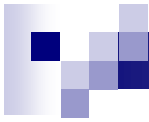


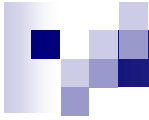


Pipeline



Visão Geral de *Pipelining*

- Instruções MIPS têm mesmo tamanho
 - Mais fácil buscar instruções no primeiro estágio e decodificar no segundo estágio
 - IA-32
 - Instruções variam de 1 *byte* a 17 *bytes*
 - Instruções traduzidas em microoperações
 - Pentium-4 usa pipeline de microoperações
- MIPS tem poucos formatos de instrução
 - Registrador origem na mesma posição
 - Segundo estágio pode ler banco de registradores ao mesmo tempo em que hardware está determinando que tipo de instrução foi lida



Hazards

- *Hazards*: situação em que próxima instrução não pode ser executada no ciclo de *clock* seguinte. Três tipos:
 - *Hazards* estruturais
 - *Hazards* de dados
 - *Hazards* de controle



Hazards

■ *Hazards* Estruturais

- Hardware não pode admitir a combinação de instruções que queremos executar no mesmo ciclo de *clock*

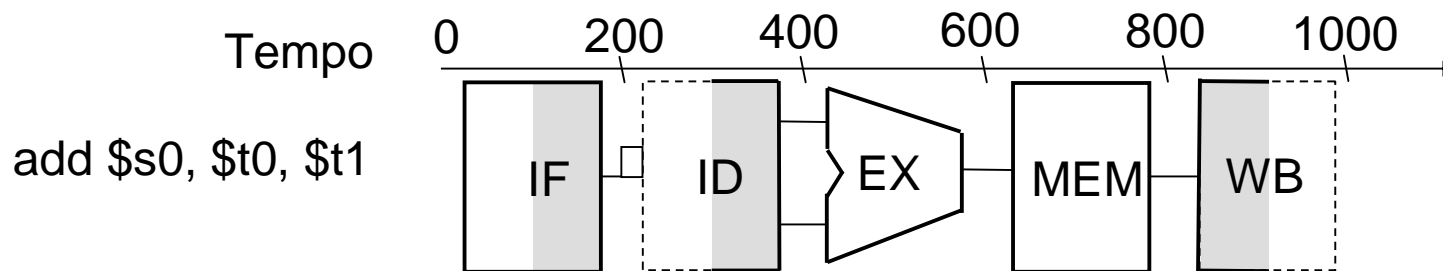
■ *Hazards* de Dados

- Pipeline precisa ser interrompido porque os dados para executar a instrução ainda não estão disponíveis
 - Ex.: **add** \$s0, \$t0, \$t1
 sub \$t2, \$s0, \$t3
 - **add** não escreve resultado até o quinto estágio
- Acontecem com muita frequência

Hazards

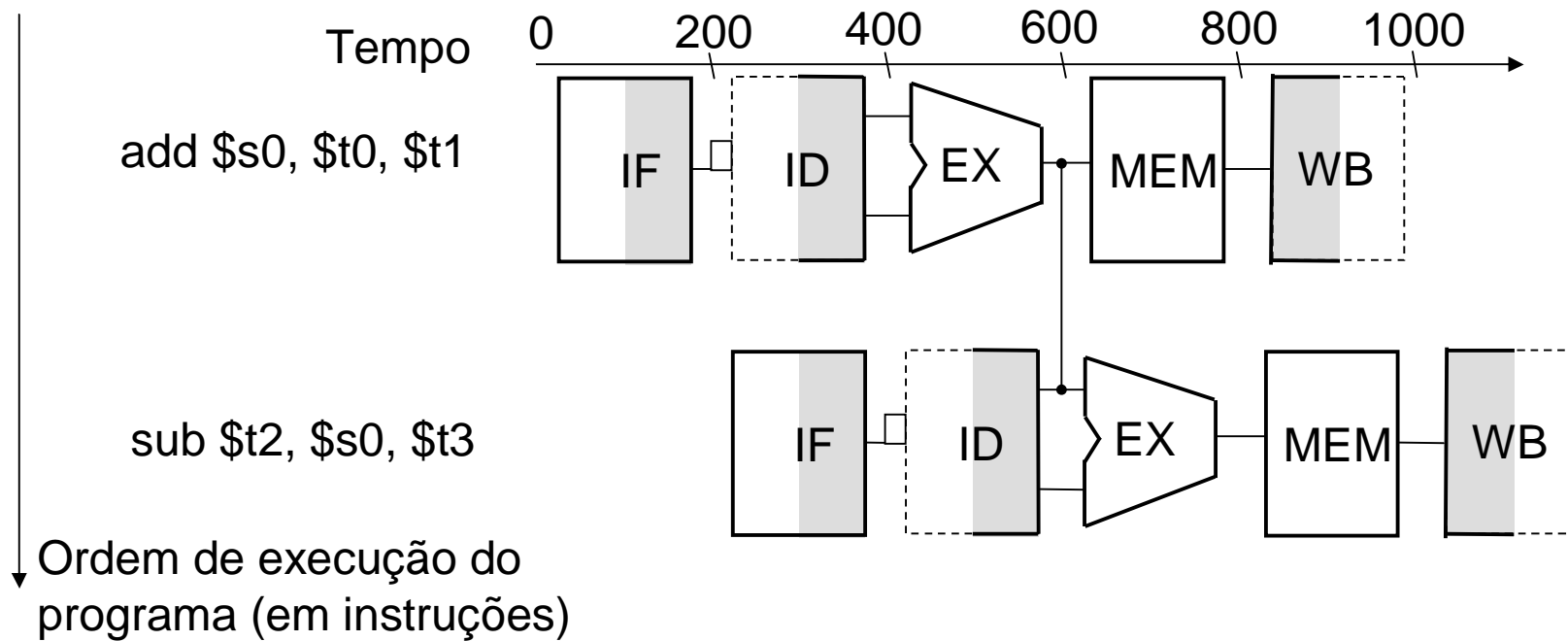
■ Solução

- Não precisamos esperar instrução terminar
- Hardware adicionado para ter o item que falta antes do previsto: *forwarding* ou *bypassing*



IF: Busca instrução, ID: decodifica/lê registrador, EX: execução, MEM: memória
WB: Escreve resultado. Sombreado lado direito: Leitura, esquerdo: escrita. Sem
Sombreado: não usado

Hazards

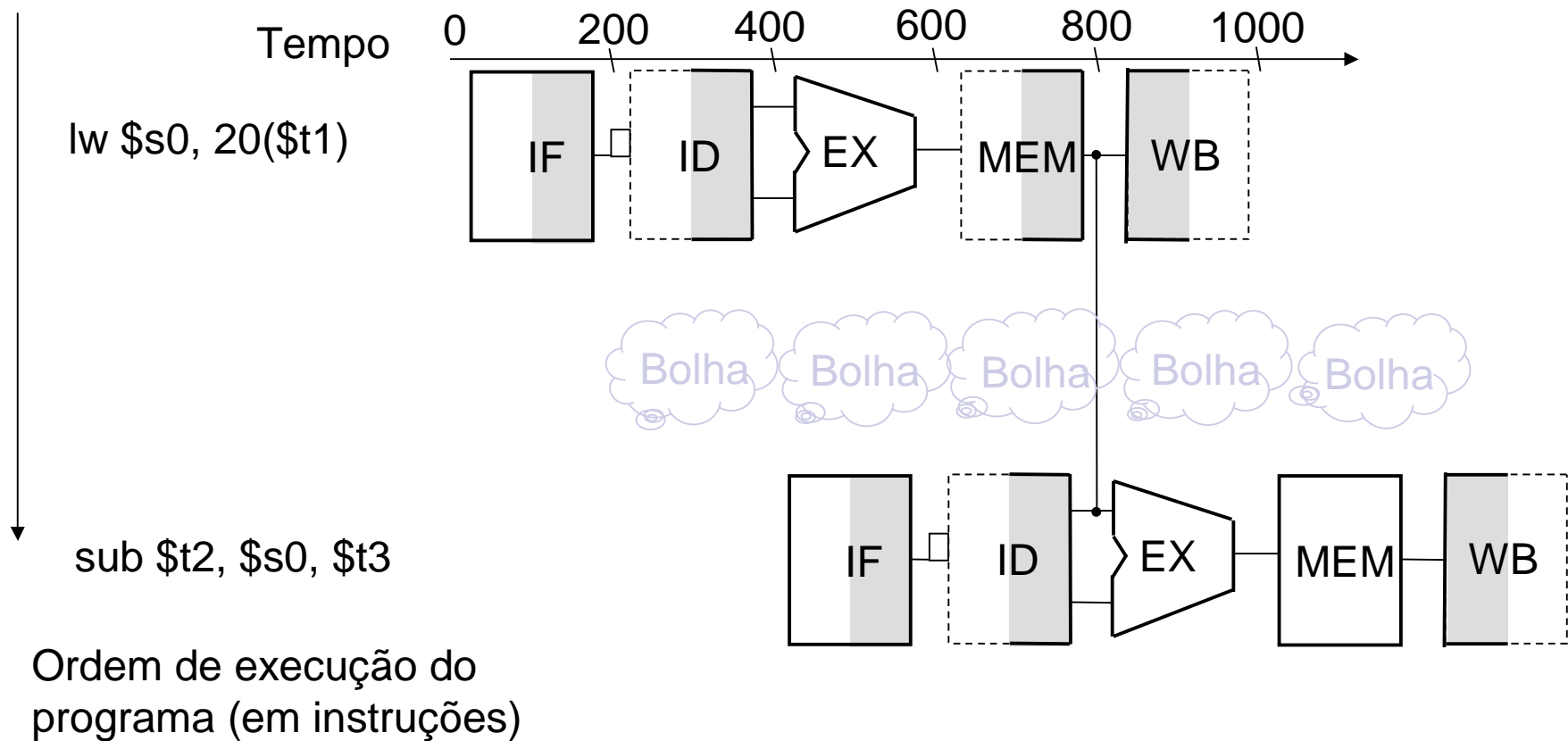


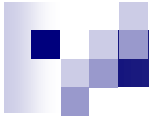


Hazards

- *Forwarding* só válido se estágio destino estiver mais adiante no tempo
- Não pode impedir todos os *stalls* no pipeline
 - Ex: **load** de \$s0 ao invés de **add**

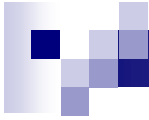
Hazards





Hazards

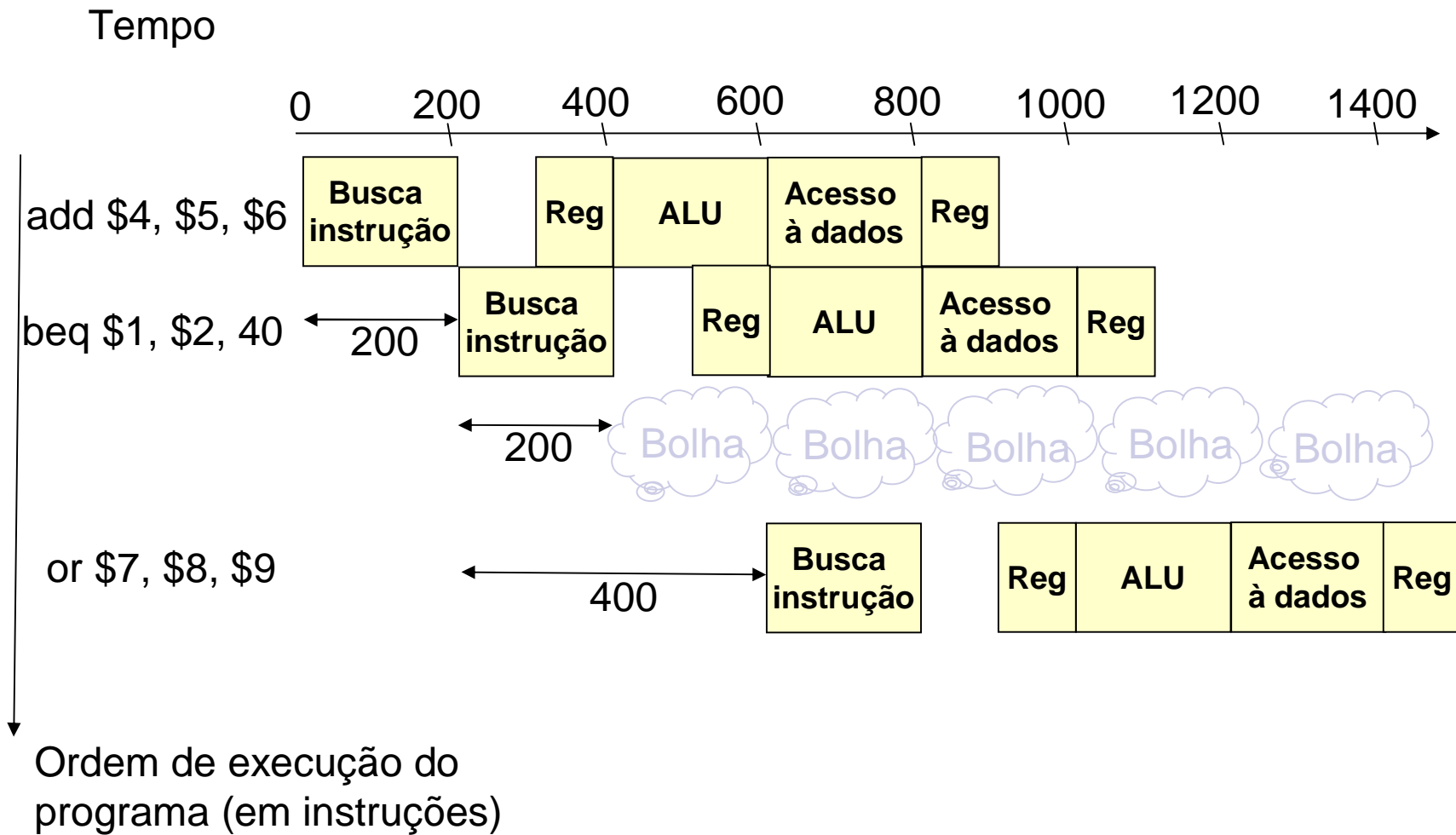
- *Hazard* de Controle (ou desvio):
necessidade de tomar decisão baseado
nos resultados de uma instrução enquanto
outras então sendo executadas
 - Ex: operação de desvio
- Três opções

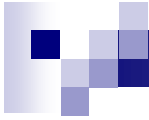


Hazards

- Primeira Opção: Causar *stall* no pipeline imediatamente após buscarmos um desvio
 - Esperar até que o pipeline determine resultado do desvio
 - Endereço então disponível para determinar próxima instrução
- Supondo hardware extra para testar registradores, calcular endereço do destino e atualizar PC no segundo estágio...

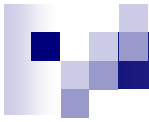
Hazards





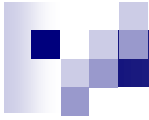
Hazards

- Segunda Opção: Previsão para tratar desvios
 - Técnica simples: prever que os desvios não serão tomados



Hazards

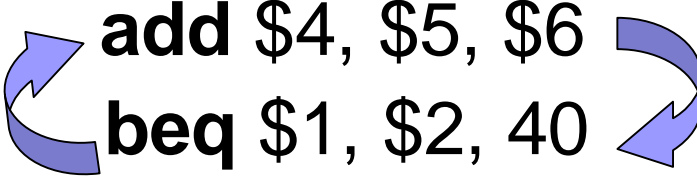
- Versão mais sofisticada: alguns desvios previstos como tomados e outros como não tomados
 - Ex.: Assumir que *loop* sempre volta para trás
- Duas versões anteriores estáticas / estereotipadas
- Previsores dinâmicos
 - Escolhas dependem do comportamento de cada desvio
 - Podem ser alteradas durante a vida de um programa
 - Mantém histórico, usando comportamento passado para prever futuro
 - Previsão pode ser superior a 90%

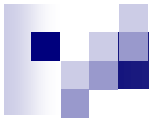


Hazards

- Quando pipeline erra, controle terá de garantir que as instruções após o desvio errado não tenham efeito
 - Pipeline reiniciado a partir do endereço de desvio apropriado
 - Pipelines mais longos aumentam o problema (aumentam o custo do erro de previsão)

Hazards

- Terceira opção: Desvio adiado
 - Sempre executa a próxima instrução sequencial, com desvio ocorrendo após esse atraso de uma instrução
 - Instrução não afetada pelo desvio
 -  **add** \$4, \$5, \$6
beq \$1, \$2, 40
 - Útil quando desvios são curtos



Hazards

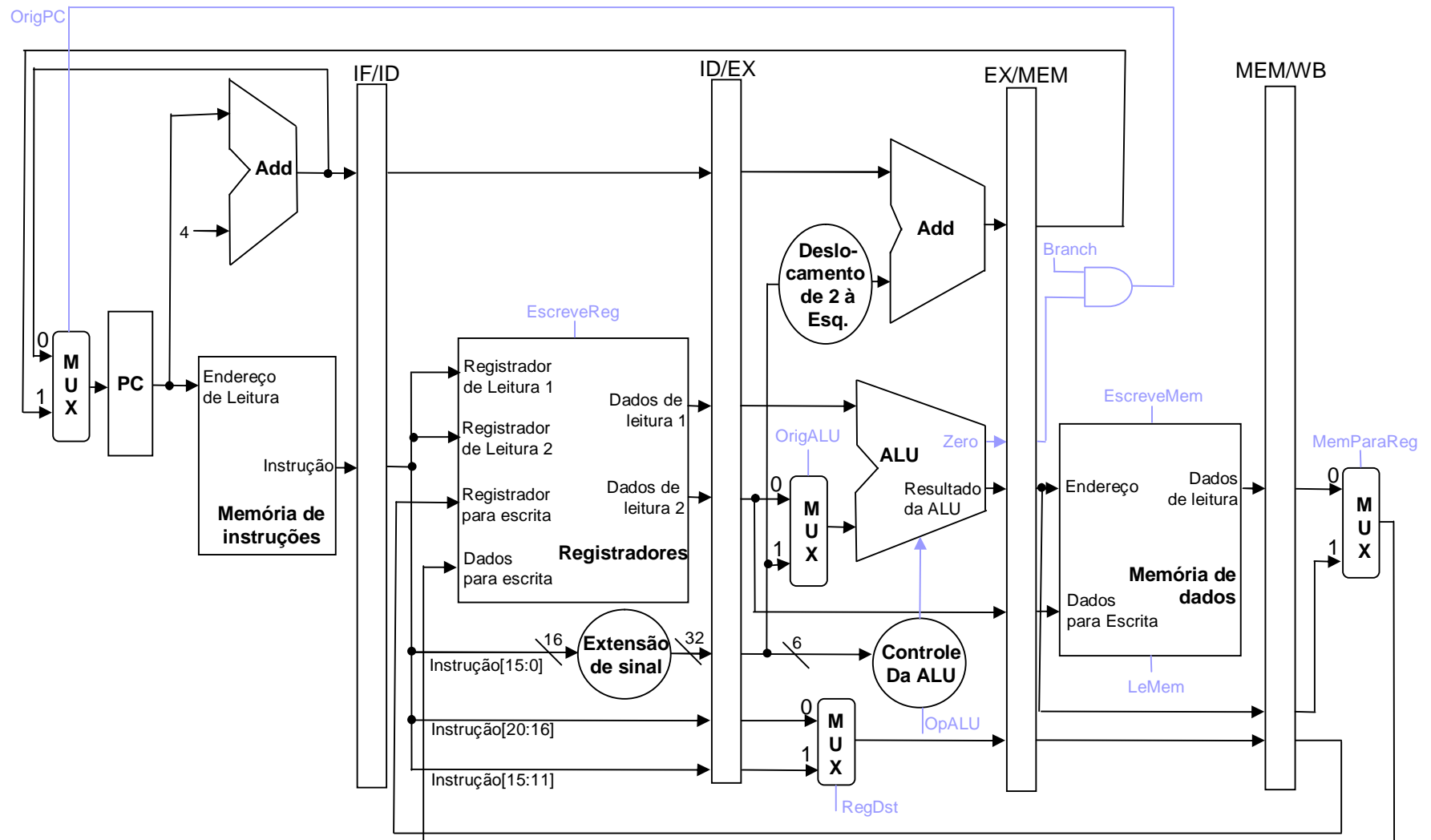
- *Harzards* estruturais: Unidade de ponto flutuante
- *Harzard* de controle: Programas de inteiros
 - Mais desvios, além de desvios menos previsíveis
- *Harzard* de dados: inteiros e ponto flutuante
 - Inteiros
 - Escalonamento de instruções mais complexo
 - Acesso menos regular e maior uso de ponteiros
 - Ponto flutuante
 - Menor frequência de desvios
 - Padrão de acesso mais regular



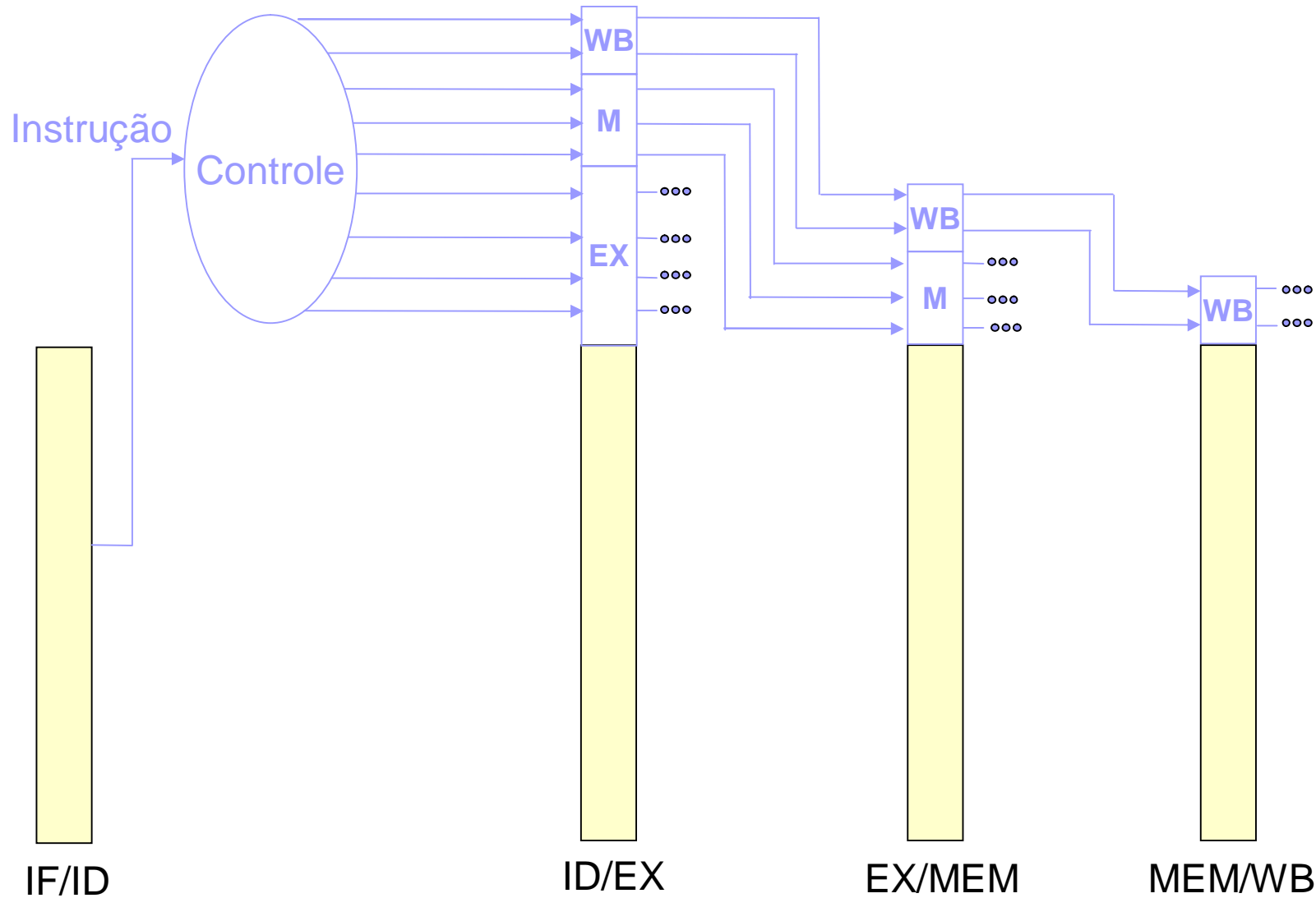
Caminho de Dados Usando *Pipeline*

- Para passar algo de um estágio anterior do *pipeline* para um posterior, informação precisa ser colocada em um registrador
 - Caso contrário, informação é perdida quando próxima instrução entrar nesse estágio
- Cada componente lógico do caminho de dados só pode ser usado dentro de um único ciclo do *pipeline*
 - Caso contrário, *hazard* estrutural

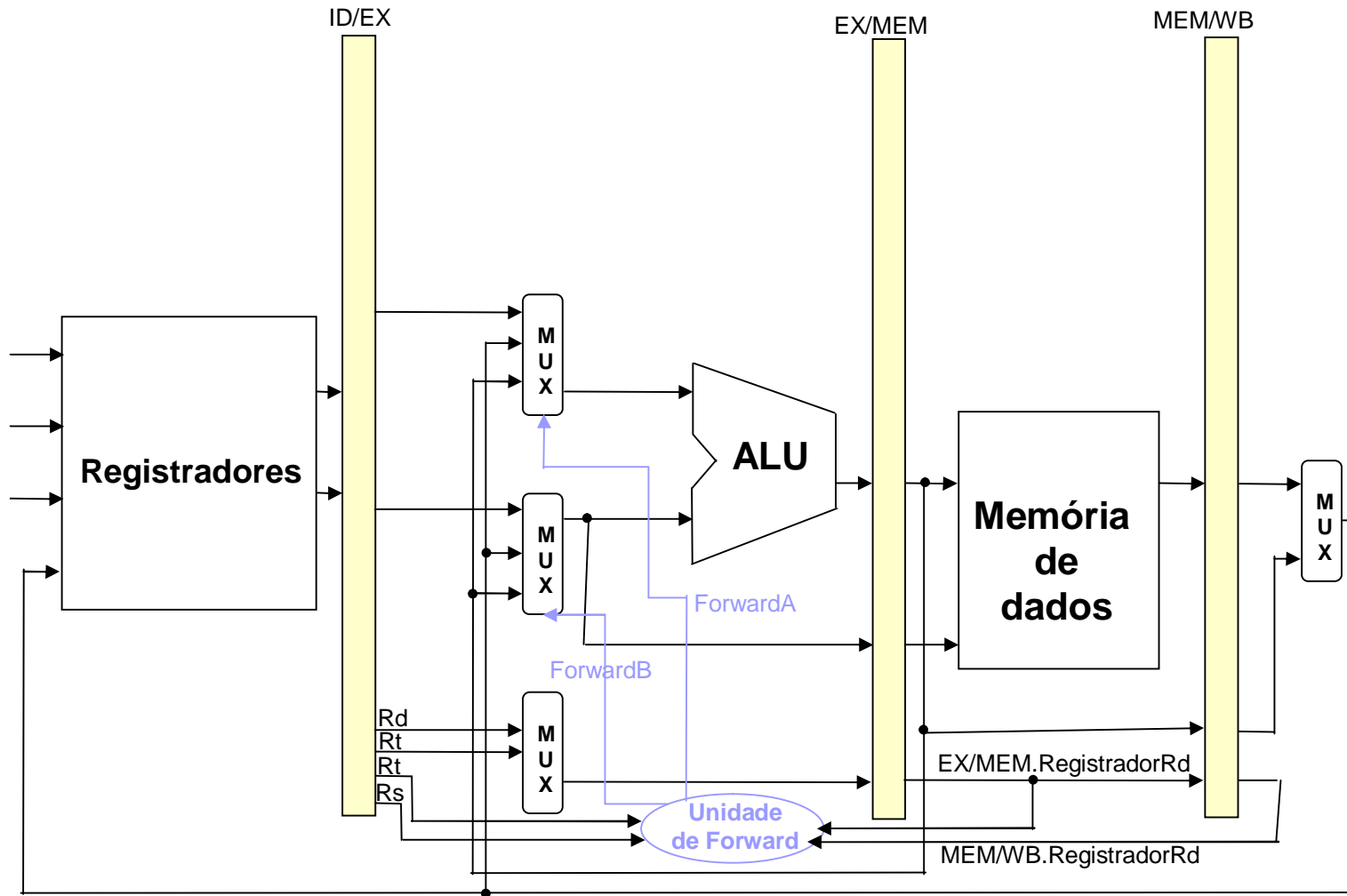
Controle de um *Pipeline*

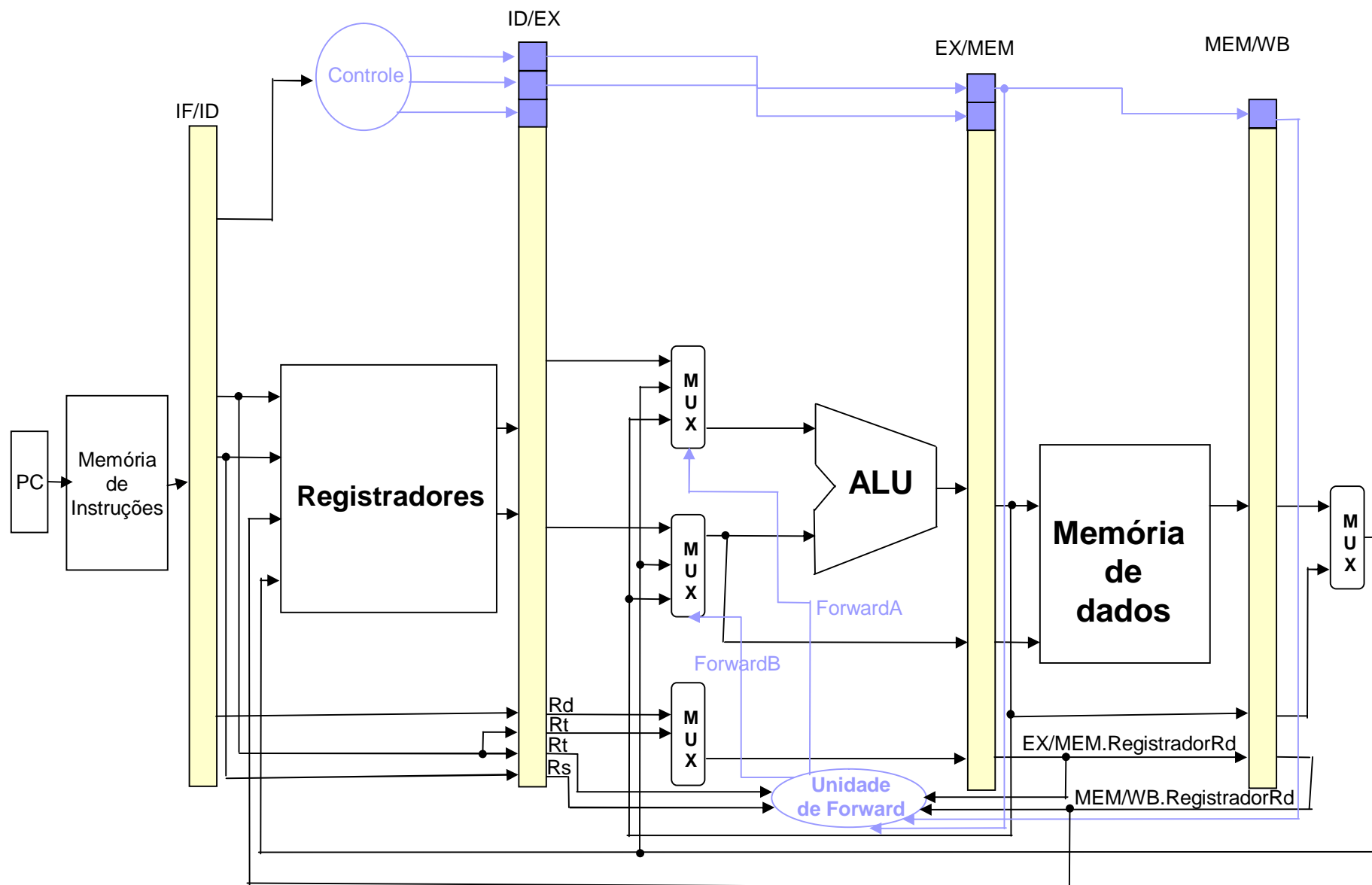


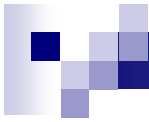
Controle de um *Pipeline*



Hazards de dados e forwarding





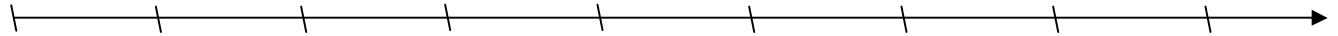


Hazards de Dados e Stalls

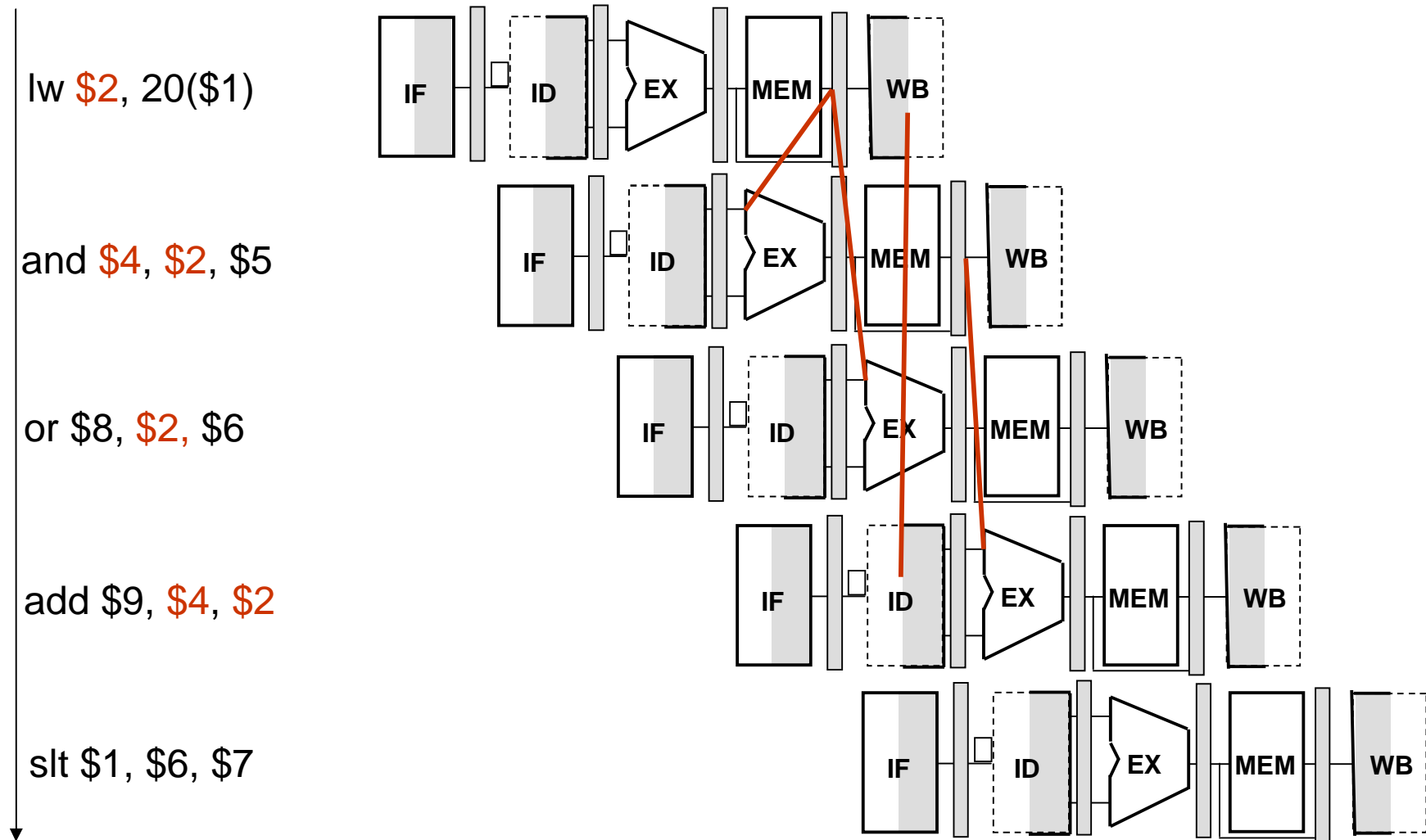
- Leitura ainda pode causar um *hazard*:
 - Uma instrução tenta ler um registrador após uma instrução *load* que escreve no mesmo registrador
- Precisamos de uma unidade de detecção de *hazard*
 - Opera durante estágio ID, inserindo *stall* entre *load* e seu uso

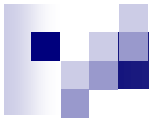


Tempo (em ciclos de clock)



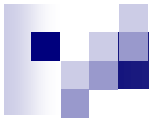
Ordem de execução do programa (em instruções)



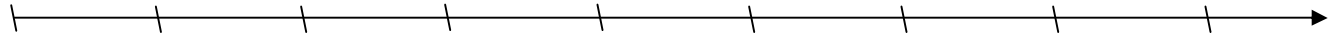


Hazards de Dados e Stalls

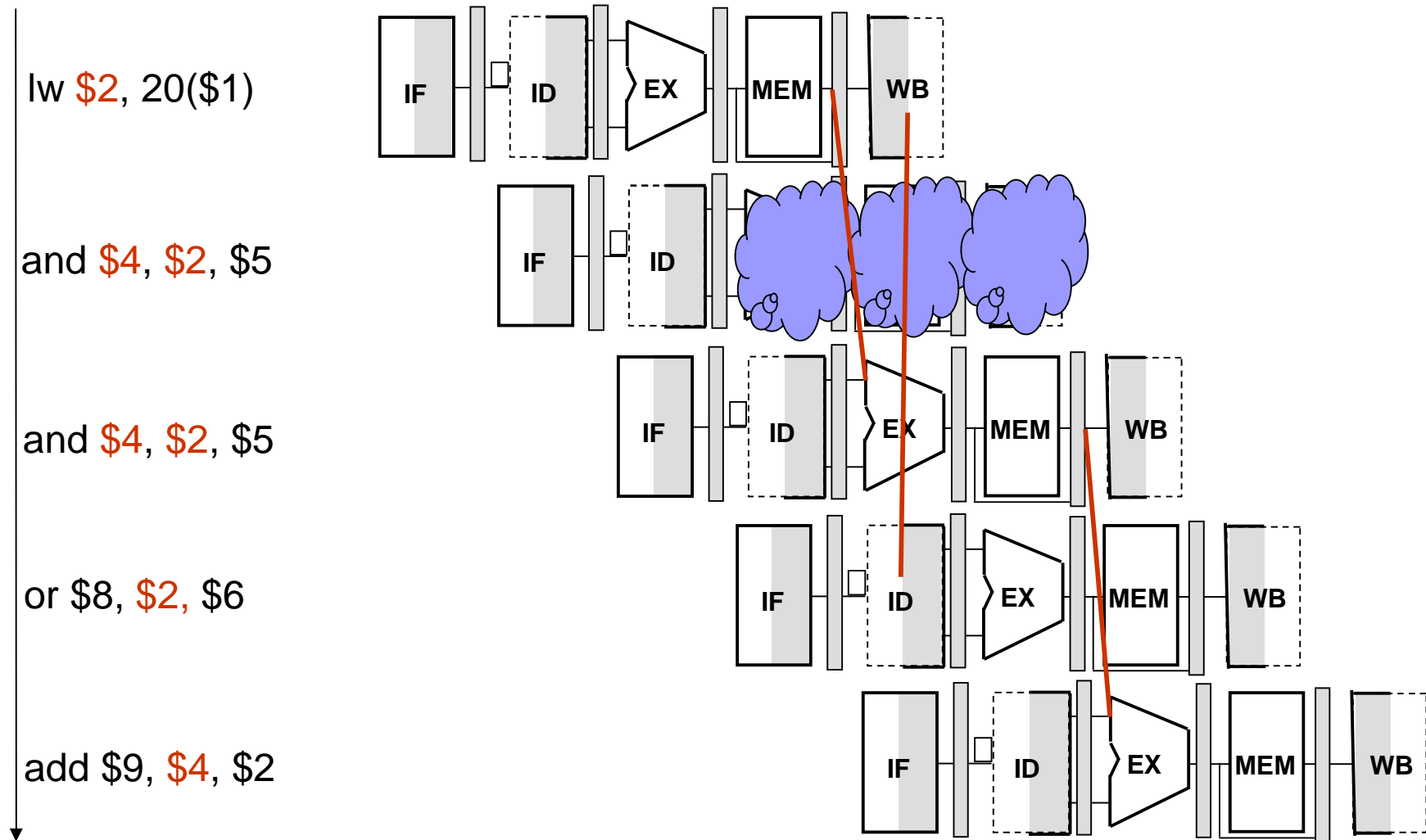
- Se instrução sofre *stall*, precisamos evitar também que próxima instrução avance
 - Não atualizar PC e IF/ID
 - Instrução no estágio IF continuará a ser lida usando o mesmo PC
 - Registradores no estágio ID continuarão a ser lidos usando os mesmos valores
 - Inserir **nops**
 - Colocar 0 em todos os sinais de controle
 - Serão filtrados: nenhum valor modificado se todos os sinais iguais a zero

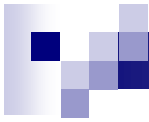


Tempo (em ciclos de clock)



Ordem de execução do programa (em instruções)





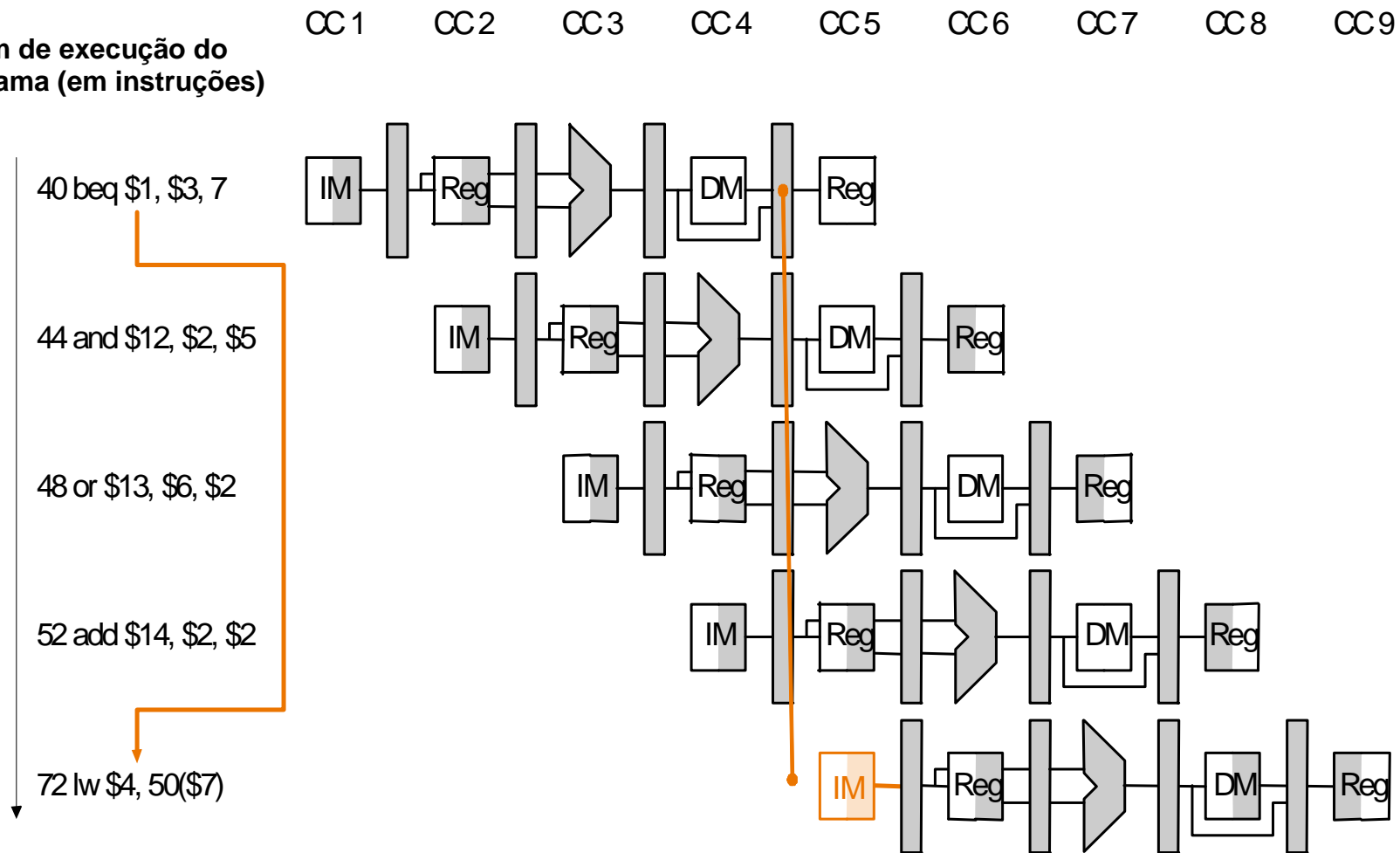
Hazard de Desvio

- Uma instrução precisa ser buscada a cada ciclo para sustentar o *pipeline*
- A decisão do desvio não ocorre até o estágio MEM
- Conforme visto, este atraso sobre a decisão de que instrução buscar é chamado de *hazard* de desvio ou controle

Hazard de Desvio

Tempo (em ciclos de clock)

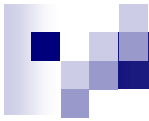
Ordem de execução do programa (em instruções)





Hazard de Desvio

- Considere que o desvio não foi tomado
 - Podemos considerar que desvio não será tomado
 - Caso seja tomado, precisamos descartar instruções buscadas e decodificadas
 - Execução continua no destino do desvio
 - Se desvio tomado na metade das vezes e descartar instruções for barato
 - Otimização reduz pela metade custo do *hazard* de desvio
 - Como descartar instruções?
 - Alteramos valor de controles para zero
 - Fazer *flush* nas instruções nos estágios IF, ID e EX do *pipeline* quando instrução de desvio atinge MEM



Hazard de Desvio

- Técnica: pesquisar o endereço da instrução para ver se desvio foi tomado da última vez
 - Caso tenha sido, busca-se instruções a partir do mesmo lugar da última vez
 - Tabela de histórico de desvios (ou *buffer* de previsão de desvios) usado na implementação
 - Pequena memória indexada pela parte menos significativa do endereço da instrução de desvio
 - Contém um *bit* dizendo se desvio foi tomado recentemente ou não



Exceções

- Outra forma de *hazard*
- Controle deve ser transferido para rotina de exceção imediatamente após instrução
 - Entrada adicional (80000180_{HEX}) no multiplexador que fornece novo PC
- Necessário *flush* nas instruções que vêm após instrução que causou exceção
 - Estágio ID: fazemos OR com o sinal de *stall* da unidade de detecção de *hazard*
 - Estágio EX: novo sinal, EX.Flush



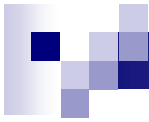
Exceções

- Suponha que **add** \$1, \$2, \$1 cause *overflow*
- Se não pararmos execução no meio da instrução, programador não verá valor \$1 que ajudou a causar exceção
 - Sinal EX.Flush pode ser usado para impedir escrita do resultado no estágio WB



Exceções

- Cinco instruções ativas em qualquer ciclo de *clock*
 - Desafio: associar exceção à instrução correta
 - Estágio do pipeline ajuda: instrução desconhecida (ID), chamada ao SO (EX), ...
- Várias exceções podem ocorrer no mesmo ciclo de *clock*
 - Priorizar exceções
 - Instrução mais antiga interrompida
 - Flexibilidade para solicitações de E/S e defeitos de hardware
 - Não estão associados a instrução específica



Exceções

- Dificuldade de associar exceção correta à instrução correta levou projetistas a relaxarem esse requisito em casos não críticos
 - Interrupções (ou exceções) imprecisas
 - SO determina que instrução causou o problema
- Interrupções (ou exceções) precisas: associadas as instruções corretas



Pipelining Avançado

- *Pipelining* explora paralelismo em potencial entre instruções
 - Paralelismo em nível de instrução
- Dois métodos para aumentar quantidade de paralelismo
 - Aumentar profundidade do *pipelining*
 - Sobrepondo mais instruções
 - Ciclo de *clock* encurtado
 - Replicar componentes (despacho múltiplo)
 - Iniciar várias instruções em cada estágio do pipeline
 - Permite que CPI seja menor que 1 => IPC (instrução por ciclo)



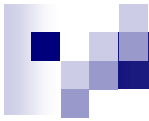
Especulação

- Técnica que permite que o compilador ou processador adivinhem as propriedades de uma instrução, para removê-la como uma dependência na execução de outras instruções
 - Ex: buscamos, emitimos e executamos instruções, como se previsão de desvio estivesse sempre correta
- Como pode estar errada, temos de verificar se a escolha foi certa
 - Caso tenha sido errada, temos de retroceder os efeitos



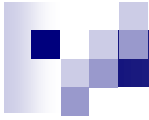
Especulação

- Pode ser feita por compilador ou hardware
- Recuperação diferente
 - Compilador
 - Adiciona instruções para verificar precisão da especulação
 - Rotina de reparo chamada quando especulação incorreta
 - Hardware
 - Resultados especulativos armazenados em *buffer*
 - Se resultados corretos, resultados escritos em registradores ou na memória
- Especular pode introduzir exceções que antes não estavam presentes
 - *Load* com endereço inválido quando especulação incorreta



O Pipeline do Pentium 4

- Vimos que P4 traduzia instruções IA-32 em microoperações
- Microoperações executadas por pipeline especulativo e escalonado dinamicamente
- Três microoperações por ciclo de *clock*
- Extensas filas
 - Até 126 microoperações pendentes
- PF inclui unidade separada para moves de pf
- *Loads/Stores* subdivididos em duas partes: cálculo do endereço e referência real a memória
- ALUs de inteiros operam com o dobro da frequência de *clock*



O Pipeline do Pentium 4

- Cache de trace mantém sequência pré-codificada de microinstruções
- Unidade de PF também trata operações multimídia (MMX e SSE2)

