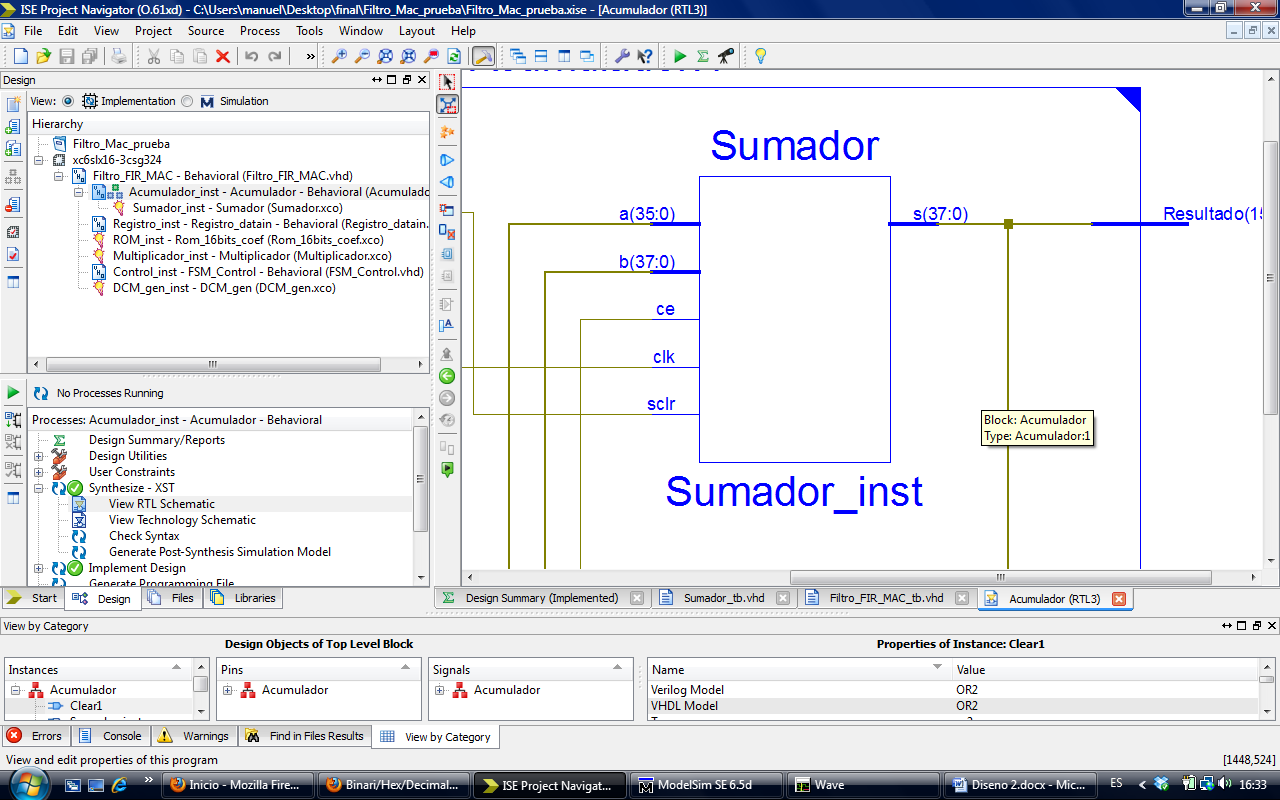
**Funcionamiento del Sumador**

Para implementar la suma del bloque acumulador hemos utilizado un IP-Core de Xilinx ya que pensamos que sería la forma más rápida de hacer la suma (más que utilizar la librería y el operando '+'.

Así pues hemos creado un sumador en C-2 de dos números:

\* El esquemático es:

- "a": 36 bits => será la salida de multiplicador

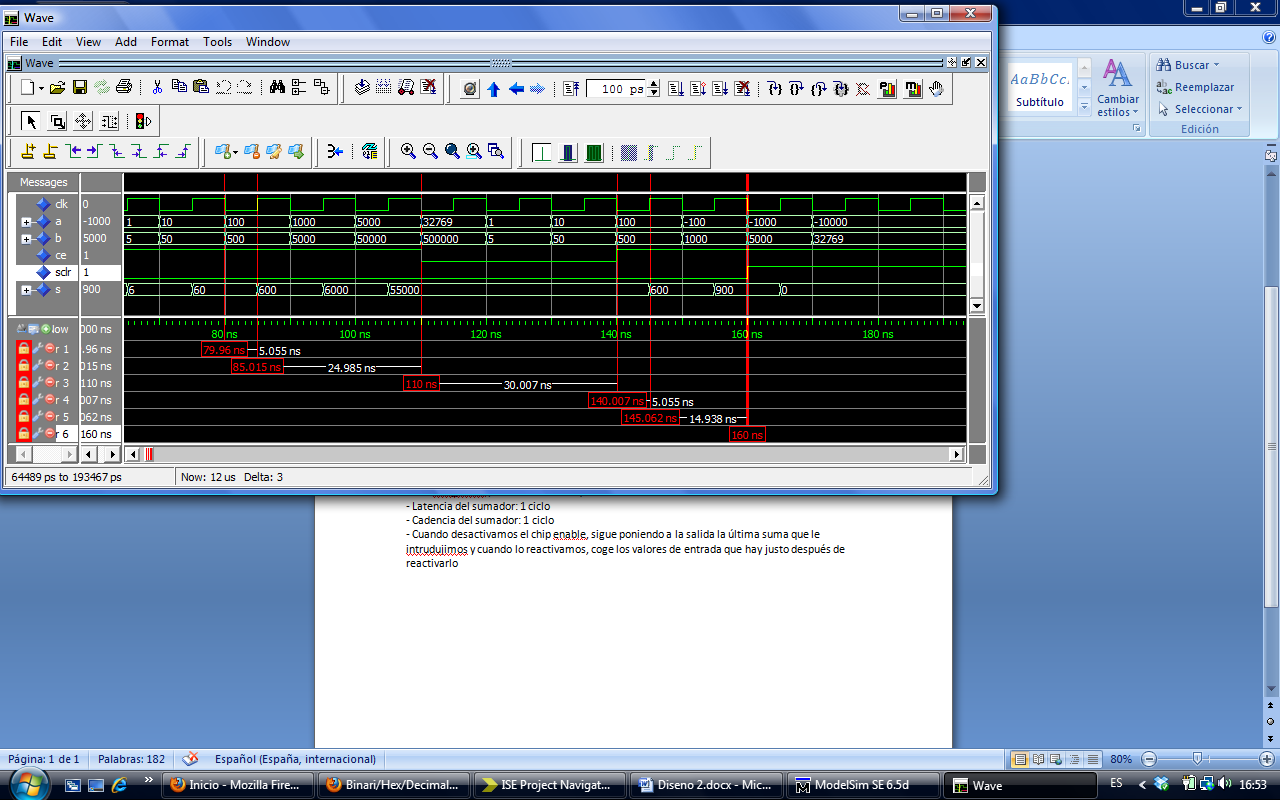
-"b": 38 bits = > será el valor donde se guarden las sucesivas sumas del acumulador

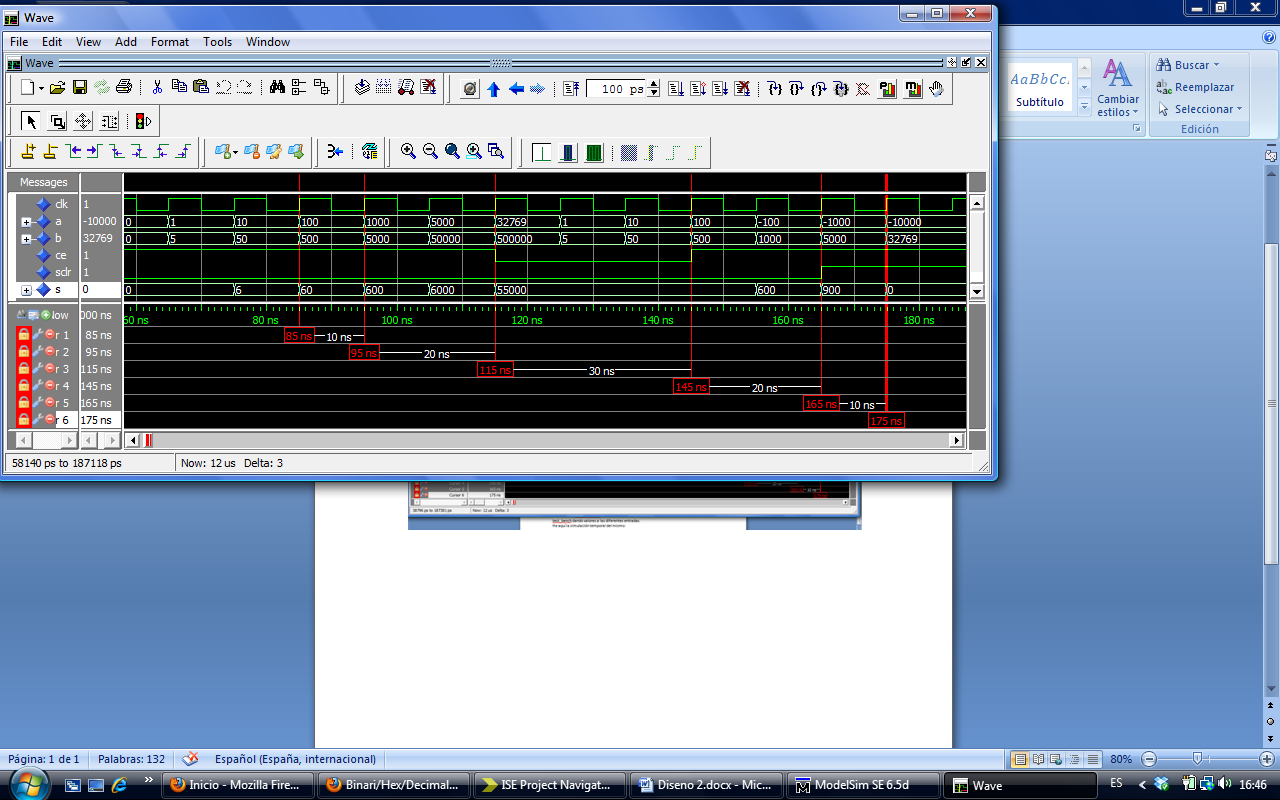
- "s" : 38bits = > será la suma total e irá realimentado a "b" dentro del acumulador

Como vemos tiene entrada de Chip\_Enable y Clear síncrono

\* Para ver su funcionamiento, así como los tiempos de latencia y cadencia hemos dispuesto un test\_bench dando valores a las diferentes entradas.

He aquí la simulación temporal del mismo, ponemos una gráfica donde los cambios de la señal de entrada se producen en los flancos de bajada y otra donde los cambios se producen en los flancos de subida:





# Del test\_bench podemos concluir que:

- Los cambios en la señal de salida "s" se realizan siempre en los flancos de subida

- El sumador tarda medio ciclo o menos en generar su salida: Latencia = 1 ciclo

- La Cadencia del sumador es 1 ciclo

- Cuando desactivamos el Chip Enable:

· Cuando hay un flanco de subida la salida no se actualiza y sigue como antes

En el caso de que se desactiva justo en el flanco de subida, vemos que la última suma si que

se realiza

- Cuando reactivamos el Chip Enable, en el siguiente flanco de subida pondrá el valor de la suma correspondiente

(La segunda gráfica será la que realmente ocurrirá en el circuito sincronizado)

(Los códigos del testbench se pueden encontrar en la carpeta de códigos)

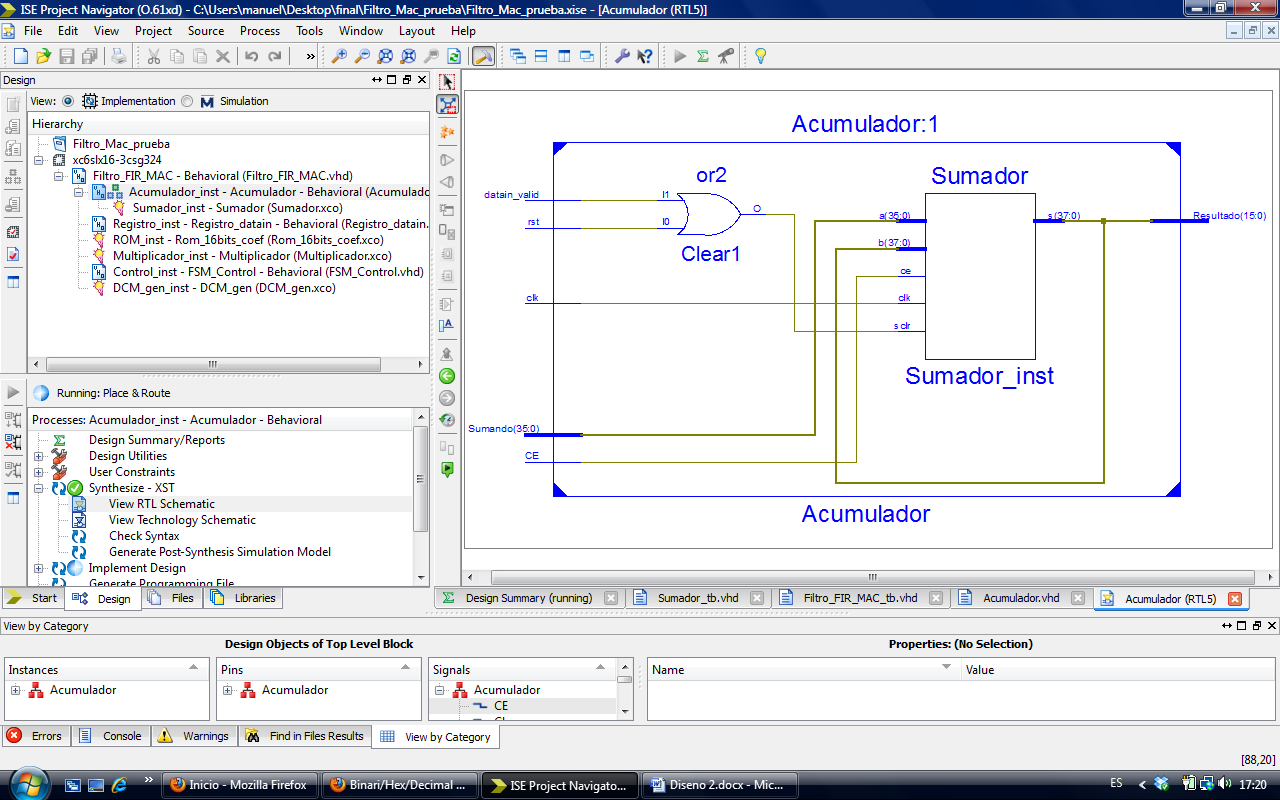
**Funcionamiento del Acumulador**

El circuito acumulador tiene como objetivo:

- Ir sumando y acumulando las diferentes sumas parciales de los productos entre los coeficientes y la señal de entrada (Para ello instanciamos el sumador del IP-Core)

- Redondear el resultado final a 16 bits.

El esquemático es:



# Como podemos apreciar:

- Para hacer la acumulación realimentamos una de las entradas con la salida.

- Como cada vez que nos llegue un dato nuevo tendremos que calcular todos los valores otra vez, tendremos que borrar la anterior salida del acumulador, para ello hacemos que el Clear sea igual a una puerta "OR" entre el "rst" y la señal de nuevo dato

- Para el redondeo del resultado de 16 bits hacemos un truncamiento y cogemos los bits que van del 30 al 15.

**- Explicación del redondeo:**

# Para explicar el redondeo vamos a ver que significado tienen los datos de 16 bits que estamos manejando, el escenario es el siguiente:

- Tenemos unos coeficientes que van de cuantificados en 16 bits en C-2, por lo que el escalón cuántico es: utilizándose el bit de mayor peso como mínimo para el signo.

- Tenemos una señal de entrada "xn" codificada en 16 bits, de los cuales, solo los 8 de menor peso tienen valor y el resto son duplicaciones del bit de signo.

Con esos 8 bits de menor peso podemos codificar desde - 128 a 128 valores enteros de señal

y los otros 8 bits de mayor peso sirven para que no se desborde el dato de salida ya que, en nuestro caso, en el peor de los casos, sumando 28 productos, tenemos un resultado de valores enteros de 8 + E{log2(28)} = 13 bits.

#Ahora bien, usamos 2 datos de 16 bits extendidos a 18 bits para la entrada de nuestro multiplicador, su salida será por tanto de 36 bits

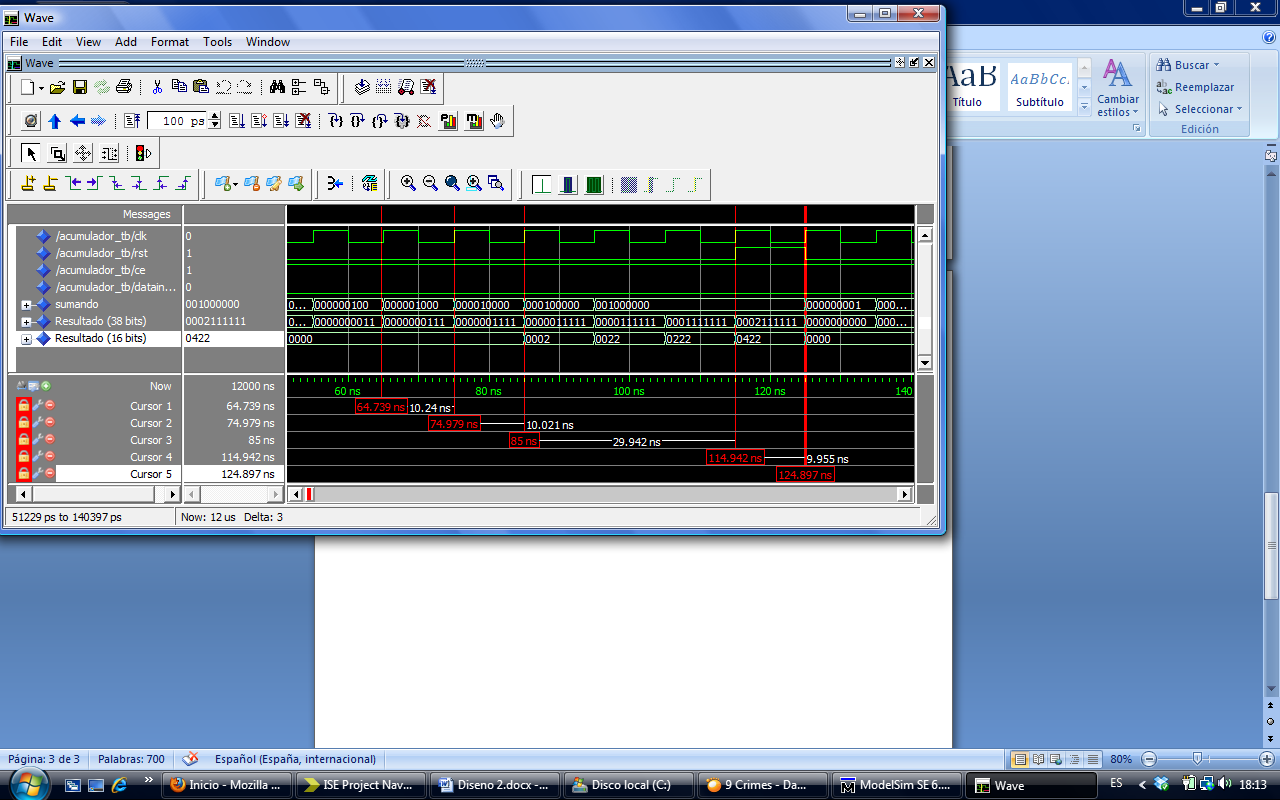
- De esos 36 bits, solo serán significativos los que van del 22 al 15.

Después hacemos 28 sumas de estos productos por lo que el resultado final podría tener como máximo 5 bits significativos más, esto no nos preocupa porque la salida es de 16 bits por que nos siguen sobrando 3.

- Así pues para el redondeo cogemos los 16 bits que van del 30 al 15.

**Simulacion Funcional**

Los valores que de muestran están en formato hexadecimal:



- En esta gráfica podemos ver el funcionamiento del acumulador y su reacción respesto al rst:

- Vemos que la latencia y cadencia de la suma son de 1 ciclo (sumador del IP-CORE)

- Vemos que el Resultado de 38 bits va acumulando todos los valores que le llegan de la entrada, esta es la salida del sumador, pero no la del acumulador.

- El Resultado de 16 bits es el redondeo por truncamiento de los 38 bits, es la verdadera salida del acumulador.

Como hemos dicho, la salida cogerá los bits (30 downto 15) del Resultado de 38 bits.

Se puede ver que cuando Resultado 38 bits es x"11111", sus bits son:

"0001\_0001\_0001\_0001\_0001" siendo el bit numero 16 igual a '1' por lo que el Resultado 16 bits tiene que ser x"2" y es lo que pasa en la gráfica.

El resto de valores pasa lo mismo.

- La respuesta del acumulador frente al Chip\_Enable es la misma que la del Sumador del que está hecho como se puede comprobar en la siguiente gráfica:



Nota:

Ver la siguiente correpondencia entre los valores del Sumando y lo que añade al Resultado 16b

x"000010000" = > x"0002"

x"000100000" = > x"0020"

x"001000000" = > x"0200"

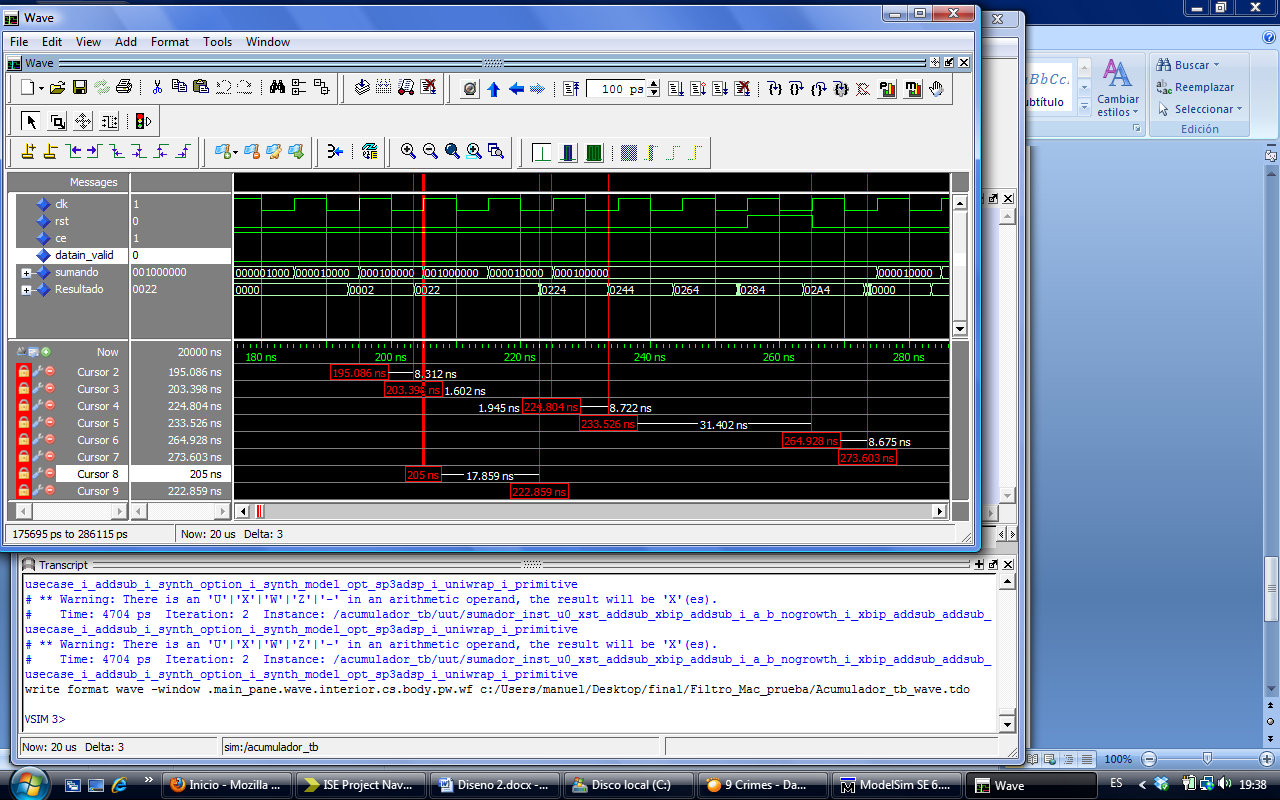
x"010000000" = > x"2000"

**Simulación Temporal**

Para la simulación temporal, una cosa que hemos cambiado en el testbench es el tiempo de rst inicial, que hemos puesto a 100 ns.

Hemos comprobado que poner tiempos menores que este, como por ejemplo 30 ns, hace que el circuito no funcione bien en los primeros ciclos, esto puede ser debido a que el circuito necesita cierto tiempo de inicialización antes de poder funcionar.

He aquí la gráfica, donde no hemos podido poner la señal de Resultado 38 bits.

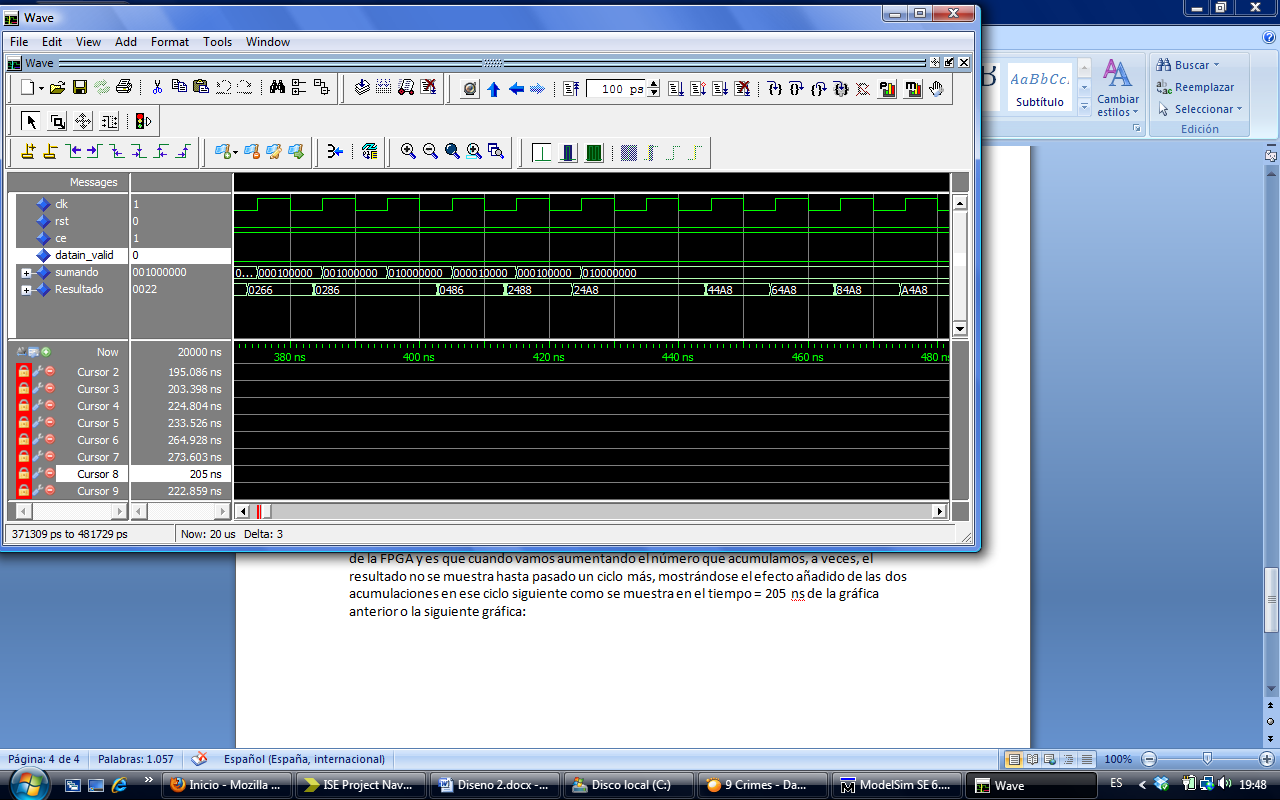


Podemos sacar las siguientes conclusiones:

- Se tarda menos de un ciclo en realizar la acumulación (unos 8,8 ns) = latencia

- La cadencia del acumulador es siempre de 1 ciclo

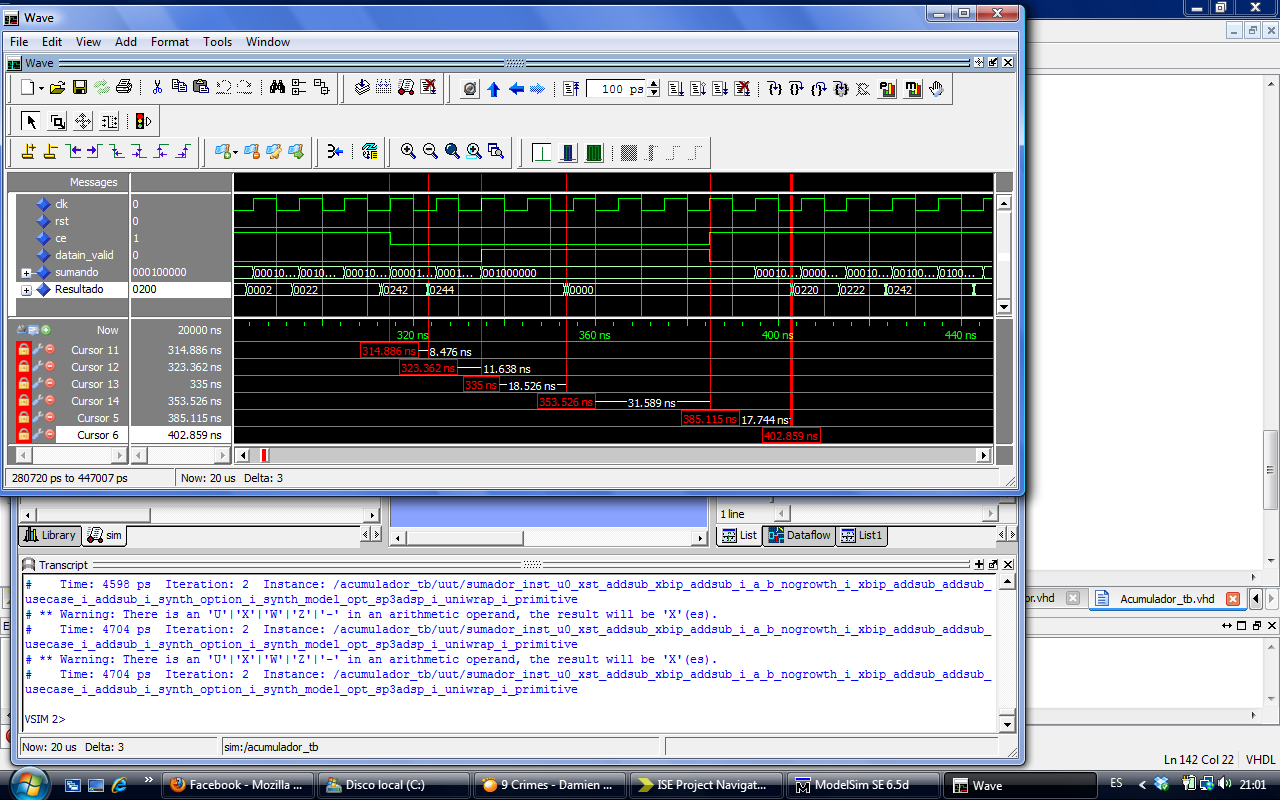
- Ocurre un fenómeno extraño y es que cuando vamos aumentando el número que acumulamos, a veces, el resultado no se muestra hasta pasado un ciclo más del que debería, mostrándose el efecto añadido de las dos acumulaciones en ese ciclo siguiente como se muestra en el tiempo = 205 ns de la gráfica anterior o la siguiente gráfica:



Como vemos aunque pase esto, las acumulaciones al final siguen siendo las correctas por lo que el problema de tiempos debe estar en la selección de los 16 bits (30 al 15)

En principio este problema no tiene que afectar a nuestro circuito, la única consideración a tomar es quizás darle a la salida un ciclo más para estabilizarse, esto se hará retrasando la señal de dataout\_valid.

Por último ver el efecto de la activación y deactivación del Chip\_Enable y datain:



- Vemos que al reactivar el acumulador, el dato que haya cuando de reactiva se acumula (pasa otra vez lo del retraso y se suman las 2 acumulaciones a la vez)

**Funcionamiento del Filtro de orden 27**

Para una primera aproximación para demostrar que nuestro filtro funciona,

lo que tenemos que hacer es:

1- Configuramos el Filtro para orden 27, esto lo hacemos poniendo el genérico M = 27

2- Creamos los 28 coeficientes del Filtro con Matlab y cargamos el ROM.coe, estos son:

ed00,e794,e53d,e6f7,ed63,f8a6,0860,1ba6,311a,4706,5b90,6ce4,796c,7fff,7fff,796c,6ce4,5b90

,4706,311a,1ba6,0860,f8a6,ed63,e6f7,e53d,e794,ed00

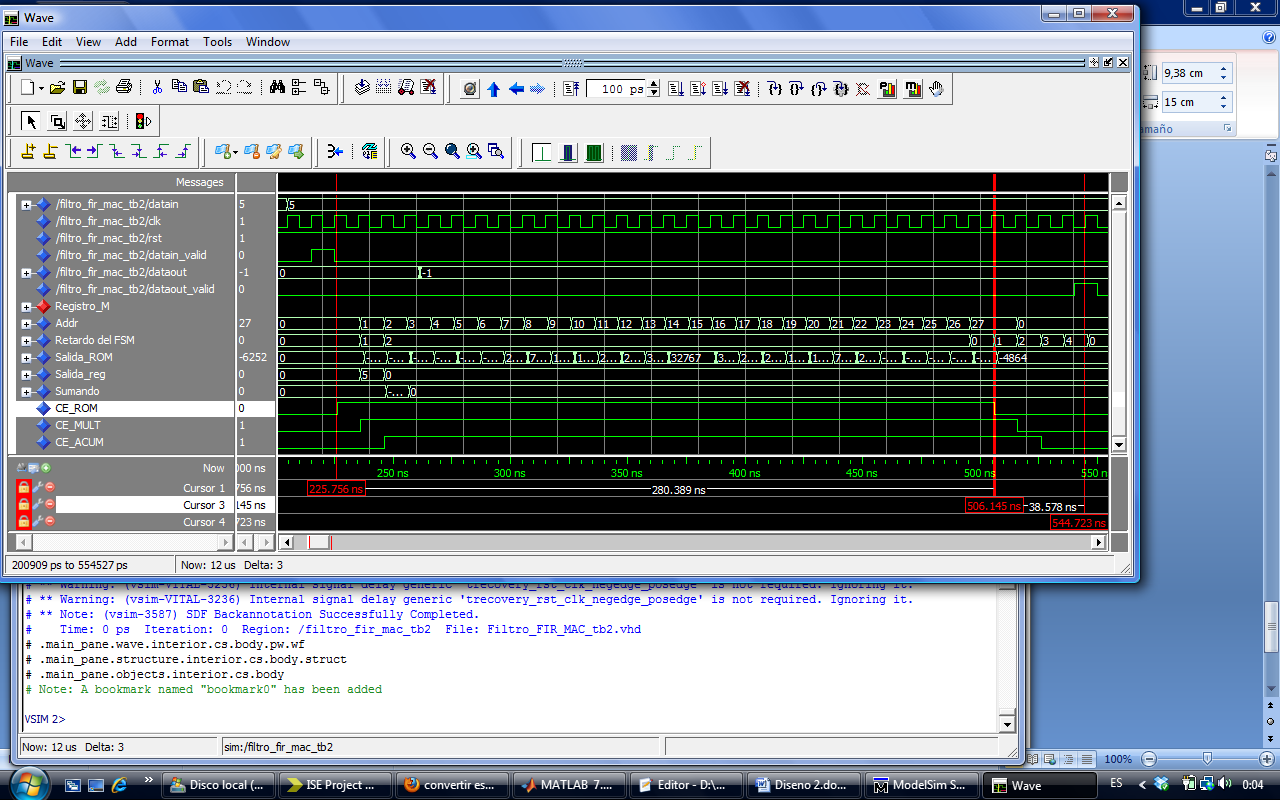
3- Creamos una señal con la que testear el Filtro, esta será:

x = [

5 10 15 20 25 30 25 20 15 10 5 0 -5 -10 -15 -20 -25 -30 -25 -20 -15 -10 -5 0 5 10 15 20 25 30 25 20 15 10 5 0 -5 -10 -15 -20 -25 -30 -25 -20 -15 -10 -5 0 ]

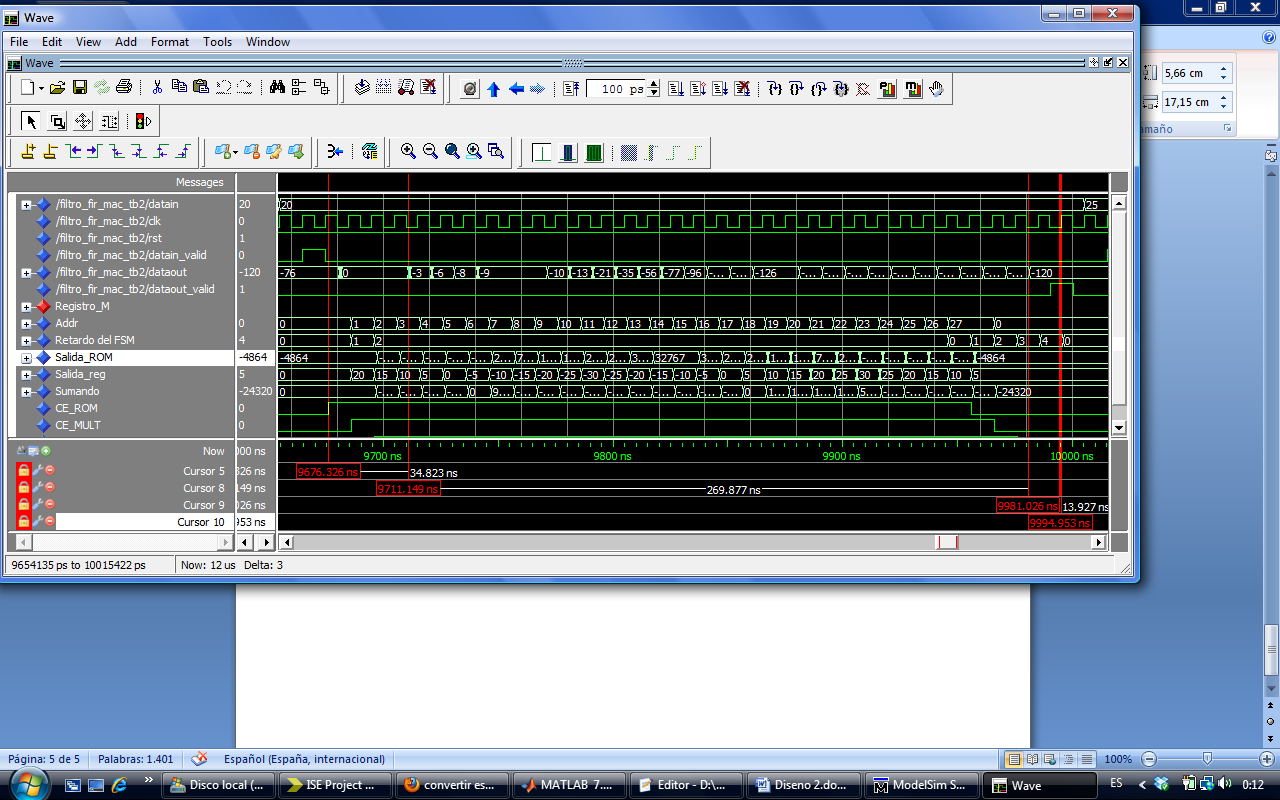
# He aquí la simulación temporal de un trozo del circuito para ver los tiempos y que aparentemente por lo menos hace las operaciones que tiene que hacer.

- Por suerte hemos podido encontrar algunas variables de la simulación temporal que nos ayudarán a comprobar que está bien, he aquí unos cuantos pantallazos que nos muestran aspectos del Filtro:



- Vemos que el circuito ejecuta las 28 multiplicaciones y 28 acumulaciones

- La latencia y cadencia del circuito son 28 + 4 = 32 ciclos



- Nuestro circuito muestra las salidas intermedias, que son los bits (30 downto 15) del acumulador. Vemos que la primera salida intermedia tarda 3,5 ciclos en generarse:

· 1 para poner las salidas de la ROM y el registro

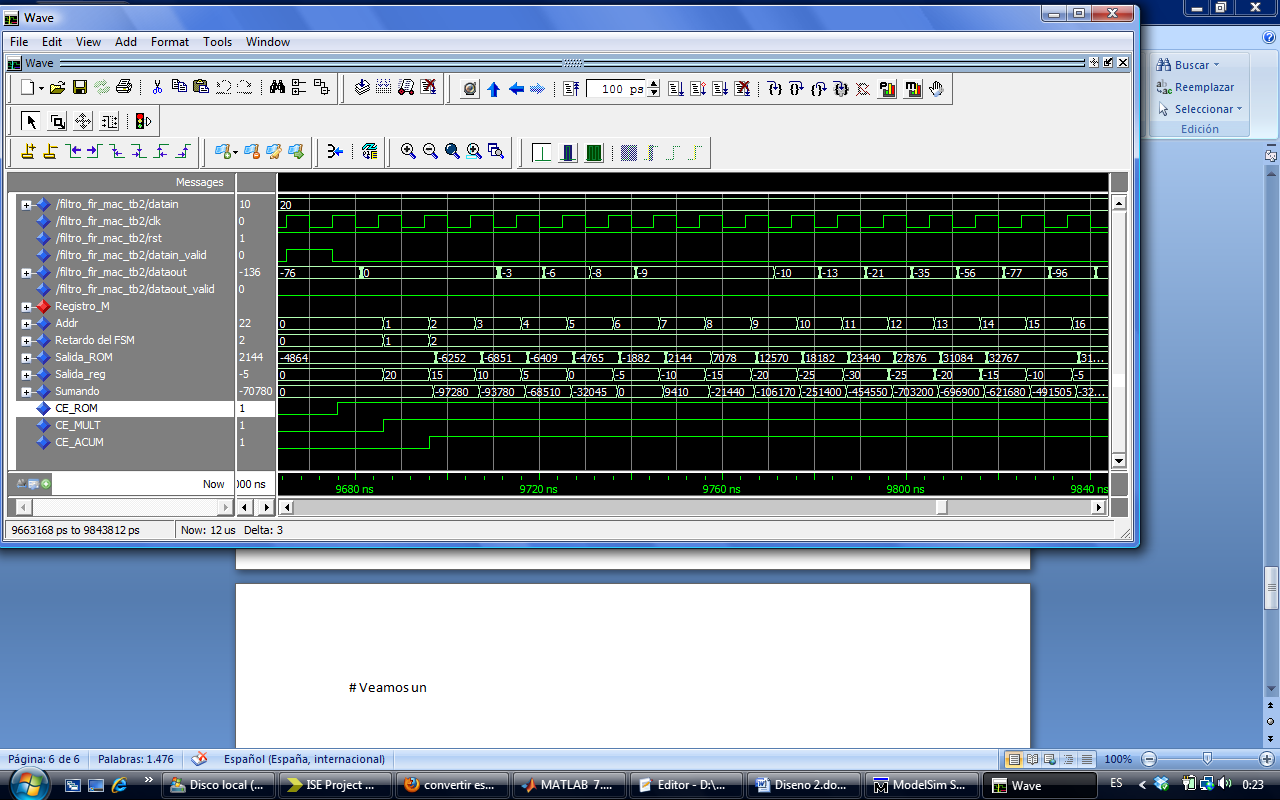
· 1 para la multiplicación

· 1 para la suma

· 0,5 de la desincronización inicial debido a que ahora los biestables funcionan en flanco de bajada

- La última salida tarda 27 ciclos más: 30,5 ciclos

- Ponemos el dataout\_valid un ciclo más tarde por si acaso.

# Veamos una imagen más cerca viendo que se generan los sumandos (salida del multiplicador) correctamente: 

# Así pues ahora comprobemos que la salias del Filtro son las correctas, para ello compararemos el resultado del Filtro con la convolución en Matlab, para ello hay que tomar las siguients consideraciones:

- El redondeo que realiza nuestro filtro es un truncamiento en C-2 lo que se traduce a un redondeo al valor más bajo que en Matlab es la función "Floor(N)"

- Tenemos que calcular los valores en Matlab de la misma manera que se hacen en el Filtro

Para ello usamos el siguiente código:

Sea Num la matriz con los coeficientes que nos genera el filtro:

Num = [

-0.0196 -0.0252 -0.0276 -0.0258 -0.0192 -0.0076 0.0086 0.0285 0.0506 0.0732

0.0944 0.1123 0.1252 0.1320 0.1320 0.1252 0.1123 0.0944 0.0732 0.0506

0.0285 0.0086 -0.0076 -0.0192 -0.0258 -0.0276 -0.0252 -0.0196 ];

b=Num/max(abs(Num));

b=round(b\*2^15);

Salida = conv(b,x)/ 2^15

Salida = floor (Salida)

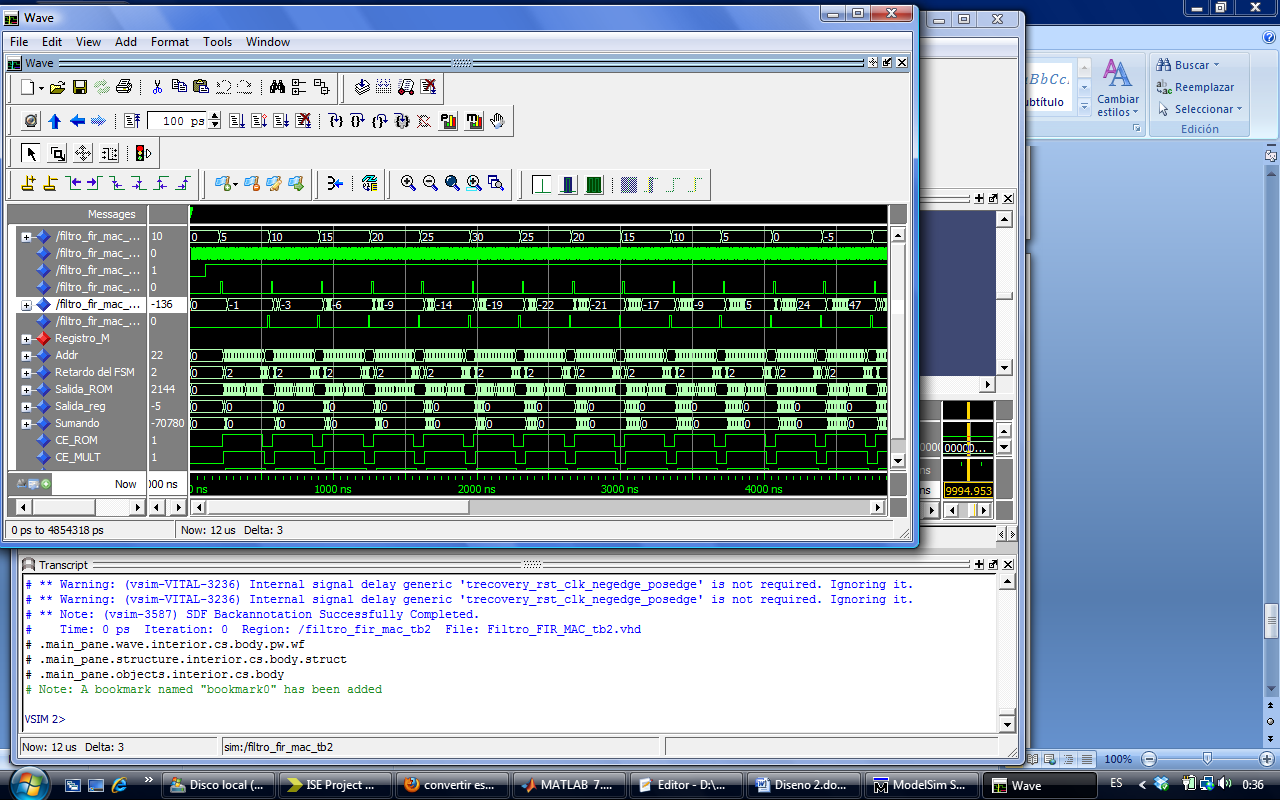
# Tenemos que:

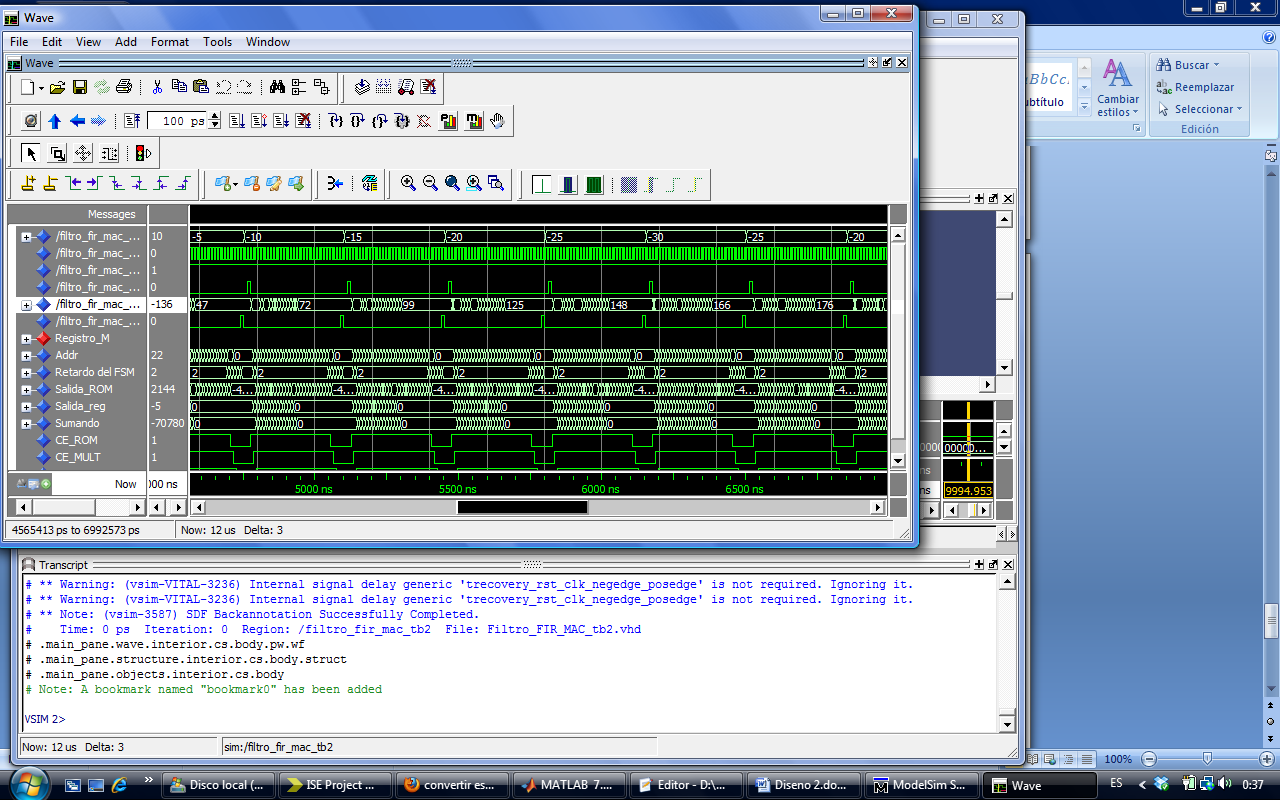
Salida = [

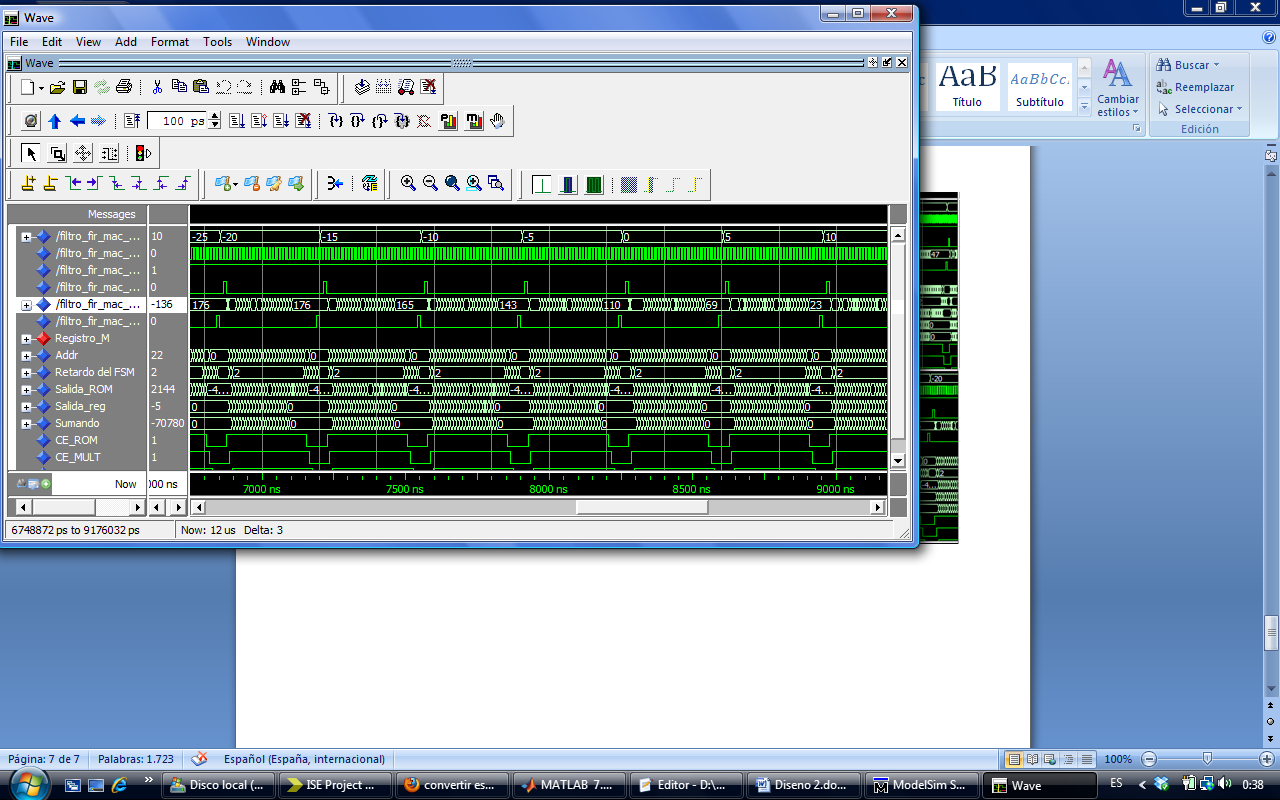
-1 -3 -6 -9 -14 -19 -22 -21 -17 -9 5 24 47 72 99 125 148 166 176 176 165 143 110 69 23 -27 -76 -120 -157 -184 -198 -198 -184 -157 -120 -76 -26 25 75 119 156 183 197 197 183 156 119 75

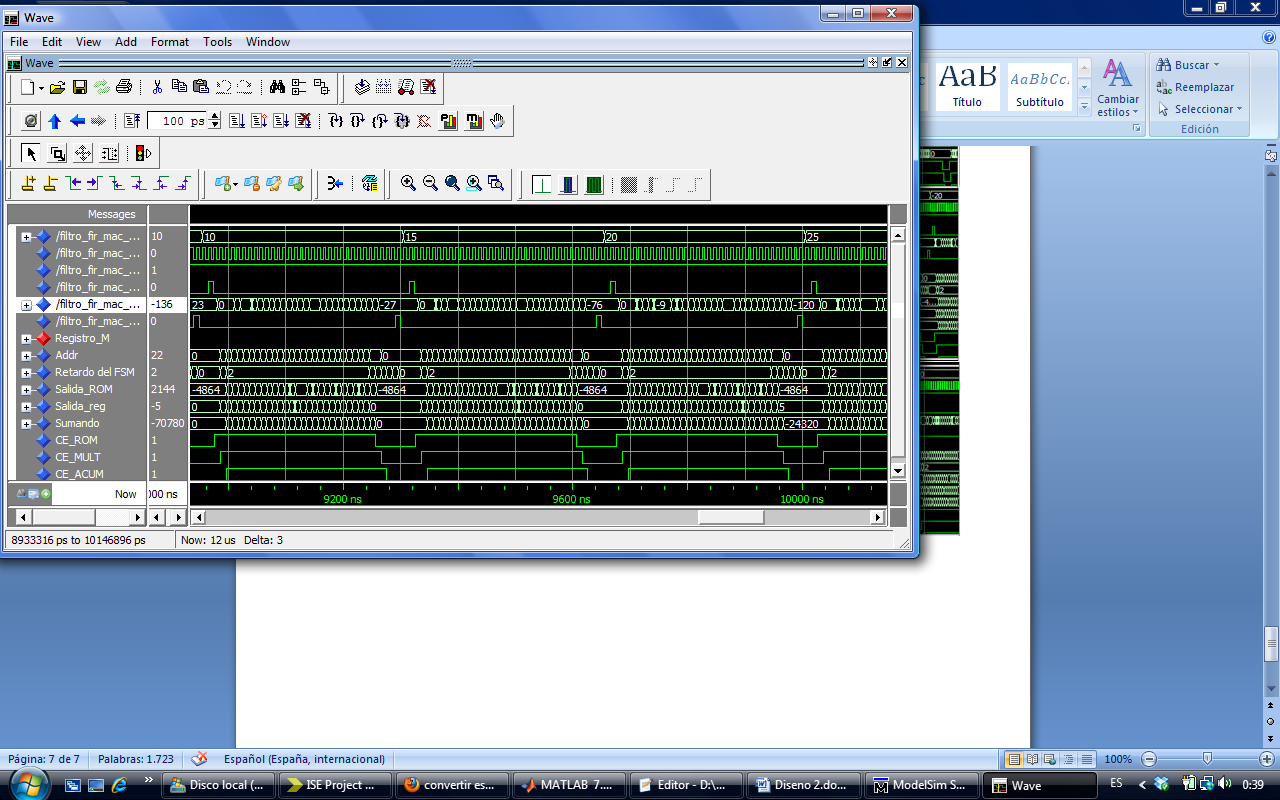
26 -24 -70 -111 -144 -166 -177 -177 -167 -149 -126 -100 -73 -48 -25 -6 8 16 20 21 18 13 8 5 2 0 0 ]

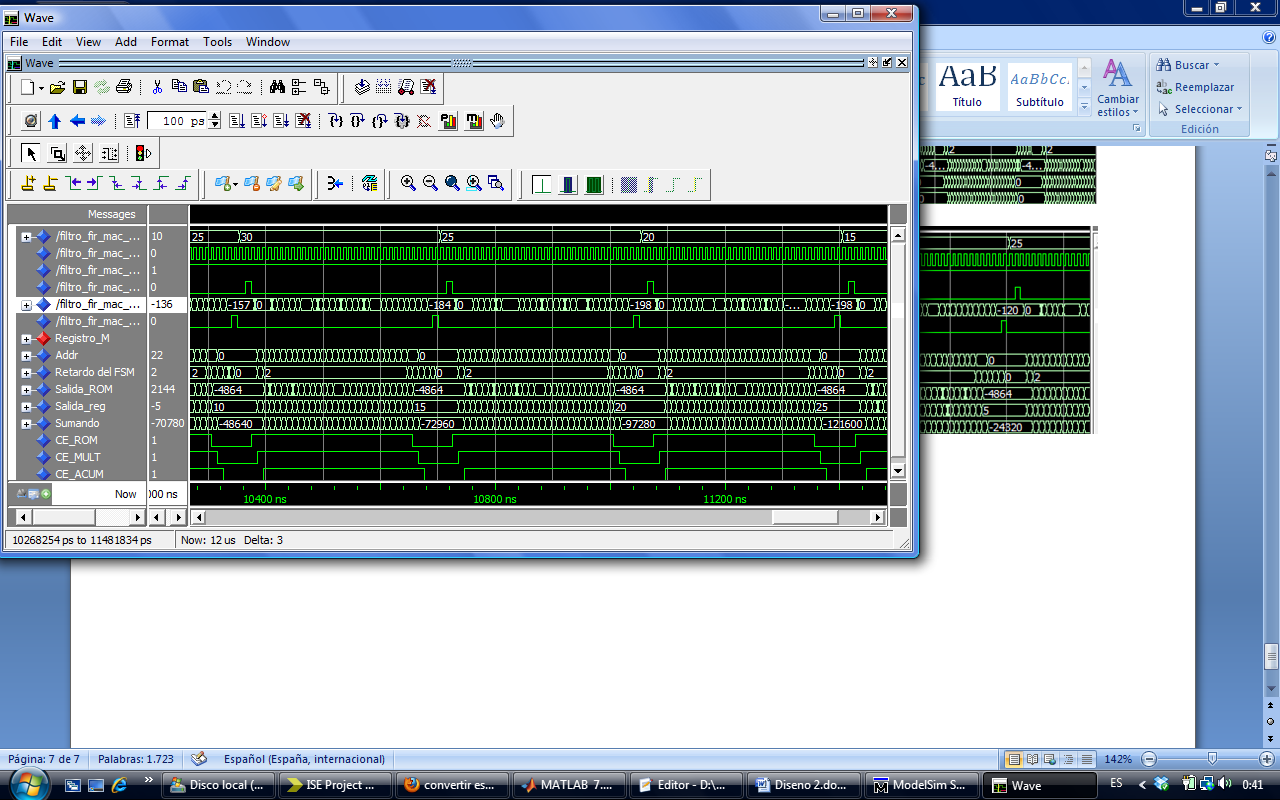
Veamos ahora unos cuantos pantallazos de la respuesta de nuestro filtro:







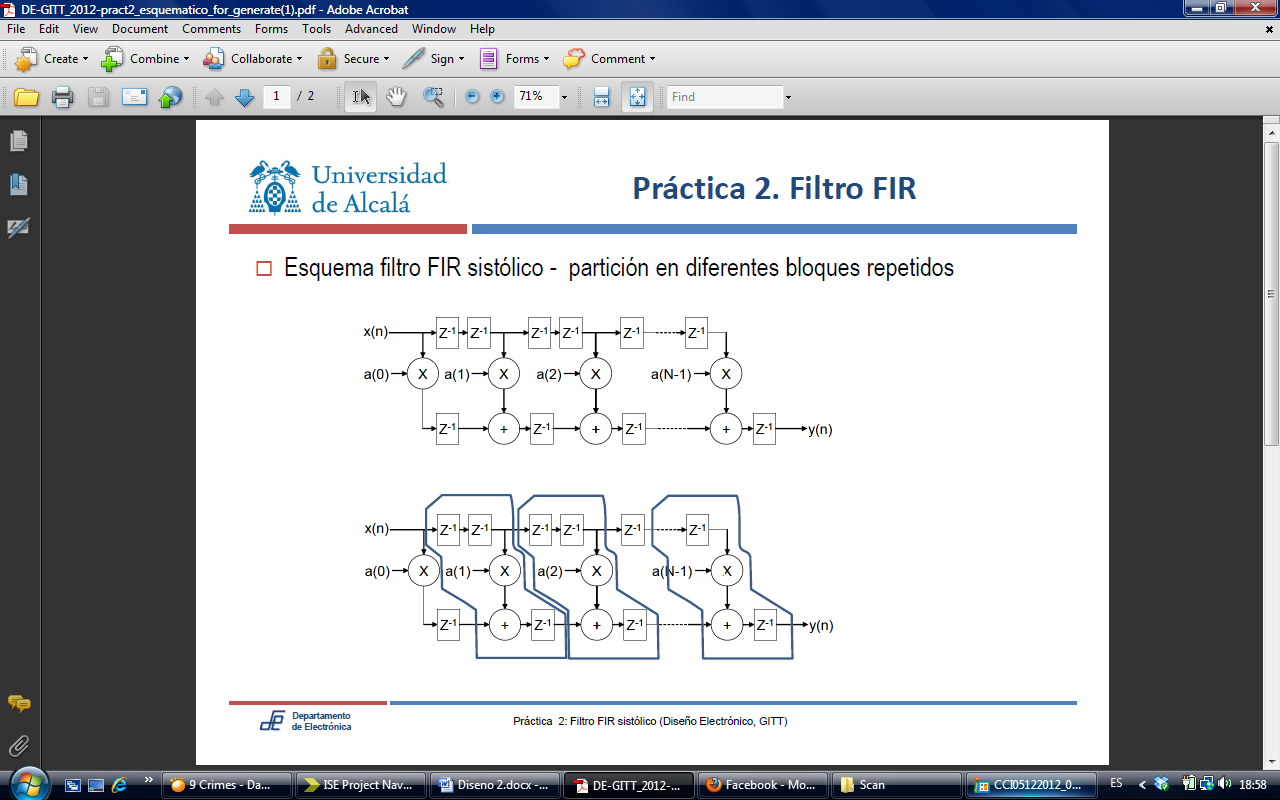




TODOS Y CADA UNO DE LOS VALORES COINCIDEN !!!!

**Funcionamiento del Bloque Propagador**

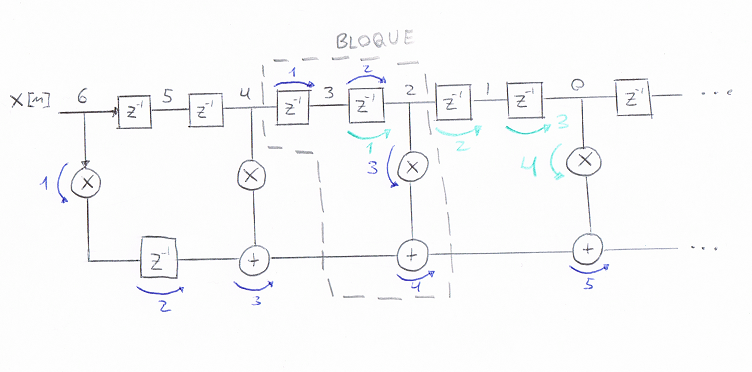
Siendo el circuito que queremos implementar:



- Para implementarlo, identificamos el patrón repetitivo y hacemos una entidad con ello que luego instanciaremos las M = 27 veces necesarias.

El primer multiplicador no obedece el patrón así que lo instanciaremos aparte en Top\_System.

- Primero de todo comprobar que el flujo de datos y de tiempos está bien y correctamente sincronizado, para esta demostración supondremos que la latencia del sumador y el multiplicador es de 1 ciclo, al igual que su cadencia.



- Como podemos ver los tiempos cuadran, pero… y si las latencias de sumador y multiplicador fueran mayores que 1 ciclo?

· Para el caso del multiplicador daría igual ya que los caminos se compensarían, pero afectaría a la latencia del filtro en su conjunto.

· Para el caso del sumador los caminos se descompensan y con ellos la sincronización,

para solucionarlo habría que añadir más registro a la entrada de cada bloque.

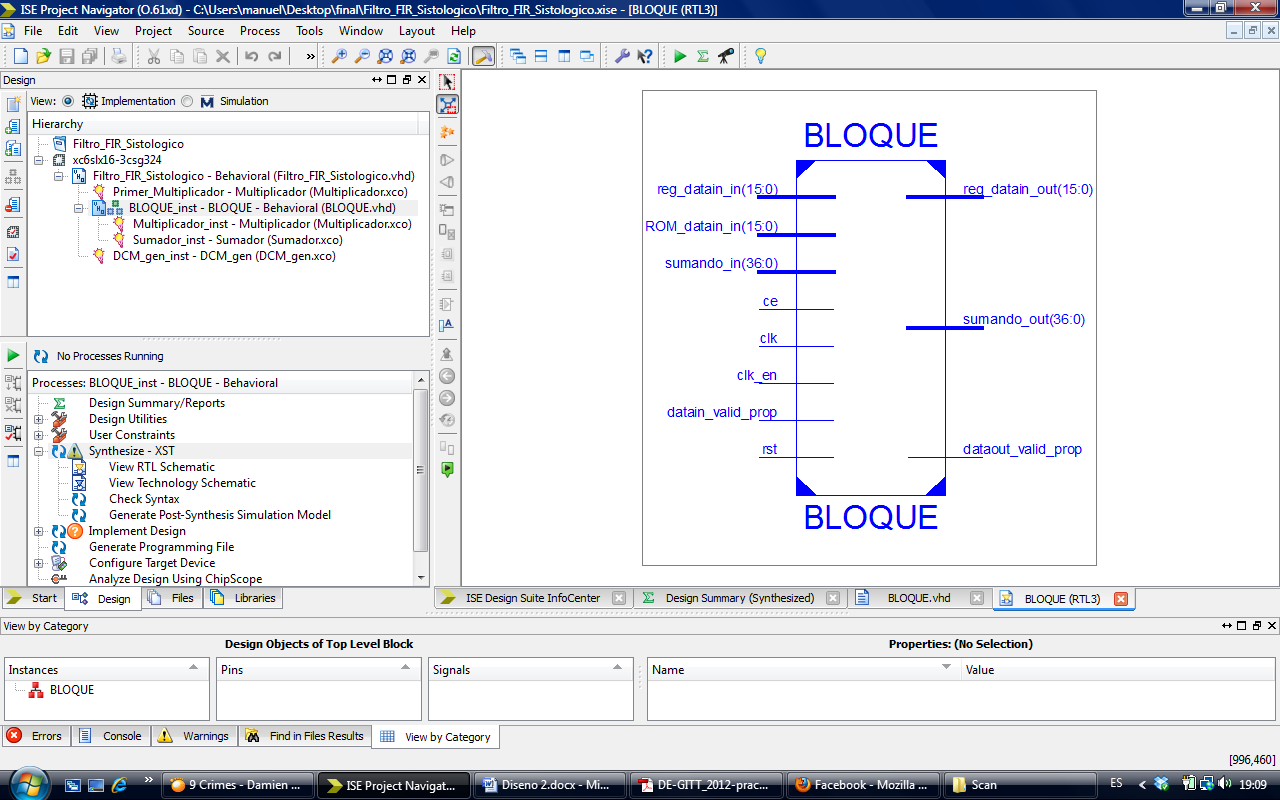
# Así pues nuestro bloque implementa el sumador y el multiplicador y los registros

- El número de registros a la entrada del bloque se implementa dinámicamente mediante el genérico lat\_sum.

- El Bloque dispone la señal de entrada "Clk\_en" que estará conectada directamente a la señal "datain\_valid", esta señal sirve para indicar a los registros del bloque si propagar sus señales o no ya que el circuito solo puede funcionar mientras le metamos nuevas señales de entrada, de lo contrario aparecerían huecos y valores falsos en los registros.

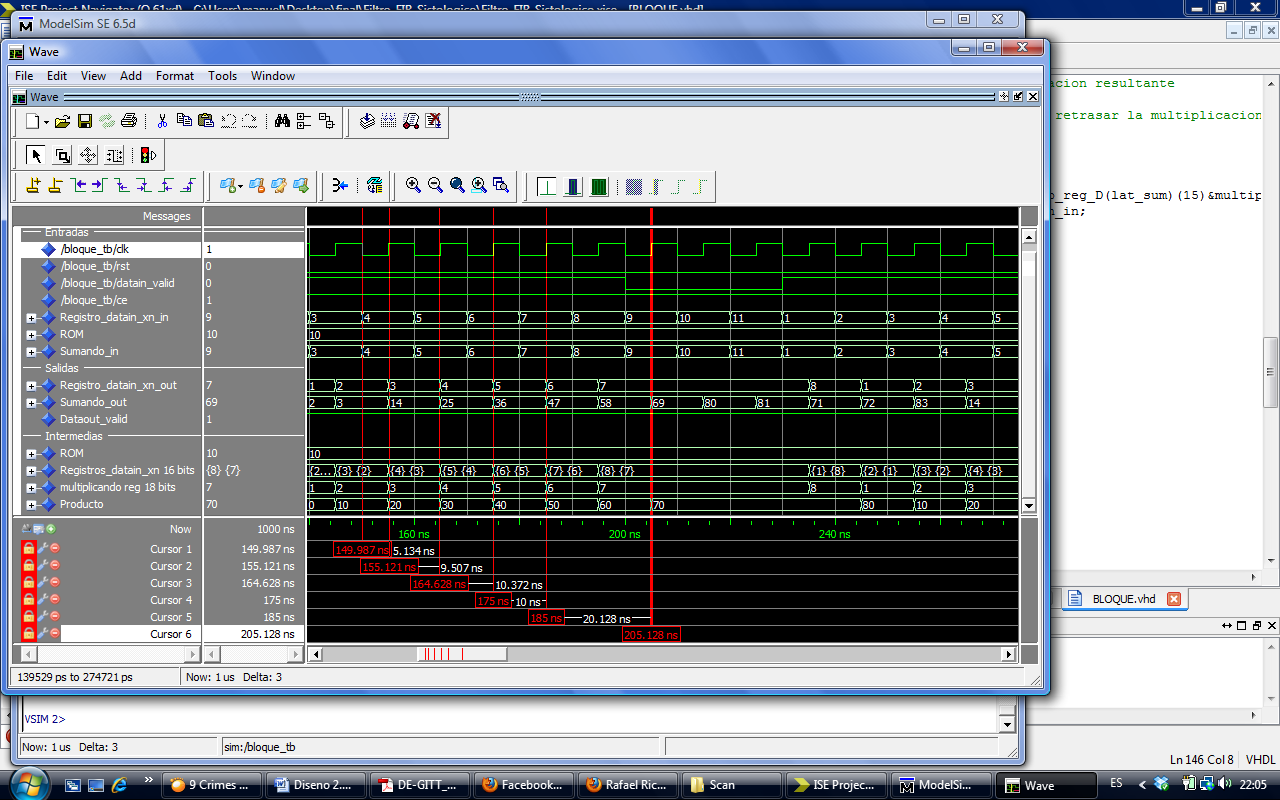
- El Bloque además propaga la señal de "datain\_valid" de Bloque en Bloque para saber cuándo activar la señal "dataout\_valid"

# El esquemático del bloque es:



# Su código así como el del test\_bench vienen explicados instrucción por instrucción en los archivos .vhd dados. Se explica el funcionamiento de la propagación del datain\_valid.

#Así pues, para comprobar si funciona aquí va la simulación funcional:



- Vamos a comentar el viaje del datain\_xn, desde que esta a la entrada hasta que sale sumado con el valor correspondiente de la suma. Los punteros van señalando las etapas en ciclos:

1)Ponemos el 4 a la entrada del bloque

2)Cuando hay un flanco de subida, se pone en el primer registro.

3)Al segundo flanco de subida, se ponea la salida del segundo registro y por ende a la salida del bloque y a la entrada del multiplicador.

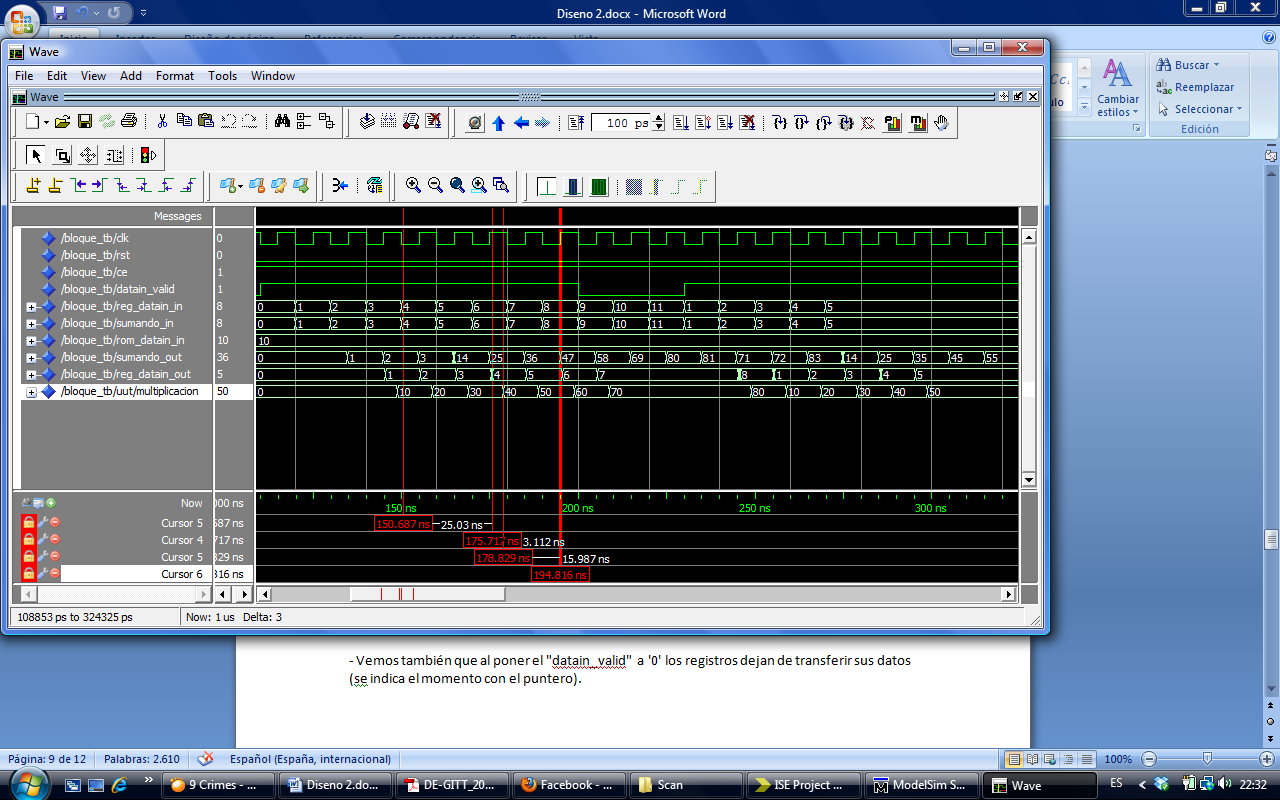
4)Después de otro flanco se multiplica por la ROM, en este caso es 10

5)Por útimo es sumado con el sumando de entrada que hay este último flanco, que en nuestro caso es 7, así pues dando como resultado 47 y cumpliendo los tiempos esperados.

- Vemos también que al poner el "datain\_valid" a '0' los registros dejan de transferir sus datos

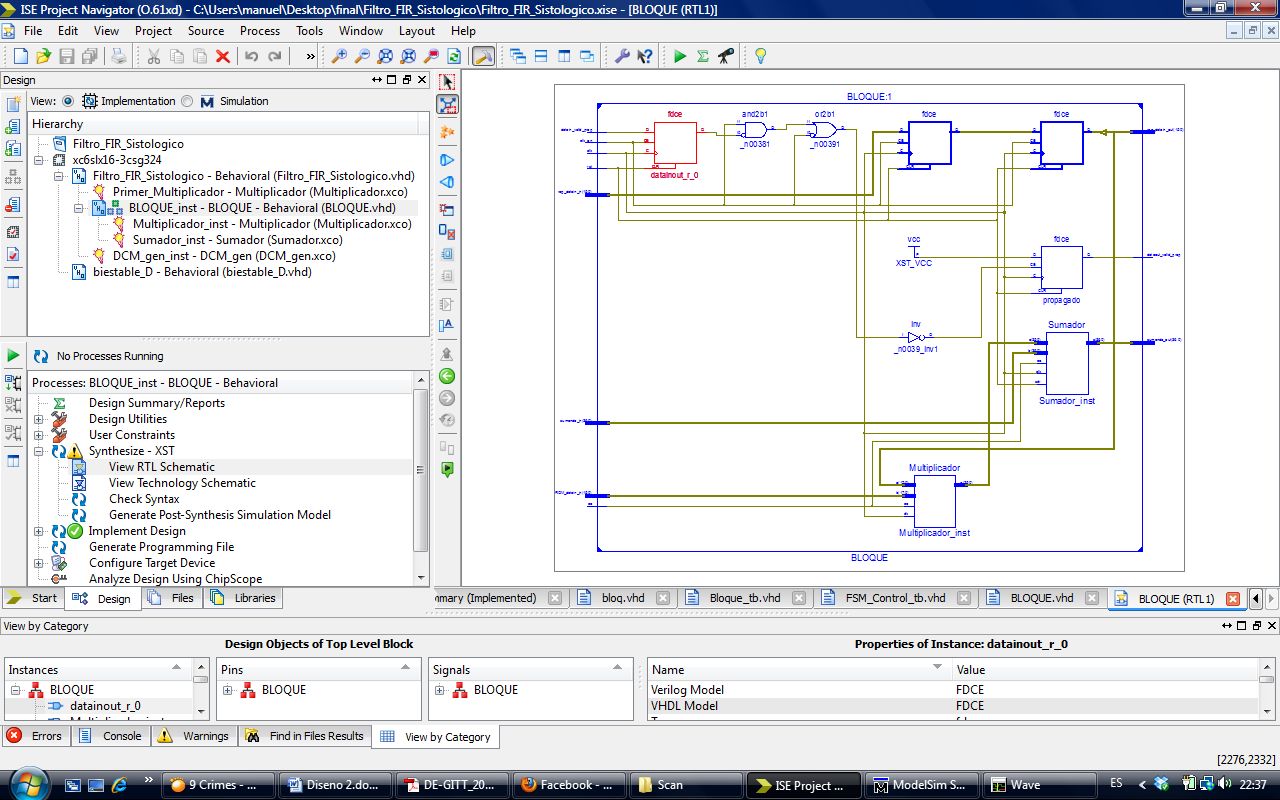
(se indica el momento con el puntero).

**Simulacion temporal:**



# Si hemos hecho bien nuestro trabajo, la simulación temporal debería tener los mismo valores que la funcional, aunque en momentos de tiempo diferentes debido a la no idealidad de la simulación temporal y a las razones ya explicadas.

Por último ponemos el esquemático del bloque:



**Diferencias entre la simulación funcional y temporal de biestables:**

# Existe una diferencia bastante notable entre la simulación funcional y temporal de un biestable, que radica en la siguiente pregunta:

Cuando hay un flanco de reloj, y justo en ese instante la señal de entrada cambia:

¿Qué valores de entrada utilizan los biestables como referencia para actuar?

¿Los que hay justo antes del flanco o los que hay después?

- La respues es que en el funcional coge el valor justo que hay en el flanco, es decir el nuevo valor y en temporal coge el que hay justo antes, es decir, el viejo.

# La implicación más dramática de este hecho recae sobre la sentencia:

**if (clk'event and clk='1') then**

**. . . . . . .**

**. . . . . . .**

que seguramente encontraremos en los biestables que creemos nosotros.

Pues bien, de acuerdo al razonamiento anterior, para la simulación funcional, esto es un flanco de subida y para la temporal es un flanco de bajada.

Por lo que, como poco, la diferencia de tiempos entre la temporal y la funcional cuando usamos registros es de medio ciclo más lo que tarde el biestable en dar su salida, que ya dependerá también del ruteo y demás.

Por lo que en el peor de los casos, cuando el biestable tarda casi un ciclo en actualizar su salida debido al hardware en sí, la diferencia de tiempos entre funcional y temporal puede llegar a ser de casi ciclo y medio.

- Deberemos sincronizar nuestros circuitos de tal manera que esta diferencia temporal respecto a lo que pensábamos que iva a ser al diseñar el código no afecte al funcionamiento del mismo.

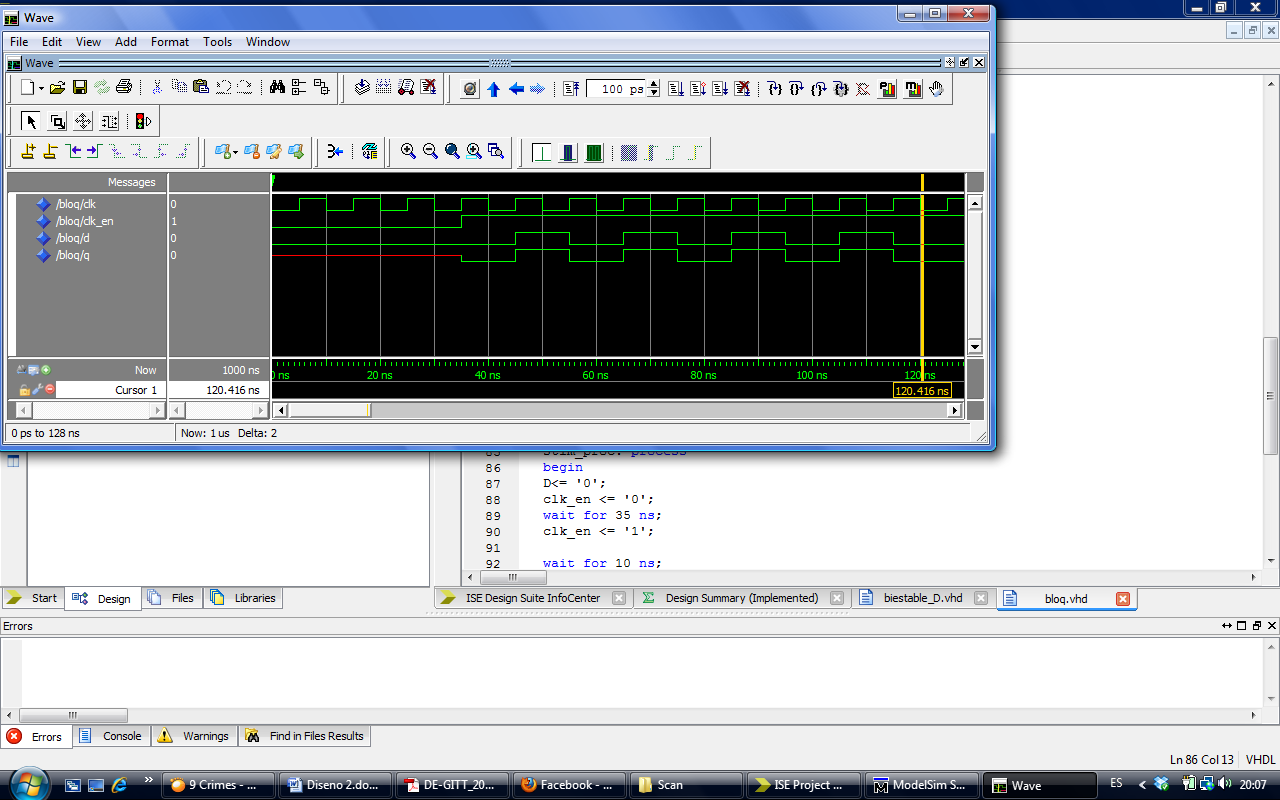
- Normalmente esta difencia simplemente añadirá una cierta latencia a nuestro circuito, pero la cadencia seguirá siendo la misma.

- Hay que tenerlo en cuenta si trabajaremos también con circuitos combinacionales .

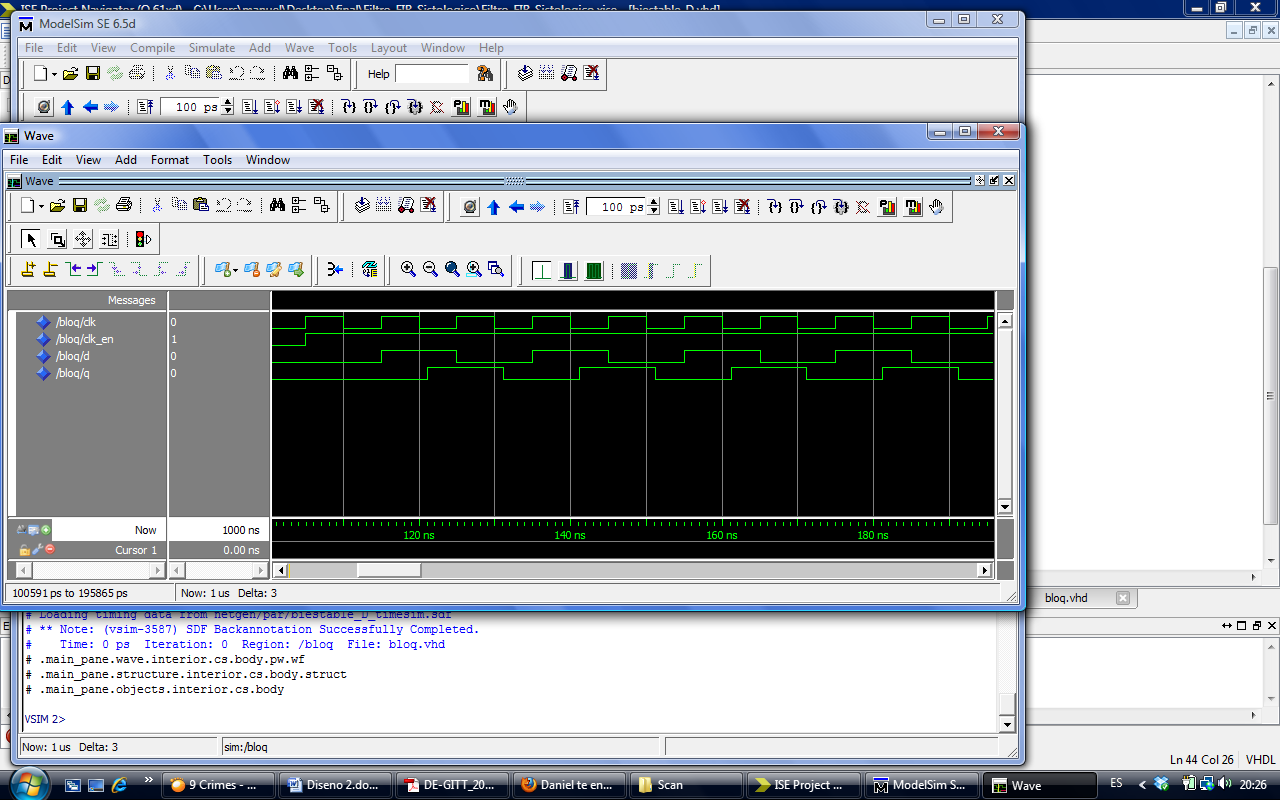
# Como ejemplo hemos creado un Biestable D con clk\_enable y usando

*if (clk'event and clk='1') then* y hemos hecho varias simulaciones:

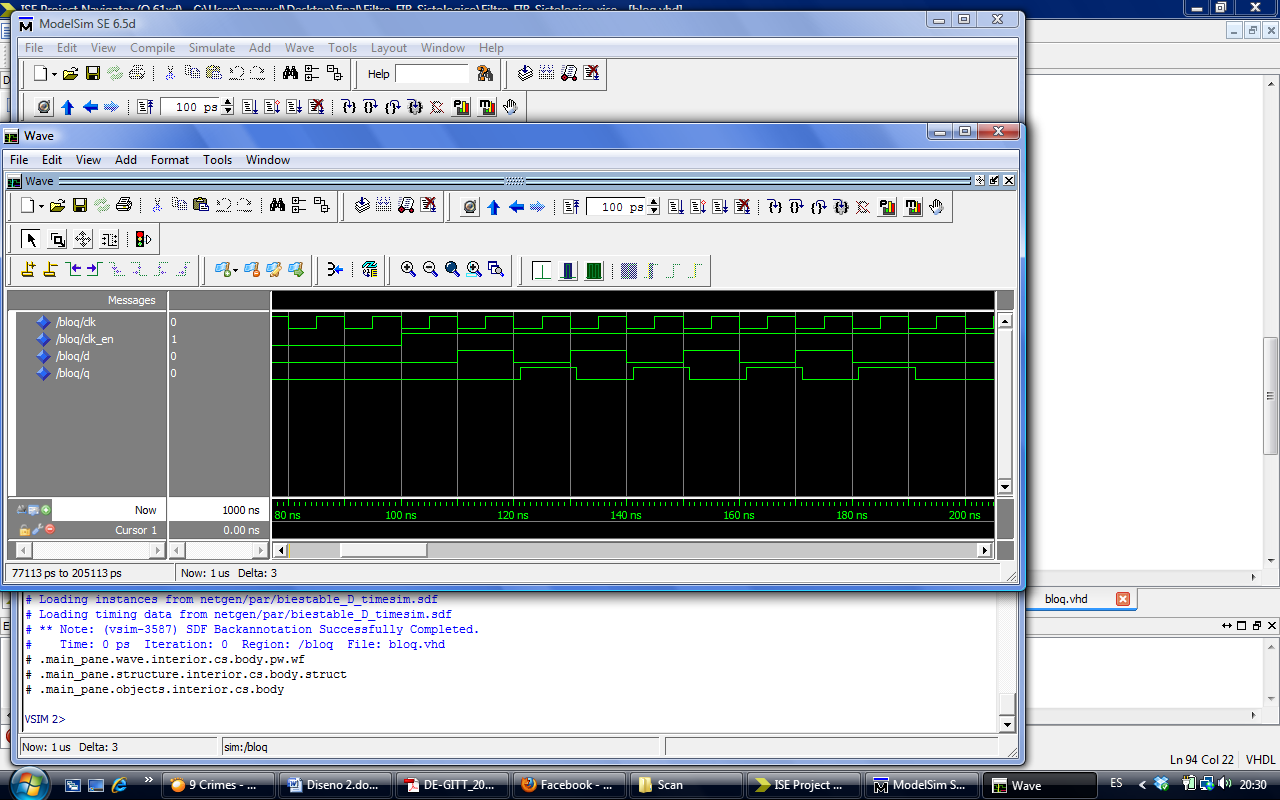
**1 - Simulacion Funcional => Cambiando la señal de entrada D en los flancos de subida**



**2 - Simulacion Temporal: => Cambiando la señal de entrada D en los flancos de subida**

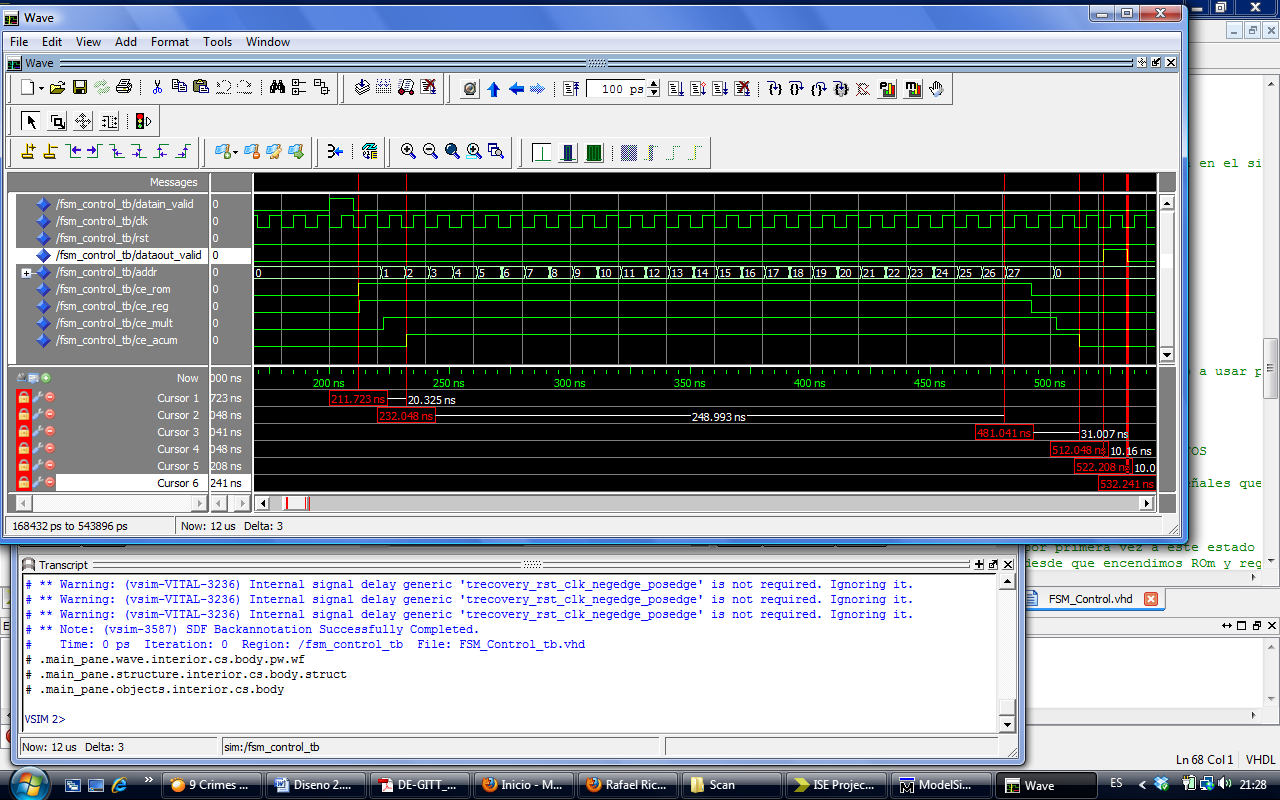


**3 -Simulación Temporal: => Cambiando la señal de entrada D en los flancos de bajada**



**MARCOS !! TEMPORAL DEL FSM CONTROL**

**CUANDO VAYAS HA HACER EL TEMPORAL DE ALGO, CREO K PARA QUE SE HAGA BIEN TIENES QUE PONER ESA MIERDA COMO "TOP MODULE" Y LUEGO SINTETIZARLA Y GENERARLA**



**Funcionamiento con Matlab**

# Vamos a probar el funcionamiento con el código de Matlab:

- Utilizamos los código .vhd y .m dados por los profesores.

- Utilizamos la función genera\_tono, donde la frecuencia de muestreo que tenemos que poner es 8000 Hz ya que así es como hemos diseñado el filtro.

- Para comprobar su funcionamiento utilizaremos 3 frecuencias significativas diferentes y veremos sus valores una vez pasada la etapa transitoria, siendo su frecuencia de corte 500 Hz.

- Junto a los códigos pondremos los "datain.txt" generados y sus respectivos "dataout.txt"

- Para comprobación de los resultados con Matlab utilizar la misma expresión:

**b=Num/max(abs(Num));**

**b=round(b\*2^15);**

**Salida = conv(b,x)/ 2^15**

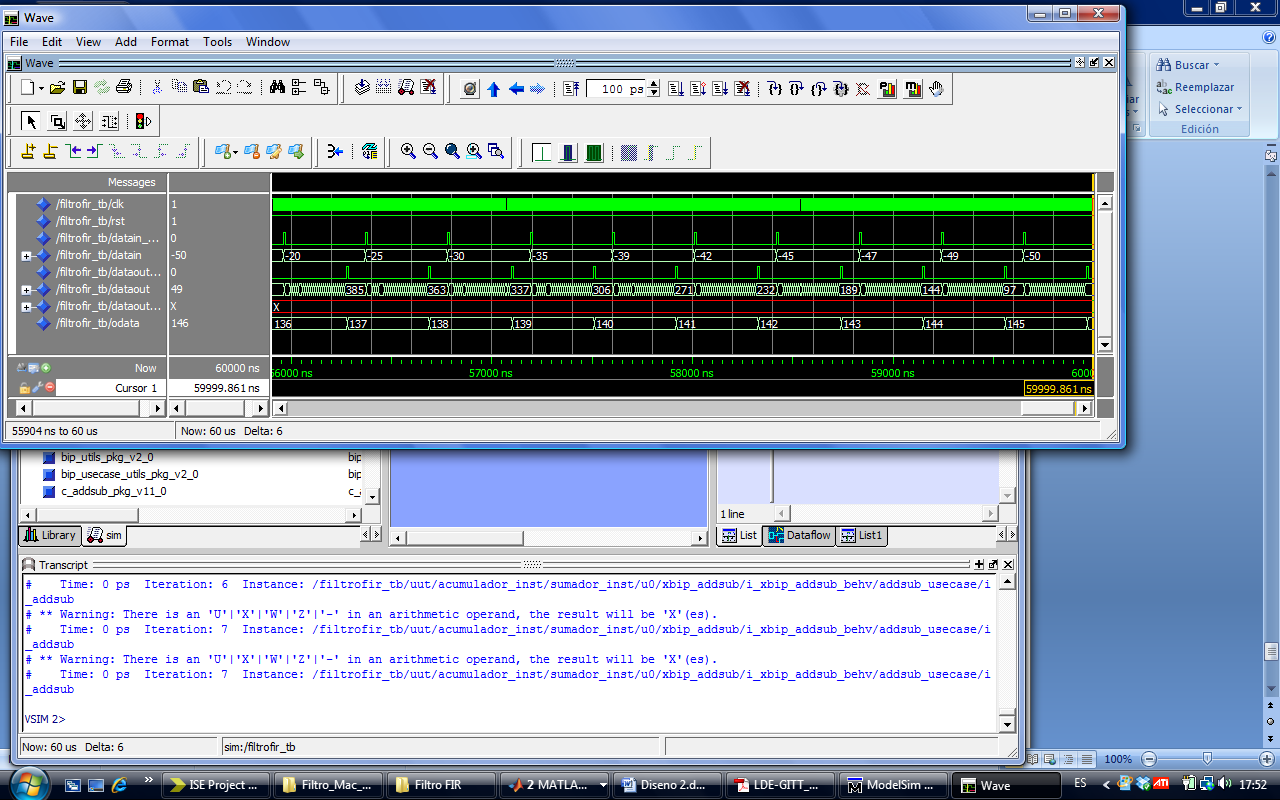
**Salida = floor (Salida)**

Siendo Num los coeficientes y "x" el resultado de "genera\_tono"

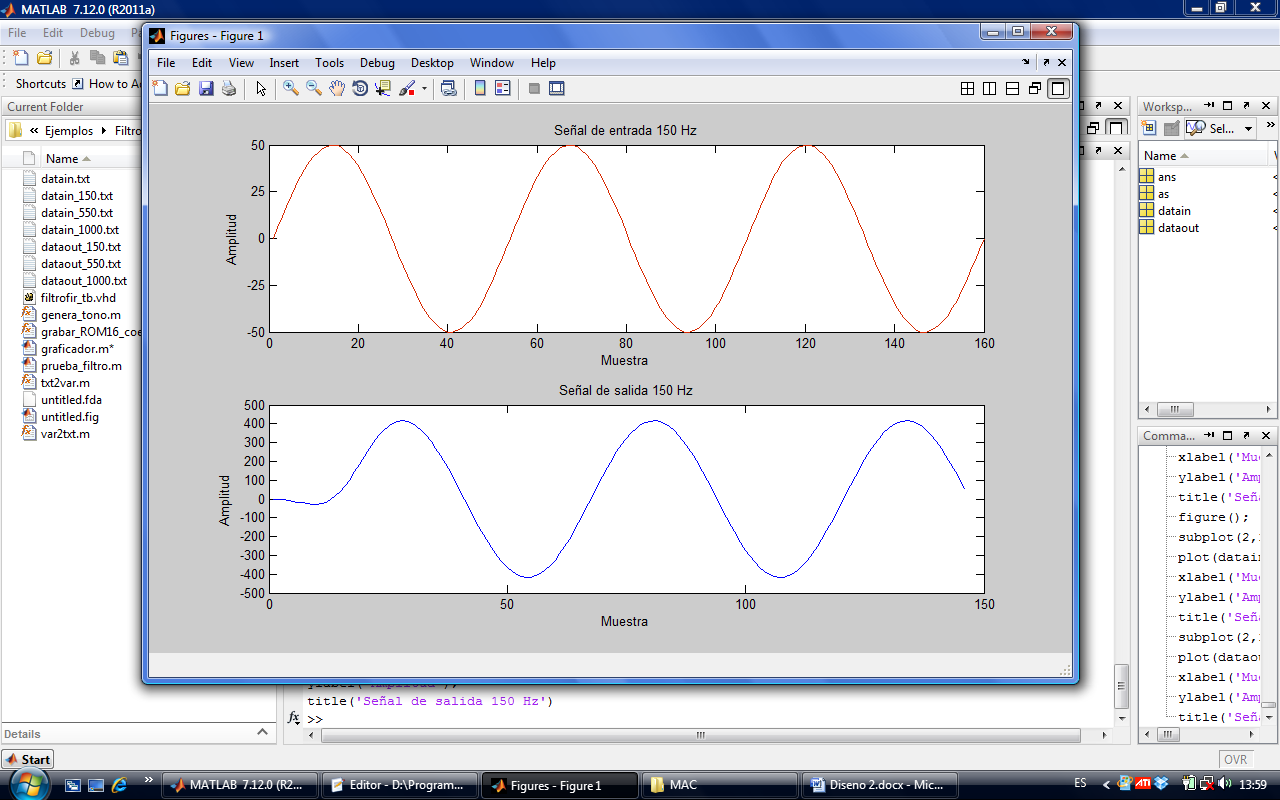
**- Frecuencia banda de paso: 150 Hz**

genera\_tono(150,0.02,50,8000,'datain.txt')

- Gráfica del filtro vhdl:



- Representación en Matlab

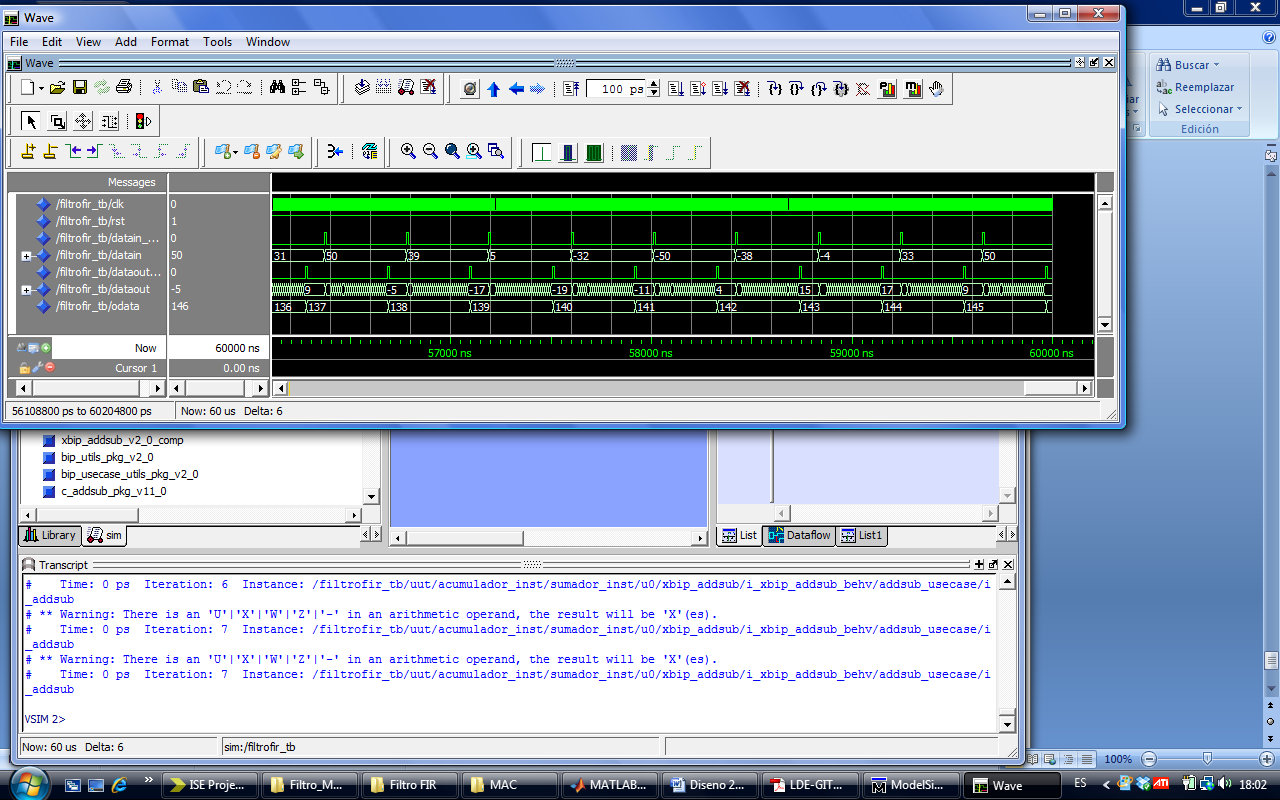


- Vemos que deja pasar la señal y la amplifica

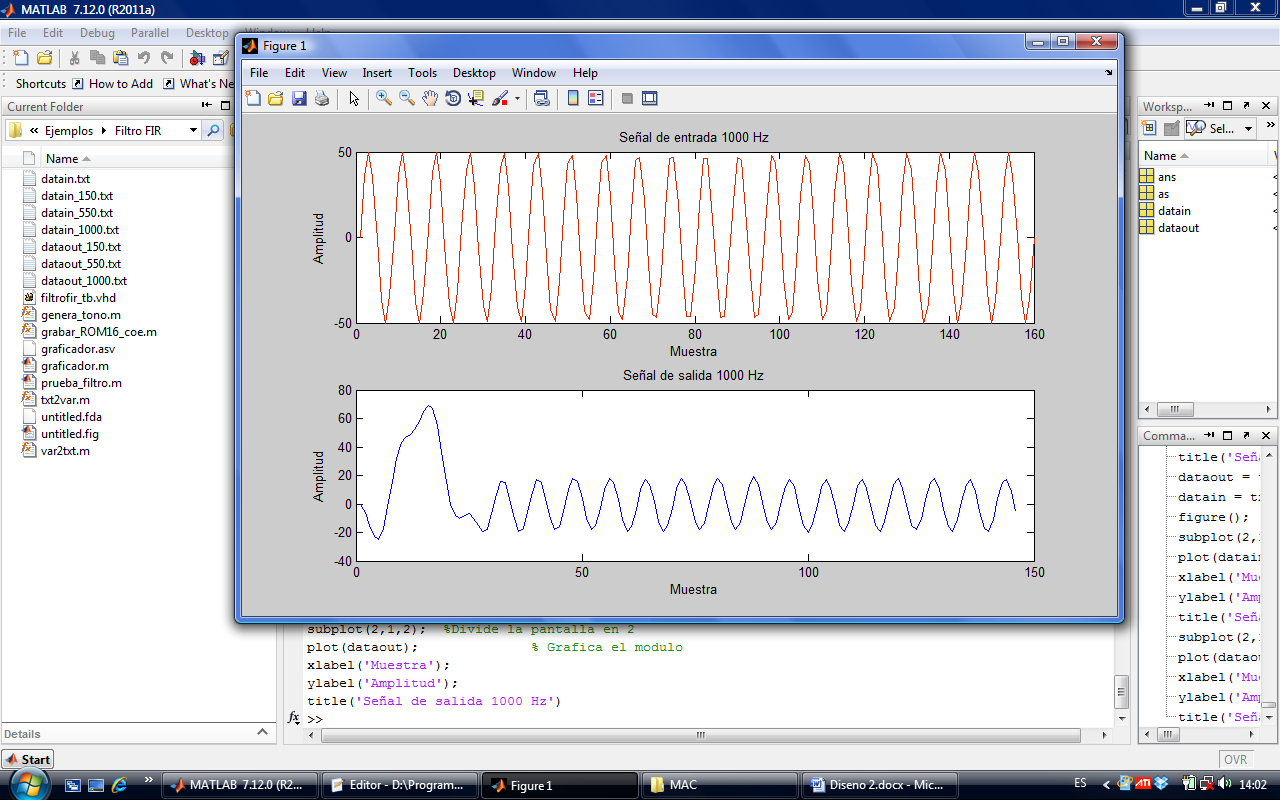
**- Frecuencia banda de rechazo: 1000 Hz**

genera\_tono(1000,0.02,50,8000,'datain.txt')

- Gráfica del filtro vhdl:

****

- Representación en Matlab

****

- Vemos que prácticamente no deja pasar la señal, en comparación con la banda de paso.

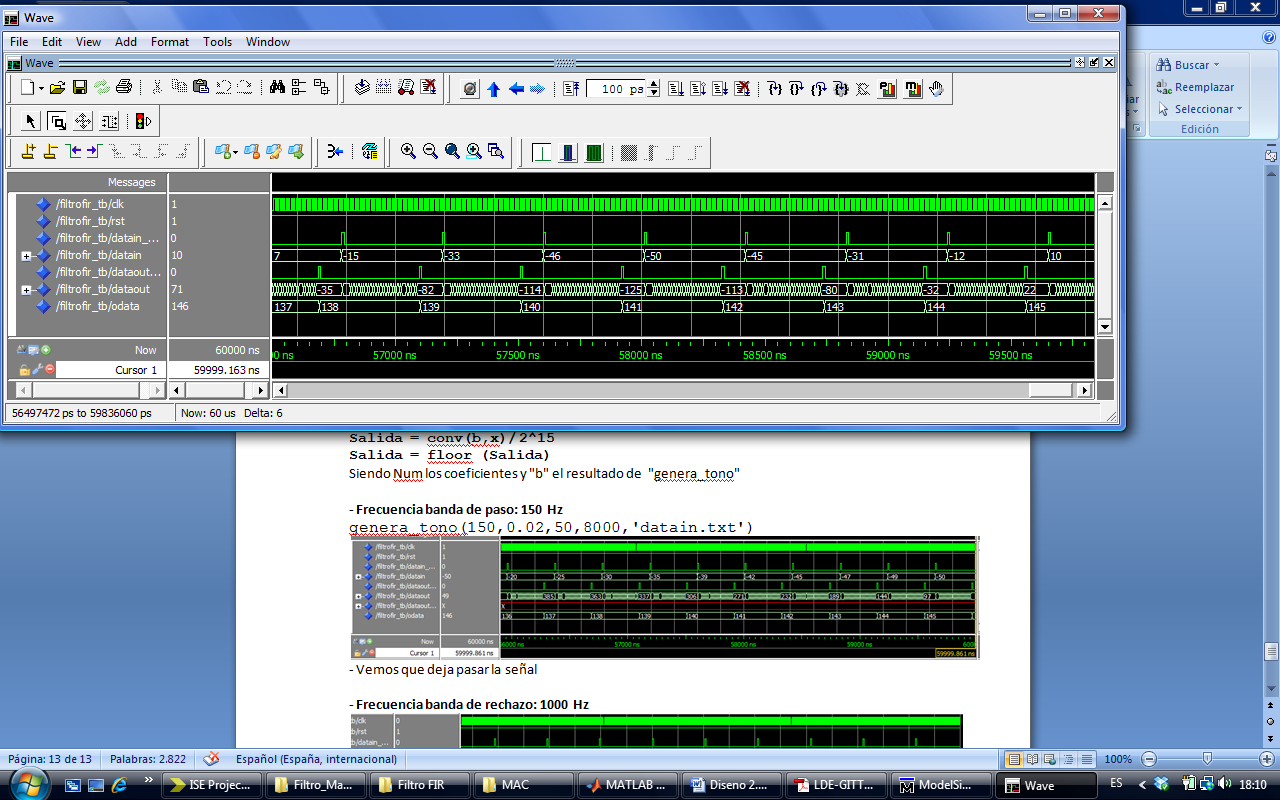
La señal de salida es 20 veces menor que en caso de la banda de corte.

Hay una caída de 13 dB.

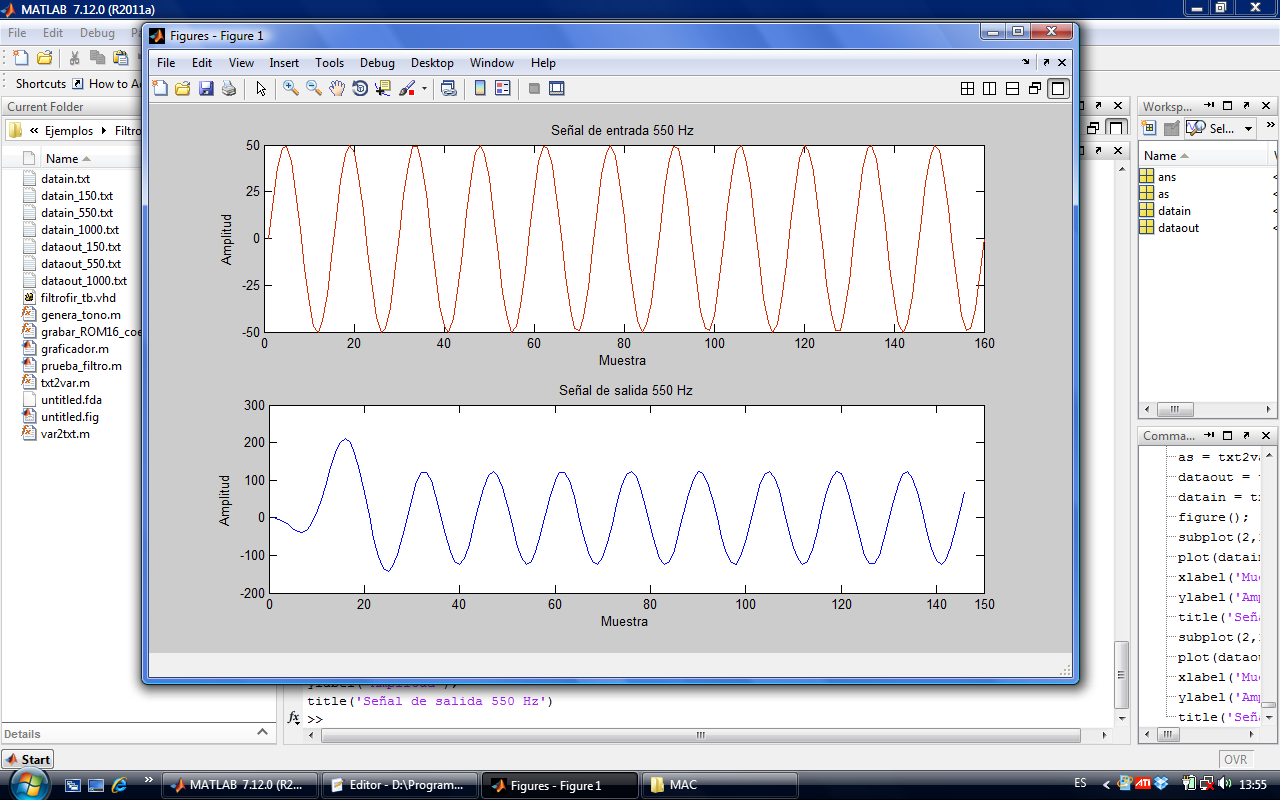
**- Frecuencia banda intermedia: 550 Hz**

genera\_tono(550,0.02,50,8000,'datain.txt')

- Gráfica del filtro vhdl:



- Representación en Matlab



- Vemos que se obtienen valores intermedios.

- La señal de salida es unas 4 veces menor => Cae unos 6 dB