



# FACULTAD DE INGENIERIA

Universidad de Buenos Aires

Departamento de Electrónica

## 66.17 Sistemas Digitales

Trabajo Final: Diseño de un motor de rotación gráfico 3D basado en el algoritmo CORDIC

**Carballeda, Ignacio L. J.** 91.646 icarballeda@fi.uba.ar

5 de marzo de 2021

Cuatrimestre / Año	2.º cuatrimestre 2020
Profesor	Ing. Nicolás Álvarez

Fecha de entrega	Firma

Nota	Fecha de aprobación			Firma

Observaciones: \_\_\_\_\_

---

---

---

---

---

# Índice

<b>1. Objetivo</b>	<b>3</b>
<b>2. Bloques Principales</b>	<b>3</b>
2.1. UART . . . . .	3
2.2. RAM . . . . .	4
2.3. Display de 7 segmentos y LEDS . . . . .	5
2.4. Maquina de estados principal . . . . .	6
2.4.1. Estado: Init . . . . .	7
2.4.2. Estado: Waiting for UART . . . . .	7
2.4.3. Estado: Reading from UART . . . . .	7
2.4.4. Estado: Write SRAM . . . . .	7
2.4.5. Estado: UART end data reception . . . . .	7
2.4.6. Estado: Read from SRAM . . . . .	7
2.4.7. Estado: Process Coords . . . . .	7
2.4.8. Estado: Print Coords . . . . .	7
2.4.9. Estado: Reset Device, Clean SRAM y Clean VRAM . . . . .	8
2.5. Procesador de Entradas . . . . .	8
2.6. Rotador . . . . .	8
2.7. Video RAM . . . . .	9
2.8. Driver de Video . . . . .	9
2.9. Controlador VGA . . . . .	10
2.10. Síntesis . . . . .	11
<b>3. Fotos del proyecto funcionando</b>	<b>13</b>
<b>4. Códigos fuente</b>	<b>14</b>
4.1. Código Python Enviador de Coordenadas . . . . .	14
4.2. top.vhd . . . . .	15
4.3. input_processor.vhd . . . . .	27
4.4. video_driver.vhd . . . . .	32
4.5. vram.vhd . . . . .	34
4.6. cordic.vhd . . . . .	35
4.7. cordic_stage.vhd . . . . .	38
4.8. rotator.vhd . . . . .	39

## 1. Objetivo

En el presente Trabajo Práctico el alumno desarrollará una arquitectura de rotación de objetos 3D basada en el algoritmo CORDIC. El objetivo principal es desarrollar tanto la unidad aritmética de cálculo como así también el controlador de video asociado. Para la realización completa del Trabajo Práctico se cargarán en memoria externa las coordenadas correspondientes a un objeto tridimensional predefinido mediante una interfaz serie UART. A partir de los valores de las componentes, se rotará el objeto alrededor de cada uno de los ejes de coordenadas según el valor que adquieran las entradas del sistema, y por último las componentes rotadas serán presentadas en un monitor VGA mediante la aplicación de una proyección plana. En la Figura (1) puede observarse un digrama en bloques del sistema completo. Deberá determinarse la cantidad mínima de bits de ancho de palabra (bits de precisión) para alcanzar las especificaciones requeridas

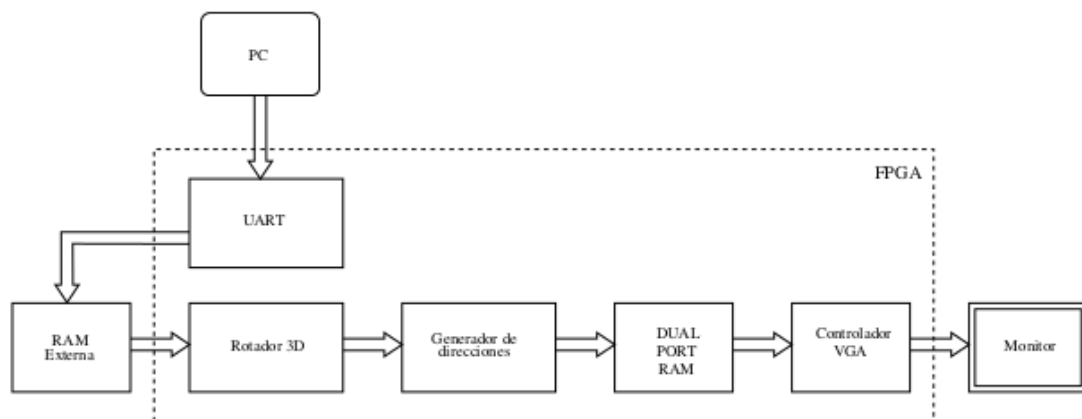


Figura 1: Esquema completo del sistema.

## 2. Bloques Principales

En la figura 1 se pueden apreciar los bloques principales que integran el sistema. Como se puede apreciar se utilizó, a diferencia del diagrama propuesto, una RAM interna.

### 2.1. UART

Es básicamente un driver que permite capturar las señales del puerto serie y transformarlas, con la ayuda de un submodulo de timing en bytes ( $D_{in}$  8 bits). El baudate fué seteado en 1200 bps. Este modulo se utiliza solamente para cargar las coordenadas a memoria RAM al comienzo del programa.

Del lado de la PC se escribió un programa en lenguaje Python muy simple que envía una a una las coordenadas (complemento a2). Por limitaciones de la memoria RAM tanto el numero de coordenadas como a resolución de las mismas fueron reducidas.

El archivo con las coordenadas facilitado por la cátedra contiene 11946 lineas, en cada linea hay tres coordenadas. Cada coordenada es un float con 10 dígitos después de la coma.

Como tenemos una memoria RAM limitada de tan solo 32 KBytes, se decidió descartar 3 de cada cuatro lineas y que cada coordenada tenga solo 8 bits de precisión (0 a 255).

A pesar de estos recortes, se logró una visualización bastante representativa de el archivo en cuestión.

Instancia del componente UART hecha en el top level:

```
component uart is
  generic(
    F: natural;
    min_baud: natural;
    num_data_bits: natural
  );
  port (
    Rx : in std_logic;
    Tx : out std_logic;
    Din : in std_logic_vector(7 downto 0);
    StartTx : in std_logic;
    TxBusy : out std_logic;
    Dout : out std_logic_vector(7 downto 0);
    RxRdy : out std_logic;
    RxErr : out std_logic;
    Divisor : in std_logic_vector;
    clk : in std_logic;
    rst : in std_logic
  );
end component;
```

## 2.2. RAM

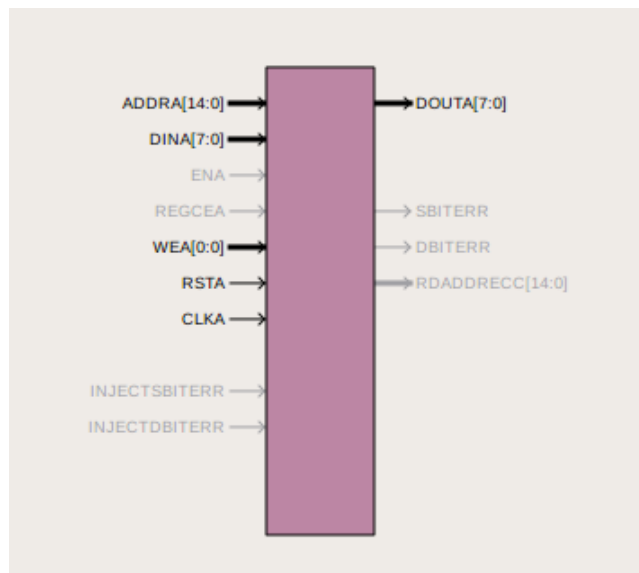
Al no tener la posibilidad de conectar una RAM externa, se procedió a sintetizar una en la misma FPGA.

Se utilizó el generador *IPCore* incluido en el software de *Xilinx ISE 14.7* para sintetizar tantos bloques como sea posible de memoria RAM no distribuida.

En la FPGA es posible sintetizar 16 modulos BRAM, los cuales son de 18 Kbits cada uno. El espacio total disponible es de 36 KBytes. Es decir, hay espacio para suficiente para almacenar 32768 datos de 8 bits c/u.

A continuación se pueden observar las opciones elegidas para la generación de la RAM.

Como se puede observar en la siguiente imagen, la RAM generada necesita de 15 líneas para Address, 8 para datos de entrada (8 bits) y 8 para salida. A su vez posee una entrada para elegir si queremos escribir o no, clock y reset. El enable fué seteado para estar siempre activo.



Instanciación de la memoria RAM en el top level:

```
component SRAM is
  port(
    clka      : in std_logic;
    rsta      : in std_logic;
    wea       : in std_logic_vector(0 downto 0);
    addra     : in std_logic_vector(RAM_ADDRESS_WIDTH-1 downto 0);
    dina      : in std_logic_vector(RAM_DATA_WIDTH-1 downto 0);
    douta     : out std_logic_vector(RAM_DATA_WIDTH-1 downto 0)
  );
end component;
```

## 2.3. Display de 7 segmentos y LEDS

Se utilizó un modulo que permitía representar números de hasta 8 bits (0 a 256) que fue muy útil a la hora de resolver problemas, como por ejemplo imprimir la cantidad de datos

cargados, o direcciones de memoria que estaban siendo accedidas.

También, se utilizaron 5 LEDS para indicar el estado actual de la maquina de estados principal.

Instanciación del display de 7 segmentos en el top level:

```
component LED8 is
  port (
    -- Reset & Clock Signal
    rst:          in std_logic;
    clk:          in std_logic;    -- 50 MHz
    -- LED8 PIN
    led_out:      out std_logic_vector(7 downto 0);  -- LED Segment
    digit_select: out std_logic_vector(3 downto 0);  -- LED Digit
    show_number:  in  std_logic_vector(7 downto 0)
  );
end component;
```

## 2.4. Maquina de estados principal

Valiéndose de dos estados, actual y próximo, esta maquina de estados ubicada va coordinando todas las acciones necesarias.

Los estados posibles son los siguientes

1. Init
2. Waiting for uart
3. Reading from uart
4. Write SRAM
5. Uart end data reception
6. Idle
7. Read from SRAM
8. Process Coords
9. Print Coords
10. Reset Device
11. Clean SRAM
12. Clean VRAM

A continuación se provee una breve descripción de la función de cada estado.

### 2.4.1. Estado: Init

Se ejecuta luego de un reset, lo unico que hace este estado es inicializar señales en cero, como cantidad de bytes recibidos y el address de la SRAM en 0x00. A su vez espera un tiempo antes de pasar al siguiente estado (esperar UART).

### 2.4.2. Estado: Waiting for UART

Simplemente espera a que el componente UART ponga su señal de *rx\_ready* en alto. Cuando detecta que se recibieron todos los bytes, pasa al estado de finalización de recepción.

### 2.4.3. Estado: Reading from UART

Asigna el dato leído a *sig\_sram\_data\_in\_next*, aumenta el contador de bytes leídos en uno y pasa al estado de escritura en SRAM. Luego vuelve al estado *Waiting for UART*

### 2.4.4. Estado: Write SRAM

Habilita la señal de escritura, para poder guardar el dato en la SRAM, aumenta en uno la dirección de memoria SRAM para la próxima iteración.

### 2.4.5. Estado: UART end data reception

Una vez que se recibieron todos los bytes necesarios y estos fueron escritos en memoria RAM, nos preparamos para comenzar a leer la RAM desde la dirección 0x00 y pasamos al estado de lectura.

### 2.4.6. Estado: Read from SRAM

Se vale de la señal *xyz\_selector\_next* para decidir si el byte que esta leyendo de RAM corresponde a una coordenada X, Y, o Z. Este estado se visitara hasta que se hayan leído todas las coordenadas.

Cada vez que se almacene una tupla de coordenadas X, Y, Z se procederá al estado *process\_coords* en donde se procesaran (rotaran) las coordenadas.

### 2.4.7. Estado: Process Coords

Cuando se llega a este estado una tupla de coordenadas X,Y,Z están cargadas en las señales *X,Y,Z\_current*. Lo que hace este estado es darle tiempo al cordic para rotarlas. En el proceso mantiene la señal de escritura de SRAM en bajo hasta que consume el tiempo definido por *CYCLES\_TO\_WAIT\_TO\_CORDIC\_TO\_FINISH*. Luego de esto pasa al estado *print\_coords*

### 2.4.8. Estado: Print Coords

Se habilita la escritura en la VRAM y se genera la dirección de memoria a escribir como se muestra a continuación

```
sig_vram_addr_wr_next <= z_coord_rotated_unsigned(7 downto 0) &  
↪ y_coord_rotated_unsigned(7 downto 0);
```

Como se puede ver, lo que se hace básicamente es una proyección sobre el ".<sup>o</sup>e X" de las coordenadas ya rotadas. Los 8 bits mas significativos corresponden a la coordenada Z y los restantes a la coordenada Y.

Con esta dirección generada ya alcanza para que el dato se escriba en el próximo ciclo y que inmediatamente se muestre en pantalla.

#### 2.4.9. Estado: Reset Device, Clean SRAM y Clean VRAM

Estos estados sirven para resetear el equipo a su estado inicial, limpian todas las señales a sus valores por defecto y limpian memorias.

### 2.5. Procesador de Entradas

Este modulo es el encargado de procesar los estímulos que el usuario crea utilizando una botonera. Permite que el usuario rote el mundo, cambie el paso de rotación y que resetee el sistema.

Instanciación del procesador de entradas en el top level:

```
component input_processor is
  generic (
    ANGLE_WIDTH          : integer := 10;
    ANGLE_STEP_INITIAL   : integer := 1
  );
  port (
    clk: in std_logic;
    matrix_buttons_col : in std_logic_vector (3 downto 0);
    matrix_buttons_row : in std_logic_vector (3 downto 0);
    angle_x : out signed (ANGLE_WIDTH-1 downto 0);
    angle_y : out signed (ANGLE_WIDTH-1 downto 0);
    angle_z : out signed (ANGLE_WIDTH-1 downto 0);
    angle_step: out natural;
    reset_button: out std_logic
  );
end component;
```

### 2.6. Rotador

Este componente es el encargado de realizar la rotación de las coordenadas. Como entrada recibe una tupla de coordenadas y el ángulo de rotación. En la salida, disponibiliza las coordenadas ya rotadas.

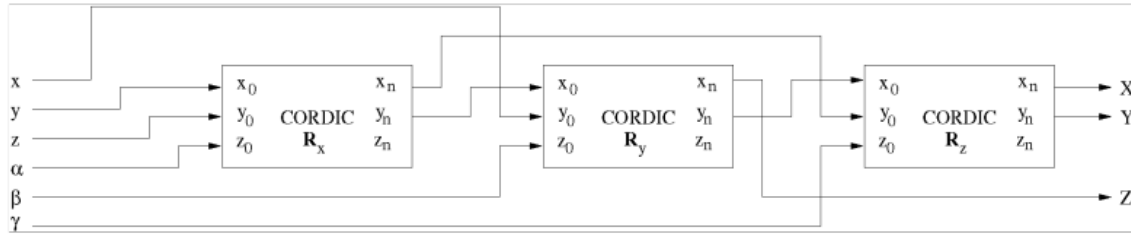
Instanciación del Rotador en el top level:

```
component rotator is
  generic (
    COORDS_WIDTH          : integer := CORDIC_WIDTH;
    ANGLES_INTEGER_WIDTH  : integer := ANGLE_WIDTH;
    STAGES                 : integer := CORDIC_STAGES
  );
  port (
    clk          : in std_logic;
    X0, Y0, Z0   : in signed(CORDIC_WIDTH-1 downto 0);
    angle_X, angle_Y, angle_Z : in signed(ANGLES_INTEGER_WIDTH-1 downto 0);
    X, Y, Z      : out signed(CORDIC_WIDTH-1 downto 0)
```



```
);
end component;
```

Dentro de el componente *rotator* reside el **CORDIC** el cual es instanciado tres veces.



Como puede apreciarse en la imagen anterior, se utilizaron tres CORDICS (Que rotan en 2D) en cascada para realizar la rotación 3D.

## 2.7. Video RAM

Memoria de tipo Dual Port RAM, que posibilita la escritura/lectura en modo simultaneo. Esto es necesario ya que básicamente estamos escribiendo las coordenadas al mismo tiempo que las leemos para graficarlas en pantalla.

Instanciación de la dual port RAM en el top level:

```
component dpram is
  generic (
    DPRAM_BITS_WIDTH : natural := 1; -- Ancho de palabra de la memoria medido en bits
    DPRAM_ADDR_BITS : natural := 16 -- Cantidad de bits de address (tamaño de la memoria)
  );
  port (
    rst      : in std_logic;
    clk      : in std_logic;
    data_wr  : in std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0);
    addr_wr  : in std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
    ena_wr   : in std_logic;
    addr_rd  : in std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
    data_rd  : out std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0)
  );
end component;
```

## 2.8. Driver de Video

Es el modulo encargado de, dado un cierto pixel (input), consultar a VRAM si decide si debe o encenderse o no (RGB output).

Como se puede ver, el componente tiene como salida una dirección de 16 bits, esto es así porque necesita consultar a las diferentes direcciones de la memoria VRAM.

Instanciación del driver de video en el top level:

```
component video_driver is
  generic (
```

```

DPRAM_BITS_WIDTH : natural := 1; -- Ancho de palabra de la memoria medido en b
DPRAM_ADDR_BITS : natural := 16 -- Cantidad de bits de address (tamaño de la me
);
port (
rst: in std_logic;
clk: in std_logic;
red_en_o: out std_logic;
green_en_o: out std_logic;
blue_en_o: out std_logic;
pixel_x: in unsigned(9 downto 0);
pixel_y: in unsigned(9 downto 0);
addr_rd : out std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
data_rd : in std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0)
);
end component;

```

## 2.9. Controlador VGA

Encargado de recorrer toda la pantalla e ir graficando los pixeles según corresponda. Trabaja en conjunto con el *Driver de Video* ya que le consulta a este ultimo si el pixel en el que se encuentra debe encenderse o no.

Instanciación del controlador VGA en el top level:


```

entity vga_ctrl is
  port (
    mclk: in std_logic;
    red_i: in std_logic;
    grn_i: in std_logic;
    blu_i: in std_logic;
    hs: out std_logic;
    vs: out std_logic;
    red_o: out std_logic_vector(2 downto 0);
    grn_o: out std_logic_vector(2 downto 0);
    blu_o: out std_logic_vector(1 downto 0);
    pixel_row: out std_logic_vector(9 downto 0);
    pixel_col: out std_logic_vector(9 downto 0)
  );
end vga_ctrl;

```

## 2.10. Síntesis

A continuación se pueden apreciar los resultados que arroja *Xilinx* al completar la síntesis del hardware. Cabe destacar que se usó el 100 % de los bloques disponibles de RAM.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	1493	4656	32%
Number of Slice Flip Flops	749	9312	8%
Number of 4 input LUTs	2809	9312	30%
Number of bonded IOBs	37	158	23%
Number of BRAMs	20	20	100%
Number of MULT18X18SIOs	6	20	30%
Number of GCLKs	1	24	4%

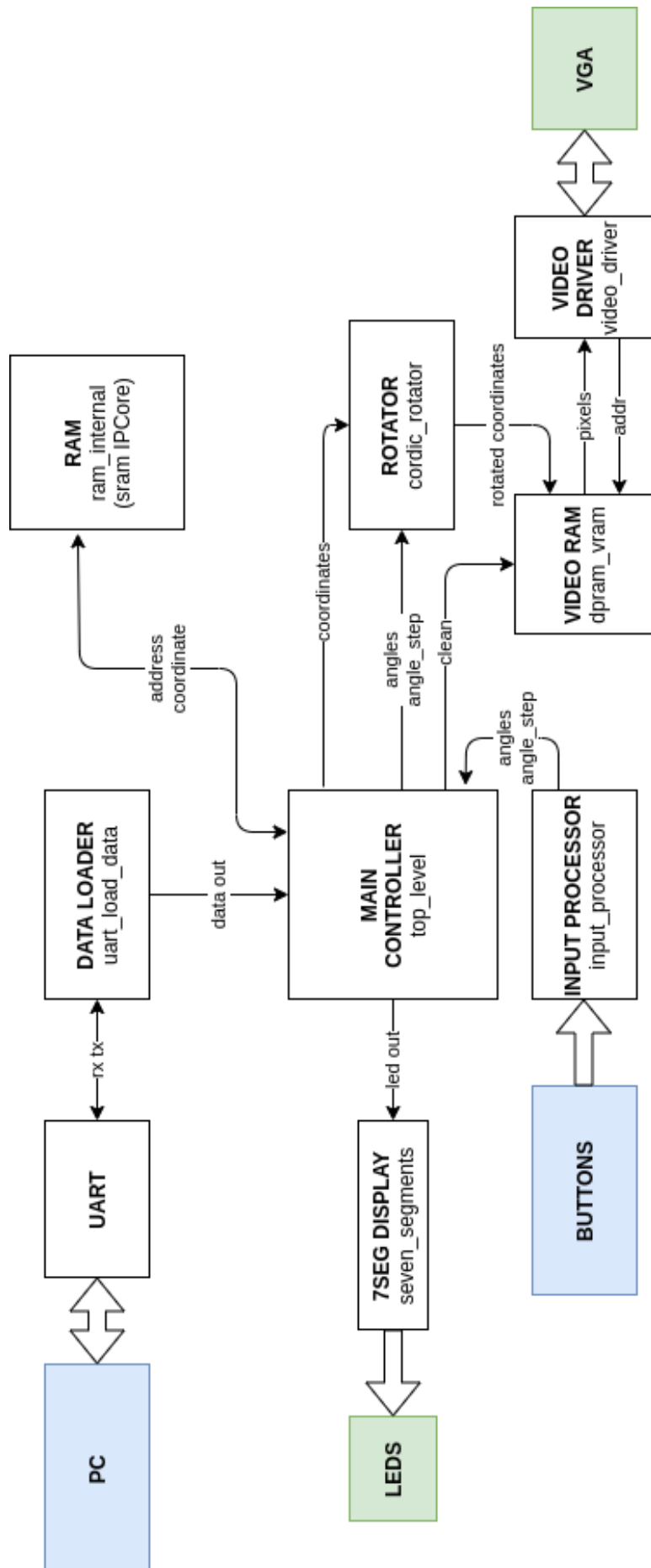


Figura 1: Diagrama de bloques principal

### 3. Fotos del proyecto funcionando



Figura 2: Kit de desarrollo utilizado

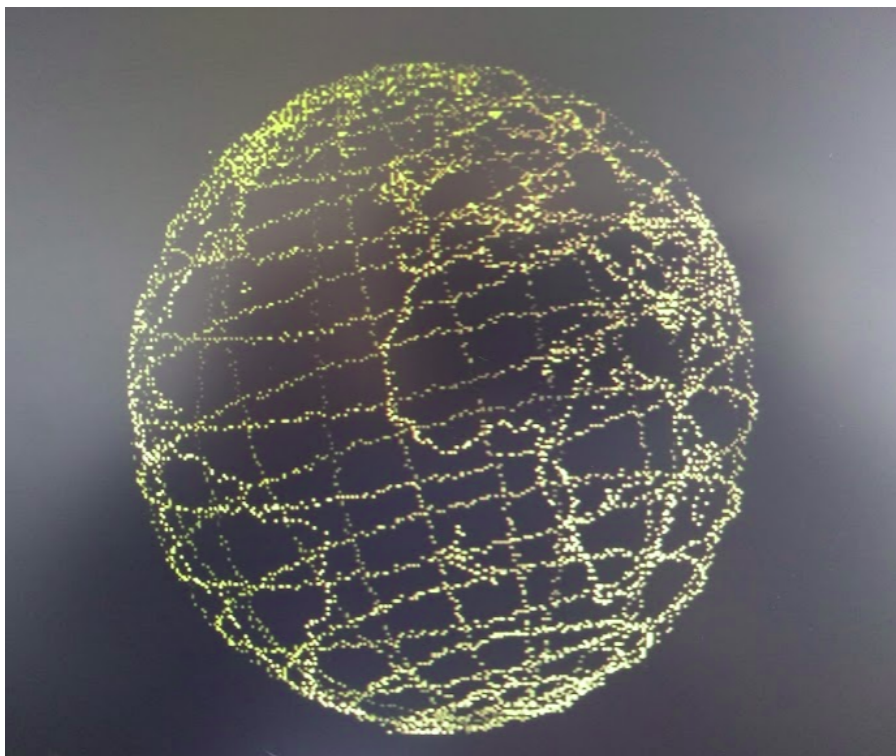


Figura 3: Mundo girando

## 4. Códigos fuente

### 4.1. Código Python Enviador de Coordenadas

```
1  #!/usr/bin/python3
2
3  import serial
4
5  COORDS_WIDTH = 8
6  LINES_TOTAL = 10923
7
8  with open('coordenadas.txt') as fp, serial.Serial('/dev/ttyUSB0', 115200, timeout=1) as s:
9      for line_number, line in enumerate(fp):
10         if line_number < LINES_TOTAL:
11
12             x = int(float(line.split('\t')[0])*2**(COORDS_WIDTH-1))
13             x_signed = True if x < 0 else False
14             y = int(float(line.split('\t')[1])*2**(COORDS_WIDTH-1))
15             y_signed = True if y < 0 else False
16             z = -1*int(float(line.split('\t')[2])*2**(COORDS_WIDTH-1))
17             z_signed = True if z < 0 else False
18
19             serial.write((x).to_bytes(1, 'big', signed=x_signed))
20             serial.write((y).to_bytes(1, 'big', signed=y_signed))
21             serial.write((z).to_bytes(1, 'big', signed=z_signed))
22
23             # imprimo coordenadas por pantalla a medida que las voy enviando, junto con
24             print(f"Line {line_number}, RAM Address: {line_number*3}: X={hex(x)} Y={hex(y)} Z={hex(z)}")
```

## 4.2. top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

entity top_level is
    generic (
        -- RAM Single Port (IP Core Generated)
        constant RAM_DATA_WIDTH           : integer := 8;
        constant RAM_ADDRESS_WIDTH        : integer := 15; --32 kBytes de RAM
        constant CYCLES_TO_WAIT           : integer := 4000;
        constant UART_BYTES_TO_RECEIVE    : natural := 32768;
        constant UART_COORDS_WIDTH        : natural := 8;
        -- CORDIC Constants
        constant COORDS_WIDTH: integer := 8;
        constant ANGLE_WIDTH: integer := 10;
        constant CORDIC_STAGES: integer := 8;
        constant CORDIC_WIDTH: integer := 12;
        constant CORDIC_OFFSET: integer := 4;
        constant ANGLE_STEP_INITIAL: natural := 1;
        constant CYCLES_TO_WAIT_TO_CORDIC_TO_FINISH: natural := 10;
        -- Dual Port RAM
        constant DPRAM_ADDR_BITS: natural := 16; -- 8 KBytes
        constant DPRAM_DATA_BITS_WIDTH: natural := 1
    );
    port (
        -- VGA signals (16I/OS2 : VGA Module)
        clk_tl : in std_logic;
        hs_tl, vs_tl : out std_logic;
        red_out_tl : out std_logic_vector(2 downto 0);
        grn_out_tl : out std_logic_vector(2 downto 0);
        blu_out_tl : out std_logic_vector(1 downto 0);
        -- UART pins (8I/OS1 : 2 pins for tx/rx)
        rx_tl : in std_logic;
        tx_tl : out std_logic;
        -- LEDs (16I/OS1 : 7-SEG-x4 board & Core Leds)
        led_tl: out std_logic_vector(3 downto 0);
        led_out_tl: out std_logic_vector(7 downto 0); -- LED Segment
        led_select_tl: out std_logic_vector(3 downto 0); -- LED Digit
        -- BUTTONS (8I/OS2 : 4x4 button Matrix)
        matrix_btn_col_tl: in std_logic_vector(3 downto 0); --ATT! Buttons
        -- with a PULLUP resistor
        matrix_btn_row_tl: in std_logic_vector(3 downto 0)
    );
end entity;

architecture top_level_arq of top_level is

    -- Prototipos a utilizar
    component vga_ctrl is
```

```

port(
    mclk, red_i, grn_i, blu_i      : in std_logic;
    hs, vs                        : out std_logic;
    red_o                         : out std_logic_vector(2 downto 0);
    grn_o                         : out std_logic_vector(2 downto 0);
    blu_o                         : out std_logic_vector(1 downto 0);
    pixel_row, pixel_col: out std_logic_vector(9 downto 0)
);
end component;

component SRAM is
    port(
        clka          : in std_logic;
        rsta          : in std_logic;
        wea           : in std_logic_vector(0 downto 0);
        addra         : in std_logic_vector(RAM_ADDRESS_WIDTH-1 downto
        ↪ 0);
        dina          : in std_logic_vector(RAM_DATA_WIDTH-1 downto 0);
        douta         : out std_logic_vector(RAM_DATA_WIDTH-1 downto 0)
    );
end component;

component uart is
    generic(
        F: natural;
        min_baud: natural;
        num_data_bits: natural
    );
    port (
        Rx          : in std_logic;
        Tx          : out std_logic;
        Din         : in std_logic_vector(7 downto 0);
        StartTx     : in std_logic;
        TxBusy      : out std_logic;
        Dout        : out std_logic_vector(7 downto 0);
        RxRdy       : out std_logic;
        RxErr       : out std_logic;
        Divisor     : in std_logic_vector;
        clk         : in std_logic;
        rst         : in std_logic
    );
end component;

component dpram is
    generic (
        DPRAM_BITS_WIDTH : natural := 1; -- Ancho de palabra de la memoria
        ↪ medido en bits
        DPRAM_ADDR_BITS : natural := 16 -- Cantidad de bits de address (tamao
        ↪ de la memoria es 2^ADDRS_BITS)
    );
    port (

```



```

    rst                : in std_logic;
    clk                : in std_logic;
    data_wr : in std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0);
    addr_wr : in std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
    ena_wr      : in std_logic;
    addr_rd : in std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
    data_rd : out std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0)
);
end component;

component video_driver is
    generic (
        DPRAM_BITS_WIDTH : natural := 1; -- Ancho de palabra de la memoria
        ↪ medido en bits
        DPRAM_ADDR_BITS : natural := 16 -- Cantidad de bits de address (tamaño
        ↪ de la memoria es 2^ADDRS_BITS)
    );
    port (
        rst: in std_logic;
        clk: in std_logic;
        red_en_o: out std_logic;
        green_en_o: out std_logic;
        blue_en_o: out std_logic;
        pixel_x: in unsigned(9 downto 0);
        pixel_y: in unsigned(9 downto 0);
        addr_rd : out std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
        data_rd : in std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0)
    );
end component;

component LED8 is
    port (
        -- Reset & Clock Signal
        rst: in std_logic;
        clk: in std_logic; -- 50 MHz
        -- LED8 PIN
        led_out: out std_logic_vector(7 downto 0); -- LED
        ↪ Segment
        digit_select: out std_logic_vector(3 downto
        ↪ 0); -- LED Digit
        show_number: in std_logic_vector(7 downto 0)
    );
end component;

component rotator is
    generic (
        COORDS_WIDTH : integer := CORDIC_WIDTH;
        ANGLES_INTEGER_WIDTH : integer := ANGLE_WIDTH;
        STAGES : integer := CORDIC_STAGES
    );
    port (

```

```

        clk                :    in std_logic;
        X0, Y0, Z0          :    in signed(CORDIC_WIDTH-1 downto 0);
        angle_X, angle_Y, angle_Z :    in signed(ANGLES_INTEGER_WIDTH-1
        ↪    downto 0);
        X, Y, Z             :    out signed(CORDIC_WIDTH-1 downto 0)
    );
end component;

component input_processor is
    generic (
        ANGLE_WIDTH          : integer := 10;
        ANGLE_STEP_INITIAL   : integer := 1
    );
    port (
        clk: in std_logic;
        matrix_buttons_col : in std_logic_vector (3 downto 0);
        matrix_buttons_row : in std_logic_vector (3 downto 0);
        angle_x : out signed (ANGLE_WIDTH-1 downto 0);
        angle_y : out signed (ANGLE_WIDTH-1 downto 0);
        angle_z : out signed (ANGLE_WIDTH-1 downto 0);
        angle_step: out natural;
        reset_button: out std_logic
    );
end component;

signal rst_tl: std_logic := '1';

-- Seales auxiliares para pasar interconexion
signal sig_aux_pixel_x_i: std_logic_vector(9 downto 0) := (others => '0');
signal sig_aux_pixel_y_i: std_logic_vector(9 downto 0) := (others => '0');
signal pixel_x, pixel_y: unsigned(9 downto 0);

-- Utilizada para entender si estamos leyendo un valor de X, Y o Z (de RAM)
signal xyz_selector_current, xyz_selector_next: natural := 0;

-- VGA
signal sig_blue_enable: std_logic := '0';
signal sig_red_enable: std_logic := '0';
signal sig_green_enable: std_logic := '0';

-- Coordenadas
signal x_coord_current, x_coord_next: std_logic_vector(COORDS_WIDTH-1 downto
    ↪ 0) := (others => '0');
signal y_coord_current, y_coord_next: std_logic_vector(COORDS_WIDTH-1 downto
    ↪ 0) := (others => '0');
signal z_coord_current, z_coord_next: std_logic_vector(COORDS_WIDTH-1 downto
    ↪ 0) := (others => '0');

-- Entradas del rotador
signal X0, Y0, Z0: signed(CORDIC_WIDTH-1 downto 0);
-- Coordenadas rotadas

```

```

signal X_coord_rotated: signed(CORDIC_WIDTH-1 downto 0);
signal Y_coord_rotated: signed(CORDIC_WIDTH-1 downto 0);
signal Z_coord_rotated: signed(CORDIC_WIDTH-1 downto 0);
-- Coord rotadas no signaladas
signal X_coord_rotated_unsigned: std_logic_vector(COORDS_WIDTH-1 downto 0) :=
  ⇨ (others => '0');
signal Y_coord_rotated_unsigned: std_logic_vector(COORDS_WIDTH-1 downto 0) :=
  ⇨ (others => '0');
signal Z_coord_rotated_unsigned: std_logic_vector(COORDS_WIDTH-1 downto 0) :=
  ⇨ (others => '0');

-- ngulos (van al cordic)
signal angle_x_t1: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_y_t1: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_z_t1: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
-- Angulos vienen del input processor
signal angle_x_from_input_proc: signed(ANGLE_WIDTH-1 downto 0) := (others =>
  ⇨ '0');
signal angle_y_from_input_proc: signed(ANGLE_WIDTH-1 downto 0) := (others =>
  ⇨ '0');
signal angle_z_from_input_proc: signed(ANGLE_WIDTH-1 downto 0) := (others =>
  ⇨ '0');
-- step
signal angle_step_t1: natural := ANGLE_STEP_INITIAL;

-- ngulos (entre 0 y 360)
signal angle360_x_current, angle360_x_next: unsigned(ANGLE_WIDTH-1 downto 0)
  ⇨ := (others => '0');
signal angle360_y_current, angle360_y_next: unsigned(ANGLE_WIDTH-1 downto 0)
  ⇨ := (others => '0');
signal angle360_z_current, angle360_z_next: unsigned(ANGLE_WIDTH-1 downto 0)
  ⇨ := (others => '0');

-- LEDS
signal sig_led_aux: std_logic_vector(3 downto 0) := (others => '0');
signal sig_binary_to_bcd: std_logic_vector(7 downto 0) := (others => '0');

-- UART
constant Divisor : std_logic_vector := "000000011011";
  ⇨ -- Divisor=27 para 115200 baudios
signal sig_uart_rx_ready : std_logic;
signal sig_uart_readed_data : std_logic_vector(7 downto 0);

-- Single Port RAM
signal sig_sram_address : std_logic_vector(RAM_ADDRESS_WIDTH-1
  ⇨ downto 0) := (others => '0');
signal sig_sram_data_out : std_logic_vector(RAM_DATA_WIDTH-1 downto 0);

-- Dual Port RAM (para video)
signal sig_vram_addr_rd : std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);

```

```

signal sig_vram_data_rd          : std_logic_vector(DPRAM_DATA_BITS_WIDTH-1
↳ downto 0);

-- Maquina de Estados
type state_t is (state_init, state_waiting_for_uart, state_reading_from_uart,
state_write_sram, state_uart_end_data_reception, state_idle,
↳ state_read_from_sram,
state_clean_vram, state_clean_sram, state_process_coords, state_print_coords,
state_reset_device, state_clean_vram_on_first_boot,
↳ state_read_from_sram_prev);
signal state_current, state_next : state_t := state_init;
signal sig_sram_address_current, sig_sram_address_next: natural := 0;
signal sig_sram_rw_current, sig_sram_rw_next: std_logic_vector(0 downto 0) :=
↳ "0";
signal sig_sram_data_in_current, sig_sram_data_in_next:
↳ std_logic_vector(RAM_DATA_WIDTH-1 downto 0) := (others => '0');
signal sig_uart_bytes_received_current, sig_uart_bytes_received_next: natural
↳ := 0;
signal sig_vram_addr_wr_current, sig_vram_addr_wr_next :
↳ std_logic_vector(DPRAM_ADDR_BITS-1 downto 0) := (others => '0');
signal sig_vram_ena_wr_current, sig_vram_ena_wr_next: std_logic := '0';
signal sig_vram_data_wr_current, sig_vram_data_wr_next:
↳ std_logic_vector(DPRAM_DATA_BITS_WIDTH-1 downto 0) := "0";
signal sig_vram_addr_wr_pointer_current, sig_vram_addr_wr_pointer_next :
↳ integer range 0 to 2**DPRAM_ADDR_BITS-1;
signal cycles_current, cycles_next: natural := CYCLES_TO_WAIT;

-- Arquitectura

begin
-- Actualizacin de registros
process(clk_tl, rst_tl) -- Agregar RESET
begin
if(rst_tl='0') then --rst_tl pin got a pullup
state_current <= state_reset_device;
elsif (clk_tl'event and clk_tl='1') then
state_current <= state_next;
sig_sram_address_current <= sig_sram_address_next;
sig_sram_rw_current <= sig_sram_rw_next;
sig_sram_data_in_current <= sig_sram_data_in_next;
cycles_current <= cycles_next;
sig_uart_bytes_received_current <= sig_uart_bytes_received_next;
x_coord_current <= x_coord_next;
y_coord_current <= y_coord_next;
z_coord_current <= z_coord_next;
xyz_selector_current <= xyz_selector_next;
sig_vram_addr_wr_current <= sig_vram_addr_wr_next;
sig_vram_ena_wr_current <= sig_vram_ena_wr_next;
sig_vram_data_wr_current <= sig_vram_data_wr_next;
sig_vram_addr_wr_pointer_current <= sig_vram_addr_wr_pointer_next;
end if;

```

```

end process;

-- Logica del estado siguiente
process(sig_uart_readed_data, sig_sram_address_current, state_current,
  ↪ sig_sram_rw_current, xyz_selector_current,
sig_sram_data_in_current, sig_uart_rx_ready, sig_sram_data_out,
  ↪ sig_sram_address_current, cycles_current, sig_uart_bytes_received_current,
x_coord_current, y_coord_current, z_coord_current, sig_vram_addr_wr_current,
  ↪ sig_vram_ena_wr_current, sig_vram_data_wr_current,
sig_vram_addr_wr_pointer_current, cycles_current, X_coord_rotated_unsigned,
  ↪ Y_coord_rotated_unsigned, Z_coord_rotated_unsigned)
begin
  -- Valores por defecto
  cycles_next <= cycles_current;
  sig_sram_rw_next <= sig_sram_rw_current;
  sig_sram_data_in_next <= sig_sram_data_in_current;
  sig_sram_address_next <= sig_sram_address_current;
  sig_uart_bytes_received_next <= sig_uart_bytes_received_current;
  xyz_selector_next <= xyz_selector_current;
  x_coord_next <= x_coord_current;
  y_coord_next <= y_coord_current;
  z_coord_next <= z_coord_current;
  cycles_next <= cycles_current;
  sig_vram_addr_wr_next <= sig_vram_addr_wr_current;
  sig_vram_ena_wr_next <= sig_vram_ena_wr_current;
  sig_vram_data_wr_next <= sig_vram_data_wr_current;
  sig_vram_addr_wr_pointer_next <= sig_vram_addr_wr_pointer_current;
  sig_led_aux <= "0000";
  case state_current is
    when state_init =>
      sig_uart_bytes_received_next <= 0;
      sig_led_aux <= "0000";
      sig_sram_address_next <= 0;
      if cycles_current = 0 then
        state_next <= state_waiting_for_uart;
      else
        cycles_next <= cycles_current - 1;
        state_next <= state_init;
      end if;
    when state_waiting_for_uart =>
      sig_led_aux <= "0001";
      if sig_uart_rx_ready = '1' then
        state_next <= state_reading_from_uart;
      elsif sig_uart_bytes_received_current = UART_BYTES_TO_RECEIVE then
        state_next <= state_uart_end_data_reception;
      else
        state_next <= state_waiting_for_uart;
      end if;
    when state_reading_from_uart =>
      sig_sram_data_in_next <= sig_uart_readed_data;
      state_next <= state_write_sram;
  end case;
end process;

```

```

    sig_uart_bytes_received_next <= sig_uart_bytes_received_current +
    ↪ 1;
when state_write_sram =>
    sig_sram_address_next <= sig_sram_address_current + 1;
    sig_sram_rw_next <= "1"; -- Necesitamos escribir
    state_next <= state_waiting_for_uart;
when state_uart_end_data_reception =>
    sig_sram_rw_next <= "0"; -- Vamos a necesitar leer
    sig_sram_address_next <= 0; -- La prox address de RAM que
    ↪ nos interesa es 0
    sig_vram_ena_wr_next <= '0';
    sig_vram_data_wr_next <= "0";
    sig_vram_addr_wr_next <= (others=> '0');
    state_next <= state_read_from_sram;
when state_idle =>
    sig_led_aux <= "0010";
    state_next <= state_process_coords;
when state_read_from_sram_prev => --Le da el ciclo de clock que la
    ↪ RAM necesita
    state_next <= state_read_from_sram;
when state_read_from_sram =>
    sig_led_aux <= "0011";
    if sig_sram_address_current > UART_BYTES_TO_RECEIVE then
        state_next <= state_clean_vram;
        xyz_selector_next <= 0;
        sig_sram_address_next <= 0;
    else
        sig_vram_ena_wr_next <= '0';
        case xyz_selector_current is
            when 0 =>
                sig_sram_address_next <= sig_sram_address_current + 1;
                x_coord_next <= sig_sram_data_out;
                xyz_selector_next <= xyz_selector_current + 1;
                state_next <= state_read_from_sram_prev;
            when 1 =>
                sig_sram_address_next <= sig_sram_address_current + 1;
                y_coord_next <= sig_sram_data_out;
                xyz_selector_next <= xyz_selector_current + 1;
                state_next <= state_read_from_sram_prev;
            when 2 =>
                sig_sram_address_next <= sig_sram_address_current + 1;
                z_coord_next <= sig_sram_data_out;
                xyz_selector_next <= 0;
                cycles_next <= 0;
                state_next <= state_process_coords;
            when others =>
                state_next <= state_idle;
        end case;
    end if;
when state_process_coords =>
    if cycles_current < CYCLES_TO_WAIT_TO_CORDIC_TO_FINISH then

```

```

        cycles_next <= cycles_current + 1;
        sig_vram_ena_wr_next <= '0';
        state_next <= state_process_coords;
    else
        state_next <= state_print_coords;
    end if;
when state_print_coords =>
    sig_vram_ena_wr_next <= '1';
    sig_vram_data_wr_next <= "1";
    sig_vram_addr_wr_next <= z_coord_rotated_unsigned(7 downto 0) &
        ↪ y_coord_rotated_unsigned(7 downto 0);
    state_next <= state_read_from_sram_prev;
when state_reset_device =>
    sig_sram_address_next <= 0;
    state_next <= state_clean_sram;
when state_clean_sram =>
    sig_led_aux <= "1000";
    if sig_sram_address_current < ((2**RAM_ADDRESS_WIDTH)-1) then
        sig_sram_address_next <= sig_sram_address_current + 1;
        sig_sram_rw_next <= "1";
        sig_sram_data_in_next <= "00000000";
        sig_vram_ena_wr_next <= '0';
        state_next <= state_clean_sram;
    else
        sig_sram_rw_next <= "0";
        sig_sram_address_next <= 0;
        sig_sram_data_in_next <= "00000000";
        sig_vram_ena_wr_next <= '1';
        sig_vram_addr_wr_pointer_next <= 0;
        state_next <= state_clean_vram_on_first_boot;
    end if;
when state_clean_vram_on_first_boot =>
    sig_led_aux <= "0100";
    if sig_vram_addr_wr_pointer_current < ((2**DPRAM_ADDR_BITS)-1)
        ↪ then
        sig_vram_addr_wr_pointer_next <=
            ↪ sig_vram_addr_wr_pointer_current + 1;
        sig_vram_ena_wr_next <= '1';
        sig_vram_data_wr_next <= "0";
        state_next <= state_clean_vram_on_first_boot;
    else
        sig_vram_addr_wr_pointer_next <= 0;
        sig_vram_ena_wr_next <= '0';
        sig_vram_data_wr_next <= "0";
        state_next <= state_init;
    end if;
    sig_vram_addr_wr_next <=
        ↪ std_logic_vector(to_unsigned(sig_vram_addr_wr_pointer_current,
        ↪ DPRAM_ADDR_BITS));
when state_clean_vram =>
    sig_led_aux <= "0100";

```

```

        if sig_vram_addr_wr_pointer_current < ((2**DPRAM_ADDR_BITS)-1)
            ↪ then
                sig_vram_addr_wr_pointer_next <=
                    ↪ sig_vram_addr_wr_pointer_current + 1;
                sig_vram_ena_wr_next <= '1';
                sig_vram_data_wr_next <= "0";
                state_next <= state_clean_vram;
            else
                sig_vram_addr_wr_pointer_next <= 0;
                sig_vram_ena_wr_next <= '0';
                sig_vram_data_wr_next <= "0";
                state_next <= state_read_from_sram_prev;
            end if;
        sig_vram_addr_wr_next <=
            ↪ std_logic_vector(to_unsigned(sig_vram_addr_wr_pointer_current,
            ↪ DPRAM_ADDR_BITS));
        when others =>
            state_next <= state_idle;
        end case;
    end process;

-- Instanciamos componentes a utilizar
-- VGA
vga_control : vga_ctrl
port map (
    mclk => clk_tl,
    hs => hs_tl,
    vs => vs_tl,
    red_o => red_out_tl,
    grn_o => grn_out_tl,
    blu_o => blu_out_tl,
    red_i => sig_red_enable,
    grn_i => sig_green_enable,
    blu_i => sig_blue_enable,
    pixel_row => sig_aux_pixel_y_i,
    pixel_col => sig_aux_pixel_x_i
);

-- LEDS
led_tl <= not sig_led_aux;

-- SINGLE PORT RAM
ram_internal : sram
port map (
    clka => clk_tl,
    wea => sig_sram_rw_current,
    addra => sig_sram_address,
    dina => sig_sram_data_in_current,
    douta => sig_sram_data_out,
    rsta => not rst_tl
);

```



```

-- UART
uart_load_data : uart
generic map (
    F          => 50000,
    min_baud => 1200,
    num_data_bits => 8
)
port map (
    Rx          => rx_tl,
    Tx          => tx_tl,
    Din => (others => '0'),
    StartTx => '0',
    TxBusy => open,
    Dout      => sig_uart_readed_data,
    RxRdy     => sig_uart_rx_ready,
    RxErr     => open,
    Divisor   => Divisor,
    clk       => clk_tl,
    rst       => not rst_tl
);

-- DUAL PORT RAM (VIDEO MEMORY)
dpram_vram : dpram
generic map(
    DPRAM_BITS_WIDTH => DPRAM_DATA_BITS_WIDTH,
    DPRAM_ADDR_BITS => DPRAM_ADDR_BITS
)
port map(
    rst => not rst_tl,
    clk => clk_tl,
    data_wr => sig_vram_data_wr_current,
    addr_wr => sig_vram_addr_wr_current,
    ena_wr  => sig_vram_ena_wr_current,
    data_rd => sig_vram_data_rd,
    addr_rd => sig_vram_addr_rd
);

-- VIDEO DRIVER
video_driver_1 : video_driver
generic map(
    DPRAM_BITS_WIDTH => DPRAM_DATA_BITS_WIDTH,
    DPRAM_ADDR_BITS => DPRAM_ADDR_BITS
)
port map (
    rst => not rst_tl,
    clk => clk_tl,
    red_en_o => sig_red_enable,
    green_en_o => sig_green_enable,
    blue_en_o => sig_blue_enable,
    pixel_y => pixel_y,

```

```

    pixel_x => pixel_x,
    data_rd => sig_vram_data_rd,
    addr_rd => sig_vram_addr_rd
);

seven_segments : LED8
port map
(
    -- Reset & Clock Signal
    rst => not rst_tl,
    clk => clk_tl,
    -- LED8 PIN
    led_out => led_out_tl,
    digit_select => led_select_tl,
    show_number => sig_binary_to_bcd
);

input_processor_tl : input_processor
generic map (
    ANGLE_WIDTH           => ANGLE_WIDTH,
    ANGLE_STEP_INITIAL    => ANGLE_STEP_INITIAL
)
port map (
    clk => clk_tl,
    matrix_buttons_col => matrix_btn_col_tl,
    matrix_buttons_row => matrix_btn_row_tl,
    angle_x => angle_x_from_input_proc,
    angle_y => angle_y_from_input_proc,
    angle_z => angle_z_from_input_proc,
    angle_step => angle_step_tl,
    reset_button => rst_tl
);

pixel_x <= unsigned(sig_aux_pixel_x_i);
pixel_y <= unsigned(sig_aux_pixel_y_i);

x0 <= signed(std_logic_vector(to_unsigned(0, CORDIC_OFFSET)) &
    ↪ X_coord_current(COORDS_WIDTH-1 downto 0)) when
    ↪ X_coord_current(COORDS_WIDTH-1) = '0' else
    signed(std_logic_vector(to_unsigned((2**CORDIC_OFFSET)-1,
    ↪ CORDIC_OFFSET)) & X_coord_current(COORDS_WIDTH-1 downto 0)) when
    ↪ X_coord_current(COORDS_WIDTH-1) = '1';
y0 <= signed(std_logic_vector(to_unsigned(0, CORDIC_OFFSET)) &
    ↪ Y_coord_current(COORDS_WIDTH-1 downto 0)) when
    ↪ Y_coord_current(COORDS_WIDTH-1) = '0' else
    signed(std_logic_vector(to_unsigned((2**CORDIC_OFFSET)-1,
    ↪ CORDIC_OFFSET)) & Y_coord_current(COORDS_WIDTH-1 downto 0)) when
    ↪ Y_coord_current(COORDS_WIDTH-1) = '1';
z0 <= signed(std_logic_vector(to_unsigned(0, CORDIC_OFFSET)) &
    ↪ Z_coord_current(COORDS_WIDTH-1 downto 0)) when
    ↪ Z_coord_current(COORDS_WIDTH-1) = '0' else

```

```

        signed(std_logic_vector(to_unsigned((2**CORDIC_OFFSET)-1,
        ↪ CORDIC_OFFSET)) & Z_coord_current(COORDS_WIDTH-1 downto 0)) when
        ↪ Z_coord_current(COORDS_WIDTH-1) = '1';

X_coord_rotated_unsigned <= std_logic_vector(X_coord_rotated(COORDS_WIDTH-1
    ↪ downto 0) + to_signed(-(2**(COORDS_WIDTH-1)), COORDS_WIDTH));
Y_coord_rotated_unsigned <= std_logic_vector(Y_coord_rotated(COORDS_WIDTH-1
    ↪ downto 0) + to_signed(-(2**(COORDS_WIDTH-1)), COORDS_WIDTH));
Z_coord_rotated_unsigned <= std_logic_vector(Z_coord_rotated(COORDS_WIDTH-1
    ↪ downto 0) + to_signed(-(2**(COORDS_WIDTH-1)), COORDS_WIDTH));

angle_x_tl <= -1*(signed(to_signed(360, angle_x_from_input_proc'length) -
    ↪ angle_x_from_input_proc)) when angle_x_from_input_proc > 180 else
    ↪ signed(angle_x_from_input_proc);
angle_y_tl <= -1*(signed(to_signed(360, angle_y_from_input_proc'length) -
    ↪ angle_y_from_input_proc)) when angle_y_from_input_proc > 180 else
    ↪ signed(angle_y_from_input_proc);
angle_z_tl <= -1*(signed(to_signed(360, angle_z_from_input_proc'length) -
    ↪ angle_z_from_input_proc)) when angle_z_from_input_proc > 180 else
    ↪ signed(angle_z_from_input_proc);

--angle_x_tl <= angle_x_from_input_proc;
--angle_y_tl <= angle_y_from_input_proc;
--angle_z_tl <= angle_z_from_input_proc;

cordic_rotator: rotator
generic map (
    COORDS_WIDTH=>CORDIC_WIDTH,
    ANGLES_INTEGER_WIDTH=>ANGLE_WIDTH,
    STAGES=>CORDIC_STAGES
)
port map(
    clk=>clk_tl,
    X0=>X0, Y0=>Y0, Z0=>Z0,
    angle_X=>angle_x_tl, angle_Y=>angle_y_tl, angle_Z=>angle_z_tl,
    X=>X_coord_rotated, Y=>Y_coord_rotated, Z=>Z_coord_rotated
);

sig_binary_to_bcd <= std_logic_vector(angle_x_tl(7 downto 0));
sig_sram_address <= std_logic_vector(to_unsigned(sig_sram_address_current,
    ↪ RAM_ADDRESS_WIDTH));

end top_level_arq;

```

### 4.3. input\_processor.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity input_processor is

```

```

generic (
    ANGLE_WIDTH          : integer := 10;
    ANGLE_STEP_INITIAL   : integer := 1
);
port (
    clk: in std_logic;
    matrix_buttons_col : in std_logic_vector (3 downto 0);
    matrix_buttons_row : in std_logic_vector (3 downto 0);
    angle_x : out signed(ANGLE_WIDTH-1 downto 0);
    angle_y : out signed(ANGLE_WIDTH-1 downto 0);
    angle_z : out signed(ANGLE_WIDTH-1 downto 0);
    angle_step: out natural;
    reset_button: out std_logic
);
end input_processor;

architecture Behavioral of input_processor is

component button_matrix is
    port (
        clk : in std_logic;
        matrix_col : in std_logic_vector (3 downto 0);
        matrix_row : in std_logic_vector (3 downto 0);
        reset_button: out std_logic;
        up_x : out std_logic;
        down_x : out std_logic;
        up_y : out std_logic;
        down_y : out std_logic;
        up_z : out std_logic;
        down_z : out std_logic;
        angle_step_up : out std_logic;
        angle_step_down : out std_logic
    );
end component;

component debounce is
    port (
        clk, reset: in std_logic;
        sw: in std_logic;
        db_level, db_tick: out std_logic
    );
end component;

-- Buttons
signal rst: std_logic := '1';

signal up_x_sig: std_logic := '0';
signal down_x_sig: std_logic := '0';
signal up_y_sig: std_logic := '0';
signal down_y_sig: std_logic := '0';
signal up_z_sig: std_logic := '0';

```

```

signal down_z_sig: std_logic := '0';
signal angle_step_up_sig: std_logic := '0';
signal angle_step_down_sig: std_logic := '0';

-- Debounced Buttons
signal up_x_debounced: std_logic := '0';
signal down_x_debounced: std_logic := '0';
signal up_y_debounced: std_logic := '0';
signal down_y_debounced: std_logic := '0';
signal up_z_debounced: std_logic := '0';
signal down_z_debounced: std_logic := '0';
signal angle_step_up_debounced: std_logic := '0';
signal angle_step_down_debounced: std_logic := '0';

-- Aux signals for angles
signal angle_x_current: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_y_current: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_z_current: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_step_current: natural := ANGLE_STEP_INITIAL;

signal angle_x_next: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_y_next: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_z_next: signed(ANGLE_WIDTH-1 downto 0) := (others => '0');
signal angle_step_next: natural := ANGLE_STEP_INITIAL;

begin

    process(clk, rst)
    begin
        if(rst='0') then --rst_tl pin got a pullup
            angle_x_current <= (others=> '0');
            angle_y_current <= (others=> '0');
            angle_z_current <= (others=> '0');
            angle_step_current <= 0;
        elsif (clk'event and clk='1') then
            angle_x_current <= angle_x_next;
            angle_y_current <= angle_y_next;
            angle_z_current <= angle_z_next;
            angle_step_current <= angle_step_next;
        end if;
    end process;

    process(up_x_debounced, down_x_debounced,
            up_y_debounced, down_y_debounced,
            up_z_debounced, down_z_debounced,
            angle_step_up_debounced, angle_step_down_debounced,
            angle_x_current, angle_y_current, angle_z_current, angle_step_current)
    begin

```

```

angle_x_next <= angle_x_current;
angle_y_next <= angle_y_current;
angle_z_next <= angle_z_current;
angle_step_next <= angle_step_current;

if up_x_debounced = '1' then
    if(angle_x_current >= (to_signed(360, angle_x_current'length) -
        ↪ angle_step_current)) then
        angle_x_next <= (angle_x_current + angle_step_current -
            ↪ to_signed(360, angle_x_current'length));
    else
        angle_x_next <= angle_x_current + angle_step_current;
    end if;
end if;
if down_x_debounced = '1' then
    if(angle_x_current < to_signed(angle_step_current,
        ↪ angle_x_current'length)) then
        angle_x_next <= to_signed(360, angle_x_current'length) +
            ↪ angle_x_current - angle_step_current;
    else
        angle_x_next <= angle_x_current - angle_step_current;
    end if;
end if;
if up_y_debounced = '1' then
    if(angle_y_current >= (to_signed(360, angle_y_current'length) -
        ↪ angle_step_current)) then
        angle_y_next <= (angle_y_current + angle_step_current -
            ↪ to_signed(360, angle_y_current'length));
    else
        angle_y_next <= angle_y_current + angle_step_current;
    end if;
end if;
if down_y_debounced = '1' then
    if(angle_y_current < to_signed(angle_step_current,
        ↪ angle_y_current'length)) then
        angle_y_next <= to_signed(360, angle_y_current'length) +
            ↪ angle_y_current - angle_step_current;
    else
        angle_y_next <= angle_y_current - angle_step_current;
    end if;
end if;
if up_z_debounced = '1' then
    if(angle_z_current >= (to_signed(360, angle_z_current'length) -
        ↪ angle_step_current)) then
        angle_z_next <= (angle_z_current + angle_step_current -
            ↪ to_signed(360, angle_z_current'length));
    else
        angle_z_next <= angle_z_current + angle_step_current;
    end if;
end if;
if down_z_debounced = '1' then

```

```

        if(angle_z_current < to_signed(angle_step_current,
        ↪ angle_z_current'length)) then
            angle_z_next <= to_signed(360, angle_z_current'length) +
            ↪ angle_z_current - angle_step_current;
        else
            angle_z_next <= angle_z_current - angle_step_current;
        end if;
    end if;
    if angle_step_up_debounced = '1' then
        if angle_step_current < 30 then
            angle_step_next <= angle_step_current + 1;
        end if;
    end if;
    if angle_step_down_debounced = '1' then
        if angle_step_current > 0 then
            angle_step_next <= angle_step_current - 1;
        end if;
    end if;
end process;

-- instantiante button_matrix
keyboard: button_matrix
port map(
    clk => clk,
    matrix_col => matrix_buttons_col,
    matrix_row => matrix_buttons_row,
    reset_button => rst,
    up_x => up_x_sig,
    down_x => down_x_sig,
    up_y => up_y_sig,
    down_y => down_y_sig,
    up_z => up_z_sig,
    down_z => down_z_sig,
    angle_step_up => angle_step_up_sig,
    angle_step_down => angle_step_down_sig
);

-- instantiate debouncers
debounce_unit_x_up: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not up_x_sig,
    db_level=>open, db_tick=>up_x_debounced
);
debounce_unit_x_down: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not down_x_sig,
    db_level=>open, db_tick=>down_x_debounced
);
debounce_unit_y_up: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not up_y_sig,

```

```

        db_level=>open, db_tick=>up_y_debounced
    );
debounce_unit_y_down: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not down_y_sig,
    db_level=>open, db_tick=>down_y_debounced
);
debounce_unit_z_up: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not up_z_sig,
    db_level=>open, db_tick=>up_z_debounced
);
debounce_unit_z_down: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not down_z_sig,
    db_level=>open, db_tick=>down_z_debounced
);
debounce_unit_step_angle_up: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not angle_step_up_sig,
    db_level=>open, db_tick=>angle_step_up_debounced
);
debounce_unit_step_angle_down: debounce
port map(
    clk=>clk, reset=>not rst, sw=>not angle_step_down_sig,
    db_level=>open, db_tick=>angle_step_down_debounced
);

reset_button <= rst;
angle_x <= angle_x_current;
angle_y <= angle_y_current;
angle_z <= angle_z_current;
angle_step <= angle_step_current;

end Behavioral;

```

#### 4.4. video\_driver.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity video_driver is
    generic (
        DPRAM_BITS_WIDTH : natural := 1; -- Ancho de palabra de la
        ↪ memoria medido en bits
        DPRAM_ADDR_BITS : natural := 16 -- Cantidad de bits de
        ↪ address (tamaño de la memoria es 2^ADDRS_BITS
    );
    port (
        rst: in std_logic;

```



```

        clk: in std_logic;
        red_en_o: out std_logic;
        green_en_o: out std_logic;
        blue_en_o: out std_logic;
        pixel_x: in unsigned(9 downto 0);
        pixel_y: in unsigned(9 downto 0);
        addr_rd : out std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
        data_rd : in std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0)
    );

```

```
end video_driver;
```

architecture Behavioral of video\_driver is

```

    signal sig_vram_addr_rd_current, sig_vram_addr_rd_next :
        ↪ std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
    signal sig_pixel_current, sig_pixel_next: std_logic := '0';

    begin

        process(clk, rst)
        begin
            if(rst='1') then
                sig_vram_addr_rd_current <= "0000000000000000";
                sig_pixel_current <= '0';
            elsif (clk'event and clk='1') then
                sig_vram_addr_rd_current <= sig_vram_addr_rd_next;
                sig_pixel_current <= sig_pixel_next;
            end if;
        end process;

        process(pixel_x, pixel_y, sig_pixel_current, sig_vram_addr_rd_current,
            ↪ data_rd)
        begin
            sig_vram_addr_rd_next <= sig_vram_addr_rd_current;
            sig_pixel_next <= sig_pixel_current;
            if ((to_integer(pixel_y) >= 112) and (to_integer(pixel_y) <=
                ↪ 368) and (to_integer(pixel_x) >= 192) and
                ↪ (to_integer(pixel_x) <= 448)) then
                sig_vram_addr_rd_next <=
                    ↪ std_logic_vector(to_unsigned(to_integer(pixel_x)-192,8)
                    ↪ & to_unsigned(to_integer(pixel_y)-112,8));
                sig_pixel_next <= data_rd(0);
            else
                sig_vram_addr_rd_next <= "0000000000000000";
                sig_pixel_next <= '0';
            end if;
        end process;

        addr_rd <= sig_vram_addr_rd_current;
        red_en_o <= sig_pixel_current;

```

```

green_en_o <= sig_pixel_current;
blue_en_o <= sig_pixel_current;

```

```

end Behavioral;

```

## 4.5. vram.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dpram is
    generic (
        DPRAM_BITS_WIDTH : natural := 1; -- Ancho de palabra de la
        ↪ memoria medido en bits
        DPRAM_ADDR_BITS : natural := 16 -- Cantidad de bits de
        ↪ address (tamaño de la memoria es 2^ADDRS_BITS
    );
    port (
        rst : in std_logic;
        clk : in std_logic;
        data_wr : in std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0);
        addr_wr : in std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
        ena_wr : in std_logic;
        addr_rd : in std_logic_vector(DPRAM_ADDR_BITS-1 downto 0);
        data_rd : out std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0)
    );
end dpram;

architecture rtl of dpram is
    -- Array para la memoria
    subtype t_word is std_logic_vector(DPRAM_BITS_WIDTH-1 downto 0);
    type t_memory is array(2**DPRAM_ADDR_BITS-1 downto 0) of t_word;
    signal ram : t_memory;
    -- Address casting
    signal rd_pointer : integer range 0 to 2**DPRAM_ADDR_BITS-1;
    signal wr_pointer : integer range 0 to 2**DPRAM_ADDR_BITS-1;
    begin
        -- Address casting
        rd_pointer <= to_integer(unsigned(addr_rd));
        wr_pointer <= to_integer(unsigned(addr_wr));
        -- Write
        process(clk)
        begin
            if clk='1' and clk'event then
                if ena_wr='1' then
                    ram(wr_pointer) <= data_wr;
                end if;
            end if;
        end process;
    end rtl;

```

```

        -- Read
        process(clk)
            begin
                if clk='1' and clk'event then
                    data_rd <= ram(rd_pointer);
                end if;
            end process;
        end architecture;

```

## 4.6. cordic.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cordic is
    generic (
        COORDS_WIDTH           : integer := 10;
        ANGLES_INTEGER_WIDTH   : integer := 8;
        STAGES                  : integer := 16
    );
    port (
        X0, Y0                  : in signed(COORDS_WIDTH-1 downto 0);
        angle                   : in signed(ANGLES_INTEGER_WIDTH-1 downto 0);
        X, Y                    : out signed(COORDS_WIDTH-1 downto 0)
    );
end entity cordic;

architecture behavioral of cordic is

    -- Constants
    constant STEP_WIDTH           : integer := 4;
    constant ANGLES_FRACTIONAL_WIDTH : integer := 16;
    constant ANGLES_WIDTH         : integer :=
        ↪ ANGLES_INTEGER_WIDTH+ANGLES_FRACTIONAL_WIDTH;
    constant CORDIC_SCALE_FACTOR   : real := 0.607252935;
    constant MAX_STAGES: natural := 16;

    -- Types
    type rom_type is array (0 to MAX_STAGES-1) of signed(ANGLES_WIDTH-1 downto
        ↪ 0);
    type coordinates_array is array (0 to STAGES) of signed(COORDS_WIDTH-1
        ↪ downto 0);
    type angles_array is array (0 to STAGES) of signed(ANGLES_WIDTH-1 downto
        ↪ 0);

    -- Cordic stage declaration
    component cordic_stage is
        generic (
            COORDS_WIDTH   : integer;
            ANGLE_WIDTH     : integer;

```

```

        STEP_WIDTH      : integer
    );
    port (
        X0, Y0          : in signed(COORDS_WIDTH-1 downto 0);
        Z0              : in signed(ANGLE_WIDTH-1 downto 0);
        atan            : in signed(ANGLE_WIDTH-1 downto 0);
        step            : in unsigned(STEP_WIDTH-1 downto 0);
        X, Y            : out signed(COORDS_WIDTH-1 downto 0);
        Z              : out signed(ANGLE_WIDTH-1 downto 0)
    );
end component cordic_stage;

-- Angles "ROM"
constant STEP2ANGLE_ROM: rom_type := (
    to_signed(integer(45.0 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(26.565051177078 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(14.0362434679265 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(7.1250163489018 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(3.57633437499735 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(1.78991060824607 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.895173710211074 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.447614170860553 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.223810500368538 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.111905677066207 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.055952891893804 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.027976452617004 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.013988227142265 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.006994113675353 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.003497056850704 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH),
    to_signed(integer(0.00174852842698 *
        ↪ real(2**ANGLES_FRACTIONAL_WIDTH)), ANGES_WIDTH)
);

-- Signal arrays
signal sX_array      : coordinates_array := (others =>
    ↪ to_signed(0, COORDS_WIDTH));

```

```

signal sY_array      : coordinates_array := (others =>
  ↪ to_signed(0,COORDS_WIDTH));
signal sZ_array      : angles_array      := (others =>
  ↪ to_signed(0,ANGLES_WIDTH));

-- Buffer signals
signal sX_multiplication_buffer      : signed(2*COORDS_WIDTH-1 downto 0)
  ↪ := (others => '0');
signal sY_multiplication_buffer      : signed(2*COORDS_WIDTH-1 downto 0)
  ↪ := (others => '0');

--
signal Xt: signed(COORDS_WIDTH-1 downto 0) := (others => '0');
signal Yt: signed(COORDS_WIDTH-1 downto 0) := (others => '0');
signal anglet: signed(ANGLES_INTEGER_WIDTH-1 downto 0) := (others => '0');

-- Cordic scale factor
signal sCordic_scale_factor : signed(COORDS_WIDTH-1 downto 0) :=
  ↪ to_signed(integer(CORDIC_SCALE_FACTOR*real(2**(COORDS_WIDTH-1))),
  ↪ COORDS_WIDTH);

begin

-- Inputs initialization
sX_array(0)    <= X0;
sY_array(0)    <= Y0;
sZ_array(0)    <= signed(std_logic_vector(anglet) &
  ↪ std_logic_vector(to_unsigned(0, ANGLES_FRACTIONAL_WIDTH)));

stages_instantiation: for i in 0 to STAGES-1 generate

  current_cordic_stage: cordic_stage
    generic map (
      COORDS_WIDTH    => COORDS_WIDTH,
      ANGLE_WIDTH     => ANGLES_WIDTH,
      STEP_WIDTH      => STEP_WIDTH
    )
    port map (
      X0              => sX_array(i),
      Y0              => sY_array(i),
      Z0              => sZ_array(i),
      atan            => STEP2ANGLE_ROM(i),
      step            => to_unsigned(i, STEP_WIDTH),
      X               => sX_array(i+1),
      Y               => sY_array(i+1),
      Z               => sZ_array(i+1)
    );

end generate stages_instantiation;

-- Scaling

```

```

sX_multiplication_buffer <= sX_array(STAGES)*sCordic_scale_factor;
sY_multiplication_buffer <= sY_array(STAGES)*sCordic_scale_factor;

-- Outputs assignement
-- Buffer has 2 bits for integer part, we only want one and the rest of
↳ fractional bits we can fit
Xt    <= sX_multiplication_buffer(2*COORDS_WIDTH-2 downto
↳ COORDS_WIDTH-1);
Yt    <= sY_multiplication_buffer(2*COORDS_WIDTH-2 downto
↳ COORDS_WIDTH-1);

angle_decode: process(angle, Xt, Yt)
begin
    if angle < to_signed(-90, angle'length) then
        anglet <= angle + to_signed(90, angle'length);
        X <= -Yt;
        Y <= Xt;
    elsif angle > to_signed(90, angle'length) then
        anglet <= angle - to_signed(90, angle'length);
        X <= -Yt;
        Y <= Xt;
    else
        anglet <= angle;
        X <= Xt;
        Y <= Yt;
    end if;
end process;

end architecture behavioral;

```

## 4.7. cordic\_stage.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cordic_stage is

    generic (
        COORDS_WIDTH      : integer := 10;
        ANGLE_WIDTH        : integer := 22;
        STEP_WIDTH         : integer := 4
    );
    port (
        X0, Y0             : in signed(COORDS_WIDTH-1 downto 0);
        Z0                  : in signed(ANGLE_WIDTH-1 downto 0);
        atan                : in signed(ANGLE_WIDTH-1 downto 0);
        step                : in unsigned(STEP_WIDTH-1 downto 0);
        X, Y                : out signed(COORDS_WIDTH-1 downto 0);
        Z                   : out signed(ANGLE_WIDTH-1 downto 0)
    );
end entity cordic_stage;

```

```

end entity cordic_stage;

architecture behavioral of cordic_stage is

    -- Buffer signals
    signal Xshifted: signed(COORDS_WIDTH-1 downto 0) := ( others => '0');
    signal Yshifted: signed(COORDS_WIDTH-1 downto 0) := ( others => '0');
    signal sigma: std_logic := '0';

begin

    Xshifted <= shift_right(X0, to_integer(step));
    Yshifted <= shift_right(Y0, to_integer(step));
    sigma <= Z0(ANGLE_WIDTH-1);

    X <=      X0 - Yshifted   when sigma = '0' else      -- si el ngulo es mayor a
    ↪      cero
        X0 + Yshifted;      -- si el ngulo es menor a
    ↪      cero
    Y <=      Y0 + Xshifted   when sigma = '0' else      -- si el ngulo es mayor a
    ↪      cero
        Y0 - Xshifted;      -- si el ngulo es menor a
    ↪      cero
    Z <=      Z0 - atan       when sigma = '0' else      -- si el ngulo es mayor a
    ↪      cero
        Z0 + atan;          -- si el ngulo es menor a
    ↪      cero

end architecture behavioral;

```

## 4.8. rotator.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rotator is

    generic (
        COORDS_WIDTH           : integer := 10;
        ANGLES_INTEGER_WIDTH    : integer := 10;
        STAGES                  : integer := 16
    );
    port (
        clk                     : in std_logic;
        X0, Y0, Z0              : in signed(COORDS_WIDTH-1 downto 0);
        angle_X, angle_Y, angle_Z : in signed(ANGLES_INTEGER_WIDTH-1
    ↪      downto 0);
        X, Y, Z                 : out signed(COORDS_WIDTH-1 downto 0)
    );

```

```

end entity rotator;

architecture behavioral of rotator is

    -- Cordic declaration
    component cordic is
        generic (
            COORDS_WIDTH           : integer;
            ANGLES_INTEGER_WIDTH   : integer;
            STAGES                  : integer
        );
        port (
            X0, Y0                 : in signed(COORDS_WIDTH-1 downto 0);
            angle                   : in signed(ANGLES_INTEGER_WIDTH-1 downto 0);
            X, Y                   : out signed(COORDS_WIDTH-1 downto 0)
        );
    end component cordic;

    -- Buffer signal
    signal X_rotator_X0           : signed(COORDS_WIDTH-1 downto 0);
    signal X_rotator_Y0           : signed(COORDS_WIDTH-1 downto 0);
    signal X_angle                : signed(ANGLES_INTEGER_WIDTH-1 downto 0);
    signal X_rotator_X            : signed(COORDS_WIDTH-1 downto 0);
    signal X_rotator_Y            : signed(COORDS_WIDTH-1 downto 0);

    signal y_rotator_X0           : signed(COORDS_WIDTH-1 downto 0);
    signal Y_rotator_Y0           : signed(COORDS_WIDTH-1 downto 0);
    signal Y_angle                : signed(ANGLES_INTEGER_WIDTH-1 downto 0);
    signal Y_rotator_X            : signed(COORDS_WIDTH-1 downto 0);
    signal Y_rotator_Y            : signed(COORDS_WIDTH-1 downto 0);

    signal Z_rotator_X0           : signed(COORDS_WIDTH-1 downto 0);
    signal Z_rotator_Y0           : signed(COORDS_WIDTH-1 downto 0);
    signal Z_angle                : signed(ANGLES_INTEGER_WIDTH-1 downto 0);
    signal Z_rotator_X            : signed(COORDS_WIDTH-1 downto 0);
    signal Z_rotator_Y            : signed(COORDS_WIDTH-1 downto 0);

begin

    process(clk)
    begin
        if (clk'event and clk='1') then

            X_rotator_X0 <= Y0;
            X_rotator_Y0 <= Z0;
            X_angle <= angle_X;

            Y_rotator_X0 <= X_rotator_Y;
            Y_rotator_Y0 <= X0;

```



```

    Y_angle <= angle_Y;

    Z_rotator_X0 <= Y_rotator_Y;
    Z_rotator_Y0 <= X_rotator_X;
    Z_angle <= angle_Z;

    X <= Z_rotator_X;
    Y <= Z_rotator_Y;
    Z <= Y_rotator_X;

end if;
end process;

-- X rotator instantiation
x_rotator: cordic
    generic map (
        COORDS_WIDTH           => COORDS_WIDTH,
        ANGLES_INTEGER_WIDTH   => ANGLES_INTEGER_WIDTH,
        STAGES                  => STAGES
    )
    port map (
        X0                      => X_rotator_X0,
        Y0                      => X_rotator_Y0,
        angle                   => X_angle,
        X                       => X_rotator_X,
        Y                       => X_rotator_Y
    );

-- Y rotator instantiation
y_rotator: cordic
    generic map (
        COORDS_WIDTH           => COORDS_WIDTH,
        ANGLES_INTEGER_WIDTH   => ANGLES_INTEGER_WIDTH,
        STAGES                  => STAGES
    )
    port map (
        X0                      => Y_rotator_X0,
        Y0                      => Y_rotator_Y0,
        angle                   => Y_angle,
        X                       => Y_rotator_X,
        Y                       => Y_rotator_Y
    );

-- Z rotator instantiation
z_rotator: cordic
    generic map (
        COORDS_WIDTH           => COORDS_WIDTH,
        ANGLES_INTEGER_WIDTH   => ANGLES_INTEGER_WIDTH,
        STAGES                  => STAGES
    )
    port map (

```

```
        X0      => Z_rotator_X0,  
        Y0      => Z_rotator_Y0,  
        angle   => Z_angle,  
        X       => Z_rotator_X,  
        Y       => Z_rotator_Y  
    );
```

```
end architecture behavioral;
```