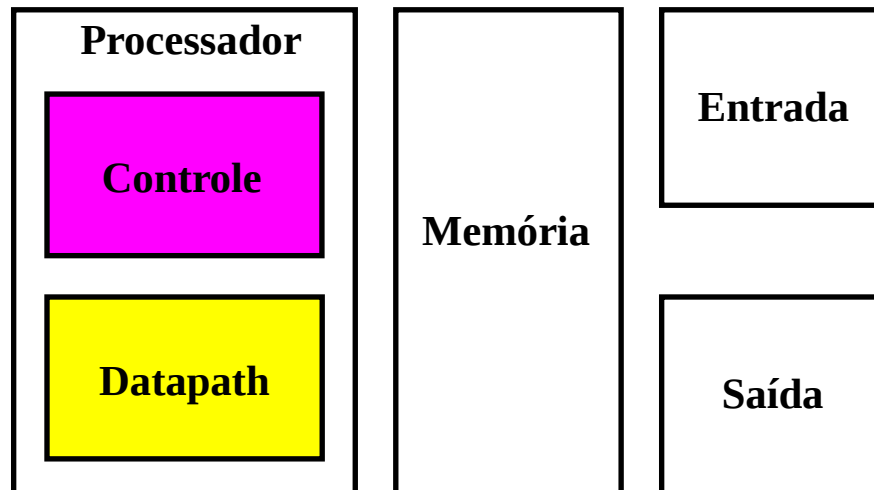


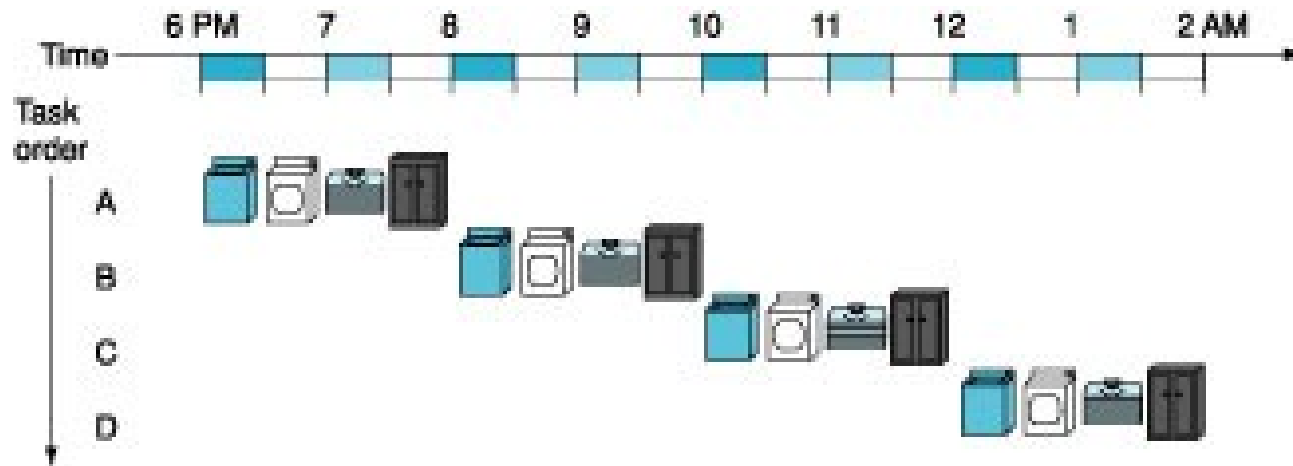
Pipelining: princípio e hazards



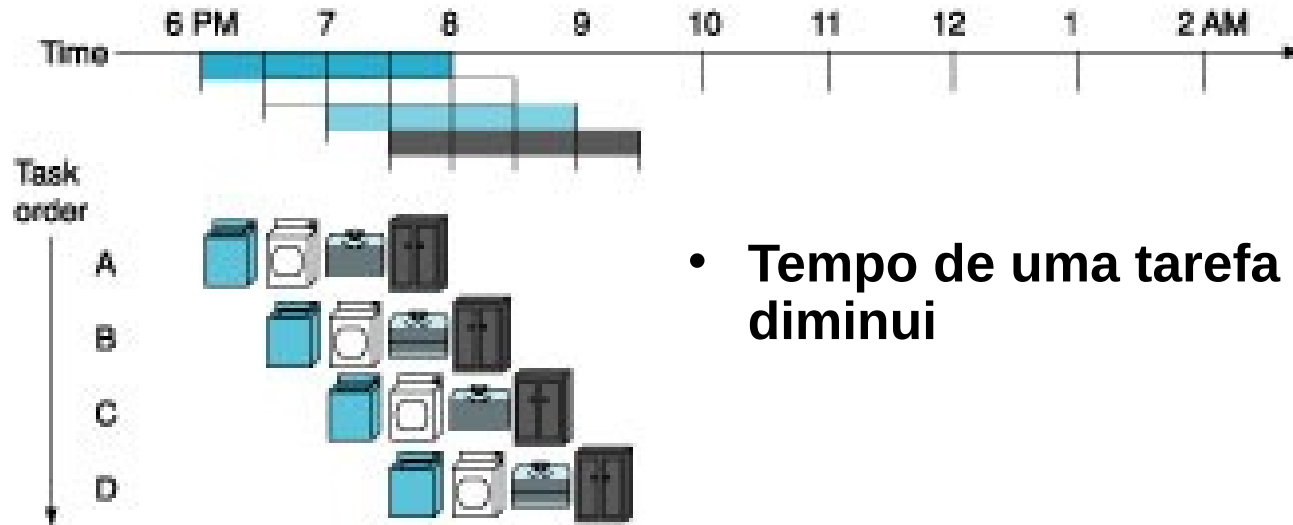
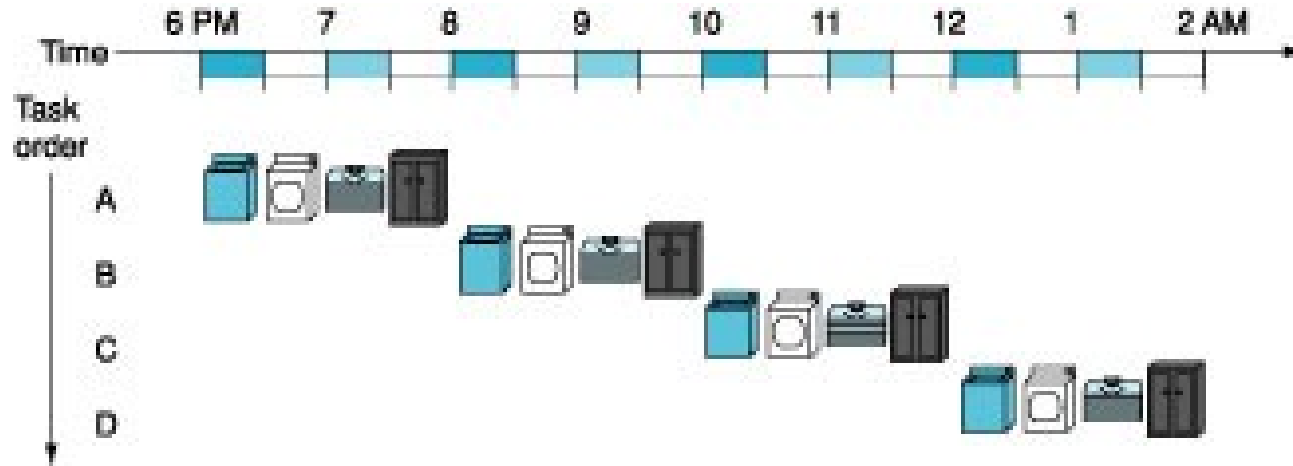
Discussão

- Pode-se afirmar que:
- “Se **cada** instrução de um programa é acelerada, então o programa é acelerado” ?
- “Se **nenhuma** instrução de um programa é acelerada, então não há como se acelerar o programa” ?

Pipelining: uma analogia

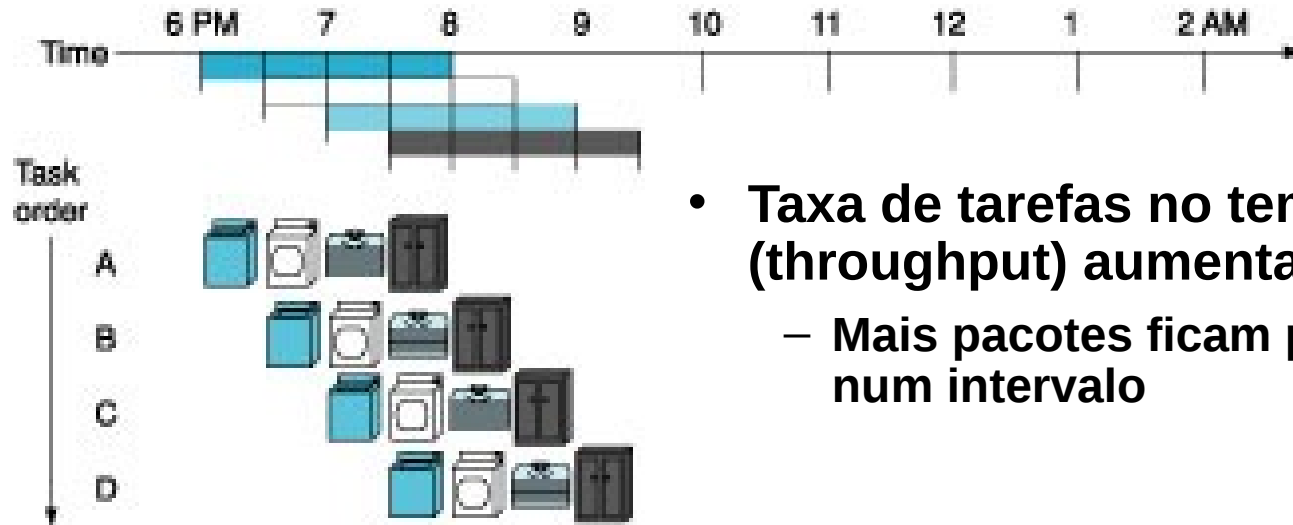
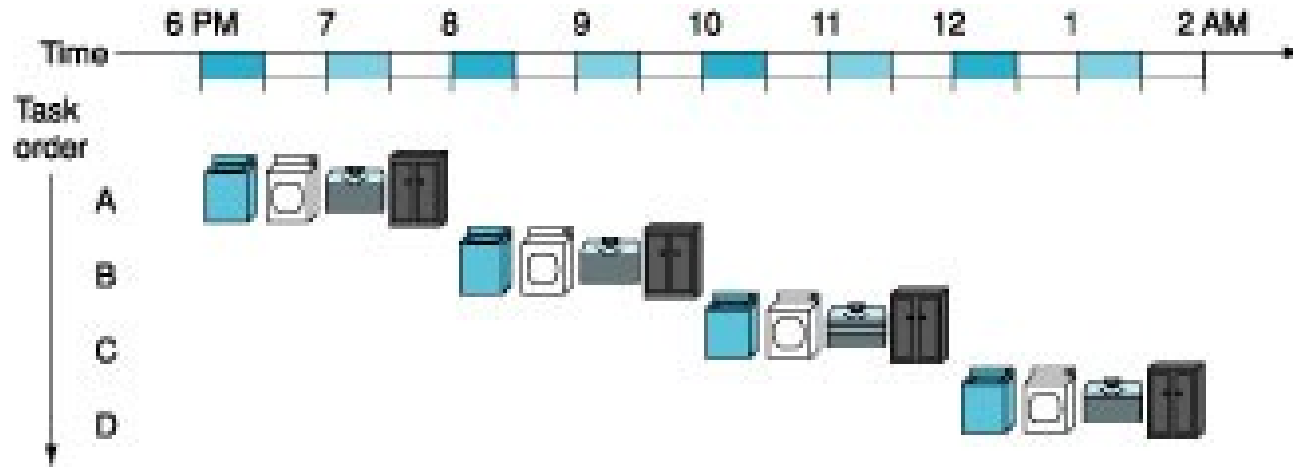


Pipelining: uma analogia



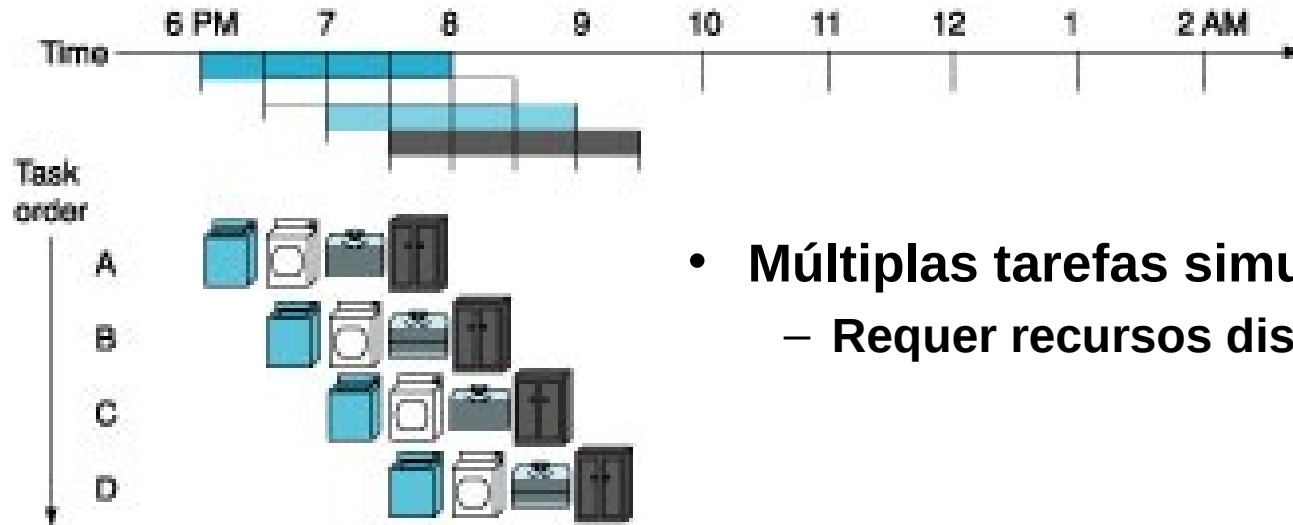
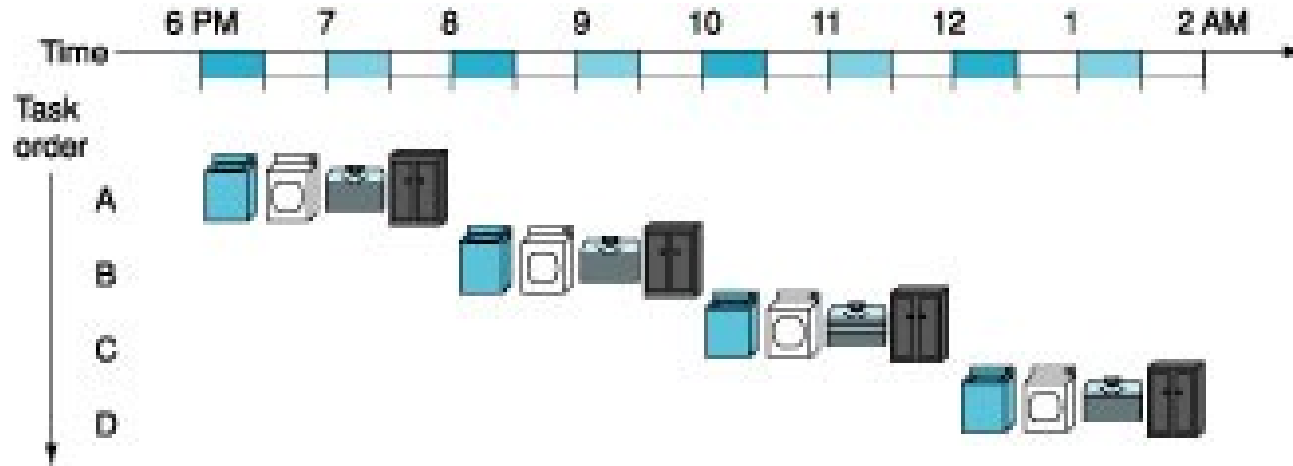
- Tempo de uma tarefa não diminui

Pipelining: uma analogia



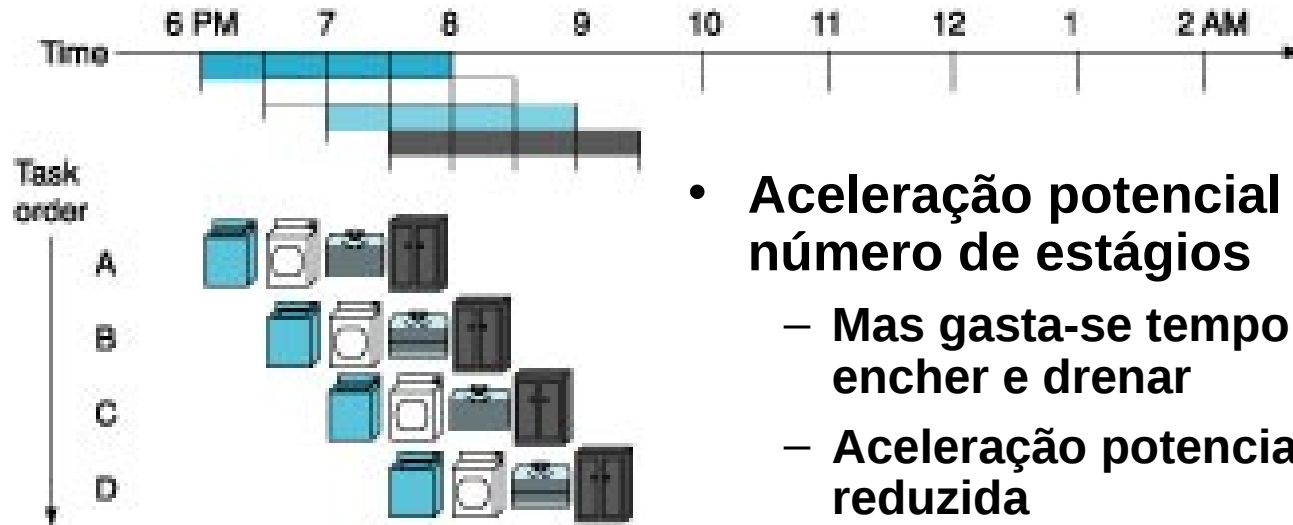
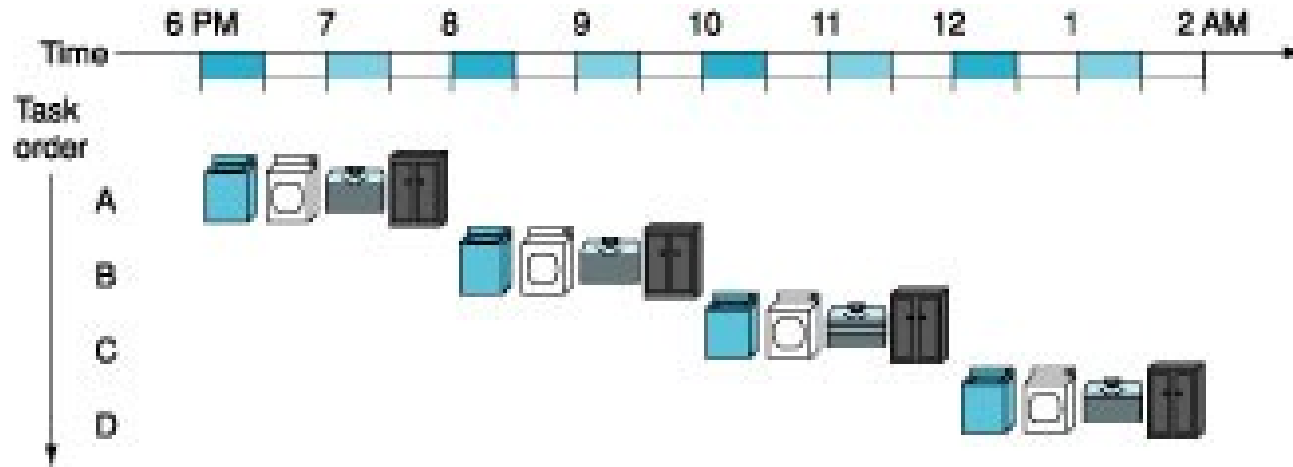
- Taxa de tarefas no tempo (throughput) aumenta
 - Mais pacotes ficam prontos num intervalo

Pipelining: uma analogia



- **Múltiplas tarefas simultâneas**
 - **Requer recursos distintos**

Pipelining: uma analogia



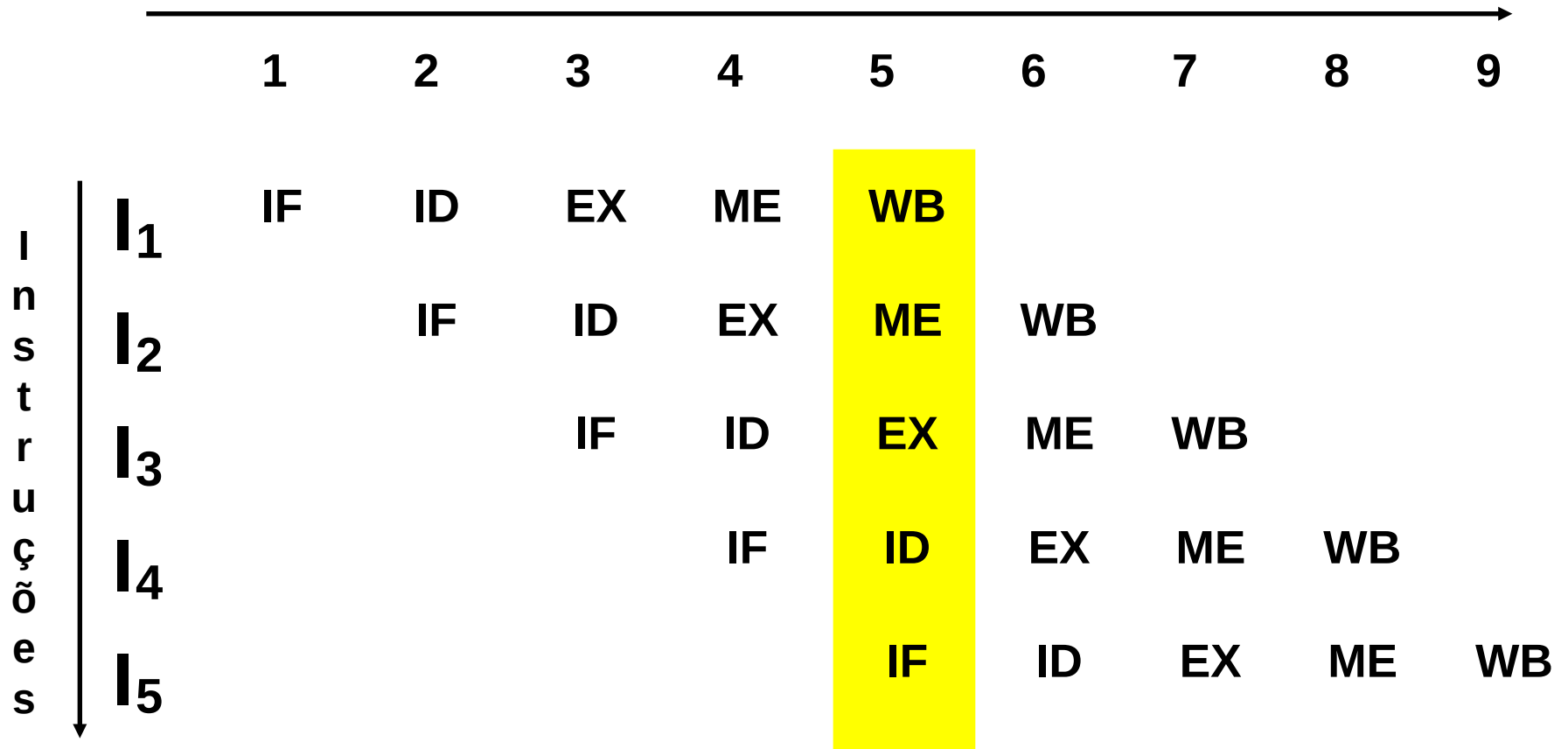
- **Aceleração potencial = número de estágios**
 - Mas gasta-se tempo para encher e drenar
 - Aceleração potencial é reduzida

Pipelining em CPUs

- **Mesmos princípios gerais se aplicam**
 - Etapas são as fases de execução de instrução
- **Instruções MIPS**
 - 1. Buscar a instrução na memória (**IF**)
 - 2. Decodificação; leitura de registradores (**ID**)
 - 3. Execução: operação/cálculo de endereço (**EX**)
 - 4. Acesso a operando em memória (**ME**)
 - 5. Escrita do resultado em registrador (**WB**)

Pipeline: visualização

Tempo (ciclos de relógio)

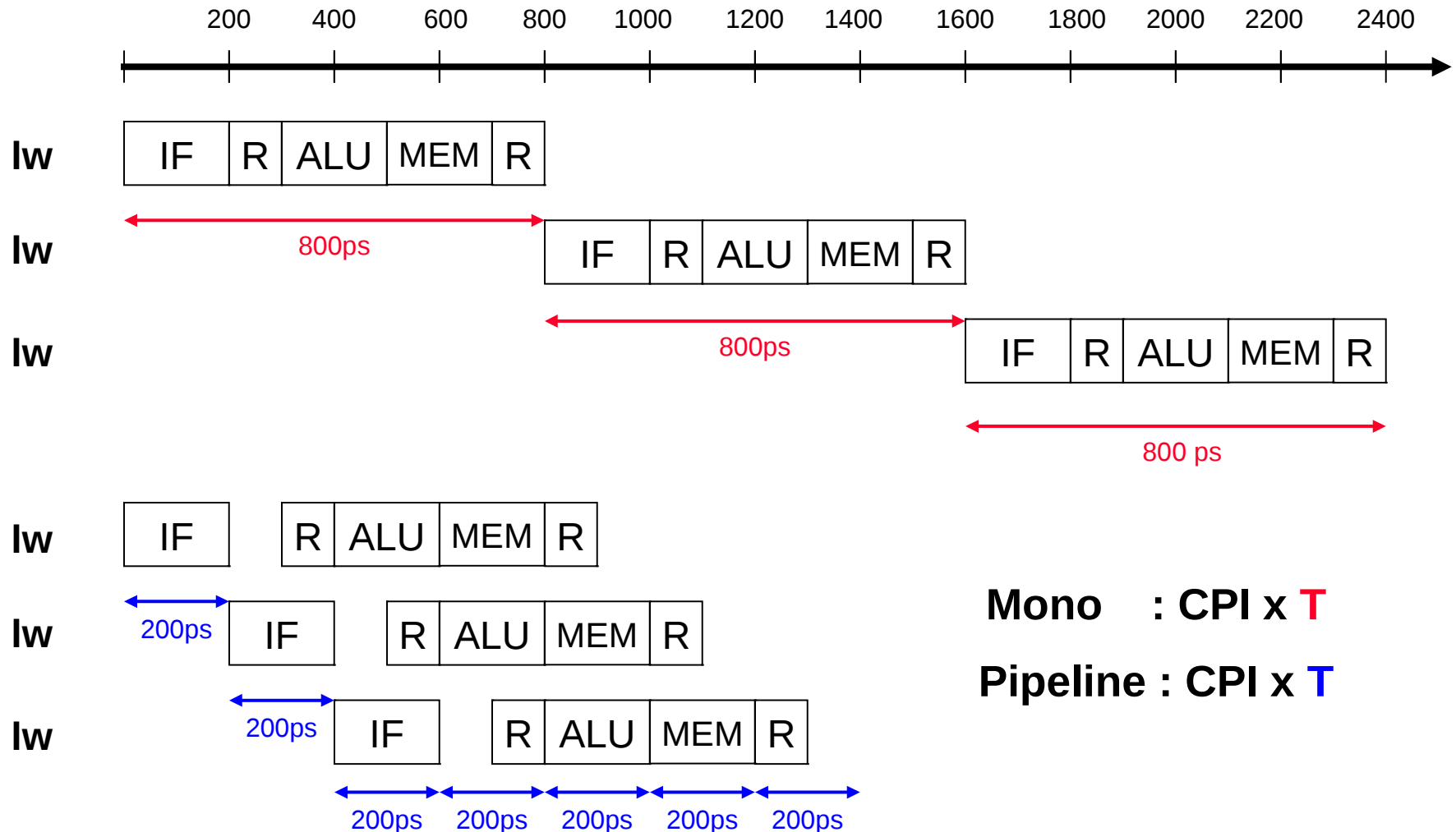


Criando um pipeline

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R	200 ps	100 ps	200 ps		100 ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps

- **Datapath monociclo:** cada instrução dura um ciclo $\Rightarrow T = 800$ ps (pior caso)
- **Datapath com pipeline:** cada estágio dura um ciclo $\Rightarrow T = 200$ ps (pior caso)

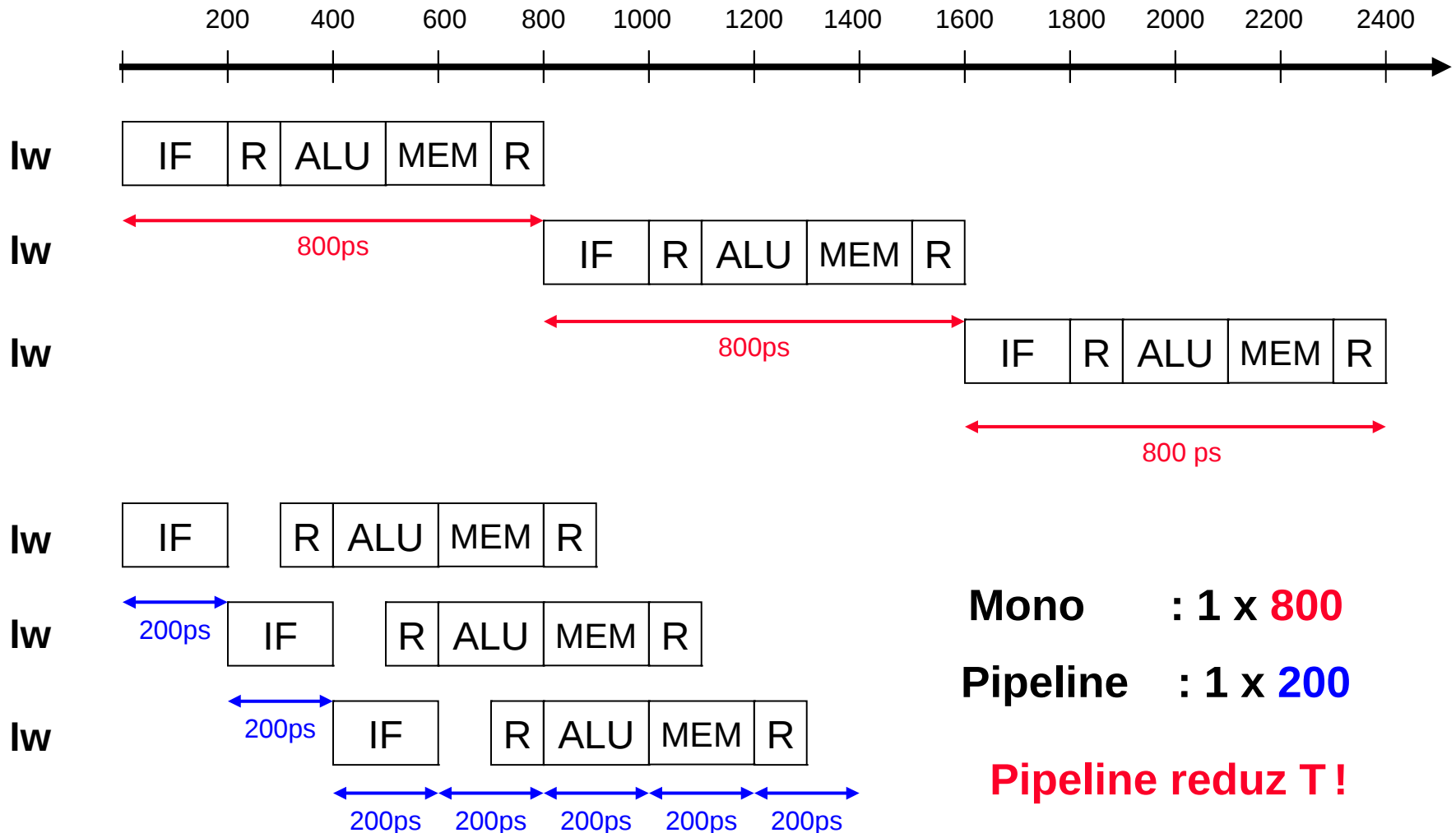
Monociclo x Pipeline



Mono : CPI x T

Pipeline : CPI x T

Monociclo x Pipeline



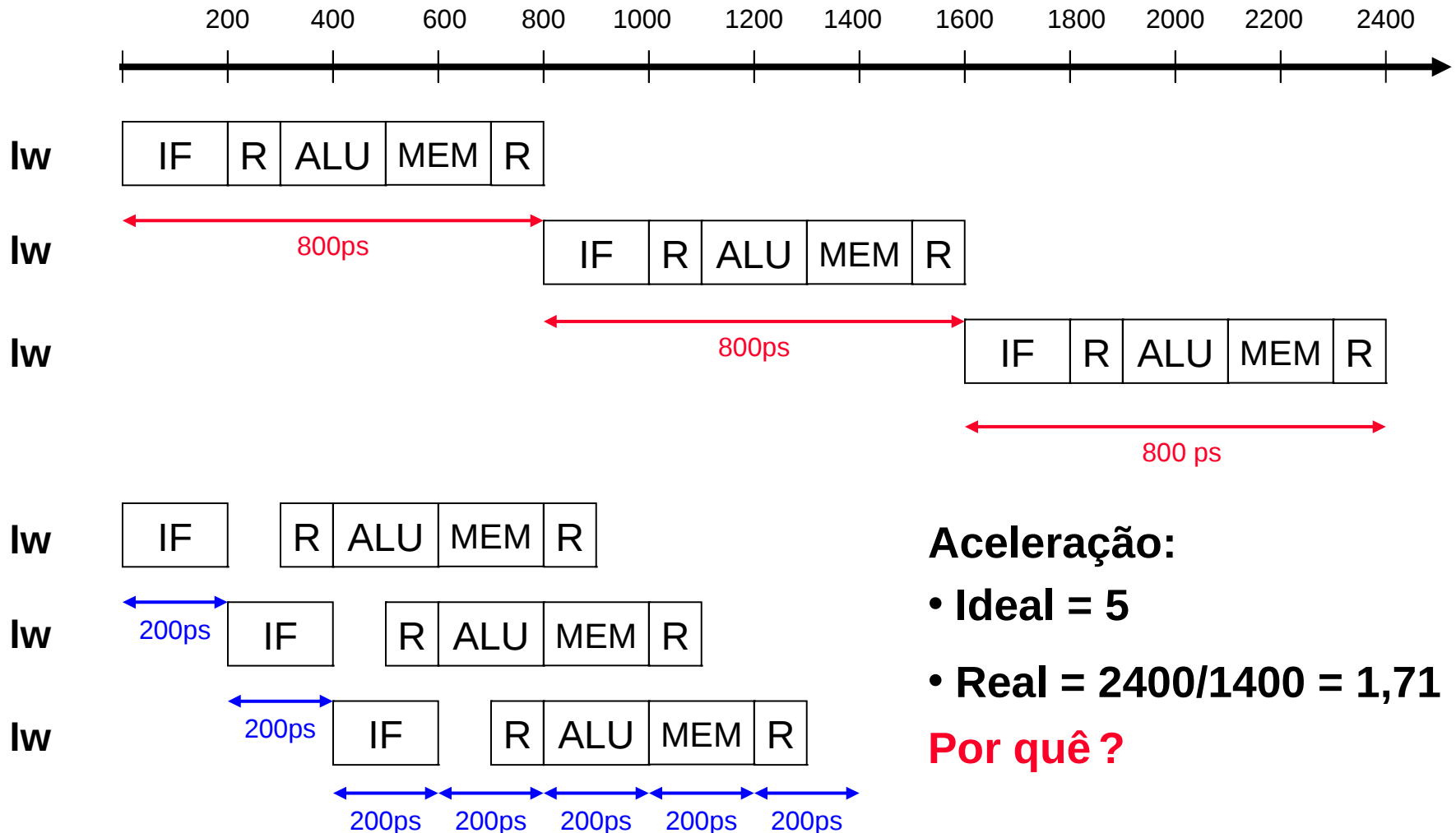
Expectativa

- **Tempo entre instruções:** (idealmente)

$$\frac{\text{tempo entre instruções sem pipeline}}{\text{número de estágios}}$$

- **Aceleração ideal = número de estágios**

Revisitando o exemplo



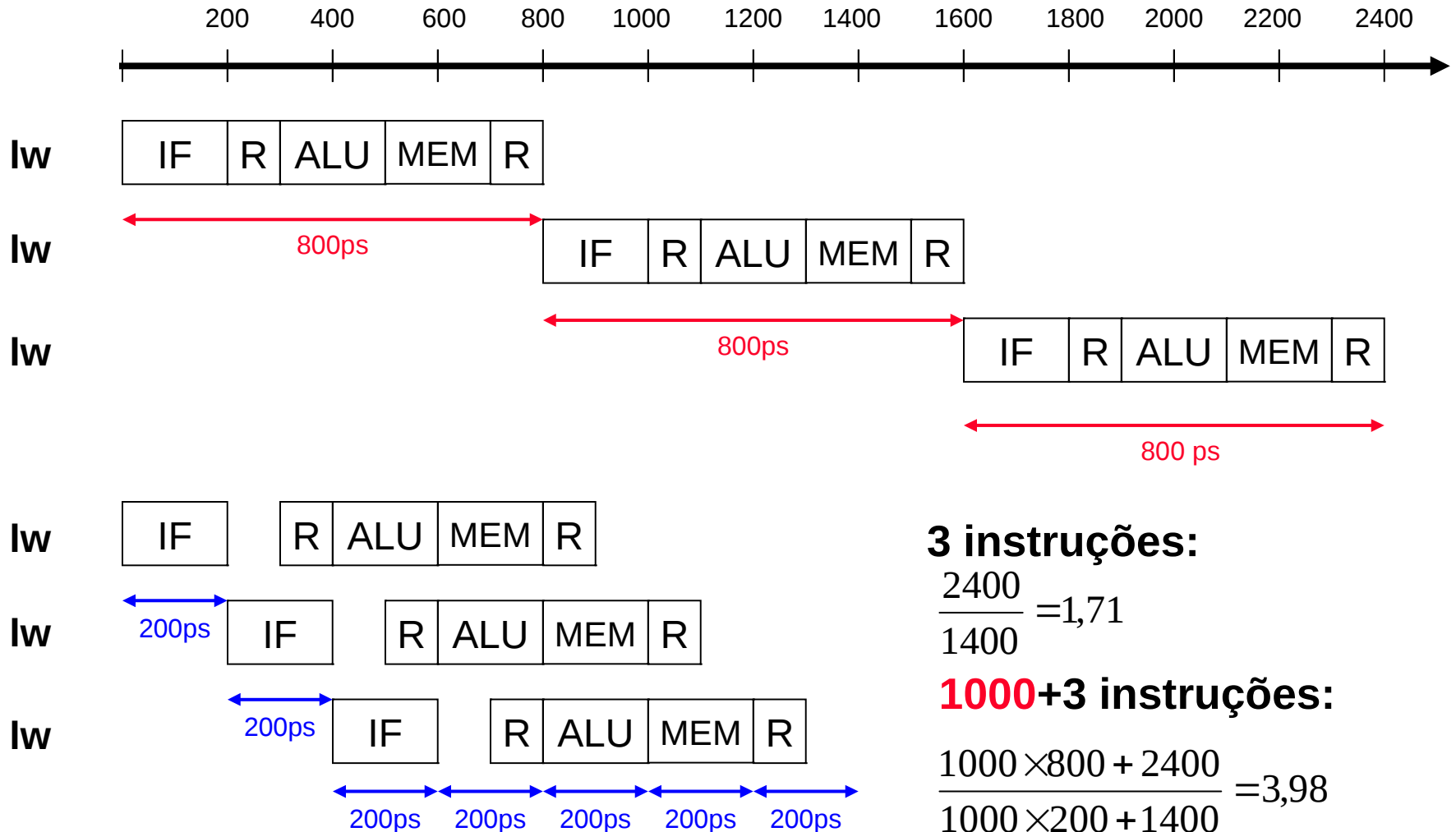
Aceleração:

- Ideal = 5

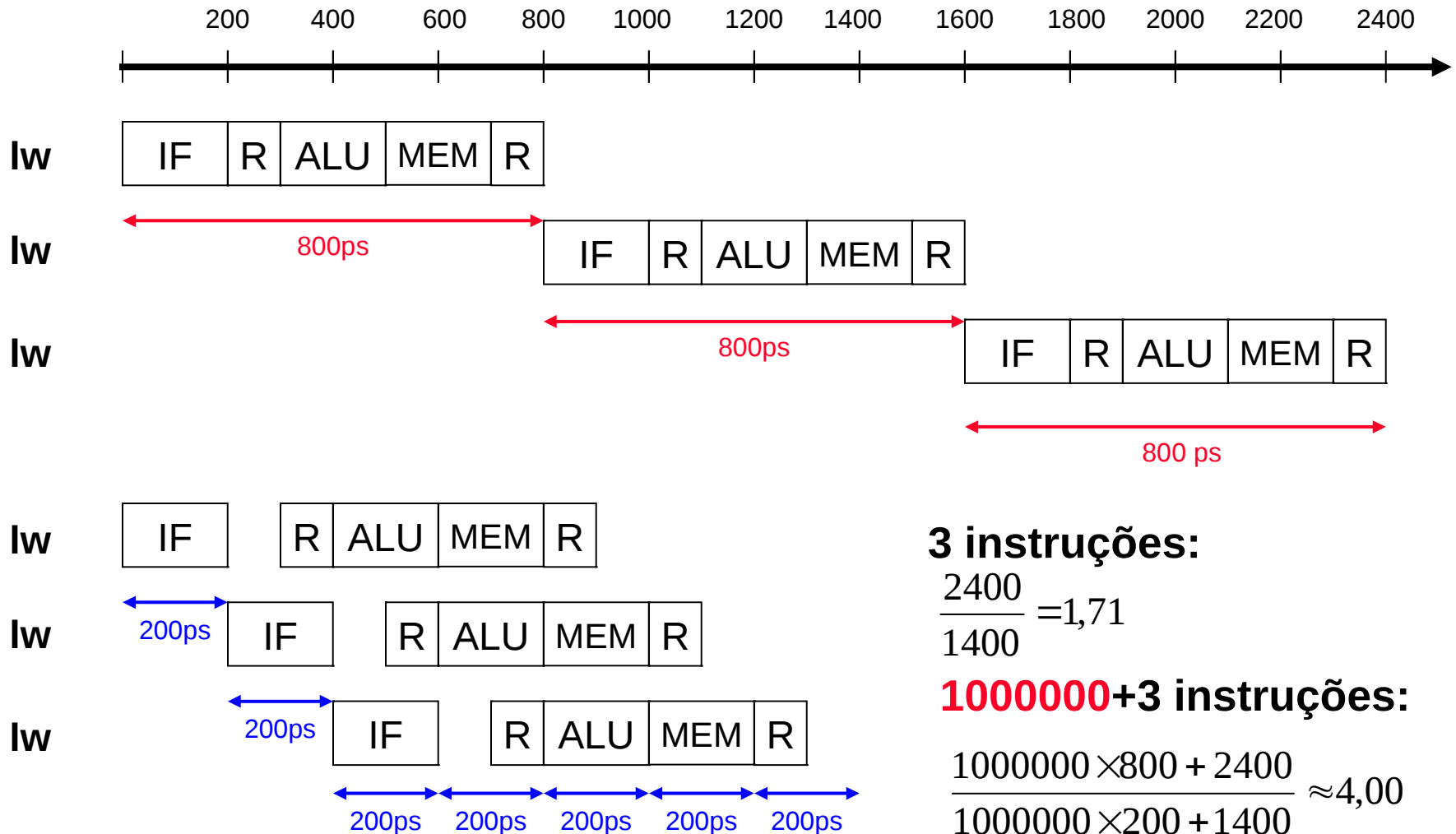
- Real = $2400/1400 = 1,71$

Por quê ?

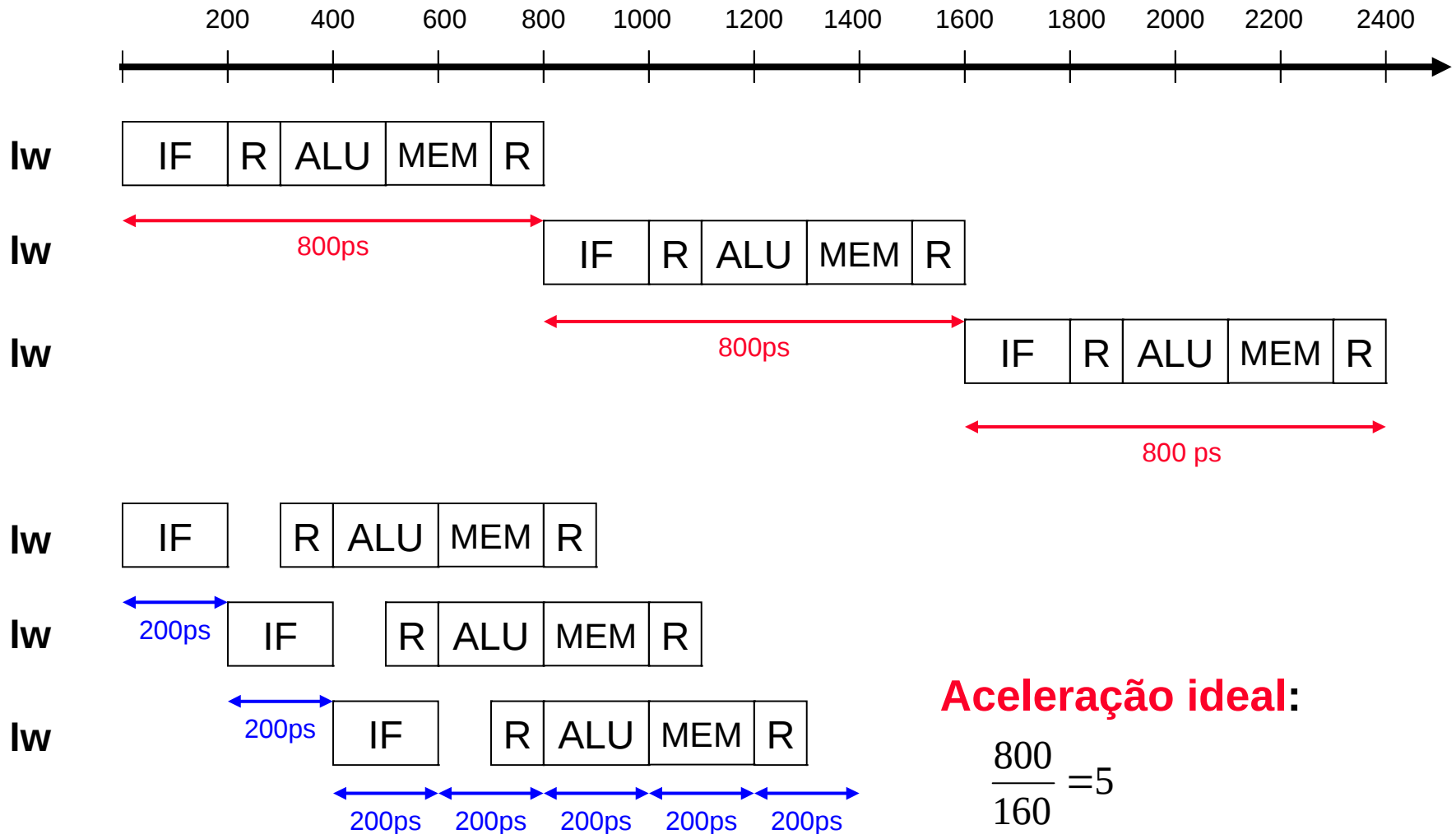
Revisitando o exemplo



Revisitando o exemplo



Revisitando o exemplo



Aceleração ideal:

$$\frac{800}{160} = 5$$

Conclusões

- **Comparado ao datapath monociclo**
 - Pipeline reduz o T
 - » Permite operação em frequências maiores
- **Aceleração ideal**
 - Seria atingida somente se estágios balanceados
- **Aceleração real máxima**
 - Atingida para um grande número de instruções

Conclusões

- **Pipeline melhora desempenho**
 - Aumentando o **throughput**
 - Em vez de diminuir o tempo da instrução individual
- **Programas reais**
 - Executam bilhões de instruções
 - Throughput é dominante
- **Pipelining é técnica de implementação**
 - **Invisível no ISA !**
 - » Exceto por instruções do tipo “delayed branch”

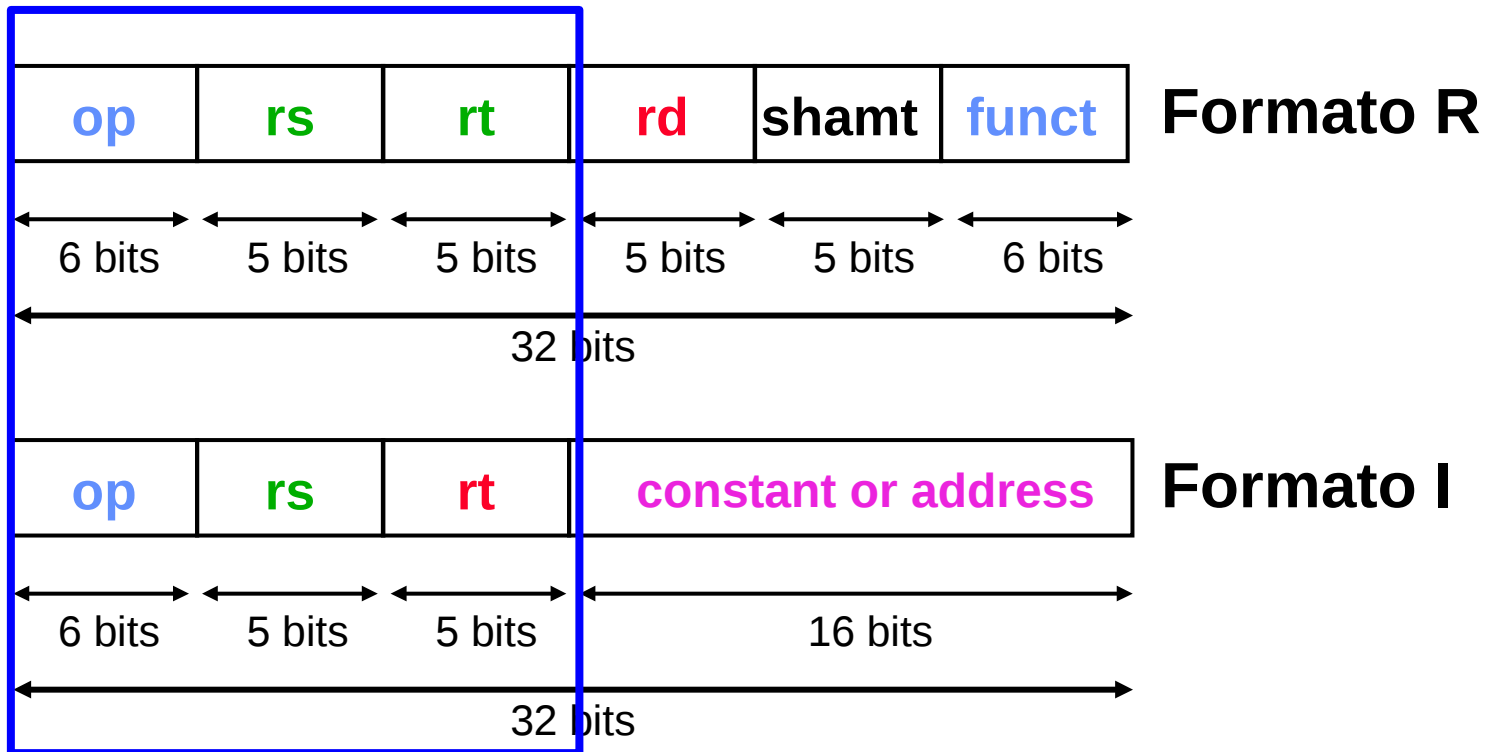
Projeto de ISA para pipelining

- Instruções com comprimento fixo
 - Busca no 1º estágio, decodificação no 2º.
 - Exemplo:
I1: IF ID EX ME WB
I2: IF ID EX ME WB
 - Contra-exemplo:
 - » No x86 o comprimento varia entre 1 e 17 bytes
 - » IF consiste em buscar de 1 a 5 palavras

I1: IF IF IF IF IF ID EX
I2: IF IF ...

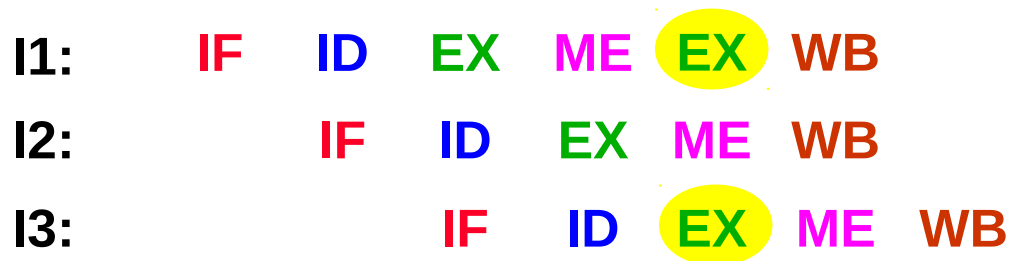
Projeto de ISA para pipelining

- Instruções com “formato fixo”
 - Registrador fonte sempre na mesma posição
 - » leitura de registrador // decodificação (único estágio)



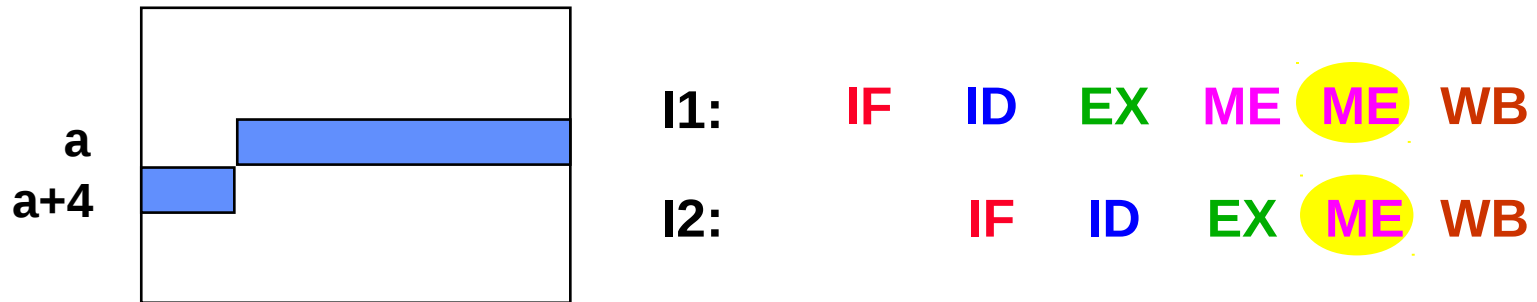
Projeto de ISA para pipelining

- Máquina load/store
 - Usa estágio EX p/ calcular endereço e acessa memória no próximo estágio
 - » Senão 3 estágios: endereço, memória, execução
 - Contra-exemplo:
 - » I1: add \$s1, \$s2, K(\$s3)
 - » I2 e I3 : adds c/ todos operandos em registrador



Projeto de ISA para pipelining

- Alinhamento de operandos na memória
 - Dados acessados em único ciclo por estágio
 - Contra-exemplo:



Hazards


- A próxima instrução **não** pode iniciar execução no próximo ciclo de relógio.
- Stall
 - **Pausa** do pipeline até que se resolva empecilho
 - Uma “bolha” é inserida no pipeline

Hazards de dados

- A primeira instrução produz um valor que a segunda consome, **MAS** o valor **não** está disponível para consumo em tempo hábil.
- Exemplo:

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3



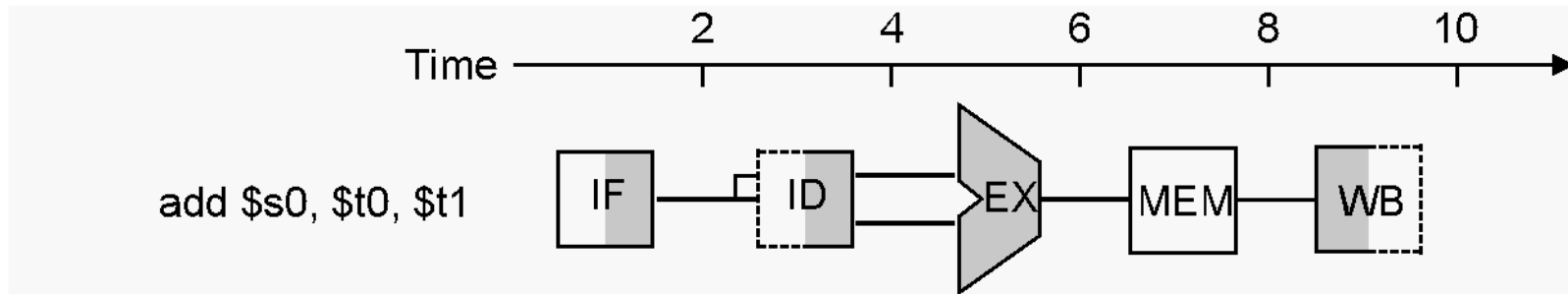
Hazards de dados

- Quando está disponível o valor de \$s0 produzido por “add” ?

- Exemplo:**

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3



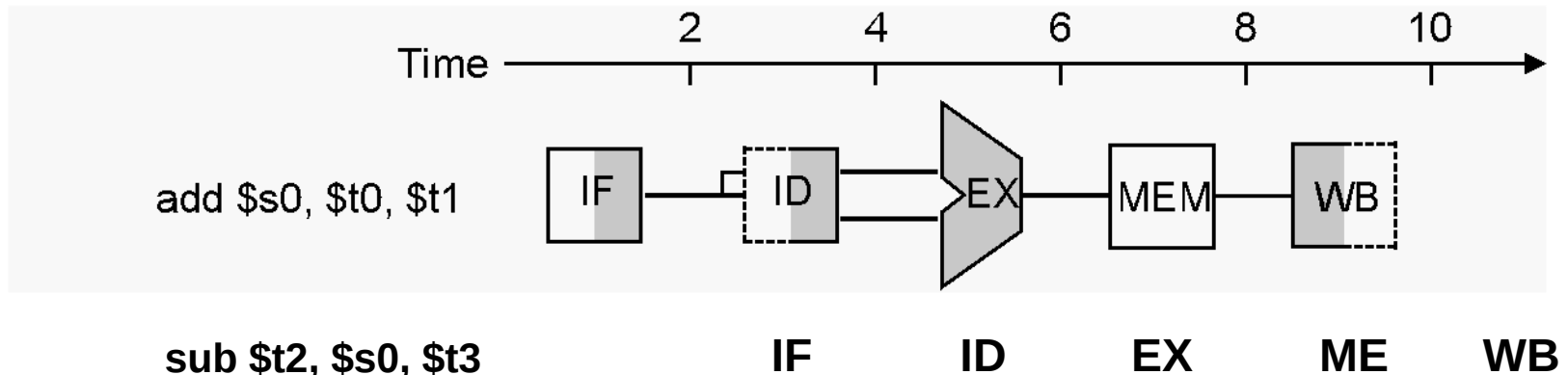
Hazards de dados

- Quando o valor de \$s0 precisaria estar disponível para ser lido por “sub” ?

- Exemplo:

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3



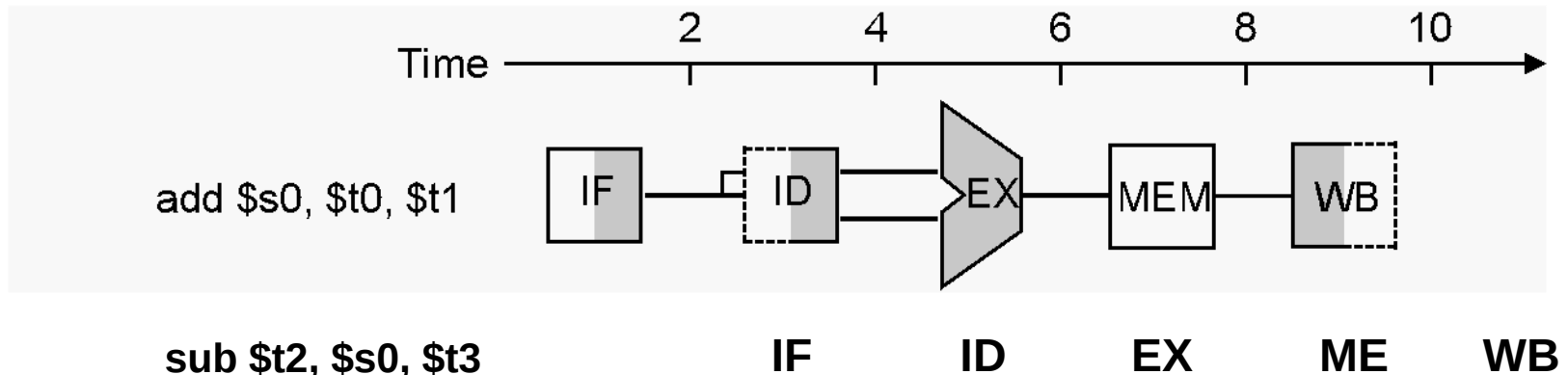
Hazards de dados

- Quantos ciclos de pausa no pipeline para que o valor correto de \$s0 seja consumido ?

- Exemplo:

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3



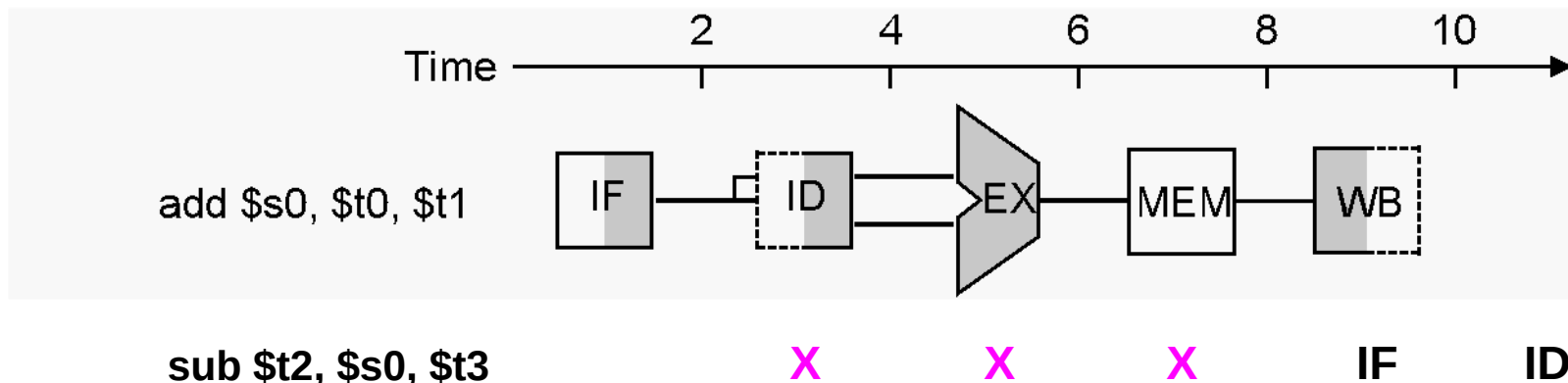
Hazards de dados

- Quantos ciclos de pausa no pipeline para que o valor correto de \$s0 seja consumido ?

- Exemplo:

add \$s0, \$t0, \$t1

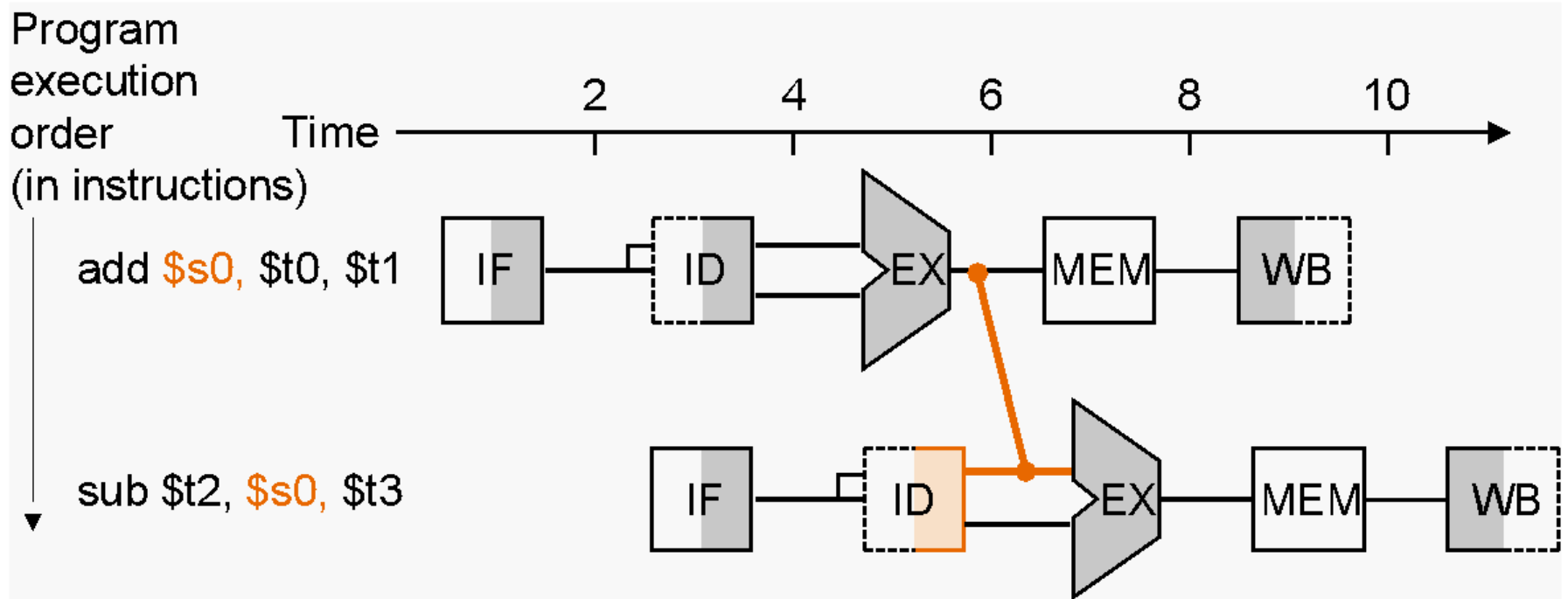
sub \$t2, \$s0, \$t3



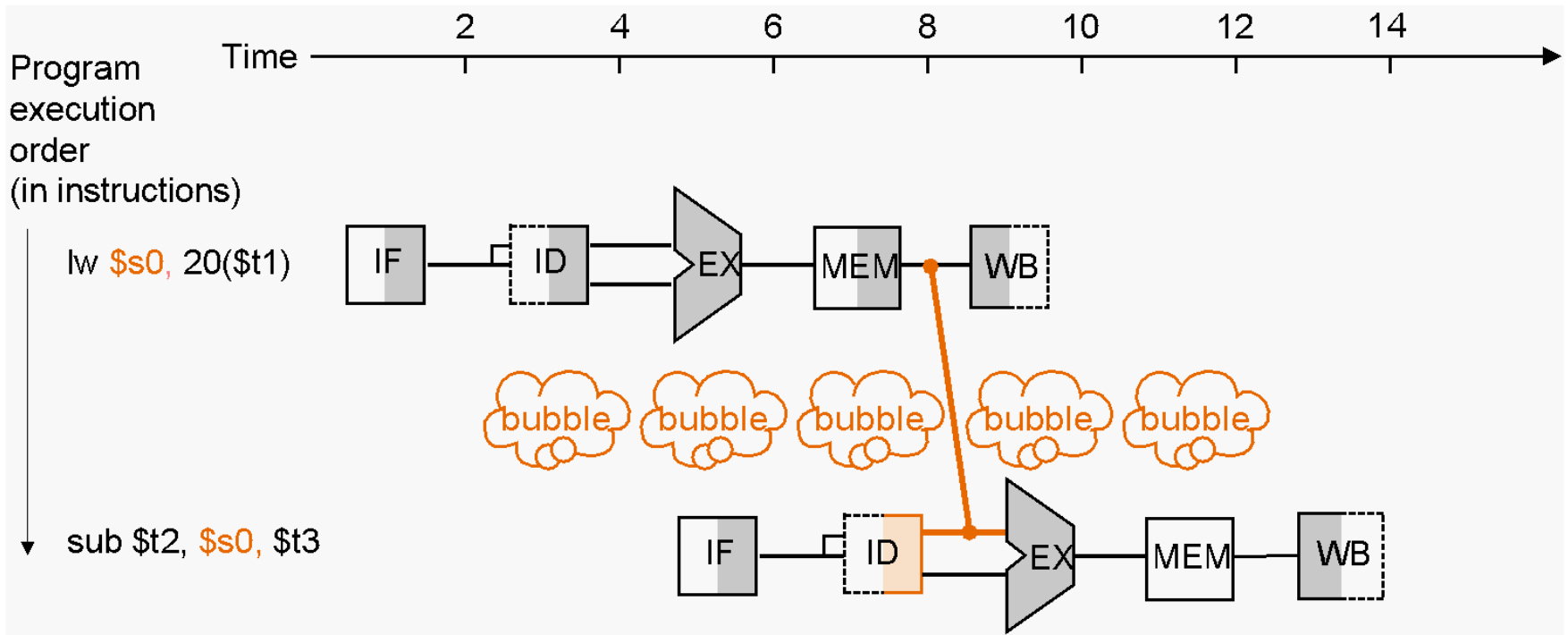
Hazards de dados

- Não é preciso esperar até a instrução produtora ser completada ...
- Para disponibilizar o valor produzido para a instrução consumidora
- Redução do impacto no desempenho
 - Técnica: “Forwarding” ou “Bypassing”
 - » Roteamento direto entre o produtor e o consumidor

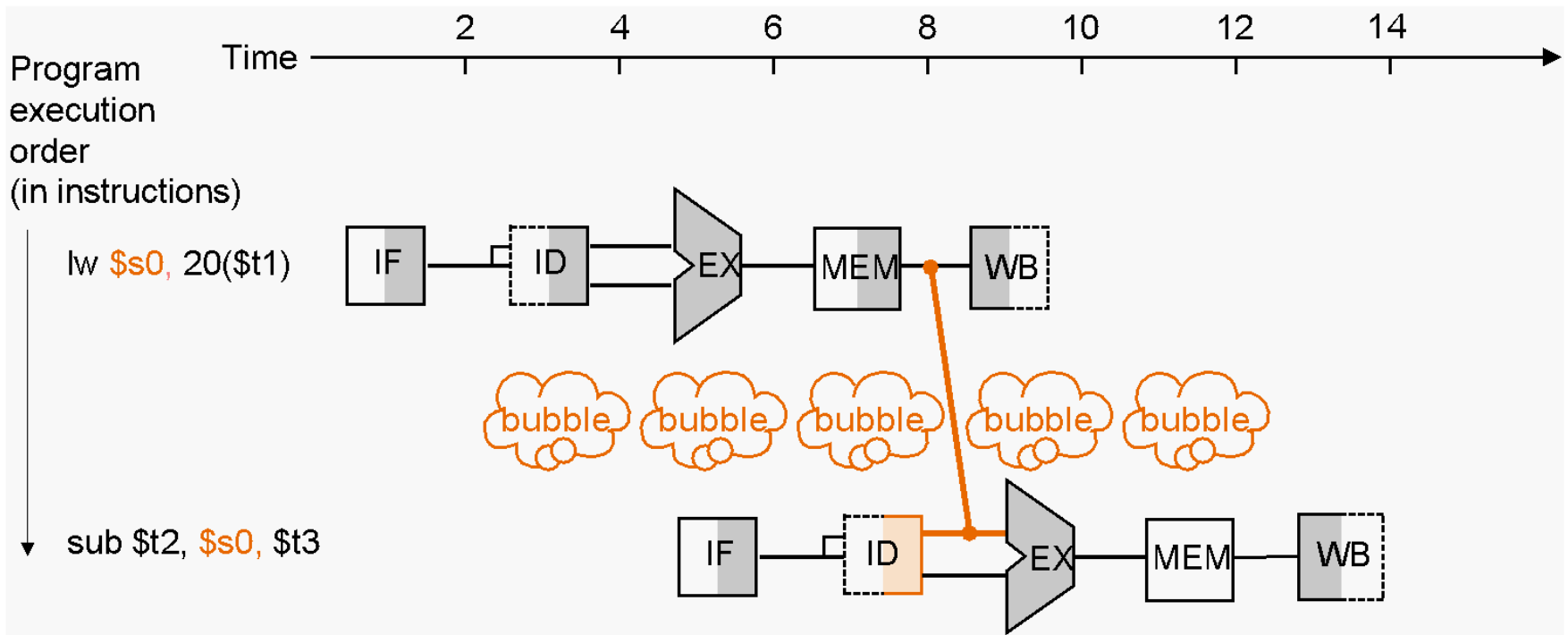
Forwarding: R → R



Forwarding: lw → R

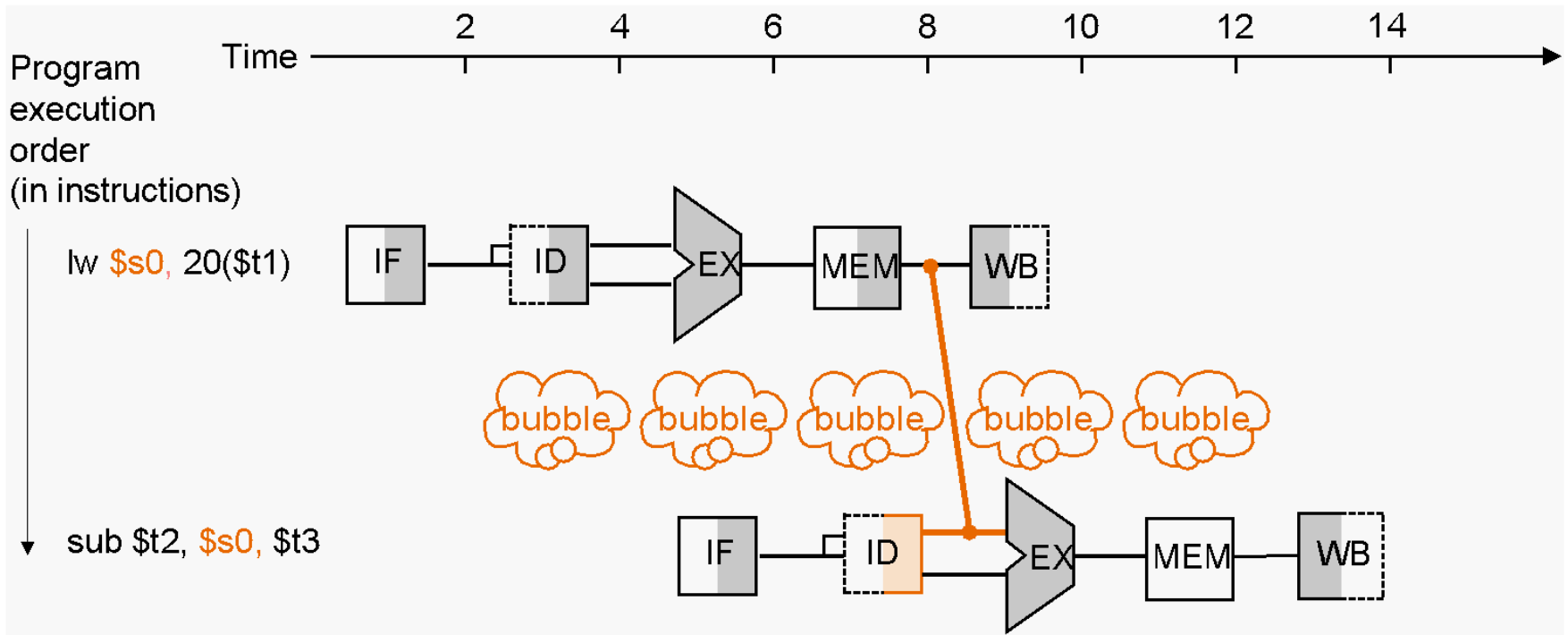


Forwarding: lw → R



Uma instrução R dependente não
pode iniciar no ciclo seguinte ao load

Forwarding: lw → R



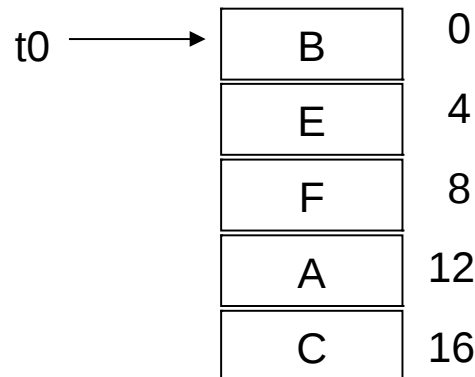
Por que não iniciar aqui uma instrução independente ?

Escalonamento de código

- “Code scheduling”
 - Reordenamento das instruções para preencher “bolhas” com instruções independentes
- Exemplo:
 - Variáveis A, B, C, E e F na memória
 - Endereçáveis a partir de um registrador-base \$t0

A = B + E;

C = B + F;



Código original

Esta dependência não causa pausa no pipeline porque lw escreve em \$t1 antes de add ler \$t1

lw **\$t1, 0(\$t0)**

lw **\$t2, 4(\$t0)**

add **\$t3, \$t1, \$t2**

sw **\$t3, 12(\$t0)**

lw **\$t4, 8(\$t0)**

add **\$t5, \$t1, \$t4**

sw **\$t5, 16(\$t0)**

Código original

lw **\$t1**, 0(\$t0)

lw \$t2, 4(\$t0)

add \$t3, **\$t1**, \$t2

Esta dependência não causa pausa no
pipeline devido ao forwarding

sw \$t3, 12(\$t0)

lw \$t4, 8(\$t0)

add \$t5, \$t1, \$t4

sw \$t5, 16(\$t0)

Código original

lw \$t1, 0(\$t0)

lw **\$t2**, 4(\$t0)

add \$t3, \$t1, **\$t2**

sw \$t3, 12(\$t0)

lw **\$t4**, 8(\$t0)

add \$t5, \$t1, **\$t4**

sw \$t5, 16(\$t0)

Estas dependências causam pausa no pipeline apesar do forwarding

Código original



Código original

lw \$t1, 0(\$t0)

lw **\$t2**, 4(\$t0)

add \$t3, \$t1, **\$t2**

sw \$t3, 12(\$t0)

lw **\$t4**, 8(\$t0)

add \$t5, \$t1, **\$t4**

sw \$t5, 16(\$t0)

Otimização: Como eliminar os ciclos de pausa ?

Solução: espaçar as instruções produtoras das consumidoras, intercalando instruções independentes

Método: Reordenamento do código original, preservando as dependências de dados

Código original

lw \$t1, 0(\$t0)

lw **\$t2**, 4(\$t0)

add \$t3, \$t1, **\$t2**

sw \$t3, 12(\$t0)

lw **\$t4**, 8(\$t0)

add \$t5, \$t1, **\$t4**

sw \$t5, 16(\$t0)

Otimização: Como eliminar os ciclos de pausa ?

Solução: espaçar as instruções produtoras das consumidoras, intercalando instruções independentes

Método: Reordenamento do código original, preservando as dependências de dados

Código original

lw \$t1, 0(\$t0)

lw \$t2, 4(\$t0)

add \$t3, \$t1, \$t2

sw \$t3, 12(\$t0)

lw \$t4, 8(\$t0)

add \$t5, \$t1, \$t4

sw \$t5, 16(\$t0)


Código escalonado

lw \$t1, 0(\$t0)

lw \$t2, 4(\$t0)

add \$t3, \$t1, \$t2

sw \$t3, 12(\$t0)



add \$t5, \$t1, \$t4

sw \$t5, 16(\$t0)

Código escalonado

lw \$t1, 0(\$t0)

lw \$t2, 4(\$t0)



add \$t3, \$t1, \$t2

sw \$t3, 12(\$t0)

add \$t5, \$t1, \$t4

sw \$t5, 16(\$t0)

Código escalonado

lw \$t1, 0(\$t0)

lw \$t2, 4(\$t0)

Nenhuma dependência causa pausa no pipeline

lw \$t4, 8(\$t0)

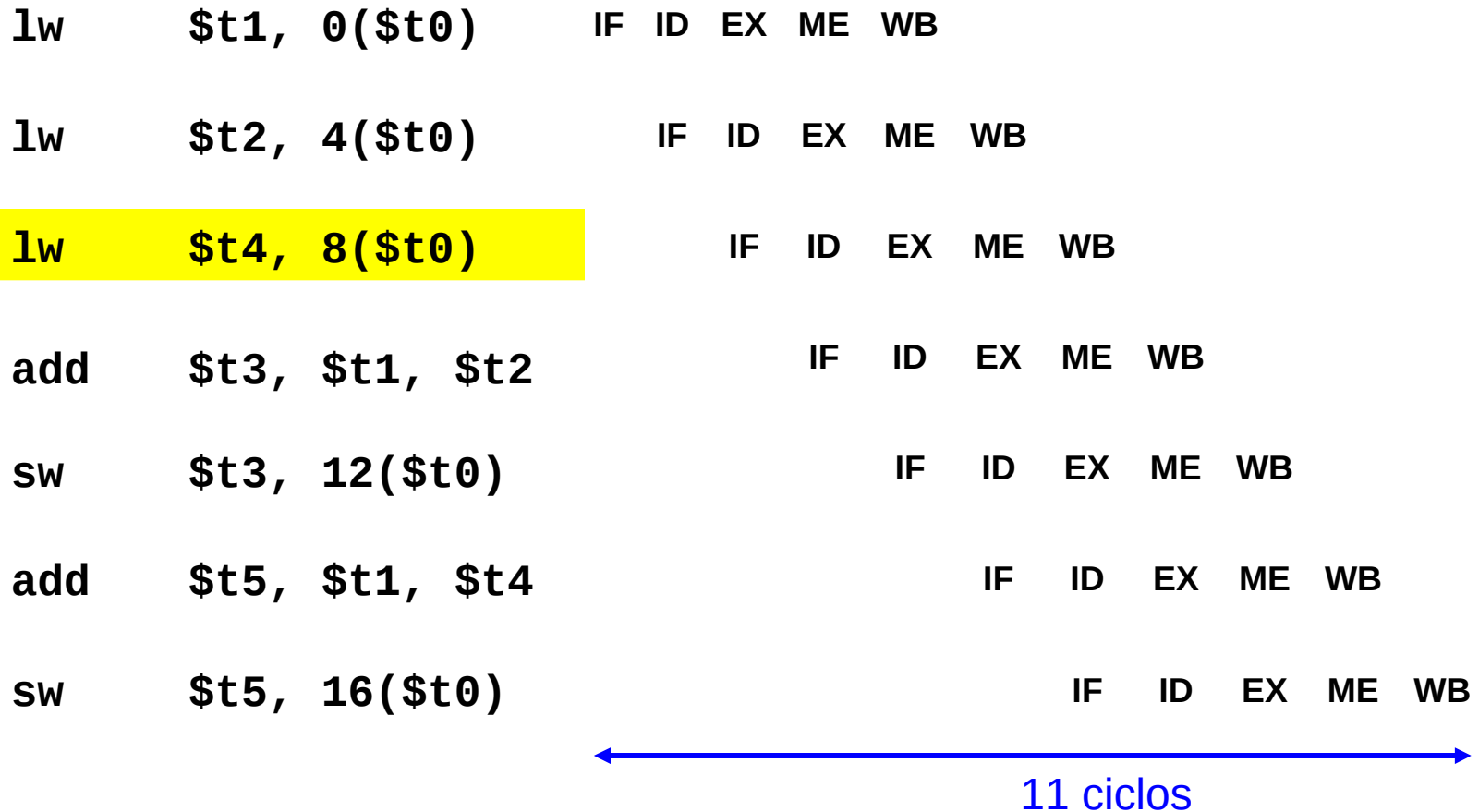
add \$t3, \$t1, \$t2

sw \$t3, 12(\$t0)

add \$t5, \$t1, \$t4

sw \$t5, 16(\$t0)

Código escalonado



Conclusão

- **Conhecimento da estrutura do pipeline**
 - Estágios e caminhos de forwarding
 - Determina a latência entre instruções
 - » Número de ciclos que uma instrução precisa esperar para executar sem pausa
 - Disponibilizadas no manual do processador
- **Escalonamento de código**
 - Uma importante otimização em compiladores
 - » Ex. gcc: **-fschedule-insns (-O2, -O3, -Os)**
 - Melhora o CPI
- **A noção de compilador-otimizador**