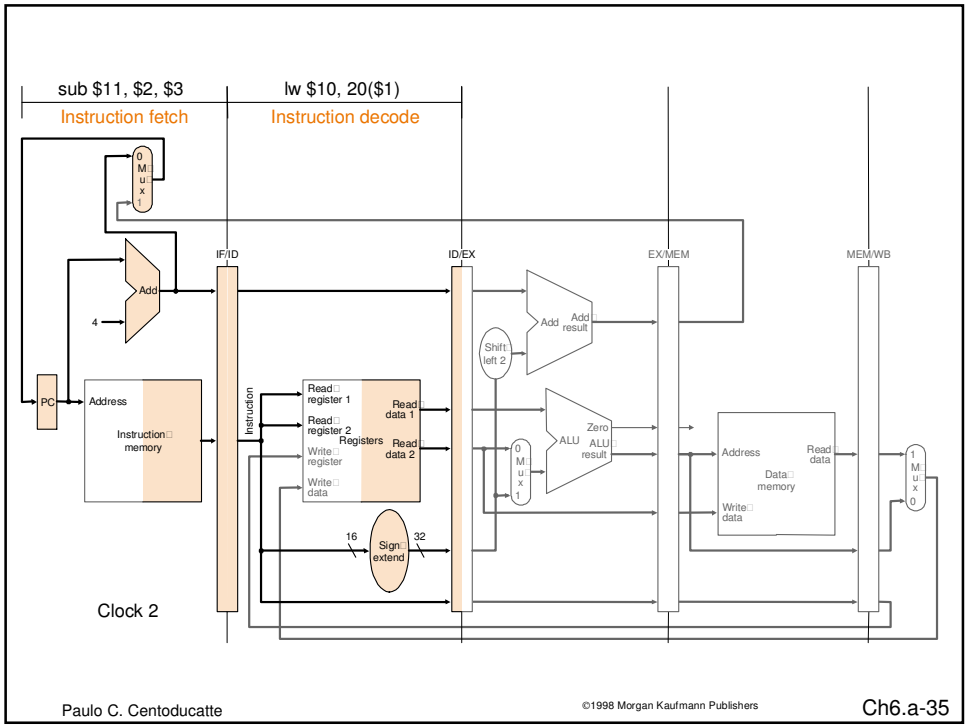
O que a ALU está fazendo durante o ciclo 10?

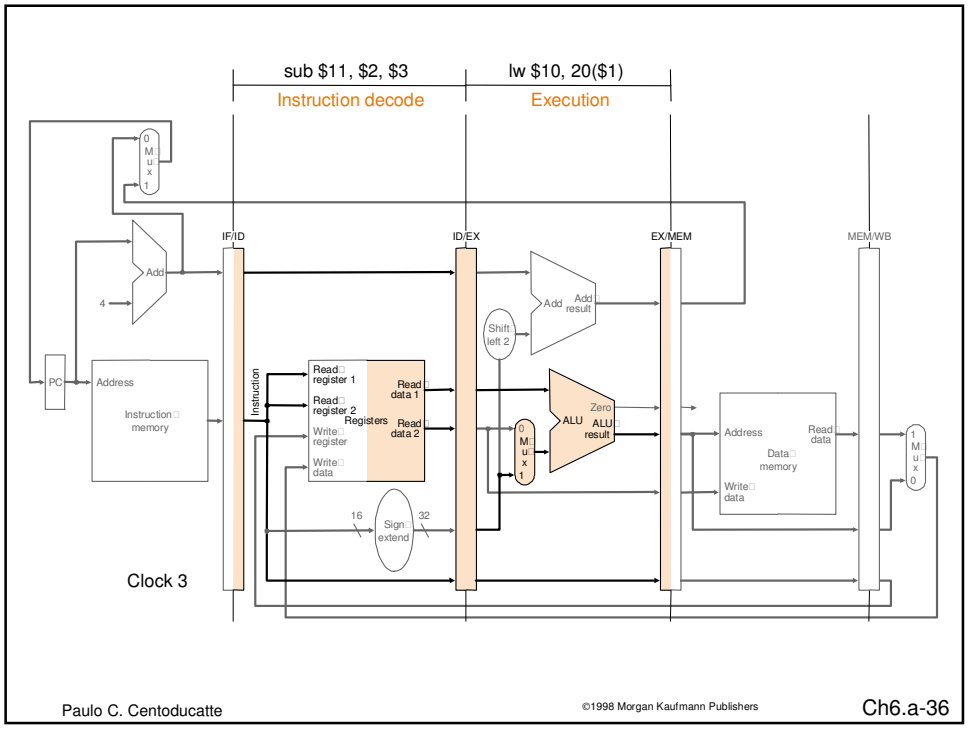
Ajuda a entender os datapaths

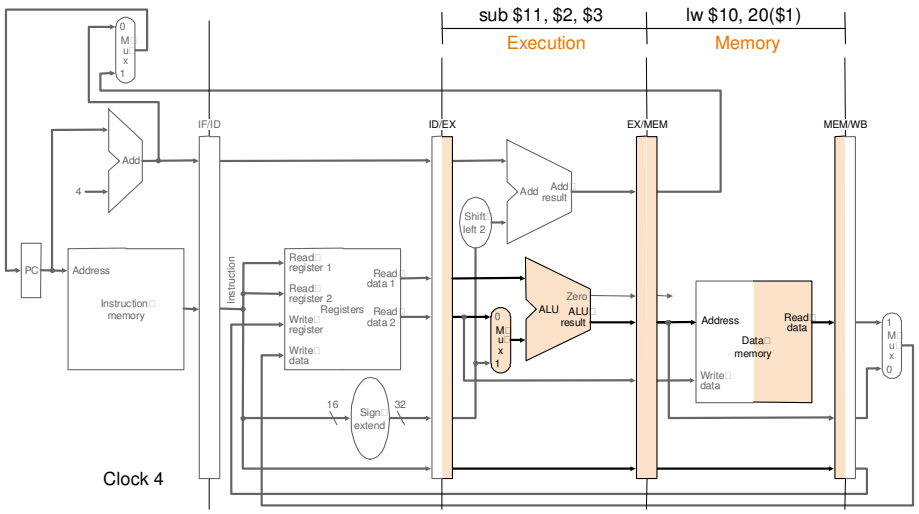
Representações Gráfica do Pipeline

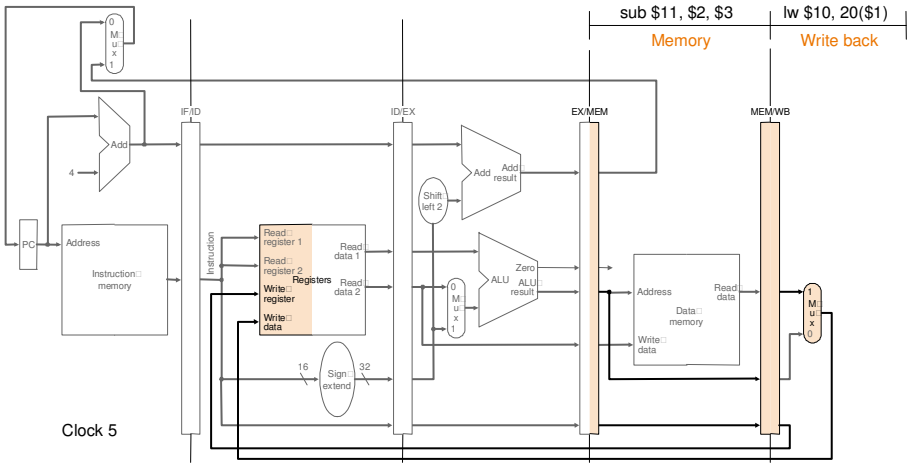


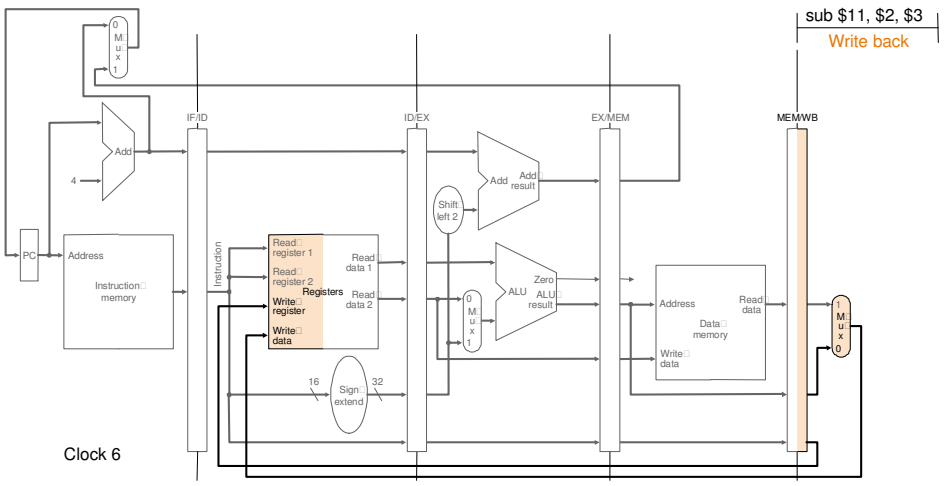




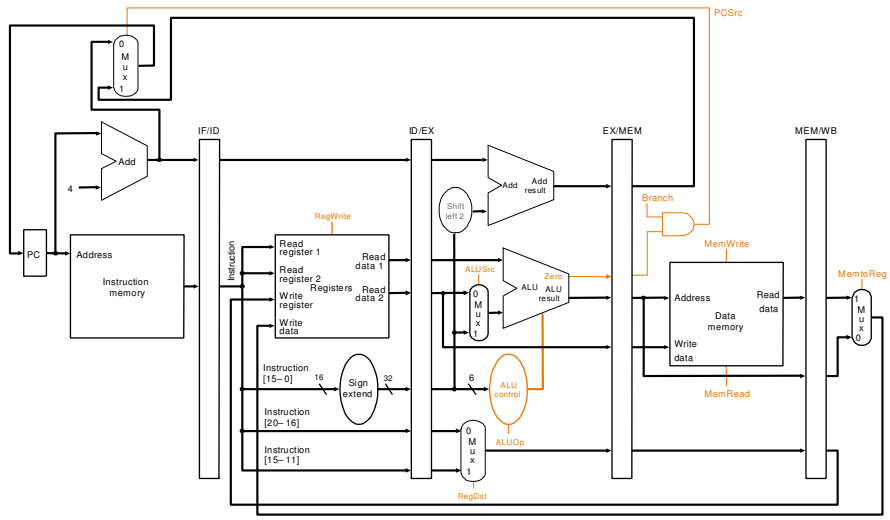








Controle do Pipeline



Controle do Pipeline

- **5 estágios. O que deve ser controlado em cada estágio?**
  - **1º: Fetch da instrução e incremento do PC**
  - **2º: Decodificação da instrução e Fetch dos registradores**
  - **3º: Execução**
  - **4º: Acesso à memória**
  - **5º: Write back**

Sinais de controle

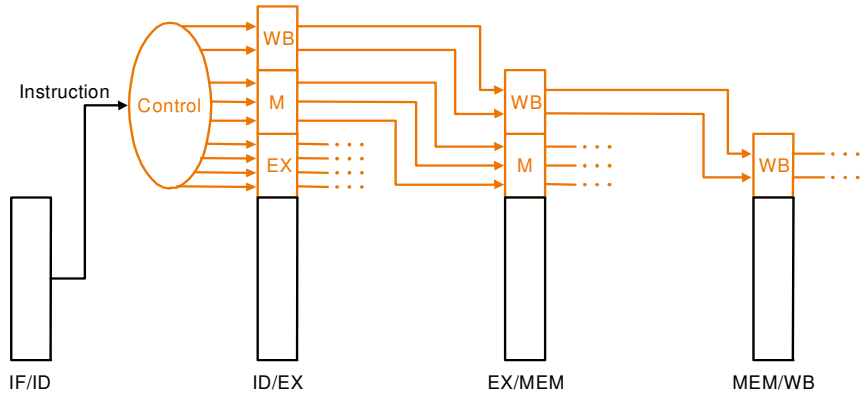
Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Sinais de controle

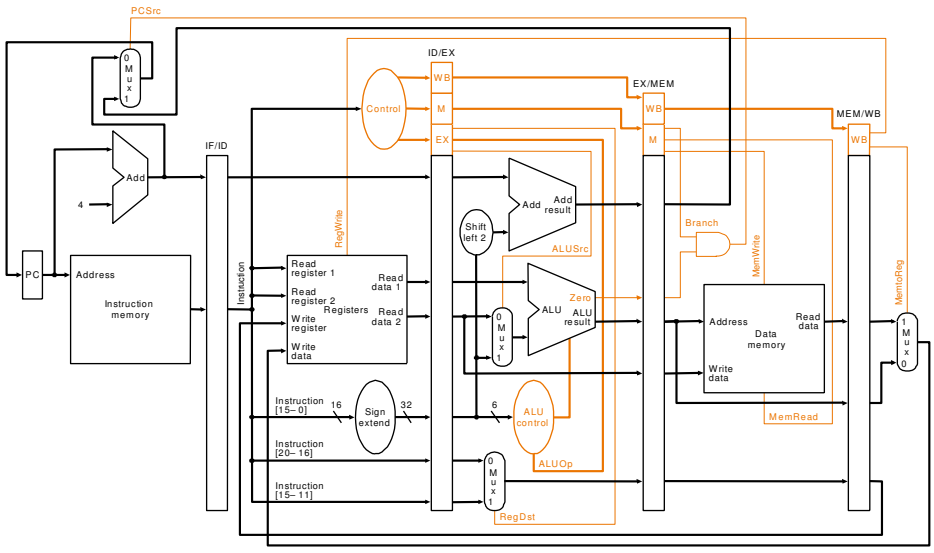
Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20–16).	The register destination number for the Write register comes from the rd field (bits 15–11).
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Sinais de controle

	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to 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

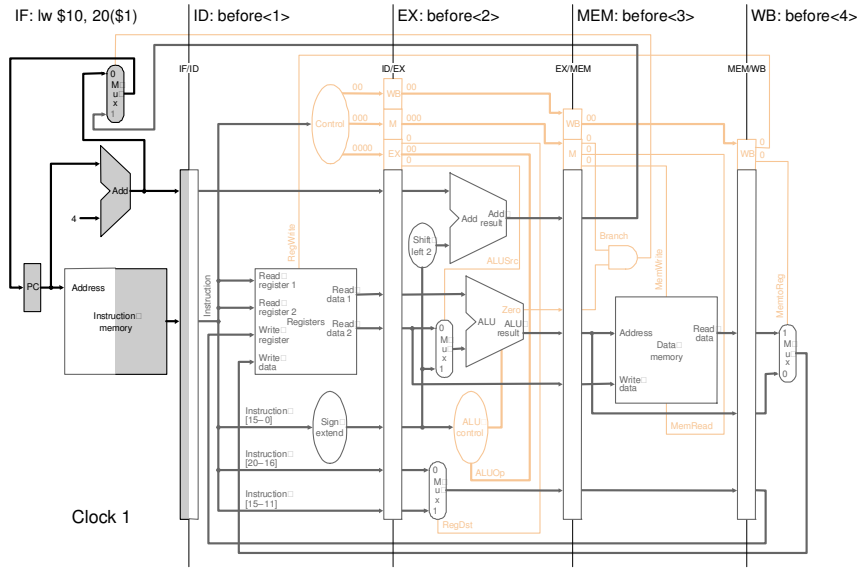


Datapath com Controle



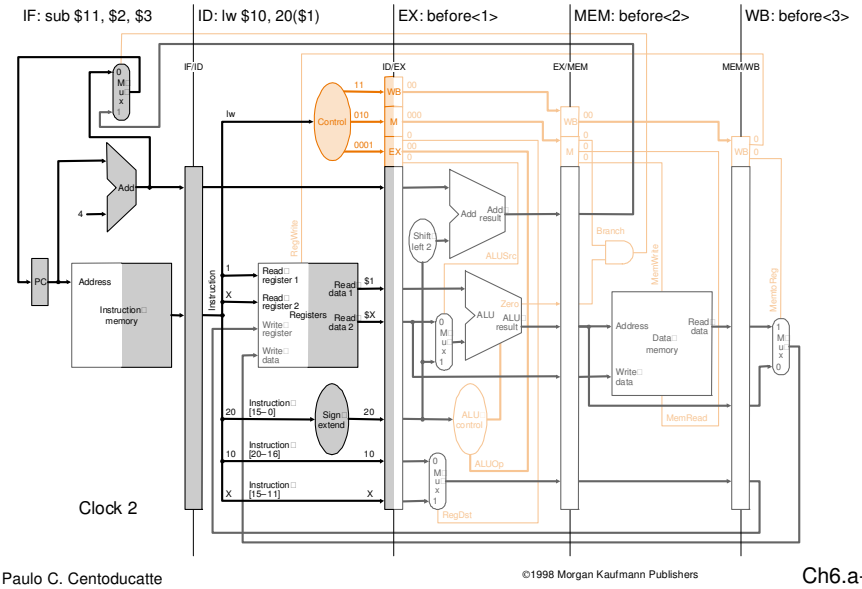
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

1º ciclo



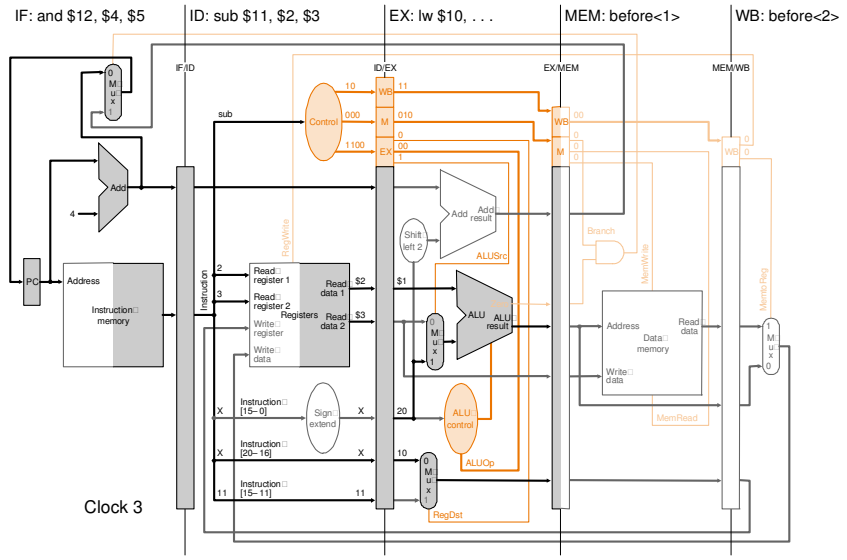
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

2º ciclo



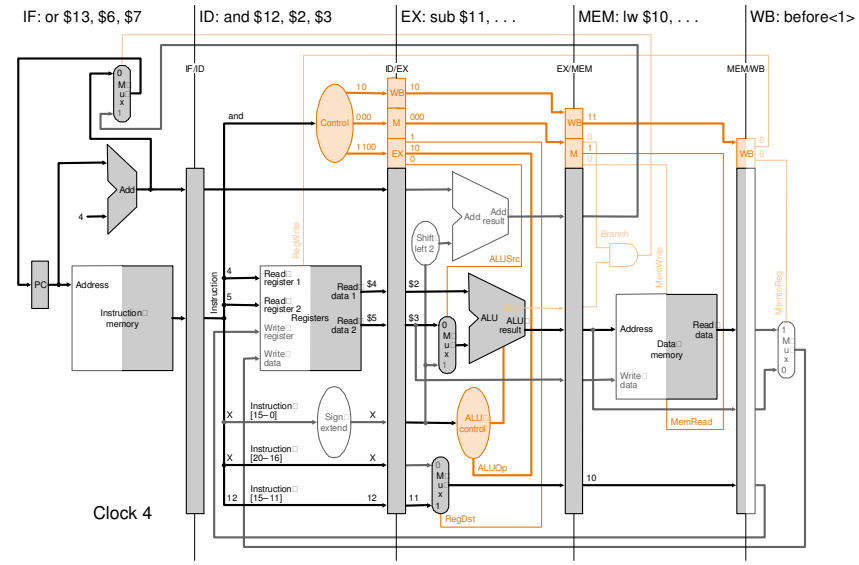
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

3º ciclo



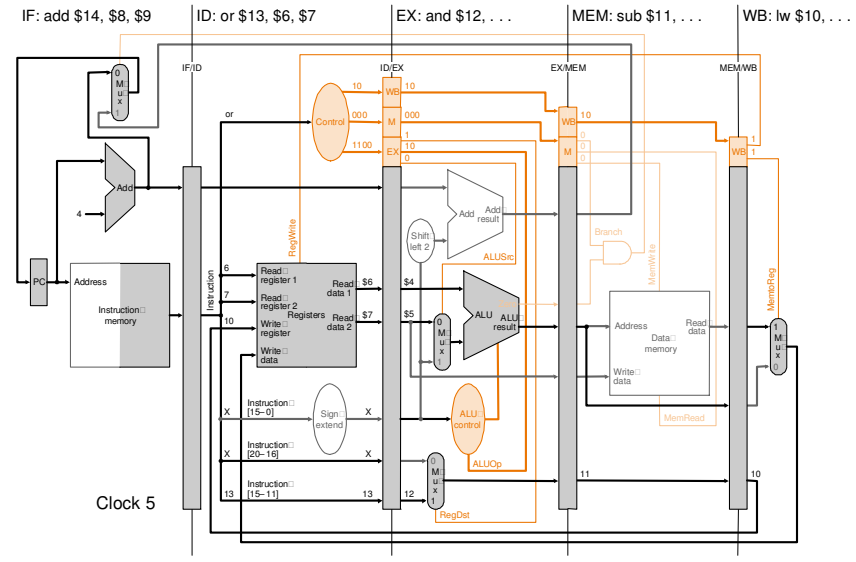
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

4º ciclo



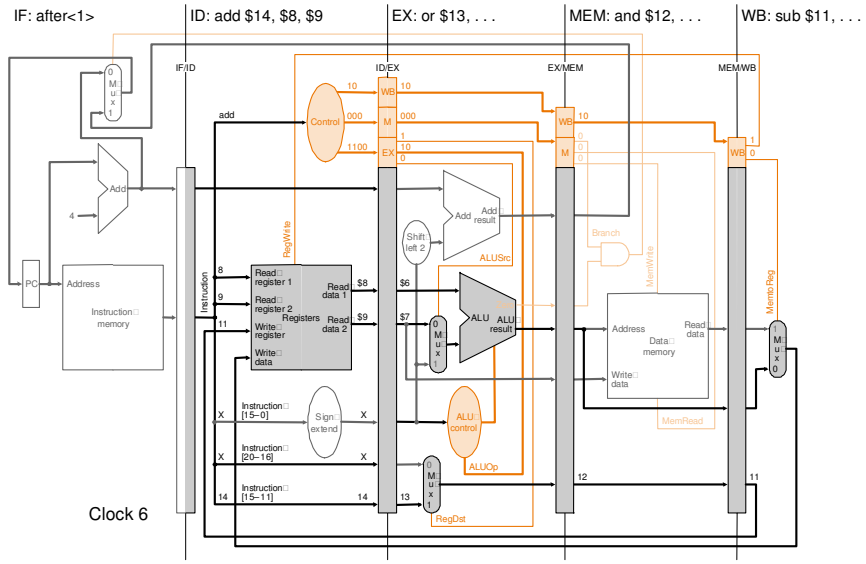
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

5º ciclo



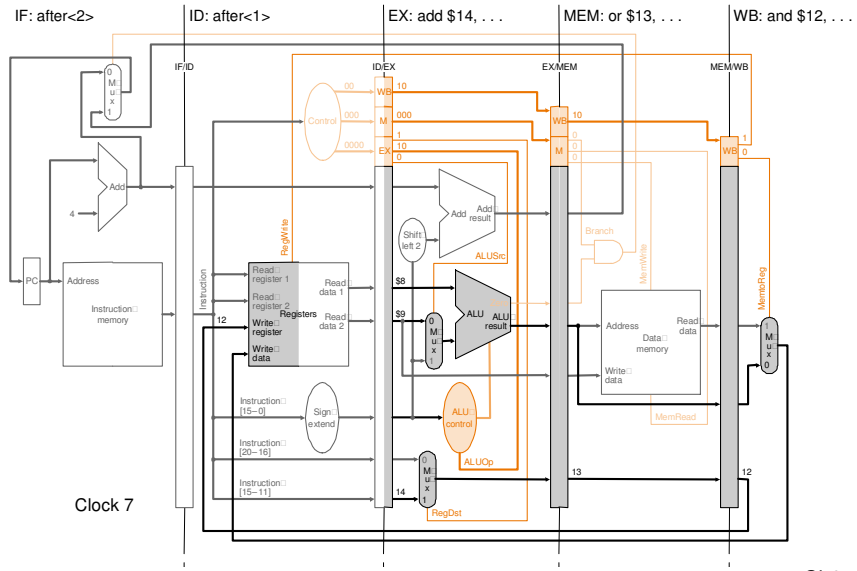
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

6º ciclo



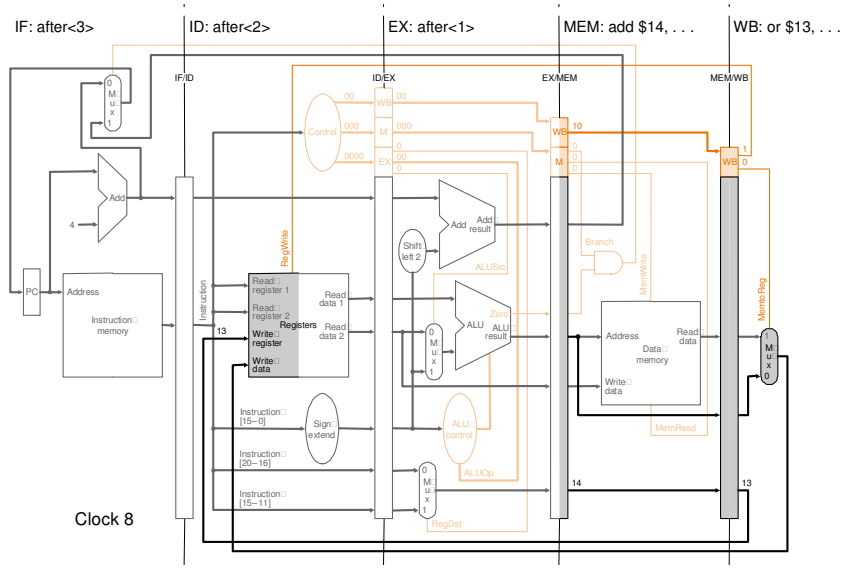
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

7º ciclo



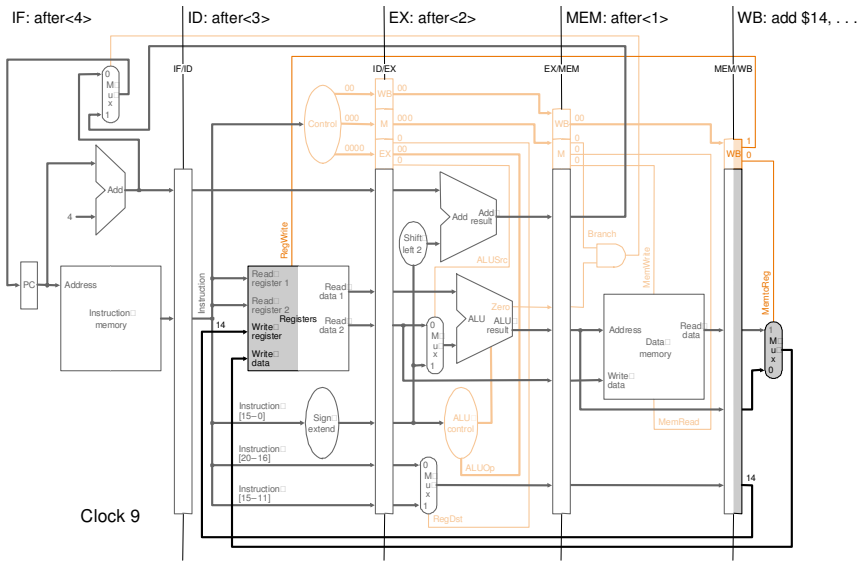
lw $10, 20 ($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9

8º ciclo



```
lw    $10, 20 ($1)
sub   $11, $2,  $3
and   $12, $4,  $5
or    $13, $6,  $7
add   $14, $8,  $9
```

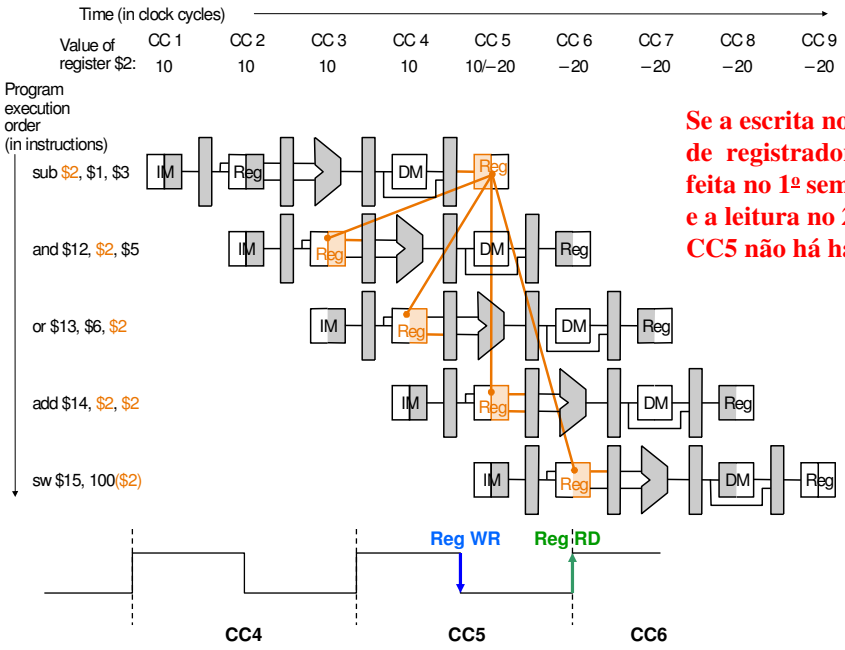
9º ciclo



Dependências

- **Problema com iniciar a execução de uma instrução antes do término da anterior**
  - **Exemplo: Instruções com dependências**

```
sub   $2,  $1, $3     #  reg $2 modificado
and   $12, $2, $5     #  valor (1º operando) de $2
                      #  depende do sub
or    $13, $6, $2     #  idem (2º operando)
add   $14, $2, $2     #  idem (1º e 2º operando)
sw    $15, 100($2)    #  idem (base do endereçamento)
```



Se a escrita no banco de registradores é feita no 1º semi-ciclo e a leitura no 2º, em CC5 não há hazard.

Solução por Software

- **Compilador garante um código sem Hazard inserindo nops**
  - **Onde inserir os nops?**

```
sub   $2,  $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

```
sub   $2,  $1, $3
nop
nop
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

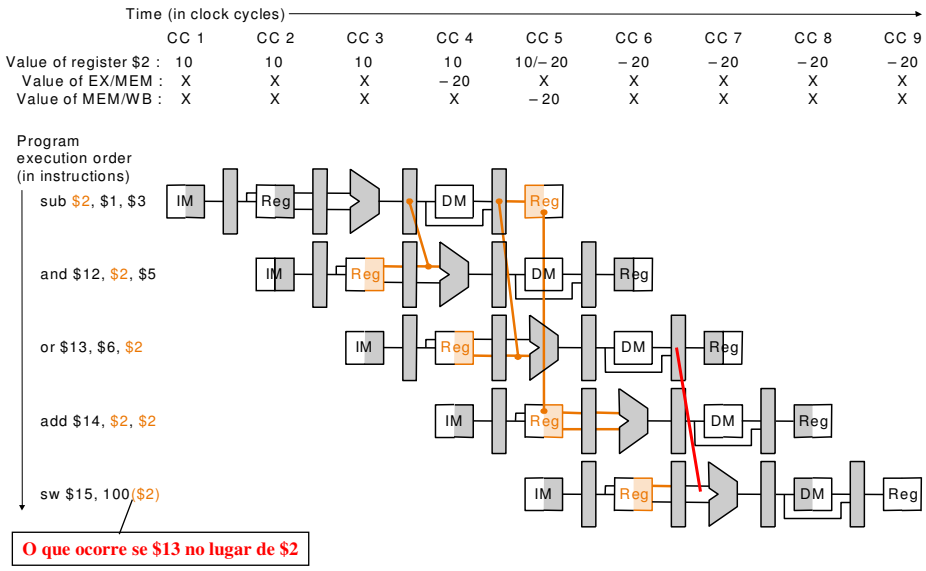
Problema: reduz o desempenho

Solução por Hardware

- **Usar o resultado assim que calculado, não esperar que ele seja escrito.**
- **Neste caso é necessário mecanismo para detectar o hazard. Qual a condição para que haja hazard?**

Quando uma instrução tenta ler um registrador (estágio EX) e esse registrador será escrito por uma instrução anterior no estágio WB

Forwarding



Condições que determinam Hazards de dados

- **EX/MEM**
  - **EX/MEM.RegisterRd = ID/EX.RegisterRs**
  - **EX/MEM.RegisterRd = ID/EX.RegisterRt**
- **MEM/WB**
  - **MEM/WB.RegisterRd = ID/EX.RegisterRs**
  - **MEM/WB.RegisterRd = ID/EX.RegisterRt**

Condições que determinam Hazard de dados

- **Exemplo:**

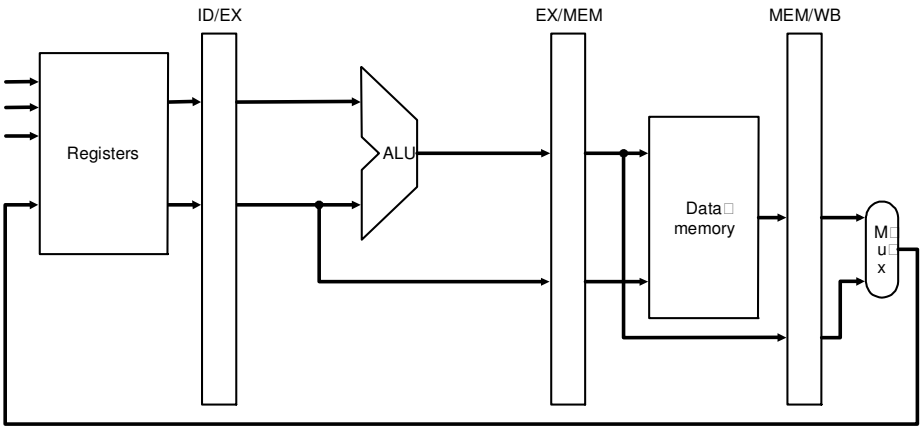
```
sub     $2, $1, $3      #  reg $2 modificado
and     $12,  $2, $5    #  valor de $2  depende do sub
or      $13, $6, $2     #  idem (2º operando)
add     $14,$2,  $2     #  idem (1º e 2º operandos)
sw      $15, 100($2)    #  idem (base do endereçamento)
```

  - **sub–and:** **EX/MEM.RegisterRd = ID/EX.RegisterRs = $2**
  - **sub-or:** **MEM/WB.RegisterRd = ID/EX.RegisterRt = $2**
  - **sub-add: não tem hazard**
  - **sub-sw: não tem hazard**

Condições que determinam Hazard de dados

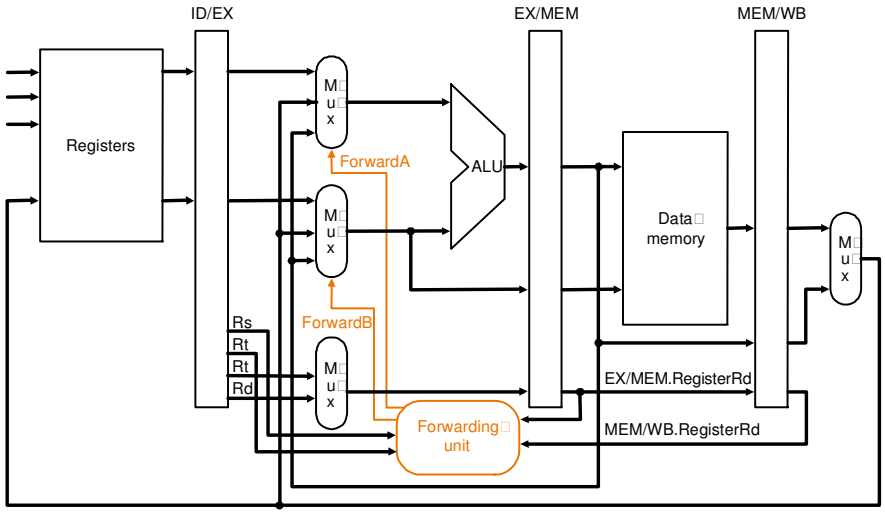
- Esta não é uma política precisa, pois existem instruções que não escrevem no register file.
  - solução seria verificar o sinal RegWrite
- MIPS usa $0 para operandos de valor 0. Para instruções onde $0 é destino?
  - sll $0, $1, 2
    - valor diferente de zero nos regs de pipeline
  - add $3, $0, $2
    - se houver fowarding, $3 terá valor errado no fim da instrução add
- Para isto temos que incluir as condições
  - EX/MEM.registerRd <> 0 (1º hazard)
  - MEM/WB.registerRd <> 0 (2º hazard)

Datapath sem Forwarding



a. No forwarding

Datapath com Forwarding (para add, sub, and e or)



b. With forwarding

Valores dos sinais ForwardA e ForwardB

Mux Control	Source	Explanação
ForwardA = 00	ID/EX	Primeiro operando da ULA ➔ register file
ForwardA = 10	EX/MEM	Primeiro operando da ULA ➔ resultado anteiror da ULA
ForwardA = 01	MEM/WB	Primeiro operando da ULA é antecipado da memória de dados ou um resultado anterior da ULA
ForwardB = 00	ID/EX	Segundo operando da ULA ➔ register file
ForwardB = 10	EX/MEM	Segundo operando da ULA ➔ resultado anteiror da ULA
ForwardB = 01	MEM/WB	Segundo operando da ULA é antecipado da memória de dados ou um resultado anterior da ULA

Detecção e Controle de hazard

- **O controle de fowarding será no estágio EX, pois é neste estágio que se encontram os multiplexadores de fowarding da ULA. Portanto devemos passar o número do registrador operando do estágio ID via registrador de pipeline ID/EX ➔ adicionar campo rs (bits 25-21).**

Condições para detecção de hazard

EX hazard

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd <> 0 )
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      FowardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd <> 0 )
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      FowardB = 10
```

MEM hazard

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd <> 0 )
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      FowardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd <> 0 )
   and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      FowardB = 01
```

Observações:

- **Não existe data hazard no estágio WB, pois estamos assumindo que o register file supre o resultado correto se a instrução no estágio ID é o mesmo registrador escrito pela instrução no estágio WB ➔ forwarding no register file**
  - **Na 1ª edição do livro P&H tem hazard**

Observações:

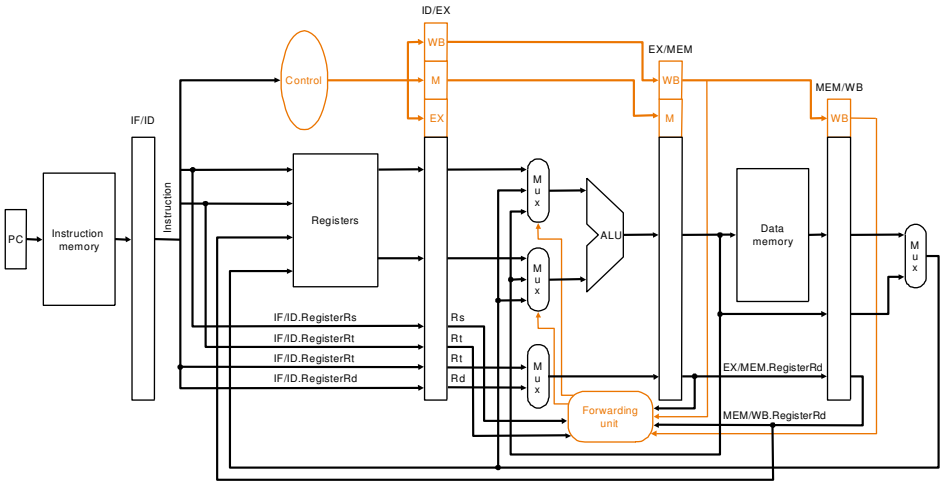
- **Um data hazard potencial pode ocorrer entre o resultado de uma instrução em WB, o resultado de uma instrução em MEM e o operando fonte da instrução no estágio da ULA**

```
add   $1, $1, $2
add   $1, $1, $3
add   $1, $1, $4
```

- **Neste caso o resultado é antecipado do estágio MEM porque o resultado neste estágio é mais recente**

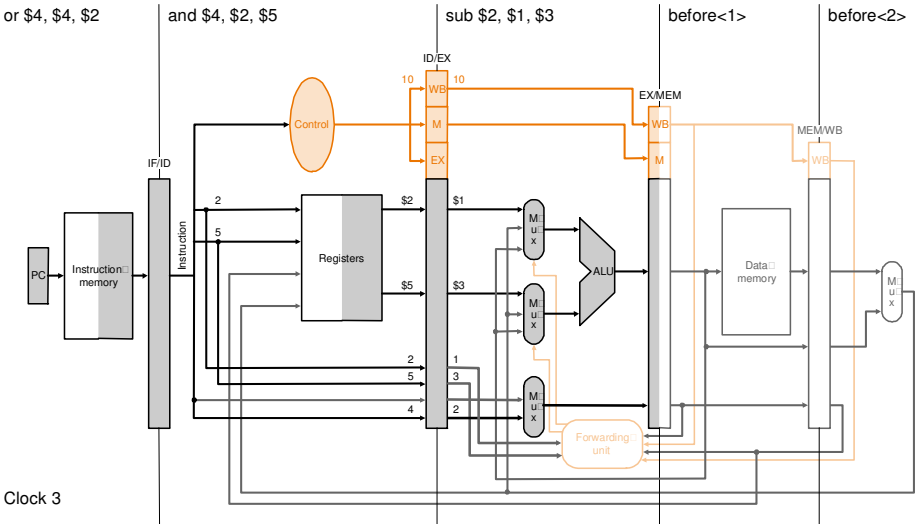
```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd <> 0 )
and (EX/MEM.RegisterRd <> ID/EX.registerRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) FowardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd <> 0 )
and (EX/MEM.RegisterRd <> ID/EX.registerRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) FowardB = 01
```

Datapath modificado para fowarding



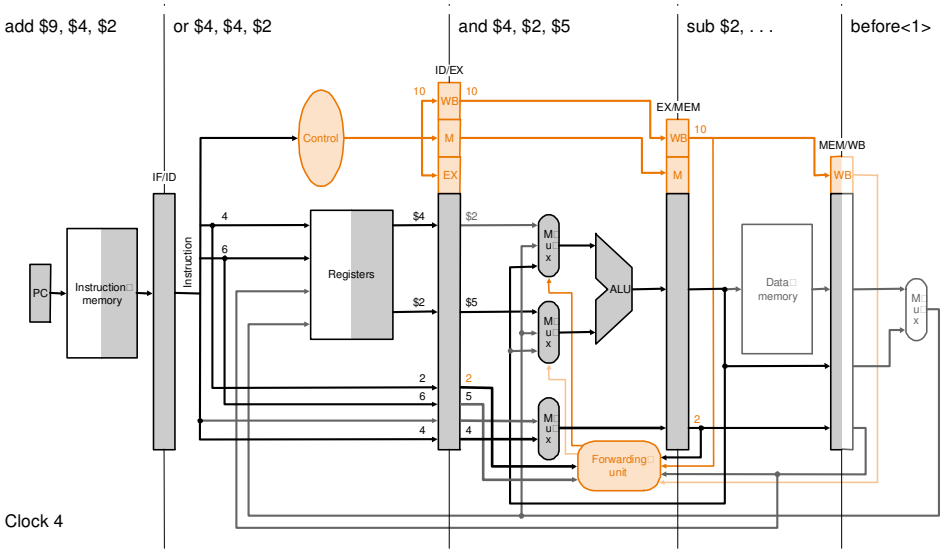
Forwarding

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2



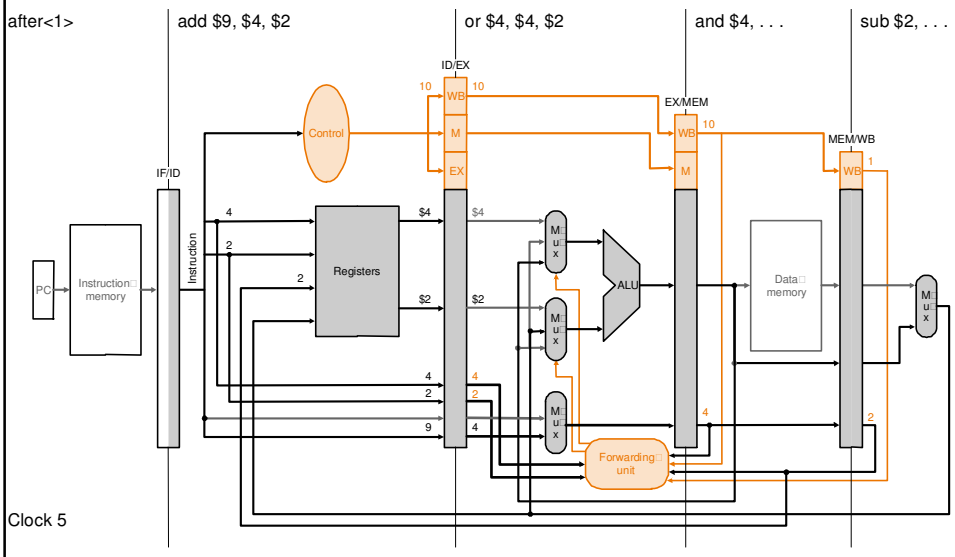
Forwarding

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2



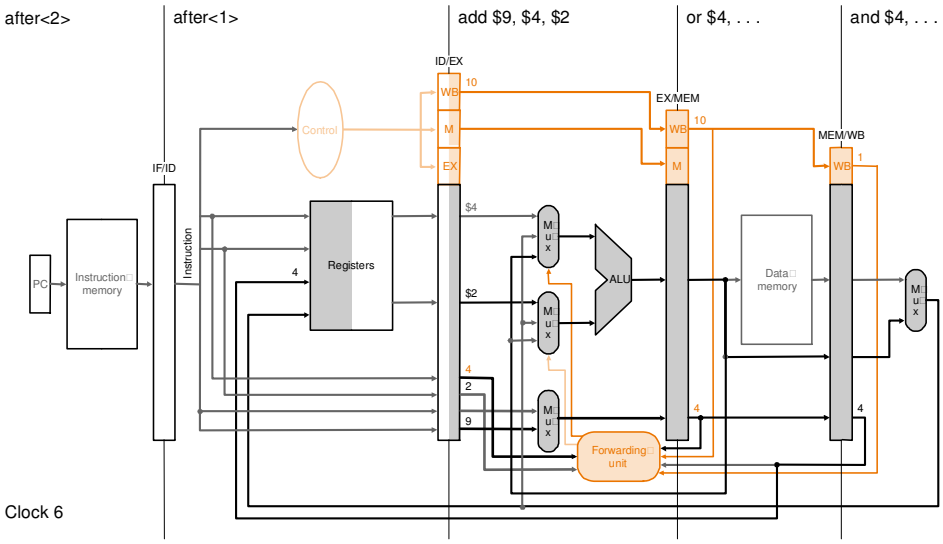
Forwarding

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

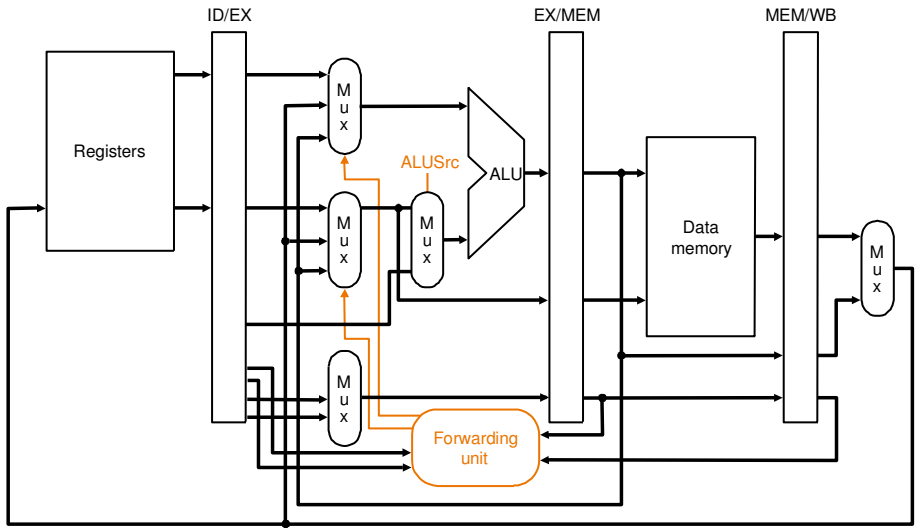


Forwarding

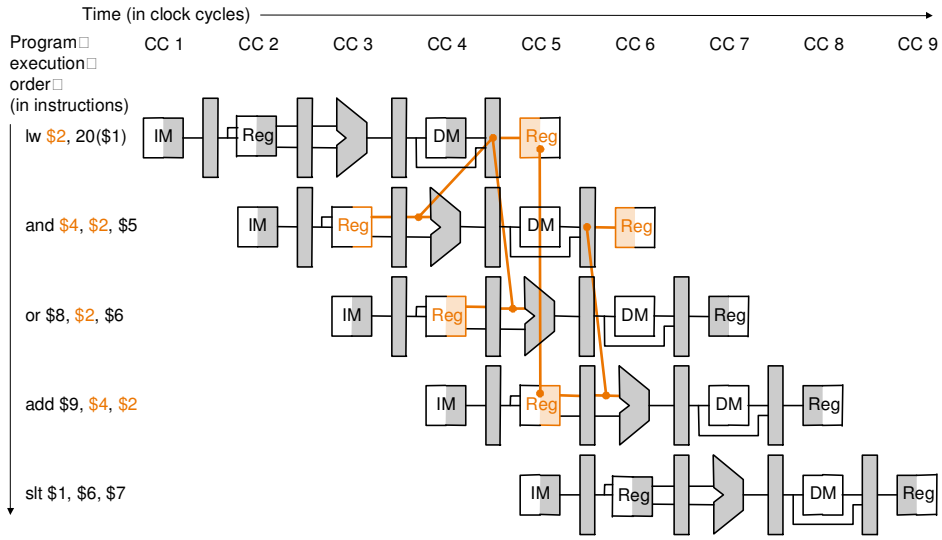
sub	**$2, $1, $3**
and	**$4, $2, $5**
or	**$4, $4, $2**
add	**$9, $4, $2**



Forwarding: Datapath para entrada signed-imediate necessária para lw e sw



Data Hazard e Stalls



Data Hazards e Stalls

- **Quando uma instrução tenta ler um registrador precedida por uma instrução de load, que escreve no mesmo registrador ➔ o dado tem que ser mantido (ciclo 4) enquanto a ULA executa a operação ➔ atrasar o pipeline para que a instrução leia o valor correto.**

- **Condição de detecção de hazard para atraso no pipeline**

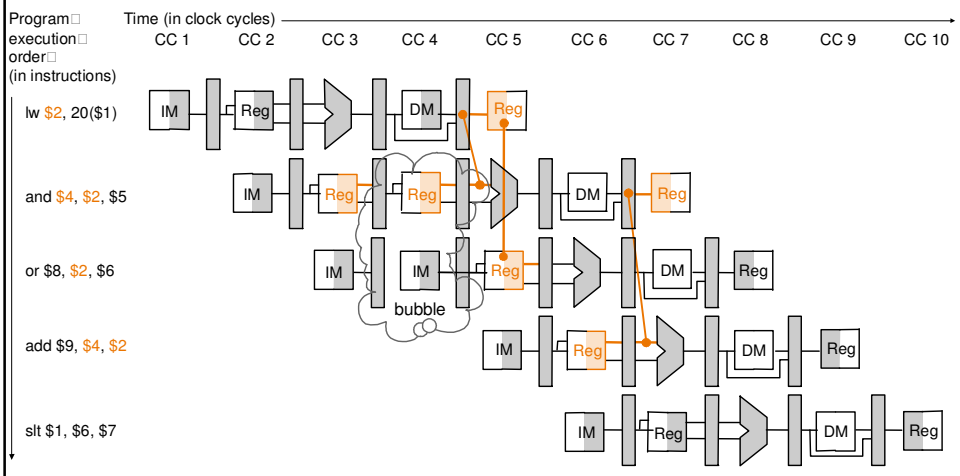
  # testa se é um load:

  if (ID/EX.MemRead and

  # verifica se registrador destino da instrução load em EX é o registrador fonte da instrução em ID:

  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
  stall pipeline

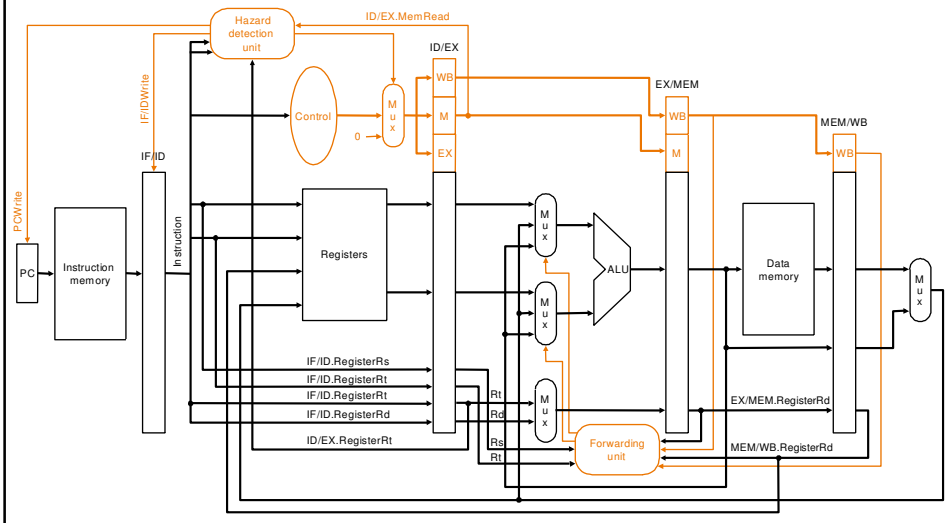
Data Hazards e Stalls



Data Hazards e Stalls

- **Se a instrução no estágio ID é atrasada, a instrução no estágio IF também deve ser atrasada**
- **Como fazer?**
  Impedir a mudança no PC e no registrador IF/IF. A instrução em IF continua sendo lida e em ID continuam sendo lido os mesmos campos da instrução.
- **Stall pipeline: mesmo efeito da instrução nop começando pelo estágio EX ➔ desativar os 9 sinais de controle dos estágios EX, MEM e WB**

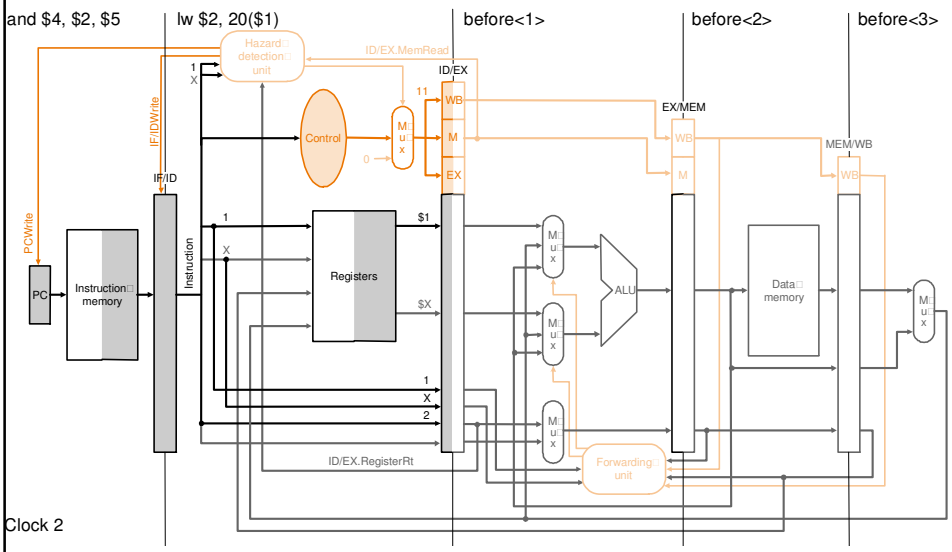
Datapath com forwarding e data hazard detection



Paulo C. Centoducatte

Ch6.a-81

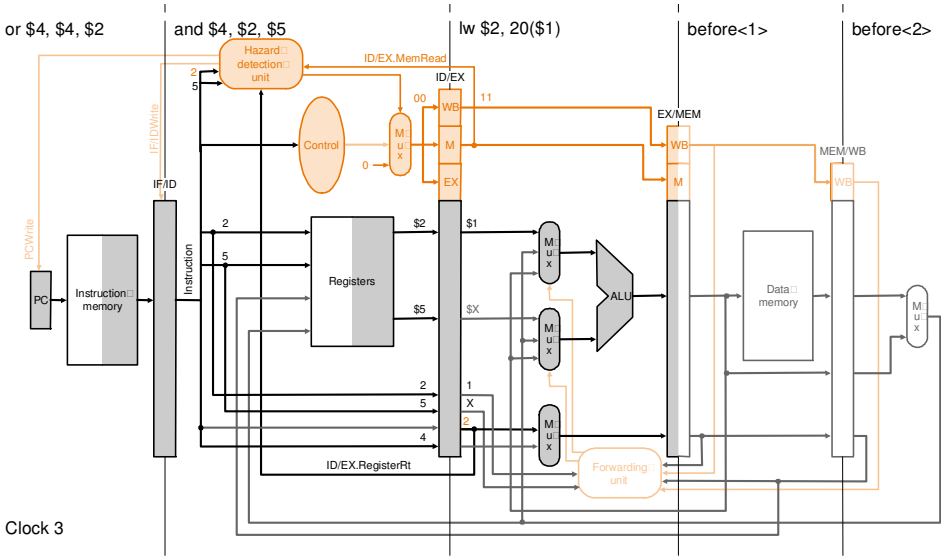
Seqüência de execução do exemplo anterior



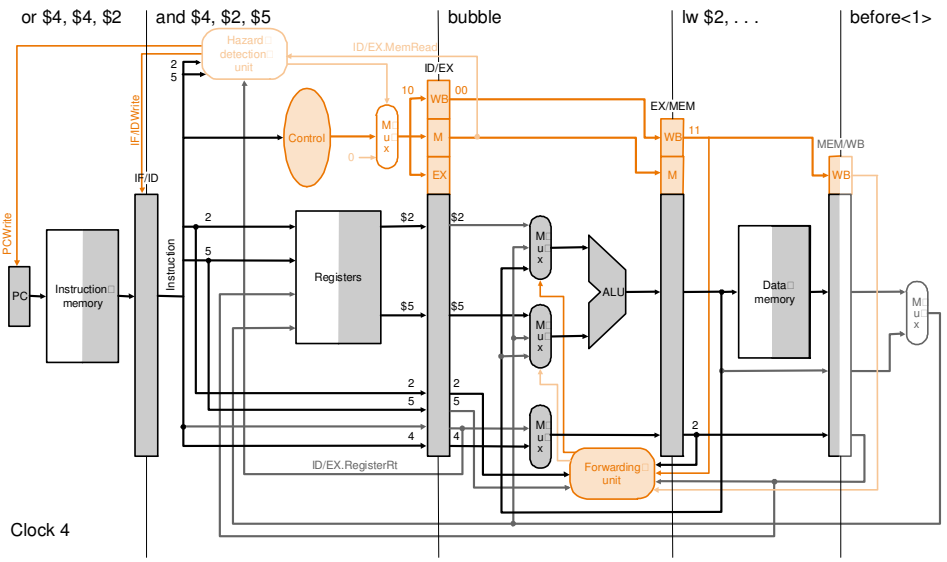
Paulo C. Centoducatte

Ch6.a-82

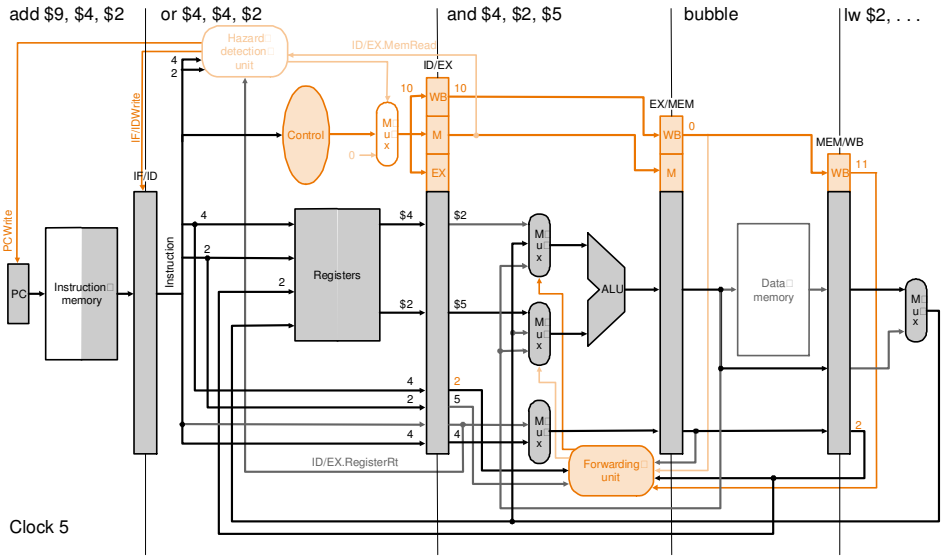
Seqüência de execução do exemplo anterior



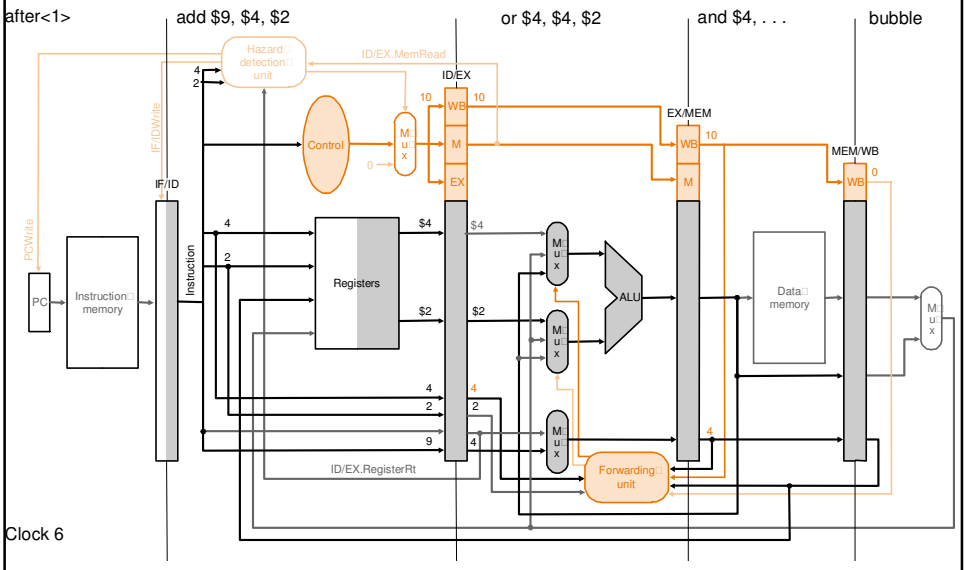
Seqüência de execução do exemplo anterior



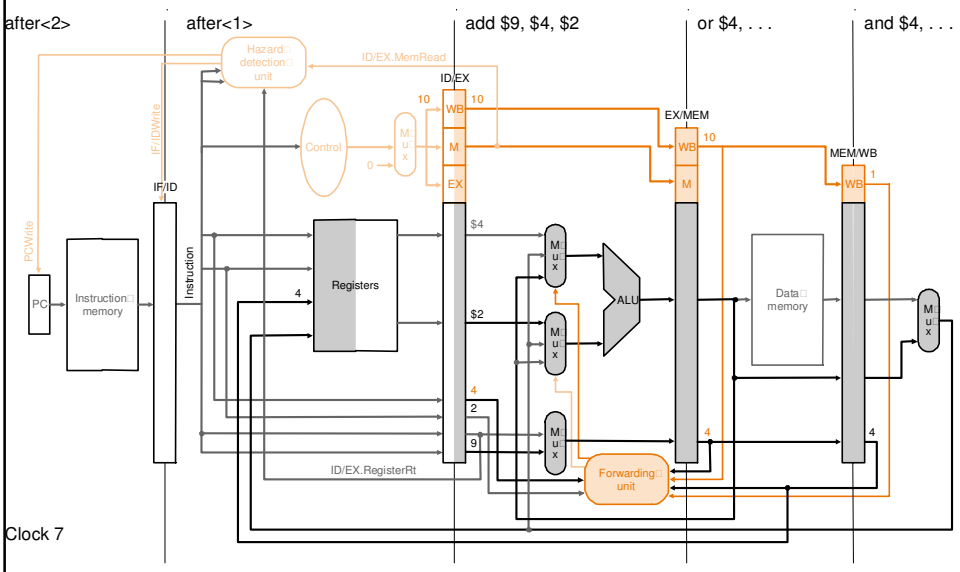
Seqüência de execução do exemplo anterior



Seqüência de execução do exemplo anterior

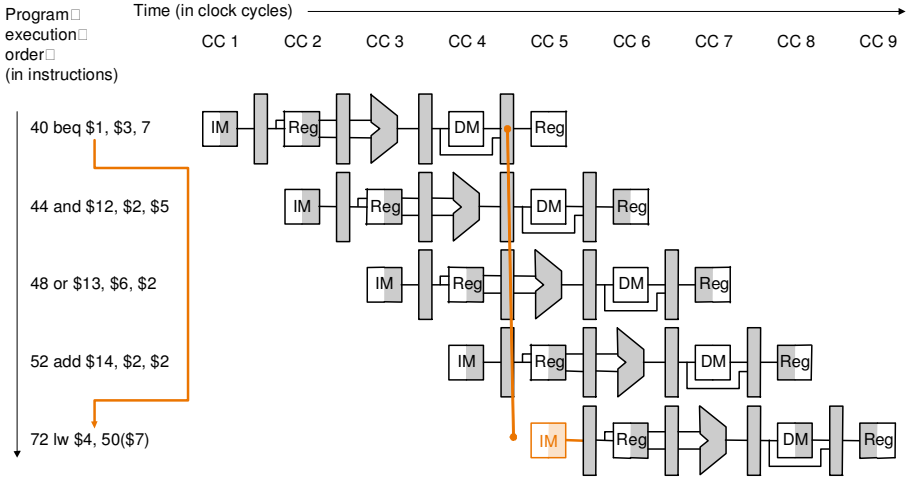


Seqüência de execução do exemplo anterior



Branch Hazards

- **Devemos ter um fetch de instrução por ciclo de clock, decisão: qual caminho de um branch deve ocorrer até o estágio MEM.**



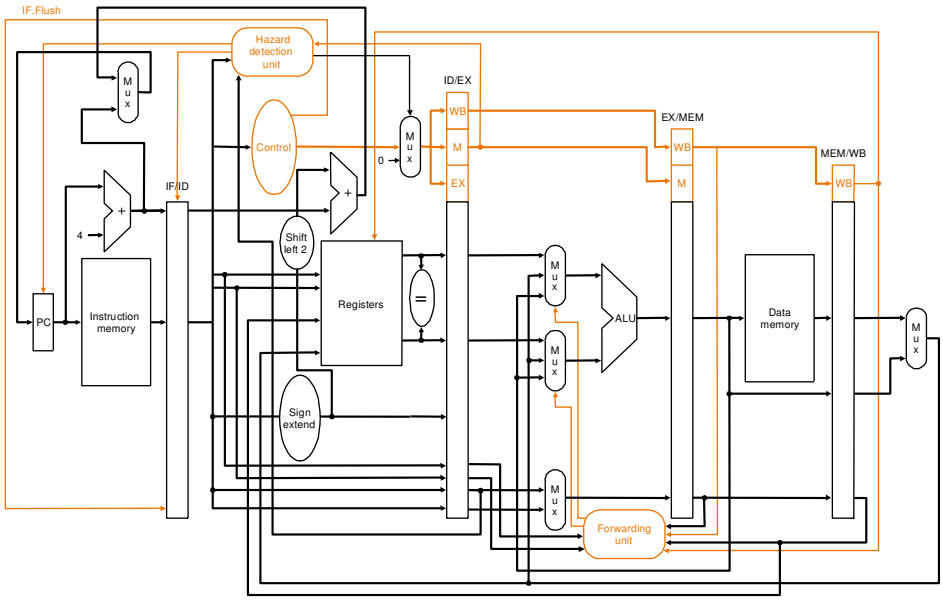
Branch Hazards

- **Tês esquemas para resolver control hazard :**
  - **Branch-delay Slots**
  - **Assume Branch Not Taken**
  - **Dynamic Branch Prediction**
- **Assume Branch Not Taken**
  - **continua a execução seqüencialmente e se o branch for tomado, descarta as instruções entre a instrução de branch e a instrução no endereço alvo, fazendo seus sinais de controle iguais a zero**

Branch Hazards

- **Redução do atraso de branches**
  - **reduzir o custo se o branch for tomado**
  - **adiantar a execução da instrução de branch.**
  - **O next PC para uma instrução de branch é selecionado no estágio MEM**
  - **executar o branch no estágio ID (apenas uma instrução será descartada)**
  - **deslocar o cálculo do endereço de branch (branch adder) do MEM para o ID e comparando os registradores lidos do register file.**

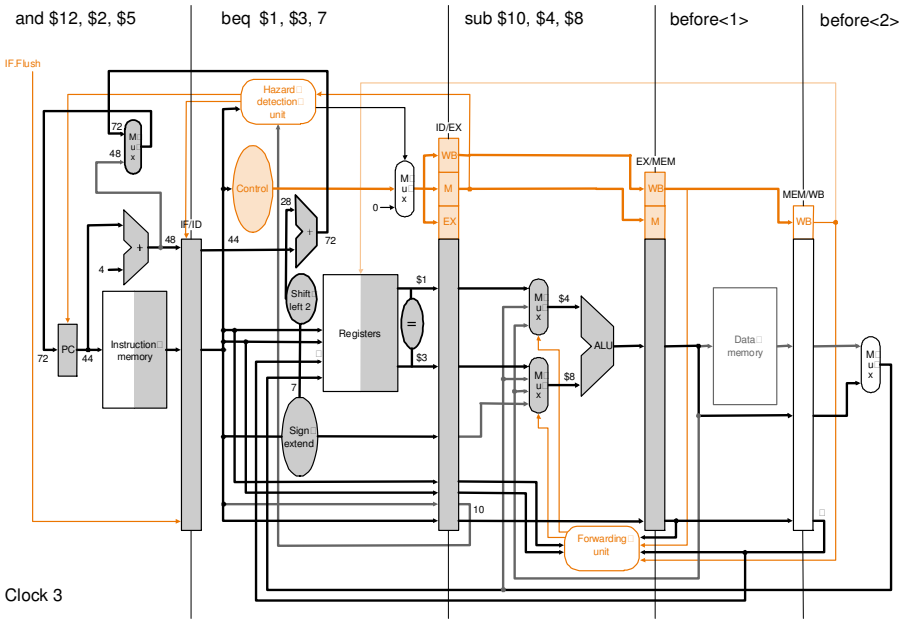
Branch Hazards



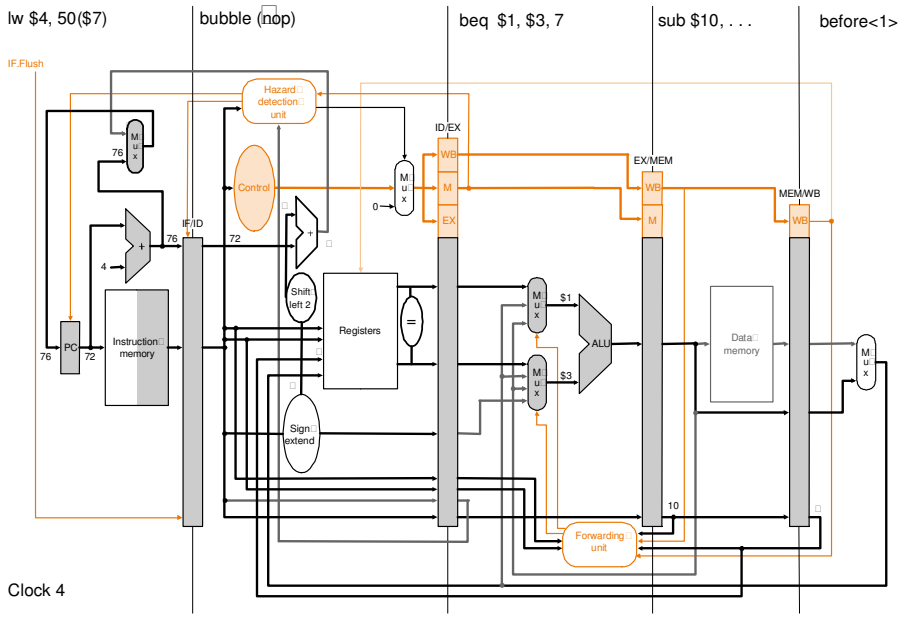
Branch Hazards (exemplo)

```
36  sub  $10, $4, $8
40  beq  $1,  $3, 7      # PC ← 40 + 4 +7*4 = 72
44  and  $12, $2, $5
48  or   $13, $2, $6
52  add  $14, $4, $2
56  slt  $15, $6, $7
....       ............
....       ............
72  lw  $4,     50($7)
```

Branch Hazards (exemplo)



Branch Hazards (exemplo)



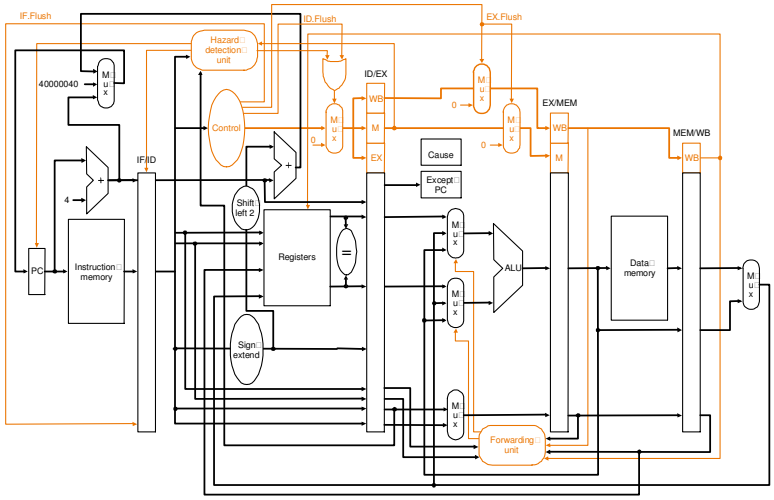
Dymanic Branch Prediction

- **Branch not taken**
  - **forma de branch predicton que assume que o branch não será tomado.**
- **Dymanic Branch Prediction**
  - **descobre se o branch foi tomado ou não na última vez que foi executado e faz o fetch das instruções pelo mesmo local da última vez.**

Dymanic Branch Prediction

- **Implementação**
  - **branch prediction buffer**
  - **branch history table**
- **Branch prediction buffer: pequena memória indexada por bits menos significativos do endereço da instrução de branch. Ela contém um bit que diz se o branch foi recentemente tomado ou não (Neste esquema não sabemos se a previsão é correta ou não, pois este buffer pode ser alterado por outra instrução de branch que tem os mesmos bits menos significativos de endereço).**

Tratamento de Exceção



Tratamento de Exceção

Exemplo:

Dado a seqüência abaixo:

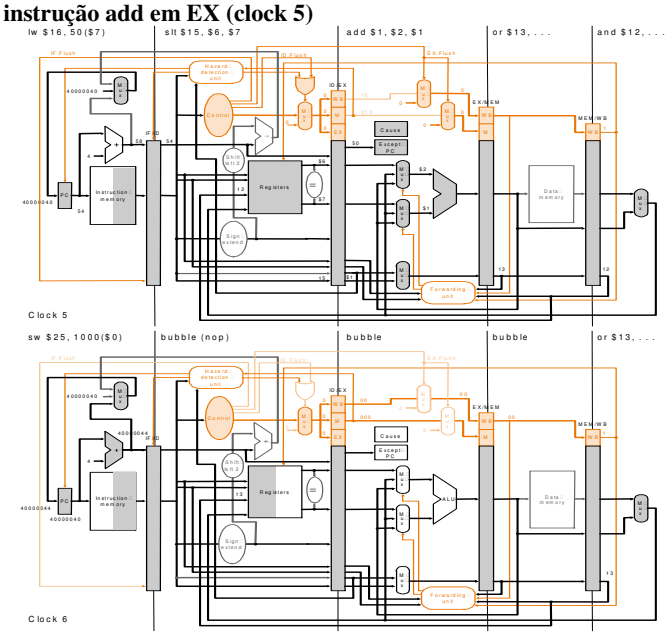
40hex	**sub**	**$11,**	**$2,**	**$4**
44hex	**and**	**$12,**	**$2,**	**$5**
48hex	**or**	**$13,**	**$2,**	**$6**
4Chex	**add**	**$1,**	**$2,**	**$1**
50hex	**slt**	**$15,**	**$6,**	**$7**
54hex	**lw**	**$16,**	**50($7)**	

Assumir que as instruções a serem chamadas em um tratamento de exceção comecem com:

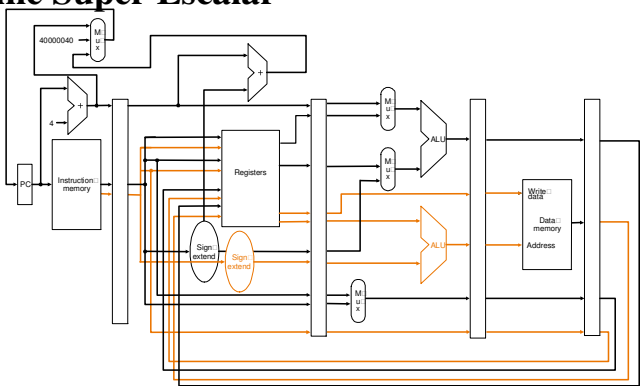
40000040hex	**sw**	**$25,**	**1000($0)**
40000044hex	**sw**	**$26,**	**1004($0)**

Mostre o que acontece no pipeline se uma exceção de overflow ocorre na instrução add.

Tratamento de Exceção - A figura abaixo mostra o que acontece a partir da instrução add em EX (clock 5)



Pipeline Super Escalar



Tipo da instrução	Estágio do pipeline							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Pipeline Super Escalar

Exemplo:

Como o loop abaixo será escalonado em um MIPS superscalar:

```
Loop:   lw    $t0, $0($s2)        # t0= elemento do array
        addu  $t0, $t0, $t2       # add escalar em $s2
        sw    $t0, 0($s2)         # armazena o resultado
        addi  $s1, $s1,-4         # decrementa o ponteiro
        bne   $s1, $zero, Loop    # branch se $s1 != 0
```

Reordenar as instruções para evitar tantos pipelines stalls quanto possível

Pipeline Super Escalar

Solução:
A primeira com a terceira e as duas últimas instruções tem dependência de dados.

	Instruções ALU ou branchs	Instr. data transfer	Ciclo de clock
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

Pipeline Super Escalar

Loop unrolling para pipelines escalares ➔ técnica para aumentar o desempenho para loops que acessam arrays ➔ múltiplas cópias do corpo do loop são feitas e instruções de diferentes iterações são escalonadas juntas

Exemplo ➔ supor exemplo anterior

Solução ➔ 4 cópias do corpo do loop

	Instruções ALU ou branchs	Instr. data transfer	Ciclo de clock
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		lw $t1, 12($s1)	2
	addu $t0,$t0,$s2	lw $t2, 8($s1)	3
	addu $t1,$t1,$s2	lw $t3, 4($s1)	4
	addu $t2,$t2,$s2	sw $t0, 0($s1)	5
	addu $t3,$t3,$s2	sw $t1, 12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	9

Datapath final