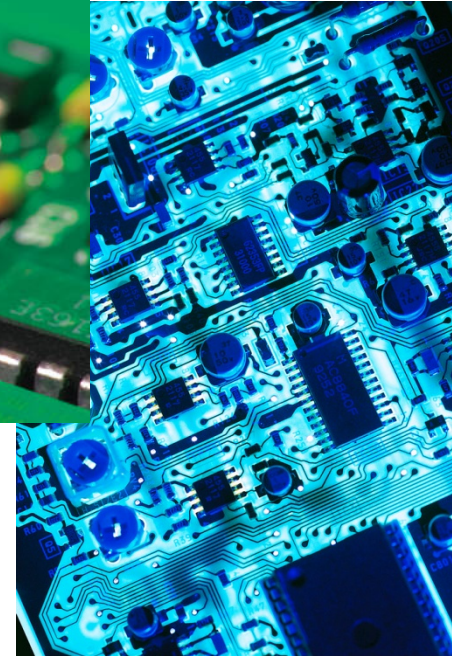
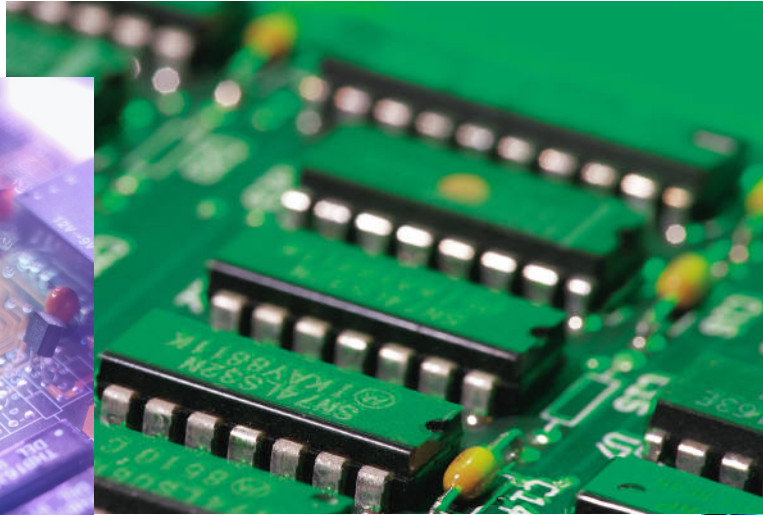
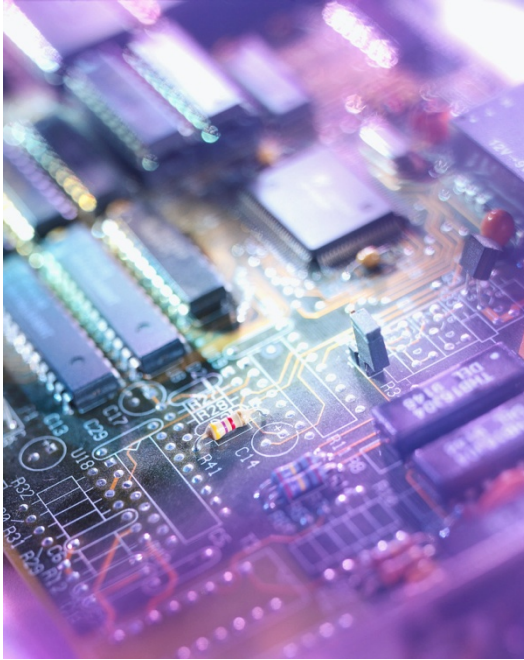


# IPPD

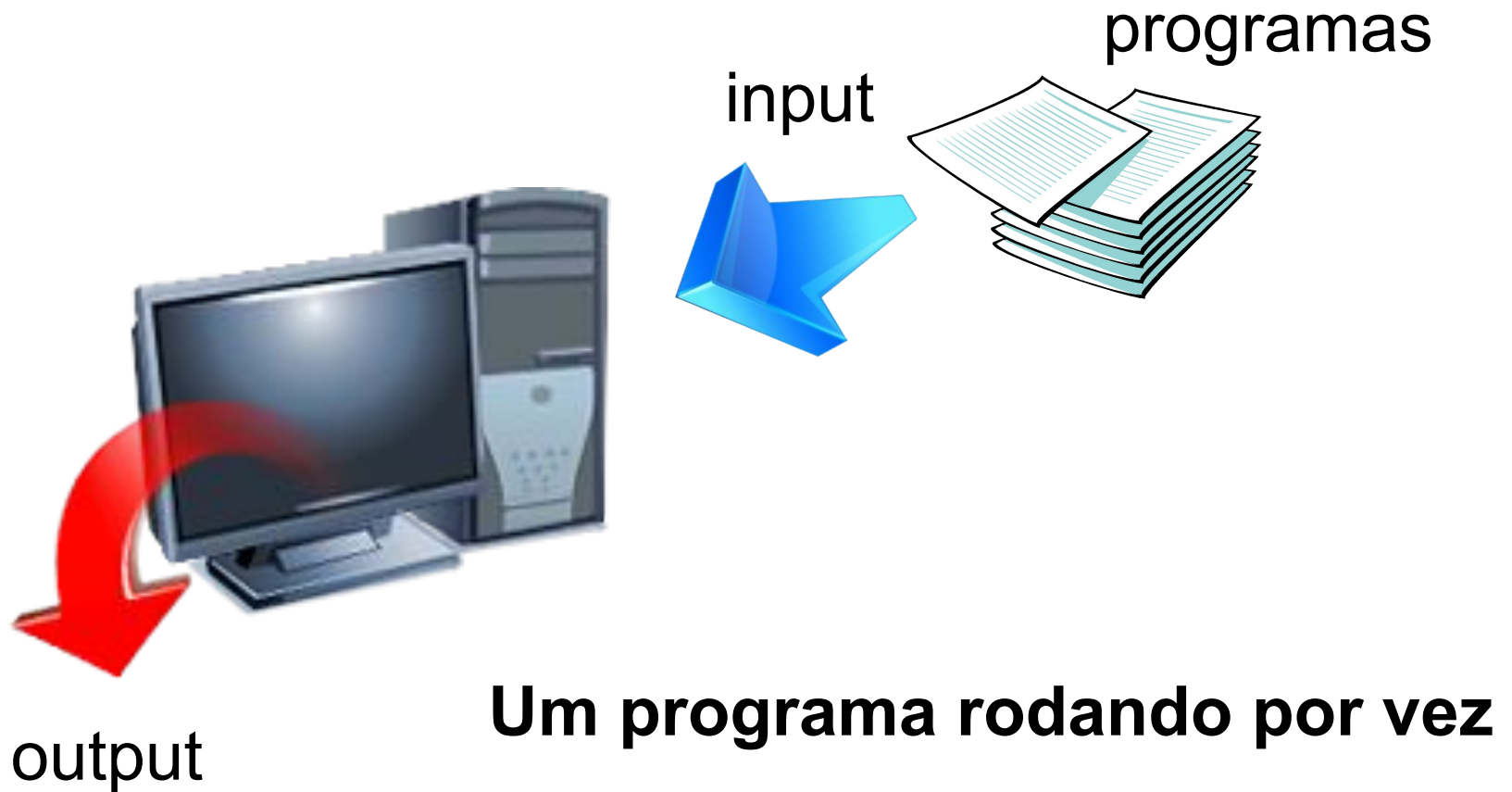
## Hoje: Hardware e software

**Prof. Dr. Rafael P. Torchelsen**  
**[rafael.torchelsen@inf.ufpel.edu.br](mailto:rafael.torchelsen@inf.ufpel.edu.br)**

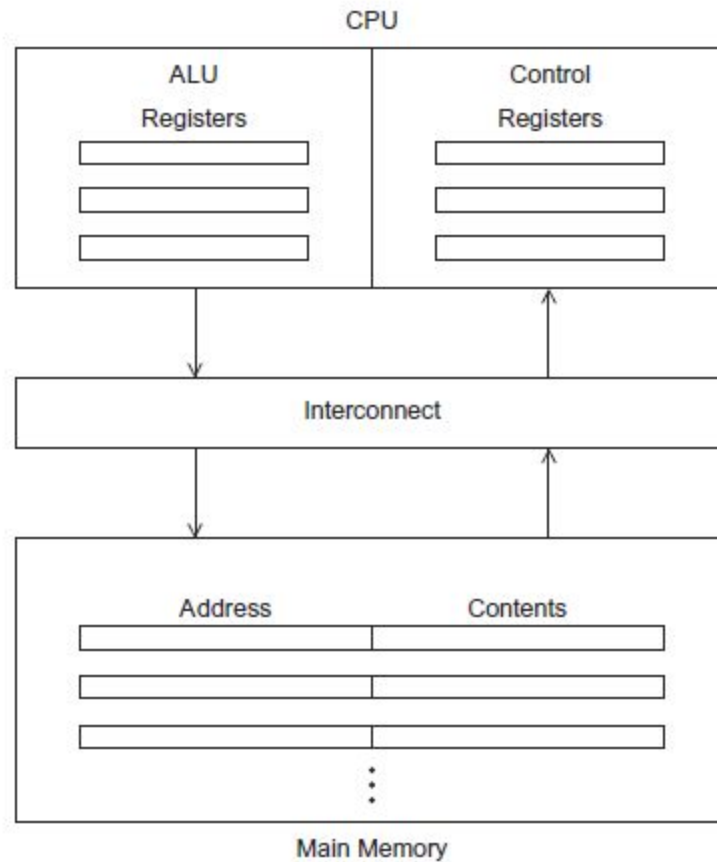
# Revisão SO



# Hardware e software serial

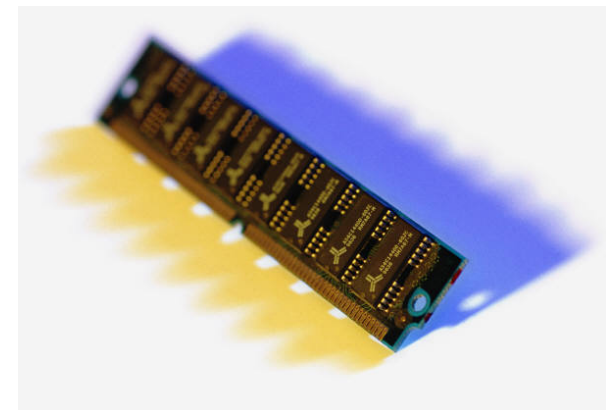


# Arquitetura de von Neumann



# Memória principal

- Coleção de posições onde cada uma é capaz de armazenar instruções e dados.
- Cada localização consiste de um endereço que é usado para acessar uma localização e o conteúdo dessa localização



# Central processing unit (CPU)

- Dividida em duas partes:
  - Unidade de controle: Responsável por decidir qual instrução em um programa deve ser executado
  - Unidade lógica e aritmética (ALU): Responsável por executar a instrução



# Termos chave

- **Registrador:** armazenamento de alto desempenho e faz parte da CPU
- **Contador:** Armazena o endereço da próxima instrução a ser executada.
- **Barramento:** conexão no hardware que transporta informação entre a CPU e a memória



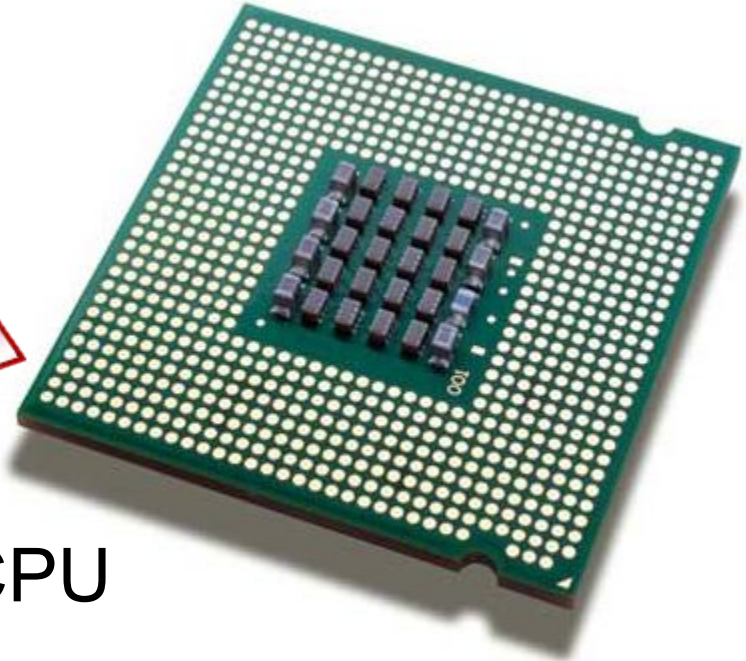
memória



**busca/leitura**



CPU





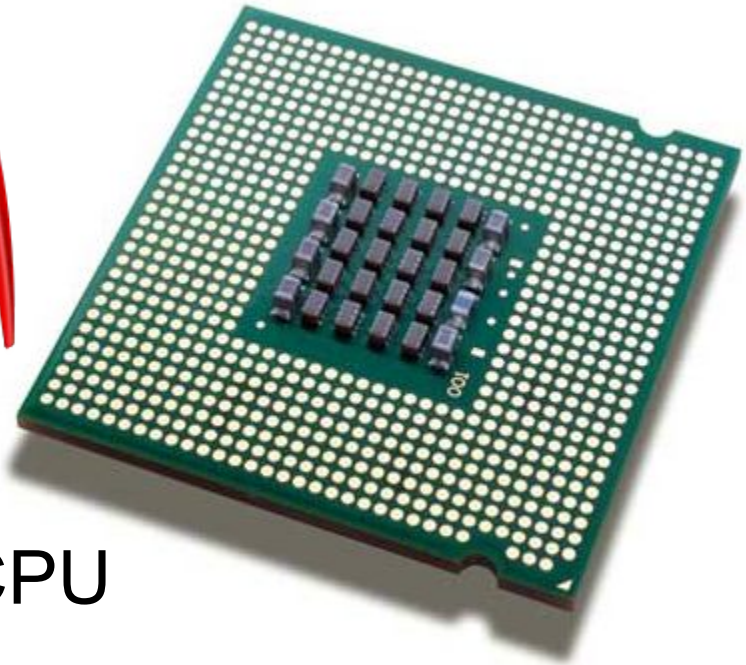
memória



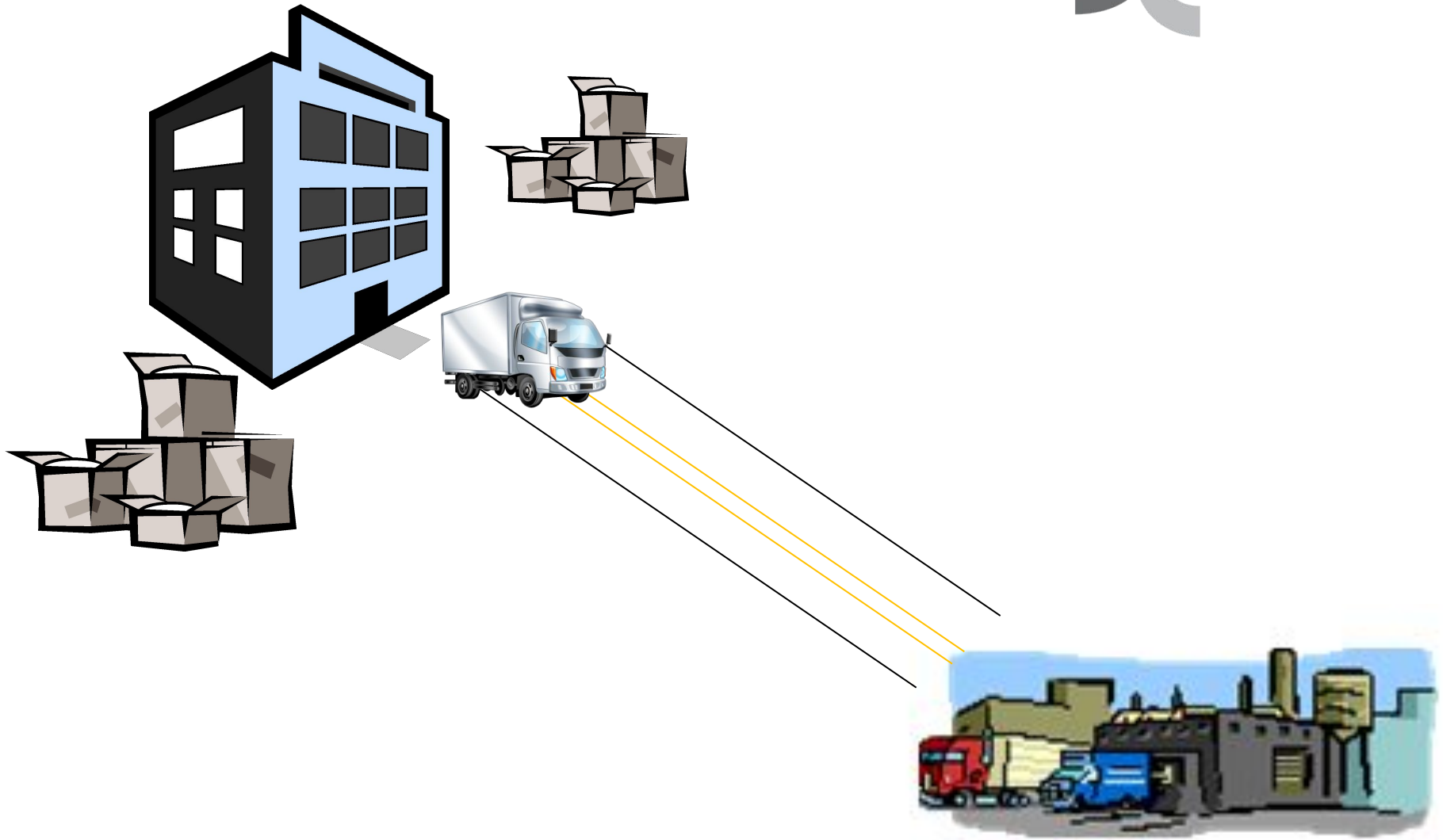
**Escrita/armazenamento**



CPU



# von Neumann bottleneck



# Um processo do SO

- Uma instância de um programa em execução
- Componentes de um processo:
  - Executável em linguagem de máquina
  - Bloco de memória
  - Descritores dos recursos que o SO alocou para o processo
  - Informações de segurança
  - Informações sobre o estado do processo

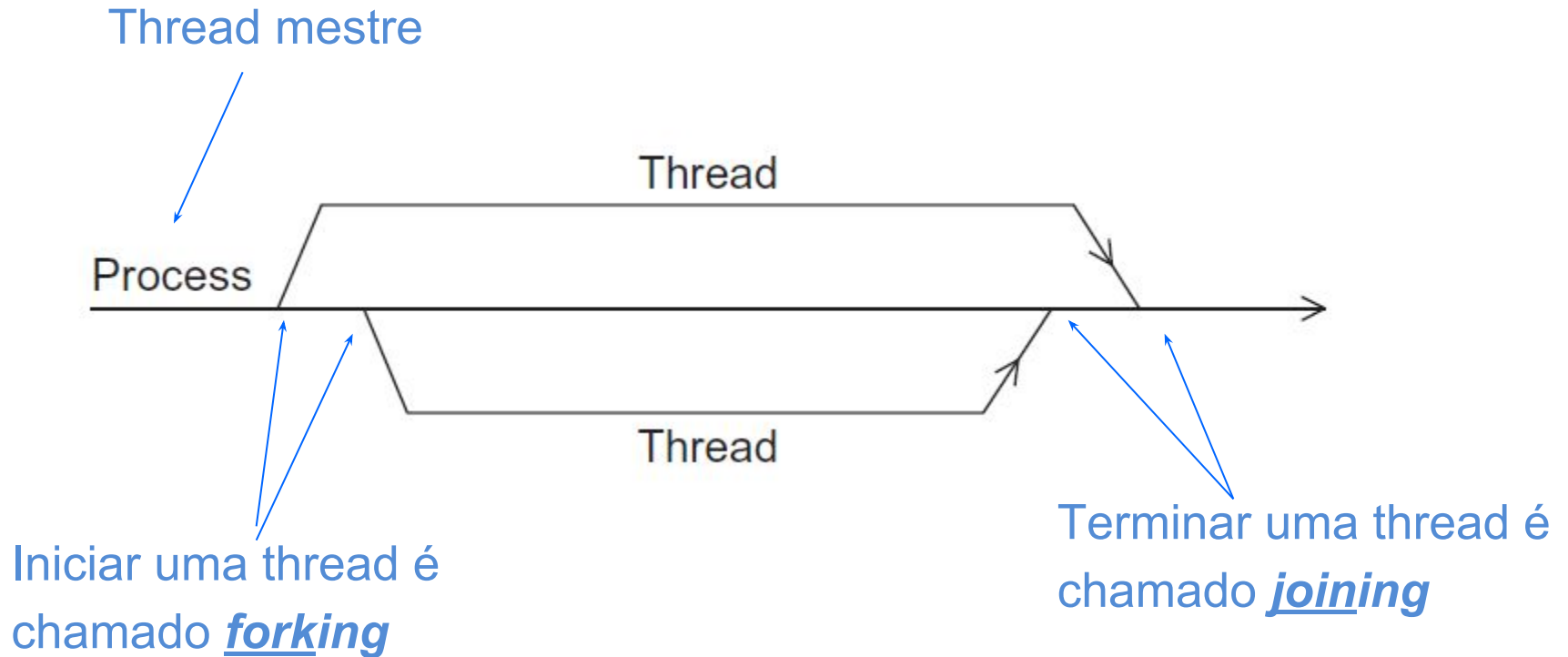
# Multi-Tarefa

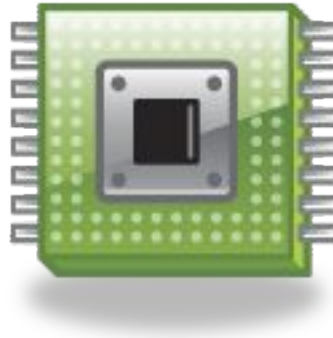
- Ilusão que um processador está executando múltiplos programas ao mesmo tempo.
- Cada processo recebe um tempo para executar
- Após o fim do seu tempo ele aguarda novamente a sua vez

# Threading

- Threads estão contidas nos processos
- Elas permitem que os programadores dividam o programa em partes (mais ou menos) independentes
- A ideia é que se uma thread tem que parar é por estar aguardando um recurso e outra thread pode executar evitando ociosidade do processador

# Um processo e duas threads





# Modificações do modelo de von neumann



# Básico sobre caching

- É uma coleção de localizações de memória que podem ser acessadas mais rapidamente que em outras localizações
- O cache de uma CPU geralmente está localizado no mesmo chip da CPU ou em uma memória que possa ser acessada muito mais rapidamente que em outros tipos de memória



# Princípios de localidade

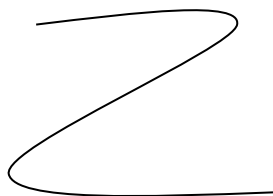
- Acessar uma localização é seguido por um acesso próximo
- **Localidade espacial:** acessar uma localização próxima
- **Localidade temporal:** acesso num futuro próximo

# Princípios de localidade

```
float z[1000];  
  
...  
  
sum = 0.0;  
for (i = 0; i < 1000; i++)  
    sum += z[i];
```

# Níveis de cache

pequena & muito veloz



L1



Por que elas existem? Por que todas não são L1?

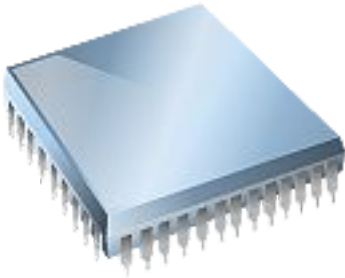
L2

L3



grande & lenta

# Cache hit



fetch x

L1

x sum

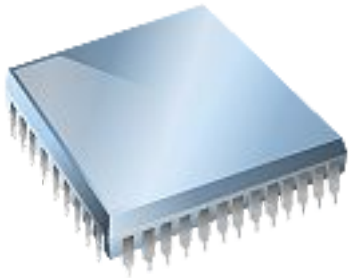
L2

y z total

L3

A[ ] radius r1 center

# Cache miss



fetch x

L1

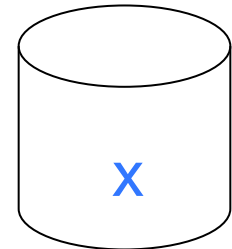
y sum

L2

r1 z total

L3

A[ ] radius center



Memória  
principal

# Problemas com cache

- Quando a CPU escreve no cache o valor pode ficar inconsistente com o valor na memória principal
- **Write-through:** o problema é resolvido com uma atualização da memória principal sempre que se escreve no cache
- **Write-back:** a cache marca um dado como lixo e a localização é liberada para uso, mas antes o dado é transferido para a memória principal

Não são os únicos algoritmos  
para resolver o problema



# Mapeamento de cache

- **Mapeamento direto:** um novo dado pode ser colocado em qualquer localização da cache
- **Mapeamento associativo:** cada dado tem uma localização única na cache onde pode ser colocado
- **Mapeamento associativo por conjunto (N-way):** meio termo direto e o associativo

# Mapeamento associativo por conjunto (N-way)

- Quando mais de uma localização na memória pode receber o mesmo dados temos que ser capazes de decidir em qual localização colocar ou evitar

# Exemplo

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

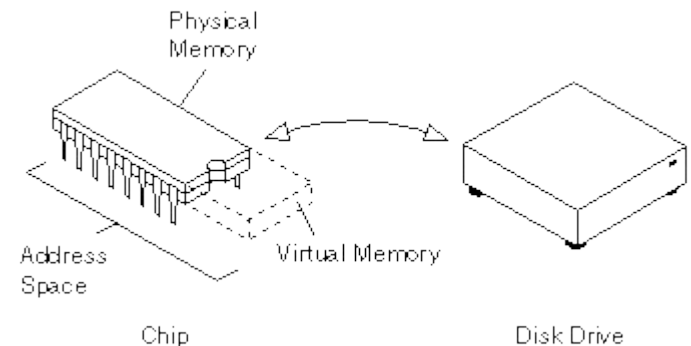
# Cache e programas

```
double A[MAX][MAX], x[MAX], y[MAX];  
.  
.  
.  
/* Initialize A and x, assign y = 0 */  
.  
.  
.  
/* First pair of loops */  
for (i = 0; i < MAX; i++)  
    for (j = 0; j < MAX; j++)  
        y[i] += A[i][j]*x[j];  
.  
.  
.  
/* Assign y = 0 */  
.  
.  
.  
/* Second pair of loops */  
for (j = 0; j < MAX; j++)  
    for (i = 0; i < MAX; i++)  
        y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

# Memória virtual

- Quando os programas consomem toda a memória novas informações e instruções podem não ter onde serem escritas
- Memória virtual funciona como um cache para memória secundarias

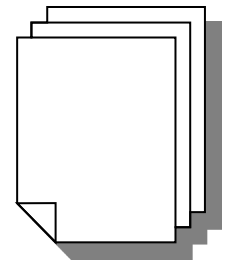


# Memória virtual

- Explora o princípio de localidade espacial e temporal
- Mantém somente as partes em execução de um programa em memória

# Memória virtual

- **Swap:** são partes que não estão em uso são guardadas na memória secundária
- **Páginas:** blocos de dados e instruções
  - Geralmente elas são relativamente grandes
  - A maioria dos sistemas tem um tamanho fixo de página, exemplo: 4 ou 16 kilobytes





# Memória virtual

- Quando um programa é compilado suas páginas recebem números de páginas virtuais
- Quando o programa é executado uma tabela é criada que faz o mapeamento entre as páginas virtuais e endereços físicos
- Quando uma página é buscada e ela não se encontra na memória principal temos um page fault

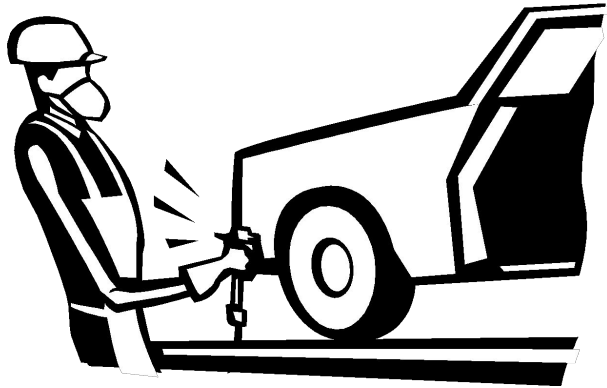
# Paralelismo em nível de instrução

- Tentativa de aumentar o desempenho em nível de processador
- Executar mais de uma instrução ao mesmo tempo
- Isso é implementado em nível de hardware ao se ter múltiplas unidades funcionais

# Paralelismo em nível de instrução

- Pipeline: as unidades são organizadas em estágios
- Múltiplas execuções: múltiplas instruções podem ser executadas ao mesmo tempo

# Pipeline



# Pipeline: Exemplo

- Soma de dois valores em ponto flutuante  
 $9.87 \times 10^4$  e  $6.54 \times 10^3$

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	$9.87 \times 10^4$	$6.54 \times 10^3$	
2	Compare exponents	$9.87 \times 10^4$	$6.54 \times 10^3$	
3	Shift one operand	$9.87 \times 10^4$	$0.654 \times 10^4$	
4	Add	$9.87 \times 10^4$	$0.654 \times 10^4$	$10.524 \times 10^4$
5	Normalize result	$9.87 \times 10^4$	$0.654 \times 10^4$	$1.0524 \times 10^5$
6	Round result	$9.87 \times 10^4$	$0.654 \times 10^4$	$1.05 \times 10^5$
7	Store result	$9.87 \times 10^4$	$0.654 \times 10^4$	$1.05 \times 10^5$

# Pipeline: Exemplo

- Assumindo que cada operação leva 1 nanosegundo ( $10^{-9}$  segundos)
- Esse loop leva 7000 nanosegundos

```
float x[1000], y[1000], z[1000];  
.  
.  
.  
for (i = 0; i < 1000; i++)  
    z[i] = x[i] + y[i];
```

# Pipeline: Exemplo

- Dividir a soma de ponto flutuante em 7 unidades funcionais em hardware
- Primeira unidade busca dois operandos, segunda compara expoentes, etc.
- Saída de uma unidade é entrada da seguinte

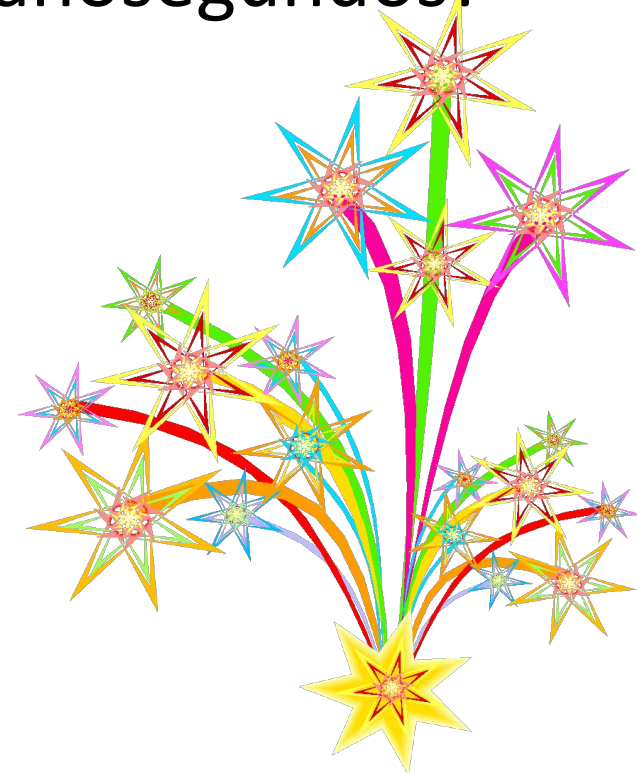


# Pipeline: Exemplo

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

# Pipeline: Exemplo

- Uma soma ainda leva 7 nanosegundos
- Porém 1000 somas leva 1006 nanosegundos!

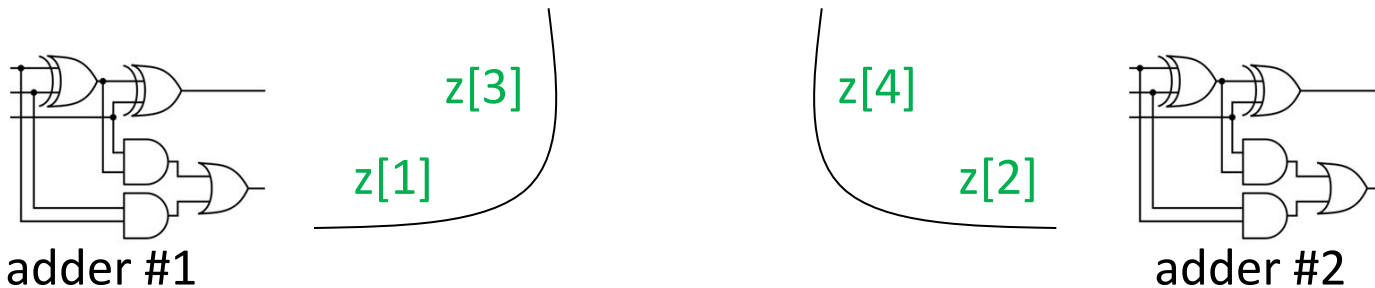


# Múltiplas execuções

- Ter vários pipelines de uma mesma operação e tentar executá-las simultaneamente

for ( $i = 0; i < 1000; i++$ )

$z[i] = x[i] + y[i];$

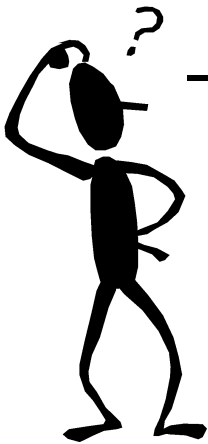


# Múltiplas execuções

- Estática: identificação das operações que podem ser executadas em paralelo em tempo de compilação
- Dinâmica: identificação em tempo de execução

# Especulação

- Para isso funcionar o sistema precisa identificar instruções que podem executar simultaneamente



- Especulação: o compilador ou processador precisa “chutar” quais instruções podem ser feitas em paralelo

# Especulação

```
z = x + y ;
```

```
if ( z > 0)
```

```
    w = x ;
```

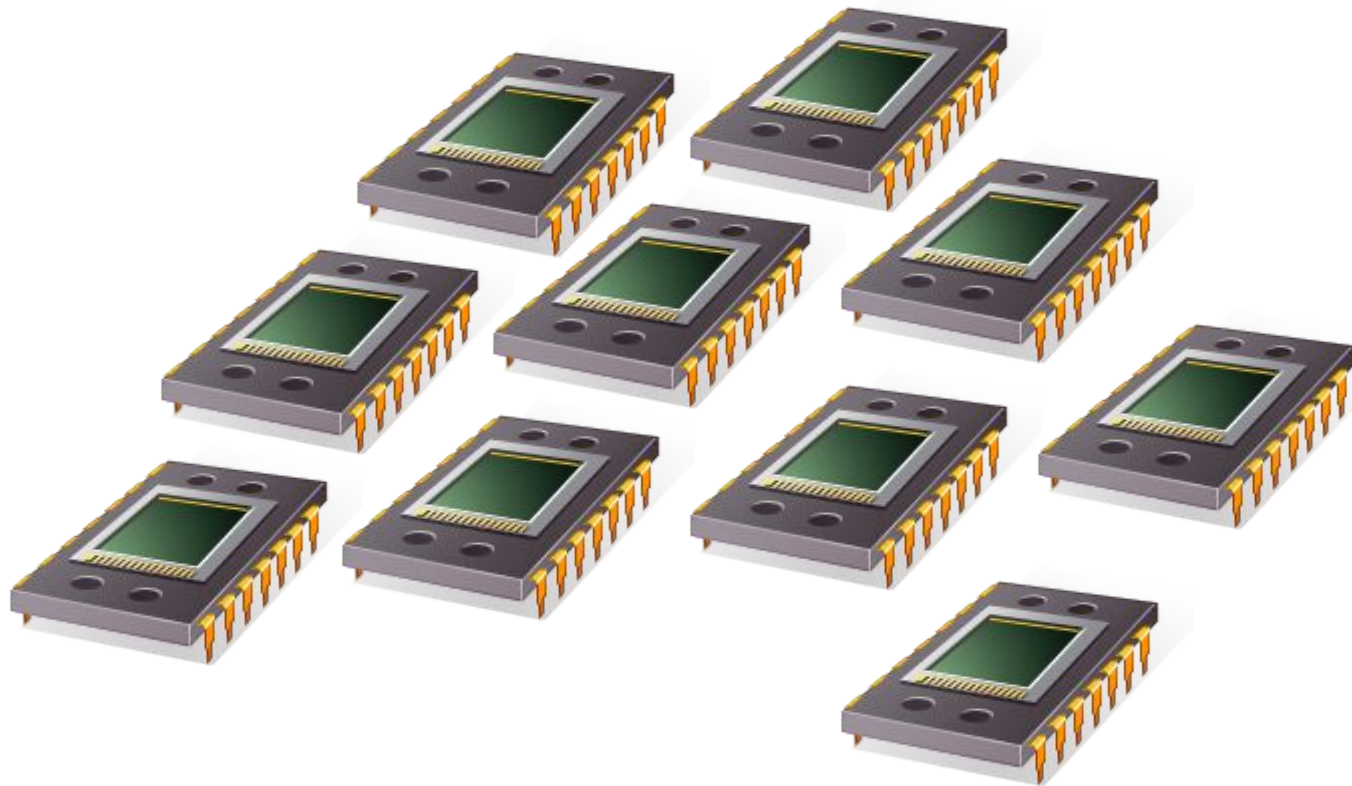
```
else
```

```
    w = y ;
```



Se errar é precisar  
refazer e calcular  $w = y$

# Hardware paralelo



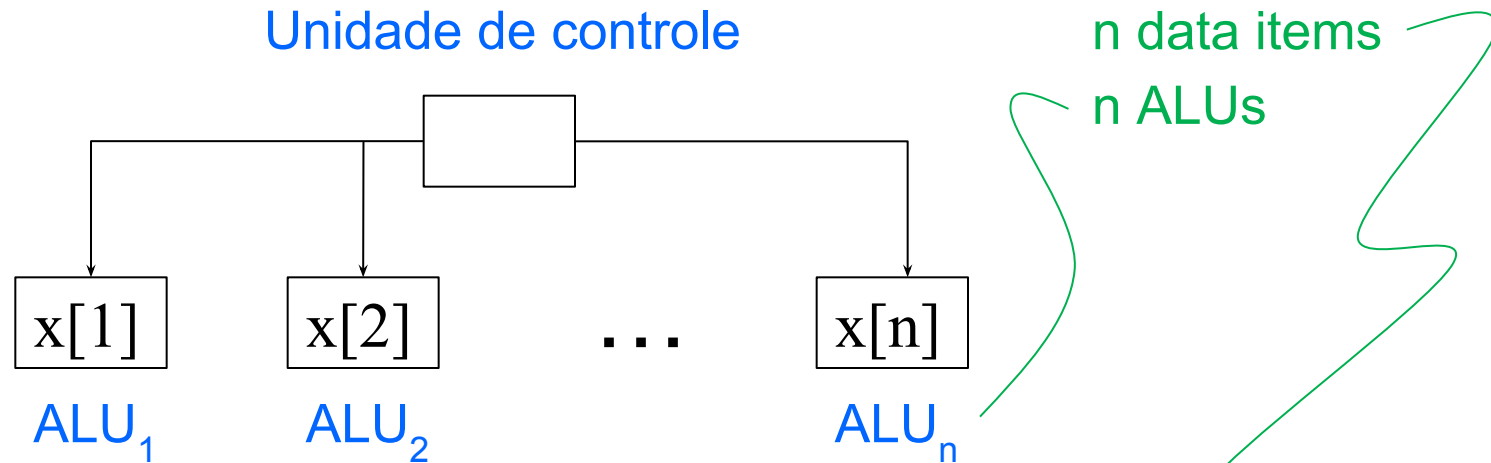
# Taxinomia de Flynn

<p><b>SISD</b></p> <p>Single instruction stream Single data stream</p>	<p><b>(SIMD)</b></p> <p>Single instruction stream Multiple data stream</p>
<p><b>MISD</b></p> <p>Multiple instruction stream Single data stream</p>	<p><b>(MIMD)</b></p> <p>Multiple instruction stream Multiple data stream</p>



- Paralelismo feito por divisão dos dados entre os processadores
- Executa a mesma instrução em múltiplos itens do mesmo conjunto de dados
- Chamado de paralelismo de dados

# Exemplo



```
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

# Exemplo

- E se não tivermos quantidade igual de ALUs quanto de dados?
- Dividimos o trabalho e executamos de forma iterativa
- Exemplo:  $m = 4$  ALUs e  $n = 15$  dados

Round3	ALU <sub>1</sub>	ALU <sub>2</sub>	ALU <sub>3</sub>	ALU <sub>4</sub>
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

# SIMD: Problemas

- Todas as ALUs precisam executar a mesma instrução ou ficam paradas
- Eficiente para problemas paralelos com grandes quantidades de dados

# GPUs

- Computação Gráfica
- Mineração de moedas (Bitcoin)
- IA



OpenCL



Existem  
linguagens  
específicas  
para CG

- Suporta múltiplas instruções em múltiplas instruções
- Geralmente é uma coleção de processadores completamente independentes com seus próprios conjuntos de unidades de controle e ALUs



Parallel software

# SPMD – single program multiple data

- Um programa SPMD consiste em um executável que se comporta como se existissem diversos programas. Isso é feito com o uso de condições e indireções

```
if (I'm thread process i)
    do this;
else
    do that;
```





# Escrevendo programas paralelos

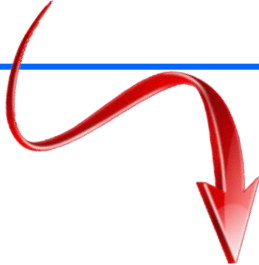
1. Divida o trabalho entre os processos/threads
  1. De forma que cada processo/thread receba mais ou menos a mesma quantidade de trabalhos...
  2. ...e a comunicação seja mínima
2. Considere a sincronização entres os processos/threads
3. Considere a comunicação entre os processos/threads

# Memória Compartilhada

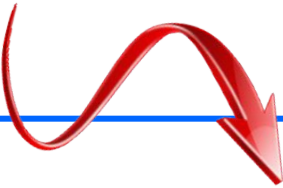
- Threads dinâmicas
  - Thread mestre espera por trabalho, cria novas threads e quando elas terminam a mestre as termina
  - Uso eficiente dos recursos, mas a criação de threads e terminação consome tempo
- Threads estáticas
  - Número fixo de threads para criar e alocar trabalho que não terminam até que o trabalho esteja completo
  - Desempenho melhor, mas tem potencial de desperdiçar recursos do sistema

# Não determinístico

```
...  
printf ( "Thread %d > my_val = %d\n" ,  
        my_rank , my_x ) ;  
...
```



Thread 1 > my\_val = 19  
Thread 0 > my\_val = 7



Thread 0 > my\_val = 7  
Thread 1 > my\_val = 19

# Não determinístico

```
my_val = Compute_val ( my_rank ) ;  
x += my_val ;
```

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19

# Não determinístico

- Condições de corrida
- Sessões críticas
- Exclusão mútua (mutex, lock)

```
my_val = Compute_val ( my_rank ) ;  
Lock(&add_my_val_lock ) ;  
x += my_val ;  
Unlock(&add_my_val_lock ) ;
```

# busy-waiting

```
my_val = Compute_val ( my_rank ) ;  
i f ( my_rank == 1 )  
    w h i l e ( ! ok_for_1 ) ; /* Busy-wait loop */  
x += my_val ; /* Critical section */  
i f ( my_rank == 0 )  
    ok_for_1 = true ; /* Let thread 1 update x */
```

# Mensagem

```
char message [ 1 0 0 ] ;
```

```
...
```

```
my_rank = Get_rank ( ) ;
```

```
i f ( my_rank == 1 ) {
```

```
    sprintf ( message , "Greetings from process 1" ) ;
```

```
    Send ( message , MSG_CHAR , 100 , 0 ) ;
```

```
} e l s e i f ( my_rank == 0 ) {
```

```
    Receive ( message , MSG_CHAR , 100 , 1 ) ;
```

```
    printf ( "Process 0 > Received: %s\n" , message ) ;
```

```
}
```

# Divisão de espaço de trabalho

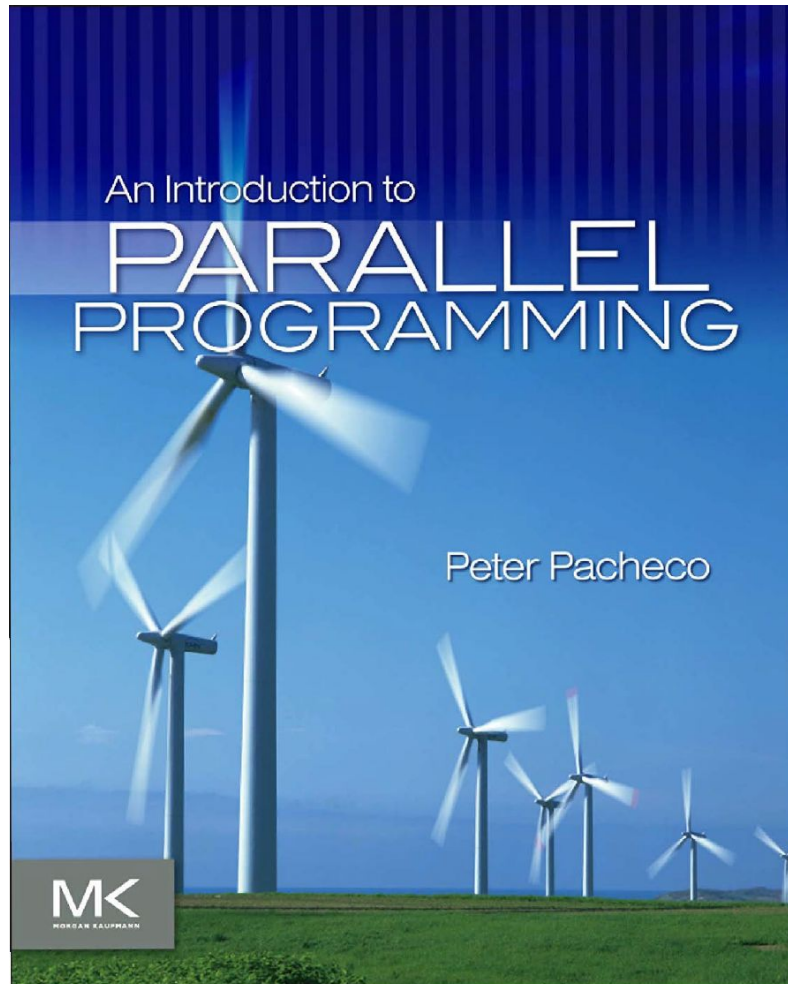
```
shared i n t n = . . . ;
shared double x [ n ] , y [ n ] ;
private i n t i , my_first_element , my_last_element ;
my_first_element = . . . ;
my_last_element = . . . ;
/* Initialize x and y */
. . .
f o r ( i = my_first_element ; i <= my_last_element ; i++)
    x [ i ] += y [ i ] ;
```



# Debug

- Qual a dificuldade de depurar esses códigos?

# Leitura



Ler capítulo 2