Universidad Tecnológica Nacional

Técnicas Digitales IV

Trabajo Práctico 3-A: Comunicación I2C

Autores:
Ignacio Rodriguez Grandi
Matías Iván Bucca
Jonás Gabriel Ramírez Torres
Federico Elizondo

Supervisor:
Sergio Olmedo
Pablo Cayuela

June 28, 2021

Contents

1 Objetivo

Realizar la descripción de una comunicación I2C para poder controlar un display inteligente de 2 x 16 caracteres mediante la utilización de un dispositivo serie/paralelo controlado por un bus I2C. En el display se debe observar la cuenta de un cronómetro de 4 dígitos, con resolución de una décima de segundo.

2 Introducción

2.1 Protocolo I2C

2.1.1 Diseño

I2C es un puerto y protocolo de comunicación serial, define la trama de datos y las conexiones físicas para transferir bits entre 2 dispositivos digitales. El puerto incluye dos cables de comunicación, SDA y SCL. Además el protocolo permite conectar hasta 127 dispositivos SLAVEs con esas dos líneas, con hasta velocidades de 100, 400 y 1000 kbits/s. También es conocido como IIC ó TWI – Two Wire Interface.

La conexión de tantos dispositivos al mismo bus, es una de las principales ventajas. Además si comparamos a I2C con otro protocolo serial, como Serial TTL, este incluye más bits en su trama de comunicación que permite enviar mensajes más completos y detallados.

Los mensajes que se envían mediante un puerto I2C, incluye además del byte de información, una dirección tanto del registro como del sensor. Para la información que se envía siempre existe una confirmación de recepción por parte del dispositivo. Por esta razón es bueno diferenciar a los distintos elementos involucrados en este tipo de comunicación.

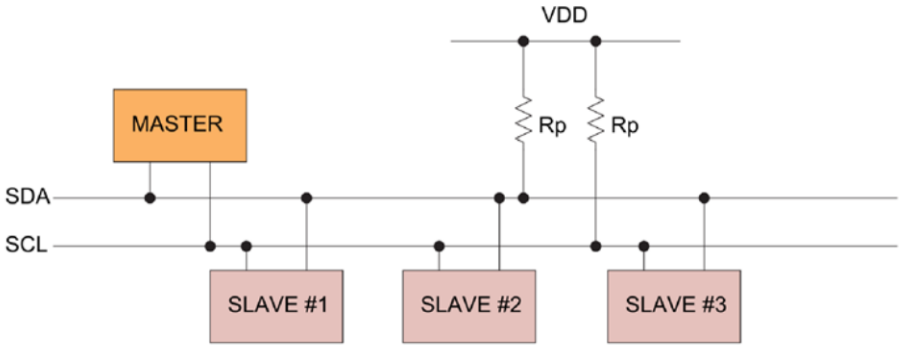

Figure 1: Esquema general de conexión para *I2C*.

2.1.2 Protocolo de transmisión

Los paquetes de datos I2C se organizan en bytes de 8 bits que comprenden la dirección del SLAVE, el número de registro y los datos que se van a transferir. La transmisión a través del bus es una operación de lectura o escritura. Los protocolos de lectura y escritura se basan en una serie de subprotocolos tales como condiciones de inicio y parada, bits de inicio repetidos, bytes de dirección, bits de transferencia de datos y bits de reconocimiento / no reconocimiento.

2.1.3 Condiciones de Start y Stop

Como sugiere el nombre, una condición de inicio siempre ocurre al inicio de una transmisión y es iniciada por el dispositivo MASTER. Esto se hace para despertar los dispositivos SLAVE inactivos en el bus. Esta es una de las dos ocasiones en que se permite que la línea SDA cambie de estado cuando la SCL es alta. Para indicar una condición de inicio, la línea SDA pasa del estado ALTO al estado BAJO, mientras que SCL es ALTO.

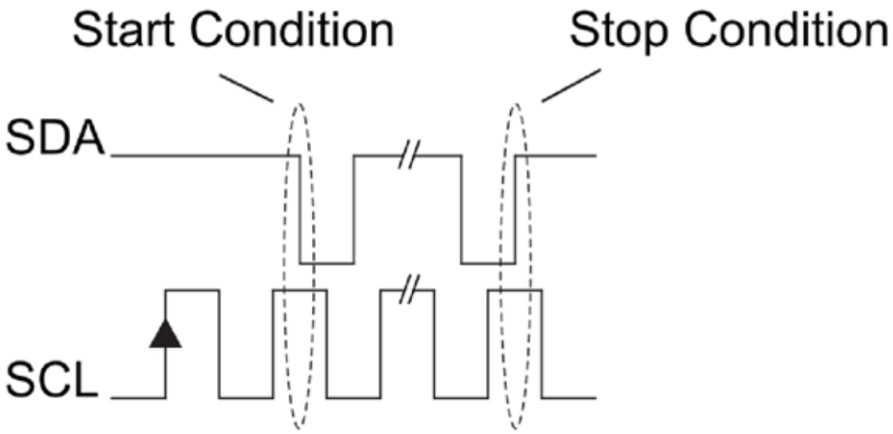

Figure 2: Esquema general de conexión para *I2C*.

Similar a una condición de inicio, la condición de parada ocurre al final de una transferencia de datos y también es generada por el MASTER. Esto significa el final de la transferencia y es un comando para decirle a los dispositivos SLAVE's que deben volver a un estado inactivo, liberar la línea SDA y no enviar más datos en el bus. Esta es la segunda instancia en la que se permite que la línea SDA cambie de estado cuando SCL es ALTA.

La línea SDA pasa del estado LOW a HIGH, mientras que SCL es HIGH, para indicar una condición de parada.

2.1.4 Dirección del Slave

Se envía una dirección de SLAVE en formato de bytes de 8 bits, MSB primero, pero el último bit significa si la transacción leerá o escribirá en el SLAVE. En efecto, los 7 bits superiores constituyen la dirección esclava, mientras que el 8º bit sirve como bit de comando READ / WRITE. Por tanto, existe un espacio de direcciones de 128 direcciones únicas para direccionar hasta 128 SLAVEs. Muchas veces.

2.1.5 Bits ACK/NACK

Como forma de retroalimentación, después de cada transmisión de bytes, el dispositivo receptor envía un bit de reconocimiento o no reconocimiento. El receptor genera un bit de reconocimiento al mantener baja la línea SDA durante un período de ALTA SCL, mientras que un bit de no reconocimiento se genera cuando el receptor deja la línea SDA pasivamente jalada a ALTO y no responde de ninguna manera. Este hecho implica que, en respuesta a un byte de dirección, todos los SLAVE no coincidentes envían un bit de no reconocimiento al no responder.

Un ACK se utiliza para indicar que un byte (dirección o datos) se transmitió y recibió con éxito y que la transmisión puede continuar con la siguiente transferencia de byte, una condición de parada o un inicio repetido. El receptor generalmente usa un NACK para indicar si ocurrió un error en algún lugar de la transmisión de datos. Esto se usa para indicar al dispositivo transmisor que termine la transmisión inmediatamente o para hacer otro intento enviando un inicio repetido.

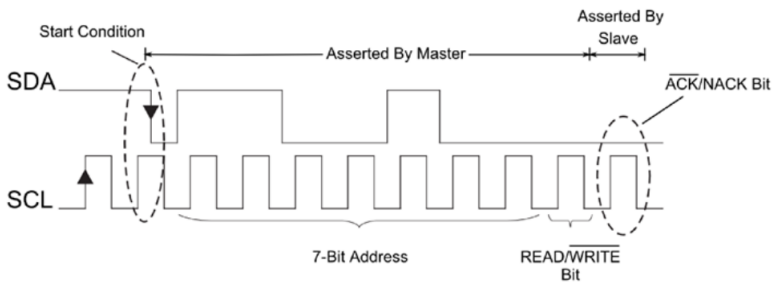

Figure 3: Condición de inicio seguida de la dirección del SLAVE y su respuesta correspondiente.

2.1.6 Enviando datos a un dispositivo

El primer bit enviado es el bit de INICIO que inicia las comunicaciones. El byte de dirección sigue a los pulsos de reloj subsiguientes. En este momento, todos los dispositivos SLAVE en el bus están escuchando su dirección de dispositivo, que constituye los primeros siete bits del byte de dirección. El dispositivo SLAVE que encuentra una coincidencia de dirección continúa escuchando el último bit (READ / WRITE bit) para entender si el MASTER quiere leer del SLAVE o escribir en él. Todos los demás dispositivos SLAVE ignoran la comunicación adicional enviando un NACK, que por definición no hace nada.

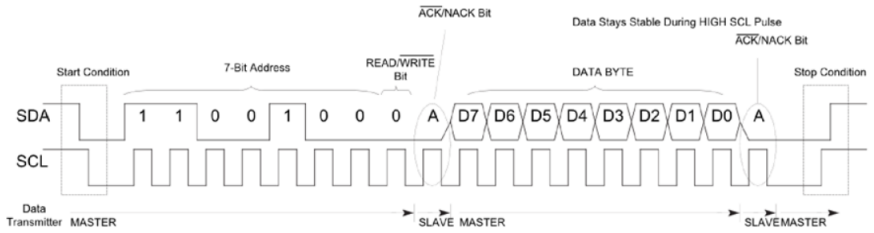

Figure 4: Transmisión de escritura exitosa.

En respuesta al reconocimiento de su dirección y el comando de escritura, el dispositivo direccionado responde enviando un bit de reconocimiento (ACK) como retroalimentación al MASTER de que un dispositivo SLAVE con la dirección correcta está presente en el bus y esperando más comunicación. El MASTER continúa con la transferencia de datos enviando datos en formato de bytes. Si el MASTER está escribiendo en un registro específico en el dispositivo SLAVE, entonces escribe el byte de comando correspondiente antes de enviar los datos. Después de cada transferencia de bytes, el SLAVE responde enviando un ACK. Una vez que el MASTER ha terminado de transferir todos los datos, finaliza la transferencia enviando una condición de PARO.

2.1.7 Recibiendo datos de un dispositivo

La transmisión es nuevamente iniciada por el MASTER con una condición de START después de la cual la dirección se transmite en pulsos de reloj subsiguientes. El dispositivo direccionado continúa escuchando la transmisión leyendo el bit READ / WRITE y responde con un bit de reconocimiento. Una vez que el SLAVE envía el acuse de recibo, asume el control de SDA y envía datos al MASTER. En respuesta a cada byte transmitido, el MASTER envía un bit de reconocimiento. Cuando el MASTER ya no quiere recibir datos, responde con un NACK después del último byte que desea recibir y luego reanuda el control del bus y envía una condición de STOP para finalizar la transmisión.

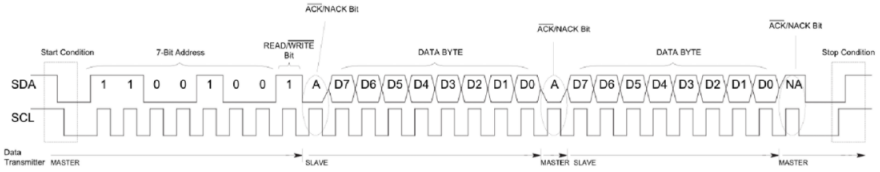

Figure 5: Transmisión de lectura exitosa.

3 Implementación

3.1 RTL

Con la implementación en *Quartus* se pueden observar la interconexión de los módulos en el RTL:

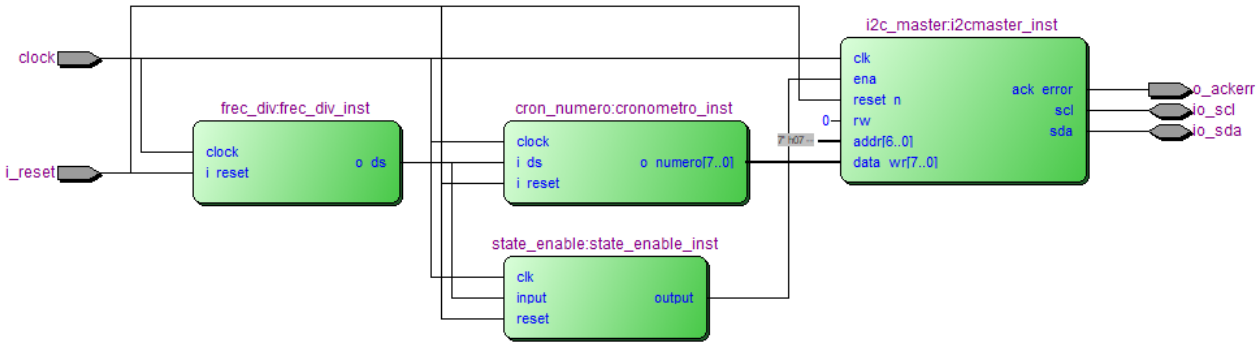

Figure 6: RTL.

3.2 Descripción en VHDL

3.2.1 Comunicación I2C

A continuación se presenta la descripción en *VHDL* del modo MASTER:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY i2c_master IS
  GENERIC(
    input_clk : INTEGER := 50_000_000; --input clock speed from user logic in Hz
    bus_clk   : INTEGER := 100_000);   --speed the i2c bus (scl) will run at in Hz
  PORT(
    clk       : IN     STD_LOGIC;                    --system clock
    reset_n   : IN     STD_LOGIC;                    --active low reset
    ena       : IN     STD_LOGIC;                    --latch in command
    addr      : IN     STD_LOGIC_VECTOR(6 DOWNTO 0); --address of target slave
    rw        : IN     STD_LOGIC;                    --'0' is write, '1' is read
    data_wr   : IN     STD_LOGIC_VECTOR(7 DOWNTO 0); --data to write to slave
    busy      : OUT    STD_LOGIC;                    --indicates transaction in progress
    data_rd   : OUT    STD_LOGIC_VECTOR(7 DOWNTO 0); --data read from slave
    ack_error : BUFFER STD_LOGIC;                    --flag if improper acknowledge from slave
    sda       : INOUT  STD_LOGIC;                    --serial data output of i2c bus
    scl       : INOUT  STD_LOGIC);                   --serial clock output of i2c bus
END i2c_master;

ARCHITECTURE logic OF i2c_master IS
```

```
  CONSTANT divider  :  INTEGER := (input_clk/bus_clk)/4; --number of clocks in 1/4 cycle of scl
  TYPE machine IS(ready, start, command, slv_ack1, wr, rd, slv_ack2, mstr_ack, stop); --needed states
  SIGNAL state         : machine;                        --state machine
  SIGNAL data_clk      : STD_LOGIC;                      --data clock for sda
  SIGNAL data_clk_prev : STD_LOGIC;                      --data clock during previous system clock
  SIGNAL scl_clk       : STD_LOGIC;                      --constantly running internal scl
  SIGNAL scl_ena       : STD_LOGIC := '0';               --enables internal scl to output
  SIGNAL sda_int       : STD_LOGIC := '1';               --internal sda
  SIGNAL sda_ena_n     : STD_LOGIC;                      --enables internal sda to output
  SIGNAL addr_rw       : STD_LOGIC_VECTOR(7 DOWNTO 0);   --latched in address and read/write
  SIGNAL data_tx       : STD_LOGIC_VECTOR(7 DOWNTO 0);   --latched in data to write to slave
  SIGNAL data_rx       : STD_LOGIC_VECTOR(7 DOWNTO 0);   --data received from slave
  SIGNAL bit_cnt       : INTEGER RANGE 0 TO 7 := 7;      --tracks bit number in transaction
  SIGNAL stretch       : STD_LOGIC := '0';               --identifies if slave is stretching scl
BEGIN

  --Generate the timing for the bus clock (scl_clk) and the data clock (data_clk)
  PROCESS(clk, reset_n)
    VARIABLE count  :  INTEGER RANGE 0 TO divider*4;  --timing for clock generation
  BEGIN
    IF(reset_n = '0') THEN                --reset asserted
      stretch <= '0';
      count := 0;
    ELSIF(clk'EVENT AND clk = '1') THEN
      data_clk_prev <= data_clk;          --store previous value of data clock
      IF(count = divider*4-1) THEN        --end of timing cycle
        count := 0;                       --reset timer
      ELSIF(stretch = '0') THEN           --clock stretching from slave not detected
        count := count + 1;               --continue clock generation timing
      END IF;
      CASE count IS
        WHEN 0 TO divider-1 =>            --first 1/4 cycle of clocking
          scl_clk <= '0';
          data_clk <= '0';
        WHEN divider TO divider*2-1 =>    --second 1/4 cycle of clocking
          scl_clk <= '0';
          data_clk <= '1';
        WHEN divider*2 TO divider*3-1 =>  --third 1/4 cycle of clocking
          scl_clk <= '1';                 --release scl
          IF(scl = '0') THEN              --detect if slave is stretching clock
            stretch <= '1';
          ELSE
            stretch <= '0';
          END IF;
          data_clk <= '1';
        WHEN OTHERS =>                    --last 1/4 cycle of clocking
          scl_clk <= '1';
          data_clk <= '0';
      END CASE;
    END IF;
  END PROCESS;

  --state machine and writing to sda during scl low (data_clk rising edge)
  PROCESS(clk, reset_n)
  BEGIN
    IF(reset_n = '0') THEN                --reset asserted
```

```
      state <= ready;                       --return to initial state
      busy <= '1';                          --indicate not available
      scl_ena <= '0';                       --sets scl high impedance
      sda_int <= '1';                       --sets sda high impedance
      ack_error <= '0';                     --clear acknowledge error flag
      bit_cnt <= 7;                         --restarts data bit counter
      data_rd <= "00000000";                --clear data read port
    ELSIF(clk'EVENT AND clk = '1') THEN
      IF(data_clk = '1' AND data_clk_prev = '0') THEN  --data clock rising edge
        CASE state IS
          WHEN ready =>                     --idle state
            IF(ena = '1') THEN              --transaction requested
              busy <= '1';                  --flag busy
              addr_rw <= addr & rw;         --collect requested slave address and command
              data_tx <= data_wr;           --collect requested data to write
              state <= start;               --go to start bit
            ELSE                            --remain idle
              busy <= '0';                  --unflag busy
              state <= ready;               --remain idle
            END IF;
          WHEN start =>                     --start bit of transaction
            busy <= '1';                    --resume busy if continuous mode
            sda_int <= addr_rw(bit_cnt);    --set first address bit to bus
            state <= command;               --go to command
          WHEN command =>                   --address and command byte of transaction
            IF(bit_cnt = 0) THEN            --command transmit finished
              sda_int <= '1';               --release sda for slave acknowledge
              bit_cnt <= 7;                 --reset bit counter for "byte" states
              state <= slv_ack1;            --go to slave acknowledge (command)
            ELSE                            --next clock cycle of command state
              bit_cnt <= bit_cnt - 1;       --keep track of transaction bits
              sda_int <= addr_rw(bit_cnt-1); --write address/command bit to bus
              state <= command;             --continue with command
            END IF;
          WHEN slv_ack1 =>                  --slave acknowledge bit (command)
            IF(addr_rw(0) = '0') THEN       --write command
              sda_int <= data_tx(bit_cnt);  --write first bit of data
              state <= wr;                  --go to write byte
            ELSE                            --read command
              sda_int <= '1';               --release sda from incoming data
              state <= rd;                  --go to read byte
            END IF;
          WHEN wr =>                        --write byte of transaction
            busy <= '1';                    --resume busy if continuous mode
            IF(bit_cnt = 0) THEN            --write byte transmit finished
              sda_int <= '1';               --release sda for slave acknowledge
              bit_cnt <= 7;                 --reset bit counter for "byte" states
              state <= slv_ack2;            --go to slave acknowledge (write)
            ELSE                            --next clock cycle of write state
              bit_cnt <= bit_cnt - 1;       --keep track of transaction bits
              sda_int <= data_tx(bit_cnt-1); --write next bit to bus
              state <= wr;                  --continue writing
            END IF;
          WHEN rd =>                        --read byte of transaction
            busy <= '1';                    --resume busy if continuous mode
            IF(bit_cnt = 0) THEN            --read byte receive finished
```

```
        IF(ena = '1' AND addr_rw = addr & rw) THEN  --continuing with another read at same address
          sda_int <= '0';               --acknowledge the byte has been received
        ELSE                            --stopping or continuing with a write
          sda_int <= '1';               --send a no-acknowledge (before stop or repeated start)
        END IF;
        bit_cnt <= 7;                   --reset bit counter for "byte" states
        data_rd <= data_rx;             --output received data
        state <= mstr_ack;              --go to master acknowledge
      ELSE                              --next clock cycle of read state
        bit_cnt <= bit_cnt - 1;         --keep track of transaction bits
        state <= rd;                    --continue reading
      END IF;
    WHEN slv_ack2 =>                    --slave acknowledge bit (write)
      IF(ena = '1') THEN                --continue transaction
        busy <= '0';                    --continue is accepted
        addr_rw <= addr & rw;           --collect requested slave address and command
        data_tx <= data_wr;             --collect requested data to write
        IF(addr_rw = addr & rw) THEN    --continue transaction with another write
          sda_int <= data_wr(bit_cnt); --write first bit of data
          state <= wr;                  --go to write byte
        ELSE                            --continue transaction with a read or new slave
          state <= start;               --go to repeated start
        END IF;
      ELSE                              --complete transaction
        state <= stop;                  --go to stop bit
      END IF;
    WHEN mstr_ack =>                    --master acknowledge bit after a read
      IF(ena = '1') THEN                --continue transaction
        busy <= '0';                    --continue is accepted and data received is available on bus
        addr_rw <= addr & rw;           --collect requested slave address and command
        data_tx <= data_wr;             --collect requested data to write
        IF(addr_rw = addr & rw) THEN    --continue transaction with another read
          sda_int <= '1';               --release sda from incoming data
          state <= rd;                  --go to read byte
        ELSE                            --continue transaction with a write or new slave
          state <= start;               --repeated start
        END IF;
      ELSE                              --complete transaction
        state <= stop;                  --go to stop bit
      END IF;
    WHEN stop =>                        --stop bit of transaction
      busy <= '0';                      --unflag busy
      state <= ready;                   --go to idle state
  END CASE;
ELSIF(data_clk = '0' AND data_clk_prev = '1') THEN  --data clock falling edge
  CASE state IS
    WHEN start =>
      IF(scl_ena = '0') THEN                --starting new transaction
        scl_ena <= '1';                     --enable scl output
        ack_error <= '0';                   --reset acknowledge error output
      END IF;
    WHEN slv_ack1 =>                        --receiving slave acknowledge (command)
      IF(sda /= '0' OR ack_error = '1') THEN  --no-acknowledge or previous no-acknowledge
        ack_error <= '1';                   --set error output if no-acknowledge
      END IF;
    WHEN rd =>                              --receiving slave data
```

```
          data_rx(bit_cnt) <= sda;                --receive current slave data bit
        WHEN slv_ack2 =>                          --receiving slave acknowledge (write)
          IF(sda /= '0' OR ack_error = '1') THEN  --no-acknowledge or previous no-acknowledge
            ack_error <= '1';                     --set error output if no-acknowledge
          END IF;
        WHEN stop =>
          scl_ena <= '0';                         --disable scl
        WHEN OTHERS =>
          NULL;
      END CASE;
    END IF;
  END IF;
END PROCESS;

--set sda output
WITH state SELECT
  sda_ena_n <= data_clk_prev WHEN start,     --generate start condition
               NOT data_clk_prev WHEN stop,  --generate stop condition
               sda_int WHEN OTHERS;          --set to internal sda signal

--set scl and sda outputs
scl <= '0' WHEN (scl_ena = '1' AND scl_clk = '0') ELSE '1';
sda <= '0' WHEN sda_ena_n = '0' ELSE '1';

END logic;
```

3.2.2 Cronometro

A continuación se presenta la descripción en *VHDL* del cronometro, la salida de este bloque alternará entre la cuenta de décimas de segundo, segundos y minutos:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cron_numero is
    Port ( clock   : in  STD_LOGIC;
                         i_reset : in  STD_LOGIC;
                         i_ds    : in  STD_LOGIC;
                         o_numero: out std_logic_vector (7 downto 0));
end cron_numero;

architecture Behavioral of cron_numero is
signal r_deci : STD_LOGIC_VECTOR (7 downto 0);
signal r_seg  : STD_LOGIC_VECTOR (7 downto 0);
signal r_min  : STD_LOGIC_VECTOR (7 downto 0);
signal i_sel  : STD_LOGIC_VECTOR (1 downto 0);
signal s_mux : integer range 1000 downto 0;
signal s_inc_s: STD_LOGIC;
signal s_inc_m: STD_LOGIC;
begin
--Decimas de segundo
process (clock ,i_reset)
begin
 if i_reset = '0' then
        r_deci <= "00000000";
 elsif clock='1' and clock'event then
                if i_ds = '1'  then
                        if r_deci = "0001001" then
                                r_deci  <= "00000000";
                                s_inc_s <= '1';
                        else
                                r_deci  <= r_deci + 1;
                                s_inc_s <= '0';
                        end if;
                end if;
 end if;
end process;
--segundos
process (clock,i_reset)
begin
 if i_reset = '0' then
        r_seg <= "00000000";
 elsif clock='1' and clock'event then
                if s_inc_s = '1' then
                        if r_seg = "00111100" then
                                r_seg   <= "00000000";
                                s_inc_m <= '1';
                        else
                                r_seg  <= r_seg + 1;
                                s_inc_m <= '0';
```

```
                        end if;
                end if;
 end if;
end process;
--minutos
process (clock,i_reset)
begin
 if i_reset = '0' then
        r_min <= "00000000";
 elsif clock='1' and clock'event then
                if s_inc_m = '1' then
                        if r_min = "00111100" then
                                r_min  <= "00000000";
                        else
                                r_min  <= r_min + 1;
                        end if;
                end if;
 end if;
end process;
--multiplexado de salida
process (clock,i_reset)
begin
 if i_reset = '0' then
        s_mux <= 0;
        i_sel <= "00";
 elsif clock='1' and clock'event then
        if s_mux = 1000 then
                s_mux <= 0;
                if i_sel= "11" then
                        i_sel<= "00";
                else
                        i_sel<=i_sel + 1;
                end if;
        else
                s_mux  <= s_mux + 1;
        end if;
end if;
end process;



o_numero  <=   r_deci   when i_sel="00" else
                                  r_seg    when i_sel="01" else
                                    r_min    when i_sel="10" else
                                       "00000000";
end Behavioral;
```

3.2.3 Divisor de frecuencia

A continuación se presenta la descripción en *VHDL* del divisor de frecuencia, este se encarga de generar un pulso a su salida por cada décima de segundo:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity frec_div is
    Port (
                            clock      : in  STD_LOGIC;
                            i_reset    : in STD_LOGIC;
                            o_ds       : out STD_LOGIC);
end frec_div;

architecture Behavioral of frec_div is
signal s_cuenta_int : integer range 50000000 downto 0;
begin
process (clock,i_reset)
begin
 if i_reset = '0' then
        s_cuenta_int <= 0;
 elsif clock='1' and clock'event then
                if s_cuenta_int = 5000000 then
                        s_cuenta_int <= 0;
                        o_ds  <= '1';
                else
                        s_cuenta_int <= s_cuenta_int + 1;
                        o_ds  <= '0';
                end if;
 end if;
end process;
end Behavioral;
```

3.2.4 Generador de Enable

A continuación se presenta la descripción en *VHDL* del generador enable, la señal generada por este se conectará al módulo *i2c master* para habilitar la transmisión:

```
library ieee;
use ieee.std_logic_1164.all;

entity state_enable is

        port(
                clk                 : in        std_logic;
                input          : in        std_logic;
                reset          : in        std_logic;
                output          : out        std_logic
        );

end entity;


architecture rtl of state_enable is
signal s_cuenta_int : integer range 10000 downto 0;
signal enable_int: std_logic;
begin
process (clk,reset)
    begin
         if reset = '0' then
                s_cuenta_int <= 0;
         elsif clk='1' and clk'event and enable_int ='1' then
                        if s_cuenta_int = 10000 then
                                s_cuenta_int <= 0;
                        else
                                s_cuenta_int <= s_cuenta_int + 1;
                        end if;
         end if;
end process;

process (clk,reset)
    begin
         if reset = '0' then
                enable_int <= '0';
         elsif clk='1' and clk' event then
                        if input = '1' then
                                enable_int   <= '1';
                        elsif s_cuenta_int = 10000 then
                                enable_int   <= '0';
                        end if;
         end if;
end process;

output <= enable_int;


end rtl;
```

3.2.5 Top

A continuación se presenta la descripción en *VHDL* del Top:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

entity top is
GENERIC(
    --input clock speed from user logic in Hz
    input_clk : INTEGER := 50_000_000;
    --speed the i2c bus (scl) will run at in Hz
    bus_clk   : INTEGER := 100_000);
port(
                clock     : in std_LOGIC;
                i_reset   : in std_LOGIC;
                o_ackerr  : out std_LOGIC;
      io_sda    : inout  STD_LOGIC;
      io_scl    : inout  STD_LOGIC
          );
end top;

architecture Behavioral of top is

COMPONENT i2c_master is
GENERIC(
    input_clk : INTEGER := 50_000_000; --input clock speed from user logic in Hz
    bus_clk   : INTEGER := 100_000);   --speed the i2c bus (scl) will run at in Hz
PORT(
    clk       : in     STD_LOGIC;                    --system clock
    reset_n   : in     STD_LOGIC;                    --active low reset
    ena       : in     STD_LOGIC;                    --latch in command
    addr      : in     STD_LOGIC_VECTOR(6 DOWNTO 0); --address of target slave
    rw        : in     STD_LOGIC;                    --'0' is write, '1' is read
    data_wr   : in     STD_LOGIC_VECTOR(7 DOWNTO 0); --data to write to slave
    busy      : out    STD_LOGIC;                    --indicates transaction in progress
    data_rd   : out    STD_LOGIC_VECTOR(7 DOWNTO 0); --data read from slave
    ack_error : buffer STD_LOGIC;                    --flag if improper acknowledge from slave
    sda       : inout  STD_LOGIC;                    --serial data output of i2c bus
    scl       : inout  STD_LOGIC);                   --serial clock output of i2c bus
END COMPONENT;

COMPONENT frec_div is
PORT(
         clock      : in  STD_LOGIC;
         i_reset    : in  STD_LOGIC;
         o_ds       : out STD_LOGIC);
END COMPONENT;

COMPONENT state_enable is
        port(
                clk                 : in        std_logic;
                input         : in        std_logic;
                reset         : in        std_logic;
                output         : out        std_logic
```

```
        );
END COMPONENT;

COMPONENT cron_numero is
 Port ( clock   : in  STD_LOGIC;
                  i_reset : in  STD_LOGIC;
                  i_ds    : in  STD_LOGIC;
                  o_numero: out std_logic_vector (7 downto 0));
END COMPONENT;




signal r_enable : std_logic;
signal s_ds  : std_logic;
signal s_busy: std_logic;
signal r_busy: std_logic;
signal r_ack_error: std_logic;
signal s_data_wr  : std_logic_vector (7 downto 0);
signal r_data_rd  : std_logic_vector (7 downto 0);
signal r_sel : std_logic_vector (1 downto 0);

begin

i2cmaster_inst: i2c_master
  generic map (
    input_clk => input_clk,
    bus_clk   => bus_clk
  )
  PORT MAP (
    clk       => clock,
    reset_n   => i_reset,
    ena       => r_enable,
    addr      => "0000111",
    rw        => '0',
    data_wr   => s_data_wr,
    busy      => s_busy,
    data_rd   => r_data_rd,
    ack_error => o_ackerr,
    sda       => io_sda,
    scl       => io_scl
  );
frec_div_inst: frec_div
 PORT MAP (
    clock   => clock,
    i_reset => i_reset,
    o_ds    => s_ds
  );

state_enable_inst: state_enable
 PORT MAP (
    clk   => clock,
    reset => i_reset,
    input => s_ds,
         output=> r_enable
  );
```

```
    cronometro_inst: cron_numero
    PORT MAP(
           clock  => clock,
      i_reset => i_reset,
      i_ds    => s_ds,
      o_numero => s_data_wr
    );


end Behavioral;
```

3.3 Simulación del comportamiento

Puede observarse la transmisión del protocolo I2C generada por la descripción:

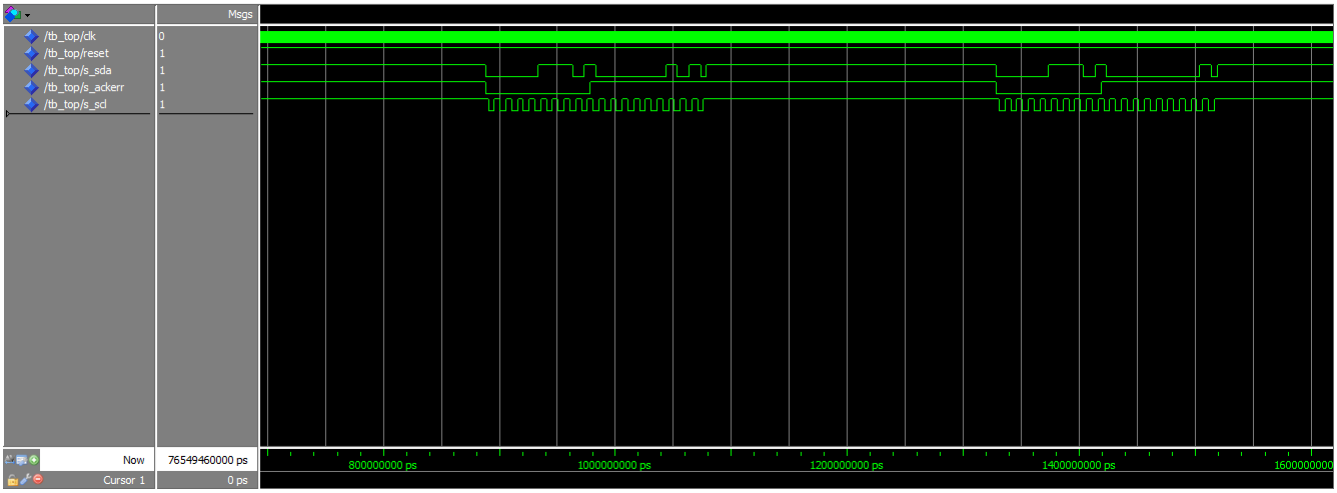

Figure 7: Simulación de la transmisión. Puede observarse la trama del protocolo.

3.4 Hardware

Para la implementación en hardware se utilizo una FPGA *Cyclon II EP2C5T144C8* como MASTER y un Arduino UNO en modo SLAVE para recibir la transmisión, juntos con los periféricos para las conexiones correspondientes. Los datos recibidos por el Arduino UNO, son procesados y transmitidos a través de un puerto serie para poder ser visualizados en la pantalla de la pc (dado que no se contaba con una pantalla lcd de 2x16).

4 Vídeos de funcionamiento

Vídeo del Hardware completo en funcionamiento: **Clic aquí para ver el video.**

5 Conclusión

Con este trabajo pudimos entender en profundidad el protocolo de comunicación serial I2C. Entre las ventajas mas notables se pueden resaltar:

- Solo utiliza dos lineas para la comunicación.
- A diferencia de muchos protocolos, dispone de mecanismos para saber si la señal a llegado con éxito (**ACK/NACK**).
- Se adapta a las necesidades de los distintos Slaves y estos pueden estar presentes en gran cantidad.

Dentro de las desventajas mas notables se encuentran:

- Baja velocidad de transmisión.
- No puede enviar y recibir información simultáneamente.