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 2005

Prefácio

As arquitecturas dos processadores têm evoluído de uma forma extremamente rápida. Um dos principais factores que motiva esta evolução prende-se com o contínuo aumento do número de transístores em cada silício, mantendo os custos por processador. Esta tendência aparenta manter-se, pelo menos, durante os próximos 10 anos. O desafio na concepção de processadores modernos está na implementação de formas que rentabilizem esse o número adicional de transístores, de forma a aumentar o desempenho do processador.

Com estes apontamentos pretende-se reunir um conjunto de conceitos presentes em arquitecturas de processadores recentes, cuja documentação se encontra dispersa em vária bibliografia. Assim, concentra-se um conjunto de conceitos relativamente recentes, num documento que se pretende, ele próprio, em constante evolução.

Tratando-se de apontamentos de uma disciplina, abrangendo uma matéria em constante evolução, todos os comentários que contribuam para a melhoria do documento são extremamente bem vindos.

Braga, 14 de Junho de 2005

João Luís Sobral

Índice

1 Introdução 1
2 Execução encadeada de instruções 3
  2.1 Princípios básicos 3
  2.2 Anomalias estruturais 4
  2.3 Anomalias de dependências de dados 5
  2.4 Anomalias de dependências de controlo 8
  2.5 Análise de desempenho da execução encadeada 11
    2.5.1 Exemplo 1: cálculo do número de ciclos 11
    2.5.2 Exemplo 2: cálculo do CPI médio 12
  2.6 Compromissos na execução encadeada 13
    2.6.1 Exemplo: comparação de arquitecturas MIPS 15
3 Execução super-escalar de instruções 18
  3.1 Princípios básicos 18
  3.2 Escalonamento estático 23
    3.2.1 *Pipelining* com operações multi-ciclo 24
  3.3 Escalonamento dinâmico 27
    3.3.1 Sem renomeação de registos 27
    3.3.2 Com renomeação de registos 28
  3.4 Exemplos de processadores comerciais 30
    3.4.1 PowerPC 604 e Pentium Pro (PIII) 30
    3.4.2 Pentium 4 31
4 Arquitecturas alternativas 32
  4.1 Vectoriais 32
  4.2 MMX/SSE/SSE2 vector para multimédia 33
  4.3 VLIW – *Very Long Instruction Word* 35
  4.4 *Hyper-threading* 36
  4.5 *Multi-core* 37
5 Exercícios 38
  5.1 Execução encadeada de instruções 39
  5.2 Super-escalaridade 43
    5.2.1 Introdução 43
    5.2.2 Medição de tempo 44
    5.2.3 Resumo de conjunto de instruções IA-32 45
    5.2.4 Exercício 1 – Processador 48
    5.2.5 Exercício 2 – Hierarquia de memória 49
    5.2.6 Exercício 3 – Codificação de software 50

Índice de Figuras

Figura 1 – a) Execução encadeada de instruções b) execução super-escalar 1
Figura 2 – Execução de instruções vectoriais... 2
Figura 3 – Execução de instruções VLIW... 2
Figura 4 – Execução encadeada de instruções... 3
Figura 5 – Execução encadeada sem balanceamento das fases.. 4
Figura 6 – Tempo de execução de instruções em cadeia... 4
Figura 7 – *stall* devido a uma anomalia estrutural 5
Figura 8 – Dependências de dados 6
Figura 9 – Troca da ordem das instruções para evitar *stalls* 6
Figura 10 – Encaminhamento de dados... 7
Figura 11 – Limitação do encaminhamento de dados 8
Figura 12 – *stall* devido a uma anomalia de controlo... 9
Figura 13 – Previsão estática de saltos 9
Figura 14 – Previsão de saltos com 2 bit por endereço de salto.. 10
Figura 15 – Tabela de previsão de saltos... 11
Figura 16 – Aumento do número de estágios do MIPS... 14
Figura 17 – Ciclos de *stall* na arquitectura MIPS com 10 estágios.................................... 14
Figura 18 – Cadeia de execução do MIPS R4000 15
Figura 19 - Penalização decorrente dos saltos e de *load* no MIPS R4000 17
Figura 20 – MIPS super-escalar com grau 2 19
Figura 21 – Combinações possíveis de instruções no MIPS super-escalar........................ 19
Figura 22 – *Datapath* de um MIPS super-escalar (grau 2).. 20
Figura 23 – Penalizações devido a dependências de e de dados em MIPS super-escalar... 20
Figura 24 – Limitação no encaminhamento em arquitecturas super-escalares 21
Figura 25 – Escalonamento de instruções em MIPS super-escalar..................................... 21
Figura 26 – Escalonamento estático com *loop unrolling* e renomeação de registos........... 22
Figura 27 – Escalonamento dinâmico 22
Figura 28 – Arquitectura MIPS com *pipeline* com operações multi-ciclo......................... 24
Figura 29 – Latência (RAW) e intervalo de iniciação... 25
Figura 30 – Esquema para determinação da latência (RAW) 25
Figura 31 – Anomalias estruturais devidas a múltiplos WB por ciclo 25
Figura 32 – Exemplo de escalonamento numa cadeia com multi-ciclo 26
Figura 33 – Redução de ciclos de *stall* através de escalonamento dinâmico 27
Figura 34 – Dependências WAW e WAR no escalonamento dinâmico 27
Figura 35 – Estrutura genérica de uma *pipeline* com escalonamento dinâmico 29
Figura 36 – *Pipeline* do PowerPC 604 30
Figura 37 – Características do PowerPC 604 vs Pentium Pro.. 31
Figura 39 – Estágios da cadeia de execução do Pentium 4 31
Figura 40 – Exemplo de um cálculo com operações vectoriais 33
Figura 41 – Registos MMX.. 34
Figura 42 – Registos SSE/SSE2... 34
Figura 43 – Formato de instrução do IA-64 (Itanium).. 35
Figura 44 – Escalonamento de instruções com SMT 37
Figura 45 – Níveis de avaliação de desempenho.. 43
Figura 46 – Escala de tempo de eventos (Máquina a 1 GHz) 44

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)

IF	ID	EXE	MEM	WB			
	IF	ID	EXE	MEM	WB		
		IF	ID	EXE	MEM	WB	
			IF	ID	EXE	MEM	WB

a)

IF	ID	EXE	MEM	WB	
IF	ID	EXE	MEM	WB	
	IF	ID	EXE	MEM	WB
	IF	ID	EXE	MEM	WB

b)

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

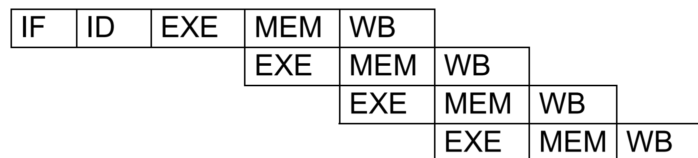
O tempo de execução de um programa é dado por:

Texe = #instruções x CPI x 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.

Mais recentemente surgiram duas linhas de arquitecturas de processadores que podem funcionar como complemento à execução encadeada de instruções: as arquitecturas vectoriais e VLIW.

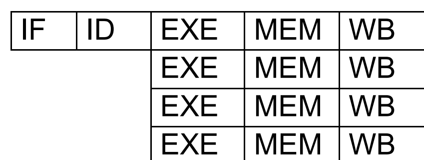
As arquitecturas vectoriais para multimédia recuperam um conceito bastante antigo das arquitecturas vectoriais, agora dirigido para aplicações multimédia. Este tipo de instruções baseia-se na especificação, numa só instrução, de uma operação sobre um vector de elementos (Figura 2)



IF	ID	EXE	MEM	WB			
			EXE	MEM	WB		
				EXE	MEM	WB	
					EXE	MEM	WB

Figura 2 – Execução de instruções vectoriais

As arquitecturas VLIW (*Very Long Instruction Word*) permitem especificar várias operações paralelas em cada instrução máquina (Figura 3).



IF	ID	EXE	MEM	WB
		EXE	MEM	WB
		EXE	MEM	WB
		EXE	MEM	WB

Figura 3 – Execução de instruções VLIW

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, incluindo as arquitecturas vectoriais e VLIW. O último capítulo apresenta um conjunto de exercícios cujo objectivo é alicerçar os conceitos apresentados nestes apontamentos.

2 Execução encadeada de instruções

2.1 Princípios básicos

A execução encadeada de instruções baseia-se na sobreposição da execução de instruções para 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 4 apresenta um exemplo de execução encadeada de instruções, para uma arquitectura com cinco estágios.

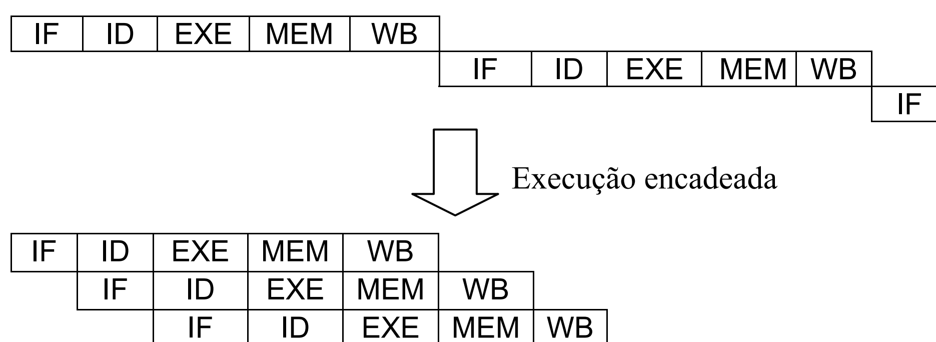


Figura 4 – Execução encadeada de instruções

Teoricamente a introdução da execução encadeada de instruções numa 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, de 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 5 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.

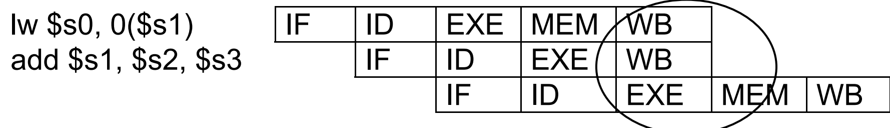


Figura 5 – Execução encadeada sem balanceamento das fases

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 6).

Texec*pl* = [#estágios + (#instruções - 1)] x Tcc

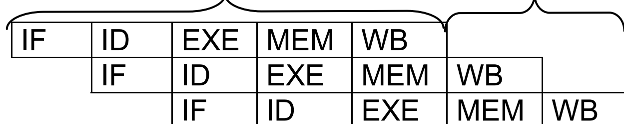


Figura 6 – Tempo de execução de instruções em cadeia

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 estágios 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 7 apresenta um exemplo de um *stall* devido a anomalias estruturais.

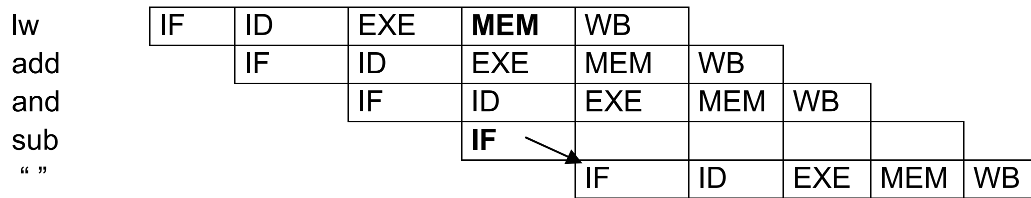


Figura 7 – *stall* devido a uma anomalia estrutural

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 8 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 8 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)
```

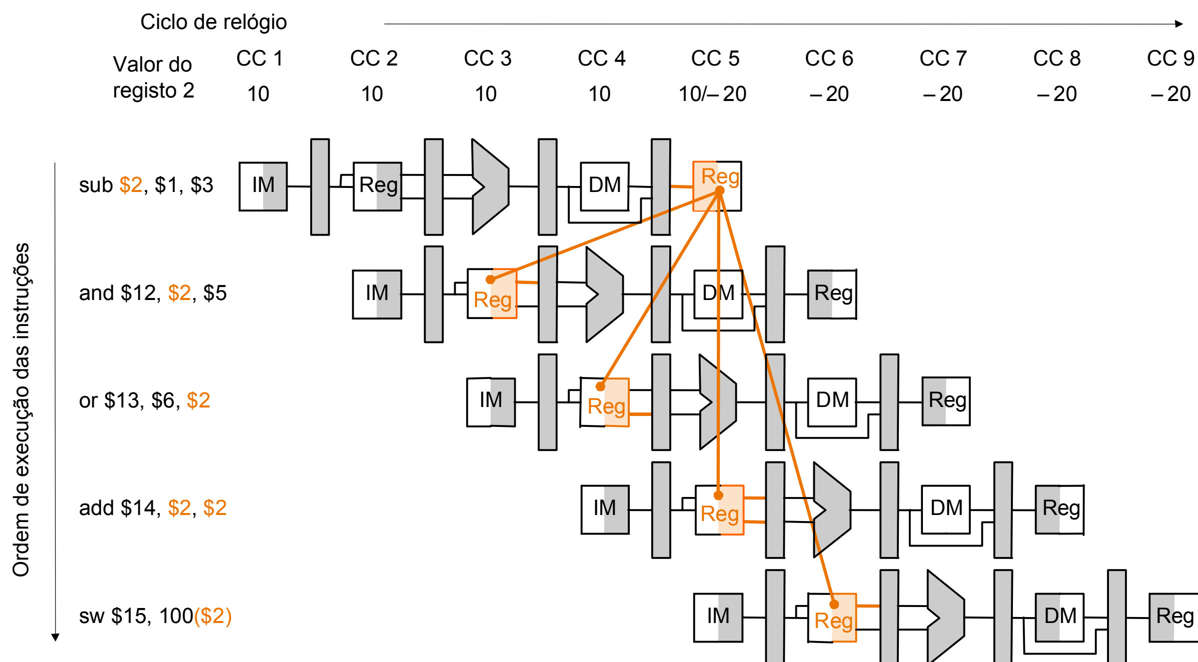
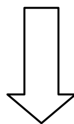


Figura 8 – Dependências de dados

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 9 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 9 – Troca da ordem das instruções para evitar *stalls*

A terceira estratégia para lidar com as dependências de dados consistem em criar atalhos na arquitectura para 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 10 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).

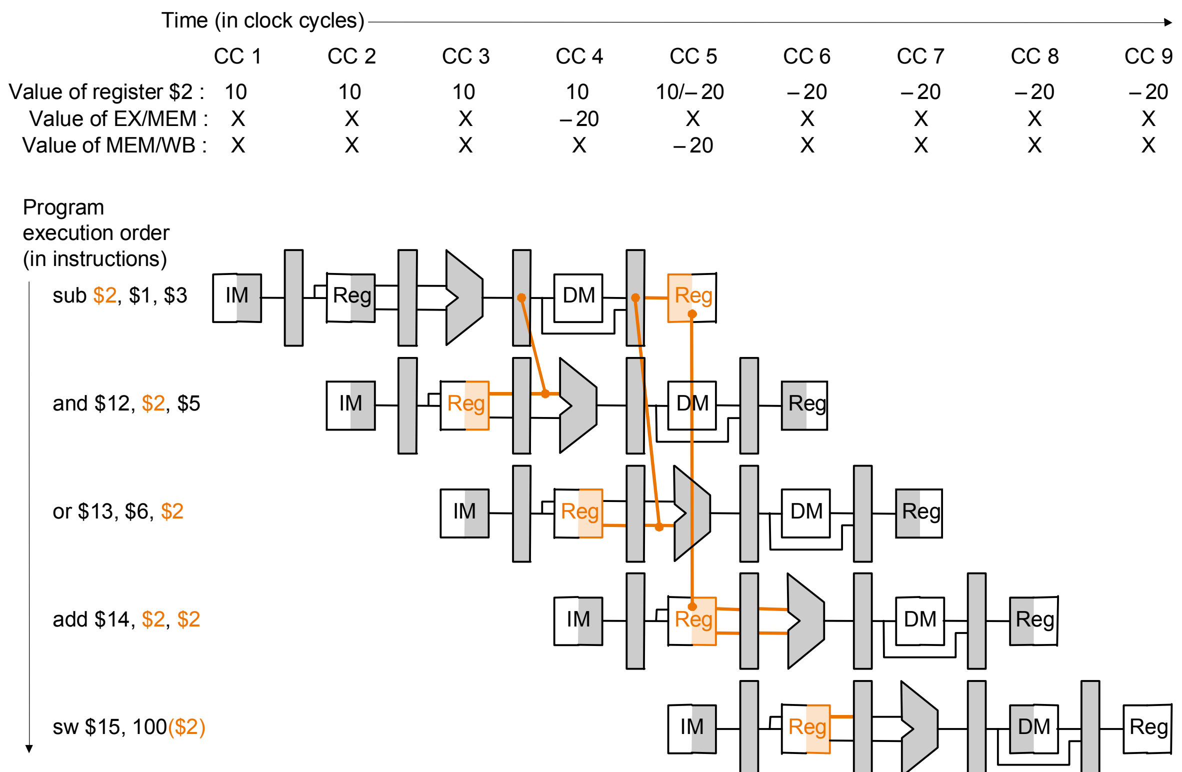


Figura 10 – Encaminhamento de dados

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 11). 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.

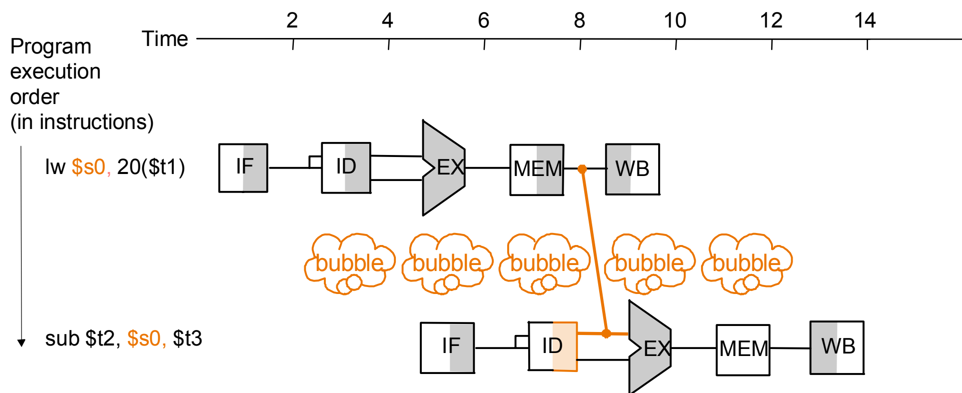


Figura 11 – Limitação do encaminhamento de dados

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 12). 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.

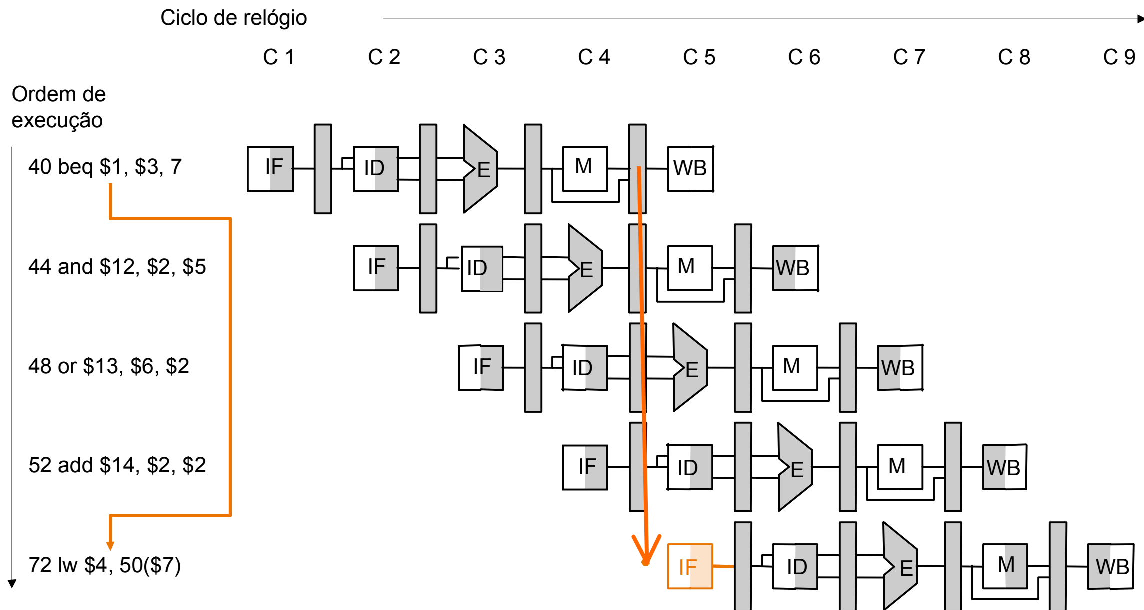


Figura 12 – *stall* devido a uma anomalia de controlo

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 13 ilustra um exemplo desta situação para uma cadeia de execução em que os saltos são resolvidos durante a fase ID.

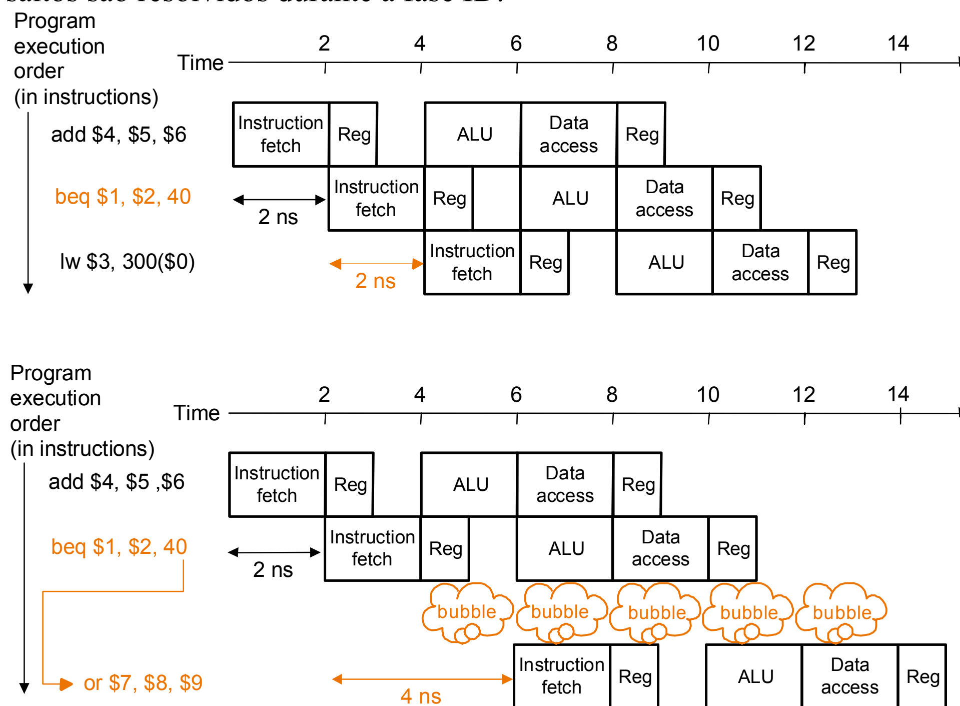


Figura 13 – Previsão estática de saltos

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 14) seguidas, o que, em geral, melhora a percentagem de previsões acertadas.

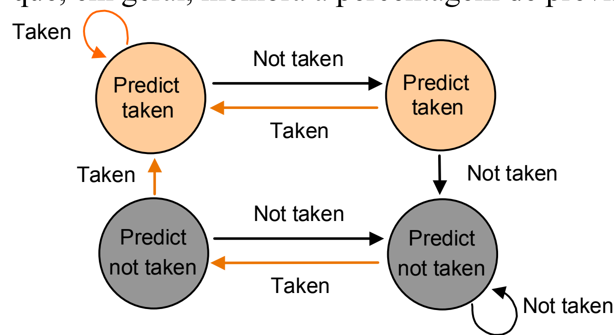


Figura 14 – Previsão de saltos com 2 bit por endereço de salto

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 15): 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 história 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 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.

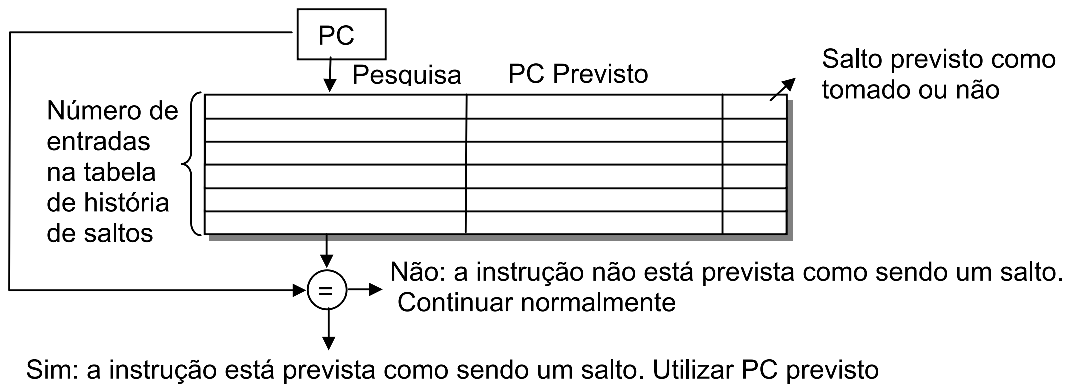


Figura 15 – Tabela de previsão de saltos

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, 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:

$$5 + (2 + 4 \times 64 - 1) + (2 + 2 \times 2 \times 64) + (3 \times 64) = 712 \text{ ciclos}$$

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, sabendo que 50% dos valores de *loads* são utilizados na instrução seguinte, que 75% dos saltos relativos são previstos correctamente e que 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 provém de duas fontes: *branch* (salto relativo ao PC) e *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 proporção 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 16). Considere-se também que se consegue obter estágios completamente balanceados, de modo a que o ciclo de relógio da arquitectura seja reduzido para metade.

IF	ID	EXE	MEM	WB			
	IF	ID	EXE	MEM	WB		
		IF	ID	EXE	MEM	WB	
			IF	ID	EXE	MEM	WB

a) MIPS 5 estágios

I1	I2	D1	D2	E2	E2	M1	M2	W1	W2			
	I1	I2	D1	D2	E1	E2	M1	M2	W1	W2		
		I1	I2	D1	D2	E1	E2	M1	M2	W1	W2	
			I1	I2	D1	D2	E1	E2	M1	M2	W1	W2

b) MIPS 10 estágios

Figura 16 – 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 17):

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

I1	I2	D1	D2	E2	E2	M1	M2	W1	W2						
	I1	I2	D1	D2	E1	E2	M1	M2	W1	W2					
		I1	I2	D1	D2	E1	E2	M1	M2	W1	W2				
			I1	I2	D1	D2	E1	E2	M1	M2	W1	W2			
				I1	I2	D1	D2	E1	E2	M1	M2	W1	W2		
					I1	I2	D1	D2	E1	E2	M1	M2	W1	W2	
						I1	I2	D1	D2	E1	E2	M1	M2	W1	W2

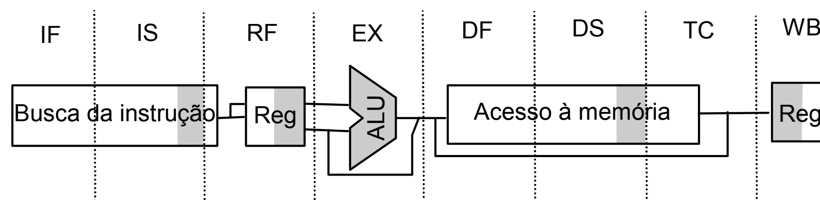
a) b) c)

Figura 17 – 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 18).



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 18 – 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 19).

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*5* = 1,5 x Tcc*8*):

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.

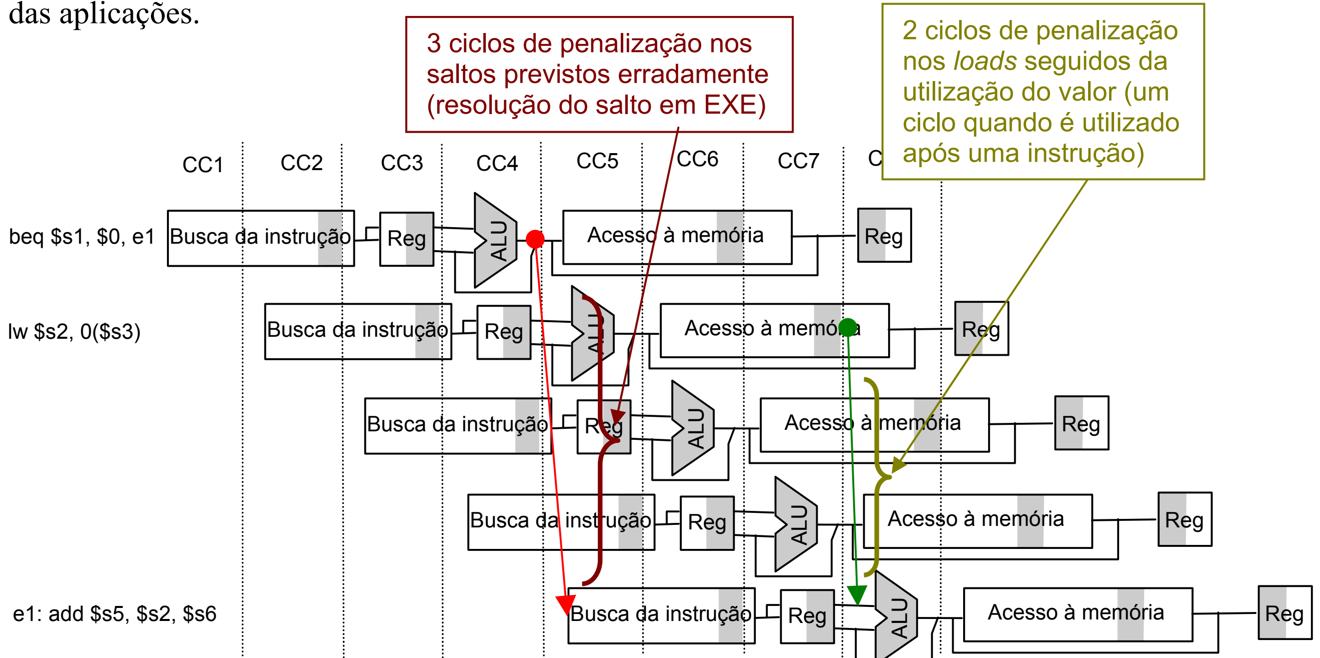


Figura 19 - Penalização decorrente dos saltos e de *load* no MIPS R4000

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 20 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.

Uma 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 transístores, 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.

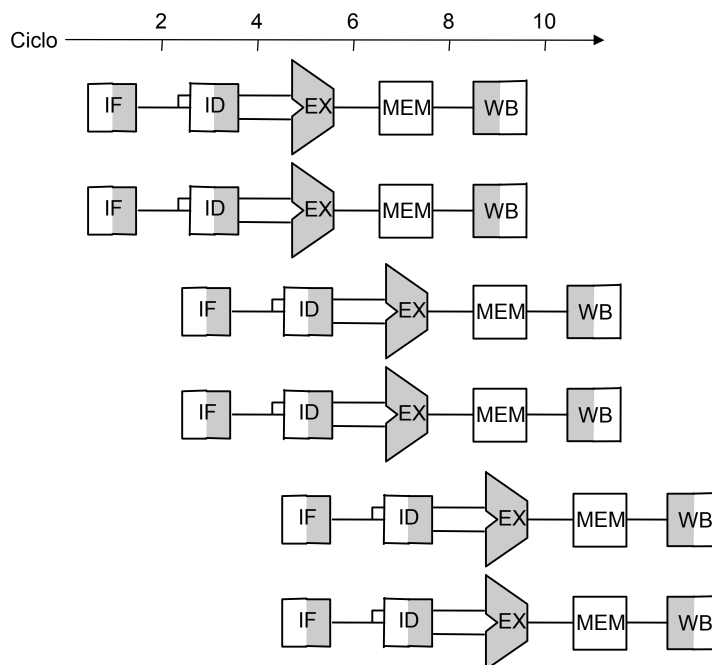


Figura 20 – MIPS super-escalar com grau 2

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 21). 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 instrução 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 21 – Combinações possíveis de instruções no MIPS super-escalar

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 (Figura 22).

As dependências de dados numa arquitectura super-escalar, tal como na execução encadeada de instruções, provocam *stalls* da cadeia de execução. No entanto, cada *stall* neste tipo de arquitectura é mais penalizante porque implica que as unidades de execução não realizam trabalho útil durante os ciclos de paragem.

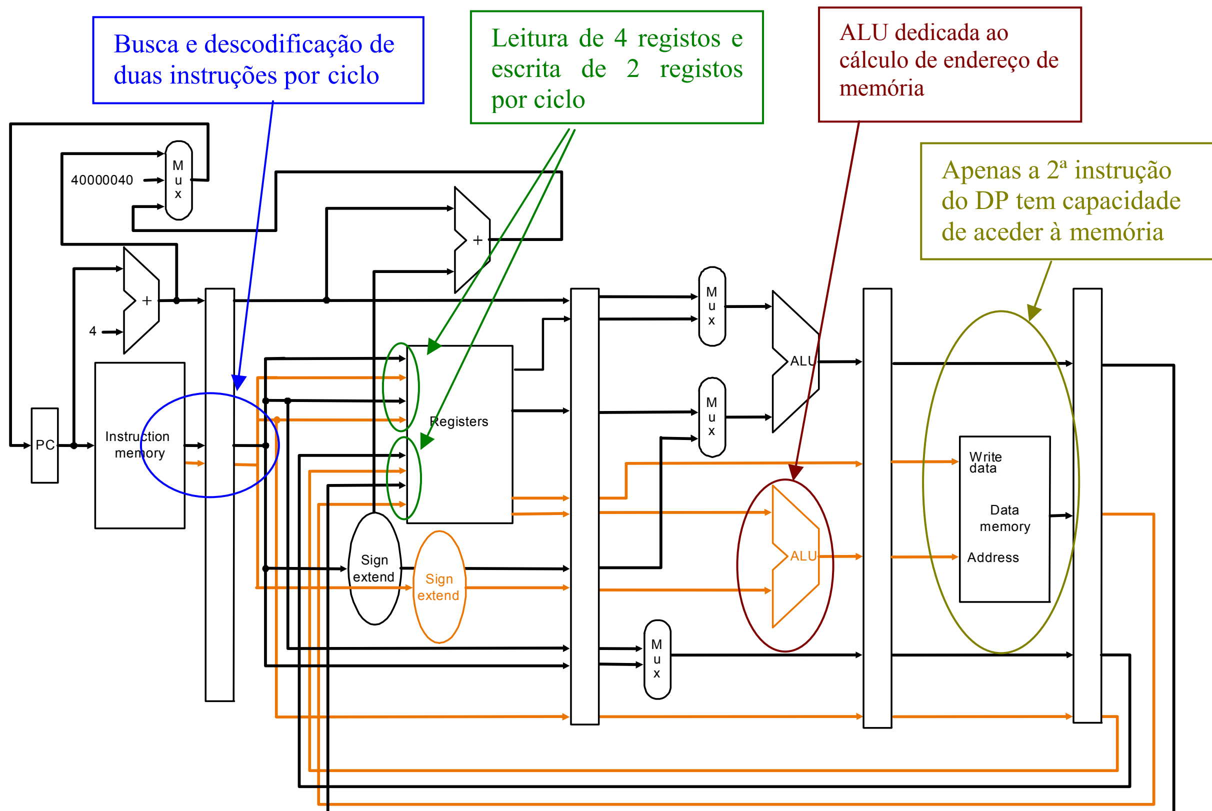


Figura 22 – *Datapath* de um MIPS super-escalar (grau 2)

A Figura 23 apresenta o número de ciclos de paragem provocados pelas dependências de controlo e de dados na arquitectura MIPS super-escalar com 5 estágios em análise.

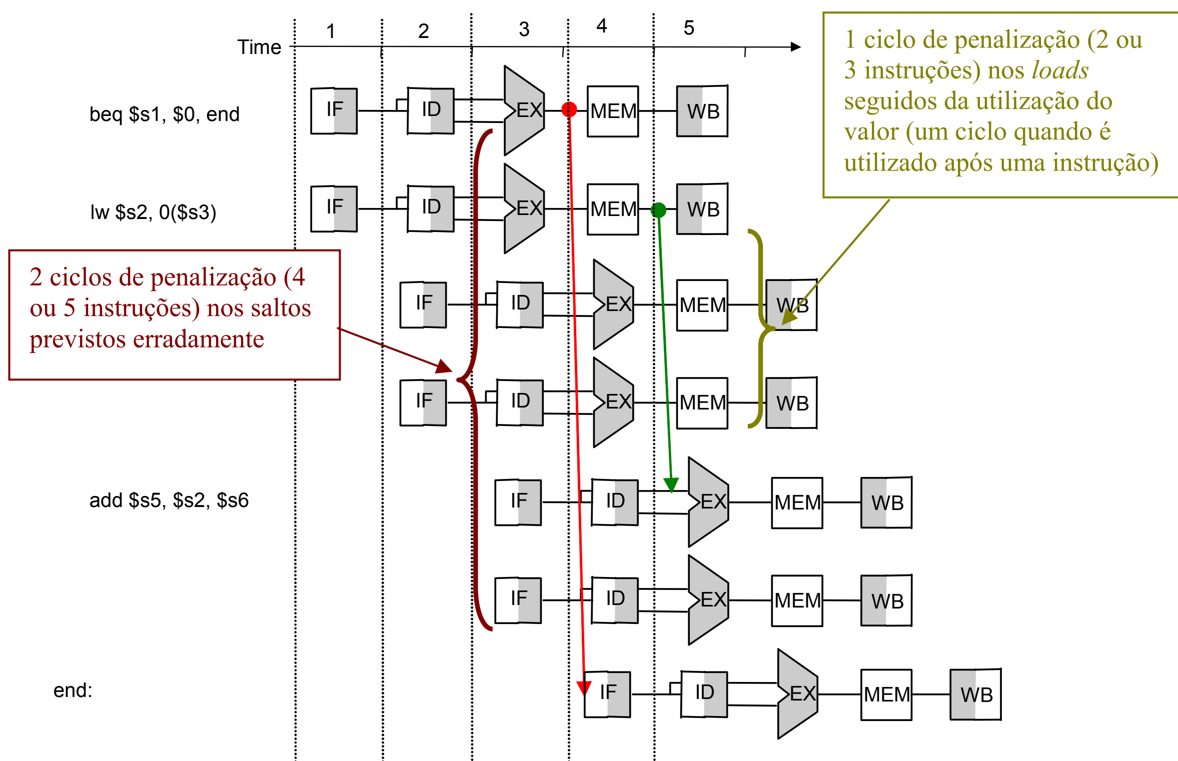


Figura 23 – Penalizações devido a dependências de e de dados em MIPS super-escalar

A detecção de anomalias e encaminhamento de dados em arquitecturas super-escalares é extremamente complexa. Adicionalmente as instruções executadas em cada ciclo devem ser independentes (Figura 24). Assim, geralmente nestas arquitecturas na fase de ID é necessário efectuar o escalonamento das instruções, i.é., escolher quais as instruções a executarem no ciclo seguinte. O escalonamento de instruções numa arquitectura super-escalar torna-se mais complexo que nas arquitecturas anteriores, porque para além de ser necessário garantir que as instruções não irão provocar *stalls* também é necessário determinar a unidade funcional onde irá ser executada cada uma das instruções.

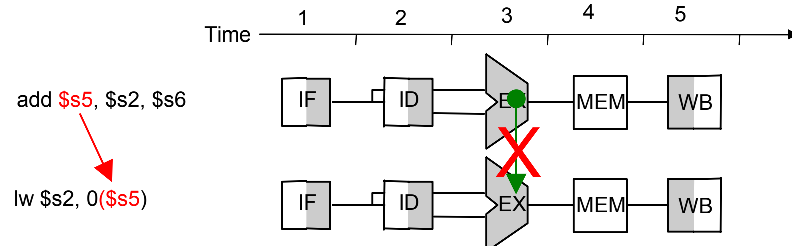


Figura 24 – Limitação no encaminhamento em arquitecturas super-escalares

A Figura 25 apresenta um exemplo de escalonamento de um programa MIPS, onde para diminuir o número de *stalls* se efectuou a reordenação de algumas das instruções do processador.

```
cic: lw    $t0, 0($s1)
     addu  $t0, $t0, $s2
     sw    $t0, 0($s1)
     addi  $s1, $s1, -4
     bne   $s1, $0, cic
```

Ciclo		ALU || *branch*	*Load* || *store*
1	cic:		lw $t0, 0($s1)
2		addi $s1, $s1, -4	
3		addu $t0, $t0, $s2	
4		bne $s1, $0, cic	sw $t0, 4($s1)

Figura 25 – Escalonamento de instruções em MIPS super-escalar

No primeiro ciclo de execução apenas foi possível executar a instrução *lw*, uma vez que as instruções *addu* e *sw* dependem do resultado desta instrução. No segundo ciclo continua a não ser possível arrancar com nenhuma destas duas instruções, uma vez que o *lw* provoca um ciclo de *stall*. Neste segundo ciclo foi iniciada *addi*, uma vez que esta não depende de nenhum resultado anterior. No terceiro e quarto ciclos já é possível iniciar, respectivamente as instruções *addu* e *sw* (note que foi necessário alterar o índice do *sw* porque o registo $s1 já se encontra alterado). No quarto ciclo é executada a instrução *bne* em simultâneo com a de *sw*. Este exemplo apresenta fortes dependências entre instruções, resultando numa utilização muito limitada da arquitectura super-escalar. Neste caso o CPI na execução do programa é de 4 ciclos / 5 instruções ou seja 0,8, muito superior ao 0,5 teórico.

A eficiência das arquitecturas super-escalares depende fortemente da qualidade do escalonador de instruções utilizado. Para tal vamos assumir que o ciclo é desdobrado 4 vezes (técnica conhecida como *loop unrolling*) e que se faz uma renomeação dos registos utilizados nas diferentes iterações do ciclo. A Figura 26 apresenta o programa da Figura 25 desdobrado 4 vezes. Após o desdobramento 3 instruções de *bne* podem ser removidas. Na segunda, terceira e quartas iterações do ciclo são utilizados respectivamente os registos $t1, $t2 e $t3, invés do $t0. Finamente, em cada iteração do ciclo original é utilizado um registo diferente de $s1, respectivamente $s3, $s4 e $s5. Note que após estas alterações o programa continua correcto (i.é., produz o mesmo resultado que o original), mas existem menos dependências entre instruções o que vai melhorar o grau de utilização das unidades funcionais. A mesma figura apresenta o escalonamento do programa depois de serem introduzidas estas alterações.

cic: lw $t0, 0($s1)
addu $t0, $t0, $s2
sw $t0, 0($s1)
addi ~~$s1~~$s3, $s1, -4
~~bne $s1, $0, cic~~
lw ~~$t0~~ $t1, 0(~~$s1~~$s3)
addu ~~$t0~~ $t1, ~~$t0~~ $t1, $s2
sw ~~$t0~~ $t1, 0(~~$s1~~$s3)
addi ~~$s1~~$s4, ~~$s1~~$s3, -4
~~bne $s1, $0, cic~~
lw ~~$t0~~ $t2, 0(~~$s1~~$s4)
addu ~~$t0~~ $t2, ~~$t0~~ $t2, $s2
sw ~~$t0~~ $t2, 0(~~$s1~~$s4)
addi ~~$s1~~$s5, ~~$s1~~$s4, -4
~~bne $s1, $0, cic~~
lw ~~$t0~~ $t3, 0(~~$s1~~$s5)
addu ~~$t0~~ $t3, ~~$t0~~ $t3, $s2
sw ~~$t0~~ $t3, 0(~~$s1~~$s5)
addi $s1, ~~$s1~~$s5, -4
bne $s1, $0, cic

Clk		ALU || *branch*	*Load || store*
1	cic:	addi $s3, $s1, -4	lw $t0, 0($s1)
2			lw $t1, 0($s3)
3		addu $t0, $t0, $s2	
4		addu $t1, $t1, $s2	sw $t0, 0($s1)
5		addi $s4, $s3, -4	sw $t1, 0($s3)
6			lw $t2, 0($s4)
7			
8		addu $t2, $t2, $s2	
9		addi $s5, $s3, -4	sw $t2, 0($s4)
10			lw $t3, 0($s5)
11			
12		addu $t3, $t3, $s2	
13		addi $s1, $s5, -4	sw $t3, 0($s5)
14		bne $s1, $0, cic	

Figura 26 – Escalonamento estático com *loop unrolling* e renomeação de registos

O número de ciclos de *stall* que surge ao efectuar o escalonamento das instruções pode ser reduzido se as instruções foram iniciadas fora da ordem do programa. A Figura 27 apresenta o programa anterior agora com a iniciação de instruções fora de ordem (assinalada a vermelho). Neste caso concreto o CPI obtido na execução do programa é de 8 ciclos / 17 instruções ou seja 0,53, valor muito próximo do limite teórico de 0,5. Note-se que para que seja possível obter este grau de utilização das unidades funcionais foi necessário introduzir a renomeação de registos e o escalonamento dinâmico de instruções.

Clk		ALU || *branch*	*Load || store*
1	cic:	addi $s3, $s1, -4	lw $t0, 0($s1)
2		addi $s4, $s3, -4	lw $t1, 0($s3)
3		addu $t0, $t0, $s2	lw $t2, 0($s4)
4		addu $t1, $t1, $s2	sw $t0, 0($s1)
5		addi $s5, $s4, -4	sw $t1, 0($s3)
6		addu $t2, $t2, $s2	lw $t3, 0($s5)
7		addi $s1, $s5, -4	sw $t2, 0($s4)
8		addu $t3, $t3, $s2	
9		bne $s1, $0, cic	sw $t3, 0($s5)

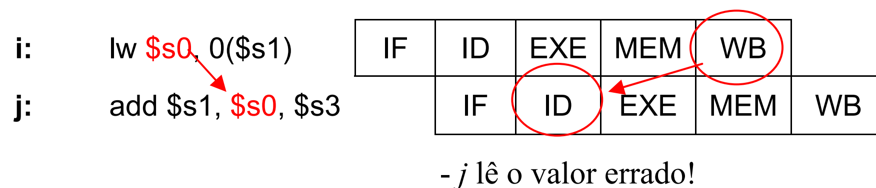
Figura 27 – Escalonamento dinâmico

As próximas secções centram-se nos vários tipos de escalonamento. Primeiro é apresentado um algoritmo de escalonamento estático, de seguida é apresentado o escalonamento dinâmico, sem renomeação de registos. Por fim é apresentado um algoritmo de escalonamento dinâmico, com renomeação de registos.

3.2 Escalonamento estático

Os algoritmos de escalonamento são utilizados nos processadores para seleccionar a(s) instrução(ões) que irão ser iniciadas em cada ciclo. Para além do escalonador ter que garantir que uma instrução quando inicia a execução tem disponíveis os recursos que necessita para executar (por exemplo, se a instrução efectuar uma operação na ALU o escalonador só pode seleccionar essa instruções quando existe um ALU livre) para que sejam evitadas as anomalias estruturais, também deve ter em atenção os três tipos de dependências entre instruções:

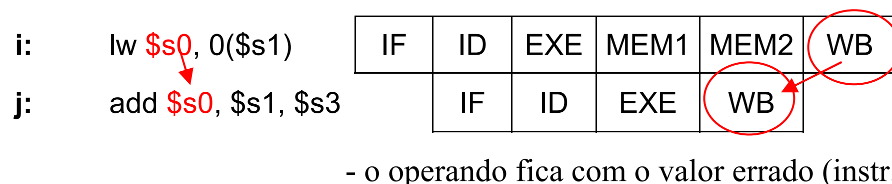
1. **RAW** (*Read After Write*) – uma instrução *j* lê um operando antes de uma instrução anterior *i* o escrever. Normalmente resolvido com *stall* ou encaminhamento de dados



- *j* lê o valor errado!

2. **WAW** (*Write After Write*) – uma instrução *j* escreve o operando antes de uma instrução anterior *i* o escrever.

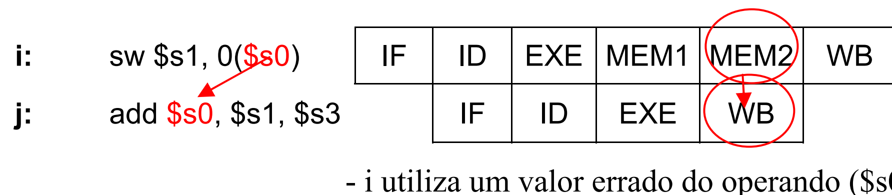
   Está apenas presente em *pipeline* que escrevem valores em mais do que um estágio da *pipeline*, ou quando as instruções não são completadas na ordem do programa



- o operando fica com o valor errado (instr. i)

3. **WAR** (*Write After Read*) – uma instrução *j* escreve um novo valor num operando antes de uma instrução anterior *i* o ler.

   Originado quando a leitura de registos pode ocorrer após o WB da instrução seguinte, nomeadamente, quando as instruções não são completadas pela ordem do programa



- i utiliza um valor errado do operando ($s0)

As dependências WAW e WAR resultam da utilização de um número limitado de registos (dependências de nome), podendo ser eliminadas através de RENOMEAÇÃO de REGISTOS (na secção anterior vimos um exemplo de renomeação de registos). Este mecanismo requer a utilização de registos adicionais (normalmente registos internos).

Estes tipos de dependências de dados também existem em acessos à memória, no entanto neste caso a sua detecção é mais complexa:

- 100 ($s0) = 20 ($s2)?
- 20 ($s2) = 20 ($s2) em iterações diferentes de um ciclo?

3.2.1 *Pipelining* com operações multi-ciclo

A complexidade do escalonamento de instruções e a análise de arquitecturas actuais, onde existe simultaneamente a execução em *pipeline* e super-escalar, pode ser mais facilmente perceptível se inicialmente se analisar uma arquitectura apenas com uma *pipeline*, mas onde as instruções não demoram todas o mesmo tempo a executar.

Numa arquitectura com *pipeline* não é normalmente viável que a execução das instruções todas com o mesmo número de estágios pelas seguintes razões:

- Na execução em *pipelining* não é viável que todas as operações demorem o mesmo número de ciclos, especialmente quando são suportadas instruções com vírgula flutuante (FP).
- A generalidade das arquitecturas possui várias unidades funcionais, cada unidade implementando uma funcionalidade específica (ex. operações em FP).
- Nem todas as unidades funcionais podem implementar a execução em *pipelining.*

Quando é suportada a execução encadeada de instruções com operações multi-ciclo o estágio EXE divide-se em vários estágios, podendo ser iniciada uma instrução em cada ciclo, caso contrário, o estágio EXE é repetido várias vezes.

A Figura 28 apresenta um exemplo de uma arquitectura deste tipo, com quatro unidades funcionais, com as seguintes características:

1. **Processamento de inteiros**, que processa operações ALU em inteiros, *load* e *store* e saltos, 1 ciclo
2. **Multiplicação de FP e de inteiros**, 7 ciclos
3. **Adição e subtracção de FP**, 4 ciclos
4. **Divisão de FP e de inteiros**, 24 ciclos, sem *pipeline*

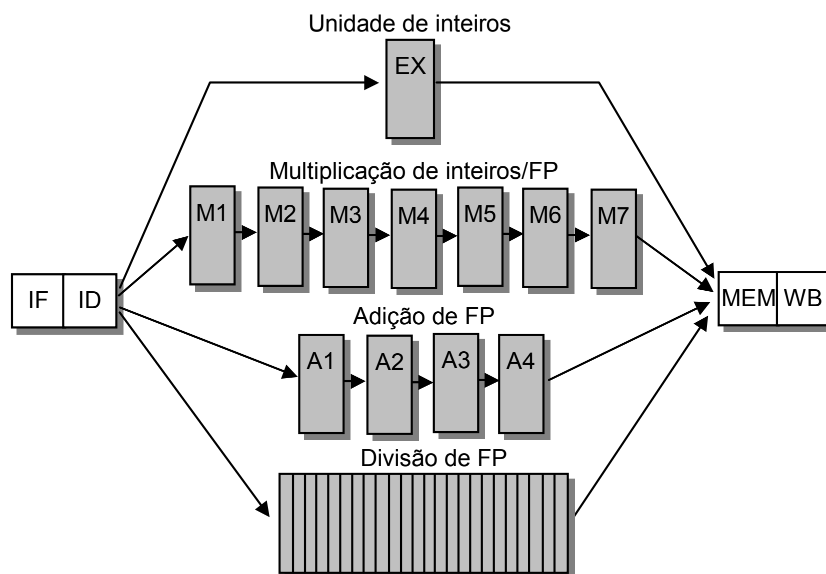


Figura 28 – Arquitectura MIPS com *pipeline* com operações multi-ciclo

Unidade funcional	Latência RAW	Intervalo de iniciação
ALU de inteiros	0	1
Memória de dados (lw)	1	1
Adição FP	3	1
Multiplicação FP	6	1
Divisão FP	24	24

Figura 29 – Latência (RAW) e intervalo de iniciação

Nesta arquitectura são efectuados, no máximo, um IF e um ID por ciclo. Após a fase de ID a instrução entra na fase EXE, que irá demorar um número diferente de ciclos, em função do tipo de instrução em execução (i.é., da unidade onde é escalonada). Adicionalmente quando existem dependências RAW é necessário assegurar que estas são resolvidas antes da instrução entrar na fase de execução. A Figura 29 resume estas características. As duas colunas apresentadas indicam o seguinte:

- **latência** RAW - número de ciclos necessários entre uma instrução que produz um valor e uma instrução que utiliza esse valor (dependências RAW, com encaminhamento de dados);
- **intervalo de iniciação** - número de ciclos que deve separar duas instruções

Note-se que, por exemplo, só pode ser iniciada uma operação de divisão de FP a cada 24 ciclos e que sempre que existe uma dependência RAW entre uma adição de FP e a instrução seguinte, deverá ser inserido um *stall* de 3 ciclos. A Figura 30 apresenta esquematicamente como foram determinadas as várias latências RAW.

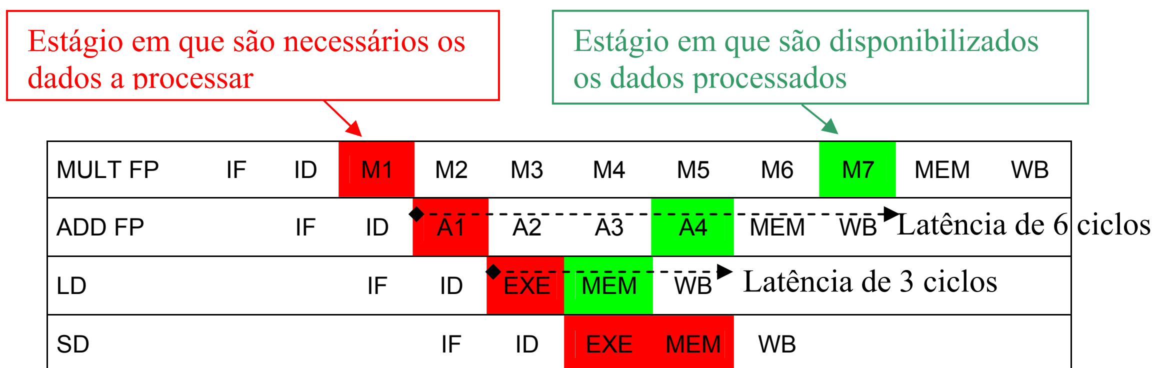


Figura 30 – Esquema para determinação da latência (RAW)

A unidade de divisão pode originar anomalias estruturais, uma vez que não implementa *pipeline*, o que origina *stalls*. O número de escritas em registos num ciclo pode ser superior a 1 (anomalia estrutural) porque a conclusão das instruções (WB) não é efectuada pela ordem que são iniciadas (Figura 31)

Instrução	Ciclo										
	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EXE	MEM	WB					
…			IF	ID	EXE	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
LD F2, …					IF	ID	EXE	MEM	WB		
						IF	ID	EXE	MEM	WB	
LD F8, 0 (R2)							IF	ID	EXE	MEM	WB

Figura 31 – Anomalias estruturais devidas a múltiplos WB por ciclo

As dependências de dados (RAW) impõem limitações às instruções que podem iniciar a execução em cada ciclo, uma vez que devem ser respeitadas a latência e intervalos de iniciação, o que origina *stalls* mais frequentes (Figura 32).

						C	i	c	l	o						
Instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
LD F4, 0(R2)	IF	ID	EX	M	WB											
MULTD F0, F4, F6		IF	ID	*st*	M1	M2	M3	M4	M5	M6	M7	M	WB			
ADDD F2, F0, F8			IF	*st*	ID	*st*	*st*	*st*	*st*	*st*	*st*	A1	A2	A3	A4	M
SD F2, 0(R2)				*st*	IF	*st*	*st*	*st*	*st*	*st*	*st*	ID	*st*	*st*	st	EX

Figura 32 – Exemplo de escalonamento numa cadeia com multi-ciclo

O escalonamento de instruções neste tipo de arquitectura tem ainda que considerar as seguintes situações:

- Podem surgir problemas com dependências WAW porque a conclusão das instruções (WB) não é efectuada pela ordem que são iniciadas;

- As dependências WAR não causam problemas porque todas as instruções lêem os operandos (2ºestágio) antes da seguinte os escrever (depôs do 5º estágio);

- Existem problemas com o processamento de excepções, uma vez que as instruções não são completadas pela ordem do programa

Um **algoritmo de escalonamento estático** para este tipo de cadeia de execução consiste em assegurar, durante a fase ID, que a instrução pode completar sem originar um *stall* da cadeia de execução. Assim na fase ID verifica-se se a instrução pode iniciar a fase de execução, caso contrário é originado um *stall* das instruções em IF e ID, tal acontece nos seguintes casos:

1. Existem anomalias estruturais - apenas uma instrução pode fazer WB em cada ciclo e a unidade funcional onde irá executar deve estar livre; (a instrução ADDD na Figura 31 provoca um ciclo de *stall*)

2. As dependências RAW não são resolvidas com encaminhamento - existem instruções pendentes que escrevem num registo utilizado (MULTD na Figura 32 origina um ciclo de *stall*)

3. A instrução origina anomalia WAW - uma das instruções pendentes escreve no mesmo registo que a actual (na Figura 31 LD F2 iria fazer *stall* durante 3 ciclos para efectuar WB depois de ADDD F2)

Este algoritmo de escalonamento origina, em média, entre 0,65 e 1,21 *stalls* por instrução (SPEC FP 1989).

O escalonamento estático com a iniciação das instruções pela ordem do programa, apenas quando não existem dependências é demasiado limitativo!

A arquitectura anterior pode ser facilmente estendida para incluir a execução super-escalar de instruções, bastando para tal permitir múltiplos IF, ID, MEM e WB por ciclo. No entanto, esse tipo de arquitecturas surge normalmente associado ao escalonamento dinâmico de instruções, para que se consiga explorar de forma eficiente as várias unidades de execução.

3.3 Escalonamento dinâmico

No escalonamento dinâmico o processador reordena dinamicamente as instruções para diminuir os *stalls* da cadeia de execução o que implica que as instruções podem ser iniciadas e/ou completadas fora de ordem. Desta forma a complexidade dos compiladores é reduzida e também é reduzida a dependência do código gerado da plataforma alvo pelo compilador. No entanto a complexidade do hardware aumenta substancialmente (como iremos ver na análise de processadores comerciais, nas arquitecturas mais recentes uma parte significativa dos estágios da cadeia de execução é dedicada ao escalonamento das instruções). Note-se que o escalonamento dinâmico não elimina os *stalls* devidos a dependências do tipo RAW mas diminui o seu impacto, uma vez que pode escalonar outras instruções para executar durante os ciclos de *stall* (Figura 33). O escalonamento dinâmico assume especial relevância nas arquitecturas super-escalares, uma vez que é necessário iniciar várias instruções em cada ciclo de relógio, para que as unidades de execução executem trabalho útil em cada ciclo de relógio.

Figura 33 – Redução de ciclos de *stall* através de escalonamento dinâmico

O escalonamento dinâmico pode introduzir dependências WAW e WAR (Figura 34) mesmo que inicialmente a arquitectura não originasse esse tipo de dependências (caso do MIPS com uma cadeia de execução com 5 estágios).

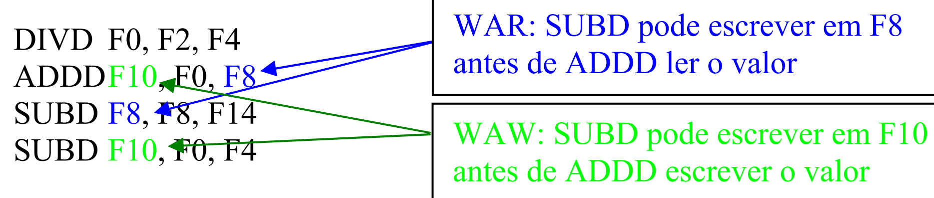


Figura 34 – Dependências WAW e WAR no escalonamento dinâmico

3.3.1 Sem renomeação de registos

O primeiro algoritmo de escalonamento dinâmico que iremos analisar baseia-se num *scoreboard* e não efectua a renomeação de registos, o que implica que tem que lidar com as dependências WAW e WAR. A ideia base do algoritmo consiste em anotar num quadro (*scoreboard*) as dependências entre instruções em execução, para determinar quando uma instrução pode iniciar a execução. O algoritmo baseia-se em 4 passos, sendo os passos 1 e 2 correspondentes a uma fase de ID, no entanto, apenas a fase 1 é realizada pela ordem do programa:

1. ***issue*** – se a unidade funcional se encontra livre e nenhuma instrução activa escreve no mesmo registo, inicia a instrução e actualiza o *scoreboard*. Caso contrário efectua o *stall* da *pipeline*. Evita anomalias WAW e estruturais.

2. **leitura dos operandos** – quando os operandos da instrução estão disponíveis e mais nenhuma instrução activa escreve nesses operandos estes são lidos dos registos e a instrução inicia a execução. Evita anomalias RAW.

3. **execução** – a unidade funcional executa a instrução, podendo demorar vários ciclos. Quando termina notifica o *scoreboard*

4. **escrita dos valores** – quando não existem instruções anteriores para execução que lêem o registo destino, este é actualizado. Evita WAR.

O primeiro passo evita que uma instrução execute sem que a unidade funcional correspondente esteja disponível. Por exemplo, se uma divisão ocupar 3 ciclos e só existir uma unidade para efectuar as divisões, esta fase garante que durante os 3 ciclos seguintes não é escalonada mais nenhuma divisão. Para que este fase seja possível é necessário anotar no quadro quais as unidades de execução que estão ocupadas (e, possivelmente, quantos ciclos permanecerão ocupadas).

O primeiro passo também evita as anomalias WAW, uma vez que uma instrução não passa para a fase 2 se existir uma anterior que escreve no mesmo registo (esta informação também pode ser anotada no quadro, marcando o registo destino de cada instrução).

Note-se que sempre que surge um *stall* da cadeia de execução durante o primeiro passo, todas as instruções seguintes do programa ficam bloqueadas (i.é., não são efectuados mais IF) até a anomalia estar resolvida.

O segundo passo garante que a instrução só entra na fase de execução quando os valores que necessita já estão calculados, i.é., quando as dependências RAW se encontram resolvidas. Note-se que nesta fase as instruções podem executar fora da ordem do programa e que, mais uma vez, o quadro pode ser utilizado para determinar quais os valores que se encontram calculados (i.é., quais os registos cujo valor não irá ser alterado por nenhuma instrução que já passou pela fase 1). Atente a que como as dependências WAW bloqueiam a cadeia de execução na fase 1, apenas uma instrução, no máximo, irá escrever num registo destino.

O terceiro passo consiste na execução da instrução.

O quarto passo requer um *buffer* uma vez que a instrução só pode escrever o resultado num registo se todas as instruções anteriores que lêem esse registo já tiverem entrado em execução. Assim, quando ainda existe uma instrução anterior que lê esse registo e que ainda não completou a execução, o resultado da execução é armazenado num *buffer*. Esta situação pode ser identificada mais uma vez anotando no quadro que existe uma instrução na fase 2 que irá ler esse registo.

3.3.2 Com renomeação de registos

Este algoritmo de escalonamento apresentado na secção anterior pode ser melhorado se for introduzida a renomeação dos registos. A ideia base consiste em evitar as anomalias WAR através de cópias dos valores dos registos. Assim, quando uma instrução passa pela fase ID e os valores dos registos que utiliza já estão disponíveis a instrução retém uma cópia desses valores (e colocada num *buffer*, designado por estação de reserva). Deste modo as instruções seguintes que escrevem no mesmo registo já podem completar, uma vez que o valor já foi copiado para local seguro. As anomalias WAW também são

resolvidas de modo semelhante, garantindo que as instruções seguintes irão utilizar o valor correcto (i.é., o resultado da segunda escrita).

Globalmente, nos algoritmos de escalonamento dinâmico, a *pipeline* é dividida em três grupos de unidades: busca de instruções/descodificação/*issue*, unidades de execução e unidade de conclusão (Figura 35):

- A unidade de busca e descodificação é responsável por descodificar as instruções e enviá-las para a unidade de execução correspondente.

- A unidade de conclusão das instruções é responsável por verificar quando é seguro tornar os resultados da instrução visíveis (saltos)

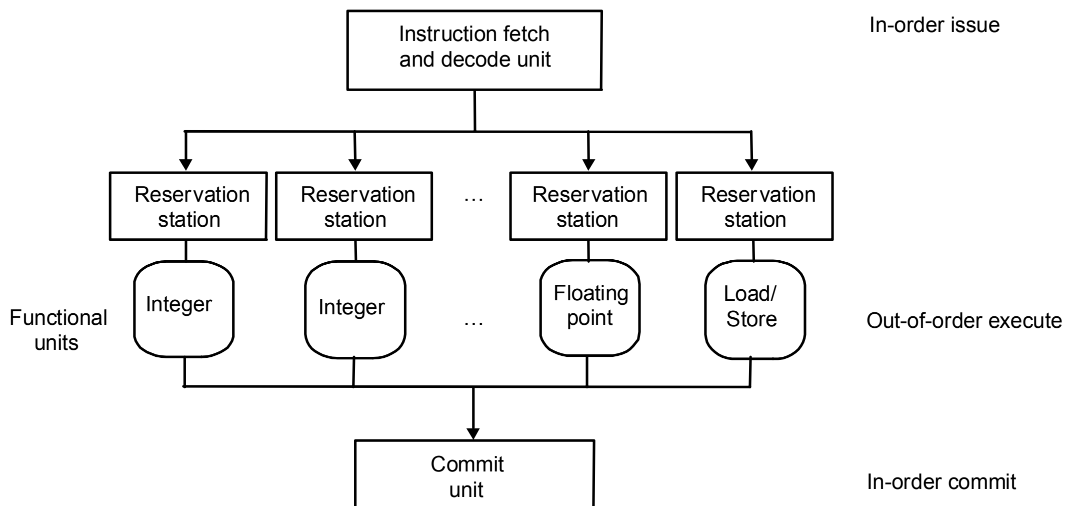


Figura 35 – Estrutura genérica de uma *pipeline* com escalonamento dinâmico

Um algoritmo de escalonamento dinâmico com renomeação de registos extremamente conhecido é o de Tomasulo. Este algoritmo baseia-se num processo de mudança de nome de registos (i.é., renomeação de registos) e em estações de reserva para eliminar as dependências WAR e WAW. As estações de reserva contêm instruções pontas para execução e cópias dos operandos que já estão disponíveis. A renomeação de registos atribui registos físicos diferentes sempre que é detectada uma dependência. Este algoritmo baseia-se em 3 fases:

1. ***issue*** – se a estação de reserva da unidade funcional se encontra livre envia a instrução para a estação de reserva correspondente com cópias dos operandos já disponíveis e atribui registos físicos os operandos. Caso contrário efectua o *stall* da *pipeline*. Evita anomalias estruturais, elimina WAR e WAW.
2. **execução** – quando os operandos estão disponíveis inicia a execução da instrução. Evita RAW.
3. **escrita do resultado** – o resultado é enviado às unidades à espera do resultado, e, posteriormente escrito no banco de registos.

A fase 1 elimina as dependências WAR e WAW através da renomeação de registos e das estações de reserva. As instruções apenas entram na fase 2 quando as dependências RAW estão resolvidas (i.é., o operados que necessitam já estão calculados).

O escalonamento dinâmico da *pipeline* é bastante mais complexo que as abordagens anteriores, especialmente, porque em geral surge ligado à execução super-escalar. Na generalidade dos programas o escalonamento dinâmico consegue iniciar e/ou concluir entre 4 e 6 instruções por ciclo.

3.4 Exemplos de processadores comerciais

Nesta secção são apresentados exemplos de processadores comerciais, sendo dado especial relevo a quatro aspectos:

1. Número de instruções descodificadas por ciclo – número de instruções que podem ser enviadas para as estações de reserva durante um ciclo; corresponde ao número de instruções na fase *issue* (1ª fase do algoritmo de Tomasulo)
2. Número de instruções iniciam a execução – número de instruções podem entram na fase EXE em cada ciclo (2ª fase do algoritmo de Tomasulo)
3. Número de instruções completadas por ciclo de relógio – corresponde ao número de instruções que podem efectuar WB em cada ciclo (3ª fase do algoritmo de Tomasulo)
4. Número/tipo de unidades funcionais – corresponde às unidades funcionais existentes para executar a fase EXE, determinado as combinações possíveis de instruções que podem iniciar a fase EXE.

3.4.1 PowerPC 604 e Pentium Pro (PIII)

A Figura 36 apresenta a arquitectura do PowerPC 604. Neste processador podem ser descodificadas 4 instruções por ciclo (caixa Decode/dispatch) e podem iniciar a fase EXE 6 instruções em cada ciclo: 2 operações sobre inteiros, um salto, uma operação sobre virgula flutuante, um *store* ou uma operação complexa sobre inteiros e *load* ou *store*. Nesta arquitectura podem completar a execução 6 instruções por ciclo. A Figura 37 resume as características desta arquitectura e compara-as com as do Pentium Pro (arquitectura similar ao Pentium III)

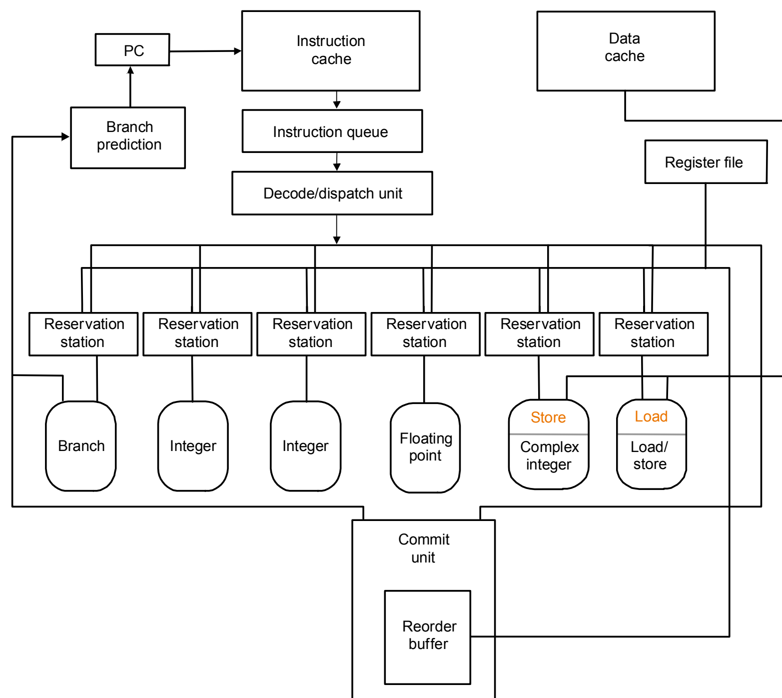


Figura 36 – *Pipeline* do PowerPC 604

Parâmetro	PowerPC 604	Pentium Pro
# máximo de instruções descodificadas por ciclo de relógio	4	3
# máximo que iniciam a execução por ciclo de relógio	6	5
Número máximo de instruções completadas por ciclo de relógio	6	3
Número de instruções no *reorder buffer*	16	40
Número de entradas na tabela de história de saltos	512 (2 bits)	512 (4 bits)
Número de *rename buffers*	12 Int. + 8 FP	40
Número total de *reservation stations*	12	20
Número total de unidades funcionais	6 2 - Inteiros + 1 - Inteiros cpl. + 1 - FP + 1 - saltos + 1 - *load* e *store*	6 2 – inteiros + 1 – FP 1 – Saltos 1 – *Load* 1 - *Store*

Figura 37 – Características do PowerPC 604 vs Pentium Pro

Os *rename buffers* são registos internos utilizados pelas instruções em execução para armazenar dos resultados até a instrução ser completada. São atribuídos às instruções na fase de descodificação (são utilizados para a implementação da renomeação de registos) Os *reorder buffers* são utilizados para manter os resultados das instruções que ainda não fizeram WB.

3.4.2 Pentium 4

A Figura 38 apresenta a arquitectura do Pentium 4. Esta arquitectura tem a particularidade de converter as instruções IA-32 em instruções RISC (designadas por uOP), característica também presente no PIII e AMD XP. No entanto, a *cache* de instruções do P4 (designada por *tracecache*) armazena estas instruções invés das instruções IA-32.

A Figura 39 apresenta os estágios da cadeia de execução do P4. Os estágios 1 a 4 correspondem a IF. Os estágios 5 e 20 são apenas utilizados para mover os operandos entre unidades funcionais. Os estágios 6 a 9 correspondem às fases de reserva de recursos, renomeação de registos e colocação das instruções na estação de reserva. As fases 10 a 16 corresponde ao escalonamento das instruções, envio para a unidade funcional correspondente e leitura dos registos. A fase 17 corresponde à execução da instrução (finalmente), podendo demorar um ou vários ciclos. As fases posteriores correspondem a WB dos resultados.

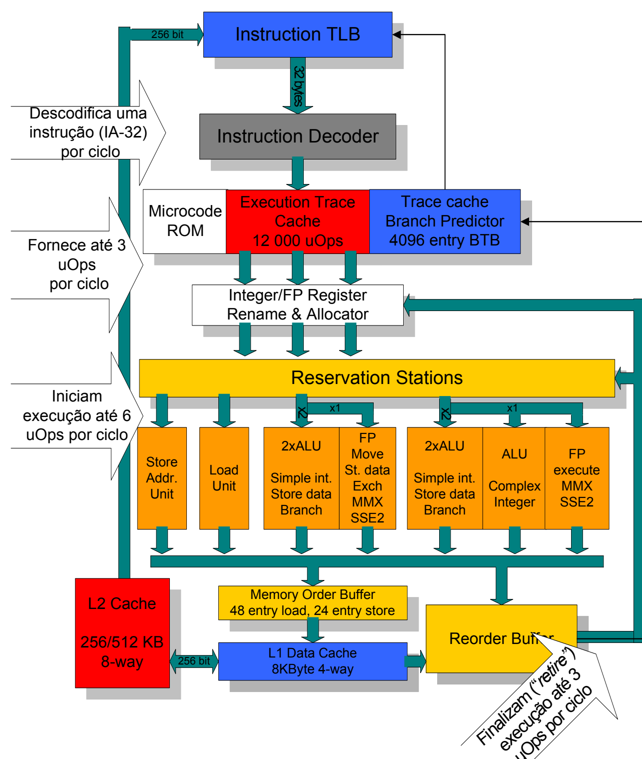


Figura 38 – *Pipeline* do Pentium 4

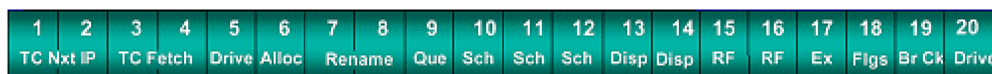


Figura 39 – Estágios da cadeia de execução do Pentium 4

4 Arquitecturas alternativas

Nesta secção são analisadas algumas arquitecturas alternativas aos conceitos clássicos de execução encandeada de instruções e à execução super-escalar. Note-se que, na maior parte dos casos, estas arquitecturas alternativas podem funcionar como complemento às arquitecturas clássicas.

4.1 Vectoriais

As arquitecturas vectoriais estão vocacionais para a execução de operações sobre vectores e matrizes de forma mais eficiente. Uma operação típica sobre vectores efectua uma adição de dois vectores de 64 elementos, em vírgula flutuante. A operação é equivalente a uma iteração sobre os elementos do vector, efectuando uma multiplicação por iteração. A principal motivação das instruções vectoriais é a maior eficiência que pode ser conseguida através da especificação das operações sobre os vectores numa só instrução, invés de utilizar um ciclo para iterar sobre os elementos do vector. As principais vantagens das instruções vectoriais são:

1. O cálculo de um resultado é independente do anterior, possibilitando a utilização de *pipeline* com bastante estágios, sem gerar anomalias de dados;

2. Uma só instrução especifica um grande número de operações, equivalente à execução de um ciclo, o que reduz a quantidade de buscas de instruções

3. Os acessos à memória para carregar os elementos do vector em registos podem tirar partido da optimização do débito da memória, amortizando o custo elevado dos acessos à memória;

4. As anomalias de controlo são reduzidas porque os ciclos são transformados numa só instrução.

Considere-se a título de exemplo que se pretende efectuar a multiplicação de um escalar α com um vector X e adicionar o resultado a um segundo vector Y, armazenado o resultado neste segundo vector:

$Y = \alpha X + Y$ Y, X - vectores de 64 elementos, α – escalar

A Figura 40 apresenta um exemplo de cálculo com operação vectoriais. Num processador sem operações vectoriais é necessário colocar o escalar num α registo (F0) e iterar sobre os elementos do vector X, carregando os elementos, um a um (LD), fazendo a multiplicação pelo escalar (MULTD), somar ao elemento correspondente de Y (ADDD) e guardar o resultado (SD). Num processador vectorial os vectores X e Y podem ser carregados numa só instrução vectorial (LV - Load vector), a multiplicação de um escalar pelo vector pode ser especificada através de uma só instrução (MULTSV), bem como a soma dos dois vectores (ADDV) e o armazenamento do resultado (SV).

MIPS	MIPS vectorial
LD F0, α	LD F0, α
ADDI R4, Rx, 512 # último elemento	
Cic: LD F2, 0 (Rx) # lê X(i)	LV V1, 0(Rx) # lê vector X
MULTD F2, F0, F2 # α X(i)	MULTSV V2, F0, V1 # α X
LD F4, 0(Ry) # lê Y(i)	LV V2, 0(Ry) # lê vector Y
ADDD F4, F2, F4 # α X(i) + Y(i)	ADDV V4, V2, V3 # add
SD F4, 0(Ry) # guarda Y(i)	SV V4, 0(Ry) # guarda Y
ADDI Rx, Rx, 8 # inc. índice X	
ADDI Ry, Ry, 8 # inc. índice Y	
SUB R20, R4, Rx # calcula limite	
BEQZ R20, Cic	

Figura 40 – Exemplo de um cálculo com operações vectoriais

No exemplo anterior assumiu-se que existiam registos que suportavam vectores de 64 elementos. Caso se pretenda efectuar uma operação sobre um vector de 128 elementos seria necessário dividir cada vector em duas partes de 64 elementos. Situações semelhantes aconteceriam para lidar com vectores de maior dimensão.

As arquitecturas vectoriais são um conceito bastante antigo, no entanto, apresentam como principal limitação o facto do seu desempenho estar dependente da existência de operações sobre vectores. Assim, este tipo de arquitectura apresenta muito bom desempenho numa classe específica de problemas, mas, em geral, o seu desempenho não é tão bom. Por esta razão, este tipo de arquitecturas não se encontra com um uso generalizado.

4.2 MMX/SSE/SSE2 vector para multimédia

As instruções para multimédia recuperaram recentemente o conceito de instruções vectoriais, mas aplicadas para operações sobre som e imagens (que não são mais do que vectores e matrizes de valores). Este tipo de instruções veio completar o conjunto de instruções existente. No caso dos processadores da Intel foram introduzidas instruções adicionais, sucessivamente enriquecidas. Assim, a Intel inicialmente introduziu as instruções MMX, seguida da SSE, SSE2 e recentemente de SSE3. Inicialmente a AMD optou por um conjunto próprio de instruções para multimédia, designado por 3D now, mas actualmente suporta também o conjunto de instruções da Intel.

O conjunto inicial, designado por MMX continha 57 instruções operando sobre registos de 64 bits que podiam representar vectores com 2 valores inteiros de 32 bits, 4 valores inteiros de 16 bits ou 8 valores inteiros de 8 bits (Figura 41).

As instruções MMX incluem operações para ler e escrever vectores na memória, aritméticas e lógicas entre vectores e conversão entre tipos de vectores:

- *Load* e *store* de 32 ou 64 bytes
- *Add, sub* em paralelo: 8 x 8 bits, 4 x 16 bits ou 2 x 32 bits
- Deslocamentos (sll, srl), And, And Not, Or, Xor em paralelo
- Multiplicação e mult-add em paralelo
- Comparações em paralelo (=,>)
- *Pack* – conversão entre tipos 32b <->16b, 16b <->8b

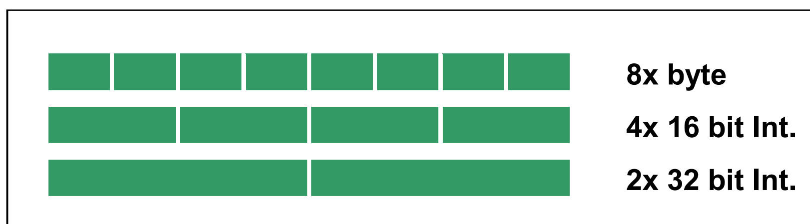


Figura 41 – Registos MMX

A título de exemplo a instrução PADDB mm1,mm0 adiciona dois registos mmx, onde cada registo contém 8 operandos de 1 byte. O resultado desta instrução é o seguinte:

mm0 [7..0] = mm0[7..0] + mm1[7..0]
mm0 [8..15] = mm0[8..15] + mm1[8..15]
mm0 [16..23] = mm0[16..23] + mm1[16..23]
…
mm0 [63..56] = mm0[63..56] + mm1[63..56]

Note-se que uma das principais diferenças entre uma instrução de ADD normal e uma ADD MMX de é que não existe *carry* entre os bits da instrução. Por esta razão existe uma adição com saturação (PADDSB), que quando existe *carry* o byte correspondente é colocado com o valor máximo. Esta operação é útil, por exemplo, para efectuar a adição de duas imagens, para que os *pixels* que excedem o valor máximo fiquem com esse valor.

Uma das principais limitações das instruções MMX originais era do facto dos registos vectoriais serem partilhados com os registos de operações sobre vírgula flutuante, não sendo possível a utilização simultânea de operações MMX e de vírgula flutuante. A segunda geração de instruções de MMX (designada por SSE) introduziu 8 registos específicos para as operações MMX, cada registo com 128 bits, e introduziu também operações sobre vírgula flutuante. A terceira geração (SSE2) aumentou a variedade de operações sobre operandos em vírgula flutuante, nomeadamente 2 *doubles* de 64 bits, 16 inteiros de 1 byte e 1 inteiro de 128 (Figura 42).

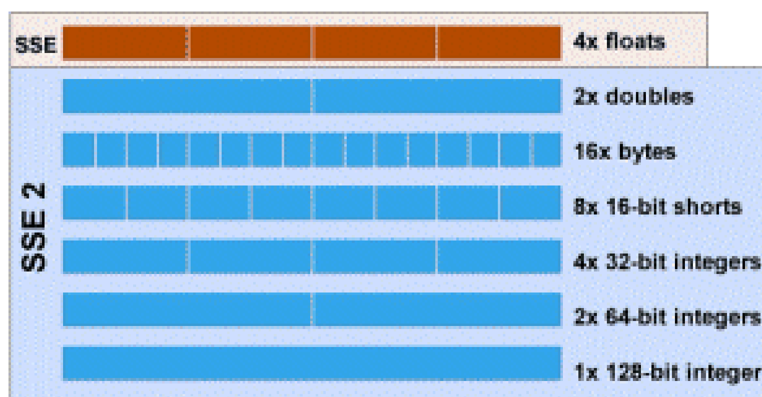


Figura 42 – Registos SSE/SSE2

4.3 VLIW – *Very Long Instruction Word*

As arquitecturas VLIW surgiram como uma alternativa às arquitecturas super-escalares, visando superar a crescente complexidade do escalonamento dinâmico, necessário para manter as unidades funcionais do processador a realizar trabalho útil.

O VLIW efectua um escalonamento estático, sendo o compilador responsável por indicar as instruções que podem ser realizadas em cada ciclo. Para tal, o formato de instrução deve poder especificar quais as operações a realizar em cada ciclo, designadamente, cada instrução por incluir vários *opcodes* e permitir a especificação de vários operandos, necessários para a execução de cada operação. As várias operações a realizar estão relacionadas com a unidades funcionais do processador, sendo, no caso limite, utilizado um *slot* da instrução para cada unidade funcional. A Figura 43 apresenta o formato de instrução utilizado da arquitectura Itanium da Intel.

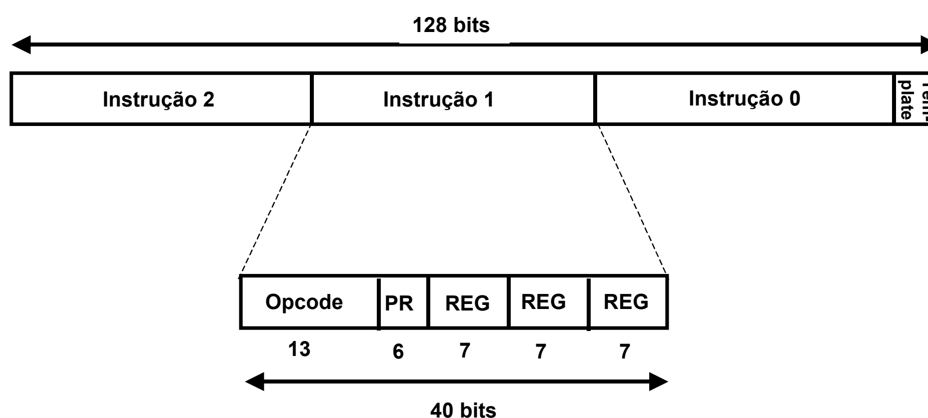


Figura 43 – Formato de instrução do IA-64 (Itanium)

As arquitecturas VLIW apresentam algumas limitações, relativamente às arquitecturas super-escalares:

1. O código gerado tende a ser de maior dimensão, porque é necessário inserir *nop* nos campos da instrução não preenchidos.

2. Compatibilidade de código entre gerações do mesmo processador é limitada, uma vez que tende a expor a arquitectura interna do processador

3. É mais penalizado com *stalls* do que o escalonamento dinâmico

A arquitectura Itanium da Intel introduziu algumas alterações relativamente aos VLIW clássicos para contornar as limitações anteriores:

- Cada instrução contém 128, permitindo especificar 3 operações, utilizando assim menos bits que VLIW clássico (LIW?), produzindo código mais compacto em 128 bits;

- As instruções podem ser ligadas entre si, permitindo especificar VLIW com um grau superior a 3 sem expor a arquitectura interna do processador;

- As implementações do processador incluem verificação de dependências em hardware para suportar compatibilidade com código existente.

Actualmente a arquitectura Itanium apresenta um sucesso muito limitado, essencialmente porque obriga à recompilação do código da aplicação para que o processador seja utilizado de forma eficiente.

4.4 Hyper-threading

O aumento de unidades funcionais nos processadores actuais não conduz a ganhos significativos de desempenho. Estudos indicam que o máximo e instruções que podem ser realizadas com paralelismo ao nível das instruções será na ordem de 10 instruções por ciclo. Adicionalmente, existem aplicações em que este número será inferior devido a uma quantidade elevada de dependências entre instruções.

Uma forma de aumentar o número de instruções independentes consiste em executar simultaneamente vários programas (ou fios de execução). Por exemplo, se um dado programa possuir em média 3 instruções sem dependências por ciclo a execução simultânea de outro programa com as mesmas características pode originar uma média de 6 instruções sem dependências por ciclo, uma vez que não existem dependências entre as instruções dos dois programas.

A execução de vários processos ou fios de execução designa-se por SMT (*simultaneous multi-threading*), ou *hyper-threading* na nomenclatura da Intel. Esta característica permite ao processador executar vários fluxos de instruções em simultâneo, normalmente simulando a existência de vários processadores. Este tipo de paralelismo apresenta um grão maior do que o paralelismo ao nível da instrução (>1000 instruções versus 10 instruções).

Segundo dados da Intel o *hyper-threading* com grau 2 (i.é., com a capacidade de executar simultaneamente dois fios de execução implica um aumento inferior a <5% de transístores por processador e conduzir a ganhos no desempenho até 35%. Estes ganhos são proporcionados por uma utilização mais eficiente dos recursos, uma vez que as ocupação média das unidades funcionais do processador aumenta devido a um menor número de dependências entre instruções (Figura 44).

Note-se que embora um processador com SMT se possa apresentar ao sistema operativo como vários processadores, fisicamente é um único processador, possuindo as unidades funcionais de um só processador. Por esta razão o desempenho com SMT não consegue atingir o desempenho de vários processadores (por exemplo, um duplo processador pode introduzir ganhos de 50%)

A principal razão para esta diferença de desempenho deve-se ao facto de a introdução de SMT num processador não ser acompanhada pelo aumento correspondente das unidades funcionais. Para que fosse possível obter o dobro do desempenho seria necessário TODOS os recursos do processador, nomeadamente, duplicar o número de unidades funcionais, de registos etc.

A duplicação de recursos e precisamente a abordagem utilizada nos processadores designados por multi-core, apresentados na secção seguinte.

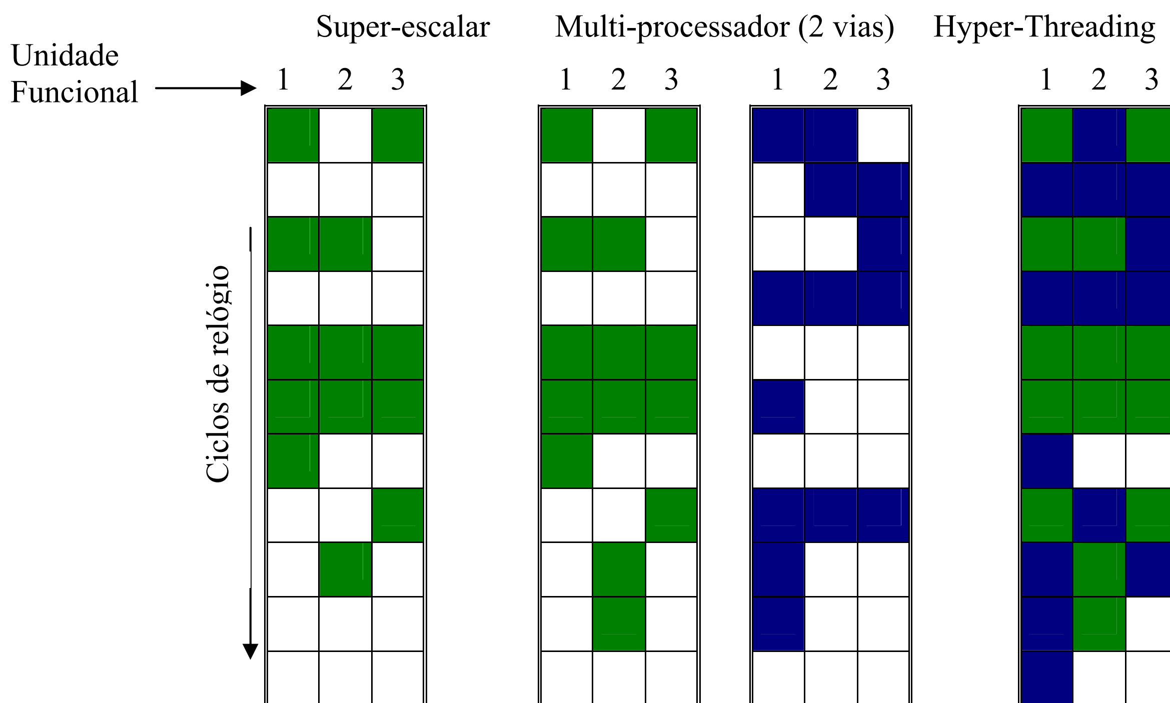


Figura 44 – Escalonamento de instruções com SMT

4.5 *Multi-core*

As arquitecturas *multi-core* consistem na inclusão no mesmo silício de vários processadores (designados por *core*). Actualmente apenas existem no mercado processadores *dual-core*, mas prevê-se a curto prazo o aparecimento de processadores com mais do que dois *core*.

Existem vários níveis de integração dos processadores num mesmo silício. Um aspecto particularmente interessante é a partilha das *caches*. A situação mais frequente é cada core possuir a sua própria cache de nível 1 e de nível 2. No entanto, existem arquitecturas, em que a cache de nível 2 é partilhada por vários processadores, nomeadamente a arquitectura Power da IBM.

Um outro aspecto também particularmente interessante é a conjugação dos *dual-core* com SMT. Recentemente o processador da Intel Pentium Extreme inclui estas duas características, o que permite que o processador se apresente ao sistema operativo como uma máquina com quatro processadores (dois *cores*, cada um com *hyper-threading*). O desafio está agora em desenvolver software que possa tirar partido deste tipo de arquitectura, uma vez que nas arquitecturas mais clássicas (i.é., com *pipelining* e super-escalaridade, explorando o paralelismo ao nível da instrução) o paralelismo é automaticamente extraído das aplicações (através do escalonador de instruções). Neste novo tipo de arquitecturas deve ser o programador a explicitar o paralelismo, quer ao nível dos fios de execução, quer ao nível dos processos.

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 uma índole mais teórico-prática. Com estes exercícios pretende-se consolidar os conhecimentos dos alunos sobre a execução encadeada de instruções com 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 várias arquitecturas 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 é formado por exercícios sobre a execução super-escalar de instruções e têm uma índole mais prática. 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

Considere o modelo de uma arquitectura MIPS com o *pipeline* da figura em anexo, sem unidade de encaminhamento de dados (*forward unit*), sem unidade de detecção de anomalias (*hazard detection unit*), com a escrita dos valores em registos (*write back*) na primeira metade do ciclo, com empates (*stall*) do *pipeline* em instruções de salto e com *cache* "quente" e de dimensão infinita. Considere agora o seguinte programa em *assembly* MIPS:

```
          mov $s1, $0
          mov $s2, $0
          mov $s0, -1024
soma:     lw $t0, 0xC400($s0)
          addu $s1, $s1, $t0
          sw $s1, 0xE400($s0)
          addu $s2, $s2, $s1
          addiu $s0, $s0, +4
          bne $s0, $0, soma
          mov $s1, $0
```

1) Identifique todas as dependências de dados existentes neste programa e introduza instruções *nop* necessárias para que a execução do programa esteja correcta. Calcule o CPI mínimo e o CPI efectivo na execução deste programa neste processador. Comente os resultados.

Sugestões:

a. Analise o significado e consequência da não existência das unidades de encaminhamento e de detecção de anomalias, bem como do facto de a operação de escrita dos valores em registo ser efectuada na 1ª metade do ciclo, permitindo a leitura do conteúdo dos registos na 2ª metade; com base neste conhecimento, é possível calcular quantas instruções deverão estar entre 2 instruções com dependências críticas de dados (que no caso deste MIPS são as de RAW).

b. Analise o significado e consequência do "empate" do *pipeline* em instruções de salto, i.e., da inserção de bolhas após a execução de qualquer instrução de salto (absoluto e relativo), logo após a sua descodificação; com base neste conhecimento, é possível calcular quantas bolhas são automaticamente inseridas por salto.

c. Considere uma *cache* "quente" aquela que já contém toda a informação que o programa necessita para a sua execução (neste caso o código do programa e a área dos dados); e se considerar ainda que ela tem dimensão infinita, então tem a certeza que toda essa informação coube na *cache*, e que nenhum acesso à *cache* de instruções ou de dados será penalizado no funcionamento do *pipeline*.

d. Conte o número de instruções *x* que são precisas executar para concluir o especificado.

e. Supondo que o *pipeline* estará sempre ocupado com tarefas úteis (modelo óptimo de arquitectura), verifique quantos ciclos de relógio são precisos para completar a execução (deverá dar mais 4 que o nº de instruções, devido à latência de execução da 1ª instrução); com base nos cálculos efectuados em d) pode-se obter o CPI_{min} - que deverá ser $(x + 4)/x$.

f. Identifique todas as dependências de dados entre as instruções, e veja, instrução a instrução, quantas "bolhas" são inseridas, quer através da execução de *nops* gerados pelo compilador para resolução das dependências de dados, quer ainda inseridas pela unidade de controlo de saltos no *hardware*; daqui pode contar e estimar o nº total de ciclos de relógio para executar este programa; usando como nº total de instruções a ser executadas no processador o valor *x* calculado em d. (que não é o correcto; porquê?), calcule agora o CPI efectivo.

2) Reordenando as instruções, rescreva o programa para minimizar o tempo de execução; calcule o tempo de execução do novo programa, em ciclos de relógio. Compare com o valor obtido em **1)** e comente.

Sugestão: altere a ordem de execução de instruções de modo a remover ao máximo as dependências de dados; de notar que cada instrução que escreve num registo obriga à inserção de 2 *nop* caso a instrução que se lhe segue precise de ler o conteúdo desse registo; portanto, se se conseguir alterar a ordem de execução de instruções de modo a existirem 2 instruções sem dependência de dados após uma operação que escreve num registo, a utilização do *pipeline* é melhorada; apenas com a simples reordenação de instruções deve conseguir eliminar pelo menos 3 instruções de *nop* (com um truque adicional - e que não compromete evoluções da microarquitectura - poderá ainda eliminar mais dois *nop*...).

3) Mostre o conteúdo do registo ID/EX no início do 6º ciclo de relógio.

Sugestão: identifique primeiro qual a instrução que no 5º ciclo se encontra no nível ID, sem se esquecer que na alínea anterior se conseguiu eliminar os *nops* no início do programa; agora consulte os apontamentos para construir essa instrução em binário e confirmar quais os sinais de controlo que a unidade de controlo gera nesse ciclo; o resto faz-se olhando apenas para o *pipeline* em anexo.

4) Considere agora que o compilador usa a técnica de desdobramento de ciclos (*loop unroll*) (para além das melhorias introduzidas em **2)**). Use-a para processar 4 elementos de cada vez, e reescreva o código de modo a minimizar o tempo de execução. Calcule o tempo de execução do novo programa, em ciclos de relógio. Compare com o valor obtido em **2)** e comente.

Sugestão:

a. A técnica de desdobramento de ciclos serve essencialmente 2 objectivos: minimizar as penalizações resultantes das instruções de controlo de execução do ciclo e espaçar mais as instruções que obrigam à inserção de "bolhas" no *pipeline* devido a dependências de dados.

b. Identifique as penalizações no funcionamento do *pipeline* resultantes das instruções de controlo de execução do ciclo (serão só as instruções de salto?...). Reordene as instruções do novo corpo do ciclo - que processa agora 4x mais dados por cada iteração - usando, se necessário, mais registos (e verificando se dispõe do nº extra de registos que são precisos) e modificando eventualmente o próprio código do programa.

c. Use estas dicas na reescrita do código; se conseguir que cada instanciação do ciclo não tenha mais de 18 instruções úteis e 2 *nops*, então chegou a um resultado excelente!

5) Considere agora que a arquitectura possui uma unidade de encaminhamento de dados (*forward unit*) e uma unidade de detecção de anomalias (*hazard detection unit*). Modifique a solução obtida em **1)** e calcule o tempo de execução do novo programa, em ciclos de relógio. Compare com os valores obtidos em **1)** e **2)** e comente.

Sugestão: lembre-se que a introdução da unidade de encaminhamento de dados vai permitir eliminar as bolhas nas dependências de dados críticas que ocorrem após a execução de operações aritméticas/lógicas; contudo, nas operações de *load* continua a haver a necessidade de o compilador inserir um *nop*; mas, se a arquitectura possuir uma unidade de detecção de anomalias, é a própria unidade de controlo da arquitectura que se encarrega de introduzir uma bolha nesses casos de dependência crítica de dados, simplificando a tarefa de geração de código do compilador (mas não melhorando o desempenho na execução...)

6) Se o valor de CPI obtido nos programas resultantes da resolução de **4)** e de **5)** fossem iguais, que conclusões tiraria quanto ao desempenho do processador na execução deste programa? (Mesmo desempenho ou distinto, e porquê.)

Sugestão: lembre-se que a principal medida do desempenho do processador na execução deste programa é o tempo que ele necessita para o executar (em ciclos de relógio), e um bom tempo não depende apenas de uma boa "máquina" - o processador - mas também de um bom "condutor" - o código gerado pelo compilador; com base nesta informação e na fórmula que nos relaciona o tempo de execução de um programa com os parâmetros da arquitectura, pode responder à questão...

7) Considere ainda a execução atrasada da instrução de salto, i.e., a arquitectura introduz um slot de salto retardado (*branch delay slot*) (para além dos melhoramentos já introduzidos na arquitectura em alíneas anteriores). Reescreva o código de modo a minimizar o tempo de execução e calcule esse tempo, em ciclos de relógio. Compare com o valor mínimo obtido em **1)** e a melhor optimização que tinha conseguido em **5)**, e comente.

Sugestão: por execução atrasada de uma instrução - de salto ou de *load* - entende-se que uma instrução só é efectivamente executada após a instrução que a segue; no caso da instrução de salto, isto significa que, quer o salto se concretize ou não, a instrução que se encontra na palavra seguinte da memória de instruções é sempre carregada e executada; como consequência para este processador, verifica-se que, em vez da arquitectura inserir 3 bolhas após a descodificação de um *branch* (ou 1 bolha após um *jump*), a unidade de controlo deixa executar a instrução que está imediatamente a seguir e apenas introduz 2 bolhas (no caso dum *branch*, e 0 no caso dum *jump*)

8) Considere ainda uma nova optimização da arquitectura relativamente a **7)**: uma previsão estática de saltos - nunca salta. Reescreva o código de modo a minimizar o tempo de execução e calcule esse tempo, em ciclos de relógio.

Sugestão: mesmo com *branch delay slot* cada instrução de salto condicional pode ainda desperdiçar 2 instruções (bolhas) quando a previsão estática falha; mas com saltos incondicionais isto já não acontece; estas dicas servem para alguma coisa?

9) Considere agora como alternativa a **8)** uma previsão dinâmica de saltos (com 1 bit): se da vez anterior saltou, também salta agora e para o mesmo endereço, se não saltou, também não salta. Reescreva o código de modo a minimizar o tempo de execução e calcule esse tempo, em ciclos de relógio. Compare com o valor obtido em **8)** e comente.

10) Considere agora, em adição a **9)**, uma arquitectura superescalar com três unidades de execução: i) execução de acessos à memória (*load* e *store*); ii) execução de operações aritméticas e iii) execução de saltos. Qual o CPI mínimo teórico e qual o CPI efectivamente obtido na execução do programa? Comente os resultados. Qual a melhoria na execução deste programa se for introduzida uma segunda unidade para execução de operações aritméticas, passando a arquitectura a possuir quatro unidades de execução?

5.2 Super-escalaridade

5.2.1 Introdução

A medição de desempenho pretende quantificar a quantidade de trabalho que um computador consegue realizar por unidade de tempo. Assim, é possível comparar o desempenho de várias máquinas e seleccionar a mais adequada para uma determinada função.

O desempenho pode ser medido a vários níveis, nomeadamente, ao nível das aplicações, determinando o número de soluções de um determinado problema calculadas por unidade de tempo (utilizando programas reais ou sintéticos, ex. SPEC2000, Dhrystones), ao nível do ISA, determinando a quantidade instruções efectuadas por segundo (MIPS, MFLOPS) e ao nível do *datapath*, utilizando a frequência de relógio ou a largura dos barramentos do processador.

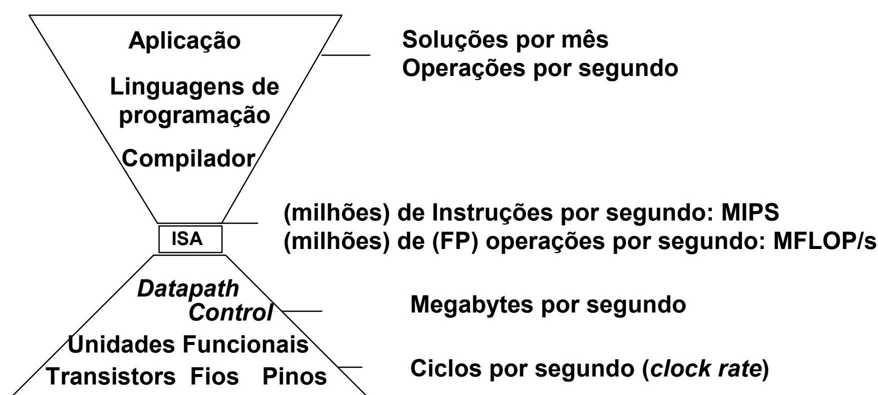


Figura 45 – Níveis de avaliação de desempenho

A medida mais indicada de desempenho é aquela que é obtida com aplicações reais, as quais serão efectivamente utilizadas na máquina que se pretende avaliar. Mesmo assim, as máquinas podem apresentar desempenhos diferentes consoante o tipo de aplicação. Por exemplo, as aplicações que envolvem gráficos dependem fortemente do desempenho da placa gráfica, enquanto as aplicações que envolvem bases de dados dependem mais do subsistema de disco. Por esta razão, actualmente o SPEC2000 é constituído por um conjunto de aplicações que pretendem medir o desempenho das máquinas em vários tipos de aplicações, permitindo seleccionar aquela que apresenta melhor desempenho no tipo de aplicações em que irá ser utilizada.

A utilização de programas sintéticos deve ser evitada, uma vez que frequentemente o desempenho obtido pouco tem a ver com o desempenho obtido com aplicações reais.

A medição do desempenho ao nível do ISA (em MIPS ou MFLOPS) apenas poderá ser utilizada quando se pretende comparar processadores que implementam um mesmo ISA. A utilização de medições obtidas a este nível para comparar processadores com ISA diferentes é errónea porque a quantidade de informação processada por cada instrução pode não ser igual nas duas arquitecturas.

A medida de desempenho menos adequada é a frequência de relógio, uma vez que a frequência de relógio pode não ter uma correspondência directa com o desempenho de aplicações reais. Recorde-se que o tempo de execução é dado por #I x CPI x Tcc.

As medições de desempenho realizadas nestes exercícios efectuam principalmente uma avaliação ao nível do ISA, não com o intuito de inferir qual o desempenho com aplicações reais mas com o intuito de determinar características do *datapath* do processador que terão impacto no tempo de execução de aplicações reais. Adicionalmente são realizados testes a subsistemas do computador.

5.2.2 Medição de tempo

A medição de tempo em computadores envolve a utilização de um relógio do sistema para medir o tempo de decorre entre o início e o fim de uma actividade. A precisão do relógio é a diferença entre o tempo medido pelo relógio do sistema e o tempo efectivamente decorrido. Por exemplo, o sistema pode indicar que decorreram 1003 segundos desde o início do programa mas na realidade podem apenas ter decorrido 1000 segundos. A resolução do relógio é a unidade de tempo mínima que decorre entre valores do relógio. Frequentemente a resolução de um relógio de pulso é na ordem do segundo, enquanto os relógios do computador têm uma resolução de milisegundos ou de microsegundos.

A resolução do relógio limita a duração mínima de eventos que podem ser medidos. Por exemplo, um relógio com uma precisão de 1 ms não permite a medição de eventos com duração inferior aos milisegundos. No entanto, esta limitação pode ser contornada se for possível medir a duração de múltiplas ocorrências do mesmo evento.

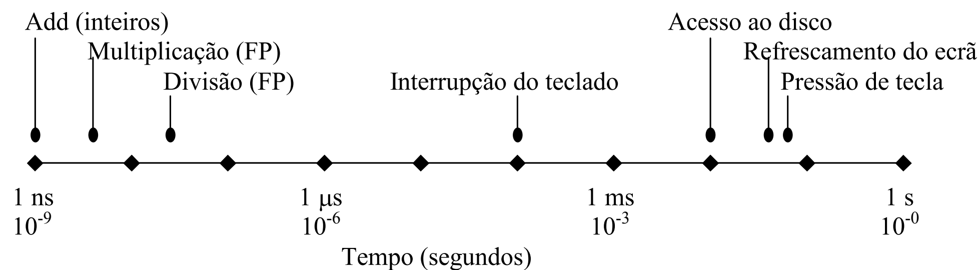


Figura 46 – Escala de tempo de eventos (Máquina a 1 GHz)

O processo de medição deve considerar eventuais sobrecargas introduzidas pelas rotinas de medição, nomeadamente, ao tempo medido deve ser descontado o tempo necessário para efectuar a chamada à rotina do relógio e o tempo introduzido por eventuais ciclos para repetir o evento em medição.

A medição de eventos cuja duração está entre os 10 milisegundos e os segundos é particularmente complexa, devido ao escalonamento de tarefas. Tal resulta do facto de a tarefa em medição poder ser suspensa e escalonada uma nova tarefa. Quando a duração da tarefa é superior a vários segundos este impacto é reduzido. Adicionalmente, no processo de medição devem ser efectuadas várias repetições e utilizar uma média de todas as medições ou das K melhores medições. Desta forma é minimizado o impacto quer do escalonamento, quer de outras tarefas realizadas pelo sistema operativo.

A microarquitectura Intel P6 (do Pentium Pro e posteriores) introduziu um relógio de elevada resolução, constituído por um contador de 64 bit que indica o número de ciclos decorridos desde que o computador foi ligado. Numa máquina de 1 GHz este contador retorna a zero (i.e., "dá a volta") após $2^{64} - 1$ segundos, ou seja, 570 anos. Note-se que se fosse implementado um contador apenas com 32 bits, numa máquina a 1 GHz, este voltaria a zero a cada 4,3 segundos.

O contador de ciclos de relógio pode ser acedido através da instrução rtdcs (*read time stamp counter*), cujo código máquina é 0x0F31. Esta instrução coloca os 32 bits mais significativos no registo EDX e os 32 bits menos significativos em EAX. A instrução rtdcs apenas está acessível quando o processador está no modo 32 bits.

Note-se que, actualmente, o contador de ciclos de relógio também está implementado nos processadores AMD Athlon e Duron, além dos já referidos Pentium Pro, Pentium II, Pentium III, Pentium IV e Pentium Xeon.

5.2.3 Resumo de conjunto de instruções IA-32

A arquitectura IA-32 apresenta 8 registos com 32 bits de uso genérico. Por questões de compatibilidade alguns dos registos podem ser divididos em registos e 16 bits, que por sua vez podem ser divididos em registos de 8 bits. Existe também um registo de controlo e de estado utilizado para controlar a execução do processador e implementar as condições de salto e um apontador de instruções.

Registos de uso genérico

32 bits	16 bits	31 … 16	15 … 8	7 … 0
EAX	AX		AH	AL
EBX	BX		BH	BL
ECX	CX		CH	CL
EDX	DX		DH	DL
EBP	BP		BP	
ESI	SI		SI	
EDI	DI		DI	
ESP	SP		SP	

Registo de controlo e de estado

EFLAGS	FLAGS		FLAGS

Apontador de instruções

EIP	IP		IP

O conjunto de instruções IA-32 apresenta múltiplos modos de endereçamento: imediato, registo, indirecto por registo, indirecto por valor imediato, indirecto por base + deslocamento, etc. A figura seguinte apresenta as alternativas para especificar os operandos em memória:

$$\overset{base}{\begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \\ ESP \end{bmatrix}} + \overset{índice * escala}{\begin{bmatrix} \begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \end{bmatrix}} + \overset{desloc}{\begin{bmatrix} nenhum \\ 8bits \\ 16bits \\ 32bits \end{bmatrix}}$$

$$endereço = [base + índice * escala + desloc]$$

Base, índice, escala e deslocamento são todos opcionais.

Alguns exemplos de endereços:

[eax]
[ebx + ecx]
[edx + 8]
[ebp + esi +12]
[esp + edi*4 + 4].

O conjunto de instruções IA-32 possibilita a referência a operandos em memória (contrariamente à arquitectura MIPS que apenas permite acesso à memória através de instruções de *lw* e *sw*). Frequentemente, devido ao número limitado de registos é necessário colocar variáveis do programa em memória. O seguinte código compara um programa em *assembly* MIPS com o mesmo programa em IA-32:

C	**MIPS**	**IA32**
`int i;`	`addi $s0, $s0, 1`	`i: dw 0`
`i= i + 1;`		`LEA  EBX, i`
		`INC dword ptr [EBX]`

Em IA-32 é implementado um modelo baseado no registo de *flags*, um registo que memoriza informações sobre o resultado da última operação. Entre as várias condições armazenadas neste registo incluem-se a indicação do sinal do resultado e a igualdade a zero. As instruções de salto condicional utilizam a informação contida no registo de *flags*. A instrução CMP (*compare*) compara o valor de dois operandos, apenas alterando o registo de *flags*.

O seguinte código compara um trecho de programa em *assembly* MIPS com o mesmo trecho em IA-32:

C	**MIPS**	**IA-32**
`if (i<12)`	`slt $t0, $s0, 12`	`CMP dword ptr [EBX],12`
`...`	`beq $t0, $0, else`	`JGE else`

O conjunto de instruções IA-32 utiliza em regra dois operandos, o primeiro funciona geralmente como fonte e destino, enquanto o segundo funciona apenas como fonte. A fonte pode ser um registo, um valor imediato ou um endereço de memória. O destino é um registo ou um endereço de memória. Quando o destino é um endereço de memória a fonte não pode ser também um endereço de memória. A tabela seguinte apresenta algumas das instruções do IA-32 mais relevantes:

```
Src = Rsrc | Mem | Imm32
Dest = Rdest | Mem
Dest e Src não podem ser simultaneamente memória
Rdest, Rsrc = EAX | EBX | ECX | EDX | ESI | EDI | EBP
Mem = [ base + índice * escala + Imm32 ]

    mov Dest, Src          # Dest = Src
    lea Rdest, End         # Rdest = End
    add Dest, Src          # Dest = Dest + Src
    sub Dest, Src          # Dest = Dest - Src
    and Dest, Src          # Dest = Dest & Src
    inc Dest               # Dest = Dest + 1
    dec Dest               # Dest = Dest – 1
    imul Rdest, Src        # Rdest = Src * Rdest
    cmp Rsrc1, Src2        # F = resultado da comparação
    j<cc> End              # <cc> = [L, LE, G, GE, E, NE]
    jmp End                # EIP = End
```

A generalidade dos compiladores de C/C++ permite a utilização de *assembly* embutido. Para tal basta incluir o *assembly* na primitiva _asm { .... }. Adicionalmente é

possível referenciar variáveis do programa C nas primitivas em *assembly*. Por exemplo, o código seguinte copia o conteúdo do registo EAX para a variável i:

```
int i;
_asm { mov i, eax }
```

É também possível forçar um determinado código, mesmo que o *assembler* não reconheça o *assembly* correspondente (por exemplo o *rtdcs*)

```
_asm {
      _emit 0x0f
      _emit 0x31 }
}  // RTDSC
```

O código anterior funciona no compilador da Microsoft Visual Studio C++ 6.0 e força a emissão do *opcode* correspondente à instrução *RTDSC*. Em compiladores mais recentes esta instrução *assemby* pode ser colocada directamente no programa.

5.2.4 Exercício 1 – Processador

Objectivo:

- Medir as características de desempenho básicas do processador
- Identificar as sequências de instruções necessárias para determinar cada característica do processador

Pretende-se com este exercício desenvolver um programa em C, com *assembly* embutido, que avalie várias características do processador, nomeadamente a frequência do relógio, o CPI de instruções aritméticas e de *load* e a penalização (i.e., ciclos de *stall*) decorrente das dependências de dados do tipo RAW e das dependências de controlo.

Desenvolva um programa que permita medir o seguinte:

a) frequência de relógio do processador. Implemente primeiro uma rotina para aceder ao contador de ciclos de relógio do processador, utilizando a instrução *RTDSC*. Depois pode utilizar essa rotina em conjunto com a função do Windows *GetTickCount*, que permite obter o tempo decorrido, para determinar a frequência de relógio.

b) CPI da instrução LOOP *end* (que decrementa o registo ECX e salta para o endereço *end* se ECX for diferente de 0) e o CPI obtido utilizando como alternativa o par de instruções DEC ECX e JNZ end. Poderá obter o número de ciclos com precisão se colocar inicialmente em ECX um valor relativamente elevado. O número de ciclos necessário para cada iteração do ciclo será igual ao número de ciclos necessário para executar todo o ciclo dividido pelo número de iterações.

c) CPI das instruções ADD, IMUL e MOV de memória para registo. Para tal altere o ciclo da alínea anterior por forma a que este inclua várias instruções ADD, IMUL ou MOV **independentes** entre si.

d) CPI das instruções ADD, IMUL e MOV quando existem dependências RAW. Para tal altere o ciclo da alínea anterior para que este inclua várias instruções ADD, IMUL ou MOV **dependentes** entre si. No caso do MOV, para que cada instrução dependa do resultado da anterior, deve ser percorrida uma estrutura de dados em memória do tipo lista ligada.

e) penalização decorrente de um salto previsto erradamente. Para tal desenvolva um ciclo que contenha um *if* que é sempre executado e compare-o com um ciclo que contém um *if* dependente de um valor aleatório (obtido com *rand*). A diferença de CPI entre as duas versões corresponde a metade da penalização dos saltos previstos erradamente, uma vez que no segundo caso a previsão falha 50% das vezes.

5.2.5 Exercício 2 – Hierarquia de memória

Objectivo:

- Medir a desempenho dos vários níveis da hierarquia de memória

Pretende-se com este exercício desenvolver um programa em C, possivelmente com *assembly* embutido, que meça o desempenho vários níveis da hierarquia de memória, desde a cache L1 até um disco na rede local.

Desenvolva um programa que permita medir o seguinte:

a) número médio de ciclos do processador necessário para aceder à memória cache nível 1, nível 2 e RAM. Simultaneamente poderá determinar a dimensão da cache nível 1 e nível 2.

b) número médio de ciclos do processador necessário para transferir cada unidade de informação do disco duro. Utilize a função *fread* para ler blocos de informação do disco.

c) idêntico à alínea b) mas criando o ficheiro numa rede local (por exemplo, através de um directório partilhado).

5.2.6 Exercício 3 – Codificação de software

Objectivo:

- Medir o impacto no desempenho de diversas formas de codificação de software

Pretende-se com este exercício desenvolver um programa em C que compare o desempenho de duas funções codificadas de forma distinta, explicando a proveniência das diferenças obtidas no desempenho.

Desenvolva um programa que permita medir o seguinte:

a) comparar o desempenho de uma função que calcula o factorial de forma iterativa e uma função que o calcula de forma recursiva. Justifique os resultados obtidos, possivelmente recorrendo à visualização do *assembly* gerado para cada uma das duas funções.

Função iterativa	Função recursiva

`int factIter(int n) {`
`  int f=1;`
`  while(n > 1) { f*=n; n--; }`
`  return(f);`
`}`
	`int factRec(int n) {`
	`  if (n <= 1 ) return(1);`
	`  return( n * factRec(n-1) );`
	`}`

b) comparar o desempenho de uma função que some os elementos de um *array* de duas dimensões, efectuando a soma por linhas e uma função que efectue a mesma soma dos elementos por colunas. Justifique os resultados obtidos.

Soma por linhas	Soma por colunas

`soma=0;`
`for(int i=0; i<M; i++)`
`   for(int j=0; j<N; j++)`
`      soma += a[i][j];`
	`soma=0;`
	`for(int j=0; j<N; ij++)`
	`   for(int i=0; i<M; i++)`
	`      soma += a[i][j];`