Trabajo Práctico Final:

Análisis de Señales y Sistemas Digitales

Grupo 1

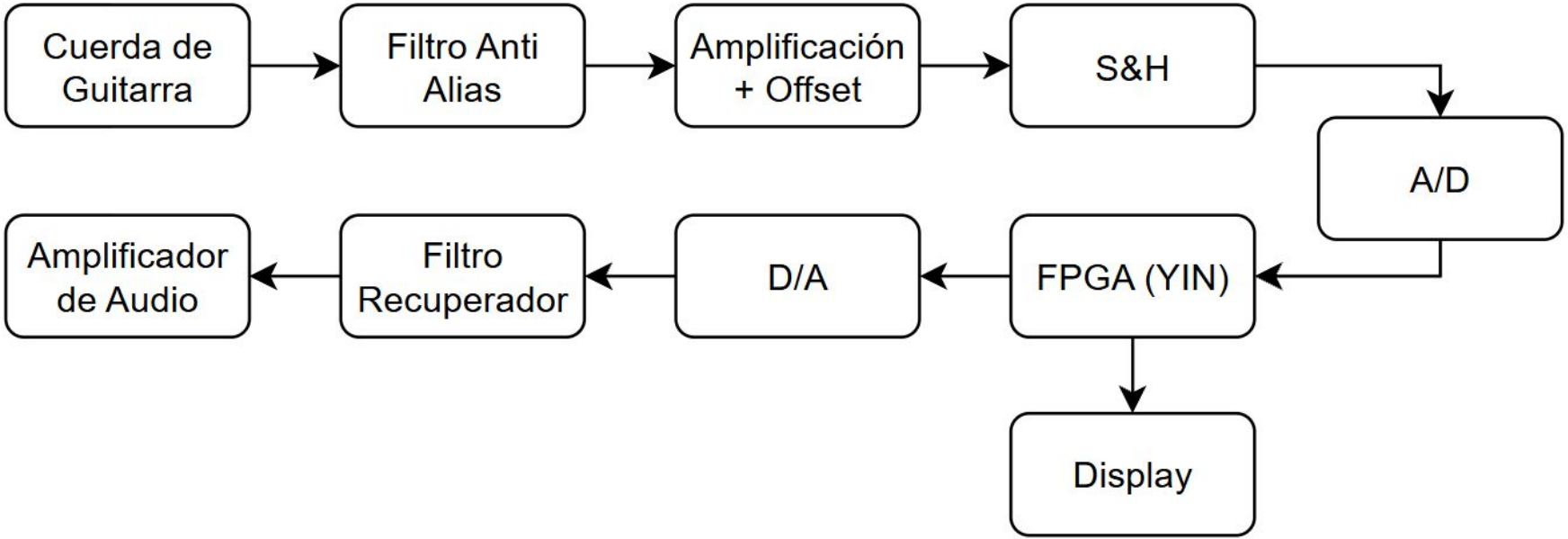
Proyecto Final ASSD

Se eligió como proyecto final crear un afinador de guitarra, utilizando conocimientos de la materia como el procesamiento de señales analógicas y digitales. El proyecto incluye sampleo con S/H, conversor A/D, conversor D/A, Filtro Anti-aliasing, utilización de una FPGA, y conexión a un amplificador de audio.

Se incluirá como parte de investigación el algoritmo de YIN el cual fue analizado previamente en el TP N4.

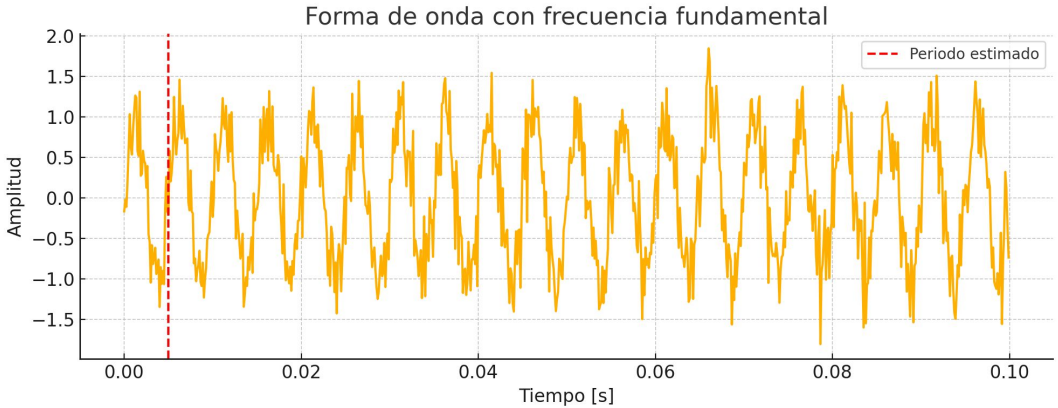
Este proyecto se implementó en Verilog con el fin de estimar la frecuencia de una señal de audio proveniente del sonido de la cuerda de una guitarra mediante análisis del algoritmo YIN.

Diagrama de Flujo



Algoritmo YIN: Problema

- Encontrar el Pitch de una nota musical, o del canto a partir de la voz humana. El pitch es la frecuencia fundamental de una señal de audio.
- Áreas de aplicación: Voz humana, música.
- Los métodos tradicionales (autocorrelación, cepstrum) presentan limitaciones.
- Problemas comunes: ruido, armónicos, errores de detección.



Entrada de datos

Para poder implementar el algoritmo mencionado, primero se debe obtener los datos de entrada para realizar el análisis correspondiente con respecto a la nota a evaluar.

A partir de los conceptos del trabajo sobre conversores, se estableció un PCB capaz de recibir entrada de audio mediante un micrófono, amplificada y con un offset tal que la señal quede entre 0 y 5V. Luego, se conecta al conversor A/D del tipo SAR, y se pueden ver los 8 bits de salida.

Como se acaba de mencionar, la entrada a la FPGA será una señal de 8 bits. Esto quiere decir que se convierte la señal de audio de la guitarra a 8 bits.

Implementación del algoritmo

Se implementó el algoritmo en una FPGA.

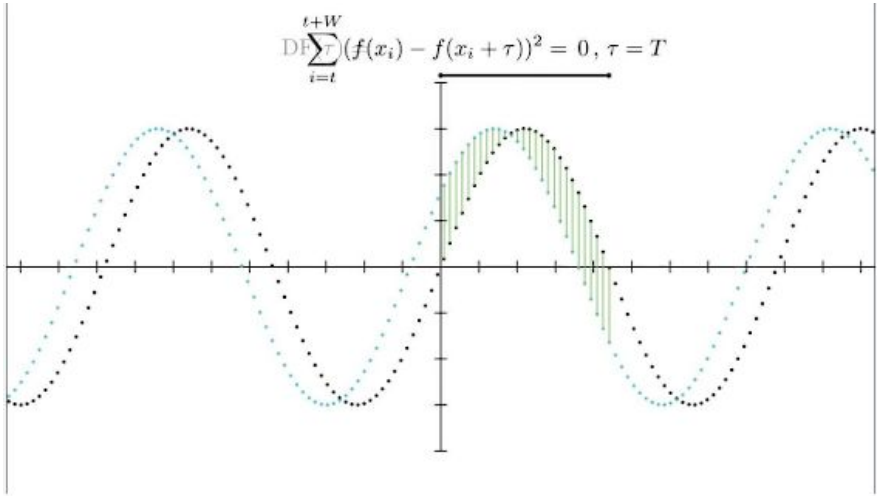
A partir de la entrada de 8 bits, se programó en verilog el código para reproducir el algoritmo YIN.

Para esta implementación, se utilizó un clock muy alto en la FPGA. De esta manera, se pueden realizar las operaciones y aproximar el valor correcto del tau (τ) asociado a la frecuencia fundamental, antes que el sample & hold vuelva a samplear y cambio los datos de entrada en la FPGA.

Aún así, para no tener problemas de este tipo, se creó un arreglo donde se guardan los datos con los que se están realizando las operaciones, y además, hay memoria para guardar una etapa más de sampling. Una vez se llega al resultado, los datos guardados en memoria se utilizan y lo nuevo que se samplea se guarda para la próxima iteración.

Algoritmo YIN

- YIN es un algoritmo robusto para estimar el pitch con alta precisión.
- Basado en la autocorrelación modificada con diferencia acumulativa.
- Preciso, robusto frente a ruido y eficiente computacionalmente.

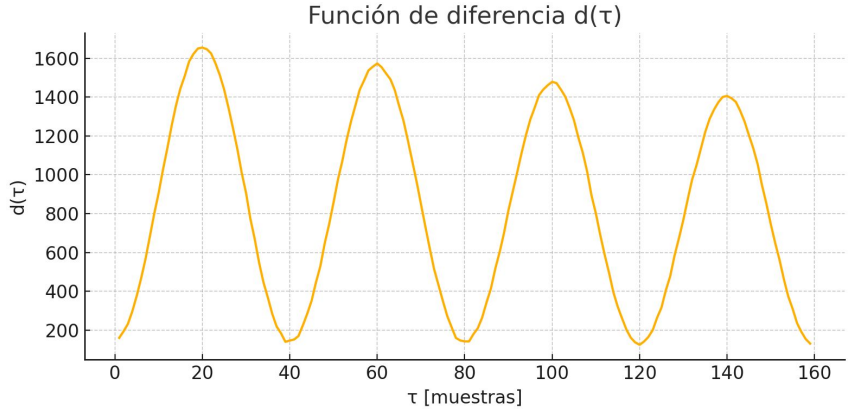


Función de diferencias

Se considera la siguiente función de diferencias con una ventana de W muestras:

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2$$

A partir de esta función, se busca encontrar el mínimo entre x(n) - x(n + T) en una ventana W.



Mejora al modelo matemático de diferencias: CMNDF

$$d'_t(\tau) = \begin{cases} 1, & \tau = 0, \\ \dfrac{d_t(\tau)}{\frac{1}{\tau} \cdot \sum_{j=1}^{\tau} d_t(j)}, & x \geq 0 \end{cases}$$

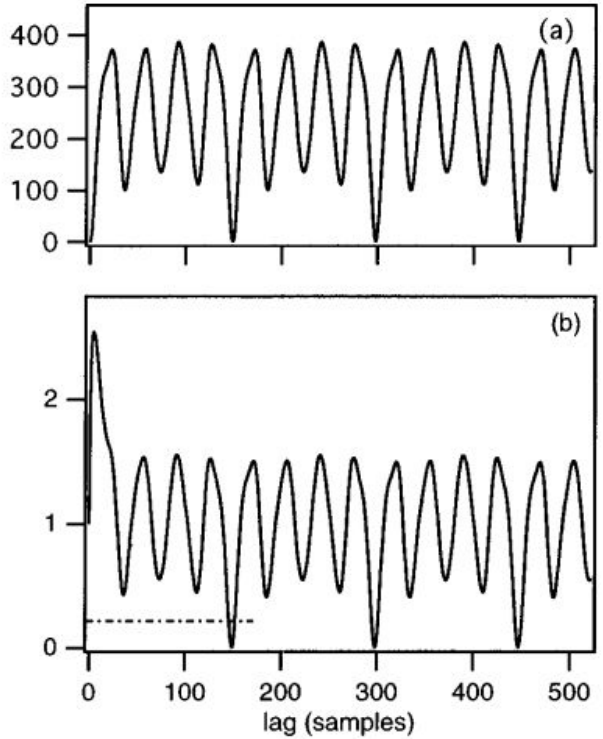


FIG. 3. (a) Difference function calculated for the speech signal of Fig. 1(a). (b) Cumulative mean normalized difference function. Note that the function starts at 1 rather than 0 and remains high until the dip at the period.

Selección de frecuencias dentro del punto de threshold

En el caso que un armónico superior al fundamental se corresponda a un valor menor de la función de diferencias modificada, el algoritmo lo reconocerá como el fundamental. Para evitar esto se propone lo siguiente:

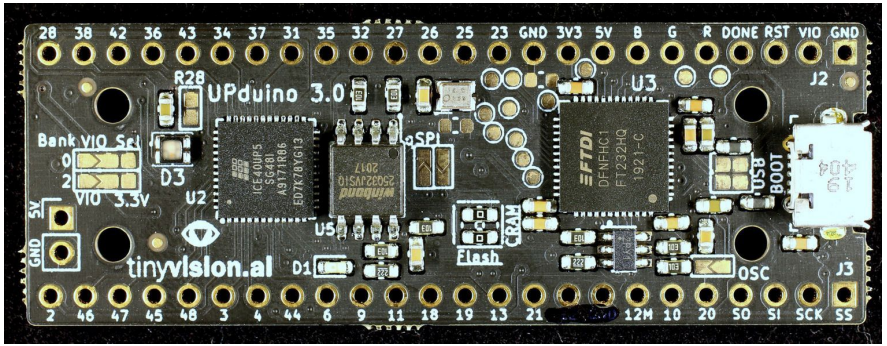
- Seleccionar un valor de threshold *a*
- Todo valor de tau que evalúe a un valor menor a *a* es agregado a una lista
- Se selecciona el menor valor de tau de la lista

Interpolación Parabólica:

La teoría anterior funciona correctamente solo si el período es múltiplo del período de muestreo. Para solucionar esto, se puede realizar una interpolación parabólica a la función de diferencias modificada. Esto es más barato en términos computacionales que realizar un upsampling.

Implementación en una FPGA

Se decidió implementar el algoritmo en una placa FPGA Upduino 3.0. A continuación se detallará el funcionamiento de cada módulo. La entrada es una señal de audio de 8 bits y un bit de EOC de un ADC. El circuito funciona a 24MHz.



Función de diferencias - Diff_module

Este módulo representa el núcleo del kernel del algoritmo. Recibe un stream de datos y un tau, y calcula la función de diferencias en una ventana W.

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2$$

```
module diff_module #(
    parameter WINDOW_SIZE_BITS = 8,
    parameter INTERMEDIATE_DATA_WIDTH = 32,
    parameter DATA_WIDTH = 16,
    parameter MAX_TAU = 40
) (
    input clk,
    input wire [(2**WINDOW_SIZE_BITS + MAX_TAU)*DATA_WIDTH-1:0] data_in,
    input wire [5:0] tau,
    input wire reset,
    output reg ready,
    output reg [INTERMEDIATE_DATA_WIDTH-1:0] accumulator
);
```

Función de diferencias - Diff_module - Precisión

Se truncan 2 bits de precisión para cada evaluación de $(x_j - x_{j+\tau})^2$. Luego se truncan otros 2 bits al resultado total de la función de diferencias. Esto se realizó para reducir el número de bits de los registros utilizados y como consecuencia reducir el costo de la implementación física.

```
// guardo y aumento
new_accumulator <= accumulator + (diff*diff >> 2);
new_sum_index <= sum_index + 1;

if (sum_index == (2 ** WINDOW_SIZE_BITS) - 1) begin
    new_ready <= 1'b1;
    new_accumulator <= accumulator >> 2;
end
```

Función de diferencias modificada - Modiff_module

Este módulo recibe el stream de datos y calcula todos los valores de la función de diferencia modificada. Pasos:

- Calcular dτ para todo τ
- Calcular promedio
- Realizar normalización

```
module modiff_module
#(
    parameter DATA_WIDTH = 8,
    parameter INTERMEDIATE_DATA_WIDTH = 64,
    parameter WINDOW_SIZE_BITS = 8,
    parameter MAX_TAU = 40
) (
    input wire clk,
    input wire reset,
    input wire [(2**WINDOW_SIZE_BITS + MAX_TAU)
                *DATA_WIDTH-1:0] data,
    output reg ready,
    output reg [MAX_TAU*INTERMEDIATE_DATA_WIDTH-1:0] results,
    output reg [INTERMEDIATE_DATA_WIDTH-5:0] average
);
```

Función de diferencias modificada - Modiff_module

Para calcular dt original utiliza 1 módulo *diff_module* y calcula secuencialmente. Posible optimización: implementar de forma paralela. Optimizar área utilizada.

Para calcular el promedio utiliza un módulo de división *sar_divisor_module*.

Para calcular los valores de dt modificada utiliza el módulo divisor y normaliza con respecto al promedio para obtener valores enteros.

$$d'_t(\tau) = \begin{cases} 1, & \tau = 0, \\ \dfrac{d_t(\tau)}{\frac{1}{\tau} \cdot \sum_{j=1}^{\tau} d_t(j)}, & x \geq 0 \end{cases}$$

Función de diferencias modificada - Modiff_module - Precisión

Para calcular cada valor de la función modificada se truncan 7 bits de precisión del numerador y del denominador. Esto se realizó dado que se deben acumular todos los valores previos. Esta precisión no es un factor que impacte en los resultados.

```
new_dividing <= 1;
curr = diff_results[sum_index*INTERMEDIATE_DATA_WIDTH+:INTERMEDIATE_DATA_WIDTH];
new_dividendo <= (((curr >> 7) * average) * sum_index);
new_divisor <= (accumulator + curr ) >> 7;
new_accumulator <= (accumulator + curr) >> 7;
new_div_reset <= 1;
```

Modulo división entera - Sar_divisor_module

Se implementó un módulo que realiza la división entre dos números enteros de N bits. El algoritmo se inspira en el principio del conversor SAR.

```
module sar_divisor_module #(
    parameter BITS = 16
) (
    input wire clk,
    input wire [BITS-1:0] dividendo,
    input wire [BITS-1:0] divisor,
    output reg [BITS-1:0] result,
    input wire reset,
    output reg ready
);
```

Modulo división entera - Sar_divisor_module - Funcionamiento

El divisor numérico consiste en un seguidor que se aproxima al dividendo hasta que la diferencia sea menor al error (divisor). Para corregir al seguidor se le suma o resta valores del divisor shifteado N veces. Una vez cambia el signo del error, se acumula el valor de N (positivo si se está sumando, negativo en caso contrario) al igual que el valor del seguidor. Cuando el error del seguidor es menor al divisor, se asegura que el error sea positivo y se devuelve el resultado calculado. Cada shifteo se realiza en su propio clock.

Selección de resultado - Min_tau_module

El modulo *min_tau_module* calcula los valores de la función de diferencias modificada y calcula el mínimo que cumpla con el threshold. Para calcular los valores utiliza un módulo *modiff_module*. Una vez estos valores están listos, calcula el valor de threshold con el promedio e itera por los resultados hasta encontrar uno que sea menor. El resultado es el valor de tau.

```
module min_tau_module #(
    parameter DATA_WIDTH = 8,
    parameter INTERMEDIATE_DATA_WIDTH = 64,
    parameter WINDOW_SIZE_BITS = 8,
    parameter MAX_TAU = 40,
    parameter THRESHOLD = 1
) (
    input wire clk,
    input wire reset,
    output reg ready,
    input wire [(2 ** WINDOW_SIZE_BITS + MAX_TAU)
                *DATA_WIDTH-1:0] data,
    output reg [7:0] min_tau
);
```

Módulo superior - *Top_module*

Este módulo se encarga de organizar el funcionamiento del dispositivo. Por un lado, con una máquina de estados controla el estado del circuito. Los estados son WAITING_DATA, COPYING_DATA, WAITING_PROCESSING y SWITCH. A medida que los datos ingresan al circuito son almacenados en un buffer doble utilizando el *buffer_module*. Una vez se llene una partición del buffer, el circuito comienza a copiar los datos para luego procesar el algoritmo. Una vez terminado el procesamiento se vuelve al estado de espera de datos. Notar que la entrada de datos ocurre de manera simultánea con cualquier otro proceso y tiene prioridad. Finalmente, este módulo se encarga de mandar la información al *display_module*.

Memoria - *Buffer_module*

Este módulo implementa una memoria de tipo RAM. Sus entradas son address, data_in, write y output_enable y sus salida es data_out.

```
module buffer_module
#(
    parameter DATA_WIDTH = 16,
    parameter DEPTH = 16,
    parameter ADDRESS_WIDTH = $clog2(DEPTH)
)
(
    input wire clk,
    input [ADDRESS_WIDTH-1:0] address,
    input [DATA_WIDTH-1:0] data_in,
    output [DATA_WIDTH-1:0] data_out,
    input wire write,
    input wire output_enable,
    input wire operational_clock
);
```

Resultados

Una vez procesada la información recibida en la FPGA. El resultado a partir del algoritmo YIN se visualiza en un display de 4 dígitos BCD.

Este display es parte de una placa la cual recibe secuencias de datos de 16 bits.

Cada vez que se terminan las operaciones del algoritmo, se activa un flanco en ‘data_ready’, el cual indica que el display ya puede mostrar el resultado.

La información que muestra el display es la frecuencia que el algoritmo resolvió para la entrada de audio.

Reproducción de la nota

Otra etapa a desarrollar es la reproducción de la nota de entrada.

El objetivo es, no solo mostrar la frecuencia de la nota muestreada, sino poder reproducirla mediante un amplificador de sonido.

Esta implementación permite, por ejemplo, ir ajustando una cuerda de guitarra. Conociendo a la frecuencia que buscamos llegar, se puede escuchar cuan afinada estará la cuerda.

Para implementar esta etapa, con los resultados del algoritmo YIN, pasa por un DAC, pasa por un filtro recuperador, y por último, pasa por el amplificador de audio.

Fuentes y Referencias

http://audition.ens.fr/adc/pdf/2002_JASA_YIN.pdf

https://www.eecs.qmul.ac.uk/~simond/pub/2014/MauchDixon-PYIN-ICASSP2014.pdf