O Processador - Parte 2 - Pipelining

Arquitetura de Computadores

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

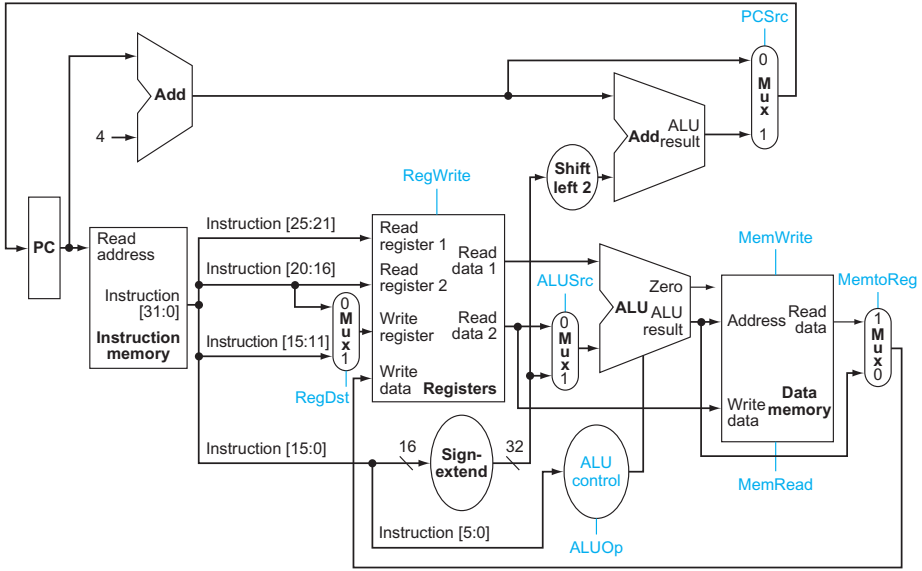
Disclaimer

UFABC

- Estes slides foram preparados para o curso de Arquitetura de Computadores na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- O conteúdo destes slides foi baseado no conteúdo do livro *Computer Organization And Design: The Hardware/Software Interface*, 5th Edition.

Pipelining

O Datapath com a unidade de Controle



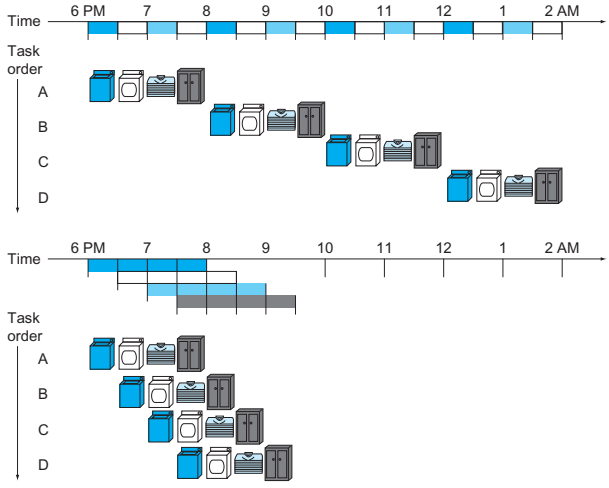
Porque este tipo de Datapath não é usado hoje em dia?

UFABC

- O delay mais longo determina o período do clock
  - Caminho crítico - Load Instruction
  - Memória de instrução $\rightarrow$ banco de registradores $\rightarrow$ ALU $\rightarrow$ memória de dados $\rightarrow$ banco de registradores
- Não é razoável alterar o período a cada instrução
- Viola o princípio de tornar rápido o caso mais comum
- Veremos como melhorar o desempenho através de pipelining

Uma analogia com lavar roupas

Execuções paralelas de tarefas para aumento de desempenho



- 4 cargas na máquina
- Tempo total: 8 horas
- Versão com pipeline
  - Tempo total: 3.5h, ou 2.3x + rápido
  - *Speedup* = $2n/0.5n + 1.5 \approx 4$ = número de estágios!

UFABC

Cinco estágios, um passo por estágio

1. `IF` *Instruction Fetch* - Busca instrução na memória.
2. `ID` *Instruction Decode* - Decodifica a instrução e lê os registradores.
3. `EX` *Execute* - Executa a operação dada pela instrução e calcula endereços.
4. `MEM` *Memory* - Acesso aos operandos em memória.
5. `WB` *Write Back* - Escreve o resultado da operação de volta ao registrador.

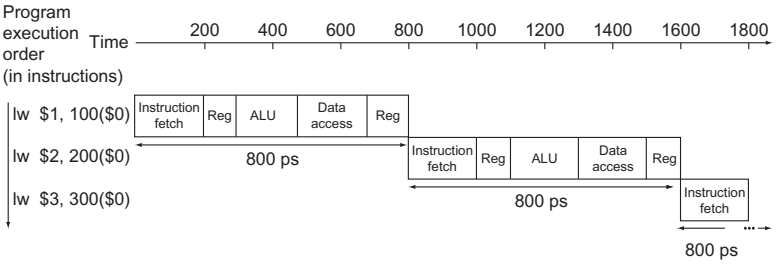
Desempenho do pipeline

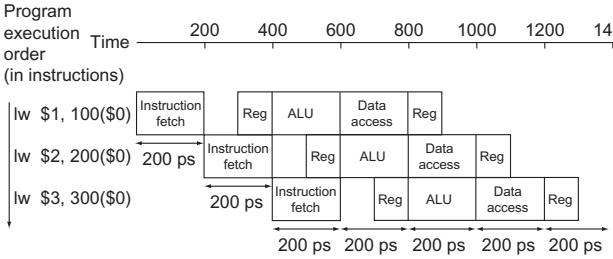
UFABC

- Assuma que o tempo por estágio do pipeline seja de
  - 100ps para leitura ou escrita de registradores
  - 200ps para os demais
- Compare o desempenho de um datapath monociclo com um datapath com pipeline

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Desempenho do pipeline





- Monociclo
  - $T_c = 800$ ps
- Pipelined
  - $T_c = 200$ ps

Speedup com o uso de pipeline

UFABC

- Se todos os estágios estiverem equilibrados (*i.e.* levam o mesmo tempo)

$$\text{Tempo entre instruções}_{\text{Pipeline}} = \frac{\text{Tempo entre instruções}_{\text{Sem pipeline}}}{\text{Número de estágios}}$$

- Se não estiverem balanceados, então o *speedup* é menor
- O *speedup* é devido ao aumento da vazão (*en:* ***throughput***)
  - Contudo, o tempo total no início ao fim da execução de uma instrução, ou a latência (*en:* ***latency***) não se altera.

UFABC

- A ISA do MIPS foi criada para ser executada em um pipeline.
  - Todas as instruções tem 32 bits.
    - Facilita o *fetch* e *decode* em um ciclo.
    - Compare com o x86 que tem instruções de 1 a 17 bytes.
  - Poucos formatos e, dentre eles, com regularidade.
    - É possível fazer a decodificação e a leitura de registradores em um único passo.
  - Endereçamento apenas por load e stores.
    - Permite calcular o endereço no 3^o estágio do pipeline e fazer. os acessos no 4^o.
  - Operandos de instruções de acesso à memória devem ser alinhados.
    - Garante que os acessos à memória tomem apenas um ciclo.

Pipeline Hazards

Hazards

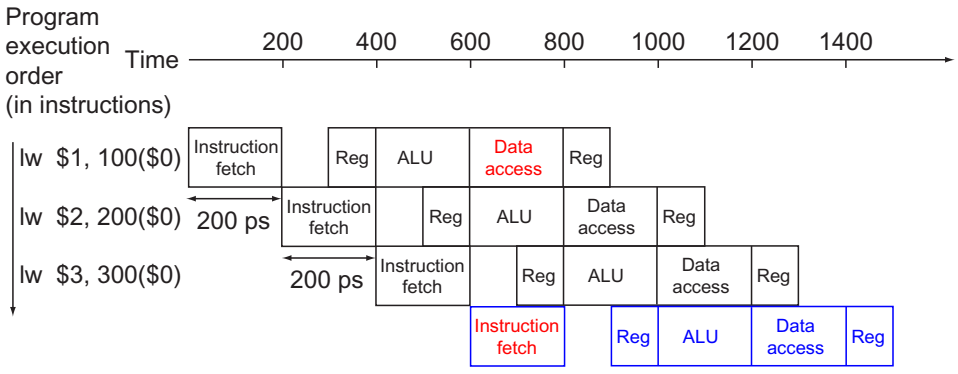
UFABC

- Em certas situações não é possível iniciar a execução de uma instrução no ciclo imediatamente após a instrução anterior
- Essas situaçoes são chamadas de *hazards* e são, comumente, divididas em 3 tipos:
  - *Structure hazards*
    - Ocorrem quando um recurso necessário está ocupado.
  - *Data hazards*
    - Ocorrem quando é necessário aguardar que uma instrução anterior termine sua leitura/escrita de dados.
  - *Control hazards* ou *Branch hazard*
    - Ocorrem quando uma decisão de controle de execução depende de uma instrução anterior.

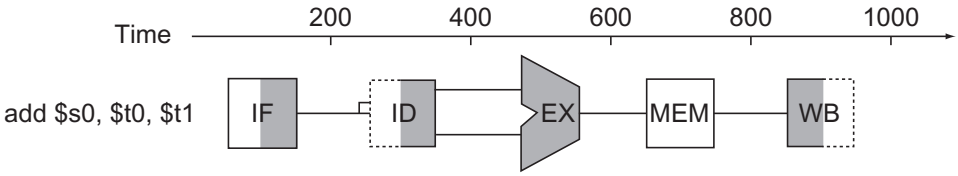
Structure hazards

UFABC

- Ocorrem quando há conflito por um recurso
- Possível no MIPS, se houver uma única memória
  - Load/Store precisam acessar dados
  - Busca de instruções terá que adiar (*en: **stall***) o ciclo
    - Acaba causando uma bolha no pipeline
  - Pipeline stalls também são conhecidos como bolhas (***bubbles***)
- Logo, datapaths com pipelines precisam de memórias separadas para instruções e dados para evitar atrasos
  - Ou, de maneira equivalente, caches independentes para dados e instruções


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
200 ps
200 ps

Representando uma instrução



- Cada estágio é representado por um bloco
- Os sombreados indicam que aquele estágio é utilizado pela instrução sendo executada
  - Sombreados à direita indicam leitura e à esquerda escrita
- No exemplo, a instrução `add` não faz acesso à memória, logo a caixa `MEM` não está sombreada

UFABC

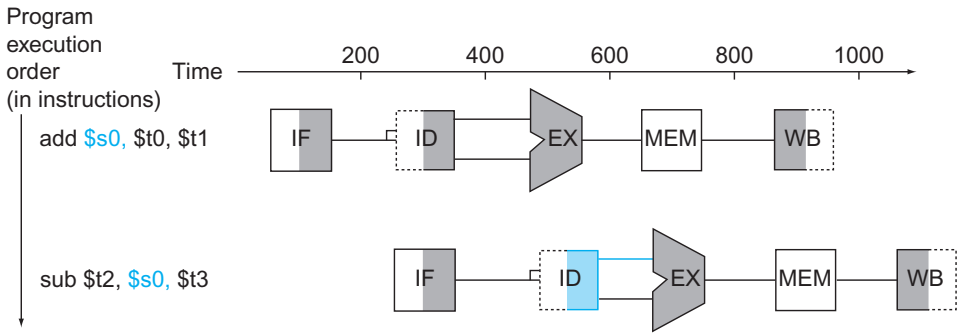
- Uma instrução depende do término do estágio de acesso aos dados de uma instrução anterior

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

- Ambas instruções usam `$s0`
  - `add` escreve em `$s0`
  - `sub` depende do resultado de `add`

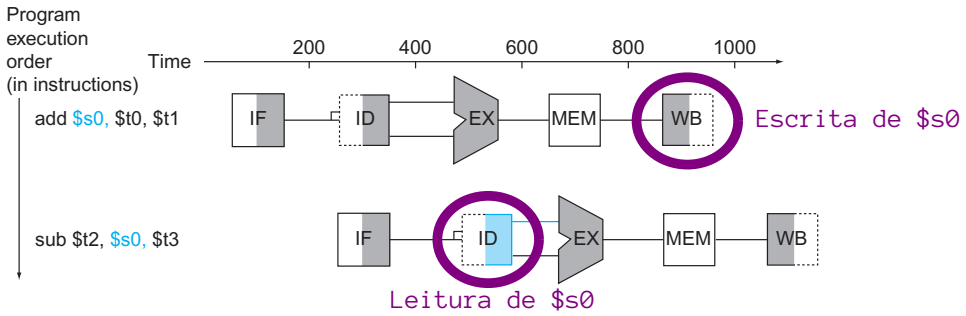
Data hazard

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



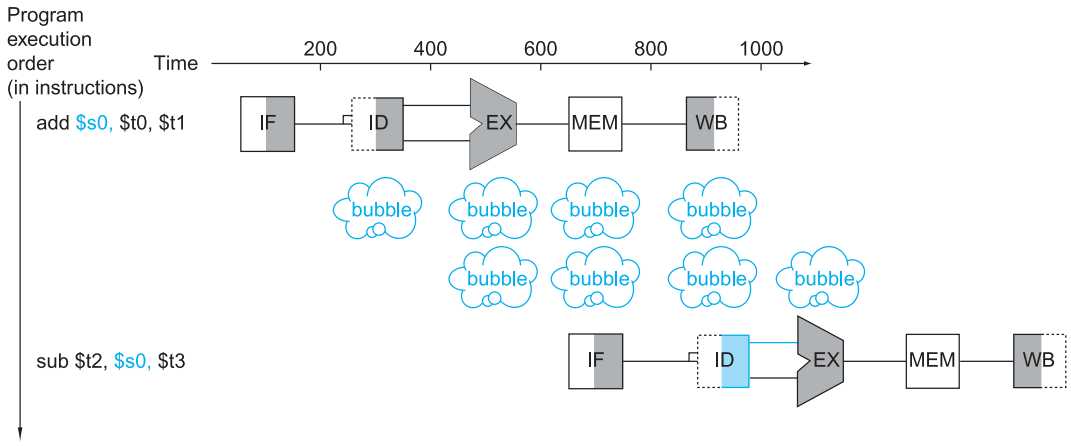
UFABC

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



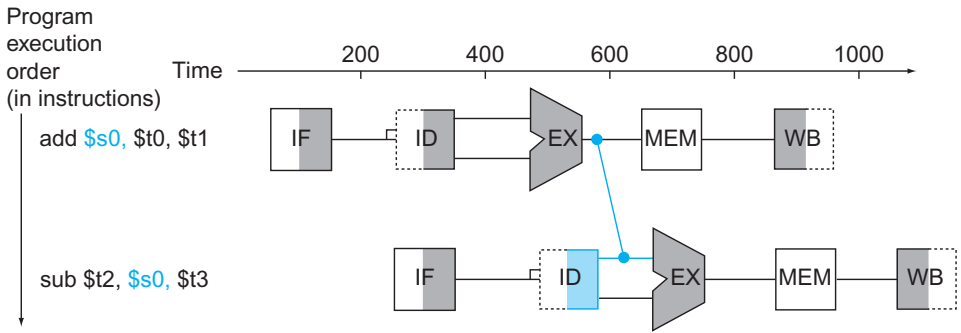
Data hazard - Solução 1

- Inserimos bolhas no pipeline para garantir que o valor necessário para a instrução já esteja disponível



Data hazard - Solução 2

- Adiantamos o valor logo depois do estágio de execução para a próxima instrução
- Chamado de *forwarding* ou *bypassing*
- Exige hardware adicional no datapath



Load-Use Data Hazard

UFABC

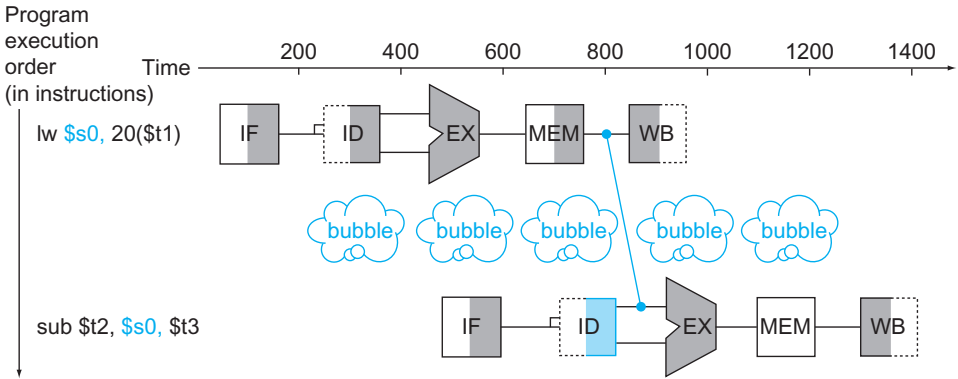
- Nem sempre apenas o fowarding é suficiente para resolver um hazard

```
1 lw $s0, 20($t1)
2 sub $t2, $s0, $t3
```

- O valor só vai estar disponível após o estágio **MEM**
- Não dá para fazer forward voltando no tempo! :D

UFABC

- Possível solução é combinar *forwarding* com *pipeline stalls*



Reordenando instruções para evitar bolhas

UFABC

- Código em C:

```
1  a = b + e;
2  c = b + f;
```

- Código MIPS (variáveis acessíveis a partir de `$t0`)

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
5  lw  $t4, 8($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

- Quais são os hazards? Quais se resolvem com forwarding?

Reordenando instruções para evitar bolhas

- Código em C:

```
1  a = b + e;
2  c = b + f;
```

- Código MIPS (variáveis acessíveis a partir de `$t0`)

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
5  lw  $t4, 8($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  lw  $t4, 8($t0)
4  add $t3, $t1, $t2
5  sw  $t3, 12($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

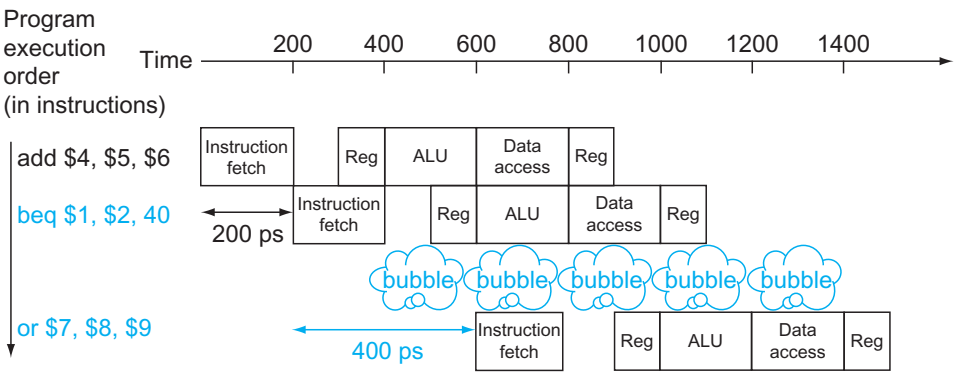
- Quais são os hazards? Quais se resolvem com forwarding?

Control hazards

UFABC

- Branches determinam o controle do fluxo do programa
  - A busca da próxima instrução depende da avaliação do branch
  - O pipeline não é capaz de sempre buscar a instrução correta
    - Ainda está trabalhando na identificação da instrução (estágio `ID`)
- No MIPS
  - Necessário comparar registradores e computar o destino do branch nos primeiros estágios do pipeline
  - O MIPS tem hardware para fazer isto no estágio `ID`

- Espera o resultado do branch antes de buscar a próxima instrução



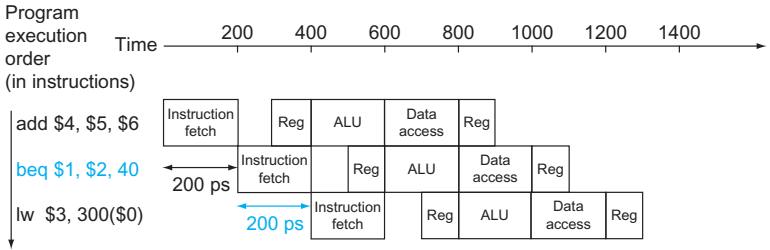
UFABC

- Pipelines mais longos não conseguem determinar o resultado de um branch de maneira adiantada como o MIPS
  - O custo de inserir bolhas se torna proibitivo
- Saída? Chuta-se (cof cof), predizemos o resultado do branch
  - Se o chute for errado, é causado um stall
- No MIPS
  - É capaz de predizer branches não tomados (ele sempre chuta que não é tomado)
  - Logo, carrega, sempre, a instrução seguinte ao branch

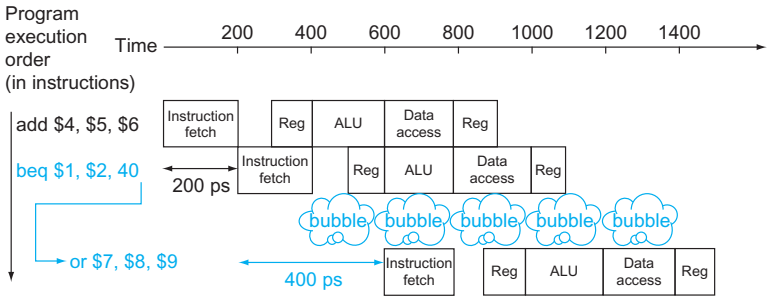
Branch Predictor no MIPS (Not Taken)

UFABC

Predição correta



Predição incorreta



UFABC

Preditor de branches estático

- Baseado no comportamento típico do branch
- Exemplo: laços e ifs
  - Prediz que branches para trás são tomados
  - Prediz que branches para frente não são tomados

Preditor de branches dinâmico

- Exige hardware adicional para medir o comportamento do branch
  - Normalmente armazena informações sobre o comportamento mais recente de cada um dos branches
  - Assume que os comportamentos passados são indicativos do comportamento futuro
    - Quando erra, insere uma bolha, carrega a instrução correta e atualiza o histórico
    - Usado em processadores modernos e tem acerto superior a 90%

Resumo sobre pipelining

UFABC

- Pipelining aumenta o desempenho pelo aumento da vazão
  - Mas não faz nada quanto à latência de cada instrução
  - Baseia-se na execução em paralelo de múltiplas instruções
- Pode sofrer de hazards
  - Estruturais, de dados ou de controle
- A ISA pode influenciar significativamente a complexidade de implementação de um pipeline