# Clase 1:

Un **programa** es:

* Una *secuencia* de acciones almacenadas en el sistema de cómputo.
* Donde cada acción es una instrucción (de máquina) y es convertida en señales.
* Una instrucción de máquina es un código binario compuesto por una cantidad determinada de bits.
* En general las instrucciones más complejas requieren de múltiples pasos o acciones. Las más sencillas se resuelven en pocos pasos.
* Un ejemplo de instrucción “relativamente compleja” es la que busca 2 operandos y produce un resultado. Para esta instrucción la CPU necesita definir:
* 1🡪 *Tipo de operación* (en base al código de operación)
* 2🡪 *Lugar del 1er operando* (dónde está)
* 3🡪 *Lugar del 2do operando* (dónde está)
* 4🡪 *Lugar del resultado* (dónde guardarlo)
* 5🡪 *Dónde está la próxima instrucción*
* Diferentes señales de control se necesitan para cada operación.
* La Unidad de Control (**UC** o CU en inglés) obtiene toda la información necesaria para resolver (ejecutar) la acción definida en la instrucción

**Modelo de Von Neumann con 5 unidades funcionales:**

* Unidad de control (UC)
* Unidad aritmético-lógica (ALU)
* Unidad de memoria
* Unidad de entrada (de datos)
* Unidad de salida (de datos)

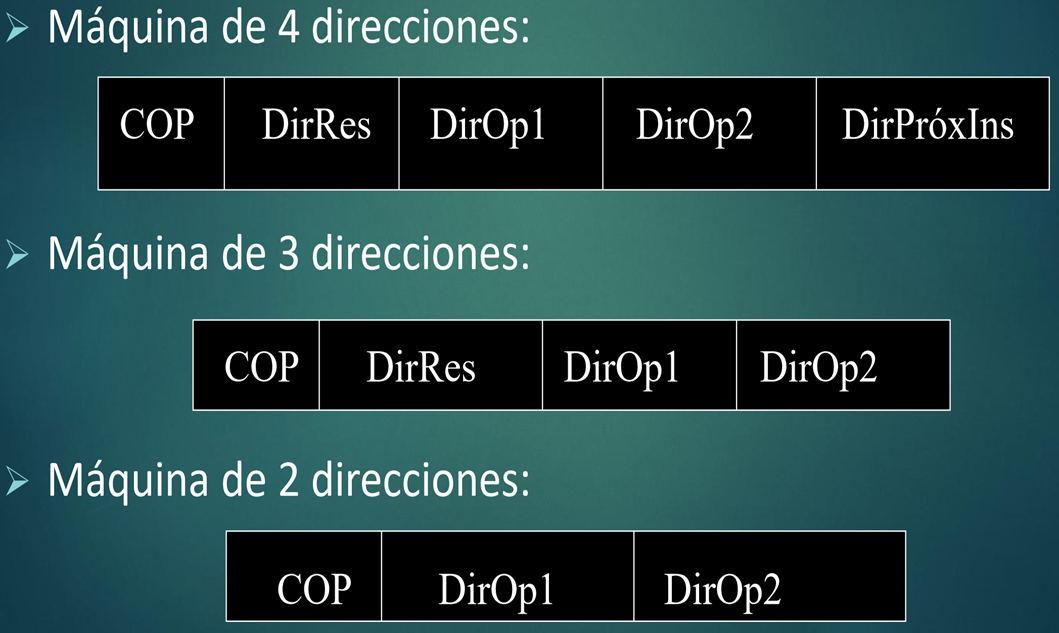
**Modelo de Von Neumann de 3 subsistemas (modelo de Harvard):**

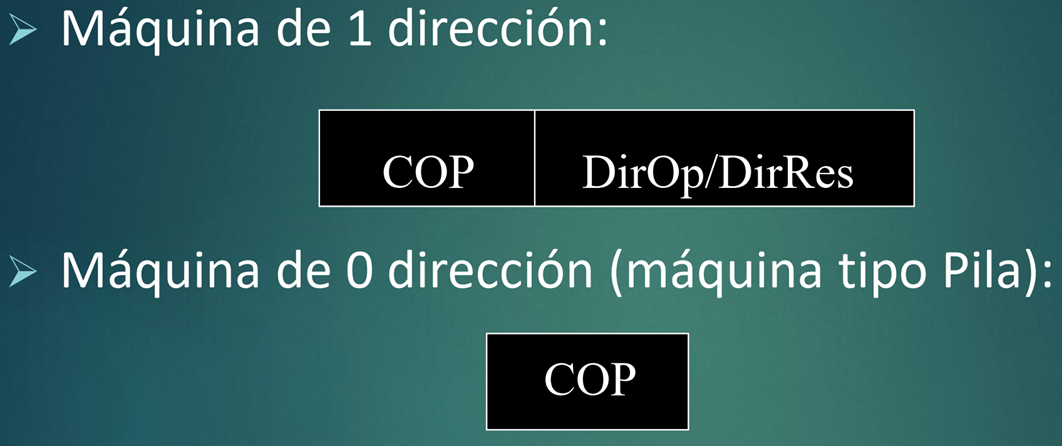
* Unidad central de procesamiento (CPU) formada por:
* Unidad de control (UC)
* Unidad aritmético-lógica (ALU)
* Registros
* Unidad de entrada/salida (de datos) (E/S)
* Unidad de memoria

# CISC vs. RISC:

Para empezar, hay que entender las diferencias en las decisiones de diseño que pueden elegirse:

* **Formato de instrucción**🡪 puede ser:
* Fijo: todas las instrucciones tienen el mismo tamaño (en bits). La captación y ejecución es más sencilla, pero aumenta la cantidad de instrucciones de los programas.
* Variable: la longitud de las instrucciones varía con el tipo de instrucción y modo de direccionamiento. La captación y ejecución es más compleja, pero disminuye la cantidad de instrucciones de los programas.
* **Cantidad de campos de direcciones:**
* A mayor cantidad de referencias de direcciones, se tienen instrucciones más complejas y largas, menos instrucciones por programa, y captación y ejecución de las instrucciones en forma más lenta.
* A menor cantidad de referencias de direcciones, se tiene instrucciones más simples y cortas, más instrucciones por programa, captación y ejecución de las instrucciones más rápida.
* Existen 5 tipos de máquinas básicos en cuanto a direcciones:





* **Tipos de datos:** Los datos con los que la máquina puede trabajar. En general son *numérico* (con o sin signo, etc.), *carácter* (ASCII, etc.) y *lógico* (booleano - manipulación de bits).
* **Repertorio de instrucciones**🡪 se consideran:
* Cantidad de instrucciones (operaciones)
* Tipos de instrucciones
* Características de las instrucciones (sencillas o complejas)
* **Registros**🡪 como son un recurso escaso, debe hacerse énfasis en su administración y uso correctos. En particular:
* Tamaño y cantidad de registros
* Tipos de registros (cómo va a ser su uso y a qué instrucciones se aplican)
* **Modos de direccionamiento**🡪 son las diferentes formas de especificar el lugar donde reside un operando, donde se va a guardar el resultado, o donde buscar la próxima instrucción a ejecutar. Algunos de los más típicos son:
* Inmediato
* Directo: a memoria (“absoluto”), a Registro
* Indirecto: a memoria (en desuso), con registro
* Base: base, base Indexado
* Relativo al PC
* Pila (o relativo al SP)

Para el diseño del procesador se tiene en cuenta:

* Cantidad de modos de direccionamiento
* Tipos de modos de direccionamiento

Visto esto, los diseños en las arquitecturas de procesadores se han orientado hacia 2 estrategias:

* Aumentar la complejidad y prestaciones del procesador (CISC).
* Simplificar y optimizar los procesadores (RISC).

“*BÁSICAMENTE*…”

Los procesadores **CISC** (Computadores de Repertorio **Complejo** de Instrucciones) tienen:

* Repertorio de instrucciones muy amplio
* Muchos modos de direccionamiento
* Formato de instrucción variable
* E/S separada de la memoria (sentencias IN/OUT)

Los procesadores **RISC** (Computadores de Repertorio **Reducido** de Instrucciones) tienen:

* Repertorio de instrucciones simplificado
* Pocos modos de direccionamiento
* Formato de instrucción fijo
* E/S compartida en memoria (CONTROL 0x10000 y DATA 0x10008)

Técnicas de almacenamiento🡪 Las técnicas de almacenamiento se refieren a la forma en que los datos se guardan en memoria, particularmente en los casos en que ocupan más de una palabra de memoria.

Existen 2 formas de almacenar el dato de acuerdo al orden en que se almacenan las diferentes partes del dato:

* **Big-Endian:** el byte **más** **significativo** se almacena en la dirección de memoria más baja. Es decir, se almacena “al derecho”. En un dato 3 2 1 0 se almacenaría:

|  |  |
| --- | --- |
| 1000 H | 3 2 |
| 1001 H | 1 0 |

*Ejemplo* de procesadores con almacenamiento Big-Endian:

* IBM S/370
* Motorola 680x0 (antiguos procesadores de las Mac)
* La mayoría de los procesadores RISC
* **Little-Endian:** el byte **menos significativo** se almacena en la dirección de memoria más baja. Es decir, se almacena “al revés”. En un dato 3 2 1 0 se almacenaría:

|  |  |
| --- | --- |
| 1000 H | 1 0 |
| 1001 H | 3 2 |

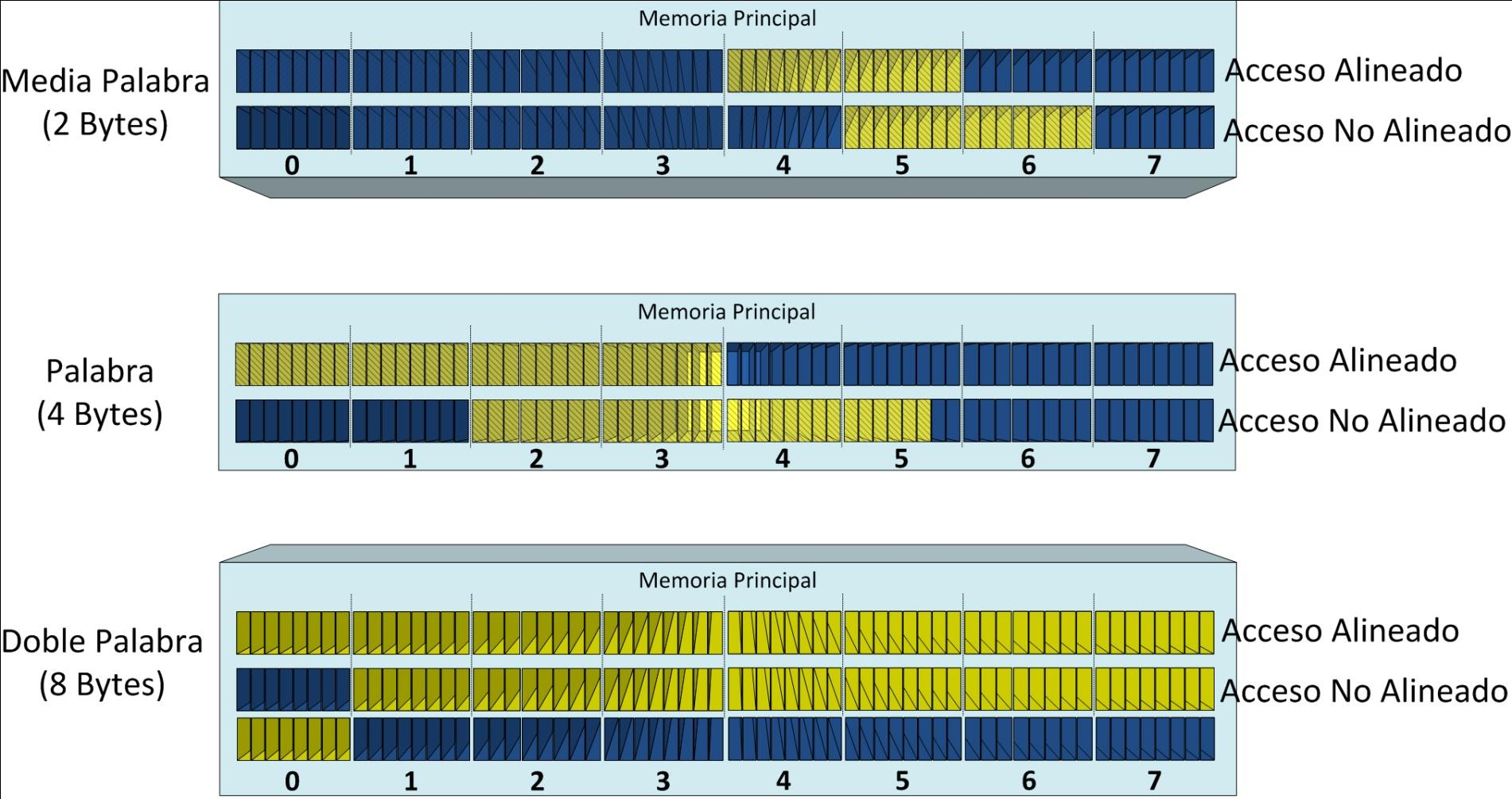
*Ejemplo* de procesadores con almacenamiento Little-Endian:

* Intel 80x86
* Pentium
* VAX

Notar que las 2 formas de almacenamiento son **incompatibles** entre sí.

El otro aspecto relacionado con el almacenamiento de los datos, es el referido a las direcciones de memoria donde se guardan los datos, particularmente en los casos en que ocupan más de una palabra de memoria. Existen 2 formas de almacenar los datos múltiples en memoria:

* **Alineados**: si un dato de n bytes empieza en una dirección de memoria múltiplo de 2m, donde m=log2n, con n>1. En otras palabras, los m bits menos significativos son 0.
* **No alineados**: si no cumple la condición anterior.



Depende el procesador, se admite uno o ambos tipos de acceso. Pero en general, **el almacenamiento alineado es más rápido** que el no alineado.

Ciclo de instrucción🡪 El ciclo de instrucción es la *secuencia* de acciones que realiza el procesador para resolver las instrucciones. En su forma más sencilla, se puede descomponer en 2 acciones básicas:

* **Captación de la instrucción:**
* La dirección de la instrucción que se debe captar se carga en el registro Contador de Programa (**PC**).
* La UC envía al bus de direcciones el valor almacenado en el PC.
* La UC lee la instrucción desde la Memoria, y la almacena en un registro temporal denominado Registro de Instrucción (IR).
* El PC se incrementa en 1 (se prepara para apuntar a la próxima instrucción).
* La UC decodifica (“interpreta”) la instrucción captada.
* **Ejecución de la instrucción**🡪 depende del tipo de instrucción. Algunas de las posibles acciones son:
* Acciones procesador – memoria: por ejemplo, transferencia de datos CPU - Memoria.
* Acciones procesador - E/S: por ejemplo, transferencias de datos entre la CPU y un dispositivo de E/S.
* Acciones de procesamiento de datos: operaciones aritméticas o lógicas con los datos.
* Acciones de control: por ejemplo, alteración de la secuencia de ejecución (salto, salto a subrutina).
* Acciones combinadas de las anteriores.

Pila🡪 La Pila (stack) es una estructura *ordenada* de datos en la que sólo uno de ellos es accesible en un instante dado, al ser de tipo **LIFO** (Last In- First Out). El punto de acceso se denomina **cabecera de la pila** y el acceso se hace mediante un registro específico denominado **Puntero de Pila** (**SP**, Stack Pointer). El número de elementos de la pila, o longitud, es variable.

Sobre la Pila se pueden realizar 2 operaciones básicas, inversas entre sí:

* Almacenar o “apilar” (**PUSH**)
* Extraer o “desapilar” (**POP**)

Subrutinas🡪 Es una sección de código, que *recibe el control* en un punto de entrada, y lo *devuelve* en un punto de salida. Su objetivo es realizar una tarea definida para la cual se le transfiere el control. Una vez finalizada la tarea, devuelve al programa que la invocó en el punto dónde lo hizo.

**Funcionamiento**:

* Al invocarla, el programa le transfiere el control. Es decir, el PC se carga con la dirección de comienzo de la subrutina, y la CPU comienza a ejecutar las instrucciones de la subrutina.
* Cuando la subrutina complete su tarea, la última instrucción que ejecuta es la de RET. En ese momento el PC se carga con la dirección de la instrucción siguiente al CALL en el programa principal.
* Las subrutinas realizan un trabajo determinado. Ese trabajo puede producir un resultado, que debe ser devuelto al programa que la invoca. Típicamente la subrutina devuelve 1 o más resultados.
* El resultado se retorna por las mismas posiciones de memoria donde la subrutina recibió los parámetros.
* Para poder recuperar la dirección de la instrucción siguiente al CALL en el programa principal, esta dirección debe haber sido guardada previamente. El lugar donde se guarda esta dirección es la Pila. Cada vez que se ejecuta una instrucción de CALL, se guarda en la Pila el valor actual del PC.
* Cuando se ejecuta una instrucción RET, el PC se carga con la dirección almacenada en el tope de la Pila y vuelve al programa donde lo dejó.

**Ventajas**:

* **Economía del programa**: permite la reusabilidad de código.
* **Modularidad**: permite dividir el programa en unidades más pequeñas y específicas.

Para realizar su trabajo, se requieren pasar parámetros entre el programa que invoca y la subrutina invocada. Esto puede hacerse mediante:

* **Vía registros:**
* Sencillo
* El número de registros es la gran limitación
* Es importante documentar que registros se usan ya que van a ser modificados
* **Vía memoria:**
* Se usa un área definida de memoria (RAM)
* Difícil de estandarizar
* **Vía pila (stack):**
* Los parámetros se pasan a través de la pila
* Es el método más ampliamente usado por las ventajas que supone respecto al resto
* Los registros no tienen que ser modificados.
* Hay que manejar correctamente la pila, ya que es usada por el usuario y por el sistema, lo que puede llevar a errores si se usa de forma irresponsable.

# Clase 2:

Una **interrupción** es un mecanismo que permite alterar el proceso de “ejecución normal” de la CPU, haciendo que esta suspenda la tarea que está haciendo y responda a una solicitud de atención para resolver otra tarea. Una vez finalizada la interrupción, el procesador retoma la tarea suspendida en el punto dónde la dejó (similar al llamado a subrutina). Para atender la interrupción se requieren 3 pasos:

* **Detener** (suspender) la tarea que está ejecutando el procesador (suspender, no terminar ni abortar). Además, hay que **salvar** tanto el contenido del PC como el **estado** del programa.
* **Bifurcar** (saltar) a otra tarea en un área de memoria definida, asociada a la solicitud de interrupción (comúnmente conocida como servicio interrupción). Además, se ponen en cola las demás interrupciones (si las hubiese).
* **Restablecer** la tarea suspendida en las condiciones en las que se encontraba en el momento en el que se la detuvo, recuperando como **mínimo** el **PC** y el **PSW**.

El origen de una interrupción es la ocurrencia de un evento que requiere la intervención de la CPU. Existen 2 tipos de eventos:

* **Interno**: debido a la ocurrencia de una situación dentro del sistema de cómputo. Algunos eventos internos que pueden producir interrupción son:
* Un error asociado a la ejecución de una instrucción
* Desbordamiento aritmético (overflow)
* División por cero
* Temporizados propios del sistema
* Fallo del hardware
* Error de paridad en la memoria
* Pérdida de energía.
* **Externo**: asociado a operaciones de E/S con periféricos. Algunos eventos típicos que pueden generar interrupción son:
* Finalización de una transferencia
* Error en la transferencia
* Dispositivo indisponible

Interrupciones múltiples🡪 La necesidad de administrar eventos de distinto origen requiere, en la mayor parte de los casos, administrar varias interrupciones. Dado el origen diverso de las interrupciones, hay algunas que son más importantes que otras:

* Las interrupciones más importantes deben tener mayor “prioridad” que las menos importantes.
* Cuanto mayor sea su prioridad, mayor es la urgencia para ser atendida, incluso si hay una interrupción en curso.
* Si las interrupciones son de **igual prioridad**, se procesan en  
  el **orden en que llegan**.
* Si las interrupciones tienen distinta prioridad, **las  
  interrupciones de mayor prioridad pueden interrumpir a las  
  de menor prioridad**. Una vez atendida la interrupción prioritaria, el procesador **vuelve** a las **interrupciones previas.**

Típicamente hay 2 categorías de interrupciones:

* **No enmascarables**: son interrupciones que **no pueden ser ignoradas**, y están asociadas a eventos críticos, peligrosos o de alta prioridad.
* **Enmascarables**: pueden ser, eventualmente, “ignoradas”. Para ello el procesador permite realizar algunas acciones que pueden inhibir la atención de la interrupción. Las interrupciones enmascarables generalmente están asociadas a operaciones menos críticas, por ejemplo, de E/S.

Interrupciones por hardware y software:

* Por **Hardware**: son generadas por señales físicas asociadas a eventos externos o internos. De acuerdo al origen de la señal de interrupción, pueden ser:
* **Externas**: son conocidas como **Interrupt request**.
* El origen de estas señales de pedido de interrupción proviene típicamente de dispositivos conectados al subsistema de **E/S**.
* Se consideran las “**verdaderas**” interrupciones porque son aleatorias (pueden ocurrir en cualquier momento).
* El sistema de cómputo tiene que ser capaz de manejar estos eventos externos “no planeados”.
* Pueden o no estar relacionadas con el proceso en ejecución en ese momento
* **Internas:** son conocidas como **Trap** o **excepciones**.
* Son señales creadas dentro del sistema de cómputo en respuesta a situaciones propias del proceso en ejecución y no vinculadas con operaciones de E/S, por tal motivo no son estrictamente aleatorias.
* Algunos eventos que pueden producir un Trap son:
* Condiciones excepcionales: overflow en ALU de punto flotante.
* Fallas de programa: tratar de ejecutar una instrucción no definida.
* Fallas de hardware: error de paridad de memoria.
* Accesos no alineados o a zonas de memoria protegidos
* Por **Software**: son producto de la ejecución de instrucciones con efecto similar a una interrupción por hardware. Son **llamadas a** **funciones propias del SO**, implementadas para y no tener que:
* Escribir el servicio (complicado).
* Buscar entre todas las llamadas a funciones del BIOS y SO, para reemplazar en el código la dirección de esa función (también muy complicado).

Detección del pedido de múltiples interrupciones🡪 Dentro del proceso de gestión de las interrupciones, lo primero que tiene que hacer la CPU es detectar el pedido de interrupción. Cuando hay múltiples fuentes de interrupción, hay varias formas para identificar el origen del pedido. Los más comunes son:

* **1 señal física de entrada a la CPU por *cada* interrupción**: la implementación es muy sencilla, pero es muy costoso, por eso se suele acotar a un número reducido (3 o 4) señales.
* **1 única señal física de entrada a la CPU *para todas* las interrupciones e *identificación por software*:** comotodas las fuentes están conectadas a la misma entrada, la CPU debe “preguntar” a cada dispositivo si ha producido el pedido de interrupción (esto se llama encuesta o **pooling**). Al tener que preguntar a todos los dispositivos uno por uno se vuelve lento e ineficiente.
* **1 única señal física de entrada a la CPU *para todas* las interrupciones e *identificación por hardware:*** como todas las fuentes están conectadas a la misma entrada, la CPU recibe un número (vector de la interrupción) que identifica a la fuente de la interrupción. Son conocidas como **interrupciones vectorizadas**.

PIC🡪 Para atender las interrupciones vectorizadas, el PIC funciona como mediador entre la CPU y el pedido de interrupción. Funcionamiento:

* El PIC recibe los pedidos de interrupción, típicamente de periféricos que piden atención.
* El PIC solicita atención a la CPU con la única señal de pedido de interrupción **IntR** (Interrupt request).
* Cuando la CPU está lista para atender la interrupción, le avisa al PIC mediante la señal **IntA** (Interrupt acknowledge).
* El PIC genera en el bus de datos el número de la interrupción (vector) a ser atendida. La CPU lee ese número y busca en la memoria el vector correspondiente al servicio de esa interrupción.
* Dado se hace por hardware, es mucho **más rápido.**

El PIC tiene 3 registros principales:

* El **ISR** que identifica la interrupción en servicio.
* El **IRR** que contiene los pedidos de interrupción provenientes de los periféricos.
* El **IMR** que se usa para habilitar/deshabilitar los pedidos de interrupción (esta funcionalidad se conoce como enmascaramiento de interrupciones).

# En MSX88:

* Los vectores de las interrupciones están en la parte más baja de la memoria.
* Cada entrada (o vector) es una palabra doble (4 bytes), que contiene la dirección del procedimiento que brinda el servicio. La parte alta del vector es 0.
* Hay 4 vectores pre asignados (int. por software):
* Tipo\* 0 – finaliza ejecución de programa
* Tipo\* 3 – punto de parada para depuración/seguimiento
* Tipo\* 6 – lectura de entrada std. Requiere el uso de BX.
* Tipo\* 7 – escritura de salida std. Requiere BX y AL

\*Donde tipo es INT x.

* Los registros internos del PIC se sitúan a partir de la dirección 20H (del espacio de direcciones de E/S). Son accedidos con operaciones lectura y escritura en el espacio de E/S, es decir, mediante las instrucciones IN y OUT.
* 20H EOI
* 21H IMR
* 22H IRR
* 23H ISR
* 24H INT0
* ... …
* 2BH INT7
* Donde:
* INT0 – tecla F10 🡪 Produce una interrupción cada vez que se presiona F10
* INT1 – Timer 🡪 Conectada a la salida del Timer
* INT2 – Handshake 🡪 Conectada a una salida para Handshake
* INT3 – DMA 🡪 Conectada a la salida del puerto a impresora
* INT4 a INT7 no usadas

# Clase 3:

A sabiendas de que **los periféricos son más lentos que la CPU**, el subsistema de E/S debe ser flexible para permitir:

* Trasmisión de diferentes cantidades de datos
* Rango de velocidades de transmisión muy amplio.
* Diferentes formatos de dato y tamaño de palabra.

Los dispositivos más sencillos para implementar las transferencias de E/S con periféricos son los **Puertos de E/S**:

* Realizan la interfaz entre el procesador/memoria y un periférico.
* Son, por lo común, administrados por el SO a través de drivers específicos.
* La conexión con el Periférico provee 2 tipos de informaciones:
* **Datos:** Información útil a transferir.
* **Control y Estado:** información que permite realizar la transferencia (en lo posible libre de errores). Por ejemplo: sentido de la transferencia (de entrada, de salida), operación (de lectura, de escritura), estado del periférico (listo, no-listo, en falla).
* El Puerto de E/S debe ejecutar 2 tipos de comunicaciones:
* **Hacia el Periférico** (comunicaciones externas con la periferia):
* Transferir datos con el periférico.
* Controlar y temporizar uno o más dispositivos externos.
* Almacenar temporalmente datos (“buffer”).
* Detectar errores.
* *Sección de manipulación de datos* (buffer/transductor). Almacena y convierte los datos a intercambiar con el periférico
* *Sección de control y estado* (Lógica de control): recibe y genera las señales de control y estado del periférico
* **Hacia el Bus** (internas con CPU y memoria):
* Interpretar las órdenes que recibe de CPU y transmitirlas al periférico.
* Transferir datos con la CPU (registros) y memoria.
* Informar a la CPU del estado del periférico.
* *Sección de manipulación de datos* (bus de datos)
* *Sección de manipulación de señales de control y de direcciones*
* **Registros** del puerto de E/S:
* **Registros de datos**: que contienen la información útil recibida desde, o transmitida a, el periférico.
* **Registros de control y estado**: que controlan las características de la transferencia, y almacenan señales de estado de la comunicación (falla, no listo, etc.), del módulo y del periférico.
* Existen 2 tipos básicos de puerto:
* **Puerto paralelo**: hay varias líneas de datos (n) que transfieren n bits simultáneamente entre el puerto de E/S y el periférico.
* Ejemplos: impresora paralela, scanner, etc.
* Se requiere disponer de una conexión mediante un cable que incluya al menos los n bits de datos, lo que lo hace un método bastante costoso.
* **Puerto serie**: hay 1 línea de dato para la transferencia entre el puerto de E/S y el periférico.
* Ejemplos: impresora serie, red Ethernet, mouse, teclado, etc.
* Se requiere disponer de una conexión mediante un cable sencillo. El costo es mucho menor.
* Los datos deben serializarse (transmitirse de 1 bit por vez, uno a continuación del otro) lo que en teoría sería mucho más lento que transmitir de a n bits simultáneamente, como lo hace el puerto paralelo.

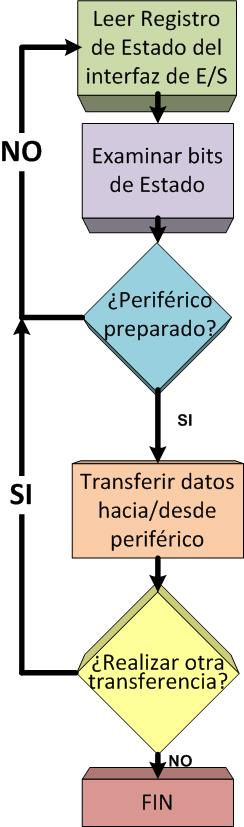
Además de los puertos, hay otros dispositivos más complejos que proveen de otras prestaciones, además de la tarea básica de implementar la transferencia de los datos, por ejemplo:

* Ocultar las propiedades particulares del dispositivo periférico a la CPU: temporizados, formatos, etc.
* Manejar múltiples dispositivos simultáneamente.
* Controlar varias funciones del dispositivo.

Acceso al subsistema de E/S🡪 Existen 2 técnicas para acceder a los registros del subsistema de E/S:

* **Espacio de E/S compartido con memoria (memory-mapped):**
* En esta técnica los registros de los dispositivos de E/S y memoria comparten un único espacio de direcciones.
* Los registros de la E/S se comportan idéntico a una memoria de lectura/escritura.
* No hay instrucciones específicas para E/S, se usan las mismas instrucciones de movimiento de datos a memoria. Ej: MOV EOI, AL
* **Espacio de E/S separado de memoria (modelo Intel)**
* En esta técnica los registros de los dispositivos de E/S y la memoria están en diferentes espacios de direcciones.
* Dado que el bus de direcciones es compartido por la memoria y el subsistema de E/S, se requieren señales de control adicionales para identificar a donde está accediendo la CPU: a la memoria, o a la E/S.
* Hay instrucciones específicas de E/S (IN/OUT), distintas de las instrucciones de acceso a la memoria.

Gestión de la transferencia**🡪** Desde el punto de vista de la gestión para transferir datos entre el Sistema de cómputo y el periférico, existen 3 estrategias básicas de implementación:

* **E/S Programada y espera de respuesta:**
* La CPU interviene directamente en la transferencia de cada unidad de información (byte, word) con el módulo.
* Es decir que la CPU tiene control casi directo sobre la operación de E/S. Entre otras acciones hace:
* Comprueba el estado del dispositivo
* Envía los comandos requeridos (por ejemplo, de lectura, escritura)
* Realiza la transferencia de todos los datos (de a uno)
* En cada dato que es transferido, la CPU espera que el módulo E/S termine la operación, típicamente que el periférico “acepte” el dato.
* Durante la espera, la CPU permanece ociosa (no deseable).
* **E/S Programada y administrada por interrupción:**
* La CPU interviene directamente en la transferencia de cada unidad de información (byte, word) con el módulo.  
  ➢ Cada vez que el módulo está listo (o completó una transferencia) avisa a la CPU con un pedido de interrupción.  
  ➢ Ahora la CPU no tiene que ejecutar el lazo de comprobación del estado del módulo; solo inicia la transferencia al recibir el pedido de interrupción del periférico.  
  ➢ Durante el tiempo que el periférico no está listo, la CPU no tiene que esperar, puede seguir ejecutando otra tarea.
* **E/S con acceso directo a memoria (DMA)🡪** El Acceso Directo a Memoria (DMA) es una técnica de transferencia de datos entre periférico y Memoria sin intervención directa de la CPU:
* No requiere de la intervención directa de la CPU.
* Comúnmente llevada a cabo por un “Controlador de DMA” (o DMAC) específico encargado de llevar a cabo la transferencia.
* Dado que la transferencia por DMA requiere el uso del Bus, tanto el DMAC como la CPU pueden tomarlo. El DMAC y la CPU “compiten” por el uso del Bus.
* Cuando el DMAC toma el Bus (a la vez que la CPU se desconecta lógicamente del mismo), actúa como “master” durante la transferencia por DMA.
* El proceso de transferencia requiere realizar una serie de acciones o fases relativamente complejas. Las principales son:
* **Fase de Inicialización**
* Inicialización interfaz de E/S
* Inicialización DMAC
* **Fase de ejecución de la transferencia**
* Cuando el periférico está listo, pide al DMAC iniciar la transferencia mediante una señal física.
* Cuando el DMAC recibe el pedido del periférico, pide el control del bus mediante alguna señal especial a la CPU.
* Cuando reconoce el pedido de DMA, la CPU entrega (libera) el bus y se “desconecta” (ya no controla el bus) lógicamente del mismo.
* La CPU avisa al DMAC que liberó el bus mediante otra señal especial.
* Al liberar la CPU el bus, el DMAC toma el control del bus y ejecuta la transferencia hasta terminarla.
* El DMAC avisa al periférico que puede iniciar la transferencia.
* El periférico comienza a transferir los datos, a través de bus, con la memoria, de a uno por vez.
* Después de la transferencia de cada palabra se actualizan los registros del DMAC:
* Nº de bytes faltantes
* Próxima dirección de memoria a donde guardar el dato
* Cuando el número de bytes faltantes es igual a 0 significa que transfirió todos los datos y terminó la transferencia.
* **Fase de finalización y análisis de la transferencia**
* El DMAC libera el bus y le avisa a la CPU por medio de una señal física.
* La CPU retoma el control del bus.
* El DMAC suele activar una señal de interrupción para indicar a la CPU la finalización de la operación de E/S solicitada.
* La CPU, mediante la interrupción, verifica el resultado de la transferencia vía los registros internos del DMAC.
* **Ventajas e inconvenientes del DMA:**
* La principal ventaja es la eficiencia, dado que la CPU se libera de tener que controlar la transferencia de los datos. Solo prepara la transmisión, y verifica el resultado de la misma.
* La principal desventaja se origina en el uso del bus. Como las transferencias por DMA pueden tener mayor prioridad que la CPU, se puede degradar el rendimiento de la CPU si el DMAC hace uso intensivo del bus. Sin embargo, no necesariamente la CPU necesite todo el tiempo el bus porque:
* En el caso de Computadoras con memoria caché:
* La mayor parte del tiempo, la CPU lee instrucciones de la cache, por lo que no necesita usar el bus de memoria.
* El DMAC puede aprovechar estos intervalos en los que la CPU está leyendo instrucciones de la cache (y por tanto no usa el bus de memoria) para realizar las transferencias.
* En el caso de computadores sin cache
* El procesador no utiliza el bus en todas las fases de la ejecución de una instrucción.
* El DMAC puede aprovechar las fases de ejecución de una instrucción en las que la CPU no utiliza el bus.
* Hay varias **formas** distintas de **implementar** **las** **transferencias** **por** **DMA**. 2 típicas son:
* **Por ráfagas (burst)**🡪 Es la vista hasta ahora. El DMAC solicita el control del bus a la CPU:
* Cuando la CPU concede el bus, el DMAC no lo libera hasta haber finalizado la transferencia de todo el bloque de datos completo.
* VENTAJAS:  
  La transferencia se realiza de forma muy rápida.
* DESVENTAJAS:  
  Durante el tiempo que dura la transferencia, la CPU no puede utilizar el bus con memoria, lo que puede degradar el rendimiento del sistema.
* **Por robo de ciclo (cycle-stealing)**🡪 El DMAC solicita el control del bus a la CPU:
* Cuando la CPU concede el bus al DMAC, se realiza la transferencia de una única palabra y después el DMAC libera el bus.
* El DMAC solicita el control del bus tantas veces como sea necesario hasta finalizar la transferencia del bloque completo.
* Es importante destacar que esto no es una interrupción, por lo que el procesador no guarda ningún contexto porque su tarea no se ve interrumpida.
* VENTAJAS:
* No se degrada tanto (un poco sí) el rendimiento del sistema y de la CPU.
* DESVENTAJAS:
* La transferencia puede tardar un poco más de tiempo.

Canales de E/S🡪 Las transferencias de E/S se pueden dividir, en función de la capacidad para interactuar con los periféricos, en varios niveles:

* **Nivel 1🡪** *CPU + Módulo de Interfaz de E/S o controlador*:
* La CPU controla directamente los periféricos e interfaz, y administra la transferencia por programa (con espera).
* **Nivel 2🡪** *CPU + Módulo de Interfaz E/S o controlador con interrupción:*
* La CPU controla directamente los periféricos y administra la transferencia con programa e interrupciones.
* **Nivel 3🡪** *DMA (DMAC + Módulo de E/S):*
* La CPU no interviene directamente, solo prepara y supervisa la transferencia.
* **Nivel 4🡪** *Canal de E/S básico (Procesador básico + módulo de E/S):*
* La CPU interviene mínimamente.
* **Nivel 5🡪** *Canal de E/S inteligente (Procesador inteligente + módulo de E/S):*
* La CPU no interviene, excepto situaciones especiales.

Hay **2 tipos básicos de canales de E/S:**

* **Canal selector de E/S:**
* El canal controla varios dispositivos de alta velocidad, de a uno por vez.
* El canal selecciona un dispositivo y efectúa la transferencia sobre dicho dispositivo.
* Cada dispositivo tiene asociado un controlador o módulo de E/S que lo maneja.
* Por lo tanto, el canal de E/S ocupa el lugar de la CPU en el control del módulo de E/S
* **Canal Multiplexor de E/S**:
* El canal controla varios dispositivos de alta velocidad, incluso simultáneamente.
* El canal mutiplexa la atención entre los dispositivos seleccionados. El multiplexado puede ser:
* *Multiplexor de bytes:* acepta y transmite de a caracteres.
* *Multiplexor de bloques:* intercala bloques de datos desde distintos dispositivos.

# Clase 4:

Hay 2 tipos de ejecución de la instrucción:

* Ejecución **MONOCICLO**:
* Cada vez que el PC envía una dirección a la memoria de instrucciones, se accede a una nueva instrucción.
* El dato fluye por las unidades funcionales (registros, ALU, memoria, MUX, etc.) hasta que se complete la instrucción.
* No se inicia un nuevo ciclo de instrucción hasta que el PC se carga con un nuevo valor
* El ciclo de reloj se hace en función de la instrucción “más larga”.
* Las instrucciones que son “más cortas” tienen que esperar un tiempo hasta que se complete el período de reloj. Si la tasa de instrucciones cortas es mucho mayor que la de las largas, se puede perder mucho tiempo inútilmente.
* Ejecución **MULTICICLO**:
* Cada instrucción requiere varios ciclos de reloj.
* Las instrucciones más rápidas usan menos ciclos de reloj que las más lentas.
* El ciclo de instrucción es variable y se ajusta al tipo de instrucción.

Segmentación y solapamiento del cauce (Pipelining)🡪 La segmentación del cauce (Pipelining) consiste en:

* Descomponer el proceso de ejecución de las instrucciones en **fases** o etapas.
* Así como se segmenta el cauce de datos (instrucciones), también se debe hacer lo mismo con las señales que controlan las unidades funcionales: MI (memoria de instrucciones), BR (banco de registros), ALU (unidad aritmético-lógica), MD (memoria de datos). Eso es así porque cada unidad funcional está operando con diferentes instrucciones (en diferentes estados de ejecución).
* Las fases o etapas son ejecutadas por estas unidades separadas y capaces de operar simultáneamente.
* Las instrucciones se van ejecutando a medida que se liberan las unidades.
* Las instrucciones no necesitan esperar la terminación de la previa para comenzar a resolverse.
* Es una forma efectiva de organizar el hardware de la CPU para realizar más de una tarea al mismo tiempo. Explota el paralelismo en el flujo secuencial de instrucciones.
* Podría decirse que el comportamiento de un procesador ejecutando una secuencia de instrucciones se parece a una línea de montaje en una planta de manufactura.
* Dado que el procesador tarda menos tiempo en resolver un conjunto de instrucciones, la segmentación mejora las prestaciones (a nivel de diseño del hardware).
* La segmentación es “invisible” al programador.
* Se puede usar tanto en procesadores RISC como CISC.
* El diseño de procesadores segmentados tiene gran dependencia del repertorio de instrucciones.
* ¿Cuál sería la ejecución ideal con esta segmentación?
* El máximo rendimiento es completar una instrucción con cada ciclo de reloj.
* Si K es el número de etapas del cauce ⇒ Vel. procesador segmentado = Vel. secuencial x K
* El incremento potencial de la segmentación del cauce es proporcional al número de etapas del cauce.
* Desde el punto de vista físico, en Mips, la segmentación del cauce de los datos se hace mediante registros.
* Los registros (sincrónicos) deben intercalarse entre cada unidad funcional.

Diagrama de estados🡪 **NanoMips** tiene una secuencia de instrucción compuesta por 5 estados:

* **Fase 1 - Búsqueda de instrucción (F, Fetch):**
* Ocurre en todas las instrucciones.
* Busca y lee la instrucción en la memoria (de instrucciones).
* Actualiza el PC (sumar 4 al valor actual del PC): cada instrucción ocupa 4 bytes, por lo que la próxima instrucción consecutiva está en la dirección actual + 4.
* **Fase 2 - Decodificación (D, Decode)** *y acceso a registros***:**
* Ocurre en todas las instrucciones.
* Se decodifica la instrucción.
* Se accede al banco de registros y se obtienen los operandos.
* Si es un salto se verifica si se toma o no. Si se toma también se actualiza el PC.
* Opcional: Extensión del signo del offset para cálculo de la dirección efectiva.
* **Fase 3 - Ejecución (X, Execute):**
* Ocurre en todas las instrucciones.
* Si es procesamiento: ejecución en la ALU.
* Si es acceso a memoria: obtención de la dirección efectiva.
* Si es salto: se ejecuta el salto.
* **Fase 4 -Acceso a memoria (M, Memory Access):**
* Ocurre solo en las instrucciones de LOAD y STORE.
* Se accede a memoria.
* **Fase 5 - Almacenamiento en Registro (W, Writeback):**
* Ocurre en las instrucciones que almacenan un dato en un registro.
* También es en esta fase cuando puede calcular el desplazamiento a sumar al PC en instrucciones de salto.

# Problemas con la segmentación:

* No todas las instrucciones necesitan todas las etapas.
* No todas las etapas pueden ser manejadas en paralelo.
* Los programas tienen instrucciones de salto.

Los conflictos que aparecen si se tienen en cuenta los problemas anteriores, se denominan **riesgos**. Existen 3 tipos:

* **Estructurales**: provocados por conflictos en el uso de los “recursos”. Los recursos típicamente son: memoria, ALU, registros.
* **Por dependencia de datos:** son conflictos originados entre 2 o más instrucciones que comparten un mismo dato. Por ejemplo, una instrucción produce un resultado que lo necesita otra, ambas instrucciones dentro del cauce de datos. Se pueden dar 3 tipos de dependencias:
* Lectura después de Escritura (**RAW**, dependencia verdadera)
* Una instrucción escribe un dato que otra lee posteriormente
* Escritura después de Escritura (**WAW**, dependencia en salida)
* Una instrucción escribe un dato que otra escribe posteriormente
* Sólo ocurre si se permite que las instrucciones se adelanten unas a otras (alteración en la secuencia de ejecución de instrucciones)
* Escritura después de Lectura (**WAR**, antidependencia)
* Una instrucción lee un dato que otra escribe posteriormente
* No se puede dar en nuestro cauce simple
* **Por dependencia de control:** Son riesgos que pueden ocurrir cuando se va a ejecutar una instrucción de salto condicional.
* Una instrucción tiene que calcular el nuevo valor que modifica el valor del PC. La próxima instrucción no puede comenzar hasta que no se resuelva el salto.
* Una forma de resolver el conflicto es retrasar varios ciclos (3) la próxima instrucción hasta que se resuelva el cálculo del salto.

# Clase 5:

# Soluciones a los problemas de la segmentación:

* **Solución elemental🡪** La solución más elemental para superar los atascos, producto de los 3 tipos de riesgos que se pueden presentar en un procesador segmentado, es agregar puntos de parada en el cauce hasta que el conflicto desaparezca:
* El gran problema que tiene este tipo de solución es que el rendimiento cae por la cantidad de tiempos muertos que se insertan, inclusive puede llegar a ser hasta inútil aplicar la segmentación.

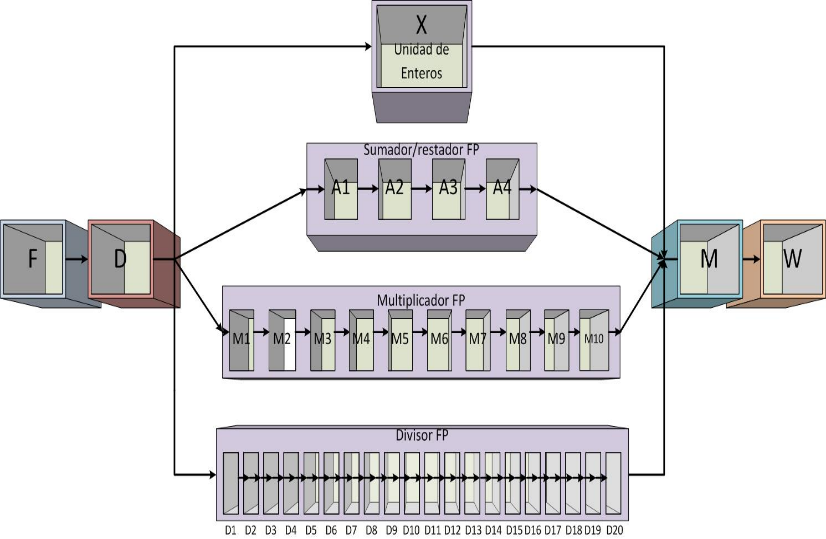
Hay varias técnicas que permiten reducir y, en algunos casos, eliminar los efectos de los riesgos:

* **Solución** a conflictos **estructurales**:
* *Duplicación de recursos de hardware*. Por ej: agregando sumadores o restadores además de la ALU.
* *Separación del recurso en conflicto.* Por ej: separar memorias de instrucciones y datos.
* *Subdividir el acceso al recurso.* Por ej: en el acceso a los registros (banco de registros) la escritura de un registro se puede hacer en el primer semiciclo de reloj, y la lectura en el segundo, con lo que en un ciclo se pueden hacer las 2 operaciones, siempre y cuando la velocidad del recurso lo soporte.
* **Solución** a conflictos por **dependencia de datos**🡪 Las soluciones se pueden implementar de 2 maneras (ambas “invisibles” al programador):
* *Por Hardware:*
* Retardos (puntos de parada) en las etapas del cauce, es la solución elemental. *Problema*: se reduce *mucho* la performance.
* Adelantamiento de operandos (**forwarding**) entre etapas del cauce:
* Consiste en pasar directamente (“adelantar”), desde una unidad funcional a otra, el resultado obtenido en una instrucción a las instrucciones siguientes que lo necesitan como operando.
* Si el dato que necesita la instrucción i+1 ya está calculado por la instrucción i, se puede llevar (adelantar) a la entrada de la etapa de la instrucción i+1 que lo necesita, sin esperar la etapa final de escritura de la instrucción i.
* Esto se hace guardando el resultado en unos multiplexores (*MUX*), que funcionan como buffers, y llevando el resultado desde allí hasta la unidad funcional que usa la próxima instrucción.
* Esta solución puede ser relativamente sencilla de implementar si se identifican todos los adelantamientos y se pueden adelantar los datos a las unidades que lo necesitan.
* *Por Software (compilador):*
* Introducción de **NOP** (equivalente a insertar retardos), *equivalente* a introducir **puntos de parada**. *Mismo problema*: se reduce fuertemente la performance.
* Reordenamiento de instrucciones. Hay que tener cuidado para no modificar el comportamiento del programa.
* **Solución** a conflictos por **dependencia de control**:
* Sabemos que los saltos pueden ser de 2 tipos:
* Incondicional: donde la dirección de destino se debe determinar lo más pronto posible para reducir la penalización.
* Condicional: introduce el riesgo adicional por la dependencia entre la condición de salto y el resultado dependiente de una instrucción previa.
* Por ejemplo: en la instrucción BEQ, recién se calcula la dirección de salto durante la fase W.
* Soluciones:
* Agregar retardos (solución elemental). Mala performance.
* Modificar la ruta de datos para reducir la cantidad de ciclos muertos, adelantando la resolución de los saltos a la etapa **D**.
* El objetivo del adelantamiento es evaluar tempranamente la condición de salto, usando, por ejemplo, un restador extra a la salida del banco de registros, en lugar de esperar a la etapa X asociada a la ALU, para determinar la condición de igualdad de la instrucción BEQ.
* Además, se puede agregar un sumador adicional para calcular la dirección de salto, en lugar de esperar a la etapa X con la ALU, para adelantar la obtención del salto.
* Difícil de implementar en máquinas más complejas. Por lo que hay otros 2 tipos de técnicas:
* *Técnicas* *por* *hardware*🡪 que consiste en resolver los conflictos a nivel de hardware mediante predicción de saltos. Pueden ser:
* Técnicas estáticas: no tienen en cuenta información previa de la ejecución del programa. Básicamente se presume una condición (salta o no salta) y se ejecuta en base a esa predicción. Puede:
  + Predecir que nunca se salta.
  + Predecir que siempre se salta.

Son sencillas de implementar y no hay que hacer grandes cambios, pero la instrucción que se descarte puede afectar al estado del procesador.

* Técnicas dinámicas: se basa en el análisis de datos previos sobre cómo se comporta el programa, para predecir la acción.
* Algunos ejemplos de técnicas de predicción por hardware son:
* **Conmutador Saltar/no saltar**🡪 Estando en uno de las 2 condiciones (predecir que salta o predecir que no salta), se requieren 2 predicciones fallidas consecutivas para conmutar al otro estado.
* **Tabla de historia de saltos** (BHT branch history table):
* Cada vez que se precapta una instrucción, se busca en la cache BHT si la instrucción es de ramificación.
* Se toma una decisión predictiva en función de los bits de la historia de uso.
* Si la predicción es saltar, se capta la dirección de salto.
* Si la predicción falla, se actualiza la tabla
* Si la instrucción de ramificación no estaba en la BHT, se la carga como nueva entrada.
* **Predicción según el código de operación🡪** Se basa en la suposición de que algunas instrucciones de salto tienen mayor probabilidad de saltar o de no saltar. La tasa de acierto puede llegar a alcanzar un 75%
* **Varios cauces** (Multiple stream):
* Se usan cauces distintos (duplicando hardware) para ejecutar simultáneamente los cauces de salto y no salto.
* Cuando se determina el resultado de la ramificación, se debe utilizar el cauce correcto y descartar el incorrecto.
* Como se están captando simultáneamente 2 cauces distintos, aumentan las búsquedas, lo que puede provocar retardos en el acceso al bus y a los registros.
* Si hay nuevos saltos en los cauces en ejecución, se necesita disponer de más cauces para contemplar esta situación, lo que aumenta la complejidad del hardware.
* **Precaptación del destino del salto** (Prefetch branch target):
* Se precapta la instrucción destino del salto, además de las instrucciones siguientes a la bifurcación.
* La instrucción se guarda hasta que se ejecute la instrucción de bifurcación
* **Buffer de bucles:**
* Se usa una memoria muy rápida que contiene las últimas instrucciones recientemente buscadas.
* Si hay una instrucción de salto, el hardware comprueba si está en el Buffer de bucles. Si es así, la próxima instrucción se busca desde el buffer.
* Muy eficaz para pequeños bucles y saltos, si el buffer es capaz de contener todo el bucle.
* *Técnicas* *por* *software*: Las técnicas de resolución por software de conflictos debido a las instrucciones de salto se basan en tratar de realizar trabajo útil mientras el salto se resuelve.
* Cuando hay una instrucción de salto, se necesita 1 ciclo (o más) de tiempo para determinar si se va a ejecutar el salto o no. Ese tiempo se llama hueco o ranura de retardo de salto (**Branch delay slot**).
* Hay máquinas que captan y ejecutan siempre la instrucción siguiente a una instrucción de ramificación (en lugar de descartarla).
* El compilador puede tratar de insertar, en dichos huecos, instrucciones útiles que en lo posible no dependan del salto. De esta manera se elimina el conflicto y se hace trabajo útil.
* Si no es posible insertar instrucciones, se necesita agregar instrucciones NOP para evitar el conflicto.
* Las instrucciones a reordenar pueden provenir de:
* Instrucciones antes del salto
* Instrucciones del destino del salto
* Instrucciones a continuación del salto
* La mejor solución sería reordenar instrucciones anteriores a la de salto, porque siempre deben ejecutarse y van a ser útiles.

# Procesador segmentado NanoMips multifuncional:

* En general, es imprescindible que los procesadores dispongan de operaciones en **punto flotante**.
* Si se quiere agregar operaciones en punto flotante en el NanoMIPS, se requiere realizar las siguientes modificaciones:
* Agregar instrucciones en PF (Ej: ADD.D, LD.D)
* Agregar registros de PF (Ej: F1, F2, etc.)
* Agregar hardware para las operaciones aritméticas de: Suma, Resta, Multiplicación y División
* El problema es que los tiempos de ejecución de las unidades de punto flotante son muy distintos de los de la unidad de enteros.
* Hasta ahora habíamos considerado que todas las etapas del NanoMIPS tenían la misma duración, pero ahora las unidades funcionales de PF pueden requerir varios ciclos para completarse.
* **Conclusión**: cuando se incorpora punto flotante, no todas las etapas del cauce van a durar lo mismo.
* Si las unidades funcionales en punto flotante tardan varios ciclos, entonces una secuencia de 2 instrucciones consecutivas en punto flotante requeriría retardar mucho la segunda instrucción hasta que la primera se complete.
* Hay una solución a este problema: segmentar las unidades funcionales de punto flotante.
* En la figura se ha segmentado:
* La unidad de suma en 4 etapas.
* La Unidad de multiplicación en 10 etapas.
* La unidad de división en 20 etapas.

# Clase 6:

* La incorporación de nuevas funcionalidades se orientó principalmente en mejorar la eficiencia en la ejecución de programas. Un aspecto de esta mejora consistió en brindar un mayor soporte a los lenguajes de alto nivel (**HLL**), sobre todo los más complejos.
* La “distancia” entre un HLL y el lenguaje de máquina se denomina **GAP** **semántico**, que es la relación entre la cantidad de sentencias de un programa en HLL y la cantidad de instrucciones de máquina que se requieren para resolverlo.
* El mayor soporte tiende a reducir el “GAP semántico”, agregando instrucciones de máquina más complejas que resuelvan las sentencias del HLL más directamente.
* ¿Analizando el funcionamiento de HLL que conclusiones podemos sacar?
* El procedimiento llamada/retorno consume mucho tiempo (demandan mucho acceso a memoria).
* Algunas instrucciones HLL conducen a *muchas* *operaciones* de código máquina.
* La optimización del programa debería centrarse en el uso de variables locales.

# Procesadores CISC (Complex Instruction Set computer):

* Los conceptos anteriores sentaron las bases de diseño de los procesadores “convencionales”:
* Repertorios de instrucciones grandes.
* Formato de instrucción complejo y de longitud variable.
* Muchos modos de direccionamiento.
* Capacidad de implementar sentencias de HLL con muy pocas instrucciones de máquina. Básicamente resolviendo una sentencia de HLL “por hardware”.
* Con el uso masivo de la tecnología VHSI (circuitos integrados de muy alta escala de integración) y el desarrollo de procesadores tipo CISC, se empezaron a realizar diferentes estudios sobre el comportamiento de estos procesadores. Los estudios se orientaron a:
* Uso del repertorio de instrucciones
* Uso de los operandos
* Secuencia de ejecución de instrucciones

Los estudios se realizaron de 2 maneras distintas:

* **Estáticos**🡪 computaban cantidades basados en el código objeto ejecutable, sin tener en cuenta la ejecución. Es una forma sencilla de obtener métricas, pero pueden resultar bastante equivocados. Por ejemplo, la cantidad de instrucciones que componen un lazo no tiene nada que ver con las veces que se ejecuta.
* **Dinámicos**🡪 computan cantidades en base a la ejecución de diferentes programas. Son más realistas porque los resultados tienen en cuenta la situación real por la que pasan los procesadores.
* Las conclusiones finales a las que arribaron los que llevaron a cabo estos estudios fueron:
* **Optimización**: se necesita optimizar el tiempo de ejecución de las instrucciones más usadas y las que consumen más tiempo.
* **Simplificación** de las instrucciones, para resolverlas más rápidamente.
* **Ajuste**: del cauce de datos para resolver las instrucciones más usadas lo más rápido posible.
* **Registros**: usar los registros en forma intensiva, para manejo de operandos. Disponer de un banco de registros significativo para reducir los accesos a memoria.
* **Ventajas de CISC:**
* El compilador es más sencillo por disponer de un repertorio amplio de instrucciones y modos de direccionamiento
* Los programas son más pequeños, tiene menos instrucciones, lo que probablemente implique que ocupa menos memoria. *Sin embargo*, la memoria hoy día es muy barata, por lo que esta ventaja es muy relativa.
* El número de bits de memoria que ocupa no tiene porqué ser más pequeño al tener menos instrucciones
* Las referencias a registros necesitan menos bits.
* Velocidad de ejecución🡪 Se supone que aumentar la complejidad del procesador debería mejorara su velocidad en ejecutar los programas. Pero:
* Los procesadores CISC casi no usan las instrucciones más complejas.
* El repertorio de instrucciones complejo exige una Unidad de control más compleja y lenta.
* Una Unidad de control más lenta aumenta el tiempo de ejecución de las instrucciones simples
* En definitiva, no está comprobado que la tendencia hacia CISC fuera la apropiada.
* **Desventajas de CISC:**
* Las instrucciones de máquina complejas son difíciles de aprovechar por el compilador, es decir las usa poco nada.
* La optimización es más difícil de realizar.
* Al tener repertorio de instrucciones más amplio, los campos de código de operación son más largos y aumentan el tamaño de la instrucción.

# Procesadores RISC (Reduced Instruction Set Computer):

Los estudios realizados sobre el comportamiento de los programas, condujeron a un nuevo enfoque sobre el diseño de procesadores, a los que se identificó como procesadores RISC.

* Los primeros RISC tenían, entre otras, las siguientes características relevantes:
* Repertorio de instrucciones **reducido y básico.**
* Formato de instrucción **fijo** y sencillo (todas las instrucciones tienen el mismo tamaño).
* Unidad de control “cableada” (no microprogramada).
* Modos de direccionamiento sencillos.
* Diseño del cauce para ejecutar 1 instrucción por ciclo.
* Máquina tipo **LOAD-STORE**: acceso a memoria solo con instrucciones de movimiento de datos. Instrucciones aritméticas y lógicas solo entre registros.
* Uso intensivo de registros con optimización por software.
* Operaciones registro a registro.
* Típicamente máquinas tipo Harvard.
* Mayor tiempo/esfuerzo de compilación.
* Comparación directa **RISC vs: CISC:**
* **Repertorio de instrucciones:** los CISC tienen muchas más instrucciones que los RISC.
* **Tamaño de instrucción:** en los CISC es muy variable, mientras que en los RISC es fija.
* **Modos de direccionamiento:** los CISC tiene una buena variedad de modos de direccionamiento, mientras que los RISC son mínimos.
* **Banco de registros:** en los RISC son amplios, y más limitados en los CISC.
* **Memoria de control:** típicamente microprogramada en los CISC, lógica “cableada” en los RISC.
* Sin embargo, a pesar de que se han realizado numerosas mediciones comparativas entre ambos, hay varios problemas:
* No existe un par de máquinas RISC y CISC directamente comparables.
* No hay un conjunto de programas de prueba definitivo.
* En los análisis, es muy difícil separar los efectos del hardware de los del compilador.
* La mayoría de las máquinas son una mezcla de ambas.

Por lo que se deduce que:

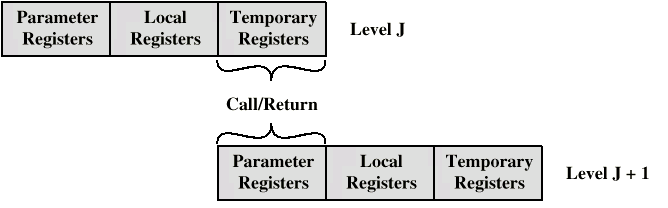
* No existe una marcada diferencia de performance entre uno y otro
* No está clara la barrera que separa uno u otro estilo.
* Un aspecto esencial de los procesadores RISC es disponer de un banco de registros amplio, para reducir los accesos a memoria, y simplificar las instrucciones. La optimización del banco de registros se puede hacer de 2 maneras:
* **Por Hardware🡪** agregando más registros.
* Ventajas:
* Reducción de accesos a memoria.
* Disponibilidad para más variables escalares locales.
* Desventajas:
* Más bits en la instrucción para identificar el registro.
* En llamadas y retornos de subrutina hay que considerar que se requiere tiempo para pasar parámetros, asignar espacio para las variables locales, y devolver los resultados.
* Por Software🡪 optimizando la asignación de registros a las variables que se usen más en un período de tiempo dado.

# Llamadas y retornos de subrutinas:

* ¿De qué depende el tiempo que tarda una llamada a una subrutina o procedimiento?
* Del número de parámetros tratados.
* Del nivel de anidamiento.
* En general, cada subrutina necesita, para operar, *3 tipos de datos*, y para cada dato, un tipo de registro:
* **Datos de entrada** (argumentos de entrada) 🡪 **registros de parámetros**.
* **Datos propios** (datos/variables locales) 🡪 **registros locales**.
* Datos de salida (para pasar argumentos)🡪 registros temporales.

Además, todas las subrutinas requieren de un conjunto de variables (globales):

* **Datos globales** (vistos por todas las subrutinas) 🡪 **registros globales**.
* Si cada subrutina administra su banco de registros (que le pertenece), la cantidad real de registros de cada subrutina se limita a un número no excesivamente grande, y, por lo tanto, requiere pocos bits para identificarlo.
* *Excepto los globales*, cada vez que se invoca una subrutina *se requiere un nuevo banco de registros.*
* Los problemas que aparecen con el uso de subrutinas son:
* Cada vez que una subrutina invoca otra subrutina (anidamiento de subrutinas) debe pasarle los argumentos, lo que produce un gasto de tiempo considerable.
* Cuantos más registros tenga la subrutina, más tiempo se consume en pasar la información.
* Cuantos más registros necesite localmente, más tiempo se requiere en reservarlos.
* Para solucionar los problemas en el uso de procedimientos y pasaje de parámetros se puede usar una estrategia conocida como **ventana de registros**:

1. Asignar a cada subrutina un banco de registros propio:
2. Superponer los registros donde recibe los parámetros una subrutina con los registros donde pasa los argumentos la subrutina que la llama. Como se ve en la imagen🡪

* Para el manejo de variables globales (es decir compartidas por todas las subrutinas) se pueden usar 2 soluciones:
* El compilador asigne posiciones de memoria a las variables: es algo ineficiente para variables globales a las que se accede frecuentemente porque requiere accesos a memoria que son lentos.
* Incorporar en el procesador un conjunto de registros para variables globales accesible en todos los niveles. Este banco de registros no está mostrado en la ventana de registros.
* **Ventana de registros vs. Memoria cache**:

|  |  |
| --- | --- |
| **Banco de registros amplio** | **Cache** |
| * Todos los datos son escalares y locales * Acceso individual a variables * Variables globales asignadas por el compilador * Salvaguarda/restauración basadas en la profundidad de anidamiento * Direccionamiento de registro | * Datos escalares locales recientemente usados * Acceso a bloques de memoria * Variables locales y globales usadas recientemente * Salvaguarda/restauración basadas en el algoritmo de reemplazo * Direccionamiento de memoria |

# Clase 7:

En el diseño de la Memoria existe un compromiso entre:

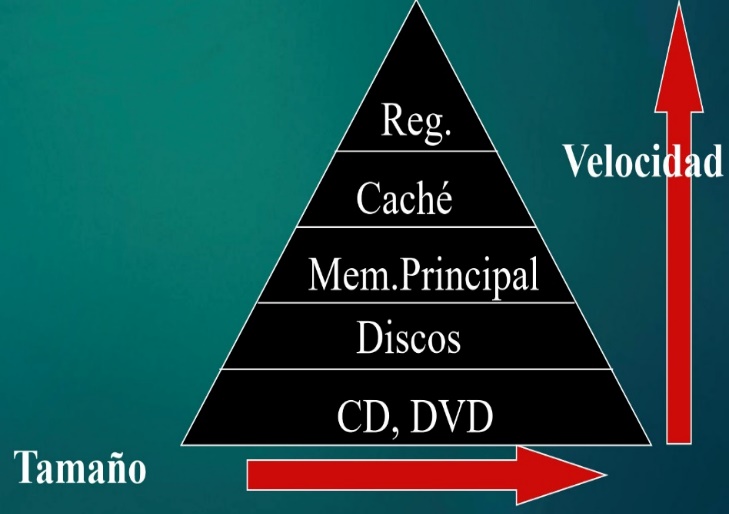
* Capacidad
* Velocidad
* Costo

Una Memoria ideal sería aquella que:

* Es infinitamente grande
* El tiempo de acceso sumamente pequeño (casi 0)
* El costo relativamente bajo

# Jerarquía de memoria:

En una computadora típica hay distintos tipos de memorias, desde las rápidas y caras (Ej: registros) hasta las lentas y baratas (Ej: discos).

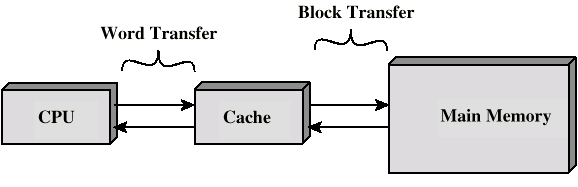
* En las computadoras actuales los diferentes tipos de memorias actúan coordinadamente y no separadas.
* Esa interacción permite un comportamiento global equivalente al que tendría con una memoria única, grande y rápida.
* La forma en que se organizan coordinadamente los distintos tipos de memoria se conoce con el nombre de **Jerarquía de Memoria.**
* La Jerarquía de memoria se puede pensar como una pirámide de múltiples capas o niveles, de diferentes tamaños y velocidades.

**\*\***En CD, DVD (almacenamiento secundario) también se incluye la memoria virtual (como copias de seguridad en la nube)

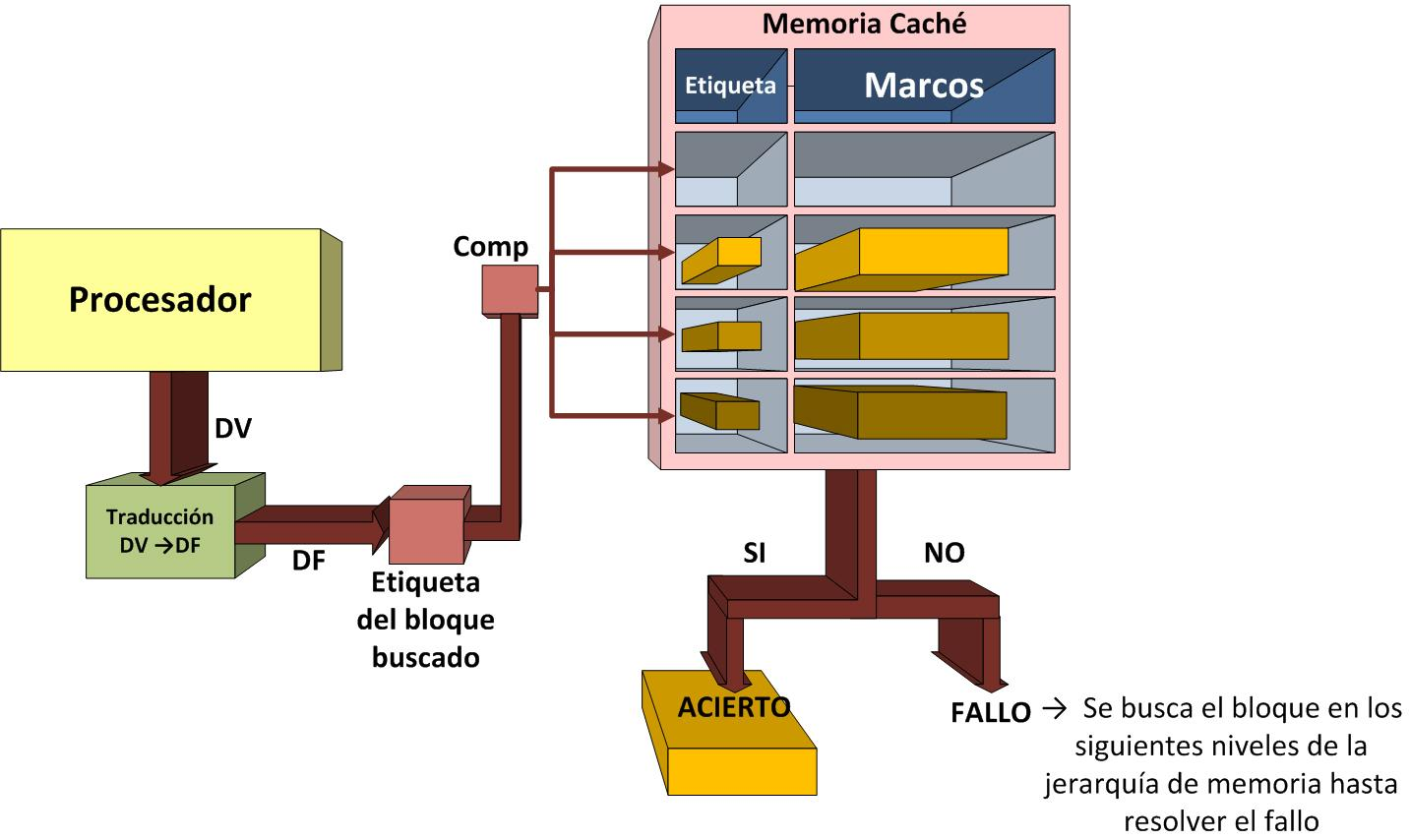
* Los **principales objetivos** de una Jerarquía de memoria son:
* **Maximizar tamaño:** idealmente disponer de una “capacidad ilimitada”, equiparada al tamaño del nivel más grande.
* **Optimizar velocidad:** simular que se dispone de un banco de memoria “ultrarápida”.
* **Minimizar el costo total:** implementar una memoria a un costo cercano al del nivel más lento.
* Para que la memoria se comporte como una jerarquía (integrada) *debe* *cumplir* las siguientes propiedades:
* **Inclusión**: los datos almacenados en un nivel han de estar también almacenados en los niveles inferiores a él.
* **Coherencia**: las copias de la misma información en los distintos niveles deben contener los mismos valores.
* El manejo de la jerarquía de memoria *es administrado* por:
* **Nivel registros**: el compilador.
* **Cache**: la administración se hace por hardware
* **Memoria principal**: la administración la pueden hacer:
* Hardware
* Sistema operativo
* Programador (archivos)

# Memoria Cache:

La memoria Cache es una memoria pequeña y muy rápida, que se ubica entre la memoria principal y la CPU:

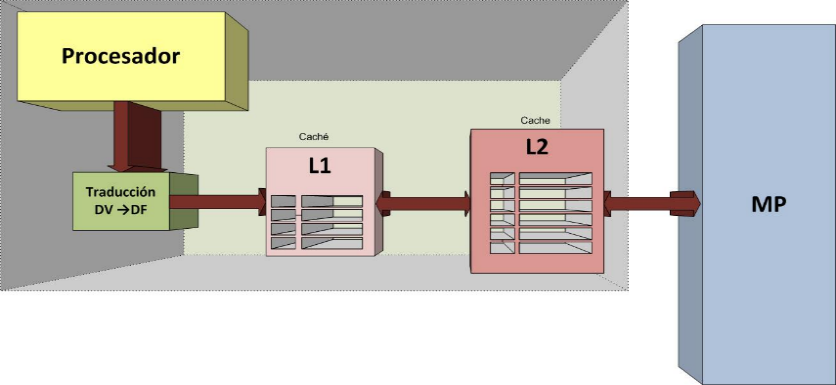
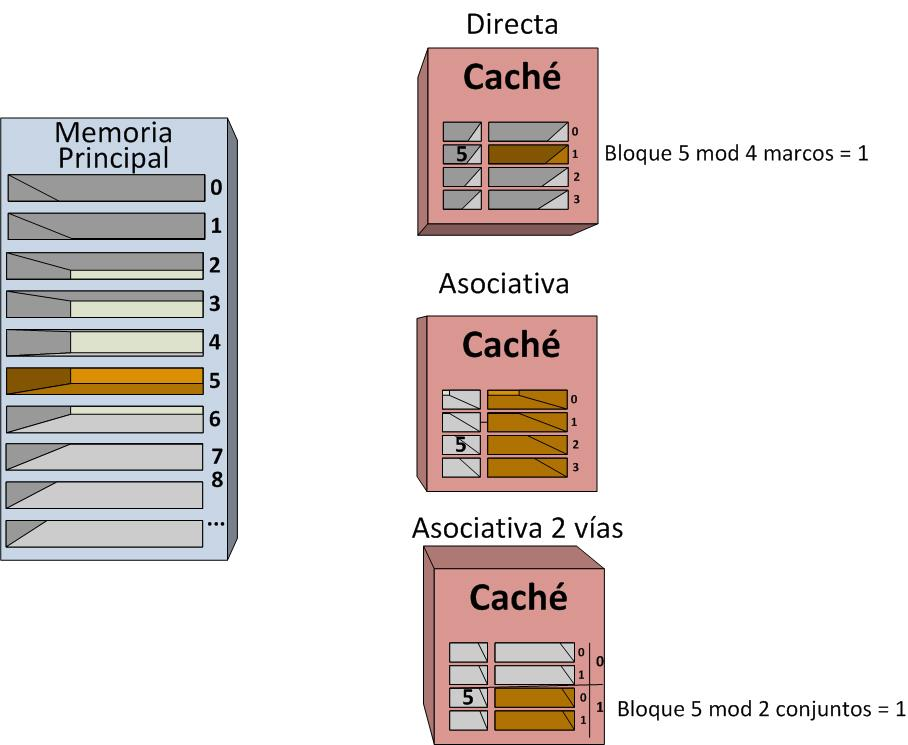
* Puede localizarse en un chip separado o dentro de la CPU, o en ambos lugares.
* Contiene algunos **sectores** (bloques) de la memoria principal.
* La información contenida en la Caché se organiza en bloques (o ranuras) de **longitud fija** (Ej: 8, 16, 32 bytes, etc.).
* En los bloques de la cache se copian algunos bloques (**de idéntico tamaño**) de la memoria principal.
* La cantidad de bloques copiados depende del tamaño de la memoria cache y del bloque.
* Cada ranura tiene asociada una etiqueta para identificar el bloque de la memoria que tiene copiado.
* El *conjunto de etiquetas forma el directorio* de la cache.
* **Funcionamiento**🡪 Cuando la CPU necesita un dato, genera 1 dirección de memoria. La cache “intercepta” esa dirección y determina si tiene ese dato. Pueden ocurrir 2 situaciones:
* **Acierto (hit)**🡪 si lo tiene, se lo envía a la CPU (a la velocidad de la cache).
* **Fallo (miss)**🡪 Si no lo tiene, se trae el bloque que contiene esa dirección desde la memoria principal, y la cache entrega el dato requerido a la CPU.
* Cuando ocurre un fallo, el bloque que contiene la palabra accedida se copia de la memoria principal a una línea de caché.
* Los fallos de caché se gestionan mediante hardware y causan que el procesador se detenga hasta que el dato esté disponible. Esta acción requiere un tiempo determinado.
* El tiempo para servir un fallo depende de la latencia y ancho de banda de la memoria principal.
* La latencia es el tiempo necesario para completar un acceso a memoria (depende de la memoria).
* El ancho de banda es la velocidad a la cual se puede transferir el dato (depende de la velocidad del bus).

En la siguiente imagen se ve el funcionamiento gráficamente:



* Si el dato que busca la CPU está en la cache, *la velocidad del acceso depende del tiempo de acceso de la cache* (relativamente muy corto).
* Si el dato no está en la cache, *la velocidad del acceso depende del tiempo de acceso de la memoria principal* (relativamente largo).
* Así, la eficiencia del uso de la cache depende de la cantidad de veces que “acierta” a la cache.
* La cantidad de veces que se acierta (la **tasa de aciertos**) no necesariamente tiene que ser proporcional al tamaño de la cache, sino de las veces que se cumplen estos 2 principios de carácter empírico:
* **Principio de localidad temporal de las referencias**🡪 Es altamente probable que los elementos de memoria referenciados recientemente (datos o instrucciones) vuelvan a ser referenciados en el corto tiempo.
* **Principio de localidad espacial de las referencias**🡪 Es altamente probable que los próximos elementos de memoria referenciados estén en las proximidades de los últimos referenciados.
* En un procesador con segmentación del cauce, se dispone de un ciclo de reloj para el acceso a memoria. Ese tiempo debe ser suficiente para el acceso a la memoria cache. *En caso de fallo* el acceso a la memoria principal *requiere de* *varios ciclos extra*.
* El **tiempo de acceso promedio de la CPU** es el promedio del tiempo que tarda en obtener los datos buscados en memoria, compuestos por accesos a la caché y accesos a la memoria principal:

**Tiempo de acceso CPU = (1 - Tasa de fallos) \* Tiempo de acceso Cache + Tasa de fallos \* Penalización por fallo**

* Para mejorar las prestaciones hay que reducir el tiempo de acceso de la memoria cache, tasa de fallos y penalización por fallo.
* Para el **diseño** de la caché hay que tener en cuenta varias consideraciones.
* **Tamaño de la memoria caché:**
* Debe ser suficientemente grande para contener la mayor cantidad posible de información.
* Pero no demasiado grande porque el tamaño tiene impacto en la velocidad y en el costo.
* **Tamaño del bloque o ranura:**
* El tamaño del bloque es muy importante en la tasa de aciertos.
* El bloque debe ser suficientemente grande para aprovechar al máximo las referencias cercanas. Al aumentar el tamaño mejora la tasa de aciertos, hasta un cierto punto (después del cual ya no produce efecto).
* Por otra parte, al aumentar el tamaño del bloque hay menos bloques de la memoria principal en la caché, lo que tiende a aumentar la tasa de fallos, y la penalización por fallos porque son más palabras a transferir entre la memoria principal y la cache.
* **Costo:**
* El costo crece fuertemente con el tamaño de la memoria.
* Y este costo es significativamente grande cuando la caché está incluida en el chip del procesador.
* **Niveles de la cache:**
* La memoria caché pude ser una sola (1 nivel) o estar dividida en varias unidades (múltiples niveles). Los múltiples niveles de caché por lo general tienen distintos tamaños.
* En esta figura, la caché está dividida en 2 niveles, el **L1** muy pequeño y rápido, y el **L2** un poco más grande y lento 🡪
* **Separación de la caché de instrucciones y operandos:**
* El mecanismo de acceso a las instrucciones es distinto al del acceso de los datos. Es por ello que la estrategia para obtener una alta tasa de aciertos también es distinta.
* Teniendo en cuenta esto, es posible mejorar la tasa de aciertos general si la caché se divide en una caché de instrucciones y una de datos, con distintas características.
* Por ejemplo, ambas caches pueden tener distintos tamaños y políticas de acceso y reemplazo.
* Además, para el **diseño** de la cache también se deben definir:
* **Organización**: tamaño de la caché, cantidad y tamaño de bloques
* **Política de asignación**: son las funciones (de mapeo) que definen la forma en que se van a asignar los bloques de la memoria principal en la memoria caché. Las políticas más empleadas son:
* **Correspondencia totalmente asociativa:** Un bloque puede almacenarse libremente en cualquier lugar de la caché.
* La implementación del directorio de la caché es compleja y costosa.
* Esta política permite una libertad absoluta para la asignación y reemplazo de los bloques.
* **Correspondencia directa:** Un bloque sólo puede estar almacenado en una única línea (fija) de la caché.
* Esta política de asignación es muy simple de implementar y poco costosa.
* Tiene un rendimiento aceptable, aunque a veces puede ser malo. Por ejemplo, si un programa accede a dos bloques que se corresponden a la misma línea de forma repetida, los fallos de cache serán muy grandes.
* **Correspondencia asociativa por conjuntos:** Un bloque puede almacenarse en bloques determinados (dentro de un conjunto restringido de lugares) en la caché. La cache se divide en un número de conjuntos N.
* Cada conjunto contiene un número de líneas o ranuras.
* Un bloque determinado corresponderá a alguna línea o ranura de un conjunto determinado.
* En general, la función asociativa por conjuntos combina lo mejor de las otras correspondencias.
* **Política de reemplazo**: Cuando hay un fallo de acceso a la memoria cache, se debe traer un bloque desde la memoria principal y almacenar un bloque existente en la cache.
* El lugar a donde va a ser ubicado el bloque a traer desde la memoria principal requiere reemplazar un bloque existente.
* Las diferentes políticas de reemplazo de los bloques son:
* **Correspondencia directa:**
* En la función de mapeado directo, la asignación de bloques de la memoria principal a bloques de la cache es fija, no hay elección.
* Sólo hay una posible línea para cada bloque.
* Por lo tanto, si se requiere traer un nuevo bloque desde la memoria principal, reemplazará el que está usando esa ranura actualmente.
* **Correspondencia asociativa y asociativa por conjuntos:**
* Para las asignaciones asociativa y asociativa por conjuntos, hay varios algoritmos de reemplazo:
* **LRU (menos usado recientemente)**🡪 Se reemplaza el bloque que lleva más tiempo sin utilizarse.
* **FIFO (1ro en entrar - 1ro en salir)**🡪 Se reemplaza el bloque que entró antes en la caché.
* **LFU (menos frecuentemente usado)**🡪 Se sustituye aquella línea que ha experimentado menos referencias.
* **Aleatorio**🡪 Se sustituye una línea al azar.
* Todos los algoritmos deben implementarse en hardware (por razones de velocidad)
* **Política de escritura**: mecanismos de escritura en memoria principal.
* Tanto en un acierto como en un fallo de la memoria cache, se debe evitar inconsistencia de información entre las memorias principal y cache, durante los procesos de escrituras.
* Es decir que, aun escribiéndose el dato en la cache, el correspondiente bloque de la memoria principal debe ser actualizado en algún momento.
* Las políticas de escritura son distintas en aciertos que en fallos.
* **Políticas de escritura en aciertos:**
* **Write-through (Escritura inmediata):**
* Se actualizan simultáneamente la posición de la caché y de la memoria principal.
* Aumenta el tráfico con la memoria principal.
* Puede haber retrasos durante múltiples escrituras.
* **Write-back (Post-escritura):**
* La información sólo se actualiza en la caché y se escribe la memoria cuando se reemplaza el bloque. El bloque requiere de un bit de “sucio” para indicar cuando se lo escribió.
* Como la memoria principal se actualiza en el reemplazo, puede contener información errónea en algún momento.
* **Políticas de escritura en fallos:**
* **No-write allocate:**
* Se escribe directamente en la memoria principal. La caché se usa solo en las lecturas. El bloque no se lleva a la memoria caché mientras no se lo tenga que leer.
* Habitual con write-through.
* **Write allocate:**
* El bloque requerido primero se copia en la cache, y luego se escribe (en la cache).
* Habitual con write-back.

# Clase 8:

Debido a las limitaciones de los procesadores para usar segmentación, surgen 2 técnicas para mejorar la performance:

Procesadores supersegmentados🡪 En la ejecución supersegmentada de instrucciones cada ciclo se divide en fracciones más pequeñas, en las que se inician nuevas instrucciones.

* Como muchas operaciones no necesitan todo un ciclo de reloj, se puede hacer más de una tarea en ese ciclo, subdividiendo el ciclo de reloj en sub-intervalos. Por ejemplo, un procesador supersegmentado de grado 2 acepta una nueva instrucción en cada semiciclo de reloj.
* El tiempo para las instrucciones individuales no varía, pero aumenta el grado del paralelismo de la máquina temporal.
* El pipeline se hace más profundo, pero está limitado por la tecnología (es decir, por la frecuencia máxima).

Procesadores superescalares🡪 Se pueden completar 2 o más instancias de cada etapa de una instrucción simultáneamente.

* Para poder iniciar/ejecutar 2 o más instrucciones simultáneamente, se requiere la *duplicación* de algunas o todas las partes de la CPU/ALU, por ejemplo:
* Captación de múltiples instrucciones al mismo tiempo.
* Ejecución (sumas y multiplicaciones) simultánea.
* Ejecución de carga/almacenamiento, mientras se lleva a cabo una operación en ALU.
* El grado de **paralelismo** y la aceleración de la máquina aumentan, ya que se ejecutan más instrucciones en paralelo.
* La ejecución superescalar involucra:
* Diferentes estrategias de captación simultánea de múltiples instrucciones.
* Lógica para determinar dependencias verdaderas entre valores de registros y mecanismos para comunicar esos valores.
* Mecanismos para iniciar o emitir múltiples instrucciones en paralelo.
* Recursos para la ejecución en paralelo de múltiples instrucciones.
* Mecanismos para entregar el estado del procesador en un orden correcto.

Paralelismo🡪 es la capacidad para ejecutar varias tareas en el mismo intervalo de tiempo (en “paralelo”). Hay 2 tipos:

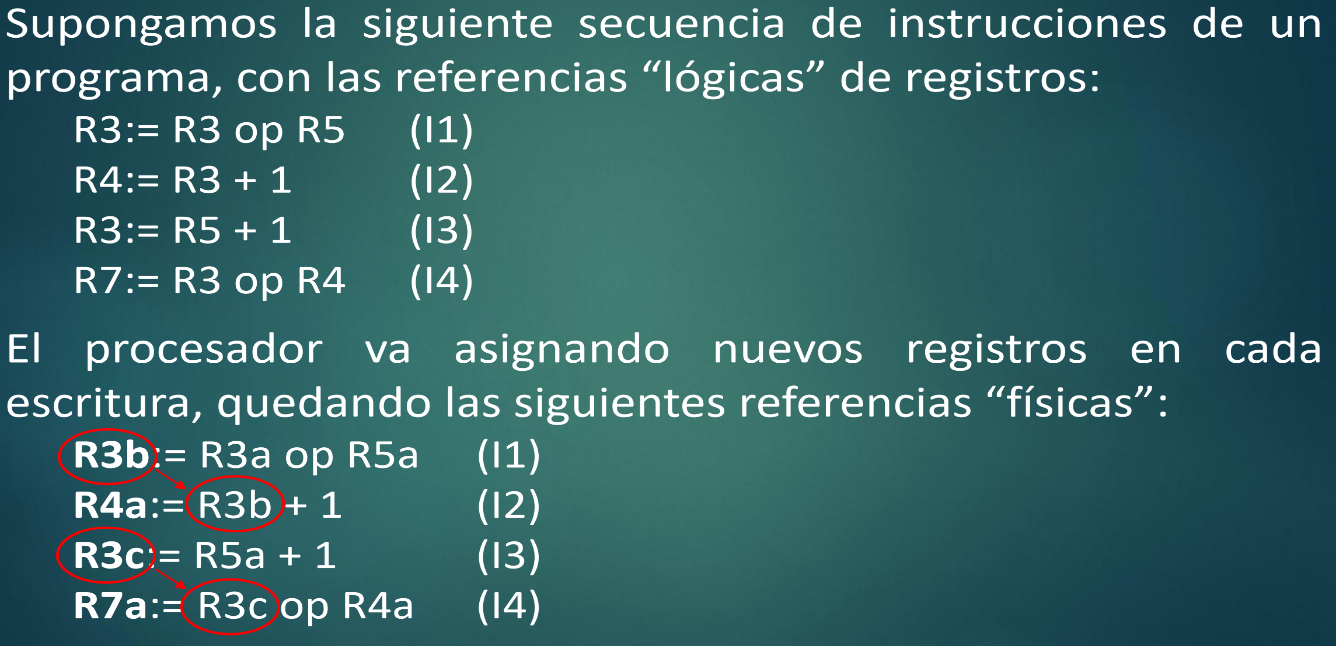
* **Paralelismo a nivel de instrucciones/programa**🡪 se refiere a las posibilidades que tiene un programa para ejecutar instrucciones en paralelo.
* Se mide usando un parámetro llamado **IPL**, que es la cantidad (promedio) de instrucciones que pueden ejecutarse en paralelo.
* Las instrucciones pueden ejecutarse en paralelo si son independientes.
* **Paralelismo a nivel de máquina**🡪 es la capacidad que tiene una máquina para encontrar y ejecutar tareas en paralelo.
* Se mide usando un parámetro llamado **MPL**, que es la capacidad que tiene una máquina para aprovechar el paralelismo de un programa.
* El MPL es función de:
* Número de instrucciones captadas por ciclo.
* Número de unidades funcionales.
* Mecanismos para localizar y ejecutar instrucciones independientes.

Tratamiento de las instrucciones🡪 En los procesadores superescalares el objetivo fundamental es localizar instrucciones que puedan ser introducidas al pipeline y ejecutadas. En este proceso de detección, es necesario distinguir:

* El orden en que se captan las instrucciones.
* El orden en que se ejecutan las instrucciones.
* El orden en que las instrucciones actualizan los registros y las posiciones de memoria.
* Cuanto más sofisticado es el procesador, menos restricciones impone a estos ordenamientos. Incluso puede alterar cualquier ordenamiento, respecto del estrictamente secuencial. La única condición es que el resultado debe ser correcto.
* Las **políticas de emisión de instrucciones** son protocolos usados para el envío de las instrucciones a las unidades funcionales, es decir, **definen** **la** **forma** **en que se** **captan**, **ejecutan** y **finalizan** (escriben los resultados) **las** **instrucciones**. En función del orden en que se captan, ejecutan y terminan las instrucciones, existen 3 políticas:
* **Emisión y finalización ordenada**🡪 Las instrucciones se emiten exactamente como están en el programa (secuencialmente), y se escriben los resultados en el mismo orden.
* **Emisión ordenada y finalización desordenada**🡪 Las instrucciones se emiten exactamente como están en el programa, pero los resultados se escriben en cualquier orden.
* Se ganan algunos ciclos de reloj.
* Al permitir la finalización desordenada, aparecen nuevas fuentes de conflictos por dependencia de datos.
* Por tal motivo, la lógica de emisión de instrucciones es más compleja que la empleada en máquinas con política de finalización ordenada.
* Con la política de emisión ordenada, el procesador no analiza si hay nuevas instrucciones que pueden emitirse sin producir conflicto.
* **Emisión y finalización desordenada**🡪 Se utiliza una política de emisión desordenada para permitir que el procesador busque más allá del punto de conflicto por nuevas instrucciones.
* Esta requiere separar, mediante un buffer llamado **Ventana de instrucciones**, las unidades de decodificación y las de ejecución.
* Cada instrucción decodificada se coloca en un buffer intermedio, desde donde se emiten las que no presentan conflictos. Mientras no se llene el buffer, nuevas instrucciones son decodificadas y colocadas para su emisión a las unidades de ejecución. Las instrucciones emitidas no deben tener conflictos por recursos ni por dependencias de datos.
* Al permitir finalización y/o emisión desordenada surgen los nuevos problemas de dependencia de datos (WAW y WAR).
* Se sabe que estos conflictos pueden resolverse con detenciones en la etapa del cauce, pero esa solución produce grandes pérdidas de tiempo.
* Dado que, en definitiva, es un conflicto de recursos (los registros), se puede resolver duplicando los recursos. Dicha técnica se conoce como ***Renombrado de Registros***.

# Renombrado de Registros:

* Los registros en las instrucciones del programa, son registros “lógicos”. Los registros físicos (es decir, los reales) los asigna el procesador.
* Cada vez que se guarda un dato en un registro el procesador le asigna un nuevo registro físico. Las instrucciones siguientes que referencien a ese registro, deben renombrarse para referenciar el registro físico que tiene el dato correcto.



* De esta manera se eliminan las dependencias WAW y WAR, quedando únicamente la dependencia verdadera RAW.
* **Estudios** sobre las ejecuciones superescalares han concluido:
* **En máquinas sin renombrado de registros se observa que:**
* **Duplicar** las unidades funcionales de acceso a la memoria y las ALU **tiene poco impacto** en la aceleración obtenida.
* Disponer de una Ventana de instrucciones más grande **no modifica significativamente** los resultados.
* **En máquinas con renombrado de registros se observa que:**
* **Duplicar** las ALU y también las unidades de acceso a la memoria **mejoran** **fuertemente** la aceleración obtenida.
* El tamaño de la Ventana de instrucciones **tiene un impacto significativo** cuando cambia de 8 a 16 instrucciones.
* Esto debido a que:
* En las máquinas **sin renombrado de registros**, todos los conflictos por las dependencias de datos se resuelven mediante detenciones del cauce.
* En las máquinas **con renombrado de registros** se eliminan los conflictos por dependencias de salida y antidependencia (WAW, WAR) quedando únicamente la dependencia directa o verdadera (RAW).

Interrupciones🡪 **En los procesadores superescalares, el tratamiento de interrupciones** *internas* o *excepciones* y *externas* **es más complicado** que en los procesadores convencionales.

* La razón es que en el momento en que se produce una interrupción hay varias instrucciones en ejecución (y desordenadas).
* **Interrupciones internas o excepciones**🡪 Existen 2 estrategias para su tratamiento:
* **Excepciones precisas**🡪 las instrucciones que preceden a la que produjo la excepción se completan, y las que le suceden se reinician. Es decir, el comportamiento es “idéntico” al que tendría la misma computadora no segmentada.
* Para garantizar un estado consistente (preciso) se requiere:
* Instrucciones anteriores terminan correctamente.
* La que origina la excepción y siguientes se abortan.
* Tras la rutina de tratamiento se comienza por la que originó la excepción.
* **Excepciones imprecisas**🡪 no se respetan las condiciones exactas de la máquina no segmentada.
* **Interrupciones externas**🡪 Pueden tener distinto grado de precisión, dado que no necesariamente tienen que ver con las instrucciones en ejecución:
* **Interrupciones imprecisas**: se completa lo que estaba en ejecución.
* **Interrupciones “algo” imprecisas**: el software (la rutina de servicio de la interrupción) resuelve algunas inconsistencias.
* **Interrupciones precisas**: solo se completan las instrucciones previas a la interrupción, para lo cual se requiere implementar la escritura con buffer de salida.

# Conclusiones:

En general se tiene:

* En los **sistemas con** **finalización desordenada**:
* Excepciones imprecisas
* Mayor performance
* En los **sistemas con finalización ordenada** (con buffer de reordenamiento):
* Excepciones precisas
* Menor performance

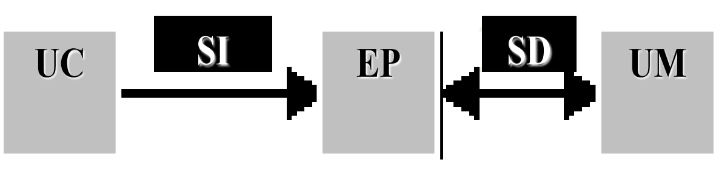
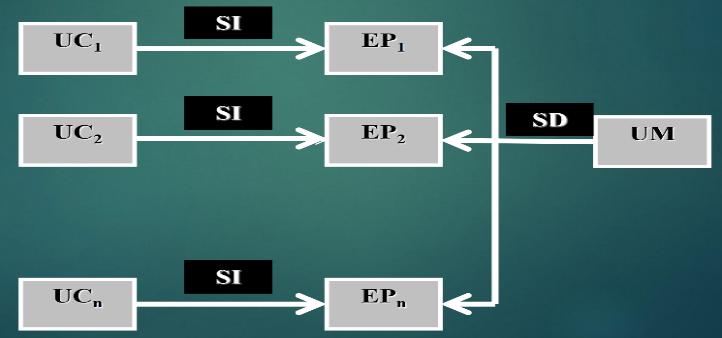
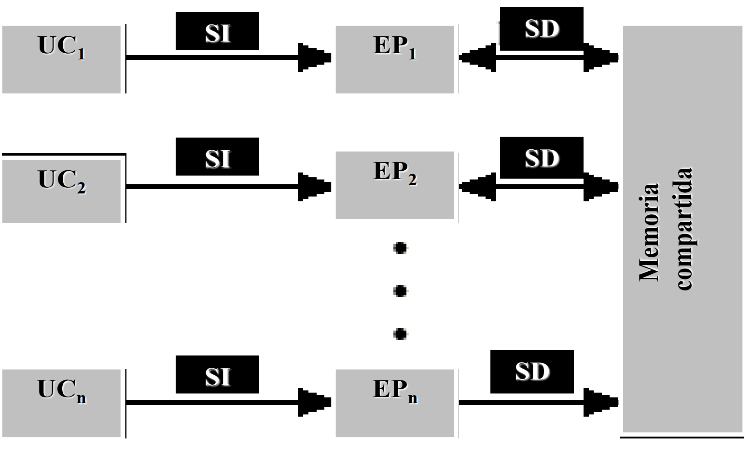
# Clase 9:

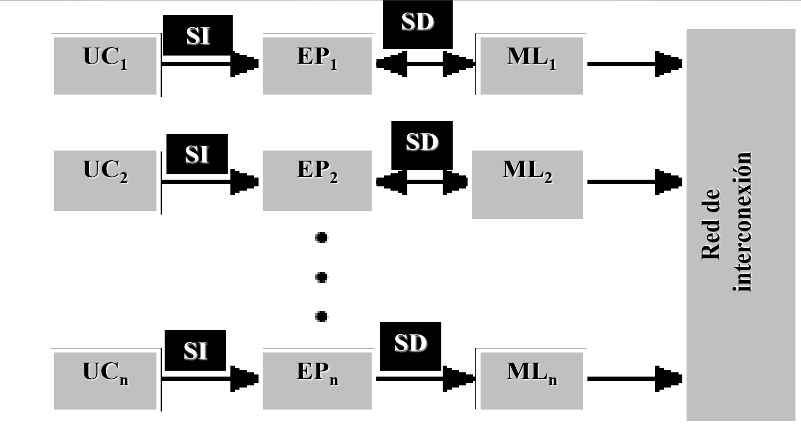
Procesamiento paralelo🡪 La demanda de máquinas de mayor rendimiento es una exigencia que surgió desde la aparición de las primeras computadoras, y continúa en forma permanente.

* Existen **2** caminos para aumentar la capacidad de procesamiento:
* **Mejorar el rendimiento de una máquina con un solo procesador**. Requiere:
* Explotar el **paralelismo a nivel instrucción (ILP).**
* Optimizar la detección del paralelismo, a nivel de hardware (**MPL**).
* **Disponer de sistemas con varios procesadores**. Requiere:
* Explotar el **paralelismo a nivel proceso**.
* Detección del paralelismo a nivel de sistema operativo, compilador, o programación.

Taxonomía de Flynn🡪 Una forma de clasificar los sistemas de cómputo (entre otras) se basa en determinar las cantidades de flujos de instrucciones y datos que se pueden transferir simultáneamente. En base a este concepto se tienen 4 tipos básicos de máquinas:

\*\*Para las abreviaciones usadas en las imágenes, [acá](#_Índice_de_abreviaciones:) hay un índice.

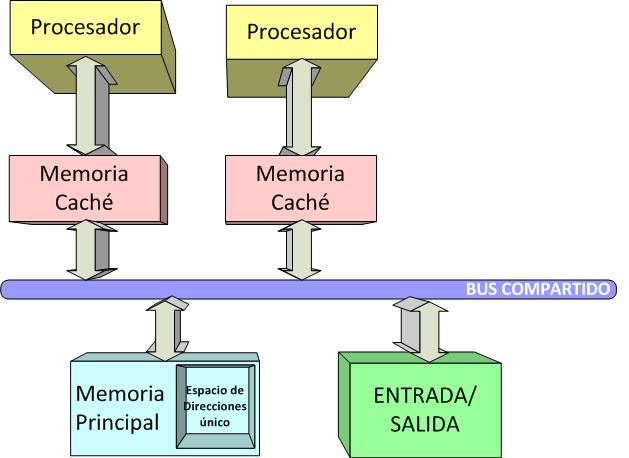
* **SISD**: una secuencia de instrucciones, una secuencia de datos.
* Es un sistema básicamente compuesto por:
* Una Unidad de Control (UC)
* Un elemento de proceso (EP)
* Una Unidad de Memoria (UM).
* La UC lee una única secuencia de instrucciones (SI), que el EP resuelve sobre una única secuencia de datos (SD) proveniente de la UM.
* **Ejemplo**: Computadoras *monoprocesador*.
* **SIMD**: una secuencia de instrucciones, múltiples secuencias de datos.
* Es un sistema compuesto por:
* Una Unidad de Control (UC).
* Una matriz de elementos de proceso (EP).
* Una Unidad de memoria local (ML) por cada EP.
* La UC interpreta una única secuencia de instrucciones (SI) que múltiples EP resuelven simultáneamente cada uno sobre una secuencia de datos (SD) proveniente de su propia ML.
* **Ejemplo**: Procesadores *vectoriales* y *matriciales*.
* Se requiere disponer de:
* Un conjunto ampliado de instrucciones para permitir manejar operaciones vectoriales.
* Se deben agregar instrucciones para transferir datos entre EPs, que es un atributo típico de un “lenguaje paralelo”.
* **MISD**: múltiples secuencias de instrucciones, una secuencia de datos.
* Es un sistema compuesto por:
* Una matriz de elementos de proceso (EP) cada uno con su propia Unidad de Control (UC).
* Una Unidad de memoria (UM) para todos los EP.
* Cada UC interpreta una secuencia de instrucciones (SI) que cada EP resuelve simultáneamente sobre una única secuencia de datos (SD) proveniente de una sola memoria.
* Cada procesador ejecuta una secuencia de instrucciones diferente.
* **Ejemplo**: es un *modelo* *teórico*, no hay registros de máquinas de este tipo.
* **MIMD**: múltiples secuencias de instrucciones y múltiples secuencias de datos.
* Es un sistema compuesto por:
* Múltiples procesadores (UC)
* Múltiples secuencias de instrucciones (MI)
* Múltiples secuencias de datos (MD).
* Se pueden dividir según la organización de la Memoria (que es también la forma de comunicarse) en:
* **MIMD de Memoria compartida**, con 2 variantes:
* *SMP* (multiprocesadores simétricos)
* *Sistemas* *NUMA*



* **MIMD de Memoria distribuida**, comúnmente llamados *Clusters*.

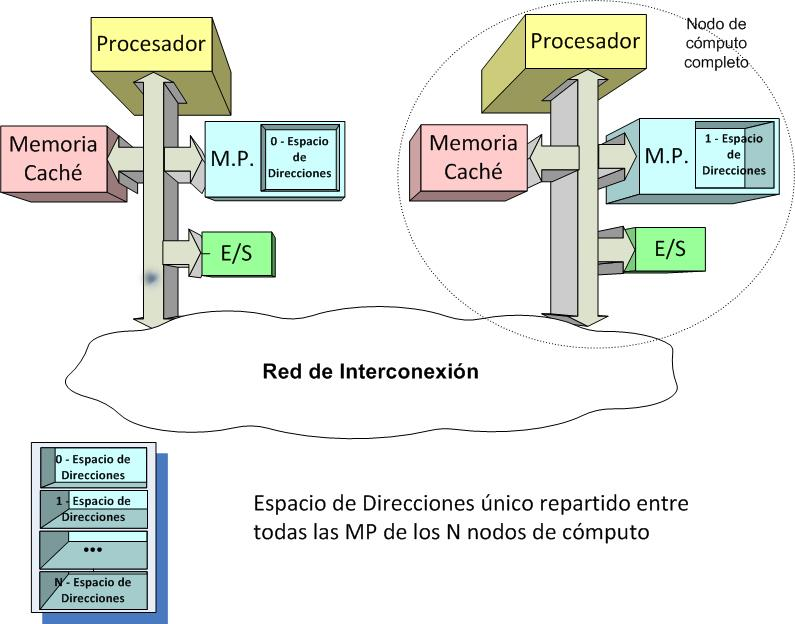
# SMP vs. NUMA vs. Clusters:

**SMP (Symmetric Multi-Processing o Multiprocesadores Simetricos)**🡪 Es un sistema con las siguientes características principales:

* Dos o más procesadores idénticos (o muy similares) de capacidades comparables (homogéneos).
* Memoria principal y E/S única (compartida) por todos los procesadores.
* Interconexión mediante un bus u otro tipo de medio similar (“fuertemente acoplados”).
* Igual tiempo de acceso a la memoria para todos los procesadores. Por eso se los identifica como del tipo “UMA”: Uniform Memory Access.
* Todos los procesadores pueden desempeñar las mismas funciones.
* Sistema operativo integrado, que proporciona la interacción entre los procesadores y sus programas.
* **Ventajas**:
* **Mayores prestaciones**: en general tienen buenos resultados, si las tareas pueden organizarse en paralelo.
* **Buena disponibilidad**: un fallo en un procesador no detiene la operación del sistema, dado que todos los procesadores pueden hacer las mismas tareas.
* **Crecimiento incremental:** pueden aumentarse las prestaciones añadiendo más procesadores, pero hay restricciones.
* **Escalado**: normalmente limitado, en función de la cantidad de procesadores.
* **Desventajas**:
* Conflictos por el bus compartido, la coherencia y consistencia de los datos, y la sincronización de tareas entre procesadores.
* Las prestaciones están limitadas por el tiempo de ciclo del bus.
* Cada procesador está equipado con una memoria cache, que reduce los accesos a memoria y mejora las prestaciones. Pero al disponer una caché en cada procesador, se pueden producir problemas de coherencia de cache (datos que pueden estar en más de una cache). Por razones de velocidad, este problema debe ser resuelto por el hardware.

\*\*Esquema de modelo SMP

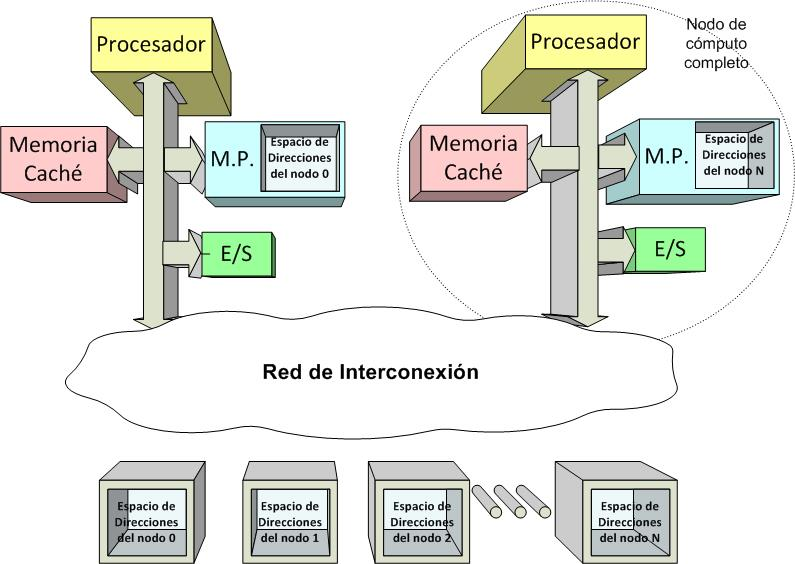
**NUMA (Non Uniform Memory Access - Multiprocesadores con memoria compartida y distribuida)**🡪 Es un sistema con las siguientes características principales:

* Dos o más procesadores idénticos (o muy similares) de capacidades comparables (homogéneos) formando “nodos”.
* Cada nodo es un procesador completo con su propia memoria local y E/S. Los nodos tienen características muy similares.
* Los nodos están interconectados mediante una red de interconexión.
* El espacio de direcciones (memoria lógica) es común a todos los procesadores, por eso es un sistema de memoria compartida. Pero el tiempo de acceso al espacio de memoria es distinto para el acceso a la memoria local que para el acceso a las memorias de los otros nodos.
* Por eso se los llama NUMA: Non Uniform Memory Access.
* Todos los procesadores pueden desempeñar las mismas funciones.
* El Sistema operativo está integrado, y proporciona la interacción entre los procesadores y sus programas.

\*\*Esquema de modelo NUMA

**Cluster (Multiprocesadores de memoria distribuida)**🡪 Es un sistema con las siguientes características principales:

* Compuesto por 2 o más nodos.
* Cada nodo es un procesador completo con su propia memoria local y E/S. Es decir, cada nodo es una computadora. Los nodos pueden tener características similares (“homogéneos”) o distintas (“heterogéneos”).
* Interconectados mediante una red de interconexión. Debido al tipo de red y protocolo de comunicación se los considera del tipo “levemente acoplados”.
* Como los espacios de direcciones son independientes es un sistema de Memoria distribuida.
* Todos los procesadores pueden desempeñar las mismas funciones.
* El Sistema operativo está integrado, y proporciona la interacción entre los procesadores y sus programas.
* La comunicación entre procesos es en base a mensajes (a resolver por el programa).
* Básicamente, son computadoras completas, interconectadas, que trabajan conjuntamente como un único recurso. Para las tareas en ejecución se comportan como si fuera una única máquina.
* En general, presentan prestaciones y disponibilidad elevadas.
* Las aplicaciones son propias de un servidor, y constituyen una alternativa a los SMP
* **Ventajas:**
* **Escalabilidad absoluta**: dependiente de la cantidad de nodos incorporados se puede disponer de mayores prestaciones.
* **Escalabilidad incremental**: posibilidad de agregar nuevos nodos fácilmente.
* **Alta disponibilidad**: capacidad de seguir operando con nodos en falla.
* **Mejor relación precio/prestaciones**: porque se usan equipos de cómputo estándar (y, posiblemente, de bajo costo).



\*\*Esquema Cluster

|  |  |
| --- | --- |
| **SMP** | **Cluster** |
| * Permiten dar soporte a aplicaciones de alta demanda de recursos. * Más fácil de administrar y configurar. * Cercano a los sistemas de un solo procesador. * La planificación es la diferencia principal. * Menos espacio físico / Menor consumo de potencia. * SMP tiene límite práctico en su número de procesadores: entre 16 y 64 por degradación de prestaciones. | * Permiten dar soporte a aplicaciones de alta demanda de recursos. * Disponibles comercialmente (SMP es más antiguo). * Superior escalabilidad incremental y absoluta * Superior disponibilidad. * Mayor redundancia * Cada nodo tiene su propia memoria principal. Así las aplicaciones no “ven” la memoria global. * La coherencia es mantenida por software y no por hardware. * Alternativa a sistemas SMP para brindar multiprocesamiento a gran escala. |

Tipos de accesos a memoria🡪 En sistemas multiprocesadores se pueden tener 3 tipos de acceso a memoria.

* **UMA - Uniform Memory Access:**
* Igual tiempo de acceso a todas las regiones de memoria.
* Igual tiempo de acceso a memoria para los diferentes procesadores.
* **Ejemplo**: SMP
* **NUMA - Non-uniform memory Access:**
* El tiempo de acceso de un procesador difiere, dependiendo de la región de memoria que accede.
* Diferentes procesadores acceden a diferentes regiones de memoria a diferentes velocidades.
* **Ejemplo**: Cluster
* **CC-NUMA - cache coherente NUMA🡪** Es un *NUMA* que mantiene coherencia de cache entre las cache de los distintos procesadores. Presenta las siguientes características:
* Cada nodo tiene 2 o más procesadores (por ejemplo, un SMP), cierta cantidad de memoria (principal) y E/S.
* Cada procesador tiene su cache (típicamente L1 y L2).
* Los nodos están interconectados por algún tipo de red.
* Existe un espacio de direcciones de memoria único para todos los procesadores de todos los nodos.
* Orden de acceso a memoria:
* Cache L1 (local al procesador).
* Cache L2 (local al procesador).
* Memoria principal (local al nodo).
* Memoria remota (petición por red).
* La coherencia se mantiene en forma automático y transparente
* **Ejemplo**: Sistemas con memoria compartida distribuida.

Arquitecturas on chip🡪 Tanto si son de memoria compartida o distribuida, pueden ser homogéneas (si todos los procesadores son idénticos) o heterogéneas (si los procesadores tienen distintas prestaciones).

Procesamiento Multi-hebra (MultiThreading)🡪 Siendo que un **proceso** está compuesto de **múltiples hebras** (o **hilos**), el procesador puede ser interrumpido fácilmente y *cambiar su control* entre hebras del mismo proceso.

* Se pueden aumentar las prestaciones de un Sistema en 2 niveles:
* **Nivel de instrucciones (IPL):** aumentando la cantidad de instrucciones ejecutadas en paralelo, explotando el paralelismo de instrucciones secuenciales.
* **Nivel de hebras (TPL):** aumentando el nivel de hebras en paralelo, explotando el paralelismo entre instrucciones pertenecientes a distintos hilos de ejecución.
* Como proceso puede tener muchos hilos o hebras. Entonces:
* Las instrucciones de diferentes hebras pueden ser paralelizadas.
* Se podrían mejorar las prestaciones generales si se pudieran ejecutar más de una hebra en paralelo (simultáneamente).
* Si hay recursos libres durante la ejecución de una hebra, ejecutando varias hebras se puede hacer un mejor aprovechamiento de los recursos.
* El procesamiento de multiples hebras (multithreading) consiste en ejecutar 2 o más hebras en forma concurrente:
* Por ejemplo, cuando se ejecuta 1 hebra únicamente, el procesador puede quedar inactivo durante los períodos de I/O.
* Pero si se ejecutan múltiples hebras, durante los períodos de I/O puede ejecutar otras hebras.
* Para poder implementar procesamiento multihebra es necesario, al menos:
* Un PC (contador de programa) distinto para cada hebra que pueda ejecutarse concurrentemente.
* Hardware para ejecución concurrente.
* La ejecución de múltiples hebras puede ser de 2 tipos:
* **Multihebras explícitas**: cuando son definidas en el programa, tanto a nivel de usuario (aplicaciones) como a nivel de Sistema operativo (núcleo).
* **Multihebras implícitas**: cuando son definidas por el compilador (estático) o por hardware (dinámico).
* **Cambios de contexto:**
* El contexto de una hebra está formado por:
* El contador de programa (PC)
* Registros de datos, de direcciones (punteros), de estado, control, de segmentos.
* Datos propios en memoria
* Datos en caché
* Cuando se cambia de una hebra a otra, se produce un cambio de contexto de hebras, que puede ser de 2 tipos:
* **Tradicional**:
* Lo produce el SO, mediante una interrupción del timer.
* La interrupción detiene la hebra en ejecución.
* El SO salva el contexto de la hebra en ejecución.
* El SO recupera el contexto de la hebra detenida anteriormente.
* La hebra detenida anteriormente se reinicia mediante un retorno de interrupción.
* La hebra detenida no se “entera” que fue interrumpida.
* **Rápido** (por hardware):
* Se produce porque la hebra queda en espera por alguna razón, típicamente por un fallo de acceso a la caché
* Si el procesador tiene duplicados, al menos, el PC y los registros, entonces el procesador conmuta de hebra sin necesidad de salvar su contexto (que consume tiempo).
* Este cambio de contexto es mucho más rápido que el tradicional.
* Existen diferentes **políticas de ejecución multihebra** (**MT**) dependiendo si la partición de las tareas es estática o dinámica, espacial o temporal. Las políticas MT pueden ser:
* **CMP🡪 Estática Espacial**
* El procesador dispone de 2 o más núcleos. Cada núcleo ejecuta una hebra.
* **FGMT🡪 Estática Temporal**
* El procesador ejecuta 2 o más hebra concurrentemente.
* El procesador conmuta de una hebra a otra en cada ciclo. La conmutación es rápida, 1 sola hebra por vez.
* Reduce las latencias en memoria, y evita y resuelve dependencias de datos.
* **CGMT🡪 Dinámica Temporal**
* El procesador ejecuta 2 o más hebras concurrentemente.
* El procesador conmuta de una hebra a otra cuando la hebra se detiene por algún evento de gran latencia (como un fallo de acceso a la cache).
* Requiere un manejo más complejo porque la conmutación es asincrónica.
* **SMT🡪 Dinámica Espacial**
* El procesador ejecuta 2 o más hebras simultáneamente. Por eso se llama SMT: Simultaneous Multithreading.
* Hay varias hebras ejecutándose en varias unidades funcionales de un procesador superescalar (1 solo núcleo).
* Se usa en los procesadores de PCs de escritorio.

# Índice de abreviaciones:

* **UC**: Unidad de Control (captura de instrucciones)
* **EP**: Elemento de Proceso (ejecuta)
* **UM**: Unidad de Memoria (almacenamiento de datos)
* **SI**: Secuencia de Instrucciones
* **SD**: Secuencia de Datos
* **ML**: unidad de Memoria Local