Pontifícia Universidade Católica do Rio Grande do Sul
Instituto de Informática
Organização de Computadores - GAPH

Introdução a Paralelismo em Processadores: *Pipelines*, *Hazards* e *Forwarding*

Profs. Fernando Gehm Moraes e Ney Laert Vilar Calazans

19 de outubro de 2022

* Adaptado e expandido a partir de apresentação de Randy Katz, Berkeley

Gaph
Grupo de Apoio ao Projeto de Hardware

Sumário

- *Pipelines*
  - Introdução
  - *Pipelines* em Computadores
  - Arquitetura MIPS
  - Organização MIPS com *Pipeline*
- *Hazards*
  - *Hazards* Estruturais
  - *Hazards* de Dados
  - *Forwarding*

Gaph
Grupo de Apoio ao Projeto de Hardware

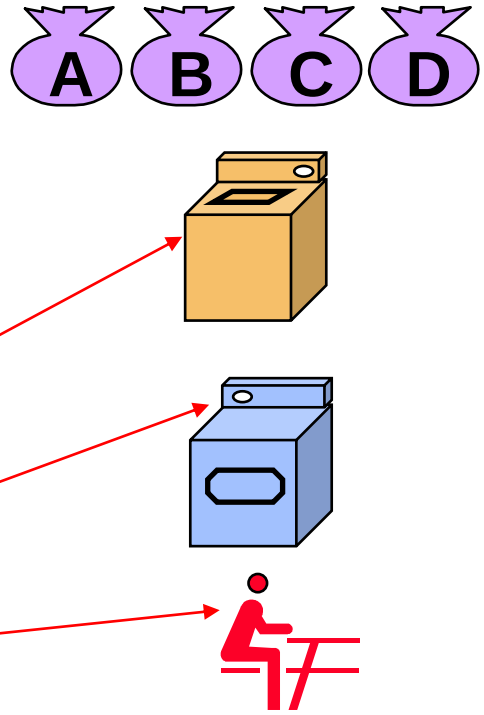
Sumário

- *Pipelines*
  - Introdução
    - Bibliografia
      - Hennessy, J. L. & Paterson, D. A. *Computer Architecture: a quantitative approach*. Morgan Kaufmann, Segunda Edição, 1996
        - Cap 3 (Pipelines com estudo de caso - DLX)
        - Cap 2 (Especificação da arquitetura DLX)
        - Cap 4 (Pipeline Avançado)

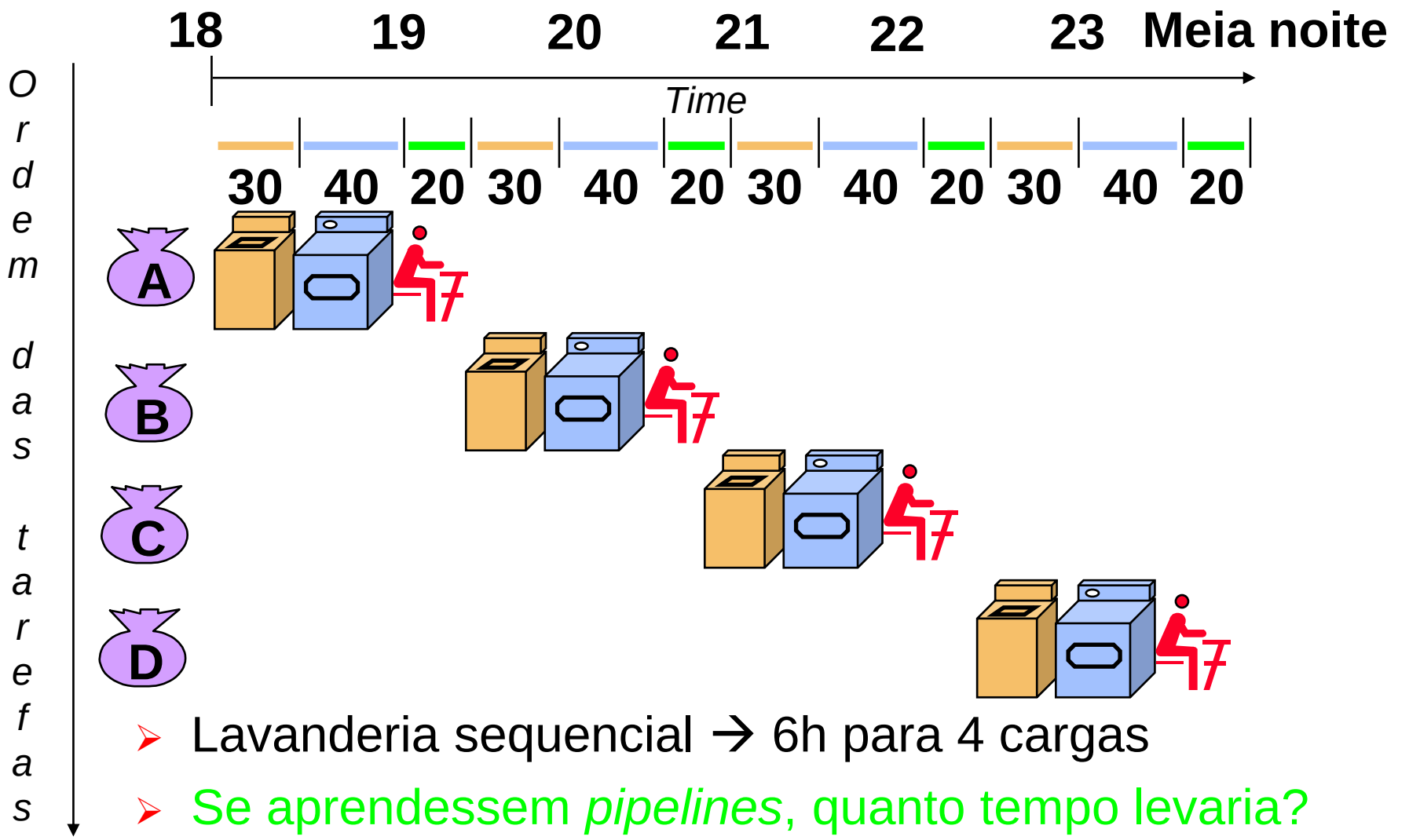
Gaph
Grupo de Apoio ao Projeto de Hardware

Pipeline é Natural!

- Exemplo da Lavanderia
- Ana, Bruno, Cristiane e Daniela têm cada um uma trouxa de roupas para lavar, secar e dobrar
- Lavagem leva 30 minutos
- Secagem leva 40 minutos
- Dobragem leva 20 minutos

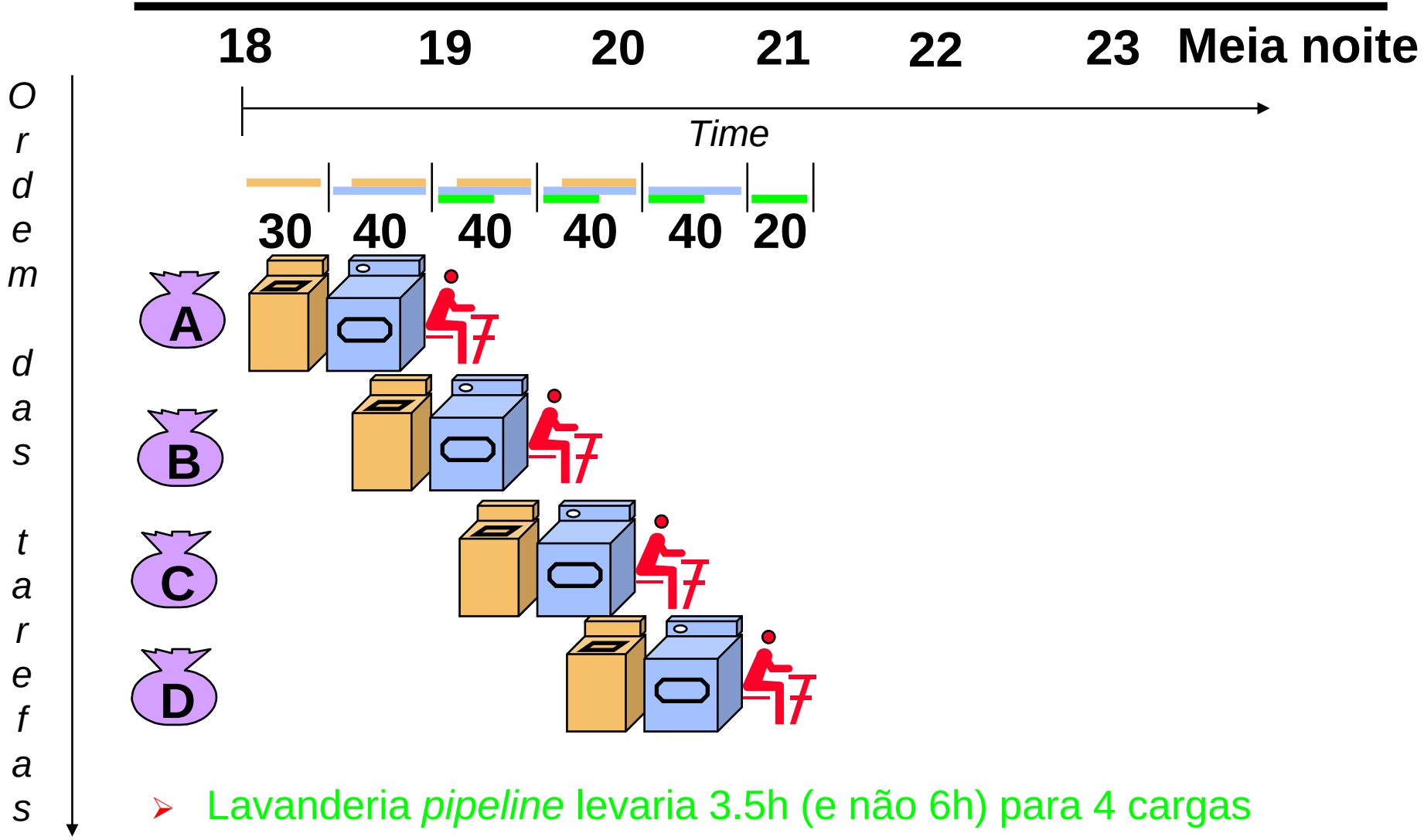


Lavanderia Sequencial



- Lavanderia sequencial → 6h para 4 cargas
- Se aprendessem *pipelines*, quanto tempo levaria?

Lavanderia *pipeline*

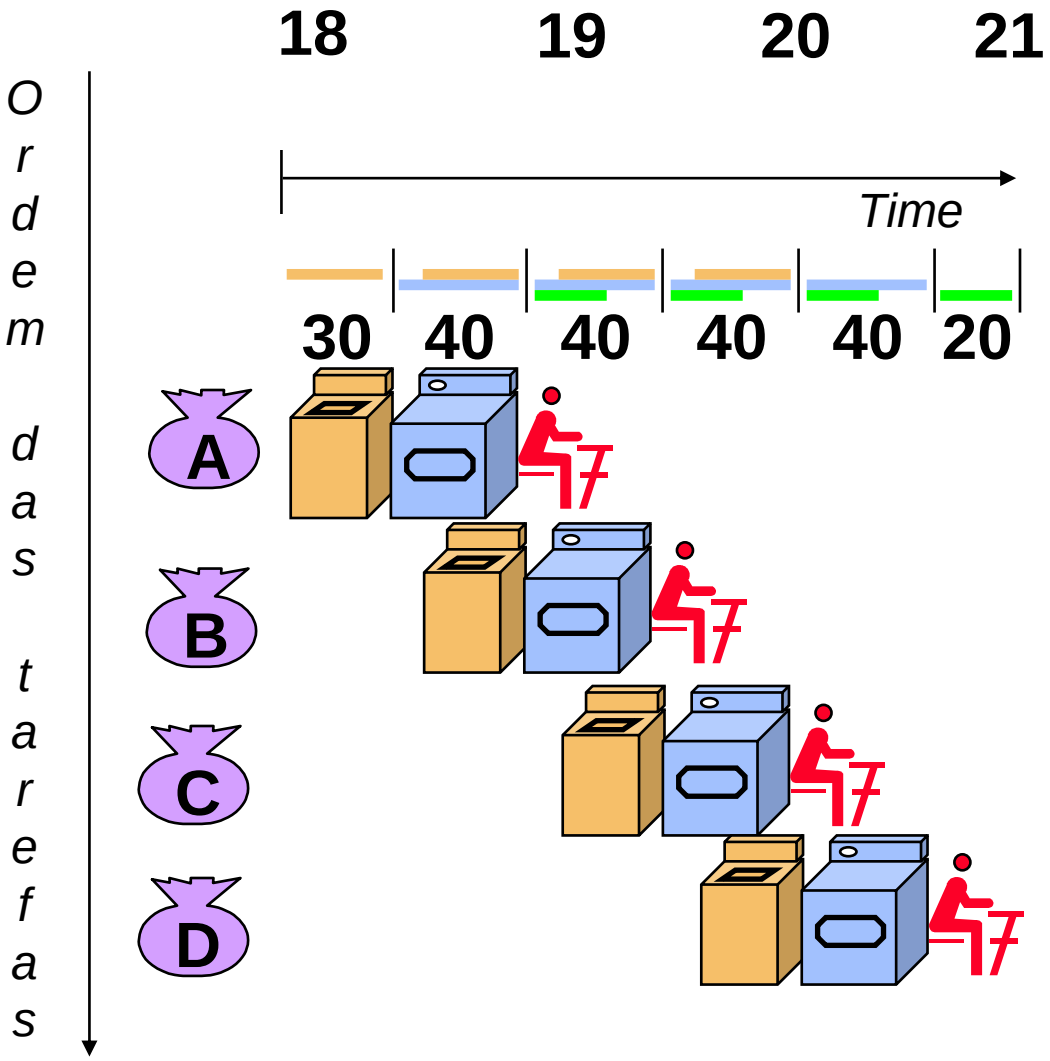


- Lavanderia *pipeline* levaria 3.5h (e não 6h) para 4 cargas

Definições para *Pipelines*

- *Pipeline* = em inglês → tubo, oleoduto - instruções entram numa ponta e são processadas na ordem de entrada
- Tubo é dividido em estágios ou segmentos
- Tempo que uma instrução fica no tubo = latência
- Número de instruções executadas na unidade de tempo = desempenho ou "*throughput*"
- Tempo que uma instrução permanece em um estágio = ciclo de máquina – em Hw, normalmente corresponde a um ciclo de relógio (excepcionalmente dois)
- Balanceamento - medida da uniformidade do tempo gasto em cada estágio

Lições ensinadas por *Pipelines*



- *Pipeline* não reduz a latência de uma única tarefa, ajuda no *throughput* de todo o trabalho
- A taxa de operação do *pipeline* é limitada pelo estágio mais lento
- Tarefas múltiplas operam de forma simultânea
- Aceleração potencial (=*speedup*) = Número de estágios do pipe
- "Comprimentos" desbalanceados de estágios reduzem o *speedup*
- Tempo para "preencher" o pipeline e tempo para "drená-lo" reduzem o *speedup*

Sumário

- *Pipelines*
  - Introdução
    - *Pipelines* em Computadores

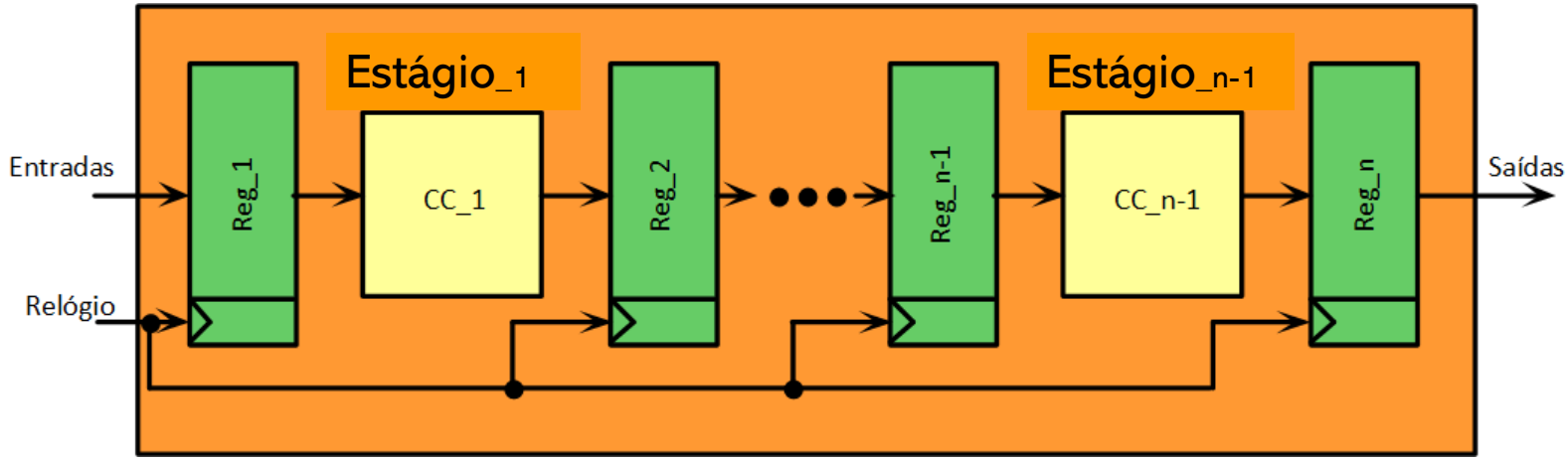
Gaph
Grupo de Apoio ao Projeto de Hardware

Pipelines em Computadores

- Técnica de implementação - múltiplas instruções com execução superposta
- Chave para criar processadores velozes, hoje
- Similar a uma linha de montagem de automóveis
  - Linha de montagem → vários estágios; cada estágio em paralelo com outros, operando sobre automóveis diferentes
  - Pipeline em computadores → cada estágio completa parte de instrução; como antes, diferentes estágios atuam sobre partes de diferentes instruções; Registradores separam estágios
  - Partes de uma instrução → Busca, busca de operandos, execução
  - **Cuidado! Diferença** → Cada carro em uma linha de montagem de automóveis é independente de todos os outros carros. Isto não acontece com instruções, porquê? Pode isto gerar problemas?

Organização Geral *Pipeline*

- Alternância de elementos de memória e blocos combinacionais
  - Memória → guarda dados entre estágios (entre ciclos de relógio)
  - CC_i → lógica combinacional, processam informação



Pipelines em Computadores

- Se estágios perfeitamente “balanceados”
  - tempo para terminar de executar instruções com pipeline = tempo por instrução na máquina sem pipeline / número de estágios no pipeline
- Meta do projetista - balancear estágios

Pipelines - Vantagens e Inconvenientes

- Vantagens
  - reduz tempo médio de execução de programas
  - reduz o CPI (*clocks* por instrução) médio
  - reduz duração do ciclo de *clock*
  - acelera processamento, sem mudar forma de programação
- Inconvenientes
  - estágios em geral não podem ser totalmente balanceados
  - implementação complexa, acrescenta custos (*hardware*, tempo)
  - para ser implementado, conjunto de instruções deve ser simples
- Conclusão
  - *pipelines* são difíceis de implementar, fáceis de usar

Sumário

- *Pipelines*
  - Introdução
  - *Pipelines* em Computadores
    - Arquitetura MIPS

Gaph
Grupo de Apoio ao Projeto de Hardware

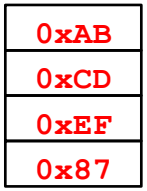
Arquitetura MIPS

- Microprocessador RISC de 32 bits, *load-store*
  - 32 registradores de 32 bits de propósito geral (GPRs) - $0-$31
  - registradores de ponto flutuante (FPRs) visíveis como precisão simples, 32x32 ($f0, $f1, ..., $f31) ou precisão dupla 16x64 ($f0, $f2, ..., $f30)
    - Não estudamos, ainda...
  - $0 é a constante 0 com nome de registrador
  - Alguns registradores especiais - BadVAddr, Status, Cause, EPC
    - Não estudamos aqui

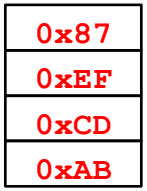
- Modos de endereçamento principais
  - a registrador - e.g. ADDU Rd, Rs, Rt
  - imediato com operando de 16 bits - ADDIU Rt, Rs, immed
  - base-deslocamento com *offset* de 16 bits - LW Rt,offset(Rs)
  - pseudo-absoluto com operando de 26 bits - J pseudo_address

Arquitetura MIPS

- Barramentos de dados e endereços de 32 bits
- Portanto, cada leitura da memória traz para dentro do processador 32 bits, i.e.
  - 4 bytes ou
  - 2 meias-palavras ou
  - 1 palavra
- Memória endereçável a *byte*, modo *Little Endian*
  - *Big Endian* - dados de mais de um *byte* são guardados em posições de memória a partir do byte mais significativo (similar a SPARC, PPC etc.)
    - Ex: 0xABCDEF87 é guardado na memória na ordem 0xAB, 0xCD, 0xEF, 0x87



  - *Little Endian* - dados de mais de um *byte* são guardados em posições de memória a partir do byte menos significativo (Intel)
    - Ex: 0xABCDEF87 é guardado na memória na ordem 0x87, 0xEF, 0xCD, 0xAB

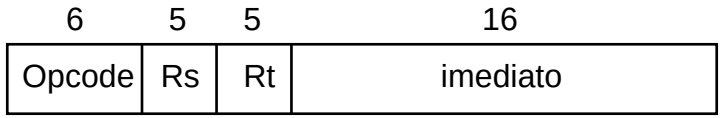


  - acesso a byte, meia-palavra (16 bits) ou palavra (32bits)

Arquitetura MIPS

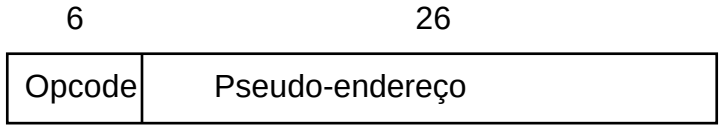
- Formatos de Instrução
  - Tipo I
    - *Loads*, *stores*, de bytes, meia-palavras e palavras, todos os imediatos, saltos condicionais (Rs e Rd registradores), salto incondicional a registrador (JR)
  - Tipo R
    - operações com a ULA e registradores, ***func*** diz a operação, operações com registradores, ***shamt*** usado para especificar número de bits a deslocar em *shifts*
  - Tipo J
    - salto incondicional, exceções e retornos de exceção

Tipo I



Tipo R

Tipo J



Sumário

- *Pipelines*
  - Introdução
  - *Pipelines* em Computadores
  - Arquitetura MIPS
    - Organização MIPS com *Pipeline*

Gaph
Grupo de Apoio ao Projeto de Hardware

Ciclos de Máquina do MIPS - *1 de 2*

Legenda:

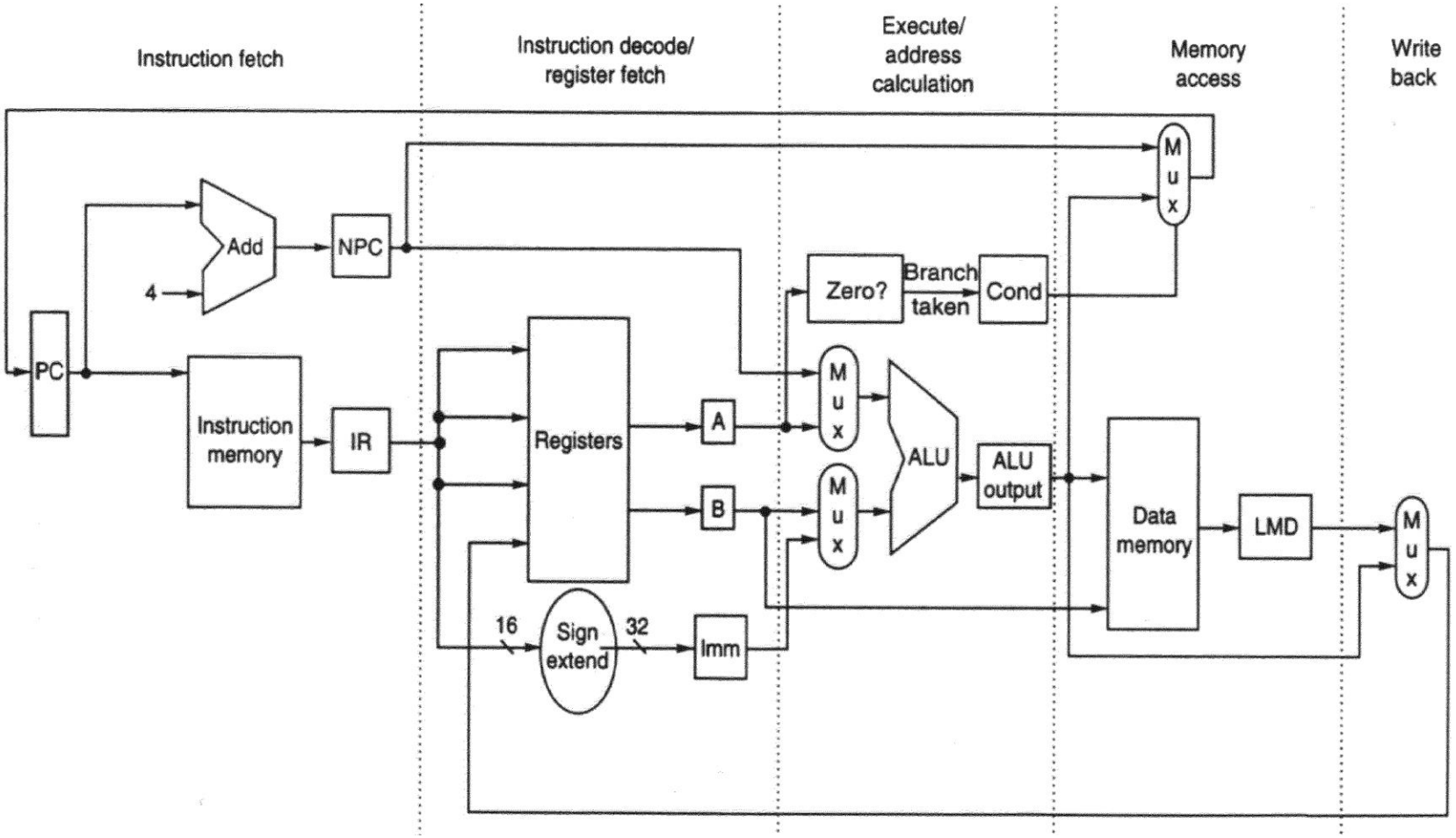
← - Escrita do valor da expressão à direita da seta no registrador/posição de memória à esquerda da seta
`&` - Concatenação dos bits a direita e esquerda do símbolo &
`;` - Separador de commandos executados em paralelo no hardware
`Mem(x)` – Conteúdo da posição de memória de endereço x
`Regs(x)` – Elemento do Banco de Resgistradores com endereço (índice) x
`op` – Algum operador disponível na ALU
`IR, PC, NPC, RALU, MDR` – registradores da organização MIPS
`salta` – fio de saída do comparador (transporta o resultado da comparação, 1 – verdadeiro, 0 – fals0)
`X[a:b]` – identificador da sequência de bits de índice a até b do vetor de bits X

- 1- Ciclo de Busca de Instrução (Instruction Fetch - IF)
  - `IR ← Mem(PC); NPC ← PC+4;`
- 2 - Ciclo de decodificação de instrução/busca de registrador (ID)
  - `R1 ← Regs(IR[25:21]); R2 ← Regs(IR[20:16]); R3 ← (IR[15])`^{16x} `& IR[15:0];`
  - `R1`, `R2`, `R3` são regs temporários; operação que gera `R3` é Extensão de sinal
- 3 - Ciclo de execução e cálculo de endereço efetivo (EX)
  - Referência à memória: `RALU ← R1 + R3;`
  - Instrução Reg-Reg/ALU: `RALU ← R1 op R2;`
    - `op` é um operador, e.g. +, -, shift left, shift right (lógico ou aritmético *through var*), AND, OR, XOR NOR
  - Instrução Reg-Imm/ALU: `RALU ← R1 op R3;`
    - `op` é um operador realizado na ALU, e.g. +, -, shift left, shift right (lógico ou aritmético), AND, OR, XOR
  - Desvios condicionais: `RALU ← NPC + R3; salta ← (R1 op 0);`
    - `op` é um operador relacional, tal como <, >, <=, = etc

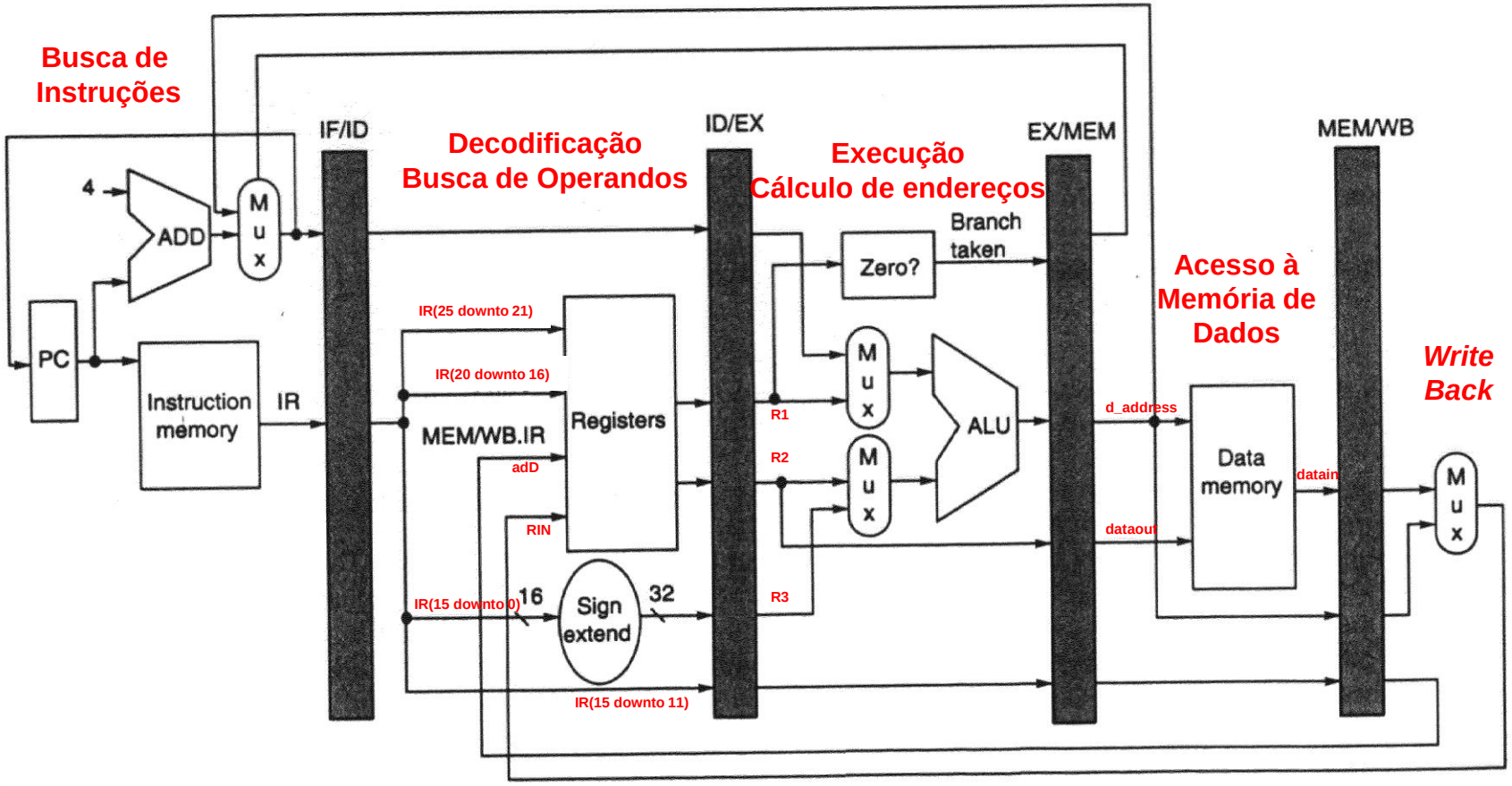
Ciclos de Máquina do *MIPS - 2 de 2*

- 4 - Ciclo de acesso à memória/término de desvio condicional (MEM)
  - Referência à memória: `MDR ← Mem[RALU] ou Mem[RALU] ← R2;`
  - Desvio Condicional: `if (salta) PC ← RALU else PC ← NPC;`
- 5 - Ciclo de atualização ou *write-back* (WB)
  - Instrução Reg-Reg/ALU: `Regs(IR[15:11]) ← RALU;`
  - Instrução Reg-Imm/ALU: `Regs(IR[20:16]) ← RALU;`
  - Instrução Load: `Regs(IR[20:16]) ← MDR;`
- Próxima página ilustra
  - A implementação do Bloco de Dados sem *pipeline*
  - Segue-se a implementação com *pipeline*

Um Bloco de Dados p/ o MIPS

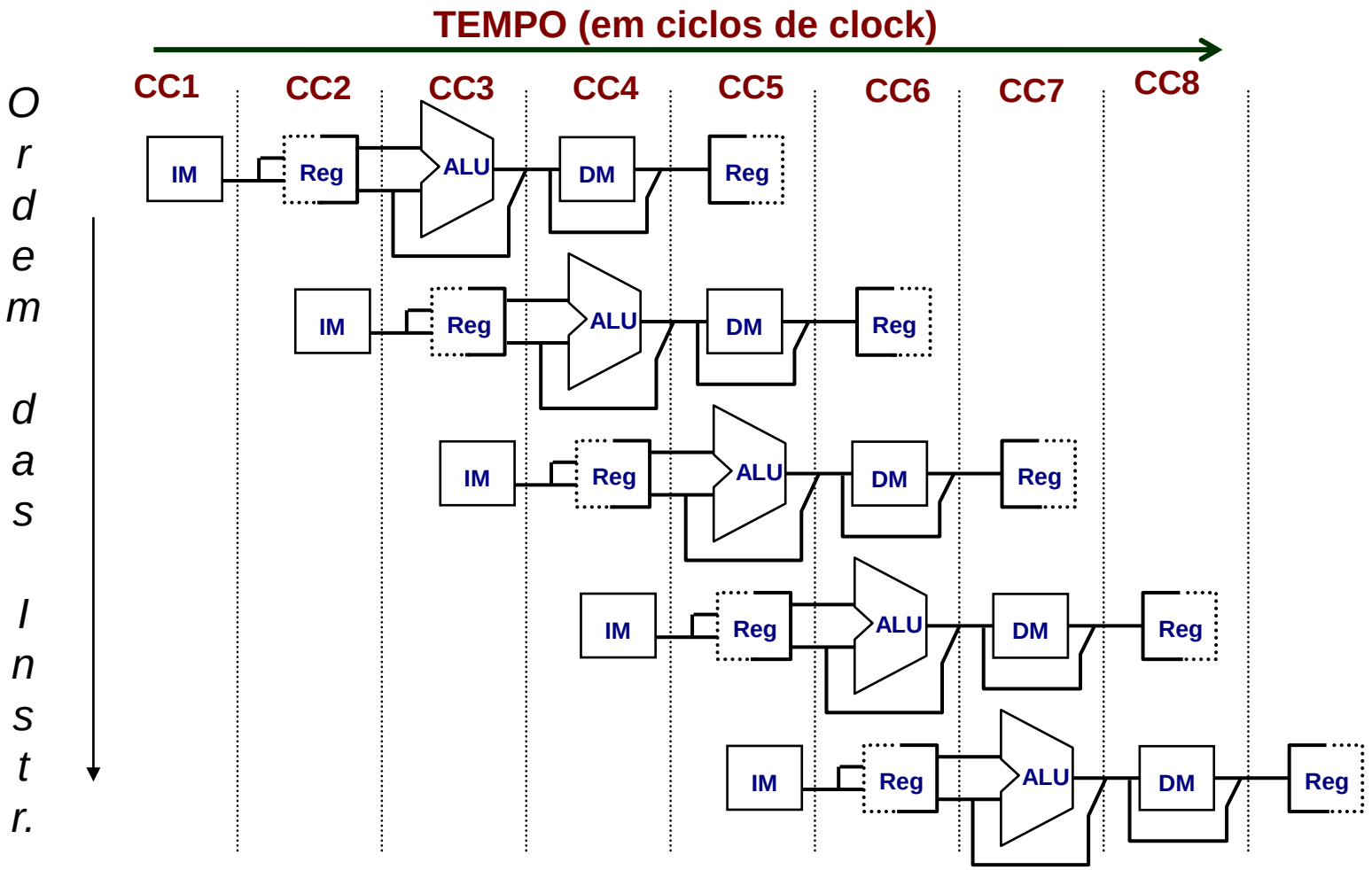


Bloco de Dados MIPS com *Pipeline*



- **Controle de Dados Estacionário**
  - **–decodificação local para cada fase da instrução ou estágio do pipeline**

Pipelines ao longo do Tempo – Caso Ideal



PIPE CHEIO no ciclo 5

Pipeline em Computadores é Complicado!

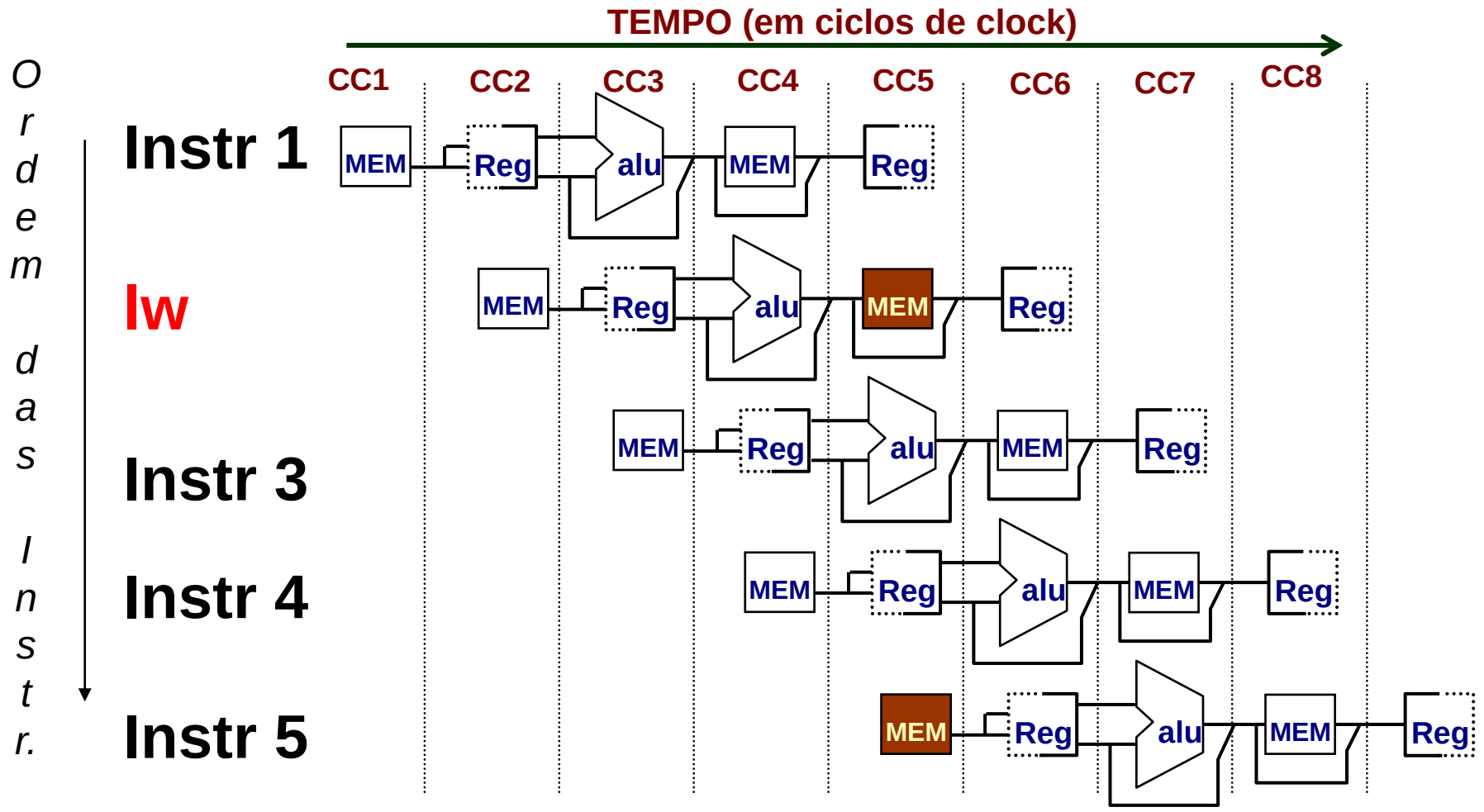
- Limitações de *pipelines*: ***Hazards*** (perigos) evitam que uma próxima instrução execute durante um determinado ciclo de clock
  - ***Hazard* estrutural**: HW não pode dar suporte a uma determinada combinação de instruções
  - ***Hazard* de dados**: Instrução depende do resultado de uma instrução anterior anda no pipeline
  - ***Hazard* de controle**: Pipeline de saltos e outras instruções que mudam o PC
- Solução simples é suspender (***stall***) o pipeline até que o *hazard* desapareça → "**bolhas**" temporais no pipeline (tratado a seguir)

Sumário

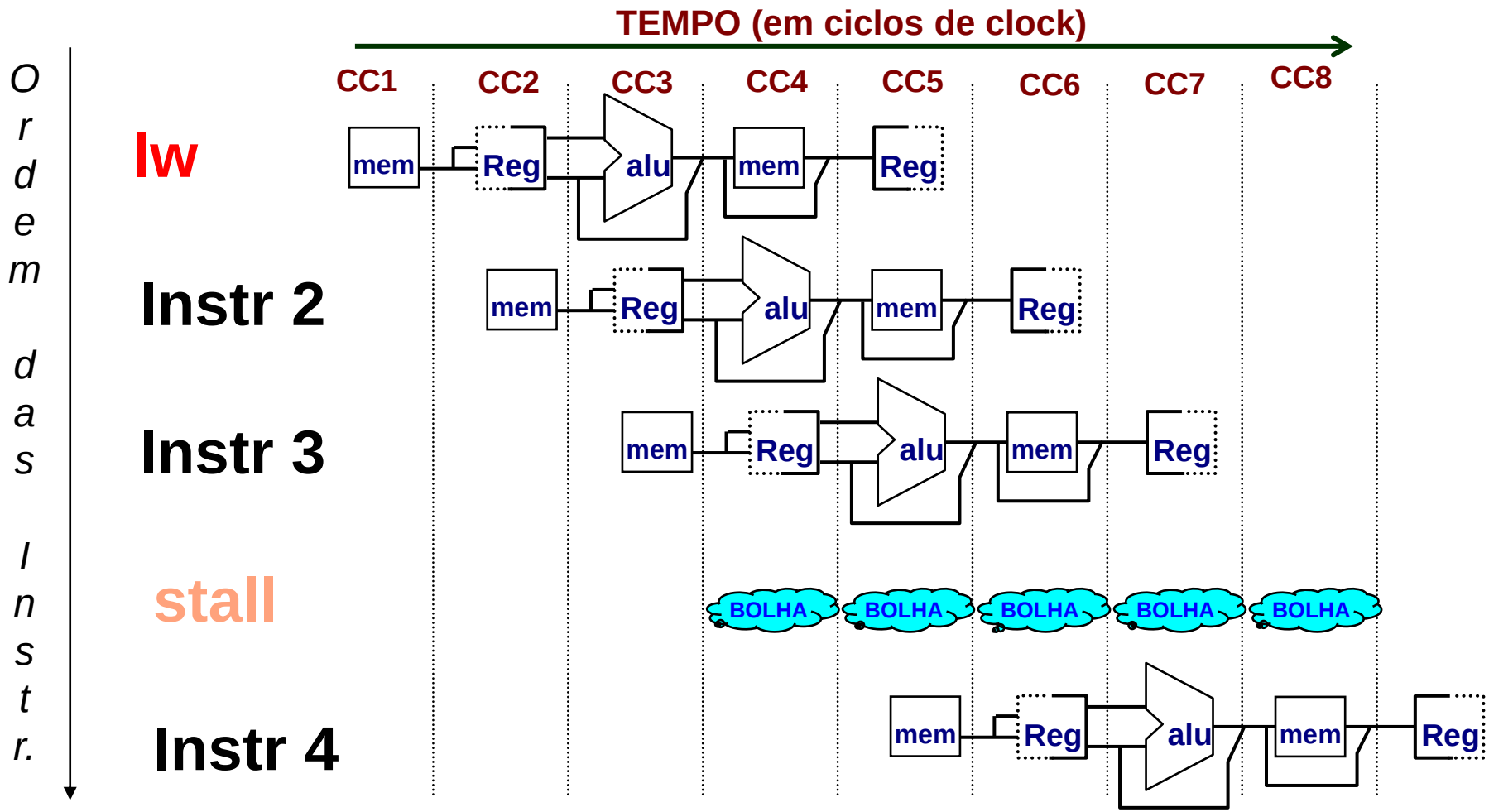
- *Pipelines*
  - Introdução
  - *Pipelines* em Computadores
  - Arquitetura MIPS
  - Organização MIPS com *Pipeline*
- *Hazards*
  - *Hazards* Estruturais

Gaph
Grupo de Apoio ao Projeto de Hardware

Hazard Estrutural / Memória de uma porta ou Org. von Neumann *Fig 3.6, Página 142*



Hazard Estrutural / Memória de uma porta ou Org. von Neumann *Fig 3.7, Página 143*



Equação de *Speed Up* para *Pipelines*

$$\text{Speedup from pipelining} = \frac{\text{Ave Instr Time unpipelined}}{\text{Ave Instr Time pipelined}}$$

$$= \frac{\text{CPI}_{\text{unpipelined}} \times \text{Clock Cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock Cycle}_{\text{pipelined}}}$$

$$= \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

$$\text{Ideal CPI} = \text{CPI}_{\text{unpipelined}} / \text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

Equação de *Speed Up* para *Pipelines*

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}$$

$$\text{Speedup} = \frac{\text{Ideal CPI x Pipeline depth}}{\text{Ideal CPI + Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

Exemplo: duas portas vs. uma porta

- Máquina A: Memória de duas portas
- Máquina B: Memória de uma porta , mas com implementação *pipeline* que possui um clock 1.05 vezes mais rápido (5%)
- CPI ideal = 1 para ambos
- Loads são 40% das instruções executadas
  - $SpeedUp_A$ = Pipeline Depth/(1 + 0) x ($clock_{unpipe}/clock_{pipe}$)
    = Pipeline Depth
  - $SpeedUp_B$ = Pipeline Depth/(1 + 0.4 x 1) x ($clock_{unpipe}/(clock_{unpipe}$ / 1.05)
    = (Pipeline Depth/1.4) x 1.05
    = 0.75 x Pipeline Depth

  $SpeedUp_A/SpeedUp_B$ = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

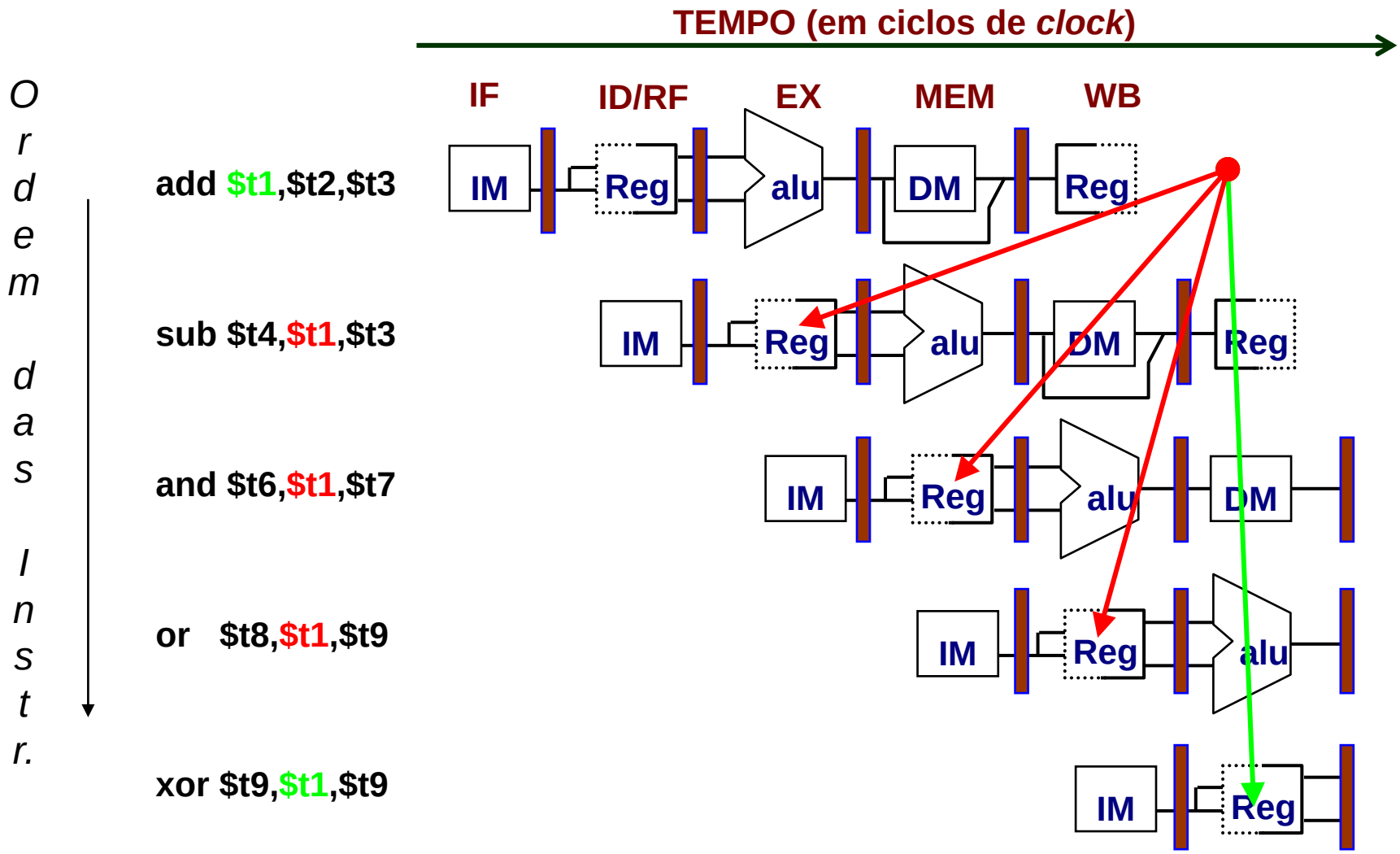
  - A máquina A é 1.33 vezes mais rápida (33% mais rápida que a original) que a máquina B

Sumário

- ✓ *Pipelines*
  - ✓ Introdução
  - ✓ *Pipelines* em Computadores
  - ✓ Arquitetura MIPS
  - ✓ Organização MIPS com *Pipelines*
- ✓ *Hazards*
  - ✓ *Hazards* Estruturais
  - ► *Hazards* de Dados

Gaph
Grupo de Apoio ao Projeto de Hardware

Hazard de Dados em R1 Fig 3.9, Página 147



Três *Hazards* de Dados Genéricos

$Instr_I$ seguida pela $Instr_J$

- Leitura Após Escrita (RAW)
  $Instr_J$ tenta ler operando antes que a $Instr_I$ escreva ele

Três *Hazards* de Dados Genéricos

$Instr_I$ seguida pela $Instr_J$

- Escrita Após Leitura (WAR)

  $Instr_J$ tenta escrever operando antes que a $Instr_I$ leia ele

- Não pode acontecer no pipeline do MIPS porque
  - Todas as instruções levam 5 estágios
  - Leituras são sempre no estágio 2 e
  - Escritas são sempre no estágio 5

Três *Hazards* de Dados Genéricos

$Instr_I$ seguida pela $Instr_J$

- Escrita Após Escrita (WAW)
  $Instr_J$ tenta escrever operando antes que a $Instr_I$ o escreva
  - Quando ocorre, dá resultados incorretos ($Instr_I$ e não $Instr_J$)
- Não pode acontecer no pipeline do MIPS, pois:
  - Todas instruções ocupam 5 estágios e
  - Escritas são sempre no estágio 5
- *Pipelines* mais complicados podem apresentar *hazards* dos tipos WAR e WAW

Sumário

- ✓ *Pipeline*
  - ✓ Introdução
  - ✓ Pipelines em Computadores
  - ✓ Arquitetura MIPS
  - ✓ Organização MIPS com Pipelines
- ✓ *Hazards*
  - ✓ *Hazards* Estruturais
  - ✓ *Hazards* de Dados
  - ► *Forwarding*

Gaph
Grupo de Apoio ao Projeto de Hardware

Forwarding pode evitar *Hazard* de Dados! Fig 3.10, Página 149

Tempo (em ciclos de clock)

Ordem das Instr.

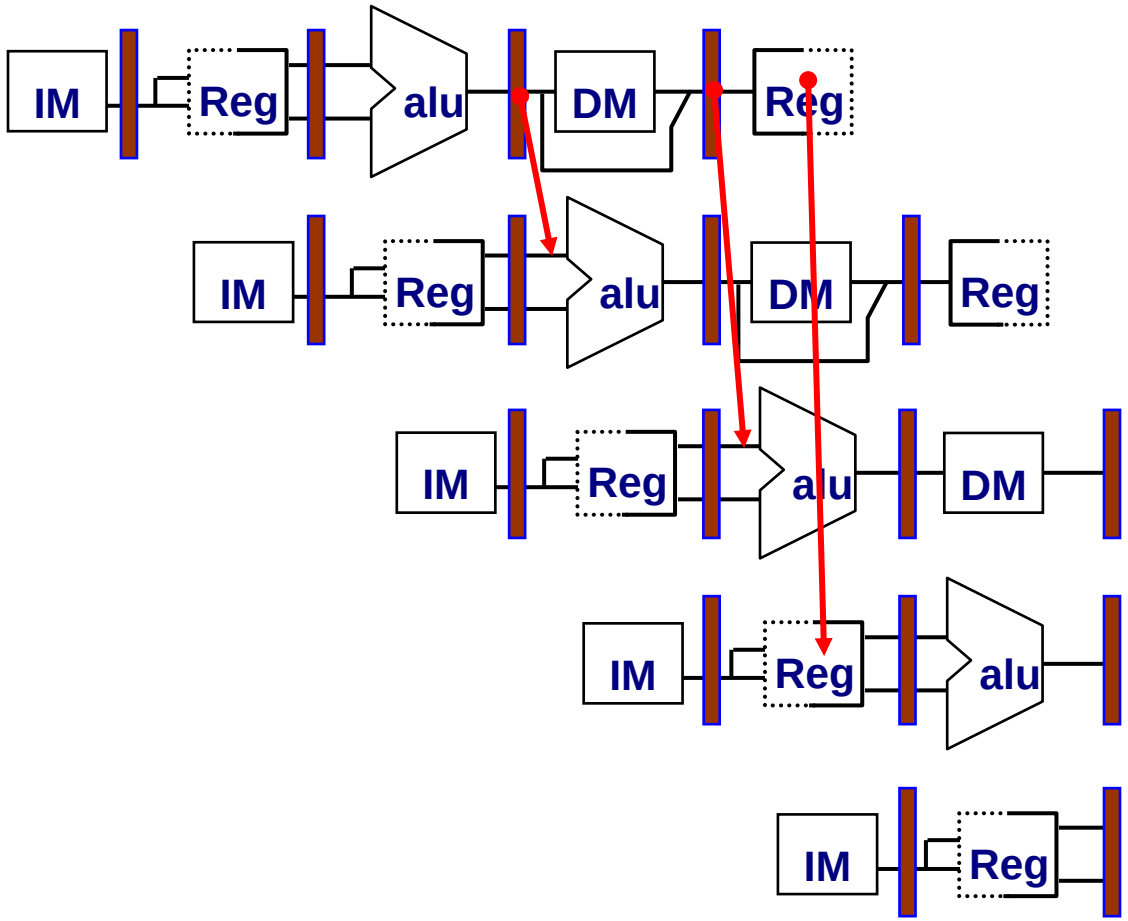
add $t1,$t2,$t3

sub $t4,$t1,$t3

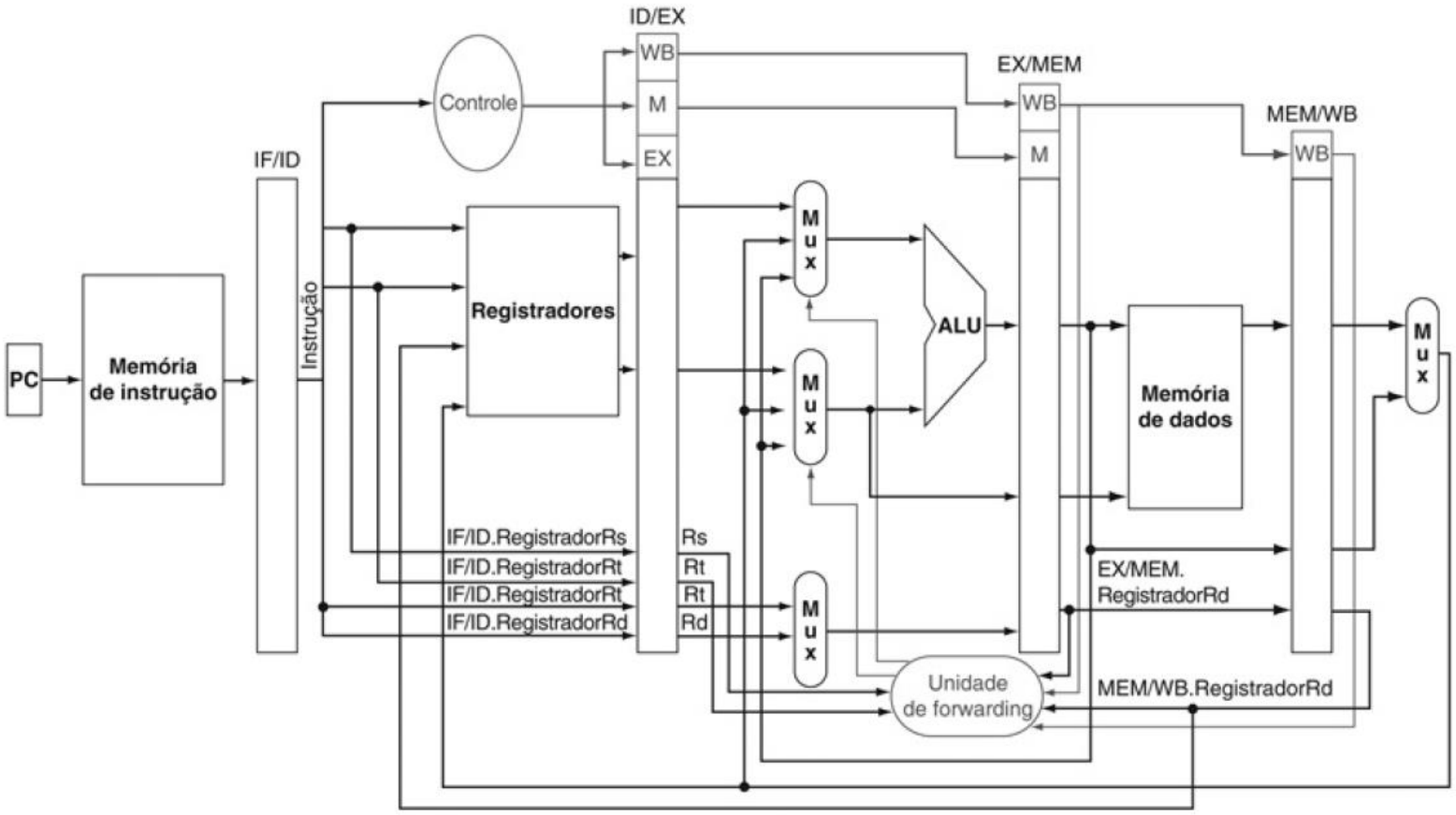
and $t6,$t1,$t7

or $t8,$t1,$t9

xor $t9,$t2,$t1



Mudança de Hw para *Forwarding – P&H 5ªed*



Hazard de Dados mesmo com *Forwarding* *Fig 3.12, Página 153*

Ordem das Instr.

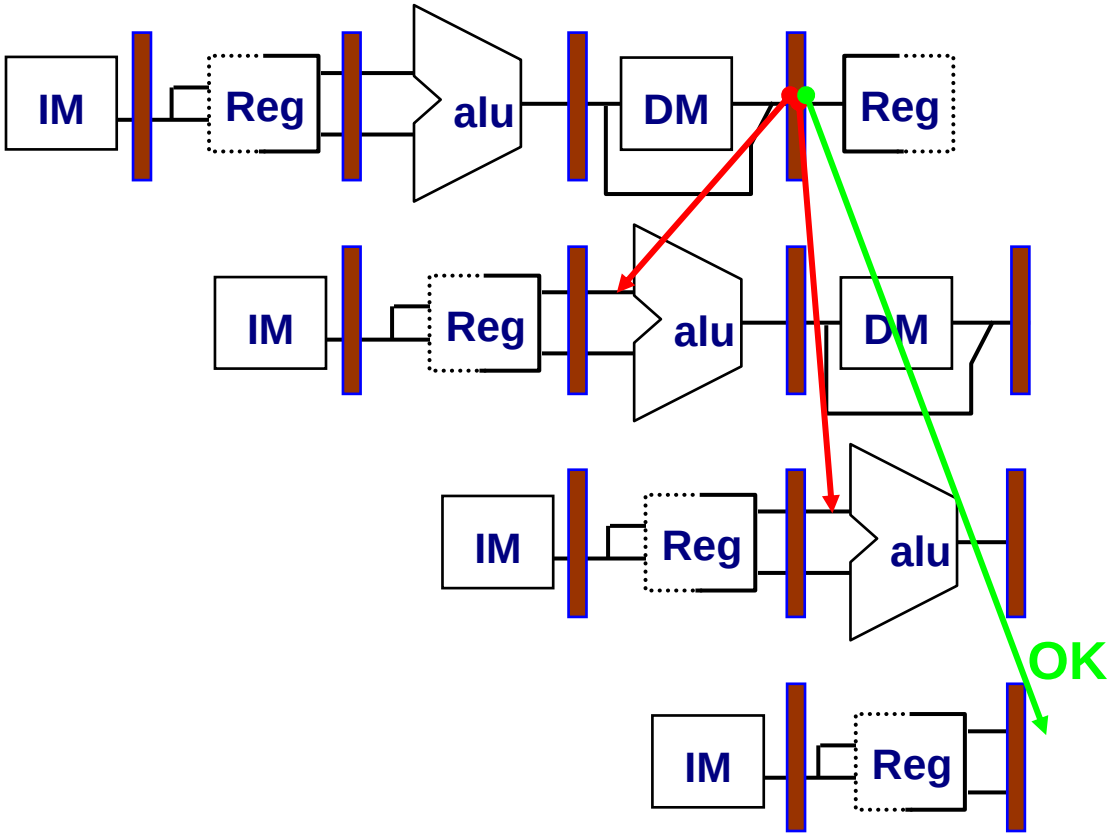
Tempo (em ciclos de clock)

lw r1, 0(r2)

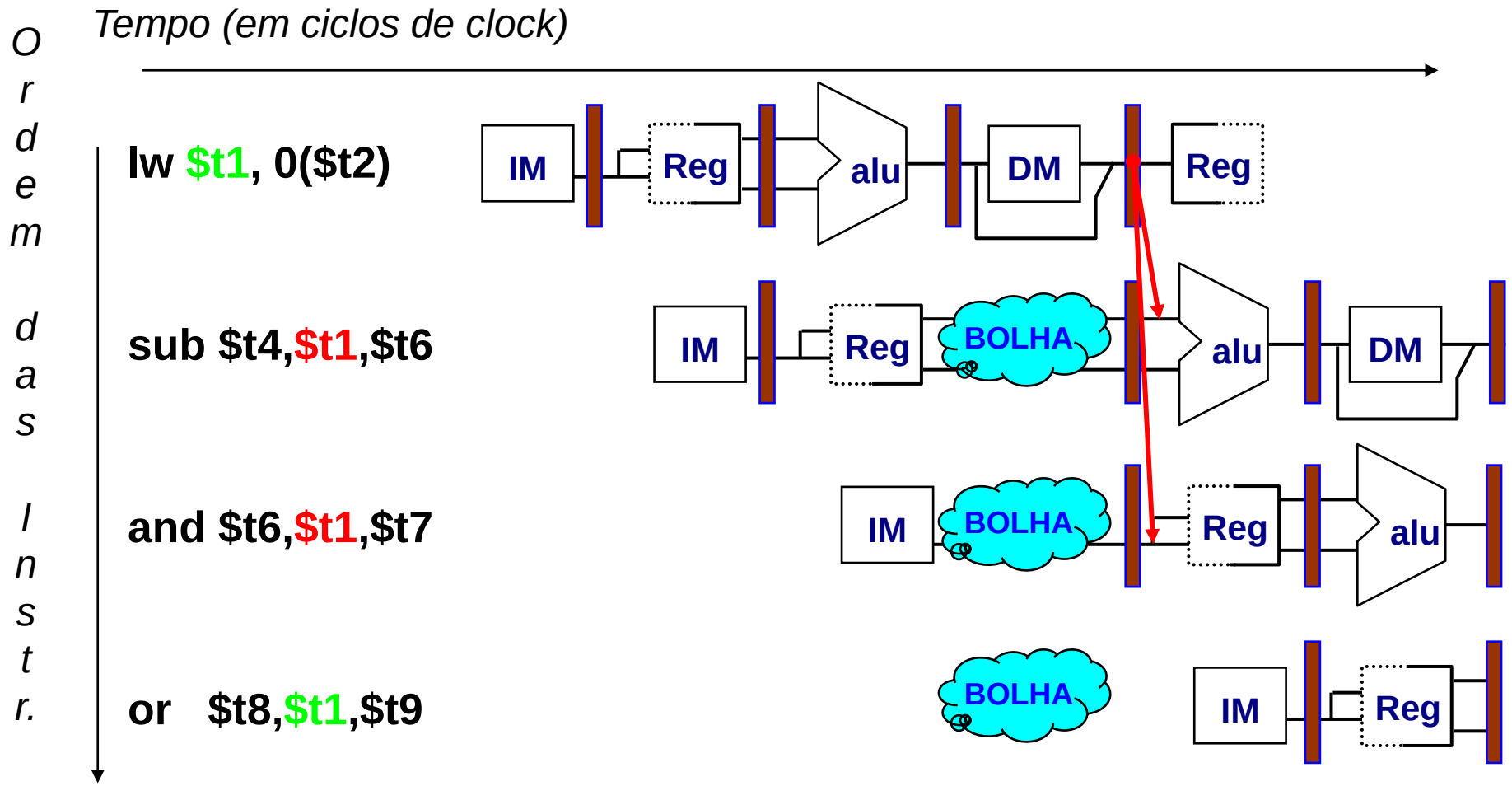
sub r4,r1,r6

and r6,r1,r7

or r8,r1,r9



Hazard de Dados mesmo com *Forwarding* *Fig 3.12, Página 153*



Escalonamento em *Software* para evitar *Hazards* de *Loads*

O compilador tenta produzir código eficiente para

a = b + c;

d = e – f;

assumindo a, b, c, d ,e, *e* f em memória.

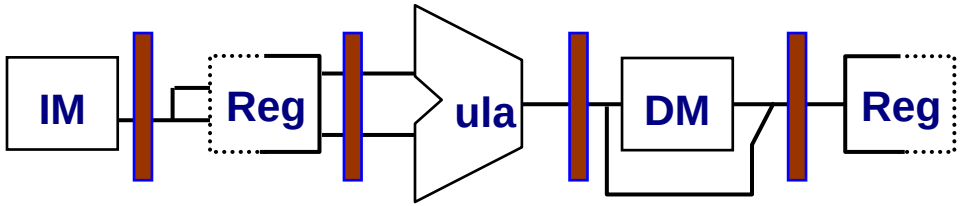
Código Lento

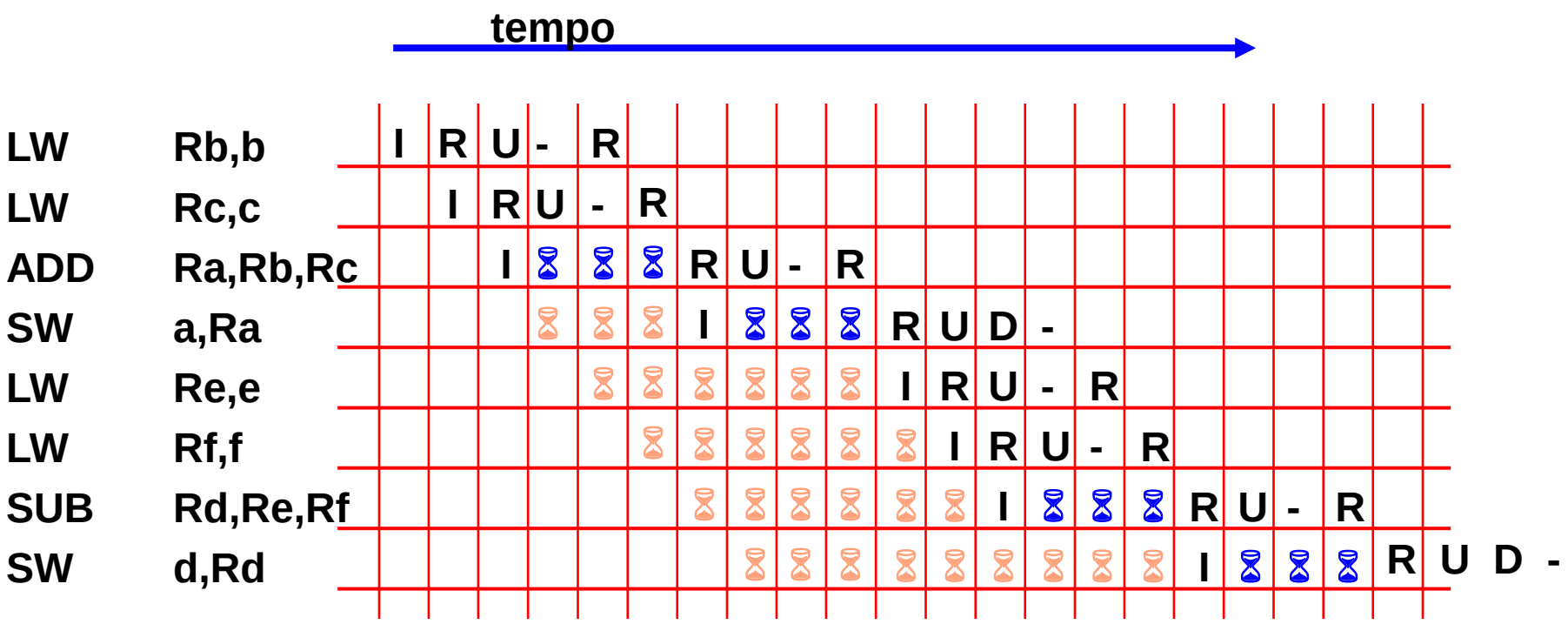
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Código Rápido

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

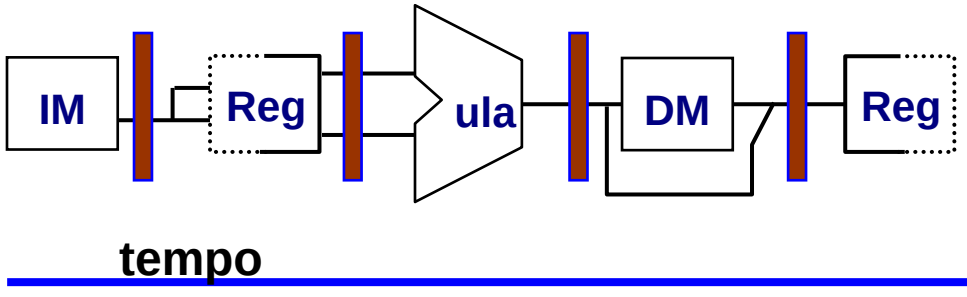
Código Lento – sem *fowarding*





- : significa estágio não utilizado na operação corrente, só transfere informação

Código Rápido - sem *forwarding*, Re-Escalonado

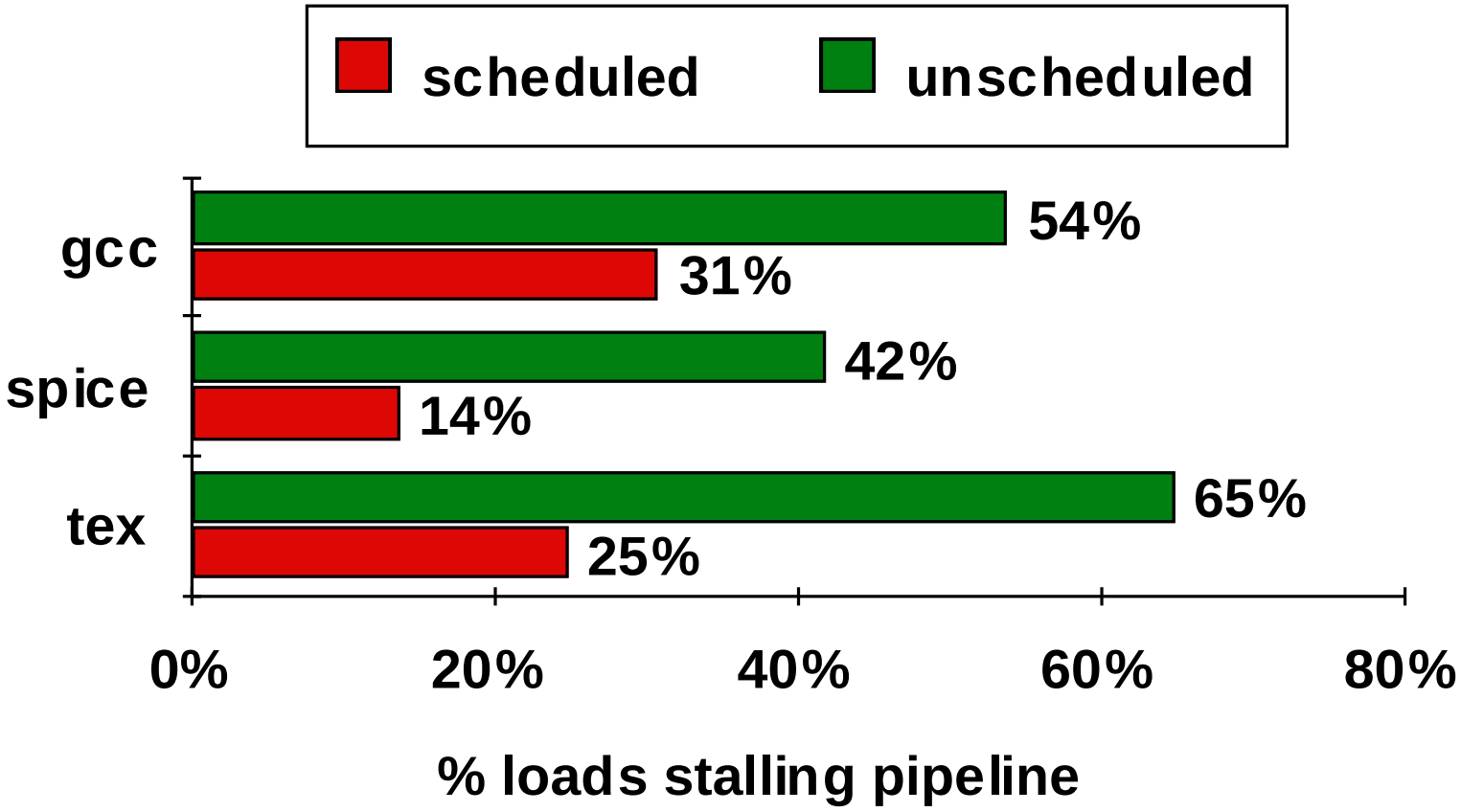


LW	Rb,b	I	R	U	D	R																
LW	Rc,c		I	R	U	D	R															
LW	Re,e			I	R	U	D	R														
ADD	Ra,Rb,Rc				I	⧖	⧖	R	U	-	R											
LW	Rf,f					⧖	⧖	I	R	U	D	R										
SW	a,Ra						⧖	⧖	I	⧖	⧖	R	U	D	-							
SUB	Rd,Re,Rf							⧖	⧖	⧖	⧖	I	R	U	-	R						
SW	d,Rd								⧖	⧖	⧖	⧖	I	⧖	⧖	⧖	R	U	D	-		

19 ciclos ao invés dos 24 anteriores. 25% mais rápido!!!!

- : significa estágio não utilizado na operação corrente, só transfere informação

Compilador pode evitar *Stalls* de *Loads*



Resumo de *Pipelines*

- Superpõe tarefas, é fácil se tarefas são totalmente independentes
- *Speed Up* $\leq$ Profundidade do Pipeline; se CPI ideal é 1, então

$$\textbf{Speedup} = \frac{\textbf{Pipeline Depth}}{\textbf{1 + Pipeline stall CPI}} \times \frac{\textbf{Clock Cycle Unpipelined}}{\textbf{Clock Cycle Pipelined}}$$

- *Hazards* limitam desempenho de *pipelines*
  - Estrutural: precisa de mais recursos de HW
  - Dados: precisa de *forwarding* e escalonamento por compilador
  - Controle: discute-se em detalhe em outra disciplina (AOC)

Era isso! Até a próxima!