ESTRUCTURA DE COMPUTADORES

Tema 5. Arquitecturas avanzadas

Dpto. Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

ÍNDICE

- Aritmética en PF
- Introducción al procesamiento paralelo
- Estructura de los multiprocesadores de memoria compartida
- Estructura de los multiprocesadores de memoria distribuida
- Arquitectura vectorial
- Extensiones multimedia
- GPUs

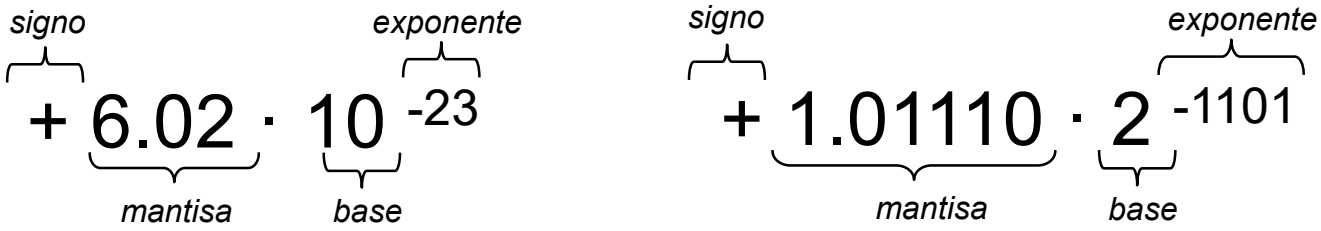
- Bibliografía
  - “Computer Architecture: A Quantitative Approach”, Hennessy & Patterson, 6th ed., 2017

ÍNDICE

- **Aritmética en PF**
- Introducción al procesamiento paralelo
- Estructura de los multiprocesadores de memoria compartida
- Estructura de los multiprocesadores de memoria distribuida
- Arquitectura vectorial
- Extensiones multimedia
- GPUs

REPRESENTACIÓN EN COMA FLOTANTE

- La representación en coma flotante está basada en la **notación científica**:
  - La coma decimal no se halla en una posición fija dentro de la secuencia de bits, sino que su posición se indica como una potencia de la base:



- En todo número en coma flotante se distinguen tres componentes:
  - **Signo**: indica el signo del número (0= positivo, 1=negativo)
  - **Mantisa**: contiene la magnitud del número (en binario puro)
  - **Exponente**: contiene el valor de la potencia de la base (sesgado)
  - La **base** queda implícita y es común a todos los números, la más usada es 2

Representación en coma flotante

- El **valor** de la secuencia de bits (s, e_{p-1}, ..., e_0, m_{q-1}, ..., m_0) es: $(-1)^s \cdot V(m) \cdot 2^{V(e)}$
- Dado que un mismo número puede tener varias representaciones

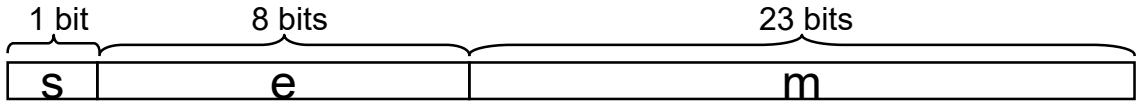
  $0.110 \cdot 2^5 = 110 \cdot 2^2 = 0.0110 \cdot 2^6$

  los números suelen estar normalizados:

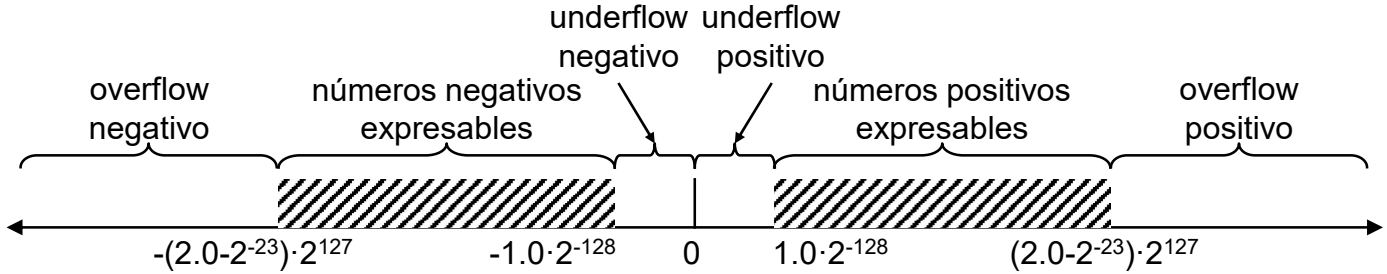
  - Un número está normalizado si tiene la forma $1.xx... \cdot 2^{xx...}$ (ó $0.1xx ... \cdot 2^{xx...}$)
  - Dado que los números normalizados en base 2 tienen siempre un 1 a la izquierda, éste suele quedar implícito (pero debe ser tenido en cuenta al calcular el valor de la secuencia)

REPRESENTACIÓN EN COMA FLOTANTE

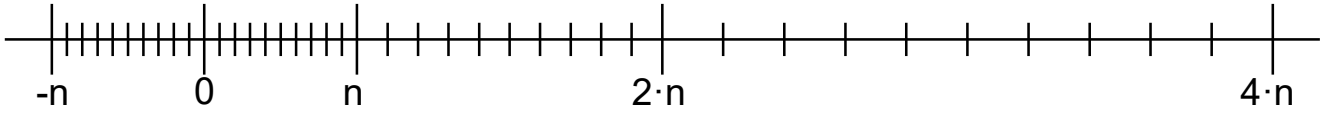
- Sea el siguiente formato de coma flotante de 32 bits (base 2, normalizado)



  - El rango de valores representable por cada uno de los campos es:
    - **Exponente** (8 bits con sesgo de 128) : -128 ... +127
    - **Mantisa** (23 bits normalizados) : los valores binarios representables oscilan entre 1.00...00 y 1.11...11, es decir entre 1 y $2-2^{-23}$ (2-ulp) (1.11...1 = 10.00...0 - 0.0...1)



  - Obsérvese que la cantidad de números representables es 2^{32} (igual que en coma fija). Lo que permite la representación en coma flotante es ampliar el rango representable a costa de aumentar el espacio entre números representable (un espacio que no es uniforme).



IEEE 754

- **2 formatos** con signo explícito, representación sesgada del exponente (sesgo igual a $(2^{n-1}-1)$), mantisa normalizada con un 1 implícito (1.M) y base 2.
  - ***precisión simple*** (32 bits): 1 bit de signo, 8 de exponente, 23 de mantisa
    - $1.0 \cdot 2^{-126}$ ... $(2-2^{-23}) \cdot 2^{127} = 1.2 \cdot 10^{-38}$ .. $3.4 \cdot 10^{38}$
  - ***precisión doble*** (64 bits): 1 bit de signo, 11 de exponente, 52 de mantisa
    - $1.0 \cdot 2^{-1022}$ ... $(2-2^{-52}) \cdot 2^{1023} = 2.2 \cdot 10^{-308}$ .. $1{,}8 \cdot 10^{308}$
- **2 formatos** ampliados para cálculos intermedios (43 y 79 bits).

IEEE 754

- **Codificaciones** con significado especial
  - ***Infinito*** (e=255, m=0): representan cualquier valor de la región de overflow
  - ***NaN*** (*Not-a-Number*) (e=255, m>0): se obtienen como resultado de operaciones inválidas
  - ***Número denormalizado*** (e=0, m>0): es un número sin normalizar cuyo bit implícito se supone que es 0. Al ser el exponente 0, permiten representar números en las regiones de underflow. El valor del exponente es el del exponente más pequeño de los números no denormalizados: -126 en precisión simple y –1022 en doble.
  - ***Cero*** (e=0, m=0): número no normalizado que representa al cero (en lugar de al 1)
- **Excepciones**:
  - **Operación inválida**: $\infty \pm \infty$, $0 \times \infty$, $0 \div 0$, $\infty \div \infty$, x mod 0, $\sqrt{x}$ cuando $x<0$, $x=\infty$
  - **Inexacto**: el resultado redondeado no coincide con el real
  - **Overflow** y **underflow**
  - **División por cero**

- Ejemplo:

0	0110 1000	101 0101 0100 0011 0100 0010

- Signo: 0 => positivo
- Exponente:
  - $0110\ 1000_{dos} = 104_{diez}$
  - Eliminación del sesgo: 104 - 127 = -23
- Mantisa:

$1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + ...$
$= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} = 1.0 + 0.666115$

Representa: $1.666115_{diez} * 2^{-23} \sim 1.986 * 10^{-7}$

Redondeo:

- El estándar exige que el resultado de las operaciones sea el mismo que se obtendría si se realizasen con precisión absoluta y después se redondease.
- Hacer la operación con precisión absoluta no tiene sentido pues se podrían necesitar operandos de mucha anchura.

- **Existen 4 modos de redondeo**:
  - Redondeo al **más cercano** (al par en caso de empate)
  - Redondeo a **más infinito** (por exceso)
  - Redondeo a **menos infinito** (por defecto)
  - Redondeo a **cero** (truncamiento)

- **Al realizar una operación ¿cuántos bits adicionales se necesitan para tener la precisión requerida?**

- Suma

  $1 \leq s1 < 2$

  $ulp \leq s2 < 2$

  Por tanto s=s1+s2 cumplirá:

  $1 < s3 < 4$

  Si s3>2 se deberá normalizar desplazando a la derecha una posición, y ajustando el exponente.

- Redondeo:

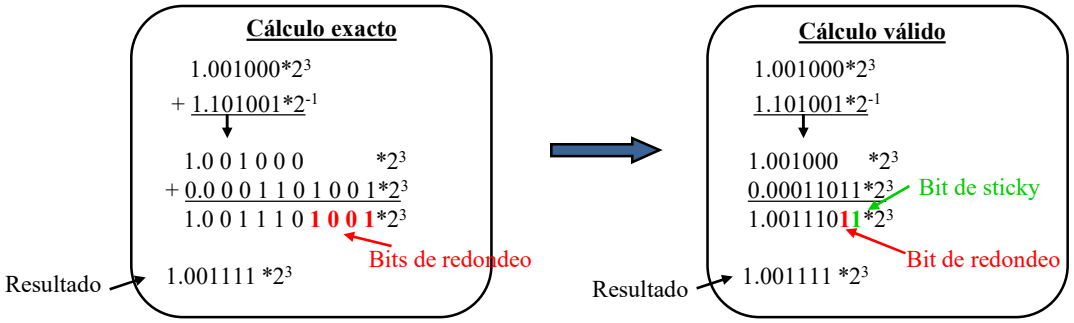
  Caso 1: e1=e2 y s3>2

$$1.001000 * 2^3$$
$$1.110001 * 2^3$$
$$10.111001 * 2^3$$
$$1.0111001 * 2^4$$

Bit de redondeo

- Suma
- Redondeo:

  Caso 2: e1-e2>0



SUMA/RESTA EN COMA FLOTANTE

- Resta:

  Caso 1: e1=e2

  $1 \leq si < 2 \Rightarrow s \subset [0,1) \Rightarrow$ Normalización

  No se necesitan bits adicionales.

$1.001111 * 2^3$

$- \underline{1.001001 * 2^3}$

$0.000110 * 2^3$

$1.100000 * 2^{-1}$

Normalización

- Caso 2: e1-e2=1 y s>0,5

  $1 \leq s1 < 2$

  $0.5 \leq s2 < 1$

  } ulp $\leq s < 1.5$

Cálculo exacto

$1.0\,0\,1\,1\,1\,1 * 2^3$

$- \underline{1.0\,0\,1\,0\,0\,1 * 2^2}$

$1.\,0\,0\,1\,1\,1\,1 \quad * 2^3$

$-\underline{0.\,1\,0\,0\,1\,0\,0\,1 \quad * 2^3}$

$0.\,1\,0\,1\,0\,1\,0\,\mathbf{1} \quad * 2^3$

Bit de guarda

Resultado $1.010101 * 2^2$

Cálculo exacto

$1.1\,1\,1\,0\,0\,0\,0 * 2^3$

$- \underline{1.1\,0\,0\,0\,0\,0\,1 * 2^2}$

$1.\,0\,0\,1\,1\,1\,1 \quad * 2^3$

$-\underline{0.\,1\,1\,0\,0\,0\,0\,1 \quad * 2^3}$

$1.\,0\,0\,0\,1\,1\,1\,\mathbf{1} \quad * 2^3$

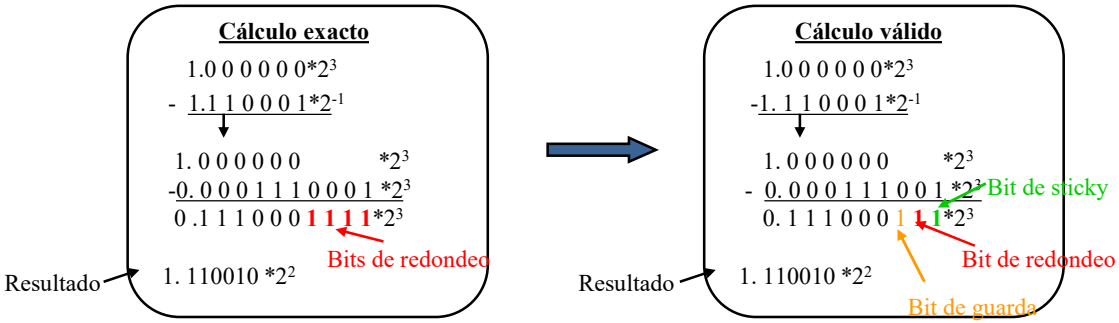
Bit de redondeo

Resultado $1.001000 * 2^3$

- Resta:

Caso3: e1-e2>1

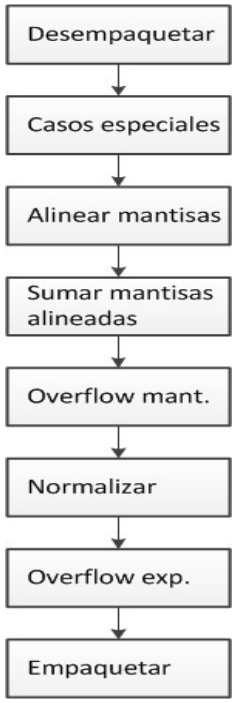
$$\left.\begin{array}{l} 1 \le s1 < 2 \\ ulp \le s2 < 0.5 \end{array}\right\} \; 0{,}5 < s \le 2$$



- Al realizar una operación ¿cuántos bits adicionales se necesitan para tener la precisión requerida?
  - Un bit **r** para el redondeo
  - Un bit **s** (sticky) para determinar, cuando r=1, si el número está por encima de 0,5

Operaciones de redondeo		
Tipo de redondeo	Signo del resultado≥0	Signo del resultado<0
- ∞		+1 si (r or s)
+ ∞	+1 si (r or s)	
0		
Más próximo	+1 si (r and p_0) or (r and s)	+1 si (r and p_0) or (r and s)

- Objetivo: Sumar en formato punto fijo de números con el exponente alineado

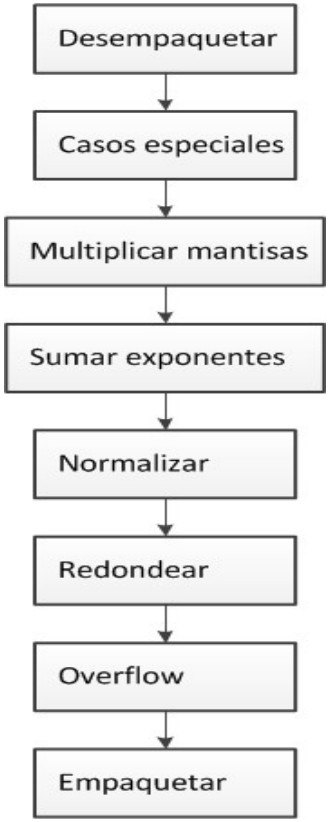


Ejemplo:

$$+\begin{array}{l} 9.555\cdot10^{4} \\ \underline{9.996\cdot10^{7}} \end{array} \longrightarrow +\begin{array}{l} 0.009555\cdot10^{7} \\ \underline{9.996000\cdot10^{7}} \end{array} \quad 10.005555\cdot10^{7} \longrightarrow 1.0005555\cdot10^{8} \longrightarrow 1.001\cdot10^{8}$$

alinear normalizar redondeo

IEEE 754: Multiplicación



Ejemplo:

$$\begin{array}{r} 4.555\cdot10^{4} \\ \times\ 2.996\cdot10^{7} \\ \hline 13.646780\cdot10^{11} \end{array} \longrightarrow 1.3646780\cdot10^{12} \longrightarrow 1.365\cdot10^{12}$$

normalizar redondeo

ÍNDICE

- Aritmética en PF
- **Introducción al procesamiento paralelo**
- Estructura de los multiprocesadores de memoria compartida
- Estructura de los multiprocesadores de memoria distribuida
- Arquitectura vectorial
- Extensiones multimedia
- GPUs

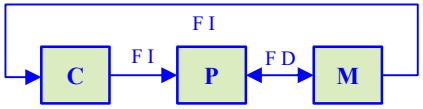
Clasificación de *Flynn* de los computadores paralelos

- Flynn clasificó los computadores paralelos atendiendo al carácter Simple (S) o Múltiple (M) del Flujo de Instrucciones y el Flujo de Datos

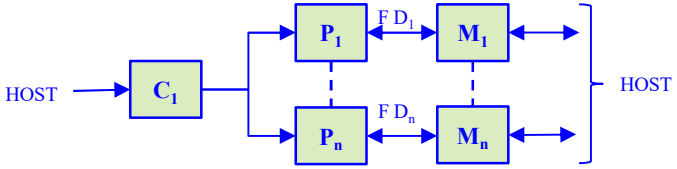
CLASIFICACION DE FLYNN		Flujo Instrucciones	
		S	M
Flujo Datos	S	SISD	MISD
	M	SIMD	MIMD

C = Control
P = Proceso (Unidad)
M = Memoria
MC = Memoria Compartida
FI = Flujo de Instrucciones
FD = Flujo de Datos

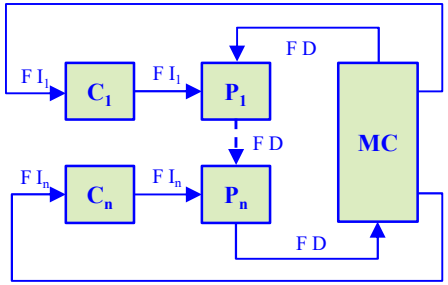
SISD (Single Instruction Single Data)



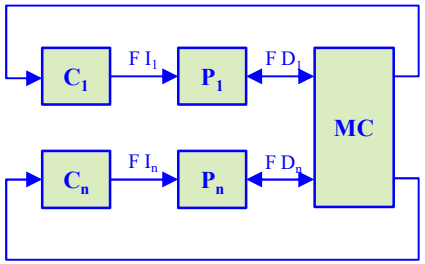
SIMD (Single Instruction Multiple Data)



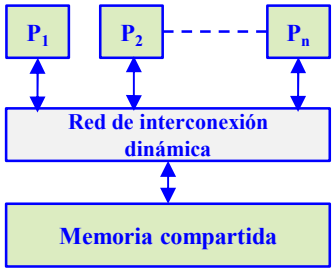
MISD (Multiple Instruction Single Data)



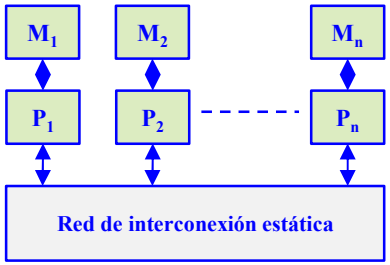
MIMD (Multiple Instruction Multiple Data)



- **Clasificación de los multiprocesadores por la ubicación de la memoria**
  - Multiprocesadores de memoria compartida
    - Todos los procesadores acceden a una memoria común
    - La comunicación entre procesadores se hace a través de la memoria



  - Multiprocesadores de memoria distribuida o multicomputadores
    - Cada procesador tiene su propia memoria
    - La comunicación se realiza por intercambio explicito de mensajes a través de una red

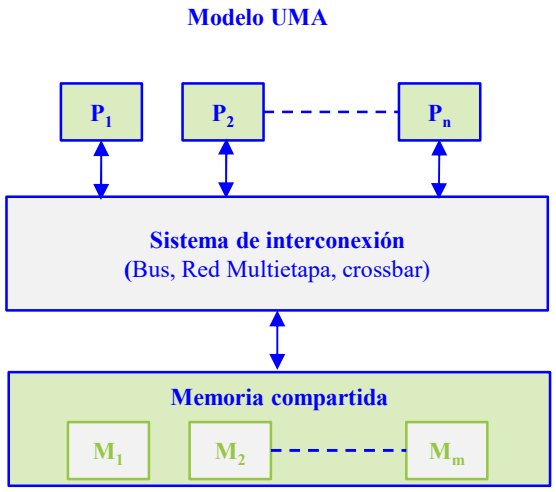


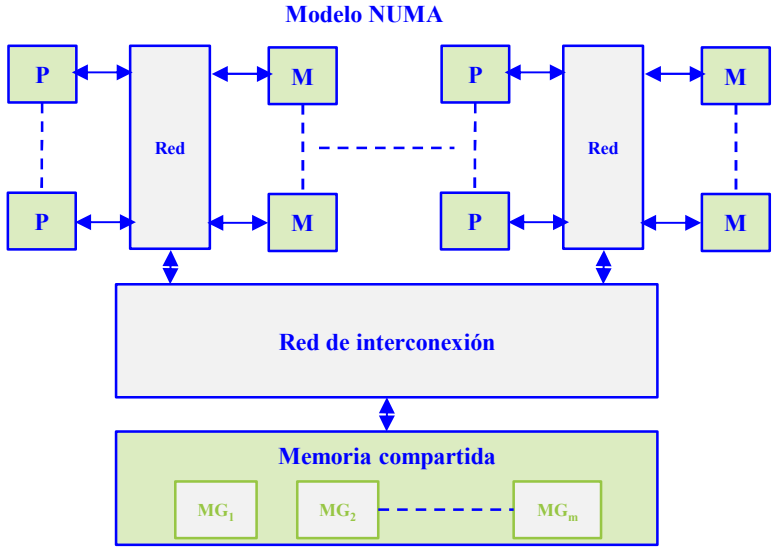
ÍNDICE

- Aritmética en PF
- Introducción al procesamiento paralelo
- **Estructura de los multiprocesadores de memoria compartida**
- Estructura de los multiprocesadores de memoria distribuida
- Arquitectura vectorial
- Extensiones multimedia
- GPUs

MULTIPROCESADORES DE MEMORIA COMPARTIDA

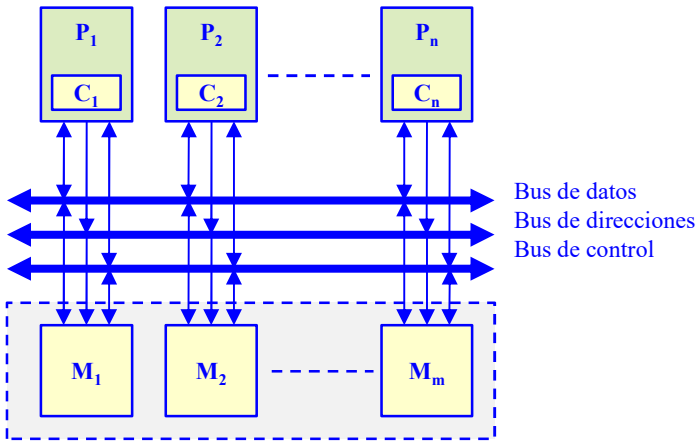
- La mayoría de los multiprocesadores comerciales son del tipo UMA (*Uniform Memory Access*): todos los procesadores tienen igual tiempo de acceso a la memoria compartida.
- En la arquitectura UMA los procesadores se conectan a la memoria a través de un bus, una red multietapa o un conmutador *crossbar* y disponen de su propia memoria caché.
- Los procesadores tipo NUMA (*Non Uniform Memory Access*) presentan tiempos de acceso a la memoria compartida que dependen de la ubicación del elemento de proceso y la memoria.





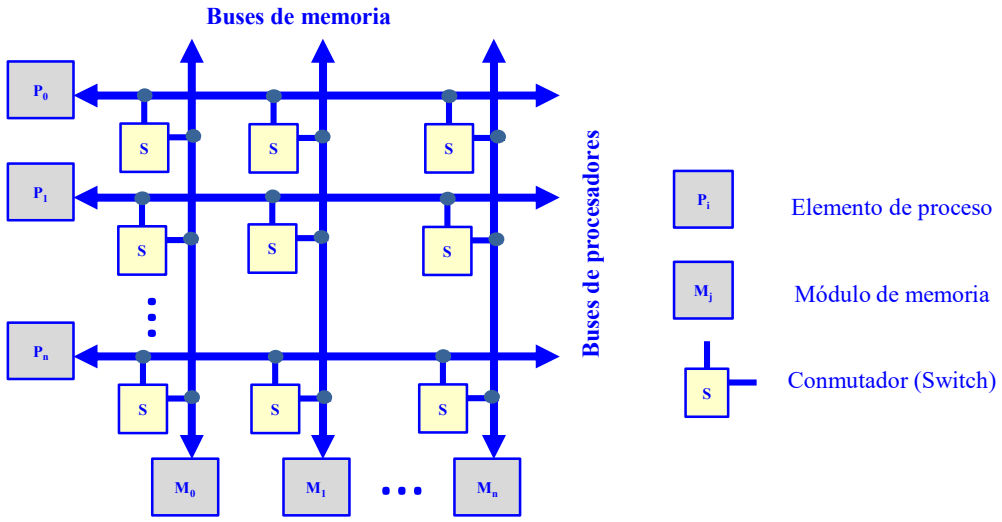
INTERCONEXIÓN DE LOS PROCESADORES CON LA MEMORIA

- **Multiprocesadores de memoria compartida: conexión por bus compartido**
  - Es la organización más común en los computadores personales y servidores
  - El bus consta de líneas de dirección, datos y control para implementar:
    - El protocolo de transferencias de datos con la memoria
    - El arbitraje del acceso al bus cuando más de un procesador compite por utilizarlo.
  - Los procesadores utilizan cachés locales para:
    - Reducir el tiempo medio de acceso a memoria, como en un monoprocesador
    - Disminuir la utilización del bus compartido.



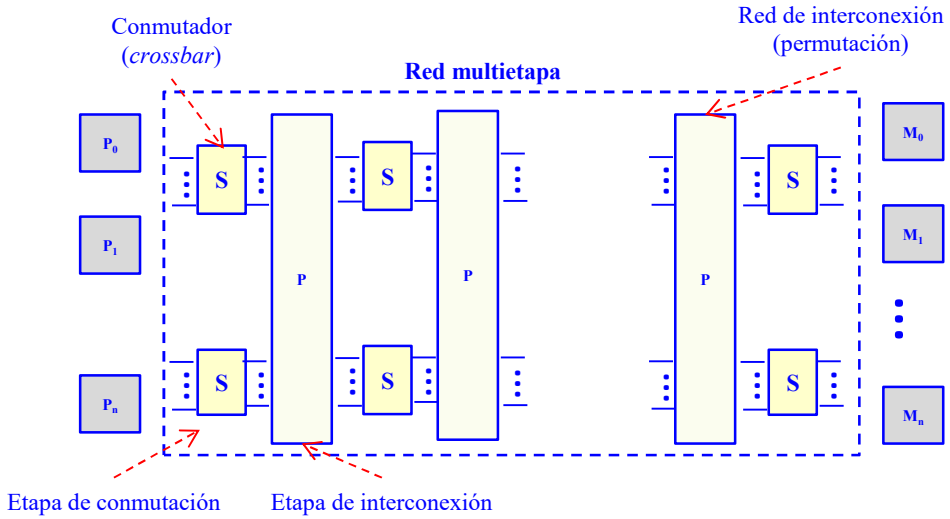
INTERCONEXIÓN DE LOS PROCESADORES CON LA MEMORIA

- **Multiprocesadores de memoria compartida: conexión por conmutadores *crossbar***
  - Cada procesador (*Pi*) y cada módulo de memoria (*Mi*) tienen su propio bus
  - Existe un conmutador (*S*) en los puntos de intersección que permite conectar un bus de memoria con un bus de procesador
  - Para evitar conflictos cuando más de un procesador pretende acceder al mismo módulo de memoria se establece un orden de prioridad
  - Se trata de una red sin bloqueo con una conectividad completa pero de alta complejidad



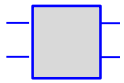
INTERCONEXIÓN DE LOS PROCESADORES CON LA MEMORIA

- **Multiprocesadores de memoria compartida: conexión por red multietapa**
  - Representan una alternativa intermedia de conexión entre el bus y el *crossbar*.
  - Es de menor complejidad que el *crossbar* pero mayor que el bus simple.
  - La conectividad es mayor que la del bus simple pero menor que la del *crossbar*.
  - Se compone de varias etapas alternativas de conmutadores simples y redes de interconexión.
  - En general las redes multietapa responden al siguiente esquema:



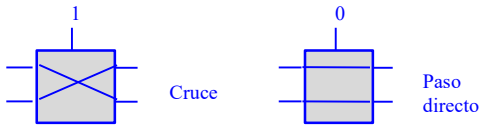
Ejemplo de red multietapa: red Omega (ω)

- Se trata de una red multietapa compuesta de conmutadores básicos de 2 entradas y 2 salidas

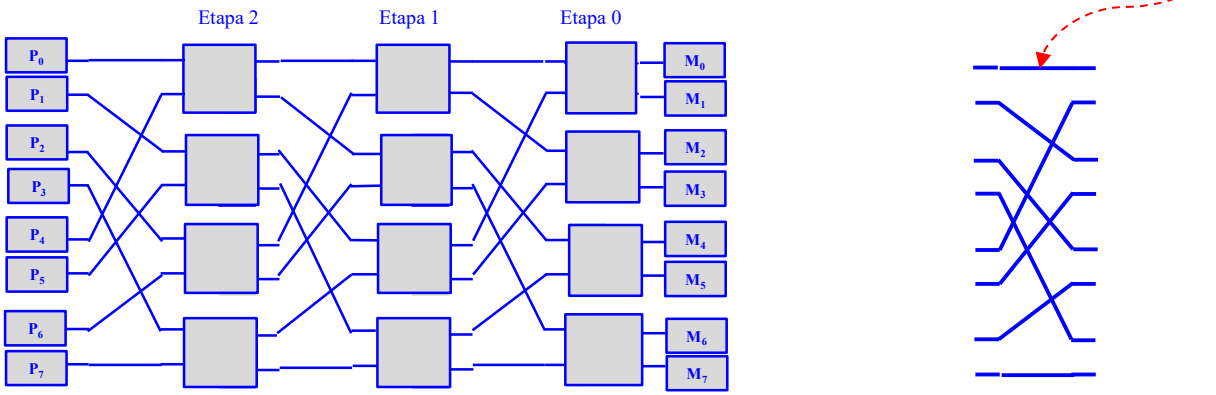


Conmutador básico

- Cada conmutador puede estar en dos estados: *paso directo* y *cruce*

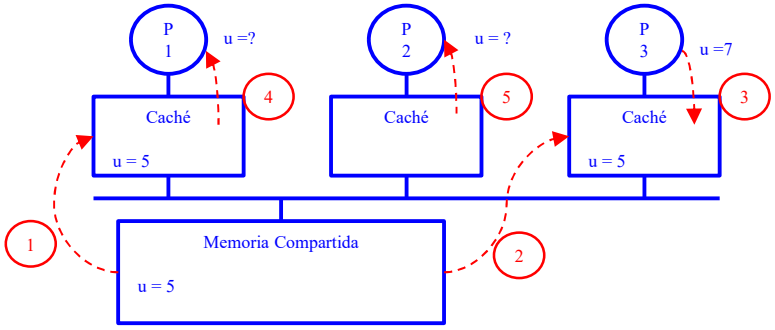


- La interconexión entre etapas se realiza con un patrón fijo denominado *barajadura perfecta.*
- La red de 3 etapas que conecta 8 procesadores con 8 módulos de memoria sería la siguiente:



Problema de coherencia caché

- El problema de la coherencia caché en multiprocesadores surge por las operaciones de escritura.
- **Ejemplo**: secuencia de operaciones de acceso a memoria (1, 2, 3, 4, 5) realizadas por tres procesadores P1, P2 y P3 sobre la posición de memoria *u*:



- Caché con política *write-through*
  - La escritura realizada por el procesador *P3* hará que el valor de *u* en la memoria principal sea 7. Sin embargo, el procesador *P1* leerá el valor de *u* de su caché en vez de leer el valor correcto de la memoria principal.
- Caché con política *writeback*
  - En este caso el procesador *P3* únicamente activará el bit de modificado en el bloque de la caché en donde tiene almacenado *u* y no actualizará la memoria principal.
  - Sólo cuando ese bloque de la caché sea reemplazado se actualizará la memoria principal.
  - No será sólo *P1* el que leerá el valor antiguo, sino que cuando *P2* intente leer *u* y se produzca un fallo en la caché, también leerá el valor antiguo de la memoria principal

CONSISTENCIA DE MEMORIA CACHE

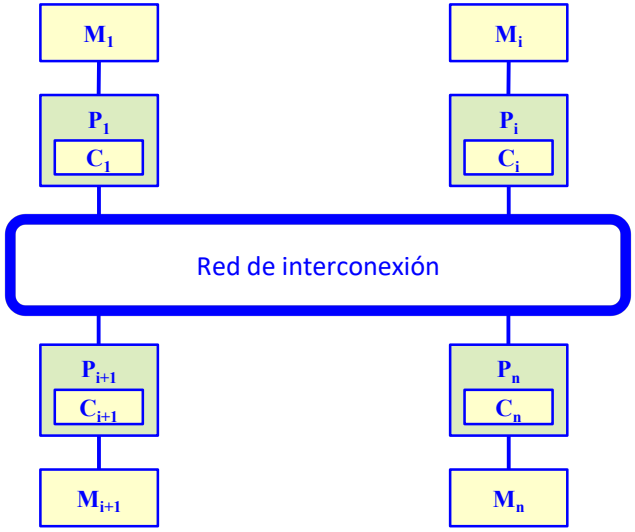
- **Solución de la coherencia caché**
  - Existen dos formas de abordar el problema de la coherencia caché.
    - Software, lo que implica la realización de compiladores que eviten la incoherencia entre cachés de datos compartidos.
    - Hardware que mantengan de forma continua la coherencia en el sistema, siendo además transparente al programador.
  - Podemos distinguir también dos tipos de sistemas multiprocesadores
    - Sistemas basados en un único bus: se utilizan protocolos de sondeo o *snoopy* que analizan el bus para detectar incoherencia. Cada nodo procesador tendrá los bits necesarios para indicar el estado de cada línea de su caché y así realizar las transacciones de coherencia necesarias según lo que ocurra en el bus en cada momento.
    - Sistemas con redes multietapa: protocolo basado en directorio, que consiste en la existencia de un directorio común donde se guardan el estado de validez de las líneas de las cachés, de manera que cualquier nodo puede acceder a este directorio común.

ÍNDICE

- Aritmética en PF
- Introducción al procesamiento paralelo
- Estructura de los multiprocesadores de memoria compartida
- **Estructura de los multiprocesadores de memoria distribuida**
- Arquitectura vectorial
- Extensiones multimedia
- GPUs

Multicomputadores

- Los multiprocesadores de memoria compartida presentan algunas desventajas:
  - Se necesitan técnicas de sincronización para acceder a las variables compartidas
  - La contención en la memoria puede reducir significativamente la velocidad.
  - No son fácilmente escalables a un gran número de procesadores.
- Un **multicomputador** consta de un conjunto de procesadores conectados por una red
- Cada procesador tiene su propia memoria local, incluida la caché, y se comunican por paso de mensajes a través de la red



Propiedades de los multicomputadores

- El número de nodos puede ir desde algunas decenas hasta varios miles (o más).
- La arquitectura de paso de mensajes tiene ventajas sobre la de memoria compartida cuando el número de procesadores es grande.
- El número de canales físicos entre nodos suele oscilar entre cuatro y ocho.
- Esta arquitectura es directamente escalable y presenta un bajo coste para sistemas grandes.
- Un problema se especifica como un conjunto de procesos que se comunican entre sí y que se hacen corresponder sobre la estructura física de procesadores.
- El tamaño de un proceso viene determinado por su *granularidad*:

$$Granularidad = \frac{Tiempo\ de\ cálculo}{Tiempo\ de\ comunicación}$$

- Al reducirse la granularidad, la sobrecarga de comunicación de los procesos aumenta.
- Por ello, la granularidad empleada en este tipo de máquinas suele ser media o gruesa.
- El programa a ejecutar debe de ser intensivo en cálculo, no intensivo en operaciones de entrada/salida o de paso de mensajes

TOPOLOGÍA DE LAS REDES ESTÁTICAS DE INTERCONEXIÓN

Redes estáticas de interconexión en multicomputadores

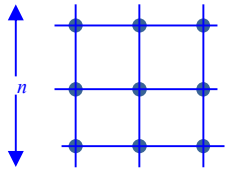
- Los multicomputadores utilizan redes estáticas con enlaces directos entre nodos
- Cuando un nodo recibe un mensaje lo procesa si viene dirigido a dicho nodo
- Si el mensaje no va dirigido al nodo receptor lo reenvía a otro por alguno de sus enlaces de salida siguiendo un protocolo de encaminamiento.
- Las propiedades más significativas de una red estática son las siguientes:
  - **Topología de la red**: determina el patrón de interconexión entre nodos.
  - **Diámetro de la red**: distancia máxima de los caminos más cortos entre dos nodos de la red.
  - **Latencia**: retardo de tiempo en el peor caso para un mensaje transferido a través de la red.
  - **Ancho de banda**: Transferencia máxima de datos en Mbytes/segundo.
  - **Escalabilidad**: posibilidad de expansión modular de la red.
  - **Grado de un nodo**: número de enlaces o canales que inciden en el nodo.
  - **Algoritmo de encaminamiento**: determina el camino que debe seguir un mensaje desde el nodo emisor al nodo receptor.

TOPOLOGÍA DE LAS REDES ESTÁTICAS DE INTERCONEXIÓN

Algunas topología de las redes estáticas

- Mallas

simple



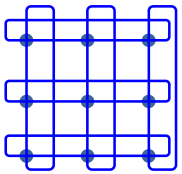
$diámetro = 2 \cdot (n-1)$

$grado = 4$

$nº\ nodos = n^2$

$nº\ enlaces = 2 \cdot n \cdot (n-1)$

toroidal



$diámetro = 2 \cdot \lfloor n/2 \rfloor$

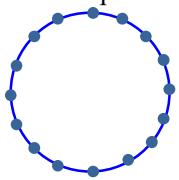
$grado = 4$

$nº\ nodos = n^2$

$nº\ enlaces = 2n^2$

- Anillos

simple



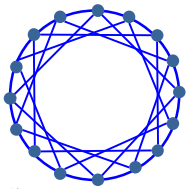
$diámetro = n-1$

$grado = 2$

$nº\ nodos = n$

$nº\ enlaces = n-1$

cordado



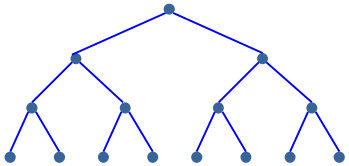
$diámetro = 3$

$grado = 4$

$nº\ nodos = n$

$nº\ enlaces = 32$

- Árbol



$diámetro = 2(k-1)$

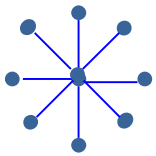
$grado = 4$

$nº\ nodos = 2^k - 1$

$nº\ enlaces = 2^k - 2$

$k = nº\ niveles$

- Estrella



$diámetro = 2$

$grado = n-1$

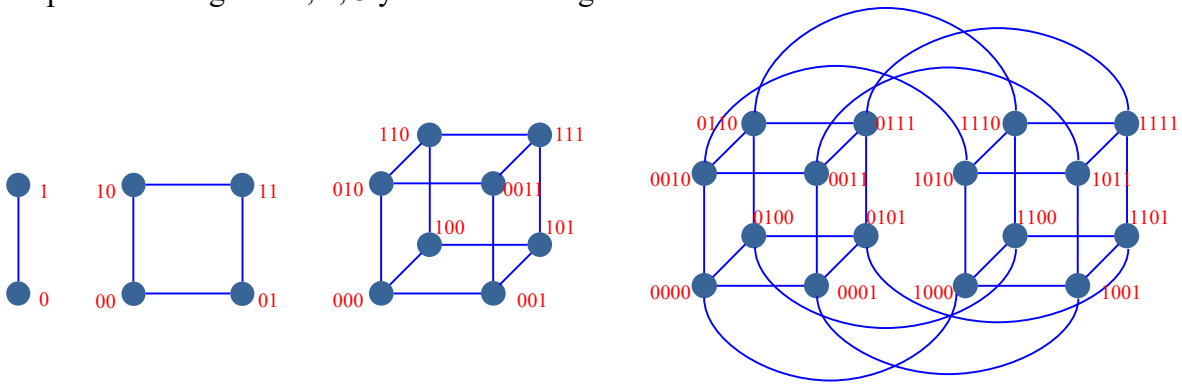
$nº\ nodos = n$

$nº\ enlaces = n-1$

TOPOLOGÍA DE LAS REDES ESTÁTICAS DE INTERCONEXIÓN

Hipercubos

- Un *cubo-n* o hipercubo de dimensión *n* consta de $N=2^n$ nodos extendidos a lo largo de *n* dimensiones.
- El grado vale *n*
- El diámetro también vale *n*
- Existen *n* caminos disjuntos entre cualquier par de nodos
- Cada nodo se etiqueta con un número binario de n bits de tal modo asignados que dos vértices conectados se diferencian en un solo bit.
- Los hipercubos de grado 1, 2, 3 y 4 serían los siguientes:



ÍNDICE

- Aritmética en PF
- Introducción al procesamiento paralelo
- Estructura de los multiprocesadores de memoria compartida
- Estructura de los multiprocesadores de memoria distribuida
- **Arquitectura vectorial**
- Extensiones multimedia
- GPUs

- SIMD (Single Instruction Multiple Data).
  - Una operación (codificada como una sola instrucción de Lenguaje Máquina) se ejecuta sobre un conjunto de datos (en contraposición a SISD).
  - Ejemplo:
    - En lenguaje matemático: $\vec{V3} = \vec{V2} + \vec{V1}$
    - En LAN:
      ```
      for (i = 0; i<N; i++)
          V3(i) = V2(i) + V1(i);
      ```
    - En LM:
      ```
      ADDV  V3, V2, V1
      ```
- Arquitectura SIMD: puede explotar una cantidad importante de paralelismo de datos en:
  - Aplicaciones de ciencia/ingeniería con abundante cálculo matricial (ámbito tradicional)
  - Nuevas aplicaciones: gráficos, visión artificial, comprensión de voz, ...

SIMD

- Eficiencia energética de SIMD: puede ser ventajosa frente a MIMD
  - Sólo es preciso hacer un “fetch” para operar sobre varios datos.
  - Ahorro de energía atractivo en dispositivos portátiles
- En SIMD el programador sigue “pensando” básicamente en un flujo secuencial de instrucciones.
- Soporte arquitectónico para explotar paralelismo SIMD
  - Arquitectura vectorial
  - Extensiones SIMD (extensiones multimedia)
  - Graphics Processor Units (GPUs)

ARQUITECTURA VECTORIAL

- Ideas básicas
  - Leer conjuntos de datos sobre “registros vectoriales”
  - Operar sobre el contenido de estos registros
  - Almacenar los resultados finales en memoria
    - Usar los registros vectoriales para ocultar la latencia de memoria
- Características de las operaciones vectoriales
  - Secuencias de cálculos independientes ➔ Ausencia de riesgos
  - Alto contenido semántico
    - Una instrucción ➔ Muchas operaciones
  - Patrón de accesos a memoria conocido
  - Explotación eficiente de memoria entrelazada
  - Disminución de instrucciones de salto (un bucle completo puede transformarse en una instrucción)
    - Reducción conflictos de control

- En el principio... Seymour Cray – CDC 6600 (1963)
- No es una arquitectura vectorial, pero...
  - Muy segmentada, con palabra de 60 bits
  - MP de 128 Kpalabras con 32 bancos
  - 10 FUs (paralelas, no segmentadas)
    - PF: sumador, 2 multiplicadores, divisor
  - Control cableado
  - Planificación dinámica de instrucciones (scoreboard)
  - 10 procesadores de E/S
  - Clock 10 MHz
    - Muy rápido para la época
    - Suma PF en 4 ciclos
  - >400,000 transistores, 750 sq. ft. (~70 m^2), 5 tons, 150 KW, refrigeración freon
  - Máquina más rápida del mundo durante 5 años (hasta el 7600)
  - Vendidas >100 (a $7-10M c.u.)

ARQUITECTURA VECTORIAL

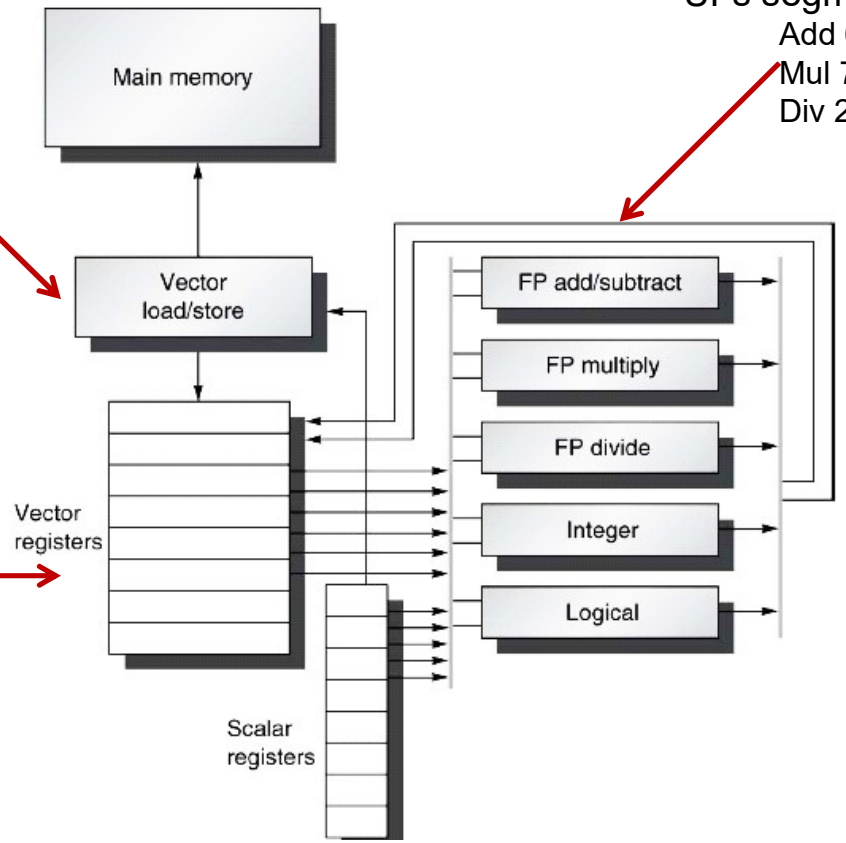
El Cray-1 (1976)

- Unidad escalar
  - Arquitectura Load/Store
- Extensión Vectorial
  - Registros Vectoriales
  - Instrucciones Vectoriales
- Implementación
  - Control cableado
  - UF muy segmentadas
  - Memoria entrelazada
  - Sin cache de datos
  - Sin memoria virtual

Estructura de un procesador vectorial: RISC-V (RV64V)

Funcionalmente equivalente a un pipe: un dato por ciclo, después de latencia inicial (12 ciclos).

UFs segmentadas:
Add 6 etapas
Mul 7 etapas
Div 20 etapas



8 registros vectoriales
1 reg = 64 elementos
1 elemento = 64 bits
16 read, 8 write ports

ARQUITECTURA VECTORIAL: REPERTORIO RISC-V

- Operaciones vectoriales aritméticas sobre registros
- Instrucciones especiales de carga/almacenamiento de vectores (vld,vst)
- Modos de direccionamiento especiales para vectores no contiguos. Ejemplos
  - vlds: Load Vector With Stride. Carga elementos equiespaciados a una cierta distancia
  - vldx: Load Vector using Index. El contenido de un registro vectorial indica las posiciones de los elementos a cargar.
- Registros especiales de longitud vectorial (vl) y predicado p_i
  - Registro vl: Indica la longitud de los vectores a procesar (≤ 64)
  - Registros p_i: Registros usados cuando los bucles incluyen sentencias if
    - Ejecución selectiva de operaciones sobre componentes

ARQUITECTURA VECTORIAL: REPERTORIO RISC-V

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

ARQUITECTURA VECTORIAL: REPERTORIO RISC-V

- El repertorio RISC-V vectorial soporta diferente tipos de datos:
  - Enteros: 8, 16, 32 y 64 bits
  - Punto flotante: 16, 32, y 64 bits
- Para no tener repetidas las instrucciones para los distintos tamaños el RV64V usa *dynamic register typing*
  - Permite definir el tamaño de los registros dinámicamente en función de las necesidades
  - Permite descapacitar los registros no usados, por lo que los cambios de contexto solo afectan a los registros sí capacitados.

```
vsetdcfg 4*FP64          #Capacita 4 vregs DP FP
vdisable                 # descapacita los vregs
```

CÓDIGO ESCALAR VS. CÓDIGO VECTORIAL

- Y = a*X + Y (AXPY, vectores de 32 elementos. Dirección de comienzo de X en x5 y dirección de comienzo de Y en x6)
  - Versión escalar

```
        fld     f0, a           # carga el escalar a
        addi    x28, x5, #256   # última dirección del vector
Loop:   fld     f1, 0(x5)       # carga X[i]
        fmul.d  f1, f1, f0      # A * X[i]
        fld     f2, 0(x6)       # carga Y[i]
        fadd.f  f2, f2, f1      # A * X[i] + Y[i]
        fsd     f2, 0(x6)       # almacena Y[i]
        addi    x5, x5, #8      # incrementa índice X
        addi    x6, x6, #8      # incrementa índice Y
        bne     x28, x5, Loop   # fin de bucle?
```

  - Versión vectorial

```
vsetdcfg    4*FP64      #Capacita 4 vregs DP FP
fld         f0, a       # carga escalar a
vld         v0, x5      # carga vector X
vmul        v1, v0, f0  # Multiplicación Vector-escalar
vld         v2, x6      # carga vector Y
vadd        v3, v1, v2  # Suma vectorial
vst         v3, x6      # almacena Y
vdisable                # descapacita los vregs
```

  - Instrucciones ejecutadas: 8*32+2= 258 vs 8

TIEMPO DE EJECUCIÓN

- Depende básicamente de tres factores:
  - Longitud de los vectores operandos
  - Riesgos estructurales: las UF necesarias están ocupadas, no hay puertos del BR disponibles
  - Dependencias de datos
- Velocidad de procesamiento
  - Las UFs del RISC-V consumen (y producen) un elemento por ciclo de reloj
  - El tiempo de ejecución de una operación vectorial es aproximadamente igual a la longitud del vector
- Convoy
  - Se denomina así a un conjunto de (una o varias) instrucciones vectoriales que potencialmente pueden ejecutarse juntas (ausencia de riesgos estructurales). Pueden tener riesgos LDE.

TIEMPO DE EJECUCIÓN

- Convoyes: ejemplo

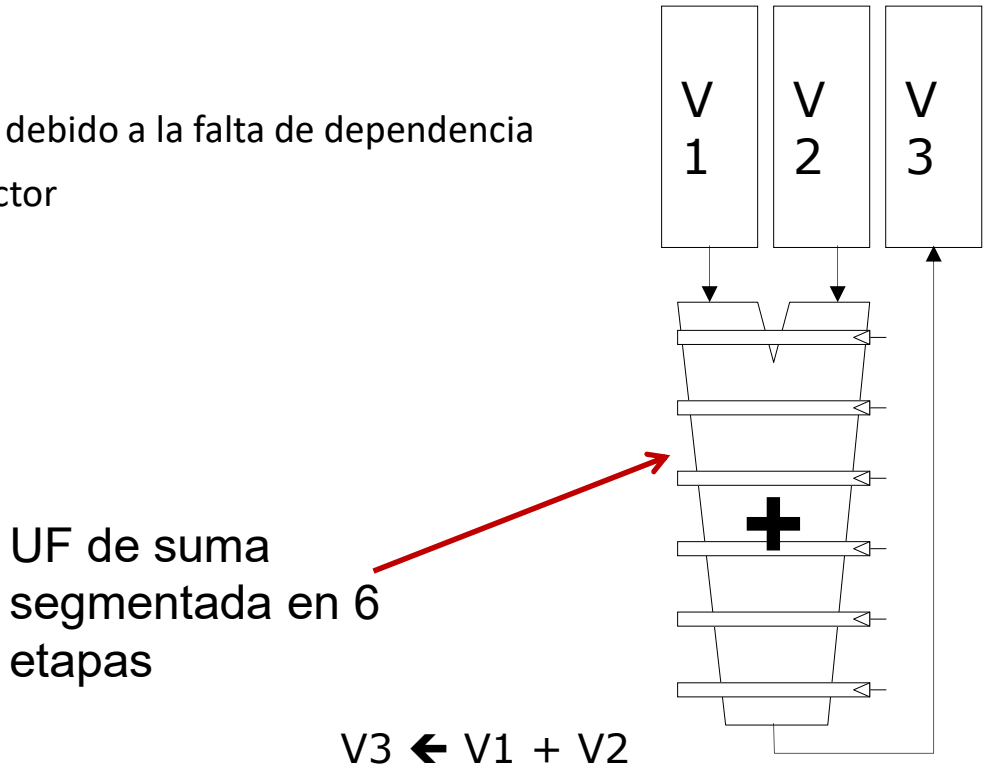
```
1: vld      v0, x5        # carga vector X
2: vmul     v1, v0, f0    # Multiplicación Vector-escalar
3: vld      v2, x6        # carga vector Y
4: vadd     v3, v1, v2    # Suma vectorial
5: vst      v3, x6        # Almacena vector Y
```

  - Conflictos:
    - 1 y 2 no tienen conflictos estructurales
    - 3 tiene conflicto estructural con 1 (una sola unidad de Load/Store)
    - 4 no tiene conflictos estructurales con 3
    - 5 tiene conflicto estructural con 3
  - Convoyes resultantes
    - 1. Formado por vld y vmul
    - 2. Formado por vld y vmul
    - 3. Formado por vst

EJECUCIÓN DE OPERACIONES ARITMÉTICAS

- Uso de un pipe profundo para la ejecución de las operaciones (reduce el ciclo de reloj)
- Alta latencia
  - No demasiado relevante debido a la falta de dependencia

entre los cálculos sobre un vector



EJECUCIÓN DE OPERACIONES ARITMÉTICAS

- Operaciones independientes

vmul V1, V2, V3

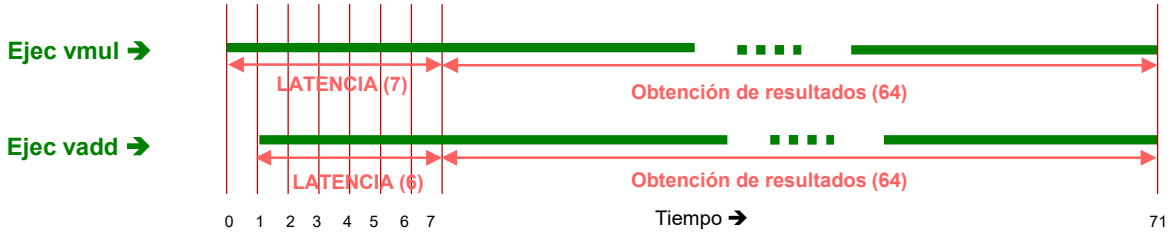
vadd V4, V5, V6

(1 convoy: la UF de * y la de + pueden actuar a la vez)

  - Comportamiento temporal (1 paso)
    - Recordar: MUL 7 ciclos, ADD 6 ciclos

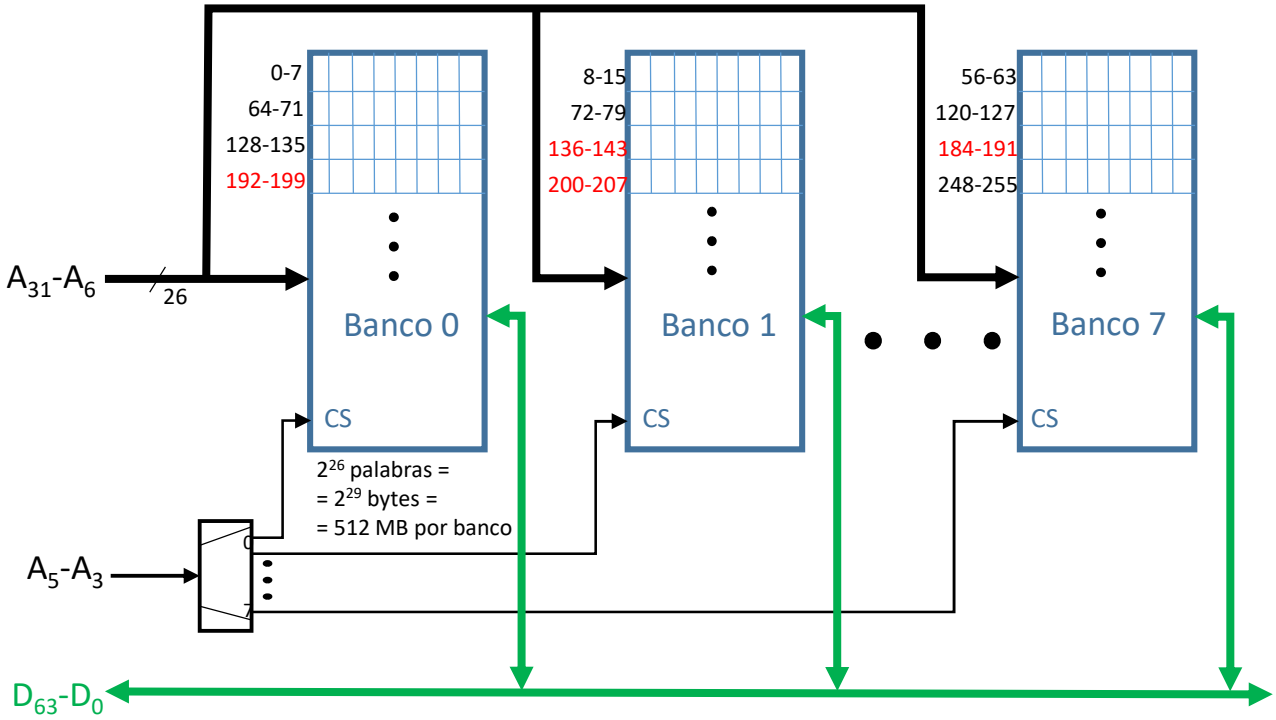
Operación	Inicio	Fin
vmul	0	7+64 = 71
vadd	1	1+6+64 = 71

  - En ausencia de conflictos lanza una instrucción por ciclo



Memoria con entrelazamiento de orden bajo (a nivel de palabra) con 8 bancos

- Longitud de palabra: 8 bytes
- Palabras consecutivas están en bancos consecutivos.
- Cada 8 palabras se vuelve a usar el mismo banco.

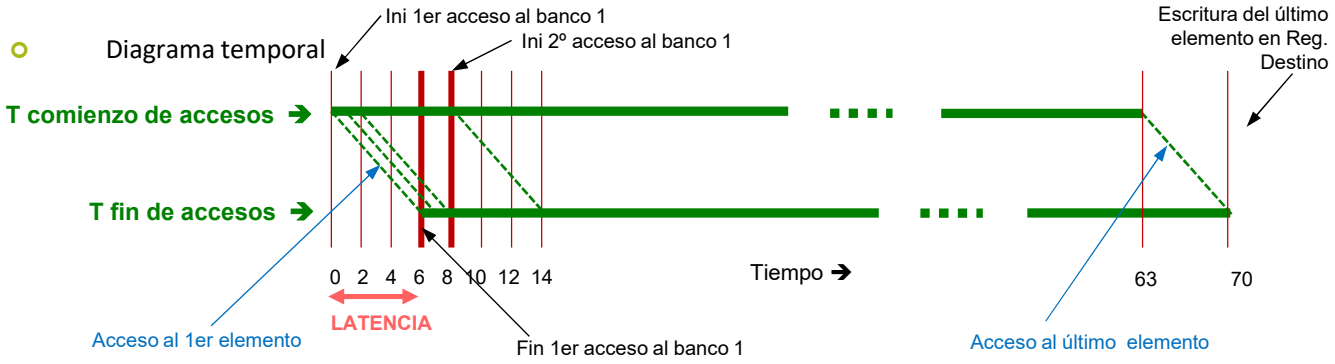


EJECUCIÓN DE ACCESOS A MEMORIA

- Memoria entrelazada (equivalencia funcional con un pipe)
  - Ejemplo: Supongamos que tenemos 8 bancos, Tiempo de acceso a memoria: 6 ciclos.
    - Carga de un vector de 64 componentes que comienza en la dirección 136 (cada componente 8 bytes)
    - Formato de dirección: ...xxxx **yyy** 000 (siendo yyy = nº de banco)

Dir	136	144	152	160	168	176	184	192	200	208	...
Banco	1	2	3	4	5	6	7	0	1	2	...
Tini	0	1	2	3	4	5	6	7	8	9	...
Tfin	6	7	8	9	10	11	12	13	14	15	...

    - Diagrama temporal



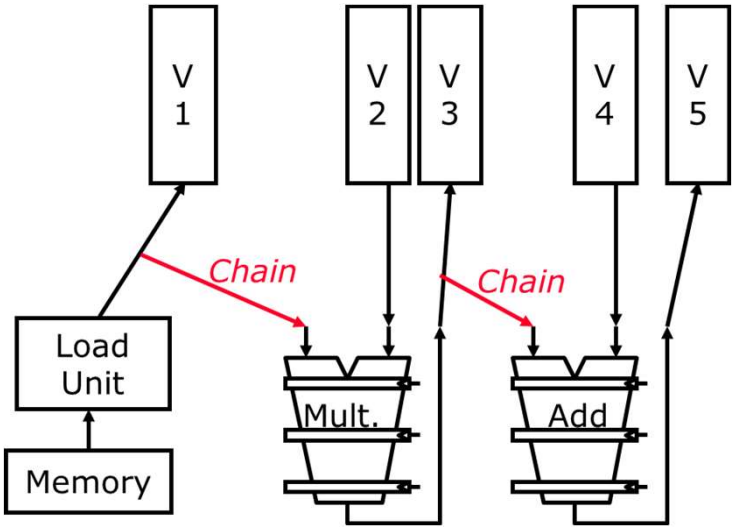
ENCADENAMIENTO

- Operaciones dependientes: Tratamiento de dependencias RAW

- Problema

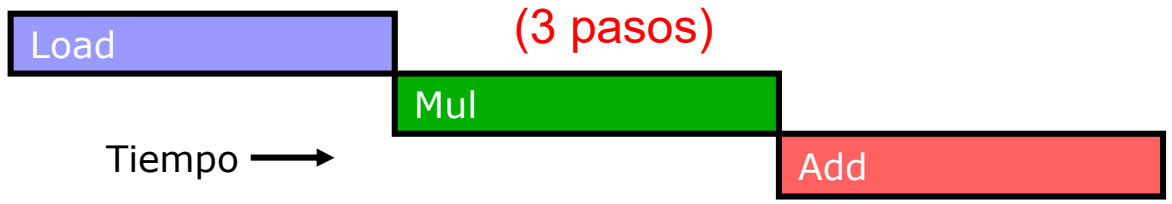
```
vld     v1, x2
vmul    v3, v1, v2
vadd    v5, v3, v4
```

- Solución:
  - Cray-1. Extensión del concepto de anticipación de operandos ⇒ Encadenamiento de operaciones (chaining)
  - 3 operaciones, pero 1 paso
  - En proc modernos: “encadenamiento flexible”

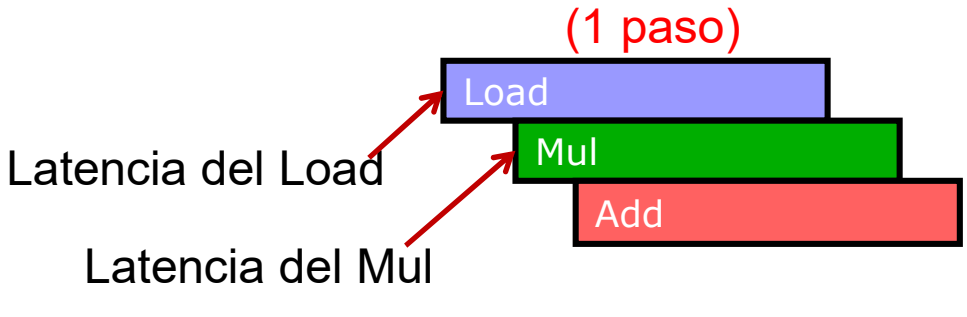


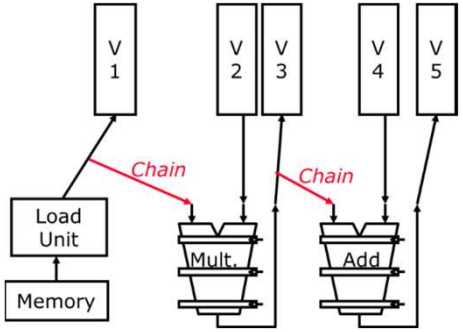
ENCADENAMIENTO

- Sin encadenamiento: esperar hasta que se haya calculado el último elemento de la operación anterior



- Con encadenamiento: Una instrucción puede comenzar cuando está disponible el primer elemento de la operación de la que depende





OPERACIONES CONDICIONALES: REGISTRO DE MÁSCARA

- Ejecutar

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

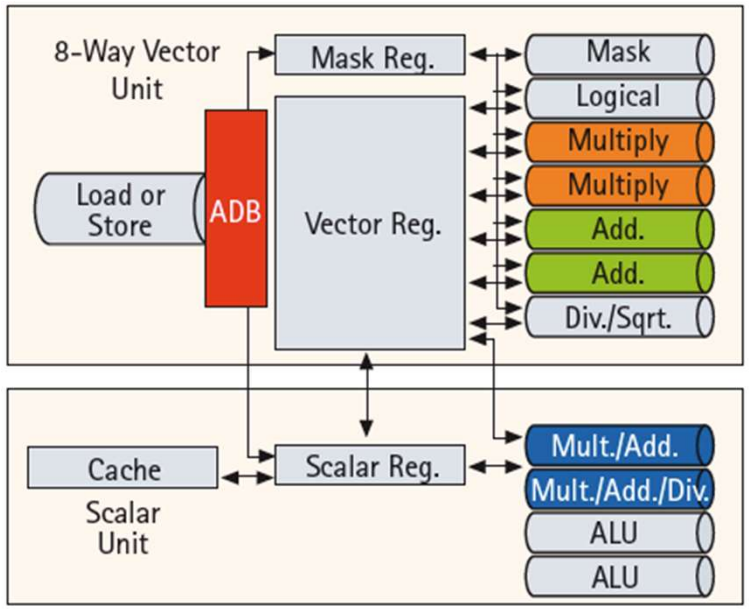
- El registro de máscara vectorial permite omitir las operaciones sobre los elementos que no cumplen la condición

```
vsetdcfg        2PFP64          # Capacita 2 registros vectoriales en PF de 64 bits
vsetpcfgi       1               # Capacita un registro de predicados
vld             v0, x5          # Carga el vector X en v0
vld             v1, x6          # Carga el vector Y en v2
fmv.d.x         f0, x0          # Pone cero en PF en F0
vpne            p0, v0, f0      # Pone p0(i) a 1 si v0(i) != 0
vsub            v0, v0, v1      # Resta con el vector máscara
vst             v0, x5          # Almacena el resultado en X
vdisable                        # Descapacita los registros vectoriales
vpdisaable                      # Descapacita el registro de predicados
```

- Obviamente, el rendimiento se reduce
  - Se consume el mismo tiempo
  - Se ejecutan menos operaciones útiles

EJEMPLO DE COMPUTADOR VECTORIAL: NEC SX-9

Esquema de 1 CPU



Introducido en 2008

- Tecnología 65nm CMOS
- Unidad Vectorial (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elementos/VReg)
  - UFs de 64-bits: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 1 load or store unit (8 x 8-byte accesos/ciclo)
- Unidad escalar (1.6 GHz)
  - Superescalar 4 vías O-o-O
  - 64KB I-cache, 64KB data cache

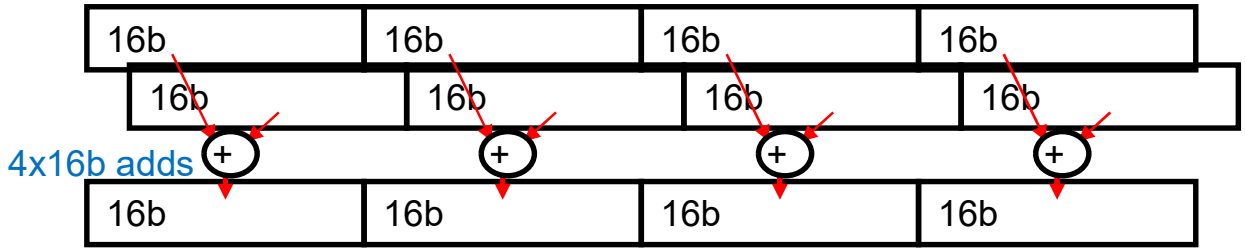
- AB Memoria: 256GB/s por CPU
  - Hasta 16 CPUs y hasta 1TB de DRAM forman un **nodo** con mem compartida
  - AB total: 4TB/s a la DRAM compartida
- Hasta 512 nodos conectados mediante enlaces de 128 GB/s (Paso de mensajes entre nodos)

ÍNDICE

- Aritmética en PF
- Introducción al procesamiento paralelo
- Estructura de los multiprocesadores de memoria compartida
- Estructura de los multiprocesadores de memoria distribuida
- Arquitectura vectorial
- **Extensiones multimedia**
- GPUs

EXTENSIONES SIMD

- También conocidas como “extensiones multimedia”
- Observación: las aplicaciones multimedia suelen operar sobre datos de menor anchura que las UFs y los registros disponibles
- Idea: Realizar varias operaciones a la vez
  - Por ejemplo: desconectar la cadena de propagación de carries
  - Una sola instrucción de suma opera sobre varios elementos almacenados en un registro ➔ equivale a una operación vectorial, aunque con un vector de pocos elementos



EXTENSIONES SIMD

- Limitaciones, comparado con operaciones vectoriales:
  - La longitud de los vectores se codifica en el Cod_op
  - No existe direccionamiento sofisticado (stride, gather,...)
  - No hay registro de máscara
- Ventajas, comparado con operaciones vectoriales:
  - Cuesta poco añadirlas al repertorio estándar y son fáciles de implementar
  - Tienen poca información de estado comparadas con las arquitecturas vectoriales por lo que los cambio de contexto son muy rápidos
  - Los procesadores vectoriales necesitan mucho ancho de banda con memoria que no siempre está disponible
  - SIMD no tiene problemas de memoria virtual. En procesadores vectoriales una instrucción de accedo a memoria puede generar un fallo de página a mitad del vector

EXTENSIONES SIMD

- Implementaciones:
  - Intel MMX (1996)
    - Ocho operaciones enteras de 8-bits o cuatro operaciones enteras de 16-bit
  - Streaming SIMD Extensions (SSE) (1999-2007)
    - Ocho operaciones enteras de 16-bit
    - Cuatro operaciones enteras/FP de 32-bit o dos operaciones enteras/FP de 64-bit
  - Advanced Vector Extensions (AVX)(2010)
    - Cuatro operaciones enteras/FP de 64-bit
  - Los operandos deben ser consecutivos y en posiciones de memoria alineadas

- Ejemplo: Instrucciones AVX para la arquitectura x86

4 operandos de 64 bits

AVX Instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ..
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

Como la longitud de los operandos va indicada en el Cod_op, puede dar la impresión de que el nº de instr de las “extensiones multimedia” es mayor de lo que en realidad es.

EJ. CÓDIGO CON EXTENSIONES SIMD EN MIPS

- Sup: Añadimos instrucciones multimedia SIMD de 256 bits al MIPS (".4D" ➔ operaciones sobre 4 operandos de 64 bits a la vez)
- Código para DAXPY:

```
        L.D F0,a                ;load scalar a
        MOV F1, F0              ;copy a into F1 for SIMD MUL
        MOV F2, F0              ;copy a into F2 for SIMD MUL
        MOV F3, F0              ;copy a into F3 for SIMD MUL
        DADDIU R4,Rx,#512       ;last address to load
Loop:   L.4D F4,0(Rx)           ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D F4,F4,F0         ;a*X[i],a*X[i+1],a*X[i+2],a*X[i+3]
        L.4D F8,0(Ry)           ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D F8,F8,F4         ;a*X[i]+Y[i], ..., a*X[i+3]+Y[i+3]
        S.4D F8,0(Ry)           ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU Rx,Rx,#32        ;increment index to X
        DADDIU Ry,Ry,#32        ;increment index to Y
        DSUBU R20,R4,Rx         ;compute bound
        BNEZ R20,Loop           ;check if done
```

ÍNDICE

- Aritmética en PF
- Introducción al procesamiento paralelo
- Estructura de los multiprocesadores de memoria compartida
- Estructura de los multiprocesadores de memoria distribuida
- Arquitectura vectorial
- Extensiones multimedia
- **GPUs**

UNIDADES PARA PROCESAMIENTO GRÁFICO (GPUS)

- Las GPUs son económicas, accesibles y contienen una gran cantidad de elementos de cómputo.
- Se han concebido con el objeto de realizar los procesamientos característicos de las aplicaciones gráficas
- ¿Cómo poder utilizar la gran potencia de los procesadores gráficos en un espectro de aplicaciones más amplio?
- Idea básica
  - Modelo de ejecución heterogéneo (CPU+GPU)
  - Desarrollar un lenguaje de programación tipo C que permita programar la GPU
  - Unificar todo el paralelismo de la GPU bajo la abstracción denominada “CUDA Thread”
  - Modelo de Programación: “Single Instruction (SIMD) Multiple Thread”

ARQUITECTURA DE NVIDIA GPU

- GPU NVIDIA
  - Multiprocesador compuesto por un conjunto de procesadores SIMD MT (Multiple Thread)
- NVIDIA vs procesadores vectoriales
  - Similitudes
    - Funciona bien en problemas con paralelismo de datos
    - Transferencias con memoria tipo dispersar/reunir (scatter/gather)
    - Registros de máscara
    - Existencia de grandes ficheros de registros
  - Diferencias
    - No hay un procesador escalar
    - Utilización de multithreading para ocultar la latencia de memoria
    - Existencia de gran cantidad de UFs
      - Contrasta con la reducida cantidad de UFs muy segmentadas, que es típica de los procesadores vectoriales

CUDA(Compute Unified Device Architecture)

- CUDA produce código C/C++ para host y dialecto de C y C++ para la GPU

  - Idea básica: crear un thread (hilo) separado para cada elemento de los vectores a procesar
    - Objetivo: generar un gran número de hilos de cómputo independientes

  - Los threads se agrupan en bloques de threads
    - El número de threads por bloque puede definirlo el programador
    - Cada bloque es ejecutado por un procesador SIMD MT de la GPU
    - Varios bloques pueden ejecutarse en paralelo sobre varios procesadores

  - El conjunto de bloques que implementan un cálculo vectorial sobre la GPU se denomina Grid (malla). La ejecución del cálculo se produce con una llamada similar a una función en C:

    - **nombre_función** <<<dimGrid,dimBlock>>> **(... lista de parámetros ...)**
      - dimGrid: nº de bloques en el Grid
      - dimBlock: nº de threads por bloque

CUDA(Compute Unified Device Architecture)

- CUDA vs C. Ejemplo DAXPY
  - Versión C

```
// Invocar DAXPY
daxpy(n, 2.0, x, y);

// DAXPY en C (bucle escalar: una iteración por elemento)
void daxpy(int n, double a, double *x, double *y)
{
   for (int i = 0; i < n; i++)
                  y[i] = a*x[i] + y[i];
}
```

No hay dependencias entre iteraciones

CUDA(Compute Unified Device Architecture)

- CUDA vs C. Ejemplo DAXPY
  - Versión CUDA

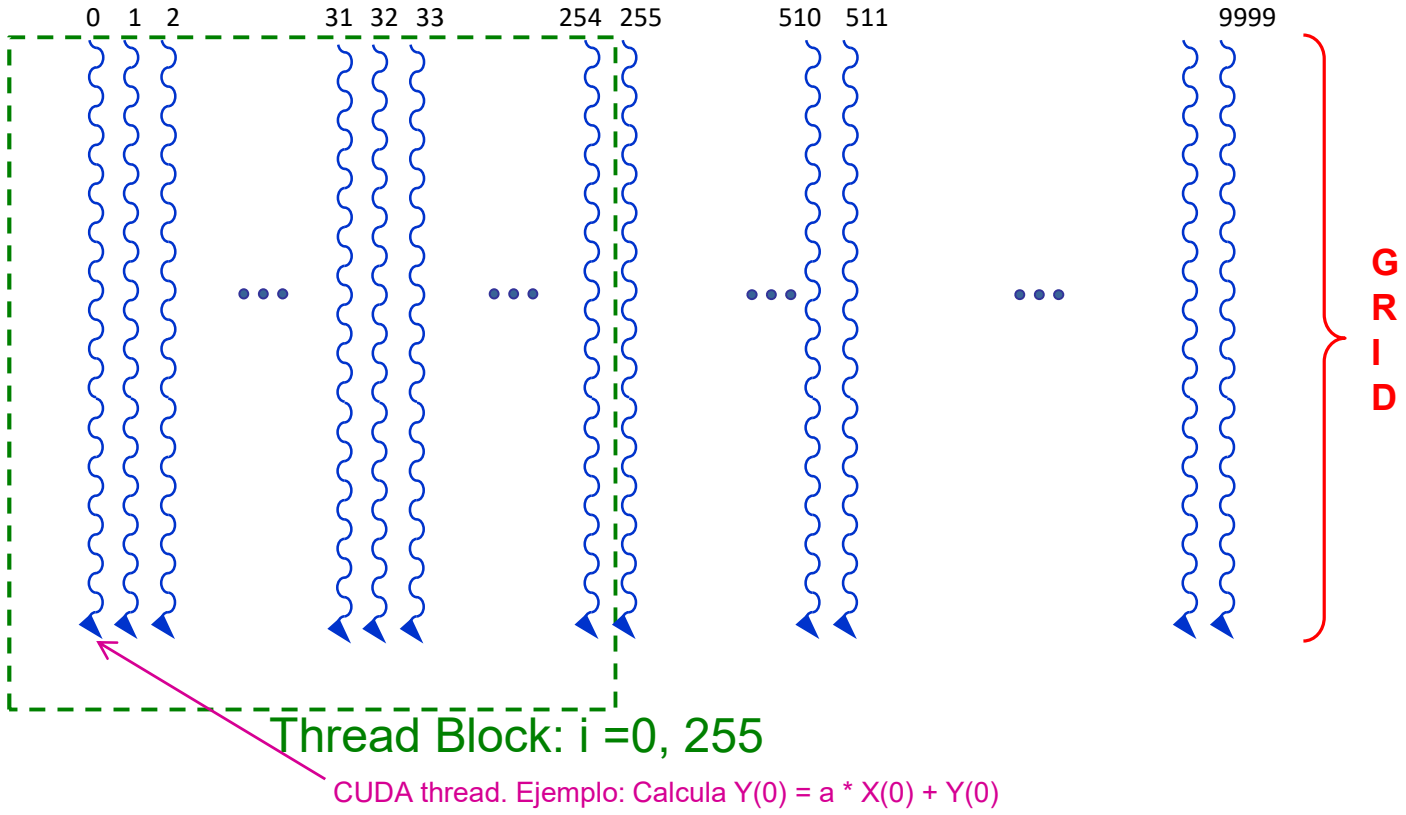
```
// Invocar DAXPY con 256 threads por Bloque (dimBlock)
__host__                                  /* código para la CPU */
int nblocks = (n+ 255) / 256;             /*  cálculo del nº total de bloques en el Grid (dimGrid) */
              daxpy <<<nblocks, 256>>> (n, 2.0, x, y);


// DAXPY en CUDA (representa el cálculo ejecutado para un elemento)
__device__                                /* código para los procesadores de la GPU */
void daxpy(int n, double a, double *x, double *y)
              {
              int i = blockIdx.x*blockDim.x + threadIdx.x;
// Si el nº elemento obtenido es mayor que el tamaño del vector, ignorar operación
              if (i < n) y[i] = a*x[i] + y[i];
              }
```

¿Qué thread soy? Calcular i = nº elemento del vector a procesar (= nº de thread), siendo nº elemento = (nº de bloque x tamaño de bloque) + (nº de thread dentro del bloque)

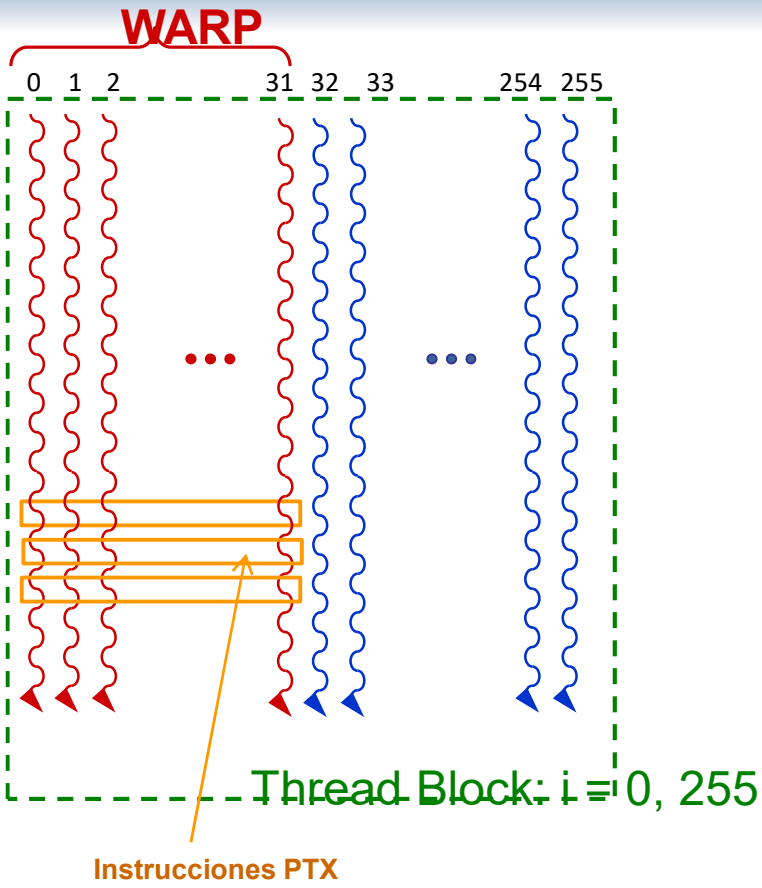
CUDA(Compute Unified Device Architecture)

Ejemplo: vectores de 10,000 componentes



THREAD BLOCKS E INSTRUCCIONES PTX

- Cada Thread Block se ejecuta en un procesador SIMD MT de la GPU.
- Varios procesadores SIMD MT de la GPU pueden procesar diferentes Thread Blocks en paralelo.
- Instrucción PTX(Parallel Thread Execution): Ejecuta un mismo cálculo sobre varios (e.g. 32) datos (Instr SIMD).
  - Resultados afectados por registro de máscara.
- WARP: Secuencia (thread) de instr PTX. El procesador ejecuta los WARP de un Thread Block en modo Multiple Thread.
  - En el ejemplo hay 256/32 = 8 WARPs .
  - Cambios de thread ocultan latencias de acceso a memoria



CÓDIGO GENERADO POR COMPILADORES DE NVIDIA

- Instrucciones PTX (Parallel Thread Execution)
  - Abstracción del repertorio de instrucciones hw
  - Formato: opcode.type dest, src1, scr2, src3
  - Instrucción que ejecuta una operación elemental sobre múltiples datos (SIMD) utilizando todas la vías del procesador
  - Ejemplo: un conjunto de instrucciones PTX representativas

Instruction	Example	Meaning	Comments
arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
add.type	add.f32 d, a, b	d = a + b;	
sub.type	sub.f32 d, a, b	d = a – b;	
mul.type	mul.f32 d, a, b	d = a * b;	
mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
mov.type	mov.b32 d, a	d = a;	move
selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space

CÓDIGO GENERADO POR COMPILADORES DE NVIDIA

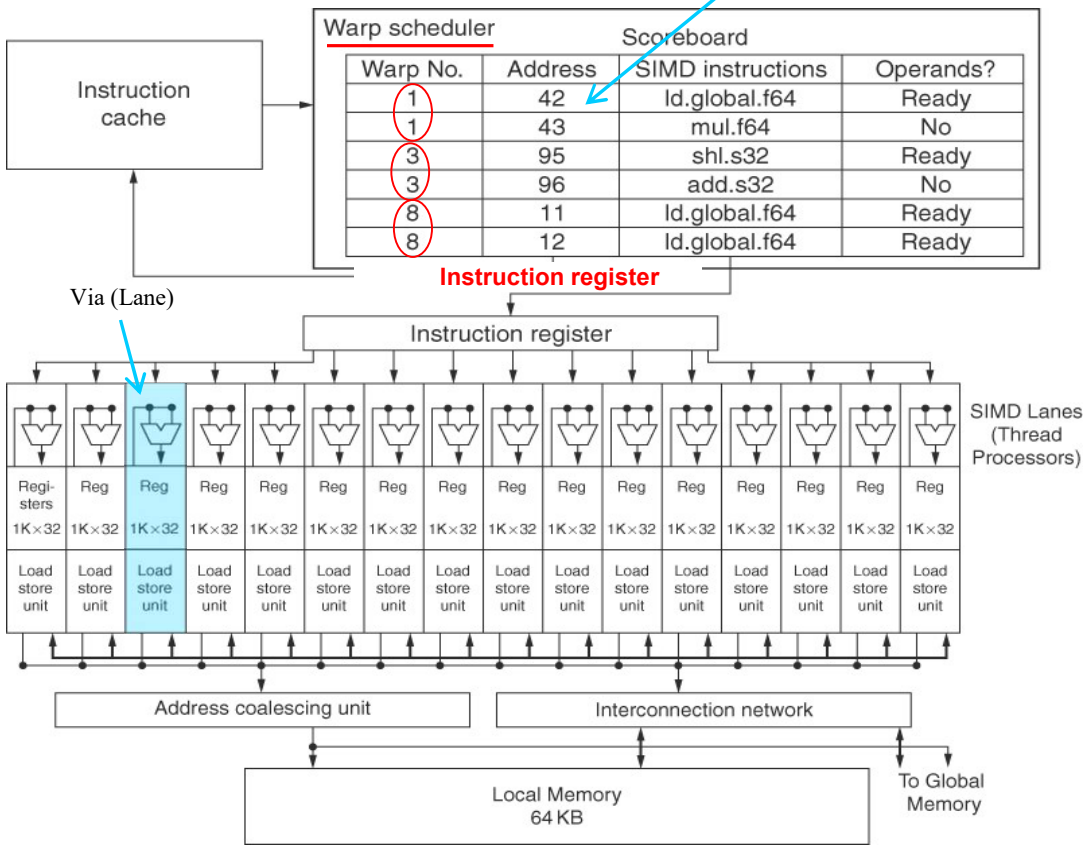
- Ejemplo: secuencia de instrucciones PTX para una iteración del bucle DAXPY
  - Usa registros virtuales: Ri (32 bits), RDi (64 bits)
  - Asigna registros físicos en el momento de la carga del programa

```
shl.u32     R8, blockIdx, 8      ; shift lesft. Thread Block ID * Block size (256 or 2^8)
add.u32     R8, R8, threadIdx    ; R8 = i = my CUDA Thread ID
shl.u32     R8, R8, 3            ; byte offset
ld.global.f64   RD0, [X+R8]      ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]      ; RD2 = Y[i]
mul.f64     RD0, RD0, RD4        ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64     RD0, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64   [Y+R8], RD0      ; Y[i] = sum (X[i]*a + Y[i])
```

Ojo! Recordar que cada instrucción PTX procesa 32 elementos

Procesador SIMD MT

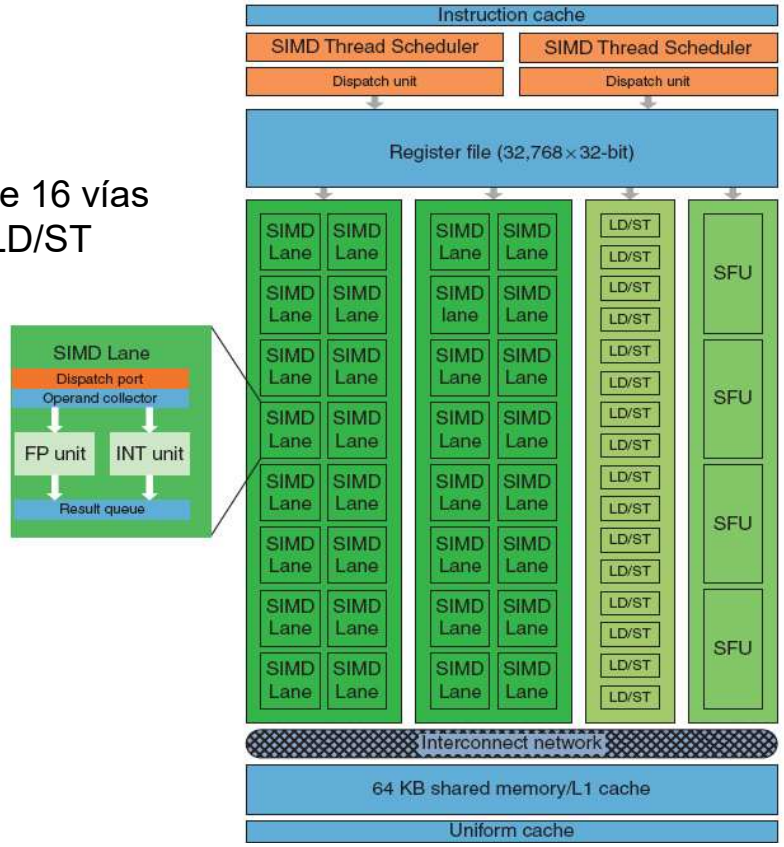
Cada Warp (thread de instrucciones SIMD) tiene su PC



- Todas las vías ejecutan la misma instrucción
-Según la máscara, unas guardan el resultado y otras no

- Fermi: Esquema de un procesador SIMD MT

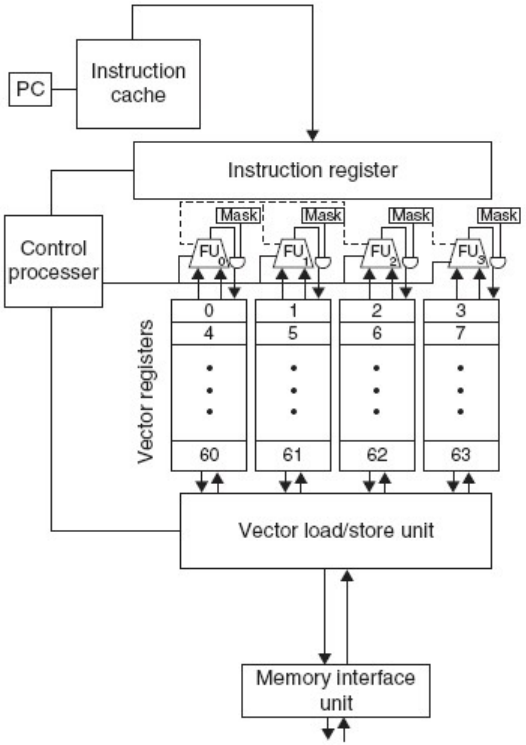
Dos conjuntos de 16 vías
16 unidades de LD/ST



Special FU:
Calcula √, sin, 1/x…

Comparación Procesador Vectorial - GPU

Procesador vectorial con cuatro vías



Procesador SIMD MT (4 PCs) con cuatro vías