CODI FONT DELS *IP CORES* ASSOCIATS AL PROJECTE

CREACIÓ D' *IP CORES* EN UNA PLATAFORMA NIOS: METODOLOGIA DE DISSENY

Memòria del Projecte Fi de Carrera
d' Enginyeria en Informàtica
realitzat per
Antoni Costa Sanfeliu
i dirigit per
Joan Oliver Malagelada
Bellaterra, 15 de Juny de 2007

ÍNDEX

1. *IP CORE:* PWM

1.1 PWM Hardware

A continuació es detalla el codi utilitzat a nivell hardware, desglossat en les tres seccions estudiades: Tasca lògica, registres i interfície Avalon.

1.1.1 Fitxer de la tasca lògica (pwm_task_logic.v)

```
module pwm_task_logic
(
	clk,
	pwm_enable,
	resetn,
	clock_divide,
	duty_cycle,
	pwm_out
);

//Entrades
input clk;                      //Senyal d'entrada de rellotge
input [31:0] clock_divide;      //Valor de la divisió del rellotge
input [31:0] duty_cycle;        //Valor del cicle de treball
input pwm_enable;               //Senyal d'enable
input resetn;                   //Reset

//Sortides
output pwm_out;                 //Sortida PWM

//Declaració de senyals
reg [31:0] counter;             //Comptador intern PWM
reg pwm_out;                    //Sortida PWM

//Codi
always @(posedge clk or negedge resetn)         //Process del comptador PWM
begin
	if (~resetn)begin
		counter <= 0;
	end
	else if(pwm_enable)begin
		if (counter >= clock_divide)begin
			counter <= 0;
		end
		else begin
			counter <= counter + 1;
		end
	end
```

```
	else begin
		counter <= counter;
	end
end

always @(posedge clk or negedge resetn)	//Comparador PWM
begin
	if (~resetn)begin
		pwm_out <= 0;
	end
	else if(pwm_enable)begin
		if (counter >= duty_cycle)begin
			pwm_out <= 1'b1;
		end
		else begin
			if (counter == 0)
				pwm_out <= 0;
			else
				pwm_out <= pwm_out;
			end
		end
	else begin
		pwm_out <= 1'b0;
	end
end

endmodule
```

1.1.2 Fitxer de registres (pwm_register_file.v)

```
module pwm_register_file
(	//Senyals del bus Avalon
	clk,
	resetn,
	chip_select,
	address,
	write,
	write_data,
	read,
	read_data,

	//Senyals de sortida del PWM
	pwm_clock_divide,
	pwm_duty_cycle,
	pwm_enable
);

//Paràmetres
parameter clock_divide_reg_init = 32'h0000_0000;
parameter duty_cycle_reg_init   = 32'h0000_0000;

//Entrades
input clk;			//Rellotge del sistema
input resetn;			//Reset del sistema
input chip_select;		//ChipSelect del bus Avalon
input [1:0] address;		//Bus d'adreces del bus Avalon
```

```
input write;                    //Senyal d'escriptura del bus Avalon
input [31:0] write_data;        //Bus de dades d'escriptura
input read;                     //Senyal de lectura del bus Avalon

//Sortides
output [31:0] read_data;        //Bus de dades de lectura
output [31:0] pwm_clock_divide; //Senyal PWM dividida del rellotge
output [31:0] pwm_duty_cycle;   //Senyal PWM del cicle de treball
output        pwm_enable;       //Senyal PWM d'enable

//Declaració de senyals
reg [31:0] clock_divide_register;      //Registre de divisió del rellotge
reg [31:0] duty_cycle_register;        //Registre del cicle de treball
reg        enable_register;            //Bit d'enable
reg [31:0] read_data;                  //Bus de dades de lectura

//Descodificació d'adreces
wire clock_divide_reg_selected, duty_cycle_reg_selected,
enable_reg_selected;
//Validació d'una escriptura en una adreça concreta
wire write_to_clock_divide, write_to_duty_cycle, write_to_enable;
//Validació d'una lectura en una adreça concreta
wire read_to_clock_divide, read_to_duty_cycle, read_to_enable;
//Validació d'un accés correcte
wire valid_write, valid_read;

//Codi Principal

//Descodificació d'adreces
assign clock_divide_reg_selected = !address[1] & !address[0];  //adreça 00
assign duty_cycle_reg_selected   = !address[1] &  address[0];  //adreça 01
assign enable_reg_selected       =  address[1] & !address[0];  //adreça 10

//Determinar si una transacció vàlida s'ha iniciat
assign valid_write = chip_select & write;
assign valid_read  = chip_select & read;

//Determinar si s'ha produït una escriptura en una adreça concreta
assign write_to_clock_divide = valid_write & clock_divide_reg_selected;
assign write_to_duty_cycle   = valid_write & duty_cycle_reg_selected;
assign write_to_enable       = valid_write & enable_reg_selected;

//Determinar si s'ha produït una lectura en una adreça concreta
assign read_to_clock_divide = valid_read & clock_divide_reg_selected;
assign read_to_duty_cycle   = valid_read & duty_cycle_reg_selected;
assign read_to_enable       = valid_read & enable_reg_selected;

//Escriure en el registre divisor del rellotge
always@(posedge clk or negedge resetn)
begin
      if(~resetn)begin //Async Reset
            clock_divide_register <= clock_divide_reg_init;
      end
      else begin
            if(write_to_clock_divide) begin
                  clock_divide_register <= write_data;
            end
            else begin
                  clock_divide_register <= clock_divide_register;
            end

      end
```

```
end

//Escriure en el registre del cicle de treball
always@(posedge clk or negedge resetn)
begin
	if(~resetn)begin //Async Reset
		duty_cycle_register <= duty_cycle_reg_init;
	end
	else begin
		if(write_to_duty_cycle) begin
			duty_cycle_register <= write_data;
		end
		else begin
			duty_cycle_register <= duty_cycle_register;
		end
	end
end

//Escriure en el registre d'enable
always@(posedge clk or negedge resetn)
begin
	if(~resetn)begin //Async Reset
		enable_register <= 1'b0;
	end
	else begin
		if(write_to_enable)begin
			enable_register <= write_data[0];
		end
		else begin
			enable_register <= enable_register;
		end
	end
end

//Llegir del bus de dades
always@(read_to_clock_divide or read_to_duty_cycle or read_to_enable or
clock_divide_register or duty_cycle_register or enable_register)
begin
	if(read_to_clock_divide) begin
		read_data = clock_divide_register;
	end
	else if(read_to_duty_cycle) begin
		read_data = duty_cycle_register;
	end
	else if(read_to_enable) begin
		read_data = {31'd0,enable_register};
	end
	else begin
		read_data = 32'h0000_0000;
	end
end

//Assignar els valors dels registres a les sortides del PWM
assign pwm_clock_divide = clock_divide_register;
assign pwm_duty_cycle = duty_cycle_register;
assign pwm_enable = enable_register;

endmodule
```

1.1.3 Fitxer d'interfície (pwm_avalon_interface.v)

```
module pwm_avalon_interface
(      clk,
       resetn,
       avalon_chip_select,
       address,
       write,
       write_data,
       read,
       read_data,
       pwm_out
 );

//Valors a passar als registres
parameter clock_divide_reg_init = 32'h0000_0000;
parameter duty_cycle_reg_init = 32'h0000_0000;

//Entrades/Sortides PWM Avalon Slave
input clk;                      //Rellotge del sistema
input resetn;                   //Reset del sistema
input avalon_chip_select;       //Avalon Chipselect
input [1:0]address;             //Avalon bus d'adreces
input write;                    //Senyal Avalon d'escriptura
input [31:0]write_data;         //Bus de dades d'escriptura
input read;                     //Senyal de lectura

output [31:0]read_data;         //Bus de dades de lectura

//Entrades/Sortides exportades PWM Avalon_Slave_
output pwm_out;                 //Senyal de sortida PWM

//Nodes Interns del PWM Avalon Slave

//Senyal divisòria de rellotge del fitxer de registres a la tasca lògica
wire [31:0] pwm_clock_divide;
//Valor del cicle de treball del fitxer de registres a la tasca lògica
wire [31:0] pwm_duty_cycle;
//Senyal d'enable PWM del fitxer del registres a la tasca lògica
wire pwm_enable;


//PWM Instància
pwm_task_logic task_logic
(
    .clk                (clk ),
    .pwm_enable         (pwm_enable),
    .resetn             (resetn),
    .clock_divide       (pwm_clock_divide),
    .duty_cycle         (pwm_duty_cycle),
    .pwm_out            (pwm_out)
);

//Instàncies del fitxer de registres
pwm_register_file #(clock_divide_reg_init, duty_cycle_reg_init)
memory_element
(
       .clk                 (clk),
       .resetn              (resetn),
       .chip_select         (avalon_chip_select),
```

```
      .address                  (address),
      .write                    (write),
      .write_data               (write_data),
      .read                     (read),
      .read_data                (read_data),
      .pwm_clock_divide         (pwm_clock_divide),
      .pwm_duty_cycle           (pwm_duty_cycle),
      .pwm_enable               (pwm_enable)
);

endmodule
```

1.2 PWM Software

A continuació es detalla el codi utilitzat a nivell software, en dos nivells. En el primer es mostren les capçaleres on es defineixen les *macros* per a accedir als registres del component PWM, i per l'altra les HAL (*Hardware Abstraction Layer*) que el processador NIOSII utilitza com a *drivers* per a l'execució dels components.

1.2.1 INC (avalon_slave_pwm_regs.h)

```
#ifndef __AVALON_PWM_REGS_H__
#define __AVALON_PWM_REGS_H__

#include <io.h>

#define IORD_AVALON_PWM_CLOCK_DIVIDER(base)       IORD(base, 0)
#define IOWR_AVALON_PWM_CLOCK_DIVIDER(base, data) IOWR(base, 0, data)

#define AVALON_PWM_CLOCK_DIVIDER_MSK              (0xFFFFFFFF)
#define AVALON_PWM_CLOCK_DIVIDER_OFST             (0)

#define IORD_AVALON_PWM_DUTY_CYCLE(base)         IORD(base, 1)
#define IOWR_AVALON_PWM_DUTY_CYCLE(base, data)   IOWR(base, 1, data)
#define AVALON_PWM_DUTY_CYCLE_MSK                (0xFFFFFFFF)
#define AVALON_PWM_DUTY_CYCLE_OFST               (0)

#define IORD_AVALON_PWM_ENABLE(base)             IORD(base, 2)
#define IOWR_AVALON_PWM_ENABLE(base, data)       IOWR(base, 2, data)
#define AVALON_PWM_ENABLE_MSK                    (0x1)
#define AVALON_PWM_ENABLE_OFST                   (0)

#endif
```

1.2.2 HAL/INC (avalon_pwm_routines.h)

```
#include "avalon_pwm_regs.h"

#define AVALON_PWM_TYPE (volatile unsigned int*)

int avalon_pwm_init(unsigned int address, unsigned int clock_divider,
unsigned int duty_cycle);
int avalon_pwm_enable(unsigned int address);
int avalon_pwm_disable(unsigned int address);
int avalon_pwm_change_duty_cycle(unsigned int address, unsigned int
duty_cycle);

//Codis de retorn
#define AVALON_PWM_OK                                              0
#define AVALON_PWM_DUTY_CYCLE_GREATER_THAN_CLOCK_CYCLE_ERROR      -1
#define AVALON_PWM_ENABLED_CONFIRMATION_ERROR                     -2
#define AVALON_PWM_DISABLED_CONFIRMATION_ERROR                    -3

//Constants
#define AVALON_PWM_ENABLED  1
#define AVALON_PWM_DISABLED 0
```

1.2.3 HAL/SRC (avalon_pwm_routines.c)

```
#include "avalon_pwm_routines.h"


int avalon_pwm_init(unsigned int address, unsigned int clock_divider,
unsigned int duty_cycle)
{
       //El registre que conté el cicle de treball ha de ser més petit o
       //igual que el divisor del rellotge
       if(duty_cycle > clock_divider)
       {
              return AVALON_PWM_DUTY_CYCLE_GREATER_THAN_CLOCK_CYCLE_ERROR;
       }
       else
       {
              IOWR_AVALON_PWM_CLOCK_DIVIDER(address, clock_divider - 1);
              IOWR_AVALON_PWM_DUTY_CYCLE(address, duty_cycle);
       }
       return AVALON_PWM_OK;
}

int avalon_pwm_enable(unsigned int address)
{
  IOWR_AVALON_PWM_ENABLE(address, AVALON_PWM_ENABLE_MSK);

  //Confirma que el PWM està activat
  if(IORD_AVALON_PWM_ENABLE(address) != AVALON_PWM_ENABLED)
  {
       return AVALON_PWM_ENABLED_CONFIRMATION_ERROR;
  }
  return AVALON_PWM_OK;
```

```
}

int avalon_pwm_disable(unsigned int address)
{
  IOWR_AVALON_PWM_ENABLE(address, ~AVALON_PWM_ENABLE_MSK);


  //Confirma que el PWM està desactivat
  if(IORD_AVALON_PWM_ENABLE(address) != AVALON_PWM_DISABLED)
  {
       return AVALON_PWM_DISABLED_CONFIRMATION_ERROR;
  }
  return AVALON_PWM_OK;
}


int avalon_pwm_change_duty_cycle(unsigned int address, unsigned int
duty_cycle)
{

       //El registre que conté el cicle de treball ha de ser més petit o
       //igual que el divisor del rellotge
       if(duty_cycle > IORD_AVALON_PWM_CLOCK_DIVIDER(address))
       {
              return AVALON_PWM_DUTY_CYCLE_GREATER_THAN_CLOCK_CYCLE_ERROR;
       }
       else
       {
              IOWR_AVALON_PWM_DUTY_CYCLE(address, duty_cycle);
       }
       return AVALON_PWM_OK;
}
```

2. *IP CORE:* I2C

2.1 I2C Hardware

A continuació es detalla el codi utilitzat a nivell hardware. Les seccions en les que està dividit són a nivell de *bit*, *byte*, interfície i un adaptador amb la interfície del bus Avalon, que a més simula els *buffer tri-state*, per a la bidireccionalitat dels dos senyals (SCL i SDA).

2.1.1 Fitxer del control de bits (i2c_master_bit_ctrl.vhd)

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

entity i2c_master_bit_ctrl is
port (
       clk    : in std_logic;
       rst    : in std_logic;
       nReset : in std_logic;
       ena    : in std_logic;                    //Senyal d'enable
       clk_cnt : in unsigned(15 downto 0);       //Valor del rellotge
       cmd     : in std_logic_vector(3 downto 0);
       cmd_ack : out std_logic;                  //Ack comanda
       busy    : out std_logic;                  //Bus ocupat
       al      : out std_logic;                  //Arbitratge perdut
       din  : in std_logic;
       dout : out std_logic;
       //Línies I2C
       scl_i   : in std_logic;  //Línia d'entrada del rellotge
       scl_o   : out std_logic; //Línia de sortida del rellotge
       scl_oen : out std_logic; //Línia de sortida d'enable. Activa a baixa
       sda_i   : in std_logic;  //Línia d'entrada de dades
       sda_o   : out std_logic; //Línia de sortida de dades
       sda_oen : out std_logic  //Línia de sortida d'enable. Activa a baixa
       );
end entity i2c_master_bit_ctrl;

architecture structural of i2c_master_bit_ctrl is
       constant I2C_CMD_NOP    : std_logic_vector(3 downto 0) := "0000";
       constant I2C_CMD_START  : std_logic_vector(3 downto 0) := "0001";
       constant I2C_CMD_STOP   : std_logic_vector(3 downto 0) := "0010";
```

```
       constant I2C_CMD_READ   : std_logic_vector(3 downto 0) := "0100";
       constant I2C_CMD_WRITE  : std_logic_vector(3 downto 0) := "1000";

       type states is (idle, start_a, start_b, start_c, start_d, start_e,
       stop_a, stop_b, stop_c, stop_d, rd_a, rd_b, rd_c, rd_d, wr_a, wr_b,
wr_c, wr_d);
       signal c_state : states;
       signal iscl_oen, isda_oen : std_logic;
       signal sda_chk            : std_logic;
       signal dscl_oen           : std_logic;
       signal sSCL, sSDA         : std_logic;
       signal clk_en, slave_wait : std_logic;
       signal ial                : std_logic;
       signal cnt : unsigned(15 downto 0) := clk_cnt;
       signal cnt : unsigned(15 downto 0);

begin
//Si l'esclau no està llest pot esperar posant el senyal SCL a baixa

       process (clk)
       begin
          if (clk'event and clk = '1') then
            dscl_oen <= iscl_oen;
           end if;
       end process;
       slave_wait <= dscl_oen and not sSCL;

       //Generar el senyal de rellotge
       gen_clken: process(clk, nReset)
       begin
           if (nReset = '0') then
             cnt    <= (others => '0');
             clk_en <= '1';
           elsif (clk'event and clk = '1') then
             if (rst = '1') then
               cnt    <= (others => '0');
               clk_en <= '1';
             elsif ( (cnt = 0) or (ena = '0') ) then
               cnt    <= clk_cnt;
               clk_en <= '1';
             elsif (slave_wait = '1') then
               cnt    <= cnt;
               clk_en <= '0';
             else
               cnt    <= cnt -1;
               clk_en <= '0';
             end if;
           end if;
       end process gen_clken;


       //Generar el controlador d'estats del bus
       bus_status_ctrl: block
         signal dSCL, dSDA          : std_logic;  //Fa esperar SCL i SDA
         signal sta_condition       : std_logic;  //Inici detectat
         signal sto_condition       : std_logic;  //Stop detectat
         signal cmd_stop            : std_logic;  //Comanda de Stop
         signal ibusy               : std_logic;  //Senyal interna d'ocupat
       begin
           //Sincronitza les entrades SDA i SCL
           synch_scl_sda: process(clk, nReset)
           begin
```

```
    if (nReset = '0') then
      sSCL <= '1';
      sSDA <= '1';

      dSCL <= '1';
      dSDA <= '1';
    elsif (clk'event and clk = '1') then
      if (rst = '1') then
        sSCL <= '1';
        sSDA <= '1';

        dSCL <= '1';
        dSDA <= '1';
      else
        sSCL <= scl_i;
        sSDA <= sda_i;

        dSCL <= sSCL;
        dSDA <= sSDA;
      end if;
    end if;
end process synch_SCL_SDA;

detect_sta_sto: process(clk, nReset)
begin
    if (nReset = '0') then
      sta_condition <= '0';
      sto_condition <= '0';
    elsif (clk'event and clk = '1') then
      if (rst = '1') then
        sta_condition <= '0';
        sto_condition <= '0';
      else
        sta_condition <= (not sSDA and dSDA) and sSCL;
        sto_condition <= (sSDA and not dSDA) and sSCL;
      end if;
    end if;
end process detect_sta_sto;

//Genera el senyal bus i2c ocupat
gen_busy: process(clk, nReset)
begin
    if (nReset = '0') then
      ibusy <= '0';
    elsif (clk'event and clk = '1') then
      if (rst = '1') then
        ibusy <= '0';
      else
        ibusy <= (sta_condition or ibusy) and not sto_condition;
      end if;
    end if;
end process gen_busy;
busy <= ibusy;


//Generació d'arbitratge perdut. Es perd quan:
//Els bus i2c està a baixa i el master posa SDA a alta
//Senyal de Stop detectada quan no s'esperava
gen_al: process(clk, nReset)
begin
  if (nReset = '0') then
    cmd_stop  <= '0';
```

```
              ial       <= '0';
            elsif (clk'event and clk = '1') then
              if (rst = '1') then
                cmd_stop  <= '0';
                ial       <= '0';
              else
                if (clk_en = '1') then
                  if (cmd = I2C_CMD_STOP) then
                    cmd_stop <= '1';
                  else
                    cmd_stop <= '0';
                  end if;
                end if;

                if (c_state = idle) then
                  ial <= (sda_chk and not sSDA and isda_oen);
                else
                  ial <= (sda_chk and not sSDA and isda_oen) or
(sto_condition and not cmd_stop);
                end if;

              end if;
            end if;
          end process gen_al;
          al <= ial;

          //Genera el senyal dout, el guarda i fa pujar el senyal SCL
          gen_dout: process(clk)
          begin
            if (clk'event and clk = '1') then
              if (sSCL = '1' and dSCL = '0') then
                dout <= sSDA;
              end if;
            end if;
          end process gen_dout;
      end block bus_status_ctrl;


      //Generació de la màquina d'estats
      nxt_state_decoder : process (clk, nReset, c_state, cmd)
      begin
          if (nReset = '0') then
            c_state  <= idle;
            cmd_ack  <= '0';
            iscl_oen <= '1';
            isda_oen <= '1';
            sda_chk  <= '0';
          elsif (clk'event and clk = '1') then
            if (rst = '1' or ial = '1') then
              c_state  <= idle;
              cmd_ack  <= '0';
              iscl_oen <= '1';
              isda_oen <= '1';
              sda_chk  <= '0';
            else
              cmd_ack <= '0'; //NACK per defecte

              if (clk_en = '1') then
                case (c_state) is
                   when idle =>
                      case cmd is
                        when I2C_CMD_START => c_state <= start_a;
```

```
     when I2C_CMD_STOP  => c_state <= stop_a;
     when I2C_CMD_WRITE => c_state <= wr_a;
     when I2C_CMD_READ  => c_state <= rd_a;
     when others        => c_state <= idle;
end case;

   iscl_oen <= iscl_oen; //SCL manté l'estat
   isda_oen <= isda_oen; //SDA manté l'estat
   sda_chk  <= '0';      //No comprovació SDA

//Inici
when start_a =>
   c_state  <= start_b;
   iscl_oen <= iscl_oen;
   isda_oen <= '1';      //SDA a alta
   sda_chk  <= '0';      //No comprovació SDA

when start_b =>
   c_state  <= start_c;
   iscl_oen <= '1';      //SCL a alta
   isda_oen <= '1';      //Mantenir SDA a alta
   sda_chk  <= '0';      //No comprovació SDA

when start_c =>
   c_state  <= start_d;
   iscl_oen <= '1';      //SCL a alta
   isda_oen <= '0';      //SDA a baixa
   sda_chk  <= '0';      //No comprovació SDA

when start_d =>
   c_state  <= start_e;
   iscl_oen <= '1';      //SCL a alta
   isda_oen <= '0';      //SDA a baixa
   sda_chk  <= '0';      //No comprovació SDA

when start_e =>
   c_state  <= idle;
   cmd_ack  <= '1';     //Comanda completada
   iscl_oen <= '0';     //SCL a baixa
   isda_oen <= '0';     //SDA a baixa
   sda_chk  <= '0';     //No comprovació SDA

//Stop
when stop_a =>
   c_state  <= stop_b;
   iscl_oen <= '0';     //SCL a baixa
   isda_oen <= '0';     //SDA a baixa
   sda_chk  <= '0';     //No comprovació SDA

when stop_b =>
   c_state  <= stop_c;
   iscl_oen <= '1';     //SCL a alta
   isda_oen <= '0';     //SDA a baixa
   sda_chk  <= '0';     //No comprovació SDA

when stop_c =>
   c_state  <= stop_d;
   iscl_oen <= '1';    //SCL a alta
   isda_oen <= '0';    //SDA a baixa
   sda_chk  <= '0';    //No comprovació SDA

when stop_d =>
```

```
            c_state  <= idle;
            cmd_ack  <= '1';   //Comanda completada
            iscl_oen <= '1';   //SCL a alta
            isda_oen <= '1';   //SDA a alta
            sda_chk  <= '0';   //No comprovació SDA

         //Lectura
         when rd_a =>
            c_state  <= rd_b;
            iscl_oen <= '0';   //SCL a baixa
            isda_oen <= '1';   //SDA a alta
            sda_chk  <= '0';   //No comprovació SDA

         when rd_b =>
            c_state  <= rd_c;
            iscl_oen <= '1';   //SCL a alta
            isda_oen <= '1';   //SDA a alta
            sda_chk  <= '0';   //No comprovació SDA

         when rd_c =>
            c_state  <= rd_d;
            iscl_oen <= '1';   //SCL a alta
            isda_oen <= '1';   //SDA a alta
            sda_chk  <= '0';   //No comprovació SDA

         when rd_d =>
            c_state  <= idle;
            cmd_ack  <= '1';   //Comanda completada
            iscl_oen <= '0';   //SCL a baixa
            isda_oen <= '1';   //SDA a alta
            sda_chk  <= '0';   //No comprovació SDA

         //Escriptura
         when wr_a =>
            c_state  <= wr_b;
            iscl_oen <= '0';   //SCL a baixa
            isda_oen <= din;   //Activa SDA
            sda_chk  <= '0';   //No comprovació SDA

         when wr_b =>
            c_state  <= wr_c;
            iscl_oen <= '1';   //SCL a alta
            isda_oen <= din;   //Guarda SDA
            sda_chk  <= '1';   //Comprova SDA

         when wr_c =>
            c_state  <= wr_d;
            iscl_oen <= '1';   //SCL a alta
            isda_oen <= din;   //Guarda SDA
            sda_chk  <= '1';   //Comprova SDA

         when wr_d =>
            c_state  <= idle;
            cmd_ack  <= '1';   //Comanda completada
            iscl_oen <= '0';   //SCL a baixa
            isda_oen <= din;   //Guarda SDA
            sda_chk  <= '0';   //No comprovació SDA

         when others =>

      end case;
   end if;
```

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


      //Assignar sortides
      scl_o   <= '0';
      scl_oen <= iscl_oen;
      sda_o   <= '0';
      sda_oen <= isda_oen;
end architecture structural;
```

2.1.2 Fitxer del control de bytes (i2c_master_byte_ctrl.vhd)

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

entity i2c_master_byte_ctrl is
      port (
            clk    : in std_logic;
            rst    : in std_logic;
            nReset : in std_logic;
            ena    : in std_logic;
            clk_cnt : in unsigned(15 downto 0);

            //Senyals d'entrada
            start,
            stop,
            read,
            write,
            ack_in : std_logic;
            din    : in std_logic_vector(7 downto 0);

            //Senyals de sortida
            cmd_ack  : out std_logic;
            ack_out  : out std_logic;
            i2c_busy : out std_logic;
            i2c_al   : out std_logic;
            dout     : out std_logic_vector(7 downto 0);

            //Línies I2C
            scl_i   : in std_logic;
            scl_o   : out std_logic;
            scl_oen : out std_logic;
            sda_i   : in std_logic;
            sda_o   : out std_logic;
            sda_oen : out std_logic
      );
end entity i2c_master_byte_ctrl;

architecture structural of i2c_master_byte_ctrl is
      component i2c_master_bit_ctrl is
      port (
            clk    : in std_logic;
            rst    : in std_logic;
            nReset : in std_logic;
```

```
            ena    : in std_logic;
            clk_cnt : in unsigned(15 downto 0);
            cmd     : in std_logic_vector(3 downto 0);
            cmd_ack : out std_logic;
            busy    : out std_logic;
            al      : out std_logic;

            din  : in std_logic;
            dout : out std_logic;

            //Línies I2C
            scl_i   : in std_logic;
            scl_o   : out std_logic;
            scl_oen : out std_logic;
            sda_i   : in std_logic;
            sda_o   : out std_logic;
            sda_oen : out std_logic
      );
      end component i2c_master_bit_ctrl;

      //Comandes per la controladora de bits
      constant I2C_CMD_NOP      : std_logic_vector(3 downto 0) := "0000";
      constant I2C_CMD_START    : std_logic_vector(3 downto 0) := "0001";
      constant I2C_CMD_STOP      : std_logic_vector(3 downto 0) := "0010";
      constant I2C_CMD_READ      : std_logic_vector(3 downto 0) := "0100";
      constant I2C_CMD_WRITE    : std_logic_vector(3 downto 0) := "1000";

      //Senyals per a la controladora de bits
      signal core_cmd : std_logic_vector(3 downto 0);
      signal core_ack, core_txd, core_rxd : std_logic;
      signal al : std_logic;

      //Senyals pels registres
      signal sr : std_logic_vector(7 downto 0);
      signal shift, ld : std_logic;

      //Senyals per la màquina d'estats
      signal go, host_ack : std_logic;
      signal dcnt : unsigned(2 downto 0);
      signal cnt_done : std_logic;

begin
      bit_ctrl: i2c_master_bit_ctrl port map(
            clk     => clk,
            rst     => rst,
            nReset  => nReset,
            ena     => ena,
            clk_cnt => clk_cnt,
            cmd     => core_cmd,
            cmd_ack => core_ack,
            busy    => i2c_busy,
            al      => al,
            din     => core_txd,
            dout    => core_rxd,
            scl_i   => scl_i,
            scl_o   => scl_o,
            scl_oen => scl_oen,
            sda_i   => sda_i,
            sda_o   => sda_o,
            sda_oen => sda_oen
      );
      i2c_al <= al;
```

```
      cmd_ack <= host_ack;
      go <= (read or write or stop) and not host_ack;
      dout <= sr;

      //Genera el registre
      shift_register: process(clk, nReset)
      begin
          if (nReset = '0') then
            sr <= (others => '0');
          elsif (clk'event and clk = '1') then
            if (rst = '1') then
              sr <= (others => '0');
            elsif (ld = '1') then
              sr <= din;
            elsif (shift = '1') then
              sr <= (sr(6 downto 0) & core_rxd);
            end if;
          end if;
      end process shift_register;

      //Genera el comptador de dades
      data_cnt: process(clk, nReset)
      begin
          if (nReset = '0') then
            dcnt <= (others => '0');
          elsif (clk'event and clk = '1') then
            if (rst = '1') then
              dcnt <= (others => '0');
            elsif (ld = '1') then
              dcnt <= (others => '1');
            elsif (shift = '1') then
              dcnt <= dcnt -1;
            end if;
          end if;
      end process data_cnt;

      cnt_done <= '1' when (dcnt = 0) else '0';

      //Màquina d'estats
      statemachine : block
          type states is (st_idle, st_start, st_read, st_write, st_ack,
st_stop);
          signal c_state : states;
      begin
          nxt_state_decoder: process(clk, nReset)
          begin
              if (nReset = '0') then
                core_cmd <= I2C_CMD_NOP;
                core_txd <= '0';
                shift    <= '0';
                ld       <= '0';
                host_ack <= '0';
                c_state  <= st_idle;
                ack_out  <= '0';
              elsif (clk'event and clk = '1') then
                if (rst = '1' or al = '1') then
                  core_cmd <= I2C_CMD_NOP;
                  core_txd <= '0';
                  shift    <= '0';
                  ld       <= '0';
                  host_ack <= '0';
                  c_state  <= st_idle;
```

```
  ack_out  <= '0';
else
  //Inicialització dels senyals de reset
  core_txd <= sr(7);
  shift    <= '0';
  ld       <= '0';
  host_ack <= '0';

  case c_state is
    when st_idle =>
       if (go = '1') then
         if (start = '1') then
           c_state  <= st_start;
           core_cmd <= I2C_CMD_START;
         elsif (read = '1') then
           c_state  <= st_read;
           core_cmd <= I2C_CMD_READ;
         elsif (write = '1') then
           c_state  <= st_write;
           core_cmd <= I2C_CMD_WRITE;
         else //Stop
           c_state  <= st_stop;
           core_cmd <= I2C_CMD_STOP;
         end if;

         ld <= '1';
       end if;

    when st_start =>
       if (core_ack = '1') then
         if (read = '1') then
           c_state  <= st_read;
           core_cmd <= I2C_CMD_READ;
         else
           c_state  <= st_write;
           core_cmd <= I2C_CMD_WRITE;
         end if;

         ld <= '1';
       end if;

    when st_write =>
       if (core_ack = '1') then
         if (cnt_done = '1') then
           c_state  <= st_ack;
           core_cmd <= I2C_CMD_READ;
         else
           c_state  <= st_write;
           core_cmd <= I2C_CMD_WRITE;
           shift    <= '1';
         end if;
       end if;

    when st_read =>
       if (core_ack = '1') then
         if (cnt_done = '1') then
           c_state  <= st_ack;
           core_cmd <= I2C_CMD_WRITE;
         else
           c_state  <= st_read;
           core_cmd <= I2C_CMD_READ;
         end if;
```

```
                        shift    <= '1';
                        core_txd <= ack_in;
                     end if;

                  when st_ack =>
                     if (core_ack = '1') then
                        if (stop = '1') then
                          c_state  <= st_stop;
                          core_cmd <= I2C_CMD_STOP;
                        else
                          c_state  <= st_idle;
                          core_cmd <= I2C_CMD_NOP;
                          host_ack <= '1';
                        end if;
                        ack_out  <= core_rxd;
                        core_txd <= '1';
                     else
                        core_txd <= ack_in;
                     end if;

                  when st_stop =>
                     if (core_ack = '1') then
                        c_state  <= st_idle;
                        core_cmd <= I2C_CMD_NOP;
                        host_ack <= '1';
                     end if;

                  when others =>
                     c_state  <= st_idle;
                     core_cmd <= I2C_CMD_NOP;
                     report ("Byte controller ha entrat en un estat
il·legal.");

                end case;

              end if;
            end if;
        end process nxt_state_decoder;

    end block statemachine;

end architecture structural;
```

2.1.3 Fitxer d'interfície d'alt nivell (i2c_master_top.vhd)

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

entity i2c_master_top is
    generic(
          ARST_LVL : std_logic := '0'
    );
    port (
          wb_clk_i  : in  std_logic;
          wb_rst_i  : in  std_logic := '0';
          arst_i    : in  std_logic := not ARST_LVL;
```

```
            wb_adr_i  : in  unsigned(2 downto 0);
            wb_dat_i  : in  std_logic_vector(7 downto 0);
            wb_dat_o  : out std_logic_vector(7 downto 0);
            wb_we_i   : in  std_logic;
            wb_stb_i  : in  std_logic;
            wb_cyc_i  : in  std_logic;
            wb_ack_o  : out std_logic;
            wb_inta_o : out std_logic;                    -

            //Línies I2C
            scl_pad_i     : in  std_logic;
            scl_pad_o     : out std_logic;
            scl_padoen_o  : out std_logic;
            sda_pad_i     : in  std_logic;
            sda_pad_o     : out std_logic;
            sda_padoen_o  : out std_logic
      );
end entity i2c_master_top;

architecture structural of i2c_master_top is
      component i2c_master_byte_ctrl is
      port (
            clk    : in std_logic;
            rst    : in std_logic;
            nReset : in std_logic;
            ena    : in std_logic;
            clk_cnt : in unsigned(15 downto 0);

            //Senyals d'entrada
            start,
            stop,
            read,
            write,
            ack_in : std_logic;
            din    : in std_logic_vector(7 downto 0);

            //Senyals de sortida
            cmd_ack  : out std_logic;
            ack_out  : out std_logic;
            i2c_busy : out std_logic;
            i2c_al   : out std_logic;
            dout     : out std_logic_vector(7 downto 0);

            //Línies I2C
            scl_i   : in std_logic;
            scl_o   : out std_logic;
            scl_oen : out std_logic;
            sda_i   : in std_logic;
            sda_o   : out std_logic;
            sda_oen : out std_logic
      );
      end component i2c_master_byte_ctrl;

      //Registres
      signal prer : unsigned(15 downto 0);
      signal ctr  : std_logic_vector(7 downto 0);
      signal txr  : std_logic_vector(7 downto 0);
      signal rxr  : std_logic_vector(7 downto 0);
      signal cr   : std_logic_vector(7 downto 0);
      signal sr   : std_logic_vector(7 downto 0);
```

```
        //Senyal interna de reset
        signal rst_i : std_logic;

        //Accés d'escriptura al bus
        signal wb_wacc : std_logic;

        //Senyal interna d' ACK
        signal iack_o : std_logic;

        //Senyal de comanda completada i neteja del registre
        signal done : std_logic;

        //Senyals de control dels registres
        signal sta, sto, rd, wr, ack, iack : std_logic;

        signal core_en : std_logic;
        signal ien     : std_logic;

        //Senyals d'estat del registre
        signal irxack, rxack : std_logic;
        signal tip           : std_logic;
        signal irq_flag      : std_logic;
        signal i2c_busy      : std_logic;
        signal i2c_al, al    : std_logic;
begin
        //Genera la senyal interna de reset
        rst_i <= arst_i xor ARST_LVL;

        //Genera la senyal de sortida de ACK
        gen_ack_o : process(wb_clk_i)
        begin
            if (wb_clk_i'event and wb_clk_i = '1') then
              iack_o <= wb_cyc_i and wb_stb_i and not iack_o;
            end if;
        end process gen_ack_o;
        wb_ack_o <= iack_o;


        //Genera el senyal d'accés a escriptura
        wb_wacc <= wb_cyc_i and wb_stb_i and wb_we_i;

        assign_dato : process(wb_clk_i)
        begin
            if (wb_clk_i'event and wb_clk_i = '1') then
              case wb_adr_i is
                when "000"  => wb_dat_o <= std_logic_vector(prer( 7 downto
0));
                when "001"  => wb_dat_o <= std_logic_vector(prer(15 downto
8));
                when "010"  => wb_dat_o <= ctr;
                when "011"  => wb_dat_o <= rxr;
                when "100"  => wb_dat_o <= sr;
                 //Senyals per depurar. No documentats.
                when "101"  => wb_dat_o <= txr;
                when "110"  => wb_dat_o <= cr;
                when "111"  => wb_dat_o <= (others => '0');
                when others => wb_dat_o <= (others => 'X');
              end case;
            end if;
        end process assign_dato;
```

```
//Generació dels registres
gen_regs: process(rst_i, wb_clk_i)
begin
    if (rst_i = '0') then
      prer <= (others => '1');
      ctr  <= (others => '0');
      txr  <= (others => '0');
    elsif (wb_clk_i'event and wb_clk_i = '1') then
      if (wb_rst_i = '1') then
        prer <= (others => '1');
        ctr  <= (others => '0');
        txr  <= (others => '0');
      elsif (wb_wacc = '1') then
        case wb_adr_i is
           when "000" => prer( 7 downto 0) <= unsigned(wb_dat_i);
           when "001" => prer(15 downto 8) <= unsigned(wb_dat_i);
           when "010" => ctr               <= wb_dat_i;
           when "011" => txr               <= wb_dat_i;
           when "100" => null;
           when others =>
              prer <= (others => 'X');
              ctr  <= (others => 'X');
              txr  <= (others => 'X');
        end case;
      end if;
    end if;
end process gen_regs;

//Genera el registre de comandes
gen_cr: process(rst_i, wb_clk_i)
begin
    if (rst_i = '0') then
        cr <= (others => '0');
    elsif (wb_clk_i'event and wb_clk_i = '1') then
        if (wb_rst_i = '1') then
            cr <= (others => '0');
        elsif (wb_wacc = '1') then
            if ( (core_en = '1') and (wb_adr_i = 4) ) then
               cr <= wb_dat_i;
            end if;
        else
            if (done = '1' or i2c_al = '1') then
                cr(7 downto 4) <= (others => '0');
            end if;

            cr(2 downto 1) <= (others => '0');
            cr(0) <= '0';
        end if;
    end if;
end process gen_cr;

//Decodificació comandes de registres
sta  <= cr(7);
sto  <= cr(6);
rd   <= cr(5);
wr   <= cr(4);
ack  <= cr(3);
iack <= cr(0);

//Decodificació del control dels registres
core_en <= ctr(7);
ien     <= ctr(6);
```

```
byte_ctrl: i2c_master_byte_ctrl port map (
       clk      => wb_clk_i,
       rst      => wb_rst_i,
       nReset   => rst_i,
       ena      => core_en,
       clk_cnt  => prer,
       start    => sta,
       stop     => sto,
       read     => rd,
       write    => wr,
       ack_in   => ack,
       i2c_busy => i2c_busy,
       i2c_al   => i2c_al,
       din      => txr,
       cmd_ack  => done,
       ack_out  => irxack,
       dout     => rxr,
       scl_i    => scl_pad_i,
       scl_o    => scl_pad_o,
       scl_oen  => scl_padoen_o,
       sda_i    => sda_pad_i,
       sda_o    => sda_pad_o,
       sda_oen  => sda_padoen_o
);


st_irq_block : block
begin
    //Genera l'estat del registre de bits
    gen_sr_bits: process (wb_clk_i, rst_i)
    begin
        if (rst_i = '0') then
          al       <= '0';
          rxack    <= '0';
          tip      <= '0';
          irq_flag <= '0';
        elsif (wb_clk_i'event and wb_clk_i = '1') then
          if (wb_rst_i = '1') then
            al       <= '0';
            rxack    <= '0';
            tip      <= '0';
            irq_flag <= '0';
          else
            al       <= i2c_al or (al and not sta);
            rxack    <= irxack;
            tip      <= (rd or wr);
            irq_flag <= (done or i2c_al or irq_flag) and not iack;
          end if;
        end if;
    end process gen_sr_bits;

    //Genera el senyal d'interrupció
    gen_irq: process (wb_clk_i, rst_i)
    begin
        if (rst_i = '0') then
          wb_inta_o <= '0';
        elsif (wb_clk_i'event and wb_clk_i = '1') then
          if (wb_rst_i = '1') then
            wb_inta_o <= '0';
          else
```

```
                wb_inta_o <= irq_flag and ien;
              end if;
            end if;
        end process gen_irq;

        //Assignació de l'estat dels bits del registre
        sr(7)          <= rxack;
        sr(6)          <= i2c_busy;
        sr(5)          <= al;
        sr(4 downto 2) <= (others => '0');
        sr(1)          <= tip;
        sr(0)          <= irq_flag;
    end block;

end architecture structural;
```

2.1.4 Fitxer d'interfície amb el bus Avalon (oc_i2c_master_top.vhd)

```
entity oc_i2c_master_top is
   port(
       //Senyals del bus Avalon
       address:       in unsigned(2 downto 0);
       readdata:      out std_logic_vector(7 downto 0);
       writedata:     in std_logic_vector(7 downto 0);
       write:         in std_logic;
       chipselect:    in std_logic;
       clk:           in std_logic;
       reset_n:       in std_logic;
       irq:           out std_logic;
       waitrequest_n: out std_logic;

       //Sortides I2C
       scl: inout std_logic;
       sda: inout std_logic
    );
end oc_i2c_master_top;

architecture bhv of oc_i2c_master_top is

component i2c_master_top is
generic(
 ARST_LVL : std_logic
);
port (

 wb_clk_i  : in  std_logic;
 wb_rst_i  : in  std_logic;
 arst_i    : in  std_logic;
 wb_adr_i  : in  unsigned(2 downto 0);
 wb_dat_i  : in  std_logic_vector(7 downto 0);
 wb_dat_o  : out std_logic_vector(7 downto 0);
 wb_we_i   : in  std_logic;
 wb_stb_i  : in  std_logic;
 wb_cyc_i  : in  std_logic;
 wb_ack_o  : out std_logic;
 wb_inta_o : out std_logic;
```

```
 //Línies I2C
 scl_pad_i     : in  std_logic;
 scl_pad_o     : out std_logic;
 scl_padoen_o  : out std_logic;
 sda_pad_i     : in  std_logic;
 sda_pad_o     : out std_logic;
 sda_padoen_o  : out std_logic
);
end component;

signal scl_pad_i     : std_logic;
signal scl_pad_o     : std_logic;
signal scl_padoen_o  : std_logic;
signal sda_pad_i     : std_logic;
signal sda_pad_o     : std_logic;
signal sda_padoen_o  : std_logic;
begin
   i2c_top : i2c_master_top
       generic map ( ARST_LVL => '0')
       port map (
           wb_adr_i  => address,
           wb_dat_i  => writedata,
           wb_dat_o  => readdata,
           wb_we_i   => write,
           wb_stb_i  => chipselect,
           wb_cyc_i  => chipselect,
           wb_inta_o => irq,
           wb_clk_i  => clk,
           wb_ack_o  => waitrequest_n,
           wb_rst_i  => '0',
           arst_i    => reset_n,

           scl_pad_i    => scl_pad_i,
           scl_pad_o    => scl_pad_o,
           scl_padoen_o => scl_padoen_o,
           sda_pad_i    => sda_pad_i,
           sda_pad_o    => sda_pad_o,
           sda_padoen_o => sda_padoen_o
       );

   scl <= scl_pad_o when (scl_padoen_o = '0') else 'Z';
   sda <= sda_pad_o when (sda_padoen_o = '0') else 'Z';
   scl_pad_i <= scl;
   sda_pad_i <= sda;

end architecture bhv;
```

2.2 I2C Software

A continuació es detalla el codi utilitzat a nivell software. A diferència del cas anterior, aquest és un component *master* i per tant, tan sols s'especificaran les capçaleres on es defineixen les *macros* per a poder accedir als registres.

2.2.1 INC (oc_i2c_master.h)

```
//Accés lectura/escriptura
#define OC_I2C_PRER_LO 0x00     //Byte baix de preescalació del registre
#define OC_I2C_PRER_HI 0x01     //Byte alt de preescalació del registre
#define OC_I2C_CTR     0x02     //Control del registre

//Registres de només escriptura
#define OC_I2C_TXR     0x03     //Transmetre el byte del registre
#define OC_I2C_CR      0x04     //Control del registre

//Registres de només lectura
#define OC_I2C_RXR     0x03     //Rebre el byte del registre
#define OC_I2C_SR      0x04     //Estat del registre

//Control del registre
#define OC_I2C_EN (1<<7)        //Bit d'enable del core:
                                //1 - Core està activat
                                //0 - Core està desactivat
#define OC_I2C_IEN (1<<6)       //Bit d'enable d'interrupció
                                //1 – Interrupció activada
                                //0 – Interrupció desactivada
                                //Els altres bits al CR estan reservats

//Bits de comanda del registre
#define OC_I2C_STA (1<<7)       //Regenerar la condició d'inici
#define OC_I2C_STO (1<<6)       //Generar la condició de stop
#define OC_I2C_RD  (1<<5)       //Llegir de l'esclau
#define OC_I2C_WR  (1<<4)       //Escriure a l'esclau
#define OC_I2C_ACK (1<<3)       //ACK de l'esclau
                                //1 - ACK
                                //0 - NACK
#define OC_I2C_IACK (1<<0)      //Interrupció d'ACK

//Estat dels bits del registre
#define OC_I2C_RXACK (1<<7)     //Ack rebut de l'esclau
                                //1 - ACK
                                //0 - NACK
#define OC_I2C_BUSY  (1<<6)     //Bit de bus ocupat
#define OC_I2C_TIP   (1<<1)     //Transferència en progrés
#define OC_I2C_IF    (1<<0)     //Flag d'interrupció

//Bits de test i macros
#define OC_ISSET(reg,bitmask)       ((reg)&(bitmask))
#define OC_ISCLEAR(reg,bitmask)     (!(OC_ISSET(reg,bitmask)))
#define OC_BITSET(reg,bitmask)      ((reg)|(bitmask))
#define OC_BITCLEAR(reg,bitmask)    ((reg)|(~(bitmask)))
#define OC_BITTOGGLE(reg,bitmask)   ((reg)^(bitmask))
#define OC_REGMOVE(reg,value)       ((reg)=(value))
```