

# MAC0344 Arquitetura de Computadores

## Quem é o professor?

## Como será essa disciplina?

Prof. Siang Wun Song

2022

# Quem é o professor?



- Meu nome é Siang. Comecei a dar aula em **1971** (+50 anos atrás!) para a primeira turma do BCC do IME-USP.
- Depois de terminar o doutorado em Computação na **Universidade Carnegie Mellon**, na área de Computação Paralela, comecei a dar a disciplina Organização de Computadores que depois mudou para Arquitetura de Computadores para o BCC.
- Já dei essa disciplina **35 vezes**. Obviamente o conteúdo mudou bastante.
- Estou aposentado. Sou Professor Sênior (voluntário).

# Quem é o professor?



- Meu nome é Siang. Comecei a dar aula em **1971** (+50 anos atrás!) para a primeira turma do BCC do IME-USP.
- Depois de terminar o doutorado em Computação na **Universidade Carnegie Mellon**, na área de Computação Paralela, comecei a dar a disciplina Organização de Computadores que depois mudou para Arquitetura de Computadores para o BCC.
- Já dei essa disciplina **35 vezes**. Obviamente o conteúdo mudou bastante.
- Estou aposentado. Sou Professor Sênior (voluntário).

# Sala e email do Professor



- Sala 10 - Térreo - Bloco C
- Email - **Anotem por favor:**  
`song@ime.usp.br`  
Fiquem à vontade de me mandar email para dúvidas, problemas, etc. e também para **entregar exercícios**
- Homepage da disciplina - **Anotem por favor:**  
<http://www.ime.usp.br/~song/mac0344-2022.html>  
Contém avisos e informações sobre a disciplina e **slides usados**.
- Essas informações estão também na E-disciplinas MAC0344.

# Como serão as aulas?



- As aulas serão presenciais.
  - Terça feira: 8:00-9:40
  - Quinta feira: 10:00-11:40
- Estou vendo a possibilidade de gravar as aulas presenciais. As aulas de 2021 foram gravadas e acessíveis no link:  
<http://www.ime.usp.br/~song/mac0344-2021.html>

# Avaliação do aproveitamento



Prova com consulta



Exercício para casa



INSTITUTO DE COMPUTAÇÃO  
Mecanização e Arquitetura dos Processadores

Universidade de São Paulo  
Instituto CPqD-USP

Site Oficial  
10 de novembro de 2020

Monografia

- **Uma prova** com substitutiva opcional: nota P (melhor nota das duas)
- **Listas de exercícios**: média E.  
Haverá 5 listas de exercícios.
- **Uma monografia** (individual ou em grupo de até 3): nota M
- Nota de aproveitamento final:  $A = 0.2E + 0.4P + 0.4M$

# Bibliografia



**Todos os slides  
já estão na  
página da  
disciplina e no  
E-disciplinas  
USP.**

- 1 William Stallings. Computer Organization and Architecture. Pearson.
- 2 Andrew S. Tanenbaum. *Structured Computer Organization*, Prentice Hall - 5th edition, 2006.
- 3 Artigos diversos.
- 4 **Não precisam comprar nenhum livro.**

Slides (pdf) a serem usados em cada aula já estão disponíveis na página da disciplina

<http://www.ime.usp.br/~song/mac0344-2022.html> e em E-disciplinas (procure MAC0344.)

# Ementa

*Conceitos básicos de arquitetura.*

*Histórico dos computadores e gerações.*

*Desempenho: pipeline, RISC, instruções superescalares, multicore.*

*Memória cache, tipos e implementações.*

*Estrutura interna da memória: DRAM, SDRAM, Flash, correções de erros.*

*Memória externa: disco magnéticos, estado sólido, discos óicos, RAID.*

*Paginação e segmentação, TLBs, memória virtual.*

*Instruções de máquina, RISC e CISC, execução fora de ordem, modos de endereçamento, interrupções e proteção.*

Para um vídeo com uma apresentação mais completa da ementa, ver [Apresentação da disciplina MAC0344 e informações](#).

Para os slides usados neste vídeo, ver [Apresentação da disciplina MAC0344](#)

- Uma breve introdução sobre a diferença entre Arquitetura de Computador e Organização de Computador
- **História de Computação:** as várias gerações - do ábaco ao Supercomputador Frontier
- Estado de arte: Os supercomputadores da lista TOP500
- **Transistor MOS e Tecnologia VLSI** (circuitos integrados em Silício )

*A evolução da computação depende fortemente da tecnologia disponível. A atual é a do Silício, responsável pelo fantástico avanço da área. Estamos no limiar de uma nova tecnologia.*

- Requisitos formais:  
MAC0329 Álgebra booleana e aplicações no projeto de arquiteturas de computadores  
MAC0121 Algoritmos e Estruturas de Dados I
- Ajuda para melhor apreciar a importância da tecnologia VLSI no avanço da área, quando introduzo essa tecnologia em algumas aulas.
- A maior parte do curso trata de temas de nível mais elevado e não requer conhecimento prévio.

# Fim da apresentação

- Desejo bom proveito nessa disciplina!

# MAC0344 Arquitetura de computadores - Introdução

Prof. Siang Wun Song

<http://www.ime.usp.br/~song/>

Slides usados: <https://www.ime.usp.br/~song/mac344/slides01-introduction.pdf>

Baseado em W. Stallings -

Computer Organization and Architecture

- Estudo de um sistema de computação sob dois pontos de vista:
  - **arquitetura** - se refere aos atributos do sistema visíveis a um programador de linguagem de máquina e
  - **organização** - as unidades operacionais e sua interconexão que realizam a arquitetura, invisíveis ao programador.
- Vamos estudar a estrutura e a função de um computador.
  - **estrutura** - a forma em que os componentes estão interconectados e
  - **função** - a operação de cada componente individualmente.
  - Cada componente pode, por sua vez, de forma hierárquica, ser decomposto em subcomponentes, descrevendo a sua estrutura e função.

# O estudo é um desafio



Source: Wikipedia

- Um computador pode ser constituído por um simples microprocessador barato a um supercomputador com milhões de processadores.
- Há entretanto vários conceitos fundamentais que se aplicam consistentemente ao longo do tempo.
- Desempenho é o tema principal do nosso estudo.  
Refere-se a vários aspectos:
  - velocidade do processador,
  - velocidade e capacidade da memória,
  - velocidade de interconexão de dados.
- É um desafio projetar um sistema balanceado que considere todos esses aspectos de desempenho.



- **Arquitetura de computador:** refere-se aos atributos de um sistema visíveis a um programador, com um impacto direto na execução de um programa.  
Exemplos de atributos arquiteturais: conjunto de instruções (*instruction set*), número de bits usados para representar vários tipos de dados, mecanismos de entrada e saída, e técnicas de endereçamento de memória.



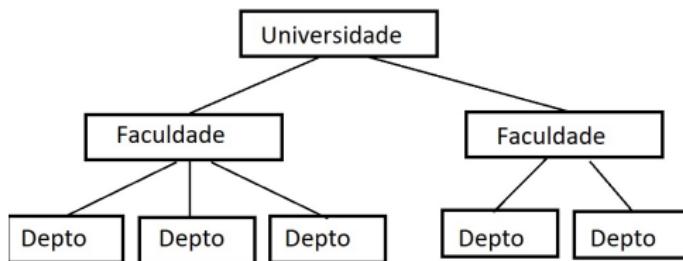
- **Organização de computador:** refere-se às unidades operacionais e sua interconexão que realizam as especificações arquiteturais, invisíveis ao programador. Exemplos de atributos organizacionais: detalhes de hardware transparentes ao programador, tais como sinais de controle, interface entre o computador e os periféricos, tecnologia de memória usada, etc.

- Exemplo: é uma questão de projeto **arquitetural** se o computador deve ter uma instrução de multiplicação.
- Mas é uma questão **organizacional** se a instrução deve ser implementada com uma unidade de multiplicação ou através de repetidas somas.

*Muitos fabricantes oferecem uma família de modelos de computadores, todos com a mesma arquitetura, mas com diferenças na parte organizacional. Resultam assim em modelos com preços e desempenhos diferentes, mas podendo executar os mesmos programas escritos.*

# Estrutura e função

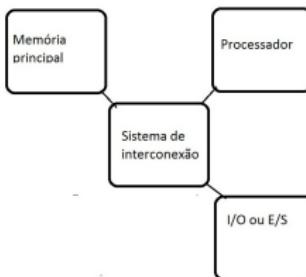
- Um computador possui milhões de componentes eletrônicos.
- Como vamos descrever um computador?
- Usamos o enfoque **hierárquico**. Exemplo:



- O projetista se preocupa com a descrição um nível por vez, descrevendo os componentes e sua interconexão.
- Os níveis são descritos de forma *top-down*, descrevendo-se os componentes de um nível, depois os de seus subníveis, e assim por diante.

# Estrutura e função

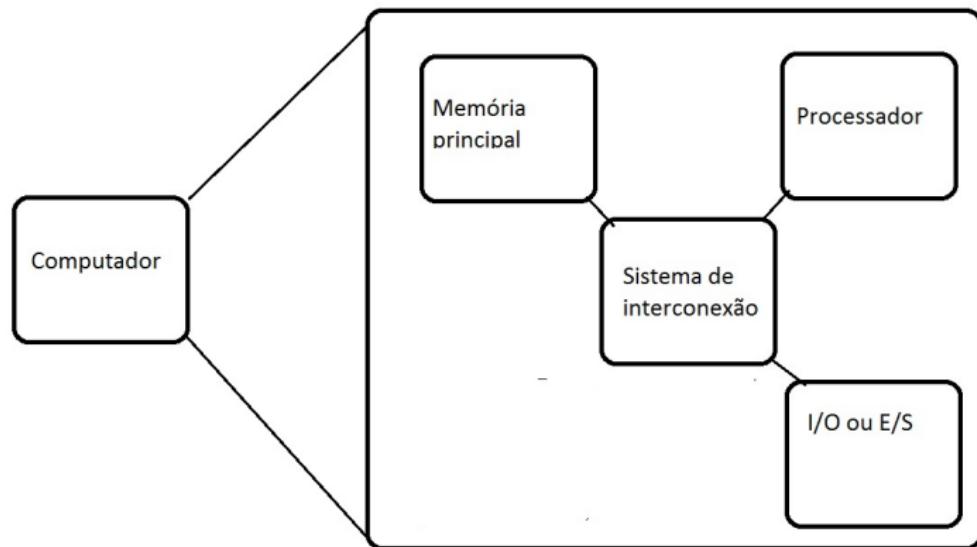
Em cada nível o projetista se preocupa com a estrutura e a função.



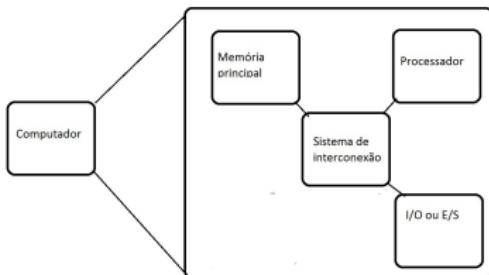
- **Estrutura:** a maneira em que os componentes são inter-relacionados. Como estão conectados?
- **Função:** a operação de cada componente individual como parte da estrutura. Para que serve?
  - Exemplos de funções: armazenamento de dados, movimentação de dados, processamento de dados, controle.

# Estrutura e função de um computador

Um computador tem como componentes:



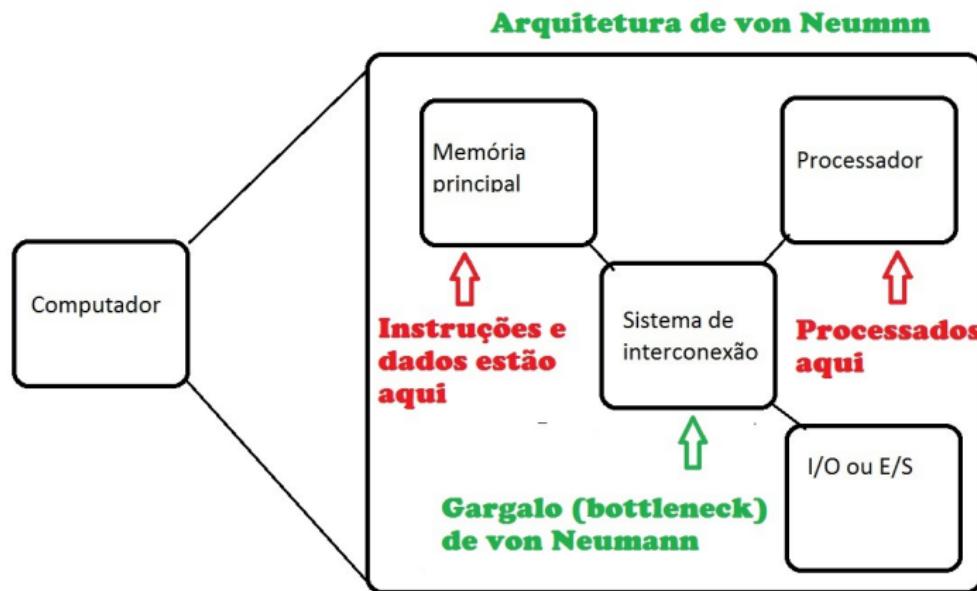
# Estrutura e função de um computador



- Processador ou CPU: tem a função de controlar a operação do computador e realizar o processamento de dados.
- Memória principal: a função é armazenar dados e instruções.
- I/O (ou E/S - entrada e saída): movimenta dados entre o computador e o ambiente externo.
- Sistema de interconexão: para comunicação entre CPU, memória e I/O, através de um barramento de sistema (*bus*).

# Estrutura e função de um computador

Um computador tem como componentes:



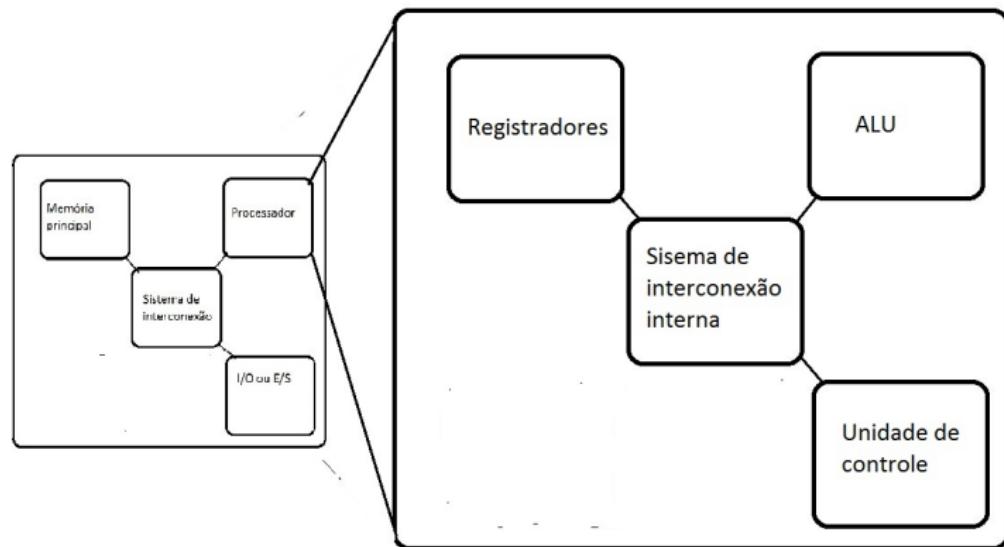


- John von Neumann (1903 - 1957)
- Húngaro-americano, matemático, cientista da computação
- Propôs a arquitetura de programa armazenado (conhecida como **Arquitetura de von Neumann**)
- Arquitetura de von Neumann é usada até hoje.
- Mesmo em um computador paralelo, cada componente é uma arquitetura de von Neumann.

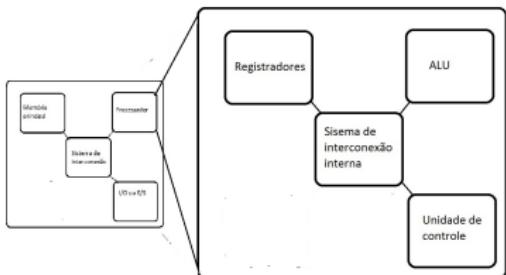
- O processador é o componente mais importante de um computador.
- Veremos técnicas para aumentar o seu desempenho.
- Dados e instruções residem na memória e são levados ao processador para processamento.
- Estudaremos os vários tipos de memória (hierarquia de memória) visando equilibrar a velocidade do processador com o tempo de acesso da memória.
- Processador e memória são implementados em pastilhas (*chips*) de Silício. Estudaremos brevemente a tecnologia VLSI.

# Estrutura e função do processador

Por sua vez, o processador tem como componentes:

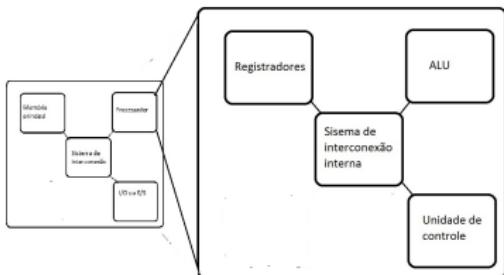


# Estrutura e função do processador



- Unidade de controle: controla a operação da CPU e portanto do computador.
- ALU (unidade aritmética e lógica): realiza as operações da função de processamento de dados.
- Registradores: fornece armazenamento interno para a CPU.
- Interconexão interna: mecanismo que faz a comunicação entre a unidade de controle, ALU e registradores.

# Unidade de controle



- Unidade de controle: fornece sinais de controle para a operação e a coordenação de todos os componentes do processador.
- Tradicionalmente o controle é feito por microprogramação (arquitetura CISC).
- Na arquitetura RISC, as instruções são mais simples e dispensa microprograma.
- Estudaremos ambas (CISC e RISC) no final da disciplina.

Demos uma visão geral.

Detalhes serão dados ao longo do curso.

# Como foi o meu aprendizado?

Quais itens abaixo têm a ver com a arquitetura e quais com a organização? (Às vezes a distinção não é tão clara. Não se preocupe se tiver dúvida.)

- 1 Representação de um número de ponto flutuante de dupla precisão.
- 2 Níveis de prioridade na execução de um processo.
- 3 Implementação do circuito somador com a técnica *carry-lookahead*.
- 4 Projeto do conjunto de instruções de máquina.
- 5 Como implementar o conjunto de instruções.
- 6 Usar um co-processador para aritmética de ponto flutuante.
- 7 Usar um co-processador especializado para processamento de imagem.
- 8 Técnicas de endereçamento.
- 9 Usar memória cache para acelerar o acesso.
- 10 Adotar técnicas de correção automática de erros de acesso à memória.

# História da evolução do computador: do ábaco ao Colossus, ENIAC, ..., Frontier

MAC0344 - Arquitetura de Computadores  
Prof. Siang Wun Song

Slides usados: <https://www.ime.usp.br/~song/mac344/slides02-evolution.pdf>

- Tecnologia expressa em gerações
  - Primeira geração: válvulas
  - Segunda geração: transistores
  - Terceira geração: circuito integrado VLSI
  - Novas gerações
- Evolução caracterizada por:
  - Aumento da velocidade do processador
  - Diminuição do tamanho dos componentes
  - Aumento da capacidade de I/O e velocidade

# História da Computação - Ábacos antigos

- Ábaco da Mesopotâmia (2700 - 2300 A.C.)
- Ábaco romano (1.o século D.C.)

Source: Museo Nazionale Romano



# Ábaco chinês

- Ábaco chinês
- Invento de Li Shou, oficial historiógrafo do imperador Huang-ti (2696 - 2598 a.C.).



Fonte: S. W. Song

- Acima temos a representação do número **2009**.
- O logotipo do [Instituto de Computação da Unicamp](#) é um ábaco indicando o ano de início do Bacharelado em Ciência da Computação da Unicamp.
- Procure esse logotipo e responda: **em que ano foi mesmo?**

# Ábaco chinês

加法口诀 Taboada para somar

加数	不进位加		进位加	
	直加	满五加	进十加	破五十加
一	一上一一下五去四	一去九进一		
二	二上二二下五去三	二去八进一		
三	三上三三下五去二	三去七进一		
四	四上四四下五去一	四去六进一		
五	五上五	五去五进一		
六	六上六	六去四进一六上一去五进一		
七	七上七	七去三进一七上二去五进一		
八	八上八	八去二进一八上三去五进一		
九	九上九	九去一进一九上四去五进一		

Source: Wikipedia

- Competição Feynman (caneta e papel) vs. Japonês (ábaco) (do livro de Richard Feynman - *Surely, You're joking, Mr. Feynman!*)

Richard Feynman (Prêmio Nobel em Física), pesquisador visitante do CBPF, Rio de Janeiro.

- Adição: ábaco ganhou fácil
- Multiplicação: ábaco ganhou
- Divisão: Feynman ganhou
- Raiz cúbica de 1729,03: Feynman escreveu imediatamente 12 no papel e calculou mais 3 decimais (12,002) quando o japonês gritou “12!”. Ganhou por sorte, pois esse número dado é quase um cubo perfeito.

# Ábaco chinês

Ábaco chinês aparece numa pintura.

*Along the River during the Qingming Festival* (Século 12)



Fonte: Wikipedia

A pintura, de mais de 5 metros de comprimento, descreve as atividades diárias de uma cidade.

[Clicar aqui para ver os detalhes da pintura em alta resolução.  
\(E vê se você acha o tal ábaco :-\)](#)

# Ábaco chinês

Ábaco chinês aparece na mesa de um boticário na pintura  
*Along the River during the Qingming Festival* (Século 12)



Fonte: Wikipedia

# Ábaco chinês

Ábaco chinês aparece na mesa de um boticário na pintura  
*Along the River during the Qingming Festival* (Século 12)



Fonte: Wikipedia

# Bagua e o sistema binário

- Durante a dinastia Zhou (ano 1.046 a.C. - 256 a.C.), o texto clássico *I Ching* (Livro das Mutações) tem como base o *Bagua* (oito trigramas), 八卦 baseado na numeração binária.
- O *Bagua* e os oito trigramas (3 bits):



坤 Kūn (Earth)	艮 Gēn (Mountain)	坎 Kǎn (Water)	巽 Xùn (Wind)
震 Zhen (Thunder)	離 Li (Fire)	兌 Dui (Lake)	乾 Qián (Heaven)

Fonte:

Wikipedia

# Sessenta e quatro hexagramas (do livro I Ching)

## Sessenta e quatro hexagramas (6 bits):

六十四卦構成表

坤(地)	艮(山)	坎(水)	巽(风)	震(雷)	离(火)	兑(泽)	乾(天)	↔ 上卦 ↓ 下卦

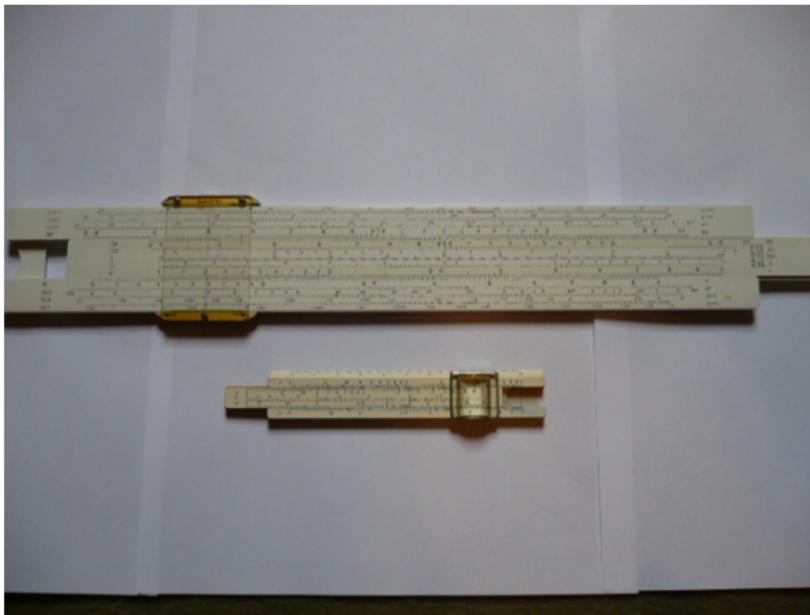
Fonte: Wikipedia

- I Ching era conhecido tanto como um oráculo como um livro da sabedoria.
- Baseado em I Ching, Leibniz (1703) desenvolveu a aritmética binária.

# Régua de cálculo

- Régua de cálculo (Século 17) (baseado no logaritmo)

Source: S. W. Song



# Geração 0 - “Computadores” mecânicos 1642 - 1945

- Wilhelm Schickard - 1623

Source: Universität Tübingen



- B. Pascal - 1645

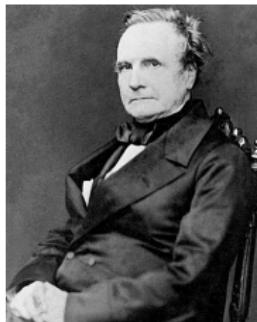
Source: Univ. of Vienna



# Charles Babbage e Ada Lovelace



Source: IEEE



Wikipedia



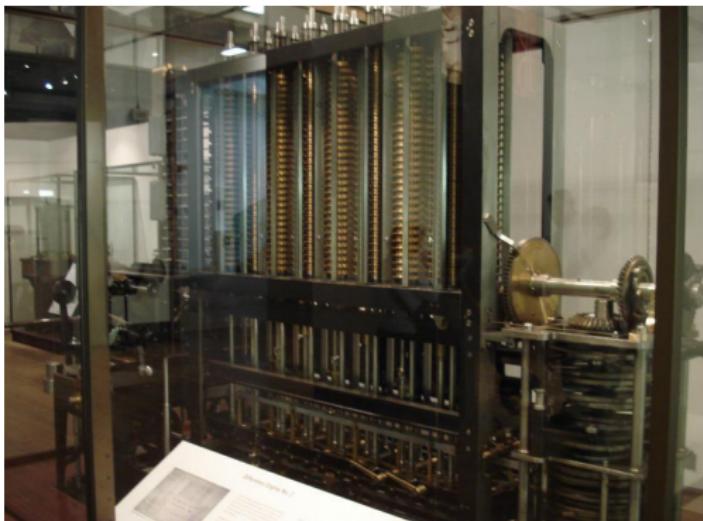
- Charles Babbage (1792-1871)  
“Pai do computador”  
Charles Babbage Award: estabelecido em 1989.
- Ada Lovelace (1815-1852)  
“Primeira programadora”

# Geração 0 Computadores mecânicos - Babbage

- Charles Babbage (1792-1871)

*Difference Engine*: executaria apenas um algoritmo (cálculo de tabela para navegação marítima)

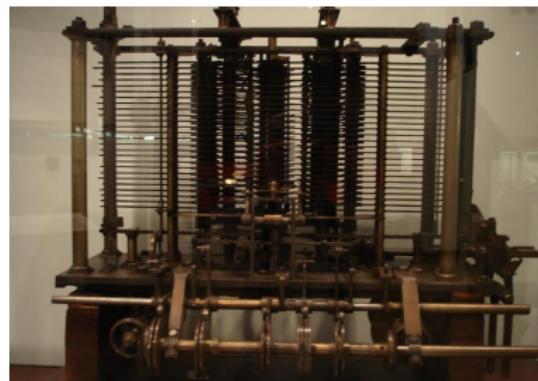
Source: London Science Museum



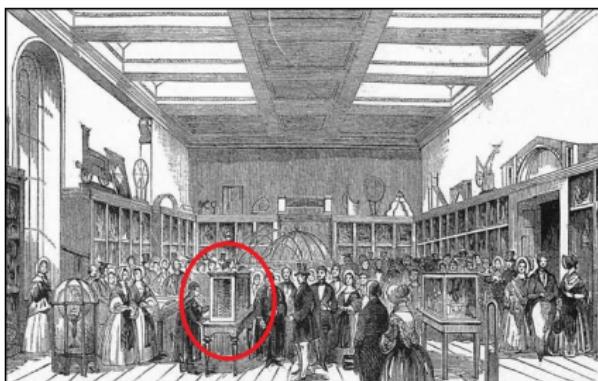
# Geração 0 Computadores mecânicos - Babbage

- *Analytical Engine*: Máquina de uso geral, mas não ficou operacional. Tem 4 partes: armazenamento, computação, entrada, saída  
Primeira programadora: Ada Lovelace.

Source: London Science Museum



Parte da máquina em exibição - Source: Wikipedia



# Geração 0 Computadores mecânicos - MARK I (1944)

- H. Aiken: MARK I (1944)

Usava relés mecânicos - Ciclo de relógio de 0,3 segundos

Source: [www.ibm.com](http://www.ibm.com)



# Invenção da válvula



Source: history-computer.com

- Sir John Ambrose Fleming (1849-1945)
- Inglês, engenheiro eletricista, físico
- Inventor da primeira válvula (*vacuum tube*) em 1904.

Fonte: <https://history-computer.com/ModernComputer/Basis/diode.html>

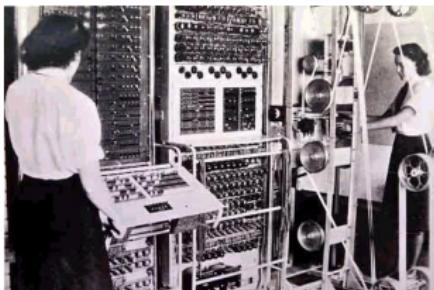
# Geração 1 - Válvulas 1945 - 1955

Source: S. W. Song



# Geração 1 Válvulas - Colossus (1943)

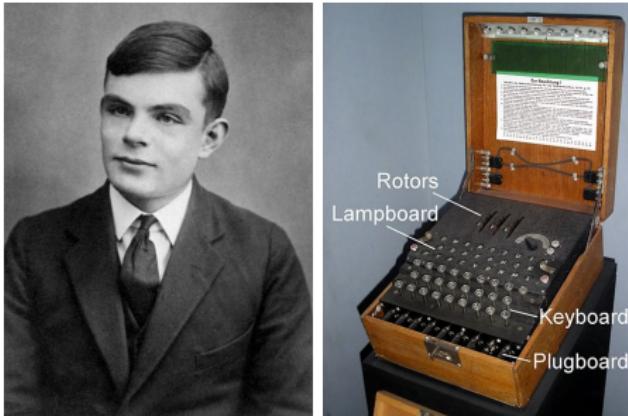
Source: Wikipedia



- Colossus (1943) foi o primeiro computador eletrônico.
- Desenvolvido por Thomas Flowers assistido por Sidney Broadhurst e William Chandler.
- Construído pelo governo britânico, com contribuição de Alan Turing, para decifrar mensagens codificadas por ENIGMA.
- Entrada por fita de papel. Programado por chaves e plugs.
- A existência do Colossus ficou em segredo até os anos 1970.

# Geração 1 Válvulas - Colossus (1943)

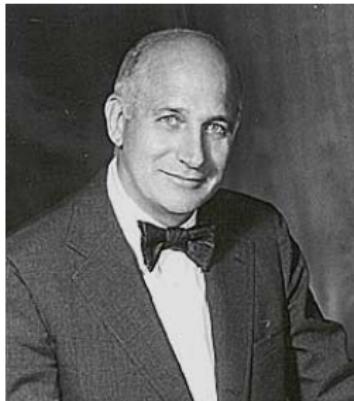
Source: Wikipedia



- Colossus foi usado pela equipe de decodificadores de Bletchley Park liderada por Alan Turing.
- ENIGMA usava 3 rotores de 5 (dando  $10^{19}$  configurações). Nova versão usava 4 rotores de 8 ( $10^{22}$  configurações).
- Tendo um *crib* (trecho do código decifrado), *bombe*s eram usados para descobrir a configuração inicial, antes de Colossus.

# Geração 1 Válvulas - ENIAC (1946)

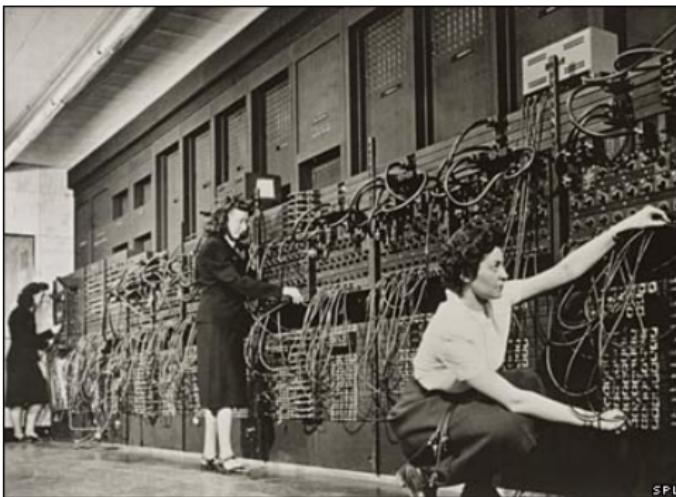
Source: history.computer.com



- Projetado por John Mauchly e John Presper Eckert - da Univ. de Pennsylvania
- Fundaram depois a UNIVAC.
- Até o conhecimento do Colossus nos anos 1970, ENIAC foi considerado como o primeiro computador eletrônico.

# Geração 1 Válvulas - ENIAC (1946)

Source: British Broadcasting Corporation - BBC



- 18.000 válvulas - Programada por 6.000 chaves
- 30 toneladas - ciclo relógio 200 micro-segundos (5 KHz)

# Geração 1 Válvulas - EDVAC (1949)

Source: Wikipedia



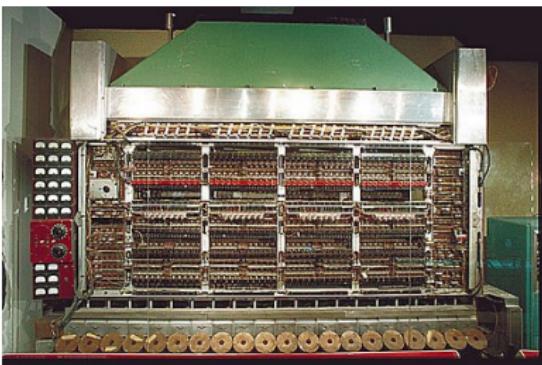
- Primeiro computador que adotou a **arquitetura de von Neumann** (programa armazenado).
- Construção do EDVAC foi proposto por John Mauchly e John Eckert (que construíram ENIAC) e finalizada em 1949
- Computador binário com memória de 1000 palavras de 34 bits.
- Adição leva 864 micro-segundos e multiplicação 2900 micro-segundos.

# Geração 1 Válvulas - IAS (1952)

Source: Wikipedia



Source: Wikipedia



- Desenvolvido em 1952 por John von Neumann em Princeton.
- Computador binário com uma memória de 1024 palavras de 40 bits.
- Cada palavra armazena duas instruções de 20 bits.
- Peso de 450 kg.

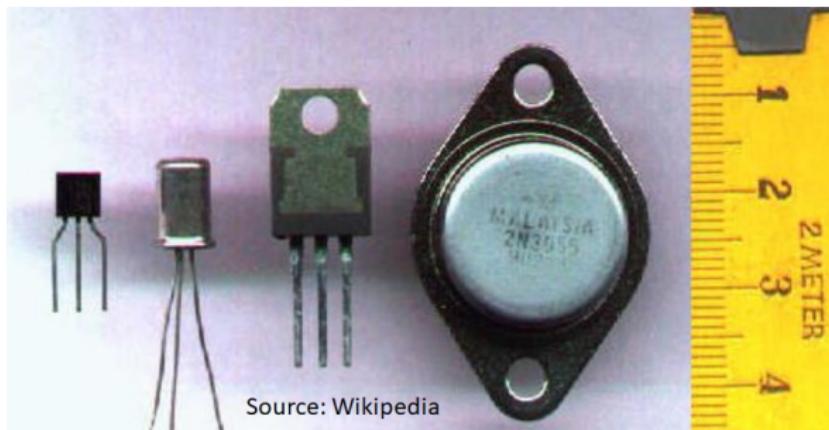
# Invenção do transistor

Source: Wikipedia



- A invenção do transistor foi atribuída a três membros da Bell Labs, em 1947.
- Foto, da esquerda para a direira: John Bardeen, William Shockley e Walter Brattain.

# Geração 2 - Transistores 1955 - 1964



Na década dos anos 60 algumas universidades começaram a receber os primeiros computadores (exceto um, todos da Geração do Transistor).

- Burroughs B-205 em 1960 na PUC-Rio (1.600 válvulas)
- **IBM 1620 em 1962 na USP** (transistorizado)
- IBM 1130 em 1965 na PUC-Rio
- IBM 1130 em 1967 na UFRGS
- IBM 1130 em 1968 na Unicamp
- IBM 1130 em 1968 na UFCG

# Geração 2 - Transistores 1955 - 1964

- IBM 1620 - primeiro computador da USP (1962)
- Memória de ferrite de 20.000 dígitos decimais (cada dígito decimal codificado com 6 bits).
- Entrada e saída por cartão perfurado.

Source: IBM



# Geração 2 - Transistores 1955 - 1964

- IBM 1620 - primeiro computador da USP (1962)
- Memória de ferrite de 100.000 bits (12,5 Kbytes)

Source: Science Museum - London



Vídeo no Youtube:

Vídeo: IBM 1620 USP - [clicar aqui](#)

Nesse vídeo aparecem os Profs. Imre Simon (*In Memoriam*), Tomasz Kowaltowski e Cláudio Lucchesi.

# Primeiros cursos de Computação

Os primeiros cursos de Bacharelado em Ciência da Computação:

- UNICAMP/IMECC (**1969**) formou a primeira turma em 1972



- USP/IME (**1971**) formou a primeira turma em 1974



Em **1969** a Universidade Federal da Bahia criou o curso de bacharelado de processamento de dados.

Fonte: UFBA - Computação - linha do tempo - primórdios.



## Pergunta

Esses primeiros cursos de Computação produziram os primeiros formados em Computação,

a pergunta é *de onde vieram os professores desses primeiros cursos?*

Problema do ovo ou galinha



# IBM-1620 na USP e a criação do CCN

No caso do Estado de São Paulo, a resposta à pergunta anterior gira em torno do **IBM-1620** e a criação do **CCN - Centro de Cálculo Numérico**. Muitas informações sobre CCN estão no texto:

[História do Centro de Cálculo Numérico \(CCN\) e suas Contribuições \(Clicar aqui para o texto - versão estendida com 28 páginas\).](#)

## História do Centro de Cálculo Numérico (CCN) e suas Contribuições\*

Isu Fang, Paulo Feofiloff, Tomasz Kowaltowski, Cláudio Leonardo Lucchesi,  
Valdemar Waingort Setzer, Siang Wun Song, Routh Terada

*In Memoriam  
Imre Simon  
José Dion de Melo Teles  
Paulo de Souza Moraes  
Ronaldo Zwicker  
e os que nos deixaram ...*

Ver também slides de uma palestra:

[Computação em SP e no Brasil - desde o início - "causos" e história](#)

# IBM-1620 na USP e a criação do CCN

- IBM 1620 - primeiro computador da USP (1962)
- Adquirido por iniciativa dos Professores da USP (com recursos orçamentários das três unidades abaixo):
  - J. O. Monteiro (Escola Politécnica)
  - Oscar Sala (Fac. de Filosofia, Ciências e Letras)
  - Flávio Fausto Manzoli (Faculdade de Economia e Administração)
- Para acomodar o IBM-1620 foi criado o **Centro de Cálculo Numérico (CCN)**, que mais tarde transformou-se no CCE (Centro de Computação Eletrônica).
- A Escola Politécnica forneceu o local para o CCN - no prédio do Biênio.



Source: IBM

# IBM-1620 na USP e a criação do CCN

O CCN (mais tarde CCE) foi uma incubadora de futuros talentos, com inquestionável contribuição:

- na formação dos **primeiros docentes** dos cursos de BCC da UNICAMP e da USP, em particular e,
- na ciência e tecnologia do País, em geral.

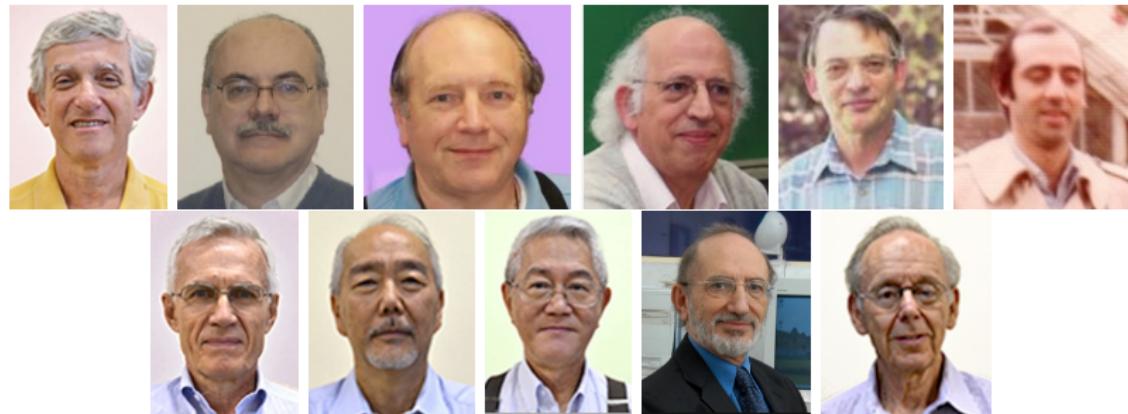
Dirigentes/estagiários do CCN e início do CCE tiveram depois importantes cargos e honrarias, como:



Presidente do CNPq e do Serpro, Presidente da Prodam e Elebra, Ordem Nacional do Mérito Científico (um Grã-Cruz e 3 Comendadores), Academia Brasileira de Ciências (um), Prêmio Mérito Científico da SBC (3), dois Diretores do IC/Unicamp, um Diretor do IME/USP, dois Professores Eméritos, Diretor-Presidente do NIC.br, 13 Professores do BCC do IME/USP, 5 Professores do BCC do IC/Unicamp e 4 Professores de Computação da FEA/USP..

# Professores do BCC do IME/USP oriundos do CCN/CCE

- Professores do BCC do IME/USP: Arnaldo Mandel, Ernesto de Vita Júnior, Geraldo Lino de Campos, Jorge Stolfi, Imre Simon, István Simon, Paulo de Souza Moraes, Paulo Feofiloff, Routh Terada, Siang Wun Song, Silvio Ursic, Tomasz Kowaltowski, Valdemar Setzer



# Professores do IC/Unicamp e FEA/USP oriundos do CCN/CCE

- Professores do BCC da Unicamp: Cláudio Leonardo Lucchesi, István Simon, János Simon, Jorge Stolfi, Tomasz Kowaltowski



- Professores de Computação na FEA/USP: Isu Fang, Nicolau Reinhard, Ronaldo Zwicker, Tomasz Kowaltowski



# Entrada por cartão

Source: S. W. Song



Source: Univ. Stuttgart

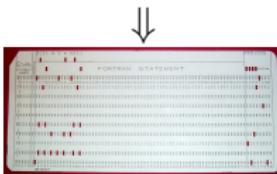


# Procedimento complicado para rodar um programa

1. Início: Folha de codificação



2. Cartão perfurado



3. Processamento em lote

```
1      K=1
2      6      IF (K.EQ.11) GO TO 8
3      READ I,J
4      IF (J.GT.I) GO TO 65
5      GO TO 66
6      65      WRITE(6,6002) J,I
7      6002    FORMAT(' ',I3,' IS GREATER THAN ',I3)
8      K=K+1
9      GO TO 6
10     66      WRITE(6,6001) J
11     6001    FORMAT(' ',I3,' IS GREATER THAN ',I3)
12     K=K+1
13     GO TO 6
14     8      CALL EXIT
15     END
```

4. Pegar resultado. Errou? Goto 1.

(Você ainda tem  $n := n - 1$   
créditos para este EP :-)

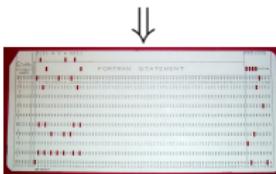
Como Vocês são sortudos e  
não sabiam :-)

# Procedimento complicado para rodar um programa

1. Início: Folha de codificação



2. Cartão perfurado



3. Processamento em lote

```
1      K=1
2      6      IF (K,EQ,11) GO TO 8
3      READ,I,J
4      IF (J,GT,I) GO TO 65
5      GO TO 66
6      65      WRITE(6,6002)I,J
7      6002    FORMAT(' ',13,' IS GREATER THAN ',I3)
8      K=K+1
9      GO TO 6
10     66      WRITE(6,6003)I,J
11     6003    FORMAT(' ',13,' IS GREATER THAN ',I3)
12     K=K+1
13     GO TO 6
14     8      CALL EXIT
15     END
```

4. Pegar resultado. Errou? Goto 1.

(Você ainda tem  $n := n - 1$   
créditos para este EP :-)

Como Vocês são sortudos e  
não sabiam :-)

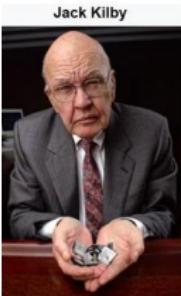
# Geração 2 - Transistores 1955 - 1964

Fonte: Wikipedia



- DEC PDP-1 (1960) - Primeiro mini-computador com 50 vendidos
- IBM-1401 (1961) - Pequeno computador comercial com enorme sucesso
- IBM-7094 (1962)- Computador para aplicações científicas
- Burroughs B-5000 (1963) - Projetada para linguagem de alto nível: Algol 60
- Control Data CDC-6600 (1964) - Uso de múltiplas unidades funcionais

# Invenção do circuito integrado



Discurso Prêmio Nobel 2000

Source: Wikipedia

- Jack Kilby, um engenheiro da Texas Instruments, produziu o primeiro circuito integrado em 1958.
- Recebeu o Prêmio Nobel de Física em 2000. O seu [discurso no recebimento do Prêmio Nobel](#) teve como título *“Turning Potential into Reality: The Invention of the Integrated Circuit”*.

- Em 1958 Jack Kilby (da Texas Instruments) produziu o primeiro circuito integrado reunindo transistores, resistores e capacitores em uma pastilha de semicondutor.
- Jack Kilby recebeu o Prêmio Nobel em Física (2000).
- IBM-360 (1964)  
Máquina microprograma  
Primeira de uma família
- Digital PDP-8 (1965)  
Primeiro mini-computador com grande venda (50.000 vendidos)
- Digital PDP-11 (1970)  
Mini-computador de grande sucesso dos anos 70

# Geração 4 - VLSI 1980 - hoje

Source: Wikipedia



- VLSI = *Very Large Scale of Integration*: componentes eletrônicos minúsculos são implementados em silício.
- Essa tecnologia é responsável pelos avanços fantásticos que temos hoje.
- Suriram os primeiros computadores pessoais (final dos anos 70) com duas grandes famílias de processadores: Intel e Motorola.
- Processador numa só pastilha contendo milhões de transistores (Pentium 4 com 42 milhões de transistores).
- Em 2019: AWS Gaviton2 64-core ARM-based (7 nm technology) chip com 30 bilhões de transistores

[Wikipedia: transistor count](#)

# Primeiro micro do IME-USP

- Prológica S700 (1982-1983)  
Processador Z-80 (8 bits)  
Emprestado por um ano ao IME - cortesia de um dos sócios da Prológica.



- Scopus Nexus 1600 (1984): Processador Intel 8088 (16 bits), 8 MHz, 704 Kbytes RAM, 2 drives diskettes 5 1/4"  
Comprado com verba FAPESP - mais de US\$ 10.000,00.

Source: Scopus



# Meios de armazenamento

- Diskette flexível de 8" (175K) e diskette de  $5\frac{1}{4}$ " (360K).
- Source: S. W. Song
- Diskette de  $3\frac{1}{2}$ " (1,44M) e disco CD/DVD.



# Evolução da Computação

- O Mark I tinha ciclo de 0,3 segundos; o ENIAC 200 micro-segundos
- Processador hoje: vários GHz - menos de um nanosegundo de ciclo
- Processador de hoje milhões de vezes mais rápido que o ENIAC
- Computação paralela aumenta mais ainda o poder computacional.
- Lista TOP500 apresenta os 500 computadores mais velozes do mundo, com base no benchmark Linpack (sistema linear). Mais sobre TOP500 na próxima aula.
- Medida de desempenho em FLOPS (*Floating Point Operations per Second*): MFLOPS, GFLOPS, TFLOPS ...

# Relógio: ciclo de relógio e frequência



- Como coordenar atividades síncronas?
- Remar de forma síncrona numa galera romana: tambor

# Relógio: ciclo de relógio e frequência



- Como coordenar atividades síncronas?
- Desfile de Escola de Samba: bateria

# Relógio: ciclo de relógio e frequência

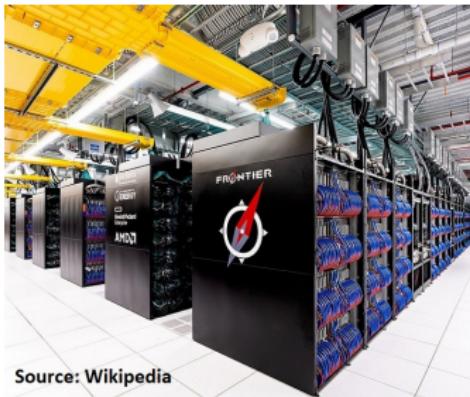
Ciclo do relógio



Exemplo: ciclo = 1 ns  
Frequência = 1 GHz

Frequência = número de ciclos em 1 segundo

- Como coordenar atividades síncronas?
- Atividades num computador: relógio
- Cada operação leva um certo número de ciclos. Por exemplo: ler a memória cache leva 10 ciclos
- Maior a frequência (ou menor o ciclo) mais veloz o computador
- Mas uma maior frequência produz mais calor.



Primeiro computador a chegar a alcançar ExaFLOPS.

- Frontier - Oak Ridge National Lab (E.U.A.)
- Total de 8.730.112 cores ou núcleos
- Processadores AMD EPYC 64C 2GHz
- LINPACK 1.102 PetaFLOPS ou 1,102 ExaFLOPS
- Velocidade de pico 1,686 ExaFLOPS

# O que vem depois ?

- Depois de ExaFLOPS vem ZettaFLOPS.
- Em que ano chegaremos à era *Zetta Computing*?
- E depois da computação VLSI com Silício, que novas tecnologias virão?

*Computers are incredibly fast, accurate, and stupid;  
humans are incredibly slow, inaccurate and brilliant;  
together they are powerful beyond imagination.*

- *Albert Einstein*

# Como foi o meu aprendizado?

- Seja curioso: não se esqueça de ver qual será o computador mais veloz do mundo.
- Anúncio de uma nova lista top500 em novembro deste ano.
- Que processadores são usados no novo Top 1?
- Quantos processadores (ou *cores*) no novo Top 1?
- Alguns computadores brasileiros no novo Top500?

# Próximo assunto: Supercomputadores da lista TOP500



- Próximo assunto: Supercomputadores da lista TOP500.
- Estado-de-Arte da supercomputação de alto desempenho. Qual o seu desempenho? Quem compra? Quem fabrica? Para que servem? Que sistema operacional usa? etc.
- Não percam!

# Evolução da Computação de Alto Desempenho sob a Ótica da Lista TOP500

MAC0344 - Arquitetura de Computadores  
Prof. Siang Wun Song

Slides usados: <https://www.ime.usp.br/~song/mac412/top500.pdf>

IME-USP

# Computação de alto desempenho

- Veremos o Estado-de-Arte da Computação de Alto Desempenho, sob a ótica da TOP500. Passar a Lista 1 de exercícios.
- Ao final dessa aula:
  - Vocês verão que hoje todos os supercomputadores usam computação paralela, alguns com milhões de processadores (*cores* ou núcleos).
  - Isso se deve ao avanço da tecnologia VLSI (microeletrônica) com o aumento da capacidade de uma pastilha de Silício, que pode conter cada vez mais componentes eletrônicos minúsculos.
  - Aumentar a frequência de relógio é uma forma de aumentar o desempenho, mas tem o limite de dissipação de calor que impede o seu rápido aumento. Daí a solução por computação paralela.



TOP500: lista dos 500 computadores mais poderosos do mundo.

- Divulgada duas vezes por ano: em junho e novembro
- Interesse tanto para fabricantes como para compradores potenciais
- Benchmark: LINPACK - solução de um sistema linear de  $n$  equações a  $n$  incógnitas.
- Os 500 computadores com melhor desempenho LINPACK entram na lista TOP500.
- Muito material é disponível no site: <http://www.top500.org/>

# Lista TOP500



Dr. Jack Dongarra



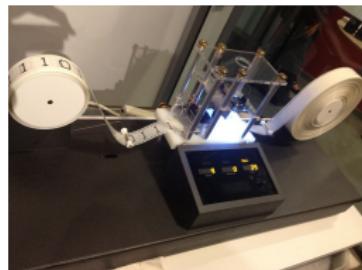
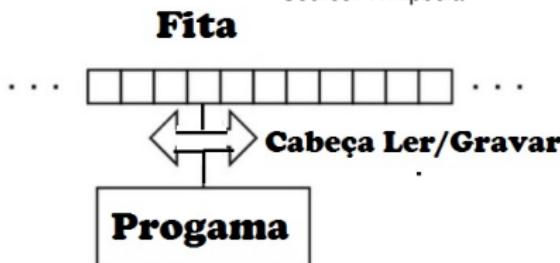
Source: ACM Association for  
Computing Machinery

- A lista Top500 é mantida e administrada por [Jack Dongarra](https://en.wikipedia.org/wiki/Jack_Dongarra) - [https://en.wikipedia.org/wiki/Jack\\_Dongarra](https://en.wikipedia.org/wiki/Jack_Dongarra).
- Pesquisador na área de Algoritmos Numéricos.
- Pelas suas contribuições no desenvolvimento de algoritmos paralelos, foi recipiente do **Turing Award** de 2021.

# Turing Award - em homenagem a Alan Turing



Source: Wikipedia



- Alan Mathison Turing (1912 - 1954)
- Inglês, matemático, cientista da computação, cripto-analista. Considerado fundador da Teoria da Computação.
- Máquina de Turing, Computabilidade, Indecibilidade (Problema da Parada)
- Turing Award: prêmio anual criado em 1966, considerado o Prêmio Nobel da Computação. Em 2014, o prêmio aumentou para US\$ 1 milhão.

# Computação de Alto Desempenho

Medida de desempenho:

1 FLOPS = uma operação ponto flutuante por segundo

- Mega FLOPS ou MFLOPS =  $2^{20} \cong 10^6$  op. aritméticas por segundo
- Giga FLOPS ou GFLOPS =  $2^{30} \cong 10^9$
- Tera FLOPS ou TFLOPS =  $2^{40} \cong 10^{12}$
- Peta FLOPS ou PFLOPS =  $2^{50} \cong 10^{15}$
- Exa FLOPS ou EFLOPS =  $2^{60} \cong 10^{18}$
- Zetta FLOPS ou ZFLOPS =  $2^{70} \cong 10^{21}$
- Yotta FLOPS ou YFLOPS =  $2^{80} \cong 10^{24}$



Primeiro computador a chegar a alcançar ExaFLOPS.

- Frontier - Oak Ridge National Lab (E.U.A.)
- Total de 8.730.112 cores ou núcleos
- Processadores AMD EPYC 64C 2GHz
- LINPACK 1.102 PetaFLOPS ou 1,102 ExaFLOPS
- Velocidade de pico 1,686 ExaFLOPS

# Supercomputador número 2 da lista TOP500

Até antes de junho/2022, o Fugaku ocupava a posição Top 1.



Fugaku supercomputer by Fujitsu. Picture: Reuters

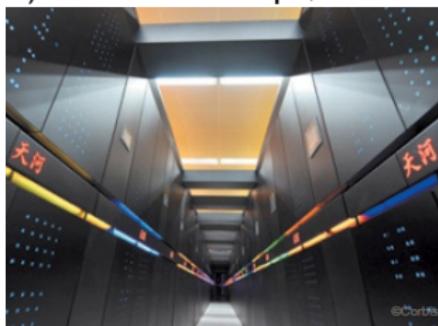
- Fugaku (Japão)
- Fujitsu 48-core ARM processor
- Total de 7.299.072 cores
- LINPACK 415,5 PFLOPS
- Velocidade de pico 513,8 PFLOPS

# Alguns campeões anteriores



Source: Jack Dongarra, Report on the Sunway TaihuLight System, June 2016

Sunway (China) SW26010 chip (TOP 1 - Junho 2016 a junho 2018)



Tianhe (China) Xeon Phi (TOP 1 - Junho 2013 a novembro 2015)

# Alguns campeões anteriores



Roadrunner (U.S.A) - IBM PowerXCell (TOP 1 - Junho 2008 a junho 2009)



Blue Gene (U.S.A) - IBM PowerPC 440 (TOP 1 - Novembro 2005 a junho 2007)



Earth Simulator (Japan) - NEC SX-ACE (TOP 1 - Junho 2002 a junho 2004 )

Qual será o primeiro ... em novembro deste ano?

Em novembro deste ano sai uma nova lista  
TOP500

e um novo TOP 1 pode surgir.

# Uma máquina da USP esteve na TOP500

- Pergunta: quantas máquinas brasileiras na lista TOP500 em junho do ano passado ainda estão na presente lista?  
Os curiosos podem consultar o site top500.

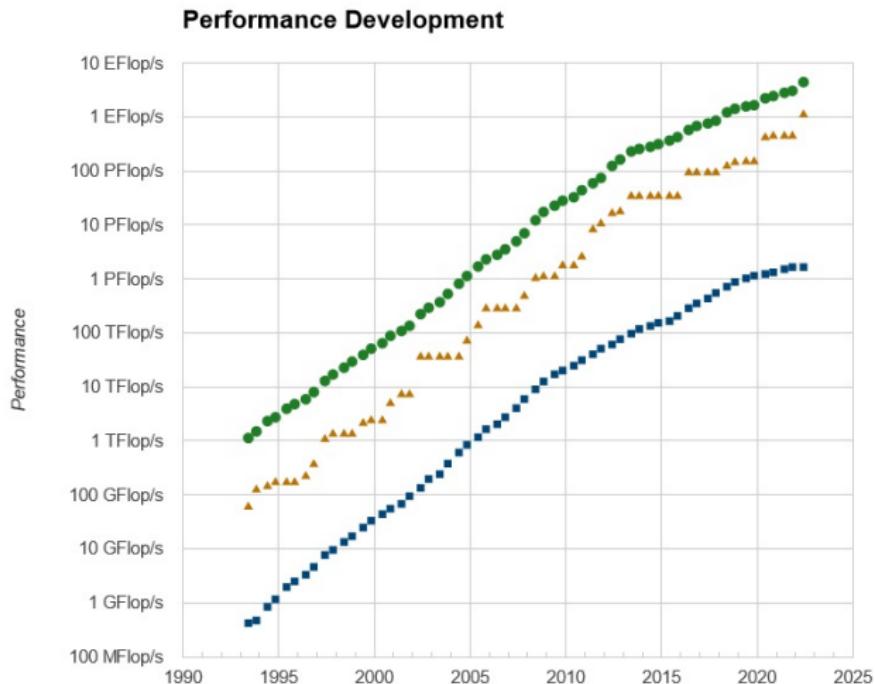


*Na lista TOP500 de novembro/2006 a USP estava na posição 363, com 3,182 TFLOPS Linpack :-)*

*A alegria só durou 6 meses pois saiu da lista em junho/2007 - :-()*

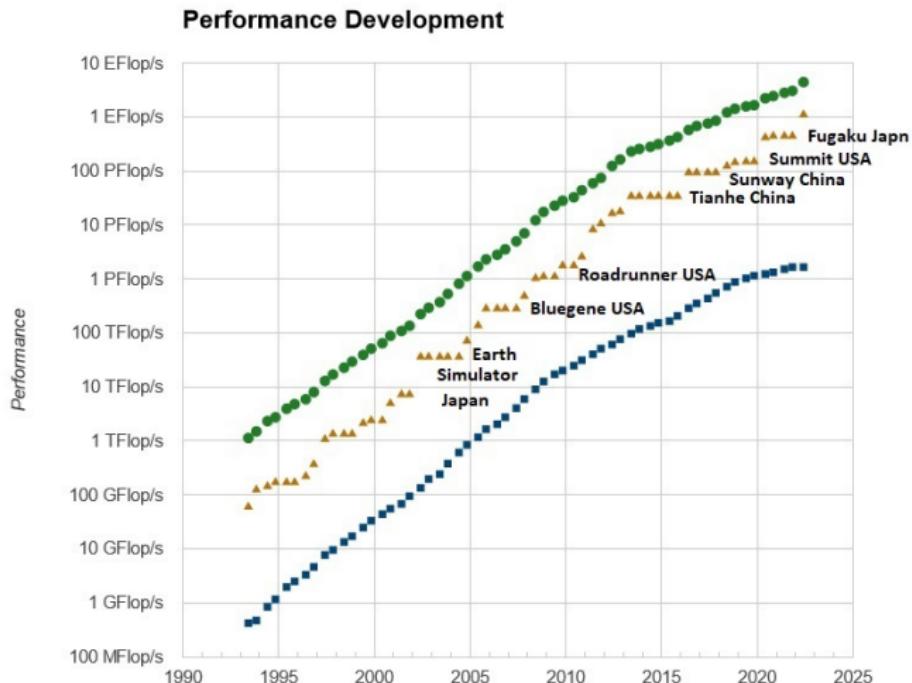
# Desempenho ao longo do tempo

Note o crescimento exponencial.



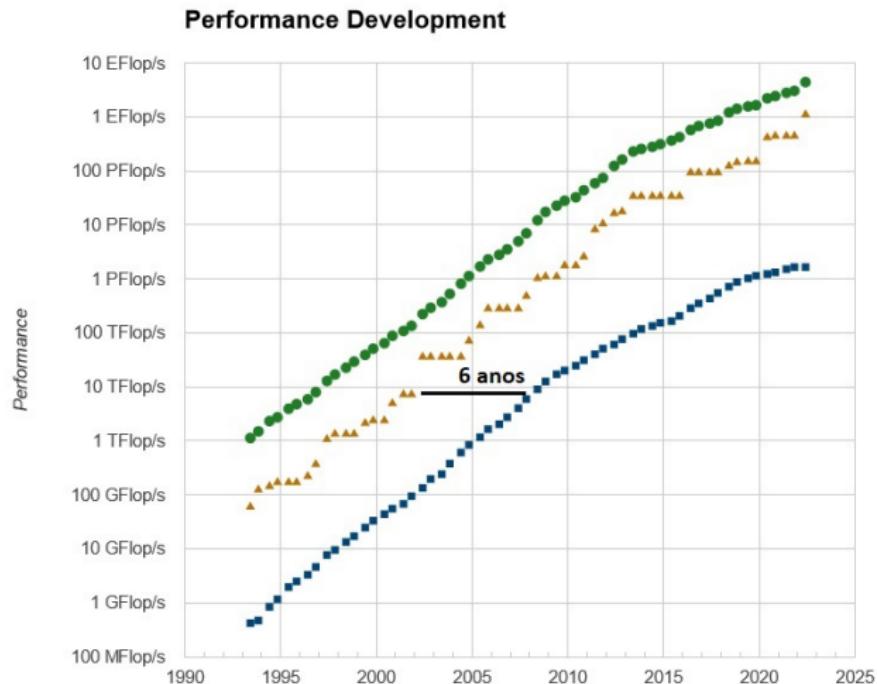
# Desempenho ao longo do tempo - Alguns Campeões

Note o crescimento exponencial.



# O primeiro se torna o último em 6 anos

Note o crescimento exponencial.



# Meu computador já foi TOP 1



Fonte: NVIDIA

- O desktop que eu tinha na minha sala da UFABC quando fui professor visitante lá já foi TOP1 :-)  
Duas placas NVIDIA Geforce GTX-680: 3.072 processadores, veloc. pico de 4,5 TFLOPS.
- O número 1 da TOP500 no período 1997 a 2000 é o Intel ASCI Red com veloc. de pico de 1,3 TFLOPS.
- Esse meu computador seria o número 1 da TOP500 até novembro 2000 :-)

# Perguntas

- O que vem depois de ExaFLOPS?

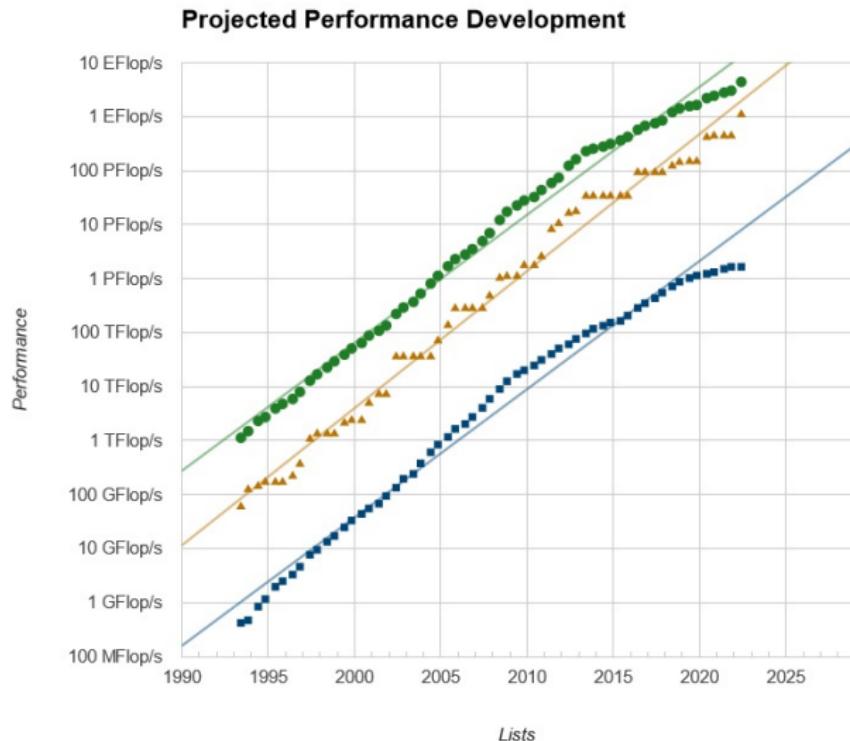
Resposta: ZettaFLOPS

$$1 \text{ ZettaFLOPS} = 2^{70} \cong 10^{21}$$

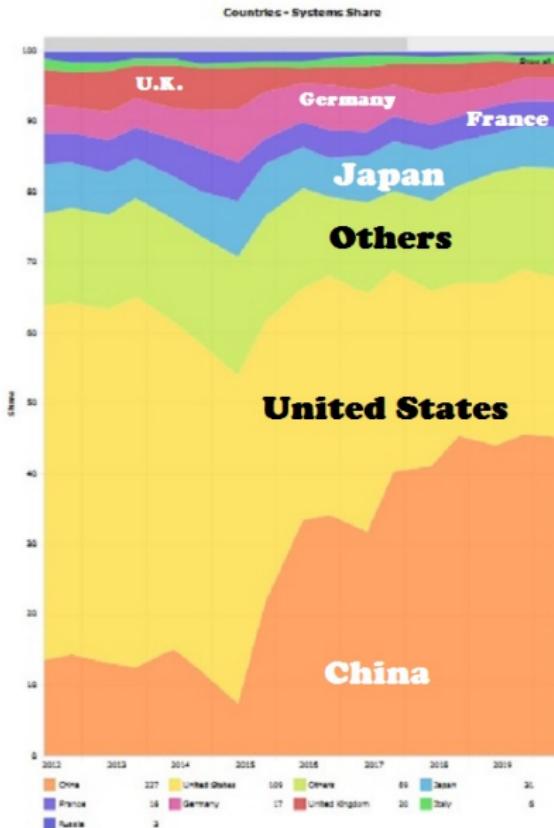
- Pergunta: Em que ano teremos computadores de desempenho de ZettaFLOPS?

Ver a próxima figura.

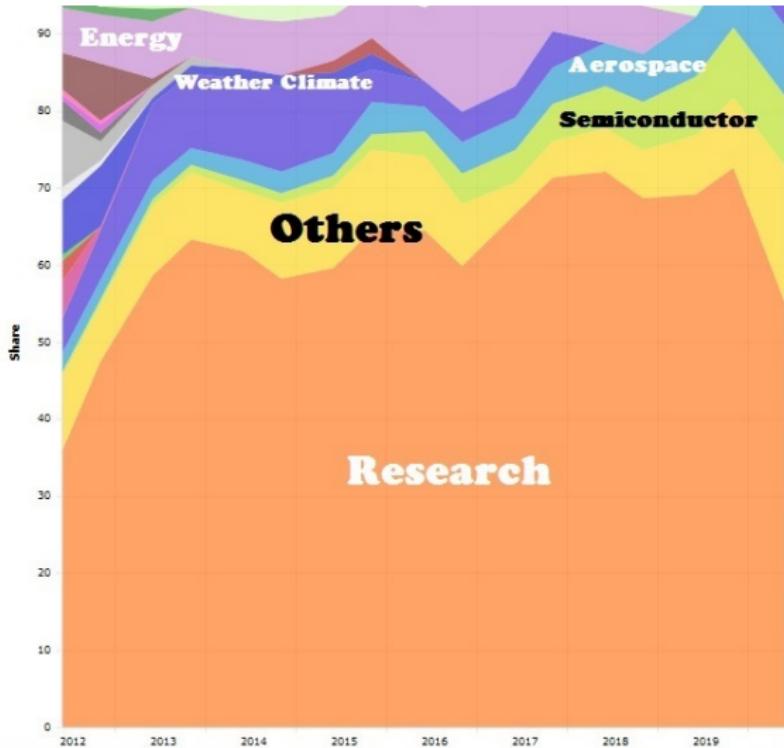
# Projeção do Desempenho



# Países Compradores

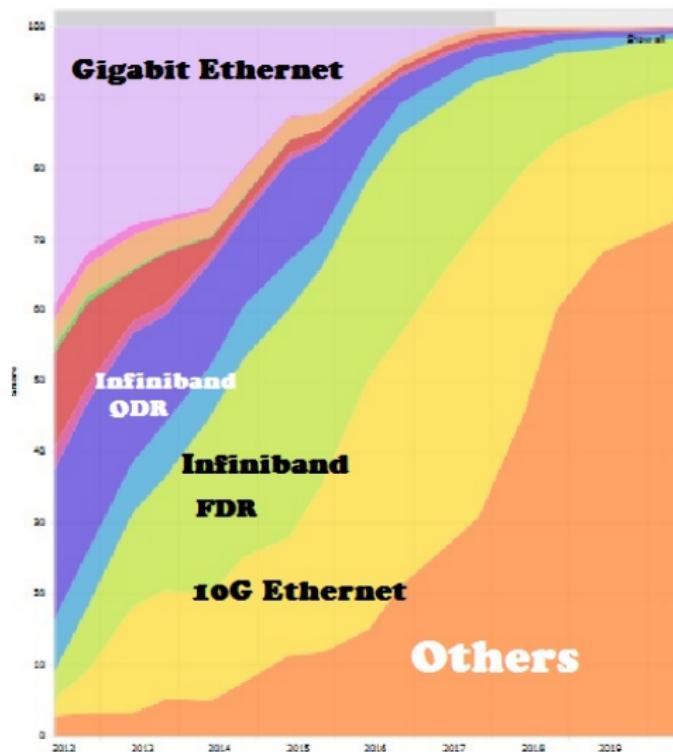


# Aplicações



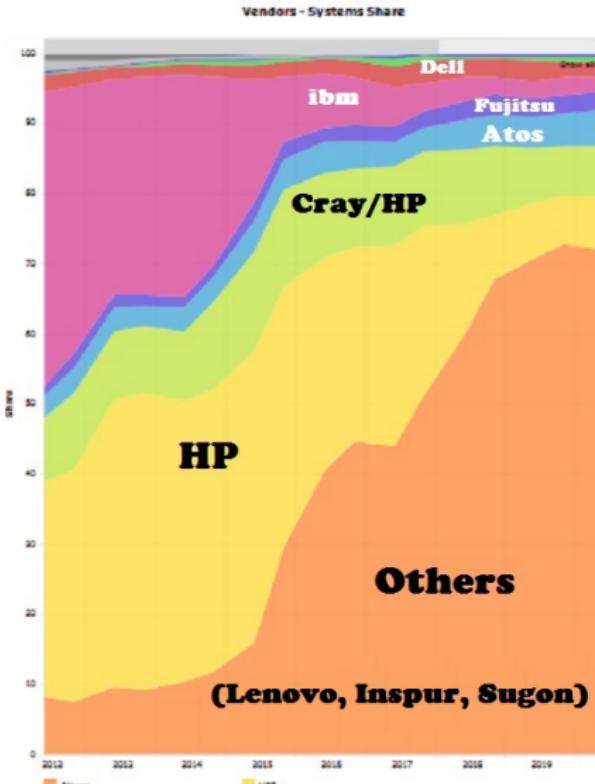
- . Petróleo
- . Previsão tempo e clima
- . Indústria: aviação, automóveis

# Interconexão



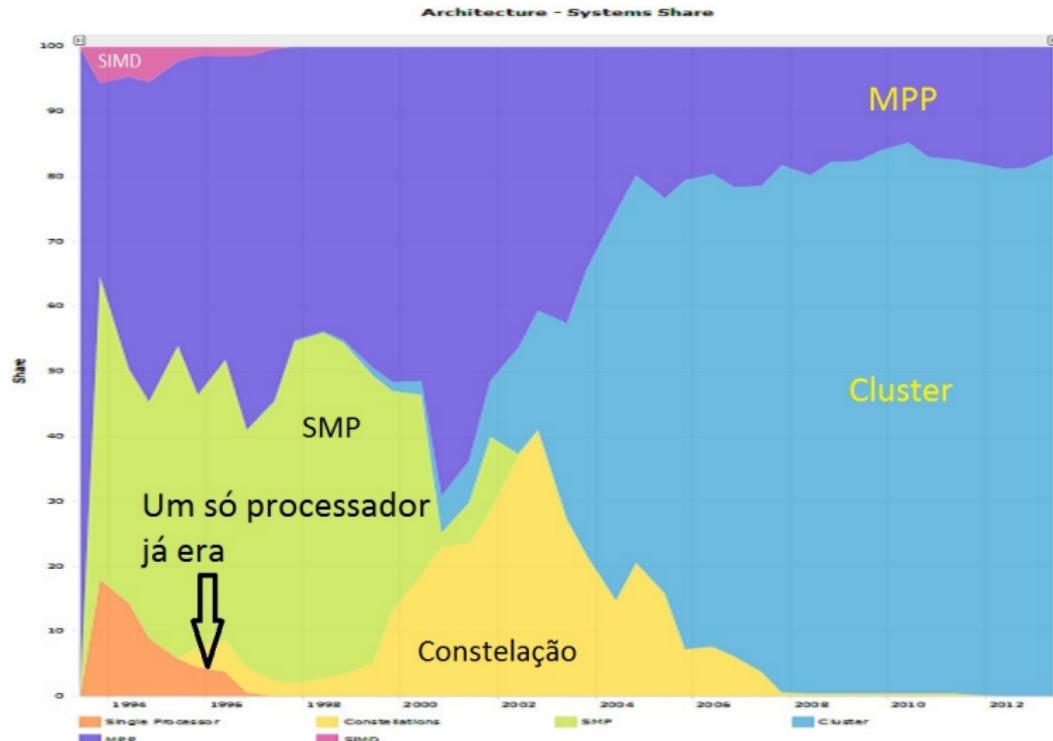
- Área de alta competição.
- Difícil prever qual melhor rumo a seguir.
- Algumas empresas permanecem; outras não.
- Notem uma grande variação ao longo do tempo.

# Fabricantes



- Um só processador.
- SMP - Symmetric Multi Processor.
- MPP - Massively Parallel Processor.
- Cluster - Um agregado ou uma rede de *workstations*.
- Constellation - “cluster of clusters”.

# Arquitetura



# Sistema Operacional



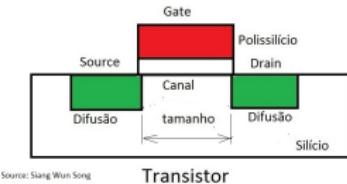
O avanço do hardware em termos de:

- Capacidade de processamento e armazenamento.
- Tamanho.
- Preço.

Esse avanço está relacionado ao avanço da tecnologia de microeletrônica ou VLSI (Very Large Scale of Integration).

# Avanço da Microeletrônica - pastilhas de silício

- Processador e memória são feitos de silício.
- O elemento básico dos circuitos digitais é o transistor MOS (Metal Oxide Semiconductor).
- Um transistor MOS é uma espécie de chave interruptora minúscula, de ordem de algumas dezenas de nanômetros quadrados de área.
- Presença de carga elétrica (voltagem alta) no Gate permite a condução de eletricidade entre os pontos Drain e Source, ao passo que a ausência de carga (voltagem baixa) no gate impede a condução.
- Veremos este assunto nas próximas aulas.



# Gordon Moore



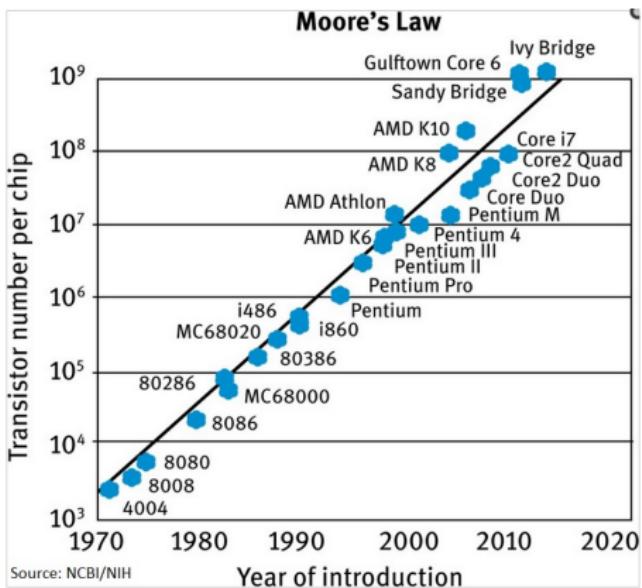
Source: Wikipedia

- Gordon Moore (1924 - )
- Engenheiro, co-fundador da Intel
- Previu o avanço da tecnologia VLSI (microeletrônica), observando o seu **crescimento exponencial**.

# Lei de Moore

"O número de transistores em uma pastilha dobra a cada 18 meses".

Não é bem uma lei. É mais uma observação ou projeção.



# Tamanho de um Transistor MOS

Tamanho (largura) de um transistor:

1963     $24 \mu\text{m}$

1978     $5 \mu\text{m}$

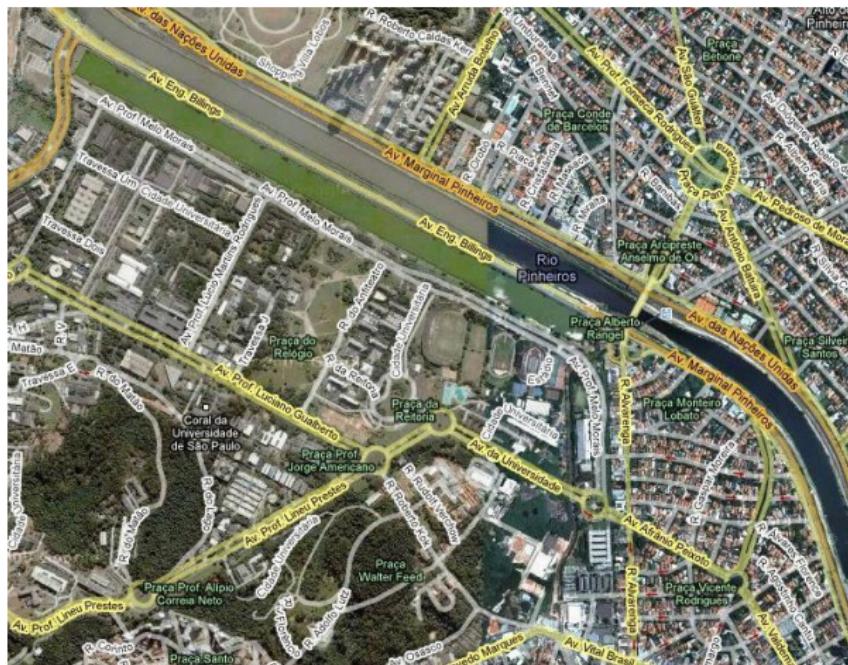
1990     $1 \mu\text{m}$

2005     $0,1 \mu\text{m}$

2017     $0,01 \mu\text{m}$

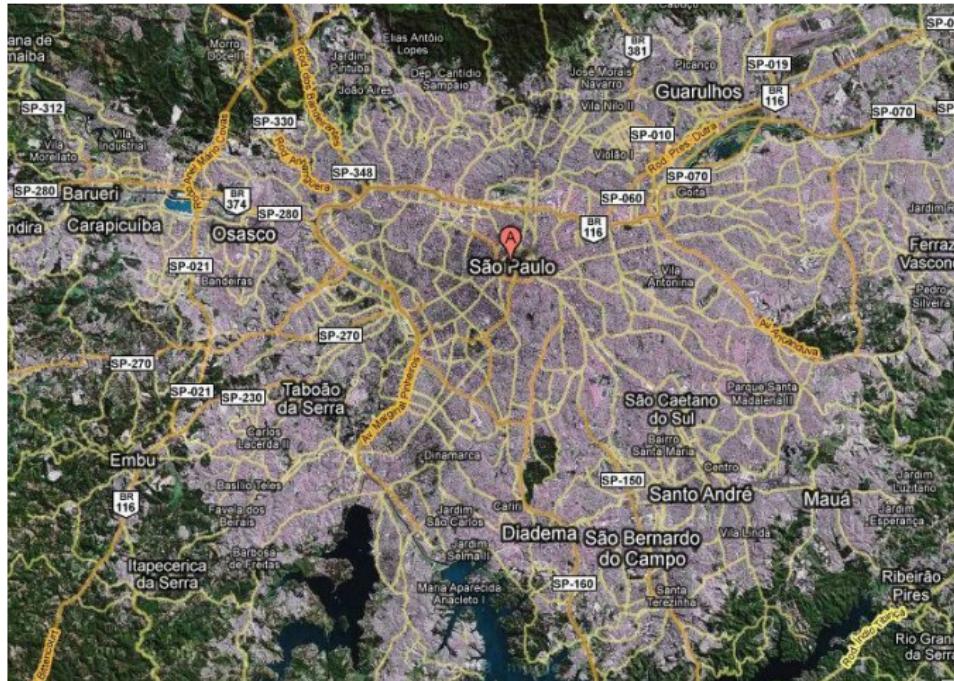
Ilustramos a seguir essa evolução, imaginando que um chip contém, ao invés de circuitos, ruas e praças de uma região geográfica.

# 1963 - tamanho 24 $\mu\text{m}$



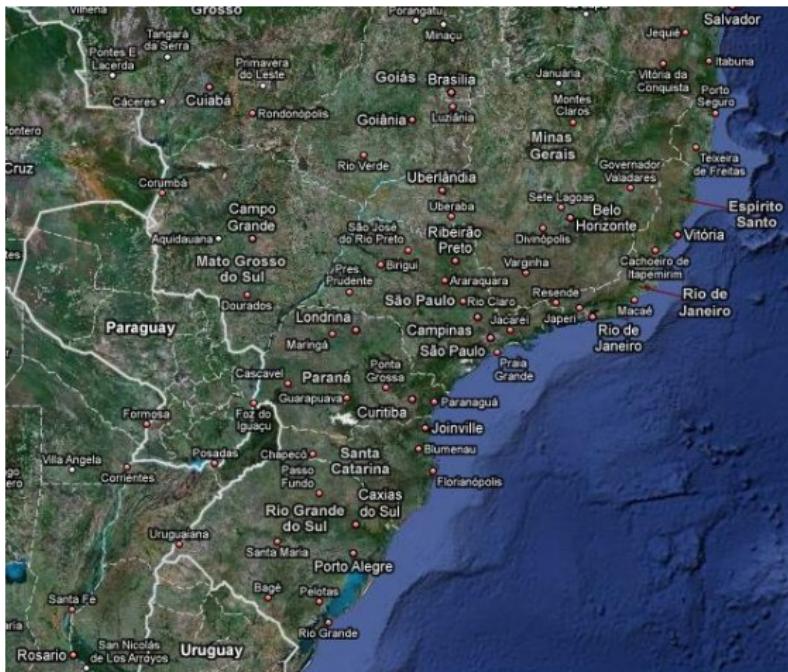
Source: Google Maps

# 1978 - tamanho 5 $\mu\text{m}$



Source: Google Maps

# 1990 - tamanho 1 $\mu\text{m}$



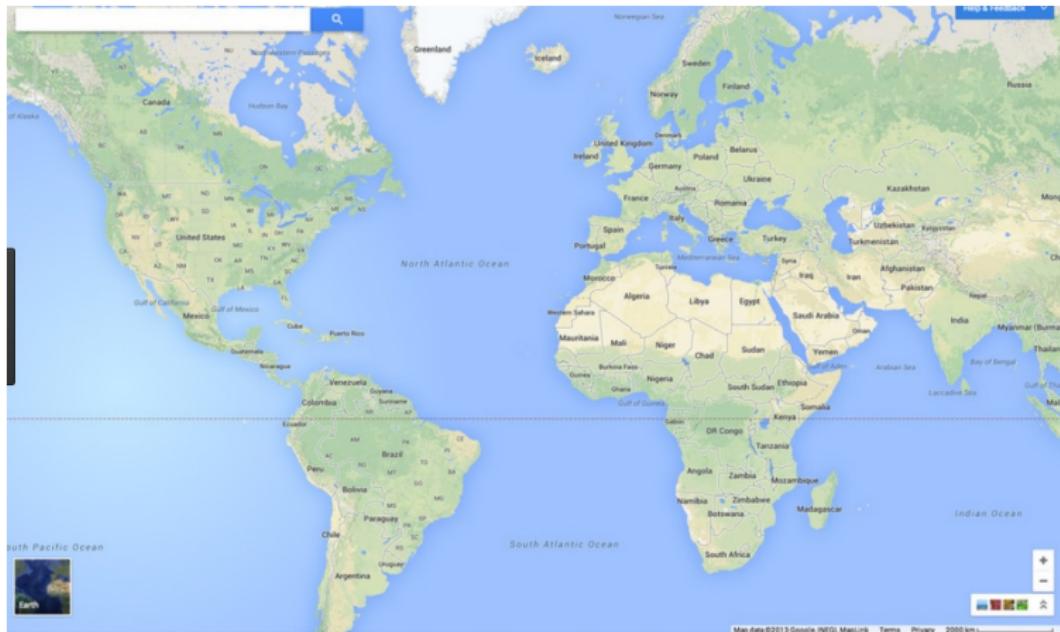
Source: Google Maps

# 2005 - tamanho 0,1 $\mu\text{m}$



Source: Google Maps

# 2017 - tamanho 0,01 $\mu\text{m}$



Source: Google Maps

- Intel Tukwila quad-core chip(2008): mais de 2 bilhões de transistores -  
Tecnologia de 65 nm ou 0,065 micrômetro.  
<http://news.bbc.co.uk/2/hi/technology/7223145.stm>
- Processador Intel Core i7 usa CMOS de 45 nm ou 0,045 micrômetro.  
<http://www.intel.com/products/processor/corei7/specifications.htm>  
<http://www.intel.com/technology/45nm/index.htm>
- Em 2016: Intel 22-core Xeon Broadwell-EP com 7,2 bilhões de transistores.
- Em 2019: AWS Graviton2 64-core ARM-based (7 nm technology) chip com 30 bilhões de transistores  
[Wikipedia: transistor count](#)

# Reflexões sobre a evolução da computação

- A computação paralela veio para ficar. Por que?
  - Um modo de aumentar a velocidade de um processador é aumentar a sua frequência do relógio, diminuindo o ciclo. Devido a problemas como dissipação de calor, a frequência não está aumentando de forma significativa ao longo do tempo.
  - Daí a popularidade cada vez maior da computação paralela: colocando-se mais *cores* numa pastilha (Lei de Moore).
- O fantástico avanço da área tem a ver com a tecnologia VLSI, conforme a projeção da “Lei” de Moore. Essa “lei” deixará de valer em breve. Que nova tecnologia surgirá?

# Como foi o meu aprendizado?

Responda se a afirmação é verdadeira ou falsa:

- 1 Pela lista TOP500, vivemos hoje na era de PetaFLOPS.
- 2 Todos os computadores da lista TOP500 hoje possuem mais do que um processador.
- 3 A Lei de Moore, por ser lei, vale sempre, no presente e no futuro.
- 4 O Brasil ainda não conseguiu colocar nenhum computador na lista TOP500.
- 5 Pela Lei de Moore, a frequência do relógio dobra em cada 18 meses.

# Lista de Exercícios 1

- Fazer e entregar por email a [Lista de Exercícios 1](#).
- Tem prazo para entregar. Procure fazer logo, com a matéria fresquinha na cabeça, e entregar dentro do prazo,

# Próximo assunto: Tecnologia VLSI

Source: Wikipedia



- Próximo assunto: Tecnologia VLSI (circuitos integrados).
- Veremos o que é um transistor MOS e como portas lógicas (AND, OR, etc) são fabricadas a partir de transistores MOS.
- Estamos ainda na Geração VLSI do Silício. Bom saber a razão do avanço fantástico que estamos vivenciando.
- Não percam!

# Tecnologia VLSI - Uma Breve Introdução

MAC0344 - Arquitetura de Computadores  
Prof. Siang Wun Song

Slides usados: <https://www.ime.usp.br/~song/mac344/slides03a-vlsi.pdf>

Baseado em parte no livro de Mead and Conway - Introduction to VLSI Systems



Source: Wikipedia

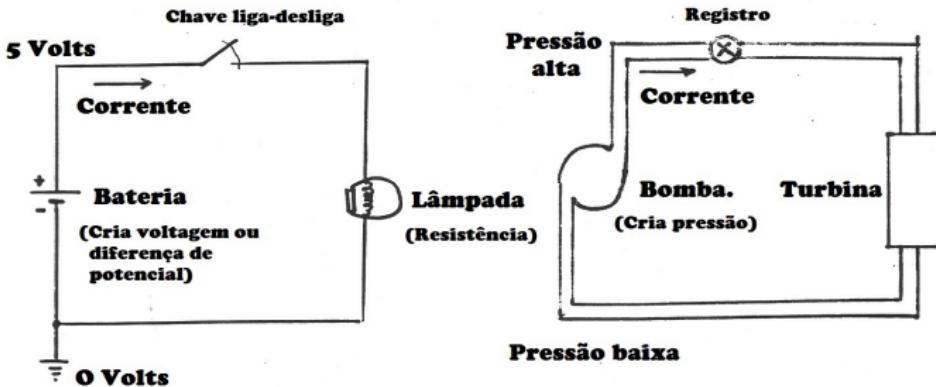
- Veremos a tecnologia VLSI. Será passada a Lista 2 de exercícios. Ao final dessas aulas, vocês saberão
  - O que é um transistor MOS e suas mil utilidades: funciona como chave, resistor, capacitor, ...
  - Portas lógicas NOT, NAND e NOR podem ser produzidas por vários transistores MOS. Transistor é portanto o tijolo para o mundo digital.
  - Em 40 anos, o tamanho do transistor diminuiu de 5 micrômetros para 7 nanômetros, aumentando a capacidade de uma pastilha de Silício de 510.000 vezes.
  - Estamos na Geração VLSI do Silício, razão do avanço fantástico que estamos vivenciando. (Mas essa geração está prestes a mudar, para uma nova ...)

Tecnologia de microeletrônica que integra uma grande quantidade de dispositivos eletrônicos (transistores) numa pastilha (chip) de silício.

Várias tecnologias surgiram, desde a sua invenção por Jack Kilby (Prêmio Nobel de Física), até chegar na atual tecnologia VLSI.

- **SSI** (Small Scale of Integration)
- **MSI** (Medium Scale of Integrations):  
Integram de dezenas ou centenas a milhares de transistores.
- **LSI** (Large Scale of Integration)
- **VLSI** (Very Large Scale of Integrations):  
Integram bilhões de transistores.

# Analogia - Recordação de circuitos elétricos



carga elétrica

corrente elétrica

voltagem

bateria

resistor

capacitor

gota de água

corrente de água

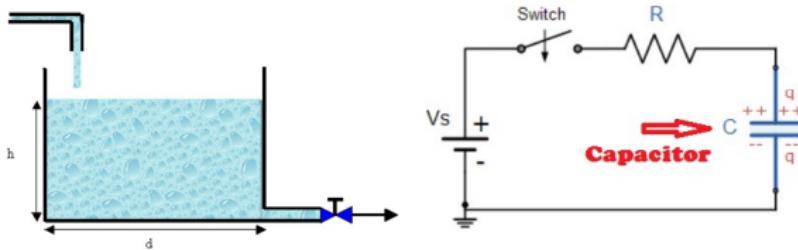
pressão

bomba

turbina

tanque de água

# Analogia - Recordação de circuitos elétricos



- Para armazenar água podemos encher um tanque.
- Para armazenar carga elétrica usamos um **capacitor**.
- O tempo para carregar totalmente um tanque de água (ou um capacitor de carga elétrica) depende, entre outros parâmetros, da dimensão do tanque (ou do capacitor).
- Em VLSI, um capacitor minúsculo é usado para representar 1 se está carregado de carga, e 0 caso contrário.

MOS = Metal Oxide Semiconductor

- Veremos o transistor MOS, que pode funcionar como uma chavinha minúscula (liga e desliga) feito de semicondutor (**Silício Si**).
- O transistor MOS é importante pois é o tijolo que constrói todo o mundo digital.

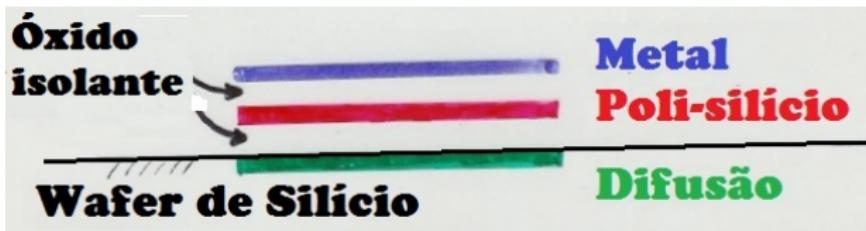
As explicações são simplificadas (usando a tecnologia NMOS) para facilitar o entendimento. CMOS, parecida com NMOS, é a tecnologia mais usada.

# Transistor MOS

Em cima de um substrato de Silício (*wafer de Silício*) são depositadas 3 camadas de material condutor:

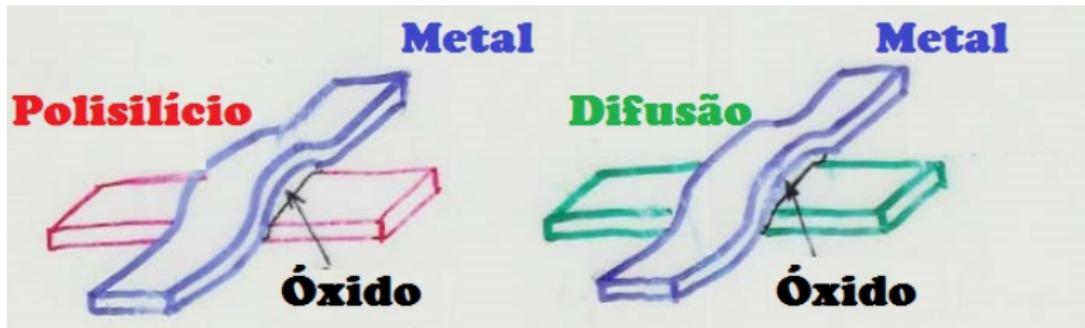
- metal
- polisilício
- difusão

As 3 camadas são separadas por óxido (isolante).



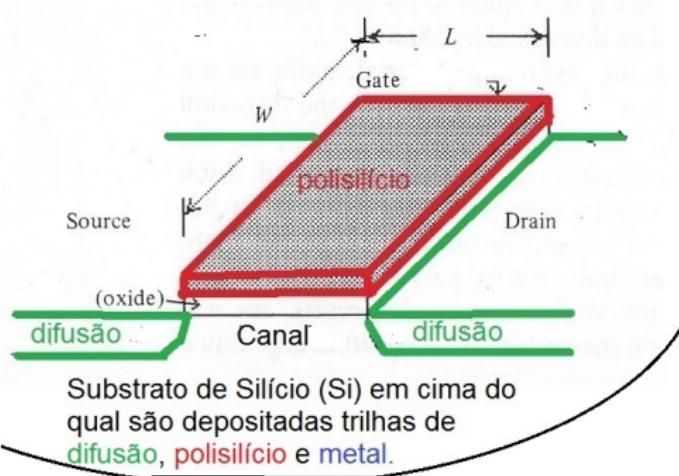
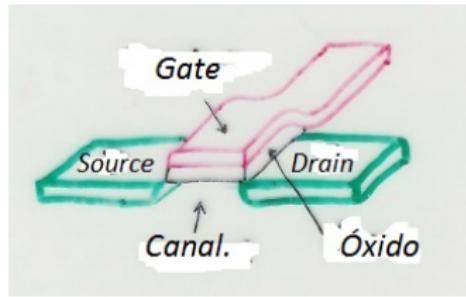
# Sobreposição de camadas

- Uma trilha de metal pode cruzar uma trilha de polisilício ou de difusão sem produzir efeito significativo.



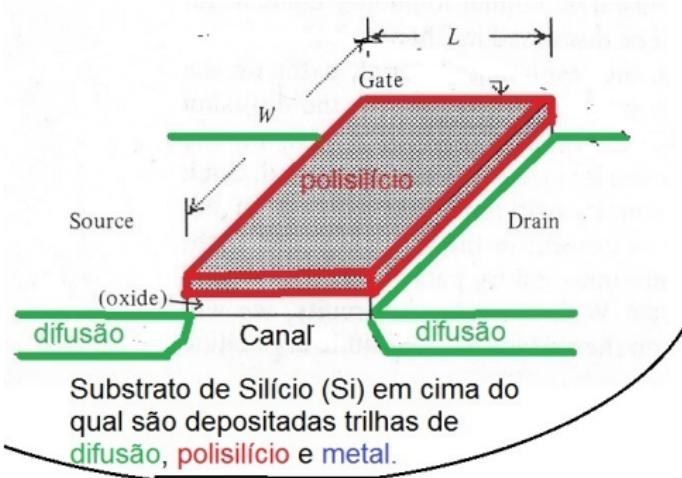
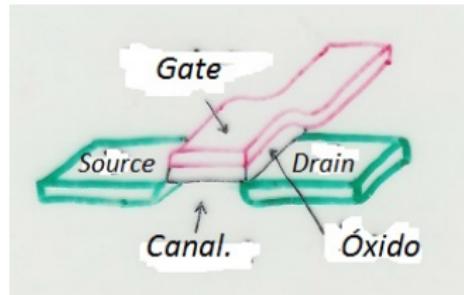
Vocês vão perguntar: E se uma trilha de **polisilício** cruzar uma de **difusão**?

# Polisilício cruzando difusão produz um transistor



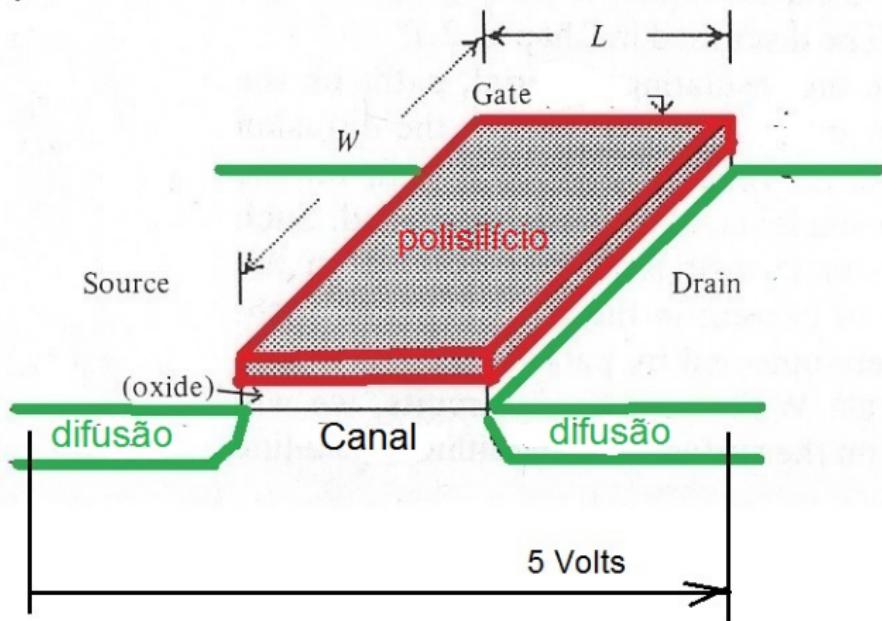
- Se uma trilha de **polisilício** cruzar uma trilha de **difusão**, então aparece um **transistor MOS** (que funciona como uma chave liga-desliga). Vejamos como.

# Polisilício cruzando difusão produz um transistor



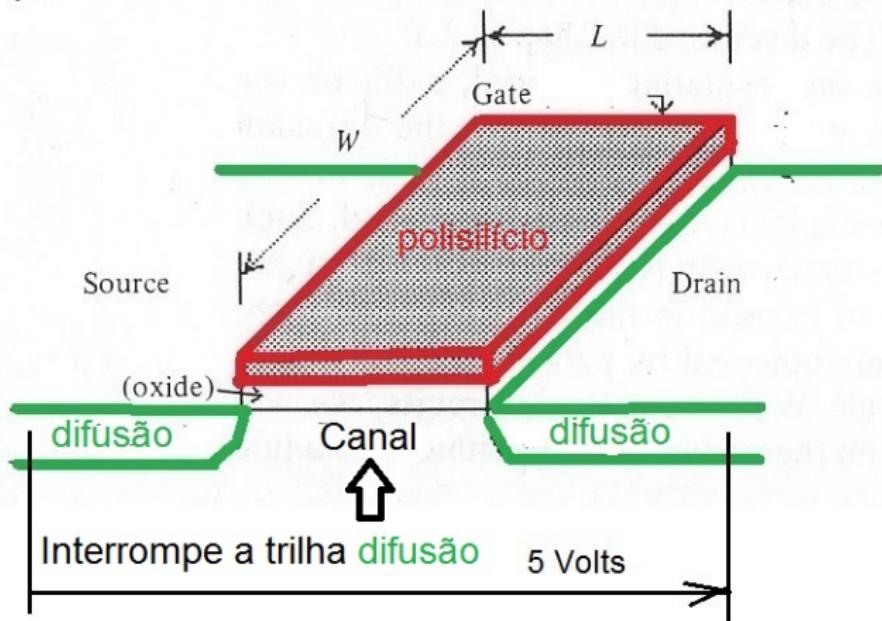
- Observemos primeiro o seguinte (veja a figura): A trilha de **difusão** não é contínua mas está interrompida no Canal. Isso foi por construção na fabricação.

# Polisilício cruzando difusão produz um transistor



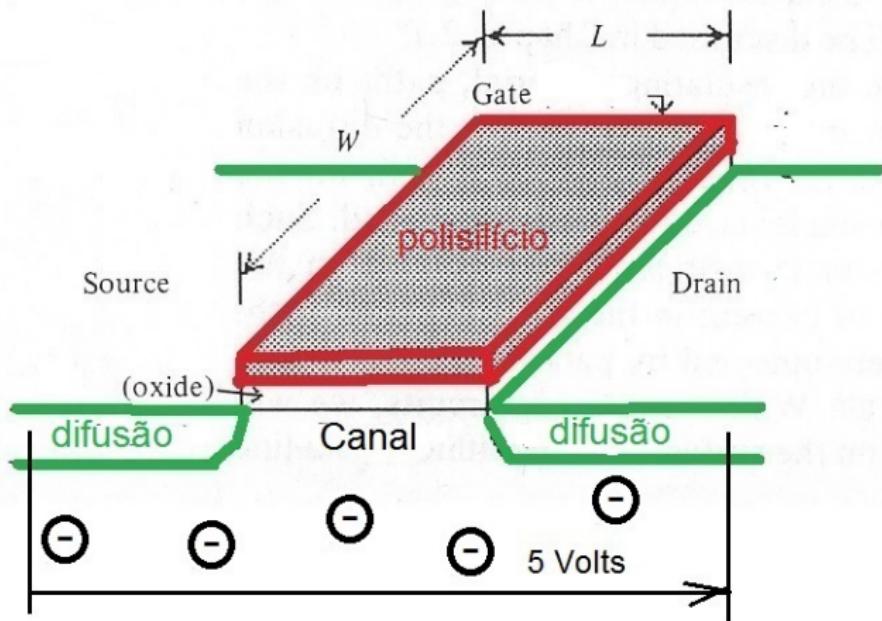
- Vamos aplicar uma voltagem, e.g. de 5 Volts, entre *Drain* e *Source* na trilha de **difusão**  $V_{DS} = 5$  Volts. Será que isso vai permitir passagem de corrente na trilha de **difusão**?

# Polisilício cruzando difusão produz um transistor



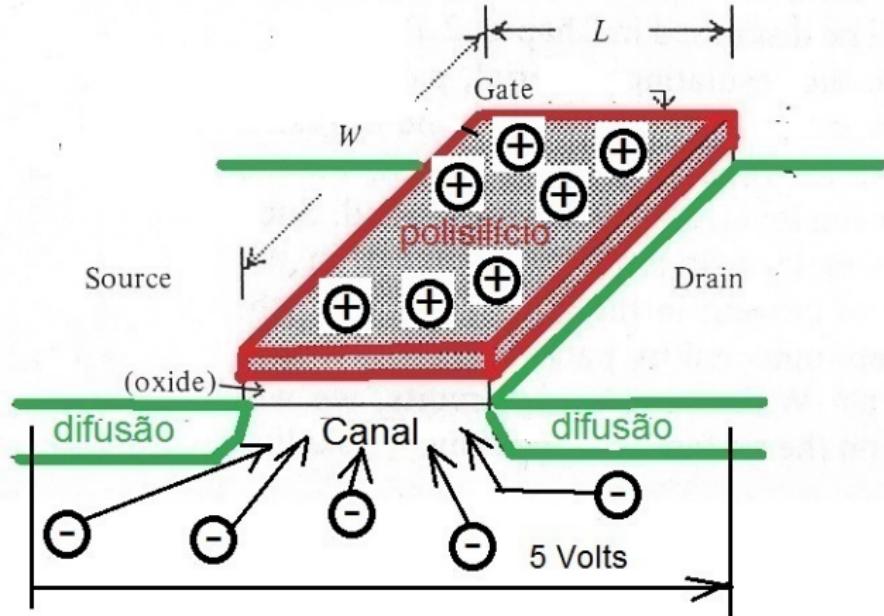
- A trilha de **difusão** está interrompida no Canal (por construção na fabricação). Por isso não passa corrente nessa trilha mesmo aplicando a voltagem VDD de 5 Volts.

# Polisilício cruzando difusão produz um transistor



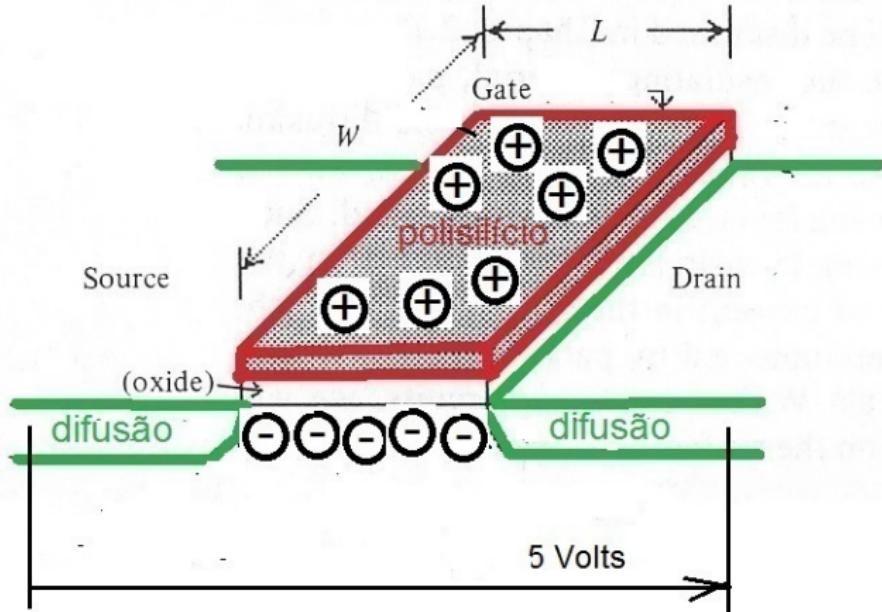
- No substrato de Silício há elétrons livres. Vejamos como podemos concentrar esses elétrons livres no Canal para permitir passagem de corrente elétrica.

# Polisilício cruzando difusão produz um transistor



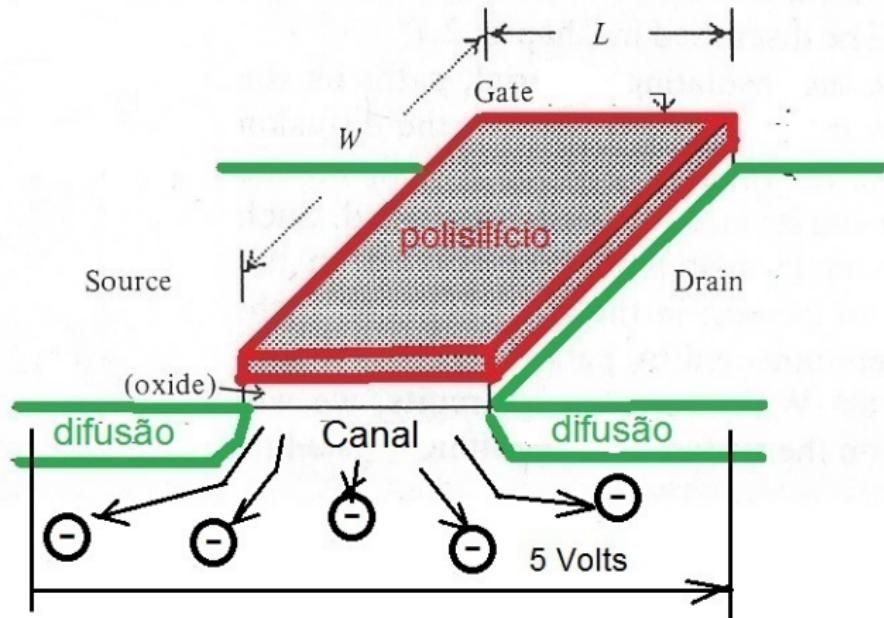
- Injetando uma corrente elétrica no Gate aumenta a voltagem entre Gate e Source  $V_{GS}$ . Cargas positivas no Gate atraem elétrons (cargas negativas) para o Canal.

# Polisilício cruzando difusão produz um transistor



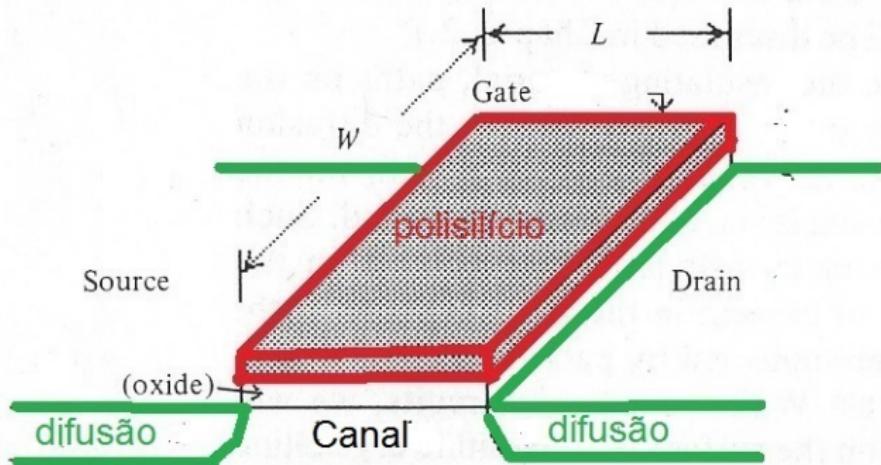
- Elétrons concentrados no Canal acabam interligando as duas partes da **difusão** (que estavam interrompidas) e permitem a passagem de corrente na trilha.

# Polisilício cruzando difusão produz um transistor



- Tirando as cargas no gate (por exemplo ligando o Gate à terra), os elétrons concentrados no Canal se despersam e cessa a corrente na trilha da **difusão**.

# Polisilício cruzando difusão produz um transistor

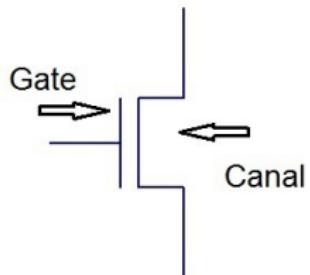
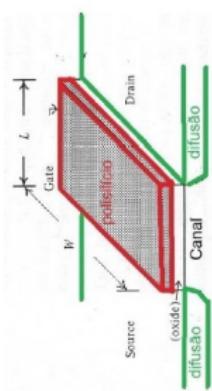


Transistor MOS é uma chave liga-desliga

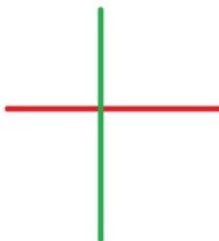
- Voltagem alta no Gate **polisilício**: passa corrente na trilha **difusão** Voltagem zero no Gate **polisilício**: cessa corrente na trilha **difusão**.

# Notação para transistor MOS

Ao invés de desenhar um transistor MOS como na figura à esquerda,



Notação **sem cor**



Notação **colorida (palito)**

podemos usar essas 2 notações simplificadas. Notem a semelhança com o desenho.

Vimos que o transistor MOS pode funcionar como uma **chave liga-desliga**.

E daí?

Acontece que o transistor MOS serve como tijolo para o mundo digital.

Veremos agora a sua importância.

# Importância do transistor MOS

Veremos que o transistor MOS, além de funcionar como chave liga-desliga, também tem as seguintes utilidades:

- Pode funcionar como **capacitor** (para armazenar carga elétrica).



Ao contrário do capacitor da figura, o transistor MOS é minúsculo (alguns nanômetros).

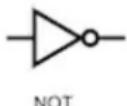
A memória do computador usa um transistor carregado com carga para representar 1 e descarregado para 0.

- Pode funcionar como **resistor** (resistência).

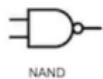


# Importância do transistor MOS

- Com dois transistores podemos construir uma **porta NOT** (para inverter um sinal lógico).



- Com três transistores podemos construir
  - uma **porta NAND** (porta AND seguida de NOT).

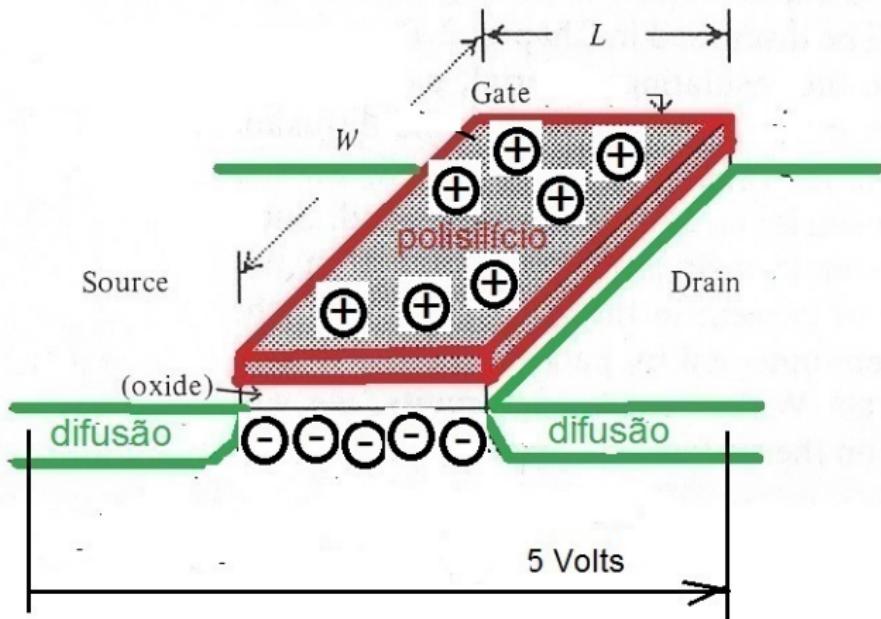


- uma **porta NOR** (porta OR seguida de NOT).



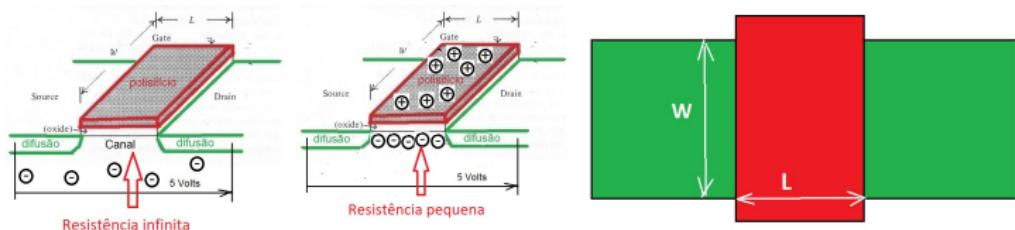
Com transistores MOS como tijolo construímos qualquer circuito digital, incluindo memórias e processadores.

# Transistor MOS como capacitor



- Voltagem alta no Gate carrega cargas elétricas no capacitor. Voltagem zero no Gate descarrega as cargas do capacitor. Um transistor pode então implementar um bit de memória.

# Transistor MOS como resistor ou resistência



Um transistor que não conduz corrente apresenta uma resistência 'infinita' pois a trilha difusão está interrompida no Canal.

Mas um transistor conduzindo ou passando corrente possui uma pequena resistência  $R$  cujo valor é diretamente proporcional ao comprimento  $L$  e inversamente proporcional à largura  $W$ .

$$R = \alpha \frac{L}{W}, \text{ onde } \alpha \text{ é uma constante.}$$

- O comprimento  $L$  e a largura  $W$  são medidas na região de **interseção entre Polissilício e Difusão** (ver figura).
- $L$  é a medida na direção do fluxo da corrente
- $W$  é a medida ortogonal ao comprimento.

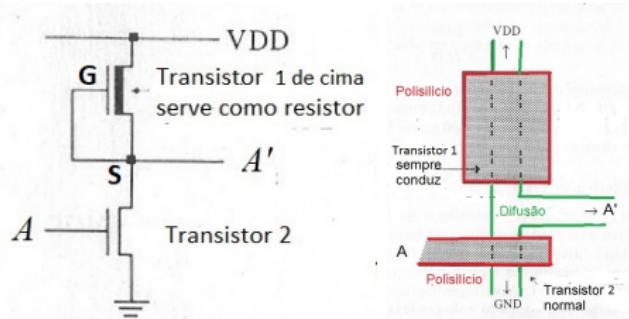
# Para entender como produzir uma porta NOT

Recordamos o divisor-de-tensão (potenciômetro).



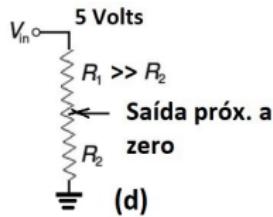
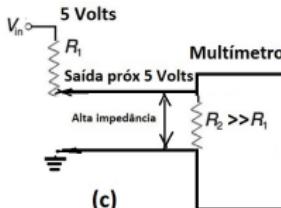
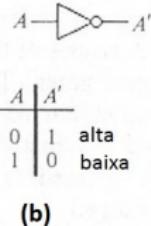
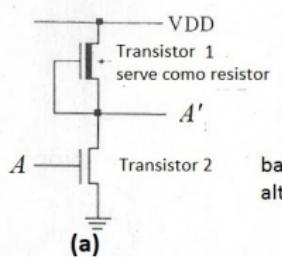
- O *dimmer* é usado para controlar a luminosidade de uma lâmpada. O botão deslizante pode ser movido para baixo para diminuir a intensidade da luz.
- O *dimmer* pode ser implementado com um potenciômetro através do divisor-de-tensão.
- $V_{out} = \frac{R_2}{R_1+R_2} \cdot V_{in}$
- Se  $R_1 = R_2$  então  $V_{out} = \frac{1}{2} \cdot V_{in} = 2.5$  Volts
- Se  $R_1 \gg R_2$  então  $V_{out}$  fica próximo a zero Volts.

# Dois transistores produzem uma porta NOT



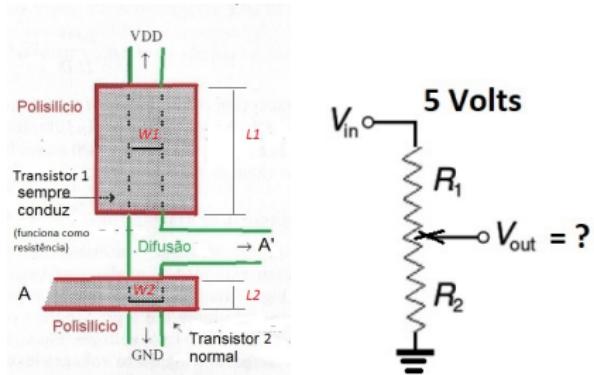
- O transistor 1 foi fabricado para sempre permitir a passagem de corrente. O seu papel é funcionar como resistência.
- Isso é feito através de uma implantação de íons no canal do transistor 1. Com isso, o transistor 1 já conduz, ao fazer voltagem entre gate e source  $V_{GS} = 0$ .
- O transistor 2 de baixo funciona com uma chave liga-desliga.

# Dois transistores produzem uma porta NOT



- Se  $A$  tem voltagem baixa (0), o transistor 2 não conduz corrente e o circuito está interrompido. Se você mede a saída com um multímetro (que possui alta impedância ou resistência), a saída  $A'$  fica com voltagem alta (1). Ver Figura (c).
- Se  $A$  tem voltagem alta (1), o transistor 2 conduz corrente e apresenta uma resistência que chamamos de  $R_2$ .
- Se a resistência do transistor 1  $R_1$  for muito maior que  $R_2$ , então teremos a saída  $A'$  com voltagem baixa (0). Ver a Figura (d).

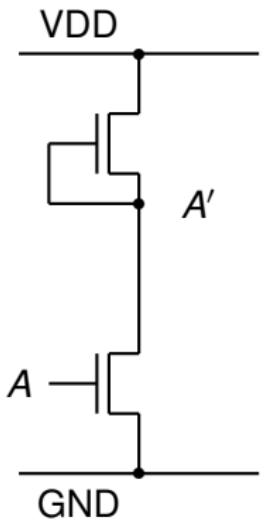
# Dois transistores produzem uma porta NOT



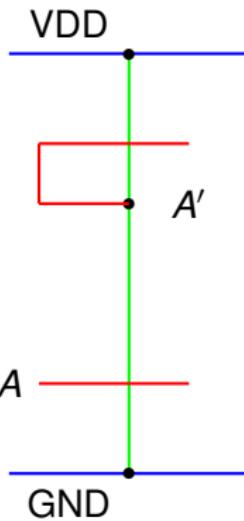
Para uma porta NOT funcionar, basta fazer a resistência de condução do transistor de cima  $R_1$  ser 4 vezes a resistência de condução do transistor de baixo  $R_2$ :

$$R_1 = 4R_2$$
$$\frac{L_1}{W_1} = 4 \frac{L_2}{W_2}$$

# Notação para porta NOT



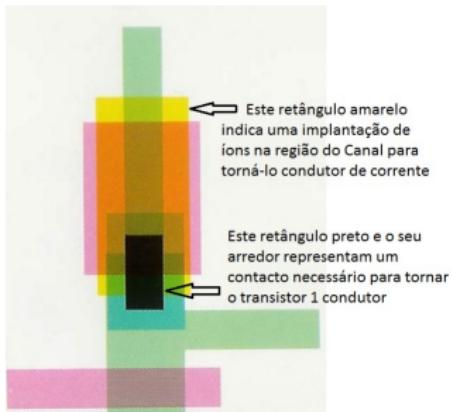
Notação sem cor



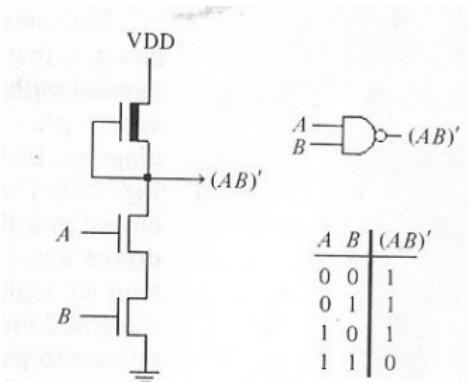
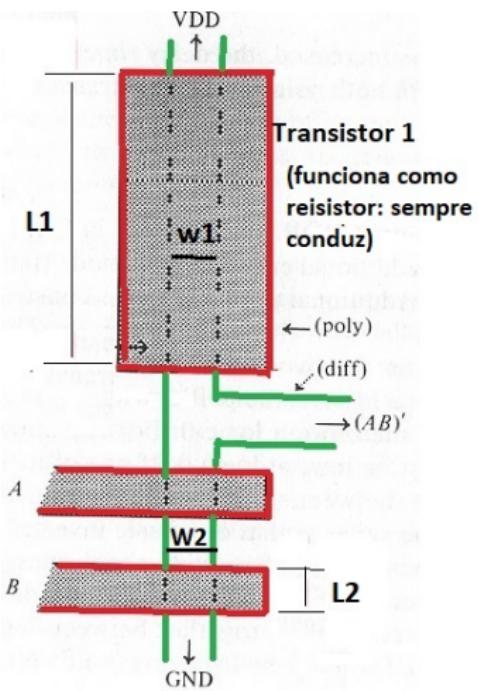
Notação colorida (palito)

# Como está o meu aprendizado?

- A figura mostra uma porta NOT em NMOS.
- A figura mostra alguns detalhes do transistor 1 para que o seu Canal seja diferente e sempre conduza corrente. (**Não se preocupe nos esses detalhes**, que apenas servem para mostrar que ele é diferente.)
- Identificar na figura abaixo as dimensões  $L_1$ ,  $W_1$ ,  $L_2$ , e  $W_2$ .

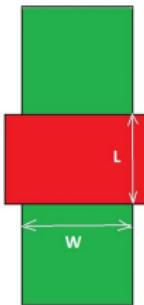


# Três transistores produzem uma porta NAND

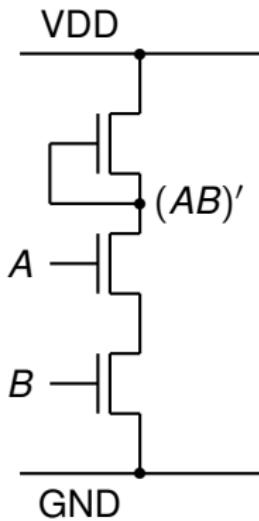


Precisamos fazer  $L_1/W_1=8$   $L_2/W_2$  pois as resistências de condução dos transistores A e B se somam.

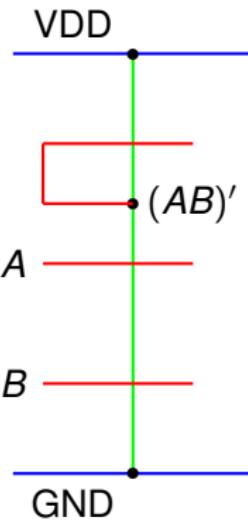
O efeito final é que a resistência do transistor 1 fica 4 vezes a resistência equivalente de A e B.



# Notação para porta NAND

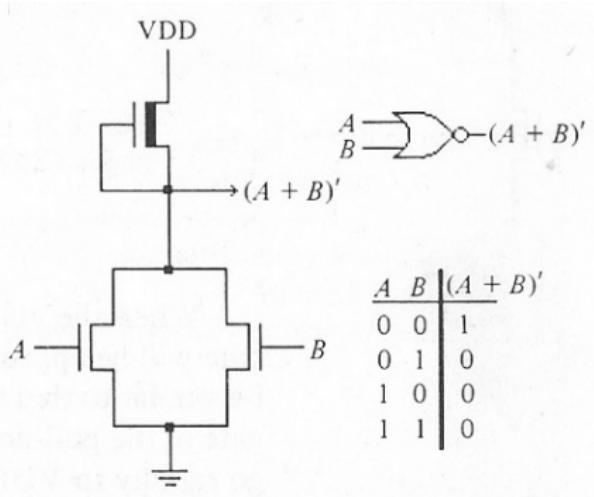
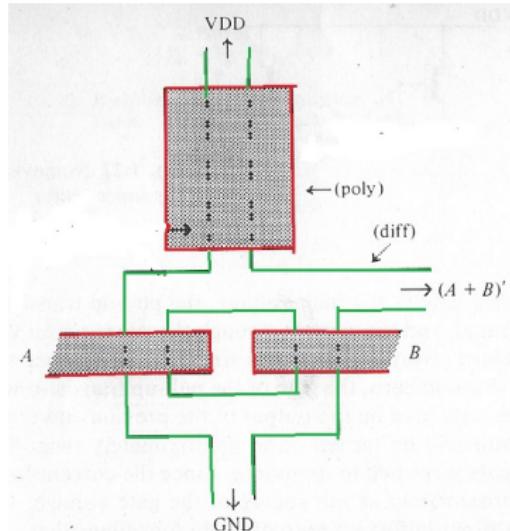


Notação sem cor



Notação colorida (palito)

# Três transistores produzem uma porta NOR



- Aqui basta fazer  $L_1/W_1 = 4L_2/W_2$  pois as resistências de condução de A e B estão em paralelo, produzindo uma resistência equivalente menor que cada uma delas (no caso igual a metade).
- Você pode desenhar a porta NOR com a notação de palito (com cor)?

# Lista de Exercícios 2

- Fazer e entregar por email a [Lista de Exercícios 2](#).
- Tem prazo para entrega. Recomendo não demorar muito.  
Bom fazer logo com a matéria fresquinha na cabeça.

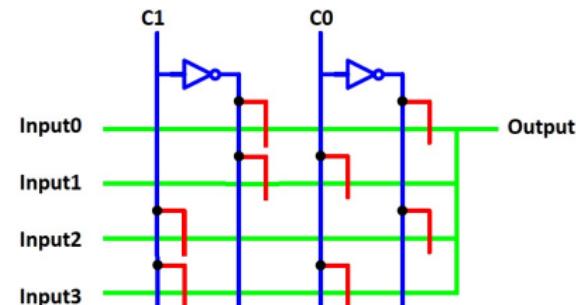
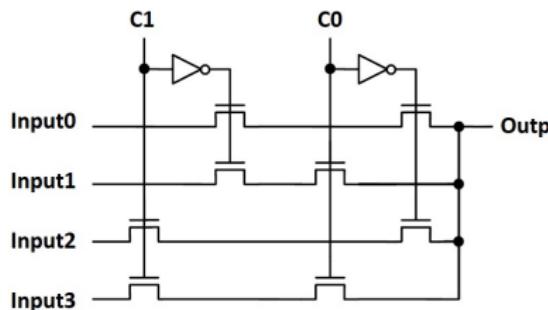
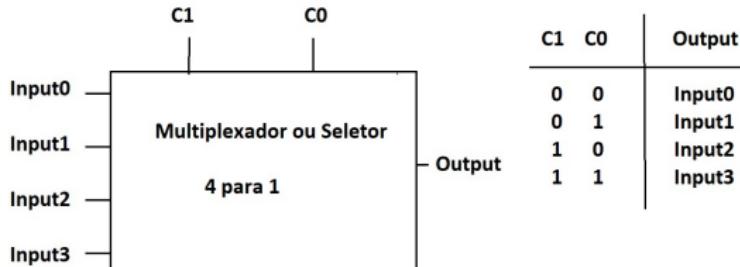
# Circuitos usando transistores de passagem



- O transistor MOS funcionando como uma chave deixa passar ou não corrente elétrica.
- O transistor com essa finalidade é conhecido como transistor de passagem (*pass transistor*).
- Vamos dar 2 circuitos interessantes que usam apenas a porta NOT e transistores de passagem.

# Multiplexador ou Seletor 4 para 1

Multiplexador ou seletor 4 para 1 usando apenas transistores de passagem e portas NOT.

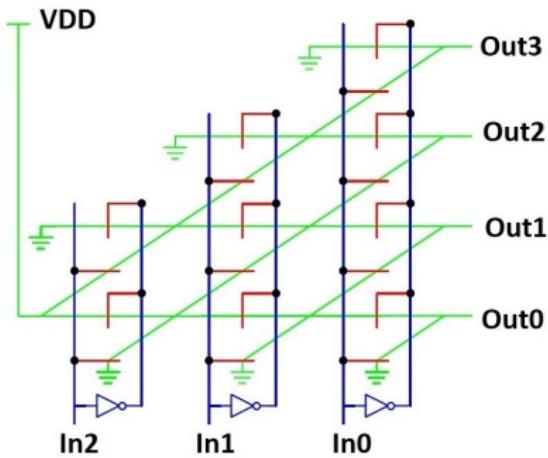


Source: VLSI Systems Univ. Cambridge 2004 (based on Mead and Conway - Intro. to VLSI Systems)

# Círcuito contador de 1's

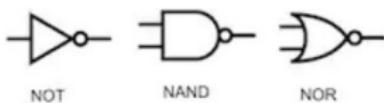
Dados 3 bits  $IN_0, IN_1, IN_2$ , o circuito conta o número de 1's. Se  $i$  bits iguais a 1, a saída  $Out_i = 1$  e as demais saídas = 0.

$IN_2$	$IN_1$	$IN_0$	$Out0$	$Out1$	$Out2$	$Out3$
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	1	0	0
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	0	1	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



Source: VLSI Systems Univ. Cambridge 2004 (based on Mead and Conway - Intro. to VLSI Systems)

# Lógica booleana com portas NAND ou NOR



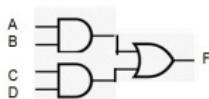
Veremos agora que qualquer circuito digital pode ser implementado

- apenas com portas NAND e NOT (NOT pode ser considerado como uma porta NAND com apenas uma entrada)
- ou apenas com portas NOR e NOT (NOT pode ser considerado como uma porta NOR com apenas uma entrada).

# Lógica booleana usando porta NAND

Considere a equação lógica expressa na forma normal disjuntiva ou disjunção de cláusulas conjuntivas (uma soma de produtos).

$$F = (A \wedge B) \vee (C \wedge D)$$

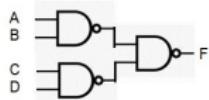


Vamos negar duas vezes o lado direito (que não altera nada):

$$F = \overline{\overline{(A \wedge B)} \vee \overline{(C \wedge D)}}$$

Aplicamos Lei de Morgan:

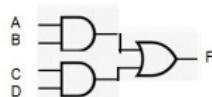
$$F = \overline{(A \wedge B)} \wedge \overline{(C \wedge D)}$$



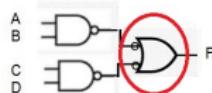
# Lógica booleana usando porta NAND

Vamos “demonstrar” a mesma coisa usando “desenhos”. Seja a equação lógica:

$$F = (A \wedge B) \vee (C \wedge D)$$

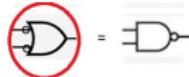


Negando a saída do AND e a entrada do OR não muda nada. Assim:

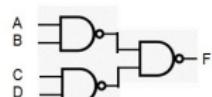


Vamos expressar a Lei de Morgan em forma de desenho:

$$\overline{X} \vee \overline{Y} = \overline{X \wedge Y}$$



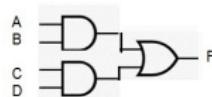
Portanto temos:



# Lógica booleana usando porta NOR

Considere de novo a equação lógica:

$$F = (A \wedge B) \vee (C \wedge D)$$

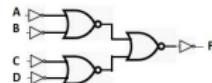


Vamos negar duas vezes o lado direito (que não altera nada):

$$F = \overline{\overline{(A \wedge B)} \vee \overline{(C \wedge D)}}$$

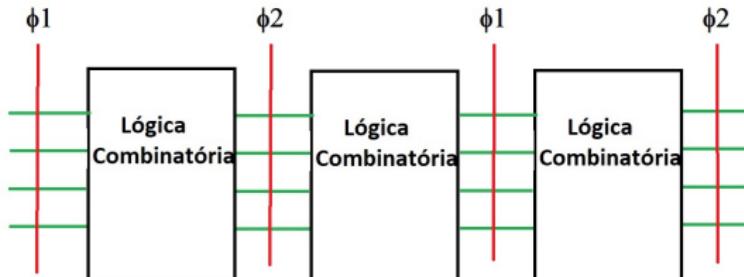
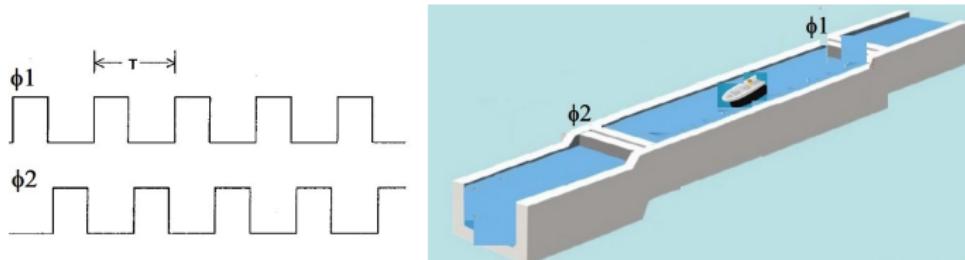
Aplicamos Lei de Morgan:

$$\begin{aligned} F &= \overline{\overline{(A \wedge B)} \wedge \overline{(C \wedge D)}} = \overline{(\overline{A} \vee \overline{B}) \wedge (\overline{C} \vee \overline{D})} = \overline{(\overline{A} \vee \overline{B})} \vee \overline{(\overline{C} \vee \overline{D})} \\ \text{Portanto } \overline{F} &= (\overline{A} \vee \overline{B}) \vee (\overline{C} \vee \overline{D}) \end{aligned}$$

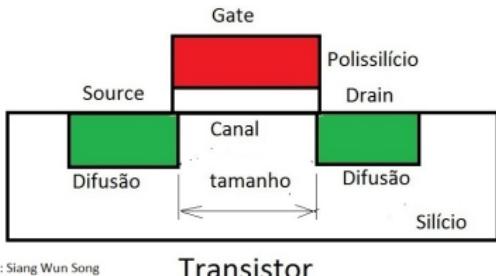


# Relógio de duas fases $\phi_1$ e $\phi_2$

Um relógio de duas fases  $\phi_1$  e  $\phi_2$  é usado para controlar o movimento dos dados num circuito MOS. (Semelhante ao funcionamento de uma eclusa e.g. Canal do Panamá.)



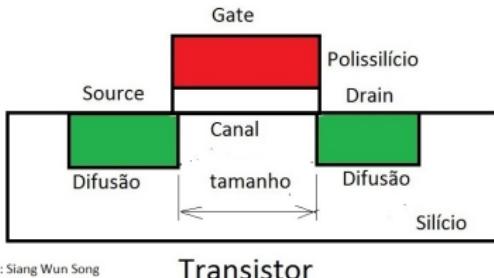
# Importância do transistor MOS



Ano	Tamanho transistor
1978	5 micrômetros
2018	7 nanômetros

- O tamanho (*feature size*) de um transistor é o comprimento do Canal entre *Drain* e *Source*.
- Esse tamanho passou de 5 micrômetros em 1978 para 7 nanômetros em 2018.

# Importância do transistor MOS

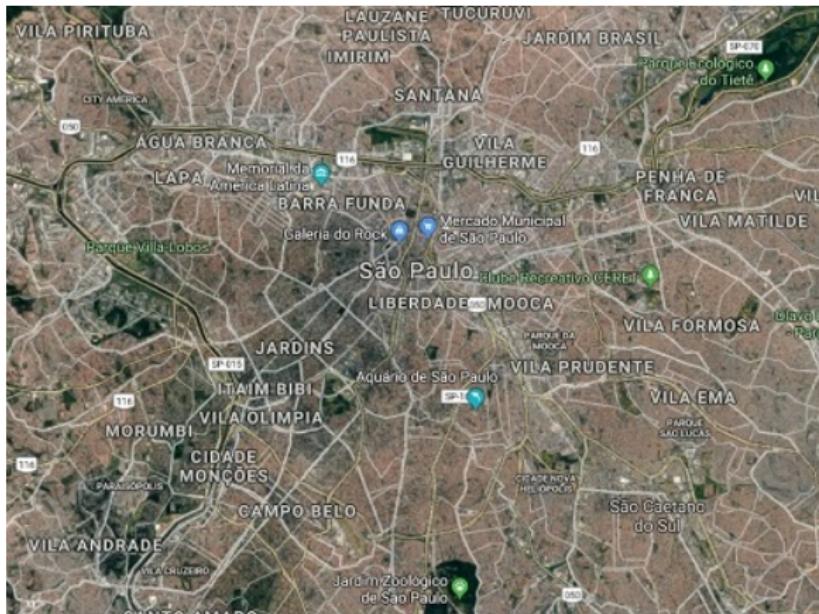


Ano	Tamanho transistor	Redução (1D)	Redução na área (2D)
1978	5 micrômetros		
2018	7 nanômetros	714 vezes	510.000 vezes

- Se uma dimensão (1D) na pastilha diminui  $5000/7 = 714$  vezes, temos uma redução de  $714^2 = 510.000$  na área (2D).
- “Lei” de Moore: o número de transistores numa pastilha de Silício dobra a cada 18 meses.

# Importância do transistor MOS

Suponha que a pastilha de 1978 continha, ao invés de circuitos VLSI, uma região geográfica de 1.000 km<sup>2</sup>.



Source: Google Maps

# Importância do transistor MOS

A mesma pastilha hoje pode conter toda a terra, com uma área de 510.000.000 km<sup>2</sup>. Isso ilustra o avanço da VLSI.



Source: Google Maps

# Importância do transistor MOS

Hoje existem 3 fabricantes no mundo capazes de produzir chips com a tecnologia de 7 nm. ([Clicar aqui para a reportagem completa.](#))

- Taiwan Semiconductor Manufacturing Company (TSMC)

Para um vídeo sobre esse fabricante, ver:

[Inside The Worlds Largest Semiconductor Factory - BBC \(4:17 minutos\)](#)

- Samsung
- Intel

A previsão é que em 2024 será possível produzir chips com a tecnologia de 5 nm. Na analogia usada, um tal chip poderá conter:

# Importância do transistor MOS



Source: Google Maps

*Ao passar da tecnologia de 7 nm (2018) para 5 nm (2024), dobrou-se a capacidade da pastilha. Note-se isso levou 6 anos, ao invés de 18 meses. Vemos que a tal “Lei” de Moore começou a falhar.*

# Como foi o meu aprendizado?

Um pequeno desafio: gostou do multiplexador 4 para 1? Você seria capaz de projetar um de 8 para 1? (Não precisa entregar...)

“Lei” de Moore:

*o número de transistores numa pastilha de Silício dobrava a cada 18 meses.*

- Não é uma lei. Foi mais uma constatação de Moore que vem se verificando ao longo dos anos.
- Mas o tamanho do transistor não pode diminuir sempre, por causa de limitações físicas.
- Para os curiosos: verifique consultando na Internet se essa ‘Lei’ está no seu fim.
- Para pensar: Se a tecnologia baseada no Silício está no fim, quais novas tecnologias vão surgir no futuro?

# Próximo assunto: Processo de fabricação e arrays sistólicos



Source: Wikipedia

- Próximo assunto: Processo de fabricação e arrays sistólicos
- Como fabricar chip VLSI: sala limpa, processo básico semelhante a revelação de fotos a partir de negativos.
- Pastilhas feitas sob medida para aplicações específicas (ASICs).
- Arrays sistólicos e Google TPU (*Tensor Processing Unit*)

# Processo de fabricação VLSI e breve introdução a arrays sistólicos TPU

MAC0344 - Arquitetura de Computadores  
Prof. Siang Wun Song

Slides usados: <https://www.ime.usp.br/~song/mac412/vlsi-fab.pdf>

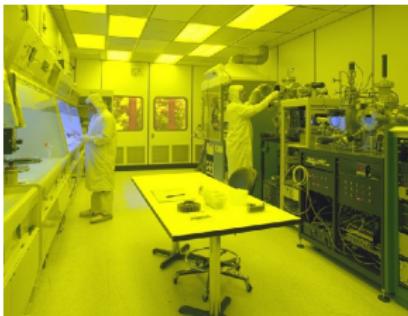
Baseado em parte em Mead and Conway - Introduction to VLSI Systems  
Esse assunto não cai em provas

# Fabricação de chips VLSI e Arrays Sistólicos

- Fabricação de chips VLSI e conceito de arrays sistólicos
- Ao final desta aula vocês saberão
  - O processo básico para fabricação VLSI
  - Pastilhas VLSI podem ser fabricadas para aplicações específicas (ASICs), por exemplo usando FPGAs (Field Programmable Gate Arrays) ou arrays sistólicos.
  - Um exemplo de um array sistólico para multiplicar duas matrizes.
  - Google TPU é um array sistólico usado em Google Search, Google Street View, Google translate para computações de redes neurais em aprendizado de máquina.

Esse assunto não cai em provas.

# Fabricação de pastilhas VLSI



Source: Wikipedia

- Instalações de alto custo (TSMC Taiwan investiu 9 bilhões de dólares e planeja uma fábrica de 20 bilhões)
- Ambiente urbano: 35 milhões de partículas de  $0,5 \mu\text{m}$  por  $\text{m}^3$ .
- Sala limpa ISO 1:  $\leq 12$  partículas de  $0,3 \mu\text{m}$  por  $\text{m}^3$ . Mais exigente do que uma sala cirúrgica.
- Controle de temperatura e umidade.
- Controle contra vibração - equipamentos ou uma sala inteira colocada em cima de isolador de vibração.

Fonte: Wikipedia - Semiconductor fabrication plant.

# Processo básico



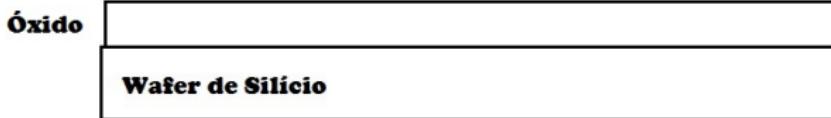
Source: PublicDomainPictures.net

- Veremos o processo básico.
- O processo básico usa uma máscara que possui partes transparentes e partes opacas.
- O processo básico produz a forma da máscara na pastilha de silício.
- A máscara é parecida com o negativo de fotografia e serve para levar a imagem do negativo para um papel fotográfico.

## **Wafer de Silício**

- Expor wafer de silício a oxigênio num forno de alta temperatura.

# Processo Básico



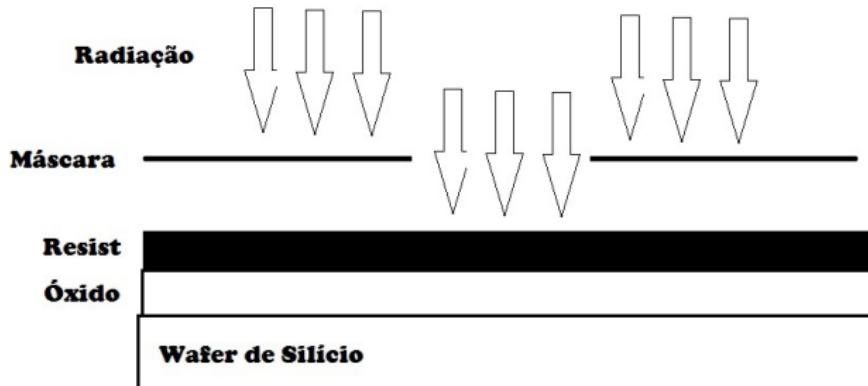
- Expor wafer de silício a oxigênio num forno de alta temperatura. Forma-se óxido  $SiO_2$  na superfície.

# Processo Básico



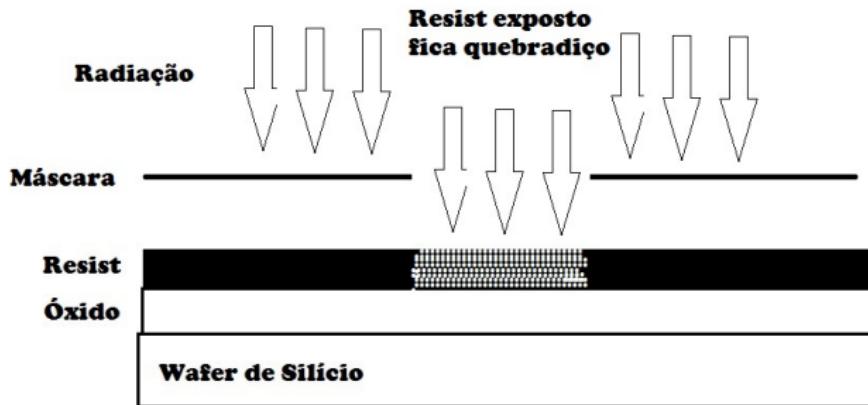
- Pintar com uma fina camada de material orgânico chamado “resist”. Secar e “assar” no forno.

# Processo Básico



- Incidir radiação intensa de luz ultravioleta ou raio-X através de uma máscara.

# Processo Básico



- Isso vai quebrar a estrutura de moléculas de parte (expostas) do resist.

# Processo Básico



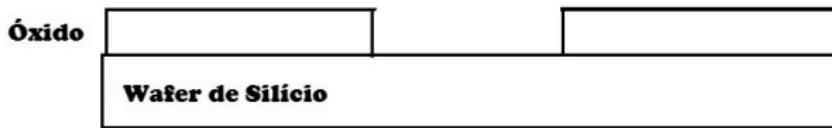
- Usar banho de solvente para tirar “resist” expostos (quebradiços).

# Processo Básico



- Usar ácido hidrofluórico que dissolve o óxido  $SiO_2$  mas não ataca o “resist”.

# Processo Básico

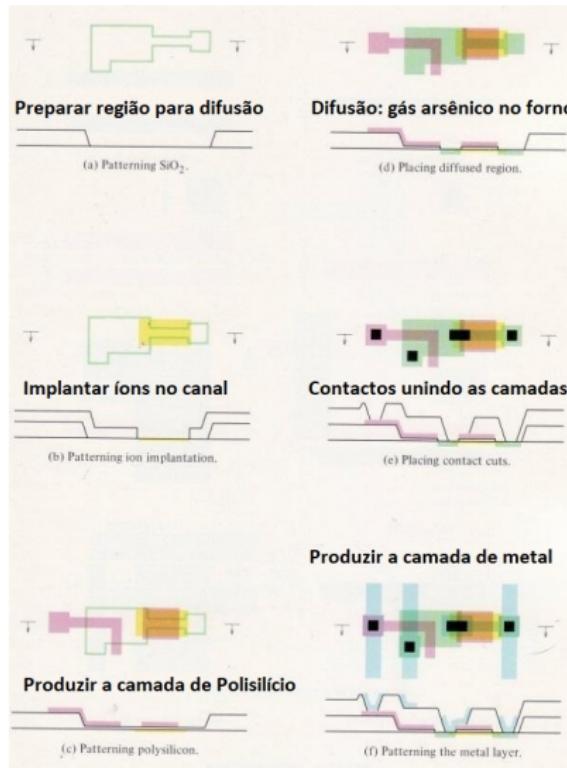


- Eliminar “resist” com solventes fortes ou ácidos. O processo básico produz a forma da máscara no chip.

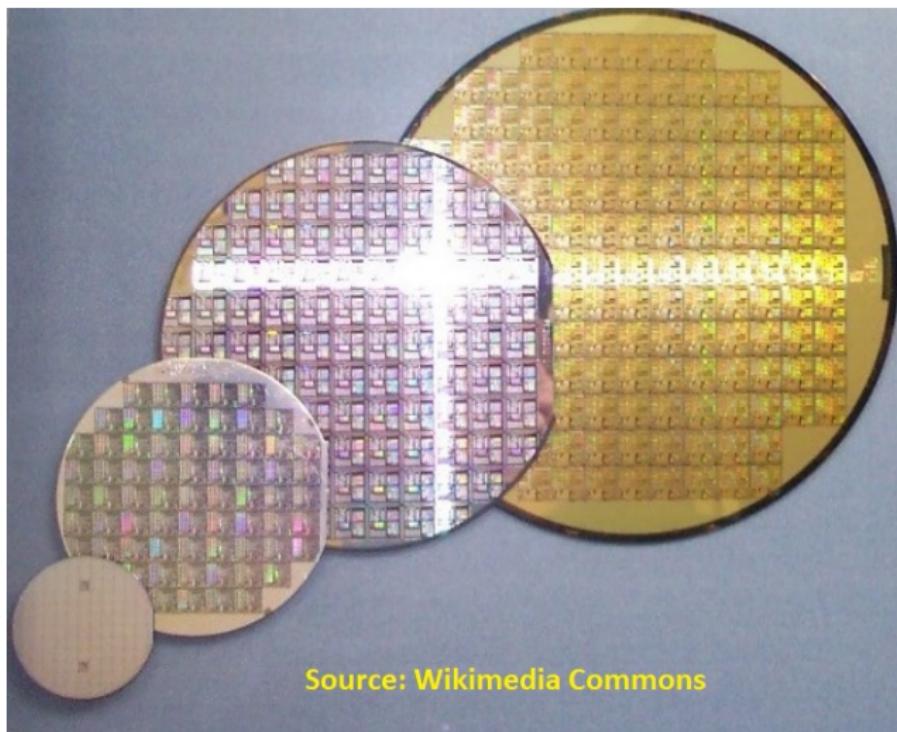
- O processo básico produz a forma da máscara no chip.
- Lentes de redução colocadas entre a máscara e o wafer reduzem a forma da máscara ao chegar no wafer.
- O processo básico é usado no processo completo para produzir as várias camadas (**difusão**, **poli-silício**, **metal**, etc.) na pastilha, conforme as respectivas máscaras.
- O próximo slide mostra as etapas do processo completo NMOS.

# Processo Completo

Source: Mead and Conway - Introduction to VLSI Systems

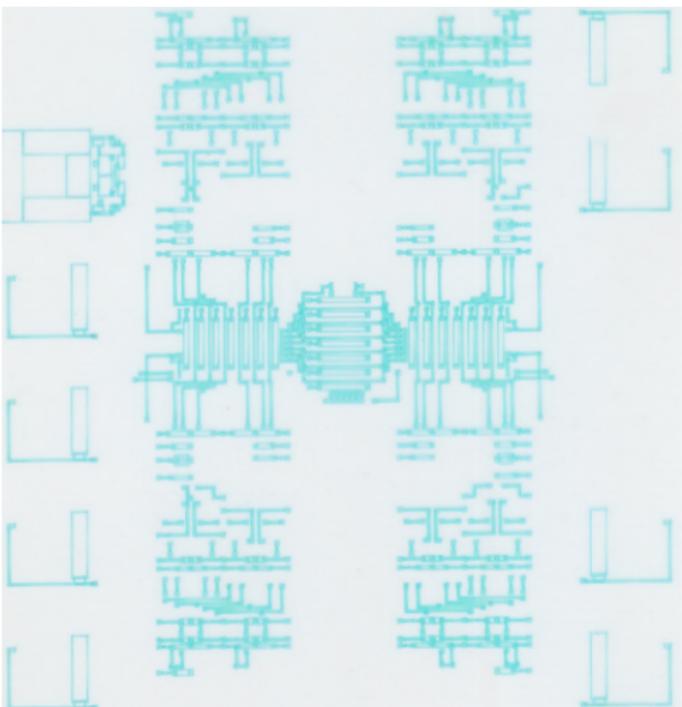


# Wafers de Silício

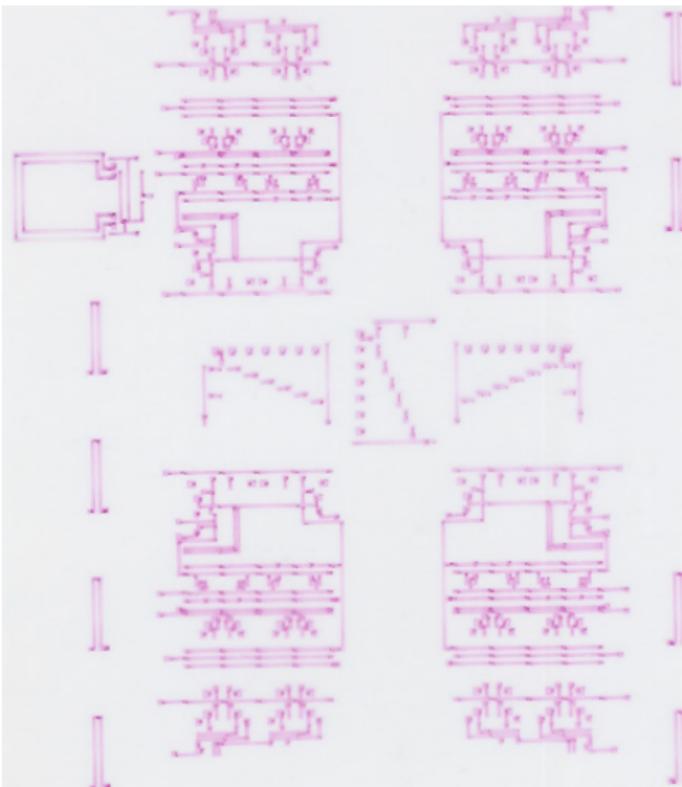


Source: Wikimedia Commons

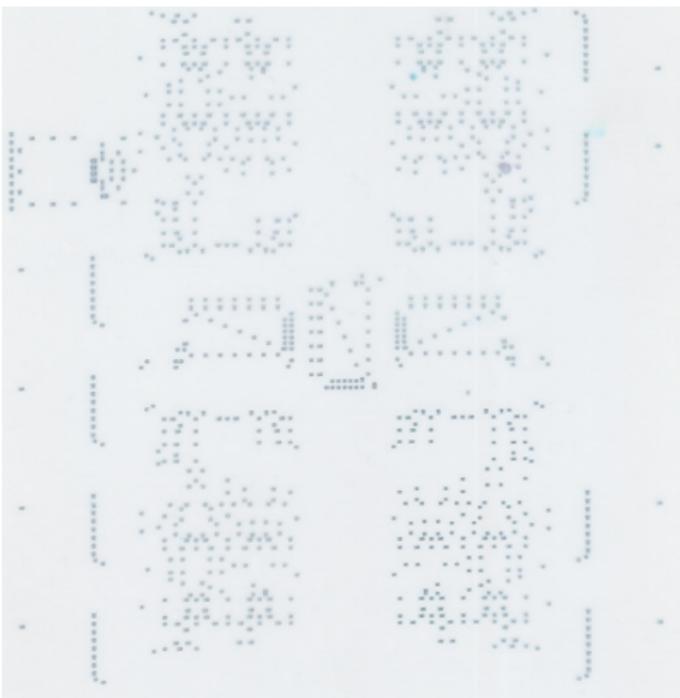
# Máscara Difusão



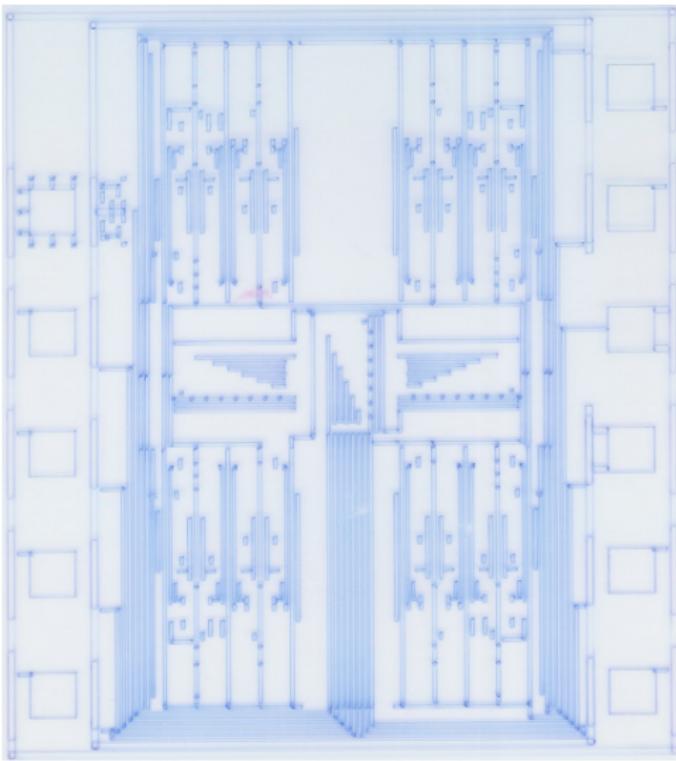
# Máscara Poli-silício



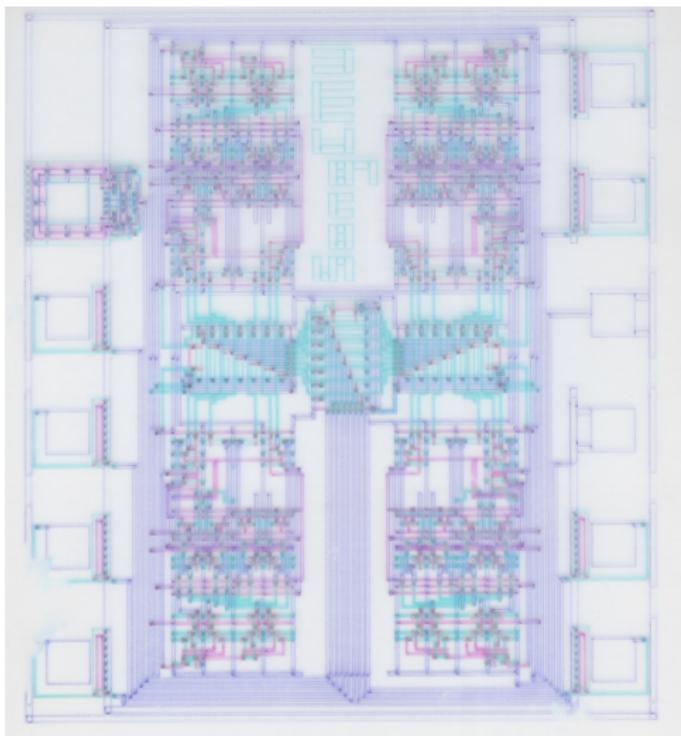
# Máscara Contatos



# Máscara Metal



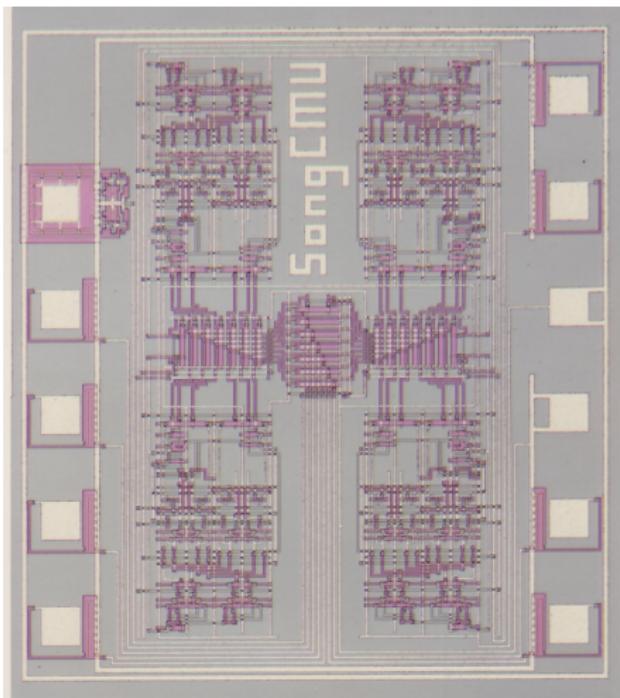
# Todas As Camadas Juntas



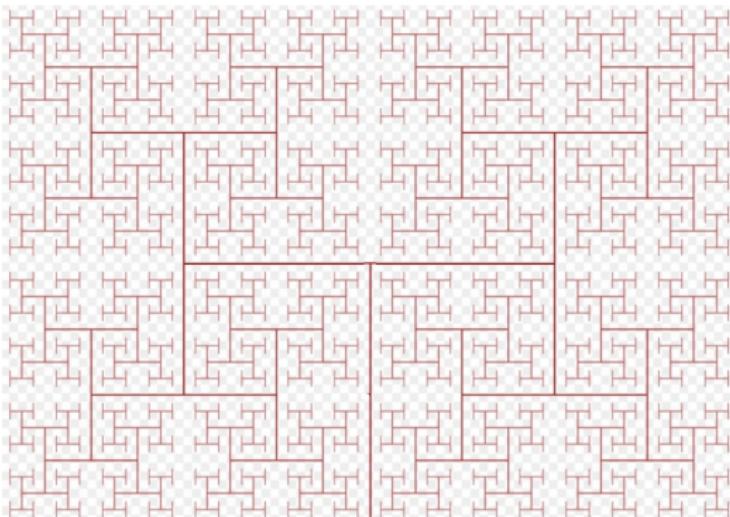
# Projeto de pastilha customizada para aplicação específica

- A tecnologia VLSI é usada para processadores e memória.
- Propicia também o projeto de pastilhas customizadas para aplicações específicas ou ASICs (Application Specific Integrated Circuits). Tipicamente isso é feito usando FPGAs (Field Programmable Gate Arrays).
- ASICs podem também ser projetados com o método de Arranjos Sistólicos (Systolic Arrays) propostos nos anos 80.
- O Systolic Array consiste de um conjuntos de células (elementos de processamento) simples interconectadas de uma forma regular no plano.

# Projeto busca por árvore binária

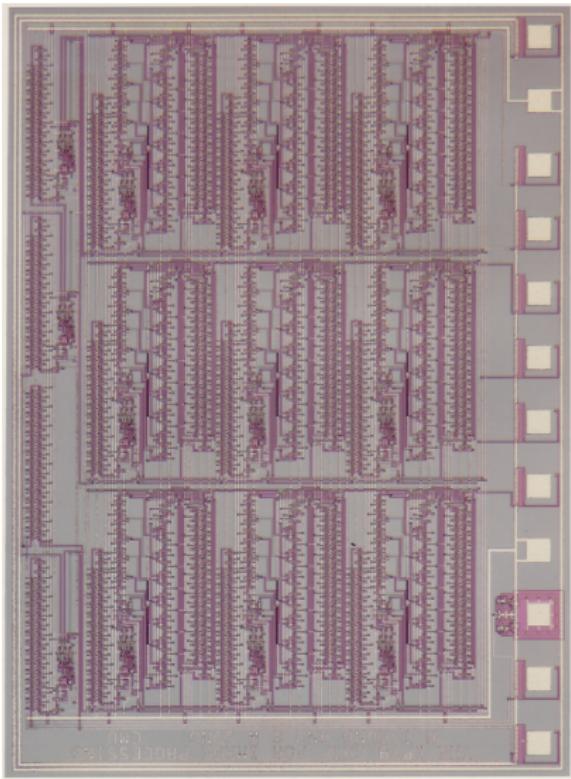


# Disposição-H de uma árvore binária no plano

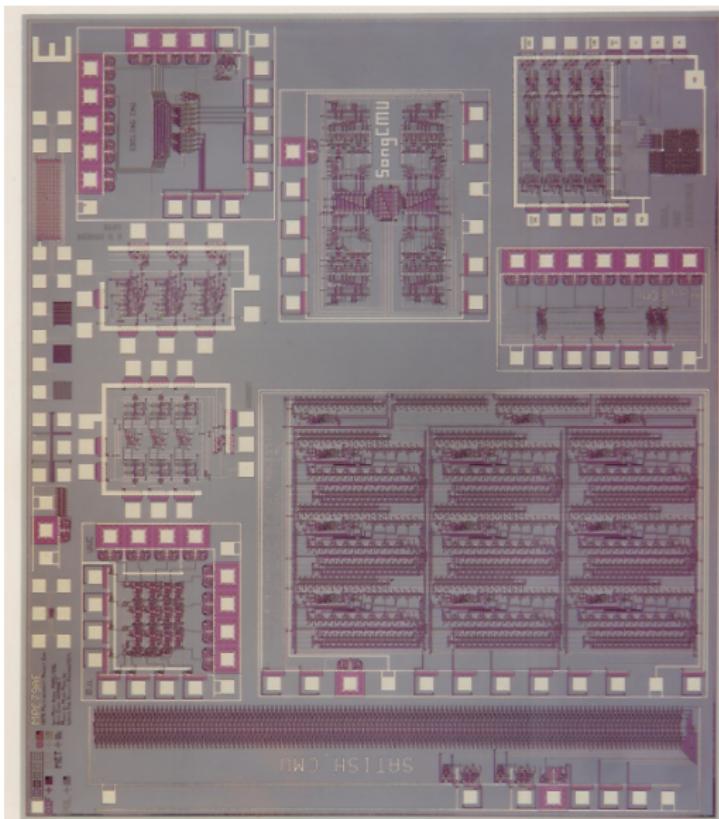


- Acima mostramos uma árvore binária em disposição-H (nome devido à forma H que aparece no desenho) que melhor utiliza o espaço.
- Quantos nós tem essa árvore acima? (Tente desenhá-la na forma usual de representar uma árvore binária (i.e.: ) no mesmo espaço acima :-)

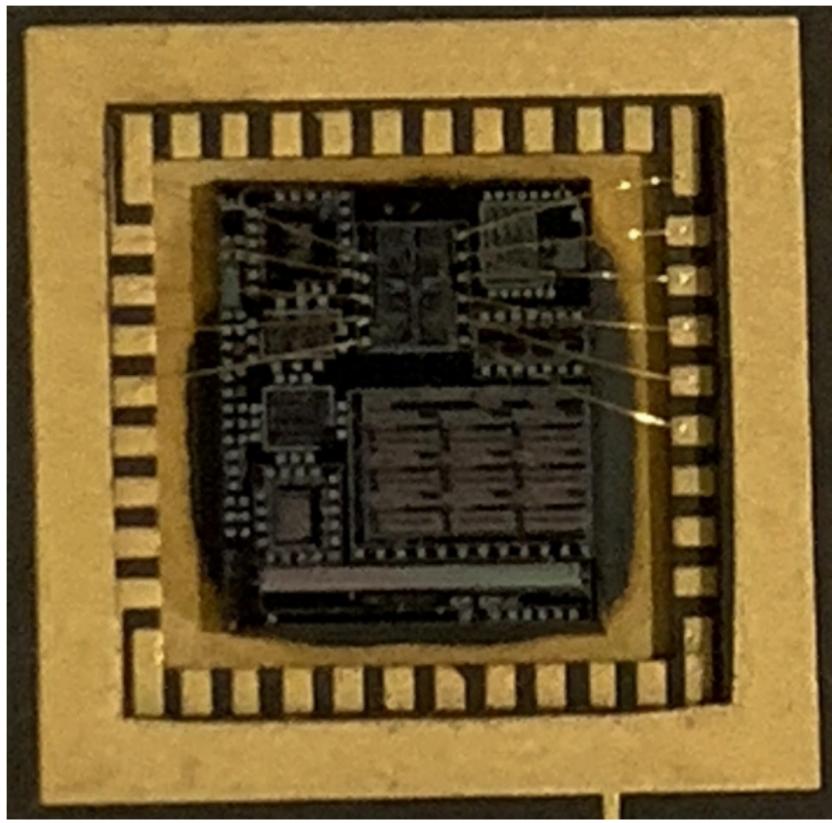
# Projeto convolução



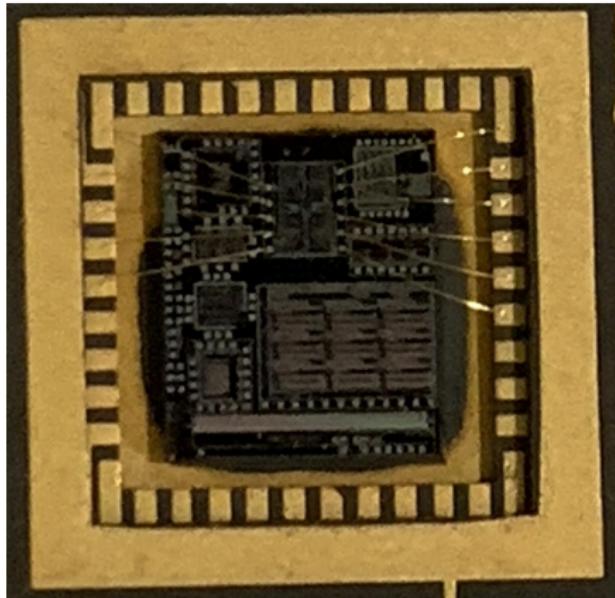
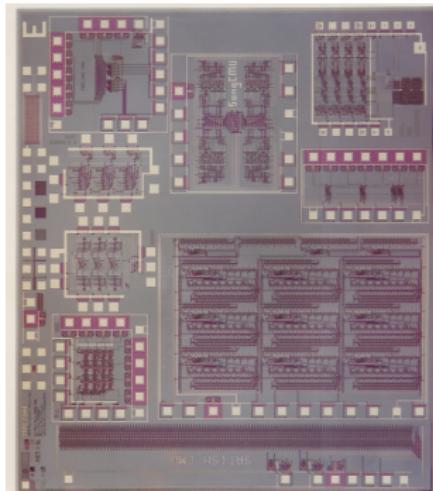
# Pastilha multiprojeto



# Pastilha multiprojeto



# Pastilha multiprojeto



# Array sistólico - um exemplo

- Vamos mostrar um exemplo de um array sistólico que multiplica duas matrizes.

[Clicar aqui para ver o exemplo \(mp4\).](#)

# Array sistólico - a moda vai e volta

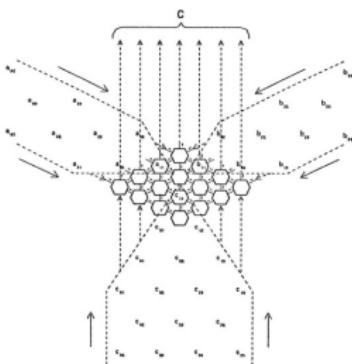


Figure 4-2: The hex-connected systolic array for the matrix multiplication problem in Figure 4-1.

- Proposto em 1978, array sistólico despertou enorme interesse na época.
- Mas com o tempo a moda passou e ficou latente durante quase trinta anos.
- Até que **ressurge em 2016** pela Google TPU (Tensor Processing Unit).

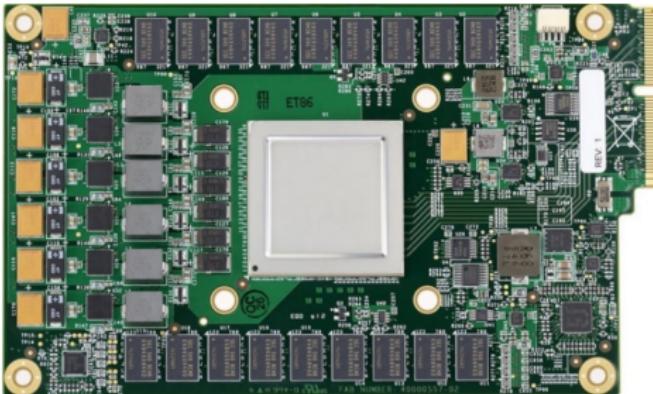


Source: Google

Google's tensor processing unit or TPU.

- Array sistólico **ressurge na figura da Google TPU** (Tensor Processing Unit) que é usado em **Google Search, Google Street View, Google translate** para acelerar as computações de **redes neurais** em aprendizado de máquina.

# Google TPU - Tensor Processing Unit 2016



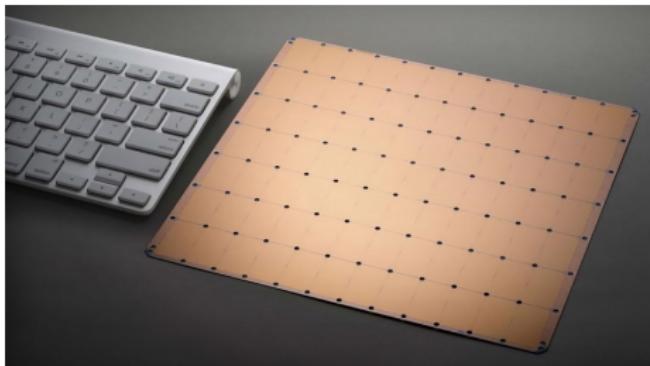
Google's first TPU

- Primeira geração TPU (2016): um  $256 \times 256$  array sistólico que realiza **multiplicação de matrizes** de números inteiros de 8 bits, e operação de **convolução**.

- Segunda geração TPU (maio 2017): multiplicação de matrizes em ponto flutuante, com desempenho de 11,5 PetaFLOPS, usada no treinamento e inferência em redes neurais para aprendizado de máquina.
- Terceira geração TPU (maio 2018): oito vezes mais rápido que TPU da segunda geração.

An in-depth look at Google's first Tensor Processing Unit (TPU). Kaz Sato (Staff Developer Advocate, Google Cloud), Cliff Young (Software Engineer, Google Brain), David Patterson (Distinguished Engineer, Google Brain) May 12, 2017.

# WSP - Wafer Scale Processing - 2,6 trilhão transistores



- WSP - Wafer Scale Processing: usar todo o wafer para uma CPU
- Cerebras WSP com 2,6 trilhão transistores e 850.000 cores.
- Tecnologia de 7 nm.

# Principais fabricantes de chips VLSI

Hoje existem 3 fabricantes no mundo capazes de produzir chips com a tecnologia de 7 nm. ([Clicar aqui para a reportagem completa.](#))

- Taiwan Semiconductor Manufacturing Company (TSMC)

Para um vídeo sobre esse fabricante, ver:

[Inside The World's Largest Semiconductor Factory - BBC \(4:17 minutos\)](#)

- Samsung
- Intel

A previsão é que em 2024 será possível produzir chips com a tecnologia de 5 nm.

# Próximo assunto: Como aumentar o desempenho do processador

- Próximo assunto: Técnicas para aumentar o desempenho do processador
- Ao longo dos anos, várias técnicas foram criadas visando maior velocidade do processador.
- Em 2018 vulnerabilidades (Meltdown e Spectre) foram descobertas que exploram essas técnicas. (Vermos Meltdown e Spectre mais tarde, primeiro vamos ver as tais técnicas...)
- Não percam!

# 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 paralela ou de forma concorrente (se não houver dependência), etc.
  - Lei de Amdahl sobre a limitação da computação paralela.

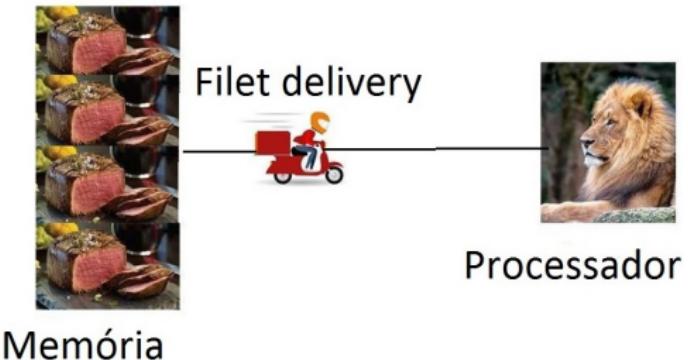
## Evolução do desempenho - frequência do relógio

- 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 numa pastila de silício.

- **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 principal 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.

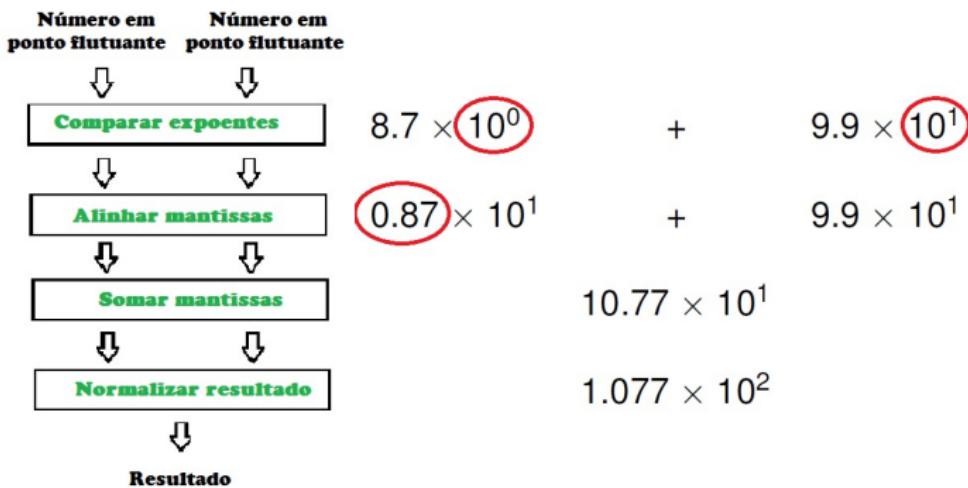
# Pipeline de instruções

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 predição de desvio.
- Ex.: Pipelining de instruções foi implementado já no Intel 80486.

# Pipeline de operações aritméticas em ponto flutuante



- 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

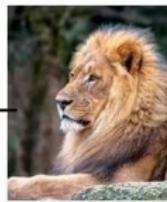
- Diversas técnicas foram desenvolvidas para balancear a velocidade do processador com os demais componentes.  
Exemplos:
  - pré-busca de instruções
  - predição de desvios
  - análise de fluxo de dados para verificar dependência de dados
  - execução especulativa
  - uso da memória cache (veremos mais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



Filet delivery



Processador

Memória

# 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**.



Memória

Filet delivery



Processador  
superescalar

# Paralelismo no nível de instrução

- Processador superescalar possui múltiplas unidades de execução de instruções.
- Vabiliza 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:  $A = \dots$   
 $\dots = A \dots$

$$1: A = A + 2$$

$$2: B = 2 \times A$$

$$3: C = B - A$$

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:  $A = \dots$   
 $\dots = A \dots$

Coloque os presentes e depois pegar os presentes: OK.



# Dependência verdadeira

Modelo:  $A = \dots \Rightarrow \Rightarrow \Rightarrow \Rightarrow \Rightarrow \dots = A \dots$   
 $\dots = A \dots$  Mudar a ordem não dá  $A = \dots$

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:  $\dots = A \dots$   
 $A = \dots$

$$1: B = A + 5$$

$$2: A = 7$$

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

1 : ... = ... A ...  
2 : A = ...

Remoção da anti-dependência

0 : A1 = A  
1 : ... = ... A1 ...  
2 : A = ...

- 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.

$A = \dots$   
Modelo:     $\dots$   
               $A = \dots$

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

Instrução 3 tem dependência de saída em relação à instrução 1 (escrevemos 1  $\rightarrow^{saída}$  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

1 :  $A = \dots$

2 :  $\dots = \dots$  se aparecer  $A \dots$

3 :  $A = \dots$

Remoção da dependência de saída

1 :  $A1 = \dots$

2 :  $\dots = \dots$  trocar por  $A1 \dots$

3 :  $A = \dots$

- A instrução 1 :  $A1 = \dots$  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:  $A = Y * Y$

Renomeamos 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 seguintes 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 seguintes 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
- $2 \rightarrow^{saída} 4$ : 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?

# 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

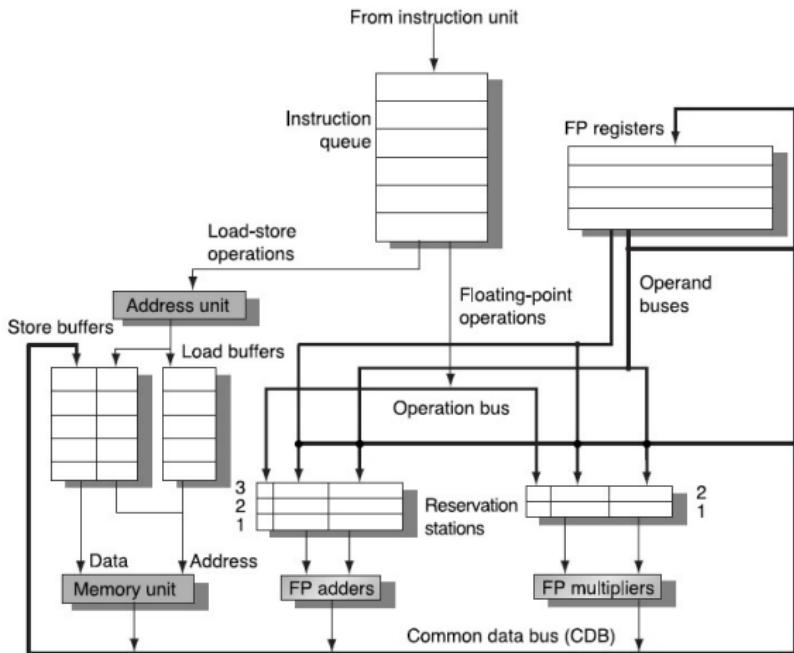
Foi recipiente do 1997 Eckert-Mauchly Award. Fez carreira na IBM onde desenvolveu o System/360 e sucessores. Trabalhou também num projeto que desenvolveu o primeiro computador de grande porte baseado em CMOS.

- 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 pelo renomeamento de 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

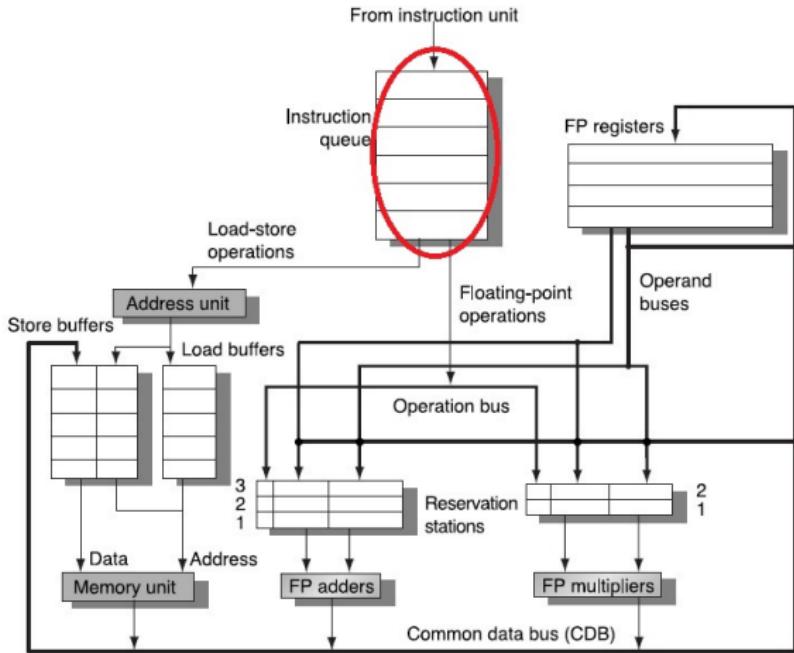
- O algoritmo de Tomasulo foi implementado em hardware no IBM 360/91. A mesma ideia é usada depois em outros processadores, como MIPS, Pentium Pro, DEC Alpha, PowerPC, etc.
- Rastreia quando operandos estão disponíveis a fim de satisfazer dependências.
- Remove anti-dependências e dependências de saída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

# Algoritmo de Tomasulo - Não cai em prova



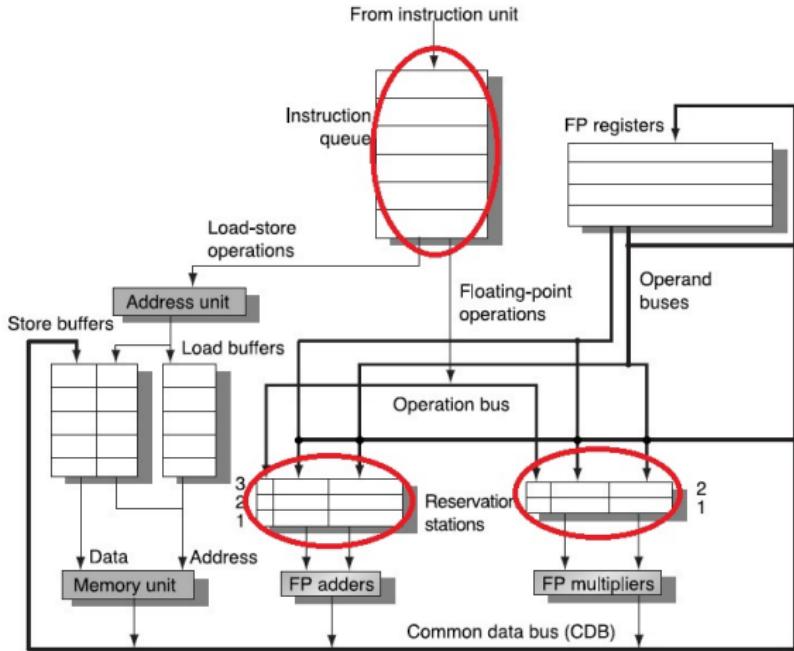
Vamos explicar de forma superficial o algoritmo de Tomasulo.

# Algoritmo de Tomasulo - Não cai em prova



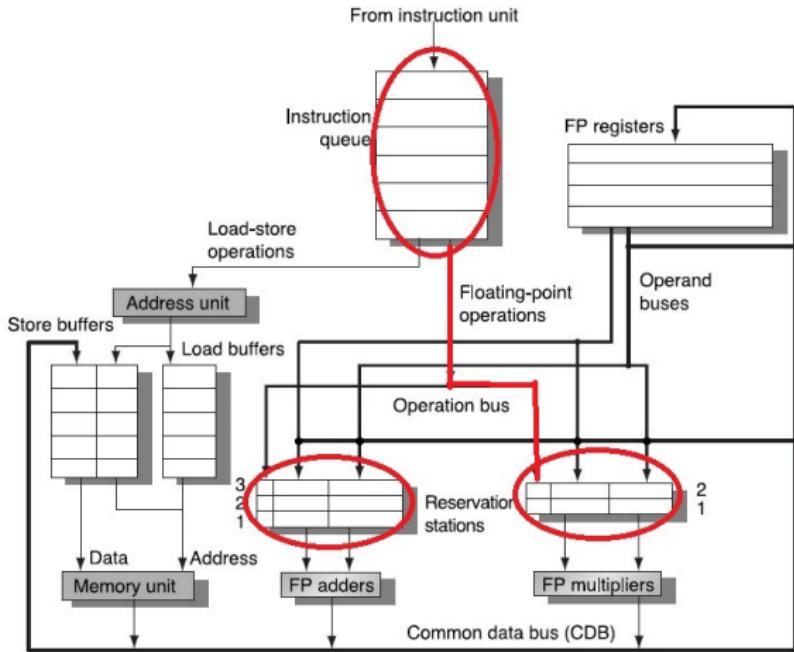
Instruções são carregadas em *instruction queue*.

# Algoritmo de Tomasulo - Não cai em prova



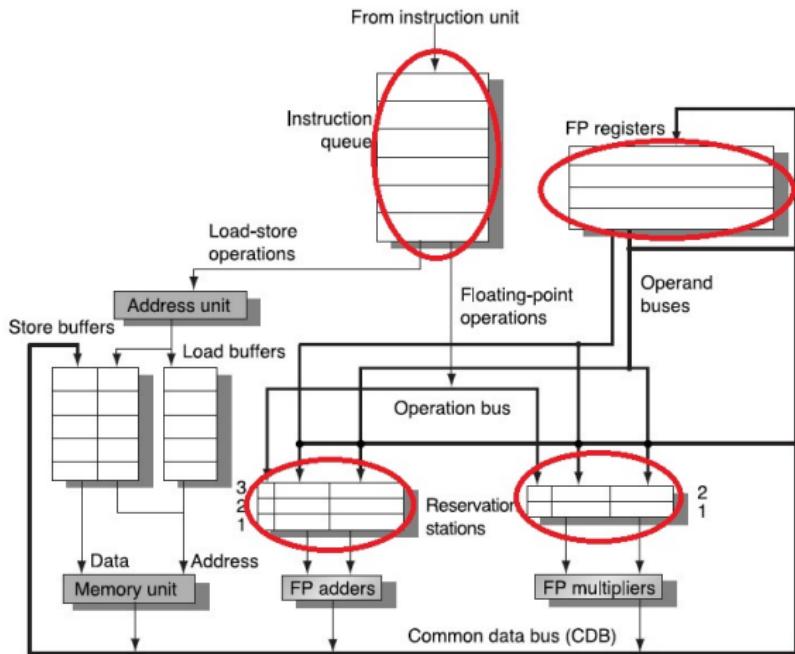
Usa um número de *reservation stations* para a execução de instruções.

# Algoritmo de Tomasulo - Não cai em prova



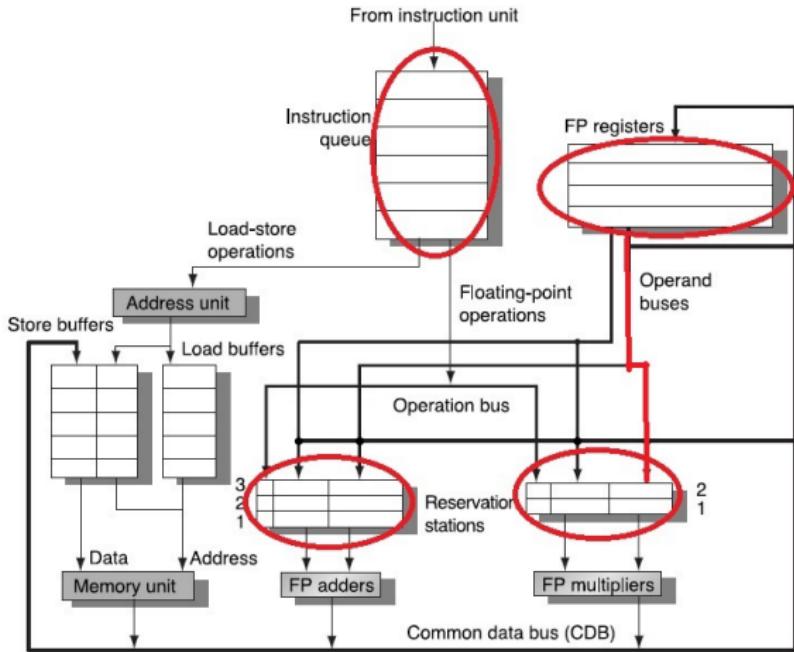
Pega uma instrução da *instruction queue* e ache uma *reservation station* livre para ela.

# Algoritmo de Tomasulo - Não cai em prova



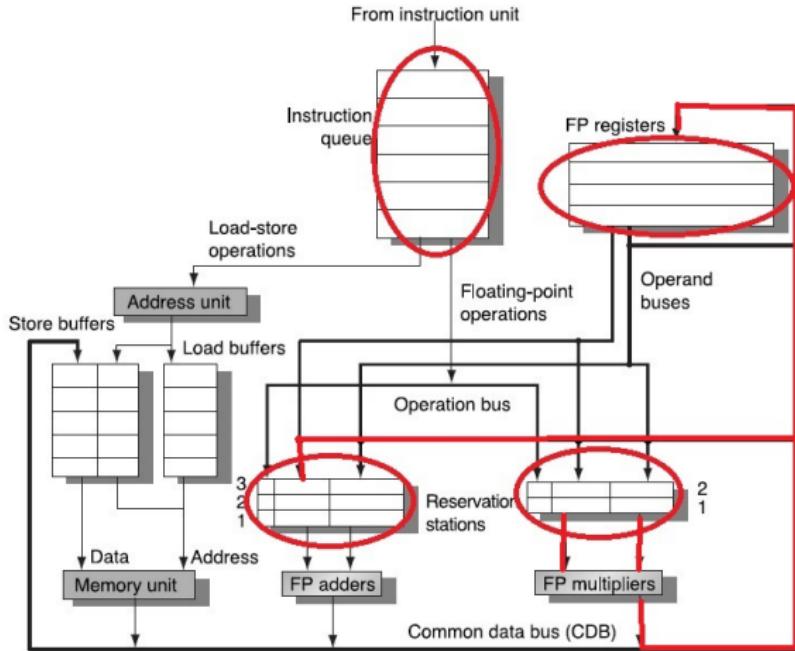
Registradores armazenam dados lidos ou resultados produzidos.

# Algoritmo de Tomasulo - Não cai em prova



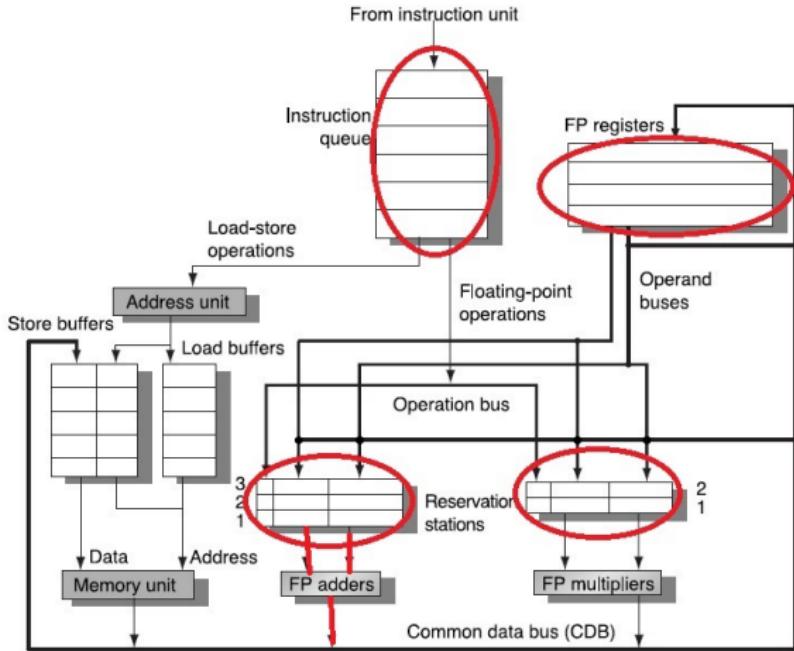
Lê operandos que estão em registradores. Se o operando não está lá localiza qual *reservation station* vai produzi-lo.

# Algoritmo de Tomasulo - Não cai em prova



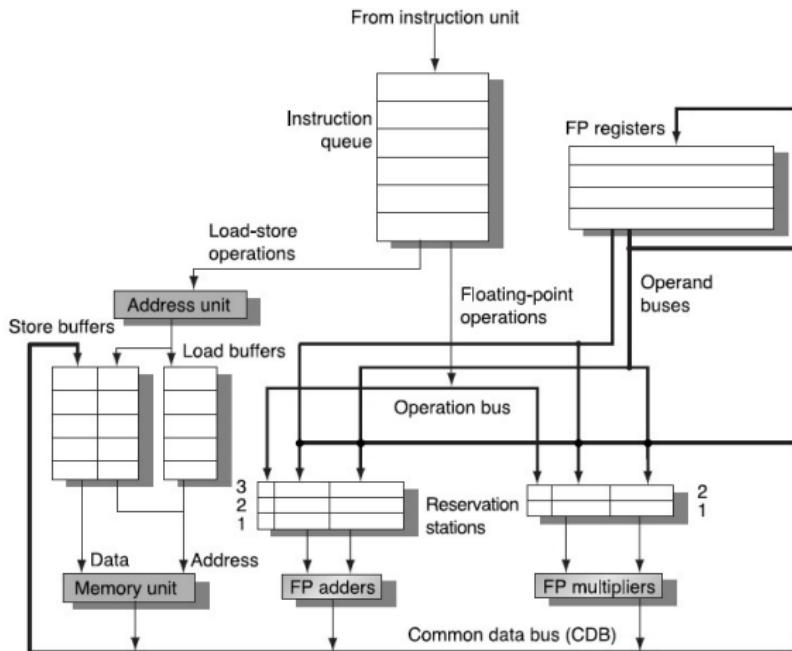
Monitora resultados quando produzidos. Coloca o resultado em todas as *reservation stations* que estão esperando por ele.

# 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 funcionamento 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.

127	87 86	46 45	5 4 0
instruction slot 2	instruction slot 1	instruction slot 0	template
41	41	41	5

[Intel Itanium Architecture - Instruction Set](#)

# Execução especulativa

- Usando predição de desvio e análise de fluxo de dados, alguns processadores especulativamente executam 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    condição    then trecho 1  
      else trecho 2
```

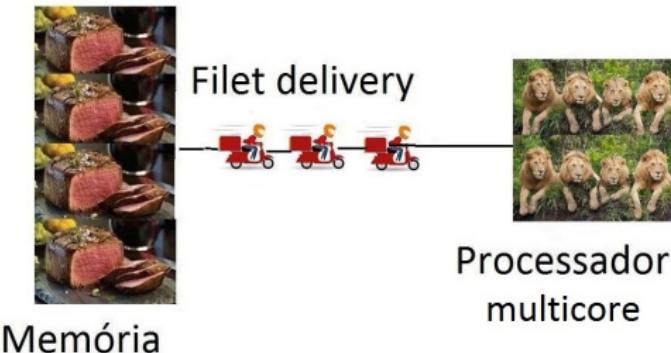
- Os resultados da execução especulativa são armazenados em locais temporários e somente validados depois.
- Essa técnica faz com que o processador fique sempre ocupado ao executar instruções que provavelmente seriam necessárias.
- Pentium Pro implementa as técnicas superescalar, predição de desvios,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

```
if condição 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.



- 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

- **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 processadores (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.”

# 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:
  - Predição de desvio
  - Execução fora de ordem
  - Execução especulativa
  - Uso de memória cache

- 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.

- 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:

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

- Considere

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

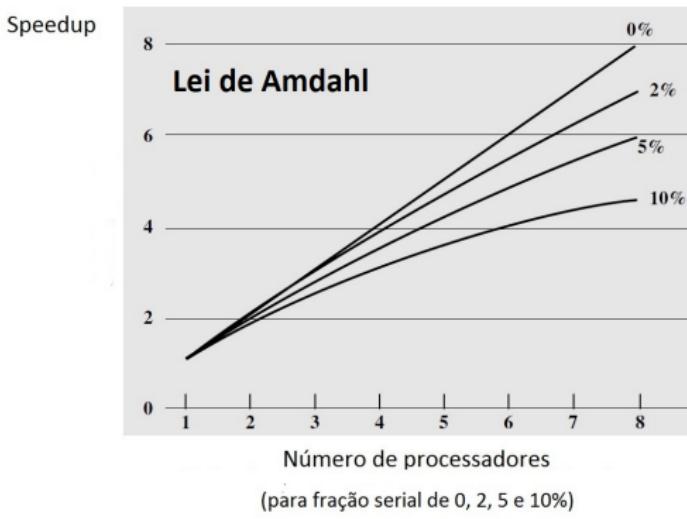
- Um programa executando com um único processador onde
  - uma fração  $f$  do código pode ser paralelizado perfeitamente
  - uma fração  $(1 - f)$  do código é inherentemente sequencial

$$\text{speedup} = \frac{T_1}{T_n} = \frac{T_1}{T_1(1-f) + \frac{T_1f}{n}} = \frac{1}{(1-f) + \frac{f}{n}}$$

- Para um grau de paralelismo muito grande ( $n \rightarrow \infty$ ), o speedup é limitado por  $\frac{1}{(1-f)}$ .
- Se  $f$  é pequeno, então mesmo o paralelismo maciço não ajuda.

# 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 inherentemente serial, i.e. fração paralela é  $f = 0.9$ .
- Então executar esse código com 8 cores produz um  $speedup = \frac{1}{(1-f)+\frac{f}{n}} = 4,7$ .



Source: W. Stallings

- 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:   A[i] = B[i] + 10
3:   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.

```
1: for i from 1 to 100 do  
2:   A[i, 0] = 0  
3: end for  
4: for i from 1 to 100 do  
5:   for j from 1 to 100 do  
6:     A[i, j] = A[i, j - 1] × 2  
7:   end for  
8: 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:

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

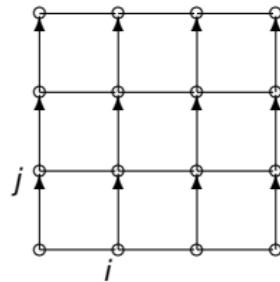
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = A[i, j - 1] × 2  
4:   end for  
5: 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:



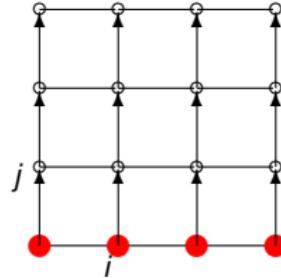
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = A[i, j - 1] × 2  
4:   end for  
5: 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:



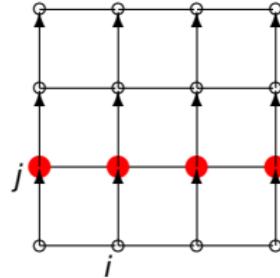
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = A[i, j - 1] × 2  
4:   end for  
5: 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:



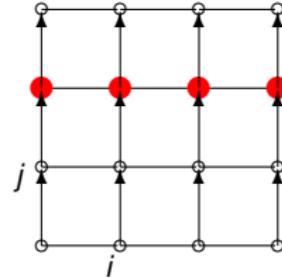
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = A[i, j - 1] × 2  
4:   end for  
5: 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:



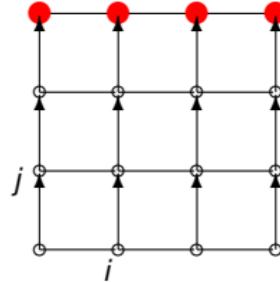
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = A[i, j - 1] × 2  
4:   end for  
5: 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.

```
1: for k from 1 to 100 do
2:   A[k, 0] = 0
3:   A[0, k] = 0
4: end for
5: for i from 1 to 100 do
6:   for j from 1 to 100 do
7:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}
8:   end for
9: 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:

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

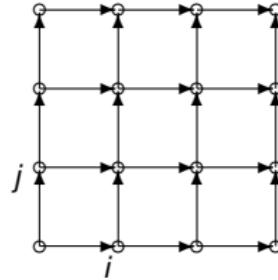
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



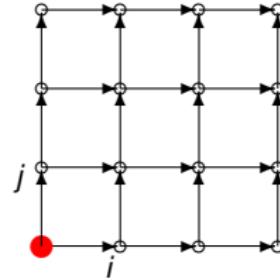
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



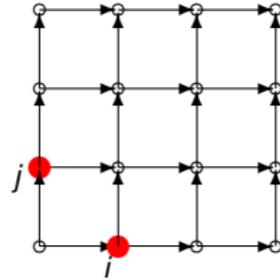
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



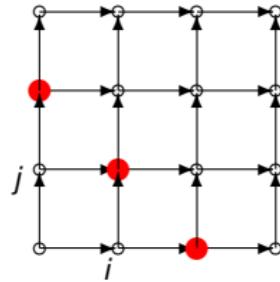
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



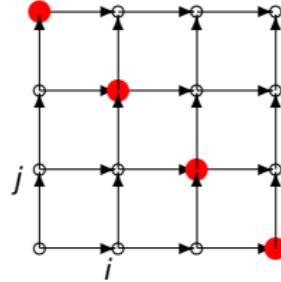
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



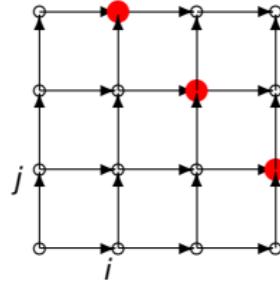
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



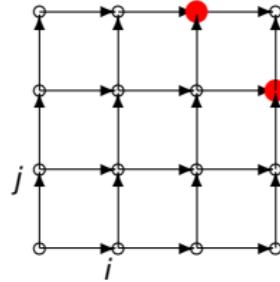
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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:



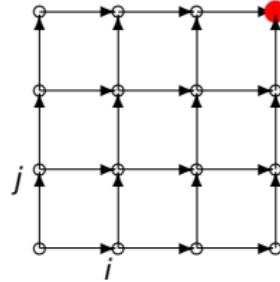
# Paralelização de laços (*loops*) - Não cai em prova

```
1: for i from 1 to 100 do  
2:   for j from 1 to 100 do  
3:     A[i, j] = max{A[i - 1, j], A[i, j - 1]}  
4:   end for  
5: 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 **inherentemente 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 como mapear a memória principal à memória cache.
- Não percam!

# Ada Lovelace - primeira programadora

## Dia da Ada Lovelace - 2.a terça-feira de outubro

MAC0344 - Arquitetura de Computadores  
Prof. Siang Wun Song

Slides usados: <https://www.ime.usp.br/~song/mac344/slides-ada.pdf>

Fonte do material: Wikipedia e Jornal da USP: [Por que as mulheres desaparecerem dos cursos de Computação?](#)

# Ada Lovelace 1815 - 1852



**12/10/2021**

**Dia da Ada**

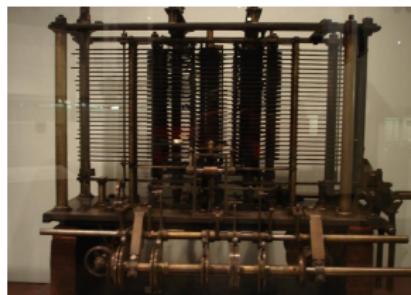
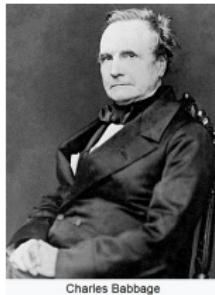
**2.a terça-feira de outubro**

- Filha de Lord Byron
- Casada com William King - Earl of Lovelace
- Estudou Matemática com Augustus De Morgan, entre outros.



Augustus De Morgan (1806–1871)

# Charles Babbage e a Máquina Analítica



Charles Babbage

- Trabalhou com Charles Babbage na programação da Máquina Analítica.
- Ada percebeu que a máquina de Babbage podia fazer mais que simplesmente cálculos aritméticos.
- Escreveu o considerado primeiro algoritmo de computador: um algoritmo para calcular Números de Bernoulli.
- Ada Lovelace é considerada a primeira programadora do mundo.
- A Máquina Analítica nunca chegou a ficar operacional e portanto o algoritmo da Ada não chegou a ser executado.

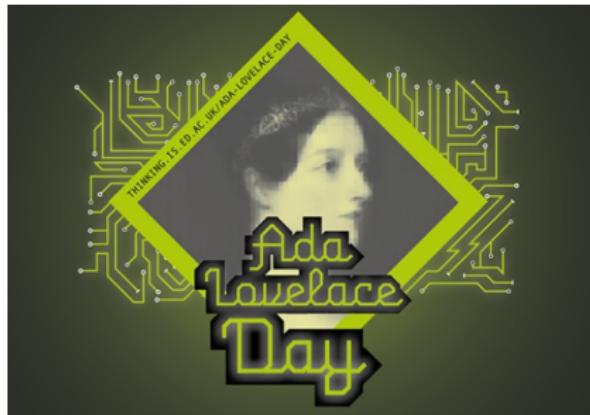
# Homenagens



Blue plaque to Lovelace in [St. James's Square, London](#)

- Linguagem de programação ADA: linguagem estruturada, imperativa e orientada a objetos.
- Aprimora a segurança do código e o compilador ajuda a achar erros.
- Em 1981, foi inaugurado o Ada Lovelace Award.
- Em 1998, British Computer Society criou a Lovelace Medal.

# Ada Lovelace Day



- Començando em 2009, Ada Lovelace Day é comemorado na segunda terça-feira de outubro de cada ano.
- A data foi instituída com o objetivo de encorajar mulheres na Ciência, Tecnologia, Engenharia e Matemática.

# 1.a turma formando(a)s do BCC do IME/USP - 1974

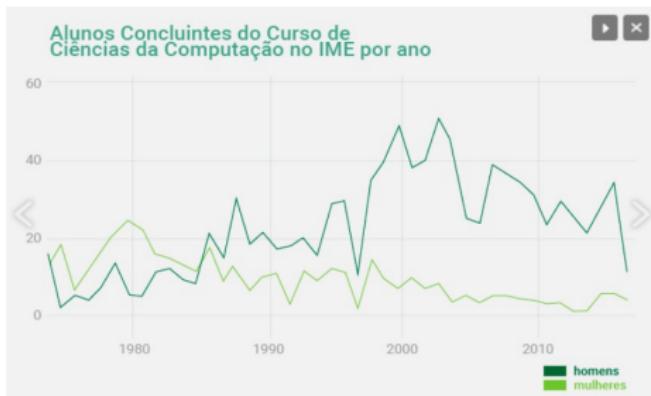


- 20 formando(a)s do Bacharelado em Ciência da Computação do IME/USP 1974: 14 mulheres e 6 homens.
- Ver o artigo do Jornal da USP [Por que as mulheres desaparecerem dos cursos de Computação?](#)

# 1.a turma formando(a)s do BCC do IME/USP - 2022



- No dia 6 de setembro de 2022, alguns formando(a)s da primeira turma se encontraram no IME-USP.
- Ver no link: [Formandos da primeira turma do BCC se encontram no IME-USP](#)



- A tendência de ter menos mulheres em Computação é mundial.
- **Meninas Digitais**: uma iniciativa da Sociedade Brasileira de Computação para despertar o interesse de meninas estudantes do ensino médio e dos anos finais do ensino fundamental sobre Computação e seguir uma carreira em Computação.



# Hierarquia de memória e a memória cache

MAC0344 - Arquitetura de Computadores

Prof. Siang Wun Song

Slides usados:

<https://www.ime.usp.br/~song/mac344/slides04-cache-memory.pdf>

Baseado parcialmente em W. Stallings -  
Computer Organization and Architecture

# Hierarquia de memória

- Veremos hierarquia de memória e depois memória cache.
- Ao final das aulas, vocês saberão
  - Os vários níveis da hierarquia de memória: desde as memórias mais rápidas (de menor capacidade de armazenamento) próximas ao processador até as mais lentas (de grande capacidade de armazenamento) distantes do processador.
  - A importância da memória cache em agilizar o acesso de dados/instruções pelo processador, graças ao fenômeno de localidade.
  - A memória interna ou principal é muito maior que a memória cache. Há vários métodos para mapear um bloco de memória à memória cache.

# Hierarquia de memória

Há vários tipos de memórias, cada uma com características distintas em relação a

- Custo (preço por bit).
- Capacidade para o armazenamento.
- Velocidade de acesso.

*A memória ideal seria aquela que é barata, de grande capacidade e acesso rápido.*

*Infelizmente, a memória rápida é custosa e de pequena capacidade.*

*E a memória de grande capacidade, embora mais barata, apresenta baixa velocidade de acesso.*

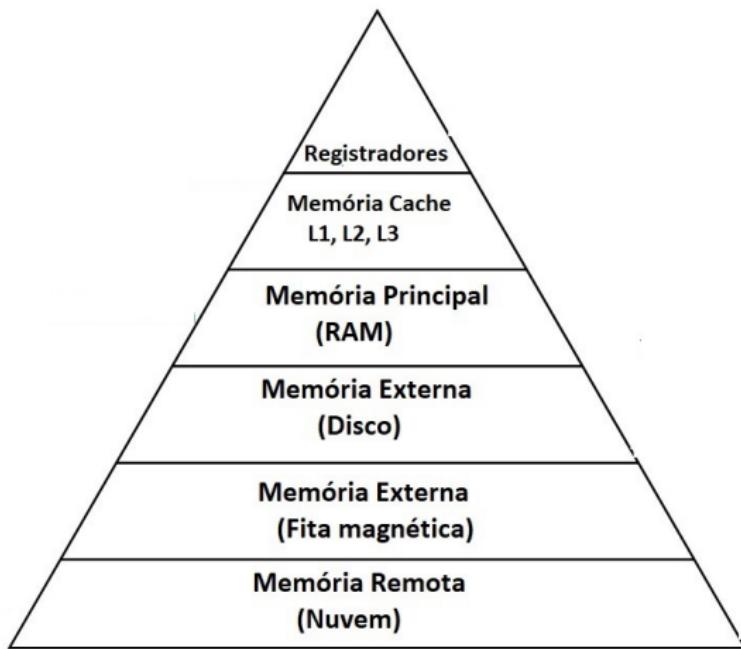
# Hierarquia de memória

A memória de um computador é organizada em uma hierarquia.

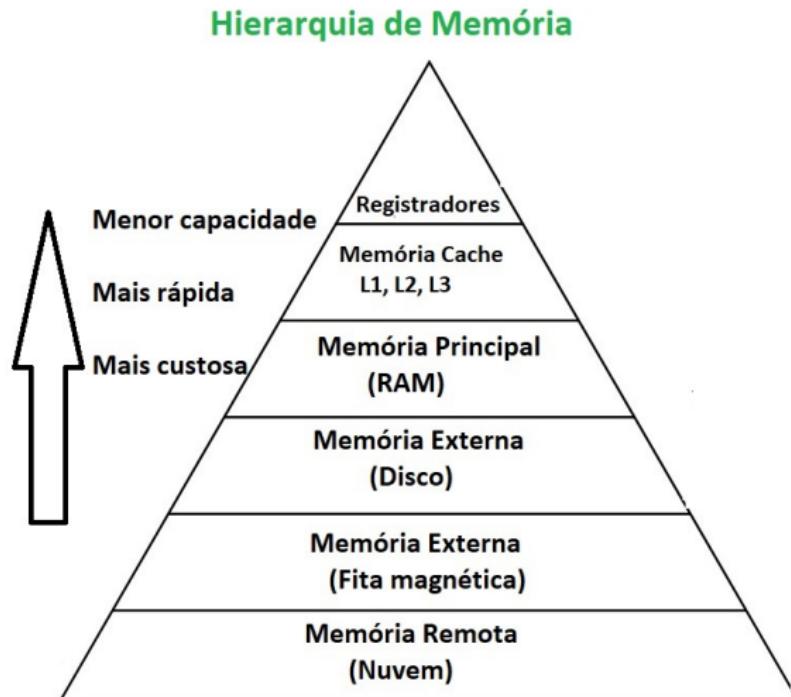
- Registradores: nível mais alto, são memórias rápidas dentro ao processador.
- Vários níveis de memória cache: L1, L2, etc.
- Memória interna ou principal (RAM).
- Memórias externas ou secundárias (discos, fitas).
- Outros armazenamento remotos (arquivos distribuídos, servidores web).

# Hierarquia de memória

## Hierarquia de Memória

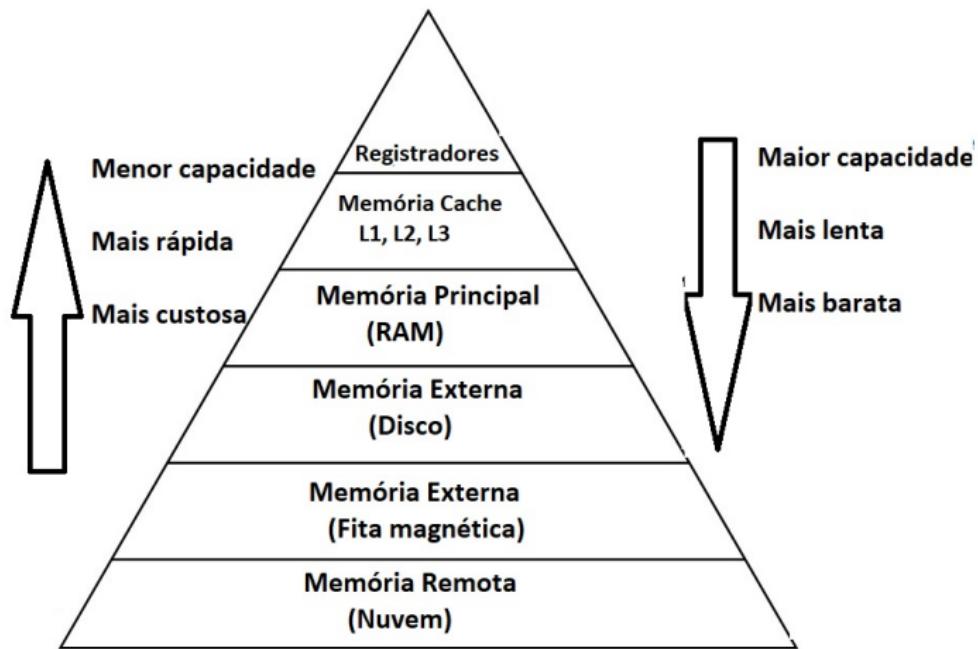


# Hierarquia de memória



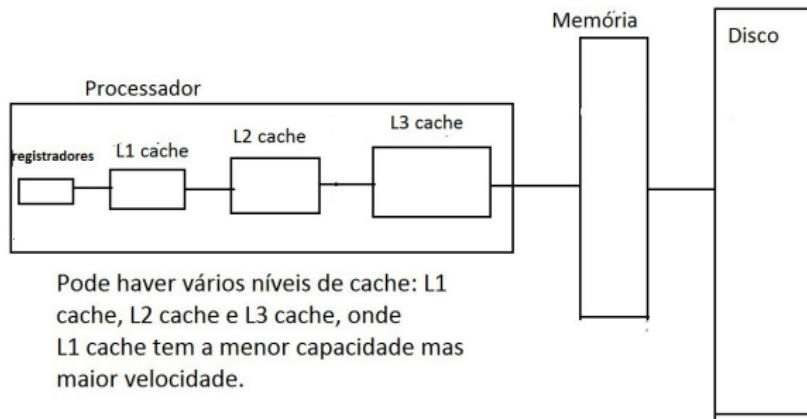
# Hierarquia de memória

## Hierarquia de Memória



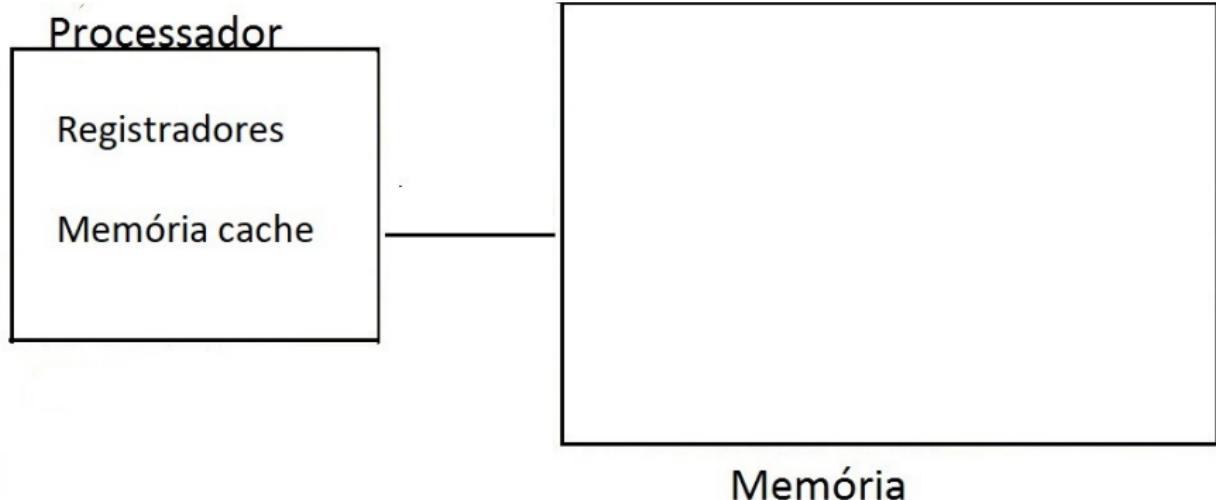
# Memória cache

Veremos memória cache e, nas próximas aulas, memória interna e memória externa.



- Um dado (ou instrução) no disco pode ter uma cópia na memória e no processador (num registrador ou na memória cache).
- Quando o processador precisa de um dado, ele pode já estar na cache (**cache hit**). Se não (**cache miss**), tem que buscar na memória (ou até no disco).
- Quando um dado é acessado na memória, um bloco inteiro (e.g. 64 bytes) contendo o dado é trazido à memória cache. Blocos vizinhos podem também ser trazidos à memória cache (*prefetching*) para possível uso futuro.
- Na próxima vez o dado (ou algum dado vizinho) é usado, já está na cache cujo acesso é rápido.

# Memória cache



- Se uma instrução ou dado já está no processador (registrador ou memória cache), o acesso é rápido. Senão tem que buscar na memória e é guardado na memória cache.

# Memória cache

Processador



Cache

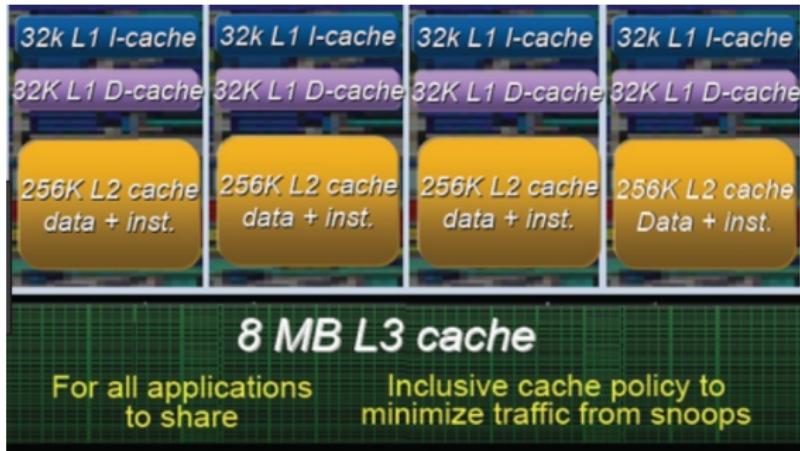


Memória

Images source: Wikimedia Commons

- Analoga: se falta ovo (dado) na cozinha (processador), vai ao supermercado (memória) e compra uma dúzia (*prefetching*), deixando na geladeira (cache) para próximo uso.

# Memória cache no Intel core i7



Intel core i7 cache (L3 cache também conhecida como LL ou Last Level cache)

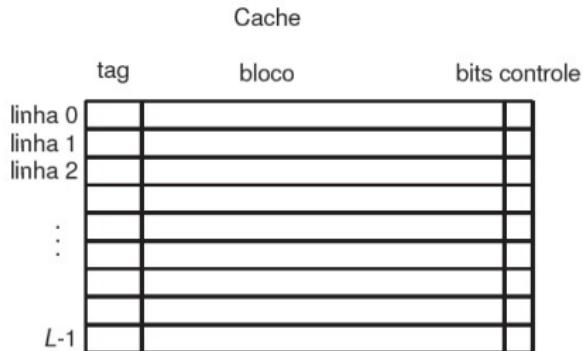
Hierarquia memória	Latência em ciclos
registraror	1
L1 cache	4
L2 cache	11
L3 cache	39
Memória RAM	107
Memória virtual (disco)	milhões

# Evolução de memória cache

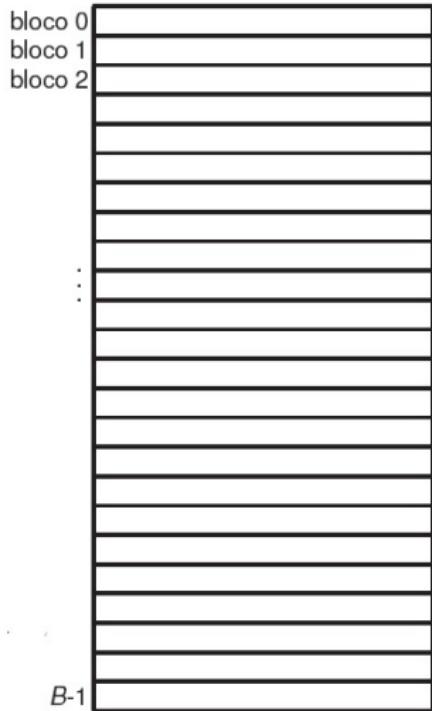
Processador	Ano fabr.	L1 cache	L2 cache	L3 cache
VAX-11/780	1978	16 KB	-	-
IBM 3090	1985	128 KB	-	-
Pentium	1993	8 KB	256 KB	-
Itanium	2001	16 KB	96 KB	4 MB
IBM Power6	2007	64 KB	4 MB	32 MB
IBM Power9 (24 cores)	2017	(32 KB I + 64 KB D) por core	512 KB por core	120 MB por chip

Alguns processadores duplicam um dado da memória cache L1 também na cache L2. Outros não compartilham o mesmo dado na cache L1 e na cache L2. Se um dado não é encontrado nas caches L1 e L2, então a procura continua na cache L3 e, se necessário, na memória principal.

# Memória cache

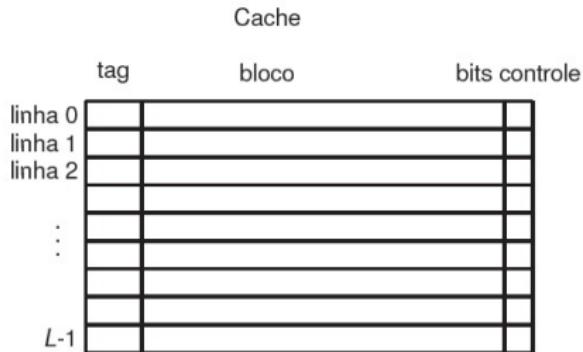


Memória organizada em  $B$  blocos ( $B >> L$ )

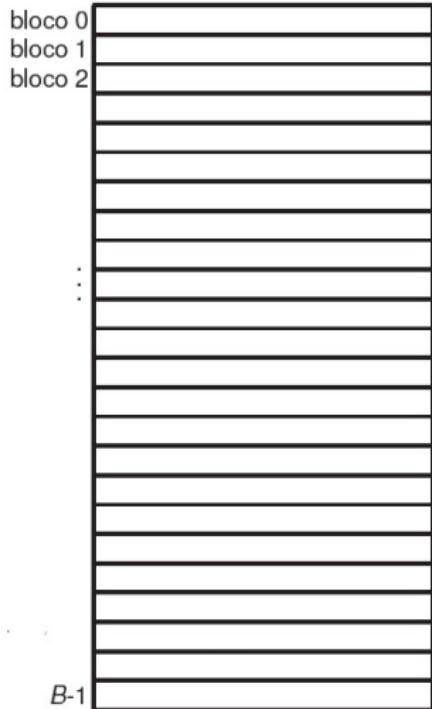


- Memória organizada em  $B$  blocos, numerados bloco 0, bloco 1, ..., bloco  $B - 1$ .
- Cache contém  $L$  linhas, numeradas linha 0, linha 1, ..., linha  $L - 1$ .  $B \gg L$ .
- Cada linha contém um tag, um bloco de memória e bits de controle.
- tag contém info sobre o endereço do bloco na memória.
- bits de controle informam se a linha foi modificada depois de carregada na cache.

# Memória cache

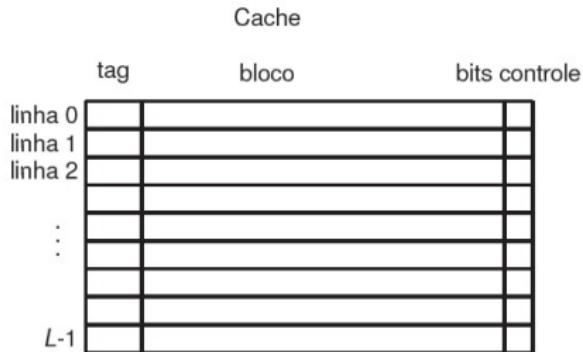


Memória organizada em  $B$  blocos ( $B >> L$ )

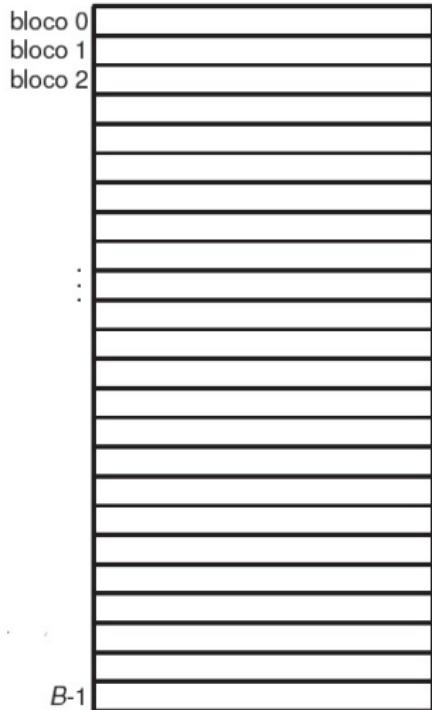


- Memória organizada em  **$B$  blocos**, numerados bloco 0, bloco 1, ..., bloco  $B - 1$ .
- Cache contém  **$L$  linhas**, numeradas linha 0, linha 1, ..., linha  $L - 1$ .  $B >> L$ .
- Cada linha contém um **tag**, um **bloco de memória** e **bits de controle**.
- tag contém info sobre o endereço do bloco na memória.
- bits de controle informam se a linha foi modificada depois de carregada na cache.

# Memória cache

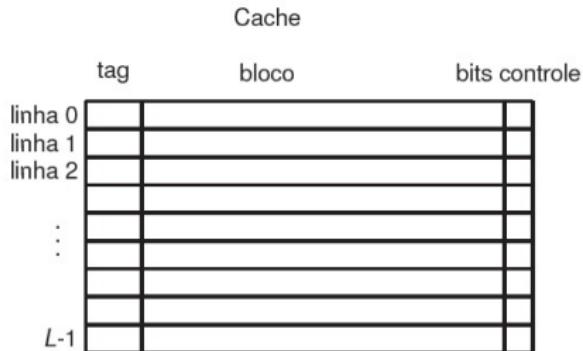


Memória organizada em  $B$  blocos ( $B >> L$ )

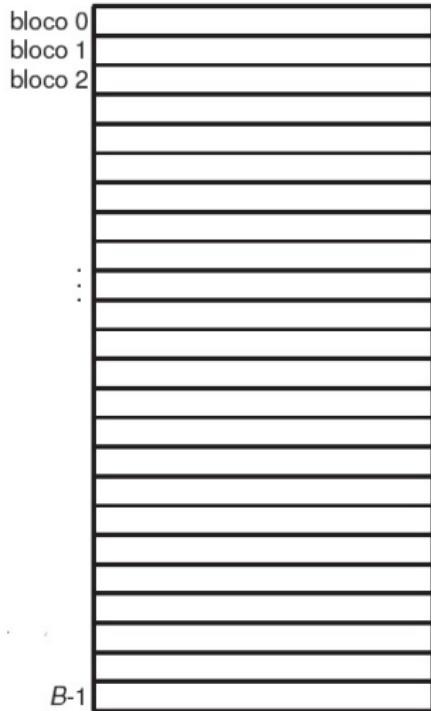


- Memória organizada em  **$B$  blocos**, numerados bloco 0, bloco 1, ..., bloco  $B - 1$ .
- Cache contém  **$L$  linhas**, numeradas linha 0, linha 1, ..., linha  $L - 1$ .  **$B >> L$** .
- Cada linha contém um **tag**, um **bloco de memória** e **bits de controle**.
- tag contém info sobre o endereço do bloco na memória.
- bits de controle informam se a linha foi modificada depois de carregada na cache.

# Memória cache

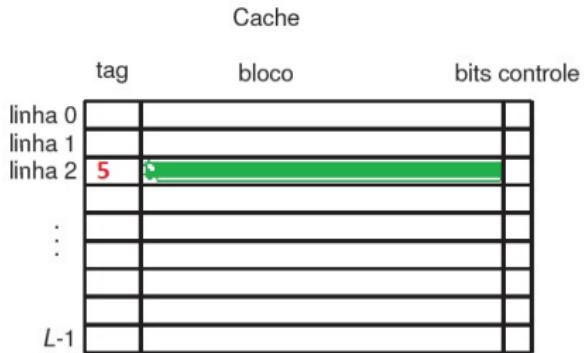


Memória organizada em  $B$  blocos ( $B >> L$ )

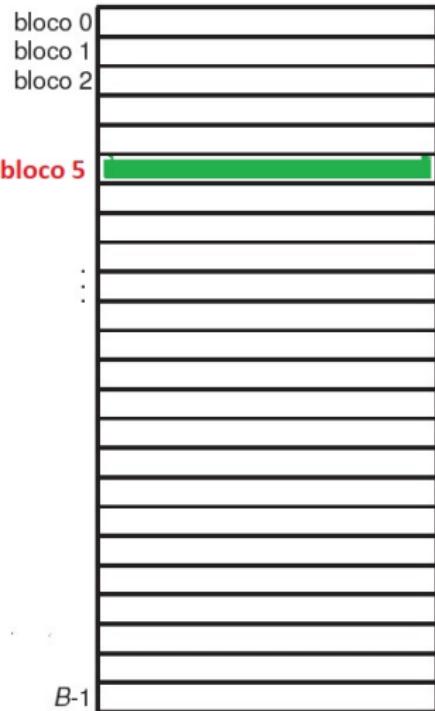


- Memória organizada em  $B$  blocos, numerados bloco 0, bloco 1, ..., bloco  $B - 1$ .
- Cache contém  $L$  linhas, numeradas linha 0, linha 1, ..., linha  $L - 1$ .  $B >> L$ .
- Cada linha contém um tag, um bloco de memória e bits de controle.
- tag contém info sobre o endereço do bloco na memória.
- bits de controle informam se a linha foi modificada depois de carregada na cache.

# Memória cache



Memória organizada em  $B$  blocos ( $B \gg L$ )



- Memória organizada em  $B$  blocos, numerados bloco 0, bloco 1, ..., bloco  $B - 1$ .
- Cache contém  $L$  linhas, numeradas linha 0, linha 1, ..., linha  $L - 1$ .  $B \gg L$ .
- Cada linha contém um tag, um bloco de memória e bits de controle.
- tag contém info sobre o endereço do bloco na memória.
- Se bloco 5 da memória foi colocado na linha 2 da cache, então tag contém 5.

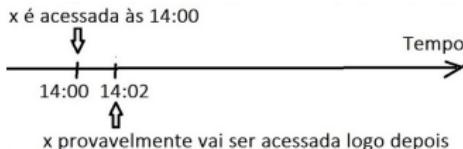
# Memória cache

- As caches L1 e L2 ficam em cada *core* de um processador multicore e a L3 é compartilhada por todos os *cores*.
- A capacidade da cache é muito menor que a capacidade da memória. (Se a memória é virtual, parte dela fica no disco.)
- A memória cache contém apenas uma fração da memória.
- Quando o processador quer ler uma palavra da memória, primeiro verifica se o bloco que a contém está na cache.
- Se achar (conhecido como *cache hit*), então o dado é fornecido sem ter que acessar a memória.
- Se não achar (conhecido como *cache miss*), então o bloco da memória interessada é lido e colocada na cache.
- A razão entre o número de *cache hit* e o número total de buscas na cache é conhecida como *hit ratio*.

# Memória cache

## Fenômeno de localidade:

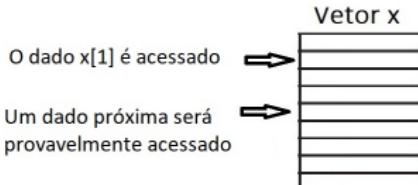
- **Localidade temporal:** Um dado ou instrução acessado recentemente tem maior probabilidade de ser acessado novamente, do que um dado ou instrução acessado há mais tempo.



```
for i = 0 to 9999 do
begin
    x = 0;
    y = z + 1;
    z = u / 2;
    .
    .
    .
end
```

Essa instrução é acessada na iteração  $i$   
Essa mesma instrução será acessada na iteração  $i + 1$ .

- **Localidade espacial:** Se um dado ou instrução é acessado recentemente, há uma probabilidade grande de acesso a dados ou instruções próximos.



```
for i = 0 to 9999 do
begin
    x = 0;
    y = z + 1;
    z = u / 2;
    .
    .
    .
end
```

Essa instrução é executada  
Uma instrução próxima provavelmente também será executada

- Há menos linhas da memória cache do que número de blocos.
- É necessário definir como mapear blocos de memória a linhas da memória cache.
- A escolha da função de mapeamento determina como a cache é organizada.

Três funções de mapeamento:

- Mapeamento direto
- Mapeamento associativo
- Mapeamento associativo por conjunto

# Memória cache - Função de mapeamento



Source: Wikipedia

- Uma analogia: Um bibliotecário cuida de um acervo de 10.000 livros. Ele notou que um livro recentemente consultado tem grande chance de ser buscado de novo.
- Ele arrumou um escaninho de 100 posições. Cada vez um livro é acessado, ele deixa um exemplar no escaninho. Às vezes ele tem que remover um livro do escaninho para ter espaço.
- Isso evitou ir aos estantes, se o livro desejado já está lá.
- Veremos como ele pode organizar o tal escaninho.

# Memória cache - Mapeamento direto



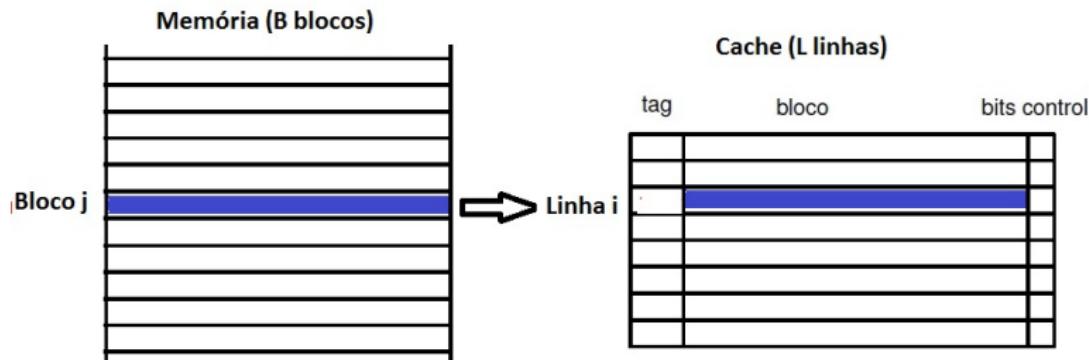
- Para simplificar o exemplo, usamos a numeração decimal. No computador é usada a numeração binária.
- Os 10.000 livros são identificados de 0000 a 9999.
- As 100 posições do escaninho são identificadas de 00 a 99.
- A capacidade do escaninho é potência de 10 e facilita o mapeamento direto. Os dois últimos dígitos do livro são usados para mapear à posição do escaninho.
- Exemplo: livro **8702** é mapeado à posição **02** do escaninho.

# Memória cache - Mapeamento direto



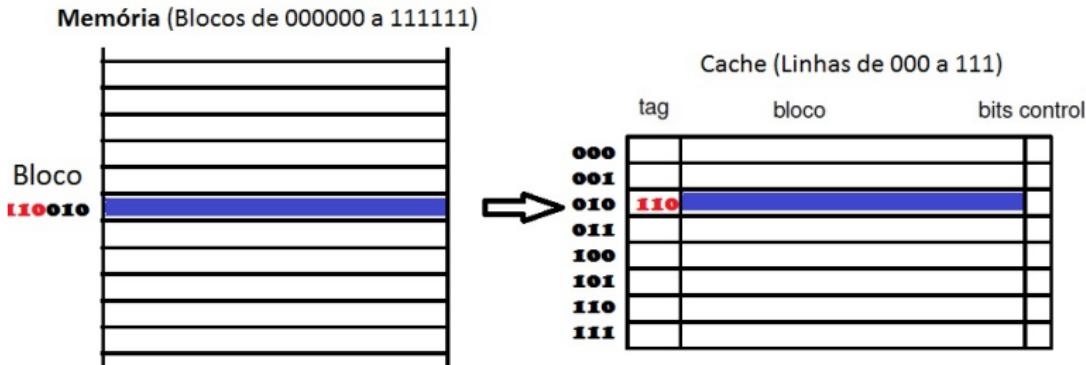
- Para buscar o livro **2513**, a posição mapeada no escaninho (**13**) contém o livro desejado. Não precisa ir buscar nos estantes.
- Para buscar o livro 5510, ele não está na posição 10. Pega nos estantes e deixa um exemplar na posição 10 do escaninho.
- Para buscar o livro **8813**, a posição **13** não contém o livro desejado (**88** é diferente de **25**). Pega o livro desejado nos estantes e coloca na posição 13 do escaninho, removendo o livro que estava lá.

# Memória cache - Mapeamento direto



- Considere  $L$  linhas de cache: linha  $i = 0, \dots, L - 1$ .
- Considere  $B$  blocos de memória: bloco  $j = 0, \dots, B - 1$ .
- No exemplo anterior, onde usamos codificação em sistema decimal,  $L$  é potência de 10. Então foi fácil mapear bloco  $j$  a linha  $i$  da cache.
- Num caso geral, para  $L$  um número qualquer, calculamos  
 $i = j \bmod L$ .

# Memória cache - Mapeamento direto



- Se  $B$  e  $L$  são potências de 2, isto é:
  - $B = 2^s$  blocos de memória (no exemplo  $s = 6$ )
  - $L = 2^r$  linhas de cache (no exemplo  $r = 3$ )
- Então é fácil determinar em que linha  $i$  bloco  $j$  é mapeado: basta pegar os últimos  $r = 3$  bits de  $j$ . Ver figura.
- tag identifica o bloco que está alocado na cache. Bastam os primeiros  $s - r = 6 - 3 = 3$  bits de  $j$  para isso. Ver figura.

# Memória cache - Mapeamento direto

Outro exemplo: para mostrar que basta ter um tag de  $s - r$  bits para identificar qual bloco está mapeado a uma determinada linha.

- Suponha  $s = 5$ , i.e. uma memória de  $B = 2^5 = 32$  blocos.
- Suponha  $r = 2$ , i.e. uma cache de  $L = 2^2 = 4$  linhas.

Bloco mapeado à linha	Linha
00000 00100 01000 01100 10000 10100 11000 11100	00
00001 00101 01001 01101 10001 10101 11001 11101	01
00010 00110 01010 01110 10010 10110 11010 11110	10
00011 00111 01011 01111 10011 10111 11011 11111	11

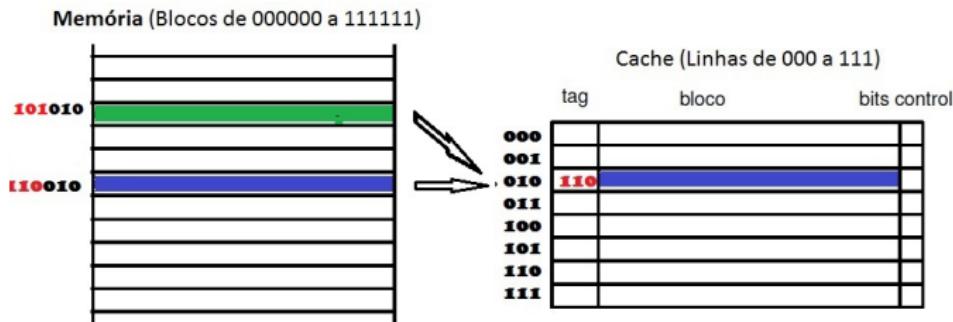
Assim, somente  $s - r = 5 - 2 = 3$  bits (os bits **vermelhos**) são necessários no tag para identificar qual bloco está mapeado, pois os últimos  $r$  bits são iguais à posição da linha.

# Como foi o meu aprendizado?

Tente fazer em casa (não precisa entregar).

- Seja  $s = 5$  e uma memória de  $B = 2^s = 2^5 = 32$  blocos.
- Seja  $r = 2$  e uma cache de  $L = 2^r = 2^2 = 4$  linhas.
- O tag terá  $s - r = 5 - 2 = 3$  bits.
- Desenhe uma memória cache com 4 linhas e uma memória com 32 blocos.
- Acesse o bloco 6 da memória (vai dar cache miss, então introduza na cache).
- Acesse o bloco 8 da memória (vai dar cache miss, então introduza na cache).
- Acesse o bloco 6 da memória (vai dar cache hit).
- Acesse o bloco 4 da memória (vai dar cache miss, então introduza na cache expulsando o que estava lá).

# Memória cache - Mapeamento direto



- O mapeamento direto é simples mas tem uma desvantagem. Se o programa acessa repetida e alternadamente dois blocos de memória mapeados à mesma posição na cache, então esses blocos serão continuamente introduzidos e retirados da cache.
- O fenômeno acima tem o nome de *thrashing*, resultando em um número grande de *hit miss* ou baixa *hit ratio*.
- Para aliviar este problema: Colocar os blocos que causam *thrashing* em uma cache chamada *cache vítima*, tipicamente de 4 a 16 linhas.

# Memória cache - Mapeamento associativo



- Voltemos à analogia. No mapeamento associativo, o livro pode ser colocado em **qualquer** posição do escaninho.
- Para buscar o livro 2513, todo o escaninho precisa ser buscado. A busca fica rápida se todas as posições são buscadas em paralelo (busca associativa).
- Se o livro desejado não está no escaninho, então pega nos estantes e depois deixa um exemplar no escaninho. Se o escaninho já está cheio, então escolhe um livro e o remove.

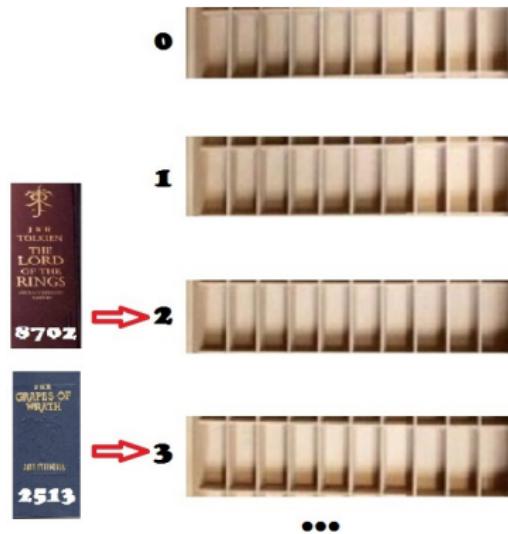
# Memória cache - Mapeamento associativo

- No mapeamento associativo, cada bloco de memória pode ser carregado em qualquer linha da cache.
- Suponha a memória com  $B = 2^s$  blocos.
- O campo tag de uma linha, de  $s$  bits, informa qual bloco está carregado na linha.
- Para determinar se um bloco já está na cache, os campos tag de todas as linhas são examinados simultaneamente. O acesso é dito **associativo**.
- A desvantagem do mapeamento associativo é a complexidade da circuitaria para possibilitar a comparação de todos os tags em paralelo.

## Algoritmos de substituição.

- Quando um novo bloco precisa ser carregado na cache, que já está cheia, a escolha da qual linha deve receber o novo bloco é determinada por um algoritmo de substituição.
- Diversos algoritmos de substituição têm sido propostos na literatura, e.g. LRU (*least recently used*) - a linha que foi menos usada é escolhida para ser substituída.  
Estudaremos esse assunto logo a seguir, nos próximos slides.

# Mapeamento associativo por conjunto



- Na analogia: são usados 10 escaninhos (numerados de 0 a 9).
- O último dígito do livro é usado para mapear a um dos 10 escaninhos. (Mapeamento direto.)
- Dentro do escaninho, o livro pode ser colocado em qualquer posição. (Mapeamento associativo.)
- Combina ambos os mapeamentos, unindo as vantagens.

# Mapeamento associativo por conjunto

É um compromisso entre o mapeamento direto e associativo, unindo as vantagens de ambos. **É o mais usado em processadores modernos.**

- A cache consiste de um número  $v$  de conjuntos, cada um com  $L$  linhas.
- Suponha  $i =$  número do conjunto.
- Suponha  $j =$  número do bloco na memória.
- Temos  $i = j \bmod v$ . I.e. o bloco  $j$  é colocado no conjunto número  $i$ .
- Dentro do conjunto  $i$ , bloco  $j$  pode ser introduzido em qualquer linha. Usa-se acesso associativo em cada conjunto para verificar se um dado bloco está ou não presente.

# Mapeamento associativo por conjunto

- A organização comum é usar 4 ou 8 linhas em cada conjunto:  $L = 4$  ou  $8$ , recebendo o nome de *4-way* ou *8-way set-associative cache*.
- Exemplo de *2-way set-associative cache*:



# Algoritmos de substituição na cache

Quando a cache já está cheia e um novo bloco precisa ser carregado na cache, então um dos blocos ali existentes deve ser substituído, cedendo o seu lugar ao novo bloco.

- No caso de mapeamento direto, há apenas uma possível linha para receber o novo bloco. O bloco velho cede o lugar para o bloco novo.
- Não há portanto escolha.

# Algoritmos de substituição na cache

- Em mapeamento associativo e associativo por conjunto, usa-se um algoritmo de substituição para fazer a escolha. Para maior velocidade, tal algoritmo é implementado em hardware.
- Dentre os vários algoritmos de substituição, o mais efetivo é o **LRU (Least Recently Used)**: substituir aquele bloco que está na cache há mais tempo sem ser referenciado.

*Esse esquema é análogo a substituição de mercadoria na prateleira de um supermercado. Suponha que todas as prateleiras estão cheias e um novo produto precisa ser exibido. Escolhe-se para substituição aquele produto com mais poeira (pois foi pouco acessado.).*



# Algoritmos de substituição na cache

- Em mapeamento associativo e associativo por conjunto, usa-se um algoritmo de substituição para fazer a escolha. Para maior velocidade, tal algoritmo é implementado em hardware.
- Dentre os vários algoritmos de substituição, o mais efetivo é o **LRU (Least Recently Used)**: substituir aquele bloco que está na cache há mais tempo sem ser referenciado.

*Esse esquema é análogo a substituição de mercadoria na prateleira de um supermercado. Suponha que todas as prateleiras estão cheias e um novo produto precisa ser exibido. Escolhe-se para substituição aquele produto com mais poeira (pois foi pouco acessado.).*



# Algoritmo de substituição LRU

O algoritmo de substituição LRU substitui aquele bloco que está na cache há mais tempo sem ser referenciado.

- Numa cache associativa, é mantida uma lista de índices a todas as linhas na cache. Quando uma linha é referenciada, ela move à frente da lista.
- Para acomodar um novo bloco, a linha no final da lista é substituída.
- O algoritmo LRU tem-se mostrado eficaz com um bom *hit ratio*.

Source: Suponha cache com capacidade para 4 linhas. O índice pequeno denota o "tempo de permanência" na cache. <https://www.cs.utah.edu/~mflatt/past-courses/cs5460/lecture10.pdf>

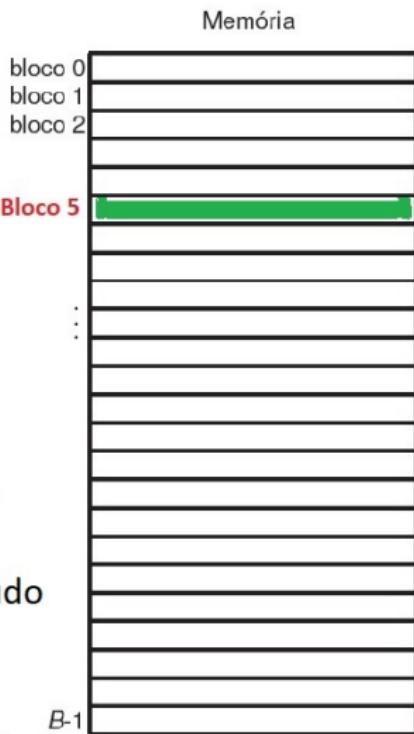
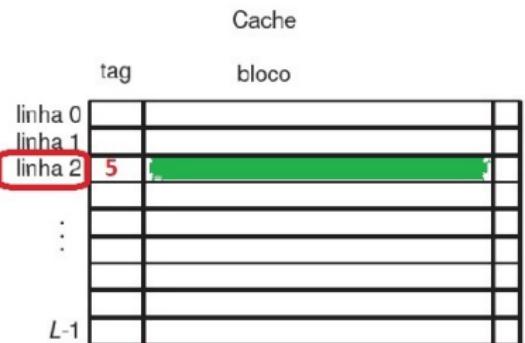
1	2	3	4	1	2	5	1	2	3	4	5
1 <sub>0</sub>	1 <sub>1</sub>	1 <sub>2</sub>	1 <sub>3</sub>	1 <sub>0</sub>	1 <sub>1</sub>	1 <sub>2</sub>	1 <sub>0</sub>	1 <sub>1</sub>	1 <sub>2</sub>	1 <sub>3</sub>	5 <sub>0</sub>
2 <sub>0</sub>	2 <sub>1</sub>	2 <sub>2</sub>	2 <sub>3</sub>	2 <sub>0</sub>	2 <sub>1</sub>	2 <sub>2</sub>	2 <sub>0</sub>	2 <sub>1</sub>	2 <sub>2</sub>	2 <sub>3</sub>	
3 <sub>0</sub>	3 <sub>1</sub>	3 <sub>2</sub>	3 <sub>3</sub>	5 <sub>0</sub>	5 <sub>1</sub>	5 <sub>2</sub>	5 <sub>3</sub>	4 <sub>0</sub>	4 <sub>1</sub>	4 <sub>2</sub>	4 <sub>1</sub>
4 <sub>0</sub>	4 <sub>1</sub>	4 <sub>2</sub>	4 <sub>3</sub>	4 <sub>4</sub>	4 <sub>5</sub>	3 <sub>0</sub>	3 <sub>1</sub>	3 <sub>2</sub>			

# Algoritmo de substituição pseudo-LRU

Há uma maneira mais rápida de implementar um pseudo-LRU.

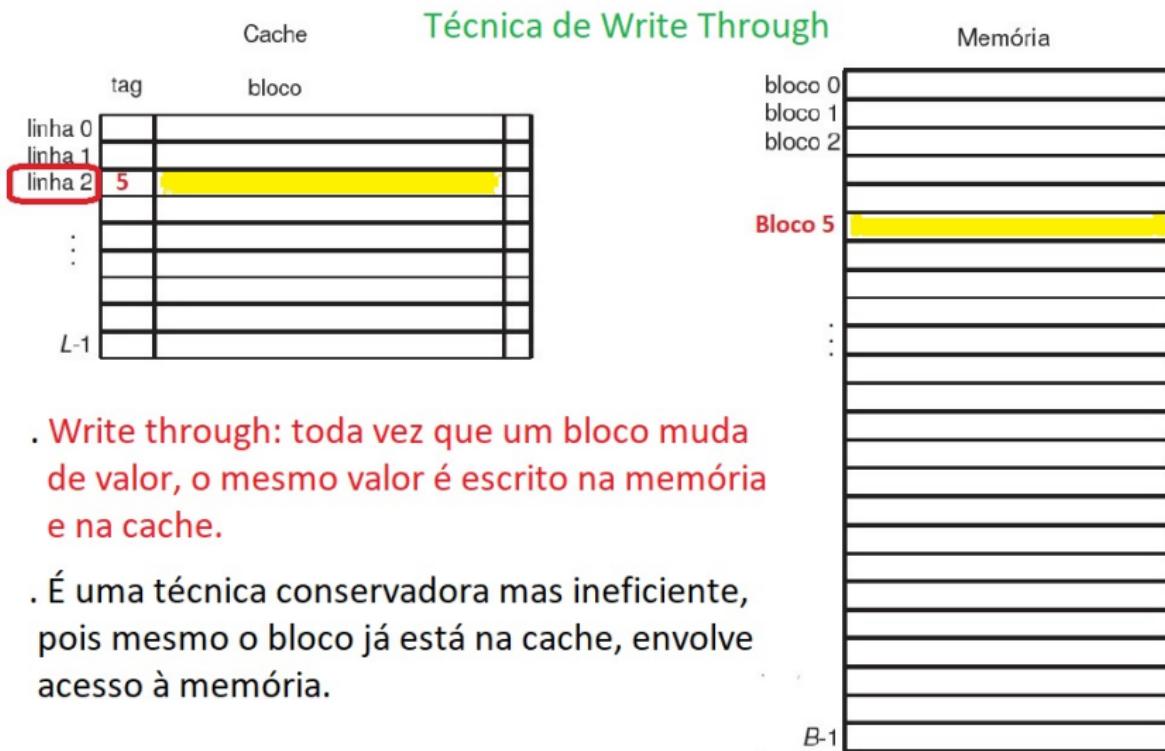
- Em cada linha da cache, mantém-se um *Use bit*.
- Quando um bloco de memória é carregado numa linha da cache, o *Use bit* é inicializado como zero.
- Quando a linha é referenciada, o *Use bit* muda para 1.
- Quando um novo bloco precisa ser carregado na cache, escolhe-se aquela linha cujo *Use bit* é zero.
- Na analogia de escolha de um produto menos procurado no supermercado para ceder o lugar a outro produto, *Use bit* = zero denota aquele com poeira.

# Cache write through e write back



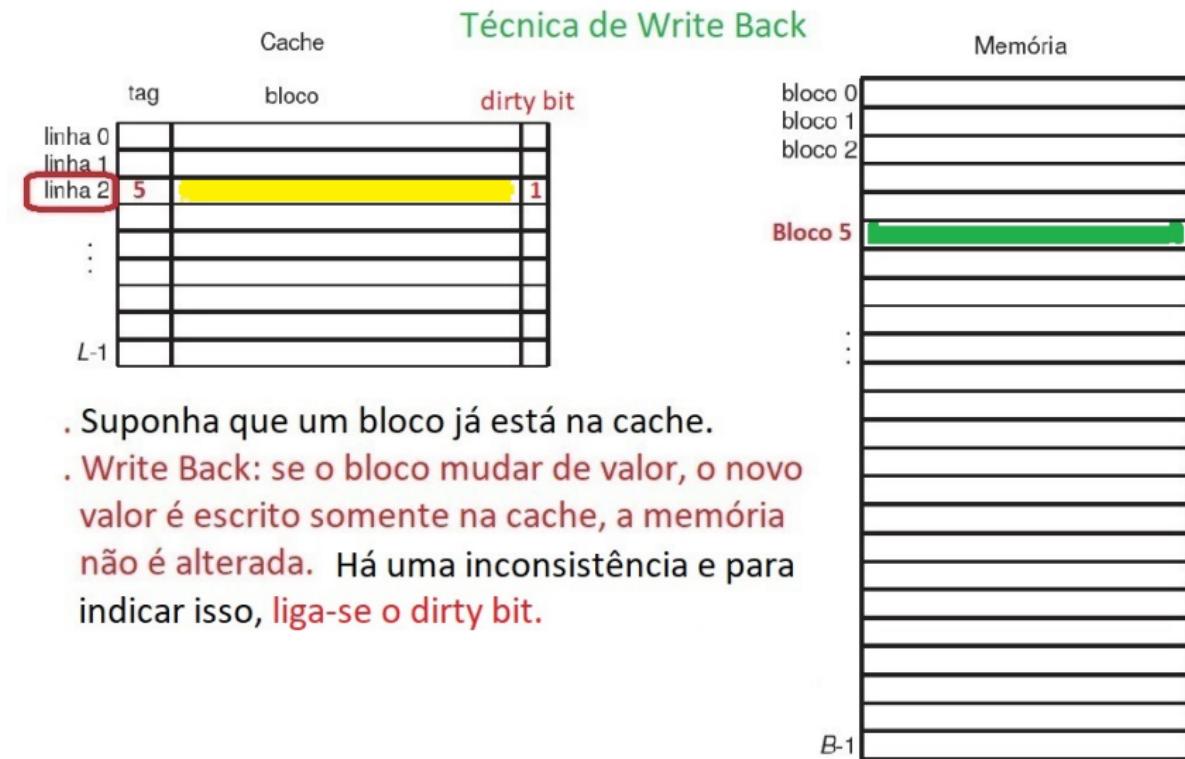
- . Suponha linha 2 da cache contém o bloco 5.
- . Se a cache está cheia e o alg. de substituição escolheu a linha 2 para ceder o seu lugar para outro bloco. A linha 2 pode ser alterada?
- . Sim, pois a memória contém o mesmo conteúdo e a cache pode ser alterada.
- . **Manter o mesmo conteúdo na cache e na memória é uma prática segura.**

# Cache write through e write back



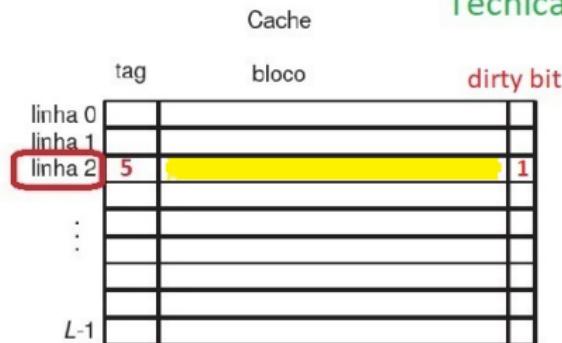
- . Write through: toda vez que um bloco muda de valor, o mesmo valor é escrito na memória e na cache.
- . É uma técnica conservadora mas ineficiente, pois mesmo o bloco já está na cache, envolve acesso à memória.

# Cache write through e write back



# Cache write through e write back

## Técnica de Write Back



- . Suponha que um bloco já está na cache.
- . Write Back: se o bloco mudar de valor, o novo valor é escrito somente na cache, a memória não é alterada. Há uma inconsistência e para indicar isso, liga-se o dirty bit.
- . Se a linha de cache vai ser alterada, então o conteúdo da cache é escrita na memória, e a consistência é mantida.

# Cache *write through* e *write back*

Vamos resumir o que foi visto.

- Um bloco de memória foi colocada em uma linha da cache.
- O bloco de memória precisa mudar de valor.
- Como o bloco tem uma cópia na cache, como a alteração deve ser feita?
  - *Write through*: Toda escrita à cache é também feita à memória.
  - *Write back*: Escritas são somente feitas na cache. Mas toda vez que isso ocorre, liga-se um *dirty bit* indicando que a linha da cache foi alterada mas a memória ainda contém o valor antigo.
  - Quando a linha da cache é escolhida pelo algoritmo de substituição para ceder o lugar a outro bloco, antes que a linha da cache é substituída, se *dirty bit* está ligado, então o bloco da memória correspondente é atualizado, mantendo-se assim a consistência.

# Como foi o meu aprendizado?

Indique a(s) resposta(s) errada(s).

- 1 Com o avanço dos vários tipos de memórias, vai desaparecer por completo a hierarquia de memória.
- 2 A vantagem do mapeamento direto é a sua simplicidade.
- 3 Outra vantagem do mapeamento direto é o fenômeno de *thrashing*.
- 4 No mapeamento associativo, a linha que contém o bloco desejado é obtida rapidamente pois todos os tags são comparados em paralelo.
- 5 Outra vantagem do mapeamento associativo é a simplicidade na implementação da circuitaria para o acesso associativo.

# Como foi o meu aprendizado?

Indique a(s) resposta(s) errada(s).

- 1 O mapeamento associativo por conjunto reúne as vantagens do mapeamento direto e o mapeamento associativo.
- 2 O mapeamento associativo por conjunto é o mais usado em processadores modernos.
- 3 Para escolher a linha ótima para ser substituída, é comum executar-se um algoritmo de substituição ótimo em que todas as possíveis linhas são testadas como objeto de substituição.
- 4 Write-through é uma técnica mais conservadora para manter a consistência. Mas write-back minimiza a escrita na memória e ainda mantém a consistência embora tardiamente.

# Próximos assuntos: Meltdown/Spectre e depois Memória Interna



- Próximo assunto: Vulnerabilidades Meltdown e Spectre.
- Depois retomamos as aulas sobre Memória: Memória interna e código de detecção/correção de erros
- Todos tipos de memória são feitos de Silício.
- Veremos memória dinâmica e estática, memória volátil e não-volátil. Memória ROM, Memória Flash, etc.
- Há métodos de detectar erros de leitura de memória. Detectado o erro, a memória é lida de novo.
- O melhor ainda é o método que detecta e corrige o erro, sem precisar ler de novo (código de Hamming).

# Meltdown e Spectre

Instituto de Matemática e Estatística  
Departamento de Ciência da Computação  
Prof. Siang Wun Song

Slides em <https://www.ime.usp.br/~song>  
Material baseado em Meltdown and Spectre: <https://meltdownattack.com>



2019 IEEE Symposium on Security and Privacy

## Spectre Attacks: Exploiting Speculative Execution

Paul Kocher<sup>1</sup>, Jann Horn<sup>2</sup>, Anders Fogh<sup>3</sup>, Daniel Genkin<sup>4</sup>,  
Daniel Gruss<sup>5</sup>, Werner Haas<sup>6</sup>, Mike Hamburg<sup>7</sup>, Moritz Lipp<sup>5</sup>,  
Stefan Mangard<sup>5</sup>, Thomas Prescher<sup>6</sup>, Michael Schwartz<sup>5</sup>, Yuval Yarom<sup>8</sup>

<sup>1</sup> Independent ([www.paulkocher.com](http://www.paulkocher.com)), <sup>2</sup> Google Project Zero,

<sup>3</sup> G DATA Advanced Analytics, <sup>4</sup> University of Pennsylvania and University of Maryland,

<sup>5</sup> Graz University of Technology, <sup>6</sup> Cyberus Technology,

<sup>7</sup> Rambus, Cryptography Research Division, <sup>8</sup> University of Adelaide and Data61

- As vulnerabilidades Meltdown e Spectre foram descobertas independentemente e divulgadas em 2018/2019, por
  - Paul Kocher (Independent [www.paulkocher.com](http://www.paulkocher.com))
  - Jann Horn (Google's Project Zero)
  - Werner Hass, Thomas Prescher (Cyberus Technology)
  - Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwartz (Graz University of Technology)
  - Daniel Genkin (Univ. Pennsylvania and Univ. Maryland)
  - Mike Hamburg (Rambus)
  - Yuval Yarom (Univ. of Adelaide and Data61)

# Meltdown e Spectre



- Meltdown explora vulnerabilidades de hardware em processadores modernos como Intel x86, produzidos depois de 1995. Alguns modelos de AMD ARM também são afetados.
- No ataque Meltdown, um processo de usuário pode ler a memória do sistema (*kernel*) e de outros usuários.
- Meltdown rompe o mecanismo que impede aplicações em acessar memória do sistema, provocando e manipulando exceção (*segmentation fault*).

# Meltdown e Spectre



- Spectre afeta quase todos os processadores modernos dos últimos 20 anos.
- Spectre procura enganar aplicações em acessar posições arbitrárias da memória, inclusive de outros processos. Nenhuma exceção é causada. Tem diversas variantes. É mais difícil de consertar.
- *Spectre* explora a técnica de Execução Especulativa (*Speculative Execution*), e parece que vai assombrar por algum tempo, daí o nome.
- Os ataques independem do sistema operacional, e não dependem de qualquer vulnerabilidade de software.

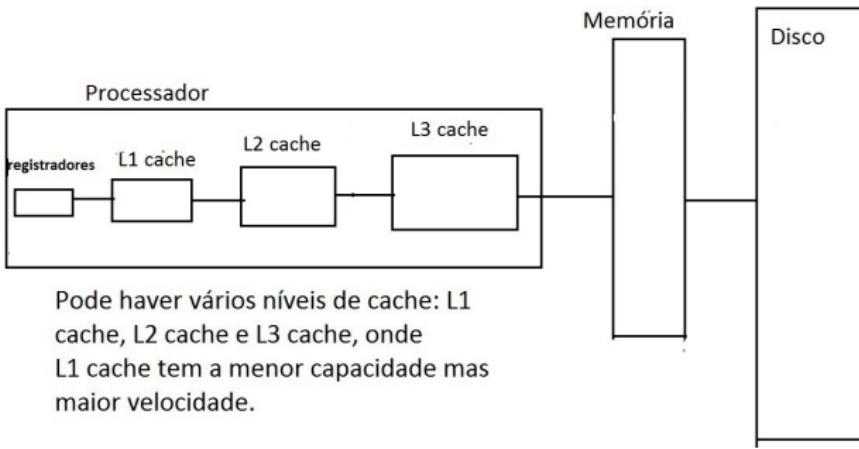
# Execução fora de ordem, execução especulativa, memória cache

- Processadores modernos usam diversas técnicas para obter maior desempenho.
- Cada técnica isoladamente pode ser considerada segura e imune a vulnerabilidades. Quando combinadas, podem gerar efeitos colaterais que podem ser explorados pelos chamados ataques por canal lateral (*side-channel attacks*).
- Técnicas exploradas por Meltdown:
  - Execução fora de ordem
  - Execução especulativa
  - Uso de memória cache
- Técnicas exploradas por Spectre:
  - Predição de desvio
  - Execução especulativa
  - Uso de memória cache

# Execução fora de ordem

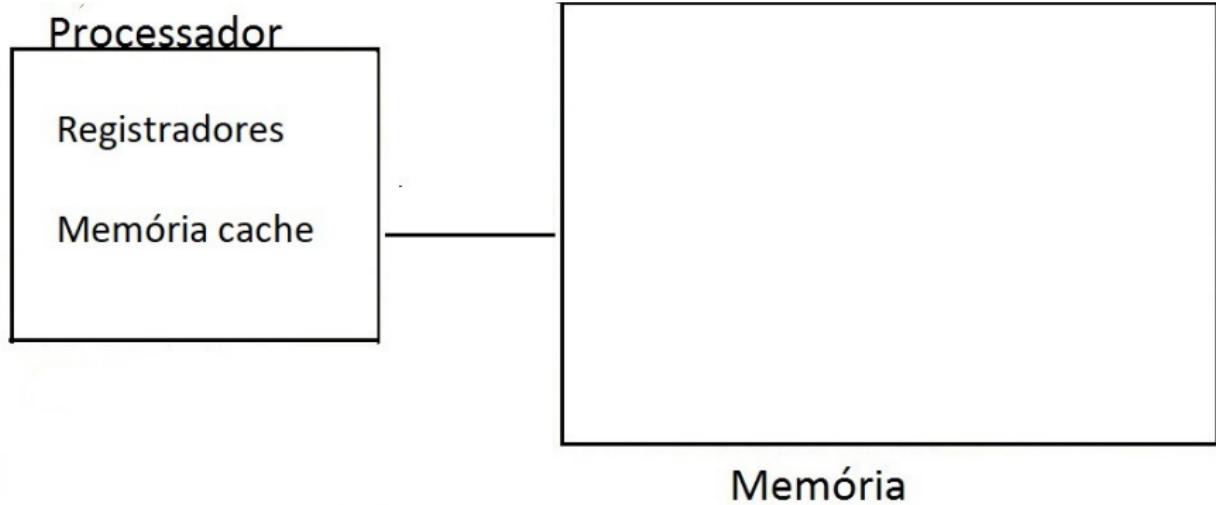
- Um núcleo (*core*) de uma CPU pode conter várias unidades para execução de instruções.
- Execução fora de ordem é uma técnica para maximizar os uso das unidades de execução.
- Ao invés de executar instruções estritamente em sequência, a CPU executa cada instrução assim que os recursos estão disponíveis.
- Enquanto a unidade de execução de uma instrução corrente está ocupada, outras unidades de execução podem se adiantar.
- O algoritmo de Tomasulo (implementado em hardware) escalona a execução de instruções fora de ordem.
- A CPU que implementa execução fora de ordem pode até executar instruções **antes da certeza se o resultado será salvo permanentemente ou se a instrução é necessária**.

# Memória cache



- Um dado (ou instrução) no disco pode ter uma cópia na memória e no processador (num registrador ou na memória cache).
- Quando o processador precisa de um dado, ele pode já estar na cache (**cache hit**). Se não (**cache miss**), tem que buscar na memória (ou até no disco).
- Quando um dado é acessado na memória, um bloco inteiro (tipicamente 64 bytes) contendo o dado é trazido à memória cache. Blocos vizinhos podem também ser acessados (**prefetching**) para uso futuro.
- Na próxima vez o dado (ou algum dado vizinho) é usado, já está na cache cujo acesso é rápido.

# Memória cache



- Se uma instrução ou dado já está no processador (registrador ou memória cache), o acesso é rápido. Senão tem que buscar na memória e é guardado na memória cache para uso futuro.

# Memória cache

Processador



Cache

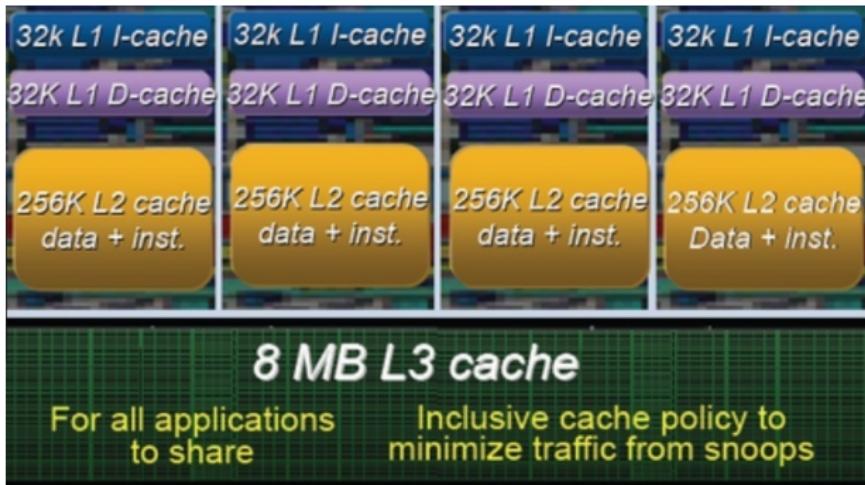


Memória

Images source: Wikimedia Commons

- Analoga: se falta ovo (dado) na cozinha (processador), vai ao supermercado (memória) e compra uma dúzia (*prefetching*), deixando na geladeira (cache) para próximo uso.

# Memória cache no Intel core i7



Note a diferença dos tempos de acesso quando o dado não está na cache.

Hierarquia memória	Latência em ciclos
registraror	1
L1 cache	4
L2 cache	11
L3 cache	39
Memória RAM	107
Memória virtual (disco)	milhões

# Predicção de desvio e execução especulativa

```
if    (condição)  then { comandos 1 }
                  else { comandos 2 }
```

- A avaliação da condição pode envolver dados que não estão na memória cache e precisam ser buscados na memória física (ou memória virtual). Isso pode envolver centenas de ciclos de relógio (ou até milhões de ciclos).
- Ao invés de ficar ocioso, o processador pode procurar prever qual ramo (**then** ou **else**) do desvio é mais provável para ser executado e já sai executando instruções deste ramo, salvando o estado dos registradores (*checkpoints*).
- Com escolha correta, reduz-se o tempo de execução. Se não, o processador desfaz o que foi feito e executa o ramo correto, que não é pior que ficar aguardando o resultado da condição.

# Predictor de desvio

```
if    (condição)  then { comandos 1 }
      else { comandos 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 alguma forma. Exemplo: Intel Core i7.

# Predicção de desvio e execução especulativa

```
if (condição)    then { comandos 1 }
                    else { comandos 2 }
```

Predicção de desvio e execução especulativa podem ser usadas na técnica de *pipelining*:



Source: Ford assembly line 1933 - Wikipedia



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

# Predicção de desvio e execução especulativa

```
if (x < array1_size) then { I6, I7, I8, I9, I10 }
else { J6, J7, J8, J9, J10 }
```

A predição de desvio e execução especulativa evitam, com a escolha correta, a descontinuidade no preenchimento da *pipeline*:

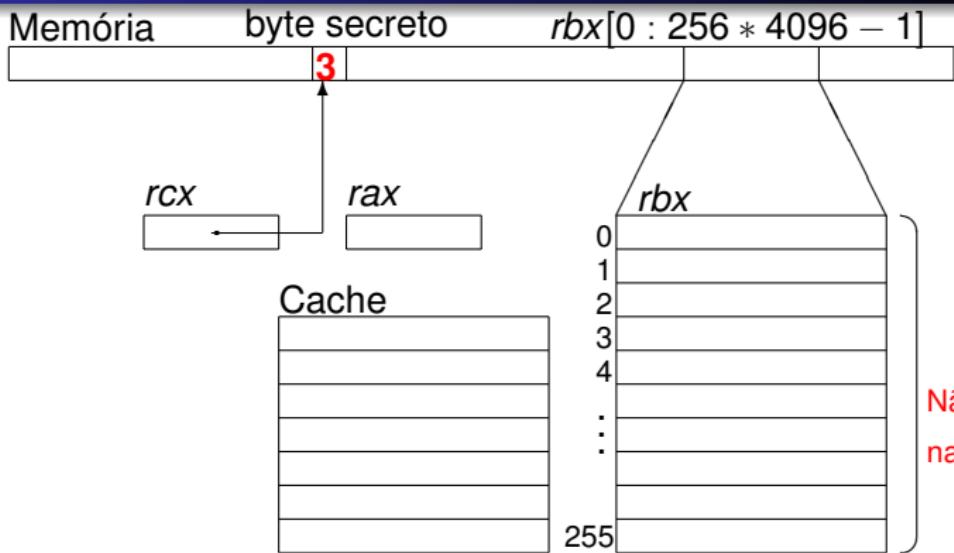
Ciclo	1	2	3	4	5	6	7	8	9	10
Busca instrução	I1	I2	I3	I4	if	I6	I7	I8	I9	I10
Decodificação		I1	I2	I3	I4	if	I6	I7	I8	I9
Endereço operando			I1	I2	I3	I4	if	I6	I7	I8
Busca operando				I1	I2	I3	I4	if	I6	I7
Execução					I1	I2	I3	I4	if	I6
Escrita resultado						I1	I2	I3	I4	if

- *Pipelining* de instruções foi implementado já no Intel 80486.
- O Pentium Pro já implementava as técnicas execução fora de ordem, predição de desvios e execução especulativa.

Apresentando ...

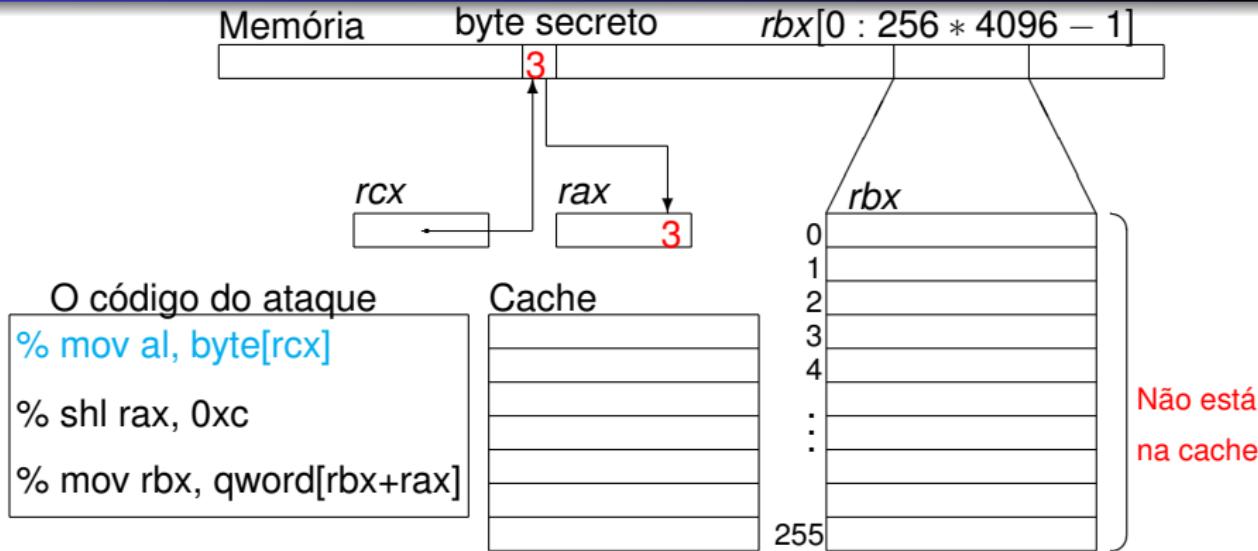


# Meltdown: preparar o ataque



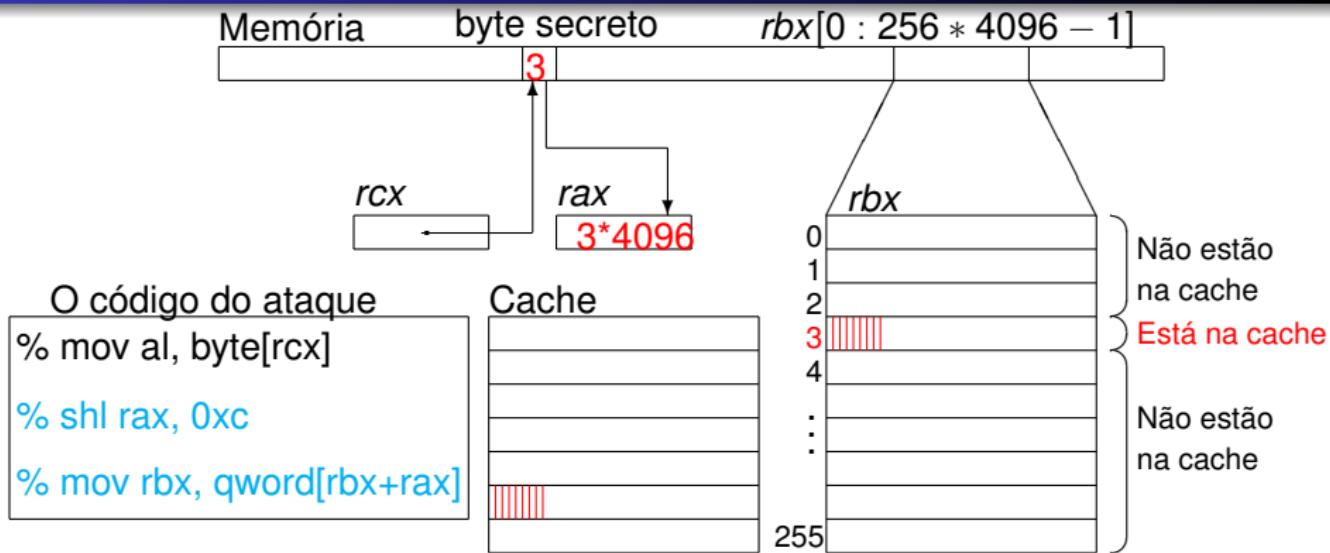
- Deseja-se ler o byte secreto com endereço em  $rcx$ .
- (O processo atacante não tem permissão para acessar este endereço.)
- Prepara-se um espaço de 1 Mbytes  $rbx = [0 : 256 * 4096 - 1]$ , que está representado na figura como um array de 256 linhas cada uma de 4 Kbytes.
- Deve-se garantir que esse espaço de 1 Mbytes **não está na memória cache**.

# Meltdown: o ataque - parte 1



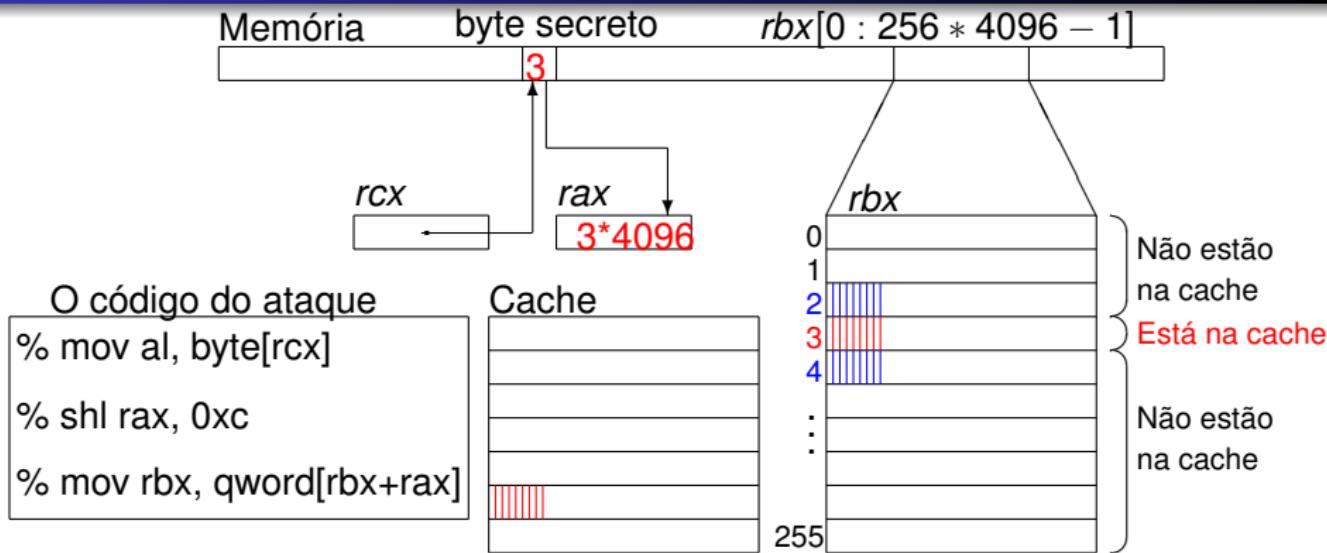
- A instrução em azul claro lê o byte secreto e coloca em al (a posição do byte menos significativo de *rax*).
- Ao mesmo tempo o sistema verifica se o acesso é permitido.
- Enquanto isso, outras instruções podem ser executadas (execução fora de ordem e especulativa).

# Meltdown: o ataque - parte 1



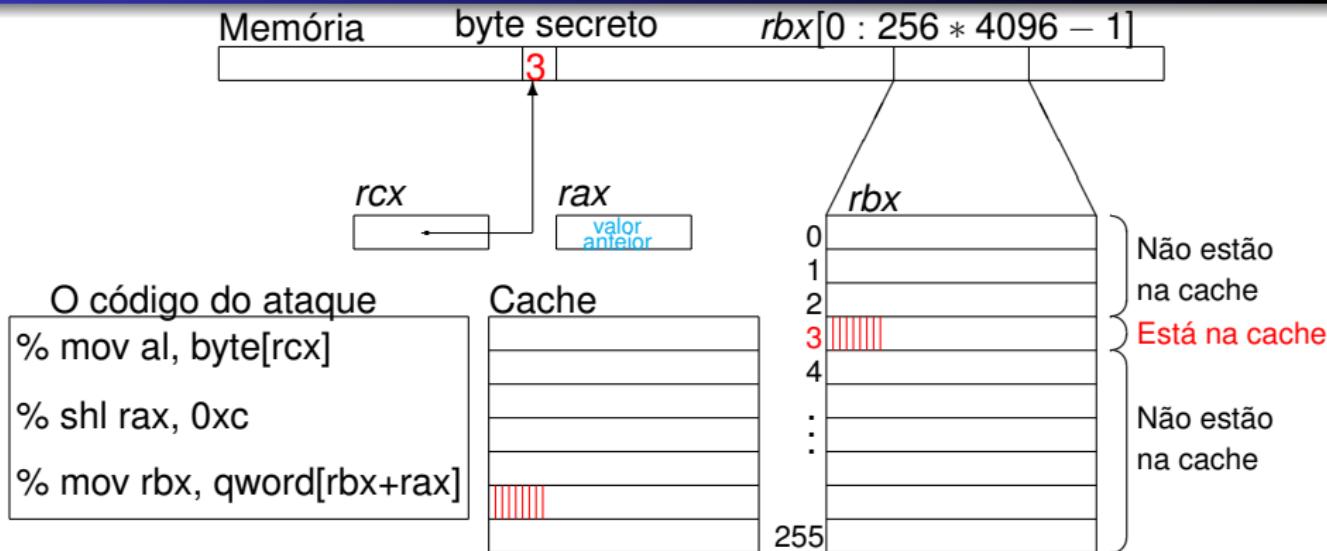
- As instruções em azul claro multiplicam o byte lido por 4096 (desloca  $rax$  12 bits para a esquerda), e usa esse endereço para acessar uma palavra (64 bits) em  $rbx[3 * 4096]$ .
- A palavra acessada em  $rbx[3 * 4096]$  vai para cache.

# Meltdown: o ataque - parte 1



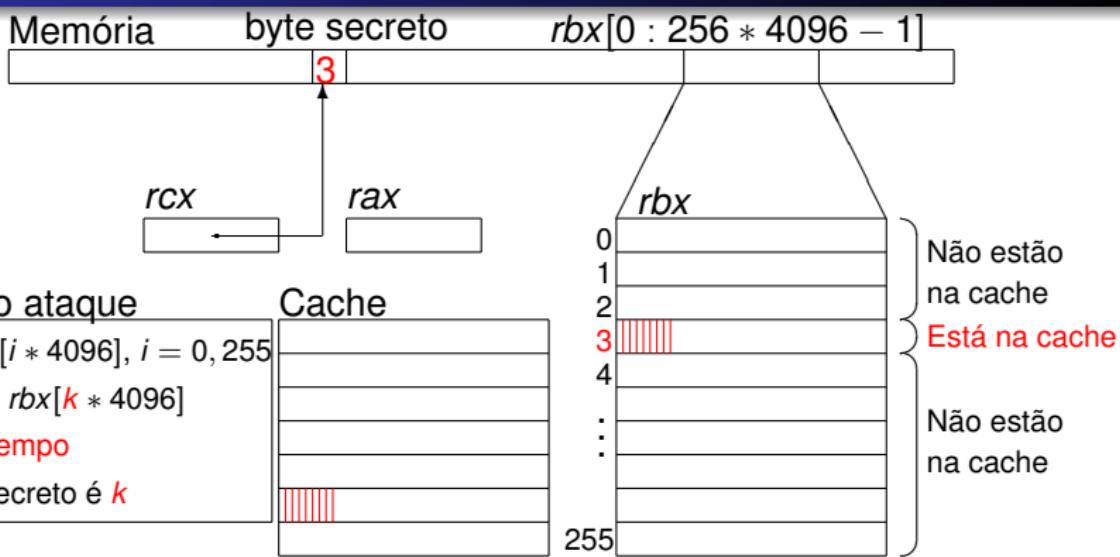
- Note que palavras “vizinhas”  $rbx[2 * 4096]$  e  $rbx[4 * 4096]$  não vão para cache mesmo com *prefetching*.
- Daí a razão de multiplicar o byte lido por 4096.

# Meltdown: o ataque - parte 1



- O sistema descobre que o processo não tem permissão para ler o endereço `rcx`. Desfaz então o que foi feito, eliminando os efeitos produzidos: `rax` volta a conter o **valor anterior** que tinham.
- Mas deixa um efeito colateral: **a palavra `rbx[3 * 4096]` continua na cache**.
- A leitura inválida causa uma exceção; o processo atacante seria suspenso. Mas há maneiras de evitar isso e o processo passa para a parte 2.

# Meltdown: o ataque por canal lateral - parte 2



- No exemplo, a palavra  $rbx[3 * 4096]$  está na cache e as demais palavras  $rbx[i * 4096]$ ,  $i \neq 3$ , não estão na cache.
- A leitura de  $rbx[3 * 4096]$  leva portanto menos tempo que as demais palavras. O byte secreto portanto é 3.

# Como evitar a suspensão do processo atacante

- Quando o sistema descobre que o processo não tem permissão para ler um endereço de memória, levanta-se uma exceção (*segmentation fault*) que causa a suspensão (*crash*) do processo. Há duas maneiras de evitar isso.
- Capturar o tratamento da exceção:
  - Cria (*fork*) um outro processo antes de acessar a memória inválida. Acessa a memória inválida com o processo filho. O processo pai é suspenso mas o processo pai continua o ataque.
  - Instalar um manipulador de exceção que é executado quando ocorre a exceção *segmentation fault*.
- Suprimir a exceção:
  - Colocar o trecho do ataque em um ramo de desvio condicional.  
**if** (condição) **then** {código de ataque};
  - Induzir o sistema a executar especulativamente o ramo errado (há maneiras de fazer isso) com uma condição falsa.
  - Ao constatar que executou o ramo executado, o sistema desfaz os efeitos e não levanta nenhuma exceção.

# Como aumentar o desempenho do ataque

O código do ataque

% mov al, byte[rcx]



Commando  
não permitido

% shl rax, 0xc

% mov rbx, qword[rbx+rax]

- O ataque se baseia na condição de corrida (*race condition*) entre:
  - O primeiro comando **% mov al, byte[rcx]** que acessa indevidamente um endereço proibido.
  - e os dois comandos em **azul claro** que acessam o byte secreto para deixar um vestígio ou pista para o ataque.
  - O ataque tem sucesso se os comandos em **azul claro** terminam **antes** de o primeiro comando levantar a exceção..

# Como aumentar o desempenho do ataque

O código do ataque

```
% mov al, byte[rcx]
% shl rax, 0xc
% mov rbx, qword[rbx+rax]
```

Commando  
não permitido

Acessa  
array rbx

- Para aumentar a chance de sucesso do ataque, pode-se fazer o seguinte
  - Os comandos em azul claro acessam o array *rbx*.
  - Em memória virtual, tabelas de páginas associam uma dada página de disco a um bloco de memória.
  - Para acesso rápido ao array *rbx*, coloca-se a tabela de páginas do array *rbx* em TLB (*Translation Lookahead Buffer*) - uma cache que armazena tabela de páginas.

# Meltdown em ação

```
meltdown@meltdown: ./meltdown
e0ld8110: 61 78 20 6f 72 20 73 74 61 74 65 20 6d 61 63 68 |ax or state mach|
e0ld8120: 69 6e 65 2c 20 69 74 20 69 73 20 62 65 69 6e 67 |ine, it is being|
e0ld8130: 20 75 73 65 64 20 77 69 74 68 20 61 75 74 68 6f | used with autho|
e0ld8140: 72 69 7a 61 74 69 6f 6e 20 66 72 6f 6d 0a 20 53 |rization from. S|
e0ld8150: 69 6c 69 63 6f 6e 20 47 72 61 70 68 69 63 73 2c |ilicon Graphics,|
e0ld8160: 20 49 6e 63 2e 20 20 48 6f 77 65 76 65 72 2c 20 | Inc. However,|
e0ld8170: 74 68 65 20 61 75 74 68 6f 72 73 20 6d 61 6b 65 |the authors make|
e0ld8180: 20 6e 6f 20 63 6c 61 69 6d 20 74 68 61 74 20 4d | no claim that M|
e0ld8190: 65 73 61 0a 20 69 73 20 69 6e 20 61 6e 79 20 77 |esa. is in any w|
e0ld81a0: 61 79 20 61 20 63 6f 6d 70 61 74 69 62 6c 65 20 |ay a compatible|
e0ld81b0: 72 65 70 6c 61 63 65 6d 65 6e 74 20 66 6f 72 20 |replacement for|
e0ld81c0: 4f 70 65 6e 47 4c 20 6f 72 20 61 73 73 6f 63 69 |OpenGL or associ|
e0ld81d0: 61 74 65 64 20 77 69 74 68 0a 20 53 69 6c 69 63 |ated with. Silic|
e0ld81e0: 6f 6e 20 47 72 61 70 68 69 63 73 2c 20 49 6e 63 |on Graphics, Inc|
e0ld81f0: 2e 0a 20 2e 0a 20 54 68 69 73 20 76 65 72 73 69 |... . This versi|
e0ld8200: 6f 6e 20 6f 66 20 4d 65 73 61 20 70 72 6f 76 69 |on of Mesa provi|
e0ld8210: 64 65 73 20 47 4c 58 20 61 6e 64 20 44 52 49 20 |des GLX and DRI |
e0ld8220: 63 61 70 61 62 69 6c 69 74 69 65 73 3a 20 69 74 |capabilities: it|
e0ld8230: 20 69 73 20 63 61 70 61 62 6c 65 20 6f 66 0a 20 | is capable of.|
e0ld8240: 62 6f 74 68 20 64 69 72 65 63 74 20 61 6e 64 20 |both direct and|
e0ld8250: 69 6e 64 69 72 65 63 74 20 72 65 6e 64 65 72 69 |indirect renderi|
e0ld8260: 6e 67 2e 20 20 46 6f 72 20 64 69 72 65 63 74 20 |ng. For direct|
```

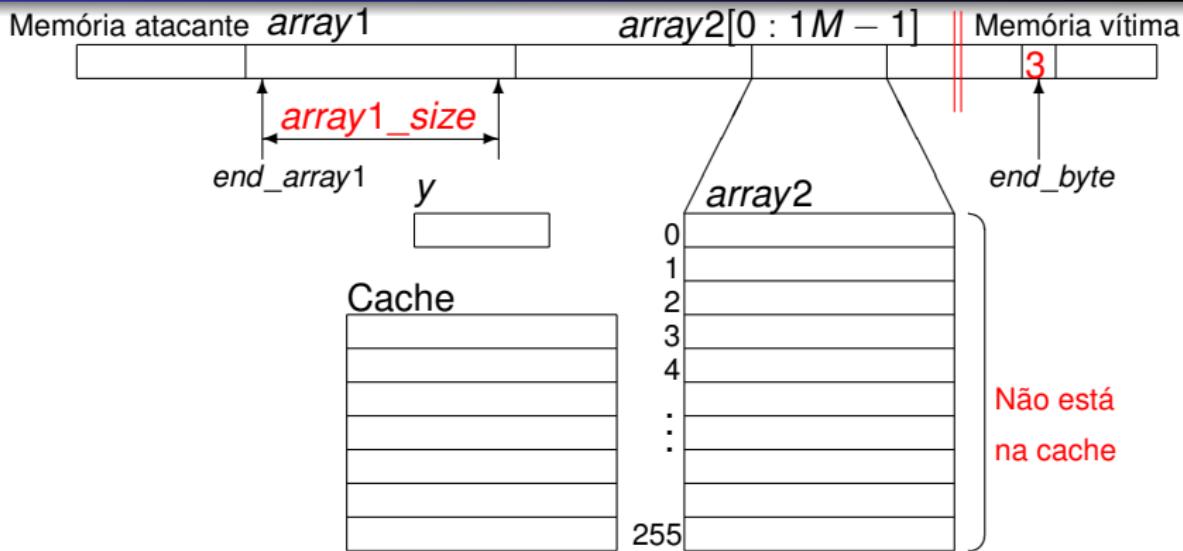
- O artigo de Moritz Lipp et al. menciona uma implementação de Meltdown que é capaz de despejar memória proibida a uma razão de 503 KB/s.
- Um [vídeo em Meltdown and Spectre](#) mostra a memória sendo acessada.

**Apresentando ...**



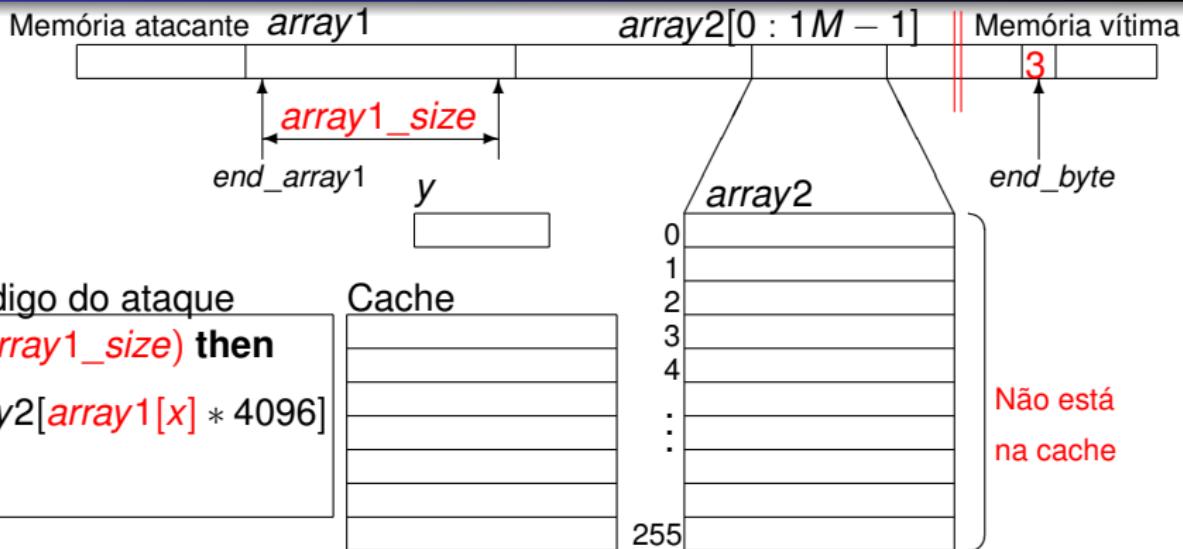
**SPECTRE**

# Spectre: preparar o ataque



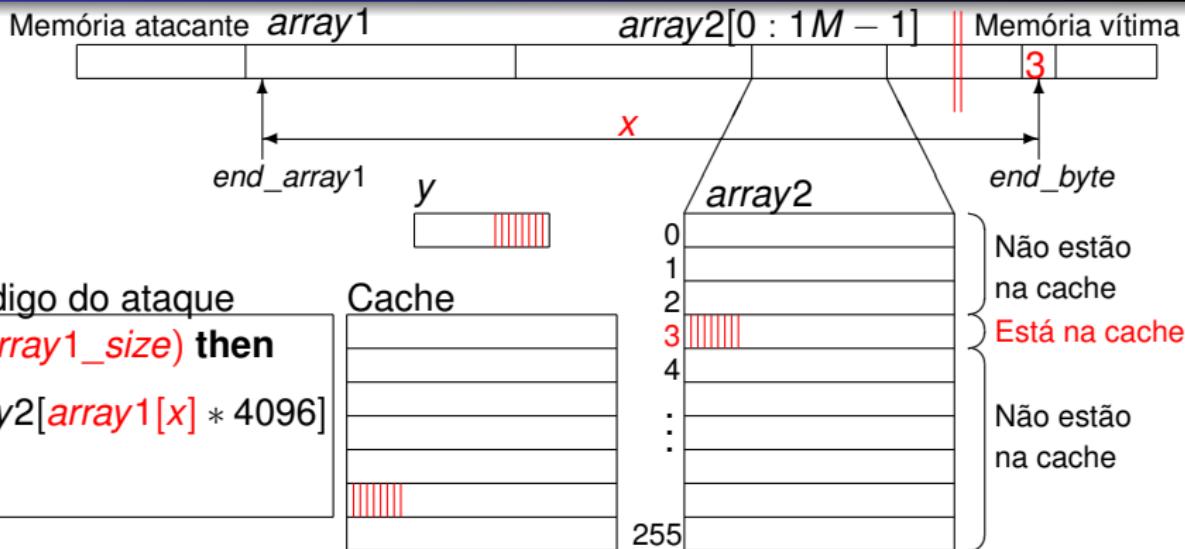
- Veremos uma variante de Spectre chamada *boundary check bypass*.
- O byte secreto está no endereço `end_byte` na memória do processo vítima.
- O processo atacante usa **um array de bytes** `array1` de tamanho `array1_size` e **um array de bytes** `array2` de tamanho 1 Mbytes ou  $256 \times 4096$  bytes.
- O array2 está representado na figura com 256 linhas cada uma de 4 Kbytes.
- Deve-se garantir que `array2` e `array1_size` não estão na memória cache.

# Spectre: o ataque - parte 1



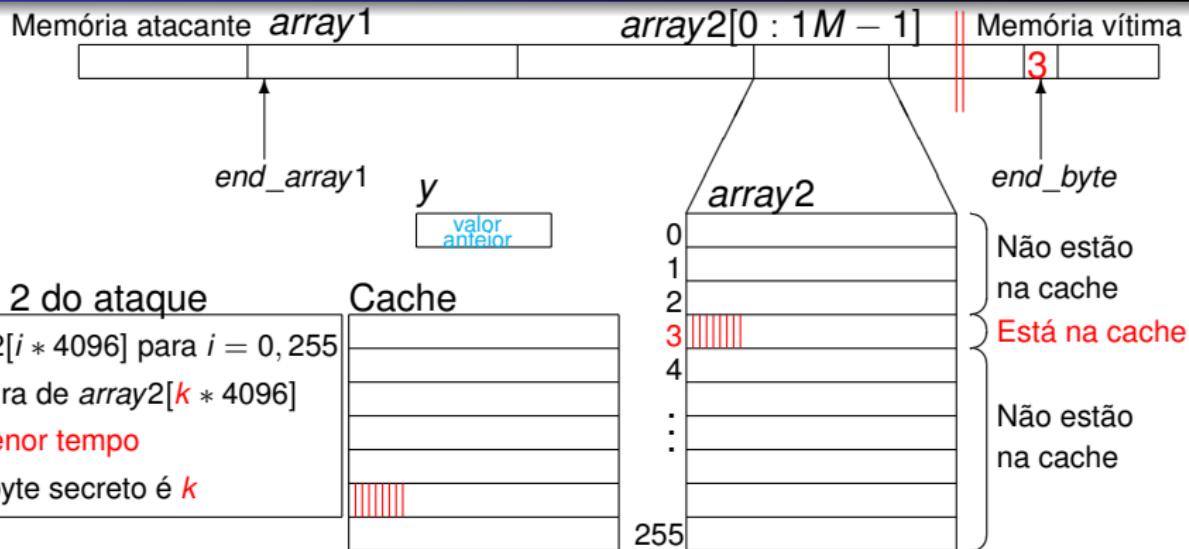
- O teste **if** garante que somente posições válidas de  $array1$  são acessadas. Mas pode levar tempo pois  $array1\_size$  não está na cache e tem que ser lida.
- Com predição de desvio, o processador pode especulativamente executar o ramo **then**:  $y = array2[array1[x] * 4096]$ .
- Assim, com escolha correta ganha-se tempo. Com escolha errada os efeitos do ramo executado são desfeitos.

# Spectre: o ataque - parte 1



- Primeiro induz o predictor de desvio a fazer uma escolha errada. Exemplo: usam-se valores de *x* válidos para o predictor preferir a execução de **then**.
- Para ler o byte secreto, faz-se  $x = end\_byte - end\_array1$ , a execução especulativa vai ler  $array1[x] = 3$  e calcular  $y = array2[3 * 4096]$ .
- O byte  $array2[3 * 4096]$  vai para a cache.

# Spectre: o ataque por canal lateral - parte 2



- Determinada a condição de desvio, o processador percebe que executou erroneamente o ramo **then**. Desfaz todos os efeitos produzidos e *y* continua com o **valor anterior**.
- Mas deixou um vestígio: byte  $array2[3 * 4096]$  continua na cache enquanto nenhum outro byte  $array2[i * 4096]$ ,  $i \neq 3$  está na cache.
- O ataque por canal lateral é idêntico ao de Meltdown.

# Spectre - uma palestra por Paul Kocher

The screenshot shows a video player interface. At the top, it says "40<sup>th</sup> IEEE Symposium on Security and Privacy". Below that is a thumbnail image of Paul Kocher speaking at a podium. The main title on the slide is "Spectre Attacks: Exploiting Speculative Execution". Below the title is the subtitle "IEEE Security & Privacy (May 20, 2019)". The slide features a small cartoon ghost icon. The video player has a progress bar showing "0:09 / 21:10". At the bottom, there are various control icons like play, volume, and settings, along with a "40" icon.

Paul Kocher: fundador de Cryptography Research Inc.

- . Recebeu 2019 Marconi Prize
- . Pioneiro em Side Channel Attacks
- . Descobriu vulnerabilidade de execução especulativa
- . Queria ser veterinário, mas depois de trabalhar com Martin Hellman (criador de criptografia de chave pública), mudou de área.

[Clicar aqui: Spectre Attacks Exploiting Speculative Execution \(21 min.\)](#)

- “Spectre Attacks: Exploiting Speculative Execution”: por Paul Kocher no 2019 IEEE Symposium on Security & Privacy.
- O palestrante menciona mais de 10 variantes de problemas explorando execução especulativa. Destaca o compromisso entre desempenho × segurança e que devemos cuidar mais da segurança, já que processadores já são rápidos.

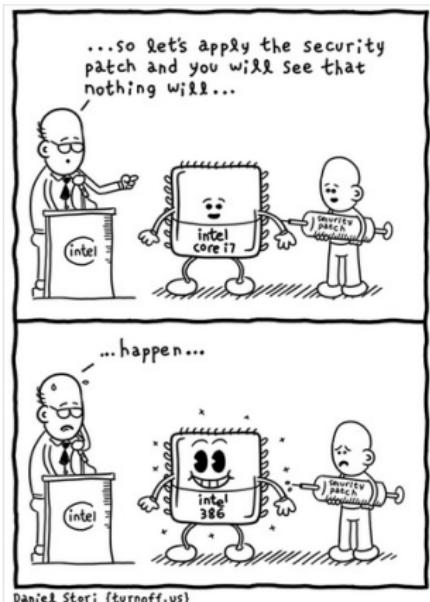


O antivirus pode detectar ou bloquear esse ataque?

- É difícil distinguir Meltdown e Spectre de aplicações benignas normais. Porém, depois que algum malware que usa esses ataques ficarem conhecidos, o antivirus pode detectar o malware comparando os códigos binários.

# Como consertar

- Há remendos (*patches*) contra Meltdown para Linux, Windows e OS X. Isso pode, entretanto, acarretar em uma **perda de desempenho (entre 17% a 23% mais lento)**.
- Spectre é uma classe de ataques. Há remendos mas não tem um simples remendo para todos.



# Enquanto isso ...

Novas vulnerabilidades estão sendo descobertas e divulgadas.



- *Foreshadow*: uma nova vulnerabilidade, semelhante a Meltdown e Spectre, divulgada em agosto de 2018. <https://foreshadowattack.eu>.
- Foreshadow é mais difícil de explorar mas, de acordo com especialistas, pode penetrar em áreas que nem Meltdown e Spectre conseguem. Foreshadow pode revelar informações sensíveis armazenadas em computadores pessoais e nuvem.
- Aplicar remendos em software pode aliviar o problema, mas compromete o desempenho.
- Contra esses ataques, Intel lançou em abril de 2019 uma nova geração de processadores denominados *Cascade Lake*.

# Referências bibliográficas



- O site [Meltdown and Spectre](#) contém muitas informações sobre essas vulnerabilidades.
- Sobre Meltdown: [Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. Meltdown, arXiv:1801.01207, January 2018, Cornell University Library.](#)
- Sobre Spectre: [Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. <https://spectreattack.com/spectre.pdf>](#)



# Obrigado!

# Meltdown e Spectre

Instituto de Matemática e Estatística  
Departamento de Ciência da Computação  
Prof. Siang Wun Song

Slides em <https://www.ime.usp.br/~song>  
Material baseado em Meltdown and Spectre: <https://meltdownattack.com>



# Memória interna e Código de Hamming

MAC0344 - Arquitetura de Computadores  
Prof. Siang Wun Song

Slides usados: <https://www.ime.usp.br/~song/mac344/slides05-memory.pdf>

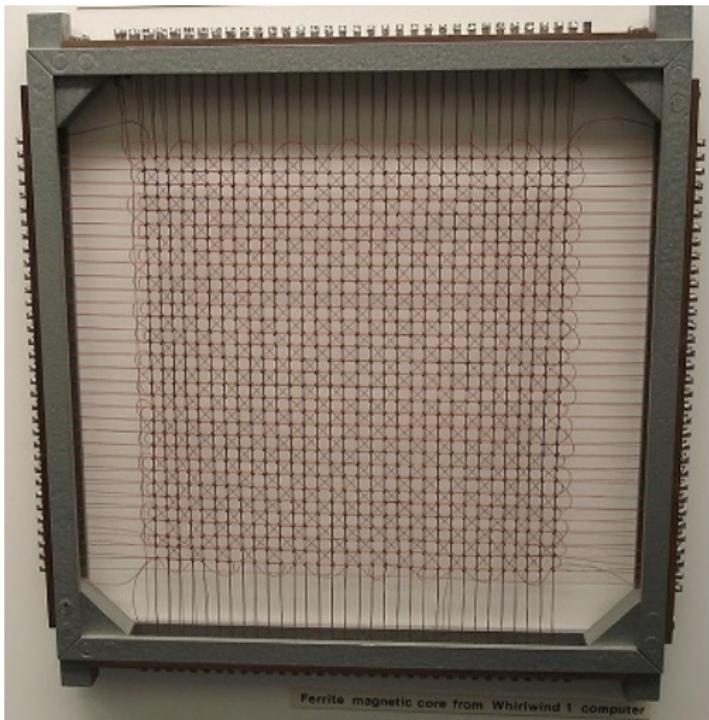
Baseado parcialmente em W. Stallings -  
Computer Organization and Architecture

# Memória interna e código de detecção/correção de erros

- Veremos Memória interna e código de detecção/correção de erros. Será passada a Lista 4 de exercícios.
- Ao final das aulas, vocês saberão
  - Todos os tipos de memória interna ou principal são feitos de Silício. As aulas sobre a Tecnologia VLSI - transistor MOS etc. - ajudam muito para entender esse assunto.
  - O que são memória dinâmica e estática, memória volátil e não-volátil. Memória ROM, Memória Flash, etc.
  - Há métodos de detectar erros de leitura de memória. Detectado o erro, a memória é lida de novo.
  - O melhor ainda é o método que detecta e corrige o erro, sem precisar ler de novo (código de Hamming).
  - Em particular, verão como esse método pode ser adaptado para corrigir erros de transmissão de grandes quantidades de dados entre dois pontos distantes.

# Memória magnética de núcleo de ferrite

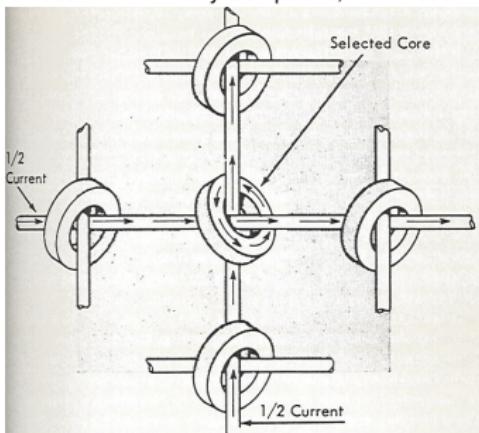
- Nos computadores antigos, a memória interna (RAM) era feita de núcleos de ferrite (*magnetic-core memory*). (Até hoje *core memory* é usado para indicar memória interna.)



Source: Science  
Museum, London

# Memória magnética de núcleo de ferrite

Source: IBM Early Computers, MIT Press

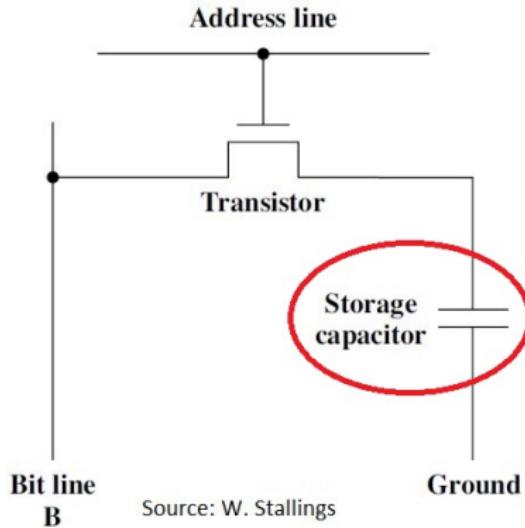


- A memória de núcleo de ferrite é do tipo **não-volátil**: o valor armazenado não se perde quando a energia é desligada e depois religada.

- A memória é feita de semi-condutor (Silício).
- Pode ser de dois tipos:
  - DRAM ou Dynamic RAM
  - SRAM ou Static RAM
- DRAM e SRAM são ambas **voláteis**: o conteúdo se perde quando o computador é desligado e depois religado.
- Conteúdo criado na memória precisa ser gravado no disco periodicamente, para não perder tudo se a energia cair.  
Algunas editores já fazem isso automaticamente.

# Memória interna DRAM - Dynamic RAM

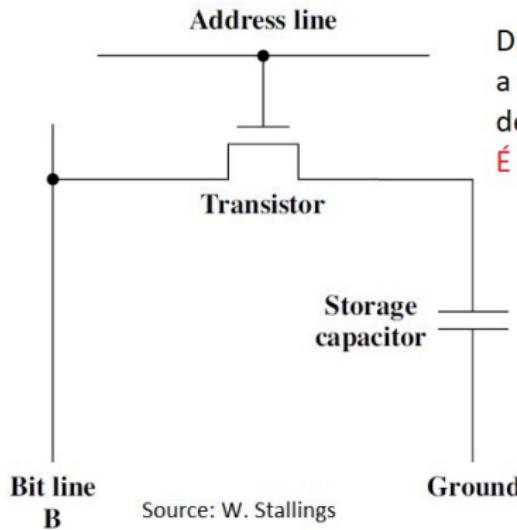
- **DRAM (Dynamic RAM)**: usada na memória principal.
- O **capacitor** com carga representa 1, senão representa 0.
- Quando carregado, a carga pode perder por vazamento.
- Para manter a carga de um capacitor que representa 1, aplica-se um pulso de **refrescamento** periodicamente.



Source: W. Stallings

# Memória interna DRAM - Dynamic RAM

- **DRAM (Dynamic RAM)**: usada na memória principal.
- O capacitor com carga representa 1, senão representa 0.
- Quando carregado, a carga pode perder por vazamento.
- Para manter a carga de um capacitor que representa 1, aplica-se um pulso de **refrescamento** periodicamente.



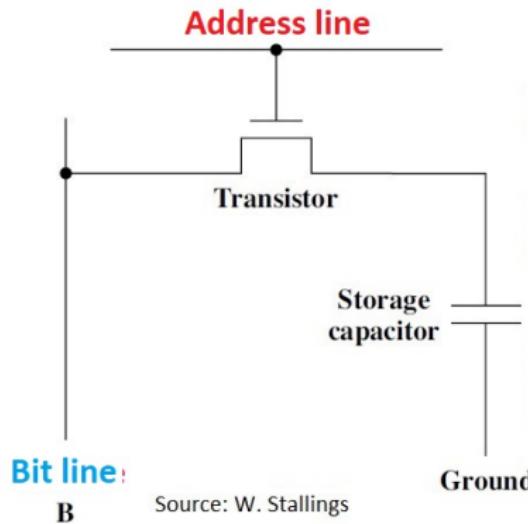
DRAM é **volátil** pois quando se desliga a eletricidade a carga se perde. Mas o pior é que, mesmo sem desligar, a carga também se perde por vazamento.  
É como um balde furado com água:



Balde furado vai perdendo água.  
Precisa sempre encher mais água:  
**Refrescamento**

# Memória interna DRAM - Dynamic RAM

- **DRAM (Dynamic RAM)**: usada na memória principal.
- O capacitor com carga representa 1, senão representa 0.
- Quando carregado, a carga pode perder por vazamento.
- Para manter a carga de um capacitor que representa 1, aplica-se um pulso de **refrescamento** periodicamente.



**Address line** controla o transistor para permitir acesso ao capacitor.

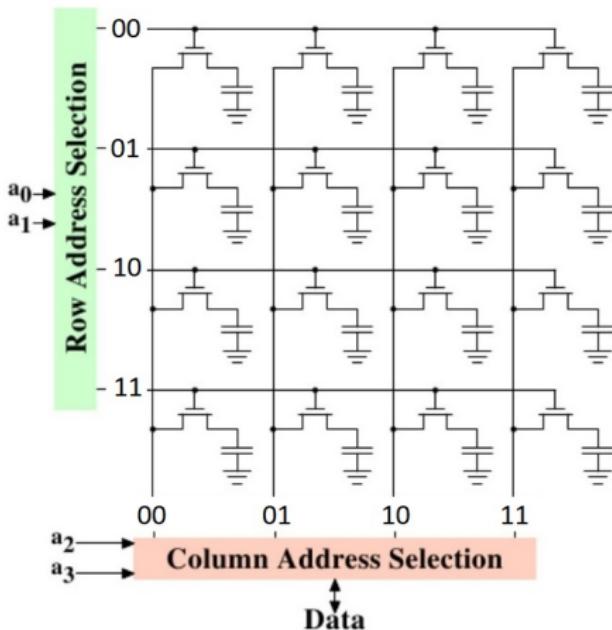
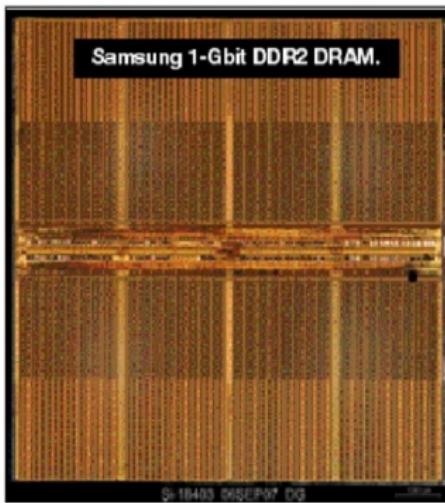
Usa-se voltagem alta no **Bit line** para escrever 1; voltagem baixa para 0.

Para ler, **Bit line** tira a carga do capacitor para ver se é 1 ou 0. A leitura descarrega o capacitor, cuja carga deve ser restaurada.

# Memória interna DRAM - Dynamic RAM

Samsung 1-Gbit DRAM. Note a disposição regular dos bits, permitindo uma maior densidade (mais bits por unidade de área do Silício). [Source: Samsung 1-Gbit DRAM.](#)

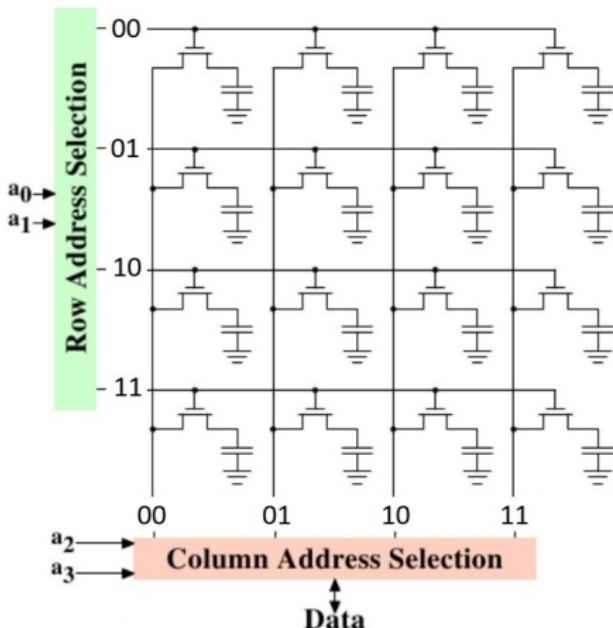
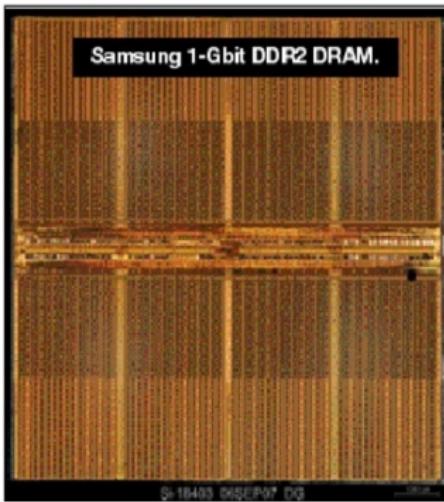
Modern DRAM chip with 8 internal memory banks.



# Memória interna DRAM - Dynamic RAM

Samsung 1-Gbit DRAM. Note a disposição regular dos bits, permitindo uma maior densidade (mais bits por unidade de área do Silício). [Source: Samsung 1-Gbit DRAM.](#)

Modern DRAM chip with 8 internal memory banks.

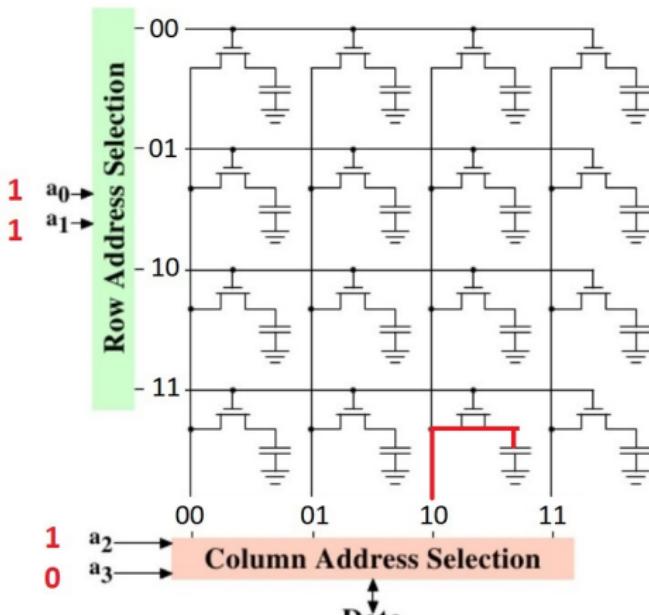
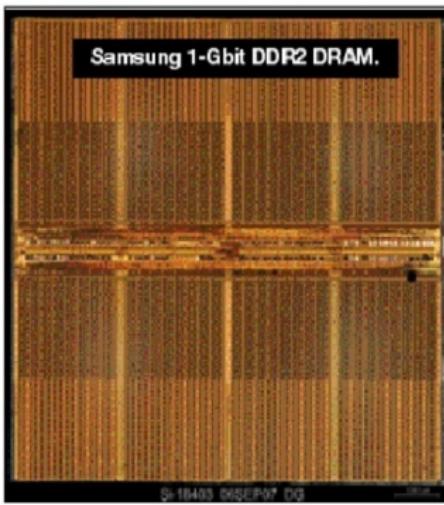


Queremos escrever 1 no endereço 1110

# Memória interna DRAM - Dynamic RAM

Samsung 1-Gbit DRAM. Note a disposição regular dos bits, permitindo uma maior densidade (mais bits por unidade de área do Silício). [Source: Samsung 1-Gbit DRAM.](#)

Modern DRAM chip with 8 internal memory banks.

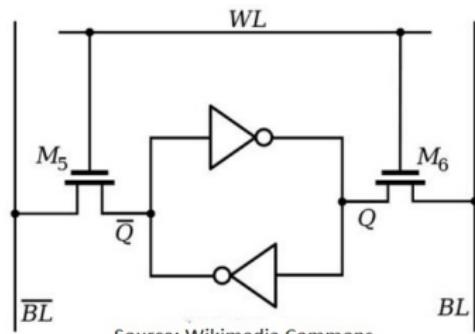


Queremos escrever 1 no endereço 1110

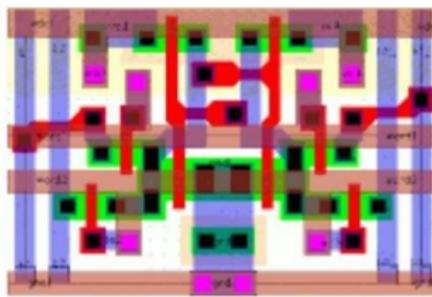
1

# Memória SRAM - Static RAM

- **SRAM (Static RAM)**: A memória estática mantém o dado inalterado, desde que haja energia. Não precisa de refreshamento.
- É usada na memória cache e registradores, sendo mais rápida, menos densa e mais custosa do que DRAM.
- Uma célula (um bit) SRAM pode ser implementada por duas portas inversoras (portas NÃO). A figura da direita mostra uma célula de um bit em CMOS.

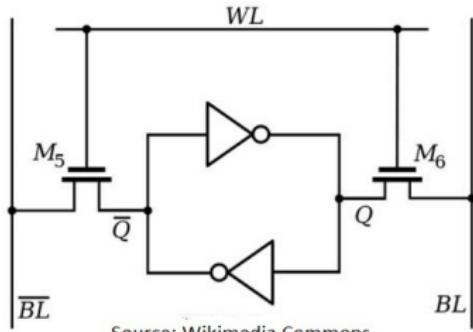


Source: Wikimedia Commons

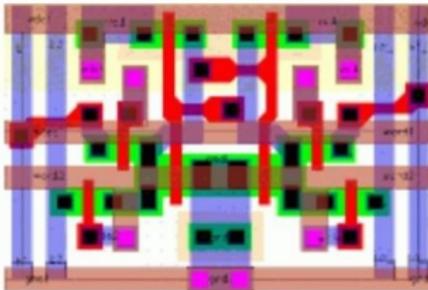


<https://iis-people.ee.ethz.ch/~kgf/aries/5.html>

# Memória SRAM - Static RAM



Source: Wikimedia Commons

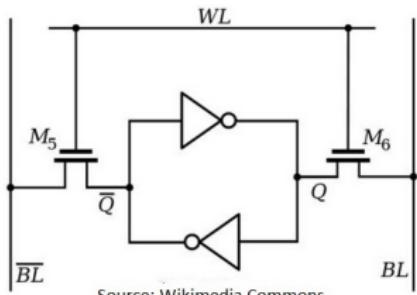


<https://iis-people.ee.ethz.ch/~kgf/aries/5.html>

- Há dois estados estáveis:
  - Memória contendo 1:  $Q = 1$  e  $\bar{Q} = 0$
  - Memória contendo 0:  $Q = 0$  e  $\bar{Q} = 1$
- Para ler: Linha  $WL$  alta liga transistores  $M5$  e  $M6$ :  
Valor  $Q$  é transmitido para  $BL$  (e valor  $\bar{Q}$  para  $\bar{BL}$ )
- Para escrever: Valor 1 ou 0 é colocado em  $BL$  e o complemento em  $\bar{BL}$ :  
Linha  $WL$  alta liga transistores  $M5$  e  $M6$

# Memória SRAM - Static RAM

Figura 1



Source: Wikimedia Commons

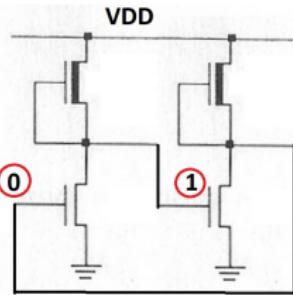
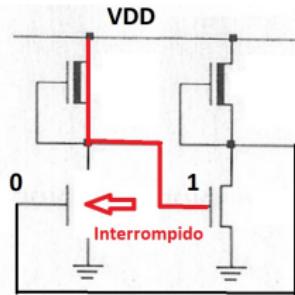


Figura 2



- Vejamos agora por quê duas portas inversoras assim podem garantir o valor 1 sem precisar de refreshamento.
- Figura 1: Porta da direita tem entrada **1**, carregando o capacitor.
- A sua saída vale **0**, que é conectado à entrada da porta da esquerda.
- Figura 2: Com a entrada igual a 0, o circuito na parte de baixo da porta da esquerda está interrompido. Assim, a corrente sai de VDD e alimenta o capacitor da direita. Portanto esse capacitor fica sempre carregado pela realimentação.

## Comparação entre DRAM e SRAM.

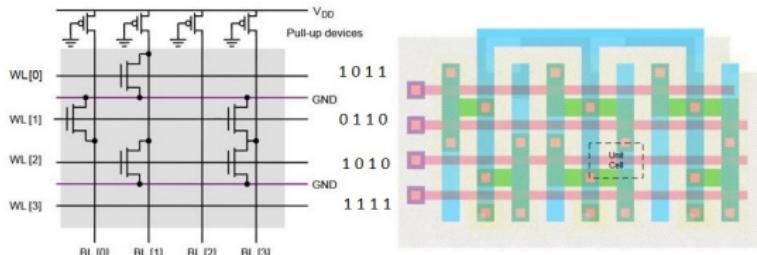
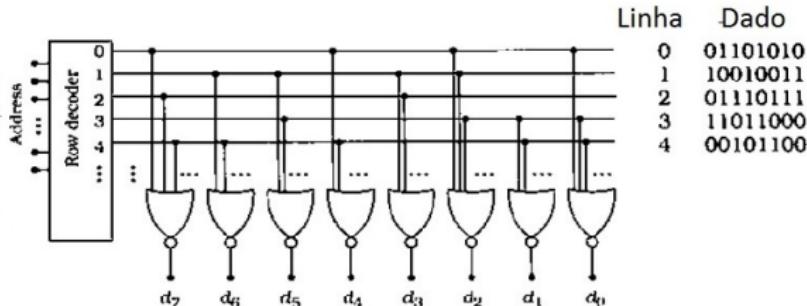
- Ambas são voláteis.
- A célula DRAM é mais simples e ocupa menos espaço que uma célula SRAM.
- Portanto DRAM é mais densa (mais células por unidade de área) e mais barata.
- Por outro lado, DRAM requer uma circuitaria de refrescamento. Para memórias grandes, esse custo fixo é mais que compensado pelo menor custo.
- Daí DRAM é preferida para memórias grandes e SRAM (que é um pouco mais rápida) é mais usada em memória cache.

# Tipos de ROM (*Read Only Memory*)

- ROM é uma memória cujo conteúdo é fixo, pré-gravado na fabricação, e não pode ser alterado.
- Há vários tipos de ROMs: todos são não-voláteis, i.e. não requerem energia para manter o seu conteúdo.
- Um importante uso de ROM é em processador CISC para armazenar o microprograma.
- ROM pode ser fabricada com portas NOR ou NAND, com um *layout* denso.
- Como ROM não pode ser alterada, erro de um bit pode acarretar em descartar um lote inteiro.

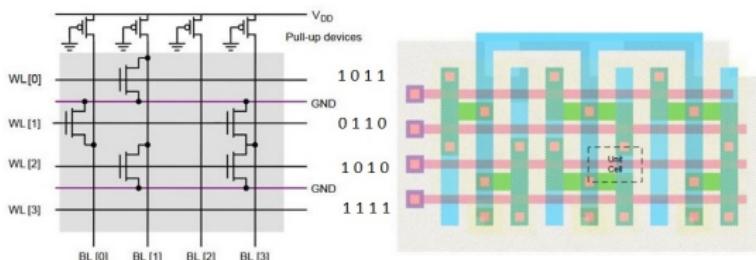
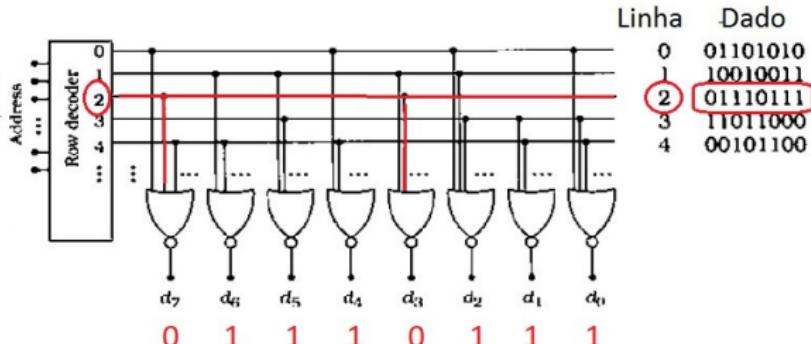
# ROM baseado em portas NOR

- Na ROM baseada em NOR, o endereço entra num decodificador e ativa uma das linhas de saída do decodificador: a linha ativada contém 1 e todas as demais 0.
- Se essa linha entra no NOR, a saída é 0, senão é 1.



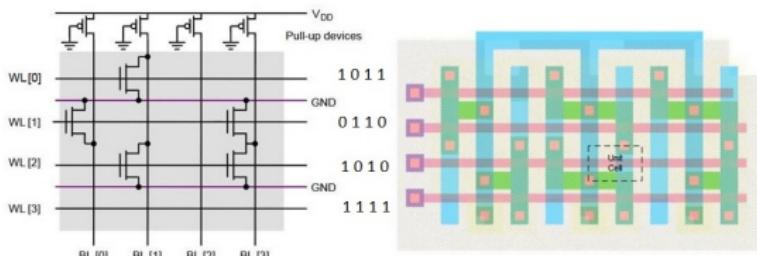
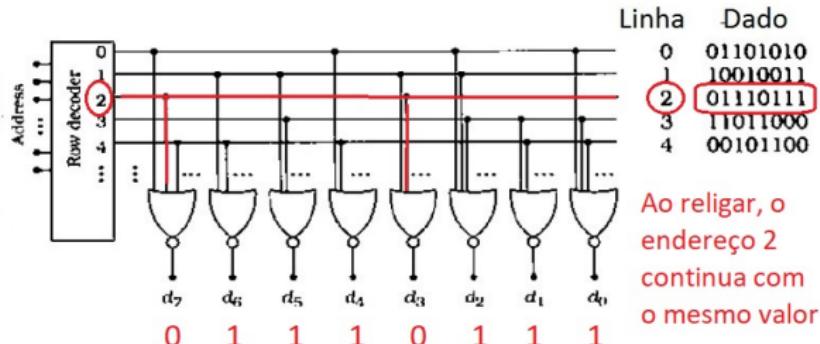
# ROM baseado em portas NOR

- Na ROM baseada em NOR, o endereço entra num decodificador e ativa uma das linhas de saída do decodificador: a linha ativada contém 1 e todas as demais 0.
- Se essa linha entra no NOR, a saída é 0, senão é 1.



# ROM baseado em portas NOR

- A ROM é **não-volátil**. Depois de desligar, ao religar os valores não se perdem. **Vocês podem explicar por quê?**
- É porque os valores estão codificados nas entradas das portas e não se perdem.

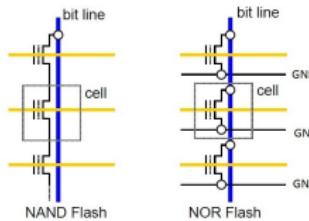


# Tipos de ROM (*Read Only Memory*)

- **PROM** (*Programmable ROM*): pode ser escrita uma só vez, por meio elétrico e pode ser feita depois de fabricada a pastilha.
- PROM oferece mais flexibilidade, mas ROM ainda é preferível para grandes quantidades.
- **EPROM** (*Erasable Programmable ROM*): leitura e escrita é como numa PROM. Porém, antes de gravar um novo conteúdo, toda a memória é apagada antes por meio de radiação ultra-violeta.
- EPROM é mais custosa.
- **EEPROM** (*Electrically Erasable ROM*): não é necessário apagar todo o conteúdo para atualização, apenas bytes selecionados são alterados. A escrita de uma EEPROM é demorada: centenas de micro-segundos por byte.
- EEPROM é mais custosa e menos densa.

# Memória flash ou *flash memory*

- **Flash memory** é uma memória intermediária entre EPROM e EEPROM, em custo e funcionalidade.
- Recebe o nome *flash* devido à velocidade com que pode ser alterada: uma memória flash por ser apagada em poucos segundos.
- É possível apagar blocos de memória, mas não no nível de byte.
- Dois tipos: NOR e NAND.
- Como EPROM, flash memory usa um transistor por bit, portanto é bastante densa.
- Solid State Drive ou SSD (“Disco de Estado Sólido”) usa a tecnologia de memória flash. Veremos SSD nas aulas sobre Memória Externa.



# Detecção e correção de erros de memória

0	0	1	1	0
---	---	---	---	---

Memória gravada

0	1	1	1	0
---	---	---	---	---

Memória lida

- Erros de leitura de memória podem ocorrer, por exemplo, por problemas de voltagem nas linhas ou radiações.
- Códigos de **detecção** e de **correção** são usados para detectar ou corrigir erros de memória.
  - Código de detecção de erro: o dado precisa ser lido novamente.
  - Código de correção de erro: o bit errado é corrigido. Não precisa ler de novo.

# Detecção e correção de erros de memória

- Preparo do código:
  - Para uma palavra original de  $M$  bits que queremos gravar na memória, calculamos  $K$  bits adicionais, obtidos em função dos  $M$  bits originais.
  - Forma-se um código de  $M + K$  bits. Esse código é gravado na memória.
- Detecção:
  - Para uma palavra de  $M$  bits, acrescentamos  $K = 1$  bit a mais, que depende dos  $M$  bits.
- Correção:
  - Para uma palavra de  $M$  bits, acrescentamos  $K$  bits a mais, onde  $K$  depende de  $M$ . Veremos isso.
  - Se há apenas um bit errado no código formado por  $M + K$  bits, então o método que veremos consegue dizer qual esse bit errado e pode corrigi-lo sem ter que ler de novo.

# Código de detecção de erro

Um **bit paridade** ( $K = 1$ ) é acrescentado a cada palavra original da memória de  $M$  bits. O código formado tem  $M + 1$  bits.



O bit paridade é escolhido de tal modo que o número de 1's do código formado ( $M + 1$  bits) é par.

Exemplo 1: A palavra contém um número par de 1's. Então o bit paridade vale **0**. Assim, o código (4 + 1 bits) contém um número par de 1's.



palavra

bit paridade

# Código de detecção de erro

Um **bit paridade** ( $K = 1$ ) é acrescentado a cada palavra original da memória de  $M$  bits. O código formado tem  $M + 1$  bits.



O bit paridade é escolhido de tal modo que o número de 1's do código formado ( $M + 1$  bits) é par.

Exemplo 2: A palavra contém um número ímpar de 1's. Então o bit paridade vale **1**. Assim, o código (4 + 1 bits) contém um número par de 1's.



# Código de detecção de erro

O bit paridade (paridade par) pode ser obtido fazendo o **ou-exclusivo** dos bits da palavra original.

Palavra =  $x_1 x_2 x_3 x_4$

o bit paridade  $x_5 = x_1 \oplus x_2 \oplus x_3 \oplus x_4$

onde  $\oplus$  representa a operação *ou-exclusivo*.

0	0	1	1	?
palavra			bit paridade	

o bit paridade  $x_5 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$

# Como detectar erro

Exemplo: Preparado o código ( $M + 1$  bits), o código é gravado na memória.

0	0	1	1	0
palavra			bit paridade	

Suponha que ao ler a memória, temos o seguinte caso.

0	1	1	1	0
palavra			bit paridade	

Calculamos a paridade do código lido: paridade **ímpar**: erro!

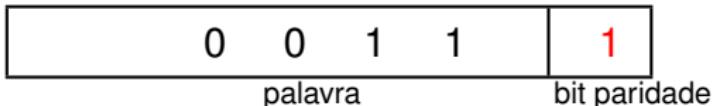
Tem que ler de novo.

# Como detectar erro

Exemplo: Preparado o código ( $M + 1$  bits), o código é gravado na memória.



Suponha que ao ler a memória, temos o seguinte caso.



Calculamos a paridade do código lido: paridade **ímpar**: erro!

Tem que ler de novo.

# Como detectar erro

Exemplo: Preparado o código ( $M + 1$  bits), o código é gravado na memória.

0	0	1	1	0
palavra			bit paridade	

Suponha que ao ler a memória, temos o seguinte caso.

1	1	1	1	0
palavra			bit paridade	

Paridade deu **par**: OK? O método não detecta erro quando há um número par de bits errados.

# Código de deteção de erro

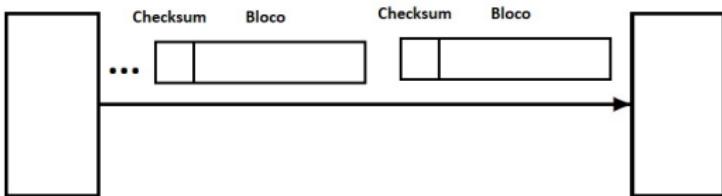


Source: Wikipedia

- Uso de bit paridade para detectar erro é um método simples.
- É usado em fitas magnéticas. A cada bloco de dados, um bit paridade é gravado.
- Na leitura, ao detectar um erro de paridade, o bloco de dados precisa ser lido de novo.

# Erros de transmissão entre 2 computadores

Comunicação entre 2 computadores



- Um esquema semelhante é usado em comunicação de dados entre 2 computadores.
- Os dados são organizados em blocos. Depois de cada bloco é introduzido um *checksum* que é função dos dados do bloco.
- Para cada bloco recebido, é recalculado o *checksum* usando a mesma função.
- O *checksum* calculado é comparado com o *checksum* recebido. Se diferente, então o bloco deve ser retransmitido.

# Como foi o meu aprendizado?

Vamos fazer um exercício juntos.

- O método de usar um bit de paridade sempre funciona?  
Isto é, o método sempre detecta quando há bits errados?
- Se a sua resposta é não, então em que situação o método funciona? E em que situação o método não funciona?

# Código de correção de erro - código de Hamming



Richard Hamming

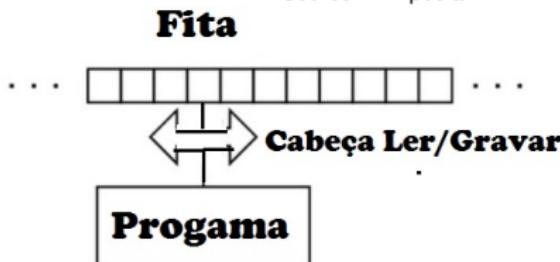
Source: Wikipedia

*Hamming queria fazer Engenharia, mas não tinha recursos. Acabou fazendo Matemática pois conseguiu uma bolsa na Universidade de Chicago, onde não havia curso de engenharia. Fez depois mestrado e doutorado em Matemática. Trabalhou na Bell Labs e inventou o famoso código de Hamming. Não se arrependeu de ter feito Matemática, pois o profundo conhecimento teórico o ajudou a resolver um problema de pesquisa de vanguarda: se o computador sabe detectar um erro de memória, por que não pode corrigí-lo? Hamming recebeu o **Turing Award** em 1968.*

# Turing Award - em homenagem a Alan Turing



Source: Wikipedia



- Alan Mathison Turing (1912 - 1954)
- Inglês, matemático, cientista da computação, cripto-analista. Considerado fundador da Teoria da Computação.
- Máquina de Turing, Computabilidade, Indecibilidade (Problema da Parada)
- Turing Award: prêmio anual criado em 1966, considerado o Prêmio Nobel da Computação. Em 2014, o prêmio aumentou para US\$ 1 milhão.

# Código de correção de erro - código de Hamming

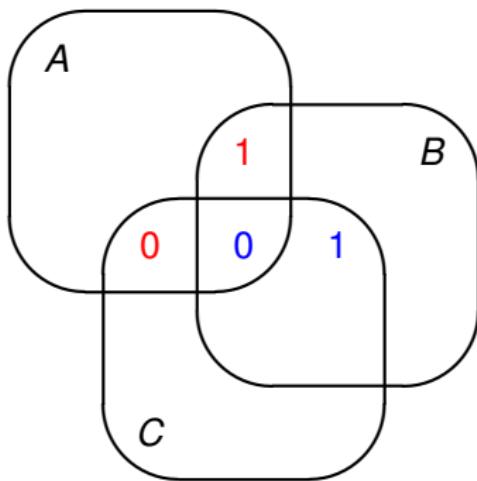
- O código de deteção pode detectar erro, mas não se sabe qual bit está erro. O dado precisa ser lido de novo ou retransmitido no caso de transmissão de dados.
- O **código de Hamming** é um **código de correção** que sabe qual bit errado, quando há **apenas 1 bit errado**. Assim é possível corrigi-lo.
- É usado para corrigir erro de memória.
- Para erros de disco RAID, é usado um código de Hamming estendido que é capaz de corrigir erro de 1 bit e detecção de erros em 2 bits.
- Veremos como usar o código de Hamming para erros em comunicação de dados entre computadores.

Vamos começar com um exemplo simples.

Seja uma palavra original de  $M = 4$  bits. Vamos acrescentar nesse caso mais  $K = 3$  bits adicionais.

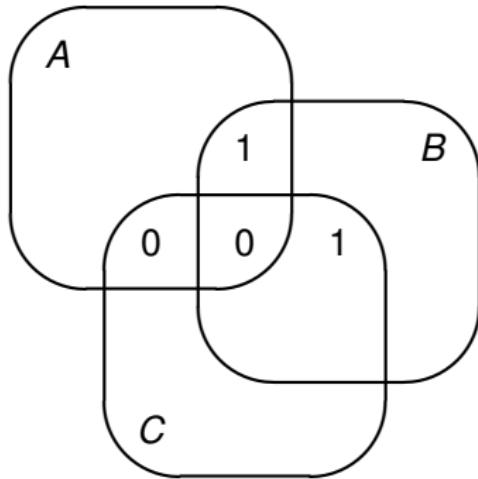
Nos próximos slides, ilustramos o método com diagrama **apenas para fim didático**, no caso específico de palavra de 4 bits. O método com diagrama **não serve** para o caso geral em que a palavra possui mais bits. Mostramos como tratar do caso geral mais tarde.

# Palavra de $M = 4$ bits e $K = 3$ bits adicionais



- Seja a palavra dada **1010**. Vamos colocar esses bits na figura acima assim:
- **1** =  $(A \cap B - C)$     **0** =  $(A \cap C - B)$   
**1** =  $(B \cap C - A)$     **0** =  $(A \cap B \cap C)$

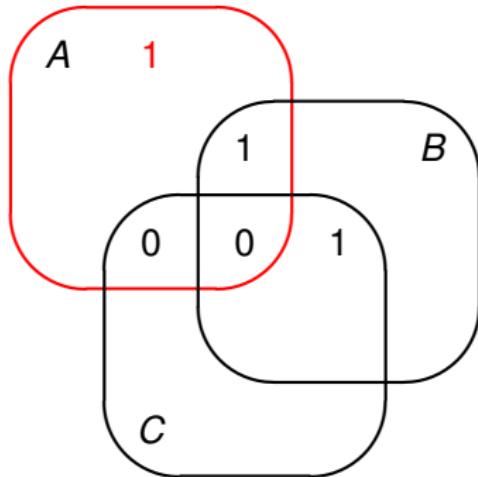
# Como obter os $K = 3$ bits adicionais



- Vamos acrescentar um bit de paridade em cada uma das 3 regiões vazias acima para dar paridade par em A, B, e C.
- Os 7 bits (4 da palavra original e 3 adicionais) vão formar o **código de Hamming**. Vejamos.

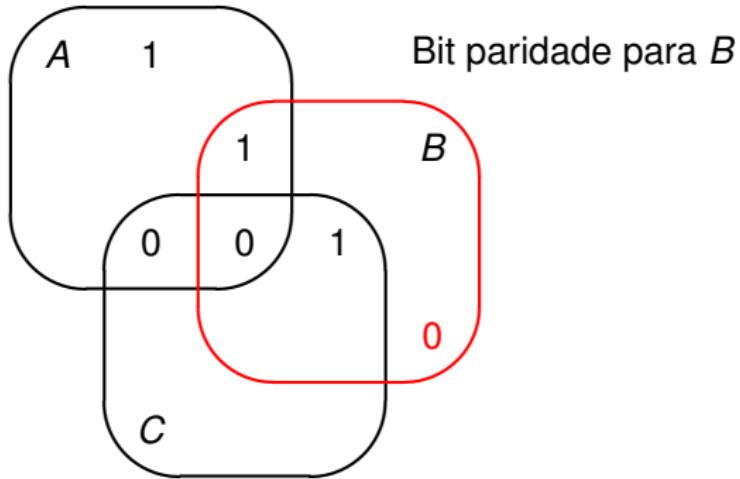
# Como obter os $K = 3$ bits adicionais

Bit paridade para  $A$



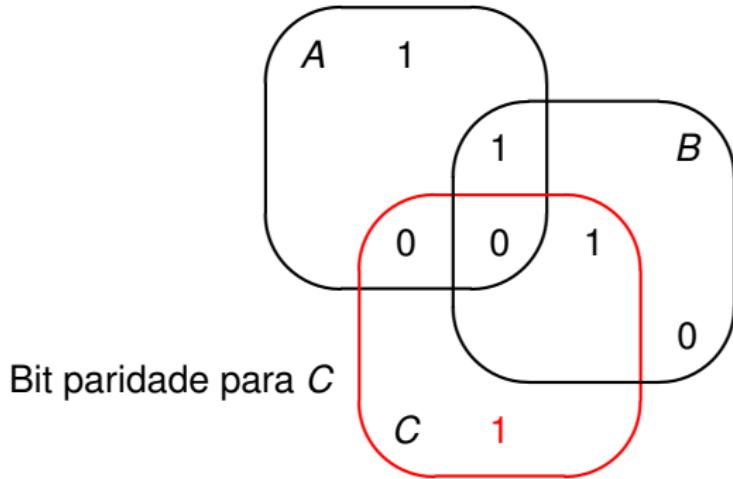
- Acrescentamos um bit de paridade em cada uma das 3 regiões vazias acima para dar paridade par em  $A$ ,  $B$ , e  $C$ .
- Os 7 bits (4 da palavra original e 3 adicionais) formam o **código de Hamming**.

# Como obter os $K = 3$ bits adicionais



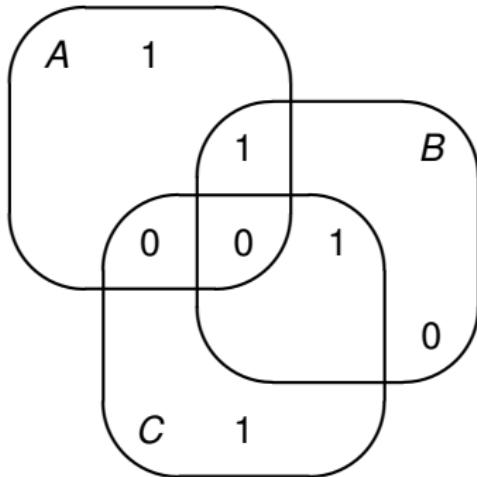
- Acrescentamos um bit de paridade em cada uma das 3 regiões vazias acima para dar paridade par em  $A$ ,  $B$ , e  $C$ .
- Os 7 bits (4 da palavra original e 3 adicionais) formam o **código de Hamming**.

# Como obter os $K = 3$ bits adicionais



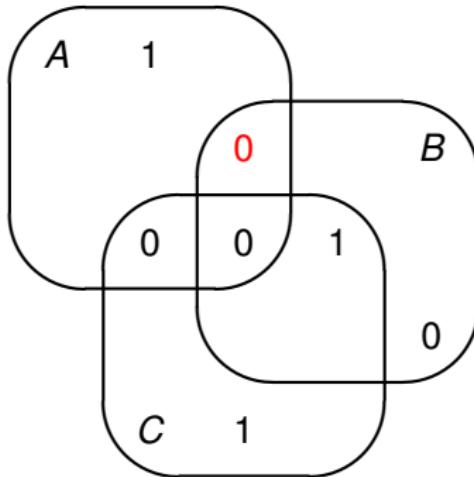
- Acrescentamos um bit de paridade em cada uma das 3 regiões vazias acima para dar paridade par em A, B, e C.
- Os 7 bits (4 da palavra original e 3 adicionais) formam o **código de Hamming**.

# Como obter os $K = 3$ bits adicionais



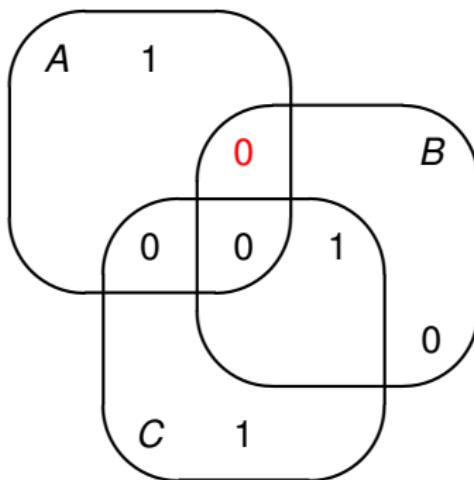
- O código de Hamming obtido (com  $M + K$  bits) é gravado na memória.
- Vamos ler esse código e supor que no máximo um bit lido errado. Vamos mostrar como saber qual o bit lido errado.

# Erro em 1 bit da palavra original



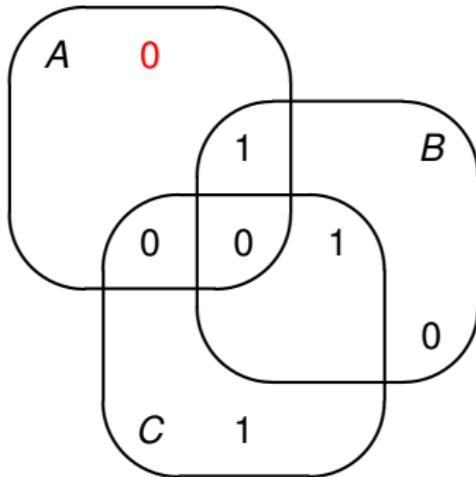
- Erro de 1 bit na palavra original pode ser localizado e corrigido.
- Tal erro pode ser detectado de modo simples, como se segue.

# Erro em 1 bit da palavra original



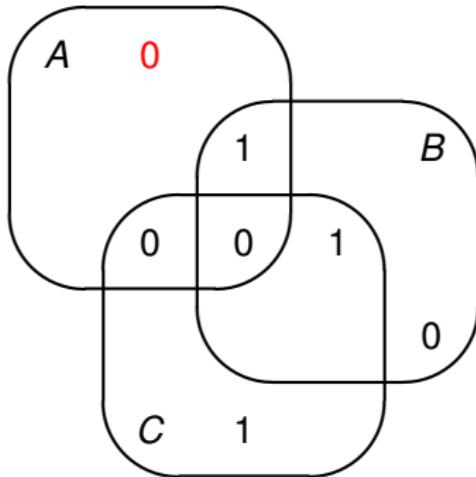
- Calculamos os bits de paridade:  
Região *A*: paridade **errada**. Região *B*: paridade **errada**.
- Região *C*: paridade OK. Temos **2 paridades erradas**. Logo região *AB* errada e o bit de ***AB*** deve ser 1.

# Erro em 1 dos $K$ bits adicionais



- Qualquer um dos 7 bits pode estar errado, por exemplo, o erro pode ser de um dos  $K$  bits adicionais.
- Tal erro pode ser detectado e corrigido como no caso anterior, como se segue.

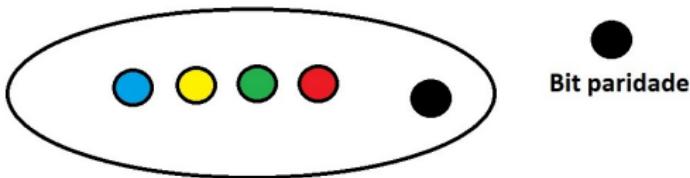
# Erro em 1 dos $K$ bits adicionais



- Calculamos os bits de paridade:  
Região *A*: paridade **errada**. Região *B*: paridade OK.
- Região *C*: paridade OK. Temos **1 paridade errada**. Logo região *A* errada e o bit de *A* deve ser 1.

# Explicação: como funciona o código de Hamming?

Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.



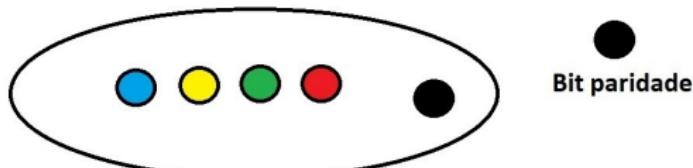
Para os M bits originais, se usamos um bit paridade:

Na leitura, se dá erro de paridade: **não sabemos qual bit está errado.**

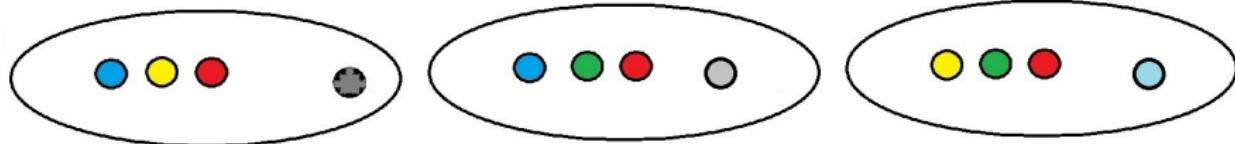


# Explicação: como funciona o código de Hamming?

Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.



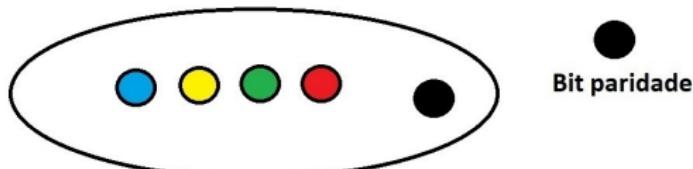
Formamos K subconjuntos: para cada subconjunto calculamos um bit paridade



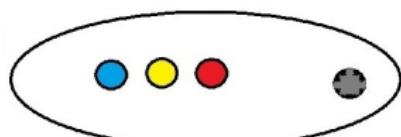
Após a leitura, vamos verificar se há erro de paridade em cada um dos subconjuntos

# Explicação: como funciona o código de Hamming?

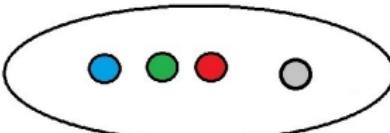
Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.



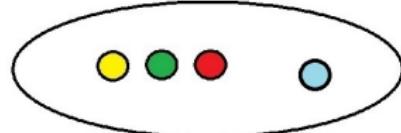
Formamos K subconjuntos: para cada subconjunto calculamos um bit paridade



Erro de paridade



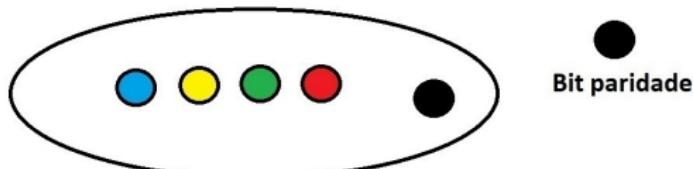
Erro de paridade



Paridade OK

# Explicação: como funciona o código de Hamming?

Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.

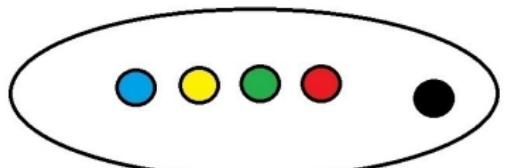


Formamos K subconjuntos: para cada subconjunto calculamos um bit paridade



# Explicação: como funciona o código de Hamming?

Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.



Bit paridade

Formamos K subconjuntos: para cada subconjunto calculamos um bit paridade



Um deles é culpado

Erro de paridade

Quem está aqui



Um deles é culpado

Erro de paridade

e aqui



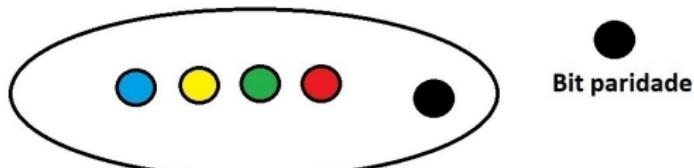
Todos são inocentes

Paridade OK

mas não está aqui?

# Explicação: como funciona o código de Hamming?

Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.

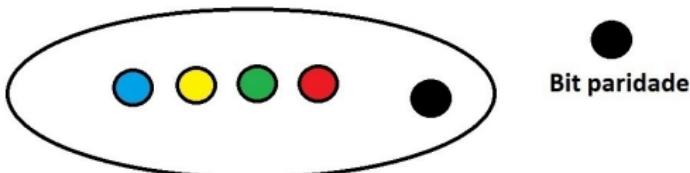


Formamos K subconjuntos: para cada subconjunto calculamos um bit paridade



# Explicação: como funciona o código de Hamming?

Supomos que após a leitura do código de Hamming, apenas um bit pode estar errado.  
Não consideramos erros de 2 ou mais bits.



Formamos K subconjuntos: para cada subconjunto calculamos um bit paridade

Quanto vale K para dado M?

Veremos:

Como formar os K subconjuntos?

Como formalizar tudo?

- Para o caso geral de uma palavra de  $M$  bits, quantos bits adicionais são necessários? Isto é:  
**Dado  $M$ , quanto vale  $K$ ?**
- Suponha que o código de Hamming lido (de  $M + K$  bits) pode ou estar correto ou errado em no máximo 1 bit. **Não consideramos erros em mais de um bit. O código de Hamming não funciona para este caso.**
- Depois de lido o código de Hamming, calculamos  $K$  paridades.
  - Se 0 paridade está errada: a palavra está correta.
  - Se 1 ou mais paridades erradas: um dos  $M + K$  bits foi lido errado.

# Uma palavra de $M$ bits precisa de $K$ bits adicionais

Calculadas  $K$  paridades, cada uma pode estar correta ou errada.

K Paridades

	1	2	3	4	...	K-2	K-1	K
	0	0	0	0		0	0	0
	0	0	0	0		0	0	1
	0	0	0	0		0	1	0
	0	0	0	0		0	1	1
	0	0	0	0		1	0	0
	0	0	0	0		1	0	1
	0	0	0	0		1	1	0
	0	0	0	0		1	1	1

**0 significa  
paridade correta**

**1 significa  
paridade errada**

1 1 1 1      1 1 1

# Uma palavra de $M$ bits precisa de $K$ bits adicionais

Calculadas  $K$  paridades, cada uma pode estar correta ou errada.

K Paridades

	1	2	3	4	...	K-2	K-1	K
	0	0	0	0		0	0	0
	0	0	0	0		0	0	1
	0	0	0	0		0	1	0
	0	0	0	0		0	1	1
	0	0	0	0		1	0	0
	0	0	0	0		1	0	1
	0	0	0	0		1	1	0
	0	0	0	0		1	1	1

**0 significa  
paridade correta**

**1 significa  
paridade errada**



$2^k$  possibilidades

1 1 1 1      1 1 1

Uma palavra de  $M$  bits precisa de  $K$  bits adicionais

Calculadas  $K$  paridades, cada uma podendo estar correta ou errada.

**0 significa  
paridade correta**

**1 significa  
paridade errada**



**$2^k$  possibilidades**

K Paridades							
1	2	3	4	...	K-2	K-1	K
0	0	0	0		0	0	0
0	0	0	0		0	0	1
0	0	0	0		0	1	0
0	0	0	0		0	1	1
0	0	0	0		1	0	0
0	0	0	0		1	0	1
0	0	0	0		1	1	0
0	0	0	0		1	1	1
.				.			
1	1	1	1		1	1	1

Sem erro: 1 possibilidade

Com erro:

$2^k - 1$  possibilidades

# Uma palavra de $M$ bits precisa de $K$ bits adicionais

Calculadas  $K$  paridades, cada uma podendo estar correta ou errada.

0 significa  
paridade correta

1 significa  
paridade errada



$2^k$  possibilidades

K Paridades							
1	2	3	4	...	K-2	K-1	K
0	0	0	0		0	0	0
0	0	0	0		0	0	1
0	0	0	0		0	1	0
0	0	0	0		0	1	1
0	0	0	0		1	0	0
0	0	0	0		1	0	1
0	0	0	0		1	1	0
0	0	0	0		1	1	1
.	.	.	.		.	.	.
1	1	1	1		1	1	1

Sem erro: 1 possibilidade

Com erro:

$2^k - 1$  possibilidades

O erro pode estar em um dos  $M + K$  bits.

Portanto:

$$2^k - 1 \geq M + K$$

- Devemos ter:  $2^K - 1 \geq M + K$ .
- Portanto  $K$  deve ser tal que  $2^K - 1 - K \geq M$ .
  - Exemplo: para  $M = 4$  e  $K = 3$  temos  $2^3 - 1 - 3 = 4 \geq 4$ .
  - Para  $M = 8$  e  $K = 4$  temos  $2^4 - 1 - 4 = 11 \geq 8$ .
- Então dado  $M$ , como calculamos  $K$ ?
  - Dado  $M$ , obtemos o menor  $K$  que satisfaz  $2^K - 1 - K \geq M$ .

# Caso geral: palavra de $M$ bits

Seja uma palavra dada de  $M$  bits. O código de Hamming precisa de  $K$  bits adicionais, com a condição:  $2^K - 1 - K \geq M$ .

Palavra de $M$ bits	Exemplo $M = 2^s$	$K$ bits adicionais
4	4	3
5 até 11	8	4
12 até 26	16	5
27 até 57	32	6
58 até 120	64	7
121 até 247	128	8

- Temos  $2^K \geq M + 1 + K > M$ .  
Para  $M = 2^s$ ,  $2^K > M$  ou  $2^K > 2^s$ .  
Assim temos  $K > s$ .  
Podemos fazer  $K = s + 1 = \log M + 1$ .
- Para  $M$  grande, o *overhead* é menor. Mas lembre-se que o código só funciona para erro de um só bit no código.

# Código de Hamming para palavra de $M = 8$ bits

Vamos ver como calcular o código de Hamming para uma palavra de  $M = 8$  bits. Numere os bits de

$$m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8$$

A essa palavra de 8 bits vamos acrescentar 4 bits adicionais, formando o código de Hamming de 12 bits.

Numere os bits do código de Hamming como sendo:

$$x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}$$

# Inserir os $M$ bits originais no código de Hamming

$x_1$  = a determinar  
 $x_2$  = a determinar  
 $x_3$  =  $m_1$   
 $x_4$  = a determinar  
 $x_5$  =  $m_2$   
 $x_6$  =  $m_3$   
 $x_7$  =  $m_4$   
 $x_8$  = a determinar  
 $x_9$  =  $m_5$   
 $x_{10}$  =  $m_6$   
 $x_{11}$  =  $m_7$   
 $x_{12}$  =  $m_8$

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$	$x_{12}$
?	?	$m_1$	?	$m_2$	$m_3$	$m_4$	?	$m_5$	$m_6$	$m_7$	$m_8$

Falta obter  $x_1, x_2, x_4, x_8$ : note índices todos potências de 2.

## Obtenção dos bits adicionais

Os 4 bits adicionais  $x_1, x_2, x_4$  e  $x_8$  são assim calculados, onde  $\oplus$  representa a operação *ou-exclusivo*:

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}$$

Observe que a operação ou-exclusivo é equivalente à paridade par.

# Explicação

Considere o conjunto dos 12 bits:

$$\{x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}\}$$

A ideia é

- Primeiro escolhemos vários subconjuntos desse conjunto de bits
- Vamos exigir que a paridade seja par para cada um desses subconjuntos
- Ao invés de um bit paridade, teremos vários bits de paridade. Isso será útil conforme veremos mais tarde.

Vejamos como escolher esses subconjuntos.

# Explicação

Considere o conjunto dos 12 bits:  $\{x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}\}$

Escrevemos os números 0 a 12 em binário:

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

São 4 colunas na tabela. Vamos ter 4 subconjuntos. Vejamos como obter o primeiro subconjunto.

# Explicação

Considere o conjunto dos 12 bits:  $\{x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}\}$

Escrevemos os números 0 a 12 em binário:

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

O primeiro subconjunto é (olhe a cor vermelha):  
 $\{x_1 x_3 x_5 x_7 x_9 x_{11}\}$

# Explicação

Considere o conjunto dos 12 bits:  $\{x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}\}$

Escrevemos os números 0 a 12 em binário:

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

Para  $\{x_1 x_3 x_5 x_7 x_9 x_{11}\}$  ter paridade par, basta fazer

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

# Explicação

Temos assim uma regra simples para calcular  $x_1, x_2, x_4$  e  $x_8$   
Veremos nos próximos slides.

# Uma regra simples para chegar às fórmulas

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Uma regra simples para chegar às fórmulas

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Uma regra simples para chegar às fórmulas

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Uma regra simples para chegar às fórmulas

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Uma regra simples para chegar às fórmulas

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Correção de erro

O código de Hamming calculado é gravado na memória:

$$x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}$$

Agora suponha que esses 12 bits são lidos como sendo:

$$y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12}$$

Se não houver erro, então cada  $y_i$  é igual seu respectivo  $x_i$ .

Se houver erro em um bit apenas, é possível detectar esse erro e corrigi-lo.

Para isso fazemos o seguinte cálculo de 4 bits de paridade, denominados  $k_1, k_2, k_3$  e  $k_4$ .

# Correção de erro

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

Se  $k_1 = k_2 = k_3 = k_4 = 0$ , então não há erro.

Senão o bit  $y_i$  (onde  $i$  é o valor decimal de  $k_4k_3k_2k_1$ ) está errado. Mais tarde vamos mostrar o por quê.

# Correção de erro

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

Exemplo, se  $k_4 k_3 k_2 k_1 = 0111$  entao o bit  $y_7$  está errado.

# Correção de erro

De onde vieram essas fórmulas?

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Correção de erro

De onde vieram essas fórmulas?

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Correção de erro

De onde vieram essas fórmulas?

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Correção de erro

De onde vieram essas fórmulas?

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

0 a 12 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

# Correção de erro

Suponha ao ler um código de Hamming, os seguintes bits foram lidos. (Supomos apenas um bit pode estar errado.)

$$y_1 \ y_2 \ y_3 \ y_4 \ y_5 \ y_6 \ y_7 \ y_8 \ y_9 \ y_{10} \ y_{11} \ y_{12}$$

Calculamos os 4 bits de paridade  $k_1, k_2, k_3, k_4$  pelas fórmulas vistas:

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

Suponha  $k_4k_3k_2k_1 = 0111$  então o bit  $y_7$  está errado. Por quê?

- $k_1 = 1$  significa que o bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- $k_2 = 1$  significa que o bit errado está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- $k_3 = 1$  significa que o bit errado está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- $k_4 = 0$  significa que o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ . Portanto em  $\{y_7\}$

Nos próximos slides vamos explicar:

*Se  $k_4k_3k_2k_1 = 0111$  entao o bit  $y_7$  está errado. Por quê?*

O que se segue não faz parte da etapa para descobrir o bit errado, mas tão somente uma explicação sobre o método.

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$ .

- O bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ . Vamos calcular isso.

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1								0	0	0	1
2								0	0	1	0
3								0	0	1	1
4								0	1	0	0
5								0	1	0	1
6								0	1	1	0
7								0	1	1	1
8								1	0	0	0
9								1	0	0	1
10								1	0	1	0
11								1	0	1	1
12								1	1	0	0

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$ .

- O bit errado **está** no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ . Vamos calcular isso.

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1	1							0	0	0	1
2								0	0	1	0
3	3							0	0	1	1
4								0	1	0	0
5	5							0	1	0	1
6								0	1	1	0
7	7							0	1	1	1
8								1	0	0	0
9	9							1	0	0	1
10								1	0	1	0
11	11							1	0	1	1
12								1	1	0	0

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$ .

- O bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também **está** no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ . Vamos calcular isso.

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1	1							0	0	0	1
2			2					0	0	1	0
3	3			3				0	0	1	1
4								0	1	0	0
5	5							0	1	0	1
6			6					0	1	1	0
7	7			7				0	1	1	1
8								1	0	0	0
9	9							1	0	0	1
10			10					1	0	1	0
11	11			11				1	0	1	1
12								1	1	0	0

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$ .

- O bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também **está** no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ . Vamos calcular isso.

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1	1							0	0	0	1
2			2					0	0	1	0
3	3			3				0	0	1	1
4					4			0	1	0	0
5	5				5			0	1	0	1
6			6		6			0	1	1	0
7	7			7		7		0	1	1	1
8								1	0	0	0
9	9							1	0	0	1
10			10					1	0	1	0
11	11			11				1	0	1	1
12					12			1	1	0	0

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$ .

- O bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado **não está** no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ . Vamos calcular isso.

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1	1							0	0	0	1
2			2					0	0	1	0
3	3		3					0	0	1	1
4				4				0	1	0	0
5	5				5			0	1	0	1
6			6		6			0	1	1	0
7	7		7		7			0	1	1	1
8					8			1	0	0	0
9	9				9			1	0	0	1
10			10		10			1	0	1	0
11	11		11			11		1	0	1	1
12					12		12	1	1	0	0

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$ .

- O bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $\mathbf{A} \cap \mathbf{B} \cap \mathbf{C} - \mathbf{D}$ . Portanto o bit errado é  $y_7$ .

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1								0	0	0	1
2								0	0	1	0
3								0	0	1	1
4								0	1	0	0
5								0	1	0	1
6								0	1	1	0
7	7		7		7			0	1	1	1
8								1	0	0	0
9								1	0	0	1
10								1	0	1	0
11								1	0	1	1
12								1	1	0	0

# Correção de erro

Seja  $k_4k_3k_2k_1 = 0111$  Portanto BIT ERRADO: 7.

- O bit errado está no conjunto  $A = \{y_1, y_3, y_5, y_7, y_9, y_{11}\}$
- O bit errado também está no conjunto  $B = \{y_2, y_3, y_6, y_7, y_{10}, y_{11}\}$
- O bit errado também está no conjunto  $C = \{y_4, y_5, y_6, y_7, y_{12}\}$
- E o bit errado não está no conjunto  $D = \{y_8, y_9, y_{10}, y_{11}, y_{12}\}$
- O bit errado portanto está em  $A \cap B \cap C - D$ .

	$A$	$\cap$	$B$	$\cap$	$C$	-	$D$	8	4	2	1
0								0	0	0	0
1								0	0	0	1
2								0	0	1	0
3								0	0	1	1
4								0	1	0	0
5								0	1	0	1
6								0	1	1	0
7								0	1	1	1
8								1	0	0	0
9								1	0	0	1
10								1	0	1	0
11								1	0	1	1
12								1	1	0	0

Recapitulando: lido o código de Hamming, a verificação se há erro é assim:

$$\begin{aligned}k_1 &= y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \\k_2 &= y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \\k_3 &= y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_{12} \\k_4 &= y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12}\end{aligned}$$

Exemplo, se  $k_4 k_3 k_2 k_1 = 0111$  entao o bit  $y_7$  está errado.

**Não é necessário calcular  $A \cap B \cap C - D$ , que serve apenas para mostrar que o método funciona.**

# Lista de Exercícios 4

- Fazer e entregar por email a [Lista de Exercícios 4](#).
- Há prazo para entrega. Recomendo não demorar muito.  
Bom fazer logo com a matéria fresquinha na cabeça.

- O código de Hamming **não** funciona para erro em mais de um bit no código.
- Há uma extensão do método que permite corrigir erros de 1 bit e detectar erros de 2 bits (mas sem corrigi-los). Esse método usa um bit a mais, i.e.  $K + 1$  bits adicionais. (Não vamos ver esse método aqui.)
- Em comunicação de dados, onde uma sequência longa de bits é transmitida de um local a outro, é comum uma série consecutiva de bits ser danificada.
- Veremos um truque que permite detectar e corrigir erros em uma sequência de bits.
- **(Uau, que legal!!)**, não posso perder essa dica! :-)

# Como foi o meu aprendizado?

Vamos fazer um exercício juntos.

- Parte 1: Escolha um número  $M$  entre 5 a 11. Invente um número de  $M$  bits e escreva o código de Hamming.
- Parte 2: Agora erre um bit nesse código e use a técnica para corrigir o erro. (Obviamente não é para olhar a primeira parte para descobrir qual bit errado :-)

Continuamos esse exercício nos próximos slides.

# Como foi o meu aprendizado?

Parte 1: Escolha um número  $M$  entre 5 a 11. Invente um número de  $M$  bits e escreva o código de Hamming.

- Por exemplo escolhemos  $M = 7$  e seja o dado de 7 bits como sendo 1001110.
- Pela tabela vista anteriormente (slide no. 60), para 7 bits de dados precisamos acrescentar 4 bits adicionais.
- Mas podemos deduzir que precisamos de mais 4 bits, assim.
- Escrevemos uma linha com  $x_1, x_2, x_3, x_4, x_5, x_6, \dots$
- Deixamos de lado os  $x_i$  onde  $i$  é potência de 2 (i.e.  $x_1, x_2, x_4, x_8, \text{etc.}$ ) e preenchemos os bits do número dado.

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$
1				0	0	1		1	1	0

- Notamos que precisamos usar  $x_1, x_2, x_4, x_8$  e portanto 4 bits adicionais. Vamos agora calcular esses 4 bits.

# Como foi o meu aprendizado?

Cálculo de  $x_1$ :

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$
?		1		0	0	1		1	1	0

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11} = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

# Como foi o meu aprendizado?

Cálculo de  $x_2$ :

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$
1	?	1		0	0	1		1	1	0

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11} = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

# Como foi o meu aprendizado?

Cálculo de  $x_4$ :

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$
1	1	1	?	0	0	1		1	1	0

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$x_4 = x_5 \oplus x_6 \oplus x_7 = 0 \oplus 0 \oplus 1 = 1$$

# Como foi o meu aprendizado?

Cálculo de  $x_8$ :

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$
1	1	1	1	0	0	1	?	1	1	0

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} = 1 \oplus 1 \oplus 0 = 0$$

# Como foi o meu aprendizado?

Resposta da Parte 1: o código de Hamming calculado foi:

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$	$x_{11}$
1	1	1	1	0	0	1	0	1	1	0

Parte 2: Agora erre um bit nesse código e use a técnica para corrigir o erro.

Os bits lidos  $y_i$  são os seguintes. Note que o bit 7 foi lido erradamente.

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$	$y_{11}$
1	1	1	1	0	0	0	0	1	1	0

Vamos agora ver como detectamos e corrigimos o erro.

# Como foi o meu aprendizado?

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$	$y_{11}$
1	1	1	1	0	0	0	0	1	1	0

Calculamos  $k_1$ :

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$k_1 = y_1 \oplus y_3 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

# Como foi o meu aprendizado?

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$	$y_{11}$
1	1	1	1	0	0	0	0	1	1	0

Calculamos  $k_2$ :

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$k_2 = y_2 \oplus y_3 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

# Como foi o meu aprendizado?

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$	$y_{11}$
1	1	1	1	0	0	0	0	1	1	0

Calculamos  $k_3$ :

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

$$k_3 = y_4 \oplus y_5 \oplus y_6 \oplus y_7 = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

# Como foi o meu aprendizado?

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$	$y_{11}$
1	1	1	1	0	0	0	0	1	1	0

Calculamos  $k_4$ :

0 a 11 em binário	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

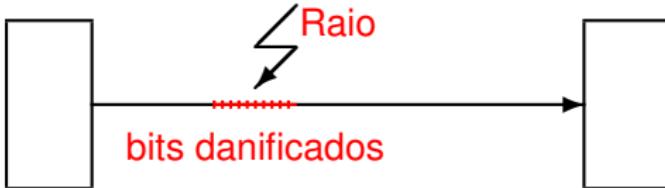
$$k_4 = y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

# Como foi o meu aprendizado?

$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$	$y_8$	$y_9$	$y_{10}$	$y_{11}$
1	1	1	1	0	0	0	0	1	1	0

Obtivemos  $k_4k_3k_2k_1 = 0111$  (7 em decimal).

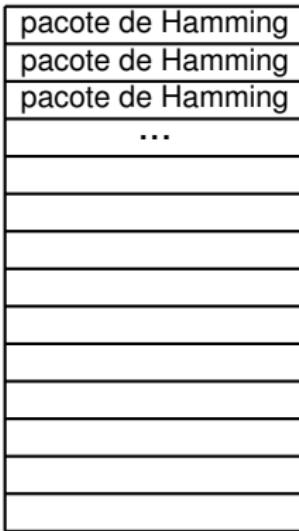
Portanto o bit  $y_7$  está errado. Ao invés de 0 corrigimos para 1.



Vamos ilustrar por um exemplo em comunicação de dados.

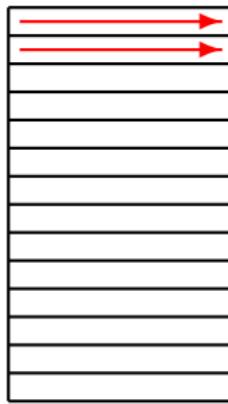
- Uma mensagem constituída de um número de pacotes (cada pacote tem  $M$  bits) deve ser enviada de um local a outro.
- O meio de transmissão é sujeito a chuvas e trovoadas :-)  
quando um raio pode danificar uma sequência de bits consecutivos.
- Não queremos apenas detectar erro de transmissão e pedir para retransmitir os pacotes errados. Queremos corrigir os erros.

# Erro em mais de 1 bit



- Vamos acrescentar a cada pacote de  $M$  bits os  $K$  bits adicionais conforme estudamos no código de Hamming. Chamamos cada pacote assim incrementado de **pacote de Hamming**.
- Colocamos os pacotes de Hamming em uma matriz.

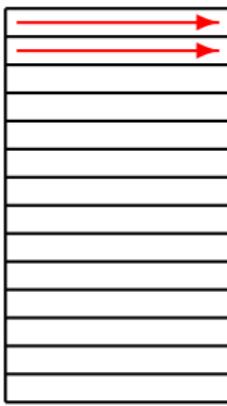
# Erro em mais de 1 bit



- Se transmitirmos esses pacotes de Hamming sequencialmente, um a um, então o dano de um raio (que estraga uma série consecutiva de bits) pode ser irrecuperável. Nada adiantou :-(.

Agora vem a ideia brilhante :-).

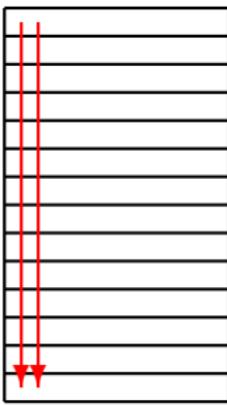
# Erro em mais de 1 bit



- Se transmitirmos esses pacotes de Hamming sequencialmente, um a um, então o dano de um raio (que estraga uma série consecutiva de bits) pode ser irrecuperável. Nada adiantou :-(.

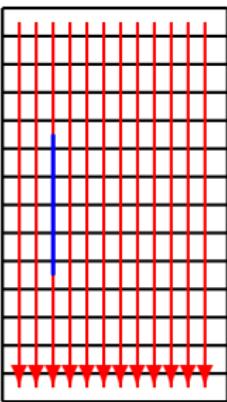
Agora vem a ideia brilhante :-).

# A ideia brilhante



- Basta transmitirmos a matriz por **coluna**. No outro lado da recepção coletamos os bits recebidos para reconstruir a matriz.
- Agora aplicamos método de Hamming para cada pacote de Hamming recebido.

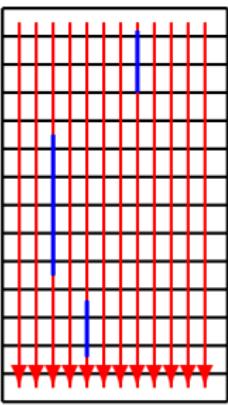
# A idéia brilhante



Cor azul = bits danificados

- Basta transmitirmos a matriz por **coluna**. No outro lado da recepção coletamos os bits recebidos para reconstruir a matriz.
- Cada bit errado (**bit azul na figura**) está num pacote de Hamming. Por isso, podemos corrigi-los.

# A idéia brilhante



Cor azul = bits danificados

- Basta transmitirmos a matriz por **coluna**. No outro lado da recepção coletamos os bits recebidos para reconstruir a matriz.
- Com sorte, pode até aguentar erros de várias sequências de bits. Basta não ter mais um erro em cada pacote.

# Como foi o meu aprendizado?

Marque as afirmações corretas.

- 1 DRAM e SRAM são ambas voláteis, mas SRAM precisa de circuitaria de refrescamento para repor as cargas que se perdem por vazamento.
- 2 DRAM e SRAM são ambas não-voláteis e assim seu conteúdo não se perde mesmo sem energia elétrica.
- 3 DRAM e SRAM são ambas voláteis, mas DRAM precisa de circuitaria de refrescamento para repor as cargas que se perdem por vazamento.
- 4 A memória ROM é não-volátil mas EPROM é volátil.
- 5 Todos os tipos de memória ROM são não-voláteis.
- 6 A memória flash pode ser regravada mas somente pelo fabricante.
- 7 O número de ciclos de escrita numa memória flash é grande mas não é ilimitado.
- 8 O uso de um bit de paridade pode corrigir erros de memória quando há apenas um bit errado.
- 9 O código de Hamming serve para corrigir erros de memória quando há apenas um bit errado.

# Como foi o meu aprendizado?

*Desejo usar o código de Hamming para corrigir erro de memória.*

*Qual das duas alternativas está mais adequada?*

- Devemos escolher  $M$  bem grande, digamos  $2^{10}$ . Assim, nesse caso, usaremos apenas 11 bits adicionais, uma grande economia.
- Para proteger contra erro de leitura de um dado grande, digamos  $2^{10}$  de bits, o melhor é dividir esse dado em blocos menores e para cada um deles usar o código de Hamming.

- Próximo assunto: Memória externa
- Disco magnético (HD), Discos RAID (Redundant Array of Independent Disks), SSD (*Solid State Disks*), etc.
- Não percam!