Pessimists, we're told, look at a glass containing 50% air and 50% water and see it as half empty. Optimists, in contrast, see it as half full. Engineers, of course, understand the glass is twice as big as it needs to be.

Bob Lewis

4

ARQUITETURA AVANÇADA DE COMPUTADORES

universidade de aveiro
theoria poiesis praxis

Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte**. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Arquitetura de Computadores Avançada, tal como foi lecionada, no ano letivo de 2016/2017, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

Dadas as disciplinas de Arquitetura de Computadores I (a2s1), Arquitetura de Computadores II (a2s2) e Sistemas de Operação (a3s1) temos agora todo um conjunto de conhecimentos necessário para avançar para a análise das tecnologias da vanguarda em termos de arquitetura de computadores. É assim objetivo desta disciplina, que se retrata neste documento, introduzir os conceitos mais relevantes em termos de *design* das mais recentes gerações de processadores e como estes afetam o desempenho de um sistema computacional, descrever a organização da hierarquia de memória (mais em particular da memória virtual e da cache) e ambientar-nos com ferramentas de simulação em arquitetura de computadores e como é que estas nos podem servir parar atestar o desempenho de um sistema [1].

Tal como referido anteriormente é objetivo então estudar a arquitetura e saber descrever a organização de um sistema computacional. Então mas há diferença entre arquitetura e organização? Vejamos um caso análogo: comparemos assim a profissão de arquiteto com a profissão de engenheiro civil. Qual é a diferença entre ambos? Como sabemos um arquiteto tem como profissão conceber uma ideia a ser realizada - desenhá-la e estudar planos estéticos e funcionais para uma construção, quer esta seja um edifício ou um jardim. Do lado do engenheiro civil este irá tomar as projeções do arquiteto e tentar otimizá-las para o contexto de uma construção, isto é, irá assegurar-se que todas as variáveis do sistema cumprem os requisitos mínimos de segurança, de funcionalidade, entre outros... Sendo análogo à diferença entre arquitetura e organização, podemos dizer que a **arquitetura** está presente quando referimos, numa abstração, a criação de componentes e a junção de vários componentes para a elaboração de um só, obtendo maior eficácia, enquanto que no lado da **organização** (o engenheiro civil) temos a exploração de todo um espaço para desenhar os vários componentes na procura dos melhores resultados de desempenho, obtendo maior eficiência.

arquitetura

organização

Na história desta área de estudo, essencial para toda a informática se desenvolver, tal como já vimos noutras disciplinas anteriores, a evolução dos equipamentos (em mais particular dos processadores) tem vindo a respeitar a **Lei de Moore** sobre a qual se enuncia que o número de transístores num circuito integrado duplica aproximadamente de dois em dois anos. Por consequência, estima-se que o desempenho dos *chips* duplica de 18 em 18 meses (ano e meio). Na Figura 0.1 podemos ver um gráfico onde se representa a descida do tamanho dos transístores ao longo dos anos, desde o ano de 1970 (os valores são aproximados) [2].

Lei de Moore
⊙ **Gordon Moore**

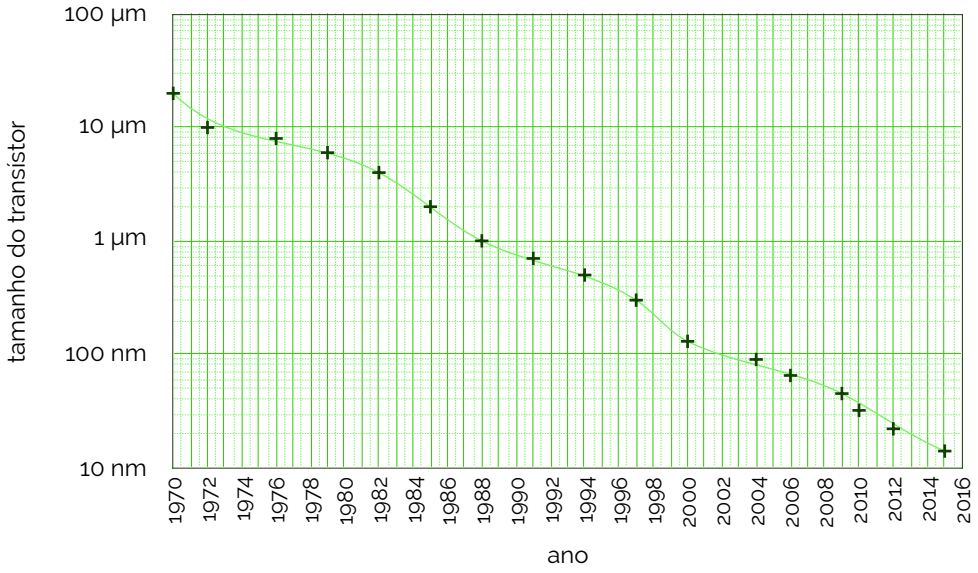


figura 0.1

Olhando para o gráfico da Figura 0.1 podemos reparar que em poucos anos o tamanho de um transístor tornou-se cerca de 10 mil vezes menor. Por curiosidade, podemos também verificar que o tamanho de um transístor tornou-se muito menor que o com-

primento de onda do espetro de luz visível, que está compreendido entre os 390 e 700 nm [3], o que nos revela a necessidade que a determinada altura houve de adquirir ferramentas espciais e dedicadas para o tratamento e criação dos processadores e restantes equipamentos que recebem estes transístores.

1. Avaliação do Desempenho dos Sistemas de Computação

Para que possamos definir se um determinado sistema é melhor que um outro há que usar um conjunto de métodos para averiguar o caso. Assim sendo há primeiro que estipular o que significa, neste contexto, ser **melhor**.

melhor

Consideremos um aeroporto, onde nos encontramos perante quatro aviões: um Boeing 777, um Boeing 747, um BAC Concorde e um Douglas DC-8. Qual dos quatro aviões é o melhor? Se estivermos a falar de rapidez, então o avião supersónico Concorde é o melhor, com uma velocidade máxima estimada de 2172 km/h. Mas, de todos os aviões listados, este é o que menos pessoas leva (levando apenas 132 pessoas) - será que o número de pessoas não conta? E a autonomia em viagem? Aqui quem vence é o Douglas DC-8, com 14033 quilómetros de autonomia. Podemos assim verificar que nem sempre "melhor" significa velocidade, capacidade ou autonomia - "melhor" pode assumir muitos papéis, no entanto há que avaliar tendo em conta um fim.

Para serem definidas formas de avaliação, foram sendo criadas, ao início, **medidas** importantes a reportar, acerca de um sistema, dado que há um conjunto de fatores que influenciam o desempenho como a adequação de um *instruction set* a um determinado problema, a execução de instruções pelo processador, entre outros... [4].

medidas

A potência e a energia em circuitos integrados

Uma das primeiras grandezas que devemos ter em conta, na aprendizagem da avaliação do desempenho de sistemas computacionais, é a **potência**. Numa evolução que durante anos teve a sua grande diferença pelo tamanho dos transístores (em inglês *feature size*), outras desvantagens começaram a surgir, como a larga dissipação do calor que se começava, de ano para ano, a propagar entre os vários componentes.

potência

Como foi abordado em Sistemas Eletrónicos (a2s2), nos *chips* de tecnologia CMOS o consumo de energia é comutado com interruptores que são os transístores, à qual damos vulgarmente o nome de **energia dinâmica**. Vimos então que a energia necessária por transístor era proporcional ao produto da sua capacidade com o quadrado da sua tensão, como podemos ver em (1.1).

energia dinâmica

$$\text{energia}_{dinâmica} \propto \text{capacidade} \cdot \text{tensão}^2 \tag{1.1}$$

Também já devemos saber que capacidade de carga é função do número de transístores ligados a uma saída e da mesma tecnologia, que por si determina a capacidade elétrica dos fios de ligação e dos transístores. Por outro lado, a **potência dinâmica** necessária por transístor é o produto da energia de uma transição pela frequência destas, como podemos ver em (1.2).

potência dinâmica

$$\text{potência}_{dinâmica} \propto \text{capacidade} \cdot \text{tensão}^2 \cdot \text{frequência de comutação} \tag{1.2}$$

Mas tanto a energia dinâmica, como a potência dinâmica, são facilmente reduzidas através da redução da tensão, isto porque é uma das parcelas da proporcionalidade, daí que em cerca de 20 anos de desenvolvimento as tensões tenham descido dos 5V para o 1V em termos de potencial elaborado para os processadores, o que do mesmo modo permitiu que a frequência aumentasse sem prejuízo maior. O que acontece há uns anos é que com a tão baixa tensão fornecida aos processadores, estes estão a tornar-se cada vez mais lentos. O problema agora está no facto de não existir apenas consumo de energia dinâmica, mas

também coexistir **energia estática**, mesmo quando as máquinas se encontram desligadas. Em termos de servidores, por exemplo, a perda de energia estática é tipicamente responsável por 40% do consumo de energia total. Contudo, aumentar o número de transístores aumenta ainda mais a necessidade de maior dissipação de potência, mesmo quando os transístores estão todos desligados. Esta é a razão principal pela qual os armazéns que possuem servidores investem tanto no arrefecimento dos seus equipamentos.

energia estática

No entanto não é só nos servidores que estes cuidados existem. Hoje em dia, os microprocessadores já oferecem diversas técnica para melhorar a eficiência energética, apesar das baixas frequências de relógio e fontes de tensão constantes. Uma possibilidade (a mais aplicada) é a de manter um registo de todas as operações que são realizadas, de modo a que se possa desligar o relógio de alguns módulos que se encontram inativos. Outro exemplo será mesmo disponibilizar uma variedade de frequências de relógio e tensões de forma a poder equilibrar o consumo real ao necessário (técnica denominada de **DVFS**, sigla para *Dynamic Voltage-Frequency Scaling*). Este último caso é aquele que se aplica na maior parte dos computadores portáteis, nos quais o desempenho ligado à corrente e sobre a bateria é completamente diferente. Já a Intel começou a oferecer uma técnica à qual eles denominaram de *Turbo Mode*, na qual o *chip*, por hardware, decide se é seguro trabalhar a uma frequência de relógio mais alta, tipicamente cerca de 10% mais que a frequência-padrão para um determinado processador, por um curto período de tempo, possivelmente em poucos *cores*, até que a temperatura comece a subir [5]. Dá-se o exemplo do computador onde este documento foi redigido, que com um processador Intel i7 de 2.9GHz tem uma frequência de 3.6GHz em *Turbo Mode*.

DVFS

Dependências

Em termos históricos os circuitos integrados foram um dos componentes mais fiáveis de um computador. Apesar de possuírem pinos e, por consequência, fragilidades, tal como pequenas baixas nos serviços de comunicação, a taxa de erro de transmissões intra-circuito era bastante baixa [6]. Quando começámos a ter tamanhos (*feature sizes*) de 32 nm para baixo, as falhas começam a tornar-se bastante comuns, pelo que os arquitetos de sistemas computacionais adquiriram novos desafios a enfrentar. Estes desafios acontecem porque um determinado componente, embora tenhamos a necessidade de ver tudo como uma abstração e assim considerar um único módulo, têm **dependências** de outros, como fontes de alimentação, entre outros controladores.

dependências

Para avaliarmos quando é que uma determinada dependência tem uma falha temos de verificar primeiro em que fase é que o sistema está, de facto, a funcionar como devido. Assim convém descrever duas fases que surgiram aquando da proliferação da Internet, a que denominamos de **SLA** (sigla para *Service Level Agreements*) - que usamos para descrever a garantia de serviços fornecidos - e **SLO** (sigla para *Service Level Objectives*) - que usamos para garantir que as dependências estão viáveis para funcionamento. Em respeito, mais particular, com os SLA, os sistemas geralmente alternam entre dois estados de serviços, entre os quais o estado de **serviço cumprido**, onde o serviço é fornecido tal como pedido, e o estado de **serviço interrompido**, onde o serviço que está a ser fornecido é diferente daquele que o SLA garante. As transições que possam existir entre estes dois estados são globalmente denominados de **falhas** (do primeiro estado para o segundo) ou **restauros** (do segundo estado para o primeiro). Para podermos classificar tais acontecimentos precisamos de concetualizar duas medidas de dependência: a fiabilidade de um módulo e a disponibilidade de um módulo.

SLA
SLO
serviço cumprido
serviço interrompido
falhas
restauros

Quando referimos a **fiabilidade de um módulo** temos de considerar que pretendemos medir o tempo de cumprimento de um serviço que é fornecido de forma contínua, ou, dualmente, o tempo de incumprimento (ou falha), a partir de um instante considerado como referência inicial. Assim estabelecemos um tempo médio para falha, **MTTF**, e um seu recíproco, a taxa de falhas, geralmente referida como falhas por milhar de milhão de horas de operação, ou **FIT** (*failures in time*): um exemplo está se o MTTF for considerado

fiabilidade de um módulo
MTTF
FIT

de 1 000 000 horas, então o FIT será de $10^9/10^6$ ou 1 000. A interrupção de um sistema também é medida como o tempo médio de reparação, **MTTR**, sendo que entre as falhas, o tempo médio entre falhas, **MTBF**, é simplesmente a soma do MTTR pelo MTTF.

MTTR
MTBF

Por outro lado, a **disponibilidade de um módulo** é avaliada pela operação do módulo de forma contínua, em relação à conjunção entre os períodos de serviço e os de interrupção. Para obter esta medida considerando sistemas não redundantes, temos que a disponibilidade de um módulo é facilmente obtida pelo quociente entre o MTTF e a soma de MTTF por MTTR: MTTF / (MTTF+MTTR).

disponibilidade de um módulo

Consideremos assim o Exercício 1.1 onde considerando um subsistema com 10 discos (cada um classificado com 1 000 000 horas de MTTF), 1 controlador ATA (classificado com 500 000 horas de MTTF), 1 fonte de alimentação (classificada com 200 000 horas de MTTF), 1 ventoinha (classificada com 200 000 horas de MTTF) e um cabo ATA (classificado de 1 000 000 MTTF) se pretende medir o MTTF do sistema num todo. Para tal precisamos de verificar a *failure rate* do subsistema, sendo este o quociente de 1 pelo MTTF de cada material (1.3).

exercício 1.1

$$\text{failure rate}_{subsistema} = 10\frac{1}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000} = \frac{23}{1000000 \text{ horas}} \quad (1.3)$$

Sendo que o MTTF é o inverso do *failure rate*, então em (1.4) podemos ver o inverso de (1.3), obtendo assim um valor aproximado de 43 500 horas.

$$\text{MTTF}_{subsistema} = \frac{1}{\text{failure rate}_{subsistema}} = \frac{1000000}{23} \approx 43500 \text{ horas} \quad (1.4)$$

Uma boa forma de cortar com as falhas é criando **redundância**, tanto a nível temporal (repetindo uma determinada operação até ver que não ocorre erro) como a nível de recursos (ter outros componentes a efetuar funções repetidas, para assegurar que uma determinada funcionalidade não é quebrada tão facilmente). Uma vez o componente substituído e o sistema totalmente reparado, a dependência de um sistema assume-se tão boa como nova. Vejamos o Exercício 1.2, onde consideramos os discos de subsistemas, que geralmente têm fontes de alimentação redundantes para melhorar as dependências. Usando os componentes e os MTTF's do Exercício 1.1, como é que podemos calcular a fiabilidade das fontes de alimentação redundantes? Consideremos também, para o caso, que uma fonte de alimentação é suficiente para executar o disco do subsistema e que pretendemos adicionar uma fonte redundante. Nós precisamos de uma fórmula que mostre o que esperar quando conseguimos tolerar uma falha, mas manter o serviço a ser fornecido. Para simplificar os cálculos, assumamos que os tempos de vida dos componentes são exponencialmente distribuídos e que não existem dependências entre as falhas de componentes. O MTTF para as nossas fontes de alimentação redundantes é o tempo médio até que uma fonte falhe, dividido pela hipótese que a outra irá falhar antes da substituição desta. Assim sendo, se a hipótese de uma segunda falha antes da reparação é pouco provável, então o MTTF do par é bastante grande. Sendo que temos duas fontes de alimentação e falhas independentes, o tempo médio até que um disco falhe é de MTTF/2. Uma boa aproximação da probabilidade de uma segunda falha é o MTTF sobre o tempo médio até que a segunda fonte falhe. Assim, uma aproximação razoável seria a que podemos ver em (1.5). Usando os valores de MTTF de (1.5), se assumirmos que um humano demora cerca de 24 horas para reparar que uma fonte de alimentação falhou e carece de substituição, então a fiabilidade da falha é (1.6), fazendo com que o par de fontes de alimentação redundantes se torne cerca de 4150 vezes mais fiável que uma única fonte de alimentação.

redundância

exercício 1.2

$$\mathrm{MTTF}_{fonte} = \frac{\frac{\mathrm{MTTF}_{fonte}}{2}}{\frac{\mathrm{MTTR}_{fonte}}{\mathrm{MTTF}_{fonte}}} = \frac{\frac{\mathrm{MTTF}^2_{fonte}}{2}}{\mathrm{MTTR}_{fonte}} = \frac{\mathrm{MTTF}^2_{fonte}}{2\mathrm{MTTR}_{fonte}} \tag{1.5}$$

$$\mathrm{MTTF}_{fonte} = \frac{\mathrm{MTTF}^2_{fonte}}{2\mathrm{MTTR}_{fonte}} = \frac{200000^2}{2\times 24} = 830000000 = 830\times 10^6 \tag{1.6}$$

Na teoria de fiabilidade usa-se a distribuição exponencial para modelar as falhas dos módulos de um sistema computacional. Recordando Métodos Probabilísticos para Engenharia Informática (a2s1), uma **distribuição exponencial** usa-se para modelar o comportamento de uma variável aleatória X, tendo em consideração que X, sendo $X \geq 0$, representa o intervalo de tempo entre dois eventos quaisquer, e que a ocorrência de um evento não afeta a probabilidade da ocorrência de um segundo. Também se considera que a frequência de ocorrência dos acontecimentos é constante e que a ocorrência de eventos sucessivos não podem coincidir com o tempo. Sobre estas considerações a função de densidade probabilística $p(t)$ é exprimida por (1.7), onde λ é a frequência de ocorrências, isto é, o número de eventos por unidade de tempo.

distribuição exponencial

$$p(t) = \lambda e^{-\lambda t} \tag{1.7}$$

Aplicando assim a distribuição exponencial para o nosso caso em estudo, temos que a variável aleatória X representa o intervalo de tempo entre falhas de sistema e a frequência λ exprime a frequência de falhas ou, quando convertida para falhas em milhares de milhões de horas de operação, em FIT. O seu recíproco é igual à média da variável aleatória X, denominando-se de tempo médio para falha (MTTF). Num sistema que consista do conjunto de n módulos cujo comportamento de falha possa ser modelado por esta distribuição e se for possível assumir que tais falhas são independentes entre si, a probabilidade de não ocorrer quaisquer falhas no intervalo $[0,\ t_0[$ é exprimida por (1.8), o que significa que sob estas condições a frequência de falhas do sistema, como um todo (*overall*), é a soma das várias frequências dos módulos constituintes.

$$\begin{aligned} P(X_1 \geq t_0, X_2 \geq t_0, \cdots, X_n \geq t_0,) &= \prod_{i=1}^{n} P(X_i \geq t_0) \\ &= \prod_{i=1}^{n} e^{-\lambda_i t_0} \\ &= e^{-(\lambda_1+\lambda_2+\cdots+\lambda_n)t_0} \end{aligned} \tag{1.8}$$

Em termos de tempo médio para falha (MTTF) este do sistema será equivalente a (1.9).

$$\mathrm{MTTF}_{sistema} = \frac{1}{\sum_{i=1}^{n} \frac{1}{\mathrm{MTTF}_n}} \tag{1.9}$$

Quando um sistema computacional falha, o módulo responsável por tal quebra deve ser localizado e substituído por um novo e funcional. Enquanto este processo acontece o sistema comuta entre dois estados: o estado de **uptime**, onde este se encontra ativo, e o estado de **downtime**, onde este se encontra inativo e/ou em manutenção/substituição.

uptime

downtime

Este período de tempo pode ser modelado como uma variável aleatória X igual à soma de duas variáveis aleatórias de distribuições exponenciais independentes, entre as quais $X_{sistema}$ que representa o intervalo de tempo entre falhas de sistema e $X_{reparação}$ que representa o seu tempo de reparação.

Sejam assim $p(t)_{sistema}$ e $p(t)_{reparação}$ duas funções de densidade probabilística das variáveis aleatórias $X_{sistema}$ e $X_{reparação}$, respetivamente, então a distribuição $p(t)$ de X (a sua soma) será determinada por (1.10) e o seu valor esperado ($E[X]$) por (1.11) [6].

$$
\begin{aligned}
p(t) &= \int_0^{\infty} p(u)_{sistema}\, p(t-u)_{reparação}\, du \\
&= \int_0^{\infty} \lambda_{sistema} e^{-\lambda_{sistema} u} \lambda_{reparação} e^{-\lambda_{reparação}(t-u)}\, du \\
&= \frac{\lambda_{sistema}\lambda_{reparação}}{\lambda_{sistema}-\lambda_{reparação}} \cdot \left(e^{-\lambda_{sistema} t} - e^{-\lambda_{reparação} t}\right), \text{ se e só se } \lambda_{sistema} \neq \lambda_{reparação} \\
&= \lambda^2 t e^{-\lambda t}, \qquad \text{se e só se } \lambda = \lambda_{sistema} = \lambda_{reparação}
\end{aligned}
\tag{1.10}
$$

$$
\begin{aligned}
E[X] &= \frac{\lambda_{sistema}\lambda_{reparação}}{\lambda_{sistema}-\lambda_{reparação}} \cdot \left(e^{-\lambda_{sistema} t} - e^{-\lambda_{reparação} t}\right) \\
&= \frac{1}{\lambda_{sistema}} + \frac{1}{\lambda_{reparação}}, \text{ se e só se } \lambda_{sistema} \neq \lambda_{reparação} \\
E[X] &= \lambda^2 t e^{-\lambda t} = \frac{2}{\lambda}, \text{ se e só se } \lambda = \lambda_{sistema} = \lambda_{reparação}
\end{aligned}
\tag{1.11}
$$

Verificando o recíproco da variável aleatória $X_{reparação}$ (MTTR) e o recíproco de X (MTBF) obtemos que, tanto no contexto de um sistema, como de um módulo, MTBF = = MTTF + MTTR.

Medição de desempenho e Lei de Amdahl

De volta à discussão do que consideramos ser o "melhor" numa comparação entre um sistema X e um sistema Y, podemos colocar estes à prova em aplicações reais de execução, que variem tanto em complexidade e tamanho. Assim nasceu o **SPEC** (acrónimo para *Standard Performance Evaluation Corporation*), sendo uma corporação sem fins lucrativos que estabeleceu um conjunto de **benchmarks** (sequências de programas de teste) relevantes e especiais para as mais recentes gerações de processadores de alto-desempenho.

SPEC

benchmarks

Os ganhos de desempenho que podem ser gerados pela melhoria de alguns artefactos nos sistemas a testes dos SPEC podem ser estimados por aquilo a que damos o nome de **Lei de Amdahl**. Nesta lei enuncia-se que o *speedup* a ser ganho por adotar novos e mais rápidos métodos de execução é limitado pela fração de tempo que o modo mais rápido usa. Esta ideia é formalizada na expressão algébrica de (1.12).

lei de Amdahl

⊙ **Gene Amdahl**

$$
\begin{aligned}
\text{speedup}_{overall} &= \frac{\text{tempo de execução para uma tarefa sem melhorias}}{\text{tempo de execução para uma tarefa com melhorias}} \\
&= \frac{1}{\left(1 - \text{fração}_{melhoria}\right) + \dfrac{\text{fração}_{melhoria}}{\text{speedup}_{melhoria}}}
\end{aligned}
\tag{1.12}
$$

Em (1.12) $frac_{melhoria}$ pretende designar a fração de tempo que o sistema computacional original poderá converter para tomar partido de um modo mais rápido de execução e $speedup_{melhoria}$ é a rapidez a ser ganha localmente pela adoção de um modo mais rápido de execução.

Por exemplo, num Exercício 1.3 teríamos que um processador usado num servidor web está para ser alterado, para atingir um *speedup* maior nas suas operações. O novo processador é dez vezes mais rápido que o original. Assumindo que atualmente o processador está ocupado em 40% das vezes e à espera de instruções em 60% do seu tempo, qual será o *speedup* total do sistema se ocorrer substituição? Para tal apenas temos de designar os 40% como fração de melhoria e o *speedup* como sendo 10. Assim sendo temos que, substituindo os valores em (1.12) obtemos uma razão de 1.56. Como podemos verificar, a Lei de Amdahl é uma lei de retornos diminuídos, porque estando à espera de um grande aumento de desempenho (dado o *speedup* de 10), obtivemos um ganho de apenas 1.56.

Quando falamos de tempo de execução temos de ter cuidado sobre o local onde a execução é realizada. Se estivermos a referir um processador, então temos de estabelecer outras variáveis que poderão interferir nas nossas resoluções. Por exemplo, o tempo de execução em CPU ao executar um programa pode ser exprimido pelo produto dos ciclos de relógio do CPU para executar com o tempo de ciclo de relógio. Noutras palavras, podemos reescrever que o tempo de execução em CPU é o produto do número de instruções com o número de ciclos de relógio por instrução com, ainda, o tempo de ciclo de relógio.

Como podemos ver o tempo de execução para um CPU, de um programa, está dependente de três aspetos: do número de instruções para a execução de um programa, do número médio de ciclos de relógio por instrução e pelo tempo de ciclo de relógio. Mais ainda, o tempo de CPU é igualmente dependente destes três parâmetros, dado que uma dada mudança num dos três aspetos causará alterações idênticas nos outros.

Infelizmente, é difícil mudar um dos três componentes de forma isolada, sendo que as tecnologias implementadas são inteiramente inter-dependentes: o número de instruções está relacionado com a arquitetura de um computador e com a sua tecnologia; o número de ciclos por instrução é tanto dependente da arquitetura do sistema, como da sua organização; já o tempo de ciclo de relógio depende do hardware montado na máquina (da sua tecnologia) e da sua organização.

2. Paralelismo de Instruções

Em Arquitetura de Computadores I (a2s1) abordámos, pela primeira vez, a forma como todo o código processado em linguagens de mais alto nível é lido e trabalhado ao nível do processador. Para tal começámos por descrever os códigos como sendo conjuntos de instruções bem definidas numa linguagem mais próxima da máquina - o Assembly. Referimos assim a arquitetura de processadores MIPS dada a sua regularidade no conjunto de instruções - todas as instruções têm o mesmo tamanho. Já no final dessa disciplina pudemos ter o primeiro contacto com as fases de descodificação de uma instrução e cálculo registo por registo, elaborado no processador - referimos assim os datapaths e abordámos duas alternativas de datapath (single e multi-cycle). E assim chegámos a uma nova fase à qual adicionámos a ideia de **paralelismo** - com a introdução de um pipeline, pudemos, dadas as fases já desenvolvidas na organização single-cycle, criar uma sequência de instruções que fosse lida em simultâneo, como uma cadeia de montagem de um produto.

paralelismo

Conceito de pipelining

Se considerarmos uma fábrica automóvel e cada um precisar de vinte fases para a sua construção (criação da estrutura-base do veículo, aplicação do parachoques dianteiro, aplicação do parachoques traseiro, montagem das portas, montagem das rodas, ...) e cada uma destas demorar uma hora a ser dada como realizada, se as unidades forem criadas de forma sequencial, então teríamos que 1 carro demoraria vinte horas para ser concluído, isto

é, por dia, só um carro é que seria construído. No entanto, se introduzirmos a ideia de paralelismo na montagem da viatura, temos que enquanto um carro está a ser montado numa fase x, outro estará a ser montado numa fase x-y. Por exemplo, se a aplicação do parachoques traseiro for efetuada após a montagem do parachoques dianteiro, enquanto que um carro está a receber o parachoques traseiro, o carro que está a ser montado imediatamente a seguir estará a receber o parachoques dianteiro.

cadeia de montagem

Não é por acaso que a ideia de uma **cadeia de montagem** automóvel é dada, vulgarmente, como exemplo e introdução ao conceito de pipelining. Na verdade, a ideia de pipelining surgiu com a criação da cadeia de montagem, por Henry Ford, que no contexto da indústria automóvel gerou a operação laboral por pipelining na empresa que criou - a Ford Motor Company, hoje, Ford -, para a criação do automóvel Ford T, em fins de 1913.

⊙ **Henry Ford**

Considerando novamente o exemplo anterior, em que cada fase de montagem demora uma hora, com um pipelining continuamos a ter um carro a ser feito a cada vinte horas, no entanto, deixamos de ter apenas uma viatura a ser produzida por dia, para ter quatro.

De volta agora ao contexto de Arquitetura de Computadores, podemos relacionar o exemplo da indústria automóvel se por montagem de carro referirmos programa e por tarefa designarmos instrução. Note-se que o número de tarefas (aqui número de ciclos), no nosso caso, será 5, isto porque na arquitetura MIPS, como nos devemos recordar, existem as fases de **instruction fetch** (receção da instrução, onde se seleciona a sequência de 4 bytes de instrução da memória de instrução), **instruction decode** (descodificação da instrução, onde se designa cada campo para cada tipo de instruções do ISA), **execute** (fase de execução, onde dados dois operandos se processa, numa unidade aritmética e lógica, vulgarmente designada de ALU, uma operação, como tal, aritmética ou lógica), **memory access** (fase de acesso/escrita na memória) e **write back** (fase de reescrita em registo, onde se regista, caso necessário, um valor de volta num registo endereçado do processador).

instruction fetch
instruction decode
execute
memory access
write back

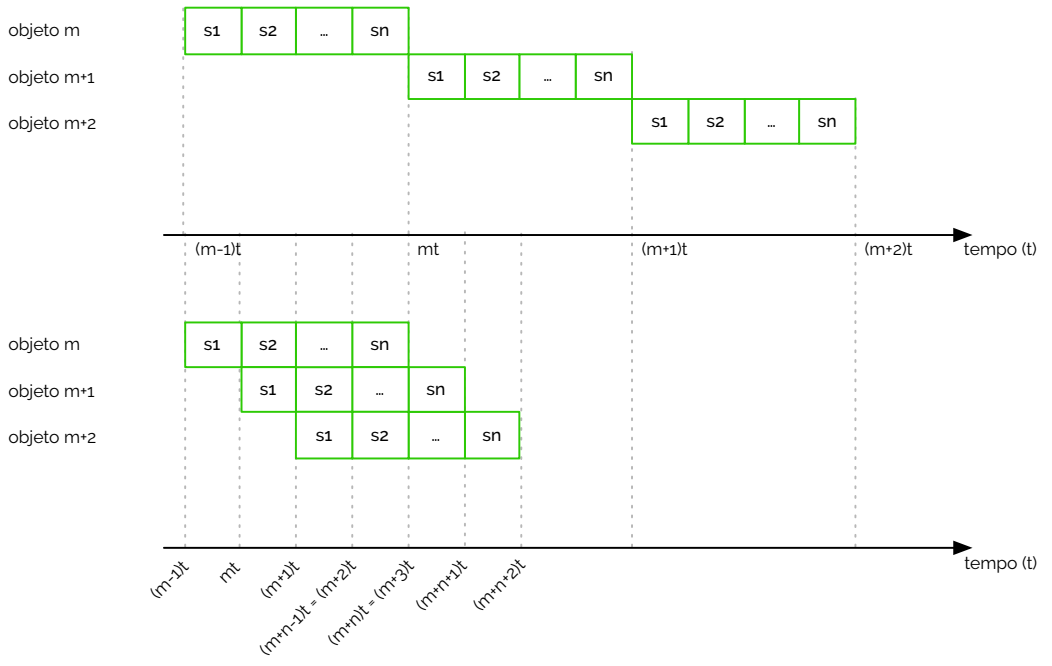


figura 2.1

Comparemos então ambas as versões single-cycle simples e *pipelined* na Figura 2.1, onde para criar um objeto m é necessário efetuar uma sequência de n subtarefas (abreviadas de $s1$, $s2$, ..., sn para identificar a subtarefa 1, subtarefa 2, ..., subtarefa n, respetivamente). O **throughput** do primeiro caso é passível de ser calculado tomando o quociente de uma unidade do objeto m realizada por t tempo, isto é, por $1/t$. Se pretendermos saber quanto demora a execução de M objetos m, e cada objeto m demora t tempo, então temos Mt tempo. Por outro lado, no caso do pipeline temos que o throughput é de uma unidade

throughput

do objeto m realizada por t tempo, isto é, por $1/t$, na mesma. No entanto, o tempo da execução de M objetos m é $(M + n - 1)t$.

Já que sabemos como calcular o *speedup*, qual será este para a execução numa implementação em pipeline de n andares *versus* uma execução sem ser *pipelined*? Para tal fazemos um quociente do tempo de execução da versão não *pipelined* sobre o tempo de execução da implementação do pipeline de n andares. Assim sendo ficamos com $Mt_{non\text{-}pipeline}/[(n - 1 + M)t_{pipeline}]$, isto é, aproximadamente $n(t_{non\text{-}pipeline}/t_{pipeline})$.Tipicamente, por valores teóricos, o rácio $(t_{non\text{-}pipeline}/t_{pipeline})$ tende para 1, fazendo com que o *speedup* se verifique como o número de andares do pipeline.

Uma pequena visão na arquitetura RISC - MIPS64

O núcleo de uma arquitetura **RISC** (acrónimo de *Reduced Instruction Set Computer*), no nosso caso em particular, MIPS64, é sempre caraterizado pelo nível de simplificação do conjunto de instruções. Para além de terem todas o mesmo tamanho e de se classificarem em 3 tipos de instruções, as únicas instruções que de facto manipulam os registos do processador são as aritméticas e lógicas. Mais, em comparação a arquiteturas **CISC** (acrónimo de *Complex Instruction Set Computer*), como é o caso da x86 da Intel, apenas existem duas instruções que manipulam operações de transferência (leitura ou escrita) de dados da memória - falamos da instrução de *load* e de *store*. Este tipo de simplificações também tiveram o seu impacto, de forma mais simples, sobre a forma como a implementação do pipeline é realizada.

RISC

CISC

Como já foi referido, neste documento ir-nos-emos basear numa arquitetura **MIPS64**, contrariamente à MIPS32 que estudámos em Arquitetura de Computadores I (a2s1) e Arquitetura de Computadores II (a2s2). Nesta versão todas as instruções continuam a ter 32 bits (4 bytes) em tamanho, mas passamos a admitir não só bytes de 8-bits, *half words* de 16-bits e *words* de 32-bits, como também temos *double words* de 64-bits. As instruções extendidas para 64-bits geralmente possuem uma notação diferente, iniciando sempre com a letra `D`, ou terminando, dependendo da mnemónica associada. Outra diferença está no banco de registos que passa a conter 32 registos de 64-bits cada, onde o registo número 0 (`r0`) é só-de-leitura e contém sempre o valor 0. Na Figura 2.2 temos os vários tipos de instruções como já conhecemos do MIPS32 [7].

MIPS64

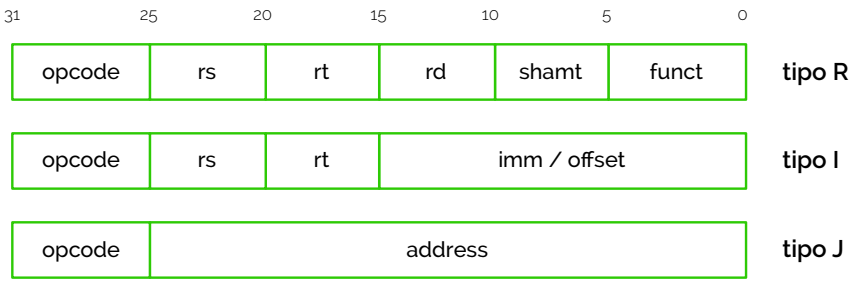


figura 2.2

Anteriormente já designámos os vários momentos de implementação de um datapath. Designámos assim primeiramente uma fase de **instruction fetch** (IF) no qual o conteúdo do registo `program counter` (PC) é usado como endereço para a próxima instrução que está alojada na **memória de instrução**. Este valor que é recolhido desta memória é guardado num segundo registo, denominado de `instruction register` (IR). Também nesta fase é atualizado o valor do `program counter` (PC) com uma soma por 4, dado que se pretende avançar 4 bytes.

instruction fetch

memória de instrução

Após o instruction fetch ter sido concluído entramos numa fase em que este é registado para o **instruction decode** (ID), na qual a instrução é descodificada para que possam ser executadas as devidas operações. Aqui, como ainda não se sabe qual é o tipo da instrução que está em causa, as instruções são todas vistas como igual, isolando linhas de comunicação (barramentos) para os conjuntos identificadores de imediato, registo `rs`, `rt`,

instruction decode

`rd`, código de operação, deslocamentos, códigos de função ou mesmo endereços para os saltos incondicionais. É nesta fase também que se calcula se o conteúdo de dois registos passados como operandos são iguais ou não, possibilitando, caso se confirme, a instrução branch com efeito imediato neste ciclo, permitindo o alojamento do endereço calculado pelo deslocamento indicado na instrução, em relação ao valor do registo PC atual, neste mesmo registo.

execute
ALU

Tendo a fase de descodificação terminada entramos na fase de **execute** (EX), onde é a **ALU** (unidade aritmética e lógica) que toma o papel principal, sendo este o componente que irá operar sobre os dois operandos fornecidos pela fase anterior, com uma relação aritmética ou lógica. Aqui a ALU poderá efetuar um cálculo entre operandos provenientes de registos (os dois), entre um proveniente de um registo e uma constante (vulgarmente denominado de **imediato**) ou entre duas referências para calcular uma terceira referência de memória.

imediato

memory access

Quando a fase de execução termina, caso seja necessário ou não, entramos numa fase de **memory access** (MEM) - acesso à memória. Nesta fase, para o caso de uma instrução como a *load* há que ter um endereço, muito possivelmente calculado pela ALU na fase anterior, e com isso, conseguimos ler o valor de uma posição de memória. Outra instrução que poderá desfrutar de um acesso à memória será a de *store*, dado que esta precisará de lhe aceder para escrever um conteúdo.

write back

Por último, para um caso como o *load* há que, dada a leitura de um valor da memória, guardá-lo num registo do nosso banco de registos. Entramos assim na chamada fase de **write back** (WB). Aqui voltamos então para o início do nosso datapath, ligando a saída da memória ao banco de registos. Outra possibilidade de usar este ciclo seria numa operação aritmética ou lógica, onde o valor que teria sido calculado pela ALU na fase EX precisa agora de ser preservado no registo previamente indicado em código, no registo `rd`.

exercício 2.1

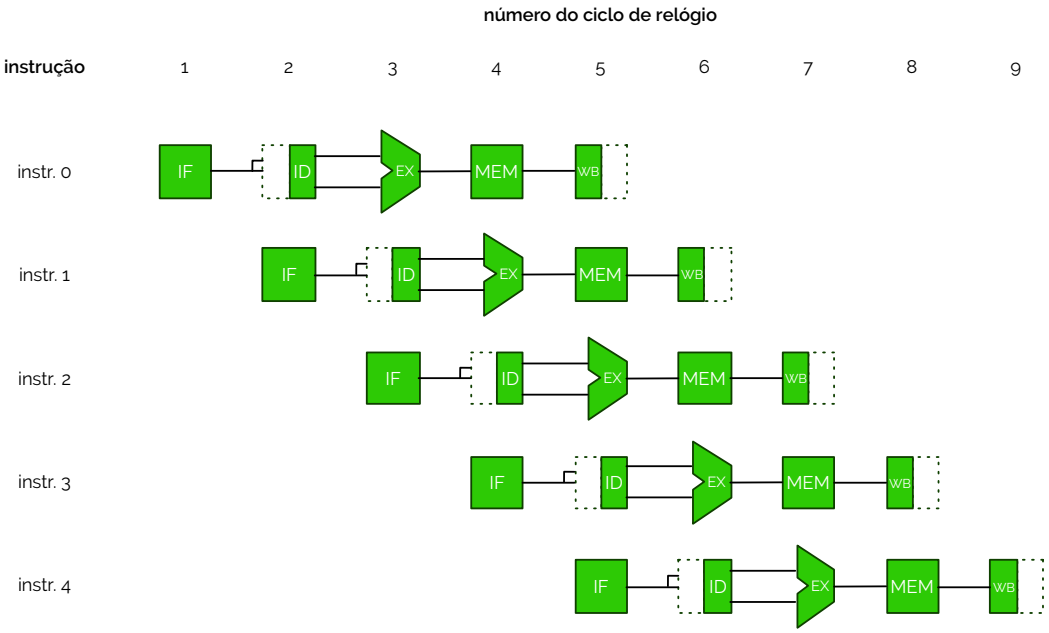
Como podemos ver, nestas cinco fases de implementação as instruções de branch apenas necessitam de 2 ciclos para serem totalmente efetuadas, contrariamente aos 3 ciclos que estudámos em Arquitetura de Computadores I (a2s1). Num Exercício 2.1 considerando isso e que as instruções de *store* consomem 4 ciclos e as de *load* 5 ciclos, assumindo que um dado *benchmark* possui uma frequência de branches de 12% e uma frequência de *loads* de 10%, qual é o CPI médio para correr esta aplicação na versão não-pipeline que acabámos de abordar? Ora, sendo o CPI geral calculado pelo somatório da frequência de instrução i pelo CPI de uma instrução i, temos que é igual a $0.12 \times 2 + 0.10 \times 4 + 0.78 \times 5 = 4.54$. Portanto, como podemos ver já não podemos ter um speedup de 5, como anteriormente teríamos previsto, mas antes de 4.54 [7].

Pipeline de 5 andares de processador RISC

Edsger Dijkstra

A execução de uma implementação não-pipeline pode ser convertida num pipeline fazendo um novo *fetch* a cada ciclo novo de execução. Esta execução poderá ser vista na representação da Figura 2.3. Se prestarmos muita atenção à Figura 2.3 podemos ver que tanto a fase de ID como de WB estão apresentadas em metade do período total do ciclo de relógio. Isto acontece porque, para que estas implementações funcionem, devemos garantir que duas operações nunca coincidam sobre o mesmo recurso de dados - vide algoritmo dos banqueiros de Dijkstra em Sistemas de Operação (a3s1). No entanto, como é que podemos garantir que não estamos a usar o mesmo recurso mais que uma vez, se no ciclo de relógio número 5 estamos a escrever um valor no banco de registos (WB da instrução 0) e a ler do banco de registos (ID da instrução 3)? Mais, no mesmo ciclo, estamos a aceder à memória na fase MEM da instrução 1 e na fase IF da instrução 4... como é que podemos então garantir? Concentremo-nos na primeira questão: no pipeline, cada fase é executada num ciclo de relógio, o qual terá um período, isto é, uma transição descendente e uma transição ascendente. Na primeira metade dos ciclos de relógio realiza-se uma escrita, enquanto que na segunda metade realiza-se uma leitura. Por outras palavras, não acontecem conflitos no banco de registos porque durante o ciclo 5, na primeira metade, o WB escreve o valor no registo e, na segunda metade, o ID lê um valor do banco de registos. Em relação à segunda

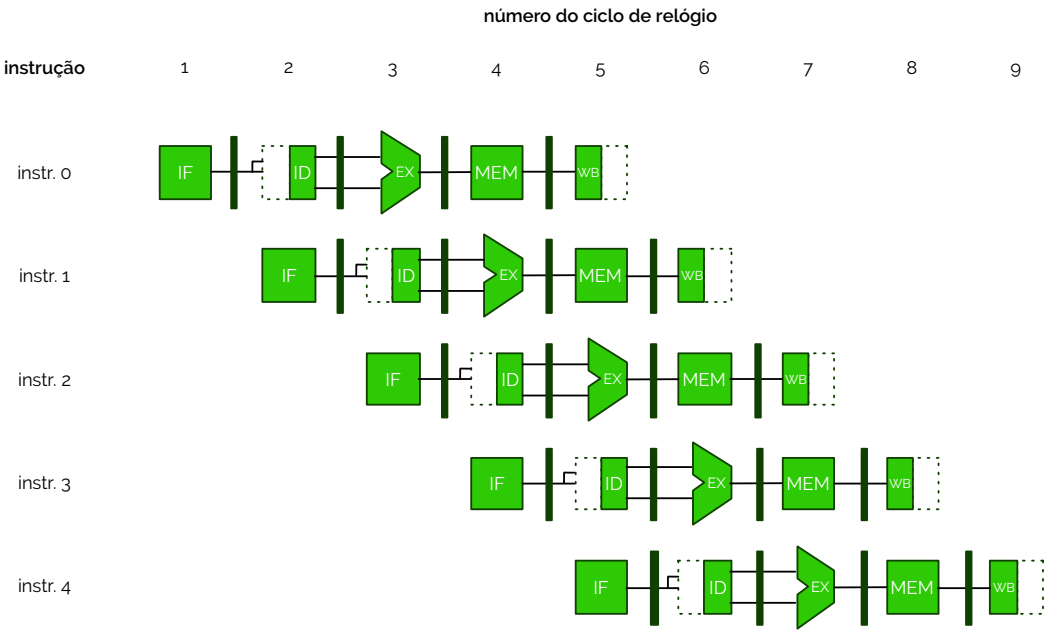
figura 2.3



questão, não acontecem conflitos entre as fases MEM e IF porque embora ambas acedam à memória, esta é diferente em cada um dos casos. Esta é a razão pela qual se distinguem a memória de instruções (usada pela fase de instruction fetch) e a memória de dados (usada pela fase de acesso à memória). Note-se que, no entanto, pelo facto das instruções de branch serem calculadas na fase de instruction decode (segunda fase), caso estas exijam um salto, o valor do registo `program counter` terá de ser alterado neste mesmo ciclo, mas como já vimos, neste ciclo apenas podemos efetuar uma leitura do banco de registos, dado que este mesmo recurso, numa situação de paralelismo, está a ser usado para escrita pela fase write back da instrução que iniciou a execução 4 ciclos antes. Iremos voltar a este problema mais à frente, tentando-o resolver.

Para podermos implementar o conceito de pipeline de forma ainda mais rigorosa, devemos ser capazes de, a olhar para o datapath (mesmo que sumário), identificar os pontos de transição entre as várias fases de execução. Assim vejamos a Figura 2.4 onde se pretendem ilustrar os demais registos que se localizam nas transições das fases, pelo nome de fase1/fase2, isto é, IF/ID, ID/EX, EX/MEM e MEM/WB.

figura 2.4



Em suma, como já vimos, a implementação de pipeline aumenta o throughput de instruções, mas não reduz o tempo de execução de uma instrução individual. Pelo contrário, até aumenta ligeiramente o tempo de execução dado que acaba por possuir um **overhead** maior para controlar o pipeline. Reconsiderando a implementação sem pipeline, num Exercício 2.2, assumamos que temos uma com um ciclo de relógio de 1 ns e que executa um *benchmark* que possui uma frequência de branches de 20% e uma frequência de *stores* de 10%. Dado um overhead para controlo do pipeline, o ciclo de relógio para uma implementação deste tipo passa para 1.2 ns. Qual é o *speedup* ideal para a taxa de execução de instruções ganha por esta implementação? Para calcularmos tal valor necessitamos de verificar qual o tempo médio de execução de uma implementação não-pipeline: assim apenas temos de efetuar o produto do CPI geral com o tempo de ciclo de relógio, isto é, aplicado ao nosso caso, $(0.2 \times 2 + 0.1 \times 4 + 0.7 \times 5) \times 1.0 = 4.3$ ns. Sabendo o tempo de execução média para uma implementação sem pipeline, agora podemos calcular o *speedup* pela forma a que já estamos habituados, formando o quociente entre o tempo de execução média para uma versão não-pipeline com uma versão pipeline, isto é, $4.3/1.2 = = 3.6$. Como podemos ver, o *speedup* que inicialmente, numa versão teórica, equivalia ao número de andares desta nova implementação (5 andares), agora foi ainda mais reduzido, para 3.6.

overhead

Hazards da implementação pipeline

Situações como a que deixámos anteriormente por resolver, de conflitos de validade do conteúdo dos registos ou de acesso aos recursos, são casos que levantam bastantes problemas para a execução dos nossos programas numa implementação pipeline de um processador. Muitas vezes temos mesmo de parar, durante um número de ciclos de relógio, o nosso pipeline, fenómeno o qual damos o nome de **stall**.

stall

O pipeline até poderia funcionar sem quaisquer problemas se as instruções fossem todas independentes entre si. Como não é assim, por vezes acontecem **hazards**, isto é, situações que inibem que o pipeline execute a sua função de forma correta. Existem três tipos de hazards, entre os quais os estruturais, de dados e de controlo.

hazards

Os **hazards estruturais** são aqueles que provêm de conflitos de partilha de recursos, isto é, quando o hardware não está preparado para que uma linha de dados aceda a um determinado componente (como a memória). Uma forma de corrigir este problema é tornando o componente que se pretende aceder *pipelined*. Mas como esta correção é bastante dispendiosa não é realista pensar em tal caso como solução ao nosso problema.

hazards estruturais

Um exemplo de aplicação para o problema dos hazards estruturais é visível na Figura 2.5, onde tentamos aceder à memória em duas instruções em simultâneo (isto considerando que a memória de instrução e a memória de dados são a mesma com apenas um porto de leitura/escrita.

Para corrigir esta situação o que é usual fazer é criar um *stall* da última instrução até que a unidade requirida esteja disponível. Basicamente, numa analogia, se tivermos um tubo por onde passa continuamente água, para fazermos uma pausa no fluxo, momentânea, colocamos ar, gerando uma bolha no sistema. É por este motivo que se designa o *stall* como uma injeção de **bubbles** no pipeline.

bubbles

Vejamos assim a Figura 2.6 onde se sugere uma solução para o problema que vimos na Figura 2.5. Esta solução passa por fazer um *stall* antes do fetch da instrução o que equivale a executar uma instrução que nada faz, tomando o sentido da bolha. Esta situação está representada por uma instrução que foi inserida e que está a branco, dado que não faz efeito para o processador - apenas o atrasa em 1 ciclo de relógio.

Suponhamos assim, num Exercício 2.2, que as instruções de transferência de dados da memória constituem 40% de um conjunto de instruções para um dado *benchmark* e que o CPI ideal para o processador *pipelined*, ignorando os hazards estruturais, é de 1. Assumindo que o processador com hazards estruturais tem uma frequência de relógio de 1.05 vezes mais que o processador sem o hazard e não contando com outras perdas, que processador é que executa as instruções de forma mais rápida? Ora, para resolvermos este exercí-

figura 2.5

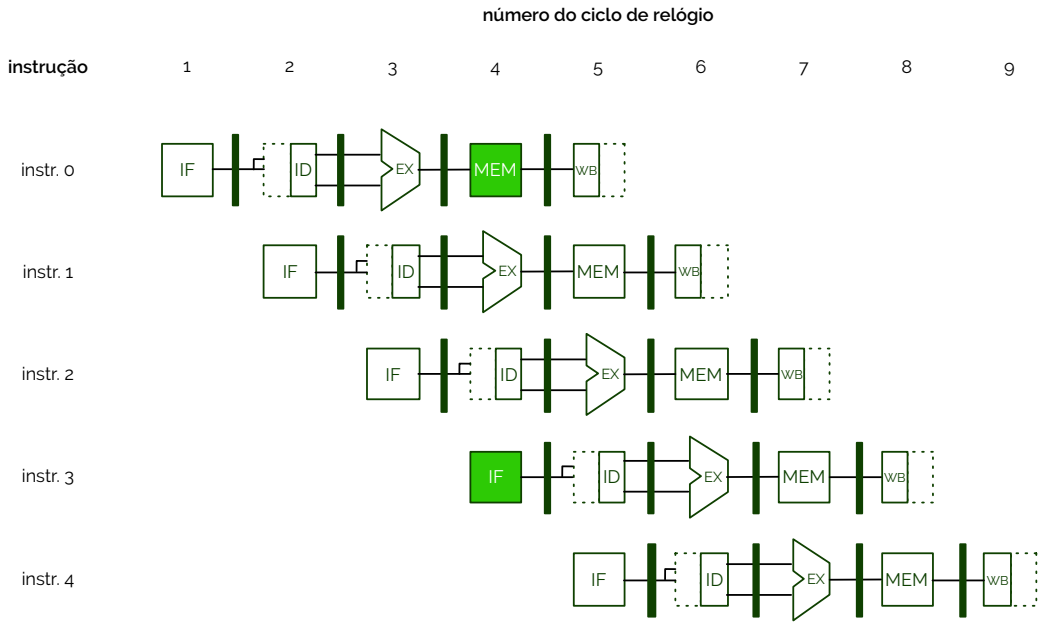
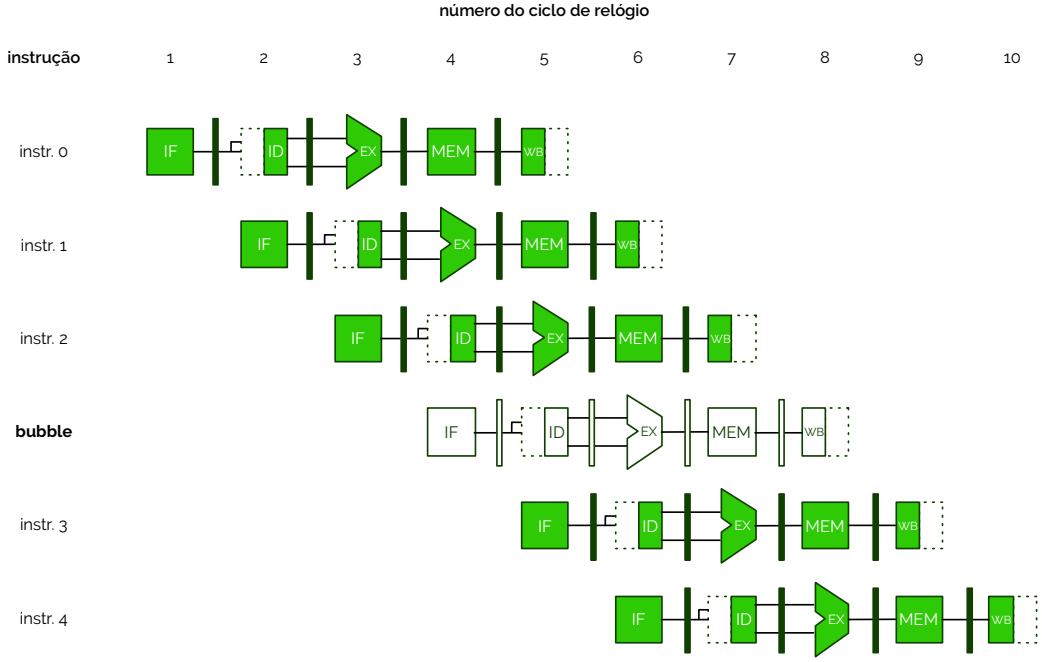


figura 2.6



cio, primeiro temos de calcular qual o tempo médio para execução das instruções sem hazard, valor o qual se pode obter através do produto do CPI total (sem hazarding) com o tempo de ciclo de relógio ideal, o que, no nosso caso, é equivalente ao tempo de ciclo de relógio ideal, dado que o primeiro é 1. Assim sendo temos de comparar este valor com o tempo médio de execução das instruções com hazarding, para percebermos que se efetuarmos o mesmo cálculo, considerando agora o hazarding ativo, podemos obter um valor de 1.3 vezes o tempo de ciclo de relógio ideal. Então, se todos os fatores forem iguais, um processador sem hazards estruturais terá sempre um CPI menor. Isto permite-se que seja feito porque abriga a necessidade de reduzir as unidades do processador, para que se possa tratar do problema de custo elevado em tornar todas as unidades em unidades com funcionalidades em pipeline, levando muitas vezes na duplicação destas. Se os hazards estruturais são raros, então podem não valer o custo evitá-los.

hazards de dados

Um segundo tipo de hazards que referimos são os **hazards de dados**, os quais ocorrem quando o pipeline altera a ordem das operações de leitura/escrita dos operandos, fazendo com que a sequência de execução seja diferente da que obteríamos numa execução

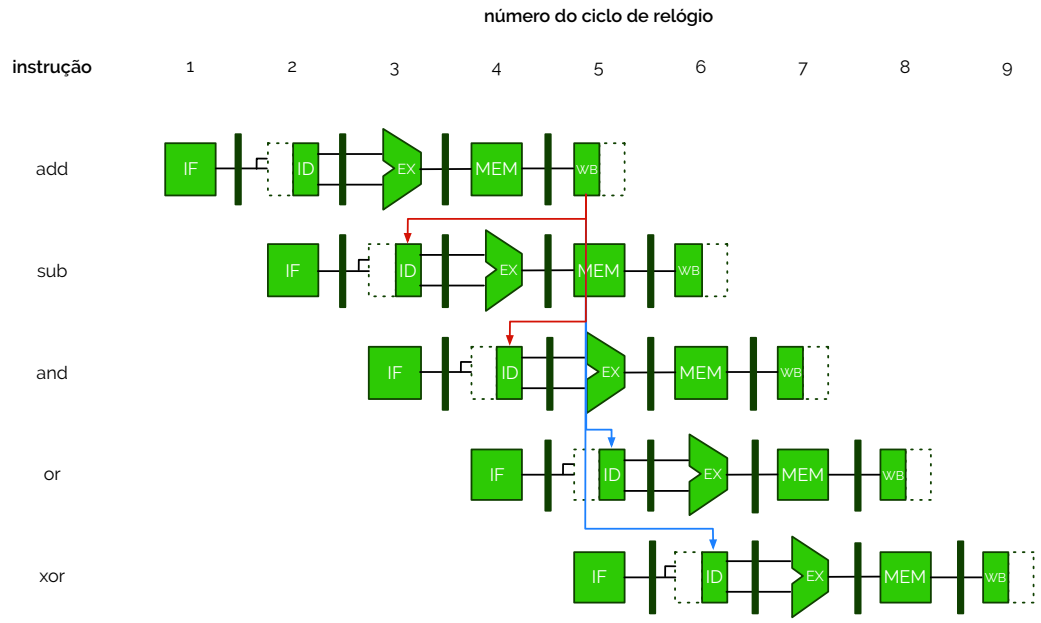
non-pipelined. Por exemplo, se executarmos uma operação aritmética como $a = b + c$ e, de seguida, $c = a + d$, estamos a tentar fazer uma segunda operação sobre um resultado (o valor de a), que no pipeline ainda não está com o valor correto. Vejamos assim o Código 2.1 onde se expõe um dos exemplos onde ocorre hazard de dados.

código 2.1

```
add     r1, r2, r3
sub     r4, r1, r5
and     r6, r1, r7
or      r8, r1, r9
xor     r10, r1, r11
```

Como podemos ver pelo Código 2.1 todas as instruções que sucedem a `add r1, r2, r3` usam o resultado calculado em `r1`, que, dado o pipeline, não está disponível e com valor correto logo na segunda instrução onde é usada (instrução `sub`). Isto acontece porque, dada a "cadeia de montagem", os valores para o registo `r1`, que serão calculados no final da fase de execução (fase EXE) ainda não se encontram disponíveis na fase atual - a fase atual será a de instruction decode (ID), para a instrução que perfaz o valor de `r1`. Num ciclo seguinte, a primeira instrução (instrução `add`) estará já em fase de execução, pelo que o resultado que pretendemos é criado neste ciclo. No entanto, este resultado ainda não está "disponível", isto porque ele necessita de ser escrito na fase de WB (fase de *write back*), a qual ocorrerá dois ciclos à frente. Mais, a própria instrução de `and` também é afetada pelo mesmo hazard de dados: a escrita em `r1` não se completa até à primeira metade do quinto ciclo de relógio. Por conseguinte, como o registo é lido na segunda metade do ciclo 4, este receberá um resultado errado. Apesar destes cenários, as instruções de `or` e `xor` executam sem problemas. Como podemos ver na Figura 2.7, com os traços a vermelho e azul pretendem-se transportar os vários valores, à medida que são necessários, entre as várias fases do pipeline, após a instrução de `add`, onde os vermelhos mostram os redireccionamentos que não são possíveis de serem realizados e os azuis o que já são, isto é, antes e depois da criação do valor correto, pela primeira instrução.

figura 2.7

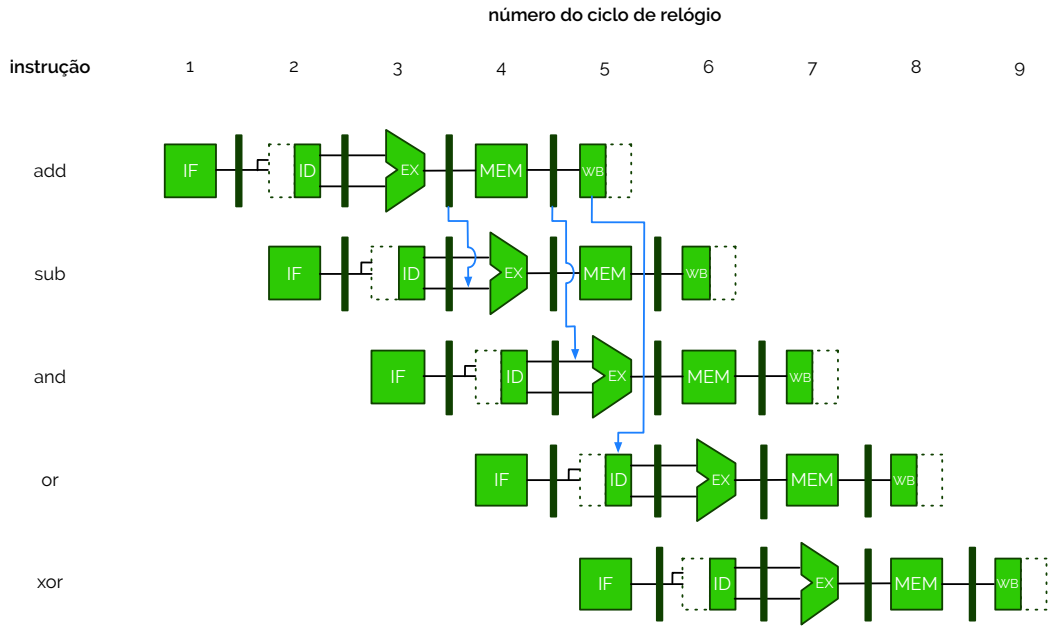


Uma forma de resolver este problema, do data hazarding, é através das técnicas de **forwarding**, isto é, reencaminhando os valores que estão presentes nos registos à medida que estes são produzidos e são válidos na altura da sua utilização. A ideia, olhando para o nosso exemplo, é que a instrução de `sub` e a de `and` não precisam dos valores calculados pela instrução de `add` antes que esta acabe a sua execução, mas também não precisam de esperar que o valor seja reescrito num registo, dado que este já existe no fim da fase de execução da primeira instrução. Dado isto, o *forwarding* (ou reencaminhamento, em português) trata-se de uma operação que permite que os valores do registo EX/MEM e

forwarding

MEM/WB, do pipeline, sejam de facto guardados e reencaminhados para as entradas da ALU, dependendo de toda uma lógica de controlo adjacente às mesmas. Tendo em conta esta técnica, na Figura 2.8, temos um exemplo de como é que as mesmas operações da Figura 2.7 poderiam ser executadas sem risco de paragens no pipeline.

figura 2.8



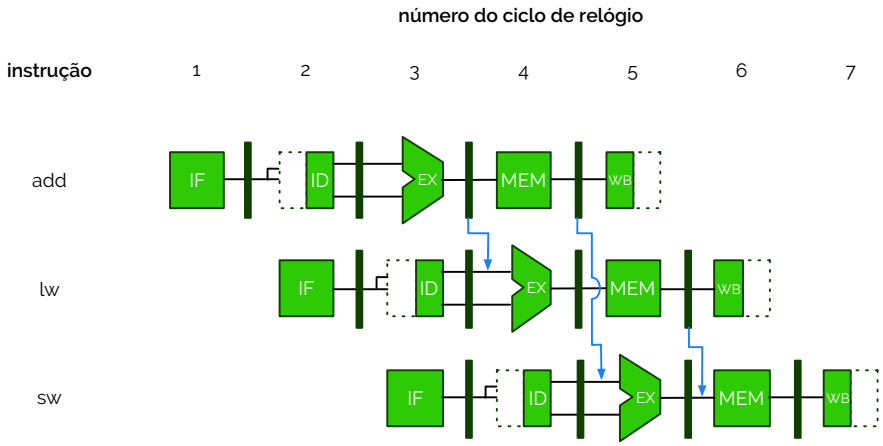
Consideremos agora outro exemplo, onde tentamos executar uma operação de transferência de dados para a memória - uma instrução de `store` - que podemos ver no Código 2.2.

código 2.2

```
add     r1, r2, r3
lw      r4, 0(r1)
sw      r4, 12(r1)
```

De forma a poder prevenir uma paragem nesta sequência de instruções, é necessário reencaminhar os valores da saída da unidade aritmética e lógica e da memória de dados para as entradas da ALU e da memória de dados. Assim sendo, na Figura 2.9, temos as técnicas de *forwarding* aplicadas ao Código 2.2.

figura 2.9



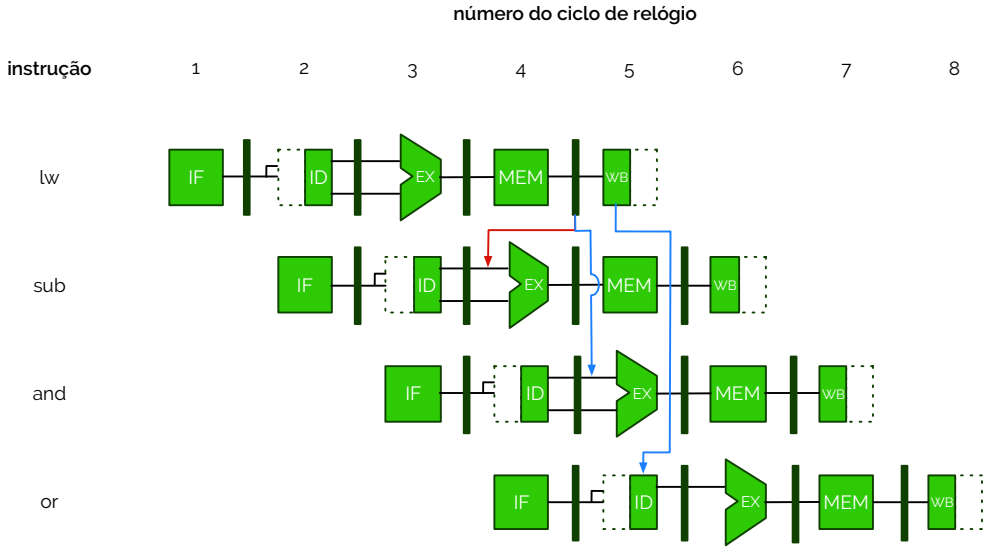
No entanto, é de notar que nem todas as situações de hazarding de dados podem ser resolvidas com as técnicas de *forwarding*. Situações onde isto acontece poderão provir de instruções cujo tempo necessário para serem concluídas são grandes, como a instrução de *load*. Consideremos assim o Código 2.3, onde efetuamos uma operação de transferência de memória, seguida de instruções aritméticas e lógicas sobre os mesmos registos.

código 2.3

```
lw      r1, 0(r2)
sub     r4, r1, r5
and     r6, r1, r7
or      r8, r1, r9
```

A instrução `sub`, do Código 2.3, usa o registo `r1` cujo valor é obtido na instrução `lw`. Para que este valor se possa considerar correto, temos, no mínimo, de esperar quatro ciclos de relógio, de forma a que este seja obtido aquando da sua captura na fase MEM. Logo por esta razão, o hazarding de dados não poderá ser totalmente tratado. Esta execução encontra-se sumariamente retratada na Figura 2.10.

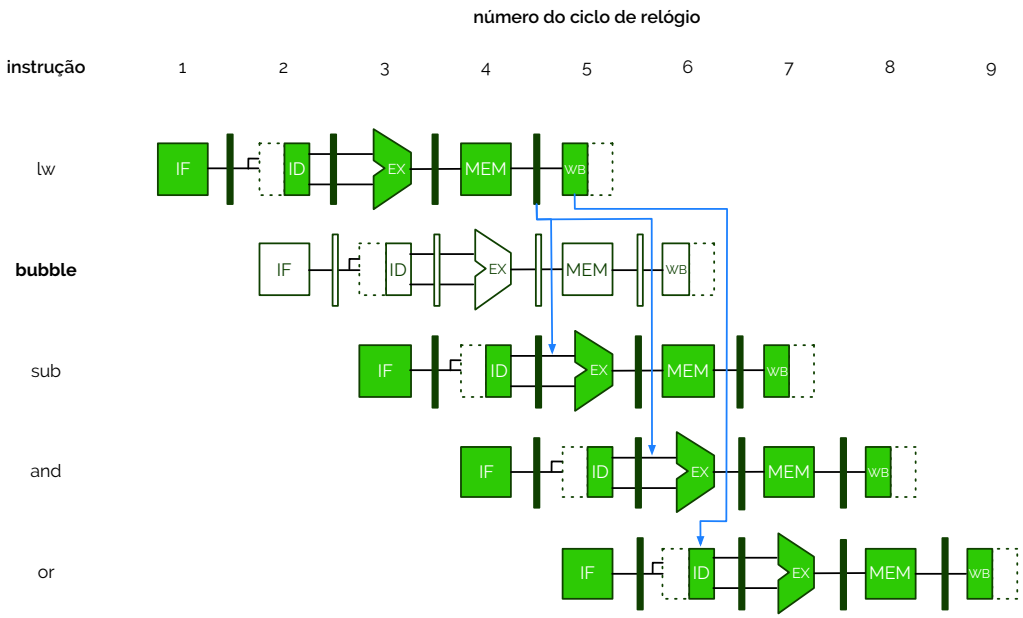
figura 2.10



De forma a tentar corrigir o problema explícito na Figura 2.10 temos que conseguir criar uma forma de provocar um *stall* antes da fase de execução da segunda instrução detalhada no código. De forma mais sumária, se estamos a tentar "adivinhar" o valor de um registo que estaria disponível um ciclo de relógio à frente, então precisamos de um ciclo de *stall*. Esta gestão de número de paragens do nosso pipeline deverá ser feita por uma segunda unidade de controlo, denominada de **unidade de interlocking** (ou unidade de hazarding). O resultante de tal aplicação de unidade seria algo como o visível na Figura 2.11.

unidade de interlocking

figura 2.11



O último tipo de hazards são os de **controlo**. Quando uma instrução de branch é executada esta pode, ou não, alterar o valor do registo `program counter`, modificando assim a sequência de execução de instruções. Quando o valor do registo é alterado por causa de um branch dizemos que o branch é **taken**. Por outro lado, quando o valor do registo `program counter` não é alterado numa instrução de branch dizemos que o branch é **untaken**. Estas alterações, se tiverem de ser feitas tendo em conta uma instrução de branch, serão tomadas no fim da fase de *instruction decode*, de forma a que no fim desta o valor do `program counter` já tenha sido atualizado. Por se tomar a decisão de saltar, ou não, logo na fase de *instruction decode*, pode dar-se o caso dos registos para a condição de salto ainda não terem o valor certo na sequência de instrução em causa, dizendo, que podem ocorrer hazards de controlo.

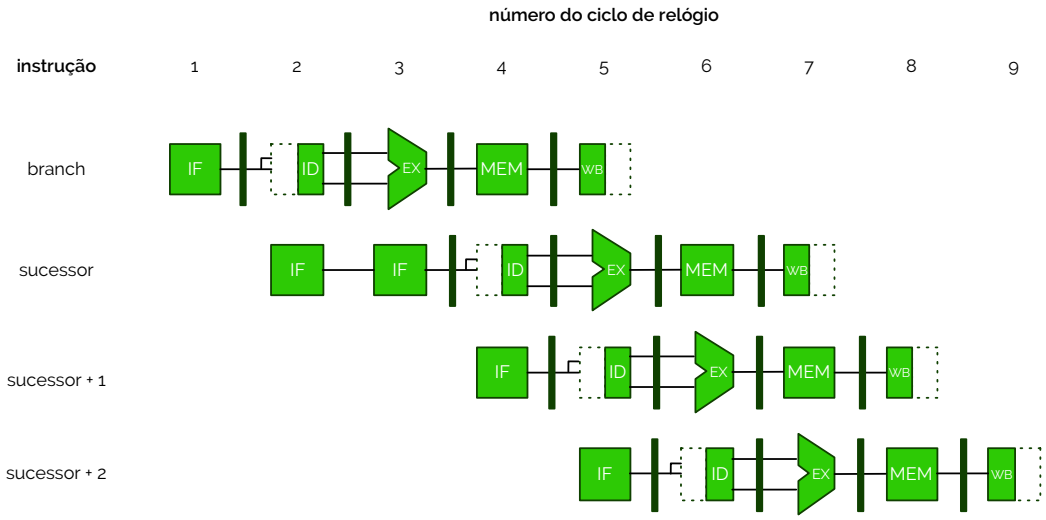
controlo

taken

untaken

Consideremos assim um caso de uma instrução de branch *b*. A instrução que será executada a seguir dependerá do acontecimento de um salto. Sendo que o salto só é verificado na segunda fase do pipeline (fase de *instruction decode*), acontece que o nosso pipeline terá, de alguma forma, de parar a execução em pelo menos um ciclo, de forma a que possa aguardar pela geração de uma condição válida para saber se a sequência de instruções deve prosseguir de forma normal ou deve contar com um salto primeiro, a partir do branch. Uma forma fácil de resolver esta situação é através da repetição de um ciclo de *instruction fetch*, isto é, executando um *stall* por repetição da leitura da instrução da memória de instrução. Assim sendo, conseguimos o atraso necessário para que, após as duas leituras (iguais) da memória de instrução, saibamos se devemos prosseguir ou não na nossa sequência de instruções. No entanto, esta não é uma solução, de longe, ótima, dado que estamos a contar com um atraso de um ciclo na nossa execução. Tal solução está representada na Figura 2.12.

figura 2.12



Uma forma mais rápida de tratar com os *branches* está no tomar partido que o branch é *untaken*, num esquema de **predicted-untaken**. Neste caso apenas temos de deixar que o *hardware* trate todas as instruções de branch como *untaken*, isto é, como sendo falsas, não alterando o valor do registo `program counter`. Sendo que, com esta solução, vemos todas as condições como falsas (sem salto) o mais natural aqui será prosseguir com a sequência de instruções, retirando um atraso desnecessário caso prossiga a execução (Figura 2.13) ou adicionando uma não-operação (`nop`) caso o salto seja verdadeiro (Figura 2.14). Isto acontece porque a previsão de que todos os saltos são falsos não é correta no ponto de vista de um programa aleatório, que poderá ter uma percentagem qualquer de branches que são verdadeiros em determinadas condições de execução e outra percentagem qualquer de branches que são falsos em outras condições de execução.

predicted-untaken

Tal como podemos tomar partido de que todas as instruções de branch são *untaken*, também é válido que pensemos que todas as instruções de branch sejam *taken*, numa perspetiva do tipo **predicted-taken**. No entanto, pensar neste prisma, significa que o processador onde tentamos implementar esta solução terá um tipo qualquer de notação por

predicted-taken

flags que notifique quando é que um determinado branch é verdadeiro (códigos de aplicação de condição - *set condition codes*) ou outras implementações ainda mais potentes.

figura 2.13

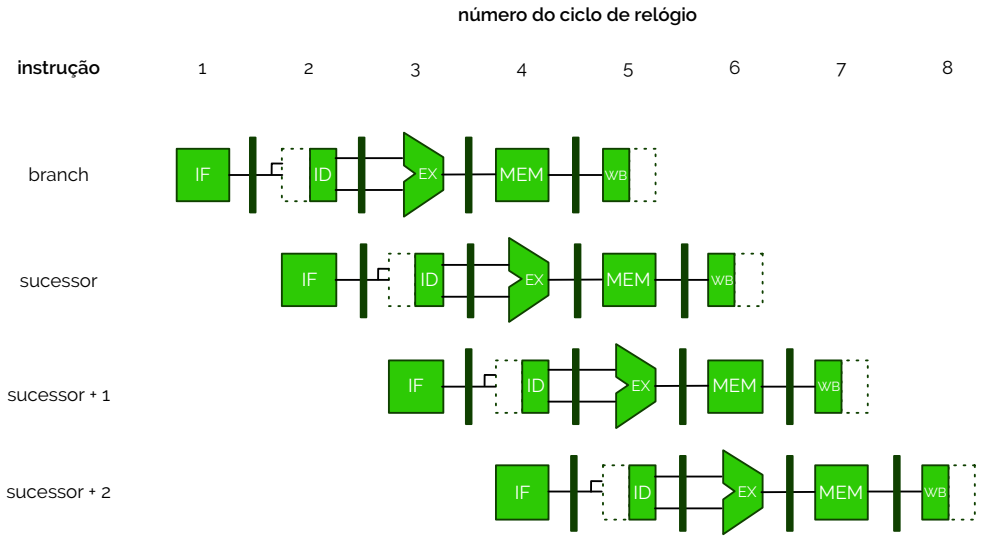
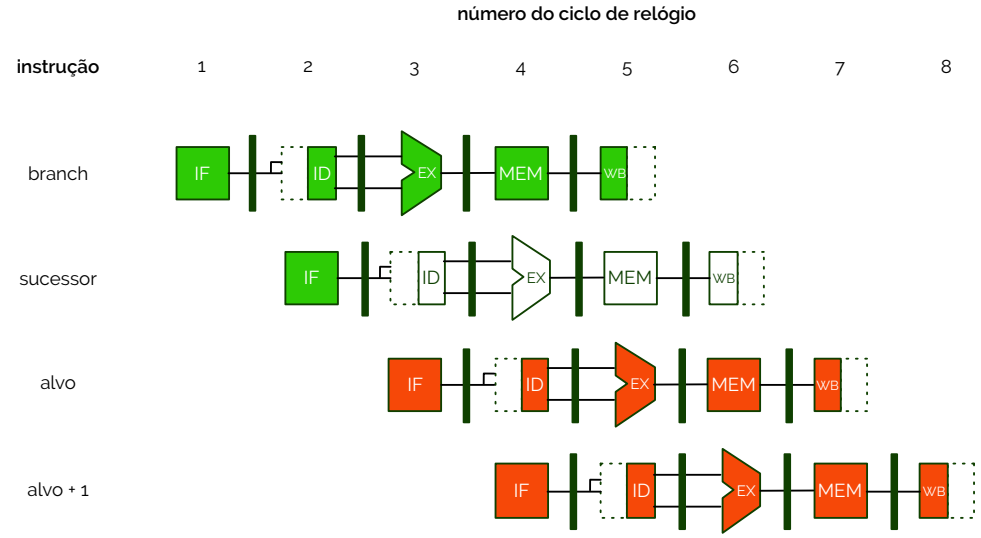


figura 2.14



Em qualquer uma das duas abordagens (*predicted-taken* ou *predicted-untaken*) o compilador tem um trabalho bastante importante em tornar o programa mais eficiente ou não, pela forma como organiza o código para que a sua execução tome o sentido mais linear possível e mais ajustado ao *hardware* em causa.

branch em atraso

Numa última solução vejamos o **branch em atraso** (esquema de *delayed branch*) o qual foi bastante usado nas arquiteturas do tipo RISC. Aqui, o ciclo de execução é definido pela instrução de branch propriamente dita, seguida do seu sucessor sequencial e do alvo caso seja tomado, o branch, como verdadeiro. Ao sucessor sequencial daremos o nome de **delay slot** e esta instrução é sempre executada, seja o branch *taken* ou *untaken*, não podendo ser uma instrução de branch. Na Figura 2.15 podemos ver este esquema em ação quando o branch é, de facto, *untaken* e, na Figura 2.16 podemos ver este esquema em ação quando o branch é *taken*. Claro está que em cada um destes casos a instrução sucessora sequencial (*delay slot*) é executada sendo que, caso o branch seja *taken*, a próxima instrução a ser executada será aquela à qual o branch calculou o salto (instrução-alvo) e, no caso de ser *untaken*, a próxima instrução a ser executada será a sucessora do branch, de forma linear e de acordo com a sequência original de instruções.

delay slot

O trabalho de melhoria de execução é uma tarefa importante e designada, principalmente, ao compilador, que é quem vai tratar de criar uma ordem de instruções melhor.

figura 2.15

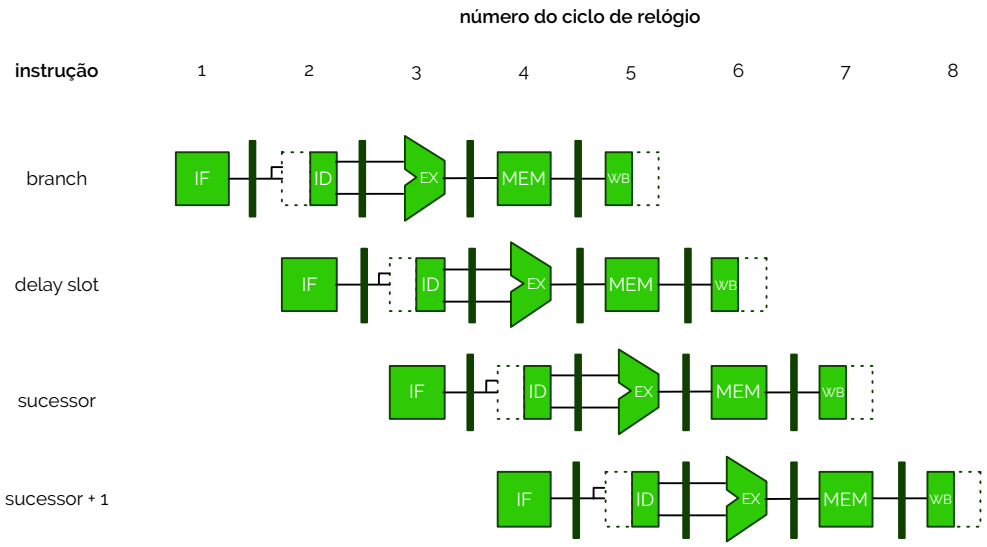
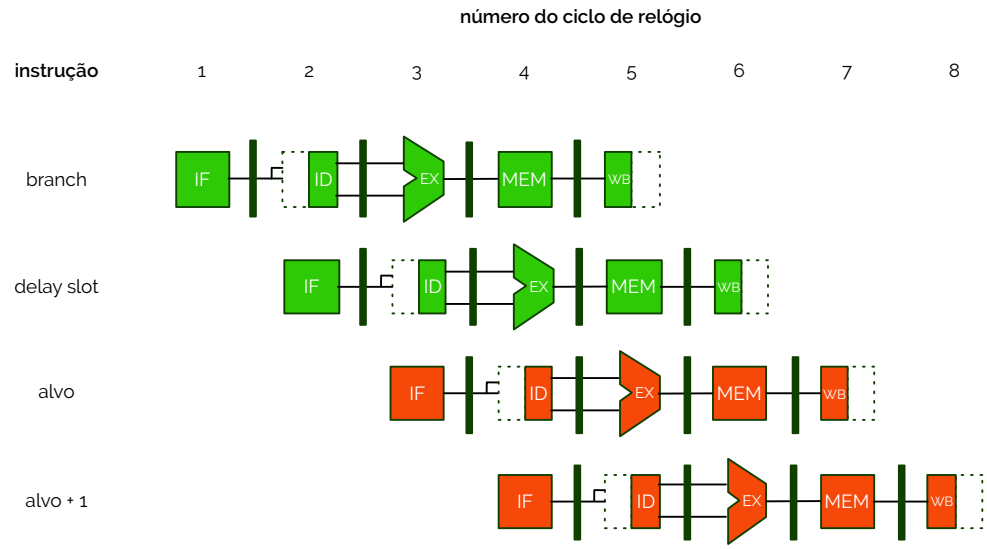


figura 2.16



Algumas alternativas de reordenação do código podem ser tomadas pelo compilador, entre elas tomar a instrução de *delay slot* sendo a anterior ao branch, a do alvo do branch ou a de *fall-through*. Vejamos a que corresponde cada caso com mais detalhe:

anterior ao slot

- **anterior ao branch** - o compilador vai inserir no *branch delay slot* a instrução supostamente executada antes da própria instrução de branch e que é independente de si (numa situação ideal, sendo que esta aplicação não interfere com nenhuma outra instrução). No Código 2.4 podemos ver um exemplo de aplicação, onde a instrução de `add` passa a ser executada após o branch.

código 2.4

```
// ordem de código antes de reordenação do código
a = b + c;
if (b == 0) then
<< branch delay slot >>
...

// ordem de código após reordenação de código
if (b == 0) then
a = b + c;
...
```

alvo do branch

- **alvo do branch** - o compilador vai inserir no *branch delay slot* a instrução a que o branch, caso seja *taken*, aponta e torna o ponto de branch ao seu sucessor (note-

se que quando o branch não é *taken*, uma instrução extra, não pretendida originalmente, é executada). No Código 2.5 podemos ver um exemplo de aplicação, onde a instrução de `sub` passa a ser executada após o branch.

código 2.5

```
// ordem de código antes de reordenação do código
a = b - c;
...
d = e + f;
if (a == 0) then
<< branch delay slot >>
...

// ordem de código antes de reordenação do código
a = b - c;
...
d = e + f;
if (a == 0) then
a = b - c;
...
```

- **fall-through** - o compilador vai inserir no *branch delay slot* a instrução que será executada de seguida se o branch não for *taken* (sendo que esta instrução estará ainda sobre execução quando o branch for *taken*). No Código 2.6 podemos ver um exemplo de aplicação, onde a instrução de `or` passa a ser executada sempre imediatamente a seguir ao branch.

fall-through

código 2.6

```
// ordem de código antes de reordenação do código
a = b + c;
if (b == 0) then
  << branch delay slot >>
  g = h | i;
  ...
d = e - f;
...

// ordem de código antes de reordenação do código
a = b + c;
if (b == 0) then
  g = h | i;
  ...
d = e - f;
...
```

De forma a melhorar as capacidades do compilador para que este preencha o *delay slot*, grande parte dos compiladores que possuem branches condicionais implementaram instruções de branch de **cancelamento** (ou anulamento). No fundo é como as instruções de *delayed branch* funcionam: quando um branch é *taken* a instrução no *delay slot* é executada; no entanto, quando o branch não é *taken*, a instrução no *delay slot* é mudada para uma não-instrução.

cancelamento

A rapidez efetiva destes esquemas para moderar os hazards de controlo aquando da execução do código podem ser expressadas por (2.1).

$$\text{speed up} = \frac{\text{overall CPI}_{\text{não-pipeline}}}{1 + \text{ciclos de stall do pipeline por instrução}_{\text{pipeline}}} \cdot \frac{\text{tempo de ciclo de relógio}_{\text{não-pipeline}}}{\text{tempo de ciclo de relógio}_{\text{pipeline}}}$$
$$= \frac{\text{overall CPI}_{\text{não-pipeline}}}{1 + (\text{frequência de branch} + \text{branch penalty})_{\text{pipeline}}} \cdot \frac{\text{tempo de ciclo de relógio}_{\text{não-pipeline}}}{\text{tempo de ciclo de relógio}_{\text{pipeline}}} \quad (2.1)$$

Estas abordagens que estudámos até agora não são propriamente as melhores, isto porque tomam sempre partido de um estado estático: ou os branches são *taken* ou são *untaken*. Por outras palavras, todas as abordagens que estudámos até agora são esquemas **estáticos** de baixo-custo que usam informação disponível em tempo de compilação e o próprio *profiling* baseia-se no mesmo código. Uma alternativa possível são os branches **dinâmicos** - nestas abordagens usam-se métodos para **prever** a direção dos branches durante a execução de um código.

estáticos

dinâmicos, prever

A forma mais simples de branching dinâmico usa um *buffer* de predição (vulgarmente denominado de **BPB**, do inglês *Branch-Prediction Buffer*) ou uma tabela de predição (vulgarmente denominada de **BPT**, do inglês *Branch-Prediction Table*). O BPB é uma memória muito pequena indexada pela parte menos significativa do endereço da instrução de branch que tem um bit por posição indicativo do branch ter sido, recentemente, tomado como *taken* ou como *untaken*. Com esta abordagem podemos assim prever, com alguma correção, o comportamento dos branches, sabendo qual foi o comportamento do mesmo num ciclo/passagem anterior, deixando uma pequena indicação, no *buffer*, que da útltima vez saltámos (ou não). No entanto, isto não nos permite saber se a predição é correta ou não, isto porque no caso de um ciclo *while*, por exemplo, enquanto a condição para permanecer em ciclo se mantém ativa, esta deixará no *buffer* de 1 bit a indicação de que o último branch foi verdadeiro (*taken*). Contudo, quando a execução largar o ciclo, estando a condição falsa, o branch é *untaken*, logo o *buffer* receberá o valor indicativo, num bit, de que o último salto foi falso - porque é que isto não nos ajuda a decidir se a predição é boa ou não? Numa próxima abordagem do branch, o processador irá verificar o valor do *buffer*, para prever o provável comportamento do branch e irá assumir que o salto é falso, iniciando a execução para a próxima instrução na sequência sem branch. Note-se que a predição é apenas uma pista sobre o que pode ser assumido no futuro, acerca de um branch, tendo as suas consequências na instrução que inicia a fase de fetch. Se esta predição falhar, então o processador perde um ciclo da sua execução e terá de efetuar a fase de fetch, da nova instrução na sequência correta, no ciclo seguinte (desperdiçando o anterior).

BPB

BPT

Portanto, esta estratégia de um *buffer* de um bit não tem o desempenho esperado, dado que mesmo quando o branch é quase sempre tomado como *taken*, porque será tomado, com maior probabilidade, de forma errada duas vezes, ao contrário de uma vez, para o caso de branches não *taken*. Para resolver este problema, uma solução possível é implementar o mesmo *buffer*, mas com 2 bits para complementar as transições entre as predições de *taken* e de *untaken*. Introduzimos, assim, alguma **histerese**, no processo de predição, este, representado numa máquina de estados na Figura 2.17.

histerese

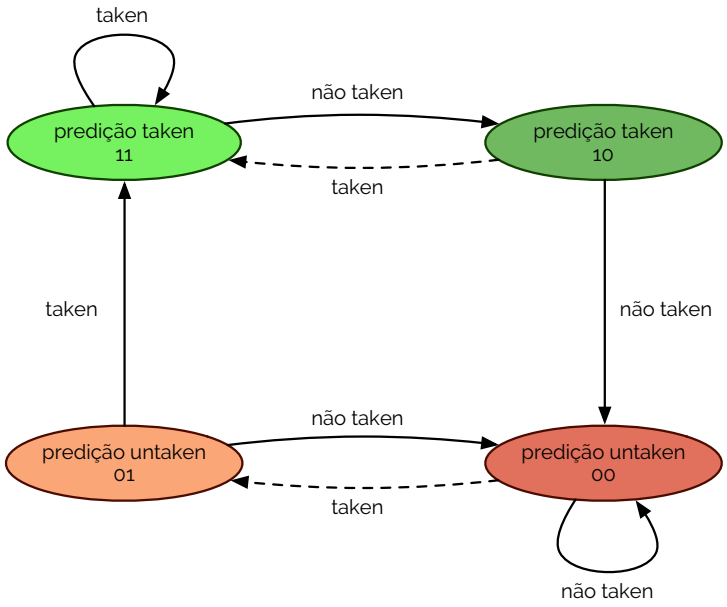


figura 2.17

Mais à frente iremos voltar a este assunto dos branches de controlo, onde estudaremos preditores em torneio ou correlacionados.

Implementação de um pipeline de 5 andares

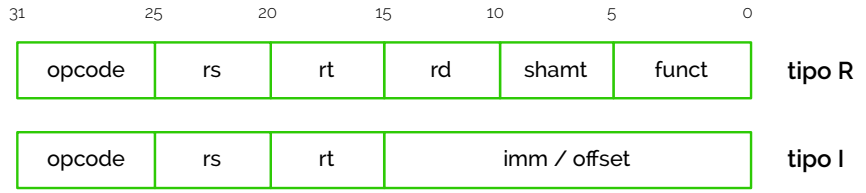
Até agora temos descrito o comportamento de um pipeline de cinco andares, no entanto ainda não o chegámos a descrever na sua forma física, isto é, exibindo os seus mó-

dulos lógicos reais, tal como fizemos na disciplina de Arquitetura de Computadores I (a2s1), onde numa primeira fase estudámos uma implementação *single-cycle* e numa última fase preparámos a mesma implementação para uma rotina pipelined, como estamos a abordar nesta disciplina.

Tal como foi descrito até ao momento, o nosso pipeline diz-se de cinco andares porque nele construímos, de forma lógica, cinco fases de execução de uma instrução: fase de *instruction fetch*, *instruction decode*, *execute*, *memory access* e *write-back*. Vimos também que é importante designar um conjunto de registos responsável por armazenar pequenos *snapshots* das várias fases nas transições entre estas, tendo assim os registos IF/ID (para o intermédio das fases de *instruction fetch* e de *instruction decode*), ID/EX (para o intermédio das fases de *instruction decode* e de *execute*), EX/MEM (para o intermédio das fases de *execute* e de *memory access*) e MEM/WB (para o intermédio das fases de *memory access* e de *write-back*).

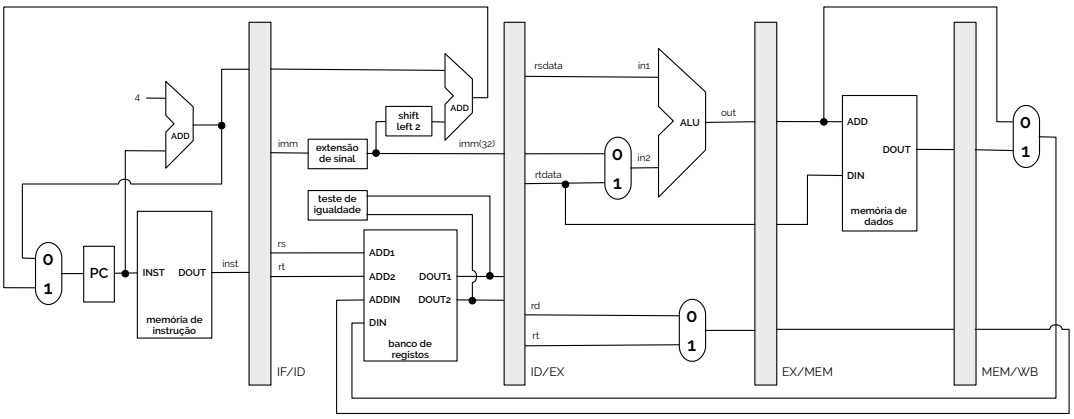
Só para recordarmos também, as instruções do nosso processador MIPS podem ser do tipo R (para operações aritméticas ou lógicas) ou do tipo I (para instruções aritméticas ou lógicas com imediatos - constantes -, instruções de transferência de dados com a memória ou branches). Na Figura 2.18 podemos recordar ambos os tipos de instrução.

figura 2.18



O pipeline em si, fora unidades de controlo, tem então a forma da Figura 2.19.

figura 2.19



Agora, o nosso pipeline não pode ser apenas a junção de todas as unidades conforme a Figura 2.19 - precisamos de uma **unidade de controlo**. A unidade de controlo do nosso pipeline contará com dois módulos: um principal, onde se regulam os acessos à memória (leitura e escrita), fontes de operandos para a ALU e controlo de branches, e um segundo, que identificará instruções em particular à ALU, para o caso de operações aritméticas ou lógicas. Note-se que, de todos os dados de uma instrução, quem comandará os módulos de controlo será a secção `opcode` e/ou `funct` (dependendo do tipo de instrução). Estes sinais de controlo e mais informações acerca dos mesmos poderão ser consultadas nos apontamentos da disciplina de Arquitetura de Computadores I (a2s1).

unidade de controlo

De forma a que as várias combinações de sinais consigam identificar de forma unívoca uma instrução qualquer do conjunto de instruções, ao longo do pipeline, é importante que estes sejam conduzidos e acompanhem o processo de uma instrução ao longo da sua execução. Assim sendo, deverá haver uma condução dos sinais (propagação) ao longo dos vários registos de transição do pipeline. Extendem assim os vários registos IF/ID, ID/

EX, EX/MEM e MEM/WB para poderem alojar, também os vários sinais de controlo ao longo das fases de execução. Esta extensão pode então ser vista na Figura 2.20.

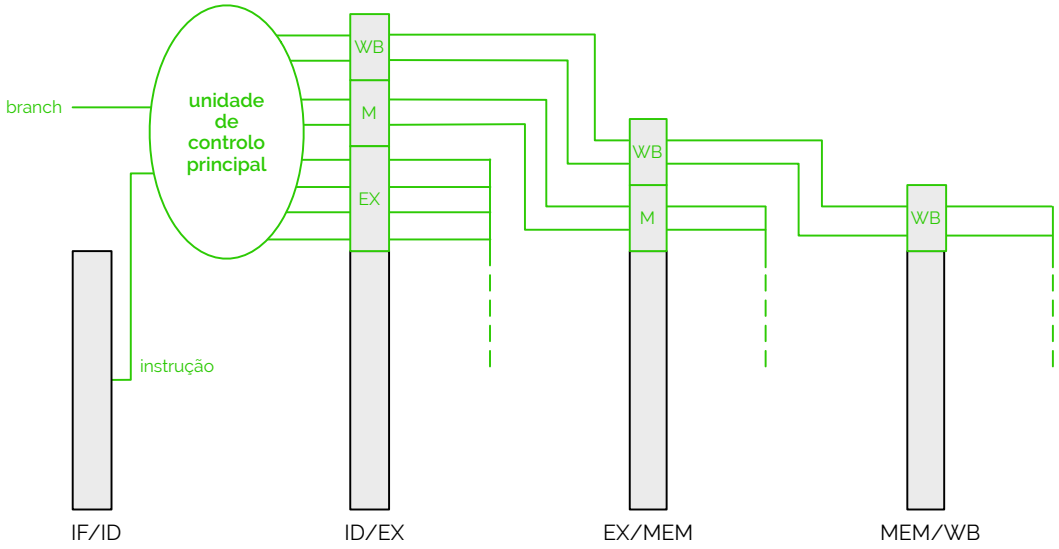


figura 2.20

Se aplicarmos esta unidade de controlo ao nosso pipeline da Figura 2.19 podemos obter a Figura 2.21, onde não exibimos as extensões dos registos de transição do pipeline para não levar mais entropia à figura. Note-se que nesta figura os componentes de controlo estão representados a verde.

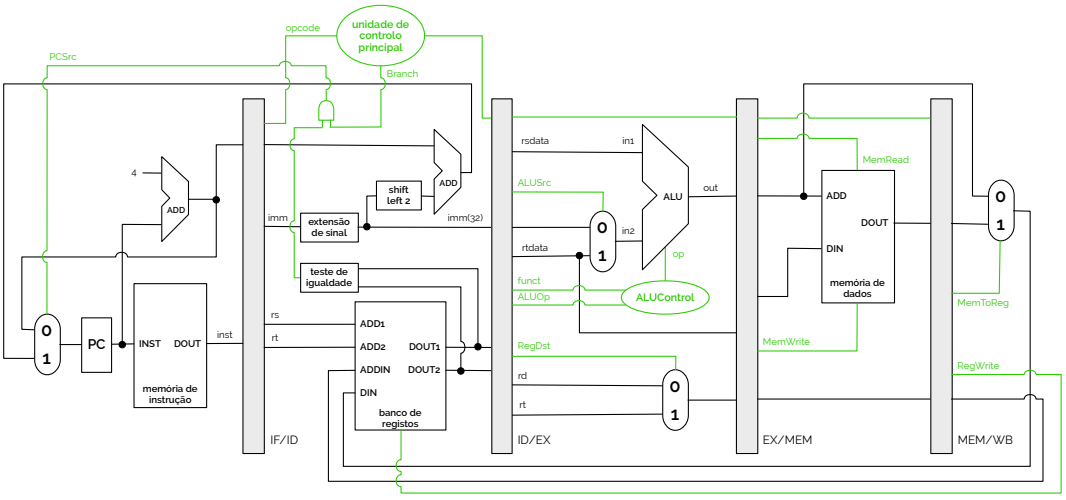


figura 2.21

Tendo a unidade de controlo já instalada agora é importante pôr em prática o que estudámos acerca das técnicas de controlo de *hazarding*. Primeiramente é importante tratar da implementação do *fowarding* no pipeline clássico de cinco andares. Para tal será importante, primeiro, conseguir designar três tipos de *hazards* de dados dadas as suas origens.

Começamos assim por designar os *hazards* de dados da fase de *instruction decode* (ID). Aqui, dizemos que temos uma situação de *hazarding*, quando os conteúdos dos registos do banco de registo são necessários para a conclusão de uma instrução como a `beq`, de forma a que se possa fazer uma comparação correta no bloco "teste de igualdade". Em alguns casos poderá não haver problema, mas no caso em que uma ou duas instruções anteriores alterem o valor de um registo cujo conteúdo será testado no ciclo seguinte (na instrução `beq`), então estaremos numa situação de *hazard* de dados. Para resolvermos este problema temos então que conseguir encaminhar os conteúdos (dados) das fases EX e MEM para a fase de ID, alimentando o "teste de igualdade" de 1 de 4 formas distintas para cada uma das suas entradas. Instalando para o efeito dois multiplexeres 4:1, em cada

uma das entradas de tal bloco, podemos escolher com um sinal `fwdIDA` e `fwdIDB` (para `DOUT1` e `DOUT2`, respetivamente), o valor de `DOUTx`, o valor de `out` (da ALU em EX), o valor do primeiro para o último resultado da ALU ou o valor da memória de dados (ou da segunda para a última operação da ALU). Estes reencaminhamentos estão todos sumariados na tabela da Figura 2.22.

figura 2.22

seleção MUX	fonte	descrição
`fwdIDA = 00`	IF/ID	O primeiro operando da ALU provém do registo dado
`fwdIDA = 01`	ID/EX	O primeiro operando da ALU é encaminhado do último resultado da ALU
`fwdIDA = 10`	EX/MEM	O primeiro operando da ALU é encaminhado do primeiro para o último resultado da ALU
`fwdIDA = 11`	MEM/WB	O primeiro operando da ALU é encaminhado do segundo (ou da memória de dados) para o último resultado da ALU
`fwdIDB = 00`	IF/ID	O segundo operando da ALU provém do registo dado
`fwdIDB = 01`	ID/EX	O segundo operando da ALU é encaminhado do último resultado da ALU
`fwdIDB = 10`	EX/MEM	O segundo operando da ALU é encaminhado do primeiro para o último resultado da ALU
`fwdIDB = 11`	MEM/WB	O segundo operando da ALU é encaminhado do segundo (ou da memória de dados) para o último resultado da ALU

Em termos mais apropriados para a análise do comportamento deste *forwarding* para a fase ID, no Código 2.7 podemos ver as regras para as repetivas ações e sinais.

código 2.7

```
se Branch e ID/EXE.RegWrite e (RegDst≠0) e (RegDst==IF/IDRegRs):
  fwdIDA = 01
  contrário se Branch e EX/M.RegWrite e (EX/M.RegDst≠0) e (EX/M.RegDst==IF/ID.RegRs):
    fwdIDA = 10
    contrário se Branch e M/WB.RegWrite e (M/WB.RegDst≠0) e (M/WB.RegDst==IF/ID.RegRs):
      fwdIDA = 11
      contrário:
        fwdIDA = 00

se Branch e ID/EXE.RegWrite e (RegDst≠0) e (RegDst==IF/IDRegRt):
  fwdIDB = 01
  contrário se Branch e EX/M.RegWrite e (EX/M.RegDst≠0) e (EX/M.RegDst==IF/ID.RegRt):
    fwdIDB = 10
    contrário se Branch e M/WB.RegWrite e (M/WB.RegDst≠0) e (M/WB.RegDst==IF/ID.RegRt):
      fwdIDB = 11
      contrário:
        fwdIDB = 00
```

Olhando agora para os *hazards* de dados que podem ocorrer na fase EX (fase de *execute*) temos que o conteúdo de um registo do banco de registos poderá ser requirido por uma instrução e ter sido modificado por uma outra instrução anterior, sendo que o seu valor atualizado se encontra num registo de transição do pipeline EX/MEM ou MEM/WB. Novamente, uma possível solução para este problema é instalar dois multiplexeres nas entradas da ALU (em `in1` e `in2`, multiplexeres com sinal de controlo `fwdEXA` e `fwdEXB`, respetivamente. Tais multiplexeres terão que encaminhar o valor do registo `rs` (ou `rt`), o valor do último resultado da ALU ou o valor da memória de dados, ou do primeiro para o último resultado da ALU. Mais, para o caso do segundo operando, ainda há mais uma opção de encaminhamento, sendo esse o valor do campo do imediato com extensão de sinal em 32 bits. Na Figura 2.23 podemos ver então uma tabela, análoga à anterior, onde se designam, sumariamente, o significado de cada um dos sinais. No Código 2.8 podemos ver uma diferente descrição do comportamento para o *forwarding* da fase EX.

código 2.8

```
se EX/M.RegWrite e (EX/M.RegDst≠0) e (EX/M.RegDst==ID/EX.RegRs):
  fwdEXA = 10
  contrário se M/WB.RegWrite e (M/WB.RegDst≠0) e (M/WB.RegDst==ID/EX.RegRs):
    fwdEXA = 11
    contrário:
      fwdEXA = 00
```

```
se EX/M.RegWrite e (EX/M.RegDst≠0) e (EX/M.RegDst==ID/EX.RegRt) e !ID/EX.ALUSrc:
  fwdEXB = 10
  contrário se M/WB.RegWrite e (M/WB.RegDst≠0) e (M/WB.RegDst==ID/EX.RegRs) e !ID/EX.ALUSrc:
    fwdEXB = 11
    contrário se ID/EX.ALUSrc:
      fwdEXB = 01
      contrário:
        fwdEXB = 00
```

seleção MUX	fonte	descrição
fwdEXA = 00	ID/EX	O primeiro operando da ALU provém do registo dado
fwdEXA = 10	EX/MEM	O primeiro operando da ALU é encaminhado do último resultado da ALU
fwdEXA = 11	MEM/WB	O primeiro operando da ALU é encaminhado do primeiro (ou da memória de dados) para o último resultado da ALU
fwdEXB = 00	ID/EX	O segundo operando da ALU provém do registo dado
fwdEXB = 01	ID/EX	O segundo operando da ALU provém do campo imediato com extensão de sinal para 32 bits
fwdEXB = 10	EX/MEM	O segundo operando da ALU é encaminhado do último resultado da ALU
fwdEXB = 11	MEM/WB	O segundo operando da ALU é encaminhado do primeiro (ou da memória de dados) para o último resultado da ALU

figura 2.23

Por fim, os *hazards* para a fase de memória, ocorrem quando os conteúdos de um registo do banco de registos é requirido por uma instrução e foram alterados por uma instrução anterior, atualmente guardada no registo de transição do pipeline MEM/WB. Para corrigir esta situação devemos então instalar novo multiplexer, neste caso 2:1, onde temos de escolher, com um sinal `fwdMEM`, se pretendemos o valor de `rt` ou o valor acabado de ler. Vejamos a tabela da Figura 2.24 para uma sumarização dos sinais de *forwarding* `fwdMEM`, e o Código 2.9 onde se poderá analisar o comportamento com mais detalhe.

seleção MUX	fonte	descrição
fwdMEM = 0	EX/MEM	O valor a ser possivelmente escrito na memória provém do valor do registo rt
fwdMEM = 1	MEM/WB	O valor a ser escrito na memória é o valor que acaba de ser lido

figura 2.24

```
se EX/M.MemWrite e M/WB.RegWrite e (EX/M.RegDst==M/WB.RegDst):
  fwdMEM = 1
  contrário:
    fwdMEM = 0
```

código 2.9

Assim sendo, tendo as nossas implementações analisadas, na Figura 2.25 podemos ver as alterações no modelo do nosso pipeline de cinco andares, onde as alterações para a unidade de *forwarding* estão representadas a roxo.

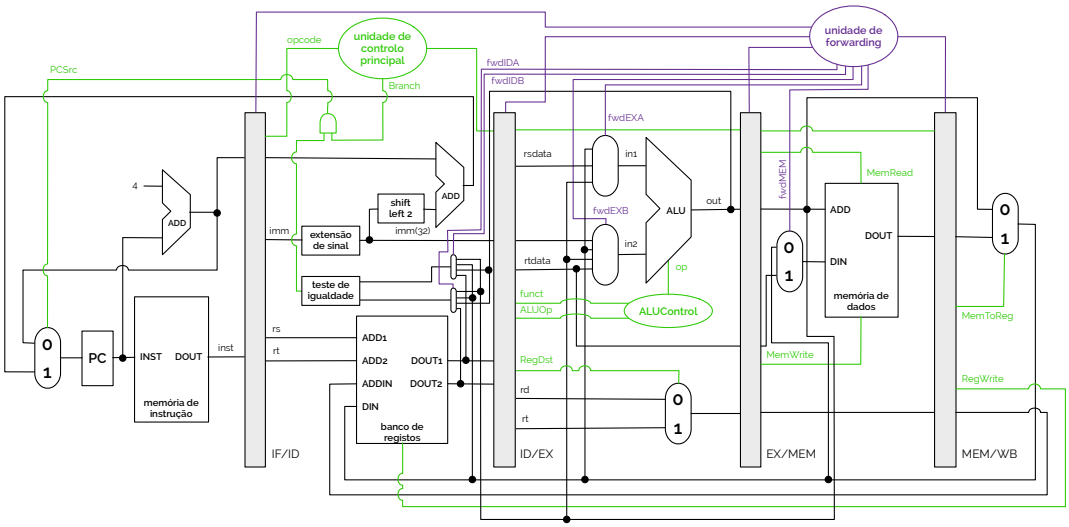


figura 2.25

Mas nem todos os *hazards* de dados, para esta implementação do pipeline de cinco andares, são solucionados com o *forwarding*. Como já tivemos oportunidade de verificar anteriormente, se uma instrução imediatamente a seguir a um *load* (carregamento da memória) tentar ler o mesmo registo que é escrito por este, então não haverá outra forma de continuar antes que o novo valor seja efetivamente produzido e retornado da memória. Isto significa que o progresso no pipeline deverá ser estagnado, isto é, deverão ser incluídos *stalls*.

Quando a instrução que segue um *load* é uma instrução da ALU, apenas é necessário um ciclo de *stall*, mas, por outro lado, quando é uma instrução como a `beq`, dois ciclos de *stall* são necessários.

De forma a que a instrução na fase ID seja parada, a instrução da fase IF deverá ser parada também. Uma forma simples de o fazer é previnido tanto o registo *program counter* (PC), como o registo de transição IF/ID, de mudarem os seus estados. Considerando assim que o conteúdo de tais registos é preservado, as instruções processadas nas fases de *instruction fetch* (IF) e de *instruction decode* (ID) são mantidas iguais. Por outro lado, as instruções nas outras fases poderão continuar a ser executadas como se nada tivesse acontecido. Isto significa que a instrução de não-operação (`nop`) deve ser gerada e introduzida na fase de execução (EX) no ciclo seguinte. Isto poderá ser concretizado através da desativação dos sinais de controlo quando estes são escritos para o registo de transição ID/EX.

Esta implementação pede então que seja introduzida uma unidade de *hazarding*, aqui denominada de **unidade de interlocking**. Na Figura 2.26 podemos ver a mesma unidade da figura anterior, mas com a adição da unidade de *interlocking*, que aqui aparece designada a cor laranja.

unidade de interlocking

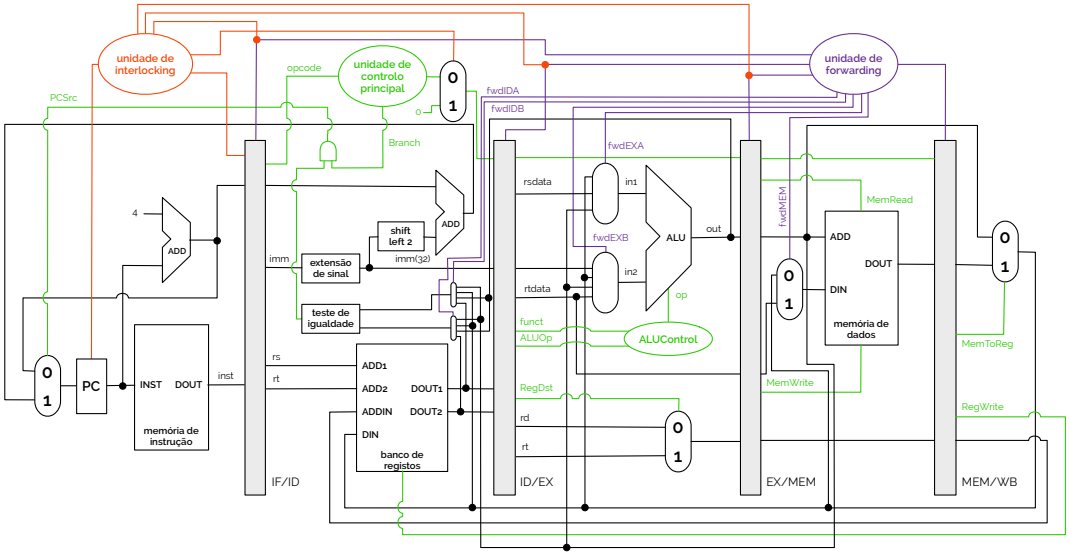


figura 2.26

A inserção de ciclos de *stall* depende assim das condições explícitas no Código 2.10.

```
se ID/EX.MemRead e !MemWrite e ((RegDst==IF/ID.RegRs) ou (RegDst==IF/ID.RegRt e !MemRead)):
  stall pipeline
  contrário se Branch e EX/M.MemRead e \
              ((EX/M.RegDst==IF/ID.RegRs) ou (EX/M.RegDst==IF/ID.RegRt)):
    stall pipeline
```

código 2.10

Exceções

Quando abordamos o desenho de processadores, o aspeto mais desafiante de todos é, sem dúvida, a unidade de controlo a ser implementada: não só por ser a parte mais difícil de engenhar, como a parte mais difícil de tornar mais rápida. Uma das partes mais difíceis de controlar são as **exceções** ou **interrupções**, como já abordámos na disciplina de Ar-

exceções, interrupções

quitetura de Computadores II (a2s2). Estes acontecimentos são tais que causam a paragem da leitura sequencial de instruções de forma não esperada, contrariamente aos saltos condicionais e incondicionais. As exceções típicas incluem pedidos de controladores associados a módulos de entrada/saída, invocação de chamadas ao sistema pelo nível de utilizador, falha de energia, entre outros...

Em implementações não-pipelined, a tarefa mais difícil é, sem dúvida, implementar o controlo de exceções nas fases de execução e de acesso à memória, sendo que estas têm sempre de ser terminadas. Tal implementação supõe assim que outro programa (geralmente o sistema operativo residente), é invocado para preservar o estado de execução do programa, corrigir a causa da exceção e restaurar o estado do programa, antes da instrução que causara o erro ser executada novamente - um mecanismo que claramente terá de ser completamente transparente ao programa em execução.

Se um pipeline tiver a capacidade de lidar com exceções, guardar o estado e restaurá-lo sem que haja afetação de qualquer outro programa já em execução, então diz-se que o processador é **restaurável** (em inglês *restartable*). Enquanto que os supercomputadores mais recentes e micro-processadores frequentemente não possuem tal função, quase todos os processadores dos dias de hoje suportam-no, pelo menos para o pipeline de cálculo inteiro, porque o resto baseia-se em tornar operacional a organização de memória virtual.

restaurável

Para que um sistema operativo, seja ele qual for, consiga manipular uma exceção, este deverá saber qual foi a razão que a causou, para além da instrução em específico que a causou ou que seria a seguinte a ser executada, caso não fosse lançada exceção. Existem assim dois métodos de fazer transportar esta informação até ao processador, que estudámos na disciplina de Arquitetura de Computadores II (a2s2).

Uma primeira forma de fazer transportar esta informação para o processador é através de um registo denominado **Cause**. O registo Cause é um registo de estado que grava num campo específico a causa para uma determinada exceção ter sido lançada, sendo que o seu valor é designado pelo *hardware* no momento da sua deteção. Quando as exceções são tratadas, este é o meio escolhido pelo sistema de operação para tomar conhecimento da causa do lançamento.

Cause

Uma segunda forma de enviar a informação de uma exceção é através de **exceções vetorizadas**, isto é, existindo vários pontos de acesso endereçáveis para a rotina de serviço à exceção, em geral, cada índice deste está associado com uma exceção em particular, pelo que a sua identificação é feita de forma bastante trivial.

exceções vetorizadas

Como já tivemos oportunidade de ver, quando uma exceção é tratada, o controlo do pipeline deve seguir os seguintes passos para guardar o estado atual do programa e permitir que o mesmo prossiga a sua execução no fim do tratamento, se assim for o caso:

1. O valor atual o *program counter* (PC) é guardado e é substituído pelo endereço do ponto de entrada correspondente à exeção para o próximo ciclo de *instruction fetch* (IF). O processador é então colocado num modo de privilégio, onde está disponível um maior conjunto de instruções para o tratamento das execeções;
2. Todas as instruções que sucedam à que provocou a exceção deverão ser tornadas não-operações, sendo que as instruções precedentes, se alguma ainda estiver em execução, deverão ser permitidas de terminar a sua execução, para que o estado preservado em memória seja consistente com o tempo de processamento da exceção;
3. Depois da rotina de tratamento de exceções ter sido iniciada, imediatamente atualiza o valor do registo *program counter* (PC) para o valor correto, o qual dependerá exclusivamente da causa da exeção. Deve-se ter noção de que, caso esteja em ação algum esquema de *delayed branch*, não será possível recriar o estado do processador com a memória de um valor do PC único - as instruções no pipeline não poderão ser sequencialmente relacionadas. Para que a rotina de serviço à exeção seja capaz de atualizar o PC guardado para o valor correto, será assim necessário ter os PC's necessários tanto quantos *slots* de *branch delay* houver, mais 1;

4. Finalmente, depois do término da rotina de serviço à exceção, instruções especiais do tipo de retorno da exceção restauram o estado original do programa e o modo anterior de execução.

Se o pipeline puder ser parado de forma a que as instruções precedentes à falha sejam completadas e este mesmo, juntamente com as instruções que lhe sucedem, possa reiniciar do zero, o pipeline diz-se ter **exceções precisas** (do inglês *precise exceptions*).

exceções precisas

Idealmente, a instrução em falha não deve alterar o estado de execução e, assim, de forma a manipular corretamente algumas exceções, é requirido que a instrução em falha não produza qualquer efeito. Para outras exceções, no entanto, como as de vírgula flutuante, a instrução em falha em alguns processadores escreve o seu resultado antes que a exceção possa ser tratada. Nestes casos, o *hardware* deverá estar preparado para devolver os operandos-fonte, mesmo se o destino seja o mesmo que um dos operandos-fonte.

Para ultrapassar este problema, muitos processadores de elevado desempenho introduziram dois modos de operação: um modo possui exceções precisas e outro, para ser mais desenvolvido, não possui. De facto, o modo de exceção precisa deverá ser mais lenta, dado que terá de permitir muito mais trocas entre instruções de vírgula flutante.

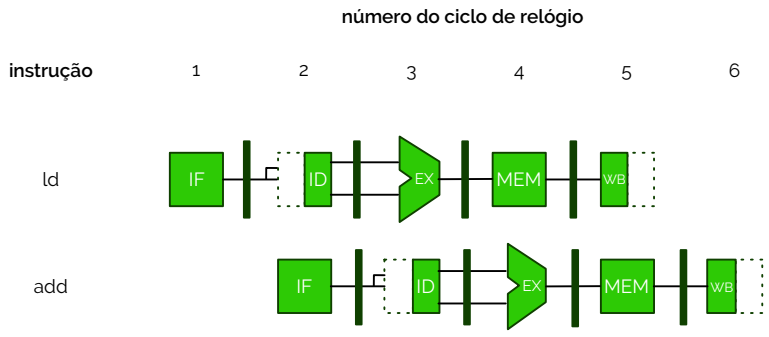
Com o pipeline, múltiplas exceções poderão ocorrer no mesmo ciclo de relógio, porque há múltiplas instruções em execução em simultâneo. Na Figura 2.27 podemos ver uma tabela com as exceções típicas (*within*) que poderão ocorrer no clássico pipeline de cinco andares, que temos estado a estudar [8].

figura 2.27

fase de pipeline	descrição
IF	page fault no fetch da instrução acesso de memória não alinhado violação da proteção de memória
ID	instrução não definida ou não implementada
EXE	exceção aritmética
MEM	page fault no fetch de dados acesso de memória não alinhado violação da proteção de memória
WB	-

Consideremos agora a seguinte sequência de instruções representadas na Figura 2.28.

figura 2.28



O par de instruções acima pode causar um *page fault* e uma exceção aritmética no mesmo ciclo de relógio. A forma de tratar este problema é respondendo à exceção de *page fault* e reiniciar a execução. A exceção aritmética irá ocorrer novamente, mas só depois é que deverá ser tratada de forma independente.

Em geral os tratamentos não são assim tão lineares. As exceções poderão até ocorrer fora-de-ordem, isto é, uma instrução poderá acionar uma exceção antes da sua precedente, que está em execução simultânea, e que também lança uma exceção. Para visualizar isto de forma mais fácil, consideremos uma situação em que a instrução de *load* gera um

page fault no *fetch* de dados na fase MEM e que a instrução `add` gera um *page fault* no *fetch* de instrução na fase IF.

Para desenvolver uma implementação pipeline de exceções precisas, a exceção acionada pela instrução de *load* carece de ser tratada primeiro. Mas como é que isso poderá ser realizado? O pipeline não pode tratar as exceções à medida que elas ocorrem no tempo, sendo que fazendo-o corre-se o risco das exceções serem processadas fora da ordem sequencial natural, *não-pipelined*. Pelo contrário, o *hardware* irá colocar todas as suas exceções causadas por uma dada instrução num vetor associado a tal instrução. O vetor de estado mantém-se junto, nas fases do pipeline, da instrução. Quando acontece uma indicação de exceção no vetor de estado, qualquer sinal de controlo que poderá causar um valor de dados ser escrito é desativado.

Quando a instrução entra na fase WB, o vetor de estado é verificado. Se alguma exceção é colocada, é tratada na ordem com que ocorre no tempo, numa implementação não-pipeline.

Operações multi-cycle num pipeline clássico de cinco andares

Até agora temos visto fundamentalmente operações inteiras no nosso processador, integrando uma versão pipeline. Se quisermos tratar também as operações de vírgula flutuante iremos ter que adaptar as nossas implementações, dado que é impraticável que todas as operações e multiplicações (e divisões) inteiras sejam implementadas num ciclo de relógio, ou mesmo em dois. Fazê-lo significaria que estenderíamos o período do relógio para permitir a execução das operações dentro do mesmo ou então que usaríamos uma larga quantidade de lógica adicional da implementação dos componentes funcionais, ou até mesmo as duas coisas. Por outro lado, o pipeline de vírgula flutuante irá comtemplar uma latência maior para as operações.

A melhor forma de pegar nesta ideia é imaginando as instruções de vírgula flutuante como tendo o mesmo pipeline que as instruções nativas inteiras, com duas diferenças relevantes: o ciclo de execução (EX) poderá ser repetido as vezes que forem necessárias até que uma instrução em execução seja completada - note-se que o número de repetições poderá variar de instrução para instrução; poderão existir múltiplas unidades funcionais.

Nesta nossa implementação devemos ter também em conta que um *stall* poderá ocorrer se a instrução a ser tomada causará tanto um *hazard* estrutural para a unidade funcional requirida, como um *hazard* de dados. Assim sendo, convém delinear então sobre que unidades é que devemos focar a nossa implementação. No nosso caso, devemos então considerar quatro unidades funcionais completamente separadas numa implementação pipeline, sendo uma unidade principal, capaz de instruções de transferência de dados com a memória e operações simples numa ALU, uma unidade capaz de fazer multiplicações inteiras e de vírgula flutuante, uma unidade que permita a soma, subtração e conversão de dados em vírgula flutuante (para inteiro e precisões diferentes) e uma unidade capaz de fazer divisões inteiras e de vírgula flutuante.

Se considerarmos que estas unidades funcionais não são pipelined, nenhuma instrução nova usando uma unidade funcional será necessária, se uma instrução anterior (usando a mesma unidade funcional) ainda está em operação e ainda não saiu da fase EX. Ainda mais, se uma instrução não puder passar para a fase EX, o pipeline será parado até este ponto.

Este tipo de implementação poderá ser visto na Figura 2.29 de forma esquemática onde porque apenas uma instrução se introduz em cada ciclo de relógio, todas as instruções entram no pipeline *standard* que estudámos até agora para operações inteiras. As operações de vírgula flutuante simplesmente entram num ciclo repetitivo quando atingem a fase EX, sendo que quando terminam prosseguem normalmente para as fases de MEM e WB para completar a execução. Para descrever este pipeline é importante redefinir dois conceitos muito importantes: a **latência** é o número de ciclos de relógios que intervêm entre uma instrução que produz um resultado e uma instrução que usa esse resultado; a **iniciação** ou intervalo de repetição é o número de ciclos de relógio que devem ser realizados

latência

iniciação

entre duas instruções lançadas para execução com operações do mesmo tipo. Por exemplo, iremos usar as latências e intervalos de iniciação mostrados na tabela da Figura 2.30 [8].

figura 2.29

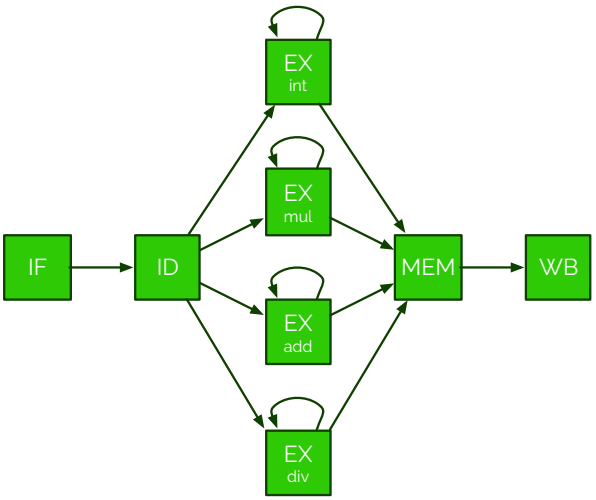


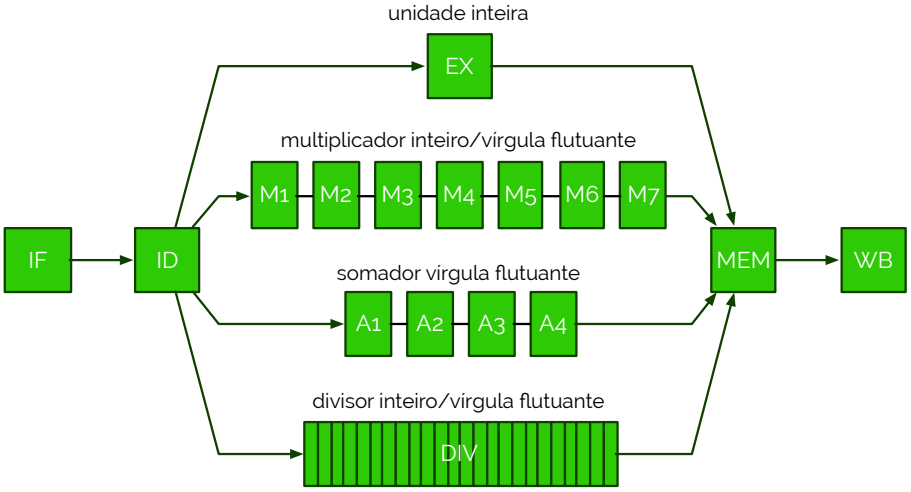
figura 2.30

unidade funcional	latência	intervalo de iniciação
ALU inteira	0	1
memória de dados (loads inteiros e de vírgula flutuante	1	1
adições em vírgula flutuante	3	1
multiplicações em vírgula flutuante (e inteiros)	6	1
divisões em vírgula flutuante (e inteiros)	24	25

As operações inteiras na ALU têm uma latência de zero simplesmente porque os resultados podem ser usados no ciclo de relógio seguinte à sua produção. As instruções de *load*, por outro lado, já possuem uma latência de um porque os resultados poderão apenas ser usados depois de um ciclo de relógio. Os seguintes, como a maior parte das operações consomem os seus operandos no início da fase de EX, a latência é usualmente o número de fases após EX que a instrução necessita para produzir o resultado: zero fases para operações inteiras sobre a ALU e uma fase para as operações de *load*. A exceção são os *stores*, onde a latência é de menos um.

Juntando ambas as informações da Figura 2.29 e da Figura 2.30, temos que podemos designar um pipeline semelhante ao da Figura 2.31.

figura 2.31



Em suma [7], podemos concluir alguns aspetos. Primeiro, como a unidade de divisão inteira/vírgula flutuante não tem uma implementação pipeline, poderão ocorrer *hazards* estruturais, havendo a necessidade, então, de os detetar de forma a parar a instrução que utiliza esta unidade funcional.

Como nem todas as instruções têm o mesmo tempo de execução, o número de escritas em registos num mesmo ciclo de relógio poderá ser maior que um. Os *hazards* de dados, que ocorrem quando uma instrução tenta escrever num registo antes de outra instrução, são assim possíveis, o que implica que a ordem de execução *não-pipelined* não seja mantida.

O facto de que as instruções possam ser completadas numa ordem diferente à qual são lançadas para execução, poderá causar alguns transtornos para o tratamento de exceções.

Os *hazards* de dados seguem assim um padrão semelhante do pipeline inteiro. Na Figura 2.32 podemos verificar um exemplo de comportamento para o pipeline de vírgula flutuante numa situação de *hazard* de dados.

figura 2.32

Na Figura 2.32 podemos reparar que a instrução de *store* foi parada um ciclo extra, de forma a que se previna um possível *hazard* estrutural de um conflito no acesso à memória na fase MEM. No entanto, porque apenas uma das instruções acede, de facto, à memória, ambas poderão correr em simultâneo na fase MEM, se *hardware* extra for adicionado.

Se considerarmos que o banco de registos apenas tem uma única porta de escrita, sequências de operações em vírgula flutuante, tal como instruções de *load* de vírgula flutuante acompanhadas de outras operações do mesmo nível, poderão dar aso a conflitos no acesso à porta de escrita. Na Figura 2.33 vejamos três instruções que tentam fazer um *write-back* no banco de registos em simultâneo, como podemos ver no ciclo 11.

figura 2.33

Note-se que o caso da Figura 2.33 não é o pior caso, sendo que uma divisão recente na unidade de vírgula flutuante terminaria, também, no mesmo ciclo de relógio. Note-se ainda, que embora as instruções estejam todas no ciclo 10 em fase MEM, somente a instrução de *load* é que realmente utiliza a memória, pelo que não é provocado qualquer *hazard* estrutural pela fase MEM.

Com uma porta de escrita única, o processador deverá serializar a realização das instruções. Para o efeito, o número de portas de escrita poderia ser aumentado, mas esta solução poderá não ser tão atrativa, pelo modo que portas extra de escrita seriam raramente usadas. Ao invés de tal solução, podemos escolher detetar o *hazard* estrutural e agendar o acesso à porta de escrita.

Para fazer tal deteção, precisamos de ir acompanhando o uso da porta de escrita na fase de *instruction decode* e fazer *stall*, caso seja necessário, à instrução atual antes que entre em execução, tal como se processa para qualquer outro *hazard* estrutural. A implementação desta solução requer um registo de deslocamento, denominado de **registo de reserva** (do inglês *reservation register*), cujo conteúdo se move na direção oposta ao fluxo de instruções no pipeline a cada ciclo de execução. O propósito é sinalizar os ciclos onde as instruções já executadas escreverão no banco de registos de vírgula flutuante. Assim, quando uma instrução na fase de *instruction decode* terá de escrever no banco de registos, a fase correspondente no registo de deslocamento é verificada. Se o bit do registo estiver *set* (a '1'), isto significa que outra instrução estará a fazer o mesmo no mesmo ciclo de relógio, pelo que ocorrerá um *stall*. Caso contrário, esse bit do registo de deslocamento será colocado a *set* e a instrução continuará a sua execução.

registo de reserva

Esta solução tem a vantagem de permitir que a unidade de *interlocking* se mantenha inalterada. O custo é basicamente a inserção de um registo de deslocamento e de toda a lógica para determinar o possível conflito de escrita.

Uma outra solução, que terá mais vantegns em termos do adiamento da deteção dos conflitos até ao momento em que se torna trivial verificá-los, é parar uma instrução conflituosa quando esta tentar aceder tanto a fase MEM, como a fase WB. Aqui, qualquer uma das instruções em conflito poderá entrar em *stall*. A desvantagem deste sistema está no facto de tornar o controlo do pipeline mais complexo, sendo que os *stalls* agora poderão aparecer em dois locais possíveis.

Os *hazards* de dados aparentemente não são dramáticos e poderão ser descartados. A noção é de que nunca deverão ocorrer num código bem escrito, porque nenhum compilador gera duas escritas num mesmo registo sem uma leitura. No entanto, se a sequência de instruções for inesperada, eles poderão de facto existir e assim produzir resultados errados. Consideremos para o efeito o Código 2.11.

código 2.11

```
        bnz     r1, foo
        div.d   f0, f2, f4
        . . .
foo:    l.d     f0,0(r3)
```

Se o branch do Código 2.11 for *taken*, a instrução de *load* seguirá para a fase de *write-back* (WB) antes da instrução de divisão terminar, criando assim uma inconsistência no estado do programa daqui para a frente.

Existem duas formas de tratar este *hazard*. Uma primeira solução é atrasar a execução da segunda instrução de escrita até que a primeira entre na fase MEM. Uma segunda solução é marcar a primeira instrução detetando um *hazard* e desabilitando a mudança para o registo - a segunda instrução poderia assim ser lançada de imediato. Como é muito raro tal *hazard* ocorrer, é indiferente qualquer uma das soluções apresentadas.

Um outro problema causado por estas instruções cuja execução é mais demorada pode ser ilustrada com a seguinte sequência de código (Código 2.12).

código 2.12

```
div.d   f0, f2, f4
add.d   f10, f10, f8
sub.d   f12, f12, f14
```

Esta sequência de código parece bastante linear, dado que não há quaisquer dependências. No entanto, um problema coloca-se - há uma instrução que, tendo sido iniciada antes, poderá terminar depois de uma seguinte. Neste exemplo do Código 2.12 podemos esperar que a operação de soma e de subtração se completem antes da de divisão. A este fenómeno damos o nome de **out-of-order completion** (em português algo como conclusão fora-de-ordem). Tal acontecimento é habitual em pipelines com operações morosas em termos de execução. Porque a deteção de *hazards* previnirá qualquer dependência entre instruções, então porque é que este fenómeno é problema? Suponhamos para o efeito que a operação de subtração causa uma exceção aritmética de vírgula flutuante em tal ponto que a instrução de soma já terminou a sua execução, mas a operação de divisão ainda não. O resultado será uma **exceção imprecisa**, algo que tentamos evitar que aconteça. Até pode parecer que isto poderá ser tratado deixando o pipeline drenar, tal como fazemos com o pipeline de inteiros. Por exemplo, se a operação de divisão decidisse lançar uma exceção aritmética de vírgula flutuante antes da operação de soma ser completada, não poderíamos ter uma exceção precisa ao nível do *hardware*. De facto, como a operação de soma destroi um dos seus operandos, não conseguiríamos restaurar o estado antes da operação de divisão, mesmo com ajuda de um *software*.

out-of-order completion

exceção imprecisa

Uma família de processadores de arquitetura MIPS que envolve todos estes detalhes de processamento conforme o que referimos até agora é a família **MIPS R4000**. Esta família implementa assim o conjunto de instruções MIPS64, mas usa um pipeline mais profundo que o nosso de cinco andares, tanto para instruções inteiras como de vírgula flutuante. Este pipeline diferente permite atingir frequências de relógio maiores, decompondo o pipeline de cinco andares em oito andares. Como o acesso à cache é tempo particularmente crítico, as fases extra do nosso novo pipeline são decomposição direta do acesso à memória. A este novo pipeline vulgarmente damos o nome de **superpipeline**. Na Figura 2.34 podemos ter uma melhor noção das oito fases de processamento do nosso pipeline do R4000.

MIPS R4000

superpipeline

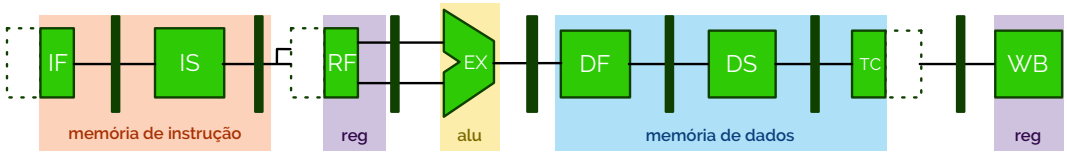


figura 2.34

As fases deste novo pipeline, sendo mais, têm a seguinte nomenclatura:

- *Instruction Fetch First* (IF) - primeira metade do *fetch* de instrução, sendo que o valor do *program counter* (PC) é atualizado nesta fase, tal como a iniciação do acesso à cache de instrução;
- *Instruction Fetch Second* (IS) - segunda metade do *fetch*, onde se completa o acesso à cache de instrução;
- *Register Fetch* (RF) - fase equivalente à de *instruction decode*, onde se faz o *fetch* dos registos, verificação de *hazards* e deteção de *hits* na cache;
- *Execution* (EX) - fase normal de execução, que inclui cálculo de endereços efetivos, operações sobre a ALU, avaliação de condições e cálculo de alvos do branch;
- *Data Fetch First* (DF) - fase sobre a qual é feito um *fetch* nos dados da memória cache;
- *Data Fetch Second* (DS) - segunda parte da fase anterior, onde se conclui o acesso à memória cache, de dados;
- *Tag Check* (TC) - onde se determina se se acertou nos dados pretendidos (deteção de *hit*);
- *Write-back* (WB) - retorno de valores para operações de carregamento da memória ou re-escrita em registos.

Embora os acessos à memória de dados e de instrução ocupem múltiplos ciclos de relógio, estes estão organizados estritamente com um pipeline pelo que uma nova instrução poderá partir a cada ciclo de relógio. De facto, como poderá ser verificado, como a deteção de *hits* na cache está sempre patente no fim de cada acesso à memória, o pipeline tenta utilizar os dados mesmo antes da deteção de *hits* acontecer e ser terminada. Se entretanto ocorrer um *miss*, então o pipeline tentará reverter um ciclo - quando os dados estiveram corretos por uma última vez.

Denote-se que esta latência maior no pipeline não só aumenta a lógica de *forwarding* necessária, como também aumenta os atrasos nos carregamentos de memória e branches. O atraso de carregamento de memória (operações de *load*) é de 2 ciclos, sendo que os dados estão disponíveis no fim da fase DS (Figura 2.35). Já no que toca ao branch, os seus atrasos são de 3 ciclos, sendo que a condição de salto é calculada somente durante a fase de execução (EX) - Figura 2.36.

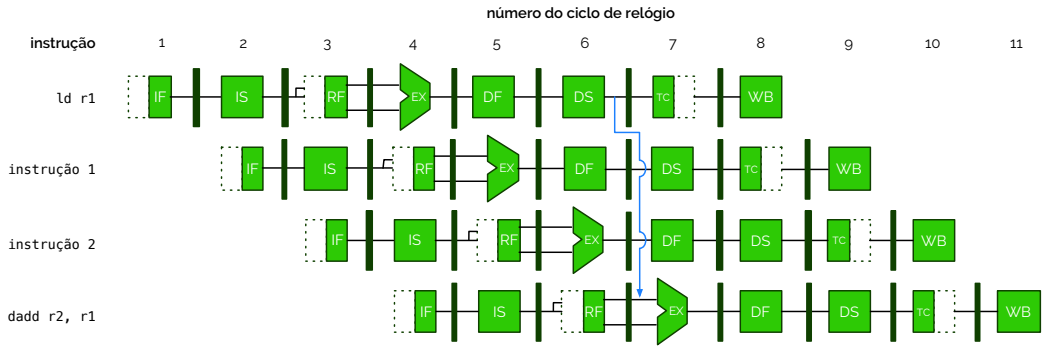


figura 2.35

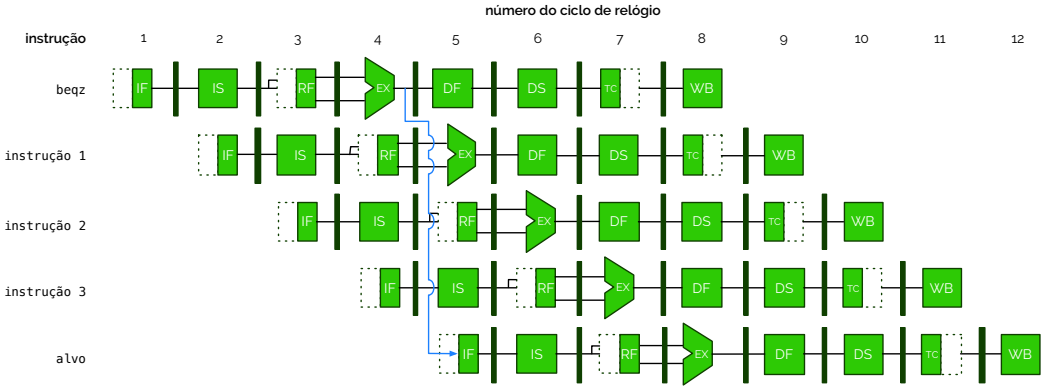


figura 2.36

A unidade de vírgula flutuante consiste em três unidades funcionais: um somador de vírgula flutuante, um multiplicador e um divisor. A lógica da unidade somadora é reaproveitada no fim dos cálculos de multiplicaões e divisões. As operações de precisão dupla poderão durar de 2 ciclos de relógio (para uma negação) a 112 ciclos (para uma raiz quadrada). Cada unidade funcional pode ser pensada como tendo até oito fases de processamento diferentes. Existe uma cópia simples de cada uma destas fases e diferentes instruções poderão usar uma fase em particular zero ou mais vezes e combiná-las numa determinada ordem. Considerando a tabela da Figura 2.37 veja-se a decomposição das instruções na tabela da Figura 2.38.

fase	unidade	descrição	fase	unidade	descrição
A	somador	soma de mantissas	R	somador	arredondamentos
D	divisor	divisões	S	somador	deslocamentos
E	multiplicador	teste de exceções	U		unpack de vírgula flutuante
M	multiplicador	primeira fase multiplicação			
N	multiplicador	segunda fase multiplicação			

figura 2.37

instrução	latência	intervalo de repetição	fases de processamento
soma - subtração	4	3	U - S+A - A+R - R+S
multiplicação	8	4	U - E+M - M - M - M - N - N+A - R
divisão	36	35	U - A - R - D^{28} - D+A - D+R - D+A - D+R - A - R
raiz quadrada	112	111	U - E - $(A+R)^{108}$ - A - R
negação	2	1	U - S
valor absoluto	2	1	U - S
comparação	3	2	U - A - R

figura 2.38

Existem quatro grandes causas para *stalls* do pipeline ou perdas que previenm que a instrução lançada para execução seja atendida nas perfeitas condições. Tais *stalls* podem ser provenientes dos *loads* (atrasos produzidos pelo uso de valores carregados um ou dois ciclos depois da execução do *load*), provenientes de perdas ou branches (dois ciclos de relógio de atraso depois de cada branch considerado *taken* e um ciclo de relógio se o *branch delay slot* não puder sido preenchido com nenhuma instrução útil), provenientes de resultados de vírgula flutuante (atrasos causados porque um operando é requirido e ainda não foi calculado) ou provenientes da estrutura de vírgula flutuante (atrasos causados porque as fases de processamento na unidade de vírgula flutuante não estão disponíveis quando são necessárias).

Eficiência do pipelining

A simplicidade de um conjunto de instruções é uma propriedade fundamental no desenho de um pipeline para a sua implementação. *Sets* de instrução mais simples oferecem ainda mais uma vantagem, pelo que facilitam a agenda de instruções individuais no processador. Estas vantagens parecem assim ser tão significantes que quase todos os mais recentes processadores CISC primeiro traduzem as suas instruções para RISC-*like* e só depois é que procedem para o seu pipeline e agendamento.

Como meio para exprimir o grau de sucesso obtido pela execução de um determinado programa num processador com pipeline, a seguinte equação poderá ser usada, onde CPI_{ideal} é o número de ciclos de relógio por instrução num caso ideal, sendo uma medida de desempenho máximo atingível pela implementação e diferentes tipos de *stalls* são assumidos a ter valores médios por instrução (2.2).

$$CPI_{programa} = CPI_{ideal} + \text{stalls estruturais} + \text{stalls de dados} + \text{stalls de controlo} \quad (2.2)$$

A quantidade de paralelismo disponível num *basic block* de um programa é relativamente pequeno. Para um programa típico MIPS, por exemplo, a frequência de branches dinâmicos ronda os 15% e 25%, o que significa que não mais que seis vezes existem instruções a serem executadas entre pares de branches. Mais, sendo que estas instruções provavelmente dependem umas das outras, a quantidade de instruções sobrepostas que ocorreriam dentro de um *basic block* seria, provavelmente, mais pequeno que o tamanho médio dos blocos.

Dependências de dados, nomes e controlo

Algo que é crítico sabermos como fazer é determinar como é que uma instrução depende de outra, isto é, para conseguirmos ter uma noção quanto paralelismo é que existe num programa e quanto é que ainda pode ser explorado. Em particular, para explorar o paralelismo dentro das instruções de forma eficiente devemos conseguir identificar que instruções é que verdadeiramente permitem a execução paralela. Se duas instruções se dizem **independentes** uma da outra, então estas poderão ser executadas em paralelo sem que novos *stalls* sejam inseridos num pipeline de profundidade arbitrário, assumindo que há

independentes

recursos suficientes. Se duas instruções se dizem **dependentes** uma da outra, então significa que haverá restrições temporais a serem respeitadas para uma correta execução do programa e tais instruções poderão apenas ser parcialmente sobrepostas. Existem assim três tipos de dependências consideráveis importantes - dependências de dados (ou de dados verdadeiros), de nomes e de controlo.

dependentes

Olhando primeiro para as dependências de dados consideremos uma instrução j que se diz dependente de dados de uma instrução i se e só se a instrução i produz um valor que é usado pela j, a instrução j é dependente de dados de uma instrução k e esta é dependente da instrução i, ou as instruções j e i estão ligadas por uma cadeia de dependências do segundo tipo. Note-se que, por exemplo, na instrução `add r1, r1, r1` não se considera qualquer dependência de dados. Por outro lado, consideremos para o efeito de análise, o Código 2.13.

código 2.13

```
loop:   l.d     f0, 0(r1)
        add.d   f4, f0, f2
        s.d     f4, 0(r1)
        daddui  r1, r1, -8
        bne     r1, r2, loop
```

Por questões de simplicidade assumamos que os efeitos de um *delayed branch* são ignorados. As dependências de dados estão representadas a vermelho, castanho e azul.

A existência de dependências implica que haja uma cadeia de um ou mais *hazards* de dados entre instruções sucessivas. Executando as instruções num processador com implementação de pipeline com unidade de *interlocking* e com uma profundidade maior que a distância entre as instruções medidas em ciclos de relógio, causará com que o processador detete um *hazard* e um *stall*, caso não seja passível de ser resolvido com uma unidade de *forwarding*, por conseguinte, reduzindo a sobreposição. Num pipeline sem unidade de *interlocking*, por outro lado, é da responsabilidade do compilador agendar instruções dependentes de forma a que quando estas corram não se sobreponham por completo com outras.

A presença de dependências de dados numa sequência de instruções reflete a dependência de dados do código-fonte de onde a sequência original fora gerada. Assim, se na sequência original já houver por si dependências, então estas deverão ser mantidas, sendo que poderão ser **propriedades** dos programas. Este caso de dependência é importante ter em conta, dado que a partir de agora devemos ser capazes de saber distinguir e analisar várias formas de como é que o paralelismo de instruções poderá ser explorado. Em suma, portanto, uma dependência de dados fornece-nos tópicos de análise para três ideias específicas: a possibilidade de haver um *hazard*, a especificação de como é que as instruções devem ser executadas, em termos de ordem, e um majorante em quanto paralelismo é suportável por uma porção de código. Uma dependência poderá assim transformar-se em dois produtos diferentes: código com as dependências mantidas, mas onde se evita a ocorrência de *hazards*; código transformado, sem dependências de dados.

propriedades

Através de um agendamento do código, numa primeira fase, é possível evitar a ocorrência de *hazards*, sem que se alterem as dependências por si. Esta tarefa pode ser cumprida tanto a nível de compilador, como a nível mais físico, de *hardware*.

Um valor de dados é comunicado entre as instruções tanto através de um registo do banco de registos, como de um local de memória. Quando o fluxo de informação ocorre através de um registo, detetando a dependência, então é linear, porque os nomes dos registos são fixos para cada instrução, embora se possa tornar bastante mais complicado se contarmos com a intervenção dos branches, dadp que as preocupações a nível de correção forçam o compilador ou o *hardware* a serem conservativos. Quando o fluxo de informação toma lugar através de locais de memória, a deteção é muito mais complicada, sendo que dois endereços poderão referir a mesma posição, mas ter aspetos diferentes. Mais, o endereço efetivo de um *load* ou de um *store* poderão alterar de uma execução de uma instrução para a seguinte, complicando ainda mais a deteção de uma dependência.

Outro tipo de dependências possível é a **dependência de nomes**. Uma dependência de nome está presente quando duas instruções usam o mesmo registo ou posição de

dependência de nomes

memória, denominado de **nome**, mas sem nenhum fluxo de informação a tomar lugar entre ambas. Existem assim dois tipos de dependências de nome a ter em consideração: as **antidependências**, que entre uma instrução j e uma instrução i ocorre quando a instrução j escreve num registo ou posição de memória que é lida pela instrução i - a ordem original das instruções deve ser preservada para garantir que o valor correto é lido; as **dependências de saída**, que entre uma instrução i e uma instrução j ocorre quando ambas as instruções escrevem um valor para o mesmo registo ou posição de memória - a ordem original das instruções deve ser preservada para garantir que o valor correto é escrito.

nome

antidependências

dependências de saída

Como uma dependência de noem não é uma dependência verdadeiramente de dados, e como não existe qualquer valor transmitido entre as instruções envolvidas, as instruções por si podem executar em simultâneo, ou serem reordenadas, desde que o nome usado nas instruções seja alterado, de forma a remover o conflito.

Com os conflitos das dependências de dados e de nomes podemos classificar os *hazards* que daí provêm em três grandes categorias. Para o efeito, consideremos uma instrução i e j, com i a preceder j no programa[1]:

- **RAW** (*Read After Write*) - este *hazard* de dados ocorre quando j tenta ler o oprando antes de i escrever um valor nele, pelo que j irá receber um valor errado. Esta é a forma mais comum de *hazard* de dados e corresponde a uma dependência de dados;
- **WAW** (*Write After Write*) - este *hazard* de dados ocorre quando j tenta escrever um valor num operando antes que i escreva o seu valor no mesmo, pelo que o valor final estará errado. Este *hazard* acontece geralmente em pipelines que permitem a escrita e mais do que uma fase de execução ou que permitam instruções fora-de-ordem (*out-of-order completion*), pelo que corresponde a uma dependência de saída;
- **WAR** (*Write After Read*) - este *hazard* de dados ocorre quando j tenta escrever um valor num operando antes que i o leia, pelo que a instrução i irá obter um valor errado. Este *hazard* não poderá ocorrer em grande parte dos pipelines, mesmo os mais profundos e de vírgula flutuante, porque usualmente todas as leituras são feitas muito cedo e todas as escritas muito tarde.

RAW

WAW

WAR

Por fim, um último tipo de dependências a referir são as **dependências de controlo**. Uma dependência de controlo determina a ordem de uma instrução i com respeito a um branch, de forma a que a instrução i seja esxecutada na ordem correta do programa e apenas onde o deverá ser. Todas as instruções, à exceção das que residem no primeiro *basic block* do programa, é dependente de controlo de um conjunto de branches e, em geral, tais dependências devem ser preservadas para manter a ordem do programa. Consideremos assim, para o efeito, o excerto de código em Código 2.14.

dependências de controlo

código 2.14

```
if (condição1) {
  S1;
}

if (condição2) {
  S2;
}
```

No Código 2.14 podemos dizer que `S1` é dependente de controlo da condição `condição1` e que `S2` é dependente de controlo da condição `condição2`, mas não da `condição1`.

Em geral, existem duas restrições que são importantes colocar pelas dependências de controlo: uma instrução que é dependente de controlo de um branch não pode ser movida para antes deste, pelo que a sua execução deixaria de ser controlada pelo branch; uma instrução que não é dependente de controlo de um branch, não poderá ser movida para depois de um, pelo que a sua execução passaria a ser controlada pelo branch.

1 RAR (Read After Read) não é um hazard de dados dado que a combinação de operações é idempotente.

Quando os processadores preservam estritamente a ordem dos programas, eles garantem também que as dependências também são preservadas. Note-se, não obstante, que é possível, embora seja ineficiente, executar instruções que não devem ter sido executadas, violando as dependências de controlo, se isto não afetar a correção do programa. Isto também significa que estas dependências de controlo naõ são uma propriedade crítica dos programas que deve ser preservada. No entanto, existem duas dependências de controlo que são críticas, entre elas, o comportamento de exceção e o fluxo de dados.

Preservar o comportamento de exceção significa que quaisquer mudanças na ordem de execução das instruções não devem alterar a forma como as exceções são lançadas no programa. Frequentemente isto é ignorado, significando que a reordenação das instruções a serem executadas não devem causar nenhuma exceção no programa. Para não haver dúvidas, consideremos o Código 2.15.

código 2.15

```
        daddu   r2, r3, r4
        beqz    r2, label1
        ld      r1, 0(r2)
label1: . . .
```

Está claro que se a dependência de dados que envolveo registo `r2` não for mantida, o resultado do programa poderá ser diferente. Menos óbvio é o facto de que se ignorarmos a dependência de controlo e movermos a instrução de *load* para um local anterior ao branch, esta instrução poderia causar uma violação da proteção de memória.

Para resolver a reordenação de instruções e ainda assim preservar as dependências de dados, seria necessário ignorar a exceção quando o branch é *taken*. Mais à frente, quando abordarmos especulação, veremos como resolver este problema com mais detalhe.

O fluxo de dados é o produto de várias instruções que o produzem e que o consomem. Os branches permitem a criação de fluxos de dados dinâmicos, dado que eles conseguem alterar a fonte dos mesmos, podendo, por vezes, tocar em instruções que possuem dependências de dados com muitas outras instruções anteriores. A ordem com que o programa executa as suas instruções é assim um facto determinante para designar que predecessor entregará um valor a cada momento e a ordem do programa é garantida pela manutenção das dependências de controlo. Consideremos, novamente, para o efeito, o Código 2.16.

código 2.16

```
        daddu   r1, r2, r3
        beqz    r4, label1
        dsubu   r1, r5, r6
label1: . . .
        or      r7, r1, r8
```

Como pode ser visto, o valor do registo `r1` usado pela instrução `or` depende no facto do branch ser *taken* ou não. A dependência de dados por si só não é suficiente para preservar a correção, pelo que a dependência de controlo também é necessária. Quanto ao problema da exceção, a especulação que iremos abordar mais à frente, também mostrará uma solução possível, mantendo o fluxo de dados.

Por vezes poderá ser determinado que violar as dependências de controlo não afetará nem o comportamento de exceção, nem o fluxo de dados. Consideremos o Código 2.17, ignorando os efeitos de *delayed branches*.

código 2.17

```
        daddu   r1, r2, r3
        beqz    r12, skip
        dsubu   r4, r5, r6
        daddu   r5, r4, r9
skip:   or      r7, r8, r9
```

Suponhamos assim que é sabido que o registo `r4`, destino para a operação de subtração, não é utilizado depois de `skip` - à propriedade de se um valor é ou não usado por uma instrução seguinte é dado o nome de **liveness**. Então, mudando o valor de `r4` mesmo antes do branch não afetaria o fluxo de dados, sendo que `r4` estaria morto na região de código depois de `skip`.

liveness

Este tipo de agendamento de código é também por si uma forma de especulação, muitas vezes denominado de **especulação por software**.

especulação por software

Técnicas básicas de exposição de paralelismo de instrução (compilação)

De forma a poder manter um pipeline cheio, o paralelismo entre instruções deverá ser explorado de forma a encontrar sequências de instruções não relacionadas e que possam ser sobrepostas no pipeline. Para evitar paragens deste (*stalls*), a execução de uma instrução dependente deverá ser separada da instrução-fonte, por uma distância em ciclos de relógio, pelo menos, igual à latência do pipeline dessa instrução-fonte.

A abilidade do compilador de desempenhar esta tarefa de agendamento depende tanto na quantidade de paralelismo de instrução disponível no programa, como nas latências das unidades funcionais do pipeline.

Como exemplo de aplicação, consideremos como é que o compilador poderá aumentar o número de paralelismo de instrução disponível transformando os ciclos do Código 2.18.

```
for (i = 999; i >= 0; i++) {
  x[i] += s;
}
```

código 2.18

Como podemos ver, no Código 2.18, temos uma situação de ciclo completamente paralelizável: cada iteração é totalmente independente de outra qualquer. Passando este código para linguagem Assembly do MIPS64 teríamos algo, ingenuamente e ignorando *delayed branches*) do género do Código 2.19.

```
loop:   l.d     f0, 0(r1)
        add.d   f4, f0, f2
        s.d     f4, 0(r1)
        daddui  r1, r1, -8
        bne     r1, r2, loop
```

código 2.19

Consideremos o Código 2.19 assumindo que o registo `r1` contém incialmente o endereço do elemento `x[999]` do array e que o conteúdo do registo `r2+8` é o endereço do elemento `x[0]`. Tendo em conta as latências visíveis na tabela da Figura 2.39, podemos verificar o Código 2.19 no Código 2.20.

instrução produzindo resultado	instrução usando resultado	latência em ciclos de relógio
operação da ALU para FP	outra operação da ALU FP	3
operação da ALU para FP	store double	2
load double	operação da ALU para FP	1
load double	store double	0

figura 2.39

```
#                               ciclo de relógio
loop:   l.d     f0, 0(r1)       #       1
        # stall                 #       2
        add.d   f4, f0, f2      #       3
        # stall                 #       4
        # stall                 #       5
        s.d     f4, 0(r1)       #       6
        daddui  r1, r1, -8      #       7
        # stall                 #       8
        bne     r1, r2, loop    #       9
```

código 2.20

O processamento de cada elemento demora 9 ciclos de relógio (assumindo que não há *forwarding* na fase de *instruction decode*).

Reordenando as instruções que não têm dependências entre si, podemos aproveitar o espaço do primeiro *stall* para incluir a instrução `daddui`, perdendo o *stall* referido e o que provinha deste, pela espera para garantir o valor correto para a comparação do branch. No Código 2.21 podemos assim verificar uma nova versão que apenas demora 7 ciclos.

código 2.21

```
        #                                ciclo de relógio
loop:   l.d     f0, 0(r1)       #      1
        daddui  r1, r1, -8      #      2
        add.d   f4, f0, f2      #      3
                # stall         #      4
                # stall         #      5
        s.d     f4, 0(r1)       #      6
        bne     r1, r2, loop    #      7
```

Como cada iteração é completamente independente, podemos transformar o nosso código numa sequência feita de instruções (desmembrando o ciclo), passando a demorar apenas 3.5 ciclos de relógio, como podemos ver no Código 2.22.

código 2.22

```
loop:   l.d     f0, 0(r1)
        l.d     f6, -8(r1)
        l.d     f10, -16(r1)
        l.d     f14, -24(r1)
        add.d   f4, f0, f2
        add.d   f8, f6, f2
        add.d   f12, f10, f2
        add.d   f16, f14, f2
        s.d     f4, 0(r1)
        s.d     f8, -8(r1)
        daddui  r1, r1, -32
        s.d     f12, 16(r1)
        s.d     f16, 8(r1)
        bne     r1, r2, loop
```

Em programas reais, o majorante dos ciclos é raramente conhecido. Assumindo que é n e desmembrando o ciclo k vezes, dois ciclos consecutivos são gerados: o primeiro itera n mod k vezes e tem um corpo semelhante ao do original; o segundo itera n/k vezes e tem o corpo desmembrado. Veja-se um último exemplo no Código 2.23.

código 2.23

```
loop:   l.d     f0, 0(r1)
        add.d   f4, f0, f2
        s.d     f4, 0(r1)
        l.d     f6, -8(r1)
        add.d   f8, f6, f2
        s.d     f8, -8(r1)
        l.d     f10, -16(r1)
        add.d   f12, f10, f2
        s.d     f12, 16(r1)
        l.d     f14, -24(r1)
        add.d   f16, f14, f2
        s.d     f16, 8(r1)
        daddui  r1, r1, -32
        bne     r1, r2, loop
```

Em suma, as decisões e transformações necessárias para desmembrar um ciclo estão enunciadas abaixo [9]:

- determinar se desenrolar o ciclo é realmente útil, procurando o grau de independência entre as iterações;
- usar diferentes registos para evitar restrições desnecessárias que possam ser levantadas pelo uso dos mesmos registos para diferentes cálculos;
- eliminar testes extra e instruções de branch, e ajustamentos de terminações de ciclos e código de iteração;
- determinar se os *loads* e *stores* no ciclo desmembrado podem ser trocados de lugar - há que confirmar os endereços de memória;
- agendar o código, preservando quaisquer dependências necessárias para repetir o mesmo resultado que o código original.

O requisito-chave que reside na base de todos estes que foram referidos é que há a necessidade de se compreender como é qe uma instrução depende de outrae como é que diferentes instruções podem ser alteradas e/ou reordenadas, dadas as dependências.

Contudo, há que saber também que há efeitos que podem limitar os ganhos do desmembramento de ciclos. Em particular existem três efeitos que facilmente podem ser

negligenciados erradamente: uma diminuição da quantidade de *overhead* guardado em cada desenrolamento, como previsto pela Lei de Amdhal; limitações de tamanho de código; e limitações do próprio compilador.

Correlacionar preditores de branch

Como já tivemos oportunidade de verificar os branches são fortes causadores de perdas de desempenho nos pipelines, porque as dependências de controlo geram muitos *hazards* que requerem que o pipeline fique parado em alguns dos seus ciclos de execução (ocorrência de *stalls*). O desenrolar de ciclos é um método possível de reduzir o número de *hazards* de controlo durante a execução, dado que reduzem o número de saltos a realizar. No entanto, o mau desempenho dos branches poderá ser muito mais bem lidado caso estejamos a **prever** o comportamento destes.

prever

Alguns preditores de branch mais simples, que se baseiam tanto na informação em tempo de compilação, como no comportamento dinâmico observado de um branch isolado, já foram estudados. À medida que vamos aumentando o número de instruções a necessidade de haver predições mais precisas torna-se muito maior.

O preditor de 2 bits que estudámos anteriormente toma em consideração o comportamento mais recente de um branch único, de forma a prever o seu comportamento futuro. A precisão desta predição será melhorada se também passarmos a considerar o comportamento mais recente de outros branches mais próximos no programa também.

De modo a termos uma melhor ideia do que estamos a tentar iniciar o estudo, veja-se o Código 2.24 onde os efeitos de *delayed branches* são ignorados.

código 2.24

```
        daddiu   r3, r1, -2
        bnez     r3, l1          ; branch b1 : R1 ≠ 2?
        dadd     r1, r0, r0      ; R1 = 0
l1:     daddiu   r3, r2, -2
        bnez     r3, l2          ; branch b2 : R2 ≠ 2?
        dadd     r2, r0, r0      ; R2 = 0
l2:     dsubu    r3, r1, r2
        beqz     r3, l3          ; branch b3 : R1 = R2?
```

Note-se que o comportamento do branch `b3` está correlacionado com o comportamento dos branches `b1` e `b2`: se ambos forem *untaken*, então `b3` será *taken*. Um preditor que apenas use o comportamento de um único branch para prever o que acontece a seguir nunca poderá captar tal condição.

Preditores de branches que adicionam o comportamento de execução de outros branches, para além do próprio, são denominados de **preditores de correlação**. Um preditor de correlação (1,2), por exemplo, considera o comportamento de execução do branch anterior para escolher entre um par de preditores de branch de 2 bits., ao prever um branch em particular. Em geral, um preditor de correlação (m, n) adiciona o comportamento de m branches anteriores para escolher entre 2^m preditores de branch de n bits, para prever um branch em particular.

preditores de correlação

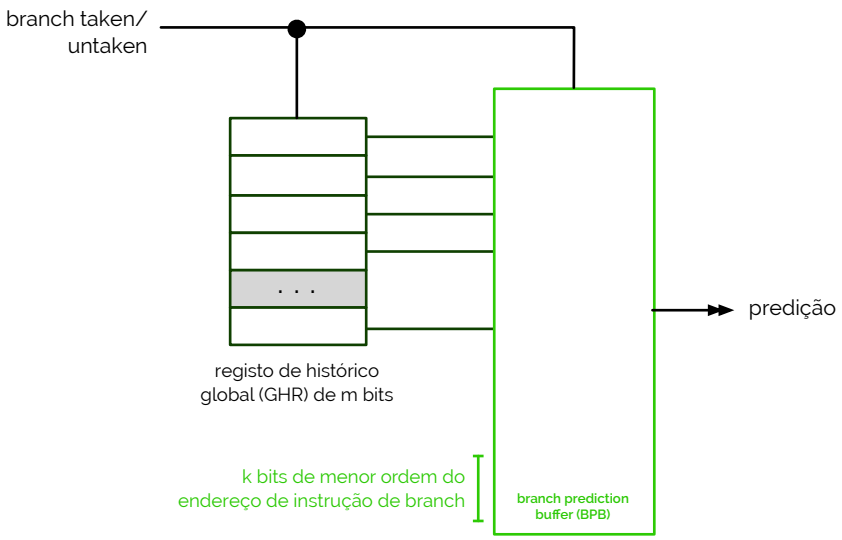
A popularidade deste tipo de preditor de branch é tal que pode resultado numa frequência maior de previsões que o esquema de predição de 2 bits, enquanto requer uma quantidade trivial de *hardware* adicional: o **histórico global** dos m branches mais recentes podem ser gravados num registo de deslocamento de m fases, denominado **global history register** (GHR), onde o conteúdo em cada fase especifica se o branch da ordem correspondente foi *taken* ('1') ou *untaken* ('0'). O *branch prediciton buffer* (BPB) é agora indexado pela concatenação dos m bits do histórico global com os k bits de menor ordem de endereço da instrução de branch. Tendo em conta este cenário, podemos considerar a Figura 2.40, onde se tenta fazer representar um preditor de correlação genérico.

histórico global

global history register

Um exemplo-limite de preditor de correlação será um caso como o de (0,2), onde basicamente estamos a designar um preditor de 2 bits, tal como já abordámos anteriormente, sem qualquer informação acerca do histórico global de branches, pelo que o tamanho do seu *buffer* deverá ser invariante, isto é, $2^m n 2^k$ = constante.

figura 2.40



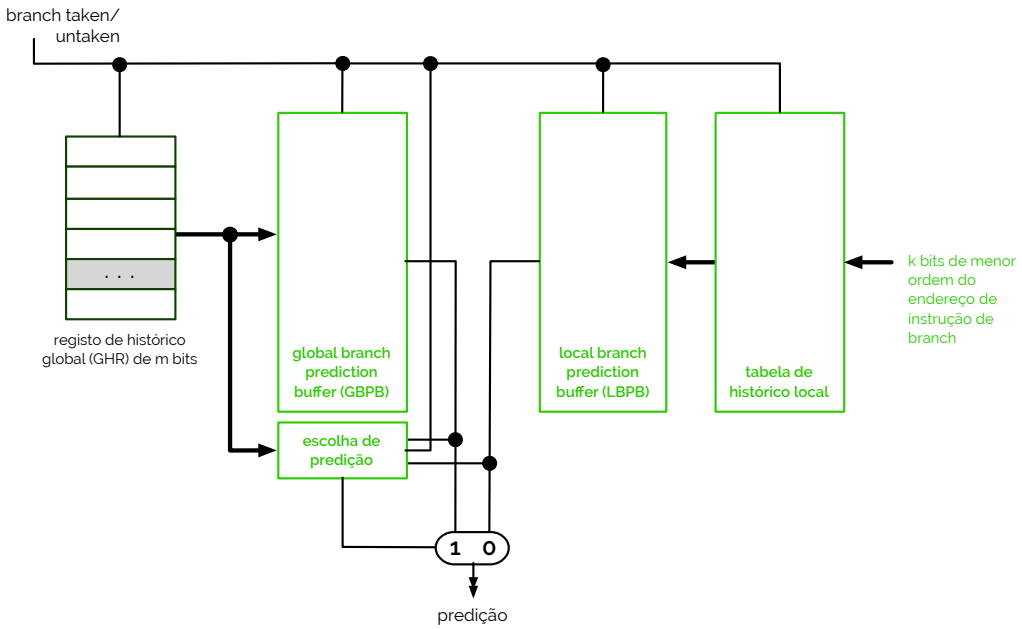
Preditores de torneio

Uma primeira motivação para a correlação de preditores de branches provém da observação de que preditores normais de 2 bits apenas usam informação local, pelo que falham alguns dos branches mais importantes ao longo de todo um programa o que, adicionando um histórico global melhoraria o seu desempenho, tal como vimos na secção anterior. Agora, com os **preditores de torneio** será possível levar a predição para um nível superior, usando múltiplos preditores em simultâneo e sabendo escolher qual é a informação mais acertada. Estes podem, assim, atingir melhores desempenhos no que toca a critérios de eficácia de resposta mesmo com BPB's de tamanho médio e fazer um melhor uso de números maiores de bits para predição.

preditores de torneio

Tendo uma forte resposta perante o histórico de branches, os preditores de torneio existentes já usam contadores de 2 bits com histerese por branch para escolher entre dois outros preditores diferentes (local, global ou um híbrido dos dois), consoante qual teve melhores resultados até ao momento. Por exemplo, um preditor com histerese de 2 bits requer duas más predições para que mude o seu estado com sucesso. Podemos ver um exemplo de preditor de torneio, generalizado, na Figura 2.41.

figura 2.41



A grande vantagem dos preditores de torneio é a escolha que estes fazem entre vários outros preditores para branches em particular, o que é especialmente crucial para programas inteiros. Estatísticas mostram que tipicamente eles selecionam o preditor global cerca de 40% das vezes para o caso de *benchmarks* inteiros e menos de 15% para *benchmarks* em vírgula flutuante. Tanto o *global branch prediction buffer* (GBPB) como a escolha de predição consiste em 2^m preditores de n bits e o *local branch prediction buffer* (LBPB) consiste em 2^u preditores de v bits. A tabela de histórico local, por outro lado, consiste em 2^k registos de histórico local.

Agendamento dinâmico

Uma das grandes limitações das implementações simples de pipeline é o facto de elas usarem instruções em-ordem, tanto para o lançamento, como para a execução de instruções. Se ocorrer uma dependência de dados entre uma instrução no pipeline e uma instrução seguinte, cenário o qual não poderá ser resolvido com técnicas de *forwarding*, a unidade de *interlocking* deverá fazer um *stall* do pipeline, iniciando assim na instrução que usa o resultado. Entretanto nenhuma outra instrução é *fetched* ou lançada até que a dependência seja tratada.

Através do **agendamento dinâmico** podemos ter uma diferente abordagem deste problema: aqui, o *hardware* rearranja a execução da instrução, enquanto mantém um fluxo de dados e comportamento de exceções, de forma a que os *stalls* sejam minimizados. No entanto, isto provoca que haja um maior aumento de complexidade. Existem, contudo, algumas vantagens no uso de agendamento dinâmico, como a permissão de código já compilado para um determinado pipeline (para que corra de forma eficiente nele) de correr noutros pipelines (eliminando a necessidade de haver vários binários), a permissão de execução de código onde algumas dependências que ocorrem em tempo de compilação são desconhecidas (como referências de memória e dependências de dados de branches ou o uso de bibliotecas dinamicamente ligadas), e a permissão do processador autogerir-se em caso de atrasos imprevistos, como *misses* da cache, executando outro código enquanto espera pela resolução do *miss*.

agendamento dinâmico

Consideremos então o segmento de código do Código 2.25.

código 2.25

```
div.d   f0, f2, f4
add.d   f10, f0, f8
sub.d   f12, f8, f14
```

A instrução `sub.d` não poderá ser executada porque a dependência da operação de soma na instrução `div.d` faz com que haja um *stall* no pipeline. Ainda assim, a operação de subtração não tem qualquer dependência de dados para com as duas instruções a si anteriores. A limitação de desempenho criada por este *hazard* poderá ser então eliminada, não requerendo instruções para a execução em-ordem do programa.

Para completar este objetivo, o processo de lançamento poderá ser decomposto em duas partes: verificação de *hazards* estruturais e espera pela ausência de *hazards* de dados. Por conseguinte, o lançamento de instruções em-ordem é ainda usado, mas uma instrução poderá iniciar a execução assim logo que os seus operandos estejam disponíveis. Um pipeline com estas caraterísticas permite execução fora-de-ordem, o que também implica *out-of-order completion*.

A execução fora-de-ordem, como já foi visto anteriormente, introduz, por si, a possibilidade de ocorrerem *hazards* do tipo WAR e WAW, o que não existe no clássico pipeline de cinco andares, muitas vezes estudado. Consideremos o Código 2.26.

código 2.26

```
div.d   f0, f2, f4
add.d   f6, f0, f8
sub.d   f8, f10, f14
mul.d   f6, f10, f8
```

Existe uma antidependência entre as instruções de soma e subtração e uma dependência de saída entre as instruções de soma e de multiplicação. Qualquer um destes *hazards* poderá ser removido se se usar a renomeação de registos.

A *out-of-order completion* também cria complicações maiores no tratamento de exceções. O agendamento dinâmico deve assim preservar o comportamento de exceções num sentido que exatamente as exceções que seriam lançadas se o programa fosse executado de forma estrita, seriam efetivamente lançadas. Tais processadores preservam assim o comportamento da exceção atrasando a notificação de uma exceção associada até que este saiba que a instrução envolvida é a próxima a ser completada.

Embora o comportamento de exceção deva ser preservado, processadores com agendamento dinâmico podem ainda gerar exceções imprecisas, estas, que poderão ocorrer se o pipeline já tiver completado algumas instruções que sucederam, na ordem natural do programa, a instrução causadora da exceção e cujos resultados não podem ser revertidos ou se o pipeline ainda não tiver completado algumas instruções que precederam, na ordem natural do programa, a instrução causadora da exceção.

De forma a permitir a execução fora-de-ordem, a fase de *instruction decode* do clássico pipeline de cinco andares é então dividido em duas fases: fase de lançamento (*issue*) e fase de leitura de operandos. Na fase de lançamento ocorre a descodificação da instrução e a verificação de *hazards* estruturais. Por outro lado, na fase de leitura de operandos, há uma espera pela limpeza de *hazards* de dados antes da leitura de operandos. Num pipeline dinamicamente agendado é possível assim que um conjunto de instruções passe em-ordem pela primeira fase, entrando numa fila para a execução da segunda fase e, nesta, passem a ser executadas fora-de-ordem.

Scoreboarding e algoritmo de Tomasulo

Existem duas técnicas que permitem que as instruções sejam executadas fora-de-ordem quando existem recursos suficientes e não existem depedências de dados entre elas: a técnica de *scoreboarding* (que foi a primeira a ser criada) e o algoritmo de Tomasulo (aplicação com vírgula flutuante).

scoreboarding

O objetivo principal do algoritmo de **scoreboarding** é tentar manter uma frequência de execução de 1 instrução por ciclo de relógio, sendo que não haverá *hazards* estruturais. Por conseguinte, as instruções são executadas o quanto antes possível. Quando um instrução que foi parada por um *stall*, outras instruções na tabela de processamento, que não dependam de nenhuma outra instrução ativa ou estagnada (por *stall*), são procuradas e, se uma for encontrada, então será executada. Aqui, o *scoreboard* tomará controlo total sobre o lançamento da instrução e execução, incluindo toda a deteção de *hazards*.

Num processador com arquitetura MIPS os *scoreboards* fazem, inicialmente, mais sentido para serem aplicados nas unidades de vírgula flutuante, sendo que a latência de outras unidades funcionais é muito baixa. Assume-se, no entanto, que existem dois multiplicadores, um somador, um divisor e uma unidade de cálculo inteiro para todas as referências de memória, branches e operações sobre inteiros. Na Figura 2.42 podemos ver uma generalização de uma aplicação do algoritmo de *scoreboarding*.

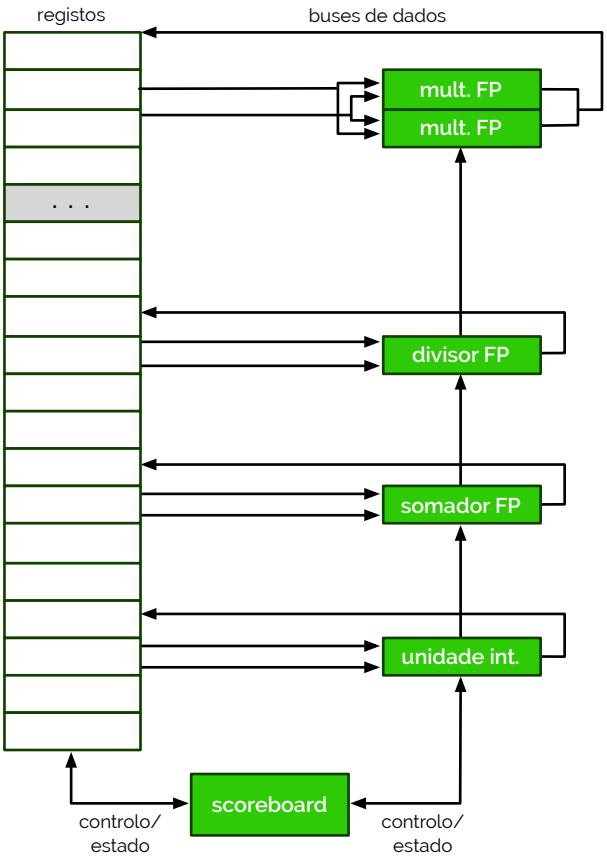
Todas as instruções entram, neste algoritmo, em quatro passos de processamento sob a tutela do controlo do *scoreboard*. Consideremos uma pequena simplificação (em relação à aplicação real) em que o acesso de memória, necessário para as operações de *load* e *store*, não faz parte dos nossos planos. Os quatro passos de processamento, que substituem as fases de pipeline de ID, EXE e WB são o lançamento, a leitura de operandos, a execução e a escrita de resultados.

lançamento

Numa primeira fase deste algoritmo temos então o **lançamento**. Nesta fase, se uma unidade funcional de vírgula flutuante do tipo requirido está livre e mais nenhuma instrução que se encontra ativa tem o mesmo registo como destino, então o *scoreboard* lança uma instrução para a unidade e atualiza a sua estrutura interna. Garantindo que mais nenhuma unidade funcional escreve o seu resultado sobre o registo de destino, assegura-se que nenhum *hazard* do tipo WAW é presente. Caso, porventura, um *hazard* de tal

tipo (ou um estrutural) se aplique, o lançamento de instruções recebe a ordem para fazer *stall* e nenhuma instrução que se siga é lançada até que o problema seja tratado. Quando a fase de lançamento é parada, isto faz com que o *buffer* entre o *fetch* de instruções e lançamento seja todo preenchido e que seja provocado um *stall* para próximos *fetches*.

figura 2.42



Numa segunda fase de processamento leem-se os operandos, sendo que o *scoreboard* monitoriza a disponibilidade dos registos-fonte: um registo-fonte permanece disponível se nenhuma instrução anterior, ainda em execução, lhe vai escrever algum valor. Quando a fonte dos operandos se torna disponível, o *scoreboard*, então, permite que a unidade funcional lhe aceda e leia os valores, iniciando a execução. *Hazards* do tipo RAW acabam por ser dinamicamente resolvidos nesta fase e as instruções poderão ser enviadas para execução, mesmo que fora-de-ordem.

execução

Terminada a fase de leitura de operandos, inicia-se uma fase de **execução**, sendo que nesta a unidade funcional inicia os trabalhos após ter recebido os operandos, já lidos. Quando os resultados estão prontos, este deverá notificar o *scoreboard* que terá completado a sua operação, demorando um número de ciclos de relógio variável.

Uma vez terminada a fase de execução, finalmente, o processador entra numa fase de escrita de resultados (algo equivalente à fase de *write-back* do pipeline clássico de cinco andares). Aqui, o *scoreboard* verifica a ausência *hazards* do tipo WAR e, caso seja necessário, previne a instrução de se completar.

Numa primeira vista, até pode parecer que o *scoreboarding* tem alguma dificuldade a distinguir os *hazards* RAW, dos WAR. Dado que os operandos para uma instrução são lidos apenas quando os conteúdos de ambos registos-fonte têm o seu valor atualizado, o *scoreboard* não tira vantagem do *forwarding*. No fundo, não é uma grande perda, sendo que as instruções escrevem o seu resultado num registo-destino mal acabem a sua execução (desde que não hajam *hazards* do tipo WAR). A consequência desta prática é a diminuída latência e benefícios do *forwarding* atingidos de uma forma mais indireta. Existe, no entanto, um ciclo extra adicionado à latência, porque tanto "escrever um resultado" como "ler um operando" não se poderão sobrepor.

A estrutura de dados interna de um *scoreboard* consiste em três elementos: estado de instrução (é uma tabela com tantas entradas quantas entradas de número de instruções a serem processadas tiver, sendo que, para cada uma, é especificada em qual dos quatro estados é que está), estado da unidade funcional (é uma tabela com tantas entradas quantas unidades funcionais houver) e estado do registo de resultado (é uma entrada única de uma tabela, com tantos campos quantos registos houver no banco de registos, indicando que unidade funcional escreverá onde se uma instrução em ativo tiver um determinado registo como destino). A tabela responsável por anotar o estado das unidades funcionais term nove campos distintos cujos significados são os seguintes:

- *busy* - indica se uma unidade está livre ou ocupada;
- *op* - indica a operação a ser executada numa unidade funcional;
- F_i - número do registo de destino;
- F_j, F_k - números dos registos-fonte;
- Q_j, Q_k - identificação das unidades funcionais cujo resultado deverá ser guardado nos registos fonte F_j, F_k;
- R_j, R_k - *flags* que assinalam quando é que os registos-fonte estão prontos para serem acedidos e ainda não foram lidos.

Para o efeito de demonstração, consideremos o Código 2.27.

```
l.d     f6, 34(r2)
l.d     f2, 45(r3)
mul.d   f0, f2, f4
sub.d   f8, f6, f2
div.d   f10, f0, f6
add.d   f6, f8, f2
```

código 2.27

Assumindo que as operações de *load* possuem uma latência de 1 ciclo de relógio, soma de 2 ciclos, multiplicação de 6 ciclos e divisão de 12 ciclos de relógio, é conveniente denotar que existem dependências de dados (verdadeiras) entre o primeiro *load* e a operação de subtração e divisão, entre a segunda instrução de *load* e a multiplicação, subtração e soma, entre a multiplicação e a divisão e entre a subtração e a soma, que potencialmente poderá levar ao acontecimento de *hazards* do tipo RAW. Existe também uma antidependência entre a instrução de subtração e de divisão e a soma e uma dependência de saída entre a primeira instrução de *load* e a soma, potencialmente levando à ocorrência de *hazards* WAR e WAW, respetivamente.

Na Figura 2.43 podemos ver então a inicialização das tabelas de *scoreboarding*, num estado inicial, em que ainda não foi lançada nenhuma instrução.

figura 2.43

estado de instrução				
instrução	lançamento	leitura de operandos	execução terminada	escrita de resultado
`l.d f6,34(r2)`				
`l.d f2,45(r3)`				
`mul.d f0,f2,f4`				
`sub.d f8,f6,f2`				
`div.d f10,f0,f6`				
`add.d f6,f8,f2`				

estado de unidades funcionais									
nome	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
inteiro	não								
mult1	não								
mult2	não								
soma	não								
divisor	não								

figura 2.43 (continuação)

estado de registos								
F0	F2	F4	F6	F8	F10	F12	...	F30

Quando começamos a trabalhar com o *scoreboarding*, de forma semelhante a tirar pequenas fotografias da evolução da nossa abordagem, podemos ir preenchendo a tabela com a informação relativa aos estados. Por exemplo, consideremos o caso em que a segunda operação de *load* está prestes a escrever o resultado no registo de destino. Verifiquemos assim a Figura 2.44, onde a nível de estados já devemos ter acabado a execução do segundo *load*, sendo que já estamos prontos para escrever o resultado. Assim sendo, nesta fase, e porque o registo destino é operando das duas operações seguintes (multiplicação e subtração), estas apenas são lançadas, em conjunto com uma terceira instrução (instrução de divisão), que também possui uma dependência de saída com a instrução de multiplicação (note-se que o registo destino da instrução de multiplicação é um registo operando da instrução de divisão).

figura 2.44

estado de instrução				
instrução	lançamento	leitura de operandos	execução terminada	escrita de resultado
`l.d f6,34(r2)`	sim	sim	sim	sim
`l.d f2,45(r3)`	sim	sim	sim	
`mul.d f0,f2,f4`	sim			
`sub.d f8,f6,f2`	sim			
`div.d f10,f0,f6`	sim			
`add.d f6,f8,f2`				

estado de unidades funcionais									
nome	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
inteiro	sim	load	F2	R3				não	
mult1	sim	mult	F0	F2	F4	inteiro		não	sim
mult2	não								
soma	sim	sub	F8	F6	F2		inteiro	sim	não
divisor	sim	div	F10	F0	F6	mult1		não	sim

estado de registos								
F0	F2	F4	F6	F8	F10	F12	...	F30
mult1	inteiro			soma	divisão			

Prosseguindo, podemos agora continuar e parar quando a instrução de multiplicação está prestes a escrever o resultado no registo de destino. Neste estado todas as instruções anteriores já cumpriram os seus objetivos, estando agora a aguardar o término da execução as operações de multiplicação e de soma. A operação de soma aguarda também a escrita porque como a de multiplicação não avançou, o registo `r0` não tem o seu valor atualizado para que seja executada a instrução de divisão que, por si, também usa o valor de `f6`, registo o qual é destino da operação de soma. No entanto, porque as instruções podem ser executadas fora-de-ordem, a instrução de subtração já terminou a sua execução e já colocou, inclusive, o seu valor no registo correspondente - o registo `f8`. Tal aconteceu porque esse registo não é usado antes e, dos que são, `f2` já tem o valor correto, carregado pelo segundo *load* do programa. Como estamos apenas a lidar com instruções cuja precisão é dupla, como podemos esperar, só as unidades de vírgula flutuante é que estão a funcionar, sendo que, neste ciclo, só iremos esperar pelo resultado da multiplicação em `f0`, pelo que Q_j para a unidade de divisão está preenchido com o registo `f0`. Esta tabela pode ser vista na Figura 2.45.

figura 2.45

estado de instrução				
instrução	lançamento	leitura de operandos	execução terminada	escrita de resultado
`l.d f6,34(r2)`	sim	sim	sim	sim
`l.d f2,45(r3)`	sim	sim	sim	sim
`mul.d f0,f2,f4`	sim	sim	sim	
`sub.d f8,f6,f2`	sim	sim	sim	sim
`div.d f10,f0,f6`	sim			
`add.d f6,f8,f2`	sim	sim		

estado de unidades funcionais									
nome	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
inteiro	não								
mult1	sim	mult	F0	F2	F4			não	não
mult2	não								
soma	sim	add	F6	F8	F2			não	não
divisor	sim	div	F10	F0	F6	mult1		não	sim

estado de registos								
F0	F2	F4	F6	F8	F10	F12	...	F30
mult1			soma		divisão			

Como a instrução de divisão é a que mais demora em termos de execução, também será esta a última a terminar a sua execução. Assim sendo, tentemos capturar, na Figura 2.46, os seus estados.

figura 2.46

estado de instrução				
instrução	lançamento	leitura de operandos	execução terminada	escrita de resultado
`l.d f6,34(r2)`	sim	sim	sim	sim
`l.d f2,45(r3)`	sim	sim	sim	sim
`mul.d f0,f2,f4`	sim	sim	sim	sim
`sub.d f8,f6,f2`	sim	sim	sim	sim
`div.d f10,f0,f6`	sim	sim	sim	
`add.d f6,f8,f2`	sim	sim	sim	sim

estado de unidades funcionais									
nome	busy	op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
inteiro	não								
mult1	não								
mult2	não								
soma	não								
divisor	sim	div	F10	F0	F6	mult1		não	não

estado de registos								
F0	F2	F4	F6	F8	F10	F12	...	F30
					divisão			

Analisando a Figura 2.46 podemos então ver que quando a operação de divisão está mesmo prestes a terminar a sua execução, escrevendo o resultado no registo `f10`, todas as outras instruções já terminaram a sua execução. Em suma, na Figura 2.47, podemos ver uma tabela onde estão presentes, para cada estado, as verificações e atualizações necessárias para as transições do *scoreboard*. Nesta figura, note-se que a sigla FU quer sig-

nificar a unidade funcional associada com a instrução, SR1 e SR2 são os registos-fonte e DR é o registo-destino. O teste na fase de escrita de resultado é transportado para fora para previnir *hazards* do tipo WAR.

figura 2.47

estado	verificações	atualizações

lançamento	(busy[FU] == não) && (regResult[DR] == vazio)	busy[FU] = sim;
		op[FU] = op;
		F_i[FU] = DR;
		F_j[FU] = SR1;
		F_k[FU] = SR2;
		Q_j[FU] = regResult[SR1];
		Q_k[FU] = regResult[SR2];
		R_j[FU] = (Q_j[FU] == vazio) ? sim : não;
		R_k[FU] = (Q_k[FU] == vazio) ? sim : não;
		regResult[DR] = FU;

leitura de operandos	(R_j[FU] == sim) && (R_jkFU] == sim)	Q_j[FU] = vazio; Q_k[FU] = vazio; R_j[FU] = não; R_k[FU] = não;
execução	operação da unidade funcional terminou	-
escrita de resultados	∀f ((F_j[f] != F_i[FU]) || (R_j[f] == não)) && ((F_k[f] != F_i[FU]) || (R_k[f] == não))	∀f **se** (Q_j[f] == FU) R_j[f] = sim; ∀f **se** (Q_k[f] == FU) R_k[f] = sim; regResult[F_i[FU]] = vazio; busy[FU] = não;

Em suma, o *scoreboard* usa o paralelismo de instruções disponível para minimizar o número de *stalls* provenientes de dependências de dados do programa. Na eliminação de *stalls*, um *scoreboard* fica limitado por vários fatores, como a quantidade de paralelismo inerente ao programa - este fator determina se instruções independentes poderão ser encontradas, sendo que se todas as instruções dependerem do seu predecessor, não poderá ser aplicado nenhum esquema de agendamento dinâmico que vá reduzir o número de *stalls* -, o número de entradas no *scoreboard* - este fator determina o quão longe poderá o pipeline olhar, à busca de instruções independentes -, o número e tipos das unidades funcionais - este fator determina o quão relevante são os *hazards* estruturais, o que poderá aumentar quando o agendamento dinâmico é usado -, e a presença de antidependências e dependências de saída - o que leva a *hazards* do tipo WAR e WAW, o que poderá gerar ainda mais *stalls*.

Até agora abordámos o algoritmo de *scoreboarding* e, quando o introduzimos, referimos que haveria um melhor, denominado de **algoritmo de Tomasulo**. A grande diferença entre eles é que o algoritmo de Tomasulo trabalha e trata antidependências e dependências de saída renomeando, dinamicamente, os registos. Adicionalmente, também poderá ser extendido para tratar a especulação, algo que veremos um pouco mais à frente e que é uma técnica cujo objetivo é reduzir o efeito das dependências de controlo, prevendo o resultado de um branch através da execução de instruções no endereço do alvo e tomando ações corretas quando a predição está errada.

algoritmo de Tomasulo

Sendo um esquema desenvolvido por Robert Tomasulo, aqui os *hazards* do tipo RAW são evitados pela execução de uma instrução só quando os seus operandos estão disponíveis, o que é exatamente o que o *scoreboard* mais simples permite. No entanto, *hazards* do tipo WAR ou WAW, que são lançados através de dependências de nome, são eliminadas através da renomeação de registos. A **renomeação de registos**, de facto, elimina os *hazards* alterando simplesmente o nome de todos os registos de destino, incluindo aqueles que têm uma leitura ou escrita pendentes de uma instrução anterior, tal que escritas fora-de-ordem não afetam quaisquer instruções que possam depender num valor mais antigo do operando.

⊙ **Robert Tomasulo**

renomeação de registos

Consideremos assim o Código 2.28, de forma a que possamos perceber melhor como é que a renomeação de registos funciona. Neste código existem antidependências entre as instruções de soma e de subtração e entre as instruções de *store* e multiplicação, tal como uma dependência de saída entre as instruções de soma e multiplicação. Esta última dependência, poderá, potencialmente, levar a *hazards* do tipo WAR ou WAW, respetiva-

mente. Ainda existem outras dependências de dados entre as instruções de divisão e de soma, entre a soma e o *store* e entre a subtração e a multiplicação.

código 2.27

```
div.d   f0, f2, f4
add.d   f6, f0, f8
s.d     f6, 0(r1)
sub.d   f8, f10, f14
mul.d   f6, f10, f8
```

As três dependências de nome que existem no Código 2.27 poderão ser todas eliminadas através de renomeação de registos. Para nossa simplicidade, assumamos a existência de dois registos temporários, de nomes `s` e `t`. O Código 2.28 mostra assim a sequência do Código 2.27 re-escrita de forma a não mais haver dependências de nome.

código 2.28

```
div.d   f0, f2, f4
add.d   s, f0, f8
s.d     s, 0(r1)
sub.d   t, f10, f14
mul.d   f6, f10, t
```

Mais, qualquer uso subsequente do registo `f8` deverá ser substituído por `t`. Neste segmento de código o processo de renomeação pode ser feito estaticamente pelo compilador. Encontrando todos os usos do registo `f8` que possam aparecer ao longo de um código, requer uma análise mais sofisticada, por parte do compilador, ou até mesmo um suporte diferente de *hardware*, sendo que poderão haver branches que possam intervir entre o segmento de código e um uso mais tardio de `f8`.

No esquema de Tomasulo a renomeação de registos é fornecida pelas **estações de reserva** (do inglês *reservation stations*), que preservam os operandos das instruções à espera de serem lançadas. A ideia por debaixo de todo esta lógica é que uma estação de reserva captura e guarda um operando quão cedo possível, eliminando a necessidade de obter o operando de um registo do banco de registos. Instruções pendentes designam o *buffer* de reserva que lhes fornecerá as suas entradas (os seus *inputs*). Finalmente, quando escritas sucessivas para um mesmo registo se sobrepõem em execução, somente a última em-ordem do programa é que irá atualizar o registo. As instruções são lançadas e os especificadores de registos para operandos pendentes são renomeados para os nomes das estações de reserva que fornecem os valores atualizados.

estações de reserva

Como podem haver mais estações de reserva que registos reais, esta aproximação poderá até eliminar *hazards* lançados através de dependências de nomes que não foram devidamente tratadas pelo compilador.

O uso de estações de reserva, ao invés de um banco de registos centralizado, leva-nos para duas propriedades muito importantes: a deteção de *hazards* e o seu controlo são distribuídos, e os resultados são passados diretamente para as unidades funcionais através das estações de reserva, onde são carregadas, ao invés de passarem através de registos do banco de registos. A organização geral deste algoritmo está então patente na Figura 2.48 [8]. Note-se que cada estação de reserva sustenta uma instrução que já fora lançada e que espera por execução na unidade funcional associada. Aqui nem os valores dos operandos esperam por tal instrução, nem estão disponíveis, mesmo os nomes das estações de reserva que irão fornecer os valores do operando após a sua computação.

Os *buffers* de carregamento e de transferência sustentam tanto endereços como dados transferidos da/para a memória e comportam-se quase como as estações de reserva, pelo que serão apenas distinguidos somente quando necessário. Em particular, tais *buffers* têm três funções: carregam o endereço da fonte ou destino, por completo, depois do seu cálculo, pela unidade de endereçamento; acompanham grandes carregamentos que esperam pela resolução de uma transferência; controlam os resultados de carregamentos completos que estão pendentes da disponibilidade do **bus de dados comum** (CDB), ou aguentam os valores a ser guardados até que a unidade de memória esteja disponível.

bus de dados comum

Todos os resultados das unidades funcionais e de memória são colocados no CDB, o qual liga tudo, exceto os *buffers* de carregamento. Todas as estações de reserva e *buffers*

de carregamento e transferência têm campos de legenda (*tag fields*) ligados pelo controlo do pipeline.

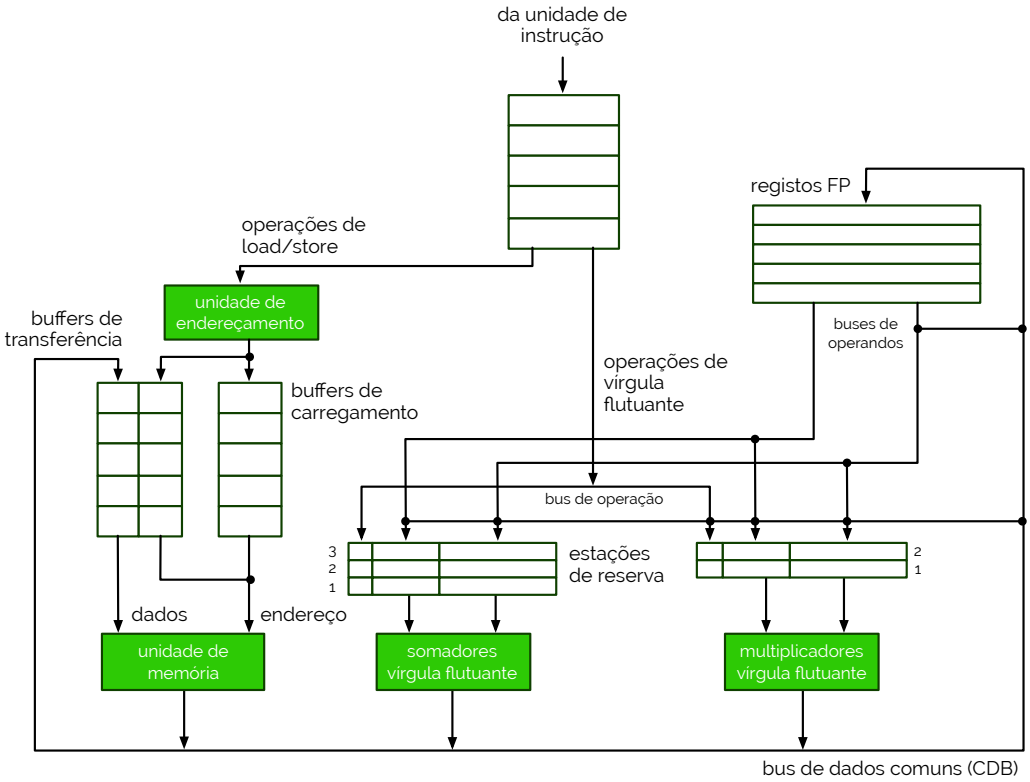


figura 2.48

Todas as instruções passam por três fases de processamento com o algoritmo de Tomasulo, embora cada uma possa demorar um número arbitrário de ciclos de relógio. As três fases, que substituem as fases de *instruction decode*, *execute* e *write-back* num pipeline regular, são primeiramente um lançamento, depois a execução e finalmente a escrita de resultados.

lançamento

Na primeira fase, fase de **lançamento**, a próxima instrução é obtida através da cabeça da fila de instrução, que é mantida com uma ordem FIFO para assegurar um fluxo de dados correto. Se uma estação ou *buffer* que encaixe na instrução estiver livre, enrão a instrução é lançada para a estação ou *buffer* juntamente com os valores dos operandos, se estes já estiverem guardados num ou mais registos do banco de registos. Por outro lado, um acompanhamento das unidades funcionais que irão produzir os valores dos operandos é desempenhado. É aqui que os registos são renomeados, eliminando a consequência de *hazards* do tipo WAR e WAW. Esta fase é por vezes denominada de *dispatch*, dado o contexto dos processadores de agendamento dinâmico.

execução

Tendo a primeira fase concluída passamos para a **execução**, sendo que se um ou mais operandos não estão disponíveis, então o bus comum de dados é monitorizado para esperar por eles ou até mesmo pelo cálculo destes. Logo que um operando se torna disponível, este é guardado tanto no banco de registos, como em qualquer estação de reserva que o requira. Quando todos os operandos estão disponíveis, a operação pode então ser executada na unidade funcional ou de memória correspondente. Atrasando a execução da instrução até que todos os operandos estejam prontos, os *hazards* do tipo RAW estão a ser evitados.

Muitas instruções, por outro lado, poderão tornar-se prontas no mesmo ciclo de relógio. Embora as unidades funcionais independentes possam iniciar a execução no mesmo ciclo de relógio, se mais do que uma instrução está pronta para executar na mesma unidade funcional, então uma seleção deverá ser realizada entre elas. Por exemplo, as estações de reserva de vírgula flutuante poderão ser escolhidas arbitrariamente.

As operações de *load* e de *store*, no entanto, são mantidas em-ordem do programa por todo o uso de uma fila de carregamentos/transferências que contém a identificação do

buffer que está a ser usado. A sua execução requer um processo em dois passos: num primeiro passo o endereço efetivo terá de ser calculado; num passo segundo, o acesso à memória para a transferência de dados terá de ser executada.

Para preservar o comportamento de exceções, nenhuma instrução está permitida de iniciar a execução antes que todos os branches que lhes precedem em-ordem do programa estejam completados. Esta restrição garante que uma instrução que causa uma exceção durante a execução terá sido realmente executada.

Finalmente, numa terceira fase, ocorre a **escrita de resultados**. Aqui, quando o resultado de uma operação está disponível, o CDB é acedido para guardar o valor num registo do banco de registos e numa estação de reserva, incluindo *buffers* de transferência (salvaguarda), que o esperam.

escrita de resultados

As estruturas de dados que detetam e eliminam *hazards* estão ligadas às estações de reserva, ao banco de registos e aos *buffers* de carregamento e de transferência, com a pequena diferença de que as informações são distintas entre as várias ligações. Estas etiquetas (*tags*) são essencialmente nomes para o conjunto extendido de registos virtuais usados no processo de renomeação.

Na ilustração da Figura 2.48, o campo da etiqueta é uma quantidade de 4 bits que denora uma de cinco estações de reserva (três somadores de vírgula flutuante para adições e subtrações e dois multiplicadores para multiplicações e divisões em vírgula flutuante) ou um de cinco *buffers* de carregamento. Isto produz o equivalente a dez registos que poderão ser designados de **registos-resultado**.

registos-resultado

No esquema de Tomasulo, os resultados são propagados num bus que é monitorizado pelas estações de reserva e *buffers* de transferência (*store buffers*) para recolha de dados. Este tipo de arranjo implementa mecanismos de *forwarding* e de *bypassing* usados num pipeline estaticamente agendado. Fazendo isto, no entanto, um esquema dinamicamente agendado adiciona um ciclo de relógio de latência entre a fonte e o resultado, sendo que a correspondência do resultado mais o seu uso não pode ser concluída antes da fase de escrita de resultados. Por conseguinte, num pipeline com agendamento dinâmico, a latência efetiva entre uma instrução produtora e uma consumidora é de, pelo menos, um ciclo mais que a latência da unidade funcional que produz o resultado.

É importante considerar que as etiquetas no algoritmo de Tomasulo referem-se à unidade ou ao *buffer* que produz tal resultado: os nomes dos registos serão descartados uma vez que a instrução tenha sido lançada. Isto é, o elemento-chave de diferença entre o esquema de Tomasulo e o de *scoreboarding*, é que no segundo, os operandos permanecem nos registos do banco de registos e são apenas lidos após as instruções de produção serem completadas e as instruções de consumo estarem prontas a executar.

Cada estação de reserva e *buffer* tem sete campos, entre os quais:

- *busy* - indica se a estação de reserva ou *buffer* está ocupado ou livre;
- *op* - indica a operação que está a ser desempenhada na unidade;
- Q_j, Q_k - identificação da estação de reserva, ou *buffer*, que produzirá o operando-fonte correspondente. Um valor de zero indica que o operando-fonte já está disponível nos registos V_j e V_k ou é desnecessário;
- V_j, V_k - o valor dos operandos-fonte. Note-se que somente um dos campos Q ou V é que é válido para cada um dos operandos, sendo que para *buffers* de carregamento (*load buffers*), o campo V_k é usado para preservar o campo de deslocamento (em inglês *offset field*);
- A - usado para guardar o endereço efetivo de memória para uma instrução de *load* ou de *store*.

Cada registo do banco de registos ainda tem um campo específico:

- Q_i - identificação da estação de reserva, ou *buffer*, que irá produzir o resultado a ser guardado num registo. Um valor de zero indica que nenhuma instrução ativa está a calcular o valor a ser guardado em tal registo.

Consideremos, então, para o efeito de demonstração, o Código 2.29.

código 2.29

```
l.d     f6, 32(r2)
l.d     f2, 44(r3)
mul.d   f0, f2, f4
sub.d   f8, f2, f6
div.d   f10, f0, f6
add.d   f6, f8, f2
```

Qual é o estado de execução quando somente a primeira instrução de *load* foi completada e escrito o seu resultado? Assumindo as latências de 1 ciclo para *load*/*store*, 2 ciclos para adições, 6 ciclos para multiplicações e 12 ciclos para divisões, vejamos o conteúdo das tabelas da Figura 2.49.

figura 2.49

estado de instrução (não faz parte do hardware)			
instrução	**lançamento**	**execução**	**escrita de resultado**
`l.d f6,32(r2)`	sim	sim	sim
`l.d f2,44(r3)`	sim	sim	
`mul.d f0,f2,f4`	sim		
`sub.d f8,f2,f6`	sim		
`div.d f10,f0,f6`	sim		
`add.d f6,f8,f2`	sim		

estações de reserva / buffers							
nome	**busy**	**op**	**V_j**	**V_k**	**Q_j**	**Q_k**	**A**
load1	não						
load2	sim	load					44+reg[r3]
add1	sim	sub		mem[32+reg[r2]]	load2		
add2	sim	add			add1	load2	
add3	não						
mult1	sim	mult		reg[f4]	load2		
mult2	sim	div		mem[32+reg[r2]]	mult1		

estado de registos									
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Q_i	mult1	load2		add2	add1	mult2			

Vejamos agora, os estados das nossas estações de reserva e *buffers*, para quando a instrução de multiplicação está pronta a escrever o seu resultado, na Figura 2.50.

figura 2.50

estado de instrução (não faz parte do hardware)			
instrução	**lançamento**	**execução**	**escrita de resultado**
`l.d f6,32(r2)`	sim	sim	sim
`l.d f2,44(r3)`	sim	sim	sim
`mul.d f0,f2,f4`	sim	sim	
`sub.d f8,f2,f6`	sim	sim	sim
`div.d f10,f0,f6`	sim		
`add.d f6,f8,f2`	sim	sim	sim

figura 2.50 (continuação)

estações de reserva / buffers							
nome	busy	op	V_j	V_k	Q_j	Q_k	A
load1	não						
load2	não						
add1	não						
add2	não						
add3	não						
mult1	sim	mult	mem[44+reg[r3]]	reg[f4]	load2		
mult2	sim	div		mem[32+reg[r2]]	mult1		

estado de registos									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
Q_i	mult1					mult2			

Em suma, na Figura 2.51 temos uma tabela onde, para cada estado de instrução temos um critério de espera e uma atualização definida, onde RS[r] é a estação de reserva / *buffer* associado com a instrução e *regStat* é o estado do registo.

figura 2.51

estado	critério de espera	atualizações

lançamento de uma operação FP	(RS[r].busy == não)	**se** (regStat[rs].Q_i != 0) {
		  RS[r].Q_j = regStat[rs].Q_i
		  caso contrário {
		  RS[r].V_j = reg[rs];
		  RS[r].Q_j = 0;
		  }
		}
		se (regStat[rs].Q_i != 0) {
		  RS[r].Q_k = regStat[rt].Q_i
		  caso contrário {
		  RS[r].V_k = reg[rt];
		  RS[r].Q_k = 0;
		  }
		}
		regStat[rd].Q_i = r;
		RS[r].busy = sim;

lançamento de um load	(RS[r].busy == não)	**se** (regStat[rs].Q_i != 0) {
		  RS[r].Q_j = regStat[rs].Q_i
		  caso contrário {
		  RS[r].V_j = reg[rs];
		  RS[r].Q_j = 0;
		  }
		}
		RS[r].A = imm;
		regStat[rt].Q_i = r;
		inserir r **em** fila load-store;
		RS[r].busy = sim;

lançamento de um store	(RS[r].busy == não)	**se** (regStat[rs].Q_i != 0) {
		  RS[r].Q_j = regStat[rs].Q_i
		  caso contrário {
		  RS[r].V_j = reg[rs];
		  RS[r].Q_j = 0;
		  }
		}
		RS[r].A = imm;
		se (regStat[rs].Q_i != 0) {
		  RS[r].Q_k = regStat[rt].Q_i
		  caso contrário {
		  RS[r].V_k = reg[rt];
		  RS[r].Q_k = 0;
		  }
		}
		inserir r **em** fila load-store;
		RS[r].busy = sim;

execução de operação FP	(RS[r].Q_j == 0) && (RS[r].Q_k == 0)	**calcular** resultado - operandos em V_j e V_k

estado	critério de espera	atualizações
execução de load/store (passo 1)	(RS[r].Q_j == 0) && r é cabeça da fila load-store	RS[r].A = RS[r].V_j + RS[A].A;
execução de load (passo 2)	passo 1 de load está completo	**ler de** mem[RS[r].A]

escrever resultado operação FP ou load	r completou execução && CDB está disponível	∀x (**se** (regStat[x].Q_i == r) {
		reg[x] = resultado;
		regStat[x].Q_i = 0;
		})
		∀x (**se** (RS[x].Q_j == r) {
		RS[x].V_j = resultado;
		RS[x].Q_j = 0;
		})
		∀x (**se** (RS[x].Q_k == r) {
		RS[x].V_k = resultado;
		RS[x].Q_k = 0;
		})
		RS[r].busy = não

escrever resultado store	r completou execução && (RS[r].Q_k == 0)	mem[RS[r].A] = RS[r].V_k; RS[r].busy = não;

Por fim, consideremos um último caso, do Código 2.30.

```
loop:   l.d     f0, 0(r1)
        mul.d   f4, f0, f2
        s.d     f4, 0(r1)
        daddiu  r1, r1, -8
        bne     r1, r2, loop
```

código 2.30

Um ciclo é considerado no Código 2.30 para ilustrar o potencial total de eliminação de *hazards* WAW e WAR através da renomeação de registos (consideremos os efeitos de *delayed branch* desprezíveis). Assume-se, também, que o registo `r1` inicialmente contém o endereço do último elemento de um array e que o conteúdo do registo `r2+8` é o endereço do primeiro elemento do array.

Se o branch previr *taken*, múltiplas execuções do ciclo irão decorrer em paralelo através do uso de múltiplas estações de reserva e de *buffers* de *load* e *store*. Esta funcionalidade é ganha sem que qualquer código, desde o ciclo, seja desmembrado dinamicamente pelo *hardware*. As estações de reserva e os *buffers* conduzem agora o papel de registos adicionais. Consideremos então as mesmas latências de anteriormente e vejamos a tabela da Figura 2.52, onde se veem duas iterações sucessivas de um ciclo a serem lançadas sem que nenhuma instrução se tivesse completado.

estado de instrução (não faz parte do hardware)				
instrução	**iteração**	**lançamento**	**execução**	**escrita de resultado**
`l.d f0,0(r1)`	i	sim	sim	
`mul.d f4,f0,f2`	i	sim		
`s.d f4,0(r1)`	i	sim		
`l.d f0,0(r1)`	i+1	sim	sim	
`mul.d f4,f0,f2`	i+1	sim		
`s.d f4,0(r1)`	i+1	sim		

figura 2.52

Quando extendido para lançamento de instruções múltiplas, a aproximação de Tomasulo pode suster mais que uma instrução por ciclo de relógio. *Loads* e *stores* podem assim, de forma segura, ser desempenhados fora-de-ordem, tendo eles acesso a diferentes endereços de memória. Quando eles referem o mesmo endereço tanto pode o *load* ser antes do *store* e mudarem-se os resultados num *hazard* WAR, ou o *load* é feito depois de um *store* em-ordem e mudarem-se os resultados num *hazard* RAW.

figura 2.52 (continuação)

estações de reserva / buffers							
nome	busy	op	V_j	V_k	Q_j	Q_k	A
load1	sim	load					reg[r1]+0
load2	sim	load					reg[r1]-8
mult1	sim	mult		reg[f2]	load1		
mult2	sim	mult		reg[f2]	load2		
store1	sim	store	reg[r1] + 0			mult1	
store2	sim	store	reg[r1] - 8			mult2	

estado de registos									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
Q_i	load2		mult2						

De forma semelhante, intercambiar a execução de dois *stores* para a mesma região de memória resulta num *hazard* do tipo WAW.

Assim sendo, para determinar se uma instrução de *load* pode ou não ser executada, o processador deverá verificar se tem alguma instrução de *store* por completar e que preceda o *load* em-ordem do programa, acedendo ao mesmo endereço de memória. Similarmente, uma instrução de *store* deverá aguardar até que não exista nenhuns *loads* ou *stores* inesperados que lhe precedam, acedendo ao mesmo endereço de memória.

Para previnir que estes *hazards* de dados ocorram, o processador deverá já ter calculado os endereços efetivos associados com qualquer operação recente realizada sobre a memória (que ainda esteja em progresso). Isto poderá ser feito de uma forma simples, mas não necessariamente ótima, desempenhando todos os cálculos para os endereços de memória efetivos em-ordem, no programa que está em execução. Se uma operação de *load* aparecer primeiro, a existência de um conflito de endereços poderá ser verificada rapidamente examinando o campo *A* de todos os *buffers* de transferência (de *stores*) ativos. Sendo detetado um conflito, a instrução de *load* não será lançada até que seja tratado o problema.

No caso de instruções de *store*, o procedimento é semelhante, à exceção de que o processador deverá também verificar a presença de conflitos entre os *buffers* de *load* e de *store*, dado que os *stores* conflituosos não poderão ser reordenados, com respeito tanto a *loads* como a *stores*.

Preditores de correlação revisitados

Como vimos, um preditor de correlação (m,n) adiciona o comportamento de exceções dos m branches anteriores para escolher entre as previsões de 2^m preditores para um branch em particular. Um preditor local de 2 níveis funciona da mesma maneira, apenas acompanhando do comportamento passado de um branch individual para prever o seu comportamento futuro.

Existe, assim, um *trade-off* entre tais preditores: os preditores de correalação necessitam de uma pequena memória para o seu histórico global (que lhe permite manter preditores de 2 bits para um largo número de branches diferentes), enquanto que preditores locais necessitam substancialmente de mais memória para o seu histórico local, daí estarem limitados a acompanhar um número relativamente pequeno de grupos de instruções de branch.

Consideremos, portanto, um preditor de correlação (1,2) que poderá acompanhar quatro branches, requirindo 16 bits de memória, *versus* um preditor local (1,2) que poderá acompanhar dois branches usando a mesma quantidade de memória. Para os resultados dos branches listados na tabela da Figura 2.53, tentemos fornecer uma predição, criando uma tabela onde as vamos gerir, com atualizações, resultados e, no final, calculemos o rácio de falhas de predição [10].

endereço do branch (em hexadecimal)	resultado
1C6	taken
21F	not taken
309	not taken
21F	not taken
309	not taken
1C6	taken
309	not taken
1C6	taken
21F	taken

figura 2.53

Para podermos criar a nossa tabela, incializemos o preditor de branches de correlação com os valores da tabela da Figura 2.54, sendo que consideramos que o registo de histórico global guardou *not taken*, da última vez.

entrada	grupo de branch	último resultado	estado interno	predição
0	0	not taken	0	not taken
1	0	taken	2	taken
2	1	not taken	3	taken
3	1	taken	0	not taken
4	2	not taken	3	taken
5	2	taken	3	taken
6	3	not taken	0	not taken
7	3	taken	1	not taken

figura 2.54

Tendo a tabela de inicialização na Figura 2.54 podemos agora tentar completar uma tabela semelhante à da Figura 2.55.

endereço de branch	resultado	registo de histórico global (GHR)	entrada	predição	atualização de predição
		not taken			

figura 2.55

Para completarmos a tabela precisamos ainda de contextualizar os nossos branches. Para tal teremos de olhar para os k bits de menor ordem conforme vimos na Figura 2.40. Assim sendo, os últimos bits do branch 1C6 são "10", do branch 21F são "11" e do branch 309 são "01". Se a estes bits juntarmos a possibilidade de cada um ser *taken* ou *not taken*, então teremos que o branch 1C6 fica codificado com "100" e "101" (entradas 4 e 5), o branch 21F fica codificado com "110" e "111" (entradas 6 e 7) e o branch 309 fica codificado com "010" e "011" (entradas 2 e 3).

Comecemos então por olhar para a linha da tabela da Figura 2.55 e nela inserir o endereço de branch 1C6, cuja entrada é a 4 (dado que está *not taken*). Sendo entrada 4, vamos agora à tabela da Figura 2.54 e verificar que a entrada 4 tem predição *taken* (a qual inserimos na nossa linha) e para a qual esperávamos uma predição *taken* - conforme a Figura 2.54. Agora, para fazer a atualização, devemos então olhar novamente para a Figura 2.17, onde reparamos que tendo estado 3, isto é, "11" (que vemos da tabela da Figura 2.54), com resultado *taken*, voltamos a ficar em 3, isto é, a atualização será 3. Ficamos então com uma linha com o aspeto da da Figura 2.56.

endereço de branch	resultado	registo de histórico global (GHR)	entrada	predição	atualização de predição
1C6	taken	not taken	4	taken	3

figura 2.56

Agora, passamos para a linha seguinte, onde preenchemos inicialmente o registo de histórico global com o resultado efetivo anterior de predição (o resultado *taken*). Esta será a instrução 21F, pelo que devemos pesquisar pela entrada "111" (21F e *taken*) na tabela da Figura 2.54. Esta entrada, a 7, tem então uma predição de *not taken* para uma espetável de *not taken*. Com este resultado, estando em "01" (estado interno de 7), passamos para "00", pelo que preenchemos uma atualização de 0. Veja-se a Figura 2.57.

endereço de branch	resultado	registo de histórico global (GHR)	entrada	predição	atualização de predição
1C6	taken	not taken	4	taken	3
21F	not taken	taken	7	not taken	0

figura 2.57

Mais uma vez, e porque o último resultado efetivo de predição foi *not taken*, o registo de histórico global passa para *not taken* também, aquando da execução do branch 309. Este branch, codificado com o resultado para "010" tem entrada em 2, onde, tendo uma predição de *taken*, possui um estado de 3 que, com o resultado de *not taken* (vide Figura 2.53) muda para 2. Ficamos assim com a Figura 2.58.

endereço de branch	resultado	registo de histórico global (GHR)	entrada	predição	atualização de predição
1C6	taken	not taken	4	taken	3
21F	not taken	taken	7	not taken	0
309	not taken	not taken	2	taken	2

figura 2.58

Prosseguindo com o mesmo algoritmo, temos que chegamos a uma tabela como a da Figura 2.59.

endereço de branch	resultado	registo de histórico global (GHR)	entrada	predição	atualização de predição
1C6	taken	not taken	4	taken	3
21F	not taken	taken	7	not taken	0
309	not taken	not taken	2	taken	2
21F	not taken	not taken	6	not taken	0
309	not taken	not taken	2	taken	2
1C6	taken	not taken	4	taken	3
309	not taken	taken	3	not taken	0
1C6	taken	not taken	4	taken	3
21F	taken	taken	7	not taken	3

figura 2.59

Note-se, no entanto, que houve erros na predição - Figura 2.60.

endereço de branch	resultado	registo de histórico global (GHR)	entrada	predição	atualização de predição
1C6	taken	not taken	4	taken	3
21F	not taken	taken	7	not taken	0
309	not taken	not taken	2	taken	2
21F	not taken	not taken	6	not taken	0
309	not taken	not taken	2	taken	2
1C6	taken	not taken	4	taken	3
309	not taken	taken	3	not taken	0
1C6	taken	not taken	4	taken	3
21F	taken	taken	7	not taken	3

figura 2.60

Os erros são assim detetados porque a predição não coincidiu com o resultado obtido. Tendo havido 3 erros, então o rácio de perdas é de 1/3.

Consideremos agora o preditor local, com uma inicialização visível na tabela da Figura 2.61 e com os últimos dois branches locais a terem resultado *taken* - *not taken* e *taken* - *taken* para as suas duas entradas.

entrada	grupo de branch	último resultado	estado interno	predição
0	0	not taken - not taken	3	taken
1	0	not taken - taken	0	not taken
2	0	taken - not taken	0	not taken
3	0	taken - taken	2	taken
4	1	not taken - not taken	0	not taken
5	1	not taken - taken	0	not taken
6	1	taken - not taken	2	taken
7	1	taken - taken	3	taken

figura 2.61

Tendo a tabela de inicialização na Figura 2.61 podemos agora tentar completar uma tabela semelhante à da Figura 2.62.

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
		taken - not taken			
		taken - taken			

figura 2.62

Tendo apenas dois grupos de branches, o melhor a fazer, para contextualizarmos as nossas instruções de branch, será verificar quais delas são pares e quais delas serão ímpares. Assim sendo, como 1C6 é par, este ficará acessível pelas entradas 0, 1, 2, e 3, e as outras, sendo ímpares, ficam acessíveis pelas entradas 4, 5, 6, 7.

Comecemos então por olhar para a linha da Figura 2.62 e tentemos preenchê-la consoante as predições. Inicialmente já temos os estados do registo de histórico local. Assim sendo, começando pela instrução 1C6 temos que esta, sendo par, tem entrada em "010" (entrada em 2 por ser par, registo de *taken* e *not taken*). Tal entrada possui uma predição de *not taken*, contrária ao esperado resultado de *taken*, com um estado interno de 0 que nos levará a uma atualização para o estado 1. Veja-se, assim, a Figura 2.63.

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
1C6	taken	taken - not taken	2	not taken	1
		taken - taken			

figura 2.63

Continuando, agora na instrução 21F, temos que para um resultado espetável de *not taken*, ficamos com acesso à entrada "111" (entrada número 7) cuja predição é *taken*. Mais, o estado interno de 7 é 3 o que, consultando a Figura 2.17, nos mostra que com o resultado de *not taken* é atualizado para 2 (vide Figura 2.64).

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
1C6	taken	taken - not taken	2	not taken	1
21F	not taken	taken - taken	7	taken	2

figura 2.64

Agora, o registo de histórico local, apresentado de forma diferente para cada grupo, será atualizado conforme estejamos a trabalhar numa instrução do grupo 1 ou 0.

Por exemplo, agora que vamos avançar para a instrução 309, que é do grupo 1 (ímpares), temos que atualizar o registo de histórico local para esta instrução, transitando o mais recente para o mais antigo e o último resultado para o mais recente, ficando algo como *not taken - taken*. Assim, podemos aceder à entrada 5, a qual tem predição *not taken* e um estado 0, que com o nosso resultado de *not taken* nos levará novamente para 0. Ficamos assim com a tabela da Figura 2.65.

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
1C6	taken	taken - not taken	2	not taken	1
21F	not taken	taken - taken	7	taken	2
309	not taken	not taken - taken	5	not taken	0

figura 2.65

Entrando a análise da instrução 21F, ímpar - logo do grupo 1 -, atualizamos o histórico para *not taken - not taken* e, com um resultado final de *not taken*, acedemos à posição "100", logo 4 com predição *not taken*, que possui um estado de 0 para uma atualização para 0.

Avançando temos ainda a instrução 309, novamente ímpar, a qual leva a atualização de *not taken - not taken* para o LHR e permite o acesso à entrada 4, com predição *not taken* e atualização, igual à anterior, de 0. Veja-se assim a Figura 2.66.

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
1C6	taken	taken - not taken	2	not taken	1
21F	not taken	taken - taken	7	taken	2
309	not taken	not taken - taken	5	not taken	0
21F	not taken	not taken - not taken	4	not taken	0
309	not taken	not taken - not taken	4	not taken	0

figura 2.66

Agora voltamos a uma instrução do grupo 0 - a 1C6. Esta instrução faz com que o LHR se atualize de *taken - not taken* para *taken - taken*. Desta forma, podemos aceder à entrada "011" (entrada 3), com resultado coincidente com predição de *taken*, onde vemos um estado interno de 2 que, com resultado *taken*, nos leva para 3. Veja-se assim a Figura 2.67.

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
1C6	taken	taken - not taken	2	not taken	1
21F	not taken	taken - taken	7	taken	2
309	not taken	not taken - taken	5	not taken	0
21F	not taken	not taken - not taken	4	not taken	0
309	not taken	not taken - not taken	4	not taken	0
1C6	taken	taken - taken	3	taken	3

figura 2.67

Prosseguindo no preenchimento da tabela, temos que esta teria o aspeto da tabela da Figura 2.68, a qual mostra três erros de predição, o que nos leva a concluir que temos uma taxa de falha de predição de 1/3.

Especulação

Manter as dependências de controlo começa a tornar-se num fardo muito grande quase tentamos explorar o paralelismo a nível de instruções com mais profundidade. A predição de branches, como já tivemos mais do que uma oportunidade de ver, reduz o número de *stalls* diretamente atribuídos a branches, mas apenas prever os branches de for-

endereço de branch	resultado	registo de histórico local (LHR)	entrada	predição	atualização de predição
1C6	taken	taken - not taken	2	not taken	1
21F	not taken	taken - taken	7	taken	2
309	not taken	not taken - taken	5	not taken	0
21F	not taken	not taken - not taken	4	not taken	0
309	not taken	not taken - not taken	4	not taken	0
1C6	taken	taken - taken	3	taken	3
309	not taken	not taken - not taken	4	not taken	0
1C6	taken	taken - taken	3	taken	3
21F	taken	not taken - not taken	4	not taken	1

figura 2.68

ma rigorosa poderá não ser suficiente para gerar a quantidade desejada de paralelismo de instrução para um processador que execute múltiplas instruções por ciclo de relógio. Um processador de amplo lançamento de instruções poderá ter necessidade de executar um branch a cada ciclo de relógio para manter o desempenho num valor pico. Desta forma, explorar mais paralelismo requer que as limitações tidas até agora de dependências de controlo sejam ultrapassadas. Isto é feito **especulando** o resultado dos branches e executando o código como se as especulações estivessem corretas. Esta ideia representa uma extensão súbtil e crucial sobre a predição de branches com agendamento dinâmico: com a especulação as instruções são carregadas, lançadas e executadas como se as predições estivessem sempre corretas. O agendamento dinâmico apenas carrega e lança tais instruções.

especulando

Este mecanismo de especulação, com base em *hardware*, baseia-se em três ideias-chave: predição dinâmica de branches, especulação (para permitir a execução de instruções antes da resolução das dependências de controlo - com noção de que os efeitos produzidos por uma especulação incorreta podem levar a situações reversíveis) e agendamento dinâmico. Em contraste, o agendamento dinâmico (sem a especulação) sobrepõe diversos *basic blocks* só parcialmente, dado que requer que um branch seja calculado antes da execução de qualquer instrução de um bloco sucessor.

Por conseguinte, a especulação segue um fluxo de dados previsto para determinar quando é que se deve executar cada instrução. Este método de execução é essencialmente uma execução de fluxo de dados: as operações são reproduzidas logo que os seus operandos-fonte estejam disponíveis.

Revisitando o algoritmo de Tomasulo, este poderá suportar a especulação se, o *bypassing* dos resultados entre as demais instruções, que é necessário para que as instruções sejam executadas através da especulação, houver uma separação da conclusão das instruções. Se tal mecanismo for cumprido, uma instrução poderá receber permissão para ser executada e para passar o seu resultado a todas as outras, sem que haja a necessidade de permissão para o desempenhar de atualizações irreversíveis até que seja estabelecido que a instrução não é mais especulativa.

Usar um valor que fora passado por alguém é como se estivesse a tratar de uma leitura de um registo de forma totalmente especulativa, sendo que pode acontecer o valor não ser conhecido no momento em que a instrução que fornece um valor para o registo-fonte está, de facto, a fornecer um valor real. Somente quando a instrução não é mais especulativa é que o registo em específico do banco de registos, ou a região de memória referida, pode ser atualizada - este passo adicional na execução de uma instrução é frequentemente denominada de **commit de instrução**.

commit de instrução

Assim, as instruções poderão ser executadas fora-de-ordem, embora sejam todas obrigadas a efetuarem um *commit* em-ordem, para que nenhuma ação irreversível, como a atualização de estados ou lançamento de exceções, seja previnida. A separação da conclusão da instrução do *commit* é essencial porque as instruções poderão terminar a sua execução consideravelmente antes de estarem prontas para o efetuarem (o *commit*).

A introdução desta fase de *commit* requer assim um conjunto de *buffers* extra (a nível de *hardware*) que preservarão as instruções completadas e que ainda não foram sujeitas a *commit*. Este banco de *buffers*, denominado de **buffer de reordenação** (do inglês *reorder buffer*, ou ROB) é também usado para passar os resultados entre as instruções. Note-se que este *buffer* fornece registos adicionais na mesma forma que as estações de reserva e os *buffers* de *load* e *store* extendem o conjunto de registos no algoritmo de Tomasulo. A grande diferença está que, no algoritmo de Tomasulo, assim que uma instrução termina a escrita de resultados, quaisquer instruções lançadas subsequentemente irão obter o valor do banco de registos, enquanto que, com a especulação o banco de registos não é atualizado até ao *commit* das instruções, o que significa que o *buffer* de reordenação fornece os operandos para outras instruções no intervalo entre a conclusão das instruções e os seus *commits*. Este *buffer* de reordenação é assim semelhante aos *buffers* de *store* no algoritmo de Tomasulo, pelo que esta funcionalidade é também integrada no ROB por questões de simplicidade.

buffer de reordenação

Na Figura 2.69 [8] podemos ver a aplicação da especulação sobre o algoritmo de Tomasulo da Figura 2.48.

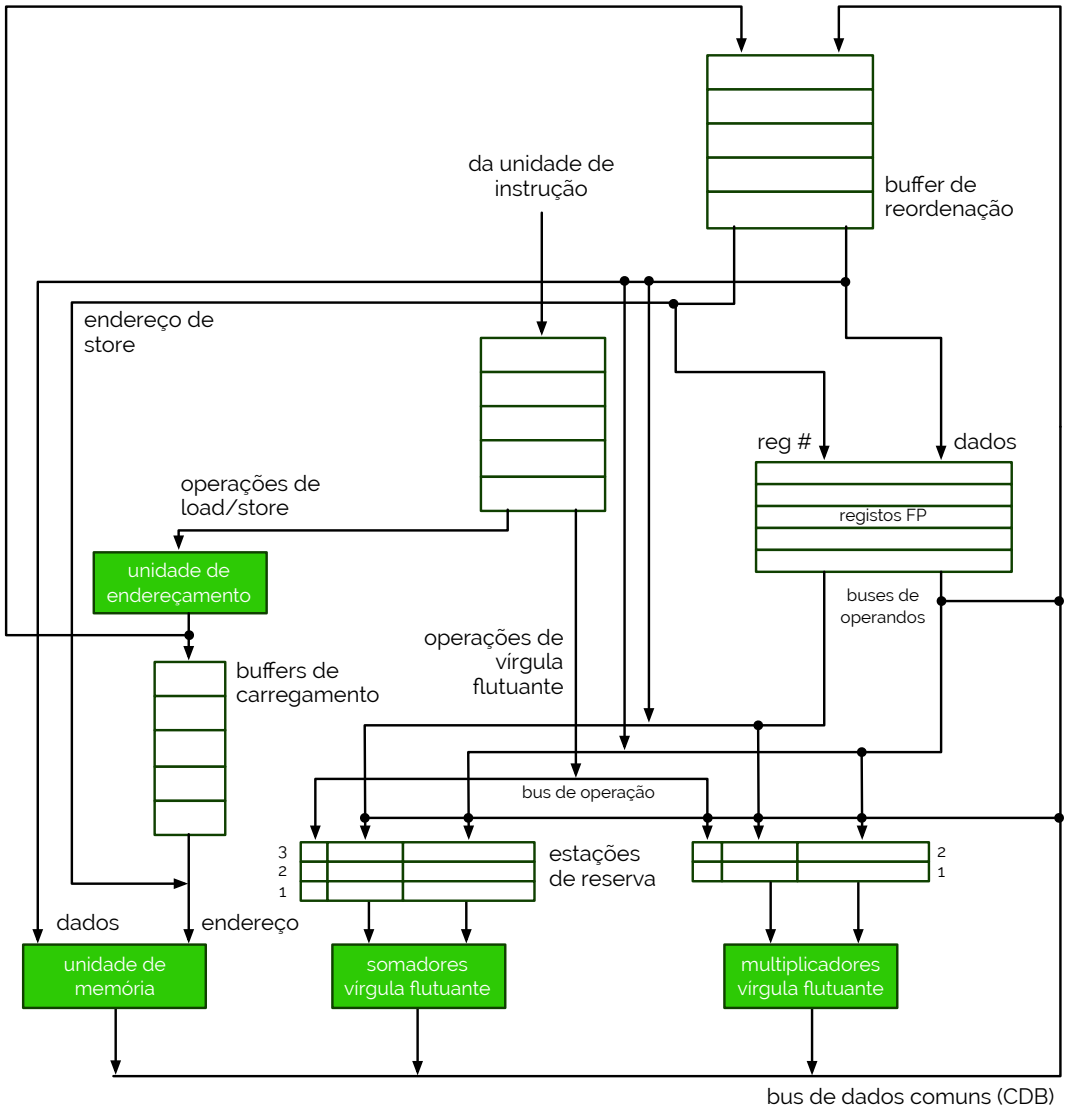


figura 2.69

As operações de *store*, com a especulação, são executadas em dois passos, mas o segundo passo é incluído no *commit* da instrução. Embora a função de renomeação das estações de reserva sejam substituídas pelo *buffer* de reordenação, será ainda necessário um espaço para que as operações e operandos sejam carregados entre o tempo em que são lançados e o tempo em que acabam a sua execução. Esta função é assim executada pelas

estações de reserva e pelos *buffers* de carregamento. Por outro lado, sendo que todas as instruções ficam com uma entrada no ROB até que façam *commit*, o resultado da operação é etiquetado usando a sua entrada no ROB, ao invés do número da estação de reserva ou *buffer* de *load*.

No algoritmo de especulação contamos assim com quatro fases de execução, entre elas a fase de lançamento, de execução, de escrita de resultados e de *commit*.

Na fase de lançamento a instrução seguinte é obtida da cabeça da fila de instruções. Se tanto uma estação de reserva ou *buffer* correspondentes e um *slot* do ROB estiver livre, a instrução é então lançada para a estação ou para o *buffer*, acompanhada dos seus valores dos operandos (se eles estiverem guardados nos registos do banco de registos ou nas entradas do *buffer* de reordenação). Por outro lado, um acompanhamento das entradas do ROB que eventualmente conterão os operandos é desempenhado. O número de entradas do ROB alocadas é também levado para as estações de reserva ou *buffer*, de forma a que o número possa ser usado para etiquetar o resultado quando for colocado no bus de dados comum (CDB).

Terminada a fase de lançamento, o CDB é monitorizado enquanto espera pelos operandos serem calculado, evitando assim *hazards* RAW. No momento em que os operandos se tornam disponíveis, a operação é executada. Tal execução poderá durar vários ciclos de relógio, sendo que os *loads* requerem dois passos de execução. Os *stores*, como já foi referido anteriormente, por outro lado, apenas calculam o endereço efetivo, sendo que o segundo passo já inclui o *commit* da instrução.

Com a execução terminada, chega agora o momento em que é permitida a escrita dos resultados. Nesta fase, o resultado de uma instrução é colocado no bus de dados comum (CDB), juntamente com a sua etiqueta fornecida pelo ROB, para ser escrito do *buffer* de reordenação e em todas as estações de reserva (e *buffers*) que esperam pelo seu valor. A estação de reserva ou *buffer* é então marcado como livre e o campo *ready* do *buffer* de reordenação (que veremos mais à frente) é colocado a *set*.[2]

Finalmente, tendo uma operação sido terminada em termos de execução, há que então fazer o *commit*. Assim o conjunto de ações associadas dependem no facto de se a instrução a ser sujeita a *commit* é um branch com uma predição incorreta, um *store* ou qualquer outra instrução (*commit* normal). Um *commit* normal ocorre quando uma instrução já comtempla a cabeça do *buffer* de reordenação e o seu campo *read* está a *set*, sendo que cabe depois ao processador atualizar o registo do banco de registos, se algum houver, e terminar a operação[3]. Para um branch com uma predição errada o ROB é limpo e a operação de *fetch* é reiniciada no endereço próprio, definido pelo comportamento do branch.

Cada entrada do *buffer* de reordenação tem cinco campos, entre os quais:

- *busy* - indica se a entrada está ocupada ou livre;
- tipo de instrução - assinala se a instrução é um branch sem resultado de destino, um *store* com destino na memória ou um *load*, ou uma operação sobre a ALU, com um destino num registo do banco de registos;
- local de destino - indica um endereço de memória para *stores* ou um número de registo para *loads* e operações sobre a ALU, onde o resultado de uma instrução deve ser escrito;
- *value* - guarda o valor do resultado de uma instrução entre as fases de conclusão e de *commit*;
- *ready* - indica se a instrução já completou a execução ou não.

Consideremos então o seguinte Código 2.31.

---

2 No caso de operações *store*, algumas ações especiais são necessárias. Entre elas, se o valor a ser guardado já se encontra disponível, então este é escrito no campo *value* da entrada do ROB. Por outro lado, se o valor ainda se encontra indisponível, então o bus comum de dados (CDB) é monitorizado de forma a capturar o seu valor, guardando-o no *buffer*, atualizando o campo *value* da entrada do ROB.

3 Fazendo o *commit* de uma operação de *store* é semelhante, à exceção de que a região de memória é atualizada, contrariamente a um registo do banco de registos.

```
l.d     f6, 32(r2)
l.d     f2, 44(r3)
mul.d   f0, f2, f4
sub.d   f8, f2, f6
div.d   f10, f0, f6
add.d   f6, f8, f2
```

código 2.31

Assumindo as latências de 1 ciclo para *load*/*store*, 2 ciclos para adições, 6 ciclos para multiplicações e 12 ciclos para divisões, vejamos o conteúdo das tabelas da Figura 2.70 quando a instrução de multiplicação está pronta para efetuar *commit*.

figura 2.70

buffer de reordenação						
entrada	estado	busy	instrução	destino	value	ready
1	commit	não	`l.d f6,32(r2)`	f6	mem[reg[r2] + 32]	sim
2	commit	não	`l.d f2,44(r3)`	f2	mem[reg[r3] + 44]	sim
3	escrita de res.	sim	`mul.d f0,f2,f4`	f0	#2 x reg[f4]	sim
4	escrita de res.	sim	`sub.d f8,f2,f6`	f8	#2 - #1	sim
5	execução	sim	`div.d f10,f0,f6`	f10		não
6	escrita de res.	sim	`add.d f6,f8,f2`	f6	#4 + #2	sim

estações de reserva / buffers								
nome	busy	op	V_j	V_k	Q_j	Q_k	destino	A
load1	não							
load2	não							
add1	não							
add2	não							
add3	não							
mult1	não	mult	mem[44+reg[r2]]	reg[f4]			#3	
mult2	sim	div		mem[32+reg[r2]]	#3		#5	

estado de registos									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
busy	sim	não	não	sim	sim	sim	não		não
ent. ROB	3			6	4	5			

Consideremos, com as mesmas latências que anteriormente, agora o Código 2.32.

```
loop:   l.d     f0, 0(r1)
        mul.d   f4, f0, f2
        s.d     f4, 0(r1)
        daddiu  r1, r1, -8
        bne     r1, r2, loop
```

código 2.32

Assume-se que o registo `r1` inicialmente contém o endereço do último elemento de um array e que o conteúdo do registo `r2+8` é o endereço do primeiro elemento do array. Se o branch previr *taken*, múltiplas execuções do ciclo irão decorrer em paralelo. Assumamos então, também, que toda as instruções no ciclo foram lançadas duas vezes, os *load* e multiplicação da primeira iteração já fizeram *commit* e todas as outras já terminaram a execução. Na Figura 2.71 podemos ver uma tabela do *buffer* de reordenação e das etiquetas dos registos quando duas iterações sucessivas de um ciclo já foram lançadas e, as instruções de *load* e de multiplicação da primeira iteração já foram sujeitas a *commit* e todas as outras já terminaram a sua execução.

Sendo que nem os valores dos registos, nem os valores residentes na memória são, de facto, escritos até que haja o *commit* de uma instrução, o processador pode facilmente anular as suas ações especulativas quando um branch é dado como mal previsto. As instruções anteriores ao branch serão então sujeitas a *commit* de forma sucessiva quando se atinge a cabeça do *buffer* de reordenação.

figura 2.71

buffer de reordenação						
entrada	estado	busy	instrução	destino	value	ready
1	commit	não	`l.d f0,0(r1)`	f0	mem[reg[r1] + 0]	sim
2	commit	não	`mul.d f4,f0,f2`	f4	#1 x reg[f2]	sim
3	escrita de res.	sim	`s.d f4,0(r1)`	reg[r1]+0	#2	sim
4	escrita de res.	sim	`daddiu r1,r1,-8`	r1	reg[r1] - 8	sim
5	escrita de res.	sim	`bne r1,r1,loop`			sim
6	escrita de res.	sim	`l.d f0,0(r1)`	f0	mem[#4]	sim
7	escrita de res.	sim	`mul.d f4,f0,f2`	f4	#6 x reg[f2]	sim
8	escrita de res.	sim	`s.d f4,0(r1)`	#4	#7	sim
9	escrita de res.	sim	`daddiu r1,r1,-8`	r1	#4 - 8	sim
10	escrita de res.	sim	`bne r1,r1,loop`			sim

estado de registos									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
busy	sim	não	sim	não	não	não	não		não
ent. ROB	6		7						

As exceções são tratadas não as reconhecendo até que a instrução que as comete seja *commited*. Se uma instrução especulada lança uma exceção, esta é gravada na entrada do ROB associada. Quando uma má-predição é dada e a instrução não deveria ter sido executada, a exceção é limpa juntamente com o *buffer* de reordenação. Se a instrução atinge a cabeça de tal *buffer*, então não é mais especulativa e a exceção é tomada.

Em suma, na tabela da Figura 2.72 podemos ver as várias verificações e atualizações do algoritmo de especulação com base em *hardware*, onde ROB[b] é uma entrada do ROB e RS[r] é uma estação de reserva/*buffer* associado com a instrução (e *regStat* é um registo de estado).

figura 2.72

estado	critério de espera	atualizações

lançamento operação FP	(RS[r].busy == não) && (ROB[b].busy == não)	**se** (regStat[rs].busy == sim) {
		  x = regStat[rs].reorder;
		  se (ROB[x].ready == sim) {
		  RS[r].V_j = ROB[x].value;
		  RS[r].Q_j = 0;
		  } **caso contrário** {
		  RS[r].Q_j = x;
		  }
		} **caso contrário** {
		  RS[r].V_j = reg[rs];
		  RS[r].Q_j = 0;
		}
		se (regStat[rt].busy == sim) {
		  x = regStat[rt].reorder;
		  se (ROB[x].ready == sim) {
		  RS[r].V_k = ROB[x].value;
		  RS[r].Q_k = 0;
		  } **caso contrário** {
		  RS[r].Q_k = x;
		  }
		} **caso contrário** {
		  RS[r].V_k = reg[rt];
		  RS[r].Q_k = 0;
		}
		regStat[rd].reorder = b;
		regStat[rd].busy = sim;
		RS[r].dest = b;
		RS[r].busy = sim;
		ROB[b].inst = opcode;
		ROB[b].dest = rd;
		ROB[b].ready = não;
		ROB[b].busy = sim;

figura 2.72 (continuação)

estado	critério de espera	atualizações

lançamento load	(RS[r].busy == não) && (ROB[b].busy == não)	**se** (regStat[rs].busy == sim) {
		  x = regStat[rs].reorder;
		  se (ROB[x].ready == sim) {
		  RS[r].V_j = ROB[x].value;
		  RS[r].Q_j = 0;
		  } **caso contrário** {
		  RS[r].Q_j = x;
		  }
		} **caso contrário** {
		  RS[r].V_j = reg[rs];
		  RS[r].Q_j = 0;
		}
		regStat[rt].reorder = b;
		regStat[rt].busy = sim;
		RS[r].A = imm;
		RS[r].dest = b;
		RS[r].busy = sim;
		ROB[b].inst = opcode;
		ROB[b].dest = rt;
		ROB[b].ready = não;
		ROB[b].busy = sim;

lançamento store	(RS[r].busy == não) && (ROB[b].busy == não)	**se** (regStat[rs].busy == sim) {
		  x = regStat[rs].reorder;
		  se (ROB[x].ready == sim) {
		  RS[r].V_j = ROB[x].value;
		  RS[r].Q_j = 0;
		  } **caso contrário** {
		  RS[r].Q_j = x;
		  }
		} **caso contrário** {
		  RS[r].V_j = reg[rs];
		  RS[r].Q_j = 0;
		}
		se (regStat[rt].busy == sim) {
		  x = regStat[rt].reorder;
		  se (ROB[x].ready == sim) {
		  RS[r].V_k = ROB[x].value;
		  RS[r].Q_k = 0;
		  } **caso contrário** {
		  RS[r].Q_k = x;
		  }
		} **caso contrário** {
		  RS[r].V_k = reg[rt];
		  RS[r].Q_k = 0;
		}
		RS[r].A = imm;
		RS[r].busy = sim;
		ROB[b].inst = opcode;
		ROB[b].ready = não;
		ROB[b].busy = sim;

execução operação FP	(RS[r].Q_j == 0) && (RS[r].Q_k == 0)	**calcular** resultado (operandos em V_j e V_k)
execução load (passo 1)	(RS[r].Q_j == 0) && não há stores anteriores na fila ROB	RS[r].A = RS[r].V_j + RS[r].A;
execução load (passo 2)	passo 1 está completo && todos os stores anteriores na fila ROB têm endereços efetivos diferentes	**ler de** mem[RS[r].A];
execução store	(RS[r].Q_j == 0) && b é cabeça da fila ROB	ROB[b].dest = RS[r].V_j + RS[r].A;

escrita de resultado de operação FP ou load	r completou a execução && CDB está disponível	b = RS[r].dest;
		(**se** (regStat[x].Q_j == b) {
		  RS[x].V_j = resultado;
		  RS[x].Q_j = 0;
		})
		(**se** (regStat[x].Q_k == b) {
		  RS[x].V_k = resultado;
		  RS[x].Q_k = 0;
		})
		ROB[b].value = resultado;
		ROB[b].ready = sim;
		RS[r].busy = não;

estado	critério de espera	atualizações
escrita de resultado de store	r completou a execução && ($RS[r].Q_k$ == 0)	ROB[b].value = $RS[r].V_k$; RS[r].busy = não;

commit	b é cabeça da fila ROB && (ROB[b].ready == sim)	**se** (ROB[b].inst == branch) {
		  se (branch mal previsto) {
		  limpar ROB;
		  reset estações de reserva/buffers de load;
		  reset registo de estado;
		  fetch at destino de branch;
		  }
		} **caso contrário** {
		  se (ROB[b].inst == store) {
		  mem[ROB[b].dest] = ROB[b].value;
		  } **caso contrário** {
		  reg[ROB[b].dest] = ROB[b].value;
		  se (regStat[ROB[b].dest].reorder == b) {
		  regStat[ROB[b].busy] = não;
		  }
		  }
		}
		ROB[b].busy = não;

figura 2.72 (continuação)

3. Paralelismo de dados

Tendo estudado os vários níveis de paralelismo de instrução, uma outra frente que poderá ser abordada e paralelizada são os **dados**. O paralelismo de dados significa que múltiplas fontes de dados estão a ser processados em simultâneo. Esta aplicação necessita de um nível de abordagem muito mais rigoroso, sendo que a própria complexidade é bem maior [11].

dados

Existem dois tipos diferentes de arquiteturas de computadores que atingem o objetivo da paralelização de dados: arquiteturas SIMD e MIMD. Uma arquitetura **SIMD**, isto é, *Single Instruction Multiple Data* (em português Instrução Singular, Dados Múltiplos) é potencialmente mais eficiente, em termos energéticos, que uma arquitetura **MIMD**, isto é, *Multiple Instructions Multiple Data* (em português Instruções Múltiplas, Dados Múltiplos), onde uma instrução é carregada por operação de dados. Uma vantagem das SIMD sobre as MIMD é que o programador poderá continuar a pensar de forma sequencial (forma a que todos estamos habituados), enquanto que com MIMD, embora haja um *speedup* pela paralelização, torna-se difícil pensar na aplicação de técnicas para operações sobre dados.

SIMD

MIMD

Arquiteturas vetorizadas

As **arquiteturas vetorizadas** pegam num conjunto de elementos de dados espalhados sobre a memória e transferem-nos para bancos de registos grandes e sequenciais, operando sobre estes através de manipulações algébricas matriciais, dispersando, no fim, os resultados de volta à memória. Assim, uma única instrução opera em **vetores** de dados, permitindo dezenas de operações de registos para registos em elementos de dados independentes.

arquiteturas vetorizadas

vetores

Estes bancos de registos grandes atuam, a fim de contas, como *buffers* controlados pelos compiladores, de forma a poder abstrair a latência e largura de banda da memória. Devido ao facto de que os carregamentos e as transferências estão fundamentalmente instituídas sobre um pipeline, o programa terá de minimizar a latência uma vez por cada carregamento ou transferência de vetor, ao invés de uma vez por cada elemento. De facto, os programas vetoriais esforçam-se por tentar manter um registo na memória de forma contígua, de forma a que possa ser dado lugar a técnicas de paralelismo executadas pelo processador.

A base para esta secção é fortemente baseada num primeiro computador com arquitetura vetorizada, de seu nome Cray-1. Sendo que o nosso estudo também se baseia na arquitetura MIPS, criamos assim uma versão denominada de VMIPS que junta à parte

escalar já estudada, uma vertente vetorial como uma extensão do próprio MIPS que já conhecemos. Agora, consideremos então que temos registos-vetor, unidades funcionais para vetores, unidades de carregamento e transferência de vetores e registos meramente escalares.

registos-vetor

Os **registos-vetor** têm em si um banco de tamanho fixo onde guardam um só vetor. A arquitetura VMIPS terá então 8 registos-vetor, cada um preservando 64 elementos de 64 bits, sendo que o banco destes registos fornece entradas e saídas suficientes para alimentar todos as unidades funcionais vetoriais com um elevado grau de sobreposição[4].

unidades funcionais vetoriais

As **unidades funcionais vetoriais** têm, cada uma, um pipeline implementado e completo, podendo iniciar uma nova operação a cada ciclo de relógio. Será claramente necessário uma unidade para o controlo de *hazards* que se encontra também implementado ao longo do pipeline.

unidades de load/store vetoriais

As **unidades de load/store vetoriais** permitem a troca de informação entre o banco de registos e a memória com a frequência de uma palavra por ciclo de relógio (dada a implementação de pipeline), depois da latência inicial. Esta unidade também tolera as operações para escalares.

instruções vetoriais

A interação com estas arquiteturas também é feita com um conjunto de instruções completamente diferente - as **instruções vetoriais**. Tais instruções permitem que seja aumentado o desempenho de simples processadores de execução em-ordem sem que haja grandes transtornos no que toca a consumo de energia ou complexidade de desenho. Na prática, os *developers* já podem e conseguem exprimir grande parte dos programas que correm bem em desenhos fora-de-ordem, mais complexos, igualmente ou superiormente mais eficientes sobre a forma de instruções vetoriais que trabalham sobre paralelismo de dados.

Com uma instrução/expressão vetorial, estamos a fornecer ao nosso processador (ou processadores) a capacidade de calcular um determinado valor de mais do que uma forma, incluíndo operar simultaneamente em elementos dados. Esta flexibilidade permite que sejam usadas unidades lentas, mas largas e com baixa potência.

Mais, também é permitido, com a adaptação vetorial, a acomodação de tipos de dados variáveis, como 64 registos-vetor de tamanho 64 bits a serem vistos como 128 registos de 32 bits ou 256 registos de 16 bits, ... Esta multiplicidade permite que uma arquitetura vetorial seja útil, por exemplo, para aplicações científicas ou de multimédia, como abordamos na disciplina de Computação Visual (a4s1).

Consideremos a operação em (3.1), de forma vetorial, isto é, considerando que X e Y são vetores, inicialmente residentes na memória, e a é um escalar.

$$Y = a \times X + Y \tag{3.1}$$

SAXPY

Este problema é vulgarmente chamado de **SAXPY** (ou DAXPY, para o caso de precisão dupla). Se considerarmos que os endereços iniciais de memória onde os vetores X e Y estão guardados são `rx` e `ry`, respetivamente, no Código 3.1 podemos ver uma versão para MIPS e no Código 3.2 podemos ver uma versão para o VMIPS.

código 3.1

```
        l.d     f0, a
        daddiu  r4, rx, 512
loop:   l.d     f2, 0(rx)
        mul.d   f2, f2, f0
        l.d     f4, 0(ry)
        add.d   f4, f4, f2
        s.d     f4, 0(ry)
        daddiu  rx, rx, 8
        daddiu  ry, ry, 8
        dsubu   r20, r4, rx
        bnez    r20, loop
```

4 Existirão 16 portas para leitura e 8 portas para escrita que estão conectadas às unidades funcionais vetoriais através de um *crossbar switch*.

```
l.d      f0, a
lv       v1, rx
mulvs.d  v2, v1, f0
lv       v3, ry
addvv.d  v4, v2, v3
sv       ry, v4
```

código 3.2

Se fizermos uma contagem do número de instruções tidas por um processador MIPS para o Código 3.1 e por um processador VMIPS para o Código 3.2, podemos chegar a valores de 578 instruções contra 6 instruções, respetivamente. Esta redução do número de instruções deve-se ao facto das instruções vetoriais trabalharem diretamente sobre 64 elementos, como inteiro, e de que as instruções de sobreposição, que representam cerca de metade das de controlo do ciclo em MIPS, não estão presentes aqui.

Quando um compilador gera instruções vetoriais para uma sequência e o código resultante toma muito tempo em execução em modo vetorial diz-se que o código é **vetorizado** (ou vetorizável). Os ciclos podem apenas ser vetorizados se neles não houver qualquer dependência de dados entre as demais iterações.

vetorizado

Outra diferença importante é a frequência de *interlocks* do pipeline, que, enquanto que no MIPS ocorreriam uma vez por cada iteração, no VMIPS, cada instrução vetorizada só parará pelo primeiro elemento de cada vetor, sendo que os outros elementos fluirão normalmente pelo pipeline. Assim, os *stalls* no pipeline só são requiridos uma vez por instrução vetorial, ao invés de uma vez por elemento. Às operações de *forwarding* damos o nome, em termos de ambientes vetorizados, de **chaining**, isto porque as operações dependentes estão encadeadas umas com as outras.

chaining

Em suma, o tempo de execução das operações vetorizadas depende essencialmente em três fatores: o tamanho dos vetores-operando, o número de *hazards* estruturais e as dependências de dados entre operações sucessivas. Note-se assim que dado um tamanho de vetor e a frequência de iniciação (frequência à qual uma unidade vetorial em particular consome novos operandos e produz novos resultados) o tempo que uma dada instrução demora é possível de ser calculado. Por questões de simplicidade assumamos que a implementação VMIPS possui todas as suas unidades funcionais com um pipeline único, isto é, com uma **pista** (do inglês *lane*), com uma frequência de iniciação de 1 elemento por ciclo de relógio para operações individuais. Assim, o tempo de execução para uma operação de vetor único é aproximadamente igual ao tamanho do vetor.

pista

Um conceito importante para a discussão do desempenho e execução de vetores é a noção de **convoy**, que é um conjunto de instruções de vetor que podem ser executadas, potencialmente, todas juntas. As instruções contidas num *convoy* não devem assim conter quaisquer razões para a existência de *hazards* estruturais. Se tais *hazards* estivessem presentes, então necessitariam de ser serializados e iniciados em *convoys*. Novamente, para não aumentar muito o nível de complexidade, consideremos sempre que um *convoy* de instruções deve terminar as suas execuções antes que qualquer outra, seja vetorial ou escalar, comece. Com isto, até pode parecer que para além de sequências de instruções com *hazars* estruturais, sequências com *hazards* do tipo RAW também devem ser executadas em *convoys* separados. Note-se, no entanto, que com a técnica de *chaining* ativa, é permitido (e garantido) que estas instruções sejam executadas todas juntas.

convoy

Consideremos assim novamente o Código 3.2, onde um mínimo de três *convoys* são necessários para o executar. O primeiro *convoy* inicia com a primeira instrução de *load*. A instrução de multiplicação é dependente dela, mas o *chaining* permite-lhe que seja incluída no mesmo *convoy*. A instrução de soma é dependente da segunda instrução de *load*, mas também poderá ser feita no mesmo *convoy*, dado que o *chaining* está presente. Por fim, a instrução de *store* induz um *hazard* estrutural, pelo que deverá ser instaurado num terceiro *convoy*.

Para tornar os *convoys* em tempo de execução uma determinada métrica deve ser necessária para estimar o tempo de um *convoy*. Esta métrica é geralmente denominada de **chime**, sendo que uma determinada sequência de código vetorial que consista de m execuções de *convoys* executa em m *chimes*: para vetores de tamanho n, então significa que estamos perante um tempo de execução de cerca de $m \times n$ ciclos de relógio para VMIPS.

A aproximação dos *chimes* ignora alguns dos *overheads* causados pelo processador, alguns dos quais estão dependentes do tamanho do vetor. Por conseguinte, medindo o tempo de execução em *chimes* é uma aproximação mais rigorosa para vetores de tamanhos maiores, do que para mais pequenos. Nesta medição, uma fonte ignorada de *overhead* é a possível limitação de iniciar múltiplas instruções vetoriais num mesmo ciclo de relógio. Se somente uma instrução vetorial poderá ser lançada por ciclo de relógio (como acontece na maior parte dos processadores vetoriais), então a contagem de *chimes* será menor que o tempo de execução do *convoy*. A mais importante fonte de *overhead* ignorada pelo modelo de *chime* é o *vector start up time*. Este tempo é principalmente determinado pela latência das unidades funcionais vetoriais.

Então como é que um processador vetorial pode executar um único vetor em mais do que um elemento por ciclo de relógio? Uma vantagem crítica de um conjunto de instruções vetoriais é que este permite que um programa passe uma grande quantidade de trabalho paralelo para o *hardware* usando apenas uma instrução pequena. Esta instrução inclui os resultados de operações independentes, ainda codificada no mesmo número de bits que uma instrução escalar convencional. Esta semântica paralela de uma instrução vetorial permite assim uma implementação para executar tais operações elementares através de tanto um pipeline mais profundo, como de um array paralelo de unidades funcionais (ou como de uma combinação de ambas).

O conjunto de instruções vetoriais do VMIPS tem assim a propriedade de que todas as instruções vetoriais aritméticas permitem que o elemento k de um registo-vetor tome parte de operações com o elemento k de outro registo-vetor. Isto simplifica drasticamente a construção de uma unidade vetorial altamente paralelizada, que poderá ser estruturada como tendo mais do que uma pista paralela. Este cenário é possível de ser visto na Figura 3.1.

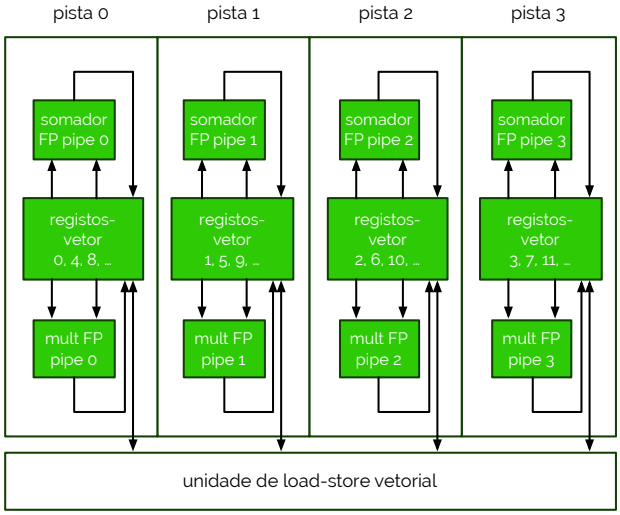


figura 3.1

Mudar de uma para quatro pistas, como na Figura 3.1, reduz o número de ciclos de relógio num *chime* de 64 para 16. Contudo, para que uma implementação multi-pista seja eficaz, tanto as aplicações como a arquitetura terão de suportar a manipulação de vetores longos, caso contrário a execução será tão rápida que haverá o risco de ficar sem largura de banda para novas instruções, pedindo a aplicação de técnicas de paralelização de instruções para que se alimentem instruções vetorais suficientes.

Cada pista contém uma porção do banco de registos-vetor e uma execução do pipeline por cada unidade funcional vetorial. Estas, assim, executam as suas instruções à frequência de um grupo de elementos por ciclo de relógio. Evitando a comunicação entre as várias pistas, existe um baixo impacto a nível de custo de implementação e o número de portas para registos também é reduzido.

Adicionar múltiplas pistas é uma técnica muito comum para melhorar o desempenho dos vetores, o que também requer um pouco mais de controlo sobre a complexidade

e não aplica esforços extra no que toca ao código máquina. Também permite aos *designers* fazer um *trade-off* da área de sílica da bolacha, frequência de relógio e energia, sem sacrificar o desempenho de pico.

Um processador vetorial tem assim um tamanho determinado pelo número de elementos em cada registo-vetor. Este tamanho é pouco provável de corresponder ao tamanho real de um vetor na realidade. Mais, num programa real o tamanho de um vetor em particular numa operação é frequentemente desconhecido em tempo de compilação, sendo que, de facto, um pequeno segmento de código pode até requirir diferentes tamanhos de vetores, como acontece se for incluído numa função, que terá como parâmetro precisamente o número de iterações de um ciclo.

A solução para estes problemas é criar um registo de tamanho de vetor (em inglês **vector length register**, VL). Este registo controla assim o tamanho do vetor numa operação, incluíndo *loads* e *stores*. O valor guardado no VL, no entanto, não pode ser maior que o tamanho dos registos-vetor, pelo que soluções baseadas no registo VL só funcionarão caso o tamanho do vetor real é menor ou igual que o tamanho máximo do vetor (MVL) - número de elementos de dados em registos-vetor para a implementação atual. Agora, quando o tamanho real de um vetor é maior que o MVL, uma técnica de seu nome **strip mining** é aplicada.

vector length register

strip mining

Acordando com a Lei de Amdahl, o *speedup* de programas com baixo a moderados níveis de paralelização (neste caso, vetorização) é muito limitado. A presença de condições, como blocos *if*, dentro de ciclos é uma das razões críticas para os baixos níveis de vetorização, dado que se introduzem dependências de dados. Consideremos o Código 3.3, por exemplo, que o demonstra.

código 3.3

```
for (i = 0; i < VL; i++) {
  if (X[i] != 0.0) {
    X[i] = X[i] - Y[i];
  }
}
```

O ciclo do Código 3.3 não pode ser vetorizado de uma forma linear devido à condição que está instituída dentro do ciclo *for*. No entanto, poderá ser feita uma execução de ciclo só para os casos que respeitem a condição `X[i] != 0.0`, onde a subtração poderia ser vetorizada.

Uma extensão própria para esta capacidade é denominada de **controlo de máscara comum** (*common mask control*). Registos de máscara fornecem então a execução condicional de cada operação elementar numa operação vetorial. O vetor de controlo de máscara usa assim uma variável booleana com uma condição para verificar se uma instrução escalar deve, ou não, ser executada. Quando o registo-vetor máscara (VM) é ativo, qualquer instrução vetorial opera somente nos elementos cujas entradas correspondentes são uma, no VM. Ao limpar o VM, isto é, colocando todos os seus bits a '1', tornam-se as instruções vetoriais sobre vetores subsequentes a operar em todos os elementos de vetores.

controlo de máscara comum

Ao assumir que os endereços de início de regiões de memória onde os vetores X e Y estão guardados estão nos registos `rx` e `ry`, respetivamente, o ciclo anterior poderá ser codificado no Código 3.4.

código 3.4

```
lv      v1, rx
lv      v2, ry
l.d     f0, 0
snevs.d v1, f0
subvv.d v1, v1, v2       ; coloca VM[i] a 1, se V1[i] != 0
sv      rx, v1
```

A execução condicional tem algum *overhead*, exposto pela necessidade de executar a instrução `S__VS`. No entanto, mesmo com um número significativo de zeros no VM, usando o controlo de máscara ainda podemos ter melhores desempenhos com o modo escalar.

Em termos de bancos de memória, o comportamento de unidades vetoriais para *load/store* é um pouco mais complicado que a unidade funcional aritmética. Aqui o tempo

de *start up* é o tempo necessário para obter a primeira palavra da memória para um registo ou de um registo para a memória. Se o resto do vetor puder ser atribuído sem necessidade de inserção de *stalls*, então a frequência de iniciação é igual à frequência a que as novas palavras são carregadas ou guardadas. Diferentemente de outras unidades funcionais, a frequência de iniciação não será necessariamente um ciclo de relógio, isto porque os bancos de memória poderão ter *stalls* que podem reduzir o *thoughput* efetivo. Assim, para atenuar este problema são usadas caches, nos processadores mais atuais, de forma a fazer descer o valor das latências de carregamentos e transferências vetoriais.

Então mas com tudo isto estudado, como é que manipulamos vetores multidimensionais? Ora, elementos sucessivos de um vetor poderão não estar em posições contíguas de memória. À distância que separa os elementos a serem colecionados dá-se o nome de **stride**. Uma vez o vetor carregado num registo-vetor, este atua como se os seus elementos sucessivos estivessem adjacentes, isto é, um processador vetorial poderá tratar *strides* maiores que 1, sendo que existem instruções de *load* e de *store* capazes de modificar o *stride*. Esta capacidade de aceder a posições de memória não-sequencial e de modelar o seu conteúdo numa estrutura mais densa é uma das maiores vantagens de um processador vetorial.

stride

O *stride* vetorial, como o endereço inicial de um vetor, podem ser carregados num registo geral e usado depois numa instrução de *load* ou *store* mais especial. No VMIPS a instrução `lvws` (*load vector with stride*) permite que seja feito o *fetch* dos elementos de um vetor, separados na memória por um *stride* maior que 1, num registo-vetor. Da mesma forma, a instrução `svws` (*store vector with stride*) faz o mesmo, mas na direção oposta.

O suporte de *strides* maiores que 1 complica, contudo, o sistema de memória. Uma vez que *strides* não-unitários são introduzidos, torna-se possível pedir acesso ao mesmo banco de memória frequentemente. Quando múltiplos acessos apontam para um banco em particular, um conflito ocorre, pelo que há um *stall* de todos os acessos, exceto um. Suponhamos assim que temos 8 bancos de memória com um *bank busy time* de 6 ciclos de relógio e suponhamos também que a latência de memória é de 12 ciclos de relógio. Quanto tempo demorará a ficar completo um *load* de um vetor de 64 elementos com um *stride* de 1 e com um *stride* de 32? Ora, para um *stride* de 1 teríamos um tempo de $12 + 64 = 76$ ciclos de relógio (ou 1.2 ciclos de relógio por elemento) e, para um *stride* de 32 teríamos $12 + 1 + 6 \times 63 = 391$ ciclos de relógio, isto é, 6.1 ciclos de relógio por elemento.

E assim terminam os apontamentos para a disciplina de Arquitetura de Computadores Avançada (a4s1), sendo que fazemos ponte com disciplinas como Computação Visual (a4s1) ou Computação Reconfigurável (a4s2).

1. Avaliação do Desempenho dos Sistemas de Computação

2. Paralelismo de Instruções

3. Paralelismo de dados

As referências abaixo correspondem às várias citações (quer diretas, indiretas ou de citação) presentes ao longo destes apontamentos. Tais referências encontram-se dispostas segundo a norma IEEE (as páginas Web estão dispostas de forma análoga à de referências para livros segundo a mesma norma).

[1] A. R. Borges, "Slides da disciplina de Arquitetura de Computadores Avançada - General Description," ed. Universidade de Aveiro, 2016.

[2] H. Sunami, "Dimension Increase in Metal-Oxide-Semiconductor Memories and Transistors," em *Advances in Solid State Circuits Technologies*, Croácia: INTECH, pp. 446 - 474.

[3] C. Starr, C. A. Evers, and L. Starr, *Biology: Concepts and Applications*. Thomson: Brooks/Cole, 2006.

[4] N. Lau, "Slides da disciplina de Arquitetura de Computadores Avançada - Apresentação," ed. Universidade de Aveiro, 2015.

[5] A. R. Borges, "Slides da disciplina de Arquitetura de Computadores Avançada - Computer Abstractions and Technology," ed. Universidade de Aveiro, 2016.

[6] A. R. Borges, "Dependability," ed. Aveiro: Universidade de Aveiro, 2016.

[7] A. R. Borges, "Slides da disciplina de Arquitetura de Computadores Avançada - Instruction-Level Parallelism (Principles)," ed. Universidade de Aveiro, 2016.

[8] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5 ed. Waltham, MA: Elsevier, 2012.

[9] A. R. Borges, "Slides da disciplina de Arquitetura de Computadores Avançada - Instruction-Level Parallelism (Complements)," ed. Universidade de Aveiro, 2016.

[10] A. R. Borges, "Ficha Prática - Branch Prediction," ed. Universidade de Aveiro, 2016.

[11] A. R. Borges, "Slides da disciplina de Arquitetura de Computadores Avançada - Data-Level Parallelism," ed. Universidade de Aveiro, 2016.

Apontamentos de Arquitetura de Computadores Avançada
1ª edição - janeiro de 2017

aca

Autor: Rui Lopes
Agradecimentos: professor António Rui Borges

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas. Grande parte do texto deste documento é uma adaptação dos conteúdos fornecidos nas aulas desta disciplina, para português.

apontamentos