Evolução do desempenho do processador

MAC0344 - Arquitetura de Computadores
Prof. Siang Wun Song
Slides usados: https://www.ime.usp.br/∼song/mac344/slides03-performance.pdf

Baseado parcialmente em W. Stallings -
Computer Organization and Architecture

Evolução do desempenho do processador

- Veremos nessas próximas aulas a evolução do desempenho do processador. Será passada a Lista 3 de exercícios.
- Ao final dessas aulas, vocês saberão
  - Aumentar a frequência do relógio tem o problema de dissipação de calor. Então outras técnicas devem ser investigadas para melhorar a velocidade do processador.
  - Ao longo dos anos, várias técnicas foram desenvolvidas para essa finalidade.
  - Várias dessas técnicas procuram atenuar o gargalo de von Neumann: pré-busca de instruções, VLIW (very Large Instruction Word), etc.
  - Várias técnicas procuram explorar o paralelismo: pipelining, processador superescalar, processadores multicore, execução foram de ordem ou de forma concorrente (se não houver dependência), etc.
  - Lei de Amdahl sobre a limitação da computação paralela.

Evolução do desempenho - frequência do relógio

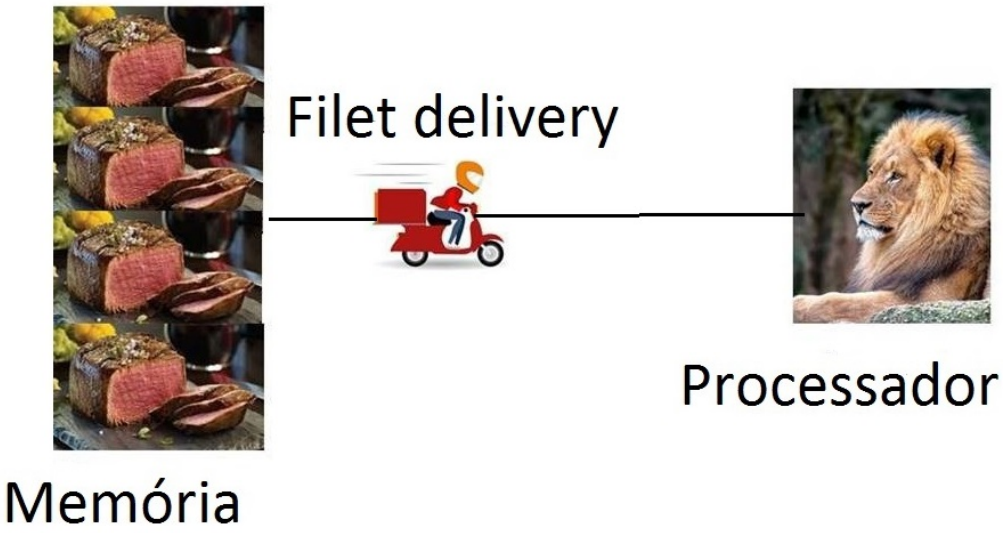
- O ENIAC (1946) tinha frequência de relógio de 100 KHz.
- O processador AMD FX-9590 tem frequência de relógio de 5 GHz.
- Aumento da frequência de relógio acarreta maior dissipação de calor. Por isso não se observa um aumento significativo na frequência do relógio ao longo do tempo.
- Portanto comparar frequências de relógio não é uma boa maneira de medir a evolução do desempenho.
- Uma melhor explicação para a evolução do desempenho é a tecnologia de circuitos integrados ou VLSI (*Very Large Scale Integration*) onde bilhões de transistores minúsculos ou mais são implementados uma pastila de silício.

Evolução do desempenho - tecnologia VLSI

- Lei de Moore: O número de transistores numa pastilha VLSI de silício vem dobrando a cada 18 meses. (Não é bem uma lei. Atualmente já leva mais de 18 meses para dobrar a capacidade da pastilha VLSI. Em breve poderá não valer mais.)
- Transistores menores significam não apenas maior capacidade mas também maior velocidade.
- A tecnologia VLSI viabilizou a chamada computação paralela: Hoje a computação paralela já é regra e não mais exceção.
- É necessário entretanto entender que o paralelismo pode ter suas limitações: veremos mais tarde a Lei de Amdahl.

Evolução do desempenho do processador

- O projeto do processador vem recebendo constantes melhorias visando maior desempenho: *pipelining* de instruções, processador superescalar, multicore, etc.
- Na arquitetura de von Neumann (usada até hoje), instruções e dados residem na memória prinicpal e precisam ser buscadas da memória e trazidas ao processador. Cria-se um gargalo conhecido como gargalo ou *bottleneck de von Neumann*.



Evolução do desempenho do processador - pipelining

Source: Ford assembly line 1933 - Wikipedia

Source: O Estado de São Paulo - Economia 28/08/2018

Vídeo Ford F-150 Assembly Line (4:41 minutos).

- *Pipelining* se assemelha a uma linha de montagem (*assembly line*).
- A execução de uma tarefa completa é dividida em estágios.
- Há uma estação separada para a execução de cada estágio.
- Pipelining possibilita a execução de diferentes estágios de várias tarefas ao mesmo tempo.
- Quando uma estação termina de executar o estágio de uma tarefa, ela passa a executar o mesmo estágio, mas da tarefa seguinte.
- A primeira tarefa leva o tempo normal para ser concluída. Mas a partir daí, uma nova tarefa é concluída logo após o seu último estágio.

Pipeline de instruções

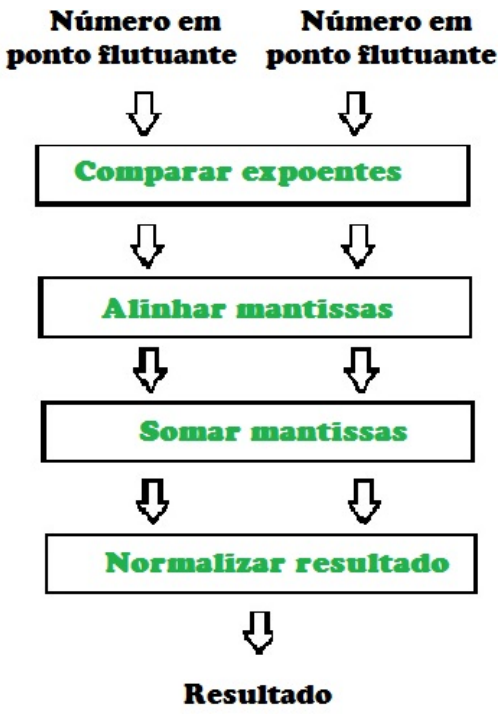
Ciclo	1	2	3	4	5	6	7	8	9	10
Busca instrução	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
Decodificação		I1	I2	I3	I4	I5	I6	I7	I8	I9
Endereço operando			I1	I2	I3	I4	I5	I6	I7	I8
Busca operando				I1	I2	I3	I4	I5	I6	I7
Execução					I1	I2	I3	I4	I5	I6
Escrita resultado						I1	I2	I3	I4	I5

Na figura, I1, I2, etc. são instruções.

- Em pipelining de instruções, a execução de uma instrução é realizada em vários estágios: busca da instrução, decodificação da instrução, determinação de endereço do operando, busca operando, execução propriamente dita e escrita do resultado.
- Quando há um desvio (*branch*), então pode ser necessário descartar toda uma pipeline já preenchida e recomeçar de novo. Veremos mais tarde a técnica de predicção de desvio.
- Ex.: Pipelining de instruções foi implementado já no Intel 80486.

Pipeline de operações aritméticas em ponto flutuante

Notação ponto flutuante normalizada: Um dígito não nulo antes do ponto decimal. Exemplo: $345.678 \rightarrow 3.45678 \times 10^2$



$$8.7 \times 10^0 + 9.9 \times 10^1$$

$$0.87 \times 10^1 + 9.9 \times 10^1$$

Ajustar o número menor

$$10.77 \times 10^1$$

$$1.077 \times 10^2$$

- Uma operação aritmética em ponto flutuante pode envolver vários estágios e ser executada através de uma pipeline.
- Exemplo: a soma de dois números em ponto flutuante pode dividida em quatro estágios. Isso agiliza a soma de dois vetores de números em ponto flutuante.

Técnica de pipelining

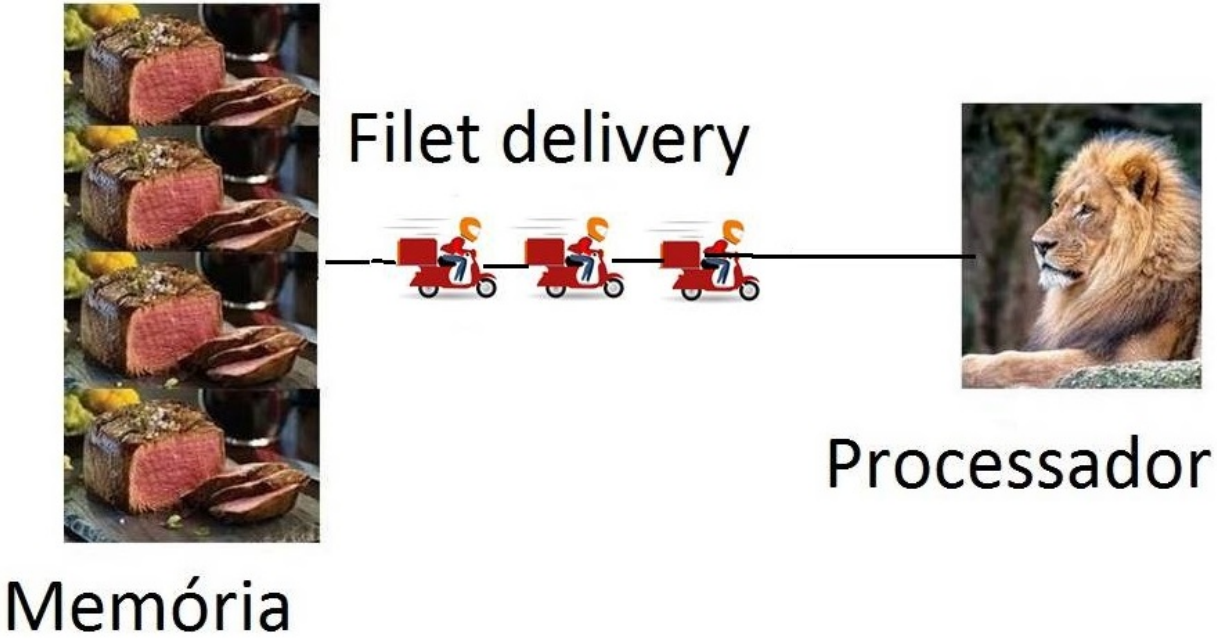
- Explora o paralelismo na execução de uma instrução.
- A execução de uma instrução é composta de várias etapas, mas uma etapa posterior depende da etapa anterior.
- Pipelining é uma forma de paralelizar a execução de etapas de diferentes instruções.

Evolução do desempenho do processador

- Diversas técnicas foram desenvolvidas para balancear a velocidade do processador com os demais componentes. Exemplos:
  - pré-busca de instruções
  - predicção de desvios
  - análise de fluxo de dados para verificar dependência de dados
  - execução especulativa
  - uso da memória cache (veremos mais tarde), etc.

Pré-busca de instruções

- Instruções são buscadas da memória e executadas no processador.
- Para deixar o processador mais ocupado possível, em geral instruções são pré-buscadas, ficando assim já disponível quando uma instrução precisa ser executada.



Predicção de desvios

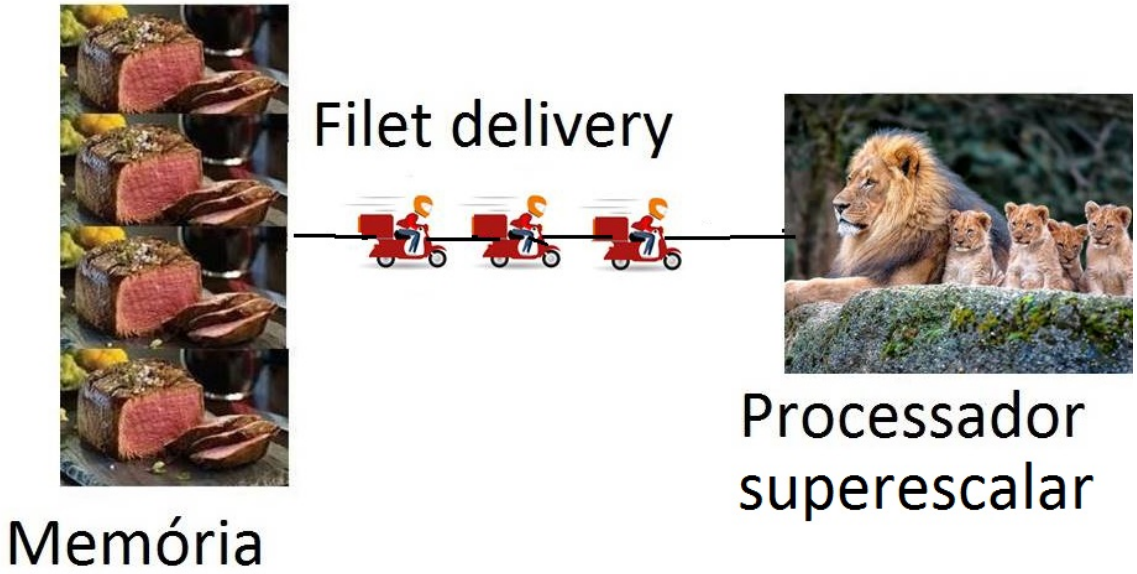
- No caso de um desvio, essa pré-busca encontra um problema, pois o ramo ou trecho a ser executado depois de um *if* depende de a condição estar satisfeita ou não (se executa o ramo *then* ou o ramo *else*). Exemplo:

  if condição **then** trecho 1
  else trecho 2

- Na predicção de desvio, o processador examina o código e procura prever, por exemplo baseado no passado, qual ramo ou trecho do desvio é mais provável para ser executado e já carrega as instruções deste trecho.
- Se a previsão for correta, então economiza-se o tempo de busca dessas instruções pois já estarão disponíveis. O processador fica então sempre ocupado.

Processador superescalar e análise do fluxo de dados

- O processador superescalar possui múltiplas unidades de execução de instruções: e.g. pode possuir duas unidades para realizar operações em inteiros e duas para ponto flutuante.
- Um processador superescalar explora o que é conhecido como paralelismo no nível de instrução.



Paralelismo no nível de instrução

- Processador superescalar possui múltiplas unidades de execução de instruções.
- Viabiliza assim o paralelismo no nível de instrução: várias instruções podem ser executadas ao mesmo tempo.
- Mas isso depende de ausência de dependência de uma instrução na outra. Veremos isso com mais cuidado.

Processador superescalar e análise do fluxo de dados

- Uma limitação fundamental para este tipo de paralelismo é a dependência de dados.
- Com análise do fluxo de dados, o processador verifica quais instruções dependem dos resultados de outras.
- Instruções independentes podem ser assim escalonadas para execução fora da ordem, ou até em paralelo, aproveitando os recursos de hardware existentes.

Exemplo: Essas operações não podem ser executadas fora de ordem:

$$A = X + Y$$
$$B = 2 \times A$$
$$C = B - A$$

Exemplo: podem ser executadas em qualquer ordem ou em paralelo:

$$A = X + Y$$
$$B = Z + 1$$
$$C = X \times Z$$

Três tipos de dependências de dados

- Dependência verdadeira ou de fluxo
- Anti-dependência
- Dependência de saída

Anti-dependência e dependência de saída podem ser removidas renomeando variáveis.

Dependência verdadeira

- Dependência verdadeira ou dependência de fluxo ou *Read-After-Write* (RAW): quando uma instrução depende do resultado de outra.

Modelo: $A = \ldots$
$\ldots = A \ldots$

1: $A = A + 2$
2: $B = 2 \times A$
3: $C = B - A$

Instrução 2 depende verdadeiramente da instrução 1 (escrevemos $1 \rightarrow^{v} 2$).

Instrução 3 depende verdadeiramente da instrução 1 (escrevemos $1 \rightarrow^{v} 3$).

Instrução 3 depende verdadeiramente da instrução 2 (escrevemos $2 \rightarrow^{v} 3$).

Dependência verdadeira

Modelo: $A = \ldots$
$\ldots = A \ldots$

Colocar antes os presentes e depois pegar os presentes: OK.

Dependência verdadeira

Modelo: $A = \ldots$ $\Longrightarrow \Longrightarrow \Longrightarrow \Longrightarrow \Longrightarrow$ $\ldots = A \ldots$

$\ldots = A \ldots$ Mudar a ordem não dá $A = \ldots$

Pegar antes de colocar os presentes: Não OK.

Anti-dependência

- Anti-dependência ou *Write-After-Read* (WAR): quando uma instrução usa uma variável que depois vai ser alterada: a ordem de executar essas duas instruções não pode ser alterada, nem executadas em paralelo.

  Modelo: $\begin{array}{l} \ldots = A \ldots \\ A = \ldots \end{array}$

  1: $B = A + 5$
  2: $A = 7$

  A instrução 2 anti-depende da instrução 1 (escrevemos $1 \rightarrow^{anti} 2$):

- Suponha que gostaríamos muito de poder executar as instruções 1 e 2 ao mesmo tempo. Isso é possível se removermos a anti-dependência. Veremos isso agora.

Remoção de anti-dependência

- Anti-dependência pode ser removida ao renomear variáveis. Isso permite executar instruções que tinham anti-dependências em paralelo.

Modelo da anti-dependência	Remoção da anti-dependência
	$0 : A1 = A$
$1 : \ldots = \ldots A \ldots$	$1 : \ldots = \ldots A1 \ldots$
$2 : A = \ldots$	$2 : A = \ldots$

- A instrução $0 : A1 = A$ deve ser executada antes das outras duas instruções.
- Depois disso, as instruções 1 e 2 podem ser executadas em qualquer ordem. A anti-dependência foi removida.

Sejam as duas instruções com anti-dependência.

1: $B = A \times X$
2: $A = Y \times Z$

Renomeamos a variável A:

0: $A1 = A$
1: $B = A1 \times X$
2: $A = Y \times Z$

Após a renomeação da variável na instrução 0 e a execução da instrução 0, podemos executar instruções 1 e 2 em paralelo. Mas note que introduzimos uma dependência verdadeira entre as instruções 0 e 1.

Dependência de saída

- Dependência de saída ou *Write After Write* (WAW): quando a ordem das instruções afeta o valor final de saída de uma variável.

Modelo:
$$\begin{array}{l} A = \ldots \\ \ldots \\ A = \ldots \end{array}$$

1: $A = X * X$
2: $B = A + 5$
3: $A = Y * Y$

Instução 3 tem dependência de saída em relação à instrução 1 (escrevemos $1 \rightarrow^{saida} 3$).

- Suponha que gostaríamos muito de poder executar as instruções 1 e 3 ao mesmo tempo. Isso é possível se removermos a dependência de saída. Veremos isso agora.

Remoção de dependência de saída

- Dependência de saída também pode ser removida ao renomear variáveis.

Modelo da dependência de saída	Remoção da dependência de saída
$1 : A = \ldots$	$1 : A1 = \ldots$
$2 : \ldots = \ldots$ se aparecer $A \ldots$	$2 : \ldots = \ldots$trocar por $A1 \ldots$
$3 : A = \ldots$	$3 : A = \ldots$

- A instrução $1 : A1 = \ldots$ deve ser executada antes da instrução 2.
- As instruções 1 e 3 podem ser executadas em qualquer ordem. A dependência de saída foi removida.

Considerem instruções 1 e 3 com dependência de saída:

1: $A = X * X$
2: $B = A + 5$
3: $A = Y * Y$

Renomeanos a variável A:

1: $A1 = X * X$
2: $B = A1 + 5$
3: $A = Y * Y$

Como está o meu aprendizado?

Identifique todas as dependências nas segintes instruções:

1: $A = B$
2: $B = C + D$
3: $E = A + D$
4: $B = 0$
5: $F = B + 1$

Como está o meu aprendizado?

Identifique todas as dependências nas segintes instruções:

1: $A = B$
2: $B = C + D$
3: $E = A + D$
4: $B = 0$
5: $F = B + 1$

Resposta:

- $1 \rightarrow^{anti} 2$: anti-dependência
- $1 \rightarrow^{v} 3$: dependência verdadeira
- $2 \rightarrow^{saida} 4$: dependência de saída
- $1 \rightarrow^{anti} 4$: anti-dependência
- $4 \rightarrow^{v} 5$: dependência verdadeira

Como está o meu aprendizado?

As 3 instruções abaixo não podem ser executadas em paralelo, pois há anti-dependências. (1 $\rightarrow^{anti}$ 3 e 2 $\rightarrow^{anti}$ 3.)

1: $B = A \times X$
2: $C = A - Z$
3: $A = X \times X$

Elimine as anti-dependências por meio de renomeação de variável.

Como está o meu aprendizado?

As 3 instruções abaixo não podem ser executadas em paralelo, pois há anti-dependências. ($1 \rightarrow^{anti} 3$ e $2 \rightarrow^{anti} 3$.)

1: $B = A \times X$
2: $C = A - Z$
3: $A = X \times X$

Elimine as anti-dependências por meio de renomeação de variável.

Resposta:

0: $A1 = A$
1: $B = A1 \times X$
2: $C = A1 - Z$
3: $A = X \times X$

- Podemos executar todas as 4 instruções acima em paralelo?
- Quais novas dependências foram introduzidas?
- Qual instrução deve ser executada primeiro?

Como está o meu aprendizado?

Identifique todas as dependências nas segintes instruções:

1: $B = Y * Y$
2: $A = B/C$
3: $C = A + X$
4: $A = X/Y$
5: $D = D + 1$

Agora, reescreva as instruções, removendo as anti-dependências e dependências de saída. Soluções a seguir.

Como está o meu aprendizado?

1: $B = Y * Y$
2: $A = B/C$
3: $C = A + X$
4: $A = X/Y$
5: $D = D + 1$

- $1 \rightarrow^{v} 2$ (dependência verdadeira)
- $2 \rightarrow^{v} 3$ (dependência verdadeira)
- $2 \rightarrow^{anti} 3$ (anti-dependência)
- $3 \rightarrow^{anti} 4$ (anti-dependência)
- $2 \rightarrow^{saida} 4$ (dependência de saída)

Como está o meu aprendizado?

1: $B = Y * Y$
2: $A = B/C$
3: $C = A + X$
4: $A = X/Y$
5: $D = D + 1$

Removendo 2 $\rightarrow^{anti}$ 3 e depois 2 $\rightarrow^{saida}$ 4 já resolve o problema.

1: $B = Y * Y$
2: $C1 = C$
3: $A1 = B/C1$
4: $C = A1 + X$
5: $A = X/Y$
6: $D = D + 1$

Lista de Exercícios 3

- Fazer e entregar por email a Lista de Exercícios 3.
- Há prazo para entrega. Recomendo não demorar muito. Bom fazer logo com a matéria fresquinha na cabeça.

Processador superescalar e Algoritmo de Tomasulo

Robert Tomasulo

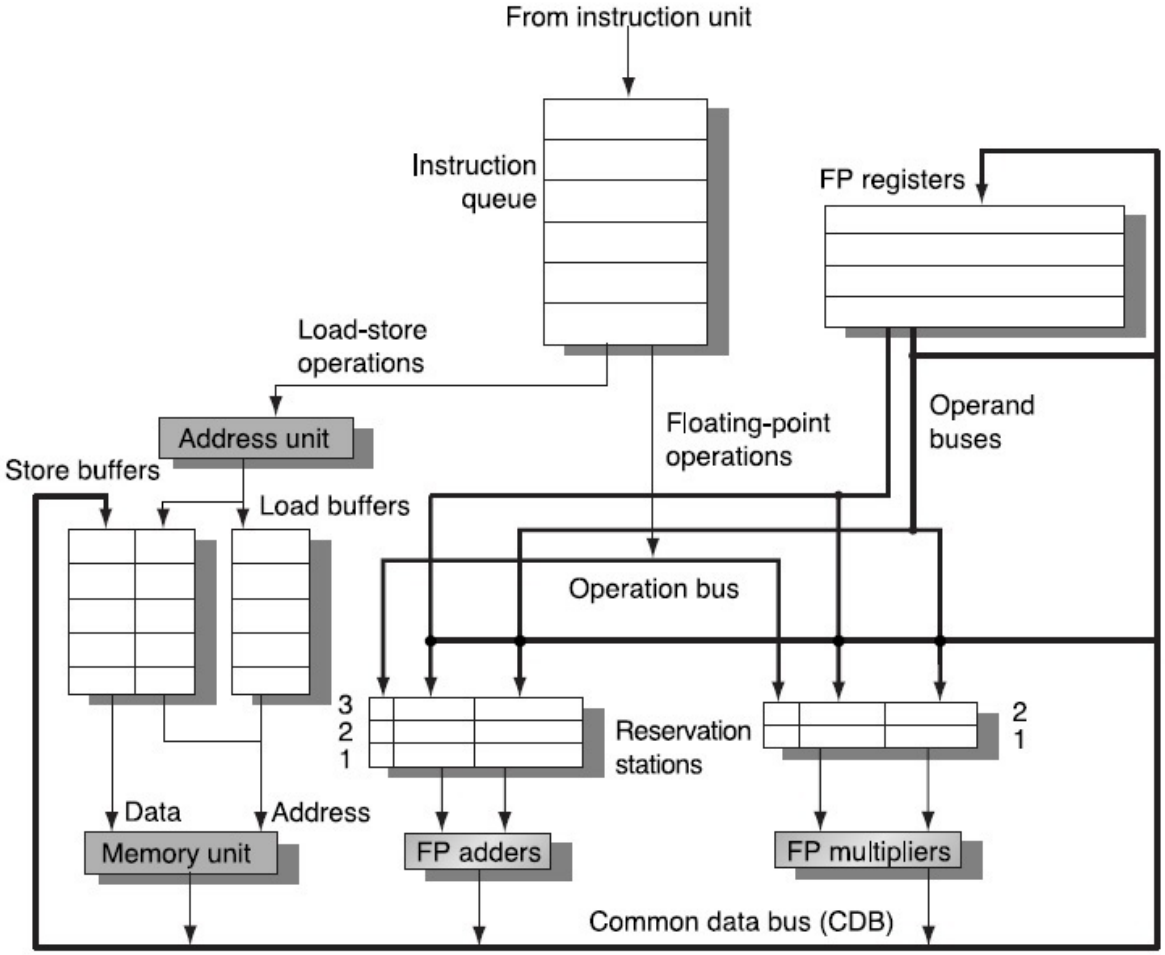
Foi recipiente do 1997 Eckert-Mauchly Award. Fez carreira na IBM onde desenvolveu o System/360 e sucessores. Trabalhou também num projeto que desenvolveu o primeiro computador de grande porte baseado em CMOS.

- O Algoritmo de Tomasulo permite a execução de instruções fora de ordem, para aproveita a capacidade de processadores com múltiplas unidades funcionais.
- Esse algoritmo é implementado em hardware, remove anti-dependências e dependências de saída ao renomear registradores.
- O conceito de processador superescalar em geral é associado a arquiteturas RISC. (Veremos arquiteturas RISC e CISC mais tarde.)
- Mas esse conceito também se aplica a CISC, como o processador Pentium 4, que possui três unidades para execução de instruções de inteiros e duas de ponto flutuante.

Algoritmo de Tomasulo - Não cai em prova

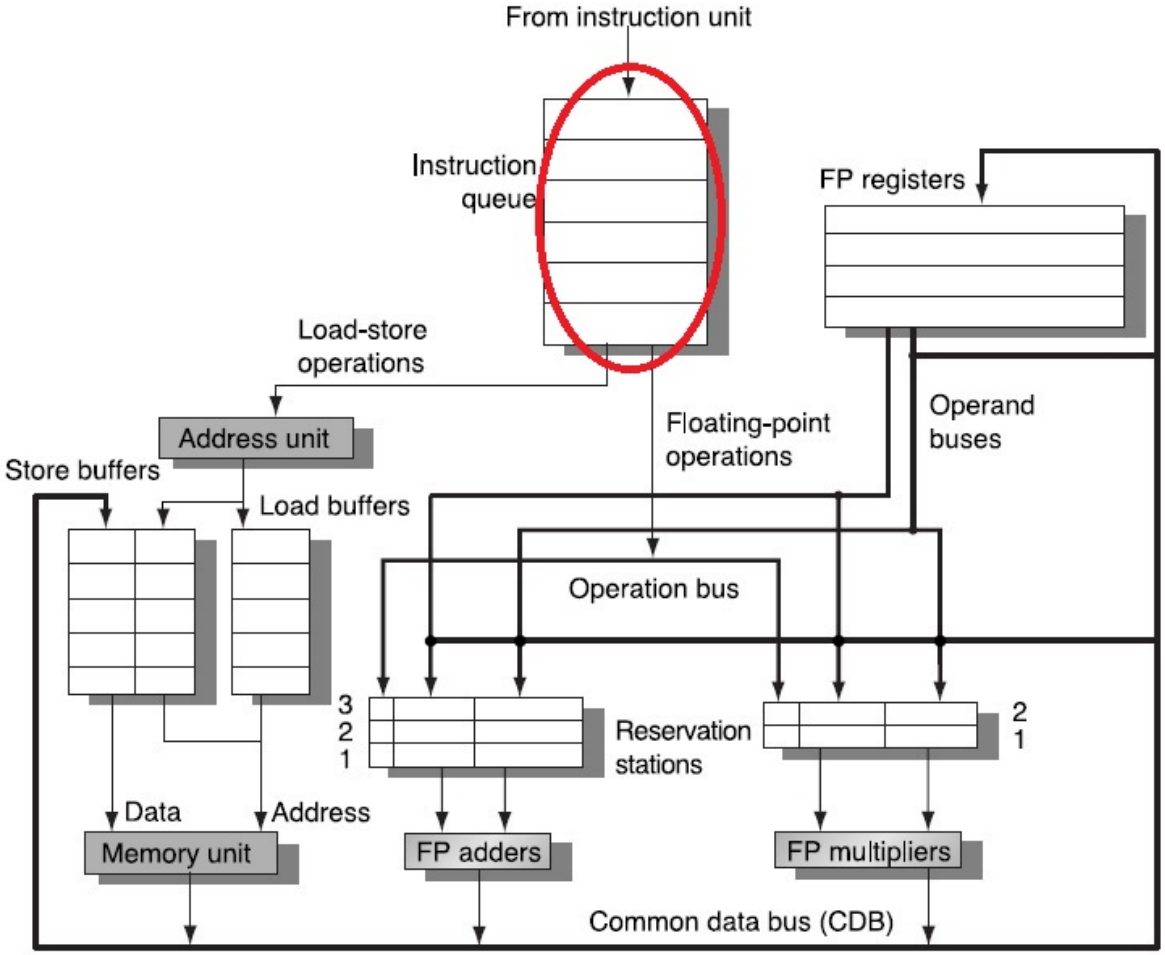
- O algoritmo de Tomasulo foi implementado em hardware no IBM 360/91. A mesma ideia é usada depois em outros processadores, como MIPS, Pentium Pro, DEC Alpha, PowerPC, etc.
- Rastreia quando operandos estão disponíveis a fim de satisfazer dependências.
- Remove anti-dependências e dependências de saída por renomeamento de registradores.

Algoritmo de Tomasulo - Não cai em prova



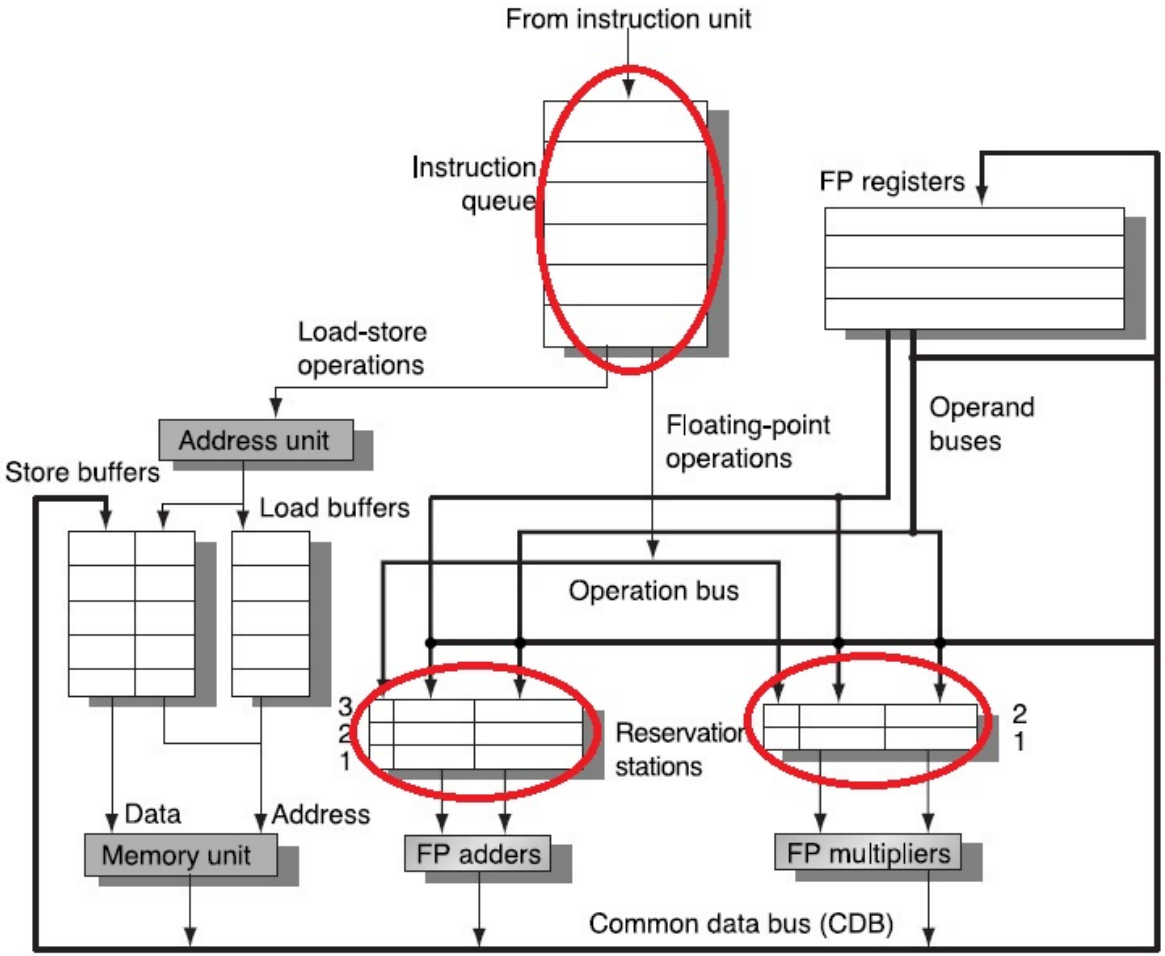
Vamos explicar de forma superficial o algoritmo de Tomasulo.

Algoritmo de Tomasulo - Não cai em prova



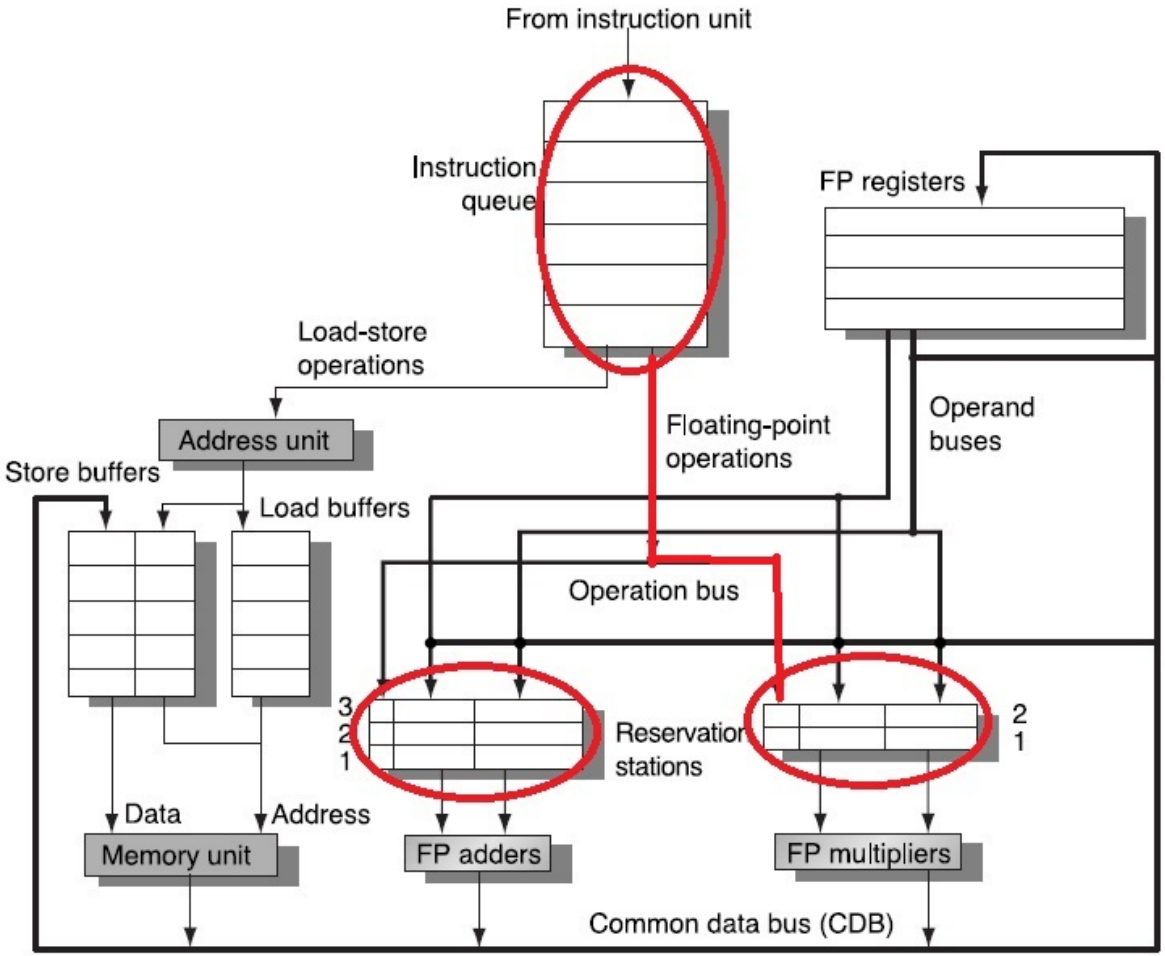
Instruções são carregadas em *instruction queue*.

Algoritmo de Tomasulo - Não cai em prova



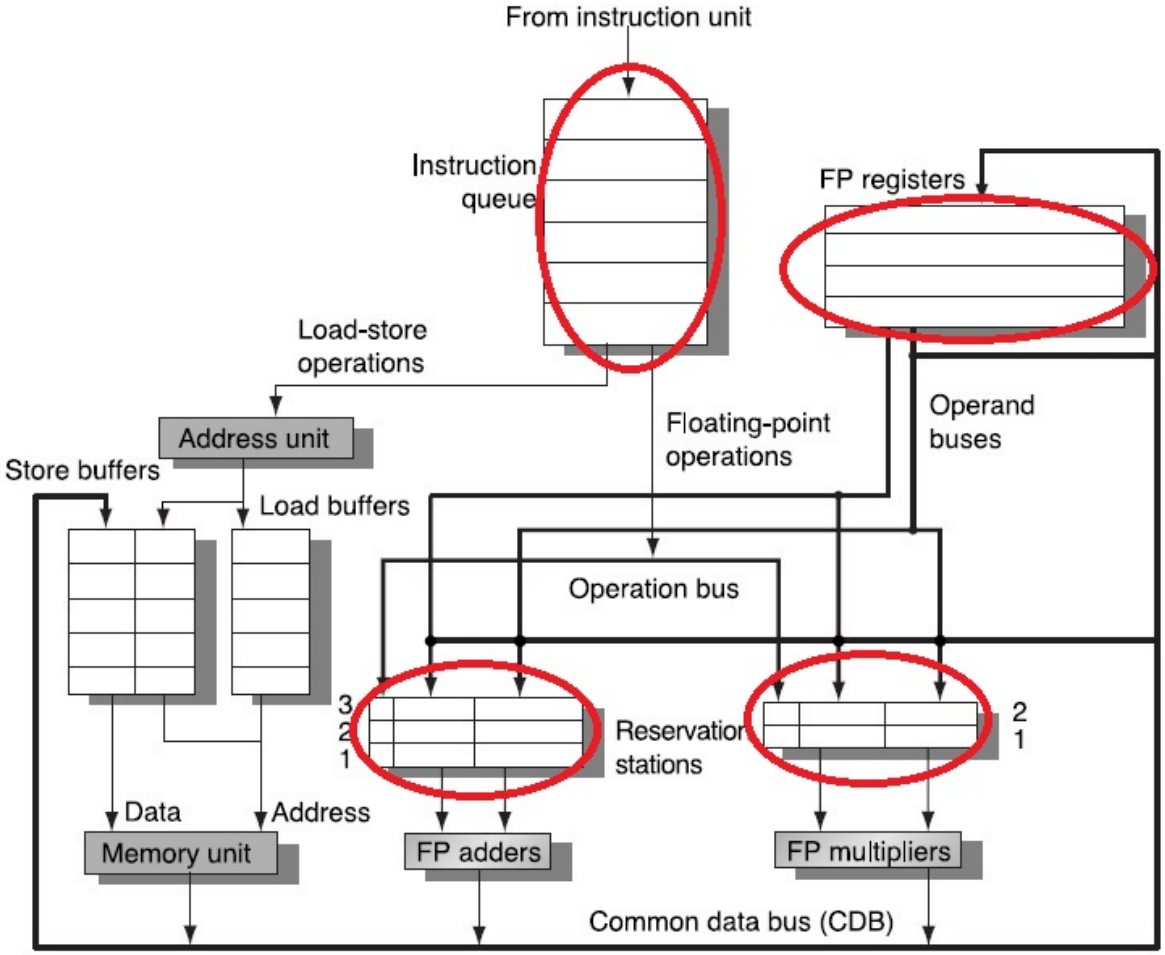
Usa um número de *reservation stations* para a execução de instruções.

Algoritmo de Tomasulo - Não cai em prova



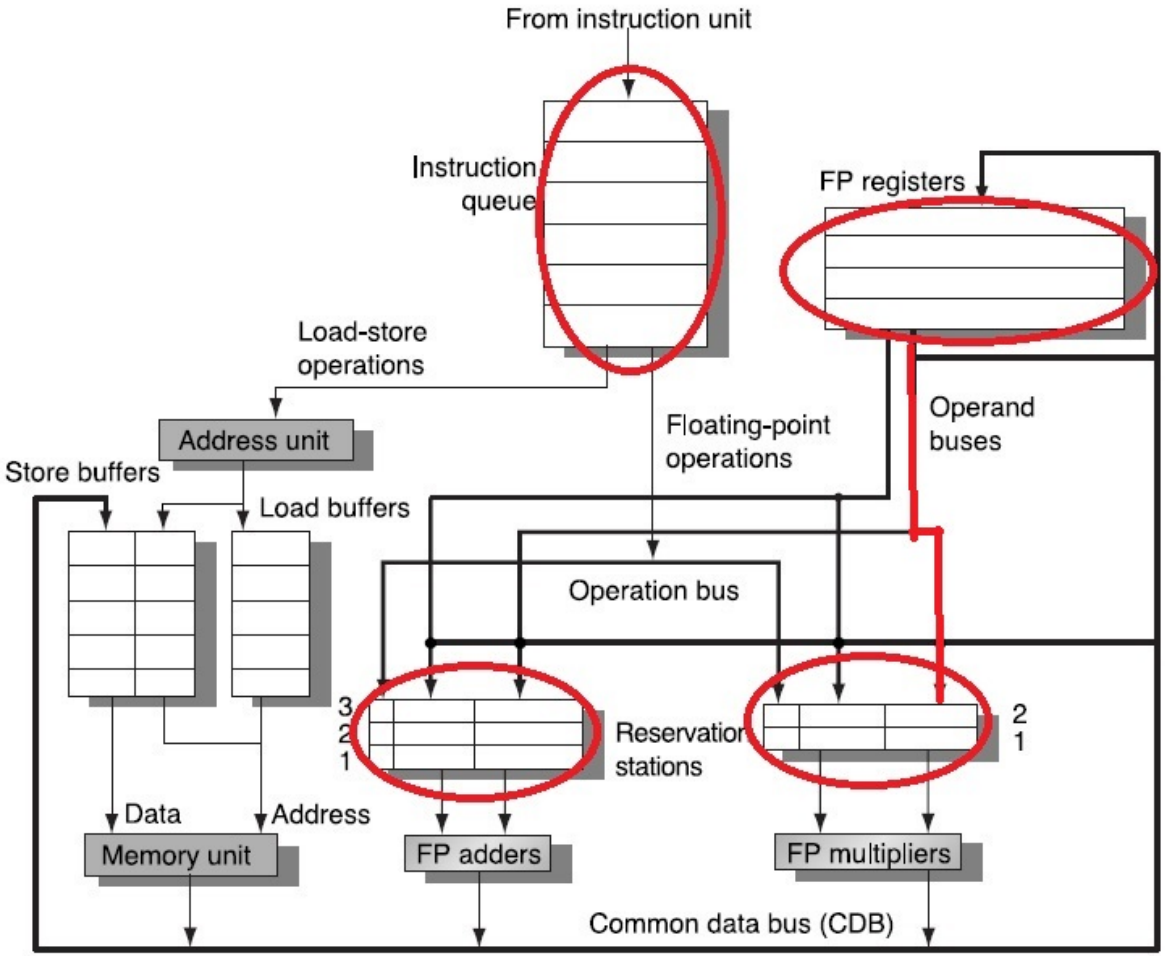
Pega uma instrução da *instruction queue* e ache uma *reservation station* livre para ela.

Algoritmo de Tomasulo - Não cai em prova



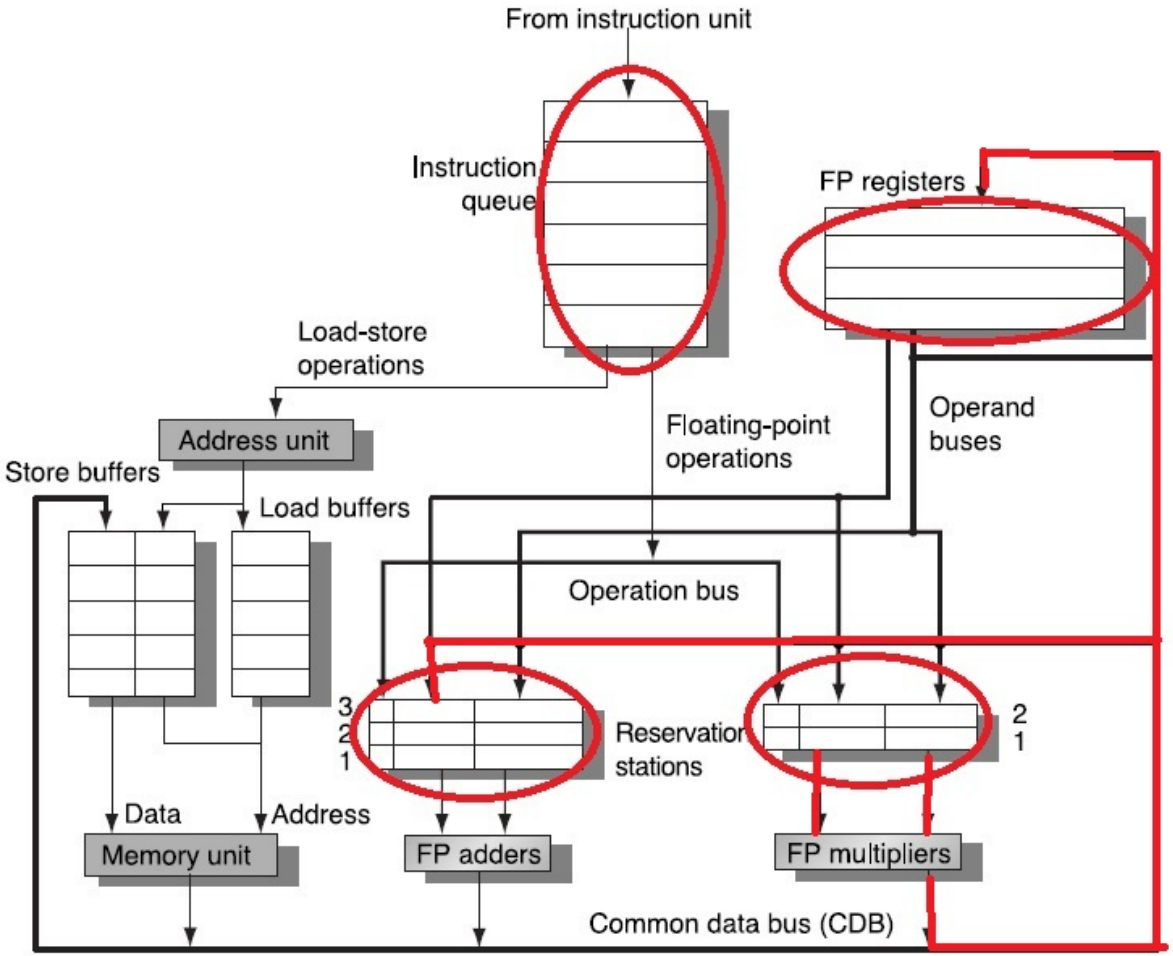
Registradores armazenam dodos lidos ou resultados produzidos.

Algoritmo de Tomasulo - Não cai em prova



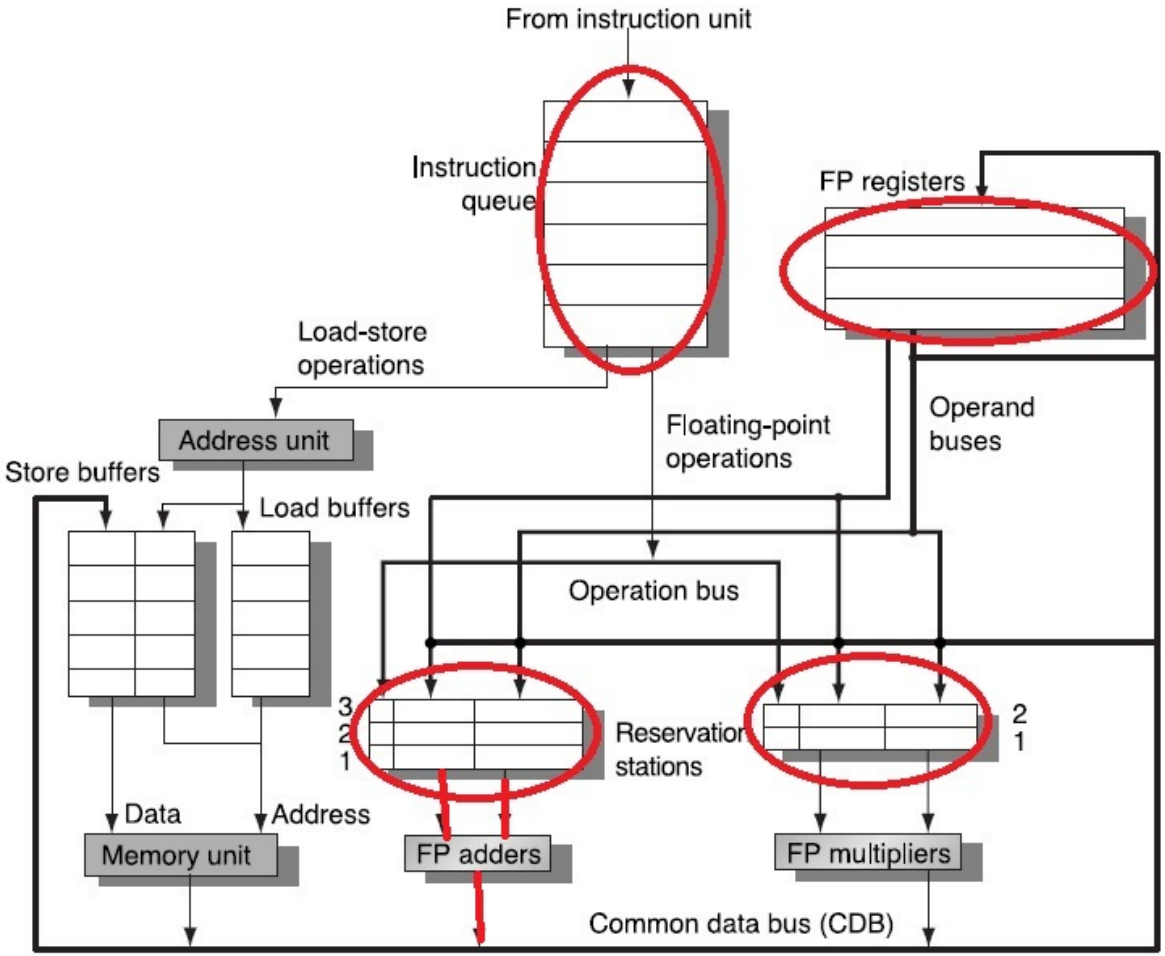
Lê operandos que estão em registradores. Se o operando não está lá localiza qual *reservation station* vai produzi-lo.

Algoritmo de Tomasulo - Não cai em prova



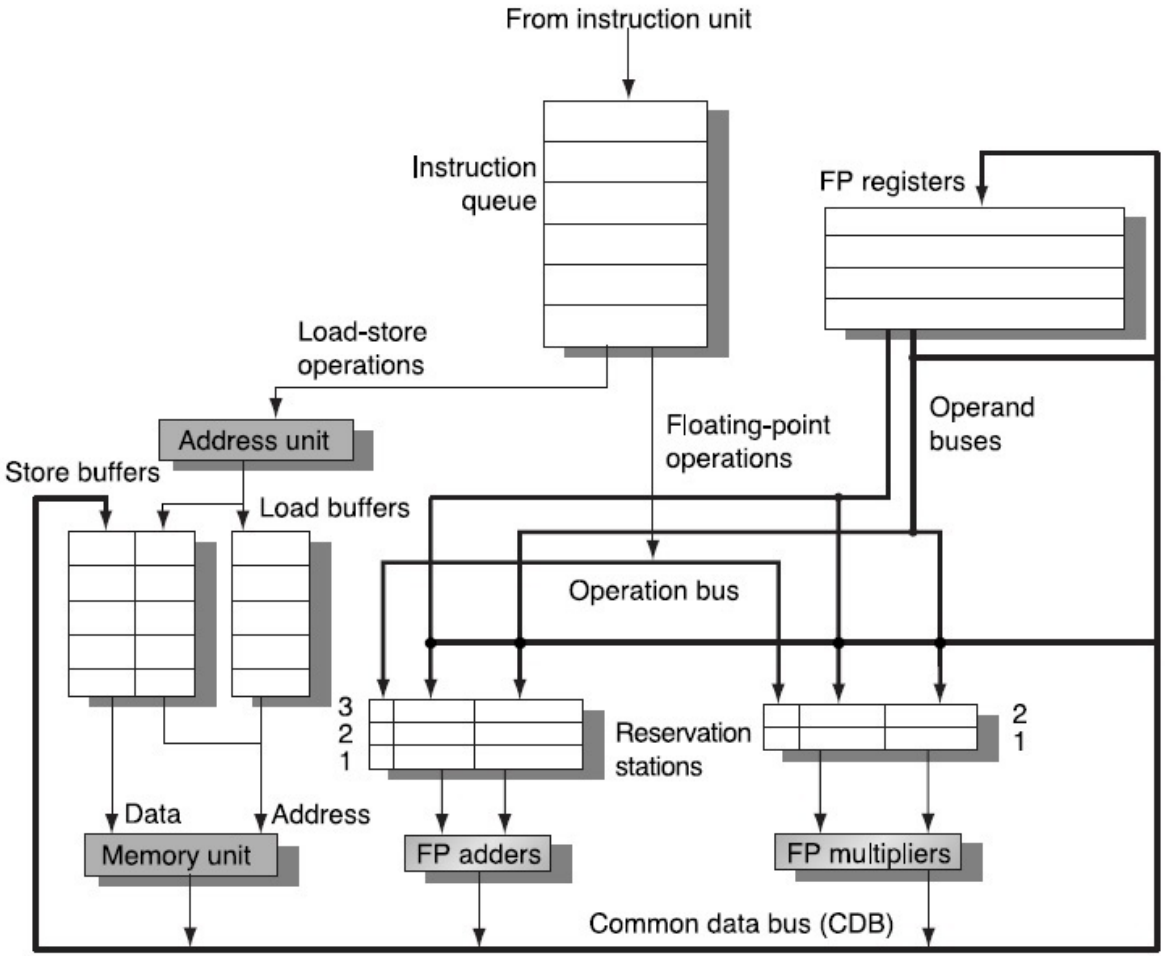
Monitora resultados quando produzidos. Coloca o resultado em todas as *reservation stations* que estão esperando por ele.

Algoritmo de Tomasulo - Não cai em prova



Quando todos os operandos de uma instrução estão disponíveis, ela é enviada para execução.

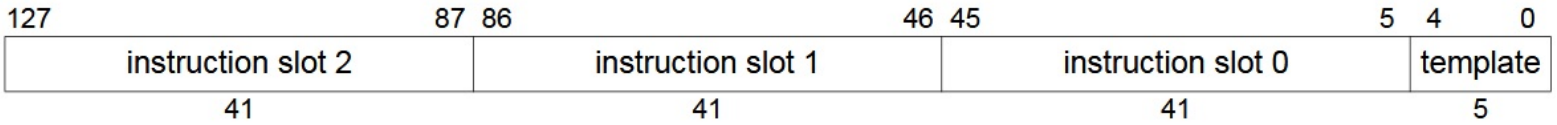
Algoritmo de Tomasulo - Não cai em prova



O funcionamente detalhado foge do escopo do nosso curso. (Há vasta literatura sobre o assunto para os curiosos.)

VLIW - Very Large Instruction Word

- Processador superescalar possui várias unidades funcionais.
- Para melhorar explorar essas múltiplas unidades, alguns processadores podem adotar um formato de instruções longas chamadas *Very Large Instruction Words - VLIW*.
- Uma VLIW contém mais que uma instrução.
- Exemplo: Itanium.



Intel Itanium Architecture - Instruction Set

Execução especulativa

- Usando predicção de desvio e análise de fluxo de dados, alguns processadores especulativamente executa instruções antes que elas aparecem realmente na sequência de instruções.
- Num desvio condicional, o processador pode apostar num dos 2 trechos e já sai executando as instruções desse trecho.

if condiçaõ **then** trecho 1
else trecho 2

- Os resultados da execução especulativa são armazenados em locais temporários e somente validados depois.
- Essa técnica faz com que o processador fique sempre ocupado ao executar instruções que provavelmente seriam necessárias.
- Pentium Pro implementa as técnicas superescalar, predicção de desvios, análise de fluxo de dados e execução especulativa.

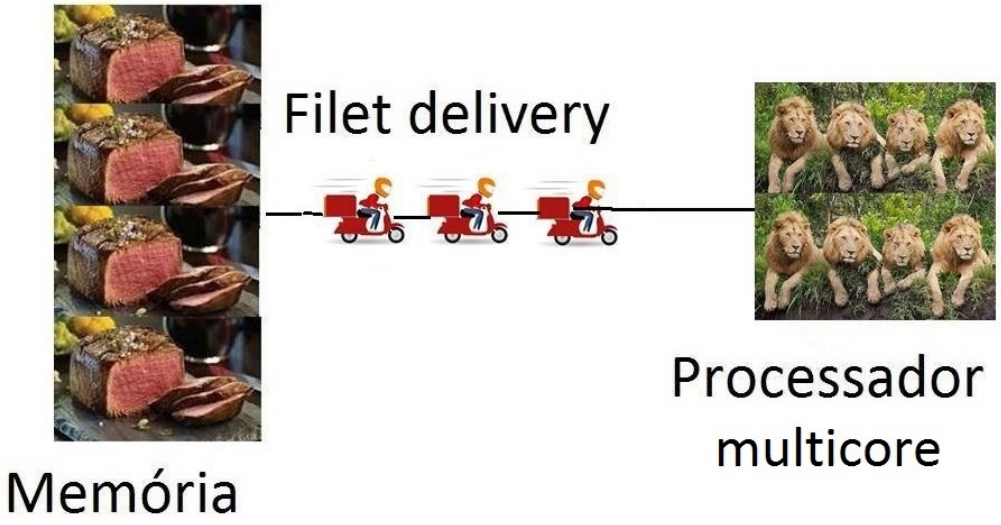
Predictor de desvio

if condiçaõ **then** trecho 1
else trecho 2

- Predictor de desvio estático: decidido em tempo de compilação. Por exemplo, desvio para trás é sempre preferido (em laços, na maioria das vezes, o desvio executado é para trás). Exemplo: Intel Pentium 4.
- Predictor de desvio dinâmico: Usa informações obtidas na execução para prever qual desvio a tomar. Basicamente os desvios realizados são registrados de algumas forma. Exemplo: Intel Core i7.

Processador multicore

- A busca por desempenho no processador concentrava em *pipelining* de instruções, *superescalar* com múltiplas unidades de execução de instruções explorando o paralelismo no nível de instruções, etc. Tais técnicas atingiram seu limite.
- Surge então o conceito de *multicore*.
- O uso de múltiplos processadores numa mesma pastilha, conhecido como multicore (ou múltiplo núcleos), é uma forma de aumentar o desempenho sem aumentar a frequência do relógio.



Processador multicore

- Tipicamente cada *core* ou núcleo contém todos os componentes de um processador independente, como registradores, ALU, unidade de controle, L1 cache, etc.
- Isso é viável com o avanço da tecnologia VLSI (Lei de Moore).

Processador	Número de *cores*
Intel Core i7	4
Intel Core i9	18
AMD Epyc	32
Intel Xeon Phi	60

Processador multicore

- Pipelining explora a possibilidade executar as etapas de cada instrução em paralelo, em forma de uma linha de montagem.
- Processador superescalar viabiliza o paralelismo no nível de instrução: executar várias instruções de um programa em paralelo.
- Multicore apresenta vários *cores* (núcleos) cada um independente.

Processador multicore

- Quantos *cores* é o ideal?
- Depende da tarefa e do software para paralelizar a tarefa. Algumas só se beneficiam com poucos *cores*.
- Ver CPU cores: how many do I need?

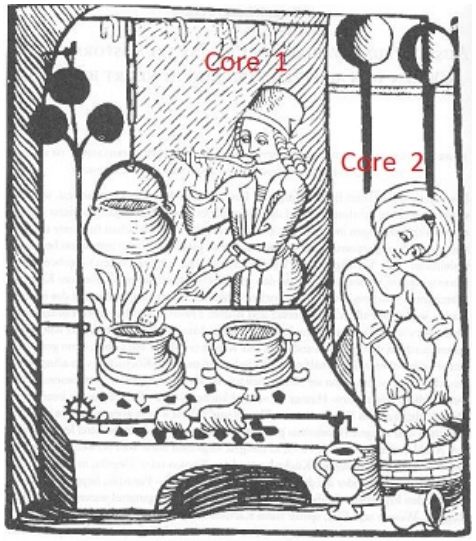


Image source: Wikimedia Commons

Há um provérbio inglês: “Too many cooks spoil the broth.”

Processador multicore

CPU cores: how many do I need?

- 2 *cores*: navegar na internet, processamento de texto, email, *streaming* de vídeo
- 4 *cores*: vídeo game leve, edição de foto e vídeo, multi-tarefa
- 6 a 8 *cores*: vídeo game mais pesado, edição de vídeo profissional, renderização ou visualização 3d
- 10 *cores* ou mais: vídeo game de alta qualidade, animação 3d, simulações estatísticas

O artigo acima não considera o uso de múltiplos *cores* para agilizar a execução de uma tarefa, usando um algoritmo paralelo. Nesse caso, por exemlo, o LINPACK pode se beneficiar com a execução paralela de milhões de *cores*.

Vulnerabilidades Meltdown e Spectre

- Processadores modernos usam diversas técnicas para obter maior desempenho. Algumas dessas técnicas, quando combinadas, podem ser exploradas afetando a segurança do sistema.
- Veremos mais tarde as vulnerabilidades Meltdown e Spectre que exploram:
  - Predicção de desvio
  - Execução fora de ordem
  - Execução especulativa
  - Uso de memória cache

Lei de Amdahl

- Para medir o ganho obtido com o uso de múltiplos processadores para agilizar a execução de uma tarefa, define-se o chamado ganho ou *speedup*:

$$\text{speedup} = \frac{\text{tempo de execução sequencial com um processador}}{\text{tempo de execução paralelo com } n \text{ processadores}}$$

- Teoricamente, com n processadores, o *speedup* pode chegar ao valor ideal n.
- A Lei de Amdahl mostrará o que se espera na prática.

Lei de Amdahl

- Quarenta anos atrás (anos 70), não havia mais que cinco computadores paralelos.
- Hoje até um *smartphone* pode possuir múltiplos núcleos ou *cores*.
- A computação paralela se tornou regra e não mais exceção.
- É importante entender a Lei de Amdahl sobre o poder e a limitação da computação paralela.
- O ganho ou *speedup* mede o potencial de um programa usando *n* processadores em comparação com um único processador:

$$\text{speedup} = \frac{\text{tempo de execução sequencial com um processador}}{\text{tempo de execução paralelo com } n \text{ processadores}}$$

Lei de Amdahl

- Considere

$$\text{speedup} = \frac{T_1}{T_n} = \frac{\text{tempo de execução sequencial com um processador}}{\text{tempo de execução paralelo com } n \text{ processadores}}$$

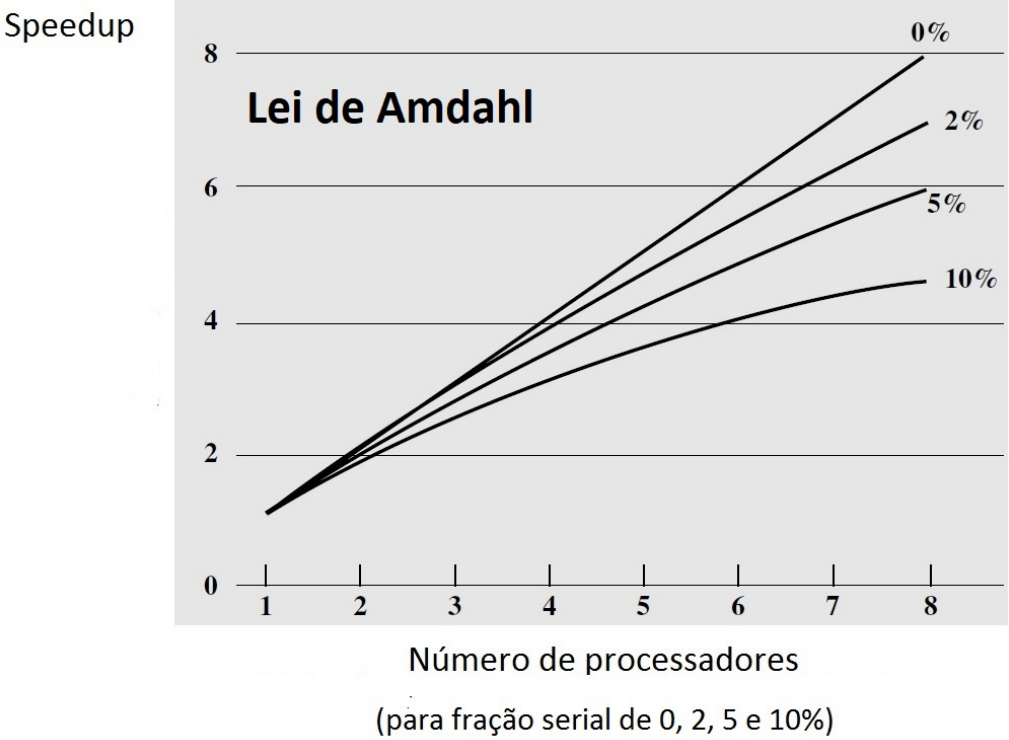
- Um programa executando com um único processador onde
  - uma fração f do código pode ser paralelizado perfeitamente
  - uma fração $(1 - f)$ do código é inerentemente sequencial

$$\text{speedup} = \frac{T_1}{T_n} = \frac{T_1}{T_1(1-f) + \frac{T_1 f}{n}} = \frac{1}{(1-f) + \frac{f}{n}}$$

- Para um grau de paralelismo muito grande ($n \to \infty$), o *speedup* é limitado por $\frac{1}{(1-f)}$.
- Se f é pequeno, então mesmo o paralelismo maciço não ajuda.
- Suponha f é razoalvente grande, digamos $f = 0,9$, ou seja, 90% do código é perfeitamente paralelizável. Você acha que ia dar um bom *speedup*? Vamos ver no próximo slide.

Lei de Amdahl

- A Lei de Amdahl ilustra o problema que a indústria enfrenta ao projetar um número cada vez maior de *cores*.
- Considere que apenas 10% do código é inerentemente serial, i.e. fração paralela é $f = 0.9$ (90% paralelizável).
- Então executar esse código com 8 *cores* produz um *speedup* $= \frac{1}{(1-f)+\frac{f}{n}} = 4,7$.
- Agora considere $f = 0,7$, i.e., 70% paralelizável. *speedup* $= 2,6$.



Source: W. Stallings

Lei de Amdahl

- Além disso, precisamos levar em conta outras sobrecargas da computação paralela como comunicação, distribuição de cargas, etc.
- Essa conclusão pessimista vale quando os n processadores são usados para acelerar a execução de um mesmo programa.
- Frequentemente os múltiplos processadores executam diversos programas independentes.

Paralelização de laços (*loops*) - Não cai em prova

- Comandos em laços se constituem em excelentes oportunidades para execução em paralelo.
- Há estudos e resultados muito interessantes na literatura. Um estudo mais detalhado foge do escopo deste curso. Nos próximos slides, vamos dar apenas um sabor desse tópico (para satisfazer os curiosos).
- Caso de dependência dentro da própria iteração do laço:

1: **for** i from 1 to 1000 **do**
2: $\quad A[i] = B[i] + 10$
3: $\quad C[i] = A[i] * 2$
4: **end for**

- Podemos executar em paralelo todos os comandos da linha 2 para $i = 1, 1000$.
- Depois fazer o mesmo para os comandos da linha 3.

Paralelização de laços (*loops*) - Não cai em prova

- Caso de dependências que cruzam iterações do laço.

```
for i from 1 to 100 do
    A[i, 0] = 0
end for
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = A[i, j − 1] × 2
    end for
end for
```

- Cada $A[i, j]$ depende do resultado de $A[i, j-1]$ calculado em iterações anteriores.
- Em outras palavras, a iteração $[i, j]$ depende da iteração $[i, j-1]$ e podemos definir um vetor de dependências (também conhecido com vetor de distâncias) assim:

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

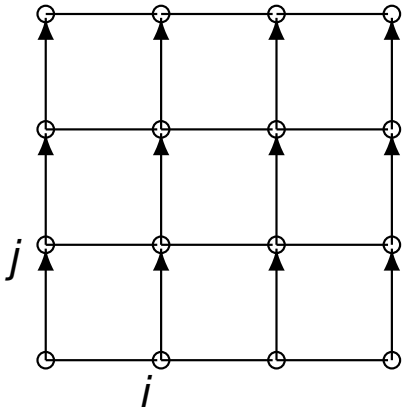
Paralelização de laços (*loops*) - Não cai em prova

```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = A[i, j − 1] × 2
    end for
end for
```

- A iteração $[i, j]$ depende da iteração $[i, j-1]$.
- Vetor de dependências

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- representado graficamente:



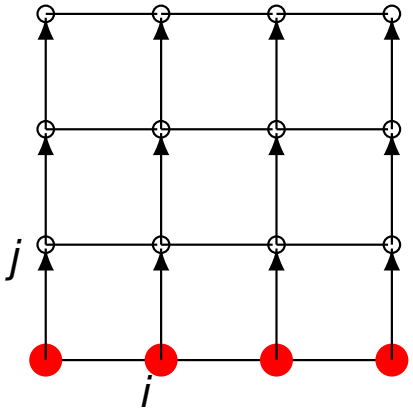
Paralelização de laços (*loops*) - Não cai em prova

```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = A[i, j − 1] × 2
    end for
end for
```

- A iteração $[i, j]$ depende da iteração $[i, j-1]$.
- Vetor de dependências

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



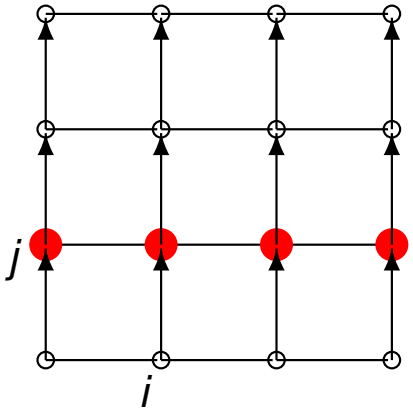
Paralelização de laços (*loops*) - Não cai em prova

```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = A[i, j − 1] × 2
    end for
end for
```

- A iteração $[i, j]$ depende da iteração $[i, j-1]$.
- Vetor de dependências

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



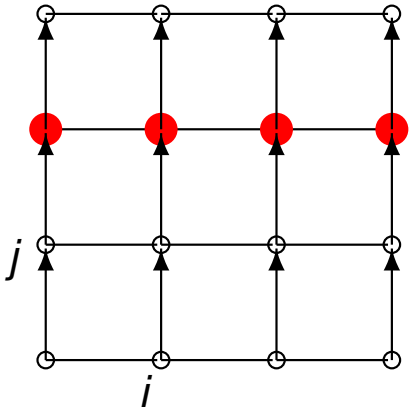
Paralelização de laços (*loops*) - Não cai em prova

```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = A[i, j − 1] × 2
    end for
end for
```

- A iteração $[i, j]$ depende da iteração $[i, j-1]$.
- Vetor de dependências

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



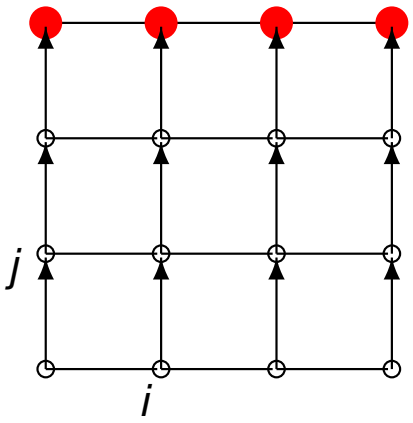
Paralelização de laços (*loops*) - Não cai em prova

```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = A[i, j − 1] × 2
    end for
end for
```

- A iteração $[i, j]$ depende da iteração $[i, j-1]$.
- Vetor de dependências

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*)

- Caso de dependências que cruzam iterações do laço.

```
for k from 1 to 100 do
    A[k, 0] = 0
    A[0, k] = 0
end for
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- Cada $A[i, j]$ depende do resultado de $A[i-1, j]$ e de $A[i, j-1]$ calculados em iterações anteriores.
- Em outras palavras, a iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$ e podemos definir vetores de dependências:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

Paralelização de laços (*loops*) - Não cai em prova

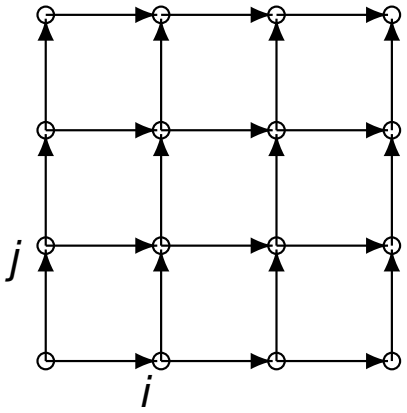
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- representados graficamente:



Paralelização de laços (*loops*) - Não cai em prova

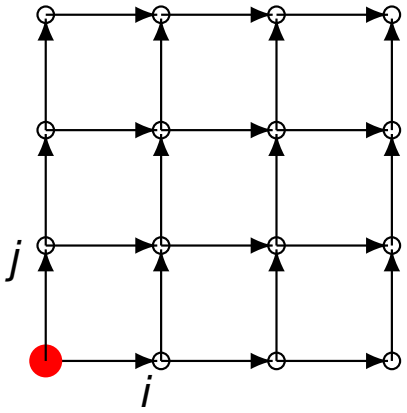
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

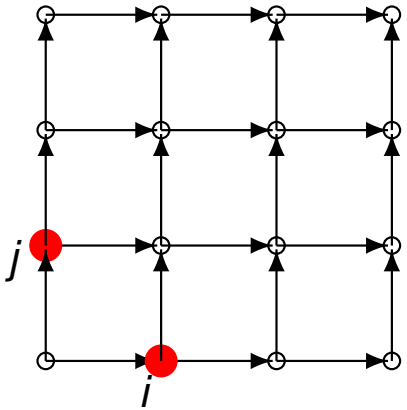
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

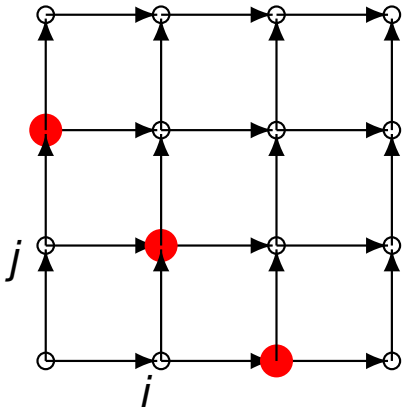
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

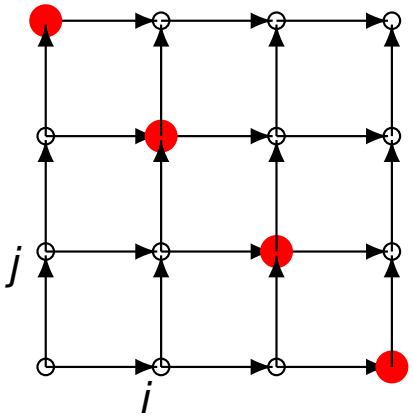
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

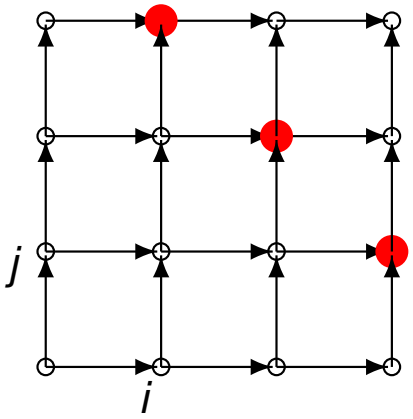
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

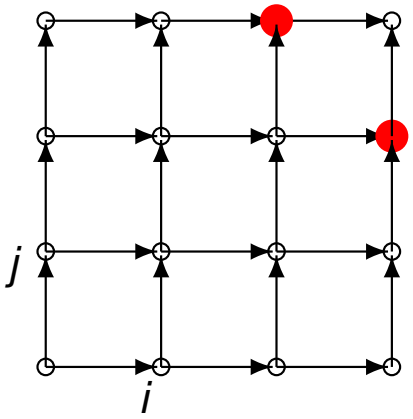
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

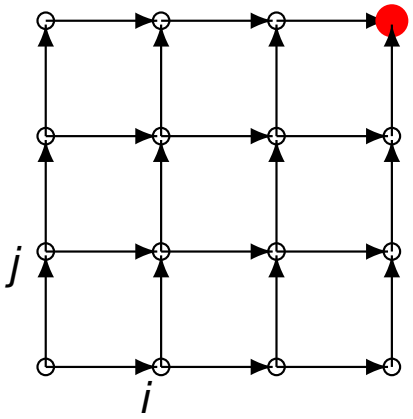
```
for i from 1 to 100 do
    for j from 1 to 100 do
        A[i, j] = max{A[i − 1, j], A[i, j − 1]}
    end for
end for
```

- A iteração $[i, j]$ depende das iterações $[i-1, j]$ e $[i, j-1]$.
- Vetores de dependências

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j-1 \end{pmatrix}$$

- executando em paralelo:



Paralelização de laços (*loops*) - Não cai em prova

- Vimos alguns exemplos simples para ilustrar a paralelização de laços.
- Existem métodos que analisam os vetores de dependências e buscam paralelismo em laços.
- Há muitos trabalhos na literatura sobre este assunto.

C. D. Polychronopoulos. Parallel Programming and Compilers. Kluwer Academic Publishers. Boston. 1988.

P. Quinton, Y. Robert. Systolic Algorithms and Architectures. Prentice-Hall Masson. 1991.

Como foi o meu aprendizado?

- Com quais afirmações abaixo você concorda?
  1. É mais fácil programar um computador com um processador do que um com milhares ou milhões de processadores.
  2. A técnica de pipelining de instruções funciona melhor quando há poucos desvios na sequência de instruções.
  3. O processador superescalar explora o paralelismo no nível de instruções.
  4. Terei um altíssimo desempenho se fabrico um processador superescalar com centenas ou milhares de unidades de execução.
  5. Implementar vários processadores (ou *cores*) numa única pastilha é uma forma de aumentar o desempenho sem aumentar o *clock*.
  6. A Lei de Amdahl mostra que nem todo problema pode ser resolvido de forma satisfatória usando um computador paralelo.

– Continua na próxima página.

Como foi o meu aprendizado?

Essas duas questões a seguir fogem do escopo do nosso curso. Mas são interessantes. Os curiosos podem tentar responder.

- Certos problemas são facilmente paralelizáveis, dando excelentes *speedups*. Você pode pensar em um desses problemas?
- Certos problemas são essencialmente sequencias e difíceis de obter bom ganho com computação paralela. Você pode tentar sugerir um desses problemas?

  Obs: É uma questão em aberto na teoria da complexidade de computação paralela a existência de problemas inerentemente sequenciais. Essa questão envolve os conceitos de classes de complexidade NC e P e P-completo, e se NC = P? (Esse tópico (interessante) foge do escopo da nossa disciplina.)

Próximo assunto: Hierarquia de memória e memória cache

- Próximo assunto: Hierarquia de memória e memória cache
- Para equilibrar as velocidades do processador e a memória, uma hierarquia de memória é usada: desde memórias rápidas com pequena capacidade próximas ao processador a memórias mais lentas e de grande capacidade distantes do processador.
- Veremos a memória cache e e como mapear a memória principal à memória cache.
- Não percam!