Arquitectura de Computadores II

3º Ano

Paralelismo ao Nível da Instrução

(execução encadeada de instruções e super-escalaridade)

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

Junho 2004

Índice

Índice de Figuras

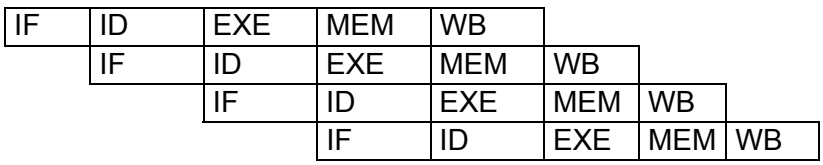
1 Introdução

A arquitectura dos processadores modernos baseia-se fortemente na capacidade em executar várias instruções em cada ciclo de relógio, o que normalmente se designa por paralelismo ao nível da instrução. Esta designação surge porque as instruções são executadas em paralelo. Duas técnicas são frequentemente utilizadas para a execução de instruções em paralelo:

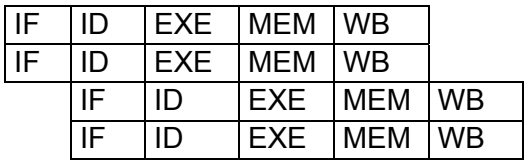
- Execução encadeada de instruções – cada instrução é executada em várias fases, sendo a execução de várias instruções sobreposta, como numa linha de montagem; as várias instruções executam em paralelo, mas em **fases diferentes**;
- Execução super-escalar de instruções – as instruções são executadas em paralelo, o que envolve a duplicação de unidades funcionais para suportar a combinação de instruções pretendida.

Estas duas técnicas são geralmente combinadas, sendo ambas utilizadas na arquitectura dos processadores modernos. A Figura 1 apresenta uma comparação das duas técnicas para um processador que executa as instruções em 5 fases:

- Busca da instrução (IF)
- Leitura dos registos e descodificação das instruções (ID)
- Execução da operação ou cálculo de endereço (EXE)
- Acesso ao operando em memória (MEM)
- Escrita do resultado em registo (WB)



a)



b)

Figura 1 – a) Execução encadeada de instruções b) execução super-escalar

O tempo de execução de um programa é dado por:

$$Texe = \#instruções \times CPI \times Tcc$$

A execução de instruções em cadeia permite reduzir a duração do ciclo de relógio (Tcc) do processador, ou seja, aumentar a sua frequência. Por outro lado, a execução super-escalar de instruções aumenta o número de instruções realizadas em cada ciclo (IPC) de relógio, ou seja, diminui o CPI. Note que qualquer uma destas duas alternativas permite reduzir o tempo de execução da aplicação.

Os próximos dois capítulos apresentam, respectivamente, a execução encadeada de instruções e a execução super-escalar. O capítulo 5 apresenta algumas arquitecturas de processadores alternativas e o último capítulo apresenta um conjunto de exercícios.

2 Execução encadeada de instruções

2.1 Princípios básicos

A execução encadeada de instruções baseia-se na sobreposição da execução de instruções por forma a que estas executem em paralelo. Para tal, as instruções são executadas em várias fases (também designadas por estágios) e, num determinado ciclo, as diferentes instruções executam fases diferentes, tal como numa linha de montagem. A Figura 2 apresenta um exemplo de execução encadeada de instruções, para uma arquitectura com cinco estágios.

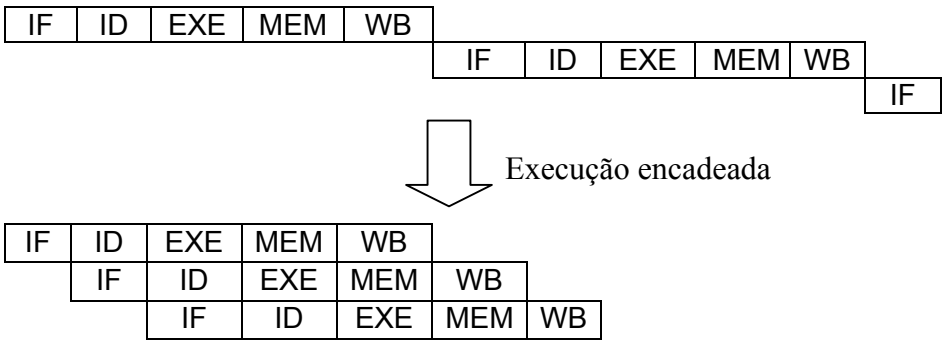


Figura 2 – Execução encadeada de instruções

Teoricamente a introdução da execução encadeada de instruções num arquitectura possibilita um ganho igual ao número de fases de execução, uma vez que o número máximo de instruções em execução num determinado ciclo de relógio é igual ao número de fases. No entanto, vários factores contribuem para que tal não se verifique.

Em primeiro lugar, a duração de cada ciclo relógio será igual ao estágio mais longo, adicionando a sobrecarga para a passagem de informação entre estágios:

Tcc *pipeline* = max [estágio*i*] + sobrecarga de passagem de informação entre estágios

Por esta razão os vários estágios deverão ter uma carga equivalente, por forma a cada fase de execução da instrução demore sensivelmente o mesmo tempo, ou seja, ao projectar uma arquitectura que suporte a execução encadeada de instruções os vários estágios deverão ter uma carga equivalente (tal como acontece numa linha de montagem). Adicionalmente não existe vantagem em ter instruções que demorem menos fases a executar, uma vez que a execução encadeada permite completar uma instrução em cada ciclo de relógio. Aliás, esta característica pode mesmo gerar conflitos pela utilização de uma dada unidade funcional. Por exemplo, na Figura 3 a instrução *add* iria efectuar a escrita do resultado em registo no mesmo ciclo de relógio que o *lw*, o que iria provocar um conflito na escrita dos dados no banco de registos, uma vez que só existe uma porta de acesso para escrita no banco de registo.

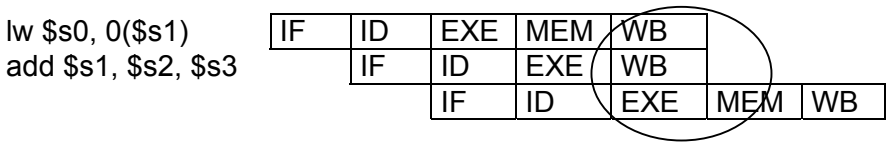


Figura 3 – Execução encadeada sem balanceamento das fases

Em segundo lugar a execução encadeada de instruções aumenta o débito de instruções (i.é., instruções completadas por unidade de tempo), mas não diminui o tempo necessário para executar cada instrução. Por essa razão o número de ciclos necessário para completar a primeira instrução é igual ao número de estágios, no entanto, numa situação ideal, cada uma das restantes instruções apenas requer um ciclo de relógio adicional (Figura 4).

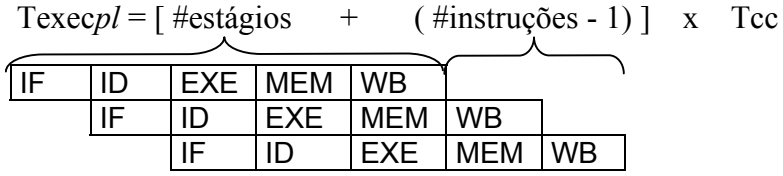


Figura 4 – Tempo de execução de instruções em cadeia

Em terceiro lugar, a execução encadeada de instruções origina três tipos de problemas (designados por anomalias):

1. estruturais – o hardware não suporta a combinação de instruções em execução
2. controlo – a execução da instrução seguinte depende de uma decisão baseada num resultado anterior
3. dados – a execução da instrução requer dados produzidos por instruções anteriores

Cada um destes tipos de anomalias é apresentado em detalhe nas secções seguintes.

2.2 Anomalias estruturais

As anomalias estruturais surgem quando uma combinação de instruções não pode ser executada de forma encadeada porque o hardware não suporta essa combinação. Normalmente a resolução deste tipo de anomalias leva à duplicação das unidades funcionais. Por exemplo, na arquitectura MIPS com cinco estágio são necessárias duas unidades de acesso à memória, uma para efectuar a busca das instruções e a outra para suportar as operações de *load* e de *store*. Por esta razão o *datapath* analisado para suporte à execução encadeada de instruções tem algumas semelhanças com um *datapath* que executa as instruções num só ciclo, tendo várias unidades duplicadas. Recorde-se que na execução de instruções em vários ciclos já não são necessárias as unidades duplicadas, uma vez que cada instrução pode utilizar uma mesma unidade funcional em ciclos diferentes.

Como exemplo de uma arquitectura com anomalias estruturais considere-se a arquitectura MIPS com 5 estágios, mas apenas com uma unidade acesso à memória. Nesta arquitectura, sempre que for executada uma instrução de acesso à memória não pode ser executada uma busca de uma instrução nesse mesmo ciclo, uma vez que ambas as fases de execução utilizam a mesma unidade. Assim, sempre que ocorrer a situação anterior terá que ser gerado um *stall* (i.é., introdução de bolhas em alguns estágios da cadeia de execução) por forma a que não seja executado um IF nesse ciclo. A Figura 5 apresenta um exemplo de um *stall* devido a anomalias estruturais.

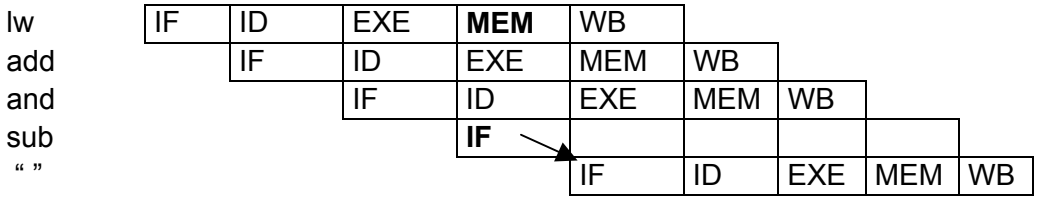


Figura 5 – *stall* devido a uma anomalia estrutural

As anomalias estruturais podem ocorrer devido a vários factores, nomeadamente, devido a um número insuficiente de registos internos do processador, de portas para acesso ao banco de registos, de portas de acesso à memória ou de unidades funcionais.

2.3 Anomalias de dependências de dados

As anomalias devido a dependências de dados surgem em situações onde uma instrução utiliza valores produzidos por instruções anteriores. Nomeadamente, numa arquitectura MIPS com cinco estágios, o resultado da execução de uma instrução (i.é., novo valor de um registo) só fica disponível após a fase WB, no entanto, o banco de registos é lido durante a fase de ID, o que pode originar a leitura do valor do registo antes de este ter sido actualizado com o novo valor. A Figura 6 apresenta um diagrama multi-ciclo resultante da execução do seguinte programa MIPS:

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

No exemplo da Figura 6 o novo valor do registo $2, calculado na instrução *sub* apenas fica visível após o ciclo 5, o que implica que as instruções *and, or,* e *add* irão ler o valor errado do banco de registos. Note-se que a instrução *sw* já lê o valor correcto de $2, uma vez que no ciclo 6 a instrução *sub* já fez o WB.

Existem 3 soluções para as dependências de dados:

1. empatar a cadeia de execução
2. delegar no compilador
3. encaminhar os dados.

A primeira solução implica um mecanismo que detecte as dependências de dados e que efectue *stall* das instruções seguintes enquanto o valor não está disponível. No exemplo anterior seriam necessários 3 ciclos de *stall*. Esta alternativa é equivalente à introdução de 3 instruções de *nop* para espaçar as instruções com dependências:

```
sub  $2, $1,  $3
nop
nop
nop
and $12, $2,  $5
or  $13, $6,  $2
add $14, $2,  $2
sw  $15, 100($2)
```

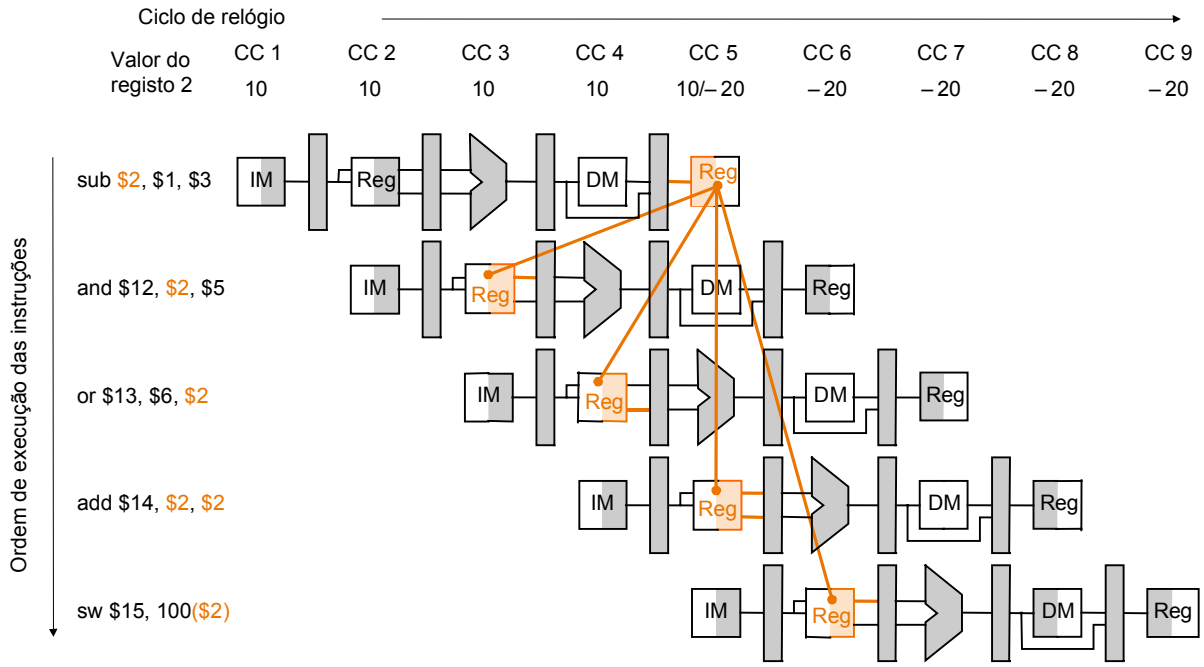
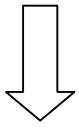


Figura 6 – Dependências de dados

Uma forma de reduzir o número de *stall* devido às dependências de dados, nesta arquitectura MIPS com 5 estágios, consiste em efectuar as escritas em registo na primeira metade do ciclo e efectuar as leituras na segunda metade do ciclo. Desta forma é possível num mesmo ciclo escrever num registo e ler o valor actualizado. Utilizando esta estratégia no exemplo anterior já só seriam necessários 2 ciclos de *stall*.

A segunda estratégia para lidar com as anomalias de dados consiste em delegar no compilador essa responsabilidade. O exemplo da Figura 7 apresenta um caso em que o compilador reorganiza a ordem das instruções por forma a remover os *stalls*. Contudo esta estratégia tem alguns inconvenientes, nomeadamente, obriga o compilador a conhecer a arquitectura do processador, embora existam optimizações em que tal pode não ser necessário. Esta reorganização de instruções tem algumas semelhanças com o escalonamento dinâmico efectuado pelos processadores com arquitecturas super-escalares, arquitecturas que irão ser analisada no capítulo seguinte.

```
                              # reg $t1 contém &v[k]
lw $t0, 0($t1)                # $t0 (temp) = v[k]
lw $t2, 4($t1)                # $t2 = v[k+1]
sw $t2, 0($t1)                # v[k] = $t2
sw $t0, 4($t1)                # v[k+1] = $t0
```



```
lw $t0, 0($t1)                # $t0 (temp) = v[k]
lw $t2, 4($t1)                # $t2 = v[k+1]
sw $t0, 4($t1)                # v[k+1] = $t0
sw $t2, 0($t1)                # v[k] = $t2
```

Figura 7 – Troca da ordem das instruções para evitar *stalls*

A terceira estratégia para lidar com as dependências de dados consistem em criar atalhos na arquitectura por forma a que os valores possam ser passados de uma instrução para uma das instruções seguintes sem passar pelo banco de registos. No caso da arquitectura MIPS com 5 estágios as instruções necessitam efectivamente dos valores apenas na fase EXE e não na fase ID. Por exemplo a instrução *and* só utiliza o registo $2 durante a fase EXE para efectuar a operação AND na ALU. Nesta mesma arquitectura os valores são calculados durante a fase EXE, o que implica que podem ser disponibilizados às instruções seguintes após a fase EXE. Todas as instruções MIPS do TIPO-R suportadas pelo *datapath* analisado obedecem a esta regra. No entanto as instruções de *lw* e de *sw* têm um comportamento diferente como iremos ver posteriormente.

A Figura 8 apresenta o exemplo anterior agora com encaminhamento de dados. Neste caso já não é necessária a introdução de *stalls* na cadeia de execução, uma vez que o atalho permite passar um valor a uma das instruções seguintes, logo a partir do ciclo em que o valor é calculado. Neste exemplo, o novo valor do registo $2 é calculado durante o ciclo 3, o que implica que esse valor pode ser passado directamente à ALU quando for necessário por uma das instruções seguintes. Note-se que neste exemplo também se assume que a arquitectura MIPS efectua a escrita dos valores no banco de registos na primeira metade do ciclo e que efectua as leituras na segunda metade (ciclo 5).

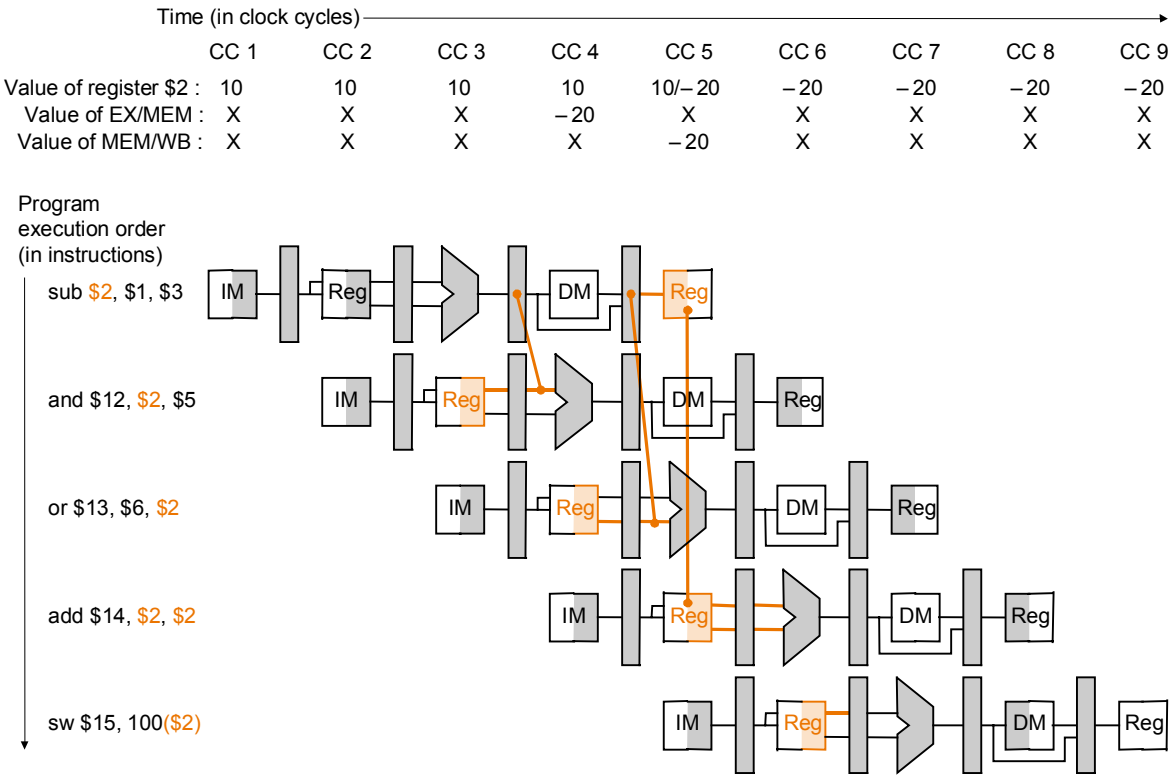


Figura 8 – Encaminhamento de dados

Embora o encaminhamento de dados seja essencial numa arquitectura com execução encadeada de instruções para garantir um bom desempenho ele não resolve todas as anomalias derivadas de dependências de dados. Designadamente, a instrução *lw* apenas disponibiliza os valor após a fase de MEM o torna impossível efectuar o encaminhamento dos dados para a instrução seguinte (Figura 9). Neste caso terá ser utilizada uma das soluções anteriores: efectuar um *stall* da cadeia de execução ou espaçar o *lw* das instruções seguinte. Note-se também que caso a instrução seguinte seja o *sw* do valor lido pelo *lw* já poderia ser efectuado o encaminhamento de dados, uma vez que o *sw* apenas necessita do

valor na fase MEM. Neste caso seria necessário um novo atalho especificamente para este caso.

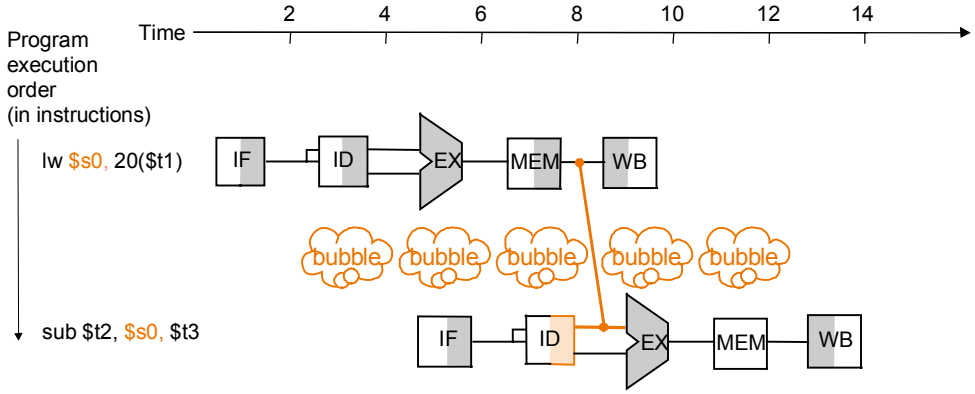


Figura 9 – Limitação do encaminhamento de dados

2.4 Anomalias de dependências de controlo

As anomalias devido a dependências de controlo surgem quando a execução da instrução seguinte depende de uma decisão tomada numa instrução anterior, designadamente, uma instrução de salto pode provocar uma alteração na sequência de execução das instruções, não sendo possível efectuar o IF da instrução seguinte antes de ter calculado o novo valor do PC. Existem duas alternativas para lidar com as dependências de controlo:

1. empatar a execução das instruções até ser conhecido o novo valor do PC
2. prever o salto continuando a execução no endereço previsto.

No segundo caso é necessário assegurar que os resultados das instruções executadas de forma especulativa só são tornados visíveis se a previsão for correcta.

Numa arquitectura MIPS com 5 estágios, onde o valor do PC só é escrito no fim da fase MEM a penalização introduzida pela estratégia de *empatar a execução* é de 3 ciclos, uma vez que não se pode fazer o IF antes da instrução de salto ter completado a fase MEM (Figura 10). Uma forma de reduzir o número de ciclos de *stall* consiste em antecipar a resolução dos saltos (i.é., o cálculo do novo valor do PC). Na arquitectura MIPS pode-se resolver o salto durante a fase ID, bastando para tal introduzir uma unidade que durante esta fase calcule o novo valor do PC e que compare os dois valores lidos do banco de registos. Neste caso apenas é necessário um ciclo de *stall* sempre que é executado um salto. Note-se que, no entanto, esta alternativa coloca mais sobrecarga na fase de ID, porque são realizadas mais operações durante esta fase.

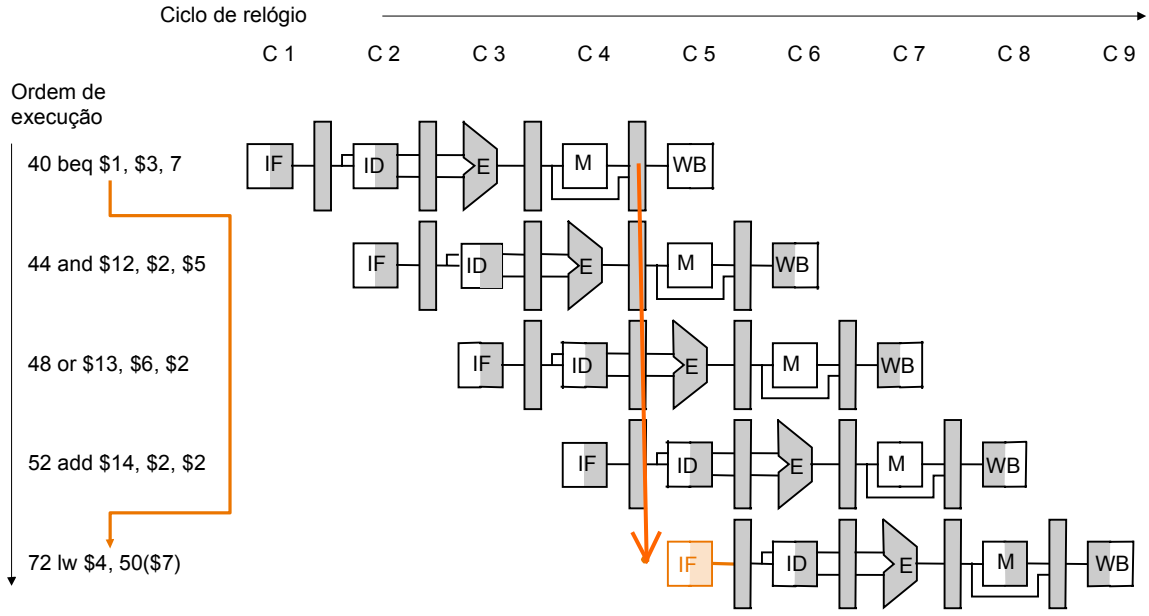


Figura 10 – *stall* devido a uma anomalia de controlo

Uma estratégia mais eficiente para reduzir as penalizações devido a anomalias de controlo consiste em prever se o salto irá ser tomado ou não. Desta forma apenas irá existir uma penalização se a previsão não estiver correcta. A forma mais simples de implementar a previsão de saltos é efectuar uma previsão estática de saltos, ou seja, prever se o salto é tomado sem considerar informação sobre a execução desse mesmo programa. Por exemplo, os saltos podem ser sempre considerados como não tomados, efectuando sempre a busca da instrução seguinte. Caso se venha a verificar que o salto foi mal previsto então será efectuado o IF da instrução correcta e introduzidos os *stalls* necessários na cadeia de execução. A Figura 11 ilustra um exemplo desta situação para uma cadeia de execução em que os saltos são resolvidos durante a fase ID.

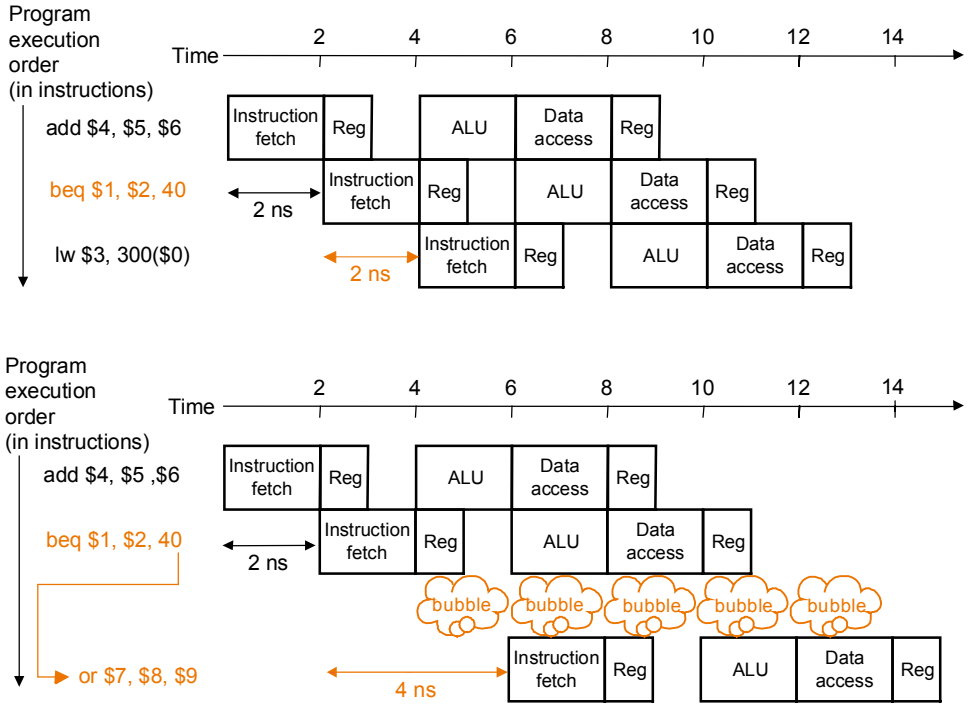


Figura 11 – Previsão estática de saltos

A previsão estática de saltos tende obter na ordem de 50% de previsões correctas. A previsão dinâmica dos saltos permite obter mais de 90% de previsões acertadas, para tal é utilizada a história dos saltos anteriores para prever qual a direcção do próximo salto. Esta previsão pode variar ao longo da execução do programa, um função dos dados do programa.

A história dos saltos anteriores é normalmente armazenada numa tabela do tipo *cache*. Se essa tabela tiver apenas um bit por endereço, tal implica o próximo salto vai ser previsto como tomando a mesma direcção que a execução anterior do mesmo salto. Genericamente a previsão de saltos pode depender de TODOS os saltos efectuados anteriormente, mas normalmente utiliza-se a história do último salto ou dos dois últimos saltos. Esta segunda alternativa requer uma tabela com 2 bits por endereço de salto e é significativamente mais eficiente que a utilização de apenas 1 bit, essencialmente porque esta falha sempre duas vezes nos ciclos *for*: na entrada (resultante da última execução) e na saída (fim do ciclo). O esquema com dois bit só altera a previsão quando erra duas vezes (Figura 12) seguidas, o que, em geral, melhora a percentagem de previsões acertadas.

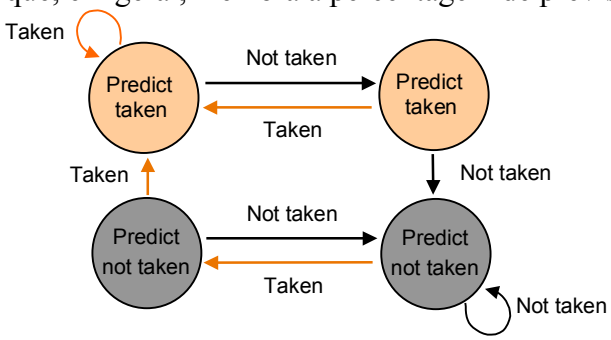


Figura 12 – Previsão de saltos com 2 bit por endereço de salto

Para efectuar a previsão de saltos é necessário armazenar dois tipos de informação, uma vez que a previsão pode ser efectuada mesmo antes do salto ser descodificado: informação sobre a história do salto (se foi tomado ou não) e informação sobre o endereço de salto (qual o endereço da próxima instrução caso o salto seja tomado). **A tabela com a história dos saltos** contém o endereço da instrução de salto e o(s) bit(s) correspondentes à informação sobre os últimos saltos realizados. Esta tabela também permite verificar se a instrução num determinado endereço é um salto. Se um endereço não se encontrar nessa tabela a instrução não é um salto ou não existe informação sobre esse salto. A previsão de saltos também requer uma **tabela com os endereços de salto tomados** anteriormente, uma vez que o endereço de salto só é calculado em EXE. As duas tabelas têm uma implementação idêntica às *caches* (são constituídas por memória associativa) e podem ser implementadas numa só tabela que pode consultada durante cálculo do próximo valor do PC ou no início da fase IF (Figura 13): Antes de efectuar a busca da instrução o valor do PC é pesquisado na tabela, se não existir informação na tabela relativa a esse endereço a próxima instrução pode não ser um salto (caso se venha a verificar que é um salto pode-se ainda utilizar uma previsão estática). Se existir informação sobre o salto na tabela então o próximo valor do PC (PC previsto) é lido da tabela. Note que a tabela com a história de saltos deve ser actualizada sempre que a execução de um salto é concluída.

A tabela de historia de saltos tem uma capacidade limitada, normalmente entre 512 e 4096 entradas, implicando um esquema de substituição das linhas da tabela. Adicionalmente, para reduzir o número de bits necessários, cada entrada da tabela pode-se apenas guardar os bits menos significativos do endereço. Tal implica que a informação sobre dois saltos em endereços distintos possa ser armazenada na mesma entrada da tabela. Estes dois factores contribuem para diminuir a percentagem de previsões acertadas.

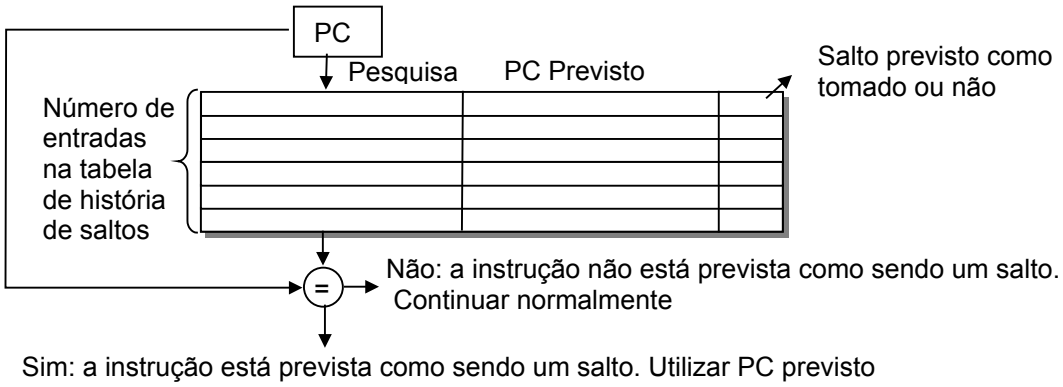


Figura 13 – Tabela de previsão de saltos

2.5 Análise de desempenho da execução encadeada

O tempo de execução de um programa pode ser calculado multiplicando o número de instruções executadas (#instruções) pelo número médio de ciclos necessário à execução de cada instrução (CPI), multiplicado pela duração de cada ciclo (Tcc):

Texec = #instruções x CPI x Tcc

O número de ciclos médio (CPI) pode ser calculado dividindo o número total de ciclos necessários para a execução de instruções pelo número de instruções executadas. No caso da execução encadeada de instruções o número de ciclos é (considerando agora também os vários tipos de anomalias):

#ciclos = #estágios + (#instruções – 1) + *stalls* anomalias estruturais + *stalls* anomalias de dados + *stalls* anomalias de controlo

A expressão anterior é utilizada quando se conhece o programa a executar. Contudo quando um programa contém um elevado número de instruções torna-se mais viável o uso de informação sobre o perfil de execução. Nesse caso o CPI médio pode ser calculado com a seguinte expressão:

CPI = 1 (CPI ideal) + CPI *stalls* anomalias estruturais + CPI *stalls* anomalias de dados + CPI *stalls* anomalias de controlo

Ou seja, o CPI é igual ao CPI ideal mais o acréscimo de CPI originado pelas anomalias estruturais, mais o acréscimo de CPI originado anomalias de dados, mais o acréscimo de CPI originado pelas anomalias de controlo.

2.5.1 Exemplo 1: cálculo do número de ciclos

Pretende-se calcular o número de ciclos necessário para executar o seguinte programa em *assembly* do MIPS:

```
        add $s2, $0, $0
        add $s0, $0, -256
ler:    lw $t0, 0x700 ($s0)
        add $s2, $s2, $t0
        addi $s0, $s0, 4
        beq $s0, $0, ler
```

Neste caso vamos considerar que toda a informação necessária se encontra em *cache*, que o programa está a ser executado numa arquitectura MIPS com 5 estágios (ver figura em anexo), sem encaminhamento de dados e que os registos são escritos na primeira metade do ciclo e que as leituras dos registos ocorrem na segunda metade do ciclo (excepto a leitura de PC e IF)

De acordo com os dados anteriores cada dependência de dados entre instruções origina um *stall* de dois ciclos e cada dependência de controlo origina um *stall* de três ciclos. O número de instruções executadas é igual a 2 + 4*(256/4). Neste programa existem três dependências de dados que empatam a execução das instruções:

1) *add **$s0**, $0, -256* e *lw $t0, 0x700 **($s0)***
2) *lw **$t0**, 0x700 ($s0)* e *add $s2, $s2, **$t0***
3) *addi **$s0**, $s0, 4* e *beq **$s0**, $0, ler*.

A primeira dependência apenas ocorre uma vez, enquanto as outras duas são executadas 64 vezes. O número de *stalls* devidas a anomalias de dados é igual a:

2 (caso *1)*) + 2 x 64 (caso *2* e *3)*) = 130 ciclos

No programa anterior apenas existe uma dependência de controlo proveniente da instrução beq. Neste caso, como a instrução é executada 64 vezes e o *stall* é de três ciclos teremos um número de ciclos igual a 3 x 64.

Finalmente, o número total de ciclos necessário para executar o programa anterior é dado por:

5 + (2 + 4 x 64 – 1) + (2 + 2 x 2 x 64) + (3 x 64) = 712 ciclos

Note que o CPI médio é dado por #ciclos / # instruções, que neste caso é igual a 712/256 = 2,78

2.5.2 Exemplo 2: cálculo do CPI médio

Considere um programa com a seguinte mistura de instruções:

49% Tipo R	22% *Load*	11% *Store*	16% *Branch*	2% *Jump*

Pretende-se calcular o CPI médio na execução deste programa numa arquitectura MIPS com 5 estágios e com encaminhamento de dados (figura em anexo), sabendo que 50% dos valores de *loads* são utilizados na instrução seguinte, que 75% dos saltos relativos são previstos correctamente os saltos absolutos têm sempre um *stall* de um ciclo.

Pelos dados anteriores podemos concluir que acréscimo de CPI devido a anomalias de dados é originado apenas pelas instruções de *Load* que constituem 22% das instruções, uma vez que existe encaminhamento de dados. Sabemos também que em apenas 50% dos

casos a instrução irá originar um *stall* e que esse *stall* é de um ciclo (MIPS de 5 estágios com encaminhamento de dados), logo, o acréscimo de CPI devido a anomalias de dados é 0,22 x 0,50 x 1 = 0,11. Este número indica que em 100 instruções do programa existem, em média, 11 *stalls* originados pelas anomalias de dados.

O acréscimo de CPI originado por anomalias de controlo provêm de duas fontes: *branch* (salto relativo ao PC) *jump*. Para o primeiro caso, temos um *stall* sempre que o salto é previsto erradamente, ou seja, em 25% das instruções de *branch*. Uma vez que os saltos condicionais são resolvidos na fase MEM cada *stall* durará 3 ciclos. Assim, este tipo de saltos provoca um acréscimo de CPI de 0,16 x 0,25 x 3 = 0,12. A instrução de *Jump* ocorre em 2% das instruções e provoca sempre um ciclo de *stall*, tal como referido no enunciado, logo, origina um acréscimo do CPI de 0,02 x 1 (100% dos casos) x 1 = 0,02.

Neste exemplo o CPI médio é então dado por:

CPI médio = 1 + 0,11 + (0,12 + 0,02) = 1,25

Uma formulação alternativa consiste em calcular a CPI médio para cada classe de instruções, sendo o CPI total dado pela soma pesada do CPI de cada classe instruções, tal como efectuado para a variante do MIPS com execução de instruções em vários ciclos de relógio. Assim o CPI médio será:

CPI médio = 0,49 x CPI *tipo-R* + 0,22 CPI *load* + 0,11 x CPI *store* + 0,16 x CPI *branch* + 0,02 x CPI *jump*

O CPI das instruções do *tipo-R* e de *store* é igual a 1, uma vez não originam *stalls* nesta arquitectura. O CPI da instrução de *load* é igual a 1 quando não há *stalls* e é igual a 2 quando a dependência provoca um *stall* (50% dos *loads*). Assim, o CPI médio do *load* é 0,5 x 1 + 0,5 x 2 = 1,5. Seguindo o mesmo raciocínio, o CPI da instrução de *branch* é 0,75 x 1 + 0,25 x 4 = 1,75 e o CPI do *jump* é 2. Assim o CPI médio é dado por:

CPI médio = 0,49 x 1 + 0,22 x 1,5 + 0,11 x 1 + 0,16 x 1,75 + 0,02 x 2 = 1,25

Esta expressão é equivalente à obtida anteriormente:

0,49 x 1 + 0,22 x 1,5 + 0,11 x 1 + 0,16 x 1,75 + 0,02 x 2 =
0,49 x 1 + 0,22 x (1 + 0,5) + 0,11 x 1 + 0,16 x (1 + 0,75) + 0,02 (1 + 1) =
(0,49 + 0,22 + 0,11 + 0,16 + 0,02) + (0,22 x 0,5) + (0,16 x 0,75 + 0,02 x 1) =
1 + 0,11 + (0,12 + 0,04)

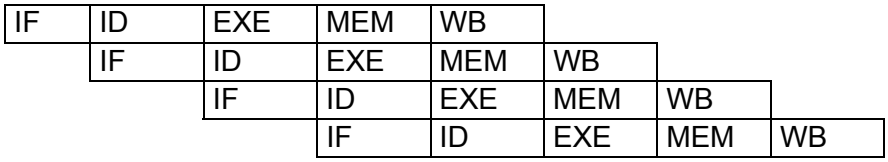
2.6 Compromissos na execução encadeada

Genericamente, um aumento do número de estágios da cadeia de execução permite teoricamente uma diminuição na mesma percentagem do período do relógio e consequentemente um aumento da frequência de relógio. No entanto, devido à sobrecarga da passagem de informação entre estágios e à necessidade de balanceamento do trabalho efectuado em cada estágio os ganhos obtidos são geralmente menores. Por exemplo, na

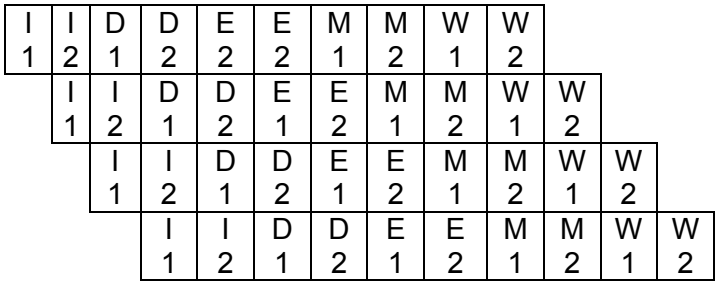
arquitectura do Pentium 4 existem dois estágios essencialmente para passar a informação entre unidades funcionais.

Um aumento do número de estágios da cadeia de execução tende a aumentar o número de ciclos de *stall* devidos às anomalias, podendo mesmo originar uma degradação de desempenho, uma vez que o CPI médio aumenta. Normalmente ao aumentar o número de estágios também são adicionados mecanismos para reduzir o número de ciclos de *stalls*.

Considere-se um exemplo em que o número de estágios da arquitectura MIPS é duplicado, desdobrando cada estágio em 2 (Figura 14). Considere-se também que se consegue obter estágios completamente balanceados, de modo a que o ciclo de relógio da arquitectura seja reduzido para metade.



a) MIPS 5 estágios



b) MIPS 10 estágios

Figura 14 – Aumento do número de estágios do MIPS

Se considerarmos que os valores são necessários no início de EXE e que os valores calculados são disponibilizados no fim da fase EXE e que os saltos são resolvidos na fase EXE verificam-se as seguintes penalizações (Figura 15):

a. um ciclo quando o valor da instrução é utilizado na instrução seguinte
b. três ciclos quando o valor de um *load* é utilizado na instrução seguinte
c. cinco ciclos nas instruções de salto

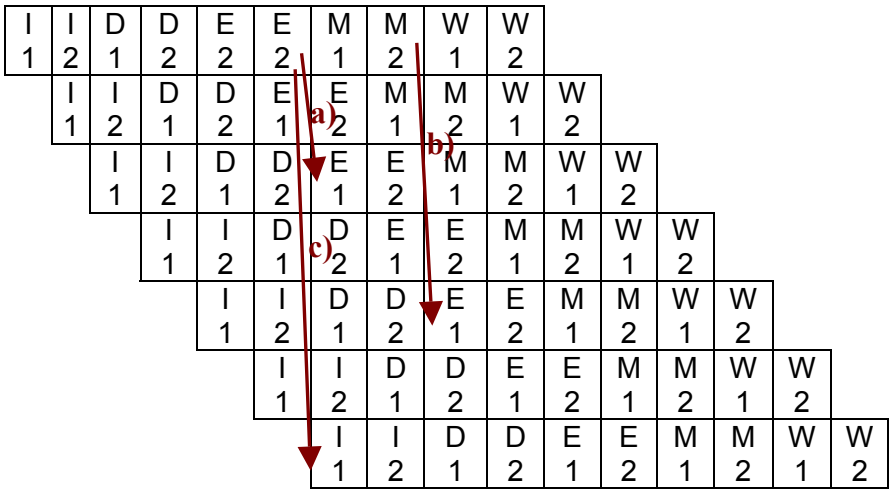
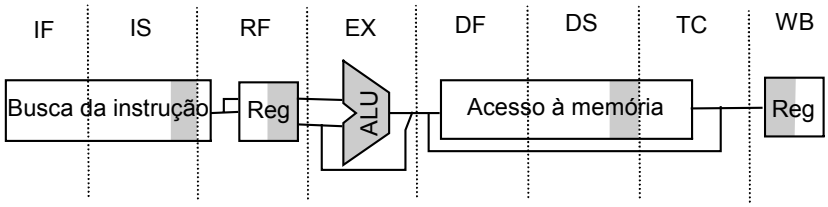


Figura 15 – Ciclos de *stall* na arquitectura MIPS com 10 estágios

Neste caso, apesar de se considerar uma unidade de encaminhamento de dados, a utilização de um valor produzido pela instrução anterior origina um ciclo de *stall*. Note-se que a implementação do encaminhamento de dados nesta arquitectura é bastante mais complexa, uma vez que são necessários mais atalhos do que na versão com 5 estágios. Note-se também que mesmo resolvendo aos saltos na fase EXE existem 5 ciclos de *stall* nos saltos não previstos correctamente. Uma solução para reduzir este impacto poderia passar pela melhoria das previsões dos saltos.

2.6.1 Exemplo: comparação de arquitecturas MIPS

O exemplo da secção anterior é apenas ilustrativo do que acontece quando se incrementa o número de estágios da cadeia de execução. Um exemplo mais realista consiste na comparação do MIPS R2000, que possui arquitectura com 5 estágios com o MIPS R4000 que incrementou este número para 8. As fases adicionais são essencialmente desdobramentos das fases de acesso à memória, ou seja, IF e MEM (Figura 16).



IF – Primeira metade da busca da instrução (selecção do PC)
IS – Segunda metade da busca da instrução, completar o acesso
RF – Descodificação e leitura do valor dos registos, detecção de *hit*
EX – Execução, incluindo a resolução dos saltos
DF – Busca dos dados, primeira metade do acesso a *cache*
DS – Segunda metade do acesso a *cache*
TC – *Tag check*, determinar se o acesso foi um *hit*
WB- Escrita dos resultados em registo para *load* e registo-registo

Figura 16 – Cadeia de execução do MIPS R4000

Nesta arquitectura, o banco de registos é lido na segunda metade de RF e é escrito na primeira metade de WB (tal como acontece no MIPS R2000). Uma vez que existe encaminhamento de dados, as instruções que envolvem cálculos na ALU necessitam dos valores no início de EXE e esses valores ficam disponíveis logo após essa fase, excepto os *loads* que disponibilizam os valores no final de DS. Note-se que nesta arquitectura para implementar o encaminhamento de dados não necessários atalhos de quatro estágios (EX/DF, DF/DS, DS/TC e TC/WB) enquanto no R2000 apenas eram necessários atalhos de dois estágios.

Na arquitectura do R4000 os *stalls* devidos a instruções de saltos resultam em 3 ciclos de penalização, quando o salto é previsto erradamente, enquanto os *loads* originam 2 ciclos de penalização, quando o valor é utilizado pela instrução seguinte (Figura 17).

2.6.1.1 Comparação de desempenho

Pretende-se comparar o desempenho das duas arquitecturas MIPS referidas anteriormente. Para tal assume-se que todos os dados necessários estão em *cache* e que se está a executar um programa com a seguinte mistura de instruções (recolhida para o programa gcc), que existem 50% de *stall* em *loads* e que 75% dos saltos são previstos correctamente):

Tipo de Instrução	Frequência
Tipo R	49%
Load	22%
Store	11%
Branch	18%

MIPS*5* (resolução dos saltos em EXE, s/ optimizar escritas em PC):

CPI*load* (50% de situações de *stall*) = 0,5 x 1 + 0,5 x 2= 1,5
CPI*branch* (75% previsões correctas) = 0,75 x 1 + 0,25 x 3 = 1,5

CPI*5* = 0,49 x 1 + 0,22 x 1,5 + 0,11 x 1 + 0,18 x 1,5 = 1,2

MIPS*8*:

CPI*load* (50% de situações de *stall*) = 0,5 x 1 + 0,5 x 3 = 2,0
CPI*branch* (75% previsões correctas) = 0,75 x 1 + 0,25 x 4 =1,75

CPI*8* = 0,49 x 1 + 0,22 x 2 + 0,11 x 1 + 0,18 x 1,75 = 1,36

Comparação mantendo a frequência de relógio:

CPI*5* / CPI*8* = 1,36 / 1,2 = 0,88 (degradação de 12%)

Neste exemplo a arquitectura com 8 estágios apresenta pior desempenho, o que é originado pelo aumento de número de ciclos de *stall* originados por um maior número de estágios. No entanto, geralmente um aumento de número de estágios visa uma diminuição do ciclo de relógio, uma vez que em cada fase pode ser efectuada uma menor quantidade de trabalho ou cada fase pode resultar de uma optimização das unidades funcionais correspondentes (mesmo que a complexidade global da arquitectura possa aumentar). No exemplo seguinte são comparadas as duas versões da arquitectura, assumindo que na arquitectura com 8 estágios o ciclo de relógio é reduzido em 50%.

Comparação aumentado a frequência de relógio em 50% (i.é., Tcc*8* = 1,5 x Tcc*5*):

Ganho = #I x CPI*5* xTcc*5* / #I x CPI*8* x Tcc*8*
= 1,5 x CPI*5* / CPI*8* = 1,5 x 1,2 / 1,36 = 1,32x

Quando se aumenta o número de estágios da cadeia de execução convém introduzir melhorias na arquitectura para não aumentar o significativamente o CPI médio, devido ao aumento do número de ciclos de *stalls* originados por uma maior profundidade da cadeia de execução. O próximo exemplo assume que simultaneamente com o aumento do número de estágios são introduzidas melhorias na unidade de previsão de saltos, passando a prever correctamente 90% dos saltos, e na unidade de acesso à memória, diminuindo um ciclo no acessos à memória que provocam *stalls*.

Comparação melhorando a previsão de saltos para 90% e reduzindo um ciclo ao impacto dos *stall* nos *load*:

CPI*load* (50% de situações de *stall*) = 0,5 x 1 + 0,5 x 2 = 1,5

CPI*branch* (90% previsões correctas) = 0,9 x 1 + 0,1 x 4 = 1,4

CPI*8a* = 0,49 x 1 + 0,22 x 1,5 + 0,11 x 1 + 0,18 x 1,4 = 1,18
Ganho = 1,5 x 1,2 / 1,18 = 1,53x

Note-se que apenas com a melhoria de duas unidades funcionais o CPI médio foi reduzido de 1,36 para 1,18, o que se traduz num ganho em termos do tempo de execução das aplicações.

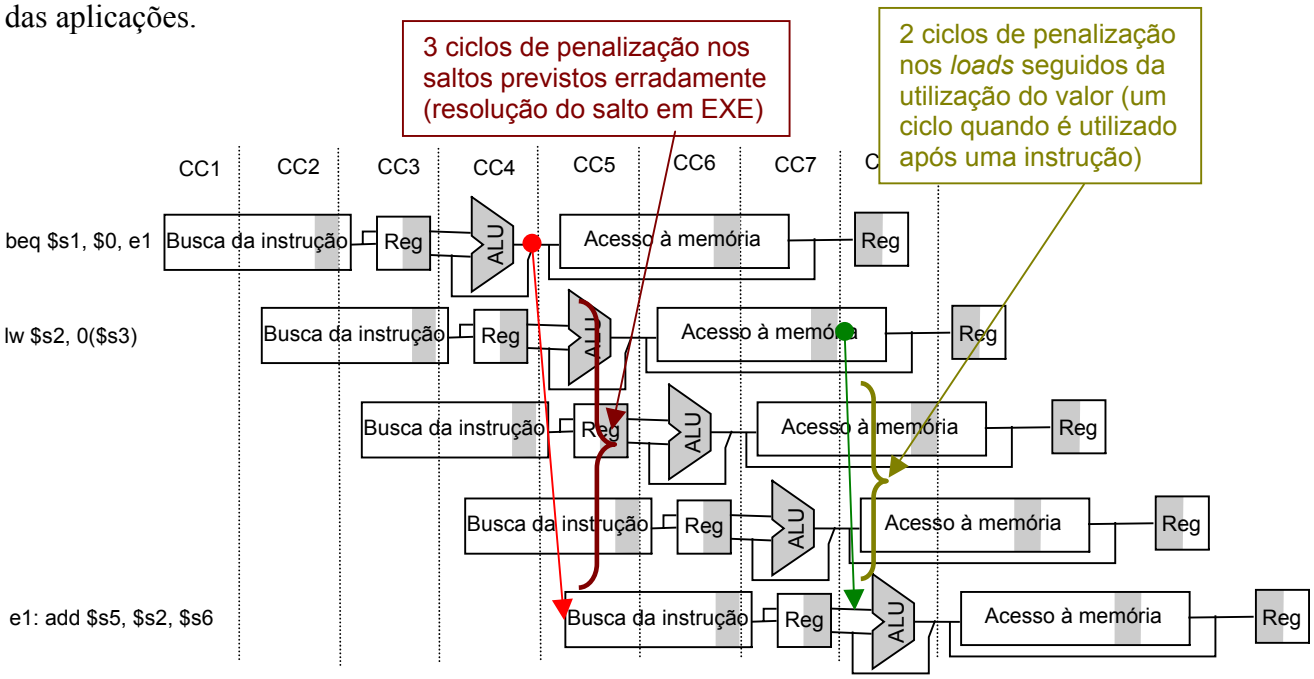


Figura 17 - Penalização decorrente dos saltos e de *load* no MIPS R4000

3 Execução super-escalar de instruções

Este capítulo apresenta as bases da execução super-escalar de instruções. Inicialmente são apresentados os princípios básicos deste tipo de arquitectura, sendo apresentado um exemplo de uma arquitectura MIPS com execução super-escalar de instruções e analisadas as anomalias que surgem neste tipo de arquitectura. Posteriormente, é apresentado o escalonamento estático de instruções. Por escalonamento entende-se a selecção das instruções a executar no próximo ciclo, o que é significativamente mais complexo nas arquitecturas com execução super-escalar de instruções. O escalonamento estático consiste em executar as instruções pela ordem do programa, assim a forma de escalonamento é fixa é não varia durante a execução do programa. Na secção 3.3 é apresentado um tipo de escalonamento mais complexo designado por escalonamento dinâmico, o que permite escolher dinamicamente a ordem de execução das instruções, conseguindo assim aumentar o número de instruções executadas por ciclo. Na última secção são apresentados alguns exemplos de arquitecturas com execução super-escalar de instruções.

3.1 Princípios básicos

A execução super-escalar de instruções baseia-se na execução de várias instruções em cada ciclo de relógio, **numa mesma fase de execução**, o que implica a duplicação de unidades funcionais para suportar a combinação de instruções em execução. Este tipo de execução pode ser combinado com a execução encadeada de instruções, o que origina uma arquitectura com várias cadeias de execução a funcionar em paralelo. A combinação destes dois tipos de arquitecturas permite obter CPI inferiores a um. O número de cadeias de execução a funcionar em paralelo é designado por grau de super-escalaridade. A Figura 18 apresenta um exemplo de execução super-escalar de instruções com grau dois, o que permite obter um CPI mínimo de 0,5.

Uma arquitectura super-escalar obriga à duplicação de unidades funcionais. Por exemplo, numa arquitectura MIPS de grau 2, seriam necessárias duas unidades de busca de instruções, descodificação, execução, acesso à memória e escrita dos valores no banco de registos. Desta forma seria possível efectuar a busca de 64 bits por ciclo (i.é., 2 instruções), descodificar duas instruções por ciclo, etc.

Um alternativa à duplicação de todas as unidades consiste em duplicar apenas algumas unidades funcionais. Assim apenas é possível executar em paralelo combinações predefinidas de instruções. Este tipo de arquitectura exige menos recursos físicos que a anterior, nomeadamente, requer menos transistores, mas também implica que o CPI mínimo é geralmente maior que o de uma arquitectura com a duplicação de todas as unidades. Esta característica deve-se ao facto de apenas as combinações predefinidas de instruções serem executadas em paralelo. No entanto, actualmente existe bastante flexibilidade na quantidade de combinações de instruções que podem ser executadas em paralelo, o que torna esta alternativa mais atractiva.

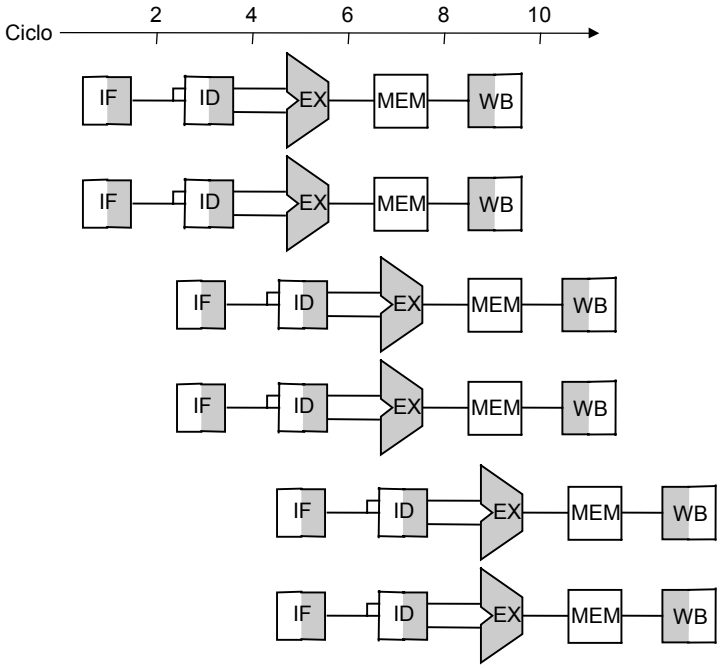


Figura 18 – MIPS superescalar com capacidade de execução de duas instruções por ciclo (grau 2)

Uma arquitectura MIPS com super-escalaridade de grau dois pode ser obtida através da introdução de uma ALU dedicada ao cálculo dos endereços de acesso à memória (necessários para o *lw* e *sw*). Assim é possível executar uma instrução do Tipo-R ou de salto em paralelo com um acesso à memória (Figura 19). Desta forma, a primeira instrução deve ser sempre do Tipo-R ou de *branch* e a segunda instrução deve ser um *load* ou *store*. Quando, num determinado ciclo não for possível executar esta combinação de instruções não irá ser escalonada uma instruções para essa cadeia de execução.

Tipo de instrução	Estágio						
ALU ou branch	IF	ID	EX	MEM	WB		
load ou *store*	IF	ID	EX	MEM	WB		
ALU ou branch		IF	ID	EX	MEM	WB	
load ou *store*		IF	ID	EX	MEM	WB	
ALU ou branch			IF	ID	EX	MEM	WB
load ou *store*			IF	ID	EX	MEM	WB

Figura 19 – MIPS super-escalar com suporte a uma Tipo-R/instruções de salto em paralelo com um acesso à memória

Note-se que nesta arquitectura MIPS é necessário efectuar o IF, ID e WB de duas instruções por ciclo, ou seja, é necessário efectuar a busca de duas instruções por ciclo (64 bits), descodificar 2 *opcodes* em paralelo, ler 4 registos por ciclo e escrever valores em dois registos. No entanto, apenas é introduzida uma segunda ALU para o cálculo dos endereços de *lw* e *sw*, o que é significativamente mais simples que uma ALU genérica, visto apenas ser necessário efectuar adições em complemento para 2.

3.2 Escalonamento estático

3.3 Escalonamento dinâmico

3.3.1 Sem renomeação de registos

3.3.2 Com renomeação de registos

3.4 Exemplos de processadores comerciais

4 Arquitecturas alternativas

4.1 Vectoriais

4.2 MMX/SSE/SSE2 vector para multimedia

4.3 VLIW – *Very Long Instruction Word*

4.4 Hyperthreading

4.5 Multi-core

5 Exercícios

Os exercícios propostos são constituídos por dois blocos. O primeiro bloco é formado por exercícios sobre a execução encadeada de instruções e têm um índole mais teórico-prático. Com estes exercícios pretende-se consolidar os conhecimentos dos alunos sobre a execução encadeada de instruções através da resolução de exemplos concretos, nomeadamente, através da análise de uma arquitectura MIPS com suporte à execução encadeada de instruções. Basicamente estes exercícios consistem no cálculo do CPI efectivo, obtido na execução de um programa em *assembly* do MIPS, em vários arquitectura MIPS com uma cadeia de execução de 5 estágios, mas as várias arquitecturas analisadas possuem sucessivamente melhorias para obter uma diminuição do número de ciclos de *stall*.

O segundo bloco de exercícios bloco é formado por exercícios sobre a execução superescalar de instruções e têm um índole mais prático. Com estes exercícios pretende-se consolidar os conhecimentos dos alunos através do desenvolvimento de pequenos programas em *assembly* IA-32 que permitam avaliar características de processadores comerciais, nomeadamente, medindo o CPI médio para diferentes classes de instruções, permitindo comparar características das arquitecturas de processadores comerciais (Pentium III, Pentium 4 e Athlon XP).

5.1 Execução encadeada de instruções

5.2 Super-escalaridade