DDS con señal comprimida por medio de SVD y huffman coding para la integración en un FPGA

Emilio Tonix Gleason

24/Marzo/2021

Contents

1 Generación de la señal en python con compresion

Para la generación de la señal se uso un script de python atravez de la libreria numpy. Se asignaron 256 espacios para generar una cuarta parte, dado que la señal se puede reconstruir en 4 estapas con un solo conetido en memoria, mas adelante se mostrara el planteamento de esto. La señal por defecto viene ya normalizada de -1 a 1, sin embargo le hacemos un offset y compresion para que quede del rago [.5,1].
La idea de esto es dejar la parte negativa del seno con valores positivos, finalmente la señal entonces quedaria de [0,1]. Finalmente se multiplica ese valor normalizado por 255. La señal queda de [0,255] con el zero en 127.

```
import math
import numpy as np
import matplotlib.pylab as plt
from sympy import Li, Line

x = np.linspace(0,np.pi/2, 256)
sine = .5*(np.sin(x)+1)
sine = sine*255
```

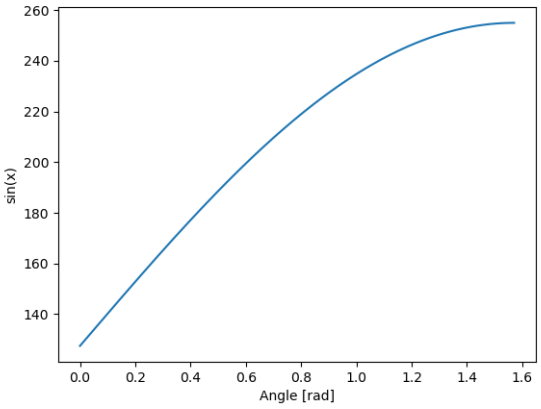

Figure 1: Cuarto de la señal zero en un rango [127,255]

1.1 Compresion de la señal para ahorro de memoria

Dado que la memoria es un recurso critico y se buscaran estrategias mas elegantes para su compresion y reconstrucción. Notese que para representar la señal estamos usando solo 7 bits [0,127]. Por lo cual el espacio total esta dado por $255 \times 7bits = 1785$. Para mejorar el espacio requerido usaremos dos preprocesamientos el primero es SVD y el segundo es compression de huffman.

La descomposición de valores singulares (SVD) consiste en obtener las caracteristicas mas signficativas de una matriz con eso logramos la compresion dado que podemos remover los valores que no aporten tanto a lo que conocemos como la forma de una senosodial, y tratar de dejar lo minimo necesario para obtener su forma. Para ejecutar la compresion SVD convertirmos la señal en un matriz cuyas dimensiones sean un par de 256. La matriz elegida fue de 64X4=256.

```
shaper = sine.reshape(64, 4)
# obtain svd
U, S, V = np.linalg.svd(shaper,full_matrices=False)
S = np.diag(S)

#principal componenet analysis
r = 1
low_rank = U[:, :r] @ S[0:r,:r] @ V[:r, :]
```

Posteriormente la matriz la regresamos a su estado vector fila y converitrmos los valores a enteros para que puedan ser usado en el FPGA.

```
sine = low_rank.flatten()
sine = np.floor(sine).astype(int)-128
```

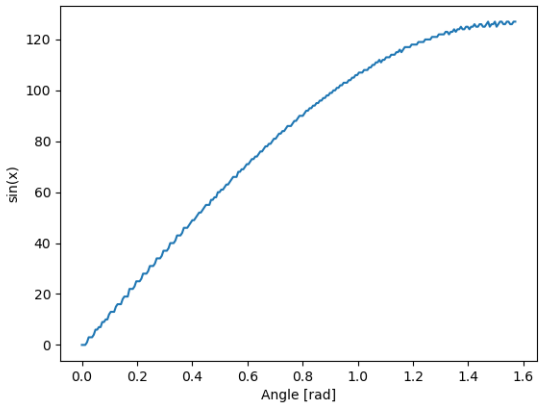

Figure 2: Señal con extración de caracteristicas SVD

La señal nos da una sensación aparante de tener mas risos sin embargo esto se debe a que se aplana mas y repite valores dado que se le removio su suavidad. Por esta razon vamos explotar la codificacion de huffman, la cual consiste en almacenar el valor junto con su número de repeticiones. En python es muy sencillo usamos la funcion unique que nos dice los valores que existen ya con el numero de repeticiones. Y dado que esta parte de la señal es monotonicamente creciente podemos usar la compresión sin problema.

```
(uniq, freq) = (np.unique(sine, return_counts=True))
print(np.column_stack((uniq,freq)))
print(len(uniq))
```

Value	0	1	3	4	6	7	9	10	12
Times	3	1	3	1	2	2	2	2	1

Table 1: Una muestra del resultado de compresión

Notese que la tabla omite valores como por ejemplo el 2,5,8,11, y esto es por que la SVD removio estas caracteristicas menos relevantes.

Finalmente el tamaño de la tabla nos queda de tamaño 109. Para los datos seguimos requieriendo 7 bits, sin embargo hay que agregar 4 bits de repticiones de los datos. Se aplica enmascaramiento para generar el dato y se guarda en un archivo de texto.

```
file = open('output.txt', 'w')
for n in range (0,len(uniq)):
    print(uniq[n]<<4|freq[n], file=file)
file.close()
```

1.2 Comparativa de ahorro

Espacio convencional

$$255 \times 7bits = 1785$$

Espacio con SVD y huffman

$$109 \times (7bits + 4(huffman)) = 1199$$

Compresión

$$compress = 1 - \frac{1199}{1785} = 32.82\%$$

2 Etapas de fase

La señal decomprimida consta de 256 valores por etapa y dado que se requieren 4 etapas para reconstruir el seno tenemos un total de 1024 puntos por cada periodo de la señal, estos puntos son la representación de la fase la cual denotamos con el siguiente simbolo.

$$\phi = \text{fase en el rango discreto}$$

Cada etapa tiene una lectura de un pedaso de la señal, y tiene un offset de 127. Los [] respresentan el indice de fase ϕ

1. Lectura de memoria Forward $\{0 \leq \phi \leq 255\}$
2. Lectura de memoria Backward $\{256 \leq \phi \leq 511\}$
3. Lectura de memoria Forward con polarización negativa $\{512 \leq \phi \leq 767\}$
4. Lectura de memoria Backward con polarización negativa $\{768 \leq \phi \leq 1023\}$

Dichos etapas estaran representadas por una maquina de estados.

2.1 Ecuaciones de fase

Se requiere hacer un mapeo de grados $0 \leq \theta \leq 360 \rightarrow \{0 \leq \phi \leq 1024\}$ a fase discreta. Para ello hacemos una simple división.

$$\alpha = \frac{1024}{360} = 2.844 \frac{val}{grad}$$

Calculamos el error de la siguiente forma redondeando$\alpha = 3$.

$$Err = \frac{1}{3-2.844} = 6.41$$

Si redondeamos α , el error incrementa por un valor cada 6.41 valores de θ . Asique cada 6 incrementos de θ restamos la sobresitmación. Y como factor de correción cada 90 grados sumamos uno. Con esto tenemos mas presición, en la ecuación inferior se considera que $\phi, \theta \in \mathbb{N}$ dado que la ecuación se encuentra en el FPGA sin valores flotantes, por esto mismo la ecuacion se factoriza.

$$\phi = \alpha\theta - \frac{\theta}{6} + \frac{\theta}{90}$$

$$\phi = 3 * \theta - \frac{\theta}{6} + \frac{\theta}{90}$$

Codigo de verilog en la parte de localización de fase.

```
1        // transform degree to discrete index form [0,360]->[0,1024]
2        phase_indx = 2'd3*phase-phase/3'd6+phase/7'd90;
```

2.2 Algoritmo de fase

1. Mandar una señal de habilitación [trigger] para habilitar la maquina de estados de la fase y colocar la dirección de lectura de memoria en un valor conocido.
2. Buscar a que rango de la etapa de reconstrucción pertenece el valor de fase.
   (a) Por ejemplo $\phi = 300$ pertenece a la segunda etapa.
3. Selecionar la maquina de estados en la etapa correspondiente.
4. Restar a ϕ un offset, segun la etapa en la que se inicie.
   (a) Si se elige la estapa 2 el offset sería 256 . $\{\phi - 256\}$
5. Restar $\phi = \phi - 1$ cada tiempo t_ϕ , este tiempo esta en sincronia con el proceso de lectura de memoria. Sin embargo la visibilidad de la señal esta deshabilitada.
6. Una vez sea $\phi = 0$ se habilita la visibilidad de la señal. La señal sera mostrada en el punto de fase requerido dado que el proceso de lectura de memoria ya estaba corriendo concurremente.

2.3 Frecuencia de la señal

La frecuencia de la señal seno esta da por el tiempo de lectura, lo cual implica que si leemos la memoria en un intervalo T_{div} tendremos una relación que nos dara la frecuencia de salida de la señal.
Un cuarto de la señal seno esta compuesta por 256 puntos. Sin embargo con la tabla de huffman se comprime a 108 espacios de memoria. El total de espacios en tiempo seran 256 tiempo de bit mas 108 lecturas de memoria.
Lecturas en memoria

$$256 + 108 = 364$$

Cada lectura se hace por un tiempo del prescaler

$$T_{sin} = 364 * T_{div}$$

Ahora lo rescribimos en terminos de frecuencia y lo multiplcamos por 4 partes.

$$T_{sin} = \frac{364}{f_{div}} * 4$$

$$f_{sin} = \frac{f_{div}}{364*4}$$

$$f_{sin} = \frac{f_{div}}{1456}$$

Generalemente tenemos como entrada la frecuencia deseada para que el modulo genera la señal. Entonces ajustamos la frequencia del divisior como una parametro configurable desde prescaler que nos permita hacer modicaciones de la frecuencia.

$$f_{div} = f_{sin} \times 1456$$

```
wire [31:0]freq_div = freq*32'd 1456;

Prescaler #(.DIV(`MHZ(5))) wave_clk
(    .src_clk(src_clk),
     .en(en_clk),
     .set_div(set_freq),
     .div(freq_div),
     .clk_div(wave_clk_en)
);
```

3 Implementación

El modelo de alto nivel tiene que poder recibir los datos del preprocesamiento para guardarlos en la memoria RAM y posteriormente elegir una fase, para comenzar la generación de la señal seno. Para la escritura de la memoria se encpasulo el modelo de RAM en el modelo final con el objetivo de hacer transparente el proceso.

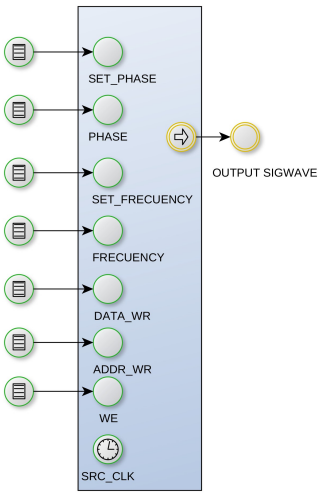

Figure 3: Diagrama de bloque DDS

Internamente cuenta con tres modulos. Prescaler configurable para el manejo de frecuencia, memoria RAM para almacenar datos y una maquina de estados para las etapas de lectura.

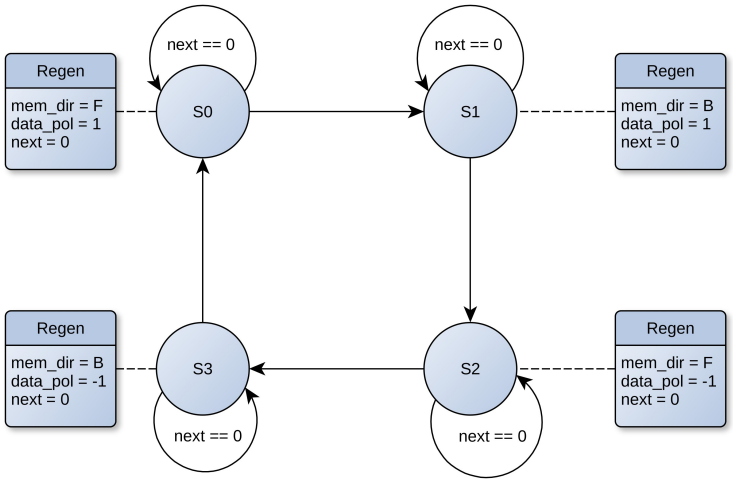

Figure 4: Maquinas de estados para lectura RAM

En la imagen debajo se puede apreciar el cambio de maquina de estados por parte de la señal

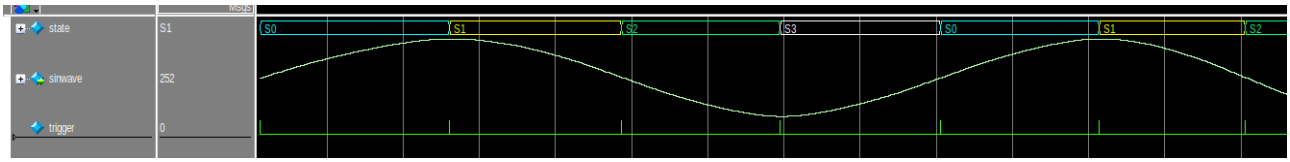

Figure 5: Cambio de maquina de estados por sección de la señal.

3.1 Escritura de la memoria

Mientras el proceso de escritura este habilitado por medio de "**WE**" estara deshabilitado la parte de lectura de memoria y de generación de la señal. Para simular la escritura de la memoria se usaran las APIS **fopen** y **fscan** durante el proceso de simulación. De esta forma se lee el archivo de texto generado en python.

```
    //write memory
    we = 1'b1;
    data_file = $fopen("../../PythonScript/output.txt", "r");
    if (data_file == `NULL)
    begin
        $display("data_file handle was NULL");
        $finish;
    end
    while(!$feof(data_file))
    begin
        scan_file = $fscanf(data_file, "%d\n", captured_data);
        $display("val = %d",captured_data);
        data_wr = captured_data;
        #4
        addr_wr = addr_wr+1;
    end
```

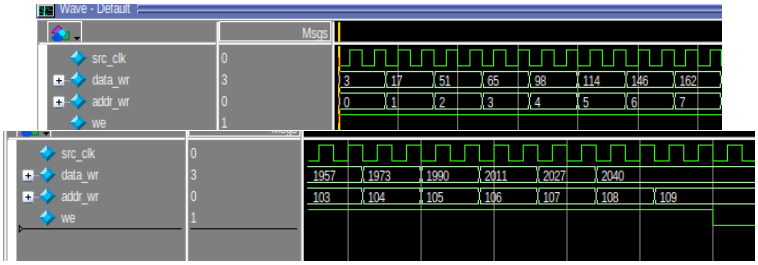

Figure 6: Lecturas de los valores

3.2 Setup de fase

Para la colacion de la fase se hace la conversion de grado θ a ϕ como se discutio en la sección 2.1, posteriormente se manda un valor donde se selecionara la posicion de la maquina de estados inicial, y finalmente se hace la lectura de la memoria hasta n puntos de fase. Cuando la fase ϕ llega a 0 se hace visible la señal empezando asi la señal en su punto de fase.

En la parte del test bench se puede colocar una fase con valor numerico. Notese que tiene que estar deshabilitada la escritura de memoria.

```
    // set phase
    #4
    we = 1'b0;
    set_phase = 1'b1;
    phase = 45;
    #2
    set_phase = 1'b0;
    #10
```

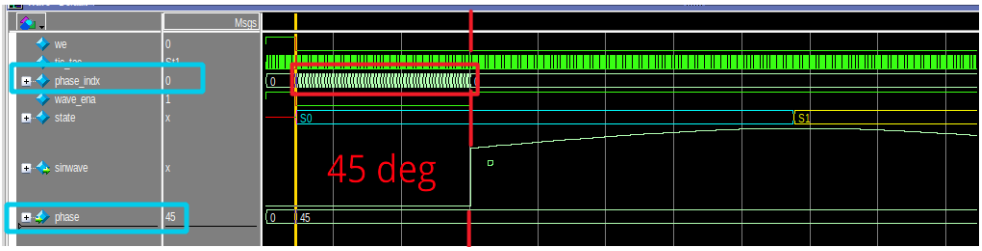

Figure 7: Fase de 45 gados

mem_addres : Indice en la memoria.

mem_dir : Forward{1},Backward{-1}

data_pol: Data polarizaqtion [1,-1]

huffman : Es el numero en el que se repite el valor logico de la señal. En hardware puede ser tomado con un delay con N ciclos muestra.

tic_tac: Es el tiempo de enable para la ejecución de la señal

A continuación se mostrata un diagrama visual de como funciona la logica del seteo de la fase, manejando tres hilos de tiempo como procesos concurrentes y los relojes como los estimulos a los que responde el cambio.

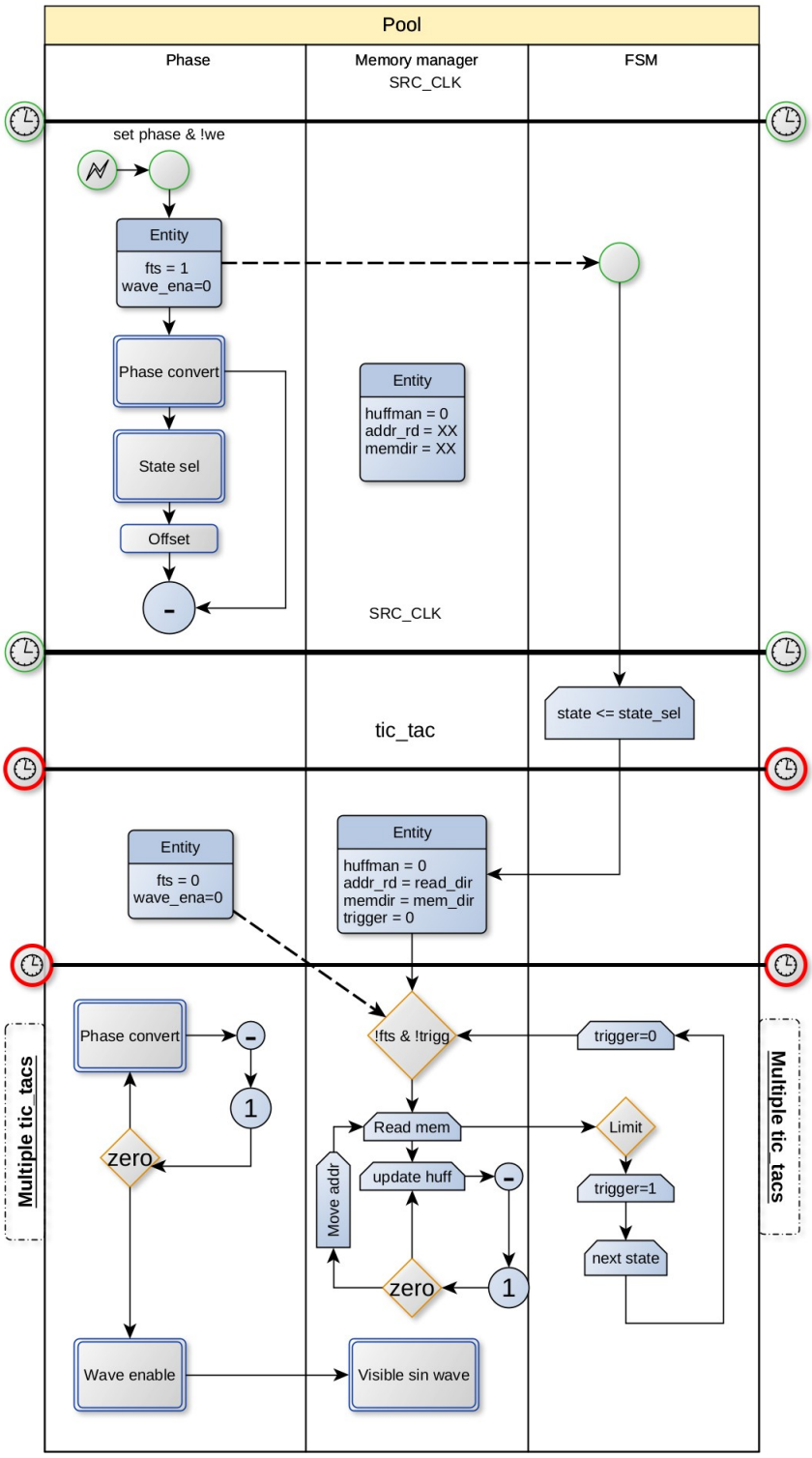

Figure 8: Diagrama de procesamiento de fase y lectura de memoria

3.3 Setup de frecuencia

Para la modulación en frecuencia se hizo un prescaler que pueda hacer el cambio en runtime, para mejor y evitar problemas de cuentas cuando se hace el cambio de frecuencia el contador interno del prescaler se hace 0 . Por defecto el prescaler tiene una frecuencia de 5 Mhz. El cambio de la frecuencia se hace atravez de la señal "**set_div**".

```
    if(set_div) begin
        MAX_CNT=`PRESCALE_CNT(SRC_CLK,div);
        cnt=0;
    end
```

El define nos deja convertir frecuencia esperada a cuentas. Se muestra aqui como se genera el define en config.v

```
`define MHZ(freq) freq*1000000
`define PRESCALE_CNT(ref_clk,freq) (ref_clk/freq)

//CLOCKS
`define SOURCE_CLK `MHZ(50)
`define Output_frequency `MHZ(5)
```

Para la prueba de modulación de frecuencia aplico la funcion chirp lineal.

$$sin(\phi(t))$$

Con un paso c y una frecuencia inicial f_0

$$\phi(t) = ct + f_0$$

En nuestro caso se hara $c = 100$, $f_0 = 500$ y como limite de $\phi(t)_{max} = 6500$.
Dado que el reloj esta a 50MHZ se hace la division del reloj con $\phi(t)$

$$delay = \frac{clk_{src}}{\phi(t)}$$

Aqui se presenta la representación codificada de dicho proceso de generación chirp.

```
    for (i=500; i < 6500; i=i+100)
    begin
        //set frecuency
        set_freq = 1;
        freq = i; //1khz
        #4;
        set_freq = 0;
        #(`MHZ(100)/i); // delay
    end
```

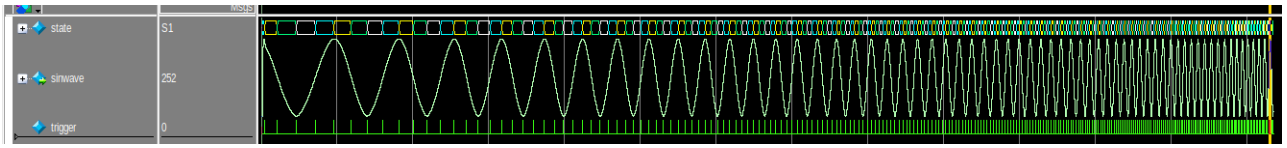

Figure 9: Chirp test bench