DISSENY I SÍNTESI DE SISTEMES DIGITALS

Model funcional I2C

Arnau Quintana Llorens

8 de juny del 2020

Índex

1. Introducció

Un model funcional s'utilitza per verificar la funcionalitat d'un dispositiu; el Màster del I2C en aquest cas. Per provar la funcionalitat es connecten mòduls no sintetitzables que simulin botons, switches, sensors o el control de les entrades del Màster. En la figura de sota tenim una aproximació del nostre disseny.

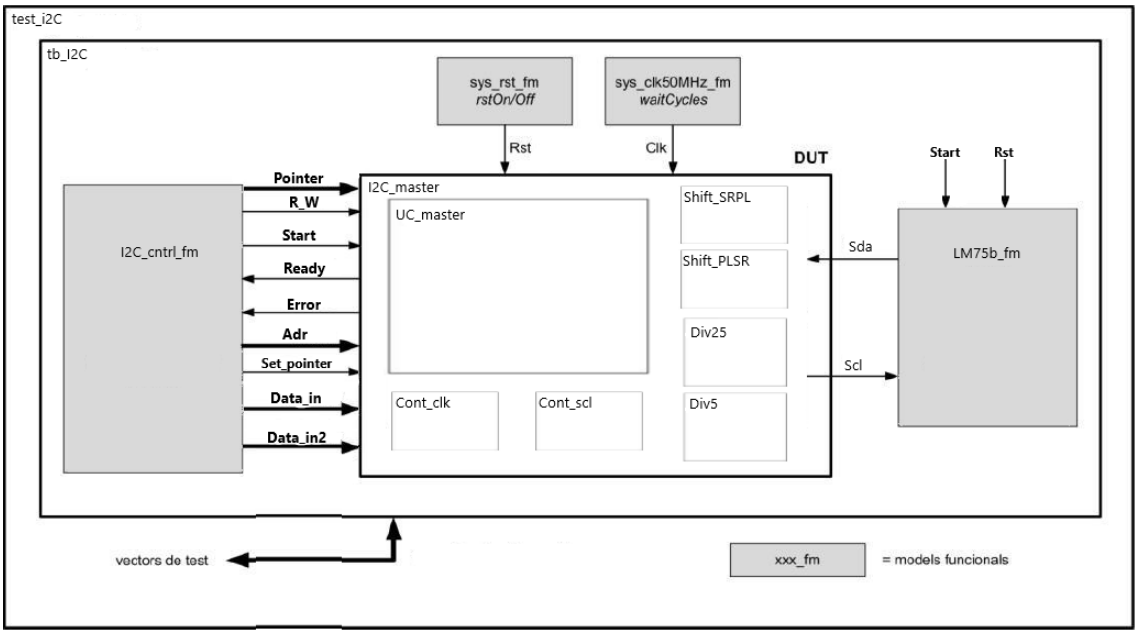


Figura 1. Model funcional de l' I2C.

I2C_cntrl_fm: simula els senyals que s'introduirien externament com les dades o l'adreça del Slave.

sys_rst_fm: simula un botó de reset.

sys_clk50MHz_fm: simula el rellotge de 50MHz de la placa.

LM75b_fm: simula el comportament del sensor davant les demandes del Màster.

I2C_master: únic mòdul sintetitzable. És el que controla la comunicació I2C.

tb_I2C: agrupa tots els mòduls que formen el model funcional.

test_I2C: es crida a tots els mòduls, a la tasca del reset i es decideix quan dura la simulació complerta.

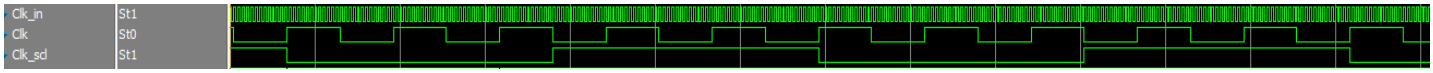


Figura 2. Dominis de rellotge del I2C.

A la figura de sobre trobem els diferents dominis de rellotge de tot el protocol. El primer és de 50MHz que entra al Màster. El segon és de 2MHz amb el que treballa el Màster. El tercer i últim és de 400KHz de la comunicació I2C; és la freqüència a la que treballen la majoria de sensors I2C.

2. Màster

2.1 TOP del Màster

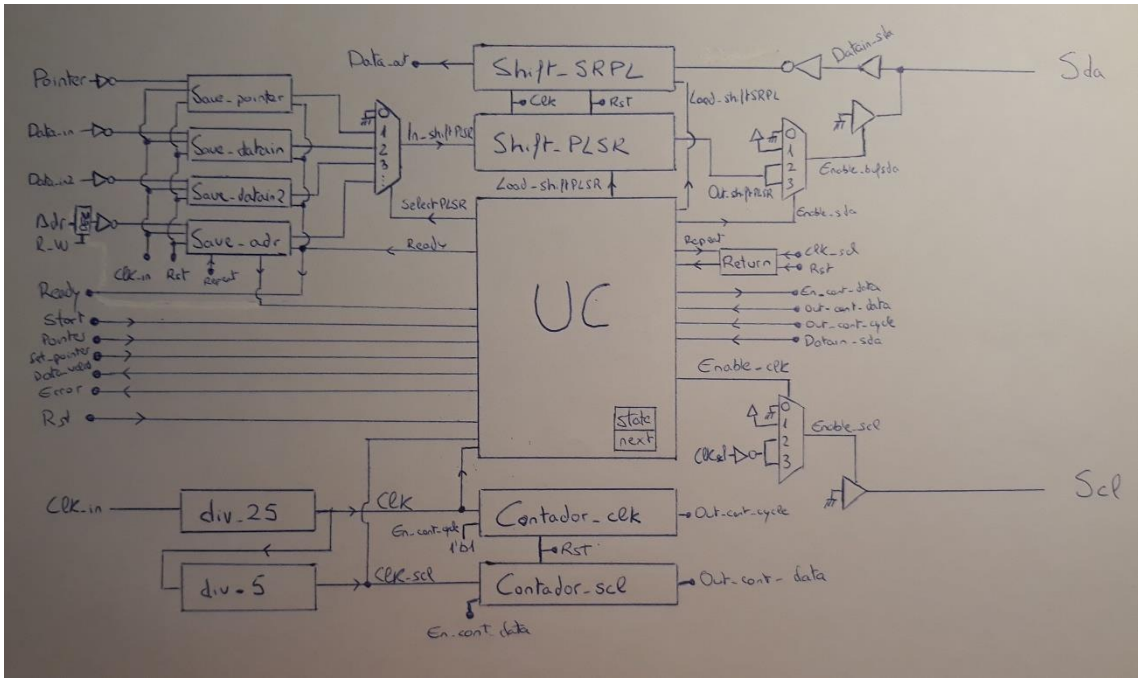


Figura 3. TOP del Màster.

A la figura trobem una aproximació del TOP del Màster. S'aprecien tots els blocs que el formen així com les senyals d'entrada i sortida de cada bloc.

Cada bloc i cada senyal estan explicats en detall al datasheet del dispositiu.

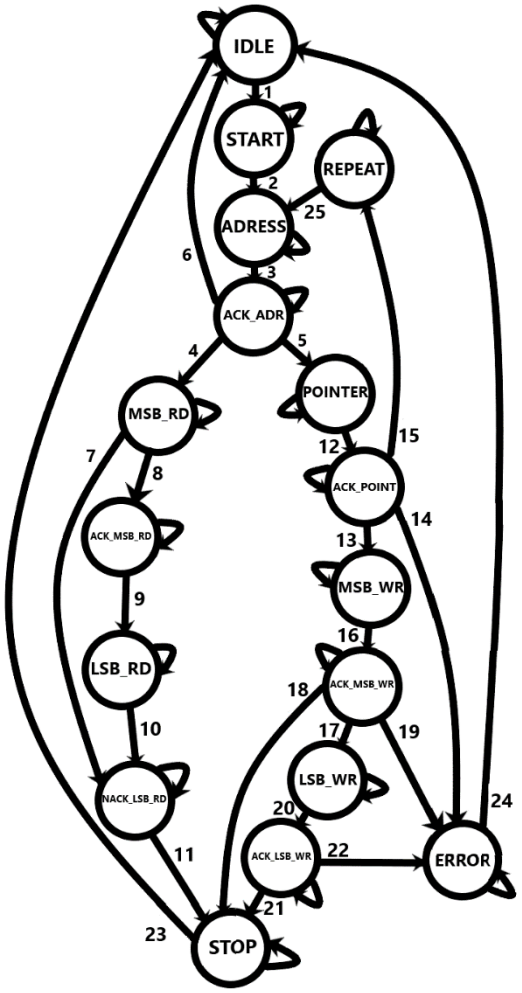
La màquina d'estats de la unitat de control està formada per 17 estats diferents com es veurà al proper apartat. El Màster permet enviar un i dos bytes així com llegir un i dos bytes d'un Slave. També permet fixar un registre del Slave al que volem dirigir-nos sense haver de fer un procés d'escriptura complet.

La màquina avisarà amb banderes de diferents situacions que poden passar com són: errors (rebre un NACK per exemple), moment en que es pot iniciar un nou procés i moment en que les dades a la sortida del mòdul són vàlides per poder tractar amb elles.

La estratègia per enviar dades es composa de varis passos. El primer consisteix en invertir tots els bits de l'entrada de dades. Seguidament, es carreguen al registre de desplaçament tot seleccionant el registre del que ens interessa agafar les dades amb el senyal 'SelectPLSR'. La sortida del registre de desplaçament passa per un MUX que, amb el senyal 'Enable_sda', es decideix quina de de les 4 possibilitats anirà cap al 'Enable' del tri-state. Com sabem, només podem enviar 0, ja que les línies Sda i Scl estan connectades a resistències de 'pullup', per això tenim un 0 a l'entrada del tri-state.

El procés de lectura consisteix en invertir el bit que arriba per Sda i anar carregant les dades a la sortida mitjançant un registre de desplaçament. Quan la màquina posi 'Data_valid' a 1, les dades a la sortida seran vàlides per treballar amb elles.

2.2 Diagrama RTL de la màquina d'estats del UC_Master



Codi	Transició	Condició
1	IDLE → START	Start = 1
2	START → ADRESS	Out_cont_cycle = 1
3	ADRESS → ACK_ADR	Out_cont_data = 8 & Out_cont_cycle = 5
4	ACK_ADR → MSB_RD	Clk_scl = 1 & Datain_sda = 1 & RW = 1
5	ACK_ADR → POINTER	Clk_scl = 1 & Datain_sda = 1 & RW = 0
6	ACK_ADR → IDLE	Clk_scl = 1 & Datain_sda = 0
7	MSB_RD → NACK_LSB_RD	Out_cont_data = 8 & Out_cont_cycle = 1 & Pointer[1:0] = 1
8	MSB_RD → ACK_MSB_RD	Out_cont_data = 8 & Out_cont_cycle = 1 & Pointer[1:0] != 1
9	ACK_MSB_RD → LSB_RD	Out_cont_cycle = 1
10	LSB_RD → NACK_LSB_RD	Out_cont_data = 8 & Out_cont_cycle = 1
11	NACK_LSB_RD → STOP	Out_cont_cycle = 1
12	POINTER → ACK_POINTER	Out_cont_data = 8 && Out_cont_cycle = 5
13	ACK_POINTER → MSB_WR	Clk_scl = 1 & Datain_sda = 0 & Set_pointer = 0
14	ACK_POINTER → ERROR	Clk_scl = 1 & Datain_sda = 1
15	ACK_POINTER → REPEAT	Clk_scl = 1 & Datain_sda = 0 & Set_pointer = 1
16	MSB_WR → ACK_MSB_WR	Out_cont_data = 8 & Out_cont_cycle = 5
17	ACK_MSB_WR → LSB_WR	Clk_scl = 1 & Datain_sda = 0 && Pointer[1] = 1
18	ACK_MSB_WR → STOP	Clk_scl = 1 & Datain_sda = 0 && Pointer[1] = 0
19	ACK_MSB_WR → ERROR	Clk_scl = 1 & Datain_sda = 1
20	LSB_WR → ACK_LSB_WR	Out_cont_data = 8 & Out_cont_cycle = 5
21	ACK_LSB_WR → STOP	Clk_scl = 1 & Datain_sda = 0 & Out_cont_cycle = 3
22	ACK_LSB_WR → ERROR	Clk_scl = 1 & Datain_sda = 1 & Out_cont_cycle = 3
23	STOP → IDLE	Out_cont_cycle = 3
24	ERROR → IDLE	Out_cont_cycle = 5
25	REPEAT → ADRESS	Out_cont_cycle = 3 & Return = 1

Figura 4 i taula 1. Diagrama RTL de la FSM del Màster juntament amb les condicions de les transicions entre estats.

3. Codis verilog

3.1 FMs

3.1.1 I2C_cntrl_fm

```
module I2C_cntr_fm(
output reg [6:0] Adr,
output reg [7:0] Pointer,
output reg Set_pointer,
output reg [7:0] Data_in,
output reg [7:0] Data_in2,
output reg R_W,
output reg Start,
input Error,
input Ready);

initial begin
  // Escriptura de dos bytes i lectura de dos bytes setejant el Pointer

  Set_pointer = 1'b0;         //Per escriure no necessitem el Set_pointer
  Pointer = 8'b00000010;      //Direcció del registre d'un byte
  Data_in = 8'b10011001;      //Primer byte de dades a escriure
  Data_in2 = 8'b00110001;     //Segon byte de dades a escriure
  Adr = 7'b1001101;           //Adreça del Slave
  R_W = 1'b0;                 //Indiquem que volem escriure
  #300 Start = 1'b1;          //Iniciem un procés
  #1000 Start = 1'b0;         //Desactivem Start
  #95000 R_W = 1'b0;          //Un cop acabada l'escriptura s'indica que es vol tornar a escriure per setejar el Pointer
  #5000  Set_pointer = 1'b1; //S'indica que es setejarà el Pointer
  #300 Start = 1'b1;          //S'inicia un procés
  #1000 Start = 1'b0;         //Es desactiva Start
  R_W = 1'b1;                 //S'indica que quan es faci un repeat start es voldrà llegir


  // Escriptura d'un byte i lectura d'un byte setejant el Pointer

  Set_pointer = 1'b0;        //Per escriure no necessitem el Set_pointer
  Pointer = 8'b00000001;     //Direcció del registre d'un byte
  Data_in = 8'b10011001;     //Dades a escriure
  Adr = 7'b1001101;          //Adreça del Slave
  R_W = 1'b0;                //Indiquem que es vol escriure
  #300 Start = 1'b1;         //Indiquem un Start a la màquina
  #1000 Start = 1'b0;        //Quan Ready = 0, desactivem Start
  #72000 R_W = 1'b0;         //Un cop s'ha acabat l'escriptura i Ready = 0, canviem a lectura
  #5000 Set_pointer = 1'b1; //Ara es vol setejar el registre al que dirigir-nos
  #300 Start = 1'b1;         //Iniciem un altre procés
  #1000 Start = 1'b0;        //Desactivem Start
  R_W = 1'b1;                //Indiquem que un cop setejat el Pointer i es faci un repeat start, es voldrà llegir

  //Rst asíncron

  Set_pointer = 1'b0;     //Per escriure no necessitem el Set_pointer
  Pointer = 8'b00000001; //Direcció del registre d'un byte
  Data_in = 8'b10011001; //Dades a escriure
  Adr = 7'b1001101;       //Adreça del Slave
  R_W = 1'b0;             //Indiquem que es vol escriure
  #300 Start = 1'b1;      //Indiquem un Start a la màquina
  #1000 Start = 1'b0;     //Quan Ready = 0, desactivem Start
  #20000 Start = 1'b1;    //Després del Rst fet al mòdul 'test_I2C' tornem a iniciar un procés
  #1000 Start = 1'b0;

  end

endmodule
```

Figura 5. Codi verilog pel al model funcional del mòdul que controla les entrades del Màster.

Cada bloc de codi s'ha d'activar per separat. En verd està comentat què fa cada un.

3.1.2 sys_rst_fm

```
module sys_rst_fm (
    Rst_n
    );

output          Rst_n           ;   // Generated reset
reg             Rst_n           ;

initial
 begin
   Rst_n = 1;
 end

// ----------------------------------------------------------------------------
// Task: sys.rstOn
// Asserts reset (Rst_n=0 & Rst=1)
// ----------------------------------------------------------------------------
task rstOn; begin
    Rst_n=0;
end
endtask // rstOn

// ----------------------------------------------------------------------------
// Task: sys.rstOff
// Deasserts reset (Rst_n=0 & Rst=1)
// ----------------------------------------------------------------------------
task rstOff; begin
    Rst_n=1;
end
endtask // rstOff

endmodule
```

Figura 6. Codi verilog per al model funcional del botó del reset.

Format per dues tasques que s'utilitzen al test com es veurà.

3.1.3 sys_clk50MHz_fm

```
module sys_clk50MHz_fm (
    Clk
    );

`define DATA_SKEW #1

output          Clk             ;   // Generated clock
reg             Clk             ;

// Initialization of all signals and variables
initial begin
    Clk=0;
end

initial begin
    forever begin
        Clk=1;
        #10;
        Clk=0;
        #10;
    end
end


// ---------------------------------------------------------------------------
// Task: sys.waitCycles(<cycles>)
// <cycles>: number of clock positive edges to wait
// ---------------------------------------------------------------------------

task waitCycles;
input [31:0] cycles;
begin
    repeat (cycles)
        @(posedge Clk);
    `DATA_SKEW;
end
endtask // waitCycles

endmodule
```

Figura 7. Codi verilog pel model funcional del rellotge de 50MHz.

La sortida és el rellotge de 50MHz que té la FPGA. A més, s'inclou un tasca que ens serveix per controlar els cicles de rellotge que volem que passin entre dues instruccions.

3.1.4 LM75b_fm

```
module LM75x_fm(inout Sda,
input Scl,
input Rst,
input Start);

  reg [15:0] Temp = 16'b1001100100110001;

  reg Enable_sda = 1'b0;
  bufif1(Sda,1'b0,Enable_sda);

  wire [3:0] Out_cont;
  reg En_cont,Rst_cont;
  Contador_rst #(9) Contador(.En(En_cont),
  .Rst(Rst_cont),
  .Clk(Scl),
  .Out(Out_cont));

  reg R_W;
  reg [1:0] Pointer = 2'b10;//per defecte seria el reg 00 (només RD de 2Bytes), per testejar s'utilitza el 10
  initial begin
  En_cont = 1'b0;
  R_W = 1'bx;
  end

  integer Times;
  always@(posedge Start) begin
    Rst_cont = 1'b0;
    #5 Rst_cont = 1'b1;
    En_cont = 1'b1;
    Times = 0;
  end

  reg Edge_start;
  always@(negedge Sda)
    Edge_start = Scl;

  always@(posedge Edge_start) begin  //quan tenim un repeat Start, es detecta per flanc
    if (Times == 2) begin
      Rst_cont = 1'b0;
      #5 Rst_cont = 1'b1;
    end
  end

//Informa del moment de la simulació en que ens trobem
  always@(posedge Scl)
  if (R_W == 1'b0)begin
    if(Out_cont == 4'b1001)
      Times = Times + 1;
```

```
    else if (Edge_start)
      Times = 0;
    else
      Times = Times;
  end
  else begin
    if(Out_cont == 4'b1000)
      Times = Times + 1;
    else
      Times = Times;
  end

  //ACK  Adress
  always@(posedge Scl)
    if (Times == 0)
      wait(Out_cont == 4'b1000) begin
      #5 R_W = Sda;
      if (R_W == 1'b0) begin
      #1245 Enable_sda = 1'b1;
      #2500 Enable_sda = 1'b0;
      end
      else begin
      #1745 Enable_sda = 1'b1;
      #2500 Enable_sda = 1'b0;
      end
      end

  //Pointer + ACK Pointer
  always@(posedge Scl)
  if (R_W == 1'b0)begin
    if (Times == 1) begin
      wait(Out_cont == 4'b0111)
        Pointer[1] = Sda;            //es guarda el primer bit dels 2 de direcció de registre
      wait(Out_cont == 4'b1000) begin
      #5 Pointer[0] = Sda;           //es guarda el segon bit dels 2 de direcció de registre
      #1250 Enable_sda = 1'b1;
      #2500 Enable_sda = 1'b0;
      end
    end
  end

  //ACK MSB WR
  always@(posedge Scl)
  if (R_W == 1'b0) begin
    if(Times == 2 && R_W == 1'b0)
      wait(Out_cont == 4'b1000) begin
      #1250 Enable_sda = 1'b1;
      #2600 Enable_sda = 1'b0;
      end
  end

  //ACK LSB WR
  always@(posedge Scl)
  if(R_W == 1'b0) begin
    if(Times == 3 && R_W == 1'b0 && Pointer[1] == 1'b1) //només s'entra si s'escriu el reg de 2 bytes
      wait(Out_cont == 4'b1000) begin
      #1250 Enable_sda = 1'b1;
      #2600 Enable_sda = 1'b0;
      end
  end

  //ACK MSB RD + ACK LSB RD
  always@(posedge Scl)
  if (R_W == 1'b1)  begin
    if(Times == 1)
      wait(Out_cont == 4'b1001) begin
      #1750 Enable_sda = Temp[15];
      #2500 Enable_sda = Temp[14];
      #2500 Enable_sda = Temp[13];
      #2500 Enable_sda = Temp[12];
      #2500 Enable_sda = Temp[11];
      #2500 Enable_sda = Temp[10];
      #2500 Enable_sda = Temp[9];
      #2500 Enable_sda = Temp[8];
      #2500 Enable_sda = 1'b0;
      end
    else if(Times == 2 && Pointer[1] == 1'b1) begin  //només s'entra si es llegeix el reg de 2 bytes
      #1750 Enable_sda = Temp[7];
      #2500 Enable_sda = Temp[6];
      #2500 Enable_sda = Temp[5];
      #2500 Enable_sda = Temp[4];
      #2500 Enable_sda = Temp[3];
      #2500 Enable_sda = Temp[2];
      #2500 Enable_sda = Temp[1];
      #2500 Enable_sda = Temp[0];
      #2500 Enable_sda = 1'b0;
    end
  end

endmodule
```

Figura 8. Codi verilog pel model funcional del sensor LM75b.

Aquest mòdul simula la resposta del sensor davant les diferents demandes del Màster. Ha de ser capaç d'enviar els ACK i les dades quan es requereixin.

Per l'enviament de dades s'utilitza la mateixa l'estratègia que al Màster: mitjançant un tri-state amb entrada connectada a terra. Si seguim mirant el codi ens trobem un comptador, s'utilitza per comptar els cicles de rellotge que han passat quan s'inicia una comunicació. Compta fins a 9 ja que són 9 els cicles que dura cada transmissió (dades + ACK).

Seguidament trobem un bloc que ens reseteja l'integer 'Times' i el comtpador quan hi ha un 'Start'. L'integer 'Times' ens ajuda a saber en quin punt de la simulació ens trobem ja que podem estar rebent l'adreça o enviant el byte de dades per exemple.

El bloc que hi ha a continuació s'utilitza per saber quan hi ha un 'repeat start', ja que significa que s'haurà de tornar a agafar el bit de lectura o escriptura per saber si ara es rebran dades o s'han d'enviar. Com veiem, aquesta detecció de flanc, per evitar errors, només es farà servir si 'Times' és igual 2, ja que és l'únic moment en que pot produir-se una repetició. És el moment en que ja s'ha passat per la rebuda de la adreça i del pointer. També estarem en 'Times' = 2 quan s'enviï el segon byte de dades en cas de lectura de 2 bytes, però, com sabem, les transicions a Sda sempre són quan Scl = 0, per tant, no hi ha perill d'una mala detecció d'aquest flanc.

A l'hora de rebre dades només s'agafa el bit de lectura i escriptura i els 2 bits finals del byte de pointer per saber si la transmissió serà de 1 o 2 bytes. Els demés bits no ens interessen per provar el funcionament del Màster.

3.2 Master

```
module Master(input Rst,
input Start,
input [7:0] Data_in,
input [7:0] Data_in2,
input [6:0] Adr,
input [7:0] Pointer,
input Set_pointer,
input R_W,
output Ready,
output Error,
output [7:0] Data_out,
output Data_valid,
output Scl,
inout Sda,
input Clk_in);

  supply0 Gnd;
  supply1 Vcc;

  wire Clk;
  div_25 div25(.clk_in(Clk_in),
  .clk_out(Clk));

  wire Clk_scl;
  div_5 div5(.clk_in(Clk),
  .clk_out(Clk_scl));

  reg [7:0] Save_datain;
  always@(posedge Clk or negedge Rst)
  if (!Rst)
    Save_datain = 8'b0;
  else if(Ready)
    Save_datain = ~Data_in;
  else
    Save_datain = Data_in;

  reg [7:0] Save_datain2;
  always@(posedge Clk or negedge Rst)
   if (!Rst)
     Save_datain2 = 8'b0;
   else if(Ready)
     Save_datain2 = ~Data_in2;
   else
     Save_datain2 = Save_datain2;

  wire Repeat,RW;
  reg [7:0] Save_adr;
  assign RW = ~Save_adr[0];
  always@(posedge Clk or negedge Rst)
    if (!Rst)
      Save_adr = 8'b0;
    else if(Ready)
      Save_adr = ~{Adr,R_W};
    else if(Repeat)
      Save_adr = {Save_adr[7:1],~R_W};
    else
      Save_adr = Save_adr;

  reg [7:0] Save_pointer;
  always@(posedge Clk or negedge Rst)
   if (!Rst)
     Save_pointer = 8'b0;
   else if (Ready)
     Save_pointer = ~Pointer;
   else
     Save_pointer = Save_pointer;

  reg Return = 1'b0;
  always@(posedge Clk_scl or negedge Rst)
   if (!Rst)
     Return = 1'b0;
   else if (Repeat)
     Return = 1'b1;
   else
     Return = 1'b0;

  wire [3:0] Out_cont_data;
  wire En_cont_data;
  Contador_rst Contador_scl(.En(En_cont_data),  //compta els cicles del rellotge del Scl
  .Clk(Clk_scl),
  .Rst(Rst),
  .Out(Out_cont_data));

  wire [3:0] Out_cont_cycle;
  wire En_cont_cycle;
  assign En_cont_cycle = 1'b1;
  Contador_rst #(5)Contador_clk(.En(En_cont_cycle),  //compta els cicles del rellotge del master
  .Rst(Rst),
  .Clk(Clk),
  .Out(Out_cont_cycle));

  wire [7:0] In_shiftPLSR;
  wire Load_shiftPLSR,Out_shiftPLSR;
  wire [2:0] SelectPLSR;
  assign In_shiftPLSR = (SelectPLSR == 3'b000) ? Gnd :
  (SelectPLSR == 3'b001) ? Save_pointer :
```

```
(SelectPLSR == 3'b010) ? Save_datain :
(SelectPLSR == 3'b011) ? Save_datain2 :
(Save_adr);

Shift_PLSR_master Shift_PLSR_master(.In(In_shiftPLSR),
.Load(Load_shiftPLSR),
.Clk(Clk),
.Rst(Rst),
.Out(Out_shiftPLSR));

wire [1:0] Enable_sda;
wire Enable_bufsda;
assign Enable_bufsda = (Enable_sda == 2'b00) ? Gnd : (Enable_sda == 2'b01 ? Vcc : Out_shiftPLSR);
bufifl buf_sda(Sda,Gnd,Enable_bufsda);

wire Enable_scl;
wire [1:0] Enable_clk;
assign Enable_scl = (Enable_clk == 2'b00) ? Gnd : (Enable_clk == 2'b01 ? Vcc : ~Clk_scl);

bufifl buf_scl(Scl,Gnd,Enable_scl);

wire Load_shiftSRPL,Datain_sda;
buf bufl(Datain_sda,Sda);
Shift_SRPL_master Shift_SRPL_master(.Clk(Clk),
.Rst(Rst),
.Load(Load_shiftSRPL),
.In(~Datain_sda),
.Out(Data_out));

UC_Master UC_Master(.Clk(Clk),
.Clk_scl(Clk_scl),
.Rst(Rst),
.Start(Start),
.RW(RW),
.Datain_sda(Datain_sda),
.Pointer(Pointer),
.Set_pointer(Set_pointer),
.Repeat(Repeat),
.Return(Return),
.Out_cont_cycle(Out_cont_cycle),
.Out_cont_data(Out_cont_data),
.En_cont_data(En_cont_data),
.Load_shiftPLSR(Load_shiftPLSR),
.Load_shiftSRPL(Load_shiftSRPL),
.Enable_sda(Enable_sda),
.SelectPLSR(SelectPLSR),
.Enable_clk(Enable_clk),
.Ready(Ready),
.Data_valid(Data_valid),
.Error(Error));

endmodule
```

Figura 9. Codi verilog pel TOP del Màster.

En aquest mòdul no es comprova si les entrades, al iniciar un procés, són vàlides ja que es pressuposa que abans de ficar 'Start' a 1, les dades a les entrades es mantenen estables (tal com es comenta al datasheet). Es podria comprovar de forma senzilla, com a la pràctica anterior, fent un AND lògica del valor de l'entrada per si mateix; si el resultat fos conegut, la dada seria vàlida ja que podríem assegurar que no hi ha 'x'.

Un altre fet a comentar del codi del Màster és que no s'entra directament 'R_W' al UC_master, sinó que es genera un senyal anomenat 'RW'. Això és degut que el valor de l'entrada 'R_W' es pot canviar durant un procés ja que, en cas, de voler repetir l'Start' és probable que no es vulgui escriure, sinó llegir, però s'ha hagut de passar per la fixació del pointer.

3.2.1 Divisor de freqüència

3.2.1.1 Divisor per 25

```
module div_25(
  input   clk_in,
  output   clk_out
    );

reg  [4:0] count = 5'b0;
reg     A1 = 0;
reg       B1 = 0;
reg     Tff_A = 0;
reg    Tff_B = 0;
wire    clock_out;
wire    wTff_A;
wire    wTff_B;

assign   wTff_A  = Tff_A;   //Connects registers to wires
assign   wTff_B  = Tff_B;   //to use combinational logic

assign   clk_out = wTff_B ^ wTff_A; //XOR gate produces output clock.

//Counter for division by N
always@(posedge clk_in)
  begin
    if(count == 5'b11000) //Count to N-1
      begin
        count <=5'b00000;
      end
    else
      begin
        count <= count + 5'b0001;
      end
  end

//Sets A to high for one clock cycle after counter is 0
always@(posedge clk_in)
  begin
    if(count == 5'b00000)
      A1 <= 1;
    else
      A1 <= 0;
  end

//Sets B to high for one clock cycle after counter is (N+1)/2
always@(posedge clk_in)
  begin
    if(count == 5'b01101) //Use (N+1)/2
      B1 <= 1;
    else
      B1 <= 0;
  end

//T flip flop toggles
always@(negedge A1)  // Toggle signal Tff_A whenever A1 goes from 1 to 0
  begin
    Tff_A <= ~Tff_A;
  end

always@(negedge clk_in)
  begin
    if(B1)    // Toggle signal Tff_B whenever B1 clk_in goes from 1 to 0
      begin    // and B1 is 1
        Tff_B <= ~Tff_B;
      end
  end

endmodule
```

Figura 10. Codi verilog pel divisor de freqüència que divideix per 25.

Aquest mòdul ens permet reduir la freqüència del rellotge d'entrada de 50MHz a 2MHz que és a la que treballa el Màster.

3.2.1.2 Divisor per 5

```
//Counter for division by N
always@(posedge clk_in)
  begin
    if(count == 4'b0100) //Count to N-1
      begin
        count <= 4'b0000;
      end
    else
      begin
        count <= count + 4'b0001;
      end
  end

//Sets A to high for one clock cycle after counter is 0
always@(posedge clk_in)
  begin
    if(count == 4'b0000)
      A1 <= 1;
    else
      A1 <= 0;
  end

//Sets B to high for one clock cycle after counter is (N+1)/2
always@(posedge clk_in)
  begin
    if(count == 4'b0011) //Use (N+1)/2
      B1 <= 1;
    else
      B1 <= 0;
  end
```

Figura 11. Part del mòdul de la figura anterior que canvia per dividir per 5.

Amb aquest mòdul aconseguim reduir la freqüència de rellotge de 2MHz a 400kHz.

3.2.2 Registre de desplaçament

3.2.2.1 Paral·lel a sèrie

```
module Shift_PLSR_master(input [7:0] In,
input Load,
input Clk,
input Rst,
output reg Out);

  reg [7:0] temp =  8'b0;
  reg [3:0] Cont = 4'b0;

  always@(posedge Clk or negedge Rst)
    if (!Rst)
      Cont = 4'b0000;
    else if (Cont < 4'b1000 && !Load)
      Cont = Cont + 4'b0001;
    else begin
      if (Cont == 4'b1000)
        Cont = 4'b0;
      else
        Cont = Cont;
    end

  always@(posedge Clk or negedge Rst)  //carga les dades en paral·lel
  if (!Rst) begin
    Out = 1'b0;
    temp = 8'b00000000;
  end
  else if (Load && Cont == 4'b0) begin
      temp = In;
      Out = Out;
  end
  else if (Load && Cont != 4'b0) begin
      temp = temp;
      Out = Out;
  end
  else begin   //sortida sèrie
    Out = temp[7];
    temp = {temp[6:0],1'b0};
  end
endmodule
```

Figura 12. Codi verilog pel registre de desplaçament paral·lel a sèrie.

Mitjançant un comptador intern i un senyal de Enable controlo la càrrega de dades al registre temporal o el desplaçament cap a la sortida sèrie. Aquest comptador és necessari ja que s'ha volgut utilitzar el mateix registre paral·lel – sèrie per enviar l'adreça, el pointer i les dades, per tant, s'ha de controlar l'entrada que es carrega el registre temporal.

3.2.2.2 Sèrie a paral·lel

```
module Shift_SRPL_master(input Clk,
input Rst,
input Load,
input In,
output [7:0] Out);

  reg [7:0] temp;

  always@(posedge Clk or negedge Rst)
  if(!Rst)
    temp <= 8'b0;
  else if(Load)
    temp <= {temp[6:0],In}; //carga les dades en sèrie
  else
    temp <= temp;

  assign Out = temp;  //sortida paral·lel

endmodule
```

Figura 13. Codi verilog pel registre de desplaçament sèrie a paral·lel.

Les dades entren en sèrie tot concatenant-se amb les dades que ja hi havia al registre i es van carregant en paral·lel a la sortida.

3.2.3 Comptador

```
module Contador_rst(input En,
input Rst,
input Clk,
output reg [3:0] Out = 4'b0);

  parameter M = 4'b1000;

  always@(posedge Clk or negedge Rst)
    if (!Rst)
    Out <= 4'b0000;
   else if (!En)
      Out <= 4'b0000;
   else begin
      if (Out == M)
        Out <= 4'b0001;
      else
        Out <= Out + 4'b0001;
    end

endmodule
```

Figura 14. Codi verilog pel comptador.

El paràmetre M és editable al instanciar el mòdul. Aquest nombre informa de quan es vol que es reinicií el comptador. Mentre Out no valgui M i En sigui 1, a cada cicle de rellotge es sumarà 1 al valor d'aquest registre.

3.2.3.1 Comptador de cicles del rellotge d'Scl

El comptador compta fins a 8 i es reinicia. M'ajuda a saber el nombre de bits enviats o rebuts en una seqüència.

3.2.3.2 Comptador de cicles del rellotge del Master

El comptador compta fins a 5 i es reinicia. M'ajuda a saber el moment de període en el que es troba la màquina. El rellotge del Màster va 5 vegades més ràpid que el de la comunicació I2C.

3.2.4 UC_Master

```
module UC_Master(input Clk,
input Clk_scl,
input Rst,
input Start,
input RW,
input Datain_sda,
input [7:0] Pointer,
input Set_pointer,
input Return,
output reg Repeat,
input [3:0] Out_cont_cycle,
input [3:0] Out_cont_data,
output reg En_cont_data,
output reg Load_shiftPLSR,
output reg Load_shiftSRPL,
output reg [1:0] Enable_sda,
output reg [2:0] SelectPLSR,
output reg [1:0] Enable_clk,
output reg Ready,
output reg Data_valid,
output reg Error);

  reg [4:0] state,next;

  parameter IDLE = 5'b00000,
            START = 5'b00001,
            ADRESS = 5'b00010,
            ACK_ADRESS = 5'b00011,
            MSB_RD = 5'b00100,
            ACK_MSB_RD = 5'b00101,
            LSB_RD = 5'b00110,
            NACK_LSB_RD = 5'b00111,
            POINTER = 5'b01000,
            ACK_POINTER = 5'b01001,
            MSB_WR = 5'b01010,
            ACK_MSB_WR = 5'b01011,
            LSB_WR = 5'b01100,
            ACK_LSB_WR = 5'b01101,
            STOP = 5'b01110,
            ERROR = 5'b01111,
            REPEAT = 5'b10000;

  always@(posedge Clk or negedge Rst)
    if (!Rst) state = IDLE;
    else      state = next;

  always@(state or Out_cont_data or Start or Out_cont_cycle or Datain_sda or Clk_scl or RW or Return or Pointer or Set_pointer) begin
  next = 4'bx;
  case(state)
    IDLE: if (Start) next = START;
          else       next = IDLE;
    START: if (Out_cont_cycle == 4'b0001) next = ADRESS;
           else next = START;
    ADRESS: if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0101) next = ACK_ADRESS;
        else  next = ADRESS;
    ACK_ADRESS: if (Clk_scl == 1'b1 && Datain_sda == 1'b0 && RW == 1'b0) next = POINTER;
        else if (Clk_scl == 1'b1 && Datain_sda == 1'b0 && RW == 1'b1) next = MSB_RD;
        else if (Clk_scl == 1'b1 && Datain_sda == 1'b1) next = IDLE;
        else next = ACK_ADRESS;
    MSB_RD: if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0001 && Pointer[1:0] != 2'b01) next = ACK_MSB_RD;
        else if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0001 && Pointer[1:0] == 2'b01) next = NACK_LSB_RD;
        else  next = MSB_RD;
    ACK_MSB_RD: if (Out_cont_cycle == 4'b0001) next = LSB_RD;
        else next = ACK_MSB_RD;
    LSB_RD: if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0001) next = NACK_LSB_RD;
        else  next = LSB_RD;
    NACK_LSB_RD: if (Out_cont_cycle == 4'b0001) next = STOP;
        else next = NACK_LSB_RD;
    POINTER: if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0101) next = ACK_POINTER;
        else  next = POINTER;
    ACK_POINTER: if (Clk_scl == 1'b1 && Datain_sda == 1'b0 && Set_pointer == 1'b0) next = MSB_WR;
        else if (Clk_scl ==1'b1 && Datain_sda == 1'b0 && Set_pointer == 1'b1) next = REPEAT;
        else if (Clk_scl == 1'b1 && Datain_sda == 1'b1) next = ERROR;
        else next = ACK_POINTER;
    MSB_WR: if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0101) next = ACK_MSB_WR;
         else  next = MSB_WR;
    ACK_MSB_WR: if (Clk_scl == 1'b1 && Datain_sda == 1'b0 && Pointer[1] == 1'b1) next = LSB_WR;
         else if (Clk_scl == 1'b1 && Datain_sda == 1'b0 && Pointer[1] == 1'b0) next = STOP;
         else if (Clk_scl == 1'b1 && Datain_sda == 1'b1) next = ERROR;
         else next = ACK_MSB_WR;
    LSB_WR: if (Out_cont_data == 4'b1000 && Out_cont_cycle == 4'b0101) next = ACK_LSB_WR;
        else  next = LSB_WR;
    ACK_LSB_WR: if (Clk_scl == 1'b1 && Datain_sda == 1'b0 && Out_cont_cycle == 4'b0011) next = STOP;
         else if (Clk_scl == 1'b1 && Datain_sda == 1'b1 && Out_cont_cycle == 4'b0011) next = ERROR;
         else next = ACK_LSB_WR;
    STOP: if (Out_cont_cycle == 4'b0011) next = IDLE;
         else  next = STOP;
    ERROR: if (Out_cont_cycle == 4'b0101) next = IDLE;
         else  next = ERROR;
    REPEAT: if (Out_cont_cycle == 4'b0001 && Return == 1'b1) next = ADRESS;
         else next = REPEAT;
  endcase
  end

  always@(state or Out_cont_cycle or Out_cont_data or Clk_scl or Return) begin
```

```
Enable_sda = 2'b00;
Enable_clk =  2'b00;
En_cont_data = 1'b0;
SelectPLSR = 3'b000;
Load_shiftPLSR = 1'b1;
Load_shiftSRPL = 1'b0;
Ready = 1'b0;
Data_valid = 1'b0;          //informa al usuari quan la dada a la Data_out és vàlida
Error = 1'b0;
Repeat = 1'b0;              //si = 1, indica que estem al estat per repetir Start
case(state)
  IDLE: begin
  Ready = 1'b1;
  SelectPLSR = 3'b100; //carrego Save_adr a al ShiftSRPL
  end
  START:begin
  Enable_sda = 2'b01;       //provoco un negedge a Sda
  SelectPLSR = 3'b100;      //selecciono el registre del qual carregar les dades al shift register
  if (Out_cont_cycle == 4'b0001)
    Load_shiftPLSR = 1'b0;
  else
    Load_shiftPLSR = 1'b1;
  end
  ADRESS:begin
  Enable_sda = 2'b10;
  Enable_clk = 2'b10;
  En_cont_data = 1'b1;
    if (Out_cont_cycle == 4'b0001)
      Load_shiftPLSR = 1'b0;  //carrego dada a la sortida del shift
    else
      Load_shiftPLSR = 1'b1;  //als demés cicles la dada es manté estable a la sortida
  end
  ACK_ADRESS:begin
  Enable_clk = 2'b10;         //Scl segueix activat per rebre l'ACK
  end
  MSB_RD:begin
  Enable_clk = 2'b10;
  En_cont_data = 1'b1;
    if (Out_cont_cycle == 4'b0100 && Out_cont_data != 4'b0000)
      Load_shiftSRPL = 1'b1;
    else
      Load_shiftSRPL = 1'b0;
  end
  ACK_MSB_RD:begin
  Enable_clk = 2'b10;
  Enable_sda = 2'b01;       //provo un 0 a Sda activant el tri-state
  Data_valid = 1'b1;        //les dades a la sortida seran vàlides
  end
  LSB_RD:begin
  Enable_clk = 2'b10;
  En_cont_data = 1'b1;
    if (Out_cont_cycle == 4'b0100 && Out_cont_data != 4'b0000)
      Load_shiftSRPL = 1'b1;
    else
      Load_shiftSRPL = 1'b0;
  end
  NACK_LSB_RD:begin
  Enable_clk = 2'b10;
  Data_valid = 1'b1;
  end
  POINTER: begin
  Enable_sda = 2'b10;
  Enable_clk = 2'b10;
  En_cont_data = 1'b1;
  SelectPLSR = 3'b001;
    if (Out_cont_cycle == 4'b0001)
      Load_shiftPLSR = 1'b0;  //carrego dada a la sortida del shift
    else
      Load_shiftPLSR = 1'b1;  //als demés cicles la dada es manté estable a la sortida
  end
  ACK_POINTER:begin
  Enable_clk = 2'b10;         //Scl segueix activat per rebre l'ACK
  end
  MSB_WR: begin
  Enable_sda = 2'b10;
  Enable_clk = 2'b10;
  En_cont_data = 1'b1;
  SelectPLSR = 3'b010;
    if (Out_cont_cycle == 4'b0001)
      Load_shiftPLSR = 1'b0;
    else
      Load_shiftPLSR = 1'b1;
  end
  ACK_MSB_WR:begin
  Enable_clk = 2'b10;
  end
  LSB_WR:begin
  Enable_sda = 2'b10;
  Enable_clk = 2'b10;
  En_cont_data = 1'b1;
  SelectPLSR = 3'b011;
    if (Out_cont_cycle == 4'b0001)
      Load_shiftPLSR = 1'b0;
    else
      Load_shiftPLSR = 1'b1;
  end
```

```
    ACK_LSB_WR:begin
    Enable_clk = 2'b10;
    end
    STOP:begin
    if (Out_cont_cycle != 4'b0011)begin
      Enable_clk = 2'b10;
      Enable_sda = 2'b01;
    end
    else
      Enable_clk = 2'b00;
    end
    ERROR:begin
    Error = 1'b1;
    if (Out_cont_cycle != 4'b0010)begin
      Enable_clk = 2'b10;
      Enable_sda = 2'b01;
    end
    else
      Enable_clk = 2'b00;
    end
    REPEAT: begin
    Enable_clk = 2'b10;
    Repeat = 1'b1;
    SelectPLSR = 3'b100;
    if (Out_cont_cycle == 4'b0101 && Return == 1'b1) begin
      Enable_sda = 2'b01;
      Load_shiftPLSR = 1'b0;  //carrego l'adreça al shift register
    end
    else if (Out_cont_cycle == 4'b0100 && Return == 1'b1) begin
      Enable_sda = 2'b01;
      Load_shiftPLSR = 1'b1;
    end
    else begin
      Enable_sda = 2'b00;
      Load_shiftPLSR = 1'b1;
    end
    end

  endcase
  end
endmodule
```

Figura 15. Codi verilog per la UC del Màster.

Amb aquest mòdul es controla tot el Màster ja que és l'encarregat d'activar els 'enable' dels registres de desplaçament, els 'enable' dels registres que guarden les entrades, els multiplexors, etc.

3.3 Tests

3.3.1 tb_I2C

```
module tb_I2C();

  wire Start,R_W;
  wire [7:0] Data_in,Data_in2,Pointer;
  wire [7:0] Data_out;
  wire [6:0] Adr;

  sys_clk50MHz_fm Clk50(.Clk(Clk_in));

  sys_rst_fm sys_rst(.Rst_n(Rst));

  Master Master(.Rst(Rst),
  .Start(Start),
  .Data_in(Data_in),
  .Data_in2(Data_in2),
  .Adr(Adr),
  .Pointer(Pointer),
  .Set_pointer(Set_pointer),
  .R_W(R_W),
  .Ready(Ready_master),
  .Error(Error_master),
  .Data_out(Data_out),
  .Data_valid(Data_valid),
  .Scl(Scl),
  .Sda(Sda),
  .Clk_in(Clk_in));

  LM75x_fm LM75x_fm(.Sda(Sda),
  .Scl(Scl),
  .Rst(Rst),
  .Start(Start));

  I2C_cntr_fm I2C_cntr_fm(
  .Adr(Adr),
  .Pointer(Pointer),
  .Set_pointer(Set_pointer),
  .Data_in(Data_in),
  .Data_in2(Data_in2),
  .R_W(R_W),
  .Start(Start),
  .Error(Error),
  .Ready(Ready));

  pullup(Sda);
  pullup(Scl);

endmodule
```

Figura 16. Codi verilog pel mòdul que agrupa tots els mòduls del model funcional.

3.3.2 test_I2C

```
`define SYSRST            tb_I2C.sys_rst
`define CLK50M            tb_I2C.Clk50
module test_I2C();

  tb_I2C tb_I2C();

  initial
  begin
   `SYSRST.rstOn;              //Rst inicial
   `CLK50M.waitCycles(2);
   `SYSRST.rstOff;


   /*`CLK50M.waitCycles(1000);  //activar per fer el test d'un Rst asíncron en mig d'un procés
   `SYSRST.rstOn;              //juntament amb el bloc de codi assignat al módul 'I2C_cntr_fm'
   `CLK50M.waitCycles(3);
   `SYSRST.rstOff;*/

  #250000 $finish();
  end

endmodule
```

Figura 17. Codi verilog pel mòdul del test que crida a tots els demés mòduls i activa el reset.

4 Test

Al mòdul intervenen moltes senyals així que per explicar els tests s'han inclòs només les que tenen més rellevància.

4.1 Escriptura y lectura de 2 bytes fixant un registre

4.1.1 Start, adreça i pointer per l'escriptura

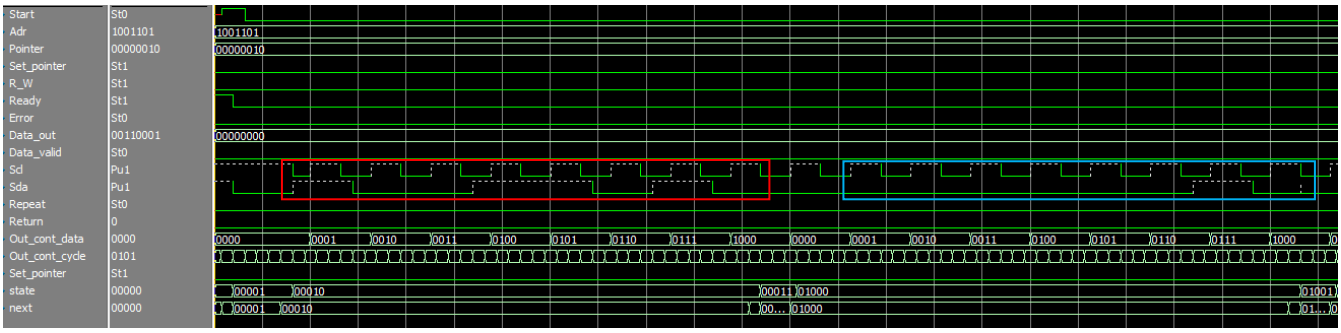


Figura 18. Diagrama d'ones del senyals més rellevants en la condició d'Start, l'enviament de l'adreça i el pointer.

4.1.2 Bytes de dades enviats

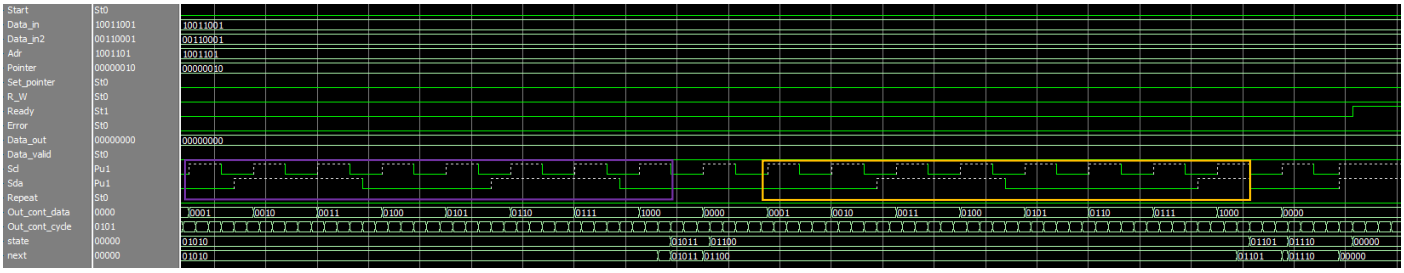


Figura 19. Diagrama d'ones del senyals més rellevants en l'enviament dels 2 bytes de dades.

4.1.3 Start, adreça i pointer per fixar el registre del Slave

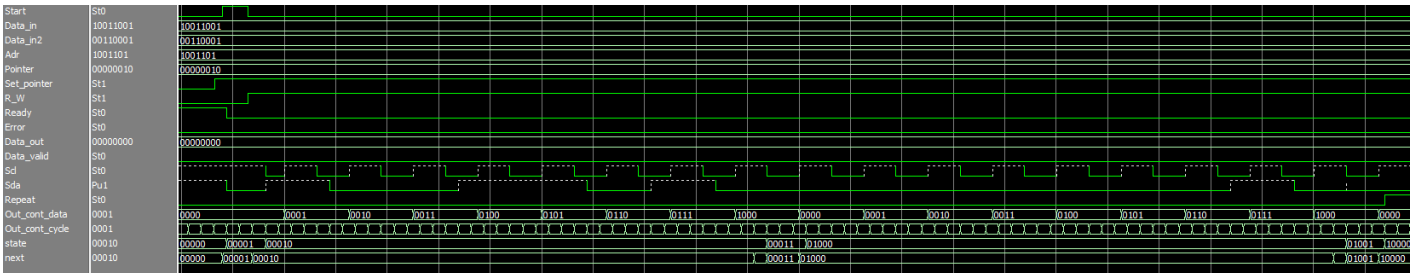


Figura 20. Diagrama d'ones dels senyals més rellevants en l'enviament del start, l'adreça i el pointer.

4.1.4 Repetició del 'start' i bytes de dades rebuts

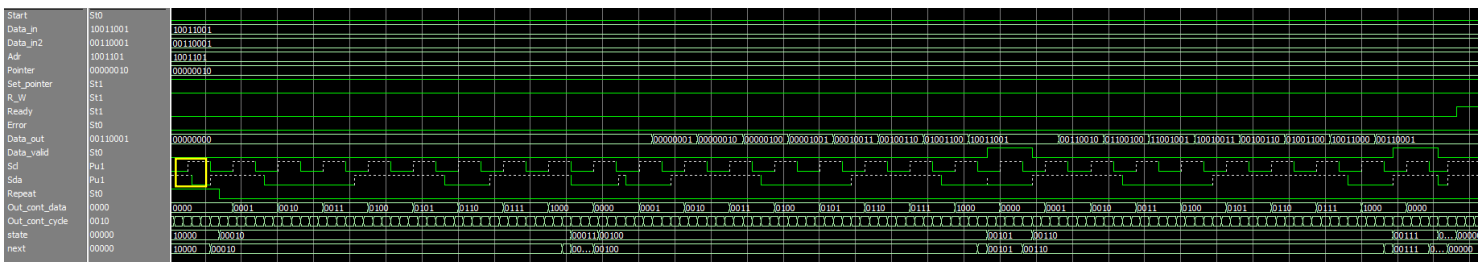


Figura 21. Diagrama d'ones dels senyals més rellevants en la repetició del 'Start' i els bytes rebuts.

A les figures 18 i 19 veiem un procés d'escriptura de 2 bytes al Slave. Es passa per l'enviament de la condició d'Start', l'adreça, el pointer i els 2 bytes.

En vermell veiem l'enviament de l'adreça seguit pel ACK del Slave. En blau s'observa el byte de pointer seguit, també, per un ACK del Slave. En lila tenim el primer byte de dades amb el corresponent ACK. Per últim tenim en taronja el segon byte de dades que el Màster envia cap al Slave amb el ACK conforme les ha rebut correctament.

A la figura 20 trobem el mateix que a la figura 18, però aquest cop Set_pointer = 1, per tant, no s'enviarà cap byte de dades, sinó que es repetirà la condició d'Start' amb l'objectiu de, al següent procés, llegir un Slave.

A la figura 21, en groc, veiem la repetició de la condició d'Start' per continuar amb l'enviament de l'adreça i la rebuda dels 2 bytes de dades amb el corresponent ACK enviat pel Màster per al primer byte i el NACK pel segon.

4.2 Escriptura y lectura d'un byte fixant un registre

4.2.1 Start, adreça, pointer i byte enviat

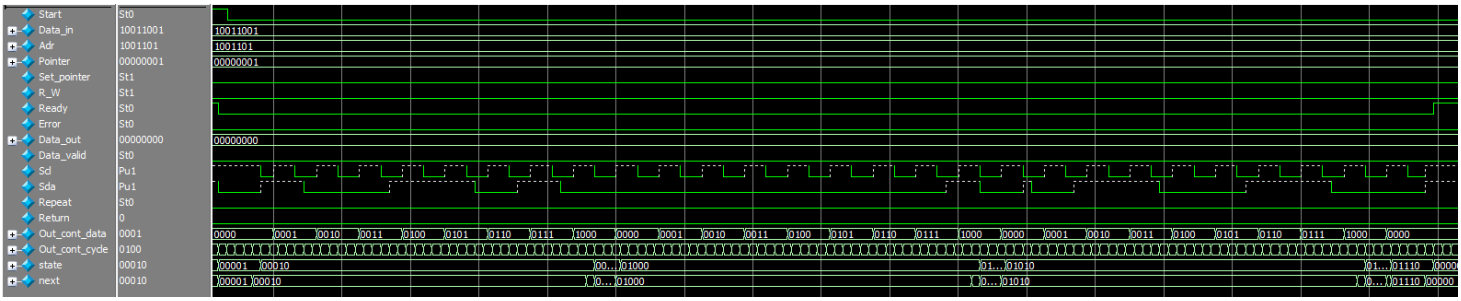


Figura 22. Diagrama d'ones dels senyals més rellevants en l'enviament del start, l'adreça, el pointer i el byte.

En aquest test veiem el mateix procediment que l'anterior però aquest cop enviant només un byte de dades.

4.2.2 Start, adreça i pointer

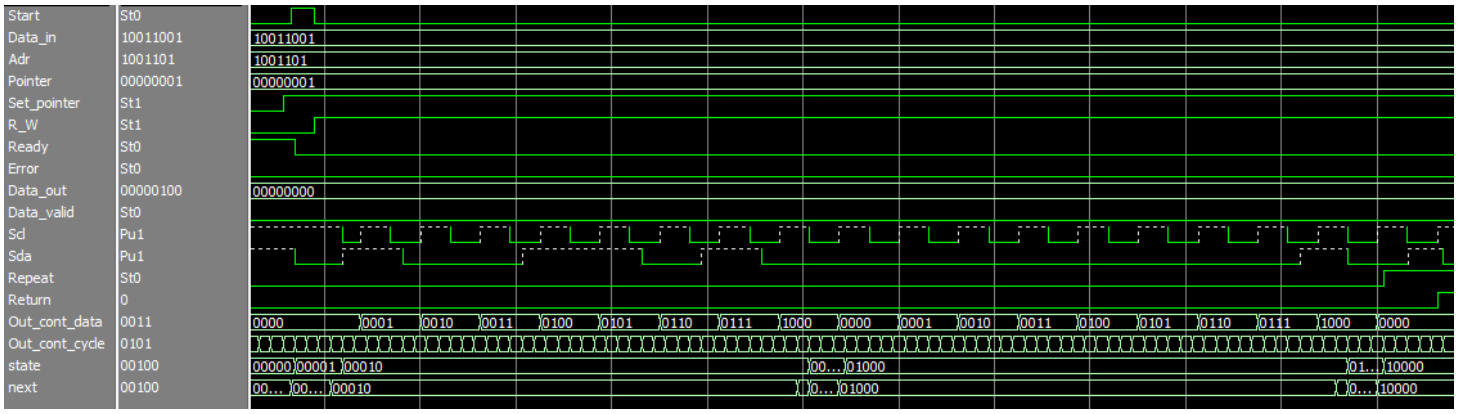


Figura 23. Diagrama d'ones de les senyals més rellevants en l'enviament del start, l'adreça i el pointer.

En aquest test veiem la primera part de la lectura d'un byte. Trobem l'enviament de la condició d'Start, l'adreça i el pointer. Aquest dos últims bytes els acompanya el seu ACK.

4.2.3 Repetició d'start, adreça i byte llegit

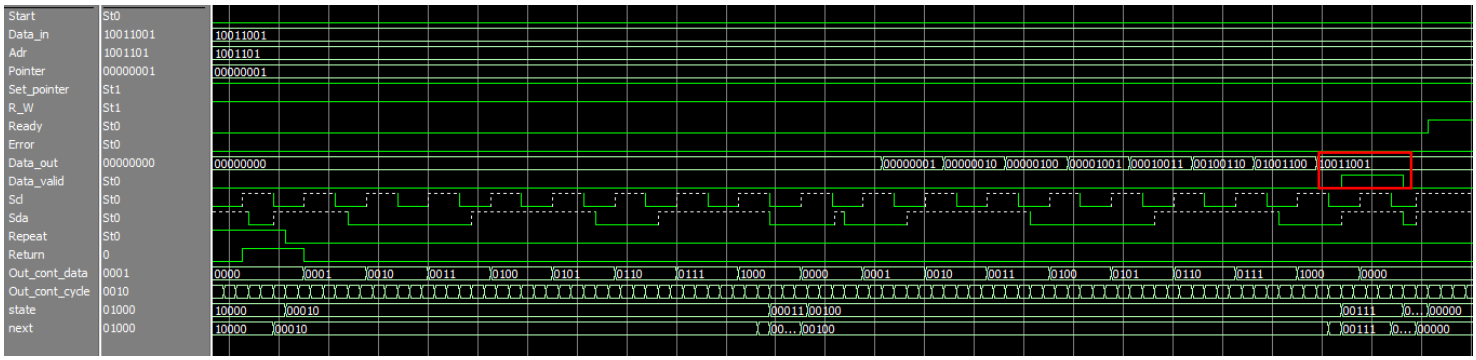


Figura 24. Diagrama d'ones de les senyals més rellevants en la repetició del 'start', l'enviament de l'adreça i el byte rebut.

En aquest test trobem la segona part de la lectura d'un byte. Veiem la repetició de la condició d'Start, es torna a enviar l'adreça i es rep el byte de dades. Com s'aprecia marcat en vermell, 'Data_valid' es posa a nivell alt quan les dades a la sortida són estables.

4.3 Rst asíncron durant un procés

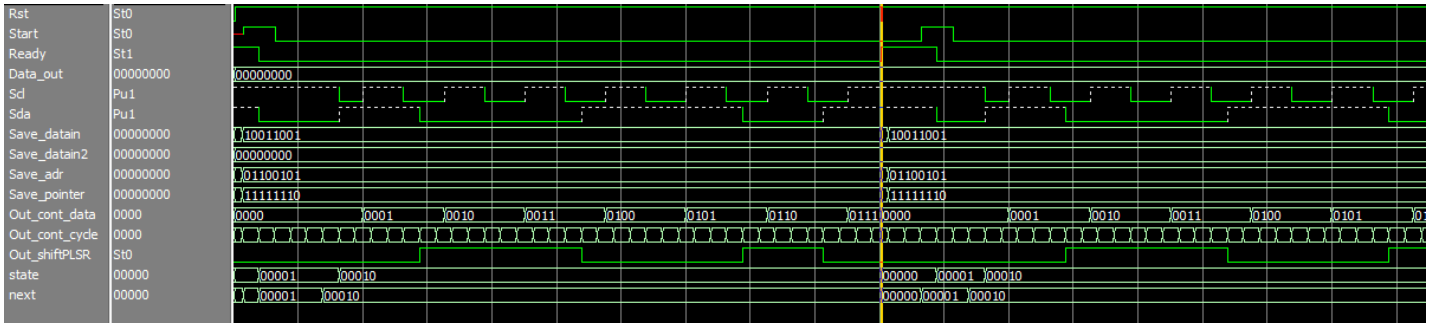


Figura 25. Diagrama d'ones de les senyals més rellevants al rebre un reset asíncron.

En aquest test trobem la reacció de la màquina davant un reset asíncron durant un procés. S'aprecia, marcat amb una línia groga, com la màquina, independentment del moment en que es trobi, torna a IDLE a esperar un nou 'Start', es reinicien tots els registres i tenim un 0 a les sortides dels registres de desplaçament i els comptadors. Quan es rep l'Start', s'inicia un nou procés amb les instruccions que s'hagin imposat.

4.4 Rebuda d'un NACK al enviar l'adreça

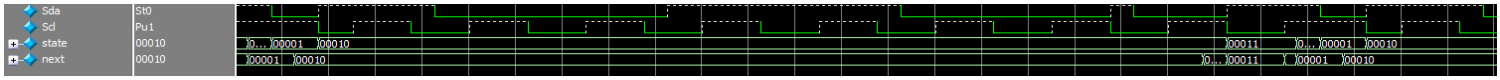


Figura 26. Diagrama d'ones de les senyals més rellevants en la rebuda d'un NACK al enviar l'adreça.

En aquest test veiem la reacció de la màquina al rebre un NACK després d'enviar l'adreça. Com s'aprecia, es torna al estat IDLE per esperar un nou 'Start'. Com he deixat 'Start' a 1, la màquina inicia un nou procés amb les instruccions que s'hagin donat (lectura o escriptura fixant o no el registre de pointer).

5 Síntesi al Quartus

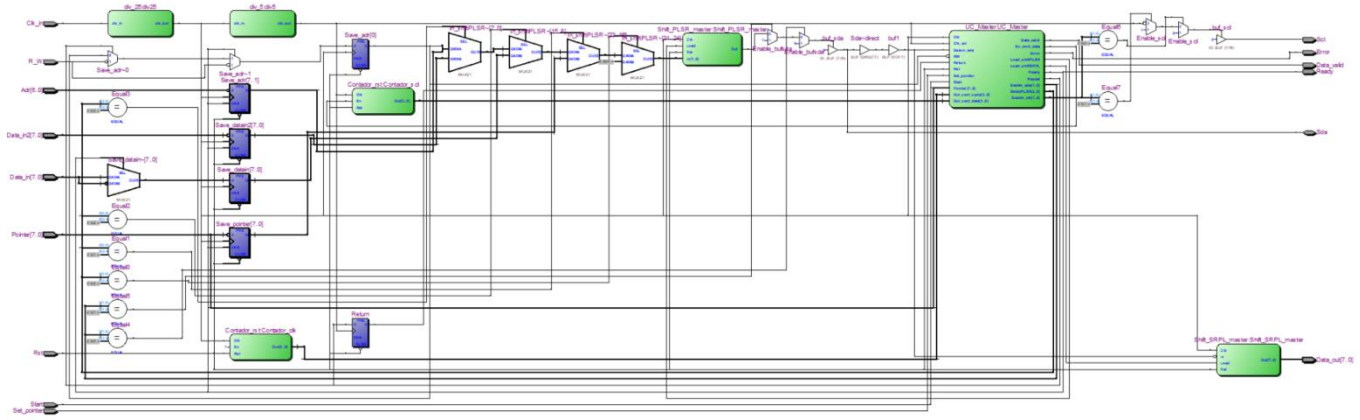


Figura 27. Diagrama de blocs del TOP del Màster generat pel Quartus al sintetitzar.

5.1 Divisor de freqüència

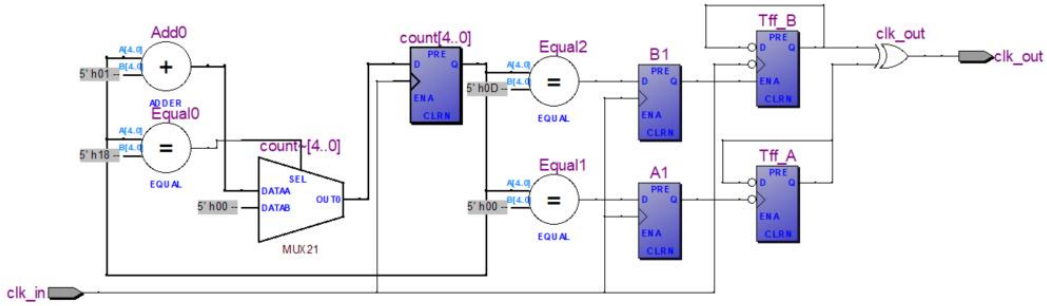


Figura 28. Circuit del divisor de freqüència generat pel Quartus a sintetitzar.

5.2 Comptador

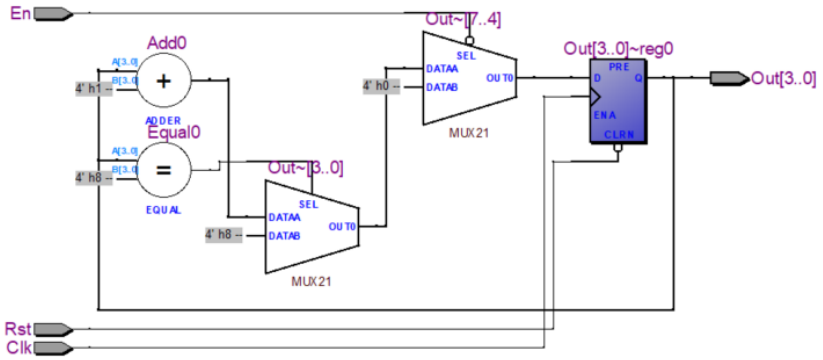


Figura 29. Circuit del comptador generat pel Quartus al sintetitzar.

5.3 Registre de desplaçament sèrie a paral·lel

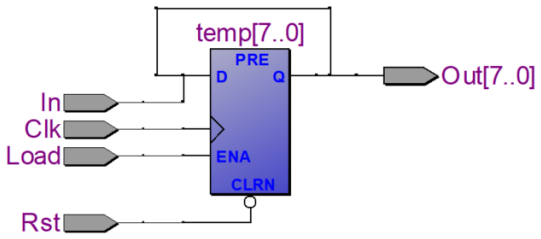


Figura 30. Circuit del registre de desplaçament sèrie a paral·lel generat pel Quartus al sintetitzar.

5.4 Registre de desplaçament paral·lel a sèrie

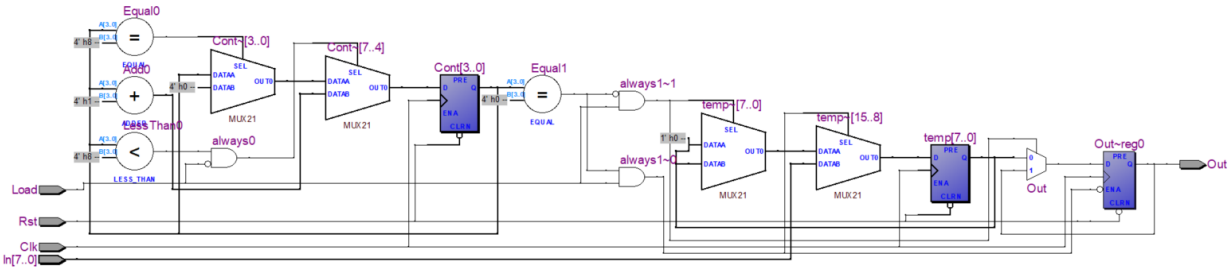


Figura 31. Circuit del registre de desplaçament paral·el a sèrie generat pel Quartus al sintetitzar.

5.5 UC

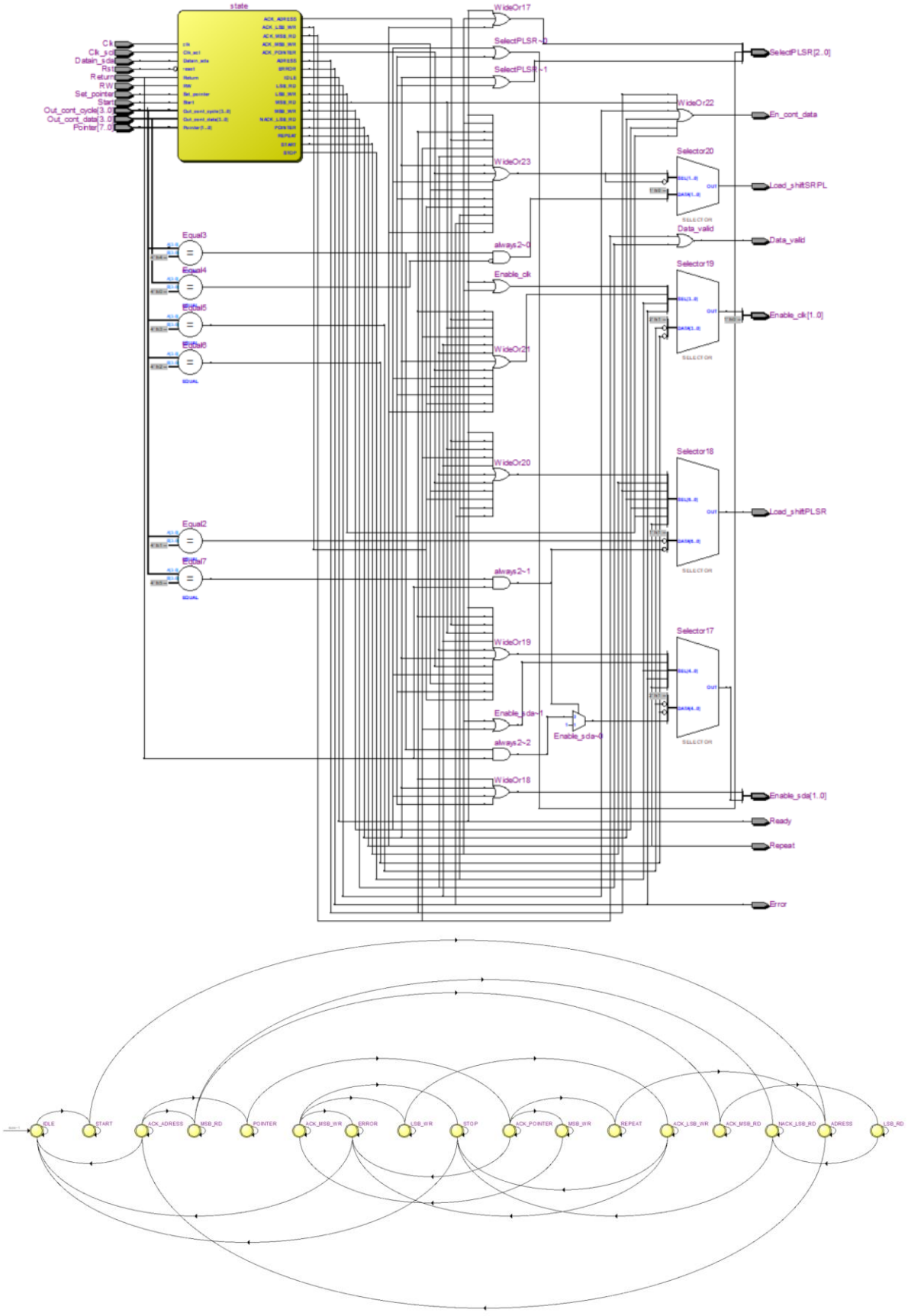


Figura 32. Circuit i diagrama RTL de la FSM del Màster generat pel Quartus al sintetizar.

6 Referències

Divisor de freqüència

https://github.com/Obijuan/open-fpga-verilog-tutorial/wiki/Cap%C3%ADtulo-15:-Divisor-de-frecuencias

Datasheet de referència del Màster

https://opencores.org/websvn/filedetails?repname=i2c&path=%2Fi2c%2Ftrunk%2Fdoc%2Fi2c_specs.pdf

Datasheet de referència del sensor

http://www.ti.com/lit/ds/symlink/lm75b.pdf?&ts=1589308161538