Introdução

- ***Pipelining*** é uma técnica de implementação de arquiteturas do *set* de instruções (ISA), através da qual múltiplas instruções são executadas com algum grau de **sobreposição temporal**

- O objetivo é aproveitar, de forma o mais eficiente possível, os recursos disponibilizados pelo *datapath*, por forma a **maximizar a eficiência global do processador**

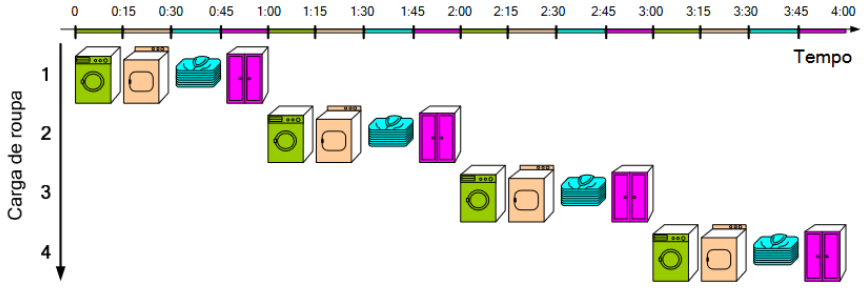
Pipelining - exemplo por analogia

- O exemplo de *pipelining* que iremos observar de seguida apoia-se num conjunto de tarefas simples e intuitivas: o processo de tratamento da roupa suja ☺

- Neste exemplo, o tratamento da roupa suja desencadeia-se nas seguintes quatro fases:
  1. Lavar uma carga de roupa na máquina respetiva
  2. Secar a roupa lavada na máquina de secar
  3. Passar a ferro e dobrar a roupa
  4. Arrumar a roupa dobrada no guarda roupa respetivo

Pipelining - exemplo por analogia

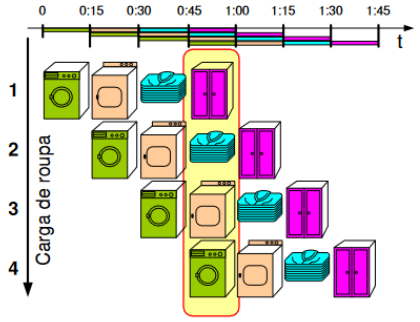
- Este processo pode então ser descrito temporalmente do seguinte modo:



Se o tempo para tratar uma carga de roupa for uma hora, tratar quatro cargas demorará **quatro horas.**

Pipelining - exemplo por analogia

- O processo de tratamento da versão *pipelined* pode então ser descrito temporalmente do seguinte modo:
- Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45 (ou seja 135 minutos menos (240 – 105).

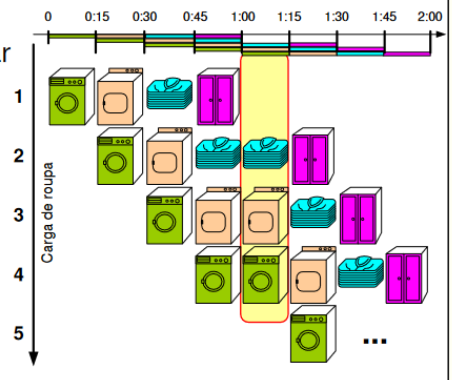


- O que acontece se, por exemplo, a carga 2 não precisar de ser engomada?

S

Pipelining - exemplo por analogia

- O que acontece se a carga 2 tiver roupa que, por alguma razão, demora mais tempo a engomar?
  - É necessária uma segunda "slot" de 15 min para completar a engomagem da carga 2
  - A carga 3 não pode avançar para a engomagem e permanece na máquina de secar
  - A carga 4 não pode avançar para a máquina de secar e permanece na máquina de lavar
  - A carga 5 só é colocada na máquina de lavar no minuto 75

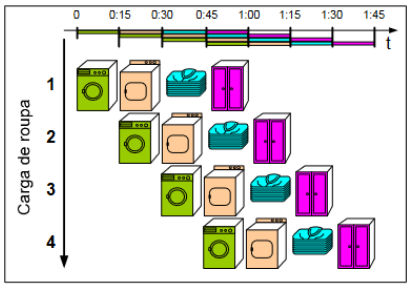


Pipelining - exemplo por analogia

- O paradoxo aparente da solução *pipelined* é que o tempo necessário para o processamento completo de uma carga de roupa não difere do tempo da solução não *pipelined*
- A eficiência da solução com *pipelining* decorre do facto de, para um número grande de cargas de roupa, todos os passos intermédios estarem a executar em paralelo
- O resultado é o aumento do número total de cargas de roupa processadas por unidade de tempo (***throughput***)
- Qual o **ganho de desempenho** que se obtém com o sistema *pipelined* relativamente ao sistema normal?

Pipelining – ganho de desempenho

- O tratamento de N cargas de roupa num sistema com F fases demorará idealmente (admitindo que cada fase demora 1 unidade de tempo):



Sistema não *pipelined*: $T_{NON\text{-}PIPELINE} = N \times F$

Sistema *pipelined*: $T_{PIPELINE} = F + (N - 1) = (F - 1) + N$

Ganho de desempenho obtido com a solução *pipelined*: $$\frac{Desempenho_{PIPELINE}}{Desempenho_{NON\text{-}PIPELINE}} = \frac{T_{NON\text{-}PIPELINE}}{T_{PIPELINE}} = \frac{N \times F}{(F - 1) + N}$$

Se N >> (F-1), então: $$Ganho \approx \frac{N \times F}{N} = F$$

Pipelining – ganho de desempenho

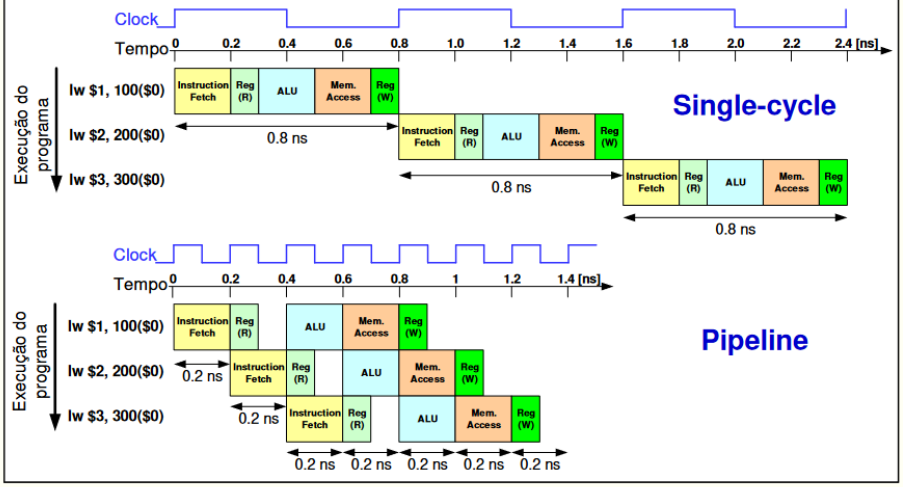
- No limite, para um número de cargas de roupa muito elevado, o ganho de desempenho (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e noutro modelo) é da ordem do **número de tarefas realizadas em paralelo** (isto é, igual ao número de fases do processo)
- Genericamente, poderíamos afirmar que o ganho em velocidade de execução é igual ao número de estágios do *pipeline* (F)
- No exemplo observado, o ganho teórico estabelece que a solução *pipelined* é quatro vezes mais rápida do que a solução não *pipelined*
- A adoção de *pipelines* muito longos (com muitos estágios) pode, contudo, limitar drasticamente a eficiência global

Pipelining - exemplo por analogia

- Na versão *pipelined*, aproveita-se para carregar uma nova carga de roupa na máquina de lavar mal esteja concluída a lavagem da primeira carga
- O mesmo princípio se aplica a cada uma das restantes três tarefas
- Quando se inicia a arrumação da primeira carga, todos os passos (chamados **estágios** ou **fases** em *pipelining*) estão a funcionar em paralelo
- Maximiza-se assim a utilização dos recursos disponíveis

Datapath pipelined para o MIPS

- Exemplo de execução de 3 instruções LW nos *datapaths* *single-cycle* e *pipelined*



Datapath pipelined para o MIPS

- O *instruction set* do **MIPS** (***M****icroprocessor without* ***I****nterlocked* ***P****ipeline* ***S****tages*) foi concebido para uma implementação em *pipeline*. Os aspetos fundamentais a considerar são:
  - **Instruções de comprimento fixo**: *Instruction Fetch* e *Instruction Decode* podem ser feitos em estágios sucessivos (a unidade de controlo não tem que ter em consideração a dimensão da instrução descodificada)
  - **Poucos formatos de instrução**, com a referência aos registos a ler sempre nos mesmos campos (isso permite que os registos sejam lidos no segundo estágio ao mesmo tempo que a instrução é descodificada pela unidade de controlo)
  - **Referências à memória só aparecem em instruções de load/store**: o terceiro estágio pode ser usado para calcular o resultado da operação na ALU ou para calcular o endereço de memória, permitindo o acesso à memória no estágio seguinte
  - Os **operandos em memória têm que estar alinhados**: qualquer operação de leitura/escrita da memória pode ser feita num único estágio

Datapath pipelined para o MIPS

- O *pipeline* implementa as cinco fases sequenciais em que são decomponíveis as instruções:
  1. (**IF**) - *Instruction fetch* (ler a instrução da memória), incremento do PC
  2. (**ID**) - *Operand fetch* (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
  3. (**EX**) - Executar a operação ou calcular um endereço
  4. (**MEM**) - *Memory access* (aceder à memória de dados para leitura ou escrita)
  5. (**WB**) - *Write-back* (escrever o resultado no registo destino)
- A solução *pipelined* para o MIPS parte do modelo do *datapath single-cycle*
- Na solução apresentada no slide seguinte não são identificados os sinais de controlo nem a respetiva unidade de controlo

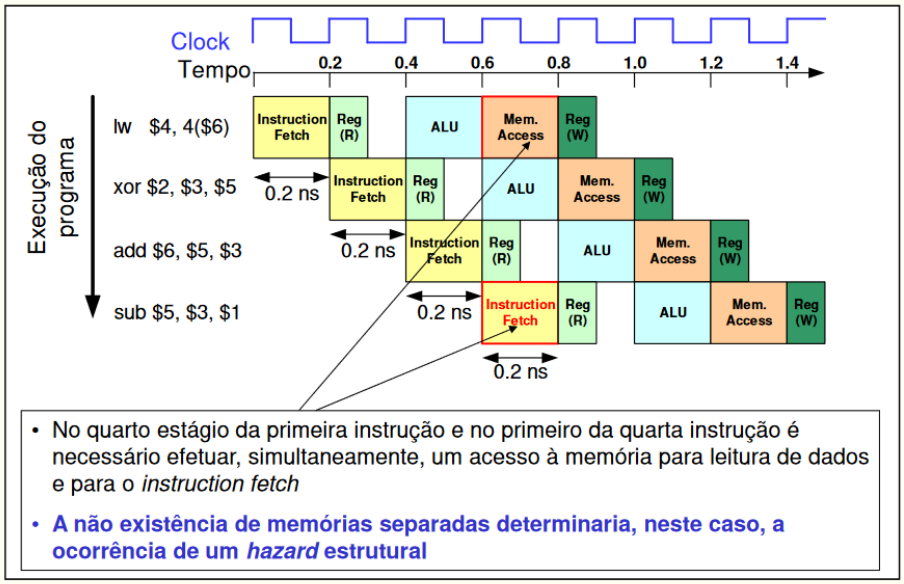
Pipeline Hazards

- Existe um conjunto de situações particulares que podem condicionar a progressão das instruções no *pipeline* no próximo ciclo de relógio
- Estas situações são designadas genericamente por ***hazards***, e podem ser agrupadas em três classes distintas:
  - ***Hazards* estruturais**
  - ***Hazards* de controlo**
  - ***Hazards* de dados**
- Nos próximos slides serão discutidas, para cada tipo de *hazard*, as origens e as consequências, mapeando depois esses aspetos ao nível da implementação da arquitetura *pipelined* do MIPS

Hazards Estruturais

- Um ***hazard* estrutural** ocorre quando mais do que uma instrução necessita de aceder ao mesmo hardware
- Ocorre quando: 1) apenas existe uma memória ou 2) há instruções no *pipeline* com diferentes tempos de execução
- No primeiro caso o *hazard* estrutural é evitado duplicando a memória, i.e., uma memória de instruções e uma memória de dados (acesso em IF não conflitua com possível acesso em MEM)
- O segundo caso está fora da análise feita nestes slides; como exemplo pode pensar-se na implementação de uma instrução mais complexa que demore 2 ciclos de relógio na fase EX, usando outro elemento operativo diferente da ALU

Hazards Estruturais



- No quarto estágio da primeira instrução e no primeiro da quarta instrução é necessário efetuar, simultaneamente, um acesso à memória para leitura de dados e para o *instruction fetch*
- **A não existência de memórias separadas determinaria, neste caso, a ocorrência de um *hazard* estrutural**

Hazards de Controlo

- Um *hazard* de controlo ocorre quando é necessário fazer o *instruction fetch* de uma nova instrução e existe numa etapa mais avançada do *pipeline* uma instrução que pode alterar o fluxo de execução e que ainda não terminou
- Exemplo:

```
        beq $5,$6,next
        add $2,$3,$4
        ...
next:   lw  $3,0($4)
        ...
```

Qual a instrução que deve entrar no *pipeline* a seguir à instrução "beq"?

- No caso do MIPS, as situações de *hazard* de controlo surgem com as instruções de salto, (*jumps* e *branches*: j, jal, jalr, jr, beq, …)

Hazards de Controlo

- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de instruções
- A resolução dos *branches* em ID minimiza o problema, e por isso a **comparação dos operandos passa a ser efetuada no 2º estágio (ID)**, através de hardware adicional
- Do mesmo modo, **o cálculo do *Branch Target Address* passa também a ser efetuado em ID**

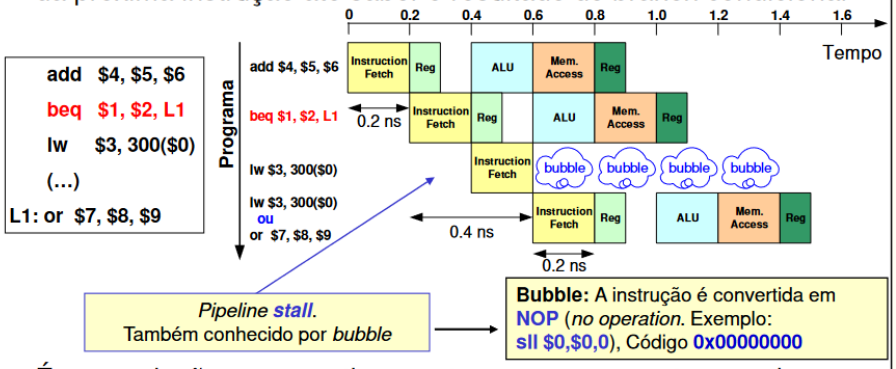
Hazards de Controlo

- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de instruções
- A resolução dos *branches* em ID minimiza o problema, e por isso a **comparação dos operandos passa a ser efetuada no 2º estágio (ID)**, através de hardware adicional
- Do mesmo modo, **o cálculo do *Branch Target Address* passa também a ser efetuado em ID**

DETI-UA Arquitetura de Computadores I Aulas 22 a 26 - 34

Hazards de controlo

- Há mais do que uma solução para lidar com os *hazards* de controlo. A primeira que vamos analisar é designada por ***stalling*** ("parar o progresso de…")
- Nesta estratégia a unidade de controlo atrasa a entrada no *pipeline* da próxima instrução até saber o resultado do *branch* condicional



Pipeline ***stall***.
Também conhecido por *bubble*

Bubble: A instrução é convertida em **NOP** (*no operation*. Exemplo: **sll $0,$0,0**), Código **0x00000000**

- É uma solução conservativa que tem um preço em termos de tempo de execução

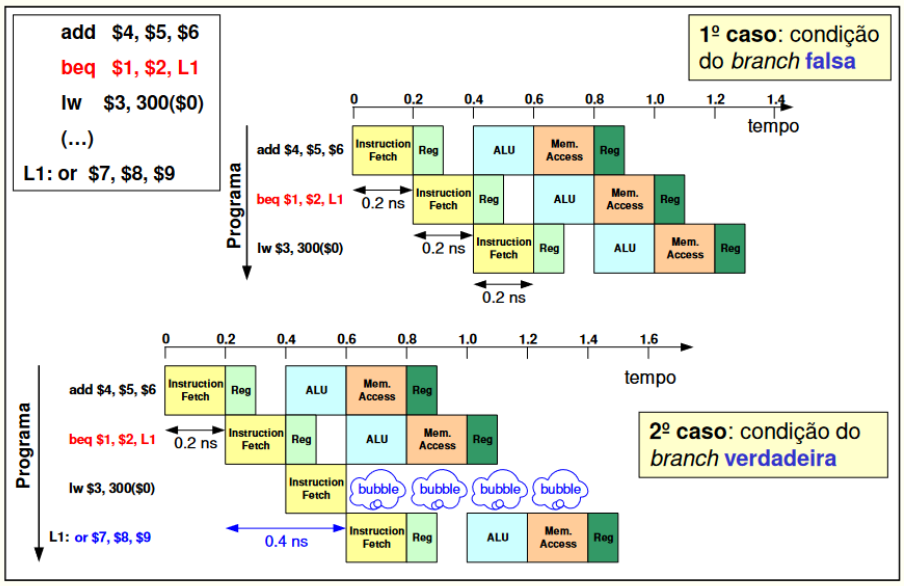
Hazards de controlo

- Uma solução alternativa ao *pipeline stalling* é designada por **previsão** (*prediction*):
  - Prevê-se que a condição do *branch* é falsa (*branch not taken*), pelo que a próxima instrução a ser executada será a que estiver em PC+4 – estratégia designada por **previsão estática *not taken***
  - Se a previsão falhar, a instrução entretanto lida (a seguir ao *branch*) é anulada (convertida em **nop**), continuando o *instruction fetch* na instrução correta
- Se a previsão estiver certa, esta estratégia permite poupar tempo
- Para o exemplo do slide anterior, se a previsão for correta 50% das vezes, a relação de desempenho passa a ser:

  CPI = 1 + 1 * 0,15 / 2 = 1,075

  Relação de desempenho = 1 / 1,075 = 0,93

Hazards de controlo – previsão estática *not taken*



S

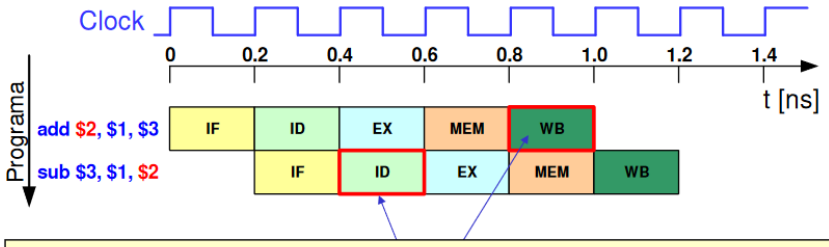
Hazards de controlo – previsão

- Os previsores usados nas arquiteturas atuais são mais elaborados
- **Previsores estáticos**: o resultado da previsão não depende do histórico da execução das instruções de *branch* / *jump*:
  - **Previsor *Not taken***
  - **Previsor *Taken***
  - **Previsor *Backward taken, Forward not taken*** (BTFNT)
- **Previsores dinâmicos**: o resultado da previsão depende da história de *branches* anteriores:
  - Guardam informação do resultado *taken/not taken* de *branches* anteriores e do *target address*
  - A previsão é feita com base na informação guardada

Hazards de dados

- O terceiro tipo de *hazards* resulta da **dependência** existente entre o resultado calculado por uma instrução e o operando usado por outra que segue mais atrás no *pipeline* (i.e., mais recente)
- Exemplo:

```
add  $2,$1,$3
sub  $3,$1,$2
```



A instrução "**sub $3,$1,$2**" não pode avançar no *pipeline* antes de o valor de **$2** ser calculado e armazenado pela instrução anterior (o valor é necessário em **t = 0.4**, mas só vai ser escrito no registo destino em **t = 1**)

Hazards de dados

- Se o resultado necessário para a instrução mais recente ainda não tiver sido armazenado, então essa instrução não poderá prosseguir porque irá tomar como operando um valor incorreto (**a escrita no registo só é feita quando a instrução chega a WB**)
- No exemplo anterior, a instrução SUB só poderia prosseguir para a fase EX em t=1.2
- O problema pode ser minorado, se a **escrita no banco de registos for feita a meio do ciclo de relógio** (i.e., na transição descendente)
  - a instrução que está na fase ID e que necessita do valor, poderá prosseguir na transição de relógio seguinte, já com o valor do registo atualizado
  - poupa-se 1 ciclo de relógio (no exemplo anterior, a instrução "**sub $3,$1,$2**" poderá prosseguir para a fase EX em t=1.0)

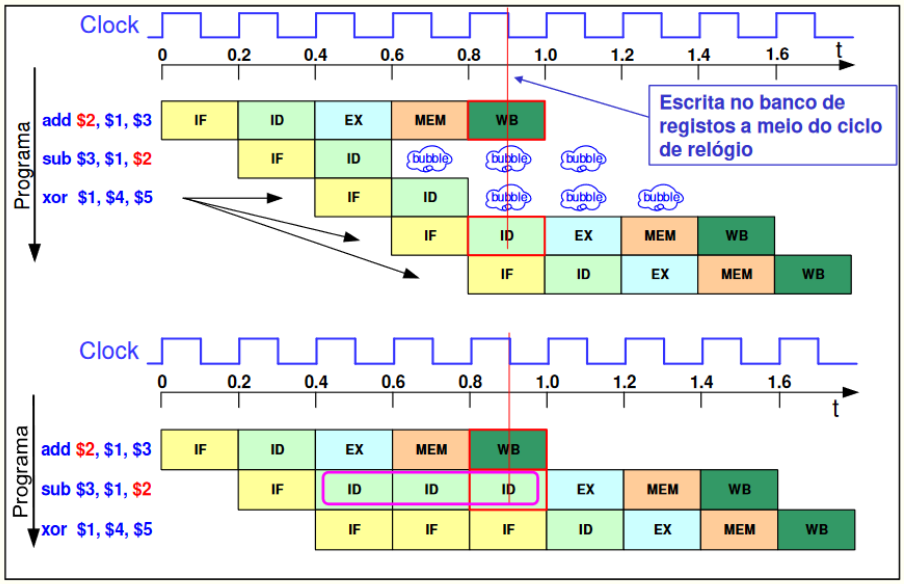
Hazards de dados – *pipeline stalling*

- Para o exemplo anterior, parar a progressão da instrução SUB no estágio ID durante 2 ciclos de relógio é equivalente a introduzir dois NOP entre as duas instruções

```
add   $2,$1,$3   # WB
nop              # MEM
nop              # EX
sub   $3,$1,$2   # ID
```

- Com a escrita no banco de registos feita a meio do ciclo de relógio, a instrução SUB lê o valor atualizado de $2, quando a instrução ADD está na fase WB
- Primeira solução – ***stall* do *pipeline***:
  - **parar a progressão no *pipeline* (*stall)*** da instrução que necessita do valor (e das anteriores), no estágio **ID**, até que a instrução que produz o resultado chegue ao estágio **WB**

Hazards de dados – *pipeline stalling*



Cansei-me FDS…