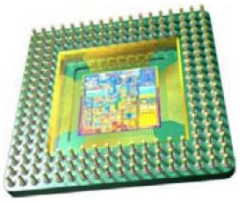
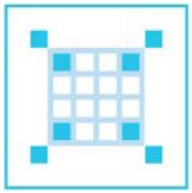


Arquitectura de Computadores

LICENCIATURA EM ENGª INFORMÁTICA
FACULDADE DE CIÊNCIA E TECNOLOGIA
UNIVERSIDADE DE COIMBRA



Pipelining

O *pipelining* é uma técnica que permite a execução de múltiplas instruções em paralelo e que em geral permite melhorar de forma dramática o desempenho de microprocessadores. Neste trabalho pretende-se ilustrar algumas das ideias chave por detrás deste conceito, bem como as soluções típicas adoptadas para resolver conflitos resultantes da interdependência de instruções.

Exercício A

O processamento de uma instrução do MIPS tipicamente requer cinco passos:

1. Ler a instrução da memória (F = "Instruction Fetch").
2. Ler os registos enquanto descodifica a instrução. O formato das instruções do MIPS permite que a leitura dos registos e a descodificação sejam feitas em simultâneo (D="Decode").
3. Executar uma operação ou calcular um endereço (A="Execute" or "Arithmetic").
4. Aceder a um operando na memória de dados (M="Memory Access").
5. Escrever o resultado num registo (R="Write Back").

Na tabela I são dados os tempos de execução de cada estágio para diferentes classes de instruções:

Classe de Instruções	Instruction Fetch	Register Read	ALU Operations	Data Access	Register Write	Tempo Total
Load Word (lw)	200ps	100ps	200ps	200ps	100ps	800ps
Store Word (sw)	200ps	100ps	200ps	200ps		700ps
R-Format (add, sub, and, or, slt)	200ps	100ps	200ps		100ps	600ps
Branch (beq)	200ps	100ps	200ps			500ps

Tabela I. Tempo total de execução de cada instrução calculado a partir do tempo de cada componente, para um processador fictício.

a) Calcule o valor mínimo do ciclo de relógio para execução de uma instrução por ciclo sem *pipelining*.

b) Repita a questão anterior para execução usando *pipelining* (em cada ciclo de relógio é executado um passo do processamento da instrução).

As questões seguintes são baseadas no seguinte excerto de código:

```
addi  $t0, $t1, 100          # Linha 1
sw    $t3, 8($t0)            # Linha 2
lw    $t5, 0($t0)            # Linha 3
or    $t5, $t0, $t3          # Linha 4
lw    $t2, 4($t0)            # Linha 5
add   $t3, $t1, $t2          # Linha 6
```

c) Calcule o tempo total de execução do programa num processador sem pipelining (nas condições da alínea **b)**).

d) Considere a representação perfeitamente "*pipelined*" do mesmo programa representada abaixo (F="Instruction Fetch", D="Decode", A="Execute" ou "Arithmetic", M="Memory Access", R="Write Back"):

```
addi  $t0, $t1, 100          F D A M R
sw    $t3, 8($t0)              F D A M R
lw    $t5, 0($t0)                F D A M R
or    $t5, $t0, $t3                F D A M R
lw    $t2, 4($t0)                    F D A M R
add   $t3, $t1, $t2                    F D A M R
```

Desenhe setas para indicar a dependência de dados entre os diferentes níveis. A seta deve começar no nível em que os dados ficam pela primeira vez disponíveis (i.e. não necessariamente no nível R) e terminar onde os dados são absolutamente necessários (i.e. não necessariamente no nível D). Nota: pode resolver o exercício noutra folha que não a do enunciado se achar que isso permite uma maior clareza na resolução.

e) Com base no esquema que obteve acima, pode reparar que algumas das setas estão dirigidas no sentido do fluxo temporal (i.e. para a frente, "*forward*"), indicando que os valores em questão são necessários pouco *depois* de estarem disponíveis, que é o que se pretende. Outras setas não têm essa propriedade, e estão dirigidas no sentido contrário ao do fluxo temporal ("*backward*"). Use estas últimas setas para preencher a tabela seguinte. Mais especificamente, liste as instruções que têm uma dependência crítica do tipo "leitura-pré-escrita", que não pode ser resolvida com um mecanismo de "*forwarding*". Note que pode não necessitar de preencher todos os espaços em branco.

_____ é dependente da instrução 1
_____ é dependente da instrução 2
_____ é dependente da instrução 3
_____ é dependente da instrução 4
_____ é dependente da instrução 5

f) Refaça a tabela da alínea (d) por forma a incluir "*stalls*" que resolvam os conflitos que detectou. Um "*Stall*" significa que instruções NOP (no operation) são introduzidas entre as instruções pertinentes, sem nenhum hiato entre os passos de processamento.

g) Quantos ciclos de relógio são necessários para executar as 6 instruções numa CPU com pipelining explorando todos os possíveis mecanismos de "*forwarding*"? A que intervalo de tempo isso corresponde? Compare os resultados com os obtidos numa CPU sem *pipelining*.

h) Quantos ciclos de relógio são necessários para executar as 6 instruções numa CPU com pipelining mas sem "*forwarding*"?

i) Suponha agora que as instruções são executadas no seguinte caminho de dados em que alguns estágios levam mais que um ciclo de relógio:

```
F F D A A M M R
```

Isto é, "*instruction fetch*", "*execution*", e "*memory*" levam dois ciclos de relógio a completar, enquanto "*decode*" e "*write back*" apenas um ciclo. Quanto tempo demoram a executar as mesmas 6 instruções através deste caminho de dados *pipelined*? Suponha novamente que os mecanismos de "forwarding" estão disponíveis, e que são introduzidos "*nops*" nos sítios apropriados por forma a que o código seja processado correctamente.

Exercício B

Considere o seguinte excerto de código MIPS:

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t1)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

a) Determine todas as dependências que não podem ser resolvidas com mecanismos de *forwarding*. Introduza "*nops*" nos sítios apropriados, e indique quantos ciclos de relógio são necessários para executar o código em questão (assuma um caminho de dados do tipo: `F D A M R`).

b) Reordene as instruções por forma a evitar a ocorrência de "*stalls*". Quantos ciclos de relógio são agora necessários para executar o código?

Exercício C

Quantos ciclos de relógio são necessários para executar um 1 milhão de instruções numa CPU com *pipelining* (com caminho de dados com 5 passos, com tempos de execução conforme a tabela I) supondo que não ocorrem "*stalls*"? E numa CPU sem *pipelining* que execute uma instrução por ciclo? Qual dos casos é mais eficiente?

Exercício D

Em CPUs com *pipelining*, a ocorrência de *stalls* pode ser originada não apenas devido às dependências entre instruções (e.g. quando o processamento de uma instrução depende do resultado da instrução anterior), mas também em situações em que é necessário tomar um "salto" condicional (*branch*). Nestas circunstâncias, o processador não sabe *qual é a instrução seguinte* antes de a condição de teste do "*branch*" ter sido avaliada. Isto naturalmente afecta o desempenho do *pipelining*. Para ilustrar isto considere o seguinte excerto de código MIPS:

```
        addi $t5, $t5, 4
        add  $t0, $t4, $0
        addi $t1, $t1, 1
        add  $t2, $t1, $t0
        beq  $t2, $0, L1
        lw   $t6, 0($t5)
L1:     add  $t5, $t1, $t4
```

a) Calcule quantos ciclos de relógio são necessários para executar o código anterior (assuma um caminho de dados do tipo: `F D A M R`). Suponha que todos os mecanismos de "*forwarding*" estão disponíveis, e que em instruções de controlo (*beq*) o processador introduz "nops" até determinar qual é a instrução seguinte.

b) Um processo ligeiramente mais eficiente de abordar os problemas de "*stall*" que ocorrem num *branch* é supor que a instrução a seguir ao salto ainda é executada seja qual for a avaliação da condição de salto. Esta técnica designa-se por «*delayed branch*». Para além disso pode-se adicionar hardware especializado para fazer a comparação envolvida no salto no estágio 2. Com base nestes mecanismos rearranje o código para tirar partido destas técnicas e calcule novamente o número de ciclos de relógio necessários para executar o código.