

Diseño de Sistemas Electrónicos Digitales

# Sistema de grabación, tratamiento y reproducción de audio

Versión 4.2

Pablo Ituero y Fernando García Redondo

Nombre y apellidos de los miembros del grupo:

Marcos Gómez Bracamonte  
Miguel Medina Antón



*Departamento de Ingeniería Electrónica*

Diseño de Sistemas Electrónicos Digitales 2019  
*Sistema de grabación, tratamiento y reproducción de audio*

El bloque de filtrado está basado en el “Mini-project” “FIR-Filter Design” del curso IL2204 “DSP Design usign HDL” del “Royal Institute of Technology” de Estocolmo, Suecia.

*Departamento de Ingeniería Electrónica*  
UNIVERSIDAD POLITÉCNICA DE MADRID  
España  
Teléfono: +34915495700 ext. 4207  
E-mail: pituero@die.upm.es

# Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Especificaciones mínimas</b>	<b>1</b>
2.1. Descripción de la estructura del sistema . . . . .	3
2.2. Uso de package . . . . .	3
<b>3. Planificación temporal</b>	<b>4</b>
<b>4. Evaluación</b>	<b>4</b>
<b>5. Bloque 1: Interfaz de audio</b>	<b>5</b>
5.1. Generador de enables y salida de reloj del micrófono . . . . .	6
5.2. Interfaz del micrófono . . . . .	8
5.2.1. Esquema de muestreo . . . . .	8
5.2.2. Diseño e implementación . . . . .	9
5.2.3. Estrategia de test . . . . .	12
5.3. Interfaz de la salida de audio . . . . .	12
5.3.1. Estrategia de test . . . . .	14
5.4. Integración de la interfaz de audio . . . . .	14
5.4.1. Estrategia de test . . . . .	15
5.5. Implementación física y test en placa . . . . .	15
<b>6. Bloque 2: Filtro FIR</b>	<b>17</b>
6.1. Filtros digitales . . . . .	17
6.2. Implementación hardware de filtros FIR . . . . .	18
6.3. Sistemas basados en rutas de datos . . . . .	19
6.4. Metodología de implementación de algoritmos de procesado de señal . . . . .	20
6.4.1. Asignación . . . . .	20
6.4.2. Planificación temporal . . . . .	21
6.4.3. Vinculación . . . . .	21
6.4.4. Implementación . . . . .	22
6.5. Notación en punto fijo . . . . .	24
6.6. Especificación del bloque . . . . .	26
6.7. Tareas prácticas . . . . .	27
6.7.1. Creación de la ruta de datos . . . . .	27
6.7.2. Creación del controlador . . . . .	32
6.7.3. Creación de un testbench avanzado . . . . .	34
<b>7. Bloque 3: Controlador y Memoria</b>	<b>36</b>
7.1. Memoria RAM . . . . .	36
7.2. Conversión entre codificaciones . . . . .	38
7.3. Controlador y sistema global . . . . .	39

<b>8. Mejoras opcionales</b>	<b>40</b>
8.1. Control de volumen logarítmico . . . . .	40
8.2. Eco y reverberación configurables . . . . .	40
8.3. Información de segundos restantes . . . . .	41
8.4. Looper . . . . .	41
<b>9. Apéndice 1</b>	<b>42</b>
<b>10. Apéndice 2</b>	<b>43</b>

## 1. Introducción

Este documento describe el proyecto final de la asignatura Diseño de Sistemas Electrónicos Digitales del plan de estudios GITST de la ETSI de Telecomunicación de la UPM.

El proyecto guía el diseño y la implementación de un sistema electrónico digital de complejidad media y está pensado para ocupar la segunda mitad de los de las horas de laboratorio totales de la asignatura.

En cuanto a los objetivos docentes, el proyecto pretende asentar y profundizar en las metodologías de diseño combinacional y secuencial, en el diseño de máquinas de estados finitos y de máquinas de estados finitos con ruta de datos asociadas, la metodología de síntesis de alto nivel y en cuestiones de temporización.

En concreto se propone un sistema que, utilizando las interfaces de entrada y salida de audio de la placa con la que han estado trabajando los alumnos en la primera mitad de las horas prácticas de laboratorio, adquiere, almacena, procesa y reproduce sonidos.

## 2. Especificaciones mínimas

El sistema tiene las siguientes especificaciones globales mínimas:

- El sistema se implementa sobre la placa Nexys 4DDR utilizando el flujo de trabajo de Vivado.
- El sistema recibe la información de audio del micrófono integrado en la placa.
- El sistema reproduce el audio a través de la salida mono de audio integrada en la placa.
- El sistema se controla mediante tres botones (BTNL, BTNC y BTNR) y dos switches (SW0 y SW1).
- El sistema funciona a una frecuencia de reloj de 12 MHz, que se genera a partir del reloj de 100 MHz disponible en la placa.
- El sistema tiene un reset global procedente del botón BTNU.
- El audio se muestrea y se reproduce a una frecuencia de 20 kHz.
- El sistema se describe en VHDL y la entidad global tiene la siguiente declaración:

```
entity dsed_audio is
  Port (
    clk_100Mhz : in std_logic;
    reset: in std_logic;
    --Control ports
    BTNL: in STD_LOGIC;
    BTNC: in STD_LOGIC;
    BTNR: in STD_LOGIC;
```

```

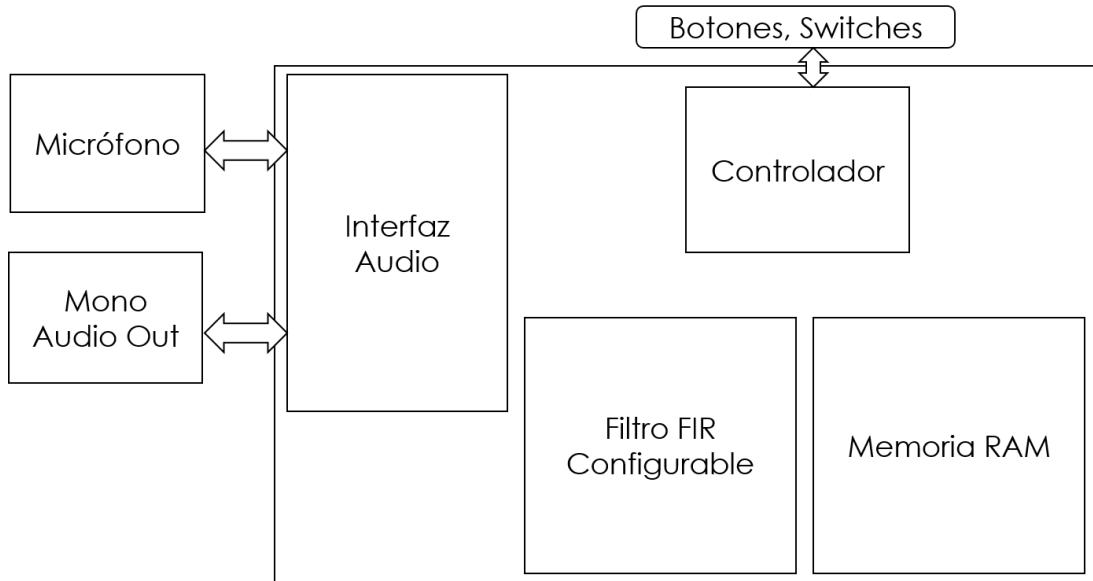
SW0:      in STD_LOGIC;
SW1:      in STD_LOGIC;
--To/From the microphone
micro_clk : out STD_LOGIC;
micro_data : in STD_LOGIC;
micro_LR : out STD_LOGIC;
--To/From the mini-jack
jack_sd : out STD_LOGIC;
jack_pwm : out STD_LOGIC
);
end dsed_audio;

```

El funcionamiento es el siguiente:

- El sistema parte de una memoria vacía de audio.
- Cuando se presiona BTNL y durante el tiempo que éste esté presionado el sistema graba el audio.
- Cada vez que se vuelve a presionar BTNL el nuevo audio se añade al final del audio que estaba previamente grabado.
- Cuando se presiona BTNC se borra todo el audio grabado.
- Cuando se presiona BTNR el sistema debe realizar acciones en función de la información de los switches:
  - Si ambos switches están a '0' el sistema reproducirá todo el audio grabado.
  - Si SW0 está a '1' y SW1 está a '0' el sistema reproducirá el audio almacenado al revés.
  - Si SW0 está a '0' y SW1 está a '1' el sistema reproducirá el audio filtrado por un filtro paso bajo.
  - Si SW0 está a '1' y SW1 está a '1' el sistema reproducirá el audio filtrado por un filtro paso alto.

## 2.1. Descripción de la estructura del sistema



El sistema está compuesto por cuatro bloques principales:

- La interfaz de audio tiene dos funciones principales: En primer lugar es la encargada de la digitalización de las señales PDM provenientes del micrófono. Por otro lado es el bloque encargado de interactuar con la salida de audio proporcionándole una señal PWM. Este bloque se va a implementar siguiendo una metodología de máquina de estados finitos con ruta de datos asociada.
- El filtro FIR se encarga del procesado digital de la señal de audio para las funcionalidades de filtro paso alto y filtro paso bajo. Este bloque se va a implementar siguiendo la metodología de síntesis de alto nivel.
- La memoria RAM almacena las muestras de audio que se han grabado previamente. Este bloque se va generar con el asistente arquitectural de Vivado.
- El controlador es el encargado de orquestar toda la actuación del sistema proporcionando señales de control a los distintos bloques para ejecutar las órdenes del usuario del sistema. En este bloque el estilo de diseño es libre.

## 2.2. Uso de package

El código emplea un package propio denominado package\_dsed. El objetivo es guardar todas las constantes y tipos de datos que vayan a ser utilizados por varios bloques en una librería común, de tal manera que el código se haga más legible, evitando el uso de expresiones numéricas que no se sabe de dónde aparecen.

A continuación se proporciona el código correspondiente a la declaración del package:

```
package package_dsed is
```

```
constant sample_size: integer := 8;  
end package_dsed;
```

La idea es ir añadiendo líneas a esta declaración en función de las necesidades del diseño. Para hacer uso de este package, hay que incluir la siguiente línea en la cabecera de nuestro código:

```
use work.package_dsed.all;
```

### 3. Planificación temporal

El proyecto está pensado para realizarse en parejas en la segunda mitad de las sesiones de laboratorio de la asignatura, esto equivale aproximadamente a 6 sesiones de 2-3 horas con el apoyo del profesor más otras 15 horas de trabajo no guiado.

El proyecto se ha dividido en hitos que se corresponden con los bloques principales de la estructura del sistema:

- Semanas 1-2. Interfaz de audio.
- Semanas 3-4. Filtro FIR.
- Semanas 5-6. Controlador y memoria.

Este documento está pensado como un cuaderno y guía de laboratorio donde ir apuntando las decisiones de diseño y los resultados de la implementación.

### 4. Evaluación

La evaluación del proyecto tendrá en cuenta los siguientes elementos:

- Una pequeña demostración en la que se mostrará el funcionamiento de la implementación en la placa, así como el resultado de distintos testbenches del sistema.
- La calidad de las decisiones de diseño reflejadas en la copia de cada pareja de este documento.
- La calidad del código VHDL. Es importante que cada pareja aplique criterios de buena legibilidad en su código (uso de comentarios explicativos, uso sistemático de prefijos y sufijos en nombres de puertos, señales y variables, etc).
- La calidad de las mejoras, que se evaluarán mediante los tres criterios anteriores.

## 5. Bloque 1: Interfaz de audio

La interfaz de audio va a tener dos funciones principales: Por un lado, en el momento de la grabación, va a recibir la información del micrófono, codificada en PDM y transformarla en una palabra digital de 8 bits a una frecuencia de muesteo de 20 kHz. Por otro lado, en el momento de la reproducción del audio, va a transformar una palabra de 8 bits en un pulso de anchura variable (codificación PWM) a una velocidad de 20000 muestras por segundo.

Tiene la siguiente declaración de entidad en VHDL:

```
entity audio_interface is
    Port ( clk_12megas : in STD_LOGIC;
            reset : in STD_LOGIC;
            --Recording ports
            --To/From the controller
            record_enable: in STD_LOGIC;
            sample_out: out STD_LOGIC_VECTOR (sample_size-1 downto 0);
            sample_out_ready: out STD_LOGIC;
            --To/From the microphone
            micro_clk : out STD_LOGIC;
            micro_data : in STD_LOGIC;
            micro_LR : out STD_LOGIC;
            --Playing ports
            --To/From the controller
            play_enable: in STD_LOGIC;
            sample_in: in std_logic_vector(sample_size-1 downto 0);
            sample_request: out std_logic;
            --To/From the mini-jack
            jack_sd : out STD_LOGIC;
            jack_pwm : out STD_LOGIC);
end audio_interface;
```

Descripción de cada puerto:

- clk\_12megas: Reloj global de la arquitectura a 12 MHz.
- reset: Reset global del sistema.
- record\_enable: Señal de control de la grabación. Cuando esté a '1', la digitalización de la información del micrófono funcionará.
- sample\_out: Dato de 8 bits (número positivo sin signo) correspondiente a la señal digitalizada obtenida del micrófono.
- sample\_out\_ready: Señal de control que proporciona un pulso activo de un periodo de reloj de duración cada vez que se proporciona un nuevo dato digitalizado.
- micro\_clk: Salida del reloj del micrófono. Un reloj de 3 MHz obtenido a partir de nuestro reloj de 12 MHz.

- micro\_data: Entrada de la señal PDM proveniente del micrófono.
- micro\_LR: Salida de control del micrófono que determina si las muestras se toman en el flanco de subida o de bajada del reloj. Dejaremos este valor estable en '1', correspondiente al flanco de subida.
- play\_enable: Señal de control de la reproducción de audio. Cuando esté a '1', se procederá a la generación de la señal PWM hacia la salida de audio mono.
- sample\_in: Dato de 8 bits correspondiente a la señal que hay que reproducir.
- sample\_request: Señal de control que proporciona un pulso activo de un periodo de reloj de duración cada vez que se requiere un nuevo dato en sample\_in.
- jack\_sd: Información de control para los operacionales de la etapa de audio mono. Dejaremos este valor estable en '1'.
- jack\_pwm: La señal PWM generada a partir del dato en sample\_in.

A nivel estructural, en este bloque se pueden distinguir tres componentes:

- Interfaz del micrófono. Encargado de gestionar los “recording ports”, excepto el micro\_clk.
- Interfaz de la salida de audio. Encargado de gestionar los “playing ports”.
- Generador de enables y salida de reloj del micrófono. Encargado de generar las señales enable que controlan las frecuencias de muestreo y de conversión de datos, así como la señal de reloj que alimenta al micrófono.

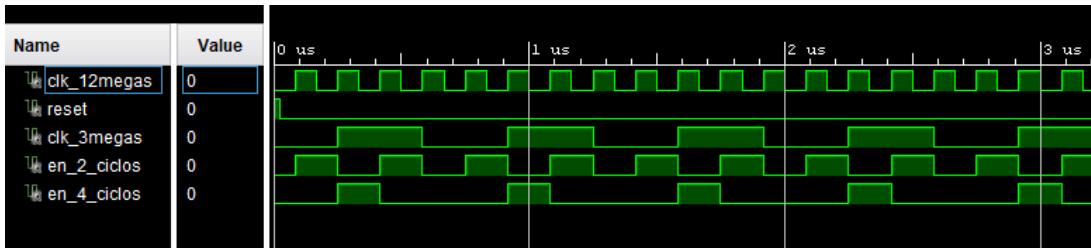
A continuación se describe cada uno de estos módulos y se guía el proceso de diseño.

### 5.1. Generador de enables y salida de reloj del micrófono

Este bloque recibe la señal del reloj global de 12 MHz y proporciona a sus salidas señales que van a ser utilizadas para temporizar el resto de módulos de la interfaz de audio. La entidad del módulo se define de la siguiente manera:

```
entity en_4_cycles is
  Port ( clk_12megas : in STD_LOGIC;
           reset : in STD_LOGIC;
           clk_3megas: out STD_LOGIC;
           en_2_cycles: out STD_LOGIC;
           en_4_cycles : out STD_LOGIC);
end en_4_cycles;
```

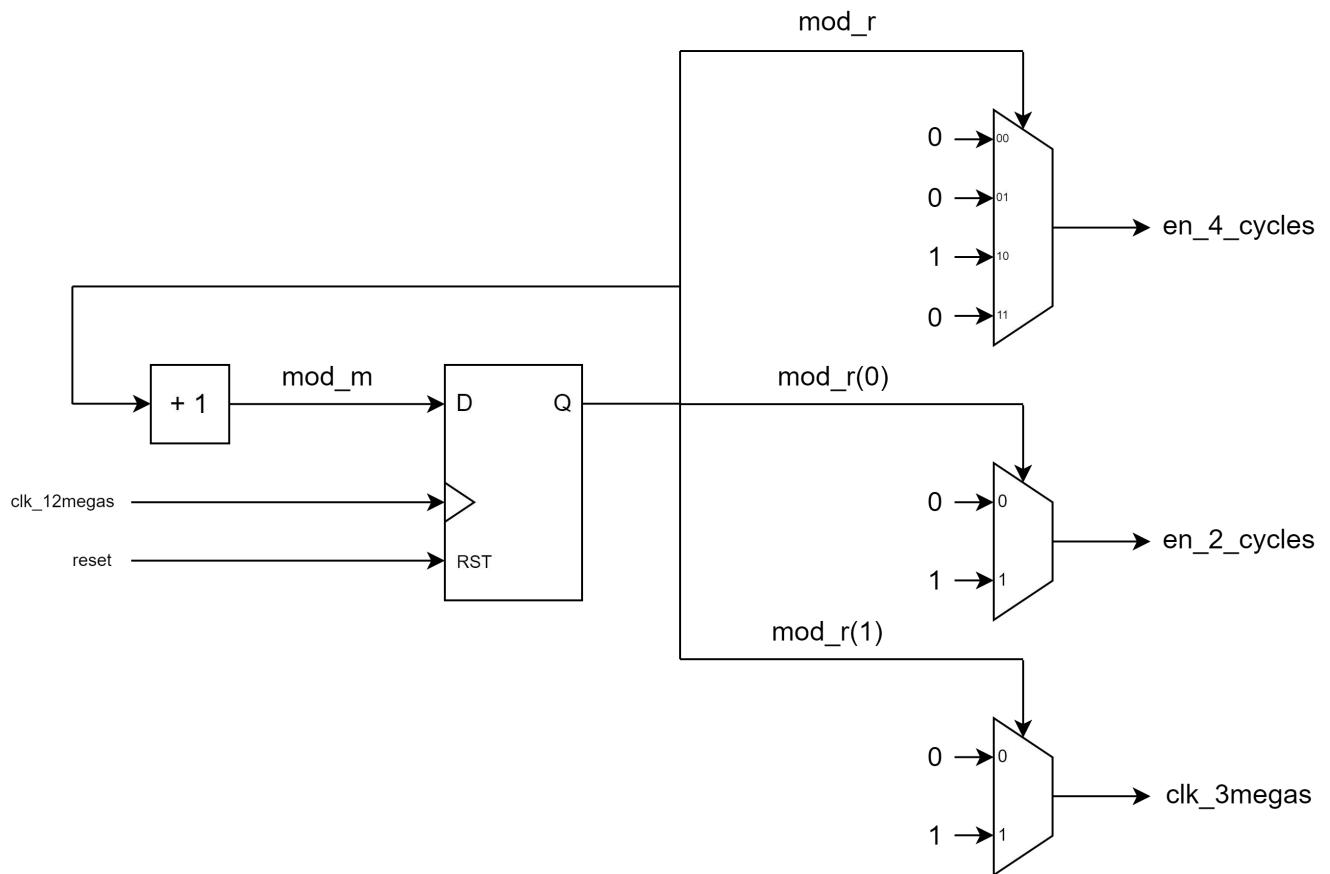
La salida clk\_3megas es utilizada por el micrófono y es un reloj de 3 MHz con un duty cycle del 50 %. La salida en\_2\_cycles es utilizada por la interfaz de salida de audio y proporciona una señal activa durante un ciclo de reloj cada dos ciclos (equivalente a un reloj de 6 MHz). La salida en\_4\_cycles es utilizada por la interfaz del micrófono y proporciona una señal activa durante un ciclo de reloj cada cuatro ciclos. La siguiente figura muestra el funcionamiento esperado.



### Tarea 1.1:



Diseña un circuito que sea capaz de generar las señales especificadas. Dibuja el esquemático correspondiente a tu diseño en el espacio inferior.





### Tarea 1.2:

Implementa tu diseño en VHDL.

Este diseño se encuentra implementado en *en\_4\_cycles.vhd*



### Tarea 1.3:

Genera un testbench que verifique la funcionalidad de tu diseño.

Las pruebas de este diseño se encuentran en *en\_4\_cycles\_tb.vhd*

## 5.2. Interfaz del micrófono

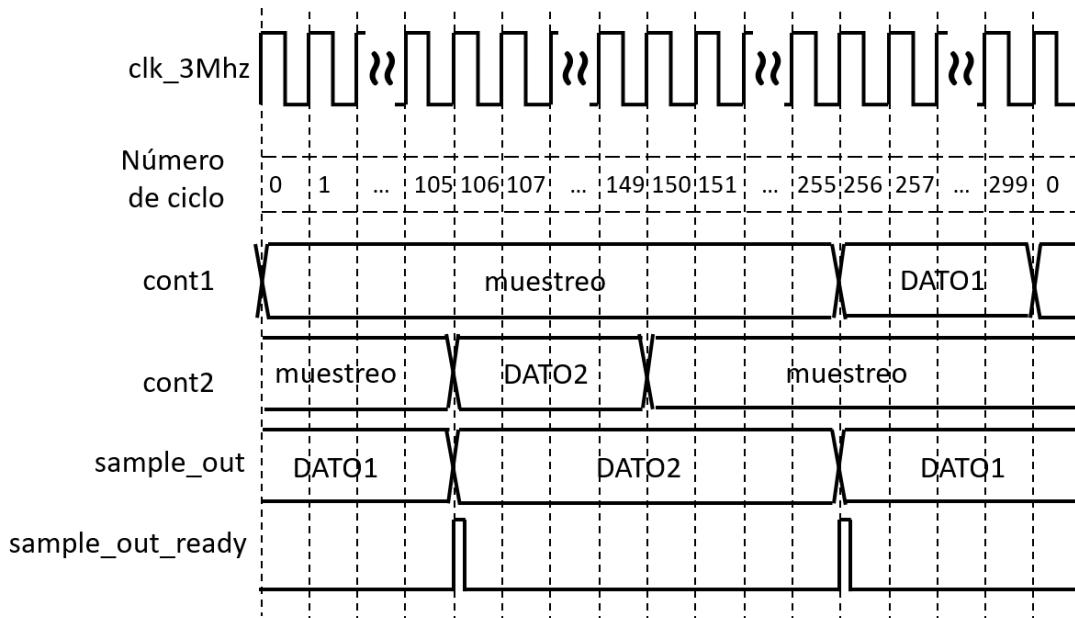
Para diseñar este módulo, es necesario conocer bien la interfaz que proporciona el micrófono omnidireccional de la placa Nexys4 DDR, la modulación de densidad de pulso (PDM) y la temporización de la interfaz del micrófono. Para ello se recomienda la lectura de la sección 15 del manual “Nexys4 DDR FPGA Board Reference Manual”.

### 5.2.1. Esquema de muestreo

El sistema va a proporcionar una frecuencia de 3 MHz en micro\_clk (333 ns de periodo) y necesita tomar una muestra cada  $50 \mu\text{s}$  ( $f_{sampling} = 20 \text{ kHz}$ ), es decir, una muestra cada 150 periodos de micro\_clk. Se va a utilizar un esquema de dos contadores similar al explicado en la figura 28 del manual de la placa.

El sistema va a trabajar con datos de 8 bits, es decir va a contar el número de unos que hay a lo largo de 256 muestras.

La siguiente imagen muestra el esquema de muestreo del sistema:



Según se muestra en la figura, hay un contador general de módulo 300 que numera los ciclos del 0 al 299. Los 300 ciclos se corresponden con  $100 \mu\text{s}$ , es decir el doble del periodo de muestreo. El primer contador muestrea entre los ciclos 0 y 255 y proporciona la señal digitalizada estable entre los ciclos 256 y 299. El segundo contador muestrea entre los ciclos 150 y 105 y proporciona

la señal digitalizada estable entre los ciclos 106 y 149. La salida sample\_out en el ciclo 105 se actualiza con el valor digitalizado por el contador 2 y en el ciclo 256 con el valor digitalizado por el contador 1, es decir se actualiza cada 150 ciclos o cada  $50 \mu\text{s}$ , la tasa de muestreo deseada. La salida sample\_out\_ready produce un pulso activo de un periodo del reloj general del sistema, de 12 MHz, cuando los nuevos datos digitalizados están disponibles.

### 5.2.2. Diseño e implementación

El diseño de la interfaz del micrófono se va a realizar empleando la metodología de FSMD: a partir de un pseudocódigo se va a generar un gráfico ASMD, este gráfico servirá como base para la descripción del módulo VHDL. Se recomienda revisar los capítulos 11 y 12 del libro “RTL Hardware Design Using VHDL” del profesor Pong P. Chu y las diapositivas de clase para familiarizarse con esta metodología. La entidad del módulo se define de la siguiente manera:

```
entity FSMD_microphone is
    Port ( clk_12megas : in STD_LOGIC;
            reset : in STD_LOGIC;
            enable_4_cycles : in STD_LOGIC;
            micro_data : in STD_LOGIC;
            sample_out : out STD_LOGIC_VECTOR (sample_size-1 downto 0);
            sample_out_ready : out STD_LOGIC);
end FSMD_microphone;
```

La señal enable\_4\_cycles es una señal de enable que se activa durante un periodo de reloj cada cuatro periodos. Proviene del generador del enables y permite trabajar a la interfaz del micrófono de manera efectiva a una velocidad de 3 MHz. El resto de puertos ya se ha descrito anteriormente.

Desde el esquema de muestreo, podemos describir el algoritmo de digitalización mediante el siguiente pseudo-código:

```
if (reset = 1)
    cuenta = 0
    dato1 = 0
    dato2 = 0
    primer_ciclo = 0
else
    if(0 <= cuenta <= 105 OR 150 <= cuenta <= 255)
        cuenta = cuenta + 1
        if(micro_data = 1)
            dato1 = dato1 + 1
            dato2 = dato2 + 1
    elsif(106 <= cuenta <= 149)
        cuenta = cuenta + 1
        if(micro_data = 1)
            dato1 = dato1 + 1
    if(primer_ciclo = 1 and cuenta = 106)
        sample_out = dato2
```

```

dato2 = 0
sample_out_ready = enable_4_cycles
else
    sample_out_ready = 0
else
    if(cuenta = 299)
        cuenta = 0
        primer_ciclo = 1
    else
        cuenta = cuenta + 1
    if(micro_data = 1)
        dato2 = dato2 + 1
    if(cuenta = 256)
        sample_out = dato1
        dato1 = 0
        sample_out_ready = enable_4_cycles
    else
        sample_out_ready = 0

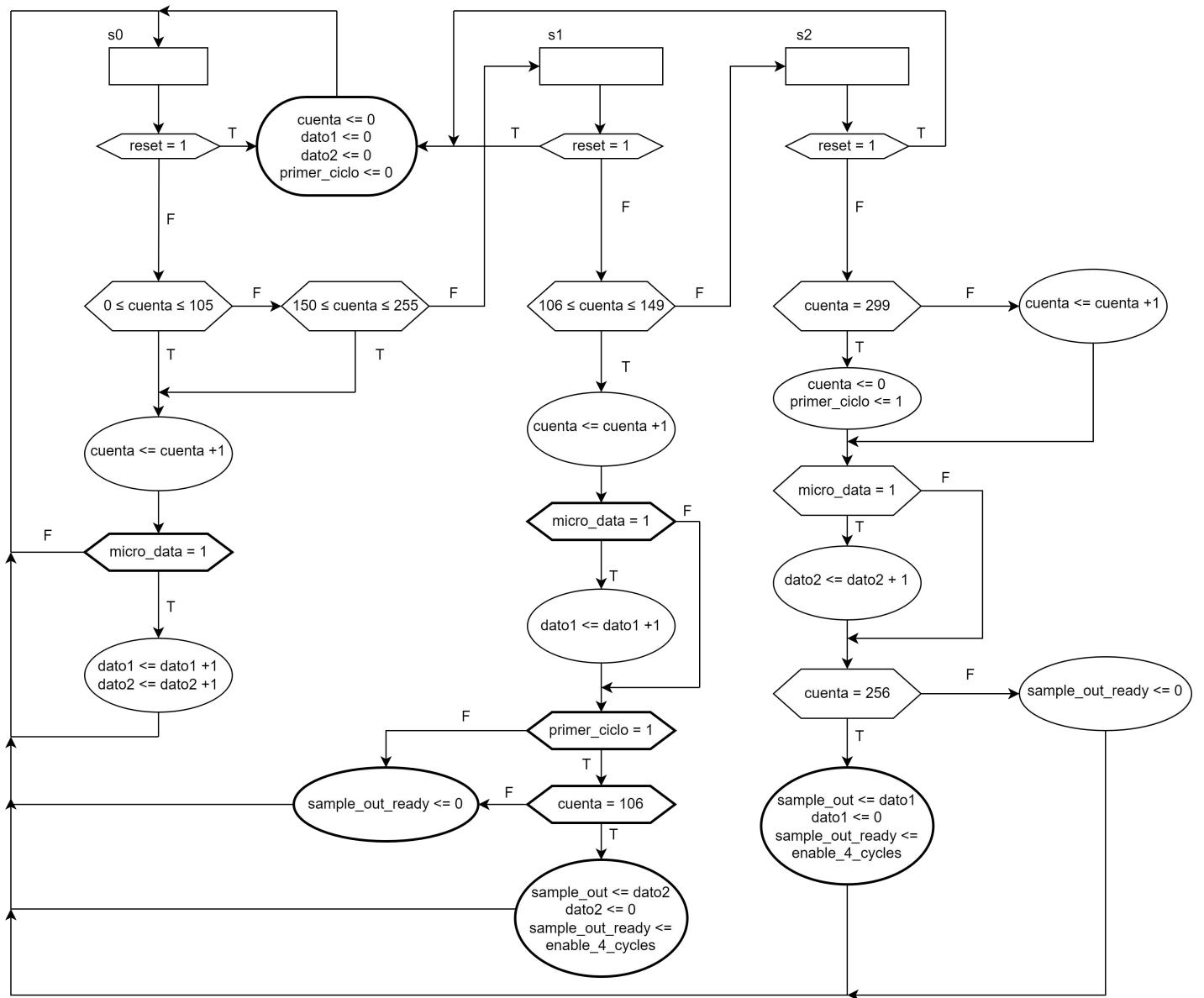
```

Trata de comprender el pseudo-código, si tienes dificultades, contacta con el profesor de la asignatura.



**Tarea 1.4:**

Realiza el diagrama ASMD que describa la misma funcionalidad que el pseudo-código anterior.





### Tarea 1.5:

Implementa el diagrama anterior en un fichero VHDL.

El diagrama anterior se encuentra implementado en *fsmd\_microphone.vhd*

#### 5.2.3. Estrategia de test

Antes de poder oír si el resultado de nuestra digitalización es correcto, necesitamos comprobar mediante testbenches que nuestro sistema funciona como esperamos. Podemos en un primer lugar comprobar que la máquina de estados recorre los estados como esperamos, generando las señales necesarias para reiniciar los contadores. Puedes instanciar el generador de enables en tu testbench para tener disponible la señal enable\_4\_cycles.



### Tarea 1.6:

Realiza un testbench que tenga la señal micro\_data fija a '1' para comprobar que las transiciones de estados, la selección de los datos de salida y la activación de sample\_out\_ready se produce correctamente.

En una segunda fase podemos comprobar que la digitalización se hace correctamente, para ello podemos introducir señales pseudo-aleatorias en la señal micro\_data y calcular el resultado esperado a mano. Podemos construir estas señales pseudo-aleatorias con sentencias del tipo:

```
a <= not a after 1300 ns;  
b <= not b after 2100 ns;  
c <= not c after 3700 ns;  
micro_data <= a xor b xor c;
```



### Tarea 1.7:

Realiza un testbench que introduzca una señal pseudo-aleatoria en micro\_data y comprueba que la digitalización se realiza correctamente.

Las pruebas de las tareas anteriores se encuentran en *fsmd\_microphone\_tb.vhd*

#### 5.3. Interfaz de la salida de audio

Para diseñar este módulo, es necesario conocer bien la interfaz que proporciona la salida de audio mono de la placa Nexys4 DDR y la modulación de anchura de pulso (PWM). Para ello se recomienda la lectura de la sección 16 del manual “Nexys4 DDR FPGA Board Reference Manual”.

Como los datos se han muestreado a 20 kmuestras/s, la tasa de generación de pulsos tiene que ser la misma: un pulso cada 50  $\mu$ s. Para la generación de la señal PWM se propone basarse en la sección 9.2.5 del libro “RTL Hardware Design Using VHDL” del profesor Pong P. Chu. En el esquema del libro, el valor de salida de un contador se compara con la palabra digital que se quiere convertir, si el valor del contador es más pequeño, la salida PWM se pone a '0' y a '1' en el caso contrario. En el caso de nuestro sistema, el contador va a aumentar su salida cada dos ciclos del reloj global, es decir, a una frecuencia de 6 MHz (166 ns de periodo). Es decir, que durante el periodo de muestreo de 50  $\mu$ s, el contador va contar de 0 a 299. Como nuestro dato digital es de 8 bits, el máximo valor representado es 255, por lo tanto todos los datos

que reproduczcamos van a tener una pequeña disminución de volumen por un factor de  $255/300 = 0.85$ .

La entidad del bloque está declarada de la siguiente forma:

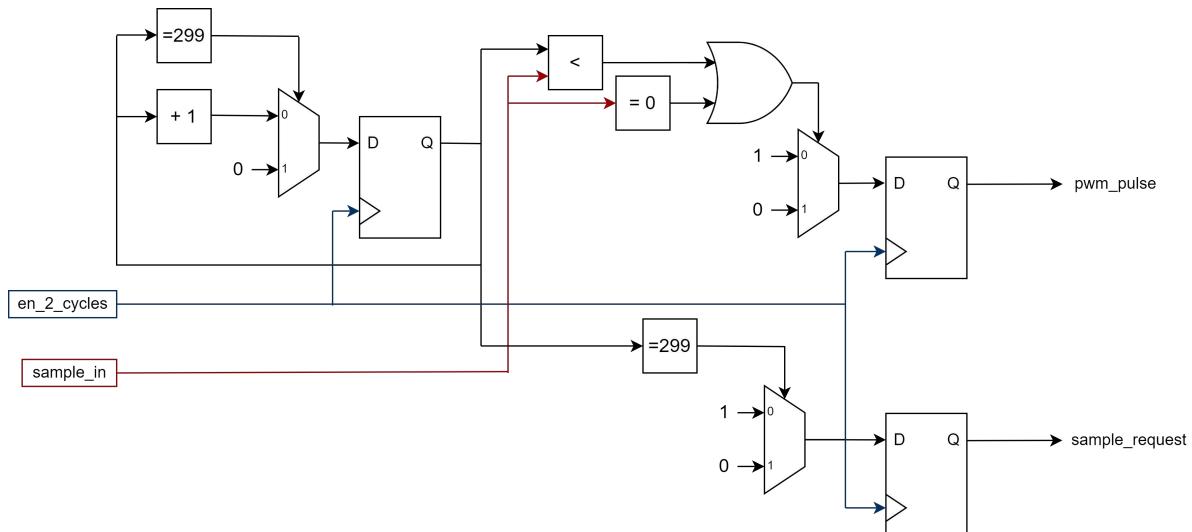
```
entity pwm is
  port(
    clk_12megas, in std_logic;
    reset, in std_logic;
    en_2_cycles: in std_logic;
    sample_in: in std_logic_vector(sample_size-1 downto 0);
    sample_request: out std_logic;
    pwm_pulse: out std_logic
  );
end pwm;
```

La señal `en_2_cycles` es una señal que se activa cada dos ciclos y es la que nos permite trabajar de manera efectiva a la mitad de la frecuencia. Cada vez que el módulo haya terminado una cuenta, esto es que haya llegado a 299 y vaya a pasar a 0, tiene que poner un pulso activo en `sample_request` de duración un sólo periodo del reloj general para solicitar una nueva muestra a la entrada.

#### Tarea 1.8:



Modifica el diseño propuesto en el libro del Dr. Chu para que sea compatible con las especificaciones de nuestro sistema. Es decir, que el contador cuente de 0 a 299 e incluya reset y enable y que el circuito produzca la salida `sample_request` en el momento apropiado y con la duración apropiada. Dibuja el esquemático de tu diseño en la parte inferior.



#### Tarea 1.9:



Implementa el esquemático anterior en un fichero VHDL.

El esquemático anterior se encuentra implementado en el archivo `pwm.vhd`

### 5.3.1. Estrategia de test

Antes de probar el módulo en la placa, necesitamos asegurarnos mediante simulaciones de que el funcionamiento es el esperado. En concreto, tenemos que comprobar que el contador cuente cada dos ciclos, que la cuenta se reinicia al llegar a 299 y que las señales `pwm_pulse` y `sample_request` son las esperadas. Para ello podemos verificar el funcionamiento introduciendo los valores extremos ("0000 0000" y "1111 1111") y algún valor aleatorio intermedio. Puedes instanciar el generador de enables en tu testbench para tener disponible la señal `enable_2_cycles`.

#### Tarea 1.10:



Realiza un testbench que verifique el funcionamiento de la interfaz de la salida de audio.

El testbench de la interfaz de salida se encuentra implementado en el archivo `pwm_tb.vhd`

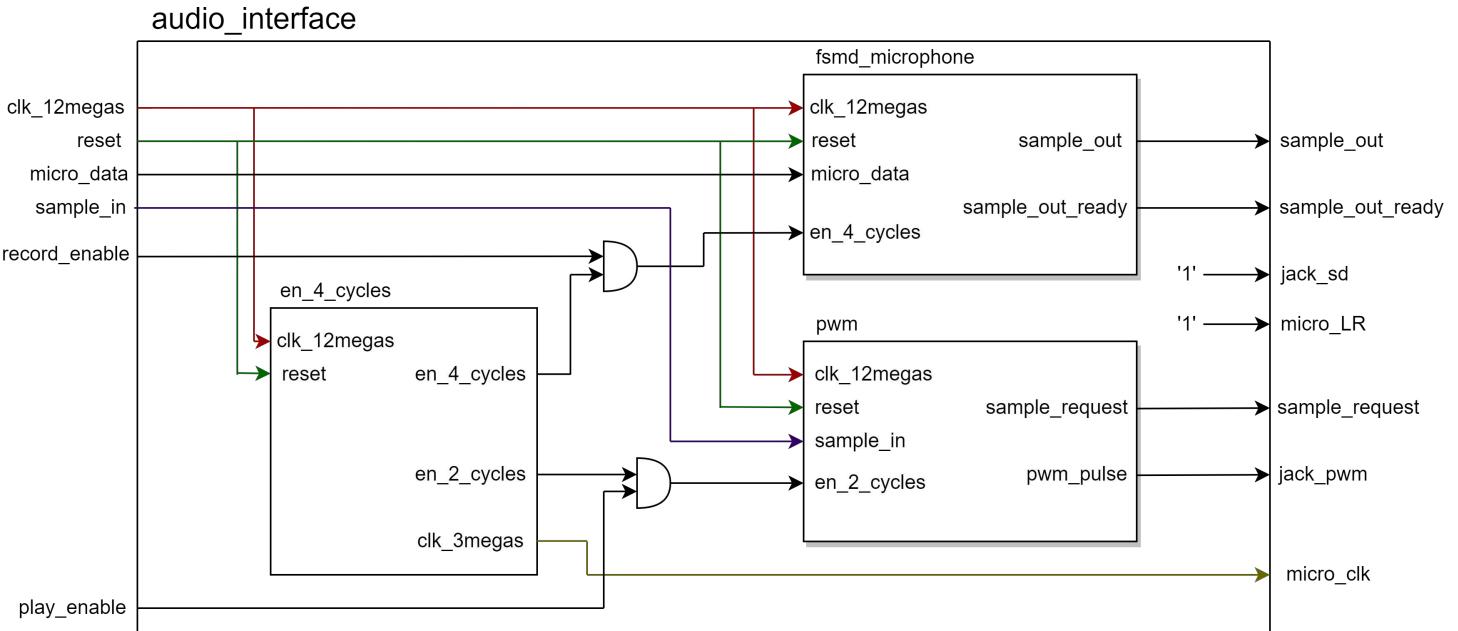
### 5.4. Integración de la interfaz de audio

Una vez implementados los tres módulos anteriores, el siguiente paso del diseño consiste en la instanciación de cada uno de estos bloques en un único módulo que formará la interfaz de audio completa.

#### Tarea 1.11:



Dibuja el esquemático a nivel bloque de la interfaz de audio completa. Ten en cuenta que `record_enable` y `play_enable` especifican cuándo están activas las interfaces del micrófono y de la salida de audio, respectivamente. Del mismo modo, tienes que asignar un '1' tanto a `micro_LR`, como a `jack_sd`.



### Tarea 1.12:



Implementa en VHDL la interfaz de audio completa. Para ello utiliza un estilo de código estructural que instancie cada uno de los tres bloques desarrollados anteriormente.

La arquitectura de la interfaz de audio completa se encuentra en *audio\_interface.vhd*

#### 5.4.1. Estrategia de test

Para verificar la funcionalidad del bloque completo, puedes reutilizar los estímulos empleados en los testbenches anteriores.

### Tarea 1.13:



Diseña un testbench que verifique la funcionalidad de la interfaz de audio completa.

El testbench de la interfaz de audio se encuentra en *audio\_interface\_tb.vhd*

#### 5.5. Implementación física y test en placa

Una vez verificada la funcionalidad del bloque completo mediante un testbench, vas a realizar la implementación física del bloque en la placa. Para poder comprobar que el sistema efectivamente digitaliza bien la señal del micrófono y transforma bien la palabra digital en un pulso PWM, el test en placa va a consistir en hacer que lo que se grabe por el micrófono salga por la salida de audio.

Para lograr esto, necesitas un pequeño controlador que genere la señal de 12 MHz a partir del reloj de 100 MHz de la placa, mantenga record\_enable y play\_enable a '1' y que introduzca la palabra digitalizada por la interfaz del micrófono (sample\_out) en la interfaz de la salida de audio (sample\_in). No necesitan ningún tipo de sincronización especial, mientras que sample\_in esté estable durante 50  $\mu$ s (no importa si cambia cuando el contador se pone a 0 o no). Este controlador tiene la siguiente declaración de entidad:

```
entity controlador is
  Port (
    clk_100Mhz : in std_logic;
    reset: in std_logic;
    --To/From the microphone
    micro_clk : out STD_LOGIC;
    micro_data : in STD_LOGIC;
    micro_LR : out STD_LOGIC;
    --To/From the mini-jack
    jack_sd : out STD_LOGIC;
    jack_pwm : out STD_LOGIC
  );
end controlador;
```

### Tarea 1.14:



Implementa en VHDL el controlador. Utiliza estilo estructural. Necesitas emplear el asistente arquitectural de Vivado para generar el reloj de 12 MHz a partir del reloj de 100 MHz.

La arquitectura implementada del controlador se encuentra en *controlador.vhd*

Se ha utilizado el *IP Catalog > Clocking Wizard* para generar un reloj de 12 MHz a partir del de 100 MHz.

**Tarea 1.15:**

Diseña un testbench que verifique la funcionalidad del controlador. Puedes utilizar los estímulos empleados en testbenches anteriores.

El testbench realizado para la entidad del controlador se encuentra en *controlador\_tb.vhd*

**Tarea 1.16:**

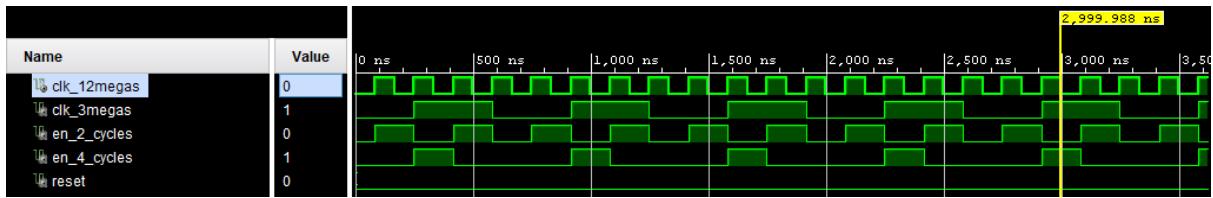
Escribe el fichero de restricciones .xdc. Sintetiza y realiza la implementación física del diseño. Genera el fichero .bit y programa la FPGA. Comprueba el funcionamiento correcto conectando unos auriculares a la salida de audio de la placa. Si la salida de audio es muy ruidosa, puedes jugar con el valor de micro\_LR y ponerlo a '0' en lugar de a '1'.

Guarda en un fichero de texto todos los mensajes de warning que hayan aparecido en el proceso anterior.

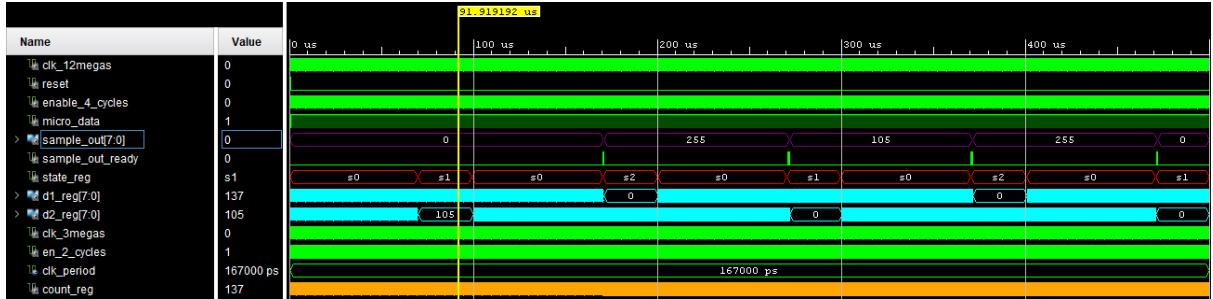
Avisa al profesor para comprobar el funcionamiento.

Puedes aprovechar para incluir pequeñas funcionalidades adicionales al controlador. Por ejemplo, puedes hacer un indicador de intensidad de señal grabada conectando 8 LEDs a la palabra digitalizada. También puedes hacer que el sistema sólo funcione cuando se aprieta uno de los botones.

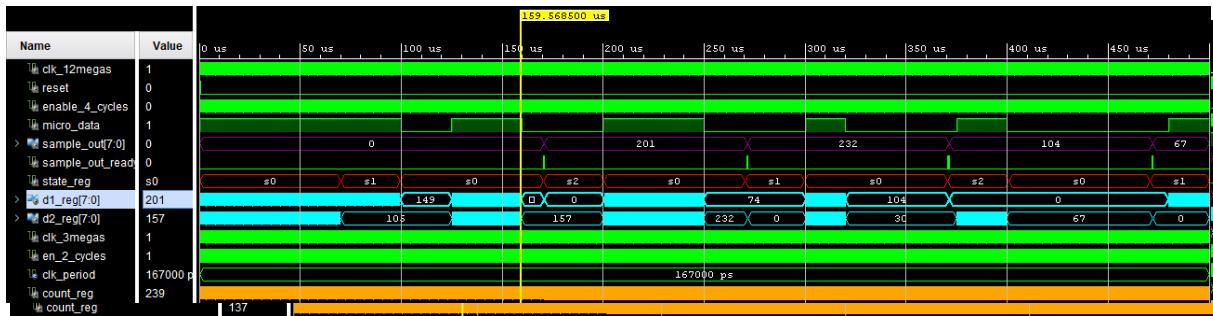
## Testbenches —— Tareas 1.3, 1.6, 1.7, 1.10, 1.13, 1.15



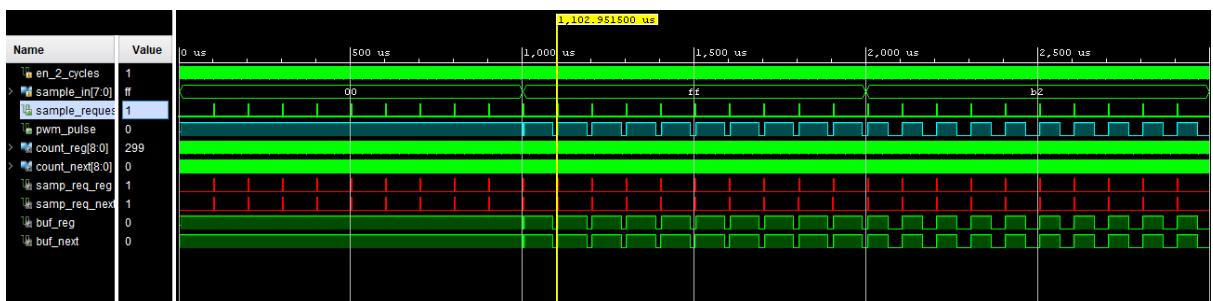
en\_4\_cycles\_tb.vhd



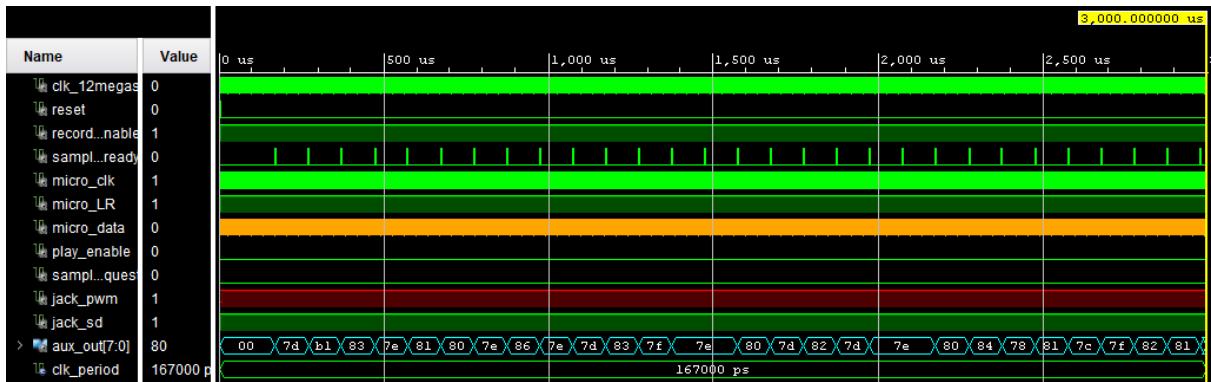
fsmd\_microphone\_tb.vhd (fixed micro\_data)



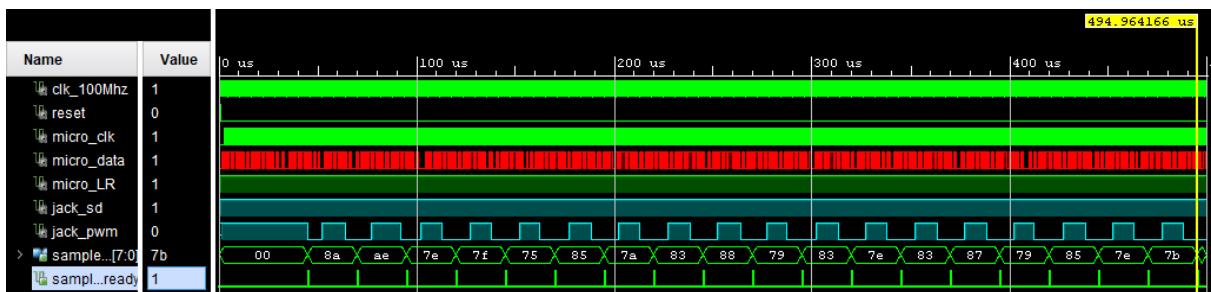
fsmd\_microphone\_tb.vhd (pseudo-random micro\_data)



pwm\_tb.vhd



audio\_interface\_tb.vhd



controlador\_tb.vhd

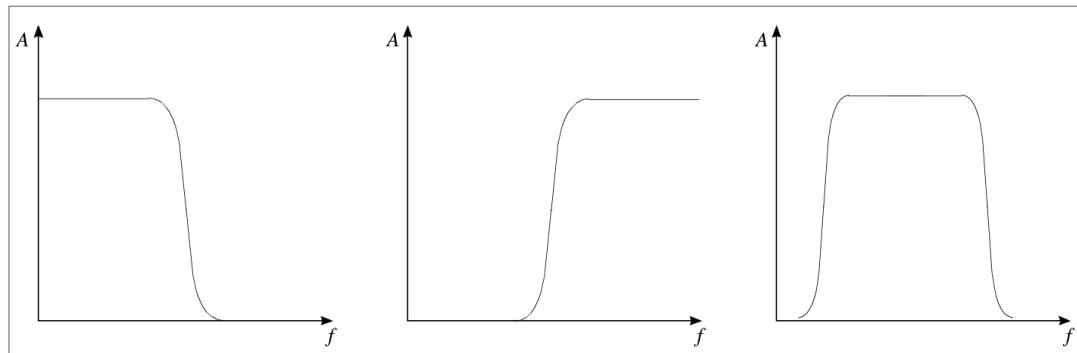
## 6. Bloque 2: Filtro FIR

Este bloque implementa un filtro FIR empleando VHDL y una arquitectura basada en una ruta de datos controlada por una máquina de estados finitos. El diseño de este bloque profundiza en conceptos de cuantificación, implementación de algoritmos, arquitecturas segmentadas y test-benches de complejidad media.

*Crédito:* Este bloque está basado en el “Mini-project” “FIR-Filter Design” del curso IL2204 “DSP Design usign HDL” del “Royal Institute of Technology” de Estocolmo, Suecia.

### 6.1. Filtros digitales

Hablamos de filtros en el contexto de procesado digital de señales para referirnos a circuitos que permiten pasar señales dentro de un determinado rango de frecuencias, mientras que bloquean a las señales que caigan fuera de este rango. El rango de frecuencias que el filtro permite pasar se conoce como banda de paso y el rango de frecuencias que no puede pasar se conoce como banda rechazada. Se pueden clasificar los filtros con respecto a la posición de las bandas de paso y rechazada: Tenemos filtros paso bajo, paso alto y paso banda, sus funciones de transferencia se ilustran en la siguiente figura.



Una manera común de expresar la relación de la salida del filtro  $y[n]$ , donde  $n$  es el número de muestra, en función de la entrada  $x[n]$  de un filtro, es la siguiente:

$$y[n] = \sum_{m=0}^{N-1} \alpha[m]x[n-m] - \sum_{m=1}^{N-1} \beta[m]y[n-m] \quad (1)$$

El primer término de la ecuación relaciona la salida del filtro con entradas pasadas. El segundo término relaciona la salida del filtro con salidas pasadas. Podemos distinguir dos tipos de filtros:

- Filtros de respuesta al impulso infinita (IIR) en los que están presentes los dos términos de la ecuación 1. También son conocidos como filtros de respuesta infinita, ya que el segundo término describe un comportamiento recursivo (la salida es función de la salida).
- Filtros de respuesta al impulso finita (FIR) en los que sólo está presente el primer término de la ecuación 1. La señal de salida de un filtro FIR es una suma ponderada de la señal de entrada. La respuesta al impulso de un filtro FIR tiene una longitud de  $N$  datos de salida.

Podemos destacar las siguientes propiedades de los filtros FIR:

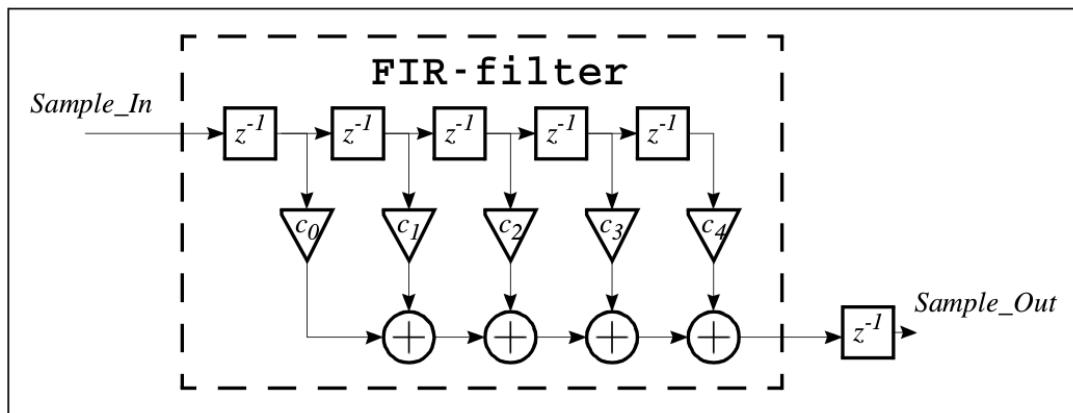
- La respuesta al impulso unidad de un filtro FIR es la secuencia de sus coeficientes  $\alpha[m]$ . Esto nos proporciona una manera sencilla de comprobar el funcionamiento del filtro. Lo único que tenemos que hacer es introducir una delta unidad ( $x(0)=1$ ,  $x(n)=0$  para  $n$  distinto de 0, también conocida como delta de Dirac) y obtendremos la secuencia de los coeficientes.
- Si todos los coeficientes se multiplican por una constante, se cambia la ganancia del filtro, no sus características.
- La ganancia en continua del filtro equivale a la suma de todos sus coeficientes.

## 6.2. Implementación hardware de filtros FIR

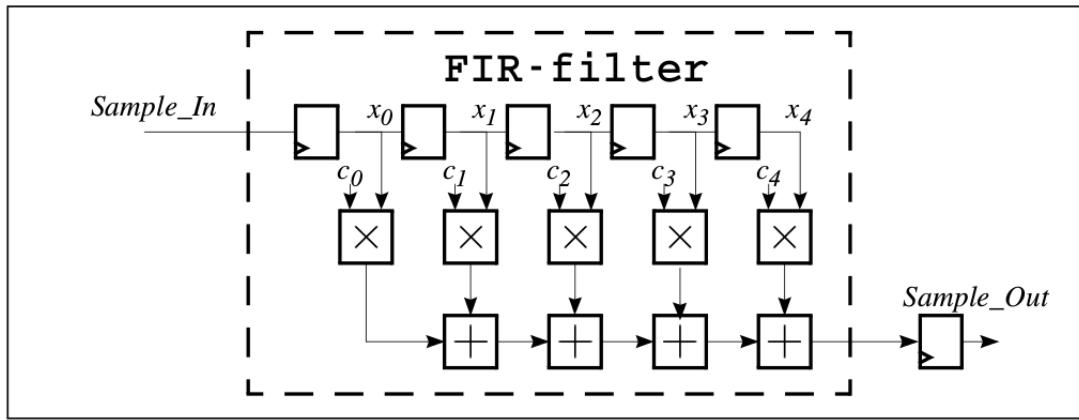
La transformada Z de un filtro FIR viene dada por la siguiente expresión

$$H_{FIR}(z) = \sum_{n=0}^{N-1} c_n z^{-n} \quad (2)$$

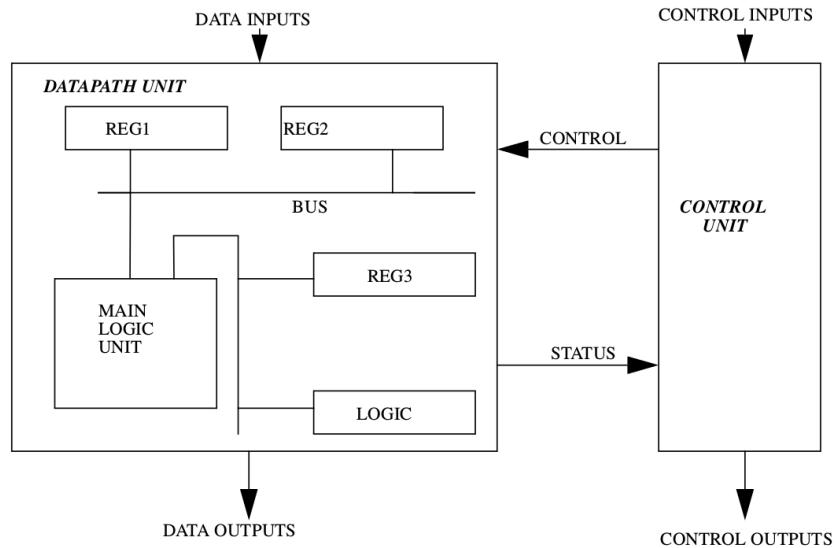
Donde  $z^{-n}$  en la ecuación corresponde a líneas de retardo y  $c_n$  son los coeficientes que ponderan las muestras de entrada, equivalentes a las  $\alpha[m]$  de la ecuación 1. En la ecuación anterior  $N$  es el orden del filtro, que también se corresponde como el número de etapas del filtro. El diagrama del flujo de señal de un filtro FIR de 5 etapas se muestra en la siguiente figura. Observa que la última línea de retardo (el registro a la salida) no forma parte de la ecuación anterior.



Las  $z^{-n}$ , correspondientes a los retardos, se implementan empleando registros. Para aplicar los coeficientes a cada muestra se emplean multiplicadores. Las sumas parciales se hacen a través de sumadores. La siguiente figura muestra la implementación directa de un filtro FIR de 5 etapas.



### 6.3. Sistemas basados en rutas de datos



La mayoría de los sistemas digitales se basa en una ruta de datos y un controlador (normalmente en forma de una máquina de estados finitos). La ruta de datos consiste en unidades de almacenamiento (como flipflops, registros y memorias) y unidades combinacionales (como ALUs, multiplexores, desplazadores, comparadores, etc.) conectados mediante buses (líneas de varias interconexiones). Los valores guardados en las unidades de almacenamiento se leen después del flanco de subida (o de bajada) de reloj, se procesan en las unidades combinacionales y se guardan en las unidades de almacenamiento en el siguiente flanco de reloj. Por otro lado, el controlador proporciona información de control y recibe información de estado de la ruta de datos. Normalmente se implementa a través de una máquina de estados finitos controlada por entradas y salidas del mundo exterior. La primera etapa en el diseño de un sistema basado en una ruta de datos es la descripción algorítmica de la especificación funcional de un sistema digital. A partir de esta descripción algorítmica se construye la ruta de datos.

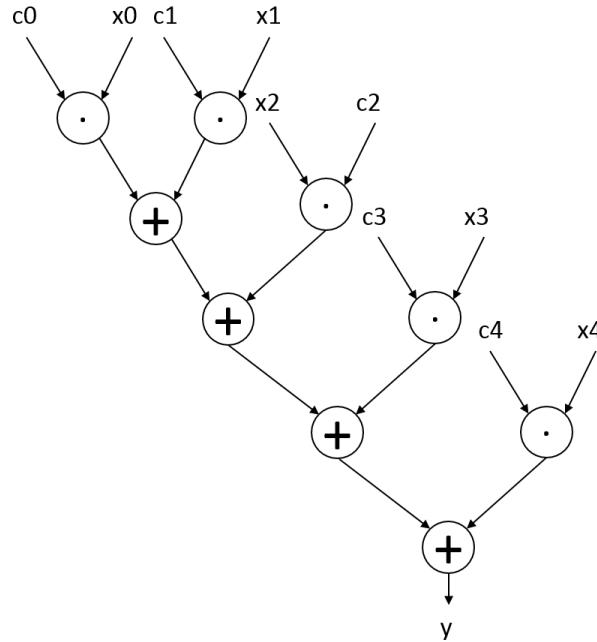
## 6.4. Metodología de implementación de algoritmos de procesado de señal

El proceso general de implementación de algoritmos para procesado digital de señales se suele dividir en las siguientes etapas:

- Asignación
- Planificación temporal
- Vinculación
- Implementación

El proceso no es estrictamente lineal, sino que puede que tengamos que retroceder alguna etapa para realizar alguna optimización concreta, o puede que podamos tomar decisiones sobre etapas posteriores sin haber finalizado algún paso intermedio.

Para ilustrar cada una de estas etapas, vamos a emplear el caso del filtro FIR de cinco etapas visto en las secciones anteriores que se puede representar por el siguiente grafo o diagrama de flujo.



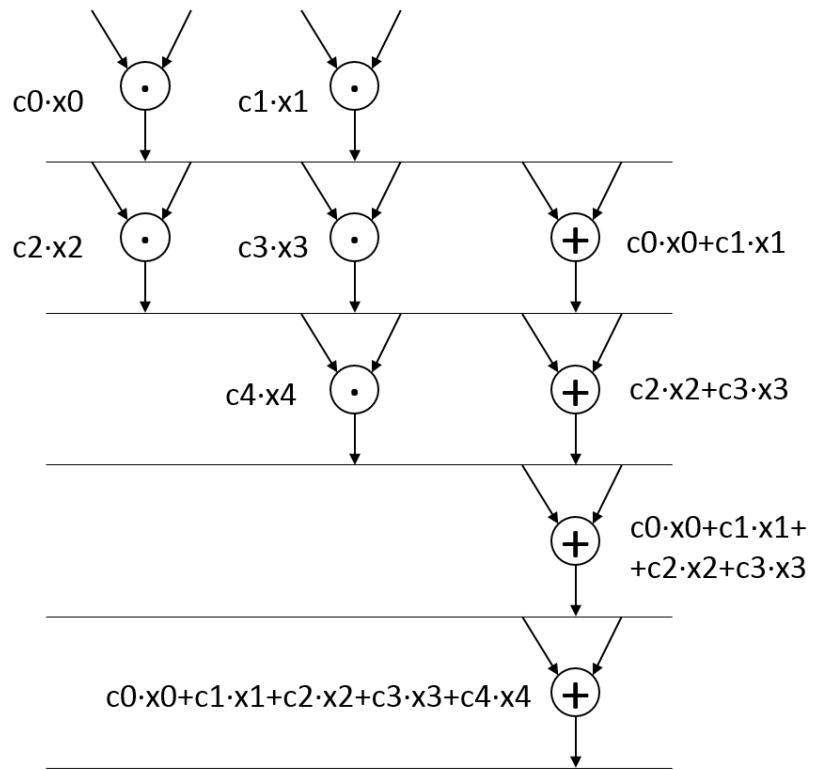
### 6.4.1. Asignación

Designamos asignación a la selección del número y tipo de unidades hardware que van a implementar las funciones descritas por el diagrama de flujo del algoritmo. En el caso del filtro FIR de cinco etapas, podríamos decidir implementarlo con un multiplicador y un sumador, dos multiplicadores y un sumador, dos medios multiplicadores y dos sumadores, etc. En función del número y tipo de unidades que se escojan se obtendrán unos valores distintos de área, throughput y latencia, con lo cual hay que tener en cuenta las restricciones del sistema para tomar esta decisión.

#### 6.4.2. Planificación temporal

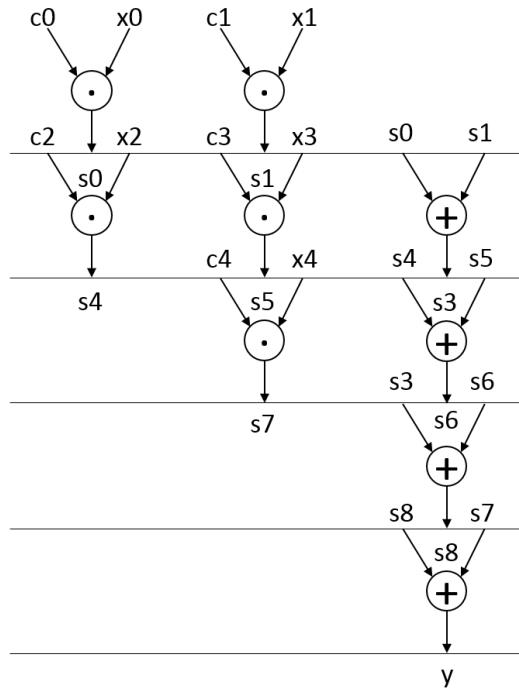
Durante esta fase, se distribuyen las operaciones del diagrama de flujo en el tiempo teniendo en cuenta que no se pueden exceder el número de unidades disponibles. En este momento se puede jugar con las propiedades distributivas de los operandos suma y multiplicación para reordenar el diagrama de flujo y obtener estructuras más interesantes desde el punto de vista hardware.

Fijando una asignación de dos multiplicadores y un sumador, la siguiente figura muestra la planificación temporal del diagrama de flujo del filtro FIR. Cada línea horizontal separa las operaciones que se producen durante un ciclo de control del siguiente. En este caso, para reducir la latencia del algoritmo, en lugar de hacer  $((c_0 \cdot x_0 + c_1 \cdot x_1) + (c_2 \cdot x_2) + (c_3 \cdot x_3))$ , que desaprovecharía la opción de utilizar dos multiplicadores simultáneamente, se ha propuesto  $((c_0 \cdot x_0 + c_1 \cdot x_1) + (c_2 \cdot x_2 + c_3 \cdot x_3))$ .



#### 6.4.3. Vinculación

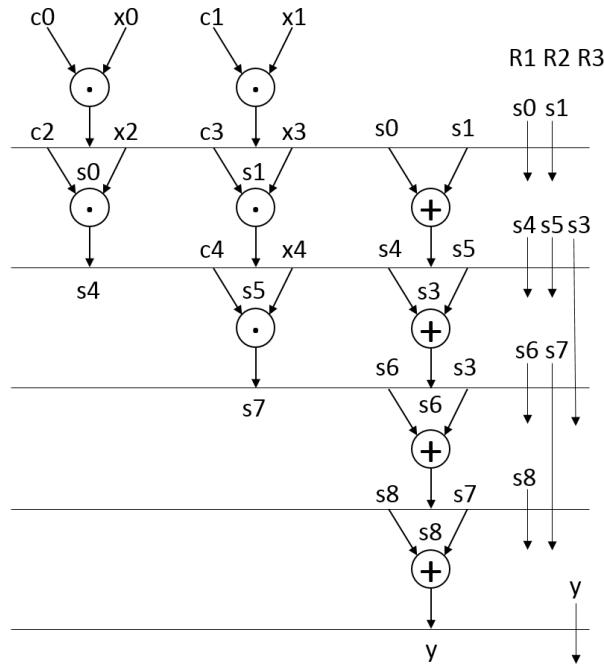
Cada unidad concreta se asocia a una operación concreta en cada unidad de tiempo. También se especifica a qué puerto concreto se conecta cada operando. Las señales internas se nombran para facilitar los procedimientos de las siguientes fases. La siguiente figura muestra el resultado de esta fase para el filtro FIR.



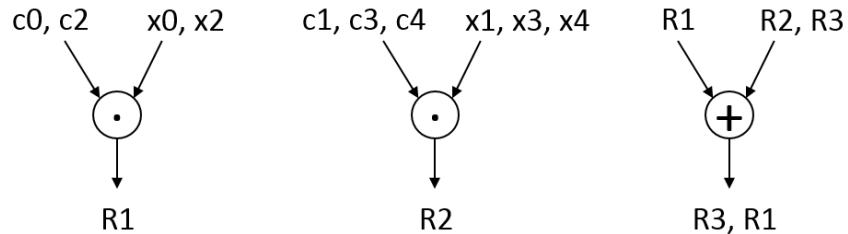
#### 6.4.4. Implementación

Esta es la fase final del diseño y es en la que se obtiene como resultado una estructura hardware que podemos implementar, por ejemplo, en una FPGA. Esta etapa se puede subdividir en distintas tareas: Análisis del tiempo de vida de las variables, minimización del número de registros y multiplexores y creación de la estructura hardware.

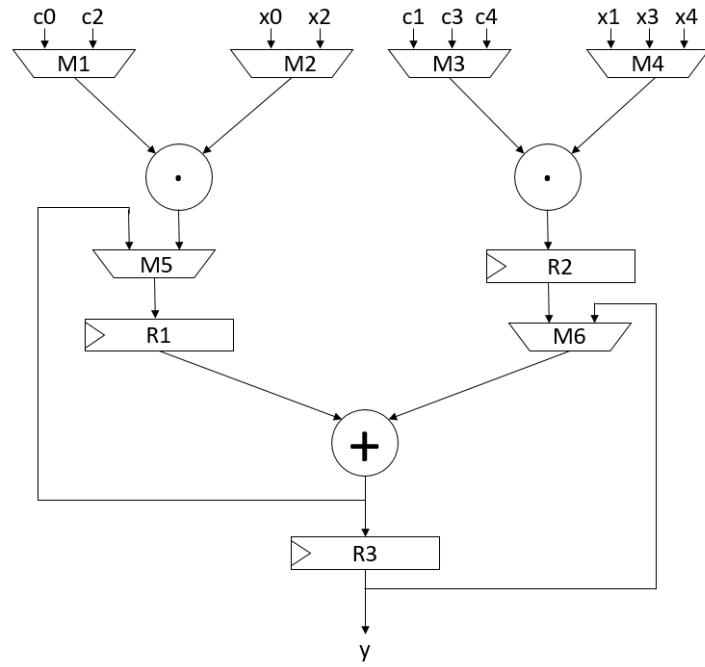
En el análisis del tiempo de vida de las variables se estudia la necesidad de almacenamiento interno (registros) de la estructura que vamos a generar. A partir del diagrama generado después de la vinculación, podemos analizar los momentos en que se producen las señales internas y los momentos en que se consumen, obteniendo así su tiempo de vida. En la siguiente figura se ha representado mediante una flecha el tiempo de vida de las distintas variables. El máximo número de flechas paralelas, tres en el ejemplo, establece el número de registros necesarios. Una vez establecido el número de registros necesarios,  $R_1, R_2$  y  $R_3$ , podemos asignar cada variable a un registro para cada momento. En el ejemplo, en el registro  $R_0$  se irán guardando  $s_0, s_4, s_6$  y  $s_8$  según se desenvuelva el algoritmo.



Cada entrada de cada operador y registro recibe distintos datos a lo largo del tiempo, es por ello que tenemos que asignar multiplexores que controlen qué entrada se conecta en cada momento. La siguiente figura estudia las posibles conexiones que pueden necesitar los módulos aritméticos del filtro FIR.



Finalmente, con toda la información que hemos ido ganando en las distintas etapas, ya podemos realizar una estructura hardware que sea capaz de realizar el algoritmo de procesado digital con las unidades que hemos elegido en la etapa de asignación. La siguiente estructura muestra la implementación del filtro FIR de cinco etapas con dos multiplicadores y un sumador.



Hasta aquí llegan los pasos para la creación de la ruta de datos, como es lógico, se espera que un controlador externo se encargue de proporcionar las señales de control de los multiplexores y los load de los registros en el momento apropiado. Para implementar el controlador, viene muy bien un cronograma que especifique en cada instante de tiempo qué señal tiene que dejar pasar cada multiplexor y qué señal tiene que tener guardada cada registro. La siguiente figura muestra el cronograma para el ejemplo del filtro FIR. En cada instante se especifica qué señal tiene que dejar pasar cada multiplexor y la señal de control asociada entre paréntesis, así como los datos guardados en los registros para cada instante de tiempo.

	t1	t2	t3	t4	t5	t6
M1	C0 (0)	C2 (1)				
M2	X0 (0)	X2 (1)				
M3	c1 (00)	c3 (01)	c4 (10)			
M4	x1 (00)	x3 (01)	x4 (10)			
M5	s0 (1)	s4 (1)	s6 (0)	s8 (0)		
M6		s1 (0)	s5 (0)	s3 (1)	s7 (0)	
R1	s0	s4	s6	s8		
R2	s1	s5	s7	s7		
R3		s3	s3			y

## 6.5. Notación en punto fijo

En este bloque vamos a trabajar con números representados en notación en punto fijo. Se puede establecer una analogía entre los números decimales en punto fijo y los binarios en punto

fijo. Un número binario en punto fijo tiene el siguiente aspecto: "10001110.101". El punto de un número binario en punto fijo se pone a la derecha de cifra que representa  $2^0$ . Las cifras a la izquierda del punto se interpretan como un binario sin punto y las cifras a la derecha representan la parte decimal o fraccionaria. La siguiente figura muestra dos ejemplos de notación en punto fijo empleando complemento a dos. Observa la figura y asegúrate de entender el procedimiento de conversión entre binario en punto fijo y base 10.

<table border="1"> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	= 110,34765625 <sub>dec</sub>
0	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1		
$S \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0, \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad 2^{-5} \quad 2^{-6} \quad 2^{-7} \quad 2^{-8}$																	
<table border="1"> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	1	0	0	0	1	0	0	0	= - 1,875 <sub>dec</sub>								
1	0	0	0	1	0	0	0										
$S \quad 2^0, \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4} \quad 2^{-5} \quad 2^{-6}$																	

Para este bloque vamos a utilizar la siguiente notación para representar los tamaños de números en punto fijo:  $< a.b >$  donde a es el número de bits que representan la parte entera y b es el número de bits necesarios para representar la parte decimal. Por ejemplo el número superior de la anterior figura necesita 16 bits, y su tamaño estaría representado por  $< 8,8 >$ . Con esta notación nos es útil para establecer los tamaños resultantes de las operaciones de multiplicación y suma: Multiplicación: Si  $< 3,4 >$  se multiplica con  $< 5,6 >$  el resultado será  $< 3+5,4+6 > = < 8,10 >$ . Si a la salida tenemos que proporcionar un dato de 10 bits, truncamos el resultado y obtenemos  $< 8,2 >$  (o  $< 7,3 >$  si sabemos que el resultado final de acumular estos números no va a producir un desbordamiento). Suma: Si  $< 3,4 >$  se suma a  $< 5,6 >$  el resultado será  $< \max(3,5) + 1, \max(4,6) > = < 6,6 >$ . El +1 en la parte entera tiene en cuenta posibles desbordamientos (acarreos de salida).

### Tarea 2.1:



¿Cuál es el rango de un número de 8 bits en punto fijo  $< 1,7 >$  en complemento a dos?  
¿Cuál es su precisión?

**El rango será desde  $-2^a$  hasta  $2^a - 2^{-b}$ , por tanto desde -1 hasta 0.9921875. La precisión será de  $2^{-7} = 0.0078125$ .**

## 6.6. Especificación del bloque

En este bloque vamos a implementar un filtro FIR de 5 etapas, cuya interfaz se describe en la siguiente entidad de VHDL:

```
entity fir_filter is
  Port ( clk : in STD_LOGIC;
         Reset : in STD_LOGIC;
         Sample_In : in signed (sample_size-1 downto 0);
         Sample_In_enable : in STD_LOGIC;
         filter_select: in STD_LOGIC; --0 lowpass, 1 highpass
         Sample_Out : out signed (sample_size-1 downto 0);
         Sample_Out_ready : out STD_LOGIC);
end fir_filter;
```

- Tanto Sample.In, Sample.Out, como los coeficientes del filtro se tienen que codificar con palabras de 8 bits en complemento a dos <1,7>.
- Sample.In\_enable es una entrada de control que informa de cuándo se ha actualizado el valor de Sample.In con un pulso activo durante un ciclo de reloj.
- filter\_select es una entrada de control que selecciona el filtrado que se va a realizar: '0' para el filtro de paso bajo, '1' para el filtro de paso alto.
- Sample.Out\_ready es una salida de control que informa de cuándo se ha actualizado el valor de Sample.Out con un pulso activo durante un ciclo de reloj.
- El filtro paso bajo utiliza los siguientes coeficientes:  $c_0 = c_4 = +0.039$ ,  $c_1 = c_3 = +0.2422$ ,  $c_2 = +0.4453$ .
- El filtro paso alto utiliza los siguientes coeficientes:  $c_0 = c_4 = -0.0078$ ,  $c_1 = c_3 = -0.2031$ ,  $c_2 = +0.6015$ .
- La implementación del filtro emplea una estructura de ruta de datos controlada por un controlador.
- La ruta de datos sólo emplea un multiplicador en pipeline y un sumador.

## 6.7. Tareas prácticas

### 6.7.1. Creación de la ruta de datos

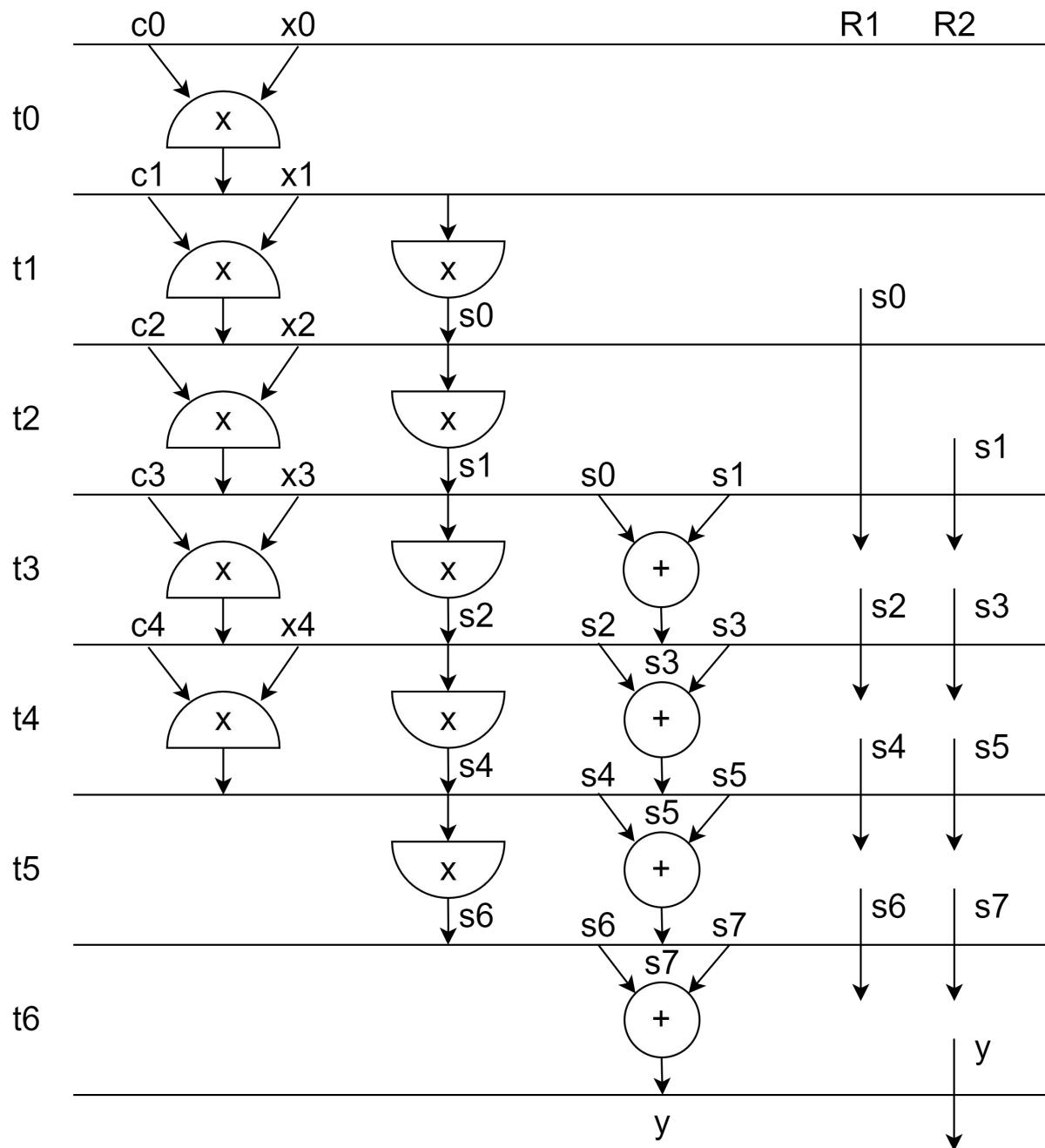
#### Tarea 2.2:



Realiza sobre papel la implementación de un filtro FIR de 5 etapas con la siguiente asignación: Dos medios multiplicadores y un sumador. Realiza todos los pasos descritos en la sección 6.4.



#### Planificación temporal:



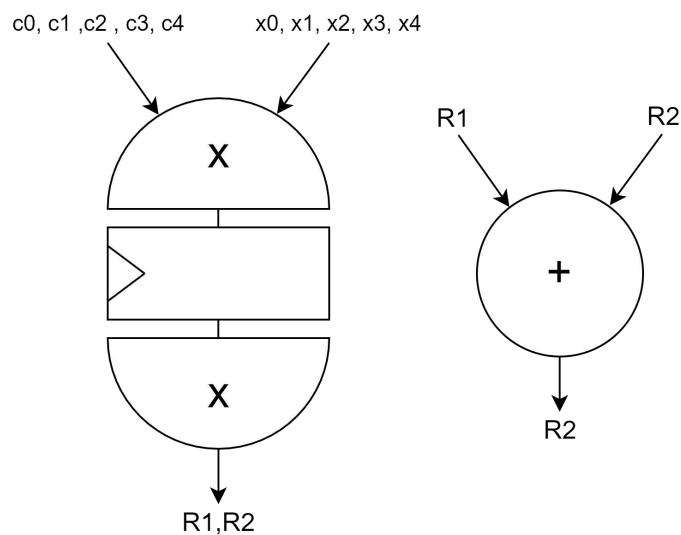


**Vinculación, análisis de tiempo de vida de variables y asignación de registros:**

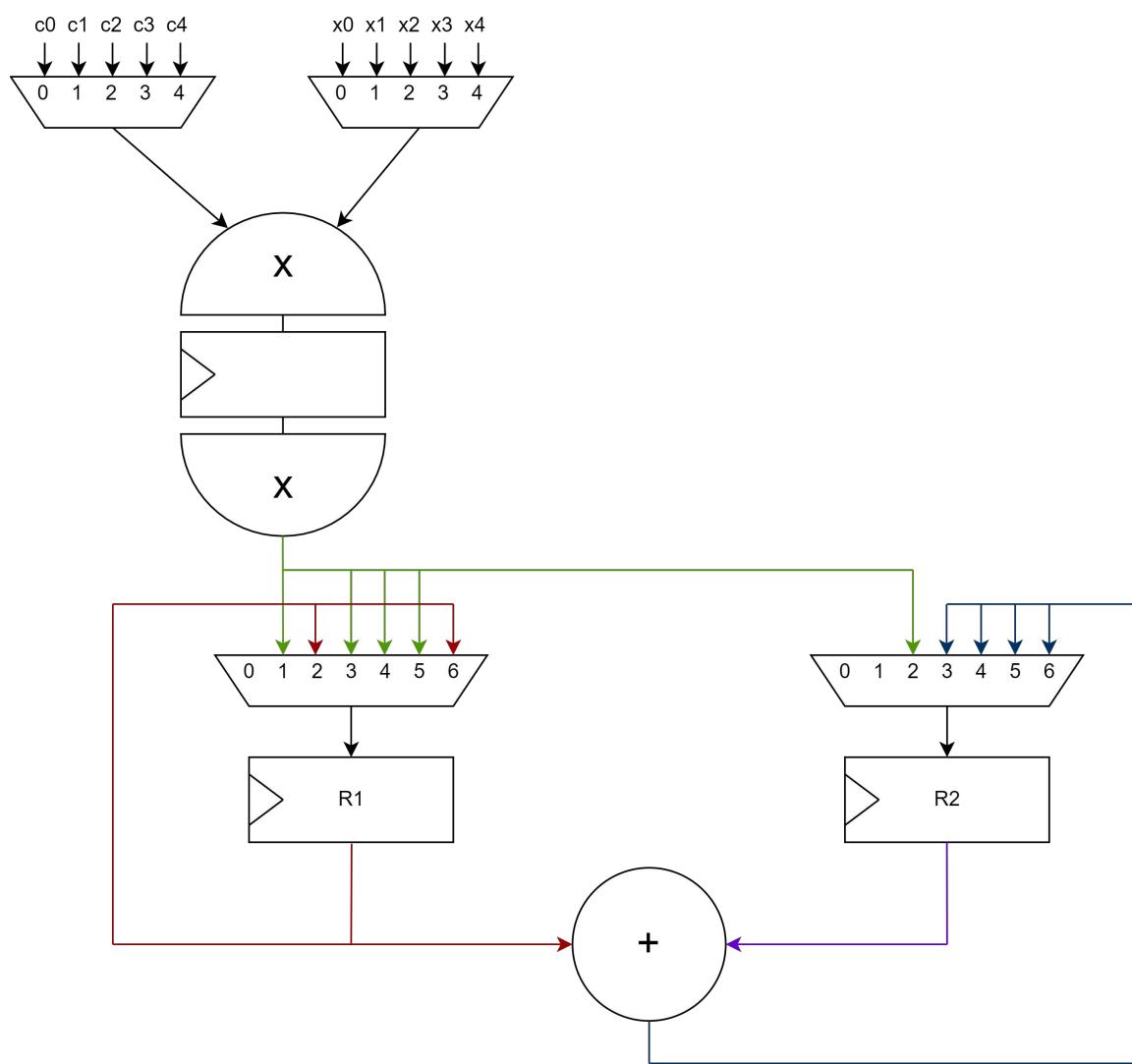
Está añadido en el apartado anterior.



## Análisis de conexiones para la extracción de multiplexores:



## Implementación:





### Cronograma:

	t0	t1	t2	t3	t4	t5	t6
M1	c0	c1	c2	c3	c4		
M2	x0	x1	x2	x3	x4		
R1		s0	R1	s2	s4	s6	
R2			s1	s3	s5	s7	y

### Tarea 2.3:



Realiza un análisis de cuantificación de las señales, de manera que especifiques cuántos bits vas a emplear en cada señal y en qué posición va a estar el punto decimal. Emplea la notación y las metodologías presentadas en clase.

En el multiplicador siempre van a entrar dos señales de <1.7>, por tanto en la salida habrá una señal de <2.14>.

En el sumador vamos a hacer 5 sumas de <2.14>, por tanto la máxima salida será de <2+log<sub>2</sub>(5).14>, que redondeando será <5.14>.

Como necesitamos que la salida sea <1.7> truncaremos los 4 MSB de la parte entera y los 7 LSB de la parte decimal.

### Tarea 2.4:



Escribe modelos VHDL para todos los componentes que se emplean en tu ruta de datos. Los operadores aritméticos (suma y multiplicación) no necesitan describirse en un estilo estructural. Para conseguir el medio multiplicador, añade un registro extra a la salida de un multiplicador completo, de esta forma se comportará como un multiplicador perfectamente segmentado. La herramienta de síntesis se encargará de mover este registro al interior del multiplicador de manera óptima. Comprueba el funcionamiento correcto de tus componentes creando los correspondientes testbenches.

Se han creado las siguientes entidades necesarias para la programación en estilo estructural: *mux5*, *mux7* en *mux5.vhd* y *mux7.vhd* respectivamente

Los testbenches empleados han sido importados de la 2<sup>a</sup> sesión de laboratorio de la asignatura.

### Tarea 2.5:



Escribe un modelo estructural de la ruta de datos completa en VHDL. Crea un testbench para la ruta de datos.

La entidad de la ruta de datos completa se encuentra en el fichero *data\_flux\_fir.vhd*

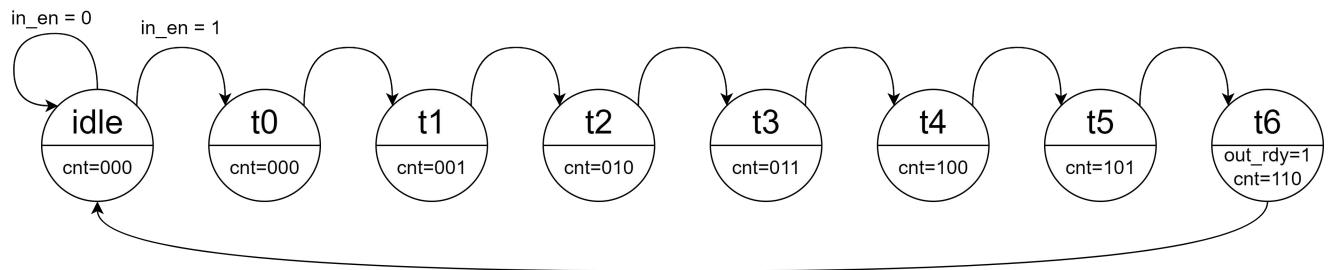
El testbench de la ruta de datos se encuentra en el fichero *data\_flux\_fir\_tb.vhd*

### 6.7.2. Creación del controlador

#### Tarea 2.6:



Dibuja el diagrama de estados que describe el controlador de tu filtro FIR. Sigue la metodología de implementación de máquinas de estados finitos vista en clase.





### Tarea 2.7:

Escribe el código VHDL correspondiente a tu controlador.

El código VHDL correspondiente a la FSM controladora se encuentra en el archivo *ctrl\_fir.vhd*



### Tarea 2.8:

Escribe el código VHDL estructural correspondiente a tu filtro completo.

El código VHDL correspondiente al filtro completo se encuentra en el archivo *fir\_filter.vhd*



### Tarea 2.9:

Escribe el código VHDL correspondiente a un testbench que introduzca un único impulso a la entrada de valor +0.5. ¿Qué secuencia esperas en Sample\_Out si en Sample\_In la secuencia es (0, 0, 0, 0, X, 0, 0, 0, 0)? Considera que X es el mayor/menor número positivo/negativo (cuatro casos) que se puede representar. Ten en cuenta la cuantificación de tus señales. Asegúrate de que tu diseño proporciona los valores esperados.

Partiendo de la operación del filtro FIR:

- 1) samp\_in = 0, samp\_out = 0
- 2) samp\_in = 0, samp\_out = 0
- 3) samp\_in = 0, samp\_out = 0
- 4) samp\_in = 0, samp\_out = 0
- 5) samp\_in = X, samp\_out = c0 \* X
- 6) samp\_in = 0, samp\_out = c1 \* X
- 7) samp\_in = 0, samp\_out = c2 \* X
- 8) samp\_in = 0, samp\_out = c3 \* X
- 9) samp\_in = 0, samp\_out = c4 \* X

Los números correspondientes a X y el resultado esperado se pueden observar en las páginas siguientes con los testbenches.



### Tarea 2.10:

Escribe el código VHDL correspondiente a un testbench que introduzca en la entrada la siguiente secuencia (0, 0.5, 0, 0.125, 0, 0, 0, 0, ...). ¿Qué secuencia esperas en Sample\_Out para esta secuencia de entrada? Asegúrate de que tu diseño proporciona los valores esperados.

Partiendo de la operación del filtro FIR:

$$\text{samp\_out} = \text{c0} * \text{x0} + \text{c1} * \text{x1} + \text{c2} * \text{x2} + \text{c3} * \text{x3} + \text{c4} * \text{x4};$$

En 8 ciclos:

1) x0 = 0					samp_out = 0
2) x0 = 0.5	x1 = 0				samp_out = 0.5 * c0
3) x0 = 0	x1 = 0.5				samp_out = 0.5 * c1
4) x0 = 0.125	x1 = 0	x2 = 0.5	x3 = 0,		samp_out = 0.125 * c0 + 0.5 * c2
5) x0 = 0	x1 = 0.125	x2 = 0	x3 = 0.5	x4 = 0	samp_out = 0.125 * c1 + 0.5 * c3
6) x0 = 0	x1 = 0	x2 = 0.125	x3 = 0	x4 = 0.5	samp_out = 0.125 * c2 + 0.5 * c4
7) x0 = 0	x1 = 0	x2 = 0	x3 = 0.125	x4 = 0	samp_out = 0.125 * c3
8) x0 = 0	x1 = 0	x2 = 0	x3 = 0	x4 = 0.125	samp_out = 0.125 * c4

El resultado expresado en coma fija de las operaciones anteriores se puede comprobar que se realiza correctamente en las páginas siguientes con los testbenches.

### 6.7.3. Creación de un testbench avanzado

Si tuviéramos que hacer pruebas más exhaustivas para comprobar el funcionamiento de nuestro filtro, el empleo de los testbenches que hemos usado hasta ahora dejaría de ser práctico, principalmente por dos razones:

- Cada vez que necesitamos probar una secuencia de entrada nueva, tenemos que modificar y recompilar el testbench.
- Para comprobar que la secuencia de salida que obtenemos es la correcta, tenemos que haberla calculado a mano previamente y, después de ejecutar el testbench, tenemos que comprobar uno a uno que los datos son correctos.

Para solucionar estos problemas podemos echar mano de las opciones de escritura y lectura de ficheros de VHDL. Los ficheros a los que se accede desde un código VHDL pueden o bien contener datos de un único tipo (std\_logic\_vector, integer, signed, etc.) o bien pueden ser de tipo TEXT. Nosotros nos vamos a ocupar únicamente de este último tipo de archivos, los que contienen datos de tipo TEXT, que son ficheros ASCII. Los ficheros de tipo TEXT se leen línea a línea empleando READLINE, que guarda la línea que corresponda en una señal o variable de tipo LINE. Los ficheros de tipo TEXT se escriben línea a línea empleando WRITELINE. Para extraer los datos contenidos en una señal/variable de tipo LINE se emplea READ, y la operación opuesta (escritura en un tipo LINE) se hace a través de WRITE. Tanto READ como WRITE funcionan con bit, bit\_vector, boolean, character, integer, real, string y time.

Matlab escribe y lee los datos en notación decimal, legible por nosotros. Por lo tanto tenemos que convertir los datos del fichero que leamos desde entero (**integer**) a binario (**signed**). Para ello vamos a emplear la siguiente función del paquete **IEEE.NUMERIC\_STD**:

- **my\_signed <= to\_signed(int\_number,8)** donde **int\_number** es el **integer** que queremos convertir, 8 es el número de bits que queremos que tenga nuestro vector y **my\_signed** es el **signed** al que se le asigna la conversión.

De la misma forma, para escribir en el fichero, tenemos que convertir nuestros datos, que son **signed** a **integer**. Para ello emplearemos la siguiente función:

- **my\_int <= to\_integer(signed\_number)** donde **signed\_number** es el **signed** que queremos convertir, y **my\_int** es el **integer** al que se le asigna la conversión.

A la hora de hacer la conversión entre **signed** y **integer** y viceversa, el programa no es capaz de trabajar con la representación  $<1,7>$  o con cualquier otra en punto fijo, siempre va a considerar la notación  $<8,0>$ , es decir, sin coma decimal. Esto implica que los datos del fichero estarán entre +127 y -128.

En el apéndice 1 puedes encontrar un ejemplo de testbench que lee de un fichero TEXT para que te hagas una idea de cómo funciona.

### Tarea 2.11:

Escribe el código VHDL correspondiente a un testbench que lea las muestras de un fichero y escriba los resultados en otro.



- Llama al fichero que contiene los datos de entrada “sample\_in.dat”; llama al fichero de salida “sample\_out.dat”.
- Rellena el fichero de entrada con una secuencia que introduzca un único impulso. Cada línea del fichero se corresponde con un dato.

### Tarea 2.12:

Utiliza el fichero “haha.wav” en Matlab para crear un fichero de entrada para tu testbench. En el apéndice 2 se detallan las secuencias que tienes que emplear en Matlab para:



- Cargar un fichero de audio en Matlab y crear otro fichero con el formato necesario para tu testbench.
- Utilizar la función *filter* de Matlab para obtener la respuesta de un filtro FIR con precisión real.
- Cargar y escuchar la salida que ha producido tu testbench en Matlab.

### Tarea 2.13:



Emplea tu testbench para procesar el fichero de entrada que te ha proporcionado Matlab. Escribe los datos filtrados en el fichero “sample\_out.dat”, que importarás posteriormente en Matlab.



### Tarea 2.14:

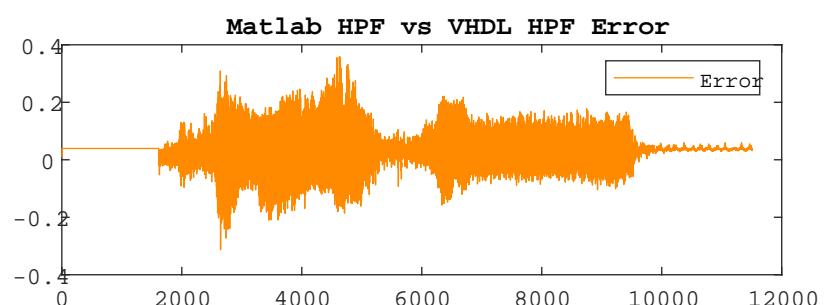
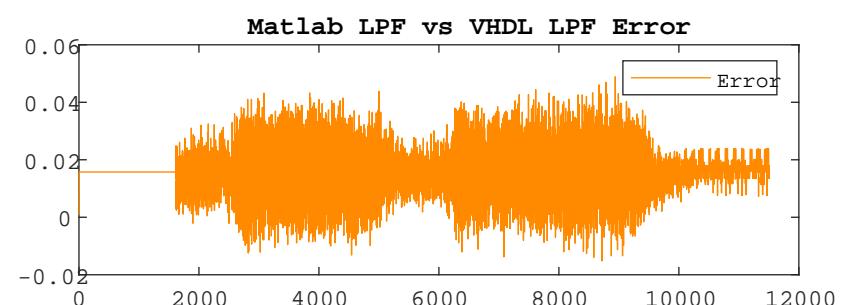
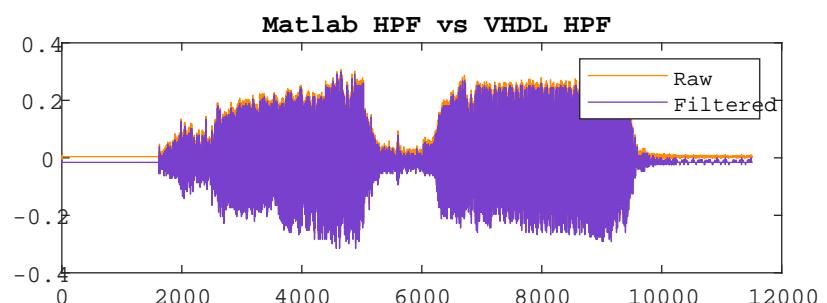
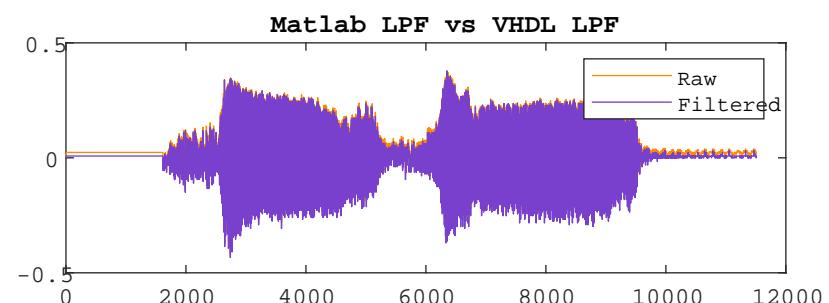
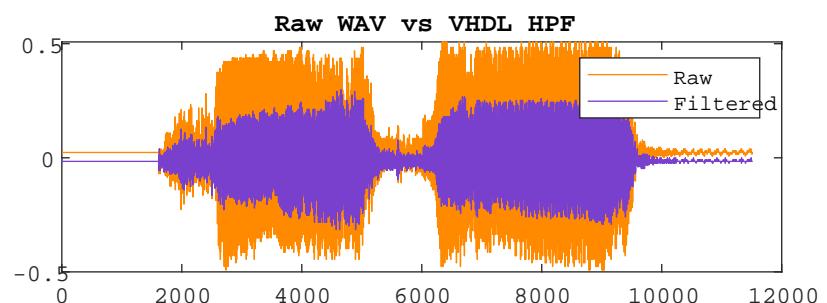
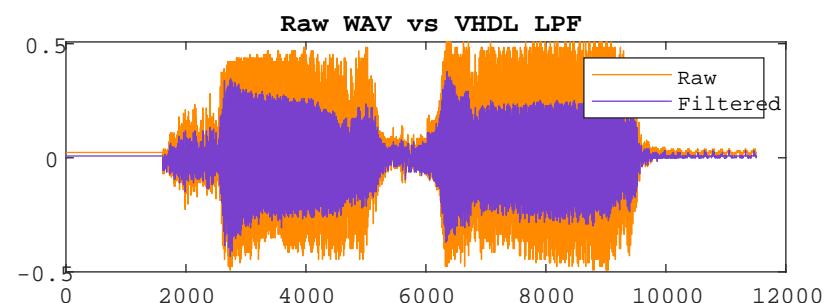
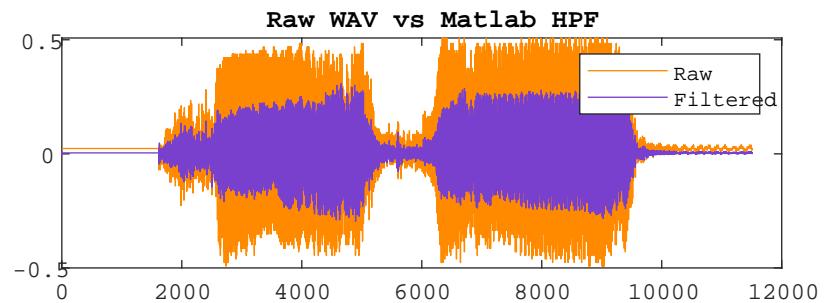
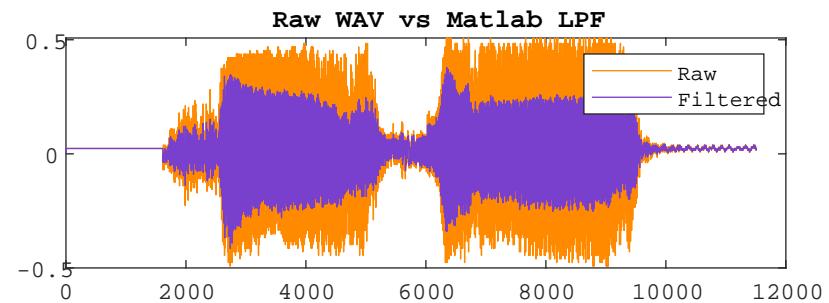
Importa tu fichero “sample\_out.dat” en Matlab. Compara los resultados del testbench con los valores con precisión real proporcionados por la función *filter* de Matlab. Haz un gráfico del error de tus resultados (resta tus resultados de los datos con precisión real).

Utiliza la función *sound* de Matlab para escuchar la forma de onda original. Compárala después con tu sonido filtrado y observa si hay alguna diferencia entre el filtro con precisión real y tu filtro.

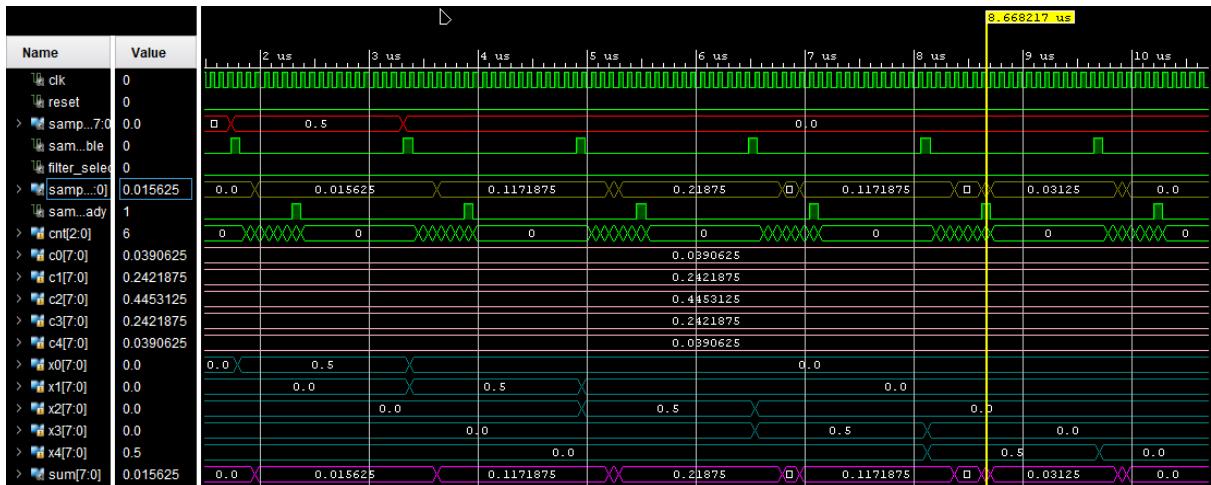
Los gráficos obtenidos se encuentran a continuación

El archivo de testbench para la tarea 2.11 y 2.13 es *adv\_tb.vhd*

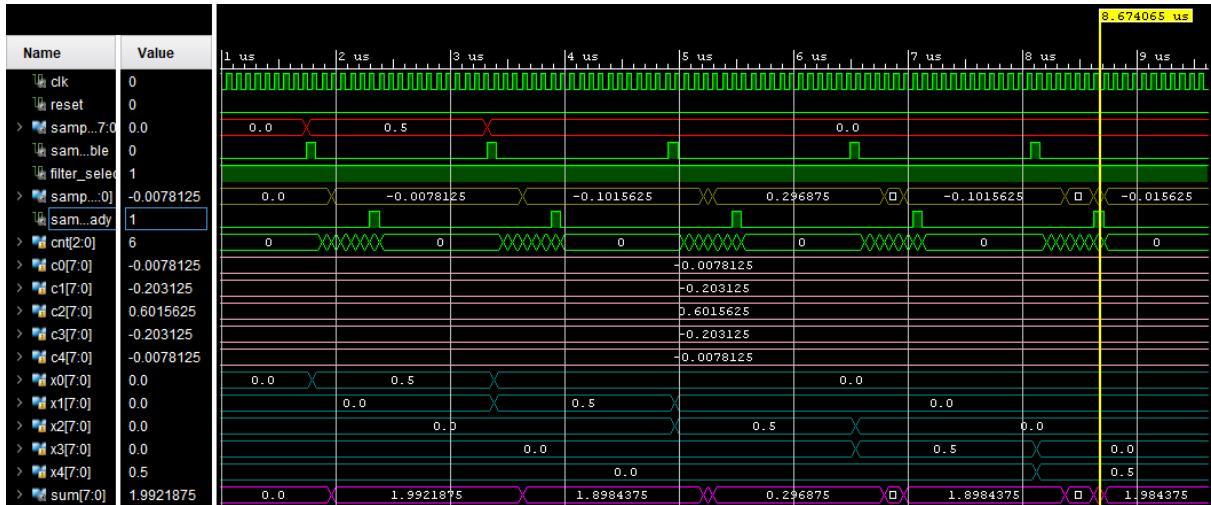
El procesamiento realizado en Matlab para la tarea 2.12 se encuentra en el archivo *data\_proc.m*

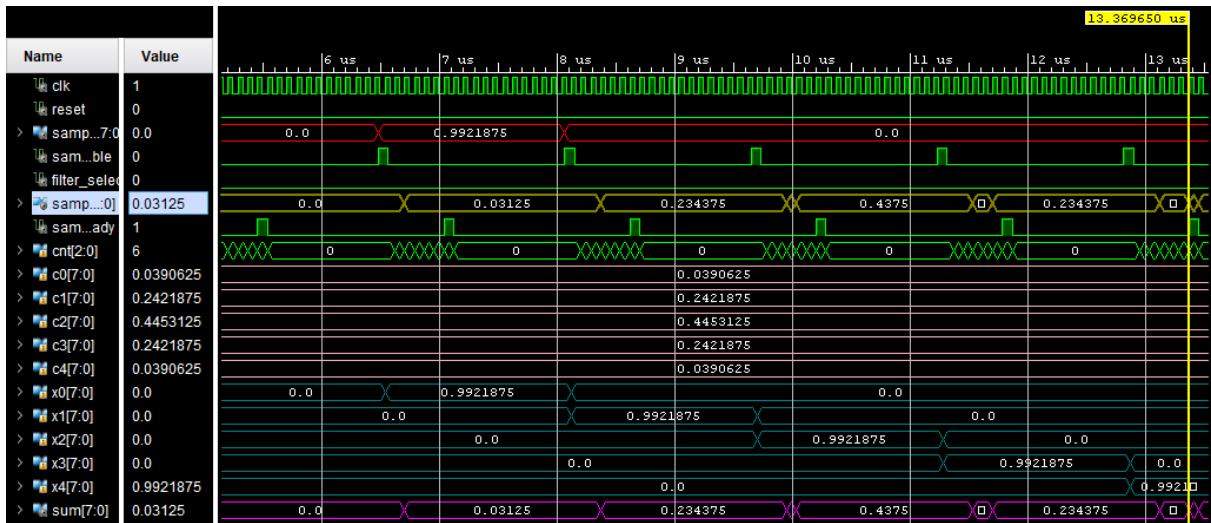


## Testbenches —— Tareas 2.5, 2.9, 2.10

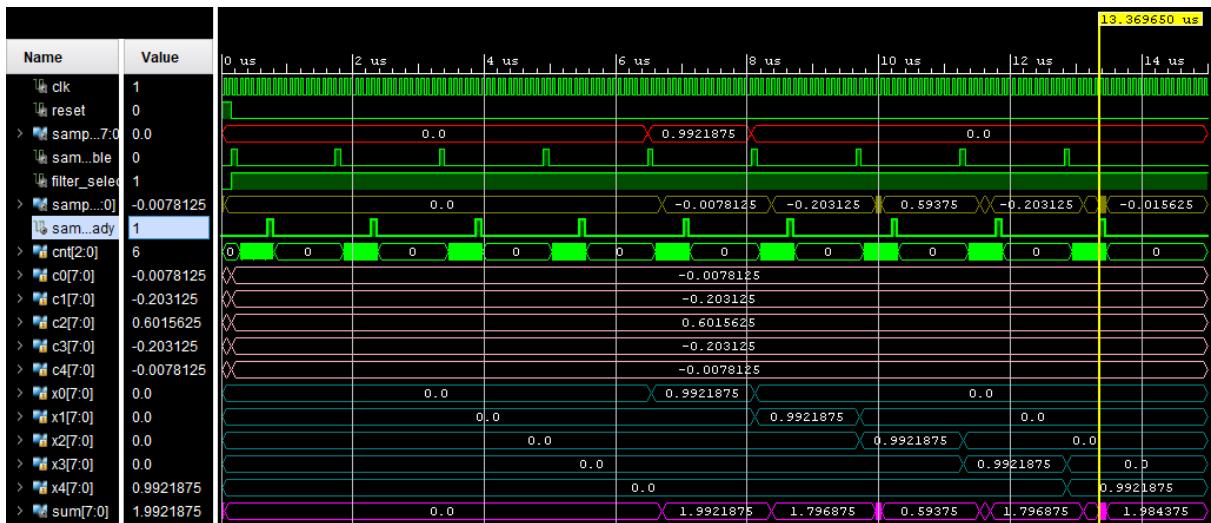


hp → fir\_filter\_tb.vhd (entrada +0.5)

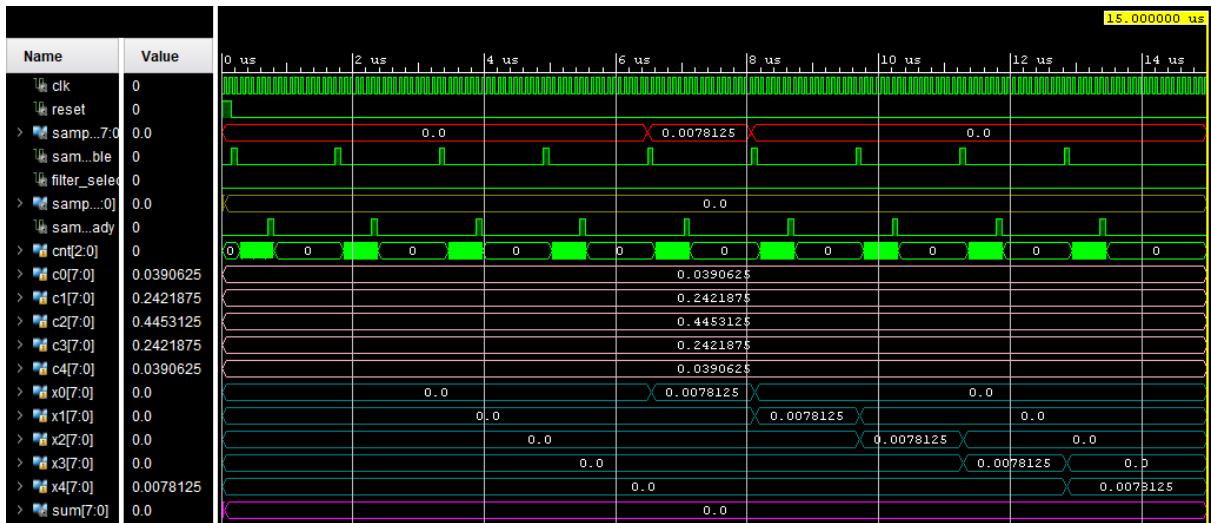




lp → fir\_filter\_tb.vhd (0, 0, 0, 0, 01111111, 0, 0, 0, 0)



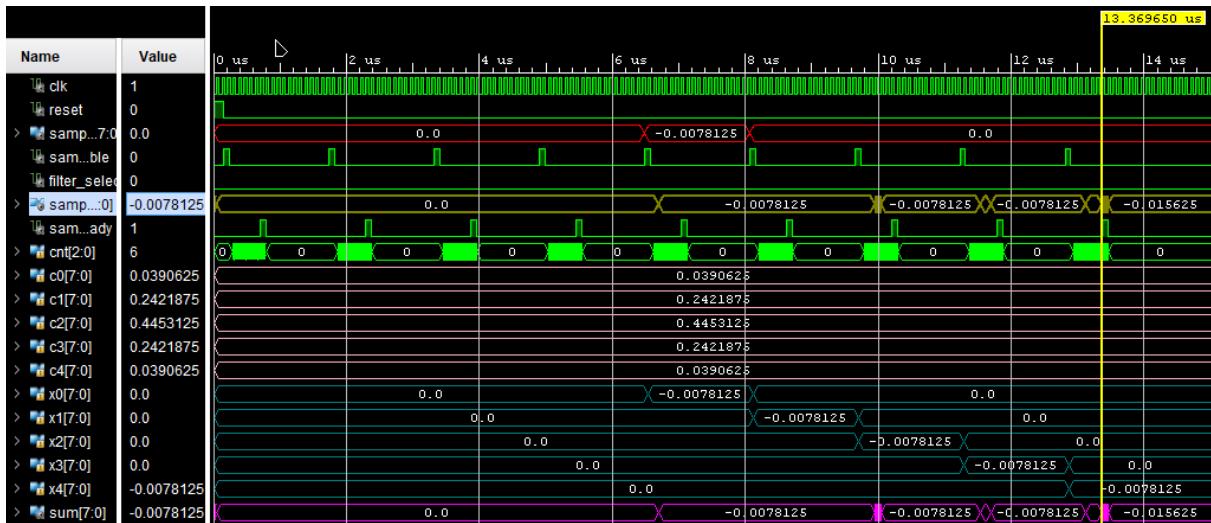
hp → fir\_filter\_tb.vhd (0, 0, 0, 0, 01111111, 0, 0, 0, 0)



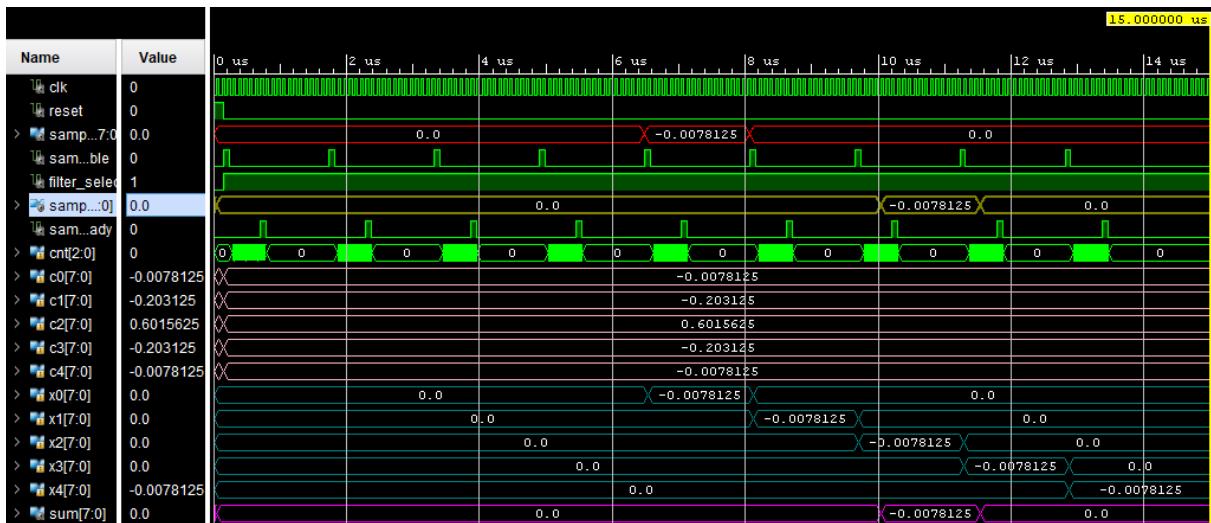
lp → fir\_filter\_tb.vhd (0, 0, 0, 0, 00000001, 0, 0, 0, 0)



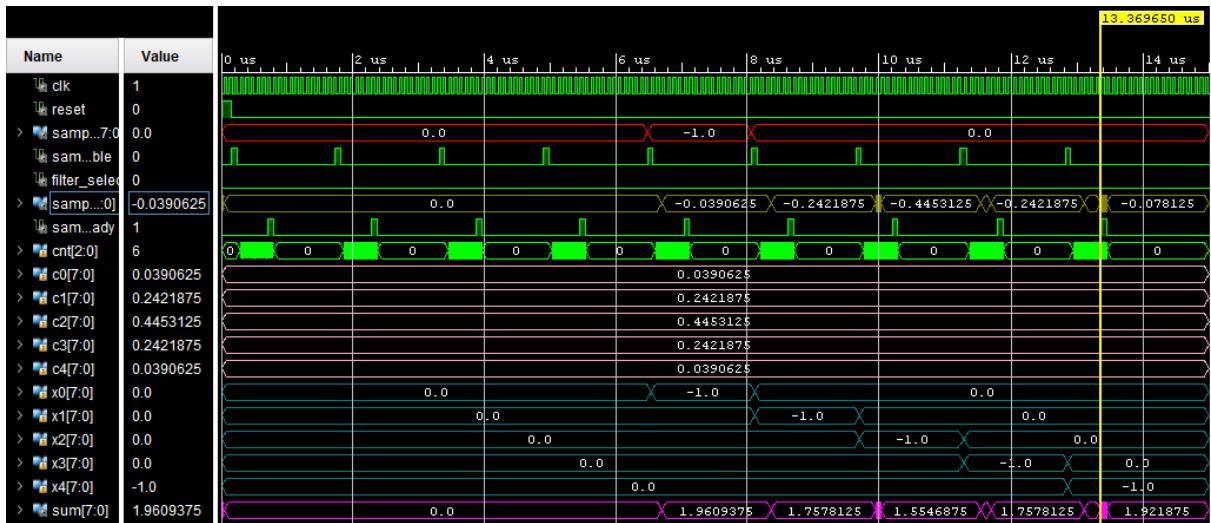
hp → fir\_filter\_tb.vhd (0, 0, 0, 0, 00000001, 0, 0, 0, 0)



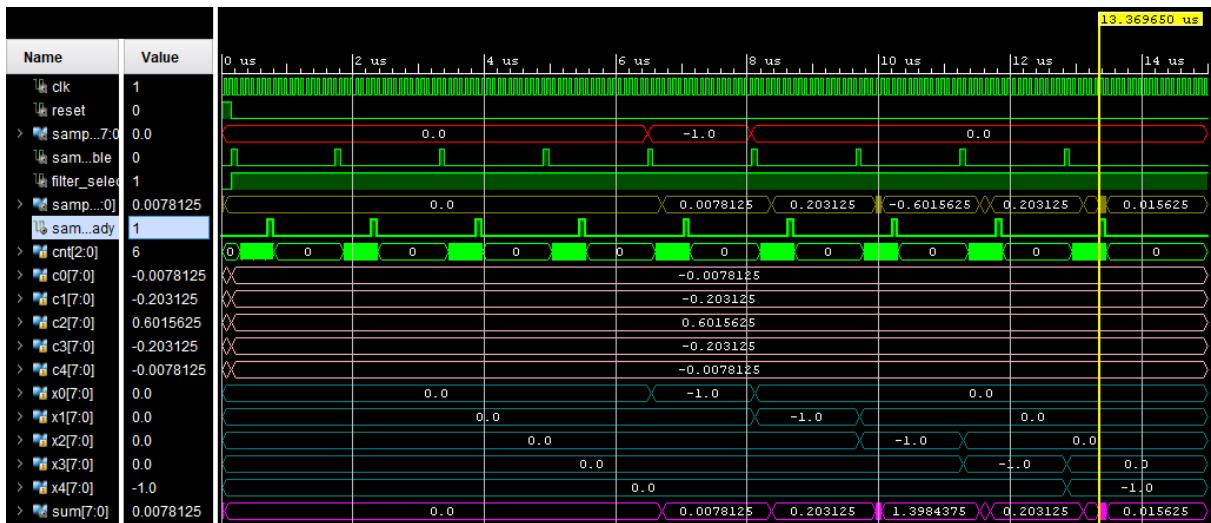
lp → fir\_filter\_tb.vhd (0, 0, 0, 0, 11111111, 0, 0, 0, 0)



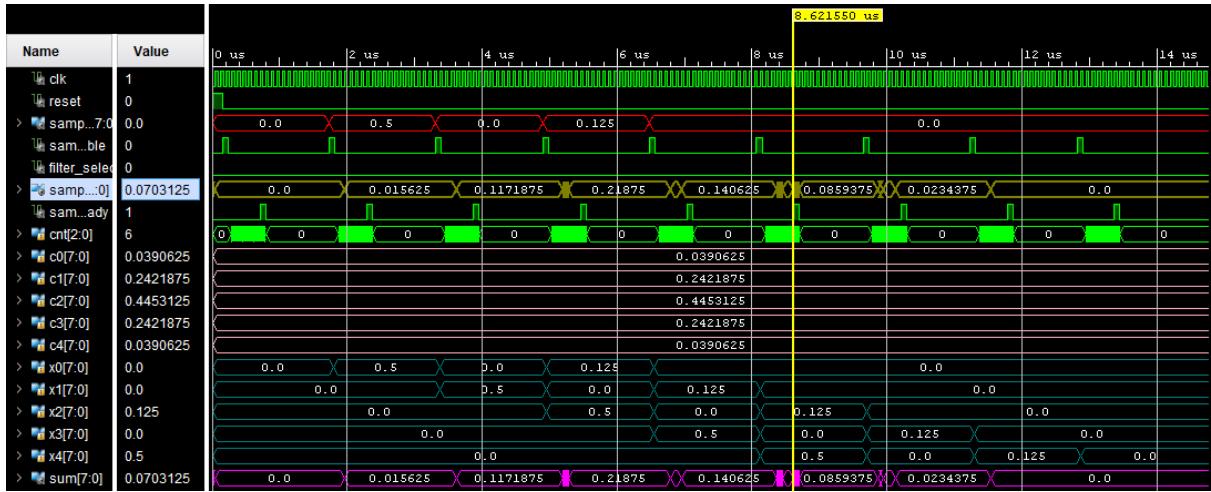
hp → fir\_filter\_tb.vhd (0, 0, 0, 0, 11111111, 0, 0, 0, 0)



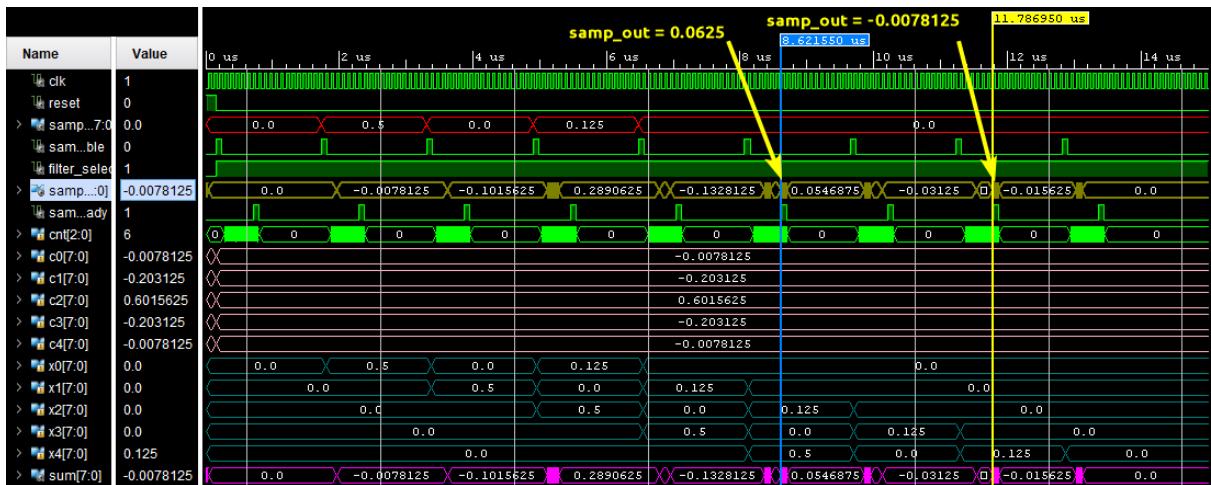
lp → fir\_filter\_tb.vhd (0, 0, 0, 0, 100000000, 0, 0, 0, 0)



hp → fir\_filter\_tb.vhd (0, 0, 0, 0, 100000000, 0, 0, 0, 0)



lp → fir\_filter\_tb.vhd (0, 0.5, 0, 0.125, 0, 0, 0, 0, ...)



hp → fir\_filter\_tb.vhd (0, 0.5, 0, 0.125, 0, 0, 0, 0, ...)

Se ha tenido en cuenta que que en la representación <1.7>:

Signo \ Magnitud (val. abs)	Mínima	Máxima
Positivos	00000001 → 0.0078125	01111111 → 0.9921875
Negativos	11111111 → -0.0078125	10000000 → -1.0000000

## 7. Bloque 3: Controlador y Memoria

En este tercer bloque se va a terminar de dar forma al sistema de audio. En concreto, se va a añadir al diseño un core IP que va a encapsular una memoria RAM donde almacenar las muestras de audio, se va a diseñar un controlador que orqueste todas las operaciones del sistema y se va a integrar todos los elementos.

El diseño del controlador está completamente abierto y se proporciona al estudiante la posibilidad de practicar las habilidades adquiridas a lo largo del curso, diseñando el sistema desde cero teniendo en cuenta el conjunto de especificaciones que lo definen.

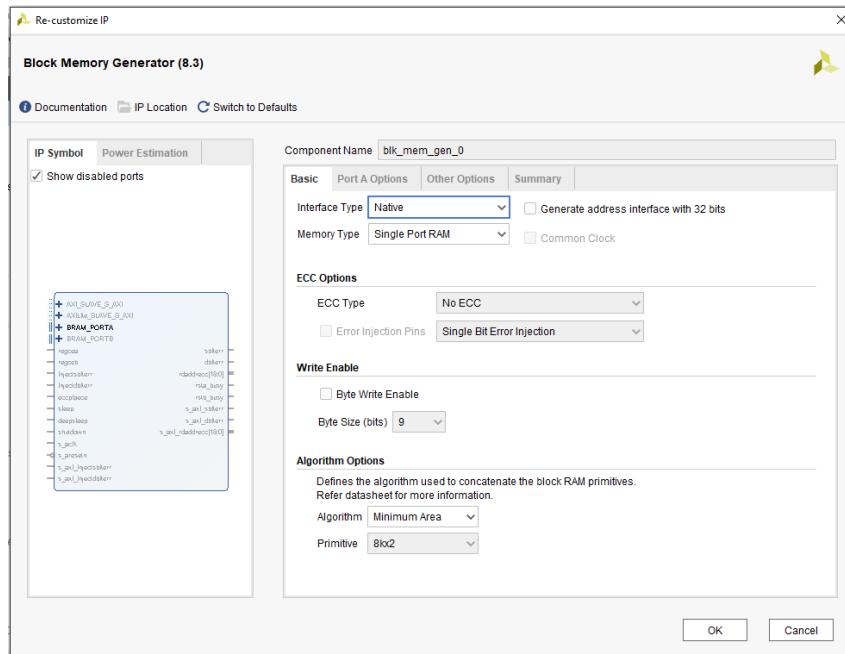
### 7.1. Memoria RAM

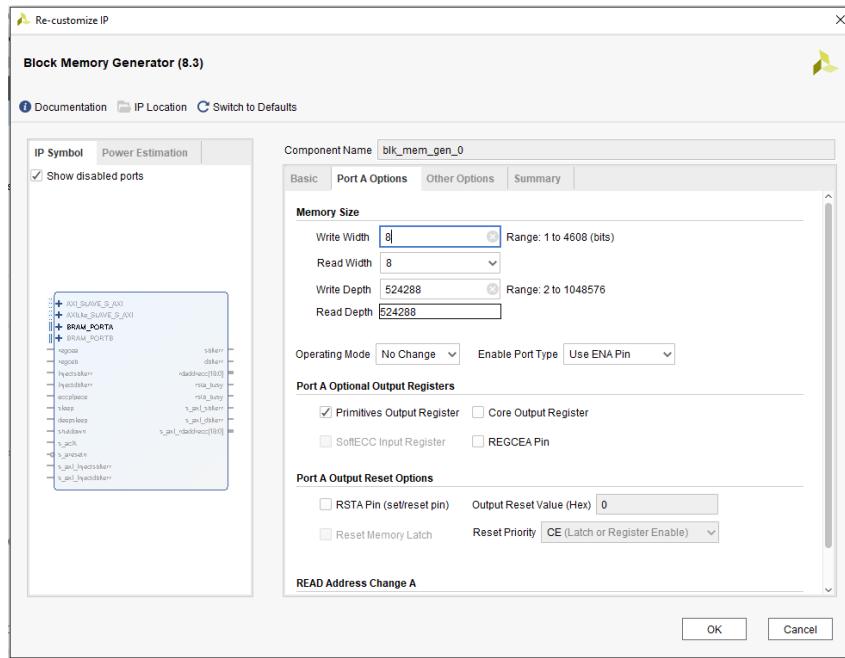
Las muestras grabadas se van a almacenar en una memoria RAM. Las características de la memoria son las siguientes:

- Tamaño de cada palabra almacenada: 8 bits.
- Tamaño del bus de direcciones: 19 bits.
- Número de palabras almacenadas.  $2^{19} = 524288$ .
- Tamaño en bits de la memoria:  $2^{19} * 8 = 4194304$ .

Para instanciar esta memoria en nuestro diseño se va a emplear el “IP Catalog” del Vivado, donde hay que buscar la siguiente opción: “Memories and Storage Elements”, “RAMs and ROMs and BRAM”, Block Memory Generator.

La siguientes dos imágenes muestran la manera de configurar el asistente para encapsular la memoria que se requiere.





### Tarea 3.1:



Crea un testbench que trabaje a la frecuencia del sistema para comprobar y entender el funcionamiento de la memoria encapsulada. Anota a continuación la función de cada puerto y la temporización de la memoria (un pequeño cronograma que incluya una escritura y una lectura).

El testbench desarrollado se encuentra en el archivo fuente *ram\_tb.vhd*

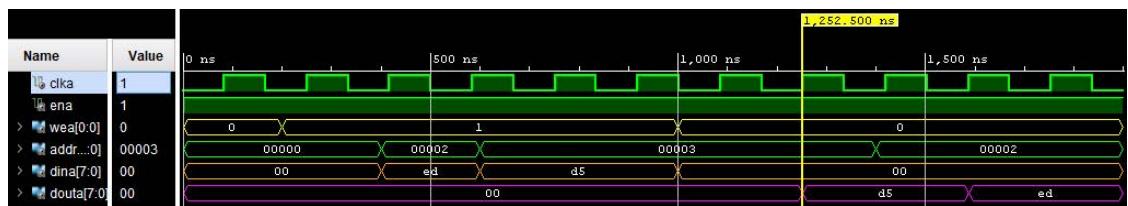
Se prueba la memoria primero realizando dos escrituras:

0xed en addr(0x2)

0xd5 en addr(0x3)

posteriormente después de un tiempo se acceden secuencialmente:

addr(0x3) y add(0x2) recibiendo 0xd5 y 0xed





### Tarea 3.2:

Determina cuánto tiempo de grabación va a poder estar almacenado en la memoria.

Partiendo de que la memoria RAM tiene capacidad para 524288 palabras de 8 bits y que cada muestra tomada ocupa 8 bits:

$$\text{Tiempo max} = \frac{(\text{palabras totales})_{\text{mem}}}{f_{\text{samp}}} = \frac{524288}{20 \cdot 10^3 \text{ Hz}} = 26.214 \text{ s}$$

## 7.2. Conversión entre codificaciones

En el Bloque 1 se obtienen datos de la interfaz del micrófono que utilizan una codificación binaria de 8 bits. Esta misma codificación se emplea en la interfaz de salida de audio. En el bloque 2, sin embargo, el filtro FIR necesita utilizar una representación en complemento a dos para poder usar coeficientes tanto negativos como positivos. Además, los datos están normalizados, de tal manera que el máximo valor con el que trabaja es 1, con lo que emplea un esquema de cuantificación <1,7> en complemento a dos.

Sin embargo, estos dos bloques van a estar trabajando con las mismas muestras (las obtenidas por el micrófono) y, por lo tanto, tenemos que buscar un mecanismo para convertir entre una codificación y otra.

La primera transformación que necesitamos hacer es convertir muestras en notación binaria a muestras en notación en complemento a dos y viceversa. La representación binaria tiene un rango de  $[0 \rightarrow 2^8 - 1]$  mientras que la representación en complemento a dos tiene un rango de  $[-2^7 \rightarrow 2^7 - 1]$ , la correspondencia de las muestras de una representación a otra es directa, es decir al elemento menor de una representación le corresponde el elemento menor de la otra ( $00000000_b$  se corresponde con  $10000000_{2c}$ ), el siguiente con el siguiente y así sucesivamente hasta el mayor ( $11111111_b$  se corresponde con  $01111111_{2c}$ ).



### Tarea 3.3:

Determina qué operación es necesaria para hacer la transformación de las muestras de binaria a complemento a dos y de complemento a dos a binaria.

La relación entre formatos se puede describir de la siguiente forma:

$$00000000_b = 10000000_{2c};$$

$$00000001_b = 10000001_{2c};$$

$$00000010_b = 10000010_{2c};$$

$$\dots$$

$$11111101_b = 01111101_{2c};$$

$$11111110_b = 01111110_{2c};$$

$$11111111_b = 01111111_{2c};$$

Por lo tanto basta que :

$$X_{2c,b} = \text{not } X_{b,2c}(7) \& X_{b,2c}(6 \text{ downto } 0);$$

La siguiente transformación consiste en pasar de un esquema <8,0> a un esquema <1,7>. Pero, ¿es realmente necesario realizar algún cambio en nuestros datos? El lugar donde colocamos el punto decimal es una abstracción que nos permite calcular el valor en base 10 equivalente y con él ajustar el valor de los coeficientes del filtro, pero el hardware se va a ser idéntico sin importar el lugar del punto decimal. Por lo tanto, esta segunda transformación no es necesaria, la primera sí.

### 7.3. Controlador y sistema global

El controlador es el único módulo del proyecto donde el proceso de diseño no está guiado y es la última pieza del puzzle para construir el sistema global. A partir de las especificaciones dadas en la sección 2 y conociendo las funcionalidades y las interfaces de los distintos bloques, decide cómo vas a implementar la funcionalidad del controlador. En concreto, decide que estilo de diseño vas a utilizar (máquina de estados, máquina de estados con ruta de datos asociada, un sistema con distintos elementos secuenciales, etc.), divide el diseño de tal manera que lo puedas ir implementando de manera progresiva añadiendo funciones nuevas cuando estén aseguradas las anteriores, planifica cómo vas a comprobar el funcionamiento de cada especificación y haz una planificación temporal.

#### Tarea 3.4:



Añade a continuación todas las hojas necesarias para describir tu diseño y la planificación para llevarlo a cabo. Puedes incluir diagramas esquemáticos, cronogramas explicativos, un plan de pruebas, una planificación temporal y todo lo que consideres necesario para explicar las decisiones que has tomado.

Avisa al profesor antes de comenzar con el diseño para obtener el visto bueno.



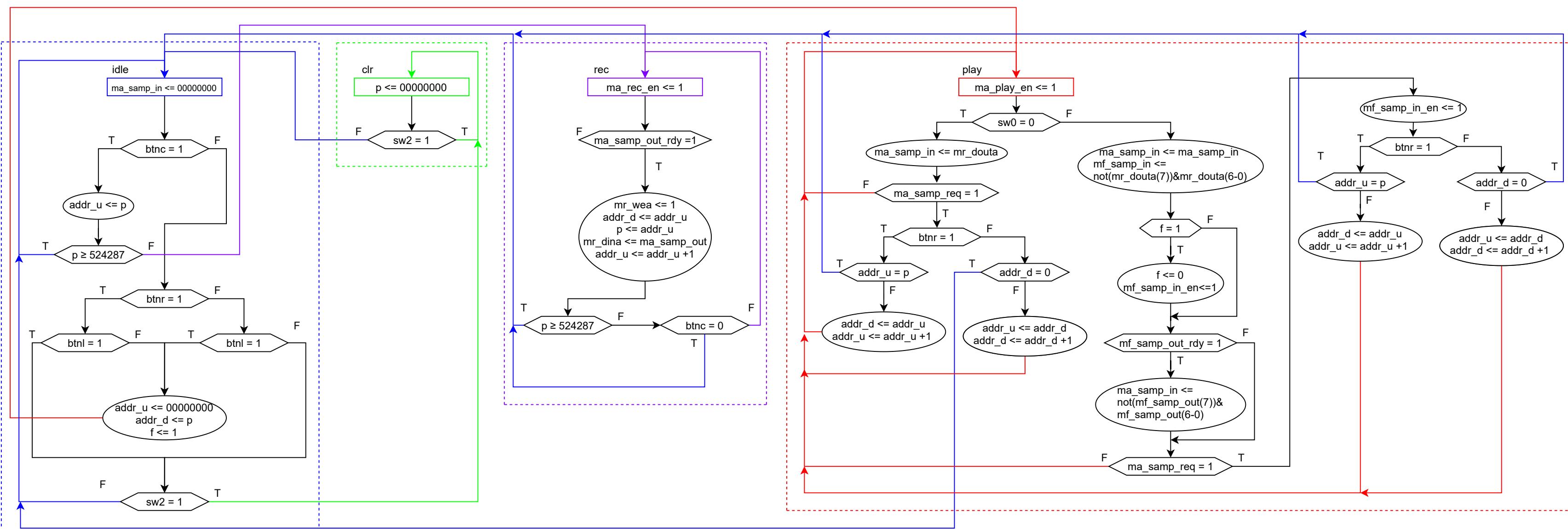
#### Tarea 3.5:

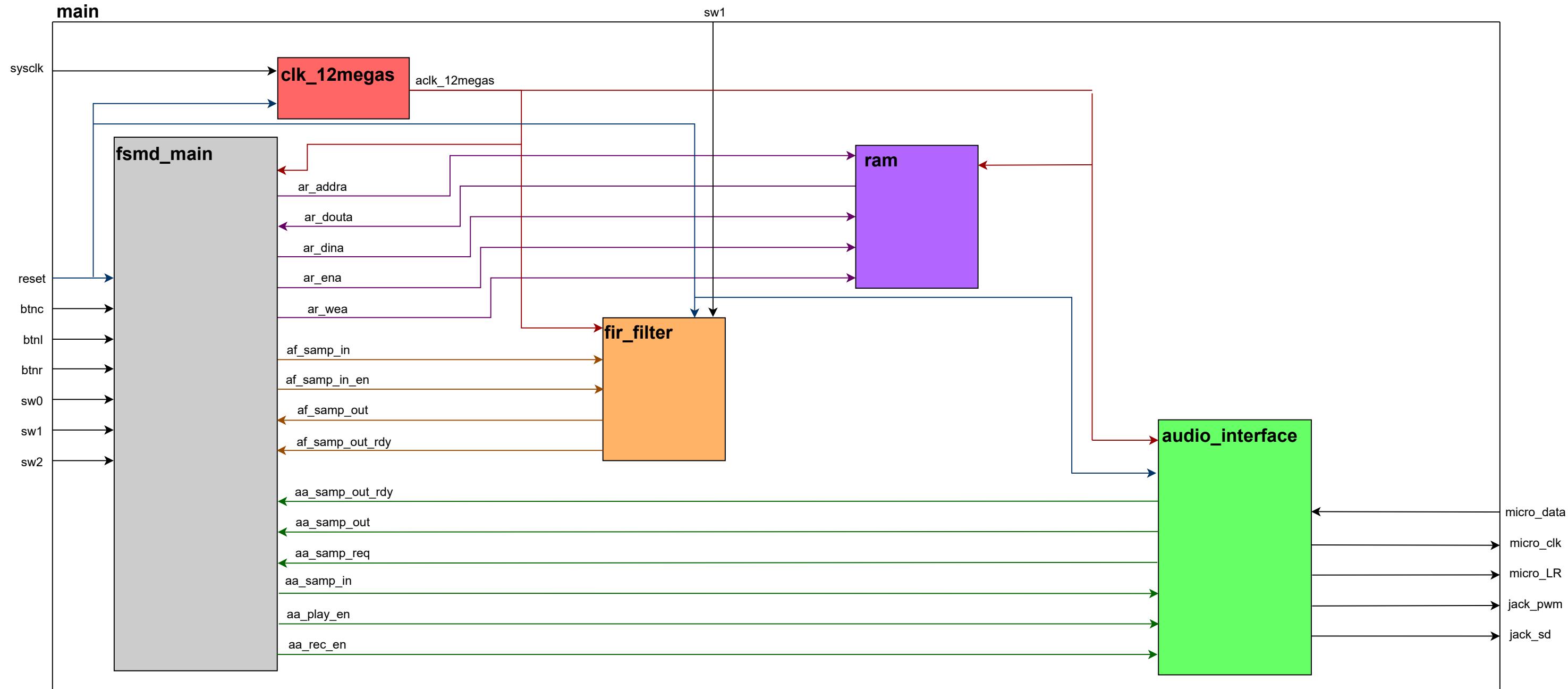
Avisa al profesor cuando tengas todas las especificaciones mínimas del sistema global cubiertas.



#### Tarea 3.6:

Sube al Moodle un fichero .vhd con todas las fuentes de tu proyecto.





## Mejoras adicionales

En el proyecto DSED AudioBox se han realizado las siguientes mejoras al sistema principal:

1. Aviso luminoso de memoria RAM casi llena y llena completamente
2. Muestra de tiempo actual de grabación almacenado, y tiempo restante de reproducción en el estado *play*
3. Muestra del estado actual del sistema
4. Control y visualización del volumen de la salida de audio

### Avisos de capacidad de la Memoria RAM

Mediante esta mejora, el usuario obtiene un aviso inicial cuando la memoria RAM está al 90% de su capacidad y también cuando se ha llegado al 100% de su capacidad.

Para ello, se necesita actuar según el contenido grabado actualmente en la memoria RAM. Esto es posible realizarlo desde el sistema principal implementado en *main.vhd*. La máquina FSMD de control del sistema almacena en todo momento en un puntero la dirección de memoria donde se ha almacenado la última muestra grabada. Emplearemos este registro y sencillas comparaciones para obtener el resultado del estado de la memoria deseado.

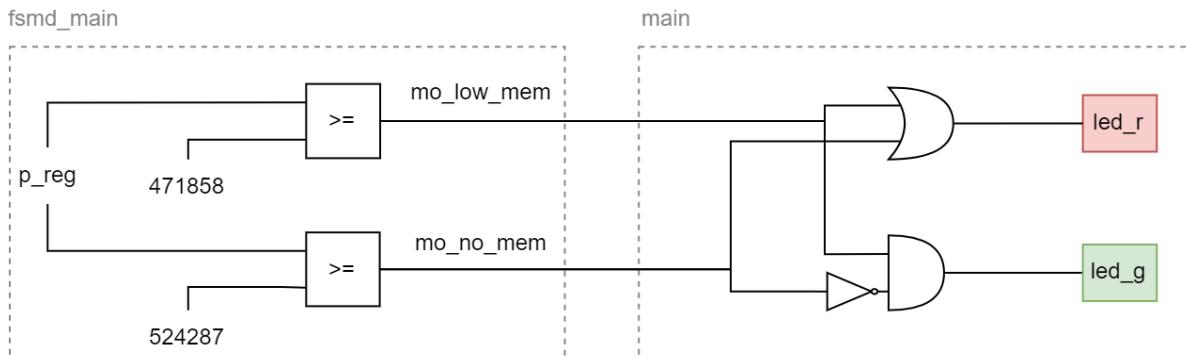
Para comenzar, se parte de que la memoria almacena en su totalidad 524287 palabras. La memoria se escribe secuencialmente desde posiciones bajas a altas. Para obtener el instante cuando la memoria se ha llenado un 90% se calcula la palabra que representa el 90% de las palabras totales en la memoria:

$$\text{floor}(0.9 \cdot 524287) = 471858$$

Es decir, si el registro del puntero contiene la dirección 471858 o superior, estamos en el último 10% de la memoria. De la misma manera, si el registro del puntero contiene la dirección 524287 o superior, habremos acabado la memoria.

Para avisar al usuario se utiliza el LED **LD16 RGB**, emitiendo un color amarillo cuando la memoria está a punto de acabarse y rojo cuando no queda más espacio.

Con esto en cuenta, se añade la siguiente lógica de salida a la FSMD de control del sistema *main\_fsmd.vhd* y *main.vhd*:



### Tiempo de grabación y reproducción

Para esta mejora se han llevado a cabo muchas adiciones y reestructuraciones importantes del sistema original. Esta mejora permite visualizar en 4 dígitos de la pantalla

de 7 segmentos de la placa, el tiempo de grabación y reproducción en segundos y centésimas de segundo.

Su implementación se ha dividido en varias partes:

- Contabilización del tiempo de grabación y de reproducción en la FSMD de control del sistema
- Traducción de los valores binarios de la cuenta a valores BCD para su manejo sencillo con el display
- Traducción de los dígitos individuales de BCD a entradas al display de 7 segmentos.

Los dos últimos apartados se tratarán en un apartado específico que habla del trabajo con el display. Este apartado se centra en la contabilización del tiempo de grabación y de reproducción en FSMD Main.

Para la contabilización del tiempo se ha creado un proceso nuevo llamado *time* que gobierna el funcionamiento de unos contadores que irán contabilizando el tiempo transcurrido.

En primer lugar, es necesaria la creación de un contador de segundos y de centésimas de segundo para almacenar por un lado el tiempo de grabación almacenado y el tiempo actual de reproducción. Para mejorar la utilización de área y explorar las capacidades del *IP Catalog* de Xilinx visto en clase, se ha preferido hacer uso de *Binary Counters* proporcionados por el fabricante y emplear el uso de slices DSP48.

Además, la frecuencia de operación de este proceso será la misma que la de la FSMD Main, 12 MHz. Para que hayan transcurrido 0.01 s, en el proceso habrán transcurrido 120000 ciclos, así podremos saber cuando contabilizar una centésima de segundo.

Entonces, se han creado dos IP de contadores: un componente contador up/down de módulo 99 y otro up/down de módulo 120000. Ambos contadores cuentan con las siguientes características habilitadas:

- Ports enabled
  - CLK: Clock
  - SCLR: Synchronous Clear
  - CE: Clock Enable
  - LOAD: Load Active
  - UP: Up Active
  - L: Load In Port
  - Q: Count Out Port
- Synthesis
  - Fabric: DSP48
  - SCLR override CE
  - Forward latency: 1
  - Count limit: none
  - Count direction: UPDOWN

El contador de módulo 99 tiene un ancho de puertos de 7 bits y el de 120000 de 16 bits. Se instancian 4 contadores mod 99 (cseg y seg para reproducción y grabación) y un contador mod 120000. Al haberse configurado como up/down, el contador no puede limitar automáticamente el conteo, por lo que se realizará externamente por el proceso controlador poniendo el contador a cero con SCLR o LOAD como sea conveniente.

Los contadores son free-running, lo que quiere decir que siempre que el contador tenga CE activo, contará. A los relojes se les suministra el mismo reloj de 12 MHz del sistema.

Se crean los registros correspondientes para almacenar las señales SCLR, CE, LOAD, UP, L, Q y permitir que el sistema continúe operando síncronamente sin situaciones de metaestabilidad entre señales. En total se han empleado 30 registros para su control.

El proceso opera en base al siguiente pseudocódigo:

```
loop: registers <= '0';
    all counters count up <= '1';
    switch state:
        case idle:
            if going into play state:
                load into play_sec from rec_sec;
                load into play_csec from rec_csec;
                do load play counters;
        case clr:
            clear all counters;
        case rec:
            cycles_count_clock_enable <= '1';
            if cycles_count is 120000:
                clear cycles_count;
                rec_cseg_count_clock_enable <= '1';
                if rec_cseg_count is 99:
                    clear rec_cseg_count;
                    rec_seg_count_clock_enable <= '1';
        case play:
            cycles_count_clock_enable <= '1';
            all counters count up <= '0';
            if cycles_count is 0:
                load into cycles_count from 120000;
                do load cycles_count;
                play_cseg_count_clock_enable <= '1';
                if play_cseg_count is 0:
                    load into play_cseg_count from 99;
                    do load play_cseg_count;
                    play_seg_count_clock_enable <= '1';
```

La FSMD entonces debe habilitar dos salidas que son los segundos y las centésimas de segundos que debe mostrar la pantalla. Para ello, dependiendo del estado actual, se multiplexan así:

```
with st_reg select mo_seg <= qp_seg_reg when play,
                  qr_seg_reg when others;

with st_reg select mo_cseg <= qp_cseg_reg when play,
                  qr_cseg_reg when others;
```

Estas señales son las que toma el controlador del display de 7 segmentos para mostrar dichas cuentas.

## Muestra del estado actual del sistema

Implementar esta funcionalidad es sencilla, simplemente el módulo FSMD Main habilita una salida con la codificación binaria del estado actual, para ser utilizada por el controlador del display de 7 segmentos:

```
with st_reg select mo_st <= "000" when idle,  
                  "001" when rec,  
                  "010" when play,  
                  "011" when clr,  
                  "111" when others;
```

El funcionamiento del display de 7 segmentos con esta mejora se detalla en el último apartado.

## Control de volumen en la salida de audio

Esta mejora se ha implementado de forma independiente al funcionamiento del procesamiento de audio grabado o del estado actual de la FSMD Main. La mejora ataca a distintos niveles:

- Modificación de la salida en el módulo PWM a razón de una nueva entrada *factor*
- Actuación sobre el control de volumen y decodificación del nivel y factor de volumen en un nuevo módulo *vol\_ctl*
- Habilitación de salida de información sobre el nivel de volumen actual a través de *audio\_interface*
- Muestra en pantalla del nivel volumen actual

El funcionamiento de muestra en pantalla quedará cubierta en el último apartado.

En primer lugar, el módulo PWM deberá aceptar una nueva entrada *factor* mediante la cual se modificará el ancho de los pulsos modulados **independientemente** al ancho ya controlado por el propio modulador.

Para ello se acepta un valor codificado en coma fija sin signo <4.5>, esta representación ha sido escogida por el rango de valores de *factor* a tratar (0-8) y por la precisión deseada.

El proceso de modulación PWM con la mejora es el siguiente:

- Se realiza la cuenta hasta 299 con el circuito adaptado del Dr. P. Chu como se haría originalmente para emitir el request de sample
- Al emitir la señal request y obtener la muestra entrante, se obtiene paralelamente una conversión de la muestra a *unsigned* (<8.0>) y este valor se multiplica por el factor (<4.5>), que resulta en una representación *unsigned* <12.5> en coma fija
- Para evitar un valor excesivo de volumen por overflow, se limita el volumen máximo a 255 en <7.5>, si el valor de volumen está dentro del margen aceptable se obtiene de su valor convertido de la operación anterior en los bits superiores (<8.0>)
- Finalmente para la modulación, se genera un '1' PWM si el contador cuenta por debajo del valor del volumen obtenido en el paso anterior o no hay sample de entrada (o hay reset), de otra manera (cuenta por encima del volumen requerido y sample de entrada presente) se genera el '0' PWM

El factor que se le proporciona al modulador PWM proviene del módulo de control del volumen, que responde a la acción en los botones BTNU (vol +) BTND (vol -), comenzando por un factor de volumen de 1 (correspondiente al nivel 10).

Este módulo está compuesto por una serie de registros con lógica del estado siguiente y lógica de salida que convierte el nivel controlado por los botones (aumentando o disminuyendo el registro que contiene el nivel de volumen) y saca el factor correspondiente para que sea utilizado por el módulo PWM.

Para convertir el valor de nivel al factor en <4.5> se emplea la fórmula proporcionada por el enunciado de la práctica como mejora sugerida. Como tenemos un número restringido de niveles que tratar, se ha optado por convertir manualmente los valores reales obtenidos de la función a los valores en la representación final y almacenarlos en una pequeña memoria ROM en la lógica de salida del módulo. La tabla siguiente contiene las equivalencias calculadas:

Level	Real	Unsigned <4.5>
0	0,000000	000000000
1	0,035802	000000001
2	0,079296	000000010
3	0,132132	000000100
4	0,196318	000000110
5	0,274292	000001000
6	0,369016	000001011
7	0,484088	000001111
8	0,623879	000010011
9	0,793700	000011001
10	1,000000	000100000
11	1,250616	000101000
12	1,555069	000110001
13	1,924922	000111101
14	2,374224	001001011
15	2,920043	001011101
16	3,583112	001110010
17	4,388617	010001100
18	5,367156	010101011
19	6,555899	011010001
20	8,000000	100000000

La pequeña memoria ROM asigna un valor de nivel codificado en binario de 9 bits a un número sin signo del formato requerido.

Finalmente el módulo genera una salida del nivel actualmente usado en una salida de 5 bits que exporta a *audio\_interface* y ésta al módulo *main* para poder ser conectado al módulo del display.

Las entradas de los botones también se hacen disponibles a través de *audio\_interface* para poder ser enrutadas desde *main* y consiguentemente desde la placa.

En las pruebas de funcionamiento se ha visto como si se incrementa el volumen a partir del nivel 14 o 15, no se escucha nada. Creemos que esto se debe a que el pulso del PWM está siendo ensanchado tanto que en el periodo del PWM no cabe.

## Display de 7 segmentos

Este apartado detalla como todas las mejoras muestran en la pantalla de 7 segmentos información relevante para el usuario. Se van a crear dos módulos para la interacción con el display: *matrix\_driver* y *matrix\_blinker*.

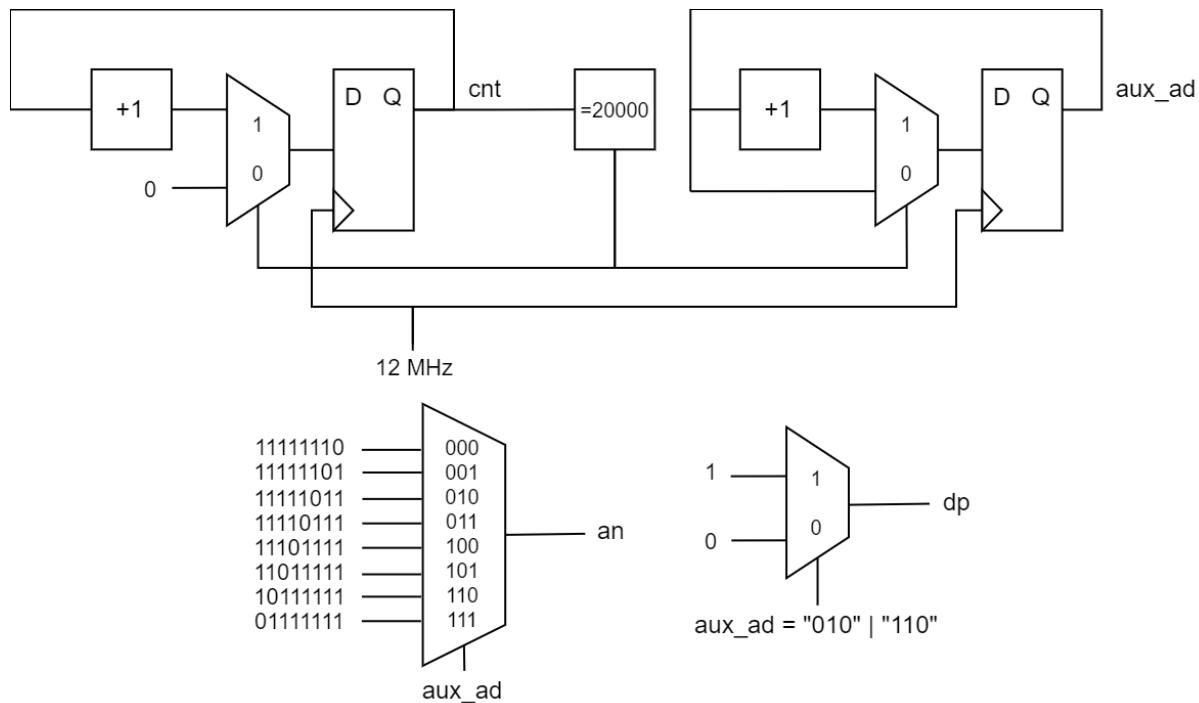
Reuniendo todas las salidas de los distintos módulos de las mejoras tenemos:

- cseg: centésimas de segundo (de rec o play) → 7 bits
- seg: segundos (de rec o play) → 7 bits
- st: estado actual: → codificación del estado en 3 bits
- level: nivel de volumen → 5 bits

Para operar con la matriz de 7 segmentos, cabe recordar que su funcionamiento es de modo ánodo común y que en un determinado instante de tiempo sólo se puede controlar uno de los 8 displays de 7 segmentos que forman la matriz. Para ello será necesario refrescar cada dígito y dependiendo del dígito que se controle enviar a los diodos de la matriz de 7 segmentos el conjunto de segmentos que queremos representar (número o letra), el módulo *matrix\_blinker* se encarga de esto último.

### matrix\_blinker

Este módulo se encarga de refrescar cada dígito a una frecuencia de 75 Hz, (pasa a controlar el siguiente dígito a 600 Hz). Este módulo se compone de un contador y de un decodificador 3 a 8 como muestra el siguiente esquemático:



### matrix\_driver

Este módulo se encarga de recoger la señal de *an* para los ánodos de la matriz y decidir según el display que se está pintando cuál es el dígito que se debe mandar

Para ello se utilizan los siguientes elementos:

- Conversor de binario (7 bits) a dos dígitos BCD (8 bits)
- Conversor de un dígito BCD (4 bits) a su representación en 7 segmentos
- Decodificador del estado actual de la FSMD Main a una representación en dos dígitos de la matriz de 7 segmentos
- Un multiplexor 8x7

La implementación de cada uno de los bloques se puede revisar en el código fuente. Cabe mencionar que los conversores de formato Binario ↔ BCD ↔ 7seg y el decodificador de estado están implementados con pequeñas memorias ROM.

La interconexión entre los componentes se puede ver en la página final adjunta

