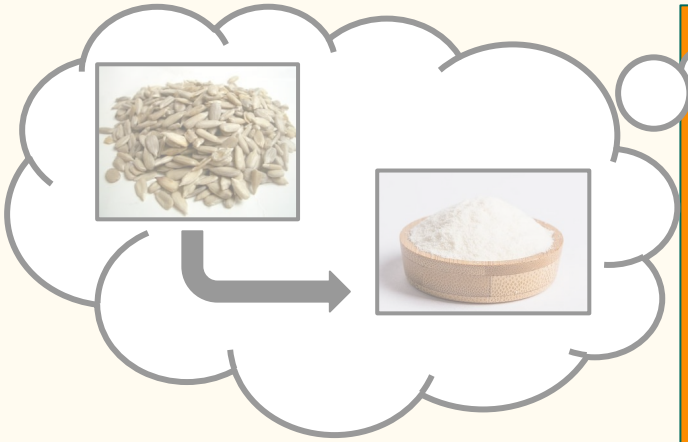
Parte 7

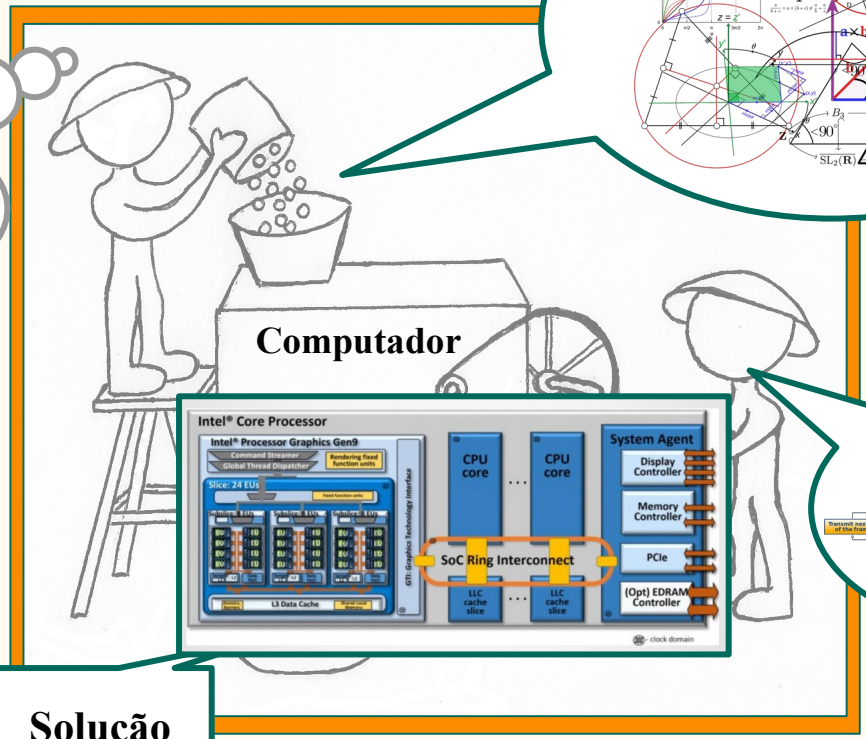
Processadores Modernos

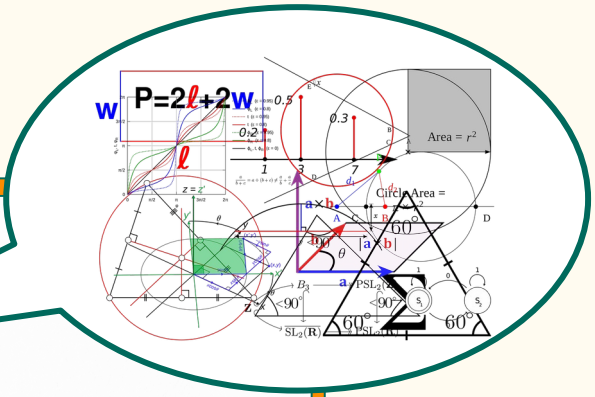
CI1164 - Introdução à Computação Científica
Profs. Armando Delgado e Guilherme Derenievicz
Departamento de Informática - UFPR

Visão geral da disciplina:

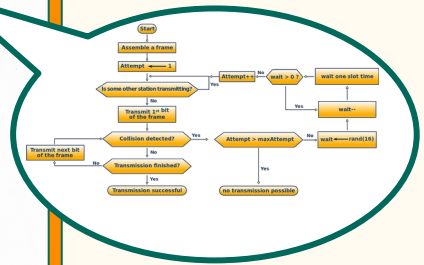


Descrição do Problema





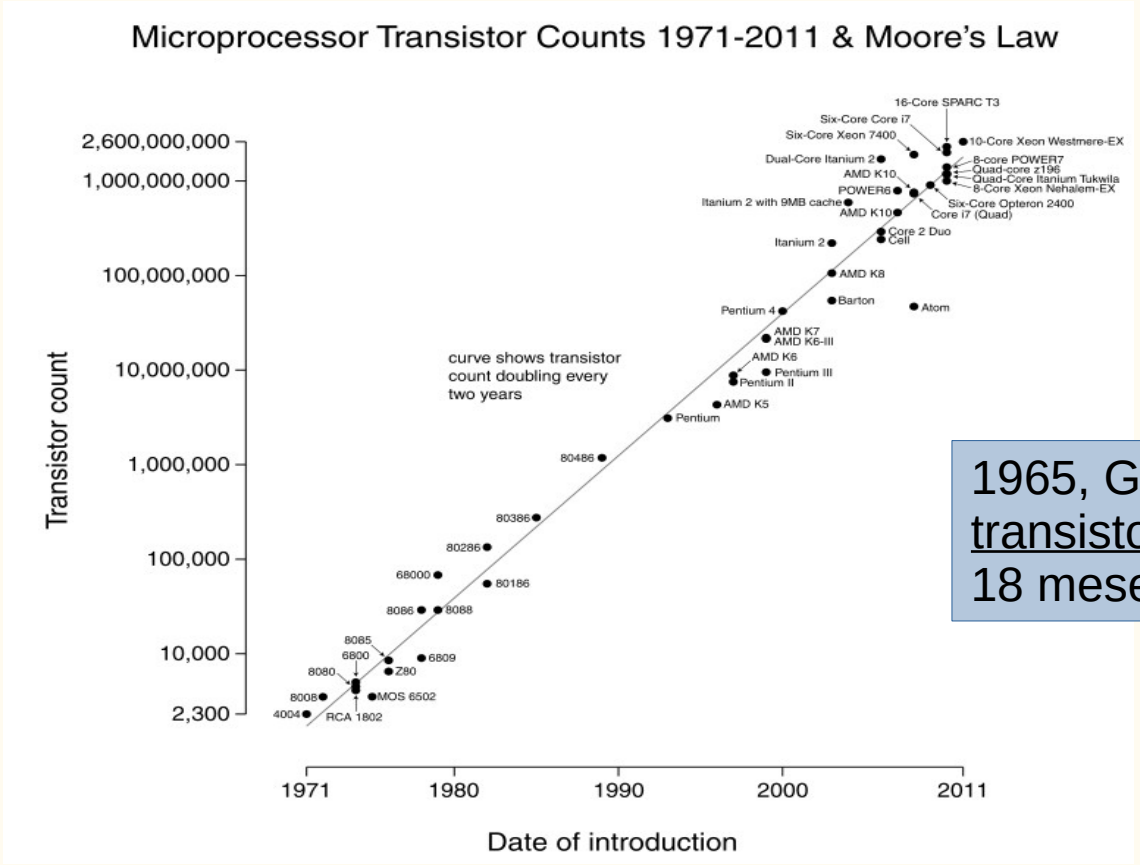
Modelagem



Método Numérico

Solução

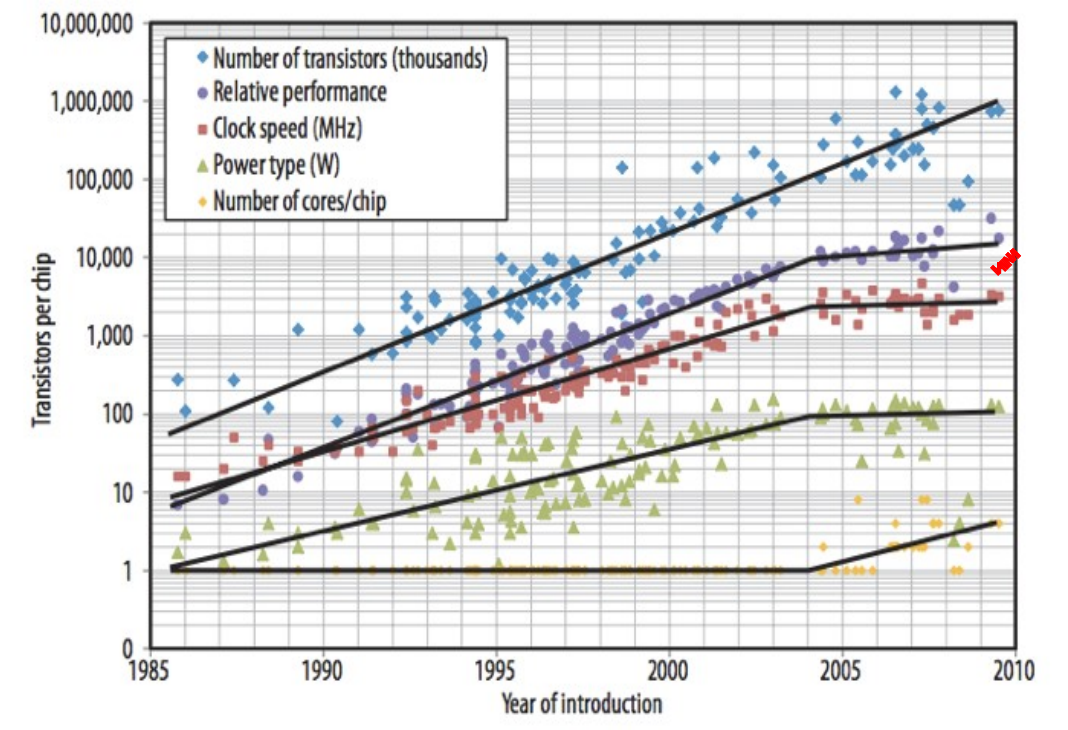
Evolução dos processadores



Nvidia Fermi: ~3.0 bilhões
Nvidia Keppler: ~7.1 bilhões

1965, G. Moore: # de transistores/chip dobra a cada 18 meses

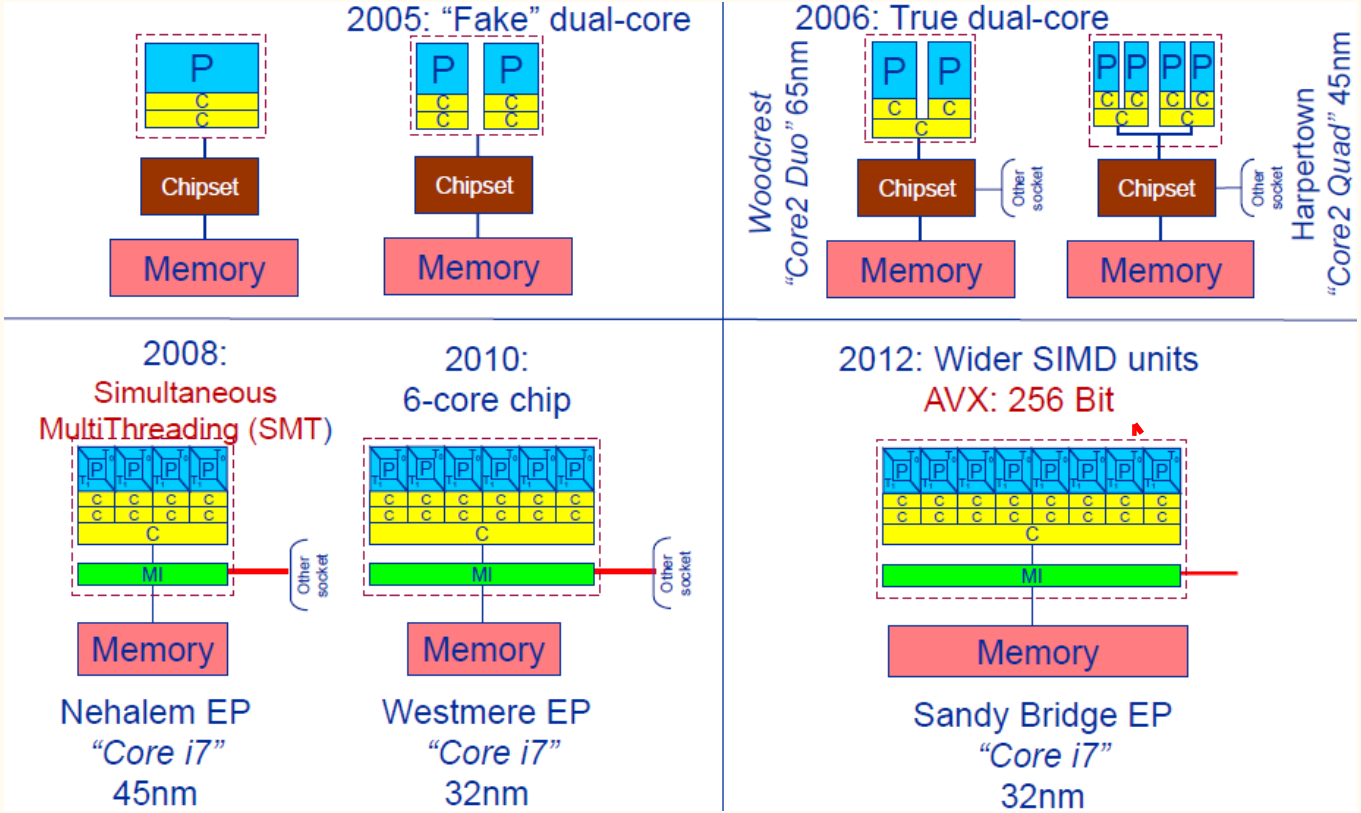
Limitações dos processadores



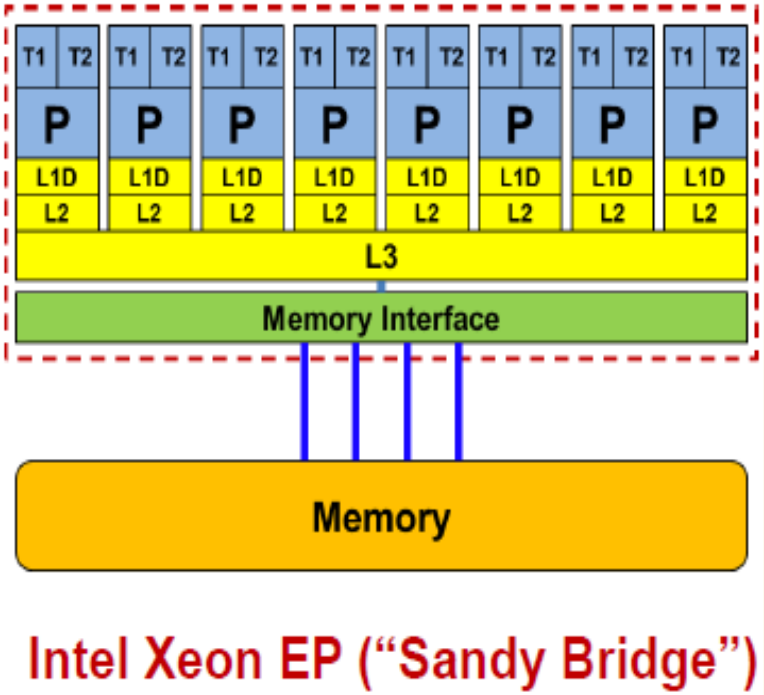
Fonte: http://intelligence.org/2014/05/12/exponential-and-non-exponential/

- Lei de Moore
  - ➔ Transistores menores e mais rápidos
- Clock mais rápidos
  - ➔ Mais operações / seg
- Single Core:
  - ➔ Pipelining,
  - ➔ Paralelismo
  - ➔ SIMD (SSE/AVX)

Evolução de processadores x86 multicore



Fatores de Performance (P)



Intel Xeon EP ("Sandy Bridge")

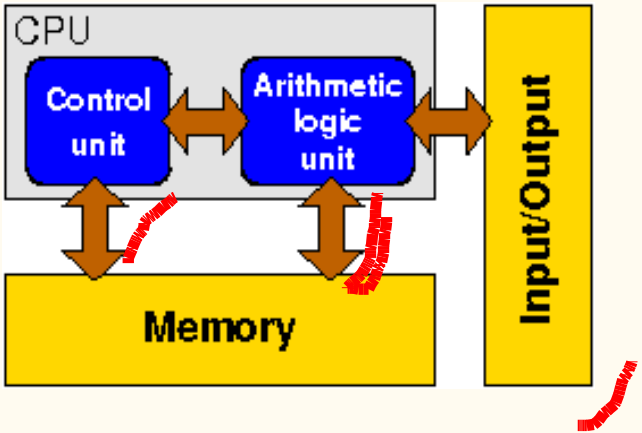
$P = n_{core} \times F \times S \times \nu$

n_{core}	número de cores:	8
F	instruções FP por ciclo:	2
S	FP ops / instrução:	4 (dp) / 8 (sp)
ν	Clock:	2.5 GHz

- P = 160 GF/s (dp) ou 320 GF/s (sp)
- P = 5.0 GF/s (código serial não-vetorizado)

Arquitetura SPC (Stored-program computer)

- Arquiteturas modernas ainda implementam SPC
  - ➔ *Stored Program Computer* (Turing, 1936)

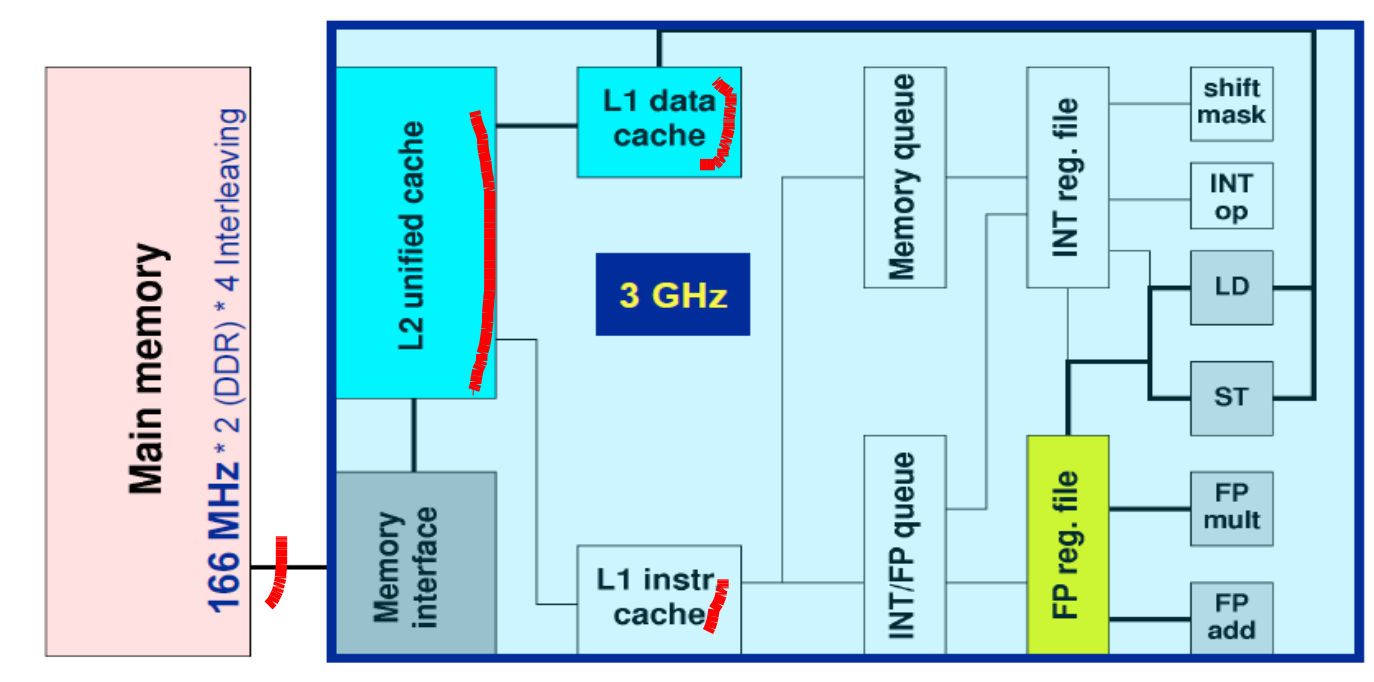


- **Unidade de Controle**: traz instruções da memória e as executa
- **Unidade Lógica e Aritmética** (ULA):
  - Realiza computações sobre dados na memória
- **IO**: comunicação com usuário
- **CPU**: Unidade de Controle + ULA

Dois gargalos identificados desde o início:

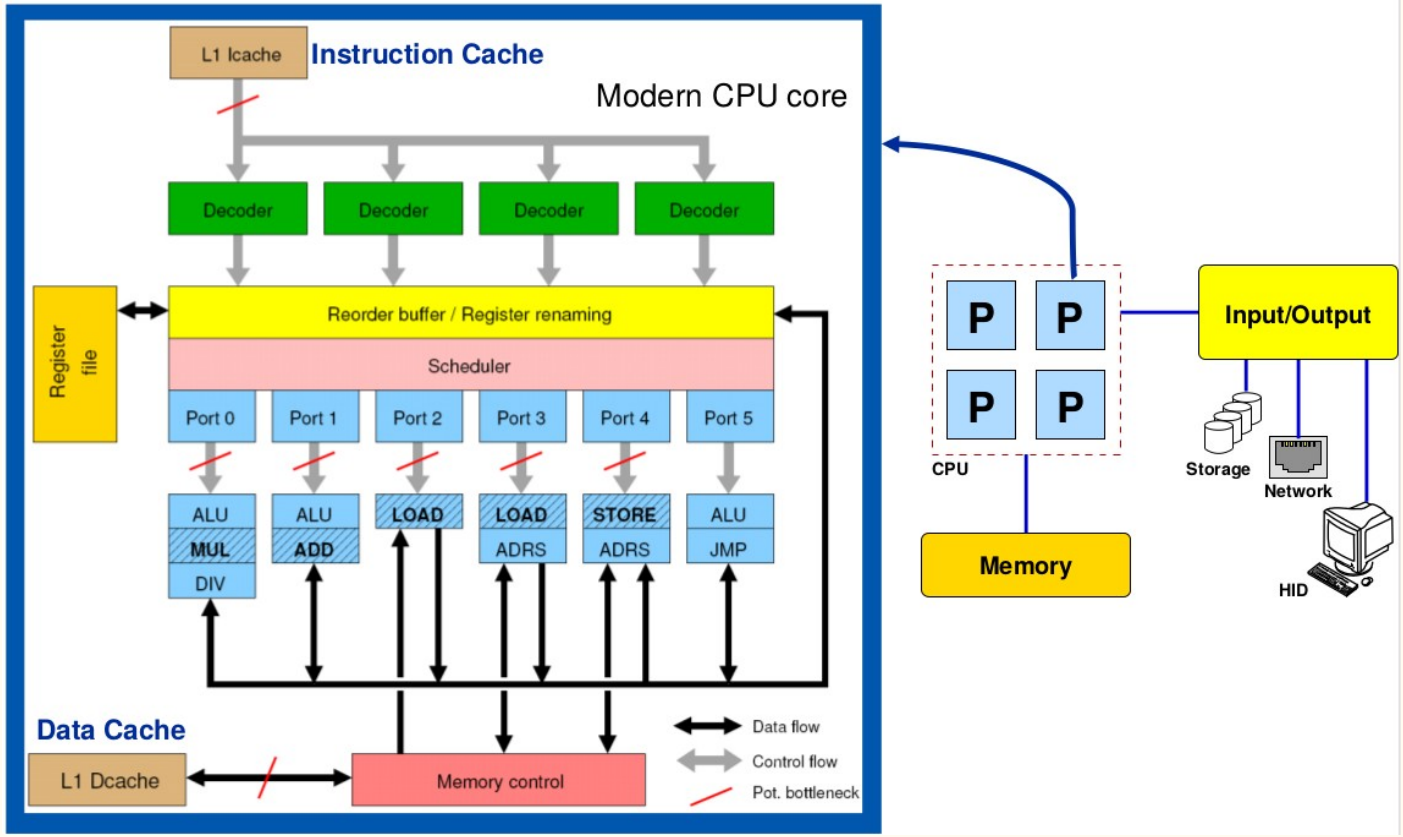
- *von Neumann:* performance limitada pela velocidade de acesso à memória
- Arquitetura inerentemente sequencial

Processadores de propósito geral



Unidade de Controle: busca/decodifica instruções, predição de desvios, ...

Processadores de propósito geral

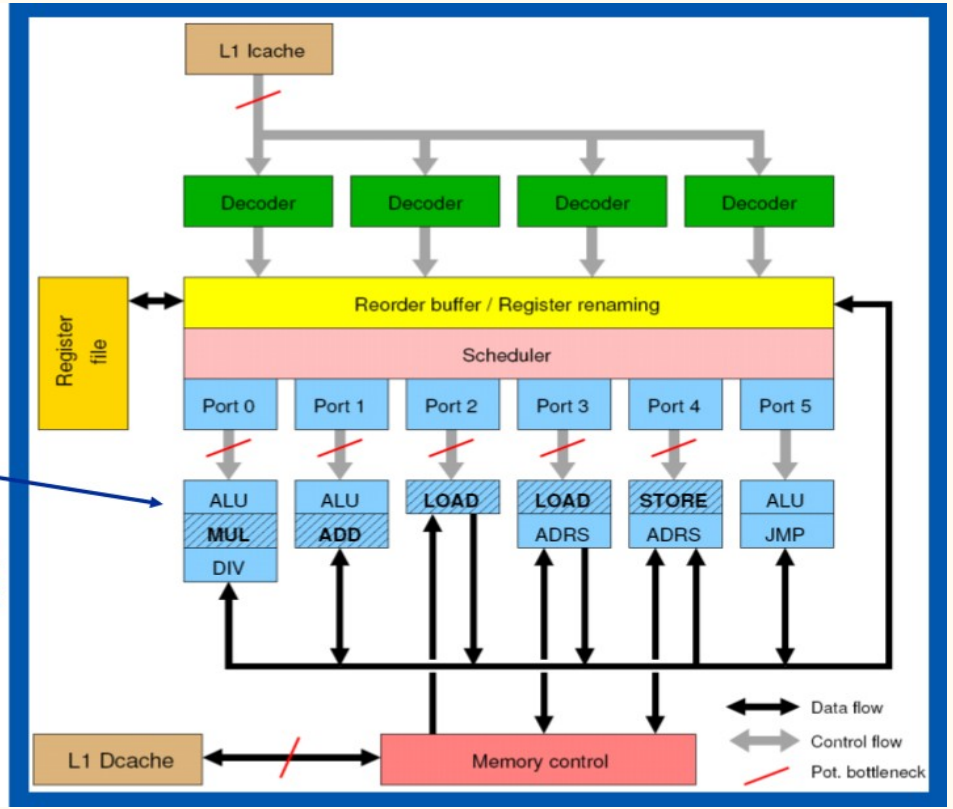


Pipelining

Pipelining

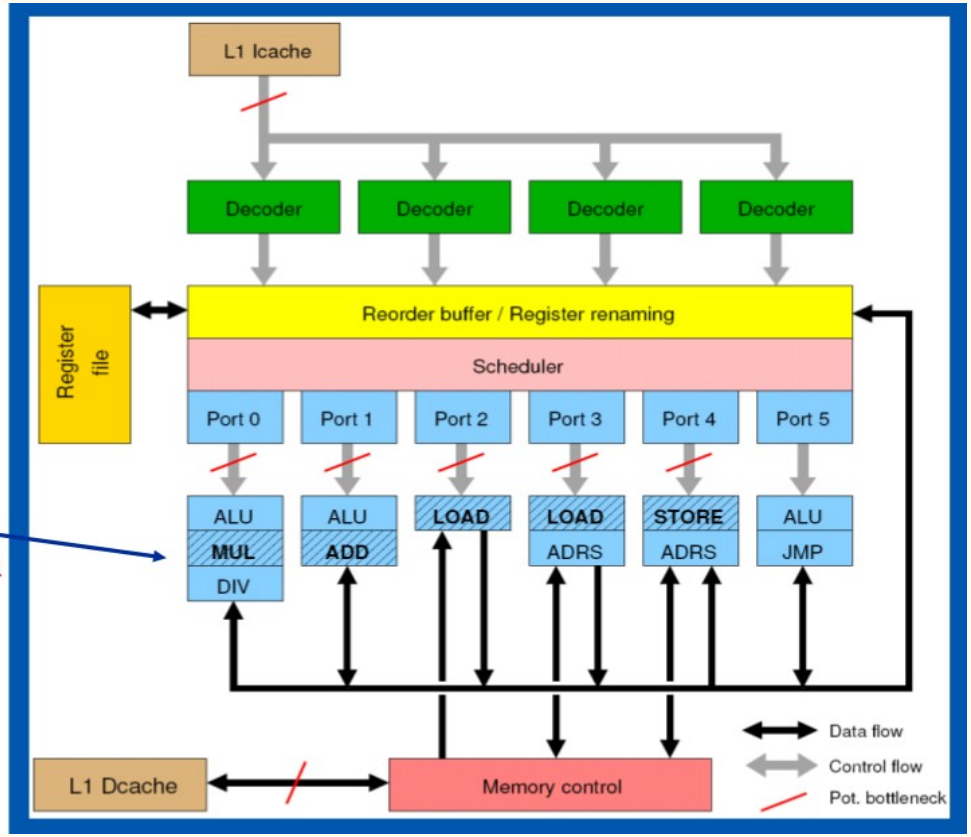
Many units can complete one instruction per cycle, e.g. MULT / ADD / LOAD /STORE

Focus on: Floating Point Instructions/Operations



SIMD: Single Instruction Multiple Data

SIMD:
Single Instruction Multiple Data
Instruction is applied to multiple operands in parallel („width of execution units/registers“)



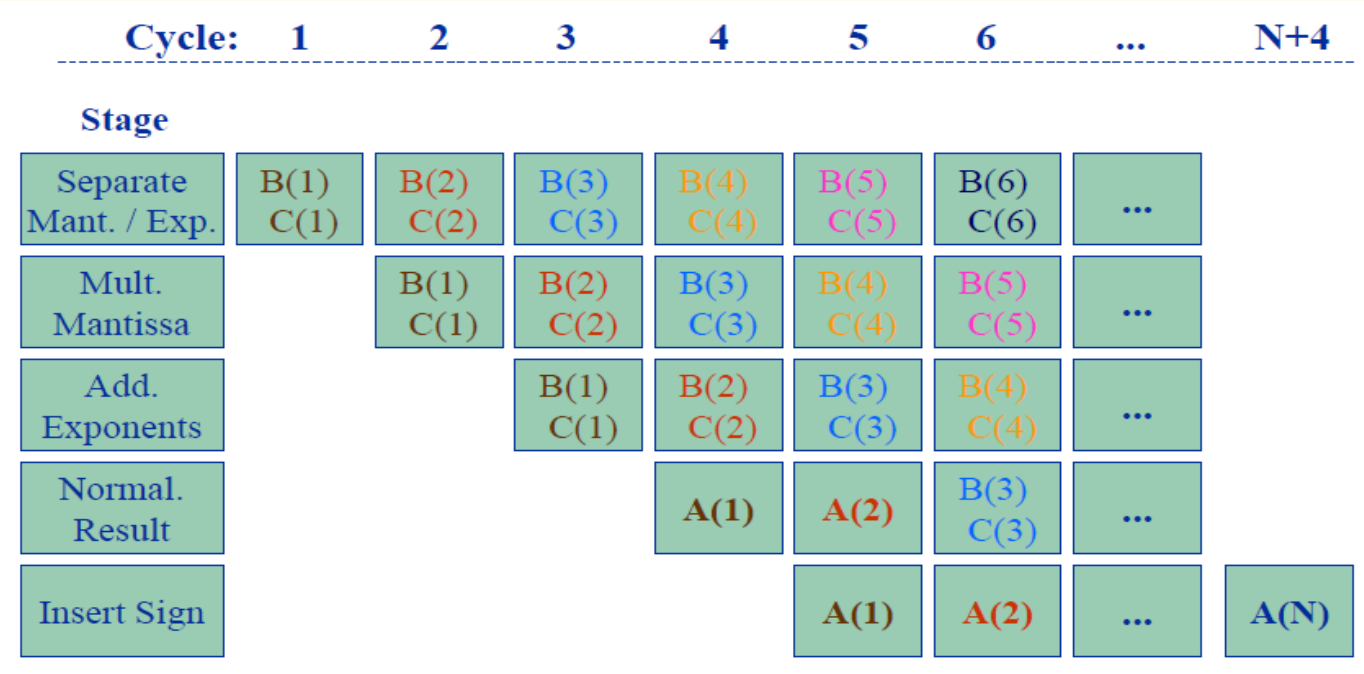
Pipelining em unidades lógica/aritmética

- Ideia:
  - ➔ Quebra instruções complexas em instruções simples / passos rápidos
  - ➔ Toda instrução leva o mesmo tempo (ciclo)
  - ➔ Executa passos diferentes simultaneamente
- Permite ciclos de relógio menores:
  - ➔ FP ops. gastam 5 ciclos, mas o processador pode executar 5 ops. Simultaneamente.
  - ➔ Um resultado por ciclo depois que o pipeline é preenchido.

Pipelining em unidades lógica/aritmética

- Problema:
  - ➔ *Pipeline* deve ser preenchido: tempo de *startup;*
  - ➔ Uso eficiente do pipeline depende de um grande número de instruções independentes;
  - ➔ Requer compilador / hardware sofisticado, capazes de escalonar, agrupar e reordenar instruções.
- Exemplo:
  - ➔ Multiplicação de dois vetores reais
    - `A[i] = B[i]*C[i]`

Pipeline de 5 estágios: A[i] = B[i] * C[i]; i=1, …, N



- Primeiro resultado disponível após 5 ciclos (= **latência** do pipeline).
  - ➔ Depois disso, **uma instrução por ciclo**

Uso eficiente do *pipeline*

- *Pipeline* por software pode ser feito pelo compilador
  - ➔ reordenamento eficiente das instruções requer conhecimento profundo de
    - aplicação (dependências de dados)
    - processador (latências das unidades funcionais);
- Reordenamento de instruções pode ser feito em tempo de execução: *out-of-order execution (OOO)*
- Dependência de dados pode impedir *pipelining* ou reordenamento:

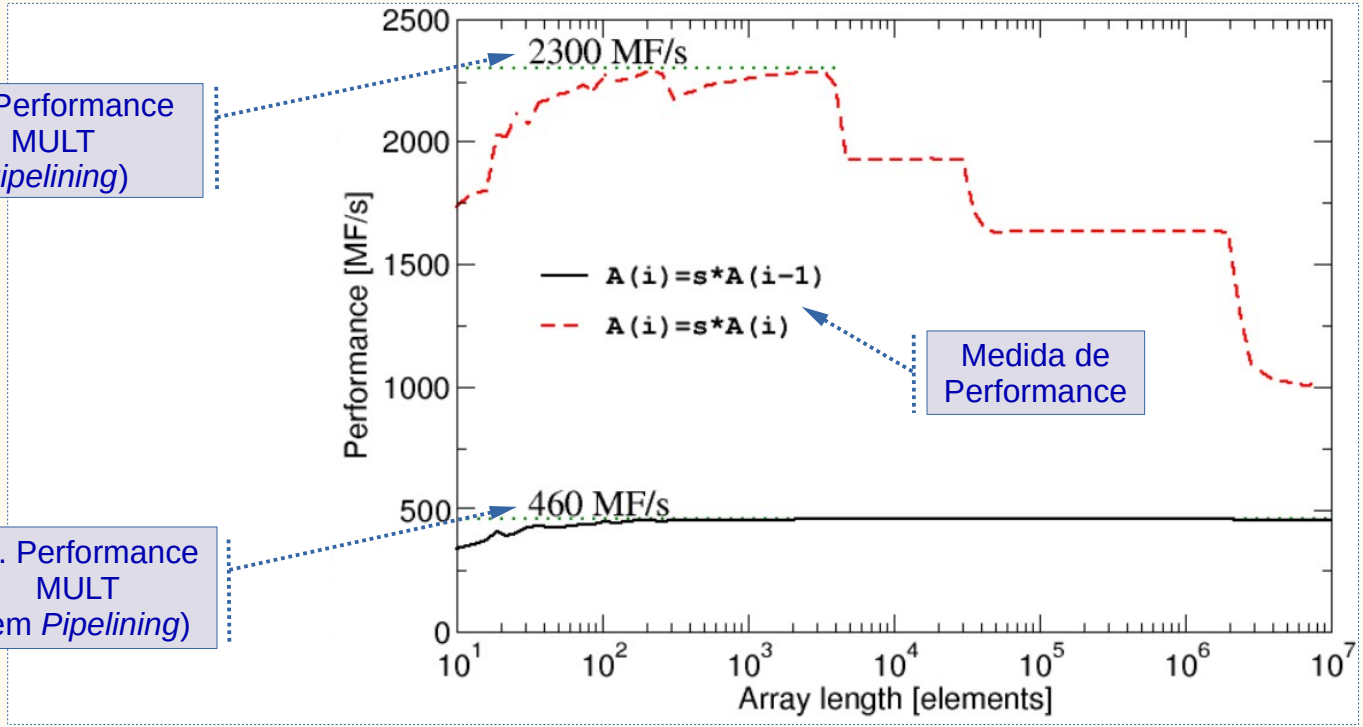
No dependency:

```
do i=1,N
  a(i) = a(i) * c
end do
```

Dependency:

```
do i=2,N
  a(i) = a(i-1) * c
end do
```

Pipelining: dependência de dados



Single core Intel E5-2695v3 (Haswell) - HW limit: **1 MULT/ciclo**

Pipelining

- Valores típicos de estágios de pipeline em CPU's modernas:
  - ➔ 2-5: LoaD; STore; MULT; ADD; FMA
  - ➔ >> 10: DIVide; SquareRooT

- Sem hardware for **pow / exp / sin ...**
  - ➔ **custo alto**

Exemplo: Processador ***Sandy Bridge***

	Instruction	Latency [cy/instruction]	Throughput [instruction/cy]
instructions	ADD DP (SP)	3 (3)	1 (1)
instructions	MULT DP (SP)	5 (5)	1 (1)
instructions	SQRT DP	45	1/44 = 0.023
instructions	SQRT SP	29	1/28 = 0.036
instructions	DIV DP	45	1/44 = 0.023
Library calls (glibc 2.12)	EXP DP	83	1/83 = 0.012
Library calls (glibc 2.12)	SIN DP	79	1/79 = 0.013

Problemas em *pipelining*

- Dependências de dados ocultas:

```
void scale_shift(double *A, double *B, double *C, int n) {

       for(int i=0; i<n; ++i)
          C[i] = A[i] + B[i];
}
```

➔ C/C++ permite "**pointer aliasing**":

- `A → &C[-1]` e `B → &C[-2]→ C[i]=C[i-1]+C[i-2]`
- **Dependência de dados**

➔ Compilador **não consegue** resolver conflitos de *aliasing*

➔ Programador deve informar ao compilador se *aliasing* não é usado:

- Opção **gcc**: `-fstrict-aliasing` (implícito com `-O2` ou `-O3`)
- passar argumentos como `restrict`

Problemas em *pipelining*

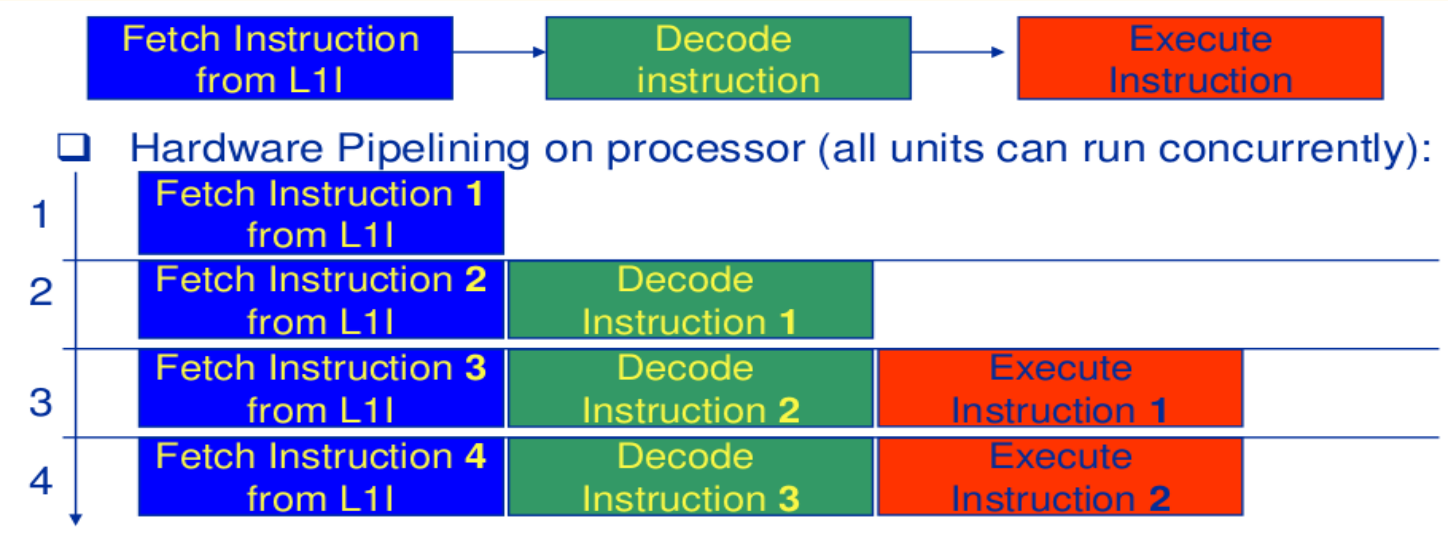
- Chamada de função simples dentro de um laço
  - ➔ Subrotina ***Inline*** (pode ser feita pelo compilador)

- Operação de redução: `sum += A[i]`
  - ➔ Aumenta pipelining por ***loop unrolling***
  - ➔ Compilador pode fazer isto (**`-fassociative-math`**)
  - ➔ Aritmética de FP **não é associativa!!**

$$\Big(\big(\big((a+b)+c\big)+d\big)+e\big)+f\Big) \neq (a+b)+(c+d)+(e+f)$$

Pipeline de Instruções

- Além de *pipeline* aritmético/lógico, existe também um para instruções com pelo menos 3 passos:

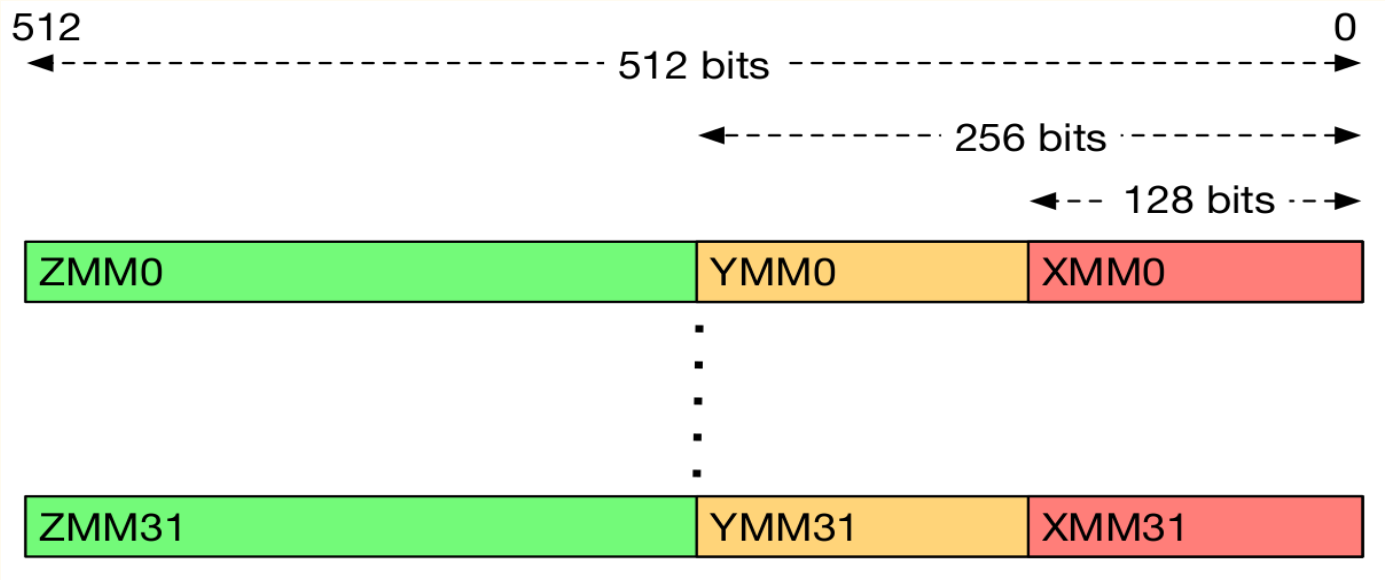


- Desvios podem parar o pipeline
  - ➔ Predição de desvios, execução especulativa

Single Instruction Multiple Data (SIMD)

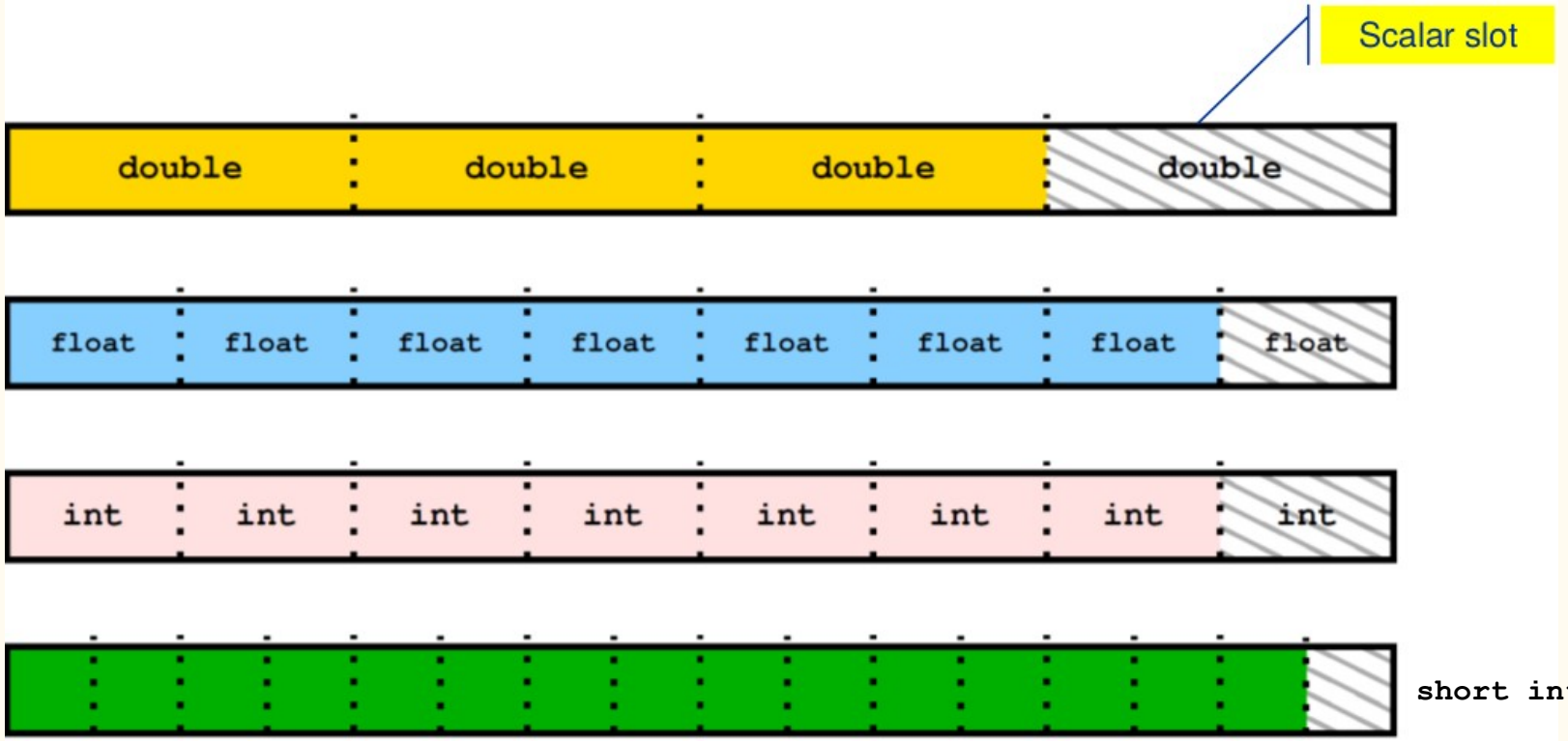
- Ideia: Aplicar a mesma instrução sobre múltiplos operandos em paralelo
  - ➔ SSE: 128 bit (2 double);
  - ➔ AVX/AVX2: 256 bit (4 double)
  - ➔ AVX512: 512 Bit (8 double)
  - ➔ Requer operações em dados independentes (*data-parallel*)
  - ➔ Compilador pode gerar código *vetorizado*.
  - ➔ Pode necessitar de *alinhamento em memória* (16- or 32-byte)

Registradores SIMD



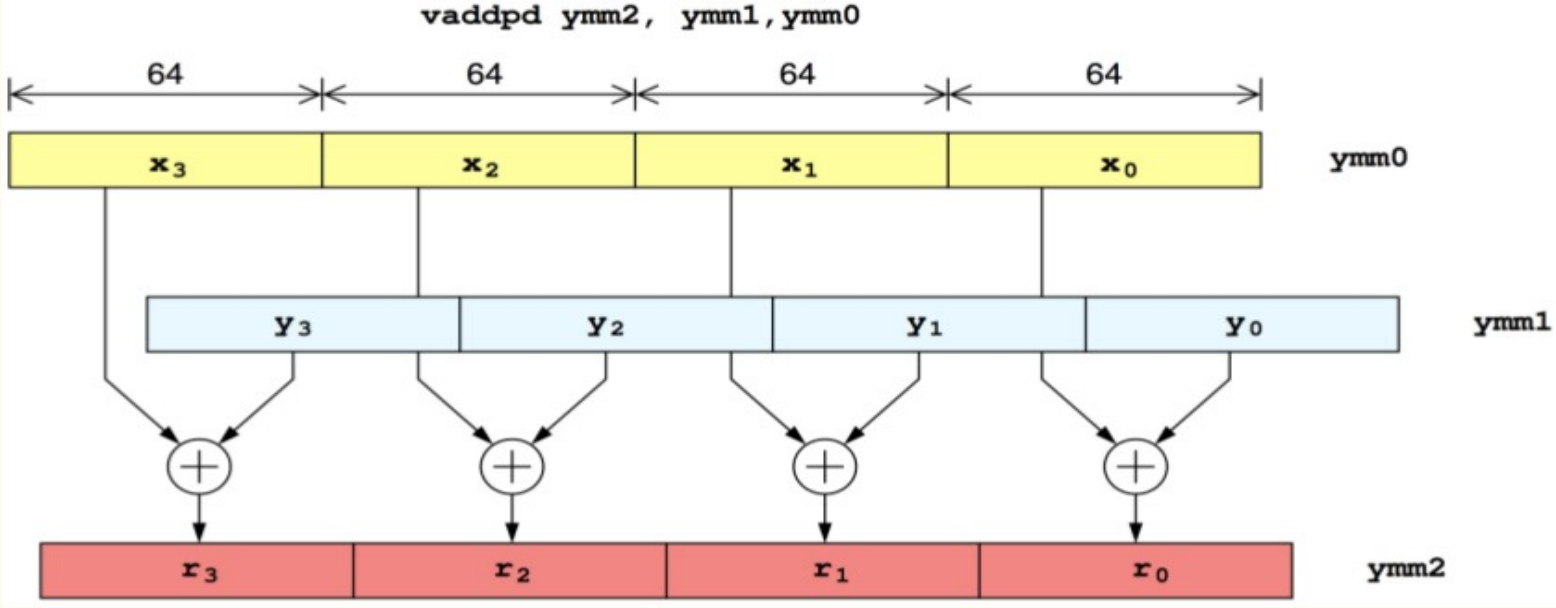
- **SSE** (XMM0-XMM15)
- **AVX** (YMM0-YMM15)
- **AVX-512** (ZMM0-ZMM31)

Tipos de dados SIMD

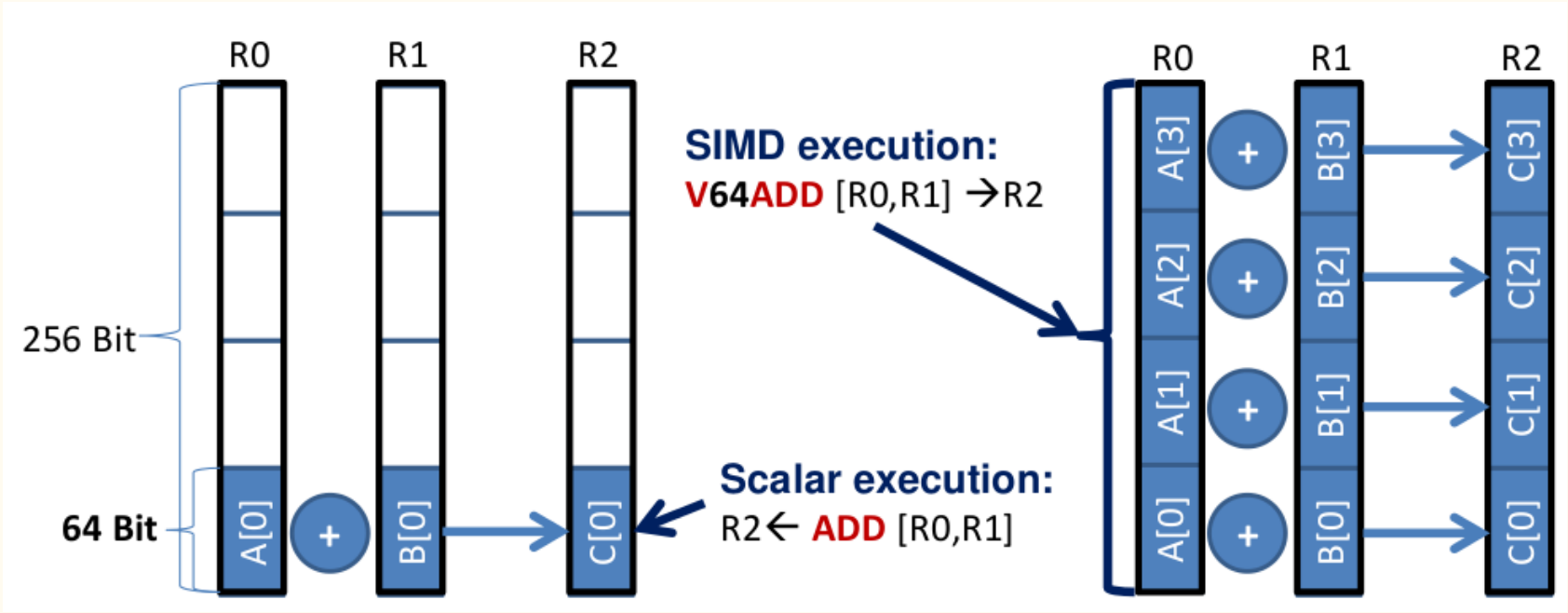


Processamento SIMD

- Exemplo: `vaddpd ymm2, ymm1, ymm0`
  - ➔ “registradores largos” guardam múltiplos operandos (e.g. 4)

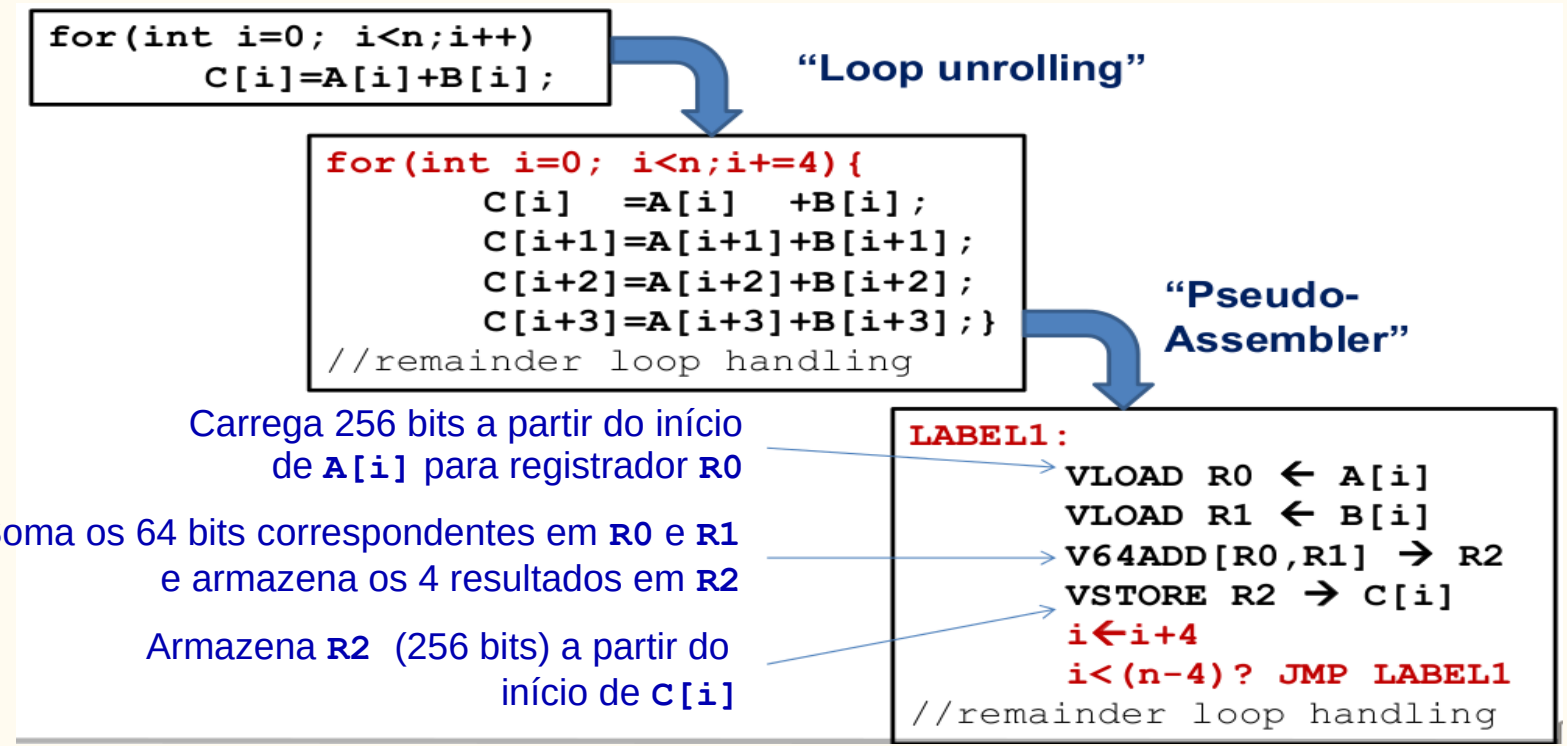


Processamento SIMD



Processamento SIMD

- Passos de processamento SIMD (gerados pelo compilador)



Processamento SIMD

- Não ocorre processamento SIMD em laços com dependência de dados
- "*Pointer aliasing*" pode evitar que compilador gere código SIMD
- Flags de compilação do **GCC** para Autovetorização
  - ➔ **`-march=native -mavx:`** Usa instruções suportadas pela CPU
  - ➔ **`-ftree-vectorize`**: Ativa autovetorização
  - ➔ **`-O3`**: Otimizações incluem autovetorização
  - ➔ **`-fopt-info-vec`**,**`-fopt-info-vec-missed`**:
    - Lista laços vetorizados (ou não) + informação adicional
  - ➔ **`-falign-functions=32`**,**`-falign-loops=32`**
    - Alinha os endereços de funções / laços em múltiplos de 32 bytes

Layout de dados compatíveis com SIMD

- As instruções SIMD **LD** (ST) **carregam** (armazenam) **mútliplos operandos *consecutivos*** a partir de um **endereço base de memória** (de um registrador) **para um registrador** (para um endereço base)

- SIMD: Aplica instrução a vetores (registradores) com operandos independentes

- Estrutura de Arrays (**SoA**) X Array de Estruturas (**AoS**)
  - ➔ Qual abordagem é compatível com SIMD?

SoA x AoS

Structure-of-arrays (SoA)

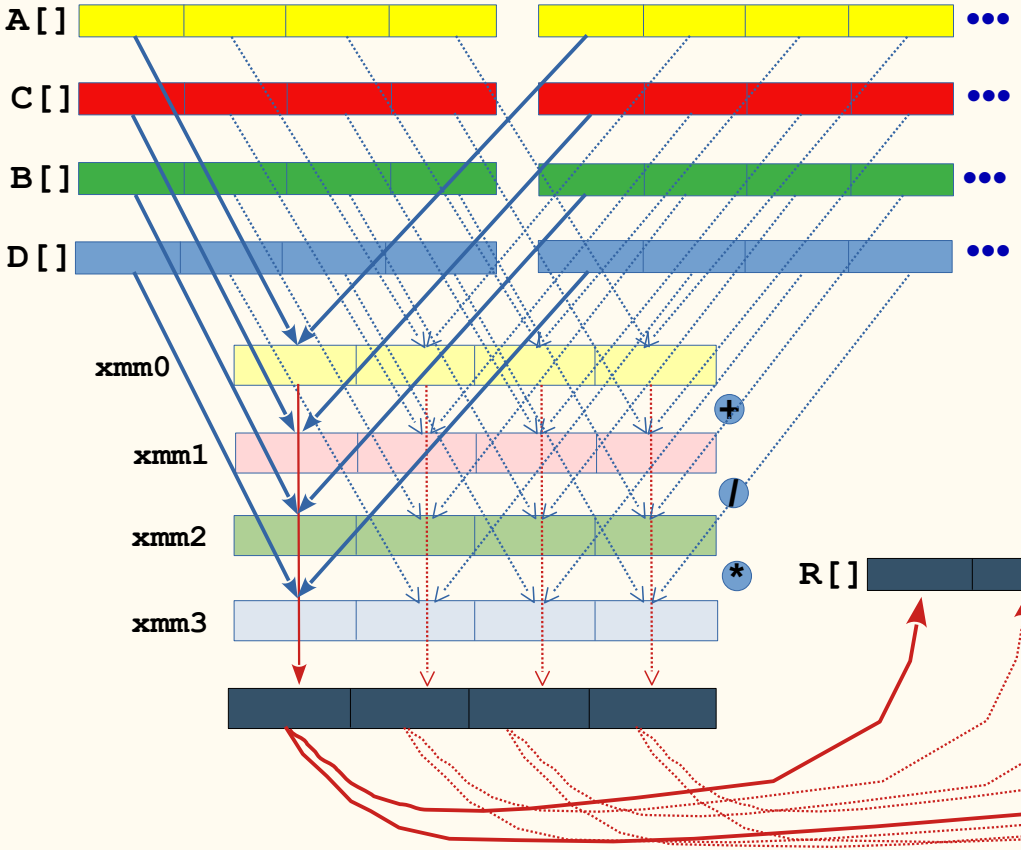
```
…
float *A,*B,*C,*D,*R;
…
for(int i=0; i<n;i++)
 R[i]=((A[i]+C[i])/B(i))*D(i);
```

Array-of-structures (AoS)

```
…
struct stype {
  float  A,B,C,D,R;
};

struct stype *S;
…
for(int i=0; i<n;i++)
       S[i].R = ((S[i].A+S[i].C)/S[i].B)*S[i].D;
```

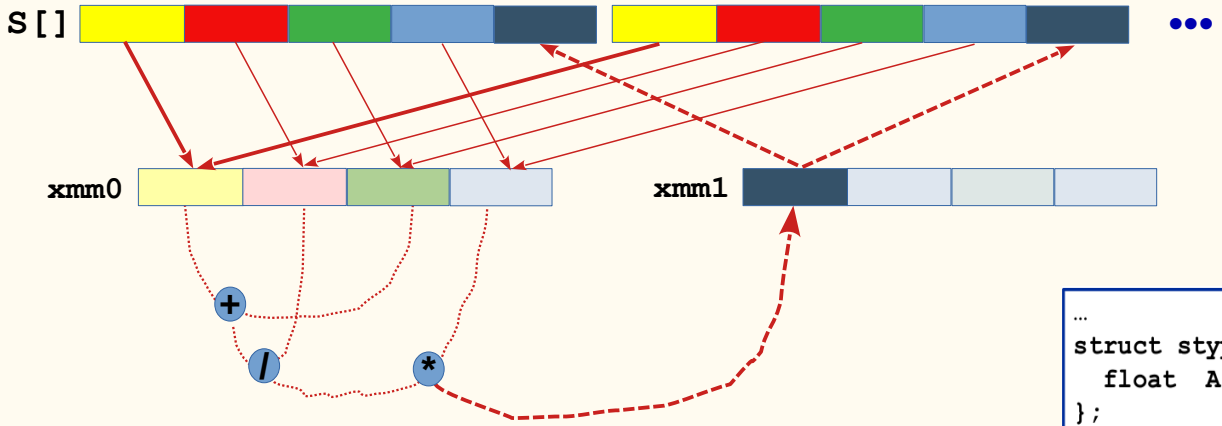
SIMD com layout **SoA**

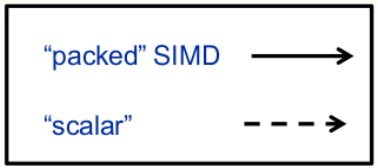


Structure-of-arrays (SoA)

```
…
float *A,*B,*C,*D,*R;
…
for(int i=0; i<n;i++)
 R[i]=((A[i]+C[i])/B(i))*D(i);
```

SIMD layout **AoS**





- Todas as instruções devem ser paralelo-SIMD
- Operações "horizontais" através de um registrador são **lentas**
- **Escolha com cuidado seu layout**

Array-of-structures (AoS)

```
…
struct stype {
  float  A,B,C,D,R;
};

struct stype *S;
…
for(int i=0; i<n;i++)
      S[i].R =((S[i].A+S[i].C)/S[i].B)*S[i].D;
```

Processamento SIMD: Regras básicas

- **Regras para laços vetorizáveis**
  1. Sempre no laço mais interno
  2. Tamanho do laço deve ser determinado na entrada do laço
  3. Entrada e saída únicas
  4. Sem condicionais dentro do laço
  5. Sem dependência de dados
  6. Sem chamada de funções
     ➔ (com exceção de funções matemáticas intrínsecas)

Processamento SIMD: Regras básicas

- **Melhor performance com:**
  1. Laços internos simples com passo unitário (acesso a dados contíguos)
  2. Layout de dados compatível com SIMD
  3. Minimizar endereçamento indireto
  4. Alinhar estruturas de dados ao comprimento SIMD
  5. Em linguagem C, use **restrict** e/ou **const** e/ou opções de compilador que evitem *array/pointer aliasing*

Referências

- Daniel Weingaertner; notas de aula da disciplina **Introdução à Computação Científica** (UFPR/DINF)

- G. Hager, G. Wellein; **Introduction to High Performance Computing for Scientists and Engineers**. CRC Press, 2011.

Créditos

Este documento foi desenvolvido pelo Prof. Armando Luiz N. Delgado (UFPR/DINF), para uso na disciplina Introdução à Computação Científica (CI1164), a partir de conteúdo de autoria do Prof. Daniel Weingaertner (UFPR/DINF).

Compartilhe este documento de acordo com a licença abaixo