# Memoria Final – Diseño Automático de Sistemas Fiables

MIEMBROS DEL EQUIPO:
ALEJANDRO CASTELLANO LAMELA
ELSA GORDILLO PEÑAS
LAURA GARCÍA CARDEÑA
MARÍA MARTÍN-TOLEDANO

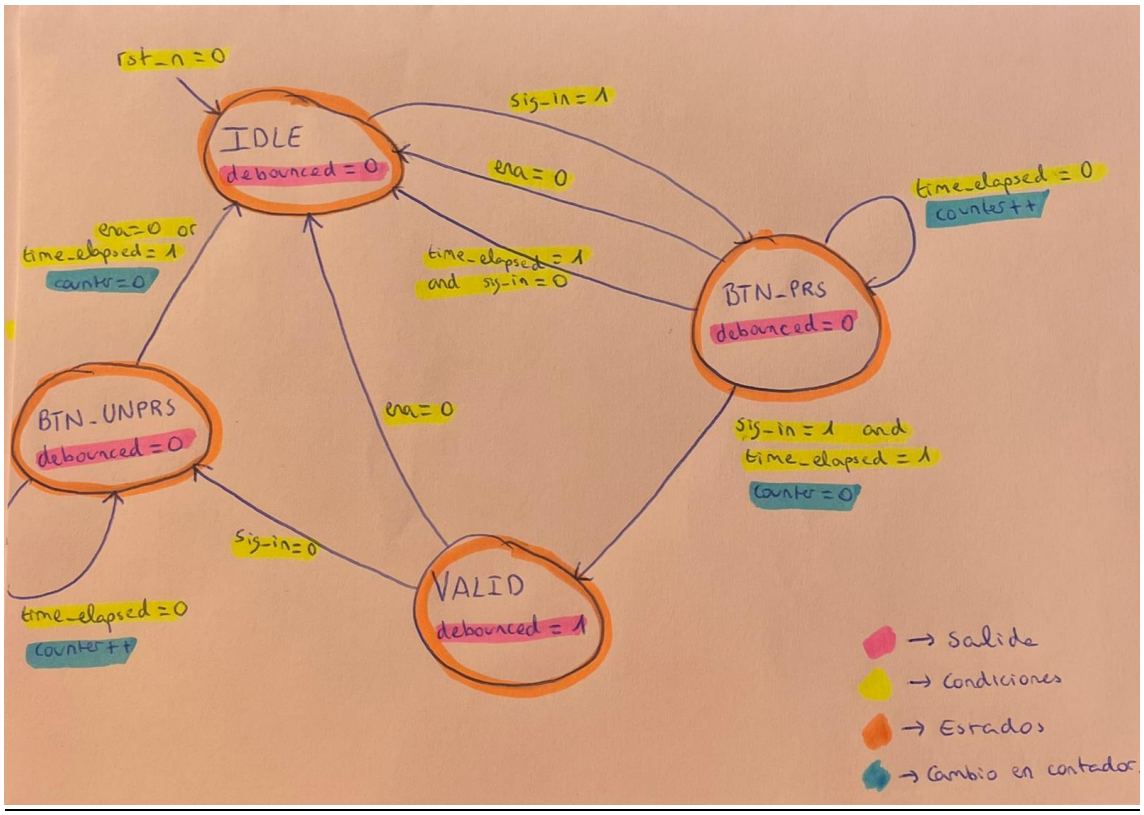
# Prácticas

# Práctica 1 –

En está práctica el objetivo es implementar un debouncer. Ya que la entrada externa generada por un botón, al pulsarlo pueda ser interpretada como una sucesión de pulsos.

**Debouncer.vhd**

Por ello para evitarlo se utiliza el antirrebote (debouncer), con el fin de transcurrido un tiempo razonable establecido mediante los genéricos del componente. Todo ello se implementará con la ayuda de una máquina de estados.



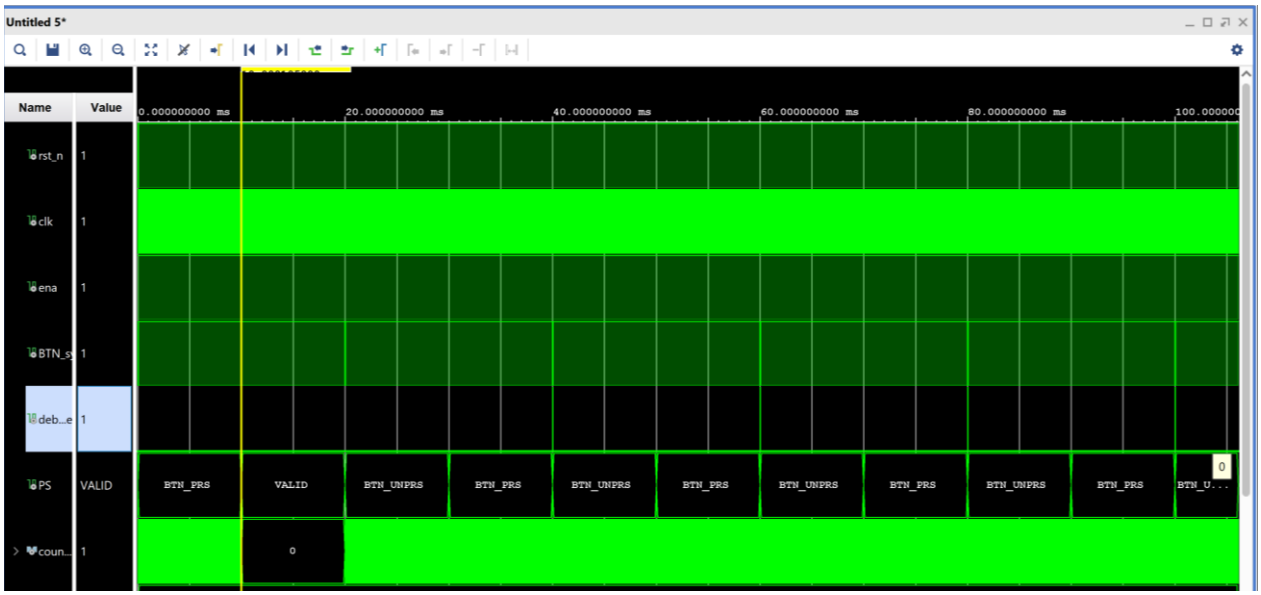

Cuando el contador llegue al máximo de ciclos establecido por c_cycles el tiempo razonable se cumplirá. Este contador solo se utiliza en el estado BTN_PRS y BTN_UNPRS.

El contador de ciclos lo realizamos de esta forma:

```
timer:process (clk, rst_n)
begin
-- ------------------------------------------------------------------------
-- Completar el timer que genera la señal de time_elapsed para trancionar en
-- las máquinas de estados
-- ------------------------------------------------------------------------

    if(rst_n= '0') then --Señal de reinicio
          counter <= (others  => '0');
          time_elapsed <= '0';
    elsif(rising_edge(clk)) then
        if(enable_count = '1')then

            if(counter < c_cycles)then
                counter <= counter + 1;

            else
                time_elapsed <= '1';
                counter <= (others  => '0');
            end if;

        else
            counter <= (others  => '0');
            time_elapsed <= '0';

        end if;
    end if;
end process;
```

El enable_count solo se activa cuando esta en los estados de botón pulsado o no pulsado.

Simulación del Debouncer.vhd:




Cuando parece que pasa del estado BTN_UNPRS al PRS está el estado IDLE entre los dos. El contador sería hasta 1.000.000 ciclos (100.000 KHz * 5ms).

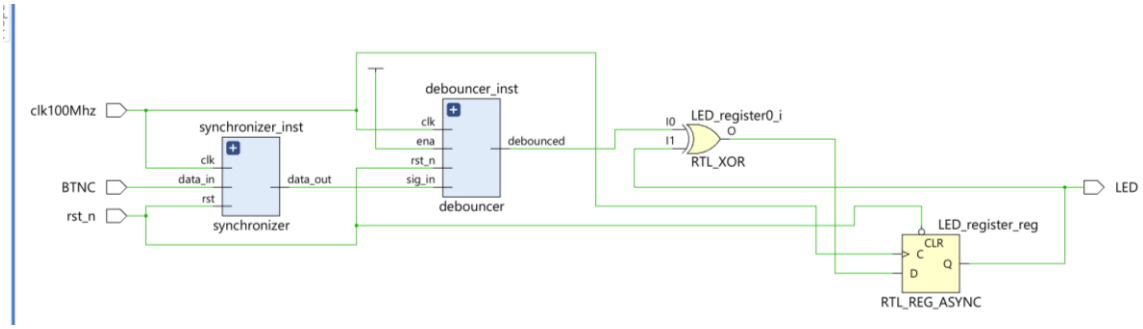
## Synchronizer.vhd

El fichero synchronizer.vhd no requiere de ninguna modificación se trata de un sincronizador como los explicados en clase. La salida de este (BTN_sync) es una señal síncrona con el sistema. Sin embargo, dicha salida fluctuará siguiendo los rebotes del pulsador y por tanto debe ser aplicado el circuito antirrebotes.

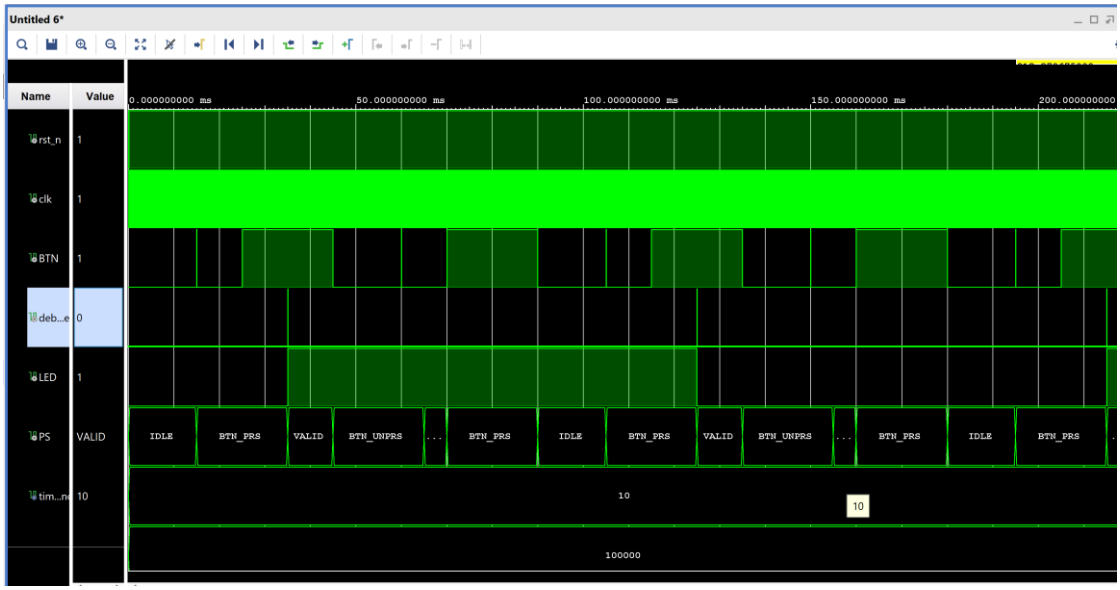
**Top_práctica1.vhd**

aplicado el circuito antirrebotes. Contiene los genéricos del sistema. Y unifica todo el sistema para establecer el resultado del LED. Dependiendo a su vez del Toggle_LED (resultado obtenido del debouncer) y del registro LED. Utilizamos una puerta xor, ya que se necesitaba que el led de salida cambiará de estado cuando Toggle_LED sea 1. Es decir si Led_register es uno, cuando el Toggle sea uno la salida sea 0 y lo mismo en el caso de el registro del led cuando sea 0.

```
begin
-- ------------------------------------------------
   state_LED <= Toggle_LED xor  LED_register;
```




**Simulación de top_práctica1.vhd:**

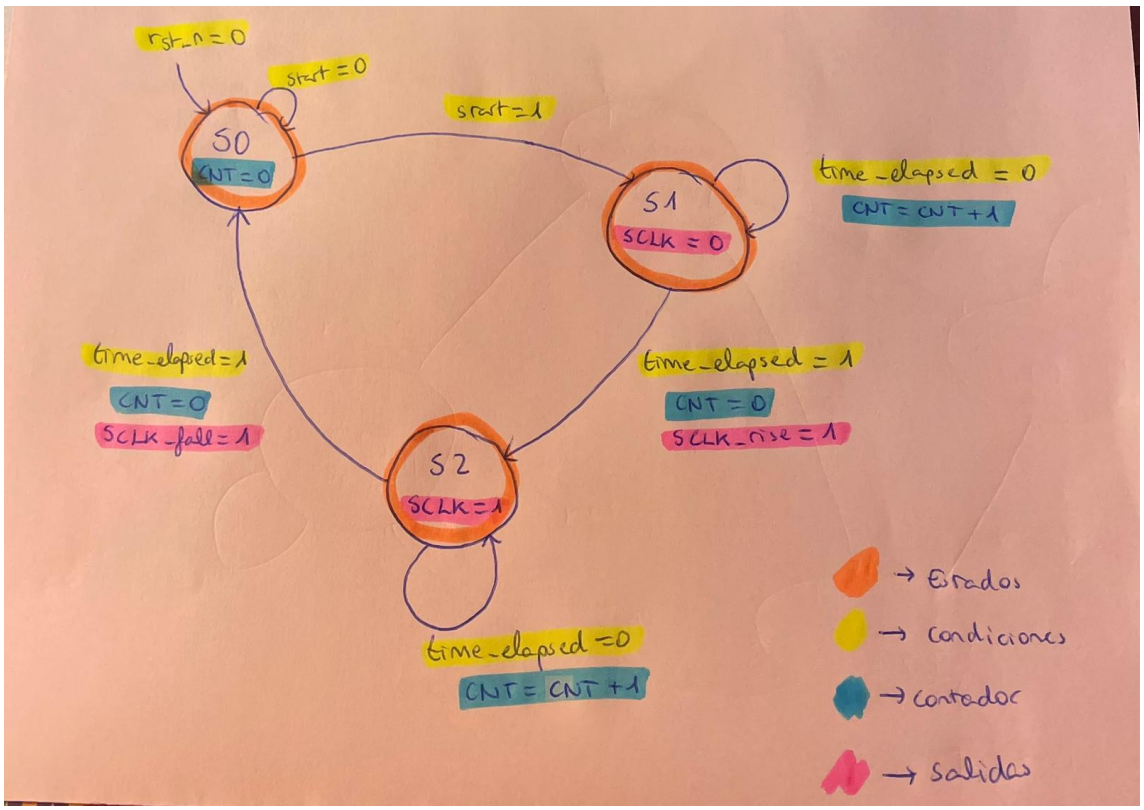


# Práctica 2 –

En está práctica implementaremos la interfaz PDM(Pulse Density Modulation) como almacenarla y después reproducirla. Para ello implementaremos un: Shift Register, un divisor de reloj, un controlador de memoria y un generador de PWM. En esta práctica solo nos centraremos en el registro de desplazamiento y en el generador de reloj.

**Fsm_sclk.vhd:**

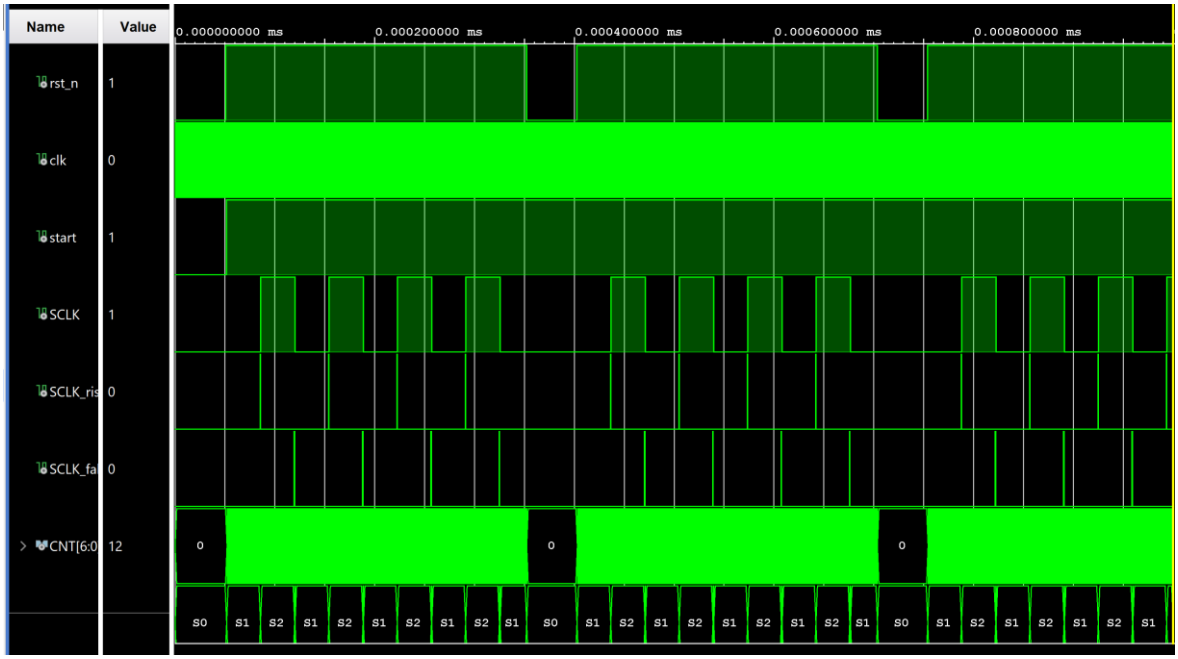
Para utilizar una señal de reloj derivada debemos implementar un contador para después compararlo con el número de ciclos deseado (periodo que queremos que dure). La máquina de estados tendrá salidas de Moore y otras de Mealy.



Contador de ciclos:

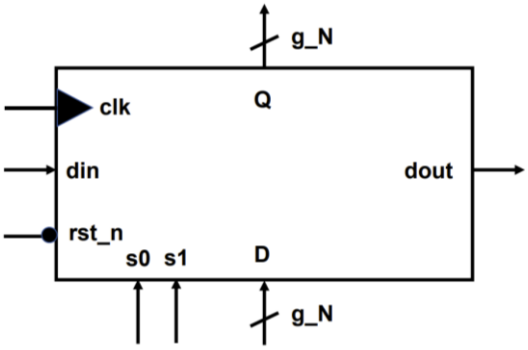
```
Contador:PROCESS (clk, rst_n)
BEGIN
    if(rst_n= '0') then --SeÃ±al de reinicio
        CNT <= (others  => '0');
        time_elapsed <= '0';

    elsif(rising_edge(clk)) then
        if(enable_count = '1')then
            if(CNT < c_half_T_SCLK)then
                CNT <= CNT + 1;

            else
                time_elapsed <= '1';
                CNT <= (others  => '0');
            end if;

        else
            CNT <= (others  => '0');
            time_elapsed <= '0';

        end if;
    end if;
END PROCESS;
```

**Simulación de fsm_sclk.vhd:**



**Shift_Register.vhd:**

El registro de desplazamiento tiene los siguientes puertos (entradas y salidas):




```
ENTITY ShiftRegister IS
    GENERIC (g_N : INTEGER := 5);
    PORT (
        d_in : IN STD_LOGIC;    -- Valor de entrada que se añadira al registro
        D : IN STD_LOGIC_VECTOR (g_N - 1 DOWNTO 0);     -- Entrada de datos para la carga en paralelo
        rst_n : in STD_LOGIC;    --Reset
        s0 : IN STD_LOGIC;       --Selector0  junto con S1 serán los que decidan que operación realizar
        S1 : IN STD_LOGIC;       --Selector1
        clk : IN STD_LOGIC;         --Señal de reloj
        Q : OUT STD_LOGIC_VECTOR (g_N - 1 DOWNTO 0);  --Valor en tiempo real del registro interno
        d_out : OUT STD_LOGIC);    --Valor de salida del registro
END ShiftRegister;
```

Proceso que funciona como máquina de estados para decidir qué acción realizar.

```
PROCESS (clk, rst_n, S0,S1)BEGIN

        IF rst_n = '0' THEN
            internal_value <= (OTHERS => '0');

        end if;
        IF rising_edge(clk) THEN
            -- No hay cambios
            IF s0 = '0' AND s1 = '0' THEN
                internal_value<= internal_value;
            -- Rotar a la derecha
            ELSIF S0 = '0' AND S1 = '1' THEN --right
            internal_value <=  d_in &internal_value(g_N - 1 downto 1) ;

            -- Rotar a la izquierda
            ELSIF S0 = '1' AND S1 = '0' THEN --left

                internal_value <= internal_value(g_N - 2 downto 0) & d_in;

            -- Carga paralelo
            ELSIF S0 = '1' AND S1 = '1' THEN
                internal_value <= D;

            END IF;
```

Diseño del componente:

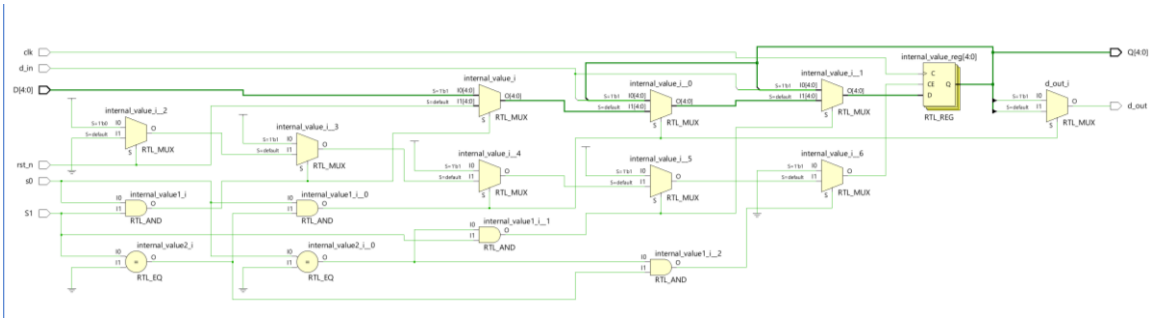
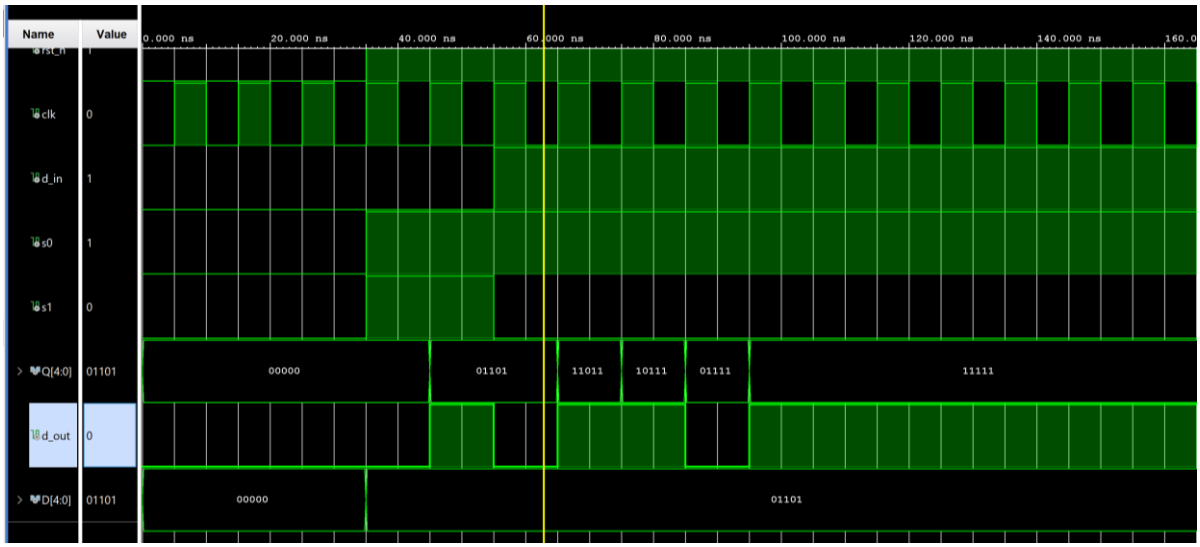


Tabla de la selección de acción:

| S0 | S1 | Acción |
|---|---|---|
| 0 | 0 | Sin cambios |
| 0 | 1 | Rotar hacia la derecha |
| 1 | 0 | Rotar hacia la izquierda |
| 1 | 1 | Carga paralelo |

**Simulación de Shift_register.vhd:**



# Práctica 3 –

En esta práctica se aplicará simulación avanzada, es decir, realizaremos lectura y escritura de archivos de texto, utilizaremos test automáticos y verificación de los resultados.

Se utilizará el fichero de texto inputs.csv para generar las entradas y salidas esperadas en el sistema de la práctica 1 y el fichero outputs.csv para escribir los resultados. En el testbench se implementarán dos procesos, uno para leer y escribir los valores de salida y otro para generar el reloj.

Primero declaramos los diferentes elementos para leer el fichero inputs.csv y para el fichero donde escribiremos los resultados. Y también crearemos las variables donde guardaremos las entradas y salidas del componente a comprobar.

```
        );
    clk <= not clk after clk_period/2;
    proc_sequencer : process
    -- Process to read the data
        file text_file :text open read_mode is "inputs.csv";
        variable text_line : line; -- Current line
        variable ok: boolean; -- Saves the status of the operation of reading

        --File para el fichero output
        file file_output : text;
        variable file_line : line;

        variable char : character; -- Read each character of the line(used when using comments)
        variable delay: time ; -- Saves the desired delay time

        variable rst_nVar: std_logic;
        variable BTNCVar: std_logic;
        variable LEDVar: std_logic;
```

Declaramos la ubicación donde estará el fichero de salida de resultados, en el caso de que no exista se creará uno en esa ubicación. Y a continuación leeremos los datos de input.csv y los guardaremos en las variables de entrada del componente a simular.

```
begin

    file_open(file_output,"C:\Users\jfgor\Vivado\Práctica1_vhdl_VIVADO\outputs.csv",write_mode);
    write(file_line, string'("Simulation of top_practica1.vhd"));
    writeline(file_output,file_line);
    while not endfile(text_file) loop

        readline(text_file, text_line);
        -- Skip empty lines and commented lines
        if text_line.all'length = 0 or text_line.all(1) = '#' then
            next;
        end if;
        -- Read the delay time
        read(text_line, delay, ok);
        assert ok
            report "Read 'delay' failed for line: " & text_line.all
            severity failure;
        -- Read first operand (rst_n)
        read(text_line, rst_nVar, ok);
        assert ok
            report "Read 'rst_n' failed for line: " & text_line.all
            severity failure;
        rst_n<= rst_nVar;
        -- Read the second operand (BTNC)
```

```
        rst_n<= rst_nVar;
        -- Read the second operand (BTNC)
        read(text_line, BTNCVar, ok);
        assert ok
            report "Read 'BTNC' failed for line: " & text_line.all
            severity failure;
        BTNC <= BTNCVar;
        -- Read the third operand (LED)
        read(text_line, LEDVar, ok);
        assert ok
            report "Read 'LED' failed for line: " & text_line.all
            severity failure;

        -- Wait for the delay
```

Después se escribirán los datos sacados de inputs.csv y los escribiremos en outputs.csv.

```
        writeline(file_output,file_line);
        write(file_line,string'(" Time: "));
        write(file_line,delay);
        write(file_line,string'("; rst_n: "));
        write(file_line,rst_n);
        write(file_line,string'("; BTNC: "));
        write(file_line,BTNC);
        write(file_line,string'("; LED: "));
        write(file_line, LED);
        write(file_line,string'(" ;"));
        writeline(file_output,file_line);
```

Por último, comprobaremos si la salida del componente simulado es igual a la salida esperada en inputs.csv. Si no coinciden se escribirá con el mensaje error con la razón y diferencia de los valores de LED.

```
if LED = LEDVar then
    writeline(file_output,file_line);
    write(file_line,string'("Correct value of LED: "));
    write(file_line,LED);
    write(file_line,string'(" ;"));
    writeline(file_output,file_line);
else
    writeline(file_output,file_line);
    write(file_line,string'("ERROR: expected LED to be "));
    write(file_line,LEDVar);
    write(file_line,string'(" actual value "));
    write(file_line,LED);
    write(file_line,string'(" ;"));

    writeline(file_output,file_line);
```

**Saldría de esta forma en outputs.csv:**

| | | | |
|---|---|---|---|
| Simulation of top_practica1.vhd | | | |
| Time: 10 ns | rst_n: 0 | BTNC: 0 | LED: 0 |
| Correct value of LED: 0 | | | |
| Time: 10 ns | rst_n: 1 | BTNC: 0 | LED: 0 |
| Correct value of LED: 0 | | | |
| Time: 10000 | rst_n: 1 | BTNC: 1 | LED: 0 |
| ERROR: expected LED to be 1 actual value 0 | | | |

| | | | |
|---|---|---|---|
| Time: 10000 | rst_n: 1 | BTNC: 0 | LED: 0 |
| ERROR: expected LED to be 1 actual value 0 | | | |
| Time: 50000 | rst_n: 1 | BTNC: 1 | LED: 1 |
| Correct value of LED: 1 | | | |
| Time: 10 ns | rst_n: 1 | BTNC: 0 | LED: 1 |
| Correct value of LED: 1 | | | |

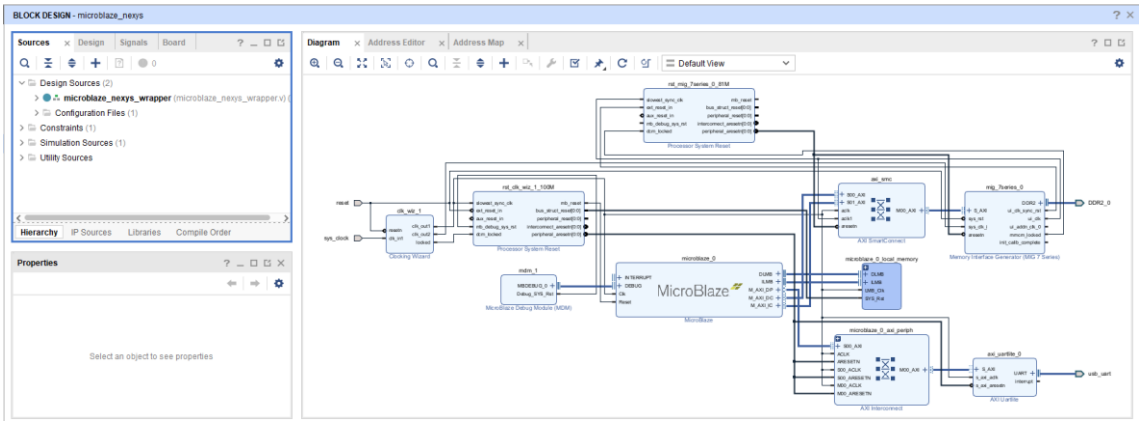
| | | | |
|---|---|---|---|
| Time: 10000 | rst_n: 1 | BTNC: 0 | LED: 1 |
| ERROR: expected LED to be 0 actual value 1 | | | |
| Time: 10 ns | rst_n: 0 | BTNC: 1 | LED: 0 |
| Correct value of LED: 0 | | | |
| Time: 10 ns | rst_n: 0 | BTNC: 1 | LED: 0 |
| Correct value of LED: 0 | | | |

| | | | |
|---|---|---|---|
| Time: 10 ns | rst_n: 0 | BTNC: 1 | LED: 0 |
| Correct value of LED: 0 | | | |
| Time: 10 ns | rst_n: 0 | BTNC: 1 | LED: 0 |
| Correct value of LED: 0 | | | |
| Finished simulation of top_practica1.vhd | | | |

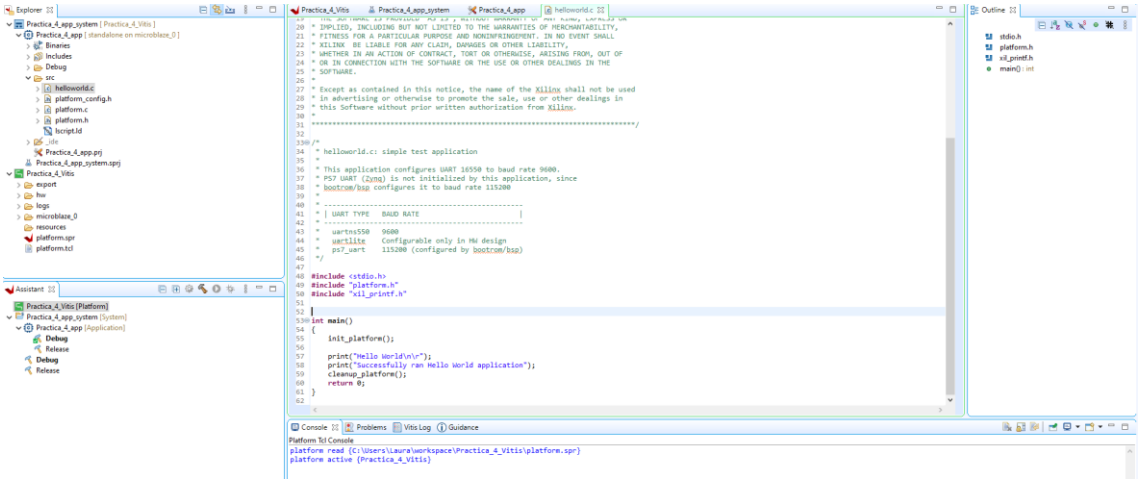
# Práctica 4 –

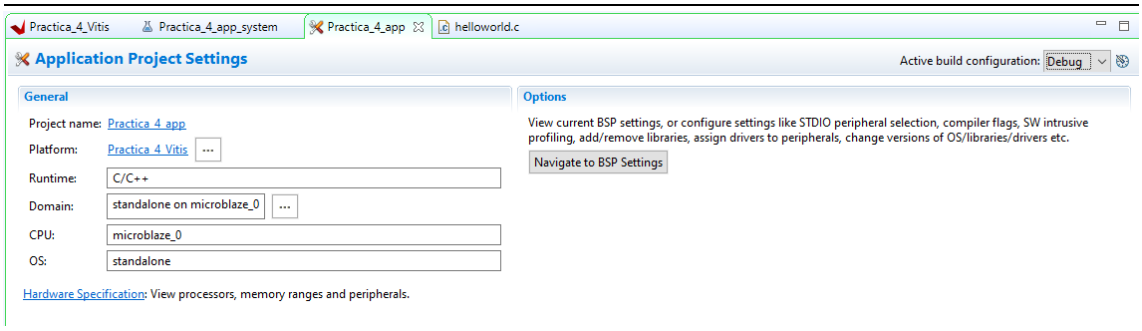
En esta práctica, tuvimos que descargar Vitis 2022 para poder realizarla, y se estuvo siguiendo el tutorial que se nos proporcionó en el campus. De esta forma, realizamos todos los pasos hasta llegar al punto 6, de tal forma que la práctica quedó así:

**Vivado:**




**Vitis:**

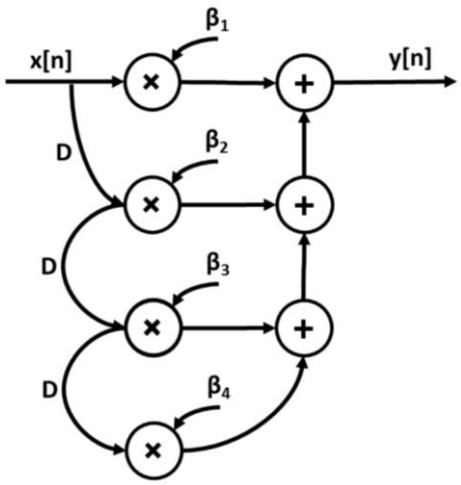



Al no disponer de una placa Hardware, no hemos podido realizar el paso 7, pero estamos dispuestos a hacerlo en la defensa si se pudiera.

# Práctica 5 –

En esta práctica se implementará un filtro digital. EN este caso será un filtro FIR que solo consta de sumas y multiplicaciones. Utilizaremos pipeline añadiendo varios registros para reducir el camino crítico, pero provocando un aumento de la latencia, la frecuencia y levemente el área.




$$y(n) = \beta 1\, x(n) + \beta 2\, x(n-1) + \beta 3\, x(n-2) + \beta 4\, x(n-3)$$

Primero declaramos todos los registros intermedios:

```
--y(n)= B1x(n) + B2x(n-1) + B3x(n-2) + B4x(n-3)
-- Multiplicacion -> ancho bits  es M +N = 8 + 8 = 16
--registros
signal B1x_reg:signed (15 downto 0);
signal B2x_reg:signed (15 downto 0);
signal B3x_reg:signed (15 downto 0);
signal B4x_reg:signed (15 downto 0);

--Operaciones mmultiplicacion
signal B1x:signed (15 downto 0);
signal B2x:signed (15 downto 0);
signal B3x:signed (15 downto 0);
signal B4x:signed (15 downto 0);

-- Suma -> ancho bits  es N+1 = 17+1 = 18
signal suma:signed(17 downto 0);

--Crear registro para x
signal xn_0:signed (7 downto 0);
signal xn_1:signed (7 downto 0);
signal xn_2:signed (7 downto 0);
signal xn_3:signed (7 downto 0);
```

```
6
7   --Crear registro para betas
8   signal B1:signed (7 downto 0);
9   signal B2:signed (7 downto 0);
0   signal B3:signed (7 downto 0);
1   signal B4:signed (7 downto 0);
2
```

Después en un proceso registraremos los diferentes valores de x durante los diferentes ciclos.

```
begin
--Registros para x(n),x(n-1),x(n-2),x(n-3)
RegistersX:PROCESS (clk,rst)BEGIN
        if(rst= '0')then
            xn_0<=(others=>'0');
            xn_1<=(others=>'0');
            xn_2<=(others=>'0');
            xn_3<=(others=>'0');

        elsif rising_edge(clk)then
            --x(n)
            xn_0<=signed(i_data);
            --x(n-1)
            xn_1<=xn_0;
            --x(n-2)
            xn_2<=xn_1;
            --x(n-3)
            xn_3<=xn_2;
        end if;
 end process;
```

Registramos también los valores de beta1, beta2, beta3 y beta4.

```
end process;
--Registros para beta1,2,3,4
RegistersBeta:PROCESS (clk,rst)BEGIN
       if(rst= '0')then
           B1<=(others=>'0');
           B2<=(others=>'0');
           B3<=(others=>'0');
           B4<=(others=>'0');

       elsif rising_edge(clk)then
           --beta1
           B1<=signed(beta1);
           --beta2
           B2<=signed(beta2);
           --beta3
           B3<=signed(beta3);
           --beta4
           B4<=signed(beta4);
       end if;

end process;
```

También registramos los valores de las cuatro multiplicaciones.

```
end process;
MultiplicaReg:PROCESS (clk,rst) BEGIN

        if(rst= '0')then
            B1x_reg<=(others=>'0');
            B2x_reg<=(others=>'0');
            B3x_reg<=(others=>'0');
            B4x_reg<=(others=>'0');

        elsif(rising_edge(clk))then

            B1x_reg<=B1x;
            B2x_reg<=B2x;
            B3x_reg<=B3x;
            B4x_reg<=B4x;
        end if;
  end process;
```

Se realizan las multiplicaciones cada vez que alguno de los valores de los registros de betas o x cambien.

```
Multiplica:PROCESS (xn_0,xn_1,xn_2,xn_3,B1,B2,B3,B4)BEGIN
          --beta1*x[n]
          B1x<=resize(xn_0 * B1,B1x'length);
          --beta2*x[n-1]
          B2x<=resize(xn_1 *B2,B2x'length);
          --beta3*x[n-2]
          B3x<=resize(xn_2*B3,B3x'length) ;
          --beta4*x[n-3]
          B4x<=resize(xn_3*B4,B4x'length) ;
end process;
```

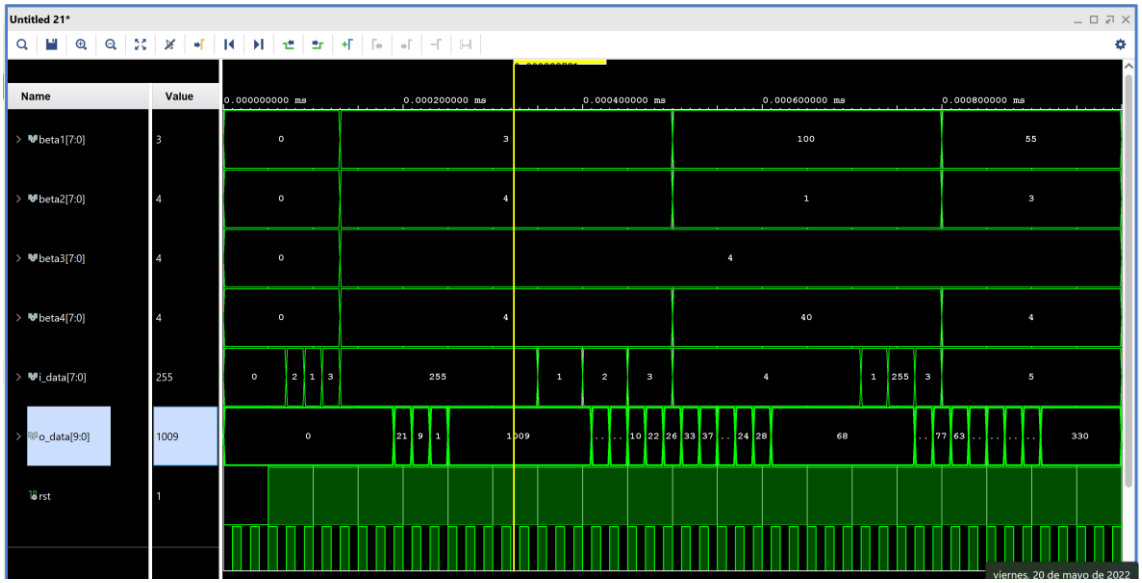
Por otro lado, se hará la suma cada vez que alguno de los registros de multiplicaciones cambie.

```
--beta1*x[n] + beta2*x[n-1] + beta3*x[n-2] + beta4*x[n-3]
Sumando:process(B1x_reg,B2x_reg,B3x_reg,B4x_reg)BEGIN

    suma<= resize(B1x_reg + B2x_reg,suma'length)  + resize(B3x_reg + B4x_reg,suma'length) ;

 end process;
```

Y por último se actualizará el valor de salida (o_data).

```
end process;
--register out
Salida:process(clk,rst,i_data,beta1,beta2,beta3,beta4)BEGIN
       if(rst= '0')then
          o_data <= (others=> '0');
      elsif(rising_edge(clk))then

          o_data <= std_logic_vector(resize(suma, o_data'length));
          end if;
 end process;
```

Simulación del filtro:




Quedando el diseño de esta forma: