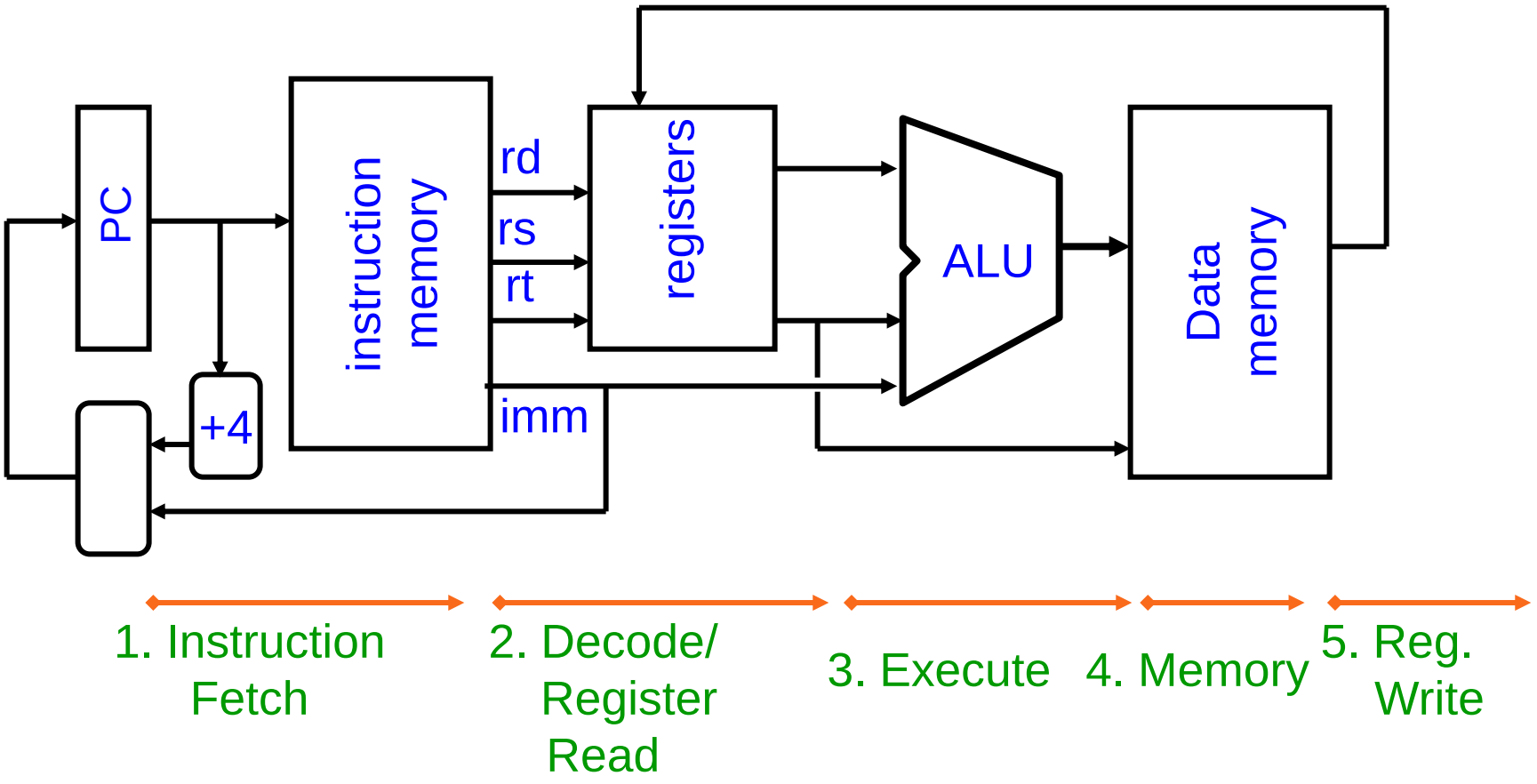
Introdução ao MIPS

- *Pipelining* para melhoria de Desempenho –

Arquitetura de Computadores 2024/2025

Datapath do MIPS



Resumo: Etapas do *Datapath* (1/6)

- O "*Instruction Set*" do MIPS é composto por instruções muito variadas: Quais serão as etapas que elas têm em comum?

- Etapa 1: *Instruction Fetch*
  - A *word* de 32-bits na qual a instrução é codificada tem de ser sempre lida a partir da memória (*instruction fetch*)
  - Para além disso o PC (*program counter*) tem de ser sempre incrementado para apontar para a instrução seguinte (PC = PC + 4)

Resumo: Etapas do *Datapath* (2/6)

- Etapa 2: *Instruction Decode*
  - Depois do *fetch*, é necessário fazer a descodificação da instrução e obter os dados associados a cada campo
  - Primeiro, ler o *opcode* para determinar o tipo de instrução e o tamanho dos campos
  - Segundo, ler os dados de todos os registos indicados de forma a definir os operandos
    - Para o `add`, lê-se dois registos
    - Para o `addi`, lê-se um único registo
    - Para o `jal`, não é necessário ler-se registos

Resumo: Etapas do *Datapath* (3/6)

- Etapa 3: ALU (Unidade de Lógica e Aritmética)
  - Na maior parte das instruções o trabalho efetivo é feito neste nível: aritmética (+, -, *, /), deslocamento, lógica (&, |), comparações (`slt`)
  - E quanto aos *loads* e *stores*?
    - `lw  $t0, 40($t1)`
    - Repare que é necessário calcular o endereço final através da adição de 40 (imediato) ao conteúdo do registo `$t1`
    - A adição para o cálculo do endereço é feita nesta etapa

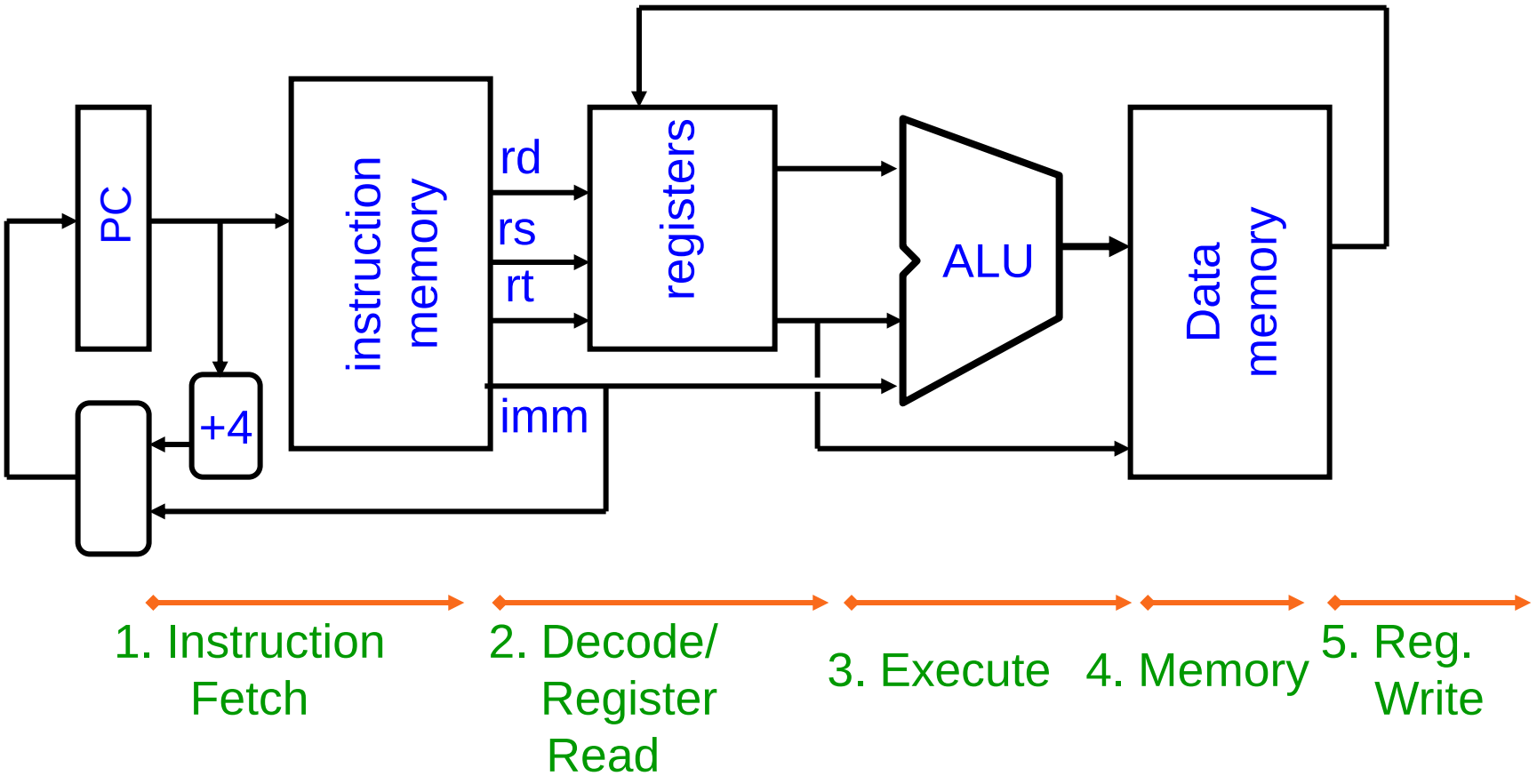
Resumo: Etapas do *Datapath* (4/6)

- Etapa 4: *Memory Access*
  - Somente as instruções *load* e *store* é que fazem trocas de informação com a memória (leitura e escrita); todas as outras instruções ficam inativas (*idle*) durante esta etapa.
  - Este é uma etapa incontornável para a implementação dos *loads* e *stores*. Assim, e apesar das outras instruções não terem este passo, o *datapath* tem de incluir esta etapa.

Resumo: Etapas do *Datapath* (5/6)

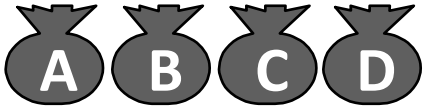
- Etapa 5: *Register Write*
  - A maioria das instruções escreve o resultado de uma determinada operação num registo destino.
  - exemplos: operações aritméticas e lógicas, deslocamentos, *loads*, `slt`
  - E quanto aos *stores*, *jumps* e *branches*?
    - Estas instruções não escrevem nenhum resultado num registo destino
    - São instruções que permanecem inativas durante esta etapa.

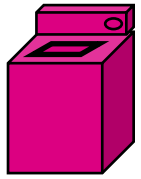
Resumo: Etapas do *Datapath* (6/6)

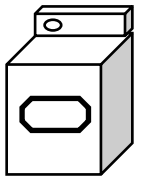


Analogia: Vamos lavar a roupa ...

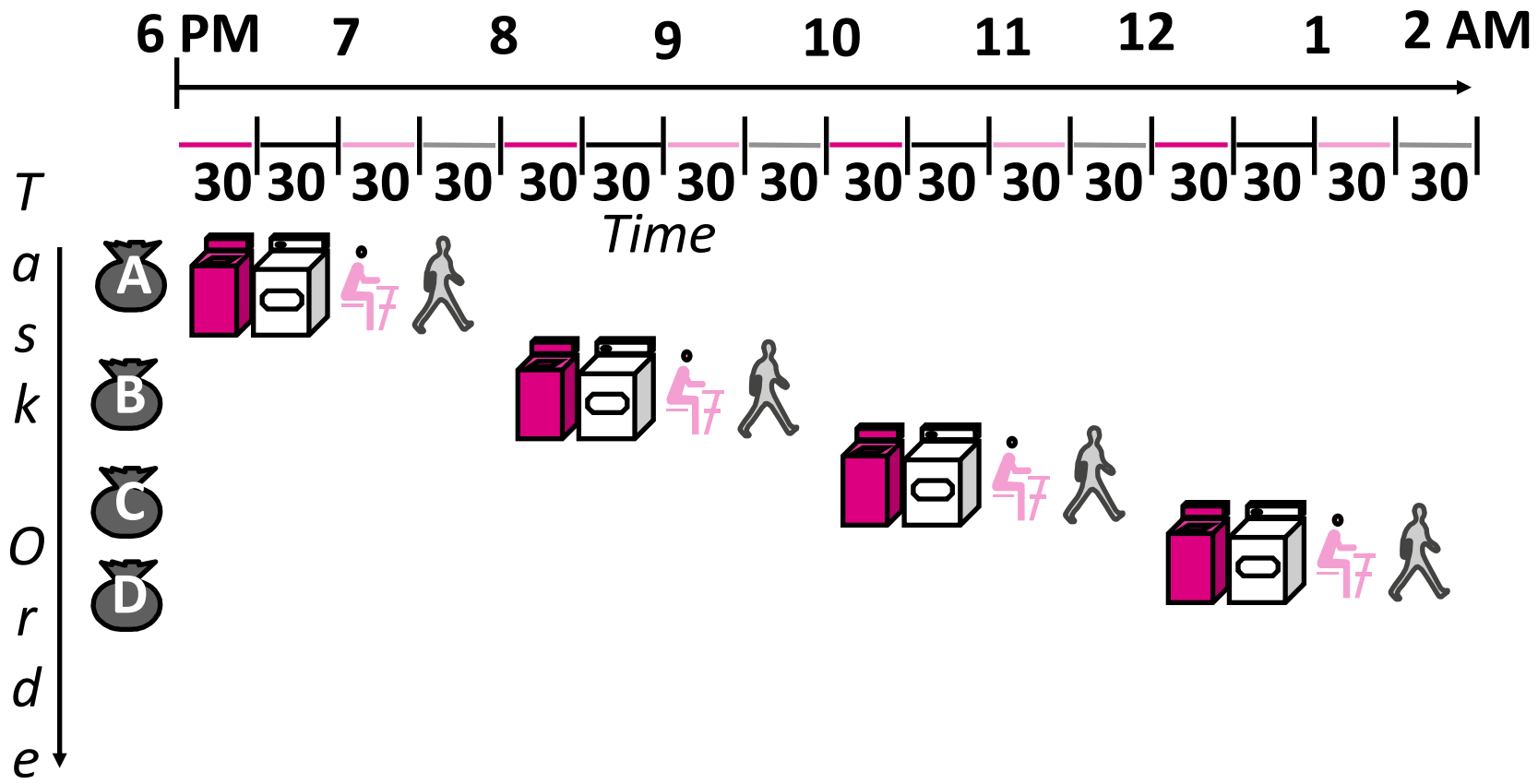
- A Ana, Bernardo, Carlos e Diana têm um saco de roupa suja para lavar, secar, dobrar e arrumar na gaveta.
- A máquina de lavar demora 30 minutos
- O secador de roupa demora 30 minutos
- A “dobragem” demora 30 minutos
- A “arrumação” na gaveta demora 30 minutos





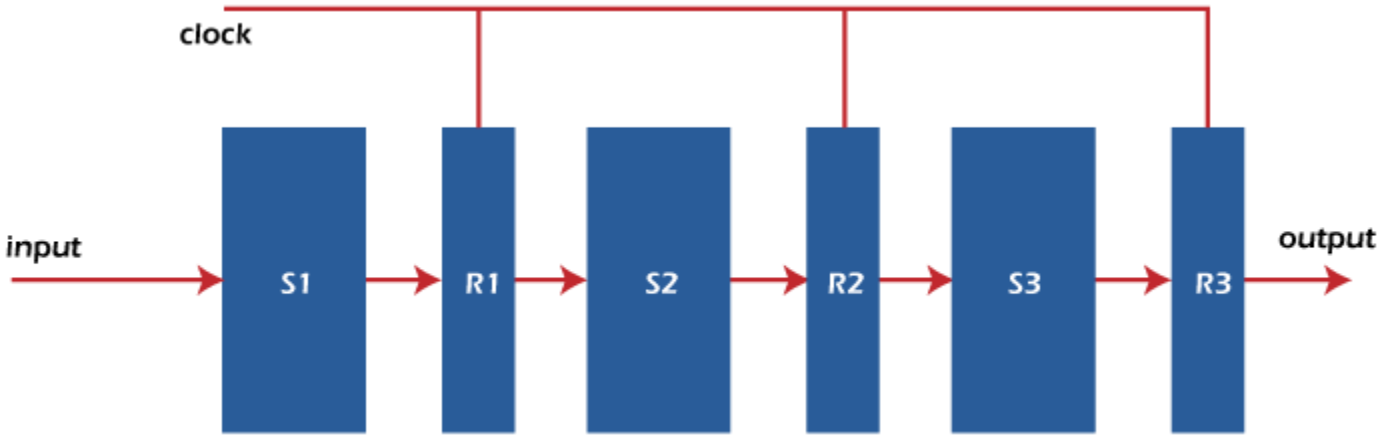


Operação Sequencial:



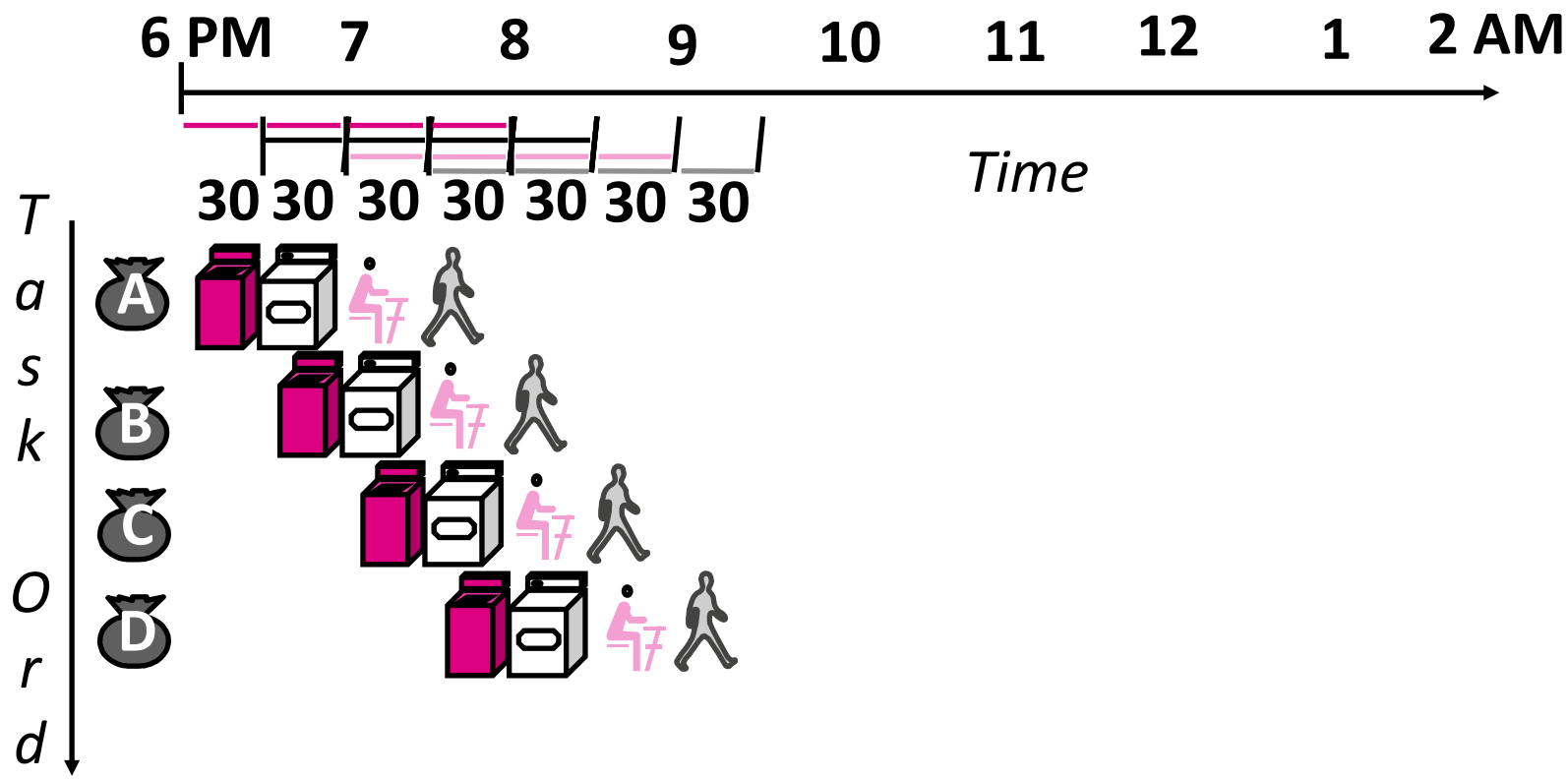
- Fazendo as coisas de forma sequencial demoramos um total de 8 horas para 4 cargas de roupa

O que é o *Pipeline*?



O *pipeline* é dividido em estágios lógicos conectados entre si para formar uma estrutura semelhante a um *pipeline*. As instruções entram por uma extremidade e saem pela outra.

Operação em *Pipeline*:



Executando em *pipeline* demoramos 3.5 horas para 4 cargas de roupa sem precisarmos de recursos adicionais (e.g. outra máquina de lavar ou secar)!

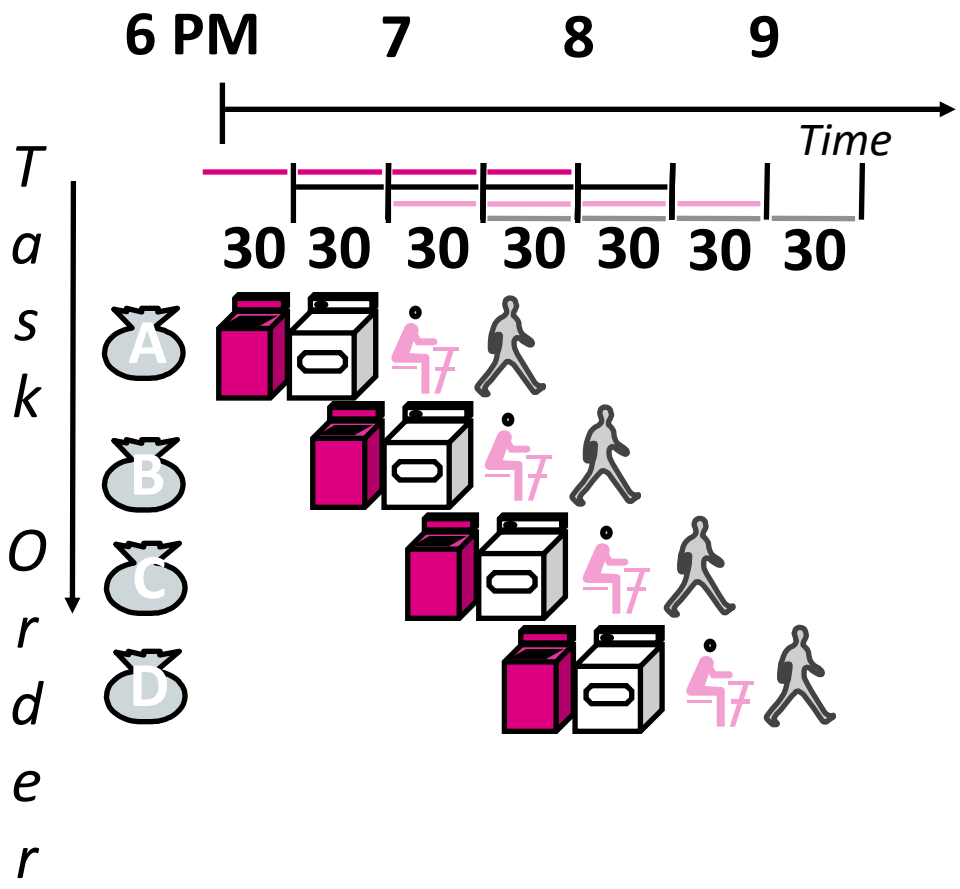
Definições:

- Latência: tempo necessário à execução de uma determinada tarefa
  - Exemplo: o tempo para ler um sector do disco é o tempo de acesso a disco ou latência do disco

- *Throughput* (Taxa de transferência): Quantidade de trabalho que conseguimos fazer durante um determinado período de tempo.

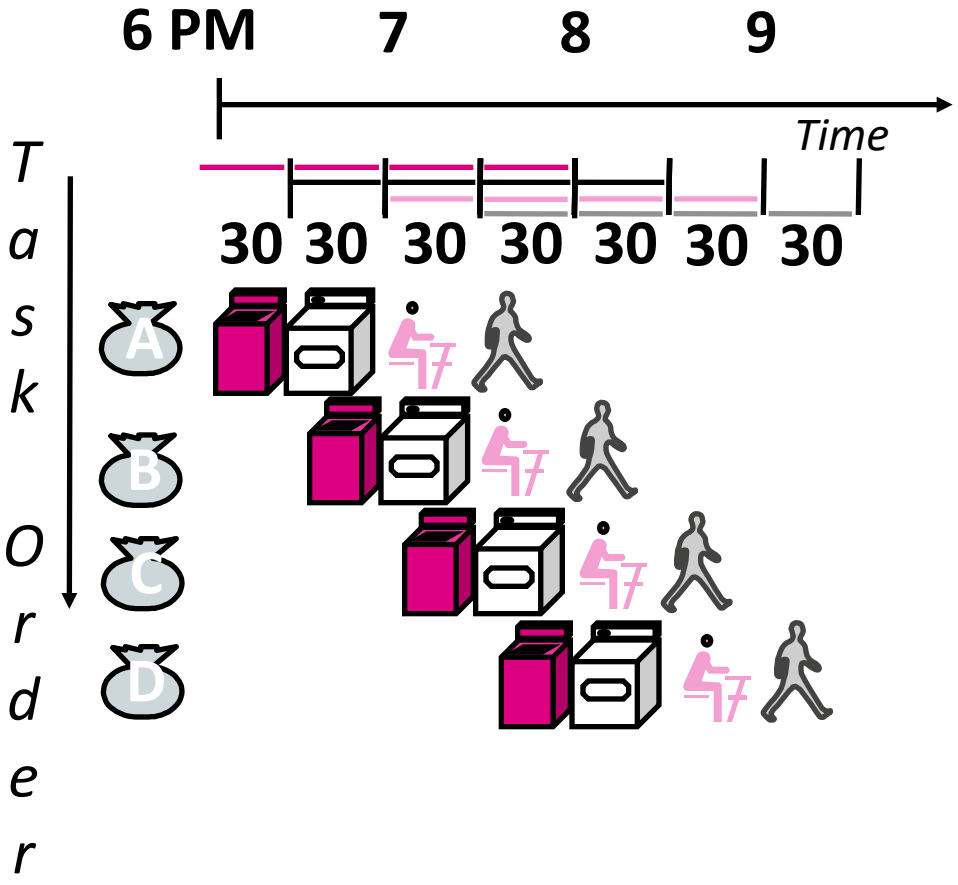
- *Speedup*: factor multiplicativo de aceleração

Lições sobre execução em *Pipelining* (1/2)



- O *Pipelining* não melhora a latência inerente a cada tarefa, aquilo que faz é melhorar o *throughput* na execução de um número de tarefas (*workload*), que podem ser total ou parcialmente paralelizáveis.
- A ideia base é executar múltiplas tarefas simultaneamente usando diferentes recursos físicos.
- O tempo necessário para "encher" e "limpar" o pipeline reduz o *speedup*:
  - 2.3X (8/3.5) versus. 4X (8/2)
- Potencial *speedup* = Número total de etapas/etapas no pipe (16/4=4)

Lições sobre execução com *Pipeline* (2/2)



- Imagine que introduzimos novas máquinas que reduzem os tempos de lavagem e secagem para 20 minutos. Será que isto vai melhorar o desempenho global?
- Não! O *pipeline* é limitado pela duração da etapa mais lenta.
- Desequilíbrios na duração dos estágios do *pipeline* implicam uma redução de *speedup*.

Estágios de *Pipeline* no MIPS

1) IFtch: Instruction Fetch, Incrementa PC

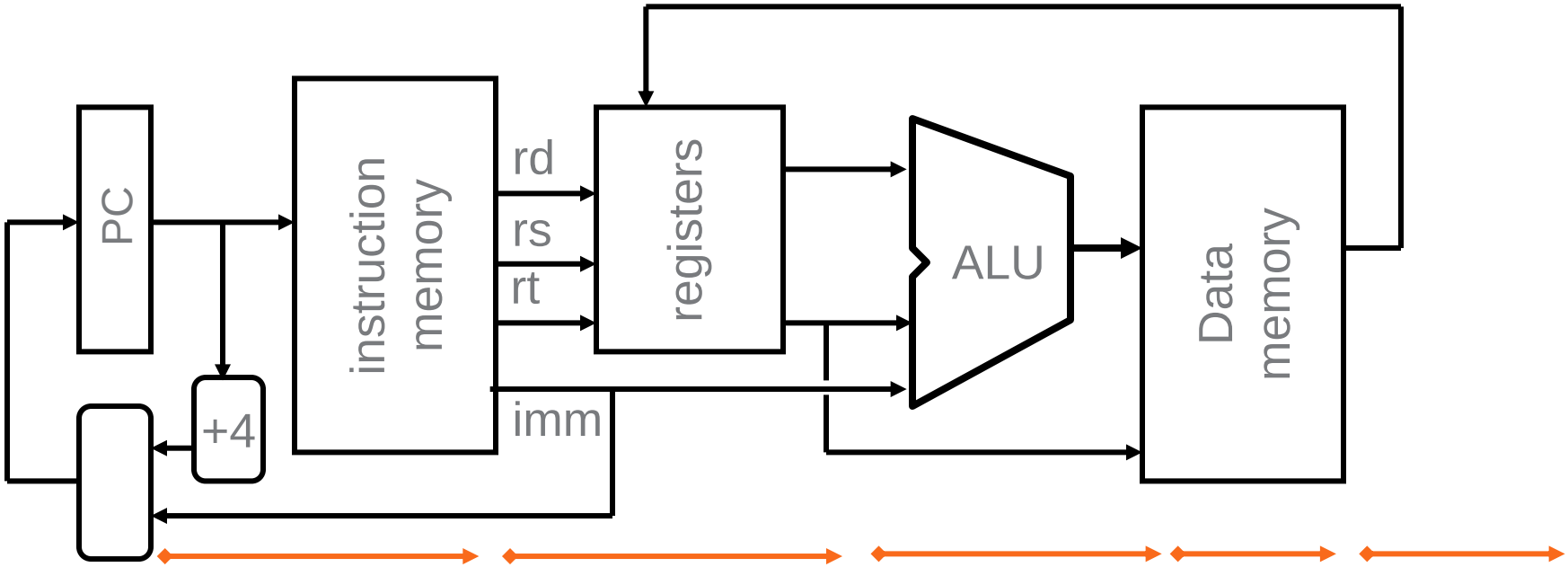
2) Dcd: Instruction Decode, Lê Registos

3) Exec:
   Mem-ref: Calcula endereços
   Arith-log: Executa a operação

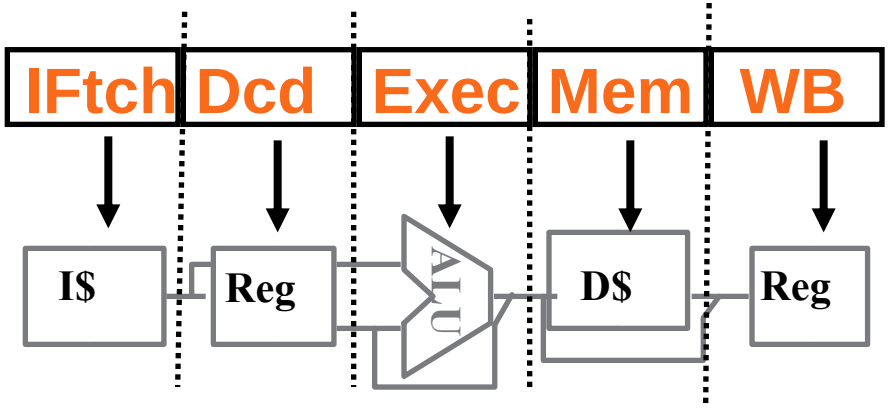
4) Mem:
   Load: Leitura de dados da memória
   Store: Escrita de dados para a memória

5) WB: Write Data Back to Register

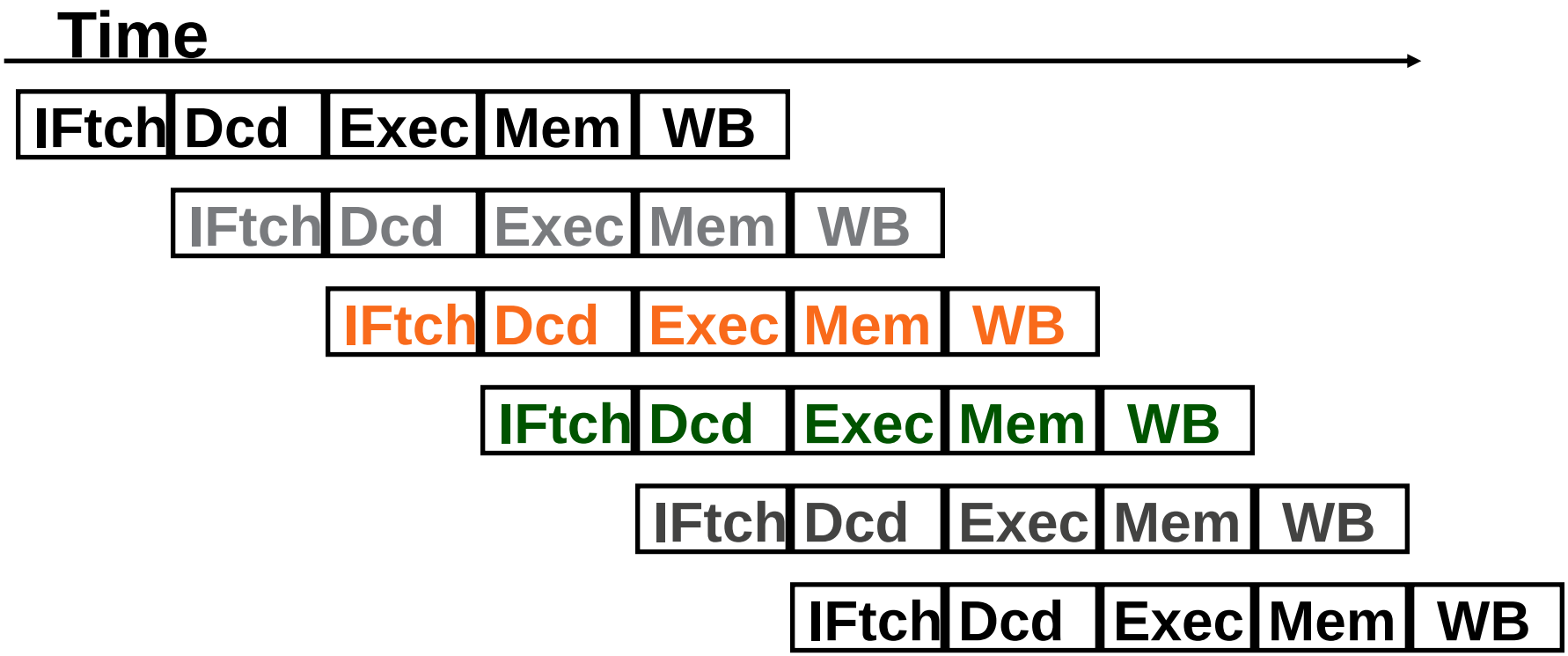
Revisão: Datapath para o MIPS



Usamos as figuras do *datapath* para representar o pipeline



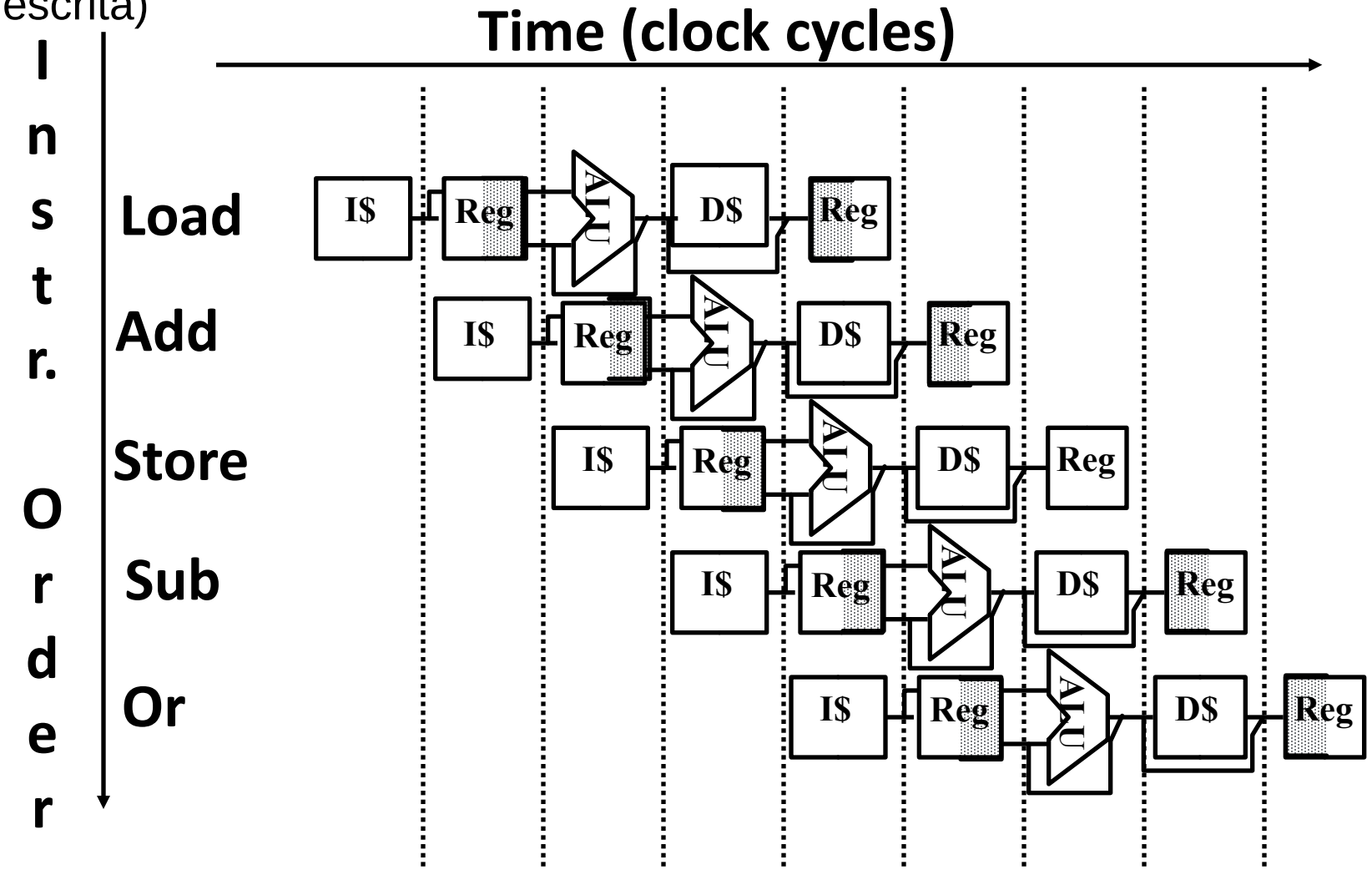
Representação da Execução em *Pipeline*



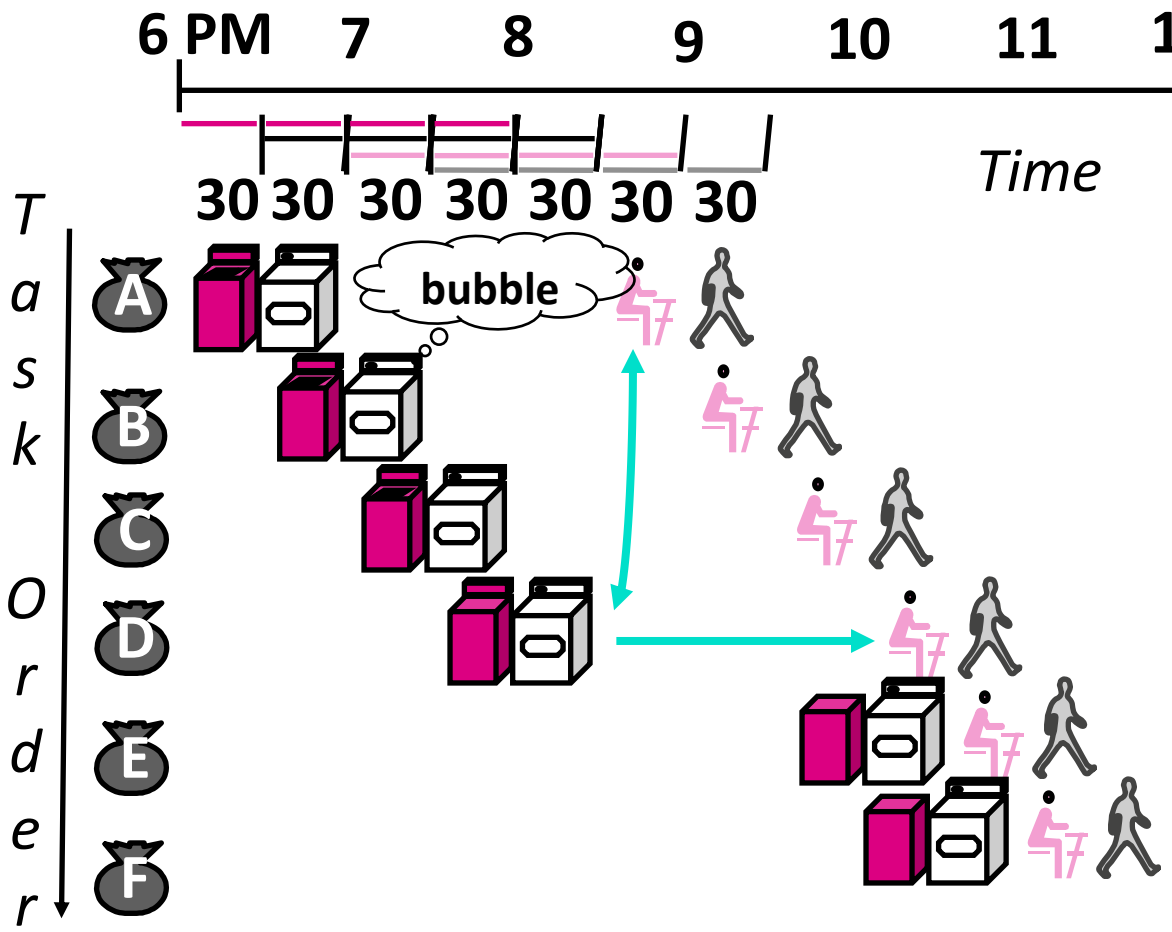
Cada instrução tem de passar pelo mesmo número de etapas, designadas por "estágios" do *pipeline*. Já vimos que algumas das instruções ficam inactivas em alguns dos estágios.

Representação Gráfica do *Pipeline*

(Nos Registos, a sombra do lado direito significa leitura, e do lado esquerdo escrita)



Conflitos no Pipeline (Pipeline Hazards)



- Suponha que o par de uma peúga no saco A seguiu por engano no saco D.
- A depende de D; isto causa um *stall* (paragem) no estágio de "dobragem"

Problemas no Pipeline 1/2

- Limitações da técnica de *pipelining*: Podem ocorrer conflitos que bloqueiem a instrução seguinte, evitando que ela seja executada no ciclo de relógio previsto:

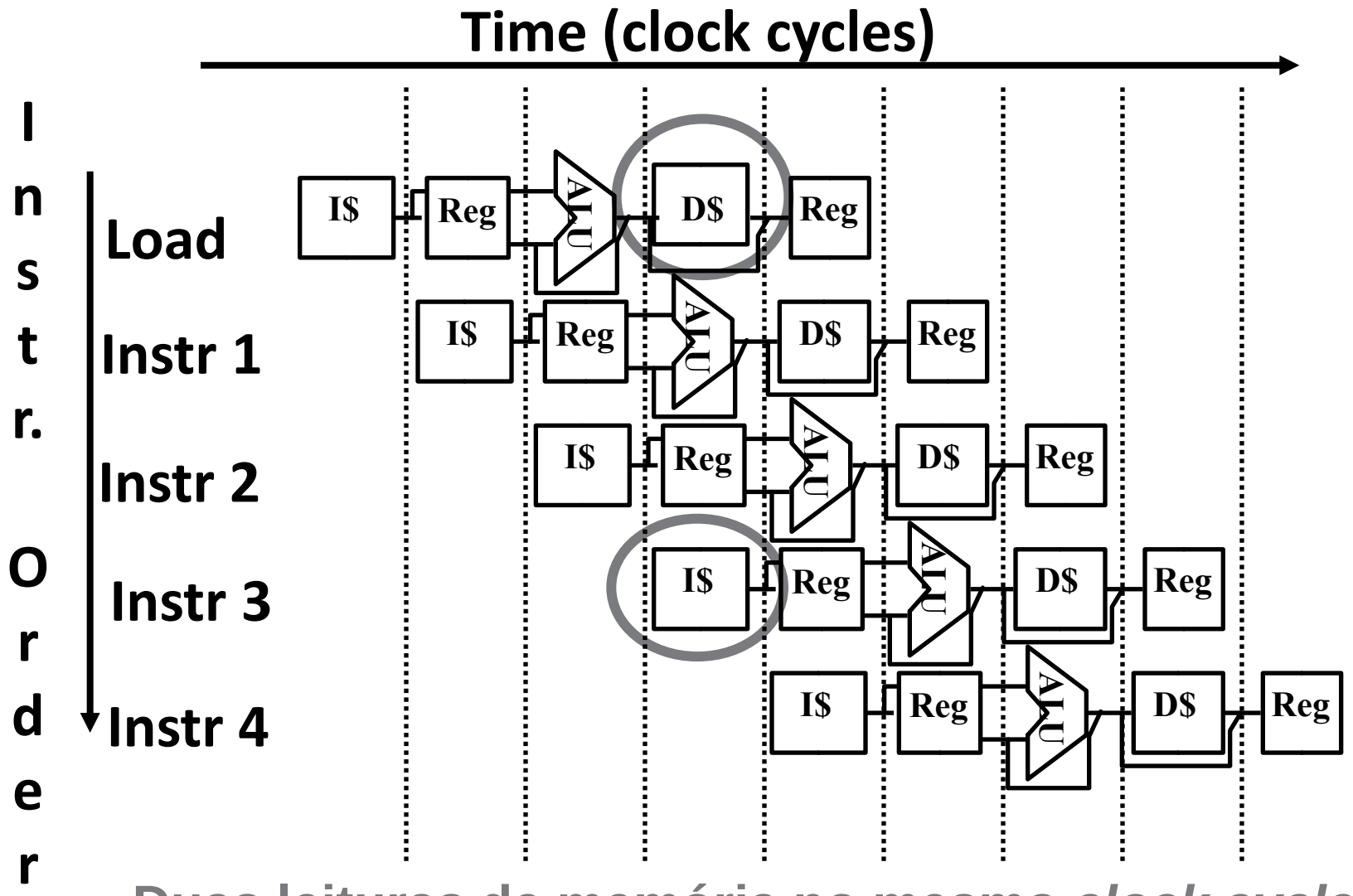
  - Conflitos Estruturais (*structural hazards*): O HW físico não permite suportar determinadas combinações de instruções (e.g. uma única pessoa não pode dobrar e arrumar a roupa simultaneamente)

  - Conflitos de Controlo (*control hazards*): Quando aparecem saltos potenciais no fluxo de execução (instruções de *branch* e *jump*) existe incerteza quanto às instruções que se seguem. Isto causa paragens e poderá levar a uma limpeza do pipeline e retrocesso na execução ("*flush*").

Problemas no Pipeline 2/2

- Limitações da técnica de *pipelining*: Podem ocorrer conflitos que bloqueiem a instrução seguinte, evitando que ela seja executada no ciclo de relógio previsto:
  - Conflitos de Dados (*data hazards*): Instruções que dependem do resultado de outras instruções que ainda estão no *pipeline* (o caso do par de peúgas)
- Qualquer um destes conflitos conduz a situações de paragem ("*stalls*"), criando "bolhas" no *pipeline*.

Conflito Estrutural #1: Acesso a Memória (1/2)

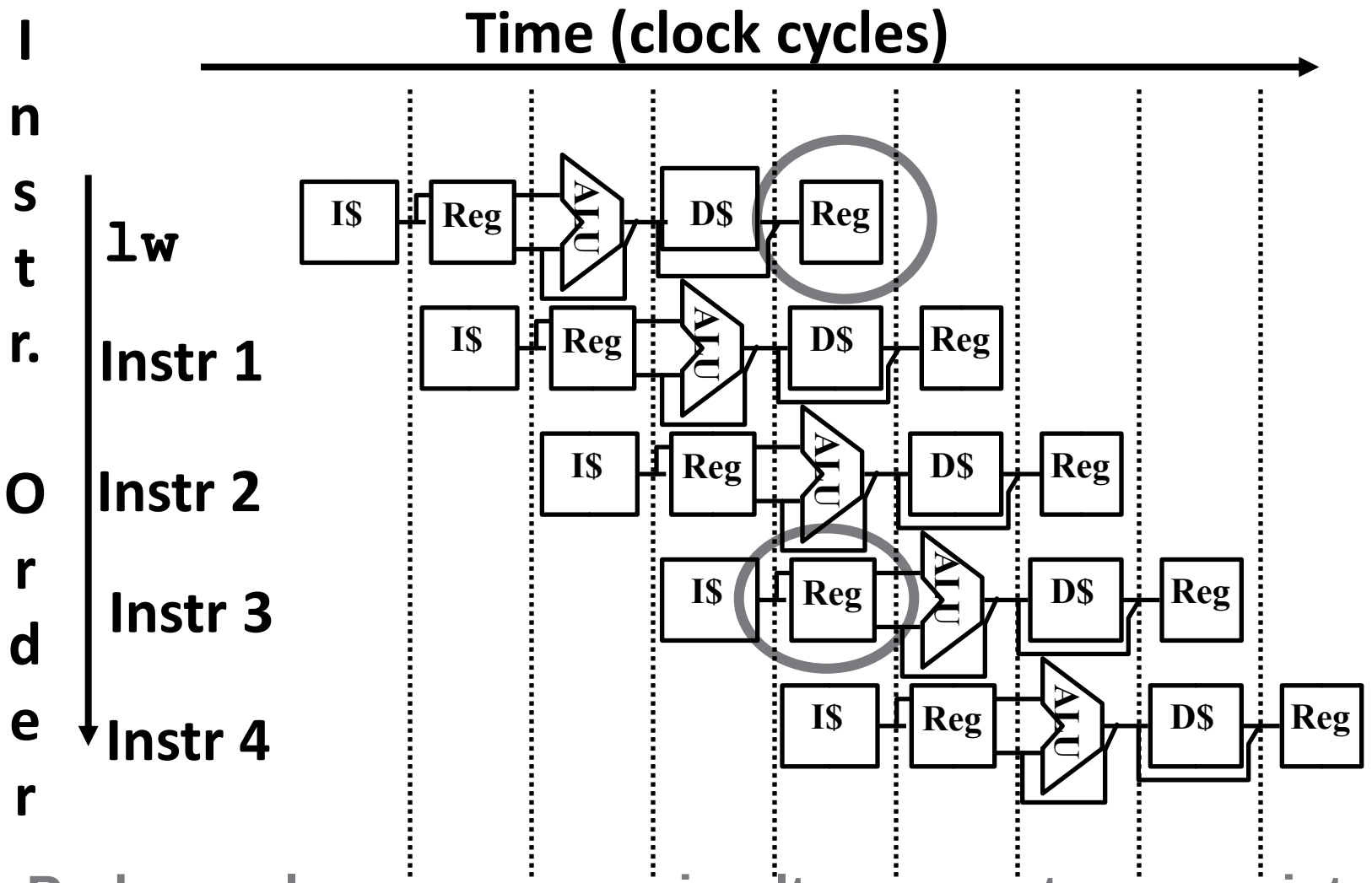


Duas leituras de memória no mesmo *clock cycle*

Conflito Estrutural #1: Acesso a Memória (2/2)

- Solução:
  - Replicar as memórias: Ineficiente e Não Exequível (recorde a hierarquia de memória)
  - Simular duas memórias usando dois níveis de Cache de Nível 1 (relembre que uma cache é uma pequena cópia temporária da memória com informação usada recentemente)
  - Neste caso teremos uma *Instruction Cache* e uma *Data Cache*, sendo o HW de controlo mais complexo no caso de haver dois "*cache misses*" simultâneos.

Conflito Estrutural #2: Registos (1/2)



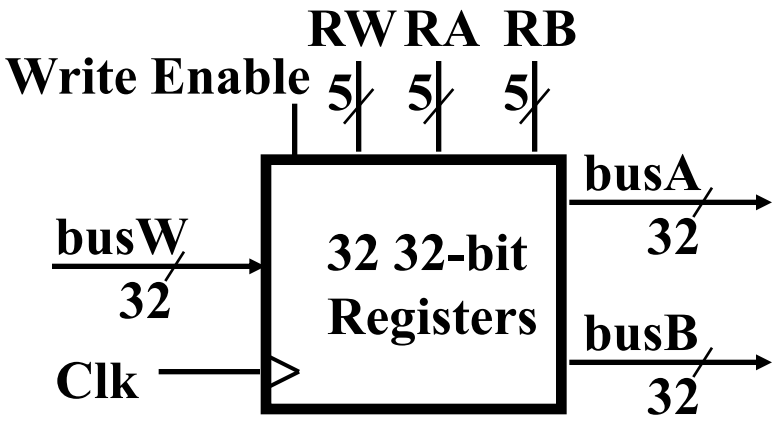
Podemos ler e escrever simultaneamente em registos?

Conflito Estrutural #2: Registos (2/2)

- Existem duas soluções diferentes para este problema:
  - 1) O acesso à memória de registos é muito rápido: demora menos de metade do tempo da etapa ALU. Assim,
    - Podemos escrever no ***RegFile*** durante a primeira metade do ciclo de relógio e Ler os registos na segunda metade do ciclo
    - *Será que faria sentido fazer primeiro a leitura e depois a escrita?*
  - 2) Implementar o RegFile em HW definindo portos independentes para leitura e escrita.
- Resultado: É possível escrever e ler os registos no mesmo ciclo de relógio

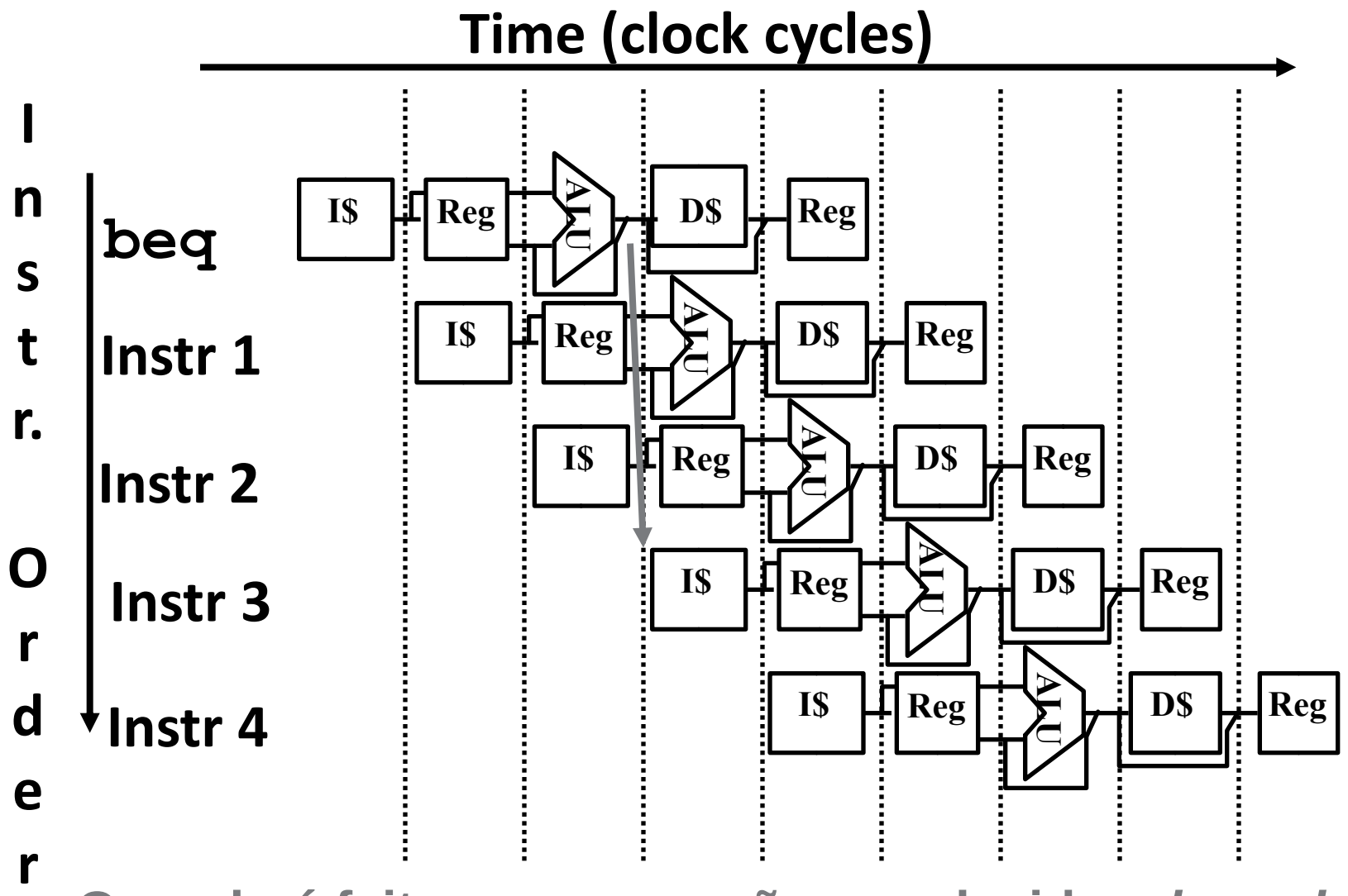
Revisão: *Register File*

- Consiste em 32 registos:
  - 2 *buses* de saída de 32-bit (busA and busB)
  - 1 *bus* de entrada de 32-bit: busW
- O Registo é seleccionado por:
  - RA (número) selecciona o registo para busA
  - RB (número) selecciona o registo para busB
  - RW (número) selecciona o registo a ser escrito via busW quando Write Enable é 1



- Repare que é possível fazer leitura e escrita simultaneamente
- *Clock input* (clk)
  - O clk input só é importante para operações de escrita
  - Na leitura o "*register file*" comporta-se como lógica combinacional:
    - RA ou RB válido => busA ou busB válido depois de "*access time*"

Conflitos de Controlo: *Branching* (1/7)



Quando é feita a comparação que decide o *branch*?

Conflitos de Controlo: *Branching* (2/7)

- Até aqui assumimos que a decisão de salto é tomada quando é feita a comparação no estágio ALU.
  - Assim, existem sempre duas instruções depois do *branch* que entram no pipeline. Se houver salto, essas instruções não são para executar, perdendo-se dois ciclos.

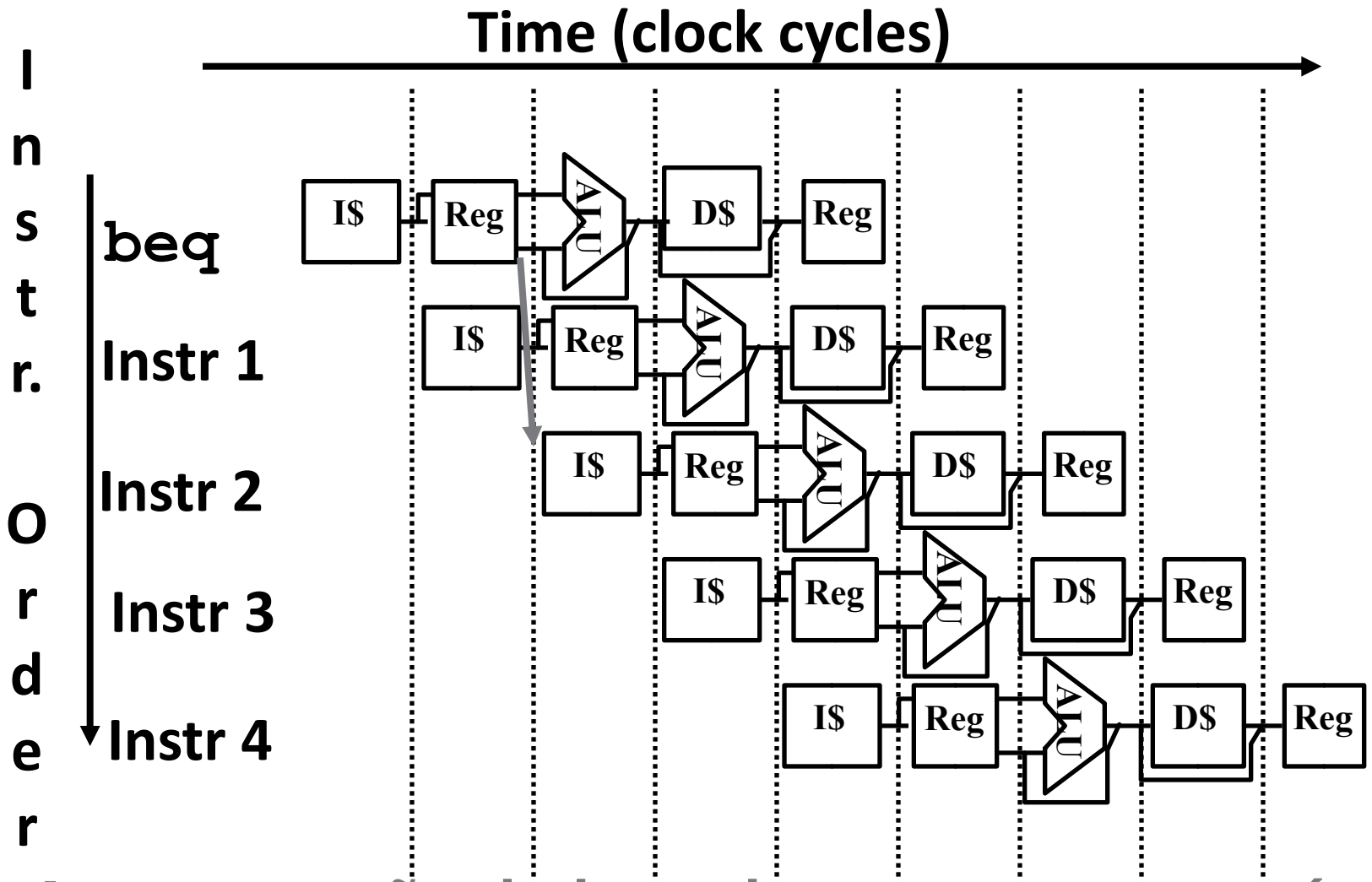
- Idealmente um *branch* deve funcionar da seguinte forma:
  - Se o salto não ocorrer, a execução deve continuar de forma normal sem perda de tempo
  - Se o salto ocorrer, as instruções a seguir ao *branch* não devem ser executados, passando a execução para o ponto indicado pelo "*label*"

Conflitos de Controlo: *Branching* (3/7)

- Solução 1 : Paragem no pipeline
  - Inserir instruções "no-op" a seguir ao *branch*, ou não fazer *fetch* de instruções até a decisão de salto ser tomada (*stall* durante 2 ciclos de relógio).
  - Desvantagem: as instruções de *branch* passam a demorar 3 ciclos de relógio em vez de um único ciclo

- Otimização #1:Implementar um comparador para "branches no estágio 2
  - Assim que uma instrução é descodificada, verifica-se se o *opcode* corresponde a um *branch*. Neste caso a decisão é imediatamente tomada e o PC é ajustado de forma adequada.
  - Vantagem: Como o *branch* é completado no estágio 2, só a instrução a seguir é que entra no pipeline, bastando um único "nop"
  - Nota: A instrução de "*branch*" está inactiva nos estágios 3, 4 e 5.

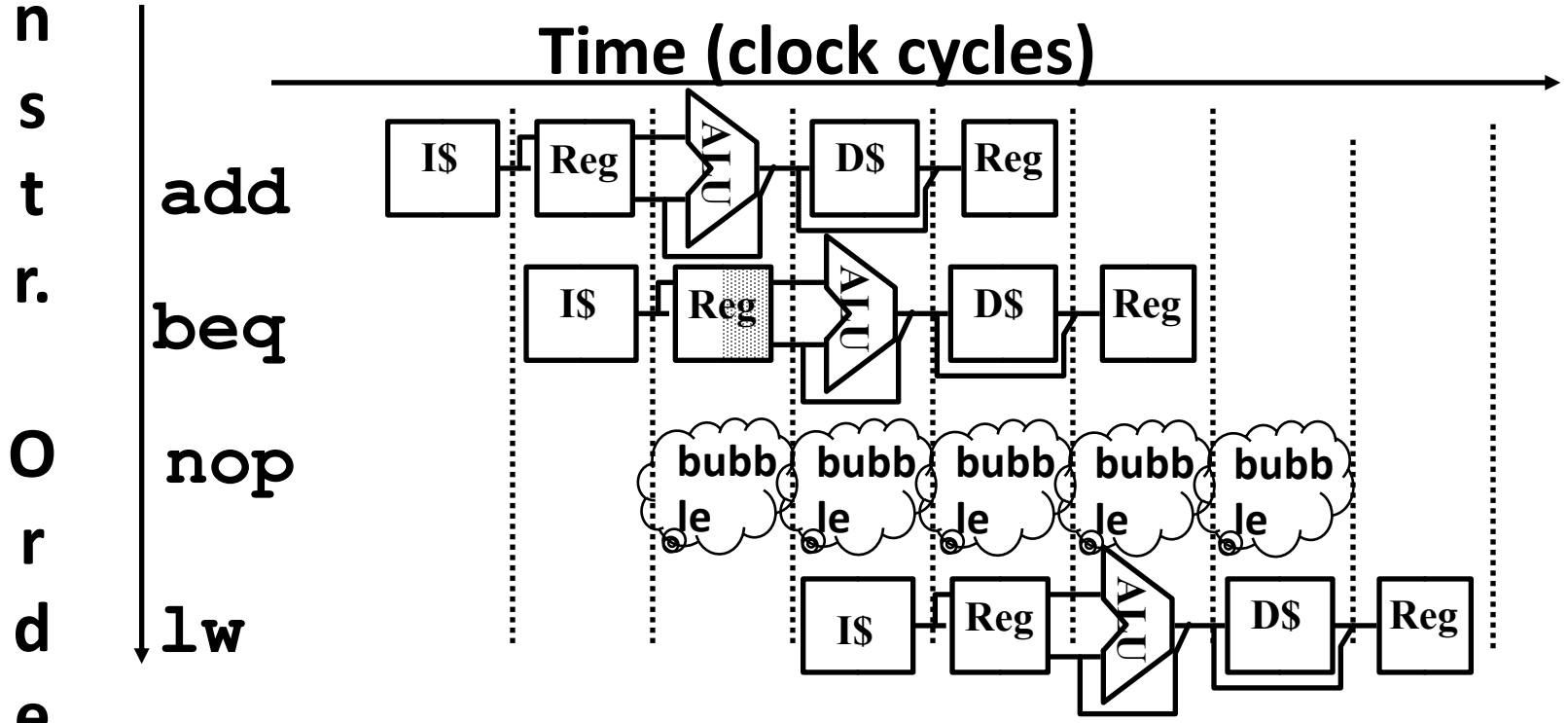
Conflitos de Controlo: *Branching* (4/7)



A comparação do branch passa para o estágio 2.

Conflitos de Controlo: Branching (5/7)

- O utilizador/programador insere uma instrução “no-op”



- Impacto: 2 ciclos de relógio / instrução de *branch* => ainda é lento

Conflitos de Controlo: *Branching* (6/7)

- Optimização #2: Redefinir o comportamento do branch
  - Definição usada até agora: se o salto acontecer, nenhuma das instruções a seguir ao "branch" deve ser acidentalmente executada.
  - Nova definição: independentemente do salto acontecer, ou não, a instrução a seguir ao branch deve ser sempre executada (chama-se a isto *branch-delay slot*)

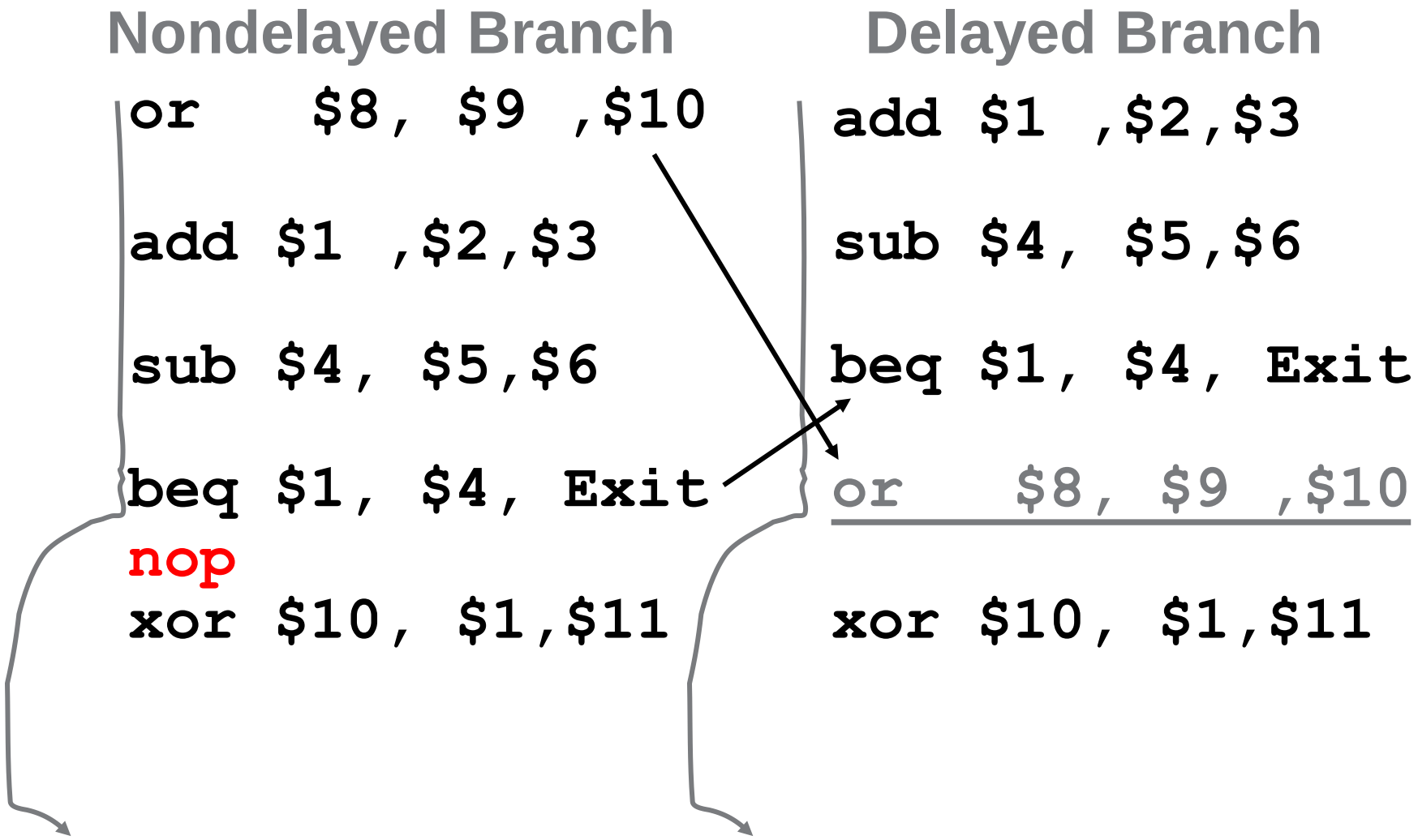
- O termo "*Delayed Branch*" significa que a instrução a seguir ao *branch* é sempre executada

- Esta optimização é utilizada no MIPS

Conflitos de Controlo: *Branching* (7/7)

- Como funciona o *Branch-Delay Slot*?
  - *Worst-Case Scenario*: colocamos uma instrução "*no-op*" no *branch-delay slot*
  - Solução mais optimizada: podemos colocar no *branch-delay slot*, uma instrução originalmente antes do "*branch*", que pode ser colocada depois sem afectar o correcto fluxo de execução.
    - A reordenação das instruções é muitas vezes utilizada para acelerar os programas
    - O compilador tem de ser muito "esperto" para fazer esta reordenação de forma automática
    - Em cerca de 50% dos casos é possível encontrar uma instrução para preencher o "*slot delay*", evitando-se completamente o conflito de controlo
    - Repare que os *jumps* têm o mesmo problema dos *branches* …

Exemplo: Nondelayed vs. Delayed Branch

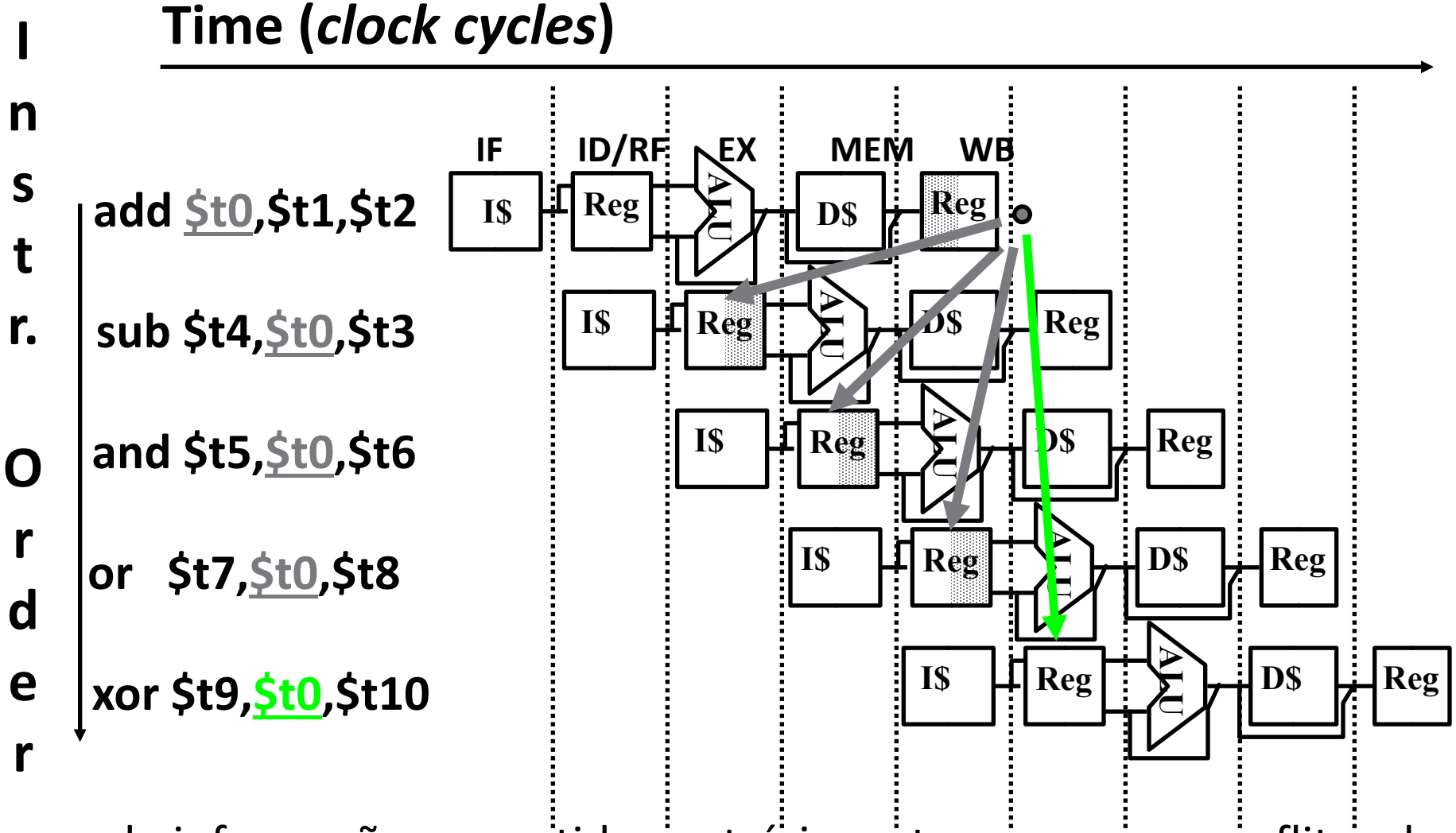


Conflitos de Dados (1/2)

- Considere a seguinte sequência de instruções

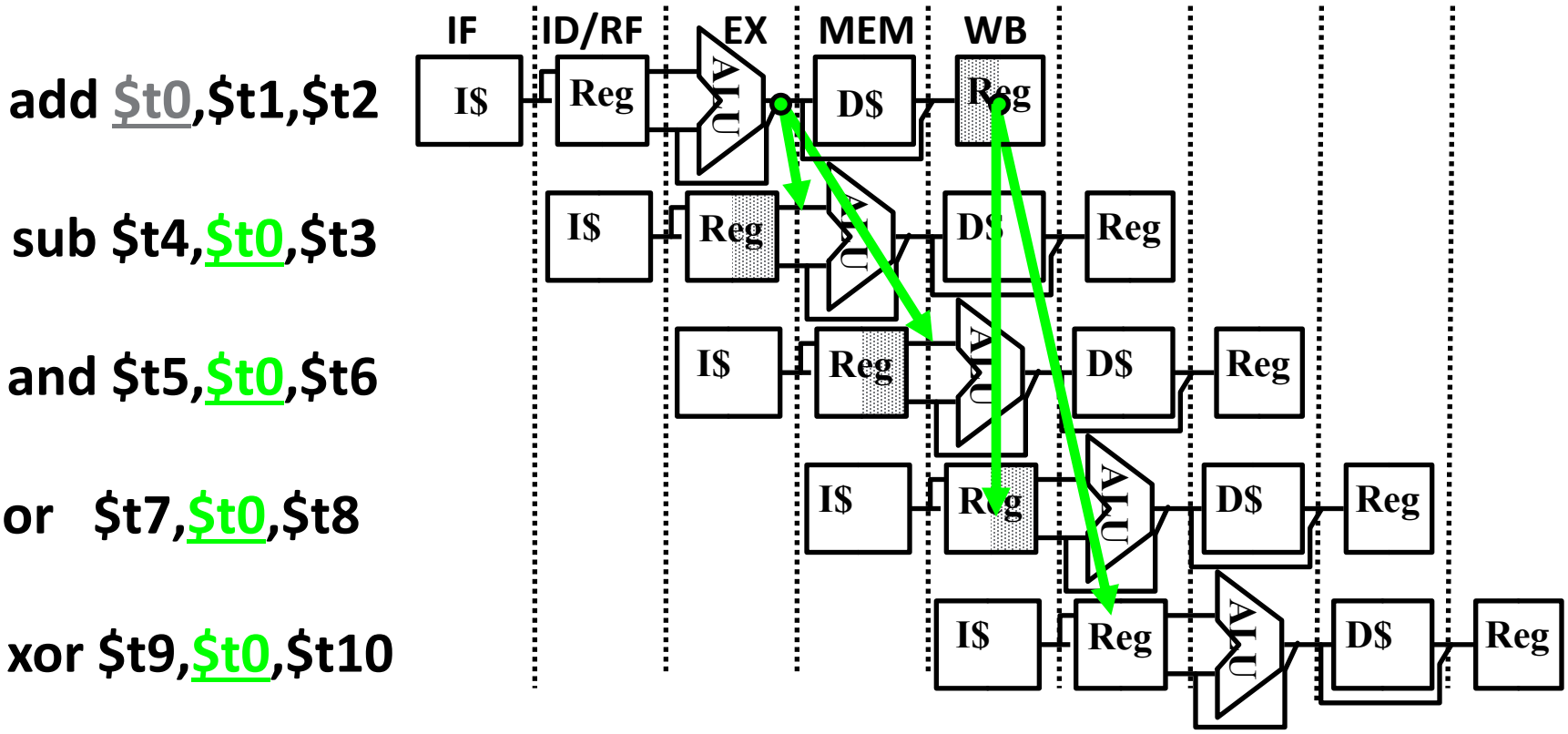
```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

Conflitos de Dados (2/2)



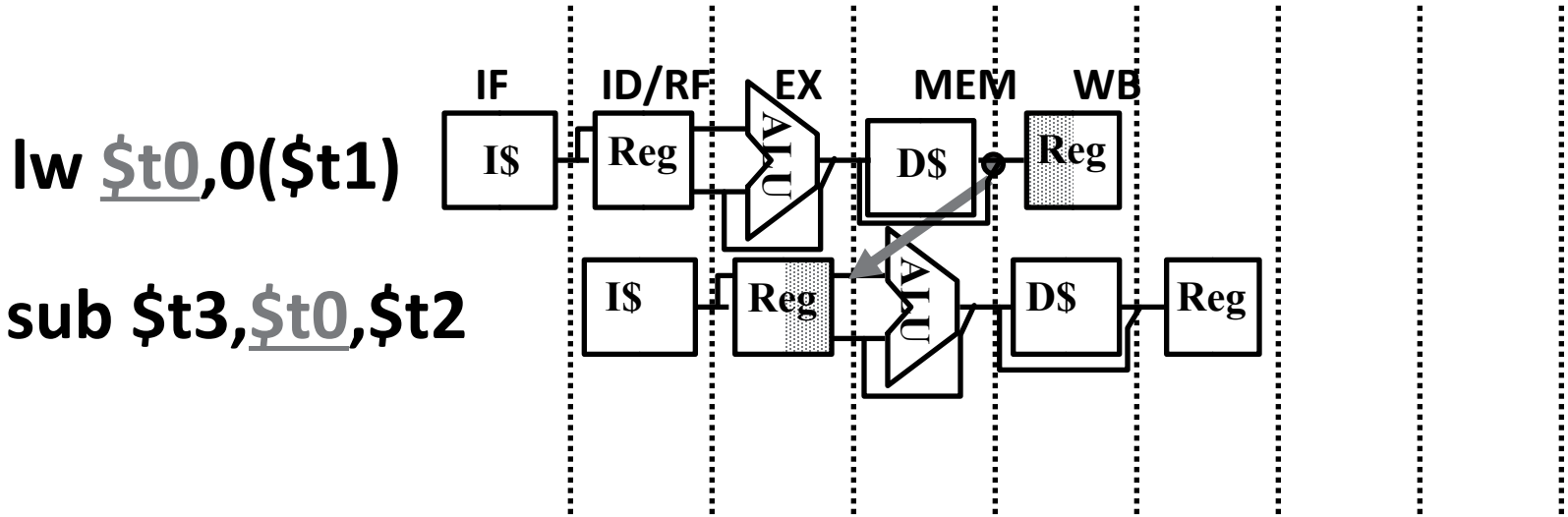
Fluxos de informação no sentido contrário ao tempo geram conflitos de dados

Solução para Conflitos de Dados: Forwarding



- Repare que o valor a ser escrito em $t0 está disponível à saída da ALU
- Podemos fazer *FORWARD* de um estágio para outro de forma a evitar conflitos
- Repare que o conflito no "or" é evitado pelo HW do *RegFile* (escrita antes da leitura)

Conflitos de Dados: *Loads* (1/4)



- Neste caso o valor para o "*sub*" não é conhecido antes de ser necessário
- A técnica de "*forwarding*" não resolve a situação
- É necessário colocar um "*stall*" depois do *load*, e depois fazer *forwarding* (mais *hardware* específico para realizar esta operação)

Conflitos de Dados: *Loads* (2/4)

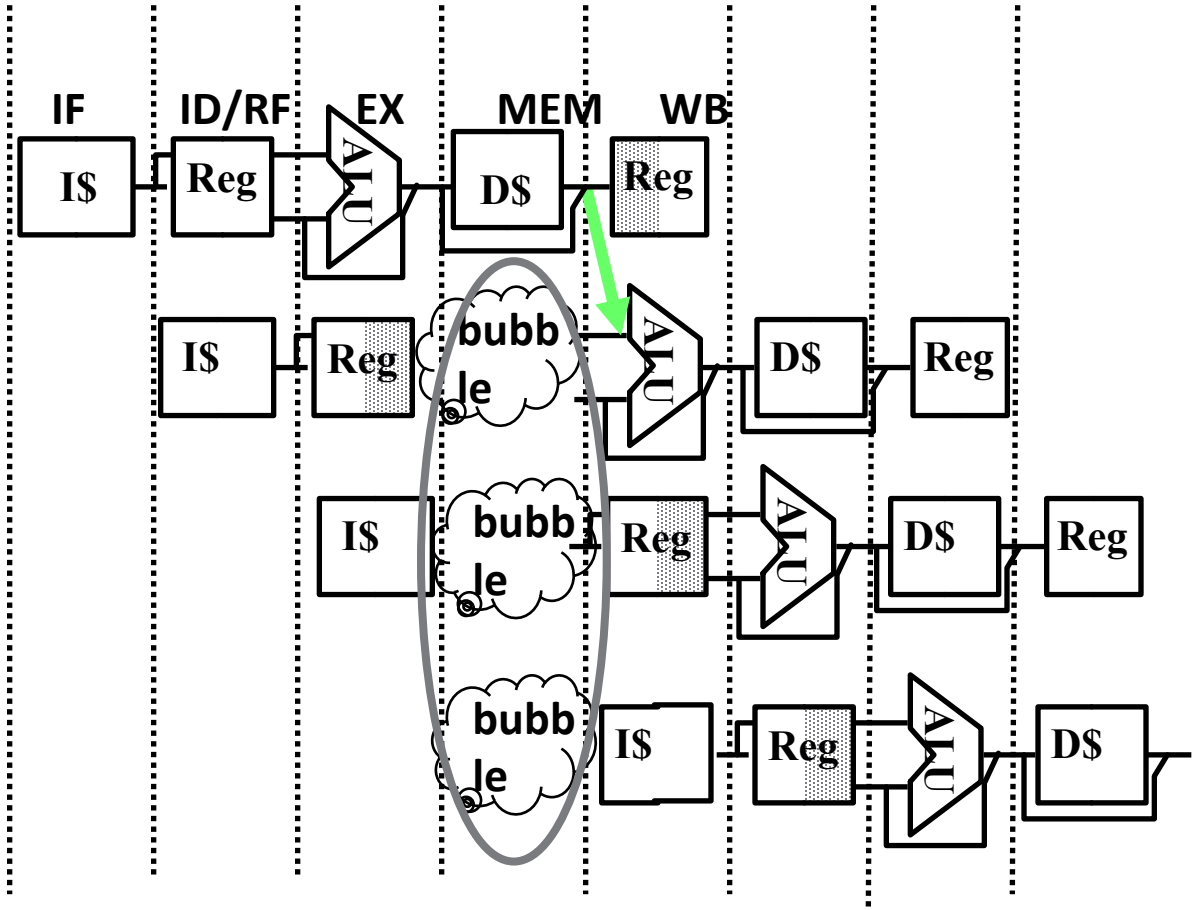
- O próprio HW faz "stall" do pipeline: chama-se a isto *"interlock"*

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



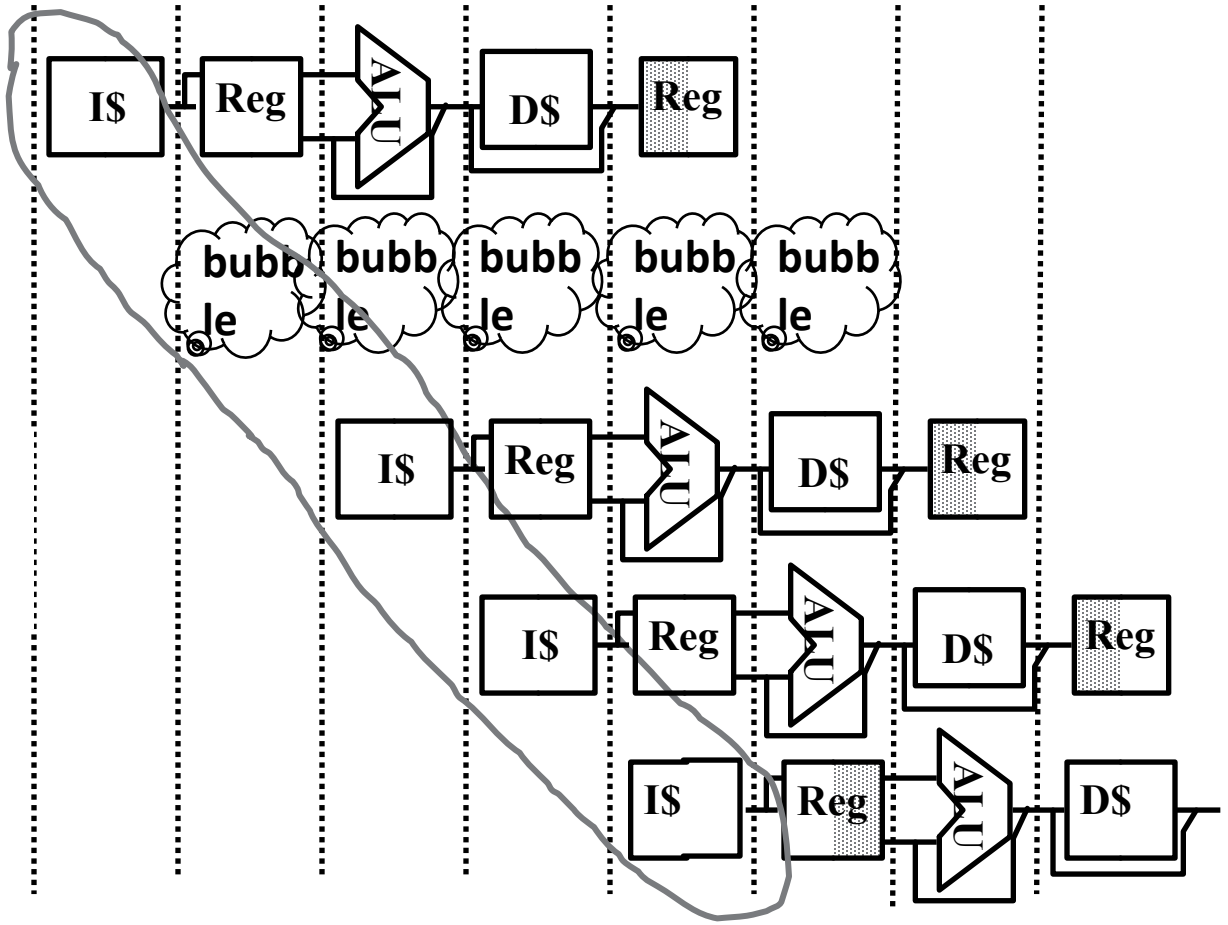
Conflitos de Dados : *Loads* (3/4)

- A *slot* depois do *load* é chamada "*load delay slot*"
- Se a instrução utilizar o resultado do *load*, então o hardware faz um *interlock* para fazer parar o pipeline durante um ciclo de relógio (*stall*).
- Repare que o HW consegue saber se deve ou não colocar o "*stall*". Já identificou o *load*, e a instrução também já foi descodificada sendo os operandos conhecidos.
- O compilador pode fazer um reordenamento de forma a que a instrução na "*load delay slot*" não dependa do *load*. Neste caso evita-se a bolha no pipeline.
- Deixar o HW fazer o "*interlock*" é equivalente a colocar uma instrução "*no-op*" a seguir ao *load*. (excepto que esta última solução implica mais espaço para código)

Conflitos de Dados: *Loads* (4/4)

- *Stall* é equivalente a *nop*:

```
lw $t0, 0($t1)
nop
sub $t3,$t0,$t2
and $t5,$t0,$t4
or  $t7,$t0,$t6
```



Curiosidade Histórica

- A primeira versão do MIPS caracterizava-se por não possuir nenhum mecanismo de "*interlock*" por hardware. A resolução de conflitos tinha que ser feita ao nível do compilador

  Microprocessor without
  Interlocked
  Pipeline
  Stages

- E não a interpretação do acrónimo "*Millions of Instructions Per Second*" que depois muita gente fez.

Sumário: Pipelining (1/3)

- *Pipelining* em circunstâncias ideais
  - Cada estágio executa uma parte da instrução num ciclo de relógio
  - Assim o processador termina a execução de uma instrução por cada ciclo de relógio.
  - Em média a execução torna-se muito mais rápida.

- Porque é que isto funciona?
  - Em geral, a semelhança e uniformidade das instruções permitem-nos usar os mesmos estágios para executar cada uma delas (filosofia dos processadores RISC).
  - A divisão em estágios/etapas é equilibrada de forma a que cada um deles tenha aproximadamente a mesma duração: minimizar o desperdício de tempo.

- O *Pipelining* é uma GRANDE IDEIA, sendo muito utilizada

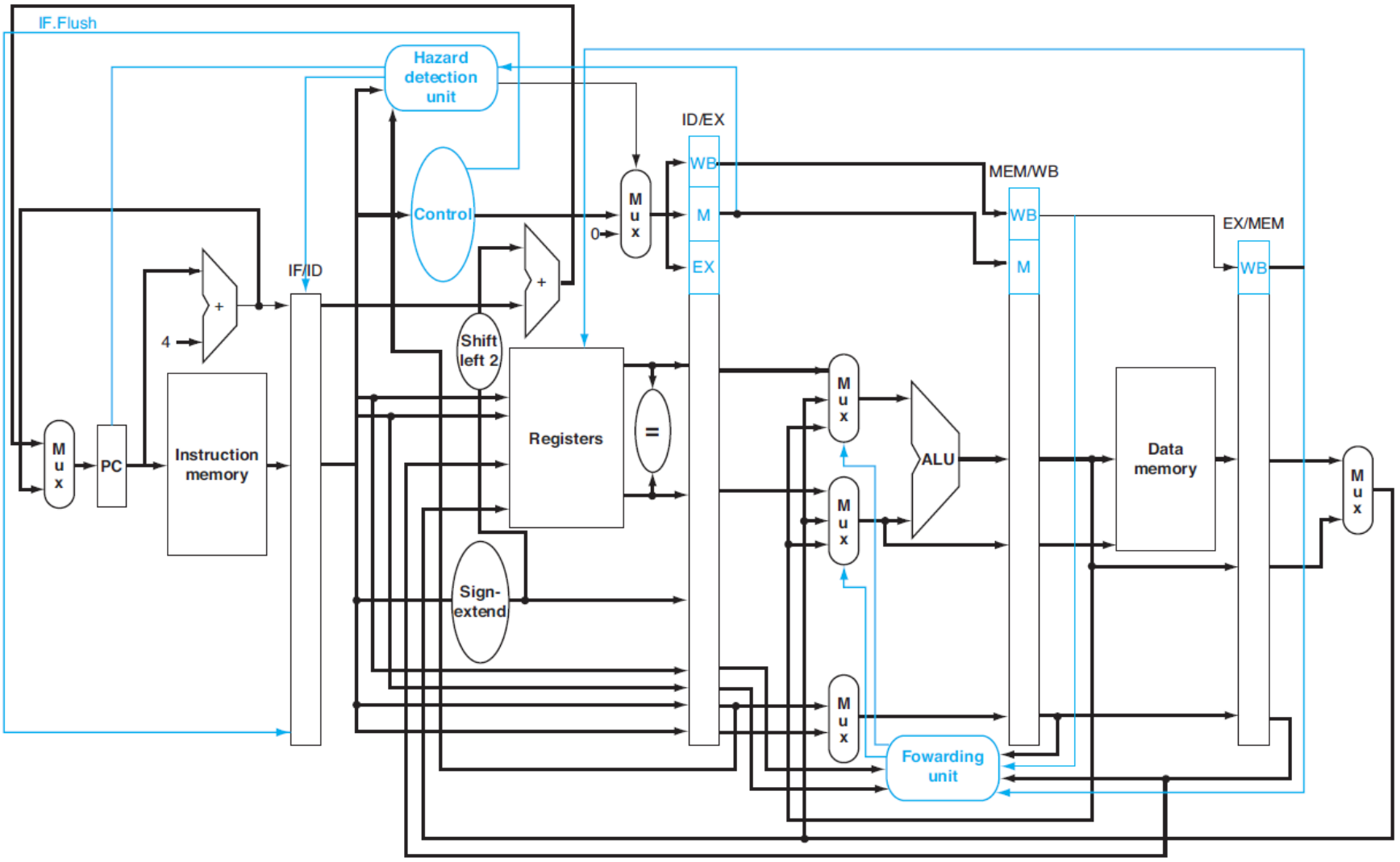
Sumário: *Pipelining* (2/3)

- Quais são os problemas e limitações inerentes a fazer *pipelining*?
  - Conflitos Estruturais: Trata-se de conflitos devidos a falta de recursos físicos. Imagine que só temos uma cache que é partilhada por dados e instruções? => A solução passa por ampliar os recursos de HW disponíveis
  - Conflitos de Controlo: Nas instruções de salto (*branches* e *jumps*) não sabemos qual é a instrução que se segue. =>Solução Possível: *Delayed branch*, ou seja re-ordenar as instruções para colocar uma instrução anterior ao *branch* na "*delay slot*" (se isto não for possível o compilador coloca um "*no-op*")

Sumário: *Pipelining* (3/3)

- Quais são os problemas e limitações inerentes a fazer *pipelining*?
  - Conflitos de Dados: Fluxo de informação no sentido contrário ao tempo / estágios do pipeline.
    - *Forwarding* evita muitos destes conflitos
    - *Load delay slot* / *interlock* é necessário porque *forwarding* não resolve

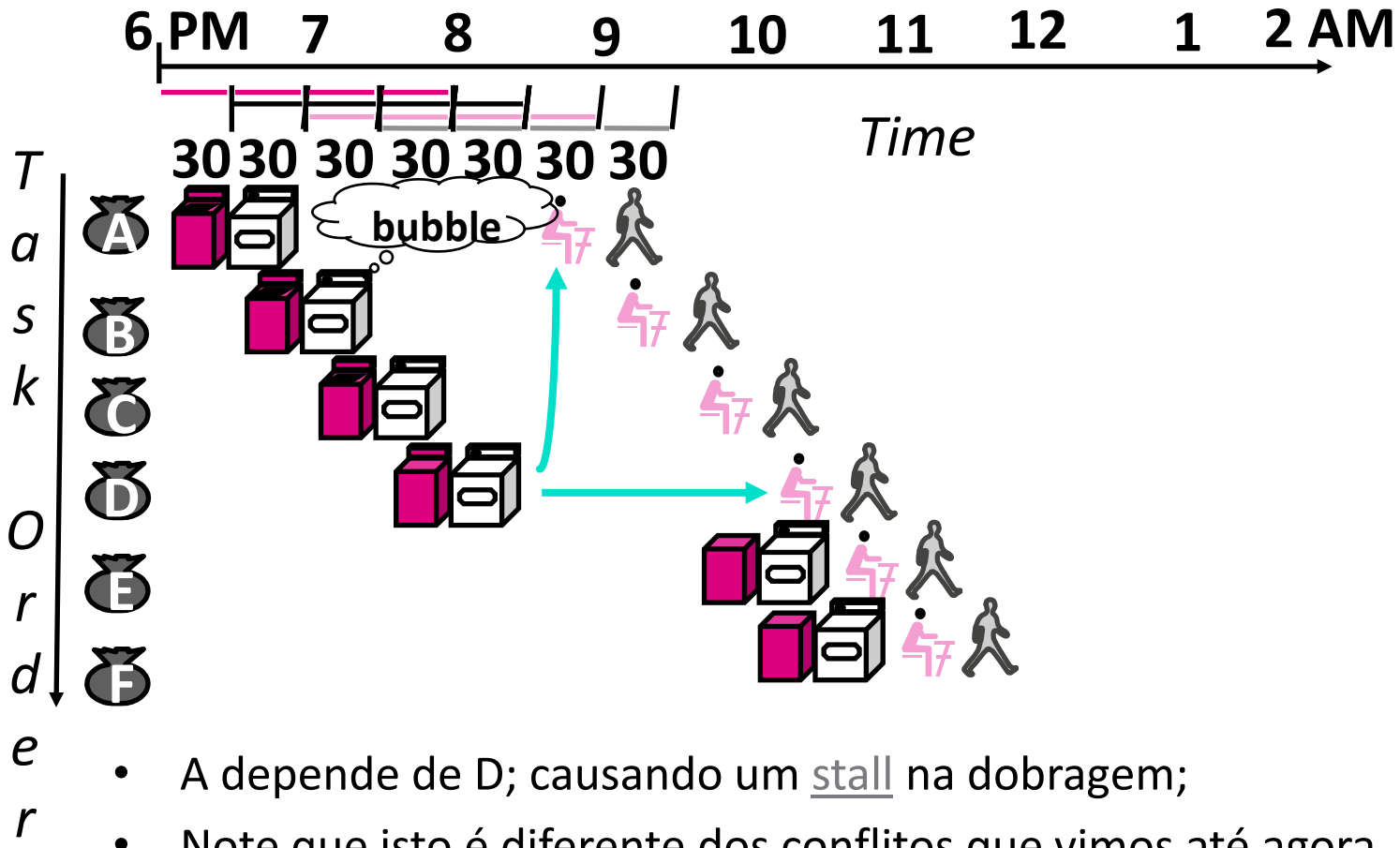
Diagrama de Blocos do *Datapath* com Pipeline



Mas a história não termina aqui ...

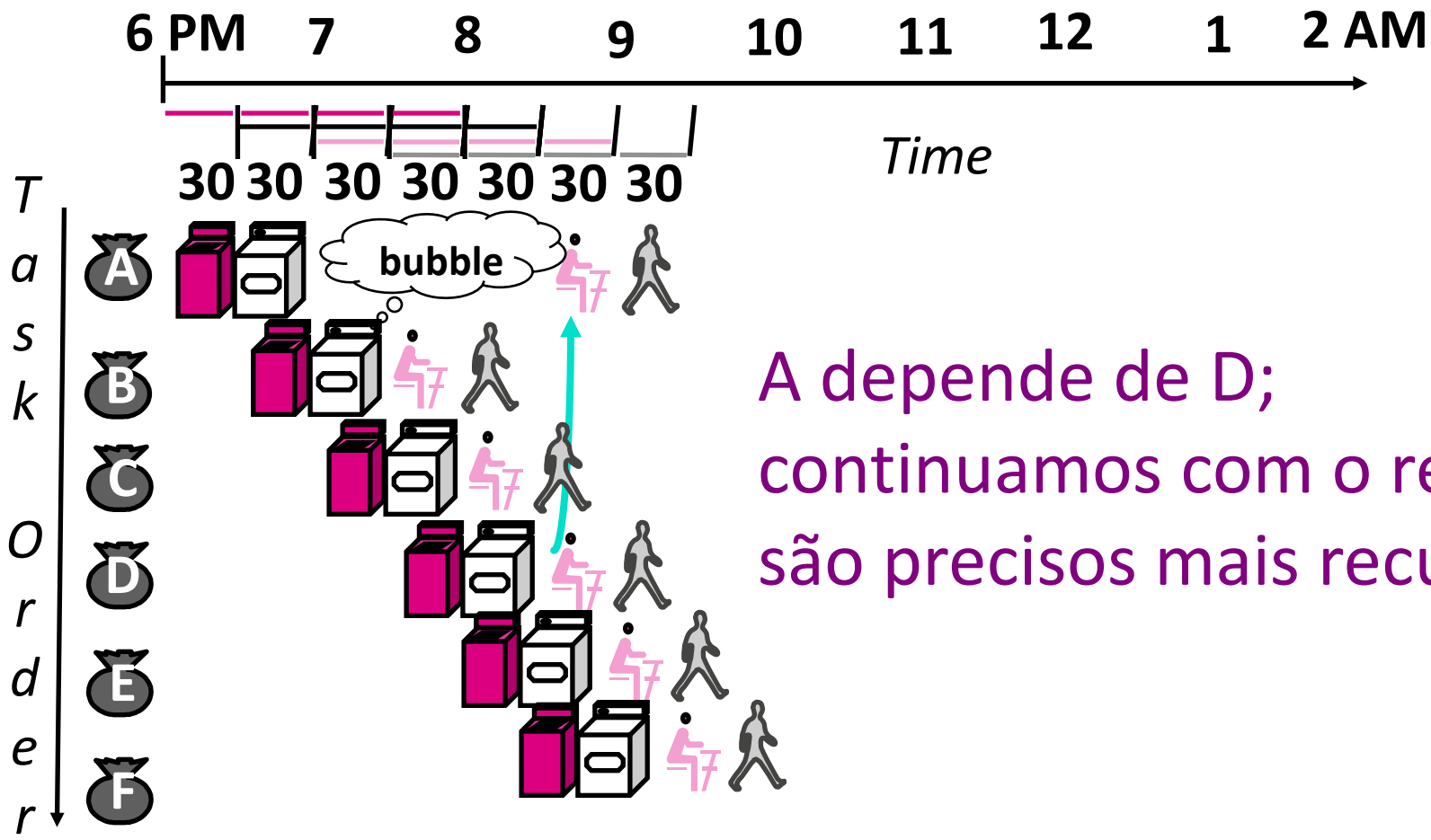
- Desempenhos mais agressivos com processadores superescalares:
  - Exemplo: Placas gráficas com vários pipelines em paralelo
- Execução fora de ordem
- Todos estes mecanismos exigem replicação de recursos de HW

Pipeline Hazard: O problema de juntar as peúgas



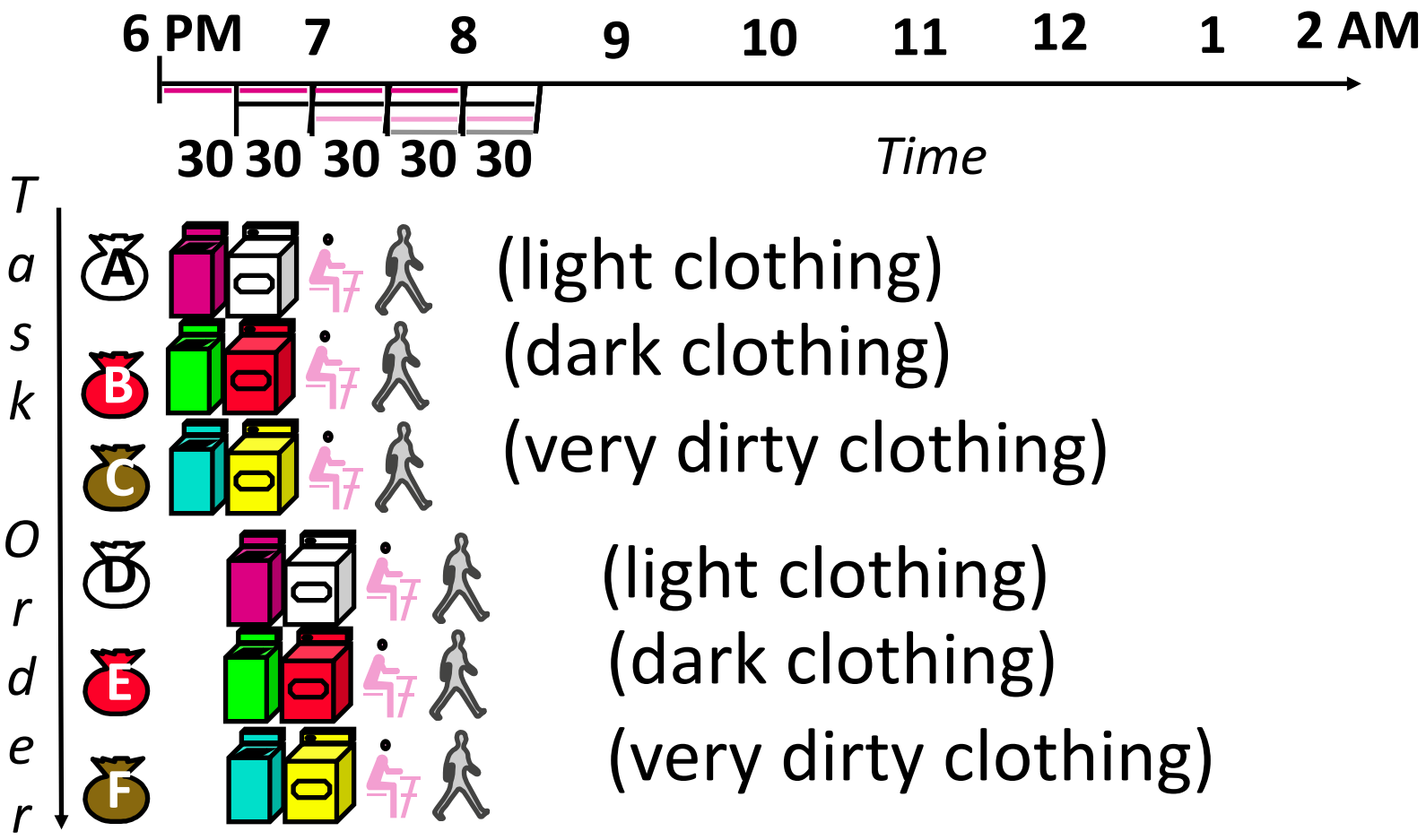
- A depende de D; causando um stall na dobragem;
- Note que isto é diferente dos conflitos que vimos até agora ... Nunca tivemos uma instrução a depender do resultado de outra instrução que vem a seguir
- Chama-se a isto execução fora de ordem

Execução Fora de Ordem: Não Espere!

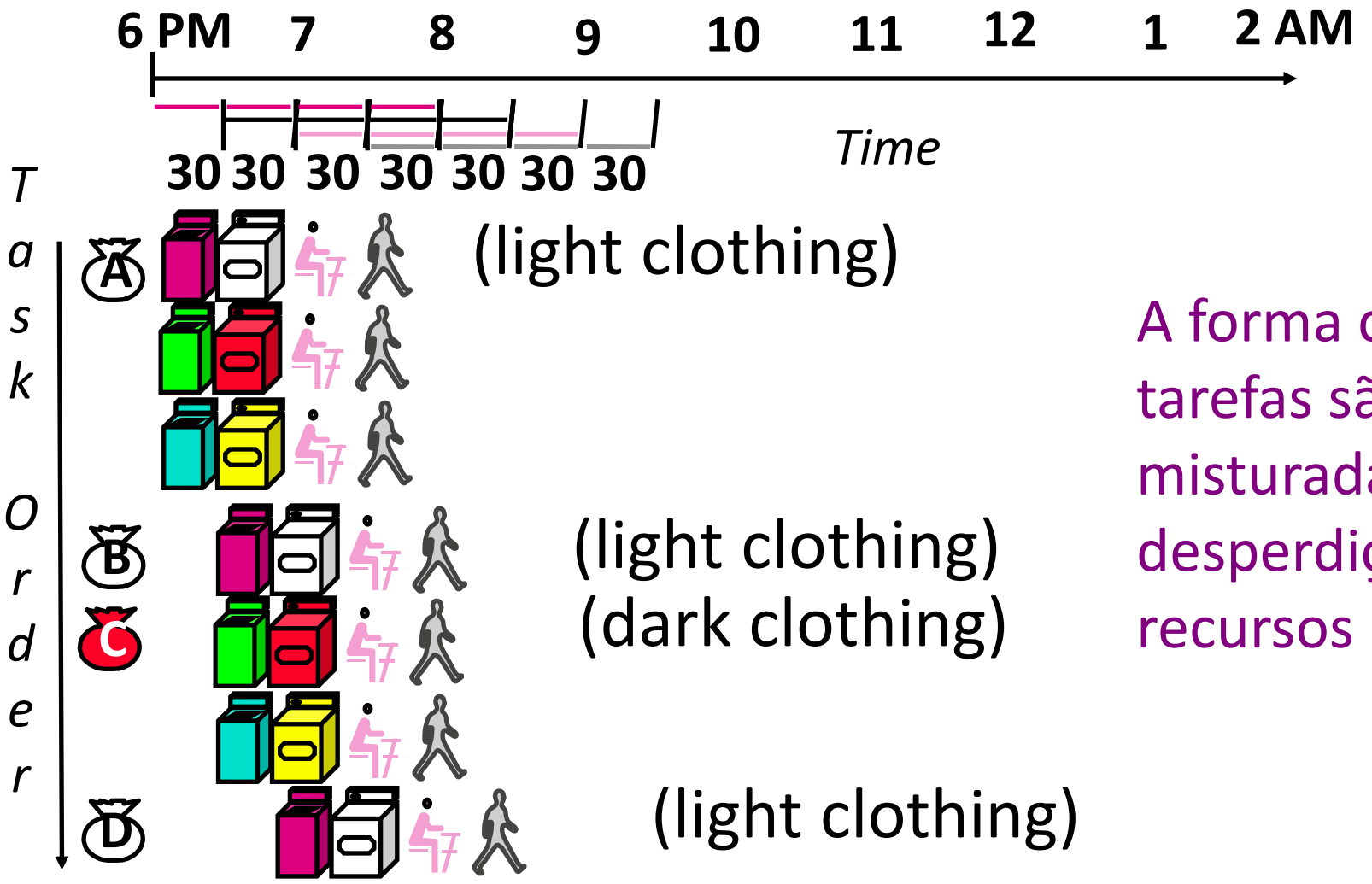


A depende de D;
continuamos com o resto;
são precisos mais recursos

Execução Superescalar : Estágios Paralelos



Execução Superescalar: Desperdício de recursos

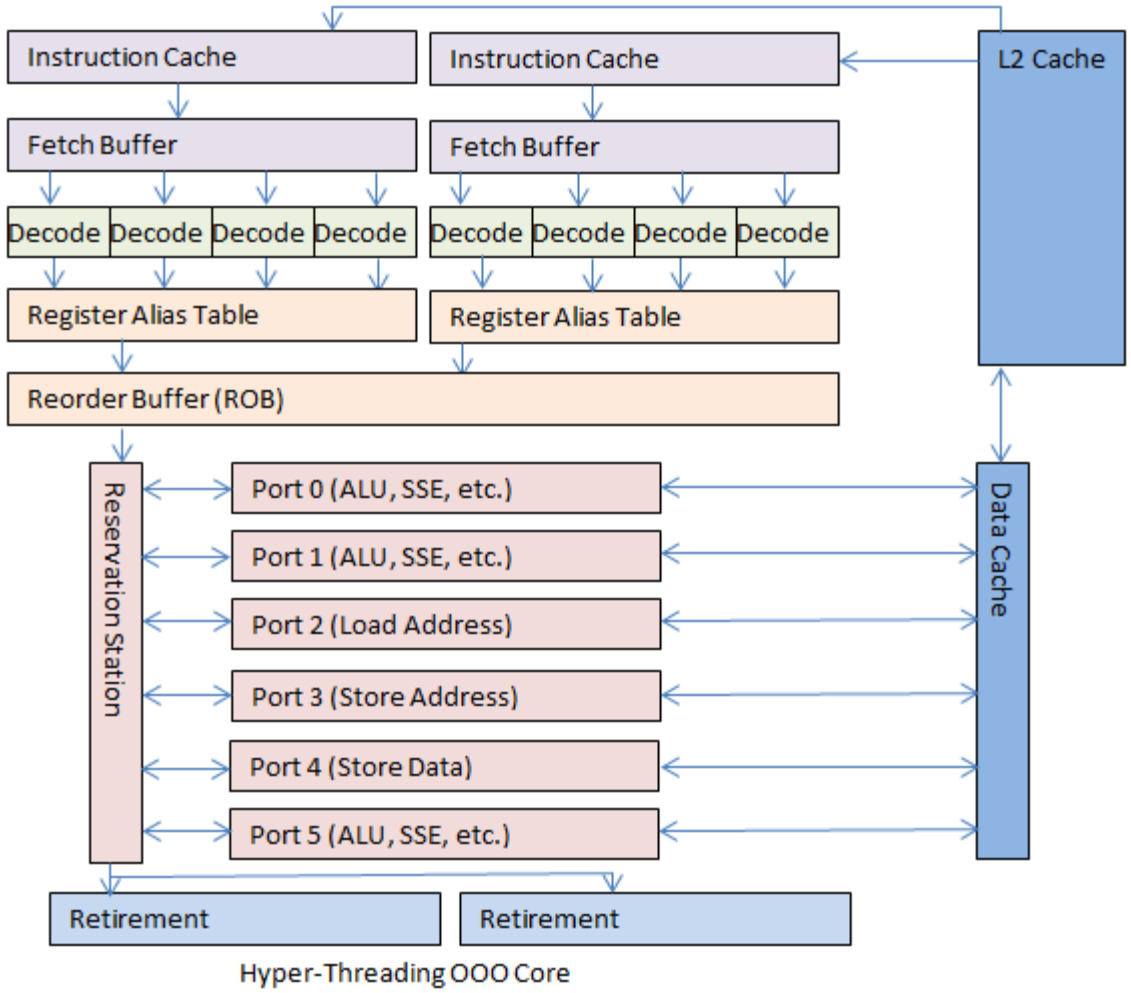


A forma como as tarefas são misturadas pode desperdiçar os recursos extra

Arquitetura Intel Pentium

F	F
D1	D1
D2	D2
EX	EX
WB	WB

Pentium's two parallel superscalar pipelines



Hyper-Threading OOO Core

QUIZ (1/2)

- Assuma 1 instrução/clock, *delayed branch*, 5 estágios de pipeline, *forwarding*, *interlock* nos conflitos de dados envolvendo o *load*. O ciclo tem 10^3 iterações (pipeline cheio).

2. (data hazard so stall)

```
Loop: 1. lw    $t0, 0($s1)
      3. addu  $t0, $t0, $s2
      4. sw    $t0, 0($s1)
      5. addi  $s1, $s1, -4
      7. bne   $s1, $zero, Loop
      8. nop
```

6. (data hazard)

(delayed branch so exec. nop)

Qual é a duração em ciclos de relógio para a execução de uma iteração do ciclo?

1 2 3 4 5 6 7 **8** 9 10

QUIZ (2/2)

- Assuma 1 instrução/*clock*, *delayed branch*, 5 estágios de *pipeline*, *forwarding*, *interlock* nos conflitos de dados envolvendo o *load*. O ciclo tem 10^3 iterações (pipeline cheio). Reescreva o código para optimizar o tempo de execução

```
Loop:   lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        addi    $s1, $s1, -4
        bne     $s1, $zero, Loop
        nop
```

- Qual é a duração em ciclos de relógio para a execução de uma iteração do ciclo?

1 2 3 4 5 6 7 8 9 10

QUIZ (2/2)

- Reescreva o código para optimizar o tempo de execução

(no hazard since extra cycle)

```
Loop:  1. lw    $t0  0($s1)
       2. addi  $s1, $s1, -4
       3. addu  $t0, $t0  $s2
       4. bne   $s1, $zero, Loop
       5. sw    $t0, +4($s1)
```

(modified sw to put past addiu)

- Qual é a duração em ciclos de relógio para a execução de uma iteração do ciclo?

1 2 3 4 **5** 6 7 8 9 10

Exercício

- Considere o seguinte conjunto de instruções MIPS:

```
addi $t0, $t1, 100
sw $t3, 8($t0)
lw $t5, 0($t6)
or $t3, $t0, $t5
lw $t2, 4($t0)
add $t1, $t3, $t2
```

Indique o número de ciclos de relógio que este trecho demorara a executar sem pipeline e com pipeline, indicando os *hazards* existentes. Pode optimizar o código para executá-lo em menos ciclos de relógio?

Para saber mais ...

- P&H - Capítulos 4.5 a 4.8 inclusive