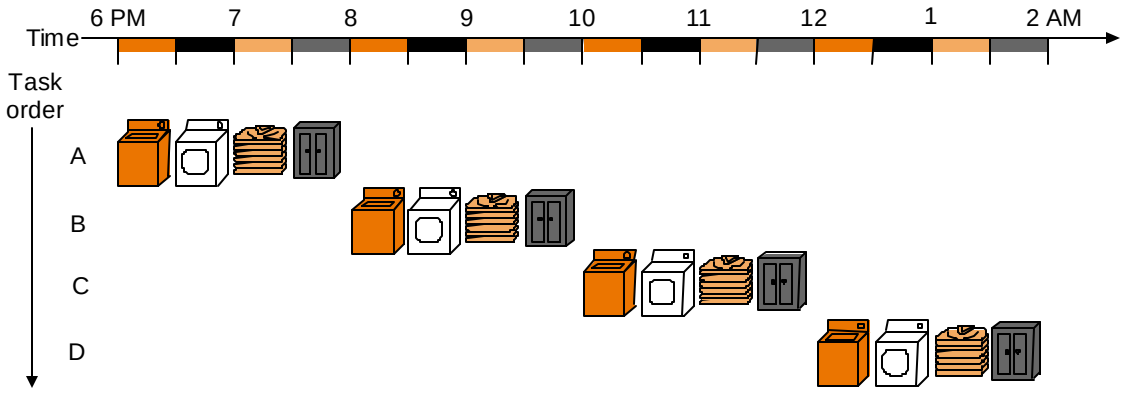
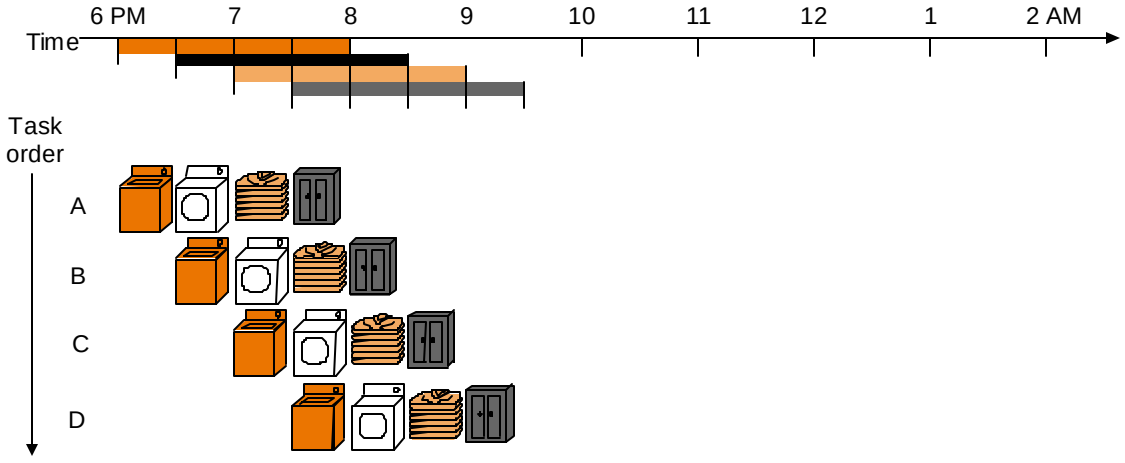
6. Pipelining

- **Overwiew**

- **Figura 6.1 - Lavanderia – analogia com o pipelining**





- **Pipeline de instruções no MIPS**

  1. **Fetch da instrução**

  2. **Leitura do registradores, decodificação**

  3. **Execução da operação ou cálculo de um endereço**

  4. **Acesso ao operando na memória**

  5. **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 a register file = 1ns. (Instrs lw, sw, add, sub, and, or slt e beq).

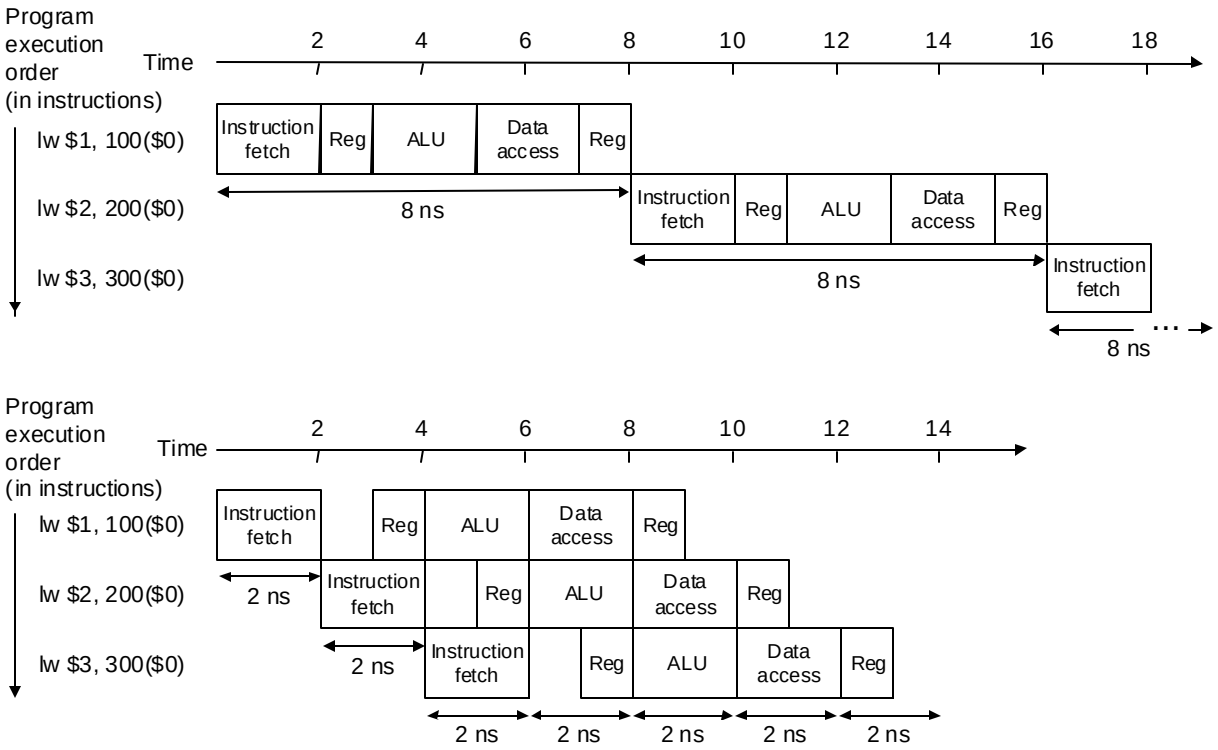
  Solução: Pior caso ➔instruções de lw (supor 3 instruções de lw – tempo entre o início da 1ª instrução e o início da 4ª instrução)

  $\text{Tempo entre instruções}_{\text{pipelined}}$ =
  $\text{Tempo entre instruções}_{\text{nonpipelined}}$ / número de estágios do pipeline

- **Figura 6.2 – 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

- **Figura 6.3 – Execução não-pipeline X pipeline**



- **Sob condições normais, com estágios balanceados, o speedup do pipeline é igual ao número de estágios do pipeline (5 estágios , 5 vezes mais rápido)**

- **Na realidade é um pouco superior (overheads) ➔ speedup é menor que o número de estágios do pipeline**

  - **Suponha 1003 instruções**

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

    - **sem pipeline ➔ 1000 X 8ns + 24 = 8024**

    - **sppedup = 8024 / 2014 = 3.98 ~~ 8 / 2**

- **Projeto de um conjunto de instruções para pipeline**

  - **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 em instruções lw e sw.**

  - **Operandos devem estar alinhados na memória ➔ o dado pode ser transferido memória ➔ CPU e CPU ➔ memória em um único estágio do pipeline.**

- **Pipeline Hazards**

  São três as situações em que a próxima instrução não pode ser executada no próximo ciclo de clock (hazards):

  - **Structural Hazards**

    O hardware não suporta uma combinação de instruções que queremos executar em um único período de clock (p. ex., escrever e ler da memória em um mesmo ciclo) ➔ não ocorre no MIPS, pois suas instruções foram projetadas para tal.

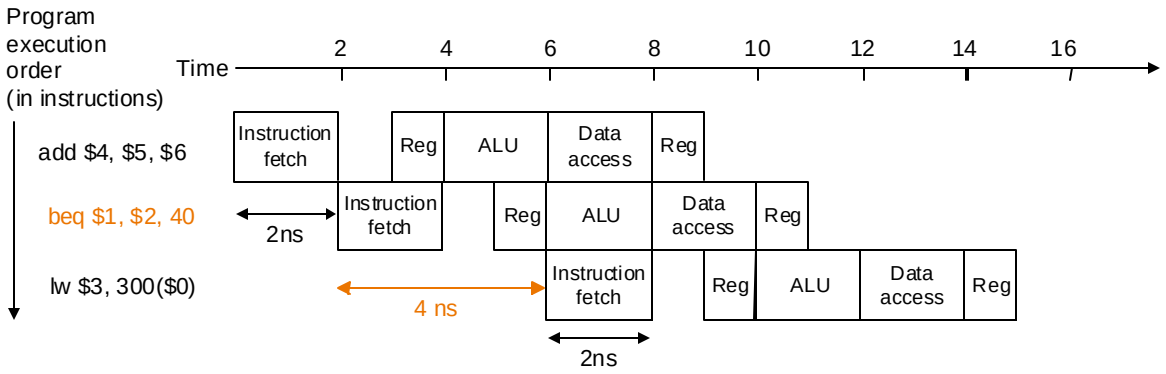
  - **Control Hazards**

    Necessidade de tomar uma decisão que dependa do resultado de uma outra instrução anterior.

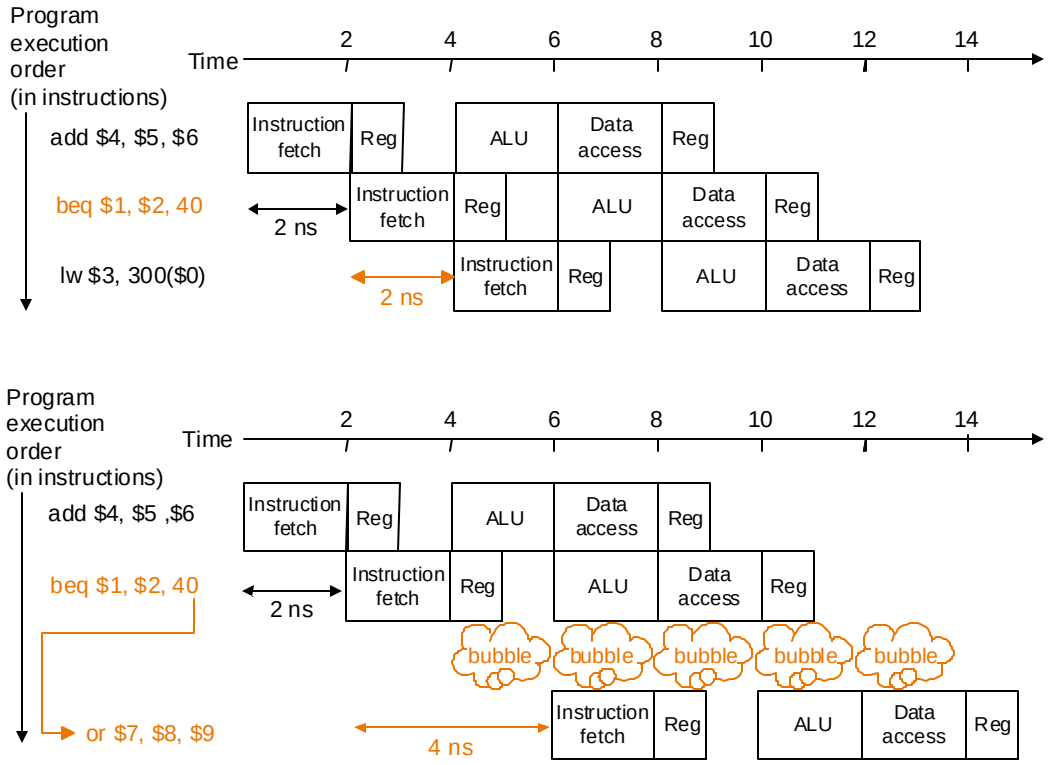
    Soluções:

    - **Pipeline stall ➔ atraso o estágio até o resultado estar disponível.**

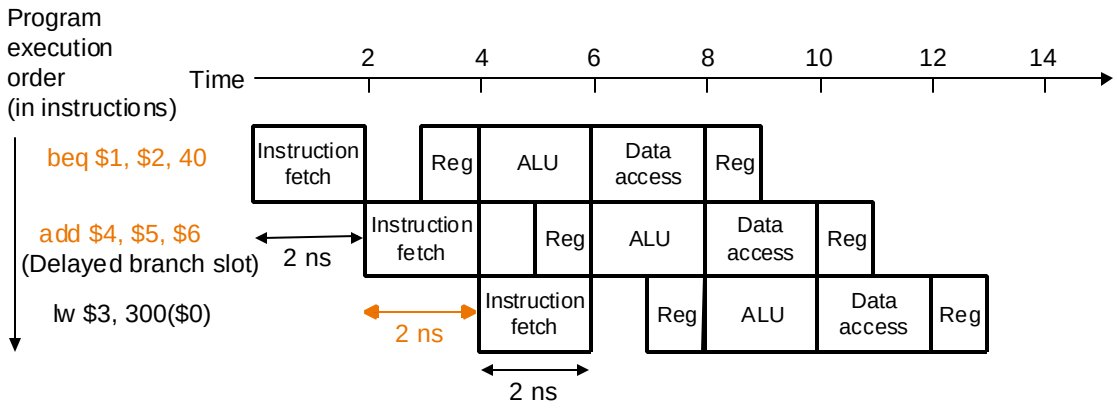
- **Figura 6.4 – Pipelining stalling para instruções branch.**



- **Figura 6.5 - Branch prediction ➔Tentar "adivinhar" qual dos caminhos do branch será tomado.**



- **Figura 6.6 - Delayed Branch ➔ "Adiantar" a instrução de branch de tal maneira a "adiantar" a avaliação da condição.**

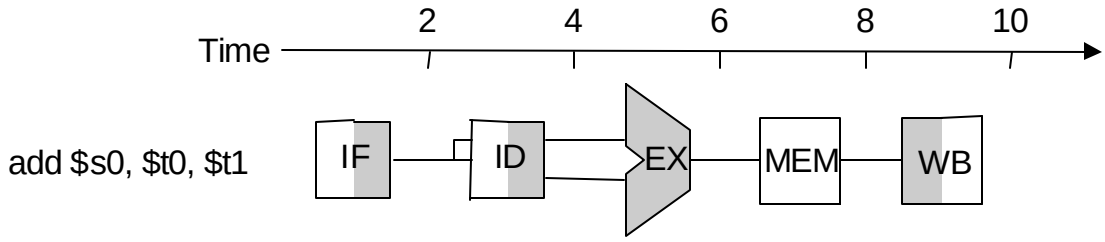


- **Data Hazards ➔ Quando uma instrução necessita de um dado que ainda não foi calculado.**

add $s0,$t0,$t1
sub $t2,$s0,$t3

Solução ➔ Forwarding ou bypassing

- **Figura 6.7 – Representação gráfica de pipeline**



- **Figura 6.8 – Representação Gráfica de Forwarding**

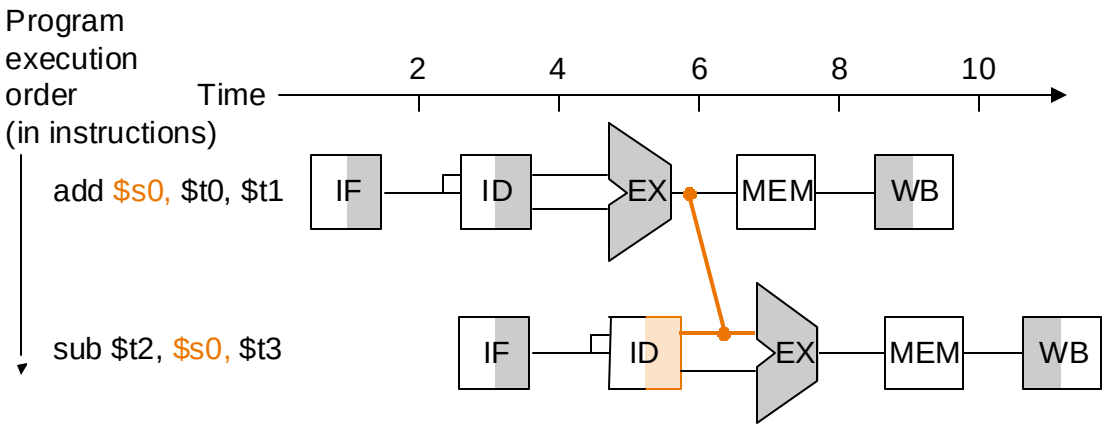
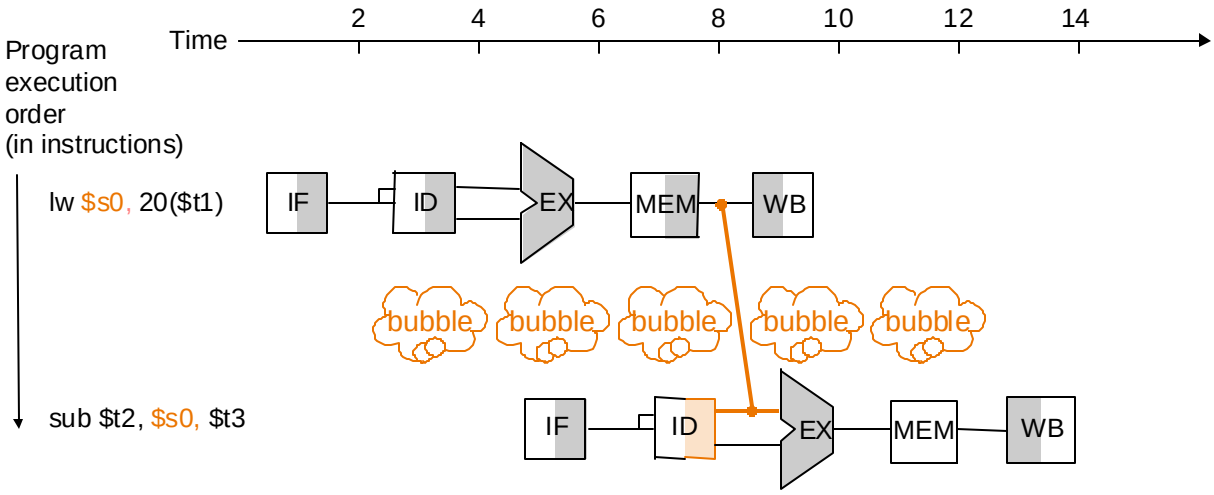


Figura 6.8 - Representação Gráfica de Forwarding com stall para instruções do tipo R



- **Exemplo**

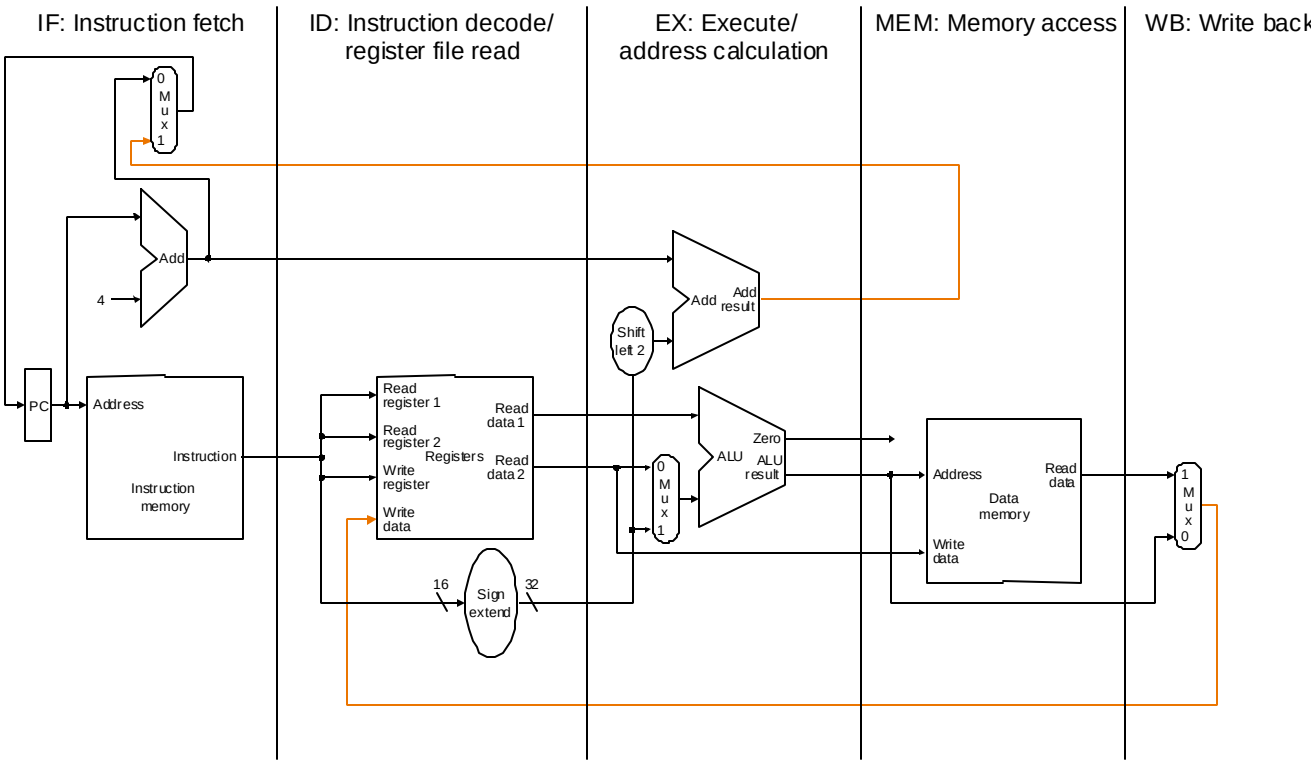
  Encontre o hazard no código abaixo:

```
                          #  $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
```

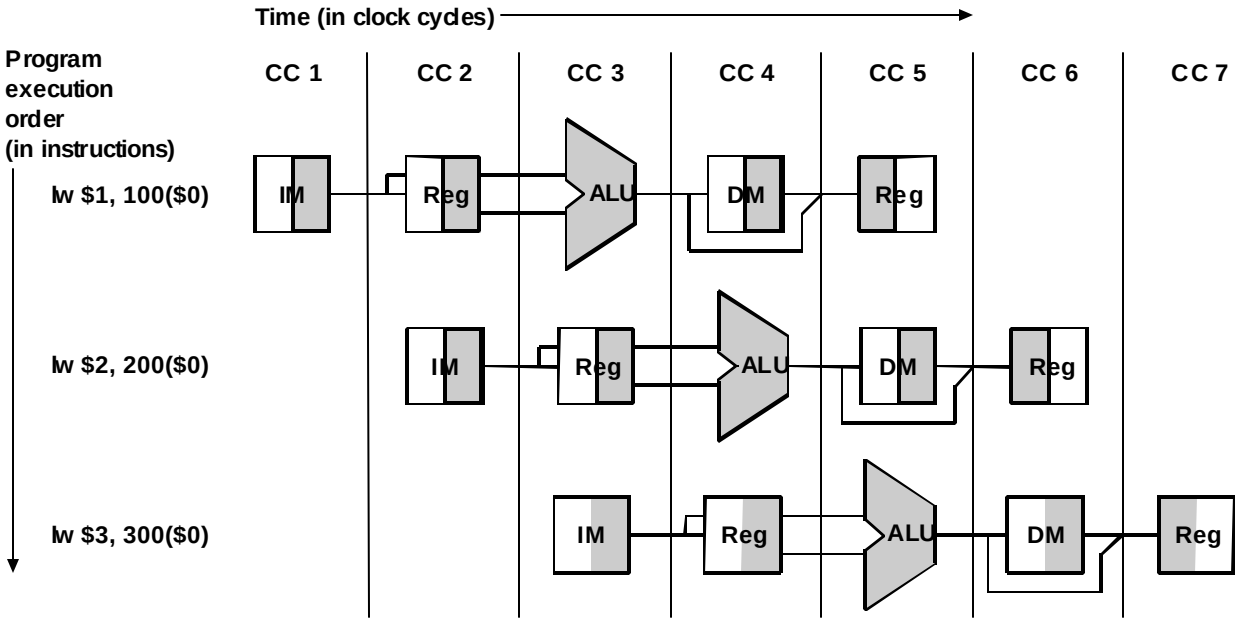
  Solução:

```
                          #  $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
```

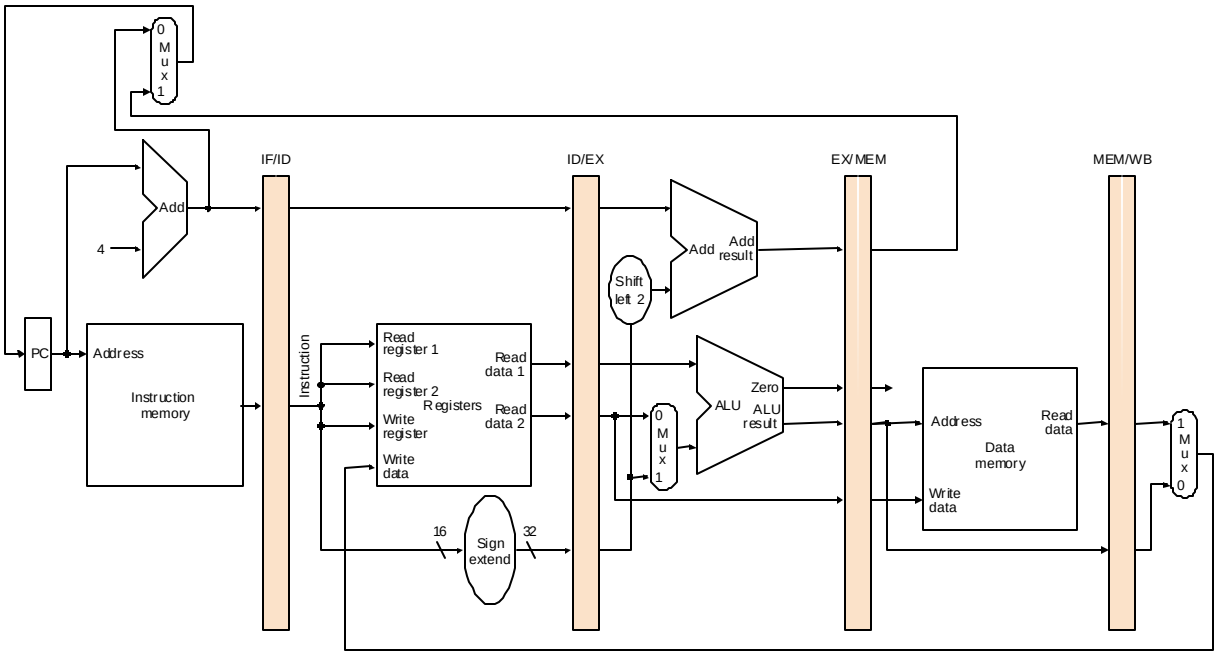
- **Pipeline Datapath**
  - **Datapath ➔ cinco estágios**
    1. **IF: Instruction fetch**
    2. **ID: Instruction decoder e register file read**
    3. **EX: execution ou address calculation**
    4. **MEM: Data memory access**
    5. **Write Back**
- **Figura 6.10 – Datapath single cycle**



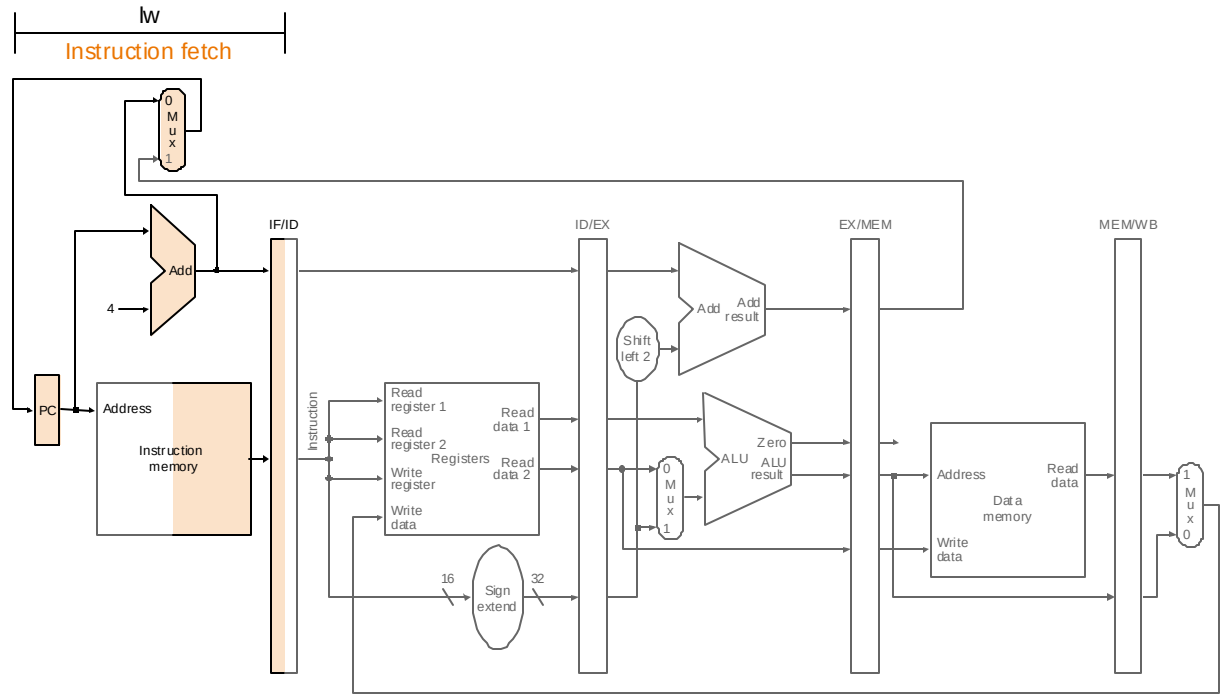
- **Figura 6.11 – Instruções sendo executadas pelo datapath single cycle**

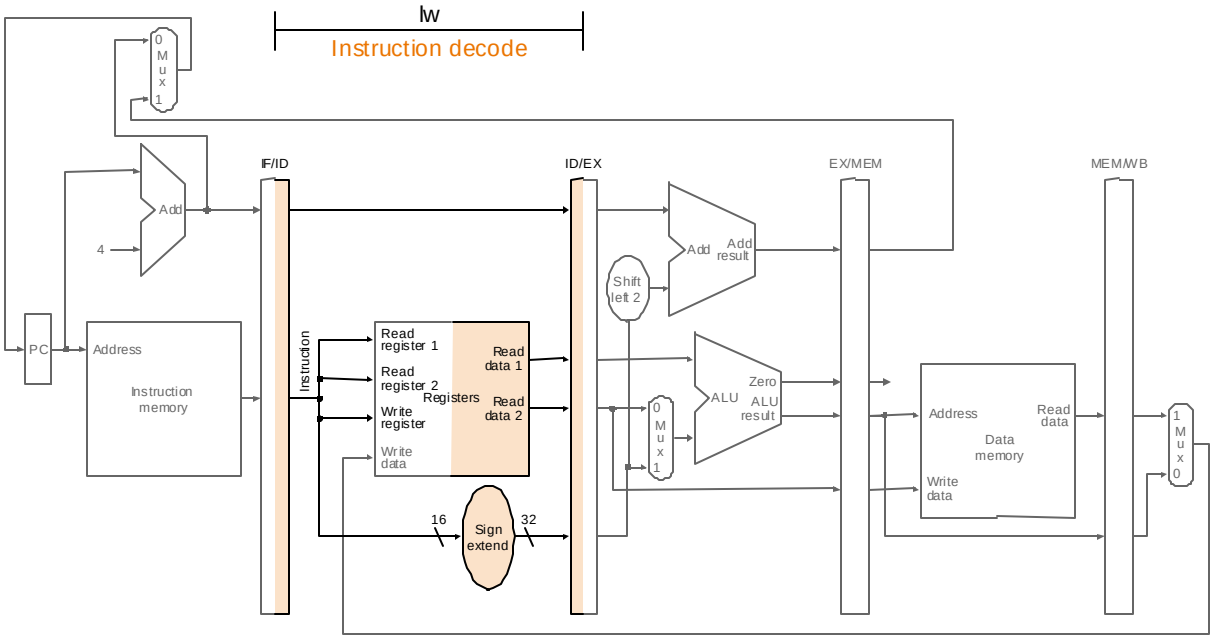


- **Figura 6.12 –Versão do datapath anterior com pipeline**

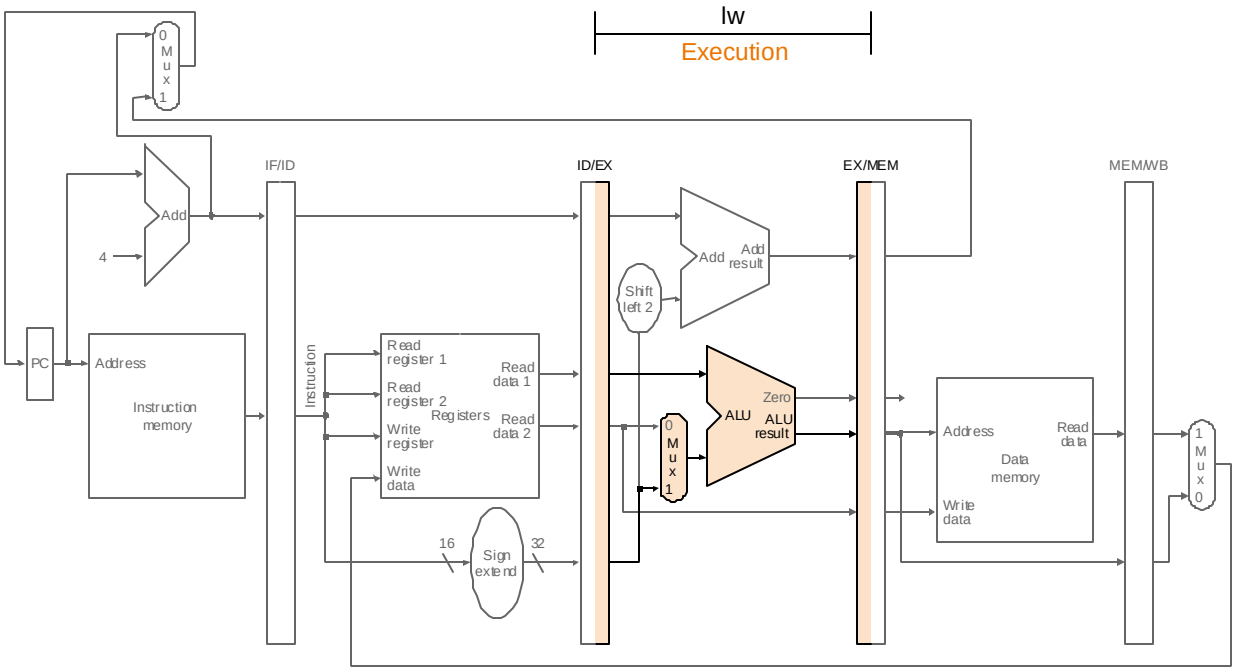


- **Figura 6.13 – Estágio IF e ID – (fetch e decodificação/leitura do register file – instruções lw & sw)**

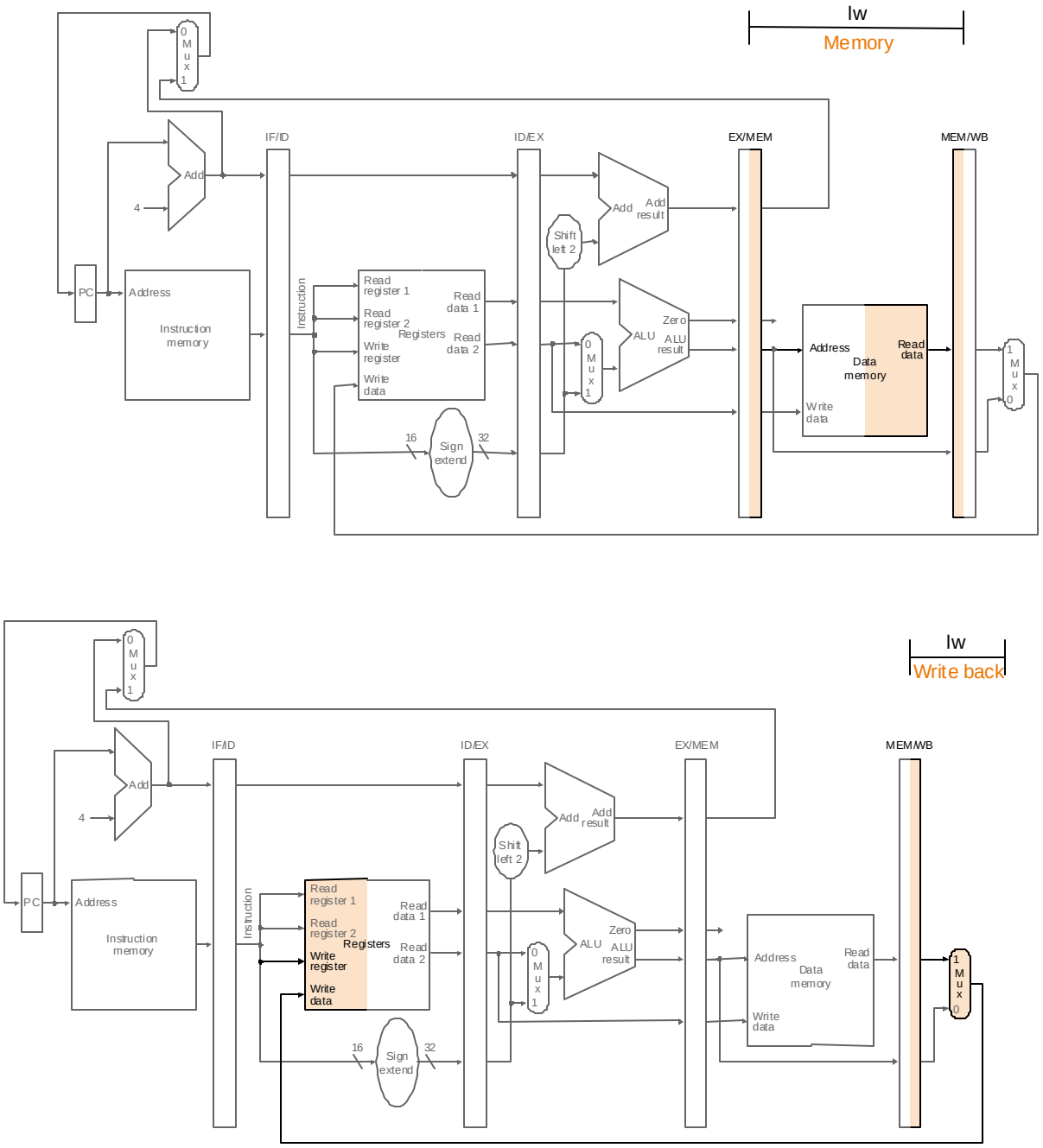




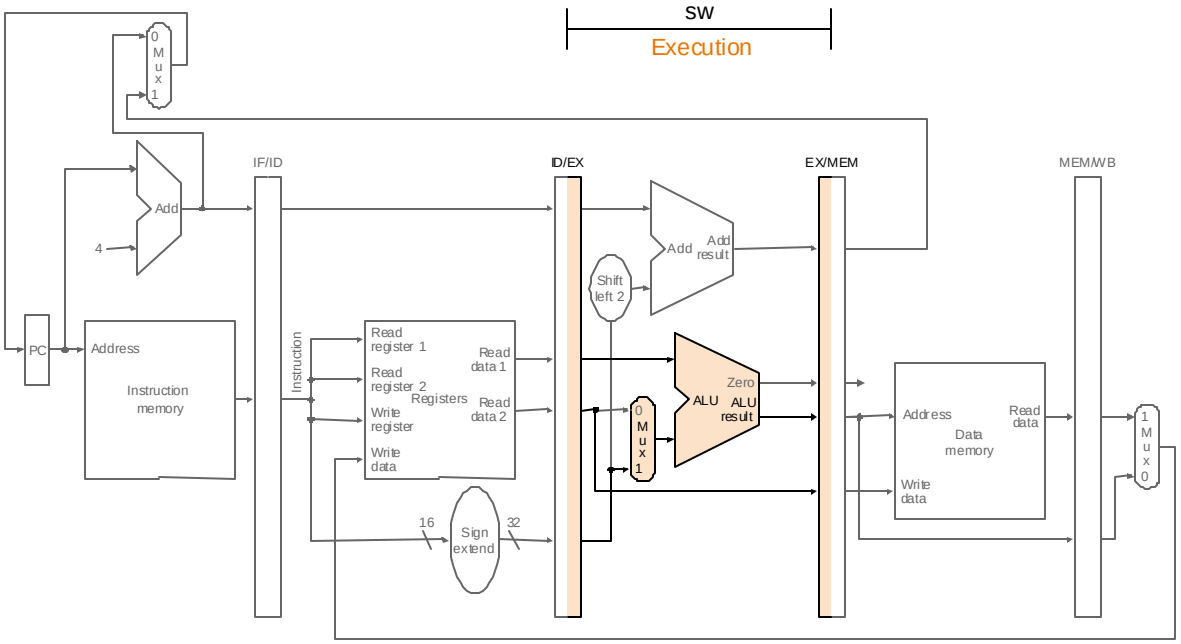
- **Figura 6.14 - Execute or address calculation (lw) ➔ reg1**



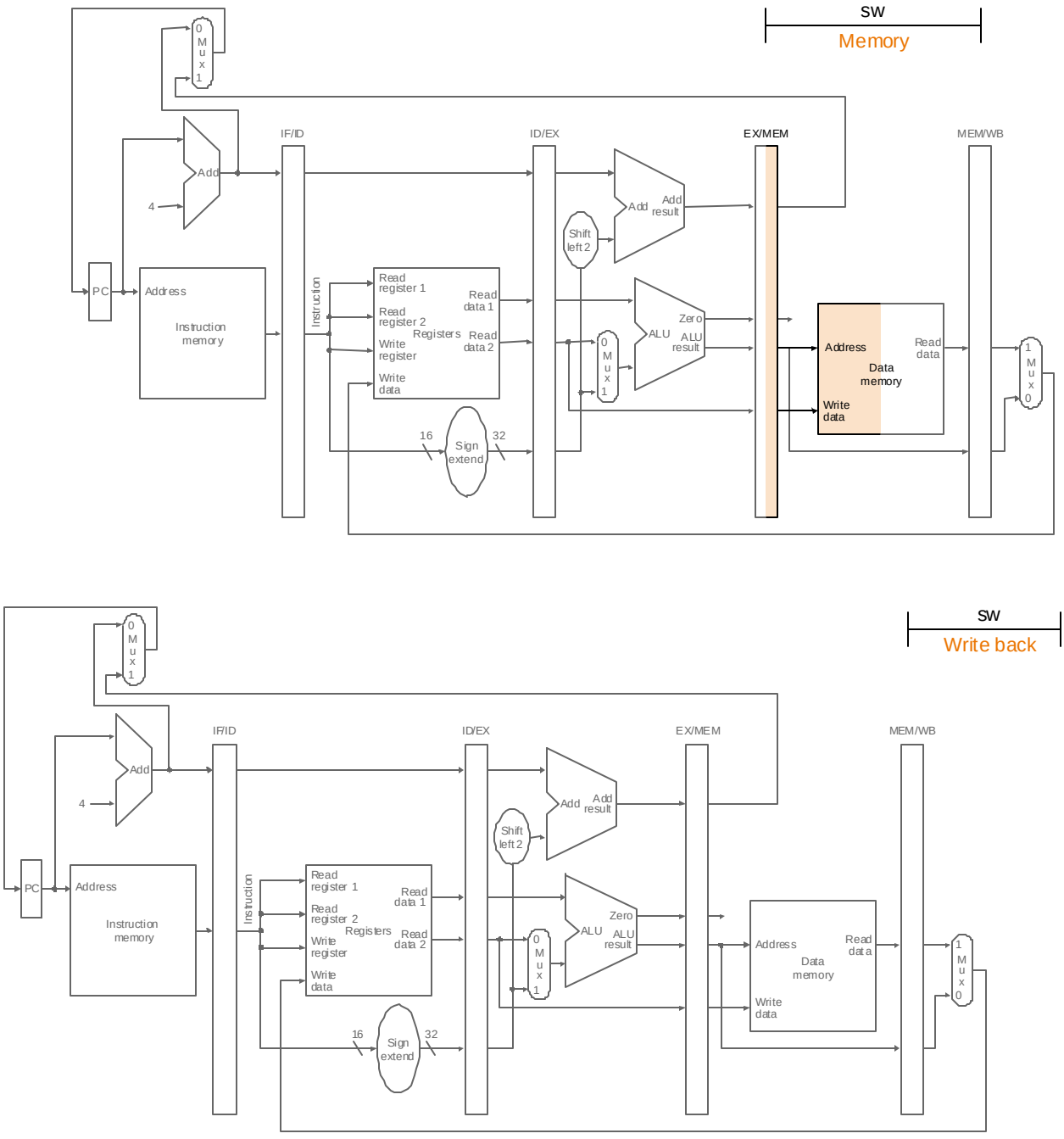
- **Figura 6.15 - Memory access e write back (sw)**



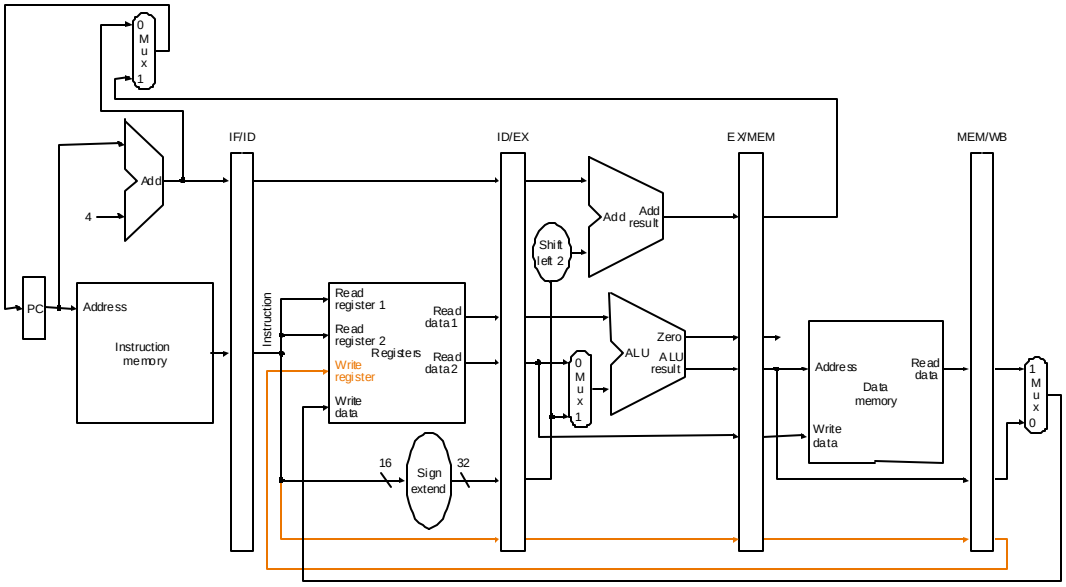
- **Figura 6.16 - Execute and address calculation (sw) ➔ reg2**



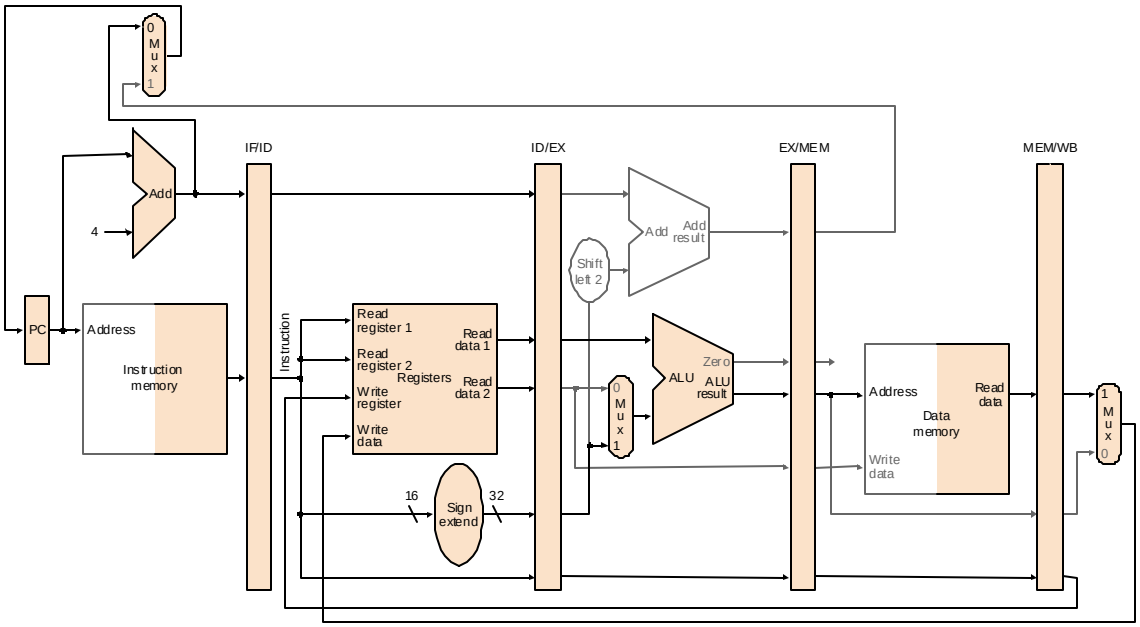
- **Figura 6.17 - Memory access e write back (sw)**



- **Figura 6.18 - Datapath correto (Qual o erro mostrados anteriormente – instrução de load ????)**

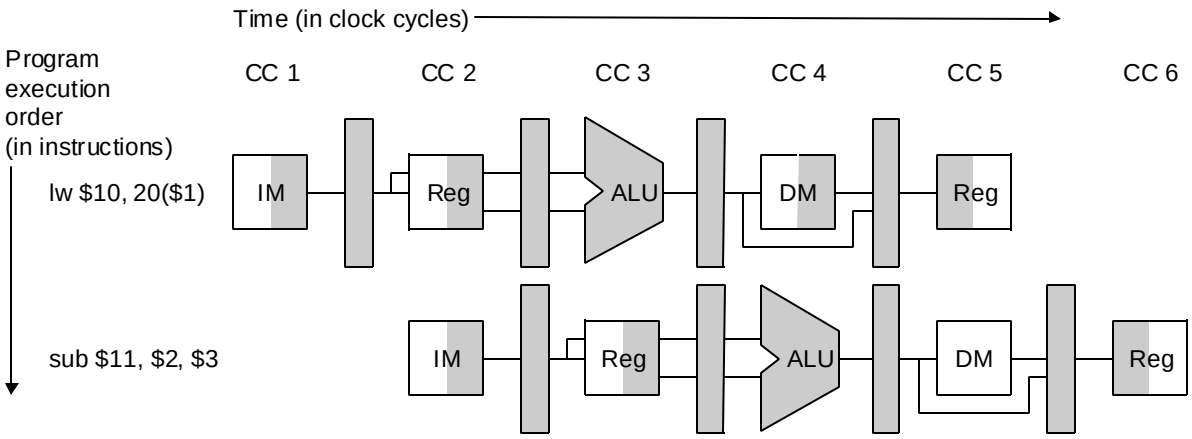


- **Figura 6.19 – Datapath com os estágios usados para um instrução lw**

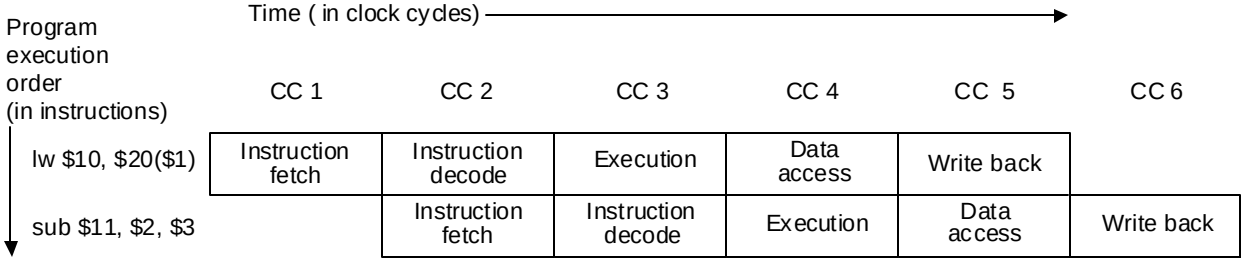


- **Representações gráficas do pipeline**

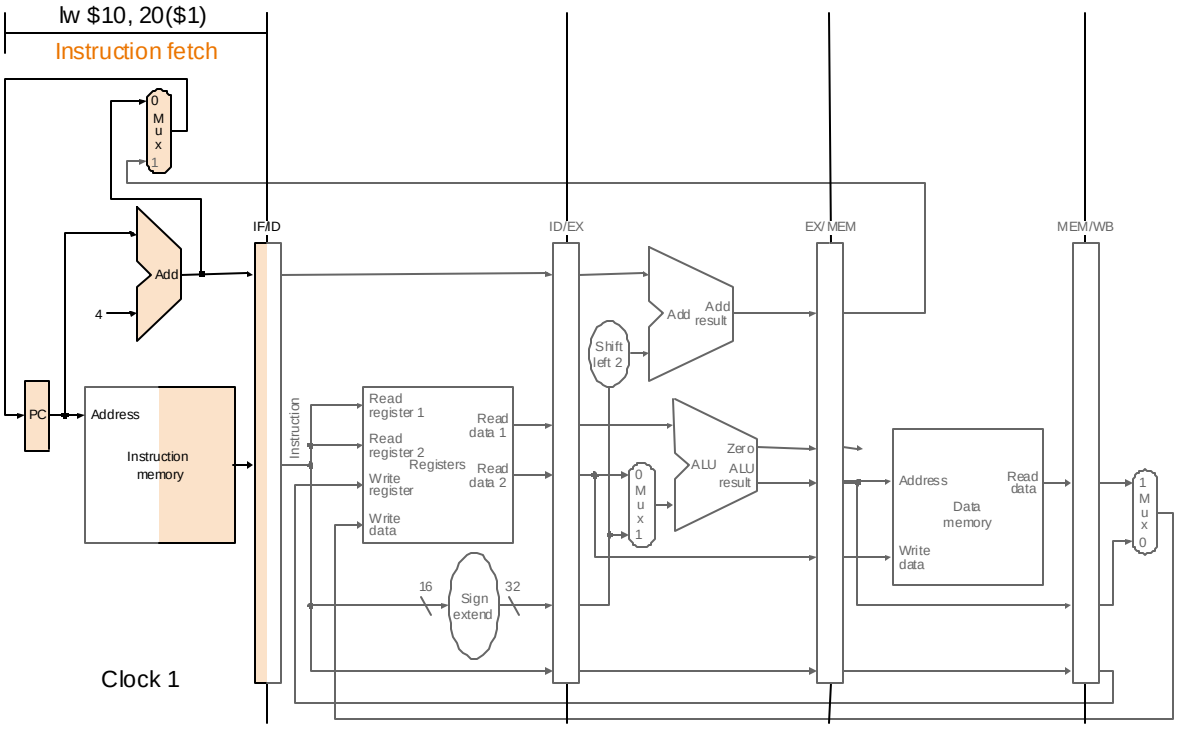
  - **Figura 6.20 - Multiple-clock-cycle pipeline diagram**

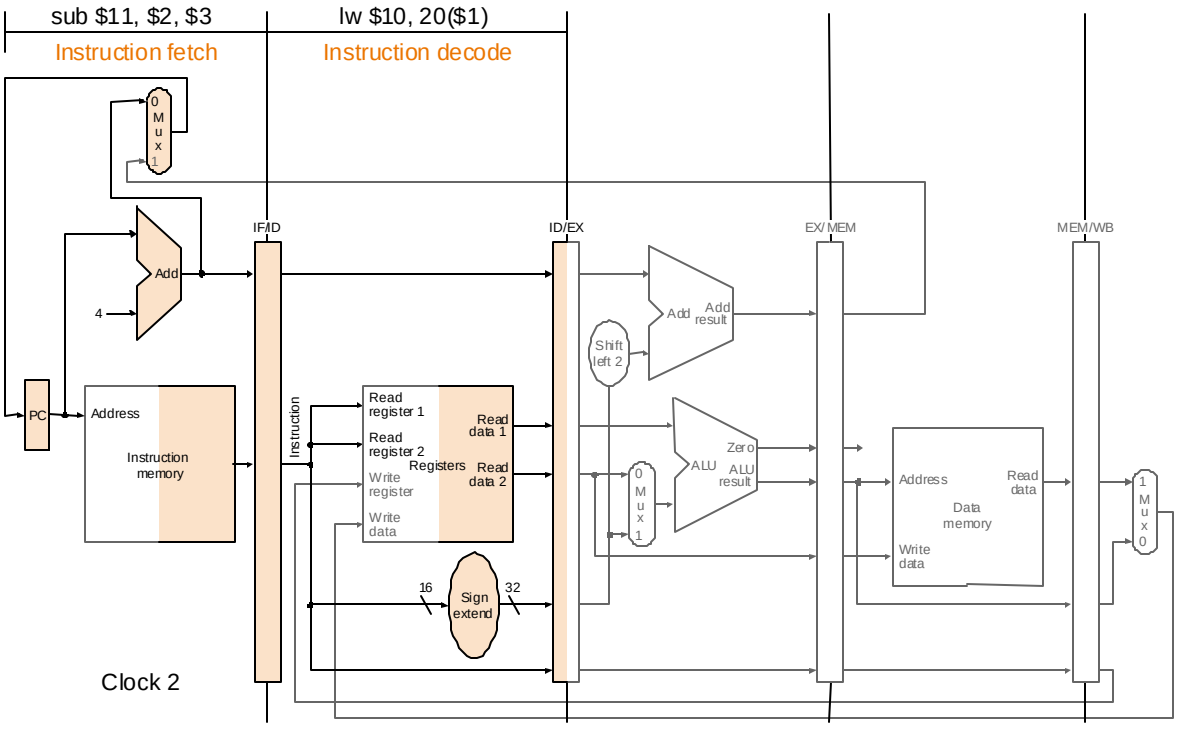


  - **Figura 6.21 - Multiple-clock-cycle pipeline diagram**

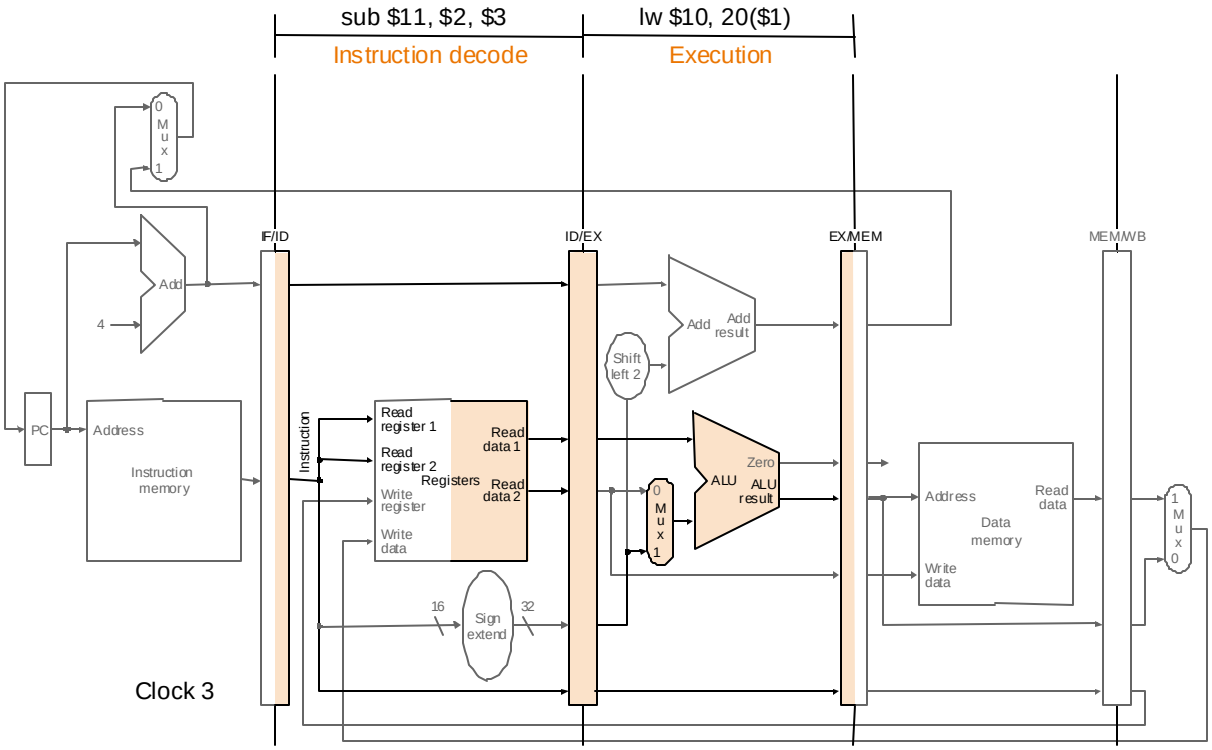


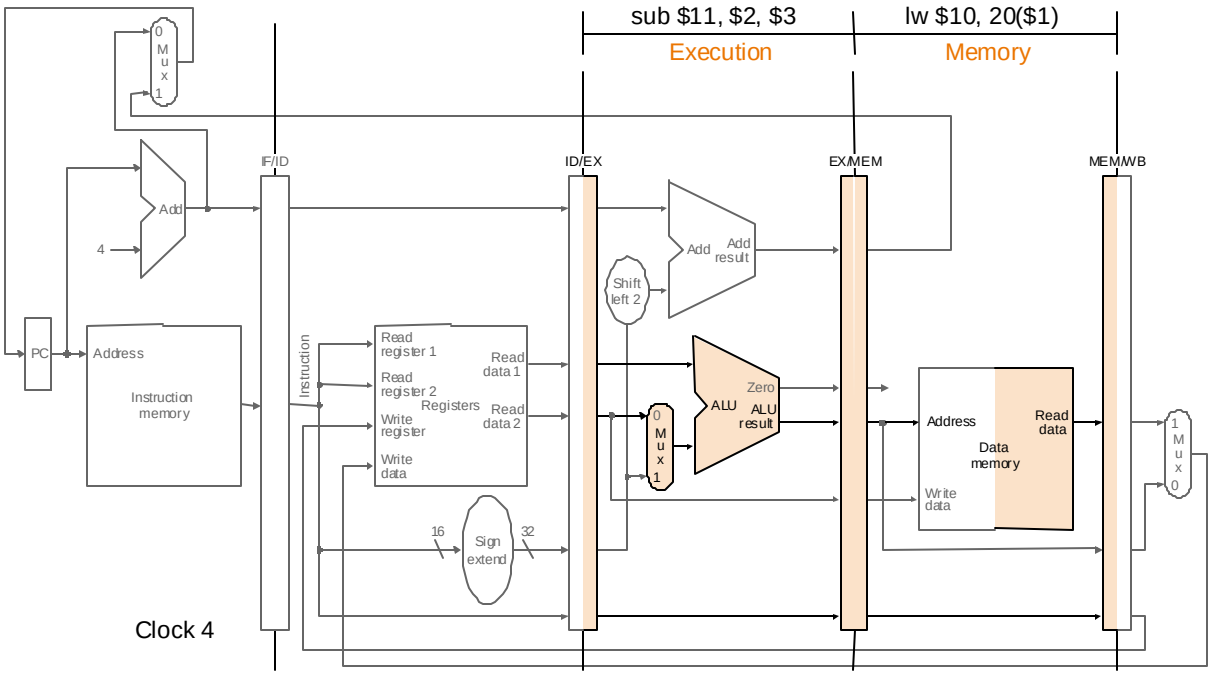
- **single-clock-cycle pipeline diagram**



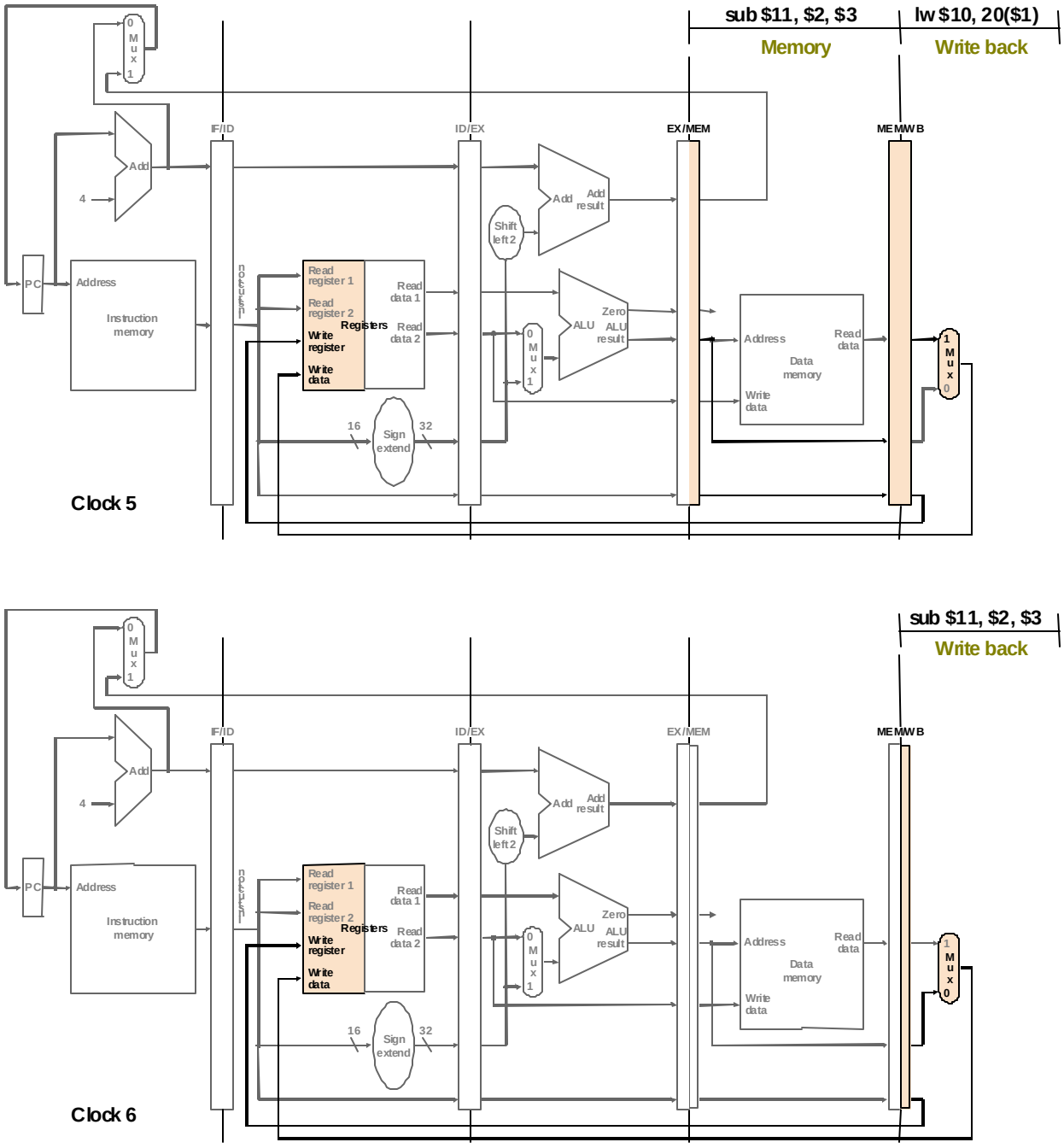


- **Figura 6.23 - single-clock-cycle pipeline diagram**

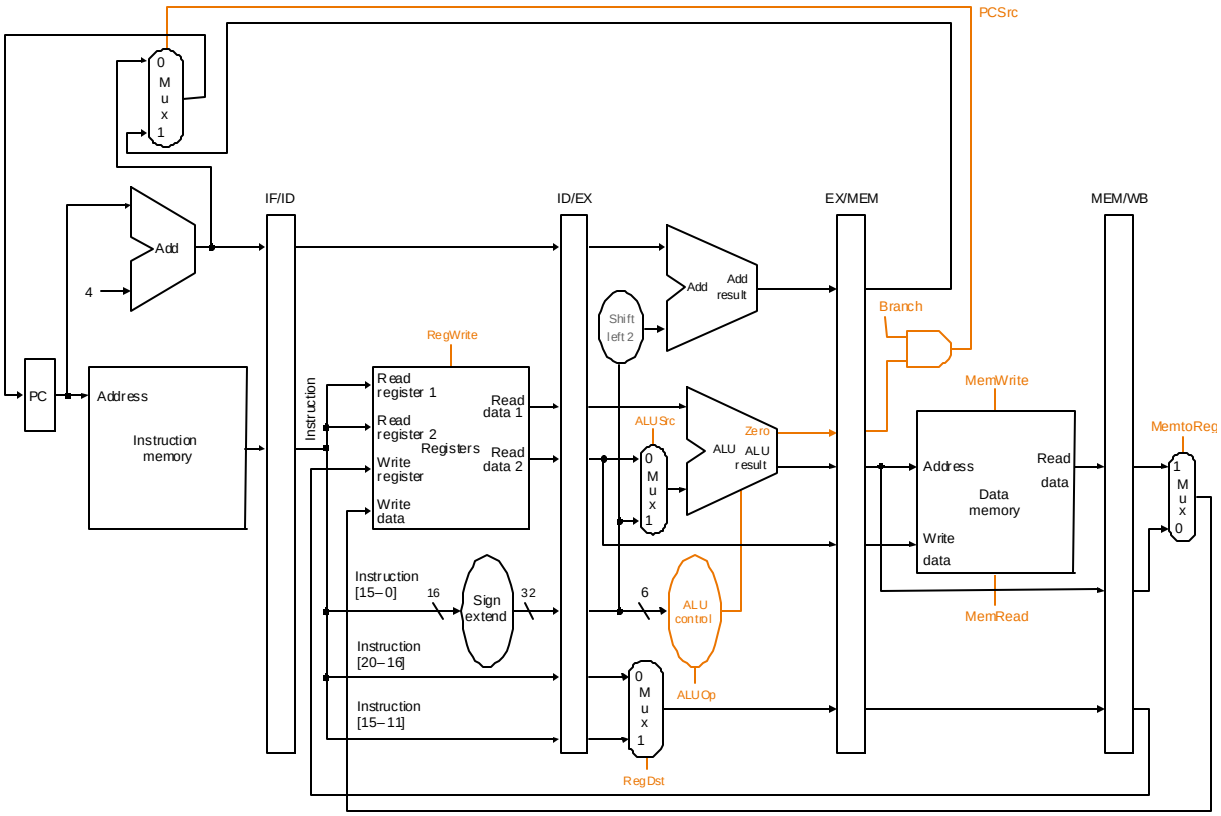




- **Figura 6.24 - single-clock-cycle pipeline diagram**



- **Figura 6.25 - Controle no pipeline**



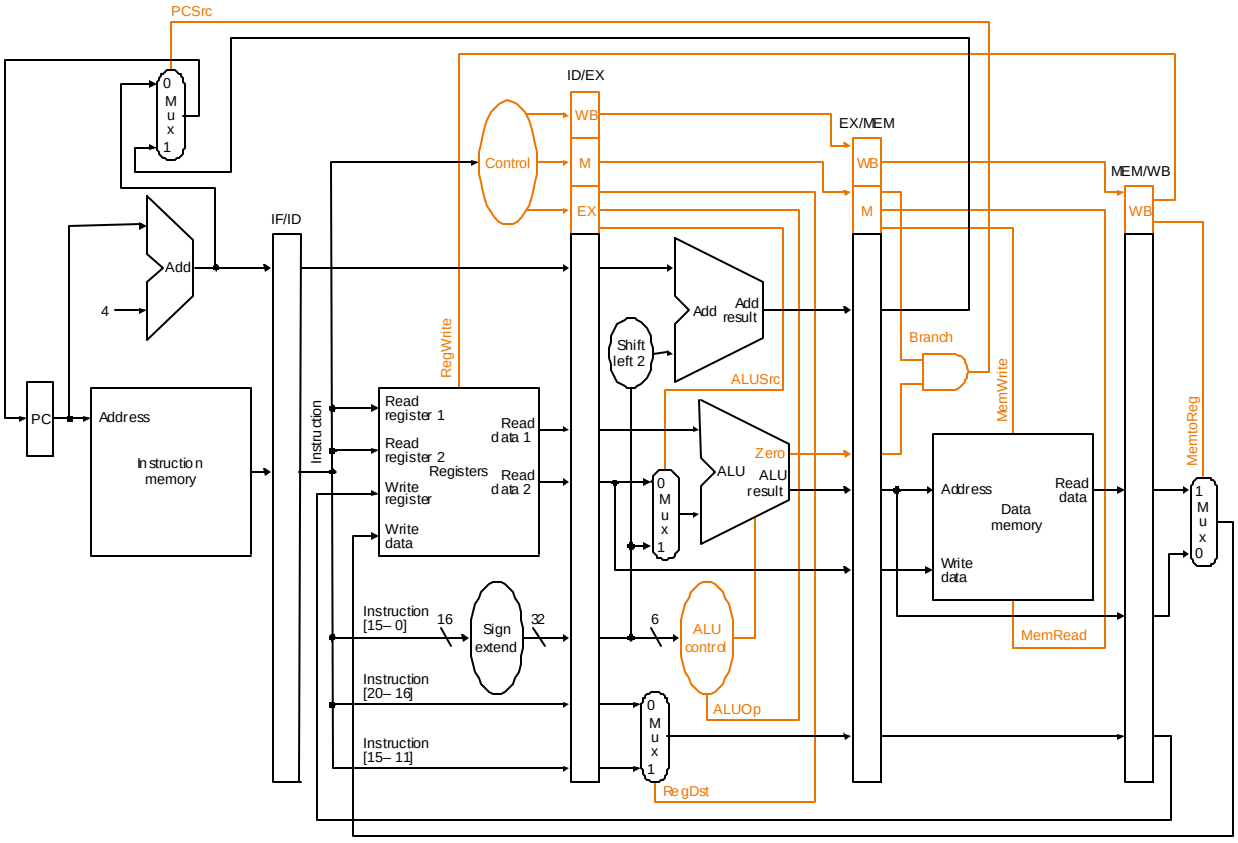
- **Figura 6.26 – 6.27 e 6.28 – quadros relativos aos sinais de controle.**

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

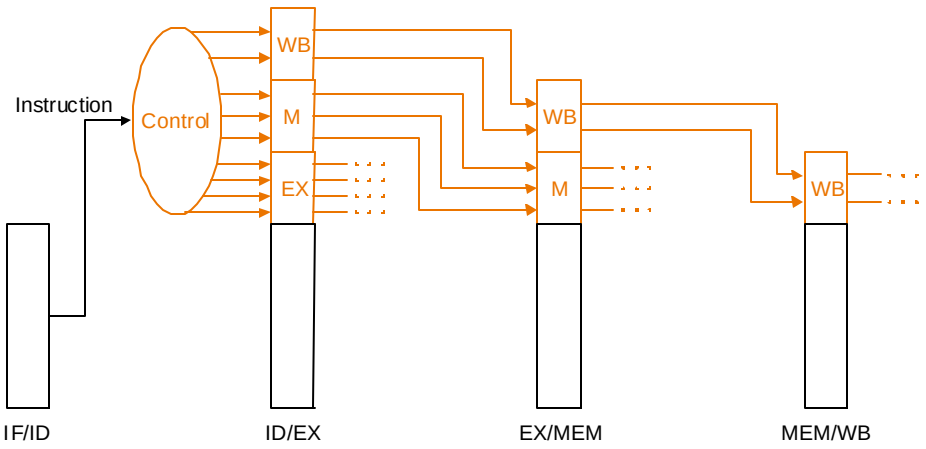
Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20–16).	The register destination number for the Write register comes from the rd field (bits 15–11).
RegWrite	None	The register on the Write register 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.

	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

- **Figura 6.30 – Novos sinais de controle para os últimos 3 estágios – os registradores de pipeline incluem informações de controle**



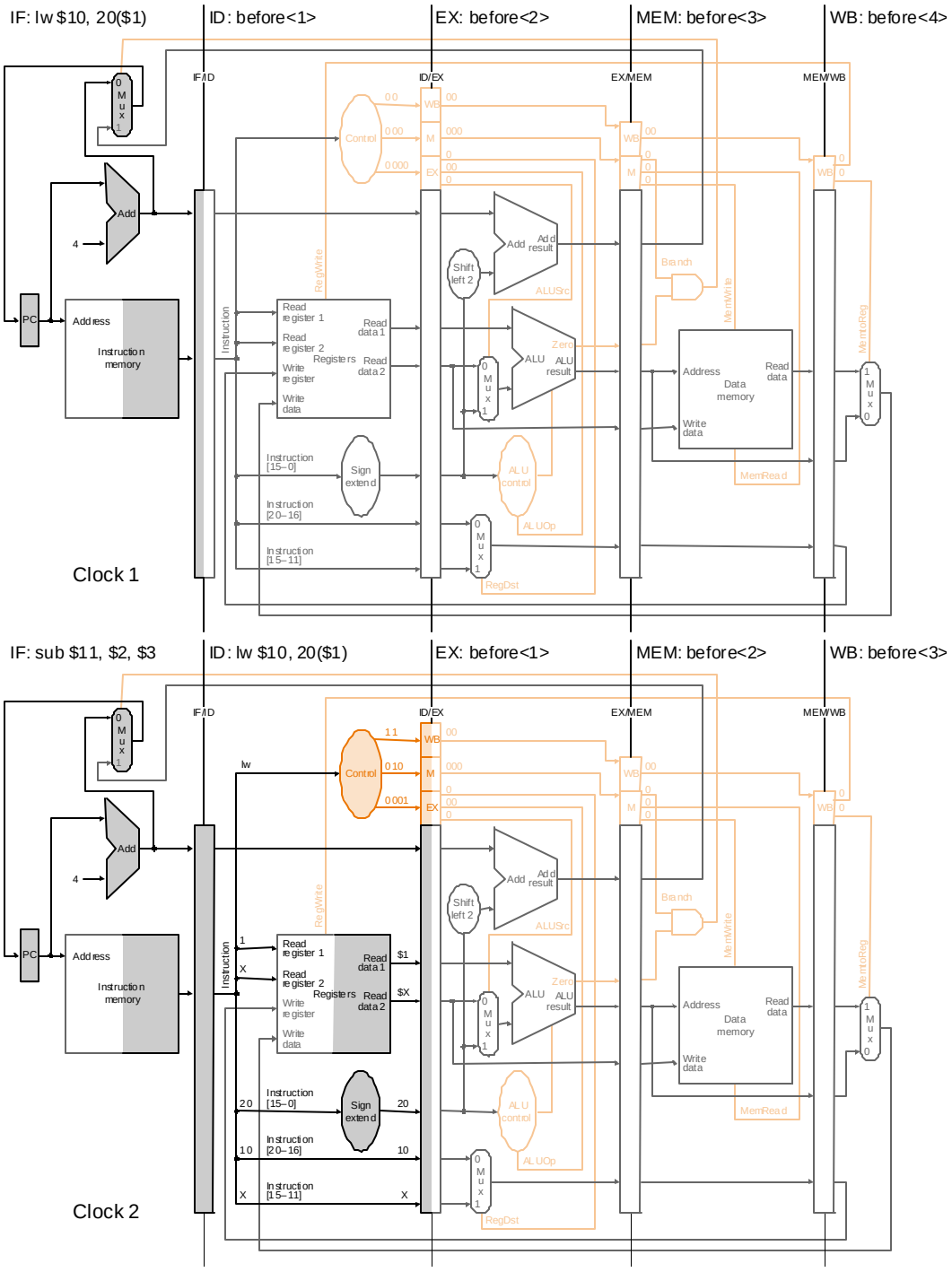
- **Figura 6.29 – Linha de controles para os três últimos estágios**



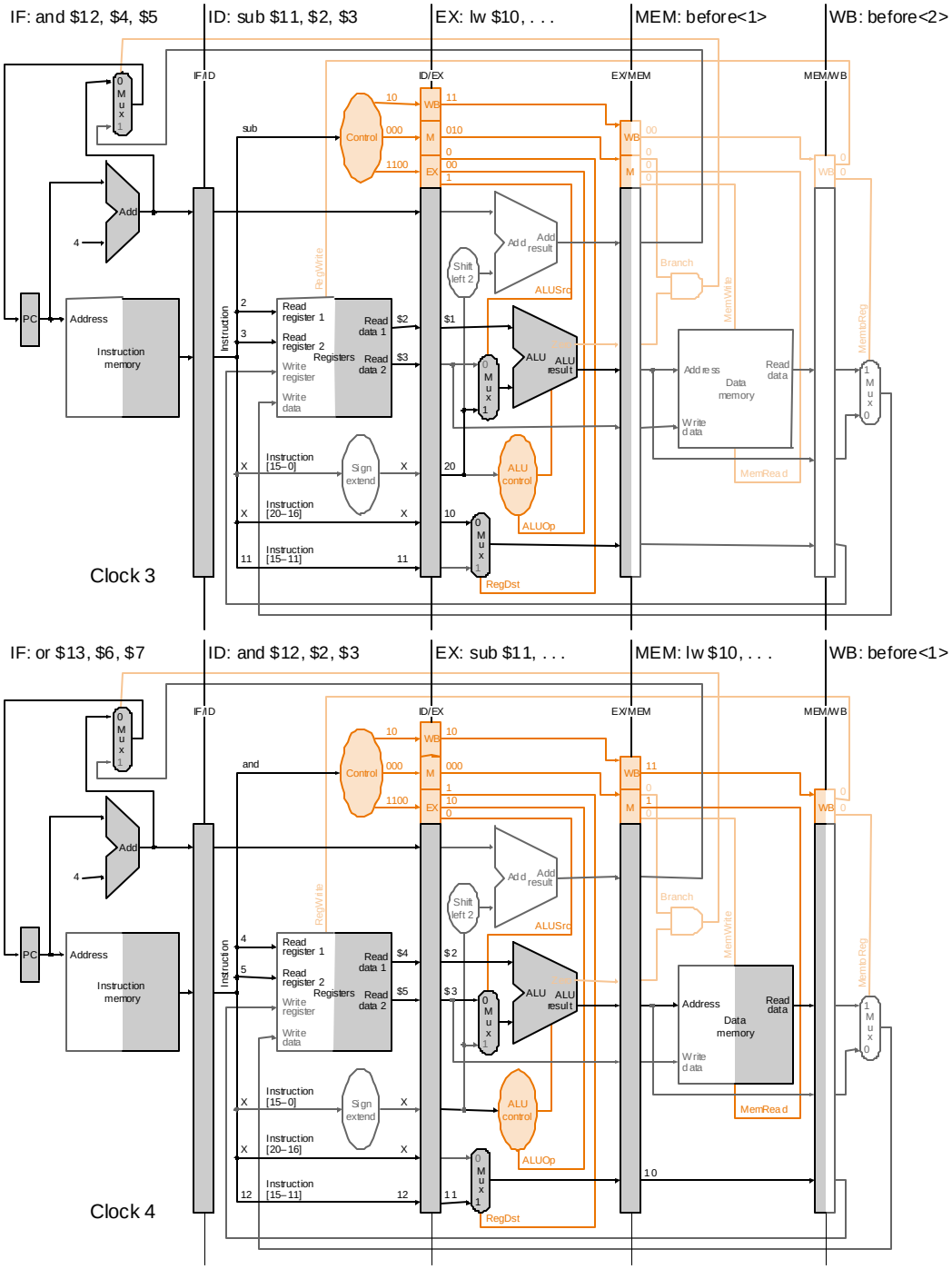
- **Exemplo**

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

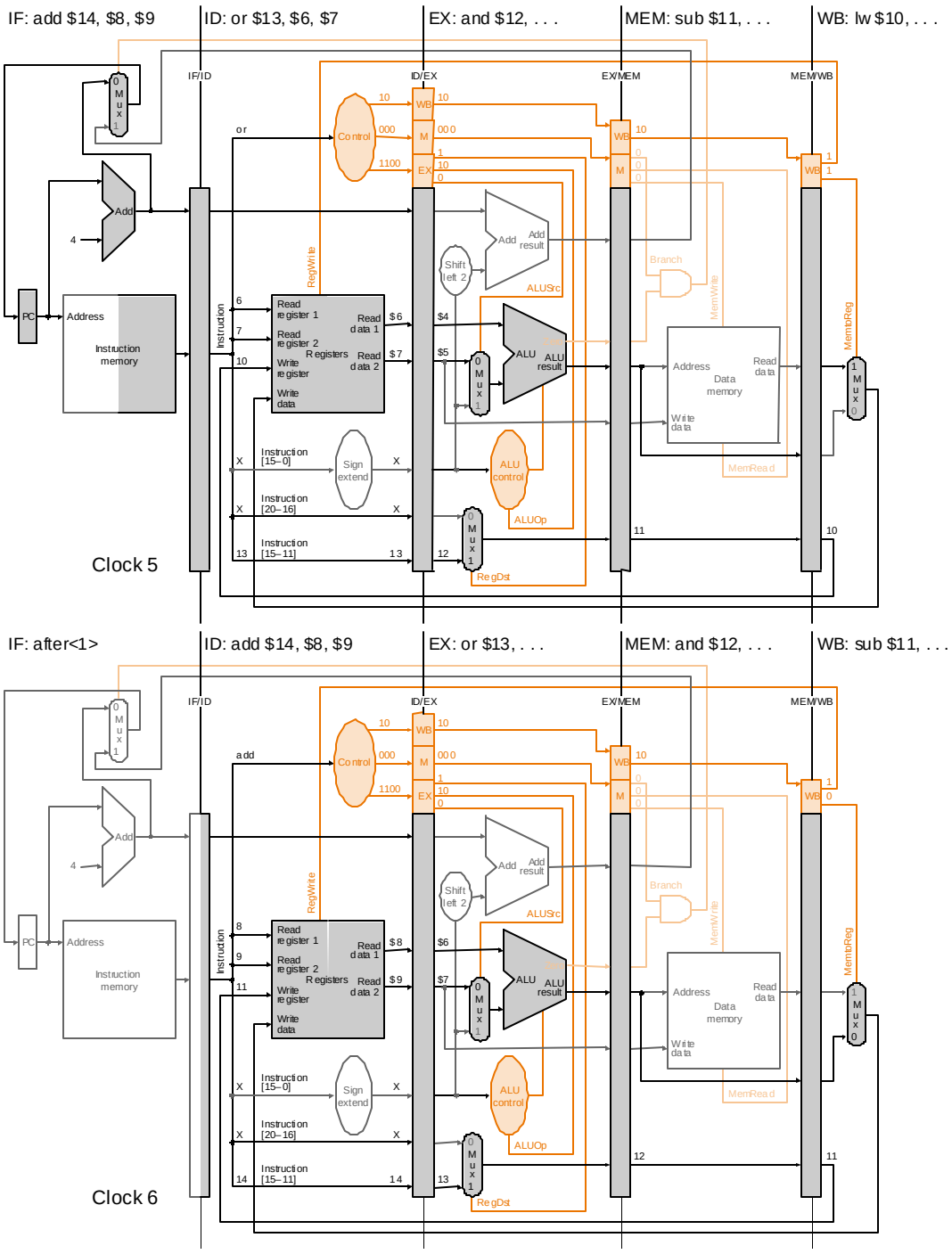
- **Figura 6.31 – Ciclos 1 e 2**



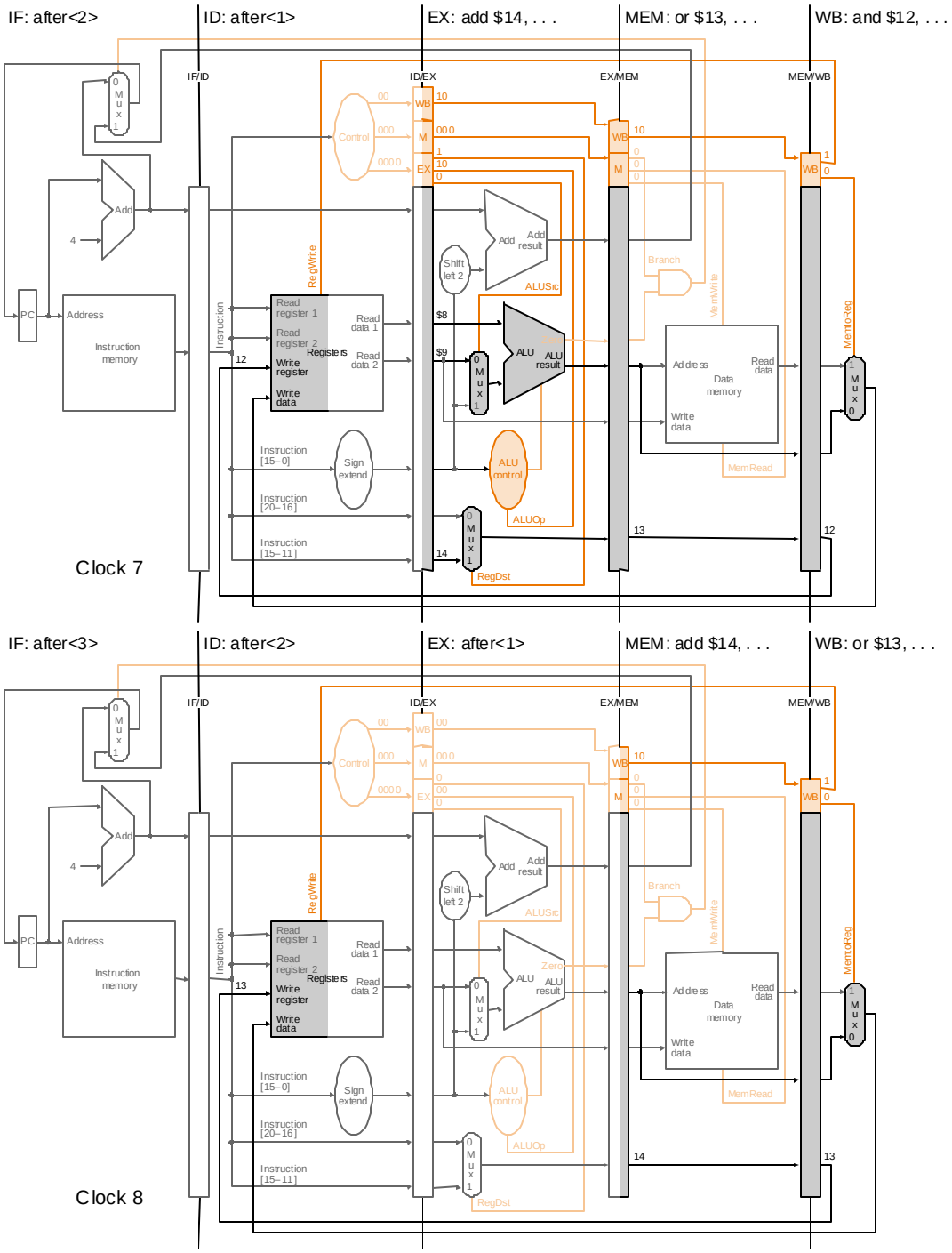
- **Figura 6.32 – Ciclos 3 e 4**



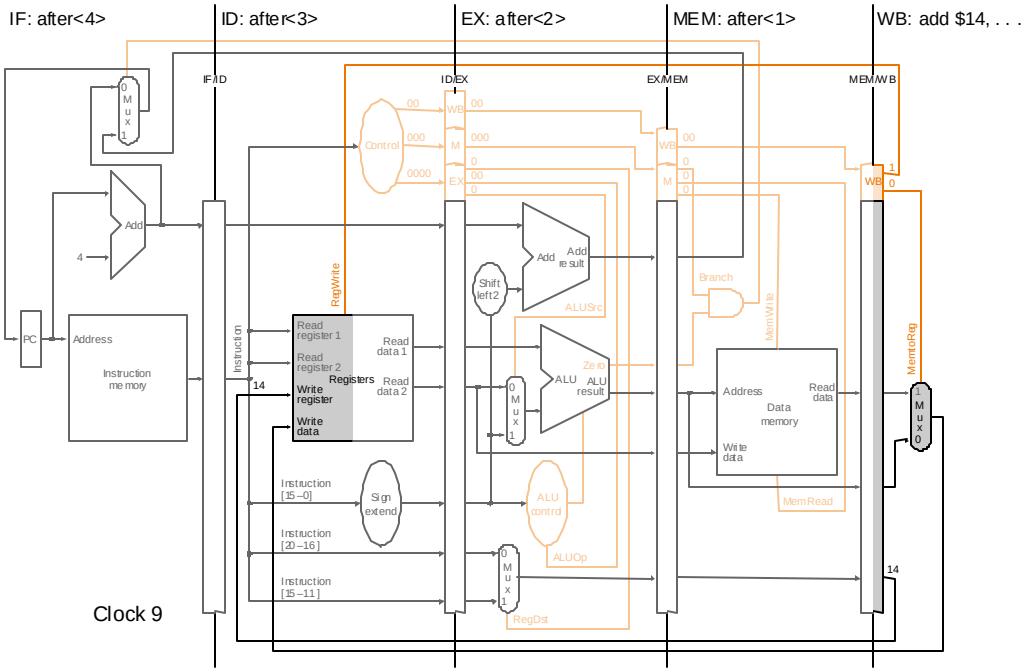
- **Figura 6.33 – Ciclos 5 e 6**



- **Figura 6.34 – Ciclos 7 e 8**



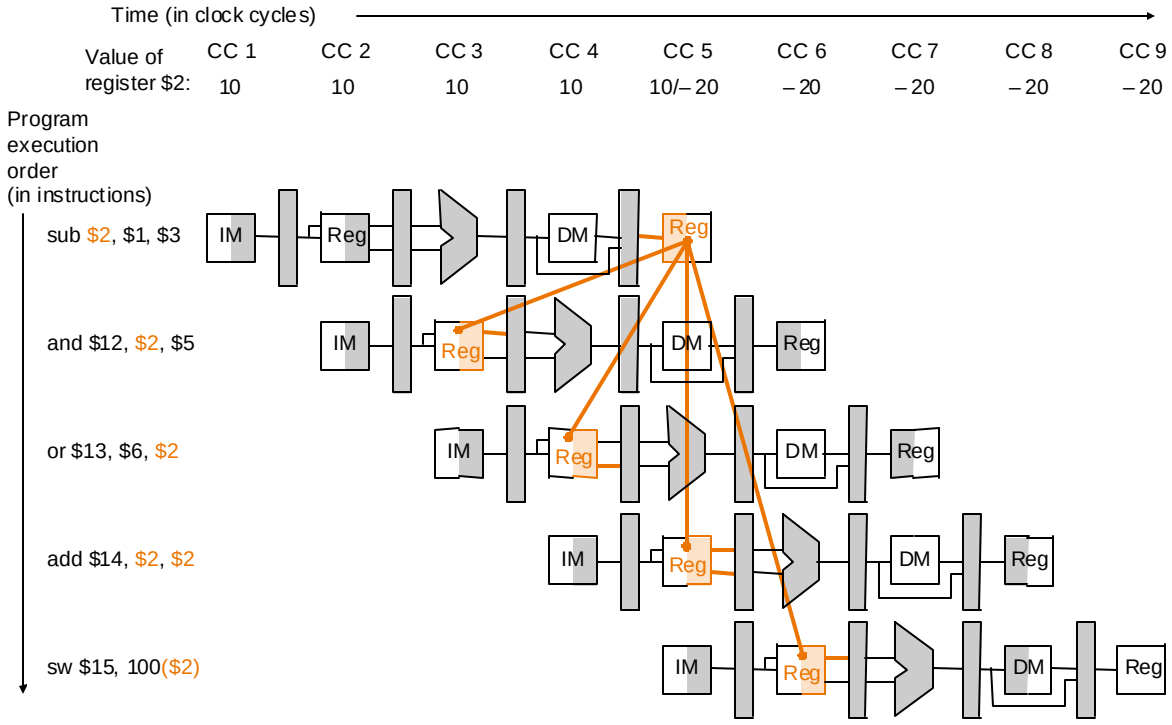
- **Figura 6.35 – Ciclo 9**



- **Data Hazards and Fowarding**

  Instruções com dependências

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



- **Para leitura e escrita de um registrador em um único ciclo de clock, o register file pode ser implementado de tal forma que a escrita seja feita no primeiro semi-ciclo e a leitura no segundo.**

- **Uma primeira (e não ótima) solução ➔ proibir o compilador de gerar seqüências com dependências. Isto é feito inserindo instruções que não dependam da anterior ou instruções do tipo nop.**

```
sub  $2,   $1,   $3     # reg $2 modificado
nop
nop
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 operando)
sw   $15,  100($2)      # idem (base do endereçamento)
```

- **Detecção de Hazard ➔ quando uma instrução tenta ler um registrador no estágio EX, que uma instrução anterior a esta ainda irá escrever no estágio WB.**

- **Condição de Hazard**

  1a ➔ EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b ➔ EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a ➔ MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b ➔ MEM/WB.RegisterRd = ID/EX.RegisterRt

Exemplo:

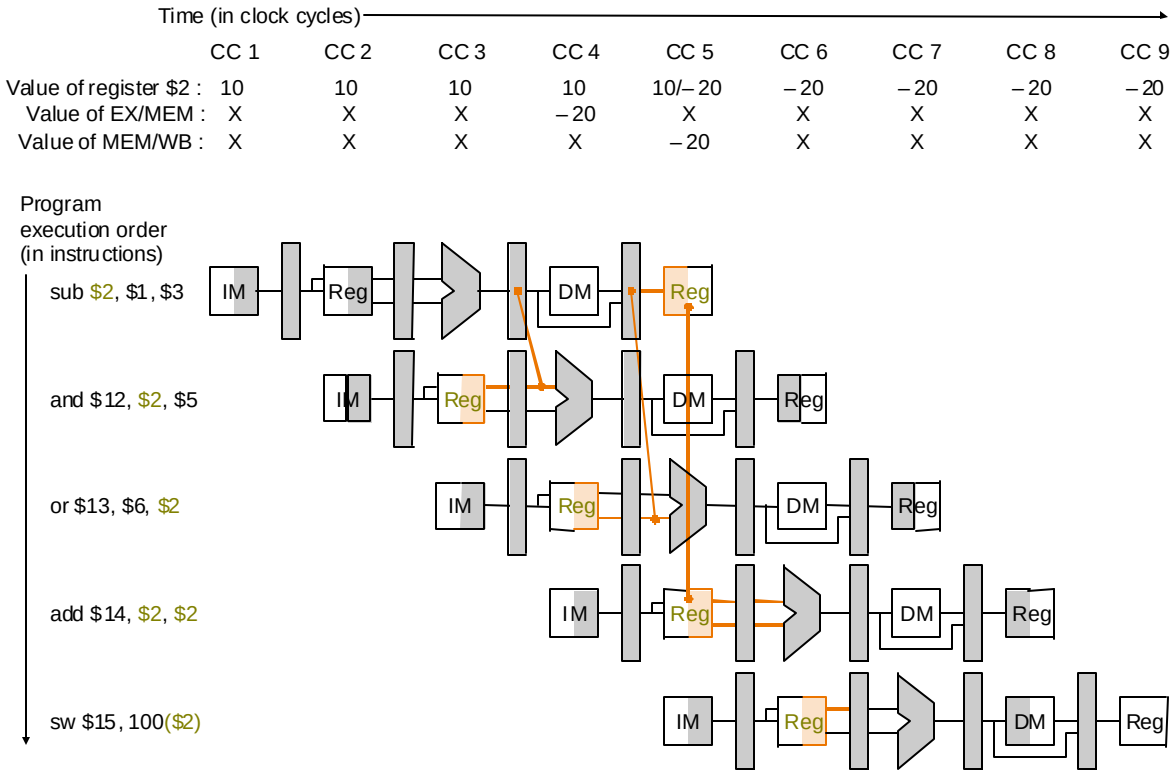
```
sub   $2,   $1,   $3     #  reg $2 modificado
and   $12,  $2,   $5     #  valor de $2  depende do sub
or    $13,  $6,   $2     #  idem (2° operando)
add   $14,  $2,   $2     #  idem (1° e 2° operando)
sw    $15,  100($2)      #  idem (base do endereçamento)
```

- **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**

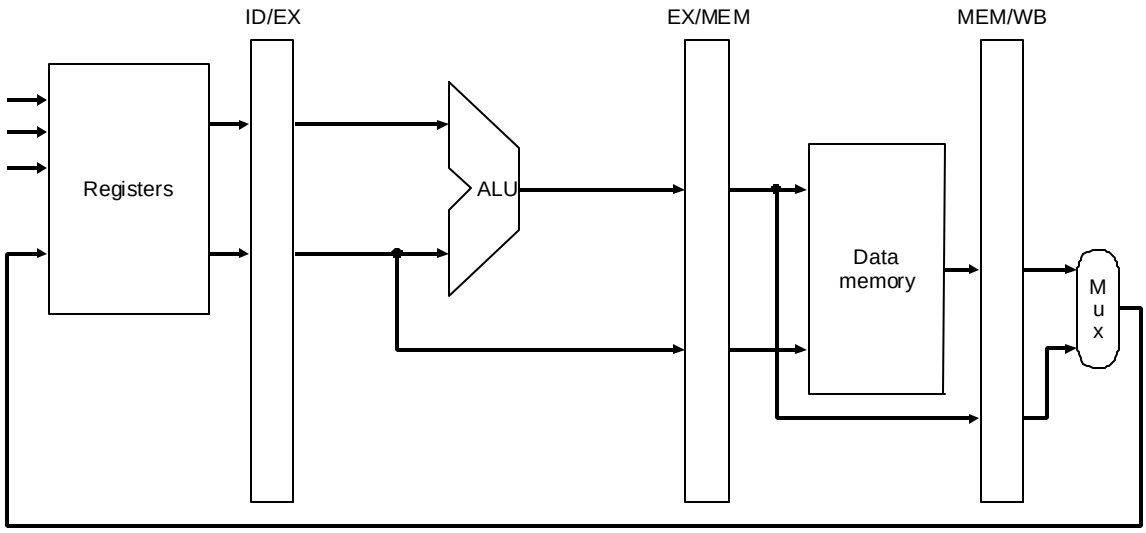
- **Esta não é uma política precisa, pois existem instruções que não escrevem no register file ➔ uma solução seria verificar o sinal RegWrite.**

- **MIPS usa $0 para operandos de valor 0. Para instruções onde $0 é destino, queremos permitir resultados diferentes de 0 ➔ ➔ sll $0, $1, 2 ➔ não fazendo fowarding permite maior liberdade para o programador e o compilador a usar o $0 como destino. Para isto temos que incluir as condições EX/MEM.registerRd <> 0 (1° hazard) e MEM/WB.registerRd <> 0 (2° hazard)**

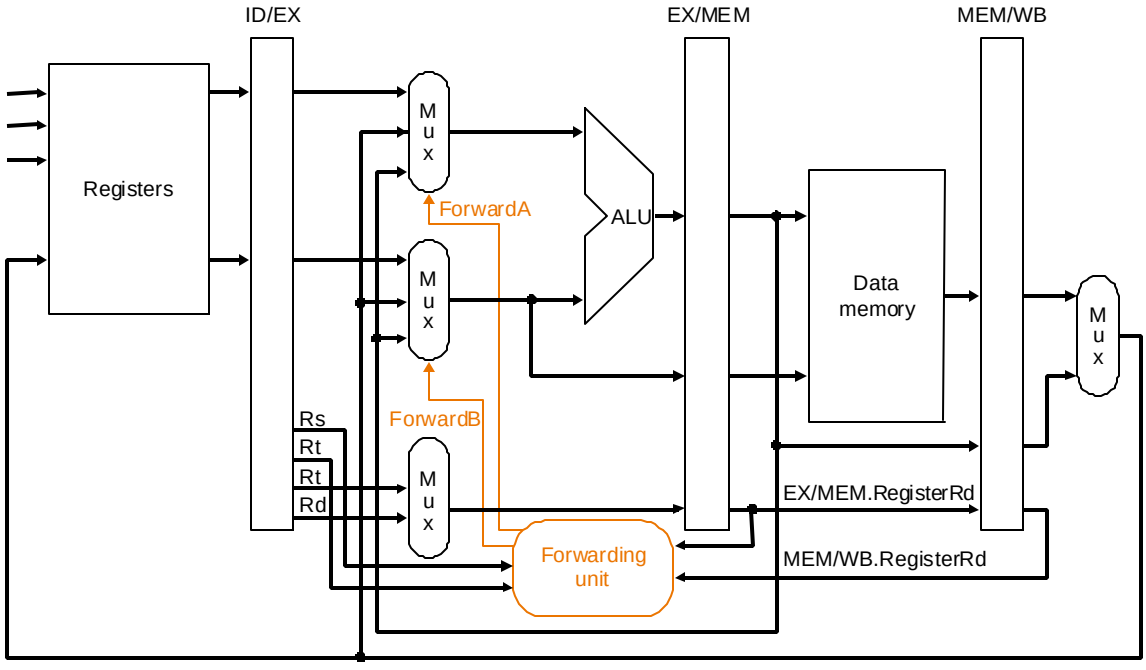
- **Dependências entre os registradores de pipeline e entradas da ULA.**



- **Datapath com fowarding (para add, sub, and e or)**



a. No forwarding



b. With forwarding

- **Valores dos sinais ForwardA e ForwardB**

Mux Control	Source	Explanation
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**

- **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**

  1. **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

2. **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

- **Não existe data hazard no estágio WB, pois nós 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.**

- **Um potencial data hazard 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.**

Exemplo: Leitura e escrita em um mesmo registrador

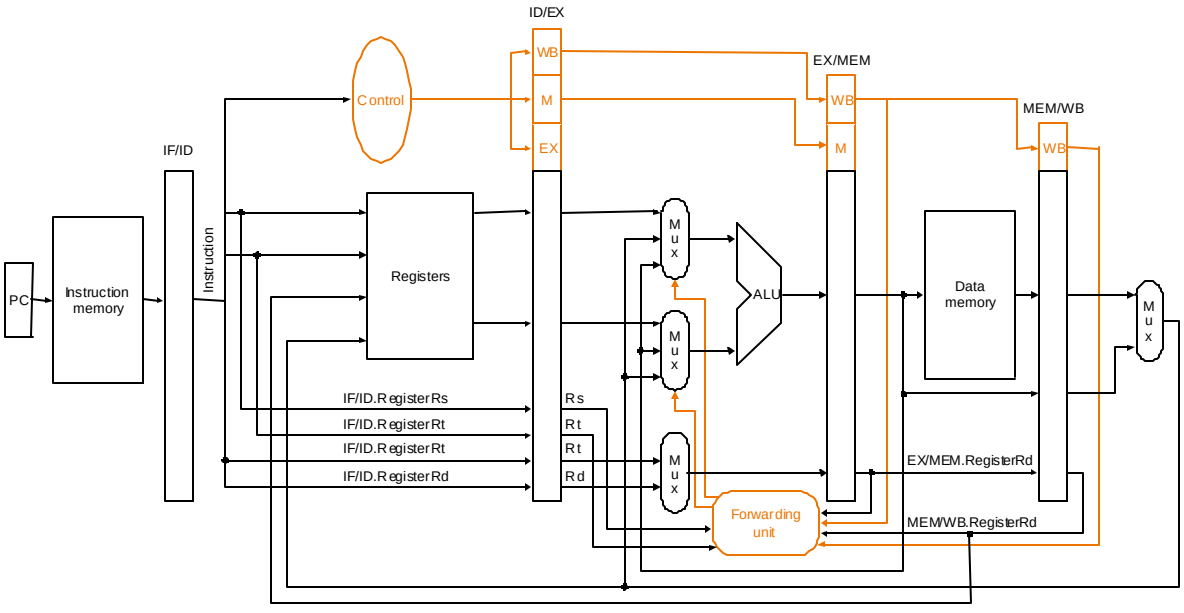
```
add  $1, $1, $2
add  $1, $1, $3
add  $1, $1, $4
```

- **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

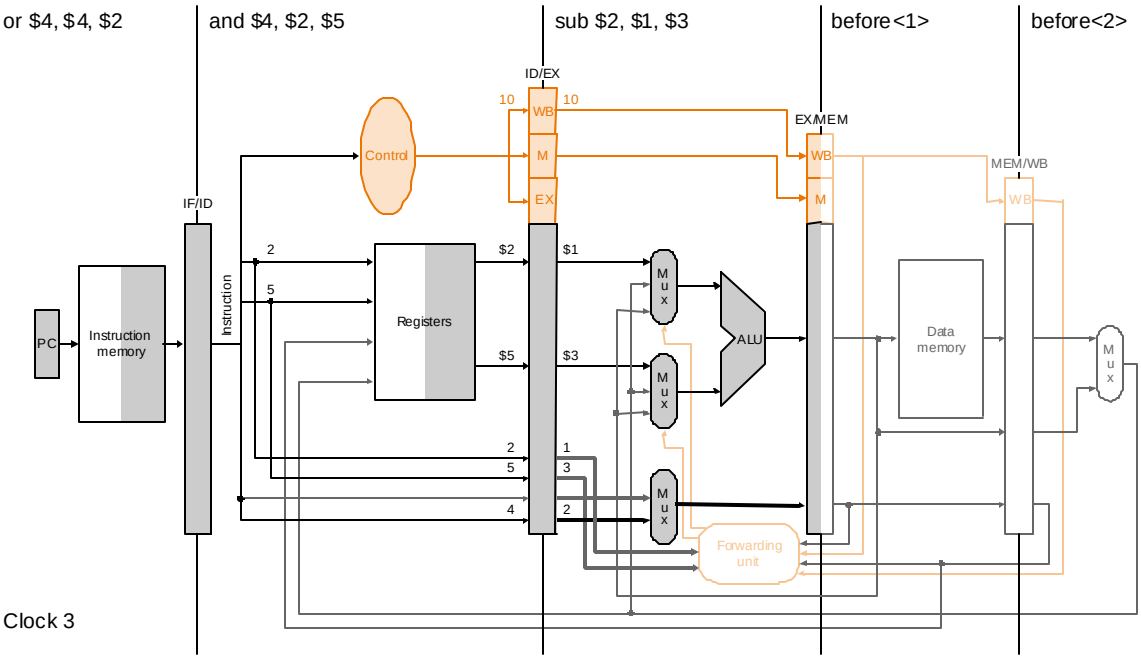
  if (MEM/WB.RegWrite and (MEM/WB.RegisterRd <> 0) and (EX/MEM.RegisterRd <> ID/EX.registerRt) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) FowardB = 01

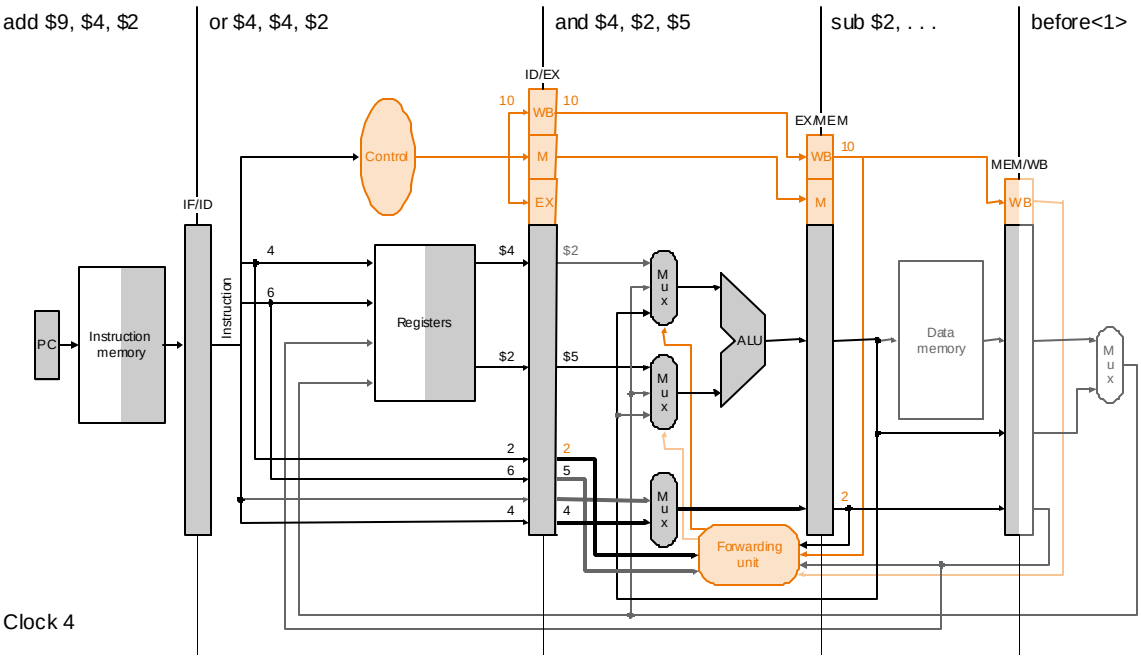
- **Datapath modificado para fowarding**

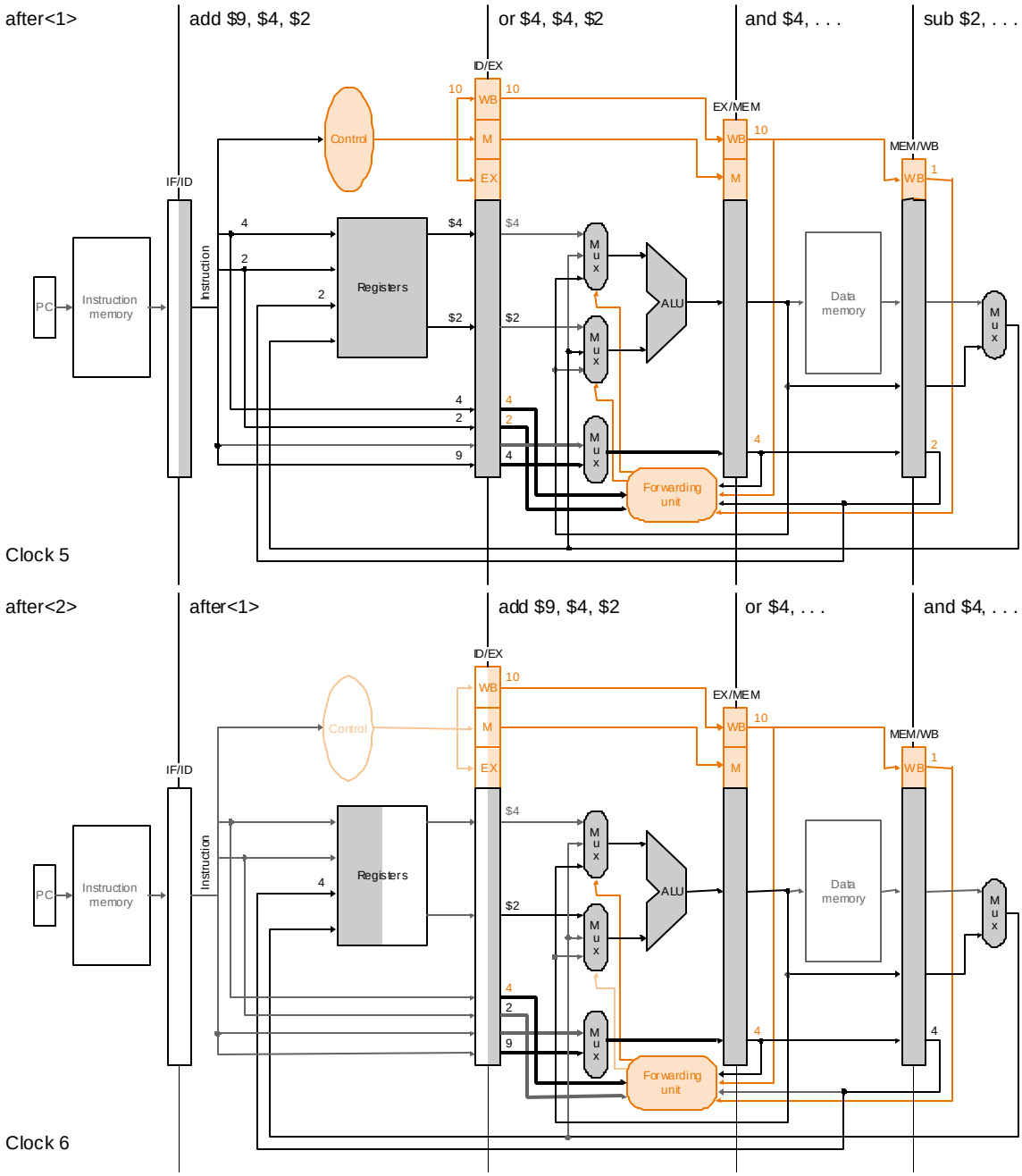


- **Exemplo**

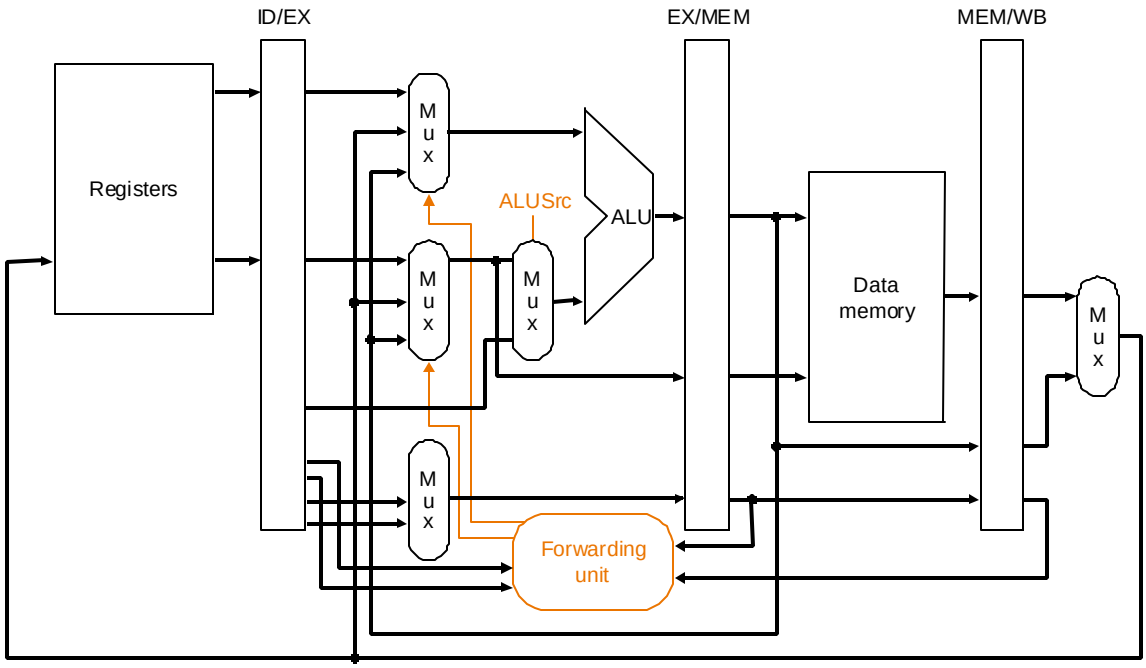
```
sub  $2, $1, $3
and  $4, $2, $5
or   $4, $4, $2
add  $9, $4, $2
```



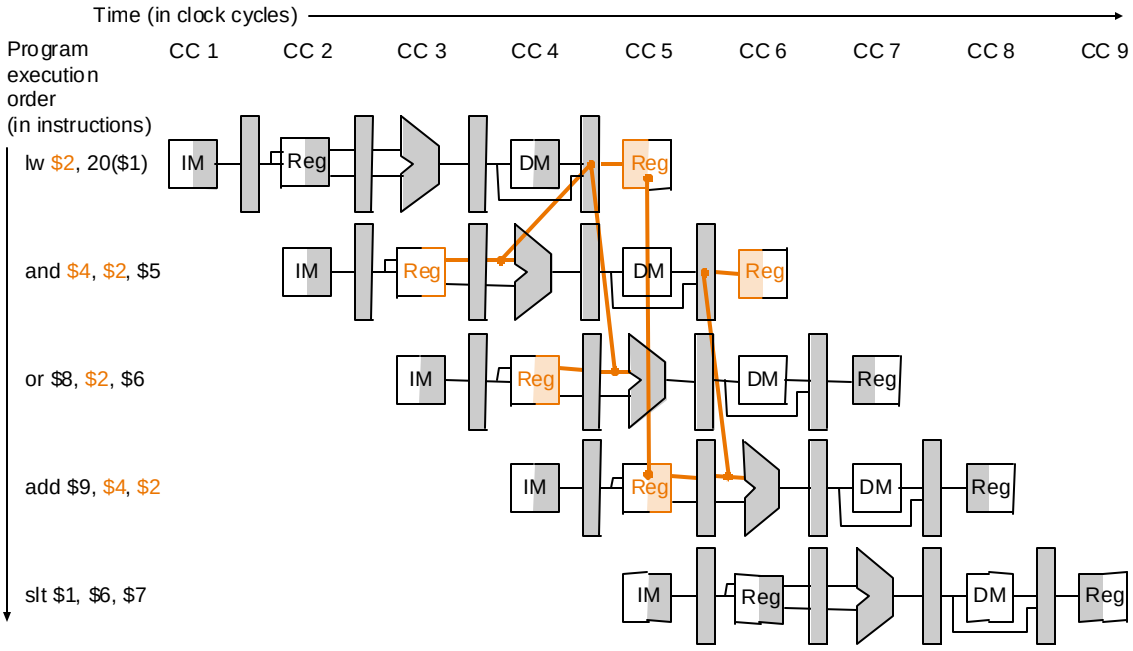



after<1>
add $9, $4, $2
or $4, $4, $2
and $4, . . .
sub $2, . . .
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Mux
Forwarding unit
Clock 5
after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
Clock 6

- **Datapath para entrada signed-imediate necessária para lw e sw**



- **Data Hazards e Stalls**

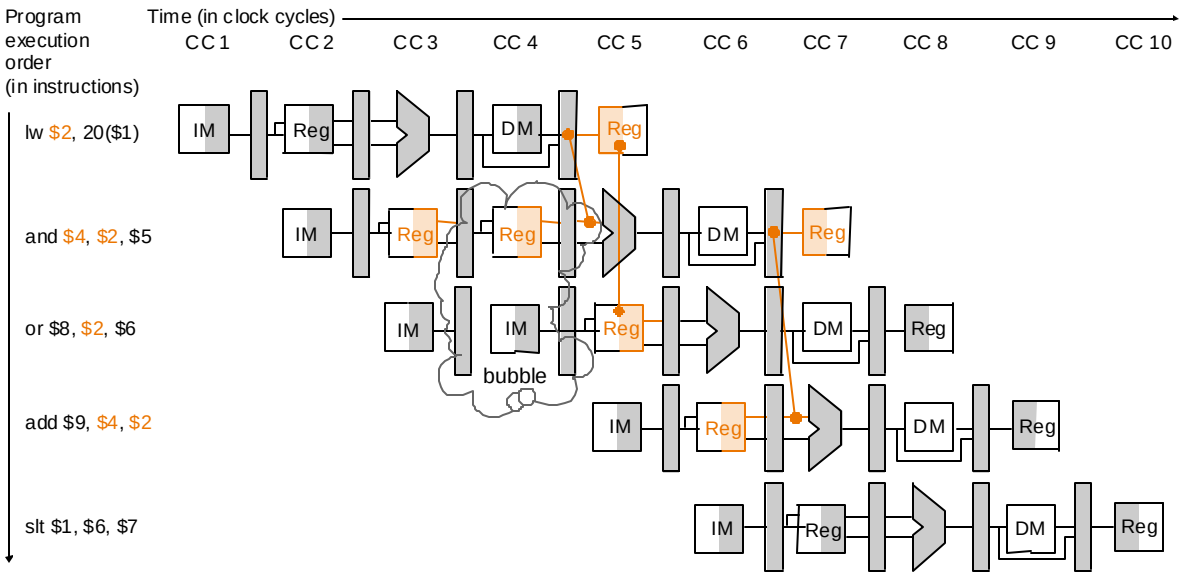


- **Quando uma instrução tenta ler um registrador precedida por uma instrução de load, que escreve no mesmo registrador ➔ o dado tem que ser mantido (ciclo 4) enquanto a ULA executa a operação ➔ atrasar o pipeline para que a instrução leia o valor correto.**

- **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

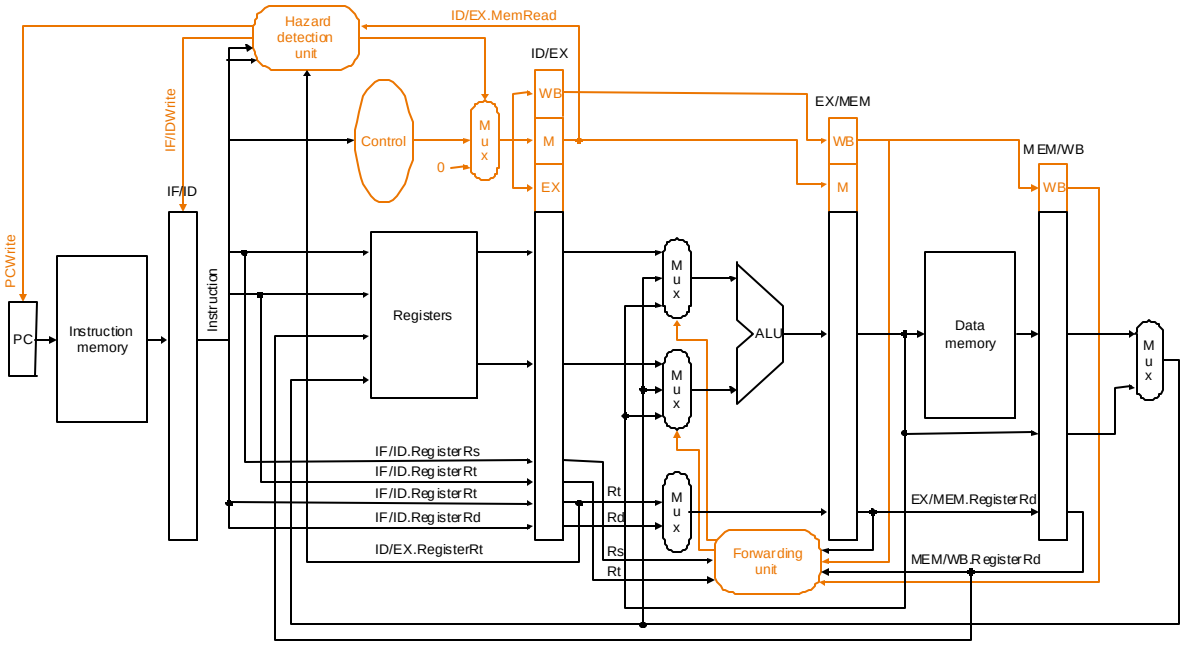


- **Se a instrução no estágio ID é atrasada ➔ a instrução no estagio IF também deve ser atrasada**

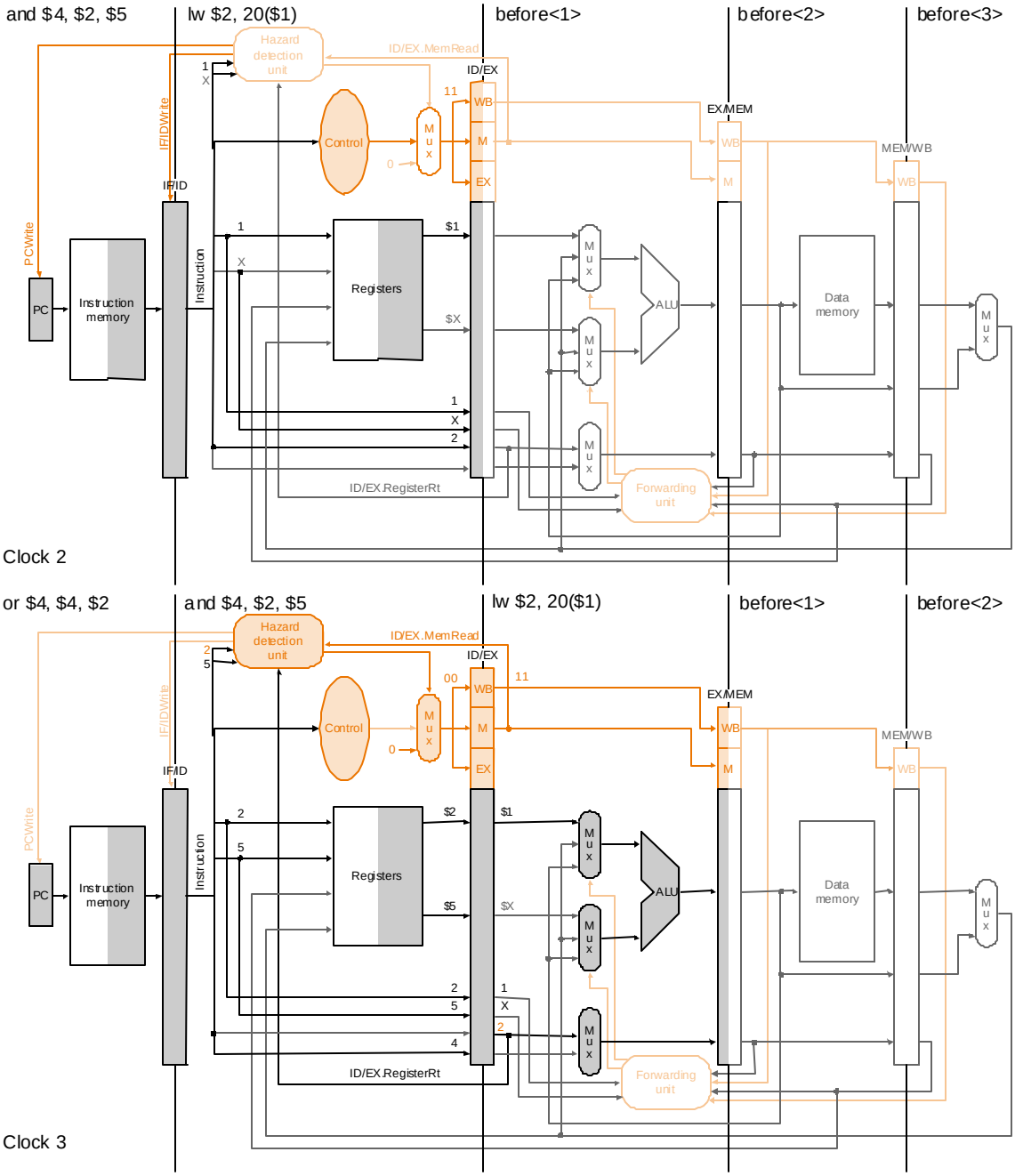
- **Como fazer ? ➔ impedir a mudança no PC e no registrador IF/ID ➔ a instrução em IF continua sendo lida e em ID continuam sendo lido os mesmos campos da instrução.**

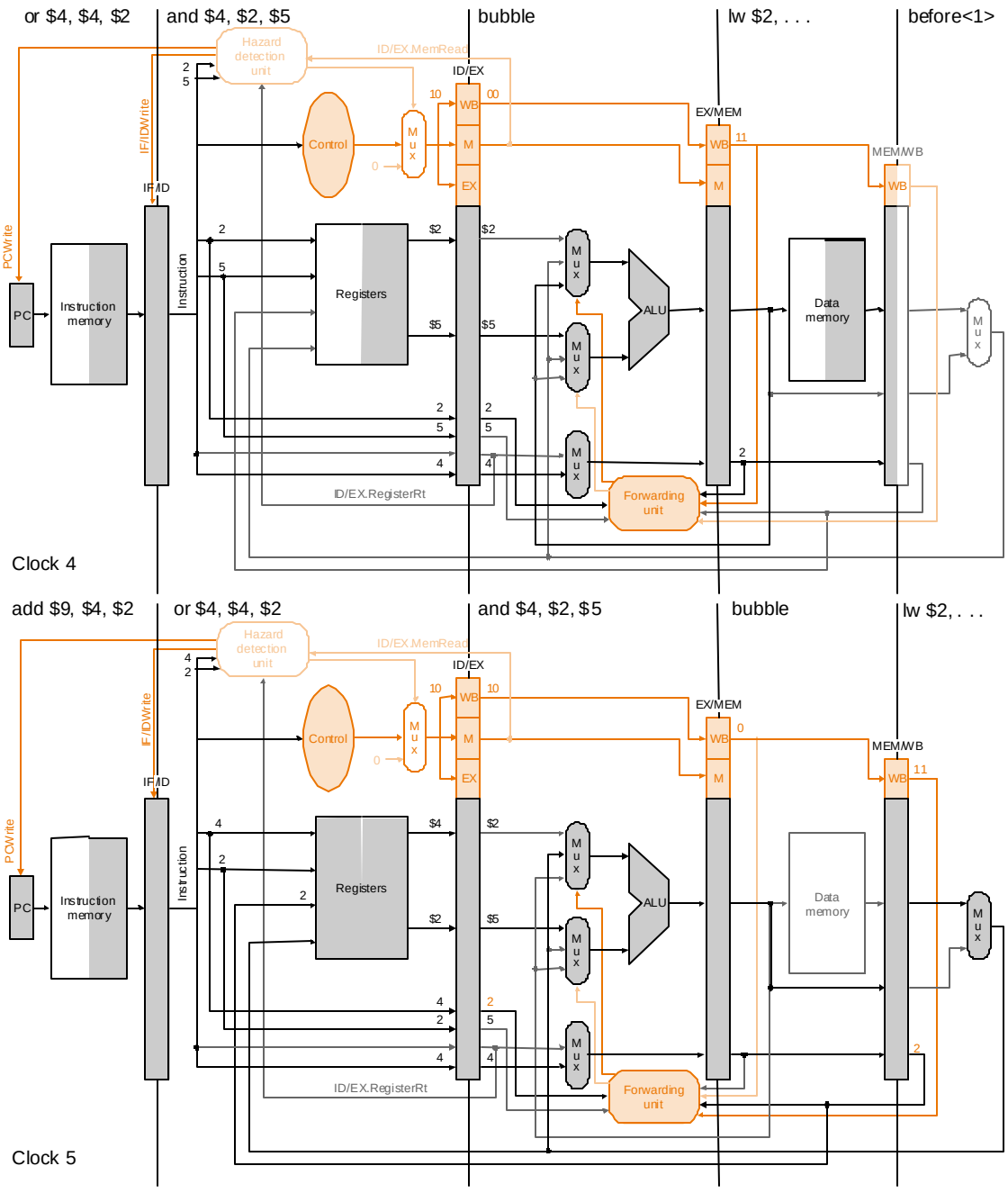
- **Stall pipeline ➔ mesmo efeito da instrução nop começando pelo estágio EX ➔ desativar os 9 sinais de controle dos estágios EX, MEM e WB**

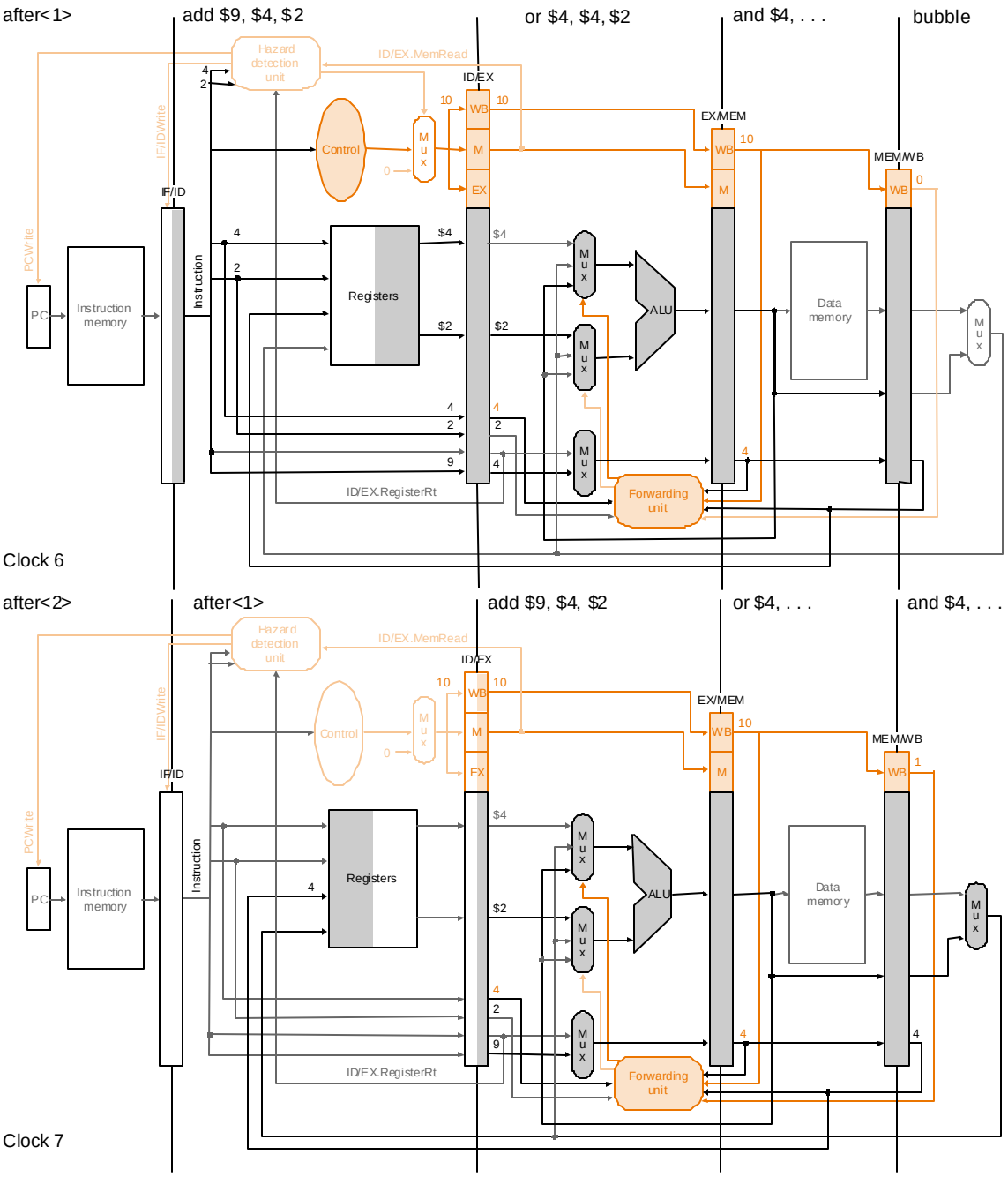
- **Datapath com forwarding e data hazard detection**



- **Seqüência de execução do exemplo anterior**

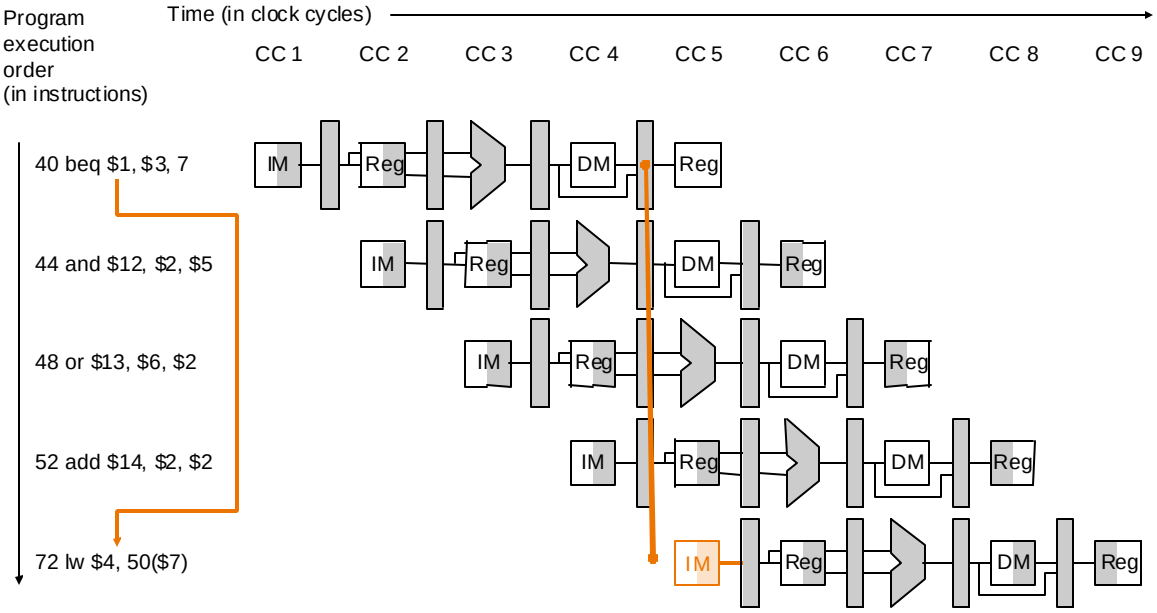



or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
before<1>
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/DWrite
PCWrite
IF/ID
Control
Mux
WB
M
EX
Instruction memory
PC
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 4
add $9, $4, $2
or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
Clock 5


after<1>
add $9, $4, $2
or $4, $4, $2
and $4, . . .
bubble
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/IDWrite
PCWrite
Control
WB
M
EX
IF/ID
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 6
after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
Clock 7

- **Branch Hazards**

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

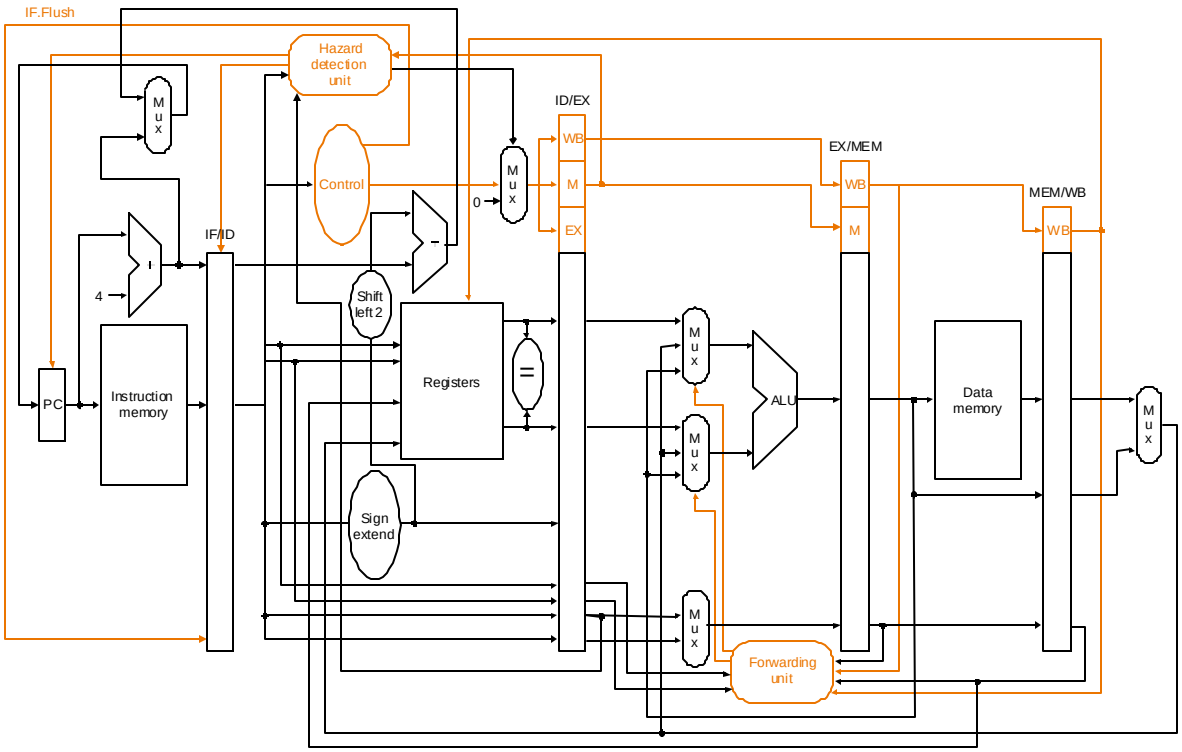


- **Dois esquemas para resolver control hazard : Assume Branch Not Taken e Dynamic Branch Prediction**

  - **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**

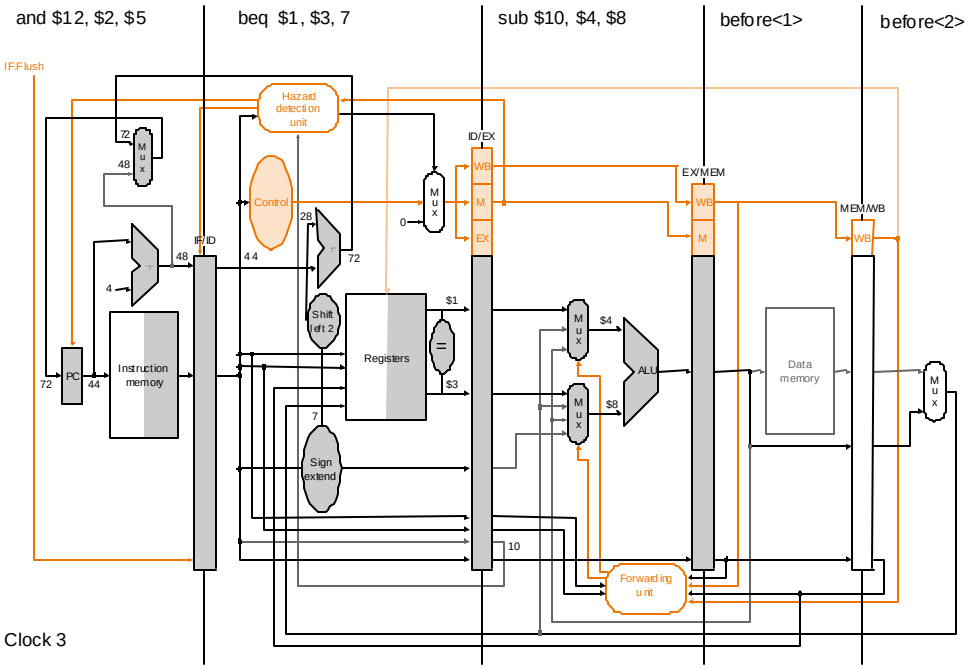
- **Redução do atraso de branches ➔ reduzir o custo se o branch for tomado ➔ adiantar a execução de uma instrução de branch.**

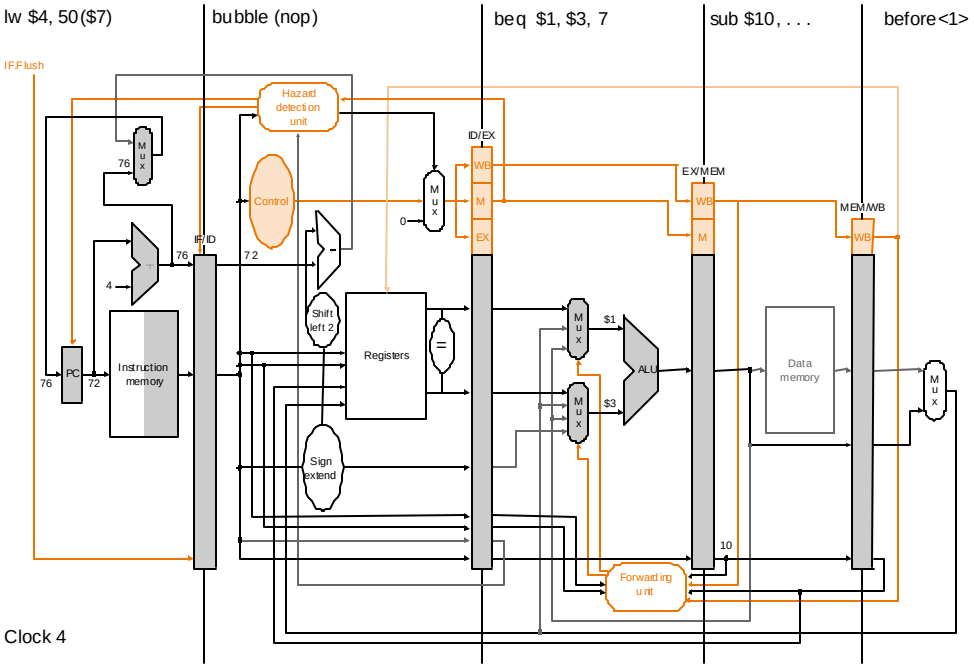
- **O next PC para uma instrução de branch é selecionado no estágio MEM ➔ executar o branch no estágio ID ➔ apenas uma instrução será descartada ➔ deslocar o cálculo do endereço de branch (branch adder) do MEM para o ID e comparando os registradores lidos do register file.**



- **Exemplo**

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





- **Dymanic Branch Prediction**

  - **Branch not taken ➔ forma de branch predicton ➔ prediction de branchs que não serão tomados.**

  - **Dymanic Branch Prediction ➔ antever o endereço da instrução se o branch foi tomado na última vez que foi executado começando o fetch das instruções pelo mesmo local da última vez.**

  - **Implementação ➔ branch prediction buffer ou 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.**

- **Dymanic Branch Prediction**

  - **Dynanic Branch Prediction ➔ antever o endereço da instrução se o branch foi tomado na última vez que foi executado começando o fetch da instrução pelo mesmo local da última vez.**

  - **Implementação ➔ branch prediction buffer ou branch history table.**

  - **Branch prediction buffer ➔ pequena memória indexada pelos bits menos significativos do endereço da instrução de branch. Ela contém um bit que diz se o branch foi recentemente tomado ou não.**

  - **Este esquema é uma implementação simples, mas não sabemos realmente 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.**

  - **Implementação com um bit ➔ se um branch é tomado sempre, teremos duas previsões incorretas**

Exemplo

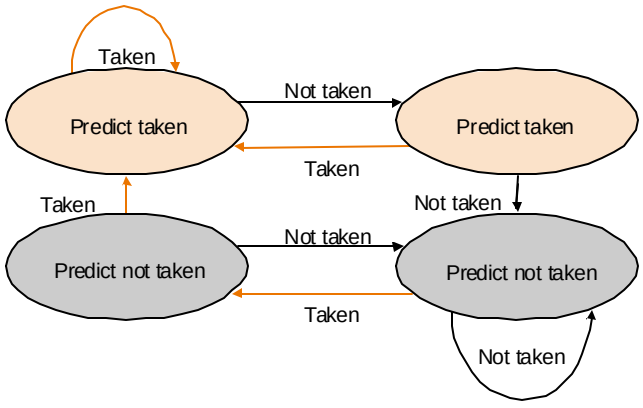
Um loop que toma um branch 9 vezes e uma vez não. Qual a precisão deste branch, assumindo um bit para a previsão no prediction buffer ?

Solução:

O comportamento será mispredict na primeira e última iteração. O último mispredict é inevitável pois o bit dirá que o branch foi tomado. Na primeira iteração ocorre porque o bit é alterado na execução anterior da última iteração. Então a precisão deste branch que é tomado 90% das vezes é de 80% (2 previsões incorretas e oito corretas)

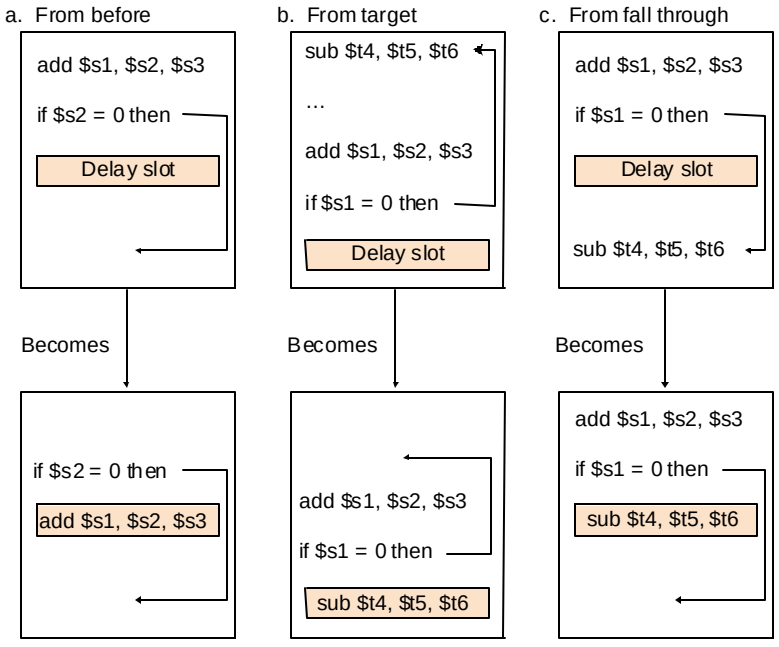
- **Esquema com dois bits ➔ A previsão deve estar errada duas vezes antes dela ser mudada**

- **Máquina de estados para esquema de previsão com 2 bits**



- **O branch prediction buffer pode ser implementado como um pequeno buffer especial contendo o endereço das instruções durante o estágio IF do pipeline. Se a instrução é prevista com tomada, o fetch se inicia da instrução alvo. Caso contrário o fetch e a execução seqüencial continuam.**

- **Delay Slot**



- **RESUMO DE PIPELINE**

  - **Comparação de Performance dos diversos esquemas de controle → single-cycle, multicycle e pipeline, usando as instruções de gcc. Tempo de operação de 2 ns para acesso à memória, 2 ns para operação da ULA e 1 ns para acesso ao register file. Para o pipeline, assumir que metade das instruções loads são seguidas imediatamente, por instruções que usam seu resultado, que o atraso em misprediction é de um ciclo de clock e ¼ dos branchs são mispredict. Assumir que jumps sempre tem 1 ciclo de atraso, portanto seu tempo médio é de 2 ciclos de clock.**

    Solução:

    Gcc → 22% de loads, 11% de stores, 49% de R-type, 16% de branchs e 2% de jumps.
    Para o pipeline, load leva 1 ciclo quando não há dependência e 2 quando há → média para load = 1.5 ciclos. Store e R-type levam 1 ciclo. Branch leva 1 quando a previsão é correta e 2 quando não → média de 1.25 ciclos. Jump leva 2.

    CPI = 1.5 X 22% + 1 X 11% + 1 X 49% + 1.25 X 16% + 2 X 2% = 1.17

    Tempo CPU = 1.17 X 2 ns = 2.34 ns (contra 4.04 ns para multicycles e 8 ns para single-cycles)

- **Control Hazards em Exceção**

  - **Supor que add $1,$2,$1 gera um overflow aritmético ➔ é necessário transferir o controle para uma rotina de tratamento (end. 4000 0040_{hex}), imediatamente após sua execução.**

  - **Temos que esvaziar o pipeline das instruções após o add e começar o fetch a partir do novo endereço. É o mesmo mecanismo de um branch tomado, só que as linhas de controle devem ser desativadas.**

  - **O estágio IF ➔ nop**

  - **O estágio ID ➔ um zero na entrada do MUX, que agora é controlado por um sinal que é um OR do sinal ID.Flush e a saída da unidade de detecção de hazard.**

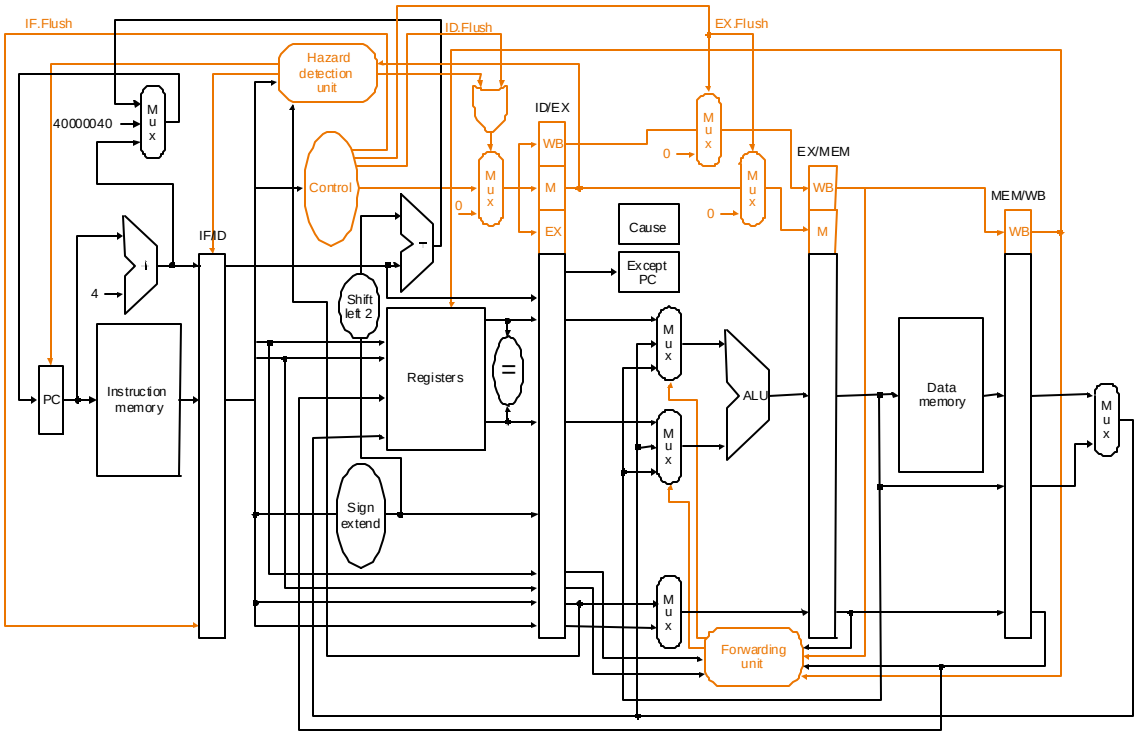
  - **O estágio EX ➔ usamos um sinal EX.Flush nos dois MUXes, para selecionar zero para os sinais de controle.**

  - **Adicionamos uma entrada no PC (endereço da rotina de exceção).**

  - **O estágio WB armazena o resultado de $1**

  - **Endereço da instrução + 4 é armazenado em EPC (Exception Program Counter)**

- **Datapath com controle de tratamento de exceção**



- **Exemplo:**

Dado a seqüência abaixo:

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

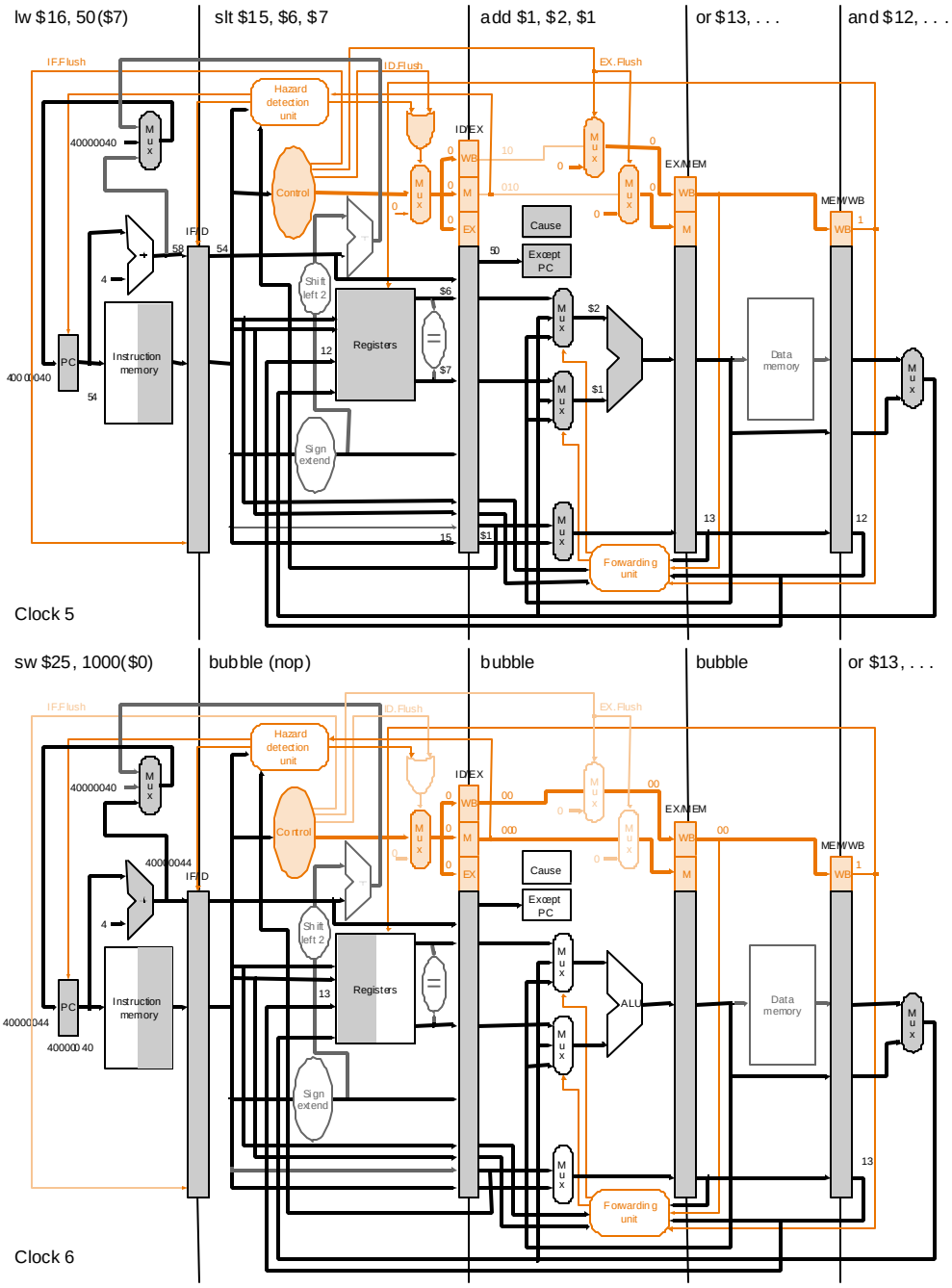
Assumir que as instruções a serem chamadas em um tratamento de exceção comecem com:

40000040_{hex}	sw	$25,	1000($0)
40000044_{hex}	sw	$26,	1004($0)

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

Solução:

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



- **Pipeline Superscalar e Dinâmico**

  - **Extensão do pipeline para torná-lo mais rápido:**

    - **superpipeline ➔ pipelines longos, ou seja, pipeline com mais estágios.**

    - **superscalar ➔ replicação de componentes internos.**

    - **pipeline dinâmico ➔ dynamic pipeline scheduling.**

      Exemplo:

```
lw      $t0, 0($s2)
addu    $t1, $t0, $t2   # data hazard
sub     $s4, $s4, $t3
slti    $t5, $s4, $t3
```

      As instruções sub e slti não precisam esperar addu e lw serem executadas, elas podem ser executadas em paralelo.

  - **Superscalar MIPS**

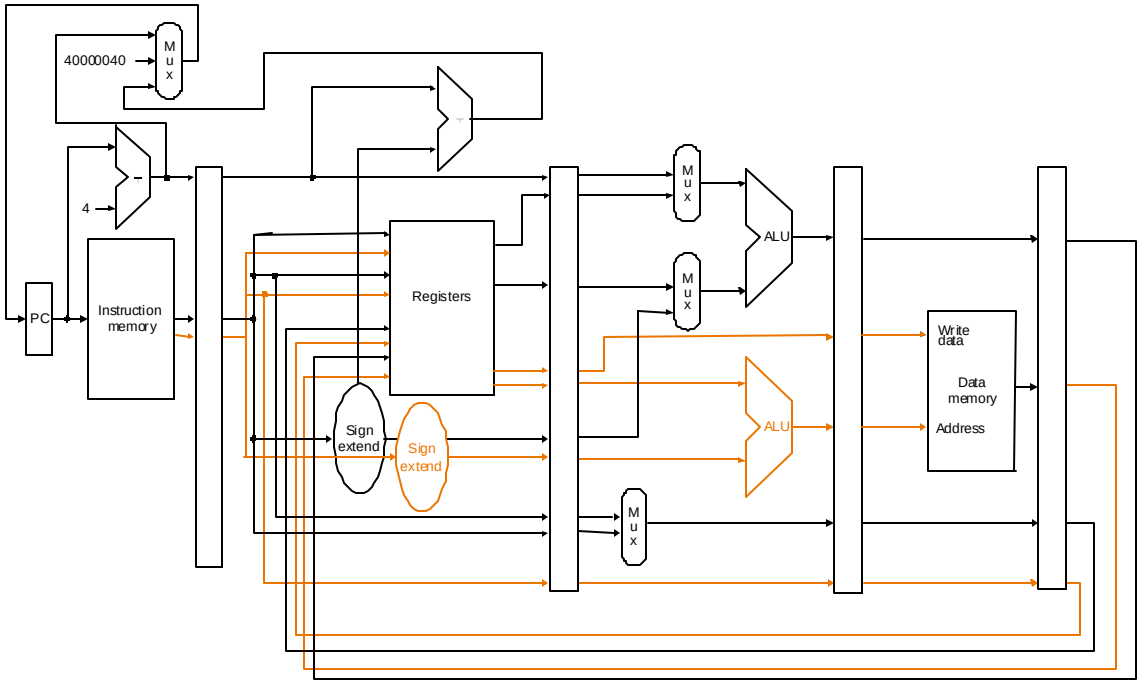
    - **Assumir duas instruções por ciclo de clock ➔ uma que faça uma operação com a ULA ou um branch e outra um load ou store**

    - **Duas instruções por ciclo ➔ fetch e decode 64 bits de instrução ➔ alinhamento por par de instruções e de 64 bits**

- **Operação em um pipeline superscalar**

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

- **Superscalar datapath**



- **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 como possível.

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

- **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

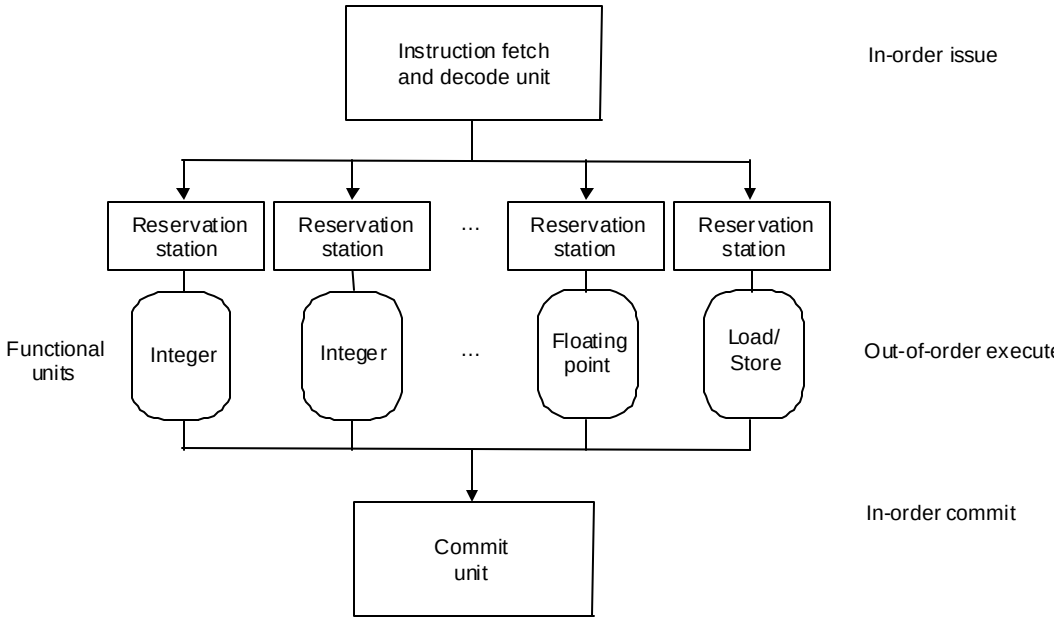
  Solução ➔ 4 cópias do corpo do loop

	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

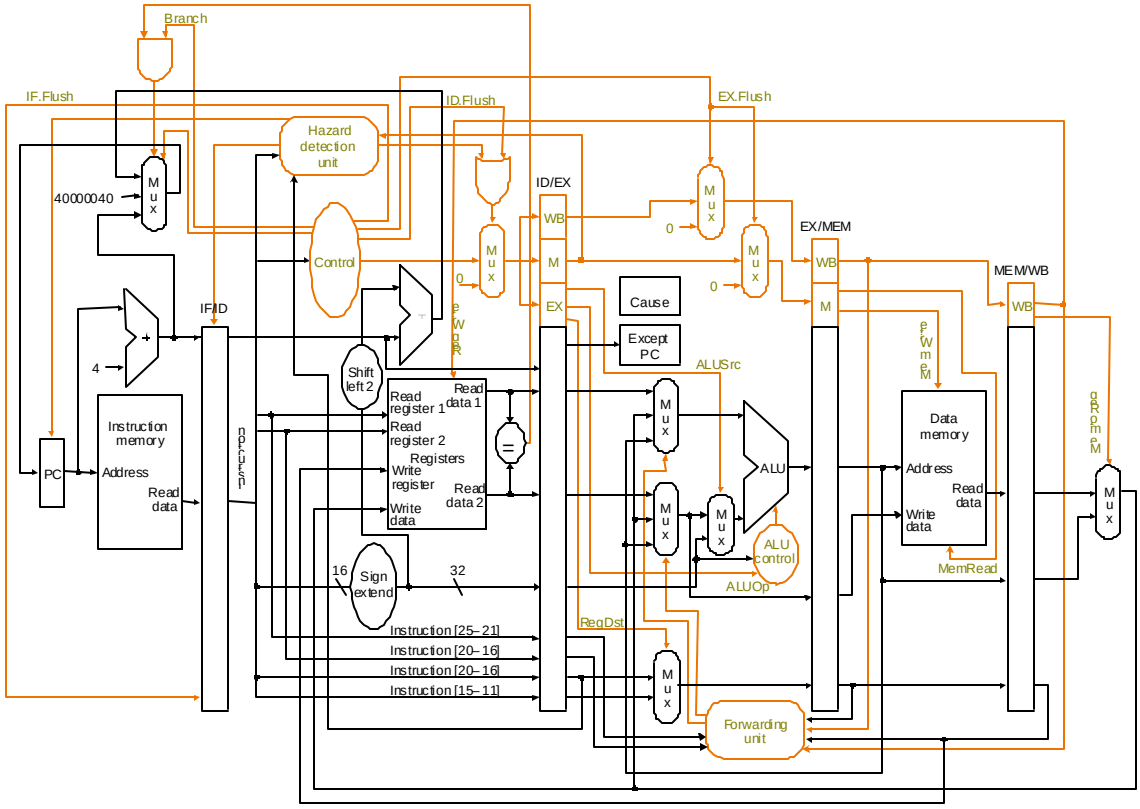
- **Dynamic Pipeline Scheduling**

- **Pipeline dividido em: instruction fetch and issue unit, execute units e commit units**

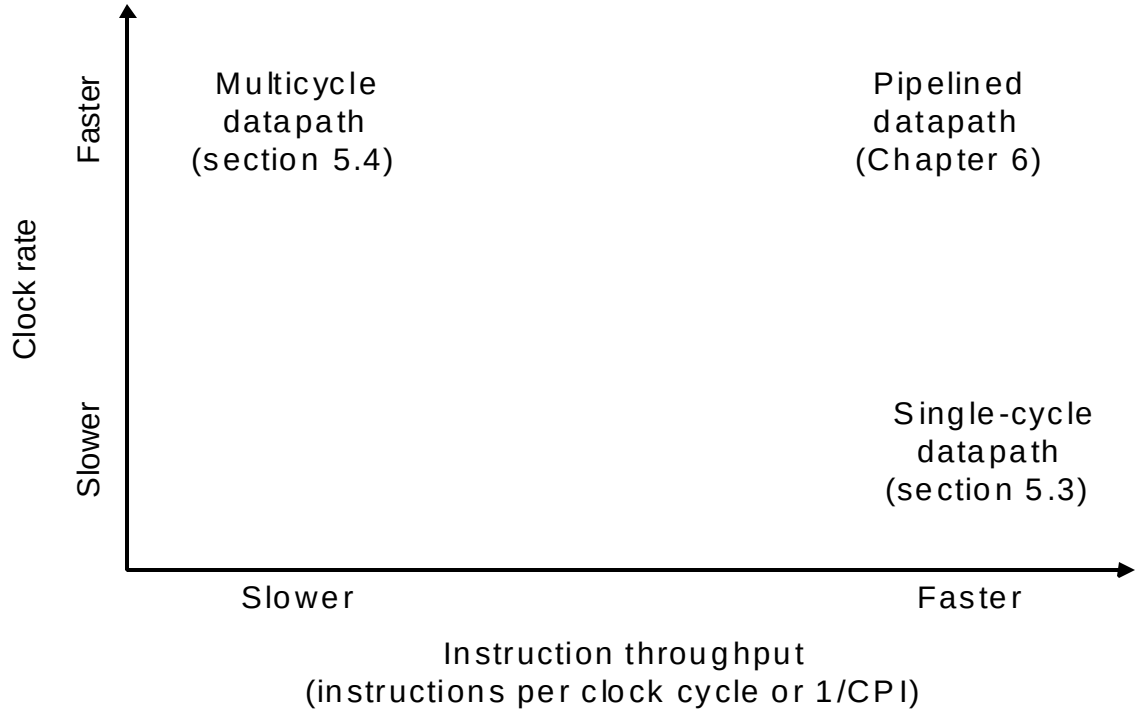
- **A primeira unidade faz o fetch das instruções, decodifica-as e envia-as, cada uma para uma unidade funcional. Cada unidade funcional tem um buffer (reservation station) onde os operandos e a operação são armazenados. O resultado é calculado nas unidades funcionais e a commit unit decide quando guardar o resultado no register file ou na memória.**



- **Datapath final**



- **Análise de performance single-cycle X pipeline**



- **Relações entre os diversos Datapath**