

UNIDAD II

ABACUS - SUPERABACUS

ARQUITECTURA VON NEUMANN

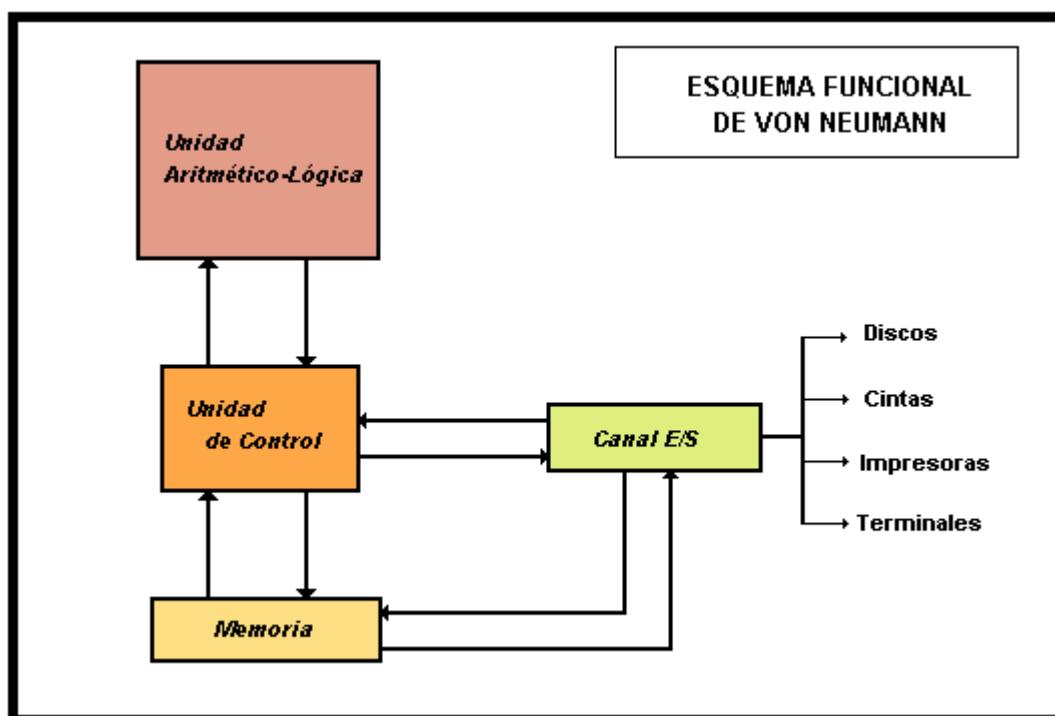
Hasta mediados de los años cuarenta los computadores se programaban por medio de la manipulación de un panel frontal, en el cual los usuarios conectaban o desconectaban determinados switches (comutadores), para que el computador realizase una u otra función y así llevar a cabo la tarea deseada. Es Von Neumann quien hacia 1945, promueve el paso decisivo hacia la mecanización del tratamiento digital de la información, con la invención de dos nuevos conceptos:

- **El programa almacenado:** el computador tendría un programa de instrucciones almacenado en su propia memoria. En lugar de ejecutar las operaciones al compás de su lectura (en una cinta perforada, por ejemplo) la nueva maquina supone almacenado en memoria el programa previamente a la ejecución de las operaciones. El uso de memoria es tanto para el almacenamiento de datos como de instrucciones de programa. Esta idea sencilla es la que ha imperado en el diseño de los computadores hasta nuestros días.
- **La ruptura de Secuencia:** esta es la clave del poder decisorio de los programas. En lugar de necesitar intervención humana cada vez que se planteaba una toma de decisión (íntimamente ligada a los resultados que se iban obteniendo) las operaciones de decisión lógica serían automáticas dotando a la máquina de una instrucción llamada *salto condicional* o *ruptura condicional de secuencia*. Según el valor de un resultado ya obtenido, la maquina ejecutaría una u otra parte del programa.

Los computadores con una arquitectura Von Neumann pueden ser considerados como un conjunto de unidades conectadas entre sí, con una función determinada dentro del esquema del computador. Constan de cinco partes componentes:

- **Memoria:** se encuentra dividida en celdas o posiciones de memoria cuyo contenido es variable y son identificadas por un número fijo llamado dirección de memoria. La capacidad total de una memoria está dada por la cantidad de celdas disponibles, en ella se almacenan dos clases de información: las instrucciones del programa que se deberá ejecutar y los datos (comúnmente llamados operandos) con los cuales deberá trabajar el programa.
- **Unidad aritmético-lógica (UAL):** es la unidad encargada de realizar las operaciones elementales de tipo aritmético (sumas y restas) y de tipo lógico (generalmente comparaciones).

- **Unidad de control (UC):** también llamada unidad de gobierno, desde ella se controlan y gobiernan todas las operaciones (búsqueda, decodificación y ejecución de instrucciones). Es la encargada de extraer de la memoria central la nueva instrucción a ejecutar, analizarla y extraer también los operandos implicados. A su vez, desencadena el tratamiento de los datos en la UAL y, de ser necesario, los almacena en la memoria central.
- **Dispositivo de entrada/salida:** gestiona la trasferencia de un conjunto de informaciones entre las unidades periféricas y la memoria central tanto en un sentido como en otro, y es su tarea advertir a la unidad de control el momento en el cual todas las informaciones han sido transferidas.
- **Bus de datos:** es un sistema digital que proporciona un medio de transporte de datos entre las distintas partes, no almacena información sólo la transmite.



En resumen, se puede representar un ordenador como un conjunto ensamblado de unidades diferentes, cuyo funcionamiento viene dictado por el programa registrado en la memoria central. La unidad de control gobierna la ejecución de las operaciones pedidas por dicho programa. Si la operación es un cálculo, es la unidad aritmético-lógica quien lo realiza, si es una transferencia de informaciones con el exterior se cede el control a un dispositivo de E/S o canal.

CONCEPTOS PRELIMINARES

Previo análisis de la máquina Abacus será necesario introducir las siguientes definiciones que se encuentran íntimamente ligadas entre sí y son de vital importancia para comprender el funcionamiento de un computador:

Registro: los registros son considerados los bloques más importantes de un computador. Una definición general los identifica como una “memoria muy rápida” que permite almacenar una cierta cantidad de bits (información).

Compuerta: las compuertas constituyen la base del hardware sobre la cual se construyen los computadores digitales. Son circuitos electrónicos biestables unidireccionales, es decir, permiten el pasaje de información en un sólo sentido y admiten únicamente dos estados (abierto / cerrado, 1 / 0).

Bus: si bien se aproximó una definición al comienzo de este apunte, podemos profundizar aun más e identificar distintos tipos de buses, a saber:

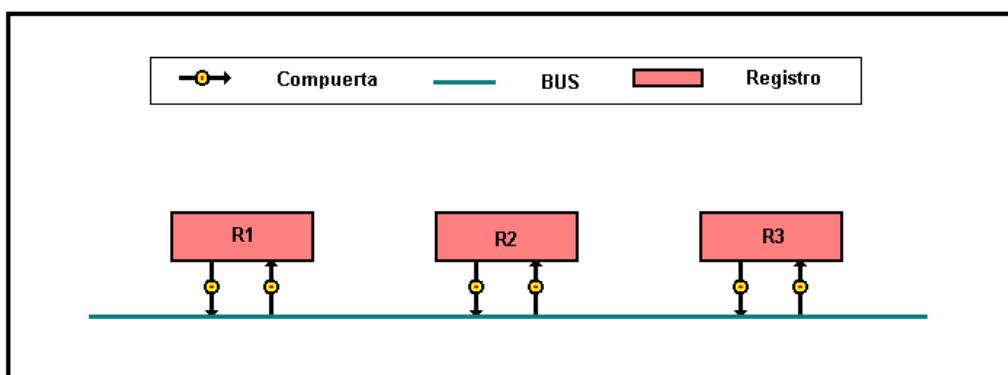
Bus de Datos: mueve la información por los componentes de hardware internos y externos del sistema tanto de entrada como de salida (teclado, Mouse etc.)

Bus de Direcciones: ubica los datos en la memoria teniendo relación directa con los procesos de la CPU.

Bus de Control: marca el estado de una instrucción que fue dada a la PC.

La transferencia de datos puede darse entre los componentes de un ordenador o entre ordenadores, es decir existen buses internos (ej: transportar datos desde y hasta la UAL) y buses externos (ej: conectar el ordenador con dispositivos de E/S).

Los tres elementos que hemos definido (Compuerta, Registro y Bus) nos permitirán comprender cómo se lleva a cabo la transferencia de información y, por ende, el proceso de ejecución de una instrucción.

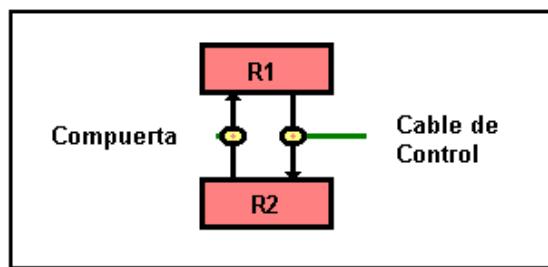


Para intercambiar información entre dos registros, es necesario que dicha información viaje a través del bus. El traspaso de la misma entre un registro y el bus de datos, por ejemplo,

esta regido por las compuertas: sólo si la compuerta está abierta se produce el pasaje de datos. Cuando se abre una compuerta la información esta disponible en el bus y mientras esté abierta los datos permanecerán en el mismo. Sin embargo, el bus no tiene capacidad de almacenamiento, por ende, al cerrar la compuerta la información desaparecerá.

La compuerta que manda información al bus es única, es decir, no pueden existir dos abiertas simultáneamente, esto es importante tenerlo presente a la hora de indicar la secuencia de apertura de compuertas en el proceso de ejecución de una instrucción.

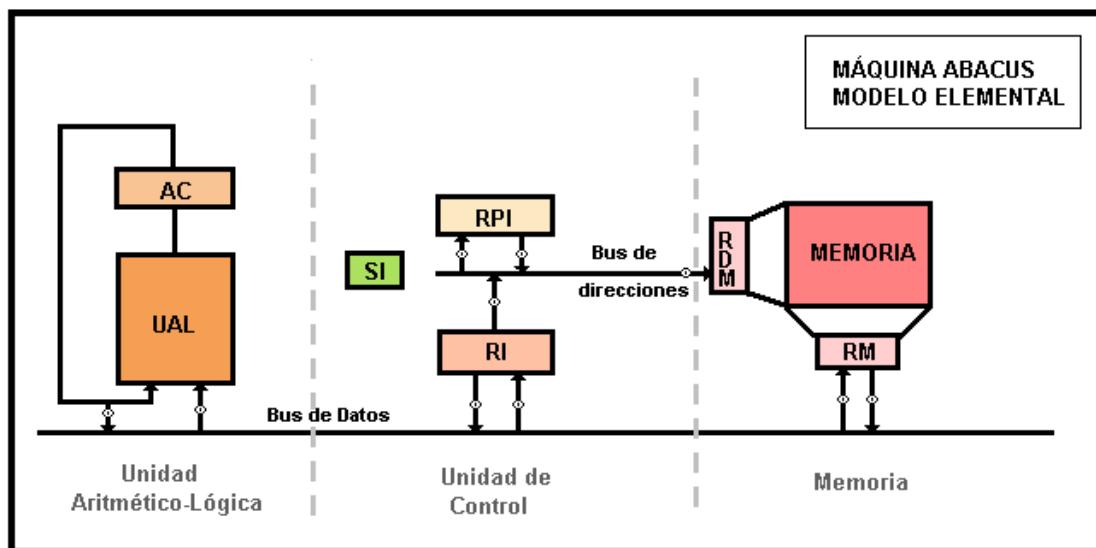
Tomemos la siguiente imagen como ejemplo del traspaso de información entre registros:



Al abrir la compuerta se copia la información de R2 a R1, al cerrarla la información queda almacenada en ambos registros. El cable de control es quien da la orden de abrir o cerrar la compuerta (que permitirá el pasaje unidireccional de datos).

ABACUS: MAQUINA ELEMENTAL

Abacus es una maquina de tipo Von Neumann cuyo esquema es el siguiente:



Registros

AC: acumulador

RI: registro de instrucción

SI: secuenciador de instrucciones

RPI: registro de próxima instrucción

RI: registro de instrucción

RDM: registro de direcciones de memoria

RM: registro de memoria

Descripción de Componentes

I) UAL:

Abacus es una máquina de una sola dirección, su unidad aritmético-lógica posee un registro particular llamado acumulador (**AC**) que sirve tanto para albergar el primer operando como para albergar el resultado. Esta característica permite instrucciones de una sola dirección: la del segundo operando.

Entre las operaciones que realiza la UAL pueden mencionarse:

- CARGA (cargar un contenido en el AC)
- ALMACENAMIENTO (enviar datos a memoria por el bus)
- SUMA (sumar un dato a lo que haya en el AC)
- LOGICAS (OR, AND, XOR)

Es importante recordar que en una maquina Abacus **todas las operaciones** se realizan “contra” el acumulador y el resultado siempre queda almacenado en él.

II) UC

La Unidad de Control, como ya hemos visto, es la encargada de extraer y analizar las instrucciones de la memoria central y, para ello, se vale de dos registros:

RPI: contiene la dirección de la próxima instrucción a ejecutar, se comunica con la memoria y con el RI a través del Bus de Direcciones. A medida que se van ejecutando las instrucciones este registro aumenta su contenido en una unidad excepto para las instrucciones de ruptura de secuencia.

RI: contiene la instrucción extraída de la memoria, en ella podemos identificar dos partes fundamentales: el código de operación y la dirección del operando. Este registro se comunica con la memoria mediante el Bus de Datos.

En cuanto al **SI** cabe destacar que tras analizar el código de operación distribuye las ordenes de la Unidad de Control a la Unidad Aritmético-Lógica y a la Memoria para ejecutar las fases de la instrucción, en otras palabras, es el encargado de administrar la apertura y cierre de las compuertas para el correcto funcionamiento y pasaje de la información.

III) MEMORIA

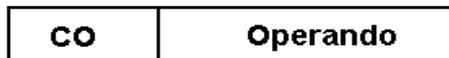
El intercambio de información entre la memoria y el resto del computador, ya sea para acceder a los datos o para almacenar información en la misma, se lleva a cabo a través de dos registros:

RDM: contiene la dirección de la celda de memoria en (de) la cual se escribirá (leerá) la información

RM: contiene el dato que debe ser leído (escrito) desde (en) la memoria.

Registro de Instrucción

Vamos a detenernos un momento en un concepto clave de la Máquina Abacus: el formato de instrucciones que soporta:



Para entender a que hacemos referencia cuando decimos que Abacus es una máquina de una sola dirección tomemos como ejemplo una instrucción SUMA, el objetivo es sumar el contenido de dos celdas de memoria distintas (de diferente dirección) y almacenar el resultado en una de ellas.

Dado que en la instrucción sólo podemos indicar la dirección de un único operando nos veremos obligados a realizar las siguientes acciones:

1. Cargar el primer operando en el AC.
2. Sumar el segundo operando al contenido del AC.
3. Almacenar en memoria el resultado contenido del AC.

Se deduce a simple vista que todas las operaciones se realizan “contra el acumulador”.

CARGAR	200	Se suman los contenidos de las celdas de memoria cuayas direcciones son 200 y 206 El resultado se almacena en la celda de memoria de dirección 200
SUMAR	206	
ALMACENAR	200	

En este ejemplo vemos que de ser posible indicar dos direcciones, bastaría tan solo con ejecutar una única instrucción que contenga las direcciones de los dos operandos que queremos sumar.

Relaciones

En base a las relaciones que se establecen entre los componentes de la Maquina Abacus se pueden deducir las siguientes equivalencias:

- Tamaño RPI = Tamaño RDM = Tamaño Op = Cantidad de Celdas Direccionalables
- Tamaño AC = Tamaño RI = Tamaño RM = Longitud de Instrucción = Longitud de Celda

Esto se verá claramente a continuación, donde detallaremos el desarrollo de una instrucción Abacus.

Desarrollo de una instrucción

El desarrollo de una instrucción en Abacus puede descomponerse en 2 fases:

- **Fase de Búsqueda:** consiste en localizar la instrucción que se va a ejecutar; esta fase es común a todos los tipos de instrucciones y, a su vez, la secuencia de acciones que se lleva a cabo es idéntica a la búsqueda de operandos. También se actualiza en forma secuencial la dirección de la siguiente instrucción a ejecutar.
- **Fase de Ejecución:** como su nombre lo indica consiste en ejecutar la instrucción, claramente esta fase es dependiente del tipo de tarea a realizar (suma, almacenamiento, salto, etc.).

Descripción de Fases

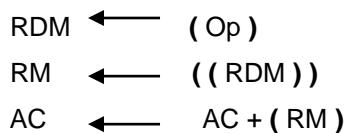
I) Búsqueda de una instrucción: La unidad de control ordena la transferencia del contenido del RPI al RDM y envía a la memoria una orden de lectura. El contenido de la celda de memoria queda almacenado en el RM, luego la Unidad de Control ordena la transferencia del contenido del RM al RI, pudiendo entonces los circuitos especializados analizar el código de operación de la instrucción. Finalmente se prepara el RPI para ejecutar la próxima instrucción.



Para realizar la búsqueda de un operando, una vez que la Unidad de control analiza el contenido del código de operación, ordena la transferencia del contenido del campo operando del RI al RDM y luego envía una orden de operación de lectura. El operando buscado queda disponible en el RM.

II) Ejecución de una instrucción: la ejecución de cada instrucción implica el movimiento de los datos y como estos pasos se realizan en forma secuencial y ordenada la UCP sigue las señales dadas por el reloj del sistema. Se ha indicado más arriba que esta fase depende exclusivamente del tipo de tarea que se deba realizar, por ende, se analizarán casos particulares.

- **Suma:** se debe sumar el contenido del RM al contenido del acumulador



- **Carga:** se debe almacenar en el acumulador un dato contenido en memoria

RDM ← (Op)
 RM ← ((RDM))
 AC ← (RM)

- **Almacenamiento:** se debe “guardar” en memoria el contenido del acumulador

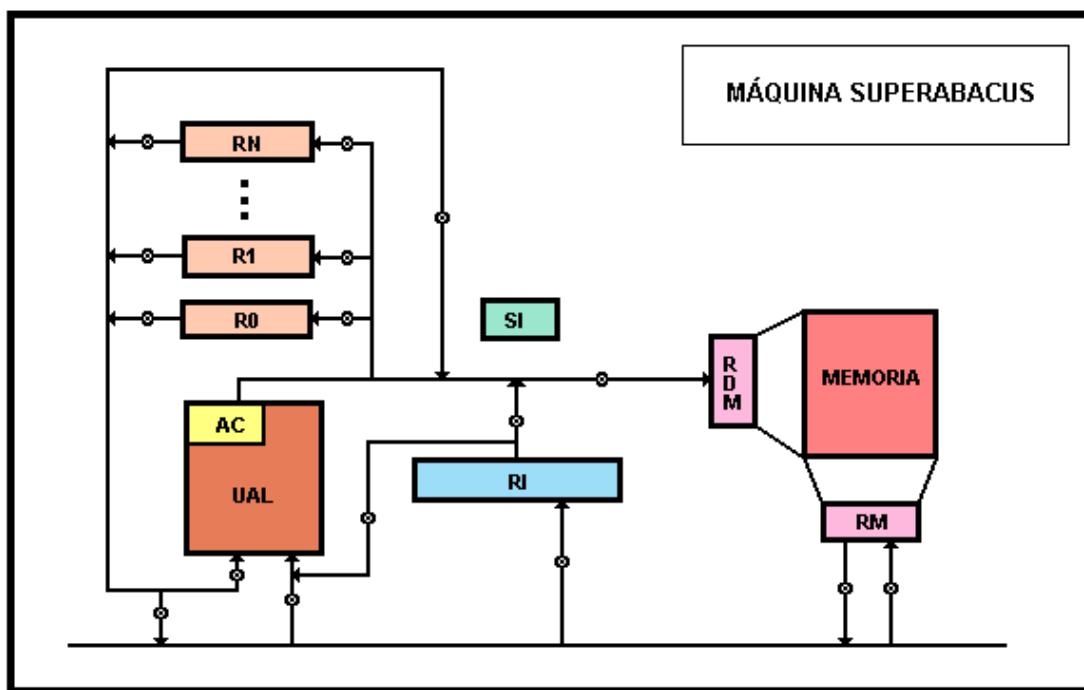
RDM ← (Op)
 RM ← (AC)
 (RDM) ← (RM)

- **Bifurcación:** se debe “saltar” a la dirección indicada en la instrucción. La dirección de bifurcación debe ser transferida al RPI (para buscar la próxima instrucción a ejecutar).

RPI ← (Op)

SUPERABACUS

Superabacus es una maquina de tercera generación ya que posee un conjunto de registros banalizados, es decir, utilizables tanto como registros aritméticos o, según las condiciones de direccionamiento, como registro base o como registro de índice para cálculo de direcciones. Todos los cálculos se realizan en un solo sumador que actúa a la vez de unidad aritmético-lógica y de unidad de calculo de direcciones.



Posee sensiblemente características idénticas a Abacus, excepto en un punto: el ciclo de memoria equivale a cuatro impulsos de reloj, en lugar de dos, lo que autoriza entre otras

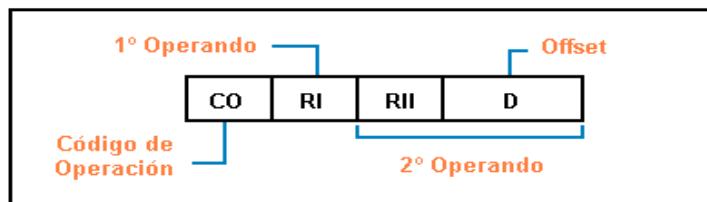
cosas a efectuar una modificación de dirección (sumar el contenido de un registro con la dirección indicada en la instrucción) sin retrasar por ello el inicio del ciclo de procesamiento.

Como características principales podemos enumerar las siguientes:

1. Posee un conjunto de registros generales éstos pueden contener datos o direcciones.
2. No tiene **RPI** se asigna esa función a **R0** (registro cero), cuyo incremento se consigue por transferencia vía el sumador.
3. Es una máquina de **dos direcciones** (**1º y 2º operando**).
4. La **UAL** se utiliza tanto para calcular direcciones como para operar con los datos.

Registro de Instrucción

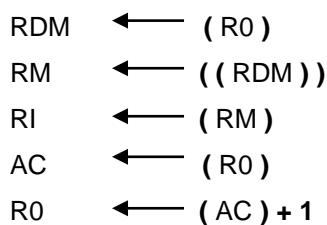
En Superabacus la estructura del **RI** es diferente respecto de la Máquina Abacus; es un registro que alberga instrucciones de dos operandos. Posee Código de Operación, 1º Operando (RI) y 2º Operando (RII - D). La siguiente figura representa la estructura del **RI**:



RI y RII representan dos registros diferentes (se los indica con su número Ej.: R1, R5, etc.). El segundo operando puede contener la dirección de una celda de memoria, se obtiene sumando el Offset (desplazamiento) al contenido del registro indicado: **A (celda) = (RII) + D**.

Fase de búsqueda

Al igual que en el caso de la máquina Abacus, la fase de búsqueda de las instrucciones de Superabacus es común para todas.



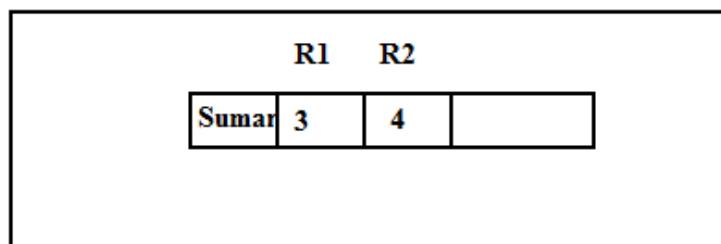
Como se puede apreciar, la gran diferencia con la máquina Abacus se da en la forma en que se incrementa el valor del R0 que hace las veces de RPI. En este caso ese incremento se resuelve usando la UAL en vez del SI.

Instrucciones

Las operaciones pueden ser entre registros, entre un registro y un dato inmediato o entre un registro y un operando en memoria. Analizaremos cada caso para una instrucción SUMAR.

1. **Sumar Registro:** se suma el contenido de ambos registros y el resultado se almacena en aquel que se indica en el primer operando.

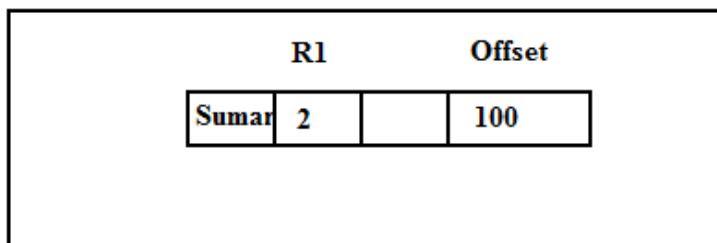
SUMAR RI, RII (Ejemplo SUMAR 3, 4)



$$\begin{array}{l}
 AC \quad \longleftarrow (R3) \\
 AC \quad \longleftarrow (R4) + (AC) \\
 R3 \quad \longleftarrow (AC)
 \end{array}$$

2. **Sumar Inmediato:** se suma al registro indicado en el 1º operando el dato inmediato almacenado en la instrucción. El resultado se almacena en el registro.

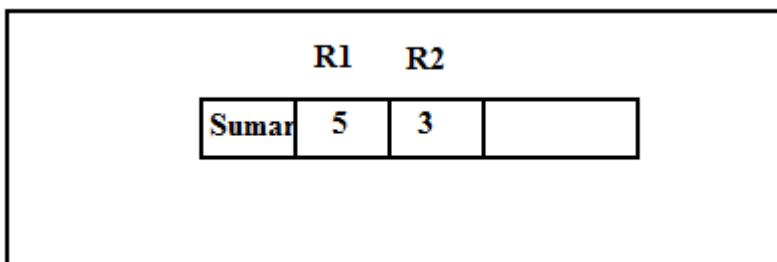
SUMAR RI, DII (Ejemplo SUMAR 2, 100)



$$\begin{array}{l}
 AC \quad \longleftarrow (R2) + 100 \\
 R2 \quad \longleftarrow (AC)
 \end{array}$$

3. **Sumar Palabra en memoria (Registro Indirecto):** se suma al registro indicado en el primer operando el contenido de la celda de memoria cuya dirección está dada por el contenido del registro indicado en el segundo operando. El resultado queda almacenado en el registro del primer operando:

SUMAR RI, (RII) (Ejemplo SUMAR 5, (3))



$$\begin{array}{l}
 RDM \quad \leftarrow (R3) \\
 RM \quad \leftarrow ((RDM)) \\
 AC \quad \leftarrow (R5) + (RM) \\
 R5 \quad \leftarrow (AC)
 \end{array}$$

En este caso podemos notar que dado que el acumulador tiene una entrada desde los registros y una desde la memoria es posible realizar simultáneamente la suma del R5 y el RM.

4. **Sumar Palabra en memoria (Desplazamiento):** se suma al registro indicado en el primer operando el contenido de la celda de memoria cuya dirección está dada por el contenido del registro indicado en el segundo operando más el Offset. El resultado queda almacenado en el registro del primer operando:

SUMAR RI, DI (RII) (Ejemplo SUMAR 5, 20 (3))

R1	R2	Offset
Sumar	5	3

$$\begin{array}{l}
 AC \quad \leftarrow (R3) + 20 \\
 RDM \quad \leftarrow (AC) \\
 RM \quad \leftarrow ((RDM)) \\
 AC \quad \leftarrow (R5) + (RM) \\
 R5 \quad \leftarrow (AC)
 \end{array}$$

En este caso podemos notar que dado que el acumulador tiene una entrada desde los registros y una desde la memoria es posible realizar simultáneamente la suma del R5 y el RM.

Mirando las instrucciones ejecutadas, nos preguntamos, ¿Como se da cuenta la maquina Superabacus en el llenado de los operandos? Y la respuesta es la siguiente, cada instrucción **Sumar** tiene un código de operación diferente en cada caso.

75.03 & 95.57 Organización del Computador

U5 – COMPONENTES DE UN COMPUTADOR ADMINISTRACIÓN DE MEMORIA

U5 – Componentes de un computador

○ Administración de Memoria

- Sistema Operativo

- “Software que administra los recursos del computador, provee servicios y controla la ejecución de otros programas”

- Algunos servicios que provee

- Schedule de procesos
 - Administración de memoria

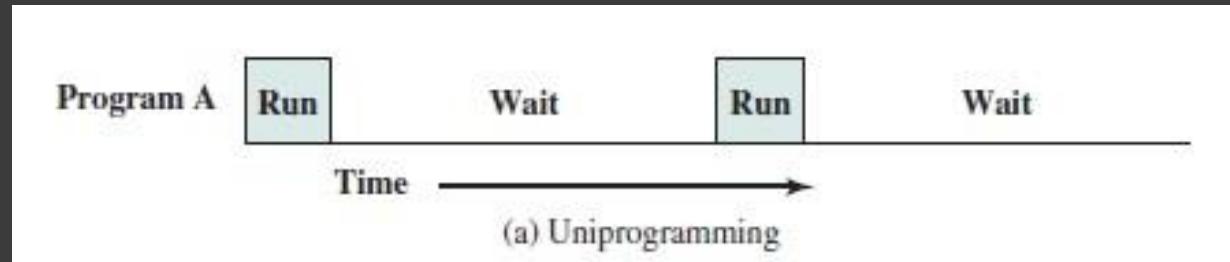
- Monitor

- Parte residente del Sistema Operativo

U5 – Componentes de un computador

○ Uniprogramación

- Un solo proceso de usuario en ejecución a la vez
- La memoria de usuario está completamente disponible para ese único proceso
- Uso del procesador a lo largo del tiempo



- Run: Tiempo efectivo de uso del CPU
- Wait: Tiempo ocioso del CPU esperando E/S (*Idle time*)

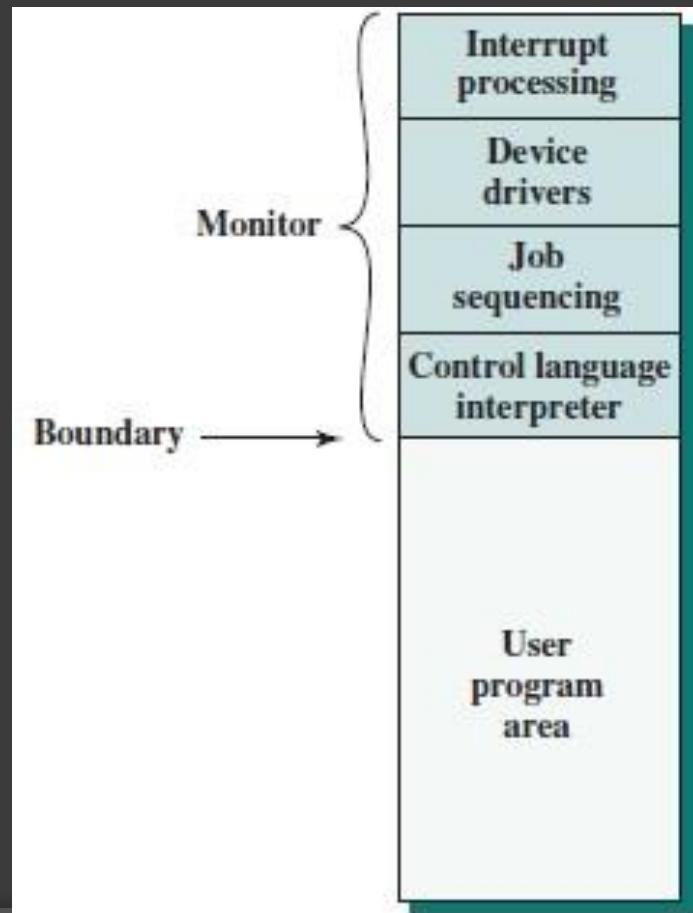
U5 – Componentes de un computador

- Administración de memoria simple
 - Sistema con uniprogramación
 - Se divide la memoria en dos partes
 - Monitor del S.O.
 - Programa en ejecución en ese momento
 - Ventajas:
 - Simplicidad
 - Desventajas:
 - Desperdicio de memoria
 - Desaprovechamiento de los recursos del computador
 - Ej. MS-DOS, iPhone OS v1-3, IBM OS/PCP (Primary Control Program)

U5 – Componentes de un computador

○ Administración de memoria simple

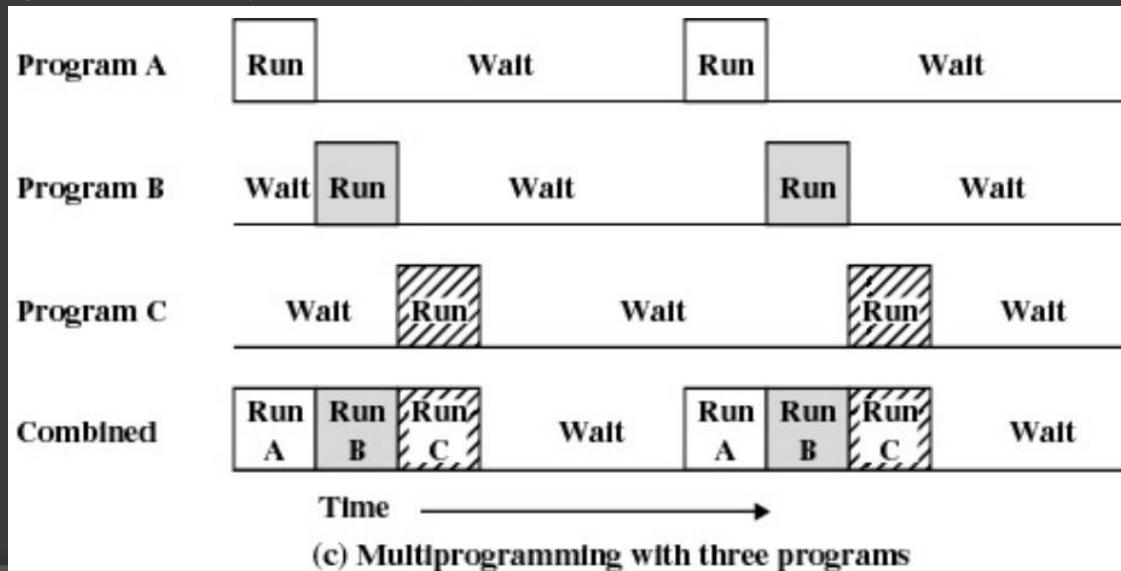
- Memoria



U5 – Componentes de un computador

○ Multiprogramación

- Varios procesos de usuario en ejecución a la vez
- Se divide la memoria de usuario entre los procesos en ejecución
- Se comparte el tiempo de procesador entre los procesos en ejecución (timeslice)



U5 – Componentes de un computador

○ Multiprogramación

- Condiciones de finalización de los procesos:
 - Termina el trabajo
 - Se detecta un error y se cancela
 - Requiere una operación de E/S (suspensión)
 - Termina el timeslice (suspensión)

U5 – Componentes de un computador

- Administración de memoria por asignación particionada
 - Sistema con multiprogramación
 - La memoria de usuario se divide en particiones de tamaño fijo:
 - Iguales
 - Distintas
 - Ventajas:
 - Permite compartir la memoria entre varios procesos
 - Desventajas:
 - Desperdicio de memoria
 - Fragmentación interna (dentro de una partición)
 - Fragmentación externa (particiones no usadas)
 - Ej. IBM OS/MFT (Multiprogramming with a Fixed number of Tasks)

U5 – Componentes de un computador

○ Admin. de memoria por asignación particionada

- Particiones Fijas
 - Iguales
 - Distintas



(a) Equal-size partitions



(b) Unequal-size partitions

U5 – Componentes de un computador

- Administración de memoria por asignación particionada reasignable
 - Sistema con multiprogramación
 - Swapping
 - La memoria de usuario se divide en particiones de tamaño variable
 - Compactación para eliminar la fragmentación
 - Se usa un recurso de hardware (registro de reasignación) para la realocación
 - Realocación dinámica en tiempo de ejecución
 - Ventajas:
 - Permite compartir la memoria entre varios procesos
 - Elimina el desperdicio por fragmentación interna. Con la compactación se elimina además la fragmentación externa
 - Desventajas:
 - La tarea de compactación es costosa
 - Ej. IBM OS/MVT (Multiprogramming with a Variable number of Tasks)

U5 – Componentes de un computador

○ Admin. de memoria por asignación particionada reasignable

- Particiones variables

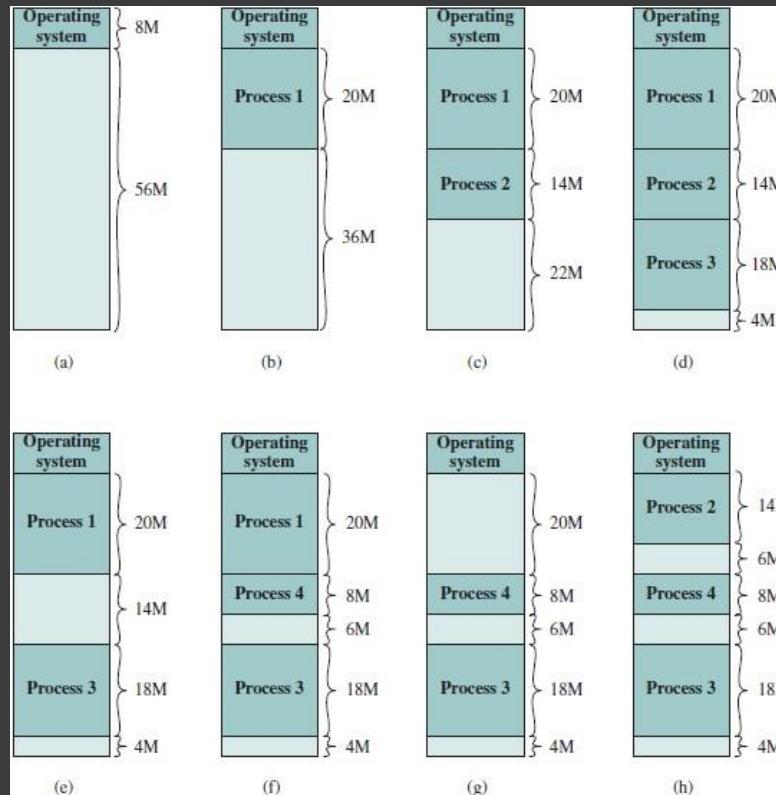


Figure 8.14 The Effect of Dynamic Partitioning

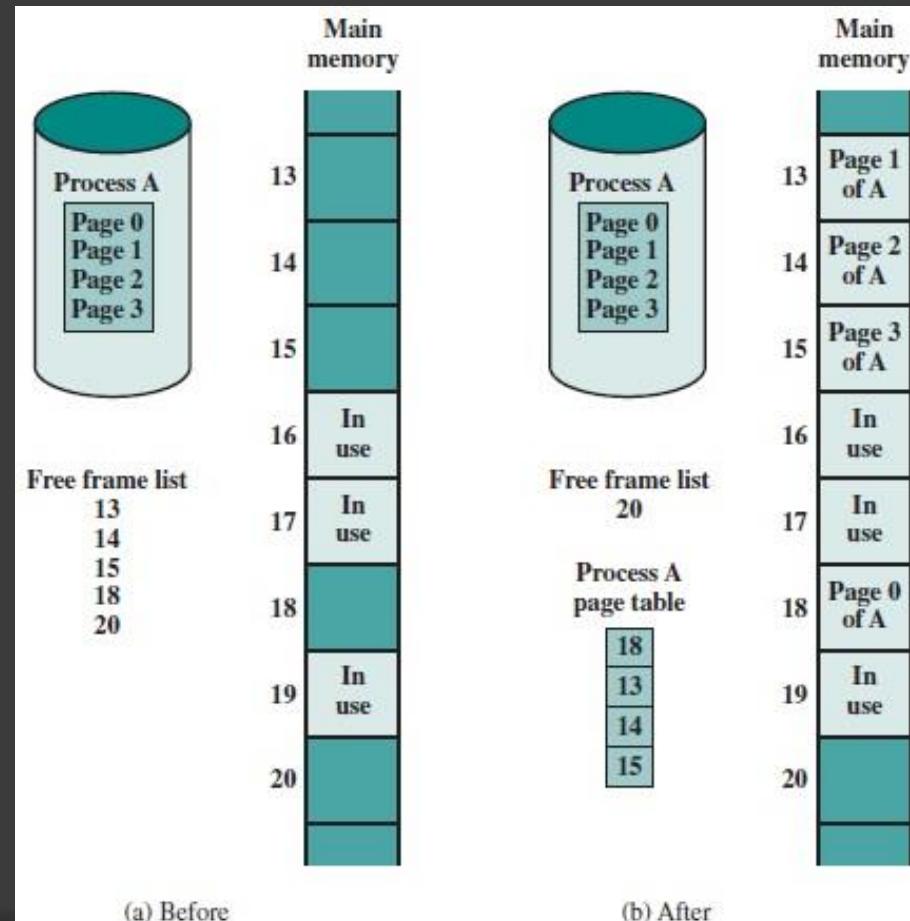
U5 – Componentes de un computador

- Administración de memoria paginada simple
 - Sistema con multiprogramación
 - Se divide el address space del proceso en partes iguales (páginas) (ej. IA-32 4KB c/u)
 - Se divide la memoria principal en partes iguales (frames)
 - Hay una tabla de páginas por proceso
 - Hay una lista de frames disponibles
 - Se cargan a memoria las páginas del proceso en los frames disponibles (no es necesario que sean contiguos)
 - Las direcciones lógicas se ven como número de página y un offset
 - Se traducen las direcciones lógicas en físicas (*address translation*) con soporte del hardware (MMU – Memory Management Unit)
 - La paginación es transparente para el programador

U5 – Componentes de un computador

○ Administración de memoria paginada simple

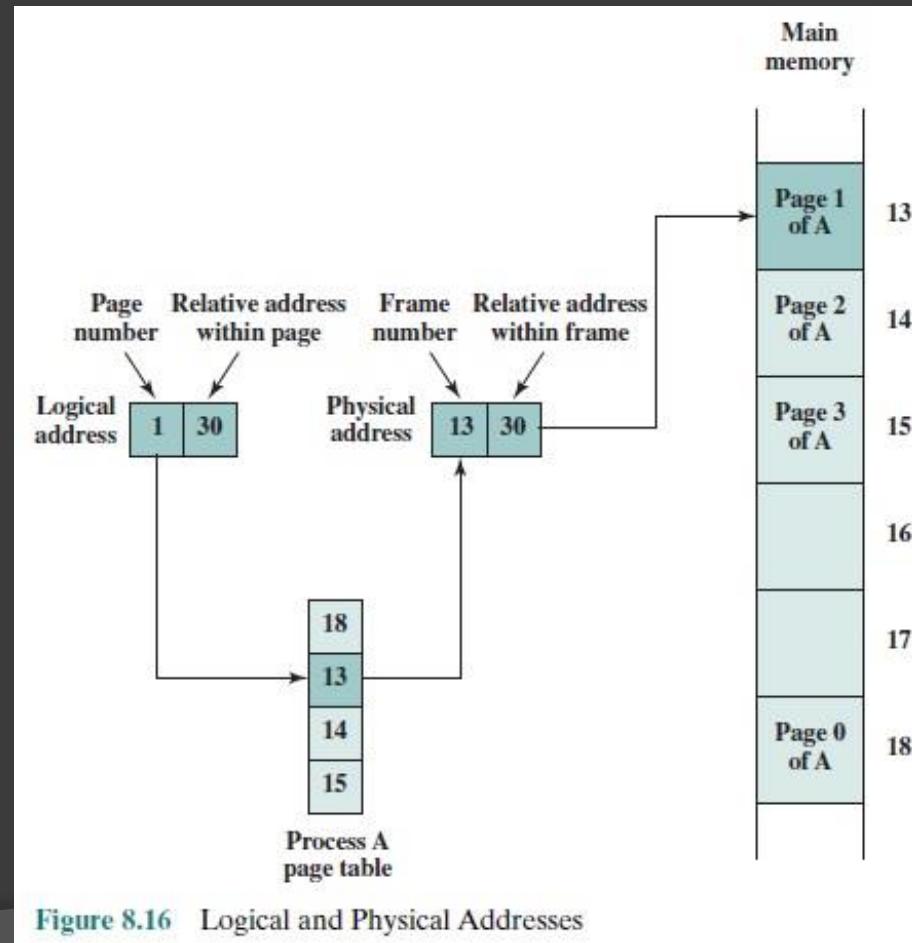
- Páginas y frames



U5 – Componentes de un computador

○ Administración de memoria paginada simple

- Traducción de direcciones lógicas a físicas



U5 – Componentes de un computador

- Administración de memoria paginada simple
 - Ventajas:
 - Permite compartir la memoria entre varios procesos
 - Permite el uso no contiguo de la memoria
 - Minimiza la fragmentación interna (solo existe dentro de la última página de cada proceso)
 - Elimina la fragmentación externa
 - Desventajas:
 - Se requiere subir todas las páginas del proceso a memoria
 - Se requieren estructuras de datos adicionales para mantener información de páginas y frames

U5 – Componentes de un computador

- Administración de memoria paginada por demanda (memoria virtual)
 - Sistema con multiprogramación
 - Solo se cargan a memoria principal las páginas necesarias para la ejecución de un proceso
 - Cuando se quiere acceder a una posición de memoria de una página no cargada se produce un *page fault*
 - El page fault dispara una interrupción por hardware (MMU) atendida por el sistema operativo
 - El sistema operativo (*page fault handler*) levanta la página solicitada desde memoria secundaria (memoria virtual)
 - Si no hay frames libres es necesario bajar páginas a memoria secundaria y reemplazarlas (*page swapping*)
 - Algoritmos para reemplazo de páginas (por ejemplo FIFO, First In First Out o LRU, Least Recently Used)
 - Thrashing: el CPU pasa más tiempo reemplazando páginas que ejecutando instrucciones

U5 – Componentes de un computador

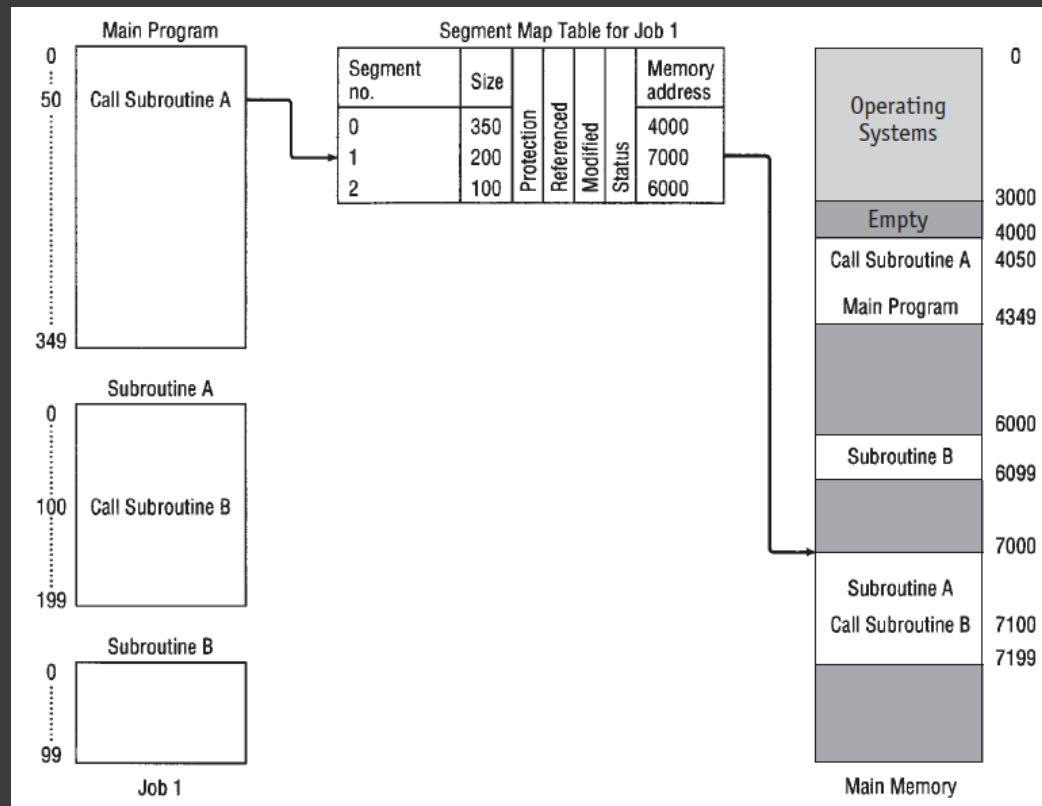
- Administración de memoria paginada por demanda
 - Ventajas:
 - No es necesario cargar todas las páginas de un proceso a la vez
 - Maximiza el uso de la memoria al permitir cargar más procesos a la vez
 - Un proceso puede ocupar más memoria de la efectivamente instalada en el computador
 - Desventajas:
 - Mayor complejidad por la necesidad de implementar el reemplazo de páginas
 - Ej. Windows 3.x en adelante, Linux

U5 – Componentes de un computador

- Administración de memoria por segmentación
 - Sistemas con multiprogramación
 - Generalmente visible al programador
 - La memoria del programa se ve como un conjunto de segmentos (múltiples espacios de direcciones)
 - Los segmentos son de tamaño variable y dinámico
 - El sistema operativo administra una tabla de segmentos por proceso
 - Permite separar datos e instrucciones
 - Permite dar privilegios y protección de memoria como por ej. lectura, escritura, ejecución. (segmentation faults como mecanismos de excepción de hardware para accesos indebidos)
 - Las referencias a memoria se forman con un número de segmento y un offset dentro de él. Con ayuda de hardware (MMU – Memory Management Unit) se hacen las traducciones de las direcciones lógicas a físicas
 - Se pueden usar para implementar memoria virtual (solo se suben a memoria física algunos segmentos por proceso)

U5 - Componentes de un computador

○ Administración de memoria por segmentación



U5 – Componentes de un computador

- Administración de memoria por segmentación
 - Ventajas:
 - Simplifica el manejo de estructuras de datos con crecimiento
 - Permite compartir información entre procesos dentro de un segmento
 - Permite aplicar protección/privilegios sobre un segmento fácilmente
 - Desventajas:
 - Fragmentación externa en la memoria principal por no poder alojar un segmento
 - Hardware más complejo que memoria paginada para la traducción de direcciones
 - Ej. Burroughs Corporation B5000-B6500, IBM AS/400, Intel x86 (por compatibilidad hacia atrás)

U5 – Componentes de un computador

○ Administración de memoria

- Distintas combinaciones (Ej. Intel Pentium)
 - Sin segmentación y sin paginación
 - Direcciones lógicas iguales a las físicas. No es útil para multiprogramación. Usado en controladores de alta performance
 - Paginación sin segmentación
 - La protección y administración de la memoria se hace a través de las páginas.
 - Ej. Berkeley UNIX
 - Segmentación sin paginación
 - La memoria se ve como una colección de espacios lógicos, con protección a nivel segmentos.
 - Segmentación con paginación
 - Segmentos para controlar el acceso a particiones de memoria.
 - Páginas para administrar la locación dentro de los segmentos
 - Ej. UNIX System V

U5 – Componentes de un computador

○ Referencias

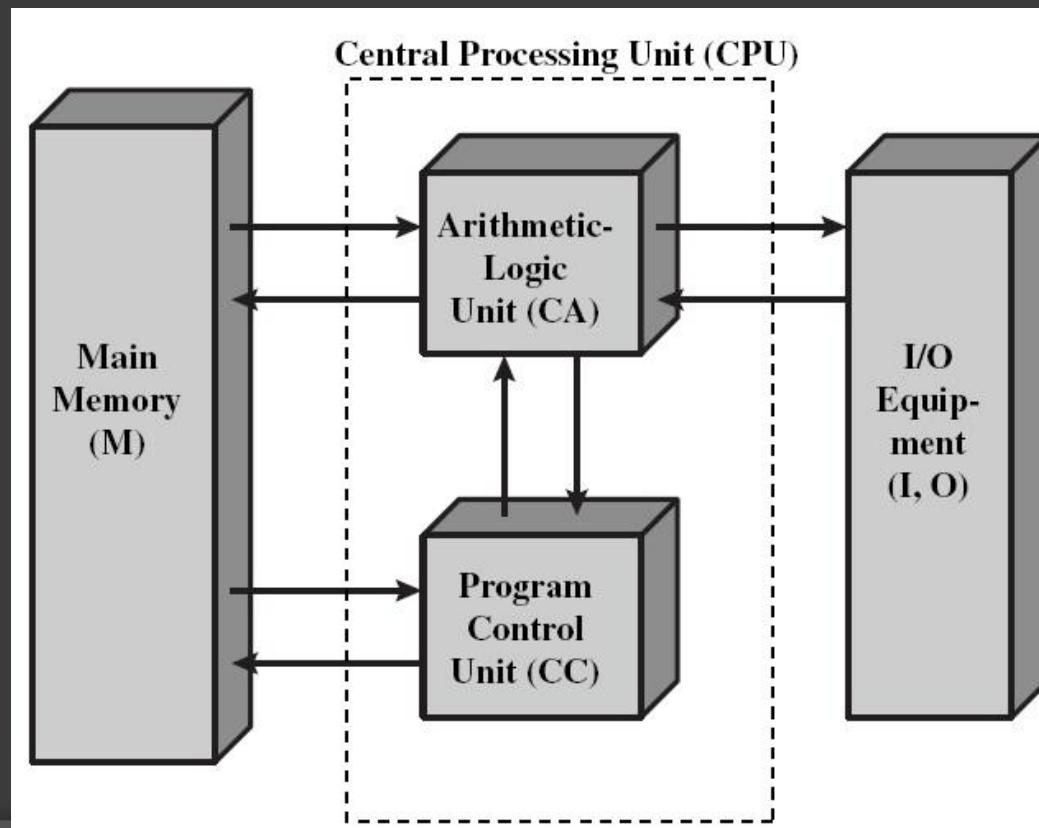
- “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings (<http://williamstallings.com/ComputerOrganization/>)
- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin (<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)
- “Understanding Operating Systems” 8va edición. Ann McHoes / Ida M. Flynn

75.03 / 95.57 Organización del Computador

U5 - COMPONENTES DE UN COMPUTADOR ENTRADA/SALIDA

U5 – Componentes de un computador

- Entrada/Salida
 - Módulo de E/S



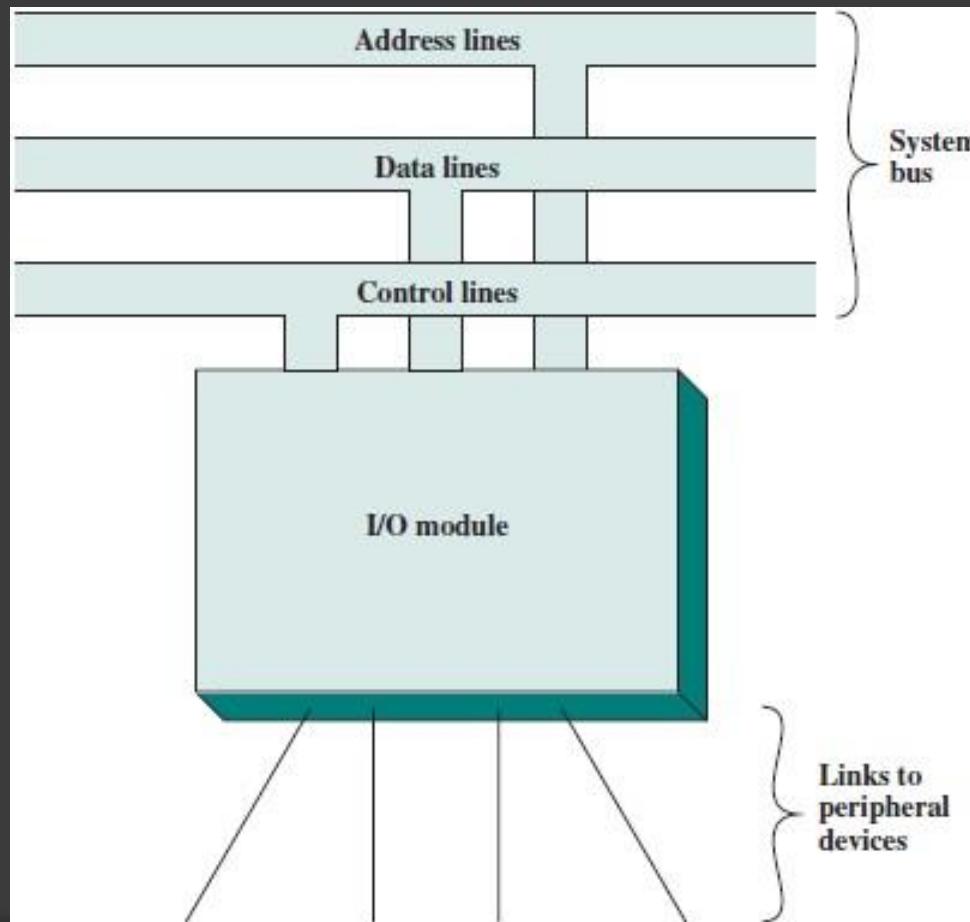
U5 – Componentes de un computador

◎ Módulo de E/S

- ¿Qué hace?
 - Conecta a los periféricos con la CPU y la memoria a través del bus del sistema o switch central y permite la comunicación entre ellos
- ¿Por qué existe?
 - Amplia variedad de periféricos con distintos métodos de operación
 - La tasa de transferencia de los periféricos es generalmente mucho más lenta que la de la memoria y procesador
 - Los periféricos usan distintos formatos de datos y tamaños de palabra
- ¿Para qué sirve?
 - Oculta detalles de timing, formatos y electro mecánica de los dispositivos periféricos

U5 – Componentes de un computador

○ Módulo de E/S



U5 – Componentes de un computador

- Módulo de E/S
 - Interface interna (bus del sistema)
 - Datos
 - Direcciones
 - Control
 - Interface externa (periféricos)
 - Datos
 - Control
 - Estado

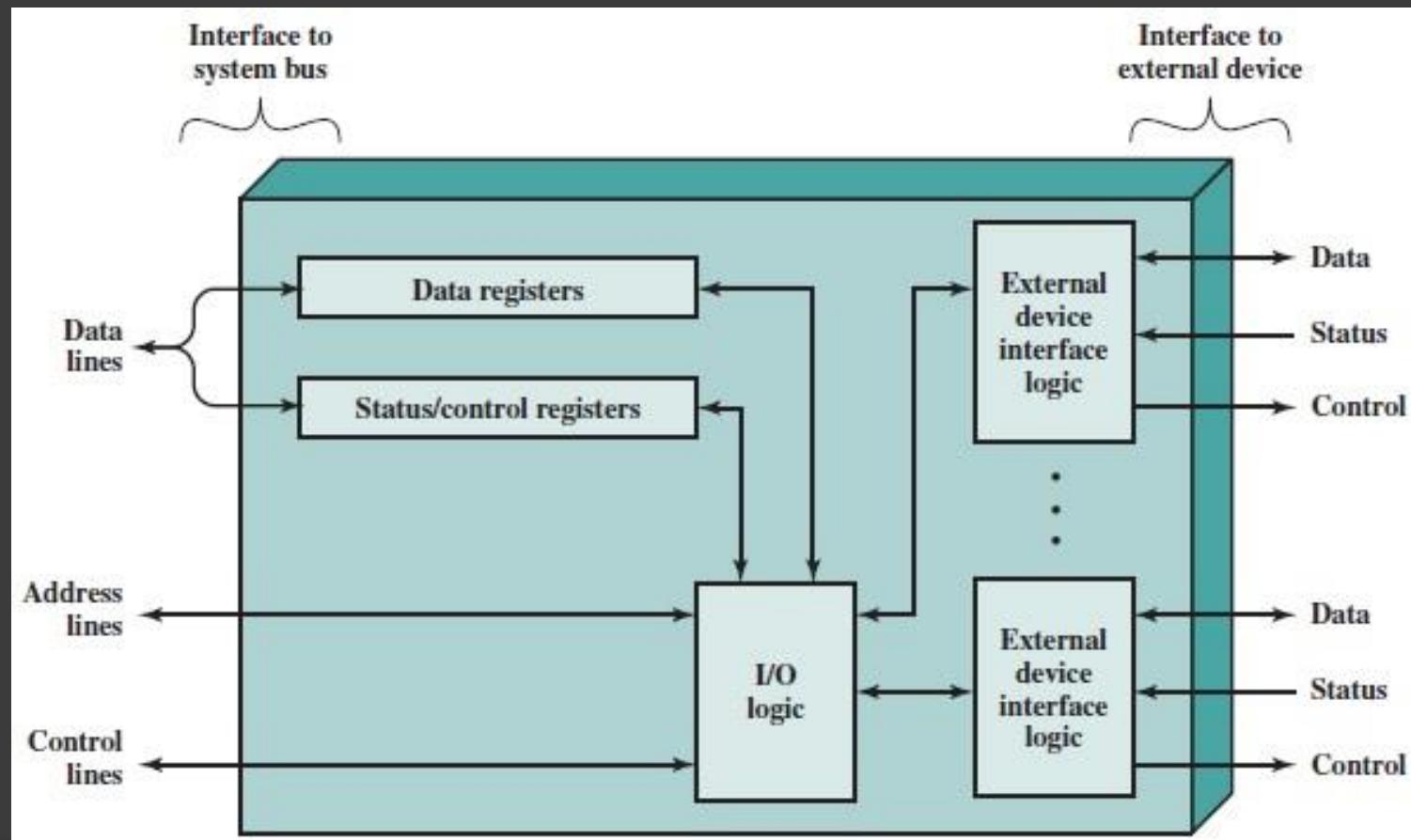
U5 – Componentes de un computador

○ Módulo de E/S

- Funciones
 - Control & Timing
 - Controla flujo de tráfico entre CPU/Memoria y periféricos
 - Comunicación con el procesador
 - Decodificación de comandos
 - Datos
 - Información de estado
 - Reconocimiento de direcciones
 - Comunicación con el dispositivo
 - Comandos
 - Información de estado
 - Datos
 - Buffering de datos
 - Detección de errores

U5 - Componentes de un computador

○ Estructura del módulo de E/S



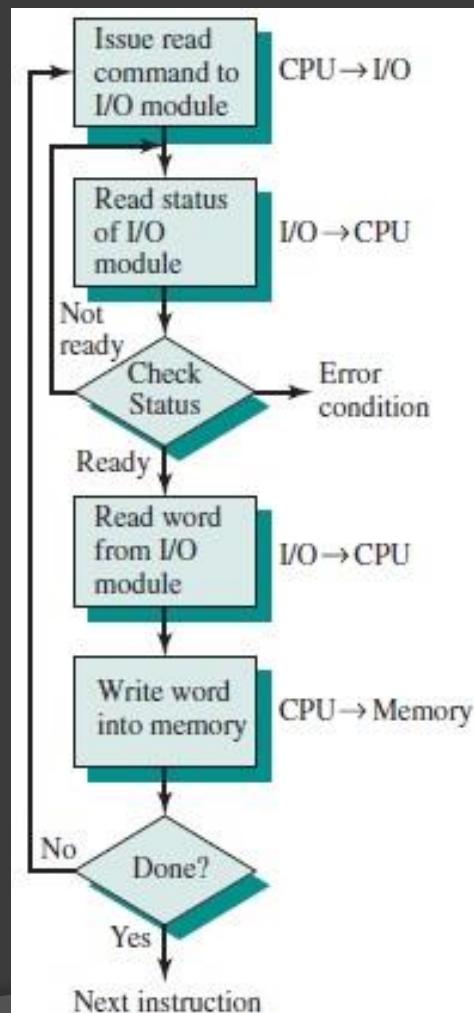
U5 – Componentes de un computador

○ Módulo de E/S

- Técnicas para operaciones de E/S
 - E/S Programada
 - E/S manejada por interrupciones
 - Acceso directo a memoria (DMA)

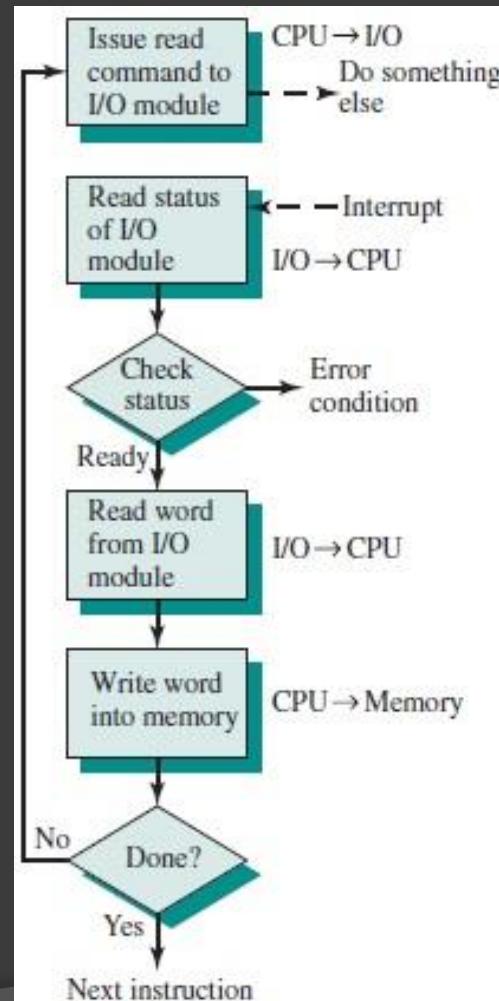
U5 – Componentes de un computador

- Módulo de E/S
 - E/S Programada



U5 – Componentes de un computador

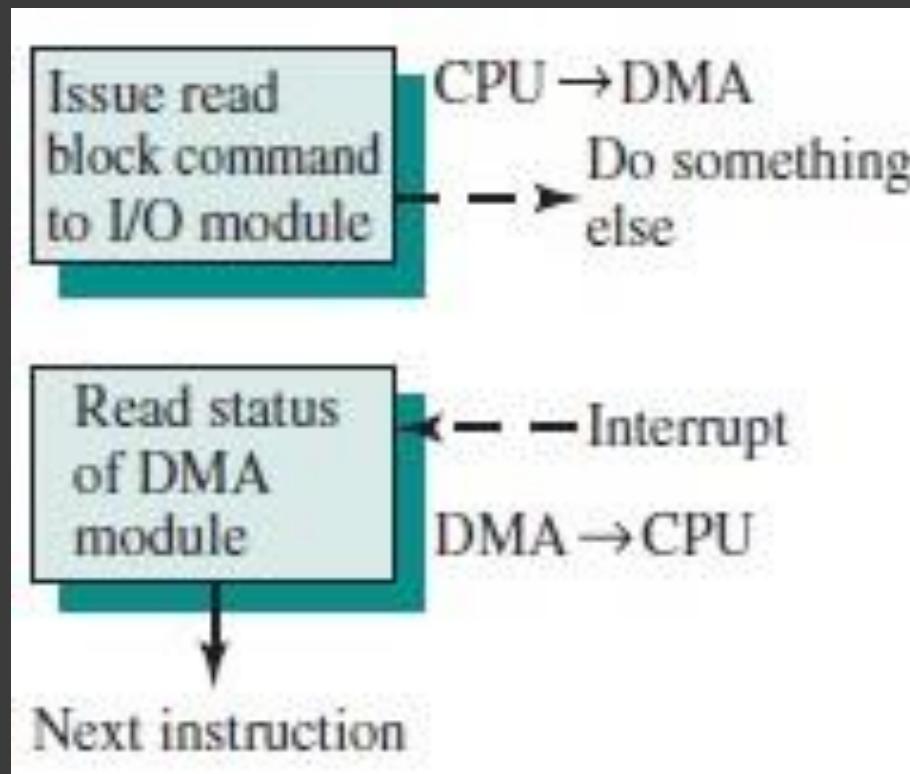
- Módulo de E/S
 - E/S manejada por interrupciones



U5 – Componentes de un computador

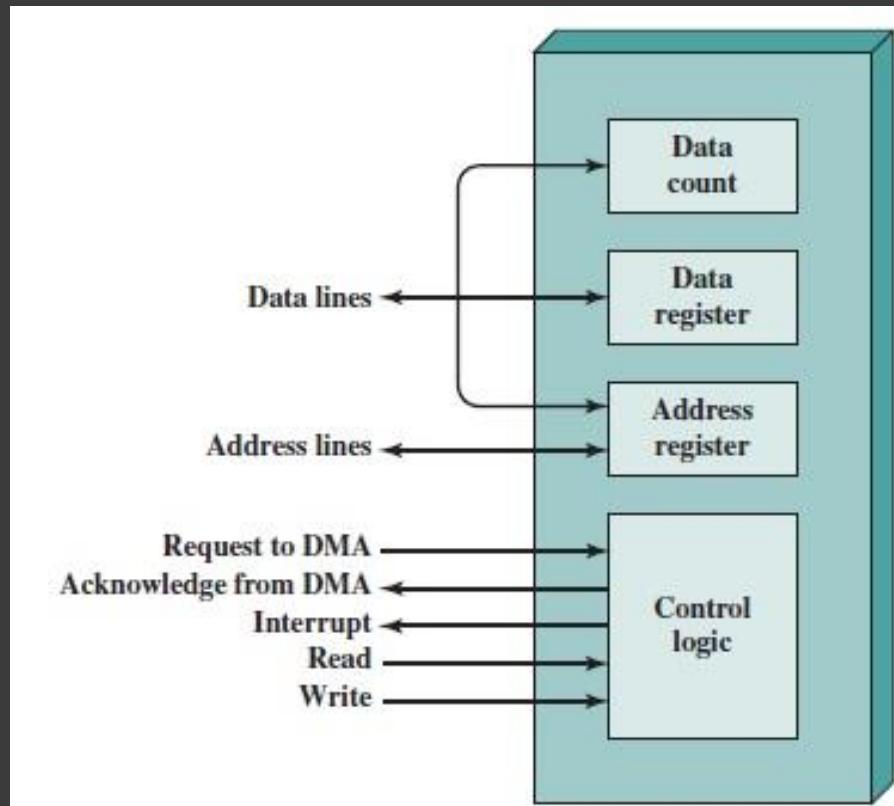
○ Módulo de E/S

- Acceso directo a memoria (DMA)



U5 – Componentes de un computador

- Módulo de E/S
 - DMA – Diagrama de bloque



U5 – Componentes de un computador

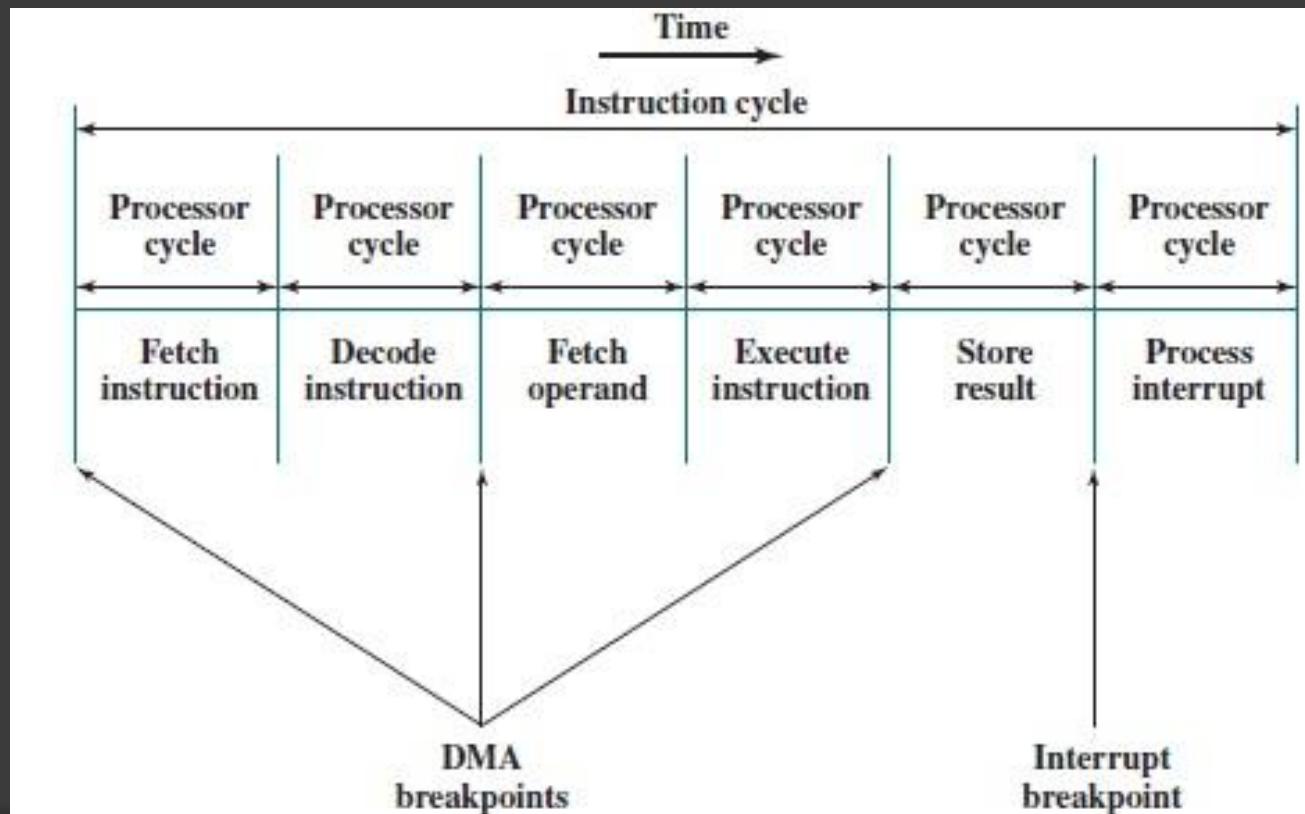
○ Módulo de E/S

- Acceso directo a memoria (DMA)
 - Información enviada por el CPU al DMA
 - Operación (READ o WRITE) vía Línea de Control
 - Dirección del dispositivo vía Línea de Dirección
 - Dirección inicial de memoria para READ o WRITE vía Línea de Datos, almacenado en el address register
 - Cantidad de palabras para READ o WRITE, vía Línea de Datos, almacenado en el data count register

U5 – Componentes de un computador

○ Módulo de E/S

- DMA – “Robo de ciclos”

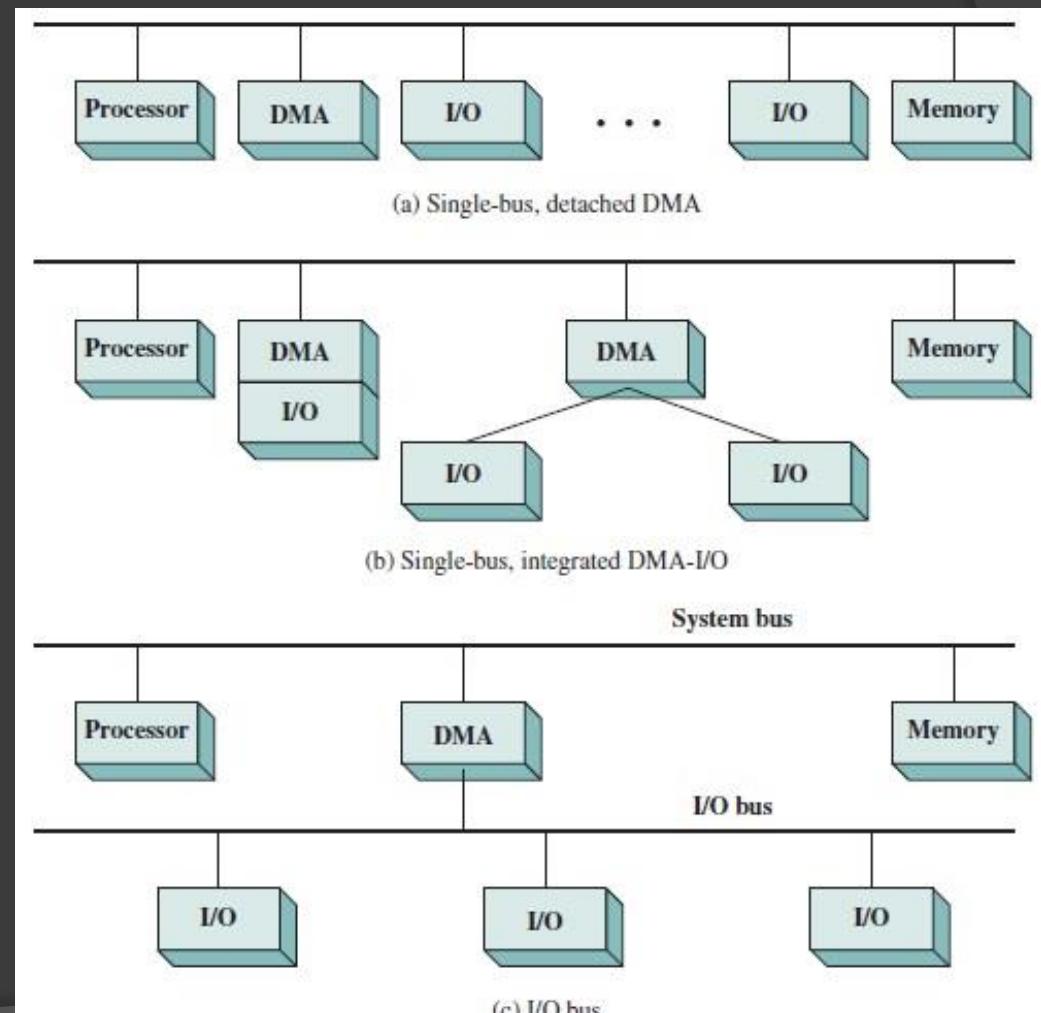


U5 – Componentes de un computador

○ Módulo de E/S

- DMA –

Topologías de configuración



U5 – Componentes de un computador

● Módulo de E/S

- Canales y procesadores de E/S
 - Canales
 - Tienen la habilidad de ejecutar instrucciones de E/S
 - La CPU principal no ejecuta instrucciones de E/S
 - Las instrucciones de E/S se almacenan en memoria principal
 - La CPU le indica al canal de E/S que inicie un programa de canal
 - Dispositivo
 - Área de memoria para storage
 - Prioridad
 - Acciones ante errores
 - Procesadores
 - Agregan a los canales memoria propia en vez de usar la memoria principal

U5 – Componentes de un computador

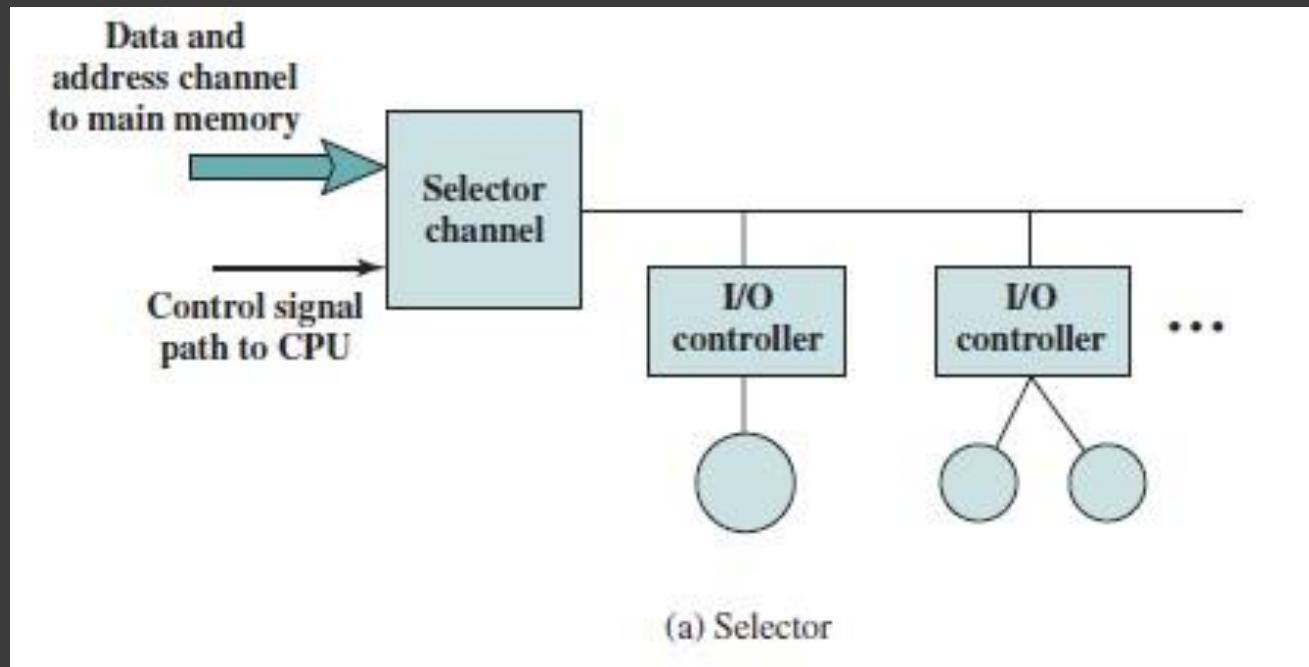
○ Módulo de E/S

- Canales y procesadores de E/S
 - Tipos de canales
 - Selectores
 - Usa un dispositivo a la vez a través de un controlador de E/S
 - Multiplexores
 - Trabaja con múltiples dispositivos a la vez (multiplexa los flujos de datos)

U5 – Componentes de un computador

○ Módulo de E/S

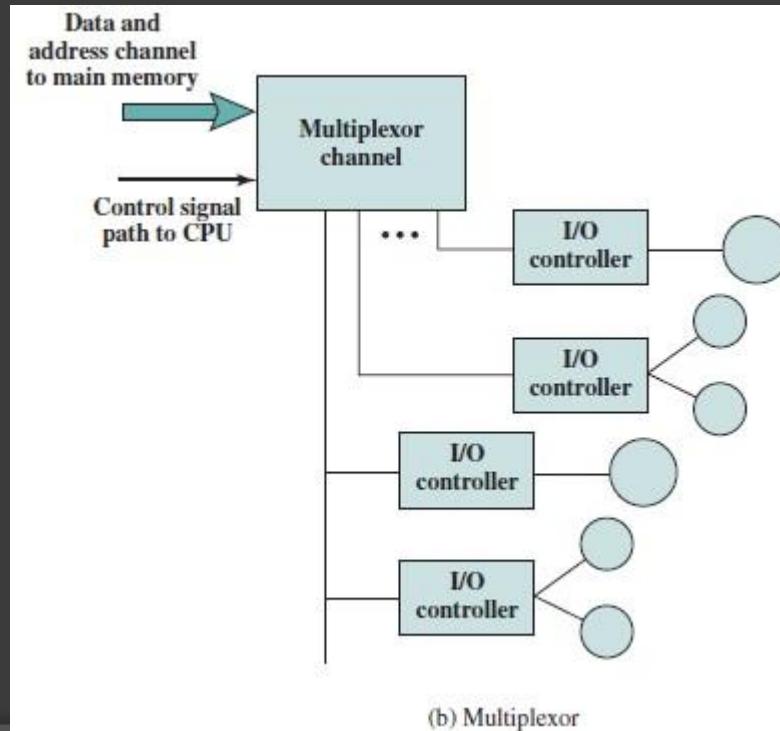
- Canales y procesadores de E/S
 - Canales selectores



U5 – Componentes de un computador

○ Módulo de E/S

- Canales y procesadores de E/S
 - Canales multiplexores



U5 – Componentes de un computador

○ Referencias

- “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings (<http://williamstallings.com/ComputerOrganization/>)
- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin (<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)

75.03 & 95.57 Organización del Computador

U5 - COMPONENTES DE UN COMPUTADOR INTERRUPCIONES

U5 – Componentes de un computador

○ Interrupciones

- ¿Qué son?

“Mecanismos por los cuales otros módulos (E/S, memoria, etc.) interrumpen el normal procesamiento del CPU”

- ¿Para qué existen?

“Para mejorar la eficiencia de procesamiento de un computador”

- Clases de interrupciones

- Hardware
 - Software

U5 – Componentes de un computador

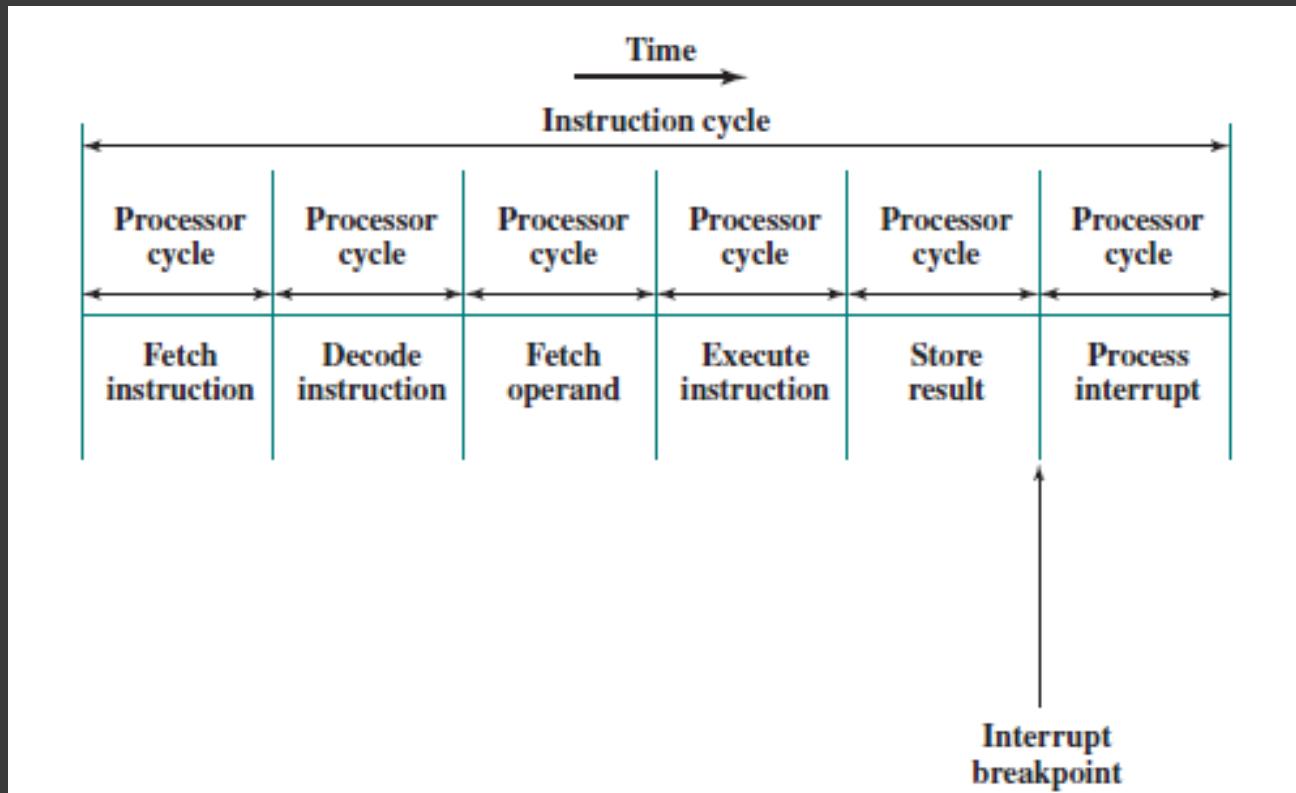
○ Interrupciones

- Clases de interrupciones
 - Hardware (asincrónicas)
 - E/S
 - Reloj (timer)
 - Fallas de hardware
 - Software
 - Excepciones de programa
 - División por cero
 - Acceso indebido a memoria
 - Overflow
 - Instrucción inválida
 - Instrucciones privilegiadas

U5 - Componentes de un computador

○ Interrupciones

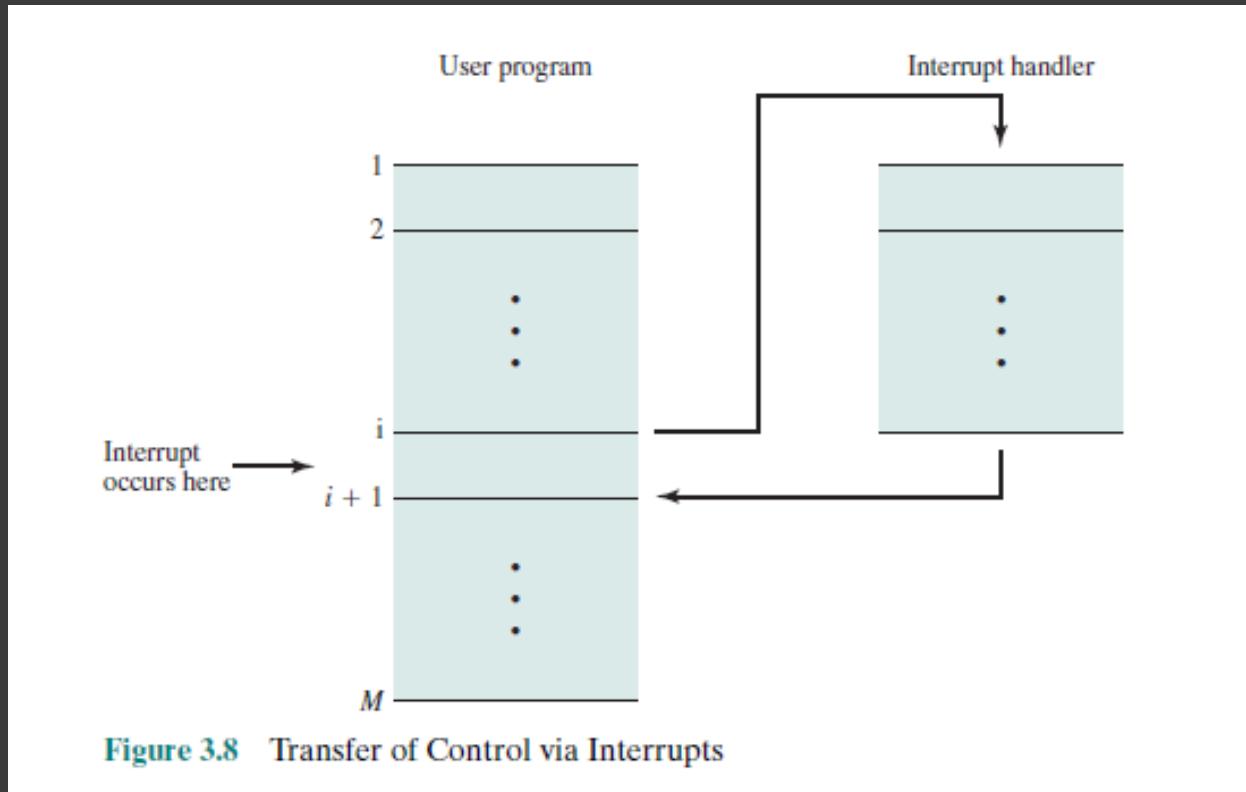
- Ciclo de instrucción



U5 – Componentes de un computador

○ Interrupciones

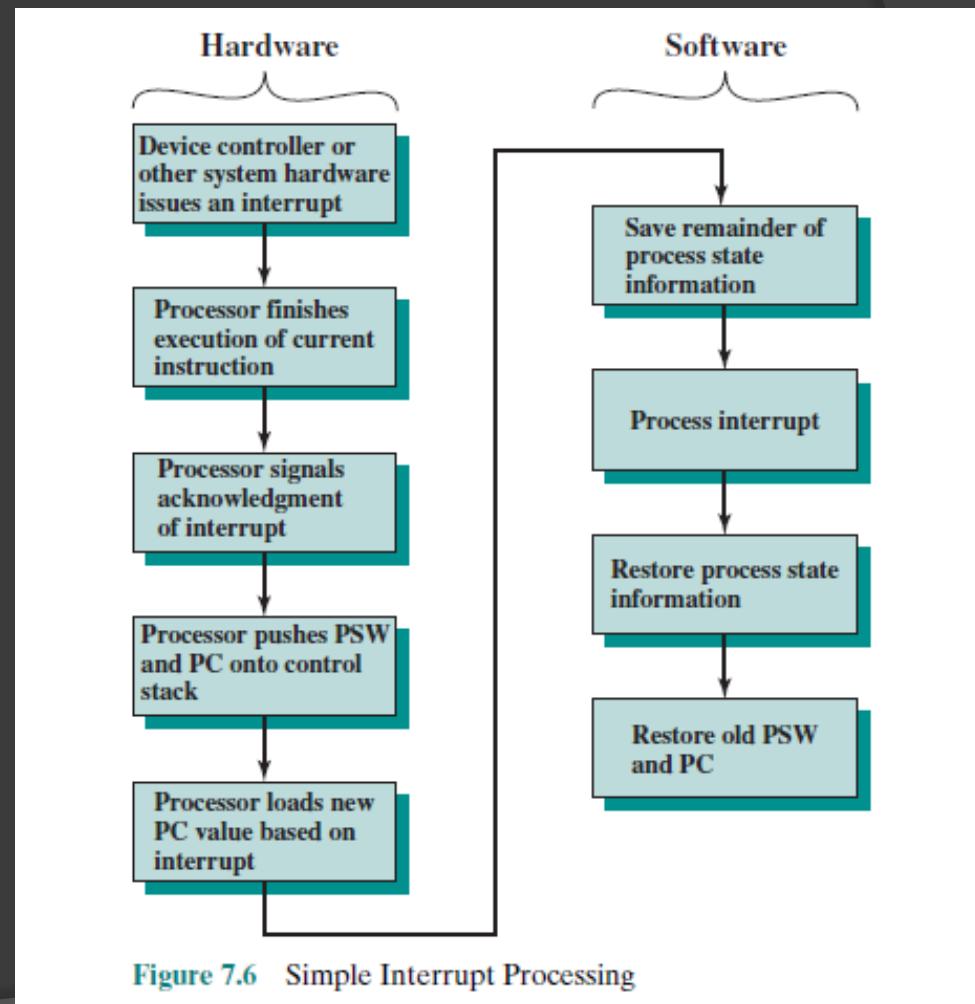
- Transferencia de control al S.O. (Handler)



U5 – Componentes de un computador

○ Interrupciones

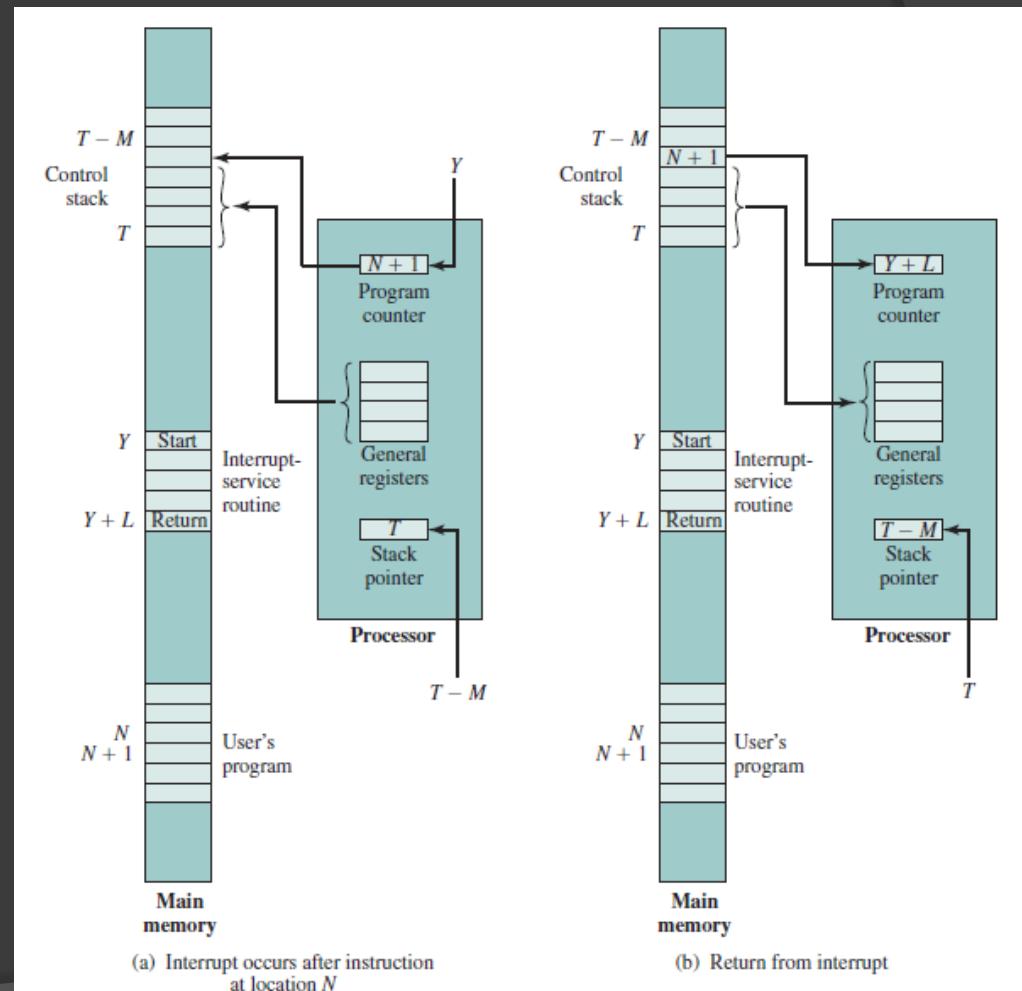
- Procesamiento de interrupciones



U5 – Componentes de un computador

○ Interrupciones

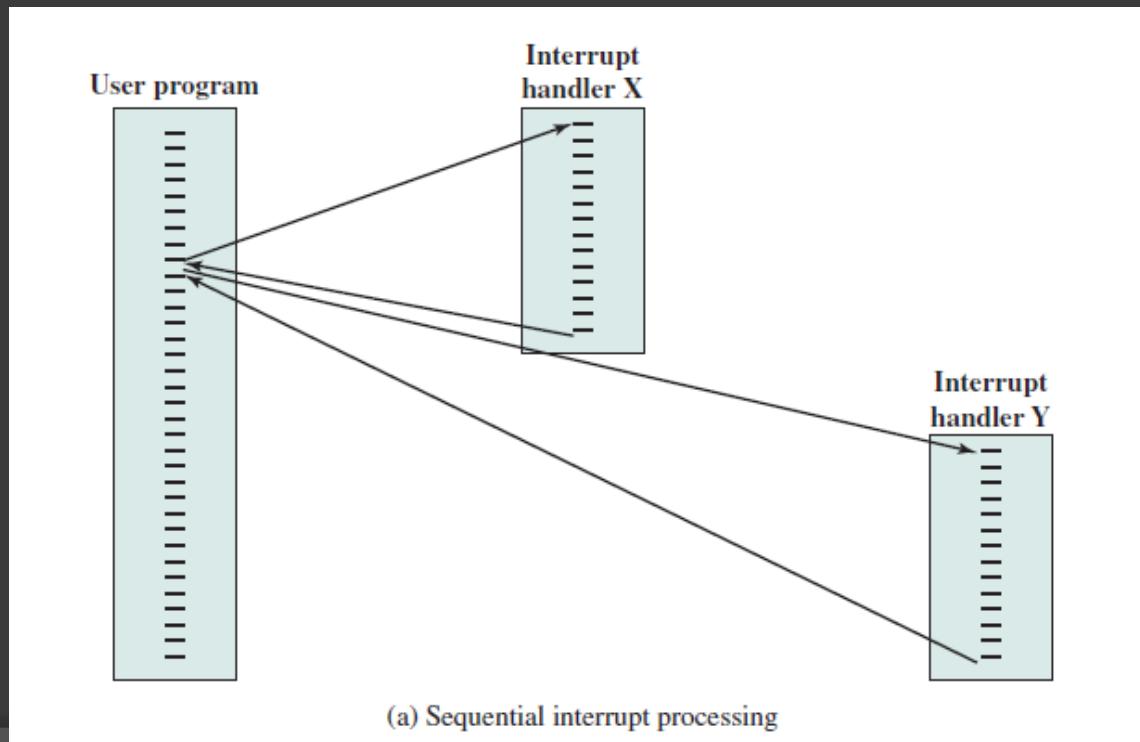
- Procesamiento de Interrupciones (ejemplo)



U5 - Componentes de un computador

○ Interrupciones

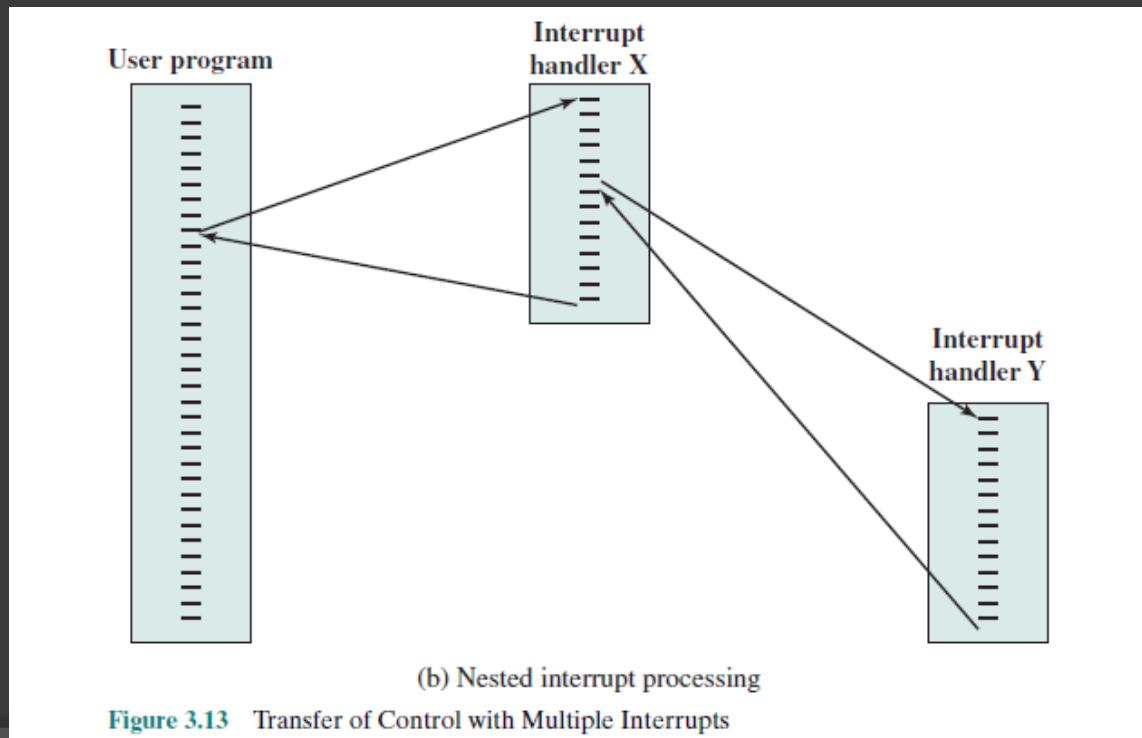
- Múltiples interrupciones
 - Deshabilitar interrupciones (secuencia)



U5 – Componentes de un computador

○ Interrupciones

- Múltiples interrupciones
 - Priorizar interrupciones (anidadas)



U5 – Componentes de un computador

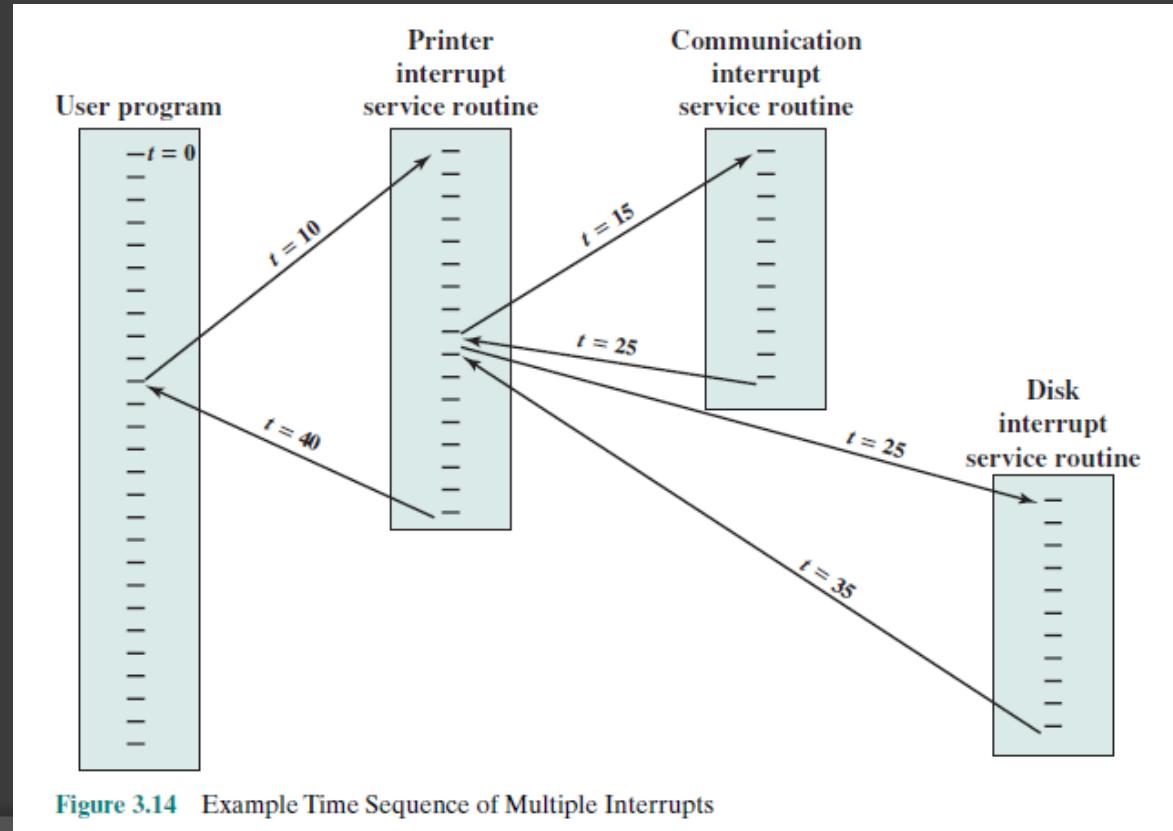
○ Interrupciones

- Múltiples interrupciones (ejemplo)
 - Tres dispositivos de E/S
 - Línea de comunicación (Prioridad 1)
 - Disco (Prioridad 2)
 - Impresora (Prioridad 3)
 - Eventos
 - $T=10$ – Interrupción de Impresora
 - $T=15$ – Interrupción de línea de comunicación
 - $T=20$ – Interrupción de disco

U5 - Componentes de un computador

○ Interrupciones

- Múltiples interrupciones (ejemplo)



U5 – Componentes de un computador

○ Referencias

- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin
(<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)
- “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings
(<http://williamstallings.com/ComputerOrganization/>)

75.03 Organización del Computador

U5 - COMPONENTES DE UN COMPUTADOR MEMORIA

U5 – Componentes de un computador

○ Memoria

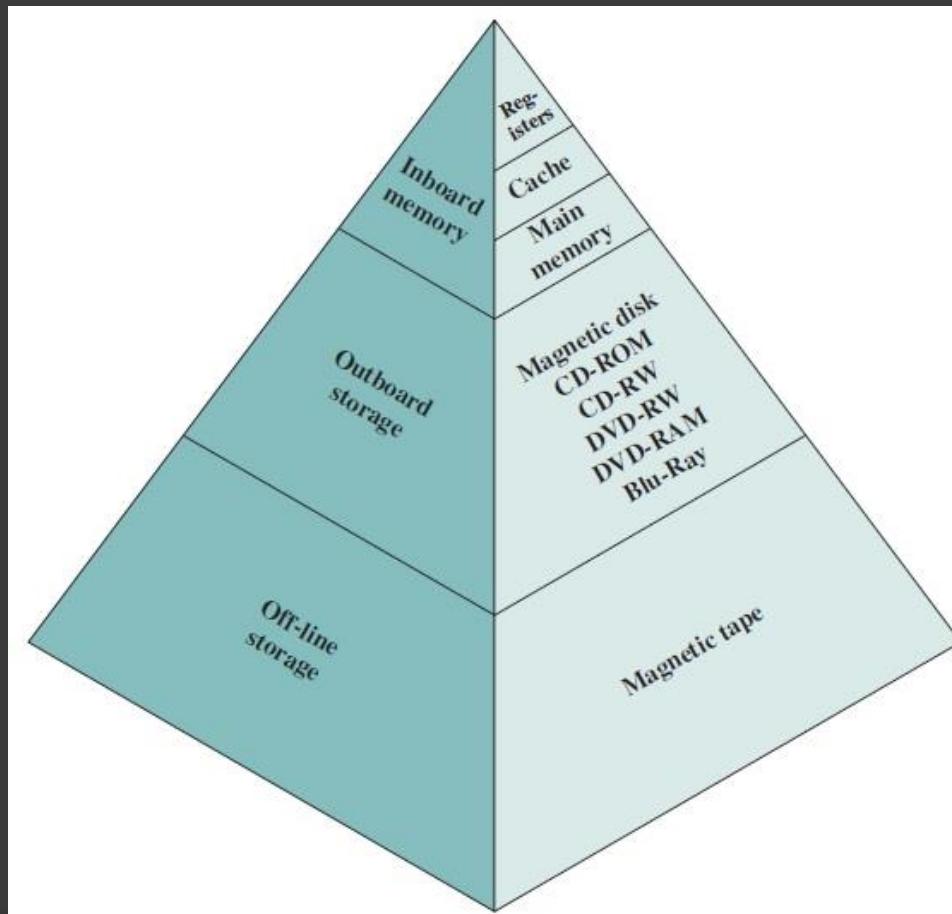
- Componente complejo (*Sistema de memoria*)
- Formado por elementos con distintas cualidades:
 - Tecnología
 - Organización
 - Performance
 - Costo
- Jerarquía de subsistemas de memoria
 - Internos al sistema (accedidos directamente por el procesador)
 - Externos al sistema (accedidos por el procesador a través de un módulo de E/S)

U5 – Componentes de un computador

- Jerarquía de memoria
 - Tres características a tener en cuenta
 - Capacidad
 - Tiempo de acceso
 - Costo
 - Subsistemas de memoria con relación de compromiso entre estas características => No se usa un solo componente de memoria

U5 – Componentes de un computador

○ Jerarquía de memoria



U5 – Componentes de un computador

- Jerarquía de memoria
 - A medida que se baja de la pirámide:
 - Costo por bit decreciente
 - Capacidad creciente
 - Tiempo de acceso creciente
 - Frecuencia de acceso de la memoria por parte de procesador decreciente

U5 – Componentes de un computador

○ Sistema de memoria

- Características

- Locación

- Interna

- Registros

- Memoria interna para unidad de control

- Memoria Cache

- Externa

- Dispositivos de almacenamiento periféricos
(discos, cintas, etc.)

U5 – Componentes de un computador

○ Sistema de memoria

- Características

- Capacidad

- Bytes / Palabras (memoria interna)
 - Bytes (memoria externa)

- Unidad de transferencia

- Número de líneas eléctricas del módulo de memoria, típicamente el tamaño de palabra o 64, 128 o 256 bits (memoria interna)
 - Bloques (memoria externa)

U5 – Componentes de un computador

- Sistema de memoria
 - Características
 - Métodos de acceso de unidades de datos
 - Acceso secuencial
 - Unidades de datos: registros (records)
 - Acceso lineal en secuencia
 - Se deben pasar y descartar todos los registros intermedios antes de acceder al registro deseado
 - Tiempo de acceso variable
 - Ej. cintas magnéticas

U5 – Componentes de un computador

- Sistema de memoria
 - Características
 - Métodos de acceso de unidades de datos
 - Acceso directo
 - Dirección única para bloques o registros basada en su posición física
 - Tiempo de acceso variable
 - Ej. discos magnéticos

U5 – Componentes de un computador

- Sistema de memoria
 - Características
 - Métodos de acceso de unidades de datos
 - Acceso aleatorio
 - Cada posición direccionable de memoria tiene un mecanismo de direccionamiento cableado físicamente
 - Tiempo de acceso constante, independiente de la secuencia de accesos anteriores
 - Ej. memoria principal y algunas memorias cache

U5 – Componentes de un computador

○ Sistema de memoria

● Características

○ Métodos de acceso de unidades de datos

- Acceso asociativo
 - Tipo de acceso aleatorio por comparación de patrón de bits
 - La palabra se busca por una porción de su contenido en vez de por su dirección
 - Cada posición de memoria tiene un mecanismo de direccionamiento propio
 - Tiempo de acceso constante, independiente de la secuencia de accesos anteriores o su ubicación
 - Ej. memorias cache

U5 – Componentes de un computador

○ Sistema de memoria

• Características

○ Parámetros de performance

- Tiempo de acceso (latencia)
 - Memorias de acceso aleatorio: tiempo necesario para hacer una operación de lectura o escritura
 - Memorias sin acceso aleatorio: tiempo necesario para posicionar el mecanismo de lectura/escritura en la posición deseada
- Tiempo de ciclo de memoria
 - Memorias de acceso aleatorio: tiempo de acceso más el tiempo adicional necesario para que una nueva operación pueda comenzar

U5 – Componentes de un computador

○ Sistema de memoria

- Características
 - Parámetros de performance
 - Tasa de transferencia
 - Tasa con la cual los datos son transferidos dentro o fuera de la unidad de memoria
 - Memorias de acceso aleatorio: $1/\text{Tiempo de ciclo de memoria}$
 - Memorias sin acceso aleatorio:

$$T_n = T_A + n/R$$

donde

T_n = Tiempo promedio para leer o escribir n bits

T_A = Tiempo promedio de acceso

n = Número de bits

R = Tasa de transferencia, en bits por segundo (bps)

U5 – Componentes de un computador

- Sistema de memoria
 - Características
 - Tipos físicos
 - Memorias semiconductoras (memoria principal y cache)
 - Memorias de superficie magnética (discos y cintas)
 - Memorias ópticas (medios ópticos)

U5 – Componentes de un computador

○ Sistema de memoria

• Características

○ Características físicas

- Memorias volátiles: se pierde su contenido ante la falta de energía eléctrica (Ej. algunas memorias semiconductoras)
- Memorias no volátiles: no se necesita de energía eléctrica para mantener su contenido (Ej. memorias de superficie magnéticas y algunas memorias semiconductoras)
- Memorias de solo lectura: (ROM – Read Only Memory) no se puede borrar su contenido (Ej. algunas memorias semiconductoras)

U5 – Componentes de un computador

○ Sistema de memoria

- Memorias semiconductoras (acceso aleatorio)

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)				
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level	Electrically	Nonvolatile
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

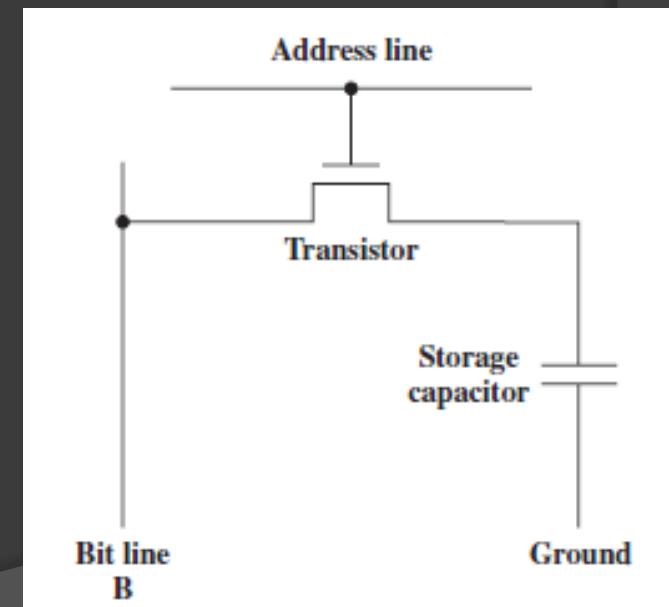
U5 – Componentes de un computador

○ Sistema de memoria

- Memorias RAM

- Dynamic RAM (DRAM)

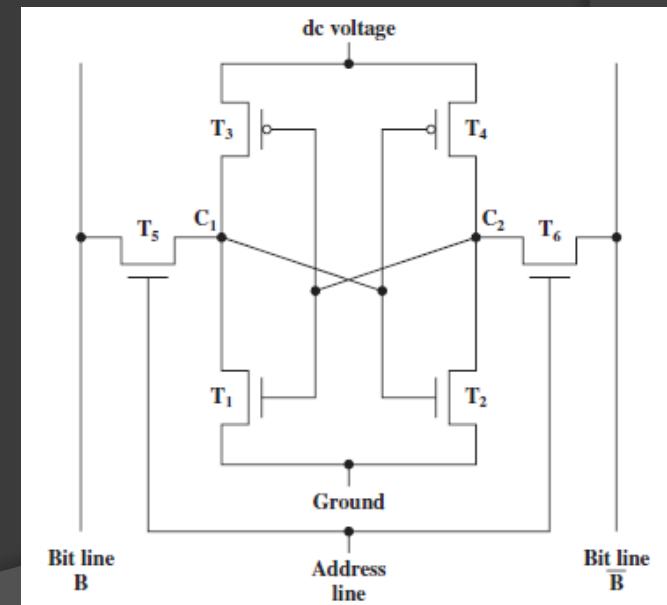
- Celdas que almacenan datos como carga en capacitores (bit 0 o 1 si hay presencia o ausencia de carga en el capacitor)
 - Requieren refrescar su carga periódicamente para mantener los datos almacenados
 - Se suelen usar en memoria principal



U5 – Componentes de un computador

○ Sistema de memoria

- Memorias RAM
 - Static RAM (SRAM)
 - Los valores binarios se almacenan usando compuertas lógicas flip-flop
 - Son más complejas y grandes que las DRAM
 - Son más rápidas que las DRAM
 - Se suelen usar en memoria cache



U5 – Componentes de un computador

○ Sistema de memoria

- Memorias RAM
 - Tipos de DRAM
 - SDRAM (Synchronous DRAM)
 - Intercambia datos con el CPU en forma sincronizada vía el reloj del sistema
 - Evita los estados de espera del CPU
 - DDR DRAM (Double data rate DRAM)
 - Incrementa la tasa de transferencia con
 - Usa un esquema de buffering

	DDR1	DDR2	DDR3	DDR4
Prefetch buffer (bits)	2	4	8	8
Voltage level (V)	2.5	1.8	1.5	1.2
Front side bus data rates (Mbps)	200–400	400–1066	800–2133	2133–4266

U5 – Componentes de un computador

- Jerarquía de memoria
 - Principio de localidad de referencia
 - “Durante la ejecución de un programa, las referencias a memoria que hace el procesador tanto para instrucciones como datos tienden a estar agrupadas”
(Ej. loops, subrutinas, tablas, vectores)

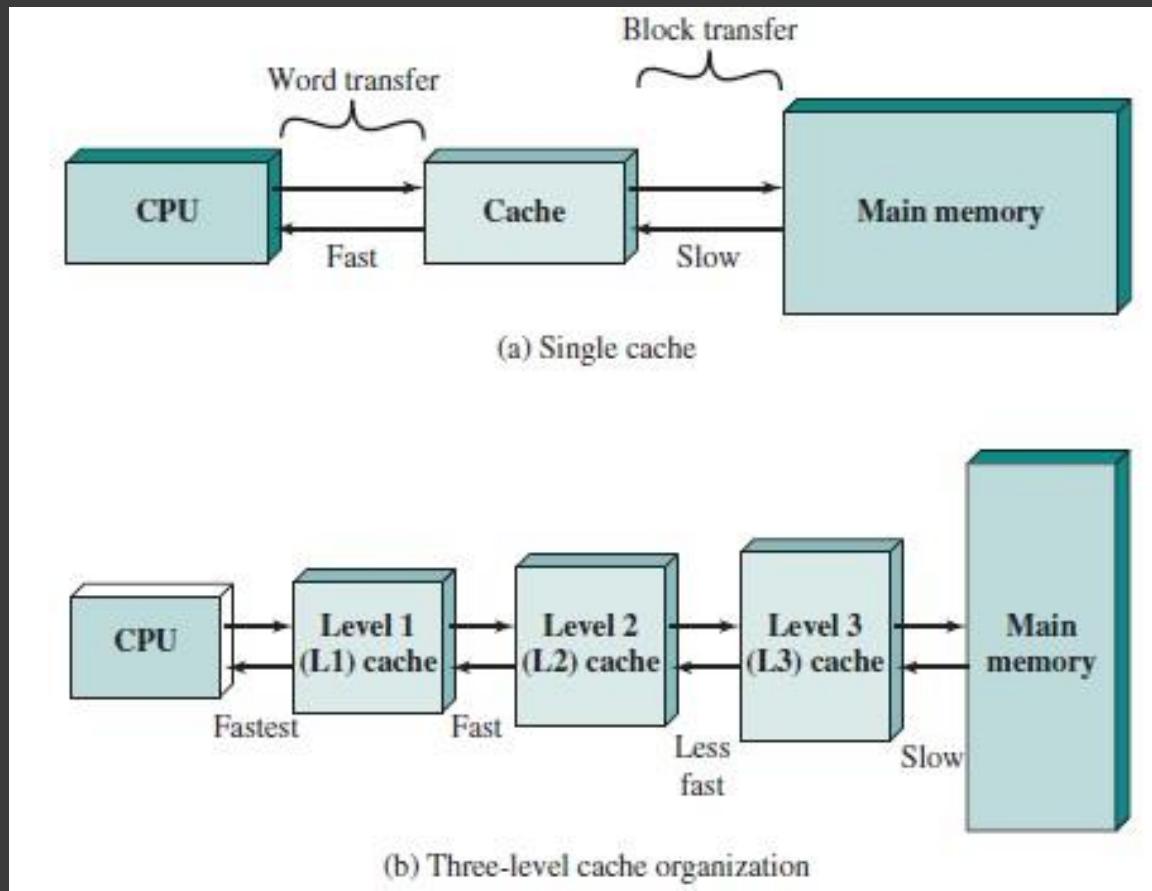
U5 – Componentes de un computador

○ Memoria Cache

- Memoria semiconductora más rápida (y costosa) que la principal
- Se ubica entre el procesador y la memoria principal
- Permite mejorar la performance general de acceso a memoria principal
- Contiene una copia de porciones de memoria principal

U5 – Componentes de un computador

○ Memoria Cache



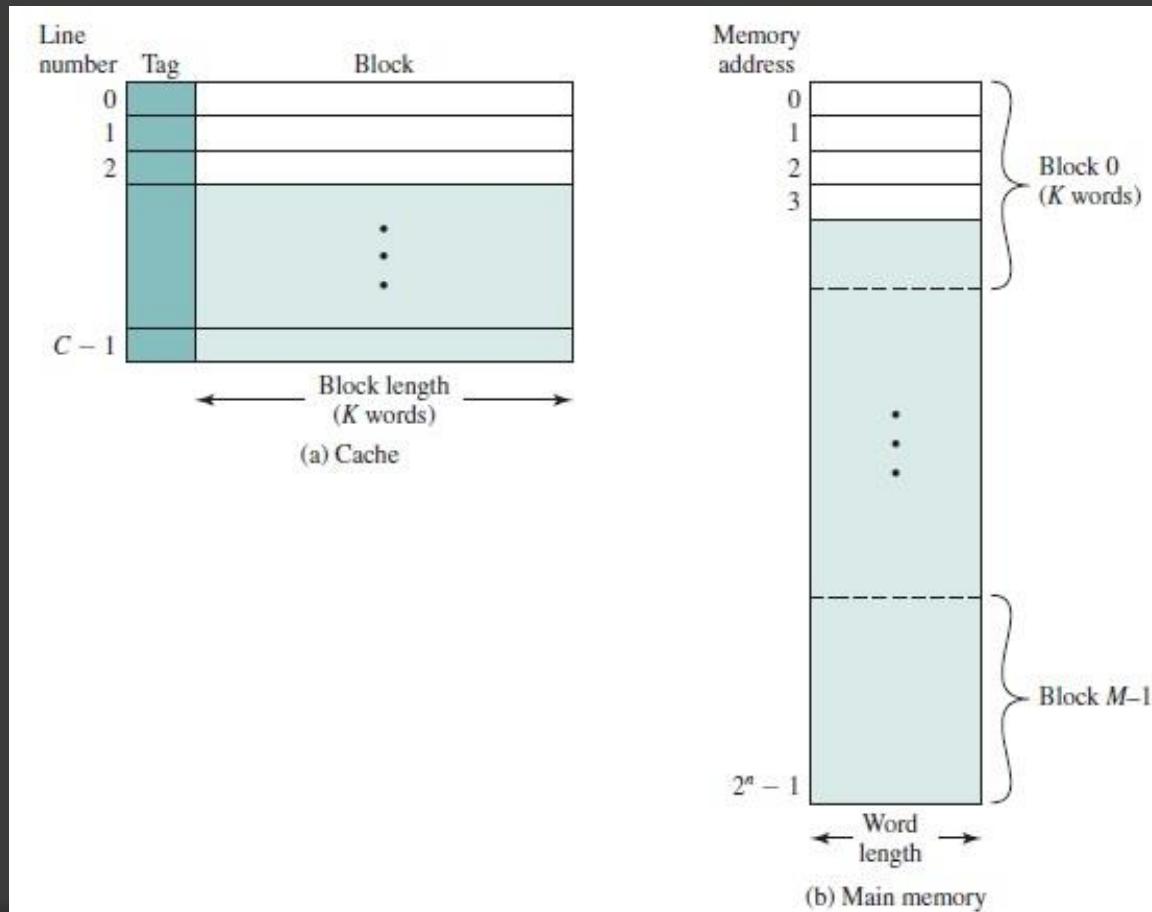
U5 – Componentes de un computador

○ Memoria Cache

- Cómo funciona
 - CPU trata de leer una palabra de la memoria principal
 - Se chequea primero si existe en la memoria cache.
 - Si es así se la entrega al CPU
 - Sino se lee un bloque de memoria principal (número fijo de palabras), se incorpora a la cache y la palabra buscada se entrega al CPU
 - Por el principio de localidad de referencia es probable que próximas palabras buscadas estén dentro del bloque de memoria subido a la cache

U5 – Componentes de un computador

○ Memoria Cache



U5 – Componentes de un computador

● Memoria Cache

- Estructura sistema cache/memoria principal
 - Memoria principal
 - 2^n palabras direccionables (dirección única de n -bits para cada una)
 - Bloques fijos de K palabras cada uno (M bloques)
 - Cache
 - m bloques llamados líneas
 - Cada línea contiene:
 - K palabras
 - Tag (conjunto de bits para indicar qué bloque está almacenado, usualmente una porción de la dirección de memoria principal)
 - Bits de control (Ej. bit para indicar si la línea se modificó desde la última vez que se cargó en la cache)

U5 – Componentes de un computador

○ Memoria Cache

- Ejemplos (1/2)

Table 4.3 Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache ^a	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16–32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128–256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256–512 kB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 kB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—

Notes:

^a Two values separated by a slash refer to instruction and data caches.

^b Both caches are instruction only; no data caches.

U5 – Componentes de un computador

○ Memoria Cache

- Ejemplos (2/2)

Table 4.3 Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache ^a	L2 Cache	L3 Cache
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA ^b	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6×32 kB/ 32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ server	2011	24×64 kB/ 128 kB	24×1.5 MB	24 MB L3 192 MB L4

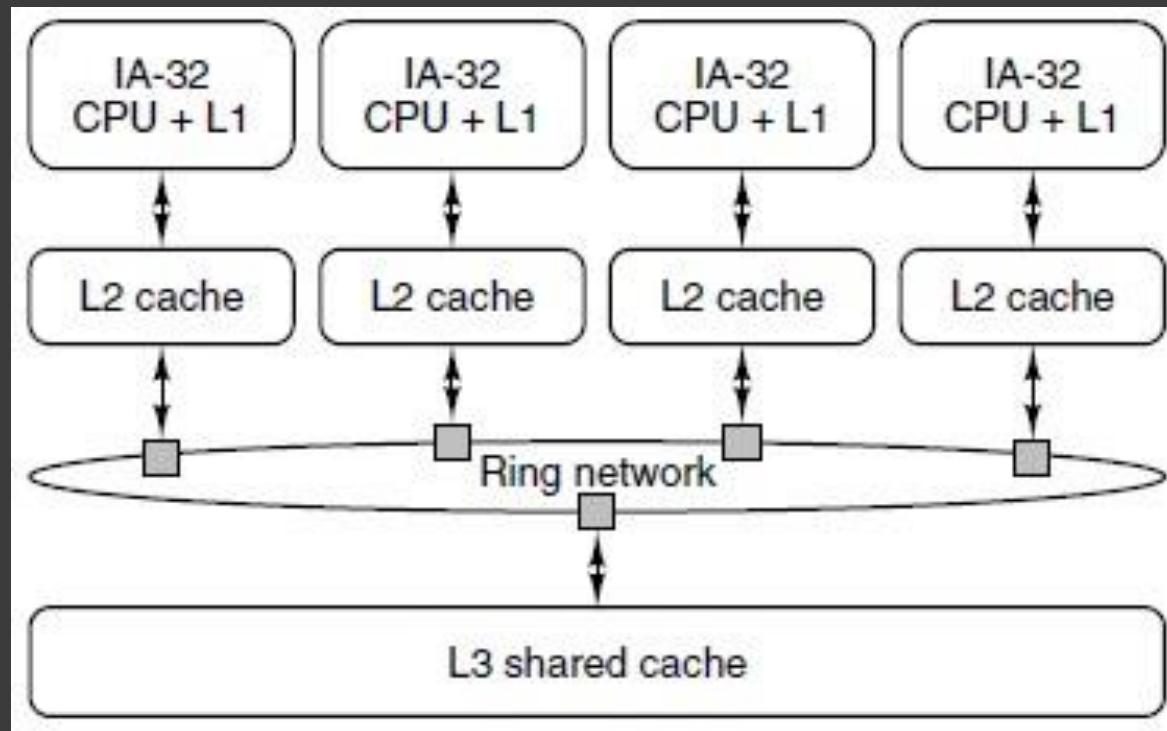
Notes:

^a Two values separated by a slash refer to instruction and data caches.

^b Both caches are instruction only; no data caches.

U5 – Componentes de un computador

- Memoria Cache
 - Intel Core i7



U5 – Componentes de un computador

○ Referencias

- “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings (<http://williamstallings.com/ComputerOrganization/>)
- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin (<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)

75.03 & 95.57 Organización del Computador

U5 - COMPONENTES DE UN COMPUTADOR PROCESADOR

U5 – Componentes de un computador

○ Procesador

- Arquitectura de procesadores
 - Historia
 - Primero orientado al hardware (hasta los '70)
 - Luego orientado al software (a partir de los '80)
 - CISC vs RISC

U5 – Componentes de un computador

- ⦿ CISC (Complex Instruction Set Computer)
 - Pocos registros de procesador (especializados)
 - Set de Instrucciones amplio
 - Muchas instrucciones para trabajar con memoria
 - Microarquitectura en software/hardware compleja
 - Instrucciones complejas (más de un ciclo de reloj)
 - Varios modos de direccionamiento
 - Muchos tipos de datos
 - Muchos formatos de instrucción (variables o híbridos)
 - Orientado al hardware, compiladores relativamente simples (tamaño de código pequeño)
 - Ejemplos: VAX, Intel x86 (hasta IA-32), Intel-64, IBM Mainframe, Motorola 68k

U5 – Componentes de un computador

- RISC (Reduced Instruction Set Computer)
 - Muchos registros de procesador de uso general
 - Set de Instrucciones pequeño
 - Solo acceso a memoria a través de LOAD/STORE
 - Microarquitectura en hardware simple
 - Instrucciones simples (un ciclo de reloj)
 - Pocos modos de direccionamiento
 - Pocos tipos de datos
 - Pocos formatos de instrucción (fijos)
 - Orientado al software, compiladores relativamente complejos (tamaño de código largo)
 - Ejemplos: SPARC, MIPS, ARM, Intel Itanium (IA-64)

U5 – Componentes de un computador

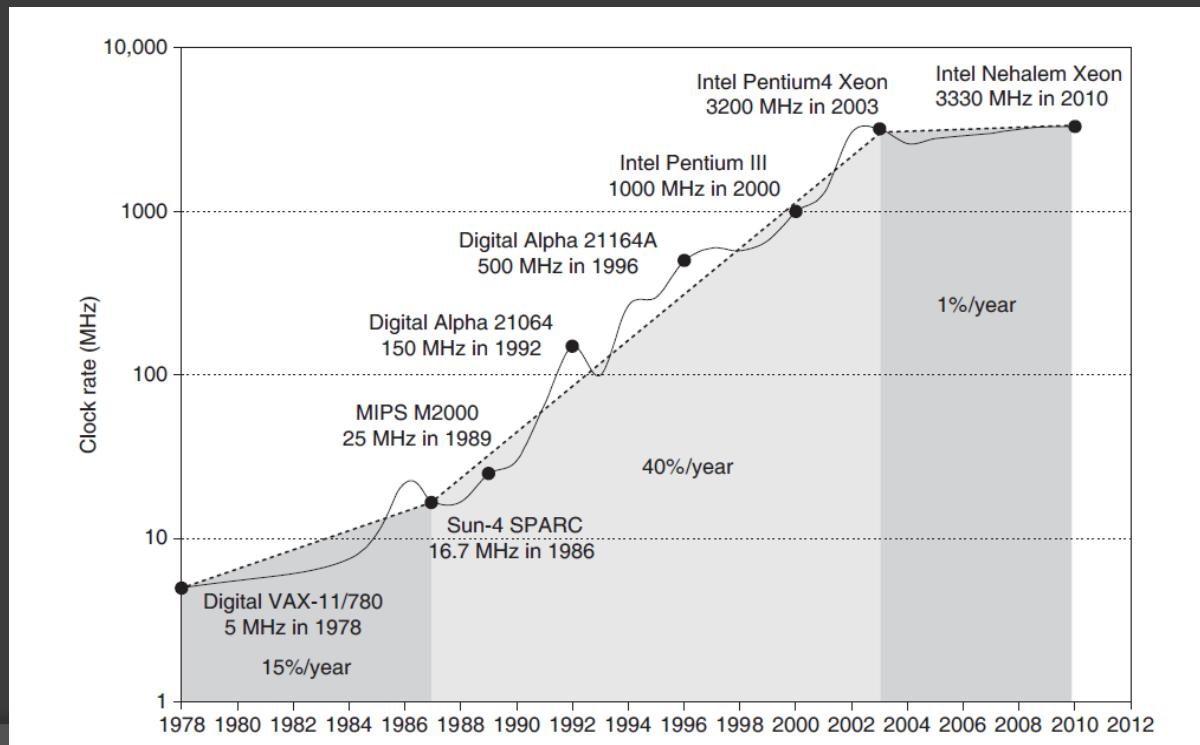
○ Procesador

- Arquitectura de procesadores
 - Ecuación de Performance
MIPS rate = Frecuencia del reloj en MHz (f) * Instrucciones por ciclo (IPC)
 - Uniprocesadores
 - Taxonomía de Flynn
 - SISD (Single Instruction Single Data) (Uniprocesadores)
 - SIMD (Single Instruction Multiple Data)
 - MISD (Multiple Instruction Single Data) (No comercial)
 - MIMD (Multiple Instruction Multiple Data)
 - Limitación de la velocidad del reloj (calor)

U5 – Componentes de un computador

○ Procesador

- Arquitectura de procesadores
 - Velocidad de reloj (uniprocesadores)



U5 – Componentes de un computador

○ Procesador

- Arquitectura de procesadores
 - Paralelismo
 - Técnicas
 - A nivel instrucción
 - Pipelining
 - Dual pipelining
 - Superscalar
 - Multithreading
 - A nivel procesador
 - Procesadores paralelos de datos
 - Multiprocesadores
 - Multicomputadores

U5 – Componentes de un computador

○ Procesador

- Paralelismo

- A nivel instrucción

- Pipelining (Stages)

- Solapa la ejecución de las instrucciones para reducir el tiempo total de una secuencia de instrucciones

- Ejecuta una instrucción por ciclo de reloj

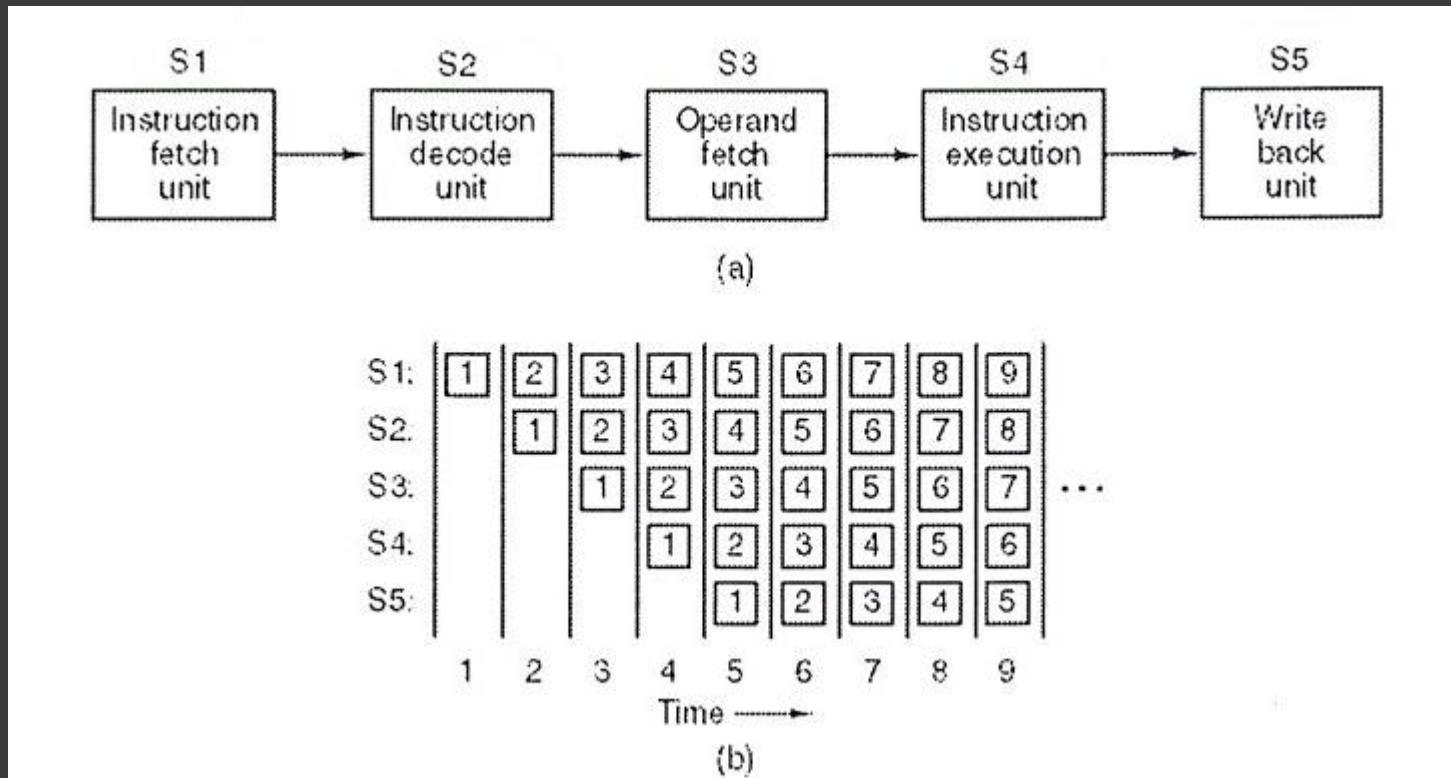
- Control de dependencia entre las instrucciones (compilador o hardware)

- Ej. Intel 486

U5 - Componentes de un computador

○ Procesador

- Pipelining (Stages)



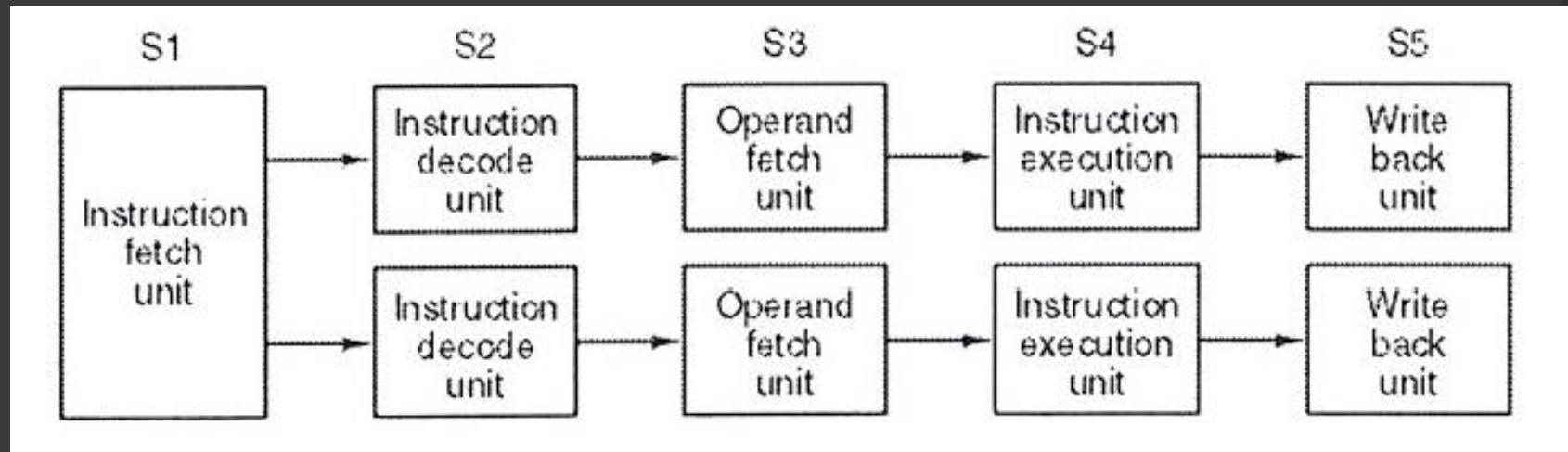
U5 – Componentes de un computador

○ Procesador

- Paralelismo
 - A nivel instrucción
 - Dual Pipelining
 - Ejecuta dos instrucciones por ciclo de reloj
 - Ej. Intel Pentium

U5 – Componentes de un computador

- Procesador
 - Dual Pipelining



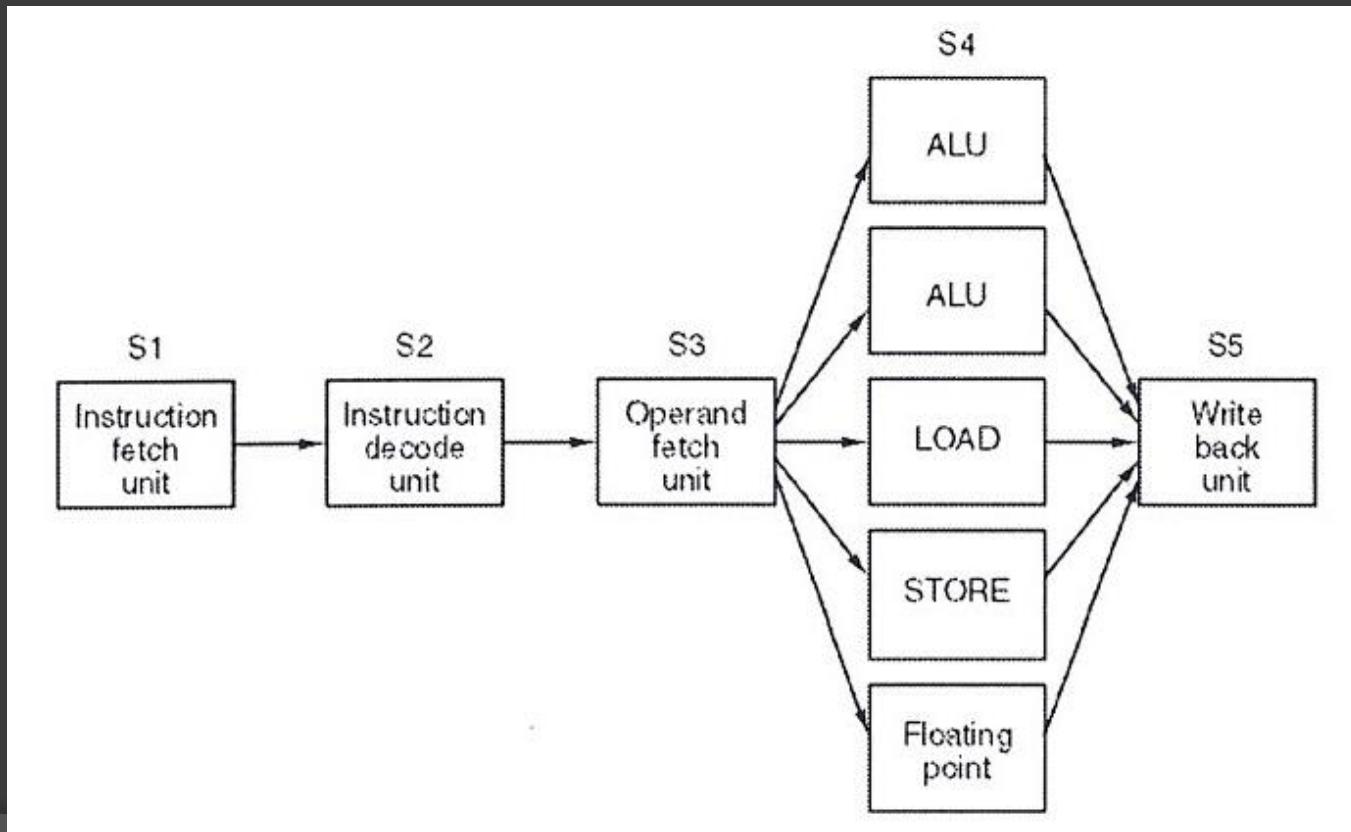
U5 – Componentes de un computador

○ Procesador

- Paralelismo
 - A nivel instrucción
 - Superscalar (múltiples unidades funcionales)
 - Ejecuta más de una instrucción por ciclo de reloj
 - N-way / N-issue (N entre 3 y 6)
 - Ej. Intel Core

U5 – Componentes de un computador

- Procesador
 - Superescalar



U5 – Componentes de un computador

- Procesador
 - Paralelismo
 - A nivel instrucción
 - Hardware multithreading
 - Busca incrementar el uso del CPU intercambiando la ejecución entre threads (hilos de ejecución) cuando uno está frenado por alguna causa
 - Thread: Contiene un PC, un conjunto de registros y la pila (stack). Comparten un mismo address space. Se los conoce como “lightweight processes”
 - Proceso: Puede tener uno o más threads, contiene un address space y un estado gestionado por el S.O.
 - El cambio de contexto entre threads es “liviano” (en un mismo ciclo de reloj) en comparación con los procesos, que requieren del S.O.

U5 – Componentes de un computador

- Procesador
 - Paralelismo
 - A nivel instrucción
 - Hardware multithreading
 - Fine-grained multithreading: se intercambia el uso del procesador entre threads luego de la ejecución de cada instrucción.
Ej. Procesadores Intel IA-32
 - Coarse-grained multithreading: se intercambia el uso del procesador entre threads solo luego de algún evento significativo, como puede ser un page fault o un “cache miss”. En este último caso se cambia la ejecución a otro thread en vez de esperar el acceso más lento a la memoria principal.
Ej. Intel Itanium 2

U5 – Componentes de un computador

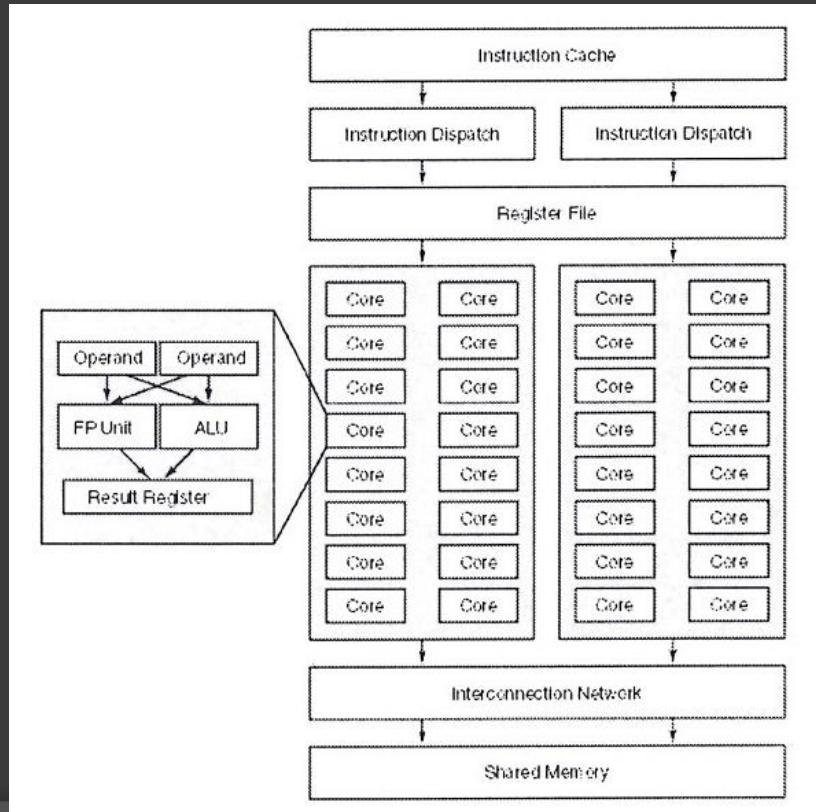
○ Procesador

- Paralelismo
 - A nivel procesador
 - Procesadores paralelos de datos
 - Una sola unidad de control
 - Múltiples procesadores
 - Métodos
 - SIMD – Single Instruction Multiple Data
 - Múltiples procesadores ejecutan la misma secuencia de pasos sobre un conjunto diferente de datos
 - Ej. GPU (Nvidia Fermi GPU)
 - Vectoriales
 - Similar a SIMD
 - Registro vectorial: conjunto de registros convencionales que se cargan desde memoria en una sola instrucción.
 - Se opera por pipelining
 - Ej. Intel Core (SSE – Streaming SIMD Extension)

U5 – Componentes de un computador

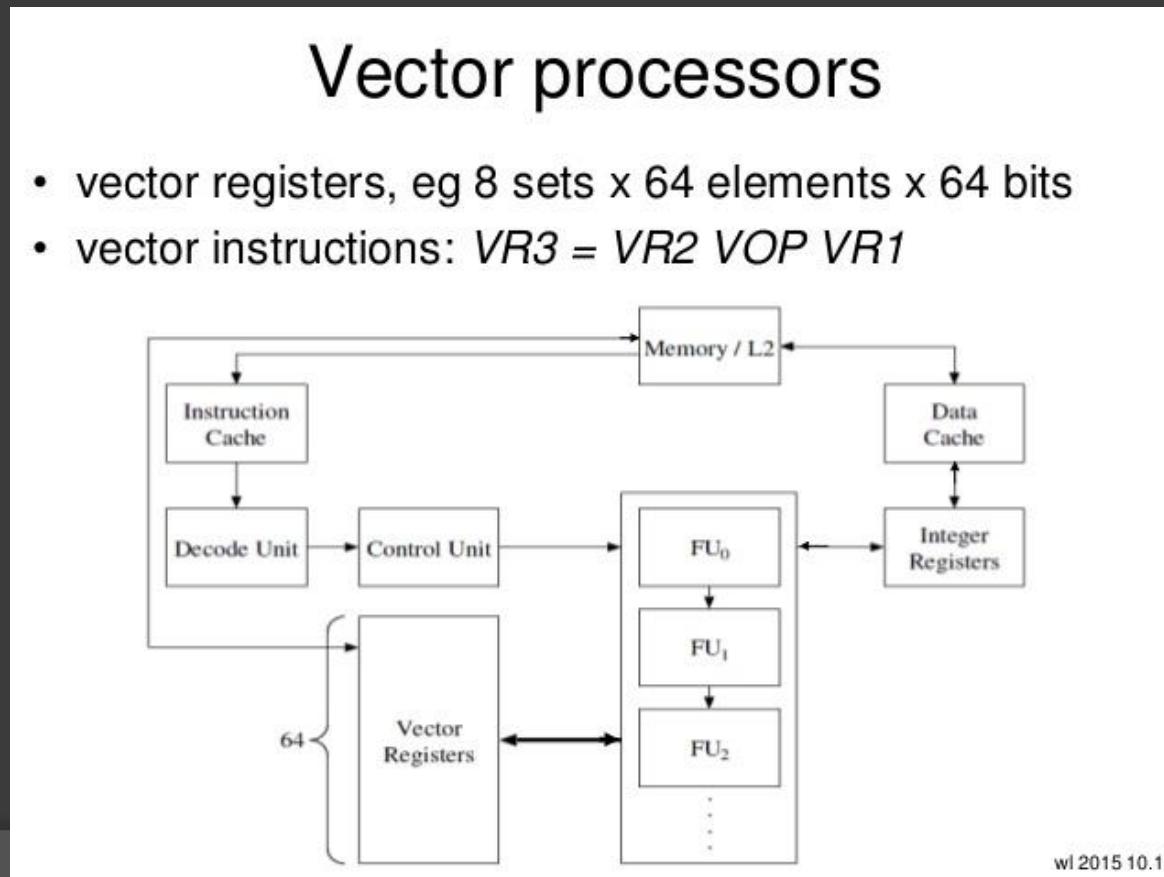
○ Procesador

- SIMD – Single Instruction Multiple Data



U5 - Componentes de un computador

- Procesador
 - Vectoriales



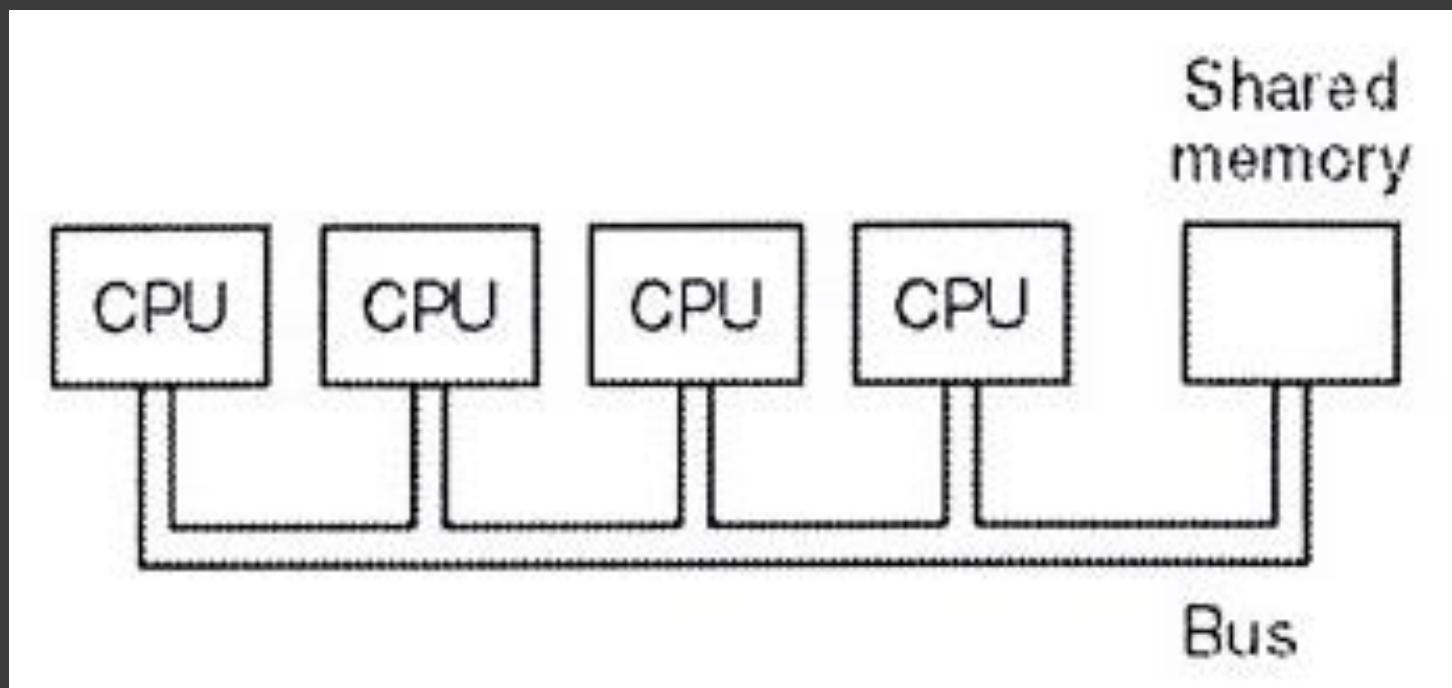
U5 – Componentes de un computador

- Procesador
 - Paralelismo
 - A nivel procesador
 - Multiprocesadores
 - Múltiples CPUs que comparten memoria común
 - MIMD (Multiple Instruction Multiple data)
 - CPUs fuertemente acoplados
 - Diferentes implementaciones
 - Single bus y memoria compartida (centralizada) (UMA – Uniform memory access) (SMP – Symmetric multiprocessor)
 - Ej. Intel Core i7
 - CPUs con memoria local y memoria compartida (NUMA – non-uniform memory access)
 - Ej. BBN Butterfly, SGI Origin 2000, Compaq AlphaServer GS320, Intel Itanium 2

U5 – Componentes de un computador

○ Procesador

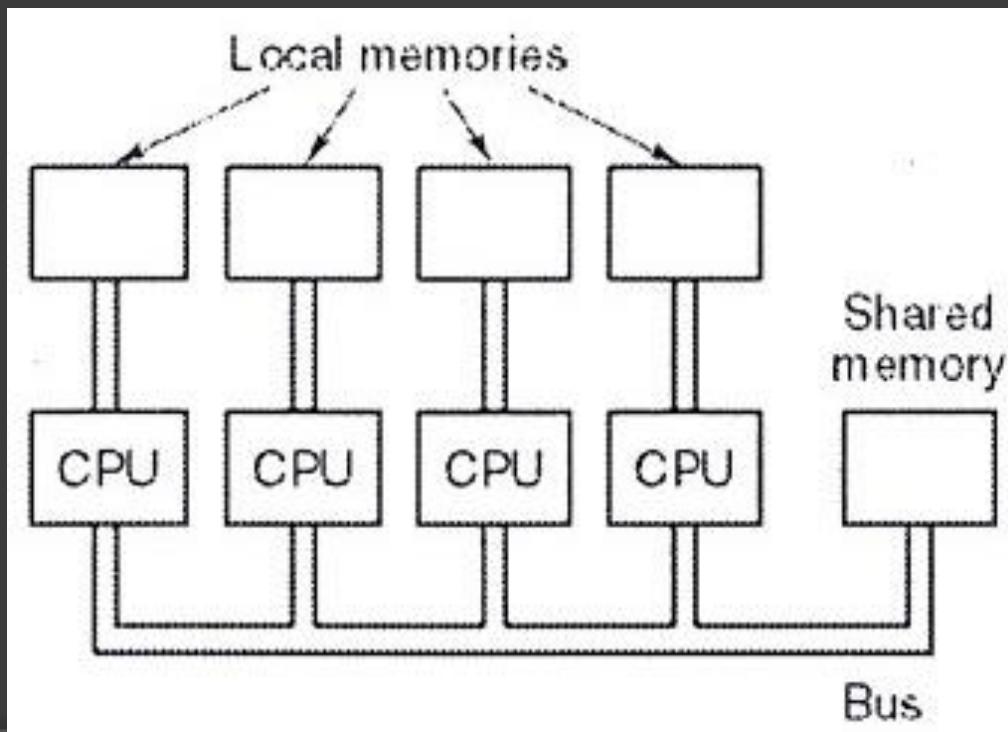
- Single bus y memoria compartida (centralizada)



U5 – Componentes de un computador

○ Procesador

- CPUs con memoria local y memoria compartida



U5 – Componentes de un computador

○ Procesador

● Paralelismo

○ A nivel procesador

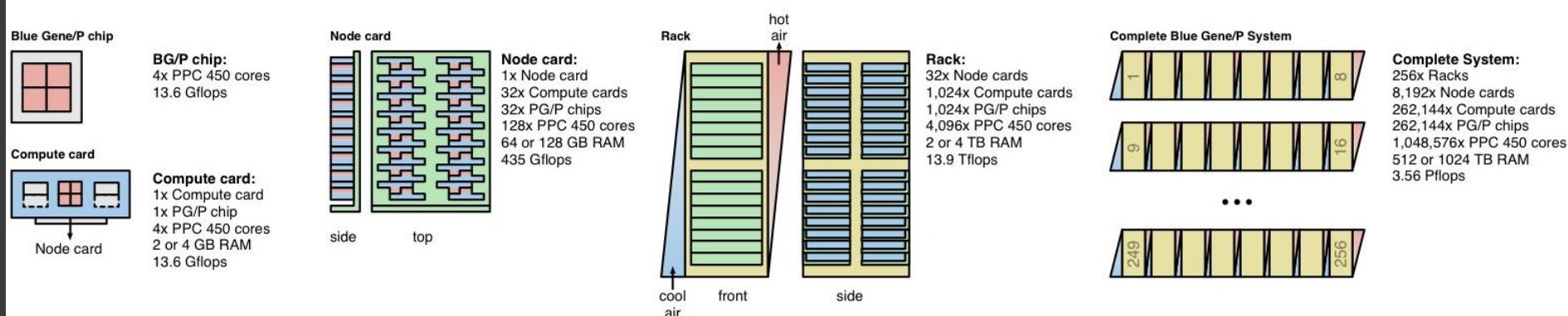
- Multicomputadores
 - Computadores interconectados con memoria local (memoria distribuida)
 - No hay memoria compartida
 - CPUs ligeramente acoplados - Clusters
 - MIMD (Multiple Instruction Multiple data)
 - Intercambio de mensajes
 - Topologías de grillas, árboles o anillos
 - Ej. IBM Blue Gene/P

U5 - Componentes de un computador

○ Procesador

- IBM Blue Gene/P

Blue Gene/P, tiered architecture



U5 – Componentes de un computador

○ Referencias

- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin
(<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)
- “Computer Organization and Architecture – Designing for Performance” 10ma edición. William Stallings
(<http://williamstallings.com/ComputerOrganization/>)
- “Server Architectures - Multiprocessors, Clusters, Parallel Systems, Web Servers, and Storage Solutions” 1ra edición. René Chevance
- “Computer Architecture a quantitative approach” 5ta edición. John Hennessy / David Patterson

75.03 / 95.57 Organización del Computador

U6 - ALMACENAMIENTO SECUNDARIO CINTAS MAGNÉTICAS

U6 – Almacenamiento secundario

○ Cintas magnéticas

- Medio
 - Poliéster flexible cubierto de material magnetizable
 - Carretes abiertos
 - Paquetes cerrados (cartuchos)
 - Ancho de cinta entre 0.38 cm (0.25 pulgadas) y 1.27 cm (0.5 pulgadas)
 - Acceso secuencial a la información: si estoy en el registro 1 y quiero llegar al N tengo que “leer” los N-1 del medio
 - Si quiero leer un registro anterior tengo que rebobinar y volver a buscar el registro

U6 – Almacenamiento secundario

○ Cintas magnéticas

- Técnicas de grabación
 - Grabación en paralelo
 - Técnica usada originalmente
 - Cabeza de grabación estacionaria
 - Se graban pistas en paralelo a lo largo de la cinta
 - Al principio eran de 9 pistas (8 bits de datos y 1 bit de paridad para detectar errores)
 - Luego fueron 18 (palabra) o 36 (doble palabra) pistas

U6 – Almacenamiento secundario

○ Cintas magnéticas

- Grabación en paralelo

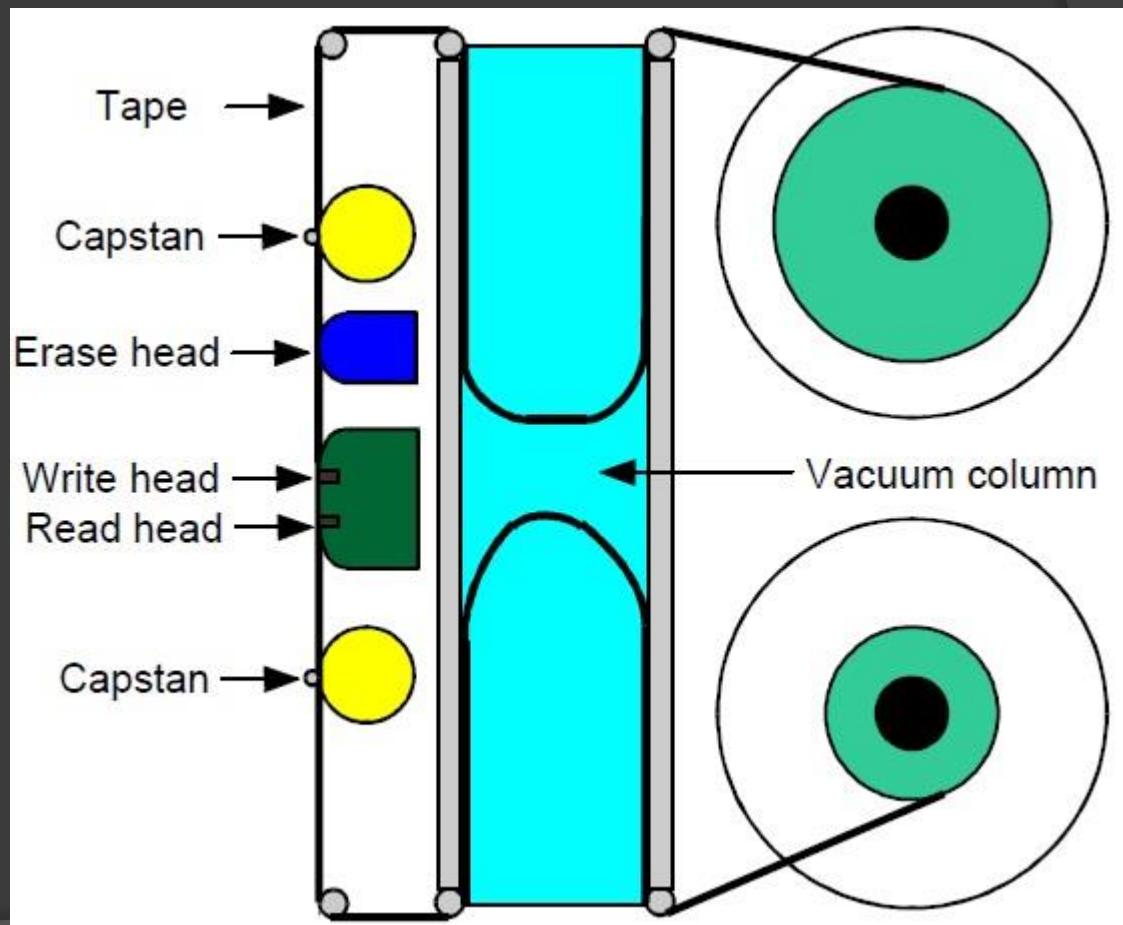
Table 1. Typical specifications of IBM reel-to-reel tape drives.

IBM Product No.	726	3420	3480
FCS (First customer shipment)	1953	1973	1985
Linear Density (BPI)	100	6250	38,000
Number of Tracks	7	9	18
Reel Capacity (MB)	2.2	156	200
Data Rate (KBytes/sec)	75	1250	3000
Recording Code	NRZI	GCR(0,2)	GCR(0,3)
Tape Transport	Vacuum	Vacuum	Cartridge

U6 – Almacenamiento secundario

○ Cintas magnéticas

- Grabación en paralelo



U6 - Almacenamiento secundario

○ Cintas magnéticas

- Reel-to-reel



U6 – Almacenamiento secundario

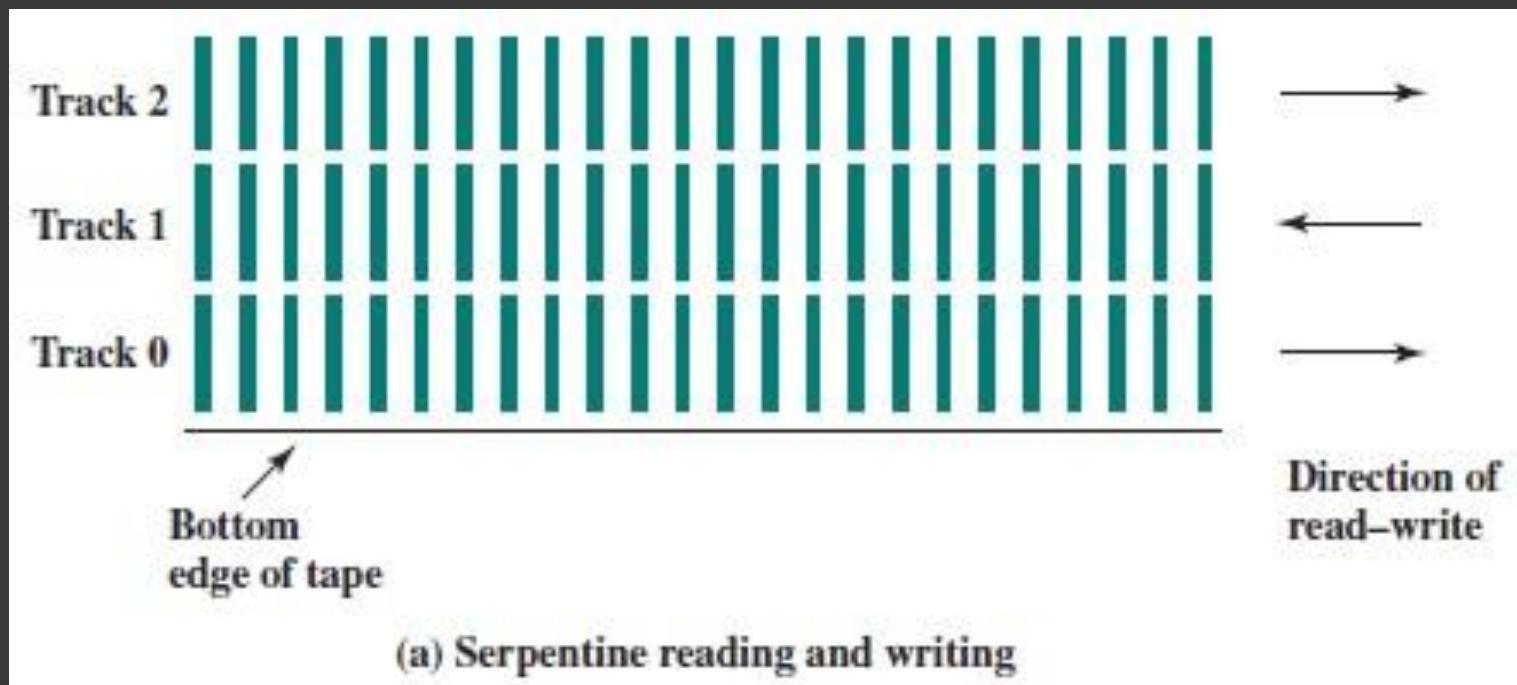
○ Cintas magnéticas

- Técnicas de grabación (cont.)
 - Grabación en serie
 - Sistema moderno de grabación
 - Cabeza de grabación estacionaria
 - Se escriben los datos a lo largo de una pista primero hasta llegar al final de la cinta y luego se pasa a otra
 - Grabación en “serpentina”
 - Pueden grabarse n pistas adyacentes en simultáneo (n entre 2 y 8)

U6 – Almacenamiento secundario

○ Cintas magnéticas

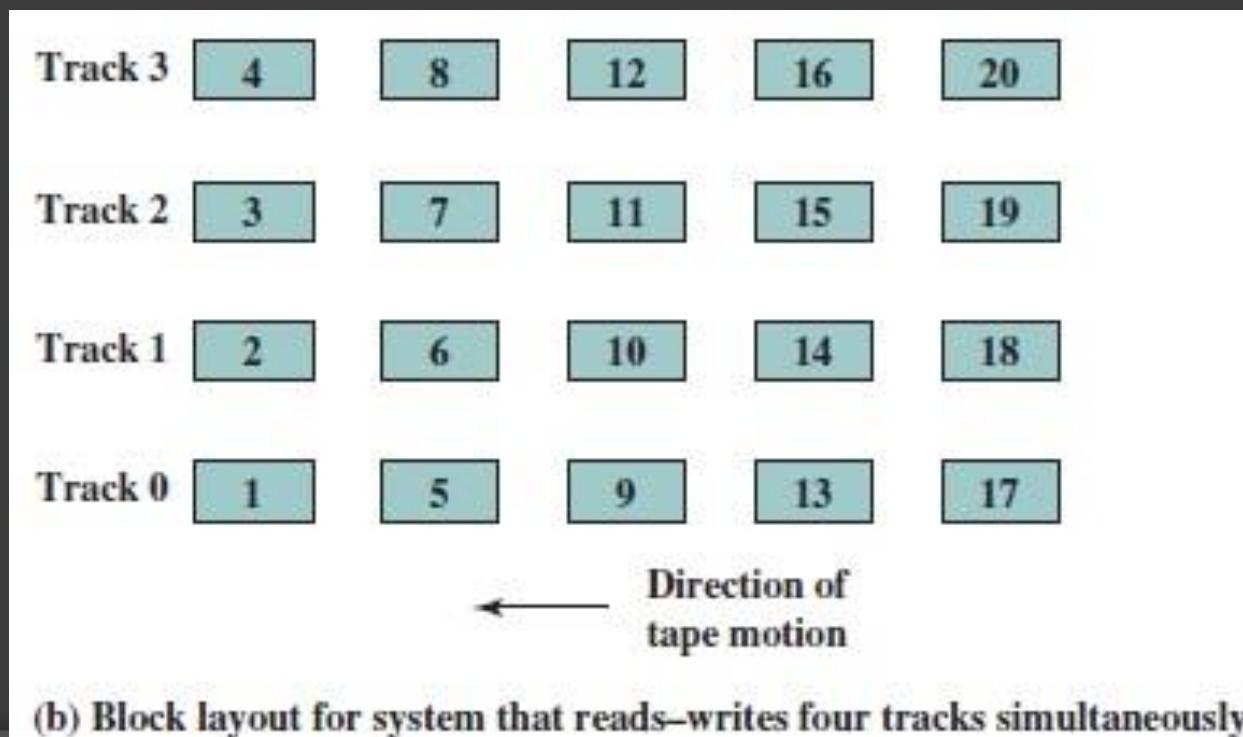
- Técnicas de grabación (cont.)
 - Grabación en serie (serpentina)



U6 – Almacenamiento secundario

○ Cintas magnéticas

- Técnicas de grabación (cont.)
 - Grabación en serie (serpentina multitrack)



U6 – Almacenamiento secundario

○ Cintas magnéticas

- Técnicas de grabación (cont.)
 - Grabación en serie
 - Estándar LTO (Linear Tape Open)
 - Creado en 1997 por HP, IBM y Seagate

	LTO-1	LTO-2	LTO-3	LTO-4	LTO-5	LTO-6	LTO-7	LTO-8	Type Mag. ^[Note 1]	LTO-9	LTO-10	LTO-11	LTO-12
Release date	2000 ^[5]	2003	2005	2007	2010 ^[6]	Dec. 2012 ^[7]	Dec. 2015 ^{[8][9][10]}	Dec. 2017		TBA	TBA	TBA	TBA
Native/raw data capacity	100 GB	200 GB	400 GB	800 GB	1.5 TB ^[11]	2.5 TB ^[12]	6.0 TB ^{[10][13]}	12 TB ^[14]	9 TB	24 TB ^{[11][15]}	48 TB ^[11]	96 TB ^[11]	192 TB ^[11]
compressed capacity	200 GB	400 GB	800 GB	1.6 TB	3.0 TB	6.25 TB	15 TB	30 TB	22.5 TB	60 TB	112.5 TB	240 TB	480 TB
Max uncompressed speed (MB/s)^{[13][Note 2]}	20	40	80	120	140	160	300 ^[16]	360	300	708	1,100	TBA	TBA
Max compressed speed (MB/s)	40	80	160	240	280	400	750	900	750	1,770	2,750	TBA	TBA
Time to write a full tape at max uncompressed speed(hh:mm)	1:25	1:25	1:25	1:50	3:10	4:35	5:55	9:15	8:20	TBA	TBA	TBA	TBA
Compression capable?	Yes, "2:1"				Yes, "2.5:1"				Planned, "2.5:1" ^{[15][17]}				
WORM capable?	No		Yes				No		Planned				
Encryption capable?	No		Yes						Planned				
Max. number of partitions	1 (no partitioning)			2	4				Planned				

1. ^ Previously unused LTO-7 tape, not an independent generation, part of LTO-8 generation. See: [Compatibility](#)

2. ^ Maximum uncompressed speeds valid for full height drives. Half height drives may not attain the same speed. Check manufacturer's specifications.

U6 – Almacenamiento secundario

○ Cintas magnéticas

- Técnicas de grabación (cont.)

- Grabación en serie

- Estándar LTO (Linear Tape Open)

Generations	LTO-1	LTO-2	LTO-3	LTO-4	LTO-5 ^[27]	LTO-6 ^[28]	LTO-7	LTO-7 Type M (M8) ^[29]	LTO-8 ^[30]	LTO-9	LTO-10	LTO-11	LTO-12
Native data capacity	100 GB	200 GB	400 GB	800 GB	1.5 TB ^[11]	2.5 TB ^{[12][31]}	6.0 TB ^{[10][13][31]}	9.0 TB	12 TB ^{[13][31]}	24 TB ^{[15][17][31]}	48 TB ^{[15][31]}	96 TB	192 TB
Tape length	609 m	680 m	820 m	846 m ^[32]			960 m						
Tape width	12.650 mm ± 0.006 mm												
Tape thickness	8.9 µm	8 µm	6.6 µm	6.4 µm	6.1 µm ^[33]	5.6 µm							
Magnetic pigment material	Metal Particulate (MP)				MP or BaFe ^[34]	BaFe ^[35]							
Base material	Polyethylene naphthalate (PEN)												
Data bands per tape	4												
Wraps per band	12	16	11	14	20 ^[11]	34	28	42	52				
Tracks per wrap (read/write elements)	8		16 ^{[11][36]}				32 ^[10]	32	32 (TMR)				
Total tracks	384	512	704	896	1,280	2,176 ^[36]	3,584	5,376	6,656				
Linear density (bits/mm)	4,880	7,398	9,638	13,250	15,142 ^[37]	15,143 ^[38]	19,094 ^[39]	19,104	20,668				
Encoding	RLL 1,7	RLL 0,13/11; PRML		RLL 32/33; PRML		32/33 RLL NPML ^[38]							
End-to-end passes required to fill tape	48	64	44	56	80	136	112	168	208				
Expected tape durability, end-to-end passes	9,600 ^[40]	16,000 ^[40]	16,000 ^[40]	11,200 ^[40]	16,000 ^[40]	20,000			20,000				

U6 - Almacenamiento secundario

- Cintas magnéticas
 - Estándar LTO (Linear Tape Open)
 - LTO-8 Data Cartridge



U6 – Almacenamiento secundario

- Cintas magnéticas
 - Estándar LTO (Linear Tape Open)
 - LTO-8 External Tape Drive



U6 - Almacenamiento secundario

- Cintas magnéticas
 - Estándar LTO (Linear Tape Open)
 - LTO-8 Tape Libraries



The image shows the Spectra Tape Libraries product line. It features seven different tape library models arranged in a row: T50e, T120, T200, T380, T680, T950, and TFinity. The TFinity model is shown with its doors open, revealing internal storage slots. Above the products, the Spectra logo and the text "Tape Libraries" are displayed.

Uncompressed Capacity LTO-8				8-frame		44-frame
600 TB	1.4 PB	2.4 PB	4.5 PB	8 PB	120 PB	641 PB
Compressed Capacity LTO-8						
1.5 PB	3.6 PB	6 PB	11.4 PB	20.18 PB	300.6 PB	1.6 EB

U6 – Almacenamiento secundario

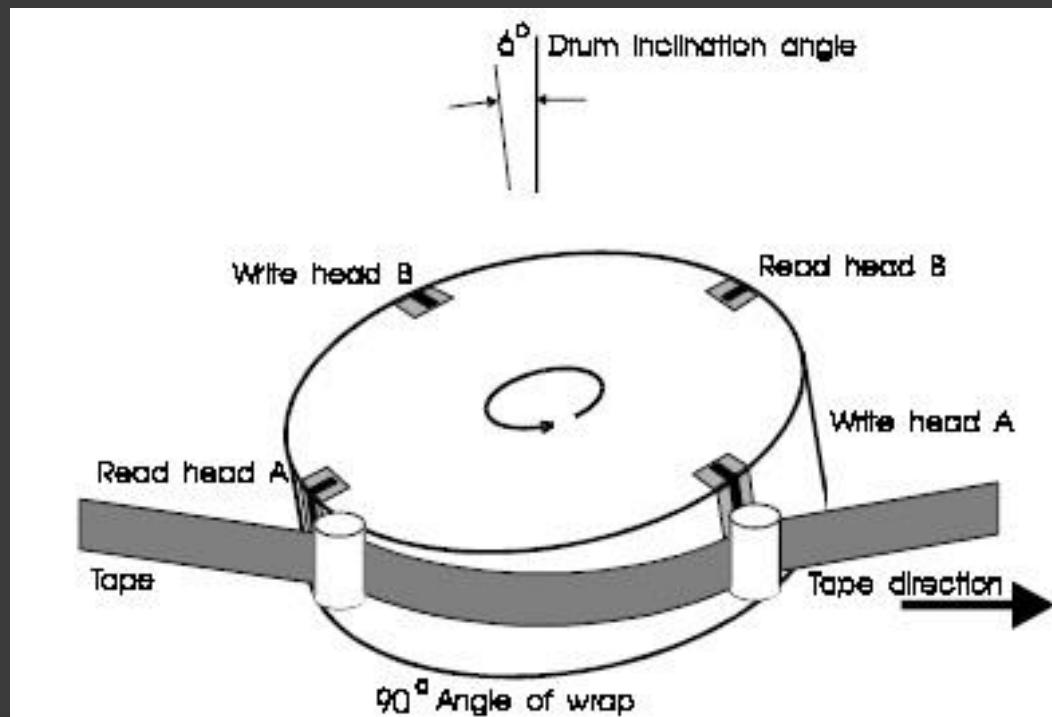
◎ Cintas magnéticas

- Técnicas de grabación (cont.)
 - Grabación helicoidal
 - Cabeza de grabación rotatoria
 - Símil video casseteras
 - Evita problema de movimiento veloz de la cinta de las otras técnicas
 - La cinta se mueve en forma lenta mientras que la cabeza rota en forma rápida
 - Las pistas pueden estar más cercanas unas a otras
 - Formatos:
 - DAT/DDS (4mm), AIT (8mm), Exabyte Mammoth (8mm), SAIT (1/2 "), etc.

U6 – Almacenamiento secundario

○ Cintas magnéticas

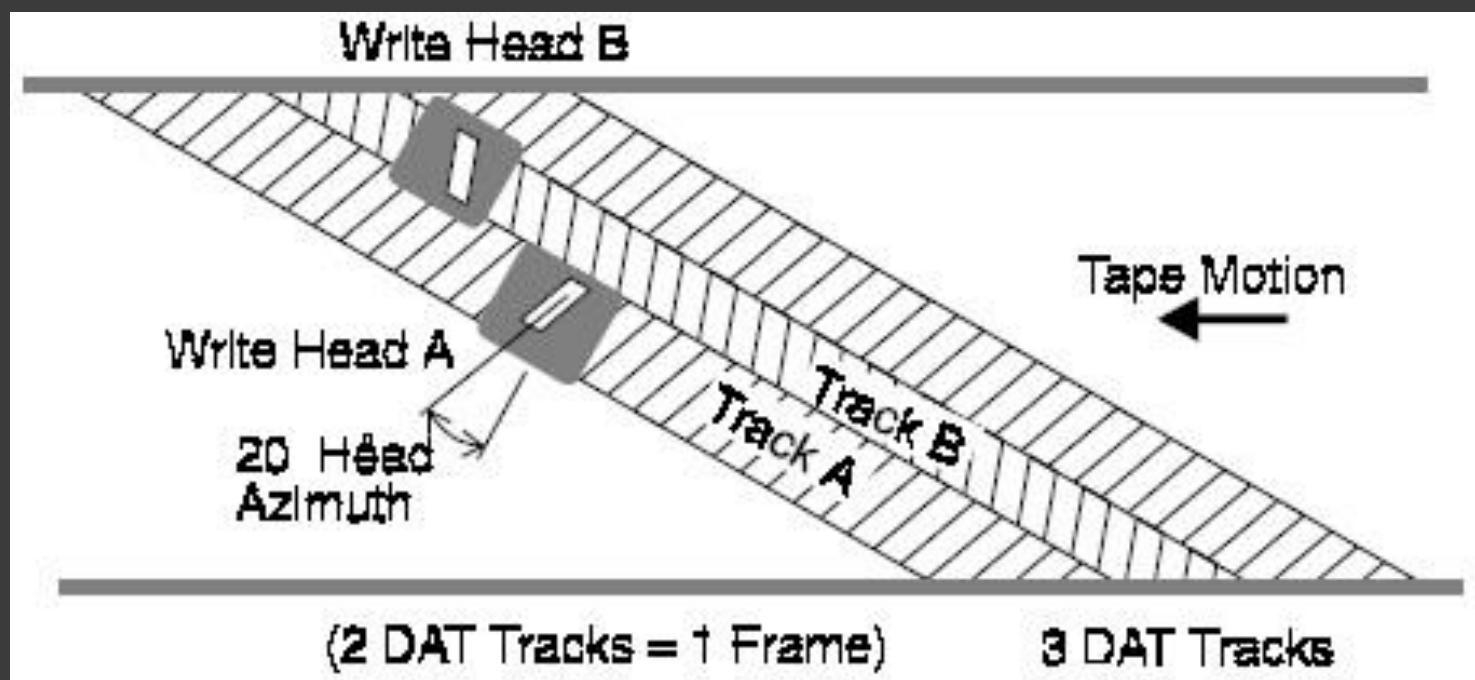
- Técnicas de grabación (cont.)
 - Grabación helicoidal



U6 - Almacenamiento secundario

○ Cintas magnéticas

- Técnicas de grabación (cont.)
 - Grabación helicoidal



U6 – Almacenamiento secundario

- Cintas magnéticas
 - Técnicas de grabación (cont.)
 - Grabación helicoidal
 - Read-Write Head (VCR)



U6 - Almacenamiento secundario

○ Cintas magnéticas

- Técnicas de grabación (cont.)
 - Grabación helicoidal



U6 – Almacenamiento secundario

○ Cintas magnéticas

- Modos de operación
 - Modo start-stop por bloque
 - Viejo uso de grabación por registro/bloque
 - La cinta se usaba para guardar archivos para procesamiento posterior
 - Se podía actualizar un registro/bloque particular siempre y cuando no cambiara su tamaño
 - Los datos se grababan en bloques físicos
 - Entre los bloques había espacios (IRG – Inter Record Gap) para sincronización de la unidad

U6 – Almacenamiento secundario

○ Cintas magnéticas

- Modos de operación
 - Modo streaming
 - Uso para backup o archivo de información
 - No se requiere operación de start-stop por bloque
 - No se requiere actualización de bloques particulares dentro de un archivo
 - Se escriben archivos completos como un “stream” de datos contiguo
 - La información se graba físicamente en bloques pero no se pueden localizar o modificar bloques particulares

U6 – Almacenamiento secundario

● Cintas magnéticas

- Usos y características
 - Fue el primer medio de almacenamiento secundario
 - Aun es usado para backup y archivo de información (30 años o más de duración) dado su bajo costo por byte y su capacidad de almacenamiento
 - Es el medio más lento de la pirámide de jerarquía de memoria
 - Marcas físicas en las cintas
 - BOT (Beginning of tape)
 - EOT (End of tape)

U6 – Almacenamiento secundario

○ Referencias

- “Computer Organization and Architecture – Designing for Performance”
9na edición. William Stallings
(<http://williamstallings.com/ComputerOrganization/>)
- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin
(<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)

75.03 & 95.57 Organización del Computador

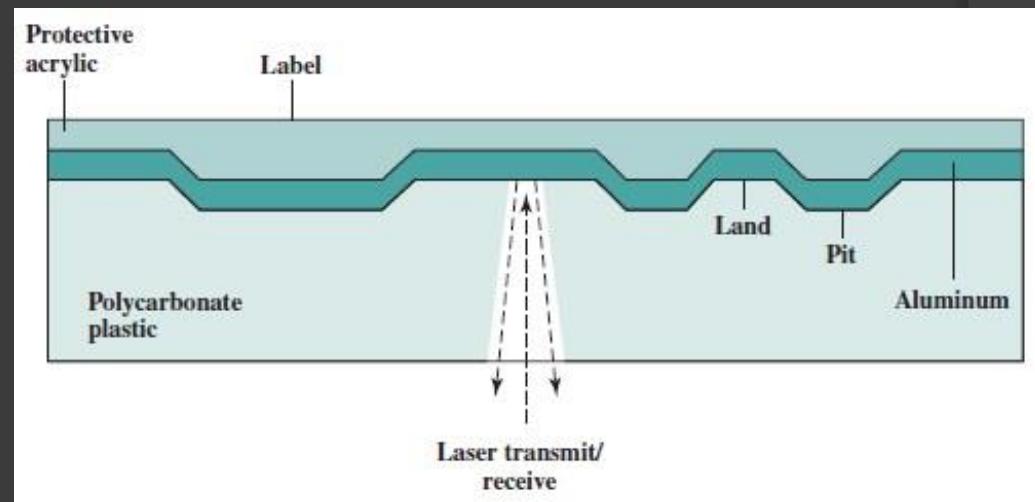
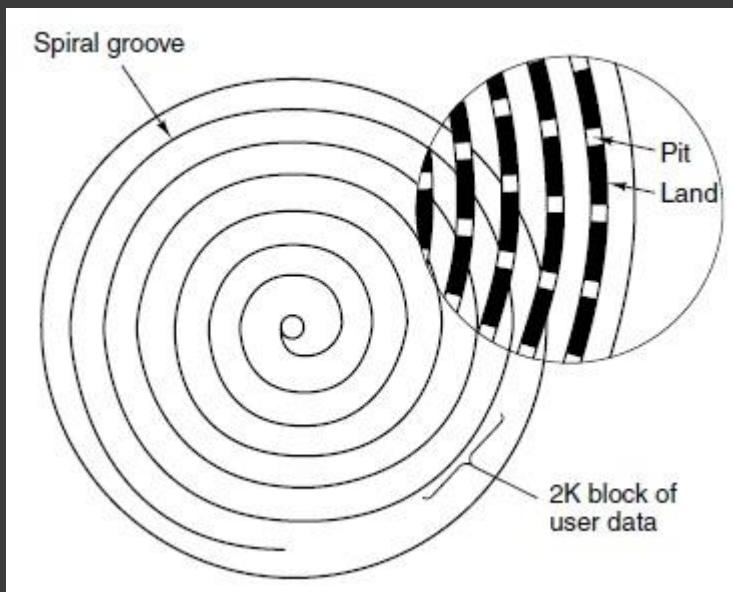
U6 - ALMACENAMIENTO SECUNDARIO MEDIOS ÓPTICOS

U6 – Almacenamiento secundario

- ◎ Medios ópticos
 - CD (Compact Disc)
 - Estándar internacional (1980) entre Philips y Sony (Red book)
 - 120 mm de ancho x 1,2 mm de espesor
 - CLV (Constant Linear Velocity)
 - Pista única en forma de espiral
 - Sectores (2352 bytes)
 - Master:
 - Se graba con un laser infrarrojo de alto poder
 - Se marcan huecos sobre un disco de vidrio
 - Se hace un molde y se le inyecta policarbonato que sigue los patrones de los huecos
 - Se pone luego una capa reflectiva de aluminio sobre el sustrato
 - Encima se coloca una capa protectora de laca y una etiqueta
 - Pit: marca física (hueco)
 - Land: espacio de la superficie sin marcas
 - Bit en 1: transición entre pit-land o land-pit
 - Bit en 0: ausencia de transición en un tiempo determinado => CLV

U6 – Almacenamiento secundario

- Medios ópticos
 - CD (Compact Disc)



U6 – Almacenamiento secundario

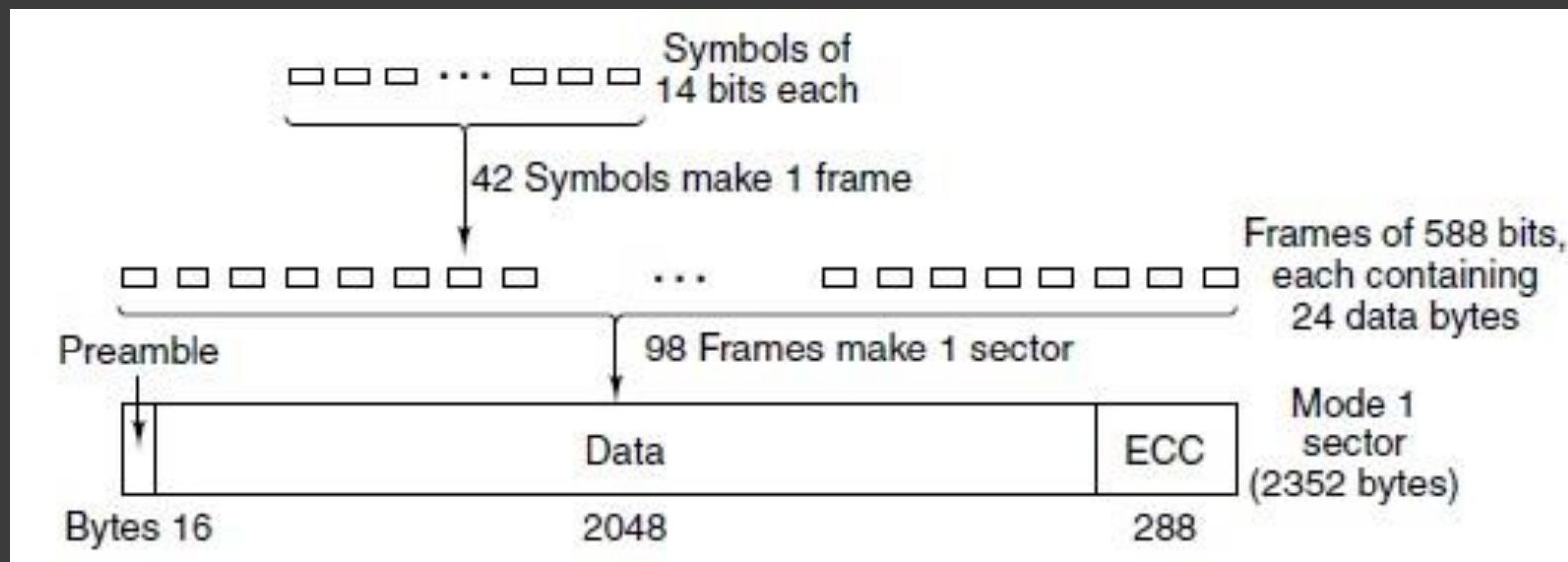
○ Medios ópticos

- CD-ROM (Compact Disc-Read Only Memory)
 - Estándar internacional (1984) (Yellow book)
 - Encoding EFM (Eight to Fourteen Modulation)
 - Modos
 - Modo 1 (Datos): 2048 bytes de datos
 - Modo 2 (Audio): 2336 bytes => 74 minutos
 - Filesystem (Modo 1): ISO 9660

U6 - Almacenamiento secundario

○ Medios ópticos

- CD-ROM (Compact Disc-Read Only Memory)



U6 – Almacenamiento secundario

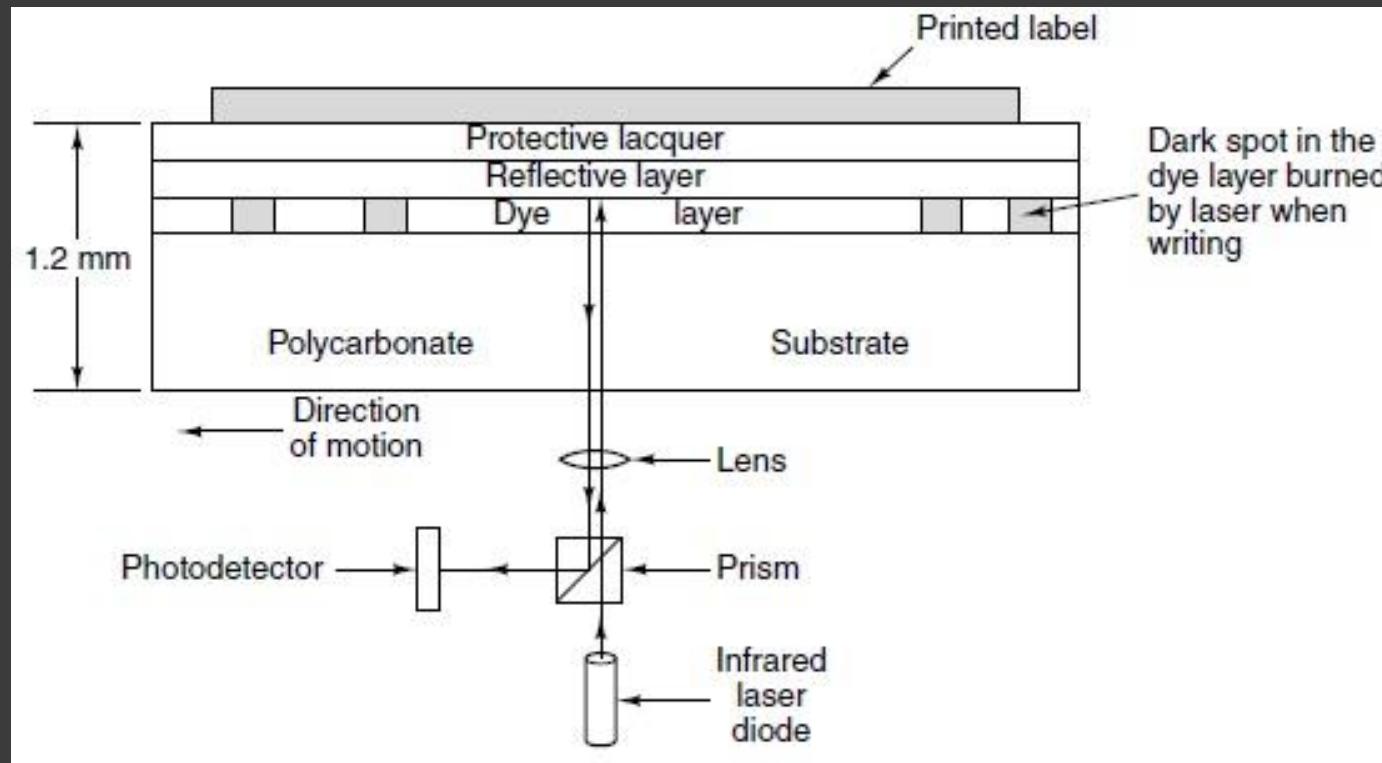
○ Medios ópticos

- CD-R (Compact Disc-Recordable)
 - Estándar internacional (1989) (Orange book)
 - Se podía escribir solo una vez
 - En vez de hacer huecos físicos se usa una capa de tinta que inicialmente es transparente y deja pasar la luz laser
 - El laser quema la capa de tinta y crea un punto negro que ya no refleja la luz simulando un pit
 - CD-XA

U6 - Almacenamiento secundario

○ Medios ópticos

- CD-R (Compact Disc-Recordable)



U6 – Almacenamiento secundario

○ Medios ópticos

- CD-RW (Compact Disc-Rewriteable)
 - Se usa un material que puede tener dos estados con reflectividades diferentes
 - Cristalino: refleja la luz
 - Amorfo: no refleja la luz
 - Para escribir se usa alto poder en el laser para pasar el material de estado cristalino a amorfo (representa un pit)
 - Para borrar se usa medio poder en el laser y se pasa el material de estado amorfo a cristalino nuevamente
 - Para leer se usa bajo poder en el laser

U6 – Almacenamiento secundario

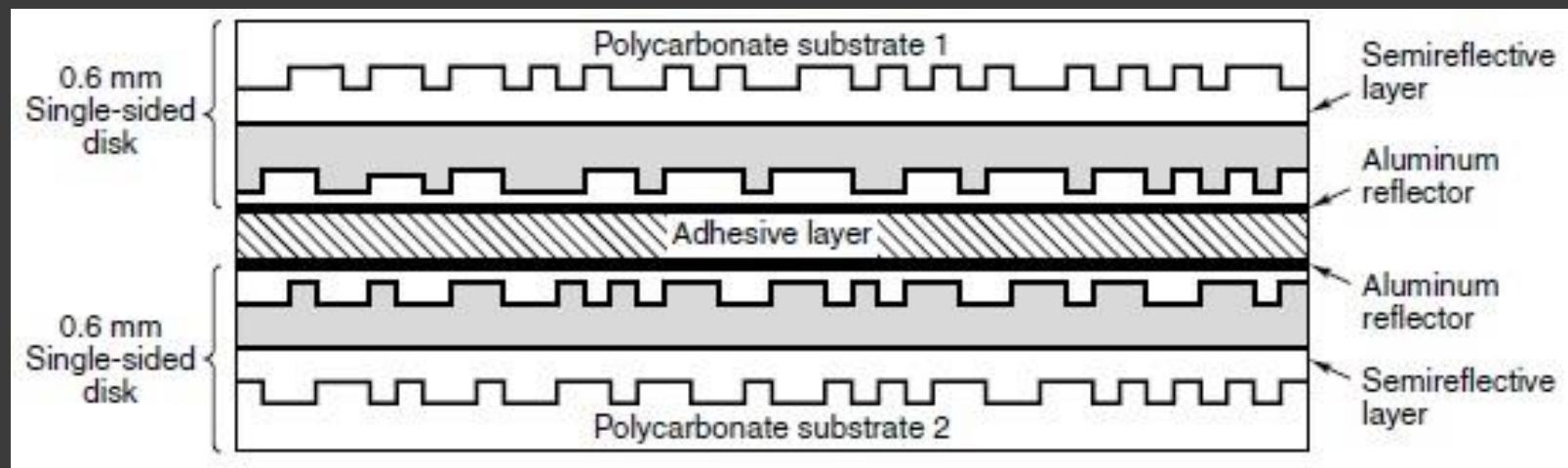
● Medios ópticos

- DVD (Digital Versatile Disk)
 - Diseño general igual al de los CDs
 - Pensados para la industria de distribución de video
 - Mejoras con respecto al CD:
 - Pits más pequeños (0,4 µm versus 0,8 µm del CD)
 - Espiral más encimada (0,74 µm entre pistas versus 1,6 µm del CD)
 - Laser color rojo (a 0,65 µm versus 0,78 µm del CD)
 - Capacidad mejorada a 4,7 GB
 - Formatos definidos:
 - Un solo lado, una sola capa (4,7 GB)
 - Un solo lado, dos capas (8,5 GB)
 - Dos lados, una sola capa (9,4 GB)
 - Doble lado, dos capas (17 GB)

U6 – Almacenamiento secundario

○ Medios ópticos

- DVD (Digital Versatile Disk)
 - Formato doble lado, dos capas



U6 – Almacenamiento secundario

- Medios ópticos
 - DVD (Digital Versatile Disk)
 - DVD-R: usa tecnología similar a CD-R para permitir hacer una grabación única
 - DVD-RW: usa tecnología similar a CD-RW para poder grabar/borrar varias veces

U6 – Almacenamiento secundario

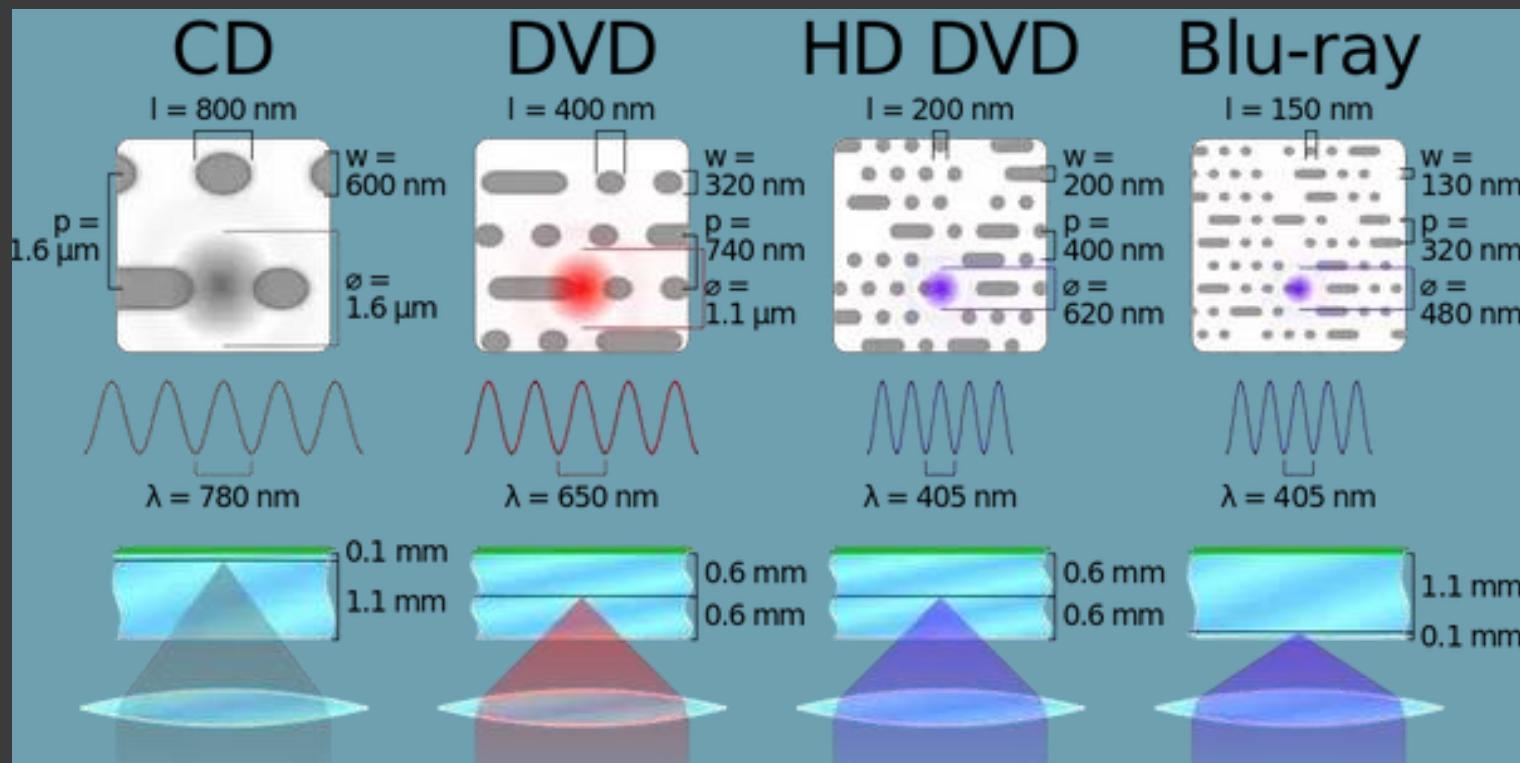
○ Medios ópticos

- HD (High Definition)
 - Se crearon dos estándares para poder distribuir video de alta definición
 - HD-DVD (High Definition – Digital Versatile Disk)
 - Blue-ray Disc (BD)
 - Blue-ray fue la tecnología que predominó
 - Se basa en un laser azul de 0,405 μm
 - La capacidad es de 25 GB para la versión de una sola capa y 50 GB para la de doble capa
 - BD-R: usa tecnología similar a CD-R para permitir hacer una grabación única
 - BD-RE: usa tecnología similar a CD-RW para poder grabar/borrar varias veces

U6 - Almacenamiento secundario

○ Medios ópticos

- Comparación CD-DVD-HD DVD-BlueRay



U6 – Almacenamiento secundario

○ Medios ópticos

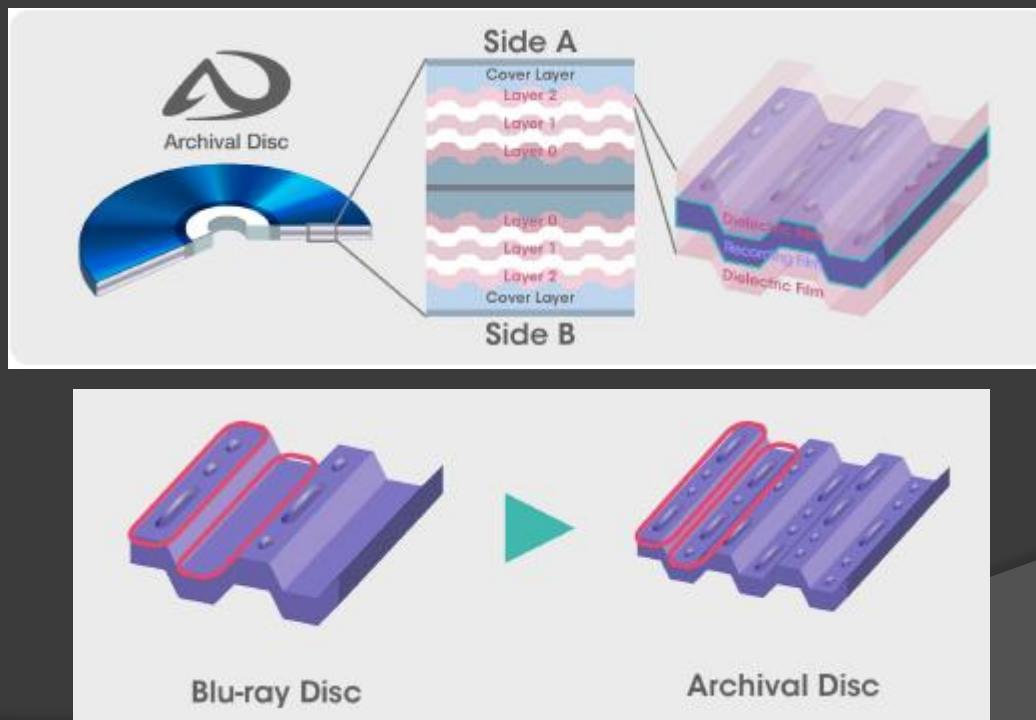
- Sony / Panasonic: Tecnología “Archival Disc”
 - Primera Generación (2015)
 - 300 GB de capacidad
 - Double-Sided
 - 3 layers por lado
 - Grabación en land y groove
 - Long. de onda 405 nm
 - Distancia entre pistas 0.225 μm



U6 – Almacenamiento secundario

○ Medios ópticos

- Sony / Panasonic: Tecnología “Archival Disc”
 - Primera Generación (2015)



U6 – Almacenamiento secundario

○ Medios ópticos

- Sony / Panasonic: Tecnología “Archival Disc”
 - Segunda Generación
 - 500 GB de capacidad
 - Mismas características que la primera generación más:
 - Alta densidad lineal
 - Tecnología de cancelación de interferencias entre símbolos
 - Tercera Generación
 - 1 TB de capacidad
 - Mismas características que la segunda generación más:
 - Tecnología de grabación multinivel

U6 – Almacenamiento secundario



Optical Disc Archive

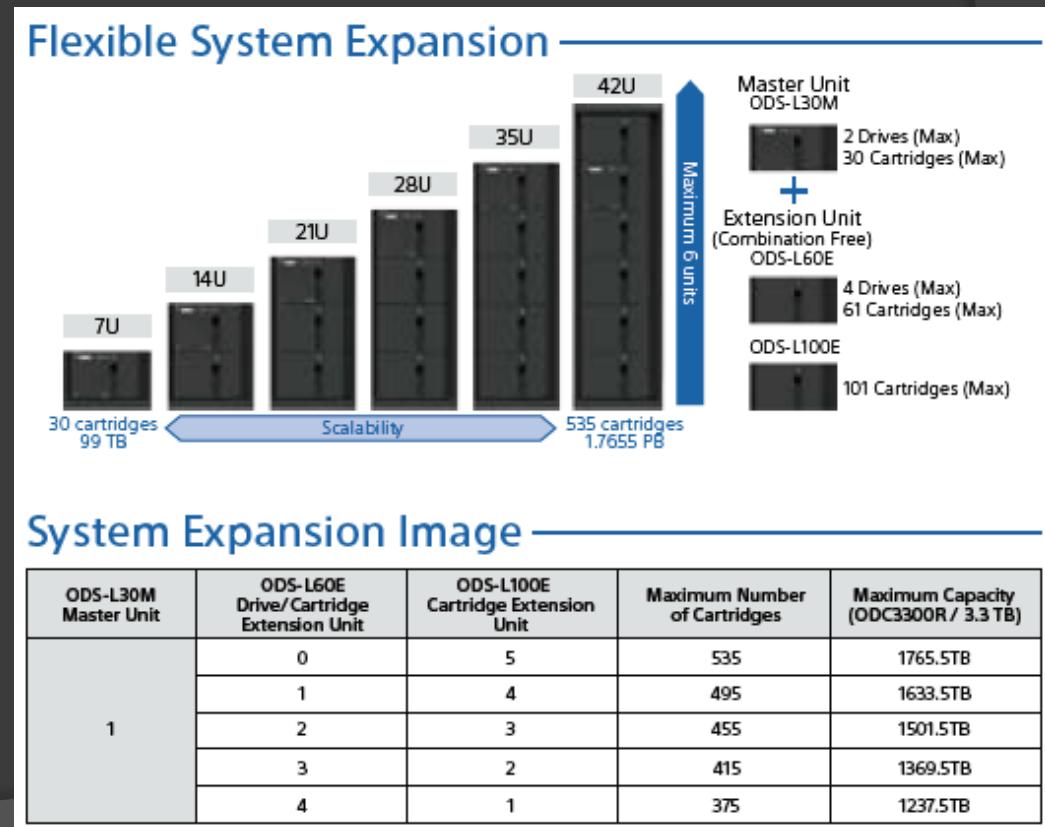
○ Medios ópticos

- Sony “Optical Disc Archive”
 - Nuevo sistema de almacenamiento basado en discos ópticos
 - Cartridges con Archival Discs (3.3 TB)



U6 – Almacenamiento secundario

- Medios ópticos
 - Sony “Optical Disc Archive”



U6 – Almacenamiento secundario

○ Referencias

- “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings (<http://williamstallings.com/ComputerOrganization/>)
- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin (<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)
- Sony “Optical Disc Archiving” https://pro.sony/en_GB/products/optical-disc

75.03 & 95.57 Organización del Computador

U6 - ALMACENAMIENTO SECUNDARIO SSD

U6 – Almacenamiento Secundario

- SSD (Solid State Drive)
 - Definición

“Dispositivo de almacenamiento secundario hecho con componentes electrónicos de estado sólido (semiconductores)”
 - Historia
 - Basados en RAM (volátiles – energía auxiliar)
 - Texas memory: 16KB (1978)
 - Basados en flash (no volátiles)
 - M-Systems (1995)
 - Tecnología actual
 - NAND Flash

U6 - Almacenamiento Secundario

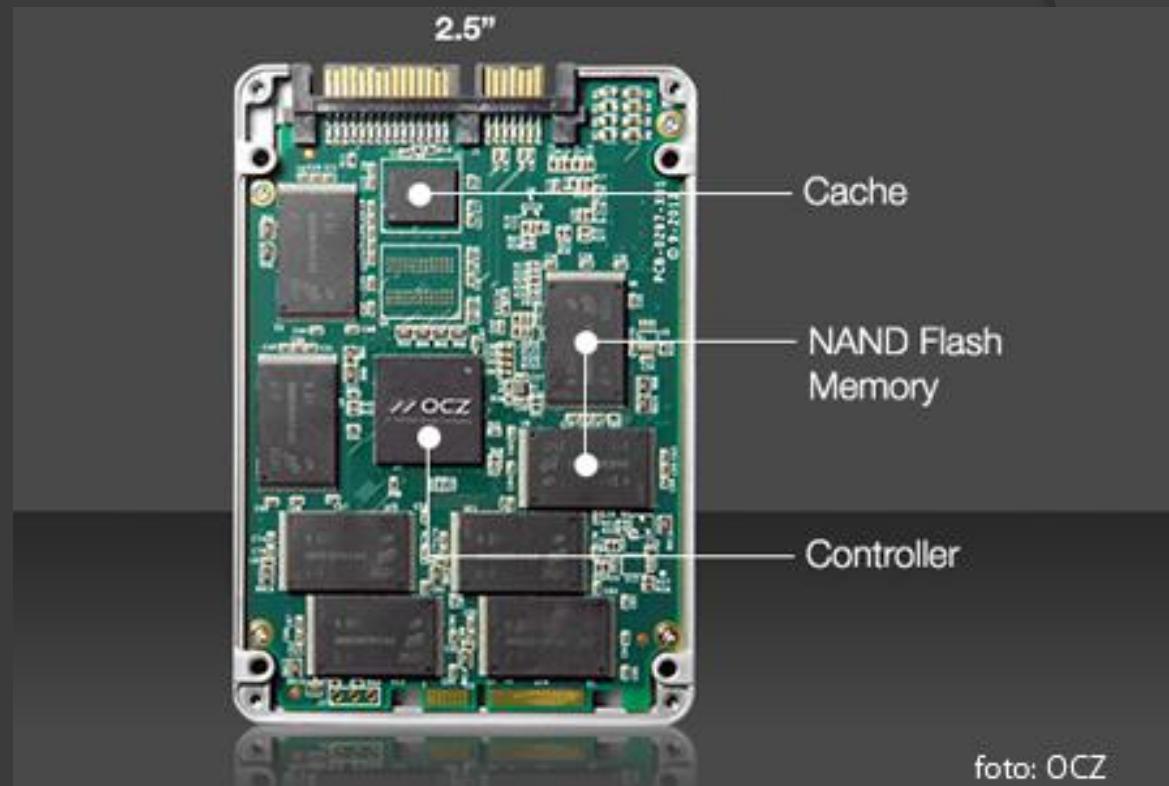
○ SSD



U6 – Almacenamiento Secundario

○ SSD (Solid State Drive)

- Arquitectura
 - Controlador
 - Cache
 - Memorias NAND flash
 - Condensador



U6 – Almacenamiento Secundario

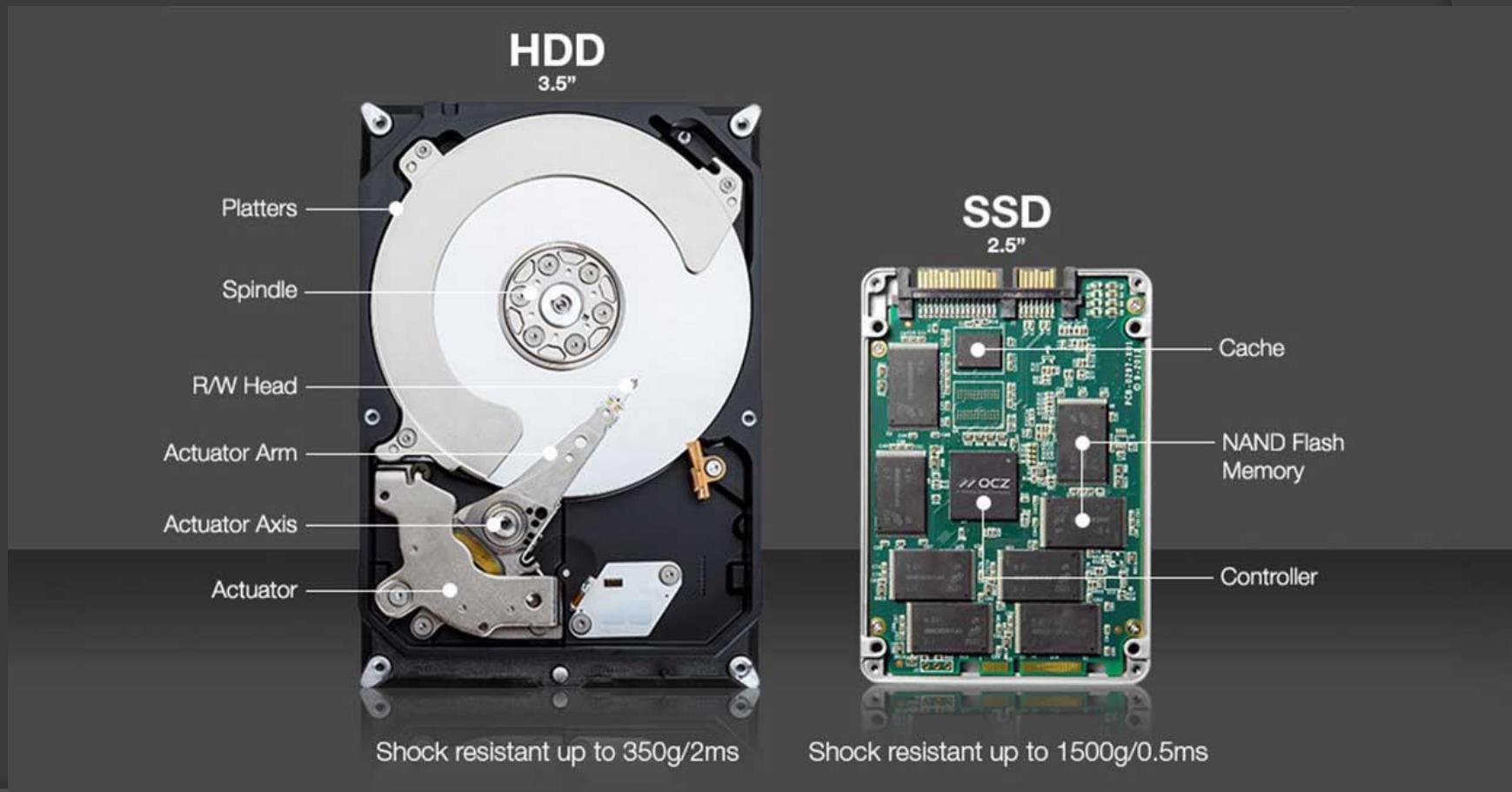
- SSD (Solid State Drive)
 - Comparación con discos magnéticos
 - Ventajas
 - Arranque más rápido
 - Gran velocidad de lectura y escritura
 - Baja latencia de lectura y escritura
 - Menor consumo de energía
 - Menor producción de calor
 - Sin ruido
 - Mejor MTBF (tiempo medio entre fallas)
 - Mayor seguridad de datos
 - Rendimiento determinístico
 - Menor peso y tamaño
 - Mayor resistencia a golpes, caídas y vibraciones

U6 – Almacenamiento Secundario

- SSD (Solid State Drive)
 - Comparación con discos magnéticos
 - Desventajas
 - Precio (\$/GB)
 - Menos recuperación ante fallos
 - Capacidad
 - Vida útil

U6 – Almacenamiento Secundario

○ Disco Magnético vs. SSD



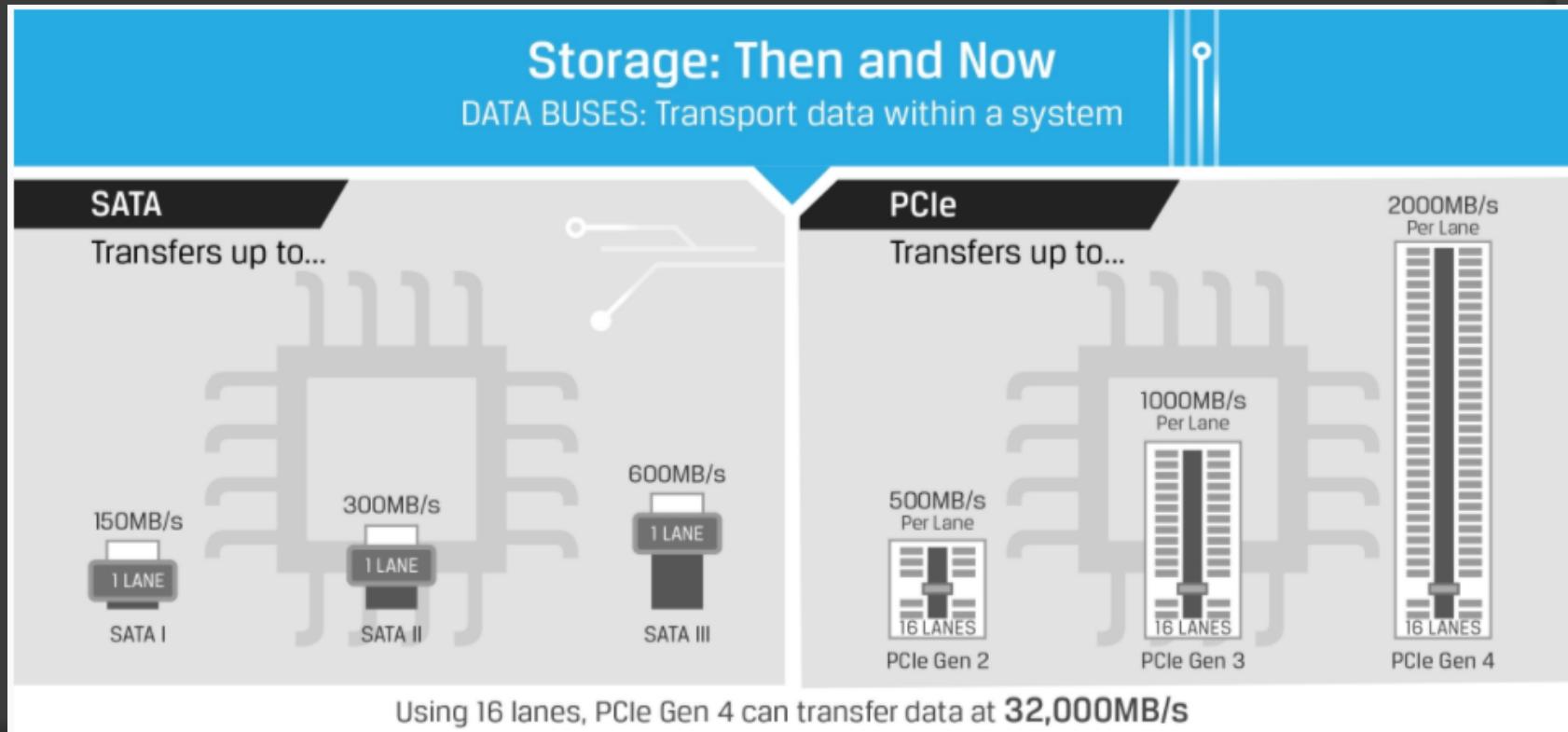
U6 – Almacenamiento Secundario

○ Tecnologías SSD

- PCIe / SATA (interface externa)
- AHCI / NVME (interface de comunicación)
- M.2 / 2.5" SATA / mSATA (form factor)

U6 – Almacenamiento Secundario

- Tecnologías SSD
 - Interface externa



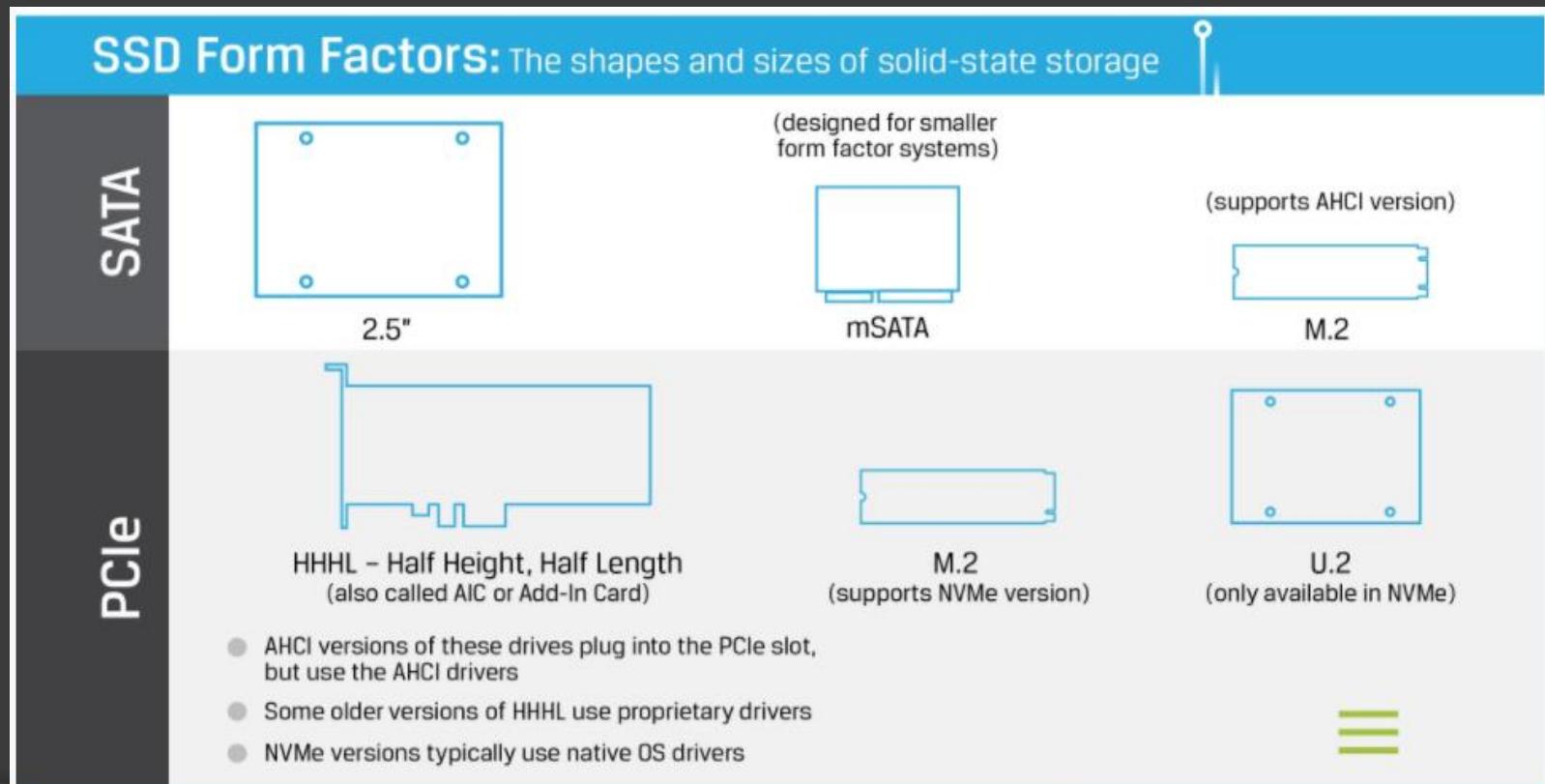
U6 - Almacenamiento Secundario

- Tecnologías SSD
 - Interface de comunicación

Communication Drivers	
Used by Operating Systems to communicate data with storage devices	
AHCI	NVMe
 Designed for Hard Drives with Spinning Disk technology	 Designed for SSDs with Flash technology
1 Has only 1 command queue	64K Has 64K command queues
32 Can only send 32 commands per queue	 Can send 64K commands per queue
 Commands utilize High CPU cycles	 Commands utilize Low CPU cycles
 Has a latency of 6 microseconds	 Has a latency of 2.8 microseconds
 Must communicate with the SATA controller	 Communicates directly with the System CPU
 IOPs up to 100K	 IOPs over 1 million

U6 – Almacenamiento Secundario

- Tecnologías SSD
 - Form factor



U6 – Almacenamiento Secundario

○ Referencias

- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin
(<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)
- “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings
(<http://williamstallings.com/ComputerOrganization/>)

95.57 Organización del Computador

Apunte ARM

Introducción	3
ARM - Arquitectura	4
Set de Registros	4
Registro de Enlace (R14/LR)	4
Contador del Programa (R15/PC)	4
Current Program Status Register	5
Condition flags	5
ARM - Sintaxis	6
Caracteres especiales	6
Directivas	6
ARM - Set de Instrucciones	8
Características principales	8
Formato de las instrucciones	8
Ejecución condicional	8
Códigos de Condición	9
Seteo de Códigos de Condición	10
Ejemplo	10
Load/Store Multiple	11
Instrucciones Aritméticas	11
Operaciones	11
Sintaxis	12
Ejemplos	12
Multiplicación	12
Restricciones	12
Instrucciones Lógicas	12
Operaciones	12
Sintaxis	13
Ejemplos	13
Instrucciones de Bifurcación	13
Bifurcaciones condicionales	14
Pseudo-Instrucciones	15
Subrutinas	15
Ejemplo	15
Programa llamador	15
Subrutina	16
Movimiento de datos	16
Operaciones	16

Sintaxis	16
Ejemplos	16
Barrel Shifter	16
Operaciones soportadas por el Barrel Shifter	16
Shift a Izquierda (LSL)	16
Shift Lógico a Derecha (LSR)	17
Shift Aritmético a Derecha (ASR)	17
Rotate Right (ROR)	17
Rotate Right Extended (RRX)	17
Ejemplos	18
Instrucciones Load/Store	18
Transferencia de datos de un registro	18
Ejecución condicional	18
Sintaxis	18
Registro Base	18
Offset desde el Registro Base	19
Direccionamiento Pre-indexado	19
Direccionamiento Post-indexado	20
Pila (Stack)	21
Pilas y subrutinas	21
Interrupción de Software (SWI)	21
SWIs en ARSim#	21
SWI Codes	21
Modos de direccionamiento	23
Modo pre-indexado	23
Modo pre-indexado con reescritura	23
Modo post-indexado	24
Anexo	25
Sumario de Set de Instrucciones	25
Introducción a la Familia de Arquitecturas ARM	29
Ejemplo en C	29
¿Por qué enseñar lenguaje assembler y arquitectura de computadoras?	29
¿Por qué aprender ensamblador?	30

Introducción

ARM significa Advanced RISC Machine y es el primer procesador de computadora de set de instrucciones reducida (RISC) para uso comercial. Originalmente fue Acorn RISC Machine, dado que fue concebida originalmente por Acorn Computers para su uso en ordenadores personales.

ARM es una arquitectura RISC (Reduced Instruction Set Computer) de 16 ó 32 bits y, a partir de la versión V8-A, también de 64 Bits.

Un enfoque de diseño basado en RISC permite que los procesadores ARM requieran una cantidad menor de transistores que los procesadores x86 CISC, típicos en la mayoría de ordenadores personales. Este enfoque de diseño nos lleva, por tanto, a una reducción en los costes y energía. Estas características son deseables para dispositivos que funcionan con baterías, como los teléfonos móviles, tablets o netbooks.

La relativa simplicidad de los procesadores ARM los hace ideales para aplicaciones de baja potencia. Como resultado, se han convertido en los dominantes dentro del mercado de la electrónica móvil e integrada, encarnados en microprocesadores y microcontroladores pequeños, de bajo consumo y relativamente bajo costo.

La primera ARM se estableció en la Universidad de Cambridge en 1978. Las computadoras del grupo Acorn desarrollaron el primer procesador comercial RISC de ARM en 1985. ARM se fundó y fue muy popular en 1990. En 2005, alrededor del 98% de los más de mil millones de teléfonos móviles vendidos utilizaban al menos un procesador ARM y en 2007, ARM era utilizada en la gran mayoría de los teléfonos móviles. Desde 2009, los procesadores ARM son aproximadamente el 90% de todos los procesadores RISC de 32 bits integrados.

El núcleo del procesador ARM es el motor dentro del sistema que obtiene las instrucciones ARM de la memoria y las ejecuta. Los núcleos ARM son muy pequeños y suelen ocupar unos pocos milímetros cuadrados del área del chip. Debido a su bajo consumo de energía y a que tiene un mejor rendimiento en comparación con otros procesadores, ARM es el procesador utilizado en los productos digitales avanzados, como teléfonos móviles, sistemas de cámaras digitales, redes domésticas, tecnologías inalámbricas y dispositivos portátiles.

ARM - Arquitectura

ARM está basado en una arquitectura load/store, reduciendo así el set de instrucciones; esto significa que el núcleo no puede operar directamente con la memoria. Todas las operaciones de datos deben realizarse mediante registros con la información que se encuentra en la memoria.

Set de Registros

ARM tiene 16 Registros visibles al programador y un Registro de estado del programa actual, CPSR (Current Program Status Register). El detalle de los registros es:

- *R0 a R12* son los registros de *uso general*
- *R13* está reservado para que el programador lo utilice como *puntero a la pila*
- *R14 o LR* es el *registro de enlace* que almacena una dirección de retorno de una subrutina
- *R15 o PC* contiene el contador del programa y es accesible al programador
- *CPSR* es el registro que contiene información sobre el estado actual del programa

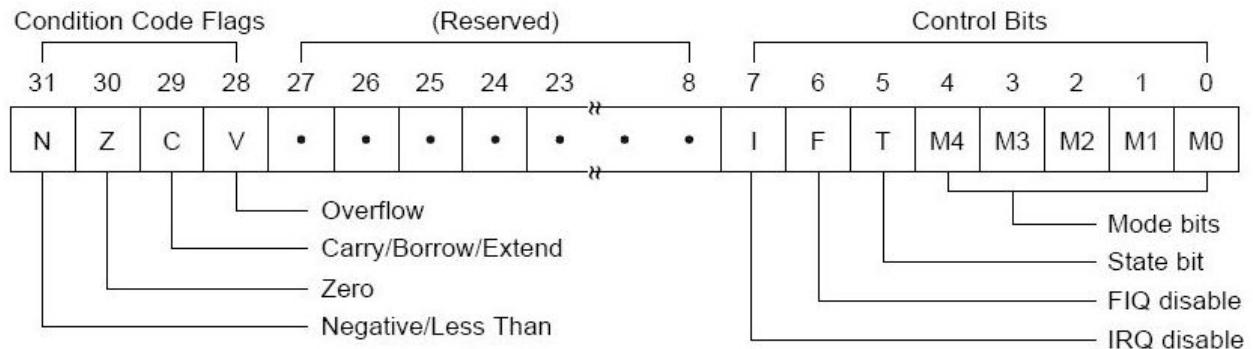
Registro de Enlace (R14/LR)

El *R14* es usado como registro enlace a subrutinas (*LR*) y almacena la dirección de retorno cuando una se realiza una operación *Branch with Link*, la cual se calcula desde el registro *PC*. Siendo así, para volver desde una subrutina puede ejecutarse: `MOV r15, r14` o `MOV pc, lr`.

Contador del Programa (R15/PC)

Como todas las instrucciones tienen longitud de 32 bits y deben estar alineadas a *word* (cada *word* es de 4 bytes, esto significa que cada instrucción comienza en una posición de memoria divisible por 4), el valor almacenado en el registro *Contador del Programa (PC)* es almacenado en los bits [31:2] con los bits [1:0] iguales a 0 (dado que las instrucciones no pueden estar alineadas a *halfword* o *byte*).

Current Program Status Register



Condition flags

Flag	Instrucción Lógica	Instrucción Aritmética
Negative	Sin significado particular	Indica un número negativo en una operación con signo
Zero	Resultado es cero	Resultado es cero
Carry	Luego de una operación Shift: Se cargó '1' en el flag de carry	Resultado es mayor a 32 bits
Overflow	Sin significado particular	Resultado es mayor a 31 bits. Indica una posible corrupción del signo en operaciones con signo

ARM - Sintaxis

Los comentarios en comienzan con # y se ignora todo, desde el signo hasta el final de la línea.

Las etiquetas pueden definirse utilizando una secuencia de caracteres alfanuméricos, barras inferiores (_) y puntos (.) que no comienzan con un número. Los códigos de operación son palabras reservadas que no pueden ser utilizados como identificadores válidos. Las etiquetas se declaran colocándolas al principio de una línea seguida de dos puntos, por ejemplo:

item:

```
.word 1
```

Las cadenas de caracteres (strings) deben ir entre comillas dobles ("") y los caracteres especiales siguen la convención de C:

newline	\n
tab	\t
quote	\"

Caracteres especiales

La presencia del carácter @ en una línea indica el comienzo de un comentario que se extiende hasta el final de la línea.

Si el carácter # aparece como el primer carácter de una línea, toda la línea es tratada como un comentario, pero en este caso la línea puede también ser una directiva de lógica numérica o un comando de control para el preprocesador.

El carácter ; puede ser usado en lugar de una nueva línea para separar sentencias.

Tanto # como \$ pueden ser usados para indicar operandos inmediatos.

Directivas

.equ sym, constant

Da el nombre simbólico *sym* a una constante *constant*.

.data <addr>

Indica que los siguientes ítems son datos y deben almacenarse en el segmento de datos. Si el argumento opcional *addr* está presente, los ítems son almacenados desde la dirección *addr*.

.align n

Alinear el siguiente dato en una posición de memoria divisible por 2^n . Por ejemplo, .align 2 alinea el siguiente valor en una dirección divisible por 4 (alineado a word) límite de palabra.

.align 0 desactiva la alineación automática de las directivas **.half**, **.word**, **.float** y **.double** hasta la siguiente directiva **.data**.

.ascii str

Almacena el string en memoria pero no agrega byte nulo al final.

.asciiz str

Almacena el string en memoria y agrega byte nulo al final.

.byte b1, ..., bn

Almacena los n valores en bytes sucesivos en memoria.

.half h1, ..., hn

Almacena los n valores de 16 bits en halfwords sucesivos en memoria.

.word w1, ..., wn

Almacena los n valores de 32 bits en words sucesivos en memoria.

.float f1, ..., fn

Almacena los n flotantes de precisión simple sucesivos en memoria.

.double d1, ..., dn

Almacena los n flotantes de precisión doble sucesivos en memoria.

.comm sym size

Aloca size bytes en el segmento de datos para el símbolo sym.

.globl sym

Declara que el símbolo sym es global y que puede ser referenciado desde otros archivos.

.label sym

Declara que el símbolo sym es una etiqueta.

.text <addr>

Indica que los siguientes ítems en memoria son instrucciones. Si el argumento opcional *addr* está presente, los ítems son almacenados desde la dirección *addr*.

.end

Marca el fin del archivo del módulo del programa.

ARM - Set de Instrucciones

Características principales

- Todas las instrucciones tienen una longitud de 32 bits
- La mayor parte de las instrucciones se ejecutan en un solo ciclo de reloj
- La mayor parte de las instrucciones pueden ser ejecutadas condicionalmente
- Arquitectura Load/Store
 - Las instrucciones de procesamiento de datos actúan únicamente sobre registros
 - Formato de tres operandos
 - ALU y Shifter combinados para alta velocidad en la manipulación de bits
 - Instrucciones específicas de acceso a memoria con potentes modos de direccionamiento de auto indexación
 - Tipos de datos de 32, 16 y 8 bits
 - Instrucciones Load y Store flexibles

Formato de las instrucciones

Cada instrucción es codificada en una 32-bit word.

El formato de codificación básico para instrucciones de carga, almacenamiento, aritméticas y lógica es:

31	28	27	20	19	16	15	12	11	4	3	0
Condition		OP code	Rn	Rd		Other info			Rm		

Una instrucción especifica un código de ejecución condicional (*Condition*), el código OP (*OP code*), dos o tres registros (*Rn*, *Rd* y *Rm*) y alguna otra información adicional.

Ejecución condicional

Una característica distintiva y algo inusual de los procesadores ARM es que todas las instrucciones se ejecutan condicionalmente dependiendo de una condición especificada en la instrucción.

La instrucción es ejecutada sólo si el estado actual del flag del código de condición del procesador satisface la condición especificada en los bits $b_{31}-b_{28}$ de la instrucción. Por lo tanto, las instrucciones cuya condición no se ve satisfecha en el flag de código de condición del

procesador no se ejecutan. Una de las condiciones se utiliza para indicar que la instrucción siempre se ejecuta.

Esta característica elimina la necesidad de utilizar muchas bifurcaciones. El costo en tiempo de no ejecutar una instrucción condicional es frecuentemente menor que el uso de una bifurcación o llamado a una subrutina que, de otra manera, sería necesaria.

Códigos de Condición

Code [31:28]	Mnemonic	Interpretación	Status flag state required
0000	EQ	Igual / Igual a cero	Z seteado
0001	NE	Distinto	Z vacío
0010	CS/HS	Carry seteado / \geq sin signo	C seteado
0011	CC/LO	Carry vacío / $<$ sin signo	C vacío
0100	MI	Menor / negativo	N seteado
0101	PL	Mayor / positivo o cero	N vacío
0110	VS	Overflow	V seteado
0111	VC	No overflow	V vacío
1000	HI	$>$ sin signo	C seteado y Z vacío
1001	LS	\leq sin signo	C vacío y Z seteado
1010	GE	\geq con signo	N igual a V
1011	LT	$<$ con signo	N distinto de V
1100	GT	$>$ con signo	Z vacío y N igual a V
1101	LE	\leq con signo	Z seteado o N distinto de V
1110	AL	Siempre	Cualquiera
1111	NV	Nunca	Ninguno

Para que una instrucción sea ejecutada condicionalmente se le agrega el sufijo con la condición apropiada. Por ejemplo, una instrucción de suma tiene la siguiente forma:

```
ADD r0, r1, r2
```

y para ejecutarla sólo si el flag cero está seteado:

```
ADDEQ r0, r1, r2
```

Esto mejora la densidad del código y la performance reduciendo el número de instrucciones de bifurcación. Ej.:

```
CMP r3, #0
BEQ skip
ADD r0, r1, r2
.skip: ...
```

```
CMP r3, #0
ADDNE r0, r1, r2
```

Seteo de Códigos de Condición

Algunas instrucciones, como Compare, dadas por `CMP Rn, Rm` que realiza la operación $[R_n] - [R_m]$ tienen como único propósito establecer los flags de código de condición en función del resultado de la resta.

Exceptuando a las instrucciones de comparación, las operaciones de procesamiento de datos no afectan a los *condition flags*. Para que los *condition flags* se vean afectados, el bit S de la instrucción necesita estar seteado. Esto se hace agregando el sufijo S a la instrucción (y a cualquier código de condición). Por ejemplo, para restar uno al R1 y afectar los *condition flags* en un loop:

```
.loop: ...
    subs r1, r1, #1
    bne loop
```

Ejemplo

```
ldr r1, n
ldr r2, puntero
mov r0, #0
.loop:
    ldr r3, [r2], #4
    add r0, r0, r3
    subs r1, r1, #1
    bgt loop
    str r0, suma
```

Suma los n enteros desde la posición apuntada por puntero y almacena el resultado en suma

GT: comparación por mayor con signo

BGT: bifurca si si Z=0 y N=0

Load/Store Multiple

En los procesadores ARM, hay dos instrucciones para cargar y almacenar múltiples operandos y son las llamadas instrucciones de transferencia de bloques. Cualquier subconjunto de los registros de propósito general se puede cargar o almacenar. Solo se permiten operandos *word* y los códigos OP utilizados son *LDM* (Load Multiple) y *STM* (Store Multiple).

Los operandos de memoria deben estar en ubicaciones de *words* sucesivas. Todas las formas de pre- y post-indexación con y sin reescritura están disponibles. Operan en un registro base *Rn* especificado en la instrucción y el desplazamiento es siempre 4:

```
ldmia R10!, {R0,R1,R6,R7}
```

IA: “*Increment After*” corresponde a post-indexación

El Registro Base puede ser actualizado si se le agrega el signo !

Instrucciones Aritméticas

La expresión básica para instrucciones aritméticas es OPcode Rd, Rn, Rm

Ejemplos básicos:

```
add r0, r2, r4 @ realiza la operación r0=[r2]+[r4]
sub r0, r6, r5 @ realiza la operación r0=[r6]-[r5]
add r0, r3, #17 @ realiza la operación r0=[r3]+17
```

El segundo operando puede ser shifteado o rotado antes de ser usado en la operación. Por ejemplo:

add r0, r1, r5, lsl #4 opera del siguiente modo: el segundo operando almacenado en r5 es shifteado a izquierda 4 bits (equivalente a [r5]x16), y se le suma el contenido de r1; la suma es almacenada en r0.

Operaciones

add	operand1 + operand2	@ suma
adc	operand1 + operand2 + carry	@ suma con acarreo
sub	operand1 - operand2	@ resta
sbc	operand1 - operand2 + carry - 1	@ resta con acarreo
rsb	operand2 + operand1	@ resta inversa

```
rsc    operand2 - operand1 + carry - 1 @ resta inversa con acarreo
```

Sintaxis

```
<operation>{<cond>} {S} Rd, Rn, operand2
```

Ejemplos

- add r0, r1, r2
- subgt r3, r3, #1
- rsbles r4, r5, #5

Multiplicación

ARM básico provee dos instrucciones de multiplicación:

```
mul{<cond>} {S} Rd, Rm, Rs          @ Rd = Rm * Rs  
mla{<cond>} {S} Rd, Rm, Rs, Rn      @ Rd = (Rm * Rs) + Rn
```

Restricciones

- Rd y Rm no pueden ser el mismo registro
- No puede usarse el registro PC

Instrucciones Lógicas

Las operaciones lógicas AND, OR, XOR, y Bit-Clear son implementadas con instrucciones con los códigos de operación AND, ORR, EOR, y BIC. Por ejemplo:

and r0, r0, r1 realiza la operación $r0 \leftarrow [r0] + [r1]$

La instrucción Bit-Clear (BIC) está estrechamente relacionada con la instrucción AND: complementa cada bit del operando Rm antes de aplicarle AND con los bits del registro Rn. Por ejemplo:

bic r0, R0, r1, siendo r0=02FA62CA y r1=0000FFFF, el resultado de la instrucción es r0=02FA0000.

La instrucción Move Negative complementa los bits del operando fuente y almacena el resultado en Rd. Por ejemplo:

mvn r0, r3

Operaciones

```
and    operand1 AND operand2
```

```
eor    operand1 EOR operand2
orr    operand1 OR  operand2
orn    operand1 NOR operand2
bic    operand1 AND NOT operand2
```

Sintaxis

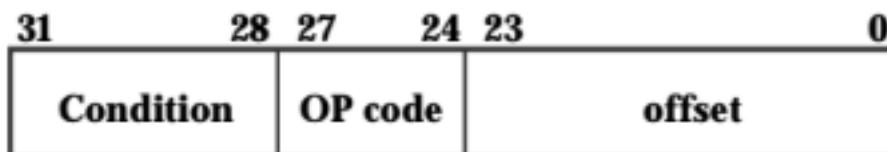
```
<operation>{<cond>} {S} Rd, Rn, operand2
```

Ejemplos

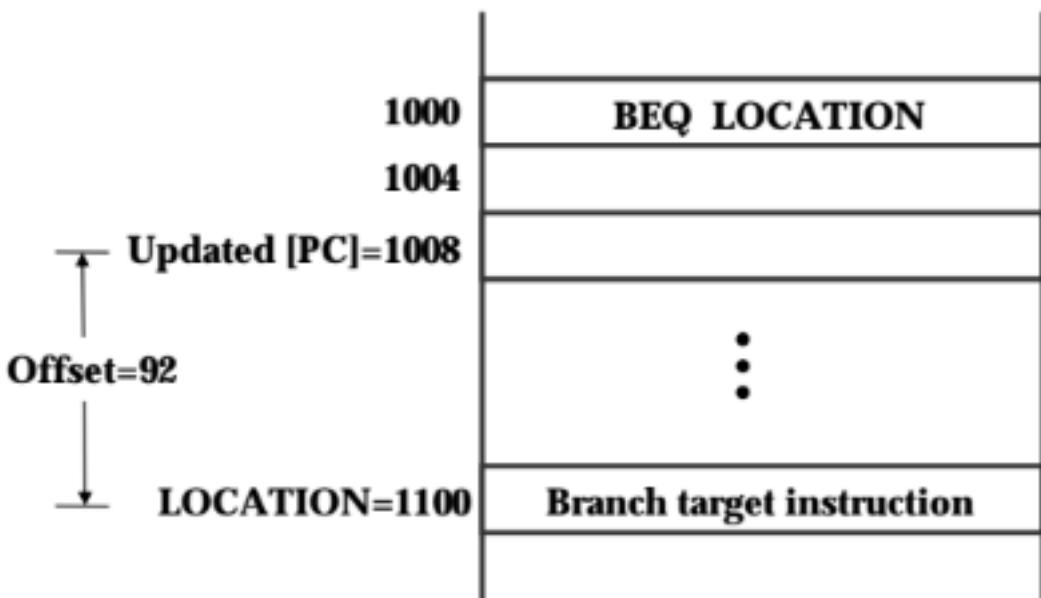
- and r0, r1, r2
- biceq r3, r3, #1
- eors r4, r5, #5

Instrucciones de Bifurcación

Las instrucciones de bifurcación condicionales contienen un desplazamiento de 24 bits con signo que se agrega al contenido actualizado del *PC* para generar la dirección de destino de la bifurcación. El formato de las instrucciones de bifurcación es el que se muestra a continuación:



Por ejemplo, la instrucción BEQ (Bifurca si igual a 0) bifurca si el flag Z está seteado en 1:



Bifurcaciones condicionales

Mnemonic	Interpretación	Aplicación
B	Incondicional	Siempre bifurcar
BAL	Siempre	Siempre bifurcar
BEQ	Igual	Comparación igual o resultado es cero
BNE	Distinto	Comparación distinta o resultado no es cero
BPL	Positivo	Resultado positivo o cero
BMI	Negativo	Resultado negativo
BCC	Flag Carry vacío	Operación aritmética no resultó en acarreo hacia afuera
BLO	Menor	Comparación sin signo con resultado menor
BCS	Flag Carry seteado	Operación aritmética resultó en acarreo hacia afuera
BHS	Mayor o igual	Comparación sin signo con resultado mayor o igual
BVC	Flag Overflow vacío	Operación de entero con signo: resultado sin overflow
BVS	Flag Overflow seteado	Operación de entero con signo: resultado con overflow
BGT	Mayor	Comparación de entero con signo: mayor

BGE	Mayor o igual	Comparación de entero con signo: mayor o igual
BLT	Menor	Comparación de entero con signo: menor
BLE	Menor o igual	Comparación de entero con signo: mayor o igual
BHI	Mayor	Comparación de entero sin signo: mayor
BLS	Menor o igual	Comparación de entero sin signo: menor o igual

Pseudo-Instrucciones

La pseudo-instrucción `adr Rd, direccion` mantiene la dirección del valor de 32 bits en `Rd`. Esta instrucción no es realmente una instrucción de máquina sino que el ensamblador elige instrucciones de máquina reales apropiadas para implementar pseudo-instrucciones. Por ejemplo, la combinación de la instrucción de máquina `ldr r2, puntero` y la directiva de declaración de datos `puntero dcd num1` es una forma de implementar la pseudo-instrucción `adr r2, num1`.

ALIGN: ajusta el contador de dirección a palabra (word)

END: nada más que ensamblar

EQU: reemplazo de símbolo

```
loopcnt EQU 5
```

Subrutinas

Se utiliza una instrucción de branch and link (`BL`) para llamar a una subrutina.

La dirección de retorno se carga en el registro `R14`, que actúa como un *link register*. Cuando las subrutinas están anidadas, el contenido del *link register* debe ser guardado en una pila por la misma subrutina. El registro `R13` se usa como puntero para esta pila.

Ejemplo

Programa llamador

```
ldr    r1, n
ldr    r2, puntero
bl    sumalista
str    r0, suma
...
```

Subrutina

```
.sumalista:  
    @ salva r3 y dir. de retorno en r14 en la pila usando r13 como puntero a pila  
    stmfd r13, {r3, r14}  
    mov r0, #0  
.loop:  
    ldr r3, [r2], #4  
    add r0, r0, r3  
    subs r1, r1, #1  
    bgt loop  
    @ restaura r3 y carga la dir. de retorno en r15  
    ldmfd r13!, {r3, r15}
```

Movimiento de datos

Operaciones

```
mov operand1 <= operand2  
mvn operand1 <= NOT operand2
```

Sintaxis

```
<operation>{<cond>} {S} Rd, operand2
```

Ejemplos

- mov r0, r1
- movs r2, #10
- mvneq r1, #0

Barrel Shifter

ARM no tiene instrucciones de shift. En su lugar tiene un barrel shifter que provee un mecanismo que lleva a cabo shifts como parte de otras instrucciones.

Operaciones soportadas por el Barrel Shifter

Shift a Izquierda (LSL)

Shift a la izquierda según la cantidad especificada (multiplica por potencias de 2). Por ejemplo:

LSL # 5 @ multiplica por 32



Shift Lógico a Derecha (LSR)

Shift a la derecha según la cantidad especificada (divide por potencia de 2). Por ejemplo:

LSR # 5 @ divide por 32



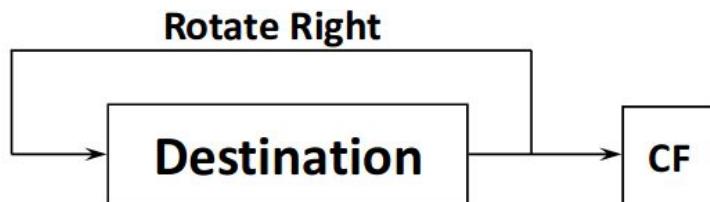
Shift Aritmético a Derecha (ASR)

Shift a la derecha según la cantidad especificada (divide por potencia de 2) preservando el bit de signo. Por ejemplo:

ASR # 5 @ divide por 32



Rotate Right (ROR)



Rotate Right Extended (RRX)



Ejemplos

```
mov r2, r0, lsl #2          @ R2 = R0x4
add r9, r5, r5, lsl #3      @ R9 = R5+R5x8 ó R9=R5x9
rsb r9, r5, r5, lsl #3      @ R9 = R5x8-R5 ó R9=R5x7
sub r10, r9, r8, lsr #4     @ R10 = R9-R8/16
mov r12, r4, ror r3         @ R12 = R4 rotado der. por el valor en R3
```

Instrucciones Load/Store

Transferencia de datos de un registro

ldr	@ Load Word
str	@ Store Word
ldrb	@ Load Byte
strb	@ Store Byte
ldrh	@ Load Halfword
strh	@ Store Halfword
ldrsb	@ Load Signed Byte (load and extent sign to 32 bits)
ldrsh	@ Load Signed Halfword (load and extent sign to 32 bits)

Ejecución condicional

Estas instrucciones pueden ser ejecutadas condicionalmente insertando el condition code apropiado luego de LDR / STR. Ejemplo: LDREQB

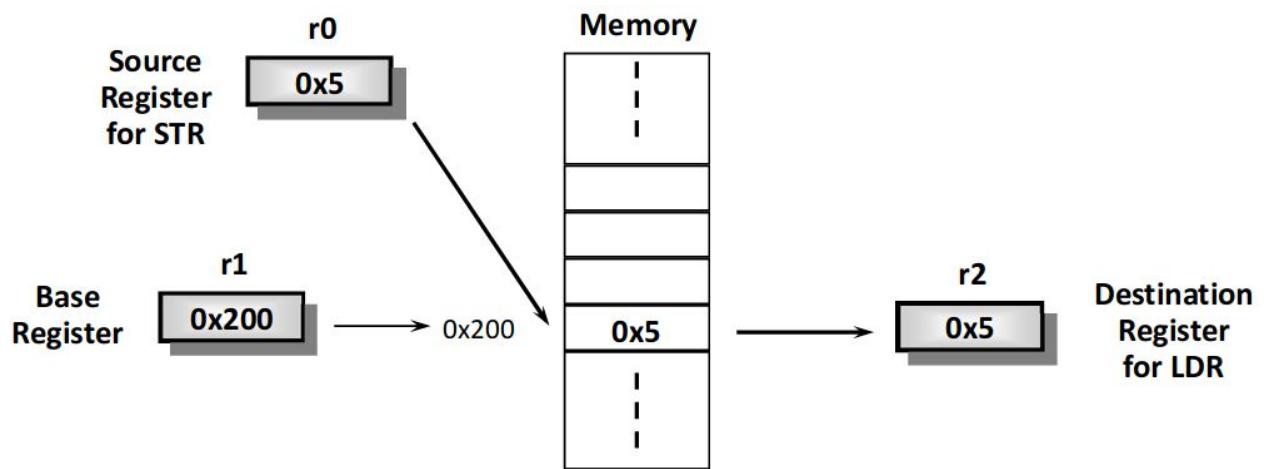
Sintaxis

```
<LDR|STR>{<cond>} {<size>} Rd, <address>
```

Registro Base

La dirección de memoria accedida está contenido en un registro base

```
str r0, [r1] @ Almacena (r0) en la memoria apuntada por r1
ldr r2, [r1] @ Carga en (r2) el valor apuntado por r1
```



Offset desde el Registro Base

Las instrucciones Load/Store pueden acceder la dirección contenida en el registro base así como una dirección offset desde el Registro Base. Este offset puede ser:

- Valor inmediato: BPF s/s de 12 bits
- Registro (opcionalmente shifteado por un valor inmediato)

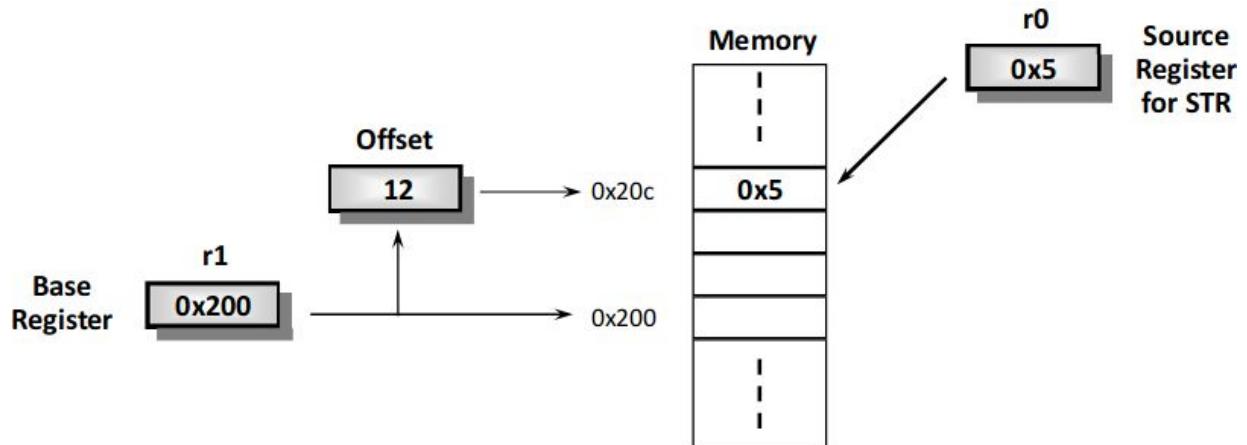
El offset puede ser sumado o restado del Registro Base prefijando el valor del offset o registro con + (por defecto) o -.

El offset puede aplicarse

- Antes de que se realice la transferencia: Direccionamiento Pre-indexado
 - Puede auto-incrementarse el Registro Base agregando ! al final de la instrucción.
- Despues de que se realice la transferencia: Direccionamiento Post-indexado
 - Causando que el Registro Base se vea auto-incrementado

Direccionamiento Pre-indexado

Ejemplo: STR r0, [r1, #12]



Para almacenar en la dirección 0x1f4

`STR r0, [r1, #-12]`

Para auto-incrementar el Registro Base a 0x20c

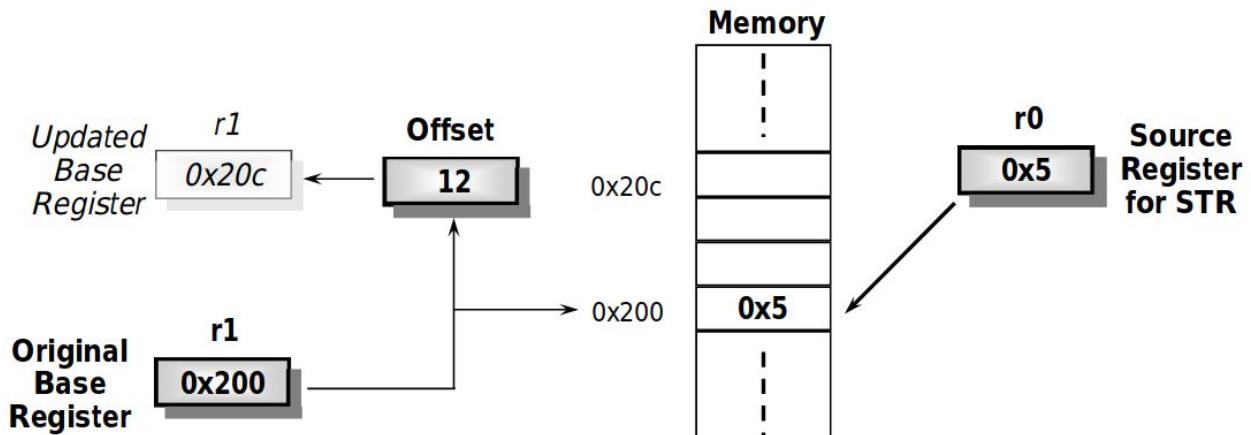
`STR r0, [r1, #12]!`

Si $(r2)=3$ puede accederse a $0x20c$ multiplicándolo por 4

`STR r0, [r1, r2, LSL #2]`

Direccionamiento Post-indexado

Ejemplo: `STR r0, [r1], #12`



Para auto-incrementar el Registro Base a la dirección 0x1f4

`STR r0, [r1], #-12`

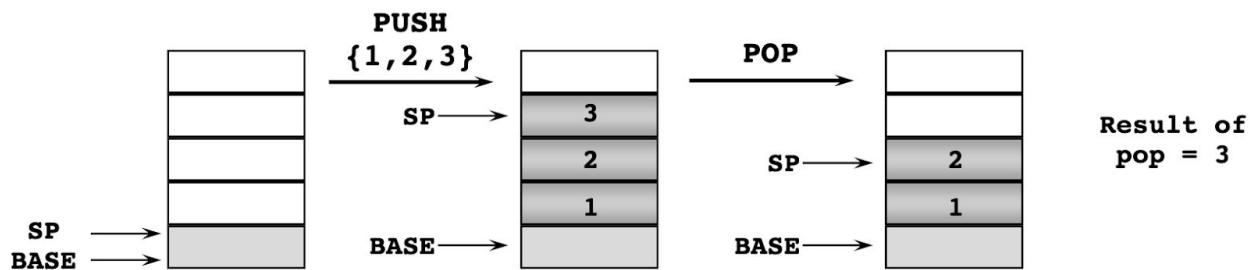
Si $(r2)=3$ puede auto-incrementarse a $0x20c$ multiplicándolo por 4

`STR r0, [r1], r2, LSL #2`

Pila (Stack)

Una pila es un área de la memoria que crece a medida que nuevos datos se insertan (push) en la parte superior de la misma, y se reduce a medida que los datos se remueven (pop) desde la parte superior. Dos punteros definen los límites actuales de la pila:

- Un puntero de base: apunta a la parte inferior de la pila (la primera posición).
- Un puntero a la pila: apunta a la parte superior actual de la pila.



Pilas y subrutinas

Uno de los usos de las pilas es crear un área de memoria temporal para los registros para las subrutinas. Cualquier registro que sea necesario puede ser agregado (push) a la pila al principio de la subrutina y tomado (pop) desde la pila para recuperar el valor antes de volver de la subrutina.

```
stmdfd sp!,{r0-r12, lr} @ apilar los registros y dir. de retorno
```

```
ldmfd sp!,{r0-r12, pc} @ desapilar los registros y retornar
```

Interrupción de Software (SWI)

Una interrupción de software es un tipo de interrupción causada por una instrucción especial en el set de instrucciones. El software invoca una interrupción de software, a diferencia de una interrupción de hardware, y se considera una de las formas de invocar llamadas al sistema.

SWIs en ARSim#

En ARMSim# se utilizan interrupciones de software para operaciones comunes de entrada/salida.

La sintaxis para hacer un llamada es: swi <swi code>

SWI Codes

Code	Description and Action	Inputs	Outputs	EQU

0x00	Mostrar carácter por consola	R0: el carácter		SWI_PrChr
0x02	Mostrar string por consola	R0: dirección de un string terminado en null		
0x11	Detener la ejecución			SWI_Exit
0x12	Asignar Bloque de Memoria	R0: tamaño del Bloque en bytes	R0: dirección del Bloque	SWI_MeAlloc
0x13	Desasignar todo Bloque de Memoria			SWI_DAlloc
0x66	Abrir Archivo (Modo 0: Input / Modo 1: Output / Modo 2: Append)	R0: dirección de un string terminado en null con el nombre del archivo R1: Modo	R0: manejador de archivo (-1 si el archivo no abre)	SWI_Open
0x68	Cerrar Archivo	R0: manejador de archivo		SWI_Close
0x69	Escribir string	R0: manejador de archivo o Stdout (1) R1: dirección de un string terminado en null		SWI_PrStr
0x6a	Leer string desde un Archivo	R0: manejador de archivo R1: dirección destino R2: max bytes a almacenar	R0: número de bytes almacenados	SWI_RdStr
0x6b	Escribir entero	R0: manejador de archivo o Stdout (1) R1: entero		SWI_PrInt
0x6c	Leer entero desde un Archivo	R0: manejador de archivo	R0: entero	SWI_RdInt
0x6d	Obtener el tiempo actual (ticks)		R0: número de ticks (milisegundos)	SWI_Timer

Modos de direccionamiento

Nombre	Nombre Alternativo	Ejemplos
Registro a registro	Registro directo	mov r0, r1
Absoluto	Directo	ldr r0, mem
Literal	Inmediato	mov r0, #15 add r1, r2, #12
Indexado, base	Registro indirecto	ldr r0, [r1]
Pre-Indexado, base con desplazamiento	Registro indirecto con offset	ldr r0, [r1, #4]
Pre-indexado, autoindexado	Registro indirecto con pre-incremento	ldr r0, [r1, #4]!
Post-indexado, autoindexado	Registro indirecto con post-incremento	ldr r0, [r1], #4
Doble registro indirecto	Registro indirecto indexado	ldr r0, [r1, r2]
Doble registro indirecto escalado	Registro indirecto indexado escalado	ldr r0, [r1, r2, lsl #2]
Relativo al PC		ldr r0, [PC, #offset]

Modo pre-indexado

La dirección efectiva del operando es la suma de los contenidos del registro base Rn y un offset.

[Rn, #offset]

$$DE = [Rn] + offset$$

[Rn, ±Rm, shift]

$$DE = [Rn] \pm [Rm] \text{ shifteado}$$

Modo pre-indexado con reescritura

La dirección efectiva del operando se genera de la misma manera que en el modo Pre-indexado pero la dirección efectiva se escribe también en Rn.

[Rn, #offset]!
DE=[Rn]+offset
Rn \leftarrow [Rn]+offset

[Rn, \pm Rm, shift]
DE=[Rn] \pm [Rm] shifteado
Rn \leftarrow [Rn] \pm [Rm] shifteado

Modo post-indexado

La dirección efectiva del operando es el contenido de Rn. El desplazamiento se agrega a esta dirección y el resultado se escribe de nuevo en Rn.

[Rn], #offset
DE=[Rn]
Rn \leftarrow [Rn]+offset

[Rn], \pm Rm, shift
DE=[Rn]
Rn \leftarrow [Rn] \pm [Rm] shifteado

Anexo

Sumario de Set de Instrucciones

Fuente: [ARM instruction summary](#)

Operation	Assembly syntax
Move	Move
	MOV{cond}{S} Rd, <Oprnd2>
	Move NOT
	MVN{cond}{S} Rd, <Oprnd2>
	Move SPSR to register
	MRS{cond} Rd, SPSR
	Move CPSR to register
	MRS{cond} Rd, CPSR
Arithmetic	Move register to SPSR
	MSR{cond} SPSR{field}, Rm
	Move register to CPSR
	MSR{cond} CPSR{field}, Rm
	Move immediate to SPSR flags
	MSR{cond} SPSR_f, #32bit_Imm
	Move immediate to CPSR flags
	MSR{cond} CPSR_f, #32bit_Imm
Logic	Add
	ADD{cond}{S} Rd, Rn, <Oprnd2>
	Add with carry
	ADC{cond}{S} Rd, Rn, <Oprnd2>
	Subtract
	SUB{cond}{S} Rd, Rn, <Oprnd2>
	Subtract with carry
	SBC{cond}{S} Rd, Rn, <Oprnd2>
	Subtract reverse subtract
Shift	Subtract reverse subtract with carry
	RSB{cond}{S} Rd, Rn, <Oprnd2>
	Subtract reverse subtract with carry
	RSC{cond}{S} Rd, Rn, <Oprnd2>
	Multiply
	MUL{cond}{S} Rd, Rm, Rs
	Multiply accumulate
	MLA{cond}{S} Rd, Rm, Rs, Rn
Floating-point	Multiply unsigned long
	UMULL{cond}{S} RdLo, RdHi, Rm, Rs

	Multiply unsigned accumulate long	UMLAL{cond}{S} RdLo, RdHi, Rm, Rs
	Multiply signed long	SMULL{cond}{S} RdLo, RdHi, Rm, Rs
	Multiply signed accumulate long	SMLAL{cond}{S} RdLo, RdHi, Rm, Rs
	Compare	CMP{cond} Rd, <Oprnd2>
	Compare negative	CMN{cond} Rd, <Oprnd2>
Logical	Test	TST{cond} Rn, <Oprnd2>
	Test equivalence	TEQ{cond} Rn, <Oprnd2>
	AND	AND{cond}{S} Rd, Rn, <Oprnd2>
	EOR	EOR{cond}{S} Rd, Rn, <Oprnd2>
	ORR	ORR{cond}{S} Rd, Rn, <Oprnd2>
	Bit clear	BIC{cond}{S} Rd, Rn, <Oprnd2>
Branch	Branch	B{cond} label
	Branch with link	BL{cond} label
	Branch and exchange instruction set	BX{cond} Rn
Load	Word	LDR{cond} Rd, <a_mode2>
	Word with user-mode privilege	LDR{cond}T Rd, <a_mode2P>
	Byte	LDR{cond}B Rd, <a_mode2>
	Byte with user-mode privilege	LDR{cond}BT Rd, <a_mode2P>
	Byte signed	LDR{cond}SB Rd, <a_mode3>
	Halfword	LDR{cond}H Rd, <a_mode3>
	Halfword signed	LDR{cond}SH Rd, <a_mode3>

	Multiple block data operations	-
	Increment before	LDM{cond} IB Rd{!}, <reglist>{^}
	Increment after	LDM{cond} IA Rd{!}, <reglist>{^}
	Decrement before	LDM{cond} DB Rd{!}, <reglist>{^}
	Decrement after	LDM{cond} DA Rd{!}, <reglist>{^}
	Stack operation	LDM{cond}<a_mode4L> Rd{!}, <reglist>
	Stack operation, and restore CPSR	LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^
	Stack operation with user registers	LDM{cond}<a_mode4L> Rd{!}, <reglist>^
Store	Word	STR{cond} Rd, <a_mode2>
	Word with user-mode privilege	STR{cond}T Rd, <a_mode2P>
	Byte	STR{cond}B Rd, <a_mode2>
	Byte with user-mode privilege	STR{cond}BT Rd, <a_mode2P>
	Halfword	STR{cond}H Rd, <a_mode3>
	Multiple block data operations	-

	Increment before	STM{cond} IB Rd{!}, <reglist>{^}
	Increment after	STM{cond} IA Rd{!}, <reglist>{^}
	Decrement before	STM{cond} DB Rd{!}, <reglist>{^}
	Decrement after	STM{cond} DA Rd{!}, <reglist>{^}
	Stack operation	STM{cond}<a_mode4S> Rd{!}, <reglist>
	Stack operation with user registers	STM{cond}<a_mode4S> Rd{!}, <reglist>^
Swap	Word	SWP{cond} Rd, Rm, [Rn]
	Byte	SWP{cond}B Rd, Rm, [Rn]
Coprocessors	Data operation	CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>
	Move to ARM register from coprocessor	MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Move to coprocessor from ARM register	MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Load	LDC{cond} p<cpnum>, CRd, <a_mode5>
	Store	STC{cond} p<cpnum>, CRd, <a_mode5>
Software interrupt		SWI 24bit_Imm

Introducción a la Familia de Arquitecturas ARM

Versión	Familia
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	Strong ARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10XE
ARMv6	ARM11
ARMv7	Cortex

Ejemplo en C

C

$y = a * (b + c);$

ARM

```
adr r4,b      @ obtener dirección de b
ldr r0,[r4]    @ obtener valor de b
adr r4,c      @ obtener dirección de c
ldr r1,[r4]    @ obtener valor de c
add r2,r0,r1  @ calcular resultado parcial
adr r4,a      @ obtener dirección de a
ldr r0,[r4]    @ obtener valor de a
mul r2,r2,r0  @ calcular valor final de y
adr r4,y      @ obtener dirección de y
str r2,[r4]    @ almacenar y
```

¿Por qué enseñar lenguaje assembler y arquitectura de computadoras?

Considerando que la ciencia de la computación está relacionada principalmente con el uso de la computadora puede argumentarse que el lenguaje ensamblador es irrelevante. Ahora, ¿el cirujano estudió metalurgia para comprender cómo funciona un bisturí? ¿estudia termodinámica el piloto para comprender cómo funciona un motor de reacción? ¿una persona que lee noticias estudia electrónica para entender cómo funciona una cámara de fotos? La respuesta a todas estas preguntas es *no*. Entonces, ¿por qué enseñar lenguaje de ensamblador y arquitectura de computadoras al estudiante? En primer lugar, porque la educación no es lo mismo que la

formación. El estudiante de ciencias de la computación no está solo capacitado para usar varias herramientas de las computadoras. Un curso universitario que lleve a un título de grado también debe cubrir la historia y las bases teóricas de la materia. Sin un conocimiento de la arquitectura informática, el científico informático no puede comprender cómo se han desarrollado las computadoras y de qué son capaces.

Se puede presentar un caso sólido para la enseñanza continua del lenguaje ensamblador dentro del plan de estudios de informática. Sin embargo, un lenguaje ensamblador no puede enseñarse como si fuera otro lenguaje de programación de propósito general. Tal vez, más que cualquier otro componente del plan de estudios de ciencias de la computación, la enseñanza de un lenguaje ensamblador admite una amplia gama de temas en el corazón de la ciencia de la computación. Un lenguaje ensamblador no debe usarse sólo para ilustrar algoritmos, sino para demostrar lo que realmente está sucediendo dentro de la computadora.

¿Por qué aprender ensamblador?

Dado el avance de los lenguajes de alto nivel, ¿por qué es necesario aprender programación en lenguaje ensamblador? Las razones son:

1. La mayoría de los usuarios industriales de microcomputadoras programan en lenguaje ensamblador.
2. Muchos usuarios de microcomputadoras continuarán programando en lenguaje ensamblador ya que necesitan el control que estos proporcionan.
3. Ningún lenguaje de alto nivel adecuado ha sido ampliamente disponible o estandarizado.
4. Muchas aplicaciones requieren la eficiencia del lenguaje ensamblador.
5. La comprensión del lenguaje ensamblador puede ayudar a evaluar los lenguajes de alto nivel.
6. Casi todos los programadores de microcomputadoras finalmente encuentran que necesitan algún conocimiento de lenguaje ensamblador, generalmente para depurar programas, escribir rutinas de E/S, acelerar o acortar las secciones críticas de los programas escritos en lenguajes de alto nivel, utilizar o modificar funciones del sistema, y entender los programas de otras personas.

¿Cuál es el concepto de palabra que maneja Intel? Dd significa double word. Cada palabra tiene 4 bytes, 32 bits. Así como específicas dd, podes poner db (data byte). Ahí siempre pones cuanta memoria reservas para el campo. Los que tienen dos puntos son etiquetas.

Caso de estudio: Intel

Isa: registros

Hay varios tipos de registros. El registro A se utiliza para instrucciones aritméticos y lógicas. El registro B es el base y se suele utilizar para direccionamiento de operandos. El registro C se utiliza para aritmética o como contador. El D se suele utilizar cuando necesitas una dupla de registros, porque con uno no alcanza.

Para los registros generales, cada registro tiene 16 bits y se subdivide en dos subregistros. Cada subregistro tiene 8 bits. Podes acceder al registro general como a los subregistros.

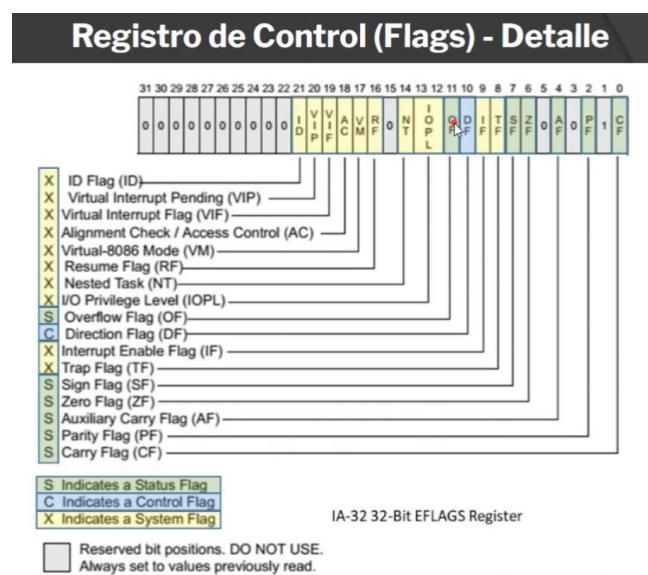
Pueden estar extendidos a 32 bits, pero aun se puede acceder a los de 16. También podes tener más generales, con registros de 64 bits. Se puede trabajar con cualquiera de ellos.

Cuando se va al registro general, se pueden subdividir en mitades, cuartos y octavos (32, 16 y 8 bits).

Después tenés los registros índices, que se suelen usar para operaciones de manejos de cadenas, para apuntar al operando “origen”. Vos tenés partes bajas y altas, las más bajas son las más chicas. También podes acceder al destino.

El registro de la memoria Stack (pila) tenés las direcciones ordenadas mayor a menor, teniendo la menor en el tope. Tenes un puntero a la base de la pila y al tope. A mayor tope, menor es la dirección.

El registro de instrucción y control contiene la dirección de la próxima instrucción a ejecutar. Los flags asociados a cada registro me indica el estado del mismo que me sirve para tomar decisiones según su valor. De todos los bits que tiene, encendes o apagas un registro según lo que haya pasado.



Direccionamiento

Se trata de que forma accedes a los datos a la hora de operar. Si está implicito, en el mismo código de operación se encuentra el dato. Por ejemplo si se usa CBW, se da por sentado que es el que está en el registro A.

Otro modo de operación es a partir de acceder a registros, donde uno explícita que registro será.

Cuando se usa un operando inmediato, este se aloja / se define en la misma instrucción.

Un operador directo es cuando a partir de corchetes y el nombre de una variable accedo a un valor,

- **Directo:** El dato está en memoria referenciado por el nombre de un campo

Ej. MOV RAX, [VARIABLE]

Uno indirecto:

- **Registro Indirecto:** El dato está en memoria apuntado por un registro base o índice.

Ej. MOV EAX, [EBX]

Relativo:

- **Registro Relativo:** El dato está en memoria apuntado por un registro base o índice más un desplazamiento.

Ej. MOV RAX, [RBX+4]

 MOV RAX, [VECTOR+RBX]

Siempre sumo un numero o variable

Base + indice:

- **Base + Índice:** El dato está en memoria apuntado por un registro base más un registro índice.

Ej. MOV [RBX+RTI], CL

Sumo dos registros.

- ◎ **Base Relativo + Índice:** El dato está en memoria apuntado por un registro base más un registro índice más un desplazamiento.
Ej. MOV RAX, [RBX+RDI+4]
MOV RAX, [VECTOR+RBX+RDI]

Memoria y tipos de dato

podes guardar ASCII, numero entero o decimal. La celda de memoria tiene 1 byte y la palabra en intel es de 2 bytes. Esto difiere de una infraestructura a otra. Podes tener dobles o cuadruples palabras.

Endianes

Es el método aplicado para almacenar datos mayores a un byte en una computadora respecto a la dirección que se le asigna a cada uno de ellos en la memoria.

Existen 2 métodos:

- **Big-Endian:** determina que el orden en la memoria coindice con el orden lógico del dato.
“el dato final en la mayor dirección”
Ej. IBM Mainframe
- **Little-Endian :** es a la inversa, el dato inicial para la lógica se coloca en la mayor dirección y el dato final en la menor.
“el dato final en la menor dirección”

Intel trabaja en Little-Endian.

Endiannes - Ejemplos

- ◎ **Caso 1:** Definición de un área de memoria con contenido inicial definido en formato carácter

```
msg      db      'HOLA'
```

48	4F	4C	41
1A1	1A2	1A3	1A4

Aquí la posición de memoria que toma cada carácter recibido es la que por intuición uno asume, o sea, la letra 'H' en la dirección menor

Endiannes - Ejemplos

- ◎ **Caso 2:** Definición de un área de memoria con contenido inicial definido en formato numérico

```
num      dw 4666 ; es 123A en he
```

3A	12
1A5	1A6

```
num2    dd 12345678h
```

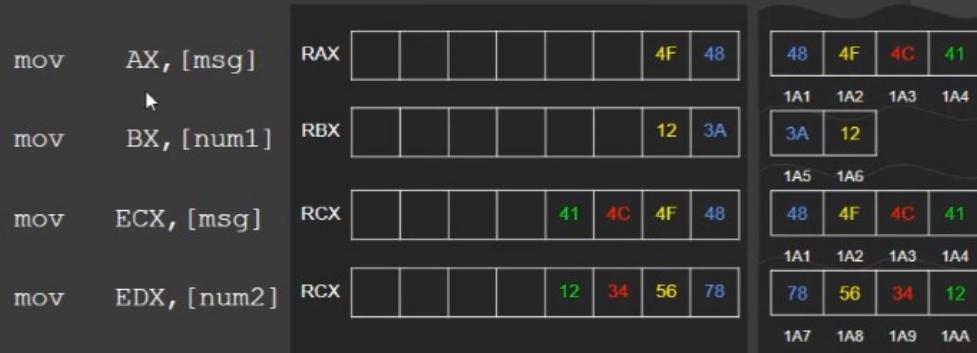
78	56	34	12
1A7	1A8	1A9	1AA

Aquí es donde se observa la ubicación de los bytes con el método Little-Endian, el byte menos significativo se ubica en la dirección de memoria menor que el byte más significativo

Cuando se le agrega la h, el número esta en base 16, por eso se guarda directamente como está escrito en el segundo ejemplo.

Endiannes - Ejemplos

- ④ **Caso 3:** Se ejecuta una copia de memoria a registro



La parte alta del registro contiene el byte de orden superior de memoria, y la parte baja del registro contiene el byte de orden inferior.

Moves según el tamaño del registro destino. Si vos tenés en la memoria little endian, se copia “al derecho en el registro.

Ensamblador NASM

Estos son temas asociados a la herramienta que ensambla, no a la arquitectura.

Directivas al ensamblador / Pseudo-instrucciones

Son instrucciones para que el ensamblador tome alguna acción durante el proceso de ensamblado. No se traducen a código máquina.

section	indica el comienzo de un segmento
global	indica que una etiqueta declarada en el programa es visible para un programa externo
extern	indica que una etiqueta usada en el programa pertenece a un programa externo (donde habrá sido declarada global)
db, dw, dd,dq, dt	sirven para definir áreas de memoria (variables) con contenido inicial
resb, resw, resid, resq, rest	sirven para definir áreas de memoria (variables) sin contenido inicial
times	repite una definición la cantidad de veces que se indica
%macro %endmacro	indican el inicio y final de un bloque para definir una macro
%include	permite incluir el contenido de un archivo

Esto sirve para decirle que es cada cosa al propio ensamblador y así sepa que hacer con eso.

Componentes minimos:

Estructura de un programa

```
global main

section .data
;variables con contenido inicial

section .bss ;block starting symbol
;variables sin contenido inicial

section .text
;instrucciones
main:
    |
    |
    ret
```

Definición y reserva de campos en memoria

- Con contenido inicial (en section .data)

db define byte (1 byte)
dw define word (2 bytes)
dd define double (4 bytes)
dq define quad (8 bytes)
dt define ten (10 bytes)

- Sin contenido inicial (en section .bss)

resb reserve byte (1 byte)
resw reserve word (2 bytes)
resd reserve double (4 bytes)
resq reserve quad (8 bytes)
rest reserve ten (10 bytes)

Definición y reserva de campos en memoria

○ Ejemplos (2/4)

Definicion	Bytes reservados	Contenido memoria
decimal1 db 11	1	0B
decimal2 dw -11	2	F5 FF
decimal3 dd 12345	4	39 30 00 00
decimal4 dq -1	8	FF FF FF FF FF FF FF FF
hexa1 db -0Bh	1	F5
hexa2 dw 0Ch	2	0C 00
hexa3 dd FFFFh	4	FF FF 00 00
hexa4 dq 96B43Fh	8	3F B4 96 00 00 00 00 00

En la memoria se guarda en base 16 y en little endian. Recordar que si tiene h, le estas pasando literalmente en base 16.

Definición y reserva de campos en memoria

○ Ejemplos (3/4)

Definicion	Bytes reservados	Contenido memoria
octal1 db 13o	1	0B
octal2 dw 71o	2	39 00
octal3 dd 10o	4	08 00 00 00
binario1 db 1011b	1	0B
binario2 dw 1011b	2	0B 00
binario3 dd -1000b	4	F8 FF FF FF
truncado db 2571 ;0A0Bh	1	0B
noTruncado dw 2571 ;0A0Bh	2	0B 0A

Definición y reserva de campos en memoria

○ Ejemplos (4/4)

Definicion		Bytes reservados	Contenido memoria
letra	db 'A'	1	41
letra2	dw 'A'	2	41 20
letra3	db 'a'	1	61
cadena	db 'hola'	4	68 6F 6C 61
cadena2	dw 'ola'	4	6F 6C 61 20
numero	db '12'	2	31 32
vector1	times 3 db 'A'	3	41 41 41
vector2	times 3 db 'A', 0	6	41 00 41 00 41 00
registro	times 0 db 'A'	0	n/a

Para las cadenas soles usar una medida de “db”.

Macros e inclusión de archivos

Macros

Son secuencias de instrucciones asignadas a un nombre que pueden usarse en cualquier parte del programa.

La sintaxis es:

```
%macro macro_name    number_of_params  
<macro body>  
%endmacro
```

Macros

Sin parámetros

```
1 %macro mPuts 0  
2     sub      rsp,8  
3     call     puts  
4     add      rsp,8  
5 %endmacro  
6 global main  
7 extern puts  
8  
9 section .data  
10    mensaje   db    "Hola mundo!",0  
11    mensaje2  db    "Chau!",0  
12  
13 section .text  
14 main:  
15     mov      rdi,mensaje  
16     mPuts  
17     mov      rdi,mensaje2  
18     mPuts  
19     ret
```

Con 1 parámetro

```
1 %macro mPuts 1  
2     mov      rdi,%1  
3     sub      rsp,8  
4     call     puts  
5     add      rsp,8  
6 %endmacro  
7 global main  
8 extern puts  
9  
10    section .data  
11    mensaje   db    "Hola mundo!",0  
12    mensaje2  db    "Chau!",0  
13  
14    section .text  
15    main:  
16    mPuts      mensaje  
17    mPuts      mensaje2  
18    ret
```

No funciona como las funciones porque en memoria ocupa como si estuviera copiado dos veces. El ensamblador lo que hace es escribirlo por mi.

Inclusión de archivos

The screenshot shows a terminal window with two tabs: 'misMacros.asm' and 'test-include.asm'. The 'misMacros.asm' tab contains the following assembly code:

```
1 %macro mPuts 1
2     mov rdi,%1
3     sub rsp,8
4     call puts
5     add rsp,8
6 %endmacro
7 extern puts
```

The 'test-include.asm' tab contains the following assembly code:

```
1 %include "misMacrosL.asm"
2 global main
3 section .data
4     mensaje db "Hola mundo!!",0
5
6 section .text
7 main:
8     mPuts mensaje
9     ret
```

Below the tabs, the terminal shows the command-line steps and the resulting output:

```
PS C:\Users\Darío\Documents\Orga\include> nasm test-include.asm -fwin64
PS C:\Users\Darío\Documents\Orga\include> gcc test-include.obj
PS C:\Users\Darío\Documents\Orga\include> ./a.exe
Hola mundo!!
PS C:\Users\Darío\Documents\Orga\include>
```

Trabajando en grupo, si todos usamos las mismas macros usamos esto.

Instrucciones

Instrucciones - Transferencia y Copia

MOV op1, op2

Copia el valor del 2do operando en el primer operando.

Combinaciones	Ejemplos en NASM
MOV <reg>,<reg>	MOV AH,BL MOV AX,BX MOV ECX, EAX MOV RDX,RCX
MOV <reg>,<mem>	MOV CH,[VARIABLE_8] (*) MOV CX, [VARIABLE_16] (*) MOV ECX, [VARIABLE_32] (*) MOV RDX, [VARIABLE_64] (*)
MOV <reg>,<inm>	MOV DL,7o MOV CX,2450h MOV EAX,0h MOV RDX,28h

MOV op1, op2

Combinaciones	Ejemplos en NASM
MOV <mem>,<reg>	MOV [VARIABLE_8],AH (*) MOV [VARIABLE_16],AX (*) MOV [VARIABLE_32],EAX (*) MOV [VARIABLE_64],RAX (*) MOV VARIABLE_64,RAX
MOV <long>,<mem>,<inm>	MOV byte[VARIABLE_8],2Ah MOV word[VARIABLE_16],777o MOV dword[VARIABLE_32],1234 MOV qword[VARIABLE_64],1234 MOV [VARIABLE_64],4321

Pulsa **Esc** para salir del modo de pantalla completa

Manejo de Parámetros

```
result funcName(p1, p2, p3, p4, p5, p6, p7, ..., pn)

Linux:    rax          rdi rsi rdx rcx r8  r9  stack...
Windows:   rax          rcx rdx r8  r9  stack...
```

los parametros se cargan antes en un call.

Ajustes para llamados a rutinas/funciones externas e internas

LINUX	WINDOWS
...	...
sub rsp,8 call rutina_externa add rsp,8	sub rsp,32 call rutina_externa add rsp,32
...	...
sub rsp,8 call rutina_interna add rsp,8	call rutina_interna

Salida por Pantalla

Función puts

Imprime un string hasta que encuentra un 0 (cero binario). Agrega el carácter de fin de línea a la salida

```
int puts(const char *str)
```

<pre>;LINUX global main extern puts section .data cadena db "Hola",0 section .text main: mov rdi,cadena sub rsp,8 call puts add rsp,8</pre>	<pre>;WINDOWS global main extern puts section .data cadena db "Hola",0 section .text main: mov rcx,cadena sub rsp,32 call puts add rsp,32</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Ver como funciona cada registro

The screenshot shows a code editor window with the file name 'holamundo.asm'. The code is as follows:

```
1 global main
2 extern puts
3
4 section .data
5     mensaje db "Hola mundo",0
6
7 section .text
8 main:
9     mov rdi,mensaje
10    sub rsp,8
11    call puts
12    add rsp,8
13
14    ret
```

Instrucciones para ensamblar:

```
dario@dario-HP-EliteBook-8440p:~/Orga/intel/Linux
File Edit View Search Terminal Help
dario@dario-HP-EliteBook-8440p:~/Orga/Intel/Linux$ nasm holam.asm -f elf64
dario@dario-HP-EliteBook-8440p:~/Orga/Intel/Linux$ gcc holam.o -o hola.out -no-pie
dario@dario-HP-EliteBook-8440p:~/Orga/Intel/Linux$ ./hola.out
Hola mundo!
dario@dario-HP-EliteBook-8440p:~/Orga/Intel/Linux$
```

como lo puso el profe:

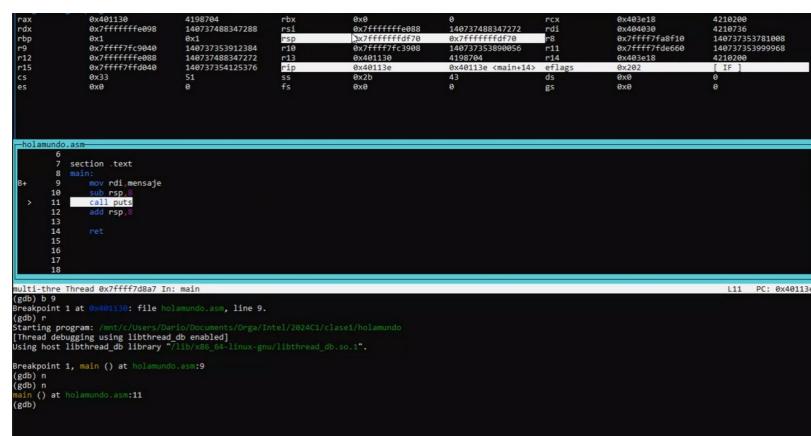
```
gl
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ nasm holamundo.asm -f elf64
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ ls
holamundo.asm holamundo.o
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ gcc holamundo.o -no-pie
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ ls
a.out holamundo.asm holamundo.o
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ gcc holamundo.o -no-pie -o holamundo
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ ls
a.out holamundo holamundo.o
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ ./holamundo
Hola mundo
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$
```

como uso debugger:

```
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ nasm holamundo.asm -f elf64 -g -F dwarf -l
holamundo.lst
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ ls
a.out holamundo holamundo.lst holamundo.o
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ gcc holamundo.o -no-pie -o holamundo
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ ./holamundo
Hola mundo
dario@DESKTOP-1KL5D7K:/mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1$ gdb holamundo
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
 <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from holamundo...
(gdb)
```

interface:



The screenshot shows the GDB debugger interface. At the top, assembly code for a function named 'main' is displayed, starting with a section .text and a call instruction to main. Below the assembly code, the register dump shows the state of various CPU registers (rax, rdx, rcx, etc.) and memory locations. The bottom part of the screen shows the GDB command prompt with some initial commands entered.

```
multi-thread Thread 0xfffff7d8a7.Ini: main
(gdb) b 9
Breakpoint 1 at 0x401130: file holamundo.asm, line 9.
(gdb) n
Starting program: /mnt/c/Users/Dario/Documents/Orga/Intel/2024C1/clase1/holamundo
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Breakpoint 1, main () at holamundo.asm:9
(gdb) n
Breakpoint 1, main () at holamundo.asm:9
(gdb) n
Breakpoint 1, main () at holamundo.asm:11
(gdb)
```


Unidad 2

Arquitectura de Von Newmann:

Es a partir de la misma que se promueve el tratamiento digital de la información dado que hasta 1945 dicho tratamiento era mecánico. Esto lo hace a partir de dos conceptos claves:

El programa almacenado: El mismo computador tiene un programa de instrucciones almacenado en su propia memoria. Las operaciones se ejecutaban al compás de su lectura antes, lo que supone esta arquitectura es que la memoria ahora no solamente es para almacenamiento de datos si no que también aloja las instrucciones del programa.

La ruptura de secuencia: Una toma de decisión involucraba una intervención humana. La ruptura de secuencia permite dotar a la máquina de poder decidir lógicamente en forma automática, a partir de la instrucción de salto condicional.

Un computador de arquitectura Von Neumann se lo puede considerar como un conjunto de unidades conectadas entre si, con una función determinada dentro del esquema. Consta de cinco componentes:

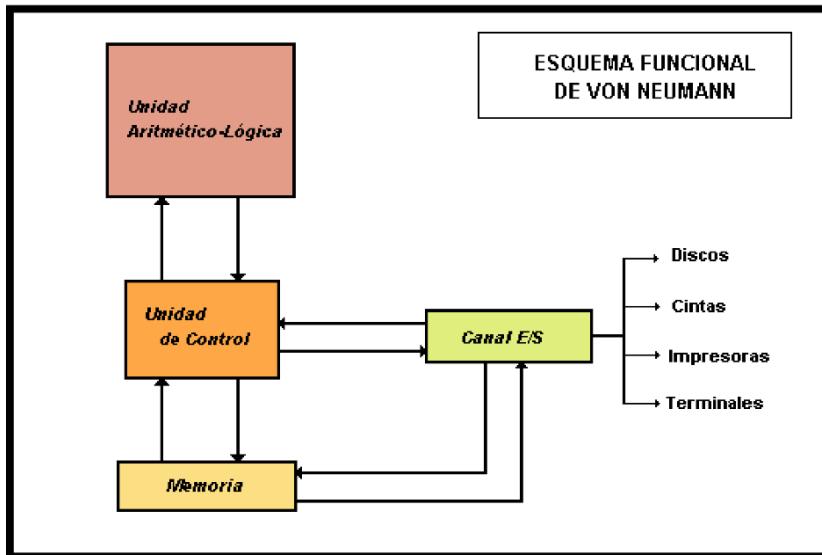
Memoria: Se divide en celdas donde el contenido de cada una es variable. Cada celda posee una identificación un número fijo. La cantidad de celdas disponibles determina la capacidad de una memoria. Aquí se alojan las instrucciones y los datos con los cuales trabaja un programa (llamados operandos).

Unidad aritmético-lógica (UAL): Realiza operaciones de tipo aritméticas (suma y resta) y lógicas (comparaciones).

Unidad de control (UC): Desde acá se controlan y gobiernan todas las operaciones (búsqueda, decodificación y ejecución de instrucciones). Extrae de la memoria central la nueva instrucción a ejecutar, la analiza y extrae los operandos implicados. Habilita el tratamiento de los datos en la **UAL** y de ser necesario los almacena en la memoria central.

Dispositivos de entrada/salida: gestiona la transferencia de información entre unidades periféricas y memoria central, en ambos sentidos. Advierte a la unidad de control cuando todas las informaciones han sido transferidas.

Bus de datos: Proporciona un medio de transporte de datos entre las distintas partes, no almacena solo transmite.



Abacus

Para estudiar esta máquina, es necesario definir ciertos conceptos que se encuentran en la misma:

Registro: Constituye una suerte de memoria de paso que puede almacenar una cierta cantidad de bits.

Compuerta: Circuitos electrónicos biestables unidireccionales, es decir, permiten el pasaje de información en un único sentido y admiten únicamente dos estados: abierto (1) y cerrado (0).

Bus: es posible discernir en tres tipos:

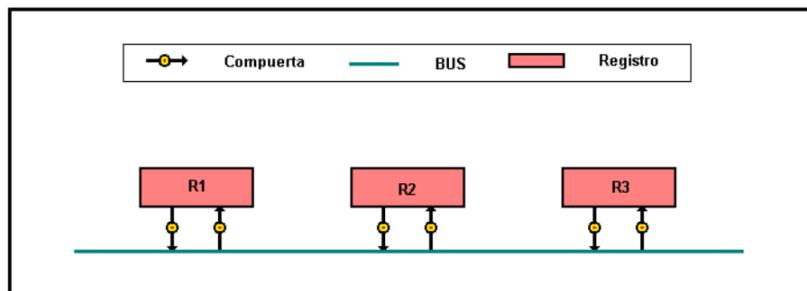
Bus de datos: mueve la información de componentes internos y externos del hardware.

Bus de direcciones: ubica los datos en memoria teniendo relación directa con los procesos del CPU.

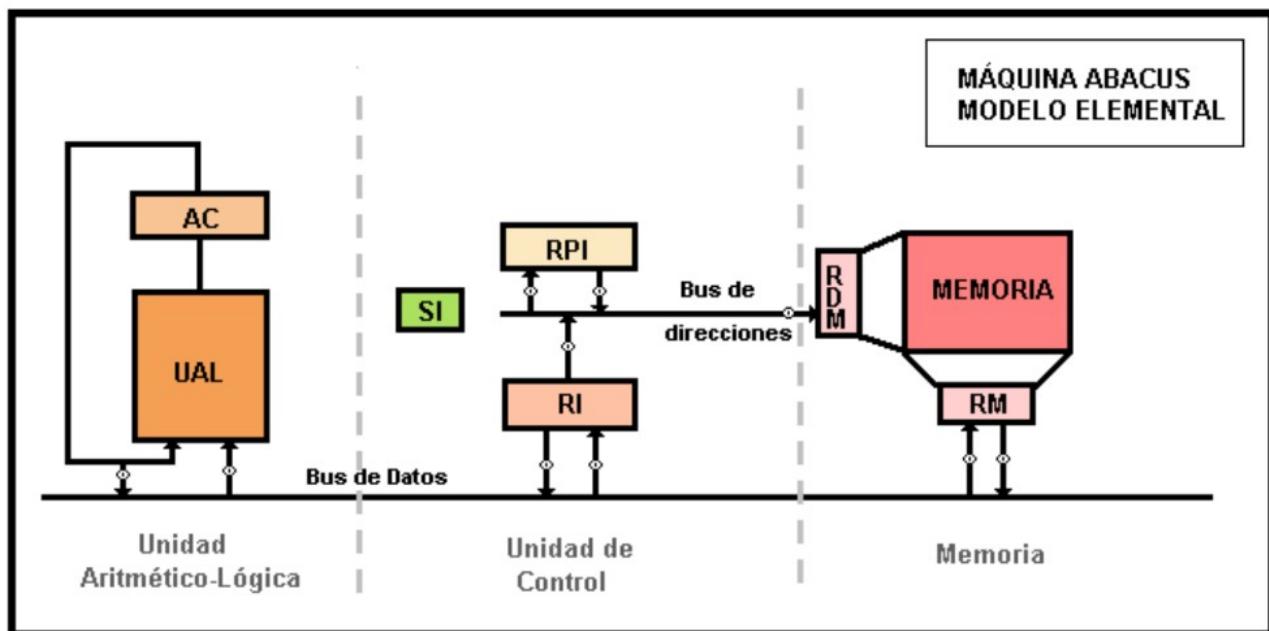
Bus de control: marca estado de una instrucción dada a la PC.

La transferencia de datos puede darse entre componentes de un mismo ordenador o entre varios. Es por esto que existen buses internos y externos.

A partir de compuertas, registros y buses es que se translada la información:



No pueden dos compuertas estar abiertas simultáneamente, tal que la compuerta que manda información al bus es única.



Registros

AC: acumulador

RI: registro de instrucción

SI: secuenciador de instrucciones

RPI: registro de próxima instrucción

RI: registro de instrucción

RDM: registro de direcciones de memoria

RM: registro de memoria

Abacus es una máquina en una sola dirección, la **UAL** (unidad aritmético lógica) posee un registro particular llamado acumulador (**AC**). Este suele albergar primer operando o bien un resultado. Las instrucciones entonces se ejecutan en una sola dirección, la del segundo operando.

Operaciones que realiza la UAL:

- CARGA (cargar un contenido en el AC)
- ALMACENAMIENTO (enviar datos a memoria por el bus)
- SUMA (sumar un dato a lo que haya en el AC)
- LOGICAS (OR, AND, XOR)

Es importante recordar que en una maquina Abacus **todas las operaciones** se realizan “contra” el acumulador y el resultado siempre queda almacenado en él.

En la **UC** (unidad de control) extraemos y analizamos instrucciones de la memoria central, a partir de dos registros:

RPI: Contiene la **dirección** de la próxima instrucción a ejecutar, se comunica con la memoria y el RI a partir del bus de direcciones. A medida que se ejecutan las direcciones, este registro aumenta en una unidad, al menos que se produzca una ruptura de secuencia.

RI: Contiene la **instrucción** extraída de la memoria y se guarda en dos partes: El código de operación y la dirección del operando. Se comunica con la memoria a partir del bus de datos.

El **SI** (secuenciador de instrucciones) se encuentra interconectado con cada componente y es el encargado de administrar la apertura y cierre de compuertas junto al correcto pasaje de la información.

La **Memoria** intercambia información con el resto del computador, ya sea para acceder a los datos o almacenar información dentro a partir de dos registros:

RDM: Contiene la dirección de la celdad de memoria en la cual se escribirá o leerá la información.

RM: Contiene el dato que debe ser leído o escrito en la memoria.

Registro de instrucción en Abacus

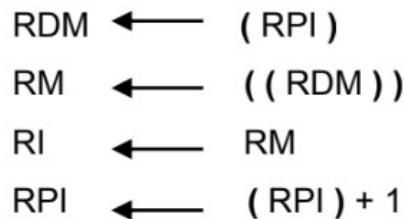
Como vimos, las instrucciones son unidireccionales y constan de dos partes: la operación y el operando. Es decir, lo que queremos hacer con un dato y el dato en sí. Para realizar una suma por ejemplo hay que cargar un operando contra el acumulador, luego sumar el segundo operando para finalmente almacenar en una celda de memoria el resultado.

CARGAR	200	Se suman los contenidos de las celdas de memoria cuyas direcciones son 200 y 206 El resultado se almacena en la celda de memoria de dirección 200
SUMAR	206	
ALMACENAR	200	

Desarrollo de una instrucción

Las instrucciones en Abacus se descomponen en dos fases

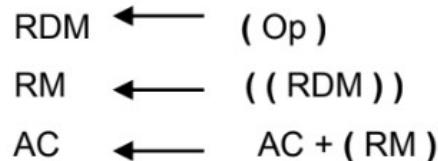
Fase de búsqueda: La instrucción inicialmente se encuentra cargada en memoria y la búsqueda de la misma inicia a partir de tener en el RPI la dirección de dicha instrucción. Del RPI se transfiere el contenido al RDM, quien realiza una orden de lectura contra la memoria del contenido alojado en dicha dirección. Una vez que obtengo el contenido, este pasa al RM para luego a partir de una orden la UC se transfiere el contenido al RI. Una vez que los circuitos especializados analizan el código de la operación de la instrucción, se vuelve al RPI y se ejecuta la siguiente, para repetir el ciclo:



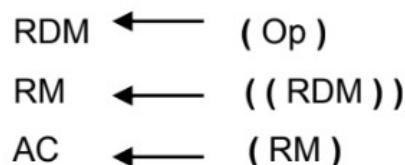
Para buscar un operando, una vez que la UC analiza el código de operación, el RI manda al RDM la orden de operación de lectura. El RDM busca dicha operación y la aloja en el RM.

Ejecución: Cada ejecución implica movimiento de datos de forma secuencial. Cada fase depende del tipo de tarea que se quiera realizar:

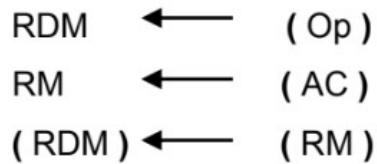
- **Suma:** se debe sumar el contenido del RM al contenido del acumulador



- **Carga:** se debe almacenar en el acumulador un dato contenido en memoria



- **Almacenamiento:** se debe “guardar” en memoria el contenido del acumulador



- **Bifurcación:** se debe “saltar” a la dirección indicada en la instrucción. La dirección de bifurcación debe ser transferida al RPI (para buscar la próxima instrucción a ejecutar).



El código de instrucción se guarda en formato hexadecimal. Las celdas en abacus son de 16 bitsm donde las direcciones ocupan 12 bits y el contenido 4bits.

Sobreescritura de memoria

Así como guardo valores directamente en las celdas, puedo guardar direcciones de otro lugar de memoria, es decir tengo puntero hacia otro lugar. Para poder acceder al valor tengo que pensar en escribir una celda con la instrucción pero en tiempo de ejecución, no en tiempo de codificación.

El contenido en sí, la dirección se encuentra escrita en 16 bits, con los primeros 4 bits en 0 ante la ausencia de una operación. Es decir, solamente tendría la dirección cargada como contenido. Tendría que poder modificar mi primer nibble con el operador que desee. Para ello, debería tener una celda donde guarde previamente, la instrucción pero en ausencia de una dirección (auxiliar de carga o de otra instrucción). De esta manera al sumar ambos números hexadecimales, podría obtener tanto mi operación como mi operando. Una vez que ya tengo la suma en el acumulador, lo guardo en una celda de memoria, dentro de la secuencia de ejecución.

ro free* | Abacus y Abacus Avanzado 6/9/22 |

Ejercicio 2: Sumar A+B.
La dirección de A está almacenada en 200
La dirección de B está almacenada en 201
Dejar el resultado de A+B en celda 202
Punto de carga: celda 300

		Contenido	Comentarios	Ayuda
200	050A			
201	050B			
202				
2FE	3000			
2FF	1000			
300	1200	(AC)=050A		
301	32FF	(AC)=150A		
302	2306			
303	1201	(AC)=050B		
304	32FE	(AC)=350B		
305	2307			
306	CCCC	(AC)=<A>		
307	AAAA	(AC)=<A>+		
308	2202			
309	F000			
30A				

La dirección de A está almacenada en 200.
La dirección de B está almacenada en 201.
Dejar el resultado de A+B en la celda cuya dirección se encuentra almacenada en 202.
Punto de carga: celda 300.



		Contenido	Comentarios	Ayuda
200	050A			
201	050B			
202	050C			
2FE	3000			
2FF	1000			
300	1200	(AC)=050A		
301	32FF	(AC)=150A		
302	230A			
303	1201	(AC)=050B		
304	32FE	(AC)=350B		
305	230B			
306	1202	(AC)=050C		
307	32FF	(AC)=150C		
308	32FF	(AC)=250C		
309	230C			
30A	1111	(AC)=<A>		
30B	3333	(AC)=<A>+		
30C	2222	(50C)=<A>+		
30D	F000			

Vectores

Un vector es una serie de elementos que se almacenan en memoria. Cuantas celdas de memoria ocupa depende de la lógica de nuestro programa. El vector trivial es que cada elemento ocupe una celda, pero no siempre es así un elemento podría ocupar más de una celda.

Dado un vector de números positivos, sumar sus elementos.

En la celda 200 se encuentra la dirección de inicio del vector.

En la celda 201 dejar el resultado de la sumatoria.

El fin del vector está indicado con el elemento -1.

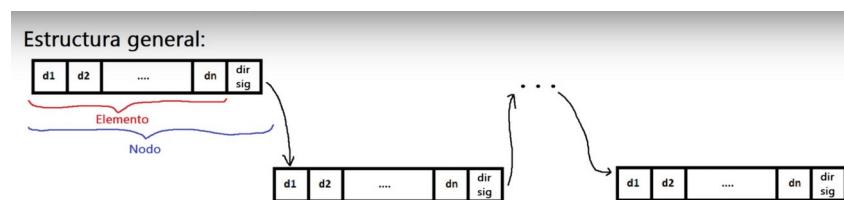
Punto de carga: celda 300.



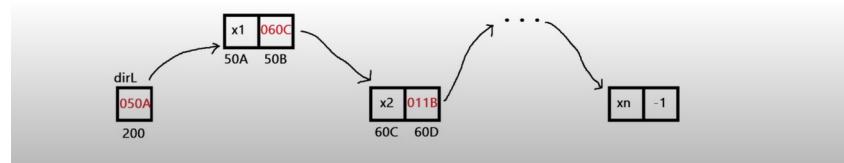
202		
2FD		
2FE	0001	
2FF	1000	
300	0000	(AC)=0000
301	2201	Inicializo sumatoria
302	1200	(AC)=050A/050B
303	32FF	(AC)=150A/150B
304	2305	
305	CCCC	(AC)=x1/x2
306	0000	
307	3201	(AC)=Sumat + x1
308	2201	
309	1200	(AC)=050A
30A	32FE	(AC)=050B
30B	2200	(200)=050B
30C	0000	
30D	7302	
30E	F000	
30F		

Listas

Las listas no necesariamente están en secuencia todos los elementos. Cada elemento es un nodo, el cual se define como una cantidad de celdas continuas con datos:



Ejemplo concreto:



La última celda del nodo suele tener la dirección del nodo siguiente.

Ejercicio 4: Dada una lista (L) de nros positivos, sumar sus elementos.

En la celda 200 se encuentra la dir de inicio de la lista

En la celda 201 dejar el resultado de la sumatoria

El fin de L esta indicado con el valor -1 en el último nodo en la celda que apunta al siguiente

Punto de carga: celda 300

200		
201		
202		
...		
2FD		
2FE	0001	
2FF	1000	
300	0000	
301	2201	
302	1200	(AC)=050A/060C
303	8311	
304	32FF	(AC)=150A/160C
305	2306	
306	CCCC	(AC)=X1
307	3201	(AC)=sumat + X1
308	2201	
309	1200	(AC)=050A
30A	32FE	(AC)=050B
30B	32FF	(AC)=150B
30C	230D	
30D	CCCC	(AC)=060C
30E	2200	
30F	0000	
310	7302	
311	F000	
312		

Ejercicio de parcial

Se tiene una lista (L) cuya dirección de inicio se encuentra almacenada en la celda 200(16). Esta lista representa los cursos de un instituto. Cada nodo de la lista (L) está formado por 3 celdas contiguas en memoria:

- * La primera contiene un número de curso.
- * La segunda contiene la cantidad de inscriptos al curso.
- * La última contiene la dirección del siguiente nodo de la lista. El final de la lista (L) se indica con un valor -1 en la última celda del último nodo.

Por otro lado se tiene un vector (V) que comienza en la celda 100(16) que contiene las novedades semanales de bajas del curso. El vector tiene dos celdas contiguas en memoria:

- * La primera celda contiene un número de curso.
- * La segunda celda contiene la cantidad de bajas del curso.

El final del vector se representa con un -1 en la primera celda.

Se pide realizar un programa ABACUS con punto de carga en la celda 300(16) que recorra el vector (V) y actualice la cantidad de inscriptos al curso, teniendo en cuenta que tanto la lista (L) como el vector (V) están ordenados por número de curso y que no hay ningún curso en el vector (V) que no esté en la lista.

Para que un número sea negativo hay que hacerle not y sumarle 1.

Superabacus

U2 – Máquina Elemental (SuperAbacus)

● Características principales:

- Registros generales (datos o direcciones)
- No posee RPI. R0 contiene dirección de próxima instrucción. Incremento vía sumador.
- Máquina de dos direcciones (dos operandos)
- UAL permite hacer cálculos con direcciones y datos

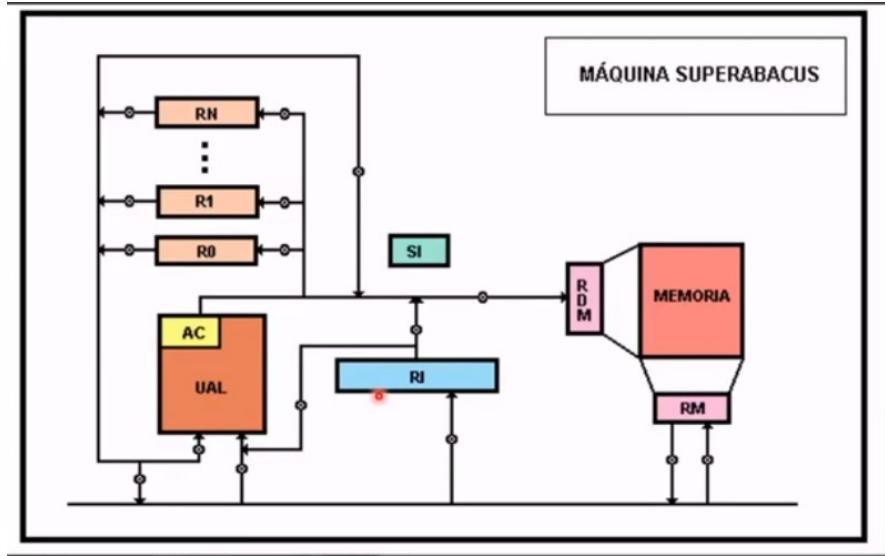
Los registros generales en superabacus tienen la particularidad de que además de ser portadores de información de transito, también es posible acceder como programadores a esta información. Es decir, no necesariamente hay que hacerles una carga. Se puede operar sobre los mismos registros.

Serán útiles para alojar punteros hacia algún lugar en memoria.

No existe un RPI en específico, pero aún así los registros tienen esta capacidad. Los registros están enumerados y el 0 se utiliza para esto (la arquitectura ARM es muy cercana a esto). El circuito de suma de la maquina se encarga de cambiar a la próxima instrucción, es decir no se cambia a la siguiente instrucción.

Voy a tener la capacidad de una misma instrucción de maquina cargar dos operandos a la vez. Ya no hay que hacer pivoteos en el acumulador.

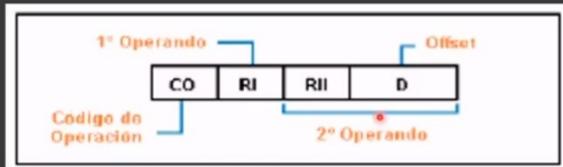
La UAL ahora no solamente puede hacer cálculos con datos, si no también con direcciones.



Los registros se encuentran físicamente dentro de la UAL. Por dentro estos cambian como se componen.

U2 – Máquina Elemental (SuperAbacus)

- Formato de instrucciones:
 - Código de operación
 - Primer operando (R I)
 - Segundo operando (R II – D)

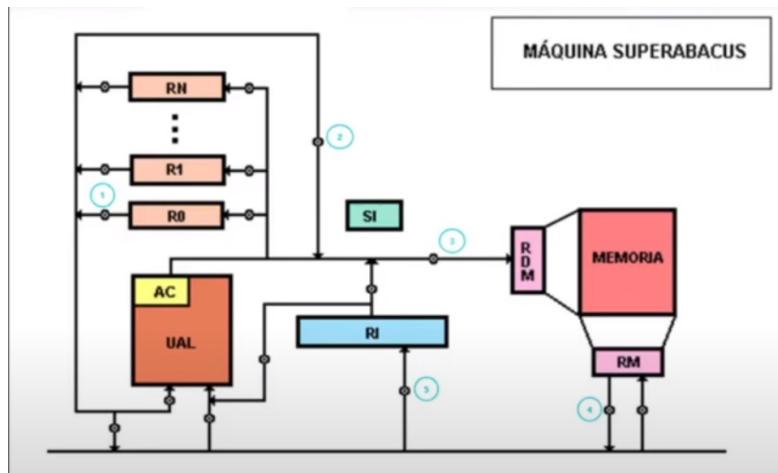


El código de operación se mantiene, no se define cual es el tamaño que alojará (en abacus eran 4 bits). Tengo un campo para el primer operando, el cual aloja el número de registro al que va a acceder. El segundo campo le da más versatilidad gracias al offset.

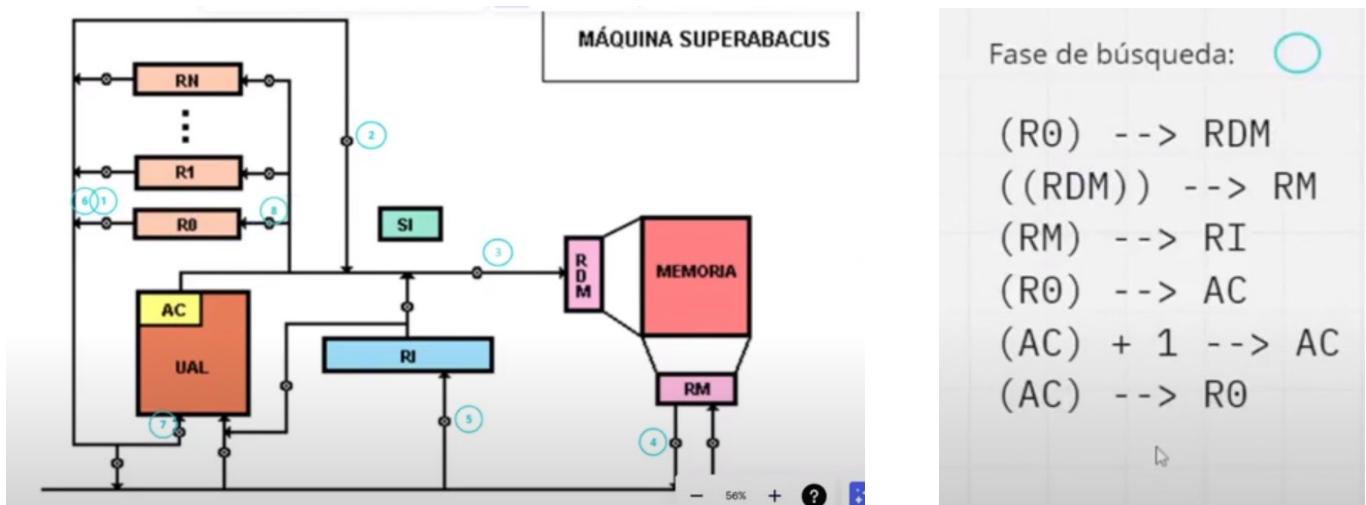
El acumulador cumple la función de alojar momentáneamente el resultado de haber procesado algún dato en la UAL, para luego alojarlo en algún registro.

Fase de búsqueda en Superabacus

Tal que el R0 cumple la función de registro de la próxima instrucción, la fase de búsqueda inicia aquí. Esta celda tiene en que dirección de memoria se encuentra la instrucción de máquina que quiero que ejecute el computador. Por ende del R0, voy al RDM. En el RDM se hace la operación de lectura y la instrucción va al RI.

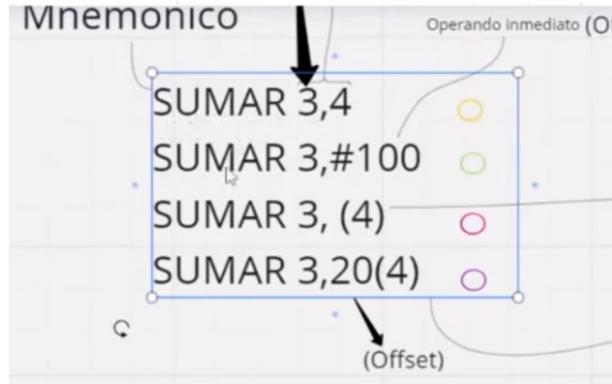


Tras haber llegado a esta fase para ir a la próxima instrucción, al R0 se le suma 1. Para ello, este valor viaja a la UAL, vuelve al acumulador y de ahí vuelve al R0.



Operación de suma en superabacus

Se le puede especificar dos operandos, pero el segundo operando puede tener distinto origen. Podes ir a buscarlo a la memoria de la máquina, como algo dentro de la propia instrucción como un operando inmediato, etc.

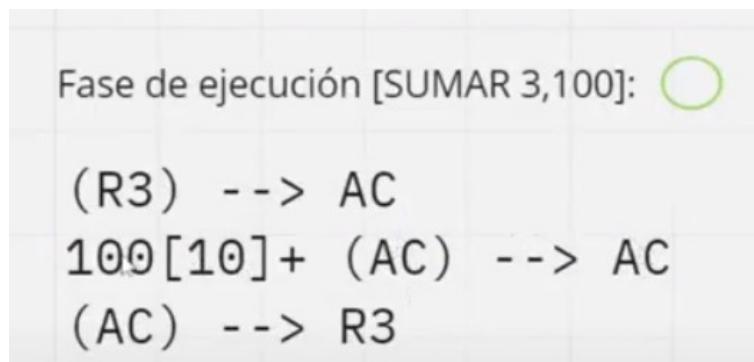


El mnemónico es una abreviación útil para el programador para saber que está haciendo. El catálogo mas usual de instrucciones son de entre 30 y 50 instrucciones de máquina. Como es tedioso acordarse 50 códigos, esto es más fácil.

La primer suma va y busca en el registro general de la máquina, el registro 3 y 4 y los suma. El resultado suele terminar en el primer registro.

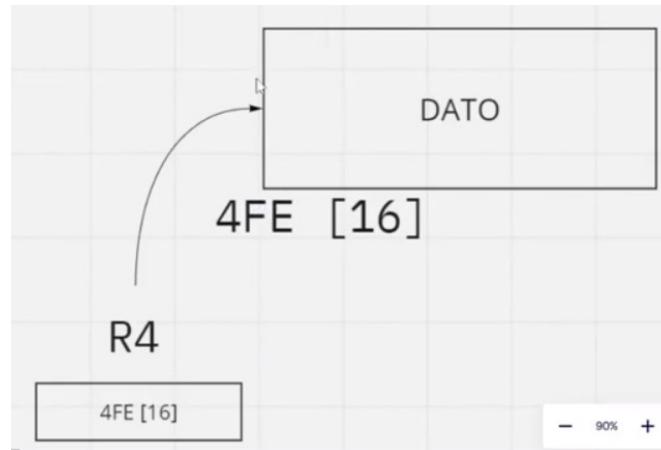
La suma en fase de ejecución para este primer ejemplo, copia el contenido del registro 3 al acumulador. Luego lleva al acumulador el registro 4 y lo suma. Luego se vuelve a llevar al registro 3.

En el segundo ejemplo, voy a poner un **operando inmediato (#)**. El número se escribe en base 10, en principio. Se acumula el registro 3 pasando por la UAL. Por la compuerta que conecta el RI con la UAL, se pasa el operando inmediato.



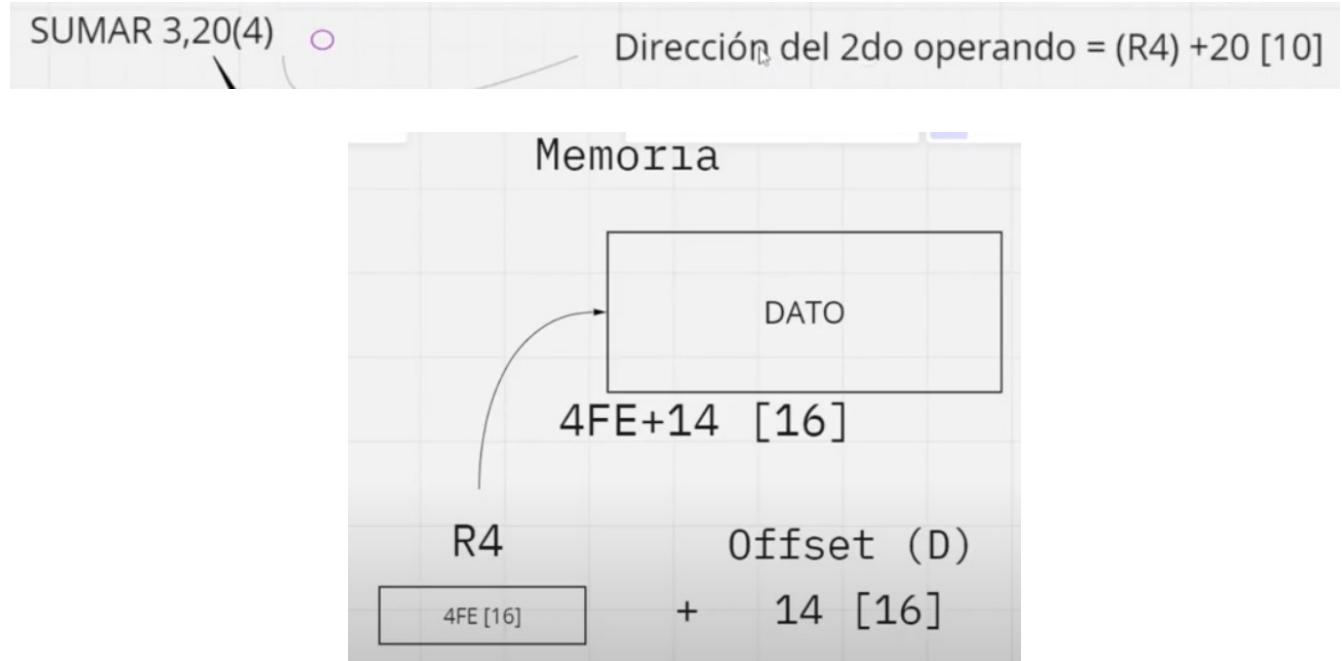
En el tercer ejemplo, se propone un acceso a la memoria de la máquina, asumiendo que en el registro se

guarda la dirección en memoria de dicho valor, es decir, el registro es un puntero a esta dirección de memoria.



Por ende, para hacer esta suma primero se guarda el valor del registro 3 en el AC, luego el registro 4 manda la dirección al RDM, para que se ejecute la operación de lectura en la memoria y baje por el RDM hacia el AC y se sume. Tras esto, la suma se almacena en el registro 3.

Finalmente, el último ejemplo muestra como es posible combinar datos en el 2do operando, a partir de un número pasado de forma inmediata y un número alojado en memoria cuya dirección se encuentra apuntada en un registro. Lo que se pasa de forma inmediata es cuanto me tengo que transladar para acceder en la memoria



En este caso, no se arranca la ejecución por el registro 3, si no por calcular la dirección del segundo operando. Entonces sumo el contenido del registro 4 y el offset, abriendo compuertas correspondientes.

Tras tener la dirección de memoria en el AC, lo mando al RDM, busco dato, sale por el RM de vuelta a la UAL. Finalmente, cargo el contenido del registro 3 a la UAL, se le suma a lo del acumulador y finalmente vuelve al registro 3.

Fase de ejecución [SUMAR 3,20(4)]: ○

(R4) --> AC
20[10] + (AC) --> AC
(AC) --> RDM
((RDM)) --> RM
(RM) --> AC
(R3) + (AC) --> AC
(AC) --> R3

ARM

ARM surge por la necesidad de tener procesadores de baja potencia y poder programar lenguajes de ensamblado para los mismos (internet de las cosas, heladeras, celulares, etc).

Características principales:

- Posee una **arquitectura Load/Store**, lo que impacta en como se procesan las instrucciones. La forma de comunicarse con la memoria se basan solamente en estas instrucciones.
- Todas las **instrucciones son de 32 bits**. En Intel son de longitud variable
- Todas las instrucciones poseen 3 direcciones: 2 registros de operandos y 1 registro de resultado. A diferencia de intel que el resultado suele quedar en el primer operando.
- ARM tiene una ejecución condicional de TODAS las instrucciones. Todas las instrucciones se ejecutan si se cumplen ciertas condiciones. Lo que hace ARM no se basa en bifurcaciones, si no que se tienen ciertas funciones que se ejecutan en base a un resultado previo.
- Existen instrucciones de Load Store de registros Múltiples. Puedo cargar información de la memoria a varios registros sin necesidad de ejecutar varias instrucciones.

Organización de registros

- R0 a R12 son registros de propósito general (32 bits)
 - Usados por el programador para (casi) cualquier propósito sin restricción
- R13 es el *Stack Pointer* (SP)
- R14 es el *Link Register* (LR)
- R15 es el *Program Counter* (PC)
- El *Current Program Status Register* (CPSR) contiene indicadores condicionales y otros bits de estado

Todos los registros son de 32 bits.

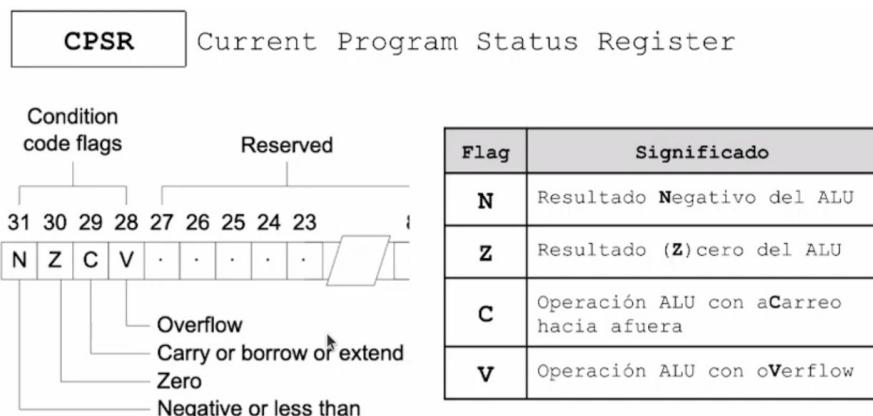
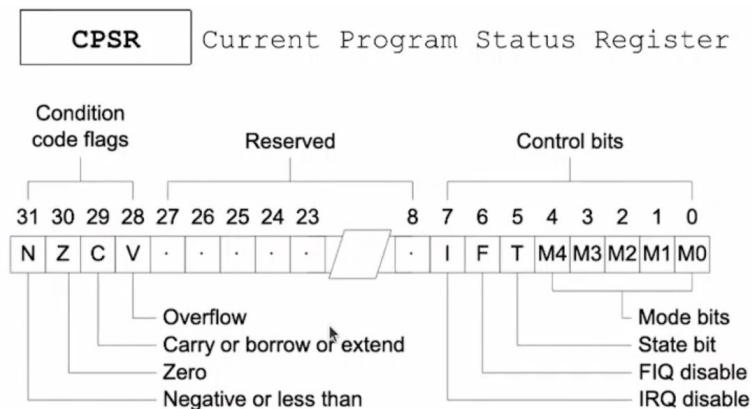
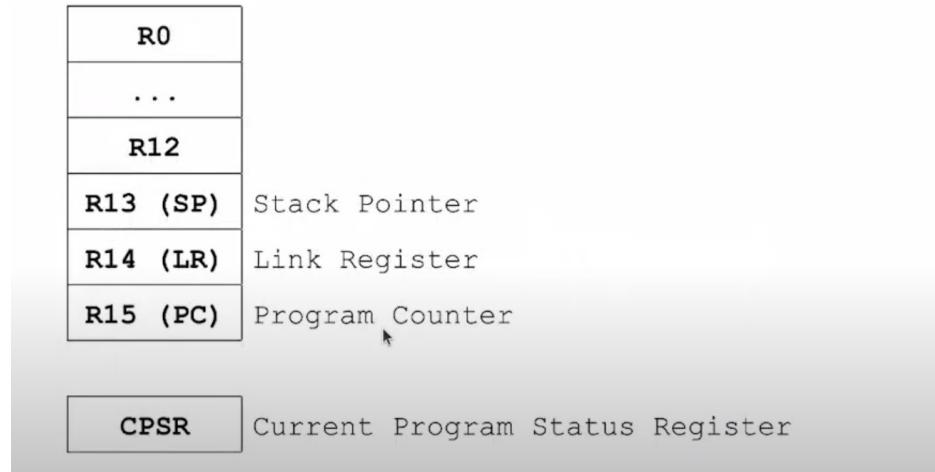
Si bien los generales los puedo usar para todo, pero hay que tener en cuenta que algunos se utilizan para interrupciones de software para resolver cosas con el sistema operativo.

R13 y SP es lo mismo, apunta a la primera posición de la pila.

Con el R14 o LR puedo volver a un flujo principal, con esto puedo crear rutinas internas.

R15 es como el RPI en abacus, tiene la dirección de la próxima dirección a ser ejecutada.

El CPSR tiene los flags de estado.



Flag	Significado
N	Resultado Negativo del ALU
Z	Resultado (Z)ero del ALU
C	Operación ALU con aCarreo hacia afuera
V	Operación ALU con oVerflow

*ALU: Arithmetic Logic Unit

Esto se accede indirectamente cuando quiero ver algún resultado lógico o aritmético.

Organización de la memoria

- Máximo: 2^{32} bytes de memoria
- *Word* = 32-bits
- *Half-word* = 16 bits
- *Words* están alineadas en posiciones divisibles por 4
- *Half-words* están alineadas en posiciones pares

Cuando se guarda algo del tamaño de una Word, las posiciones dentro de esa WORD son divisiones de 4 bits. En una Half-words se alinean las posiciones de a pares (8 bits). Si yo defino una WORD en una posición no divisible por cuatro me quedan 2 bytes libres (???).

Estructura de un Programa

- La forma general de una línea en un módulo ARM es:
`label <espacio> opcode <espacio> operandos <espacio> @ comentario`
- Cada campo debe estar separado por uno o más espacios.
- Las instrucciones no empiezan en la primer columna, dado que deben estar precedidas por un espacio en blanco, incluso aunque no haya label.
- ARM acepta líneas en blanco para mejorar la claridad del código.

el opcode no puede ser lo primero de la instrucción, como mínimo tiene que haber un espacio antes.

```

.text
@ Indica que los siguientes
@ items en memoria son
@ instrucciones

start:
    mov    r0, #15          @ Seteo de parámetros
    mov    r1, #20
    bl     func             @ Llamado a subrutina
    swi   0x11              @ Fin de programa
func:
    add    r0, r0, r1      @ r0 = r0 + r1
    mov    pc, lr           @ Retornar desde subrutina
    .end                  @ Marcar fin de archivo

```

Con # agarro operandos inmediatos numéricos. No necesariamente es inmediato, depende de la instrucción. Puedo querer guardarlos en memoria y referenciarlos también.

Para llamar a una subrutina uso **bl func (branch with link)**. La diferencia entre branch y branch with link es que puedo volver a la parte desde donde se llamó.

Los operandos se ordenan así:

```

func:
    add    r0, r0, r1      @ Subrutina
    mov    pc, lr           @ r0 = r0 + r1
                                @ Retornar desde

```

Para volver a la subrutina, al program counter tengo que decirle que copie el valor del link register, que cuando hice bl, me guarde la instrucción que le sigue a la bifurcación para volver.

```

add    r0, r0, r1      @ Bifurcación
mov    pc, lr           @ Retornar desde subrutina

```

Interrupciones de Software

Mediante los mismos le solicitamos al sistema operativo que vaya a buscar información y que nos devuelva un resultado. Es una manera de decirle al sistema operativo que tome el control.

Por ejemplo para imprimir algo:

ARM cuenta con una instrucción *swi* que provoca una interrupción de software

Para que el SO sepa que es lo que quiero que haga se le pasa un entero que significa que es lo que quiero que haga.

Tenemos la atención del SO pero ¿cómo sabe lo que queremos?

Operando swi: recibe un entero que dice qué hacer

El SO también puede leer los registros para obtener información adicional si es necesario



- "Entero que dice qué hacer": por ejemplo, "5" significa "imprimir algo"
- Con la lectura de los registros, estos podrían incluir exactamente qué imprimir

Ejemplo:

Imprimiendo un entero

El operando que indica “imprimir un entero” es 0x6B

El registro r1 contiene el entero a imprimir

El registro r0 contiene dónde imprimirlo

l significa “imprimir en stdout (pantalla)”

```
mov  r0, #5
mov  r1, #7
add  r2, r0, r1
mov  r1, r2      ; r1: entero a imprimir
mov  r0, #1      ; r0: dónde imprimir
swi  0x6B      ; 0x6B: imprimir entero
```

Finalizar el programa

Necesitamos indicar el fin del programa

¿Cómo puede hacerse?

swi con un operando particular (0x11)

```
mov r0, #5
mov r1, #7
add r2, r0, r1
mov r1, r2      ; r1: entero a imprimir
mov r0, #1      ; r0: dónde imprimir
swi 0x6B        ; 0x6B: imprimir entero
swi 0x11        ; 0x11: salir del programa
```

Secciones del programa

Entendiendo al todo como un gran conjunto de bits, las secciones le dicen al ensamblador que bits deben colocarse en qué parte de la memoria. Por ejemplo la directiva **.text** indica que lo de abajo que tenemos es código. La directiva **.data** especifica la sección de variables:

```
.data
string1:
    .asciz "hola"
string2:
    .asciz "chau"
```

Podes escribir **.ascii** o **.asciz**, la diferencia que hay entre ambos es que **.asciz** te define un cero al final para marcar el fin de string.

ARMSim#

Estructura básica de un programa

```
.data
cadena:
    .asciz "linea"
entero:
    .word 78
.text
.global _start
_start:
    @ Comentario
    swi 0x11
.end
```

La razón de la identación rara es porque las directivas (lo que señalan las secciones del programa) no pueden arrancar con identación 0, si no que tienen que tener un espacio antes. En cambio, los label si que tienen que arrancar con espacio 0 para que sean tomados como tal. Por eso parece que .text y .global _start están definidos como parte de entero, pero en realidad dan arranque a una sección distinta.

Imprimir cadena de caractéres

ARMSim#

Salida standard de cadena de caracteres

```
.data
cadena:
    .asciz "Soy una cadena"
.text
.global _start
_start:
    ldr r0, =cadena
    swi 0x02
```

Contexto en el que surge ARM

ARM surge ante la necesidad de un procesador para los asistentes digitales apple. Los procesadores hasta ese entonces en el mercado se caracterizaban por tener como prioridad más que nada el desempeño y este se caracterizó por tener en cuenta también el consumo como un factor primordial, dado que se requería para dispositivos portátiles de baterías con tecnología no tan avanzada como al día de hoy.

El modelo de negocio de ARM era a partir de licencias, en vez de lo que comúnmente se hacía que cada empresa creaba su propio procesador. A partir de esto las empresas compran su licencia y cada una a partir de conocer el código fuente fue integrando distintos periféricos a partir del mismo procesador.

Otra característica distintiva de ARM para la época fue desligar el hardware del software. Antes el software de cada procesador era exclusivo para el mismo. ARM en cambio crea el ISA (Instructions Set Architecture). Esto refiere a la generalización del set de instrucciones para el desarrollo del software, para que sea independiente el hecho de que el procesador sea orientado al desempeño o al consumo.

Stack + Subrutinas

Al igual que en Intel, para llamar a subrutinas tengo que reservar espacio en el Stack tal que se alteran las instrucciones ya dadas de por sí por el sistema operativo. Para interactuar con el Stack (tal como se hace en Intel con push y pop) se puede usar **LDM (Load Multiple)** y **STM (Store Multiple)**. El propósito de estas instrucciones es la de cargar múltiples direcciones en memoria en registros y viceversa

Load Multiple

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

Store Multiple

```
STM{addr_mode}{cond} Rn{!}, reglist{^}
```

Es posible determinar el sentido en el cual se hace con dos letras más reservadas para ello:

Los modos de instrucciones de STM y LDM tienen alias para acceder a la pila:

- FD = Full Descending
 - STMFD/LDMFD = STMDB/LDMIA
- ED = Empty Descending
 - STMED/LDMED = STMDA/LDMIB
- FA = Full Ascending
 - STMFA/LDMFA = STMIB/LDMDA
- EA = Empty Ascending
 - STMEA/LDMEA = STMIA/LDMDB

Por lo general se usa el Full descending:

Los modos de instrucciones de STM y LDM tienen alias para acceder a la pila:

- FD = Full Descending
 - STMFD/LDMFD = STMDB/LDMIA
- ED = Empty Descending
 - STMED/LDMED = STMDA/LDMIB
- FA = Full Ascending
 - STMFA/LDMFA = STMIB/LDMDA
- EA = Empty Ascending
 - STMEA/LDMEA = STMIA/LDMDB

Se suele utilizar el modo Full Descending

Acá para llamar a una subrutina usas exclusivamente bl (branch linker). La razón por la cual solamente hay una única instrucción y no con condicionales como en Intel es porque ya de por si las instrucciones de ARM son de ejecución condicional.

Por ejemplo, en esta secuencia de instrucciones, añado al stack pointer el r0 y el lr a partir de store multiple. Luego cargo r0 con el valor 1 (imprimir por stdout), llamo a la subrutina de impresión (que se encargará de mostrar el R1 por consola). Se imprime un carácter de fin de linea como una cadena de caractéres (cuando se muestra el EOL por cónsola).

Llamado a subrutina

```
    bl print_r1_int  
    ...
```

Subrutina

```
print_r1_int:  
    stmfd sp!, {r0,lr} @ Stack R0  
    ldr r0, #0x1  
    swi SWI_Print_Int @ Mostrar entero en R1 por consola  
    ldr r1, =EOL  
    swi SWI_Print_Str @ Mostrar EOL por consola  
    ldmfd sp!, {r0,pc} @ Unstack r0
```

Instrucciones aritméticas

Add

ADD{cond}{S} Rd, Rn, <Oprnd2>

Subtract

SUB{cond}{S} Rd, Rn, <Oprnd2>

Reverse subtract

RSB{cond}{S} Rd, Rn, <Oprnd2>

Multiply

MUL{cond}{S} Rd, Rm, Rs

La condición puede estar como no, si no se pone nada se ejecuta siempre

Instrucciones lógicas

And

AND{cond} {S} Rd, Rn, <Oprnd2>

Exclusive Or

EOR{cond} {S} Rd, Rn, <Oprnd2>

Or

ORR{cond} {S} Rd, Rn, <Oprnd2>

Desplazamientos

El desplazamiento lógico hace una rotación entre los operandos y el desplazamiento aritmético se hace cuando se completa con 1 o 0 para conservar el signo

Se puede rotar una cantidad específica en otro registro con **Barrel Shifter**:

Cuando se especifica que el segundo operando es un registro *shifteado*, la operación del Barrel Shifter es controlada por el campo Shift en la instrucción.

Este campo indica el tipo de *shift* a realizar.

La cantidad de bits a *shiftear* puede estar contenida en un campo inmediato o en el byte inferior de otro registro (que no sea el R15)

Cuando hago un AND entre dos números, comparo bit a bit:

@ r0 = 5 = 0101
@ r1 = 2 = 0010
@ r5 = 000 0 ^I

Archivos

Definiciones

```
.data
filename:
    .asciz "archivo.txt"
    .align
InFileHandle:
    .word 0
```

Apertura

```
ldr r0,=filename          @ (R0) -> nombre de archivo
mov r1,#0                 @ (R1) -> modo: entrada
swi 0x66
bcs InFileError
ldr r1,=InFileHandle     @ (R1) -> InFileHandle
str r0,[r1]               @ almacena handler
```

en r1 pones el modo. 0 = lectura y 1 = escritura

Ler entero desde archivo

```
ldr r0,=InFileHandle     @ (R0) -> InFileHandle
ldr r0,[r0]               @ (R0) = InFileHandle
swi 0x6CI                @ (R0) = Entero leido
```

Cerrar archivo

```
ldr r0,=InFileHandle    @ (R0) -> InFileHandle  
ldr r0,[r0]             @ (R0) = InFileHandle  
swi 0x68
```

Salida standard de entero

```
mov r0,#1 @ (R0) = Stdout (salida por pantalla)  
mov r1,r2 @ (R1) = entero a mostrar  
swi 0x6B
```

NOTA:

MOV es para mover datos entre registros.

LDR es para traer algo de la memoria a un registro (Load register)

STR es para guardar algo de un registro en la memoria (Store register)

Compare

La instrucción compare funciona a partir de la resta de dos operandos, descartando el resultado. En este proceso se setean los flags, cuyos bits de estado me reflejan el resultado de la comparación:

Los bits de estado dicen algo sobre el resultado de las comparaciones aritméticas

Los operandos son iguales	mov r0, #5 cmp r0, #5	Setea bit/flag cero (resultado es cero)
1º operando < 2º operando	mov r0, #5 cmp r0, #20	Setea bit/flag negativo (resultado es negativo)

Ejecución condicional

A cada instrucción podes añadirle dos letras para que se ejecuten condicionalmente por un bit de estado:

movmi r0, #42 mover si el bit negativo está encendido

mi = minus. Esto sucederá por ejemplo si cmp activo el bit de negativos. Por ende tiene que haber una ejecución previa que justamente lo haga.

movpl r0, #42 mover si el bit negativo **no** está encendido

Pl = plus

moveq r0, #42 mover si el bit cero está encendido

eq = equal

```
movne r0, #42      mover si el bit cero no está encendido
```

ne = not equal

Instrucción Branch

Sirve para saltar a una etiqueta en específico. Si quiero guardar la siguiente instrucción en el link register para volver hacia donde estaba, uso bl. En este caso mov r0, #5 no se ejecuta.

```
    mov r0, #1
    b otra
    mov r0, #5
otra:
    mov r1, r0
```

Branch condicional

Tras un cmp puedo usar los postfijos eq, min, pl, ne , etc para hacer un salto condicional:

```
    mov r0, #0
    mov r1, #5
    cmp r1, #5
    beq otrolado
    mov r0, #25
otrolado:
    mov r2, r0
```

En este caso tras hacer el cmp el flag de 0 esta en 1 y salta para el otro lado, haciendo que mov r0, #25 no se ejecute.

Loops

Se utiliza la instrucción Branch con saltos al inicio o al final del loop.

Operaciones en memoria

La arquitectura ARM sabemos que se basa en una lógica Load/Store, en el sentido de que no se interactua directamente con la memoria cuando se quiere operar con algún dato guardado allí, si no que se translada dicha información a un registro y es a partir de ahí que se interactua con el mismo.

Esta característica de la arquitectura permite un mejor desempeño del hardware, permitiendo que una parte del procesador se encargue de dicha instrucción de Load mientras otra está ejecutando la instrucción actual.

De por sí sabemos que con un **ldr** es posible cargar una dirección en memoria a un registro:

```
.data
mensaje:
    .asciz "hola_mundo"
entero1:
    .word 42
entero2:
    .word *38
```

Teniendo en cuenta que estamos trabajando en un procesador de 32 bits, pudo definir un entero de dicho tamaño bajo un rotulo:

```
.data
mensaje:
    .asciz "hola_mundo"

.text
ldr r0, =mensaje
```

Leer enteros desde memoria

En principio lo que hago es cargar la dirección en memoria de dicho entero a un registro con ldr:

```
.data  
entero1:  
    .word 42  
entero2:  
    .word 38  
.text  
    ldr r0, =entero1
```

Una vez que hago esto, en otro registro me cargo el contenido a partir de corchetes:

```
ldr r0, =entero1  
ldr r1, [r0]
```

Almacenar enteros en memoria

En este caso tengo que tener en un registro cargado la dirección a donde voy a querer almacenar mi entero. A partir de tener dicho entero en otro registro, hago **str (store)** al “contenido” del registro original donde tengo donde lo quiero guardar:

```
.data  
entero1:  
    .word 42  
entero2:  
    .word 38  
.text  
    ldr r0, =entero1  
    mov r1, #57  
    str r1, [r0]
```

Arrays

De la misma manera en la que yo me guardaba enteros bajo un rotulo, puedo guardarme una secuencia de numeros bajo dicho rotulo:

```
.data  
entero1:  
.word 42  
array:  
.word 32, 65, 76, 87
```

Visto desde un enfoque básico, cada celda en memoria tendría 4 bytes, por ende cuando me quiero desplazar entre los distintos campos tengo que ir sumando de a 4:

```
.data  
arr:  
.word 32, 65, 76  
.text  
ldr r0, =arr  
ldr r1, [r0]  
add r0, r0, #4  
ldr r2, [r0]  
add r0, r0, #4  
ldr r3, [r0]
```

ARM 64 BITS

Los procesadores ARM se caracterizaron desde un inicio por tener software compatible independientemente del hardware, de modo que la manera en que se gestionaba el consumo de las aplicaciones era algo que se encargaba el sistema operativo. De esta forma las aplicaciones se podían desligar de en que hardware se estaba corriendo.

Si vamos a lo que es arquitectura de procesadores, sabemos que la arquitectura intel siempre estuvo orientada a desempeño y que tiene software retrocompatible. Todos los procesadores de intel son capaces de leer software más antiguo.

La desventaja de intel es que en un principio las instrucciones que interactuaban con el procesador se escribían a mano, lo que llevaba a generar instrucciones complejas en las que se realizaban más de una operación, comparación, etc. El tener estas instrucciones requería tener un hardware complejo tal que pueda realizar todo eso que se solicitaba. Esto si bien es cómodo para el programador, trae dificultades para arquitecturas pipelined.

El tamaño y tiempo de ejecución de un software en intel depende directamente de la cantidad de comparaciones y operaciones que se hagan en dicha instrucción. Esto no es bueno tal que para medir performance del software yo no quiero tener variaciones de lo que puede llegar a pesar una instrucción.

El formato en el que se da la instrucción también hace variar la complejidad de su decodificación. En intel tengo que leer la instrucción, entender qué es lo que hace y en función de lo que hace interpretar cada campo de bits de dicha instrucción. Esto en ARM no pasa tal que tengo ciertos códigos representativos para cada instrucción. La forma en la que Intel solucionó este problema es a partir de **microarquitecturas**. Esta remplaza una secuencia de instrucciones en assembler a una instrucción más simple con un tiempo definido de ejecución. Esto ayuda más a hacer predicciones.

Arquitectura Intel

Ventajas:

- ❑ Procesadores orientados a desempeño
- ❑ ISA retrocompatible desde i8086 (1978) a Core i9 13th gen Raptor Lake (2023)

Desventajas:

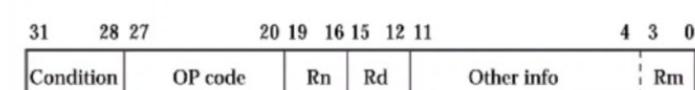
- ❑ ISA originariamente CISC (dificultad en arquitecturas pipelined)
- ❑ ISA de tamaño y tiempo de ejecución variable
- ❑ Estructura de las instrucciones dependiente del tipo de instrucción (complejidad en la decodificación)
- Solución para mitigar estos problemas: Microarquitectura

ARM intenta ofrecer un sistema más simple para que esto no pase. Todas las instrucciones son simples (constantes), de modo que todas las instrucciones ocupan 32 bits (o 64 a partir de ISA ARMv8). El tamaño y estructura es fijo y la ejecución condicional se encuentra en el mismo opcode.

La estructura de LOAD/STORE me permiten poner todos los argumentos dentro de una misma instrucción sin necesidad de hacer algo como una búsqueda en memoria.

Tengo pseudo instrucciones similares a las macros y los procesadores se pueden configurar según como registre el sistema (little o big endian).

Arquitectura ARM

- ❑ Arquitectura Superescalar RISC de 32bits (64 a partir de ISA ARMv8)
 - ❑ Instrucciones de tamaño fijo
 - ❑ Instrucciones con estructura fija
 - ❑ Ejecución condicional en el opcode
 - ❑ Estructura LOAD/STORE: Las operaciones se hacen solo entre registros
 - ❑ Pseudo instrucciones (similar a Macros)
 - ❑ Procesador Configurable mediante Registros de Sistema (BigEndian/LittleEndian, etc)
- 
- | | | | | | | | |
|-----------|---------|-------|-------|------------|----|---|---|
| 31 | 28 27 | 20 19 | 16 15 | 12 11 | 4 | 3 | 0 |
| Condition | OP code | Rn | Rd | Other info | Rm | | |

Otras ventajas de ARM 64bits:

ISA ARMv8-A: Comparación de Arquitecturas

- ❑ Se incrementa la cantidad de registros de propósito general de 15 a 31
- ❑ Todos los registros pasan a ser de 64 bits
- ❑ El espacio de memoria virtual es de 64bits
- ❑ Un proceso puede usar mas de 4GB de memoria

Hay más...

75.03/95.57 Organización del Computador

U4 - CASO DE ESTUDIO INTEL

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

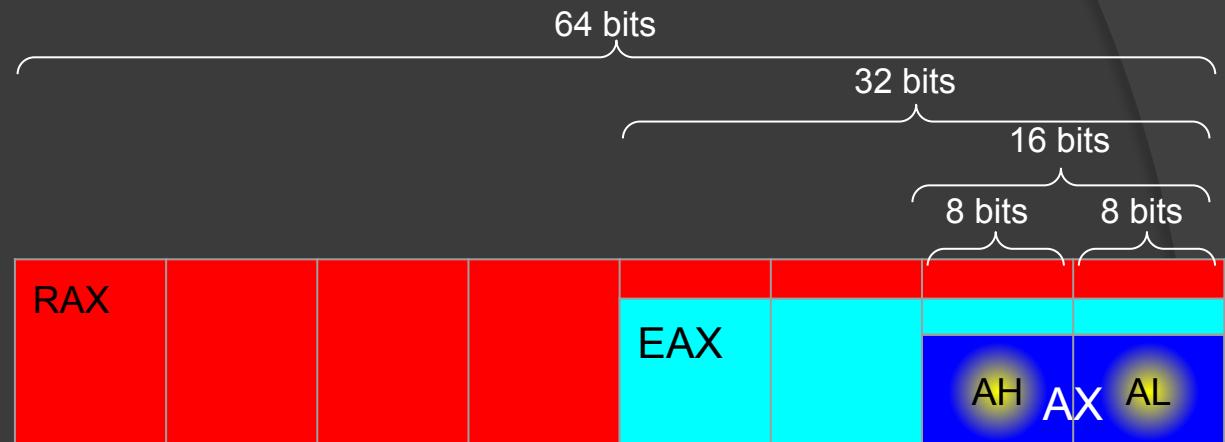
Detalle de la Arquitectura de Programación ISA (Instruction Set Architecture)

- **Registros**
 - Generales
 - Índices
 - Pila
 - Instrucción
 - Control

Registros Generales

Acumulador

operando de instrucciones aritméticas y lógicas



Base

direccionamiento de operandos



Contador

operaciones aritméticas o de string

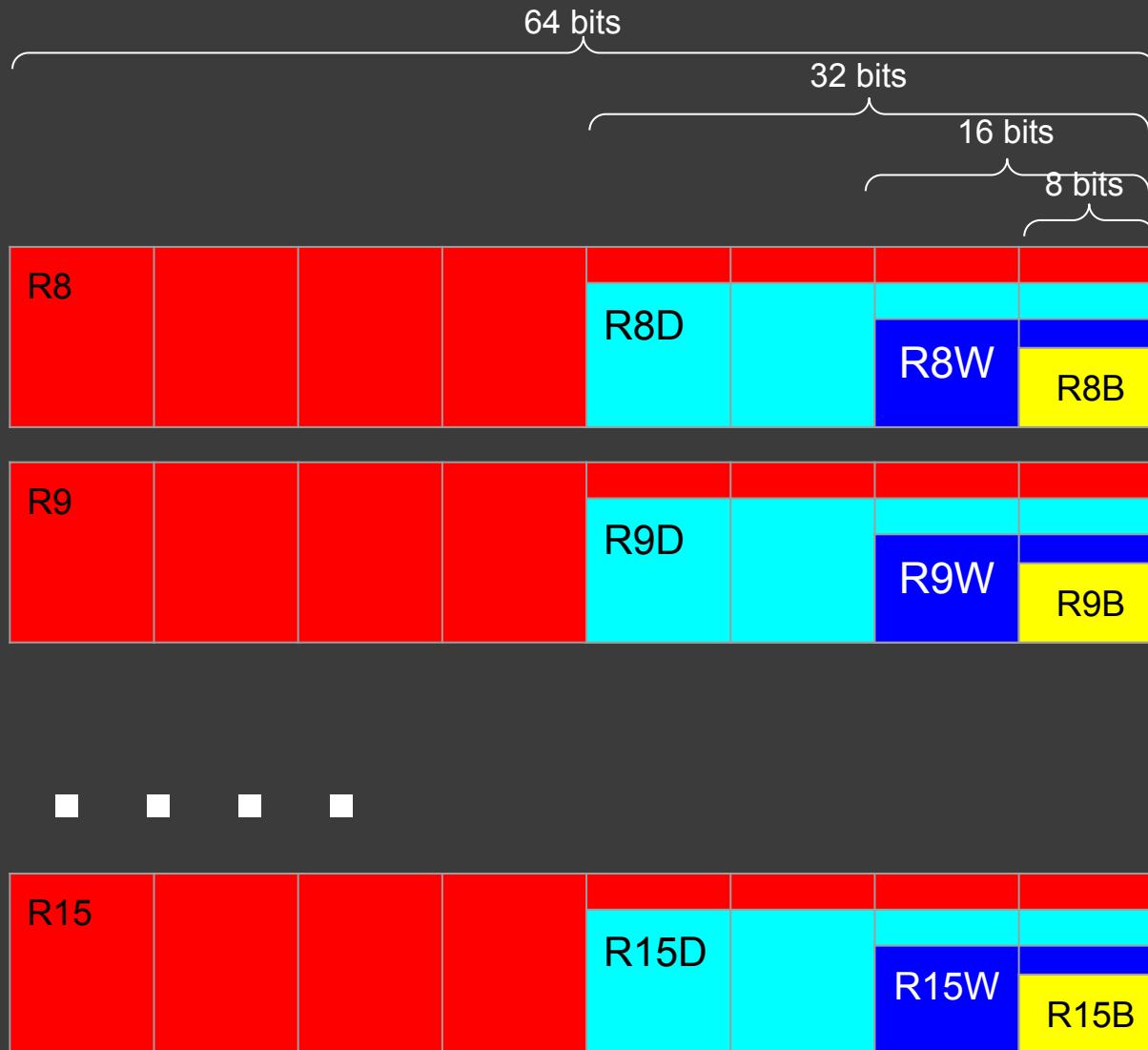


Data

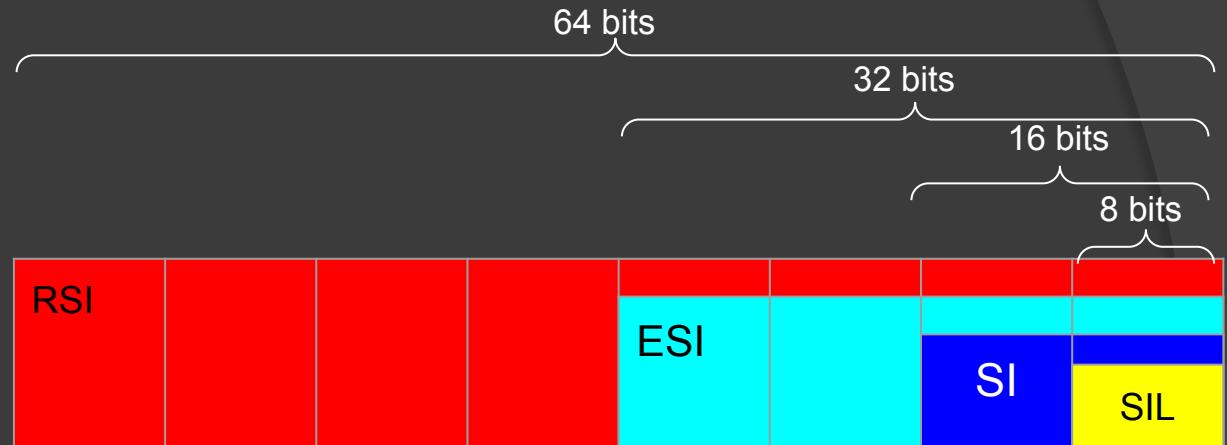
operaciones que requieren duplas de registros



Registros Generales



Registros Indice

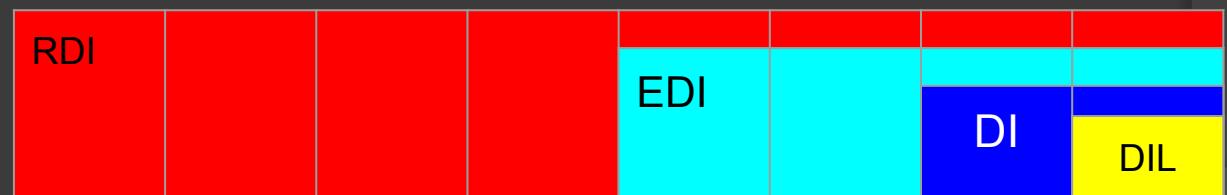


Source

operaciones de manejo de cadenas para apuntar al operando “origen”

Destination

operaciones de manejo de cadenas para apuntar al operando “destino”



Registros de Pila

Memoria programa

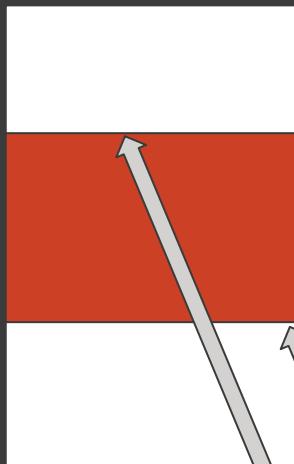
- dir

ultimo elem

1er elem

+ dir

Pila (Stack)



Base Pointer

dirección de memoria de la
base de la pila

Stack Pointer

dirección de memoria del
tope de la pila

64 bits

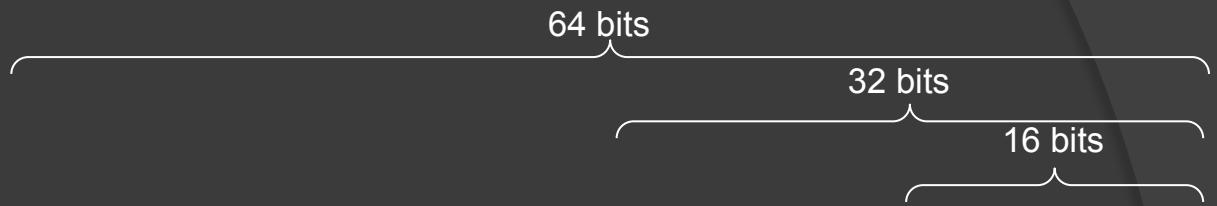
32 bits

16 bits

8 bits

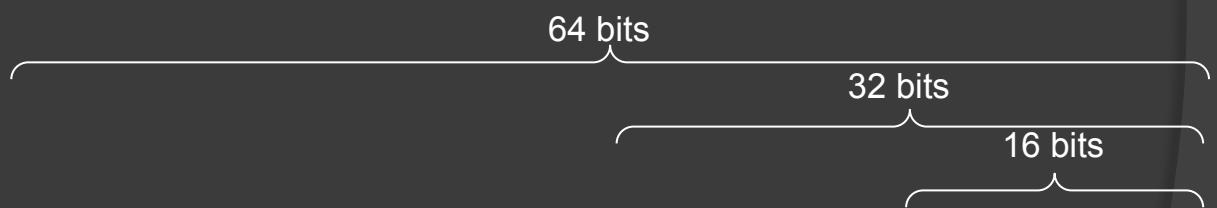
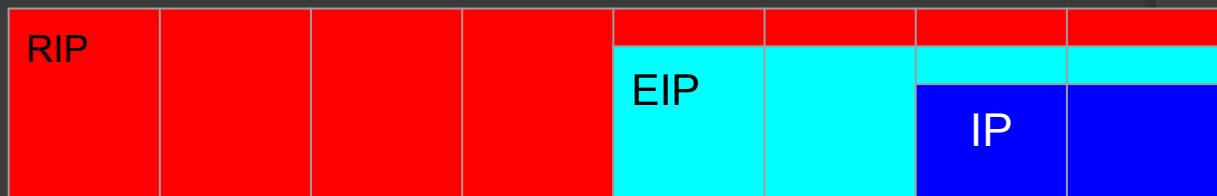


Registros de Instrucción y Control



Instruction Pointer

Actúa como registro contador de programa (PC) y contiene la dirección efectiva de la instrucción siguiente que se ha de ejecutar

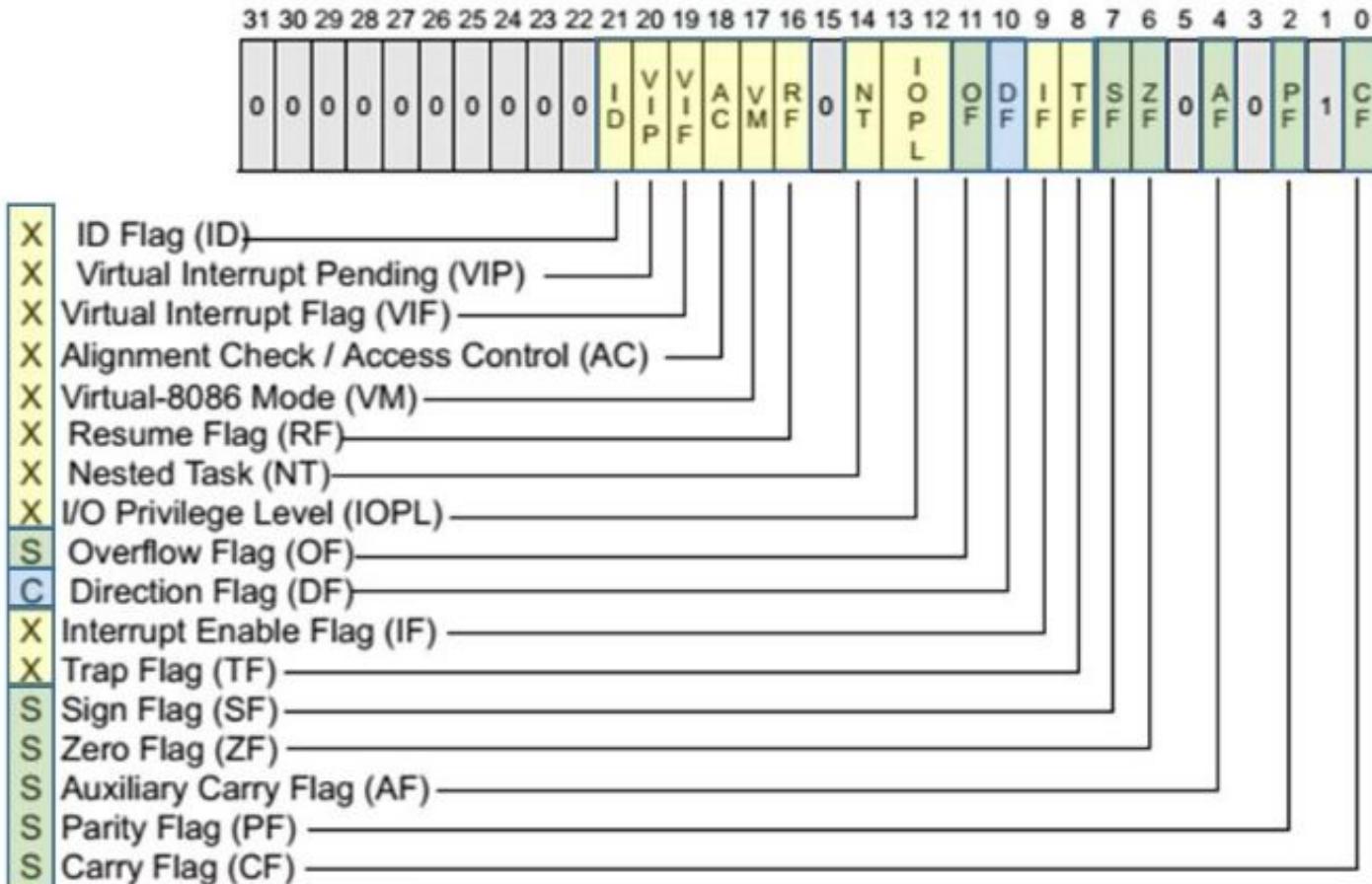


Flags

se usa para almacenar el estado general de la CPU; indica el resultado de la ejecución de cada instrucción



Registro de Control (Flags) - Detalle



IA-32 32-Bit EFLAGS Register

Reserved bit positions. DO NOT USE.
 Always set to values previously read.

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - **Direccionamiento**
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Modos de Direcciónamiento

- **Implícito:** El dato está implícito en el código de operación.

Ej. CBW

- **Registro:** El dato está en un registro.

Ej. MOV RAX, 5

- **Inmediato:** El dato está dentro de la instrucción

Ej. MOV RAX, 5

- **Directo:** El dato está en memoria referenciado por el nombre de un campo

Ej. MOV RAX, [VARIABLE]

MOV RAX, [VARIABLE + 2]

- **Registro Indirecto:** El dato está en memoria apuntado por un registro base o índice.

Ej. MOV EAX, [EBX]

MOV EAX, [ESI]

Modos de Direcciónamiento

- ◎ **Registro Relativo:** El dato está en memoria apuntado por un registro base o índice más un desplazamiento.
 - Ej. MOV RAX, [RBX+4]
 - MOV RAX, [VECTOR+RBX]
 - MOV [RDI+3], RAX
- ◎ **Base + Índice:** El dato está en memoria apuntado por un registro base más un registro índice.
 - Ej. MOV [RBX+RDI], CL
- ◎ **Base Relativo + Índice:** El dato está en memoria apuntado por un registro base más un registro índice más un desplazamiento.
 - Ej. MOV RAX, [RBX+RDI+4]
 - MOV RAX, [VECTOR+RBX+RDI]

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Tipos de dato

- ◎ **Numérico Entero**: Binario de punto fijo con Signo.
- ◎ **Numérico Decimal**: Binario de punto Flotante IEEE.
- ◎ **Caracteres**: ASCII

Memoria

- ◎ **Celda de Memoria**: 1 Byte
- ◎ **Palabra** : 2 Bytes
- ◎ **Doble Palabra** : 4 Bytes
- ◎ **Cuádruple Palabra** : 8 Bytes

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Endiannes

Es el método aplicado para almacenar datos mayores a un byte en una computadora respecto a la dirección que se le asigna a cada uno de ellos en la memoria.

Existen 2 métodos:

- **Big-Endian**: determina que el orden en la memoria coincide con el orden lógico del dato.

“el dato final en la mayor dirección”

Ej. IBM Mainframe

- **Little-Endian** : es a la inversa, el dato inicial para la lógica se coloca en la mayor dirección y el dato final en la menor.

“el dato final en la menor dirección”

Ej. Intel

Endiannes - Ejemplos

- ◎ **Caso 1:** Definición de un área de memoria con contenido inicial definido en formato carácter

```
msg    db  'HOLA'
```

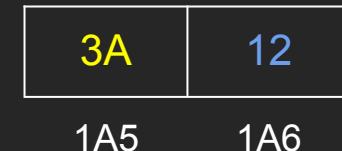
48	4F	4C	41
1A1	1A2	1A3	1A4

Aquí la posición de memoria que toma cada carácter recibido es la que por intuición uno asume, o sea, la letra 'H' en la dirección menor

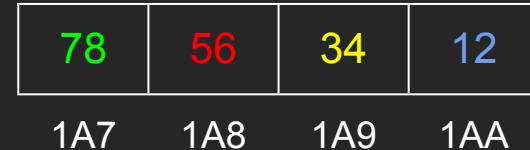
Endianes - Ejemplos

- ◎ **Caso 2:** Definición de un área de memoria con contenido inicial definido en formato numérico

numdw 4666 ; es 123A en hexa



num2 dd 12345678h



Aquí es donde se observa la ubicación de los bytes con el método Little-Endian, el byte menos significativo se ubica en la dirección de memoria menor que el byte más significativo

Endiannes - Ejemplos

- ◎ **Caso 3:** Se ejecuta una copia de memoria a registro

mov AX, [msg]



mov BX, [num1]



mov ECX, [msg]



mov EDX, [num2]



1A1 1A2 1A3 1A4



1A5 1A6



1A1 1A2 1A3 1A4



1A7 1A8 1A9 1AA

La parte alta del registro contiene el byte de orden superior de memoria, y la parte baja del registro contiene el byte de orden inferior.

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Directivas al ensamblador / Pseudo-instrucciones

Son instrucciones para que el ensamblador tome alguna acción durante el proceso de ensamblado. No se traducen a código máquina.

section	indica el comienzo de un segmento
global	indica que una etiqueta declarada en el programa es visible para un programa externo
extern	indica que una etiqueta usada en el programa pertenece a un programa externo (donde habrá sido declarada global)
db, dw, dd,dq, dt	sirven para definir áreas de memoria (variables) con contenido inicial
resb, resw, resd, resq, rest	sirven para definir áreas de memoria (variables) sin contenido inicial
times	repite una definición la cantidad de veces que se indica
%macro %endmacro	indican el inicio y final de un bloque para definir una macro
%include	permite incluir el contenido de un archivo

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - **Estructura de un programa**
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Estructura de un programa

```
global main

section .data
;variables con contenido inicial

section .bss ;block starting simbol|
;variables sin contenido inicial

section .text
;instrucciones
main:

        ret
```

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Definición y reserva de campos en memoria

- Con contenido inicial (en section .data)

db define byte (1 byte)

dw define word (2 bytes)

dd define double (4 bytes)

dq define quad (8 bytes)

dt define ten (10 bytes)

- Sin contenido inicial (en section .bss)

resb reserve byte (1 byte)

resw reserve word (2 bytes)

resd reserve double (4 bytes)

resq reserve quad (8 bytes)

rest reserve ten (10 bytes)

Definición y reserva de campos en memoria

● Ejemplos (1/4)

Definición	Bytes reservados
campo1 resb 1	1
campo2 resb 2	2
campo3 resw 1	2
campo4 resw 2	4
campo5 resd 1	4
campo4 resd 2	8
campo5 resq 1	8
campo6 resq 2	16
vector1 times 2 resb 2	4
vector2 times 3 resw 2	12

Definición y reserva de campos en memoria

● Ejemplos (2/4)

Definicion	Bytes reservados	Contenido memoria
decimal1 db 11	1	0B
decimal2 dw -11	2	F5 FF
decimal3 dd 12345	4	39 30 00 00
decimal4 dq -1	8	FF FF FF FF FF FF FF FF
hexa1 db -0Bh	1	F5
hexa2 dw 0Ch	2	0C 00
hexa3 dd FFFFh	4	FF FF 00 00
hexa4 dq 96B43Fh	8	3F B4 96 00 00 00 00 00

Definición y reserva de campos en memoria

● Ejemplos (3/4)

Definicion		Bytes reservados	Contenido memoria
octal1	db 13o	1	0B
octal2	dw 71o	2	39 00
octal3	dd 10o	4	08 00 00 00
binario1	db 1011b	1	0B
binario2	dw 1011b	2	0B 00
binario3	dd -1000b	4	F8 FF FF FF
truncado	db 2571 ;0A0Bh	1	0B
noTruncado	dw 2571 ;0A0Bh	2	0B 0A

Definición y reserva de campos en memoria

● Ejemplos (4/4)

Definicion		Bytes reservados	Contenido memoria
letra	db 'A'	1	41
letra2	dw 'A'	2	41 20
letra3	db 'a'	1	61
cadena	db 'hola'	4	68 6F 6C 61
cadena2	dw 'ola'	4	6F 6C 61 20
numero	db '12'	2	31 32
vector1	times 3 db 'A'	3	41 41 41
vector2	times 3 db 'A', 0	6	41 00 41 00 41 00
registro	times 0 db 'A'	0	n/a

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Macros

Son secuencias de instrucciones asignadas a un nombre que pueden usarse en cualquier parte del programa.

La sintaxis es:

```
%macro macro_name    number_of_params  
<macro body>  
%endmacro
```

Macros

Sin parámetros

```
1 %macro mPuts 0
2     sub      rsp,8
3     call     puts
4     add      rsp,8
5 %endmacro
6 global main
7 extern puts
8
9 section .data
10    mensaje   db    "Hola mundo!",0
11    mensaje2  db    "Chau!",0
12
13 section .text
14 main:
15     mov      rdi,mensaje
16     mPuts
17     mov      rdi,mensaje2
18     mPuts
19     ret
```

Con 1 parámetro

```
1 %macro mPuts 1
2     mov      rdi,%1
3     sub      rsp,8
4     call     puts
5     add      rsp,8
6 %endmacro
7 global main
8 extern puts
9
10 section .data
11    mensaje   db    "Hola mundo!",0
12    mensaje2  db    "Chau!",0
13
14 section .text
15 main:
16     mPuts    mensaje
17     mPuts    mensaje2
18     ret
```

Inclusión de archivos

Mediante el uso de una directiva es posible incluir el contenido de un archivo dentro del código.

La sintaxis es:

```
%include "file_name"
```

Inclusión de archivos

misMacros.asm X

```
1 %macro mPuts 1
2     mov             rdi,%1
3     sub             rsp,8
4     call            puts
5     add             rsp,8
6 %endmacro
7 extern puts
```

test-include.asm X

```
1 %include "misMacrosL.asm"
2 global main
3 section .data
4     mensaje    db      "Hola mundo!!",0
5
6 section .text
7 main:
8     mPuts    mensaje
9     ret
```

```
PS C:\Users\Dario\Documents\Orga\include> nasm test-include.asm -fwin64
PS C:\Users\Dario\Documents\Orga\include> gcc test-include.obj
PS C:\Users\Dario\Documents\Orga\include> ./a.exe
Hola mundo!!
PS C:\Users\Dario\Documents\Orga\include>
```

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Instrucciones - Transferencia y Copia

MOV op1, op2

Copia el valor del 2do operando en el primer operando.

Combinaciones	Ejemplos en NASM
MOV <reg>,<reg>	MOV AH,BL MOV AX,BX MOV ECX, EAX MOV RDX,RCX
MOV <reg>,<mem>	MOV CH,[VARIABLE_8] (*) MOV CX, [VARIABLE_16] (*) MOV ECX, [VARIABLE_32] (*) MOV RDX, [VARIABLE_64] (*)
MOV <reg>,<inm>	MOV DL,7o MOV CX,2450h MOV EAX,0h MOV RDX,28h

Instrucciones - Transferencia y Copia

MOV op1, op2

Combinaciones	Ejemplos en NASM
MOV <mem>,<reg>	MOV [VARIABLE_8],AH (*) MOV [VARIABLE_16],AX (*) MOV [VARIABLE_32],EAX (*) MOV [VARIABLE_64],RAX (*) MOV VARIABLE_64,RAX
MOV <long><mem>,<inm>	MOV byte [VARIABLE_8],2Ah MOV word [VARIABLE_16],7770 MOV dword [VARIABLE_32],1234 MOV qword [VARIABLE_64],1234 MOV [VARIABLE_64],4321

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Instrucciones - Comparación

CMP op1, op2

Compara el contenido del op1 contra el op2 mediante la resta entre los dos operando sin modificarlos.

Combinaciones	Ejemplos en NASM
CMP <reg>,<reg>	CMP AH,BL CMP AX,BX CMP EAX,EBX CMP RCX,RAX
CMP <reg>,<mem>	CMP CH,[VARIABLE_8] (*) CMP CX,[VARIABLE_16] (*) CMP EAX,[VARIABLE_32] (*) CMP RBX,[VARIABLE_64] (*)

Instrucciones - Comparación

CMP op1 , op2

Combinaciones	Ejemplos en NASM
CMP <reg>,<inm>	CMP CH,10o CMP CX,2Ah CMP EAX,1000 CMP RBX,432h
CMP <mem>,<reg> (*) Considera la long del REG	CMP VARIABLE,AX CMP [VARIABLE_8],RAX (*) CMP [VARIABLE_8],AH CMP [VARIABLE_64],RCX
CMP <long><mem>,<inm>	CMP VARIABLE,245h CMP byte[VARIABLE_8],2Ah CMP word[VARIABLE_16],2Ah CMP dword[VARIABLE_32],2Ah CMP qword[VARIABLE_64],2Ah

Instrucciones - Saltos/Bifurcaciones

JMP op

Bifurca a la dirección indicada del operando.

Jcc op

Bifurca a la dirección indicada del operando si se cumple la condición.

Combinaciones	Ejemplos en NASM
JMP <etiqueta>	JMP finalizar
Jcc <etiqueta>	JE esIgual

Instrucciones - Saltos/Bifurcaciones

Condicionales generales

JE	op	-> por igual ($ZF=1$)
JNE	op	-> por no igual ($ZF=0$)
JZ	op	-> por igual a cero ($ZF=1$)
JNZ	op	-> por distinto a cero ($ZF=0$)
JRCXZ	op	-> por contenido de RCX igual cero
JC	op	-> por carry flag distinto a cero ($CF=1$)
JO	op	-> por overflow ($OF=1$)

Instrucciones - Saltos/Bifurcaciones

Condicionales con signo

JG	op	-> por mayor (ZF=0 and SF=OF)
JGE	op	-> por mayor o igual (SF=OF)
JL	op	-> por menor (SF<> OF)
JLE	op	-> por menor o igual (ZF=1 or SF<> OF)
JNG	op	-> por no mayor (ZF=1 or SF<>OF)
JNGE	op	-> por no mayor o igual (SF<>OF)
JNL	op	-> por no menor (SF=OF)
JNLE	op	-> por no menor o igual (ZF=0 and SF=OF)

Instrucciones - Saltos/Bifurcaciones Condicionales sin signo

JA	op	-> por mayor (CF=0 and ZF=0)
JAE	op	-> por mayor o igual (CF=0)
JB	op	-> por menor (CF=1)
JBE	op	-> por menor o igual (CF=1 or ZF=1).
JNA	op	-> por no mayor (CF=1 or ZF=1)
JNAE	op	-> por no mayor o igual (CF=1)
JNB	op	-> por no menor (CF=0)
JNBE	op	-> por no menor o igual CF=0 and ZF=0)

Instrucciones - Aritmeticas

Suma

ADD op₁, op₂

Suma los valores de los dos operando (binarios de punto fijo con signo) dejando el resultado en el primero.

Combinaciones	Ejemplos en NASM
ADD <reg>,<reg>	ADD AH,BL ADD AX,BX ADD ECX,EAX ADD RDX,RCX
ADD <reg>,<mem>	ADD AL,[VARIABLE_8] (*) ADD BX,[VARIABLE_16] (*) ADD ECX,[VARIABLE_32] (*) ADD RDX,[VARIABLE_64] (*)

Instrucciones - Aritmeticas

Suma (continuación)

ADD op1 , op2

Combinaciones	Ejemplos en NASM
ADD <reg>,<inm>	ADD DL,10b ADD AX,10h ADD ECX,1234 ADD RCX,1234h
ADD <mem>,<reg> (*) Considera la long del REG	ADD VARIABLE_16,AX ADD [VARIABLE_8],AL ADD [VARIABLE_8],BX (*) ADD [VARIABLE_32],ECX ADD [VARIABLE_64],RDX
ADD <long><mem>,<inm>	ADD VARIABLE_123 ADD byte[RDI],245h ADD word[VARIABLE_16],2Ah ADD dword[VARIABLE_32],123 ADD qword[VARIABLE_64],2Ah

Instrucciones - Aritméticas

Resta

SUB op₁, op₂

Resta los valores de los dos operando (binarios de punto fijo con signo) dejando el resultado en el primero.

Combinaciones	Ejemplos en NASM
SUB <reg>,<reg>	SUB AH,BL SUB AX,BX SUB ECX,EAX SUB RDX,RBX
SUB <reg>,<mem>	SUB AL,[VARIABLE_8] (*) SUB BX,[VARIABLE_16] (*) SUB ECX,[VARIABLE_32] (*) SUB RDX,[VARIABLE_64] (*)

Instrucciones - Aritméticas

Resta (continuación)

SUB op1, op2

Combinaciones	Ejemplos en NASM
SUB <reg>,<inm>	SUB DL,10b SUB AX,10h SUB ECX,1234 SUB RDX,1234h
SUB <mem>,<reg> (*) Considera la long del REG	SUB VARIABLE,AX SUB [VARIABLE_8],AL SUB [VARIABLE_8],BX(*) SUB [VARIABLE_32],ECX SUB [VARIABLE_64],RDX
SUB <long><mem>,<inm>	SUB VARIABLE,123 SUB byte[EDI],245h SUB word[VARIABLE],2Ah SUB dword[VARIABLE],123 SUB qword[VARIABLE],123h

Instrucciones - Aritmeticas

Incremento y Decremento

INC op

Suma uno al operando (binarios de punto fijo con signo)

DEC op

Resta uno al operando (binarios de punto fijo con signo)

Combinaciones	Ejemplos en NASM
INC/DEC <reg>	INC/DEC BH <input type="checkbox"/> $BH = BH + 1$ / $BH = BH - 1$ INC/DEC CX <input type="checkbox"/> $CX = CX + 1$ / $CX = CX - 1$ INC/DEC EAX <input type="checkbox"/> $EAX = EAX + 1$ / $EAX = EAX - 1$ INC/DEC RDX <input type="checkbox"/> $RDX = RDX + 1$ / $RDX = RDX - 1$
INC/DEC <mem>	INC byte[VAR_8B] INC word[VAR_16B] INC dword[VAR_32B] <input type="checkbox"/> $VAR_32B = VAR_32B + 1$ INC qword[VAR_64B] DEC byte[VAR_8B] DEC word[VAR_16B] <input type="checkbox"/> $VAR_16B = VAR_16B - 1$ DEC dword[VAR_32B] DEC qword[VAR_64B]

Instrucciones - Aritmeticas

Multiplicación - Formato 1 operando

IMUL op

Si longitud de op es 8 bits:

Multiplica AL * op y deja el resultado en AX

Si longitud de op es 16 bits:

Multiplica AX * op y deja el resultado en DX : AX

Si longitud de op es 32 bits:

Multiplica EAX * op y deja el resultado en EDX : EAX

Si longitud de op es 64 bits:

Multiplica RAX * op y deja el resultado en RDX : RAX

Los operandos son interpretados como binario de punto fijo CON signo

MUL op

Igual que IMUL pero los operandos son interpretados como binario de punto fijo SIN signo

Instrucciones - Aritméticas

Multiplicación - Formato 1 operando (cont)

IMUL op

MUL op

Combinaciones	Ejemplos en NASM
MUL/IMUL <reg>	MUL/IMUL BH <input type="checkbox"/> AX = (AL) *(BH) MUL/IMUL BX <input type="checkbox"/> DX:AX = (AX) *(BX) MUL/IMUL EBX <input type="checkbox"/> EDX:EAX = (EAX) *(EBX) MUL/IMUL RCX <input type="checkbox"/> RDX:RAX = (RAX) *(RCX)
MUL/IMUL <mem>	MUL/IMUL byte[VAR_8] <input type="checkbox"/> AX = (AL) *(VAR_8) MUL/IMUL word[VAR_16] <input type="checkbox"/> DX:AX = (AX) *(VAR_16) MUL/IMUL dword[VAR_32] <input type="checkbox"/> EDX:EAX = (EAX) *(VAR_32) MUL/IMUL qword[VAR_64] <input type="checkbox"/> RDX:RAX = (RAX) *(VAR_64)

Instrucciones - Aritméticas

Multiplicación - Formato 2 operandos

IMUL op1, op2

Multiplica el contenido de los operandos y almacena el resultado en el primero.

El op1 debe ser un registro siempre y ambos operandos deben tener la misma longitud.

Si el resultado no entra en el operando 1, se trunca.

Los operandos son interpretados como binario de punto fijo CON signo

MUL op1, op2

Igual que IMUL pero los operandos son interpretados como binario de punto fijo SIN signo

Instrucciones - Aritméticas

Multiplicación - Formato 2 operandos (cont)

IMUL op1, op2

MUL op1, op2

Combinaciones	Ejemplos en NASM
IMUL <reg>, <inm>	IMUL CX,4 \square CX = (CX)*4
IMUL <reg>,<reg>	IMUL EAX,EBX \square EAX = (EAX)*(EBX)
IMUL <reg>,<log><mem>	IMUL RBX, qword [VAR_64] \square RBX = (RBX)*(VAR_64)

Instrucciones - Aritméticas

Multiplicación - Formato 3 operandos

IMUL op1, op2, op3

Multiplica el contenido de los operandos 2 y 3 y almacena el resultado en el operando 1.

El operando 1 es siempre un registro y el operando 3 siempre un valor inmediato. Tanto el op1 como el op2 deben tener la misma longitud.

Si el resultado no entra en el operando 1, se trunca.

Los operandos son interpretados como binario de punto fijo CON signo

MUL op1, op2, op3

Igual que IMUL pero los operandos son interpretados como binario de punto fijo SIN signo

Instrucciones - Aritméticas

Multiplicación - Formato 3 operandos (cont)

IMUL op1, op2, op3

MUL op1, op2, op3

Combinaciones	Ejemplos en NASM
IMUL <reg>, <reg>,<inm>	IMUL CX,BX,10h □ CX = (BX)*10h
IMUL <reg>,<mem>,<inm>	IMUL RBX, qword [VAR_64],4 □ RBX = (VAR_64)*4

Instrucciones - Aritméticas

División

IDIV op

Si longitud de op es 8 bits:

AX / op resto en AH y cociente en AL

Si longitud de op es 16 bits:

DX : AX / op resto en DX y cociente en AX

Si longitud de op es 32 bits:

EDX : EAX / op resto en EDX y cociente en EAX

Si longitud de op es 64 bits:

RDX : RAX / op resto en RDX y cociente en RAX

Los operandos son interpretados como binario de punto fijo CON signo

DIV op

Igual que IDIV pero los operandos son interpretados como binario de punto fijo SIN signo

Instrucciones - Aritméticas

División (cont)

IDIV op

DIV op

Combinaciones	Ejemplos en NASM
DIV/IDIV <reg>	<p>DIV/IDIV BX □ DX:AX / BX</p> <ul style="list-style-type: none">• Cociente = AX• Resto = DX <p>DIV/IDIV EBX □ EDX:EAX / EBX</p> <ul style="list-style-type: none">• Cociente = EAX• Resto = EDX <p>DIV/IDIV RCX □ RDX:RAX / RCX</p> <ul style="list-style-type: none">• Cociente = RAX• Resto = RDX
DIV/IDIV <long><mem>	<p>DIV/IDIV byte[VAR_8] □ AX / [VAR_8]</p> <ul style="list-style-type: none">• Cociente = AL• Resto = AH <p>DIV/IDIV dword[VAR_32] □ EDX:EAX / [VAR_32]</p> <ul style="list-style-type: none">• Cociente = EAX• Resto = EDX <p>DIV/IDIV qword[VAR_64] □ RDX:RAX / [VAR_64]</p> <ul style="list-style-type: none">• Cociente = RAX• Resto = RDX

Instrucciones - Aritméticas

Conversión

CBW

Convierte el byte almacenado en AL a una word en AX.

CWD

Convierte la word almacenada en AX a una double-word en DX:AX

CWDE

Convierte la word almacenada en AX a una double-word en EAX

CDQE

Convierte la doble-word almacenada en EAX a una quad-word en RAX

Expandan en signo del operando

Ejemplo en NASM

MOV AL,byte[VAR_8B]
CBW
CWDE
CDQE

VAR_8B db 9Bh □ AL = 9B
AX = FF9B
EAX = FFFFFFF9B
RAX = FFFFFFFFFFFFF9B

MOV AX,word[VAR_16B]
CWD
CWDE
CDQE

VAR_16B dw FF9Bh □ AX = FF9B
DX:AX = FFFFh:FF9B
EAX = FFFFFFF9B
RAX = FFFFFFFFFFFFF9B

Instrucciones - Aritméticas

Complemento

NEG op

Realiza el complemento a 2 del operando, es decir, le cambia el signo.

Combinaciones	Ejemplos en NASM
NEG <reg>	NEG BH NEG AX NEG ECX NEG RDX
NEG <mem>	NEG byte[VAR_8] NEG word[VAR_16] NEG dword[VAR_32] NEG qword[VAR_64]

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Instrucciones - Saltos/Bifurcaciones

Loop

LOOP op

Resta 1 al contenido del registro RCX y si el resultado es distinto de 0, bifurca al punto indicado por el operando, sino continua la ejecución en la instrucción siguiente.

El desplazamiento al punto indicado debe estar en un rango entre -128 a 127 bytes (*near jump*)

```
        mov    rcx, 5
inicio:
    . . .
    . . .
loop  inicio
    . . .
```

Instrucciones - Saltos/Bifurcaciones

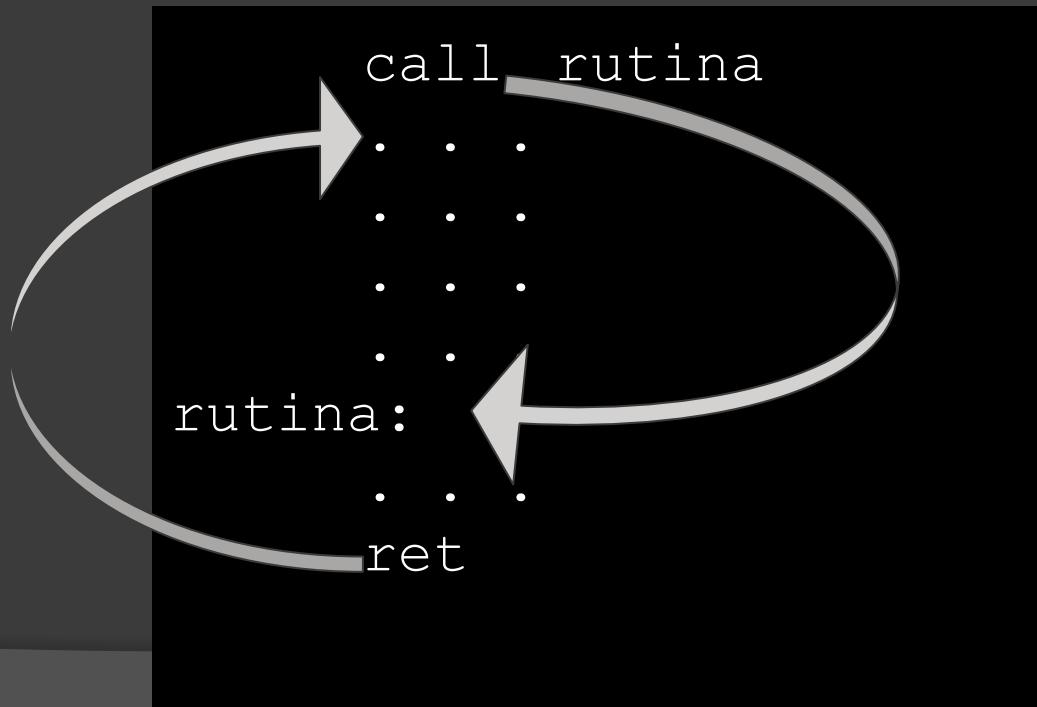
Llamada y Retorno de procedimiento

CALL op

Almacena en la pila la dirección de la instrucción siguiente a la call y bifurca al punto indicado por el operando.

RET

Toma el elemento del tope de la pila que debe ser una dirección de memoria (generalmente cargada por una call) y bifurca hacia la misma.



Instrucciones - Saltos/Bifurcaciones

Rutinas externas (1/2)

```
moduleHello.asm X  
1 global sayHello  
2 extern puts  
3 section .data  
4 mensaje db "Hello",0  
5 section .text  
6 sayHello:  
7     mov rdi,mensaje  
8     sub rsp,8  
9     call puts  
10    add rsp,8  
11    ret
```

```
moduleBye.asm X  
1 global sayGoodbye  
2 extern puts  
3 section .data  
4 mensaje db "Goodbye",0  
5 section .text  
6 sayGoodbye:  
7     mov rdi,mensaje  
8     sub rsp,8  
9     call puts  
10    add rsp,8  
11    ret
```

```
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ nasm moduleHello.asm -f elf64  
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ nasm moduleBye.asm -f elf64  
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ █
```



Instrucciones - Saltos/Bifurcaciones

Rutinas externas (2/2)

```
* main_external_routines.asm ×

1  global  main
2  extern   sayHello
3  extern   sayGoodbye
4  section  .data
5  section  .text
6  main:
7      sub    rsp,8
8      call   sayHello
9      add    rsp,8
10
11     sub    rsp,8
12     call   sayGoodbye
13     add    rsp,8
14     ret
```

```
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ nasm main_external_routines.asm -f elf64
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ gcc main_external_routines.o moduleHello.o moduleBye.o -no-pie
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$ ./a.out
Hello
Goodbye
dario@dario-VirtualBox:/media/sf_Orga/rutinas externas$
```

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Tablas

Tira de bytes en memoria destinada a usar como una estructura de Vector o Matriz

```
tabla      times 40 resb 1  
vector     times 10 resw 1  
matriz     times 25 db "*"
```

Posicionamiento en el elemento i de un vector

$$(i - 1) * longitudElemento$$

Posicionamiento en el elemento i,j de una matriz

$$(i-1)*longitudFila + (j-1)*longitudElemento$$

longitudFila= longitudElemento*cantidadColumnas

○ Tablas (Cont.)

Dada una matriz (4 columnas x 3 filas) del tipo doble → se pide cargar un valor en el elemento de la fila 2 y columna 3

```
posx      dd 02
posy      dd 03
longfil   dd 16
longele   dd 4
matriz    times 12 resd 1
. . . . .
mov rbx,matriz ;pongo el pto al inicio de la matriz
mov rax,dword[posx] ;guardo el valor de la fila
sub rax,1
imul    dword[longfil] ;me desplazo en la fila
add rcx,rax
mov rax,dword[posy] ;guardo el valor de la fila
sub rax,1
imul    dword[longele] ;me desplazo en la columna
add rcx,rax      ;sumo los desplazamientos
add rbx,rcx      ;me posicione en la matriz

mov dword[rbx],XXX ;muevo el valor
```

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Validación

Código destinado a verificar que los datos de un programa provenientes del exterior (teclado, archivos) cumplen las condiciones esperadas y/o necesarias.

Clasificación según el contenido del dato:

- **Lógica:** que se adecúe al significado lógico
 - Ej: Dia de la semana: lunes, martes, miércoles, etc
 - Respuesta “S” o “N”
- **Física:** que el dato esté en un formato particular
 - Ej: Campo en formato empaquetado
 - Fecha en formato DD/MM/AA

Clasificación según el mecanismo aplicado

- **Por valor:** comparar contra uno o varios valores válidos
- **Por Rango:** comparar que esté dentro de un rango continuo válido
- **Por tabla:** buscar que exista en una tabla de valores válidos

Caso de estudio Intel

Agenda

- **ISA (Instruction Set Architecture)**
 - Registros
 - Direccionamiento
 - Tipos de dato
 - Memoria
 - Endianess
- **Ensamblador NASM (Netwide Assembler)**
 - Directivas/Pseudo-instrucciones
 - Estructura de un programa
 - Definición y reserva de campos de memoria
 - Macros e Inclusión de archivos
 - Instrucciones
- **Conceptos generales**
 - Tablas
 - Validación

Instrucciones - Lógicas

And

AND op1, op2

Ejecuta la operación lógica AND entre el operando 1 y 2 dejando el resultado en el 1

Combinaciones	Ejemplos en NASM
AND <reg>,<reg>	AND AH,BL / AND AX,BX / AND ECX,EAX
AND <reg>,<long><mem>	AND AL, byte [VAR_8B] AND ECX, dword [VAR_32B]
AND <reg>,<inm>	AND CH,00h / AND CX,250h / AND CX,3456
AND <long><mem>,<inm>	AND word [VAR_16B], 1010b AND dword [VAR_32B],FFCC00AAh
AND <long><mem>,<reg>	AND byte [VAR_8B],BL AND dword [VAR_32B],EAX

Instrucciones - Lógicas

Or

OR op1, op2

Ejecuta la operación lógica OR entre el operando 1 y 2 dejando el resultado en el 1

Combinaciones	Ejemplos en NASM
OR <reg>,<reg>	OR AH,BL / OR AX,BX / OR ECX,EAX
OR <reg>,<long><mem>	OR AL, byte [VAR_8B] OR ECX, dword [VAR_32B]
OR <reg>,<inm>	OR CH,00h / OR CX,250h / OR CX,3456
OR <long><mem>,<inm>	OR word [VAR_16B], 1010b OR dword [VAR_32B],FFCC00AAh
OR <long><mem>,<reg>	OR byte [VAR_8B],BL OR dword [VAR_32B],EAX

Instrucciones - Lógicas

Exclusive or

XOR op1, op2

Ejecuta la operación lógica EXCLUSIVE OR entre el operando 1 y 2 dejando el resultado en el 1

Combinaciones	Ejemplos en NASM
XOR <reg>,<reg>	XOR AH,BL / XOR AX,BX / XOR ECX,EAX
XOR <reg>,<long><mem>	XOR AL, byte [VAR_8B] XOR ECX, dword [VAR_32B]
XOR <reg>,<inm>	XOR CH,00h / XOR CX,250h / XOR CX,3456
XOR <long><mem>,<inm>	XOR word [VAR_16B], 1010b XOR dword [VAR_32B],FFCC00AAh
XOR <long><mem>,<reg>	XOR byte [VAR_8B],BL XOR dword [VAR_32B],EAX

Instrucciones - Lógicas

Not

NOT op

Ejecuta la operación lógica NOT en el operando

Combinaciones	Ejemplos en NASM
NOT <reg>	NOT AH NOT BX NOT ECX
NOT <mem>	NOT byte[VAR_8B] NOT word[VAR_16B] NOT dword[VAR_32B]

Instrucciones - Transferencia y Copia

LEA op1, op2

Copia en el operando 1 (un registro) la dirección de memoria del operando 2.

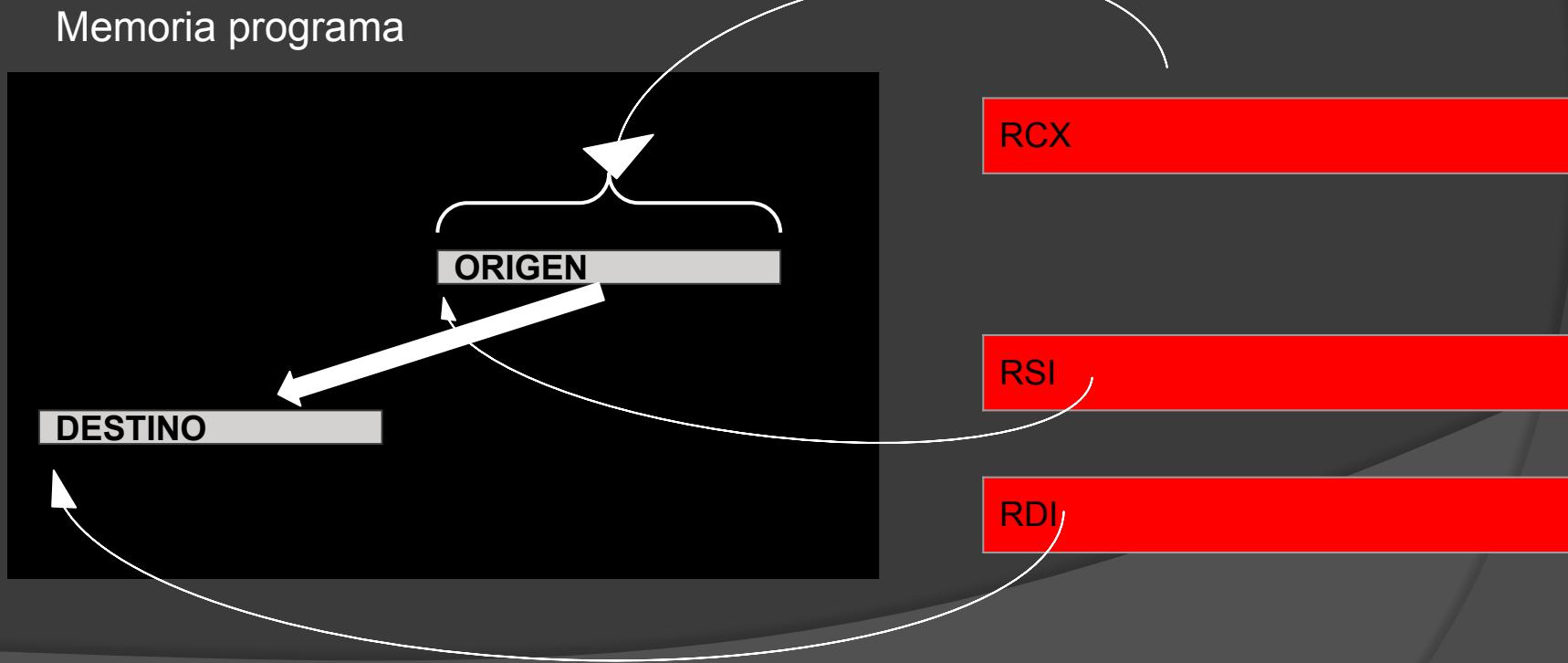
Combinaciones	Ejemplos en NASM
LEA <reg>,<mem>	LEA RAX,[VARIABLE] es equivalente a hacer MOV RAX,VARIABLE ;notar q aca NO hay corchetes

Instrucciones - Transferencia y Copia

Copia de strings

MOVSB

Copia el contenido de memoria apuntado por RSI (origen/source) al apuntado por RDI (destino/destination). Copia tantos bytes como los indicados en el registro RCX



Instrucciones - Transferencia y Copia

Copia de strings (cont)

MOVSB

```
    . . .
    MOV    RCX, 4
    LEA    RSI, [MSGORI]
    LEA    RDI, [MSGDES]
    REP    MOVSB
    . . .
```

Instrucciones - Comparación

Comparación de strings

CMPSB

Compara el contenido de memoria apuntado por RSI (origen/source) con el apuntado por RDI (destino/destination). Compara tantos bytes como los indicados en el registro RCX

```
    . . .
    MOV    RCX, 4
    LEA    RSI, [MSG1]
    LEA    RDI, [MSG2]
    REPE  CMPSB
    JE     IGUALES
    . . .
```

Instrucciones - Transferencia y Copia

Manejo de la pila (stack)

PUSH op

Inserta el operando (de 64 bits) en la pila. Decrementa (resta 1) el contenido del registro RSP

POP op

Elimina el último elemento insertado en la pila (de 64 bits) y lo copia en el operando. Incrementa (suma 1) al contenido del registro RSP

Combinaciones	Ejemplos en NASM
PUSH / POP <reg>	PUSH / POP RDX
PUSH / POP qword <mem>	PUSH / POP qword [VARIABLE]

◦ Manejo de Pila

- La pila es usada para almacenar transitoriamente datos, direcciones de retornos de subrutinas o pasar parámetros a funciones o subrutinas.
- El ultimo que entra es el primero que sale □ LIFO.
- PUSH sirve para poner el dato en la pila mientras que POP se usa para recuperar el dato.
- El stack Pointer (SP) se incrementa con el POP y se decrementa con el PUSH.
- El operando puede ser un registro o posición de memoria(ambos 64bits).

Combinaciones	Ejemplos en NASM
PUSH / POP <reg>	PUSH / POP RDX
PUSH / POP qword <mem>	PUSH / POP qword [VARIABLE]

Código objeto

El **código objeto** es una representación en lenguaje máquina o bytecode del código fuente de un programa, generado por un compilador o ensamblador. Este código no es directamente ejecutable; para obtener un programa ejecutable final, es necesario un proceso de enlace realizado por un enlazador (linker), que combina uno o más archivos de código objeto y resuelve las referencias entre ellos.

es.wikipedia.org

La estructura típica de un archivo de código objeto incluye las siguientes secciones:

1. **Cabecera (header):** Contiene información esencial para interpretar el archivo, como el formato del archivo, la arquitectura de destino y otros metadatos relevantes.
2. **Tabla de símbolos:** Esta tabla es crucial para el proceso de enlace, ya que permite resolver referencias entre diferentes módulos o bibliotecas. Incluye:
 - **Símbolos definidos dentro del módulo:** Aquellos que pueden ser referenciados desde otros módulos.
 - **Símbolos externos:** Aquellos que el módulo utiliza pero que se definen en otros lugares.
3. **Código de máquina y datos:** Contiene el código de máquina generado a partir del código fuente, así como las constantes y datos definidos en el programa.
4. **Información de reubicación:** Incluye entradas que indican al enlazador qué direcciones deben ajustarse cuando el módulo se carga en una dirección de memoria diferente a la originalmente asumida. Esto permite que el código objeto sea flexible y pueda ser cargado en diferentes ubicaciones de memoria sin problemas.

Es importante destacar que, aunque el código objeto contiene instrucciones en lenguaje máquina, no es directamente ejecutable. Requiere el proceso de enlace para combinarlo con otros módulos y bibliotecas, resolviendo todas las referencias y generando el archivo ejecutable final que puede ser cargado y ejecutado por el sistema operativo.

End of module

Relocation
dictionary

Machine instructions
and constants

External reference table

Entry point table

Identification

Componentes de un computador

Memoria

Se le denomina memoria al conjunto de elementos de una computadora destinados al almacenamiento de información. Dentro de las características de la misma entra la temporalidad, que puede ser volátil o persistente (ram o disco duro). Luego hay todo un mundo que es el almacenamiento secundario.

Se pueden diferenciar cualidades como la tecnología, la organización, el performance y el costo. Se identifica una jerarquía en memoria que se divide entre internos y externos al sistema:

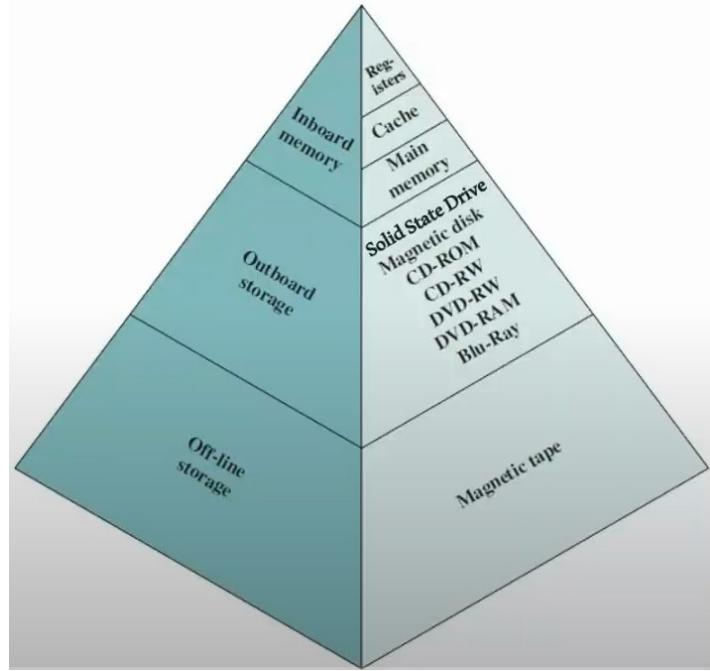
○ Memoria

- Componente complejo (Sistema de memoria)
- Formado por elementos con distintas cualidades:
 - Tecnología
 - Organización
 - Performance
 - Costo
- Jerarquía de subsistemas de memoria
 - Internos al sistema (accedidos directamente por el procesador)
 - Externos al sistema (accedidos por el procesador a través de un módulo de E/S)

Dentro de estos subsistemas se mide la **capacidad, el tiempo de acceso y el costo**. De acuerdo a que tanta capacidad, que tan veloz es y que tan alto es el costo, el elemento estará más alto en la “pirámide” de jerarquía. Los subsistemas de memoria se ven comprometidos al hacer un balance entre estas características.

○ Jerarquía de memoria

- Tres características a tener en cuenta
 - Capacidad
 - Tiempo de acceso
 - Costo
- Subsistemas de memoria con relación de compromiso entre estas características =>
No se usa un solo componente de memoria



Memoria interna (guardado de información volatil / temporal)

El primer elemento que figura en la piramide son los registros y los tomamos como un almacenamiento interno. Se los reconoce como un conjunto de biestables, donde se permite almacenar pequeñas cantidades de información en forma de bits . Almacena los elementos de forma temporal, no se persiste. Las instrucciones cargadas en los mismos suceden millones de veces por segundo. Se encuentran por encima de la piramide ya que son los elementos más rápidos gracias a su definición tecnológica y esta dentro de la UAL. También se destaca que su costo es elevado y su capacidad es pequeña.

El segundo elemento, la memoria caché existe para mejorar la performance de la pc, a partir del guardado de datos de cierta información dentro de un proceso. Se la reconoce como una memoria intermedia entre el cpu y la memoria principal. La memoria cache es la segunda más rápida y de menor capacidad, aún así son muchisimo más grandes que los registros. Son más económicas que los registros también.

El tercer escalón es la memoria principal o memoria RAM, en forma de chips constituidos por circuitos semiconductores, donde se persiste la información de forma electrónica. Constituye un circuito un poco más lento que la memoria cache, pero si es bastante más barato.

Hasta acá llegamos al límite de la memoria interna. Todos estos elementos están interconectados a través de un bus.

Memoria externa (Persistencia de datos)

Lo que le sigue se consideran almacenamientos externos que se conectan con el cpu con la entrada y salida del mismo.

Los dispositivos de estado sólido son muy similares a la memoria ram en cuanto a lo tecnológico (la información se guarda bajo una configuración electrónica), su principal diferencia es que la información persiste y que su capacidad de almacenamiento es mucho mayor.

El disco magnético es el siguiente en jerarquía, siendo aún común en el mercado. Este es un medio con elementos mecánicos. Tienen capacidades muchísimas más grandes que lo anterior (no así respecto a un disco de estado sólido), el costo es menor y su velocidad es menor dada sus características tecnológicas. Tanto un disco sólido como uno magnético se utilizan para la transacción de datos que requerirán los softwares de las computadoras para realizar dicho proceso, no se suelen utilizar para persistir datos en muchísimo tiempo (en el caso hipotético de una empresa por ejemplo). Para esto se utilizan cintas / cartuchos magnéticos o dispositivos ópticos.

La cinta magnética es un medio de almacenamiento histórico de información que mantiene más usabilidad que los medios ópticos. Tiene capacidad similar a un dispositivo de estado sólido pero en cuanto a velocidad es muy lento ya que lee de forma secuencial la información. No se puede saltar a una posición particular de la cinta, si no que hay que recorrer toda la información anterior (como en un cassette).

● Jerarquía de memoria

- A medida que se baja de la pirámide:
 - Costo por bit decreciente
 - Capacidad creciente
 - Tiempo de acceso creciente
 - Frecuencia de acceso de la memoria por parte de procesador decreciente

Accesos en memoria

Entre los distintos tipos de memoria es importante destacar la diferencia entre un acceso secuencial y directo. El secuencial ya se mencionó y el directo es aquel que a partir de una dirección única para bloques, basada en su posición física se puede acceder directamente. El tiempo de acceso es variable.

Luego se encuentra el acceso aleatorio, donde cada posición direccionable en memoria se accede con la misma velocidad dado el mecanismo de direccionamiento cableado físicamente. El tiempo de acceso es constante e independiente de la secuencia de accesos anteriores. Lo aleatorio acá es que no importa cuando pidas acceso, la velocidad es la misma. Distinto es en un disco magnético, que depende de donde este la lectora o la degradación del material donde está grabado, por ejemplo.

Por último tengo accesos asociativos que a partir de una comparación de patrón de bits se accede o no, esto se ve en la memoria caché. En vez de buscar una dirección, busca un patrón de bits particular. La velocidad también es constante.

Parámetros de performance

Sabemos que para cualquier opción elegida voy a tener que realizar el proceso de acceder a la información. A esto se lo reconoce como tiempo de acceso o latencia. Esto no es igual en todos los elementos en memoria. En la ram se refiere al tiempo que se necesita para ejecutar una operación completa de lectura o escritura, siendo indistinto a que celda de memoria voy. Me interesa cuánto tarda en llegar y persistirse. Ahora, cuando uno se refiere a un disco rígido nos referimos a dejarle lista la lectora/grabadora para ser utilizada, pero no incluye el tiempo en si de la lectura / escritura:

○ Sistema de memoria

● Características

○ Parámetros de performance

- Tiempo de acceso (latencia)
 - Memorias de acceso aleatorio: tiempo necesario para hacer una operación de lectura o escritura
 - Memorias sin acceso aleatorio: tiempo necesario para posicionar el mecanismo de lectura/escritura en la posición deseada
- Tiempo de ciclo de memoria
 - Memorias de acceso aleatorio: tiempo de acceso más el tiempo adicional necesario para que una nueva operación pueda comenzar

Luego se mide como parámetro de performance el **tiempo de ciclo** en memoria. Este se lo define como el tiempo de latencia más el tiempo adicional que se requiere para estar lista la memoria para hacer otra operación.

Lo que le sigue en parámetro de performance es la tasa de transferencia:

- **Sistema de memoria**
 - Características
 - Parámetros de performance
 - Tasa de transferencia
 - Tasa con la cual los datos son transferidos dentro o fuera de la unidad de memoria
 - Memorias de acceso aleatorio: $1/\text{Tiempo de ciclo de memoria}$
 - Memorias sin acceso aleatorio:
$$T_n = T_A + n/R$$

donde

T_n = Tiempo promedio para leer o escribir n bits

T_A = Tiempo promedio de acceso

n = Número de bits

R = Tasa de transferencia, en bits por segundo (bps)

Esto refiere cuantas unidades de información por unidad de tiempo pueden ser transferidas desde o hacia esa memoria. Esto varia según si es una memoria de acceso aleatorio (ram) tengo que de forma genérica será $1/\text{tiempo de ciclo de memoria}$. Para cualquier otro tipo de memoria es la formula indicada: El tiempo promedio de escritura / lectura es el tiempo de acceso sumado a la cantidad de información que quiero guardar (n) dividido la tasa de transferencia (R).

Fisicamente, se separan las memorias de la siguiente manera:

Características

- **Tipos físicos**
 - Memorias semiconductoras (memoria principal y cache)
 - Memorias de superficie magnética (discos y cintas)
 - Memorias ópticas (medios ópticos)

La volatidad se define de como se comporta con la falta de electricidad. Luego tengo memorias que pueden estar en uno periférico o circuito interno (algunas memorias semiconductoras) que se definen como ROM (Read Only Memory).

Características

○ Características físicas

- Memorias volátiles: se pierde su contenido ante la falta de energía eléctrica (Ej. algunas memorias semiconductoras)
- Memorias no volátiles: no se necesita de energía eléctrica para mantener su contenido (Ej. memorias de superficie magnéticas y algunas memorias semiconductoras)
- Memorias de solo lectura: (ROM – Read Only Memory) no se puede borrar su contenido (Ej. algunas memorias semiconductoras)

Hay una categorización de todas las memorias semiconductoras:

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level	Nonvolatile	
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

Las memorias RAM tienen distintas características tecnológicas. Tengo rams estáticas y dinámicas que varía su electrónica más que nada.

Memoria caché

El concepto de caché responde al **Principio de localidad de referencia**. Esto ocurre para todo software. Cuando se ejecuta un programa, los accesos a la memoria ram a las instrucciones y datos tienden a agruparse, tienden a estar juntos por ejemplo en un loop, subrutina, tabla, vectores, etc:

● Jerarquía de memoria

- Principio de localidad de referencia
 - “Durante la ejecución de un programa, las referencias a memoria que hace el procesador tanto para instrucciones como datos tienden a estar agrupadas”
(Ej. loops, subrutinas, tablas, vectores)

La idea es guardarme cosas en la memoria caché ya que es más rápida que la memoria ram. La idea de guardarme ciertas porciones de memoria es para que la siguiente vez que yo ejecute lo que le sigue a dicho bloque, empiece justo desde ahí. Cuando hago un loop por ejemplo, voy momentaneamente a la caché para hacerlo más rápido. No así sucede cuando hago una bifurcación. Lo que pasa es que a nivel probabilístico tiendo a que esto de guardarme en bloques me sirva para acceder a la siguiente instrucción más rápido, además de darme más opciones de acceso en general.

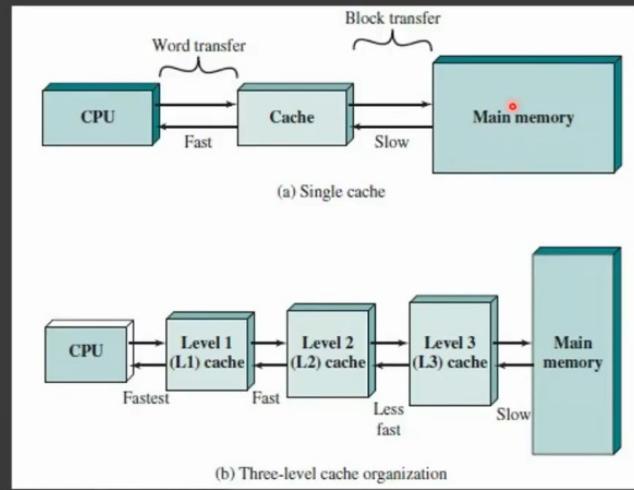
● Memoria Cache

- Memoria semiconductora más rápida (y costosa) que la principal
- Se ubica entre el procesador y la memoria principal
- Permite mejorar la performance general de acceso a memoria principal
- Contiene una copia de porciones de memoria principal

En general entonces, sabemos que la memoria cache es semiconductora al igual que la ram pero es más rápida y costosa que esta.

Ahora el procesador no le va a pedir directamente a la memoria principal, si no que primero se lo pide a la cache y esta es la que va a buscar a la ram lo que se solicita, es por eso que se encuentra entre dos elementos.

○ Memoria Cache

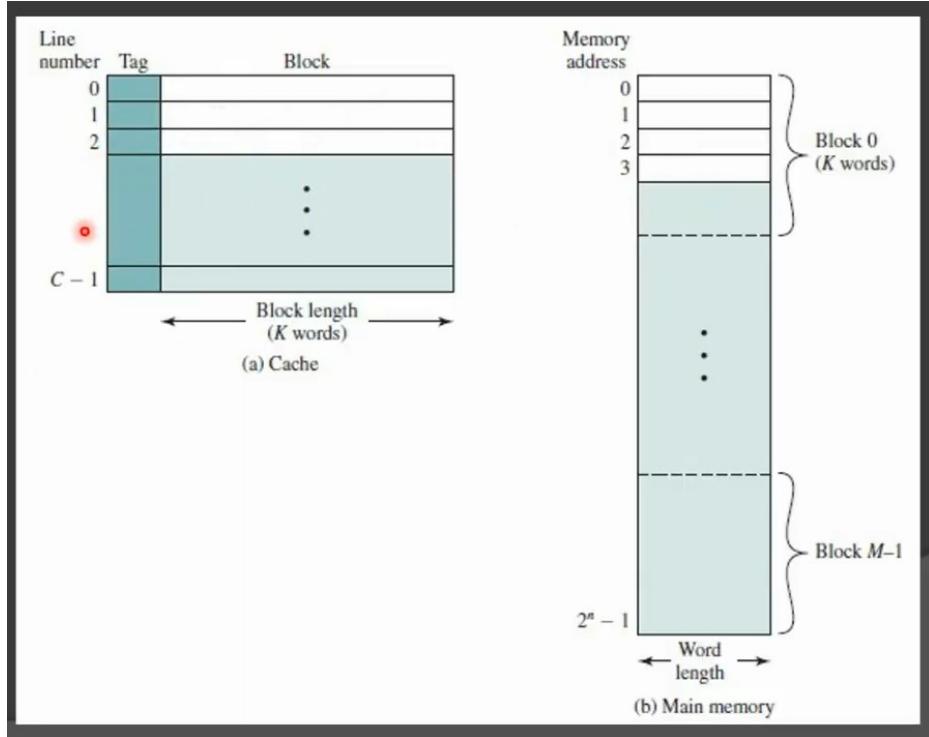


Este gráfico me dice que el cpu con la caché tiene una velocidad más rápida de transferencia y de la memoria principal a la caché es más lenta. A partir de este guardado de bloques es que ahorro tiempo. Cuanto menor sea el nivel, más rápida, menos capacidad y más costosa es.

○ Memoria Cache

- Cómo funciona
 - CPU trata de leer una palabra de la memoria principal
 - Se chequea primero si existe en la memoria cache.
 - Si es así se la entrega al CPU
 - Sino se lee un bloque de memoria principal (número fijo de palabras), se incorpora a la cache y la palabra buscada se entrega al CPU
 - Por el principio de localidad de referencia es probable que próximas palabras buscadas estén dentro del bloque de memoria subido a la cache

Mediante un sistema de referencia por tags es que se enlazan los bloques guardados en la cache referenciando a porciones que se encuentran en la memoria principal.



Cache

- m bloques llamados líneas
- Cada línea contiene:
 - K palabras
 - Tag (conjunto de bits para indicar qué bloque está almacenado, usualmente una porción de la dirección de memoria principal)
 - Bits de control (Ej. bit para indicar si la línea se modificó desde la última vez que se cargó en la cache)

Los bits de control me pueden advertir si hay una modificación de un dato dentro de la ram, referenciado en la cache, se modificó de modo que me provoque una inconsistencia.

Se suele subdividir la memoria cache para guardar datos y otra parte para instrucciones de modo que la memoria principal en si suele tener separado estas partes, por ende tiene sentido que así sea también en la caché.

Processor	Type	Year of Introduction	L1 Cache ^a	L2 Cache	L3 Cache
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA ^b	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6 × 32 kB/ 32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ server	2011	24 × 64 kB/ 128 kB	24 × 1.5 MB	24 MB L3 192 MB L4

Memoria principal

A la hora de como administrar la memoria RAM toma rol fundamental el sistema operativo. Este es un software que está presente en todos los ámbitos computacionales, salvo en excepciones como microprocesadores.

En si el sistema operativo es el software encargado de administrar los recursos del hardware, provee servicios y controla la ejecución de los programas.

Uno de los servicios principales es el **schedule de procesos** que monitorea todos los procesos en ejecución en ese momento en la memoria principal de la computadora. Lo que hace el sistema operativo es asignarle una porción de tiempo de cpu para cada uno de esos procesos. Se muestra bajo que criterio se administra el tiempo del cpu el sistema operativo.

Otro servicio es la administración de la memoria. El sistema operativo actua de gobierno ante el uso de la misma al ser un recurso escaso. El servicio de la administración de la memoria ayuda a entender como se interconecta el hardware para funcionar en su conjunto.

○ Administración de Memoria

- Sistema Operativo
 - “Software que administra los recursos del computador, provee servicios y controla la ejecución de otros programas”
 - Algunos servicios que provee
 - Schedule de procesos
 - Administración de memoria
 - Monitor
 - Parte residente del Sistema Operativo

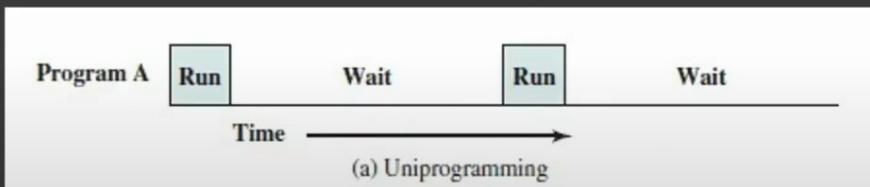
Uniprogramación

Hace referencia a aquellos cpus que tienen un unico software ejecutando sin necesidad que haya un sistema operativo por encima que lo gestione. Hay un unico proceso a la vez ejecutandose. No es el caso masivo pero es una solución para ciertos casos para la administracion de memoria.

Cuando hay un unico proceso sucede que toda la memoria esta empleada por este. En este caso el cpu alterna entre un tiempo de espera del proceso en cuestion hasta que vuelve a haber otra petición. El tiempo “ocioso” del cpu es un desperdicio de energia.

○ Uniprogramación

- Un solo proceso de usuario en ejecución a la vez
- La memoria de usuario está completamente disponible para ese único proceso
- Uso del procesador a lo largo del tiempo



- Run: Tiempo efectivo de uso del CPU
- Wait: Tiempo ocioso del CPU esperando E/S (*Idle time*)

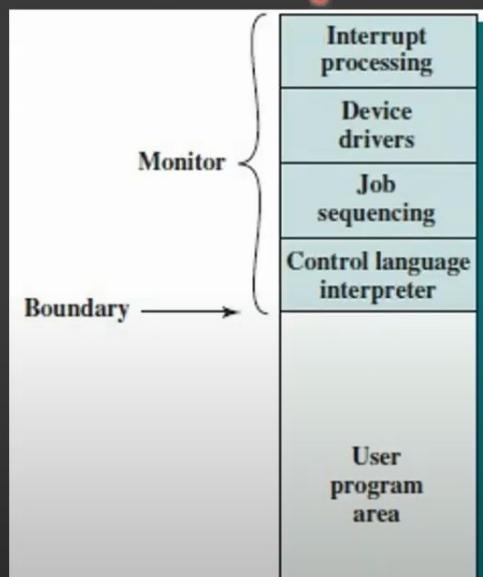
○ Administración de memoria simple

- Sistema con uniprogramación
- Se divide la memoria en dos partes
 - Monitor del S.O.
 - Programa en ejecución en ese momento
- Ventajas:
 - Simplicidad
- Desventajas:
 - Desperdicio de memoria
 - Desaprovechamiento de los recursos del computador
- Ej. MS-DOS, iPhone OS v1-3, IBM OS/PCP (Primary Control Program)

Hay una pequeña parte llamada monitor que el sistema operativo gestiona para su propia ejecución. Lo único que hace es subir y bajar el proceso cuando lo necesita.

○ Administración de memoria simple

- Memoria



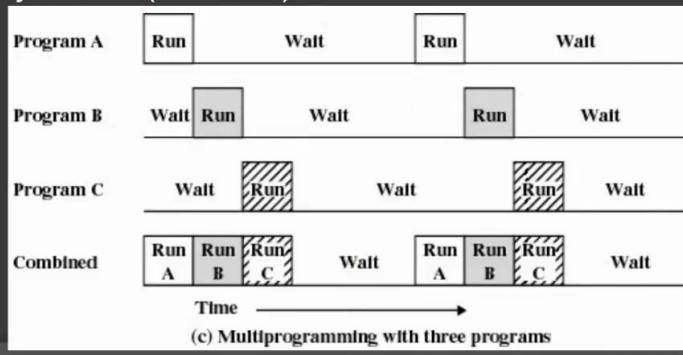
Multiprogramación

Este es el contexto universal, donde varios procesos de usuario se ejecutan a la vez. Sucede que hasta hay varios procesos de un mismo software, por ejemplo un navegador maneja los múltiples procesos.

En este contexto el sistema operativo debe gestionar el espacio que se tenga de ram. Cada vez que un proceso usa el cpu, el resto se encuentra en espera. Esa porción de tiempo que se le da al cpu para cada programa se le llama timeslice (porción de tiempo).

● Multiprogramación

- Varios procesos de usuario en ejecución a la vez
- Se divide la memoria de usuario entre los procesos en ejecución
- Se comparte el tiempo de procesador entre los procesos en ejecución (timeslice)



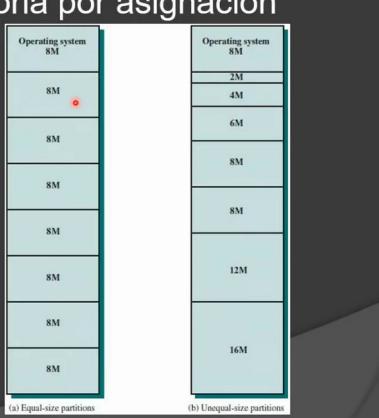
Un proceso puede finalizar por distintas condiciones. Una puede ser porque el usuario lo termino o el proceso en si llego a su fin. También puede terminar por un error o por una suspensión. Esta suspensión es cuando el sistema operativo lo “frizza”. Un proceso puede suspenderse porque ese lapso de tiempo que le dio el sistema operativo termino y para concluir el proceso hay que esperar a que el so le vuelva a dar el tiempo para terminarlo (termina el timeslice). Otra razón muy común es cuando el proceso inicia alguna operación de entrada o salida (cualquier interacción con un periférico). La capacidad de que un proceso pueda hacer esto son soluciones para maximizar el rendimiento del cpu.

Administración de memoria por asignación particionada

En estos contextos, la memoria ram se dividía en porciones iguales o de distinto tamaño pero siempre fijas. El sistema operativo a cada proceso le asigna un bloque. No se podía compartir con otros procesos las asignaciones, si había procesos que requerían menor tamaño quedaba sin usar una parte.

● Admin. de memoria por asignación particionada

- Particiones Fijas
 - Iguales
 - Distintas

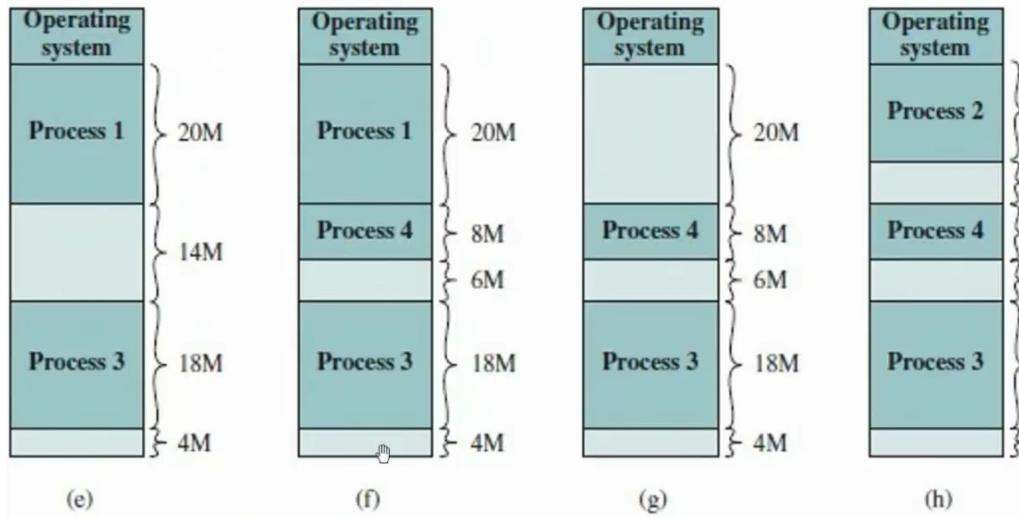


Esta era una forma relativamente sencilla de gestionar multiprocesos, como ventaja. Como desventaja había estos desperdicios de memoria en el sentido de una fragmentación interna (cuando te quedan partes parciales de un bloque sin usar) o cuando te quedaba memoria disponible pero no bloques del tamaño que se precisa para un proceso en particular.

- Administración de memoria por asignación particionada
 - Sistema con multiprogramación
 - La memoria de usuario se divide en particiones de tamaño fijo:
 - Iguales
 - Distintas
 - Ventajas:
 - Permite compartir la memoria entre varios procesos
 - Desventajas:
 - Desperdicio de memoria
 - Fragmentación interna (dentro de una partición)
 - Fragmentación externa (particiones no usadas)
 - Ej. IBM OS/MFT (Multiprogramming with a Fixed number of Tasks)

Asignación particionada reasignable

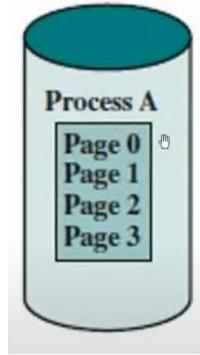
Esta en vez de asignar a los bloques tamaños fijos, se le asignaba a cada proceso la cantidad de memoria que requería. A medida que le voy asignando procesos a la memoria, va quedando ciertas partes libres. El problema estaba que a medida que sacaba o ponía procesos me quedan porciones sin usar de la memoria:



Mediante un recurso de hardware lo que se hacía es tomar todos estos pequeños bloques vacíos y unificarlos para poder aprovechar el espacio y poder alojar otro proceso potencial. Esta relocación de procesos consumía recursos y esa es una de las principales desventajas.

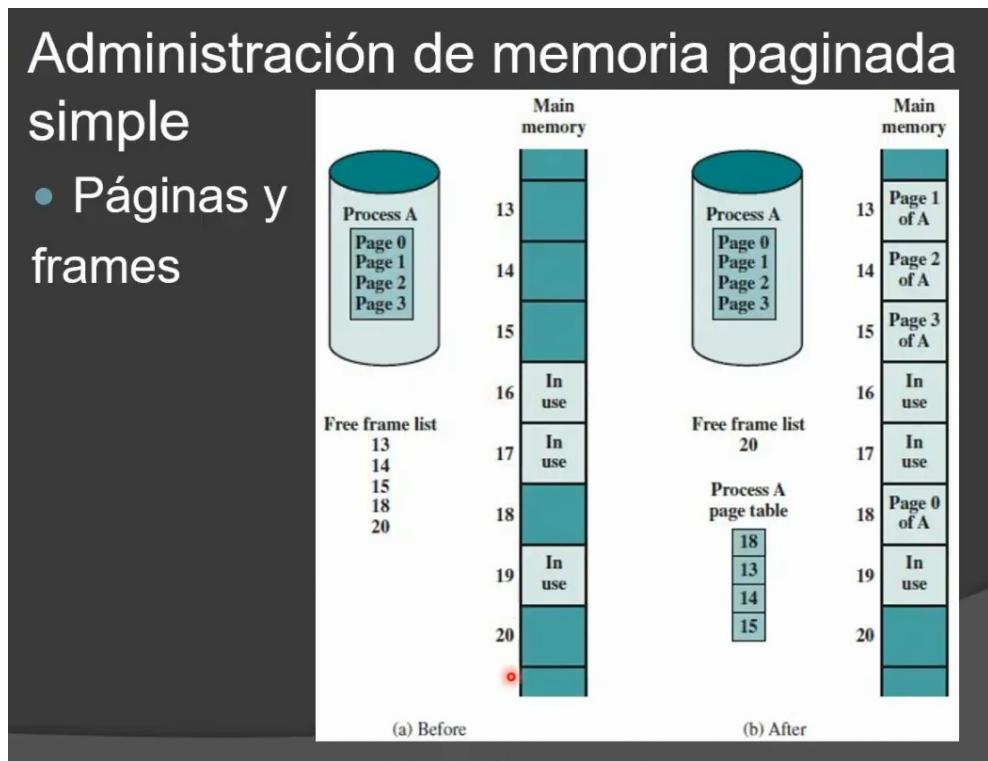
Administración de memoria paginada simple

Este es casi igual al que se usa hoy en día. Mediante esta forma de subdividir se obtiene mucha versatilidad a partir de, en forma lógica subdividir a cada proceso en partes iguales a las que llama pagina. Todo el espacio de direcciones se mapea en páginas:



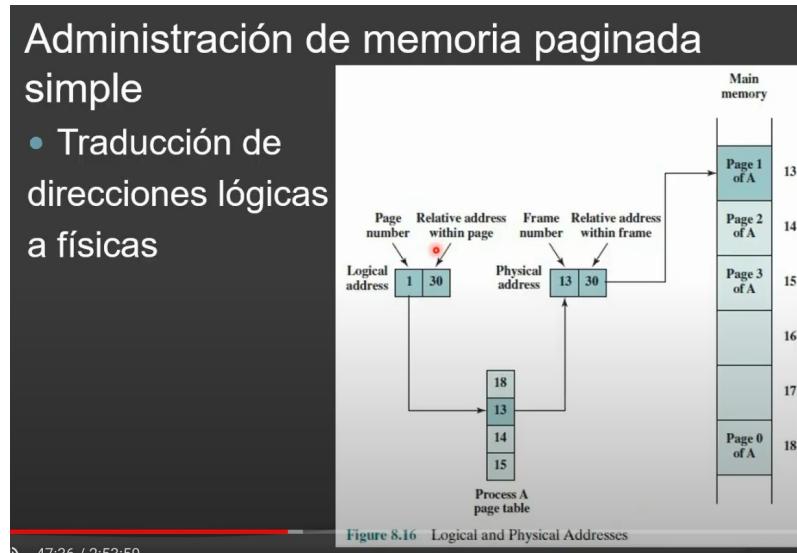
Por ejemplo, en arquitectura intel de 32 bits, cada página ocupa 4kb.

A su vez, la memoria ram tambien se subdivide en partes iguales pero se llaman frames. Allí se van subiendo cada una de las páginas del proceso. En vez de meterlo todo junto como un bloque se partitiona y se va metiendo de una forma más dinámica y versatil.



De esta manera el sistema operativo identifica dos estados: Libre y en uso. El so tiene una lista de los frames que estan libres que se pueden utilizar para subir paginas de los procesos. De esta forma los procesos no necesariamente estan todos juntos y ni siquiera estan en orden las paginas.

Para poder acceder a cada página distribuida por la memoria tengo que poder traducir la posición relativa lógica mediante la que yo programe a la dirección real donde se va a guardar en la memoria. Esto se hace a partir de un hardware dedicado:



A partir de un número de página y una dirección relativa desde esa página hacia la dirección que quiero traducir, como el sistema operativo sabe en qué frame se guarda (sabiendo el sistema operativo cuál es la dirección real de este frame) hay que realizar la traducción lógica a una traducción física. Se traduce la página y el offset en un frame y el offset. A cada página se le asigna un frame.

Administración de memoria paginada simple

- Sistema con multiprogramación
- Se divide el address space del proceso en partes iguales (páginas) (ej. IA-32 4KB c/u)
- Se divide la memoria principal en partes iguales (frames)
- Hay una tabla de páginas por proceso
- Hay una lista de frames disponibles
- Se cargan a memoria las páginas del proceso en los frames disponibles (no es necesario que sean contiguos)
- Las direcciones lógicas se ven como número de página y un offset
- Se traducen las direcciones lógicas en físicas (*address translation*) con soporte del hardware (MMU – Memory Management Unit)
- La paginación es transparente para el programador

Este modo de gestionar la memoria permite un uso más dinámico y eficiente y se evita la fragmentación interna y no existe fragmentación externa: cualquier frame libre puede usarse para cualquier página. Entre otras ventajas:

Administración de memoria paginada simple

- Ventajas:
 - Permite compartir la memoria entre varios procesos
 - Permite el uso no contiguo de la memoria
 - Minimiza la fragmentación interna (solo existe dentro de la última página de cada proceso)
 - Elimina la fragmentación externa
- Desventajas:
 - Se requiere subir todas las páginas del proceso a memoria
 - Se requieren estructuras de datos adicionales para mantener información de páginas y frames

La gran desventaja es la que se indica, hay que subir todas las páginas a memoria.

Administración de memoria paginada por demanda

Es la forma por defecto en la que hoy en dia se administra la memoria, salvo en algún nicho de mercado. El hardware debe estar construido para soportar esta forma de administración (debe tener el MMU, por ejemplo).

Lo que cambia a diferencia del anterior es que el sistema operativo no necesita que esten todas las páginas subidas a memoria. Se cargan algunas y quedan esperando las otras en una memoria secundaria.

Cuando tengo subida una página en específico a la memoria y quiero hacer una bifurcación hacia la siguiente página que no está en memoria, se acude a un evento que se llama **page fault (fallo de página)**. Este es un evento habitual y no necesariamente es un error. Lo que hace es disparar una **interrupción** (evento que ocurre en la computadora para que el cpu deje de hacer lo que está haciendo y resuelva una solicitud) por hardware. En este caso alerta de que se suba la página no subida en memoria. El cpu usa el mmu y a través de una rutina del sistema operativo que a través de tablas internas da cuenta de la página que tiene que subir, levanta la página que desea, cambiando el frame en el que está. Dicha página esta en la memoria virtual / memoria secundaria (disco rígido). Esta pequeña porción de disco rígido simula ser una extensión de la ram, solamente para guardar las páginas siguientes a la página cargada en la memoria principal. Este proceso implica hacer la traducción de dirección lógica a dirección física.

Si se llega a subir muchos procesos a la memoria principal puede ocurrir que no haya más frames libres en la memoria. Para resolver un pagefault ante esta situación el sistema operativo realiza un **page swapping** consistente en bajar páginas a memoria secundaria y reemplazarlas.

Cuando sucede que se ralentiza una pc/celular muy frecuentemente es porque se están cargando más procesos de los que soporta la memoria de dicho equipo y comienza a ser muy frecuente el swapping.

Puede llegar un momento que el cpu se use más para hacer swappings que las instrucciones en sí de máquina para ser ejecutadas. A este proceso se le denomina **Trashing**

Administración de memoria paginada por demanda (memoria virtual)

- Sistema con multiprogramación
- Solo se cargan a memoria principal las páginas necesarias para la ejecución de un proceso
- Cuando se quiere acceder a una posición de memoria de una página no cargada se produce un *page fault*
- El *page fault* dispara una interrupción por hardware (MMU) atendida por el sistema operativo
- El sistema operativo (*page fault handler*) levanta la página solicitada desde memoria secundaria (memoria virtual)
- Si no hay frames libres es necesario bajar páginas a memoria secundaria y reemplazarlas (*page swapping*)
- Algoritmos para reemplazo de páginas (por ejemplo FIFO, First In First Out o LRU, Least Recently Used)
- Thrashing: el CPU pasa más tiempo reemplazando páginas que ejecutando instrucciones

Administración de memoria paginada por demanda

- Ventajas:
 - No es necesario cargar todas las páginas de un proceso a la vez
 - Maximiza el uso de la memoria al permitir cargar más procesos a la vez
 - Un proceso puede ocupar más memoria de la efectivamente instalada en el computador
- Desventajas:
 - Mayor complejidad por la necesidad de implementar el reemplazo de páginas
- Ej. Windows 3.x en adelante, Linux

Administración de memoria por segmentación

Esto se usaba antes en Intel pero por retrocompatibilidad podría implementarse una arquitectura con esta lógica. Es similar a la paginación dado que el espacio de direcciones de los procesos se divide en segmentos al igual que la memoria ram. No obstante, la gestión es diferente.

En la segmentación uno como programador podía intervenir como se dividía los segmentos, que tamaños, que privilegios, etc. Hoy en día quedó en desuso ya que la paginación es más eficiente.

Los segmentos en general son mucho más grandes que las páginas y se podían definir segmentos de tamaño variable.

El sistema operativo también administraba a partir de una tabla el mapeo.

● Administración de memoria por segmentación

- Sistemas con multiprogramación
- Generalmente visible al programador
- La memoria del programa se ve como un conjunto de segmentos (múltiples espacios de direcciones)
- Los segmentos son de tamaño variable y dinámico
- El sistema operativo administra una tabla de segmentos por proceso
- Permite separar datos e instrucciones
- Permite dar privilegios y protección de memoria como por ej. lectura, escritura, ejecución. (segmentation faults como mecanismos de excepción de hardware para accesos indebidos)
- Las referencias a memoria se forman con un número de segmento y un offset dentro de él. Con ayuda de hardware (MMU – Memory Management Unit) se hacen las traducciones de las direcciones lógicas a físicas
- Se pueden usar para implementar memoria virtual (solo se suben a memoria física algunos segmentos por proceso)

● Administración de memoria por segmentación

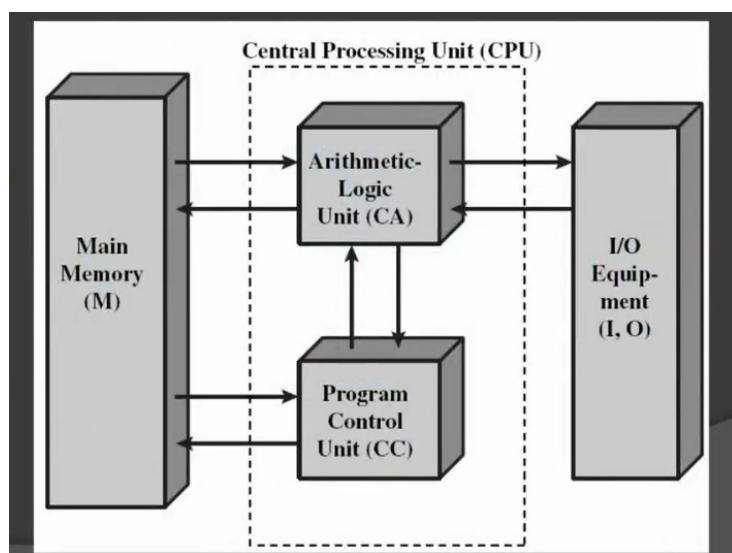
- Ventajas:
 - Simplifica el manejo de estructuras de datos con crecimiento
 - Permite compartir información entre procesos dentro de un segmento
 - Permite aplicar protección/privilegios sobre un segmento fácilmente
- Desventajas:
 - Fragmentación externa en la memoria principal por no poder alojar un segmento
 - Hardware más complejo que memoria paginada para la traducción de direcciones
- Ej. Burroughs Corporation B5000-B6500, IBM AS/400, Intel x86 (por compatibilidad hacia atrás)

○ Administración de memoria

- Distintas combinaciones (Ej. Intel Pentium)
 - Sin segmentación y sin paginación
 - Direcciones lógicas iguales a las físicas. No es útil para multiprogramación. Usado en controladores de alta performance
 - Paginación sin segmentación
 - La protección y administración de la memoria se hace a través de las páginas.
 - Ej. Berkeley UNIX
 - Segmentación sin paginación
 - La memoria se ve como una colección de espacios lógicos, con protección a nivel segmentos.
 - Segmentación con paginación
 - Segmentos para controlar el acceso a particiones de memoria.
 - Páginas para administrar la locación dentro de los segmentos
 - Ej. UNIX System V

Entrada / Salida

Siguiendo el esquema básico de Von Neumann, sabemos que los tres elementos principales en una computadora son la memoria, el cpu y la entrada/salida.

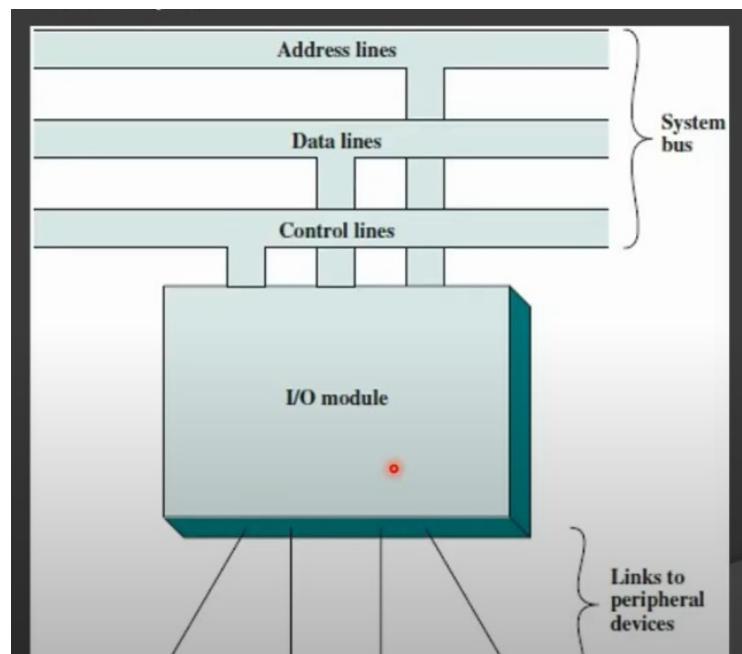


El modulo de entrada y salida permite la conexión de la cpu con los periféricos. Ya sea un teclado, un mouse, un touchpad o bien un disco rígido dentro de la misma carcasa. La razón de que este modulo funcione como intermediario ya que cada periférico tiene una forma de operar, una taza de transferencia distinta y formatos / tamaño de palabras distintos. Permite al cpu no tener que lidiar con toda esta complejidad.

Módulo de E/S

- ¿Qué hace?
 - Conecta a los periféricos con la CPU y la memoria a través del bus del sistema o switch central y permite la comunicación entre ellos
- ¿Por qué existe?
 - Amplia variedad de periféricos con distintos métodos de operación
 - La tasa de transferencia de los periféricos es generalmente mucho más lenta que la de la memoria y procesador
 - Los periféricos usan distintos formatos de datos y tamaños de palabra
- ¿Para qué sirve?
 - Oculta detalles de timing, formatos y electro mecánica de los dispositivos periféricos

El modulo de entrada / salida se conecta a partir de un bus con 3 carriles con información de direcciones, de datos y de control respectivamente. La interfaz en sí del modulo de entrada y salida luego se conecta con los distintos periféricos. La proporción entre cantidad de modulos y periféricos es variable. El modulo también se encarga de interpretar los estados de cada periférico para hacerse saber a la pc, a partir de un medio físico ya sea un cable usb o wifi, por ejemplo.



Se pueden diferenciar una interface interna y externa. Esta última refiere al enchufe en sí, mientras que la interface interna refiere a que cosas se transfieren por dentro del sistema en sí.

Funciones del modulo

En primera instancia este se encarga de controlar el flujo de información entre el CPU/memoria y los periféricos. Esto es algo que sucede de forma constante. Hay una gestión de la información de entrada para obtener una salida.

Las dos grandes funciones del modulo, por un lado es el intercambio de información con el cpu. La entrada/salida en primera instancia tiene la tarea de decodificar los comandos que le envía el CPU para luego mandarlos al periféricos. La decodificación en sí de comandos entonces es un comando genérico para traducir una solicitud a los periféricos.

Luego, obviamente se tiene que transmitir datos, por ejemplo cuando se envían los bytes que representan una imagen a una impresora.

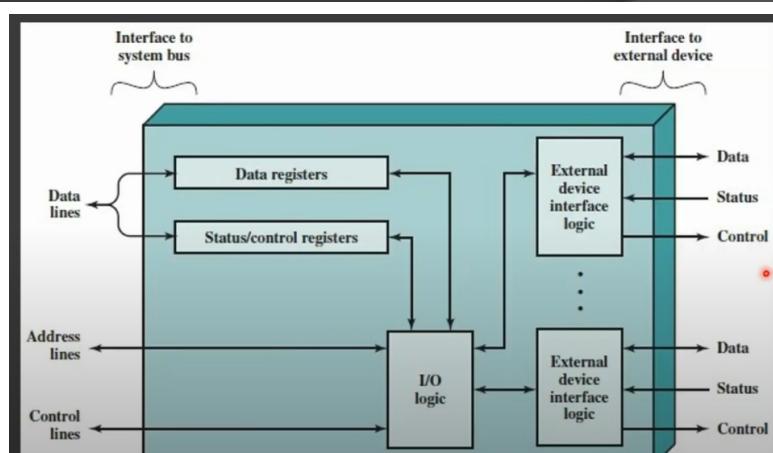
Todos los periféricos conectados a una computadora tienen un ID único lo que permite hacer un reconocimiento de direcciones al módulo, para decirle al cpu que tiene conectado y para que el mismo sepa cuando quiere hacer algo a que dispositivo con que ID mandarlo.

Luego tenemos la comunicación en sí del modulo con el dispositivo periférico. El modulo le puede decir que grabe datos en el disco, le envía comandos, información de estado, datos etc.

El buffering de datos refiere a que como los periféricos tiene distintas velocidades, ocurre dentro del modulo que tiene algunos registros para guardar información momentaneamente para después en el momento adecuado, transmitirlos.

La detección de errores viene enlazado a la detección y envío de información de Estado.

- **Funciones**
 - Control & Timing
 - Controla flujo de tráfico entre CPU/Memoria y periféricos
 - Comunicación con el procesador
 - Decodificación de comandos
 - Datos
 - Información de estado
 - Reconocimiento de direcciones
 - Comunicación con el dispositivo
 - Comandos
 - Información de estado
 - Datos
 - Buffering de datos
 - Detección de errores



Modos de operación de una entrada/salida

○ Módulo de E/S

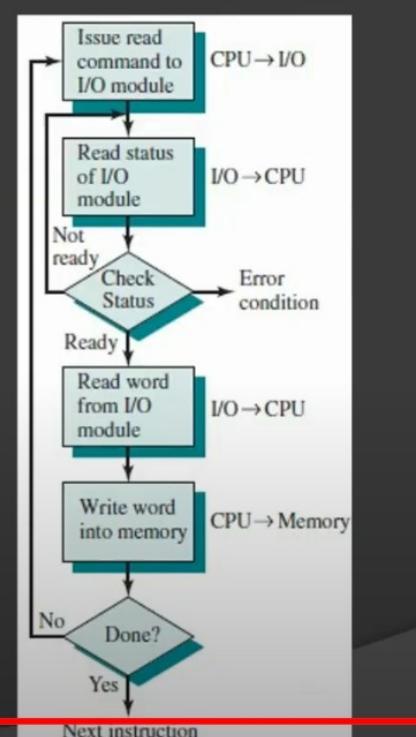
- Técnicas para operaciones de E/S
 - E/S Programada
 - E/S manejada por interrupciones
 - Acceso directo a memoria (DMA)

La entrada/salida programada se puede esquematizar como un envío de un programa de un comando genérico de lectura hacia un periférico determinado, que recibe el módulo de entrada/salida. Esto viaja a partir del bus de sistema. De forma secuencial, lo siguiente que ocurre es leer el Estado que devuelve al CPU el módulo de entrada/salida. Esta respuesta puede ser un rechazo del comando, que puede ser interpretado como un error o como la habilitación de un estado de espera que se desataba por el error en sí o el cambio a un estado de “listo”, que surge a partir de que el periférico tome la solicitud y devuelva una respuesta en forma de bytes al módulo de entrada/salida, el cual tendrá que ubicar en alguna dirección de memoria a partir de un buffer de lectura. Cuando esto se resuelve se pasa a la siguiente instrucción.

Lo que no es eficiente es que el CPU se quede esperando a que la operación de entrada/salida ocurra. Esto es un desperdicio enorme de potencial que tiene el mismo tal que en el tiempo que tarda una respuesta de periférico se pueden ejecutar cientos de miles de instrucciones.

○ Módulo de E/S

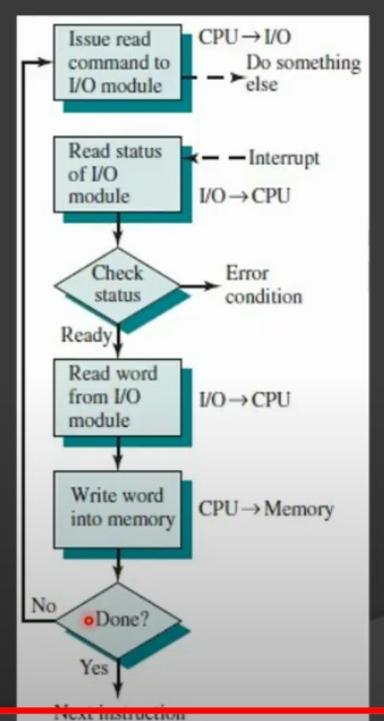
- E/S Programada



Lo que aparece para resolver esto último es el concepto de interrupción, a partir de la **E/S manejada por interrupciones**. Acá lo que cambia es que entre la acción de envío del comando al módulo se deja de esperar una respuesta y al CPU se le asigna otro proceso. Cuando se obtiene una respuesta el CPU la recibe a partir de las interrupciones, que permite hacer que el CPU deje de hacer lo que esté haciendo y evaluar qué hacer con ella. Cada instrucción de máquina que se ejecuta dedica un tiempo a ver si hay alguna instrucción de interrupción vigente. Cuando vuelve a leer la interrupción el CPU sucede lo mismo de chequear el estado para tirar un error o no. Cuando se leen los datos enviados por el periférico se escribe en memoria y cuando termina se pasa a la siguiente instrucción.

○ Módulo de E/S

- E/S manejada por interrupciones

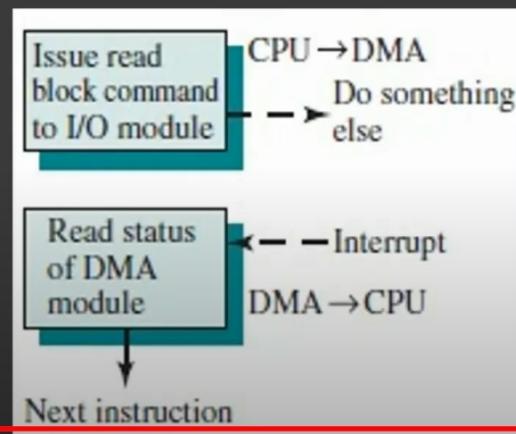


La inefficiencia acá es escribir la información que deja el módulo de entrada/salida en el CPU para que se escriba en la memoria principal. Hay que tener en cuenta que el CPU es un recurso escaso con una velocidad mucho mayor a cualquier otro componente. Cualquier cosa que el CPU haga que no sea manejo de instrucciones es un uso inefficiente.

Es por esto último que existen manejos más sofisticados para que el módulo tenga un acceso directo a la memoria para resolver lo de la entrada/salida sin tener que ocupar el CPU:

○ Módulo de E/S

- Acceso directo a memoria (DMA)

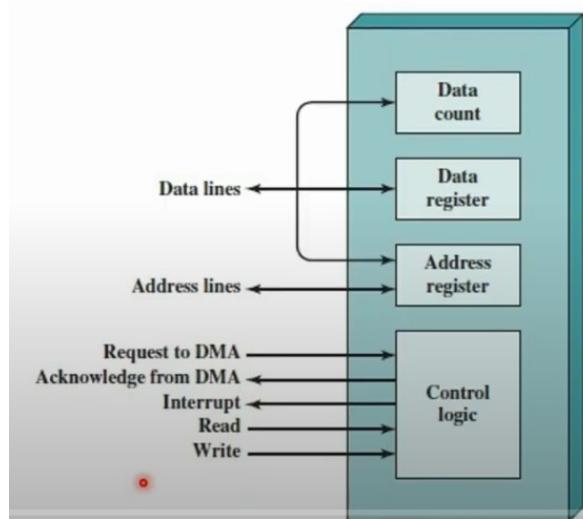


Acá el diagrama de bloque cambia rotundamente, de modo que el DMA (Direct Memory Access) es otro componente que se encarga de resolver de que la info que se lee, se escriba en la memoria. Y acá aparece el dilema de tener un componente adicional que puede interactuar con la memoria. Hasta este momento el único que interactuaba con la misma es el CPU.

Con la presencia del DMA, es posible leer bloques enteros de datos a leer y no palabra por palabra. Tiene la inteligencia para capturar el estado del entrada/salida, ver si está listo, hacer la escritura de la información a la memoria para finalmente emitir la interrupción al CPU.

Interacción del DMA con el CPU

Para desentenderse del tipo de acciones ligadas a la escritura en memoria, la relación que tiene el DMA con el CPU conlleva que el CPU envíe al DMA el tipo de operación (**Read o Write**) que se enviará al dispositivo conectado vía Línea de Control (Control Logic):



Luego se debe indicar al DMA con el cual se quiere interactuar, y esto lo hace a partir de su ID. Esto se hace vía Línea de Dirección (Address lines). Luego el CPU debe decirle al DMA con qué dirección en memoria va a interactuar, ya sea para leer los datos e enviarlos al dispositivo (**Read**) o escribir datos recibidos del dispositivo en memoria (**Write**). Esto era algo que resolvía el CPU y no se enviaba al modulo de entrada / salida, pero si es algo que se envía al DMA. Esta dirección inicial de memoria para Read O Write se envía vía Línea de Datos (Data lines) y lo guarda en Data register.

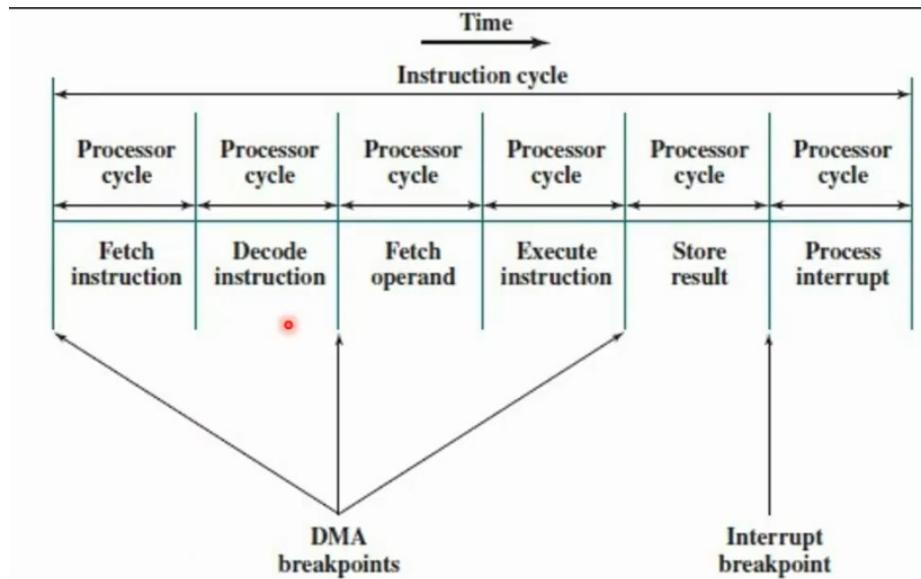
Por último, el CPU debe decirle cuantas palabras para READ o WRITE se utilizarán a partir de la dirección inicial y vía Línea de Datos se almacenará en el data count register.

El componente Control Logic es el componente que decodifica el comando y manda información tanto al modulo de entrada y salida para que este se lo envíe al periférico.

El DMA es el que tiene la capacidad de enviar la señal de Interrupt al CPU.

DMA – “Robo de ciclos”

Al tener un nuevo actor que accede a la memoria, cuando antes era únicamente el CPU el que podía hacerlo es necesario arbitrar cuando accede el DMA y cuando accede el CPU a dicha memoria. El robo de ciclo consiste en lapsos dentro de un ciclo de instrucción en los que el DMA le dice al CPU que va a acceder a la memoria y este tiene que esperar a que la acción del DMA finalice:



Hay tres momentos en específico en los que el DMA pueda acceder a la memoria, que son momentos claves en los que se necesitan ir a buscar algo a la memoria. Potencialmente en cada breakpoint el CPU requiera datos de la memoria. Cuando se hace el fetch de la instrucción, potencialmente haya que ir a buscar donde esta se encuentra alojada para luego traerla al registro de instrucción (RI) dentro del CPU.

Dentro de lo que es la ejecución de la instrucción en primera instancia tenemos la decodificación de la misma (interpretación de cada uno de los bits dentro del registro de instrucción para entender que es lo que se le pide hacer. Código de operación, operandos, etc). Antes del fetch del operando se abre un DMA breakpoint dada la potencialidad de que el operando se vaya a buscar a la memoria.

Finalmente, luego de que se haya ejecutado la instrucción tengo un breakpoint en el store del resultado, para por ejemplo guardar la suma de dos registros en memoria.

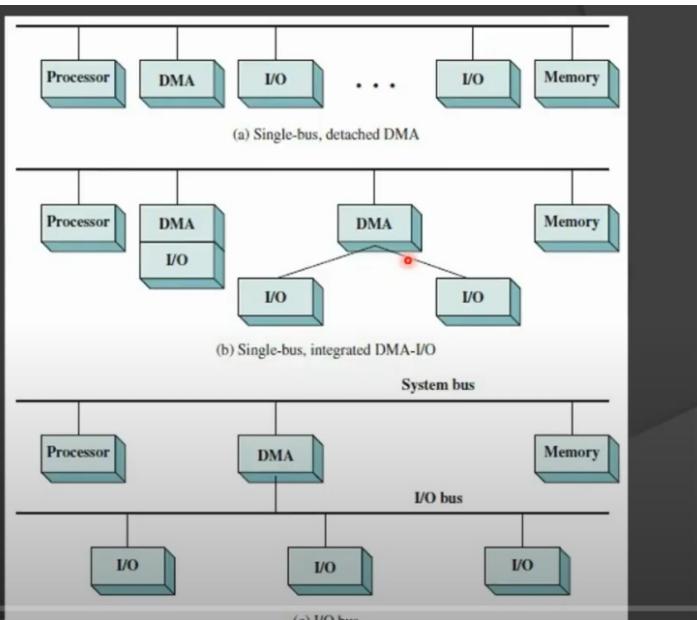
Luego lo que es el Interrupt breakpoint, es el momento en específico en el que el cpu se pone a escuchar si hay algún evento que lo quiera interrumpir. La interrupción se lee solo cuando se termino de ejecutar una instrucción. Si se intenta hacer una interrupción en el medio de la ejecución de una instrucción, esta interrupción queda “encolada” hasta que esta termine de ejecutarse.

Los esquemas de como los componentes se interconectan responden a distintas topologías de configuración, donde se entienden las líneas como los bus que interconectan a cada componente:

○ Módulo de E/S

- DMA –

Topologías de configuración



Canales y procesadores de Entrada y Salida

Ante grandes demandas de este tipo de interacciones como en servidores, aparecen los conceptos de canales y procesadores de entrada y salida. El concepto es como el del modulo de entrada y salida pero tienen una complejidad tecnológica mayor, dando más versatilidad.

Un canal tiene una CPU propia, de modo que ejecuta las instrucciones de entrada/salida sin ningún tipo de intervención del CPU principal de la máquina.

La diferencia entre un canal y un procesador es que este CPU secundario del canal sigue interactuando con la memoria principal. Tiene que leer las instrucciones guardadas allí. En el caso del procesador, estos tienen una memoria ram propia, donde se ejecutan las operaciones de entrada y salida, el propio cpu del procesador, yendo a buscar dentro de su propia memoria:

○ Módulo de E/S

- Canales y procesadores de E/S

- Canales

- Tienen la habilidad de ejecutar instrucciones de E/S
 - La CPU principal no ejecuta instrucciones de E/S
 - Las instrucciones de E/S se almacenan en memoria principal
 - La CPU le indica al canal de E/S que inicie un programa de canal
 - Dispositivo
 - Área de memoria para storage
 - Prioridad
 - Acciones ante errores

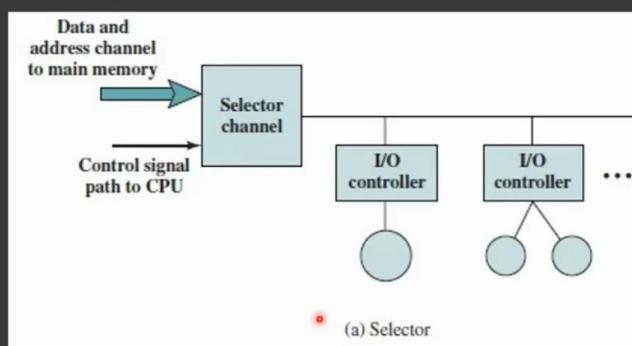
- Procesadores

- Agregan a los canales memoria propia en vez de usar la memoria principal

En caso de los canales, el CPU principal lo único que hace es indicar que se debe ejecutar una **operación de canal**, que indica información relevante para que este canal lo procese. Esto es el dispositivo, el área de memoria para storage, la prioridad (en contextos en el que haya miles de operaciones de entrada y salida al tiempo) y acciones ante errores.

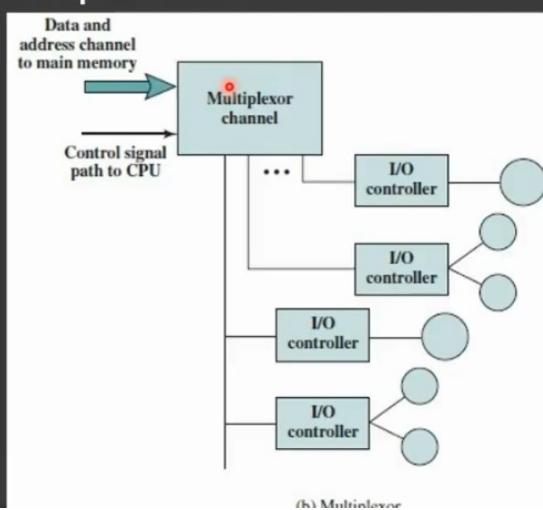
Los canales pueden operar de dos formas. La forma más simple es el canal selector, que es aquel que usa un dispositivo a la vez para operar. Se conecta con los distintos módulos de entrada y salida y puede operar con uno solo a la vez:

- Módulo de E/S
 - Canales y procesadores de E/S
 - Canales selectores



Luego están los multiplexores son aquellos que pueden trabajar con múltiples pedidos a la vez. La conectividad es en paralelo, de modo que puede recibir peticiones de todos los controladores a la vez:

- Módulo de E/S
 - Canales y procesadores de E/S
 - Canales multiplexores



Interrupciones

Son mecanismos mediante los cuales otros módulos pueden interrumpir el normal procesamiento del CPU. Los eventos en paralelo que suceden tienen que congeniar con el habitual procesamiento de instrucciones del CPU. Es mediante las interrupciones que este es capaz de registrar eventos de cualquier tipo y que dichos eventos sucedan al tiempo de los procesos que ya de por sí se están ejecutando.

● Interrupciones

- ¿Qué son?

“Mecanismos por los cuales otros módulos (E/S, memoria, etc.) interrumpen el normal procesamiento del CPU”

- ¿Para qué existen?

“Para mejorar la eficiencia de procesamiento de un computador”

- Clases de interrupciones

- Hardware
- Software

Es por esto que las interrupciones existen para mejorar la eficiencia del procesamiento de un computador.

En general hay dos tipos de interrupciones. Las más generales son aquellas que son producidas por algún elemento del hardware en particular. El otro tipo de interrupciones son las de software, aquellas que se producen porque algo ocurrió a nivel programación dentro de algún proceso que se está ejecutando

Las de hardware se suele enlazar con el concepto de asincronías tal que son eventos que no están alineados con el proceso que está corriendo, por lo contrario se hace de forma no sincronizada.

Otros componentes además de los componentes de entrada/salida que pueden generar las interrupciones son el reloj: Este se encuentra dedicado a cronometrar los ciclos de instrucción y dedicarle un tiempo a cada proceso. Por último tenemos las fallas de hardware, cuando algún periférico falla.

Luego para las interrupciones de Software se pueden identificar excepciones de programa: eventos que en general son sincrónicos, ya que ocurren dentro del proceso que el cpu atiende. Cuando de repente sucede algo que hace que se corte la normal ejecución del programa:

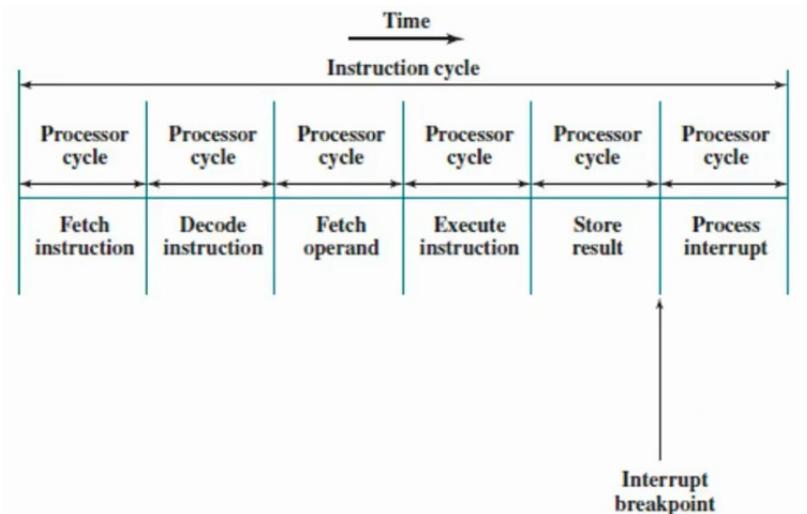
○ Interrupciones

- Clases de interrupciones
 - Hardware (asincrónicas)
 - E/S
 - Reloj (timer)
 - Fallas de hardware
 - Software
 - Excepciones de programa
 - División por cero
 - Acceso indebido a memoria
 - Overflow
 - Instrucción inválida
 - Instrucciones privilegiadas

Las instrucciones privilegiadas son aquellas excepciones generadas por el propio código del programa (como el swi en ARM para acceder a un servicio del sistema operativo).

Interrupciones dentro del ciclo de instrucción

Dentro del propio ciclo de instrucción, el momento en el cual se permite leer interrupciones es tras ejecutar la instrucción máquina completa (tras haber guardado el resultado). Luego entra el procesamiento de dicha interrupción. Estas interrupciones suceden millones de veces por segundo.



El comportamiento asincrónico de una interrupción puede verse de la siguiente manera:

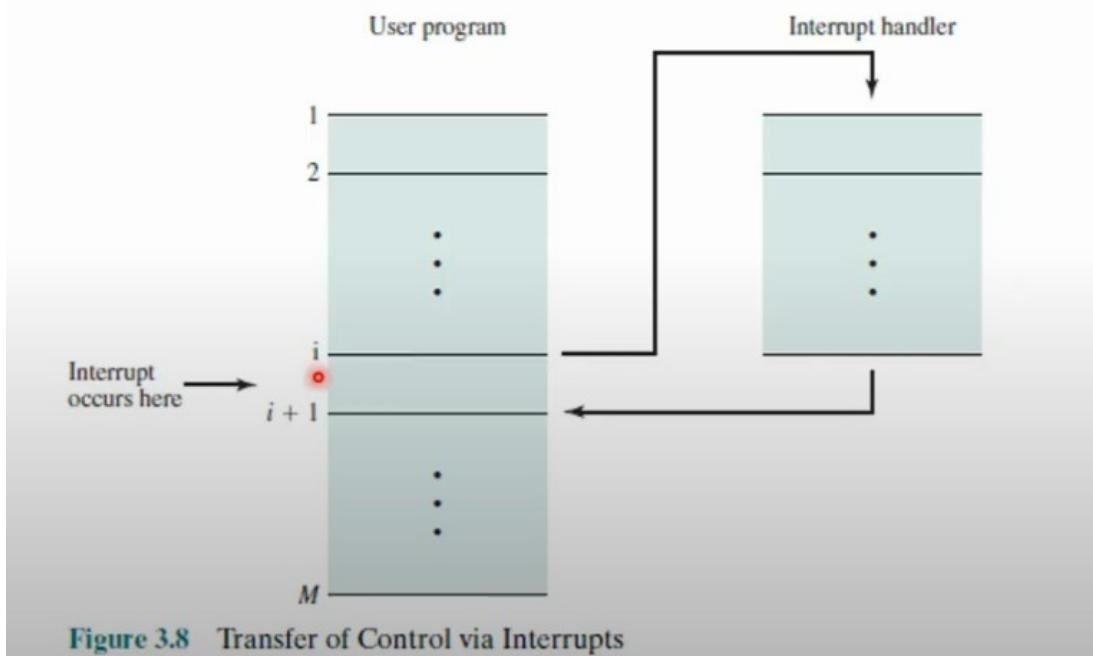


Figure 3.8 Transfer of Control via Interrupts

Si en el medio de la interrupción-ísema ocurre que un hardware lanza otro interrupción, esta se ejecutará entre la isema y la isema + 1. Atender una interrupción significa que algún servicio del sistema operativo escuchará dicha instrucción y hará algo.

Los handler son rutinas de software encargadas de atender las posibles interrupciones que pueden ocurrir de cada hardware que sea capas de lanzar una interrupción.

Primeramente en una interrupción, el elemento del hardware genera la interrupción. El CPU finaliza la ejecución de la instrucción actual para escuchar y confirma al modulo de entrada/salida que esa interrupción será atendida.

El CPU cuando recibe la interrupción resguarda la PSW (Program Status Word) que es un área de memoria donde se guarda para cada proceso información de control (estado de algunos bits, etc). Luego también resguarda el Program Counter, es decir, el valor de la próxima instrucción que se debería haber ejecutado del proceso actual. Estos datos tiene que guardarlo en una pila de control ya que se van a pisar con los que van a obtener de la interrupción.

Finalmente para efectivizar la interrupción se carga en el program counter, en el rpi, el valor de la rutina que va a entrar a partir de la interrupción:

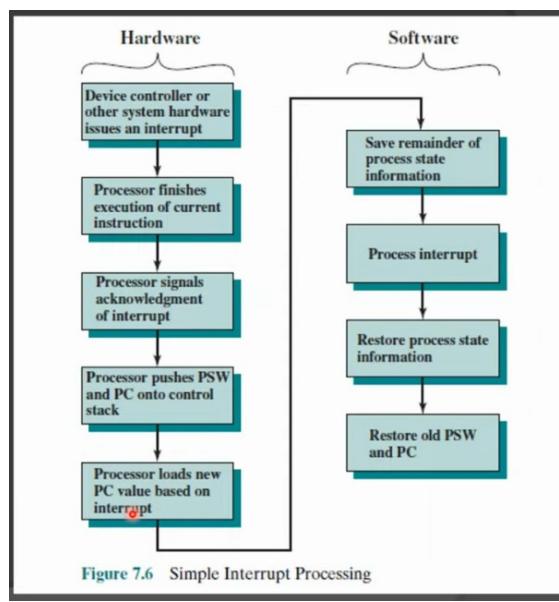
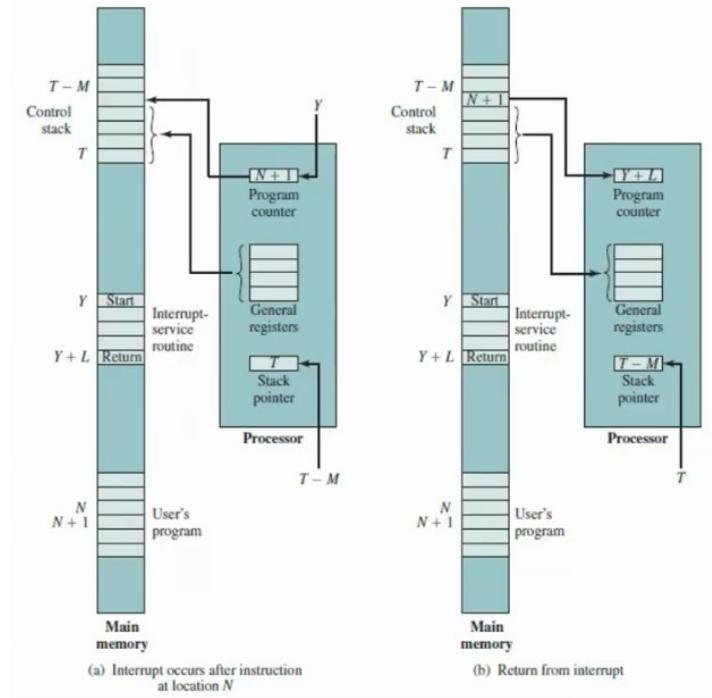


Figure 7.6 Simple Interrupt Processing

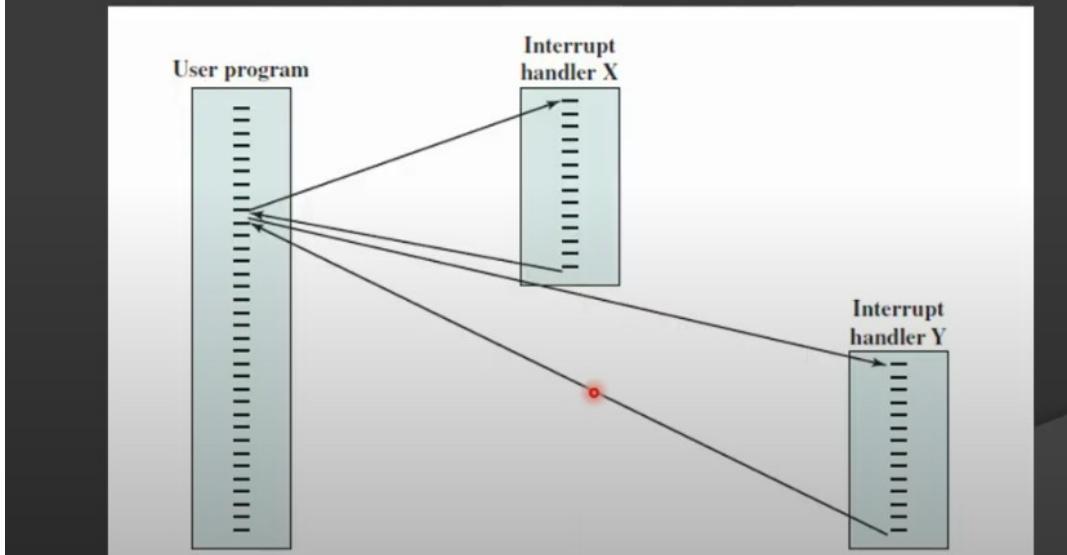
La rutina de software que atiende la interrupción lo primero que hace es resguardar el estado del proceso anterior en memoria interna tal que la interrupción potencialmente va a utilizar los registros que estaba usando este. Luego se ejecuta la interrupción, se restauran los valores anteriores de los registros y finalmente se busca el anterior PSW y PC de la pila de control.



Multiples interrupciones

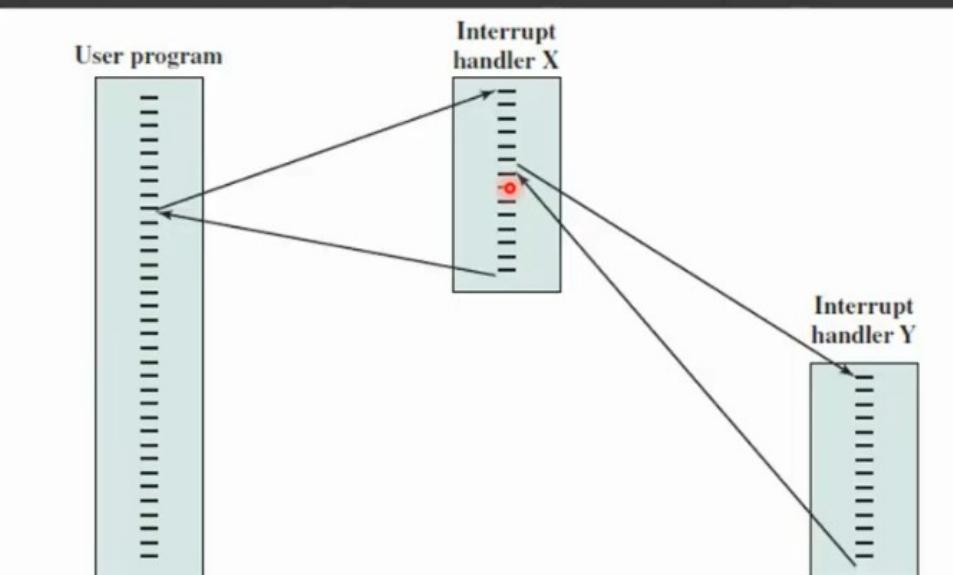
En la vida real suceden multiples interrupciones al mismo tiempo. Para atender eficientemente cada interrupción. Una opción es hacerlo de forma secuencial, pero esto trae algunos problemas en el sentido que puede haber interrupciones con más prioridad que se posterguen mucho por una cuestión de que llegaron antes interrupciones de menor prioridad.

○ Deshabilitar interrupciones (secuencia)



Lo que habitualmente ocurre con el hardware es que las interrupciones se manejan por prioridad, es decir, de forma anidada:

○ Priorizar interrupciones (anidadas)



(b) Nested interrupt processing

Figure 3.13 Transfer of Control with Multiple Interrupts

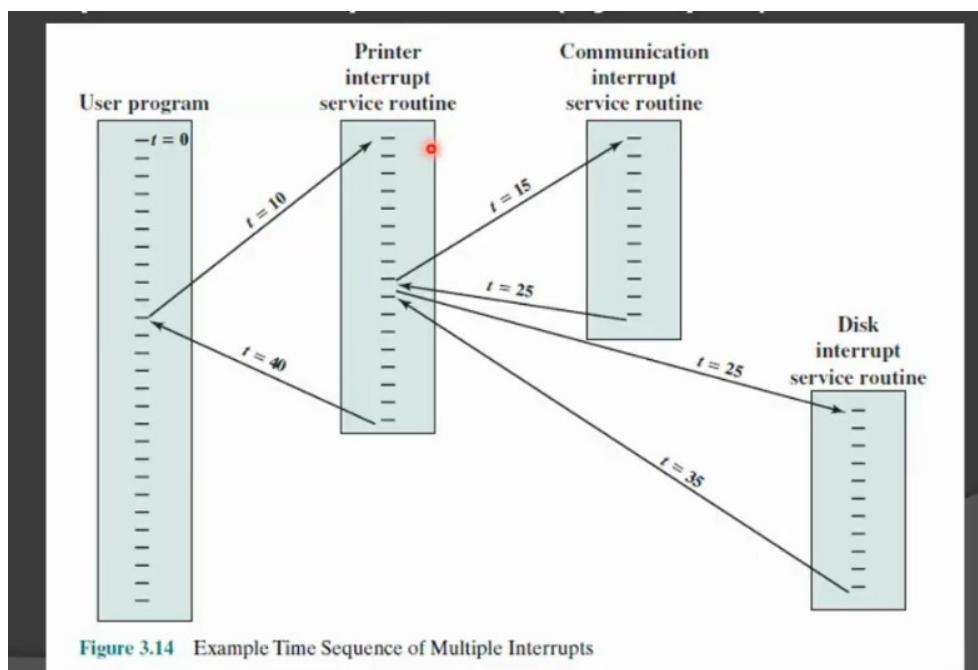
Múltiples interrupciones (ejemplo)

- Tres dispositivos de E/S

- Línea de comunicación (Prioridad 1)
- Disco (Prioridad 2)
- Impresora (Prioridad 3)

- Eventos

- T=10 – Interrupción de Impresora
- T=15 – Interrupción de línea de comunicación
- T=20 – Interrupción de disco



Procesador (CPU)

Se pueden diferenciar 2 tipos de arquitecturas principalmente: **CISC (Complex Instruction Set Computer)** y **RISC (Reduced Instruction Set Computer)**.

El CISC son los que predenominaron hasta fine los 70'.

El RISC es un diseño de arquitectura de computadores más minimizado, simple o elemental. Aparece a principios de los 80'. Todo el diseño de hoy en dia tiende a ser de este tipo. Esta arquitectura está orientada a Software.

A principios de los 70' no estaba comercializada tanto, ni había computación hogareña. El software se programaba para cada arquitectura en específico, hasta principio de los 70'.

En el diseño de procesadores se buscaba que estos tengan una arquitectura robusta que pueda resolver muchas cosas tal que no había tanto software desarrollado aún. Las instrucciones de máquina ocupaban muchos ciclos de reloj, se hacía mucho uso de la memoria, etc.

A partir de la evolución de la industria del software y la masificación del uso de la computación empieza a haber en el mercado la tendencia del RISC, de modo que el hardware se va simplificando más y más tal que la industria del software más avanzada venía a resolver un montón de las complejidades que antes necesitaba resolver el hardware. Los lenguajes de programación son cada vez más difundidos y resuelven más problemas, de tal manera que se advierte que es más performante una arquitectura minimalista. No obstante aún hay en el mercado arquitecturas como Intel x86 que por su retrocompatibilidad mantiene la arquitectura CISC.

CISC

No tiene muchos registros accesibles para un programador y por lo general son especializados. El conjunto de instrucciones es muy amplio. Tiene muchas instrucciones que trabajan directamente con memoria.

Lo que es la microarquitectura compleja muestra múltiples caminos de datos. Las instrucciones son complejas y pueden tardar más de un ciclo de reloj. Posee varios tipos de direccionamiento, datos y formatos de instrucción. Está orientado al hardware de modo que es el hardware el que principalmente resuelve las instrucciones.

RISC

Tiene muchos registros de uso general e intercambiables, a diferencia de CISC. A nivel de circuito no tiene ninguna restricción si quizás a nivel programación. Posee un set de instrucciones pequeño y se accede a la memoria solo a través de LOAD/STORE. La microarquitectura en hardware es simple porque las instrucciones son simples. Esto último indica que dichas instrucciones no durarán más de un ciclo de reloj. Posee pocos modos de direccionamiento, pocos tipos de datos y pocos formatos de instrucción. El formato de instrucción es fijo lo que es clave para la performance. La búsqueda de las instrucciones es más simple gracias a esto, el tamaño será siempre el mismo.

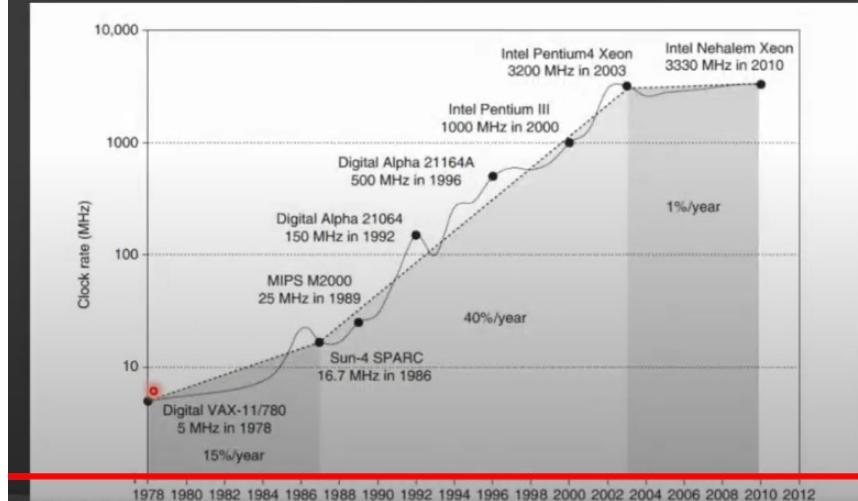
Está orientada al software y el hardware es mínimo. Aprovecha los lenguajes de programación cada vez más eficientes y complejos.

Ecuación de performance

MIPS = Frecuencia del reloj en Mhz (f) * Instrucciones por Ciclo (IPC) → Esto me da un ratio de performance. Esto calcula cuantos millones de instrucciones puede ejecutar el cpu por segundo.

Los primeros CPUS de un solo núcleo tenían una frecuencia de reloj baja y fue subiendo con el tiempo:

Velocidad de reloj (uniprocesadores)



En los últimos años el aumento de la frecuencia con los nuevos procesadores es cada vez más lento tal que el aumentar la frecuencia de un procesador implica una generación de calor que efectúa como limitante.

Ante estas limitaciones para maximizar el uso que se le puede dar al procesador nace el concepto de **parallelismo**. No se le exige al cpu que ejecute más instrucciones por frecuencia, si no que se lleva al CPU que ejecute más instrucciones por ciclo de reloj a partir de técnicas de procesamiento en paralelo:

Parallelismo

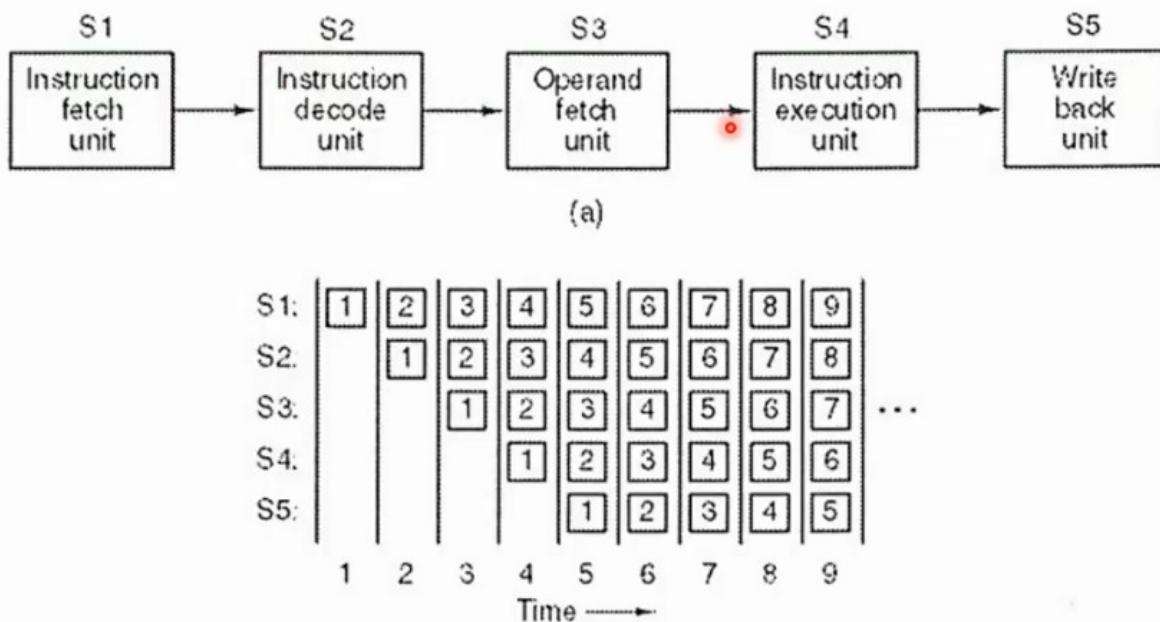
- Técnicas
 - A nivel instrucción
 - Pipelining
 - Dual pipelining
 - Superscalar
 - Multithreading
 - A nivel procesador
 - Procesadores paralelos de datos
 - Multiprocesadores
 - Multicomputadoras

Las primeras parallelizan procesos en dentro de un solo core. Luego las técnicas a nivel procesador se busca que el procesamiento se haga en varios procesadores al tiempo, obteniendo rendimientos altísimos con los multiprocesadores y/o multicomputadoras.

Técnicas a nivel instrucción

Pipelining

A partir de un CPU, el mismo trata de ejecutar en paralelo más de una instrucción a partir del solapamiento de ejecuciones de instrucción, dividiendo a los ciclos de instrucción en etapas:



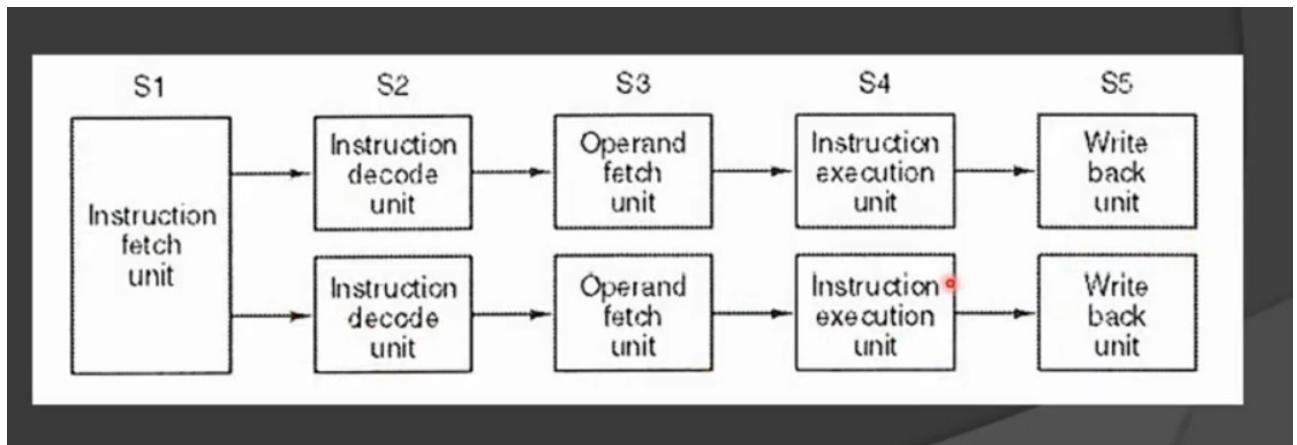
A medida que pasan las instrucciones a las siguientes unidades de máquina, otras se van procesando en las anteriores. Por ciclo de reloj sigue ejecutando una a la vez.

Uno de los problemas de ir ejecutando instrucciones de esta manera, teniendo en distintas secciones del cpu a cada instrucción, puede ser si alguna de estas instrucciones es una bifurcación, tal que dicha bifurcación me llevará a otras instrucciones y no se ejecutarían en el orden en el que estaban encoladas las demás instrucciones.

También puede suceder que una instrucción modifique algún operando de una instrucción por detrás procesándose. Para evitar esto se debe hacer un **control de dependencia** entre las instrucciones, ya sea a nivel de hardware o a nivel de software (compilador / hardware). Para solucionar esto dicho control de dependencia lo que hace para dos instrucciones donde se comparta un mismo operando es dejar un “tiempo muerto” (Not Operation) para que la instrucción termine de ejecutarse y de esta forma la que le sigue antes obtiene el valor esperado.

Dual pipelining

A partir del diseño del pipeline, algunos fabricantes implementan el Dual Pipelining. Acá si hay una genuina ejecución en paralelo de dos instrucciones.



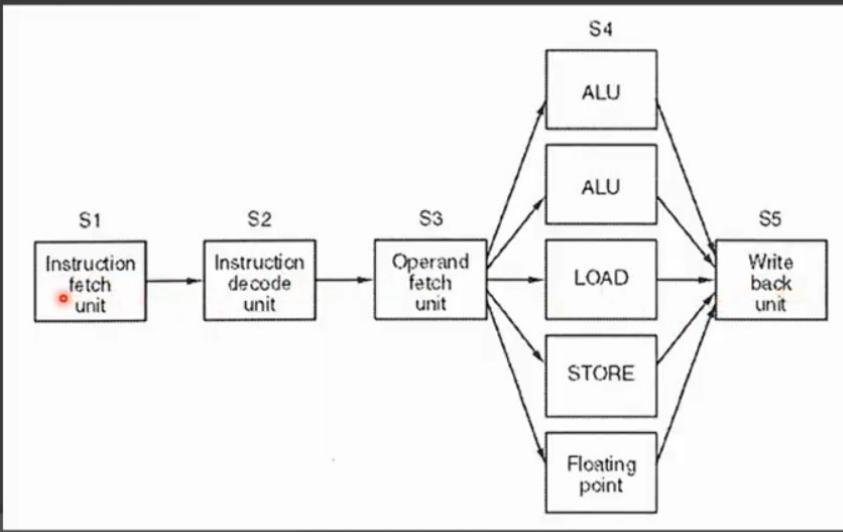
El acceso a la memoria es único pero el resto se paralleliza, ejecutando dos instrucciones por ciclo de reloj. Esto implica lugar físico del CPU, un costo de fabricación y generación de calor, no se escala más la cantidad de Pipelines por estos limitantes. Acá se requiere capacidad de ejecución de dos instrucciones al tiempo.

Superscalar (Múltiples unidades funcionales)

Esta se da a nivel instrucción y es complementaria (al igual que todas las otras) a las demás técnicas. Es decir, pueden combinarse entre sí, no son excluyentes.

Esta técnica genera un paralelismo en una de las etapas, la más lenta, se busca parallelizar en múltiples unidades funcionales (múltiples circuitos) para que puedan ejecutar varias instrucciones en paralelo y mejorar la performance:

- Superescalar



Se paraleliza la etapa 4 (la de ejecución) al ser la más lenta. Cuando no se tiene esta técnica hay un “cuello de botella” en esta etapa tal que para seguir procesando instrucciones hay que terminar de ejecutar la que estaba delante y suele tardar mucho en esta etapa.

Estas múltiples unidades funcionales (piezas de hardware) se dedican a ejecutar una acción en particular. Por ejemplo las ALU (Arithmetical Logic Unit) se dedican a hacer operaciones lógicas y matemáticas. Por ende, cuando entra más de una instrucción que se requiera una ALU, ambas se pueden ejecutar al tiempo tal que dispongo de dos. Si tuviera una instrucción luego para hacer un LOAD o STORE, se puede hacer tal que va a esa unidad libre. Suelen tener entre 3 y 6 de estas unidades.

○ Procesador

- Paralelismo

- A nivel instrucción

- Superscalar (múltiples unidades funcionales)
 - Ejecuta más de una instrucción por ciclo de reloj
 - N-way / N-issue (N entre 3 y 6)
 - Ej. Intel Core

Hardware multithreading / multiframe

Acá es cuando aparecen los famosos “hilos” (thread) y “núcleos” (core). Acá cada core paralleliza la ejecución de una instrucción en hilos diferentes. Cuando un hilo se frena por alguna causa, la instrucción se sigue ejecutando sobre otro hilo. De esta forma se maximiza el uso del CPU y este no se queda esperando a que finalice una determinada ejecución.

Un hilo es una línea de ejecución con cierta autonomía y dependencia. Cada hilo tiene un program counter (el valor del RPI). Cada hilo puede dejar ejecutando en una posición del programa distinta, es decir en un RPI distinto. La autonomía la consigue de tener registros de instrucciones propios para no hacer llamados a la memoria y una pila propia para hacer stacking de contextos.

Lo que comparten todos los hilos es el mismo espacio de direcciones. Los procesos se parallelizan pero todos usan la misma memoria. No usan una memoria independiente. Por eso se los conoce como procesos “livianos”.

Un proceso pesado es el proceso en sí tal que este tiene su propio espacio de direcciones (único) y un estado gestionado por el S.O. El cambio de contexto entre procesos es una gestión pesada, el cambio de procesos entre hilos es una gestión liviana:

○ A nivel instrucción

- Hardware multithreading
 - Busca incrementar el uso del CPU intercambiando la ejecución entre threads (hilos de ejecución) cuando uno está frenado por alguna causa
 - Thread: Contiene un PC, un conjunto de registros y la pila (stack). Comparten un mismo address space. Se los conoce como “lightweight processes”
 - Proceso: Puede tener uno o más threads, contiene un address space y un estado gestionado por el S.O.
 - El cambio de contexto entre threads es “liviano” (en un mismo ciclo de reloj) en comparación con los procesos, que requieren del S.O.

El hardware multihilo puede operar bajo varias técnicas.

Fine-Grained multitherading: intercambiando el uso del proceso entre hilos luego de la ejecución de cada instrucción. Es decir hace un barrido de los mismos.

Coarse-grained multithreading: El intercambio de hilos solo ocurre si hay algún evento significativo. Algo que frene la ejecución de un hilo en particular y valga la pena hacer el cambio de contexto. Esto sucede por ejemplo en el page fault (querer acceder a una página en memoria no mapeada, lo que dispara dicho evento y genera una interrupción). Cuando se llega a un proceso como este en un hilo, el procesador cambia a otro hilo para ejecutar otra instrucción de máquina.

El uso de esta técnica también puede verse presente cuando hay un “cache miss”. Cuando el CPU le pide a la caché determinada palabra y no la encuentra, esta memoria cache tiene que ir a pedirla a otro nivel de cache o a la memoria ram. Como esto implica quedarse esperando, se puede resolver que no se quede esperando pasando a otro hilo. Este tipo de técnicas está más orientada a servidores.

Técnicas a nivel procesador

Acá se paraleliza el uso de multiples cores, multiples cpus o multiples computadoras.

Procesadores paralelos de datos

SIMD

Son arquitecturas que trabajan una única instrucción de máquina con multiples procesadores. De ahí viene la sigla SIMD – Single Instruction Multiple Data. Son un subconjunto de instrucciones dentro de la ISA. En estos casos el CPU tiene múltiples core y una única unidad de control (CPU). Los multiples nucleos me permiten paralelizar operaciones.

Hay dos formas de operar, una es esta de tener multiples procesadores que ejecutan la misma secuencia de pasos sobre un conjunto diferente de datos. Esto sirve por ejemplo para hacer la misma operación para distintos operandos, cada core realiza la misma operación pero varia los elementos con los que trabaja. Esto es común en los procesadores dedicados para la gráfica. El calculo gráfico implica hacer muchas operaciones aritmeticas con muchos datos en paralelo.

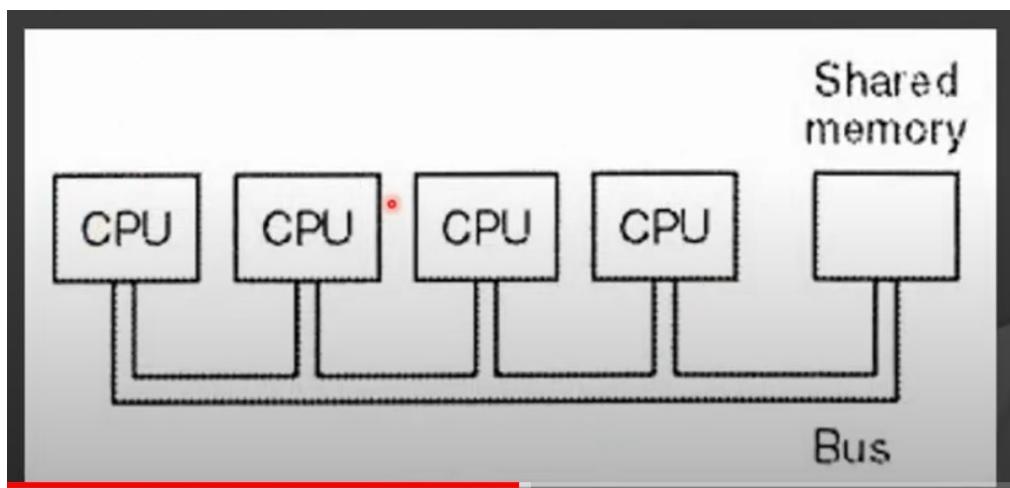
La otra técnica también se tiene una única instrucción para multiples datos, pero se cambia como se implementa en el hardware. Acá hay un registro vectorial: Un conjunto de registros que trabajan como un único bloque y hay una instrucción vectorial para trabajar con ese bloque. Esto opera bajo la técnica SSE (Streaming SIMD Extension).

MIMD

(Multiple Instruction Multiple Data)

Multiprocesadores: Esta técnica utiliza CPUs diferentes, completos y trabajando en paralelo. Estos CPUs están fuertemente acoplados tal que siguen utilizando una memoria compartida.

Dentro de las posibles implementaciones de multiprocesadores se puede tener un único BUS y una memoria compartida centralizada (Intel Core I7):



- Paralelismo

- A nivel procesador

- Multiprocesadores

- Múltiples CPUs que comparten memoria común

- MIMD (Multiple Instruction Multiple data)

- CPUs fuertemente acoplados

- Diferentes implementaciones

- Single bus y memoria compartida (centralizada) (UMA – Uniform memory access) (SMP – Symmetric multiprocessor)

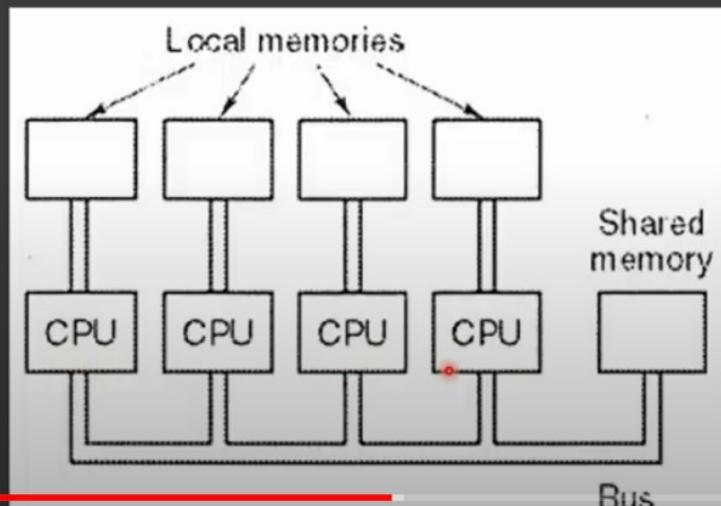
- Ej. Intel Core i7

- CPUs con memoria local y memoria compartida (NUMA – non-uniform memory access)

- Ej. BBN Butterfly, SGI Origin 2000, Compaq AlphaServer GS320, Intel Itanium 2

Después tenés implementaciones donde hay una mezcla de memoria local con memoria compartida (más común en servidores):

- CPUs con memoria local y memoria compartida



Multicomputadores

Ahora no solo se tienen CPUs independientes, si no que se tienen PCs interconectadas con su propia memoria local y se integran mediante mensajería entre computadoras. Así funcionan las supercomputadoras. Es una técnica de MIMD

- Paralelismo
 - A nivel procesador
 - Multiprocesadores
 - Múltiples CPUs que comparten memoria común
 - MIMD (Multiple Instruction Multiple data)
 - CPUs fuertemente acoplados
 - Diferentes implementaciones
 - Single bus y memoria compartida (centralizada) (UMA – Uniform memory access) (SMP – Symmetric multiprocessor)
 - Ej. Intel Core i7
 - CPUs con memoria local y memoria compartida (NUMA – non-uniform memory access)
 - Ej. BBN Butterfly, SGI Origin 2000, Compaq AlphaServer GS320, Intel Itanium 2

Primera pasada:

- **Lectura y análisis del código fuente:** El ensamblador lee el programa línea por línea, analizando la sintaxis y estructura de cada instrucción.
- **Construcción de la tabla de símbolos:** Durante esta fase, el ensamblador identifica y registra todas las etiquetas (símbolos) definidas en el código. Cada etiqueta se asocia con una dirección de memoria relativa, que indica su posición dentro del programa. Esta dirección es relativa al punto de inicio del programa y se calcula acumulando el tamaño de las instrucciones anteriores.
- **Cálculo de direcciones:** Aunque en esta etapa no se genera el código objeto final, el ensamblador determina la longitud de cada instrucción para calcular las direcciones relativas de las etiquetas y las instrucciones subsiguientes. Esto es esencial para que, en la segunda pasada, se puedan resolver correctamente las referencias a estas etiquetas.

Segunda pasada:

- **Generación del código objeto:** Con la tabla de símbolos completa y las direcciones relativas conocidas, el ensamblador procede a traducir cada instrucción del código fuente en su equivalente en código máquina, asignando las direcciones de memoria correspondientes.
- **Resolución de referencias a etiquetas:** Cuando una instrucción hace referencia a una etiqueta (por ejemplo, en una instrucción de salto), el ensamblador utiliza la tabla de símbolos para obtener la dirección de memoria relativa asociada a esa etiqueta y la inserta en el lugar correspondiente dentro del código objeto.

75.03 & 95.57 Organización del Computador

U4 – LENGUAJE ENSAMBLADOR

U4 – Lenguaje ensamblador

○ Introducción

- Lenguaje de máquina / código máquina
 - Definición: “Representación binaria de un programa de computadora el cual es leído e interpretado por el computador. Consiste en una secuencia de instrucciones de máquina”
- Lenguaje ensamblador
 - Definición: “Representación simbólica del lenguaje de máquina de un procesador específico.”

U4 – Lenguaje ensamblador

○ Lenguaje ensamblador

- Transición entre lenguaje de máquina y ensamblador.

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Figure 13.13 Computation of the Formula $N = I + J + K$

U4 – Lenguaje ensamblador

- Lenguaje ensamblador
 - ¿Por qué usarlo aún hoy?
 - Debugging y verificación
 - Desarrollar compiladores
 - Sistemas embebidos
 - Drivers de hardware y código de sistema
 - Acceder a instrucciones no disponibles en un lenguaje de alto nivel
 - Código automodificable
 - Optimizar código en tamaño
 - Optimizar código en velocidad
 - Biblioteca de funciones

U4 – Lenguaje ensamblador

- Lenguaje ensamblador
 - Elementos que lo componen
 - Etiquetas
 - Mnemónicos
 - Operandos
 - Comentarios

U4 – Lenguaje ensamblador

- Lenguaje ensamblador
 - Tipos de sentencias
 - Instrucciones
 - Directivas (pseudoinstrucciones)
 - Macroinstrucciones

U4 – Lenguaje ensamblador

- Lenguaje ensamblador
 - Ejemplo Intel x86

```
global          _main
section         .data
    num1 dd 5
    num2 dd 6
section         .bss
    resMay resd 1
    resMen resd 1
section         .text
_main:
    mov     eax, [num1]
    mov     ebx, [num2]
    add     eax, ebx
    cmp     eax, 10

    jg      mayor
    mov     [resMen], eax

    jmp     fin
mayor:
    mov     [resMay], eax
fin:
    ret
```

U4 – Lenguaje ensamblador

- Lenguaje ensamblador
 - Ejemplo ARM

```
AREA      ARMex, CODE, READONLY
                  ; Name this block of code ARMex
ENTRY      start          ; Mark first instruction to execute
start      MOV   r0, #10       ; Set up parameters
           MOV   r1, #3
           ADD   r0, r0, r1    ; r0 = r0 + r1
stop       MOV   r0, #0x18     ; angel_SWIreason_ReportException
           LDR   r1, =0x20026   ; ADP_Stopped_ApplicationExit
           SVC   #0x123456      ; ARM semihosting (formerly SWI)
           END
                  ; Mark end of file
```

U4 - Lenguaje ensamblador

- Lenguaje ensamblador
 - Ejemplo IBM Mainframe

```
*****  
* Copyright 2005 Automated Software Tools Corporation          *  
* This source code is part of z390 assembler/emulator package   *  
* The z390 package is distributed under GNU general public license *  
* Author - Don Higgins                                         *  
* Date - 09/30/05                                              *  
*****  
  
TESTDCBB SUBENTRY  
        WTO    'TESTDCBB TEST USE OF DCBREC WITH READ/WRITE'  
        OPEN   (SYSUT1,(INPUT),SYSUT2,(OUTPUT),SYSOUT,(OUTPUT))  
LOOP    EQU    *  
        READ   DECB1,SF,SYSUT1  
        CHECK  DECB1  
        AP     PTOT,=P'1'  
        MVC    DTOT,=X'40202020'  
        ED     DTOT,PTOT  
        PUT    SYSOUT,MSG  
        WRITE  DECB2,SF,SYSUT2  
        CHECK  DECB2  
        B      LOOP  
EOF     CLOSE  (SYSUT1,,SYSUT2,,SYSOUT)  
        WTO   'TESTDCBB ENDED OK'  
        SUBEXIT  
SYSUT1  DCB    DSORG=PS,DDNAME=SYSUT1,EODAD=EOF,MACRF=R,  
                RECFM=F,BLKSIZE=80,RECORD=RECORD           X  
SYSUT2  DCB    DSORG=PS,DDNAME=SYSUT2,MACRF=W,RECFM=F,BLKSIZE=80,  
                RECORD=RECORD           X  
SYSOUT  DCB    DSORG=PS,DDNAME=SYSOUT,RECFM=FT,BLKSIZE=120,MACRF=PM  
PTOT    DC     PL2'0'  
MSG     DS     OCL120  
        DC     C'REC#='  
DTOT    DC     CL4'  ;C' TEXT='  
RECORD  DC     CL80' ;  
        DC     (MSG+120-* )C' '  
        DCBD  
        DECBD  
        EQUREGS  
        END
```

U4 – Lenguaje ensamblador

- Traducción versus Interpretación
 - Traductor
 - Definición: “Programa que convierte un programa de usuario escrito en un lenguaje (fuente) en otro lenguaje (destino)
 - Clasificación:
 - Compiladores
 - Ensambladores
 - Intérprete
 - Definición: “Programa que ejecuta directamente un programa de usuario escrito en un lenguaje fuente”

U4 – Lenguaje ensamblador

- Traducción versus Interpretación
 - Traducción
 - Lenguajes compilados:
 - C, C++, Go, Rust
 - Lenguajes ensambladores:
 - Intel 64/IA-32, ARM, SPARC, MIPS, IA-64 (Itanium)
 - Interpretación
 - Lenguajes interpretados:
 - Python, JavaScript, Ruby
 - Bytecode
 - Java

U4 – Lenguaje ensamblador

○ Ensambladores

- Definición: “Programa que traduce un programa escrito en lenguaje ensamblador y produce código objeto como salida”
- Traducción 1 a 1 a lenguaje máquina
- Hay dos tipos:
 - Dos pasadas
 - Una pasada

U4 – Lenguaje ensamblador

- Ensambladores
 - Dos pasadas
 - Primera
 - Definición de etiquetas → Tabla de símbolos
 - LC: location counter (empieza en 0 con el 1er byte del código objeto ensamblado)
 - Se examina cada sentencia de lenguaje ensamblador
 - Determina la longitud de la instrucción de máquina (reconoce opcode + modo de direccionamiento + operandos) para actualizar el LC
 - Revisa directivas al ensamblador.
 - Ejemplo: definición de áreas de storage
 - Por cada etiqueta encontrada se fija si está en la tabla de símbolos. Si no lo está la agrega (si es la definición, registra el LC como tal, sino lo registra como referenciando a la etiqueta)

U4 – Lenguaje ensamblador

○ Ensambladores

- Dos pasadas
 - Segunda (Traducción)
 - Traduce el mnemónico en el opcode binario correspondiente
 - Usa el opcode para determinar el formato de la instrucción y la posición y tamaño de cada uno de los campos de la instrucción
 - Traduce cada nombre de operando en el registro o código de memoria apropiado
 - Traduce cada valor inmediato en un string binario en la instrucción
 - Traduce las referencias a etiquetas en el valor apropiado de LC usando la tabla de símbolos
 - Setear otros bits necesarios en la codificación de la instrucción.
 - Ejemplo: indicadores de modo de direccionamiento, bits de código de condición, etc.

U4 – Lenguaje ensamblador

○ Referencias

- “Computer Organization and Architecture – Designing for Performance”
9na edición. William Stallings
(<http://williamstallings.com/ComputerOrganization/>)
- “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin
(<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>)

75.03 & 95.57 Organización del Computador

U4 - LENGUAJE ENSAMBLADOR (SEGUNDA PARTE)

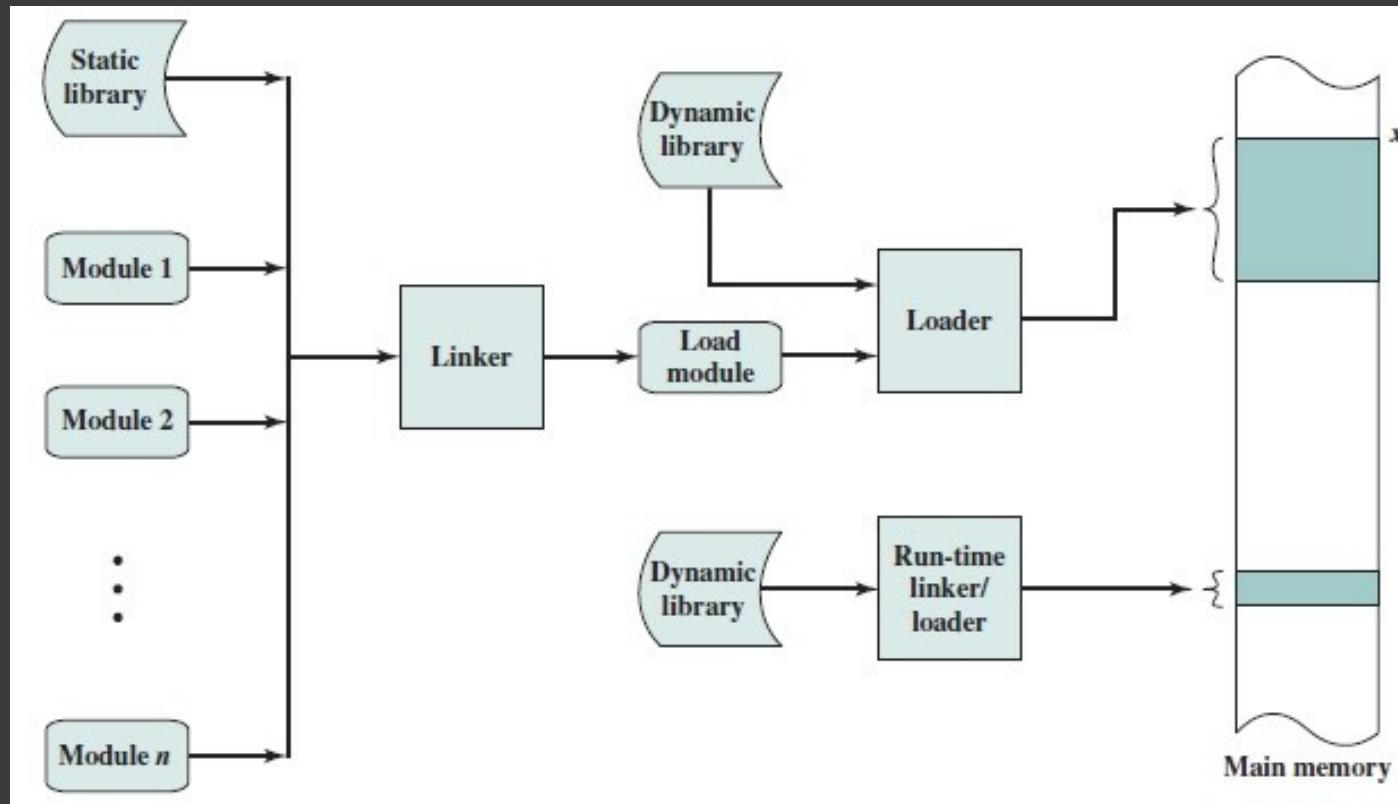
U4 - Lenguaje ensamblador

- Ensamblador
 - Código objeto
 - Definición: “Es la representación en lenguaje de máquina del código fuente programado. Es creado por un compilador o ensamblador y es luego transformado en código ejecutable por el linkeditor”

U4 - Lenguaje ensamblador

- Más allá del ensamblado
 - Linker
 - Definición: “Programa utilitario que combina uno o más archivos con código objeto en un único archivo que contiene código ejecutable o cargable”
 - Loader
 - Definición: “Rutina de programa que copia un ejecutable a memoria principal para ser ejecutado”

U4 - Lenguaje ensamblador



U4 - Lenguaje ensamblador

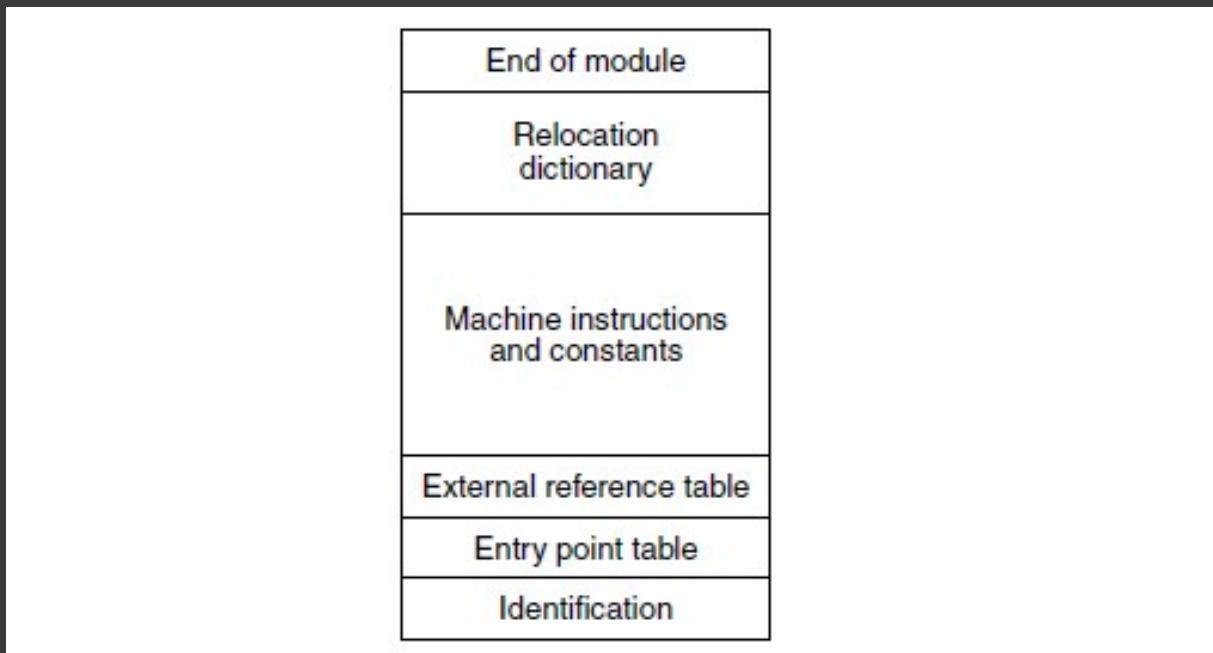
- Dos problemas a resolver
 - Direcciones externas
 - Existen direcciones en el código objeto que no se pueden resolver en tiempo de ensamblado
 - Reubicabilidad
 - ¿Por qué es necesaria?
 - No se sabe que otro programa habrá en memoria a la vez
 - Swap a disco en un entorno de multiprogramación

U4 - Lenguaje ensamblador

- Código objeto
 - Estructura interna
 - Identificación: nombre del módulo, longitudes de las partes del módulo
 - Tabla de punto de entrada: lista de símbolos que pueden ser referenciados desde otros módulos
 - Tabla de referencias externas: lista de símbolos usados en el módulo pero definidos fuera de él y sus referencias en el código
 - Código ensamblado y constantes
 - Diccionario de reubicabilidad: lista de direcciones a ser reubicadas
 - Fin de módulo

U4 - Lenguaje ensamblador

- Código objeto
 - Estructura interna



U4 - Lenguaje ensamblador

- Código objeto
 - Formatos estandarizados
 - OMF (Object Module Format)
 - COFF (Common Object File Format)
 - ELF (Executable and Linkable Format)

U4 - Lenguaje ensamblador

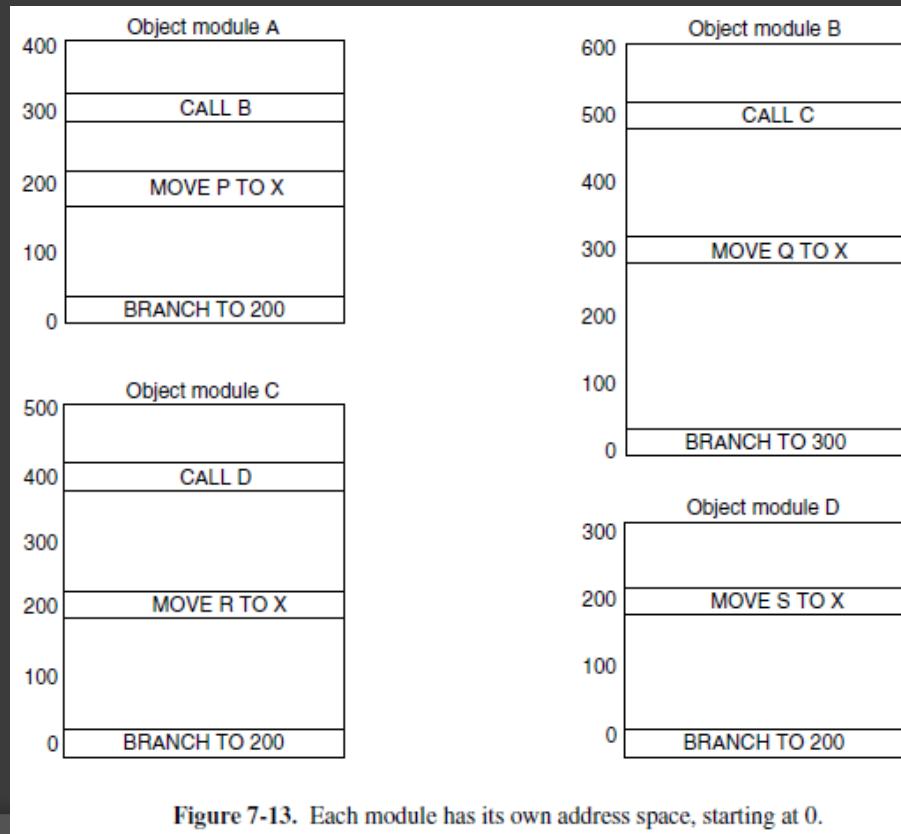
- Linking
 - Estático (linkage editor)
 - Dinámico
 - Load time dynamic linking
 - Run time dynamic linking

U4 - Lenguaje ensamblador

- Linking
 - Estático (linkage editor)
 - Cada módulo objeto compilado o ensamblado es creado con referencias relativas al inicio del módulo
 - Se combinan todos los módulos objeto en un único load module reubicable con todas las referencias relativas al load module

U4 - Lenguaje ensamblador

- Linking estático
 - Módulos objeto



U4 - Lenguaje ensamblador

- Linking estático
 - Generación del load module
 1. Construye tabla de todos los módulos objeto y sus longitudes
 2. Asigna dirección base a cada módulo en base a esa tabla
 3. Busca todas las instrucciones que refieren a memoria y les suma una *constante de reubicación* igual a la dirección de inicio de su módulo objeto
 4. Busca todas las instrucciones que refieren a otros procedimientos e inserta su dirección

U4 - Lenguaje ensamblador

- Linking estático
 - Generación del load module

Module	Length	Starting address
A	400	100
B	600	500
C	500	1100
D	300	1600

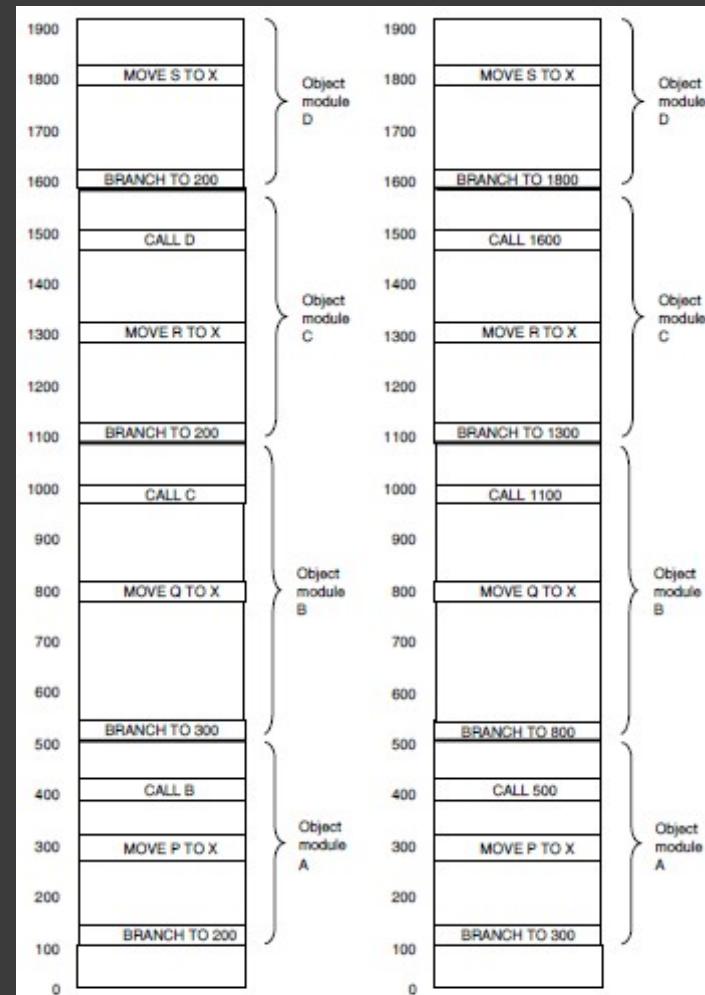


Figure 7-14. (a) The object modules of Fig. 7-13 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed.

U4 - Lenguaje ensamblador

- Linking dinámico
 - Se difiere la linkificación de algún módulo hasta luego de la creación del load module
 - Dos tipos:
 - Load time dynamic linking
 - Run time dynamic linking

U4 - Lenguaje ensamblador

- Linking dinámico
 - Load time dynamic linking
 1. Se levanta a memoria el load module
 2. Cualquier referencia a un módulo externo hace que el loader busque ese módulo, lo cargue y cambie la referencia a una dirección relativa desde el inicio del load module
 - Ventajas
 - Facilita la actualización de versión del módulo externo porque no hay que recompilar
 - El sistema operativo puede cargar y compartir una única versión del módulo externo
 - Facilita la creación de módulos de lindeo dinámico a los programadores (ej. Bibliotecas .so en Unix)

U4 - Lenguaje ensamblador

- Linking dinámico
 - Run time dynamic linking
 - Se pospone el lindeo hasta el tiempo de ejecución
 - Se mantienen las referencias a módulos externos en el programa cargado
 - Cuando efectivamente se invoca al módulo externo, el sistema operativo lo busca, lo carga y linkea al módulo llamador.
 - Ventajas
 - No ocupo memoria hasta que la necesito (ej. Bibliotecas DLL de Windows)

U4 - Lenguaje ensamblador

- Loading
 - Loading absoluto
 - El compilador/ensamblador genera direcciones absolutas
 - Solo se puede cargar en un único espacio de memoria
 - Loading reubicable
 - El compilador/ensamblador genera direcciones relativas al LC=0
 - El loader debe sumar un valor X a cada referencia a memoria cuando carga el módulo en memoria
 - El load module tiene que tener información para saber cuales son las referencia a memoria a modificar (diccionario de reubicación)
 - Loading por registro base
 - Arquitecturas que usan registros base para el direccionamiento
 - Se asigna un valor para el registro base asociado a la ubicación en la que se cargó el programa en memoria
 - Loading dinámico en tiempo de ejecución
 - Se difiere el cálculo de las direcciones absolutas hasta que realmente se vaya a ejecutar
 - El load module se carga a memoria con las direcciones relativas
 - La dirección se calcula solo al momento de ejecutar realmente la instrucción (con soporte de hardware especial)

U4 - Lenguaje ensamblador

- Linking (resumen de momentos de lindeo)

(b) Linker

Linkage Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile or assembly time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module creation	All object modules have been assembled using relative addresses. These modules are linked together and all references are restated relative to the origin of the final load module.
Load time	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main or virtual memory.
Run time	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted and the desired module is linked to the calling program.

U4 - Lenguaje ensamblador

- Loading (resumen de momentos de “binding”)

(a) Loader	
Binding Time	Function
Programming time	All actual physical addresses are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.
Load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.
Run time	The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

U4 - Lenguaje ensamblador

- Referencias
 - “Computer Organization and Architecture – Designing for Performance” 9na edición. William Stallings (
<http://williamstallings.com/ComputerOrganization/>)
 - “Structured Computer Organization” 6ta edición. Andrew Tanenbaum / Todd Austin (
<http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page>
)
 - “Linkers & Loaders” 1ra edición. John R. Levine
(<https://www.johnlevine.com/index.phtml>)

Linking dinamico

Enlace Dinámico

El **enlace dinámico** permite generar un ejecutable sin incluir todo el código binario necesario para su ejecución. En su lugar, ciertas referencias externas pueden resolverse posteriormente, ya sea en el **tiempo de carga** o en el **tiempo de ejecución**. Esto significa que, al principio, el ejecutable contiene referencias sin resolver que serán gestionadas por el sistema en el momento adecuado.

Enlace Dinámico en Tiempo de Carga (Load-Time Dynamic Linking)

En este enfoque, el **loader** combina el **load module** con las bibliotecas dinámicas requeridas en el momento en que se carga el ejecutable en memoria. Cuando el programa hace referencia a un módulo externo, el loader lo busca, lo carga en memoria y actualiza la referencia dentro del ejecutable para que apunte correctamente a la dirección donde se ubicó el módulo.

Proceso del Enlace Dinámico en Tiempo de Carga:

1. Se carga en memoria el **load module**.
2. Se identifican referencias a módulos externos.
3. El **loader** busca los módulos externos requeridos.
4. Los módulos externos son cargados en memoria y se actualizan las referencias en el programa.

Ventajas:

- Permite actualizar módulos externos sin necesidad de recompilar el ejecutable principal.
- El sistema operativo puede cargar y compartir una única versión del módulo externo entre varios procesos.
- Facilita el desarrollo de bibliotecas de enlace dinámico (ejemplo: archivos **.so** en Unix/Linux).

Enlace Dinámico en Tiempo de Ejecución (Run-Time Dynamic Linking)

En este caso, el ejecutable contiene referencias a módulos externos que **no se resuelven en el momento de la carga**, sino que se gestionan en **tiempo de ejecución**. El sistema operativo reconoce estas referencias cuando el programa intenta utilizarlas y, en ese momento, busca la biblioteca dinámica correspondiente, la carga en memoria y ajusta los enlaces.

Proceso del Enlace Dinámico en Tiempo de Ejecución:

1. Se ejecuta el programa con referencias a módulos externos sin resolver.
2. Cuando se invoca una función o recurso externo, el sistema operativo detecta la referencia.
3. Se busca el módulo requerido en una biblioteca dinámica.

4. El módulo se carga en memoria y se actualizan las referencias en el programa.

Ventajas:

- Reduce el uso de memoria, ya que los módulos externos solo se cargan cuando realmente se necesitan (ejemplo: **DLLs en Windows**).
- Permite la carga y descarga dinámica de funcionalidades, optimizando el rendimiento del sistema.

Linking Estatico

El **enlace estático** es un proceso en el que un enlazador (linker) combina varios módulos de código objeto en un único archivo ejecutable. Cada módulo objeto contiene referencias relativas a su propio inicio, pero el enlazador ajusta estas referencias para que apunten correctamente dentro del ejecutable final.

Pasos del proceso de enlace estático:

1. Construcción de la tabla de módulos objeto:

El enlazador analiza todos los módulos objeto y registra sus longitudes en una tabla, permitiendo determinar la ubicación relativa de cada módulo dentro del ejecutable final.

2. Asignación de direcciones base:

A partir de la tabla de módulos, el enlazador asigna una dirección base a cada módulo. La dirección base del primer módulo suele comenzar después de una cabecera (por ejemplo, en la dirección 100 si la cabecera ocupa ese espacio). Los módulos subsiguientes se colocan inmediatamente después del anterior en la memoria.

3. Ajuste de referencias a memoria:

Dentro de cada módulo, las referencias a direcciones de memoria son relativas a su inicio. Para hacerlas coherentes en el ejecutable final, el enlazador suma a cada referencia una constante de reubicación igual a la dirección base del módulo donde se encuentra. Esto garantiza que los saltos e instrucciones de acceso a memoria apunten correctamente a sus destinos.

4. Resolución de referencias externas:

Cuando un módulo hace referencia a un procedimiento o variable definida en otro módulo, el enlazador busca la dirección real de ese símbolo en la tabla de símbolos global y actualiza las referencias dentro del código objeto.

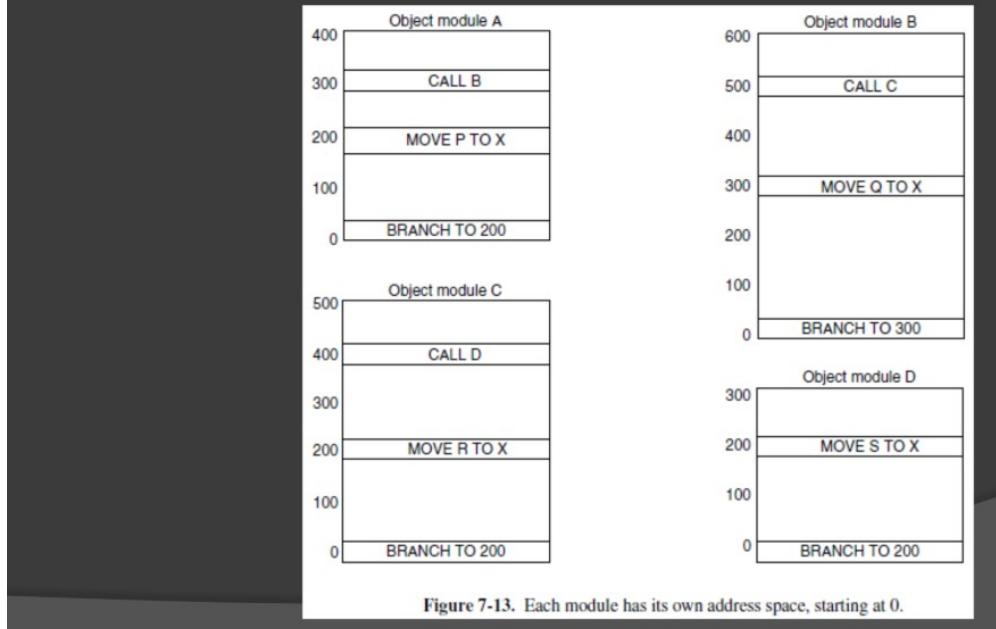
5. Generación del ejecutable final:

Una vez que todas las referencias han sido ajustadas y resueltas, el enlazador combina los módulos objeto en un único **load module reubicable**. Este ejecutable final puede ser cargado en memoria y ejecutado por el sistema operativo sin necesidad de modificaciones adicionales.

Este proceso permite que todas las referencias de memoria y llamadas a funciones sean coherentes y correctas en el contexto del ejecutable final.

○ Linking estático

- Módulos objeto



○ Linking estático

- Generación del load module

Module	Length	Starting address
A	400	100
B	600	500
C	500	1100
D	300	1600

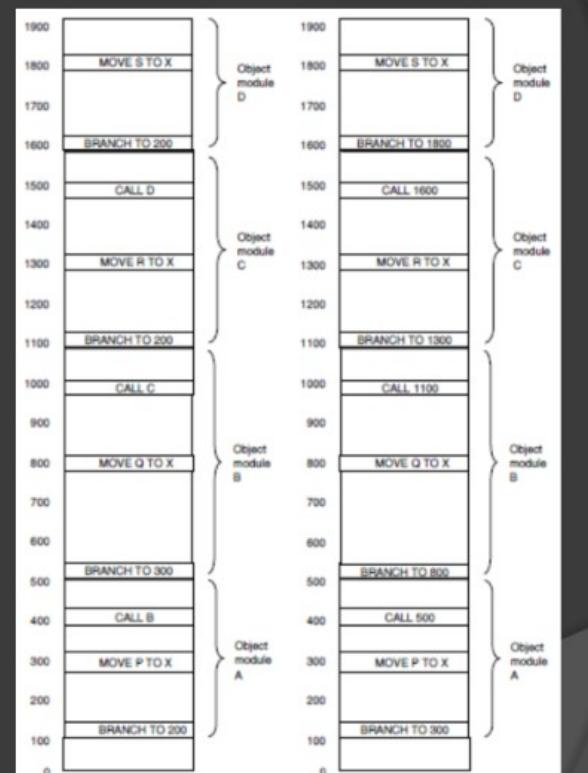


Figure 7-14. (a) The object modules of Fig. 7-13 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed.

Loading

He revisado y reescrito el texto sobre los diferentes métodos de carga de programas en memoria, añadiendo correcciones y mejoras para mayor claridad:

Proceso de Carga (Loading)

El proceso de carga implica situar un programa en la memoria principal para su ejecución. Existen varios métodos de carga, cada uno con sus características y aplicaciones:

1. Carga Absoluta

- El compilador o ensamblador genera direcciones de memoria absolutas en el código objeto.
- El programa solo puede cargarse en una ubicación específica de la memoria principal.
- Este método es rígido y poco flexible, ya que cualquier cambio en la ubicación requiere modificar y recomilar el código fuente.☛

2. Carga Reubicable

- El compilador o ensamblador produce direcciones relativas, generalmente con un contador de ubicación (LC) comenzando en 0.☛
- Al cargar el módulo en memoria, el cargador (loader) suma un valor de reubicación a cada referencia de memoria, adaptando el programa a la posición en la que se carga.☛
- El módulo de carga debe contener información que indique qué referencias de memoria necesitan ser ajustadas, conocida como "diccionario de reubicación".☛

3. Carga por Registro Base

- Utilizada en arquitecturas que emplean registros base para el direccionamiento.☛
- Se asigna un valor al registro base correspondiente a la ubicación en la que se cargó el programa en memoria.☛
- Las direcciones relativas dentro del programa se resuelven sumando el contenido del registro base, permitiendo flexibilidad en la ubicación del programa.☛

4. Carga Dinámica en Tiempo de Ejecución

- El cálculo de direcciones absolutas se pospone hasta el momento de la ejecución del programa.☛
- El módulo de carga se sitúa en memoria con direcciones relativas.☛
- Durante la ejecución, un componente de hardware, como la Unidad de Gestión de Memoria (MMU), traduce las direcciones relativas a absolutas en tiempo real.☛

- →Este método ofrece gran flexibilidad, permitiendo que los programas se carguen en cualquier región de la memoria y facilitando su intercambio entre disco y memoria según sea necesario.→
- En la actualidad, la carga dinámica en tiempo de ejecución es ampliamente utilizada debido a su flexibilidad y eficiencia en la gestión de la memoria.→ Sin embargo, la elección del método de carga depende de las necesidades específicas del sistema y de la aplicación en cuestión.→

95.57/TB023 Organización del Computador

U1 – Sistemas de Numeración

Agenda

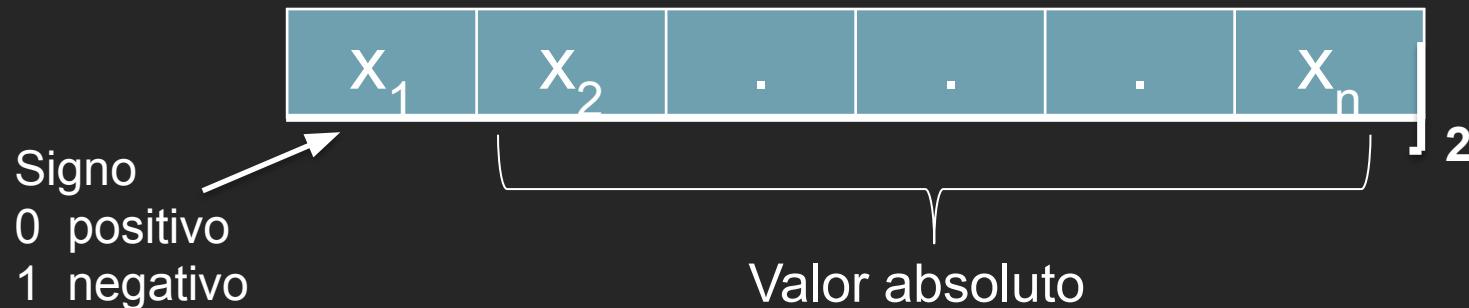
- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

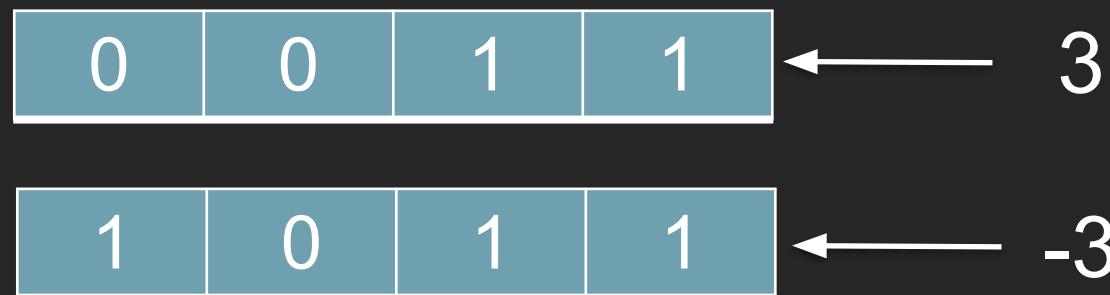
Bit de signo y valor absoluto

Base = 2 Precisión = n



Ejemplo:

Base = 2 Precisión = 4



Bit de signo y valor absoluto

Paso por paso

Base = 2 precisión = 4

Representar -6

- 1) Ver signo para determinar el primer bit: **1** (por ser negativo)
- 2) Pasar valor absoluto a base 2: $|-6_{10}| = 6_{10} = \textcolor{green}{110}_2$
- 3) Concatenar los bits: **1110**

Indicar número almacenado en **1101**

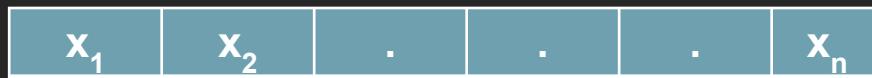
- 1) Ver primer bit para determinar el signo: negativo (por ser **1**)
- 2) Pasar a base 10 los bits descartando el primero: $\textcolor{blue}{101}_2 = 5_{10}$
- 3) Indicar el número según signo y valor obtenidos: **-5**

Antes de seguir veamos B^n

$$B^n = ?$$

base
cantidad de simblos

precisión
cantidad de dígitos



$$\begin{aligned} & B * B * \dots \dots * B \\ & = B^n = \# \text{ total de números representables} \end{aligned}$$

Ejemplos:

$$B = 10$$

$$n = 2$$

$$B^n = 10^2 = 100$$



[00, 01, ..., 99]

100

$$B = 2$$

$$n = 4$$

$$B^n = 2^4 = 16$$



0000	16
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Bit de signo y valor absoluto

Rango de Representación

Mínimo: $-(2^{n-1}-1)$

Máximo: $2^{n-1}-1$

Ventajas

- Rango simétrico

Desventajas

- Doble representación del 0
- No permite operar aritméticamente

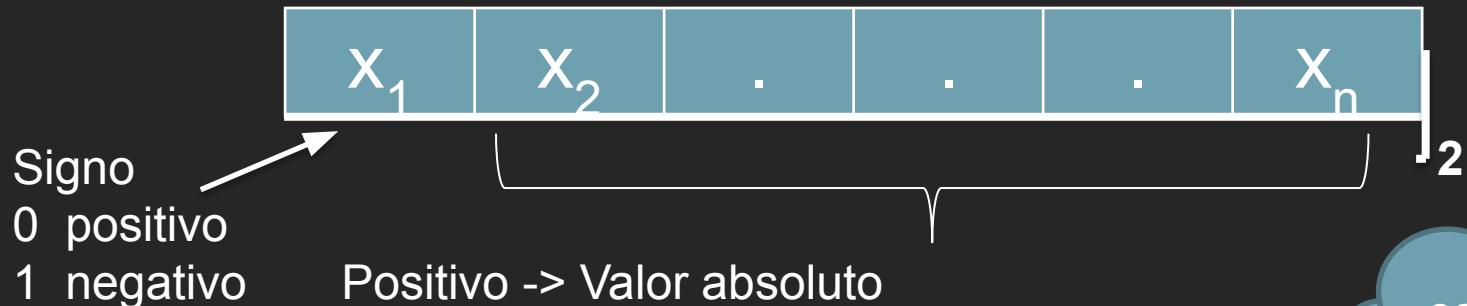
$$\begin{array}{r} 0001 \quad 1 \\ +1001 \quad + -1 \\ \hline \text{----} \quad \text{--} \\ 1010 \quad \neq \quad 0 \end{array}$$

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Complemento a 1

Base = 2 Precisión = n

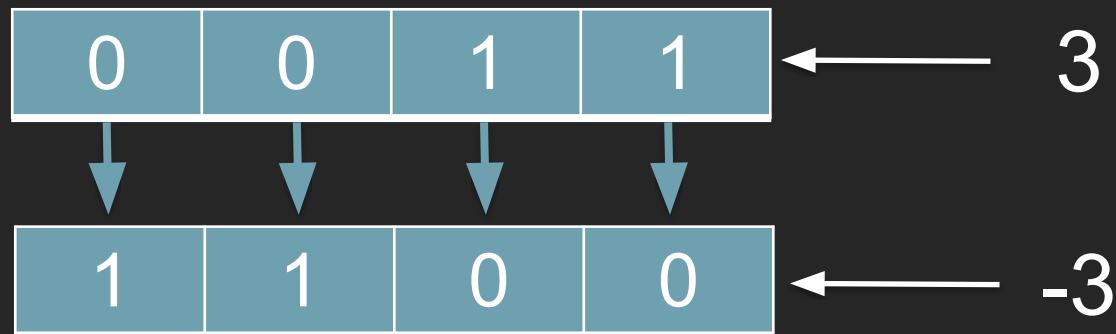


Positivo -> Valor absoluto
Negativo -> Complemento (aplicar NOT)
al valor absoluto

NOT
es cambiar
1 por 0
0 por 1

Ejemplo:

Base = 2 Precisión = 4



Complemento a 1

Paso por paso

Base = 2 precisión = 4

Representar -6

1. Pasar valor absoluto a base 2: $|-6_{10}| = 6_{10} = 110_2$
2. Completar con 0 a izquierda hasta completar n: 0110
3. Si es negativo, complementar (hacer NOT): 1001

Indicar número almacenado en 1101

1. Si primer bit es 1 (es negativo), complementar (hacer NOT): 0010
2. Pasar a base 10: $0010_2 = 2_{10}$
3. Indicar el número según signo y valor obtenidos: -2

Complemento a 1

Rango de Representación

Mínimo: $-(2^{n-1}-1)$

Máximo: $2^{n-1}-1$

Desventajas

- Doble representación del 0

Ventajas

- Rango simétrico
- Permite operar aritméticamente sumando el "end-around carry"

$$\begin{array}{r} \textcolor{red}{11} \\ 0101 \quad \quad 5 \\ +1110 \quad + \quad -1 \\ \hline \hline \\ 0011 \quad \neq \quad 4 \\ + \quad \textcolor{red}{1} \\ \hline \hline \\ 0100 \quad = \quad 4 \end{array}$$

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - **Complemento a la base**
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Complemento a la Base

Base = B Precisión = n

$$\text{Rep}(x_b) = x_b \quad \text{si } x \geq 0$$

$$\text{Rep}(x_b) = Cb(|x_b|) \quad \text{si } x < 0$$

Ejemplos:

$$B = 10 \quad n = 2 \Rightarrow B^n = 10^2 = 100$$

$$\text{Rep}(3) = 03$$

$$\text{Rep}(-3) = Cb(3) = 100 - 3 = 97$$

$$\text{Rep}(-1) = Cb(1) = 100 - 1 = 99$$

$$\text{Rep}(-50) = Cb(50) = 100 - 50 = 50$$

Rep(50) => NO SE PUEDE

=> 49 es el mayor positivo representable

¿ Pero que es Cb?

$$\begin{aligned} Cb(r) + r &= B^n \\ \Rightarrow Cb(r) &= B^n - r \end{aligned}$$

Notar que:

Si

k es complemento de r

$$\Rightarrow k + r = B^n$$

=> r es complemento de k

Complemento a la Base

Rango de Representación

Mínimo: $-(B^n/2)$

Máximo: $(B^n/2) - 1$

Desventajas

- Rango asimétrico (un negativo más)

Ventajas

- Única representación del 0
- Permite operar aritméticamente

Complemento a la Base

Permite operar aritméticamente:

Sumas (Con B = 10 y n = 2)

$$5 + 2 \quad (\text{es } 7)$$

$$5 + (-2) \quad (\text{es } 3)$$

$$-5 + 2 \quad (\text{es } -3)$$

$$-5 + (-2) \quad (\text{es } -7)$$

$$05 + 02$$

$$05 + 98$$

$$95 + 02$$

$$95 + 98$$

$$\begin{array}{r} 00 \\ 05 \\ +02 \\ \hline \end{array}$$

$$\begin{array}{r} 11 \\ 05 \\ +98 \\ \hline \end{array}$$

$$\begin{array}{r} 00 \\ 95 \\ +02 \\ \hline \end{array}$$

$$\begin{array}{r} 11 \\ 95 \\ +98 \\ \hline \end{array}$$

$$\begin{array}{r} \cancel{+}07 \\ \hline \cancel{+}07 \end{array} \quad (\text{A})$$

$$\begin{array}{r} \cancel{+}03 \\ \hline \cancel{+}03 \end{array} \quad (\text{B})$$

$$\begin{array}{r} \cancel{+}97 \\ \hline \cancel{-}97 \end{array} \quad \rightarrow -3 \quad (\text{C})$$

$$\begin{array}{r} \cancel{+}93 \\ \hline \cancel{-}93 \end{array} \quad \rightarrow -7 \quad (\text{D})$$

Restas: al plantear A-B se transforma en A+Bcomp

$$5 - 2 \quad (\text{es } 3)$$

$$5 - (-2) \quad (\text{es } 7)$$

$$-5 - 2 \quad (\text{es } -7)$$

$$-5 - (-2) \quad (\text{es } -3)$$

$$\begin{array}{r} 05 - 02 \\ 05 + \text{Cb}(2) \\ = 05 + 98 \\ = \cancel{\pm}03 \end{array}$$

$$\begin{array}{r} 05 - 98 \\ 05 + \text{Cb}(98) \\ = 05 + 02 \\ = 07 \end{array} \quad (\text{A})$$

$$\begin{array}{r} 95 - 02 \\ 95 + \text{Cb}(2) \\ = 95 + 98 \\ = \cancel{\pm}93 \end{array} \quad (\text{D})$$

$$\begin{array}{r} 95 - 98 \\ 95 + \text{Cb}(98) \\ = 95 + 02 \\ = 97 \end{array} \quad (\text{C})$$

Complemento a la Base

Permite operar aritméticamente: CONCLUSIÓN

$$A - B$$

Se trabaja como

$$A + \text{Comp}(B)$$

*** NO IMPORTA EL SIGNO DE B ***

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - **Complemento a 2**
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Complemento a 2

Base = 2 Precisión = n

$$\begin{aligned} \text{Rep}(x_{10}) &= x_2 && \text{si } x \geq 0 \\ \text{Rep}(x_{10}) &= \text{NOT}(|x_2|) + 1 && \text{si } x < 0 \end{aligned}$$

Ejemplos:

$$B = 2 \quad n = 4$$

$$\text{Rep}(3) = 0011$$

$$\begin{aligned} \text{Rep}(-3) &= \text{NOT}(0011) + 1 \\ &= 1100 + 1 \\ &= 1101 \end{aligned}$$

$$\begin{aligned} \text{Rep}(-1) &= \text{NOT}(0001) + 1 \\ &= 1110 + 1 \\ &= 1111 \end{aligned}$$

$$\text{Rep}(-8) = \text{NOT}(1000) + 1$$

$$\begin{aligned} &= 0111 + 1 \\ &= 1000 \end{aligned}$$

$\text{Rep}(8) \Rightarrow$ NO SE
PUEDE
 $\Rightarrow 7 = 0111_2$ es el mayor
positivo representable

Veamos...

0000	0001
0001	+1111
0010	---
0011	10000
0100	0011
0101	+1101
0110	---
0111	10000
1000	0111
1001	+1001
1010	---
1011	10000
1100	0111
1101	+1001
1110	---
1111	10000

NOT + 1
es Cb
con b=2

Complemento a 2

Paso por paso

Base = 2 precisión = 4

Representar -6

1. Pasar valor absoluto a base 2: $|-6_{10}| = 6_{10} = 110_2$
2. Completar con 0 a izquierda hasta completar n: 0110
3. Si es negativo, complementar (hacer NOT + 1): $1001 + 1 = 1010$

Indicar número almacenado en 1101

1. Si primer bit es 1 (es negativo), complementar (hacer NOT+1): $0010 + 1 = 0011$
2. Pasar a base 10 los bits: $0011_2 = 3_{10}$
3. Indicar el número según signo y valor obtenidos: -3

Complemento a 2

Rango de Representación

Mínimo: $-(2^{n-1})$

Máximo: $2^{n-1} - 1$

Desventajas

- Rango asimétrico (un negativo más)

Ventajas

- Única representación del 0
- Permite operar aritméticamente.

Aplica la misma mecánica de resolver $X-Y$ como $X+Cb(Y)$

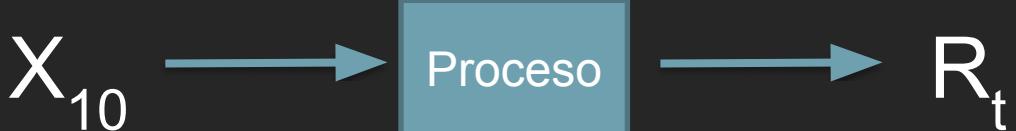
Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso (*)
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754 (*)

Formato y Configuración

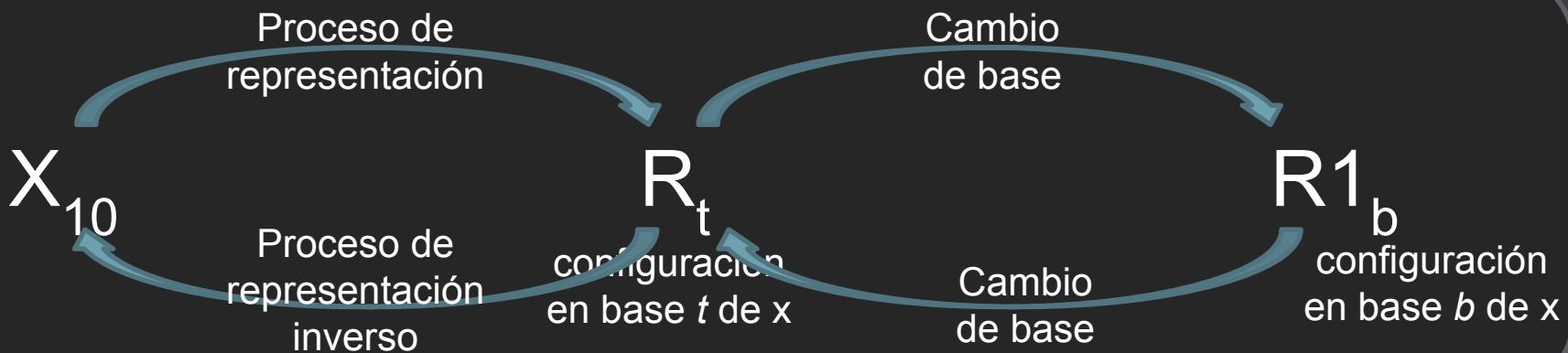
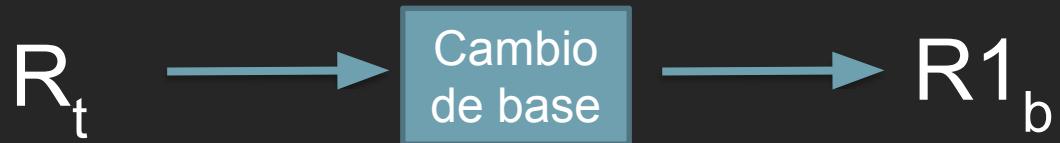
Formato:

Representación computacional de un número



Configuración:

Expresión en una determinada base de un número en un formato

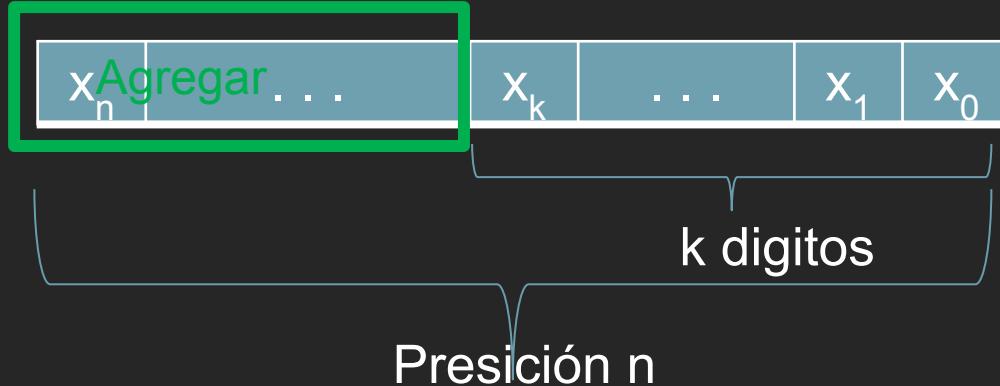


Agenda

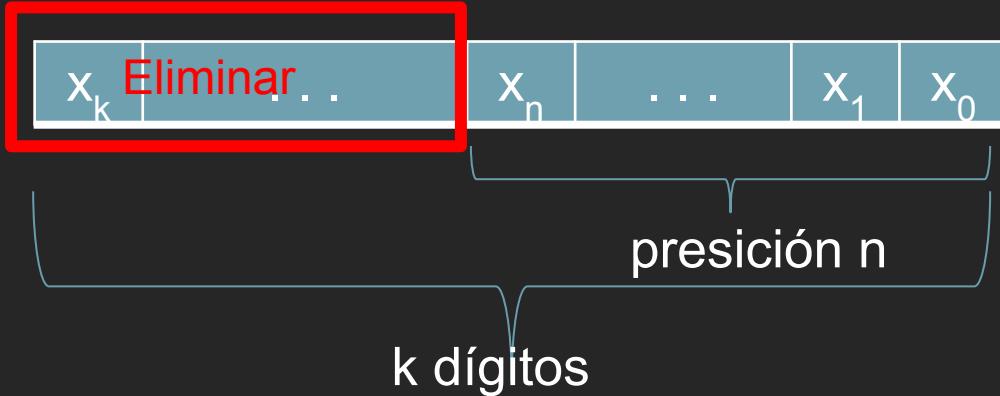
- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Expansión y Truncamiento

Expansión:



Truncamiento:



Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Binario de punto fijo sin signo

Base = 2 Precisión = n Enteros positivos

Como almacenar un número

- 1) Pasar el nro a base 2
- 2) Completar con 0 a izquierda hasta alcanzar n digitos

Como recuperar un número almacenado

Pasos anteriores en orden inverso

Rango de representación

Mínimo: 0

Máximo: $2^n - 1$

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Binario de punto fijo con signo

Base = 2 Precisión = n Enteros positivos y negativos

Es la implementación del método complemento a 2

Como almacenar un número

- 1) Pasar el nro a base 2
- 2) Completar con 0 a izquierda hasta alcanzar n dígitos
- 3) Si el nro es negativo, complementar usando método de "complemento a 2" (Not +1)

Como recuperar un número almacenado

- 1) Si el primer bit es 1 (es negativo), complementar.
- 2) Quitar 0 a izquierda.
- 3) Pasar a base 10 y colocar el signo que corresponda.

Binario de punto fijo con signo

Validación Overflow en operaciones aritméticas

B=2 n=4

Resolver $7 + 1$

$$\begin{array}{r} 7_{10} = 0111_2 \\ 1_{10} = 0001_2 \end{array}$$

0111 Ultimos 2 acarreos distintos
0111 => **OVERFLOW**

$$\begin{array}{r} + 0001 \\ \hline \end{array}$$

1000

Resolver $7 - 1$

$$\begin{array}{r} 7_{10} = 0111_2 \\ 1_{10} = 0001_2 \end{array}$$

Hallo C(1) para hacer $7+C(1)$

$$\begin{array}{r} \text{NOT}(0001) = 1110 \\ + 1 \\ \hline \end{array}$$

1111

Ahora sumo

$$\begin{array}{r} 1111 \quad \text{Ultimos 2 acarreos iguales} \\ 0111 \quad => \text{VALIDO} \\ + 1111 \\ \hline \\ 0110 \end{array}$$

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - **BCD Empaquetado**
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

BCD Empaquetado

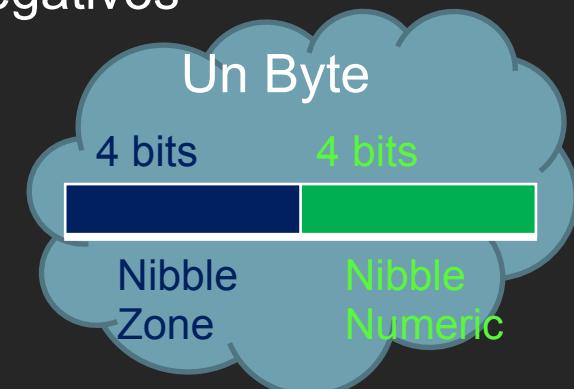
Base = 16 Precisión = n Enteros positivos y negativos

Como almacenar un número

- 1) Pasar el nro a base 10
- 2) Colocar c/dígito en los nibbles dejando libre el último (el de la derecha)
- 3) Colocar en el último nibble el signo siendo
C, A, F o E para positivos
B o D para negativos

Ej. n=3 $+123_{10} \rightarrow 00123A_{16}$

$-456_{10} \rightarrow 00456B_{16}$



Como recuperar un número almacenado

- 1) Tomar cada dígito de los nibbles (excepto el último) para armar la cadena en base 10
- 2) Colocar el signo según el dígito del último nibble

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Zoneado

Base = 16 Precisión = n Enteros positivos y negativos

Como almacenar un número

- 1) Pasar el nro a base 10
- 2) Colocar c/digito en los nibbles numeric
- 3) Colocar una F en cada nibble zone excepto en el último
- 4) Colocar en el último nibble zone el signo siendo
C, A, F o E para positivos
B o D para negativos

Ej. n=4 +123₁₀ --> F0F1F2A3₁₆ -456₁₀ --> F0F4F5B6₁₆

Como recuperar un número almacenado

- 1) Tomar cada digito de los nibbles numeric para armar la cadena en base 10
- 2) Colocar el signo según el dígito del último nibble zone

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

ASCII / EBCDIC / UNICODE

- Representación en forma digital/numéricas de los caracteres
 - ASCII (American Standard Code for Information Interchange)
 - 7 bits □ ASCII Básico
 - 8 bits □ ASCII Extendido
 - EBCDIC (Extended Binary Coded Decimal Interchange Code)
 - 8 bits
 - UNICODE Consortium
 - Codificación Universal/Estandard de los Caracteres
 - Consorcio Unicode □ Unicode 14.0
 - UTF-8 / UTF-16 / UTF-32T
 - <https://home.unicode.org/>

ASCII / EBCDIC / UNICODE

Hex	Dec	EBCDIC	ASCII
23	35		#
30	48		0
31	49		1
42	66		B
62	98		b
7B	123	#	{
82	130	b	
C0	192	{	
C2	194	B	
F0	240	0	
F1	241	1	

[Documento tablas ASCII & EBCDIC](#)

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Exceso a la base

Base = B Precisión = n

Rep(x_b) = $x_b + \text{exceso}$ (para todo x)

Con $B = 2$

$$\begin{aligned}\text{Exceso} &= 2^n/2 \\ &= 2 \cdot 2 \cdot \dots \cdot 2/2 \\ &= 2^{n-1}\end{aligned}$$

Ejemplos:

$$B = 10 \quad n = 2$$

$$\Rightarrow B^n/2 = 10^2/2 = 50$$

$$\text{Rep}(3) = 3 + 50 = 53$$

$$\text{Rep}(-3) = -3 + 50 = 47$$

$$\text{Rep}(0) = 0 + 50 = 50$$

$$\text{Rep}(-50) = -50 + 50 = 0$$

Rep(-51) NO SE PUEDE (da -1)

$$\text{Rep}(49) = 49 + 50 = 99$$

Rep(50) NO SE PUEDE (da 100)

¿ Pero que es el exceso?

Es $B^n/2$

Entendamos por qué:

Con $B=10$ y $n = 1$

$B^n = 10$ valores posibles

Cuál sería el rango de números a representar "más justo"?



Cuánto hay q sumar como mínimo a cada negativo para que "desaparezca" el signo?

5

Que termina siendo $B^n/2$

Números a representar

-5 -4 -3 -2 -1 0 1 2 3 4

Como se representan

0 1 2 3 4 5 6 7 8 9

Agenda

- **Métodos de representación de números negativos**
 - Bit de signo y valor absoluto
 - Complemento a 1
 - Complemento a la base
 - Complemento a 2
 - Exceso
- **Conceptos de Formato y Configuración**
- **Conceptos de Expansión y Truncamiento**
- **Formatos de representación de números enteros**
 - Binario de punto fijo sin signo
 - Binario de punto fijo con signo
 - BCD Empaquetado
 - Zoneado
- **Formatos de representación caracteres**
 - ASCII
 - EBCDIC
 - UNICODE
- **Formatos de representación números decimales**
 - Binario punto Flotante IEEE 754

Binario punto Flotante IEEE 754

- Notación Científica

$$S M \times B^E$$

- $-/+ 765,987 \times 10^{-3}_{10}$
- $-/+ 765987 \times 10^{-6}_{10}$
- $-/+ 7,65987 \times 10^{-1}_{10}$
- $-/+ 0,0765987 \times 10^1_{10}$

- Mantisa Normalizada

$$0 < M < B$$

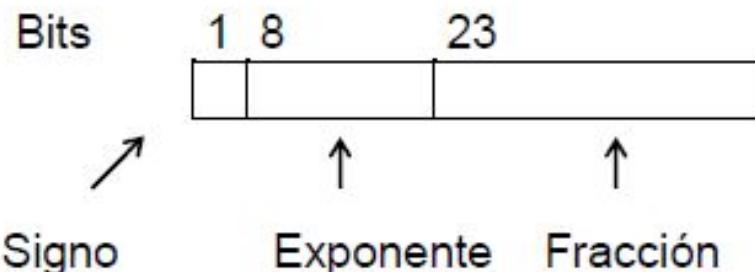
¿Cuál es el dígito de la cifra significativa en Binario?

Binario punto Flotante IEEE 754

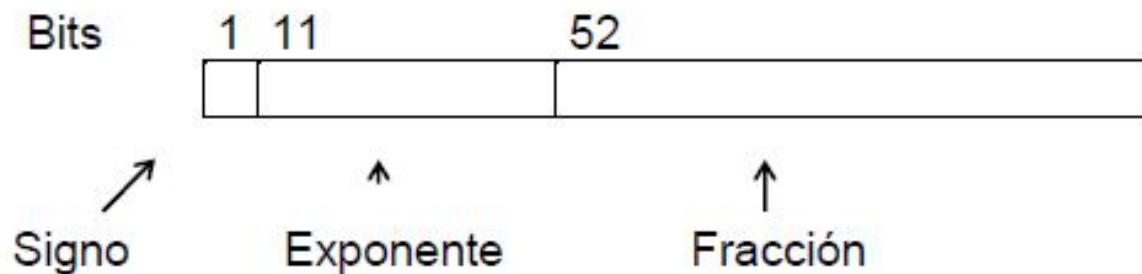
- 80s □ Institute of Electrical and Electronics
- Precisión
 - Simple (32) / Doble (64) / Extendida (128)
- Representación
 - Signo
 - Exponente
 - Mantisa
- Exponente en Exceso
 - 127 / 1023 / 16383
- Mantisa normalizada

Binario punto Flotante IEEE 754

Simple precisión



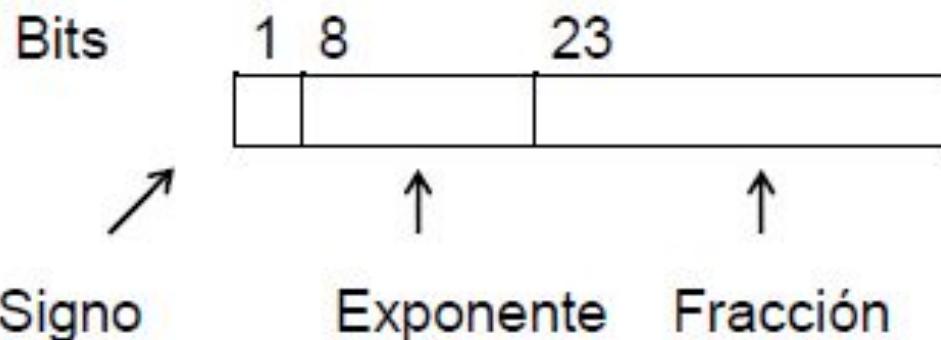
Doble precisión



Binario punto Flotante IEEE 754

- Precisión Simple

Simple precisión



- Exceso de 127
 - $E_{\text{Exceso}} = \text{Exp} + 127_{10}$
- Mantisa Normalizada
 - 1 implícito

Binario punto Flotante IEEE 754

-123,456₁₀

1) Paso de base 10 □ base 2

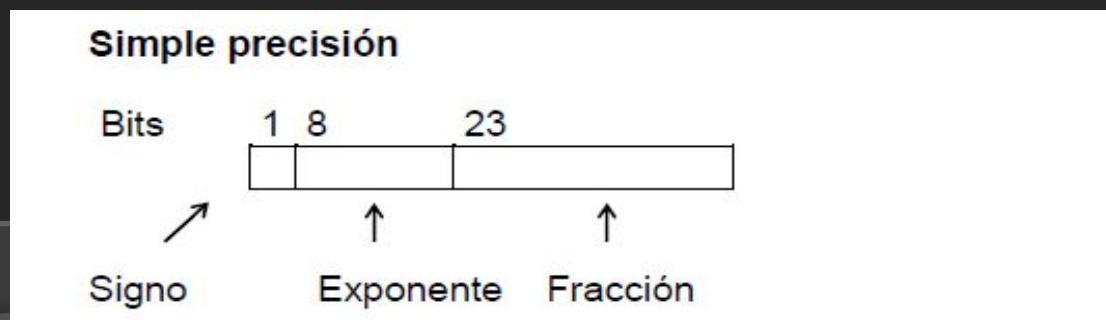
$$\begin{array}{rcl} 123_{10} & \square & 1111011_2 \\ 0,456_{10} & \square & 01110100_2 \end{array}$$

$$-123,456_{10} \square -1111011,01110100_2$$

2) Normalizo

$$-1,11101101110100 \times 10^{110}_2$$

3) Almaceno



Binario punto Flotante IEEE 754

3) Almaceno

Signo = 1

EExceso

$$6 + 127 = 133_{10}$$

$$133_{10} \square 10000101_2$$

Mantisa

$$1,11101101110100_2$$

4) Nro. Final

$$1 | 10000101 | 11101101110100\textcolor{red}{0000000000}_2$$

Binario punto Flotante IEEE 754

Normalizado	\pm	0 < Exp < Max	Cualquier patrón de bits
Desnormalizado	\pm	0	Cualquier patrón de bits $\neq 0$
Cero	\pm	0	0
Infinito	\pm	111...1	0
NAN	\pm	111...1	Cualquier patrón de bits $\neq 0$

Unidad 4 – Segunda parte

El lenguaje de ensamblador es una representación simbólica del lenguaje de máquina para un procesador en específico.

El ensamblador realiza dos pasadas mediante el proceso de traducción. Primero transforma el código fuente (código assembler) al código destino (código de máquina). Esto sucede para cualquier lenguaje de programación compilable. El producto del código ensamblado / compilado es el archivo código objeto.

Las pasadas significa abrir el código fuente y lo lee linea por linea para generar algo. En la segunda pasada vuelve a hacer el barrido y utiliza el resultado de la primer pasada para completar la traducción.

En la primer pasada se genera una tabla de símbolos (listing de ensamblado). Este es un archivo de texto que el ensamblador genera como resultado de la traducción. Una tabla de símbolos tiene filas y en cada una de ellas el ensamblador va poniendo las etiquetas del programa. El ensamblador genera esto en la primer pasada para generar la traducción. Cuando encuentra un símbolo en la segunda pasada, se asocia una dirección de memoria referente a donde se encontraba este símbolo. Cada etiqueta señala a una dirección en memoria particular y a partir de esta se genera una dirección relativa.

La tabla de símbolos se genera a partir del nombre del símbolo y la dirección relativa. Se genera a partir de esto un puntero “location counter”. Este location counter empieza en 0 con el 1er byte del código objeto ensamblado. Este contador se va actualizando, guardando solamente la longitud de cada instrucción de máquina, saltando las etiquetas y guardando solamente la información relevante.

Sabemos que la instrucción de máquina tiene una longitud distinta según en qué procesador estemos. La instrucción en intel tiene un tamaño variable. En ARM tenemos un formato fijo, donde cada instrucción tiene el tamaño de 32 bits.

Cuando el ensamblador de intel lee la instrucción de máquina tiene que interpretar el opcode, el modo de direccionamiento y los operandos. Todo este trabajo lo hace para saber cuánto mide la longitud de la instrucción de máquina. De esta forma sabe cuántos bytes debe actualizar el Location Counter. Si la primera instrucción arranca de 0 y tiene 2 bytes de tamaño, la segunda instrucción tendrá comienzo a partir de los 2 bytes de acuerdo a su posición relativa. De esta manera para cada símbolo se le asigna un valor del location counter para que en la segunda pasada se tenga esta información a mano.

Segunda pasada:

En el caso de Intel se tiene que descifrar el opcode, modo de direccionamiento y operandos mientras que en ARM no, tal que ya por defecto se sabe que la instrucción mide 32 bits. Entonces para cada etiqueta se le otorga 4 bytes. Si la primera instrucción tiene una dirección relativa de 0, la segunda tendrá de 4 y así. La excepción a esta regla es cuando se definen áreas de memoria específicas, donde el desplazamiento obviamente tendrá que ser calculado en base a esto.

En la segunda pasada en definitiva se termina de definir el código máquina necesario tras haber definido donde empieza y donde termina todo. La idea en la primera es dejar todo referenciado con el location counter que eso en definitiva es lo que le interesa al ensamblador en la segunda pasada para transformar a código máquina. Se realiza lo que sería la traducción. Cuando se realiza dicha traducción se genera el código objeto donde esta el código máquina.

La traducción primeramente transforma el mnemónico de la instrucción, en el opcode binario correspondiente. En algunos casos solo con el opcode se puede deducir el formato de la instrucción y posición y tamaño de cada uno de sus campos. Los compiladores son aquellos que tienen dicha información para realizar las traducciones.

Una vez que identifica el formato es capas de traducir cada nombre de operando en el registro o código de memoria apropiado. Todo valor en definitiva se convierte a código máquina.

Si en la traducción tengo una etiqueta/referencia en memoria lo que hago es remplazar dicha etiqueta por su equivalente escrito en la tabla de simbolos. En esta traducción también puedo haber indicadores de modo de direccionamiento, bits de codigo de condición, etc.

Código objeto

El código objeto es una representación del lenguaje de maquina del código fuente, generado por un compilador. El código ejecutable se crea despues por un linkeditor, algo aparte de lo que es el código máquina este. Cuando vos compilas/ensamblas no tenés todavía el ejecutable, internamente trabaja el link editor.

El código objeto contiene distintas secciones. La primera es una identificación (header). Contiene el nombre del módulo y sus respectivas longitudes. Luego se guarda la lista de símbolos que pueden ser referenciados desde otros módulos. Esto quiere decir que la tabla de punto de entrada no necesariamente tiene solamente lo que yo codeé, si no que guardo referencias a modulos (como funciones de C) para luego cuando se haga una link edición se pueda acceder a los mismos.

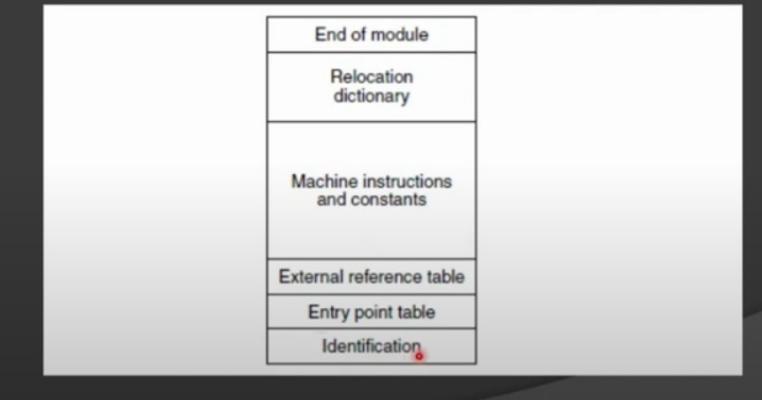
Después hay otra tabla de referencias externas, cuya lista de símbolos son los que se utilizan dentro del módulo pero se definen fuera de el (por ejemplo, sscanf). Mediante esta referencia a etiquetas externas luego puedo traerlas a partir de la link edición.

El centro del código objeto es el código de máquina y las constantes (todo lo que definí en lenguaje ensamblador).

Por último se tiene un **diccionario de reubicabilidad**, que tiene una lista de referencias de campos en memoria que van a hacer necesarias que alguien más las traduzca en una dirección real efectiva. Esto sirve para transformar las direcciones relativas en direcciones reales.

Código objeto

- Estructura interna



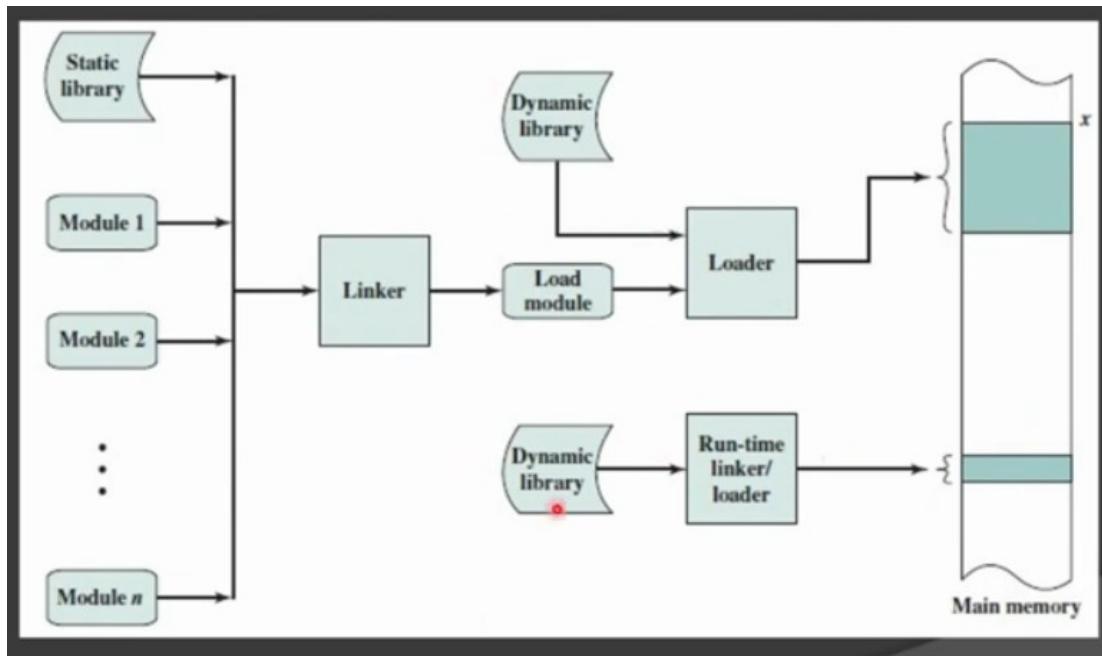
Los códigos objetos están estandarizados bajo un formato específico:

Código objeto

- Formatos estandarizados
 - OMF (Object Module Format)
 - COFF (Common Object File Format)
 - ELF (Executable and Linkable Format)

Link editor

La forma en la que el link editor trabaja esta resumido de esta forma. A la izquierda tengo cada código objeto (modules). Por lo general son varios. El linker agarra cada uno y carga cada modulo, generando el “.exe”. Después a partir de este ejecutable (Load module) tengo opciones como el ensamblado dinámico. El loader tiene una rutina del sistema operativo, que es lo que hace que el sistema operativo transforme el código ejecutable como un proceso dentro de la memoria ram.



El linker es un programa, un software. Un programa que combina todos los código objeto y genera un único archivo ejecutable. El loader no es un programa, si no que es una rutina del sistema operativo que copia el ejecutable a la memoria principal, convirtiéndolo en un proceso.

¿Por qué es necesario todos estos pasos?. Lo que da sentido a separar el ensamblado / compilado de la generación del programa ejecutable es la existencia de las direcciones externas y la reubicabilidad del código.

Las direcciones externas referencian a todo aquello que yo no programe y con lo que aún así interactuo. Todo eso es código pre-existente. Todo esto lo puedo usar por una invocación a partir de dicha dirección.

La posición relativa dentro del código es necesaria ya que dependiendo del sistema operativo lo que yo vaya a buscar puede estar ubicado en otro lado.

Dos problemas a resolver

- Direcciones externas
 - Existen direcciones en el código objeto que no se pueden resolver en tiempo de ensamblado
- Reubicabilidad
 - ¿Por qué es necesaria?
 - No se sabe que otro programa habrá en memoria a la vez
 - Swap a disco en un entorno de multiprogramación

Linking

El linking tiene como entrada los códigos objetos y como salida el ejecutable. En principio existen dos modos:

○ Linking

- Estático (linkage editor)
- Dinámico
 - Load time dynamic linking
 - Run time dynamic linking

En el lindeo estatico, cada módulo objeto que se crea tiene una referencia relativa al inicio del módulo. El link editor combina todos los modulos objetos y genera el código ejecutable que tiene todas las direcciones relativas, pero ya no en relación a cada modulo por separado, si no que ya se hace referencia a un único load module reubicable con todas las referencias relativas de cada modulo por separado.

¿Como resuelve el link editor la generación de un único load module en el link estatico? Primero crea una tabla con todos los módulos objetos y sus respectivas longitudes. Luego declara una dirección base a cada modulo (arranca en 100 porque primero tengo un header)

○ Linking estatico

- Generación del load module

Module	Length	Starting address
A	400	900
B	600	500
C	500	1100
D	300	1600

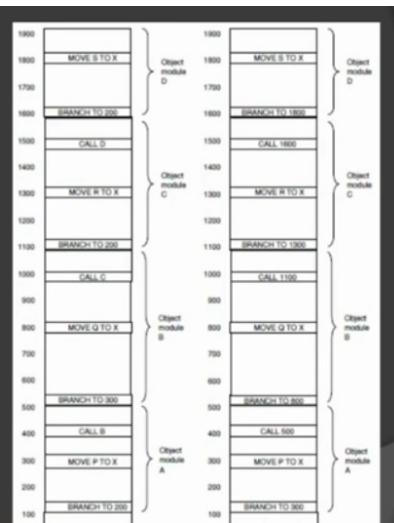


Figure 7.14. (a) The object modules of Fig. 7-13 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed.

Una vez que genera esta tabla, el link editor va a buscar todas las referencias a memoria y deberá sumarle **una constante de reubicación** igual a la dirección de inicio del módulo objeto. Esto es

necesario ya que saltar a una instrucción cuando había un módulo solo no es lo mismo que hacerlo cuando esta unido a todos los demás. Para hacer esto utiliza el diccionario de reubicabilidad.

Finalmente, el link editor resuelve las referencias a las instrucciones u otros procedimientos externos e inserta su correspondiente dirección en mi tabla de direcciones externas en mi código objeto.

El link dinámico yo puedo generar un ejecutable que no tenga todo el código binario para correr, si no que puede aparecer luego, por ejemplo en tiempo de carga o en tiempo de ejecución. Es decir, en principio tengo referencias externas sin resolver que el loader identifica.

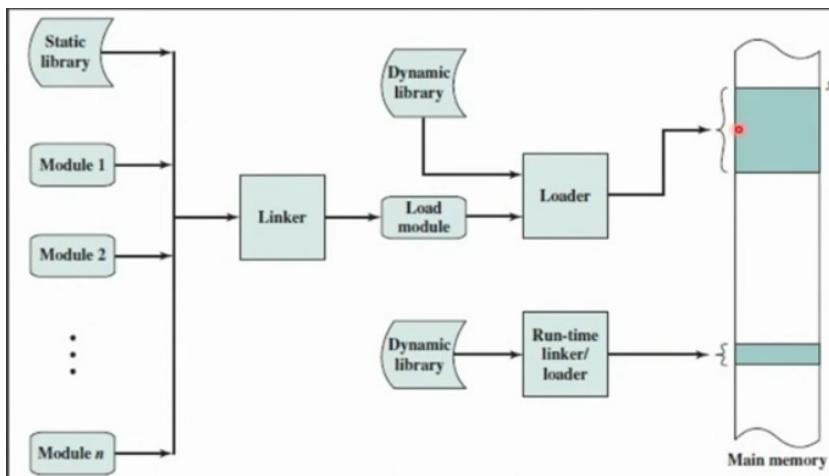
Linking dinámico en tiempo de carga

En tiempo de carga lo que hace el loader es combinar aquello generado por el load module y la librería dinámica, de modo que cuando se hace referencia a los módulos estáticos a alguna librería externa, cuando sea el tiempo de carga por ejemplo sabrá donde ubicar dicho llamado externo gracias a la referencia que tiene dentro del header de cada código objeto:

- **Linking dinámico**
 - Load time dynamic linking
 1. Se levanta a memoria el load module
 2. Cualquier referencia a un módulo externo hace que el loader busque ese módulo, lo cargue y cambie la referencia a una dirección relativa desde el inicio del load module
 - Ventajas
 - Facilita la actualización de versión del módulo externo porque no hay que recomilar
 - El sistema operativo puede cargar y compartir una única versión del módulo externo
 - Facilita la creación de módulos de lindeo dinámico a los programadores (ej. Bibliotecas .so en Unix)

Linking dinámico en tiempo de ejecución

Acá el ejecutable tiene una referencia externa pero ni siquiera se carga al loader. Esta referenciado de manera que recién sobre la memoria es que se carga la información. El sistema operativo es el encargado de reconocer dicha marca y va a ir a buscar a una biblioteca de modulos dinámicos y lo va a ubicar dentro de la memoria ram.



○ Linking dinámico

- Run time dynamic linking
 - Se pospone el lindeo hasta el tiempo de ejecución
 - Se mantienen las referencias a módulos externos en el programa cargado
 - Cuando efectivamente se invoca al módulo externo, el sistema operativo lo busca, lo carga y linkea al módulo llamador.
 - Ventajas
 - No ocupo memoria hasta que la necesito (ej. Bibliotecas DLL de Windows)

Proceso de Loading

Este proceso hoy en dia está generalizado en el loading dinámico en tiempo de ejecución. Este, cuando el loader carga el programa a memoria lo deja con sus correspondientes direcciones relativas y lo que ocurre es una traducción por linea y constantemente de las direcciones reales en absolutas. Es decir, en tiempo de ejecución, un componente de hardware (MMU – Memory Management Unit) se encarga de hacer la traducción de lo relativo a lo absoluto.

Después existen otras implementaciones donde todas las direcciones no se cargan relativas a un cero, si no que se hace a un registro de la máquina (loading por registro base). Al registro base se le asigna el valor de donde está cargado el programa y a partir de este todas las direcciones quedan resueltas tal que las direcciones relativas al inicio se mantendrían.

El loading absoluto y reubicable son mas obsoletos. El primero, las direcciones reales están prefijadas desde un inicio. El segundo cambia las direcciones en el momento de carga pero no era tan estable.