Diseño de Sistemas Electrónicos Digitales

Audio System

Amadeo de Gracia Herranz

Victor Bautista Loza

Samuel López Asunción

Pablo Ituero Herrero

Version 2024

ETSIT UPM

Departamento de Ingeniería Electrónica

Diseño de Sistemas Electrónicos Digitales 2024
Audio System

Based on "Sistema de grabación, tratamiento y reproducción de audio."
Reformatted and adapted by Authors
Departamento de Ingeniería Electrónica
UNIVERSIDAD POLITÉCNICA DE MADRID
Spain

Contents

1 Introduction

This document describes the final project of the course "Diseño de Sistemas Electrónicos Digitales " of the GITST UPM. The project is about the design and implementation of a digital electronic system of medium complexity. The project aims to establish and deepen the methodologies of combinational and sequential design, the design of finite state machines and finite state machines with associated data paths, the methodology of high-level synthesis and timing issues. Specifically, a system is proposed which, using the audio input and output interfaces of the board with which the students have been working in the first half of the practical laboratory hours, acquires, stores, processes and reproduces sounds.

2 System description and specifications

The system has the following minimum global specifications:

- The system is implemented on the Nexys 4DDR board using the workflow of Vivado workflow.
- The system receives audio information from the on-board microphone.
- The system plays the audio through the on-board mono audio output.
- The system is controlled by three buttons (BTNL, BTNC and BTNR) and two switches (SW0 and SW1).
- The system operates at a clock frequency of 12 MHz, which is generated from the 100 MHz clock available on the board.
- The system has a global reset from the BTNU button.
- The audio is sampled and played back at a frequency of 20 kHz.

The working principle is the following:

- The system starts from an empty audio memory.
- When BTNL is pressed and for as long as it is pressed, the system records the audio.
- Each time BTNL is pressed again the new audio is added at the end of the previously recorded audio.
- When BTNC is pressed all recorded audio is erased.
- When BTNR is pressed, the system must perform actions according to the information from the switches:
  - If both switches are set to '0' the system will play all recorded audio.
  - If SW0 is set to '1' and SW1 is set to '0' the system will play the stored audio backwards.
  - If SW0 is set to '0' and SW1 is set to '1' the system will play the audio filtered by a low pass filter.

– If SW0 is set to '1' and SW1 is set to '1' the system will play the audio filtered by a high pass filter.

The entity of the top file of the system is provided in the next lines:

```
-------------------------------------------
-- Top file audioSystem.vhd
-------------------------------------------
-- Libs
entity audioSystem Is
port (
    clk_100Mhz : in STD_LOGIC;
    rst : 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 audioSystem;
```

2.1 Description of the system structure

The system is composed of four main blocks:

1. *Audio Interface* has two main functionalities. First, it is in charge of converting the Pulse Density Modulated (PDM) input signal coming from the microphone. On the other hand, it is the block in charge of interacting with the output audio amplifier by providing it with a proper Pulse Width Modulated (PWM) signal. In addition another module could be necessary to generate a bank of enables of different frequencies. This module is going to be implemented following a finite state machine methodology with an associated data path.

2. *Filtering Block* is in charge of the digital processing of the audio signal for the high-pass filter and low-pass filter functionalities. This block will be implemented following the high-level synthesis methodology.

3. *RAM memory* stores the audio samples that have been previously recorded. This block will be generated with Vivado's architectural wizard as we have seen with the clock before.

4. *Controller* is in charge of orchestrating the entire system performance by providing control signals to the different blocks to execute the system user's commands. In this block, the design style is free.

The Figure 1 provides a general outline of the blocks and their interrelationships. Make the effort to understand it and add the parts you consider necessary.

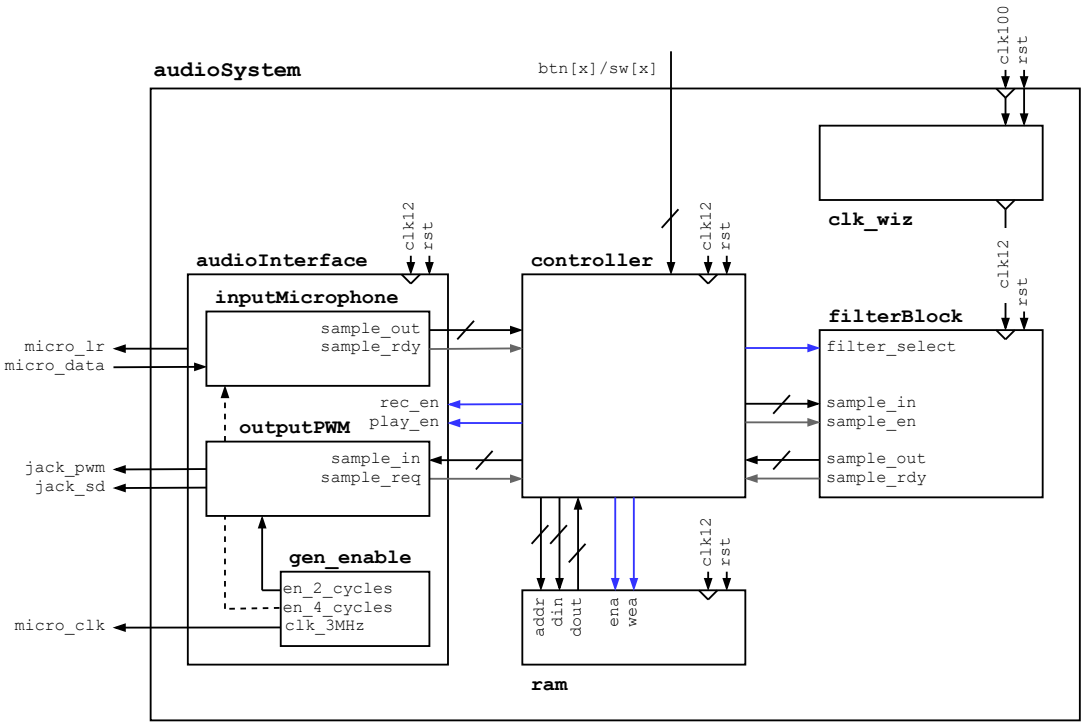


Figure 1: Block diagram of the full system.

2.2 Package usage

The code employs a package defined by us called `packageDSED.vhd`. The aim is to store all constants and data types to be used by several blocks in a common library, so that the code becomes more readable, avoiding the use of numeric expressions that you don't know where they come from. The next code provides the declaration of de package:

```
1 ---------------------------------------------
2 -- packageDSED.vhd
3 ---------------------------------------------
4 package packageDSED is
5     constant sample_size: integer :=8;
6 end packageDSED;
```

The idea is to add lines to this declaration according to the needs of the design. To make use of this package, we have to include the following line in the header of our code:

```
1 use work.packageDSED.all;
```

3 Time Scheduling

The project is designed to be carried out in pairs in the second half of the laboratory sessions of the course, which is equivalent to approximately 6 sessions of 2-3 hours with the support of the teacher plus another 15 hours of unguided work. The project has been divided into milestones that correspond to the main blocks of the system structure:

- Weeks 1-2. Audio interface.
- Weeks 3-4. FIR filter.
- Weeks 5-6. Controller and memory.

This document is intended as a notebook and a guide where to write down the design decisions and the results of the implementation.

4 Evaluation

The evaluation of the project will take into account the following elements:

- A short demonstration showing how the implementation works on the board, as well as the results of various system testbenches.
- The quality of the design decisions reflected in a short memory.
- The quality of the VHDL code. It is important that the code is readable and well organized (use of explanatory comments, systematic use of prefixes and suffixes in port, signal and variable names, etc.).
- The quality of the improvements, which will be evaluated using the three criteria above.

5 Block 1: Audio Interface

The *audioInterface* module is going to be your interface to the external audio signals. The Nexys A7 board sports an on-board PDM (Pulse Density Modulation) **microphone**, which we will use to digitise, process and record audio. Its specification defines a frequency range of 10 kHz, so we will be sampling all our audio signals at twice that frequency $f_s = 20$ kHz, or equivalently, $T_s = 50$ μs.

The board also features mono audio output through a 4th-order Sallen-Key operational amplifier filter, which is able to convert a **PWM encoded audio** signal into an **analog output**. It has a bandwidth of about 20 kHz, which is more than enough to play our sampled data. This output audio signal is available through the green audio jack connector (J8).

This module is going to be a simple wrapper containing two independent modules in charge of actually getting the audio information in and out of the FPGA: one module called *inputMicrophone* for the conversion of PDM to 8-bit signed valued samples at the mentioned sampling frequency, and another one called *outputPWM*, for the reverse conversion of those samples into a valid PWM signal for the output amplifier.

The next code is the declaration of this module:

```
----------------------------------------------
-- Top file audioInterface.vhd
----------------------------------------------
-- Libs
entity audioInterface Is
port (
    clk_12Mhz : in STD_LOGIC;
    rst : 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 audioInterface;
```

Port description:

- *clk_12Mhz*: Global clk of the architecture at 12MHz.
- *reset*: Global reset.
- *record_enable*: Recording control signal. When '1' the microphone digitization is working.

- *sample_out*: 8-bit data (unsigned) that corresponds to the digitization of the microphone.
- *sample_out_ready*: Control signal that fires a pulse of one clk_12MHz duration each time a new digitized data is ready.
- *micro_clk*: Microphone clk out. A 3MHz clk derived from the 12MHz clk. Derivation clocks are a malpractice in digital design but in this specific case we will allow it due to special circumstances.
- *micro_LR*: Microphone control output that determines whether samples are taken on the rising or falling edge of the clock. We will leave this value set to '1', corresponding to the rising edge.
- *play_enable*: Audio play control signal. When it is set to '1', the PWM signal will be generated to the mono audio output.
- *sample_in*: 8-bit data corresponding to the signal to be reproduced.
- *sample_request*: Control signal that provides an active pulse of one clock period duration each time a new data is required in sample in.
- *jack_sd*: Control information for the mono audio stage op-amps. We will leave this value set to '1'.
- *jack_pwm*: The PWM signal generated from the data in the sample_in

Structurally, three components can be distinguished in this block:

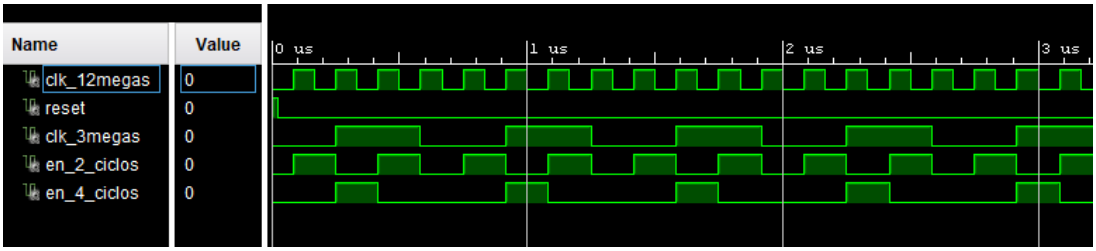
- *Microphone Interface*. In charge of the "recording ports" except for micro_clk.
- *Jack/PWM Interface*. In charge of the "playing ports".
- *Enable and microphone clk generation*. In charge of generate the controlling signals of data sampling, data conversion and microphone clk.

5.1 Enables and microphone clk generation

This block receives the 12 MHz global clock signal and provides its outputs with signals that will be used to time the rest of the audio interface modules. The module entity is defined as follows:

```
1  ----------------------------------------------
2  -- Top file gen_enables.vhd
3  ----------------------------------------------
4  -- Libs
5  entity gen_enables Is
6  port (
7      clk_12Mhz : in STD_LOGIC;
8      rst : in STD_LOGIC;
9      clk_3MHz: out STD_LOGIC;
10     en_2_cycles: out STD_LOGIC;
11     en_4_cycles: out STD_LOGIC
12   );
13 end gen_enables;
```

The 3MHz clk output is used by the microphone and is a 3 MHz clock with a duty cycle of 50%. The 2 cycle output is used by the audio output interface and provides an active signal for one clock cycle every two cycles (equivalent to a 6 MHz clock). The 4-cycle output is used by the microphone interface and provides an active signal for one clock cycle every four cycles. The following figure shows the expected operation.



☞ **Task 1.1:**
Design a circuit that is capable of generating the specified signals with the same phase. Draw the schematic corresponding to your design and add it to the final report.

☞ **Task 1.2:**
Implement the design in VHDL.

☞ **Task 1.3:**
Generate a testbench and verify the output with the provided example.

5.2 Microphone Interface

To design this module, it is necessary to have a good knowledge of the interface provided by the omnidirectional microphone of the Nexys4 DDR board, the pulse density modulation (PDM) and the timing of the microphone interface. For this, it is recommended to read section 15 of the "Nexys4 DDR FPGA Board Reference Manual".

5.2.1 Sampling scheme

The PDM microphone mounted on the Nexys A7 board is a PDM digital microphone. It has a 1-bit output signal running at a frequency between 1 MHz and 3.3 MHz (in our case, we will be using a 3 MHz signal), much higher than the sampling frequency $f_s = 20$ kHz. It encodes information in the density of '1's, meaning that a high value in the physical audio signal will generate a stream with many '1's, while a low physical value will generate more '0's. This means we need to group and count the number of '1's that it generates to get the sample.

The system provides a 3 MHz clock signal to the `micro_clk` output (333.3 ns period). The module needs to take a sample every 50 μs, that is, one sample every 150 `micro_clk` periods. The system counts with 8-bit unsigned data, that is, it will count the number of ones integrating the input bits in groups of 256 samples. The following image shows the sampling scheme of the system:

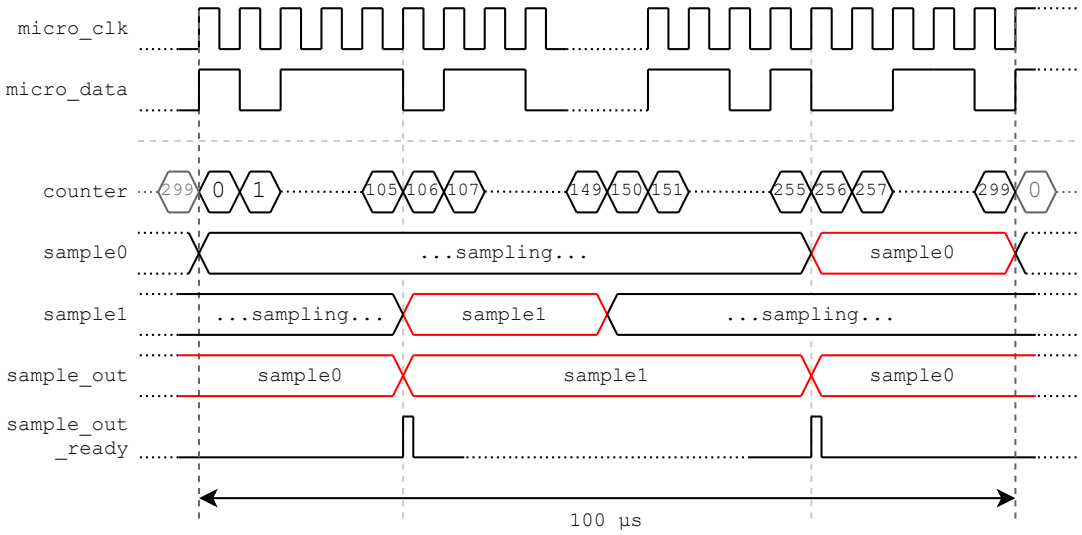


Figure 2: Chronogram of the input microphone module.

As shown in the figure, there is a counter with a modulus of 300 that counts cycles from 0 to 299. These 300 cycles correspond to 100 µs, which is twice the sampling period. The first counter `sample0` samples between cycles 0 and 255 and outputs the result between cycles 256 and 299. The second counter `sample1` samples between cycles 150 and 105 and provides its own between cycles 106 and 149. The `sample_out` signal is updated at cycle 106 with the sampled value from `sample1` and at cycle 256 with the value from `sample0`, meaning it updates every 150 cycles or 50 μs to get the desired sampling rate. The `sample_out_ready` output produces a one-period-long pulse to notify when new digitised data is available. This period corresponds to the system's main clock $f_s = 12$ MHz.

5.2.2 Design and Implementation

The design of the microphone interface will be carried out using the FSMD methodology. From a pseudocode an ASMD graphic will be generated, this graphic will serve as a basis for the description of the VHDL module. It is recommended to review chapters 11 and 12 of the book "RTL Hardware Design Using VHDL" by Professor Pong P. Chu and the class slides to become familiar with this methodology. The module entity is defined as follows:

```
entity inputMicrophone Is
port (
    clk_12Mhz : in STD_LOGIC;
    rst : in STD_LOGIC;
    en_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 inputMicrophone;
```

The enable 4 cycles signal is an enable signal that is activated for one clock period every four periods. It comes from the enable generator and allows the microphone interface to operate effectively at a speed of 3 MHz. The rest of the ports have already been described above. From the sampling scheme, we can describe the digitization algorithm using the following pseudo-code:

```
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
```

☞ **Task 1.4:**
Draw the ASMD diagram describing the same functionality as the previous pseudo-code.

☞ **Task 1.5:**
Implements the ASMD diagram in VHDL code.

5.2.3 Test strategy

Before we can hear if the result of our digitization is correct, we need to check through testbenches that our system works as we expect. We can first check that the state machine runs through the

states as expected, generating the necessary signals to reset the counters. You can instantiate the enables generator in your testbench to have the enable_4_cycles signal available.

☞ **Task 1.6:**
Perform a testbench that has the micro data signal set to '1' to check that state transitions, selection of output data and activation of sample out ready occurs correctly.

In a second phase we can check that the digitization is done correctly, for this we can introduce pseudo-random signals in the micro data signal and calculate the expected result by hand. We can construct these pseudo-random signals with sentences of the type:

```
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;
```

☞ **Task 1.7:**
Performs a testbench that inputs a pseudo-random signal into micro data and checks that the digitization is performed correctly.

5.3 Output interface

To design this module, it is necessary to have a good understanding of the interface that provides the Nexys4 DDR board's mono audio output and pulse width modulation (PWM). For this purpose, we recommend reading section 16 of the "Nexys4 DDR FPGA Board Reference Manual".

Since the data has been sampled at 20 ksamples/s, the pulse generation rate has to be the same: one pulse every 50 mus. For the PWM signal generation, it is proposed to rely on section 9.2.5 of the book "RTL Hardware Design Using VHDL" by Professor Pong P. Chu. In the scheme of the book, the output value of a counter is compared with the digital word to be converted, if the counter value is smaller, the PWM output is set to "0" and to "1" otherwise. In the case of our system, the counter is going to increment its output every two cycles of the global clock, i.e. at a frequency of 6 MHz (166 ns period). That is, during the sampling period of 50 mus, the counter will count from 0 to 299. As our digital data is 8 bits, the maximum value represented is 255, therefore all data that we reproduce will have a small decrease in volume by a factor of $255/300 = 0.85$.

The entity of the block is declared as follows:

```
entity inputMicrophone Is
port (
    clk_12Mhz : in STD_LOGIC;
    rst : 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 inputMicrophone;
```

The signal in en_2_cycles is a signal that is activated every two cycles and is the one that allows us to work effectively at half the frequency. Every time the module has finished a count, i.e. it has reached 299 and is going to go to 0, it has to put an active pulse in sample_request of duration only one period of the general clock to request a new sample to the input.

☞ **Task 1.8:**

Modify the design proposed in Dr. Chu's book to be compatible with our system specifications. That is, have the counter count from 0 to 299 and include reset and enable and have the circuit produce the sample_request output at the appropriate time and for the appropriate duration. Draw the schematic of your design.

☞ **Task 1.9:**

Implements the above schematic in a VHDL file.

5.3.1 Test strategy

Before testing the module on the board, we need to make sure through simulations that the operation is as expected. Specifically, we have to check that the counter counts every two cycles, that the count is reset when it reaches 299 and that the pwm_pulse and sample_request signals are as expected. For this we can verify the operation by introducing the extreme values ("0000 0000" and "1111 1111") and some random value in between. You can instantiate the enables generator in your testbench to have the enable_2_cycles signal available.

☞ **Task 1.10:**

Perform a testbench to verify the operation of the audio output interface.

5.4 Audio interface integration

Once the three previous modules have been implemented, the next step of the design consists in the instantiation of each of these blocks in a single module that will form the complete audio interface.

☞ **Task 1.11:**

Draw the block level schematic of the complete audio interface. Note that record_enable and play_enable specify when the microphone and audio output interfaces are active, respectively. Likewise, you have to assign a '1' to both microphone_LR and jack.

☞ **Task 1.12:**

Implements in VHDL the complete audio interface. To do so, base it on a structural code style that instantiates each of the three blocks developed above.

5.4.1 Test Strategy

To verify the functionality of the entire block, you can reuse the stimuli used in the previous testbenches.

☞ **Tarea 1.13:**
Design a testbench that verifies the functionality of the complete audio interface.

5.5 Board Test

Once you have verified the functionality of the complete block by means of a testbench, you are going to perform the physical implementation of the block on the board. In order to verify that the system actually digitizes the microphone signal well and transforms the digital word into a PWM pulse, the on-board test will consist of making what is recorded by the microphone come out of the audio output.

To achieve this, we need a simplification of our audioSystem, without input buttons or switches. We will need our 12 MHz clock generation block from the 100 MHz one on the board. We keep record_enable and play_enable set to '1' and feeds the digitized word from the microphone interface (sample_out) to the audio output interface (sample_in). They do not need any special synchronization, as long as sample_in is stable for 50 mus (it does not matter if it changes when the counter is set to 0 or not).

☞ **Task 1.14:**
Implements it in VHDL. Use structural style. You need to use the Vivado architectural wizard to generate the 12 MHz clock from the 100 MHz.

☞ **Task 1.15:**
Design a testbench that verifies the functionality of the controller. You can use the stimuli used in previous testbenches.

☞ **Task 1.16:**
Writes the .xdc constraints file. Synthesizes and performs the physical implementation of the design. Generate the .bit file and program the FPGA. Check the correct operation by connecting headphones to the audio output of the board. If the audio output is very noisy, you can play with the value of micro_LR and set it to '0' instead of '1'.
Save in a text file all the warning messages that have appeared in the previous process.
Notify the professor to check the operation.

6 Block 2: Filter Block

6.1 Digital filters

In digital signal processing, filters are circuits designed to permit signals within specific frequency ranges to pass through while attenuating or blocking those outside of these ranges. The frequency range that a filter allows to pass through is known as the pass band, while the range it attenuates or blocks is referred to as the reject band. Filters are categorised based on the positioning of these bands: there are low-pass, high-pass, and band-pass filters. The transfer functions of these filters are depicted in the Figure 3 below.

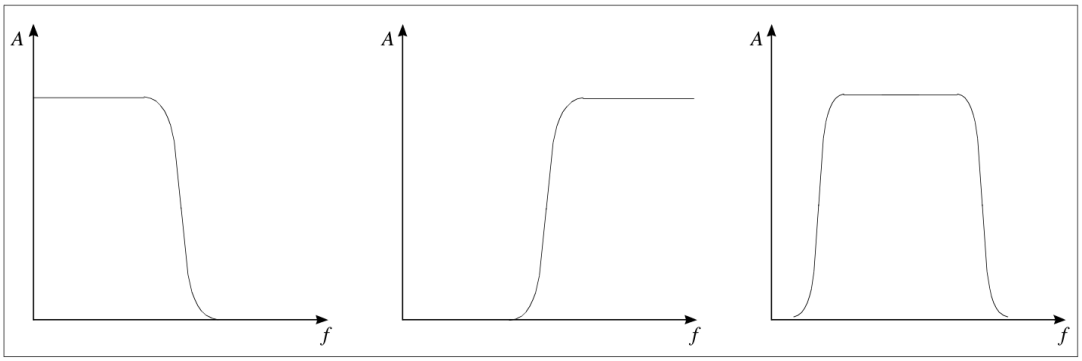


Figure 3: From left to right low-pass, high-pass, band-pass filter.

A common way of expressing the transfer function of the filter output $y[n]$, where n is the sample number, as a function of the input $x[n]$ of a filter, is as follows:

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

The first term in the right part of the equation relates the output of the filter to pass-through inputs. The second term relates the output of the filter to past outputs. We can distinguish two types of filters:

- *Infinite Impulse Response (IIR)* filters in which both terms of equation are present. They are also known as infinite response filters, since the second term describes recursive behaviour (output is a function of output).
- *Finite impulse response (FIR)* filters in which only the first term of equation is present. The output signal of a FIR filter is a weighted sum of the input signal. The impulse response of an FIR filter has a length of N output data. This is the filter that we are going to implement during this project.

We can highlight the following properties of FIR filters:

- The unit impulse response of an FIR filter is the sequence of its coefficients $alpha[m]$. This gives us a simple way to check the performance of the filter. All we have to do is introduce a unit delta (x(0)=1, x(n)=0 for n other than 0, also known as Dirac delta) and we get the sequence of coefficients.

- If all coefficients are multiplied by a constant, the gain of the filter is changed, not its characteristics.
- The continuous gain of the filter is equal to the sum of all its coefficients.

6.2 Hardware implementation of FIR filters

The Z-transform of an FIR filter is given by the following expression:

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

Where z^{-n} in the equation corresponds to delay lines and c_n are the coefficients that weight the input samples, equivalent to the $\alpha[m]$ in equation 1. In the above equation N is the order of the filter, which also corresponds as the number of filter stages. The signal flow diagram of a 5-stage FIR filter is shown in the figure below. Note that the last delay line (the register at the output) is not part of the above equation.

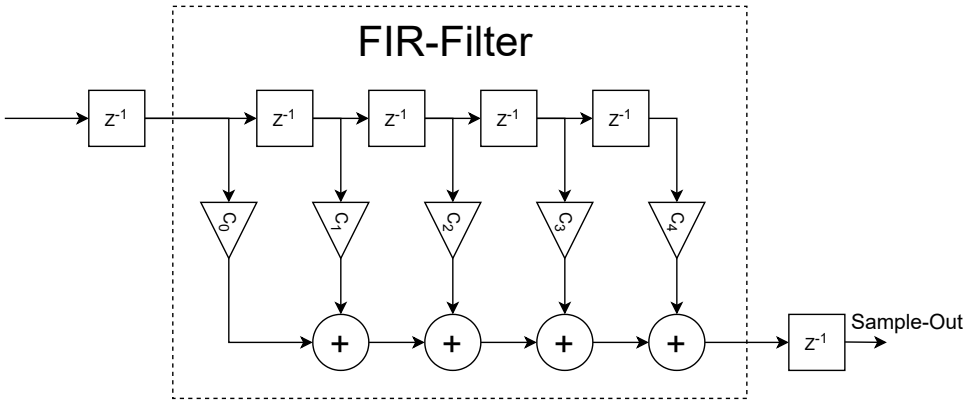


Figure 4: FIR filter 5 stages freq domain.

The z^{-n}, corresponding to the delays, are implemented using registers. Multipliers are used to apply the coefficients to each sample. Partial sums are done through adders. The following figure shows the direct implementation of a 5-stage FIR filter.

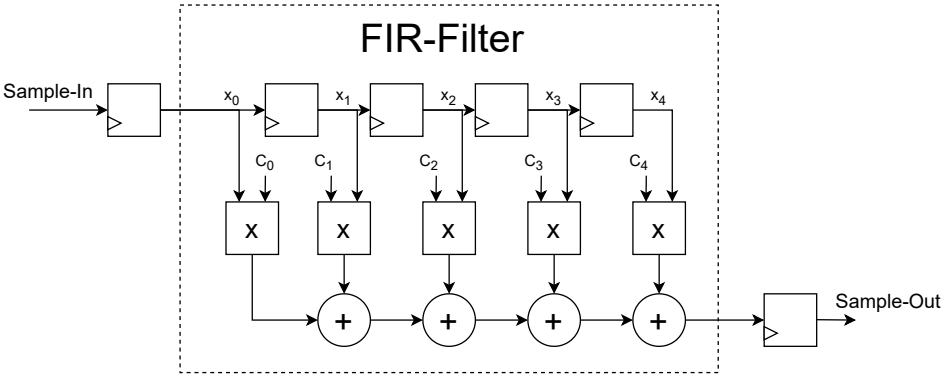


Figure 5: FIR filter 5 stages time domain.

6.3 Data Path based systems

Most digital systems are based on a data path and a controller (usually in the form of a finite state machine). The data path consists of storage units (such as flipflops, registers and memories) and combinational units (such as ALUs, multiplexers, shifters, comparators, etc.) connected by buses (lines of various interconnections). The values stored in the storage units are read after the rising (or falling) clock edge, processed in the combinational units and stored in the storage units on the next clock edge. On the other hand, the controller provides control information and receives status information from the data path. It is usually implemented through a finite state machine controlled by inputs and outputs from the outside world. The first stage in the design of a data path-based system is the algorithmic description of the functional specification of a digital system. From this algorithmic description the data path is constructed.

6.4 Implementation methodology for signal processing algorithms

The general process of implementing algorithms for digital signal processing is usually divided into the following stages:

- Allocation.
- Temporal planning.
- Binding.
- Implementation.

The process is not strictly linear, but we may have to go back a stage to perform a particular optimisation, or we may be able to make decisions about subsequent stages without having completed an intermediate step.

To illustrate each of these stages, we will use the case of the five-stage FIR filter seen in the previous sections, which can be represented by the following graph or flowchart.

6.4.1 Allocation

Allocation refers to the selection of the number and type of hardware units that will implement the functions described by the algorithm's flowchart. In the case of the five-stage FIR filter, we could decide to implement it with one multiplier and one adder, two multipliers and one adder, two half multipliers and two adders, and so on. Depending on the number and type of units chosen, different values of area, throughput and latency will be obtained, so the system constraints must be taken into account when making this decision.

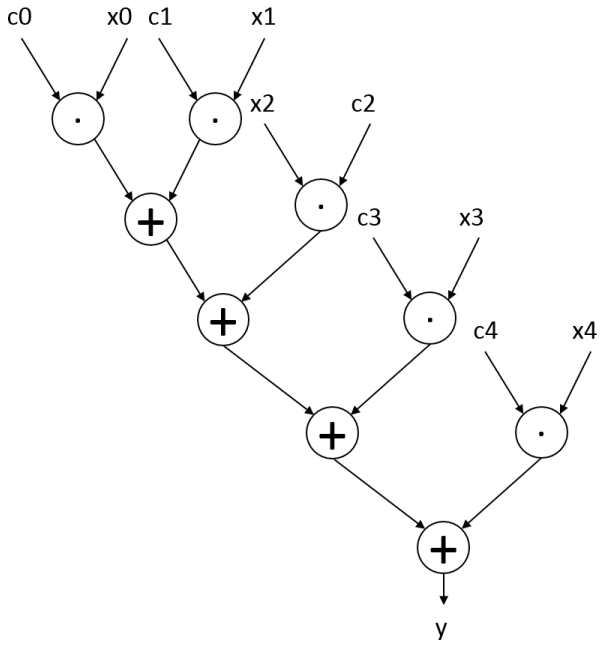


Figure 6: Step 1 designing a 5 stages FIR filter.

6.4.2 Temporal planning

During this phase, the operations of the flowchart are distributed over time, taking into account that the number of available units cannot be exceeded. At this point, you can play with the distributional properties of the addition and multiplication operands to rearrange the flowchart and obtain more interesting structures from a hardware point of view.

By setting an allocation of two multipliers and one adder, the following figure shows the timing of the FIR filter flowchart. Each horizontal line separates the operations that occur during one control cycle from the next. In this case, to reduce the latency of the algorithm, instead of doing $((c0 \cdot x0 + c1 \cdot x1) + (c2 \cdot x2) + (c3 \cdot x3))$, which would waste the option of using two multipliers simultaneously, $((c0 \cdot x0 + c1 \cdot x1) + (c2 \cdot x2 + c3 \cdot x3))$ has been proposed.

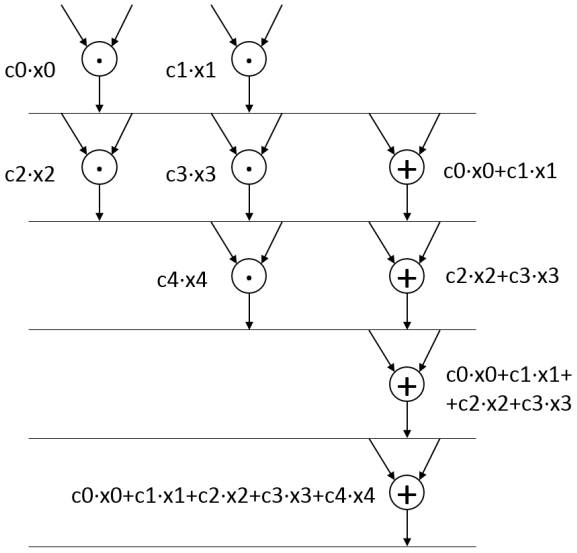


Figure 7: Step 2 designing a 5 stages FIR filter.

6.4.3 Binding

Each particular unit is associated with a particular operation in each time unit. It is also specified to which particular port each operand is connected. The internal signals are named to facilitate the procedures of the following phases. The following figure shows the result of this phase for the FIR filter.

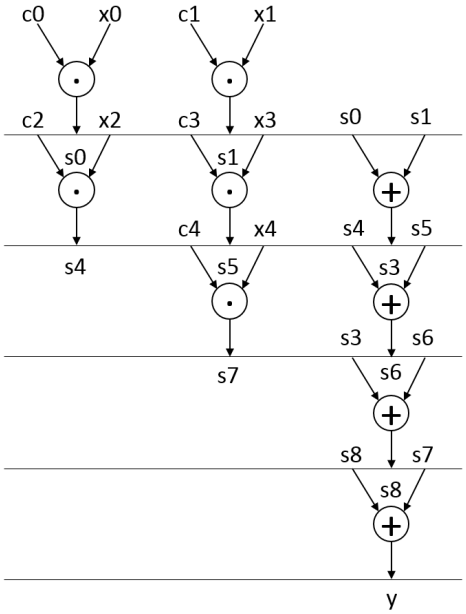


Figure 8: Step 3 designing a 5 stages FIR filter.

6.4.4 Implementation

This is the final phase of the design and results in a hardware structure that can be implemented, for example, in an FPGA. This stage can be subdivided into different tasks: analysis of the lifetime of the variables, minimisation of the number of registers and multiplexers and creation of the hardware structure.

In the analysis of the lifetime of the variables, the need for internal storage (registers) of the structure that we are going to generate is studied. From the diagram generated after linking, we can analyse the moments in which the internal signals are produced and the moments in which they are consumed, thus obtaining their lifetime. In the following figure, the lifetime of the different variables is represented by an arrow. The maximum number of parallel arrows, three in the example, establishes the number of registers required. Once the number of necessary registers, R1, R2 and R3, has been established, we can assign each variable to a register for each moment. In the example, register R0 will store s0, s4, s6 and s8 as the algorithm unfolds.

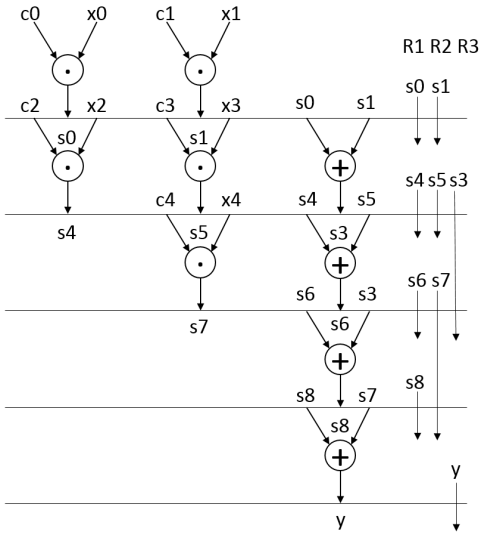


Figure 9: Step 4 designing a 5 stages FIR filter.

Each input of each operator and register receives different data over time, so we have to assign multiplexers that control which input is connected at which time. The following figure studies the possible connections that the arithmetic modules of the FIR filter may need.

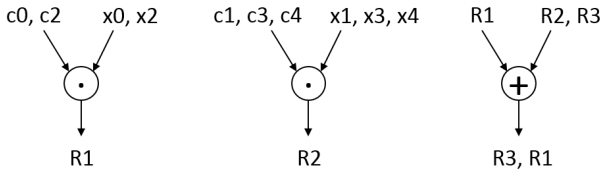


Figure 10: Step 5 designing a 5 stages FIR filter.

Finally, with all the information we have gained in the different stages, we can now realise a hardware structure that is capable of performing the digital processing algorithm with the units

we have chosen in the allocation stage. The following structure shows the implementation of the five-stage FIR filter with two multipliers and an adder.

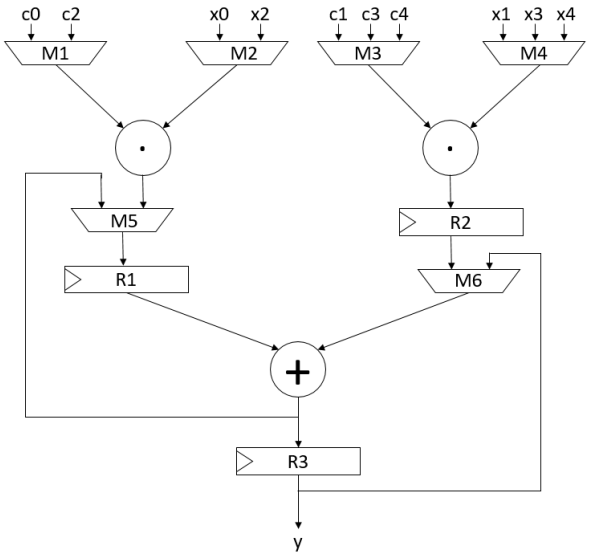


Figure 11: Step 6 designing a 5 stages FIR filter.

This is as far as the steps for the creation of the data path go. Logically, an external controller is expected to provide the control signals for the multiplexers and the register loads at the appropriate time. To implement the controller, it is useful to have a timeline that specifies at each instant of time which signal each multiplexer has to let through and which signal each register has to have stored. The following figure shows the timeline for the FIR filter example. At each time instant it is specified which signal each multiplexer has to let through and the associated control signal in brackets, as well as the data stored in the registers for each time instant.

	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

Figure 12: Step 7 designing a 5 stages FIR filter.

6.5 Fixed point notation

In this block we are going to work with numbers represented in fixed-point notation. An analogy can be drawn between fixed-point decimal numbers and fixed-point binary numbers. A binary fixed-point number looks like this: "10001110.101". The dot of a fixed-point binary number is placed to the right of the digit representing 2^0. The digits to the left of the dot are interpreted as binary without a dot and the digits to the right represent the decimal or fractional part. The figure below shows two examples of fixed-point notation using two's complement. Look at the figure and make sure you understand the conversion procedure between fixed-point binary and base 10.

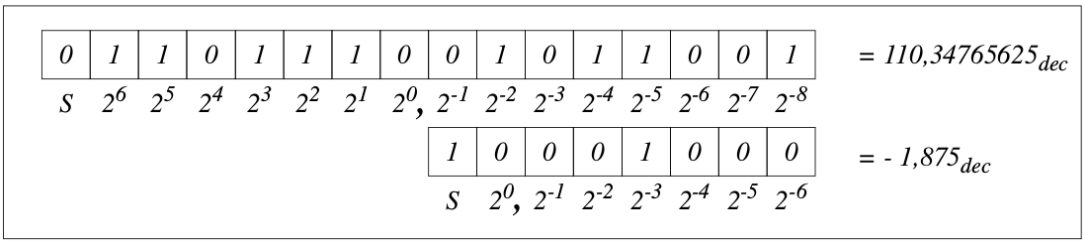


Figure 13: Fixed point examples.

For this block we will use the following notation to represent the sizes of fixed-point numbers: $< a.b >$ where a is the number of bits representing the integer part and b is the number of bits needed to represent the decimal part. For example the top number in the figure above needs 16 bits, and its size would be represented by $< 8.8 >$. This notation is useful to establish the sizes resulting from multiplication and addition operations: Multiplication: if $< 3.4 >$ is multiplied with $< 5.6 >$ the result will be $< 3+5.4+6 >=< 8.10 >$. If at the output we have to provide a 10-bit data, we truncate the result and get $< 8.2 >$ (or $< 7.3 >$ if we know that the final result of accumulating these numbers will not produce an overflow). Addition: If $< 3.4 >$ is added to $< 5.6 >$ the result will be $< max(3,5)+1$, $max(4,6) > = < 6.6 >$. The +1 in the integer part takes into account possible overflows (output carries).

Task 2.1:

☞ What is the range of an 8-bit fixed-point number $< 1.7 >$ in two's complement? What is its precision?

6.6 Filter block specifications

In this block we will implement a 5-stage FIR filter, whose interface is described in the following VHDL entity:

```
-- filterBlock.vhd
Entity filterBlock Is
port (
    clk_12Mhz : in STD_LOGIC;
    rst : in STD_LOGIC;
    --Input samples
    sample_in : in signed (sample_size-1 downto 0);
    sample_in_enable : in STD_LOGIC;
    --Output samples
    sample_out : out signed (sample_size-1 downto 0);
    sample_out_ready : out STD_LOGIC;
    --Filter selection
    filter_select : in STD_LOGIC
  );
end filterBlock;
```

- Both `sample_in`, `sample_iut`, and filter coefficients must be encoded with 8-bit two's complement words $< 1.7 >$.
- `sample_in` is a control input that reports when the value of `sample_in` has been updated with an active pulse during a clock cycle.
- `filter_select` is a control input that selects the filtering to be performed: '0' for low pass filter, '1' for high pass filter.
- `sample_out` is a control output that reports when the `sample_out` value has been updated with an active pulse during a clock cycle.
- The data path uses only a multiplier in pipeline and an adder.
- Coefficients for LPF: c0 = c4 = +0.039, c1 = c3 = +0.2422, c2 = +0.4453.
- Coefficients for HPF: c0 = c4 = -0.0078, c1 = c3 = -0.2031, c2 = +0.6015.

Task 2.2:

Draw on paper the implementation of a 5-stage FIR filter with the following assignment: Two half-multiplier and one adder. Perform all the steps described in the section 6.4.

- Task 2.2.1: Temporal planning.
- Task 2.2.2: Binding, life time of variables and register allocation.
- Task 2.2.3: Connection analysis for multiplexer extraction.
- Task 2.2.4: Implementation.
- Task 2.2.5: Timeline.

Task 2.3:

☞ Perform a quantisation analysis of the signals, specifying how many bits you are going to use in each signal and in which position the decimal point is going to be. Use the notation and methodologies presented in class.

The recommended filterBlock design is depicted in the Figure 14. Feel free to change the design if you want, in that case show your new design to the professors before the next steps.

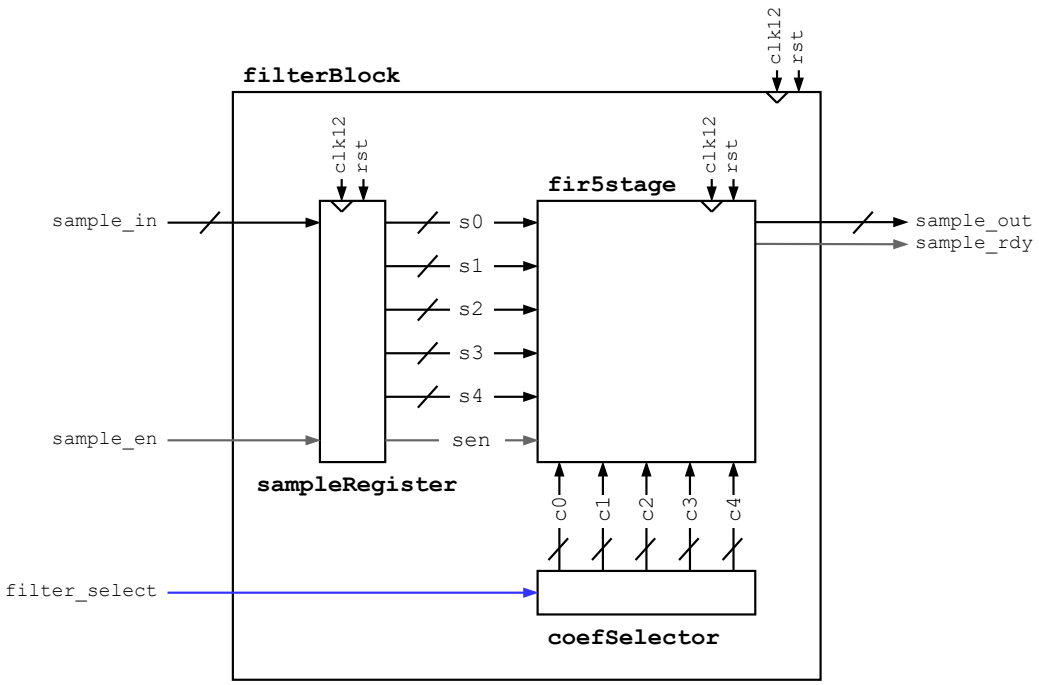


Figure 14: filterBlock recommended design.

Task 2.4:

☞ Write VHDL skeletons modules for all components of your `filterBlock` and interconnect them in your toplevel.

Task 2.5:

☞ Code the behavioural of `sampleRegister` and `coefSelector`. Remember that the second one should be full combinational logic.

Task 2.6:

Code the `fir5stage`. To do so we recommend to follow one of the next ideas:

1.
   - Divide fir5stage in submodules. One for the datapath and one for the state controller and mux.
   - Implement the datapath module alone and check it behavioural using a testbench.
   - Draw a state diagram for the filter controller and follow the FSM methodology to implement it.
   - Write the code of the fir5stage and join all the parts.
2. Code all together. Try to differentiate all the blocks above even if you are coding then in the same vhd file.

Task 2.7:

Test the fir5stage block with a testbench. Use a pulse of 0.5 as an input. What do you expect at the output when the input is (0,0,0,0,X,0,0,0,0)? Consider X the biggest and lowest positive and negative values (four cases). Perform a testbench to check the correct working process on each of those cases. Use the LPF coef.

Task 2.8:

Test the fir5stage with the next input sequence (0, 0.5, 0, 0.125, 0, 0, 0, 0, 0, ...) What do you expect in the output? Check that the expected values match with the obtained values.

6.7 Advanced Testbench

If we were to do more extensive testing to check the performance of our filter, the use of the testbenches we have used so far would become impractical, mainly for two reasons:

- Every time we need to test a new input sequence, we have to modify and recompile the testbench.
- To check that the output sequence we get is correct, we have to have previously calculated it by hand and, after running the testbench, we have to check one by one that the data is correct.

To solve these problems we can make use of VHDL file writing and reading options. Files accessed from VHDL code can either contain data of a single type (`std_logic_vector`, `integer`, `singed,` etc.) or they can be of TEXT type. We are going to deal only with the latter type of files, those containing TEXT data, which are ASCII files. TEXT files are read line by line using READLINE, which stores the corresponding line in a signal or variable of type LINE. TEXT files are written line by line using WRITELINE. READ is used to extract the data contained in a LINE type signal/variable, and the opposite operation (writing to a LINE type) is done through WRITE. Both READ and WRITE operate on bit, bit-vector, boolean, character, integer, real, string and time.

Matlab writes and reads the data in decimal notation, readable by us. Therefore we have to convert the data in the file we read from integer (`integer`) to binary (`signed`). To do this we are going to use the following function from the `IEEE.NUMERIC_STD` package. In the same way, to write to the file, we have to convert our data, which is `signed` to `integer`.

- `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.
- `my_int <= to_integer(signed_number)` where `signed_number` is the `signed` that we want to cast, and `my_int` is the `integer` where the cast is assigned.

When converting between `signed` and `integer` and vice versa, the program is not able to work with the representation $< 1.7 >$ or with any other fixed-point representation, it will always consider the notation $< 8.0 >$, that is, without decimal point. This implies that the data in the file will be between +127 and -128.

In appendix 1 you can find an example of a testbench reading from a TEXT file to give you an idea of how it works.

Task 2.9:

Write VHDL code for a testbench that reads samples from one file and writes the results to another.

☞
- Call the file containing the input data `sampleIn.dat` call the output file `sampleOut.dat` .
- Fill the input file with a sequence that introduces a single pulse. Each line of the file corresponds to a data item.

Task 2.10:

Use the `haha.wav` file in Matlab to create an input file for your testbench. Appendix 2 details the sequences you need to use in Matlab to create an input file for your testbench:

☞
- Load an audio file in Matlab and create another file with the necessary format for your testbench.
- Use the Matlab function `filter` to obtain the response of a FIR filter with real precision.
- Load and listen to the output produced by your testbench in Matlab.

Task 2.11:

☞ Use your testbench to process the input file provided by Matlab. Write the filtered data to the file "sample_out.dat", which you will later import into Matlab.

Task 2.12:

☞ Import your "sample_out.dat" file into Matlab. Compare the testbench results with the real precision values provided by the Matlab function "*filter*". Make an error plot of your results (subtract your results from the real precision data).
Use the Matlab function *sound* to listen to the original waveform. Then compare it with your filtered sound and see if there is any difference between the real precision filter and your filter.

7 Block 3: Controller and Memory

In this third block, the audio system is going to be finalised. Specifically, a core IP will be added to the design, which will encapsulate a RAM memory to store the audio samples, a controller will be designed to orchestrate all the operations of the system and all the elements will be integrated.

The design of the controller is completely open and the student is given the possibility to practice the skills acquired throughout the course, designing the system from scratch taking into account the set of specifications that define it.

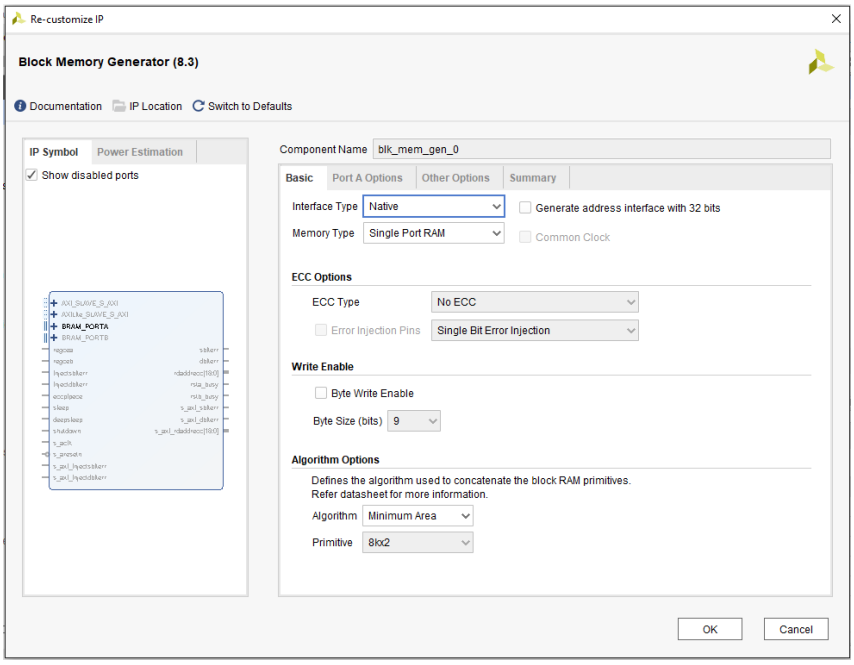
7.1 RAM

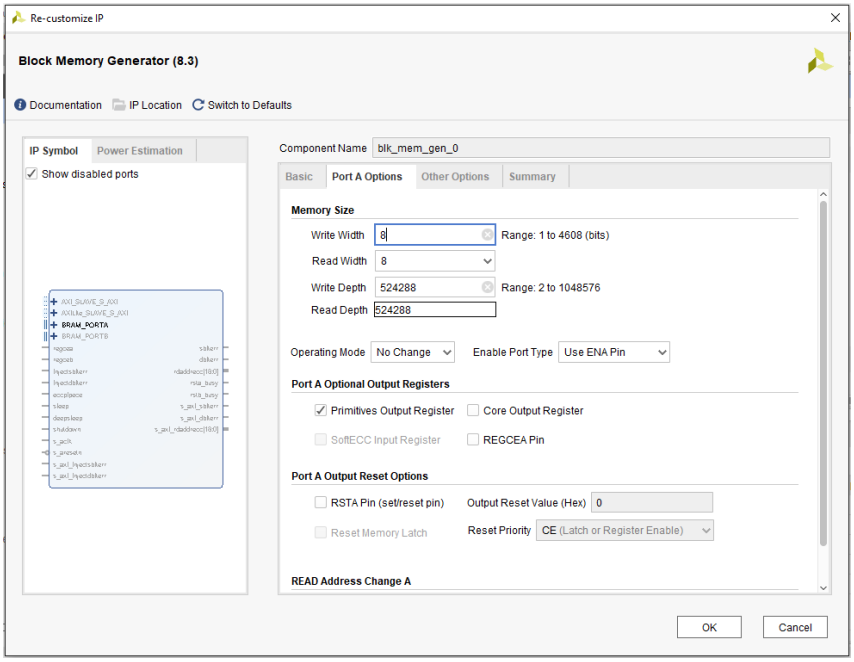
The recorded samples are to be stored in a RAM memory. The characteristics of the memory are as follows:

- Size of each stored word: 8 bits.
- Size of the address bus: 19 bits.
- Number of words stored. $2^{19} = 524288$.
- Size in bits of memory: $2^{19} * 8 = 4194304$.

To instantiate this memory in our design we will use the "IP Catalog" of Vivado, where we have to look for the following option: "Memories and Storage Elements", "RAMs and ROMs and BRAM", Block Memory Generator.

The following two images show how to configure the wizard to encapsulate the required memory.





Task 3.1:

Create a testbench running at system frequency to test and understand the operation of the encapsulated memory. Then write down the function of each port and the timing of the memory (a short schedule including a write and a read).

Task 3.2:

Determine how much recording time can be stored in memory.

The next transformation is to go from a $< 8.0 >$ scheme to a $< 1.7 >$ scheme. But is it really necessary to make any changes to our data? The place where we place the decimal point is an abstraction that allows us to calculate the equivalent base 10 value and with it adjust the value of the filter coefficients, but the hardware is going to be identical regardless of the place of the decimal point. Therefore, this second transformation is not necessary, the first one is.

7.2 Controller full system

The controller is the only module of the project where the design process is not guided and it is the last piece of the puzzle to build the overall system. From the specifications given and knowing the functionalities and interfaces of the different blocks, decide how you are going to implement the functionality of the controller. In particular, decide which design style you are going to use (state machine, state machine with associated data path, a system with different sequential elements, etc.), divide the design in such a way that you can implement it progressively adding new functions when the previous ones are secured, plan how you are going to test the operation of each specification and make a time schedule.

Task 3.3:

☞ Add all the figures necessary to describe your design and the planning to carry it out. You can include schematic diagrams, explanatory timelines, a test plan, a time schedule, and anything else you feel necessary to explain the decisions you have made. Notify the professor before starting the design to get their approval.

Final tasks:

☞ Write a short report of the work done and the decisions taken as well as the designs made and export it as a pdf file. Don't worry about the figures, a clear photo of the design on paper will be fine.
Export all the vhd files, including the tests, in a zip file together with the memory and upload it to moodle.

8 Appendix 1

Example of testbench reading from a text file. Remember to enter the full path to the file you want to read or write.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;
use std.textio.all;

ENTITY ejemplo_tb IS
END ejemplo_tb;

ARCHITECTURE behavior OF ejemplo_tb IS
   -- Clock signal declaration
   signal clk : std_logic := '1';

   -- Declaration of the reading signal
   signal Sample_In : signed(7 downto 0) := (others => '0');

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

  -- Clock statement
   clk <= not clk after clk_period/2;

read_process : PROCESS (clk)
  FILE in_file : text OPEN read_mode IS "sample_in.dat";
  VARIABLE in_line : line;
  VARIABLE in_int : integer;
  VARIABLE in_read_ok : BOOLEAN;
BEGIN
if (clk'event and clk = '1') then
  if NOT endfile(in_file) then
    ReadLine(in_file,in_line);
    Read(in_line, in_int, in_read_ok);
    sample_in <= to_signed(in_int, 8); -- 8 = the bit width
  else
    assert false report "Simulation Finished" severity failure;
  end if;
end if;
end process;

END;

```

9 Appendix 2

Load a .wav file into Matlab and creating another file in the format used by the testbench.

```
[data, fs] = audioread('haha.wav');
file = fopen('sample_in.dat','w');
fprintf(file, '%d\n', round(data.*127));
```

Use of the Matlab function *filter* to obtain the response of a FIR filter with real precision.

```
test = filter([0.5, -0.5, 1.0, -0.5, 0.5],[1, 0, 0, 0, 0], data);
sound(test);
```

Remember to modify this command with the coefficients of the filters you are going to use.

Load and listen to a file with the output format of the testbench. Attention, it is important not to forget the division by 127 in order not to destroy the loudspeakers.

```
vhdlout=load('sample_out.dat')/127;
sound(vhdlout);
```