



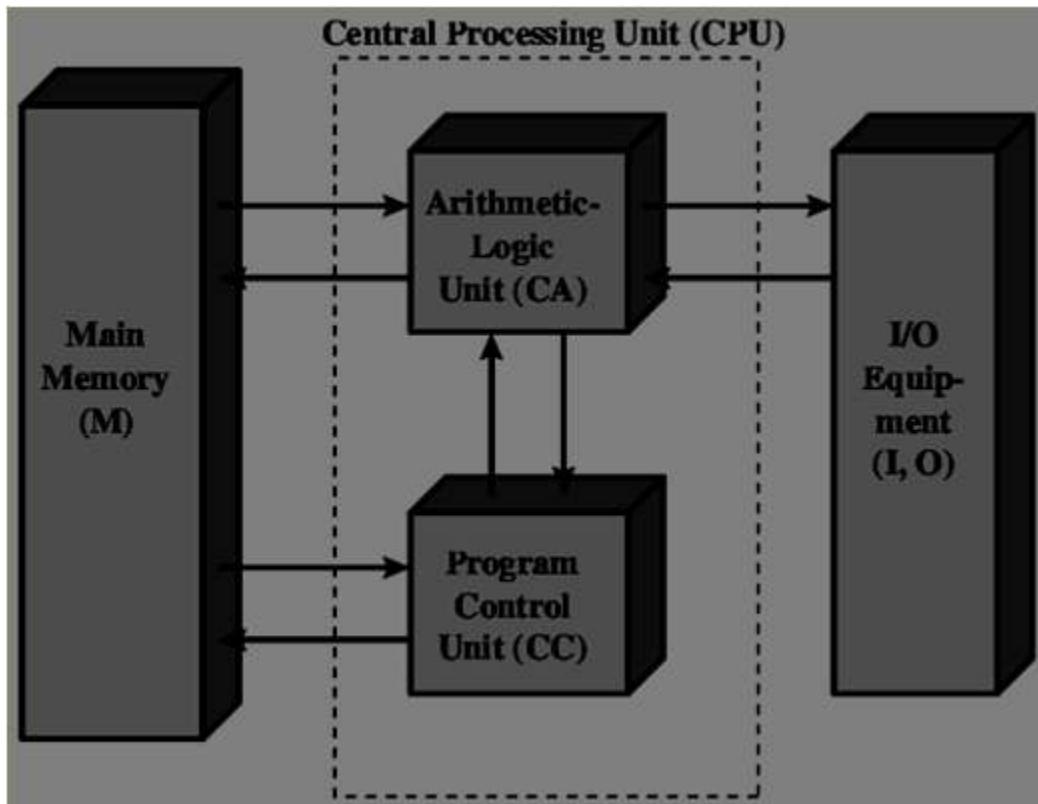
## Organizacion de Computadoras Resumen

Organizacion De Computadoras (Universidad Nacional del Sur)

# Diapositiva 1

Arquitectura Von Newmann se basa en 4 componentes principales:

- Un CPU (unidad central de proceso), el cual cuenta con su respectiva ALU (unidad aritmético - logica)
- La memoria principal
- Uno o mas dispositivos de entrada y salida
- Un componente de control que orquesta la interaccion entre los restante componentes



La invención del transistor permitió construir computadoras tan diferentes a las anteriores que se habla de distintas generaciones.

- La primera generación está compuesta por las computadoras construidas principalmente por válvulas
- La segunda generación está compuesta de las nuevas computadoras construidas usando transistores
- La tercera generación está compuesta por las computadoras construidas usando circuitos integrados

Niveles de integración (scale integration):

- Usando integrados SSI y MSI (1965 - 1971)
  - SSI hasta 100 transistores por chip

- MSI de 100 a 3.000 transistores por chip
- Usando integrados LSI (1971- 1977)
  - De 3.000 a 100.000 transistores por chip
- Usando integrados VLSI (1978- 1991)
  - De 100.000 a 100.000.000 transistores por chip
- Usando integrados ULSI (1991- 2011)
  - Más de 100.000.000 de transistores por chip

**IBM system / 360:** Esta computadora introducida en 1964 constituye la prier familia de computadoras, donde todas tienen un set de instrucciones similares o incluso idénticos y usan el mismo sistema opetarivo. La principal diferencia entre las distintas versiones de la famlia es su velocidad y su capacidad de memoria. Este sistema popularizo el concepto de que un byte tiene 8 bits

**Dec PDP-8:** DEC creo en 1964 la primera mini computadora, el nombre hace referencia a otra invención contemporánea “la mini falda”. No requeria un equipo de aire acondicionado propio. Era lo suficientemente pequeña como para entrar en un escritorio. Su precio era mucho mas inferior, \$16.000 comparados con 100.000 que salía la sytem 360 de IBM. La PDP-8 se caracterizó por hacer uso de un innovativo bus único llamado ómnibus, el cual interconecta a todos sus componentes

**Ley de Moore:** Godon Moore, uno de los fundadores de Intel observo que la densidad de los circuitos integrados crecia constantemente. Predijo que la cantidad de transistores en los chips iba a duplicarse cada año. Tan precisa fue esa predicción que hoy se la conoce como “Ley de Moore”

**Beneficios de mayor cant de transistores por chip:** los transistores de los integrados CMOS cuanto mas pequeños resultan mas veloces y disipan menos calor. Al ser mas pequeños todos los componentes, sus interconexiones son mas cortas, y por ende pueden funcionar a mayor velocidad. El costo de frabricar un chip se a mantenido relativamente constante, por lo que al aumentar el nivel de integración baja el costo (i.e. pagando lo mismo podemos hacer mas que antes)

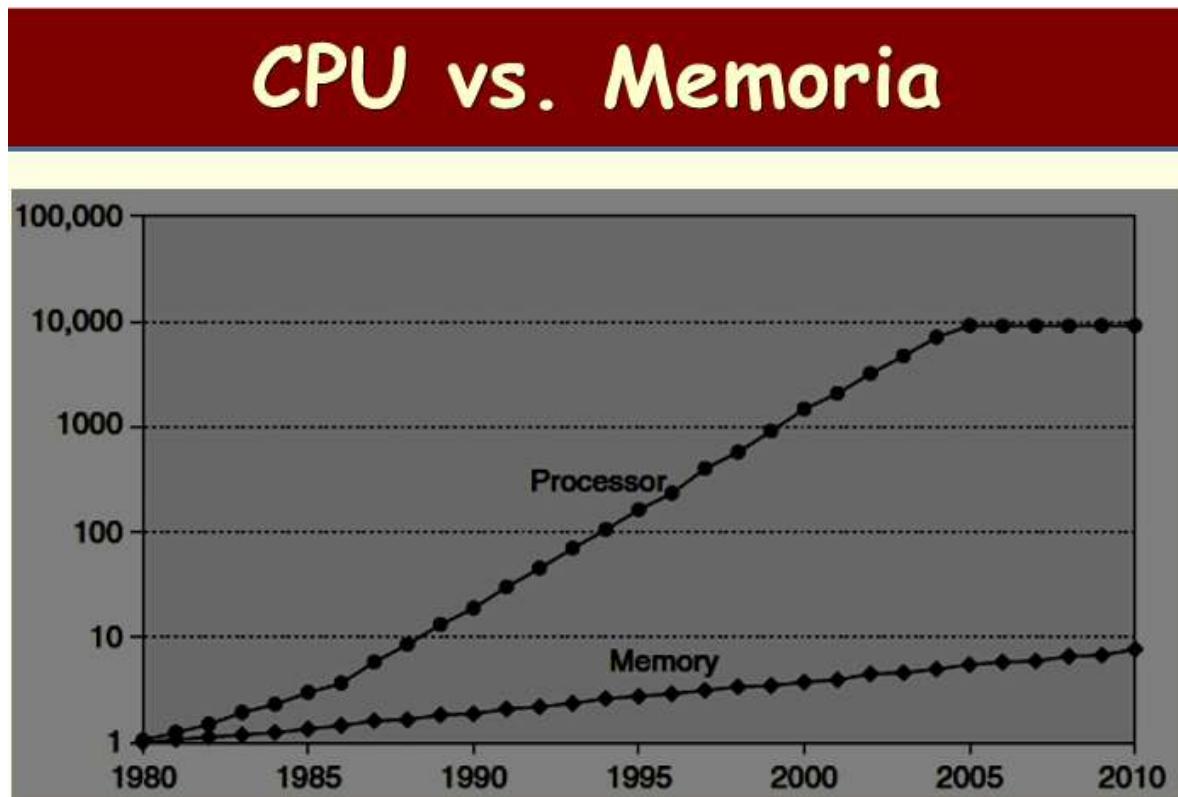
#### Evolución de la memoria:

la tecnología de la memoria principal de las primeras computadoras era bien diversa. Una de las alternativas mas populares de la época (1955-1975) fue la memoria de nucleo, donde cada bit en un pequeño toroide magnético. Se trata de una memoria persistente, pero el contenido se destruye al leerlo. Era fabricado solo por mujeres.

En la década del 70 finalmente se pudo aplicar la tecnología de los circuitos integrados a la producción de memoria. La tecnología de los semi conductores permitio empaquetar unos 256 bits en el espacio ocupado por un único toroide de la memoria de nucleo. Las lecturas no eran destructivas y además el ciclo de lectura era mucho mas veloz, esos si, se trata de una memoria dinámica, el contenido se pierde pasado un cierto tiempo.

Una vez implementada la memoria dentro de un circuito integrado, se pudo sacar provecho de los avances en el nivel de integración: la capacidad de los mudulos de memoria, se duplica cada un año. No obstante, solo la capacidad de los modulos se incrementa de forma exponencial, no es asi

su velocidad. Existe una separación cada vez mayor entre las velocidades del procesador y de la memoria.



A lo largo del tiempo se han ensallado distintas alternativas para atemperar el impacto de esta creciente diferencia de desempeño:

- Traer mas información a la vez (que los modulos de memoria tengan mas "patitas")
- Incorporar multiples niveles de memoria caché
- Minimizar los accesos de memoria (tarea tanto del programamdr como del compilador)
- Mejorar la interconección entre cpu y memoria

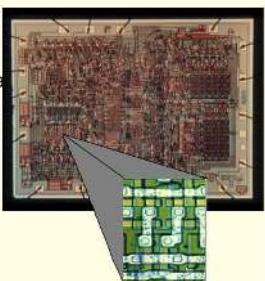
### Intel

Intel introdujo en 1971 el i4004, el primer microprocesador. Un microprocesador contiene todos los componentes del cpu dentro de único chip. Se trata de una arquitectura de 4 bits. En 1972 introdujo el i8008, de 8 bits. Tanto el i4004 como el i8008 eran de propósito específico. Luego en 1974 introdujo el i8080, el primer microprocesador de propósito general

## Intel 4004 (1971)

### Características:

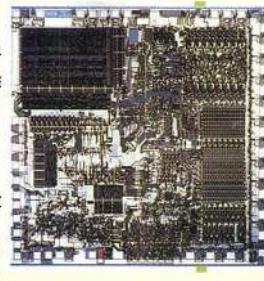
- Arquitectura de 4 bits.
- Capacidad de memoria: 4 KB (12 bits)
- Transistores: 2.250
- Tamaño: 12 mm<sup>2</sup>
- Frecuencia: 108 KHz



## Intel 8086 (1978)

### Características:

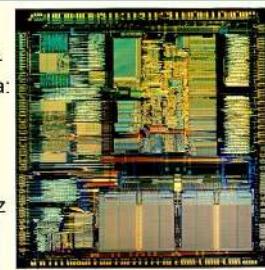
- Arquitectura de 16 bits.
- Capacidad de memoria: 1 MB (20 bits)
- Transistores: 29.000
- Tamaño: 23 mm<sup>2</sup>
- Frecuencia: 5.000 KHz



## Intel 80386 (1985)

### Características:

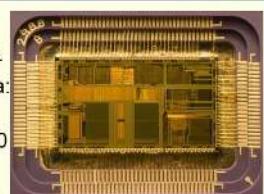
- Arquitectura de 32 bits.
- Capacidad de memoria: 4 GB (32 bits)
- Transistores: 275.000
- Tamaño: 104-39 mm<sup>2</sup>
- Frecuencia: 12-33 MHz



## Intel 80486 (1989)

### Características:

- Arquitectura de 32 bits.
- Capacidad de memoria: 4 GB (32 bits)
- Transistores: 1.200.000
- Tamaño: 81 mm<sup>2</sup>
- Frecuencia: 16-100 MHz
- Cache: 8 KB



Incluye por primera vez un **coprocesador matemático** en el mismo chip.

**Coprocessador matemático:** El coprocesador matemático es un procesador especial que sirve como complemento del microprocesador principal.

El coprocesador matemático puede encargarse de operaciones como la aritmética de punto flotante, gráficos, procesamiento de señales, procesamiento de cadenas, encriptación, del filtro de Savitzky–Golay (método para cálculo de derivadas), etc.

Por lo tanto, el coprocesador no es un procesador de propósito general. Algunos coprocesadores no pueden buscar instrucciones desde la memoria, ejecutar instrucciones de control de flujo, hacer operaciones de entrada/salida, administrar la memoria, entre otras cosas, que sí pueden hacer los procesadores de propósito general.

El coprocesador depende de un procesador anfitrión o "host" para entregarle instrucciones al coprocesador.

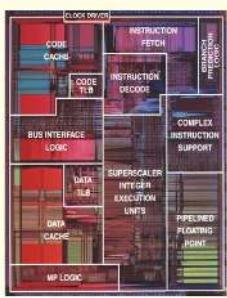
De todas maneras, en algunas arquitecturas, el coprocesador tiene un funcionamiento más de propósito general, pero con un limitado rango de funciones y siempre bajo la supervisión del procesador principal.

El uso de coprocesadores disminuyó debido a la dificultad de integrar este con los nuevos microprocesadores de altas velocidades. De todas maneras hay un resurgimiento de estos, especialmente para aquellos dedicados a los gráficos, que cada vez son más complejos en los juegos.

## Intel Pentium (1993)

### Características:

- Arquitectura de 32 bits.
- Capacidad de memoria: 4 GB (32 bits)
- Transistores: 3.100.000
- Tamaño: 294 mm<sup>2</sup>
- Frecuencia: 60-233 MHz
- Cache: 16-32 KB
- Primer arquitectura x86 superescalar.

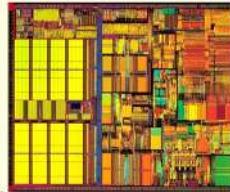


Organización de Computadoras - Ing. A. G. Stankewiclus 48

## Intel Pentium III (1999)

### Características:

- Arquitectura de 32 bits.
- Capacidad de memoria: 4 GB (32 bits)
- Transistores: 9.500.000
- Tamaño: 125 mm<sup>2</sup>
- Frecuencia: 450-1400 MHz
- Cache: 256-512 KB



Organización de Computadoras - Ing. A. G. Stankewiclus 49

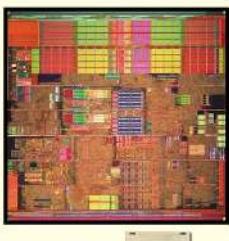
**Arquitectura superescalar:** Superescalar es el término utilizado para designar un tipo de microarquitectura de procesador capaz de ejecutar más de una instrucción por ciclo de reloj. El término se emplea por oposición a la microarquitectura escalar que sólo es capaz de ejecutar una instrucción por ciclo de reloj.

La microarquitectura superescalar utiliza el paralelismo de instrucciones además del paralelismo de flujo, éste último gracias a la estructura en pipeline. La estructura típica de un procesador superescalar consta de un pipeline con las siguientes etapas: FETCH - DECODE - EFFECTIVE ADDRESS – EXECUTE – MEMORY – WRITE BACK.

## Intel Pentium IV (2000)

### Características:

- Arquitectura de 32 bits.
- Capacidad de memoria: 4 GB (32 bits)
- Transistores: 42.000.000
- Tamaño: 145 mm<sup>2</sup>
- Frecuencia: 1.3-3.8 GHz
- Cache: 256-2048 KB
- Primera implementación de la técnica de **hyperthreading**.

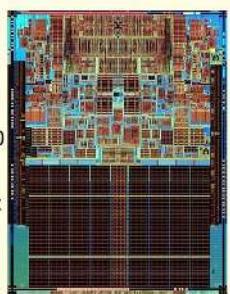


Organización de Computadoras - Ing. A. G. Stankewiclus 50

## Intel Core2 (2006)

### Características:

- Arquitectura de 64 bits.
- Capacidad de memoria: 64 GB - 8 EB (36-64 bits)
- Transistores: 291.000.000
- Tamaño: 143 mm<sup>2</sup>
- Frecuencia: 1.06-3.5 GHz
- Cache: 1-12 MB
- Núcleos: 1, 2 ó 4.



Organización de Computadoras - Ing. A. G. Stankewiclus 51

**HyperThreading** (también conocido como H Technology) es una marca registrada de la empresa Intel para denominar su implementación de la tecnología Multithreading Simultáneo también conocido como SMAT. Permite a los programas preparados para ejecutar múltiples hilos (multi-threaded) procesarlos en paralelo dentro de un único procesador, incrementando el uso de las unidades de ejecución del procesador.

Esta tecnología consiste en simular dos procesadores no lógicos dentro de un único procesador imaginario. El resultado es una mejoría en el rendimiento del procesador, puesto que al simular dos procesadores se pueden aprovechar mejor las unidades de cálculo manteniéndolas ocupadas durante un porcentaje mayor de tiempo. Esto conlleva una mejora en la velocidad de las aplicaciones que según Intel es aproximadamente de un 60%.

La tecnología HyperThreading tiene grandes capacidades de procesamiento y rapidez. Algunas de sus ventajas son: mejora el apoyo de código “multi-hilos”, que permite ejecutar múltiples hilos simultáneamente, mejora de la reacción y el tiempo de respuesta.

De acuerdo con el primer informe de Intel, los Pentium 4 que incorporan esta tecnología tienen un rendimiento entre un 15% y un 30% superior al de los procesadores sin HyperThreading, y utilizan sólo un 5% más de recursos.

## Intel Core i7 (2008)

**Características:**

- Arquitectura de 64 bits.
- Capacidad de memoria: 64 GB - 8 EB (36-64 bits)
- Transistores: 1.600.000.000
- Tamaño: 160 mm<sup>2</sup>
- Frecuencia: 1.06-3.5 GHz
- Cache: 5-15 MB
- Núcleos: 2, 4 ó 6.



Organización de Computadoras - Mg. A. G. Stankevicius 52

**Principio de equivalencia:** el principio de equivalencia nos brinda un puente entre el hardware y el software, “todo lo que puede ser implementado a nivel de software también puede ser re implementado a nivel de hardware y todo lo que puede ser implementado a nivel de hardware puede ser re implementado de nivel de software”

La decisión de qué implementar en hardware y qué en software depende de otros parámetros, como la velocidad, el costo, facilidad de mantenimiento, etc.

**Microprogramación:** Una aplicación inmediata de la ley de equivalencia se puede observar, por caso, en las mini computadoras de la compañía Deck. En ocasiones, las instrucciones maquina más complicadas se implementaban en un software de bajo nivel llamado micro código. Parte de la instrucciones estaban implementadas directamente en hardware, pero parte también se implementaban también a través de microprogramación. La idea era balancear costo versus desempeño. La técnica de microprogramación floreció principalmente durante el periodo que la memoria principal era excesivamente lenta. El contar con un set de instrucciones complejo permite generar programas mas compactos. Y con menor requerimiento de ancho de banda a memoria. Este tipo de arquitectura se denomina casualmente complex instruction set computer (CISC). La familia intel x86 es quizás el ejemplo mas conocido de arquitectura CISC. La utilidad de la microprogramación empezo a ceder de la mano de dos avances:

- Por un lado el crecimiento exponencial de la cantidad de transistores disponibles permitió re implementar en hardware mas de una funcionalidad.
- Pero el mayor impacto vino de la mano de la mejora en el desempeño de la memoria principal

En la actualidad se opta por simplificar el set de instrucciones (RISC), mejorando el desempeño mediante el uso de pipe line y multiples nucleos,

### Ciclo básico del CPU

El ciclo básico de CPU consta de 6 etapas :

FETCH: se busca la próxima instrucción a ser ejecutada, la cual es apuntada por el registro pc.

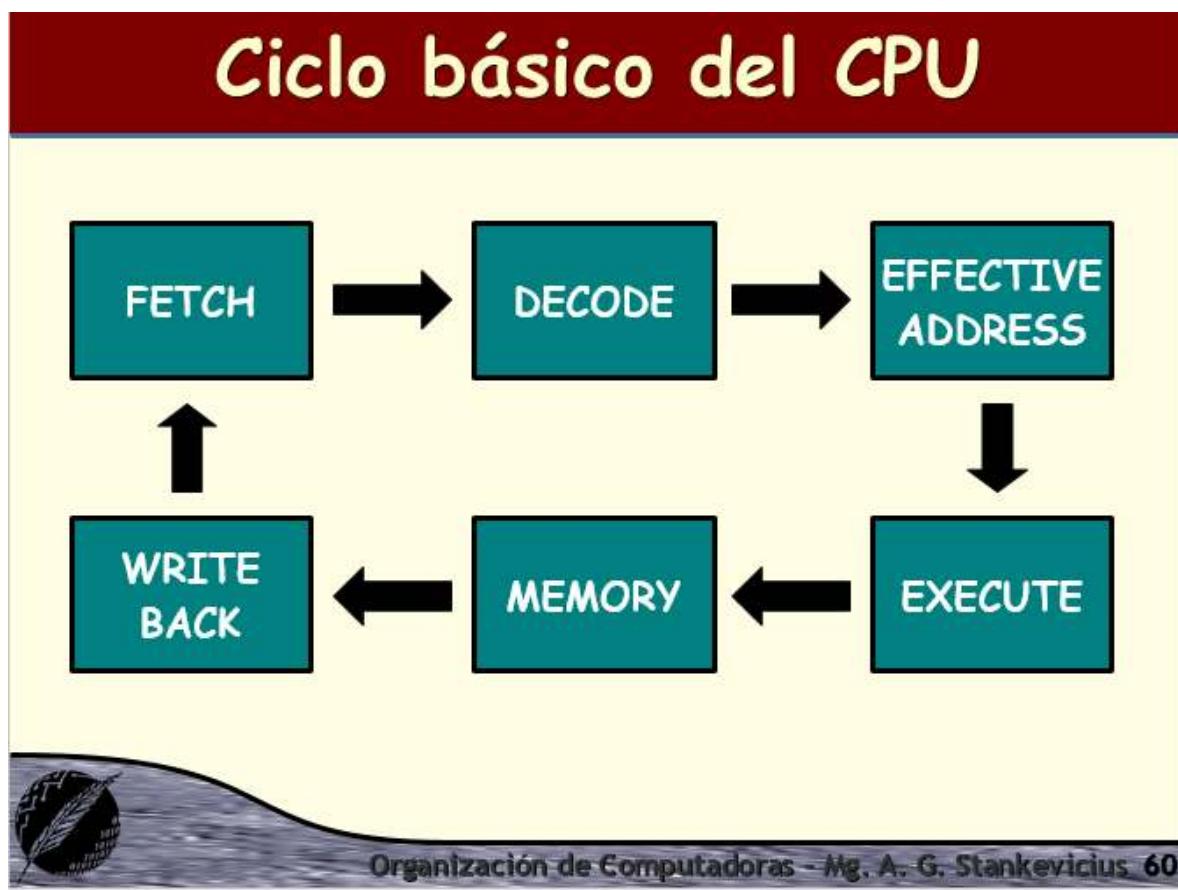
DECODE: luego, se determina de qué tipo de instrucción se trata.

EFFECTIVE ADDRESS: se calcula la dirección efectiva referida por la instrucción (si es que alguna) y/o se busca en memoria los operandos para poder ejecutar la instrucción.

EXECUTE: sabiendo de qué instrucción se trata y contando con los operandos que se necesiten se puede enviar todo a la ALU para que sea ejecutado.

MEMORY: las arquitecturas que no permiten acceder a operandos en memoria cuantan con una etapa específica donde se accede a memoria para leer o escribir una determinada locación

WRITE BACK: usualmente el resultado de la operación se almacena en algunos de los registros de la máquina durante esta etapa.



### Organización de la memoria:

La memoria se organiza como un arreglo de  $n$  ubicaciones, cada una de  $K$  bits. Cada ubicación cuenta con un identificador que la caracteriza denominado dirección. De igual forma cada ubicación

almacena un contenido de K bits. Las operaciones básicas con la memoria son leer un valor de memoria y escribir un valor de memoria.

#### **Interfaz con memoria:**

La unidad controladora de la memoria cuenta esencialmente con dos registros:

- El registro MAR el cual almacena la dirección de una locación de memoria
- El registro MDR el cual almacena el contenido de una locación de memoria

Operación de lectura (LOAD): el procesador escribe la locación que se desea acceder en registro MAR, luego se activa la señal de lectura y finalmente se accede al valor en cuestión el cual estará disponible en el registro MDR.

Operación de escritura (STORE): el procesador escribe la locación que se desea acceder en el registro MAR y el contenido que se desea almacenar en el registro MDR y por último activa la señal de escritura

## **Diapositiva 2**

### **Aritmetica en computadoras**

Las computadoras se usan en esencia para computar. Una de las acepciones de computar es precisamente calcular. Un pre-requisito para poder calcular es contar con un sistema de representación y un conjunto de operaciones definidas sobre ese sistema, es decir, necesitamos definir una aritmética.

Sea  $M$  el conjunto de números representables en una cierta computadora (observe que  $M$  es un subconjunto estricto de  $\mathbb{R}$ , puesto que su cardinalidad es finita) es posible definir una aritmética adecuada para computadoras sobre  $M$  ( $G: M \times M \rightarrow M$ ). Esta aritmética se denomina aritmética real aproximada.

Relación aritmética real con aritmética maquina: la función maquina  $G$  se relaciona con la real  $F$  a través de la función compuesta  $G: H \circ P$ . A su vez, el mapeo  $H$  se define como  $H = F_{M \times M} \rightarrow R$ , esto es la función real  $F$  restringida al dominio maquina  $M \times M$ . Por otra parte la función  $P$  debe tener como dominio e imagen los conjuntos  $P: R \rightarrow M$ , esta función representa el esquema de redondeo a ser aplicado. El esquema de redondeo tiene por objeto decidir cuál es el número máquina que corresponde asociar a cada número real

### **Sistema de representación**

La definición precisa de estas funciones y la implementación de las operaciones aritméticas dependen de la representación interna que se adopte para los números. En otras palabras, la elección del sistema de representación afecta tanto al diseño del hardware como del software. El diseñador del sistema tiene que tener en cuenta tanto requerimientos de desempeño como de precisión y de capacidad de representación.

En función del objetivo que se priorice se han ensayado distintas alternativas:

- Sistemas numéricos con base: este es el sistema tradicional al que estamos acostumbrados. Los números se representan fijando una base y haciendo uso de un conjunto de dígitos. Por caso, para una base  $b$  se utilizan los  $b$  dígitos comprendidos en el rango  $[0..b-1]$ . El aporte de cada dígito al valor representado tiene en cuenta la posición del mismo. Cada número tiene exactamente la misma representación dentro del sistema.
- Sistema numéricos de dígitos signados: el sistema dígito signado es una extensión del sistema anterior, en la cual se incorpora un signo a nivel de cada dígito. Por caso, para una base  $b$  se admite ahora el conjunto extendido de dígitos  $[-b+1, \dots, 0, \dots, b-1]$ . Se puede señalar como aspecto negativo que es posible encontrar múltiples representaciones para un determinado valor numérico. Es decir, se trata de un sistema de representación redundante. Este sistema sigue siendo un sistema posicional.
- Sistema numérico de residuo: el sistema de residuo es un sistema no posicional, donde el aporte de cada dígito al valor representado es independiente de su posición. Un valor numérico se representa a través de una tupla  $X=(R_1, R_2, \dots, R_n)_M$ , la cual depende de otra tupla  $M=(M_1, M_2, \dots, M_n)$ . Cada uno de los dígitos  $R_i$  del número  $X$  representan el residuo que se obtiene al calcular  $X \bmod M_i$ . Como restricción adicional,  $M_i$  tiene que ser elegido de forma que no resulten primos entre sí. En este sistema no existe el concepto de acarreo ya que los dígitos no tienen relación entre sí, por lo que es posible procesar cada uno de manera independiente del resto (ej: en el sistema residuo  $M=(5,3)$  el número 13 se representa  $(3,1)_M$ , ya que  $13 \bmod 5 = 3$  y  $13 \bmod 3 = 1$ )
- Sistema número racional: en el sistema número racional se representan los valores numéricos mediante fracciones. Cada fracción está compuesta por dos valores enteros, un numerador y un denominador. Las operaciones básicas entre fracciones dan siempre como resultado otra fracción, la cual no necesita apelar a una cantidad infinita de dígitos. Naturalmente, tanto el numerador como el denominador pueden crecer rápidamente. Este es un sistema redundante ya que  $\frac{1}{2}$  y  $\frac{2}{4}$  representan el mismo valor.
- Sistema logarítmico: el sistema logarítmico se basa en adoptar una magnitud real  $N > 1$  como base y representar los distintos valores numéricos a través de sus respectivos logaritmos. Una determinada base  $N$  induce el espacio logarítmico  $L_N$ , el cual se define como  $L_N = \{x : |x| = N^i, i \in \text{Nat}\} \cup \{0\}$ . Al hacer uso de una representación logarítmica, se simplifica notablemente alguna de las operaciones.

### Representación de punto fijo

Elegido uno de los sistemas de representación podemos profundizar su tratamiento:

- En este sistema fijada una base y un conjunto de dígitos, el aporte de cada dígito al valor representado depende de su posición.
- La idea es que los primeros  $n$  dígitos denotan la parte entera y los restantes  $k$  dígitos la parte fraccionaria.
- La elección de  $n$  y de  $k$ , decisión que toma el diseñador del hardware, fija de alguna manera la posición del punto decimal. Por ende, este tipo de representación se denomina de punto fijo (FPX).

La naturaleza posicional del sistema se evidencia al considerar el valor representado por una tupla de dígitos: Sea  $X = (d_{n-1}, \dots, d_1, d_0, d_{-1}, \dots, d_{-k})_r$  un número representado en el sistema con base  $r$ . como

se trata de un sistema posicional, el valor representado por el numero  $x$  se calcula como el producto interno  $X^*W$ , donde  $W$  denota al conjunto de pesos asociados a cada posición, esto es,  $(r^{n-1}, \dots, r^1, r^0, r^{-1}, \dots, r^{-k})$

### Elección de la base

Para un dado sistema con base  $r$ , el disponer de  $n$  dígitos impacta lo siguiente:

- La precisión del sistema, es decir, la cantidad de números representables se calcula como  $N = r^n$
- La ineficacia en el uso de espacio, es decir, la cantidad de símbolos distintos que se deben almacenar se cuantifica como  $E=n*r$

Para determinar cual es la mejor base se debe primero fijar la precisión para luego minimizar la ineficacia en el uso del espacio. En otras palabras, queremos determinar qué  $r$  minimiza a  $E$  para un  $N$  constante. Despejando  $n$  en la ecuación  $N = r^n$  nos queda que  $n = \ln(N)/\ln(r)$ . usando el resultado anterior, queremos minimizar la expresión  $E= n*r = r * \ln(N)/\ln(r)$ . derivando con respecto a  $r$  nos queda  $dE/dr = \ln(N*(\log(r)-1))/\ln(r^2)$ . Igualando a cero la solución es  $r = e$ . como la base tiene que ser entera,  $r = 3$  sería la mejor opción

### Conversión entre bases

Es posible convertir un numero de una base a otra mediante 2 métodos:

- Método de la multiplicación
- Método de la división

A su vez también se deben distinguir entre la conversión de números enteros o de números fraccionarios. En definitiva, debemos contemplar 4 métodos de conversión entre bases

### Conversión de enteros

- Método de la multiplicación: se opera usando la aritmética destino (base  $r_d$ ). idea es hacer uso de la naturaleza posicional del sistema, calculando el producto interior  $|X| = \sum_{i=0}^n (x_i * r_o^i)$
- Método de la división: se opera usando la aritmética origen (base  $r_o$ ). Una vez mas la idea es hacer uso de la naturaleza posicional del sistema teniendo en cuenta que los exponentes en esta ocasión son negativos. Los dígitos se calculan sucesivamente dividiendo por la base destino hasta llegar a un cociente nulo.  $X = Q_0 * r_d + X_0$ , con  $X_0 = |X| \bmod r_d$ . Luego,  $Q_0$  es igual a  $Q_1 * r_d + X_1$ , con  $X_1 = |Q_0| \bmod r_d$  y así sucesivamente hasta que  $Q_i$  es igual a cero

**Conversión de fracciones:** Todo número entero representable con una cantidad finita de dígitos en una cierta base tendrá una representación con una cantidad también finita de dígitos en cualquier otra base. En contraste, en los números fraccionarios es posible que un número representado con una cantidad finita de dígitos en una dada base requiera una cantidad infinita en otra. No obstante, siempre es posible calcular los primeros  $k$  dígitos como su representación aproximada

- Método del producto: se opera usando la aritmética origen. La idea es ir obteniendo los distintos dígitos  $x_i$  multiplicando la fracción por la base  $r_d$ . el primer dígito  $x_1$  se obtiene

como  $X_{-1} = \text{piso}(|X| * r_d)$ . Luego, sea  $\sim X_{-1}$  la parte fraccionaria  $|X| * r_d$ , los restantes dígitos se deben calcular como  $X_{-2} = \text{piso}(\sim X_{-1} * r_d)$  y así sucesivamente

- **Método de la división:** se opera usando la aritmética destino. Es análogo al método del producto para los enteros, sacamos provecho de la naturaleza posicional del sistema calculando el producto interior  $|X| = \sum_{i=1}^n (X_i * r_d^{-i})$

Los humanos hemos de preferir naturalmente operar en base 10, por lo que usaremos cualquier método que nos permita hacer uso de nuestra familiar base. La computadoras ha de hacer algo análogo, prefiriendo toda vez que se pueda la base 2, si bien se intenta evitar hacer un uso excesivo de la operación división ya que se trata de una operación muy costosa a nivel de tiempo de ejecución

## Diapositiva 3

Las operaciones aritméticas se clasifican en tres grandes categorías:

- Operaciones aritméticas estándares
- Funciones aritméticas elementales
- Operaciones pseudo-aritméticas

A su vez se dispone esencialmente de dos modos de operación: punto fijo y punto flotante

**Funciones aritméticas estándares:** esta categoría incluye las cuatro funciones aritméticas primitivas: suma resta multiplicación y división. Toda otra función matemática podrá ser expresada como una composición de estas cuatro operaciones

**Funciones aritméticas elementales:** esta categoría incluye aquellas operaciones usadas frecuentemente en cómputos matemáticos, tales como exponencial, raíz cuadrada, funciones hiperbólicas, etc. No todas las computadoras implementan estas funciones en hardware, por lo que en general se suele implementar en software o en firmware (a nivel de micro código)

**Operaciones pseudo-aritméticas:** Esta categoría incluye operaciones que requieren un cierto grado de cálculo aritmético, pero están relacionadas con la ejecución de un programa. Consta de dos subcategorías:

- Aritméticas de direccionamiento: operaciones relacionadas al cómputo de la dirección efectiva en memoria de los datos.
- Aritméticas de edición de datos: operación lógica y de transformación de datos tales como, load/store, empaquetados/desempaquetados, etc.

### Modos de operación:

**Operación en punto fijo:** este modo de operación es usado en problemas comerciales o cálculos estadísticos y se caracteriza por tener el punto decimal en una posición prefijada. Consta de dos categorías: operación entera, donde los resultados se alinean en el extremo derecho como si el punto decimal ocupara esa posición, y operación fraccionaria, donde los resultados se alinean en el extremo izquierdo como si el punto decimal ocupara esa posición.

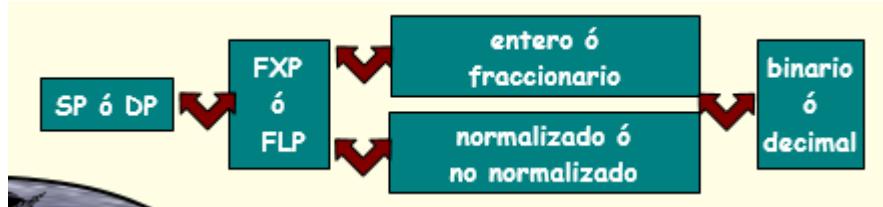
**Operación en punto flotante:** este modo de operación es usado en problemas de tipo ingenieril o científico, donde frecuentemente se requiere escalamiento para manejar tanto magnitudes muy grandes como muy pequeñas. Consta de dos subcategorías: operación normalizada, donde el resultado de toda operación es normalizado antes de ser retornado, y no normalizado, donde se retorna tal cual fue obtenido.

Las operaciones aritméticas también pueden ser clasificadas de acuerdo a su precisión:

- **precisión simple (SP):** se refiere a las operaciones definidas sobre operandos de tamaño estándar, esto es, con operandos de longitud igual a una palabra
- **precisión doble (DP):** se refiere a las operaciones definidas sobre operandos de tamaño doble, esto es, con operandos de longitud igual a dos palabras.
- **Triple cuádruple y restantes precisiones:** pueden definirse de manera análoga

### Binario vs decimal

Algunas arquitecturas ofrecen la posibilidad de operar directamente sobre la base decimal. La conversión (operación de pack y unpack), requiere de instrucciones para manejar los datos directamente en forma decimal. Por caso la instrucción add puede hacer referencia a 16 operación aritméticas distintas, a saber:



Para codificar cada dígito digital es necesario hacer uso de  $k$  bits, donde  $K = \lceil \log_2 10 \rceil = 4$

### Código BCD 2421

El código BCD 2412 denota en su nombre el peso asignado en cada posición en la codificación. Se trata de un sistema posicional. Presenta una línea de simetría por auto-complementación. Esta característica facilita el cálculo de la diferencia entre magnitudes de igual signo y la suma de magnitudes de distinto signo

### Código BCD 8421

El código BCD 8421 codifica cada dígito decimal directamente en binario. También se lo conoce como código bcd natural. Se trata de un sistema posicional. El peso de cada posición coincide con las sucesivas potencias de 2. No es posible encontrar ninguna línea de simetría que resulte de utilidad

### Código BCD exceso 3

El código BCD exceso 3 también codifica cada dígito decimal en binario, si bien le suma previamente un exceso. Producto del exceso deja de ser un sistema posicional. Como se puede apreciar, resulta simétrico por auto complementación. En esta codificación el acarreo binario coincide con el acarreo decimal

## Código Gray

El código gray es simétrico por construcción, también se lo conoce como código progresivo cíclico. Se trata de una codificación no posicional. Como se puede observar presenta múltiples líneas de simetría. Entre la codificación de un dígito y el siguiente se modifica a lo sumo un bit.

2421		8421		----		----	
0	0000	0	0000	0	0011	0	0000
1	0001	1	0001	1	0100	1	0001
2	0010	2	0010	2	0101	2	0011
3	0011	3	0011	3	0110	3	0010
4	0100	4	0100	4	0111	4	0110
5	1011	5	0101	5	1000	5	0111
6	1100	6	0110	6	1001	6	0101
7	1101	7	0111	7	1010	7	0100
8	1110	8	1000	8	1011	8	1100
9	1111	9	1001	9	1100	9	1101

## Aritmética en punto fijo:

Recordemos que elegida una base  $r$  y una precisión  $p$  (cantidad de dígitos), el aporte de cada dígito al valor denotado depende exclusivamente de su posición. La idea es que los primeros  $n$  dígitos denotan la parte entera y los restantes  $k$  dígitos la parte fraccionaria. La lección de  $n$  y de  $k$  tiene como efecto colateral fijar la posición del punto decimal, razón por la cual hablamos de aritmética de punto fijo. Un número signado representará una magnitud positiva o negativa, pero no ambas. Usualmente se reserva el dígito de más a la izq (esto es, el más significativo), para denotar el signo. De los  $r$  dígitos válidos solo se han de utilizar 2 para codificar el signo. Por caso, sea  $x = (d_{n-1}, \dots, d_1, d_0, d_{-1}, \dots, d_{-k})_r$ , un número representado en una base  $r$  entonces  $d_{n-1}$  es 0 cuando  $x \geq 0$  y  $r-1$  cuando  $x < 0$ .

Las posiciones del punto decimal y del signo dependerán de  $n$  y  $k$ . cuando  $k=0$ , el signo se ubica en el extremo izquierdo y el punto decimal en el extremo derecho. Cuando  $n=0$ , el signo se ubica en el extremo izquierdo y el punto decimal inmediatamente a su izq (justo antes del primer dígito fraccionario). En la conversión entre estas variantes es sencillo, basta con multiplicar o dividir por una potencia de  $r$ . La representación interna del punto decimal es implícita, no requiere reservar espacio de almacenamiento. Para representar un número positivo, todos los dígitos salvo el de signo codifican su magnitud. En cambio para representar un número negativo aparecen múltiples alternativas.

## Signo Magnitud:

Esta representación es análoga a la utilizada cotidianamente. La ausencia de signo representa un numero positivo, y la presencia del signo menos representa un numero negativo. En la codificación signo magnitud el digito de signo toma el valor cero en los positivos y  $r-1$  en los negativos. Notece que  $+0$  y  $-0$  denotan el mismo valor, por lo que la representación del cero no es única. Sea  $X'$  el complemento de un dado numero  $X$ , en esta representación  $X'$  y  $X$  difieren solo en el digito que codifica el signo

$$\begin{array}{ll} (00101011)_2 = +(43)_{10} & (10101011)_2 = -(43)_{10} \\ (01111111)_2 = +(127)_{10} & (11111111)_2 = -(127)_{10} \\ (00000000)_2 = +(0)_{10} & (10000000)_2 = -(0)_{10} \end{array}$$

El menor numero posible es  $(r-1\ r-1\ r-1\ \dots\ r-1)_r$ . El mayor numero posible es  $(0\ r-1\ r-1\ \dots\ r-1)_r$ . es decir el rango de reprentacion para signo magnitud es  $[-(r^{n-1}-1), r^{n-1}-1]$

### Complemento a la base

Las computadoras suelen hacer uso de algún esquema de complemento para representar los números negativos. Recordemos que esto simplifica las operación suma de números de distinto signo, o lo que es lo mismo, la resta de numero de igual signo

Supongamos que un odómetro tiene solo 3 digitos, al tratar de retroceder 5 kilómetros estando inicialmente en el kilometro 003, quedamos en el 998. En algún sentido, 998 tiene que representar al numero -2. Notece que al sumar un valor positivo con su complemento negativo, se obtiene siempre el mismo resultado. El resultado anterior permite determinar en general el valor de  $X'$  para cualquier  $X$ . Como comprobamos,  $X+X' = r^n$ , por lo que despejando  $X'$  nos queda  $X' = r^n - |X|$ . no es conveniente tener que computar una resta toda vez que se deseé saber qué valor representa un cierto valor positivo en complemento a la base. Por suerte, como  $r^n = ((r^{n-1}) - |X|) + 1$ . Observece que  $r^{n-1}$  es un numero especial, esta compuesto de  $n$  dígitos iguales, de valor  $r-1$ . En síntesis, el mecanismo simplificado para expresar un cierto numero negativo en complemento a la base consiste en primero expresar el valor absoluto del numero en cuestión en sistema complemento a la base (que por tratarse de numero positivo coincide con su representación en SM).

Complementar cada digito del numero, esto es reemplazar cada digito  $d_i$  por el valor  $(r-1)-d_i$  (el signo también es complementado). Finalmente, incrementar en uno el valor obtenido

$$\begin{array}{ll} (00001111)_2 = +(15)_{10} & (01001101)_2 = +(77)_{10} \\ (01111111)_2 = +(127)_{10} & (00000000)_2 = +(0)_{10} \end{array}$$

$$\begin{array}{ll} (10001111)_2 = & (10000000)_2 = \\ -[(01110000)_2 + 1] = & -[(01111111)_2 + 1] = \\ -(113)_{10} & -(128)_{10} \end{array}$$

El menor numero posible es  $(r-1 \ 0 \ 0 \dots \ 0)_r$  y el mayor numero posible es  $(0 \ r-1 \ r-1 \dots \ r-1)_r$ . Es decir el rango de representación en complemento a la base es  $[-r^{n-1}, r^{n-1}-1]$

#### Complemento a la base disminuida (DRC):

Recordemos que al representar un numero negativo en complemento a la base debemos incrementar en uno luego de complementar. Esta operación puede ser costosa en tiempo de ejecución, sobre todo si existen multiples acarreos. Una posibilidad para evitar esta operación consiste en representar los números negativos haciendo uso de complemento a la base disminuida. Por caso, para un cierto numero positivo X, su complemento a la base disminuido  $X'$  se calcula directamente como  $X' = (r^n - 1) - |X|$ . Revisando el ejemplo del odómetro, el valor representado se debe retrasar una unidad (ya que falta tener en cuenta el incremento final)

$$\begin{array}{cccccccccc} 0003 & \rightarrow & 0002 & \rightarrow & 0001 & \rightarrow & 0000 & \rightarrow & 9999 & \rightarrow & 9998 & \rightarrow & 9998 \\ +3 & \rightarrow & +2 & \rightarrow & +1 & \rightarrow & +0 & \rightarrow & -0 & \rightarrow & -1 & \rightarrow & -2 \end{array}$$

Como se puede apreciar existen nuevamente dos representaciones para el cero:

- $+0 = (0 \ 0 \dots \ 0)_r$
- $-0 = (r-1 \ r-1 \dots \ r-1)_r$

El mecanismo simplificado para expresar un cierto número negativo en complemento a la base disminuida ahora consiste en primero expresar el valor absoluto del numero en cuestión en el sistema complemento a la base disminuido (que por tratarse de un numero positivo coincidirá con su representación en signo magnitud y en complemento a la base). Finalmente, complementar cada digito del número, esto es, remplazar cada  $d_i$  por el valor  $(r-1)-d_i$  (el signo también es complementado).

$$\begin{array}{ll} (00001111)_2 = +(15)_{10} & (01001101)_2 = +(77)_{10} \\ (01111111)_2 = +(127)_{10} & (00000000)_2 = +(0)_{10} \\ (10001111)_2 = & (10000000)_2 = \\ -(01110000)_2 = & -(01111111)_2 = \\ -(112)_{10} & -(127)_{10} \end{array}$$

El menor numero es  $(r-1 \ 0 \ 0 \dots \ 0)_r$  y el mayor  $(0 \ r-1 \ r-1 \dots \ r-1)_r$ . Es decir, el rango de representación en complemento a la base disminuido  $[-(r^{n-1}-1), r^{n-1}-1]$

signo-magnitud	$0000001000100011$	
complemento a 1	$0000001000100011$	$\rightarrow +(547)_{10}$
complemento a 2	$0000001000100011$	
$-(547)_{10}$	$1000001000100011$	signo-magnitud
	$1111110111011100$	complemento a 1
	$1111110111011101$	complemento a 2

### Extencion de signo:

En el ejemplo anterior el numero en binario no contaba con la cantidad requerida de dígitos. El mecanismo para obtener la cantidad deseada de dígitos se denomina extensión de signo. En el caso de representar un numero positivo se completar con ceros mientras que en representar un numero negativo, se debe completar con 1's ya que de esta forma no se altera el valor representado

### Analisis:

En la elección de un sistema de representación de números signados se deben tener en cuenta diversos aspectos:

- Que tan sencillo resulta detectar el signo?: la detección del signo es equivalente en los 3 casos, consiste en inspeccionar el primer digito
- Resultan equivalentes los rango de representación para positivos y negativos: en relación a la simetría del rango, signo magnitud y complemento a la base disminuida resultan simétricos, mientras que complemento a la base no
- El cero tiene una representación unívoca: el cero tiene representación doble en signo magnitud y en complemento a la base disminuido, pero en complemento a la base no
- Que tan eficiente resulta la implementación de las operaciones básicas y de la operación de complementación (suma en SM y RC DRC)

### Suma en SM

Para sumar los números codificados en signo magnitud se aplica el siguiente algoritmo:

- Sean  $X=(X_{n-1} \dots X_1 X_0)$  e  $Y=(Y_{n-1} \dots Y_1 Y_0)$  los números a ser sumados
- Si  $X_{n-1}$  es igual a  $Y_{n-1}$  (numero de igual signo), se suman las magnitudes  $X=(X_{n-2} \dots X_1 X_0)$  e  $Y=(Y_{n-2} \dots Y_1 Y_0)$ . En este caso el signo del resultado será  $X_{n-1}$ . Si hay acarreo en la posición del signo, entonces el resultado es invalido pues se produjo un overflow
- En cambio si  $X_{n-1}$  es distinto de  $Y_{n-1}$  se deben comparar las magnitudes y en caso de que el valor de absoluto de X sea menor que el valor absoluto de Y se proceden a intercambiar los valores X e Y. independientemente de que sea alla intercambiado o no, el resultado buscado se obtiene restando la magnitud  $Y=(Y_{n-2} \dots Y_1 Y_0)$  a  $X=(X_{n-2} \dots X_1 X_0)$ . En este caso el signo de resultado también será  $X_{n-1}$  (notece que será el signo correcto, es decir, el signo del numero mas grande en valor absoluto)

#### → Con operandos de un mismo signo:

$$\begin{array}{r} (0 \textcolor{blue}{0011001})_2 = +(25)_{10} \\ + (0 \textcolor{blue}{1111110})_2 = +(62)_{10} \\ \hline (0 \textcolor{blue}{1010111})_2 = +(87)_{10} \end{array}$$

#### → Con operandos de distintos signo:

$$\begin{array}{r} (0 \textcolor{blue}{1010100})_2 = +(84)_{10} \\ + (1 \textcolor{blue}{1111000})_2 = -(120)_{10} \\ \hline \end{array} \quad \begin{array}{r} (1 \textcolor{blue}{0111000})_2 = -(120)_{10} \\ - (0 \textcolor{blue}{1010100})_2 = +(84)_{10} \\ \hline (1 \textcolor{blue}{0100100})_2 = -(36)_{10} \end{array}$$

### Detección de Overflow:

Al operar en signo magnitud solo se puede producir overflow al sumar números de igual signo. El mecanismo de detección de overflow consiste en inspeccionar el acarreo en la posición del signo: si hubo acarreo se debe descartar el resultado por invalido. En contraste, no es posible provocar un overflow al sumar números de distinto signo. En ese caso el resultado siempre será mas pequeño (en valor absoluto) que el mayor de los operandos (también en valor absoluto)

#### → Con operandos positivos:

$$\begin{array}{r} (0 \ 1010111)_2 = +(87)_{10} \\ + (0 \ 0111110)_2 = +(62)_{10} \\ \hline (1 \ 0010101)_2 = 0V \end{array}$$

#### → Con operandos negativos:

$$\begin{array}{r} (1 \ 10010100)_2 = -(20)_{10} \\ + (1 \ 1111110)_2 = -(126)_{10} \\ \hline (1 \ 0010010)_2 = 0V \end{array}$$

### Suma en DRC

Para sumar dos números en complemento a la base disminuida se usa el siguiente algoritmo:

- Sean X e Y los números a ser sumados y sean S=(S<sub>n-1</sub> ... S<sub>1</sub> S<sub>0</sub>) los dígitos del resultado y C=(C<sub>n</sub> C<sub>n-1</sub> ... C<sub>1</sub>) los acarreos que se generan
- La suma preliminar se obtiene sumando la totalidad de los dígitos, signo incluido
- Si X<sub>n-1</sub> es igual a Y<sub>n-1</sub> es igual a 0 (ambos operandos positivos), C<sub>n</sub> es necesariamente 0, y el resultado preliminar es el definitivo, pero si C<sub>n-1</sub> es igual a 1, se produjo overflow. Si X<sub>n-1</sub> es distinto de Y<sub>n-1</sub> (operando de distinto signo), se estudia el acarreo de salida (C<sub>n</sub>). si C<sub>n</sub> es igual a 0 el resultado preliminar es definitivo. Si C<sub>n</sub> es 1, se descarta el acarreo y se incrementa en 1 el resultado preliminar y se obtiene el definitivo
- Finalmente, si X<sub>n-1</sub> = Y<sub>n-1</sub> = 1 (ambos operandos negativos), C<sub>n</sub> es necesariamente 1, y el resultado definitivo se obtiene incrementando en 1 al resultado preliminar, pero si S<sub>n-1</sub> es igual a 0 se produjo overflow

#### → Con operandos positivos:

$$\begin{array}{r} (00101001)_2 = +(41)_{10} \\ + (0100110)_2 = +(38)_{10} \\ \hline (01001111)_2 = +(79)_{10} \end{array}$$

→ Como era de esperar, C<sub>n</sub> = 0. Por otra parte, como S<sub>n-1</sub> = 0, el resultado preliminar es el definitivo:

$$(01001111)_2 = +(79)_{10}$$

#### → Con operandos de distinto signo:

$$\begin{array}{r} (11010110)_2 = -(41)_{10} \\ + (01100010)_2 = +(98)_{10} \\ \hline (00111000)_2 = +(56)_{10} \end{array}$$

→ En esta oportunidad, C<sub>n</sub> = 1. Se descarta el acarreo de salida, pero se debe incrementar en 1 el resultado preliminar para obtener el definitivo:

$$(00111001)_2 = +(57)_{10}$$

### Detección de overflow:

Al operar en complemento a la base disminuida solo se puede producir overflow al sumar números de igual signo. El mecanismo de detección de overflow consiste en inspeccionar el bit de signo del resultado: en caso de no ser el esperado, se produjo un overflow. Esta detección

equivale a computar  $C_n \text{ xor } C_{n-1}$  al igual que en las restantes representaciones no es posible provocar un overflow al sumar numeros de distinto signo

→ Con operandos negativos:

$$\begin{array}{r} (\textcolor{red}{1}10110111)_2 = -(72)_{10} \\ +(\textcolor{blue}{1}10000000)_2 = -(63)_{10} \\ \hline (\textcolor{blue}{0}1110111)_2 = \textcolor{red}{OV} \end{array}$$

- Como era de esperar,  $C_n = 1$ . No obstante, el signo del resultado es incorrecto, se produjo overflow.
- Nótese que  $C_n \oplus C_{n-1} = 1$ , pues  $C_n = 1$  y  $C_{n-1} = 0$ .

Suma en RC:

Para sumar dos números en complemento a la base se usa el siguiente algoritmo:

- Sean X e Y los números a ser sumados y sean S=(S<sub>n-1</sub> ... S<sub>1</sub> S<sub>0</sub>) los dígitos del resultado y C=(C<sub>n</sub> C<sub>n-1</sub> ... C<sub>1</sub>) los acarreos que se generan
- La suma definitiva se obtiene sumando la totalidad de los dígitos, signo incluido
- Si X<sub>n-1</sub>=Y<sub>n-1</sub>=0, C<sub>n</sub> es necesariamente C, pero si S<sub>n-1</sub> es igual a 1 (el signo del resultado es incorrecto), se produjo overflow
- Si X<sub>n-1</sub> es distinto de Y<sub>n-1</sub>, en caso de generarse acarreo de salida, se descarta
- Finalmente, si X<sub>n-1</sub>=Y<sub>n-1</sub>=1, C<sub>n</sub> es necesariamente 1, este acarreo también se descarta pero si S<sub>n-1</sub>=0 (el signo del resultado es incorrecto), se produjo un error

→ Con operandos positivos:

$$\begin{array}{r} (\textcolor{red}{0}0010110)_2 = +(22)_{10} \\ +(\textcolor{blue}{0}0010010)_2 = +(18)_{10} \\ \hline (\textcolor{blue}{0}0101000)_2 = +(40)_{10} \end{array}$$

- Como era de esperar,  $C_n = 0$ . A su vez, considerando que  $S_{n-1} = 0$ , el resultado es correcto pues no se produjo overflow.

→ Con operandos de distinto signo:

$$\begin{array}{r} (\textcolor{red}{1}1010111)_2 = -(41)_{10} \\ +(\textcolor{blue}{0}1100010)_2 = +(98)_{10} \\ \hline (\textcolor{blue}{0}0111001)_2 = +(57)_{10} \end{array}$$

- En esta oportunidad,  $C_n = 1$ . Se descarta el acarreo de salida y como no se puede producir overflow con operandos de distinto signo el resultado obtenido es necesariamente correcto.

## Detección de overflow

Al operar en complemento a la base solo se pueden producir overflow al sumar números de igual signo. El mecanismo de detección de overflow es análogo al anterior: se inspecciona el bit de signo del resultado y si no es el esperado es porque se produjo overflow. Esta detección equivale a computar  $C_n \text{ xor } C_{n-1}$ . Recordemos que al igual que en las restantes representaciones, no es posible provocar un overflow al sumar números de distinto signo.

### → Con operandos negativos:

$$\begin{array}{r} (\textcolor{red}{1}10000000)_2 = -(128)_{10} \\ +(\textcolor{blue}{11111111})_2 = -(\textcolor{blue}{1})_{10} \\ \hline (\textcolor{blue}{01111111})_2 = \textcolor{red}{0V} \end{array}$$

- Como era de esperar,  $C_n = 1$ . No obstante, el signo del resultado es incorrecto, se produjo overflow.
- Nótese que  $C_n \oplus C_{n-1} = 1$ , pues  $C_n = 1$  y  $C_{n-1} = 0$ .

Para completar el análisis comparativo de las tres representaciones alternativas nos resta estudiar la eficiencia de sus implementaciones. Para signo magnitud debe considerarse que para implementar una suma se debe llevar adelante una comparación de magnitudes; esta comparación se implementa a nivel de hardware analizando el signo de la diferencia entre las magnitudes. Es decir, toda vez que se sume en signo magnitud, se debe en el peor caso, primero restar y luego sumar (por lo que hay que disponer de ambos circuitos).

Para DRC sucede algo análogo: en el peor caso (esto es, cuando se produce acarreo de salida), para sumar dos magnitudes se deben realizar también dos operaciones sucesivas de suma

RC se destaca por ser la representación más eficiente, para realizar una suma simplemente se lleva a adelante la suma propiamente dicha. Finalmente, RC y DRC permiten implementar la operación de resta usando el propio circuito sumador al combinarlo con la operación de complementación

## Cambio de precisión

Al cambiar la precisión de un cierto operando se debe tener en cuenta la representación numérica subyacente. En el caso de SM, aumentar la precisión de un operando consiste en agregar ceros a la izquierda de la magnitud representada, sin afectar al signo. En caso de RC y DRC, aumentar la precisión de un operando consiste en extender su signo, replicando su valor tantas veces como sea necesario

## Desplazamiento

La implementación de las operaciones de desplazamiento también debe tener en cuenta la representación numérica subyacente. Para desplazar a derecha o a izquierda un número codificado en SM, se desplazan libremente todos los bits salvo el de signo, el cual debe quedar fijo, para desplazar a derecha un número codificado en RD o DRC, se desplazan todos los bits recordando que

deben ir ingresando copias del bit de signo. En contraste, para desplazar a izq se desplazan todos los bit salvo el de signo, el cual queda fijo

### Suma en BCD exceso 3

En caso de operar usando la codificación exceso 3, se deben contemplar que sucede con el acarreo en cada uno de los dígitos decimales. Si al sumar codificación en binario de dos dígitos decimales se produce acarreo, los excesos se cancelan, por lo que deben ser restaurados. Caso contrario si al sumar la codificación en binario de dos dígitos decimales no se produce acarreo, los excesos se acumulan, por lo que se debe contrarrestar el exceso adicional

- Se desea calcular la suma de los siguientes operandos codificados en **BCD** exceso-3, usando el esquema signo-magnitud.

$$\begin{array}{r}
 +(148)_{10} = ( \textcolor{blue}{0011^0} \textcolor{blue}{0100^1} \textcolor{blue}{0111^0} \textcolor{blue}{1011} )_{\text{BCD-3}} \\
 +(71)_{10} = +(\textcolor{blue}{0011} \textcolor{blue}{0011} \textcolor{blue}{1010} \textcolor{blue}{0100})_{\text{BCD-3}} \\
 \hline
 (\textcolor{blue}{0011} \textcolor{blue}{1000} \textcolor{blue}{0001} \textcolor{blue}{1111}) \\
 \textcolor{green}{-0011} \textcolor{green}{+0011} \textcolor{green}{-0011} \\
 \hline
 +(219)_{10} = (\textcolor{blue}{0011} \textcolor{blue}{0101} \textcolor{blue}{0100} \textcolor{blue}{1100})_{\text{BCD-3}}
 \end{array}$$

en rojo se indica  
 el acarreo de salida  
 de cada dígito decimal

en verde se destaca  
 el ajuste de los excesos

- Se desea calcular la suma de los siguientes operandos codificados en **BCD** exceso-3, usando el esquema **DRC** (complemento a 9).

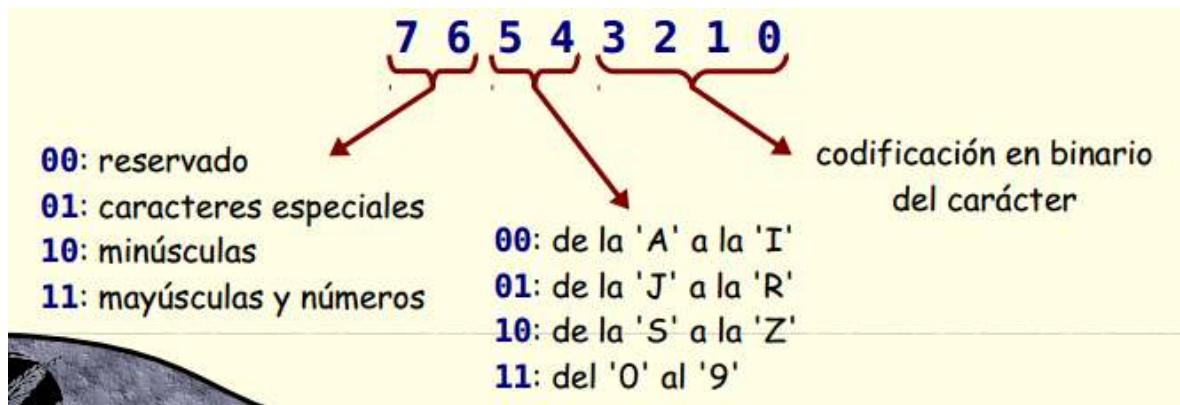
$$\begin{array}{r}
 -(148)_{10} = (\textcolor{red}{1} \textcolor{blue}{1100^1} \textcolor{blue}{1011^0} \textcolor{blue}{1000^0} \textcolor{blue}{0100})_{\text{BCD-3}} \\
 -(71)_{10} = +(\textcolor{blue}{1100} \textcolor{blue}{1100} \textcolor{blue}{0101} \textcolor{blue}{1011})_{\text{BCD-3}} \\
 \hline
 (\textcolor{blue}{1001} \textcolor{blue}{0111} \textcolor{blue}{1101} \textcolor{blue}{1111}) \\
 \textcolor{green}{+0011} \textcolor{green}{+0011} \textcolor{green}{-0011} \textcolor{green}{-0011} \\
 \hline
 (\textcolor{blue}{0} \textcolor{blue}{1100^0} \textcolor{blue}{1010^0} \textcolor{blue}{1010^1} \textcolor{blue}{1100})_{\text{BCD-3}} \\
 +(\textcolor{blue}{0011} \textcolor{blue}{0011} \textcolor{blue}{0011} \textcolor{blue}{0100})_{\text{BCD-3}} \\
 \hline
 (\textcolor{blue}{1111} \textcolor{blue}{1101} \textcolor{blue}{1110} \textcolor{blue}{0000}) \\
 \textcolor{green}{-0011} \textcolor{green}{-0011} \textcolor{green}{-0011} \textcolor{green}{+0011} \\
 \hline
 -(219)_{10} = (\textcolor{blue}{1100} \textcolor{blue}{1010} \textcolor{blue}{1011} \textcolor{blue}{0011})_{\text{BCD-3}}
 \end{array}$$

### Detección de overflow:

El operar usando la codificación bcd exceso 3 no afecta al mecanismo de detección de overflow del esquema de representación para números negativos que se halla elegido

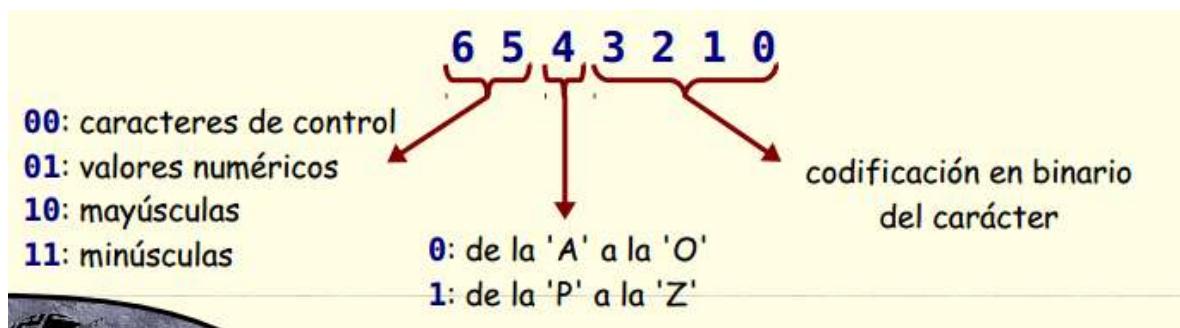
### Codificación EBCDIC

El código EBCDIC es una extencion del código bcd diseñado por ibm en década del 60. Codifica directamente caracteres de texto, usando 8 bits por cada carácter. Los bits se organizan de la siguiente manera



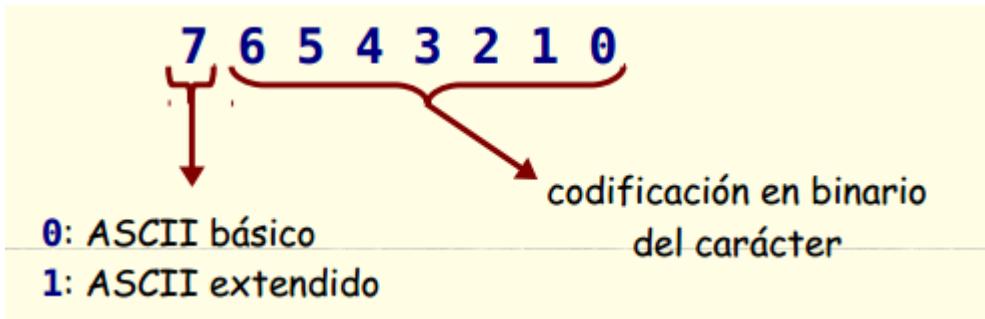
### Código ascii

El código ascii es un estándar también creado en los 60 espesificamente para los americanos. Codifica cada carácter usando 7 bits, lo que permite 128 combinaciones (razón por la cual no contempla, entre otros, vocales acentuadas ni la letra Ñ). los bits se organzan de la siguiente manera



### Código ascii extendido

El código ascii extendido es una extensión del código ascii original. Con el objeto de modificar algunos de los caracteres faltantes se incorpora un bit mas por carácter. Existen multiples extensiones, una para cada región o zona del planeta. Se organiza de la sig manera:



### Codificación UNICODE

El código Unicode fue propuesto para unificar el conjunto de códigos incompatibles existentes al momento de su concepción. Al tratarse de un estar reciente, el comité que lo diseñó tuvo la chance de corregir los inconvenientes identificados en los restantes códigos. Usa un esquema de codificación extensible, por lo que se puede seguir agregando nuevos caracteres

### UTF-8

La codificación UTF-8 es una de las maneras que existe para codificar el código Unicode. La codificación se compone de una cantidad variable de bloques de 8bits. Como objetivo de diseño se desea maximizar la compatibilidad con el código ascii. Por esta razón las primeras 128 codificaciones coinciden con el mapeo del código ascii. De ahí en adelante entra en acción el esquema extensible, agregando nuevos bloques de 8 bits a medida que vallan siendo requeridos

### UTF-16 y UTF-32

Las codificaciones UTF-16 y UTF-32 aparecen como alternativas a la codificación UTF-8. UTF-16 solo codifica una porción del código Unicode. Hace uso de una cantidad variables de bloques de 16 bits. UTF-32 codifica el mismo subconjunto que UTF-16, pero la principal diferencia es que hace uso de exactamente un único bloque de 32 bits

## Diapositiva 4 (parte 1)

### Concepto de error:

Toda vez que una pieza de información es transmitida existe la posibilidad de que lo enviado no coincida con lo recibido. Para contrarrestar el efecto de los errores en la transmisión existen 3 alternativas:

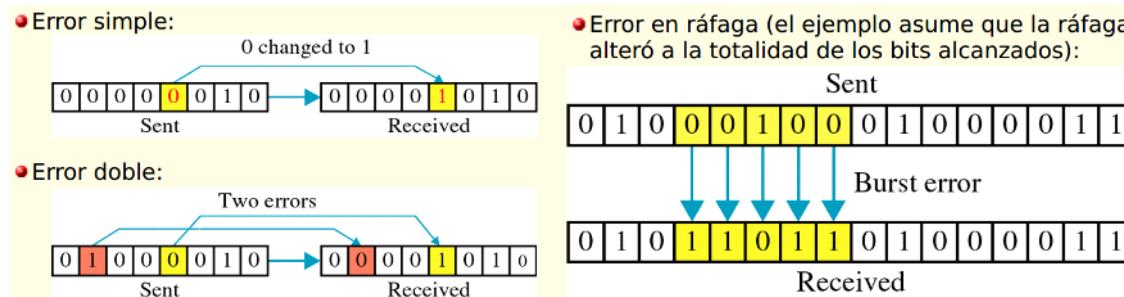
- aceptar que se produzcan: en ciertas circunstancias es posible que la información transmitida siga siendo relevante aun ante la presencia de un error
- Impedir que se produzcan: tomar todos los recaudos necesarios para asegurar que nunca se produzca un error. Esta alternativa suele tener un costo prohibitivo
- Contemplar que se produzcan: incorporar mecanismos que permitan atemperar el impacto de estos errores. En este caso la idea es incorporar un mecanismo que se encargue de

cancelar el impacto de los eventuales errores de transición. Estos mecanismos, cuyo propósito es lidiar con los errores, se clasifican en dos categorías: código detectores de error, los cuales incorporan información adicional junto con los datos transmitidos de manera que se pueda determinar si se produjo o no un error durante la transición; códigos correctores de error, los cuales incorporan más información que uno detector, ya que la idea es, además de detectar si se produjo un error tener la certeza de en qué lugar se produjo a fin de poder corregirlo sin requerir la retransmisión del dato en cuestión

### Tipos de Error

Al trabajar con información binaria, el error se trata simplemente de un intercambio (toggle) de valor de uno o más bits. Los errores se clasifican en:

- Error a nivel de bit: el error afecta a n bits del dato transmitido. En función de n, hablamos de error simple, doble, etc.
- Error en ráfaga: el error afecta a m bits consecutivos, estando el primero y el último en error (notece que los bits entre medio pueden o no estar en error)



### Código

Denominaremos código a un determinado conjunto de bits usualmente de longitud fija. Sean p y q dos patrones de bits, en este contexto denominaremos distancia entre p y q, notado  $d(p,q)$  a la cantidad de posiciones de bits en los cuales los patrones p y q difieren. Sea p un patrón de bits, denominaremos peso de la palabra p, notado como  $w(p)$ , al número de bits puestos a 1 dentro de esa palabra

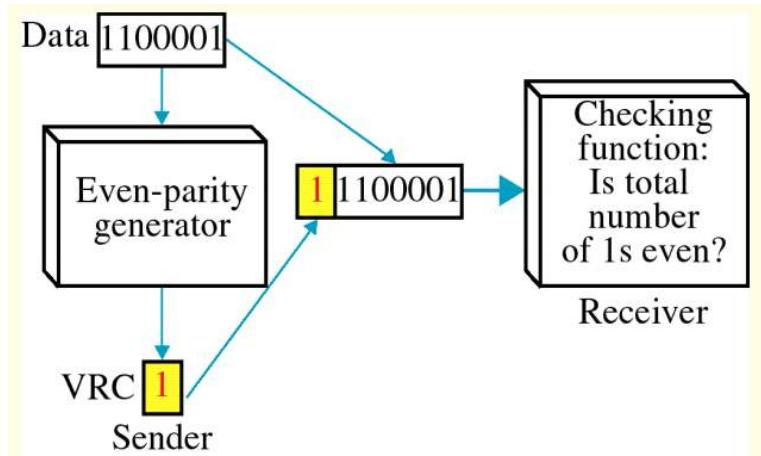
### Minima distancia

Sea C un código compuesto de patrones de bits de longitud fija. En este marco denominaremos mínima distancia del código C a la menor distancia que se observe entre dos patrones de bits no idénticos tomados de C. tener en cuenta que para determinar la mínima distancia de un cierto código es necesario calcular un gran número de instancias ya que se deben analizar la distancia entre cada patrón de bits con respecto a los restantes patrones de bits

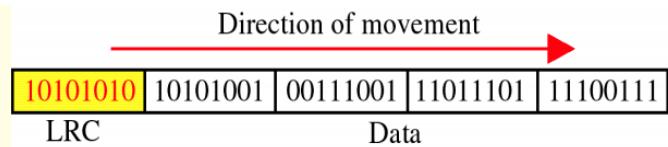
En general, podemos concluir que toda vez que se produzca una cantidad menor de errores que la mínima distancia del código adoptado, el error será siempre detectado. Pero, si se produce una cantidad igual o mayor de errores, a veces se detectará pero a veces no.

Para detectar errores se debe incorporar alguna forma de redundancia de dato transmitido, a fin de que se pueda determinar si la información se recibió correctamente. Se han ensayado distintas alternativas:

**VRC:** el código VRC (vertical redundancy check) añade un bit de paridad a cada unidad de datos de manera que la cantidad de bits en 1 sea par o impar (en función de la paridad adoptada). También se lo conoce como TRC (tranverse redundancy check). La idea es que se use un bit de paridad para cada unidad de datos, en vez de usar un único bit para la totalidad del mensaje. Al igual que paridad, detecta la totalidad de los errores simples a nivel de unidad de dato



**LRC:** el código LRC (longitudinal redundancy check) aplica la misma idea que el código VRC, pero computa la paridad en sentido longitudinal. La idea es que un bloque de bits se divida en filas para luego añadir una fila de bits de redundancia. La intención es permitir la detección de errores en ráfaga. El código LRC resulta mas complicado de analizar producto de la manera en la que se ordena los bits del mensaje a ser transmitido. En relación a los errores a nivle de bits, este código sigue detectando correctamente a lo sumo errores simples. El principal beneficio es que ahora estamos en condiciones de detectar errores en ráfaga.



**Código CRC:** el código CRC (cyclic redundancy check) se basa en ciertas propiedades matemáticas que satisface el cociente entre polinomios. La idea en pocas palabras es agragar al patrón de bit que compone el mensaje a ser enviado un conjunto de bits adicionales de manera tal que el patrón resultante, a ser considerado como un polinomio binario, resulte divisible de manera exacta por un cierto polinomio denominado polinomio generador.

Para poder llevar adelante el cosiente entre polinomios, se debe interpretar el mensaje original como si se tratara de un polinomio. La clave está en pensar los  $n$  bits que componen al patrón original como los coeficientes binarios de las primeras  $n-1$  potencias de  $X$ . notece que el polinomio resultante es de grado a lo sumo  $n-1$ . Por caso, el patrón 010011 denota el polinomio  $X^4 + X + 1$ .

Sea  $M(X)$  (mensaje) el polinomio binario representado en el mensaje original, sea  $G(X)$  el polinomio generador que se esté usando y sea  $r$  su grado. En este contexto, el mensaje a ser transmitido  $T(X)$  es  $X^r M(x) + r(X)$ , donde  $r(X)$  es el resto de dividir  $X^r M(x)$  por  $G(X)$ .  $T(X)$  resulta divisible de manera exacta por  $G(X)$  puesto que al tratarse de polinomios binarios se puede mostrar que  $X^r M(x) + r(X)$  equivale a  $X^r M(x) - r(X)$ .

Paso para calcular los bits que deben ser agregados a un cierto mensaje  $M(X)$ :

- Primero se añaden  $n$  bits en cero a la derecha de  $M(X)$  (esto es, se añaden tantos ceros como grado tenga el polinomio generador)
- Luego se divide al polinomio obtenido por el polinomio generador. Esta división se realiza en modulo 2, que es igual que la división binaria, con dos excepciones: no hay carries ni borrows
- Finalmente, para obtener  $T(X)$  se suma el resto  $r(X)$  al polinomio original  $M(X)$  (desplazado en  $r$  bits)

$$\begin{array}{r}
 x^r M(x) = \textcolor{blue}{110101101110000} \\
 \textcolor{blue}{\oplus 10011} \\
 \hline
 \textcolor{blue}{010011} \\
 \textcolor{blue}{\oplus 10011} \\
 \hline
 \textcolor{blue}{0000010111} \\
 \textcolor{blue}{\oplus 10011} \\
 \hline
 \textcolor{blue}{0010000} \\
 \textcolor{blue}{\oplus 10011} \\
 \hline
 \textcolor{blue}{0001100} = R(x)
 \end{array}$$

El algoritmo CRC cumple dos roles: por un lado permite determinar los bits que se deben agregar al mensaje original y a su vez, también permiten verificar si el mensaje recibido contiene o no errores. Notece que el receptor puede ir dividiendo el mensaje a medida que va recibiendo los bits, es decir, no hace falta recibir en su totalidad para recién ahí comenzar a verificar el CRC. Este código es muy simple de implementar en HW.

El polinomio generador a ser usado debe ser elegido con cuidado, ya que la capacidad de detección de errores dependerá de las características del mismo

## ● Existen varios polinomios actualmente en uso:

- **CRC-4-ITU:**  $x^4 + x + 1$
- **CRC-16-IBM:**  $x^{16} + x^{15} + x^2 + 1$
- **CRC-CCITT:**  $x^{16} + x^{12} + x^5 + 1$
- **CRC-32:**  $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Los polinomios generados mas interesantes dan lugar a códigos cíclicos. Un código se denomina cíclico cuando el mensaje  $T(X)$  puede ser corrido cíclicamente a derecha o izquierda un numero arbitrario de lugares sin perder la propiedad de ser divisible exactamente por  $G(X)$ . Sea  $k$  la longitud del mensaje original  $M(X)$  y sea  $n$  la del mensaje a transmitir  $T(X)$ . se puede demostrar que para obtener un código cíclico para el par  $(n,k)$  basta con tomar como polinomio generador de grado  $n-k$  a alguno de los factores del polinomio  $X^n + 1$

## ● Se desea obtener un polinomio generador para $k = 11$ y $n = 15$ .

- En primer lugar se debe factorizar  $x^{15} + 1$ :

$$\begin{aligned}x^{15} + 1 &= (x^4 + x + 1) \times (x^4 + x^3 + 1) \times \\&\quad (x^4 + x^3 + x^2 + x + 1) \times \\&\quad (x^2 + x + 1) \times (x + 1)\end{aligned}$$

- Luego, se puede tomar cualquiera de los factores de grado  $r = n - k = 4$ .
- Por caso, en el último ejemplo desarrollado usamos como polinomio generador  $(x^4 + x + 1)$ .

CRC se distingue de los métodos antes vistos de detección de error ya que a priori parece no depende de la incorporación de bits de paridad entre medio de los bits del mensaje. No obstante, sigue existiendo un vínculo entre CRC y paridad, agregar un bit de paridad equivale a aplicar el código CRC usando  $X+1$  como polinomio generador ( $G(X)=11$ )

### Analisis de detección para CRC:

sean  $T(X)$  el patrón de bits enviado y sea  $T(X)+EX$  el patrón recibido, en caso de que  $E(X)=0$  hablamos de una transmisión sin errores, caso contrario, cuando  $E(X)\neq0$ , la mayor y menor potencia de  $x$  marcarán el comienzo y el fin de la ráfaga en error. Como por construcción  $G(X)$  divide a  $T(X)$  la capacidad de detección del código CRC pivote en que  $G(X)$  no divida a  $E(X)$ , por caso, para un error en ráfaga de  $p$  bits comenzando a partir de la posición  $q$ , se verifica que  $E(X)=$

$X^q(X^{p-1} + \dots + 1)$  (notece que en el termino de la derecha pueden faltar factores, salvo el primer y el ultimo)

Analicemos por casos lo que sucede para distintos valores de p:

Error en ráfaga de longitud  $p \leq r$ : en este caso no hay forma de que  $G(X)$  divida exactamente a ninguno de los dos términos que lo componen, es decir, las ráfagas en error de una longitud menor o igual al grado del polinomio generador son siempre detectadas.

Error en ráfaga de longitud  $p = r+1$ : en este caso, como estamos calculando el cociente entre polinomios de igual grado,  $G(X)$  dividirá a  $E(X)$  únicamente cuando  $G(X)$  coincide con el término de la derecha de  $E(X)$ . En este escenario ¿Cuál es la probabilidad de no detectar el error? Pues bien, para que coincidan los  $r+1$  en error, basta con que coincidan los  $r-1$  bits internos (ya que en el polinomio generador, al igual que en  $E(X)$ , el primer y último bits deben ser 1), es decir, CRC falla en solo 1 de las  $2^{r-1}$  posibilidades

Error en ráfaga de longitud  $p > r+1$ : en este caso la situación es análoga, debemos analizar bajo qué condiciones  $G(X)$  divide exactamente al término de la derecha de  $E(X)$ . Este análisis se reduce a considerar que sucede con el último paso de la división (para que el resto sea 0 en el último resto parcial debe coincidir con  $G(X)$ ). Esto equivale a que coincidan solo los primeros  $r$  bits de  $G(X)$  con ese último resto, ya que ese bit más significativo de  $G(X)$  es necesariamente 1

En síntesis, la capacidad de detección de ráfaga en error de un código CRC que haga uso de un polinomio generador de grado  $r$  es:

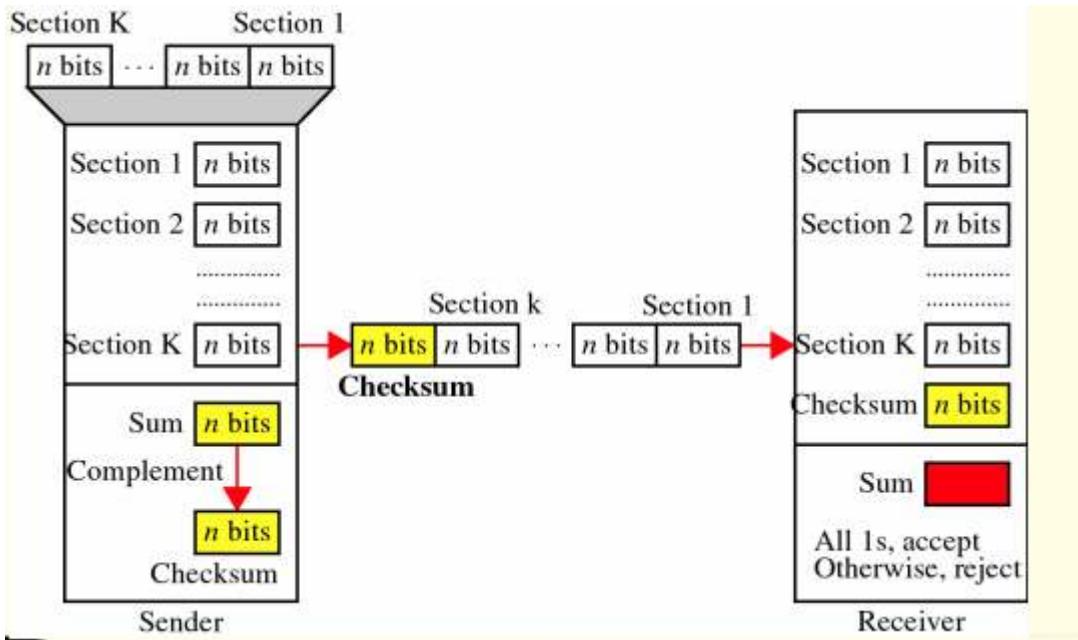
- Detectar con un probabilidad del 100% a las ráfagas en error de longitud  $k \leq r$ .
- Detectar con una probabilidad de  $1 - 2^{-(r-1)}$  a las ráfagas en error de longitud  $k = r+1$ .
- Detectar con una probabilidad de  $1 - 2^{-r}$  a las ráfagas en error de cualquier otra longitud

## Funcion HASH

Si bien la capacidad de detección del código CRC es directamente proporcional al grado del polinomio generador que se está usando, no se ha popularizado el uso de polinomios de grado mayor a 32 (el uso de polinomios generadores de grado tan alto degrada el desempeño). De hacer falta un mayor nivel de detección se puede hacer uso de funciones hash tales como MD5 o SHA-1

## Checksum:

El checksum (suma de comprobación) es un mecanismo relativamente sencillo para la verificar la integridad de un mensaje. La idea central consiste en ir sumando segmentos de datos de  $n$  bits en complemento a 1 de forma que la longitud de la suma sea también  $n$  bits, para luego complementar ese total obtenido antes de anexarlo al bloque de datos. ¿Cuánto dará la suma de comprobación del bloque original al agregarle el complemento del resultado anterior?



Analisis: asumiendo un tamaño de segmento de datos de 8 bits, el checksum calculado no se altera en ninguno de los siguientes escenarios: al redondear sin cambiar los bytes del mensaje, al insertar o eliminar bytes con sus bits todos en 0 o todos en 1 y al presentarse errores múltiples o en ráfaga pero que justo se cancelan unos a otros. En síntesis, checksum no constituye un mecanismo propio de detección de error.

### Fletcher's Checksum

John G. Fletcher propuso en la década del '70 una modificación para subsanar algunos de los cuestionamientos anteriores. La idea central es llevar dos sumas de comprobación, la primera como es usual acumulando los segmentos de datos pero la segunda acumulando los valores parciales de la primera suma de comprobación. Al hacerlo, ahora es posible detectar las alteraciones en el orden de los segmentos

## Diapositiva 4 (parte 2)

### Corrección de errores

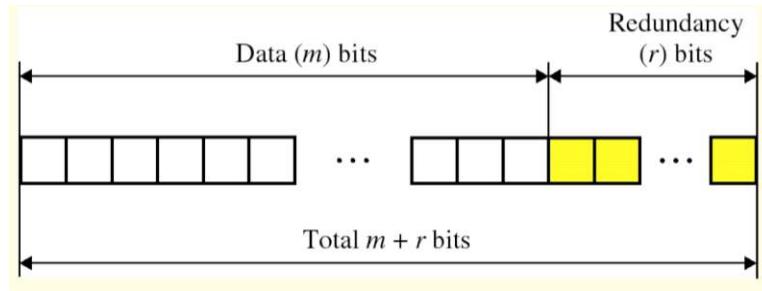
Usualmente cuando se detecta un error se suele solicitar la retransmisión del mensaje recibido incorrectamente. En los casos donde resulta costosa o incluso imposible la retransmisión del mensaje original tiene sentido intentar establecer dónde se produjo el error, a fin de corregirlo.

Ante los errores simples es posible reconocer con certeza en qué bit se produjo el error. No obstante, el adoptar esta política de corrección de errores afecta la capacidad de detección.

Por caso, si al recibir el patrón 010100 interpretaremos que se produjo un error simple ya que el patrón original era 010101, no detectaremos el error doble cuando el patrón original fuera

000000. Es decir, en el código del ejemplo si decidimos corregir errores simples, debemos dejar de detectar los errores dobles.

Supongamos que se desea diseñar un código para corregir errores simples con  $n=m+r$ .



En este escenario, para cada patrón válido debemos reservar cada uno de los  $m+r$  patrones que se obtienen al adulterar un bit. Es decir, para cada uno de los  $2^m$  mensajes debemos disponer de  $m+r+1$  patrones, pero todos estos patrones tienen que codificarse con  $m+r$  bits:  $2^m \cdot (m+r+1) \leq 2^{m+r} \rightarrow m+r+1 \leq 2^r$ . En otras palabras, una vez elegido un cierto  $m$ , es posible derivar matemáticamente cuál será el límite inferior para  $r$ .

Extendiendo este análisis para el caso de errores dobles se observa lo siguiente: Para cada patrón válido se debe reservar por un lado  $m+r$  patrones para corregir errores simples pero por otro lado hacen falta  $[(m+r) \cdot (m+r-1)]/2$  patrones para corregir los errores dobles. En síntesis, ahora tenemos que:

$$m + r + [(m+r) \cdot (m+r-1)]/2 + 1 \leq 2^r$$

### Mínima Distancia

El concepto de mínima distancia de un código desempeña un rol central en la determinación de la capacidad de detección y de corrección. Para una cierta distancia mínima  $M$  es posible detectar hasta  $d=M-1$  bits en error. No obstante, al incorporar la capacidad de corrección, por cada error corregido debemos disminuir de manera acorde la capacidad de detección. En otras palabras,  $d=M-1-c$ , lo que equivale a afirmar que  $M-1=c+d$ .

Por otra parte, como por cada bit corregido debemos asegurarnos que ningún otro patrón válido esté a la misma distancia aparte del elegido, también se debe verificar que  $c \leq (M-1)/2$ .

Reemplazando  $M-1$  por el valor antes obtenido se deriva que  $2c \leq c+d$ , es decir que  $c \leq d$ .

- Para  $M=2$  (por caso, paridad), se observa que la única solución posible es  $c=0$  y  $d=1$ .
- Para  $M=3$  (por caso, Hamming), existen dos soluciones posibles:  $c=1$  y  $d=1$  o bien  $c=0$  y  $d=2$ .
- Para  $M=4$  (por caso, Hamming extendido), también existen dos soluciones:  $c=1$  y  $d=2$  o bien  $c=0$  y  $d=3$ . Nótese que ni  $c=2$  y  $d=1$  ni tampoco  $c=3$  y  $d=0$  satisfacen que  $c \leq d$

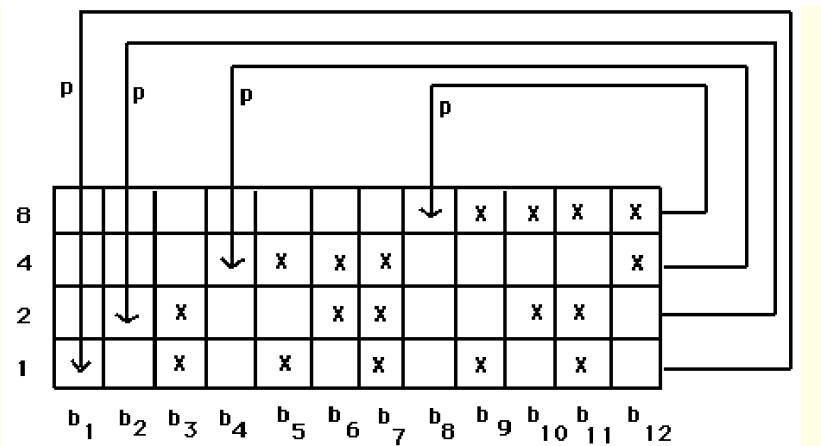
### Código corrector Naive

Una manera no muy eficiente de introducir redundancia consiste en replicar los datos. Por ejemplo, una posibilidad es repetir  $k$  veces cada bit del mensaje original.

## Código Hamming

Se basa en conceptos conocidos. El mensaje se divide en dos partes, los datos a ser transmitidos y la redundancia que se le agregará. La redundancia agregada al dato se compone esencialmente de bits de paridad.

Se trata de un código mínima distancia 3, el cual intercala bits de código con bits de datos. Para esto se reservan las posiciones potencias de 2 para alojar bits de código, usando las restantes para los bits de dato. Por caso, para  $m=8$  se deben incorporar tantos bits de código como sea necesario para satisfacer la inecuación  $8+r+1 \leq 2^r$ . Recién  $r=4$  satisface esta restricción, por lo que los bits de código ocuparan las posiciones 1, 2, 4 y 8.



## Calculo del síndrome (hamming)

Esta matriz de posiciones binarias se puede usar en conjunción al patrón de bits recibidos para determinar si se produjeron o no errores.

$$\begin{array}{l}
 \text{matriz de posiciones} \\
 \left[ \begin{array}{ccccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{array} \right] \times = \left[ \begin{array}{c} c_1 \\ c_2 \\ d_4 \\ c_3 \\ d_3 \\ d_2 \\ d_1 \end{array} \right] = \left[ \begin{array}{c} S_1 \\ S_2 \\ S_3 \end{array} \right] \\
 \text{síndrome calculado}
 \end{array}$$

El síndrome obtenido permite establecer si se produjo algún error en la transmisión: Si se obtiene un síndrome nulo (esto es,  $[0\ 0\ 0]^t$ ), no se produjeron errores. Caso contrario, el síndrome marca la posición donde aparentemente se produjo el error. Este cálculo equivale a calcular paridad par sobre los bits del mensaje, puesto que:

$$S_1 = c_1 \text{ xor } d_4 \text{ xor } d_3 \text{ xor } d_1$$

$$S_2 = c_2 \text{ xor } d_4 \text{ xor } d_2 \text{ xor } d_1$$

$$S_3 = c_3 \text{ xor } d_3 \text{ xor } d_2 \text{ xor } d_1$$

La capacidad de detección y corrección de errores del código Hamming no depende del esquema de paridad elegido (paridad par o impar).

### Ordenamiento de los bits (hamming)

Existen dos formas de numerar las posiciones en un cierto patrón de bits:

- De derecha a izquierda, con la primera posición en el extremo derecho y la última en el extremo izquierdo.
- De izquierda a derecha, con la primera posición en el extremo izquierdo y la última en el extremo derecho.

El funcionamiento del código hamming no se verá afectado por el ordenamiento de los bits, siempre y cuando los bits de código se sigan computando correctamente.

### Cálculo de los bits de código (hamming)

Supongamos que se desea transmitir el patrón de bits 01010101, usando Hamming, paridad par, ordenando los bits de izquierda a derecha.

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
?	?	0	?	1	0	1	?	0	1	0	1
$c_1$	$c_2$	$d_8$	$c_3$	$d_7$	$d_6$	$d_5$	$c_4$	$d_4$	$d_3$	$d_2$	$d_1$
$c_1 = p(d_8, d_7, d_5, d_4, d_2) = p(0, 1, 1, 0, 0) = 0$											
$c_2 = p(d_8, d_6, d_5, d_3, d_2) = p(0, 0, 1, 1, 0) = 0$											
$c_3 = p(d_7, d_6, d_5, d_1) = p(1, 0, 1, 1) = 1$											
$c_4 = p(d_4, d_3, d_2, d_1) = p(0, 1, 0, 1) = 0$											

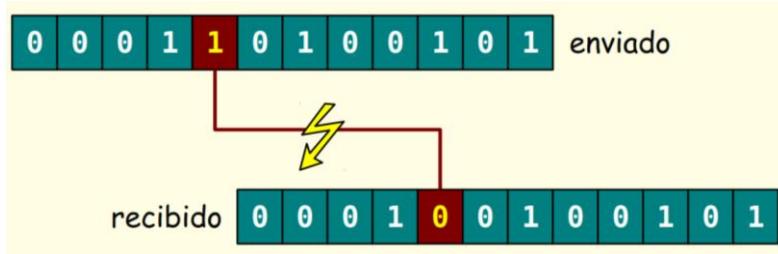
### Política (hamming)

Recordemos que Hamming por tratarse de un código con  $M=3$ , admite dos soluciones a las inecuaciones que vinculan  $M$ ,  $d$  y  $c$ . Una posibilidad es detectar y corregir errores simples, y la otra es sólo detectar errores simples y dobles.

En caso de hacer uso de la política de detección y corrección de errores simples, el síndrome necesariamente apuntará a la posición en la cual se produjo el error. No obstante, en caso de hacer uso de la política de sólo detección de errores simples y dobles, el síndrome no va a representar una posición en concreto, por lo que simplemente se analiza si es nulo.

### Detección y corrección simple

Asumida una política de detección y corrección de errores simples, supongamos que se produjo el siguiente error simple sobre el dato antes calculado:



0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
0	0	0	1	0	0	1	0	0	1	0	1
$c_1$	$c_2$	$d_8$	$c_3$	$d_7$	$d_6$	$d_5$	$c_4$	$d_4$	$d_3$	$d_2$	$d_1$
$c'_1 = p(d_8, d_7, d_5, d_4, d_2) = p(0, 0, 1, 0, 0) = 1$											
$c'_2 = p(d_8, d_6, d_5, d_3, d_2) = p(0, 0, 1, 1, 0) = 0$											
$c'_3 = p(d_7, d_6, d_5, d_1) = p(0, 0, 1, 1) = 0$											
$c'_4 = p(d_4, d_3, d_2, d_1) = p(0, 1, 0, 1) = 0$											

Al haber adoptado la política de detección y corrección de errores simples, el síndrome apuntará al bit en error, en caso de existir. En primer lugar se recalculan los bits de código:

$$c'_4 = 0 \quad c'_3 = 0 \quad c'_2 = 0 \quad c'_1 = 1$$

Luego cotejamos estos valores con los recibidos, a fin de determinar el síndrome:

$$S_4 = 0 \text{ xor } 0 \quad S_3 = 1 \text{ xor } 0 \quad S_2 = 0 \text{ xor } 0 \quad S_1 = 0 \text{ xor } 1$$

Como el síndrome apunta a la posición 5 (0101) ese bit fue el afectado por el error, se lo corrige.

### Detección de errores simples y dobles

00	0	1	1	0	1	0	0	1	0	1	enviado
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
0	0	1	1	1	0	1	0	1	1	0	1
$c_1$	$c_2$	$d_8$	$c_3$	$d_7$	$d_6$	$d_5$	$c_4$	$d_4$	$d_3$	$d_2$	$d_1$
$c'_1 = p(d_8, d_7, d_5, d_4, d_2) = p(1, 1, 1, 1, 0) = 0$											
$c'_2 = p(d_8, d_6, d_5, d_3, d_2) = p(1, 0, 1, 1, 0) = 1$											
$c'_3 = p(d_7, d_6, d_5, d_1) = p(1, 0, 1, 1) = 1$											
$c'_4 = p(d_4, d_3, d_2, d_1) = p(1, 1, 0, 1) = 1$											

Bits de código recalculados:

$$c'_4 = 1 \quad c'_3 = 1 \quad c'_2 = 1 \quad c'_1 = 0$$

Calculo del síndrome:

$$S_4 = 0 \text{ xor } 1 \quad S_4 = 1 \text{ xor } 1 \quad S_4 = 0 \text{ xor } 1 \quad S_4 = 0 \text{ xor } 0$$

El síndrome apunta a la posición 10 (1010) que no está en error. Como no es nulo, se detecta el error, pero se desconoce si fue simple o doble.

### Detección de ráfagas de error

Si ocurre una ráfaga de longitud  $k$  en el bloque de  $k^*n$  (y ningún otro error), se verá afectado a lo sumo un bit de cada patrón. Por ende, el código Hamming será capaz de reconstruir cada patrón correctamente. Es decir, esta codificación está en condiciones de reconstruir el bloque por completo. En síntesis, utilizar  $K_r$  bits de control permite hacer que  $km$  bloques de bits de datos resulten inmunes a ráfagas hasta longitud  $k$ .

### Hamming mínima distancia 4 (o extendido)

La mínima distancia del código Hamming puede ser incrementada de 3 a 4 incorporando un bit de paridad que cubra la totalidad del mensaje.

Al incrementar la mínima distancia a 4, aparecen nuevas soluciones a las inecuaciones que relacionan a  $M$ ,  $c$  y  $d$ : ahora es posible hacer uso de la políticas  $c=0$  y  $d=3$  ó  $c=1$  y  $d=2$ . Notar que Hamming extendido ahora permite distinguir los errores simples de los errores dobles. Esto posibilita corregir al estar en presencia de un error simple y no hacerlo ante un error doble.

### Detección y corrección – $c=1$ y $d=2$ (hamming ext)

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	----
1	0	1	0	1	1	1	0	0	1	0	1	0
$c_1$	$c_2$	$d_8$	$c_3$	$d_7$	$d_6$	$d_5$	$c_4$	$d_4$	$d_3$	$d_2$	$d_1$	P
$c'_1 = p(d_8, d_7, d_5, d_4, d_2) = p(1, 1, 0, 1, 1) = 0$												
$c'_2 = p(d_8, d_6, d_5, d_3, d_2) = p(1, 1, 0, 0, 1) = 1$												
$c'_3 = p(d_7, d_6, d_5, d_1) = p(1, 1, 0, 0) = 0$												
$c'_4 = p(d_4, d_3, d_2, d_1) = p(1, 0, 1, 0) = 0$												

Paridad y bits de código recalculados:

$$c'_4 = 0 \quad c'_3 = 0 \quad c'_2 = 1 \quad c'_1 = 0 \quad P' = 0$$

Calculo del síndrome:

$$S_4 = 0 \text{ xor } 0 \quad S_4 = 0 \text{ xor } 0 \quad S_4 = 0 \text{ xor } 1 \quad S_4 = 1 \text{ xor } 0$$

Como  $P \text{ xor } P' = 1$ , se detecta una cantidad impar de errores, por lo que se corrige el bit señalado por el síndrome (esto es, el bit en la posición 0011).

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	....
<b>c<sub>1</sub></b>	<b>c<sub>2</sub></b>	<b>d<sub>8</sub></b>	<b>c<sub>3</sub></b>	<b>d<sub>7</sub></b>	<b>d<sub>6</sub></b>	<b>d<sub>5</sub></b>	<b>c<sub>4</sub></b>	<b>d<sub>4</sub></b>	<b>d<sub>3</sub></b>	<b>d<sub>2</sub></b>	<b>d<sub>1</sub></b>	<b>P</b>

$$\begin{aligned}
 c'_1 &= p(d_8, d_7, d_5, d_4, d_2) = p(1, 0, 0, 1, 1) = 1 \\
 c'_2 &= p(d_8, d_6, d_5, d_3, d_2) = p(1, 1, 0, 0, 1) = 1 \\
 c'_3 &= p(d_7, d_6, d_5, d_1) = p(0, 1, 0, 0) = 1 \\
 c'_4 &= p(d_4, d_3, d_2, d_1) = p(1, 0, 1, 0) = 0
 \end{aligned}$$

Paridad y bits de código recalculados:

$$c'_4 = 0 \quad c'_3 = 1 \quad c'_2 = 1 \quad c'_1 = 1 \quad P' = 1$$

Calculo del síndrome:

$$S_4 = 0 \text{ xor } 0 \quad S_4 = 0 \text{ xor } 1 \quad S_4 = 0 \text{ xor } 1 \quad S_4 = 1 \text{ xor } 1$$

Como  $P \text{ xor } P' = 0$ , se detecta una cantidad par de errores, por lo que el síndrome al ser no nulo permite detectar que se produjo un error doble.

Sólo detección –  $c=0$  y  $d=3$  (hamming ext)

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	....
<b>c<sub>1</sub></b>	<b>c<sub>2</sub></b>	<b>d<sub>8</sub></b>	<b>c<sub>3</sub></b>	<b>d<sub>7</sub></b>	<b>d<sub>6</sub></b>	<b>d<sub>5</sub></b>	<b>c<sub>4</sub></b>	<b>d<sub>4</sub></b>	<b>d<sub>3</sub></b>	<b>d<sub>2</sub></b>	<b>d<sub>1</sub></b>	<b>P</b>

$$\begin{aligned}
 c'_1 &= p(d_8, d_7, d_5, d_4, d_2) = p(1, 0, 0, 1, 1) = 1 \\
 c'_2 &= p(d_8, d_6, d_5, d_3, d_2) = p(1, 0, 0, 0, 1) = 0 \\
 c'_3 &= p(d_7, d_6, d_5, d_1) = p(0, 0, 0, 0) = 0 \\
 c'_4 &= p(d_4, d_3, d_2, d_1) = p(1, 0, 1, 0) = 0
 \end{aligned}$$

Paridad y bits de código recalculados:

$$c'_4 = 0 \quad c'_3 = 0 \quad c'_2 = 0 \quad c'_1 = 0 \quad P' = 0$$

Calculo del síndrome:

$$S_4 = 0 \text{ xor } 0 \quad S_4 = 0 \text{ xor } 0 \quad S_4 = 0 \text{ xor } 0 \quad S_4 = 1 \text{ xor } 0$$

Como  $P \text{ xor } P' = 1$ , se detecta una cantidad de errores; a su vez, como el síndrome es no nulo, se desconoce si se produjo un error simple o triple.

## Diapositiva 5

### Organización en capas

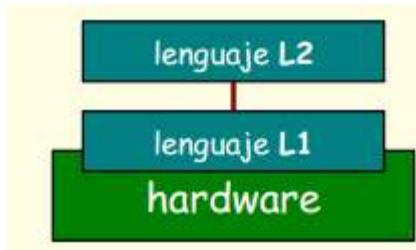
Una computadora es una maquina que puede resolver problemas ejecutando las instrucciones que le van siendo indicadas. La secuencia de instrucciones que describen como hacer una cierta tarea se denomina programa. El conjunto de instrucciones que puede ejecutar en cierto procesador

constituyen su lenguaje maquina. El lenguaje coloquial por las personas difiere considerablemente del lenguaje maquina.

Hacer uso de una computadora implica que las personas logren hablar con las maquinas. ¿Cómo podemos reconciliar estos lenguajes tan diferentes? Una posibilidad es hacer uso de una jerarquía de niveles en los que cada uno de estos haga uso de un lenguaje con mayor nivel de abstracción. La idea es que a medida que ascendamos en la jerarquía el lenguaje se vaya elejando de la maquina y al mismo tiempo se vaya acercando a las personas.

### Vision multinivel

El contar con multiples niveles permite crear nuevas maquinas sobre un mismo hardware:



L1 estara compuesto por aquellas primitivas de uso conveniente. Esto permite implementaciones flexibles, ya que independiza a la maquina del hardware subyacente.

L2 estará mas próximo a los algoritmos que se quieren implementar. Esto facilita la programación y también posibilita la resolución de problemas mas complejos

### Vinculacion entre los niveles

Existen esencialmente dos mecanismos que nos permiten vincular a los niveles entre si:

- Traducción: la traducción consiste en tomar un programa escrito en un cierto lenguaje para transformarlo en una secuencia de instrucciones equivalente de un lenguaje de nivel inferior
- Interpretación: la interpretación consiste en tomar instrucciones de un cierto lenguaje para ir transformadolas en tiempo de ejecución en un conjunto de instrucciones equivalentes del lenguaje a nivel inferior

### Mecanismo de traducción

El programa que se utiliza para llevar adelante la traducción se denomina compilador y está compuesto de instrucciones del nivel inferior. El programa original de nivel superior solo se utiliza durante la traducción. Luego, la computadora se desentiende de este programa ya que en tiempo de ejecución solo accedera al nuevo programa de nivel inferior

### Mecanismo de interpretación

El programa que se utiliza para llevar adelante la traducción se denomina interprete y también está compuesto de instrucciones de nivel inferior. En contraste, este mecanismo no genera un nuevo programa en el nivel inferior, ya que la conversión se hace dinámicamente en tiempo de

ejecución. Por esta razón, se debe conservar el programa original de nivel superior hasta el tiempo de ejecución, puesto que recién ahí se lleva adelante la conversión

### Concepto de maquina virtual

Mas que pensar en términos de compilación o de interpretación para vincular las distintas capas entre si, es convenientes abstenerse completamente de las capas inferiores. La idea es imaginar que existe una maquina virtual que directamente ejecuta las instrucciones de en nivel superior. El lenguaje de ese nivel superior se convierte en un nuevo lenguaje maquina, en esta ocasión de una maquina virtual.

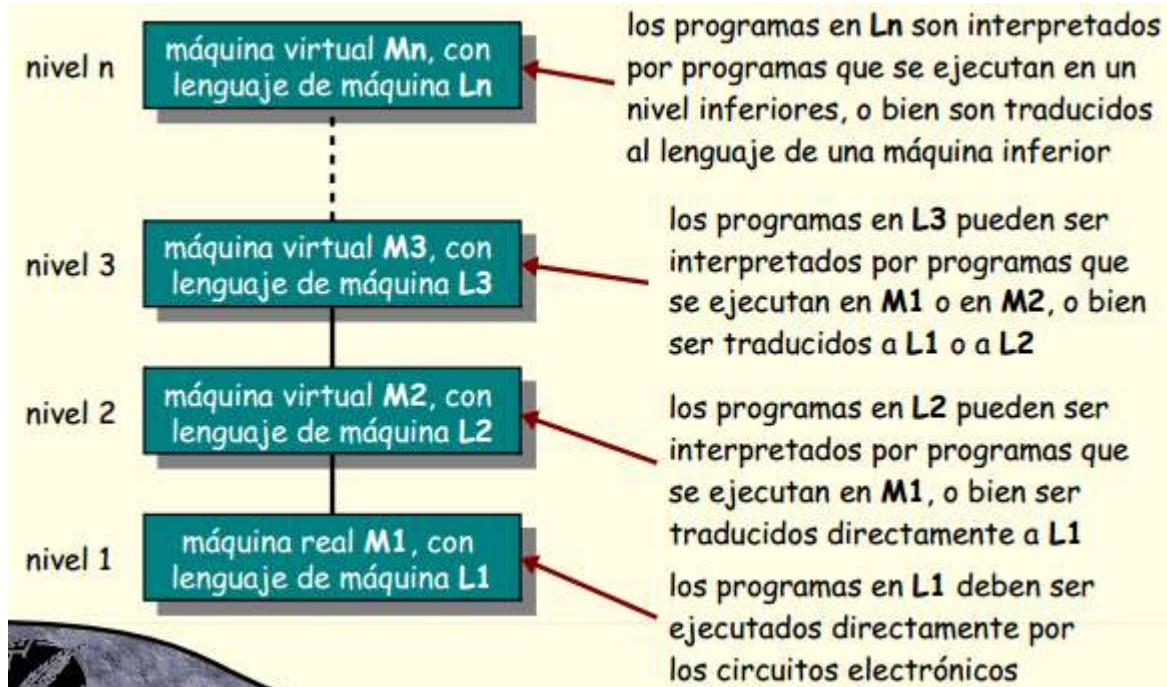
### Relacion entre niveles

Con el objeto de que la traducción y/o la interpretación resulten viables y eficientes es esencial que los niveles adyacentes no difieran en gran medida. Recordemos nuestro objetivo inicial, cada nuevo nivel tiene por objeto acercarnos mas a los usuarios. En principio podemos ir incorporando nuevas capas, cada una con su respectivo lenguaje maquina, hasta que alcancemos un nivel cuyo lenguaje resulte satisfactorio, es decir, lo suficientemente próximo a los usuarios del sistema.

### Lenguaje vs maquina

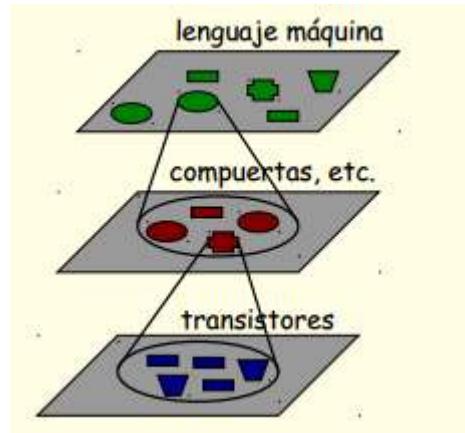
En cada nivel de esta organización es capas se identifican dos componentes principales: la maquina (virtual o no), caracterizado por el conjunto de instrucciones que puede ejecutar. El lenguaje, caracterizado por las facilidades a disposición del programador en ese nivel. Obsérvese que una maquina determina a su lenguaje de la misma manera que un lenguaje determina a su máquina.

### Organización multinivel



Recordemos que esta organización multinivel permite a su vez atacar el complicado problema de relacionar humanos con computadoras de una manera paulatina.

- La idea es que cada capa se construya con las capacidades provistas por la capa inferior
- A su vez, cada capa proveerá de mayores capacidades a las capas superiores.



### Maquinas contemporáneas

La organización multinivel de una maquina contemporánea se compone de ods o mas niveles. Mas aun, no ha de resultar llamativo que abarque hasta seis o incluso mas niveles. Nuestro repaso de los distintos niveles que usualmente componen estas maquinas comienza a nivel del hardware. Un electrónico estaría en condiciones de concebir incluso varias capas adicionales por debajo

Nivel 0 (la maquina real, no virtual): Este es el nivel de corte de nuestro análisis, por debajo existen otros niveles cuyo estudio en detalle corresponde a otras ramas de la ciencia. Esta compuesto del conjunto de compuertas lógicas que implementa cada uno de las unidades funcionales que conforman la computadora. Cabe destacar que se construye usando componentes analógicos los cuales, sin embargo, terminan evidenciando un comportamiento digital

Nivel 1 (microprogramaion): es el verdadero lenguaje maquina, pues es el nivel anterior no existe el concepto de programa como una secuencia de instrucciones a ser ejecutadas. En este nivel corre un programa denominado microprograma el cual es capaz de interpretar las instrucciones del nivel 2. Puede tener mas de un interprete corriendo en este nivel, y naturalmente cada uno definirá un nuevo lenguaje de nivel 2 junto con su correspondiente maquina virtual. Recordemos que la microprogramación aparece como alternativa al hacer uso de la ley de equivalencia entre HW y SW, es decir, en este nivel es posible optar por ejecutar directamente en hardware las instrucciones mas simples y al mismo tiempo resolver via software las mas complejas. La idea de los diseñadores del sistema era alcanzar un adecuado balance entre costo y desempeño.

Recordemos a su vez que la técnica de microprogramación floreció en la época en la que la memoria principal era muy lenta. En ese contexto contar con un set de instrucciones complejo permitia generar programas mas compactos y con menor requerimiento de ancho de banda. Este tipo de arquitectura se denomina justamente Complex Instruction Set Computer (CISC). La familia Intel x86 es posiblemente el ejemplo mas conocido de arquitectura CISC. Finalmente la utilidad de la microprogramación empezó a ceder de la mano de dos avances: por un lado el crecimiento exponencial de la cantidad de transistores disponibles permitio implementar en hardware mas y mas funcionalidad del nivel 2 y naturalmente el mayor impacto vino de la mano de la mejora en el desempeño de la memoria principal. Por esto, en la actualidad se opta por simplificar el set de instrucciones, dando a lugar a lo que hoy en dia se conoce como arquitectura RISC

Nivel 2 (maquina convencional): Este es el primer nivel al alcance del programador, mas alla de que el verdadero leguaje maquina sea el provisto por el nivel inmediato inferior. Es posible que un

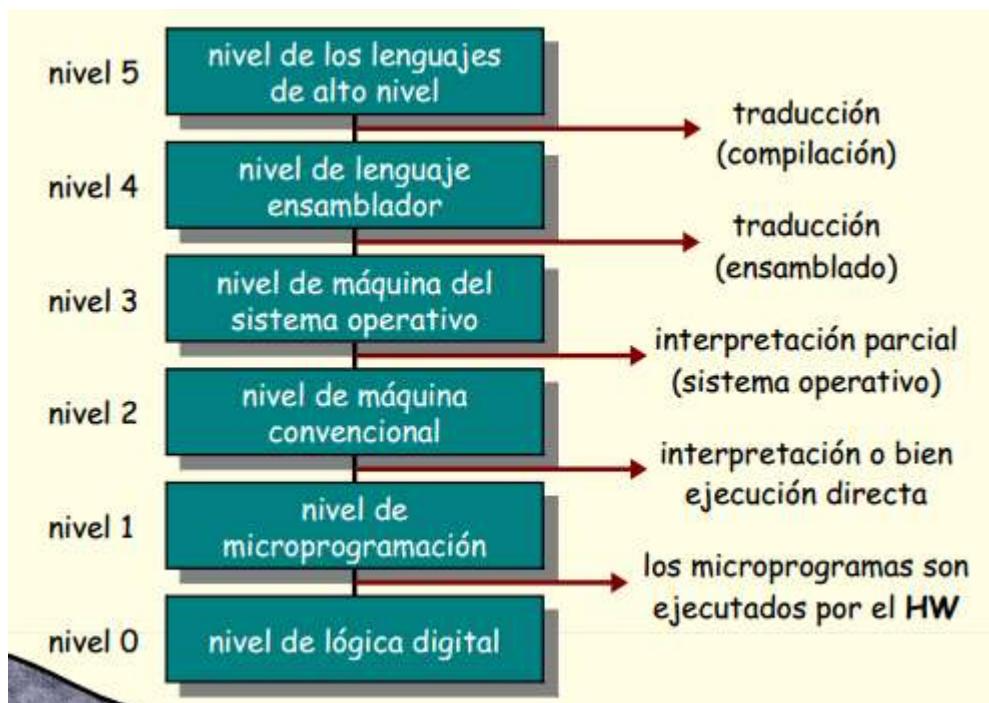
el mismo HW soporta multiples maquinas convencionales (a través de multiples interpretes corriendo en el nivel anterior). En las computadoras de tipo Risc se ejecutan las instrucciones de este nivel directamente en HW, es decir, no cuentan con la capa de microprogramación.

Nivel 3 (sistema operativo): se trata de un nivel híbrido, gran cantidad de las instrucciones de este nivel son las mismas que las del nivel inmediato inferior. Existen muchas mas variantes de maquinas en este nivel que en los niveles anteriores. Las nuevas facilidades provistas por este nivel fueron incorporadas por el interprete llamado sistema operativo por razones historicas. En este nivel aparece la posibilidad de correr dos mas programas en simultaneo

Existe una separación fundamental entre los niveles 3 y 4. A diferencia de los niveles superiores, los niveles 1, 2 y 3 no fueron diseñados para ser usados directamente por el programador de aplicaciones. Los primeros tres niveles tienen por objeto sustentar a los interpretes necesarios para que las maquinas virtuales de nivel superior funcionen correctamente. Los encargados de implementar estos programas se denominan programadores de sistema.

Nivel 4 (lenguaje ensamblador): este es el primer nivel en donde se hace uso de los programas traductores. Por otra parte, es el primer nivel que encuentra con un lenguaje simbolico (los niveles inferiores solo cuentan con un lenguaje numérico, compuesto de 1's y 0's). Este cambio hace que el nivel 4 sea el primer nivel inteligible por personas. El programa encargado de traducir los programas de este nivel se denomina ensamblador.

Nivel 5 (lenguaje de alto nivel): este nivel esta compuesto por los distintos lenguajes de programación de alto nivel, tales como java, pascal y C entre otros. Los programas escritos en estos lenguajes suelen ser traducidos a instrucciones de los niveles 3 o 4 mediante compiladores, si bien tambien existen algunos lenguajes de programación que son directamente interpretados.



## Evolucion Historica

Las primeras computadoras ala por la década del '40 solo contaban con dos niveles: el nivel de la maquina convencional donde se realizaba la programación y el nivel de la lógica digital donde eran ejecutados los programas. No había manera alguna de que la computación en estos términos se tornara popular. Los programadores eran en ocasiones los propios constructores de la computadora

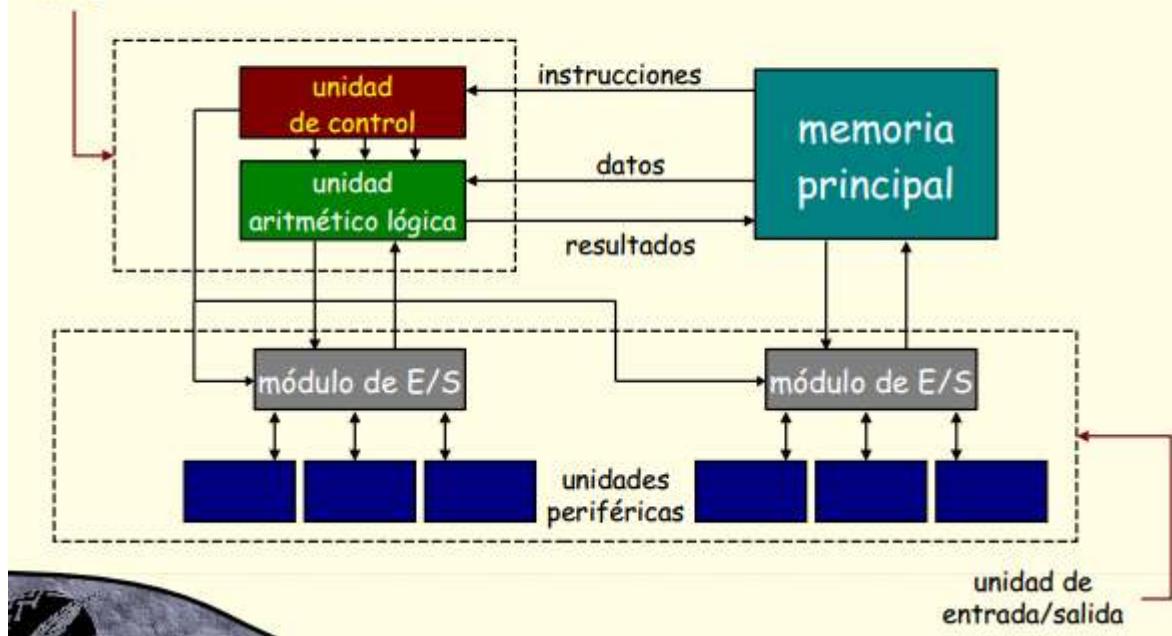
En 1951 Maurice Wilkes propuso una arquitectura con tres niveles, con el objetivo de simplificar el hardware del nivel inferior. Wilkes en esencia invento la microprogramación. Es simplificación a nivel de HW se logra en virtud de que el repositorio de las microinstrucciones es menor que el de las instrucciones a nivel de la maquina convencional. En esa época las computadoras todavía se construian con válvulas, por lo que esta simplificación a nivel de HW implicaba un cuantioso ahorro.

La década del '50 tambien se caracteriza por el desarrollo de los primeros lenguajes de programación. Los lenguajes ensamblador de las distintas computadoras de la época (tales como la EDVAC, la primer computadora de programa almacenado). Los lenguajes de programación imperativos Fortram (1957) y Cobol (1959) así como el lenguaje de programación funcional Lisp (1958) fueron diseñados e implementados a lo largo de esa década.

Alrededor de la decada del '60 se intento automatizar las repetitivas tareas que tenían que llevar a cabo los operadores. Recordemos lo enjundioso que resultaba ejecutar un programa escrito en alguno de estos lenguajes de programación en la época de las tarjetas perforadas. Un programa llamado sistema operativo cuyo propósito era asistir al operador se mantenía en ejecución todo el tiempo. Recordemos también que el lenguaje de este nuevo nivel es híbrido, esta compuesto mayormente por las instrucciones del nivel inmediato inferior y solo unas pocas nuevas instrucciones propias. Las nuevas instrucciones se conocían como macros al sistema operativo o también llamadas al supervisor. Un importante avance que incorporaron los sistemas operativos fue posibilitar el compartir el HW entre multiples usuarios en simultaneo, dando a lugar a lo que se conoce como sistema de tiempo compartido

# Arquitectura von Neumann

## CPU



Recordemos que en el modelo sugerido por la arquitectura von Neumann, la memoria principal es utilizada para almacenar tanto programas como datos.

La memoria se clasifica en función de distintos aspectos:

Por la forma de acceso:

- Memoria de acceso secuencial: este tipo de memoria estipula que los datos en ella almacenados deben ser accedidos en un determinado orden secuencial
- Memoria de acceso aleatorio: esta es la memoria convecional, en la cual no existe restricción alguna en el orden en el cual se deben acceder los datos en ella almacenados
- Memoria de acceso por contenido: esta memoria, denominada asociativa, se caracteriza por ser capaz de resolver búsqueda por contenido.

Por la forma de retención de los datos:

- Memoria estática: en este tipo de memoria una vez almacenado un dato el mismo queda retenido sin ser alterado hasta que vuelva ser modificado. Logra este comportamiento insumiendo mayor cantidad de componentes por cada celda
- Memoria dinámica: en este tipo de memoria los datos son retenidos por breves intervalos de tiempo. Es decir, el sistema debe leer y volver a escribir cada locación de memoria antes de que se pierda su contenido. El beneficio es que insume una menor cantidad de componentes por cada celda.

Por el grado de permanencia de la información:

- Volátiles: en este tipo de memoria la información almacenada se pierde en el mismo instante que se corte el suministro de corriente eléctrica
- No volátiles: en este tipo de memoria la información almacenada perdura en el tiempo, más allá de que se disponga o no de una fuente ininterrumpida de alimentación

### Tecnologías de memoria

- Memoria ram (random access memory)
- Memoria rom (read only memory)
- Memoria prom (programmable rom)
- Memoria Eprom (erasable prom)
- Memoria eeprom (electrically erasable prom)}
- Memoria mram (magnetoresistive ram)
- Memoria estilo FLASH

### Organización de la memoria

La memoria se organiza como un arreglo de  $n$  celdas, cada una de  $k$  bits. Cada celda o locación cuenta con un identificador de  $\log_2(n)$  bits que la caracteriza denominado dirección. De igual forma, cada locación o celda almacena un contenido de  $k$  bits. Notece que a menor unidad direccionable es la celda, si bien la unidad básica es el bit. El tamaño en bits de cada una es independiente de la cantidad de bits de cada dirección.

La norma dicta que  $o$  bits componen un byte y que los bytes se agrupan en palabras. Una palabra suele tener 8, 16, 32 o 64 bits. La cantidad de bits por palabra está relacionada con el tamaño en bits de los registros del procesador. Las instrucciones en general operan sobre palabras. Las operaciones básicas son la memoria son: leer una palabra de la memoria. Escribir una palabra a la memoria

### Ordenamiento de los bytes

En caso de que una palabra esté compuesta por múltiples bytes se pueden ordenar de dos formas: el esquema big-endian postula que se debe numerar los bytes de izquierda a derecha y el esquema Little-endian postula que se deben numerar los bytes derecha a izquierda. La decisión respecto a qué esquema utilizar poco afecta al desempeño del procesador. No obstante, resulta

conflictivo transmitir palabras entre computadoras que no usen el mismo esquema

tamaño de palabra: 32 bits  
mensaje original: Hola Mundo! 01/10/2011

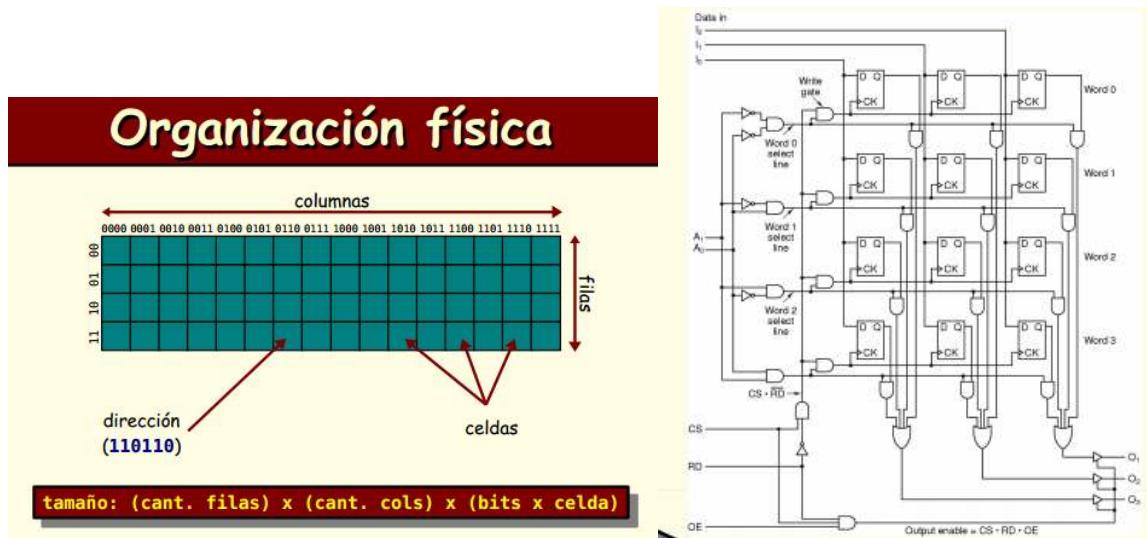
	00	01	10	11		00	01	10	11	
	'H'	'o'	'l'	'a'		'a'	'l'	'o'	'H'	
	' '	'M'	'u'	'n'		'n'	'u'	'M'	' '	
	'd'	'o'	'!'	0		0	'!'	'o'	'd'	
	1	'/'	'10'	'/'		'/'	'10'	'/'	1	
	221	7				7	221			
	big-endian					little-endian				

### Interfaz con la memoria

La unidad controladora de la memoria cuenta esencialmente con dos registros: el registro MAR el cual almacena la dirección de una locación de memoria y el registro MDR el cual almacena el contenido de una locación de memoria

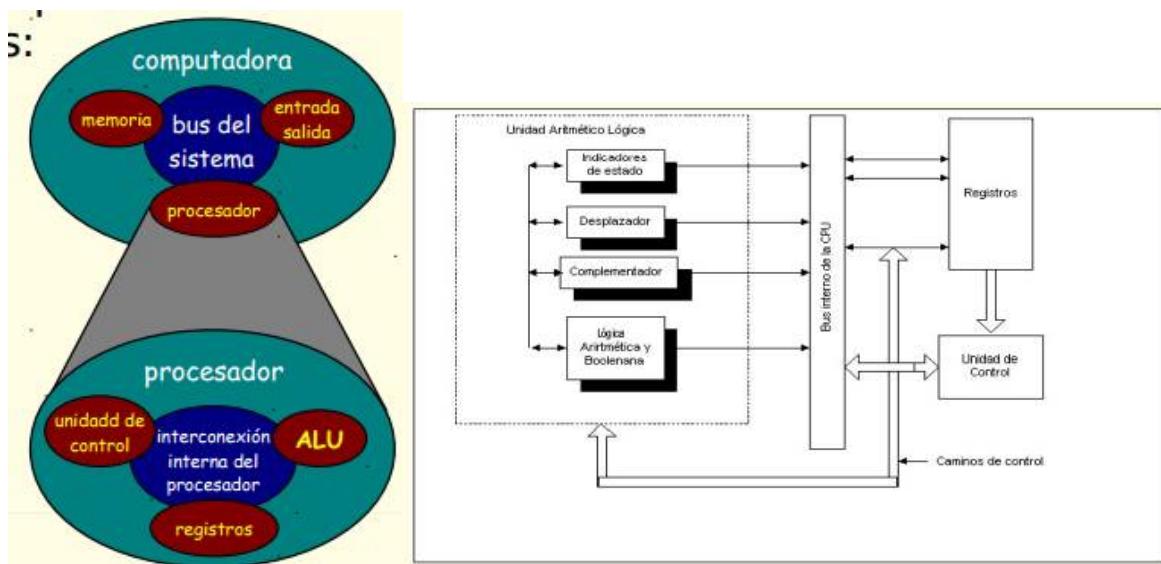
Operación de lectura(load): el procesador escribe la locación que se desea acceder en el registro MAR, luego activa la señal de lectura y finalmente accede a la palabra en cuestión la cual estará disponible en el registro MDR

Operación de escritura(store): el procesador escribe la locación que se desea acceder en el registro MAR y el contenido que se desea almacenar en el registro MDR y por ultimo activa la señal de escritura



### Organización del procesador

El procesador incluye otros dos componentes básicos de la arquitectura von Neumann: la unidad aritmético lógica (ALU) y la unidad de control. El procesador tiene como principal tarea orquestar el adecuado funcionamiento de la computadora. A través de la ALU lleva adelante el procesamiento de los datos que le indique el programa en ejecución. Internamente el procesador se puede descomponer en cuatro partes: la unidad de control, la ALU, los registros y las interconexiones. Nótece la similitud entre la organización del procesador y de la computadora.



### Funciones del procesador:

- Traer de memoria principal la próxima instrucción del programa en ejecución
- Determinar de que instrucción se trata ((decodificador)
- Agenciarse de los operandos mencionados en la intrucción en curso
- Llevar adelante el procesamiento de los datos indicados por esa instrucción
- Almacenar donde corresponda los resultados obtenidos

### Direccionamiento simbolico:

En un lenguaje de alto nivel es frecuente ver expresiones simbólicas tales como  $x=y+z$ . al bajar de nivel, estas referencias simbólicas deben ser reemplazadas por la dirección en memoria donde se almacenen esas variables. Debemos tener en cuenta que la dirección de una locación de memoria es a su vez un numero  $x=y+z \rightarrow 123=15+83$ . Para diferenciar una dirección de su contenido haremos uso de pares de paréntesis  $(123)=(15)+(83)$

- Para cumplir con las tareas asignadas el procesador repite infinitas veces el siguiente **ciclo básico de operación**:
  - **Etapa Fetch:** se almacena en el registro **IR** la instrucción apuntada por el registro **PC**.
  - **Etapa Decode:** durante esta etapa se determina de qué instrucción se trata.
  - **Etapa Effective Address:** se calcula la dirección efectiva referida por la instrucción (si es que alguna) y/o se recuperan los operandos necesarios para comenzar a ejecutar la instrucción en curso.

### Continúa:

- **Etapa Execute:** conociendo de qué instrucción se trata y contando con los operandos que sean necesarios, durante esta etapa se programa a la **ALU** para que lleve adelante el procesamiento correspondiente.
- **Etapa Memory:** las arquitecturas **RISC** cuentan con una etapa específica donde se accede a memoria para leer o escribir una determinada locación.
- **Etapa Write-Back:** usualmente el resultado obtenido se almacena en alguno de los registros del procesador durante esta etapa.

## Registros del procesador

Los registros son un almacenamiento temporal de muy alta velocidad. Se organizan en un banco de registros, el cual suele contar con multiples puertos de lectura y/o escritura. La cantidad de bits de los registros coincide con el tamaño de palabra del procesador. El tamaño de palabra nos da una idea aproximada de la capacidad de procesamiento del hardware (en particular, de la ALU)

Los registros del procesador se clasifican en dos categorías:

- Registros accesibles por el usuario: estos registros puede ser accididos por el usuario a través de las distintas instrucciones.
- Registros internos del procesador: estos registros son utilizados exclusivamente por la unidad de control durante el desarrollo del ciclo básico del procesador

A su vez los resitros accesibles por el usuario se clasifican a su vez en otras dos categorías:

- Registros de propósito general: estos registros están disponibles para que el programador haga un uso discrecional de los mismos (por caso, el registro acumulador EAX o el registro ECX).
- Registros de propósito específico: estos registros usualmente tiene asignado un rol por defecto (por caso, el registro ESP que siempre apunta a tope de la pila del programa)

Uno de los registros internos del procesador es PSW (processor status Word), el cual codifica el estado actual del procesador. Los registros internos del procesador solo pueden ser alterados de manera indirecta. Por caso, el devenir de la ejecución de un programa va alterando los registros internos PC e IR. De manera análoga, el resultado de la ultima operación realizada afectara al registro interno PSW

## Acceso a los operandos

Una de las etapas del ciclo básico del procesador requiere ganar acceso a los operando de la instrucción en curso. A lo largo de la evolución de la computación se ensayaron muy diversas formas de especificar la ubicación de estos argumentos. Estas marcadas diferencias permiten clasificar a las distintas arquitecturas en función de cuantas referencias a memoria tienen las instrucciones

**ARquitcturas 0-address:** solo cuenta con instrucciones que refieren a sus operandos de manera implícita. También se la conoce como arquitectura pila. La pila del sistema es origen y destino implícito de todas las operaciones aritmético-logicas. Naturalmente, las instrucciones de acceso a memoria son una excepción, ya que en si explicitan la locación que se esta accediendo.

**Arquitectura 1-address:** cuenta con instrucciones en las que se especifica solo uno de los argumentos. También se la conoce como arquitectura de acumulador único. Existe un registro especial llamado acumulador el cual es origen y destino implícito de todas las operaciones aritmético-logicas. Puede contar con registros auxiliares los que se emplean para implementar modos de direccionamiento mas complejos

**Arquitectura 1-address+1:** cuenta con instrucciones en las que esta permitido especificar un registro general a la par de una dirección de memoria. La arquitectura Intel x86 pertenece a esta

categoría. En las operaciones con dos operandos de entrada el registro específico dentro de la instrucción se convierte en el destino implícito del resultado. Este tipo de arquitectura cuenta con la instrucción MOV para mover información desde y hacia memoria y también entre los registros.

**Arquitectura 2-address:** cuenta con instrucciones en las que esta permitido especificar hasta dos direcciones de memoria. En las operaciones con dos operandos de entrada, uno de los argumentos especificados dentro de la instrucción se convierte en el destino implícito. Este tipo de arquitectura también cuenta con la instrucción MOV para mover información libremente entre memoria y registros

**Arquitectura 3-address:** cuenta con instrucciones en las que esta permitido especificar hasta tres direcciones de memoria. Este tipo de arquitectura evidentemente brinda la máxima flexibilidad a los programadores del sistema que estén a cargo de implementar los distintos compiladores. No obstante, el tamaño de las instrucciones has de variar en gran medida en función de la cantidad de direcciones de memoria que se especifiquen.

**Arquitectura reg a reg:** cuenta con instrucciones que solo operan sobre registros. Todas las arquitecturas RISC adoptan esta configuración. De manera análoga a la arquitectura pila, se cuenta con instrucciones específicas de acceso a memoria las cuales permiten especificar usualmente a lo sumo una dirección de memoria. Los modos de direccionamiento mas avanzados se logran combinando esa dirección de memoria con uno o mas registros.

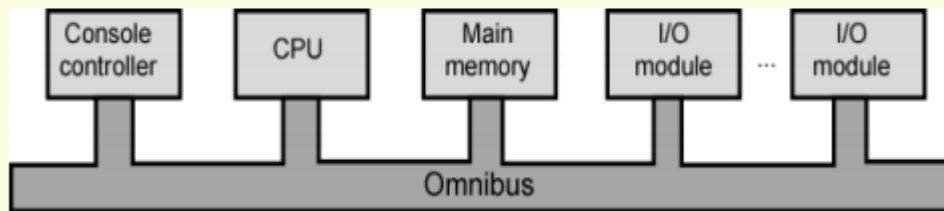
### Subsistema de entrada/salida

El ultimo componente de la arquitectura von Neumann que nos resta abordar se lo conoce como subsistema de entrada/salida. Las computadoras rápidamente demandaron almacenar más información de la que se podía almacenar en la memoria principal. En consecuencia, a lo largo del tiempo se han ensayado distintas tecnologías para implementar esta memoria secundaria

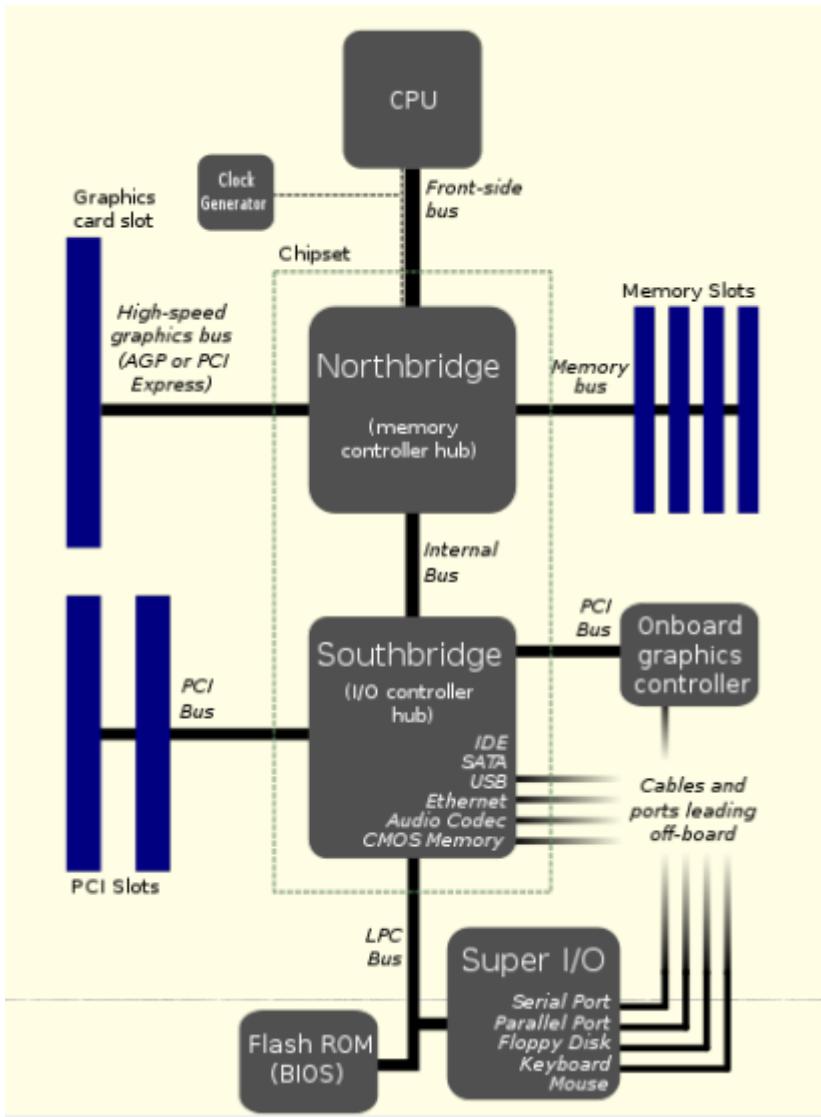
### Bus del sistema:

La arquitectura von Nuemann presenta un serio cuello de botella: el canal de comunicación entre el procesador y el resto de la computadora. Las primeras computadoras adoptaban un esquema de bus único.

#### → Por ejemplo, el Omnibus de la PDP-8:



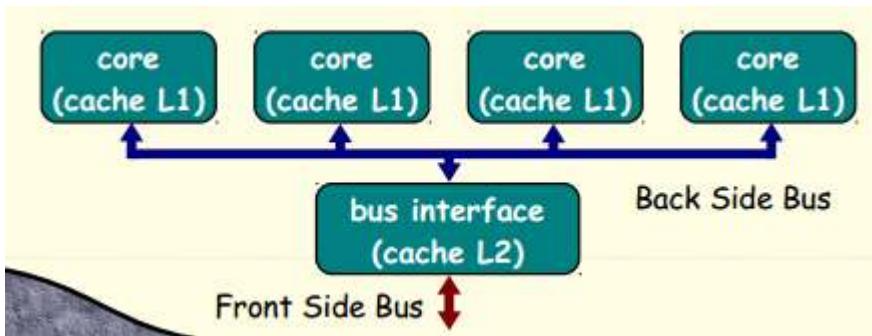
Las arquitecturas mas recientes empezaron a hacer uso de multiples buses independientes. Por caso una típica pc de escritorio cuenta con gran cantidad de buses de propósito específico. Recientemente las funciones de nothbridge se han incorporado al procesador



### FBS vs BSB

En lo que al CPU respecta, cuenta básicamente con dos buses:

- El Front Side Bus (FSB) para interconectar el procesador con el resto de la computadora
- El Back Side Bus (BSB) para interconectar los componentes internos del procesador entre si.



# Diapositiva 6

## Tipos de Instrucción

De procesamiento: Las instrucciones usuales aritméticas lógicas.

De acceso a memoria: para transferir información desde y hacia memoria.

De transferencia de datos: para enviar y recibir información de los dispositivos de entrada/salida.

De control: Las instrucciones que alteran el flujo de control del programa en ejecución.

## Formato de instrucción

Las instrucciones a nivel de lenguaje de máquina están compuestas por una secuencia de unos y ceros. Esta secuencia debe codificar:

- De qué instrucción se trata.
- El conjunto de argumentos (tanto de origen como destino) sobre los cuales se ha de operar. No necesariamente todas las instrucciones contienen la misma cantidad de argumentos.
- Opcionalmente, una referencia a la próxima instrucción.

Una instrucción máquina se compone de: un campo opcode, de tamaño fijo o variable, el cual codifica de qué instrucción se trata; y de cero o más campos argumento, dependiendo el tipo de instrucción codificado por el campo de opcode. Estos últimos describen cómo y dónde encontrar los argumentos.

## Diseño del set de instrucciones

En el diseño del set de instrucciones se define el conjunto de operaciones, los tipos de datos nativos y la codificación elegida para las instrucciones. En esta etapa se busca: asegurar la completitud del set de instrucciones; poder hacer uso de la totalidad de memoria disponible; lograr que el set de instrucciones sea ortogonal con respecto a los modos de direccionamiento implementados por la arquitectura; y contar con un tamaño fijo de instrucciones.

## Ortogonalidad

Denominaremos a un set de instrucciones como ortogonal cuando los modos de direccionamiento implementados estén disponibles a todos los tipos de instrucciones. La ortogonalidad brinda una mayor libertad a los programadores, pero suele extender la longitud promedio de las instrucciones, lo que implica que hará falta un mayor ancho de banda a memoria.

## Tamaño fijo de instrucciones

Todas las instrucciones tienen el mismo tamaño. Esto simplifica notablemente la etapa de decodificación de la instrucción, ya que el opcode ocupa una posición conocida y fija dentro de la secuencia de 1's y 0's que codifica la instrucción. No obstante, el tamaño debe ser lo suficiente

grande como para codificar todas las instrucciones, lo cual obliga a contar con un amplio ancho de banda a memoria.

### Tamaño variable de instrucciones

Todas las instrucciones tienen el mínimo tamaño posible. Esto posibilita contar con instrucciones de gran tamaño, por caso, con múltiples referencias a memoria pero se complica y posiblemente ralentiza la decodificación de las instrucciones. Esta alternativa no es apropiada para las arquitecturas superescalares, ya que requiere decodificar múltiples instrucciones en cada ciclo.

### Modo intermedio

Un cierto formato de instrucciones fijo de  $k+n$  bits, cuenta con  $k$  bits para opcode  $n$  bits para un único operando. Este formato permite a lo sumo  $2^k$  opcodes y hacer referencia a lo sumo  $2^n$  direcciones de memoria.

### Tamaño variable de operandos

En general, se requieren menos bits para codificar un registro que una dirección completa de memoria. La expansión del opcode también permite la coexistencia de operandos de distinto tamaño.

Código automodificable: Para recorrer tipos de datos complejos (tales como arreglos o registros), se tiene que modificar en tiempo de ejecución el campo argumento de la instrucción. A este tipo de código se lo denomina automodificable.

### Modos de Direccionamiento

Modo Inmediato: Se accede al valor deseado directamente.

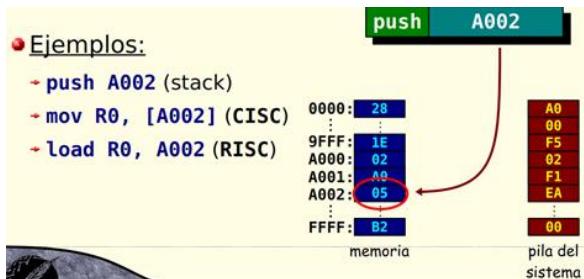
Modo Indirecto: la idea detrás del modo indirecto consiste en agregar un paso adicional en la búsqueda del argumento. Se accede a una dirección en la cual luego encontraremos el valor deseado.

Inmediato (literal): el contenido del campo argumento es operando. Altamente eficiente, pues no requiere de acceder a memoria. El rango puede estar acotado a la cantidad de bits disponibles en el formato de la instrucción. Se lo utiliza para operar con valores constantes.

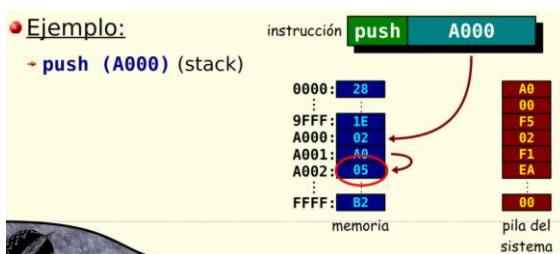


Absoluto (Memoria directo): el contenido del campo argumento es la ubicación en memoria del operando. Implica un acceso de memoria adicional (aparte del acceso inicial para traer la

instrucción). La instrucción contiene la dirección efectiva del argumento. El espacio de direcciones está acotado por la cantidad de bits permitidos por el formato de instrucción.



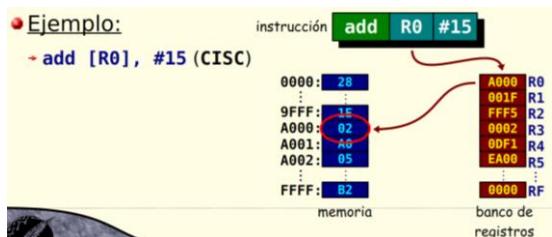
Absolute Indirecto (memoria indirecto): El contenido del campo argumento es la ubicación en memoria de la ubicación del operando. Implica dos accesos adicionales a memoria (aparte del acceso inicial para traer la instrucción). Es extremadamente flexible.



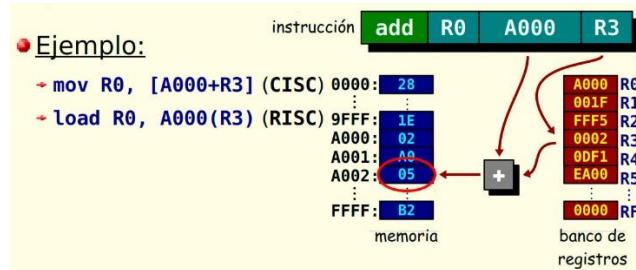
Registro (Registro directo): El contenido del campo argumento es la ubicación del operando dentro del banco de registros. Es análogo al modo absoluto, usando un banco de registros del procesador como espacio de direcciones. Es altamente eficiente, ya que no requiere de acceder a memoria (aparte del inicial).



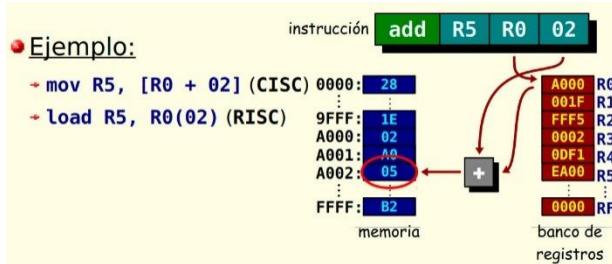
Registro Indirecto: El contenido del campo argumento es la ubicación dentro del banco de registros de la dirección del operador. Es análogo al modo absoluto indirecto, pero recuperando la dirección del operando del banco de registros. Se puede codificar en muy pocos bits. Al ser indirecto retiene la capacidad de direccionar la totalidad del espacio de direcciones. Es altamente flexible.



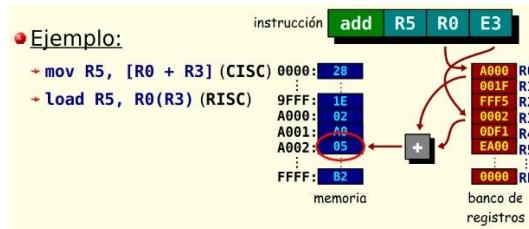
**Indexado:** El campo argumento contiene a una dirección de base fija y un desplazamiento variable que sumados dan la ubicación en memoria del operando. Es relativamente eficiente, puesto que requiere de sólo un acceso a memoria (aparte del inicial). La dirección de base de la estructura tiene que ser una dirección completa. Está pensado para recorrer múltiples elementos de una única estructura. En el modo indexado codificamos una dirección completa, esto es, un entero positivo que necesita de tantos bits como tengan las direcciones de memoria.



**Base:** El campo argumento contiene un desplazamiento fijo y una dirección de base variable que sumados dan la ubicación en memoria del operando. Es relativamente eficiente, puesto que requiere sólo un acceso a memoria (aparte del inicial). Está pensado para acceder a un único elemento en múltiples estructuras de datos. En el modo base codificamos un desplazamiento signado, el cual es práctico incluso en caso de contar con pocos bits.



**Base-Indexado:** El campo argumento contiene una dirección de base y un desplazamiento, ambos variables, que sumados dan la ubicación en memoria del operando. Es relativamente eficiente, puesto que requiere sólo un acceso a memoria (aparte del inicial). Además es muy flexible, permite recorrer múltiples elementos almacenados en múltiples estructuras de datos.

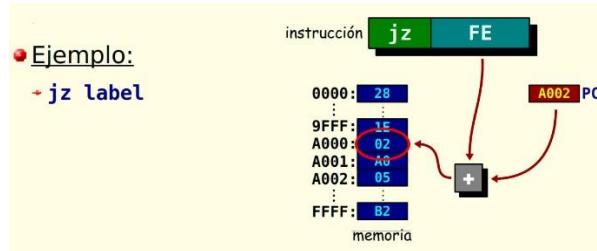


**Base-indexado indirecto:** Es combinación de los modos Base-Indexado y Absoluto Indirecto. La misma crítica realizada acerca del modo absoluto indirecto se aplica en este nuevo contexto.

**Pre-Indexado Indirecto:** el registro base y el registro índice son sumados para obtener la dirección en memoria de la dirección del operando.

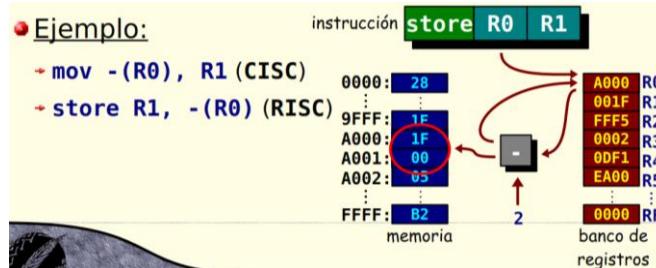
**Post-Indexado Indirecto:** El registro base contiene la dirección en memoria de un valor el cual sumado al registro índice permiten obtener la dirección del operando.

**PC Relativo:** El campo argumento contiene un desplazamiento fijo el cual de ser sumado al contenido del registro PC para obtener la dirección referida. Requiere un único acceso a memoria (aparte del acceso inicial para traer la instrucción). El desplazamiento, un entero signado, suele estar representado en dos complementos.



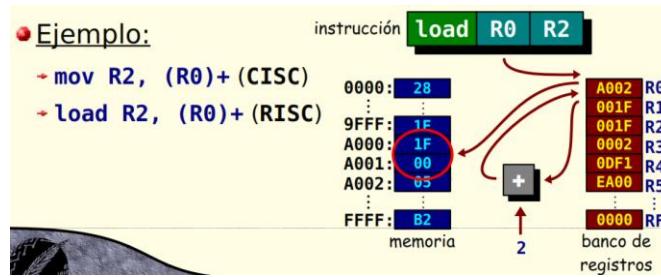
\*Auto-decrement: El campo argumento contiene una referencia a un registro el cual primero es decrementado y luego es usado como la dirección en memoria del operando. Es relativamente eficiente, puesto que requiere sólo un acceso a memoria (aparte del inicial). Permite gestionar con facilidad una estructura de pila, especialmente a la hora de implementar la operación de apilado (push). Es equivalente a ejecutar la siguiente secuencia de instrucciones (asumida una arquitectura de 16 bits):

```
dec R0; dec R0; store R1, -(R0);
```



\*Auto-increment: El campo argumento contiene una referencia a un registro es usado como la dirección en memoria del operando y luego es incrementado. Es relativamente eficiente, puesto que requiere sólo un acceso a memoria (aparte del inicial). Permite gestionar con facilidad una estructura de pila, especialmente a la hora de implementar la operación de desapilado (pop). Es equivalente a ejecutar la siguiente secuencia de instrucciones (asumida una arquitectura de 16 bits):

```
load R2, (R0); inc R0; inc R0
```



### Código independiente de la posición

Los modos de direccionamiento base y relativo permiten la construcción de código cuyo funcionamiento es independiente de la posición en la cual termine siendo cargado en memoria. La clave a fin de obtener este tipo de código radica en evitar toda forma de direccionamiento absoluto. Estos se clasifican en:

- Estático: cuando el código inicialmente puede ser cargado en cualquier locación de memoria, pero una vez cargado en memoria deja de ser relocable.
- Dinámico: cuando el código puede ser cargado en cualquier locación de memoria y de ser necesario también puede ser relocalizado en todo momento.

### Arquitectura OCUNS

OPCODE	DESCRIPCIÓN	FORMATO	PSEUDOCÓDIGO
0	add (add)	I	$R[d] \leftarrow R[s] + R[t]$
1	sub (subtract)	I	$R[d] \leftarrow R[s] - R[t]$
2	and (and)	I	$R[d] \leftarrow R[s] \& R[t]$
3	xor (xor)	I	$R[d] \leftarrow R[s] ^ R[t]$
4	lsh (left shift)	I	$R[d] \leftarrow R[s] << R[t]$
5	rsh (right shift)	I	$R[d] \leftarrow R[s] >> R[t]$
6	load (load)	I	$R[d] \leftarrow \text{mem[offset} + R[s]\text{]}$
7	store (store)	I	$\text{mem[offset} + R[d]\text{]} \leftarrow R[s]$
8	lda (load address)	II	$R[d] \leftarrow \text{addr}$
9	jz (jump zero)	II	$\text{if } (R[d] == 0) \text{ PC} \leftarrow \text{PC} + \text{addr}$
A	jg (jump greater)	II	$\text{if } (R[d] > 0) \text{ PC} \leftarrow \text{PC} + \text{addr}$
B	call (jump and link)	II	$R[d] \leftarrow \text{PC}; \text{PC} \leftarrow \text{addr}$
C	jmp (jump register)	III	$\text{PC} \leftarrow R[d]$
D	inc (increment)	III	$R[d] \leftarrow R[d] + 1$
E	dec (decrement)	III	$R[d] \leftarrow R[d] - 1$
F	hlt (halt)	III	exit

Es una arquitectura estilo RISC. Sus instrucciones son de tamaño fijo de 16 bits. El espacio de direcciones es de 256 bytes. Cuenta con 16 registros de propósito general, si bien el registro F se encuentra cableado a cero. El formato de instrucción adoptado hace uso de la técnica de expansión del opcode para determinar el tipo y número de los argumentos. La arquitectura OCUNS cuenta con un reducido número de modos de direccionamiento:

- El modo registro directo está disponible en las operaciones aritmético lógicas (formatos I, II, y III).
- El modo registro más desplazamiento está disponible en las instrucciones de transferencia de información desde y hacia memoria (formatos I y II).
- El modo absoluto y PC-relativo están disponibles en las instrucciones de transferencia de control (formatos II).

FORMATO							
	1	5	4	3	2	1	0
<b>I</b>	0	x	x	dest. d	src. S	src. t/off.	
<b>II</b>	1	0	x	x	dest. d	address	addr
<b>III</b>	1	1	x	x	dest. d	-	

## Diapositiva 7

### Aritmética punto fijo

La FXP es conveniente para representar números de base pequeña con órdenes de magnitud acotados. Por caso, operando con 32 bits de precisión, el rango a manejar está acotado por  $\pm(2^{31}-1)$ , lo cual corresponde en base 10 a  $\pm(10^{11})$ . Evidentemente este rango es inadecuado en un contexto ingenieril o para aplicaciones científicas, donde se manejan magnitudes más grandes y también más pequeñas.

### Aritmética punto flotante

Una alternativa superadora consiste en hacer uso de la notación científica, también conocida como punto flotante (FLP). Un cierto número real  $f$  se denota como el producto entre una mantisa  $m$  y un exponente  $e$ , de la forma  $f=m \cdot b^e$ . Tanto  $m$  como  $e$  son números signados expresados en una cierta base, mientras que  $b$  (implícito) es un entero que denota la base adoptada para la representación.

### Inconvenientes con FXP

Las primeras computadoras hacían uso exclusivo de aritmética FXP. Al realizar cálculos científicos se debía redondear las magnitudes a menudo, a fin de mantener acortado el número de dígitos significativos. Los problemas generados en este tipo de aplicaciones versan en torno al rango, la precisión y el nivel de significancia de los valores representados. Estas primeras computadoras hacían un uso intensivo del intervalo unitario  $[-1,+1]$ . La idea es que cuando el rango de un número se hace muy grande o muy pequeño durante el cómputo, el programador se encargue de seguir la posición del punto en todos los resultados intermedios. En muchos casos, el número debía ser escalado hacia arriba o hacia abajo para caber en ese intervalo. Naturalmente, al final del cómputo el resultado debía ser transformado nuevamente al dominio requerido por el usuario del sistema.

El problema del escalamiento incluye también la selección de un factor de escala adecuado: Todo número en notación FXP de  $n$  dígitos de precisión en una base  $b$  al hacer uso de un factor de escala  $b^k$  tendrá un valor absoluto menor que  $b^k$ , resultando en un máximo error de  $b^{k-n}$ . Para incrementar la precisión, se propuso hacer uso de aritmética de punto fijo de múltiple precisión (es decir, usar múltiples palabras). Claro está, hacer uso de más de una palabra para cada magnitud normalmente implica un mayor costo en tiempo de ejecución y en espacio de almacenamiento.

Para cálculos largos o complicados, este escalamiento y extensión de precisión implica un arduo análisis matemático y cómputo lateral para seguir los factores de escala o controlar las longitudes

de las palabras. Usualmente el máximo factor de escala será el usado en todo el conjunto de números.

Por caso, la diferencia actual entre un factor de escala común  $b^k$  para un orden de magnitud  $b^z$  en un cierto número con algún  $z < k$ , creará  $k-z$  ceros a la izquierda en los números en notación FXP, dando sólo un máximo posible de  $n-(k-z)$  dígitos significantes, en lugar de los  $n$  esperables. En una cadena larga de cálculo la perdida sucesiva de significancia originaria situaciones singulares, las cuales el hardware no está en condiciones de manejar excepto con la intervención del programador.

En respuesta a todos estos cuestionamientos, se introdujo la representación de punto flotante en la década del '40.

FLP Normalizada: Opera con operandos normalizados a fin de preservar en todo momento la mayor cantidad de bits de significancia.

FLP No Normalizada: opera con operando sin normalizar, dejando la decisión de llevar adelante el proceso de normalización en manos del programador.

### **Formatos de punto flotante**

La elección de un formato adecuado para representar números en punto flotante debe contemplar multiples aspectos:

- Evidentemente las características elegidas para  $m$ ,  $e$  y  $b$  afectarán tanto el rango y como precisión de la representación resultante. En particular,  $m$  y  $e$  comparten una o más palabras, por lo que asignar más bits de precisión a uno redunda en quitárselos al otro
- Incrementar el rango asignando más dígitos al exponente va a restringir a la cantidad de dígitos disponibles para la mantisa, lo cual disminuirá la precisión. En contraste, asignar una menor cantidad de bit al exponente conduce exactamente a lo opuesto (mejora la precisión a costa del rango).
- Por último, el tercer parámetro, la base  $b$  elegida para la representación, tiene a su vez impacto tanto en el rango como en la precisión.

También hay otras decisiones menos críticas:

- La mantisa puede ser entera o fraccionaria (según se asuma el punto binario en el extremo derecho o izquierdo, respectivamente). A su vez, la mantisa puede estar representado de cualquiera de las formas vistas.
- El exponente será necesariamente un entero signado, si bien existen distintas formas de representación.
- Finalmente, la base se elegirá de acuerdo a la longitud de palabra y a la precisión y al rango deseado, pero siempre ha de ser potencia de 2.

### **Consideraciones acerca del exponente**

El exponente se suele representar en notación exceso. Para  $n$  bits, los valores representables se polarizan en  $2^{n-1}$ , es decir, el menor exponente representable es 0 y el mayor es  $2^n - 1$ , simplificando las comparaciones.

### Consideraciones acerca de la base

La base, al ser implícita, no consume espacio en la representación, por lo que podemos vernos tentados de adoptar una base grande con el objeto de aprovechar un mayor rango. No obstante, el resultado al hacer uso de bases elevadas no fue satisfactorio (se perdía precisión).

### Consideraciones acerca de la mantisa

Con respecto a la mantisa, se deben tomar dos decisiones: qué representación adoptar y si se va a trabajar con mantisas normalizadas o no. Estas decisiones están relacionadas, ya que la normalización de las mantisas es un procedimiento que variará en función de la representación elegida. La idea básica del proceso de normalización consiste en descartar aquellos dígitos que no estén aportando significancia a la mantisa (por así decir, los ceros a la izquierda).

### Proceso de Normalización

El proceso de normalización consiste en alterar la mantisa de manera controlada, esto es, sin que se afecte el valor representado. Es decir, en caso de desplazar los bits de la mantisa a derecha, se debe compensar el corrimiento a izquierda del punto binario incrementando el exponente de manera acorde. Caso contrario, en caso de desplazar los bits de la mantisa a izquierda, se debe compensar el corrimiento a derecha del punto binario excrementando el exponente de manera acorde.

### Normalización en SM

La representación FLP es inherentemente redundante, el mismo número puede ser representado en más de una forma. En general es deseable en una implementación adoptar una única forma canónica. Si la mantisa está codificada en SM fraccionaria y la base adoptada es 2, se dice que la mantisa está normalizada si el dígito a la derecha del punto binario es distinto de 0. Ej:  $0.1 \times 10^{19}$ .

### Normalización en RC y DRC

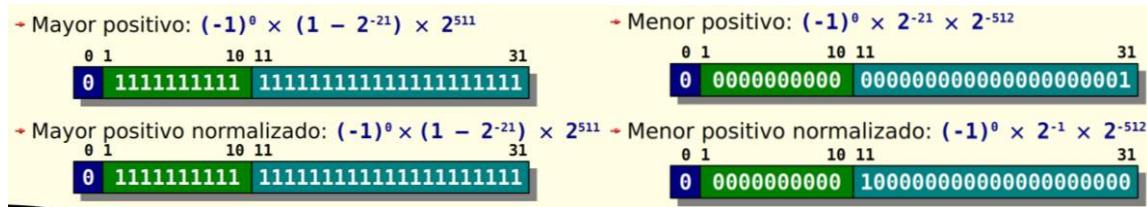
Una fracción binaria en RC o en DRC se dice normalizada cuando el bit de signo difiere del bit a su lado (el bit más significativo de la mantisa). Notar que no habrá ceros al comienzo de la mantisa de un número normalizado positivo ni habrá unos al comienzo de un número normalizado negativo. El rango de valores para una mantisa normalizada  $m$  representada en complemento a la base  $r$  resulta  $1/b \leq |m| < 1$ .

### Hidden bit

En caso de trabajar con mantisas normalizadas y una base implícita 2, aparece la posibilidad de optimizar la significancia. Considerando que sólo se opera con magnitudes normalizadas, sabemos de antemano que el primer bit de significancia  $s$  siempre distinto al bit de signo. Por ende, como el bit de signo siempre se almacena, se puede prescindir de este primer bit de significancia. Esta optimización, que permite duplicar la precisión de la mantisa, sólo es aplicable en este contexto en particular.

## Overflow y underflow

Las operaciones de punto flotante pueden dar lugar a situaciones singulares aun cuando estas sean aplicadas a datos perfectamente válidos. Por caso, para una representación FLP con 32 bits de precisión, 10 bits para el exponente (representado en exceso-512) y 22 para la mantisa fraccionaria en SM y base 2:



## Orden de Magnitud Cero (OMZ)

Todo número en notación FLP cuya mantisa m sea cero se denominan orden de magnitud cero. El exponente e puede asumir cualquier valor válido. Estos números surgen de efectuar el siguiente cálculo, para la mantisa m y un exponente e arbitrarios:  $(m,e)-(m,e)=(0,e)$ . Esta familia de números representa un amplio rango de valores indeterminados que para una base b y p bits de precisión en la mantisa satisfacen que  $-b^{e-p} < (0,e) < b^{e-p}$ .

Entre los valores que pertenecen al OMZ se distingue aquel que tiene exponente cero como el verdadero cero. La totalidad de los números positivos y negativos representables son:

$$+\varepsilon < +N < +\infty ; -\infty < -N < -\varepsilon$$

$+\infty$  y  $-\infty$  = Quasi-infinitos positivos y negativos correspondientes a los números en notación FLP cuyo exponente excede el máximo valor positivo representable.

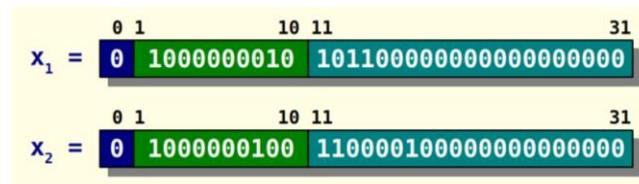
$+\varepsilon$  y  $-\varepsilon$  = Números en notación FLP cuyo exponente excede al máximo valor negativo representable.

El verdadero cero constituye la línea divisoria entre estos rangos positivos y negativos.

## Suma y Resta

Detectar el operando con mayor exponente. Desplazar la mantisa del operando con menor exponente  $|e_1 - e_2| * \log_2 b$  lugares a la derecha, ajustando según corresponda su exponente, Sumar o restar las mantisas (ya que ahora se trata de dos números con un mismo exponente). De ser necesario, normalizar el resultado obtenido.

Ejemplo: Haciendo uso de un formato FLP con p = 22, q = 10 y b = 2, se desea calcular la suma de los valores  $x_1=2.75$  y  $x_2=12.125$ .



Desplazamos las mantisas del operando con menor exponente,  $x_1$ ,  $|514-516| * \log_2 2 = 2$  lugares a derecha, ajuntando el exponente de manera acorde.

$x_1 =$	0 1	10 11	31
	0 1000000100	00101100000000000000000000000000	

• Luego, sumamos las mantisas:

$x_1 + x_2 =$	0 1	10 11	31
	0 1000000100	11101110000000000000000000000000	

Nótese que el resultado intermedio obtenido ya está normalizado, por lo que ese es el resultado final. Finalmente, podemos volver a convertir a notación FXP el resultado antes obtenido, a manera de verificación de que todo haya salido bien:

$$\begin{aligned}x_1 + x_2 &= (-1)^{sg} * m * b^e \\&= (-1)^0 * 0.9296875 * 2^4 \\&= 14.875\end{aligned}$$

## Multiplicación

Sumar los dos exponentes (este será en principio el exponente del resultado). Multiplicar las dos mantisas (lo que a su vez determina el signo del resultado). Normalizar el resultado de ser necesario, corriendo la mantisa a la izquierda y ajustando el exponente de manera acorde.

Ejemplo: Haciendo uso de un formato FLP con  $p = 22$ ,  $q = 10$  y  $b = 2$ , se desea calcular el producto de los valores  $x_1 = 2.5$  y  $x_2 = 12.125$ .

• Inicialmente tenemos que:
$x_1 =$ 0 100000010 10100000000000000000000000000000
$x_2 =$ 0 1000000100 11000010000000000000000000000000

El exponente del resultado resulta (luego de ajustar de manera adecuada el exceso):

$x_1 \times x_2 =$	0 1	10 11	31
	- 1000000110	-----	

Luego, la mantisa y el signo resultan (en este paso en principio descartamos los  $p$  bits menos significativos del producto):

$x_1 \times x_2 =$	0 1	10 11	31
	0 1000000110	100001010110000000000000	

Nótese que el resultado intermedio obtenido nuevamente ya está normalizado, por lo que ese es el resultado final. Finalmente, podemos volver a convertir a notación FXP el resultado antes obtenido, a manera de verificación de que todo haya salido bien:

$$\begin{aligned}x_1 * x_2 &= (-1)^{sg} * m * b^e \\&= (-1)^0 * 0.52099609375 * 2^{64} \\&= 33.34375\end{aligned}$$

## División

Restar los dos exponentes (este será en principio el exponente del resultado). Dividir las dos mantisas (lo que a su vez determina el signo del resultado). Normalizar el resultado de ser necesario, corriendo la mantisa a la derecha y ajustando el exponente de manera acorde.

## Observaciones

Al calcular una suma o una resta, la mantisa del resultado se ubica necesariamente en el siguiente rango:  $0 \leq |m| < 2$ . Esto constituye un inconveniente cuando  $1 \leq |m| < 2$ , situación que se denomina overflow de mantisa u overflow virtual. Para solucionar esta situación simplemente se corre la mantisa un lugar a derecha, ajustando el exponente de manera acorde. En caso de obtenerse un OMZ como resultado, se debe tener en cuenta que no será posible normalizarlo.

Al calcular una multiplicación o una división es posible calcular en paralelo la mantisa y el exponente del resultado:

$$1 / b^2 \leq |m_1 * m_2| < 1 \text{ (si } m_1 \neq 0 \text{ y } m_2 \neq 0\text{)}$$

$$1 / b \leq |m_1 / m_2| < b \text{ (si } m_2 \neq 0\text{)}$$

Luego de la multiplicación, pueden suscitarse dos escenarios:

- Cuando  $1 / b \leq |m_1 * m_2| < 1$ , no hace falta corregir la mantisa (pues ya está normalizada).
- Cuando  $1 / b^2 \leq |m_1 * m_2| < 1/b$ , hay que corregir la mantisa (pues no está normalizada).

La corrección de la mantisa, de requerirse, equivale a efectuar el siguiente cálculo:

$$(m_1, e_1) * (m_2, e_2) = (m_1 * m_2 * b, e_1 + e_2 - 1)$$

Para el caso de la división, nunca hace falta normalizar, pues presentan los siguientes escenarios:

- Si  $m_1 < m_2$ , entonces  $m_1 / m_2$  ya está normalizado pues se verifica que  $1/b \leq |m_1 / m_2| < 1$ .
- Caso contrario, cuando  $m_1 \geq m_2$ , con  $m_1 \neq 0$ , se presenta un overflow virtual si  $1 \leq |m_1/m_2| < b$ , cuyo ajuste correspondiente es el siguiente:

$$(m_1, e_1) * (m_2, e_2) = (m_1 / m_2 * b^{-1}, e_1 - e_2 + 1)$$

# Diapositiva 8

## Precision de la mantisa:

En ocasiones el resultado de una operación en FLP puede exceder los p dígitos de precisión reservados para la mantisa.

**Truncado:** para el caso de la suma, el overflow virtual se compensa corriendo la mantisa a derecha una posicion. Esto usualmente tiene como efecto colateral eliminar el ultimo bit del resultado (el bit menos significativo), de manera independiente de su valor.

**Redondeo:** para el caso de la operación de multiplicación, una forma razonable de descartar los p dígitos menos significativos de entre los 2p dígitos del resultado consiste en analizar el bit mas significativo entre los que serán descartados: si ese bit es 1, entonces se deben sumar 1 LSB al resultado preliminar formado con los p dígitos mas significativos que serán conservados. Caso contrario, si ese bit es 0, entonces el resultado preliminar es el definitivo

Notese que si el bit mas significativo descartado es 1 el resultado esta en la mitad o pasada la mitad, y caso contrario al ser 0 el resultado esta necesariamente por debajo de la mitad. En otras palabras, el redondeo siempre condice al numero maquina de p dígitos de presicion mas próximo al numero completo de 2p dígitos. En consecuencia el máximo error que se puede cometer esta acotado por la expresión:  $E = b^{-p}/2 = \frac{1}{2}LSB$

Cabe señalar que independientemente de la base adoptada, sabemos que los dígitos serán representados eventualmente en base 2. Nos obstante, el umbral se puede resolver inspeccionando solo el bit mas significativo entre los dígitos a ser descartados, mas alla de la base adoptada en la representación. Esto se debe a que ese bit será necesariamente un 1 al representar valores por encima de  $b/2$  (recordemos que la base debe ser potencia de 2)

### Máximo error

Para analizar el máximo error que se puede cometer bajo cada esquema de redondeo hace falta detectar el escenario en el que peor se desempeña:

- Para el truncado: el peor caso se verifica cuando el resultado aproxima por debajo a un numero maquina; en ese caso, la magnitud descartada representará a lo sumo 1 LSB
- Para el redondeo: la situación mejora, ya que el peor caso se verifica cuando el resultado equidista de los números maquina, descartando en ese caso  $\frac{1}{2}$  lsb.

### Teoria de redondeo

Este análisis naturalmente solo contempla valores absolutos, una descripción algebraica mas rigurosa debe distinguir números positivos y negativos a los efectos de estandarizar los métodos existentes. Recordemos que la función de redondeo se define formalmente como el mapeo  $p:R \rightarrow M$  tal que verifica que para todo  $a, b$  pertenecientes a  $R$  se verifica que:  $p(a) \leq p(b)$ , toda vez que  $a \leq b$

### Redondeo Optimal

Una función de redondeo  $p$  de denoma optimal si, y solo si, para todo  $a \in M$  se verifica que  $p(a)=a$ . un redondeo optimal asegura que para todo número  $a \in R$ , con  $m_1, m_2$  dos numeros consecutivos en  $M$  tales que  $m_1 < a < m_2$ , se verifica que necesariamente  $p(a)=m_1$  o  $p(a)=m_2$

Demostración:

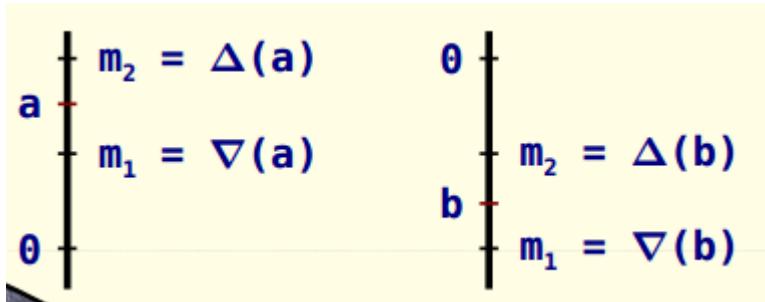
- Asumamos que  $p(a)=m'$ , con  $m'$  no perteneciente a  $[m_1, m_2]$ .
- Como  $m_1 < a < m_2$ , por definición de  $p$  obtenemos que  $p(m_1) < p(a) < p(m_2)$

- Como  $p$  es óptimo,  $p(m_1)=m_1$  y  $p(m_2)=m_2$ , por lo que  $m_1 < m' < m_2$ , absurdo, que provino de asumir que existía tal  $m'$  no perteneciente a  $[m_1, m_2]$ .
- Luego,  $m' \in [m_1, m_2]$ , lo que implica que necesariamente  $m'=m_1$  o  $m'=m_2$

se dice que un redondeo es dirigido hacia arriba si, y solo si,  $p(a) \geq a$ . De manera análoga, se dice que un redondeo es hacia abajo si, y solo si,  $p(a) \leq a$ . Finalmente, un redondeo se dice simétrico, si y solo si, para todo  $a \in \mathbb{R}$  se verifica que  $-p(-a) = p(a)$ .

- El redondeo óptimo dirigido hacia abajo  $V: \mathbb{R} \rightarrow M$  se define como:  $V(a) = \max\{m \in M \mid m \leq a\}$
- El redondeo óptimo dirigido hacia arriba  $\Lambda: \mathbb{R} \rightarrow M$  se define como:  $\Lambda(a) = \min\{m \in M \mid m \geq a\}$

En cierta forma, el redondeo óptimo dirigido hacia arriba redondea el resultado alejándose del cero en los positivos y acercándose en los negativos. El redondeo óptimo dirigido hacia abajo se comporta de manera complementaria, acercándose al cero en los positivos y alejándose en los negativos



### Redondeo simétricos

A partir de estos dos redondeos óptimos es posible definir tres redondeos simétricos que se presentan un comportamiento interesante:

- Truncado: redondea positivo y negativos siempre hacia cero
- Aumentación: redondea positivos y negativos siempre alejándose de cero
- Proximidad: redondea siempre hacia el número máquina más próximo, eligiendo el de mayor magnitud en el caso límite, al equidistar entre dos números máquina consecutivos.

#### • Truncado:

$$T(a) = \begin{cases} \nabla(a) & \text{cuando } a \geq 0 \\ \Delta(a) & \text{cuando } a < 0 \end{cases}$$

#### • Aumentación:

$$A(a) = \begin{cases} \Delta(a) & \text{cuando } a \geq 0 \\ \nabla(a) & \text{cuando } a < 0 \end{cases}$$

#### • Proximidad biased:

$$P(a) = \begin{cases} \nabla(a) & \text{cuando } \nabla(a) \leq a < \text{umbral} \\ \Delta(a) & \text{cuando } \text{umbral} < a \leq \Delta(a) \\ \nabla(a) & \text{cuando } a = \text{umbral} \text{ y } a < 0 \\ \Delta(a) & \text{cuando } a = \text{umbral} \text{ y } a \geq 0 \end{cases}$$

$$\text{con } \text{umbral} = \frac{\nabla(a) + \Delta(a)}{2}$$

La mayoría de las implementaciones hardware hacen uso de T o P. El diseñador del sistema del punto flotante debe tomar decisiones que afectan tanto la velocidad de computo como a la exactitud. Se puede demostrar que la mejor precisión se logra a través del uso de bases pequeñas y mecanismos de redondeo sofisticados. Por contraposición, la velocidad de computo se incrementa usando grandes bases y mecanismo de redondeo simples como T o A.

## Biased vs Unbiased

P presenta un comportamiento sesgado (biases) porque cuando el numero a redondear equidista se prefiere aumentar su magnitud. Es decir, existe un sesgo a la hora de resolver el caso limite entre dos números maquina. La norma IEE-754 implementa un redondeo por proximidad unbiased, prefiriendo en el caso limite los resultados pares. Esto disminuye el error acumulando producto de sucesivos redondeos.

### • Proximidad unbiased:

$$P(a) = \begin{cases} \nabla(a) & \text{cuando } \nabla(a) \leq a < \text{umbral} \\ \Delta(a) & \text{cuando } \text{umbral} < a \leq \Delta(a) \\ \Delta(a) & \text{cuando } a = \text{umbral}, a < 0 \text{ y } p_0 = 0 \\ \nabla(a) & \text{cuando } a = \text{umbral} \text{ y } a < 0 \text{ y } p_0 = 1 \\ \nabla(a) & \text{cuando } a = \text{umbral} \text{ y } a \geq 0 \text{ y } p_0 = 0 \\ \Delta(a) & \text{cuando } a = \text{umbral} \text{ y } a \geq 0 \text{ y } p_0 = 1 \end{cases}$$

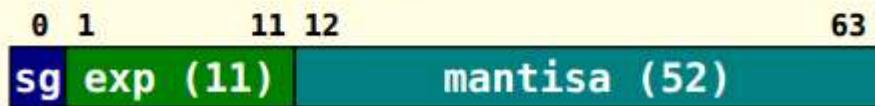
$p_0$  denota al bit menos significativo  
entre los conservados

La norma IEE-754 de 1985 es el estándar de facto a la hora de implementar en hardware una representación de punto flotante. Fue escrito en su mayor parte por el profesor W. M. Kahan (Turing Award de 1989). La norma IEE-754 contempla en principio dos formatos para los números FLP:

### • Precisión simple (32 bits):



### • Precisión doble (64 bits):



Características:

- En todos los casos se hace uso de una base  $b=2$
- La mantisa es representada usando la notación SM
- La norma sanciona una operación normalizada, por lo que la mantisa contará con un bit adicional de precisión (el denominado hidden bit).

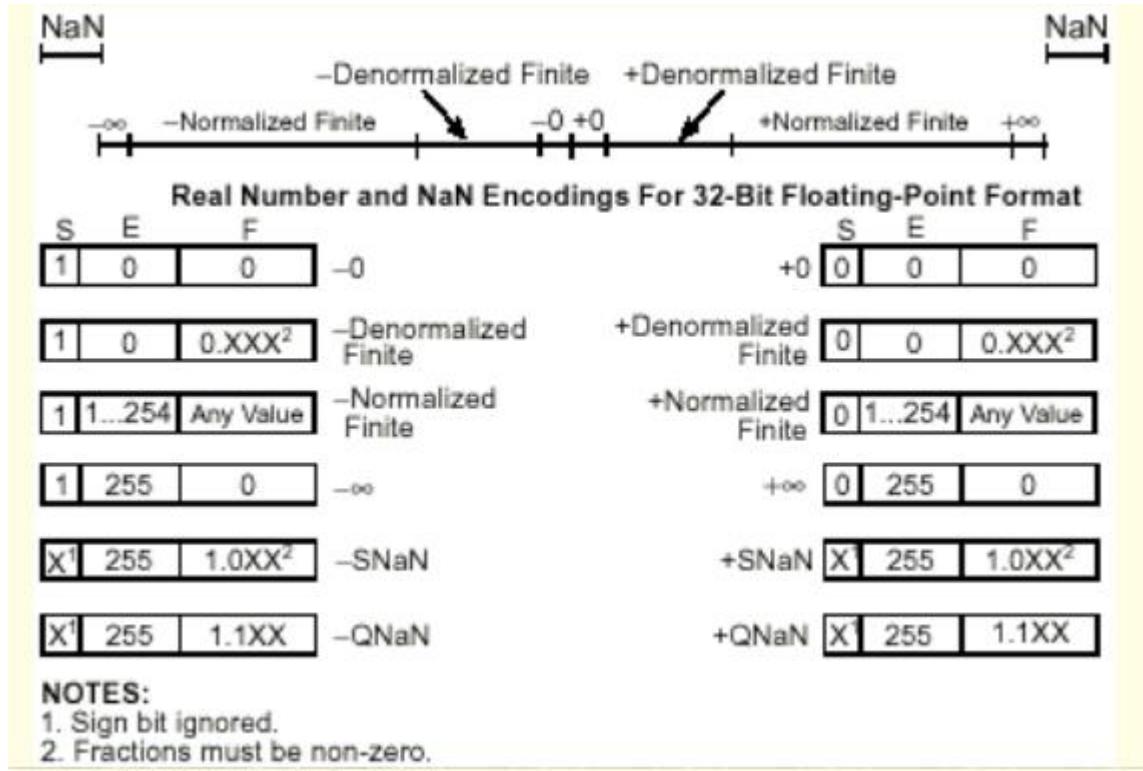
- El exponente es representado usando la notación exceso, que para el caso del formato precisión simple corresponde a exceso-127.

En síntesis, los bits almacenados por caso en el formato precisión simple se deben interpretar de la siguiente manera:



Analicemos la semántica otorgada por la norma a las combinaciones de los distintos valores posibles para los campos mantisa y exponentes:

	exp = 0	0 < exp < 255	exp = 255
mantisa = 0	0	potencias de 2	$\pm\infty$
mantisa $\neq 0$	no normalizado (denormal)	números comunes	NaN (not a number)



En síntesis, los formatos de la norma precisión simple y precisión doble brindan el siguiente rango de representación:

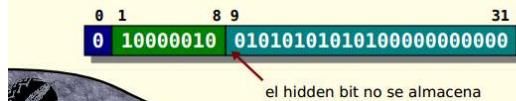
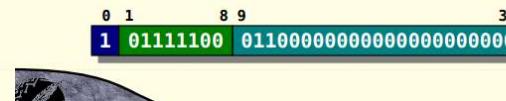
	denormalizado	normalizado	rango decimal
precisión simple	$\pm[2^{-149}, (1-2^{-23}) \times 2^{-126}]$	$\pm[2^{-126}, (2-2^{-23}) \times 2^{127}]$	$\pm[-10^{-44.85}, -10^{38.53}]$
precisión doble	$\pm[2^{-1074}, (1-2^{-52}) \times 2^{-1022}]$	$\pm[2^{-1022}, (2-2^{-52}) \times 2^{1023}]$	$\pm[-10^{-323.3}, -10^{308.3}]$

### Normalización en la norma

La norma adopta una mantisa peculiar, con exactamente un dígito a la izquierda del punto. La norma también estipula que opera solo con mantisas normalizadas, por lo que debemos adecuar su definición a este contexto: diremos que una mantisa está normalizada cuando cuenta con exactamente un 1 a la izquierda del punto. Por caso la mantisa 101.110 no está normalizada, pero la mantisa 1.01110 si lo está.

El procedimiento para convertir un número en notación FXP a la norma es bastante directo, abarca los siguientes pasos:

- Convertir a binario la parte entera usando cualquiera de los métodos vistos
- Convertir a binario la parte decimal, usando nuevamente cualquiera de los métodos vistos.
- Normalizar el resultado obtenido anteriormente, ajustando el exponente de acuerdo.
- Finalmente, expresar la mantisa y el exponente usando alguno de los formatos de la norma.

Convertir a la norma el valor $(10.666015625)_{10}$ :	Convertir a la norma el valor $(-0.171875)_{10}$ :
Parte entera: $(10)_{10} = (1010)_2$	Parte entera: $(0)_{10} = (0)_2$
Parte fraccionaria: $(.666015625)_{10} = (.101010101)_2$	Parte fraccionaria: $(.171875)_{10} = (.001011)_2$
Combinando los resultados anteriores resulta: $1010.101010101 \times 2^3$	Combinando los resultados anteriores resulta: $0.001011 \times 2^{-3}$
Normalizando, nos queda: $1.0101010101 \times 2^3$	Normalizando, nos queda: $1.011 \times 2^{-3}$
Exponente en exceso: $3 + 127 = 130 = 10000010$	Exponente en exceso: $-3 + 127 = 124 = 01111100$
	
el hidden bit no se almacena	

### Conversión entre FXP y FLP

Consideremos el siguiente fragmento de código C:

```
Int i; float f;
f = (float) i;
i = (int) f;
```

¿Son redundantes las conversiones explícitas de tipos usadas en este fragmento de código?

¿Pierdo precisión al convertir un int en un float? ¿Y al convertir un float en un int?

Denormals: existen combinaciones de valores para la mantisa y el exponente a los que se les reserva una interpretación especial. Los números cuyo exponente es cero pero su mantisa no, se denominan denormals. Los denormals admiten mantisas sin normalizar, por lo que permiten representar valores más pequeños que al utilizar mantisas normalizadas. Note que los denormals, al no estar normalizados, no cuentan con el hidden bit.

### • Número positivo normalizado más pequeño:

0	1	8	9	31
0	00000001	00000000000000000000000000000000		

$$(-1)^0 \times 1.0000000000000000000000000000000 \times 2^{-126}$$

### • Próximo positivo todavía más pequeño (se trata de un denormal):

0	1	8	9	31
0	00000000	11111111111111111111111111111111		

$$(-1)^0 \times 0.111111111111111111111111 \times 2^{-126}$$

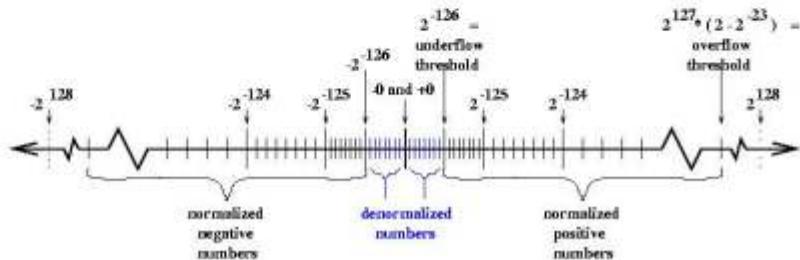
#### Normalizado vs denormals

El exponente asociado a un denormal es -126, más allá de que el valor decodificado es cero. Es decir, los denormals no codifican su exponente en exceso, ya que en exceso 0 representaría -127. Esto se debe a que la idea es permitir representar números próximos a los números ordinarios, por eso el exponente representado es el mismo (esto es, -126). Naturalmente, el exponente almacenado difiere, para poder distinguir normalizados de no normalizados.

- El número positivo más pequeño de todos (se trata de un denormal):

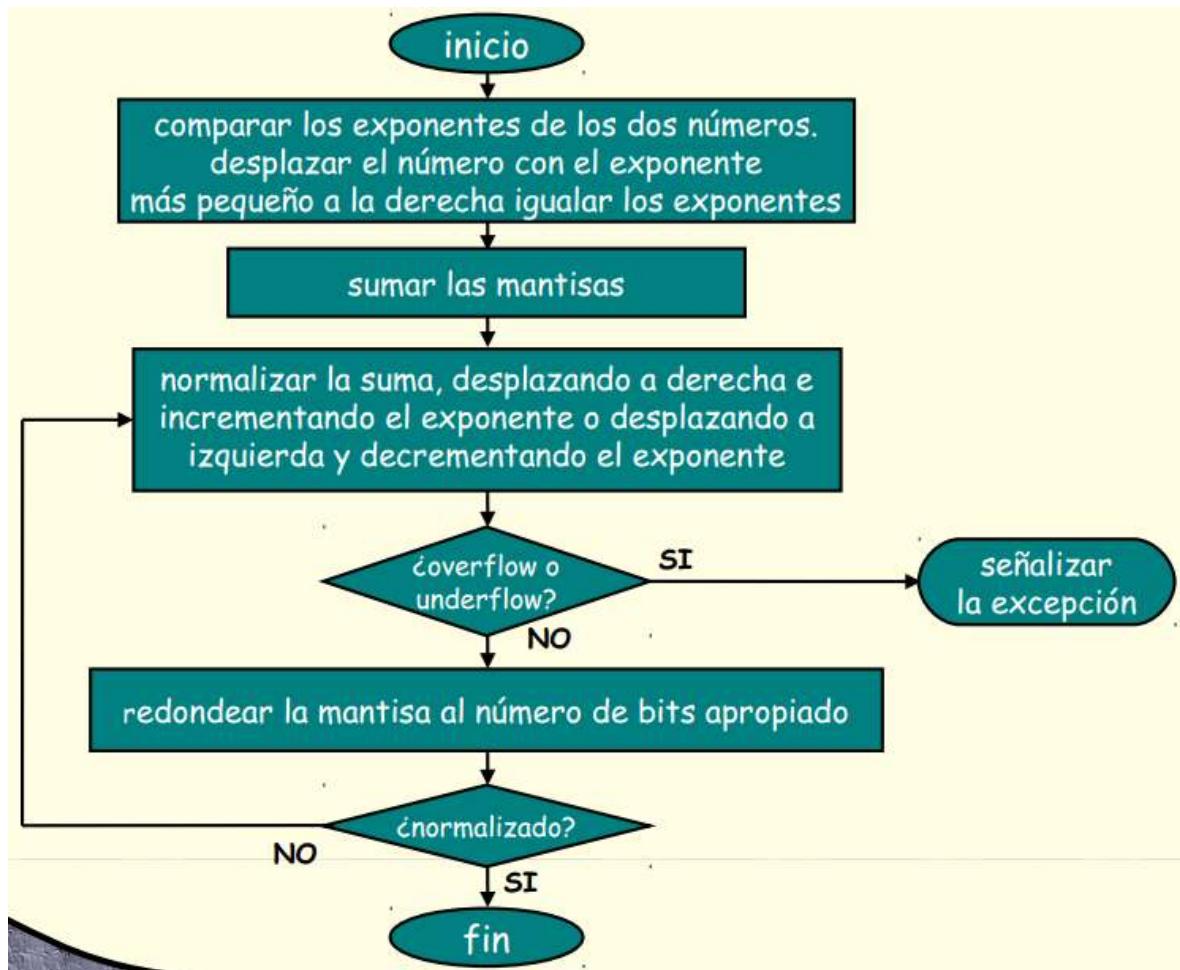
0	1	8	9	31
0	00000000	00000000000000000000000000000001		

$$(-1)^0 \times 0.00000000000000000000000000000001 \times 2^{-126} = 2^{-149}$$



Algoritmo básico para la suma de dos números en la norma IEE-754:

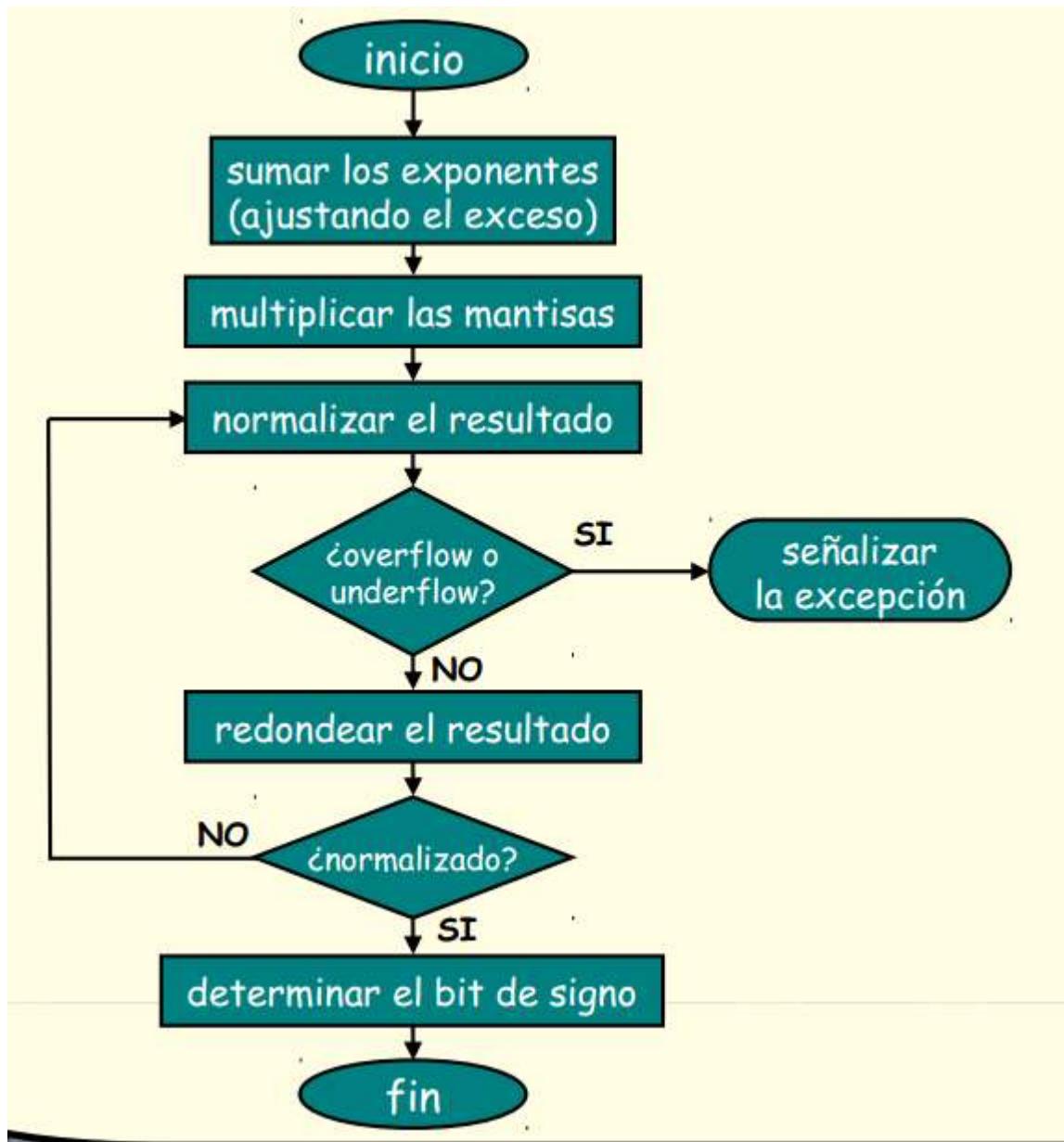
- Alinear los puntos decimales (“desnormalizando” a uno de los operandos de forma tal que ambos exponentes sean igual), para esto desplazar a la derecha la mantisa con el exponente más pequeño.
- Sumar las mantisas.
- Normalizar el resultado, verificando el eventual overflow en el exponente.
- Redondear el resultado al número correcto de dígitos significativos.



### Multiplicación en la norma

Algoritmo básico para la multiplicación de dos números en la norma IEE-754:

- Sumar los exponentes, compensando el exceso
- Multiplicar las mantisas
- Normalizar el resultado y comprobar si se produjo overflow
- Redondear la respuesta al número correcto de dígitos significativos.
- Normalizar nuevamente en caso de ser necesario
- Determinar el signo del resultado



### Redondeos en la norma

La norma IEEE-754 contempla cinco modalidades de redondeo, a saber

- Hacia arriba (hacia +OO)
- Hacia abajo (hacia -OO)
- Hacia el cero (truncado)
- Proximidad unbiased (hacia los pares)
- Proximidad biased (hacia afuera del cero)

Todas estas modalidades de redondeo requieren el uso de bits adicionales de precisión

## Bit de guarda

Denominaremos bits de guarda a los bits de precisión adicionales a la derecha del bit menos significativo. Los bits de guarda son conservados temporalmente hasta que sean requeridos. Usualmente son empleados en la normalización y resultan fundamentales durante el redondeo. La norma IEEE-754 contempla tres bits de guarda, denominados guard (G), round (R) y sticky (S). note se que esto implica que la ALU debe ensancharse en tres bits. Al hacer un uso acertado de los bits de guarda logra obtener el mismo resultado que se obtendría al hacer uso de una precisión infinita. En otras palabras, con solo tres bits de guarda se puede computar el mismo resultado que se obtendría al operar sobre la totalidad de los bits de los operandos.

Por caso, analicemos como a travez de solo dos bits de guarda se puede obtener el redondeo correcto en la siguiente suma con una precisión de dos dígitos decimales:

$$\begin{aligned}
 x_1 + x_2 &= 2.561 \times 10^0 + 2.34 \times 10^2 \\
 &= 0.02561 \times 10^2 + 2.34 \times 10^2 \\
 &= 2.36 \underset{\substack{\swarrow \\ R = 5}}{\text{561}} \times 10^2 \\
 &\quad \underset{\substack{\searrow \\ S \neq 0}}{\text{1}}
 \end{aligned}$$

Contando con estos dígitos de guarda, es posible implementar un correcto redondeo:

- Para  $0 \leq R \leq 4$  se redondea para abajo
- Para  $6 \leq R \leq 9$  se redondea para arriba.
- Para  $R=5$ , se mira el estado de S.

Con respecto al sticky, solo importa si es cero o no (independientemente de cual sea la base). En la norma IEEE-754, S se calcula haciendo OR entre todos los bits descartados (esto es, los que no forman parte de resultado ni tampoco de G ni R). la siguiente tabla resume el rol de los bits R y S en los distintos esquemas de redondeo (donde  $p_0$  es el valor del digito menos significativo):

redondeo	resultado $\geq 0$	resultado $< 1$
hacia $-\infty$		+1 LSB si (R OR S)
hacia $+\infty$	+1 LSB si (R OR S)	
hacia el 0		
proximidad hacia pares	+1 LSB si ((R AND $p_0$ ) OR (R AND S))	+1 LSB si ((R AND $p_0$ ) OR (R AND S))
proximidad afuera del 0	+1 LSB si R	+1 LSB si R

A manera de ejemplo, veamos como se aplican estos esquemas a la hora de redondear a un dígito de precisión los siguientes números:

redondeo	+1.400	-1.600	+1.500	-2.500	+2.502
hacia $-\infty$	+1	-2	+1	-3	+2
hacia $+\infty$	+2	-1	+2	-2	+3
hacia el 0	+1	-1	+1	-2	+2
prox. a pares	+1	-2	+2	-2	+3
prox. fuera 0	+1	-2	+2	-3	+3

→ Cálculo con precisión infinita:

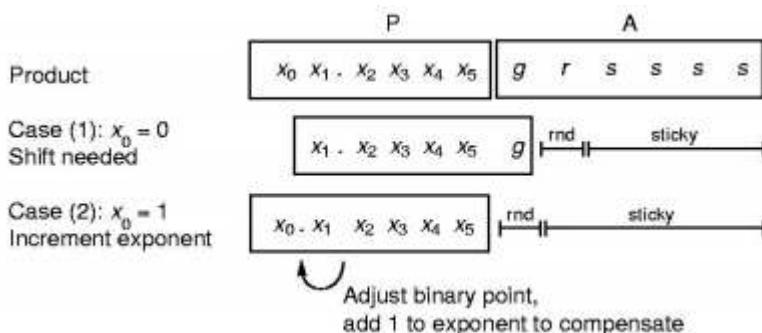
$$\begin{aligned}x_1 &= 1.001000 \times 2^3 \\x_2 &= +1.101001 \times 2^{-1} \\&\quad \downarrow \\&1.001000 \times 2^3 \\&+0.0001101001 \times 2^3 \\&\underline{1.0011101001 \times 2^3} \\&\quad \text{1.001111} \times 2^3\end{aligned}$$

→ Cálculo usando sólo round y sticky:

$$\begin{aligned}x_1 &= 1.001000 \times 2^3 \\x_2 &= +1.101001 \times 2^{-1} \\&\quad \downarrow \\&1.001000 \times 2^3 \\&+0.00011011 \times 2^3 \\&\underline{1.00111011 \times 2^3} \quad \left. \begin{array}{l} R = 1 \\ S = 0 \text{ OR } 0 \text{ OR } 1 = 1 \end{array} \right. \\&\quad \text{1.001111} \times 2^3\end{aligned}$$

Si R y S se usan para redondear, ¿para qué se usa el tercer bit de guarda, el bit G? el bit G se emplea al normalizar el resultado preliminar de las operaciones aritméticas, esto es, antes de aplicar el redondeo. En el caso en que no haga falta usar el bit G, se deben ajustar los valores de los bits R y S de la siguiente manera:  $R_{\text{nuevo}} = G_{\text{anterior}}$ ,  $S_{\text{nuevo}} = R_{\text{anterior}} \text{ OR } S_{\text{anterior}}$ .

En la multiplicación, el rol del bit G se torna aparente cuando el bit más significativo del resultado preliminar es cero



## Implementación en HW

Concretamente, la implementación en hardware de estos algoritmos capitalizará los beneficios de las distintas representaciones ya vistas. Por caso, no tiene sentido contar con un circuito para sumar y otro independiente para la resta cuando se puede sacar provecho de las representaciones complementarias. Notese que si bien la norma establece que la manisa se representa en SM, nada impide a la ALU que cambie internamente la representación de los operandos. A continuación

repasaremos un algoritmo para la suma en punto flotante el cual incorpora la mejora antes comentada:

aborda los siguientes pasos:

- Convertir a binario la parte entera usando cualquiera de los métodos vistos
- Convertir a binario la parte decimal, usando nuevamente cualquiera de los métodos vistos.
- Sea  $X_1 = (e_1, m_1)$  y  $X_2 = (e_2, m_2)$  los dos operandos que se desean sumar.
- Comparar los exponentes. En caso de que  $e_1 < e_2$  intercambiar los operandos, a fin de que la diferencia  $d = e_1 - e_2$  sea positiva.
- Si los signos de  $x_1$  y  $x_2$  difieren, reemplazar la mantisa  $m_2$  por su complemento a dos
- Colocar  $m_2$  en un registro de desplazamiento y correr  $d$  lugares a la derecha. De los bits que se descartan por el extremo derecho armar los bits G, R, y S
- Computar el resultado preliminar  $S = m_1 + m_2$ . Si los signos de los operandos son distintos y no hay carry de salida, reemplazar a S por su complemento a dos.
- Normalizar a S de acuerdo al siguiente detalle: si los signos de los operandos son igual y hay carry de salida, desplazar a S un lugar a la derecha, ingresando el 1 del carry (hubo overflow virtual).
- Caso contrario, desplazar S a la izquierda hasta obtener una mansita normalizada. Notese que el primer desplazamiento hace ingresar a G, pero los restantes deben ingresar ceros.
- En todos los casos, ajustar el exponente en función de la cantidad de desplazamientos realizados
- Ajustar a R y s en caso de ser necesario, es decir, si G no pasó a formar parte de la mantisa en el último paso, hacer  $R=G$  y  $S=R \text{ OR } S$
- Aplicar el redondeo elegido en función de los valores de los bits R y S. en caso de producirse carry out, desplazar la mantisa del resultado a derecha, ajustando el exponente de manera acorde.
- Computar el signo del resultado, si ambos operandos tienen el mismo signo, ese será el signo del resultado. Caso contrario, se debe tener en cuenta cual de los dos operandos era en negativo, si se intercambiaron los operandos en el primer paso, y de si se dos complementó el resultado preliminar.

Las distintas características que deben ser tenidas en cuenta a la hora de determinar el signo del resultado pueden resumirse en la siguiente tabla

<b>¿Hubo swap?</b>	<b>¿Se comp?</b>	<b>signo(<math>x_1</math>)</b>	<b>signo(<math>x_2</math>)</b>	<b>signo(S)</b>
<b>SI</b>		+	-	-
<b>SI</b>		-	+	+
<b>NO</b>	<b>NO</b>	+	-	+
<b>NO</b>	<b>NO</b>	-	+	-
<b>NO</b>	<b>SI</b>	+	-	-
<b>NO</b>	<b>SI</b>	-	+	+

# Diapositiva 9

Sistema Operativo: Es el encargado de gestionar la interacción entre el usuario y la computadora y de administrar sus recursos de manera eficiente. El sistema operativo brinda sus servicios a los distintos programas bajo la forma de llamadas al sistema.

Pila del Programa: la pila del programa (stack) es una de las regiones que integran los programas en ejecución. Su tamaño es dinámico, crece o disminuye según haga falta. Crece hacia abajo desde la dirección más alta. Es utilizada para almacenar datos temporalmente, tales como la dirección de retorno al invocar a un procedimiento o el valor actual de un registro el cual será alterado, por caso durante la invocación a un servicio del sistema operativo.

## Operaciones sobre la pila del programa

Instrucción push: resta el tamaño en bytes del objeto apilado al valor actual del registro ESP (stack pointer) y lo almacena en el lugar ahora apuntado por ESP.

Instrucción pop: recupera el objeto a ser desapilado de la locación apuntada por el registro ESP y lo actualiza sumándole el tamaño en bytes de ese objeto. Nótese que el objeto sigue en memoria, pero será sobrescrito cuando el tamaño de la pila vuelva a crecer.

## Caller sabe vs Callee sabe

A la hora de salvar el contenido de los registros que serán alterados durante la invocación a un procedimiento surgen dos alternativas:

- Una posibilidad (denominada “caller save”) es que quien llama al procedimiento se encargue de salvar en la pila del programa aquellos registros que se necesiten preservar. Esta opción permite optimizar múltiples llamados a un procedimiento, evitando que cada una de las invocaciones preserve y recupere de la pila múltiples registros.
- Otra posibilidad (denominada “callee save”) es que el procedimiento que es llamado se encargue de preservar en la pila del programa sólo aquellos registros que serán modificados. Callee sabe posibilita que el procedimiento sólo preserve los registros que serán afectados (el llamador usualmente no tiene acceso a esta información).

## Pasaje de parámetros

Los procedimientos en ocasiones requieren recibir información adicional al ser invocados, esto es, requieren un conjunto de parámetros. El pasaje de parámetros se puede resolver de diversas formas:

- Usando registros: Este es por lejos el mecanismo más eficiente (pues no implica accesos adicionales a memoria). La cantidad de parámetros está limitada a la cantidad de registros disponibles.
- Usando la memoria: Este es el mecanismo más flexible que en anterior, pues permite una cantidad mucho mayor de parámetros. Naturalmente, tiene como desventaja que implica accesos adicionales a memoria.

- Usando la pila del programa: El pasaje de parámetros usando registros o usando la memoria no resulta del todo apropiado a la hora de implementar procedimientos recursivos. En contraste, el pasaje de parámetros usando la pila del programa permite la implementación de procedimientos recursivos con relativa simpleza. Nótese que la dirección de retorno seguirá ocupando el tope de la pila por debajo (lo que implica que la pila no será accedida como pila).

## **Reentrancia**

Usando la pila obtenemos lo que se denomina un procedimiento reentrant o puro. Para alcanzar la reentrancia hace falta que el código no sea automodificable y que solo se opere sobre los registros o la pila del sistema. Para apreciar las ventajas de este tipo de código hay que recordar que los sistemas en la actualidad son multiprogramados, es decir, existe más de un programa en ejecución a la vez. Bajo la definición, la recursión consiste meramente en sacar provecho de la reentrancia de una rutina.

## **Ejecución de un programa**

Un programa atraviesa distintas etapas desde su concepción hasta llegar a estar corriendo en una arquitectura. En primer lugar hay que escribir el código fuente. Más tarde hay que convertir ese código fuente en código objeto (instrucción por instrucción). Luego, se debe convertir el código objeto (de uno o más archivos fuentes) en el código ejecutable. Finalmente, se debe cargar en memoria el código ejecutable y comenzar su ejecución.

## **Etapa de Compilación**

Durante esta etapa se resuelven la totalidad de las referencias locales, se identifican y dejan pendientes de resolver a la totalidad de las referencias externas. El código objeto es prácticamente ejecutable, sólo resta incorporar el código objeto asociado a otros módulos o bien a las funciones de librería que se hayan utilizado.

## **1-pasada vs 2-pasadas**

En relación a la resolución de las referencias locales, es interesante tener en cuenta cómo se lleva adelante esa resolución. El compilador cuenta con una estructura de datos interna denominada tabla de símbolos, donde anota la dirección que asocia a las distintas declaraciones de etiquetas que va encontrando durante el ensamblado. Naturalmente, la tabla de símbolos sólo contiene la dirección de las etiquetas cuyas declaraciones ya visitó. Es decir, en principio no es posible resolver las referencias a etiquetas aun no declaradas (por caso, saltos hacia adelante). Esto implica que el ensamblador deba realizar dos pasadas sobre el código fuente.

Pero es factible resolver las referencias en sólo una pasada, esto radica en tener presente que tanto el programa fuente como la tabla de símbolos coexisten en el mismo espacio de direccionamiento. Al encontrar una referencia a una etiqueta aún no declarada, se coloca en la tabla de símbolos una referencia a esa locación (en la cual posteriormente se deberá actualizar la dirección de la etiqueta). De aparecer otras referencias a esa misma etiqueta, se van enlazando las locaciones donde eventualmente se deberá poner la dirección correcta. Cuando eventualmente se alcance la declaración de esa etiqueta, se recorre la “lista enlazada” arrancando en la dirección

almacenada en la tabla de símbolos reemplazando las referencias con la dirección recién descubierta. Caso contrario, de alcanzar el fin del archivo fuente simplemente se señala el error de compilación. Cabe acotar que esta idea sólo es aplicable si en el espacio asignado a la etiqueta dentro de la instrucción cabe una dirección completa.

### **Etapa de vinculación**

Durante esta etapa el código objeto de uno o más archivos fuentes es convertido en código ejecutable. La principal tarea del vinculador es resolver la totalidad de las referencias externas. Las referencias a otros módulos se resuelven a partir de código objeto de ese módulo (el cual debe suministrarse en ese momento). Las referencias a librerías se resuelven usando el código objeto de la librería, el cual estará disponible en alguna ubicación conocida de antemano.

### **Etapa de carga**

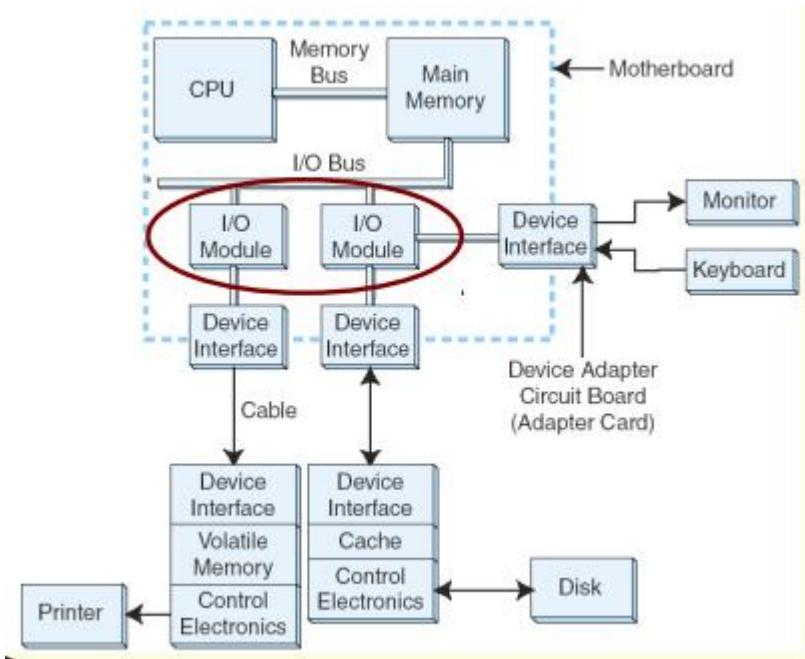
El cargador, un componente del sistema operativo, es el encargado de cargar a memoria el código ejecutable de un cierto programa. A esta altura todas las referencias externas están resueltas, sólo resta conocer la dirección inicial a partir de la cual será cargado el código ejecutable. Los modelos más avanzados de manejo de memoria han simplificado notablemente esta tarea (por caso, al contar con un espacio de direccionamiento de uso exclusivo para cada proceso).

### **Vinculación Dinámica**

Como es frecuente que múltiples programas comparten la misma librería (por caso, *stdio*), en la actualidad es posible vincular múltiples programas a una misma instancia en memoria de la librería. Desde ya, esta librería debe estar compuesta exclusivamente de procedimientos reentrantes. Nótese que esas referencias externas no podrán ser resueltas en tiempo de vinculación, razón por la cual es este tipo de librería se las denomina de vinculación dinámica (por caso, las librerías .DLL).

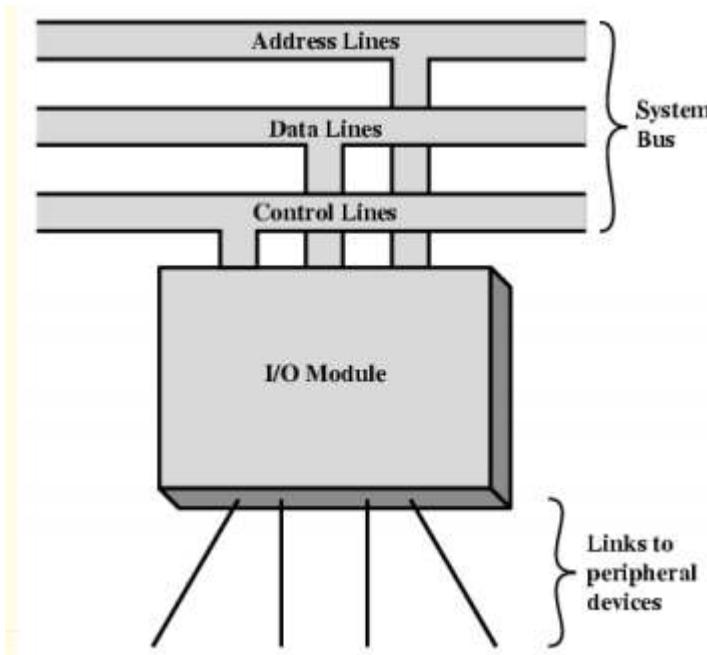
## **Diapositiva 10**

### **Dispositivos de E/S**



Cada módulo de E/S se conecta al bus del sistema y controla uno o más dispositivos. Los módulos de E/S no son solo un conjunto de pines y conectores, contienen en realidad una cierta “inteligencia”, como para poder comunicar al dispositivo con la computadora. Si tenemos en cuenta que existe una amplia variedad de dispositivos, cada uno con su método particular de interacción, no es nada práctico tener que incorporar toda esa lógica en el propio procesador. Más aun, como la tasa de transferencia de datos de los dispositivos es bastante menor que la de acceso a la memoria o la de procesamiento del propio procesador, tampoco resulta práctico hacer uso del veloz bus del sistema para comunicarse de manera directa con los dispositivos. Por otra parte, los dispositivos pueden emplear formatos de datos y/o longitudes de palabra diferentes a los adoptados por la computadora. Todas estas razones justifican la existencia de los módulos de E/S.

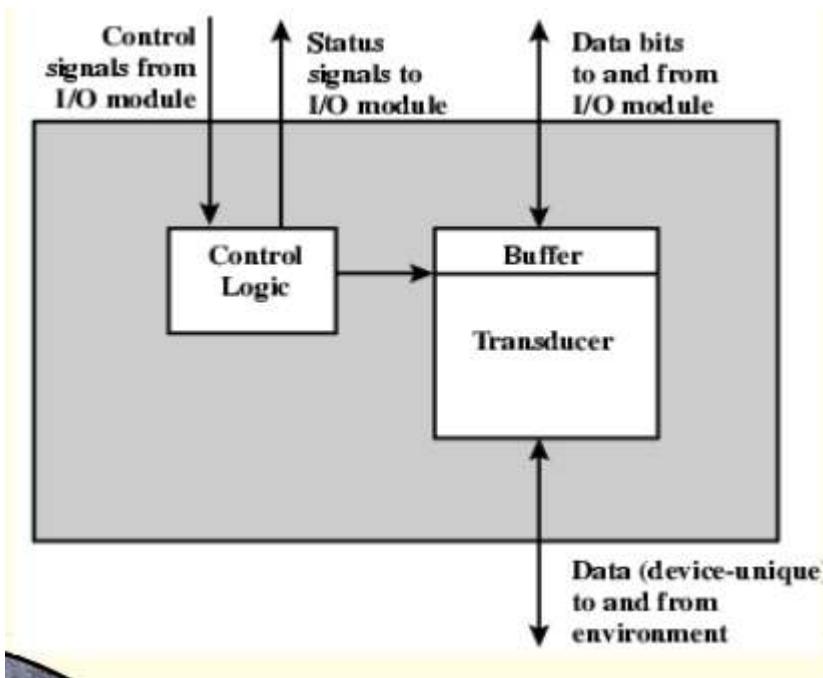
Un módulo de E/S es una interfaz que brinda acceso al CPU y/o a la memoria principal, a partir de la utilización del bus del sistema, a partir de la utilización de un conmutador central. Un mismo módulo de E/S puede servir de interfaz a uno o más dispositivos



Los dispositivos externos se pueden clasificar de la siguiente manera:

- De interaccion con humanos: posibilitan la comunicacion con el usuario de la computadora (monitor, teclado, etc)
- De interaccion con maquinas: posibilitan la comunicacion con otros componentes de la computadora (disco rigido, lectoras CD/DVD, etc)
- De comunicacion: posibilitan la comunicacion con dispositivos remotos (placa de red)

#### Estructura de un modulo



Un odulo genérico esta constituido por los siguientes componentes:

- La señal de control, las cuales determinan la acción que llevará a cabo el dispositivo (por caso, enviar o recibir información, etc)
- Los bits de datos, esto es, el conjunto de bits a ser enviados/recibidos hacia/desde el modulo de E/S
- Las señales de estado, las cuales indican el estado del dispositivo (por caso, READY o BUSY).
- La lógica de control asociada al dispositivo, la cual controla la operación del dispositivo en respuesta a la directiva del modulo E/S
- El transductor, el cual convierte los datos binarios en formato eléctrico a otros formatos durante la salida y de los otros formatos a la forma binaria eléctrica durante la entrada
- El buffer asociado al transductor, el cual mantiene de manera temporaria los datos que se están enviado/recibiendo hasta que sean procesados

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

## Responsabilidades

Las principales funciones y tareas de la cual todo modulod de E/S debe hacerse cargo son las siguientes:

- control y temporizado
- comunicación con el procesador
- comunicación con los dispositivos
- almacenamiento temporal de datos (buffering)
- detección de errores

un modulo de E/S puede resultar transparente o no al CPU, en función de si oculta o bien revela las características de los dispositivos a los cuales brinda acceso. De manera análoga, un modulo de E/S puede hacerse cargo del control del funcionamiento del dispositivo, o bien delegar esa responsabilidad en el propio CPU. Al modulo de E/S también a veces se lo conoce como canal de E/S o procesador de E/S

## Operación prototípica

La tranferencia de datos desde un dispositivo externo podría involucrar los siguientes pasos:

- el procesador interroga al modulo de E/S para comprobar el estado del dispositivo
- el modulo de E/S presenta su estado
- si el dispositivo estaa preparado para trannmitir, el procesador solicita la trasnferencia de datos

- el modulo de E/S obtien los datos del dispositivo
- los datos son puestos por el modulo de E/S a disposición del procesador

### E/S programada

Si algo se le puede criticar a esta operación es que CPU parece quedar de alguna manera atado al dispositivo de E/S. si la velocidad del dispositivo es inferior a la del procesador, estaremos desperdiizando capacidad de computo, al quedar a la espera de que el dispositivo termine de resolver la ultima solicitud. Este tipo de implementación se la denomina E/S programada, ya que el procesador ejecuta un programa que controla directamente al modulo de E/S (y por ende al dispositivo)

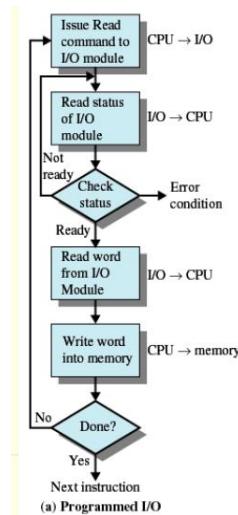
Bajo la modalidad de entrada/salida el procesador realiza las siguientes tareas:

- sensa el estado del modulo
- envía los comandos de lectura/escritura
- transfiere desde/hacia el modulo los datos

como se puede apreciar, el procesador debe esperar a que el modulo y el dispositivo completen el ultimo requerimiento. Se dice que el procesador realiza un “busy waiting”, es decir, un “espera ocupada”

análisis en detalle de la implementación de la operación “busy waiting”

- el CPU solicita un operación de E/S
- el modulo de E/S efectua la operación
- El modulo de E/S activa los bits de estado de manera acorde
- El CPU monitorea periódicamente los bits de estado
- El modulo de E/S no informa al CPU de forma directa (no lo interrumpe)



### Comandos de E/S

El CPU para enviar un comando de E/S a un cierto modulo debe suministrar su dirección, esta dirección identifica únicamente al modulo de E/S (y al dispositivo en caso de haber mas de uno). Una vez identificado, el CPU envía el comando. Puede ser de control, indicándole al modulo que hacer (por caso, encender el laser de una grabadora), o bien una consulta acerca del estado (por caso, se produjo alguno error al realizar la ultima orden), o sino, una orden de lectura/escritura, instruyendo al modulo que transfiera los datos del buffer al dispositivo

## **Direccionamiento del dispositivo**

Bajo una E/S programada la transferencia de datos es similar a los accesos a memoria: las instrucciones de E/S que el procesador trae de memoria durante su ciclo básico de ejecución se van mapeando en comandos de E/S. Cada dispositivo cuenta con un identificador único. Las instrucciones del CPU contienen ese identificador como argumento (el cual es esencial una dirección de memoria).

A la hora de direccionar a un dispositivo de E/S, se han de ensayar principalmente dos variantes:

Una posibilidad es hacer uso de una E/S mapeada en memoria principal, es decir, que los dispositivos y la memoria comparten el mismo espacio de direccionamiento. La E/S es indistinguible de un acceso convencional a memoria, como ventaja esto implica que no hace falta contar con instrucciones especiales para E/S.

La segunda alternativa consiste en aislar el espacio de direccionamiento para E/S. En otras palabras, el espacio de direccionamiento de la memoria principal es independiente del espacio de direccionamiento de los dispositivos de E/S. Nota que esto implica contar con líneas adicionales (es decir, más pines en el CPU) para poder hacer referencia a los objetos dentro de este nuevo espacio de direccionamiento. Esto implica que hace falta contar con instrucciones específicas para E/S, distintas de las convencionales.

## **Concepto de interrupción**

La interrupción es un mecanismo que permite indicarle al procesador que se produjo un evento que requiere atención inmediata. Para poder atender este evento es necesario interrumpir la ejecución del programa en curso. El estado del programa en ejecución debe ser guardado para poder ser reanudado posteriormente. Al producirse una interrupción se ejecuta usualmente un pequeño programa de propósito específico a la interrupción actual denominado manejador de la interrupción (interrupt handler).

Las interrupciones fueron concebidas para atender principalmente excepciones aritméticas y señalizar eventos de tiempo real. Con el paso del tiempo, la alta flexibilidad que brinda este mecanismo permitió que se lo aplique a otras situaciones, tales como: comunicación con dispositivos E/S, invocación a servicios del sistema operativo, resolución de los page faults.

Las interrupciones se pueden clasificar de acuerdo a las siguientes dimensiones:

- Síncronicas vs asíncronas.
- Solicitadas por el usuario vs forzadas
- Enmascarables vs no enmascarables
- Entre instrucciones vs intra-instrucciones
- Recuperables vs insalvables

Pasos para llevar adelante una atención de interrupción generada por un dispositivo de E/S

- 1- Un cierto dispositivo señala una interrupción
- 2- El procesador finaliza la ejecución de la instrucción en curso (esto hace más fácil guardar el contexto)

- 3- El procesador antes de ejecutar la próxima instrucción revisa si se produjo una interrupción: al detectar que hubo una, envía una señal de ACK al dispositivo para informarle que ya se puede bajar la señal de interrupción.
- 4- El procesador se prepara para transferir el control al manejador; para esto guarda en la pila del sistema la palabra de estado del procesador y la dirección de la próxima instrucción a ser ejecutada
- 5- Apelando al mecanismo del ual se disponga en la arquitectura, se determina la dirección de comienzo de la rutina que manejará la interrupción
- 6- Esta rutina suele comenzar por guardar en la pila del sistema los distintos registros que van a ser modificados durante la atención de la interrupción
- 7- El manejador lleva adelante todas las acciones que requiera para completar la atención de la interrupción en curso
- 8- Al terminar la atención de la interrupción, se produce a restaurar los valores de los registros que fueran oportunamente guardados en la pila del sistema
- 9- Finalmente, se restaura la palabra del procesador y la dirección de la próxima instrucción a ser ejecutada (es decir, continua la ejecución del programa interrumpido al principio)

### **E/S con interrupciones**

La E/S con interrupciones fue concebida como una alternativa superadora a la E/S programada. La clave de este tipo de E/S consiste en evitar que el CPU entre en el “busy waiting”. A tal efecto, la idea es que el CPU envíe un comando de E/S para inmediatamente dedicarse a atender otros asuntos, y recién cuando el comando de E/S finalice su ejecución se interrumpirá al CPU para anotarlo de la novedad. La operación básica de una E/S con interrupciones abarca las siguientes etapas:

- El CPU envía un comando, por caso un READ
- El módulo de E/S procede a leer los datos desde el dispositivo mientras el CPU atiende otros asuntos.
- Una vez finalizada la operación, el módulo de E/S interrumpe al CPU
- El CPU solicita los datos obtenidos del dispositivo
- El módulo de E/S finalmente pone a disposición el procesador estos datos

La E/S con interrupciones constituye sin duda una mejora sobre la E/S programada. El CPU no está más “atado” al dispositivo de E/S, ya que entre el envío de un comando y la recepción de los datos queda libre para atender a otros procesos. No obstante, el CPU sigue interviniendo en las transferencia de la totalidad de los bytes enviados/recibidos. Más aun, cada vez que se interrumpe al CPU se debe llevar adelante un costoso cambio de contexto

### **Quién interrumpió?**

Para determinar qué módulo interrumpió al CPU se puede hacer uso de múltiples señales de interrupción, una para cada dispositivo. Claro está, la cantidad de señales que entran y salen de CPU están muy acotadas, por ende, esta técnica solo permite usar una cantidad limitada de dispositivos.

Otra posibilidad es hacer uso de polling (interrogación), es decir que el CPU consulte a los módulos de E/S en undeterminado orden hasta dar con el que produjo la interrupción.

Una alternativa mas eficiente es hacer uso de la técnica de Daisy chain (“corona de margaritas”, o también “conexión encadenada”). Los modulos de E/S comparten una línea común para solicitar interrupciones. La línea de reconocimiento (ACK) se conecta encadenando los modulos. El modulo que interrumpe debe poner su vector en el bus del sistema. El CPU usa este vector para identificar al dispositivo que produjo la interrupción

Finalmente otra posibilidad para identificar al modulo de E/S que produjo la interrupción es hacer uso de la técnica de bus master (arbitraje del bus). El modulo de E/S que desea interrumpir al CPU debe primero reclamar como suyo al bus antes de poder levantar la señal de interrupción. Esta política es utilizada de manera satisfactoria por la tecnología para bus de sistema denominado PCI y su evolución PCI-Epress

### **Interrupciones Simultaneas**

La otra cuestión que resta resolver es que hacer cuando se producen mas de una interrupción en simultaneo. Existen dos alternativas, a saber: se puede resolver la aparición de una interrupción en simultaneo apelando a prioridades a nivel de hardware o de software o alternativamente, se puede impedir que se produzca la interrupción en simultaneo

Un ejemplo de la primera política es que cada línea de interrupción tenga asociada un prioridad. La idea es que las líneas de mayor prioridad puedan interrumpir a las de menor prioridad. En caso de usar polling, la interrupción en simultáneo queda resulta a partir del ordenamiento impuesto por el indagado, es decir, se resuelve via software.

Otra posibilidad es usar Daisy-chaining, resolviendo la interrupción en simultaneo a partir del encadenado de los dispositivos, es decir, se resuelve via hardware.

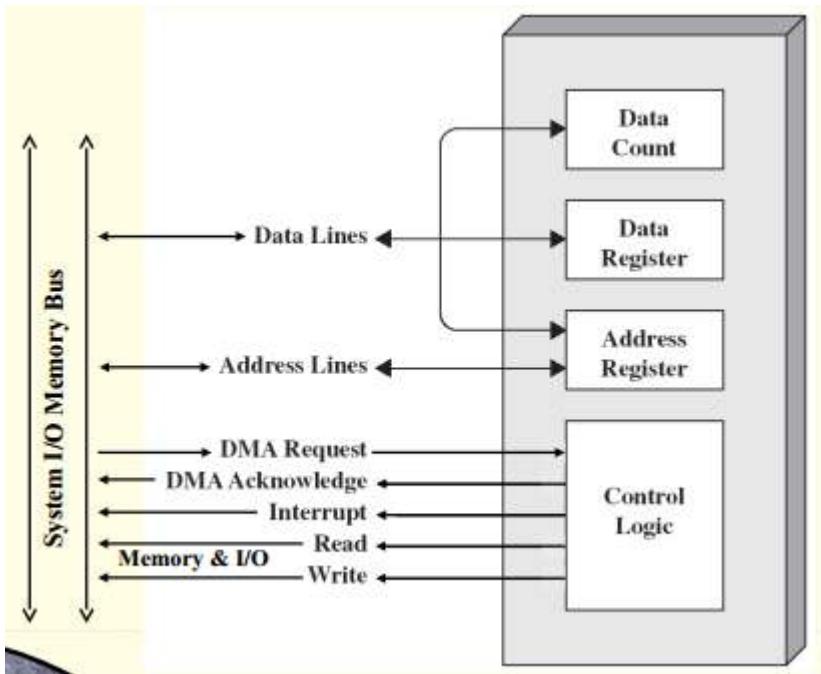
Finalmente, un ejemplo de la segunda política es usar la técnica de bus master, donde únicamente el modulo que se adueñe de bus podrá interrumpir. En otras palabras, es imposible que se produzcan interrupciones en simultaneo.

### **E/S con DMA**

La E/S con interrupciones parece mejorar el desempeño por sobre la E/S programada, si bien también se paga un costo oculto:

- La E/S programada alcanza una levada tasa de transferencia a instancias de mantener autómatico al CPU, el cual no podrá atender otros requerimientos
- La E/S con interrupciones libera al procesador de ese uso exclusivo, pues cada byte leído tiene que pasar por las manos del CPU, lo que a su vez requiere de un costoso cambio de contexto.

Para capitalizar lo mejor de los dos escenarios se concibió la técnica de E/S con DMA (Direct Memory Access). Esta técnica consiste en agregar un modulo DMA al bus del sistema el cual asume la responsabilidad de gestionar la E/S. este modulo debe usar el bus solo cuando el CPU no lo esté usando, o bien debe forzar al CPU a que difiera temporalmente su uso. Esta ultima variante es la mas frecuente y se la denomina cycle stealing (robo de ciclos).



### Operación del módulo DMA

El CPU informa al módulo DMA: tipo de operación (READ/WRITE), dirección del dispositivo, dirección de memoria de comienzo y cantidad de información a trasferir. Luego, el CPU trabaja en otra cosa, el módulo DMA se encarga de la transferencia (un Word a la vez) desde o hacia memoria. Por último, el módulo DMA interrumpe al CPU.

### Robo de ciclos (cycle stealing)

Para transferir un dato, el modulo DMA debe tomar el bus por un ciclo. Transfiere de a una palabra por vez. Transfiere de a una palabra por vez. No se trata de un interrupción: el cpu no hace cambio de contexto y el CPU se suspende justo antes de acceder al bus. Si bien esto ralentiza al CPU, no lo es tanto como cuando esta a cargo de la E/S (esto es, bajo la E/S programada o con interrupciones)

### Canales de E/S

Los dispositivos de E/S son cada vez mas sofisticados y requieren mayores tasas de transferencia. La E/S con DMA parece ser la solución mas adecuada, si bien el procesador todavía dese ser interrumpido una vez por cada bloque de datos transferido (al finalizar la misma). Para poder mejorar el desempeño de la E/S con DMA solo resta hacer uso de procesadores de E/S de propósito específico.

Los canales de E/S cuentan con la capacidad de ejecutar instrucciones de E/S. también se los conoce como procesadores de E/S. los canales de E/S cuentan con un control completo sobre las operaciones de E/S (por ejemplo, controladoras de video 3D). La intención es que el CPU instruya al canal de E/S para hacer una transferencia y el canal de E/S realice la totalidad de la misma.

### Almacenamiento rotativo

El almacenamiento rotatorio ha sido desde los arboles de la computación la forma de memoria secundaria por excelencia. Es lo conocemos vulgarmente como disco rígido, o también como HDD (Hard Disk Drive). Está compuesto por uno o mas superficies circulares denominados platos los cuales giran a alta velocidad. Cuentan con múltiples cabezas de lecto-escritura, usualmente una para cada cara de los platos. Información se almacena en pistas circulares concéntricas.

### **Discos rígidos**

El disco rígido es un medio de almacenamiento de información electro-magnético. Los platos están recubiertos de fino sustrato ferromagnético, al estilo del utilizado en los cassettes de audio y los discos flexibles. El cabezal de lecto-escritura magnetiza el sustrato en distintas direcciones en función de si se está almacenando un cero o un uno. Para leer la información almacenada se produce en sentido inverso, el cabezal va detectando el cambio en la dirección de la magnetización

Parámetros a tener en cuenta al elegir un disco rígido:

- Costo: el costo de un disco rígido resulta directamente proporcional a los dos siguientes: evidentemente un rígido de más tamaño tendrá un mayor costo, lo mismo uno de mejor desempeño.
- Capacidad: con respecto a esto existen varios factores
  - o Tamaño de los platos: los platos vienen en múltiples tamaños, desde 8", pasando por 5½" y 3½", hasta los más pequeños, aptos para notebooks, de 2½".
  - o Cantidad de platos: se puede incrementar la capacidad de un disco rígido incorporando platos adicionales, lo cual incurrirá en un costo adicional
  - o Densidad de la información: al disminuir la distancia entre los bits almacenados es posible aumentar la cantidad de información. Con el objeto de incrementar la densidad de la información es necesario que el cabezal esté lo más cerca posible del plato. El disco rígido saca provecho de la corriente de aire que genera la rotación del plato, dejando que la cabeza flote a escasos 3 nanómetros del plato
- Finalmente, también hay otros aspectos para tener en cuenta en relación al desempeño:
  - o Tiempo promedio de búsqueda (seek time): es el tiempo que le toma al disco posicionar el cabezal en la pista conteniendo la información buscada.
  - o Tiempo promedio de latencia rotacional: es el tiempo que le toma al disco desde que el cabezal se posiciona en la pista correcta hasta que el dato deseado pasa por debajo del mismo
  - o Tiempo de transferencia: es el tiempo que toma la transferencia de la información propiamente dicha.

El tiempo promedio de búsqueda se ve afectado por los siguientes factores:

- Tecnología del disco: distintos factores tecnológicos como el peso del cabezal o la potencia del motor que mueve al brazo actuador también afectan el tiempo promedio de búsqueda
- Cantidad de platos. A menor cantidad de platos, menos cabezales y por ende el brazo actuador resulta más liviano y más ágil
- Tamaño de los platos: a menor tamaño de plato el cabezal debe recorrer menor distancia

El tiempo promedio de latencia rotacional se ve afectado por los siguientes factores:

- Velocidad de rotación de los platos: cuanto mas rápido gire el plato mas rápido llegara la información a posicionarse bajo el cabezal de lecto/escritura.
- Organización de la información en cada pista: la técnica de interleaving posibilita optimizar el acceso secuencial a la información contenida en una pista de forma que una vez concluida la lectura de una parte la siguiente justo se posicione debajo del cabezal

El tiempo de transferencia de la información depende de los siguientes factores:

- Velocidad de rotación de los platos: a mayor velocidad de rotación, mas rápidamente pasa la información por debajo del cabezal
- Densidad de la información: si la información esta almacenada de manera mas compacta, a una misma velocidad de rotación se logra mejorar la tasa de transferencia

En síntesis, el tiempo de acceso proedio es  $T_{access} = T_{seek} + T_{latency} + T_{transfer}$ . Donde  $T_{seek}$  mide el tiempo de le toma al cabezal para recorrer la mitad de las pistas.  $T_{latency}$  mide el tiempo que le toma a un plato dar media vuelta.  $T_{transfer}$  mide el tiempo que tarda el cabezal en leer la cantidad mínima de información (un sector)

Los discos rígidos apelan a otras técnicas adicionales para mejorar su desempeño promedio. Hacen uso de almacenamiento intermedio tipo memoria cache, posibilitando cancelar en parte el impacto del retardo de búsqueda y de la latencia rotacional, a mayor cantidad de cache, mejor desempeño promedio. También suelen implementar alguna técnica de preposiciónamiento, con el objeto de anticiparse al próximo requerimiento por parte del procesador

### **Organización Física**

La organización física de la información en un disco rígido responde a su estructura. La información se almacena en un conjunto de pistas, ubicadas en las distintas caras de los platos. Cada pista contiene una cantidad variable de sectores, con las pistas exteriores almacenando más sectores que las interiores. Cada sector solía almacenar exactamente 512 bytes, si bien en la actualidad se está migrando hacia un tamaño de sector mayor (4096 bytes).

### **Organización lógica**

La organización lógica de la información estandariza el acceso al contenido. La técnica de LBA (Logical Block Addressing) permite acceder al conjunto de sectores que pertenecen a un cierto disco rígido sin conocer su geometría (esto es, su configuración en platos, pista, y sectores). El conjunto de sectores se dirige como si se tratara de un arreglo, dejando que la electrónica interna al disco se encargue de traducir estas referencias lineales en las correspondientes referencias tridimensionales

### **Discos de estado sólido**

Recientemente el nivel de integración de las memorias de tipo flash ha permitido que estas almacenen en capacidad a los discos rígidos. En consecuencia, comenzaron a aparecer en el mercado los denominados discos en estado sólido (SSD). Un SSD se asemeja externamente a un

disco rígido convencional, pero internamente se asemeja mas a un pendrive o incluso a una memoria RAM persistente

Ventajas:

- Al no contar con partes móviles, se elimina el tiempo de búsqueda y la latencia rotacional, redundando en una mejora notable del tiempo de acceso promedio.
- Por esta misma razón recuden significativamente el consumo de electricidad, lo que implica un mayor tiempo de vida de la batería en las portátiles
- Generan mas calor dentro de la computadora
- Toleran mucho mejor los golpes, ya que no cuenta con multiples cabezales flotando en proximidad a la superficie de uno o mas platos
- Resultan inmunes a la fragmentación, un problema que acucia a la mayoría de los sistemas de archivos
- Son absolutamente silenciosos. Es increíble el nivel de contaminación sonora producto de los discos rígidos con el que convivimos a diario
- La ausencia de partes móviles también los torna inmunes a las fallas mecánicas, es decir, este tipo de almacenamiento parecería ser mas confiable
- La información almacenada no se altera en presencia de campos magnéticos

Desventajas:

- El costo por gigabyte es superior (alrededor de seis veces más caro)
- En consecuencia, la capacidad de estos discos suele ser mas escasa que en los discos convencionales
- La escritura son en bloques, es decir, no se puede alterar solo un bit o solo un byte, por lo que resultan mas lentas las lecturas
- Dado que un bloque de datos puede ser reescrito un cantidad acotada de veces (de 1.000 a 100.000 veces), luego de lo cual se inutiliza el bloque

## RAID

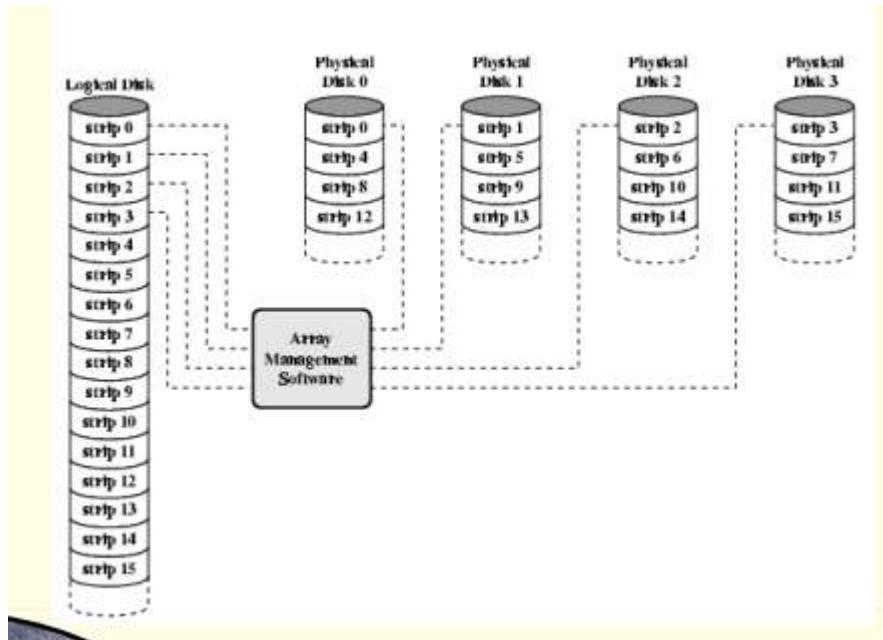
Como vimos la mejora en los mecanismos de E/S no siempre se ve acompañada de una mejora en la tecnología de los dispositivos. En este sentido, una técnica que permite mejorar el desempeño de los dispositivos de almacenamiento es la técnica de RAID (Redundant array of independent Disk). La clave radica en combinar multiples discos haciendo uso de adecuados canales de E/S con el objeto de mejorar los parámetros de desempeño

Existen siete variantes de esta técnica, cada una con sus propias características:

- Raid 0 (striping de bloques sin paridad)
- Raid 1 (mirroring)
- Raid 2 (striping de bits con paridad)
- Raid 3 (striping de bytes con paridad)
- Raid 4 (striping de bloques con paridad dedicada)
- Raid 5 (striping de bloques con paridad distribuida)
- Raid 6 (striping de bloques con doble paridad)

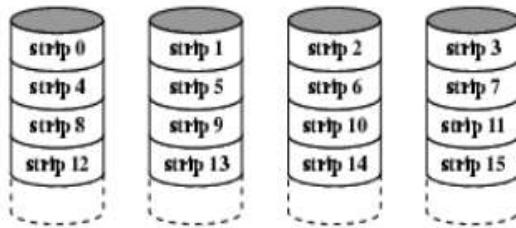
Todos los esquemas de RAID comparten un conjunto de características en común: las unidades físicas son consideradas por el sistema operativo como una única unidad lógica. Los datos se distribuyen a través de las distintas unidades físicas. En los discos redundantes se almacena información de paridad la cual posibilita la recuperación de datos ante alguna falla.

**RAID 0:** No maneja redundancia (se puede argumentar que no califica como un esquema raid). Los datos son organizados en tiras de datos (stripes) a través de los discos disponibles (stripe set). Este esquema disminuye el tiempo de acceso promedio, pues ante cada acceso se está aprovechando la tasa de transferencia de los distintos canales disponibles. Permite usar la totalidad del espacio disponible inicialmente.

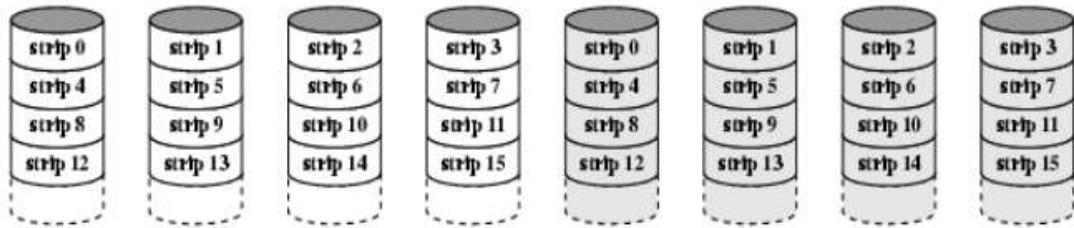


**RAID 1:** la idea es que múltiples discos contengan una imagen especular de los datos almacenados. Los datos deben ser escritos en todos los discos, si bien las lecturas se pueden resolver a partir de cualquiera de ellos. La recuperación ante fallos es simple, si una unidad falla se puede continuar trabajando con las restantes. Para n discos, permite usar a lo sumo 1/n del espacio disponible inicialmente.

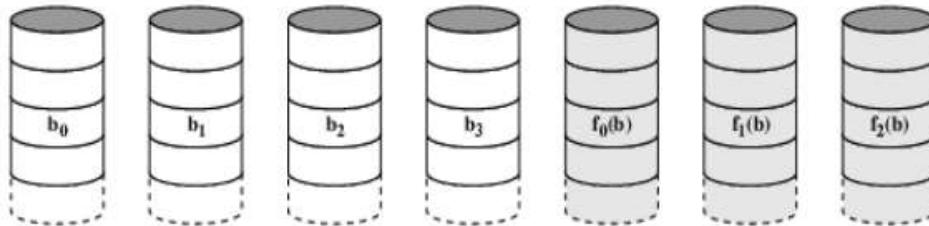
**RAID 2:** los discos deben operar sincronizados (esto es, con sus cabezales de lecto-escritura en la misma pista). Los datos se almacenan en cadenas muy cortas (incluso de a un byte o de a una palabra). Se hace uso de varios discos de paridad, los cuales almacenan un código hamming de los datos. Es el único esquema que actualmente no se utiliza. Permite alcanzar altas tasas de transferencia, análogas a las observadas bajo RAID 0



(a) RAID 0 (non-redundant)



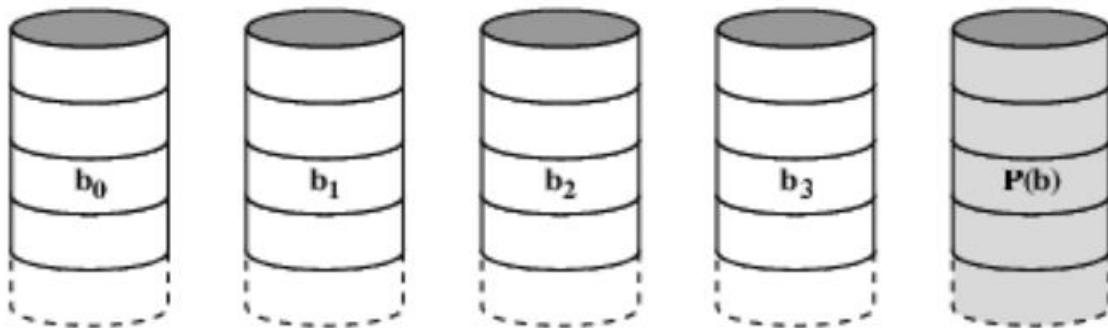
(b) RAID 1 (mirrored)



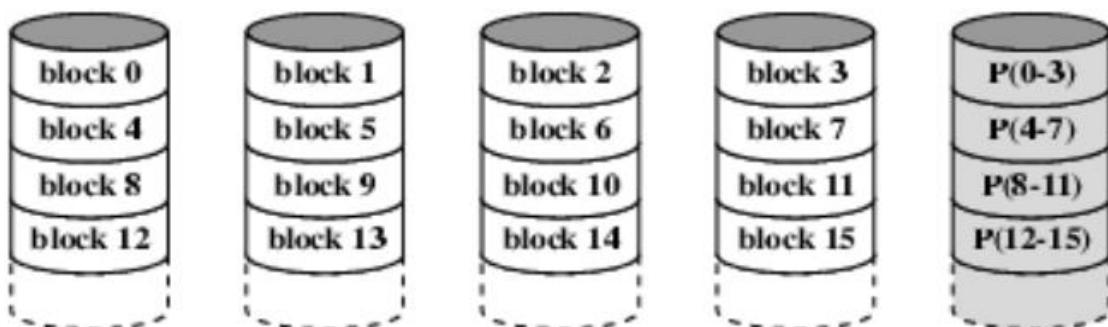
(c) RAID 2 (redundancy through Hamming code)

**RAID 3:** en líneas generales es análogo a raid 2. Utiliza un único disco redundante. En lugar de usar haming, utiliza bits de paridad calculados sobre los bits que ocupan la misma posición de todos los discos. En caso de fallar algún disco, se accede a la información flarante usando los bits de paridad. No permite resolcer pedidos simultaneos, pues cada operación requiere actividad en todos los discos

**RAID 4:** permite el acceso independiente a los discos. Las particiones E/S se atienden en paralelo. Hace uso de tiras de mayor tamaño. Se calcula la paridad a partir de los bits de una tira en cada disco y se almacena en la correspondiente tira de disco de paridad. Una escritura de na tira de datos implica dos lecturas (tira de datos y paridad anterior) y dos escrituras (tira de datos y paridad nueva)



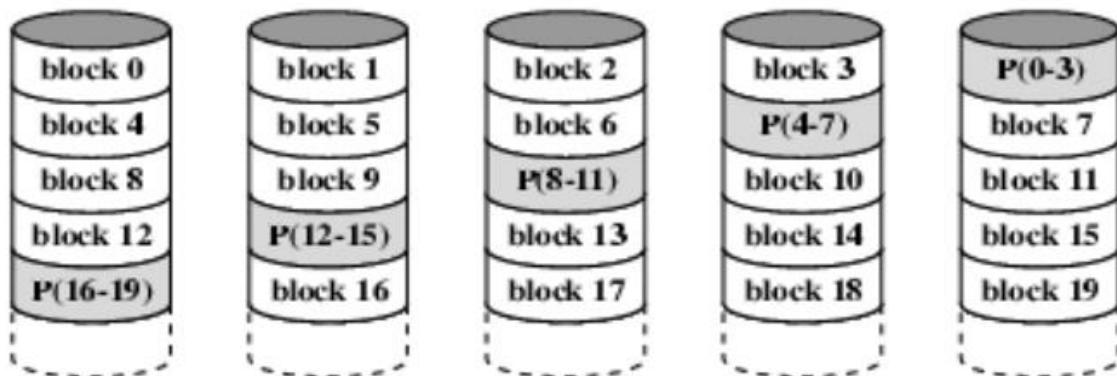
(d) RAID 3 (bit-interleaved parity)



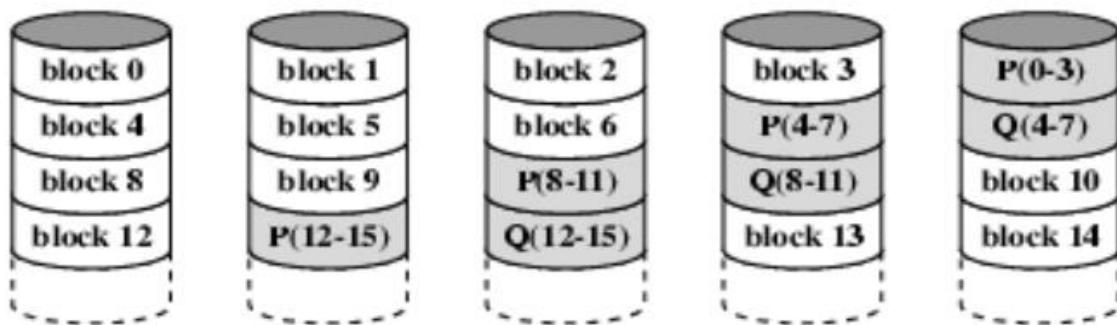
(e) RAID 4 (block-level parity)

**RAID 5:** análogo a RAID 4, excepto que los bits de paridad se distribuyen en todos los discos disponibles (es decir, no usa un disco exclusivo para paridad). Evita el cuello de botella que representa el disco contenido la paridad bajo RAID 4. En la actualidad es el esquema RAID más popular (tanto a nivel de software como de hardware). Al igual que bajo RAID 0, se puede hacer uso de discos de distintos tamaños.

**RAID 6:** análogo a RAID 5. Utiliza dos bits de paridad, calculados por funciones distintas, almacenados en discos distintos (es decir, extiende a RAID 5 agregando otro bit de paridad). Por ende, requiere de dos discos redundantes. Como particularidad es el único esquema que permite la recuperación de los datos aun cuando se produzca un segundo fallo durante el proceso de recuperación ante un fallo anterior.



(f) RAID 5 (block-level distributed parity)



(g) RAID 6 (dual redundancy)

### RAID anidado

La técnica de RAID se puede anidar, aplicando una capa de RAID sobre otra. Por eso, el esquema RAID 10 (también llamado esquema RAID 1+0) implementa RAID 1 sobre múltiples arreglos de discos los que a su vez implementan RAID 0. Análogamente se puede concebir el esquema RAID 0+1 (el cual no se lo denomina RAID 01 para evitar las confusiones con el RAID 1) o el esquema RAID 5+0.