TEMA 3: FILTROS DIGITALES: ESTRUCTURAS

1.- INTRODUCCIÓN

- Diseño de filtros: especificaciones → aproximación → realización

  especificaciones → $H_d(e^{j\omega})$

  aproximación → $H(e^{j\omega})$

  realización

- realización: dada una $H(z)$ objetivo, pretendemos determinar el algoritmo para calcular muestras de salida a partir de las muestras de entrada

$$x[n] \xrightarrow{H(z)} y[n]$$

En principio una $H(z)$ admite infinitos algoritmos para implementarla.

> algoritmo = estructura = $h[n]$ + diagrama de bloques

Ejemplo: $H(z) = \dfrac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}} \xrightarrow{TZ^{-1}} \sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$

ecuación en diferencias (en el tiempo)

2) estructura = forma posible de barajar $y[n]$

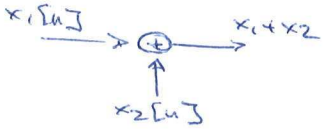
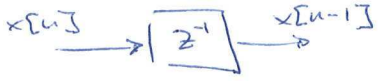
Diferentes estructuras se caracterizan por:

- coste computacional: nº operaciones / muestra y celdas de memoria necesarias
- sensibilidad frente a la aritmética de precisión finita

2.- REPRESENTACIÓN DE ESTRUCTURAS

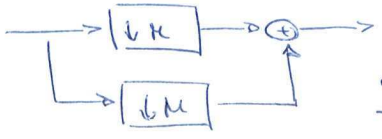
Vamos a ver 2 formas de representar los algoritmos: grafos / diagramas de bloques

Ambas tienen el mismo significado.

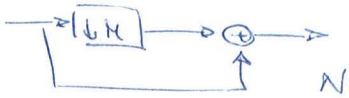
	diagrama de bloques	grafos
suma ⊛		
producto por constante		
retardo		

⊙ : nodo

⊛ → solo se pueden sumar señales del mismo nivel:

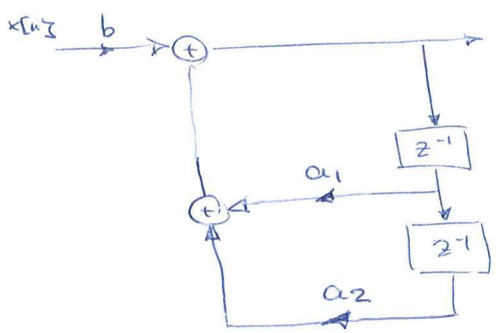


OK!



NO!

Ejemplo: $H(z) = \dfrac{b}{1 - a_1 z^{-1} - a_2 z^{-2}}$ $\Rightarrow$ $y[n] = b\,x[n] + a_1 y[n-1] + a_2 y[n-2]$



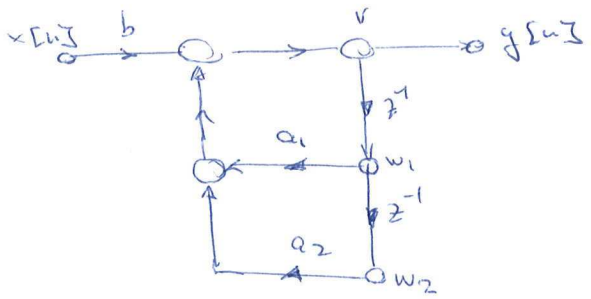
sumadores de 2 entradas, más parecidos al HW típico

por cada muestra de salida, se necesita:

- 3 productos
- 2 sumas
- 2 células de memoria

en este ejemplo, se necesita HW mínimo.

Con grafos: cada nodo representa una variable o señal, y las ramas son operaciones.



Código de un programa:

$v = x[n] \cdot b$

$v = v + w_1 a_1$

$v = v + w_2 a_2$

$y[n] = v$

} operaciones intercambiables, aunque no siempre será así $\Rightarrow$

$x = y$

$y = \square$

$\neq$

$y = \square$

$x = y$

3.- ESTRUCTURAS BÁSICAS PARA FILTROS IIR

IIR $\Rightarrow$ polos y ceros $\rightarrow$

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}}$$

3.1.- ESTRUCTURAS DIRECTAS

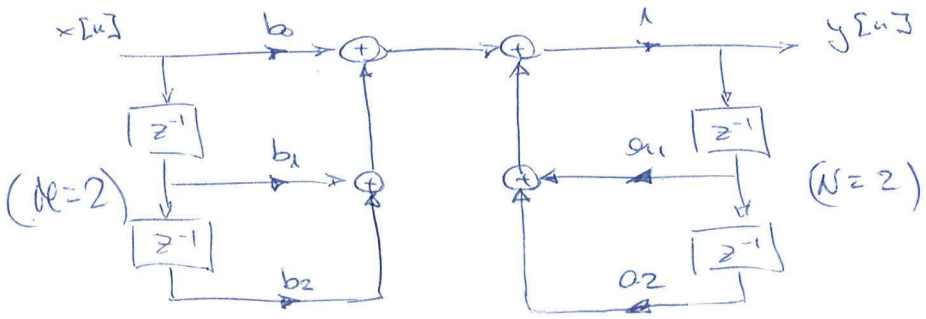
$$H(z) = \left(\sum_{k=0}^{M} b_k z^{-k}\right)\left(\frac{1}{\sum_{k=0}^{N} a_k z^{-k}}\right) = \left(\frac{1}{\sum_{k=0}^{N} a_k z^{-k}}\right)\left(\sum_{k=0}^{M} b_k z^{-k}\right)$$

forma directa I — forma directa II canónica

Ejemplo: $H(z) = \dfrac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$

- Directa I: $b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] = w[n]$ (numerador)

$y[n] = w[n] + a_1 y[n-1] + a_2 y[n-2]$ (denominador)

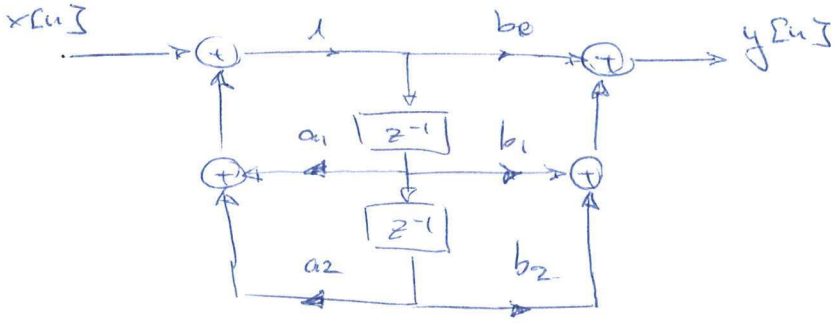


para cada muestra:
- 5 productos
- 4 sumas
- 4 celdas de memoria

en general, por cada muestra:

M+N+1 productos
M+N sumas
M+N celdas de memoria

- directa II:



para cada muestra:
- 5 productos
- 4 sumas
- 2 celdas de memoria

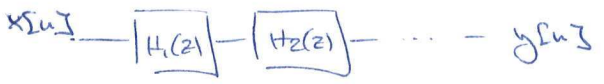
en general, para cada muestra:

M+N+1 productos
M+N sumas
máx(M,N) celdas de memoria

✓ necesita menos celdas de memoria

3.2- ESTRUCTURA EN CASCADA

Se factoriza $H(z)$, expresándolo como producto de secciones, y cada factor se implementa en estructura directa II.



factorización: 1.- Buscar polos y ceros

2.- $$H(z) = \frac{\prod_k (1-c_k z^{-1}) \prod_k (1-d_k z^{-1})(1-d_k^* z^{-1})}{\prod_k (1-p_k z^{-1}) \prod_k (1-q_k z^{-1})(1-q_k^* z^{-1})}$$

c_k = ceros simples; d_k = ceros complejos conjugados

p_k = polos simples; q_k = polos complejos conjugados

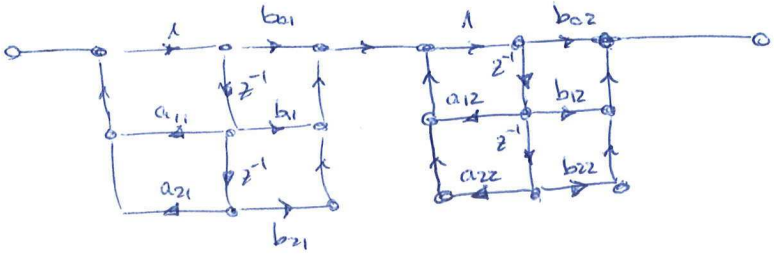
3- Criterio para factorizar: agrupar polos y ceros entre sí

- sistemas resultantes con coeficientes reales ⇒ los ceros o polos complejos conjugados deben ir juntos, al mismo subsistema
- cada sistema implementado con forma directa II. Para ello, agrupar polos con ceros, o polos y ceros sueltos.

Normalmente, 2 polos y 2 ceros por sistema.

Sistema resultante:

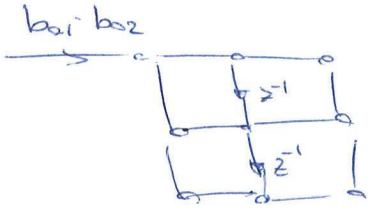
$$H(z) = \prod_k \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}}$$



✓ se reduce el efecto de los errores de cuantificación

Coste computacional:

- mismo número de celdas de memoria que forma directa II.
- mismo número de productos, pero ojo:



3.3- Estructura en Paralelo

Se usa menos que la estructura en cascada.

Se obtiene descomponiendo $H(z)$ en fracciones simples:

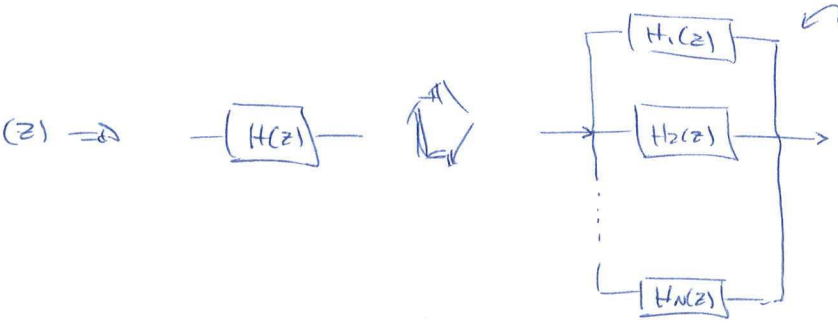
$$H(z) = \sum_{k=0}^{M-N} c_k z^{-k} + \sum_{k=1}^{N} \frac{A_k}{1 - p_k z^{-1}}$$

p_k: polos

$M \geq N \Rightarrow$ numerador de orden mayor o igual $\Rightarrow$ aparece el cociente N/D

Suele $\Rightarrow$ conectar en paralelo

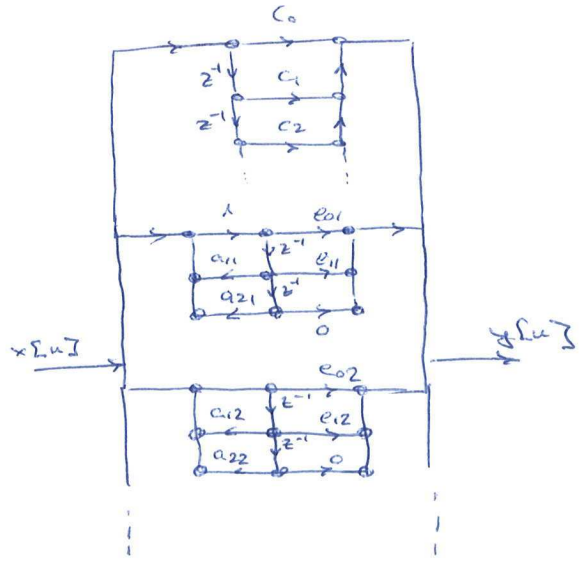
$$H(z) = \sum_k H_k(z) \Rightarrow$$



↳ cada $H_k(z)$ en forma directa II

NOTA: hacer $H_k(z)$ agrupando los polos complejos conjugados en el mismo k.

Sistema/estructura resultante: $H(z) = \sum_{k=0}^{M-N} c_k z^{-k} + \sum_{k=1}^{N/2} \frac{e_{0k} + e_{1k} z^{-1}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}}$



- Cascada: $H(z) = \frac{N_1}{D_1}\frac{N_2}{D_2}\ldots$ $\Rightarrow$ $H(z) = \frac{N_1 N_2}{D_1 D_2}$

- Paralelo: $H(z) = \frac{N_1}{D_1} + \frac{N_2}{D_2} + \ldots$ $\Rightarrow$ $H(z) = \frac{N_1 D_2 + N_2 D_1}{D_1 D_2}$

· Cascada: los ceros de $H(z)$ están repartidos en las distintas secciones, y ocurre lo mismo con los polos.

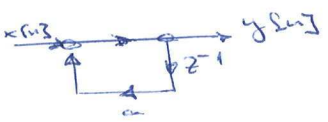
· Paralelo: polos de $H(z)$ están en las distintas secciones, pero los ceros no tienen porqué coincidir.

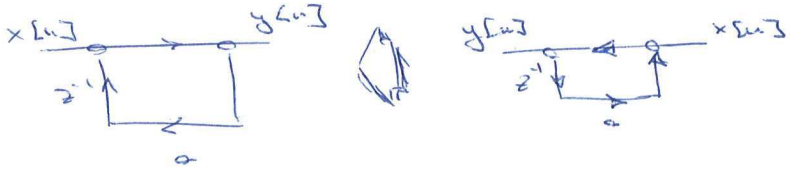
3.4.- FORMAS TRASPUESTAS

La teoría de grafos dice que si en una estructura se cambia el sentido de las ramas y la entrada por la salida, se obtiene la misma función de transferencia.

Para cada una de las estructuras anteriores, tendremos la correspondiente traspuesta.

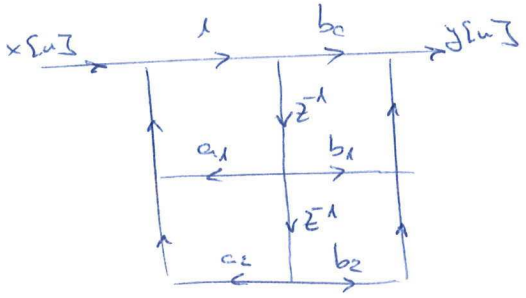
Ejemplo: $H(z) = \frac{1}{1 - a z^{-1}}$



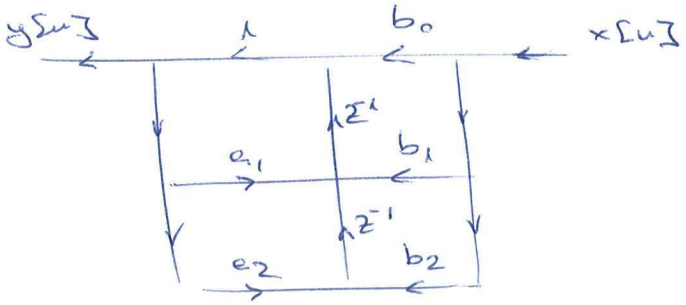


Ejemplo: $H(z) = \dfrac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$

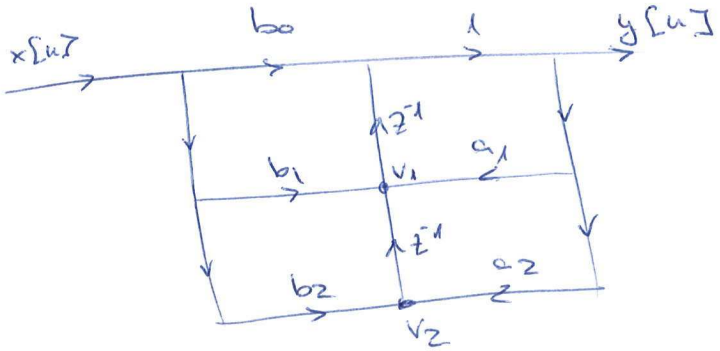
grafo forma directa II:



transposición:



reordenando:



Comprobando la equivalencia: $\dfrac{Y(z)}{X(z)} = ?$

$$Y(z) = b_0 X(z) + z^{-1} V_1(z)$$
$$V_1(z) = b_1 X(z) + a_1 Y(z) + z^{-1} V_2(z)$$
$$V_2(z) = b_2 X(z) + a_2 Y(z)$$

$$(\dots) \Rightarrow H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

¡Ok!

4.- ESTRUCTURAS BÁSICAS PARA FILTROS FIR

Los filtros FIR son un caso particular de los IIR:

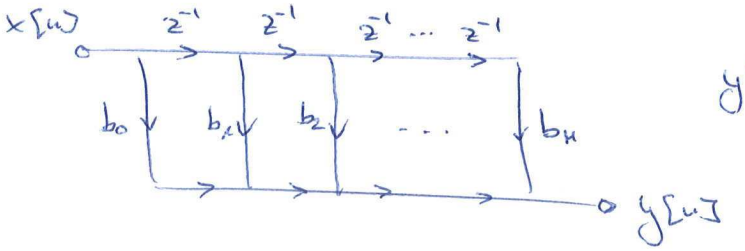
$$H(z) = \sum_{k=0}^{M} b_k z^{-k}$$

valen las mismas estructuras

4.1.- ESTRUCTURAS DIRECTAS

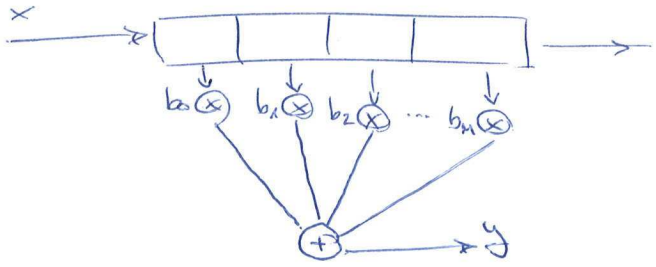
Son iguales que para los IIR en ambas formas.

Se suele pintar horizontalmente: línea de retardo



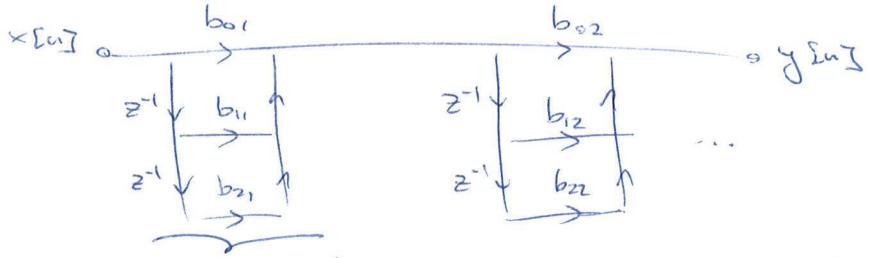
$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

Implementación hardware sencilla: registro de desplazamiento



4.2.- ESTRUCTURA EN CASCADA

$$H(z) = \prod_k \left(b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2} \right)$$

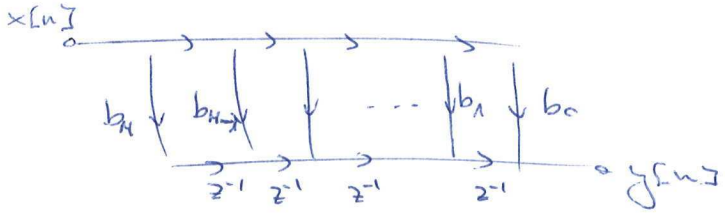


2 ceros complejos conjugados a reales para tener la estructura repetitiva

4.3.- ESTRUCTURA EN PARALELO

No hay cociente (denominador = 1), por lo que se obtiene la propia forma directa.

4.4.- FORMA TRASPUESTA



4.5.- FILTROS FIR CON FASE LINEAL GENERALIZADA

AKA "ca respuesta simétrica"

$h[n] = \pm h[M-n]$, bien par o impar

$\equiv$ $\underline{b_n = \pm b_{M-n}}$

$$y[n] = b_0 x[n] + b_1 x[n-1] + \dots + b_{M-1} x[n-M+1] + b_M x[n-M]$$

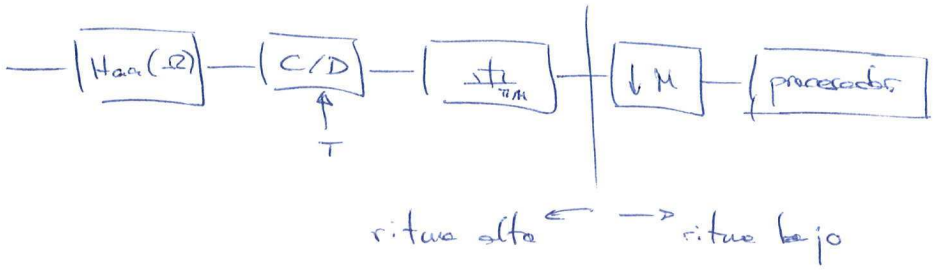
(iguales ($\pm$))

$$y[n] = b_0 \left(x[n] \pm x[n-M] \right) + b_1 \left(x[n-1] \pm x[n-M+1] \right) + \dots$$

=> se pueden simplificar las operaciones a realizar (unidad de multiplicaciones)

2 => estructura plegada (* 3.3 *)

4.6.- FILTROS FIR CON DIEZMADO



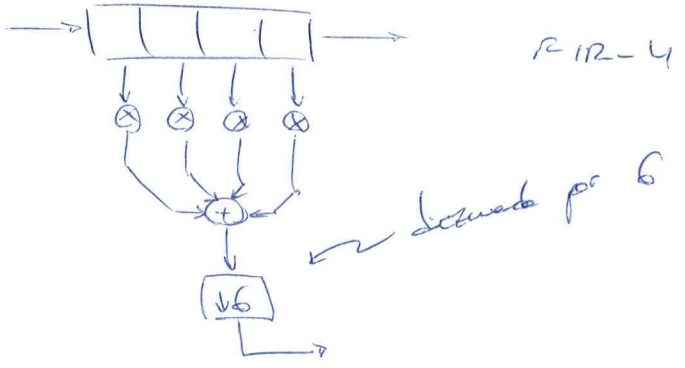
posible inconveniente: filtro de diezmado que funciona a régimen alto.

ej: filtro paso bajo FIR-100 (100 coeficientes) y $f_s = 6$ MHz

=> 6 millones y pico de productos por segundo

Total, para que luego llegue el diezmador y desearte un montón de resultados

ejemplo:

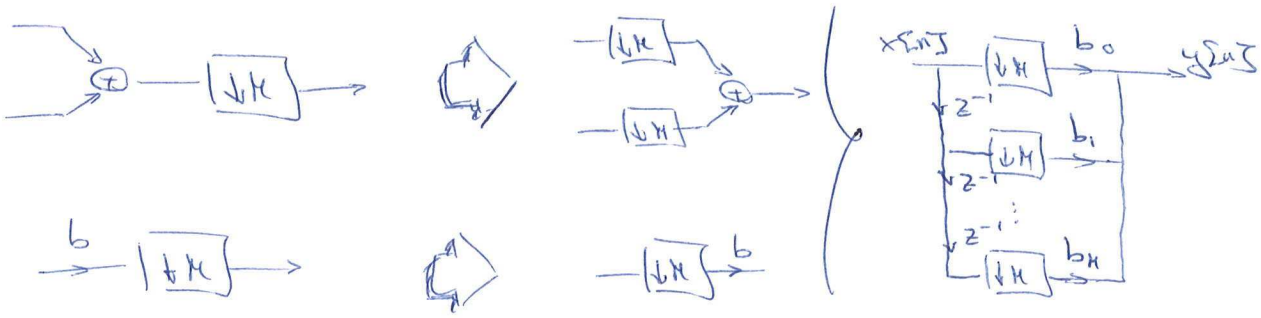


Sol: si se toca quedarse con el valor, paso de operar todo, sólo desplaza

operación → desplaza 5 → operación → ...

=> si luego hay un diezmado, se pueden simplificar los filtros.

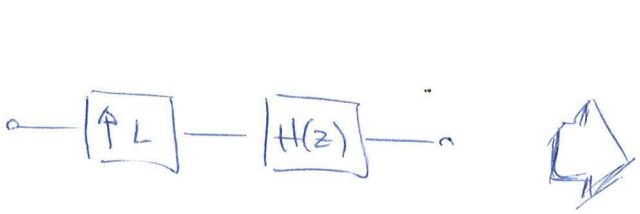
(* 3.4 *)

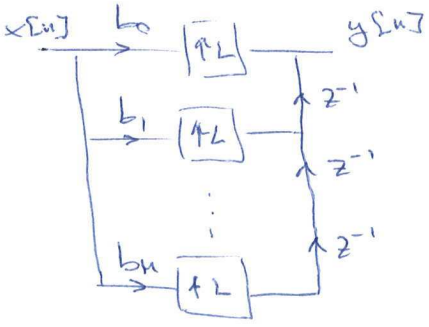


OJO: el factor de diezmado NO tiene porqué coincidir con el orden del filtro fir

Se reduce el número de multiplicaciones necesarias en un factor M

4.7.- EXPANSOR CON FILTRO FIR





(* 3.5 *)

OJO: debe ser en la estructura traspuesta

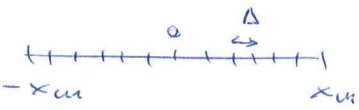
El expansor introduce ceros => no sirve multiplicar

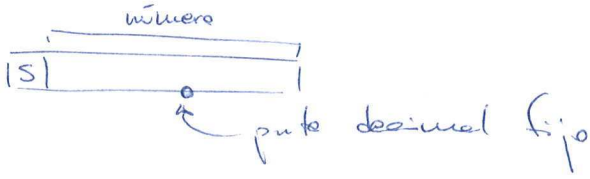
5.- EFECTOS DE LA ARITMÉTICA DE PRECISIÓN FINITA

Hoy 2 formas de representar números en TDS:

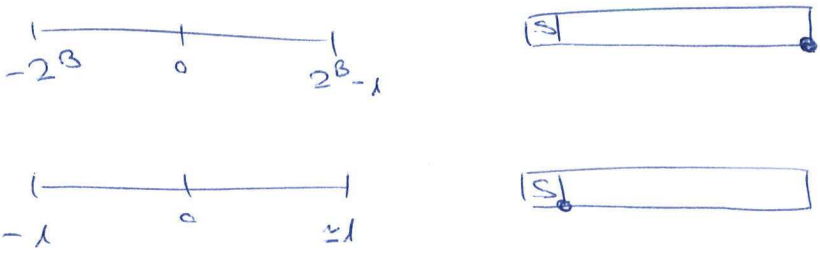
- punto fijo: cuantificación uniforme, B+1 bits

→ error constante





rango de números representables: viene según la posición del punto y del nº de bits



- punto flotante: bits organizados en signo - mantisa - exponente

→ cuantificación no uniforme → error proporcional al valor de la señal

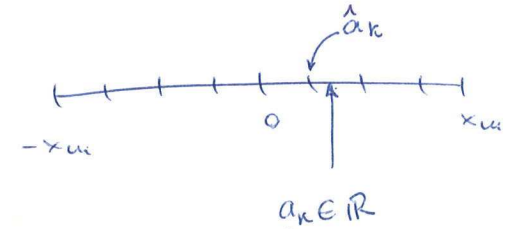
Los errores en punto fijo son mucho mayores

	Complejidad aritmética	Velocidad	errores	
punto fijo	menor	mayor	no despreciable	← más complicado de programar
punto flotante	mayor	menor	despreciable	

5.1.- CUANTIFICACIÓN DE COEFICIENTES

Trabajamos en punto fijo.

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$



$a_k \in \mathbb{R}$

hay que representarlo por el más cercano (redondeo)

$\Rightarrow$ error en la cuantificación de los coeficientes

Los ceros dependen de los b_k y los polos de los a_k

$$\Rightarrow \hat{H}(z) = \frac{\sum_{k=0}^{M} \hat{b}_k z^{-k}}{1 - \sum_{k=1}^{N} \hat{a}_k z^{-k}}$$

cómo afectan las variaciones de a_k y b_k a las posiciones de polos y ceros?

ceros: $c_i = f(b_k)$

polos: $p_i = f(a_k)$

$\rightarrow$ sensibilidad

$$\frac{\partial p_i}{\partial a_k} = \frac{p_i^{N-k}}{\prod_{\substack{j=1 \\ j \neq i}}^{N} (p_i - p_j)}$$

- válida para formas directa I y II
- la sensibilidad aumenta mucho si los polos están cerca (eso es malo)
  - $\hookrightarrow p_i - p_j$ en el denominador

Igual para los ceros:

$$\frac{\partial c_i}{\partial b_k} = \frac{c_i^{M-k}}{\prod_{\substack{j=1 \\ j \neq i}}^{M} (c_i - c_j)}$$

en la práctica el peligro lo suelen provocar los polos

Solución: usar una estructura de filtros en cascada, colocando polos próximos en etapas distintas.

Una estructura en paralelo no vale, porque los polos aparecen en todas las estructuras.

Problema adicional: polos complejos conjugados próximos

⇒ inseparables.

⇒ estructura acoplada

Ejemplo: filtro orden 4 → (x 3.6 − 3.9 x)

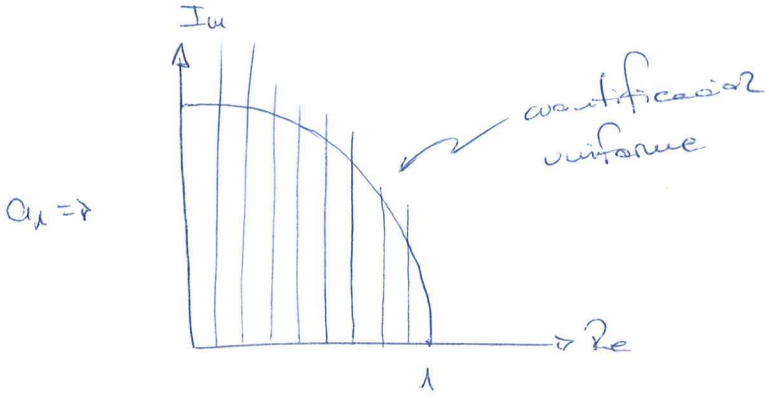
- Polos complejos conjugados muy próximos:

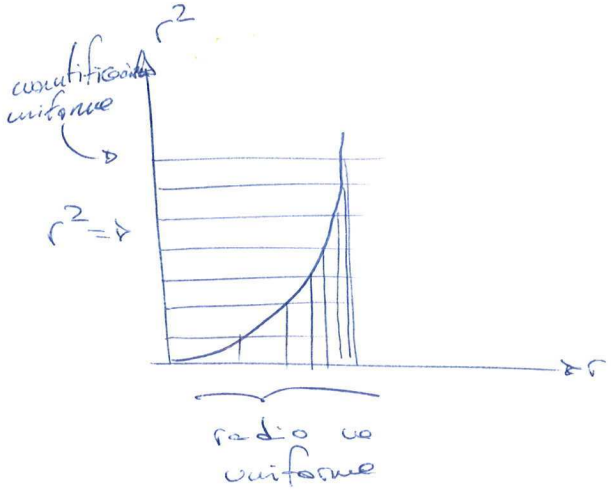
$$p = r e^{j\theta} \quad \text{con} \quad \theta \simeq 0 \text{ ó } \pi$$

$$H(z) = \frac{1}{(1 - p z^{-1})(1 - p^* z^{-1})} = \frac{1}{1 - 2r\cos\theta\, z^{-1} + r^2 z^{-2}}$$

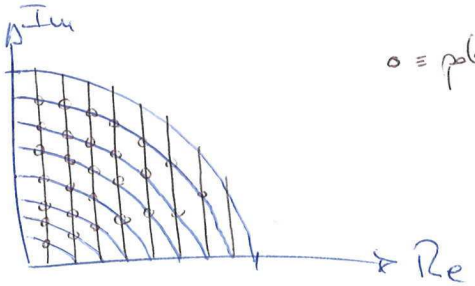
$$= \frac{1}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

$$\rightarrow \begin{cases} a_1 = 2r\cos\theta \\ a_2 = -r^2 \end{cases} \quad a_1 = 2\,\mathrm{Re}(p)$$





Llevando los polos al a_k:



o ≡ polos repartidos uniformemente, implementables

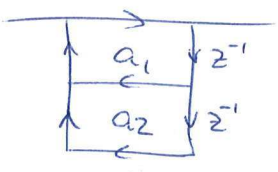
polos cercanos al eje real ⇒ mucho error

(* 3.9 *)

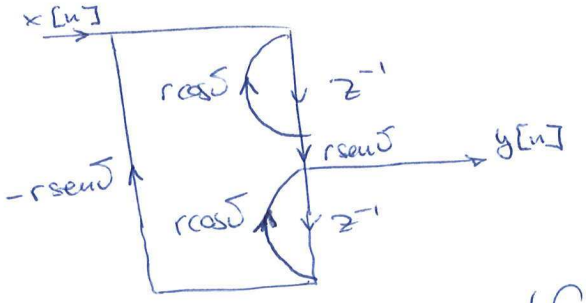
Solución: en vez de forma directa II de orden 2, usar estructura acoplada de orden 2:

(coeficientes estructura ≠ coeficientes H(z))

estructura directa



estructura acoplada

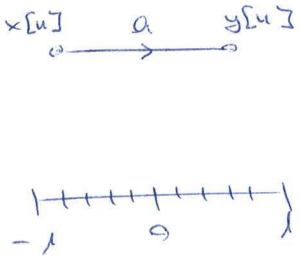


En este caso, al cuantificar los coeficientes se cuantifica $r\cos\theta$, $r\sin\theta$ y $-r\sin\theta$ ⇒ cuantificación uniforme de las partes real e imaginaria del polo

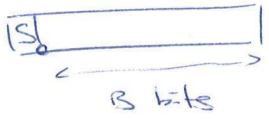
⇒ se distribuye de forma uniforme en el círculo unidad

los polos siguen cerca, pero influye menos la cuantificación

5.2.- REDONDEO EN LOS PRODUCTOS

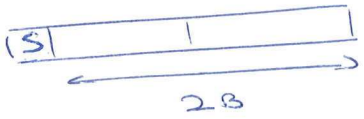


→ x[n], y[n] y a son de B+1 bits



en punto fijo

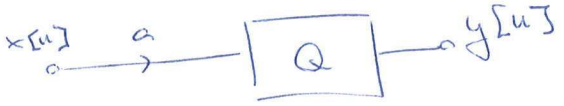
Para hacer el producto hacen falta 2B+1 bits:



→ truncar / redondear

⟹ error

cuantificador uniforme:

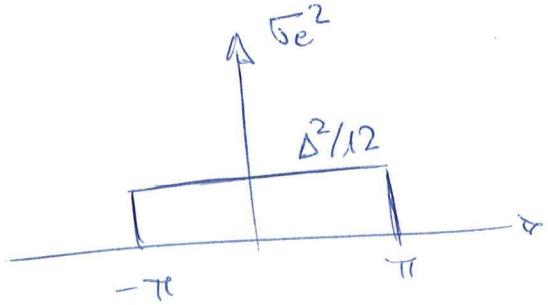


$$\Delta = \frac{x_m}{2^B} = \frac{1}{2^B} = 2^{-B}$$

Efecto del error de redondeo en procesos:

- error de cuantificación Q: espectro blanco, independiente de la señal, distribución uniforme $[-\Delta/2, \Delta/2]$

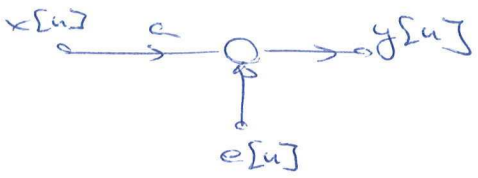
$$\sigma_e^2 = \frac{\Delta^2}{12}$$



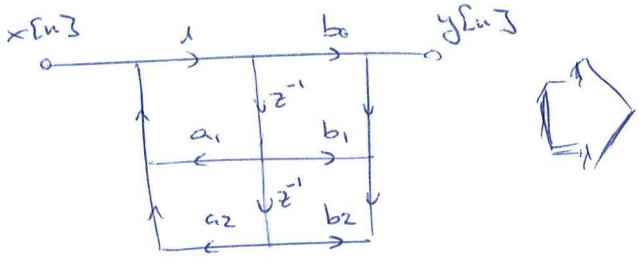
Si se usa truncamiento:

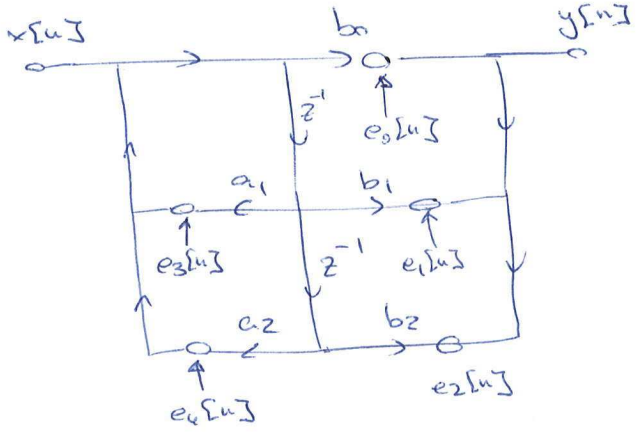
- uniforme en $[0, \Delta]$
- misma potencia: $\sigma_e^2 = \frac{\Delta^2}{12}$
- media mayor: $\Delta/2$

En resumen, se modela como ruido:



Ejemplo: estructura directa II



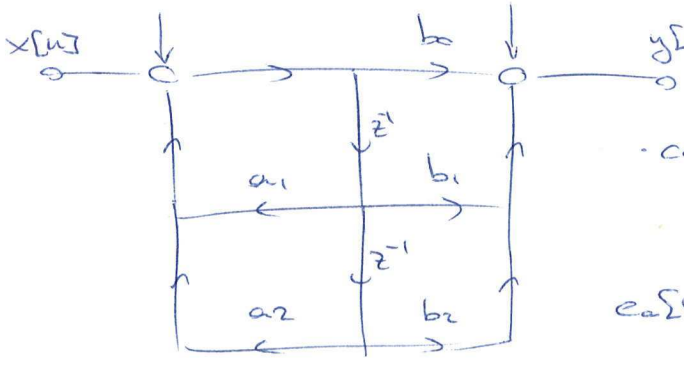


$x[n] \longrightarrow y[n] + f[n]$

$f[n] \equiv$ ruido a la salida

Aplicando movilidad de generadores:

$e_a[n] = e_3[n] + e_4[n]$ $\quad e_b[n] = e_0[n] + e_1[n] + e_2[n]$



· como son ortogonales, incorrelados, blablabla, ...:

$e_a[n] \equiv$ blanco, DEP plana, valor $2\Delta^2/12$
(suma de 2 señales incorrelados)

$e_b[n] \equiv$ blanco, $3\Delta^2/12$

Para la DEP de salida:

$$P_f(\omega) = \frac{3\Delta^2}{12} + \frac{2\Delta^2}{12} |H(e^{j\omega})|^2$$

potencia: $\sigma_f^2 = \frac{1}{2\pi} \int_{-\pi}^{\pi} P_f(\omega) d\omega$

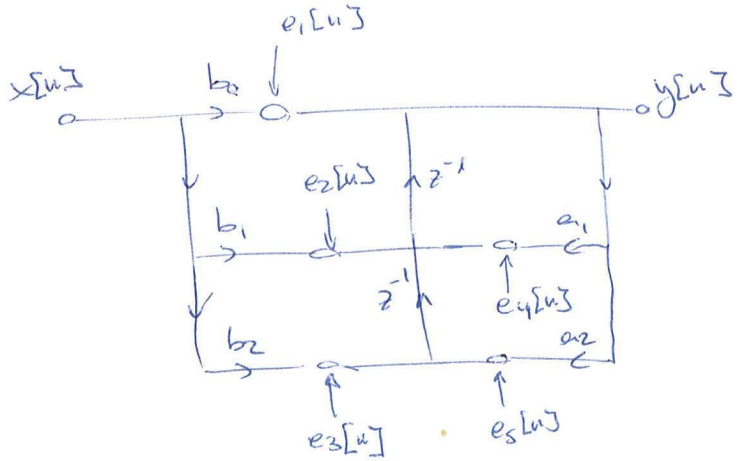
→ depende de $|H(e^{j\omega})|^2$

Se puede usar el teorema de Parseval:

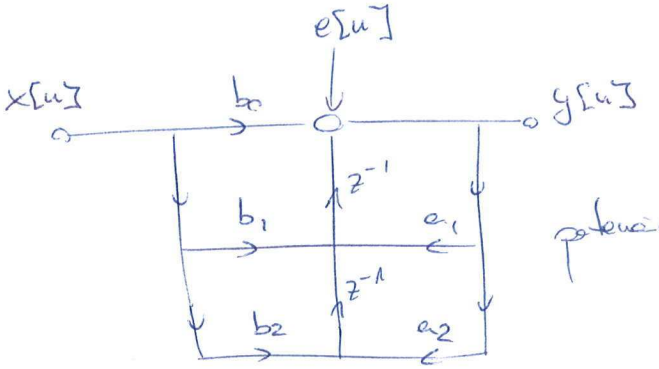
$$P = \frac{1}{2\pi} \int_{-\pi}^{\pi} P(\omega) d\omega = \frac{3\Delta^2}{12} + \frac{2\Delta^2}{12} \frac{1}{2\pi} \int_{-\pi}^{\pi} |H(e^{j\omega})|^2 d\omega$$

$$= \frac{3\Delta^2}{12} + \frac{2\Delta^2}{12} \sum_{n=-\infty}^{\infty} h^2[n]$$

Ejemplo: estructura directa II traspuesta



Aplicando movilidad de generadores: como son estadísticos, no afectan los z^{-1}



potencia$(e[n]) = \sum$ potencia$(e_i[n])$

$\sigma_e^2 = \frac{S\Delta^2}{12}$

$P_f(\omega) = \frac{S\Delta^2}{12} \frac{1}{|A(e^{j\omega})|^2}$ → peligro por los polos en $A(z)$

Si $H(z) = \frac{B(z)}{A(z)}$: 

Propuesta: estructura directa I orden 2.

Sol: $P_f(\omega) = \frac{S\Delta^2}{12} \frac{1}{|A(e^{j\omega})|^2}$

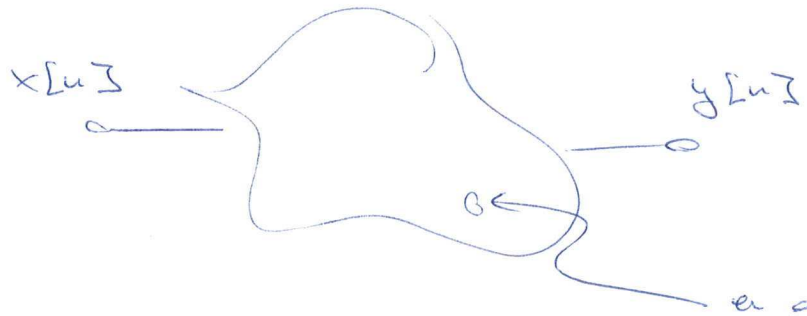
¿Mejor estructura para reducir en productos?

→ depende del filtro en concreto

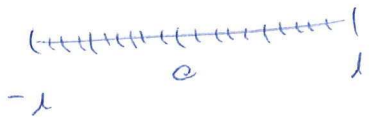
¿Mejor para cuantificación de coeficientes?

→ cascada

5.3.- DESBORDAMIENTO



si algún punto podemos salirnos del rango representable



⇒ saturación en conversores

⇒ al operar se puede salir del rango y el error se desborda

- Evitar el desbordamiento:

· modo bestia: para cada nodo intermedio $v_i[n]$, calcular $h_i[n]$

debe ser $|v_i[n]| < 1 \rightarrow |x[n] * h_i[n]| = \left|\sum_k h_i[k] \cdot x[n-k]\right| < 1$

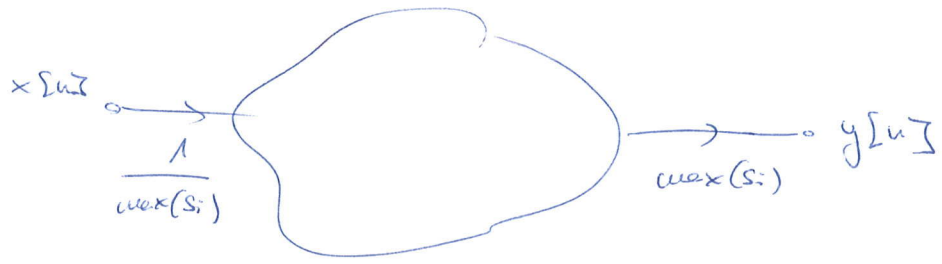
acotando la suma: $|\Sigma \ldots| \leq \sum_k |h_i[k] x[n-k]| < 1$

$x[n] \in [-1, 1]$ ← en nuestra representación

$$\Rightarrow \boxed{\sum_k |h_i[k]| < 1} \quad \forall i \in \mathbb{N}$$

Condición muy restrictiva, que garantiza ausencia de desbordamiento.

Si en $S_i = \sum_k |h_i[k]| > 1$, para que no haya desbordamiento, se puede dividir la entrada por S_i:



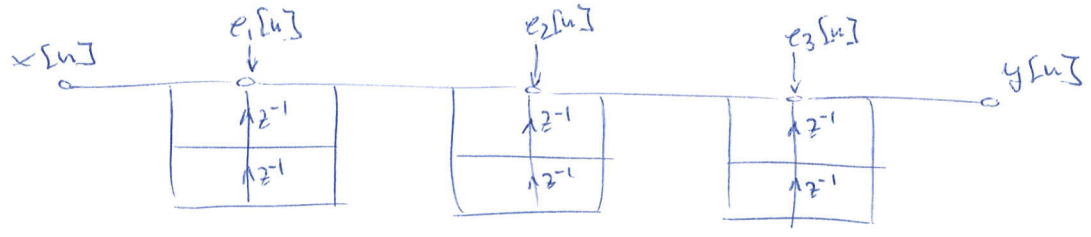
Problema: estamos disminuyendo mucho la amplitud de entrada
→ SNR cuantif ↓↓

ruido de redondeo de coeficientes independiente de la señal → SNR ↓↓ también

Compromiso entre desbordamiento y SNR

Ejemplo: $H(z) = K \prod_{k=1}^{3} \frac{1 + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}}$

implementada en forma directa II en cascada (trasperfes):



constante $K = b_{01} b_{02} b_{03}$

1.- Si $K<1$, ¿dónde la ponríamos?

☐ al principio: como es menor que 1, atenúa la entrada
⇒ SNR ↓↓ → no vale

☐ al final: no influye en la SNR, pero es más posible que haya desbordamiento

☒ repartirla entre etapas: compromiso entre principio y final.
En la práctica, haríamos un reparto de la K global en las estructuras

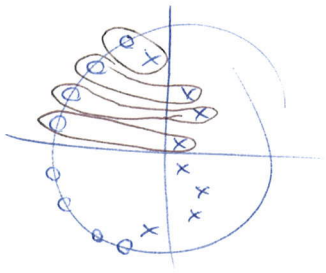
2.- Si hay una etapa conflictiva (polos muy cerca de $|z|=1$) ¿dónde colocarla?

☐ al principio: ganancia muy grande (polos en $|z|\simeq 1$)
⇒ desbordamiento

☐ al final: amplifica mucho el ruido de los polos de su propia etapa (no hay nada para atenuarlo después)
⇒ SNR ↓

☒ reglas de diseño en base a diagrama polo/cero.

Ejemplo:



en cascada, 2° orden

regla práctica

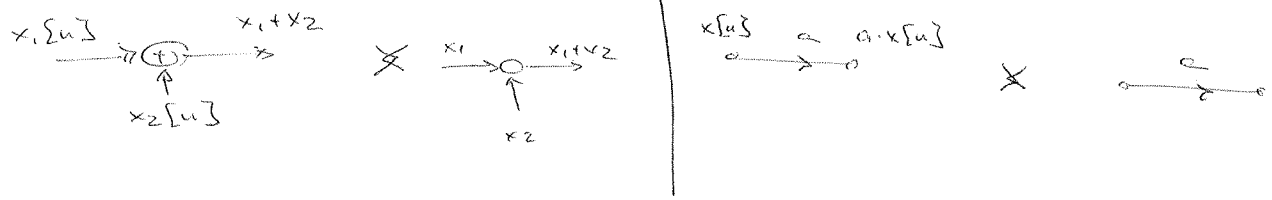
- polo más cercano al círculo unidad y se le asocia su cero más cercano al polo
- seguimos igual, por orden creciente de distancia al círculo.

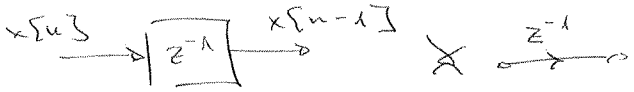
Esto evita, a cierta medida, las "etapas conflictivas" de polos cercanos al círculo unidad con ceros alejados

- orden de las etapas: en sentido bien creciente bien decreciente de proximidad de polos al círculo unidad.

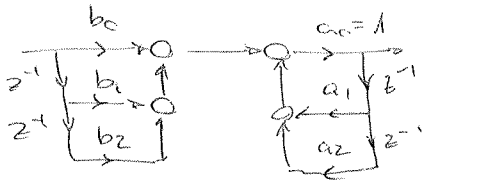
TDSI

TEMA 3



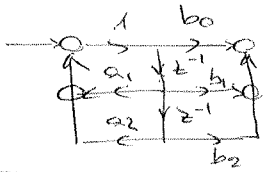


- FORMAS DIRECTAS: $H(z) = \sum_{k=0}^{M} b_k z^{-k} \cdot \frac{1}{\sum_{k=0}^{N} a_k z^{-k}} = \frac{1}{\sum_{k=0}^{N} a_k z^{-k}} \cdot \sum_{k=0}^{M} b_k z^{-k}$



(I) N+M+1 productos
N+M sumas
N+M celdas

(II) N+M+1 productos
N+M sumas
máx(N,M) celdas



- ESTRUCTURA EN CASCADA:

$$H(z) = \frac{\prod_k (1 - c_k z^{-1}) \prod_k (1 - d_k z^{-1})(1 - d_k^* z^{-1})}{\prod_k (1 - p_k z^{-1}) \prod_k (1 - f_k z^{-1})(1 - f_k^* z^{-1})}$$

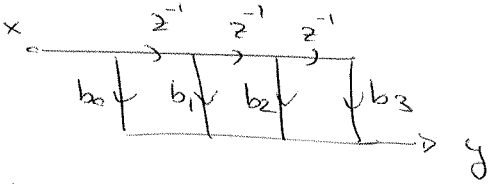
c_k = ceros simples
p_k = polos simples
d_k = ceros complejos conjugados
q_k = polos complejos conjugados

$$H(z) = \prod_k \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 - a_{1k} z^{-1} - a_{2k} z^{-2}}$$

✓ reduce el efecto de los errores de cuantificación

- ESTRUCTURAS FIR: $H(z) = \sum_{k=0}^{M} b_k z^{-k}$

Línea de retardo



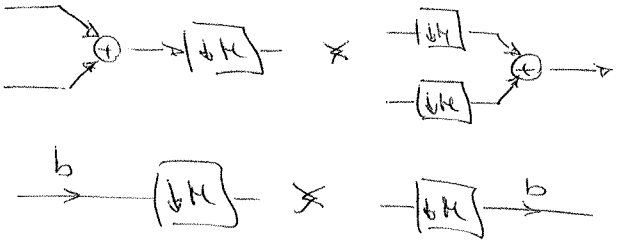
Cascada:

$$H(z) = \prod_k (b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2})$$

- FIR-FLG: $h[n] = \pm h[M-n] \Rightarrow b_n = \pm b_{M-n}$

→ estructura plegada ⇒ menos productos

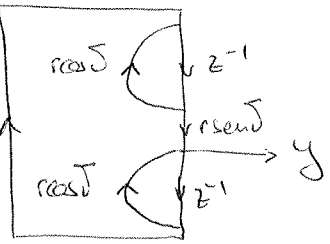
- FIR + lazada:



- FIR + expansión: ≡ a forma traspuesta

- EFECTOS DE LA PRECISIÓN FINITA:

→ estructura acoplada: $\times$



$p = r \cdot e^{jJ}$

⇒ distribución uniforme en el círculo

$$\text{sensibilidad} = \frac{\partial p_i}{\partial a_k} = \frac{p_i^{N-k}}{\prod_{\substack{i=1 \\ i \neq i}}^{N} (p_i - p_i)}$$

(polos → más peligrosos)

- acción de desbordamiento: $\sum_k |h_i[k]| < 1$

FILTROS DIGITALES: REALIZACIÓN

Representación de valores con precisión finita

- Las muestras de señal se almacenan en registros binarios de longitud finita
- Las constantes del algoritmo (ganancias de rama) también
- Una palabra binaria, sólo puede representar:
  - un rango de valores,
  - con una precisión limitada

Hay que tener en cuenta ...

- **Cuantificación** de **señales**
- **Cuantificación** de **coeficientes**
- **Redondeo** después de una multiplicación (para poder almacenar el resultado)
- **Desbordamiento**: el resultado de una operación supera el rango representable

Los efectos del uso de aritmética de precisión finita...

dependen de la estructura

FILTROS DIGITALES: CONTRUCCION

1 Solución Hardware

- Registros, sumadores, multiplicadores
- Puede haber operaciones simultáneas si el algoritmo lo permite
- Rápido
- La cantidad de hardware crece con la complejidad del algoritmo

2 Solución Software: Procesador digital de señal (DSP)

- Registros, ALU, Multiplicador, Unidad de Control, Memoria (datos y programa)
- Todas las operaciones son secuenciales: más lento
- Si la complejidad del algoritmo crece:
  - La cantidad de HW no crece (excepto la memoria)
  - El tiempo de procesado de una muestra crece
  - La frecuencia máxima admisible de las señales decrece

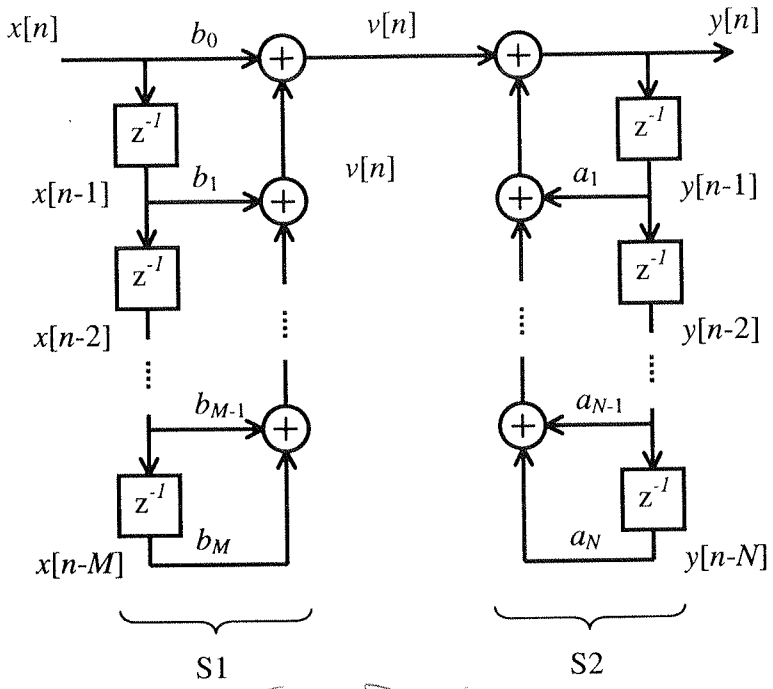
3 Solución mixta: Procesador con paralelismo

Varias: ALU, multiplicadores, memorias separadas, etc.

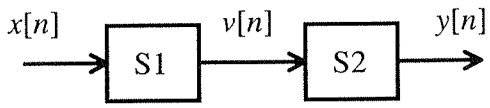
Los efectos del uso de aritmética de precisión finita...

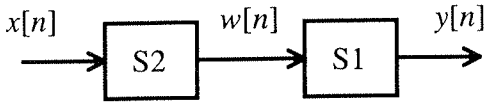
- **Sólo dependen de la estructura**
- **No dependen de la construcción**

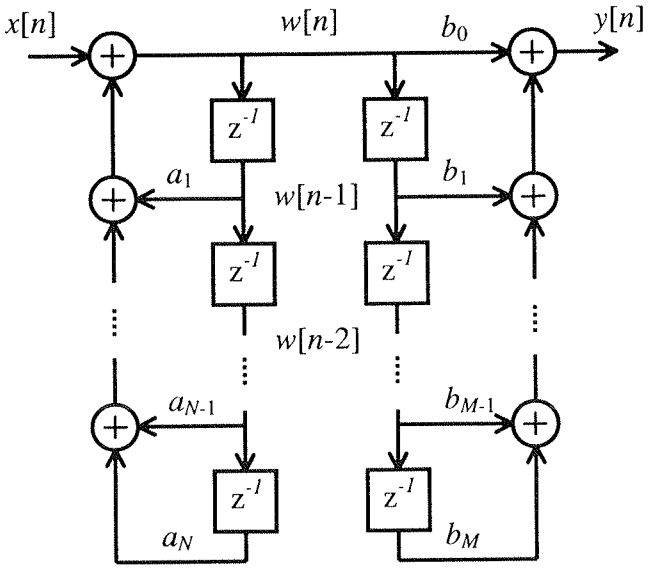
(a igualdad de longitud de palabra)

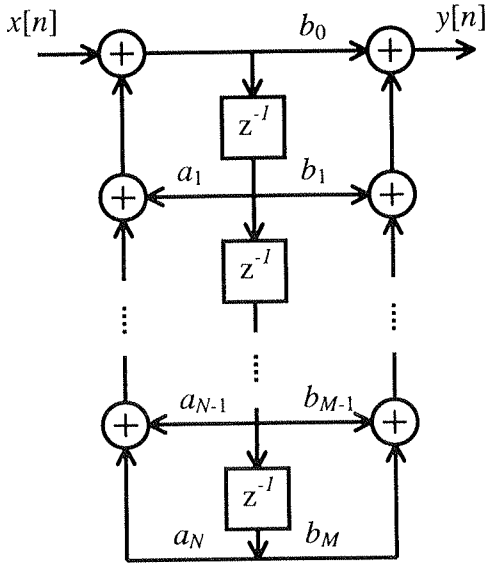


Forma Directa I









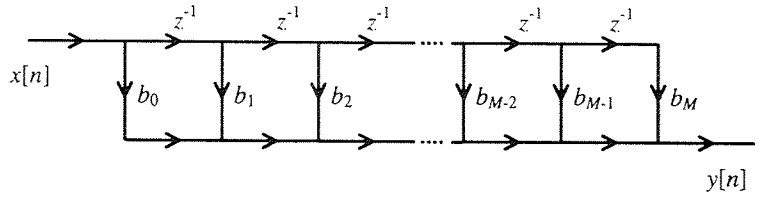
(M=N)

Forma Directa II

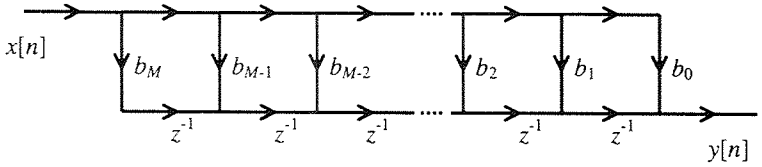
máxima optimización, ya que se aprovechan todas las celdas de memoria

34

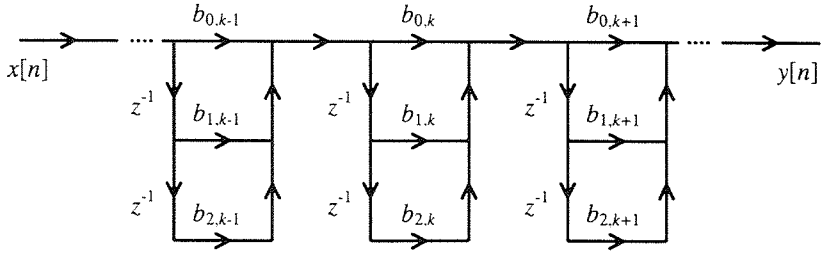
Estructuras FIR



Estructura Directa



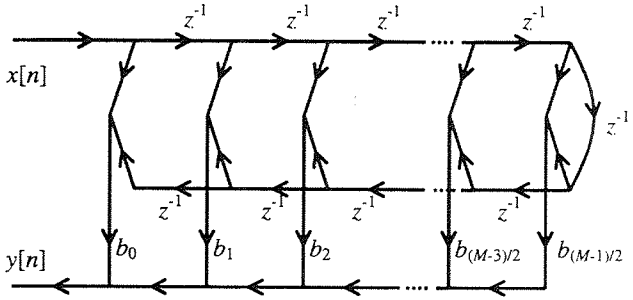
Estructura directa transpuesta



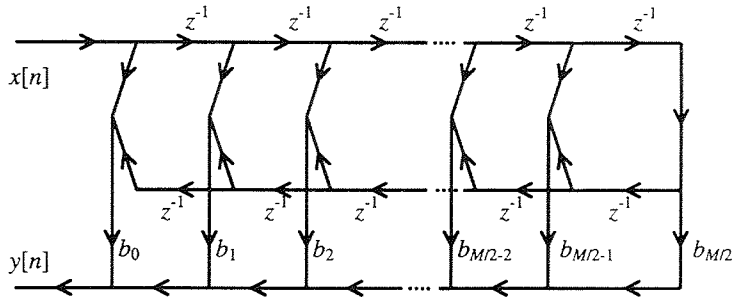
Estructura en cascada

Estructuras FIR

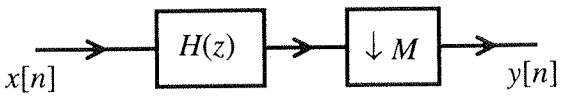
estructuras plegadas



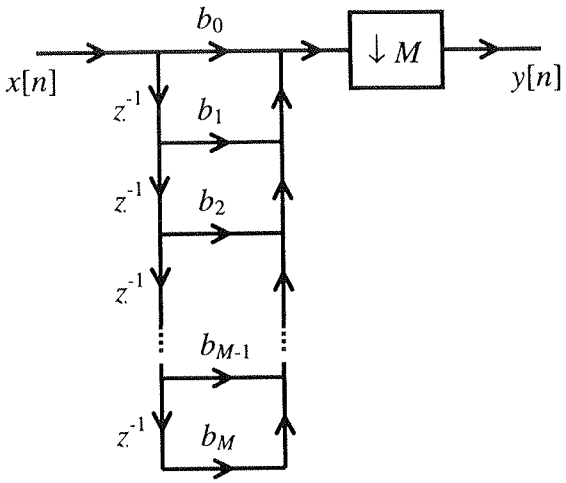
FIR fase lineal (M impar)

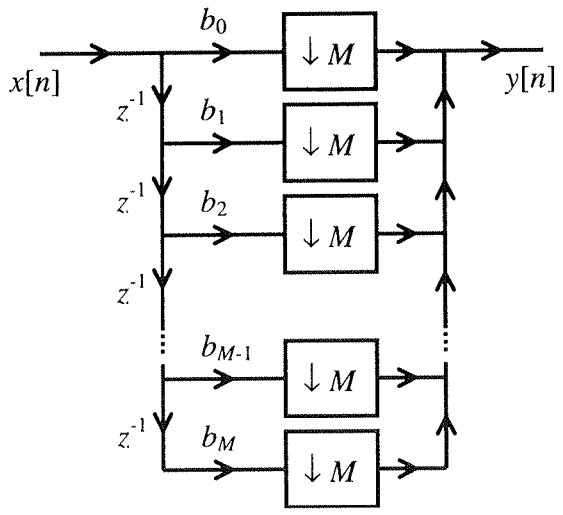


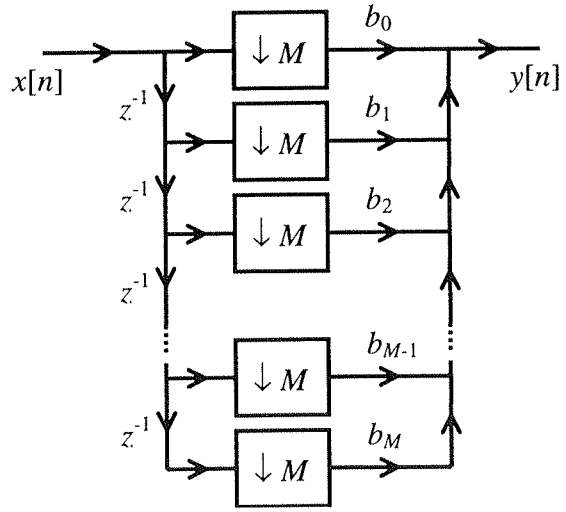
FIR fase lineal (M par)



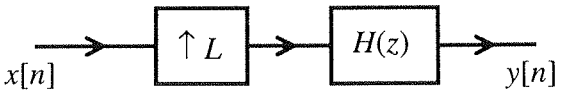
$$H(z) = \sum_{k=0}^{M} b_k z^{-k}$$



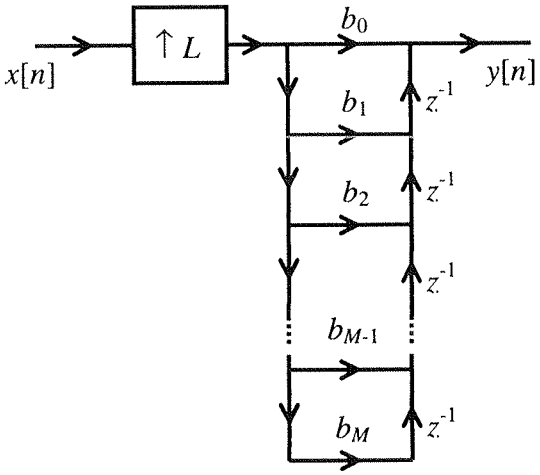


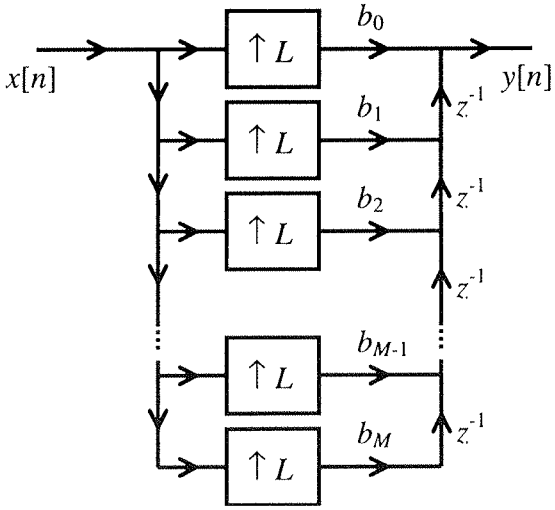


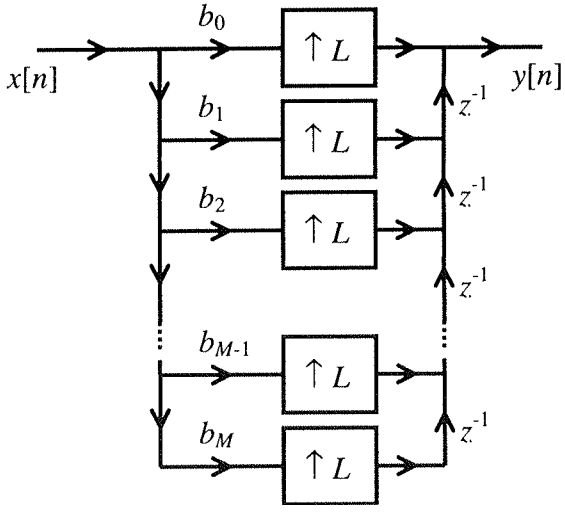
Estructura para diezmado con filtro FIR *antialiasing*



$$H(z) = \sum_{k=0}^{M} b_k z^{-k}$$

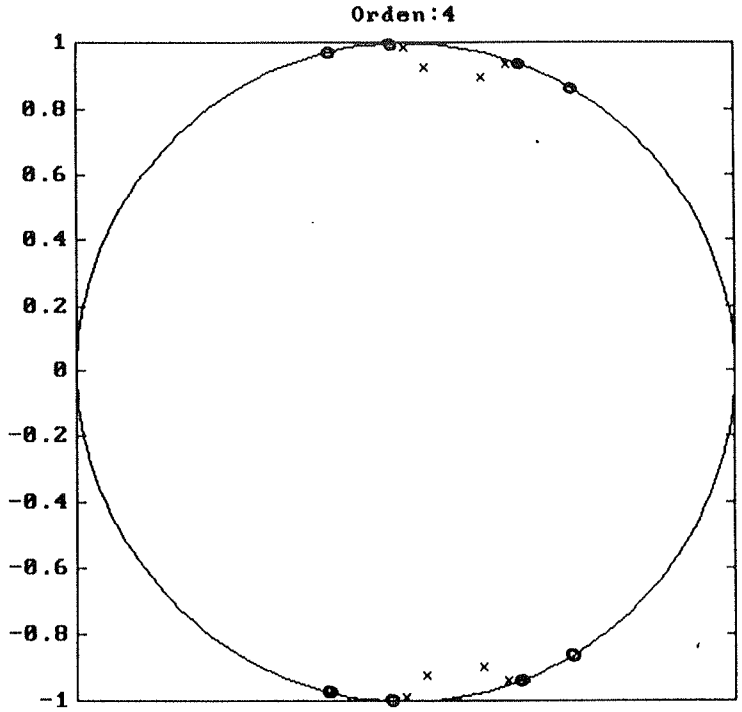


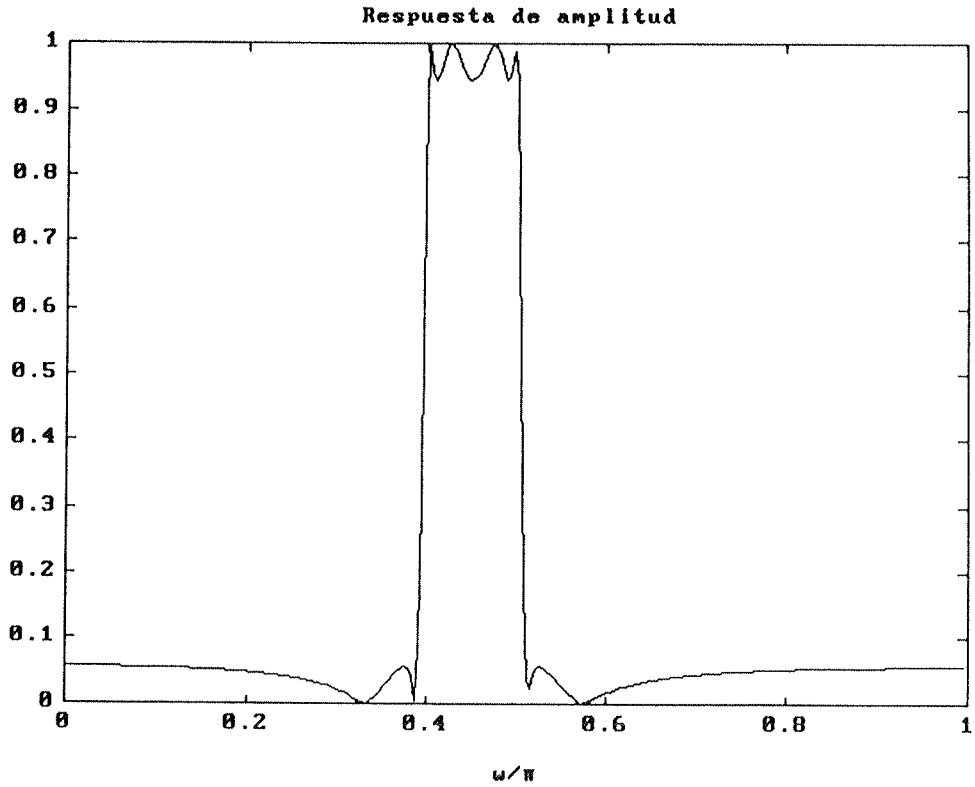




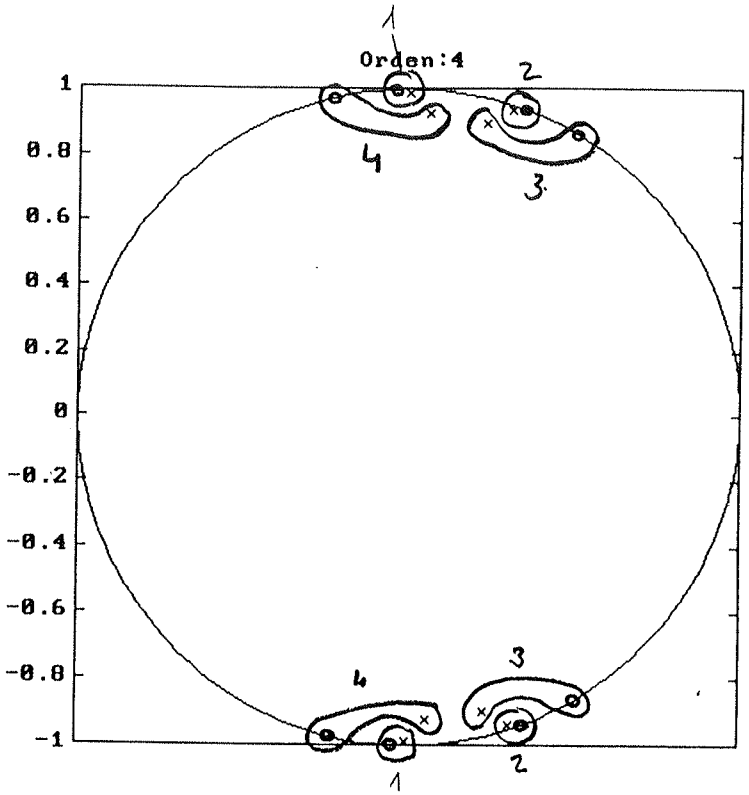
Estructura para interpolación con filtro FIR

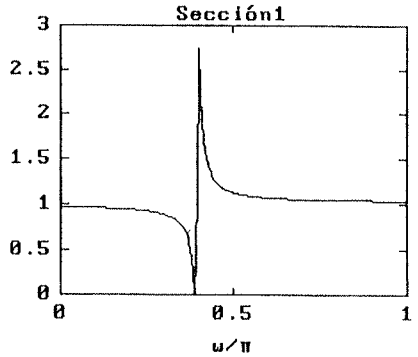
Filtro paso banda de orden 4

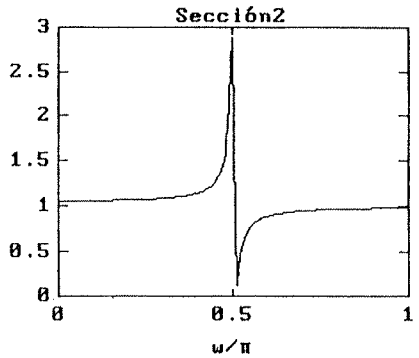


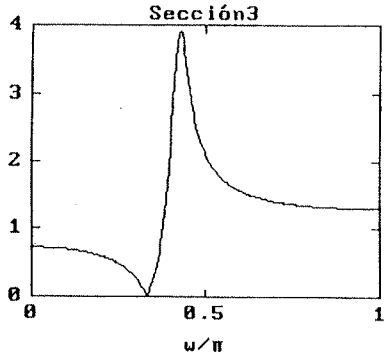


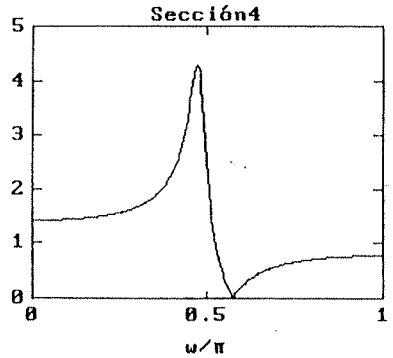
3.6


Orden:4
1
2
3
4
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1

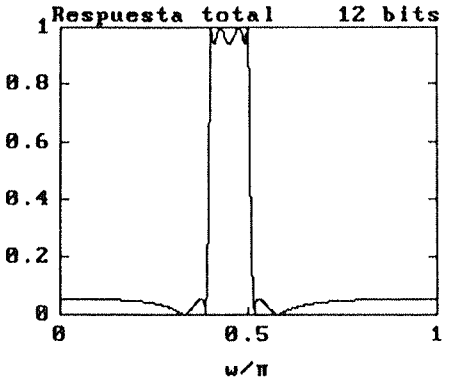

Sección1
3
2.5
2
1.5
1
0.5
0
0
0.5
1
w/π

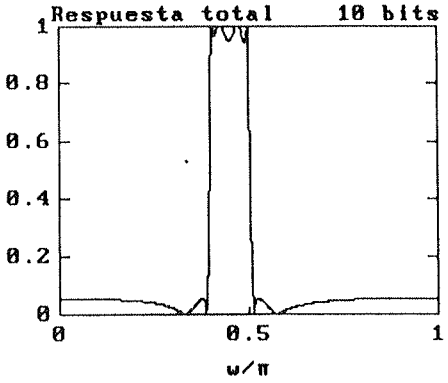

Sección2
3
2.5
2
1.5
1
0.5
0
0
0.5
1
w/π

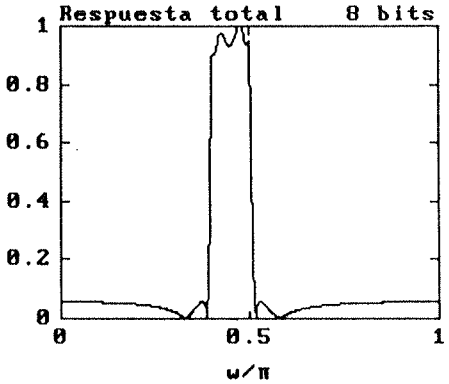

Sección3
4
3
2
1
0
0
0.5
1
w/π

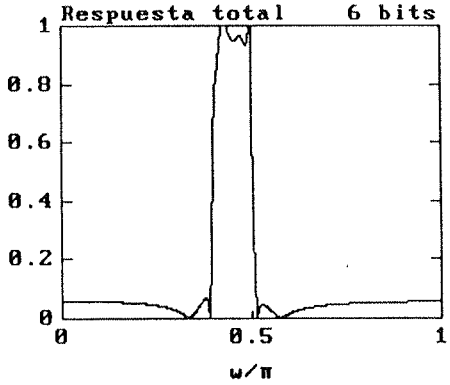

Sección4
5
4
3
2
1
0
0
0.5
1
w/π

Con estructuras en cascada: menos sensible al número de bits

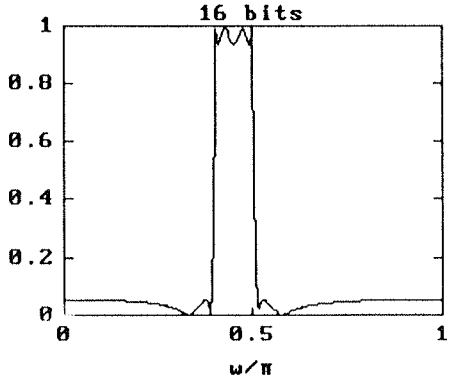


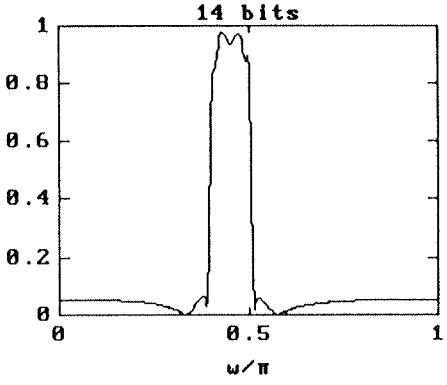


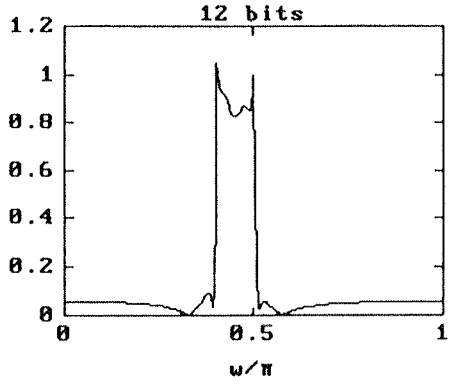


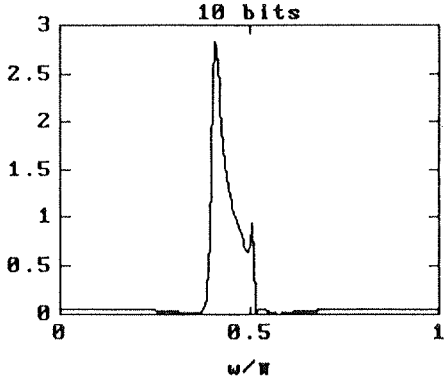


Con estructura directa:

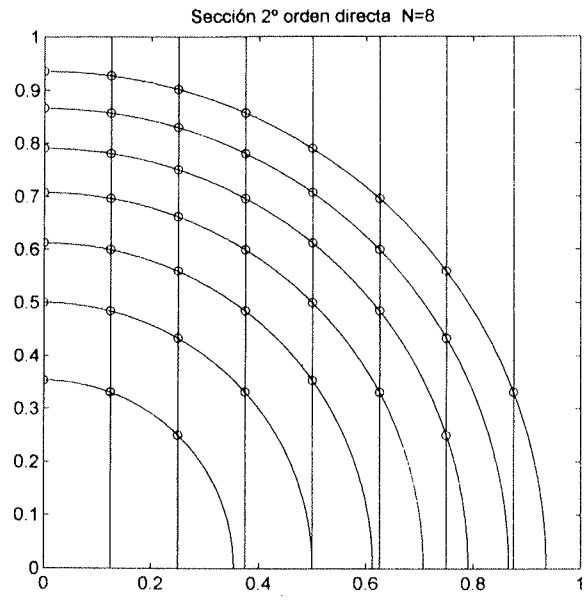


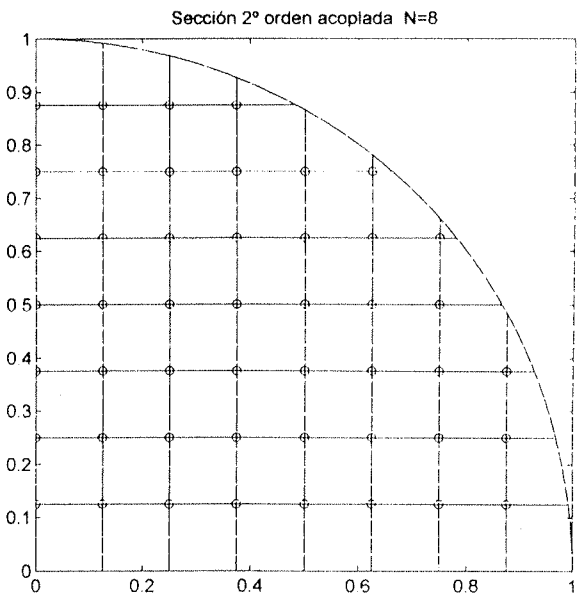


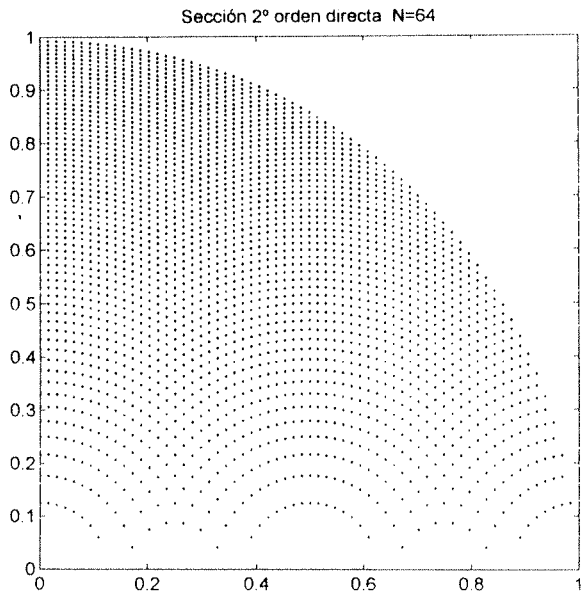


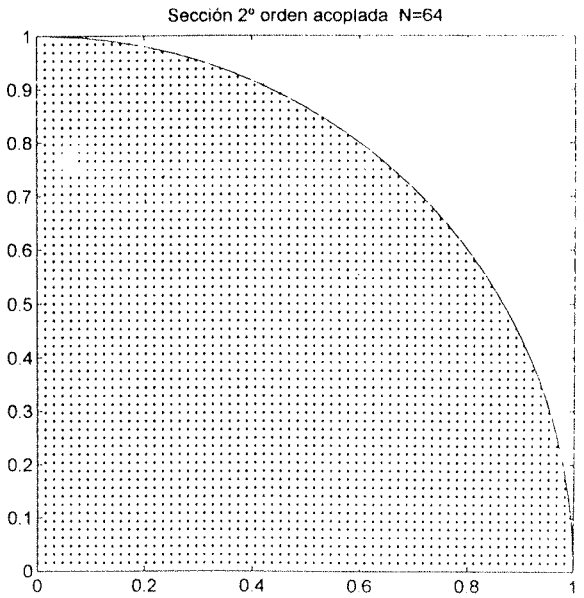


estructura directa vs estructura acoplada









1. REPRESENTACION DE DATOS

1.1 INFORMACION Y DATOS.

Un programa consiste, esencialmente, de dos partes: la descripción de las *acciones* que realizará el proceso representado y la descripción de los *datos* que serán manipulados por esas acciones.

Las acciones se describen en el programa mediante los *comandos*.

Los datos se describen a través de *declaraciones* y *asignaciones*.

1.2 REPRESENTACION INTERNA DE DATOS

Concepto de bit: Existe una relación muy estrecha entre el concepto de BIT (Binary Digit) y los estados que puede asumir un circuito eléctrico en un determinado momento. Aunque es posible que el voltaje de un circuito sea controlado gradualmente ente más de dos valores, se puede decir que es bastante más sencillo si sólo se considera dos estados para éste: activo o con cierto valor de carga e inactivo o sin carga. La corriente eléctrica se ajusta a la misma descripción.

Ahora, si se considera que los componentes de un computador son, en su mayor parte, electrónicos, entonces naturalmente la representación de la información que procesa será más sencilla si se utiliza el sistema binario.

1.3 SISTEMAS NUMERICOS.

Definición: Sea r un entero igual o mayor que 2. Se puede representar cualquier entero decimal no negativo como un "string" de dígitos seleccionados de entre los dígitos 0, 1, ... r-1. Este "string" es la representación en base r del entero decimal.

Ejemplos:

- El sistema binario (r=2) sólo conoce la existencia de dos dígitos: 0 y 1.
- En el sistema decimal (r=10), los enteros se representan por "strings" de dígitos entre {0,... ,9}.
- En el sistema octal (r=8) los enteros se representan por "strings" de dígitos entre {0, ... ,7}.

Supongamos que se tiene un número $(a_1\ a_2\ \ldots\ a_k)$ en base r, donde cada uno de los a_i pertenece a {0, 1, ... ,r-1}. Se puede calcular el entero en base decimal representado por $a_1\ a_2\ \ldots\ a_k$ utilizando el siguiente algoritmo:

```
function evaluacion_polinomial (a1,a2,...,ak:0..r-1):integer;
begin
        numero := 0;
        for i := 1 to  k do numero := numero + ai * r^k-i;
        evaluacion_polinomial := numero;
end;
```

Representación de Fracciones: Se puede usar el "punto decimal" en sistemas con bases diferentes de 10. El valor de la fracción "decimal" $.b_1\ b_2 \ldots b_m$ en base r es

$$b_1 * r^{-1} + b_2 * r^{-2} + \ldots + b_m * r^{-m}$$

en base 10.

Si se tiene un número con parte entera y "decimal", se puede calcular su valor equivalente haciendo en forma separada el proceso.

En los computadores, las bases más usadas son decimal, binaria, octal y hexadecimal.

Palabras de Computador y Representación de Números: Un computador puede ser visto como un dispositivo con una gran cantidad de lugares en los cuales son almacenados secuencias de bits. Estas secuencias son de una longitud fija, y se les llama *palabras de computador*.

La longitudes típicas de palabras de computador están generalmente en el rango 8 a 64 bits.

Por otra parte, es posible interpretar un "string" de 0's y 1's de varias maneras. Comenzaremos considerando la interpretación de "strings" de 0's y 1's como números enteros.

1.3 REPRESENTACION EN PUNTO FIJO.

La representación de números en punto fijo tiene varias formas de tratar el signo de un número. El término punto fijo se refiere al hecho de que el "punto decimal" se puede considerar en una posición fija en una palabra de computador. Si se considera esta posición al extremo derecho, todos los números son enteros, positivos o negativos.

Las representaciones a tratar son:

- notación signo - magnitud
- notación complemento a dos
- notación complemento a uno.

1.4.1 Notación signo - magnitud.

En esta notación, el bit de más a la izquierda en la palabra (bit más significativo [BMS]) representa el signo. Usualmente, 0 denota + (cantidad positiva) y 1 denota - (cantidad negativa). El resto de los bits representan la magnitud.

Ejemplo: Tomemos palabras de computador de 6 bits. Entonces, en notación signo - magnitud, +13 es representado por la cadena 001101 y -13 es representado por 101101. Cero es representado por 000000 y por 100000, pero todos los otros números entre -31 (representado por 111111) y +31 (representado por 011111) tienen representación única.

1.4.2 Notación Complemento a 2.

Supongamos que tenemos palabras de computador de k bits. En notación complemento a 2 los enteros no negativos menores que 2^{k-1} son representados por su equivalente binario, con 0's adelante si es necesario.

Nótese que todos los números no negativos comienzan con 0, puesto que el mayor número representable, $2^{k-1} - 1$, tiene a 0111...1 como representación.

Los números negativos entre -2^{k-1} y -1, son representados agregando primero 2^k y escribiendo luego la representación binaria de la suma.

Nótese que cada suma está entre $+2^{k-1}$ y $+2^k - 1$, así la representación tiene a 1 como el bit de más a la izquierda (BMS).

Nuevamente el bit de más a la izquierda representa el signo, como en la notación signo - magnitud, aunque la interpretación de su valor es diferente.

De hecho, los números no negativos tienen la misma representación en ambas notaciones, pero la representación de los números negativos es diferente.

Ejemplo: Supongamos nuevamente palabras de computador de largo 6. En notación complemento a 2, +13 se representa por 001101. Ahora, para -13, ya que k=6, se tiene que $2^k = 64$; luego: $2^k - 13 = 64 - 13 = 51$. La representación binaria de 51 es 110011, que corresponde a la representación de -13 en notación complemento a 2.

En esta notación, a diferencia de la anterior, el cero tiene sólo una representación, todos los bits en 0. De hecho, todos los números entre -2^{k-1} y $+2^{k-1}- 1$ tienen representación única. Sin embargo, hay pérdida de simetría; -2^{k-1} es representable, pero $+2^{k-1}$ no lo es.

1.4.3 Notación Complemento a 1.

En esta notación, los números no negativos se representan de la misma forma que en las dos notaciones anteriores.

Los números entre $-2^{k-1}+1$ y -1 pueden ser representados agregando 2^k - 1, y luego convirtiendo a un número binario de k bits.

Nuevamente los números negativos se distinguen por tener un 1 en el bit de más a la izquierda, pero ahora cero tiene dos representaciones, todos ceros o todos 1. El rango de números representable es $[-(2^{k-1}- 1), +2^{k-1}- 1]$.

Ejemplo: Suponiendo nuevamente k = 6, -13 es, 2^k - 1 - 13 = 63 - 13 = 50; y convirtiendo 50 a binario, se tiene 110010.

Podría pensarse que la notación signo - magnitud es más "natural" que las otras dos, pero si se considera la aritmética en estas notaciones, el uso de los complementos lleva a ahorro en la circuitería necesaria en un computador.

1.4.4 Aritmética en Punto Fijo

La ley asociativa de la suma establece que

$$(x + y) + z = x + (y + z),$$

sin embargo, en la aritmética computacional no necesariamente ocurre lo mismo. Consideremos palabras de computador de largo 6 bits y los números :

$$x = 011111 = (+31)$$
$$y = 101100 = (-12)$$
$$z = 111000 = (-24)$$

Si sumamos x e y, se tiene 010011 = (+19). Ahora, sumando z se obtiene 100101 = (-5), lo que es correcto.

Sin embargo, si primero sumamos y + z , no tenemos respuesta, ya que la suma es -36, que no tiene representación en 6 bits. Recordemos que para este caso (6 bits) -31 es el menor número representable.

Suma usando notación signo - magnitud

a) Si los signos son iguales, se suman las magnitudes. El signo del resultado es el signo de los sumandos, y la magnitud es la suma de las magnitudes. Sin embargo, si el resultado es demasiado grande para encajar en el espacio asignado, no se obtiene respuesta y ocurre lo que se conoce como *overflow*.

b) Si los signos son diferentes, se resta la magnitud menor a la mayor. La diferencia es la magnitud del resultado. El signo del resultado es el signo de la cifra con magnitud mayor. Sin embargo, si las magnitudes son iguales, no está claro cual signo debe elegirse (es decir, que representación usar para cero). Una buena elección es hacer que 000...0 represente a cero.

Suma usando notación complemento a 2

La suma aquí se puede tratar como si fuera el caso de enteros sin signo. Si existe "acarreo" más allá del dígito más significativo, no debe ser considerado en el resultado y además debe existir acarreo hacia el bit más significativo. Si no hay acarreo más allá del dígito más significativo, tampoco debe haberlo hacia el dígito más significativo. Si no se cumplen estas condiciones, la respuesta es incorrecta.

Ejemplo: Una vez más consideremos palabras de largo 6.

```
  1 1 1 0 1 0     ( -6)
+ 0 1 0 0 0 1     (+17)
-----------------
  1 0 0 1 0 1 1   (+11)
```

Resultado que es correcto. Sumemos ahora (+15) y (+19)

```
  0 0 1 1 1 1
+ 0 1 0 0 1 1
-------------
  0 1 0 0 0 1 0
```

En este caso, hay acarreo desde el 5º al 6º bit, pero no existe acarreo desde el 6º, de modo que la respuesta está errada. Esto es, 100010 no representa (34) que, de hecho, no tiene representación en 6 bits. Se ha detectado un overflow.

Suma usando notación complemento a 1

Aquí nuevamente podemos ignorar el signo al realizar la adición. La primera regla de acarreo de complemento a 2 también se aplican aquí, aunque, si ocurre el acarreo más allá del último bit significativo, no se debe ignorar sino que debe ser agregado al resultado.

Ejemplo:

```
  1 1 0 1 0 1    (-10)
  1 0 1 0 1 0    (-21)
-------------
1 0 1 1 1 1 1
```

esto lleva a:

```
    0 1 1 1 1 1
+             1
-------------
  1 0 0 0 0 0    (-31)
```

Nótese que en este caso el acarreo en el lugar más significativo no ocurre hasta el paso final.

1.5 REPRESENTACION EN PUNTO FLOTANTE.

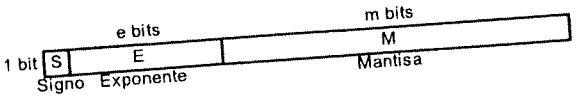
Existe una segunda forma de interpretar las palabras de computador como números. Este método se conoce como punto flotante y es similar a lo que se entiende por notación científica. La ventaja de la notación en punto flotante es que las palabras de k bits pueden usarse para representar números mucho mayores que 2^{k-1}. La desventaja es que no se obtienen k-1 bits significativos en una palabra.

En esta notación un bit de la palabra es reservado para el signo, algunos bits, generalmente pocos, se reservan para el exponente y los bits restantes forman la mantisa.

Supongamos que los bits del exponente representan al entero E y los bits de la mantisa representan la fracción M, $0 <= M < 1$.

En cierto esquema, el valor de la palabra es $+2^E * M$ ó $-2^E * M$, dependiendo del bit de signo.

El formato general de una palabra de computador tratada como número en punto flotante es:



Se considera que los m bits de la mantisa representan una fracción con el punto decimal al extremo izquierdo. Así, M es siempre un número entre 0 y 1, excluyendo el 1.

La interpretación usual de E, el valor del exponente, es más "exótica". E se codifica en "exceso 2^{e-1}" o en complemento a 2..

En general, se le puede dar a los enteros un código en exceso X, agregando (sumando) X y convirtiéndolo a binario. En el caso particular de un exponente de e bits, se suma 2^{e-1} a E.

Puesto que e bits pueden representar los enteros entre cero y 2^{e}-1, se puede representar valores de E entre -2^{e-1} y 2^{e-1}-1.

Ejemplo: Considérese palabras de 12 bits, con e=4 y m=7.

$$\underset{S}{0}\ \underbrace{1\,0\,1\,1}_{E}\ \underbrace{1\,1\,0\,0\,0\,0\,0}_{M}$$

Como e=4 y 2^{e-1} = 8, se debe interpretar 1 0 1 1 (el exponente) en código exceso 8. Restando 8 a 11 se obtiene que E=3.

Se puede evaluar M calculando (.1 1 0 0 0 0 0). Así, M=0.75

Luego, el número representado es $+2^{3} * 0.75 = 6$.

Ejemplo: Convirtamos (1.79) en base 10, a un número en punto flotante en el formato anterior, esto es, e=4 m=7.

En realidad, no hay posibilidad de representar exactamente 1.79 en una palabra de 12 bits con ese formato, por lo tanto, debemos tratar de aproximar lo más exactamente posible.

La estrategia es tener siete dígitos significativos en la mantisa (tantos como sea posible). Esto se puede lograr sólo si hacemos el primer bit igual a 1, esto es, escogiendo $0.5 <= M < 1$.

Es fácil ver que la única manera en que $2^{E} * M$ sea 1.79, con M entre 0.5 y 1 (estrictamente menor que 1) y E un entero, es que E=1 y M=0.895

Luego se elige E=1, que codificado en exceso 8 es 1001; la mantisa se aproximará a 0.895 tanto como se pueda.

$$(0.895)_{10} = (0.111001010001111)_{2}$$

Este número redondeado a siete lugares es (.1110011). Luego, una representación aproximada de 1.79 en punto flotante es

$$\underset{S}{0}\ \overbrace{1\,0\,0\,1}^{E}\ \overbrace{1\,1\,1\,0\,0\,1\,1}^{M}$$

Es deseable que en un computador cada número tenga una única representación, tanto en punto fijo como en punto flotante. Por ejemplo, si se tiene una palabra de computador que se sabe que está siendo usada para representar un número en punto flotante, podríamos estar interesados en comparar ese número con cero. Pero cero tiene muchas representaciones en punto flotante, en particular, cualquier palabra de computador en la que todos los bits de la mantisa son ceros. Así, la comparación tendría que hacerse con todas las representaciones posibles de cero.

Por esta razón, los sistemas de computación se construyen en forma tal que la aritmética realizada genera siempre una respuesta específica (única) para cada valor, denominándoseles a estos sistemas normalizados.

Se considera que la representación de un número en punto flotante es normalizada si:

1. Si los bits de la mantisa son todos ceros, entonces, todos los bits de la palabra son ceros. Esto garantiza la representación única de cero.

2. Si la mantisa no es cero, entonces, o el bit de más a la izquierda es uno (en la mantisa) o los bits del exponente son todos cero.

O sea, para normalizar un número diferente de cero, se desplaza (shift) la mantisa a la izquierda un lugar, y se substrae 1 del exponente hasta que el primer bit de la mantisa es 1, o el exponente alcanza su mínimo valor (más negativo).

Ejemplo: Sea una palabra de 12 bits, e=4, m=7.

0 0110 0011000

para esta palabra, se tiene E= -2, M= 3/16 por lo que su valor es $2^{-2} * 3/16 = 3/64$

Primer desplazamiento:

0 0101 0110000

Segundo desplazamiento:

0 0100 1100000

el valor para esta palabra es, siendo E= -4 y M= 3/4, $2^{-4} * 3/4 = 3/64$

Cuando un número de punto flotante es almacenado en una palabra de computador, podemos hablar de *precisión simple*.

Cuando la aritmética de precisión simple no entrega la suficiente exactitud (precisión) en los resultados, para una aplicación dada, la precisión puede ser aumentada mediante técnicas adecuadas de programación que permiten usar dos o más palabras de memoria para representar cada número.

Suma en punto flotante (precisión simple)

Sumemos $X1 = (signo)\ 2^{E1} * M1$ y $X2 = (signo)\ 2^{E2} * M2$

El primer paso es alinear las mantisas, desplazando la mantisa correspondiente a min(E1, E2) suficientes lugares a la derecha para hacer los exponentes iguales.

Supongamos que E1 >= E2.

i) Desplazar la mantisa de X2, E1-E2 lugares a la derecha, agregando ceros desde la izquierda y reemplazar E2 por E1 en X2 (los últimos E1-E2 lugares de la mantisa se pierden).

ii) Si los signos de X1 y X2 son iguales, sumar M1 y M2 para obtener M3. Si los signos son diferentes, restar el menor del mayor, llamando al resultado M3.

iii) Si hay m+1 bits en M3, desplazarla un lugar a la derecha y agregar 1 al exponente (si E1 es inicialmente 1 1 ... 1, el resultado es overflow).

iv) La respuesta tiene el signo del X de mayor magnitud. En este punto la normalización es opcional, pero recomendable.

1.6 REPRESENTACION DE CARACTERES.

Se ha visto como una palabra de computador puede interpretarse de varias maneras, en punto fijo (en varios formatos) o en punto flotante. Estas representaciones son esencialmente

numéricas. Existe también la posibilidad de que el contenido de las palabras de un computador represente datos no numéricos, en particular, "strings" de caracteres.

Asociado con cada computador existe un conjunto de caracteres que se puede representar internamente en el computador. La representación interna consiste de una secuencia de bits, normalmente seis a nueve. El número de bits usados para representar un caracter es llamado *longitud de byte* del computador.

Ejemplo: Existen varios métodos de codificación para representar caracteres. En general, no codifican los mismos conjuntos de caracteres, aunque ciertos caracteres son comunes a todos ellos.

	EBCDIC	ASCII	BCD
A	11000001	1000001	010001
B	11000010	1000010	010010
		• • •	
0	11110000	0110000	000000

BCD : Binary Coded Decimal

ASCII : American Standard Code for Information Interchange

EBCDIC : Extended BCD Interchange Code

1.7. TIPOS DE DATO

Un *tipo de dato* define el conjunto de valores que una variable puede asumir, las operaciones y relaciones asociados a dichos valores. Cada variable que aparece en un programa debe pertenecer un único tipo de dato.

Las relaciones entregan siempre valores correspondientes al tipo de dato lógico o booleano (verdadero o falso).

Los tipos de datos primitivos disponibles en la mayoría de los lenguajes son:

- los números enteros
- los números reales
- los valores lógicos o booleanos
- los caracteres
- los "strings" o cadenas de caracteres

1.7.1 Tipos de Dato Primitivos

i) Tipo Booleano.

Conjunto de valores: { true, false }

operaciones	**operador**
negación	not
disyunción	or
conjunción	and

ii) Tipo Entero.

El conjunto de valores del tipo entero es un subconjunto de los enteros (Z/), definido por la implementación del lenguaje.

operaciones	**operador**
producto	*
división	div
módulo	mod
adición	+
substracción	-

iii) Tipo Real.

El conjunto de valores del tipo real es un subconjunto de los reales (IR), definido por la implementación del lenguaje.

operaciones	**operador**
producto	*
división	/
adición	+
substracción	-