Trabajo SED

Microcontrolador para un I2C maestro

Objetivos:

- Profundizar en el manejo del lenguaje VHDL.
- Diseño de la estructura básica de un controlador que ejecuta instrucciones.
- Diseño de una comunicación serie I2C para la placa BASYS 3.
- Aplicación a la comunicación con el dispositivo comercial PCF8574A.

Profesor encargado del proyecto:

- Millán, Rafael (Responsable de la asignatura) rmillan@us.es

1. Comunicación serie síncrona I2C

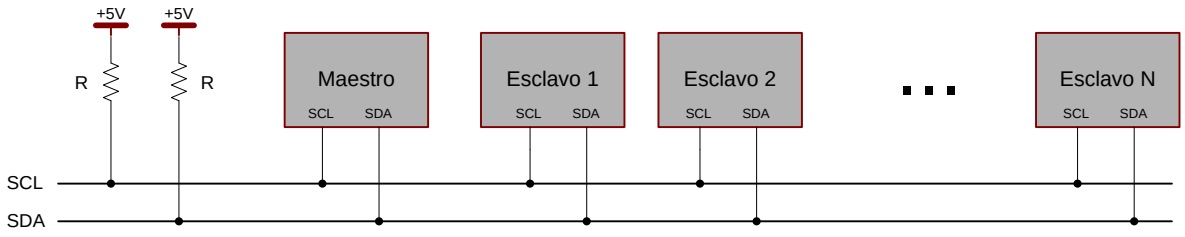
El bus I^2C se ha convertido en un estándar de comunicación serie síncrona entre dispositivos. Requiere solo dos líneas para la comunicación y permite velocidades muy lentas. Los valores típicos de transmisión van de 100Kbits a 400Kbits por segundo.

Las características más importantes del bus I^2C son las siguientes:

1. El bus solo requiere dos líneas: línea de datos serie (SDA) y línea de reloj serie (SCL ó SCK). Todos los dispositivos se conectan a estas 2 líneas. Ambas tienen sendas resistencias de pull-up que fijan un nivel alto en ellas en estado de reposo.
2. Cada dispositivo conectado al bus tiene una dirección única que lo identifica. Lo normal es que los bits más significativos de esta dirección estén fijados por hardware y los bits menos significativos se puedan modificar a través de pines que se pueden poner a nivel bajo o alto.
3. El maestro siempre envía un primer byte –en serie- a todos los esclavos conectados al bus. Este byte contiene un número que identifica el periférico con el que quiere hablar. El periférico seleccionado reconoce ese número y responde poniendo la línea de datos a '0' en el noveno ciclo de reloj que genera el maestro.
4. Tanto el maestro como el esclavo puede ser transmisores o receptores de datos en el transcurso de una comunicación; una vez que el maestro envía el primer byte de selección de esclavo. El último bit de este primer byte establece la dirección de la comunicación: maestro→esclavo o esclavo→maestro. Un dispositivo transmite y el otro recibe (semi-duplex).
5. El bus I^2C se ha convertido en un estándar. En el mercado hay multitud de dispositivos diseñados para comunicarse a través de este bus: memorias, convertidores ADC y DCA, termómetros, relojes en tiempo real,…
6. La comunicación no tiene límite de frecuencia mínima.

1.1. Señales de la comunicación serie I2C

- **SCL** (**S**erial **Cl**ock): es la señal de reloj que sincroniza los datos. Solo puede enviarla un maestro. Dado que el bus I2C está diseñado para sistemas multimaestro, estos deben tener salida a drenador abierto para evitar cotocircuitos entre ellos si accedieran simultáneamente a dicha línea. En este trabajo se verá el caso un sistema que solo contempla un único maestro.
- **SDA** (**S**eria **DA**ta): cada bit de datos se envía por esta línea. Cualquier dispositivo puede escribir en esta línea (maestro y esclavos) por lo que la salida de estos dispositivos debe ser a drenador abierto también.



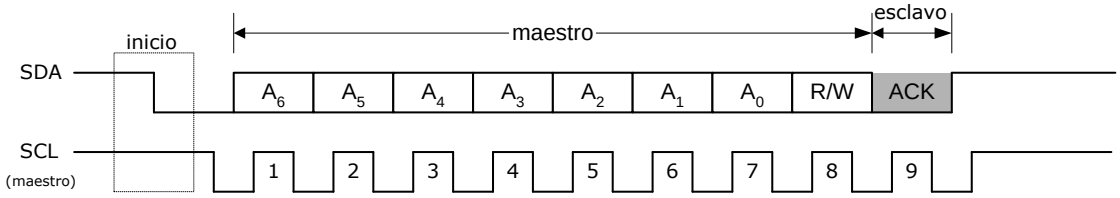
El valor de las resistencias no es crítico. Puede ser cualquiera entre 2k y 10k. Si el valor de la resistencia disminuye se incrementa el consumo en las resistencias pero mejoran otros parámetros: la sensibilidad al ruido, los flancos de las señales,... Lo habitual es utilizar resistencias de 4k7 o 10k.

1.2 Etapas de la comunicación I2C

El procedimiento para la comunicación entre un maestro y un esclavo se va a dividir en 4 etapas: (a) inicio de la comunicación, (b) selección del esclavo por parte del maestro (envío del primer byte por parte del maestro), (c) envío de datos desde el transmisor al receptor. El transmisor puede ser bien el maestro o el esclavo; obviamente, el receptor sería el contrario, y (d) fin de la comunicación. A continuación se detallan las 4 etapas pero no de una forma ordenada.

1.2.1. Inicio de la comunicación

Partiendo de un estado de reposo (SCL='1' y SDA='1') el maestro pone a '0' la línea SDA estando la línea SCL a '1'. Esto solo puede ocurrir al comienzo de una transmisión.



1.2.2. Primer byte: selección del esclavo por parte del maestro

El maestro envía un primer byte para seleccionar al esclavo: los 7 primeros bits identifican al esclavo (se envían en orden decreciente) y el octavo bit indica la dirección de la transmisión de datos (lectura o escritura) que desea el maestro. Hay un noveno bit de **reconocimiento** (ACK) que lo envía el esclavo, una vez que ha comprobado que la dirección recibida es la suya. Durante este noveno bit el maestro libera la línea SDA (pone su salida en alta impedancia y la línea pasa a nivel alto) para que el esclavo la pueda poner a '0' lógico (bit ACK).

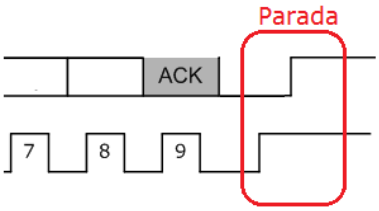
En la figura se aprecia que los bits con la dirección del esclavo los envía el maestro en orden decreciente (A_6-A_0). El octavo bit, $R/\overline{W}$, indica el sentido de los datos:

- $R/\overline{W}$**=0**. Orden de escritura. El maestro es el transmisor de los datos que vienen a continuación y el esclavo es receptor. Este es el caso del trabajo a realizar. En el esquema de proteus, éste bit se ha conectado a tierra.
- $R/\overline{W}$**=1**. Orden de lectura. El maestro es el receptor y el esclavo es el transmisor de los datos.

1.2.3. Envío de datos desde el transmisor al receptor

- La información la envía el transmisor en secuencias de 8 bits de datos más un bit adicional de reconocimiento (enviado por el receptor). El transmisor es el maestro si el bit $R/\overline{W}$='0' o el esclavo si el bit $R/\overline{W}$='1'.
- El dispositivo que gobierna la transmisión es el que genera la señal del reloj I2C (SCL). El esclavo podría bloquear temporalmente el reloj –fijándolo a '0'- si no es lo suficientemente rápido y se ve saturado en la comunicación.

1.2.4. Fin de la comunicación

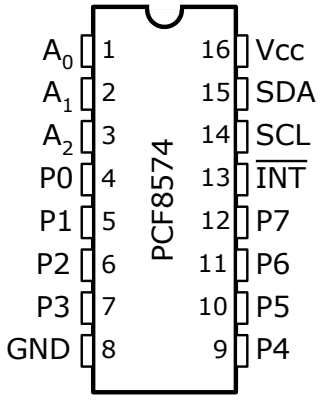


La secuencia de finalización de la comunicación I2C la envía el maestro. Se caracteriza porque la línea SDA pasa de '0' a '1' estando la línea SCL a '1'. El sistema queda en estado de reposo con amba líneas a '1'.

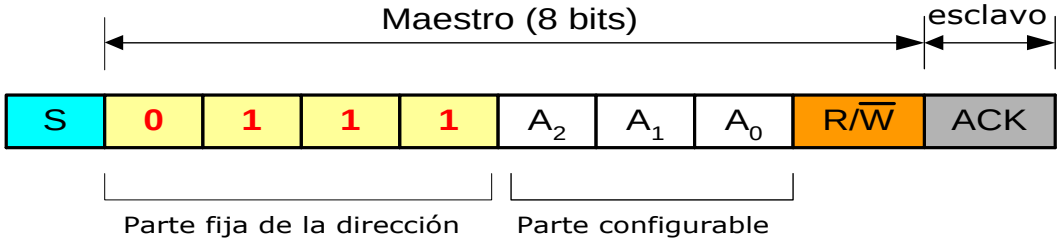
1.3. Dispositivo PCF8574

Este dispositivo permite disponer de un puerto de 8 líneas digitales bien de entrada bien de salida según se configure. La configuración del dispositivo viene implícita con el tipo de comunicación I^2C que establece el maestro: si el último bit del primer byte que envía el maestro es '0' (bit $R/\overline{W}$='0') entonces actúa como un puerto digital de salida y si es '1' (bit $R/\overline{W}$='1') entonces actúa como un puerto digital de entrada. Las características más importantes de este dispositivo son las siguientes:

- Se puede alimentar con una tensión entre 2.5V y 6V, siendo el valor típico 5V.
- Cuando actúa como entrada dispone de un pequeño pull-up que fija la entrada a '1' lógico en ausencia de señal.
- Cuando actúa como salida la información enviada se captura con latches y se fija en los pines hasta la recepción de un nuevo dato. En este caso la corriente máxima que suministra por un pin cuando la salida toma nivel alto es tan solo de 300µA. Imposible para alimentar un led a nivel alto ya que éste necesita 5mA como mínimo.
- Las 3 líneas de dirección (A_2, A_1 y A_0) permiten configurar la dirección del dispositivo. Por tanto, se pueden conectar varios PCF8574A en el mismo bus si tienen valores diferentes en estos pines.

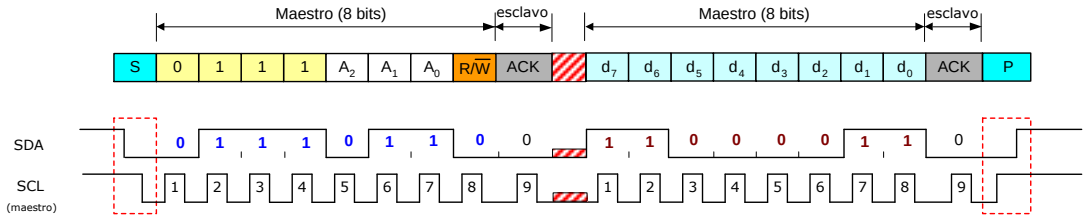


- La línea de interrupción $\overline{INT}$ es a drenador abierto para avisar al microcontrolador por interrupción cuando cambia alguna de las 8 líneas digitales que están configuradas como entradas. En este trabajo no se necesita ya que los pines del dispositivo trabajarán como salidas.
- El identificador contiene una parte fija de 4 bits (**0111**) y una parte variable que se puede fijar con los pines A_2, A_1 y A_0. Este es el primer byte que envía el maestro a todos los esclavos para identificar con cuál de ellos quiere hablar.



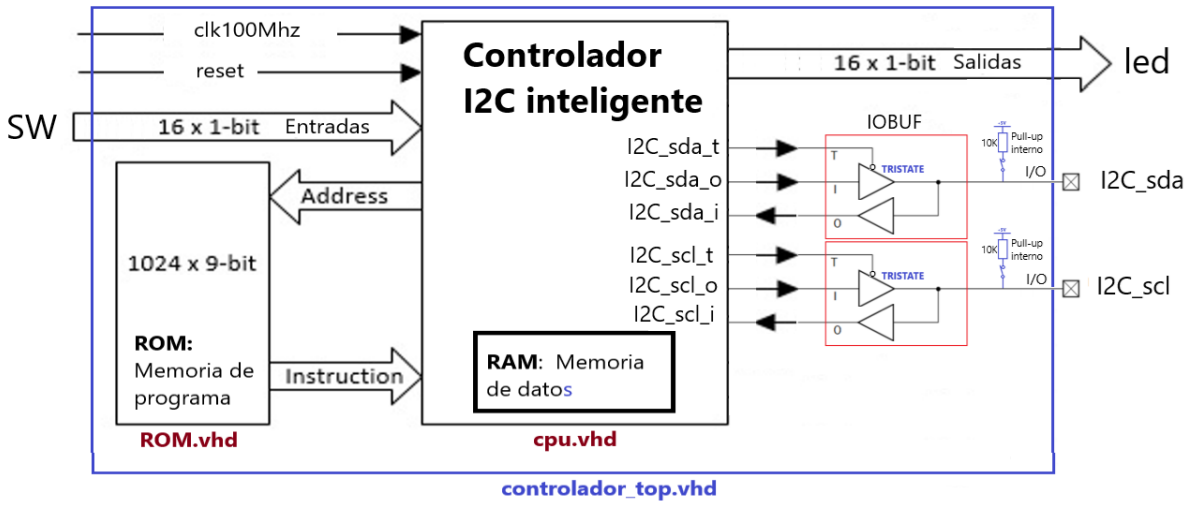
1.3.1. Formato de la trama para comunicar con un esclavo-receptor

Se particulizará para un dispositivo y unos datos en particular. El primer byte identificará el esclavo con número de identificación 0x76 (**01110110**) y el byte que recibirá será **11000011**.



2. Descripción del trabajo

El objetivo de este trabajo no es el diseño de un microcontrolador completo pero sí de una máquina de estado programable mediante instrucciones de un ensamblador específico para gobernar los pines de entrada/salida de la BASYS 3 así como dos pines del puerto JA para las señales de reloj y datos de una comunicación I2C maestro.

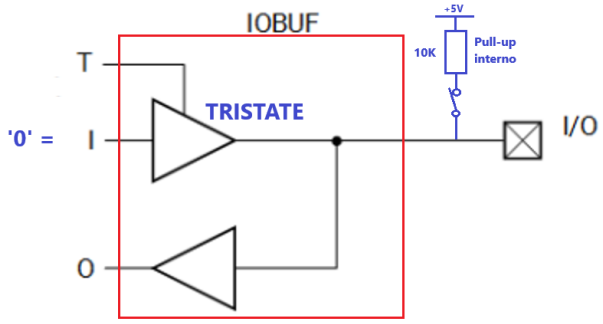


Ficheros: El proyecto consta de 3 ficheros de diseño más los ficheros de simulación que se vayan añadiendo. En el dibujo anterior se aprecian las cajas correspondientes a los 3 ficheros:

- ROM.vhd: Fichero que contiene la memoria de programa (instrucciones). Lo genera el ensamblador que se deja (asm.exe), a partir de un fichero de instrucciones (ROM.asm). Las instrucciones se ubicarán en posiciones consecutivas, a partir de la dirección 0.
- cpu.vhd: Fichero que contiene la unidad central de procesos (cpu) así como la memoria de variables (RAM). La memoria RAM será un vector en el que cada elemento es un std_logic_vector de 8 bits.
- Controlador_top.vhd: Fichero que conecta las dos cajas anteriores (ficheros ROM.vhd y cup.vhd) para obtener el sistema completo, con las entradas y salidas que salen al exterior. La caja de la CPU se parametriza con un GENERIC que permite dividir el reloj de 100Mhz para obtener el reloj I2C (50Khz) contando 2000 ciclos del primero. A la hora de simular, se dividirá por 4 ciclos en lugar de 2000. En la práctica la comunicación I2C no puede funcionar con un reloj SCL de 25Mhz pero en la teoría sí. En Vivado hay que modificar el GENERIC -en este fichero- para que el I2C pueda funcionar a 50khz.

Entradas/Salidas: Todas estas señales están en la BASYS 3. En el dibujo se tiene el reloj, reset, los 16 interruptores (entradas), los 16 leds (salidas) y dos señales del puerto J1 para la comunicación I2C. Todas estas señales ya están configuradas en el fichero de restricciones de Vivado (fichero de pines).

Nos vamos a fijar en las dos señales bidireccionales de la comunicación. Hay que utilizar dos cosas: (a) el elemento de librería IOBUF y (b) el pullup interno de cada pin.



T	TRISTATE	I/O
'0'	ON	'0'
'1'	OFF	'1'

Si la señal T=’0’ (ver tabla del dibujo), el buffer triestado deja pasar la señal I a la salida. En la entrada I se fijará un ‘0’ por lo que pasará a la salida. Si la señal T=’1’, el buffer se pone en alta impedancia por lo que la salida queda al aire. En este caso la resistencia de pullup pone un ‘1’ en la línea de salida. Este pullup está ya activo para los pines JA1 y JA2 en el fichero de restricciones:

```
set_property -dict { PACKAGE_PIN J1 IOSTANDARD LVCMOS33 SLEW SLOW PULLUP true} [get_ports {i2c_scl}]; # JA1
set_property -dict { PACKAGE_PIN L2 IOSTANDARD LVCMOS33 SLEW SLOW PULLUP true} [get_ports {i2c_sda}]; # JA2
```

2.1. Ensamblador específico

Opcode	Instrucción	Acción
00nnnnnnn	**salta** m	PC=m (contador de programa apunta a etiqueta m)
01000nnnn	**brincaSi0** n	Salta la siguiente instrucción si Pin de entrada n vale '0' (sw(n)=0)
01001nnnn	**brincaSi1** n	Salta la siguiente instruccón si Pin de entrada n vale '1' (sw(n)=1)
01010nnnn	**pin0** n	Pin de salida n a '0' => led(n)=’0’
01011nnnn	**pin1** n	Pin de salida n a '1' => led(n)=’1’
01100nnnn	**recibe** n	Recibe dato por I2C y lo pone en el registro n (0 <= n <= 15)
01101nnnn	**lee** m	Retarda 'm' clock cycles (n = log2(m))
01110nnnn	**retardo** n	Retarda 'm' ciclos de I2C_scl (n = log2(m))
011110000	**brincaSiNack**	Salta la siguiente instrucción si NACK est activo
011110001	**recibeBajo**	Recibe dato por I2C y lo pone en el byte alto del puerto de salida
011110010	**enviaBajo**	Envia el byte bajo de las entradas (7 downto 0) al bus I2C
011110100	USER0	A disposición del usuario
.....		
011111011	USER7	A disposición del usuario
011111101	**masterAck**	Bit noveno en el pin I2C_sda será ’0’
011111110	**nop**	No operar (no hace nada)
011111111	**stop**	Envía Stop al bus I2C
1nnnnnnnn	**envía** n	Envía n on I2C bus (el operando está en la propia instrucción)

Las instrucciones **salta, brincaSi1, pin1** y **nop** ya están programadas en la plantilla que se deja. Cada instrucción del fichero ensamblador se ubica en una posición consecutiva a partir de la dirección 0. Solo está la excepción de que haya una etiqueta. En este caso, el ensamblador introduce instrucciones nop hasta que la etiqueta se ubique en la próxima dirección que sea múltiplo de 8.

A continuación se muestra un pequeño ejemplo y los códigos de cada instrucción. Está prohibido saltar a una etiqueta posterior a la instrucción de salto.

```
          pin1      5      ; 010110101      led(5)='1'
bucle:    brincaSi1 5      ; 010000101      Salta si sw(5)='1'
          pin0      5      ; 010100101      led(5)='0'
          salta bucle      ; ¿?
```

En el ejemplo anterior la etiqueta *bucle* está en la dirección 1 que no es múltiplo de 8. El programa ensamblador introducirá 6 instrucciones *nop* para que la etiqueta caiga en la dirección 8. El resultado del ensamblador sería el siguiente:

Dirección	Código	Instrucción
0 - 00000000	010110101	pin1 5
1 - 00000001	011111110	nop
2 - 00000010	011111110	nop
3 - 00000011	011111110	nop
4 - 00000100	011111110	nop
5 - 00000101	011111110	nop
6 - 00000110	011111110	nop
7 - 00000111	011111110	nop
8 - 00001000	010000101	bucle: brincaSi1 5
9 - 00001001	010100101	pin0 5
10 - 00001010	000000001	salta bucle

2.2. Ficheros del diseño

2.2.1 Fichero ROM.vhd

Si el pequeño ejemplo anterior se copia en un fichero llamado ROM.asm y se ensambla con la instrucción: asm.exe ROM.asm, se tiene el fichero ROM.vhd cuyo contenido tiene que ser el mismo de la tabla anterior pero en formato vhdl.

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

entity ROM is
   Port ( clk     : in    STD_LOGIC;
          data    : out std_logic_vector(8 downto 0);
          address : in std_logic_vector(9 downto 0)
         );
end ROM;

architecture Behavioral of ROM is
begin
   process(clk)
   begin
      if rising_edge(clk) then
         case address is
           when "0000000000" => data <= "010110101";   --        pin1  5
           when "0000000001" => data <= "011111110";   --        nop
           when "0000000010" => data <= "011111110";   --        nop
           when "0000000011" => data <= "011111110";   --        nop
           when "0000000100" => data <= "011111110";   --        nop
           when "0000000101" => data <= "011111110";   --        nop
           when "0000000110" => data <= "011111110";   --        nop
           when "0000000111" => data <= "011111110";   --        nop
           when "0000001000" => data <= "010010101";   -- bucle: brincaSi1 5
           when "0000001001" => data <= "010100101";   --        pin0      5
           when "0000001010" => data <= "000000001";   --        salta  bucle
           when others => data <= (others =>'0');
         end case;
      end if;
   end process;
end Behavioral;
```

2.2.2 Fichero controlador_top.vhd

Este fichero está en la cima de la cima de la jerarquía. Interconecta los diferentes elementos y sus entradas/salidas son las que se ven desde el exterior. Conecta el componente de la ROM (fichero ROM.vhd) con la CPU (fichero CPU.vhd) y los elementos IOBUF que son de librería.

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

library unisim;
use unisim.vcomponents.all;


entity controlador is
    Port (  clk100Mhz: in  STD_LOGIC;
            reset    : in  STD_LOGIC;
            i2c_sda  : INOUT STD_LOGIC;
            i2c_scl  : INOUT STD_LOGIC;
            sw       : in std_logic_vector(15 downto 0);     -- pines de entrada (conectados a los interruptores)
            led      : out std_logic_vector(15 downto 0)     -- pines de salida  (conectados a los leds)
         );
end controlador;

architecture controlador_arq of controlador is
    COMPONENT CPU
        Generic( clk_divide : STD_LOGIC_VECTOR (11 downto 0));
        Port (  clk          : IN  std_logic;
                reset        : IN  STD_LOGIC;
                i2c_sda_i    : IN  std_logic;
                i2c_sda_o    : OUT std_logic;
                i2c_sda_t    : OUT std_logic;
                i2c_scl_i    : IN  std_logic;
                i2c_scl_o    : OUT std_logic;
                i2c_scl_t    : OUT std_logic;
                ROM_data     : IN  std_logic_vector(8 downto 0);    -- bus de datos para las instrucciones: 9 bits
                ROM_address  : OUT std_logic_vector(9 downto 0);    -- bus de direcciones: memoria de 1024 instrucc.
                sw           : in std_logic_vector(15 downto 0);    -- pines de entrada (conectados a los interruptores)
                led          : out std_logic_vector(15 downto 0)    -- pines de salida  (conectados a los leds)
             );
    END COMPONENT;

    COMPONENT ROM
    PORT(
          clk     : IN std_logic;
          address : IN std_logic_vector(9 downto 0);     -- bus de datos para las instrucciones: 9 bits
          data    : OUT std_logic_vector(8 downto 0)     -- bus de direcciones: memoria de 1024 instrucc.
        );
    END COMPONENT;

    signal inst_address : std_logic_vector(9 downto 0);   -- bus de datos para las instrucciones: 9 bits
    signal inst_data    : std_logic_vector(8 downto 0);   -- bus de direcciones: memoria de 1024 instrucc.

    -- I2C signals
    signal i2c_sda_i : std_logic;
    signal i2c_sda_o : std_logic;
    signal i2c_sda_t : std_logic;
    signal i2c_scl_i : std_logic;
    signal i2c_scl_o : std_logic;
    signal i2c_scl_t : std_logic;

begin

   Inst_ROM: ROM PORT MAP(
        clk     => clk100Mhz,
        address => inst_address,
        data    => inst_data
    );

-- Inst_CPU: CPU GENERIC MAP(clk_divide => "011111010000") PORT MAP( -- 2000 (100Mhz/2000 = 50Khz I2C clock. Vivadoo)
   Inst_CPU: CPU GENERIC MAP(clk_divide => "000000000100") PORT MAP( -- 4 (100Mhz/4 = 25MHz I2C clock, no es realista)
        clk          => clk100Mhz,
        reset        => reset,
        ROM_address  => inst_address,
        ROM_data     => inst_data,
        i2c_sda_i    => i2c_sda_i,
        i2c_sda_o    => i2c_sda_o,
        i2c_sda_t    => i2c_sda_t,
        i2c_scl_i    => i2c_scl_i,
        i2c_scl_o    => i2c_scl_o,
        i2c_scl_t    => i2c_scl_t,
        sw           => sw,
        led          => led
    );

    Inst_sda_obuf : IOBUF port map (
       IO => I2C_SDA,   -- Buffer inout port (connect directly to top-level port)
       O  => i2c_sda_i, -- Buffer output (to fabric)
       I  => i2c_sda_o, -- Buffer input  (from fabric)
       T  => i2c_sda_t  -- 3-state enable input, high=input, low=output
    );

    Inst_scl_obuf : IOBUF port map (
       IO => I2C_SCL,   -- Buffer inout port (connect directly to top-level port)
       O  => i2c_scl_i, -- Buffer output (to fabric)
       I  => i2c_scl_o, -- Buffer input  (from fabric)
       T  => i2c_scl_t  -- 3-state enable input, high=input, low=output
    );

end controlador_arq;
```

El componente Inst_CPU recibirá un valor para clk_divide de 4 (GENERIC) para la simulación. En el caso de la implementación, se cambiará este valor por 2000 para que el reloj del I2C sea de 50Khz.

El esquema del diseño tiene una salida para el pin I2C_sda y otro para I2C_scl (línea de datos y reloj del I2C). Así se recoge en la ENTITY del fichero *controlador_top.vhd*. Sin

embargo, en el fichero *cpu.vhd* estas señales se manejan a bajo nivel de forma que hay 3 líneas por cada una de ellas que se conectarán a un componente IOBUF. Por ejemplo, la línea de reloj I2C_scl tendría las señales I2C_scl_i, I2C_scl_o, I2C_scl_t. En el fichero controlador_top.vhd se tiene la conexión de estas señales con el componente IOBUF en los dos últimos componentes que se instancia en el fichero: Inst_sda_obuf y Inst_scl_obuf.

La entrada I del componente IOBUF se conecta de forma permanente a nivel bajo '0'. En nuestro caso, las dos primeras instrucciones de la arquitectura son:

```
I2c_scl_o  <= '0';
I2c_sda_o <= '0';
```

La combinación del componente IOBUF -con la entrada I a '0' y la resistencia de pullup- implica que la salida I/0 toma el valor de la señal T (I/O=T):

I2C_sda='1' si i2c_sda_t='1', I2C_sda='0' si i2c_sda_t='0'
I2C_scl ='1' si i2c_scl_t= '1', I2C_scl ='0' si i2c_sda_t='0'

2.2.3. Fichero cpu.vhd

Este es el fichero que se tiene que diseñar para el trabajo.

La CPU tiene como único objetivo ejecutar instrucciones. Cada instrucción se ejecutará en un ciclo de reloj de 100Mhz salvo que la instrucción implique una espera (retardo, envío del bit de start o de stop de una comunicación I2C, envío de datos I2C).

La entidad del fichero cpu.vhd contiene las entradas y salidas de la caja que le corresponde en el esquema.

```
entity CPU is
   Generic( clk_divide : STD_LOGIC_VECTOR(11 downto 0) );

    Port (  clk          : IN  std_logic;
            reset        : IN  STD_LOGIC;
            i2c_sda_i    : IN  std_logic;
            i2c_sda_o    : OUT std_logic;
            i2c_sda_t    : OUT std_logic;
            i2c_scl_i    : IN  std_logic;
            i2c_scl_o    : OUT std_logic;
            i2c_scl_t    : OUT std_logic;
            ROM_data     : IN  std_logic_vector(8 downto 0);    -- bus de datos para las instrucciones: 9 bits
            ROM_address  : OUT std_logic_vector(9 downto 0);    -- bus de direcciones: memoria de 1024 instrucc.
            sw           : in std_logic_vector(15 downto 0);    -- pines de entrada (conectados a los interruptores)
            led          : out std_logic_vector(15 downto 0)    -- pines de salida  (conectados a los leds)
        );
end CPU;
```

Las señales externas SCL y SDA son bidireccionales en el fichero *controlador_top.vhd*. En este fichero se manejan a bajo nivel. Como se comentó anteriormente, las salidas *i2c_sda_o* e *i2c_scl_o* se ponen a '0' al comienzo del programa y se mantienen a ese valor todo el tiempo. Además, las resistencias de pullups internos están activas en el fichero de restricciones. El resultado es que las salidas bidireccionales coinciden con el valor de la señal de control de los buffers triestado.

```
I2c_sda = i2c_sda_t
I2c_scl  = i2c_scl_t
```

La salida *ROM_address* es el bus de direcciones que va a la memoria de programa que generó el ensamblador (fichero ROM.vhd). La entrada *ROM_data* es el bus de datos de la memoria de programa. Cuando se pone una dirección (ya se verá que es en un flanco de subida del reloj de 100Mhz), la memoria pone el dato en *ROM_data* justo un ciclo de reloj después.

La arquitectura del fichero cpu.vhd es la siguiente:

```
architecture CPU_arq of CPU is
   TYPE ESTADOS_CPU IS (STATE_BOOT, STATE_RUN, STATE_DELAY, STATE_I2C_START, STATE_I2C_BITS, STATE_I2C_STOP);
   signal   estado_s, estado_c : ESTADOS_CPU;

   constant OPCODE_SALTA          : std_logic_vector( 3 downto 0) := "0000";  --  0
   constant OPCODE_BRINCA_SI_0    : std_logic_vector( 3 downto 0) := "0001";  --  1
   constant OPCODE_BRINCA_SI_1    : std_logic_vector( 3 downto 0) := "0010";  --  2
   constant OPCODE_PIN_0          : std_logic_vector( 3 downto 0) := "0011";  --  3
   constant OPCODE_PIN_1          : std_logic_vector( 3 downto 0) := "0100";  --  4
   constant OPCODE_I2C_RECIBE     : std_logic_vector( 3 downto 0) := "0101";  --  5
   constant OPCODE_LEE            : std_logic_vector( 3 downto 0) := "0110";  --  6
   constant OPCODE_RETARDO        : std_logic_vector( 3 downto 0) := "0111";  --  7
   constant OPCODE_BRINCA_SI_NACK : std_logic_vector( 3 downto 0) := "1000";  --  8
   constant OPCODE_I2C_ENVIA_BAJO : std_logic_vector( 3 downto 0) := "1001";  --  9
   constant OPCODE_I2C_RECIBE_BAJO: std_logic_vector( 3 downto 0) := "1010";  -- 10
   constant OPCODE_MASTER_ACK     : std_logic_vector( 3 downto 0) := "1011";  -- 11
   constant OPCODE_NOP            : std_logic_vector( 3 downto 0) := "1100";  -- 12
   constant OPCODE_I2C_STOP       : std_logic_vector( 3 downto 0) := "1101";  -- 13
   constant OPCODE_I2C_ENVIA      : std_logic_vector( 3 downto 0) := "1110";  -- 14
   constant OPCODE_UNKNOWN        : std_logic_vector( 3 downto 0) := "1111";  -- 15

   signal opcode    : std_logic_vector( 3 downto 0);
   signal brinca    : std_logic ;
   -- contadores
   signal pcnext    : unsigned(ROM_address'high downto 0);  -- contador de programa. Dirección de la próxima instr.

begin
   i2c_sda_o <= '0';
   i2c_scl_o <= '0';      -- si i2c_scl_t='0' (buffer de salida)  -> saldrá un  '0'
                          -- si i2c_scl_t='1' (buffer de entrada) -> entrará un '1' por el pullup interno

   -- Se asigna la salida "ROM_address" con la direccion de la próxima instruccion
   -- que la recibe la memoria de programa (ROM) como entrada "ROM.address".
   ROM_address <= std_logic_vector(pcnext);

   -- La memoria ROM_address pone en su salida "data" la instruccion actual que llega a este
   -- módulo de la CPU como la entrada "ROM_address_data".
   -- La siguiente instrucción decodifica (identifica) la instruccion -recibida por
   -- la entrada "ROM_address_data"- según sus bits más significativos y la almacena en la señal "opcode".
   opcode <= OPCODE_SALTA          when ROM_data(8 downto 7) = "00"        else
             OPCODE_BRINCA_SI_1    when ROM_data(8 downto 4) = "01001"     else
             OPCODE_PIN_1          when ROM_data(8 downto 4) = "01011"     else
             OPCODE_UNKNOWN;

   -- -------------------------------------------------------
   -- Se asigna  estado_s . Proceso sincrono. Depende del reloj.
   -- -------------------------------------------------------
   PROCESS (clk,reset)
   BEGIN
    IF (reset='1') THEN                    -- Reset activo a nviel alto
       estado_s  <= STATE_BOOT;
    ELSIF (rising_edge(clk)) THEN
       estado_s  <= estado_c;               -- Cambio de estado
    END IF;
   END PROCESS;

   -- -------------------------------------------------------
   -- En este proceso se asigna la señal para el cambio de estado:
   --       > estado_c
   -- -------------------------------------------------------
   PROCESS (estado_s, brinca, opcode)
   BEGIN
      -- ----------------------------------
      estado_c <= estado_s;   -- Valor por defecto
      -- ----------------------------------
      case estado_s is
          when STATE_BOOT =>
              estado_c    <= STATE_RUN;

          when STATE_RUN =>

          when STATE_I2C_START =>

          when STATE_I2C_BITS =>

          when STATE_I2C_STOP =>

          when STATE_DELAY =>

          when others =>
             estado_c  <= STATE_BOOT;
       end case;
   end process;

    -- -------------------------------------------------------
    -- En este proceso se asignan las señales:
    --       > i2c_scl_t
    --       > i2c_sda_t
    --       > pcnext
    --       > led  (pines de salida)
    --       > brinca
    -- -------------------------------------------------------
    PROCESS (clk)
    BEGIN
       IF (rising_edge(clk)) THEN
         case estado_s is

            when STATE_BOOT =>
                led       <= (others => '0');
                i2c_scl_t <= '1';   -- pin I2C_scl de entrada (se verá un '1' por el pull-up interno del pin)
                i2c_sda_t <= '1';   -- pin I2C_sda de entrada (se verá un '1' por el pull-up interno del pin)

                brinca    <= '1';   -- La ROM es síncrona y tarde un ciclo en poner la instrucción
                                    -- desde que se pone la direccion de la misma. La CPU no hace
                                    -- nada en el primer ciclo de reloj después del reset hasta que
                                    -- disponible la primera instruccion
```

```
                -- contadores
                pcnext          <= (others => '0');

            when STATE_RUN =>

               if brinca = '1' then
                  -- No hace nada durante un ciclo de 100Mhz.
                  brinca <= '0';
                  pcnext <= pcnext+1;
               else
                  case opcode is
                     when OPCODE_SALTA =>
                        -- Ignora la siguiente instruccion mientras recoge (fetch) el destino de "salta"
                        brinca <= '1';
                        pcnext <= unsigned(ROM_data(6 downto 0)) & "000";

                     when OPCODE_BRINCA_SI_0 =>


                     when OPCODE_BRINCA_SI_1 =>
                        brinca <= '0';
                        -- Salta una instruccion si el pin de entrada n es '1' -> si sw(n)= '1'
                        -- La posición n está en la propia instrucción
                        --    n= to_integer(unsigned(ROM_data(3 downto 0))
                        if ( sw(to_integer(unsigned(ROM_data(3 downto 0)))) = '1' ) then
                           brinca <= '1';
                        end if;
                        pcnext <= pcnext+1;

                     when OPCODE_PIN_0 =>


                     when OPCODE_PIN_1 =>
                        -- La salida con la posición de 0-15 dada por la propia instruccion
                        -- [ROM_data(3 downto 0)] se pone a 0. (ROM_data(4).  Ej.: led(5) <= '0'
                        led(to_integer(unsigned(ROM_data(3 downto 0)))) <= '1';
                        pcnext <= pcnext+1;

                     when OPCODE_NOP =>

                     when others =>

                  end case;
               end if;

            when STATE_I2C_START =>


            when STATE_I2C_BITS =>     -- scl está a '0' cuando entra


            when STATE_I2C_STOP =>

            when STATE_DELAY =>
               -- divclk permite obtener un ciclo del reloj i2c_scl

            when others =>
               pcnext <= (others => '0');
               brinca   <= '1';

         end case;
      end if;
   end process;

end CPU_arq;
```

El contador de programa es una señal internal llamada *pcnext.* Apunta a la dirección de la próxima instrucción que se va a ejecutar. Por tanto, ROM_address recibe el contenido de pcnext para que proporcione la siguiente instrucción. Dado que ROM_address es del tipo std_logic_vector y pcnext es del tipo unsigned, hay que convertir las señales para poder asignarlas:

ROM_address <= std_logic_vector(pcnext);

La plantilla del fichero *cpu.vhd* contiene los procesos del autómata que ejecuta las instrucciones. Hay un proceso secuencial para la gestión del cambio de estado, un proceso combinacional para preparar el próximo cambio de estado y, finalmente, un proceso para la asignación de todas las señales internas e internas necesarias que, en este ejercicio particular, conviene que sea síncrono ya que ciertas señales (aun no citadas) son contadores para dividir el tiempo.

El automata de la cpu tiene 5 estados: STATE_BOOT, STATE_RUN, STATE_DELAY, STATE_I2C_START, STATE_I2C_BITS, STATE_I2C_STOP.

Cuando se da un reset, el sistema va al estado STATE_BOOT. En este estado se da un valor inicial a todas las señales internas y externas (excepto las señales *estado_c* y *estado_s*). Este estado no es estable y pasa al estado STATE_RUN al siguiente ciclo de

reloj de 100Mhz. En la plantilla, el automáta se queda en este estado ya que, por ahora, solo se ejecutarán instrucciones inmediatas que no implican cambio de estado: salta, brincaSi0, brincaSi1, pin0, pin1, lee y nop.

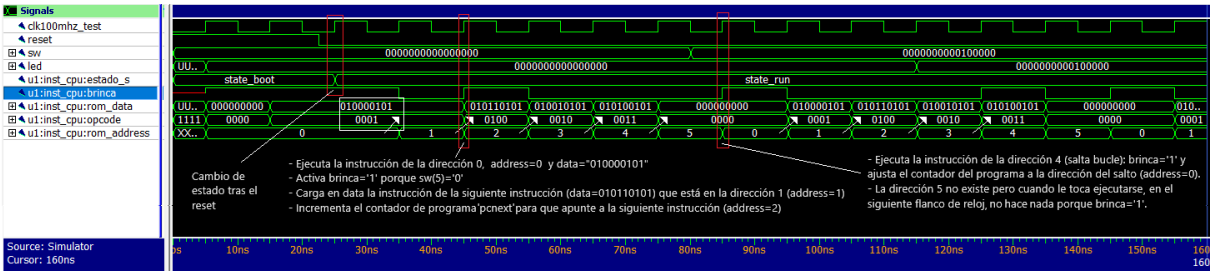
El último proceso es el que contiene el código para la ejecución de las instrucciones según el estado en el que nos encontremos.

6. Ejercicios

6.1. Programación de instrucciones básicas. (1 punto)

La plantilla tiene programadas las instrucciones salta, brincaSi1, pin1. Hay que programar las instrucciones brincaSi0, pin0 y nop; así como completar el fichero de estímulos (los estímulos son las entradas: clk100Mhz, reset y sw). Se muestra un ejemplo de la simulación del siguiente programa (ROM.asm) con un cambio en los estímulos en el instantes 80ns para que sw(5) cambie a '1'. Es importante mostrar señales internas del componente CPU. El siguiente ensamblador hace que la salida led(5) siga a esta entrada.

```
bucle:   brincaSi0   5   ;  brinca si sw(5)='0'
         pin0        5   ;  led(5)='1'
         brincaSi1   5   ;  brinca si sw(5)='0'
         pin1        5   ;  led(5)='1'
         salta bucle      ;   salta a la etiqueta bucle
```



Se pide simular el siguiente programa (ROM.asm) y los estímulos para que cambie la entrada sw(2). Este programa es similar al anterior y hace que la salida led(2)= sw(2) pero el ensamblador reubica la etiqueta bucle en la posición 8 en lugar de la 1. Se recuerda que las instrucciones 'nop' no hacen nada, solo dejan pasar un ciclo de reloj e incrementar el contador de programa para que apunte a la siguiente instrucción.

```
         pin0        2   ;  led(2)='0'
bucle:   brincaSi0   2   ;  brinca si sw(2)='0'
         pin0        2   ;  led(2)='1'
         brincaSi1   2   ;  brinca si sw(2)='0'
         pin1        2   ;  led(2)='1'
         salta bucle      ;   salta a la etiqueta bucle
```

6.2. Programación de la instrucción 'lee'. (1 punto)

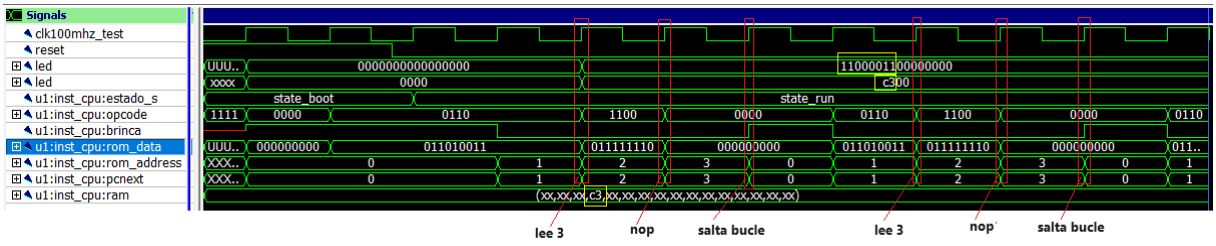
Diseño y simulación de la instrucción 'lee n'. Esta instrucción lee el byte almacenado en la dirección 'n' de la memoria RAM y pone esos bits en el puerto alto de las salidas (led(15 downto 8). La memoria RAM (16x8bytes) se declara como una matriz –interna- en la arquitectura.

```
type RAM_array is array(0 to 15) of STD_LOGIC_VECTOR (7 downto 0);
signal RAM : RAM_array;   -- Memoria RAM para almacenar datos/variables: 16 elementos de 8 bits
```

Excepcionalmente, se asignará el valor "11000011" en la posición 3 de la memoria RAM justo después del *begin* de la arquitectura. La instrucción en VHDL para esta asignación sería: RAM(3) <= "11000011". En el caso de que el número 3 proceda de un std_logic_vector, entonces habría que hacer una conversión a tipo entero: to_integer(unsigned("0011")).

Este valor se leerá con la instrucción de ensamblador 'lee 3'. El resultado se verá en la simulación en los leds de salida que tendrán los valores led(15 downto 8) = 11000011. La siguiente simulación corresponde al ensamblador siguiente. Conviene resaltar que la instrucción de salto necesita 2 ciclos de reloj de 100Mhz. **(1 punto)**.

```
bucle: lee 3    ;  lee el contenido de la memoria RAM(2) y pone los 8 bits en led(15 downto 8)
       nop
       salta bucle
```



Se pide simular el siguiente programa (ROM.asm). Se escribirá el valor 11110011 en la posición 1 de la memoria RAM. En el fichero de estímulo las entradas sw estarán inicialmente a 0 de forma que la primera instrucción brincará la primera vez. Unos ciclos después se cambiará sw(2)='1' para que no salte la instrucción *lee* y la ejecute.

```
bucle:   brincaSi0   2   ;  brinca si sw(2)='0'
         lee         1   ;  led(15 downto 0)= 11110011
         salta bucle     ;   salta a la etiqueta bucle
```

6.3. Programación de la instrucción 'retardo'. (1.5 puntos)

Diseñar la instrucción de 'retardo' que permita a la cpu realizar una espera de un tiempo controlado antes de ejecutar la siguiente instrucción. La instrucción *retardo* tiene un operando que será el número de ciclos del reloj *I2C_scl* (comunicación I2C) que la cpu se mantiene en espera. Un ciclo del reloj I2C se obtiene contando los ciclos de reloj de 100Mhz que indica el valor del GENERIC (clk_divide). El reloj I2C no debe ser mayor a 400Khz para que la comunicación I2C funcione apropiadamente. En la placa BASYS 3 el GENERIC será de clk_divide=2000 para que un ciclo I2C sea de 50Khz (100Mhz/2000).

El operando de la instrucción retardo no puede ser un número cualquiera. Tiene que ser portencia de 2 (1,2,4,8,16,…,32768). La instrucción 'retardo 256' implica que la cpu espera un total de '256*clk_divide' ciclos del reloj de 100Mhz. Si clk_divide fuera de 2000, entonces la espera sería de 195hz (100Mhz/(2000*256).

Se deja una plantilla con elementos nuevos para este apartado que se añadirán al diseño ya realizado de los apartados anteriores. Se comentan estos elementos:

- Se han añadido dos nuevas señales dentro de la arquitectura que actuarán como contadores descendentes (hasta llegar a 0).

```
signal divclk  : unsigned(clk_divide'high downto 0);  -- ciclos de 100Mhz para obtener un ciclo de I2C_scl
signal retardo : unsigned(15 downto 0); -- número de ciclos I2C_scl para conseguir el retardo
```

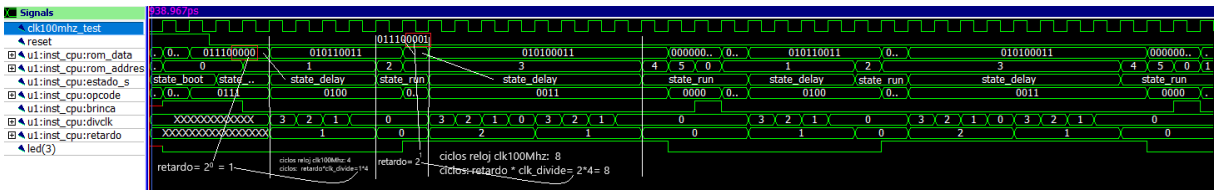
- Hay que decodificar la instrucción 'retardo' en la asignación de 'opcode' que se realiza con un when-else.
- El proceso de asignación de la señal estado_c tendrá nuevos parámetros en la lista sensible: PROCESS (estado_s, divclk, brinca, opcode, retardo).

Además, se añadirán dos transiciones entre estados: (a) del estado STATE_RUN al estado STATE_DELAY cuando la instrucción sea de retardo; y (b) del estado STATE_DELAY al estado STATE_RUN cuando se haya completado el retardo (lo contadores divclk y retardo hayan contado decrecientemente hasta 0 como se muestra en las curvas que hay más abajo).

- En el último proceso hay que realizar dos tareas: (a) en el estado STATE_RUN hay que iniciar los contadores *divclk* y *retardo* para que cuenten decrecientemente y se pueda medir el retardo; y (b) en el estado STATE_DELAY se realiza los decrementos de los contadores de forma que por cada valor que se decrementa 'retardo', el contador 'divclk' tiene que decrementarse desde clk_divide hasta 0.

A continuación se muestra la simulación del siguiente fichero en ensamblador que hace parpadear el led(3) usando ciertos retardos. En la figura se aprecia que los 4 bits menos significativos de la instrucción son el exponente para obtener el número de cuentas de la señal *retardo*. Para simular, el GENERIC clk_divide se ha puesto a 4 en el fichero controlador_top.vhd (el reloj I2C_scl iría a 25Mhz en lugar de 50khz, lo cual no es realista) y el retardo en el ensamblado es de 1 y 2 ciclos del reloj I2C_scl para que la simulación quepa en una pantalla. Si se implementara en Vivado, siempre hay que poner el GENERIC a 2000 (reloj I2C a 50Khz) y el retardo en el ensamblador se pondría a 32768 (para que el led parpadee a 1,5hz y sea visible). **(2 puntos)**

```
bucle: retardo  1   ;  el retardo de 1 ciclo del reloj SCL (I2C_scl= clk_divide * clk100Mhz)
       pin1     3   ;  Led(3)= '1'
       retardo  2   ;  el retardo es 2 ciclos del reloj SCL (I2C_scl= clk_divide * clk100Mhz)
       pin0     3   ;  Led(3)= '0'
       salta bucle
```



Se pide simular el siguiente programa (ROM.asm). Se escribirá el valor 11110011 en la posición 1 de la memoria RAM. En el fichero de estímulos, las entradas *sw* estarán inicialmente a 0. La primera instrucción brincará. Unos ciclos después se cambiará sw(2)='1' para que no salte y ejecute la instrucción *lee*.

```
bucle:   brincaSi0   2   ;  brinca si sw(2)='0'
         lee         1   ;  led(15 downto 0)= 11110011
         salta bucle     ;   salta a la etiqueta bucle
```

6.4. Programación de la instrucción 'stop'. (1.5 puntos)

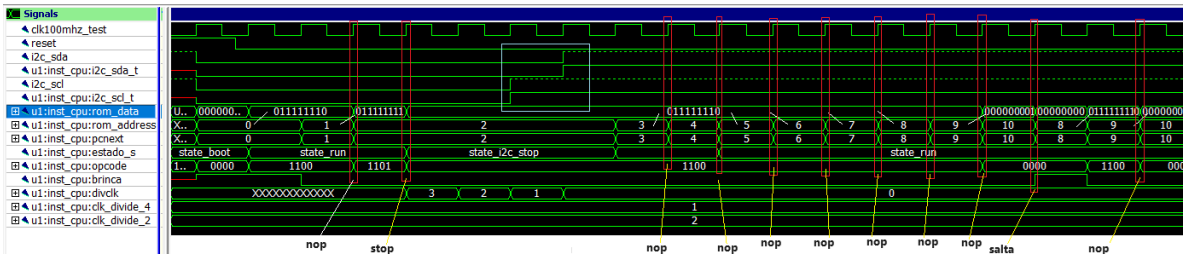
Diseño de la instrucción *stop* que finaliza una comunicación I2C poniendo las señales I2C_scl e I2C_sda a '1' con una secuencia explicada anteriormente.

- Las señales clk_divide_2 y clk_divide_4 que son la mitad de ciclos y la cuarta parte de ciclos del GENERIC *clk_divide*. Se utilizarán para comparar la cuenta decreciente de divclk y saber si va por la mitad o un cuarto del tiempo necesario para un ciclo de reloj I2C_scl.
- Excepcionalmente se pondrá a '0' la señal bidireccional del reloj I2C_scl. Para ellos se cambiará I2C_scl_t='0' en el estado STATE_BOOT del último proceso.
- Se decodificará la instrucción stop (OPCODE_I2C_STOP) en el when-else de la señal *opcode*.

- El proceso de asignación de la señal estado_c tendrá los parámetros del punto anterior en la lista sensible. Además, se añadirán dos transiciones entre estados: (a) del estado STATE_RUN al estado STATE_I2C_STOP cuando la instrucción sea 'stop'; y (b) del estado STATE_I2C_STOP al estado STATE_RUN cuando se haya completado la secuencia de stop (el contador *divclk* haya contado decrecientemente hasta 0.
- En el último proceso hay que realizar varias tareas: (a) en el estado STATE_RUN hay que iniciar el contador *divclk* para que cuente decrecientemente y se pueda medir un ciclo del reloj I2C_scl; y (b) en el estado STATE_I2C_STOP hay que hacer varias cosas a medida que divclk cuenta decrecientemente desde 'clk_divide-1' hasta 0:
  - Poner la señal I2C_sda a '0' por si acaso cuando ha pasado un cuarto de la cuenta anterior. Como ésta es decreciente será cuando divclk es igual a clk_divide - clk_divide_4. Se les pondrá el molde unsigned(--) a clk_divide y clk_divide_4.
  - Poner I2C_scl a '1' cuando la cuenta de divclk vaya por la mitad
  - Poner I2C_sda a '1' cuando la cuenta de divclk vaya por ¾ del total. Al ser decreciente, le quedará un cuarto para terminar, es decir, divclk sea igual a clk_divide_4
  - Cuando la cuenta de divclk haya llegado a 0, se pasará a la siguiente instrucción (pcnext= pcnext+1). En caso contrario continuará con la cuenta decreciente.

A continuación se muestra la simulación del siguiente fichero en ensamblador que ejecuta la instrucción 'stop' para realizar una secuencia de parada en la comunicación. **(1.5 puntos)**

```
        nop
        stop ; Necesita un ciclo de I2C_scl ('clk_divide' ciclos del reloj clk100Mhz para completarse)
bucle:  nop
        salta bucle
```



Se pide simular el siguiente programa (ROM.asm). En el fichero de estímulos, las entradas *sw* estarán inicialmente a 1. La primera instrucción brincará. Unos ciclos después se cambiará sw(3)='1' para que no salte y ejecute la instrucción *lee*.

```
bucle:   brincaSi1   3   ;  brinca si sw(3)='1'
         stop            ;
         salta bucle     ;  salta a la etiqueta bucle
```

6.5. Programación parcial de la instrucción 'envia'. (2 puntos)

Diseño parcial de la instrucción *envia* en la que no se transmitirán los bits de la trama por la señal I2C_sda. Se enviará el start, 9 ciclos del reloj y el stop. Se deja una plantilla que contiene los siguientes nuevos elementos:

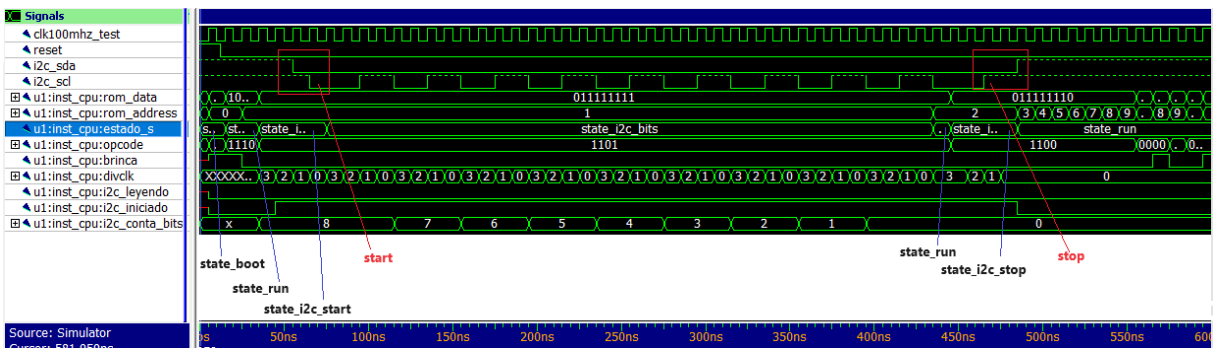
- La señal I2C_scl_t recuperará su valor correcto de '1' en el estado inicial STATE_BOOT. De esta forma el sistema arranca con las señales I2C a '1'.
- Se decodificará la instrucción *envia* (OPCODE_I2C_ENVIA) en el when-else de la señal *opcode*.

- El proceso de asignación de la señal estado_c tendrá los parámetros del punto anterior en la lista sensible. Además, se añadirán nuevas transiciones entre estados:
  - Del estado STATE_RUN al estado STATE_I2C_START (si i2c_iniciado='0') o al estado STATE_I2C_BITS (si i2c_iniciado='1') al ejecutar la instrucción *envia*;
  - Del estado STATE_I2C_START al estado STATE_I2C_BITS cuando se haya completado la secuencia de start (el contador *divclk* haya contado decrecientemente hasta 0).
  - El estado STATE_I2C_BITS retorna al estado STATE_RUN cuando se hayan enviado los 9 ciclos de reloj I2C_scl de la trama que equivalen a 9**clk_divide* ciclos de clk100Mhz. Habrá una señal nueva que contará de forma decreciente los 9 ciclos del reloj I2C_scl (ver figura más abajo).
  - La siguiente instrucción del ensamblador será *stop* por lo que pasará del estado STATE_RUN (estado que ejecuta instrucciones) al estado STATE_I2C_STOP como se vio en el apartado anterior. Finalmente, se este nuevo estado retornará al estado STATE_RUN para ejecutar la siguiente instrucción.

- En el último proceso hay que realizar varias tareas:
  - En el estado STATE_RUN, cuando se ejecuta OPCODE_I2C_ENVIA, hay que iniciar el contador *divclk*, el contador i2c_conta_bits al valor de 8 (contará de 8 a 0 que suman 9 ciclos del reloj I2C_scl), y la señal i2c_recibiendo='0' ya que la instrucción es de envío.
  - En el estado STATE_I2C_START es muy similar al estado STATE_I2C_STOP pero con la peculiaridad que no se vuelve al estado STATE_RUN -para ejecutar otra instrucción- por este motivo no se incrementará el contador de programa (pcnext).
  - En el estado STATE_I2C_BITS se decrementará el contador divclk en cada ciclo del reloj de 100Mhz. Cuando llega a 0 se decrementará la señal I2C_conta_bits hasta que llegue a 0 y retorna al estado STATE_RUN para que ejecute la siguiente instrucción. Por tanto, antes de irse, incrementará el registro contador de programa (pcnext). Para cada cuenta de *divclk* desde *clk_divide-1* hasta 0, se comparará con dos valores: (a) si vale clk_divide_2 pondrá la señal i2c_scl_t='1' (flanco de subida del reloj I2C_scl), y (b) si vale 0 pondrá la señal anterior a '0' (flanco de bajada del reloj I2C_scl).

A continuación se muestra la simulación del siguiente fichero en ensamblador que ejecuta la instrucción 'stop' para realizar una secuencia de parada en la comunicación.

```
        envia 0x70
        stop ; Necesita un ciclo de I2C_scl ('clk_divide' ciclos del reloj clk100Mhz para completarse)
bucle:  nop
        salta bucle
```



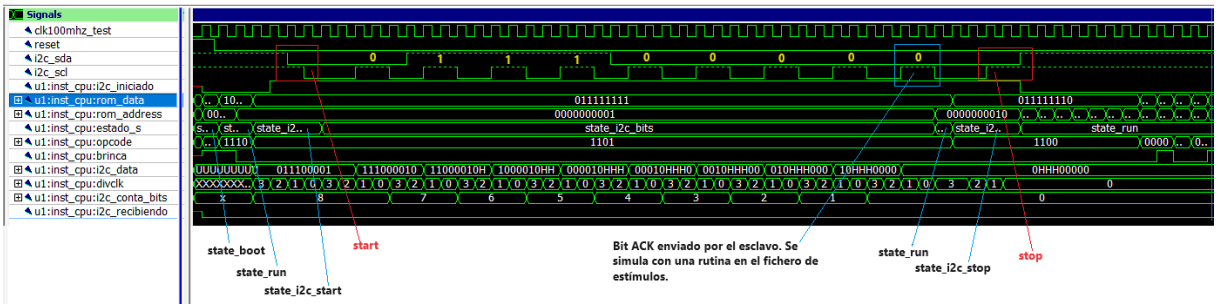
Se pide simular el siguiente programa (ROM.asm).

```
bucle: envia 0x70
       envia 0x0F
       stop
       nop
       salta bucle
```

6.6. Programación completa de la instrucción 'envia' y 'enviaBajo'. (2 puntos)

Se necesitará un registro de desplazamiento de 9 bits llamado I2C_data(8 downto 0) que contendrá los 8 bits de datos a enviar más un noveno bit para el acuse de recibo del esclavo. El desplazamiento es hacia la izquierda de forma que sale el bit I2C_data(8) para poner en la línea I2C_sda y por la derecha entra el dato que hay en la línea I2C_sda (i2c_sda_i), es decir, el dato que ha enviado a excepción del 9 bit que será el acuse de recibo del esclavo (ACK). Para este noveno bit, el maestro cree poner el bit '1' en la línea pero el esclavo puede dar acuse de recibo fijando la línea a '0'. En el fichero de estímulos se deja un 'procedure' que es una función que permite simular el acuse de recibo del esclavo cuando éste quiere que el bit de acuse de recibo sea '0'.
A continuación se muestra la simulación del siguiente fichero en ensamblador que ejecuta la instrucción 'stop' para realizar una secuencia de parada en la comunicación.

```
        envia 0x70
        stop ; Necesita un ciclo de I2C_scl ('clk_divide' ciclos del reloj clk100Mhz para completarse)
bucle:  nop
        salta bucle
```



Se pide simular el siguiente programa en ensamblador. Se le dará un valor inicial a los interruptores en el fichero de estímulos. Por ejemplo: 0x00F0 para que la instrucción enviaBajo envíe el valor 0xF0.

```
bucle: envia 0x70    ; Se comunica con el dispositivo PFC8574 en escritura
       envia 0x0F    ; Envía el dato 0x0F
       stop ; Necesita un ciclo de I2C_scl ('clk_divide' ciclos del reloj clk100Mhz para completarse)
       retardo 1     ; En Vivado se cambiará 1 por 32768
       envia 0x70    ; Se comunica con el dispositivo PFC8574 en escritura
       enviaBajo     ; Envía el contenido de los interruptores bajos, sw(7 downto 0)
       retardo 1     ; En Vivado se cambiará 1 por 32768
       salta bucle
```

6.7. Programación de las instrucciones 'masterAck', 'recibe' y 'recibeBajo'. (1 punto)

La instrucción 'masterAck' siempre se ejecutará antes de una instrucción 'recibe' para que el maestro de acuse de recibo (ACK='0') en el noveno bit. Así el esclavo sabe que la comunicación continúa y se le va a solicitar que transmita un nuevo dato. Si no está la instrucción 'masterAck', antes de la instrucción 'recibe' (en el fichero ensamblador), entonces el esclavo no tiene acuse de recibo (ACK='1' por el pullup en I2C_sda) y sabe que terminará la comunicación.
La instrucción 'masterAck' pondrá a '1' una señal std_logic –nueva, llamada *ack_flag* de la arquitectura-. Esta instrucción se ejecuta en el estado STATE_RUN y, como no requiere ningún tiempo de espera, no necesita un estado asociado a ella. La siguiente instrucción

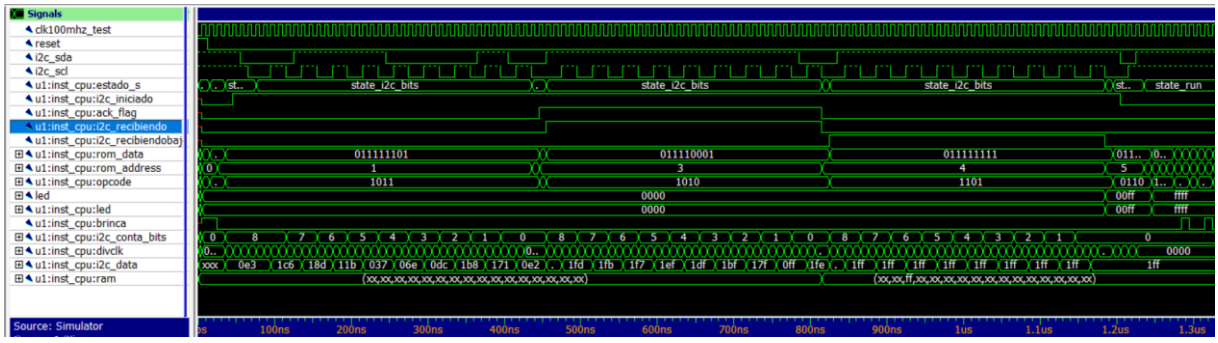
del ensamblador debe ser 'recibe' o 'recibeBajo' que preparará el registro de desplazamiento con el valor: I2C_data <= 0xFF & (not ack_flag) y pasa al estado STATE_I2C_BITS. En este estado el controlador cree que está enviando '1' por la señal I2C_sda pero en realidad está recibiendo los datos que el esclavo está poniendo en esa misma línea (se leerá la señal interna I2C_sda_i) que se introducirán en el registro de desplazamiento I2C_data por la derecha. Justo en el noveno bit, el esclavo no pone nada en la línea y escucha lo que el maestro fija en la línea I2C_sda que es el acuse de recibo. En ese momento, los 8 bits más significativos del registro de desplazamiento contienen el dato recibido (I2C_data).

En la simulación es complicado simular el envío de datos por parte del esclavo. No se programará nada especial en el fichero de estímulos para la recepción de datos, por lo que se recibirá el valor de la línea I2C_sda en reposo, es decir, se recibirá el byte 11111111.

Mientras se ejectua la instrucción *recibe* se pondrá a '1' la señal interna I2C_recibiendo y mientras se ejecuta la instrucción *recibeBajo* se pondrá a '1' la señal interna I2C_recibiendoBajo. De esta forma, en el estado state_i2c_bits se sabrá donde hay que ubicar los 8 bits recibidos (bien en la memoria RAM bien en el byte bajo de las salidas, led(7 downto 0).

A continuación se muestra la simulación del siguiente fichero en ensamblador que utiliza estas instrucciones:

```
        envia 0x71 ; Establece comunicación I2C en lectura con el PFC8574
        masterAck
        recibe 2    ; Recibe el dato y lo ubica en la posición de memoria 2. El maestro fija ACK='0'
        recibeBajo ; Recibe el dato y lo poone en led(7 downto 0). El bit ACK='1' (NACK)
        stop ; Necesita un ciclo de I2C_scl ('clk_divide' ciclos del reloj clk100Mhz para completarse)
        lee    2    ; Lee el dato de la posición de memoria 2 y lo pone led(15 downto 0)
bucle:  nop
        salta bucle
```



Se pide simular el siguiente programa en ensamblador.

```
bucle: envia 0x71    ; Se comunica con el dispositivo PFC8574 en lectura
       masterAck
       recibeBajo    ; Recibe el dato y lo poone en led(7 downto 0). El bit ACK='1' (NACK)
       stop
       nop
       retardo 1     ; En Vivado se cambiará 1 por 32768
       salta bucle
```

7. Observaciones

- Fecha de finalización: 8 **de abril a las 24h**.
- Se evaluará la presentación del fichero PDF así como el código. Se penalizará si el código no está bien comentado ni indentado.
- El diseño de trabajo es secuencial. La realización de un apartado depende de lo realizado anteriormente. En los apartados se dejan plantillas con los nuevos elementos que se deben añadir pero el diseño deber realizarse sobre los ficheros diseñados con los apartados anteriores.
- Los diez primeros trabajos serán bonificados con un 10% extra de nota. Si la nota supera los 10 puntos, saturará a este valor. Solo se permite subir los trabajos una sola vez a la enseñanza virtual. Hay que estar seguro de lo que se envía.
- El trabajo se subirá a la EV con el título "TRABAJO GRUPO N". Se adjuntará un fichero comprimido ZIP. Se incluirá la misma estructura de la plantilla sustituyendo el fichero MEMORIA_trabajo1.docx por el fichero MEMORIA_trabajo1.pdf obtenido a partir del anterior. El fichero comprimido se llamará trabajo_grupo_N.zip siendo N el número del grupo.
- Todos los miembros del grupo deben subir el trabajo para que el sistema permita evaluarlos. En la primera página del fichero MEMORIA_trabajo1.pdf se indicará el nombre de todos los miembros del grupo de forma clara. La fecha de entrega se considerará la del último miembro del grupo que suba la práctica.
- Si el trabajo se entrega fuera de plazo se penalizará con un 10% por cada día de retraso respecto a la fecha límite.
- Los ficheros entregados se pasarán por un programa que detecta el porcentaje de similitud con los ficheros entregados por otros grupos. Si se detecta que dos grupos se han copiado, la nota del trabajo se dividirá por dos y se asignará a los miembros de ambos grupos sin distinguir qué grupo lo hizo y cual lo copió. Si son 3 los grupos que copian, la nota se dividirá por 3 y así sucesivamente.
- Sólo se permitirán grupos de 3 miembros.
- Cualquier explicación, comentario o sugerencia por parte del profesorado será universal a través del foro. No se resolverán dudas en el despacho de los profesores porque podrían suponer una ventaja respecto al resto de grupos. Tanto alumnos como profesores pueden contestar a cualquier cuestión planteada en el foro. El foro será anónimo y un usuario puede borrar sus mensajes si no han tenido respuesta.
- En el fichero comprimido deben estar presente todos los ficheros de simulación necesarios en cada apartado, así como el fichero de curvas. Además, se incluirá una subcarpeta en cada apartado que se llamará 'vivado'. Est carpeta contendrá los ficheros *.vhd que se usarán para la implementación así como el fichero que genera Vivado (*.bit) , a partir de estos. Se recuerda que para generar el fichero de Vivado hay que modificar el fichero *controlador_top.vhd* para que el reloj de 100Mhz se divida por 2000 para obtener el reloj I2C en lugar de 4:

```
    Inst_CPU: CPU GENERIC MAP(clk_divide => "011111010000") PORT MAP( -- 2000 (100Mhz/2000 = 50Khz I2C clock. Vivadoo)
--  Inst_CPU: CPU GENERIC MAP(clk_divide => "000000000100") PORT MAP(  --     4 (100Mhz/4 = 25MHz I2C clock, no es realista)
```

  En el caso de que el fichero ensamblador tenga alguna instrucción de retardo, el operando se cambiará de un valor pequeño al máximo 32768. Se ensamblará para obtener un nuevo fichero ROM.vhd que se utilizará para generar el fichero de Vivado (*.bit) que habrá que incluir en el proyecto.
- El proyecto se evaluará si se ha llegado a completar el diseño de la transmisión completa de un dato (apartado 6).