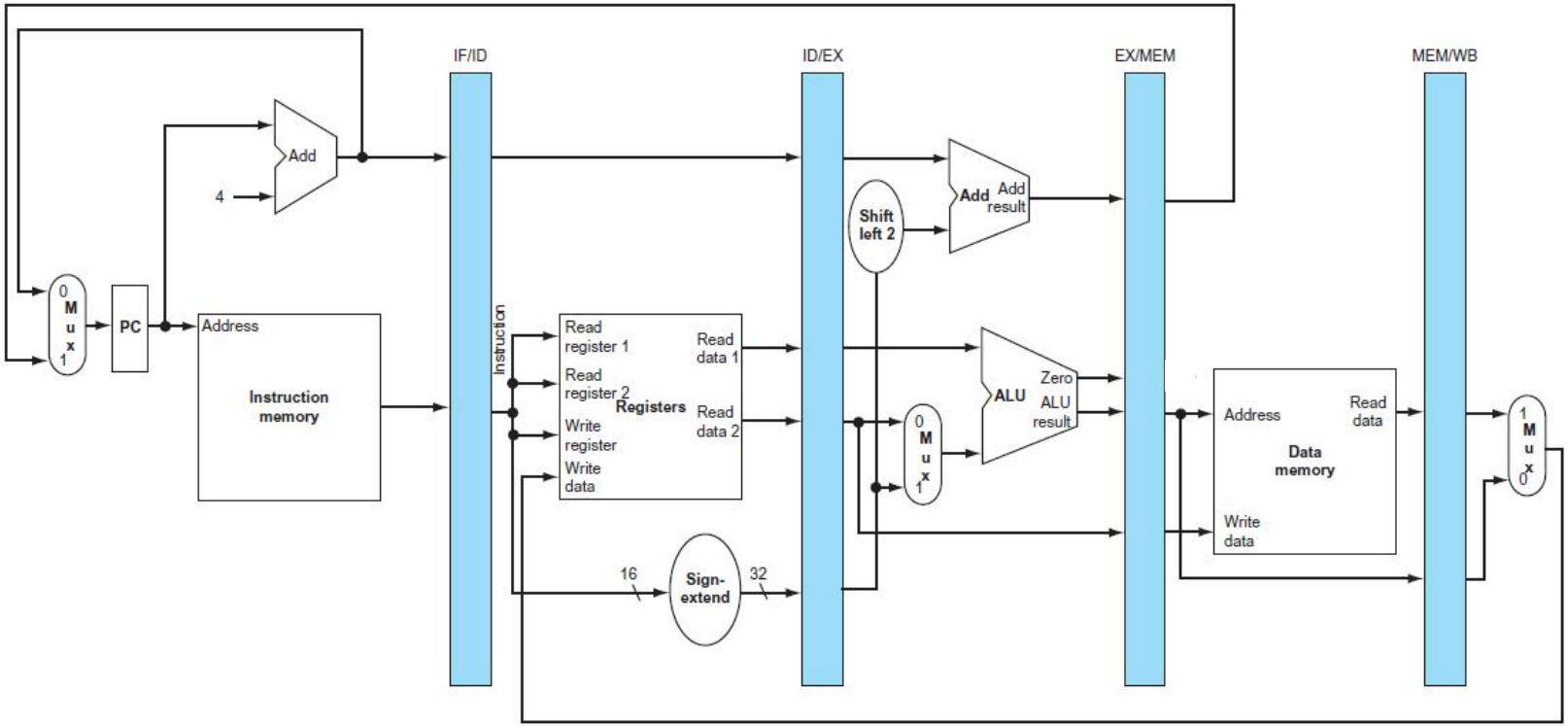
Conflitos e Controle da CPU Pipeline

Anotações do material suplementar (apresentações PPT) ao Livro do Hennessy e Patterson e do material do Prof. Celso Alberto Saibel Santos (DI/CT).

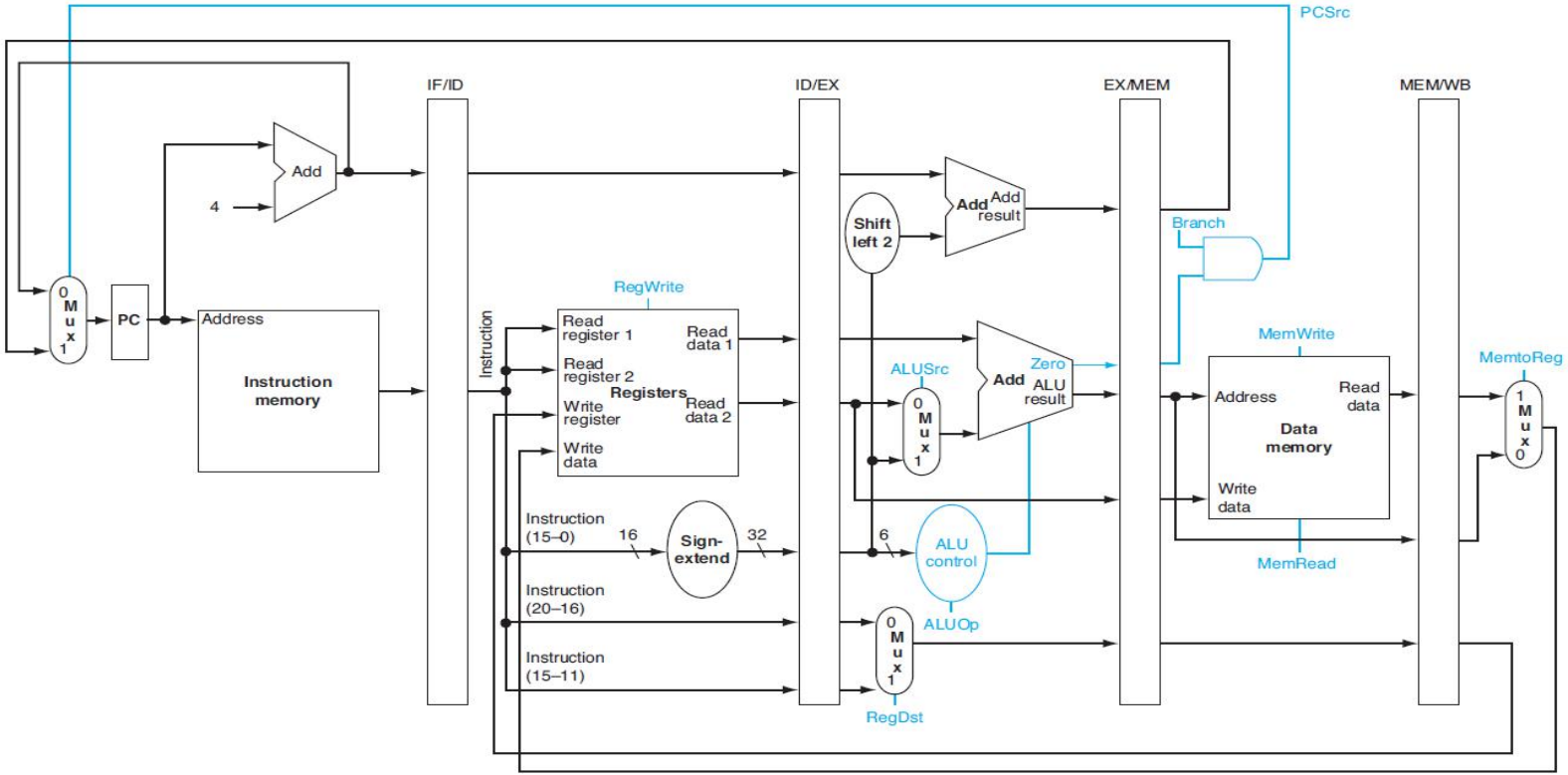
Controle do Pipeline

Aulas anteriores...

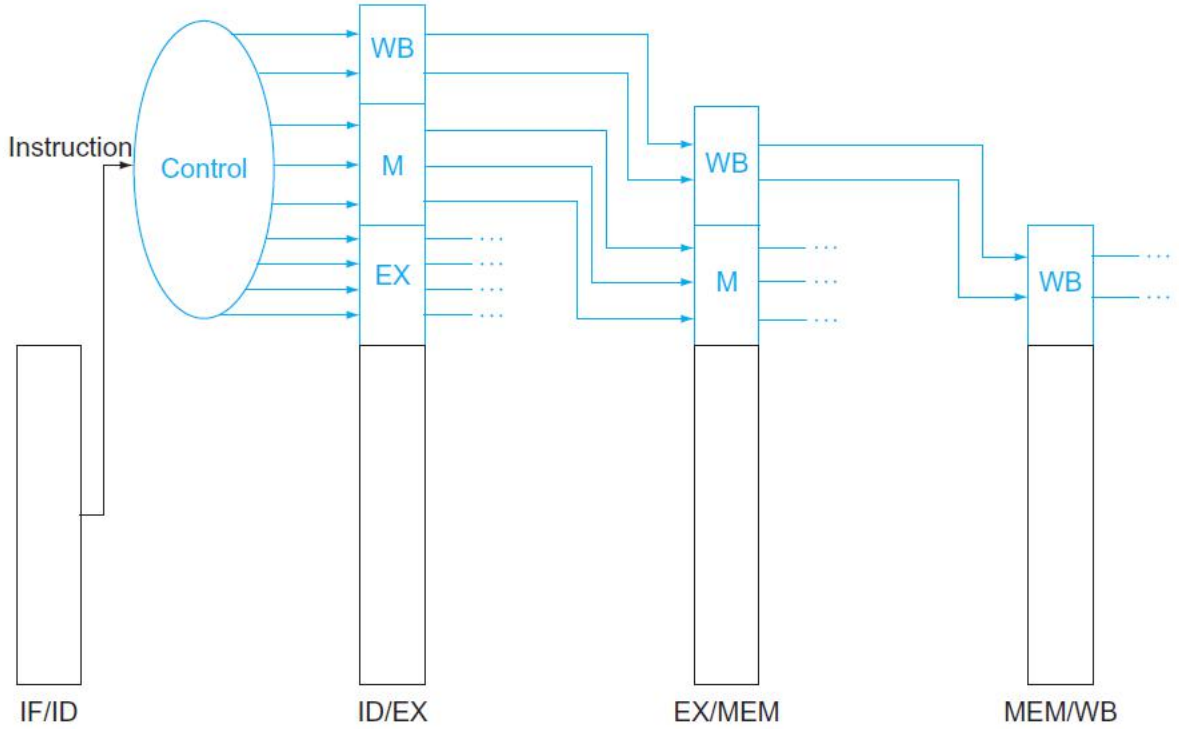
Via de Dados da CPU Pipeline

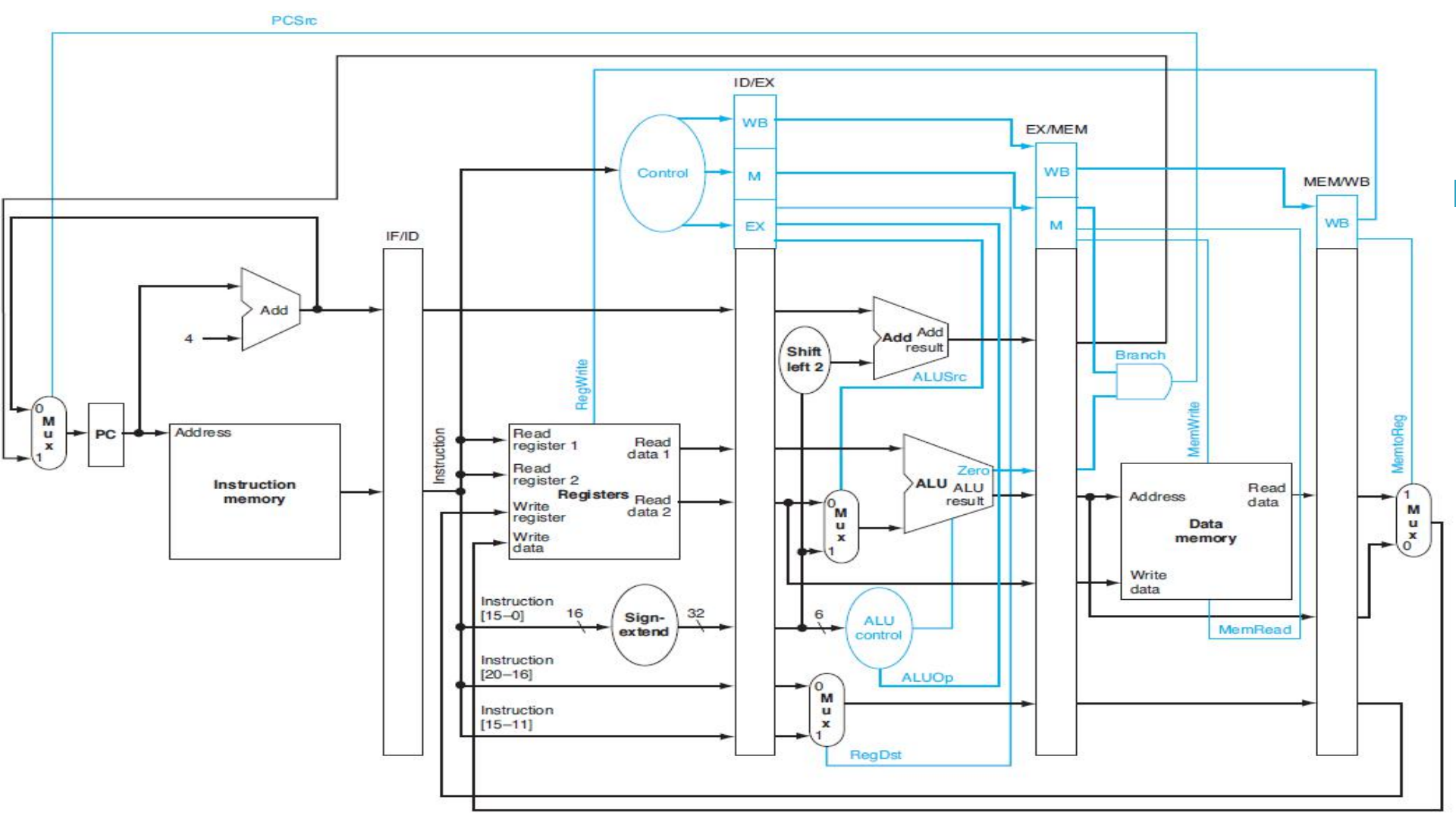


Sinais de Controle da CPU Monociclo



Sinais de Controle dos 3 últimos estágios da CPU pipeline




PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Control
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Branch
ALU
Zero
ALU result
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign-extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst
Mux
0
1

Conflitos (*hazards*) no pipeline

Tipos

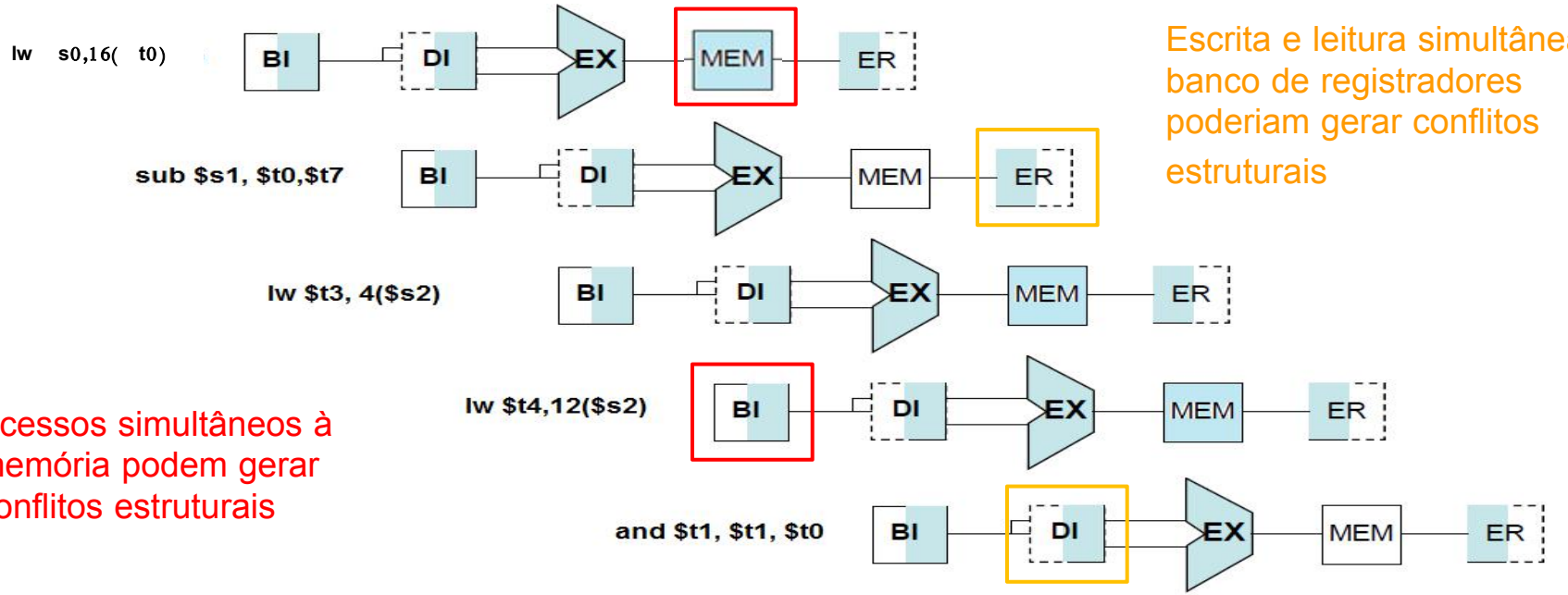
- Existem situações de execução no pipeline em que a instrução seguinte não pode ser executada no próximo ciclo de relógio:
  - Conflitos (*hazards*) de pipeline!
- Podem causados por:
  1. Problemas estruturais (concorrência por recurso);
  2. Mudança de fluxo de controle;
  3. Dependência de dados.

Conflitos ESTRUTURAIS

Conflitos estruturais (*structural hazards*)

- O hardware poderia não suportar a combinação de instruções que entraram no pipeline em um dado ciclo de relógio.
  - Como o conjunto de instruções MIPS foi projetado para permitir a execução em pipeline, evitar conflitos estruturais é uma tarefa simples;
- Usar duas memórias (dados e instruções) evita um conflito estrutural "óbvio":
  - Na memória principal ambos os segmentos estão "juntos"...
- A escrita e a leitura no banco de registradores podem ser requisitadas simultaneamente por duas instruções em fases diferentes no pipeline:
  - Escrita na 1ª metade e leitura na 2ª metade do ciclo evitam o conflito

Solucionando conflitos estruturais



Escrita e leitura simultâneas no banco de registradores poderiam gerar conflitos estruturais

Acessos simultâneos à memória podem gerar conflitos estruturais

Solução 1: duas memórias separadas (dados e instruções)

Solução 2: escrita na 1ª metade do ciclo e leitura na 2ª metade do ciclo de clock

Conflitos DE DADOS

Conflitos de dados

- Causa: A execução de uma instrução depende de um resultado (normalmente, um operando) de outra, cuja execução ainda não foi concluída:

  - Instruções aritméticas em sequência, construções do tipo "load" seguido de uso de Reg em uma instrução tipo "R", ex:

```
lw $10, 4($9)

add $11, $10, $7
```

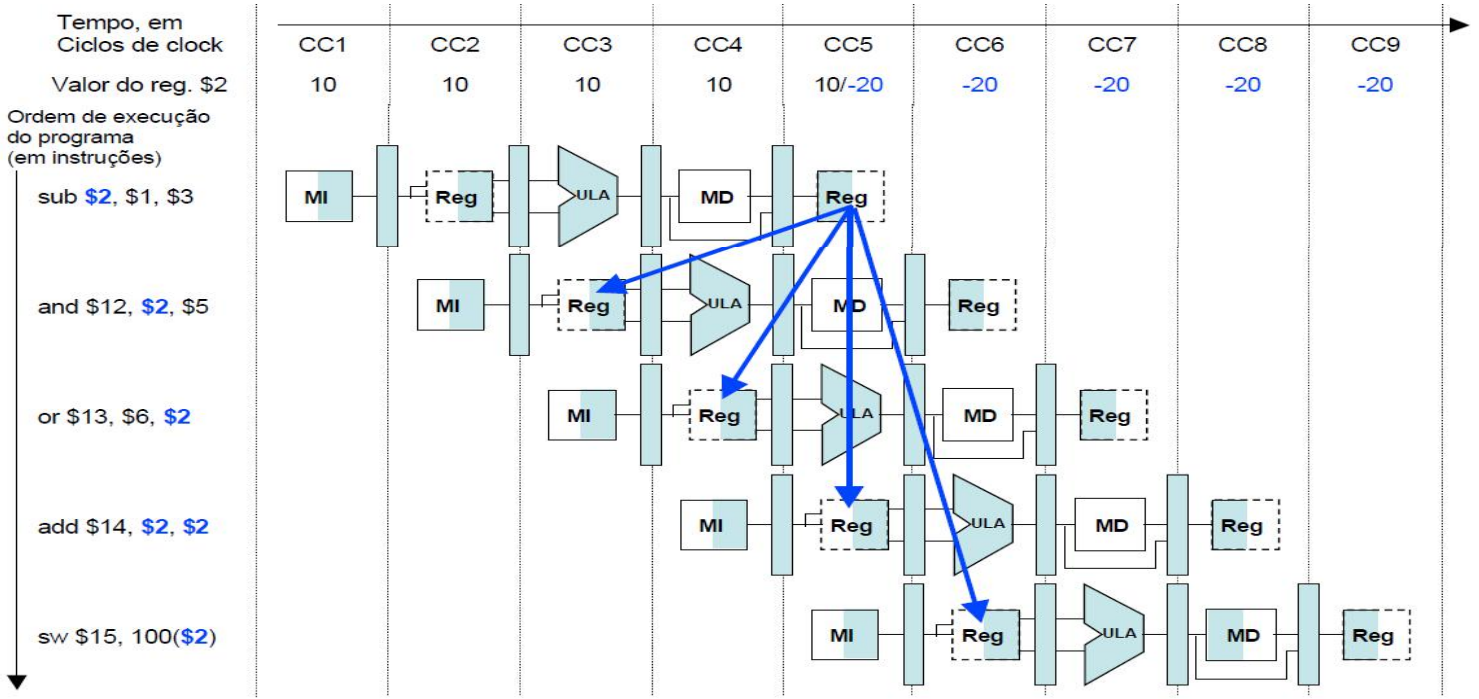
- Nem sempre é possível passar a solução para o compilador porque este tipo de conflito é muito frequente!

Exemplo de conflito de dados

- Considere a sequência de instruções MIPS:

```
sub   $2, $1, $3     # registrador $2 é escrito pela instrução sub
and   $12, $2, $5    # primeiro operando ($2) depende de sub
or    $13, $6, $2    # segundo operando ($2) depende de sub
add   $14, $2, $2    # primeiro e segundo operandos ($2) dependem de sub
sw    $15, 100($2)   # base ($2) depende de sub
```

Conflito de dados no pipeline



Solução simples: Compilador insere NOP's

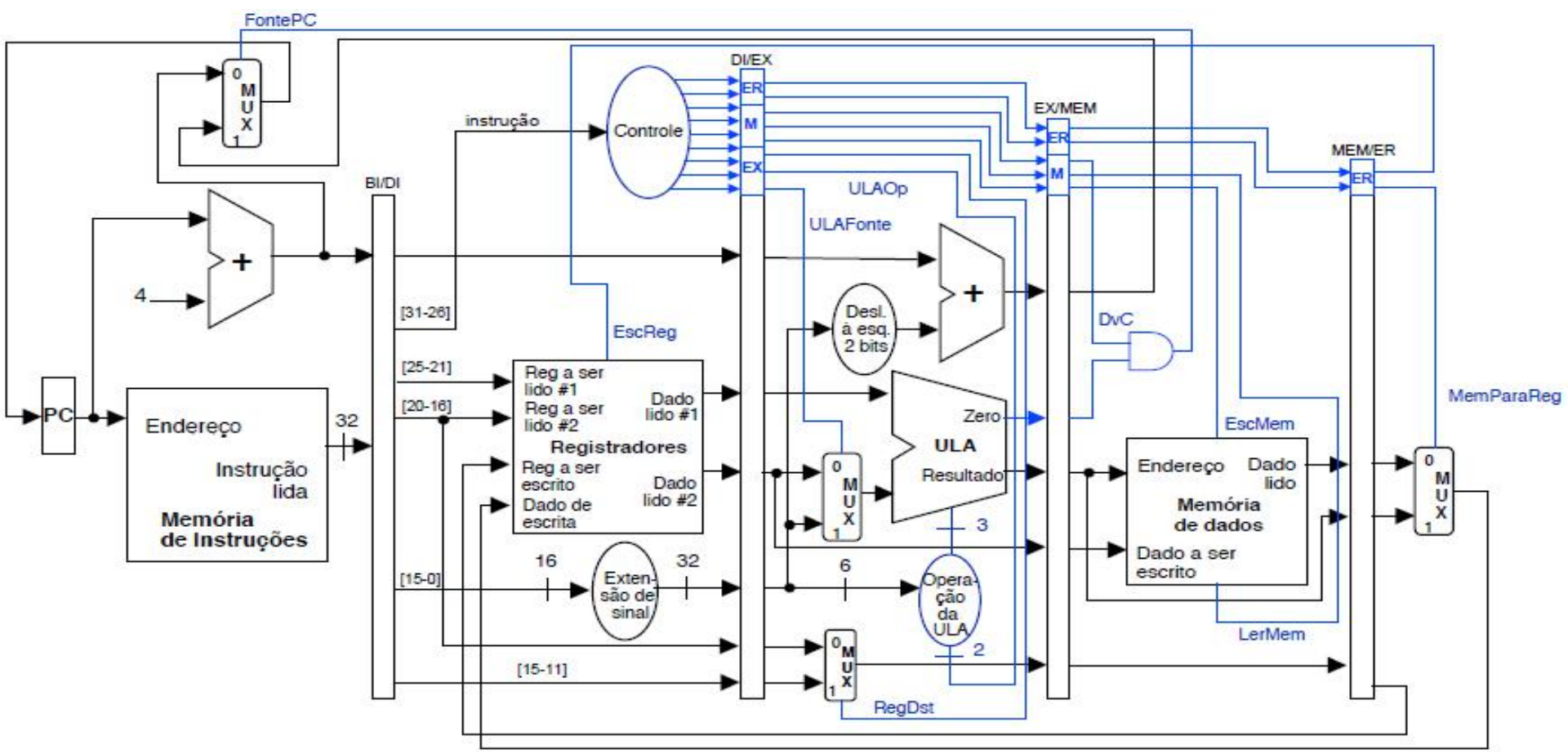
```
sub   $2, $1, $3      # registrador $2 é escrito pela instrução sub
nop                   # na falta de instruções que sejam independentes, o
nop                   # compilador inseriria instruções "nop"
and   $12, $2, $5     # primeiro operando ($2) depende de sub
or    $13, $6, $2     # segundo operando ($2) depende de sub
add   $14, $2, $2     # primeiro e segundo operandos ($2) dependem de sub
sw    $15, 100($2)    # base ($2) depende de sub
```

NOP = *NO OPeration*
→ parada do *pipeline*

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000

Outra solução: adiantamento de dados

- Problemas da solução com NOPs:
  - Frequência desses conflitos;
  - Perda óbvia de desempenho com paradas do pipeline.
- **Melhor solução**: Detectar o conflito e, se necessário, adiantar os dados vindos da ULA ou da memória de dados via hardware:
  - **Comparar registradores entre estágios** para detecção dos conflitos;
  - **Usar a saída da comparação para escolher entradas adequadas** para a escolha de operandos nas instruções subsequentes.



Testes ➡

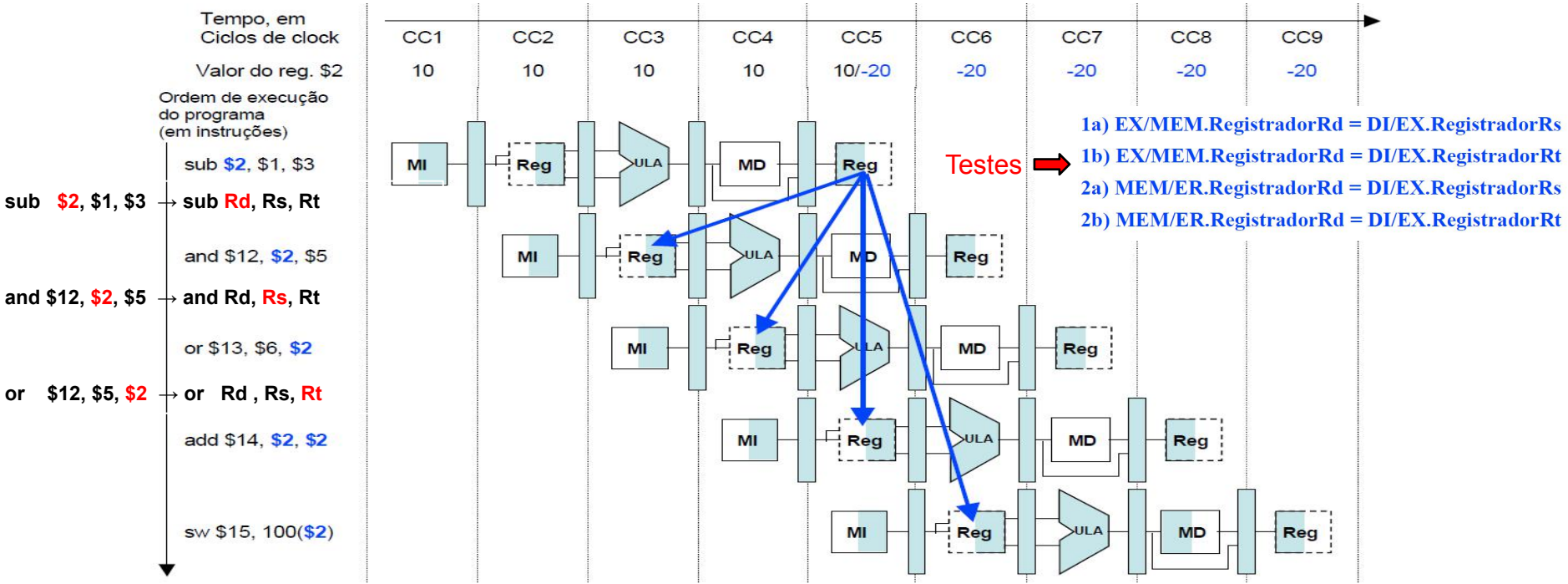
1a) EX/MEM.RegistradorRd = DI/EX.RegistradorRs

1b) EX/MEM.RegistradorRd = DI/EX.RegistradorRt

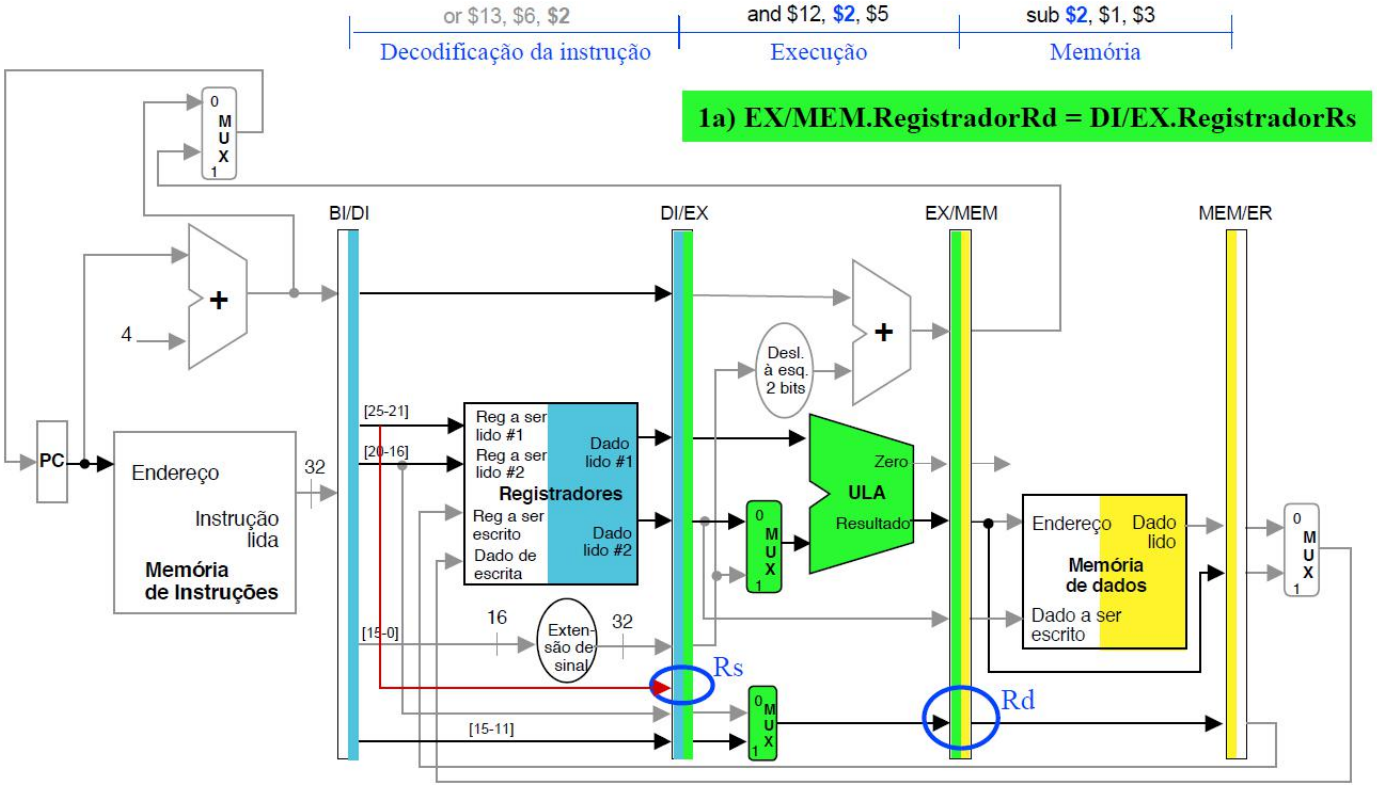
2a) MEM/ER.RegistradorRd = DI/EX.RegistradorRs

2b) MEM/ER.RegistradorRd = DI/EX.RegistradorRt

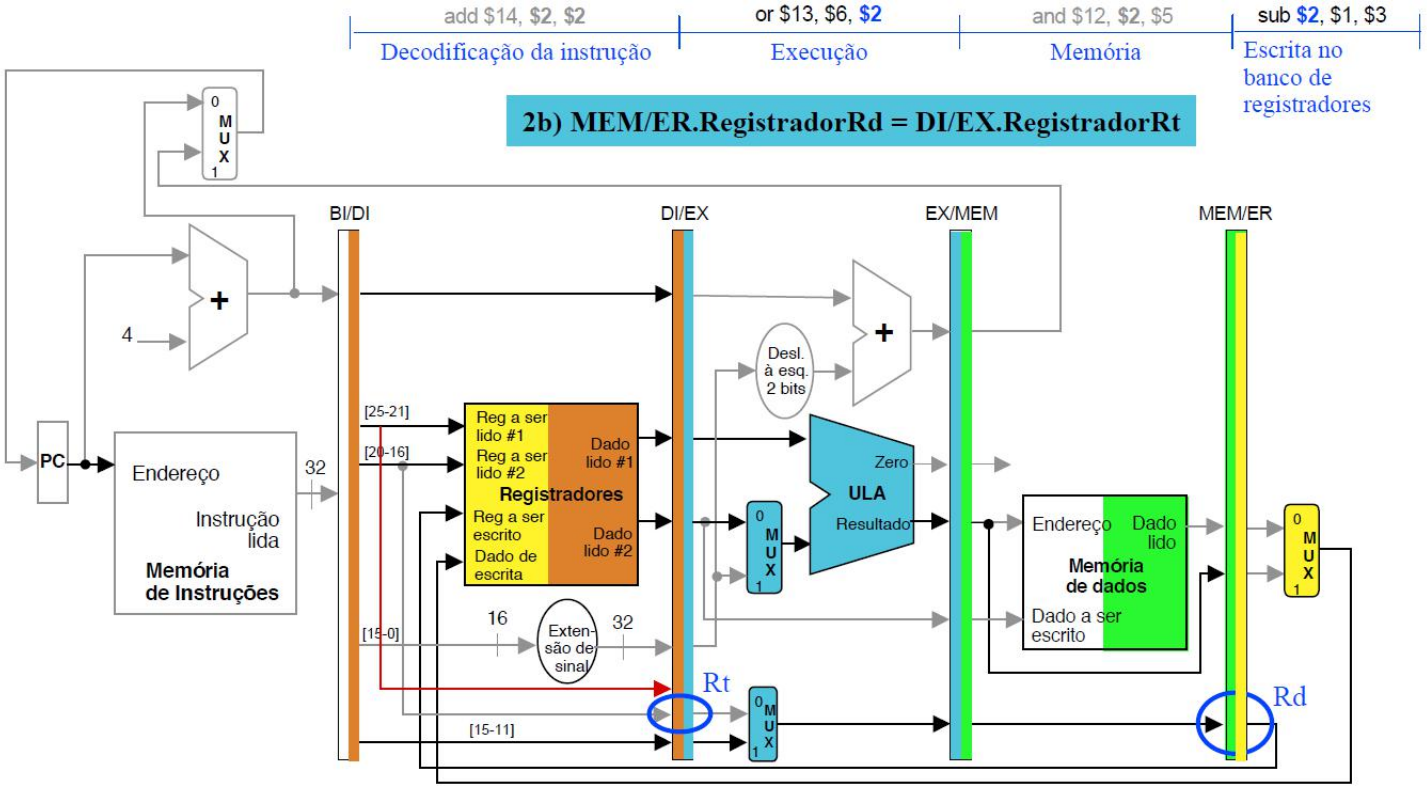
Conflito de dados no pipeline



Execução na CPU



Execução na CPU



Execução na CPU

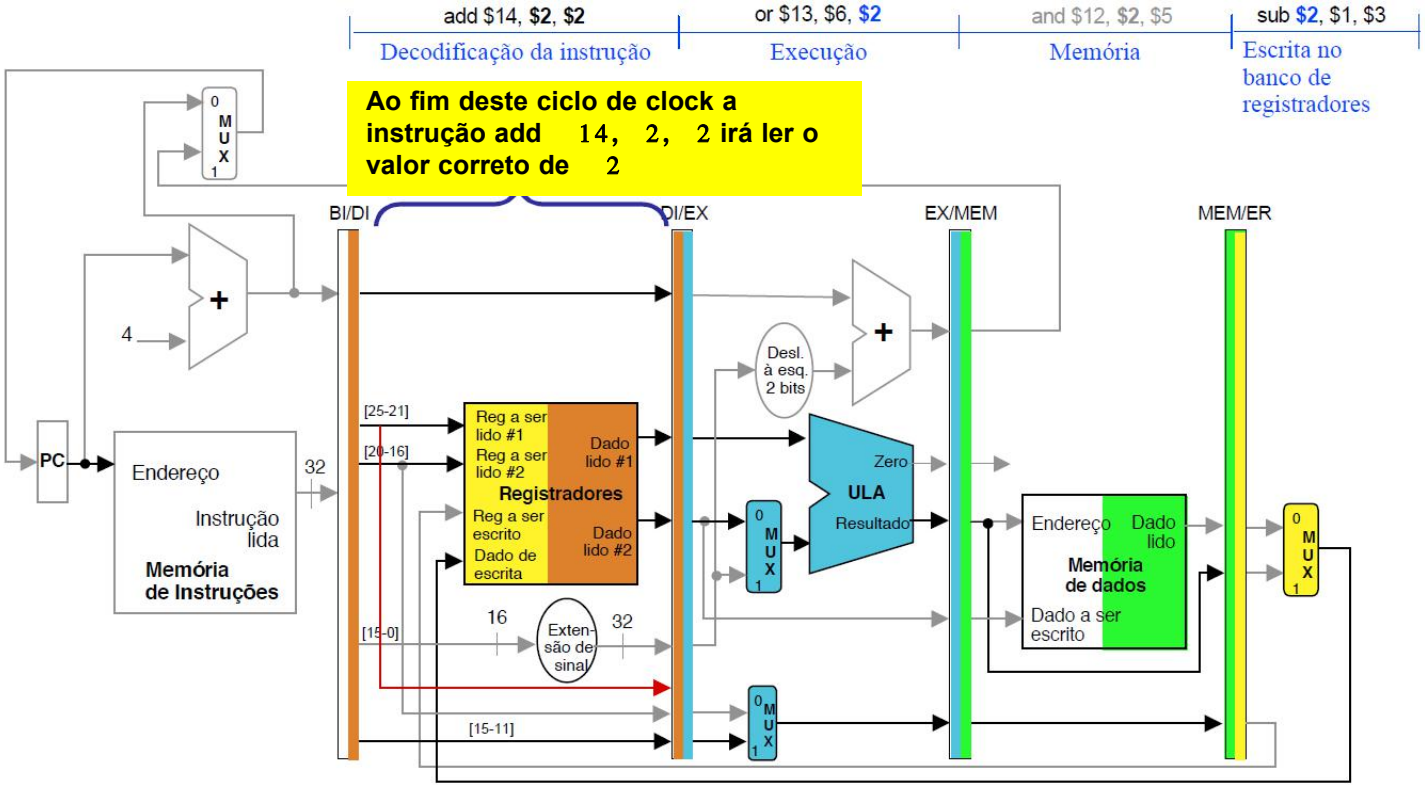
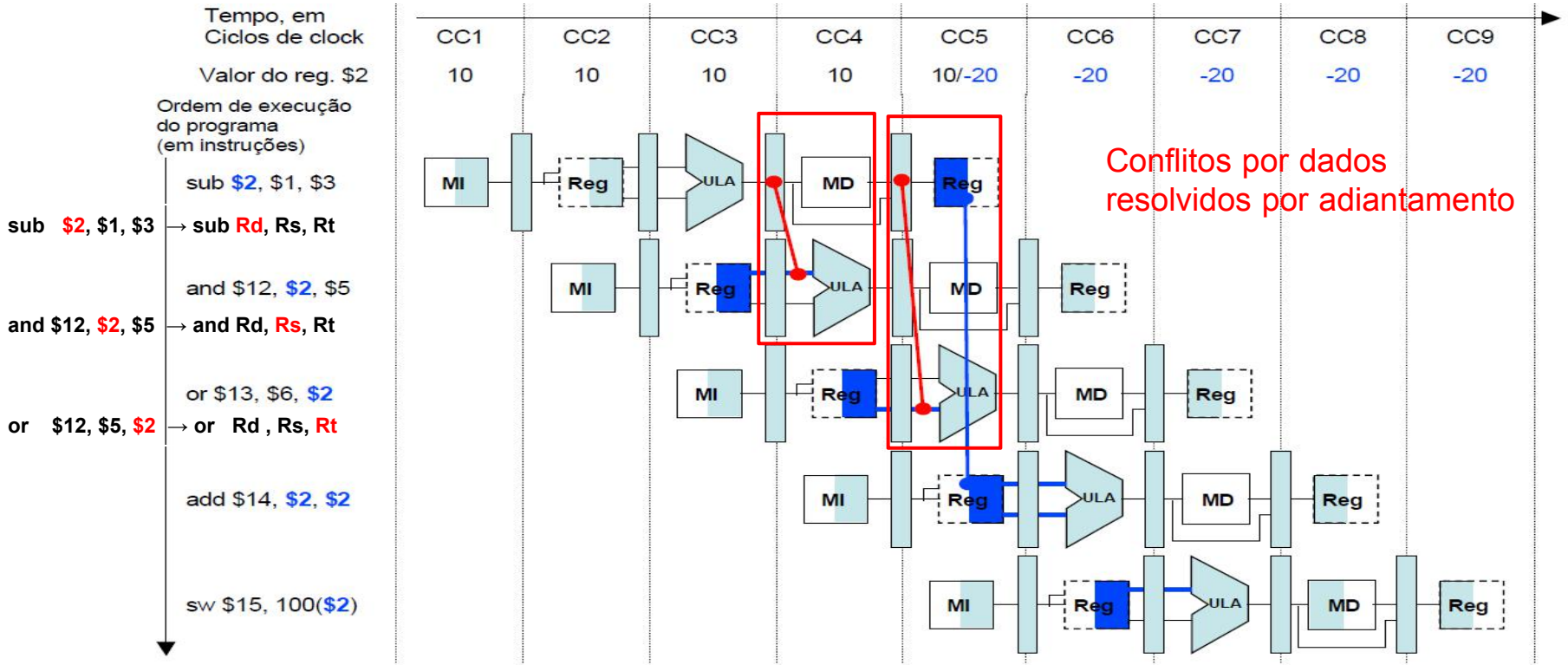
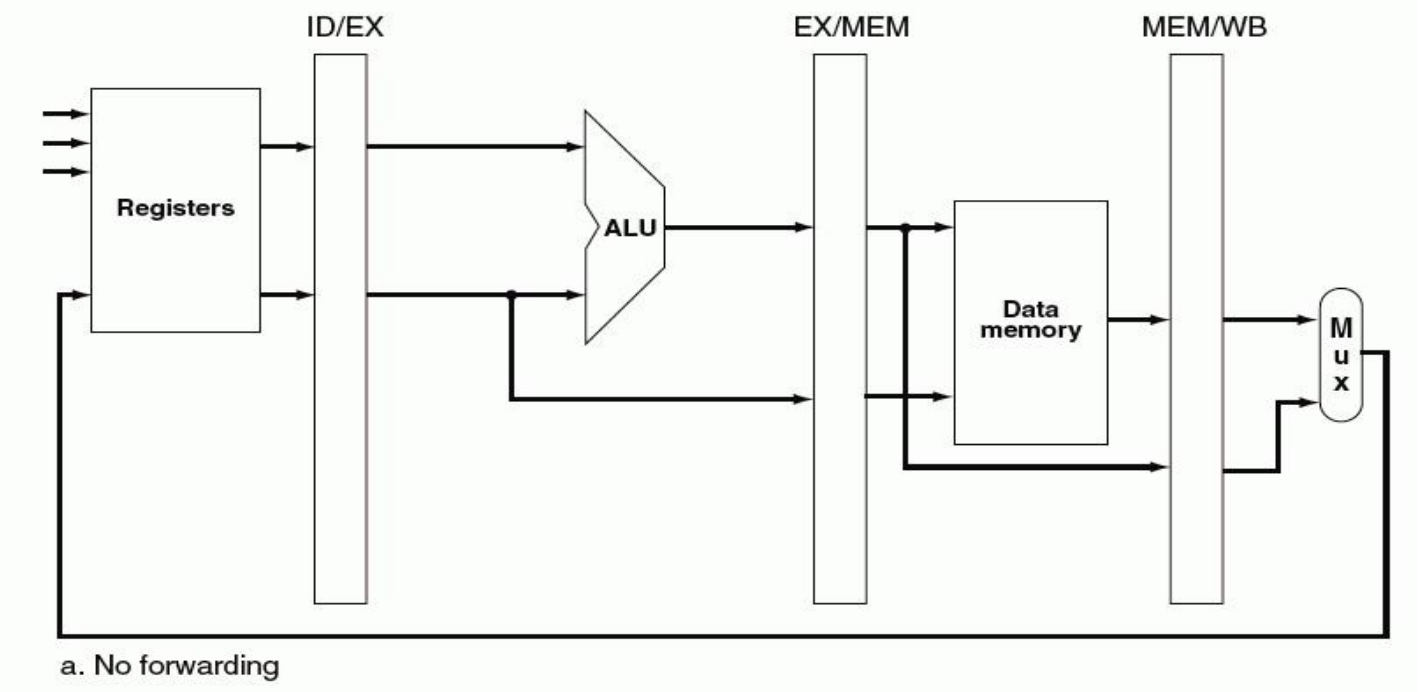


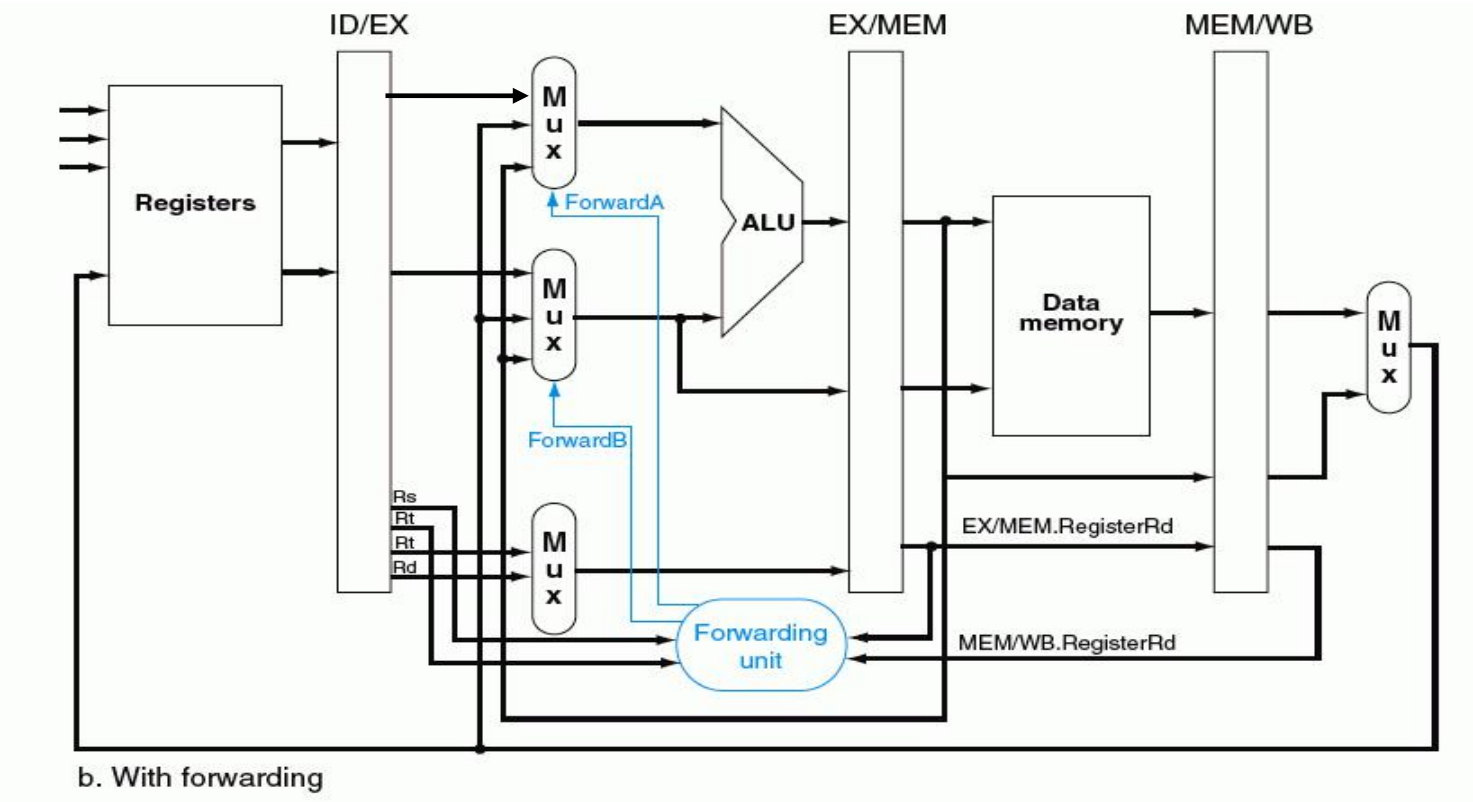
Diagrama de execução



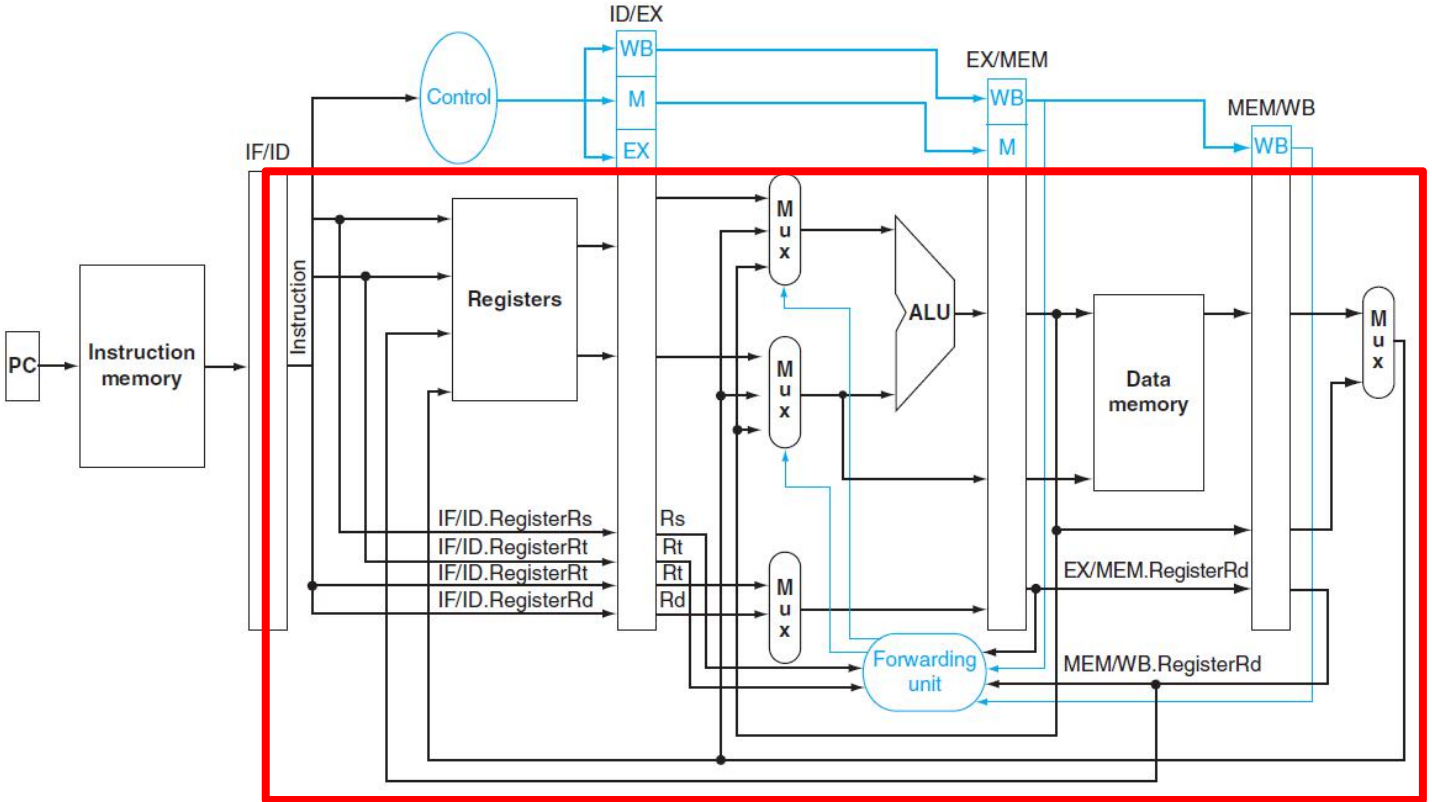
CPU Pipeline sem adiantamento



CPU Pipeline com adiantamento



CPU Pipeline com controle de adiantamento



Sinais de controle para adiantamento

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Casos não resolvidos por adiantamento

Conflitos não resolvidos por adiantamento

- Esta sequência de instruções MIPS possui dependências de dados que não são resolvidas pelo adiantamento de dados:

```
lw    $2, 20($1)
and   $4, $2, $5
or    $8, $2 , $6
add   $9, $4, $2

slt   $1, $6, $7
```

Diagrama de execução no pipeline

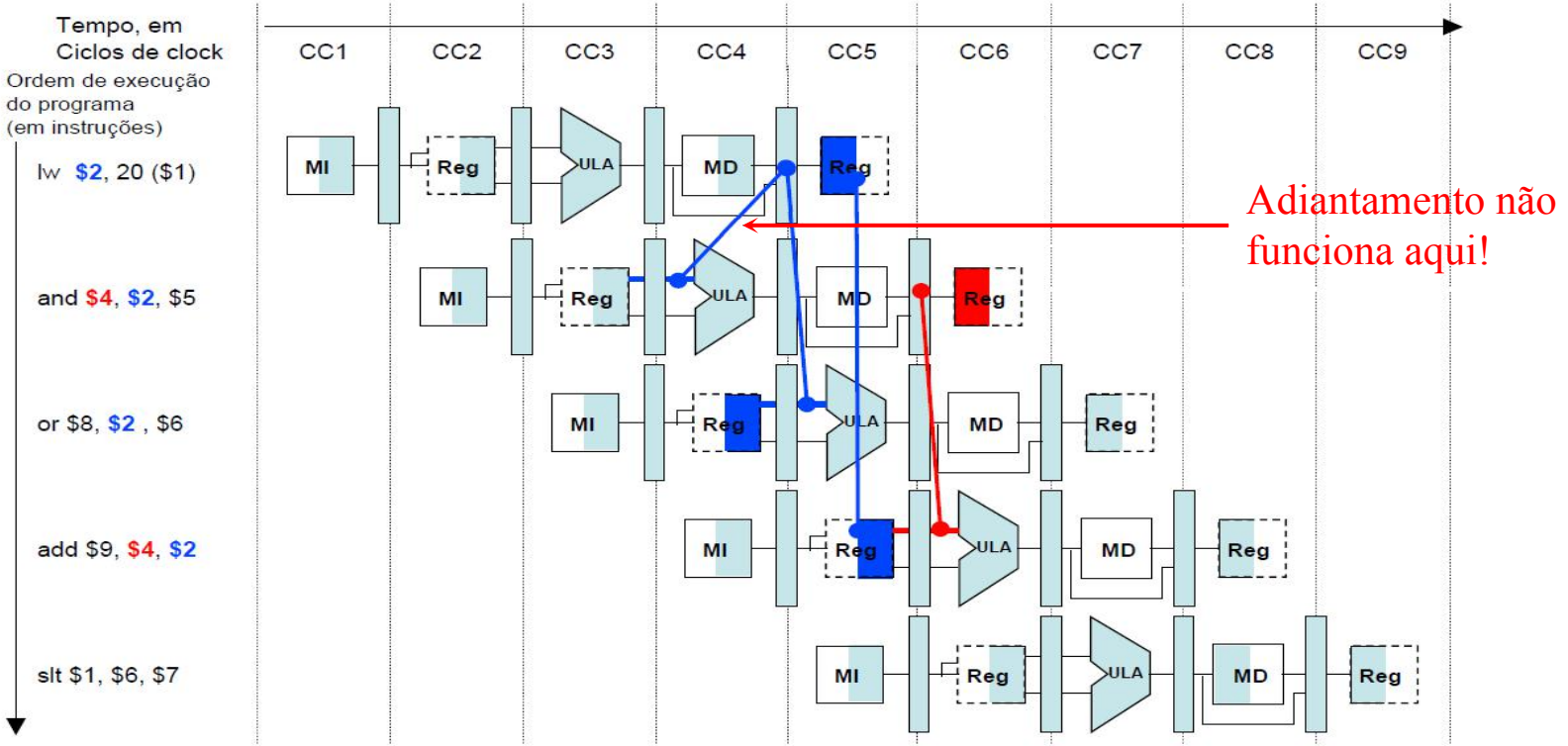
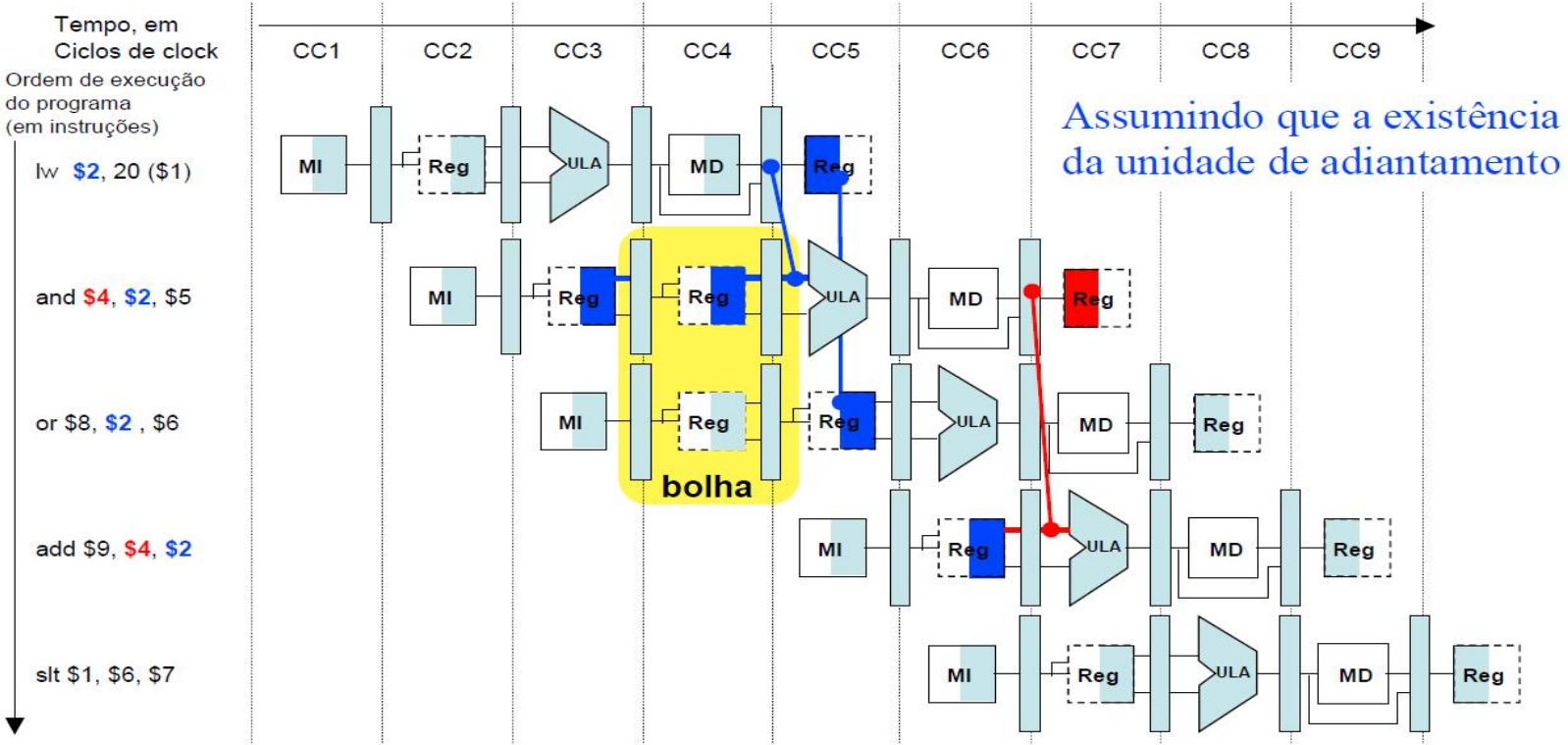


Diagrama de execução no pipeline



Parada no pipeline

Detecção de Conflito pelo controle da CPU:

1. Faz o pipeline parar quando houver uma construção do tipo "*lw* seguida de uma instrução que deve ler o mesmo registrador escrito pela instrução de *lw*"
2. Vai operar durante o estágio DI, inserindo uma parada entre a instrução *lw* e a instrução seguinte que usa seu resultado

- A condição a ser verificada para parar (*stall*) o pipeline agora é a seguinte:

Se (DI/EX.LerMem = 1 E
((DI/EX.RegistradorRt = BI/DI.RegistradorRs) OU
(DI/EX.RegistradorRt = BI/DI.RegistradorRt)))
Então para o pipeline por um ciclo de relógio

Parando todo o pipeline após o conflito

- Se a instrução que está no estágio DI estiver parada, então o estágio BI (anterior) também precisa parar;

- Impedir o avanço de instruções pelo pipeline = evitar que PC e o registrador BI/DI sejam escritos/atualizados. Com isso, no ciclo de relógio seguinte à detecção do conflito:

1. A instrução que está no BI será novamente lida;

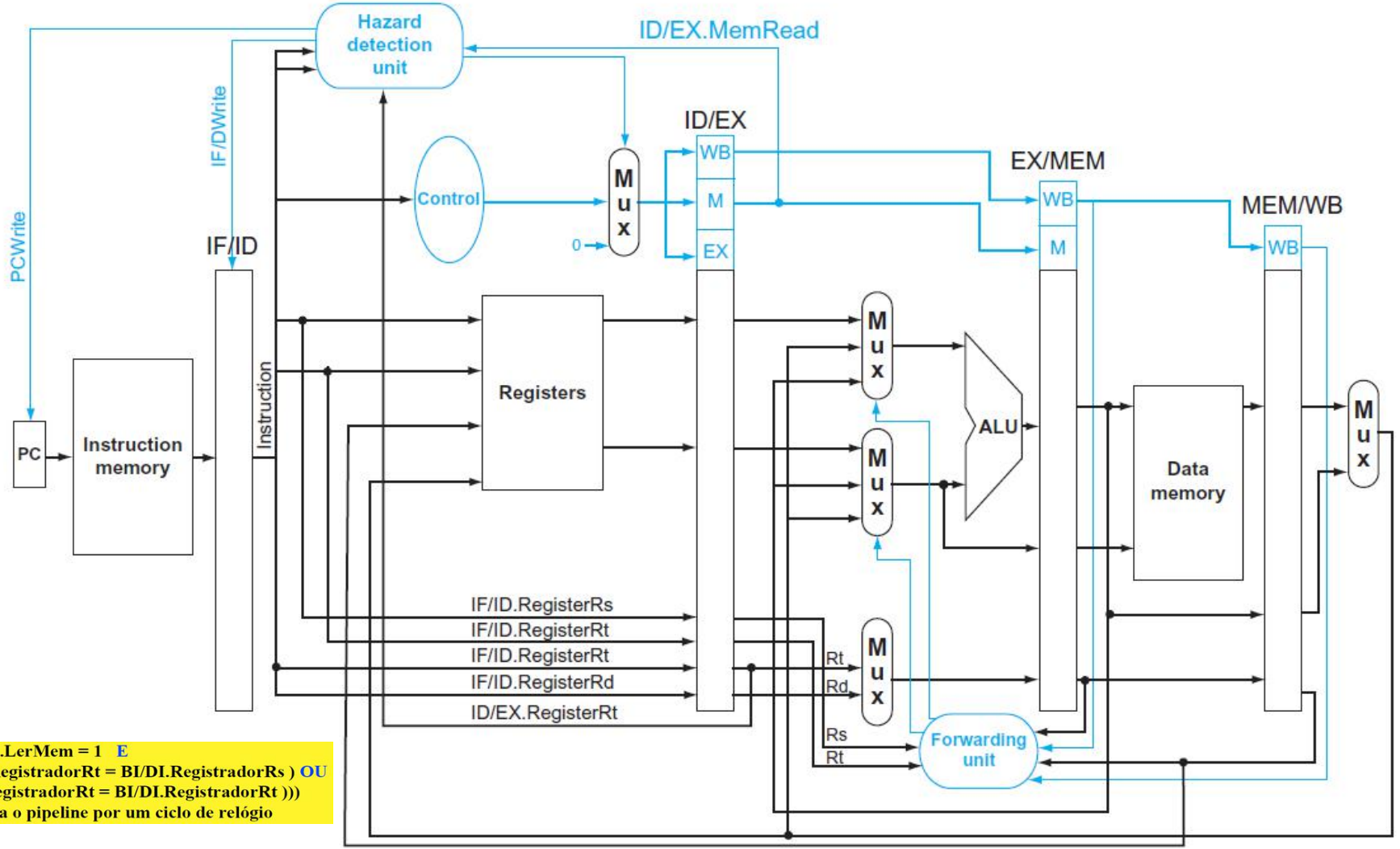
2. Os registradores lidos no estágio DI serão novamente lidos: do ponto de vista de execução, **executar 2 vezes a mesma coisa é igual a dobrar o tempo de execução**;neste caso, acrescenta-se 1 ciclo de espera até que seja possível resolver o conflito.

Retomando o fluxo no pipeline

- Como a instrução *lw* prossegue pelo pipeline, cria-se uma “bolha” de execução; esta bolha também prossegue pelo pipeline
- O efeito de uma “bolha” no pipeline é o mesmo da execução de uma instrução do tipo NOP, isto é:
  1. Todos os sinais de controle são levados a 0 (zero) nos estágios EX, MEM e ER.
  2. Os valores de sinais de controle são passados adiante a cada clock, produzindo o efeito desejado pela inclusão de uma “bolha” → nenhum registrador ou endereço de memória é escrito(a) e o estado da via de dados não é alterado.

NOOP -- ***no operation***

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000



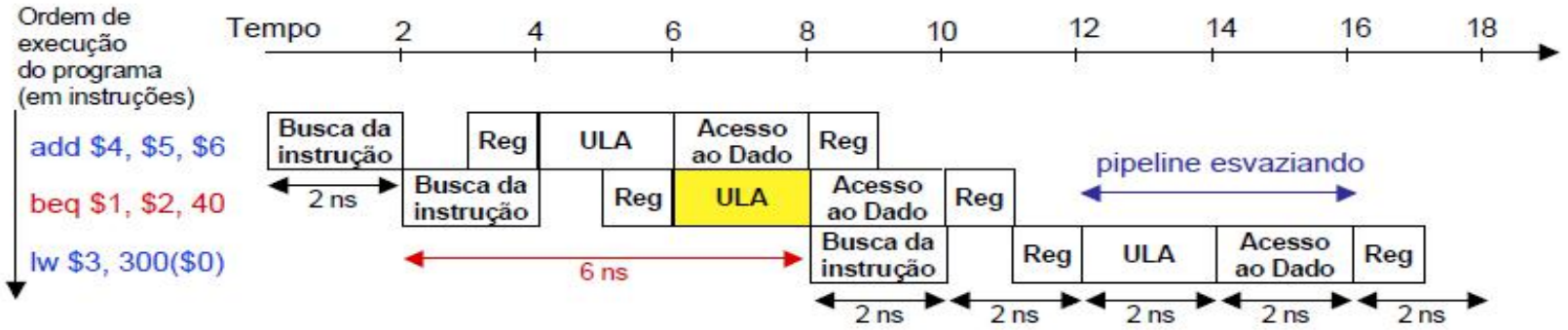
Se (DI/EX.LerMem = 1 E ((DI/EX.RegistradorRt = BI/DI.RegistradorRs) OU (DI/EX.RegistradorRt = BI/DI.RegistradorRt)))
Então para o pipeline por um ciclo de relógio

Conflitos de CONTROLE

Conflitos de controle (control hazards)

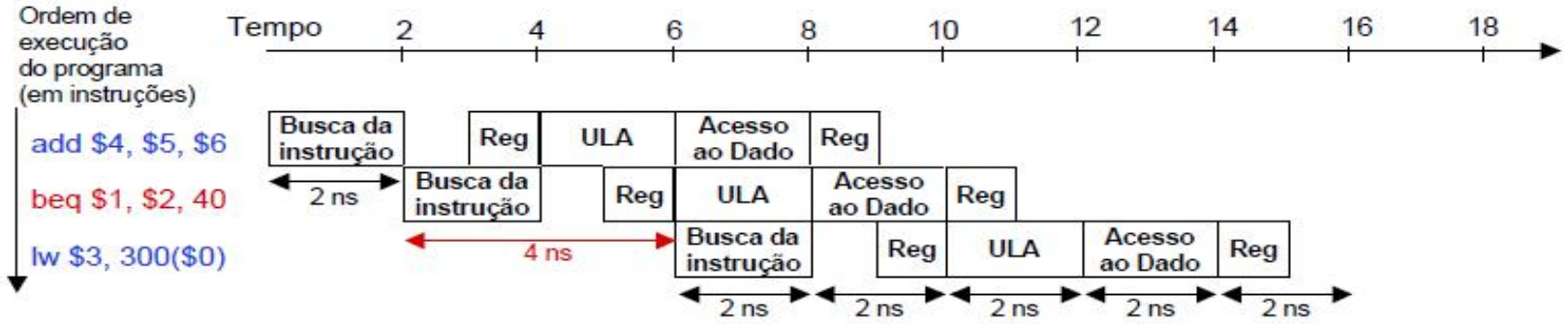
- Causa: deve ser tomada uma decisão de desviar ou não (próxima instrução a ser buscada) baseada nos resultados de uma instrução, a qual ainda não foi concluída.
- Instruções de desvio condicional (`beq`) ocasionam essa situação
- Solução óbvia: parada ("bolha") do pipeline com a interrupção da progressão das instruções por ele.

Exemplo de conflito com beq



Obs: Os exemplos com `beq` consideram a existência de um hardware extra (comparador de igualdade) que permita testar registradores, calcular o endereço de desvio condicional e atualizar o PC, tudo isso no 2º estágio. Nessa situação, não é necessário esperar a execução do 3º estágio (que usa a ULA) para determinar o desvio

Conflito com beq com o novo hardware



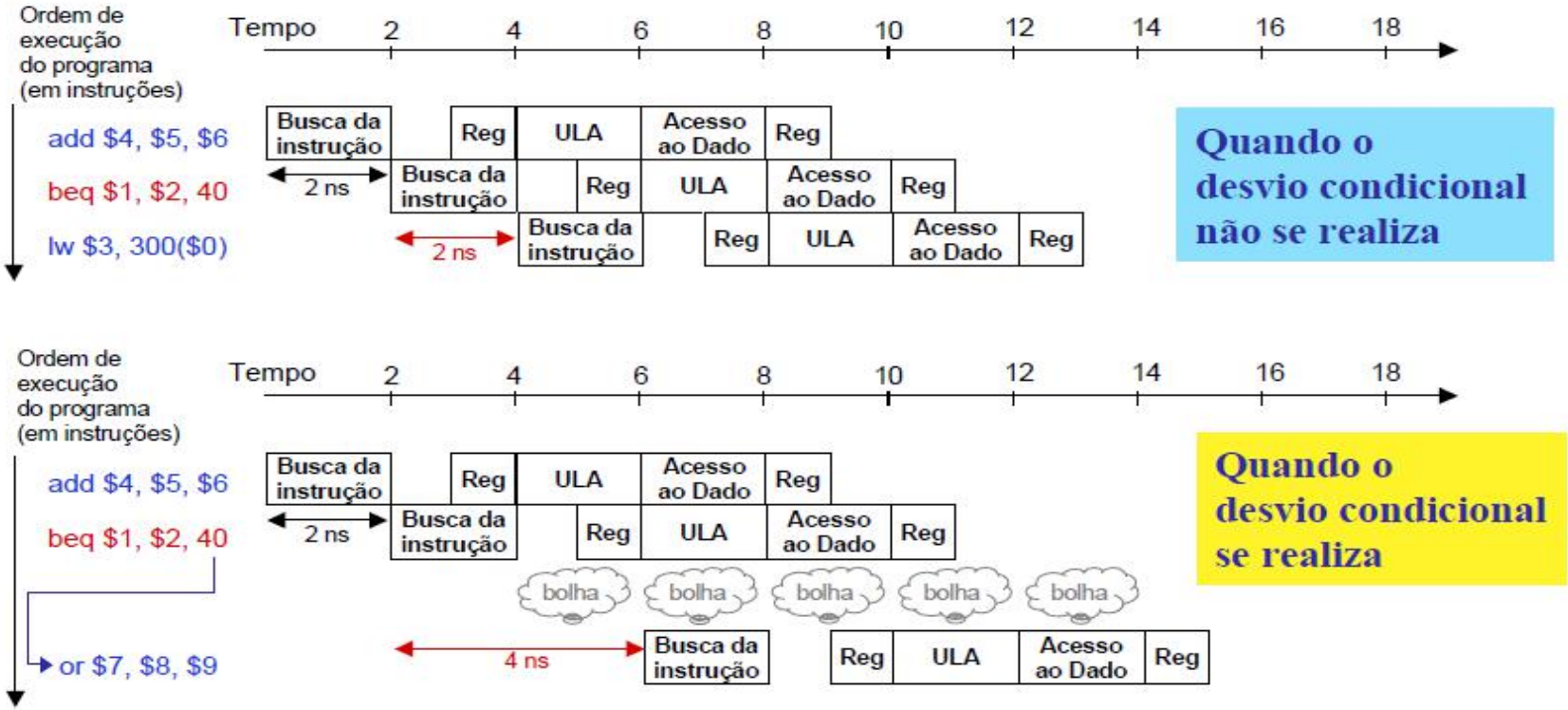
Um esquema simples de predição pode ser usado para minimizar o problema: assumir sempre que os desvios condicionais sempre irão falhar.

– Se acertar, o pipeline prossegue normalmente
– Se errar, esvaziar o pipeline

Soluções para o conflito de controle

- A parada no avanço das instruções (bolha) não é uma solução viável para o desvio condicional;
- Uma alternativa comum é *considerar que os desvios condicionais sempre ocorrem*, considerando a sequência normal de execução das instruções:
  - *Se o desvio se realizar*, será necessário *descartar as instruções* que estiverem sendo buscadas e executadas; a execução deve continuar a partir da instrução armazenada no endereço-alvo do desvio condicional;
- Descartar instruções (*flush*) significa:
  - Colocar em 0 os valores dos sinais de controle;
  - Alterar as instruções que estiverem nas etapas BI, DI e EX assim que a instrução de desvio condicional chegar ao estágio MEM.

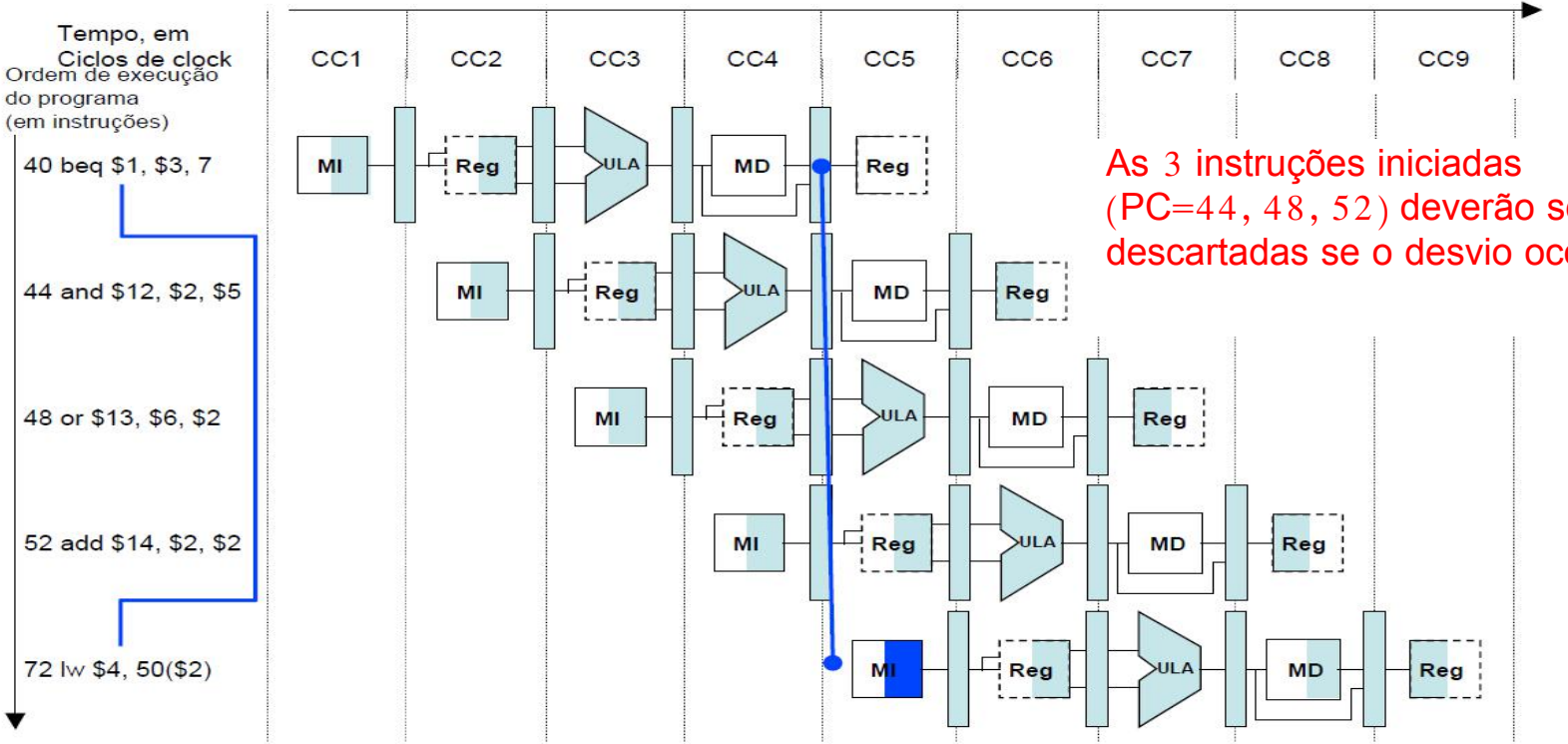
Uso de predição para resolver o conflito



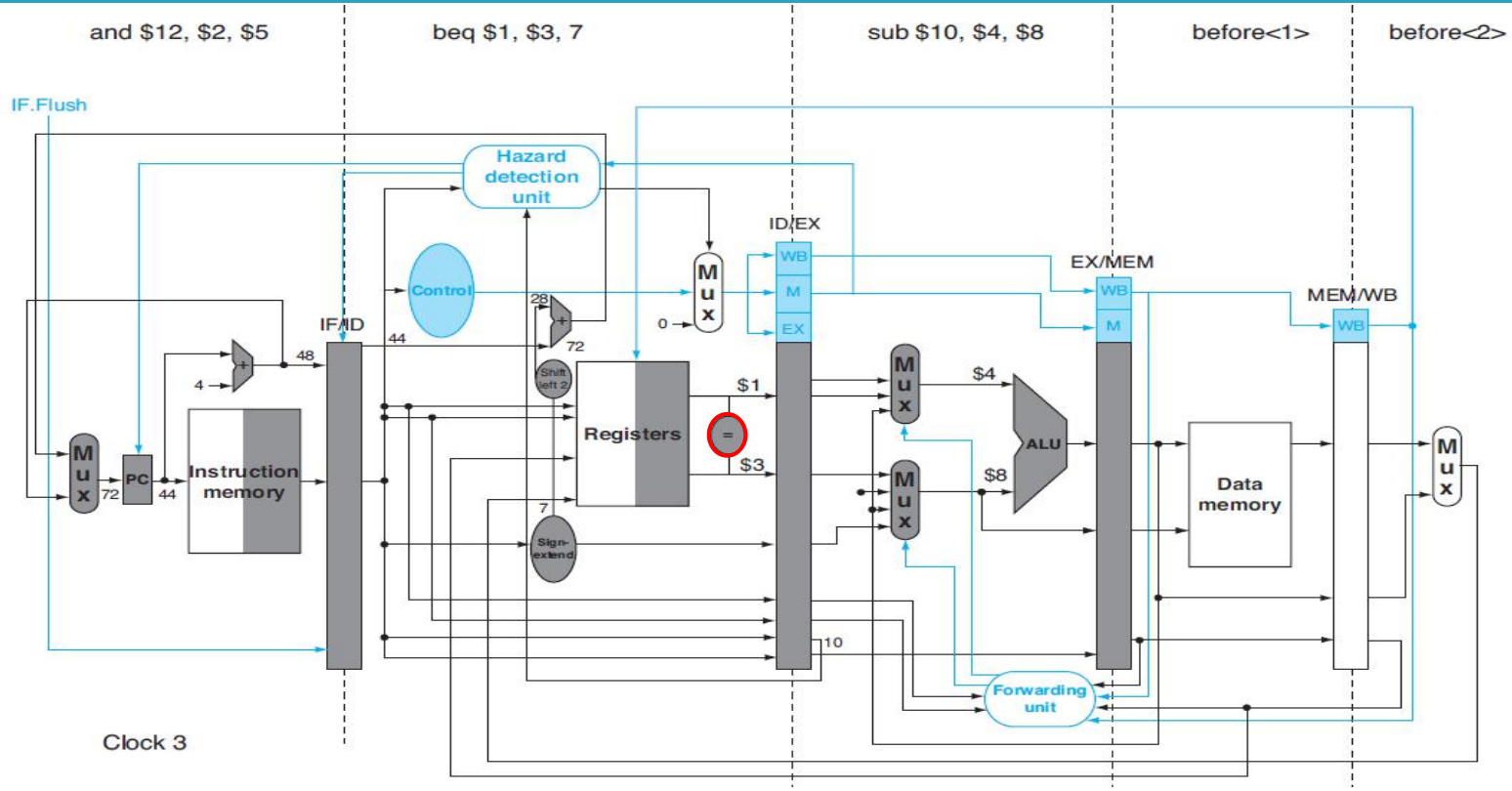
Conflito de controle em código MIPS

```
36    sub    $10, $4, $8
40    beq    $1, $3, 7      # desvio relativo ao 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)
```

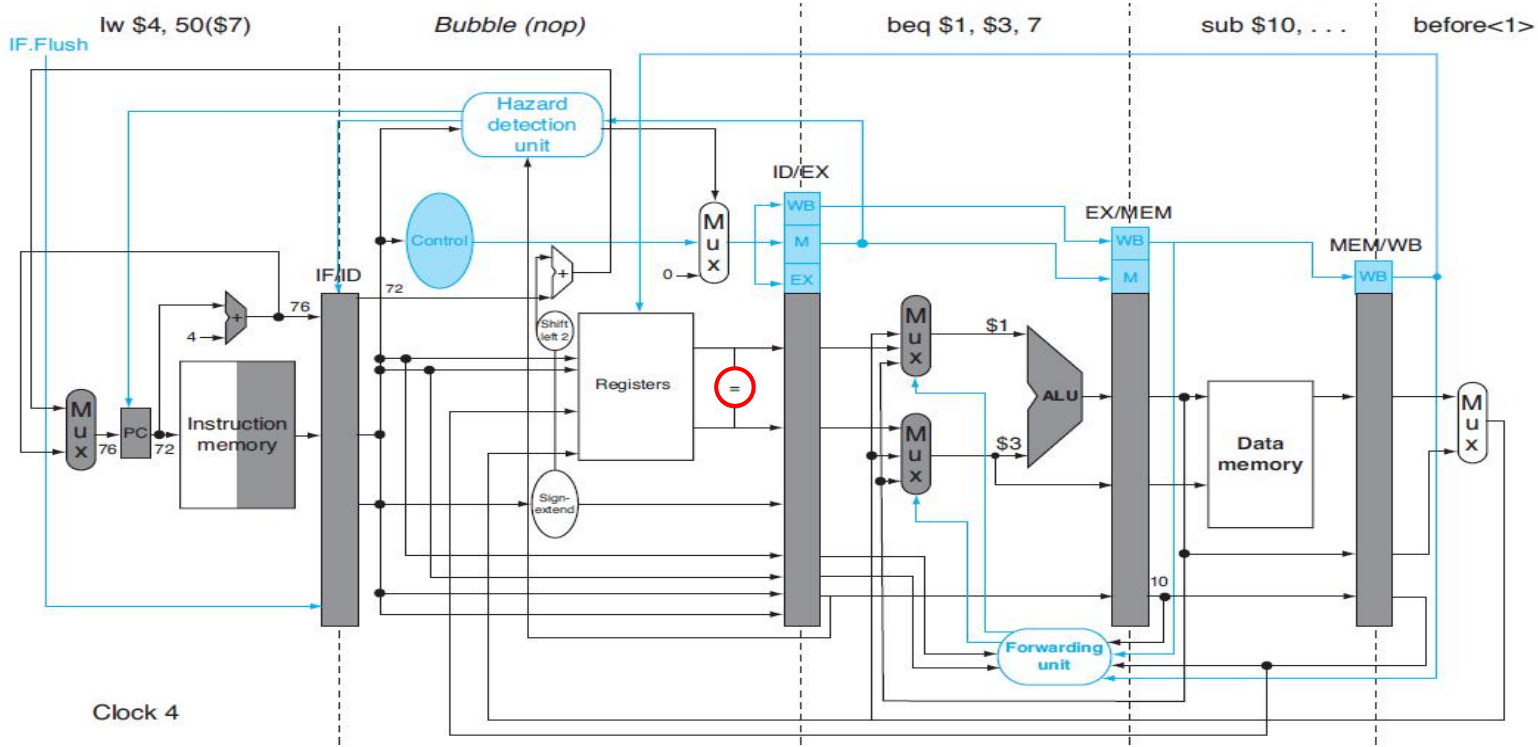
Conflito de controle no pipeline

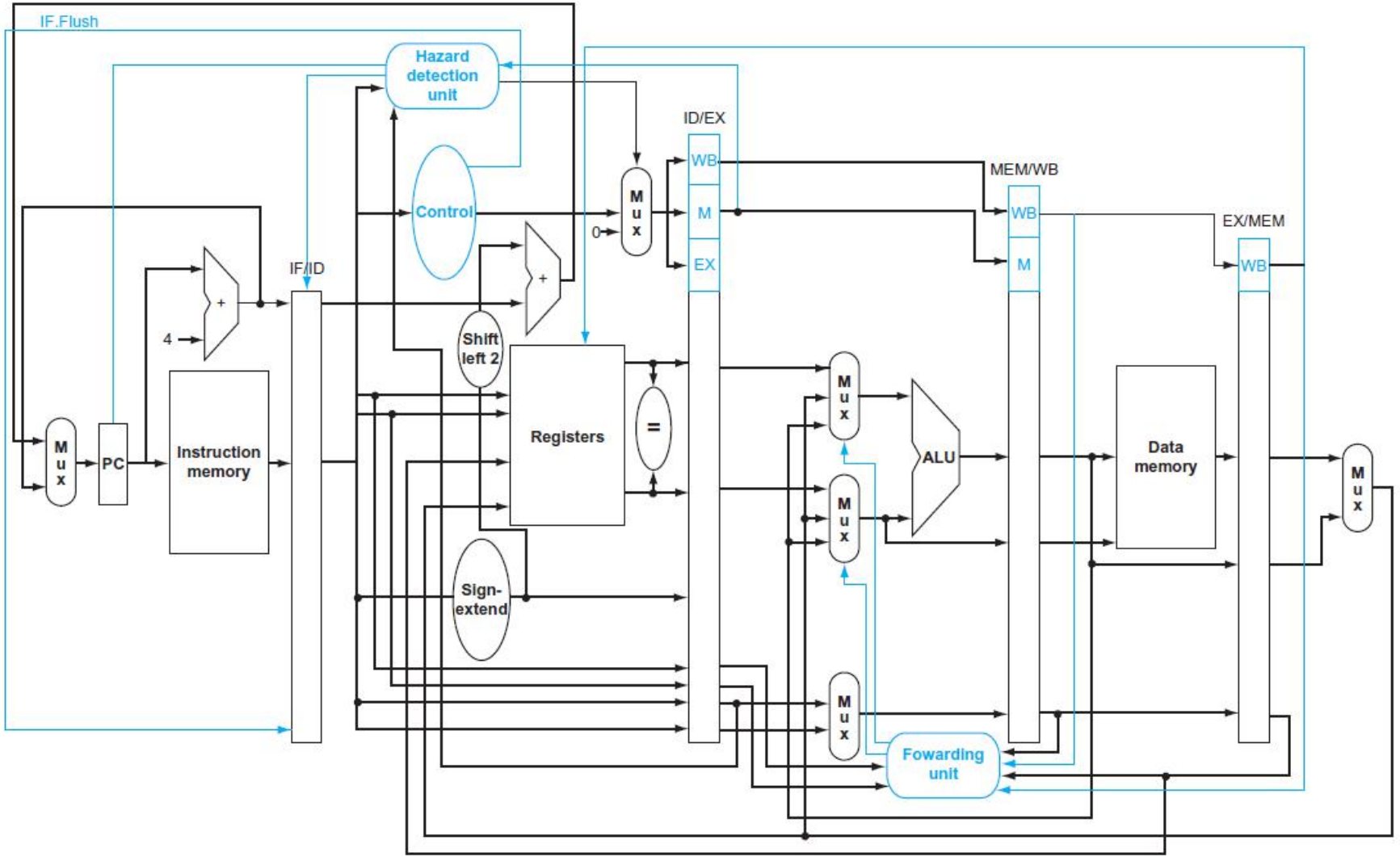


Clock 3 – beq em execução



Clock 4 – após o beq ter definido o desvio




IF.Flush
Hazard detection unit
Control
ID/EX
WB
M
EX
MEM/WB
WB
M
EX/MEM
WB
0
M u x
IF/ID
+
4
Shift left 2
Registers
=
Instruction memory
PC
ALU
Data memory
Sign-extend
Fowarding unit

Exceções e interrupções

CPU PIPELINE MIPS

Tratamento de exceções na arq. MIPS

- Exceções e interrupções: eventos que mudam o fluxo normal de execução das instruções: diferente de desvios condicionais ou incondicionais (saltos)
- Tratamento de eventos “inesperados” (overflow, I/O, erro de endereço,...)
- No MIPS:
  - – Exceção ou *trap*: qualquer mudança inesperada no controle (PC muda inesperadamente…)
    - Não se pode precisar se o evento causador é interno ou externo
  - – Interrupção: evento tem causa externa (I/O)
    - Alguns autores não distinguem interrupção de exceção (CPU Intel, tudo é interrupção)

Exemplos de situações no MIPS

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

- O controle da CPU deve dar suporte à detecção e tratamento de ambas as situações
- Detectar e tratar uma exceção ou interrupção pode ser visto como um dos caminhos críticos de tempo da CPU: impacto no desempenho
- Na implementação MIPS atual, apenas 2 exceções são possíveis: Instrução indefinida $(\text{código } 10)$ e overflow aritmético $(\text{código } 12)$

Tratamento de exceções

- Se uma exceção ocorre, um SO deve tomar alguma ação apropriada:
  - Providenciar algum serviço ao programa usuário (syscall);
  - Tomar alguma ação predefinida em resposta à exceção (rotina);
  - Parar a execução do programa e reportar o erro (mensagem na tela);
- No MIPS, após tomar a ação, um SO usa os valores do *Exception Program Counter* (*EPC*) para decidir se recomeça ou não a execução do programa interrompido pela exceção;
- O SO precisa saber qual instrução causou a exceção e qual o tipo dela, e existem 2 métodos para isso:
  - Usar um registrador de status ou *cause register* (MIPS): o registrador armazena um código para o tipo de interrupção e
  - Usar um vetor de interrupções (Intel): a causa da interrupção define o endereço de uma das rotinas de tratamento que estão armazenados em um vetor)

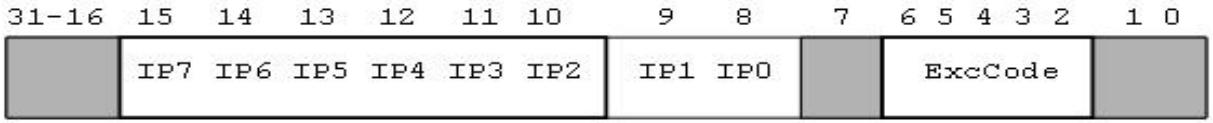
Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Procedimento: Tratamento de exceções

- A CPU opera em 2 modos: kernel (SO) e usuário (programas):
  - Se ocorre uma exceção; CPU em modo kernel o procedimento é o seguinte:
    1. Salvar o PC no *Exception Program Counter* (EPC);
    2. Transferir o controle para o SO:
       - Chamar rotina de tratamento, tomar ação predefinida ou terminar a aplicação e informar o erro;
       - Depois de tratar a exceção (se conseguir), o SO retoma a execução do programa recuperando o EPC.

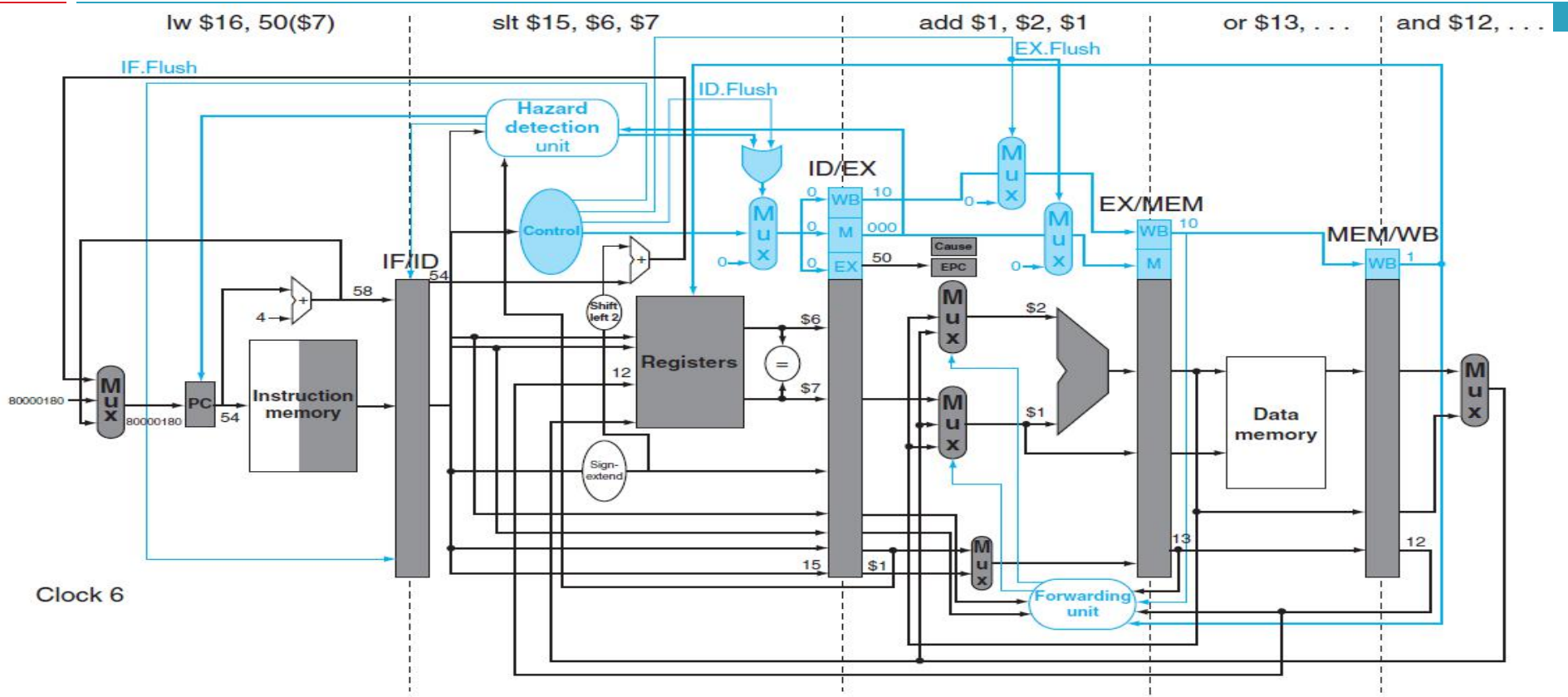
Procedimento: Tratamento de exceções

- Implementação MIPS – 2 registradores 32 bits;
- EPC: guarda o endereço da inst. que gerou a exceção:
  - Cause: guarda a causa da exceção (overflow aritmético ou inst inválida) num campo de 5 bits (ExCode); IP_n = interrupções pendentes



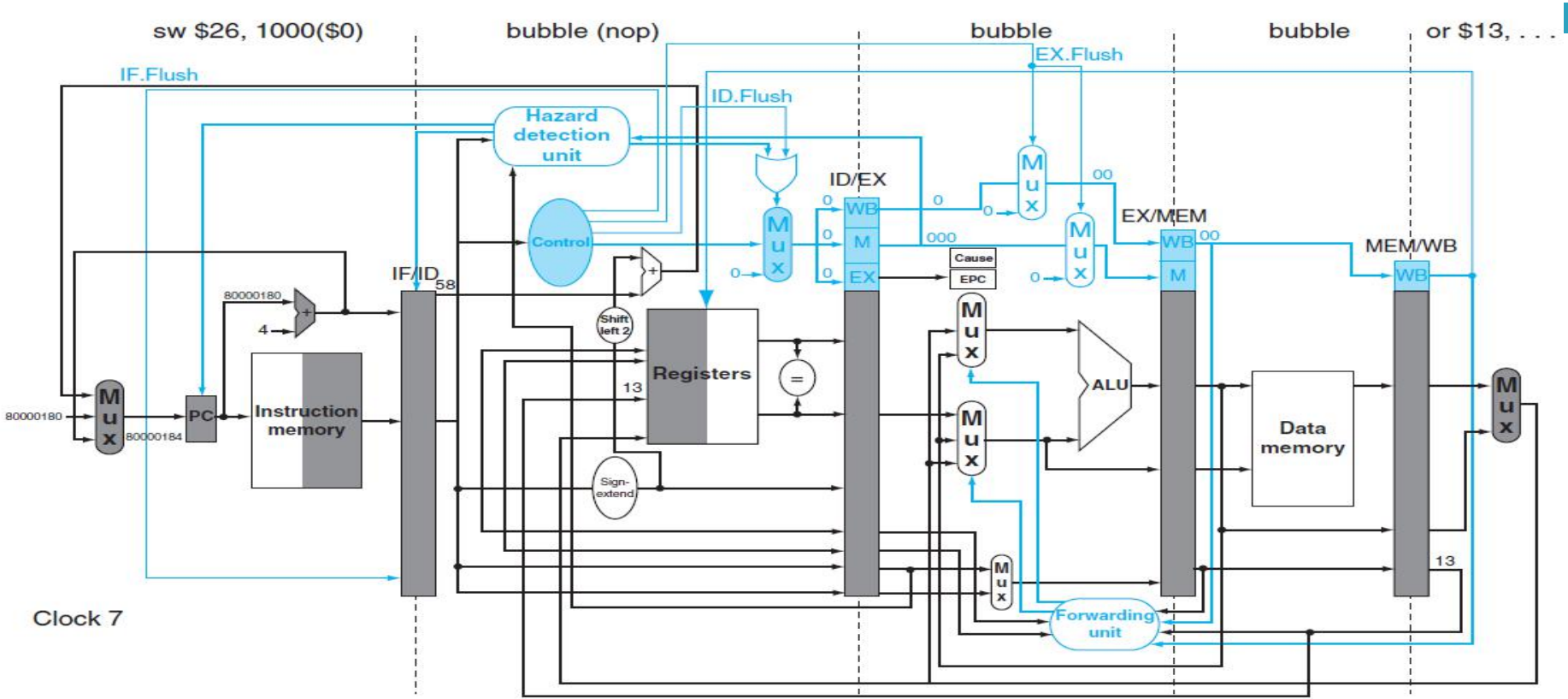
  - Na CPU pipeline MIPS, uma exceção é tratada de modo similar a um conflito – supondo uma exceção por overflow:
    - As instruções que seguem devem ser descartadas;
    - O PC é carregado com o endereço (80000180) da rotina de interrupção.

Overflow no pipeline

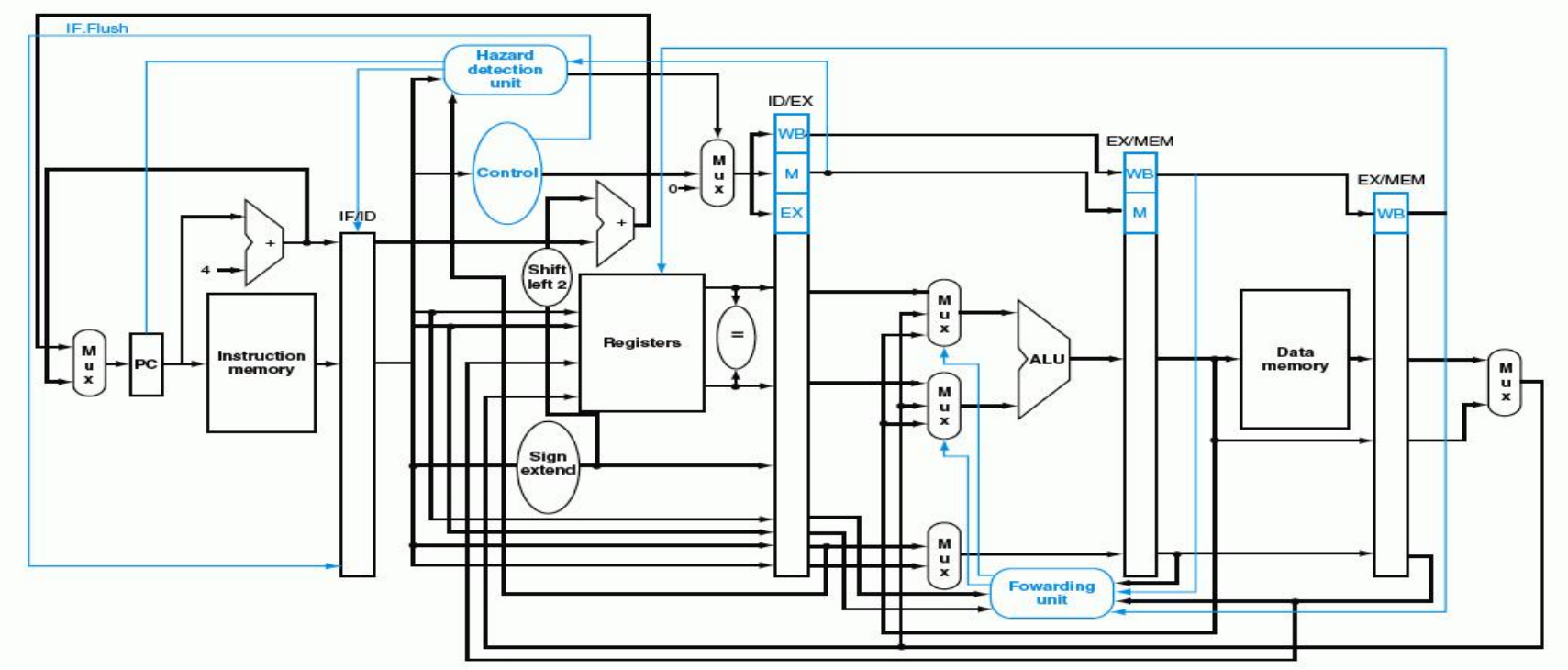


Overflow no pipeline

O sinal de overflow vai para o controle, apesar de não indicado na Figura!



CPU Pipeline: dados & controle & exceções



Antes do tratamento de exceções/interrupções

CPU Pipeline final com tratamento de exceções