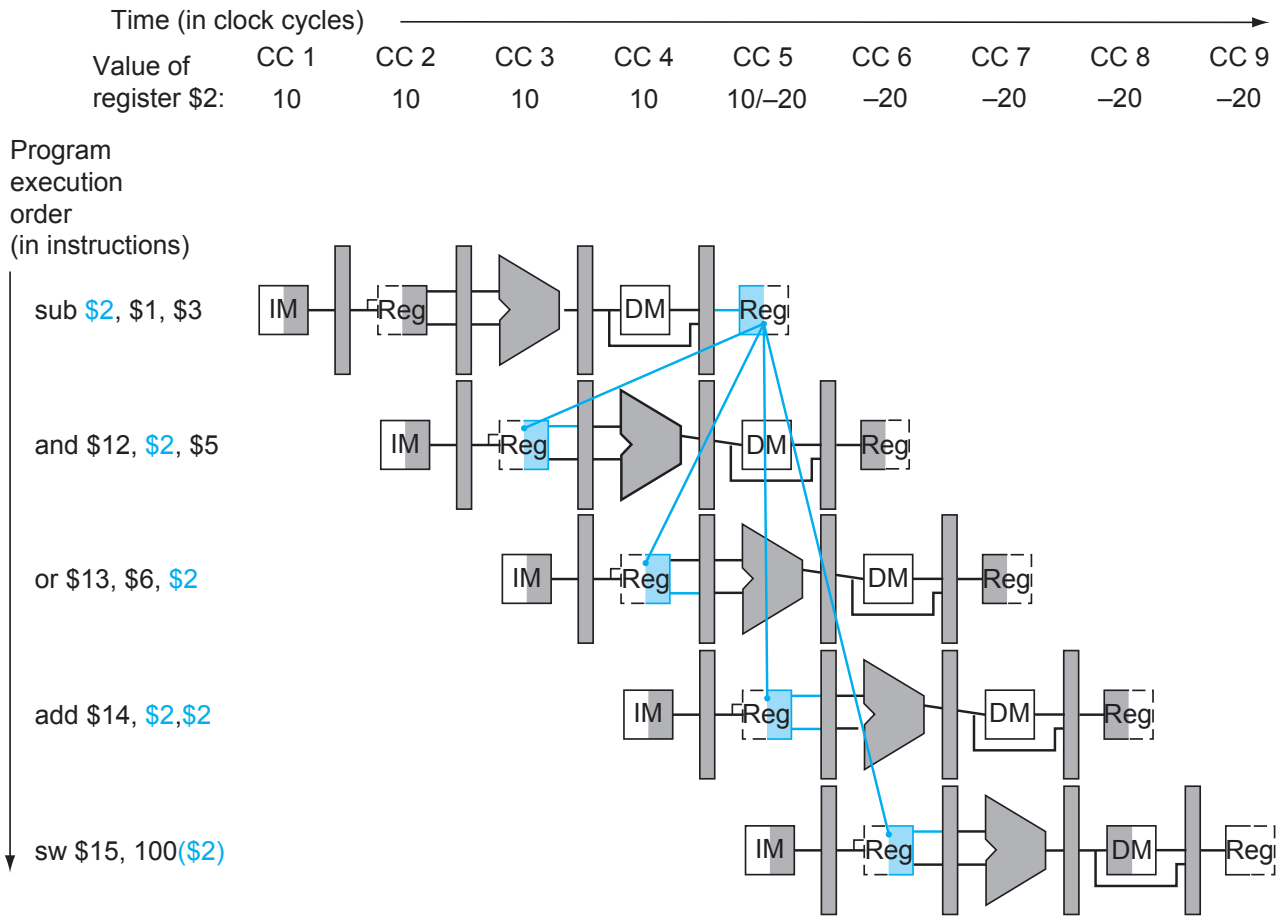
Dependências de dados

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

Dependências

Todas as instruções dependem do valor que a primeira calcula para $2

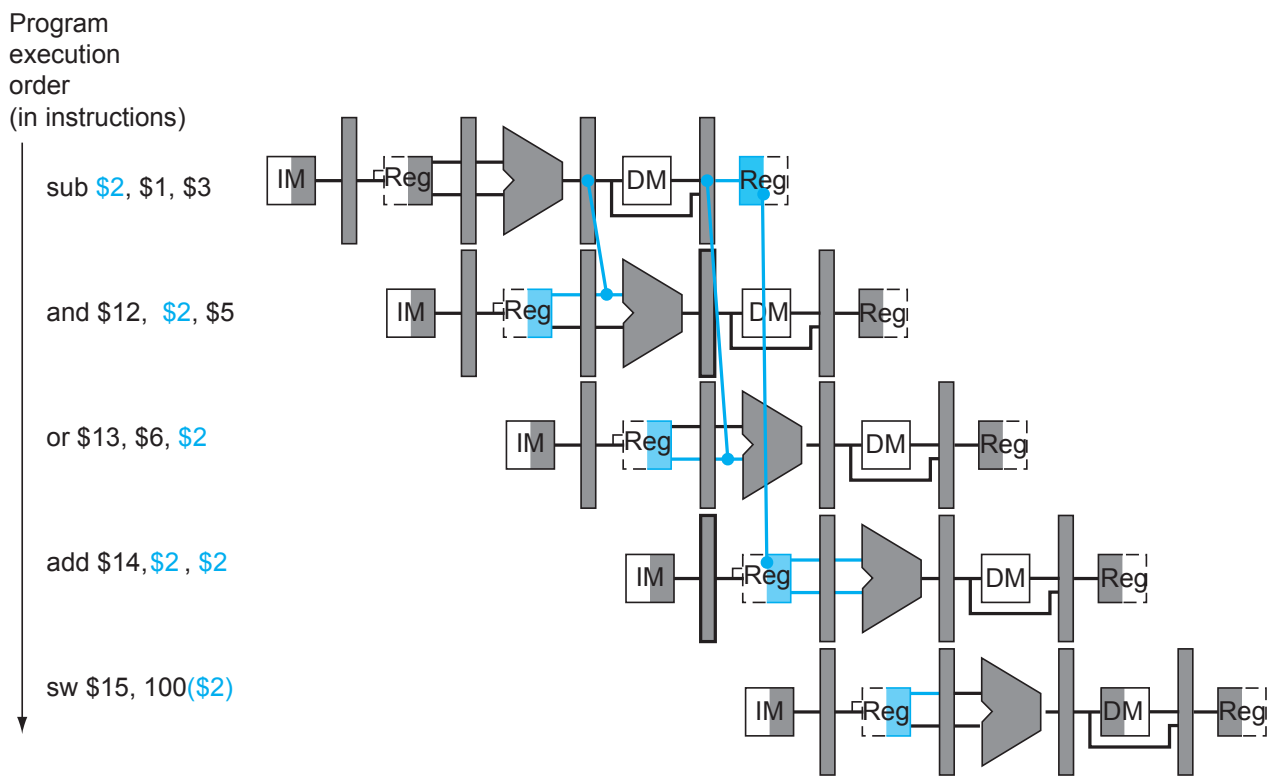


Dependências e conflitos de dados

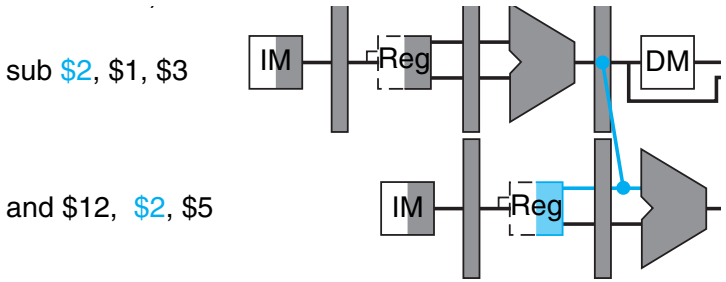
- Só há conflito de dados nas instruções 2 e 3
- Que podem ser resolvidos por *forwarding*
  - Do registo EX/MEM para o andar EX (ciclo 4)
  - Do registo MEM/WB para o andar EX (ciclo 5)

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



Detecção de conflitos de dados (1)



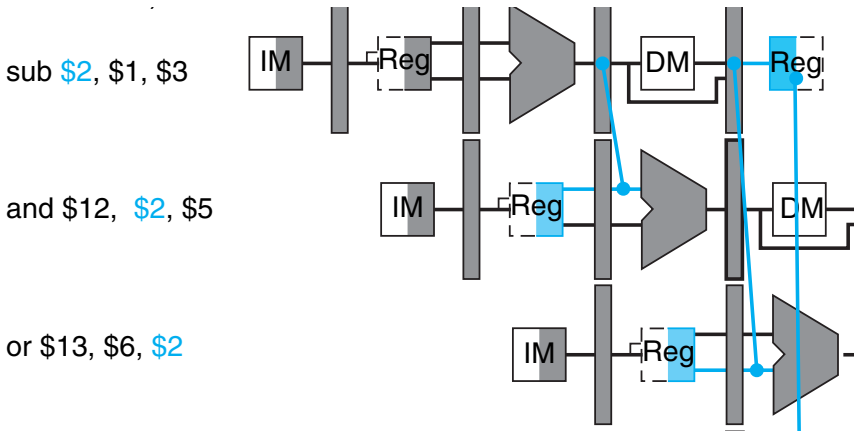
Caso 1

O valor que irá estar no registo rs do `and` está no registo EX/MEM do `sub`, que o irá escrever no seu registo rd

Há conflito nas seguintes condições

1a. ID/EX.RegisterRs = EX/MEM.RegisterRd

1b. ID/EX.RegisterRt = EX/MEM.RegisterRd



Caso 2

O valor que irá estar no registo rt do `or` está no registo MEM/WB do `sub`, que o irá escrever no seu registo rd

Há conflito nas seguintes condições

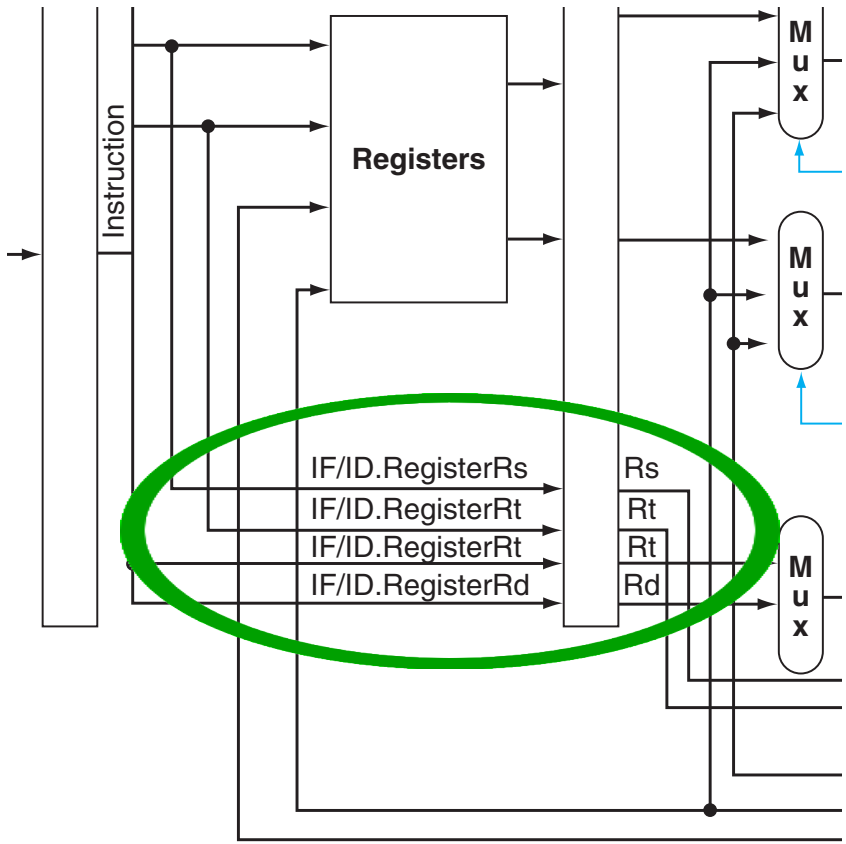
2a. ID/EX.RegisterRs = MEM/WB.RegisterRd

2b. ID/EX.RegisterRt = MEM/WB.RegisterRd

Implementação de *forwarding*

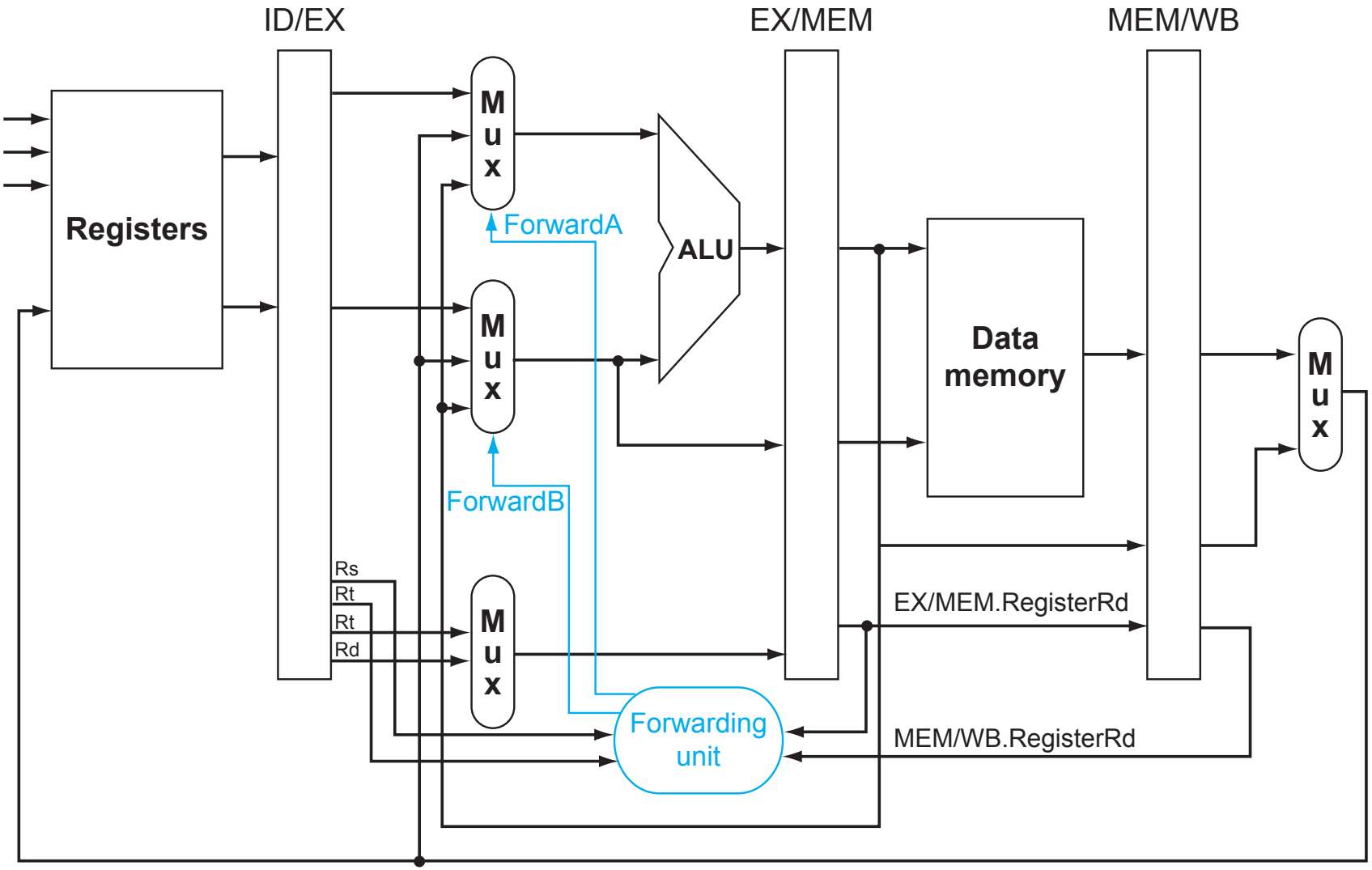
Suporte à detecção de conflitos no andar EX

O registo ID/EX deverá identificar os registos rs, rt e rd



Os registos EX/MEM e MEM/WB já identificam o registo rd

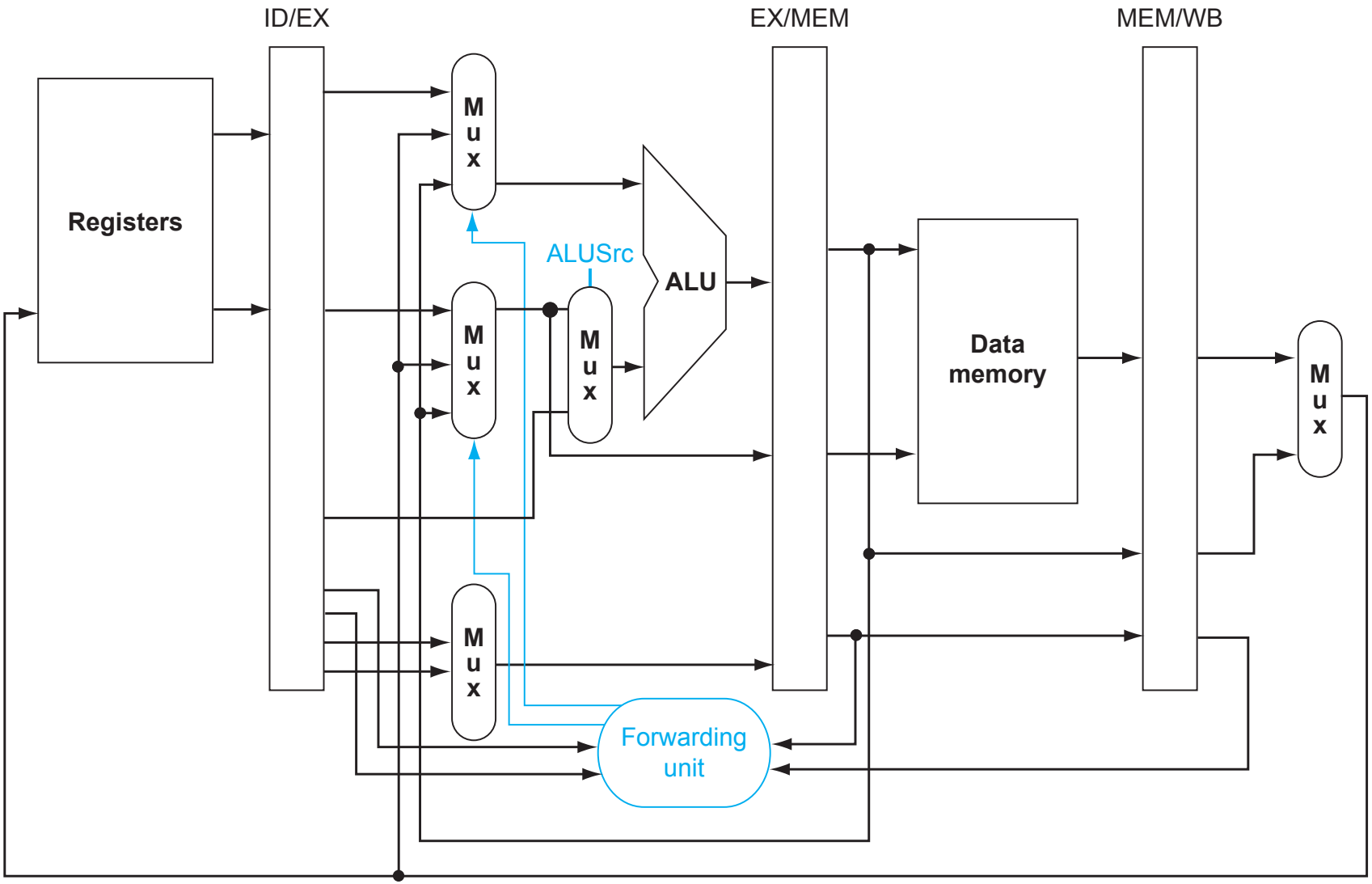
Inserção de *forwarding* no caminho de dados



Onde fica o mux(ALUSrc)?

Inserção de *forwarding* no caminho de dados

Com o mux(ALUSrc)



Controlo do *forwarding*

Forwarding unit

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.

Detecção de conflitos de dados (2)

De MEM para EX

Haverá conflito se a instrução que está no andar MEM

- Escreve um registo
- Esse registo é o registo rs ou/e rt da instrução no andar EX
- Esse registo não é $0

Controlo do *forwarding*

if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd $\neq$ 0)
  and (ID/EX.RegisterRs = EX/MEM.RegisterRd) ForwardA $\leftarrow$ 10

if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd $\neq$ 0)
  and (ID/EX.RegisterRt = EX/MEM.RegisterRd) ForwardB $\leftarrow$ 10

Detecção de conflitos de dados (3)

De WB para EX

Haverá conflito se a instrução que está no andar WB

- Escreve um registo
- Esse registo é o registo rs ou/e rt da instrução no andar EX
- Esse registo não é $0

Controlo do *forwarding*

if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd $\neq$ 0)
  and (ID/EX.RegisterRs = MEM/WB.RegisterRd) ForwardA $\leftarrow$ 01

if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd $\neq$ 0)
  and (ID/EX.RegisterRt = MEM/WB.RegisterRd) ForwardB $\leftarrow$ 01

Múltiplos conflitos

```
                              Andar
add $1, $1, $2                WB
add $1, $1, $3                MEM
add $1, $1, $4                EX
```

Qual o valor que deverá ser *forwarded*?

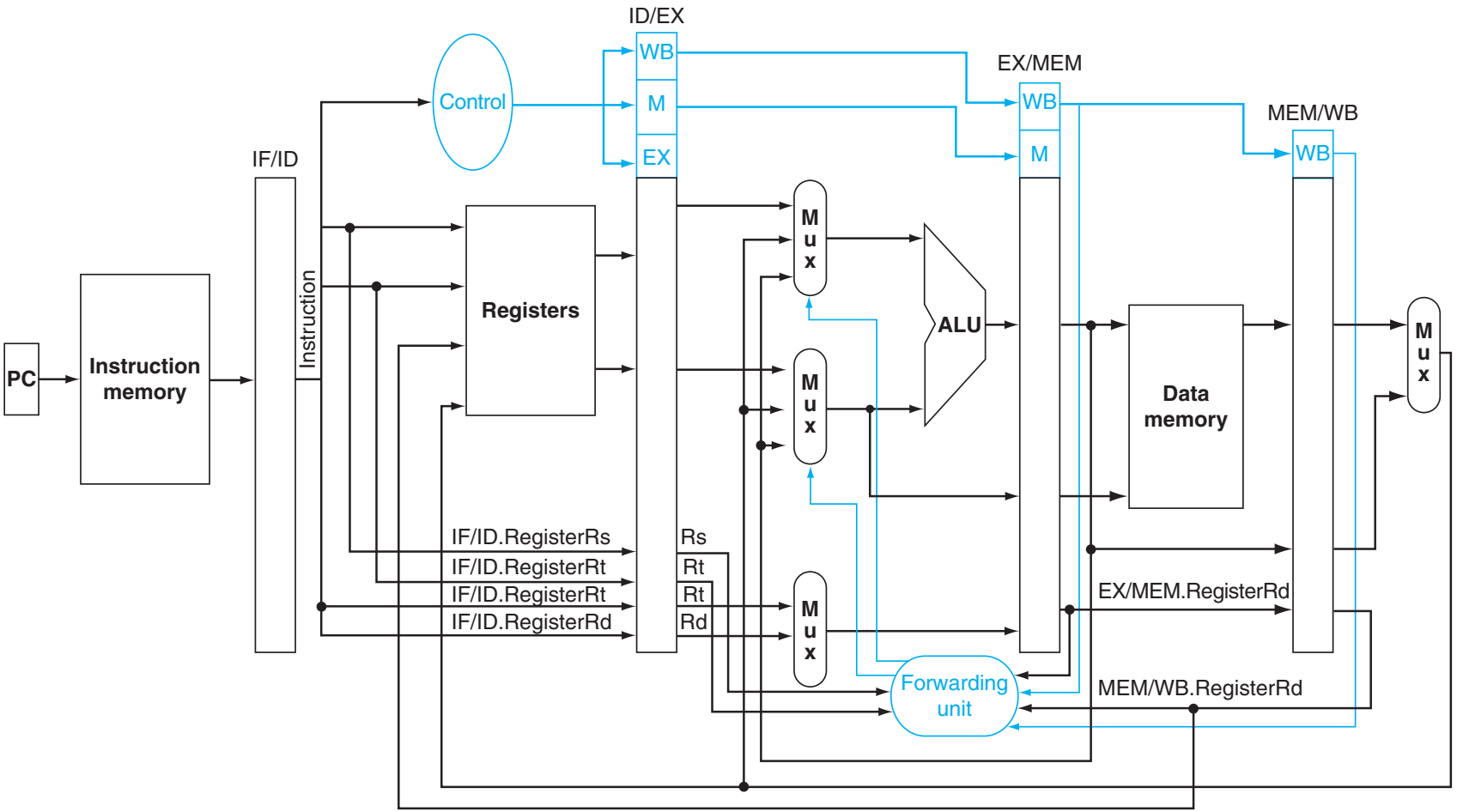
A prioridade pertence a MEM

if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd $\neq$ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 0)
            and (ID/EX.RegisterRs = EX/MEM.RegisterRd))
  and (ID/EX.RegisterRs = MEM/WB.RegisterRd) ForwardA $\leftarrow$ 01

Para ForwardB será semelhante

Caminho de dados com *forwarding*

Ligação dos sinais de controlo

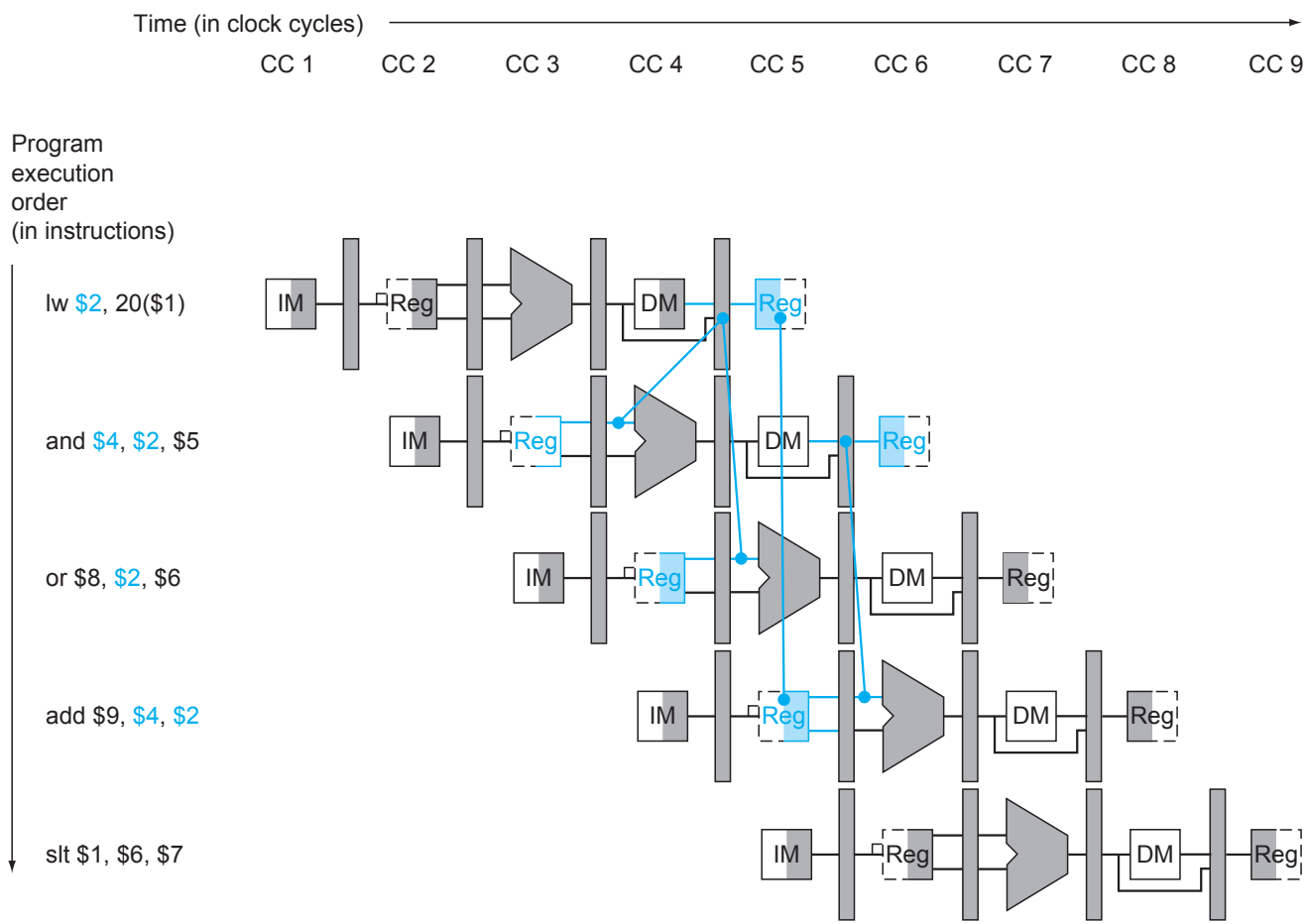


Não é mostrado o mux(ALUSrc)

Conflitos de dados e atrasos do *pipeline* (1)

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

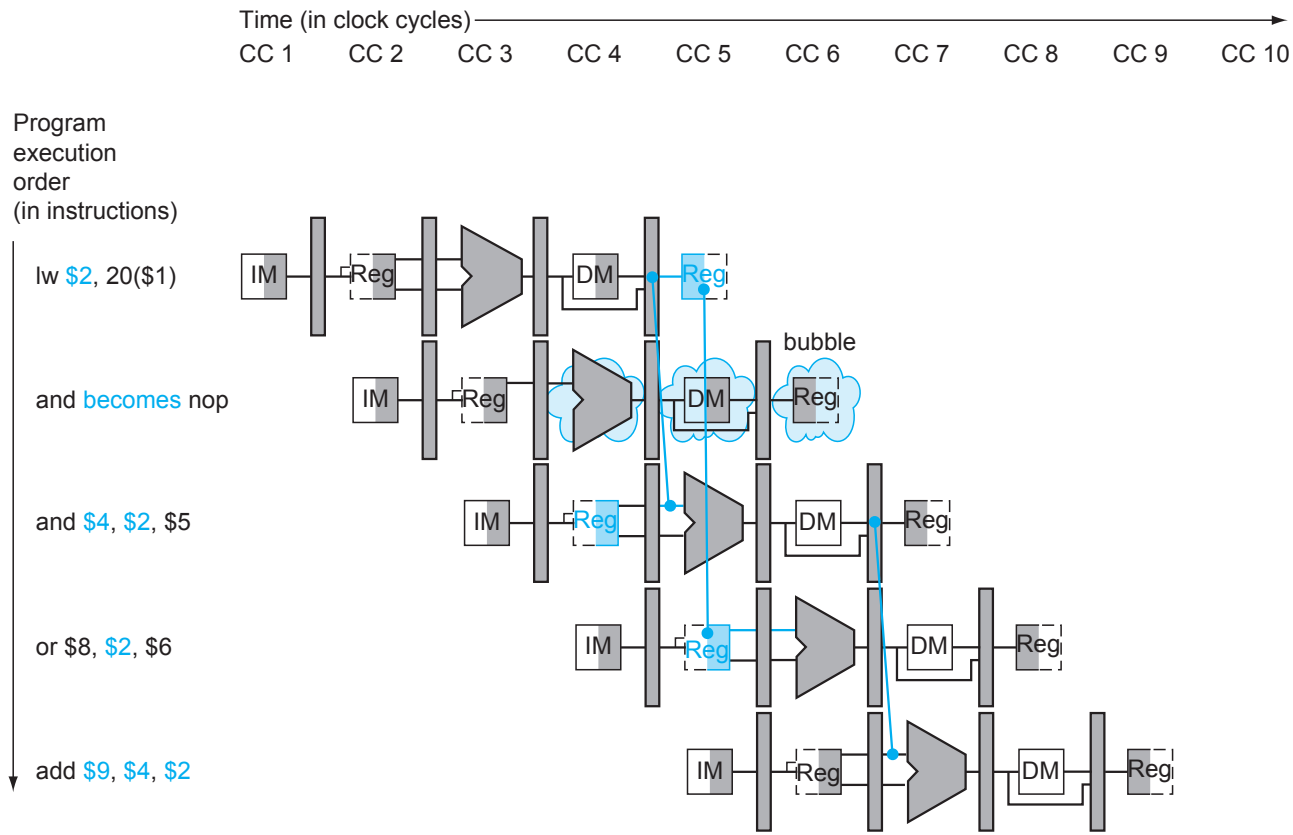
O valor lido por `lw` só fica disponível quando a instrução passa para o andar WB



Conflitos de dados e atrasos do *pipeline* (2)

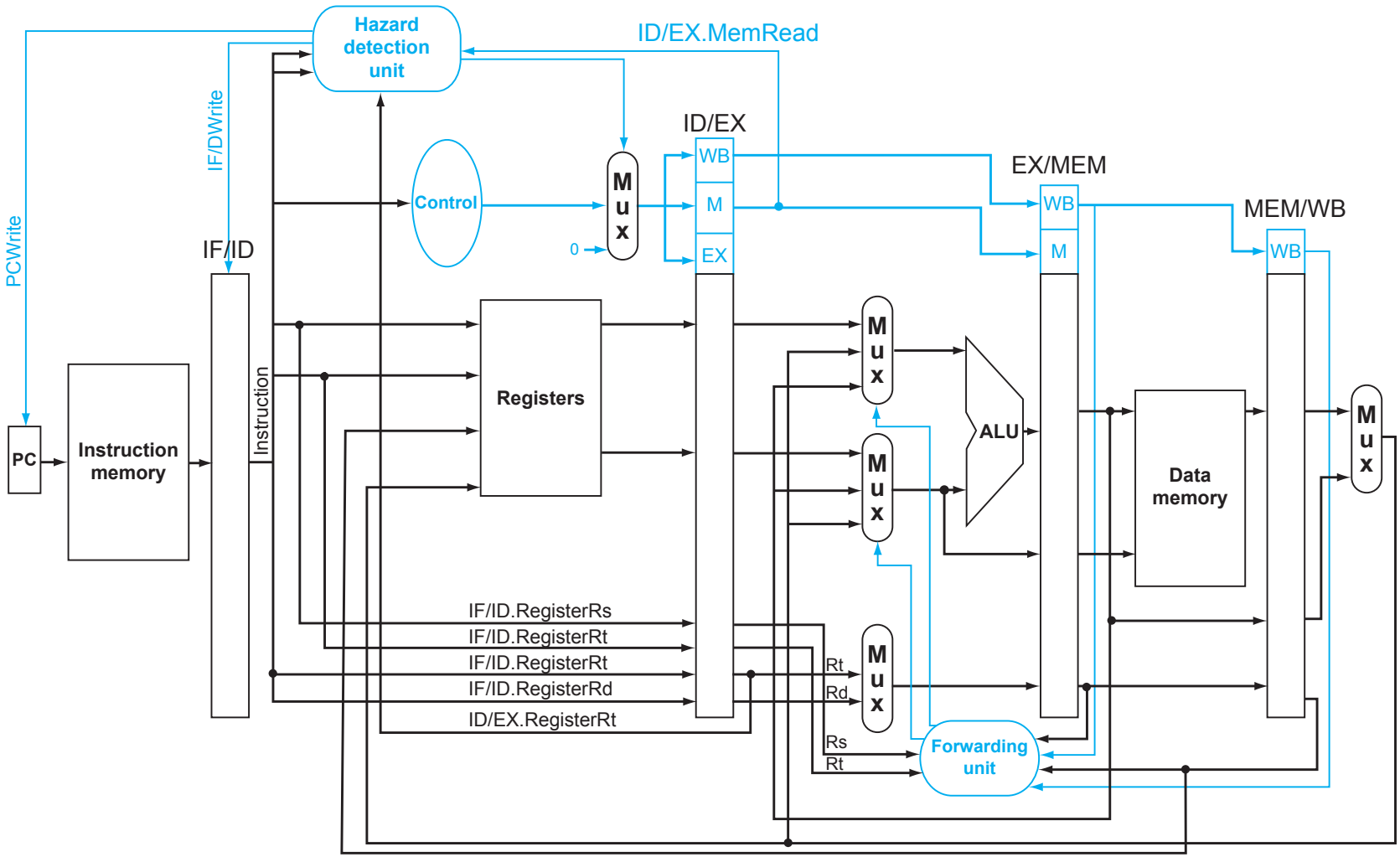
É necessário atrasar o `and` um ciclo de relógio

1. Impedindo-o de avançar no *pipeline*
2. Impedindo uma nova instrução de entrar no *pipeline*
3. Inserindo uma *bolha* (equivalente a um `nop`) entre o `lw` e o `and`



Conflitos de dados e atrasos do *pipeline* (3)

Detecção: if (ID/EX.MemRead and (ID/EX.RegisterRt $\neq$ 0)
and ((IF/ID.RegisterRs = ID/EX.RegisterRt) or
(IF/ID.RegisterRt = ID/EX.RegisterRt)))
stall the pipeline...



Introdução de um atraso no *pipeline*

Como é criado um *pipeline stall*

Atrasar o `and` um ciclo de relógio consiste em

1. Impedi-lo de avançar no *pipeline*

   O novo sinal IF/IDWrite tem valor 0 para que o registo IF/ID mantenha a instrução `and`

2. Impedir uma nova instrução de entrar no *pipeline*

   O novo sinal PCWrite tem valor 0 para que o PC mantenha o endereço do `or`

3. Inserir uma *bolha* entre o `lw` e o `and`

   É seleccionada a entrada com valor 0 do novo *multiplexer* para os sinais de controlo no registo ID/EX, de modo a estes terem o valor zero, inserindo uma *bolha* no andar EX do *pipeline*, entre o `lw` e o `and`

Conflitos de controlo (1)

Que fazer face a um salto condicional?

Alternativas

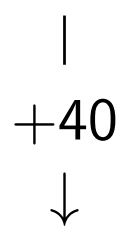
1. Atrasar o *pipeline* até saber que instrução deverá ser executada
2. Tentar prever o resultado do salto:
   a. Assumir que o salto não será efectuado e começar a executar a instrução que se segue
   b. Se o salto for para uma instrução anterior (como os saltos no fim de um ciclo), assumir que será efectuado
   c. Empregar técnicas mais sofisticadas para tentar prever se um salto será ou não efectuado

   Se a previsão se revelar errada, o *pipeline* deve ser "limpo" das instruções que não deveriam ser executadas
3. Usar *delayed branches* para minimizar os efeitos adversos dos conflitos de controlo (a solução do MIPS)

Conflitos de controlo (2)

Exemplo

```
add   $4, $5, $6
beq   $1, $2, 40
lw    $3, 300($0)
```



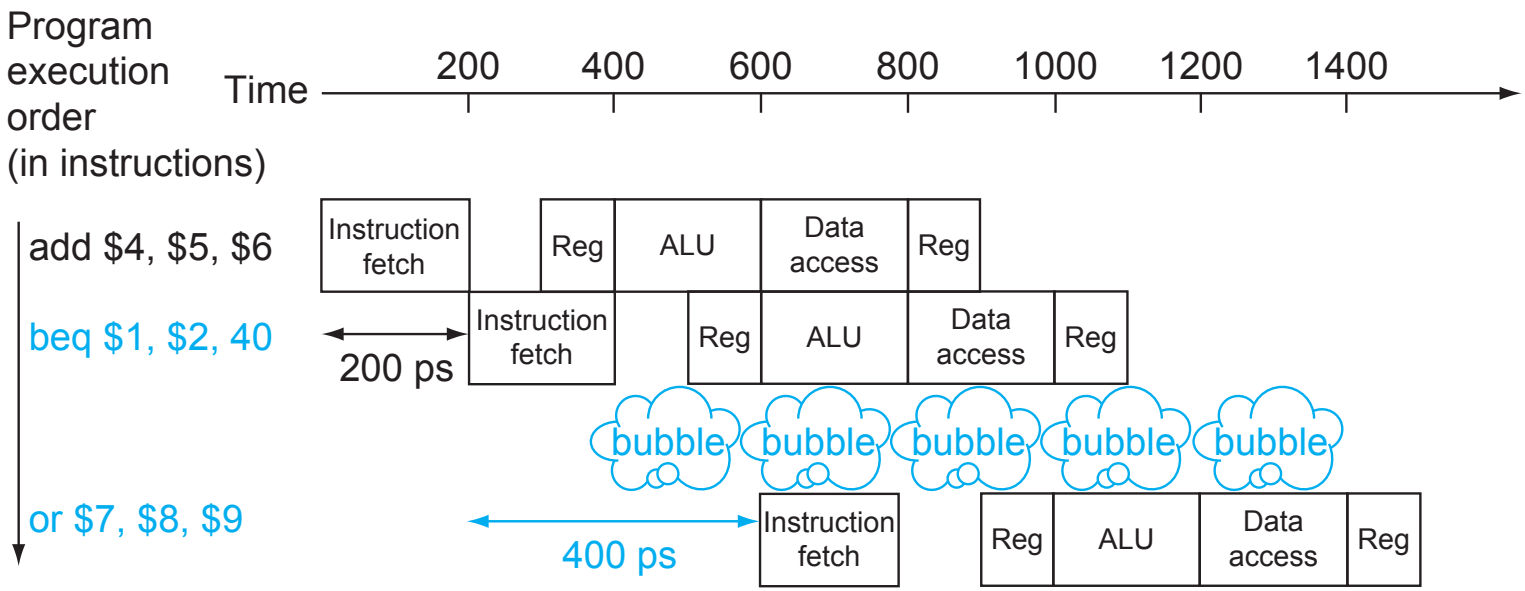

```
or    $7, $8, $9
```

Comportamentos possíveis do processador quando encontra um salto condicional (não considerando o *delay slot*) . . .

Conflitos de controlo (3)

Atraso do *pipeline* (*stall on branch*)

O início da execução da instrução a executar a seguir é atrasado um ciclo de relógio (assumindo que a decisão sobre se o salto é efectuado pode ser tomada no 2º andar do *pipeline*)

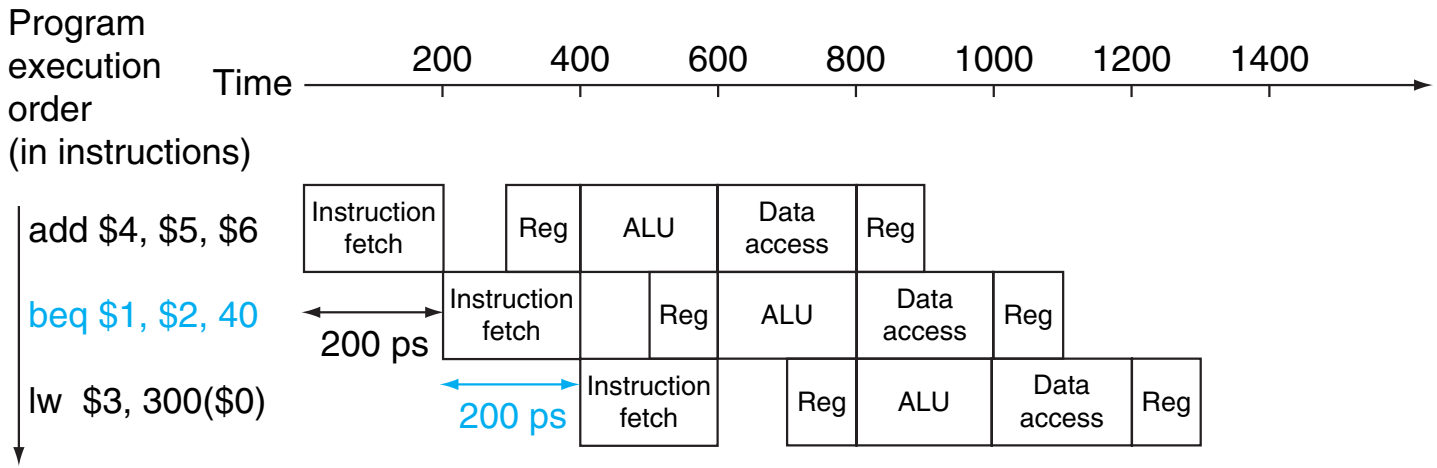


- A execução dos saltos condicionais passa a requerer 6 ciclos (*vs* 5 para as restantes instruções)

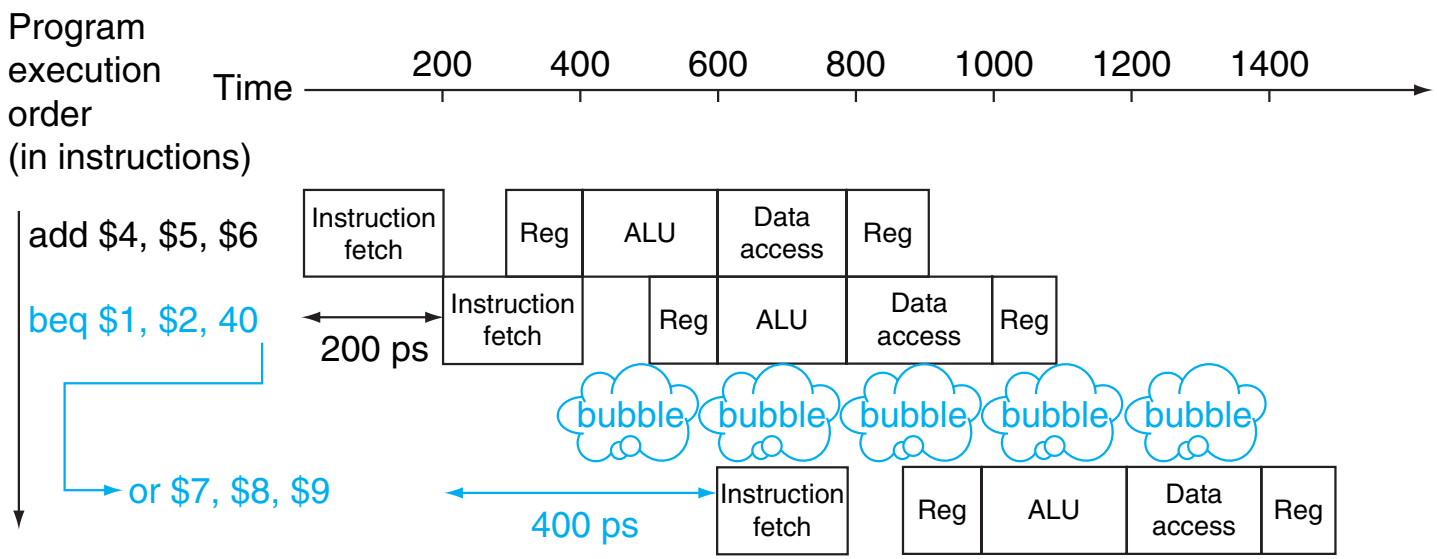
Conflitos de controlo (4)

Prevendo que o salto não será efectuado

O processador começa a executar a instrução que se segue ao salto



Se o salto é efectuado, o *pipeline* é limpo e recomeça na instrução correcta



Previsão do comportamento de um salto condicional (1)

Estratégias usadas para prever o efeito de um salto condicional

Fixa O salto não será efectuado

O processador continua a executar as instruções sequencialmente

Ciclos Saltos de volta ao início do ciclo serão efectuados

Se o salto for para trás (para o início do ciclo), o processador vai executar as instruções a partir do endereço para onde o salto é efectuado, caso contrário continua sequencialmente

Passado O salto terá o mesmo comportamento que da vez anterior

O processador prevê que a instrução terá o mesmo efeito que teve na última vez que foi executada

São guardados o endereço e o destino dos saltos

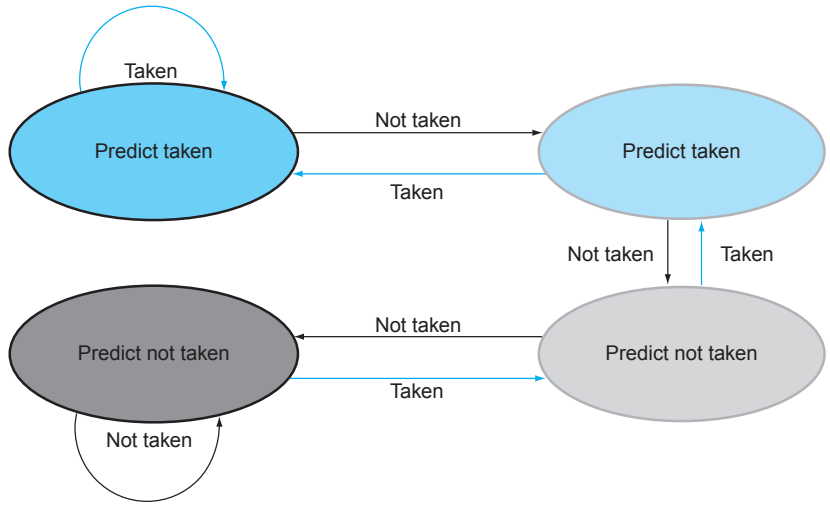
Erra duas vezes em cada ciclo

Previsão do comportamento de um salto condicional (2)

Estratégias usadas para prever o efeito de um salto condicional (cont.)

Histórica Usa a **história** do comportamento da instrução

O efeito da instrução é previsto recorrendo ao seu comportamento passado



Global (*correlating predictor*) Escolhe estratégia baseado em **outros saltos**

Por exemplo, pode usar uma previsão histórica se o salto anterior foi efectuado e outra se não foi

Torneio (*tournament branch predictor*) Escolhe estratégia com **maior sucesso**

Recorre à estratégia que, até ao momento, se revelou mais precisa

Os saltos condicionais e o *pipeline*

Resumo

O endereço da instrução a executar a seguir a uma instrução de salto condicional `beq` só é conhecido depois de o processador ter tido oportunidade de ler os valores nos registos e de os comparar

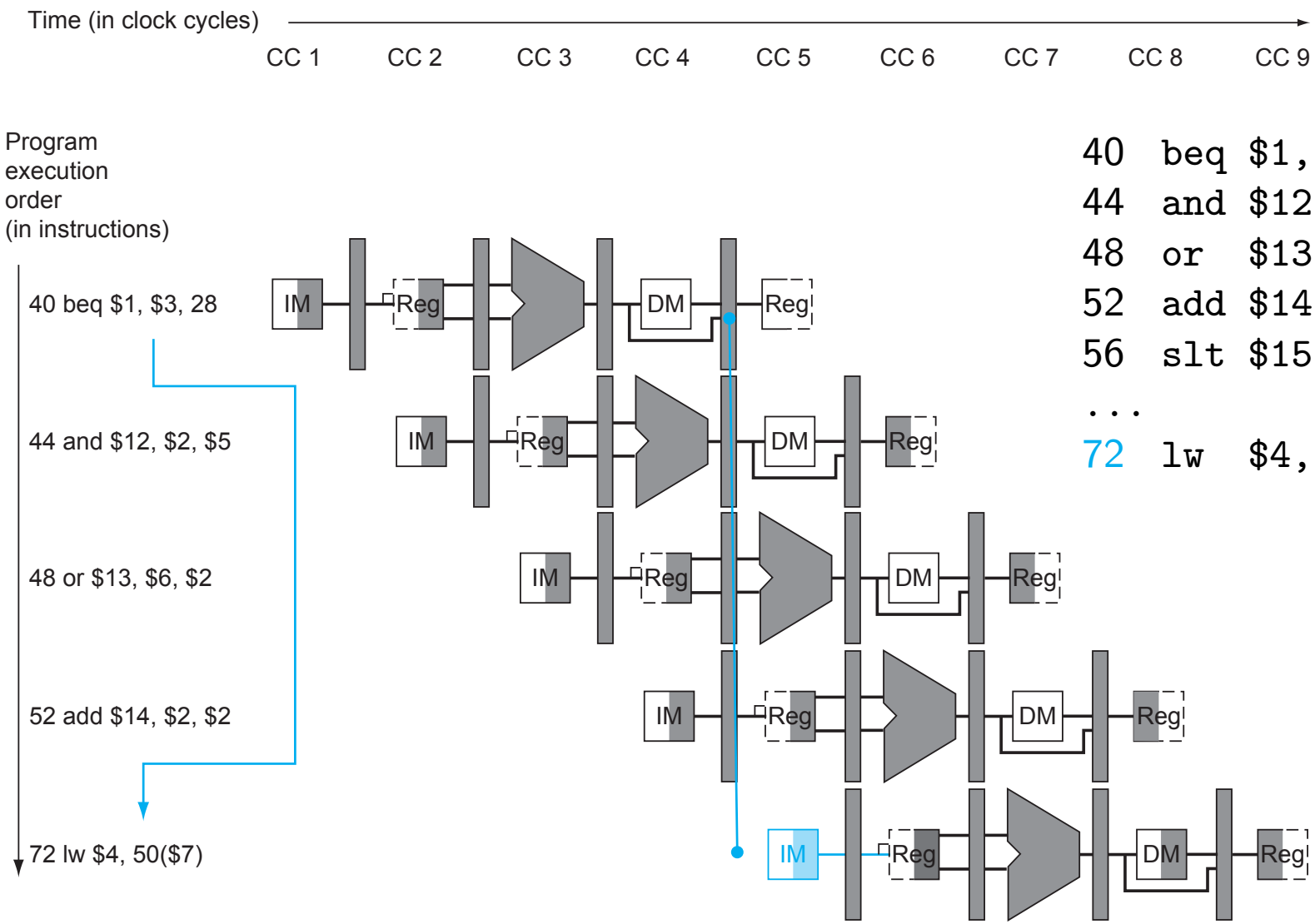
O processador vai continuar a introduzir instruções no *pipeline*, antes de saber o endereço da instrução que deverá ser executada, para não desperdiçar ciclos de relógio

Para minimizar o trabalho feito em vão quando as instruções executadas não são as que deveriam ser, o processador tenta adivinhar (ou prever) se o salto será efectuado

Se a previsão estiver errada, parte do trabalho do processador foi inútil e o *pipeline* tem de ser limpo

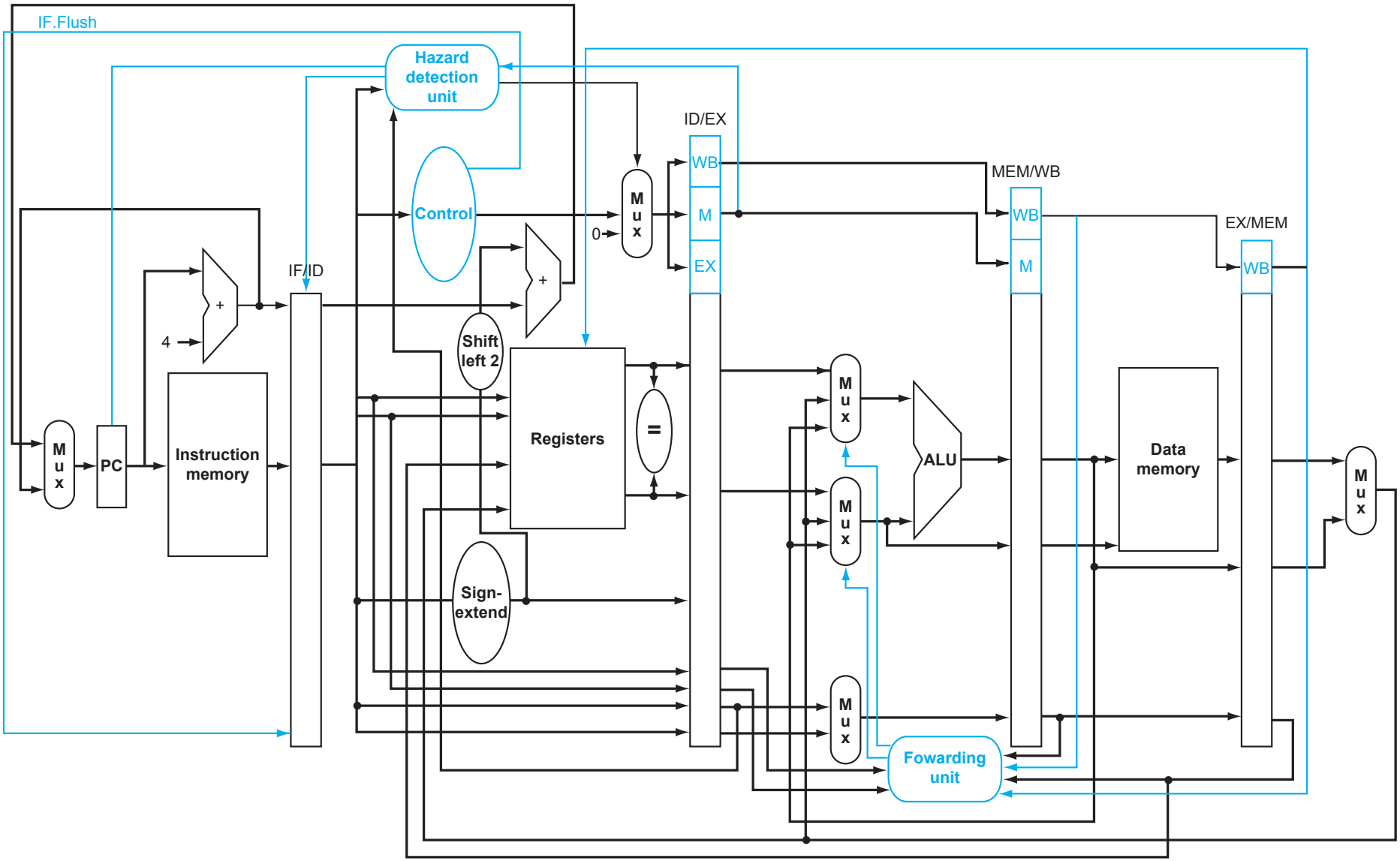
Quanto mais precisa for a previsão, menos serão os ciclos desperdiçados em trabalho inútil e melhor será o desempenho

MIPS com decisão dos saltos condicionais em MEM




```
40  beq $1,  $3, 7
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)
```

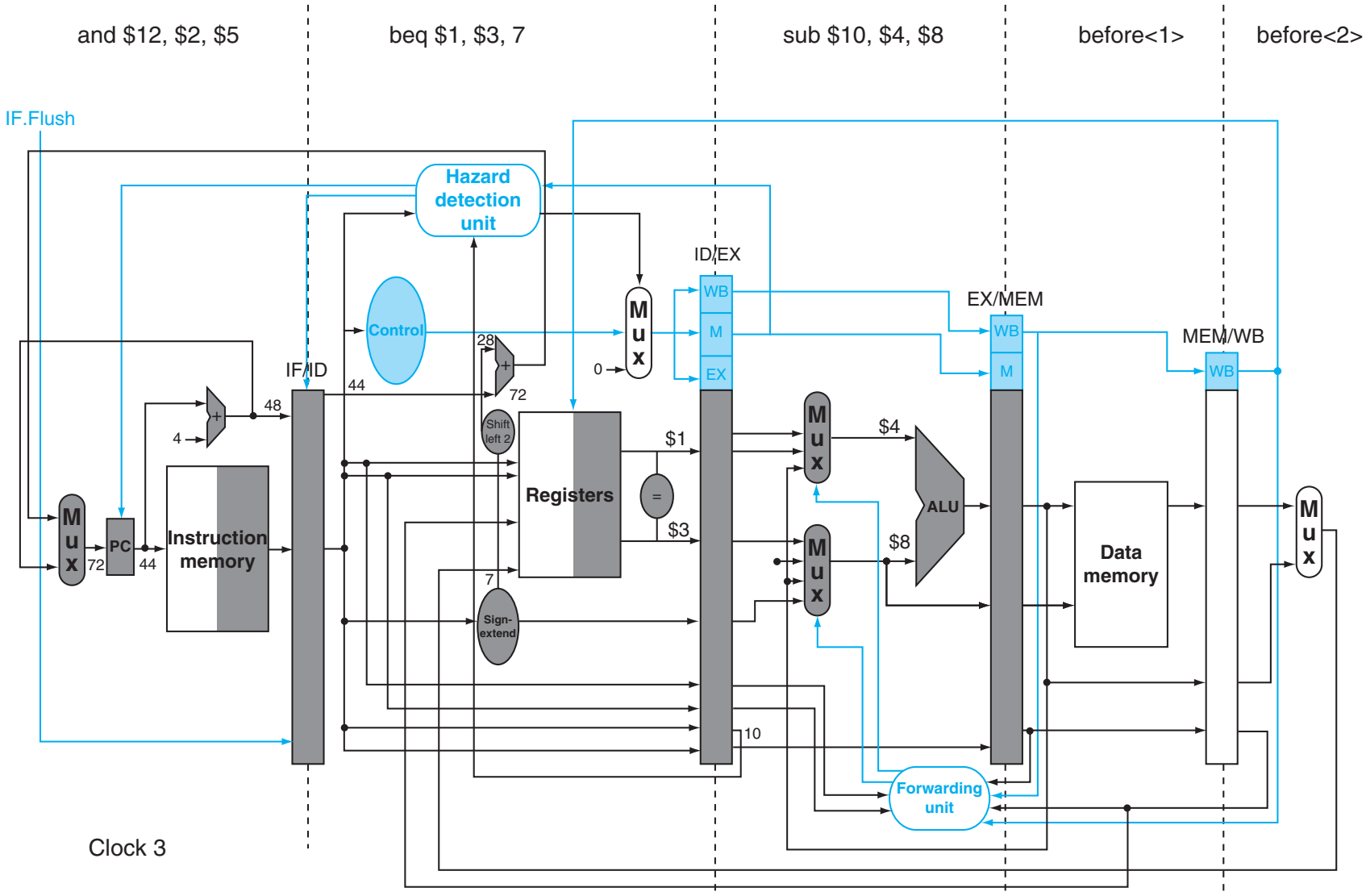
Pipeline MIPS com decisão dos saltos condicionais em ID



Faltam o mux(ALUSrc) e vários sinais de controlo

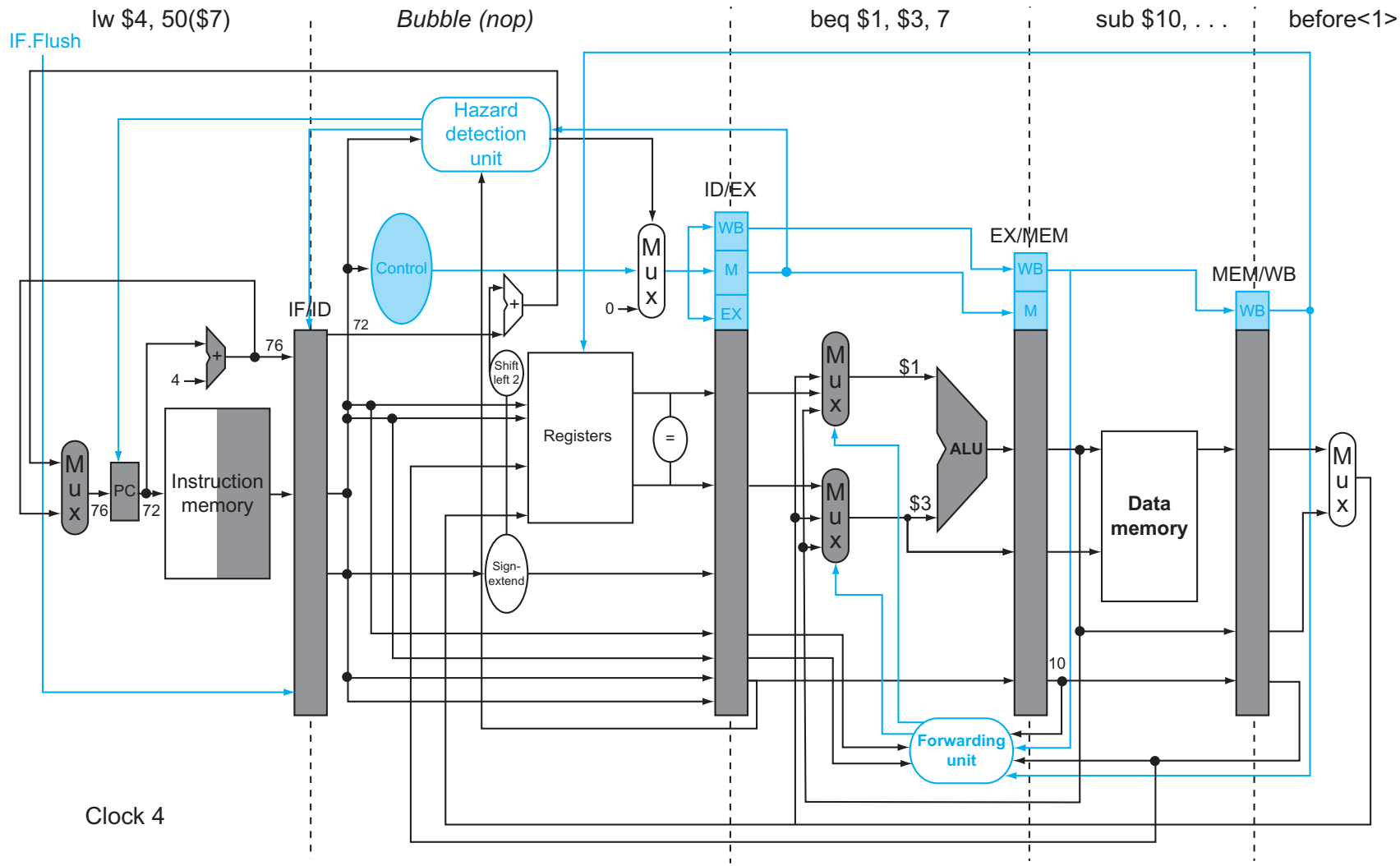
O *pipeline* na presença de saltos condicionais (1)

Passando a decisão de `beq` para o andar ID, haverá no máximo uma outra instrução no *pipeline* quando é decidido se o salto será efectuado



O *pipeline* na presença de saltos condicionais (2)

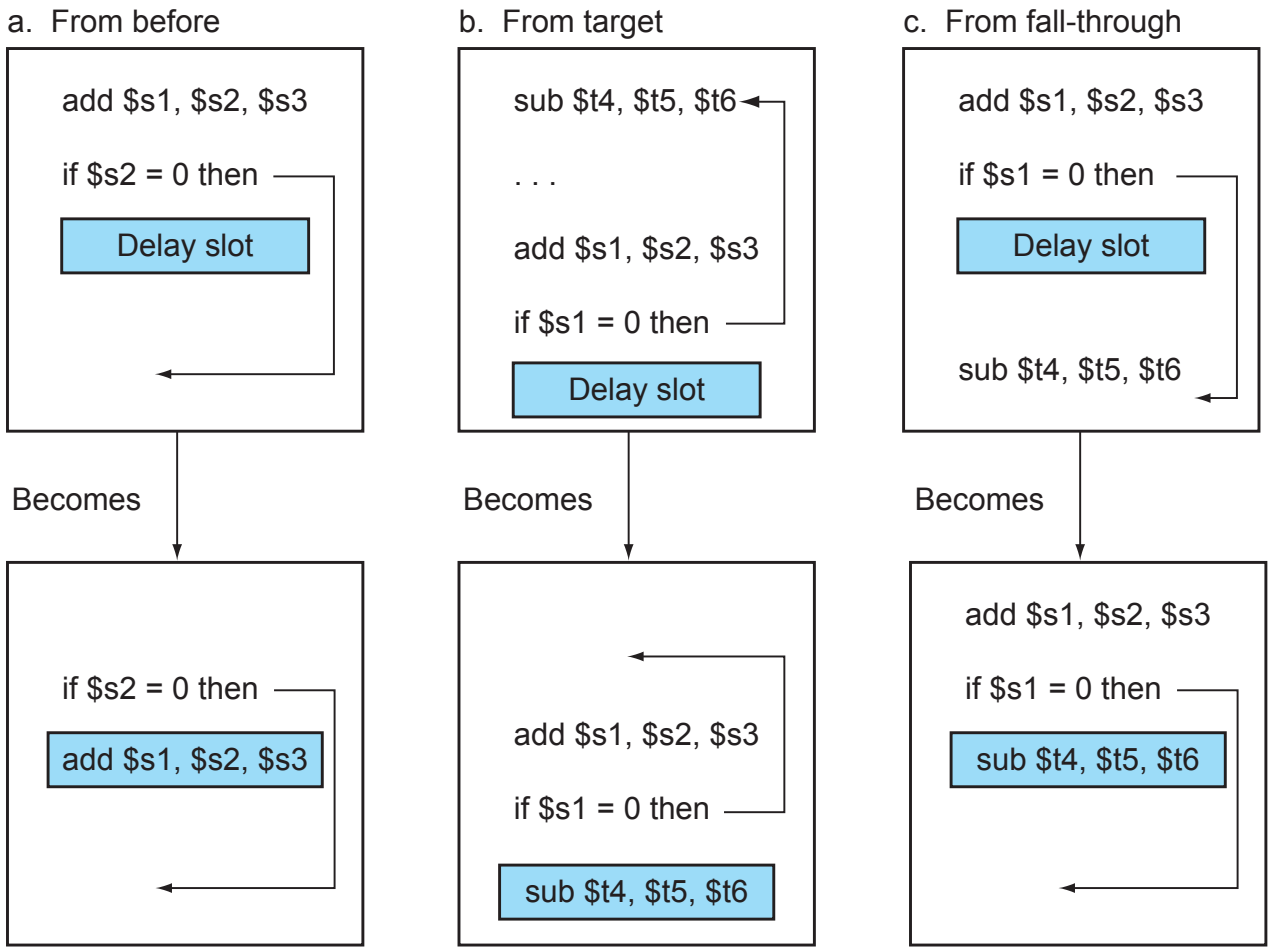
Sem o *delay slot*, se o salto fosse efectuado, a instrução incorrecta seria substituída por um `nop`



Redução do desperdício de ciclos de relógio no MIPS

1. Decide se efectua ou não o salto no andar ID do *pipeline*
2. Usa *delayed branching* com um *delay slot* (executa sempre a instrução que se segue a uma instrução de salto)

Onde obter uma instrução para ocupar o *delay slot*?



Compreender o funcionamento da máquina (2)

Como se explica?

```
#define SIZE 32768
int array[SIZE];

main()
{
  for (int i = 0; i < SIZE; ++i)
    array[i] = rand() % 256;

  for (int t = 0; t < 10000; ++t)
    {
      int s = 0;

      for (int i = 0; i < SIZE; ++i)
        if (array[i] >= 128)
          s += array[i];
    }
}
```

← `qsort(array, ...);`

Tempo de execução

4.509 s

2.032 s