Arquitetura de Computadores

MIEI – 2021/22

DI-FCT/UNL

Desenho e Implementação do CPU

A unidade de processamento gráfico (GPU)

Sumário

- Desenho e Implementação do CPU
  - RISC vs CICS
  - Paralelismo ao nível da instrução
  - Instruções vetoriais
- Taxonomia de Flynn
  - Arquiteturas MIMD de memória partilhada
    - Revolução Multicore
    - Caching
    - Arquiteturas NUMA
  - Arquiteturas MIMD de memória distribuída
- Unidade de processamento gráfico (GPU)
- Top 500

Bibliografia:

Bryant, O'Hallaron Computer Systems: A Programmer's Perspective, 2nd ou 3rd Ed, secções 4.1, 4.5 e 4.6 (não considerando os detalhes do SEQ+)

GPU Architecture and Models **(**https://www.cs.utah.edu/~jeffp/teaching/MCMD/S20-GPU.pdf)

RISC vs CISC

- RISC – Reduced Instruction Set Computer
- Nova abordagem (anos 70/80) no desenho dos CPU. Simplificar para conseguir melhor desempenho:
  - Suportar um pequeno conjunto de instruções: as mais usadas
  - Instruções de tamanho fixo: fetch mais simples e eficiente
  - Descodificação mais simples e eficiente
  - Menos instruções a optimizar, a execução pode ser mais eficiente
  - Usar espaço no CPU para mais registos e mais cache
  - Permitir explorar mais optimizações no hardware e nos compiladores…
- A abordagem antiga passou a ser referida por:

  CISC – Complex Instruction Set Computer

RISC vs CISC

CISC

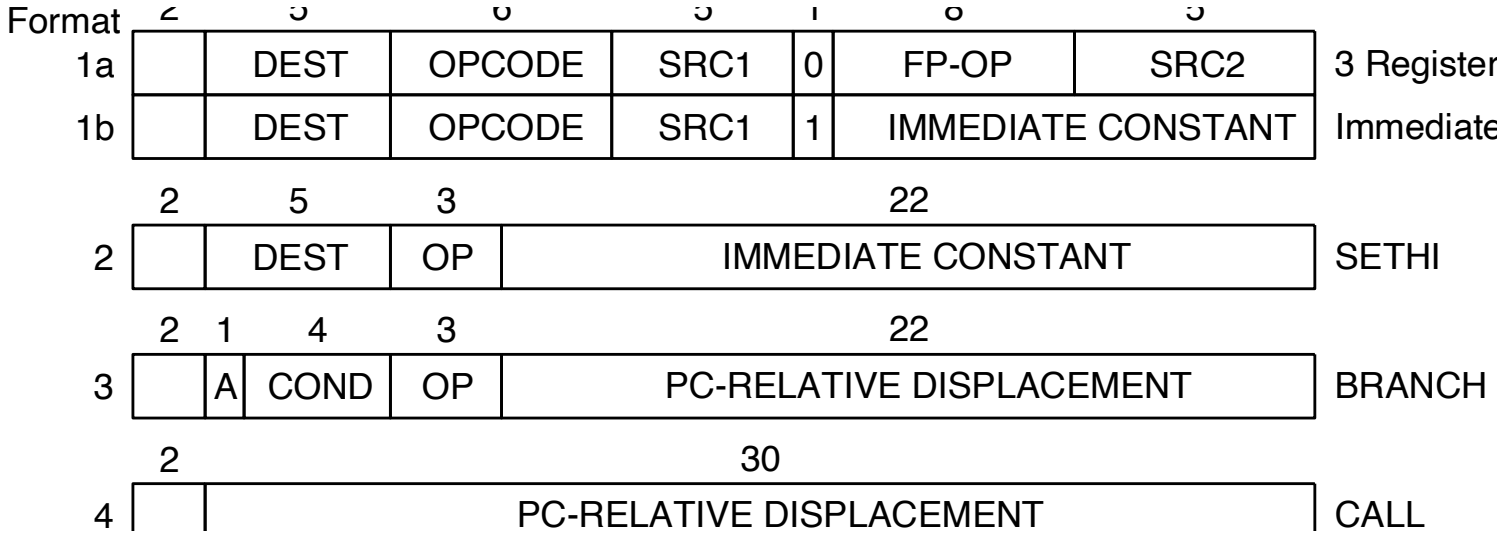
- Instruções complexas → decodificação complexa.
- Instruções de tamanho variável
  - As instruções podem ser maiores do que o tamanho de uma palavra.
- A instrução pode demorar mais do que um ciclo de relógio a ser executada.
- Operações aritméticas com operandos REG-REG, MEM-REG, REG-MEM
  - Menor número de registos de uso geral, dado que as operações podem ser executadas na própria memória.
- Modos de endereçamento complexos.
- Mais tipos de dados.
- Código pequeno (menos instruções)
- Implementação com **pipeline** requer uso de **microcódigo**
  - Operaões em hardware outras em microcódigo

RISC

- Instruões mais simples → decodificação mais simples.
- Instruções de tamanho fixo
  - Instrução cabe numa palavra.
- A instrução demora um único ciclo de relógio a ser executada.
- Operações aritméticas só REG-REG
  - Mais registos de uso geral.
- Modos de endereçamento simples.
- Menos tipos de dados.
- Todas as instruções do ISA implementadas diretamente em hardware
- Pode-se implementar com um **pipeline**
- Código maior (mais instruções)

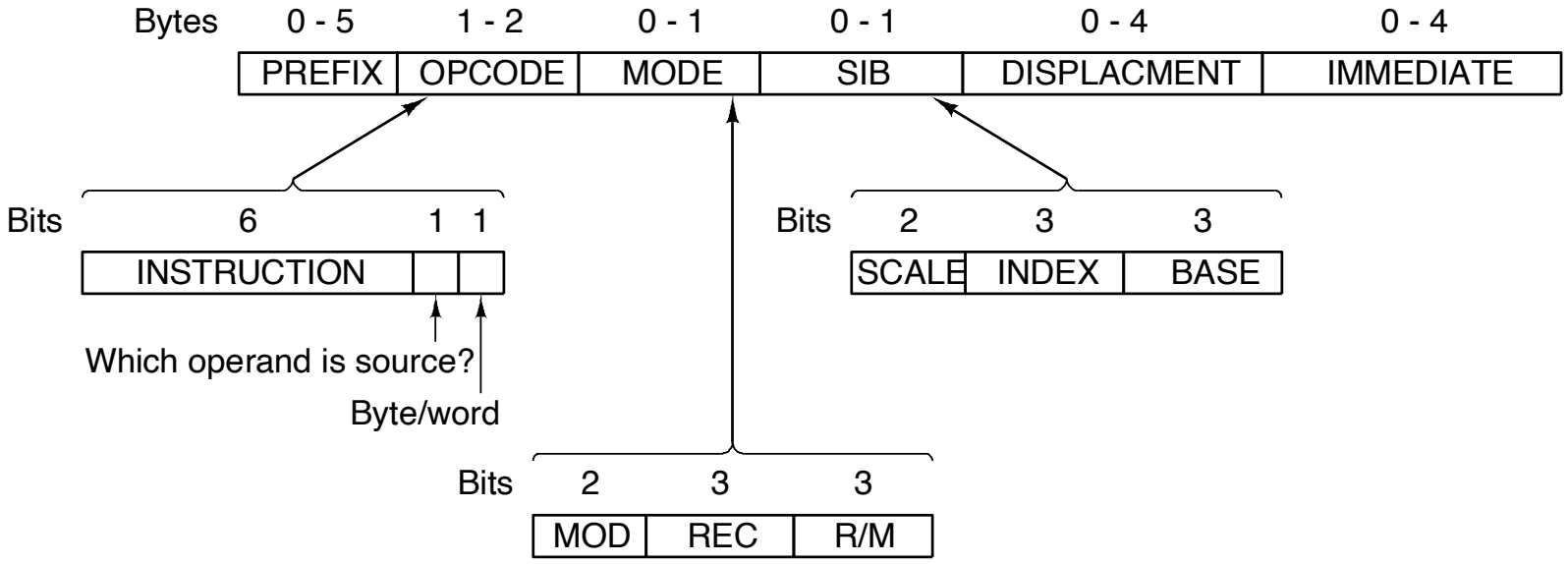
Exemplo de arquitectura RISC

- Família SPARC/UltraSPARC, MIPS, PowerPC, ARM
- Exemplo de um conjunto de instruções: SPARC
  - Todas têm o mesmo tamanho



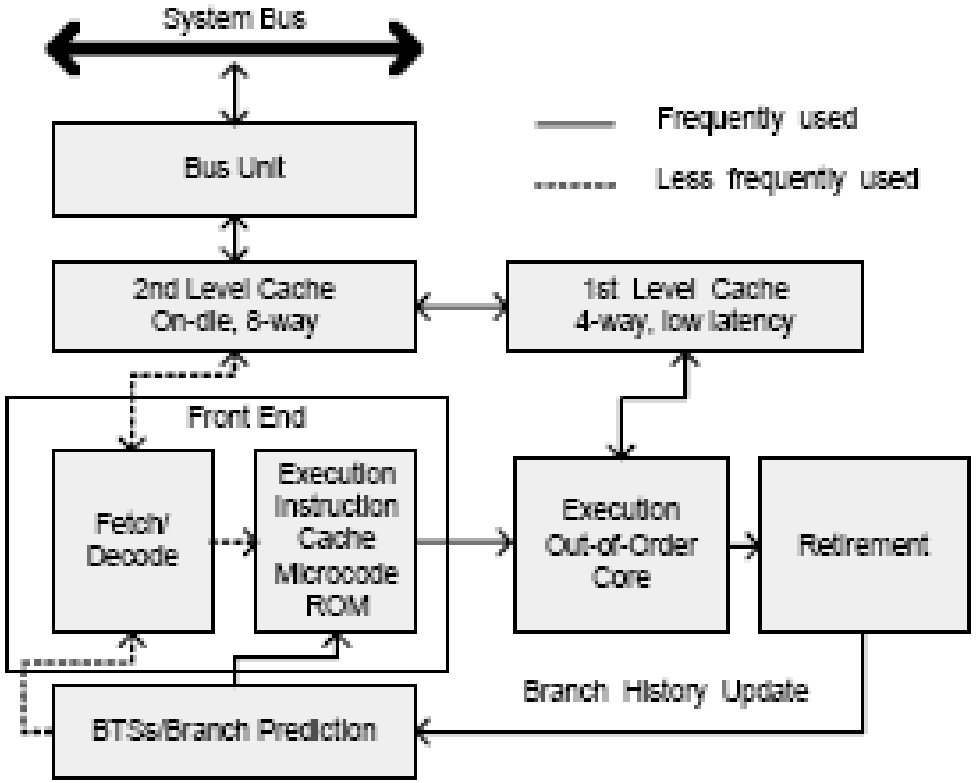
A arquitectura Intel é CISC?

- Ao nível do ISA, sim
  - O conjunto de instruções é complexo
  - Assim como a sua representação em código máquina



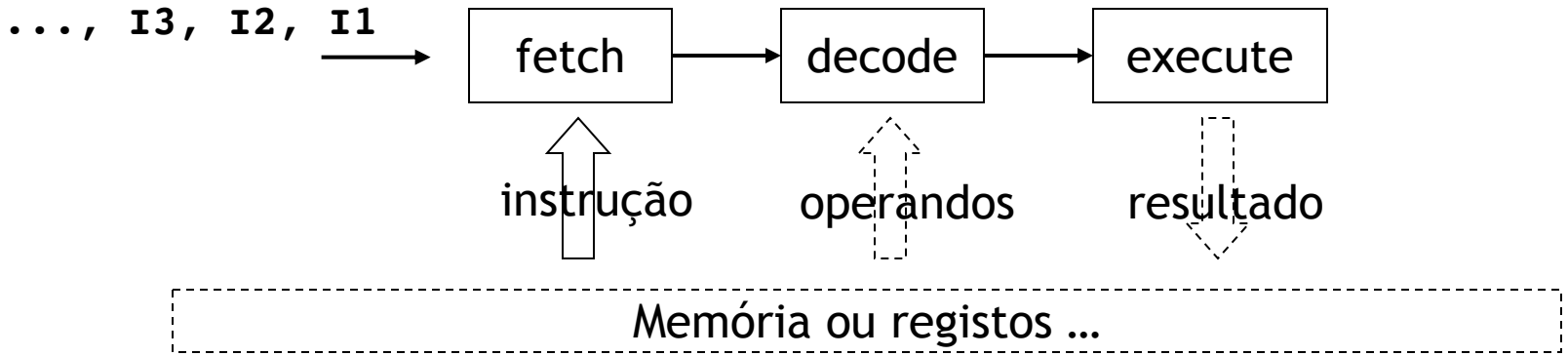
A arquitectura Intel é CISC?

- Ao nível da execução, não
- As instruções mais complexas são traduzidas para um microcódigo RISC



Pipeline

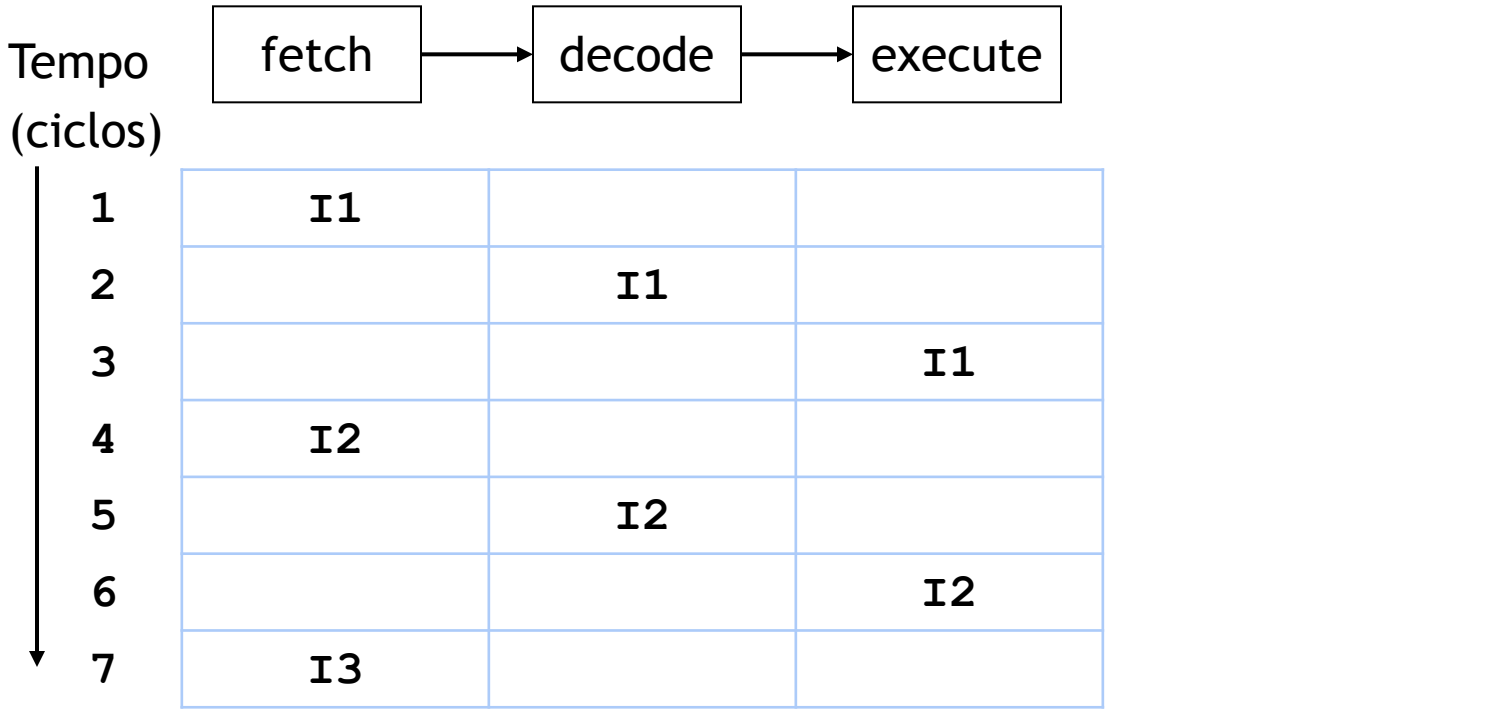
- A execução de uma instrução passa por várias fases:
  - Vimos o ciclo: fetch, decode, execute…



  - Se em média, cada instrução usar 1 ciclo em cada fase, em média, cada instrução demora 3 ciclos para executar
    - Mas os acessos a memória são “caros”

Uma instrução de cada vez

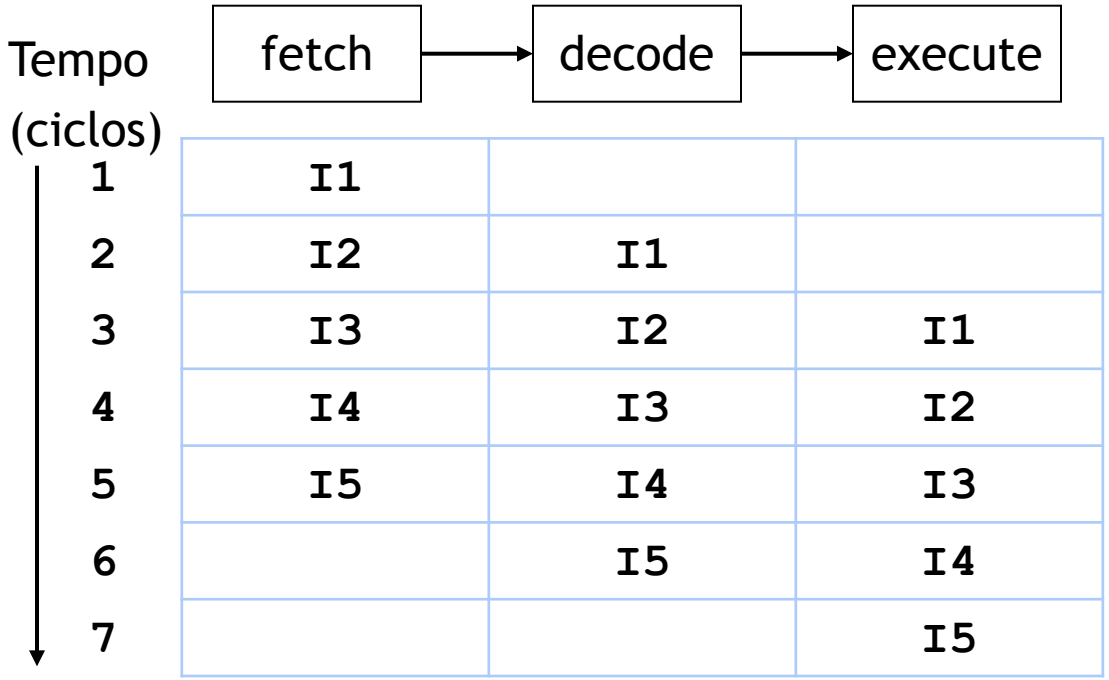
- Execução de várias instruções em sequência:



2 inst / 6 ciclos

Pipelining

- Supondo que a arquitetura permite manter o pipeline sempre ocupado, como numa linha de montagem:



5 inst / 7 ciclos

Pipelining

- Mesmo que cada instrução demore 3 ciclos, o CPU é capaz de concluir **uma instrução em cada ciclo**!

  (mesma latência, mas melhor débito)

- Tempo para executar uma sequência de 1000 instruções:
  - Sem pipelining: 1000x3 = 3000 ciclos
  - Com pipelining:

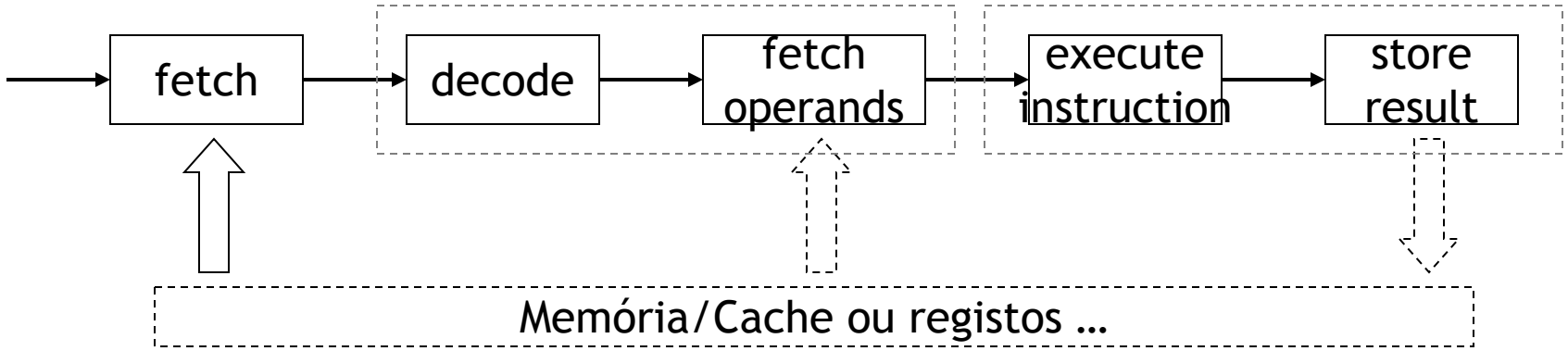
    3 ciclos para a primeira instrução (pipeline vazio)

    + 1 ciclo por cada uma das restantes

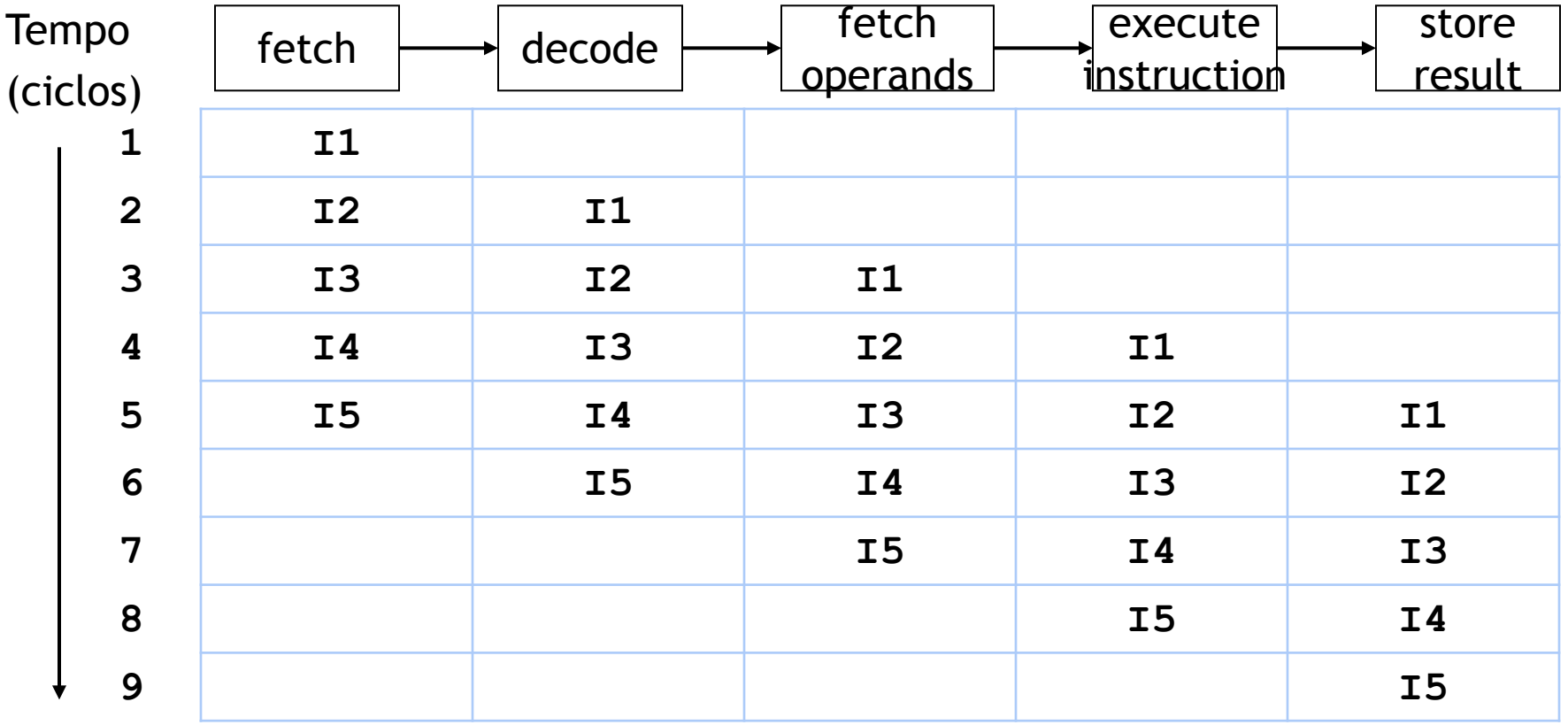
    → 3+999x1 = 1002 ciclos
  - Speedup = 3000/1002 = 2,99 (aprox. 3)

Aumentando o pipeline

- Supondo que podemos introduzir fases intermédias sem penalizar o tempo total de cada instrução
  - Por exemplo: a arquitectura é mais eficiente em cada fase ou permite aumentar o clock
- Exemplo: com um pipeline de 5 fases:



Pipeline maior



1 inst /5 ciclos vs 5 inst / 9 ciclos

Pipelining

- Tempo para executar uma sequência de 1000 instruções:
  - Sem pipelining: 1000x5 = 5000 ciclos
  - Com pipelining: 5 ciclos para a primeira instrução (pipeline vazio) + 1 ciclo por cada uma das restantes → 5+999x1 = 1004 ciclos
  - Speedup = 5000/1004 = 4,98 (aprox. 5)

  - Mas se cada ciclo demorar o mesmo, o débito é igual ao pipeline de 3 fases

Aumentando o pipeline

- É possível? Sim.
- Cada fase é mais simples e pode demorar menos tempo
  - Exemplo: antes CPU 1GHz → 3 ciclos = 3ns
  - Se agora CPU 1,66GHz → 5 ciclos ≈ 3ns
    - Manter o pipeline ocupado e aumentar a frequência é conseguido mais facilmente nos CPU RISC
- Tempo para as 1000 instruções:
  - Sem pipelining: 3000x1 = 3000 ns
  - Com pipeline de 3: 1002x1 = 1002 ns (speedup 2,99)
  - Com pipeline de 5: 1004x0,6 = 602 ns (speedup 4,98)
- Idealmente, Speedup ≈ número de fases do pipeline

Premissas e desafios

- Podemos aumentar a profundidade do pipeline (número de fases) penalizando pouco a latência (tempo de individual de cada instrução)
- Cada componente do pipeline está sempre ocupado (o pipeline está sempre cheio)
  - O que pode correr mal?
  - Conflito no acesso a recursos
  - Dependências entre operandos (dados)
  - Uma instrução ser um salto (branch)
  - Interrupções

Conflito no acesso a recursos

- Algumas fases podem entrar em conflito no acesso a partes do CPU ou exterior. Exemplos:
  - Duas instruções seguidas que usam o mesmo registo
  - Os acessos a memória no fetch, para obter operando e para guardar o resultado

  - Nos RISC só os load/store acedem à memória
  - Bus para dados e instruções separados: caches L1 para código e dados separadas

Dependências entre operandos

- Dependência entre os dados de instruções diferentes
  - Instrução I_n necessita do resultado da I_{n-k}

```
add $5, %R1
add %R1, %R2
add $4, %R3
add %R3, %R4
```

→ Reordenar instruções para adiar I_n

Fetch	Desc	Oper	Exec	Store	
Add R1					
Add R2	Add R1				
Add R3	Add R2	Add R1			
Add R4	Add R3	Add R2	Add R1		
Add R4	Add R3	Add R2	X	Add R1	*stall*
	Add R4	Add R3	Add R2	X	
		Add R4	Add R3	Add R2	
		Add R4	X	Add R3	*stall*
			Add R4	X	
				Add R4	

Dependências entre operandos

- Reordenando instruções:
  - add $5, %R1
  - add $4, %R3
  - add %R1, %R2
  - add %R3, %R4

- Os compiladores podem ter este problema em conta
- Os CPU modernos possuem um componente que reordena as instruções no pipeline

Fetch	Desc	Oper	Exec	Store
Add R1				
Add R3	Add R1			
Add R2	Add R3	Add R1		
Add R4	Add R2	Add R3	Add R1	
	Add R4	Add R2	Add R3	Add R1
		Add R4	Add R2	Add R3
			Add R4	Add R2
				Add R4

Instruções de salto

- Saltos (jumps/calls …)
  - Alteram o IP, logo a próxima instrução a executar pode não ser a que está no pipeline
  - Nos jumps condicionais nem sabemos qual vai ser a próxima instrução.
    - Se não saltar vai executar a próxima instrução, mas se saltar, vai executar outra…
  - Se salto incondicional: avalia o destino durante a descodificação
  - Se salto condicional: usar **branch prediction**

Branch prediction

- Acrescenta-se um componente ao CPU responsável por “avaliar” o próximo valor do IP
  - Tenta “adivinhar” os *jumps* condicionais assim que entram no pipeline
  - Exemplos de heurísticas:
    - O *jump* vai ocorrer se salta para trás (ciclos?)
    - Não ocorre se é um salto para a frente (saída de um ciclo?)
    - O *jump* vai ocorrer se antes ocorreu…

Ainda mais rápido …

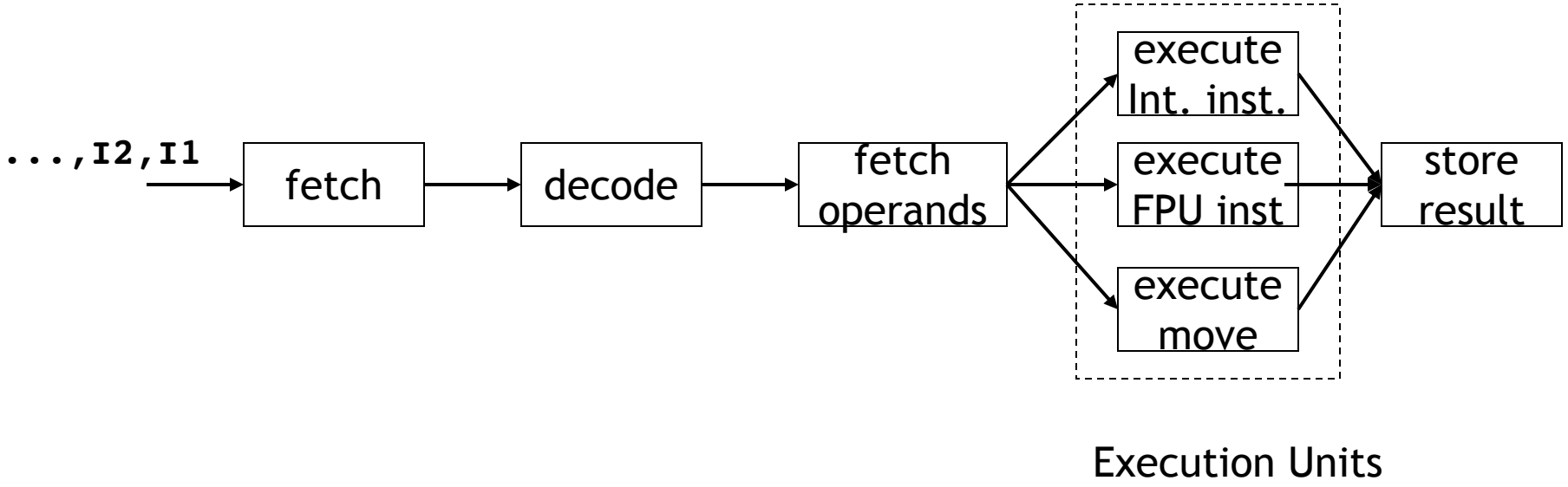
- Ainda é possível mais?

  - Tipicamente a fase de execução é a mais demorada
  - Os CPU possuem unidades diferentes para os vários tipos de instruções (FPU, ALU, etc…)
  - Estas podem trabalhar em paralelo

  Pode conseguir uma média de mais de uma instrução por ciclo de relógio → CPU ***Superscalar***

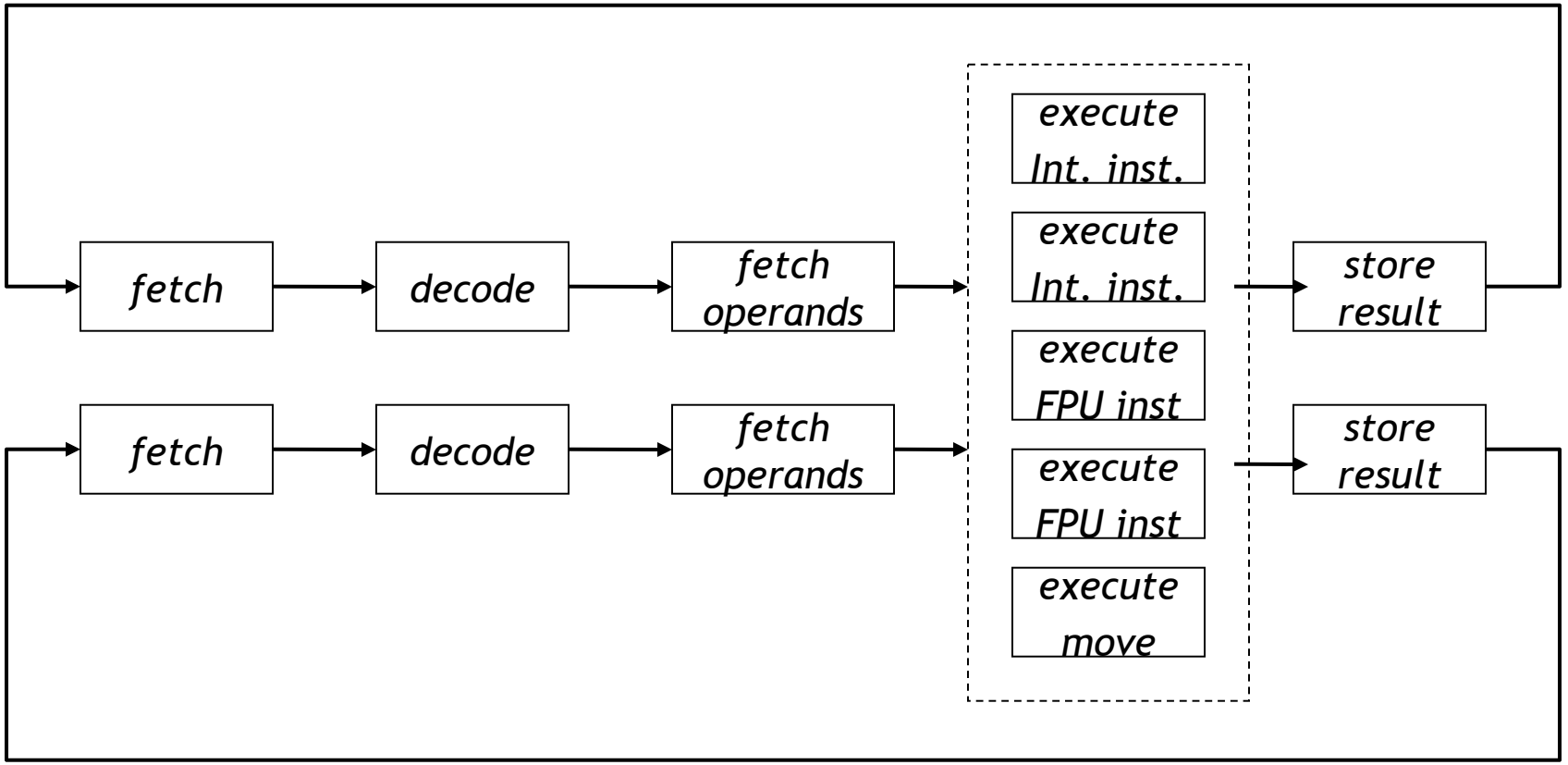
Paralelizando a execução

- Mais de uma instrução executada em simultâneo
  - várias instruções estão simultaneamente no mesmo estágio da sua execução
  - Convém que a sequência de instruções no programa alterne entre os vários tipos
  - Reordena instruções



Vários pipelines e várias unidades de execução

- Hyperthreading
  - Tecnologia usada nos processadores Intel para "oferecerem" 2 processadores lógicos
  - Duplica registos, pipelines e PIC



Paralelismo ao nível das instruções

- O desenho da arquitetura do CPU permite o paralelismo na execução de várias instruções:
- Pipelining
  - Sobrepõem a execução das várias fases do pipeline, para várias instruções
- Super-escalar
  - Várias unidades de execução executam várias instruções em simultâneo
  - Permite executar mais de uma instrução por ciclo de relógio!
- Vários pipelines e conjuntos de registos
  - Múltiplos threads (*hyperthreading*)

Limitações

- Não se pode aumentar sempre o relógio
- Os problemas com o pipelining aumentam se aumentar o pipeline
- A memória (mesmo a cache) tem limite no *throughput* que é capaz, para alimentar o pipeline
- É difícil manter as várias unidades de execução ocupadas em simultâneo
  - Depende de uma boa sequência/reordenação de instruções

Outro modelo de instruções

- Tradicional: operações sobre escalares
  - Exemplo (add): R3 ← R1 + R2

    Uma soma de dois escalares para obter um terceiro

- Vetorial: operações sobre grupos de escalares
  - Exemplo: V3 ← V1 + V2

    Uma soma de todos os elementos no vetor V1 com os de V2 para obter um terceiro vetor
  - Equivale a:

```
for ( i=0; i<N; i++ )
    V3[i] = V1[i] + V2[i];
```

Comparação (em pseudo-assembly)

Exemplo: vetores 3 D

mov $3, %R1

Loop:

mov V1(%R1), %R2

add V2(%R1), %R2

mov R2, V3(%R1)

sub $1, %R1

jnz Loop

Com registos vetoriais (dim. 3) :

loadV (V1), %R1

loadV (V2), %R2

addV %R1, %R2

storeV %R2, (V3)

- Menos instruções → menos fetchs
- Garantidamente não existem *branchs*

Primeiro exemplo comercial

- CRAY – 1 (1976)
  - 80 MHz
  - 8 registos de 64 bits
  - 8 registos vectoriais de 64 elementos de 64 bits
  - 6 unidades de execução

https://www.chipsetc.com/cray-research.html

Operações Vectoriais

- Vantagens
  - Menos instruções no programa
  - Menos fetch/decode
  - Paralelismo sobre grupos de valores
  - Menos acessos à memória
  - Menos problemas no aproveitamento do *pipelining*
- Desvantagens
  - Custa o mesmo operar um escalar que um vetor
  - Que código é realmente “vetorizável”?

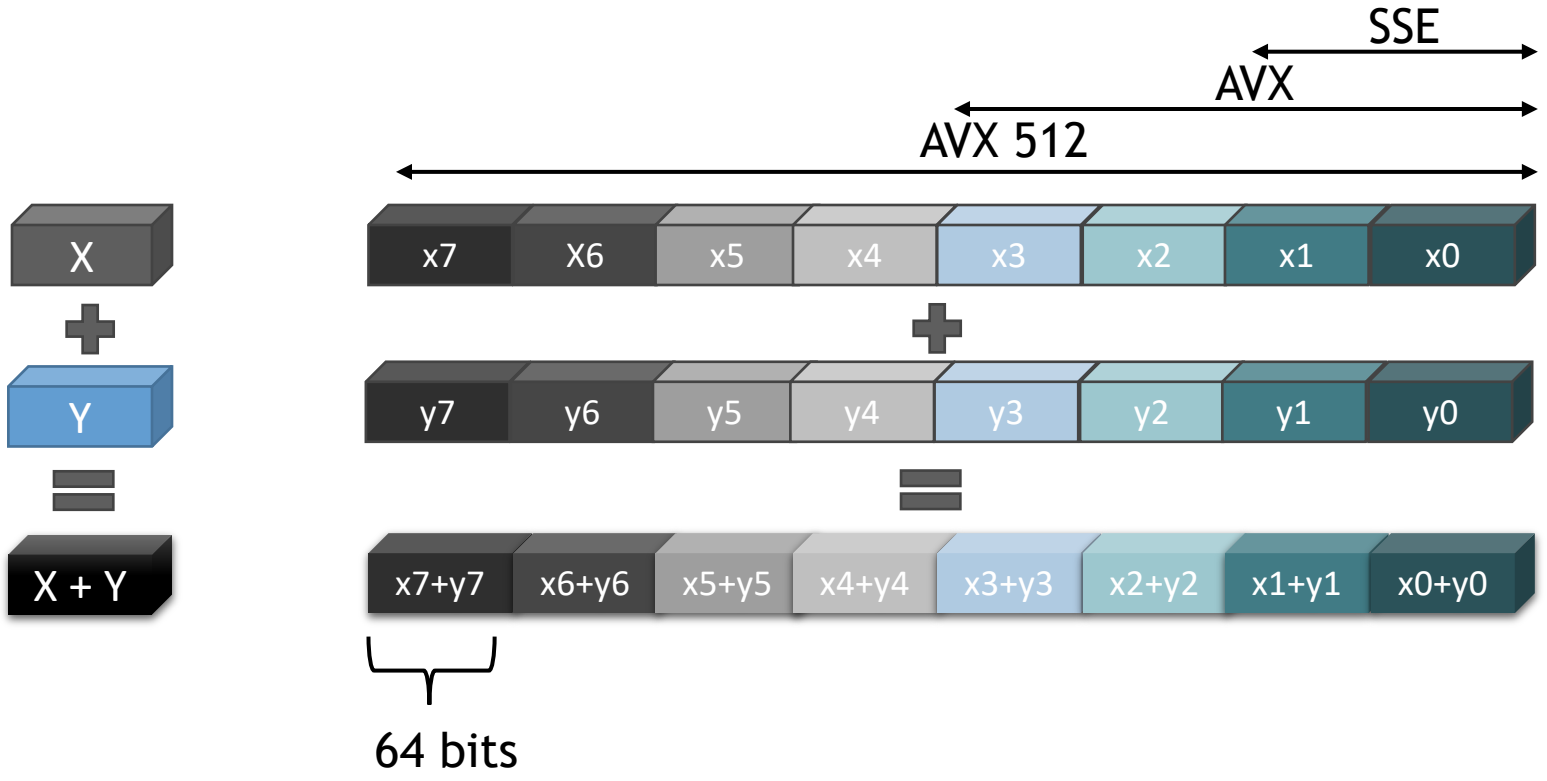
Código vetorizável

- Muitos algoritmos dependem de vetores ou matrizes
  - Podem facilmente ser implementados usando instruções vetoriais
- Muitos algoritmos dependem da aplicação das mesmas operações a grupos de valores
  - Agrupamos estes valores em vetores

Hoje em dia

- Instruções vetoriais são incorporadas nas arquiteturas tradicionais
  - Intel/AMD IA-32 e x64 têm as extensões:
    - MMX, SSE, SSE2, SSE3, SSE4, AVX, …
  - IBM/Freescale PowerPC tem:
    - AltiVec
  - Arcon ARM tem
    - NEON
  - Etc...

Intel AVX


SSE
AVX
AVX 512
X
Y
X + Y
x7
X6
x5
x4
x3
x2
x1
x0
y7
y6
y5
y4
y3
y2
y1
y0
x7+y7
x6+y6
x5+y5
x4+y4
x3+y3
x2+y2
x1+y1
x0+y0
64 bits

SIMD vs SISD

- As instruções vectoriais são um exemplo do modelo:

  SIMD - **Single Instruction Multiple Data**

  Uma instrução opera sobre múltiplos dados produzindo múltiplos resultados

- Por oposição do tradicional:

  SISD - **Single Instruction Single Data**

  Uma instrução opera sobre um ou dois dados produzindo um resultado

Outras combinações

- MISD - **Multiple Instruction Single Data**

  Múltiplas instruções operam sobre os mesmos dados e produzem um único resultado

  - Exemplo: Redundância

- MIMD - **Multiple Instruction Multiple Data**

  Múltiplas instruções operam sobre múltiplos dados e produzem múltiplos resultados

  - Arquiteturas paralelas / Sistemas distribuídos
    - Muitas classes destes sistemas …

Taxonomia de Flynn

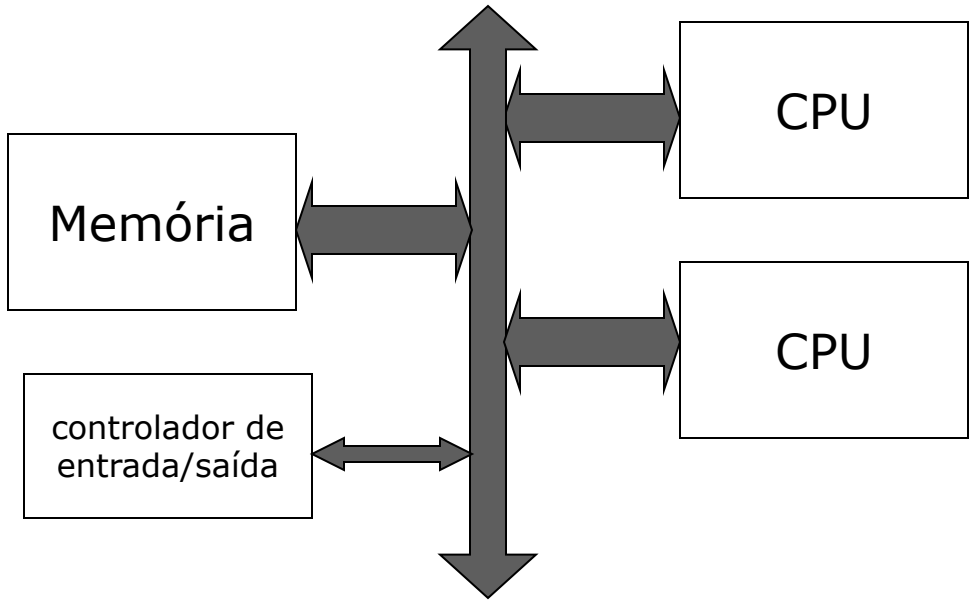
- Procura classificar todas as arquiteturas de computadores com base no processamento das instruções e dos dados

	uma instrução	Múltiplas instruções
um resultado	**SISD**	**MISD**
Múltiplos resultados	**SIMD**	**MIMD**

Exemplo de arquitectura do tipo MIMD

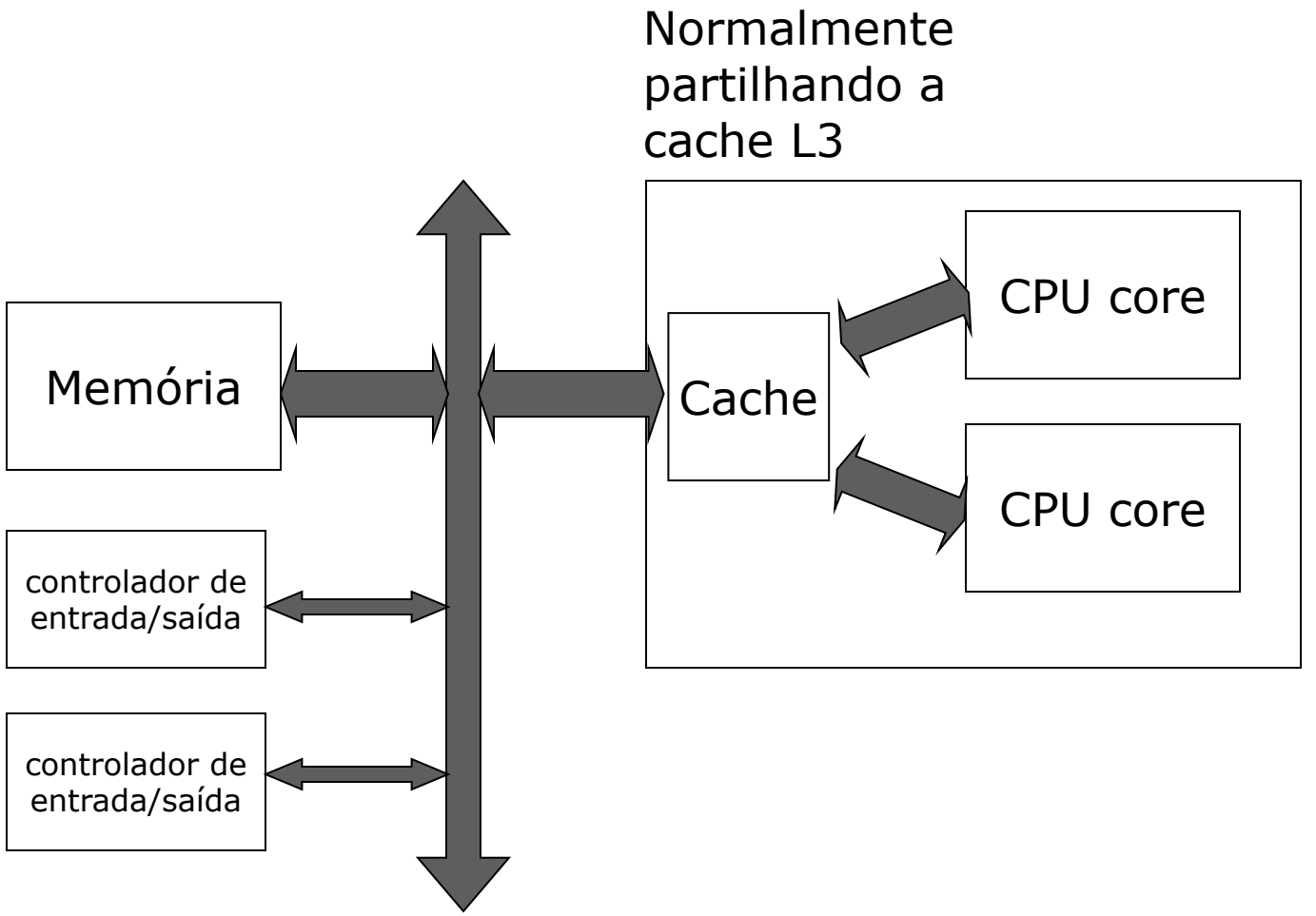
- MIMD - Multiple Instruction Multiple Data
  - Com memória partilhada: todas as unidades processadoras partilham a mesma memória

→ SMP - Symmetric Multiprocessors



Variante: Multicore

- Processador com vários núcleos (cores) de processamento
  - Pipeline e unidades de processamento próprias
  - Caches próprias
    - Normalmente L1 e L2
    - L3 partilhada por todos ou alguns cores

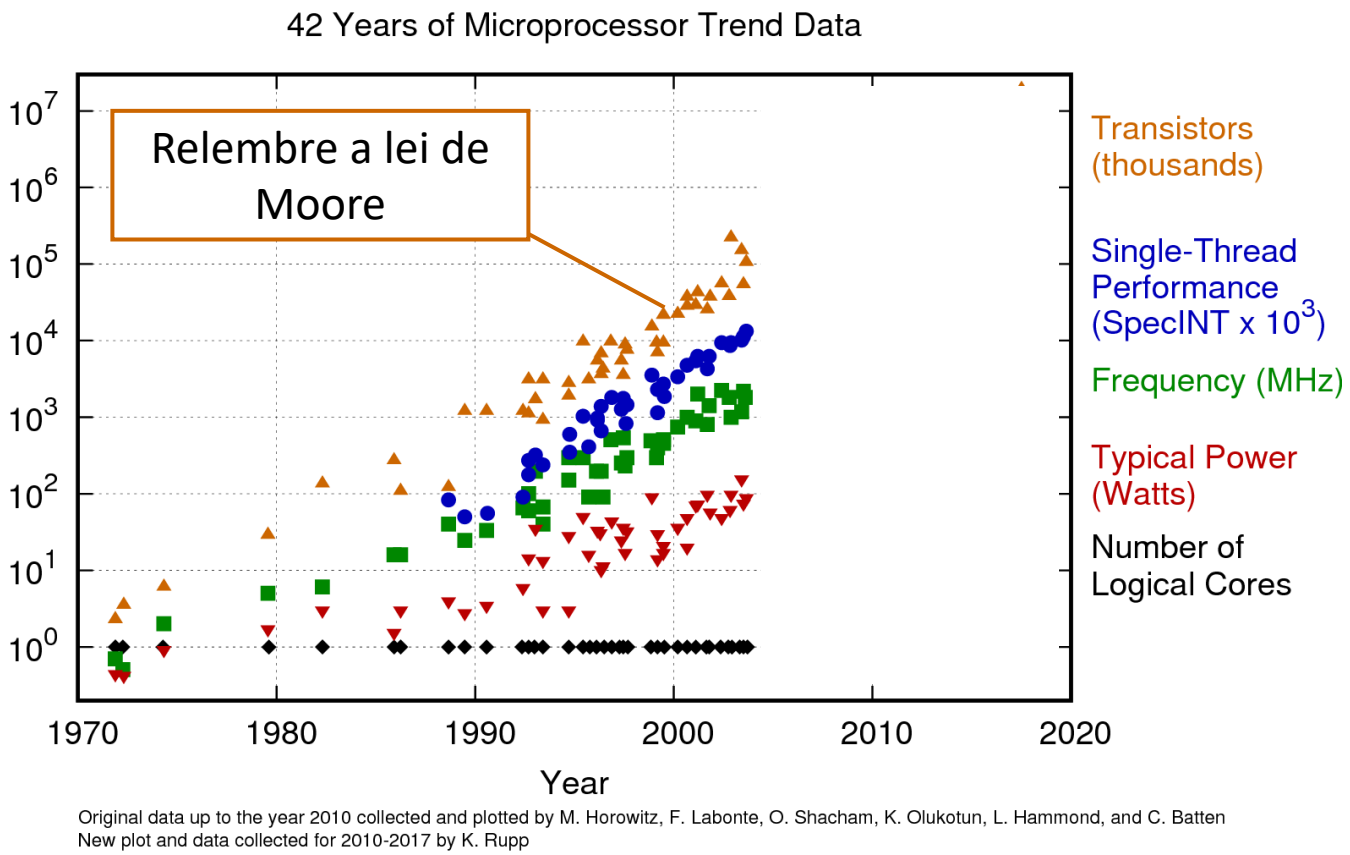


CPUs para o SO

- Um Core pode ser hyperthreaded
- O número de fluxos de execução (threads) que um computador pode executar em simultâneo dá-se o nome de **hardware thread**
- Do ponto de vista do SO, um hardware thread é apresentado como um CPU (seja este físico, como um Core, ou lógico, como no caso de hyperthreading)

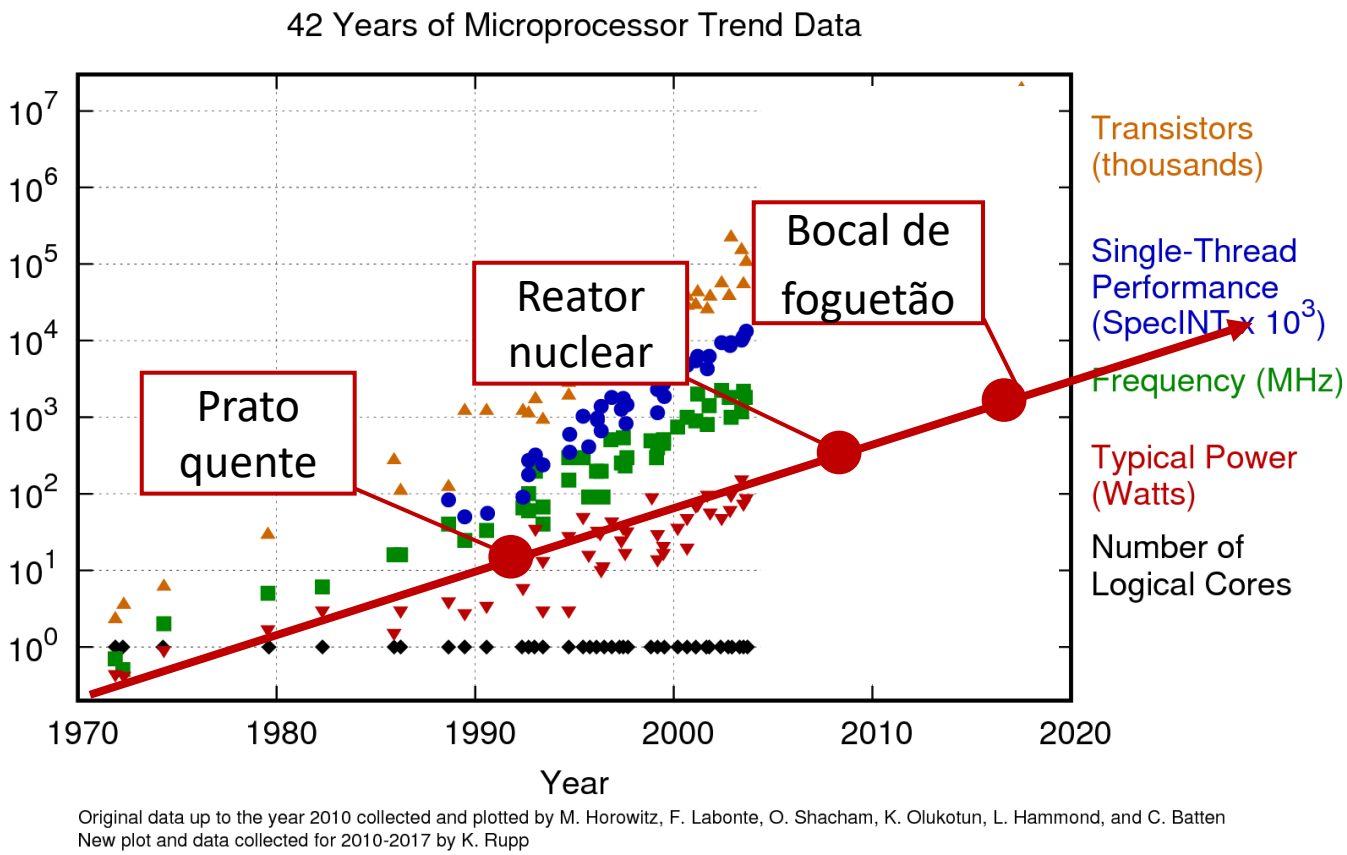
	Cores físicos	Cores lógicos	Hardware threads	CPUs para o SO
1 CPUs com 1 core	1	0	1	1
1 CPU com 4 cores	4	0	4	4
1 CPU com 2 cores hyperthreaded	2	2	4	4
1 CPU com 4 cores hyperthreaded	4	4	8	8
2 CPU com 4 cores hyperthreaded	8	8	16	16

A Revolução Multicore



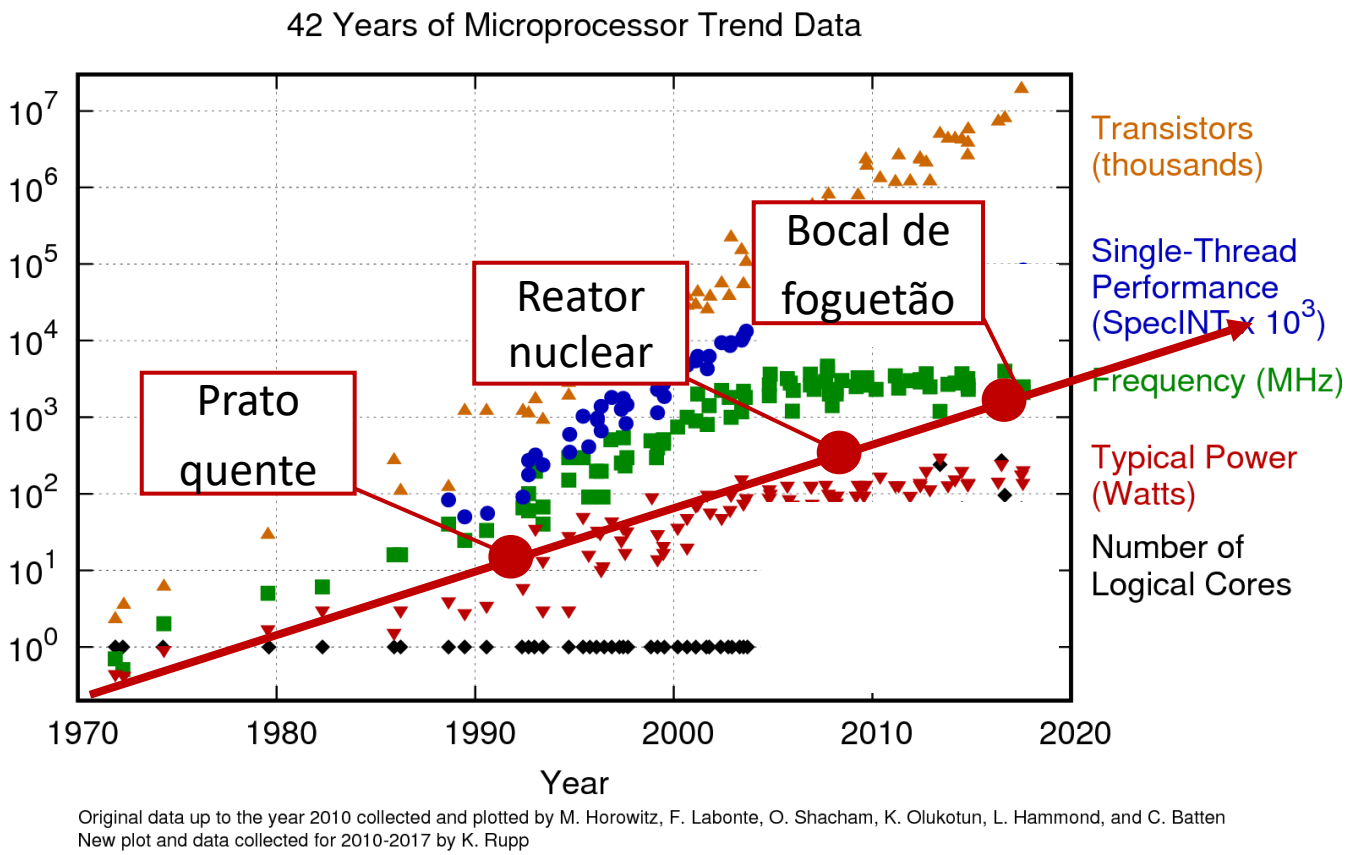
Taken from https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

A Revolução Multicore



Taken from https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

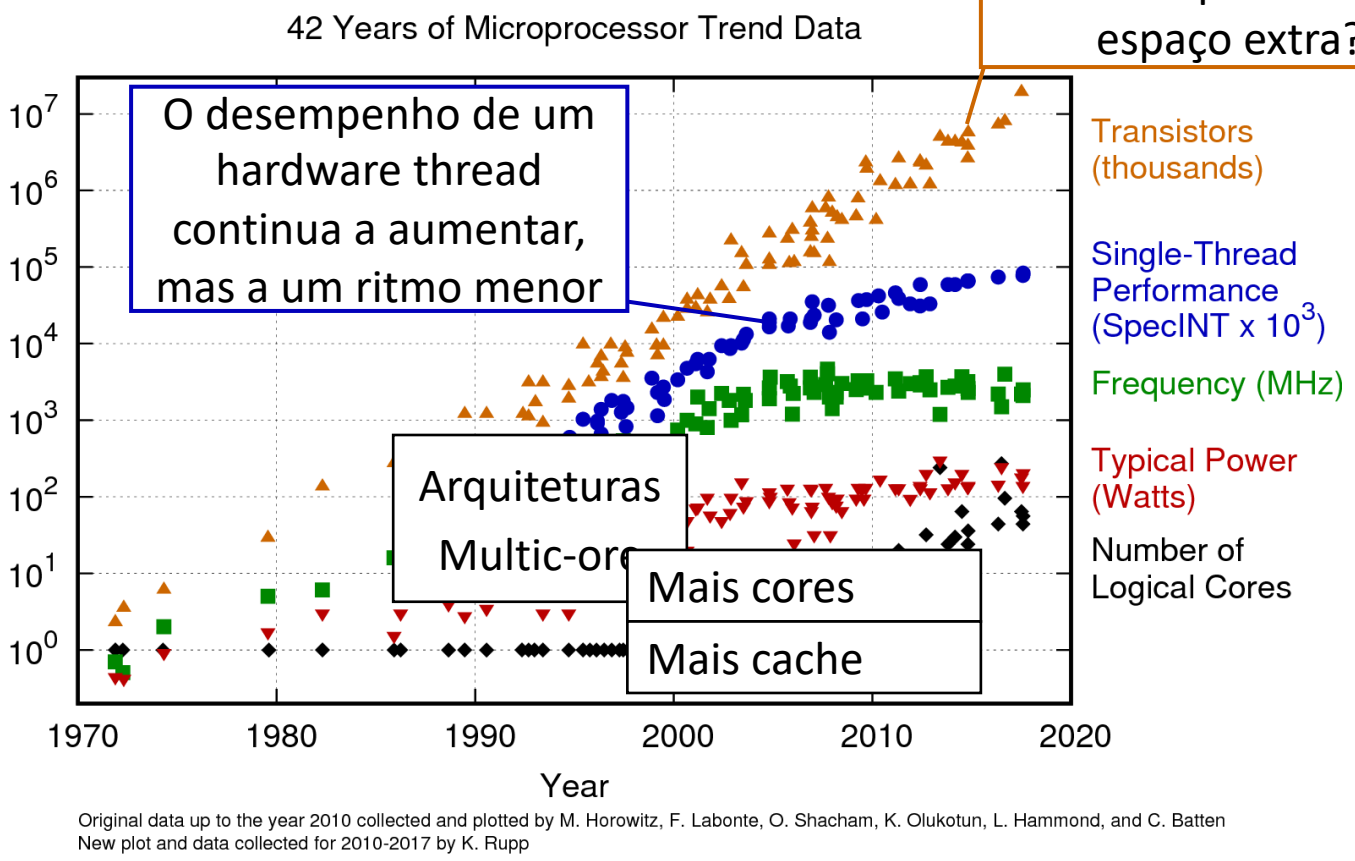
Reason 1 — Processor Evolution



Taken from https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Reason 1 — Processor Evolution

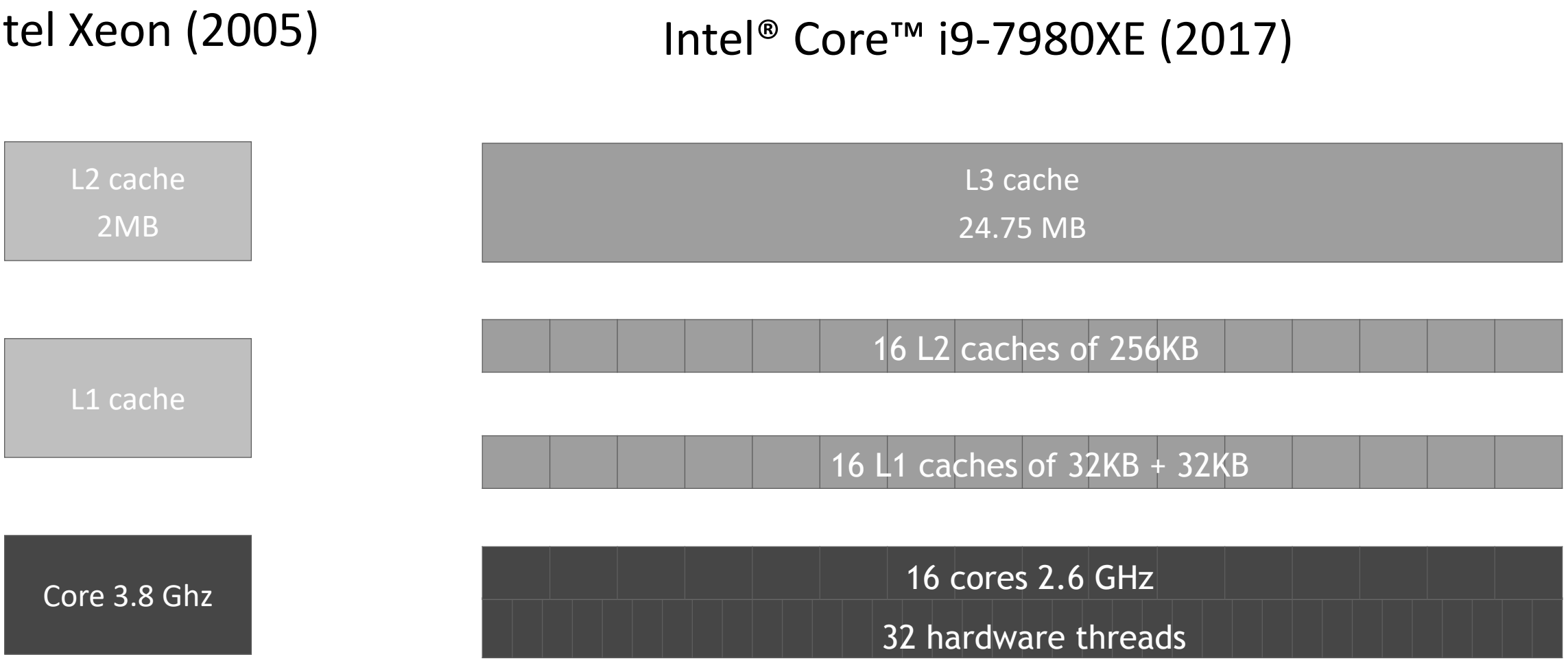
A lei de Moore continua a ser válida. Como aproveitar o espaço extra?



A programação sequencial, por si só, não chega para se extrair todo o potencial do hardware

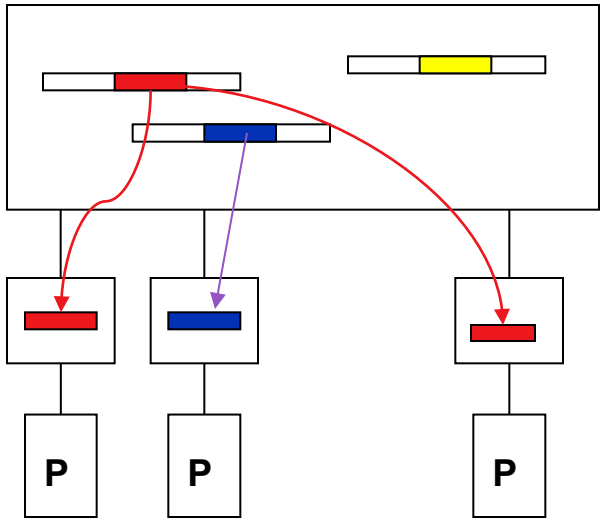
Taken from https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Exemplo de Evolução dos CPUs



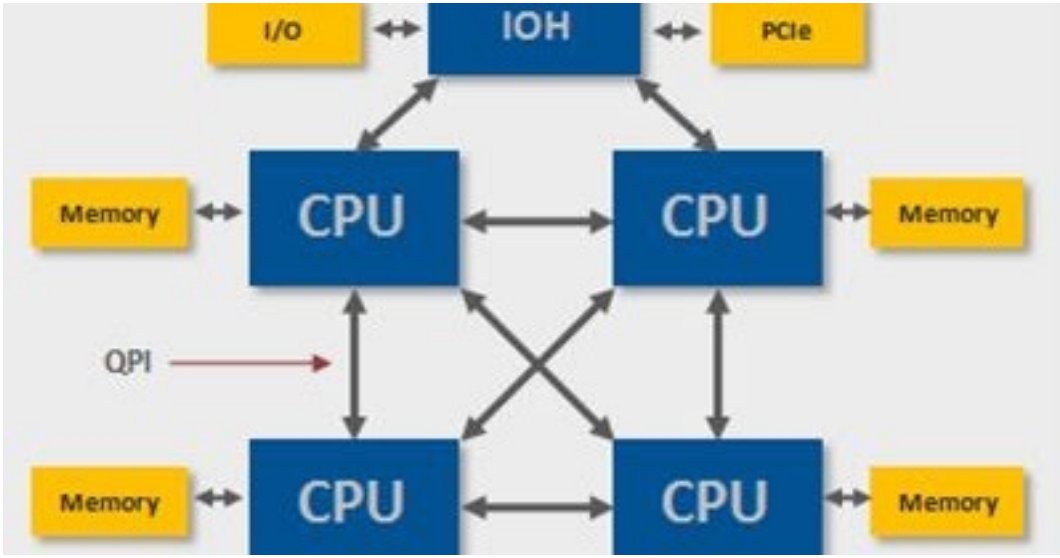
Caching em Arquiteturas de Memória Partilhada

- O que é que acontece quando leituras e escritas são efetuadas em processadores diferentes?
  - **Problemas de coerência das caches**
  - Para resolver estes problema, as caches implementam um **protocolo de coerência**
    - Existem vários algoritmos
    - A ideia é garantir que: se há uma escrita numa linha da cache que guarda um bloco B, todas as restantes caches que também têm uma cópia do bloco B têm o mesmo **atualizado** ou **invalidado**



Non-Uniform Memory Access (NUMA) SMPs

- Memória distribuída
- Cada conjunto de CPUs (pode ser apenas 1) partilham fisicamente uma memória
- A memória total ainda é logicamente partilhada por todos os CPUs
- Acesso não uniforme à memória (NUMA)
  - Memória local e memória remota
- O acesso à memória local é mais rápido, a memória remota mais lenta
  - O acesso não é uniforme
- O desempenho dependerá da localidade dos dados
- A coerência da cache ainda é um problema (mais sério)

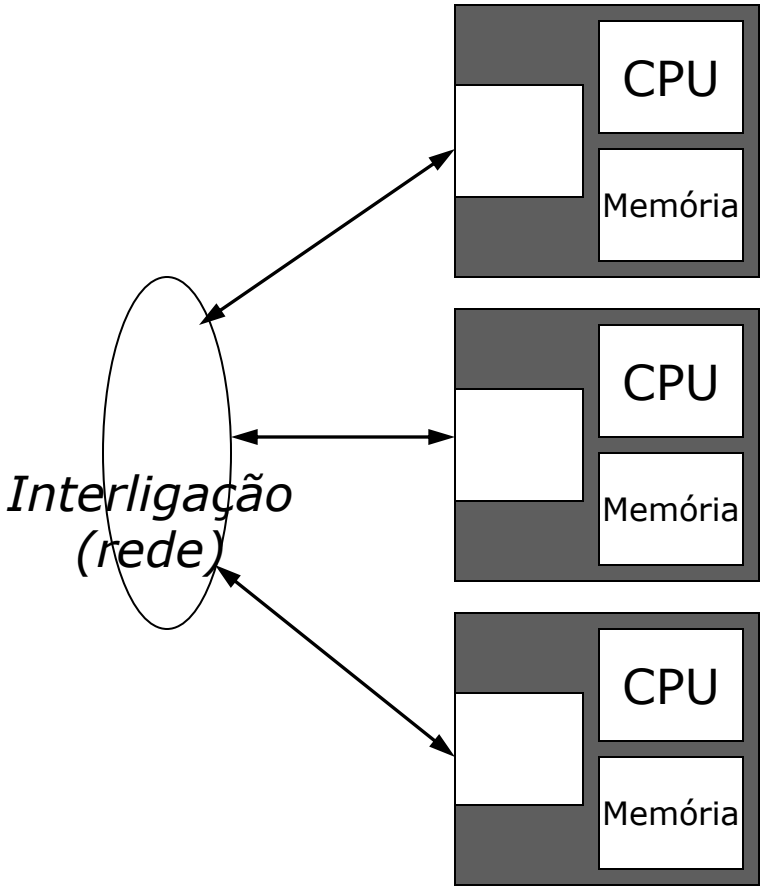


MIMD de Memória Partilhada

- Vantagens
  - Todos os CPU partilham o programa e os dados
  - Fácil e eficiente partilhar as alterações nos dados
  - O SO tira partido destas arquiteturas
- Desvantagens
  - Congestionamento no acesso à memória
  - As caches L1 e L2 separadas por CPU-core podem aliviar este problema, mas colocam o problema das escritas poderem tornar as caches incoerentes
  - Impraticável para grande escala (muitos CPUs)

Mais arquiteturas do tipo MIMD

- MIMD - Multiple Instruction Multiple Data
  - Sem memória partilhada: cada unidade processadora tem a sua própria memória
  - Instanciação comum: **cluster**
    - Conjunto de computadores com hardware "comuns" ligados por redes rápidas
    - Cluster do DI: https://cluster.di.fct.unl.pt
  - Outras:
    - Supercomputadores
    - Clusters de computadores pessoais



MIMD de Memória Distribuída

- Vantagens
  - Podemos conseguir arquiteturas com milhares de CPUs
  - Permite a ligação de equipamentos fisicamente distribuídos

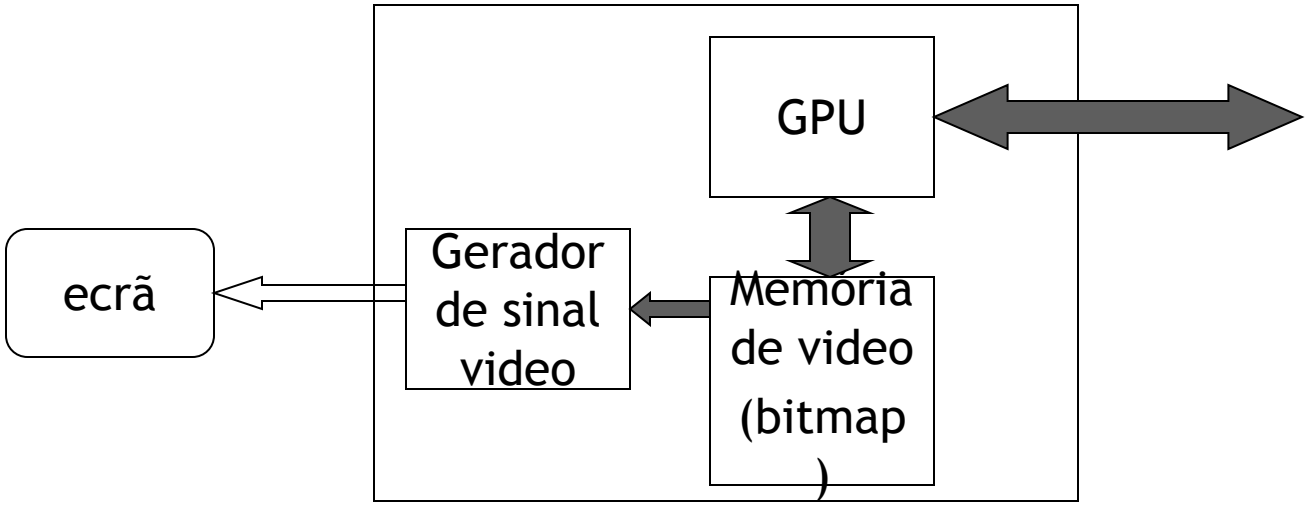
- Desvantagens
  - Interligações “lentas e de grande latência”
  - Não suportam bem muitas dependências entre dados
  - Difícil tirar o melhor desempenho destes sistemas, de os usar e de os gerir

Como tirar partido dos multicore

- Os multicore, como as arquiteturas MIMD em geral, são úteis quando:
  - Executamos vários processos em simultâneo
    - O escalonamento fica a cargo do SO ou de um outro gestor de recursos (usual em clusters)
  - Decompomos um programa em vários fluxos de execução que possam sejam executados em simultâneo:
    - Noção de processo e thread a ser estudado em FSO
    - Decomposição de um programa que possam sejam executado em simultâneo a ser estudado em Concorrência e Paralelismo
    - Programação paralela avançada em Computação de Alto Desempenho

Coprocessador gráfico

- Por volta de 1987 o controlador gráfico passa a ser “acelerado”
  - Inclui uma unidade co-processadora que ajuda o CPU na manipulação de gráficos
  - GPU (Graphics Processing Unit) – Unidades de Processamento Gráfico

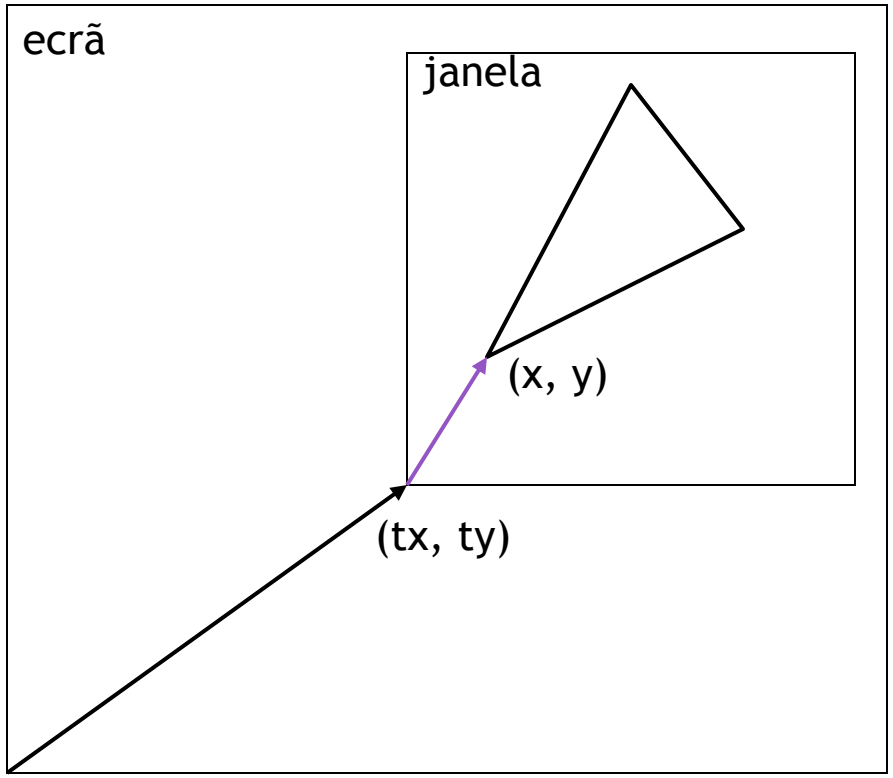


GPUs

- GPUs (Graphics Processing Units) – Unidades de Processamento Gráfico
  - Primeira aparição em PCs por volta de 1987
  - Início do anos 90: APIs para processamento gráfico Direct3D and OpenGL

“Aceleração” 2D

- O GPU suporta comandos para desenho 2D
  - Em vez do CPU “pintar” os pixeis, manda a GPU fazer essas operações
    - Poupa o CPU e o BUS
  - Exemplos de comandos suportados: linhas, figuras geométricas (elipses, polígonos, etc), e também copia/move zonas da imagem…
  - Estas operações podem também incluir operações de translação, rotação e escala
    - Operações vectoriais



Posição do triângulo na janela (x,y)

Posição da janela (tx, ty)

Posição do trângulo no ecrã (x+tx, y+ty)

Fases no desenho 3D

- Descrição da cena
  - Os objectos 3D.
    - Exemplo: por aproximação de triângulos 3D
  - Descrição das superfícies:
    - Cor e textura
  - Descrição da cena:
    - Posição dos objectos e da iluminação
- Projectar a cena 3D no plano do ecrã (2D)
  - Modelo da máquina fotográfica

  (resolver: posições de cada objecto e o aspecto das partes visíveis)

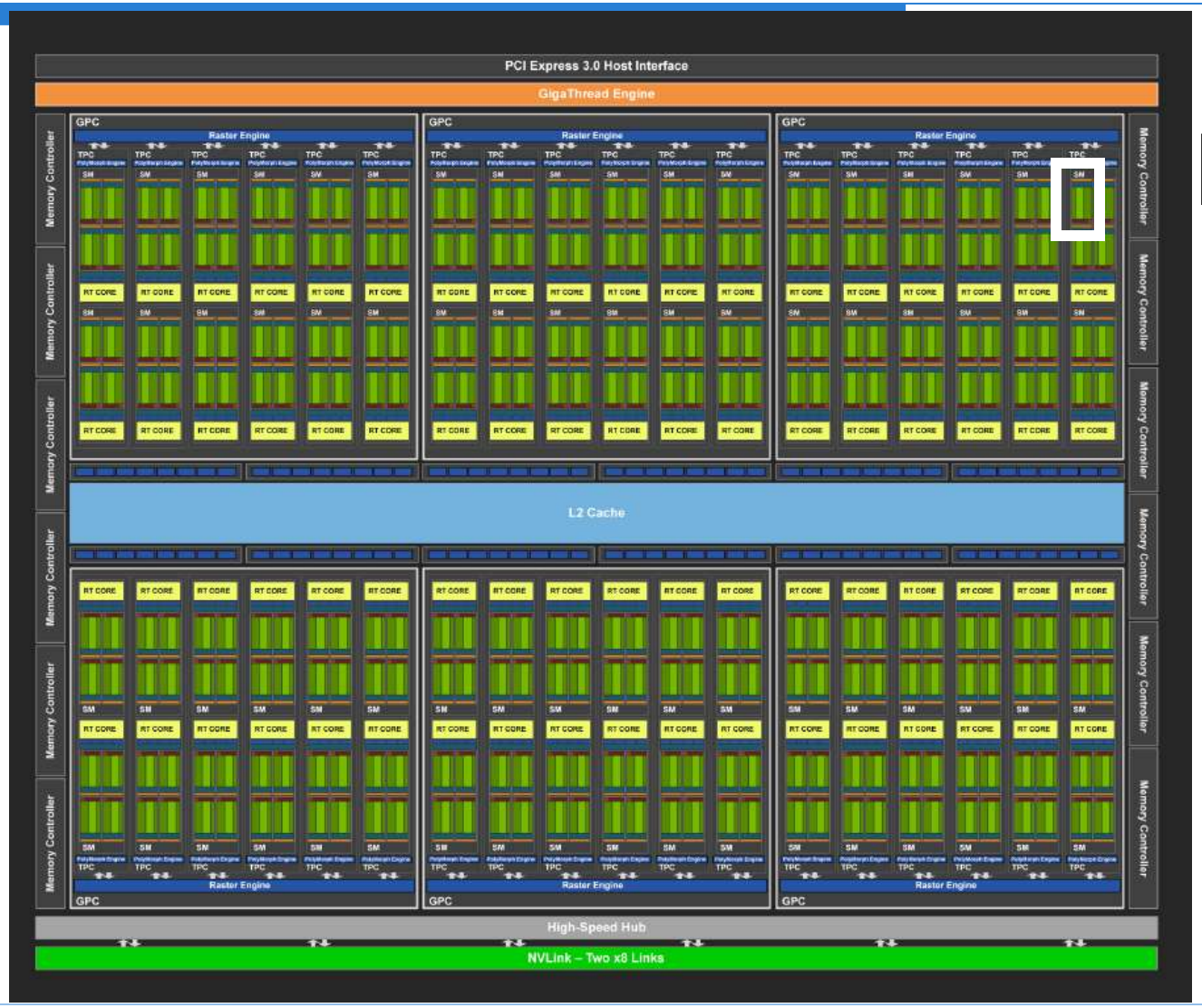
“Aceleração” 3D

- O GPU receba a descrição de toda a cena
- O GPU implementa um pipeline de operações:
  1. Aplica todas as transformações nos objetos para os posicionar e efetua a transformação 3D para 2D
     - Translações, rotações e escala
  2. Calcula o “aspeto” de cada face, tendo em conta as características dos objecos e da luz que lhe incide
  3. Identificando as partes visíveis e calculando a cor de cada pixel no ecrã

GPUs Programáveis

- GPUs (Graphics Processing Units) – Unidades de Processamento Gráfico
  - Primeira aparição em PCs por volta de 1987
  - Início do anos 90: APIs para processamento gráfico Direct3D and OpenGL
  - Início dos anos 2000: Shaders programáveis
  - Meio dos anos 2000: Unified Shading Architecture
    - Passou-se de hardware com uma função fixa para hardware programável (ou seja, que executa programas)

Pascal GP100 – 60 SM Units

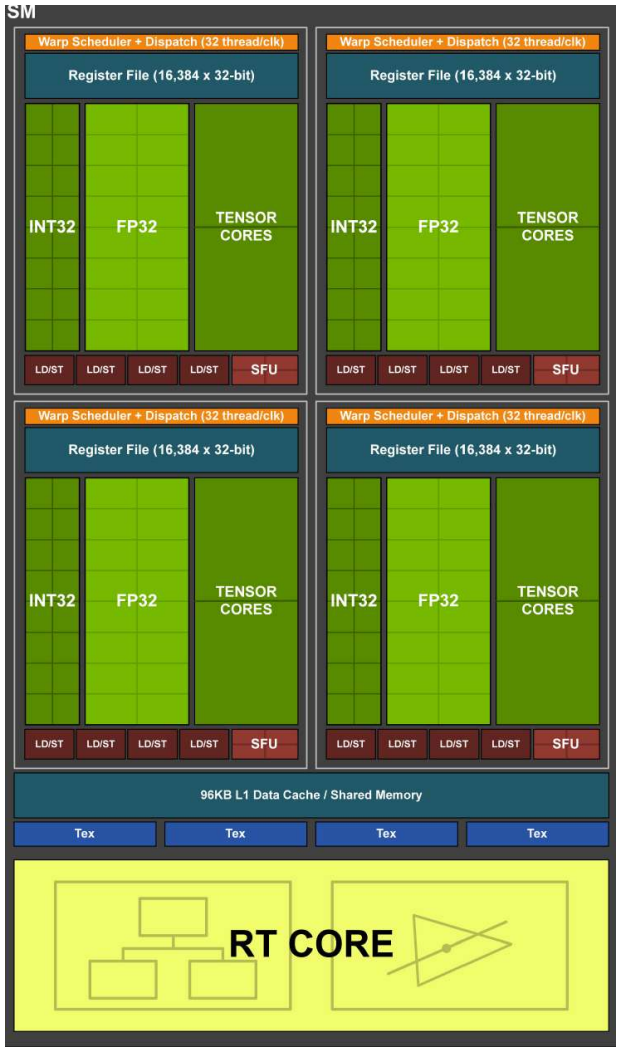


Streaming Multiprocessor (SM)

Pascal SM

Contém

- ALUs – INT32
- FPUs – FP32
- Tensor Cores
- SFU – operações especiais, como raiz quadrada, seno, ...



GPGPU

- GPGPU (General Purpose Computing on GPUs)
  - Usar GPUs para computação não gráfica
  - Termo proposta por Mark Harris em 2002

GPGPU

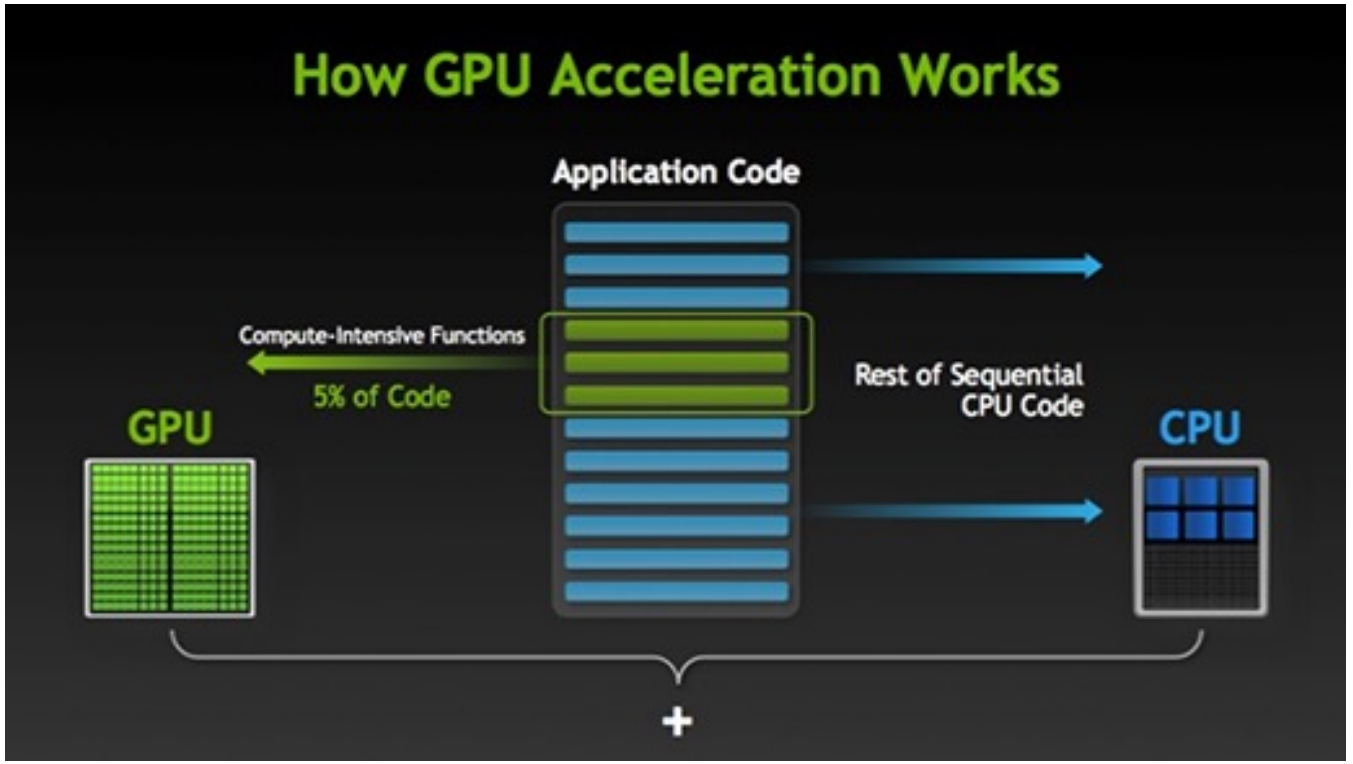


Image taken from http://www.nvidia.com/object/what-is-gpu-computing.html

CPU versus GPUs

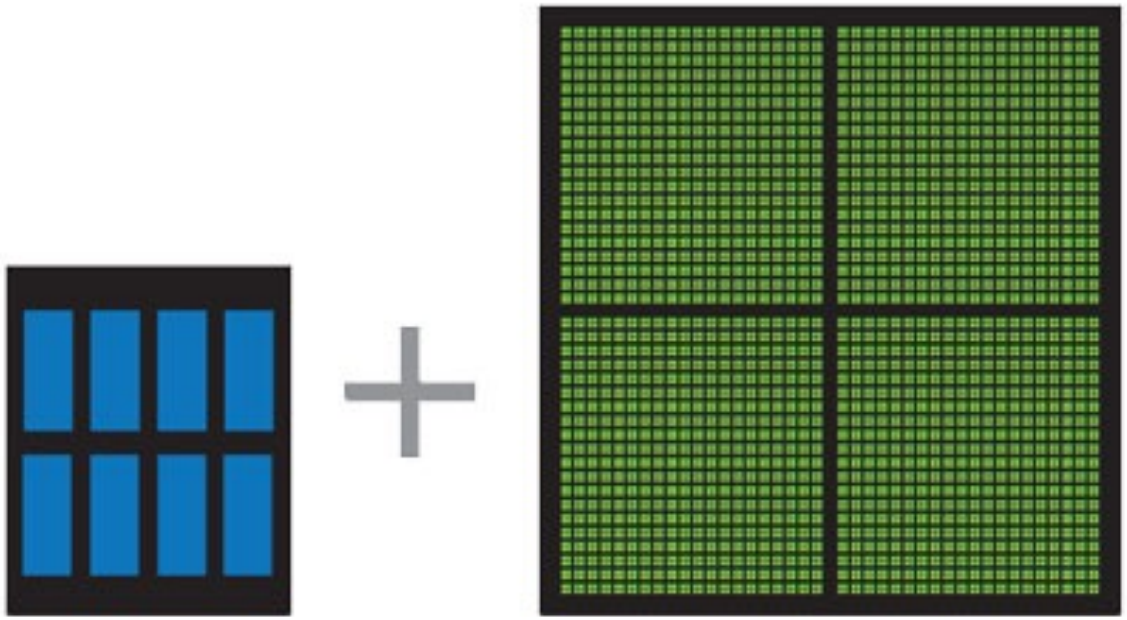
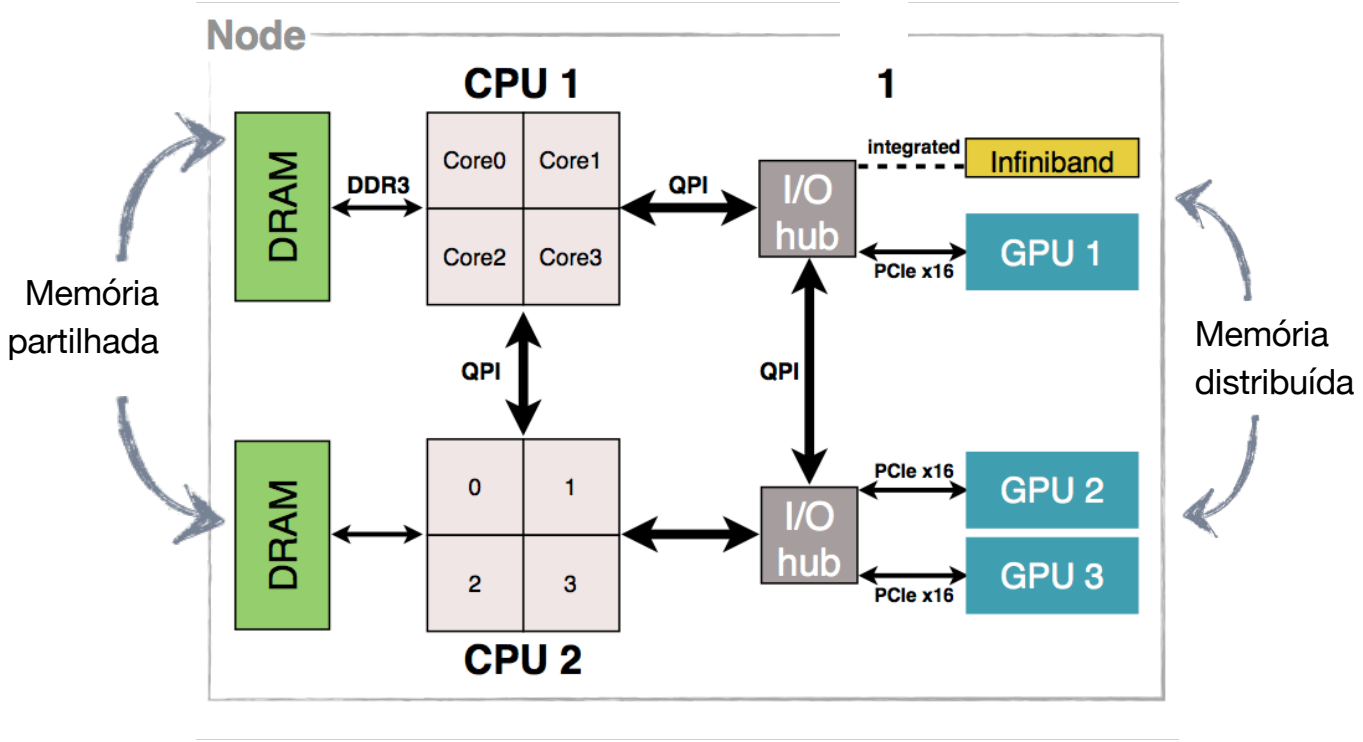


Image taken from http://www.nvidia.com/object/what-is-gpu-computing.html

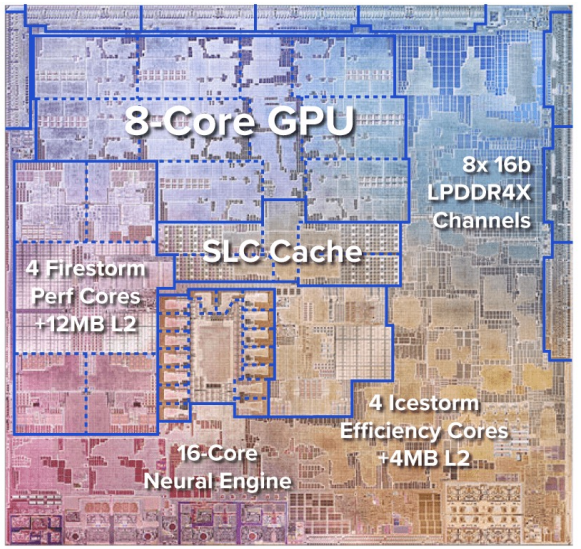
Multicore CPUs Ligados a GPUs

- GPU normalmente ligado ao bus PCIe (PCI Express)

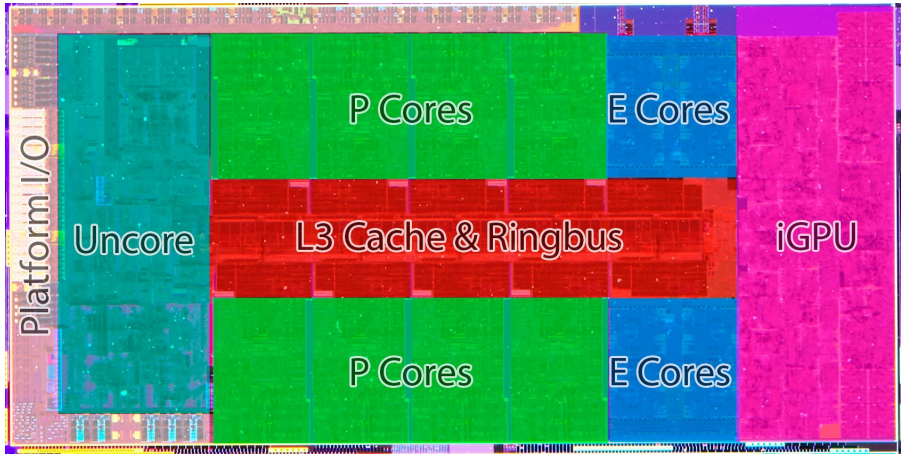


Exemplos de Processadores com GPUs Integrados

- ARM M1
  - Cores de desempenho
  - Cores eficientes energeticamente
  - Cores para machine learning (neural engine)
  - GPU
  - Caches



- Intel Core I9 12900K
  - Cores de desempenho
  - Cores eficientes energeticamente
  - GPU
  - Caches



Top500.org (Junho 2022)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438