TRABAJO DE SISTEMAS ELECTRÓNICOS INDUSTRIALES

CURSO 2021/22

Parte FPGA:

DESCIFRA EL CÓDIGO

Integrantes:

Gómez-Pamo González-Cela, Jorge	54637
González Alday, Javier Pío	54639
González Denia, Adrián	54647

Grupo de clase: A404

ÍNDICE

Introducción

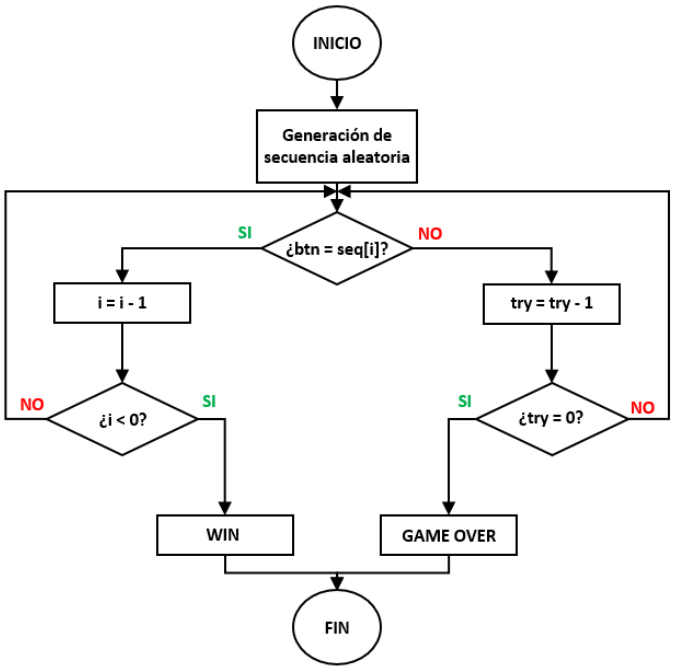
Explicación y normas del juego

Se propone el juego Descifra el Código que trata, como su nombre indica, de adivinar una secuencia de cuatro elementos utilizando botones. Dicha secuencia se genera de forma aleatoria y será representada mediante leds de colores. La base del juego trata de hacer coincidir el botón pulsado con el led correspondiente en orden.

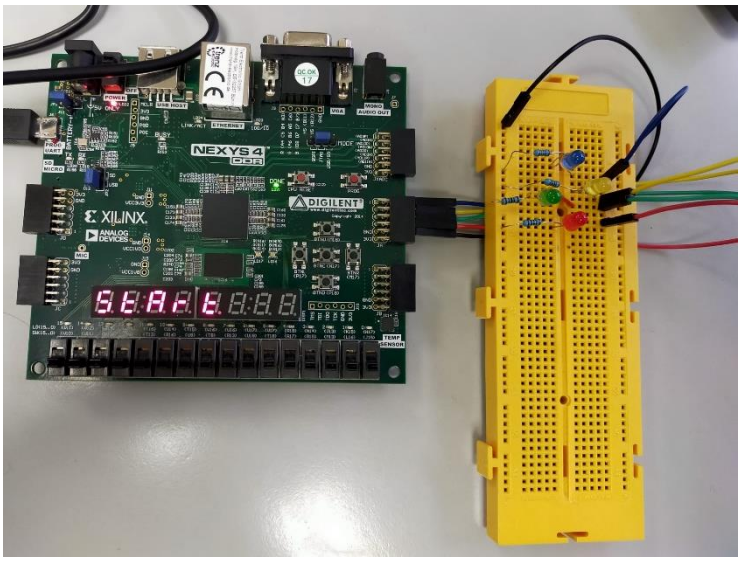
En caso de acertar la secuencia completa, el usuario habrá ganado y el juego acabará.

Por el contrario, si una pulsación no corresponde con algún parámetro del orden de la secuencia, se perderá un intento y habrá que volver a introducir el código desde el principio. Si el usuario se queda sin intentos, habrá perdido y el juego acabará.

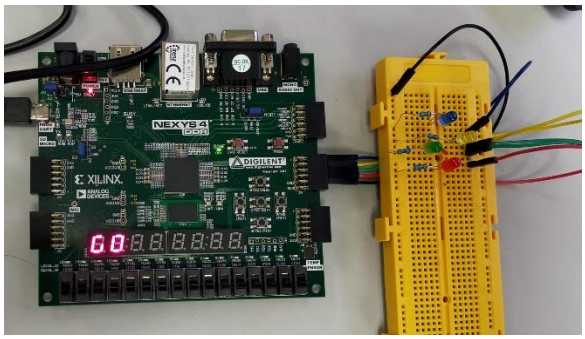
A continuación, se presenta el diagrama de flujo del diseño:



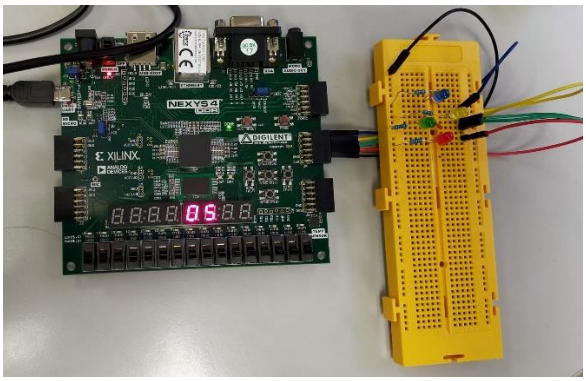
Fotos del proyecto



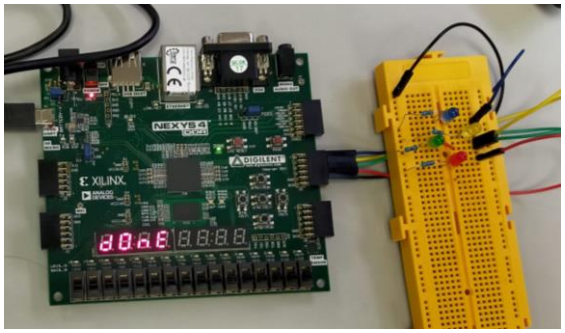
Espera al inicio del juego. Mensaje START.

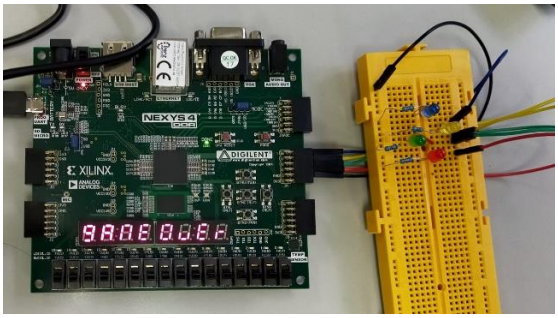


Indicación al jugador de comienzo de introducción de la secuencia. Mensaje GO.

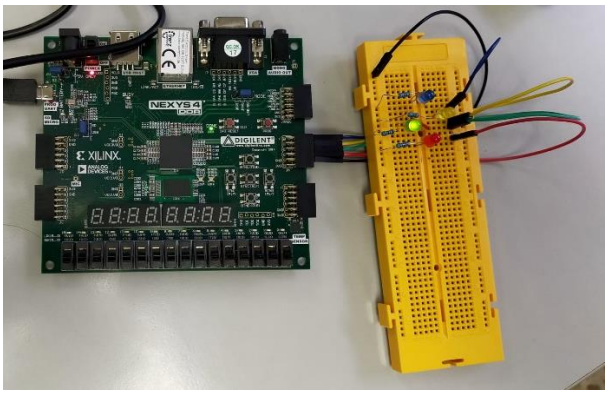


Intentos restantes que le quedan al jugador para adivinar la secuencia (5 en este caso).





En caso de ganar o perder se mostrarán los mensajes de la izquierda o derecha, respectivamente.



Ejemplo de muestra de un elemento de la secuencia final (Led Verde, Botón Izquierda).

Estructura del código

Jerarquía de ficheros del proyecto

A continuación se muestra el árbol de ficheros del proyecto. En el se muestran todos los componentes que constituyen el sistema del juego:

Como puede comprobarse, la entidad general es DESCIFRA_EL_CODIGO_TOP. Esta está definida internamente por otros tres componentes diferentes:

- **BOTON_TOP:** Entidad encargada de la recepción y coordinación de la señales producidas por las pulsaciones de los botones de la placa. Al utilizar los 5 botones disponibles, cada uno es tratado por separado por una instancia diferente de la misma entidad, BOTON_DEBOUNCER.
- **FSM_TOP:** Entidad dedicada a la coordinación de los eventos y fases del juego. Está compuesta por una máquina de estado maestra (FSM_MASTER) encargada de determinar las fases del juego, y varias esclavas encargadas de coordinar los eventos e interacciones con el jugador.
- **VISUALIZER_TOP:** Entidad encargada de la muestra de mensajes a través de los displays de 7 segmentos integrados en la placa. Estos mensajes le indicarán al jugador en la fase en la que se encuentra (inicio del juego, si ha ganado o perdido, etc…).

A lo largo de los siguientes puntos se realizará una descripción más detallada de cada componente por separado.

DESCIFRA_EL_CODIGO_TOP

Descripción del componente

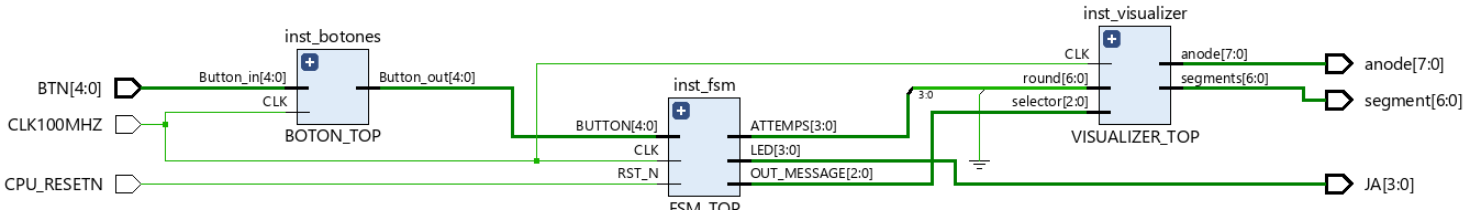
Como se ha indicado anteriormente, esta entidad es la encargada de juntar de forma general todos los componentes necesarios para el funcionamiento del juego. De esta forma, es posible simplificar la estructura general del sistema para considerarlo una "caja negra" con las siguientes entradas y salidas:

- **BTN:** Vector de 5 elementos de entrada. Cada elemento se corresponde con el valor de la señal de cada uno de los 5 botones de la placa (OK, UP, DOWN, LEFT, RIGHT).
- **CLK100MHZ:** Entrada la de señal de reloj de 100 MHz. Utilizada por todos los componentes del sistema.
- **CPU_RESETN:** Entrada de la señal de RESET del sistema. Es activa a nivel bajo.
- **anode:** Salida de un vector de 8 elementos. Utilizado para indicar el display de 7 segmentos en el que se desea encender. Cada elemento corresponde a un display diferente.
- **segment:** Salida de un vector de 7 elementos. Utilizado para indicar los segemntos ha encender en cada display según el mensaje que se quiera mostrar.
- **JA:** Salida de un vector de 4 elementos. Estos corresponden a 4 de los puertos digitales laterales de la placa. Estos son utilizados para controlar el encendido y apagado de los LEDs externos que muestra la secuencia que el jugador tendría que haber introducido para ganar.

Estructura interna del componente (diagrama RTL)

El código de esta entidad es del estilo estructural, ya que se interconectan los distintos componentes, entradas y salidas mediante señales. El código es el siguiente:

De esta forma, el diagrama de bloques RTL de la entidad quedará de la siguiente forma:



Funcionalidades y componentes del sistema

A continuación se numerarán las distintas funcionalidades del sistema y los componentes que se han programado para cada una de ellas.

FUNCIONALIDAD	COMPONENTE
Recepción de las señales del jugador mediante los 5 botones de la placa.	BOTON_TOP
Control de las fases del juego según la interacción del jugador.	FSM_TOP
Interfaz visual del juego, incluyendo el control de los displays de 7 segmentos y leds externos a la placa.	VISUALIZER_TOP

BOTON_TOP

Descripción del componente

Se trata de la entidad top que instancia los cinco botones que se deben utilizar mediante el componente BOTON_DEBOUNCER, siendo una conexión entre las entradas asíncronas y la máquina de estados principal.

Considerándose como "caja negra" se establecen como entradas:

- **Button_in:** Vector de 5 elementos de entrada. Correspondientes con el vector de entradas BTN de la entidad top.
- **CLK:** Señal de reloj que sincronizará el proceso.

Siendo las salidas:

- **Button_out:** Vector de 5 elementos de salida. Dicha salida será la pulsación sincronizada a un pulso de reloj evitando rebotes.

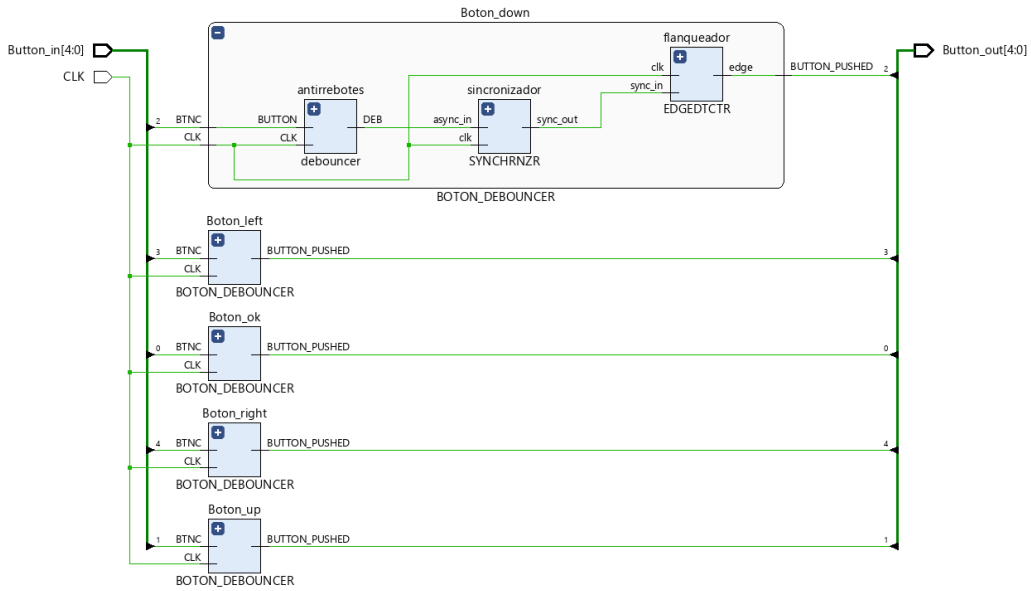
A su vez, el componente BOTON_DEBOUNCER se trata de una entidad top donde se instancian los componentes correspondientes con la sincronización de los pulsos. Las entradas y salidas de dicho componente son las mismas que la entidad BOTON_TOP, de tal forma que al instanciarse cinco veces, cada instancia de entrada como salida corresponden con un elemento del vector.

Estructura interna del componente (diagrama RTL)

El código de esta entidad es del estilo estructural, ya que se interconectan los distintos componentes, entradas y salidas mediante señales. El código es el siguiente:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity BOTON_TOP is
    port(
        Button_in   :in  std_logic_vector(4 downto 0);
        Button_out  :out std_logic_vector(4 downto 0);
        CLK         :in  std_logic);
end BOTON_TOP;
architecture structural of BOTON_TOP is
  component BOTON_DEBOUNCER is
    port (
              CLK   : in     std_logic;
              BTNC      : in     std_logic;
              BUTTON_PUSHED:out  std_logic);
  end component;
begin
    Boton_ok:       BOTON_DEBOUNCER port map(CLK => CLK, BTNC => Button_in(0), BUTTON_PUSHED => Button_out(0));
    Boton_up:       BOTON_DEBOUNCER port map(CLK => CLK, BTNC => Button_in(1), BUTTON_PUSHED => Button_out(1));
    Boton_down:     BOTON_DEBOUNCER port map(CLK => CLK, BTNC => Button_in(2), BUTTON_PUSHED => Button_out(2));
    Boton_left:     BOTON_DEBOUNCER port map(CLK => CLK, BTNC => Button_in(3), BUTTON_PUSHED => Button_out(3));
    Boton_right:    BOTON_DEBOUNCER port map(CLK => CLK, BTNC => Button_in(4), BUTTON_PUSHED => Button_out(4));

end architecture structural;
```

De esta forma, el diagrama de bloques RTL de la entidad quedará de la siguiente forma:

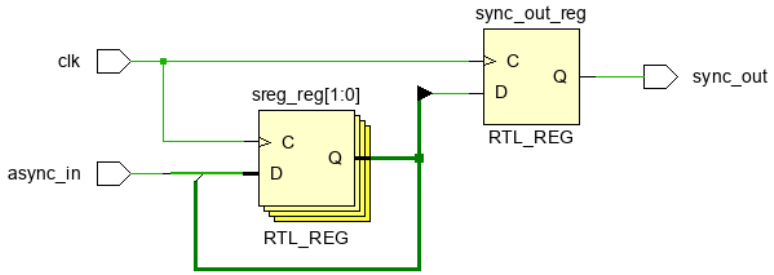


Componentes internos

A continuación se describen todos los componentes internos de la BOTON_TOP.

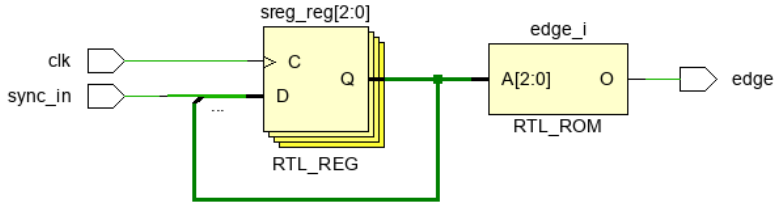
SYNCHRNZR

Se utiliza para evitar la inestabilidad ante la pulsación de los botones donde se quiere detectar el flanco de subida en la pulsación. Consiste en varios Flip-Flop de manera que la entrada asíncrona, en este caso la pulsación, se sincronice con la señal de reloj.



EDGEDTCTR

Complementa al sincronizador de manera que cuando se produce el pulso del botón, la duración del pulso es mucho mayor que la duración de un ciclo de reloj. Con este componente se produce un único pulso del tamaño del periodo del reloj con el flanco de bajada del botón.

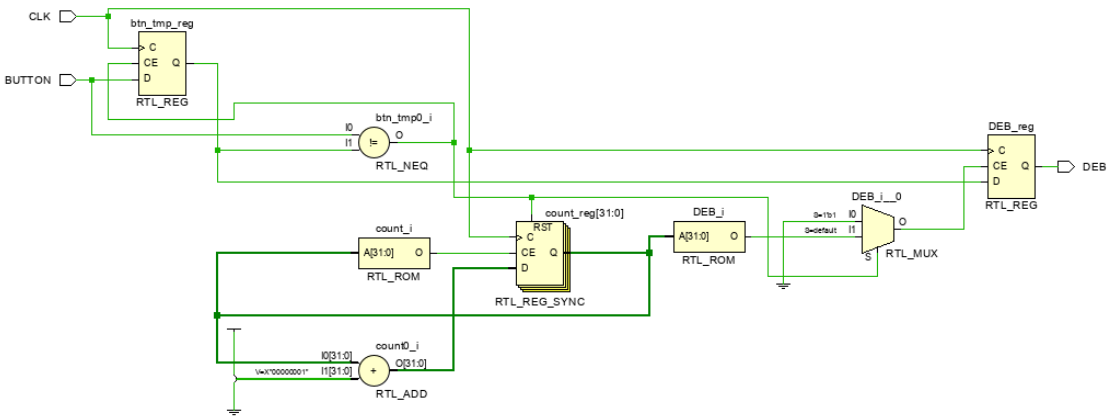


DEBOUNCER

En la Nexys4 DDR suele ser suficiente con los componentes mencionados anteriormente, sin embargo, para la realización de este proyecto, en cuanto se produce una pulsación del botón, es necesario que se realicen una serie de procesos, en este caso la cuenta de un contador, por lo que aún se producen rebotes.

Para ello se propone el siguiente componente antirrebotes, que consiste en la espera de un determinado tiempo, contado mediante ciclos de reloj, para la detección del pulso correcto.

Este pulso pasará a los dos siguientes componentes, consiguiendo unos botones correctamente sincronizados y sin rebotes.



FSM_TOP

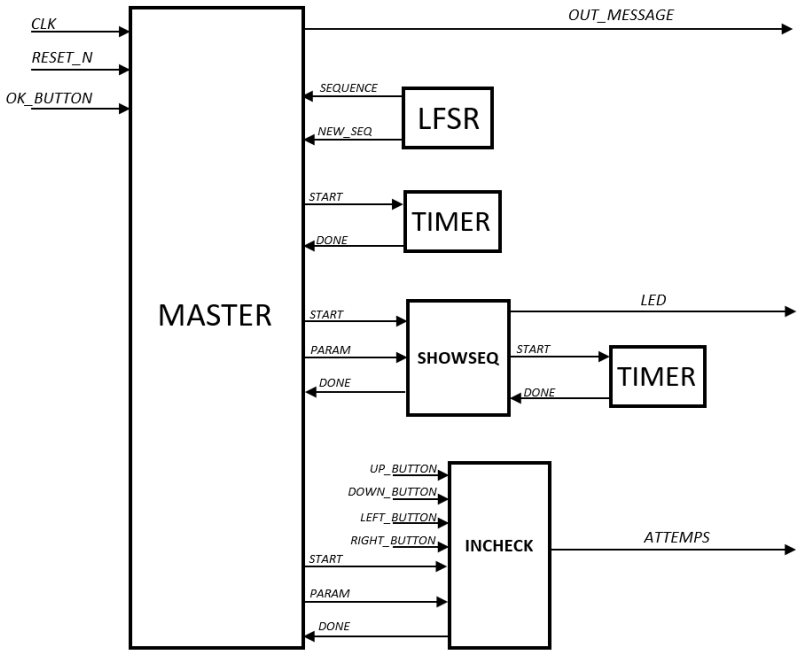
Descripción del componente

El componente FSM_TOP es el encargado de juntar todos los elementos necesarios para conformar la lógica secuencial del sistema. Como realizar todas las tareas en una sola arquitectura, se ha optado por dividir el diseño en varios componentes dedicados a cada funcionalidad. La siguiente tabla muestra las distintas funcionalidades de FSM_TOP:

FUNCIONALIDAD	COMPONENTE
Control general de las distintas fases del juego, transmitiendo las distintas señales a los componentes esclavos. Según su estado, se muestra un mensaje diferente por los displays.	FSM_MASTER
Generación de secuencias aleatorias para cada partida.	LFSR
Temporizador para la realización de esperas de varios segundos	SLAVE_TIMER
Comprobación de los inputs del jugador y conteo de los intentos restantes.	FSM_INCHECK
Muestra de la secuencia que habría que haber introducido mediante unos LEDs externos a la placa.	FSM_SHOWSEQ

En los siguientes puntos se describirá de forma detallada cada uno de los componentes que conforma FSM_TOP.

A modo aclarativo, a continuación se muestra un **esquema de diseño** de FSM_TOP:



La **entidad** de la FSM_TOP es la siguiente:

```
entity FSM_TOP is
    port (  CLK : in std_logic;
            RST_N : in std_logic;
            BUTTON    : in std_logic_vector(4 downto 0); -- ( 0 OK - 1 UP -  2 DOWN - 3 LEFT -  4 RIGHT)
            LED       : out std_logic_vector(3 downto 0); --
            ATTEMPS : out natural range 0 to 10;
            OUT_MESSAGE : out std_logic_vector(2 downto 0));
            -- "000" si nada // "001" si START // "010" si GO // "011" si GAME OVER // "100" si WIN
end FSM_TOP;
```

Explicación de las entradas y salidas de control:

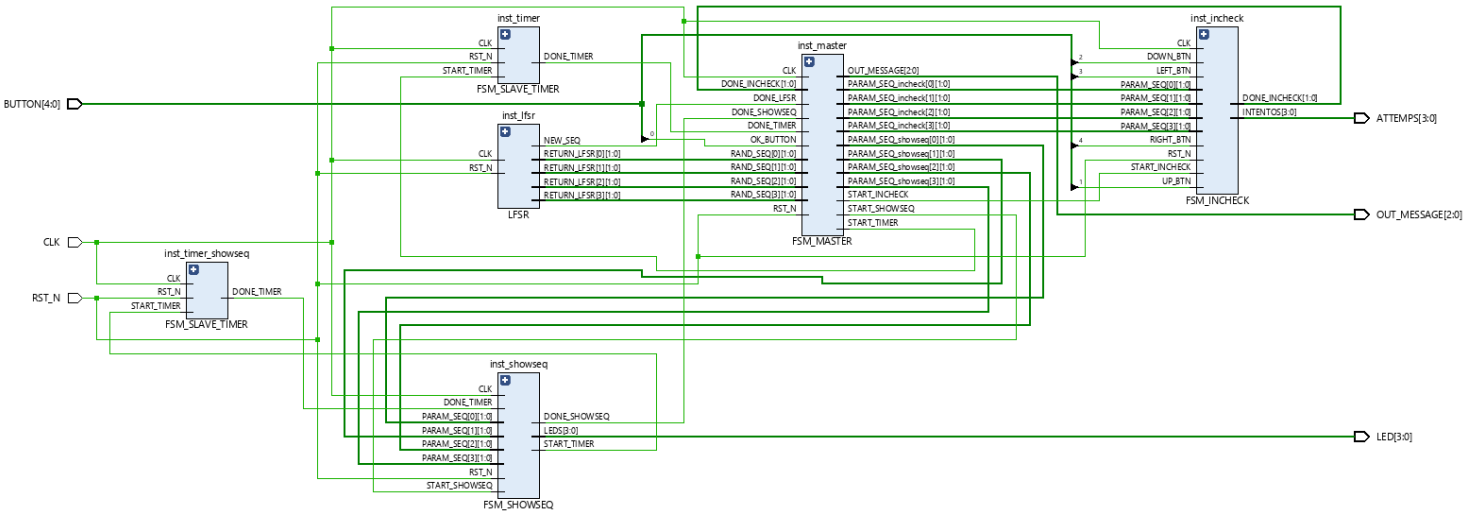
- **BUTTON**: Señales correspondientes a los botones de la placa. Vector de 5 elementos al que cada botón le corresponde un elemento.

- **LED**: Vector de 4 elementos que indica a visualizar_top que led hay que encender en cada momento.
- **ATTEMPS**: Número que indica a VISUALIZER_TOP cuántos intentos le quedan al jugador para mostrarlos por los displays.
- **OUT_MESSAGE**: Vector de 3 elementos que indica a VISUALIZER_TOP que mensaje mostrar por los displays.

Estructura interna del componente (diagrama RTL)

El código de esta entidad es del estilo estructural, ya que se interconectan los distintos componentes, entradas y salidas mediante señales. El código es el siguiente:

De esta forma, el diagrama de bloques RTL de la entidad quedará de la siguiente forma:



Componentes internos

A continuación se describen todos los componentes internos de la FSM_TOP.

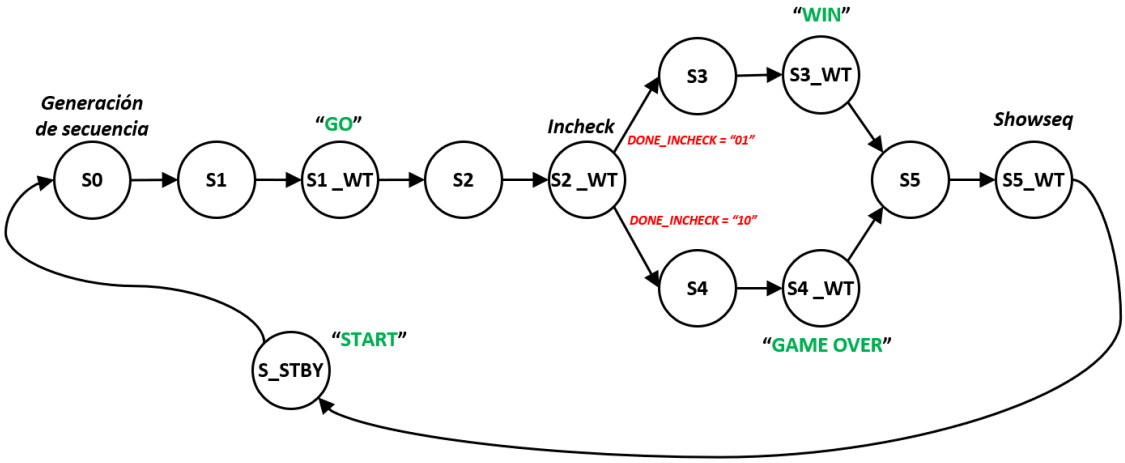
FSM_MASTER

Esta entidad consiste en una máquina de estados maestra encargada de coordinar las distintas fases del juego y las máquinas de estado y componentes esclavos. Por lo tanto, su entidad (interfaz con el exterior) es la siguiente:

```
entity FSM_MASTER is
    generic(    TEST_SEQ : SEQUENCE_T := ("0001", "0010", "0100", "1000");
                TEST_SEQ_2 : SEQUENCE2_T := ("00", "01", "10", "11"));
    port (  CLK : in std_logic;
            RST_N : in std_logic;
            OK_BUTTON    : in std_logic;
            OUT_MESSAGE : out std_logic_vector(2 downto 0);

            -- Interfez entre MASTER y TIMER
            START_TIMER : out std_logic;
            DONE_TIMER  : in std_logic;

            -- Interfaz entre MASTER y LFSR
            RAND_SEQ    : in SEQUENCE2_T;
            DONE_LFSR   : in std_logic;

            -- Interfaz entre MASTER e INCHECK
            START_INCHECK     : out std_logic;
            PARAM_SEQ_incheck : out SEQUENCE2_T;
            DONE_INCHECK      : in std_logic_vector(1 downto 0);

            -- Interfaz entre MASTER e SHOWSEQ
            START_SHOWSEQ     : out std_logic;
            PARAM_SEQ_showseq : out SEQUENCE2_T;
            DONE_SHOWSEQ      : in std_logic
            );
end FSM_MASTER;
```

Como puede observarse, a FMS_MASTER solo entra la señal del botón OK desde el componente de botones. Otro aspecto a destacar es la salida OUT_MESSAGE, que tomará distintos valores en función del mensaje que se quiera mostrar por los displays.

El diagrama de estados de la máquina FSM_MASTER es el siguiente:



A continuación se describe el significado de los diferentes estados:

- **S_STBY**: El juego aún no se ha iniciado. Para iniciarlo, se espera una pulsación del botón central (OK) de los 5 de la placa. Hasta que no se inicie, se muestra un mensaje de "START" por los displays.
- **S0**: Estado de generación de la secuencia a adivinar por el jugador. El componente LFSR está continuamente generando nuevas secuencias aleatorias a adivinar. Cuando se recibe una señal de **DONE_LFSR** = '1', se guarda la nueva secuencia a adivinar en una variable, y se continua con el desarrollo del juego.
- **S1**: Disparo del temporizador durante 2 segundos para mostrar un mensaje.
- **S1_WT**: Estado de espera de muestra del mensaje de "GO", indicándole al jugador que comience a introducir la secuencia que considere.
- **S2**: Disparo de la máquina de estados esclava de INCHECK.
- **S2_WT**: Estado de espera de la máquina maestra mientras la máquina esclava INCHECK está en funcionamiento. Existen dos posibilidades: Que se finalice el juego ganando (se recibe un **DONE_INCHECK** = "01"), o que se finalice el juego perdiendo (se recibe un **DONE_INCHECK** = "10").
- **S3**: Disparo del temporizador durante 2 segundos para mostrar un mensaje.
- **S3_WT**: En caso de ganar, se muestra un mensaje de "WIN" por los dispalys.
- **S4**: Disparo del temporizador durante 2 segundos para mostrar un mensaje.
- **S4_WT**: En caso de perder, se muestra un mensaje de "GAME OVER" por los dispalys.
- **S5**: Disparo de la máquina de estados esclava de SHOWSEQ.
- **S5_WT**: Estado de espera hasta la finalización de SHOWSEQ

Para la programación de esta máquina de estados se ha declarado un tipo especial denominado **STATE_MASTER_T**. En el quedan incluidos todos los estados mostrados anteriormente.

Además, cabe destacar que se ha utilizado un modelo de una máquina de estados a partir de tres procesos diferentes:

- **state_register**: Encargado de actualizar el estado de la máquina de estados a cada pulso de reloj. En caso de llegar una señal de RESET, se vuelve la S_STBY.
- **nxt_state_decoder**: En este proceso quedan reflejadas todas las transiciones entre los diferentes estados, producidas por las entradas a la máquinas de estados.
- **output_decoder**: Proceso dedicado a, según el estado actual de la máquina de estados, activar unas salidas u otras.

FSM_INCHECK

Este componente consiste en una máquina de estados esclava encargada de la recepción y comprobación de los valores introducidos por el jugador. Cada vez que se realiza una pulsación, FSM_INCHECK comprueba si el valor introducido coincide con el de la posición correspondiente de la secuencia generada aleatoriamente por LFSR.

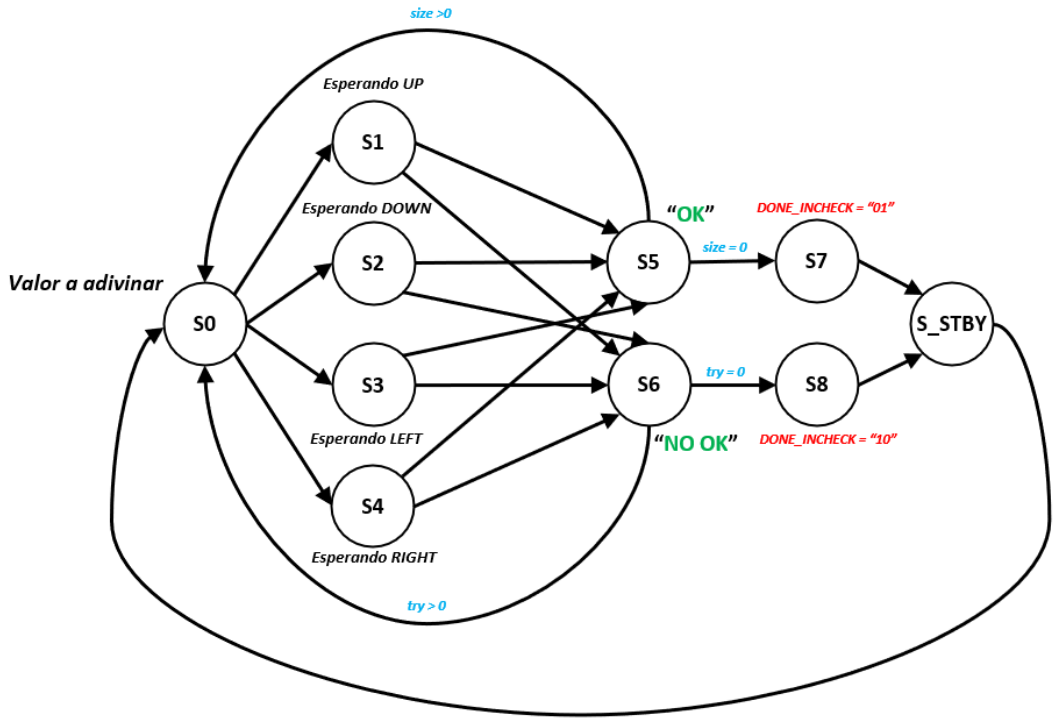
Cabe destacar que los elementos de la secuencia a adivinar solo pueden tomar cuatro valores diferentes (00: UP; 01: DOWN; 10: LEFT; 11: RIGHT).

La entidad del componente es la siguiente:

```
entity FSM_INCHECK is
    port (  CLK : in STD_LOGIC;
            RST_N           : in std_logic;
            START_INCHECK   : in std_logic;
            PARAM_SEQ       : in SEQUENCE2_T;

            UP_BTN          : in std_logic;
            DOWN_BTN        : in std_logic;
            LEFT_BTN        : in std_logic;
            RIGHT_BTN       : in std_logic;

            DONE_INCHECK    : out std_logic_vector(1 downto 0);
            INTENTOS        : out natural range 0 to 9);
end FSM_INCHECK;
```

A este componente entran el resto de las señales de los botones (**x_BTN**) y la secuencia a adivinar (**PARAM_SEQ**). Por otro lado además de la salida que indica el fin de la ejecución de la máquina de estados (**DONE_INCHECK**) que tomará 3 valores diferentes para indicar el estado de finalización de la máquina, está la salida de **INTENTOS**, que le indicará a **VISUALIZER** los intentos restantes del jugador que hay que mostrar por los displays.

El diagrama de estados de la máquina FSM_INCHECK es el siguiente:



A continuación se describe el significado de los diferentes estados:

- **S_STBY**: Estado de reposo de la máquina de estados. Hasta que la FSM_MASTER no indique el comienzo de la ejecución de la máquina esclava, no se produce ninguna transición.
- **S0**: Estado en el que se comprueba el elemento actual a adivinar de la secuencia. Dependiendo del valor del elemento, se realizará una transición
- **S1 a S4**: Estados de espera al input correcto que el jugador debe adivinar.
- **S5**: En caso de que el jugador acierte el botón que debía ser pulsado, se pasa a este estado. Llegados ha este punto hay que determinar si aún quedan elementos por adivinar o si se ha llegado al final de la secuencia. Ambas comparaciones se realizan mediante la señal **size**. Si su valor es 0, significa que el jugador adivinó toda la secuencia (paso a S7). En caso contrario, significa que aún quedan elementos de la secuencia que adivinar (paso a S0).
- **S6**: Es el estado opuesto a S5. El jugador ha introducido erróneamente el valor a adivinar de la secuencia, por lo que se le restará un intento y se volverá al inicio de la secuencia. En caso de no haber consumido todos los intentos (**try**>0), se vuelve al estado S0. En caso contrario, se ha perdido, por lo que se pasa al estado S8.
- **S7**: Estado de terminación del juego WIN. Se manda un valor de terminación para indicar a FSM_MASTER que se ha ganado el juego (**DONE_INCHECK = "01"**).
- **S8**: Estado de terminación del juego GAME OVER. Se manda un valor de terminación para indicar a FSM_MASTER que se ha perdido el juego (**DONE_INCHECK = "10"**).

La salida **DONE_INCHECK** toma un valor de "00" en todo momento excepto en los estados S7 y S8. De esta forma se le indica a FSM_MASTER.

La máquina de estados FSM_INCHECK también se ha planteado con una **estructura de 3 procesos** diferentes que se mostró en el apartado de FSM_MASTER.

Otro aspecto a destacar de la programación del componente son las señales **size** y **try**. Con el fin de simplificar la programación, en vez de tener que utilizar un contador para cada una de las señales para recordar sus valores, se ha optado por hacerlas **variables de estado**. Sus valores se van actualizando continuamente en el proceso de **state_register**, y en el **nxt_state_decoder** se varía su valor dependiendo del estado en el que se encuentre la máquina de estados.

FSM_SHOWSEQ

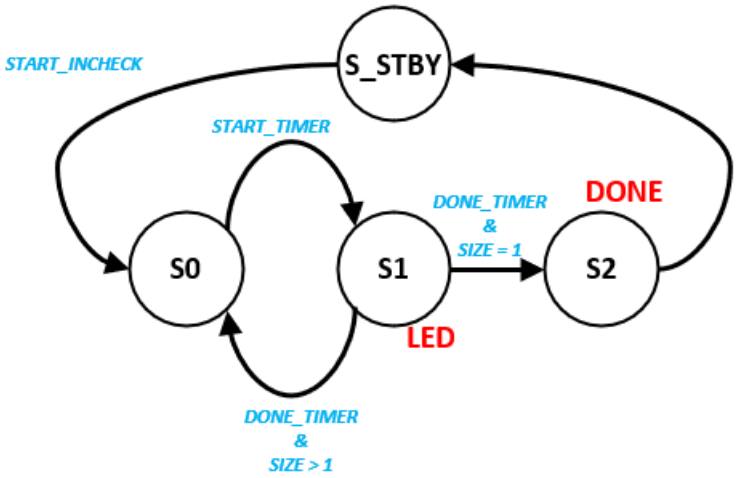
Esta es la segunda máquina de estados esclavas utilizada en el sistema. Su objetivo es, al final del juego, mostrar la secuencia que el jugador debería haber adivinado. Esto se realiza a través de los LEDs dispuestos en la protoboard externa a la placa de desarrollo. Cada LED corresponde a un valor que puede tomar cada elemento de la secuencia, que quedará encendido 2 segundos.

Entonces, las entradas y salidas requeridas para el funcionamiento de la máquina de estados serán las que se muestran en la entidad del componente:

```
entity FSM_SHOWSEQ is

port (
        CLK          : in std_logic;
        RST_N        : in std_logic;
        PARAM_SEQ    : in SEQUENCE2_T; -- Secuencia aleatoria a adivinar por el jugador

        START_TIMER : out std_logic;
        DONE_TIMER   : in std_logic;

        START_SHOWSEQ : in std_logic;
        DONE_SHOWSEQ  : out std_logic;

        LEDS         : out std_logic_vector(3 downto 0)
        --STATE        : out STATE_SHOWSEQ_T
);
end entity;
```

A FSM_SHOWSEQ entra la secuencia del juego, que será mostrada a partir de los LEDS. Además se incluye una interfaz con la FSM_SLAVE_TIMER para realizar las esperas en la muestra de los elementos.

El diagrama de estados de la máquina FSM_SHOWSEQ es el siguiente:



A continuación se describe el significado de los diferentes estados:

- **S_STBY**: Estado de reposo de la máquina de estados.
- **S0**: Estado para el disparo del temporizador.
- **S1**: Estado de espera al temporizador. Aquí se muestra el led correspondiente al valor del elemento de la secuencia.
- **S2**: Estado de finalización. Mandamos señal de DONE a FSM_MASTER.

Cabe destacar que también se utiliza una señal **size** como **variable de estado** en vez de utilizar un contador anexo, como se ha explicado anteriormente.

LFSR

Este componente es el encargado de la creación de la secuencia de 4 valores aleatorios a adivinar por el jugador.

Al ser 4 elementos los que hay que rellenar, este componente está continuamente creando nuevas secuencias, y cuando llegue a una nueva secuencia completa, avisa a FSM_MASTER co una señal de DONE.

La entidad es la siguiente:

```
entity LFSR is
    generic( WIDTH : positive := 10); -- Elementos del registro de desplazamiento (Aplicar los TRAPS en 10 y 7)

    port (  CLK          : in std_logic;
            RST_N        : in std_logic;
            -- Señal de salida que indica cuando se ha generado una nueva secuencia
            NEW_SEQ      : out std_logic;
             -- Señal de reset asincrona. OJO! Nunca reinicial el valor del registro de estados con TODO 0. Se bloquea.
            RETURN_LFSR : out SEQUENCE2_T);
end LFSR;
```

Para la generación de estos valores se ha programado un **Linear Feedback Shift Register (LFSR)**. Consiste en un registro de desplazamiento realimentado por su entrada con un valor producto de una operación lógica de varios de sus valores internos. En este caso es un registro de 10 elementos, que se realimenta con el resultado del XOR de los elementos 7 y 10. De esta forma se consigue la generación de una secuencia pseudoaleatoria de $2^n - 1$ elementos.

La secuencia se irá rellenando con los 2 elementos finales del registro en cada ciclo de reloj.

La programación se lleva a cabo a partir de 2 procesos concurrentes. El primero, **register_updater**, es el encargado de actualizar el registro de desplazamiento en cada ciclo de reloj. Y el segundo, **output_updater**, se encarga de ir rellenando la secuencia con los 2 últimos valores del registro. Cuando esta secuencia queda rellenada, activa la salida NEW_SEQ para indicar al FSM_MASTER que hay una nueva secuencia disponible.

FSM_SLAVE_TIMER

Este último componente consiste en un contador de ciclos de reloj. Su entidad está programada para únicamente recibir una señal de entrada de inicio de la cuenta y otra señal de salida para indicar la finalización de la misma.

La entidad del componente es la siguiente:

```
entity FSM_SLAVE_TIMER is
    generic( DELAY : natural := 200000000);

    port (  CLK         : in std_logic;
            RST_N       : in std_logic;
            START_TIMER : in std_logic;
            DONE_TIMER  : out std_logic);

end FSM_SLAVE_TIMER;
```

Como es posible comprobar en el genérico DELAY, este contador se cargará con un valor de doscientos millones, lo que ha una velocidad de 100MHz del reloj, el periodo del reloj será de 2 segundos.

VISUALIZER_TOP

Descripción del componente

El componente tiene como entradas:

- **CLK**: corresponde con la señal del reloj, se utiliza como referencia para el encendido y apagado de cada uno de los displays de 7 segmentos.
- **round**: es un natural con rango de 0 a 99 que representa el número de vidas que tiene el jugador, para así poder mostrarlo en los displays.
- **selector**: es una señal que de tipo std_logic_vector(2 downto 0) que informa según su valor a visualizar de qué debe mostrar por pantalla. El visualizar puede poner por pantalla un mensaje predeterminado que tiene guardado como gameover, start, go… o poner por pantalla el número de vidas que le queda al jugador.

Como salidas tiene:

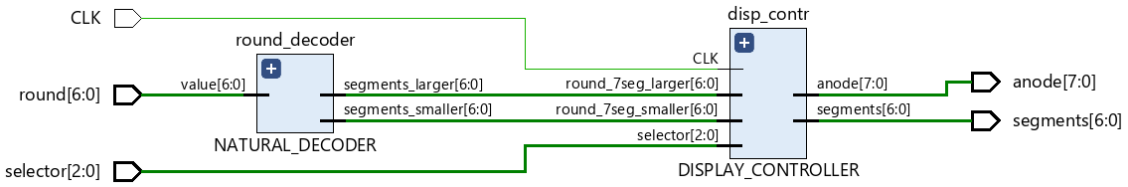
- **segments**: está conectado a los cátodos del display y representa los segmentos del display que se deben de encender, un valor bajo significa que se encenderá el segmento y valor alto que estará apagado.
- **anode**: está conectado a los ánodos del display y podemos seleccionar que displays estén encendidos y cuales apagados. Un nivel bajo significa que le display estará encendido.

Estructura interna del componente (diagrama RTL)

El código de esta entidad es del estilo estructural, ya que se interconectan los distintos componentes, entradas y salidas mediante señales. El código es el siguiente:

```
entity VISUALIZER_TOP is
  Port (
    CLK       : in std_logic;
    round     : in natural range 0 to 99;
    selector  : in std_logic_vector(2 downto 0);
    segments  : out std_logic_vector(6 downto 0);
    anode     : out std_logic_vector(7 downto 0));
end VISUALIZER_TOP;
```

De esta forma, el diagrama de bloques RTL de la entidad quedará de la siguiente forma:



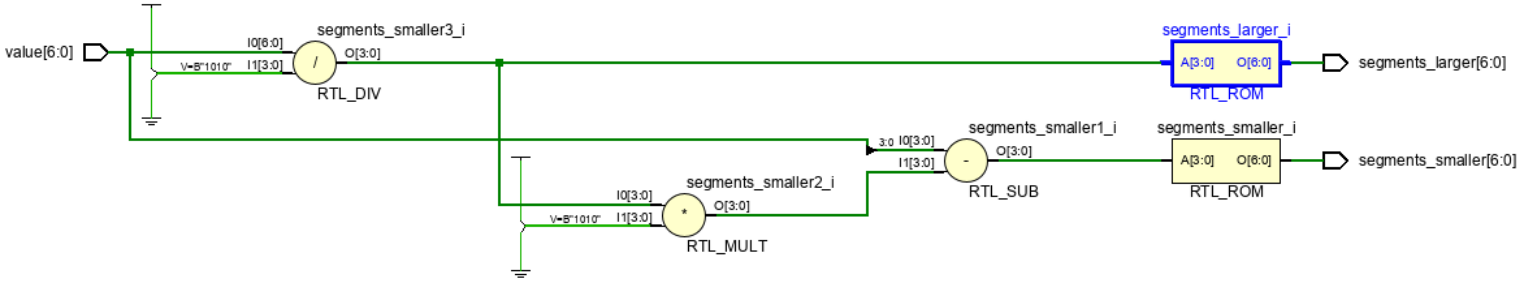
Componentes internos

A continuación, se describen todos los componentes internos de la VISUALIZER_TOP.

NATURAL_DECODER

Se encarga de convertir la variable de entrada "**round**" de entero de dos dígitos a su representación en un display de siete segmentos por dígito. Primero se separa el numero natural en sus dígitos y después en función de su valor se les asigna a las salidas segment_larger, que corresponde con el digito de las decenas, y segment_smaller el estado (apagado o encendido) de los segmentos del display para dibujar el dígito.

```
entity NATURAL_DECODER is
  port (
    value : in natural range 0 to 99;
    segments_smaller : out std_logic_vector(6 downto 0);
    segments_larger : out std_logic_vector(6 downto 0)
  );
end NATURAL_DECODER;
```



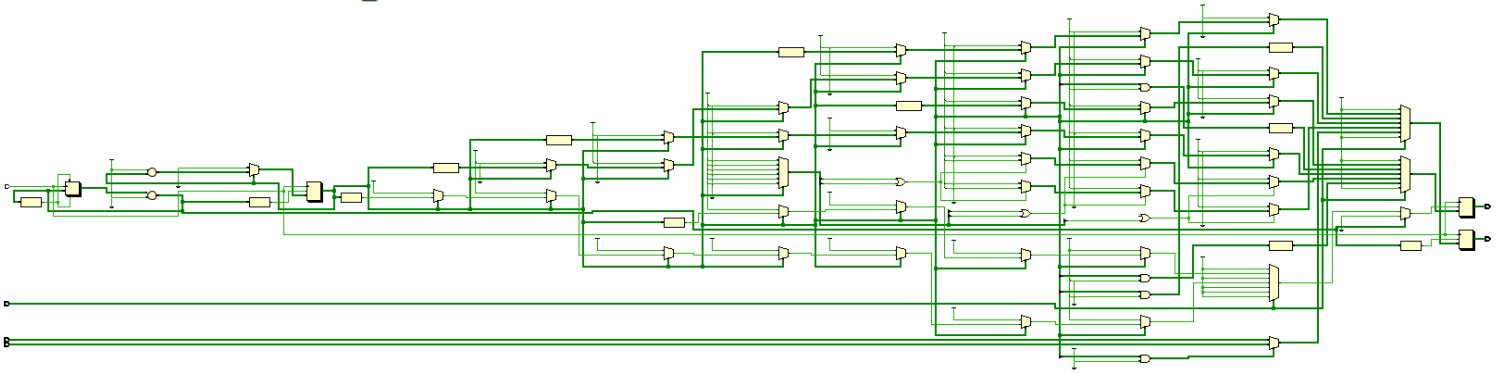
DISPLAY_CONTROLLER

Se encarga de seleccionar qué se representa en los displays y el encendido y apagado de estos para poder da la ilusión de que se representan valores distintos en cada display aunque los cátodos estén todos unidos. Para la selección de qué ánodo se activa en cada momento se hace uso de la señal "anode_s", la cual empieza valiendo cero pero que va aumentando hasta llegar al valor siete y vuelve a valer cero. Esta señal aumenta de valor cada ver que la variable "counter" llega a cero. Esta variable empieza valiendo 10000 y en cada flanco de subida de la señal "CLK" que representa el reloj esta variable se reduce en una unidad y cuando llega a cero vuelve a pasar a valer 10000.

La variable de entrada "selector" informa de qué se debe mostrar,

- si vale "000" no se muestra nada
- si vale "001" se muestra el mensaje "start" en la pantalla
- si vale "010" se muestra el mensaje "go"
- si vale "011" se muestra el mensaje "gameover"
- si vale "100" se muestra el mensaje "done"
- si vale "101" se el valor del numero de vidas que le queda al jugador, dibujando el valor de las señales de entrada "round_7seg_smaller" y "round_7seg_larger".

```
entity DISPLAY_CONTROLLER is
  Port (
    selector           : in  std_logic_vector(2 downto 0);
    round_7seg_smaller : in  std_logic_vector(6 downto 0);
    round_7seg_larger  : in  std_logic_vector(6 downto 0);
    CLK                : in std_logic;
    anode              : out std_logic_vector(7 downto 0) := "11111111";
    segments           : out std_logic_vector(6 downto 0) := "1111111"
   );
end DISPLAY_CONTROLLER;
```



Conclusiones e impresiones

El proceso de diseño y programación del juego ha seguido un desarrollo lineal partiendo de una base sencilla que consistía en introducir una secuencia de botones preestablecida. Una vez obtenido un primer proyecto funcional de tal forma que el juego iniciase correctamente y concluyese tanto al introducir la secuencia de test como una secuencia distinta, se proponen diversas mejoras.

En primer lugar, se introduce la mejora del visualizar, estableciendo una comunicación máquina-usuario, con la que se permite al jugador saber en qué estado del juego se encuentra, con mensajes como "START", "GO", "DONE" y "GAME OVER".

Posteriormente, se añade la mejora aleatoriedad de la secuencia a adivinar, eliminando el código de test y dificultando el juego. Puesto que el nivel de dificultad aumenta ahora que se desconoce el código, se añaden oportunidades al usuario para que pueda seguir jugando aun habiendo fallado en alguna parte de la secuencia. Se añaden un total de 5 intentos.

Para finalizar, debido a que la secuencia es distinta en cada partida y desconocida para el jugador, se añade como última mejora un conjunte de leds en la posición de los botones. De esta forma, al final de cada partida, tanto ganada como perdida, se mostrará la secuencia correcta mediante los leds.

Enlace al repositorio:

https://github.com/JPioGA/TRABAJOS_SED