

Informe Pràctica 3 Sessió 3: Controlador de Bit

Errors en el RTL:

#Error	Línies	Descripció
1	104	<p>Descripció de l'error: Encara que per tal de generar tant una condició de "start" com de "stop" és necessari que SCL estigui en 1, tal com està definida la condició de "start" al codi original no és correcte. Hem canviat ~sSCL per sSCL</p> <p>Codi original: sta_condition <= ~sSDA & dSDA & ~sSCL</p> <p>Codi corregit: sta_condition <= ~sSDA & dSDA & sSCL</p>
2	193 i 198	<p>Descripció de l'error: Al càlcul del següent estat en mode d'escriptura, hi ha un bucle infinit d'estats degut a que se salta l'estat "C" i passa directament al "D". Després, un cop està a l'estat "D", la condició <i>else</i>, enlloc de mantenir l'estat actual fa un salt a l'estat "B"; generant així un bucle infinit.</p> <p>Codi original: WR_B : if(clk_en) next = WR_D; else next = WR_B; WR_C : if(clk_en) next = WR_D; Else next = WR_C; WR_D : if(clk_en) next = IDLE; else next = WR_B;</p> <p>Codi corregit: WR_B : if(clk_en) next = WR_C; else next = WR_B; WR_C : if(clk_en) next = WR_D; Else next = WR_C; WR_D : if(clk_en) next = IDLE; else next = WR_D;</p>

3	230	<p>Descripció de l'error: A la seqüència de "start", a la primera part (B), hi ha un error a l'hora d'assignar el valor de "Sda_oen" ja que, segons la seqüència del gràfic del guió, a la part B el valor d'aquesta ha de commutar a 1 i en aquest cas està assignat a 0.</p> <p>Codi original: START_B : begin Scl_oen <= 1'b1; <i>// keep SCL high</i> Sda_oen <= 1'b0; <i>// keep SDA high</i> sda_chk <= 1'b0; <i>// don't check SDA output</i> end</p> <p>Codi corregit: START_B : begin Scl_oen <= 1'b1; <i>// keep SCL high</i> Sda_oen <= 1'b1; <i>// keep SDA high</i> sda_chk <= 1'b0; <i>// don't check SDA output</i> end</p>
4	239	<p>Descripció de l'error: Aquest és només un error de comentari, ja que el que s'està fent no es correspon amb el comentari associat.</p> <p>Codi original: START_D : begin Scl_oen <= 1'b1; <i>// set SCL low</i> Sda_oen <= 1'b0; <i>// set SDA low</i> sda_chk <= 1'b0; <i>// don't check SDA output</i> end</p> <p>Codi corregit: START_D : begin Scl_oen <= 1'b1; <i>// keep SCL high</i> Sda_oen <= 1'b0; <i>// set SDA low</i> sda_chk <= 1'b0; <i>// don't check SDA output</i> end</p>

5	244	<p>Descripció de l'error: Aquest altre és també només un error de comentari, ja que el que s'està fent no es correspon amb el comentari associat.</p> <p>Codi original: START_E : begin Scl_oen <= 1'b1; <i>// keep SCL low</i> Sda_oen <= 1'b0; <i>// keep SDA low</i> sda_chk <= 1'b0; <i>// don't check SDA output</i> end</p> <p>Codi corregit: START_E : begin Scl_oen <= 1'b1; <i>// keep SCL high</i> Sda_oen <= 1'b0; <i>// keep SDA low</i> sda_chk <= 1'b0; <i>// don't check SDA output</i> end</p>
---	-----	---

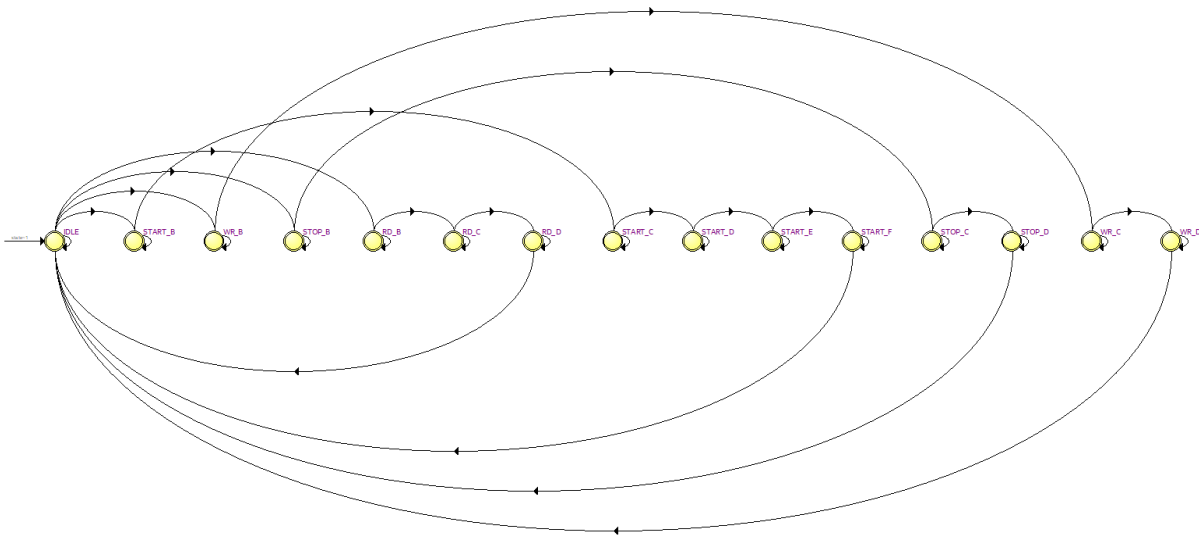
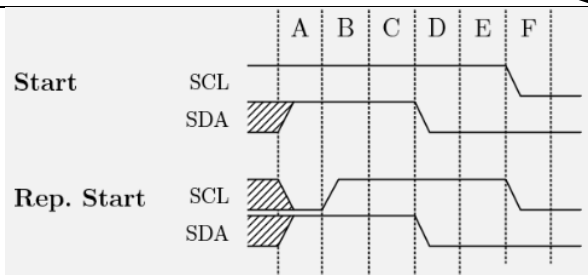
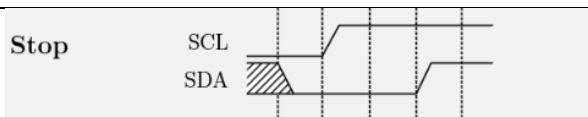
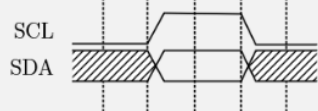
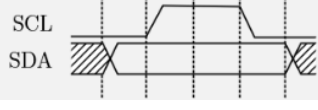


Figura 2: Diagrama d'estats (generat automàticament pel Quartus) del controlador de Bit de la unitat de control I2C.

Taula 1: Llista dels diferents estats de la màquina d'estats implementada. Aquí fem referència a l'habilitació d'outputs però li diem SCL i SDA directament per conveniència.

Estat	Descripció	Dibuix
IDLE	Aquest és l'estat de repòs. En aquest estat no es fa res més que esperar a que arribi una comanda que dicti quina operació s'ha de començar.	
START_B	En aquest estat es comença la operació de Start, mantenint tant SCL com SDA en estat alt. En cas de repetició de Start, es força la pujada de SCL.	
START_C	Aquí es mantenen tant SCL com SDA en estat alt, assegurant així que s'està en un estat desitjat per a començar la seqüència "important" que determinarà un Start.	
START_D	Aquí comença la seqüència "important" de Start, canviant SDA a un nivell baix (mantenint SCL)	
START_E	En aquest estat es mantenen els valors de l'estat anterior.	
START_F	Aquest és l'últim estat per a la seqüència de Start, canviant SCL a baix (mantenint SDA).	
STOP_B	En aquest estat es comença la seqüència de Stop, la qual portarà els valors de SCL i SDA als de l'estat d'IDLE. En aquest pas es canvia SCL de baix a alt.	
STOP_C	Aquí es mantenen els valors de l'estat anterior.	
STOP_D	Aquí s'acaba la seqüència de Stop, tornant a canviar SDA a nivell alt i mantenint SCL en nivell baix.	

RD_B	En aquest estat s'habilita SCL deixant SDA en tri-estat, permetent que l'emissor envii el bit d'informació a llegir.	Read 
RD_C	Aquí es mantenen els valors de l'estat anterior.	
RD_D	Aquí es deshabilita SCL finalitzant la seqüència de Read.	
WR_B	En aquest estat s'habilita SCL mentre a SDA se li assigna el valor que es desitgi transmetre.	Write 
WR_C	Aquí es mantenen els valors de l'estat anterior.	
WR_D	En aquest estat es finalitza la seqüència de Write, deshabilitant SCL.	

Qüestions

1. Perquè la generació del senyal Start és de 6 etapes en lloc de 4?

Per tal de gestionar el fet que es pot tornar a començar una seqüència de Start sense haver passat per un Stop abans, cal afegir 2 estats més que permetin el restabliment de la seqüència com si s'estigués fent un Start per primer cop.

2. Quina és la formula que ens dona la freqüència de rellotge o temps de bit?

La freqüència de rellotge (temps de bit) vindrà donat pel *prescaler*, el qual canviarà la freqüència efectiva del sistema. Aquesta freqüència es pot calcular a través de la següent expressió:

$$SCK = CLK / (2 * PRER) \quad \text{on PRER es el valor de pre-escalat.}$$

3. Com s'implementa la detecció de col·lisions? I el clock stretching? Dibuixeu l'esquema RTL de la lògica i mostreu una captura de simulació a nivell RTL que ho mostri com funcionen.

La detecció de col·lisions s'implementa mitjançant el control de pèrdua d'arbitratge sobre el Bus I2C. Hi han dues situacions possibles en les que es produeix aquesta pèrdua: (1) Quan el mestre I2C deixa anar la línia SDA (pull-up), però aquesta roman en estat lògic baix, i (2) quan es detecta una condició de *Stop* no sol·licitada.

```
// generate arbitration lost signal
// arbitration lost when:
```

```
// 1) master drives SDA high, but the i2c bus is low
// 2) stop detected while not requested
always @(posedge Clk or negedge Rst_n)
    if(!Rst_n)      cmd_stop <= 1'b0;          // cmd_stop bit is set to 0 upon reset
    else if(clk_en) cmd_stop <= Cmd == `I2C_CMD_STOP; // cmd_stop bit is set to 1 if
requested Cmd is Stop Command
    else            cmd_stop <= cmd_stop;      // if not requested, keep cmd_stop as in
previous cycle

always @(posedge Clk or negedge Rst_n)
    if(!Rst_n) I2C_al <= 1'b0; // No arbitration loss on reset
    else      I2C_al <= (sda_chk & ~sSDA & Sda_oen) | (sto_condition & ~cmd_stop);
```

Figura 3. Codi de gestió d'arbitratge de Bus I2C.

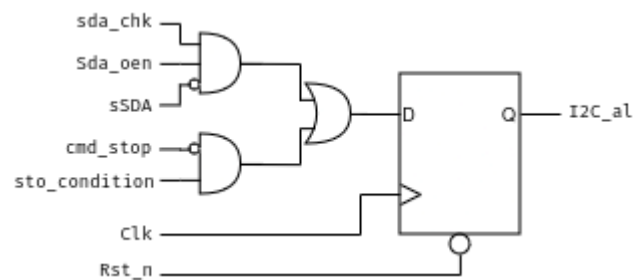


Figura 4. Esquemàtic RTL de la lògica de control de pèrdua d'arbitratge.

El codi de la Figura 3 permet la detecció de col·lisions a partir de la situació que s'hagi donat en la pèrdua d'arbitrarietat. Aquest codi sintetitza en l'estructura de la Figura 4.

```
# [Info- 18685.00 ns] Test: Arbitration Lost (condition1)
# [Info- 21045.00 ns] Test: Arbitration Lost (condition2)
```

Figura 5. Missatge de test de pèrdua d'arbitratge.

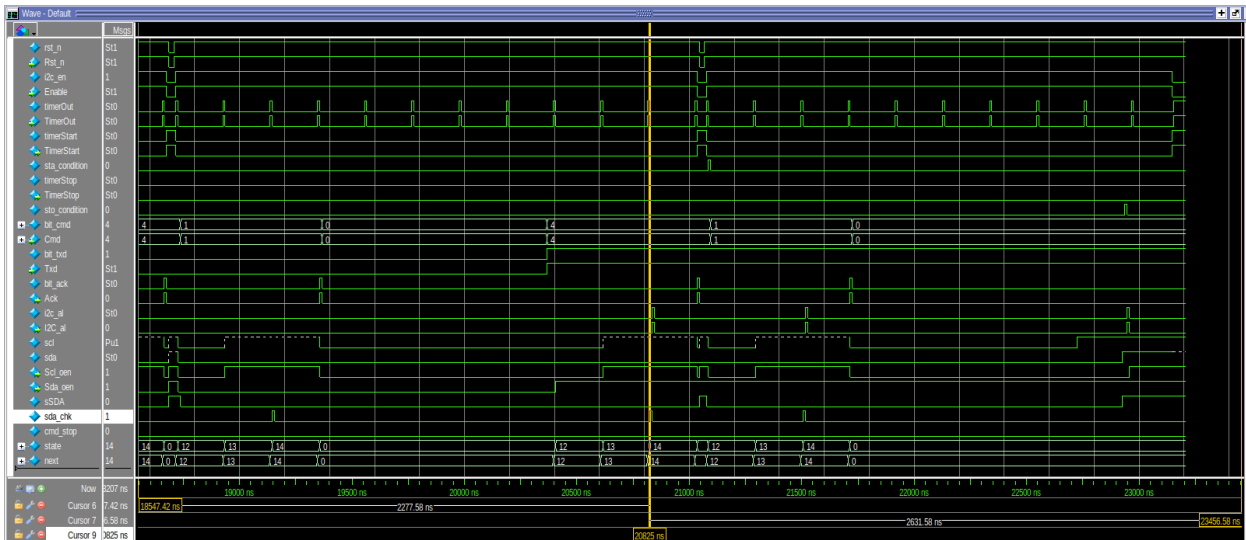


Figura 6. Diagrama d'ones de la primera situació de pèrdua d'arbitratge.

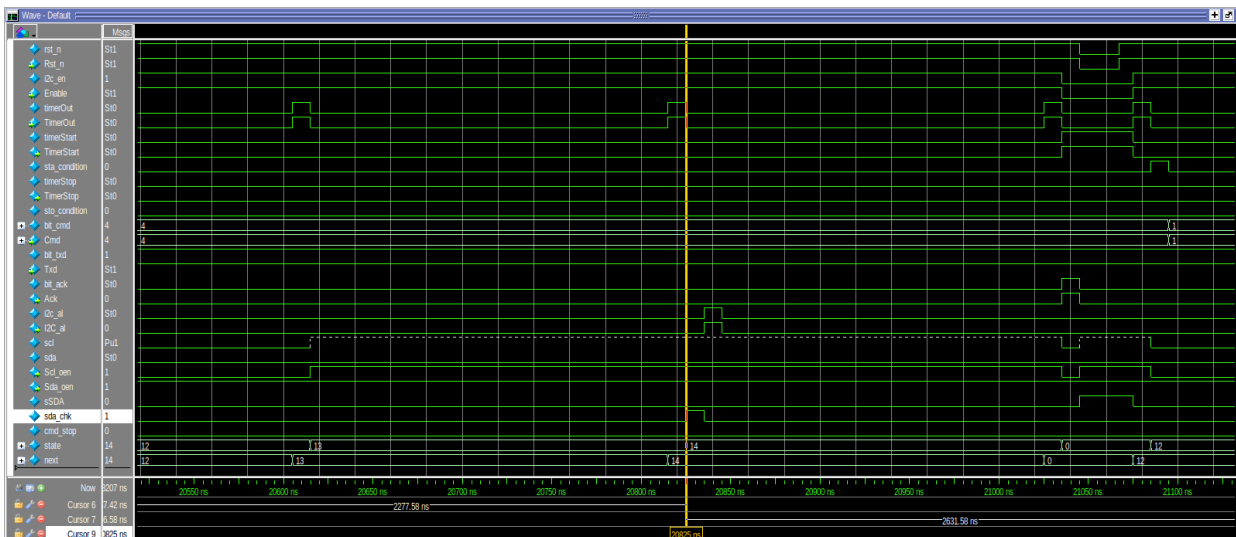


Figura 7. Ampliació de la Figura 6 a la zona d'interès.

Inspeccionant els diagrames de les Figures 6 i 7, s'observa que en el moment en que s'acaba d'escriure a l'esclau ($state = 13$ (WR_C) $\rightarrow state = 14$ (WR_D)), el senyal sda_chk commuta de nivell baix a nivell alt i alhora els senyals $Sda_oen = 1$ i $sSDA = 0$ fan que es produeixi una pèrdua d'arbitratge, indicada per $I2C_al = 1$ al següent cicle de rellotge. El que passa és que el mestre detecta que la línia SDA està a nivell baix ($sSDA = 0$) quan hauria d'estar en nivell alt, ja que la trama de dades a transferir s'ha acabat. Aleshores, al fer la comprovació, salta la bandera de pèrdua d'arbitratge. Aquesta situació es pot donar si més d'un dispositiu mestre intenta accedir a la línia al mateix temps.

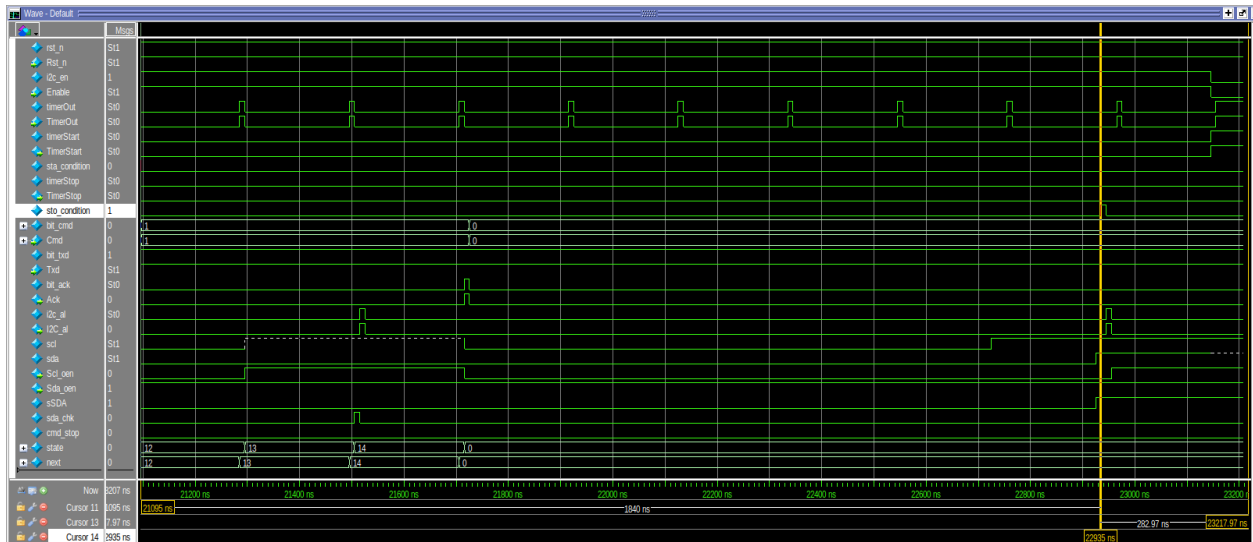


Figura 8. Diagrama d'ones de la segona situació de pèrdua d'arbitratge.

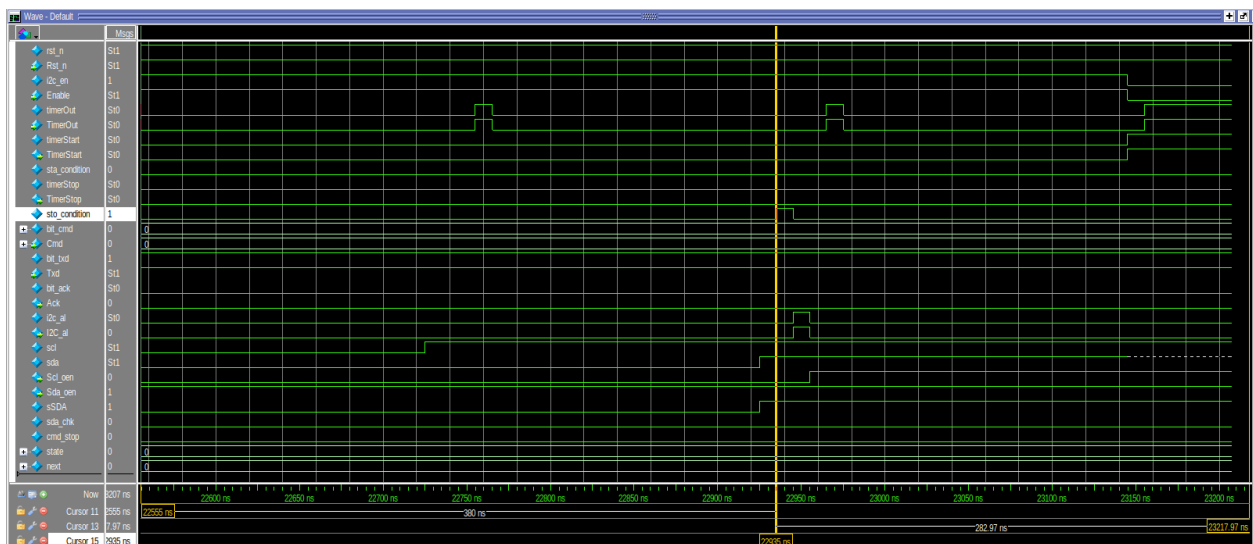


Figura 9. Ampliació de la Figura 8 a la zona d'interès.

Inspeccionant els diagrames de les Figures 8 i 9, s'observa que es genera una condició de Stop, donat que $sto_condition = sSDA \& \sim dSDA \& sSCL$ i a la transició anterior forcem $scl = sda = 1$ i $dSDA = 0$. No obstant, com que en cap moment s'ha sol·licitat una comanda de Stop ($cmd_stop = 0$), es detecta pèrdua d'arbitratge ($I2C_al = 1$, vegeu codi Figura 3).

El *clock stretching* es pot produir quan un dispositiu mestre comença una lectura de dades que proporciona un dispositiu esclau connectat a la línia de comunicació per I2C. En la transmissió d'una trama de dades, l'esclau ha d'enviar el bit d'Ack i seguit immediatament de la trama a llegir per el mestre. No obstant, si l'esclau, per la raó que sigui, necessita més temps de l'esperat per preparar la informació, aquest mantindrà la línia SCL a un nivell baix fins que estigui llest

per enviar les dades. D'aquesta manera li està senyalitzant al mestre que ha d'esperar-se fins que la línia SCL sigui alliberada.

```
// slave_wait is asserted when master wants to drive SCL high, but the slave pulls it low
// slave_wait remains asserted until the slave releases SCL
always @(posedge Clk or negedge Rst_n)
  if (!Rst_n) slave_wait <= 1'b0; // No clock stretching on reset
  else      slave_wait <= (dScl_oen & ~sSCL) | (slave_wait & ~sSCL); // Clock stretching
when slave keeps SCL line low
```

Figura 10. Codi de generació d'estat d'espera *slave_wait*.

```
// generate clk_en signal
wire clk_en = TimerOut; // Core system clock given by the SCL timer output
assign TimerStart = !Enable | scl_sync; // Timer starts whenever it is enabled or a master
drives SCL high
assign TimerStop = slave_wait; // Timer stops whenever a slave sends wait states
```

Figura 11. Codi de control de d'aturada de Timer.

Com es mostra a les Figures 5 i 6, el *clock stretching* es gestiona via estats d'espera o *wait states*, al codi designats per la variable *slave_wait*. Es generen estats d'espera quan l'esclau manté la línia SCL en nivell baix ($\sim sSCL$) i s'atura el Timer ($TimerStop = slave_wait$). Les línies de codi anteriors sintetitzen l'estructura de la Figura 7.

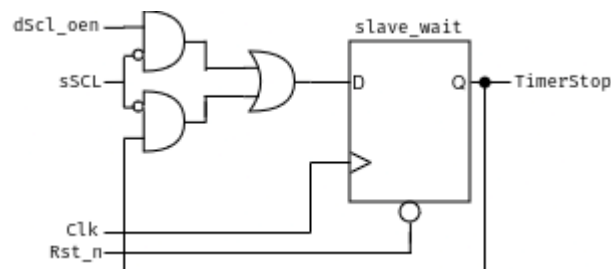


Figura 12. Esquemàtic RTL de la generació de *clock stretching*.

```
# [Info- 14245.00 ns] Test: clock streaching
```

Figura 13. Missatge de test de *clock stretching*.

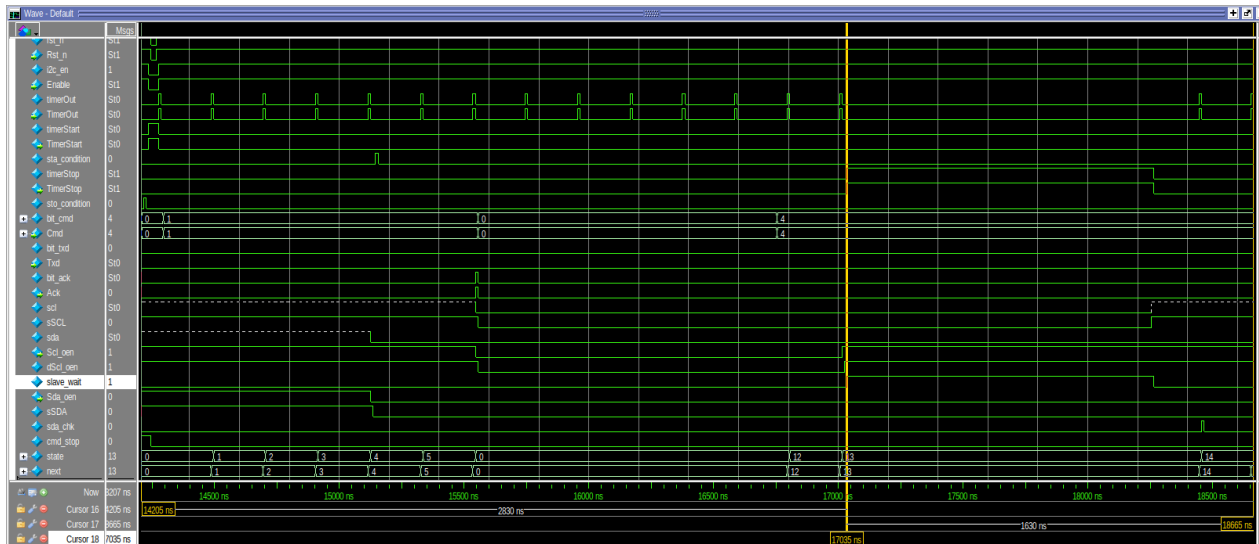


Figura 14. Diagrama d'ones de la simulació de *clock stretching*.

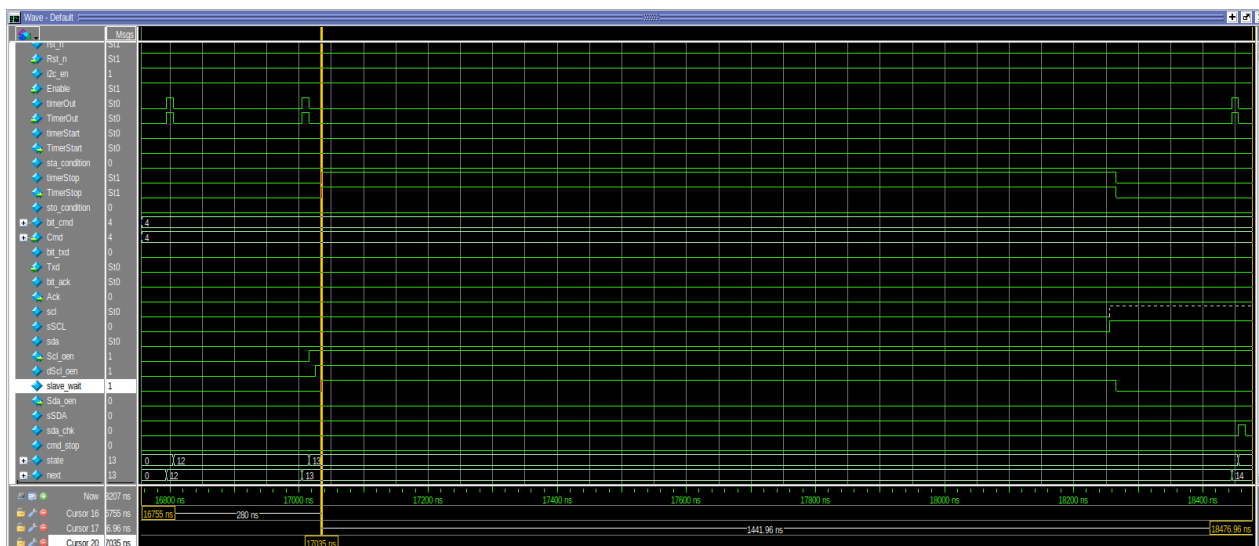


Figura 15. Ampliació de la Figura 14.

D'acord amb el codi de la Figura 10, a la posició del cursor de la Figura 14 es produeix un estat d'espera *slave_wait*, ja que les variables *dSCL_oen* i *sSCL* estan en nivell alt. Concurrentment, el *TimerStop* commuta, donat que està assignat continuament a *slave_wait* (vegeu codi Figura 11). Es roman en aquest estat fins que l'esclau allibera la línia SCL, aleshores el *slave_wait* i, en conseqüència, el *TimerStop* es posen a nivell baix i el *clock_stretching* cessa.

La síntesi, realitzada sobre la placa Cyclone IV-E EP4CE22F17C6, genera la següent lògica:

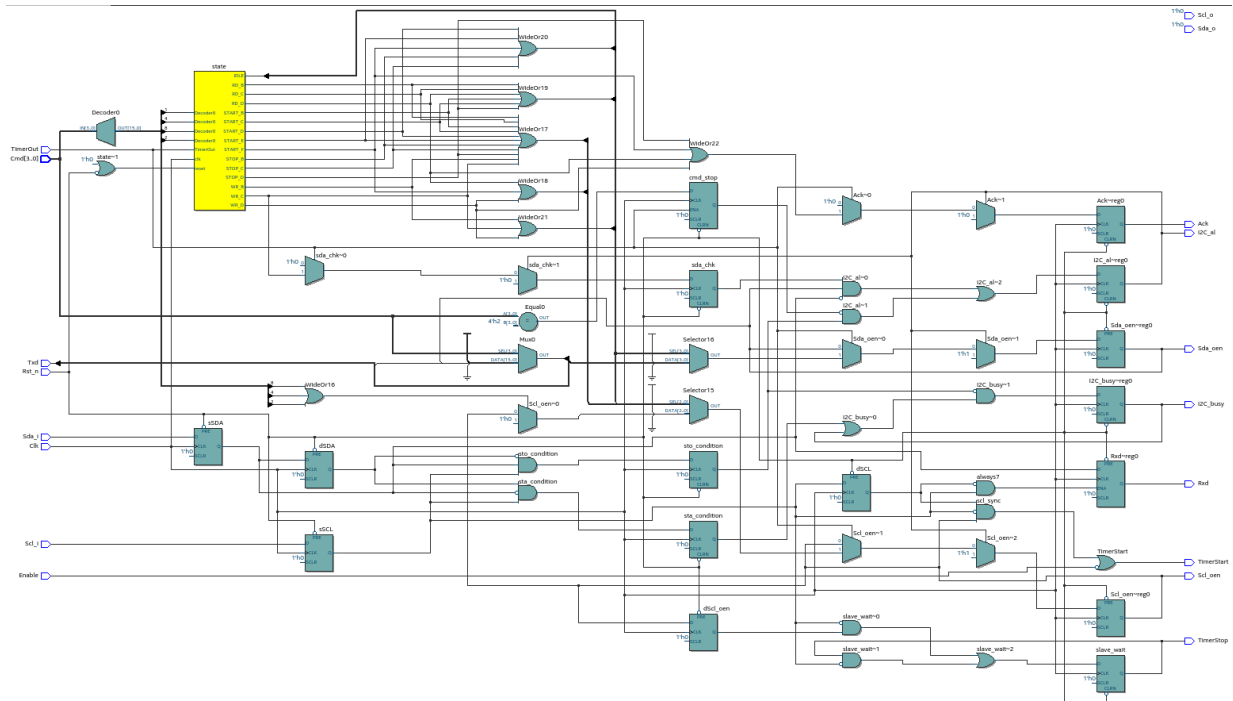


Figura 16. Esquemàtic RTL resultant de la síntesi del controlador de Bit de la unitat de control I2C.

	Resource	Usage
1	Estimated Total logic elements	47
2		
3	Total combinational functions	34
4	▼ Logic element usage by number of LUT Inputs	
1	-- 4 Input functions	18
2	-- 3 Input functions	10
3	-- <=2 Input functions	6
5		
6	▼ Logic elements by mode	
1	-- normal mode	34
2	-- arithmetic mode	0
7		
8	▼ Total registers	31
1	-- Dedicated logic registers	31
2	-- I/O registers	0
9		
10	I/O pins	21
11		
12	Embedded Multiplier 9-bit elements	0
13		
14	Maximum fan-out node	Clk-input
15	Maximum fan-out	31
16	Total fan-out	251
17	Average fan-out	2.35

Taula 2. Llista de recursos utilitzats en la implementació del controlador de Bit de la unitat de control I2C.