Pipeline

Luciano L. Caimi

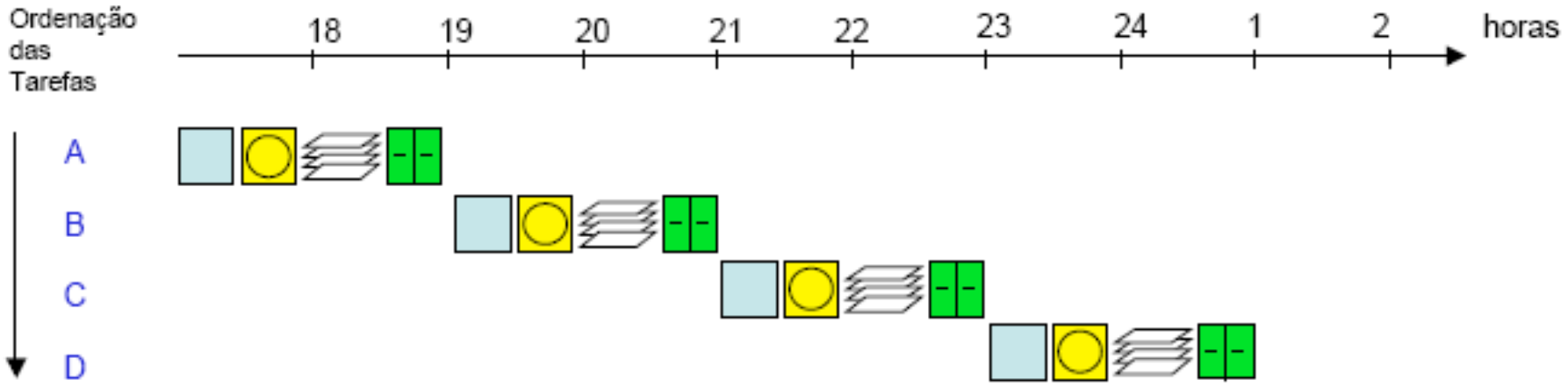
lcaimi@uffs.edu.br

(material original Prof. Edson Moreno - PUCRS)

Organização do MIPS: pipeline

- Visão geral do pipeline
  - Analogia com uma "Lavanderia doméstica" 1

- **Execução seqüencial**



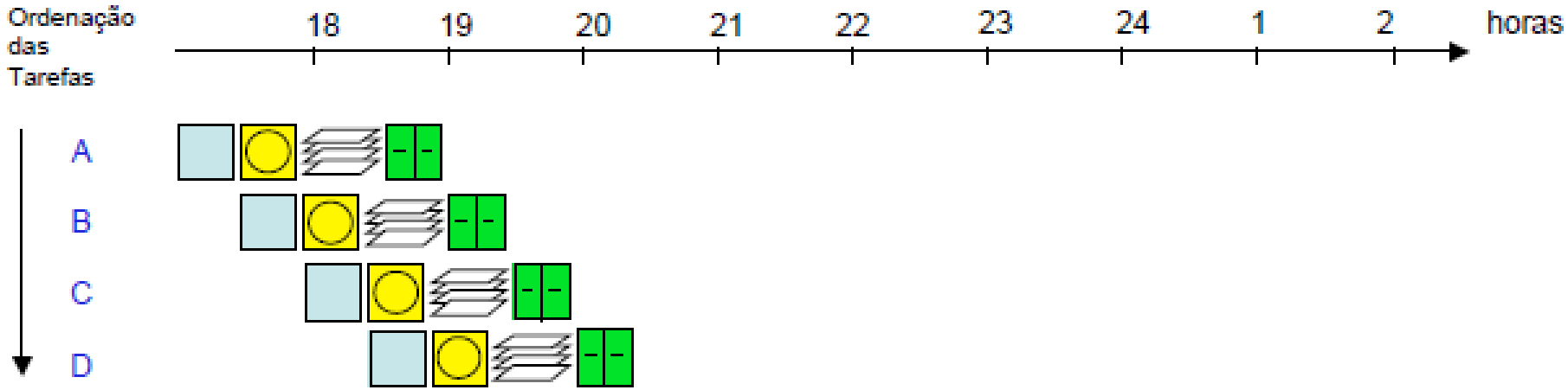
Tempo total: 8 horas

Tempo entre duas mudas de roupas prontas: 2 horas

Organização do MIPS: pipeline

- Visão geral do pipeline
  - Analogia com uma "Lavanderia doméstica" 2

Execução em pipeline



Tempo total: 3,5 horas

Tempo entre duas mudas de roupas prontas: 1/2 hora

Visão geral do pipeline

- Pipelining
  - Técnica aplicada para viabilizar o aumento de desempenho
  - Busca de paralelismo em nível de instrução
  - Suporte dado em nível de hardware
    - Solução "invisível" ao programador

- Princípio básico
  - Divide uma tarefa em N estágios
  - N tarefas executadas em paralelo, uma em cada estágio

Visão geral do pipeline

- Tradicionalmente, as instruções do MIPS são executadas em 5 etapas:
  - 1. Busca da instrução na memória
  - 2. Leitura dos registradores, enquanto a instrução é decodificada
  - 3. Execução de uma operação ou cálculo de um endereço
  - 4. Acesso à memória (load ou store)
  - 5. Escrita do resultado em um registrador
- No MIPS com Pipeline cada etapa é executada em um estágio do Pipeline.

Desempenho do pipeline

- Exemplo 1: compare o tempo de execução da versão monociclo do MIPS com o tempo da versão pipeline.
  - Considere os tempos de operação para as principais unidades funcionais (e os tempos das instruções) dados na tabela abaixo:

Classe da instrução	Busca da instrução	Leitura de registrador	Operação na ULA	Acesso ao dado	Escrita no registrador	Tempo total
Load word (`ld`)	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
Formato R (`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

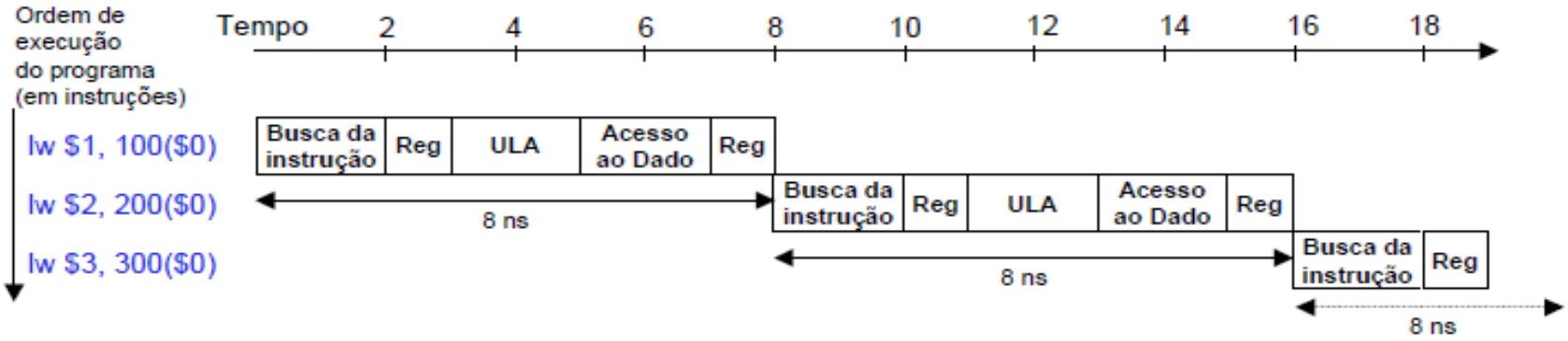
Desempenho do pipeline

- Consideremos 3 execuções seguidas da instrução lw, qual é a mais lenta em ambos casos (monociclo e pipeline).

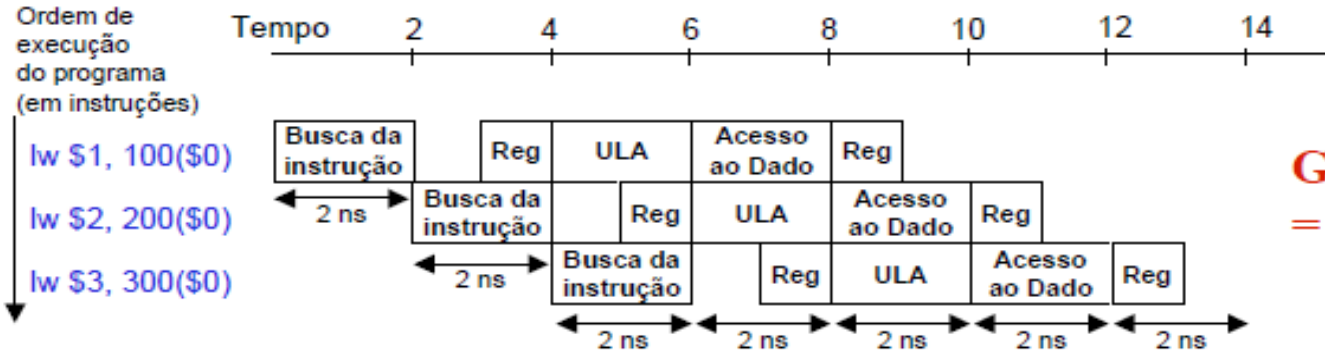
Classe da instrução	Busca da instrução	Leitura de registrador	Operação na ULA	Acesso ao dado	Escrita no registrador	Tempo total
Load word (`ld`)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (`sw`)	2 ns	1 ns	2 ns	2 ns	---	7 ns
Formato R (`add, sub, and, or, slt`)	2 ns	1 ns	2 ns	---	1 ns	6 ns
Branch (`beq`)	2 ns	1 ns	2 ns	---	---	5 ns

Desempenho do pipeline

sem pipeline



com pipeline



Ganho do pipeline = 24/6 = 4

Desempenho do pipeline

- **Sem pipeline**
  - O tempo decorrido entre o início da primeira instrução e o início da quarta instrução é 3x8 ns = 24 ns

- **Com pipeline**
  - O tempo decorrido entre o início da primeira instrução e o início da quarta instrução é 3x2 ns = 6 ns

- Logo, a melhora do desempenho advinda do uso do pipeline é de 24 ns/6 ns = **4 vezes**!

Desempenho do pipeline

- Se os estágios de um pipeline forem perfeitamente balanceados, então:

$$\text{Tempo entre instruções}_{\text{pipeline}} = \frac{\text{Tempo entre instruções}_{\text{não-pipeline}}}{\text{Número de estágios do pipe}}$$

- Estas condições são ideais e portanto, nunca atingidas!
- Porém, costuma-se assumir que o ganho máximo teórico de um pipeline é igual ao número de estágios.

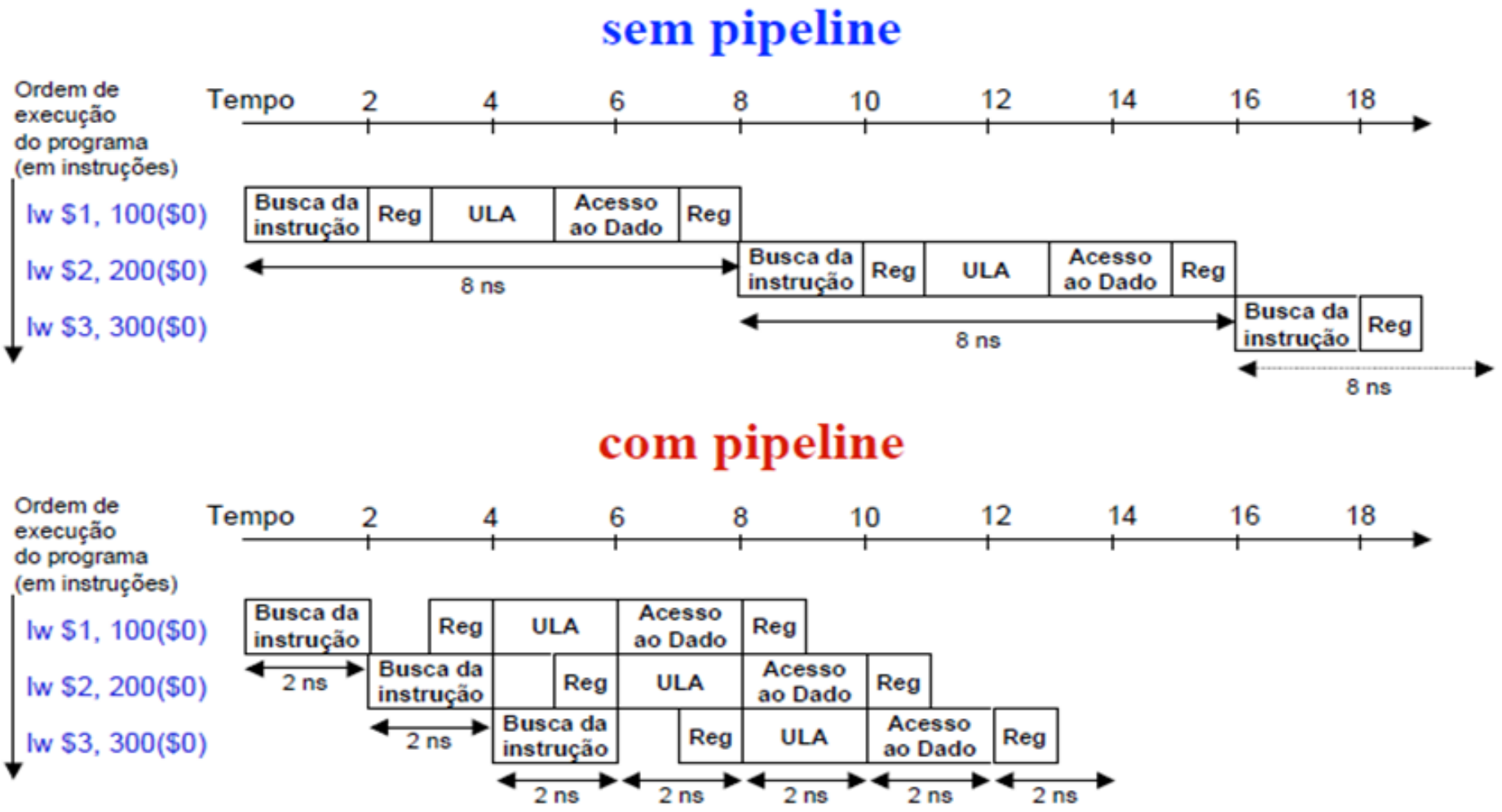
Desempenho do pipeline

- Motivos pelos quais o ganho teórico não é atingido:
  - Os estágios de um pipeline não são perfeitamente balanceados
    - Monociclo = tempo entre instruções = 8 ns
    - Pipeline
      - Tempo estimado entre instruções = 1,6 ns
      - Tempo real entre instruções = 2 ns (estágio de pior tempo)
  - Existe um *overhead* referente ao preenchimento do pipeline.
- Em função do segundo motivo, o tempo entre instruções é mais importante do que o tempo total.

Desempenho do pipeline

- Exemplo 2: Ainda com relação ao exemplo 1, considere que ao invés de 3 instruções LW, sejam executadas 1003 instruções LW seguidas.
  - Qual será o tempo total em cada caso (monociclo e pipeline) e qual será o ganho obtido pelo uso do pipeline?

Desempenho do pipeline



Desempenho do pipeline

- **Sem pipeline**
  - 24 ns + (1000 x 8 ns) = 8024 ns
- **Com pipeline**
  - 14 ns + (1000 x 2ns) = 2014 ns
- **Ganho** = 8024 ns / 2014 ns = 3,98

Desempenho do pipeline

- Qual efeito causado pelo pipeline que garante aumenta o desempenho?
  - Freqüência de relógio?
  - Vazão?
  - Tempo de resposta?
  - Feitiçaria?

Desempenho do pipeline

- O pipeline aumenta o desempenho por meio do aumento da vazão (*throughput*) das instruções, ou seja, aumentando o número de instruções executadas por unidade de tempo (e não por meio da diminuição do tempo de execução de uma instrução individual).

Organização do MIPS pipeline

- O conjunto de instruções do MIPS foi projetado visando a execução em pipeline:
  - Todas as instruções têm o mesmo tamanho
  - Poucos formatos de instruções, sempre com o registrador-fonte localizado na mesma posição
  - Somente as instruções load word e store word manipulam operandos na memória

Conflitos do pipeline

- Existem situações de execução no pipeline em que a instrução seguinte não pode ser executada no próximo ciclo de relógio. Tais situações são chamadas de conflitos.
- Tipos de Conflitos:
  - Estruturais
  - De controle
  - De dados

Conflitos estruturais (Structural Hazards)

- Acontecem quando o Hardware não pode suportar a combinação de instruções que o pipeline deseja executar em um dado ciclo de relógio.
- Se houvesse somente uma memória (para dados e instruções) e se uma instrução tentasse acessar um dado na memória enquanto tivesse que ser buscada uma nova instrução, ocorreria um conflito estrutural.

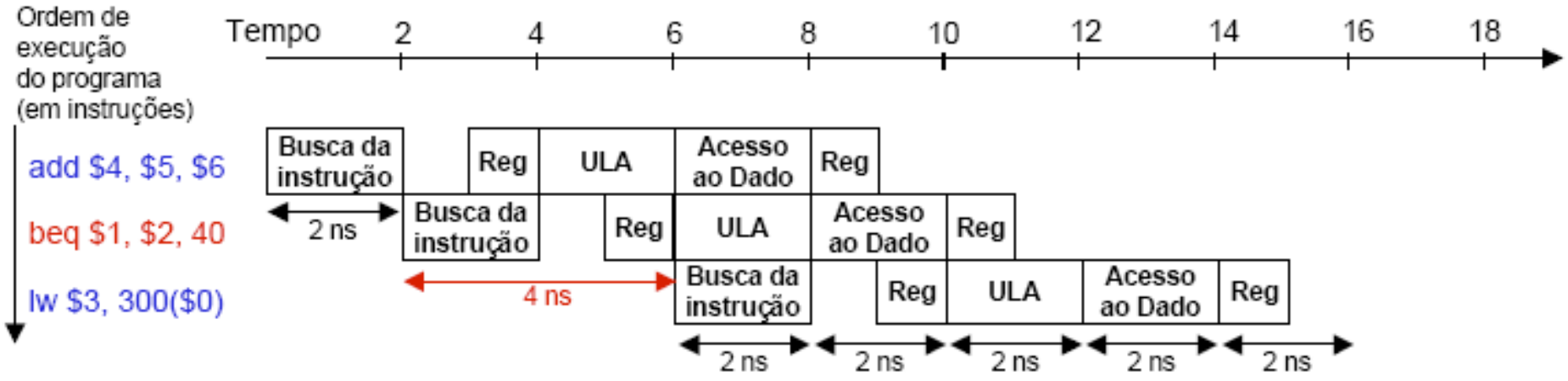
Conflitos de controle (Control Hazards)

- Originam-se na necessidade de se tomar uma decisão baseada nos resultados de uma instrução, a qual ainda não foi concluída.
- Muito comuns em instruções de salto condicional (beq, no caso do MIPs reduzido)
- Uma possível solução é a parada (também chamada de "bolha"), ou seja, interromper a progressão das instruções pelo pipeline.

Conflitos de controle (Control Hazards)

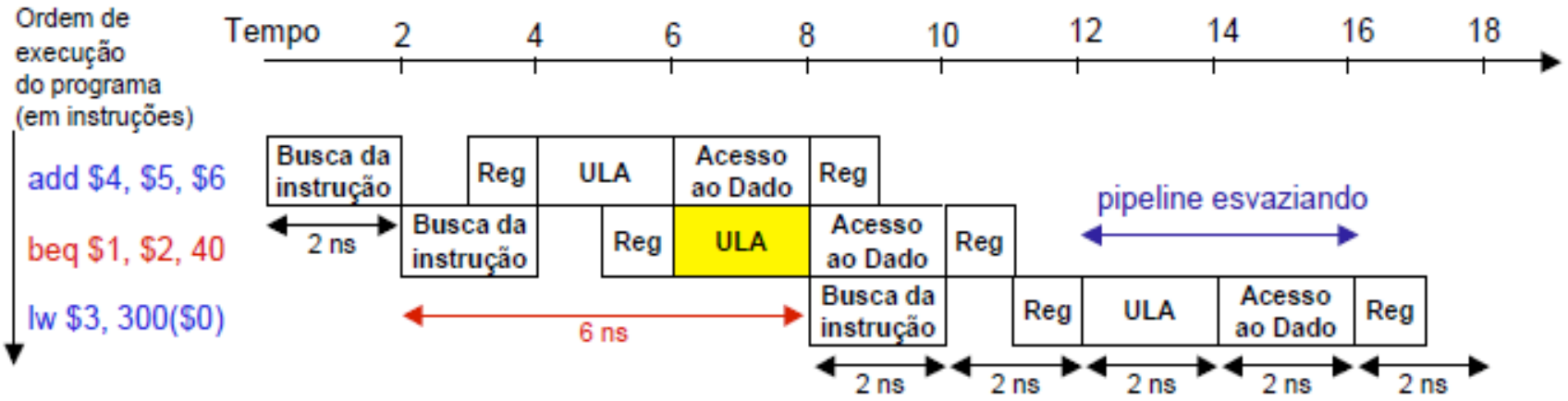
- Exemplo: analisar o efeito da parada no desempenho do desvio condicional.

Considere que se tenha *hardware* extra que permita testar registradores, calcular o endereço de desvio condicional e atualizar o PC, tudo isso no segundo estágio.



Conflitos de controle (Control Hazards)

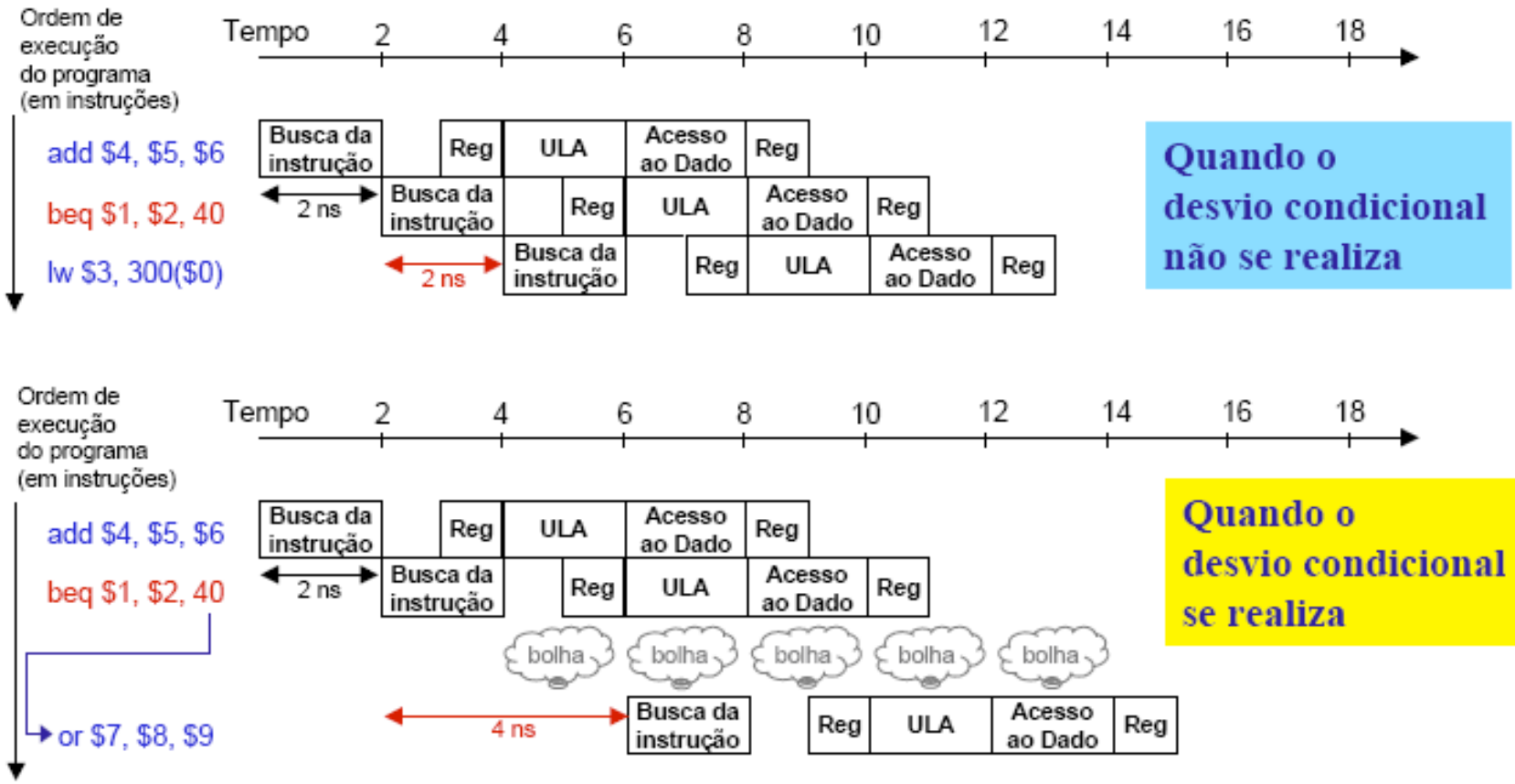
- Se for necessário resolver o desvio condicional em um estágio posterior ao segundo (o que é o ocorre na maioria dos processadores reais), ocorrerá uma perda maior de desempenho em função da necessidade de encher novamente o pipeline



Predição estática

- **Outra solução para o conflito de controle**
- É a técnica geralmente adotada nos processadores reais!
- Um esquema simples de predição é assumir sempre que os desvios condicionais sempre irão falhar.
  - Se acertar, o pipeline prossegue com velocidade máxima
  - Se errar, irá atrasar o avanço das instruções pelo pipeline

Predição estática



Predição estática

- Exemplo de loop:

```
…
add …
meu_laco:
    ble $t0, $t1, fim_laco
    lw …
    sub …
    mul …
    j meu_laco
fim_laco:
sw …
add …
```

loop

Predição estática

- Outro esquema mais sofisticado de predição consiste em se considerar parte dos desvios se realizando e parte não se realizando

- Exemplo de uso prático: nos loops a última instrução é sempre um desvio para um endereço anterior, que na maioria das vezes, se concretiza

- Dada a grande probabilidade de se realizarem, podemos predizer que os desvios para endereços anteriores sempre se realizam…

Predição estática

- Exemplo de loop:

```
…
add …
go_laco:
add …
lw …
sub …
mul …
ble $t0, $t1, go_laco
sw …
add …
```

loop

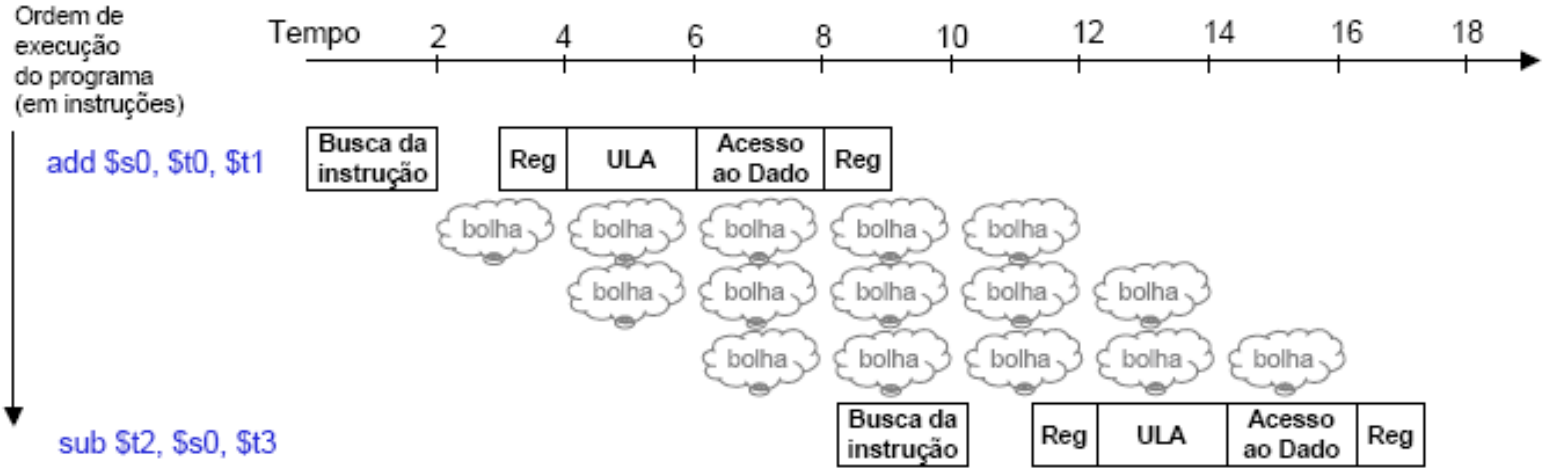
Predição dinâmica

- **Outra solução para o conflito de controle**
- Predição dinâmica realiza a predição em função do comportamento anterior de cada desvio
- Podem mudar a predição para um mesmo desvio com o decorrer da execução do programa
- Um esquema comum é manter um histórico para cada desvio realizado ou não-realizado
- Preditores dinâmicos são implementados em hardware e representam entre 99% e 82% de acerto

Conflitos de dados (Data Hazards)

- A execução de uma instrução depende do resultado de outra, a qual ainda está no pipeline.
- Exemplo:

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



Conflitos de dados (Data Hazards)

- Tipos de conflito de dados
  - Dependência verdadeira
    - Leitura após escrita (RAW)
    - Um operando é modificado a partir de uma instrução e lido em seguida
  - Dependências falsas
    - **Anti Dependência**
      - Escrita após leitura (WAR)
      - Lê um operando a partir de uma instrução e escreve logo em seguida
    - **Dependência de saída** (WAW)
      - Escrita em um mesmo registrador a partir de instruções consecutivas

Conflitos de dados (Data Hazards)

- Dependência verdadeira
  - *Read after write* (RAW)
  - Refere-se a uma situação a qual o resultado ainda não está calculado

  - Exemplo

```
instr1:   ADD $1, $2, $3
instr2:   SUB $4, $1, $5
```

    - Ou seja, a instr1 estará calculando o valor a ser armazenado no registrador $1 e a instr2 irá realizar a leitura do valor contindo no registrador $1 para executar sua operação
    - Problema: quando a instr2 busca o registrador $1 no segundo estágio do pipeline, este ainda está sendo computado pela instr1

Conflitos de dados (Data Hazards)

- Ante dependência
  - *Write after read* (WAR)
  - É um problema relaciona com 'condições de corrida' em sistemas de execução concorrente

  - Exemplo

```
instr1:   LW $1, 0($2)
instr2:   SUB $2, $4, $5
```

    - Se a arquitetura implementada permite que ocorram execuções em paralelo, e supondo que a instr1 sofra um trancamento por uma dependência anterior ou tempo de execução mais longo, a instr2 poderia ser executada antes, alterando o valor a ser lido pela instr1

Conflitos de dados (Data Hazards)

- Dependência de saída
  - *Write after write* (WAW)
  - É um problema que também está relacionado com 'condições de corrida' em sistemas de execução concorrente

  - Exemplo

```
instr1:   ADD $2, $3, $4
instr2:   SUB $2, $5, $6
```

    - A operação sucessiva de escrita em um mesmo registrador destino torna desnecessária a instrução executada anteriormente.

Conflitos de dados (Data Hazards)

- Dependências verdadeiras
  - O pipeline precisa ser parado durante certo número de ciclos
  - Colocação de instrução de "NOP" ou escalonamento adequado das instruções pelo compilador
  - O adiantamento dos dados pode resolver em alguns casos

- Dependências falsas
  - Não é um problema em pipelines onde a ordem de execução das instruções é mantida
  - Problema em processadores superescalares
  - A renomeação dos registradores é uma solução usual neste problema

Conflitos de dados (Data Hazards)

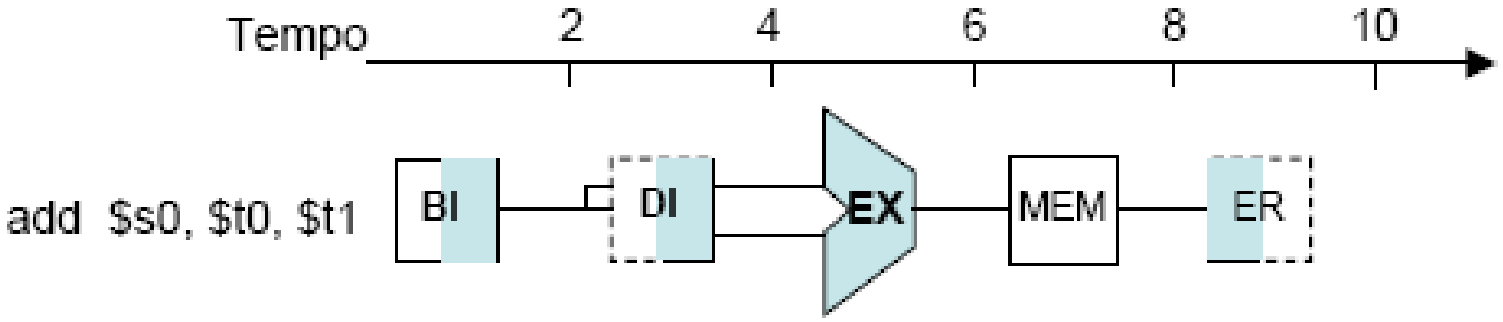
- Uma solução: Adiantamento (Forwarding/Bypass)
  - Solução:Não é preciso esperar pelo término de uma instrução aritmética/lógica.
  - Tão logo a ULA chegue ao resultado da operação, este resultado pode ser disponibilizada para as instruções que vem a seguir.

Forwarding

- Exemplo:
  - Considerando as duas instruções abaixo, mostre as conexões necessárias entre os estágios do pipeline de modo a tornar viável o adiantamento.

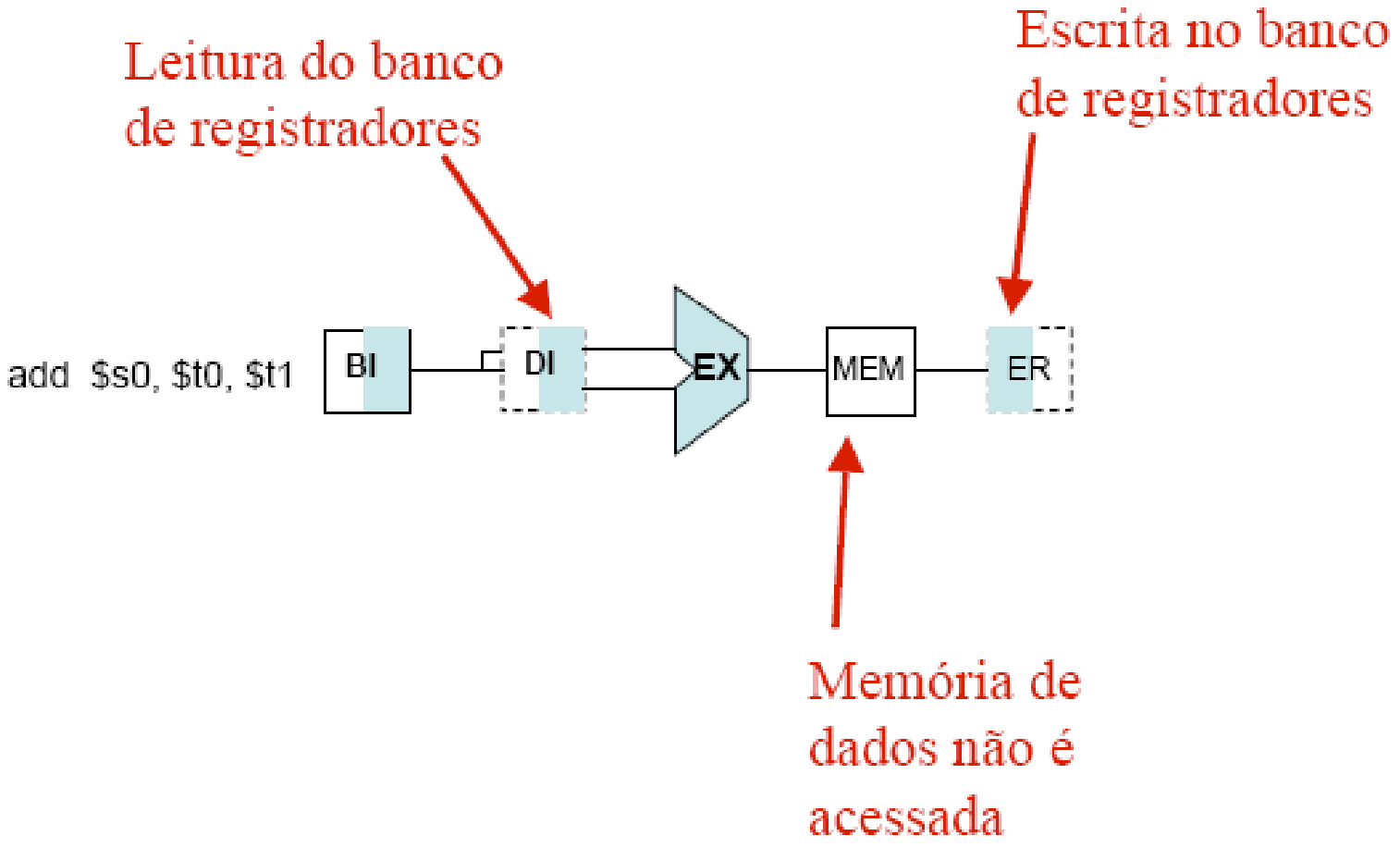
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Forwarding



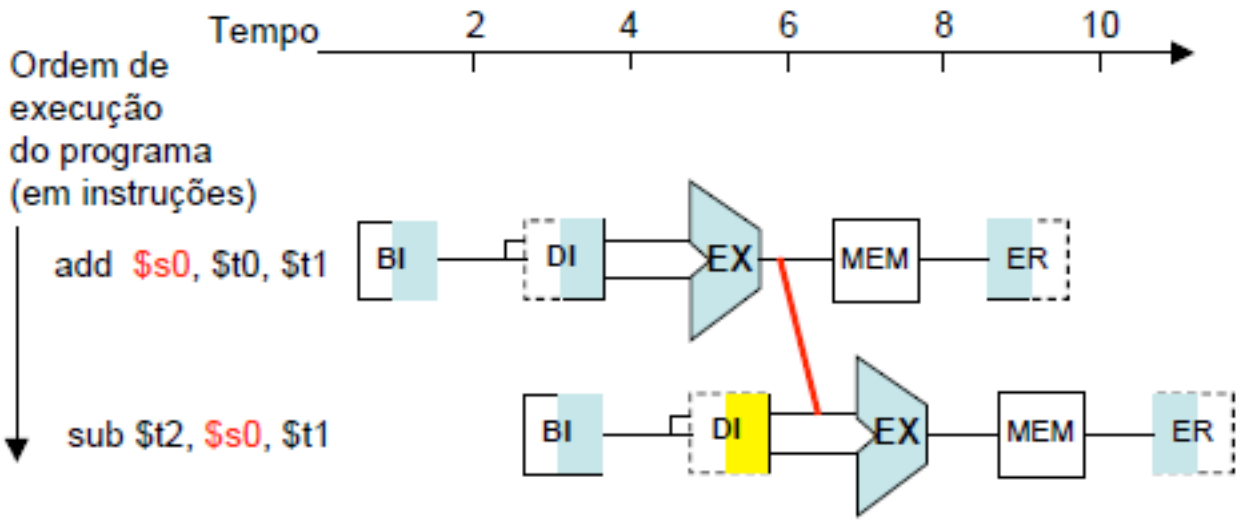
- Estágios
  - BI – Busca de instrução (memória de instruções)
  - DI – Decodificação da instrução e leitura do banco de registradores
  - EX – Execução da instrução (ULA)
  - MEM – Acesso a memória (leitura/escrita)
  - ER – escrita do resultado no banco de registradores

Forwarding


Leitura do banco de registradores
Escrita no banco de registradores
add $s0, $t0, $t1
BI
DI
EX
MEM
ER
Memória de dados não é acessada

Forwarding

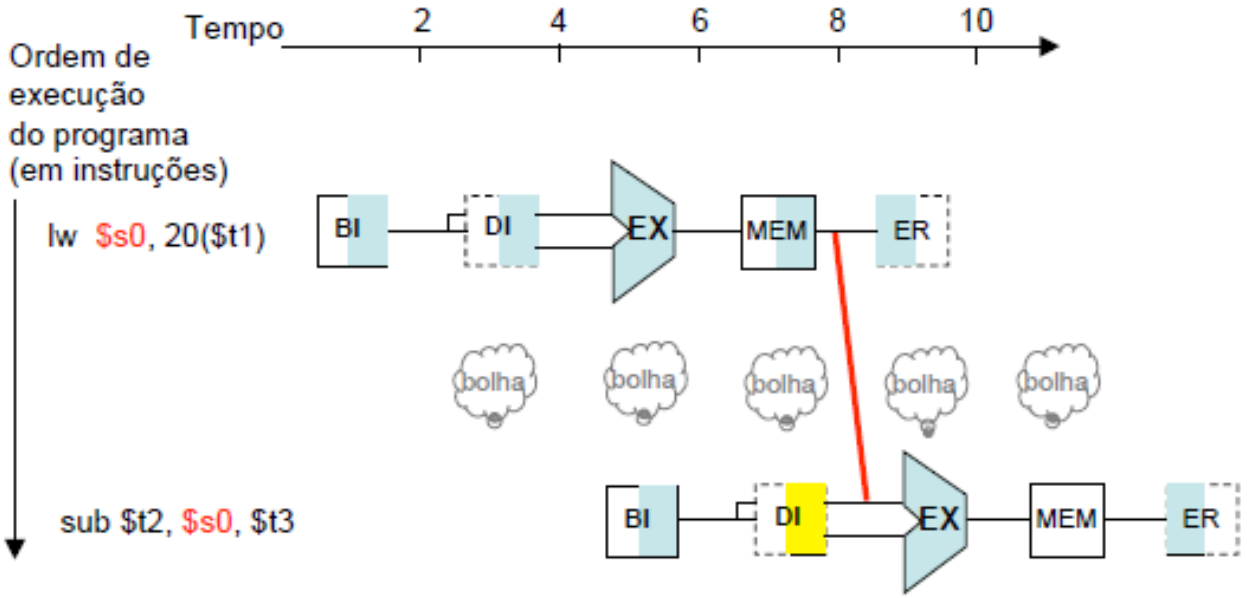
- Conexão entre a saída da ULA e a entrada da própria ULA



- Esta solução evita bolhas

Forwarding

- Conexão entre a saída da memória de dados e a entrada da ULA
- Utilizada para instruções do tipo load



- Mesmo com adiantamento, havera a perda de um ciclo de relogio (a bolha inevitável)

Reordenação do código

- Outra solução para os conflitos de dados
- Exemplo: considere o trecho de código abaixo

```
                    # reg $t1 possui o endereco de v[k]
lw $t0, 0($t1)      # reg $t0 (temp) = v[k]
lw $t2, 4($t1)      # reg $t2 = v[k+1]
sw $t2, 0($t1)      # v[k] = reg $t2
sw $t0, 4($t1)      # v[k+1] = reg $t0 (temp)
```

- O conflito aparece no registrador $t2, entre a 2ª. e a 3ª. linha.

Reordenação do código

- Reordenando o código: invertendo as duas últimas linhas

```
                    # reg $t1 possui o endereco de v[k]
lw $t0, 0($t1)      # reg $t0 (temp) = v[k]
lw $t2, 4($t1)      # reg $t2 = v[k+1]
sw $t0, 4($t1)      # v[k+1] = reg $t0 (temp)
sw $t2, 0($t1)      # v[k] = reg $t2
```

- É possível reduzir o conflito se não houver adiantemente
- O conflito é extinto se houver adiantamento

Exercícios

- Dado o trecho de código abaixo sendo executado em um pipeline de 5 estágios (**[B]** Estágio de busca de instrução, **[D]** Estágio de decodificação da instrução e busca de operadores, **[E]** Estágio de execução, **[M]** Estágio de salvamento de resultado em memória, **[R]** Estágio de salvamento de resultado no registrador alvo) faça

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

1. Apresente graficamente a execução no pipeline sem forwarding

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

1. Apresente graficamente a execução no pipeline sem forwarding

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I0																									
I1																									
I2																									
I3																									
I4																									
I5																									
I6																									

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

1. Apresente graficamente a execução no pipeline sem forwarding

2. Apresente graficamente a execução no pipeline com forwarding

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

2. Apresente graficamente a execução no pipeline com forwarding

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I0																									
I1																									
I2																									
I3																									
I4																									
I5																									
I6																									

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

1. Apresente graficamente a execução no pipeline sem forwarding
2. Apresente graficamente a execução no pipeline com forwarding
3. Identifique os tipos de dependência de dados presentes no código

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

1. Apresente graficamente a execução no pipeline sem forwarding
2. Apresente graficamente a execução no pipeline com forwarding
3. Identifique os tipos de dependência de dados presentes no código
4. Proponha a reordenação do código e mostre graficamente sua execução no pipeline sem forwarding

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

4. Execução no pipeline sem forwarding pós reordenação

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I0																									
I1																									
I2																									
I3																									
I4																									
I5																									
I6																									

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

1. Apresente graficamente a execução no pipeline sem forwarding
2. Apresente graficamente a execução no pipeline com forwarding
3. Identifique os tipos de dependência de dados presentes no código
4. Proponha a reordenação do código e mostre graficamente sua execução no pipeline sem forwarding
5. Proponha a reordenação do código e mostre graficamente sua execução no pipeline com forwarding

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```

Exercícios

5. Execução no pipeline com forwarding pós reordenação

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I0																									
I1																									
I2																									
I3																									
I4																									
I5																									
I6																									

```
I0: LW  $T0,0($T9)
I1: ADD $T2,$T0,$T1
I2: SW  $T2,0($T8)
I3: LW  $T3,4($T9)
I4: ADD $T4,$T3,$T1
I5: ADD $T5,$T4,$T6
I6: SW  $T5,4($T8)
```