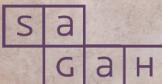


ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Aline Zanin



SOLUÇÕES
EDUCACIONAIS
INTEGRADAS

Execução (*pipeline*)

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Descrever os problemas no uso de *pipeline*.
- Discutir o desempenho do *pipeline*.
- Reconhecer a diferença entre *pipeline*, *superpipeline* e *pipeline* superescalar.

Introdução

O *pipeline* é um modelo de paralelismo utilizado para aumentar o desempenho na execução de instruções. Se um conjunto de instruções fosse executado sequencialmente, com uma tarefa após a outra, seria necessário muito tempo até a conclusão da execução de todas as instruções. Já com o *pipeline*, a execução das instruções pode ser sobreposta, reduzindo o tempo previsto para a sua conclusão.

Embora o *pipeline* traga vantagens com relação à execução sequencial das instruções, a sua utilização também traz alguns problemas e desafios. Neste capítulo, você vai aprender sobre os problemas encontrados no uso de *pipeline* e as técnicas que podem ser empregadas para resolvê-los. Você também vai estudar sobre como avaliar o desempenho da execução de um *pipeline* e como melhorar esse desempenho com a aplicação de outras técnicas de paralelismo, conhecidas como *superpipeline* e *pipeline* superescalar. Além disso, você compreenderá as diferenças entre esses dois tipos de paralelismo e o *pipeline* convencional, apontando as vantagens e desvantagens de cada um.

1 O uso de *pipeline*

Você pode pensar no *pipeline* como uma linha de montagem de automóveis, na qual as instruções são as tarefas que devem ser executados. Enquanto uma equipe trabalha na pintura, outra trabalha no sistema elétrico e outra, no

motor. Se esse processo fosse executado de forma sequencial por apenas uma pessoa, o tempo para a montagem do automóvel seria muito maior. O mesmo ocorre ao empregar o *pipeline*, que, embora seja útil para acelerar o processo de execução das instruções, também apresenta alguns problemas que tornam a sua aplicação um pouco menos eficiente na prática (WANDERLEY NETTO, 2005). Na prática, há certo desperdício com a passagem das instruções pelos estágios do *pipeline*, causando um problema no desempenho da execução. Porém, ainda existem problemas mais sérios que podem deteriorar a técnica de *pipeline* (ASSIS, 2017). Nesta seção, você vai conhecer quais são os problemas encontrados na execução de um *pipeline* e aprenderá como resolvê-los.

O conflito de recursos

Considere um *pipeline* com os seguintes estágios executados (STALLINGS, 2005; WANDERLEY NETTO, 2005; WEBER, 2012):

- um estágio de busca com acesso à memória de instruções para ler a instrução que deve ser executada (*fetch* – F), que gasta 6 nanossegundos para ser executado;
- um estágio de leitura dos registradores enquanto a instrução é decodificada a partir de uma lógica combinacional (*decode* – D), que gasta 2 nanossegundos para ser executado;
- um estágio de envio dos dados para a unidade lógica e aritmética (ALU, do termo em inglês *Arithmetic Logic Unit*) com o objetivo de executar a operação ou calcular um endereço (*execution* – E), que gasta 4 nanossegundos para ser executado;
- um estágio para acessar a memória de dados e realizar a leitura ou escrita de um operando (*memory* – M), que gasta 6 nanossegundos para ser executado; e
- o último estágio, de escrita do resultado no registrador (*write* – W), que gasta de 2 nanossegundos para ser executado.

Nesse caso, o ciclo de *clock* é de 6 nanossegundos, pois os estágios de acesso às memórias F e M demandam mais tempo para serem executados, sendo que ambos gastam, exatamente, 6 nanossegundos cada para serem executados (STALLINGS, 2005; WANDERLEY NETTO, 2005).



Fique atento

A unidade lógica e aritmética (ULA) — em inglês, *Arithmetic Logic Unit* (ALU) — é o *hardware* que realiza adição, subtração e, normalmente, operações lógicas, como AND e OR (PATTERSSON; HENNESSY, 2005).

A Figura 1 demonstra esse *pipeline* após a execução do seguinte conjunto de instruções:

- add;
- lw;
- beq; e
- sub.

Na prática, essa execução gera um problema chamado de conflito de recursos, uma vez que o *pipeline* tenta utilizar o mesmo componente por mais de uma instrução ao mesmo tempo. Exemplos de componentes ou recursos incluem memórias, caches, barramentos, bancos de registradores e ALU (STALLINGS, 2005). Nesse caso, o *pipeline* tenta escrever no banco de registradores para a instrução **add** e ler o banco de registradores para a instrução **sub** ao mesmo tempo. Assim, ocorre o conflito de recursos, pois o banco de registradores é capaz apenas de ler duas instruções, mas não de ler e escrever ao mesmo tempo (WANDERLEY NETTO, 2005).



Saiba mais

O *clock* é a frequência de processamento de um processador. Para compreender melhor o funcionamento do ciclo de *clock* de processadores, assista ao vídeo *Processador Clock e Frequência* do canal, do canal Dicionário de Informática, no YouTube.

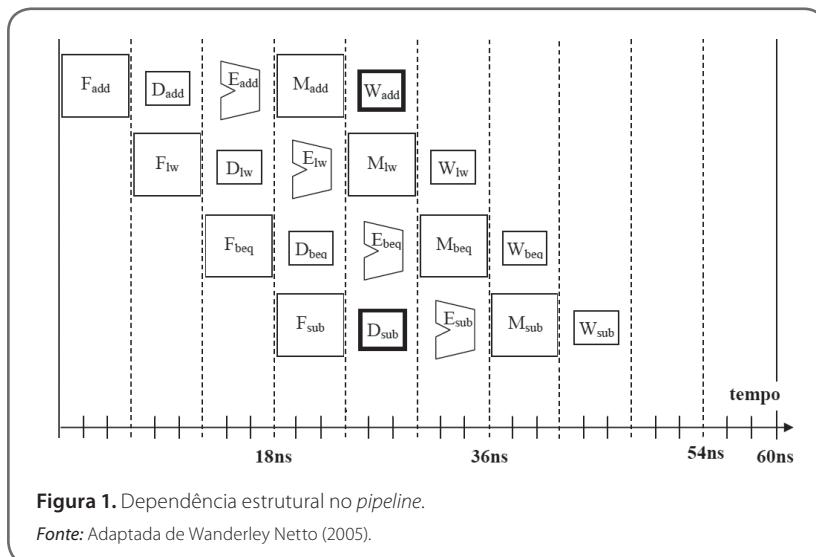


Figura 1. Dependência estrutural no *pipeline*.

Fonte: Adaptada de Wanderley Netto (2005).

Nesse exemplo, você pode resolver o problema ao realizar uma escrita e uma leitura dos dados dentro do mesmo período de *clock*. Na Figura 1, você pode observar que o ciclo de *clock* é de 6 nanossegundos. Logo, como a escrita e a leitura do banco de registradores ocorre em 2 nanossegundos, é possível realizá-las dentro do mesmo período de *clock*. Dessa forma, você pode utilizar a primeira metade do ciclo de *clock* para realizar a escrita no banco de registradores e a segunda metade para realizar a leitura nesse banco (WANDERLEY NETTO, 2005).

O conflito de dados

Existe outro problema que pode ser encontrado na execução de um *pipeline*. A Figura 2 ilustra um *pipeline* com a execução das instruções **add** e **sub**. Se você tentar executar as seguintes instruções conforme o trecho de código abaixo, encontrará um problema denominado **conflito de dados** (WANDERLEY NETTO, 2005).

```
add $8, $9, $10
sub $11, $12, $8
```

Nesse código, a instrução **sub** utiliza como operando o resultado da instrução **add**. Observando a Figura 2, você vai perceber que a instrução **sub**

precisa do resultado da instrução **add** no terceiro ciclo de *clock* para poder executar o estágio **decode**. Porém, o resultado de **add** só estará disponível no quinto ciclo de *clock*. Logo, ocorre uma dependência de dados na execução do *pipeline* (WANDERLEY NETTO, 2005).

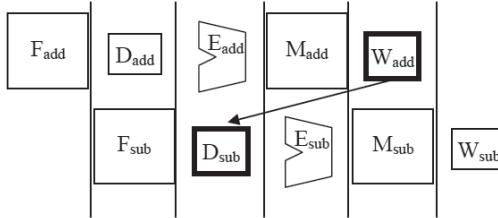


Figura 2. Dependência de dados no *pipeline*.

Fonte: Wanderley Netto (2005, p. 113).

Para resolver uma dependência de dados entre as instruções **add** e **sub**, a execução da instrução **sub** deve ser atrasada de acordo com o número de ciclos de *clock* necessários até remover a dependência. Nesse caso, a instrução **sub** só poderá ser executada após a instrução **add** produzir os dados de entrada para a instrução **sub**. Em geral, para que você consiga resolver um problema de dependência de dados, é necessário atrasar a instrução que causa a dependência até que todos os seus dados de entrada tenham sido produzidos (STALLINGS, 2005). Para atrasar a execução da instrução, você pode utilizar um conceito oficialmente conhecido como ***pipeline stall***, mais conhecido como **bolha**. Como a instrução **add** não escreve o seu resultado até o quinto estágio, você precisa acrescentar três bolhas ao *pipeline* (PATTERSON; HENNESSY, 2005).

Outra solução para esse caso é chamada de *forwarding* ou *bypassing* e fundamenta-se na observação de que você não precisa esperar até o término da execução da instrução antes de resolver o conflito de dados. Para solucionar o exemplo anterior, você pode realizar a execução do estágio (E) da instrução **add** e fornecê-la diretamente como entrada para a subtração. Para fornecer o item que falta antes do previsto, diretamente dos recursos internos, é necessário acrescentar um *hardware* extra. A Figura 3 ilustra essa solução, sendo que a conexão mostra o caminho do *forwarding* desde a saída do estágio E da instrução **add** até a entrada do estágio E para a instrução **sub**, substituindo o valor do registrador \$8 lido no segundo estágio de **sub** (PATTERSON; HENNESSY, 2005).

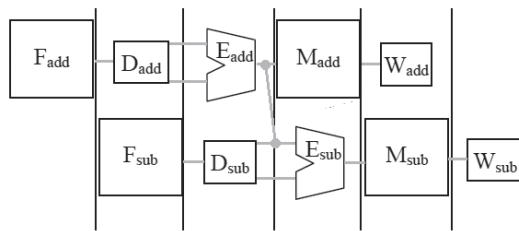


Figura 3. Representação gráfica do *forwarding*.

Fonte: Adaptada de Patterson e Hennessy (2005).

A técnica de *forwarding* é uma boa solução para o conflito de dados, porém não consegue evitar a inserção de bolhas em todos os casos. Por exemplo, se você tivesse uma instrução **load** \$8 em vez de uma instrução **add**, os dados necessários para executar a instrução **sub** só estariam disponíveis após o quarto estágio da instrução **load**. Isso significa que o valor de \$8 só estaria disponível após ser carregado na memória no estágio M da instrução **load**. Como a instrução **sub** precisa do dado no estágio E, ainda que você utilize a técnica de *forwarding*, deverá atrasar a execução de um estágio dessa instrução. Como a instrução **load** não escreve o seu resultado até o quarto estágio, você precisaria apenas acrescentar uma bolha ao *pipeline*. A Figura 4 ilustra essa solução. Sem a utilização de bolhas, a saída do estágio M da instrução **add** como entrada do estágio E da instrução **sub** estaria ao contrário no tempo e, portanto, seria impossível acontecer (PATTERSON; HENNESSY, 2005).

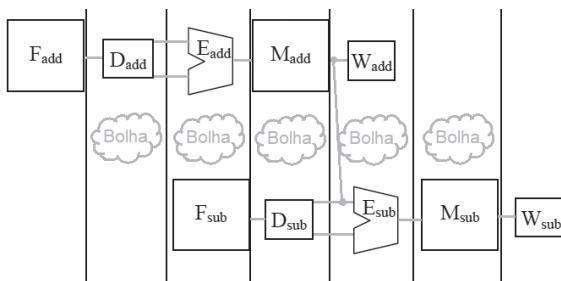


Figura 4. Representação gráfica de um *pipeline tall* com *forwarding*.

Fonte: Adaptada de Patterson e Hennessy (2005).

O conflito de controle

Outro tipo de problema muito comum é o chamado **conflito de controle**, que acontece quando um *pipeline* executa uma instrução de salto ou desvio condicional. Se uma instrução de salto for executada, o processador só vai saber se a condição de salto foi ou não atingida no estágio de execução do *pipeline* (E). Se o salto for tomado, as instruções que seguem no *pipeline* deverão ser descartadas e haverá penalidades, tais como a perda de desempenho (WANDERLEY NETTO, 2005). Em síntese, o conflito de controle nada mais é do que a necessidade de tomar uma decisão com base nos resultados de uma instrução (PATTERSON; HENNESSY, 2005).

A Figura 5 ilustra um *pipeline* com a execução de um conjunto de instruções. Se a instrução de salto **beq** fosse tomada, o *pipeline* deveria executar a próxima instrução **sw**. Caso **beq** não fosse tomada, o *pipeline* seguiria com a execução da instrução **lw**. Como o teste só foi realizado no final do estágio de execução (E), as instruções **lw** e **sub**, que viriam a seguir, foram descartadas e a instrução correta passou a ser buscada (WANDERLEY NETTO, 2005).

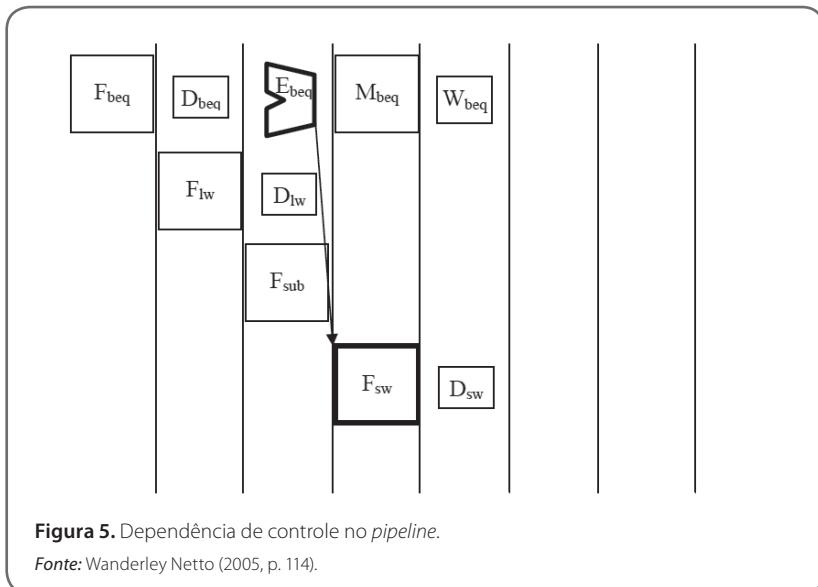


Figura 5. Dependência de controle no *pipeline*.

Fonte: Wanderley Netto (2005, p. 114).

Você dispõe de três alternativas para resolver o problema do conflito de controle, sendo que a primeira é a utilização de bolhas. Como o processador só vai saber se a condição de salto foi ou não atingida no estágio de execução

do *pipeline* (E), basta executar a instrução **beq** sequencialmente até você descobrir o resultado do desvio para saber de qual endereço na memória deve-se apanhar a próxima instrução. Porém, essa técnica é lenta. Supondo que você adicione um *hardware* extra capaz de testar os registradores, calcular o endereço do desvio e atualizar as informações durante o segundo estágio de *pipeline*, ainda seria necessário inserir uma bolha. Conforme pode ser observado na Figura 6, se o desvio fosse tomado, a instrução **sw** precisaria ter a sua execução atrasada no *pipeline* (PATTERSON; HENNESSY, 2005).

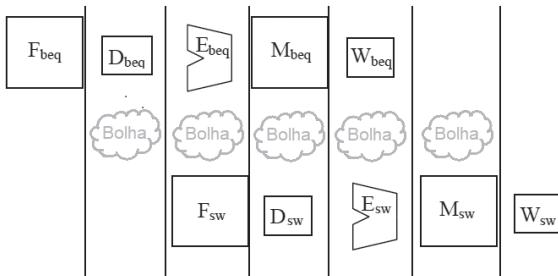


Figura 6. Representação gráfica de um *pipeline* com *stall* em desvio condicional como solução para controlar um conflito de controle.

Fonte: Adaptada de Patterson e Hennessy (2005).

Outra forma de resolver esse problema reside na previsão de desvio, que consiste em tentar prever se os saltos ou desvios serão tomados ou não. Uma técnica muito utilizada na previsão de desvio é a previsão dinâmica, na qual você mantém o histórico de cada desvio tomado ou não tomado e, em seguida, usa os dados resultantes desse comportamento para prever os desvios futuros. Nesse sentido, as **previsões de desvio dinâmico** conseguem prever desvios de maneira adequada, com uma precisão superior a 90%, devido à grande quantidade de dados e ao tipo de histórico já mantidos. No entanto, se a previsão estiver errada, o *pipeline* deverá ser reiniciado a partir do endereço de desvio correto (PATTERSON; HENNESSY, 2005).

Por fim, a terceira alternativa para solucionar o problema do conflito de controle é a decisão adiada (*delayed branch*). Com essa técnica, sempre que houver um salto, você poderá adiar uma instrução que não depende do resultado desse salto. Com a técnica de decisão adiada, a próxima instrução sempre é executada sequencialmente, de modo que o salto ocorre após o atraso criado com a execução da instrução. No exemplo da Figura 6, caso existisse uma

instrução antes da instrução de salto **beq** que não dependesse do seu resultado, ela poderia ser movida para depois desse salto. Essa solução esconde o atraso de instruções de salto e, assim, é útil para lidar com desvios curtos de um ciclo de *clock*. Quando o atraso criado por um salto é maior do que um ciclo de *clock*, normalmente emprega-se a previsão de desvio baseada em *hardware* (PATTERSON; HENNESSY, 2005).

2 O desempenho do *pipeline*

Como você já estudou ao longo deste capítulo, o uso de *pipeline* de instruções melhora o desempenho na execução dessas instruções de forma sequencial. Nesta seção, você vai aprender algumas medidas para avaliar o desempenho do processo de execução de instruções com o uso de *pipeline* (STALLINGS, 2005).

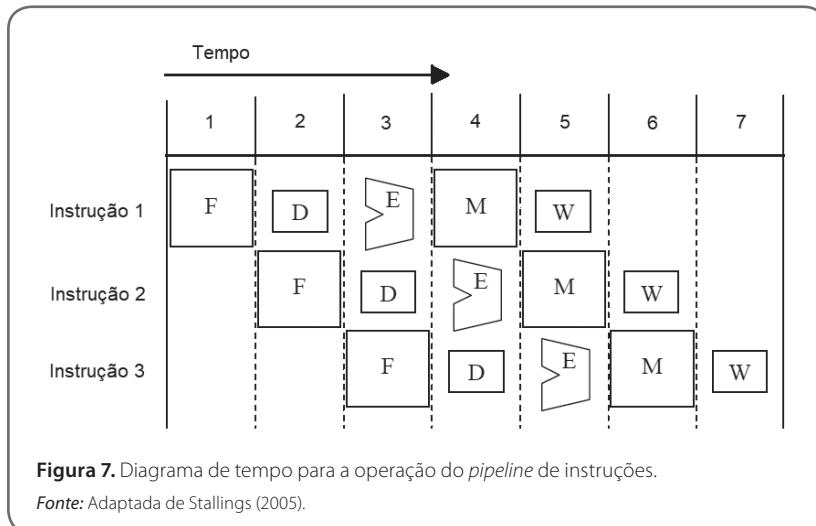
O tempo necessário para executar um conjunto de instruções em um *pipeline* é chamado de tempo de ciclo. Cada uma das colunas da Figura 7 representa o tempo de um ciclo τ , que pode ser calculado a partir da seguinte equação:

$$\tau = \max[\tau_i] + d = \tau_m + d \quad i, 1 \leq i \leq k$$

Na equação referida, tem-se que:

- τ_m é o atraso máximo de estágio, que é igual ao atraso do estágio de maior atraso;
- k é o número de estágios do *pipeline* de instruções; e
- d é o tempo necessário para enviar os sinais e os dados do estágio anterior ao seguinte.

Geralmente, o tempo de atraso d é igual a um pulso de *clock* e $\tau_m \gg d$ (STALLINGS, 2005).



Suponha que você executou n instruções em um *pipeline* sem que ocorresse nenhum salto. O tempo total T_k requerido para executar as n instruções é determinado pela seguinte equação (STALLINGS, 2005):

$$T_k = [k + (n - 1)]\tau$$

Na equação referida, tem-se que:

- k é o total de ciclos necessários para completar a execução da primeira instrução;
- d é o número da instrução; e
- τ é o tempo de ciclo de um *pipeline*, que será desconsiderado nesse exemplo.

Para verificar essa equação, basta observar a Figura 7, na qual a primeira instrução é completada em cinco ciclos, de modo que:

$$[5 + (1 - 1)] = 5$$

Utilizando a mesma fórmula, você consegue verificar que a segunda instrução é completada em seis ciclos e a terceira, em sete ciclos:

$$[5 + (2 - 1)] = 6$$

$$[5 + (3 - 1)] = 7$$

Se houvesse uma nona instrução, ela seria completada em 13 ciclos, posto que:

$$[5 + (9 - 1)] = 13$$

Para calcular o desempenho da execução com *pipeline* com relação à execução sequencial das instruções sem *pipeline*, utiliza-se o *speedup*. Essa medida representa o quanto a versão com *pipeline* é melhor se comparada à versão sem esse tipo de paralelismo. O *speedup* é a razão de desempenho da máquina sequencial e o desempenho da máquina *pipeline*, o qual pode ser calculado a partir da seguinte equação (STALLINGS, 2005):

$$\text{Speedup} = \frac{T_1}{T_k}$$

Contudo, é mais comum realizar o cálculo do *speedup* a partir da equação conhecida como **Lei de Amdahl**, que define que o *speedup* de um sistema depende do *speedup* de um componente em particular, além de depender do tempo em que esse componente é utilizado pelo sistema. A fórmula da Lei de Amdahl é a seguinte (WANDERLEY NETTO, 2005):

$$\text{Speedup} = \frac{1}{(1 - f) + f/k}$$

Na equação referida, tem-se que:

- f é a fração de tempo em que o componente em particular é utilizado; e
- k é o *speedup* do componente em particular.

Um fator que, teoricamente, pode aumentar o desempenho do *pipeline* é o número de estágios. A Figura 8 apresenta um gráfico do *speedup* em função do número de instruções executadas sem a ocorrência de saltos. O aumento do número de estágios do *pipeline* aumenta o desempenho, cujo fator de aceleração é igual a k .

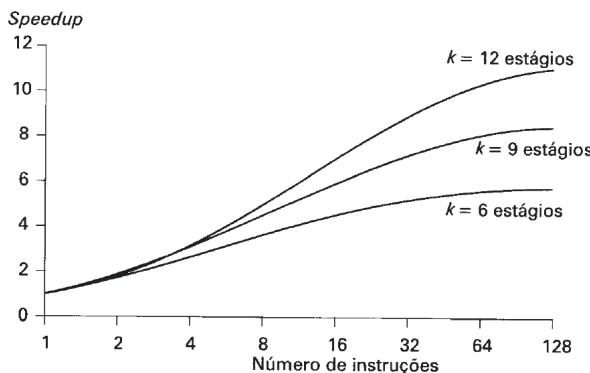


Figura 8. Speedups obtidos com o uso de *pipeline* de instruções.

Fonte: Adaptada de Stallings (2005).

A Figura 9 apresenta um gráfico do *speedup* em função do número de estágios do *pipeline*. Nesse caso, o fator de aceleração aproxima-se do número de instruções que podem ser executadas no *pipeline* sem a ocorrência de saltos. Logo, quanto maior for o número de estágios em um *pipeline*, maior será o seu *speedup*. Porém, na prática, esse ganho é minimizado com o aumento do custo causado por atrasos entre estágios e pelo esvaziamento do *pipeline* quando ocorrem saltos. Assim sendo, utilizar mais de nove estágios em um *pipeline* não traria ganhos com relação ao desempenho (STALLINGS, 2005). Além disso, há formas mais efetivas de aumentar o desempenho do *pipeline*, tais como o *superpipeline* e o *pipeline* superescalar, modelos de paralelismo que você conhecerá em detalhes na próxima seção deste capítulo.

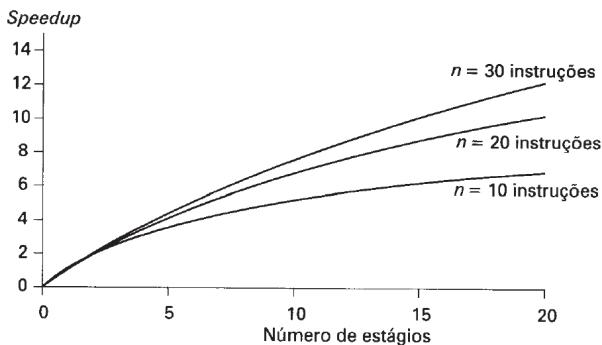


Figura 9. Speedups obtidos com o uso de *pipeline* de instruções.

Fonte: Adaptada de Stallings (2005).



Fique atento

Gene Amdahl (1922 – 2015) foi um matemático reconhecido pelo desenvolvimento da fórmula para medir o *speedup*. Em 1967, Amdahl quantificou as suas observações sobre o desempenho e derivou a fórmula que leva o seu nome e configura a Lei de Amdahl (WANDERLEY NETTO, 2005).

3 Pipeline, superpipeline e pipeline superescalar

Além da execução convencional em *pipeline*, existem outros modelos de paralelismo que podem ser explorados, tais como o *superpipeline* e o *pipeline* superescalar. Nesta seção, você vai estudar em maiores detalhes como esses dois tipos de *pipeline* funcionam, quais são as diferenças entre eles e qual deles apresenta o melhor desempenho se comparado aos outros.

Como você já aprendeu anteriormente, o *pipeline* é uma técnica de implementação na qual várias instruções são sobrepostas na sua execução. A técnica de *pipeline* demanda muito menos tempo para executar seis instruções sobrepostas, visto que gasta apenas 60 nanossegundos nessa execução, com um ciclo de *clock* de 6 nanossegundos, conforme mostra a Figura 10 a seguir (PATTERSON; HENNESSY, 2005). Afinal, a execução dessas seis instruções

sequencialmente, uma após a outra, gastaria 120 nanossegundos. No entanto, embora o *pipeline* ofereça um melhor desempenho com relação à execução das instruções em sequência, existem outros modelos de paralelismo que conseguem fornecer um desempenho ainda melhor, tais como o *superpipeline* e o *pipeline* superescalar.

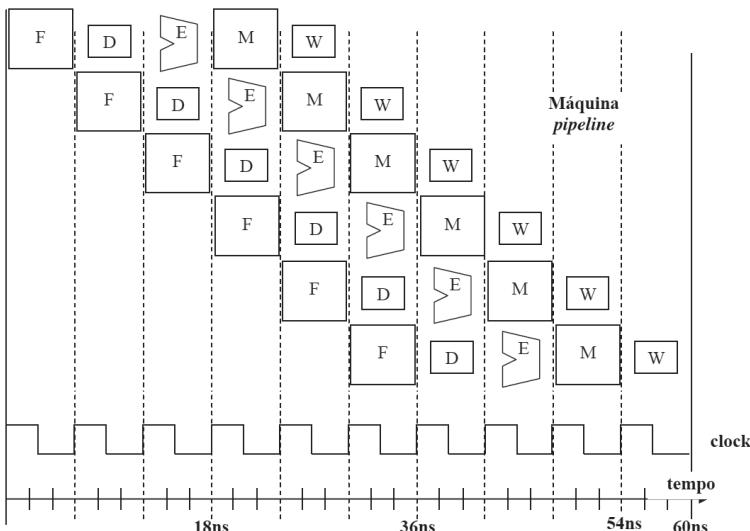


Figura 10. Máquina *pipeline* convencional.

Fonte: Adaptada de Wanderley Netto (2015).

O *superpipeline* surgiu da observação de que muitas tarefas em um estágio de *pipeline* convencional podem ser quebradas em subtarefas, permitindo que você aumente a frequência de operação e produza mais resultados por ciclos de *clock*. Por essa razão, o *superpipeline* é considerado um refinamento da estrutura do *pipeline* que utiliza um número maior de estágios. Com mais estágios, mais instruções podem estar no *pipeline* ao mesmo tempo.

A Figura 11 ilustra um modelo de *superpipeline* executando seis instruções, com um ciclo de *clock* de 6 nanossegundos. Nesse exemplo, a tarefa realizada em cada estágio é quebrada em duas partes, de modo a permitir que um mesmo ciclo de *clock* realize duas operações ao mesmo tempo. Dessa forma, o modelo *superpipeline* consegue concluir as seis tarefas mais rápido do que o *pipeline* convencional (STALLINGS, 2005; WANDERLEY NETTO, 2005). O *superpipeline* da Figura 11 conseguiu executar em 48 nanossegundos o

mesmo número de instruções do *pipeline* convencional que você visualizou na Figura 10.

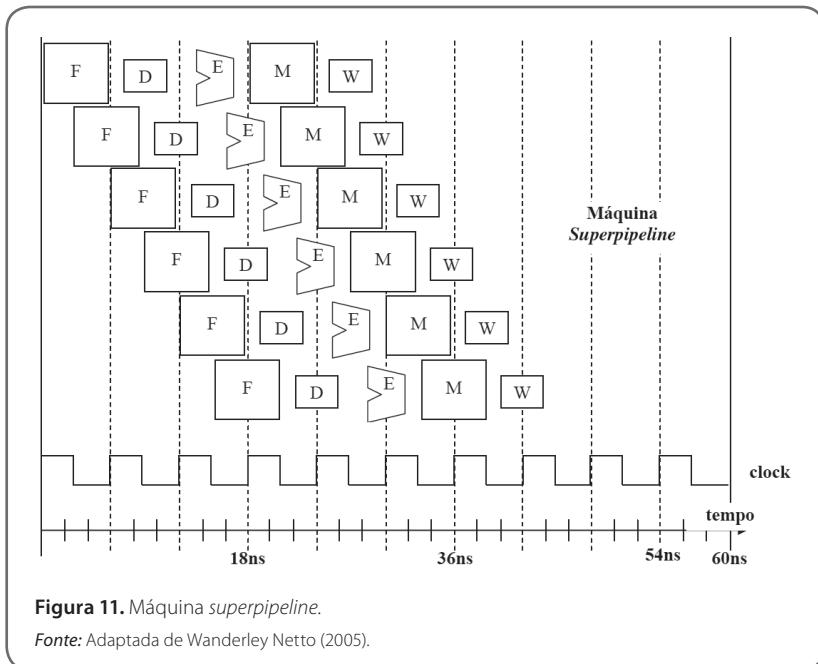


Figura 11. Máquina superpipeline.

Fonte: Adaptada de Wanderley Netto (2005).

Outro modelo de paralelismo é o *pipeline* superescalar, que consiste em uma técnica de *pipeline* avançada que permite ao processador executar mais de uma instrução por ciclo de *clock*, uma vez que o processador seleciona as instruções durante a execução (PATTERSON; HENNESSY, 2005). Em suma, a máquina superescalar replica cada um dos estágios do *pipeline* para que seja possível processar de forma simultânea duas ou mais instruções em um mesmo estágio (STALLINGS, 2005).

A Figura 12 ilustra um modelo de *pipeline* superescalar de dois níveis executando seis instruções, também com um ciclo de *clock* de 6 nanossegundos. Os níveis do *pipeline* superescalar indicam o número de instruções que serão tratadas por vez (WANDERLEY NETTO, 2005). O *pipeline* superescalar da Figura 12 conseguiu executar em 42 nanossegundos o mesmo número de instruções do *pipeline* convencional da Figura 10 e do *superpipeline* da Figura 11.

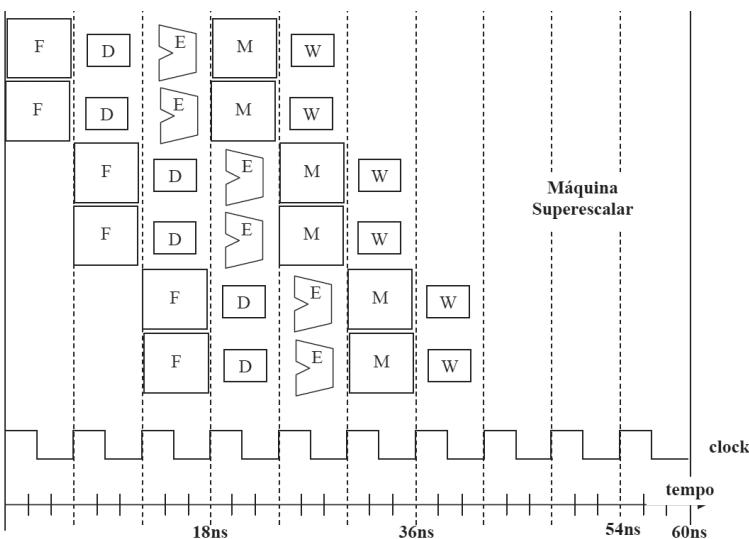


Figura 12. Máquina *pipeline* superescalar.

Fonte: Adaptada de Wanderley Netto (2005).

Tanto o *superpipeline* como o *pipeline* superescalar conseguem obter um desempenho melhor do que o *pipeline* convencional, já que reduzem o tempo necessário para a execução de todas as instruções. Porém, ambas as abordagens apresentam as suas desvantagens. A desvantagem da abordagem *superpipeline* está na sobrecarga criada durante a transferência de instruções de um estágio a outro. Por sua vez, o *pipeline* superescalar pode apresentar retardos na execução das instruções devido ao problema de dependência entre instruções que são executadas em *pipelines* diferentes, ao que chamamos conflito de dados. Esse conflito de dados não permite explorar ao máximo o paralelismo de instrução de uma máquina superescalar. Além disso, o *pipeline* superescalar exige uma lógica especial na organização do fluxo de instruções para coordenar as dependências (STALLINGS, 2005). Apesar das desvantagens apresentadas, o *pipeline* superescalar ainda é o mais vantajoso, pois é capaz de produzir mais instruções por ciclo de *clock* (WANDERLEY NETTO, 2005).



Referências

ASSIS, M. R. M. *Informática para concursos públicos de informática*. São Paulo: Novatec, 2017.

PATTERSON, D. A.; HENNESSY, J. L. *Organização e projeto de computadores*. 4. ed. Rio de Janeiro: Elsevier, 2005.

STALLINGS, W. *Arquitetura e organização de computadores: projeto para o desempenho*. 5. ed. São Paulo: Prentice Hall, 2005.

WANDERLEY NETTO, E. B. *Arquitetura de computadores: a visão do software*. Natal: CEFET-RN, 2005.

WEBER, R. F. *Fundamentos de arquitetura de computadores*. 4. ed. Porto Alegre: Bookman, 2012. (Livros Didáticos Informática UFRGS, v. 8).

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:



SOLUÇÕES
EDUCACIONAIS
INTEGRADAS