

CE4302 – Arquitectura de Computadores II

Introducción al paralelismo a nivel de datos

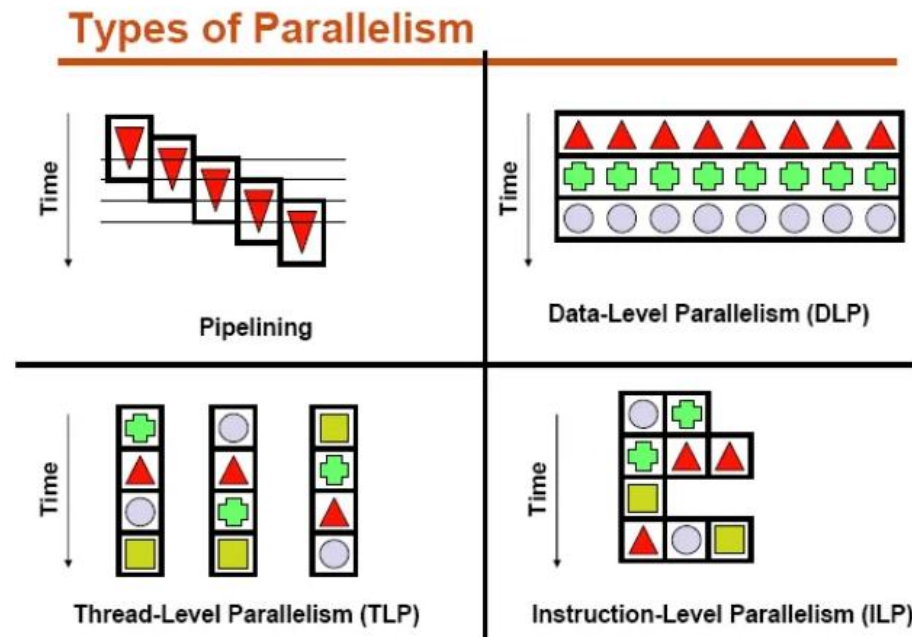
PROFESOR: ING. LUIS BARBOZA ARTAVIA

Agenda

- Introducción.
- SISD - MIMD.
- SIMD.

Introducción

¿Qué es Paralelismo a Nivel de Datos?

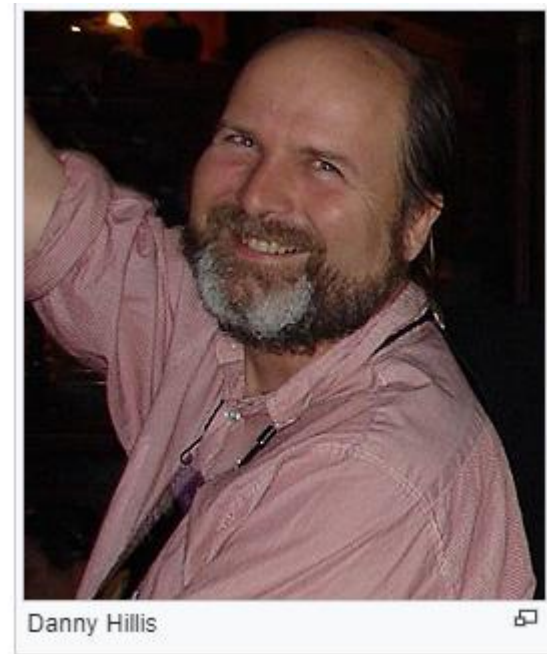


<https://medium.com/@geminae.stellae/introduction-to-parallel-computing-with-opencl-2ee91c30b8b6>

Introducción

We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data rather than from multiple threads of control.

W. Daniel Hillis and Guy L. Steele,
“Data parallel algorithms,” *Commun. ACM* (1986)

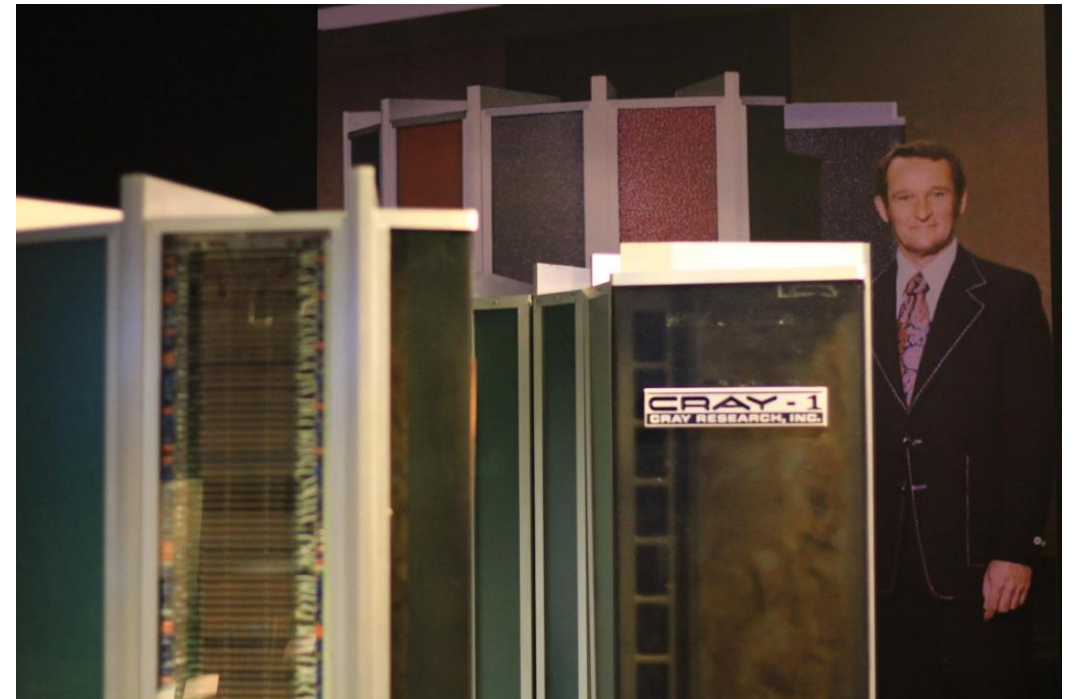


Introducción

If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?

Seymour Cray, Father of the Supercomputer

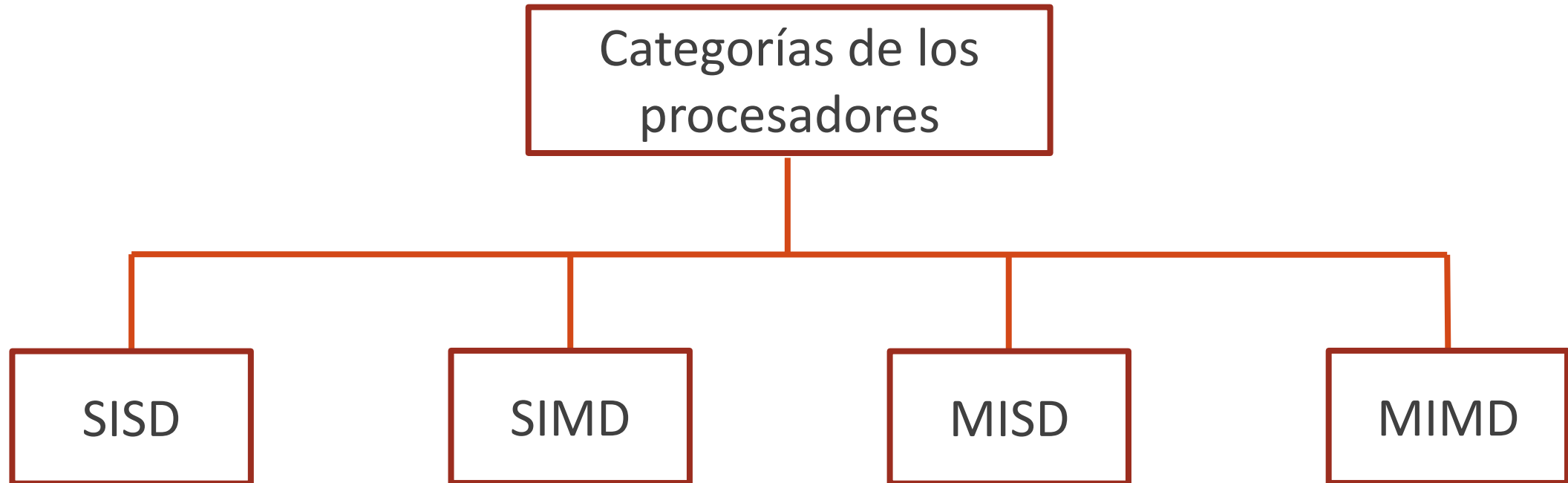
*(arguing for two powerful vector processors
versus many simple processors)*



Taxonomía de procesadores

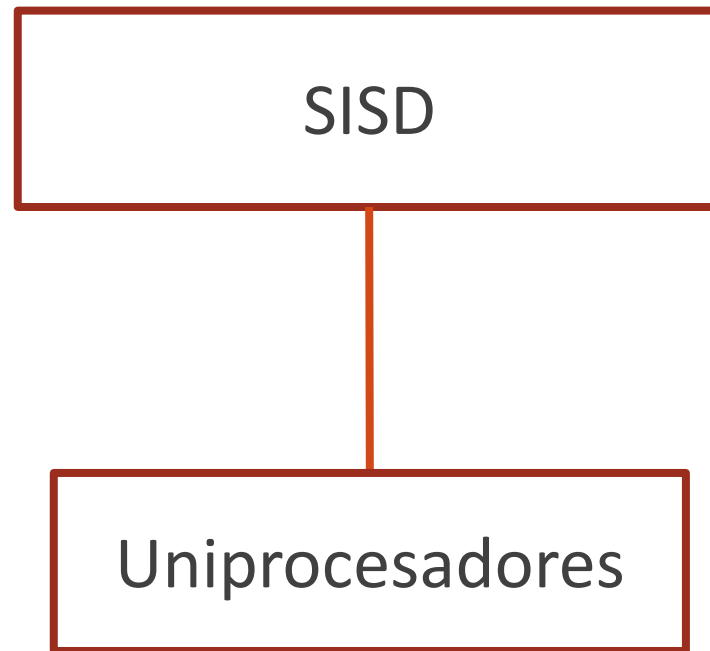
- Clasificación según los CPU interactúan entre sí.
- La selección la podemos hacer según la necesidad de nuestra aplicación.
- Flynn fue uno de los primeros que introdujo una taxonomía.
- Sigue siendo una de la más populares para categorizar sistemas con capacidades de paralelismo.
- Toma en cuenta la cantidad de información procesada y el conjunto de instrucciones.

Taxonomía de procesadores Flynn



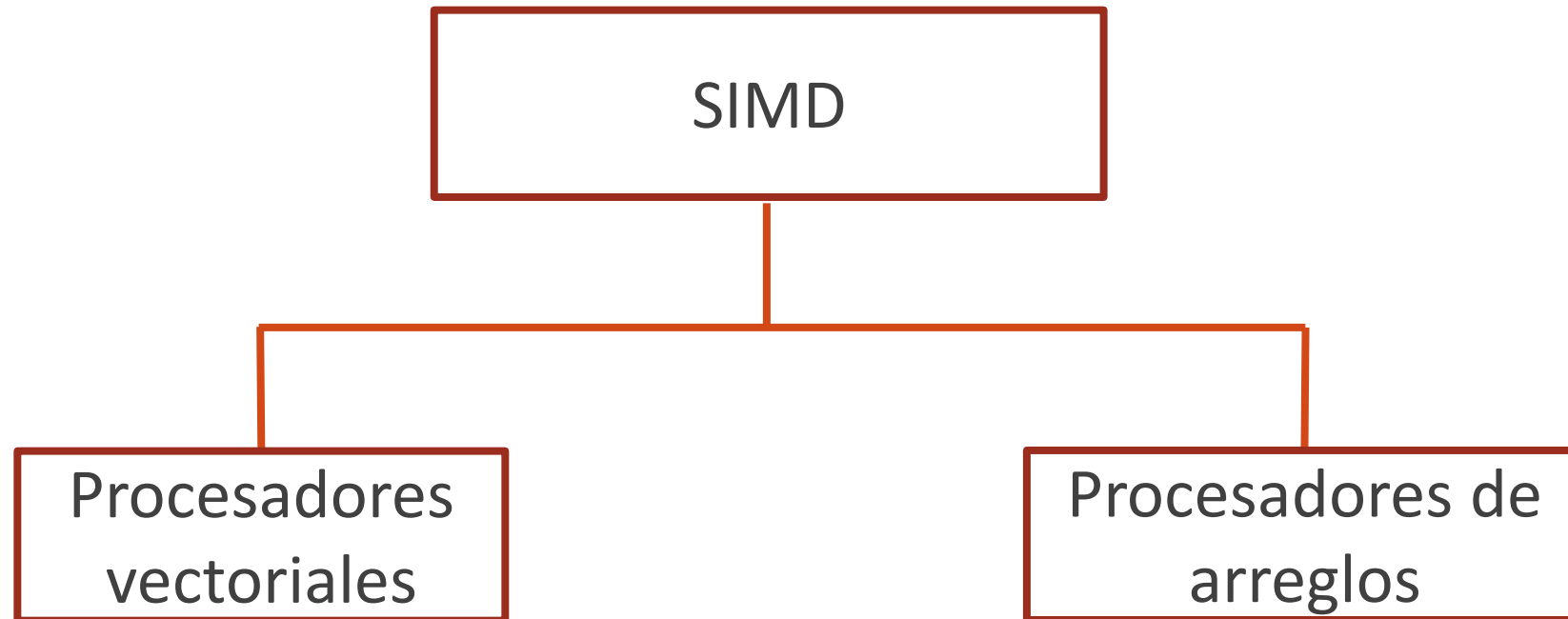
Taxonomía de procesadores - SISD

- Un único procesador ejecuta una sola secuencia de instrucciones para operar con datos almacenados en una sola memoria.



Taxonomía de procesadores - SIMD

- Cada elemento de procesamiento tiene una memoria de datos asociada, de modo que los procesadores ejecutan las instrucciones en diferentes conjuntos de datos.



Taxonomía de procesadores - MISD

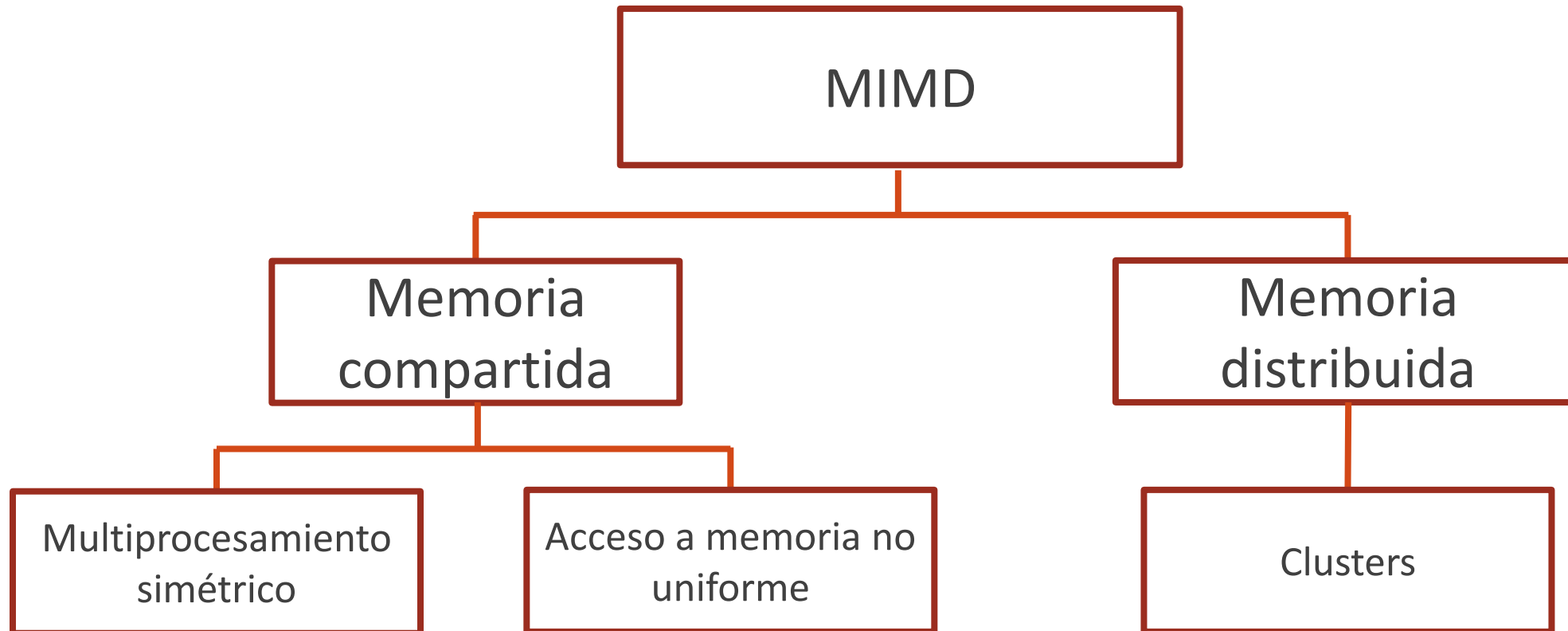
- Una secuencia de datos es transmitida a un conjunto de procesadores, donde cada uno ejecuta una secuencia diferente de instrucciones.



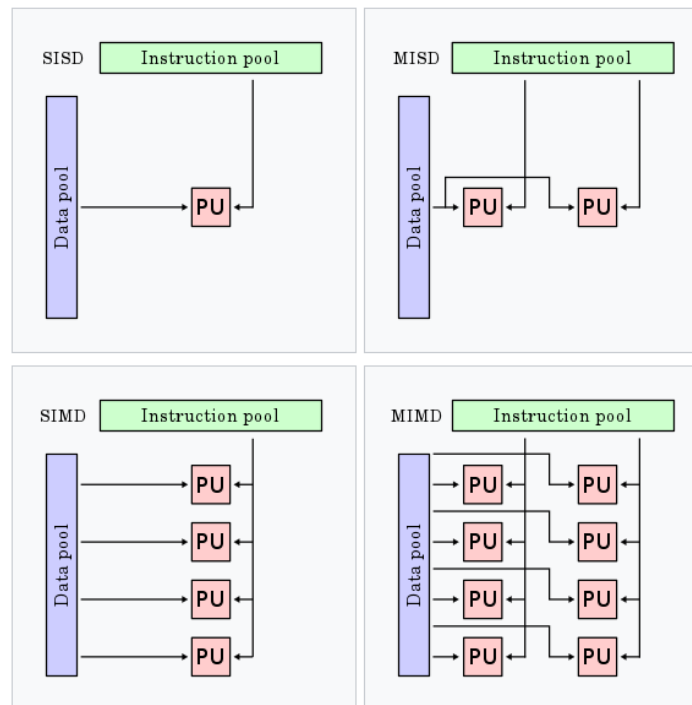
MISD

Taxonomía de procesadores - MIMD

- Un conjunto de procesadores ejecuta simultáneamente secuencias diferentes de instrucciones sobre conjuntos diferentes de datos.



Taxonomía de procesadores



https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

Paralelismo a nivel de instrucciones - SISD

- Se realizaba una ejecución “simultánea” de las instrucciones.
- Pipeline, Superpipeline.
- Superescalares.
 - Ejecución fuera de orden.
 - Ejecución dinámica (Tomasulo, Marcador).
- Calendarización estática:
 - VLIW.

Paralelismo a nivel de instrucciones - SISD

Surgen limitaciones que son inevitables:

- Dependencias
- Riesgos

MIMD

Los sistemas multiprocesador deben establecer las reglas para distribuir las tareas según las siguientes categorías:

- **Simetría:** simétricos o asimétricos.
- **Tiempo de acceso:** UMA o NUMA.
- **Organización de memoria:** centralizada o distribuida.
- **Direccionamiento:** unificado o privado.

MIMD

Surgen limitaciones que son inevitables:

- Necesitamos más hardware.
 - Mayor área y potencia.
- Sincronización
 - Tenemos más recursos compartidos.
- Memoria
 - Se realizan accesos simultáneos a memoria.

Taxonomías

Flynn's taxonomy

SISD – Single Instruction, Single Data stream

Defines serial computers

MISD – Multiple Instruction, Single Data stream

Multiple processors applying different instructions to a single data stream.

Possible but impractical.

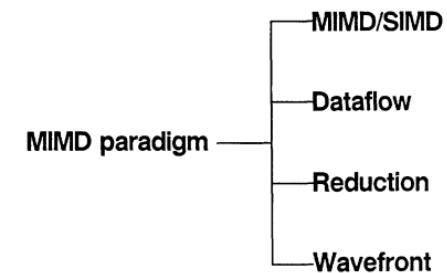
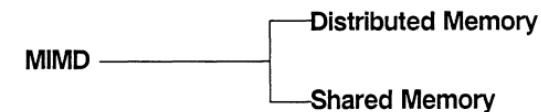
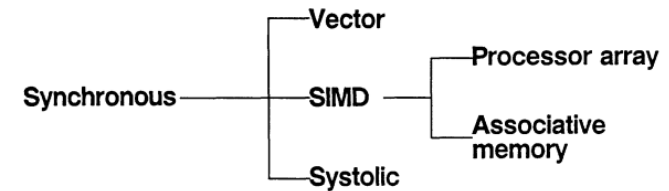
SIMD – Single Instruction, Multiple Data streams

Multiple processors simultaneously executing the same instruction on different data.

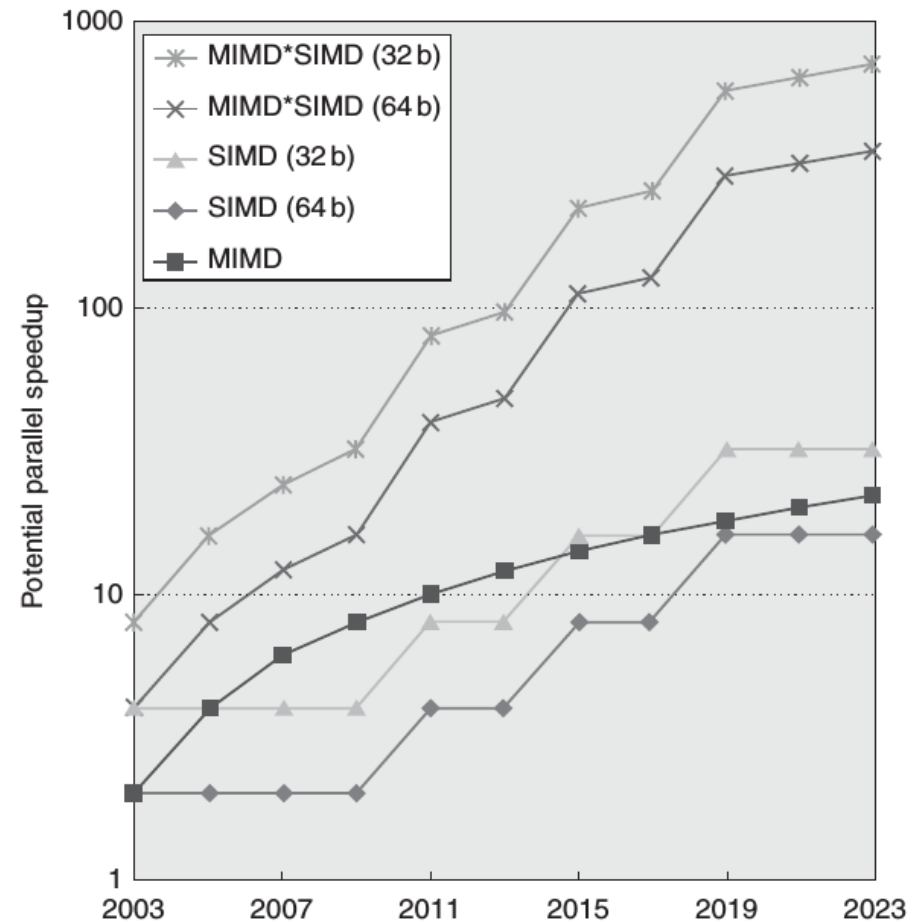
MIMD – Multiple Instruction, Multiple Data streams

Multiple processors autonomously executing diverse instructions on diverse data.

Duncan's taxonomy



Mejora potencial en computadoras x86



SIMD

- Los procesadores ejecutan las instrucciones en diferentes conjuntos de datos de forma paralela.
- Se utiliza en operaciones vectoriales.
- Toma un conjunto de datos, los coloca de manera secuencial, opera sobre esos registros y luego envía los datos a memoria.
- Al tratarse de vectores, sólo se hace una penalización de memoria por vector y no por cada elemento.

SIMD

- Sumar 64 números con una sola instrucción.
- Hardware envía 64 datos a 64 ALUs para obtener 64 resultados en el mismo ciclo de reloj.
- Máxima eficiencia en programas con alto paralelismo de datos.
 - Por ejemplo: un arreglo en un *for*.
- Arm Neon: extensión para procesadores Arm para instrucciones SIMD.
 - Mejorar multimedia, gráficos 2D/3D y videojuegos.

Tipo de procesamiento

- Escalar: instrucciones sobre un dato y generan un único resultado.
- Vectorial: instrucciones operan sobre un vector y generan vectores o escalares.

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

Aplicaciones

- **Procesamiento de imágenes.**
 - Filtros digitales, detección de formas, conversión de colores.
- **Procesamiento de audio.**
 - Filtros de frecuencia y convolución, detección de tonos.
- **Computación gráfica.**
 - Visualización, renderizado.
- **Aceleración de computación gráfica.**
 - GPGPU, algoritmos de encriptación.

Ejemplo – Flujo

Realizar la operación $C = (s \times A) + B$ requiere el siguiente flujo:

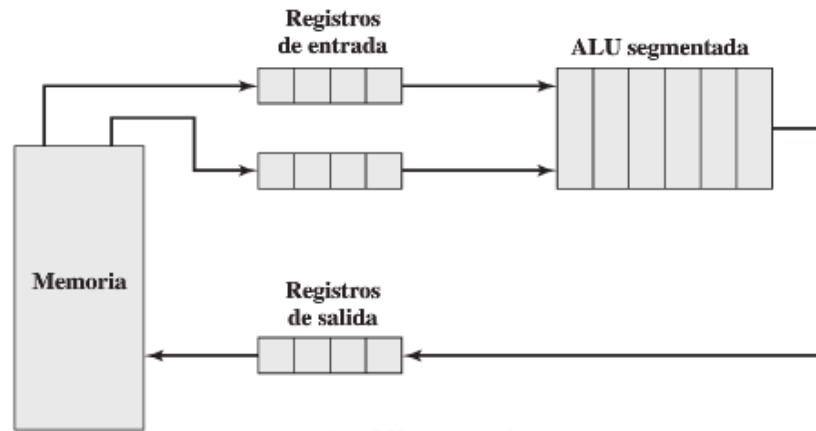
1. Cargar el vector A. $A \rightarrow r(VR1)$
2. Cargar el vector B. $B \rightarrow r(VR2)$
3. Multiplicar $(s \times A)$. $VR3 \rightarrow s \times VR1$
4. Sumar el resultado con B. $VR4 \rightarrow VR1 + VR2$
5. Guardar en C. $VR4 \rightarrow C$

Organizaciones vectoriales

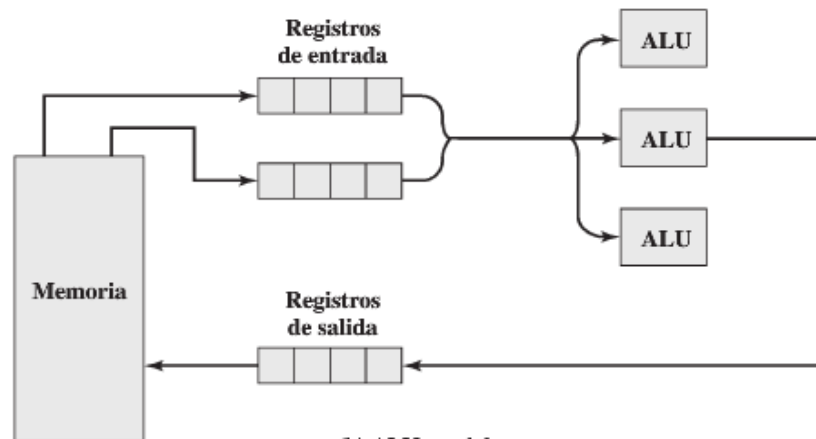
Para implementar las operaciones vectoriales se puede utilizar:

- ALUs segmentadas.
- ALUs paralelas.
- Procesadores paralelos.

Organizaciones vectoriales

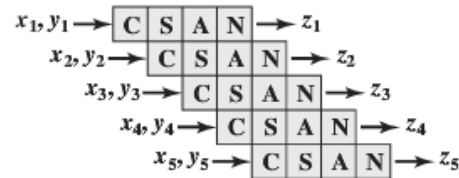


(a) ALU segmentada

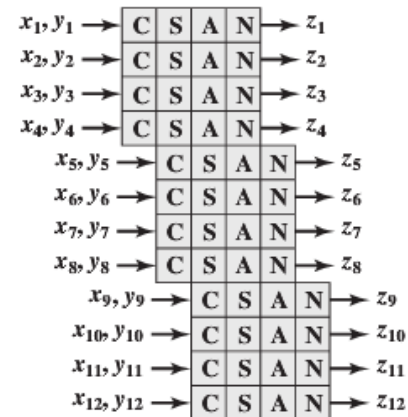


(b) ALU paralelas

Organizaciones vectoriales



(a) ALU segmentada

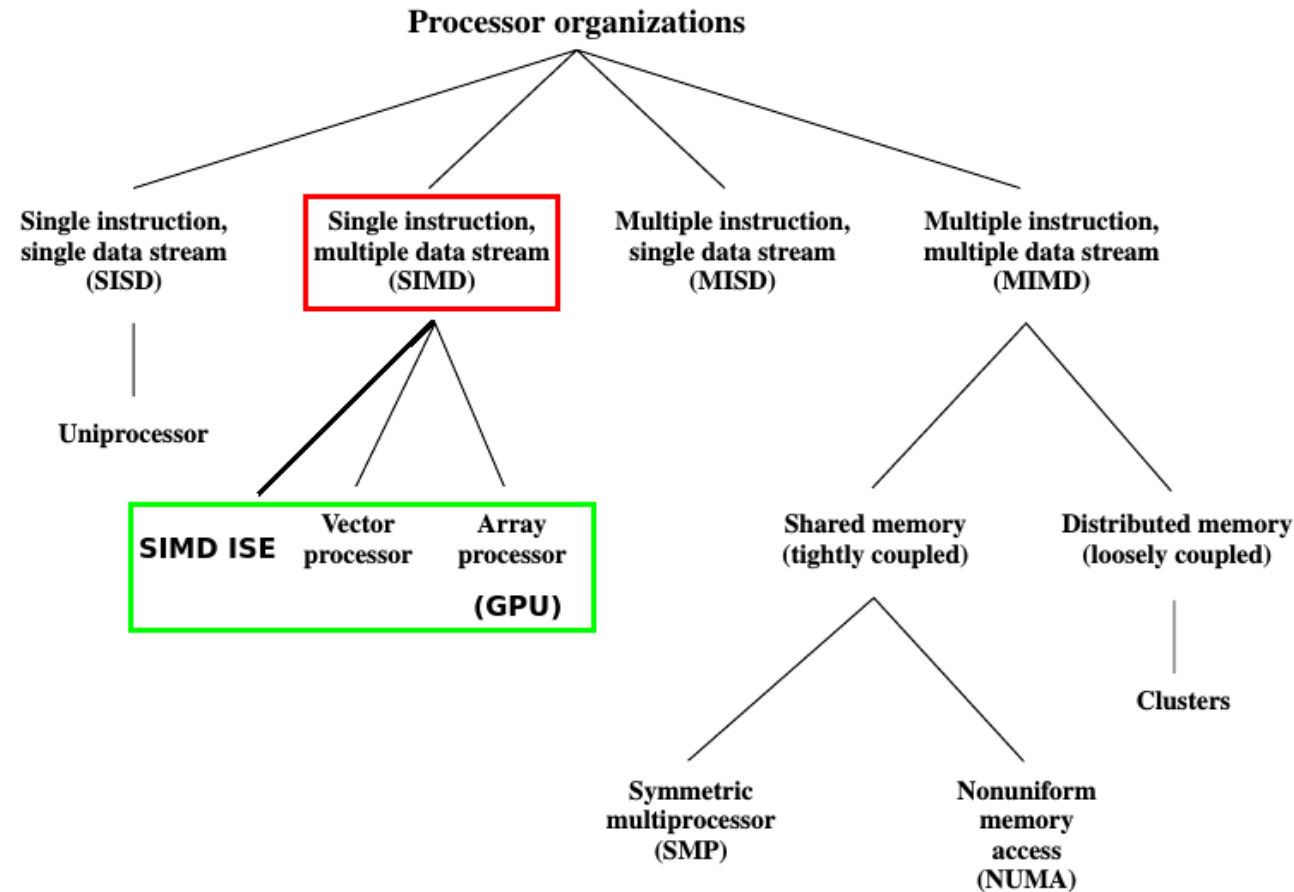


(b) Cuatro ALU paralelas

Arquitectura vectorial

- Soportar operaciones vectoriales.
 - Operaciones aritméticas, cargas, almacenamientos.
- No necesariamente hay operaciones de control de flujo.
- Deben existir registros vectoriales.

Flynn - SIMD

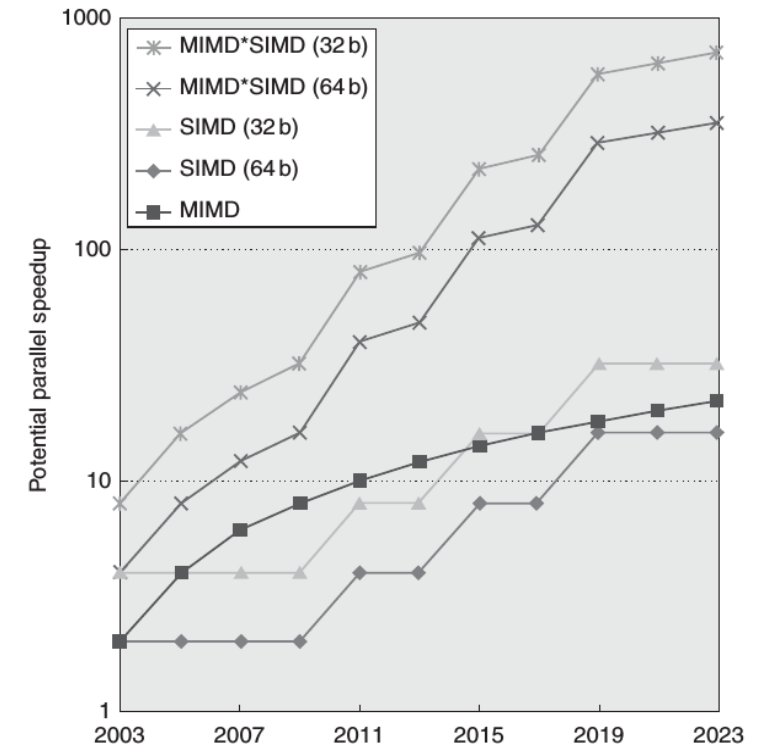


Variaciones de SIMD

- **Arquitecturas vectoriales.**
 - Ejecución por *pipeline* de muchas operaciones de datos.
- **Extensión SIMD.**
 - Habilitar la posibilidad de brindar operaciones de datos en paralelo.
- **Unidades de procesamiento gráfico.**
 - Ofrecer mayor rendimiento que en los multicore tradicionales.

SIMD vs MIMD

- ¿Por qué usar procesamiento de múltiples datos?
- ¿Qué aplicaciones requieren esta cantidad de procesamiento? 2 Ejemplos.
- Ventajas y desventajas de MIMD.
- Ventajas y desventajas de SIMD.

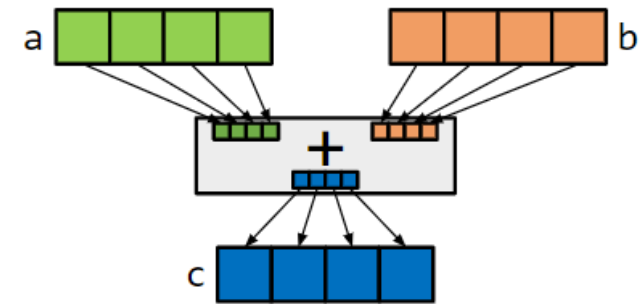


Arquitectura vectorial

Casos para procesamiento de datos

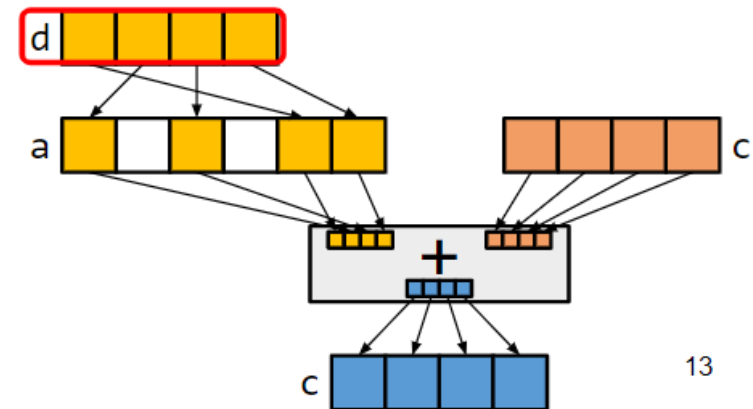
- Datos contiguos (en memoria).

```
38 void* add(void* arg)
39 {
40     for (int i = 0; i <= MAX_SIZE; i++) {
41         c[i] = a[i] + b[i];
42     }
43 }
```

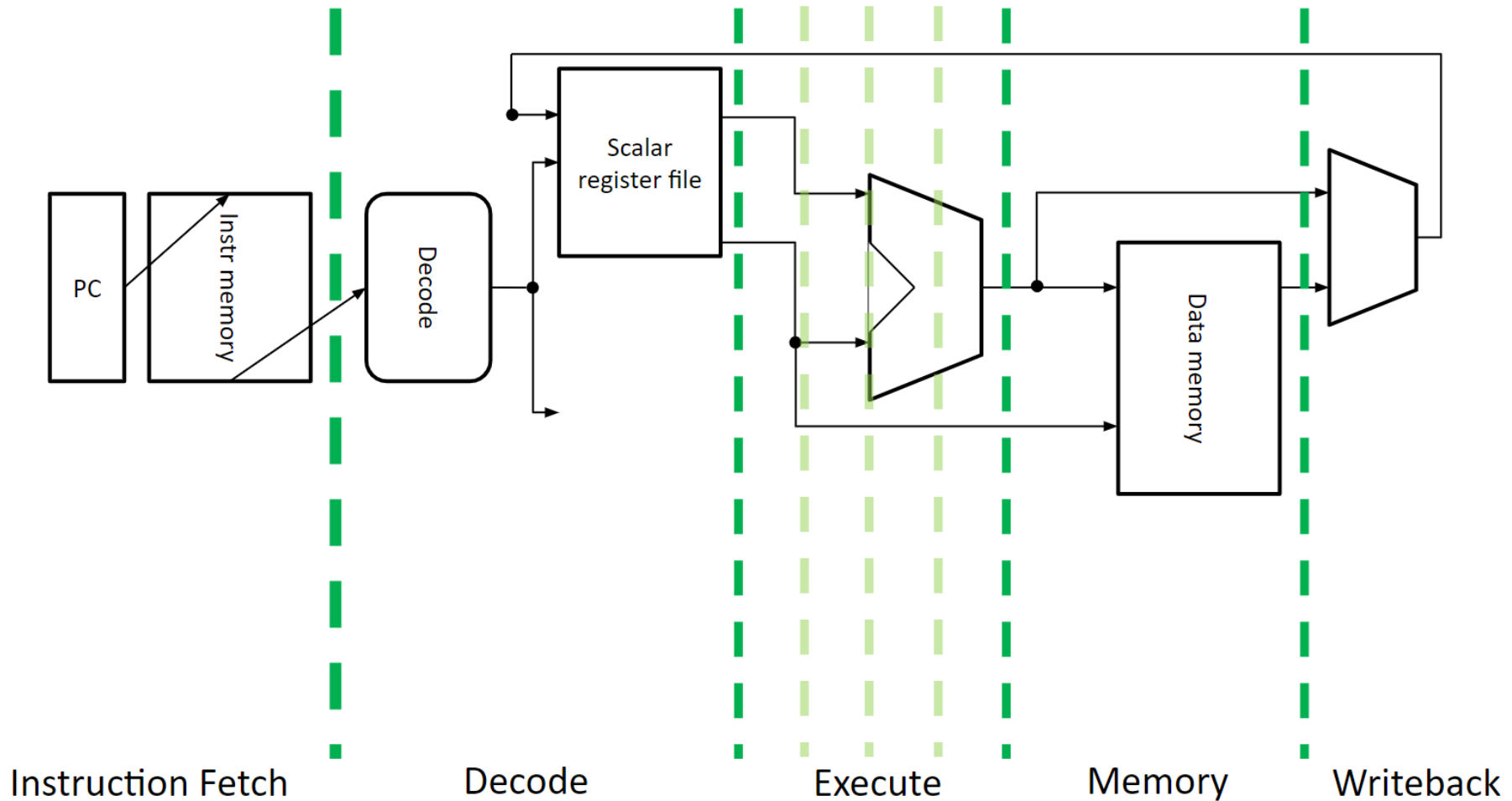


- Datos dispersos (en memoria).

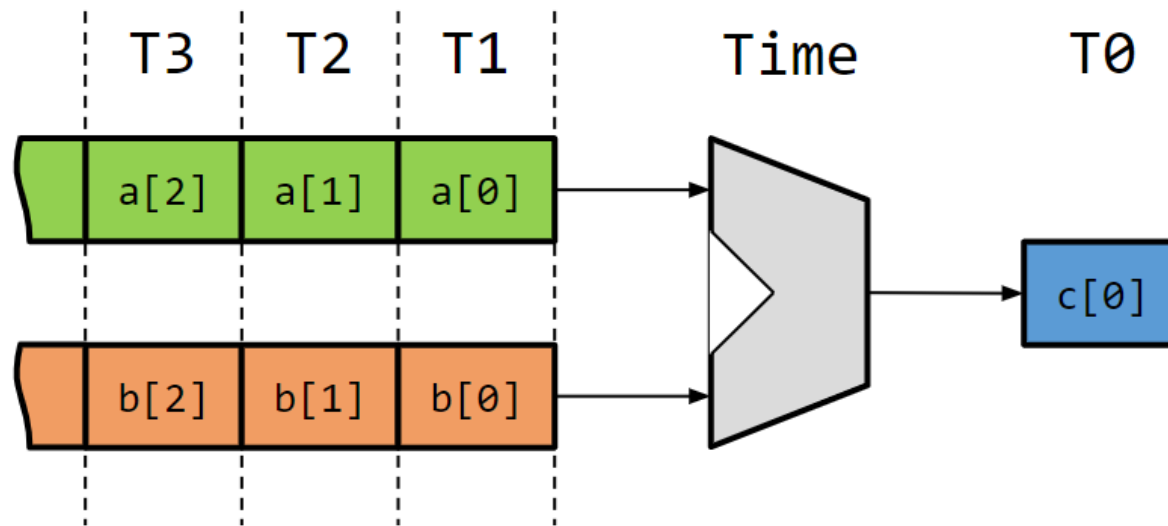
```
38 void* add(void* arg)
39 {
40     for (int i = 0; i <= MAX_SIZE; i++) {
41         c[i] = a[d[i]] + b[i];
42     }
43 }
```



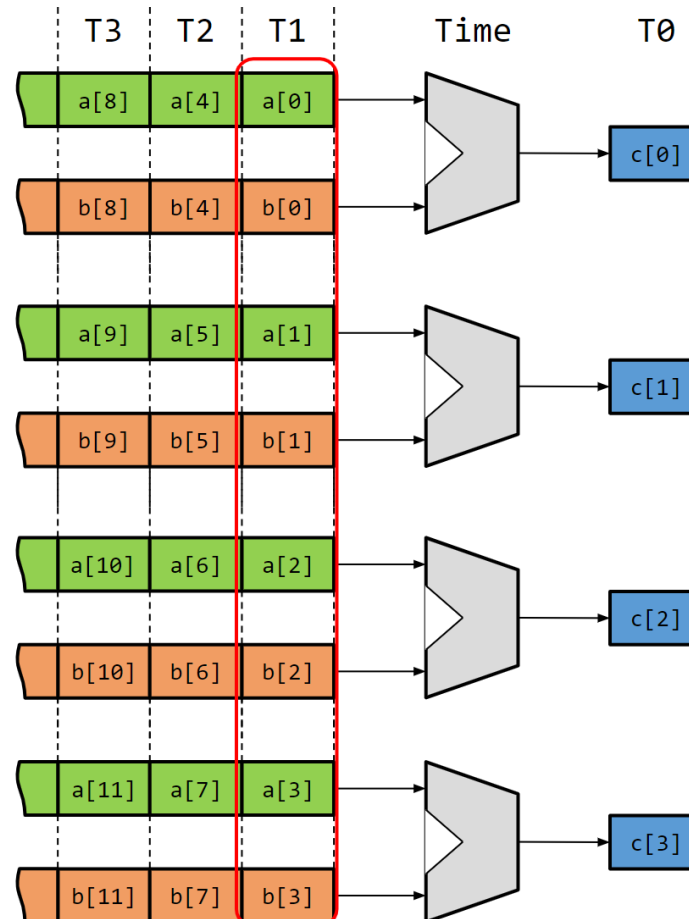
Arquitectura vectorial



Arquitectura vectorial



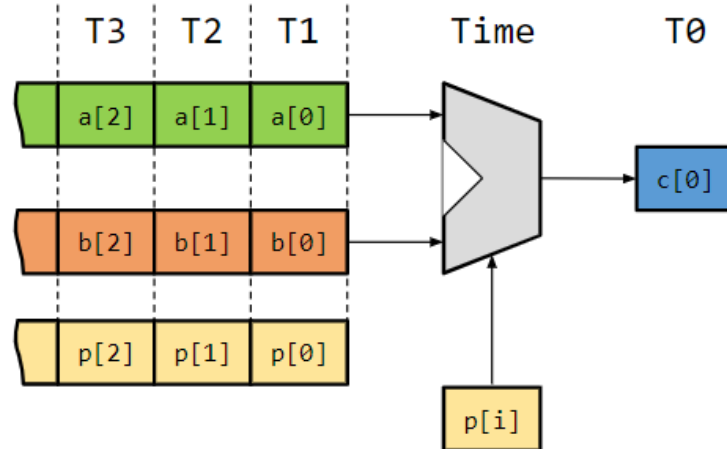
Arquitectura vectorial



Arquitectura vectorial

Uso de predicado / Vector máscara

- Arreglo de un bit. Permite control sobre las operaciones de vector.

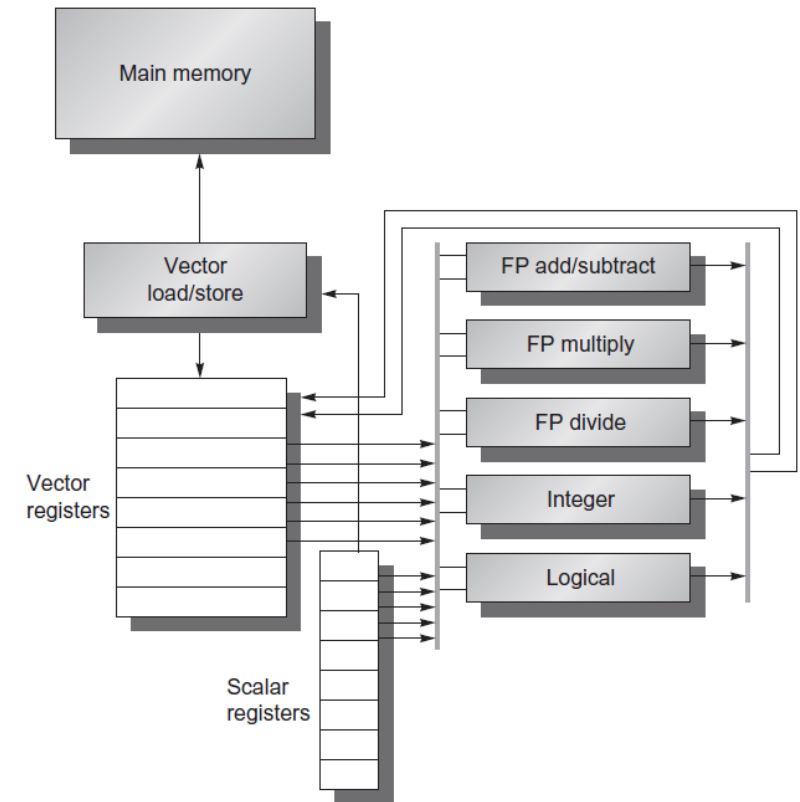


```
1  if (p[i] == 1) {  
2      c[i] = a[i] + b[i];  
3  }  
4  else {  
5      c[i] = a[i];  
6  }
```

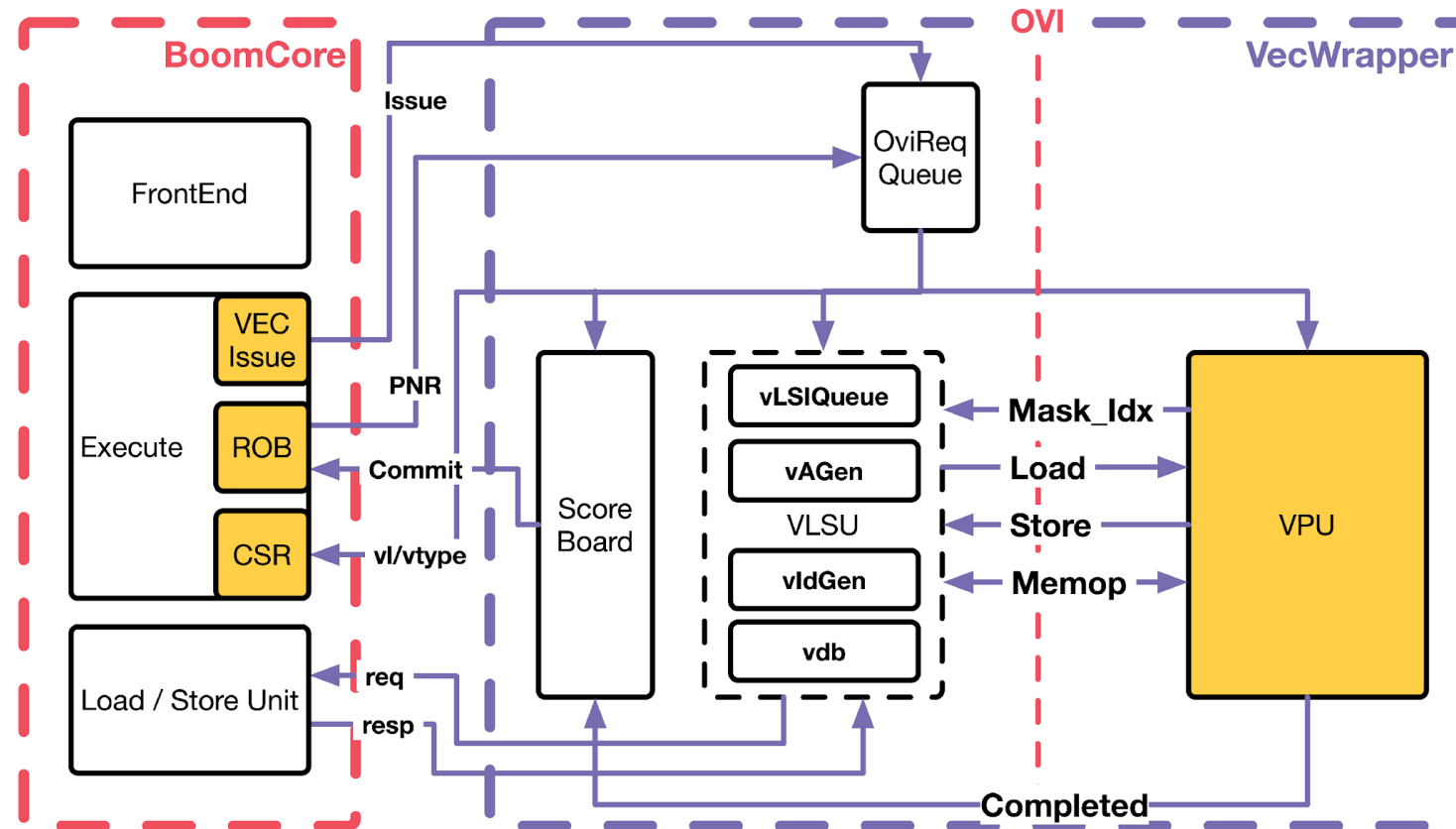
| | | | |
|------|------|------|------|
| 0 | 1 | 0 | 1 |
| a[3] | c[2] | a[1] | c[0] |

Arquitectura vectorial – RV64V

- Registros vectoriales. Cada registro almacena un vector. RV64V tiene 32 de 64 bits.
- FU vectoriales. Puede estar con pipeline.
- Unidad load/store.
- Grupo de registros escalares.



Arquitectura vectorial – Ocelot



Arquitectura vectorial – Ocelot

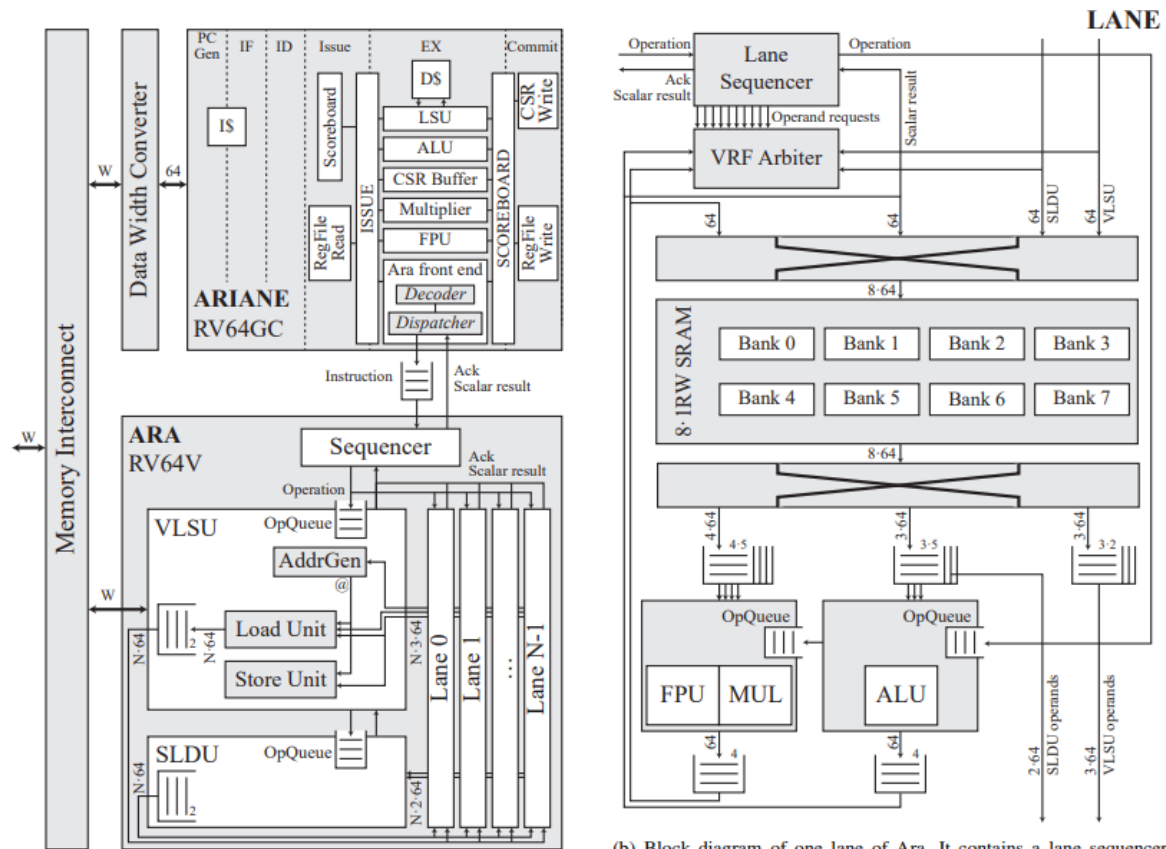
In-order Vector Issue Queue: Functioning similarly to other issue queues, it monitors write-back signals from INT/FP/MEM units, activating vector instructions with dependencies on INT/FP registers.

Request Queue: This queue manages vector instructions that have resolved their INT/FP register dependencies. It employs a Point of No Return (PNR) mechanism to halt the process if the oldest vector instruction is speculative.

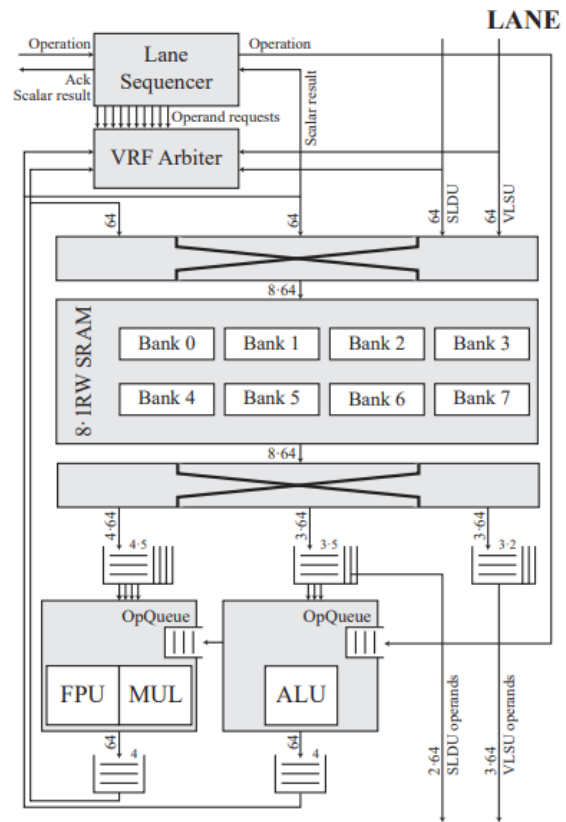
Detached Vector Load/Store Unit: In alignment with the OVI specifications, the core must generate vector load/store requests. Leveraging this requirement, we have created a fully detached unit to enhance performance, enabling unit-stride and strided loads to be executed out-of-order.

Augmented Load/Store Unit: Modifications to the existing Load/Store Unit (LSU) have been made to handle dependencies between scalar and vector Load/Store requests effectively.

Arquitectura vectorial – RV64



(a) Block diagram of an Ara instance with N parallel lanes. Ara receives its commands from Ariane, a RV64GC scalar core. The vector unit has a main sequencer; N parallel lanes; a Slide Unit (SLDU); and a Vector Load/Store Unit (VLSU). The memory interface is W bit wide.



(b) Block diagram of one lane of Ara. It contains a lane sequencer (handling up to 8 vector instructions); a 16 KiB vector register file; ten operand queues; an integer Arithmetic Logic Unit (ALU); an integer multiplier (MUL); and a Floating Point Unit (FPU).

Ara: A 1 GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI

Matheus Cavalcante,* Fabian Schuiki,* Florian Zaruba,* Michael Schaffner,* Luca Benini,*[†] *Fellow, IEEE*

RV64V <https://arxiv.org/pdf/1906.00478.pdf>

Arquitectura vectorial

Código fuente para DAXPY. X y Y tienen 32 elementos. x5 y x6 son las direcciones iniciales de X y Y.

Here is the RISC-V code:

```
        fld      f0,a           # Load scalar a
        addi     x28,x5,#256    # Last address to load
Loop:   fld      f1,0(x5)       # Load X[i]
        fmul.d   f1,f1,f0      # a × X[i]
        fld      f2,0(x6)      # Load Y[i]
        fadd.d   f2,f2,f1      # a × X[i] + Y[i]
        fsd      f2,0(x6)      # Store into Y[i]
        addi     x5,x5,#8       # Increment index to X
        addi     x6,x6,#8       # Increment index to Y
        bne      x28,x5,Loop    # Check if done
```

Here is the RV64V code for DAXPY:

```
vsetdcfg  4*FP64              # Enable 4 DP FP vregs
fld       f0,a                # Load scalar a
vld       v0,x5               # Load vector X
vmul      v1,v0,f0            # Vector-scalar mult
vld       v2,x6               # Load vector Y
vadd      v3,v1,v2            # Vector-vector add
vst       v3,x6               # Store the sum
vdisable
```

{S,D}APY operations

Given X and Y as vectors and A as a scalar $\Rightarrow Y \leftarrow AX + Y$

S \Rightarrow Single Precision

D \Rightarrow Double Precision

Arquitectura vectorial

<https://godbolt.org/z/xPYh7Ws84>

```
1 double X[32] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0};
2 double Y[32] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0, 32.0};
3 double a = 1.0;
4 void add(){
5
6     for (int i = 0; i < 32; ++i){
7         Y[i] += a*X[i] ;
8     }
9 }
```

RISC-V (64-bits) gcc (trunk) (Editor #1)

RISC-V (64-bits) gcc (trunk) -O2 -march=rv64gv

Output... Filter... Libraries Overrides Add r

```
1  add:
2      lui    a4,%hi(a)
3      lui    a5,%hi(.L2)
4      fld    fa5,%lo(a)(a4)
5      addi   a5,a5,%lo(.L2)
6      addi   a4,a5,256
7      mv     a3,a4
8      vsetivli zero,2,e64,m1,ta,ma
9      vfmv.v.f v3,fa5
10
11 .L2:
12     vle64.v v1,0(a4)
13     vle64.v v2,0(a5)
14     vfmadd.vv v1,v3,v2
15     vse64.v v1,0(a5)
16     addi   a5,a5,16
17     addi   a4,a4,16
18     bne    a3,a5,.L2
19     ret
```

Arquitectura vectorial

<https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-config>

```
RISC-V (64-bits) gcc (trunk) (Editor #1) x
RISC-V (64-bits) gcc (trunk) -O2 -march=rv64gv
A Output... Filter... Libraries Overrides + Add r
1 add:
2     lui    a4,%hi(a)
3     lui    a5,%hi(.LANCHOR0)
4     fld    fa5,%lo(a)(a4)
5     addi   a5,a5,%lo(.LANCHOR0)
6     addi   a4,a5,256
7     mv     a3,a4
8     vsetivli zero,2,e64,m1,ta,ma
9     vfmv.v.f v3,fa5
10 .L2:
11     vle64.v v1,0(a4)
12     vle64.v v2,0(a5)
13     vfmadd.vv v1,v3,v2
14     vse64.v v1,0(a5)
15     addi   a5,a5,16
16     addi   a4,a4,16
17     bne    a3,a5,.L2
18     ret
```

A set of instructions is provided to allow rapid configuration of the values in `v1` and `vtype` to match application needs. The `vset{i}v1{i}` instructions set the `vtype` and `v1` CSRs based on their arguments, and write the new value of `v1` into `rd`.

```
vsetvli rd, rs1, vtypei # rd = new v1, rs1 = AVL, vtypei = new vtype setting
vsetivli rd, uimm, vtypei # rd = new v1, uimm = AVL, vtypei = new vtype setting
vsetvl rd, rs1, rs2 # rd = new v1, rs1 = AVL, rs2 = new vtype value
```

13.16. Vector Floating-Point Move Instruction

The vector floating-point move instruction *splats* a floating-point scalar operand to a vector register group. The instruction copies a scalar floating-point register value to all active elements of a vector register group. This instruction is encoded as an unmasked instruction (`vm=1`). The instruction must have the `vs2` field set to `v0`, with all other values for `vs2` reserved.

```
vfmv.v.f vd, rs1 # vd[i] = f[rs1]
```

7.4. Vector Unit-Stride Instructions

Vector unit-stride loads and stores

vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)

```
vle8.v vd, (rs1), vm # 8-bit unit-stride load
```

```
vle16.v vd, (rs1), vm # 16-bit unit-stride load
```

```
vle32.v vd, (rs1), vm # 32-bit unit-stride load
```

```
vle64.v vd, (rs1), vm # 64-bit unit-stride load
```

Investigar

- ¿Qué afecta el tiempo de ejecución vectorial?
- ¿Qué diferencia las extensiones SIMD a un procesamiento vectorial genérico?
- ¿Qué son x86 AVX/AMX, y ARM SVE/SME?
- ¿Qué son intrinsics?

Referencias

- Stallings, W. (2003). Computer organization and architecture: designing for performance. Pearson Education India.
- Hennessy, J., & Patterson, D. (2012). Computer Architecture: A Quantitative Approach (5th ed.). Elsevier Science.

CE4302 – Arquitectura de Computadores II

Introducción al paralelismo a nivel de datos

PROFESOR: ING. LUIS BARBOZA ARTAVIA