Diseño Automático de Sistemas Fiables

Memoria de práctica

David Martínez Campos

ÍNDICE

[**Introducción**](#_heading=h.o0lsufpuj9o1) **3**

[**Práctica 1. Debouncer**](#_heading=h.3znysh7) **4**

[**Sincronizador**](#_heading=h.liqtvyav5474) **4**

[**Debouncer**](#_heading=h.qesefhmnkorp) **5**

[**Top\_practica1**](#_heading=h.6693tfc6bqlz) **13**

[**Constraints**](#_heading=h.pdo4xhf93pdu) **15**

[**Comentarios/Observaciones**](#_heading=h.5jwuj39shf3l) **16**

[**Práctica 2. Micrófono DDR**](#_heading=h.4936ioa6uq7n) **17**

[**Generador de reloj síncrono**](#_heading=h.ndrazdt660q1) **17**

[**Registro de desplazamiento con carga en paralelo**](#_heading=h.ke609ujyxz49) **19**

[**MEMS**](#_heading=h.eyteuae1czwa) **20**

[**Constraints**](#_heading=h.y9y1j3mfip8y) **21**

[**Comentarios/Observaciones**](#_heading=h.6m5pjgijhlrf) **21**

[**Práctica 3. Verificación VHDL**](#_heading=h.8djm5rxqdvcn) **22**

[**Testbench (tb\_top1\_csv)**](#_heading=h.r7tdgqn24o8l) **22**

[**Comentarios**](#_heading=h.rtg117oejmgl) **24**

[**Práctica 4. Softcore**](#_heading=h.75uojep7w6y3) **25**

[**Comentarios**](#_heading=h.agqgfuxmukus) **26**

[**Práctica 5. Filtro Digital**](#_heading=h.okvdpt1nwldc) **27**

[**Comentarios**](#_heading=h.uv4km2bbk0dj) **30**

# **Introducción**

Memoria para exponer los resultados y explicaciones de las prácticas de diseño automáticos de sistemas fiables. Aclarar que parte del código de estas prácticas ha sido hecho en la versión 2019.2.1 y otra parte en la version 2021.2.

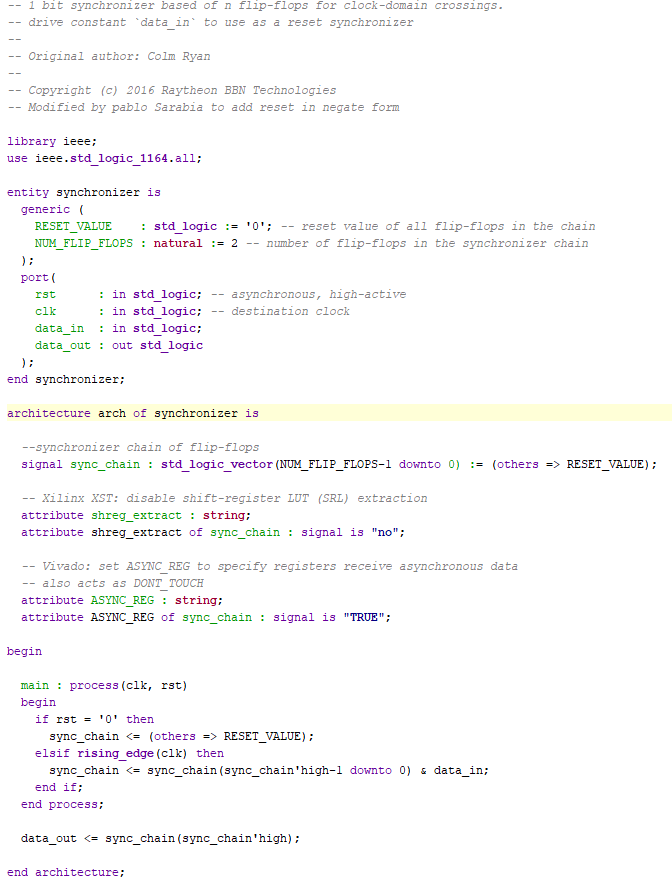
El código se encuentra en el repositorio: <https://github.com/dmartinezc4/Dis_automatico_sistemas>

# **Práctica 1. Debouncer**

Como bien dice la práctica vamos a tener tres ficheros: el sincronizador, el debouncer y el fichero que los une a los dos llamado top\_practica1. Aparte de esto tenemos las constraints que hay que cambiar para que en caso de ser puesto en la placa este funcione.

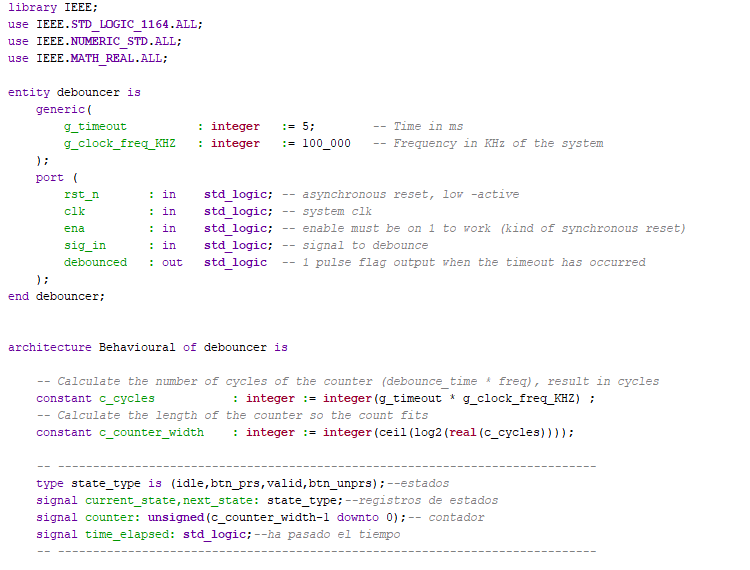
## **Sincronizador**

Este es el código del sincronizador hecho por el profesor



## **Debouncer**

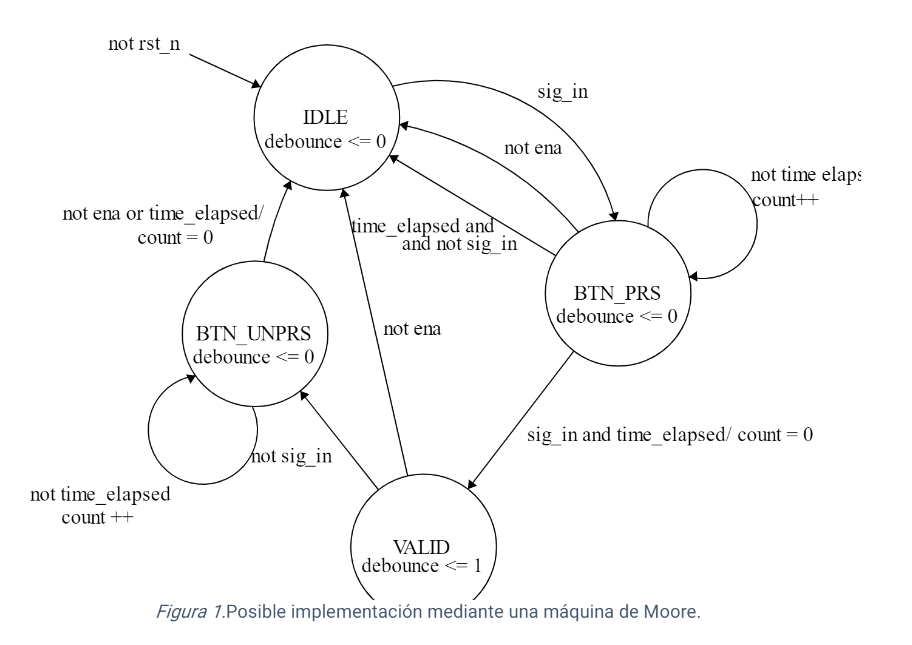
Debido al código ser muy grande primero vemos como se instancia la source del debouncer y las signals y constantes necesarias.



Nada mas terminar tenemos los dos primeros process el del contador y el de cambiar de estado respectivamente.

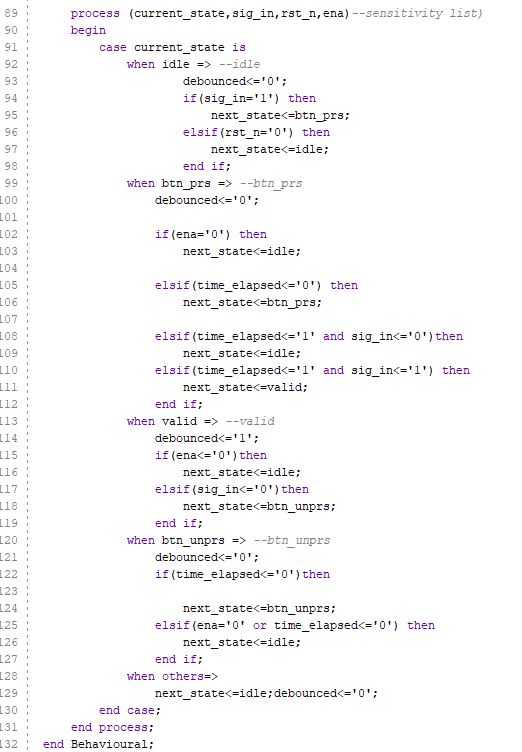


Tras esto implementaremos la siguiente máquina de Moore propuesta en un process



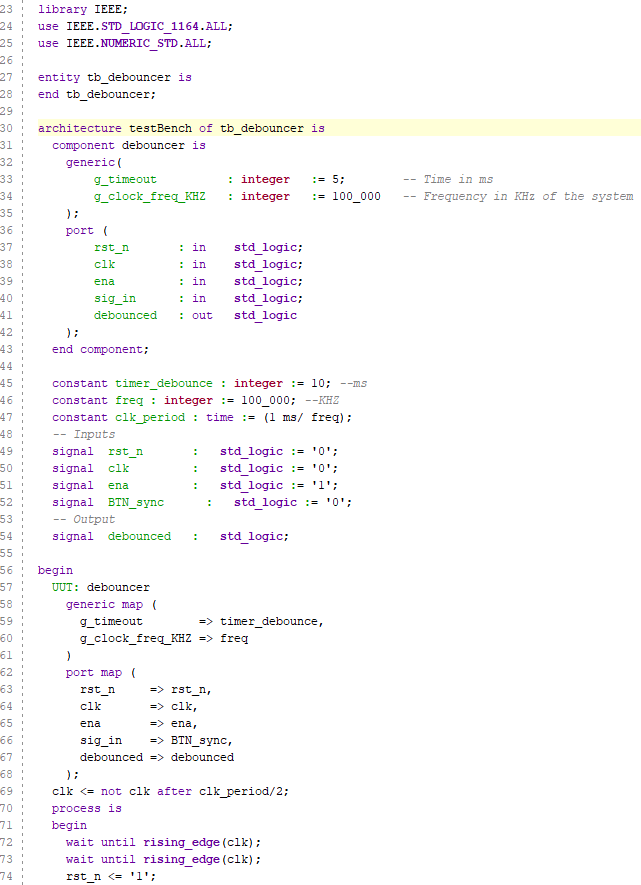
La lista de sensibilidad del process será: el current state (porque es lo necesario para todos los process de fsm tras hacer los de clk); y luego sig\_in, rst\_n y ena porque, como se ve en el diagrama, si una de estas se activa puede cambiar el flujo de trabajo.

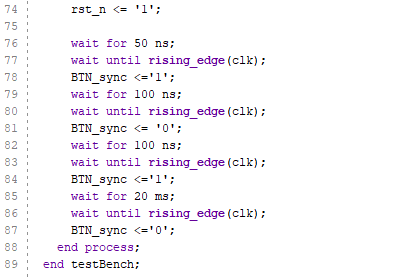
Al ser hecho con case en vez de if, necesitamos un when others, que en este caso hace que la señal de debounced sea 0 y hace que el siguiente estado al que saltar sea el de idle.

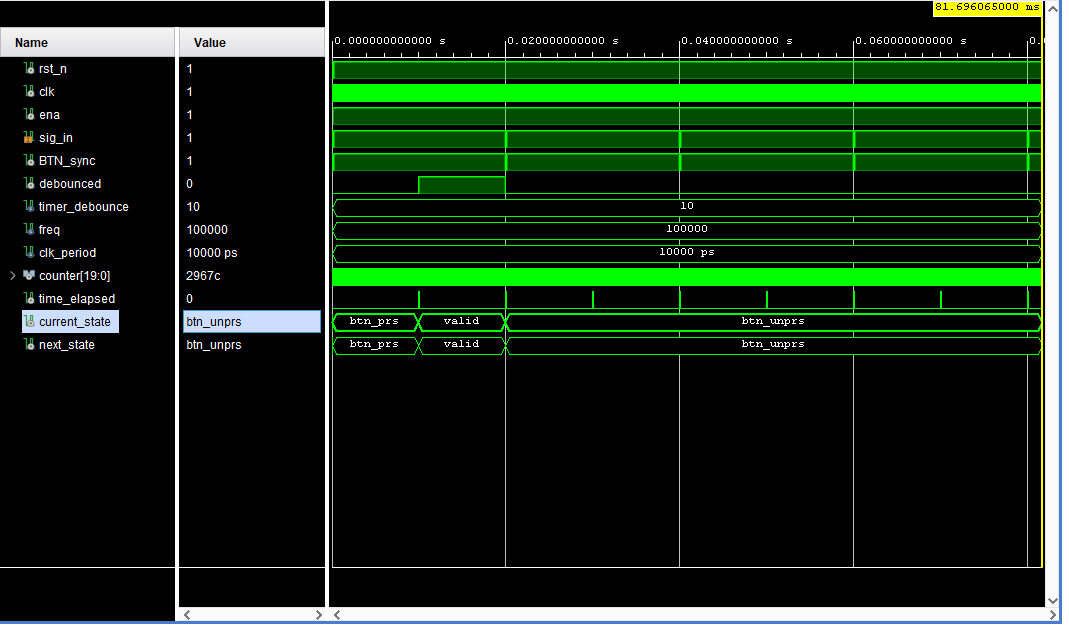


Respecto a los testbench, debido a como lo hemos hecho nunca pasa a idle tras btn\_unprs. Además como el clk depende del reloj que es muy grande cambia cada mucho tiempo, pero vamos a mostrar la imagen general y un poco de instantes específicos.

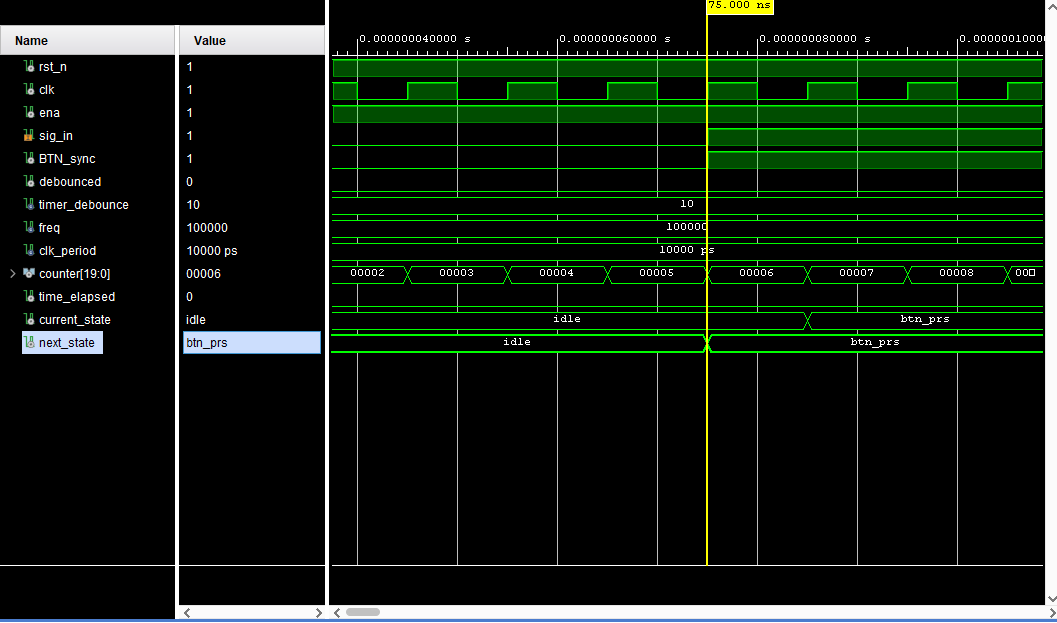
Primero el código del testbench usado para el debouncer



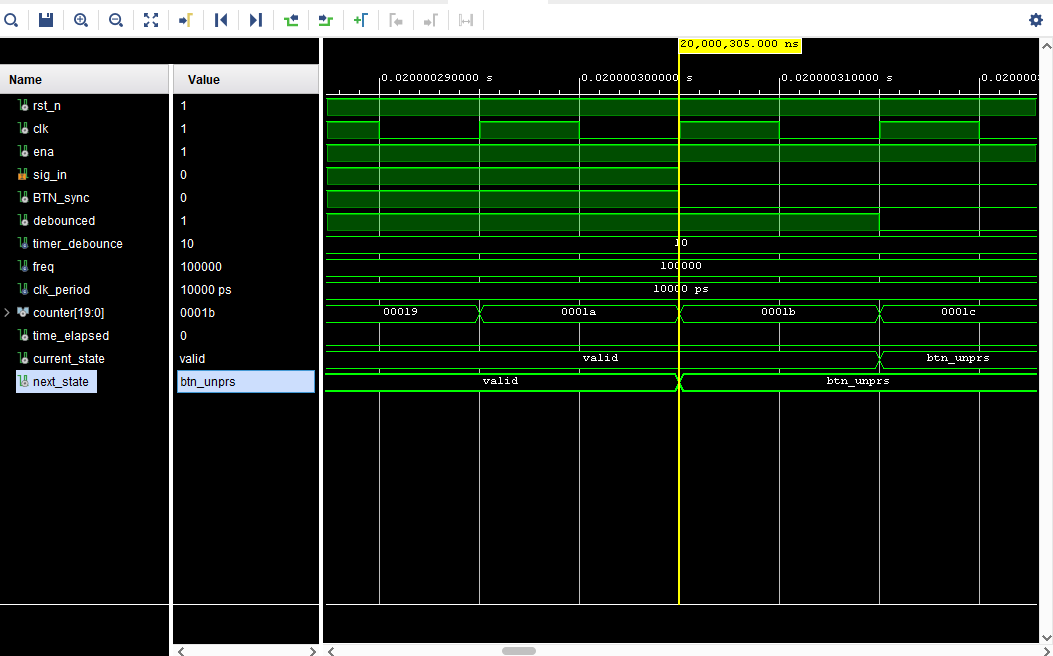




Esto sería la imagen general, seleccionando el estado btn\_unprs, donde se cumple que el debounced es 0 y debido a que ena y time elapsed es 1 el siguiente estado es btn unprs.



Esta sería la imagen del principio del testbench en el que está en estado idle, y tras 5 segundos este pasará al estado btn\_prs.

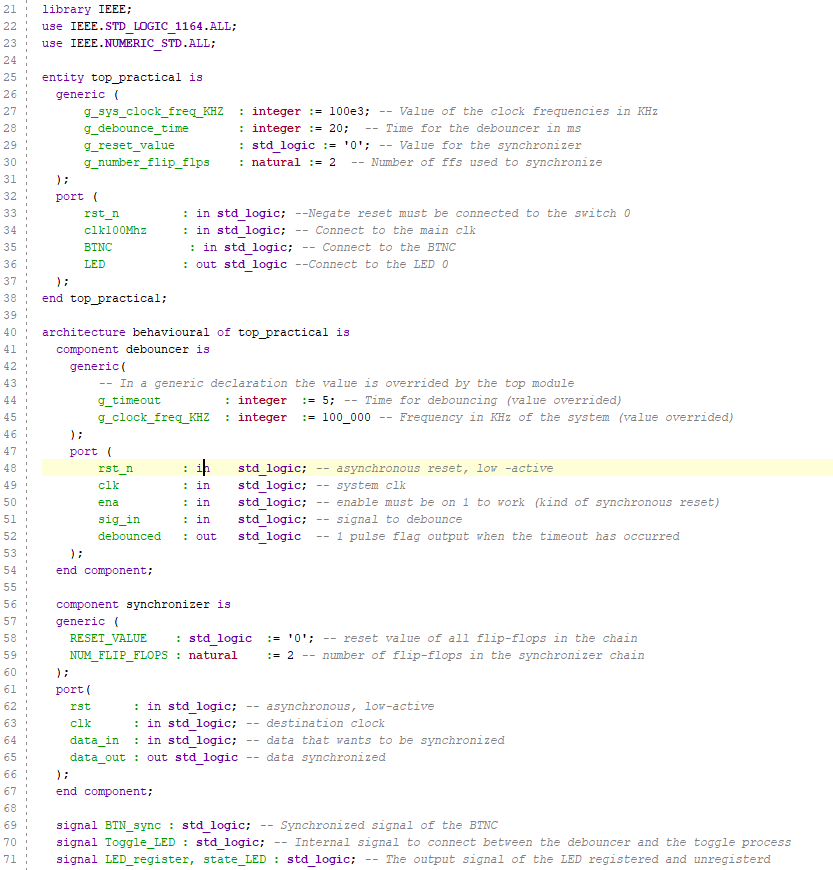


Y por último un instante de tiempo donde ocurren muchos cambios muy seguidos

## **Top\_practica1**

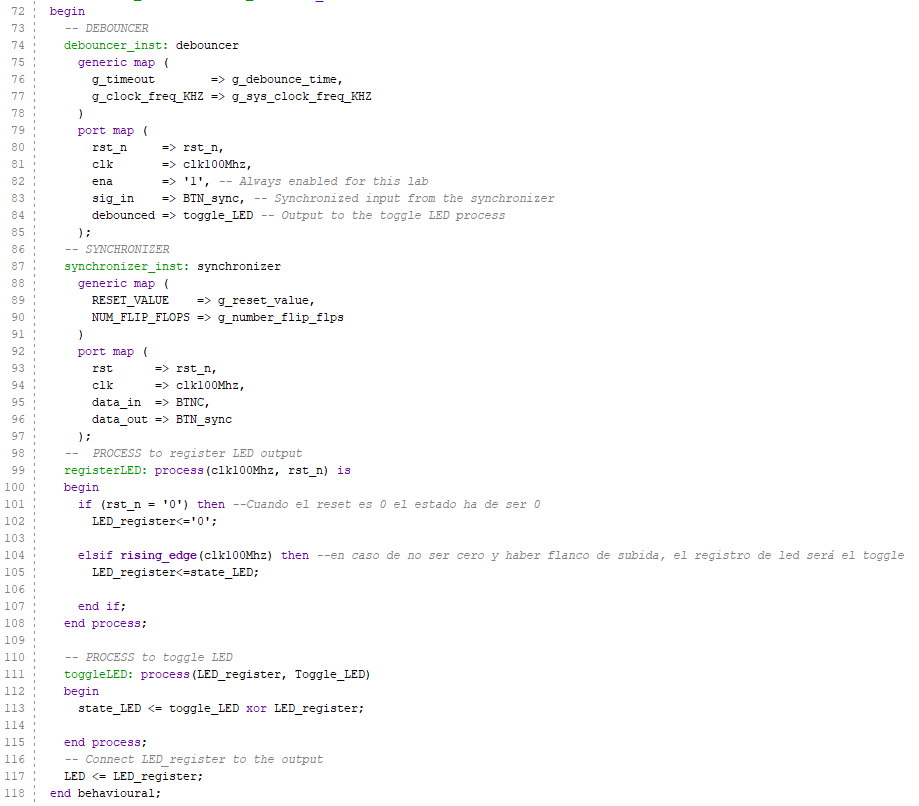
De nuevo al ser un fichero bastante grande lo mostraremos en partes.

Primero la instanciación de la entidad en si y la instanciación de los componentes que esta usa, además de las signals necesarias.



Tras esto empezamos el begin de la architecture.

Desde la línea 73 hasta la 97 hacemos un mapeado de las variables para que se puedan comunicar, así como de los genéricos.

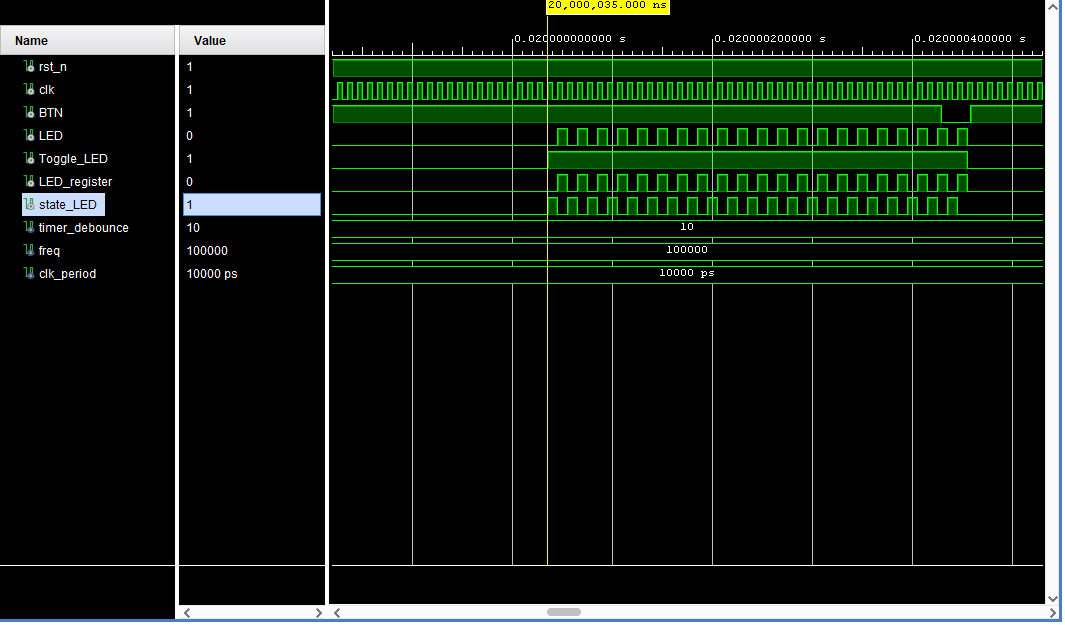


Como se puede observar hay dos process uno que lo que hace es hacer registro del output led y otro que lo que hace es encender el led. Fuera de todo process se hace la asignación para que la salida LED valga lo que haya en el LED\_register.

Respecto al testbench



Esto sería la imagen general del testbench que como nos pasaba en el testbench anterior.



Esta sería la única parte (por como hemos hecho el debouncer), en la que el led cambia en ciertos instántes a 1.

## **Constraints**

## **Comentarios/Observaciones**

Respecto a los testbench, principalmente del debouncer: no sabemos donde poner time\_elapsed<=’0’ sin que nos saliese unsigned y nos permitiese un muestreo cómodo en los tb. La idea sería poner time\_elapsed<=’0’ en el process del timer en el momento en el que el counter todavia no ha llegado a tomar el tiempo; pero eso nos dejaría (en un testbench) un instánte de tiempo ínfimo con el que muestrear.

El testbench del debouncer que muestra que no pasa a idle por como

# 

# 

# **Práctica 2. Micrófono DDR**

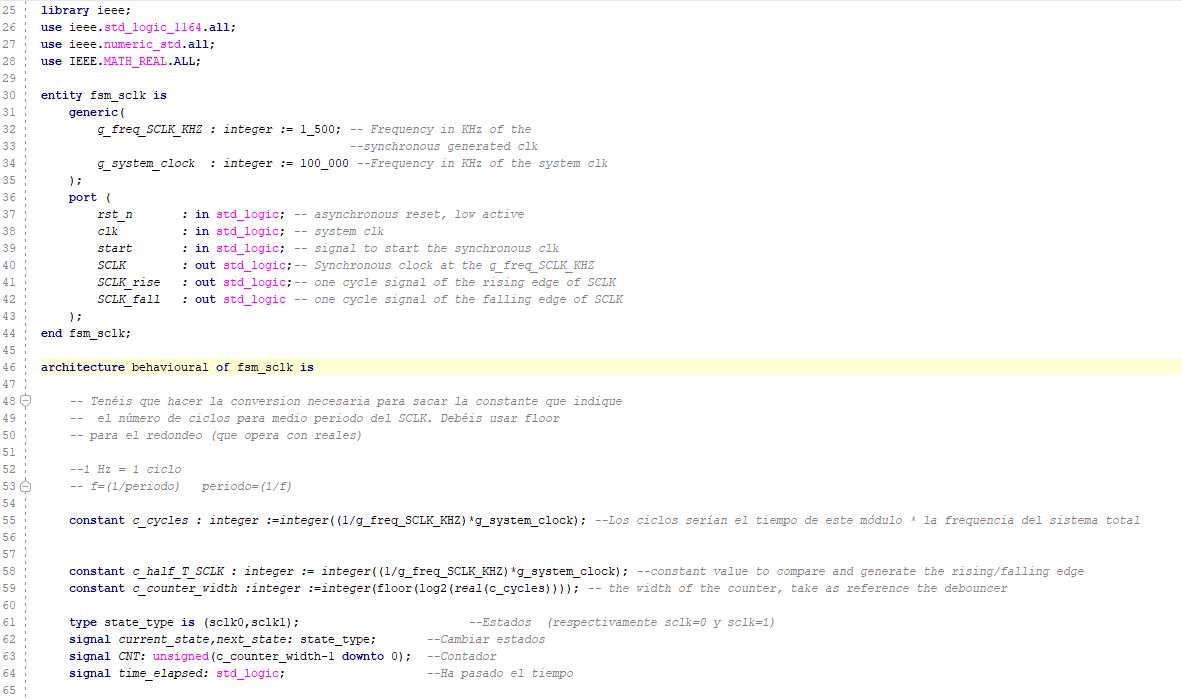
En esta práctica no le he podido dedicar todo el tiempo necesario y solo tengo los .vhd de la práctica.

Aparte he considerado un reset asíncrono al no especificarse nada en la memoria.

## **Generador de reloj síncrono**

Este módulo ocupa muchas líneas lo primero que vamos a ver es la entidad que simplemente es la declarada por el profesor y las señales creadas para poder hacer un reloj síncrono mas las señales para la máquina de estados.

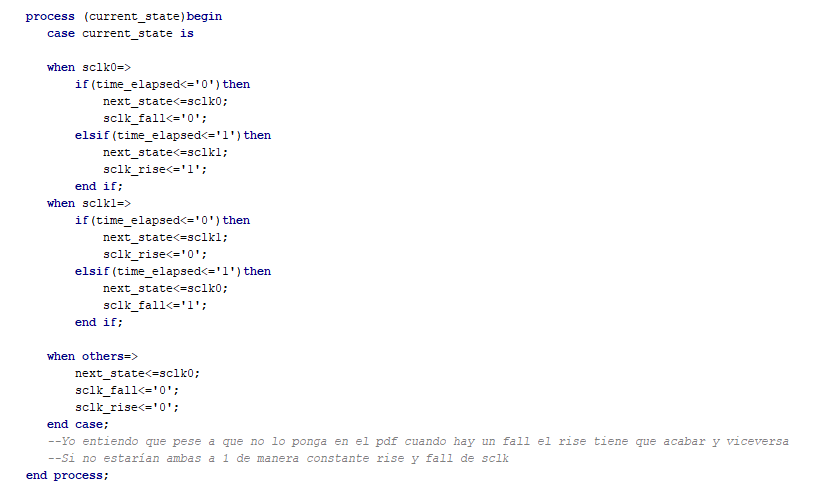
Necesitamos la libreria mathreal para poder hacer el redondeo “floor”.



Despues tenemos los dos siguientes process: el del clk con con el contador y otro tambien de clk pero para poder cambiar de estados.

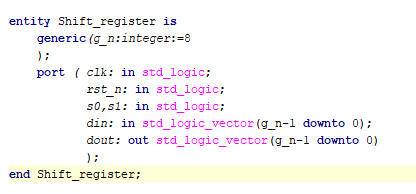


Tras estos llega el process final que nos dice el comportamiento en cada estado.

He supuesto que si alguna de las señales sclk está activada la otra no puede estarlo; no puede haber un flanco de subida y bajada al mismo tiempo. 

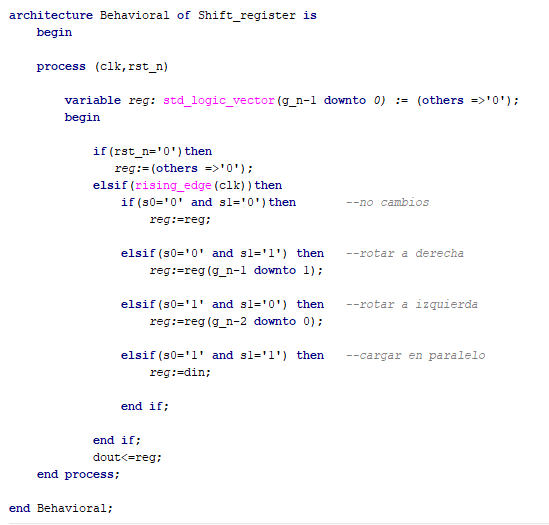
## **Registro de desplazamiento con carga en paralelo**

La entidad en si no es muy complicada se explica basicamente con la imagen proporcionada en la práctica. He decidido que el generico sea de 8 bits porque en la memoria “palabras de bits”; entiendo que se refiere a de 8 bits. En cualquier caso en caso de necesitar otro tamaño simplemente se puede cambiar el número tras el integer de g\_n, ya que din y dout dependen de este g\_n.



La arquitectura requiere que nos declaremos una variable para poder hacerle cambios sin problema durante el único process. Esta variable también tiene el la anchura igual que los din y dout.

En el reset “limpiamos” o ponemos todo a 0’s; luego los casos son los expuestos en la memoria.

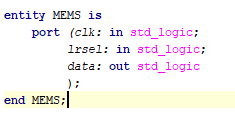


## **MEMS**

No ha sido posible terminar esta entidad a tiempo, y en estos momentos tiene errores de sintaxis y compilación.

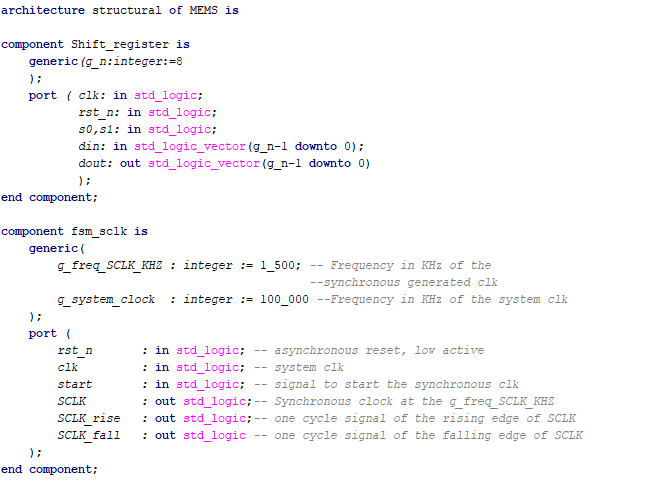
MEMS es un entidad estructural, es decir toma los otros archivos mencionados y los usa como componentes.

Nuestra idea es que la entidad es de este estilo



Como la mostrada en el diagrama, faltaría añadirle los genericos y constantes usados en las otras entidades para poder hacer un port correcto.

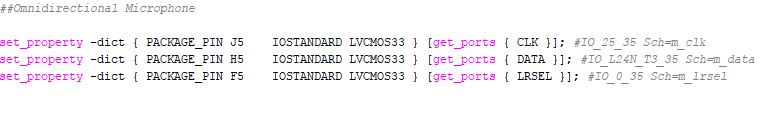
A parte de esto declaramos los componentes.



Y el resto que no se ha terminado y tiene errores sería; portear y describir el diseño de este source

## **Constraints**

Hemos descomentado las líneas y porteado las conexiones a las señales correctas respecto al fichero de constraints



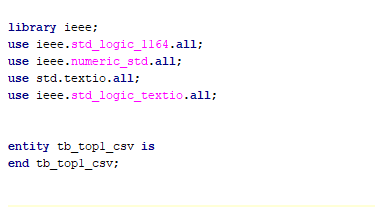
## **Comentarios/Observaciones**

Obviamente no he podido trabajar en esta práctica como en las otras por la falta de tiempo y disposicion a trabajar de mi compañero de equipo.

# **Práctica 3. Verificación VHDL**

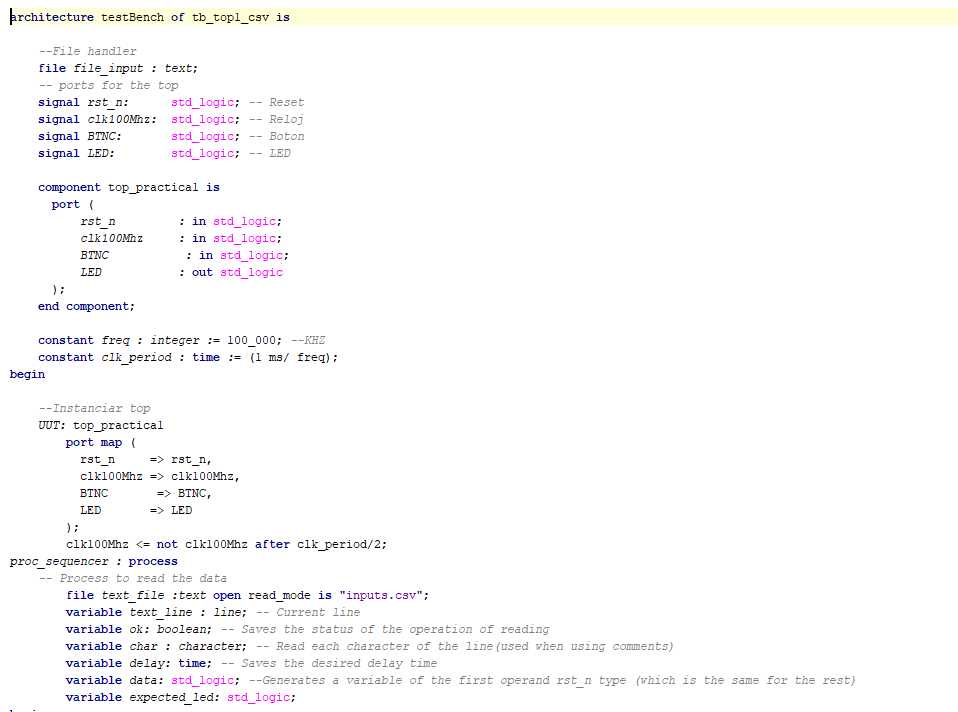
## **Testbench (tb\_top1\_csv)**

Estas son las librerías que necesitamos para leer archivos y la entidad vacia habitual de los testbenchs

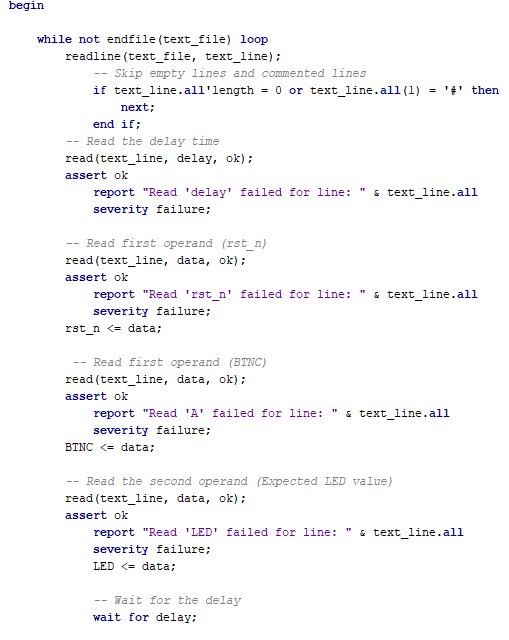


El resto simplemente se ha realizado de acuerdo a los pasos de la guia 3 de vhdl provista en el campus.

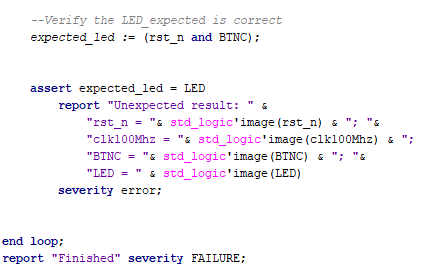
Declarar las signals para portearlas al UUT y luego ya un process para leer el fichero csv que se integrado como source



Tras esto vendría la parte del process donde se le dice que satarse del inputs.csv y que leer



Tras leer el fichero se espera el delay, y se hace un assert del led y led esperado para comprobar el resultado.



## **Comentarios**

De nuevo debido a la reiterada escasez de tiempo, no ha habido tiempo para hacer un testbench satisfactorio.

# 

# 

# 

# 

# 

# 

# 

# 

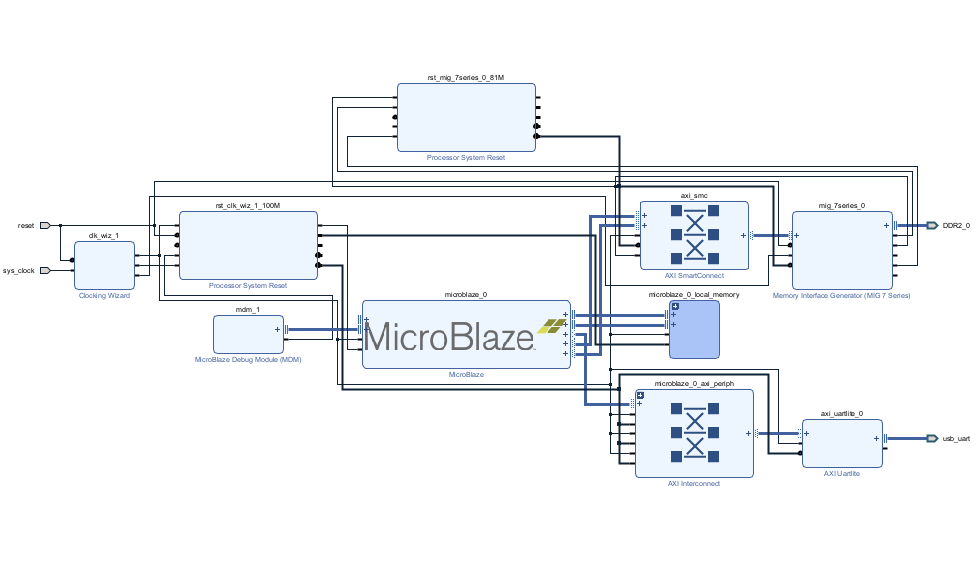
# 

# **Práctica 4. Softcore**

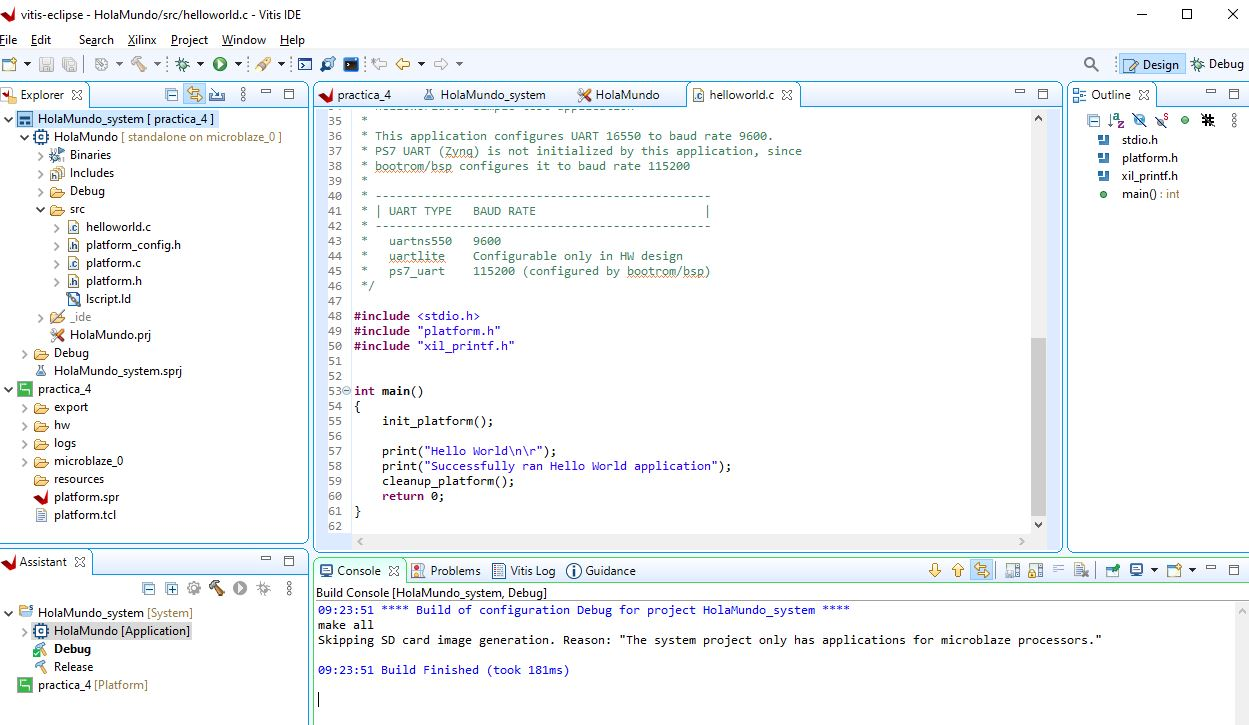
Esta práctica consiste en realizar un proyecto en vitis y ponerlo en la placa.

Como no disponemos de ningún portátil que pueda tener vitis y vivado y funcionar al mismo tiempo, solo puedo argumentar que hemos seguido los pasos de la práctica guiada y que por tanto debería de funcionar.

El diagrama de bloques generado queda de la siguiente manera:



Al exportar este proyecto a vitis y compilar con un hola mundo, nos sale que la compilación es correcta.



## **Comentarios**

Debido al no disponer de un ordenador, que pueda arrancar vivado y vitis carezco de la capacidad de comprobar los resultados reales en una placa pero, he comentado paso a paso lo que he hecho con un par de compañeros y debería de ejecutarse de manera correcta.

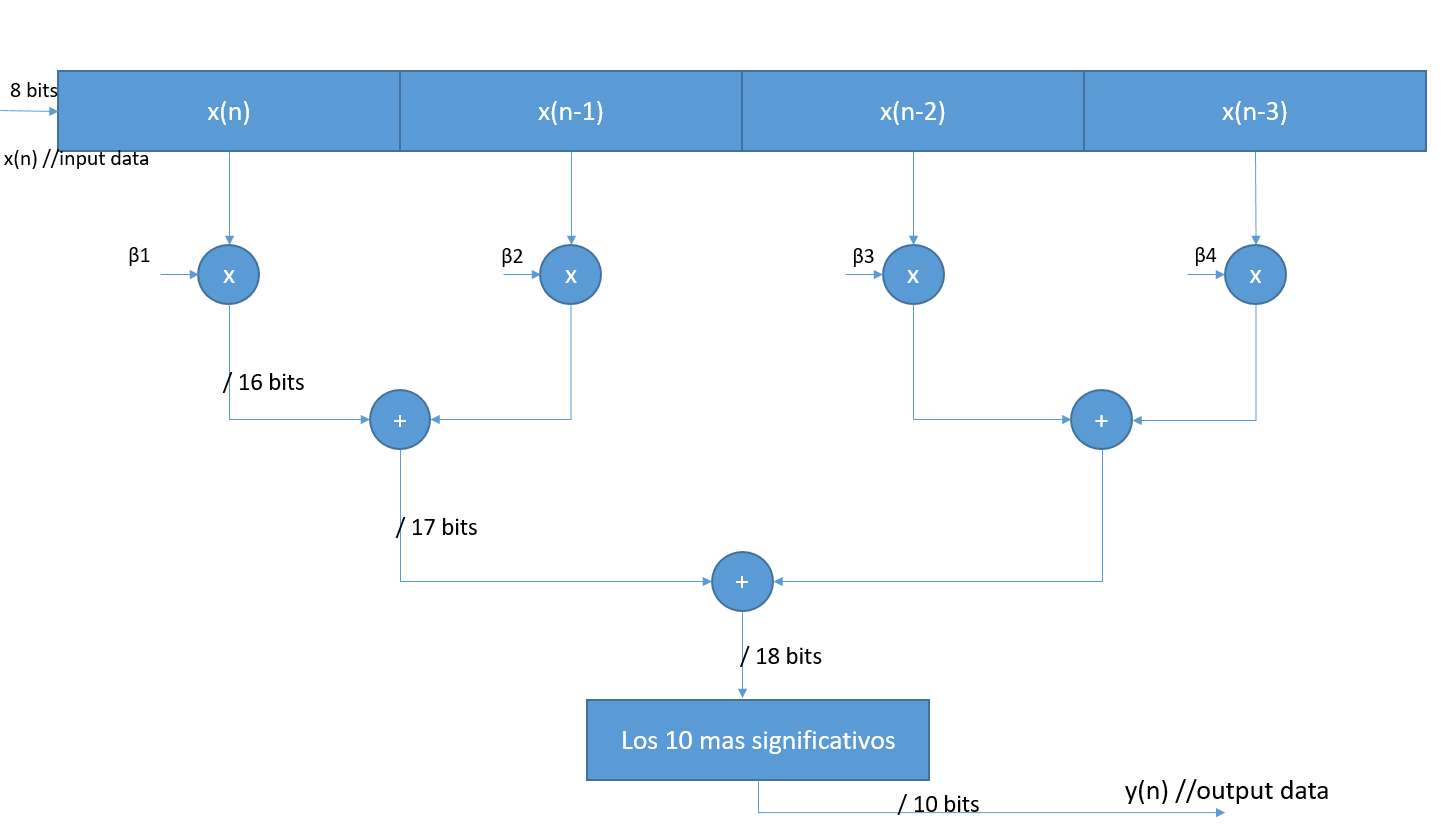
# 

# 

# **Práctica 5. Filtro Digital**

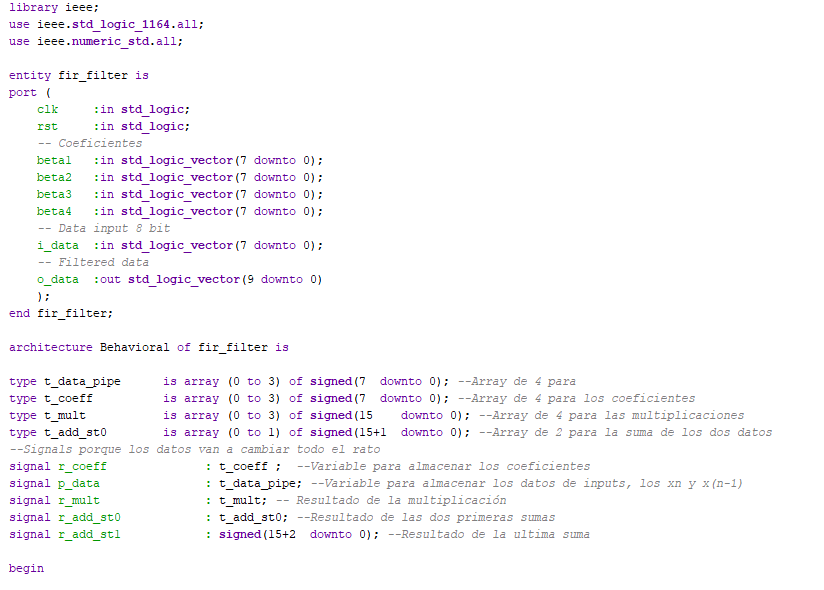
A la hora de crear un filtro digital solo necesitaremos una source, la cual hemos llamado fir. Pero antes de empezar a programar nos hemos puesto de acuerdo en como hacerlo. También seguiremos la recomendación del profesor de usar arrays para poder hacer las sumas y multiplicaciones con mas tranquilidad

Nuestro diagrama a la hora de realizar las operaciones será el siguiente:



Intentamos hacerlo lo más paralelizable posible. Por tanto siguiendo el diagrama vamos a necesitar vamos a necesitar 8 vectores de tamaño 8 para los coeficientes (betas) y para los datos de i\_input (x(n)). Vamos a necesitar otros cuatro arrays de tamaño 16 para las multiplicaciones. Y finalmente vamos a necesitar dos arrays para las sumas uno de 17 para las “sumas intermedias”, y otro de 18 para la suma final. Justo después vamos a hacer un resize del array de 18 para quedarnos con los 10 más significativos y que este sea el valor de o\_output (y(n)).

Primero la entidad y declaración de variables, antes del begin, que ya hemos comentado

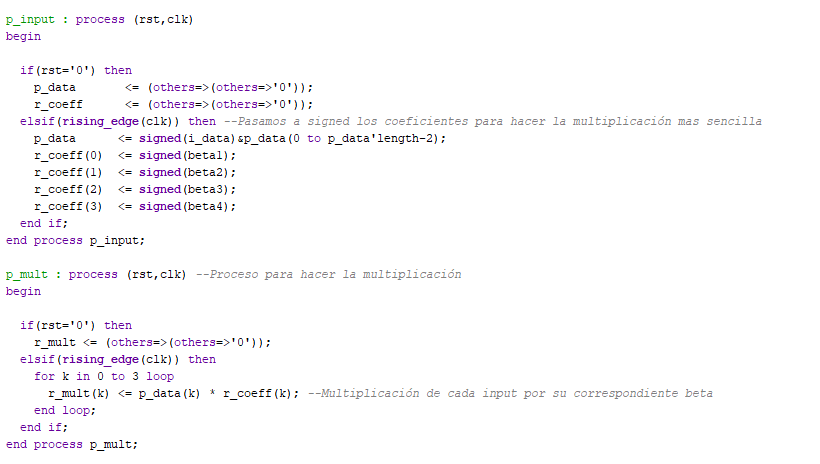


Como todos los process no caben en un mismo recorte vamos poco a poco. Los process que haremos serán: asignación de input y coeficientes, multiplicación, sumas intermedias, suma final y resize.

Todos los process dependen de rst y clk, con un reset asíncrono activo a nivel bajo (hace reset si está a 0). El reset en cada uno de los process pone las variables llenas a 0.

En el input en caso de que no haya reset simplemente asignamos a nuestras signals que son arrays, los valores correspondientes.

Mientras que en el process de multiplicación ponemos en nuestro array de resultados de multiplicación, el proceso de multiplicar los coeficientes por los datos.

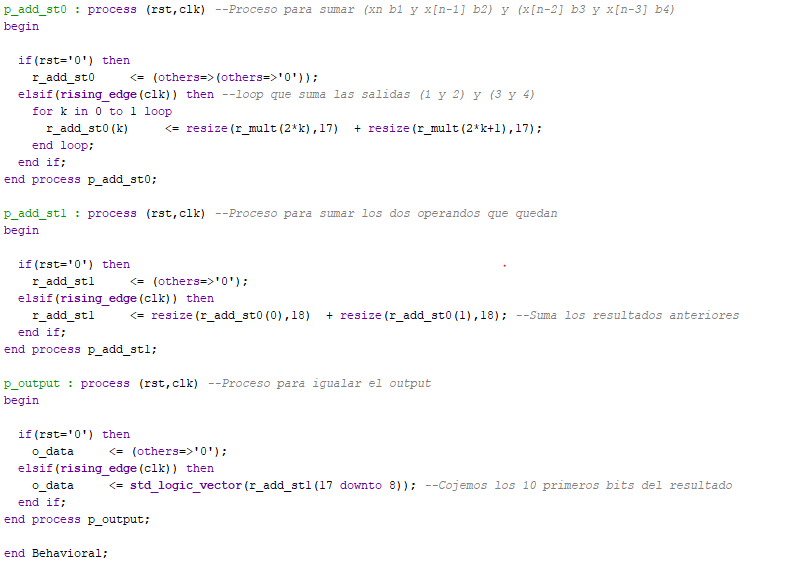


El siguiente es el process de las “sumas intermedias”; lo que hacemos es un for que suma los resultados de la multiplicación de manera que sea: resultado 1 + resultado 2 y resultado 3 + resultado 4; haciéndoles a todos estos un resize para que sean de tamaño 17.

Después el process de la “suma final” es una suma sin necesidad de for porque solo hay dos cosas que sumar, las dos “sumas intermedias” y estas también sufren un resize pero esta vez a tamaño 18.

Finalmente el process de asignar el output. Aquí asignamos el valor de la “suma final” a el output, pero como solo queremos los 10 bits mas significativos lo que hacemos es asignar como un vector nuestro array pero desde el bit 17 al bit 8.

Tras esto cerramos la arquitectura Behavioral



## **Comentarios**

Me resulta un poco tarde cambiar la estructura ya, pero, si no me equivoco con los types y los arrays; el r\_coeff podria ser del mismo tipo que p\_data, de manera que podriamos ahorrarnos la declaración de un tipo especifico de array.