

Segundo Parcial

Código de FSM semáforo

El código lo dividimos en 3 módulos:

FSM_TOP:

En este mandamos a llamar las funciones de clk_psc y FSM_Semaforo como instancias. En este también tenemos declaradas las entradas y salidas, tenemos los switches del 0 al 5 como entradas y los leds del 0 al 10 como salidas.

```
1  `timescale 1ns / 1ps
2
3
4  module FSM_Top(input  logic CLK100MHZ,
5                 input  logic [5:0]sw,
6                 output logic [10:0]led
7                 );
8
9     logic internal_psc_clock;
10    logic int_reset = 1'b0;
11
12    clk_psc fsmclock (.my_clk(CLK100MHZ), .my_output(internal_psc_clock));
13
14    FSM_Semaforo semaforo (.clk(internal_psc_clock), .reset(sw[0]), .TA(sw[1]), .TB(sw[2]),
15                          .E(sw[3]), .PeatonB(sw[4]), .PeatonAv(sw[5]),
16                          .LARojo(led[0]), .LAAmarillo(led[1]), .LAVerde(led[2]), .LBRojo(led[5]), .LBAmarillo(led[6]),
17                          .LBVerde(led[7]), .PeatonAvRojo(led[3]), .PeatonAvVerde(led[4]), .PeatonBRojo(led[8]), .PeatonBVerde(led[9])
18                          );
19    assign led[10] = internal_psc_clock;
20
21 endmodule
```

Clk_PSC:

En este programa lo que hacemos es aprovechar el pulso de el reloj de 100Mhz que tenemos en la basys 3 para crear un programa que haga gastar recursos lo que hace que se cree un delay.

```
1  `timescale 1ns / 1ps
2
3
4  module clk_psc(input logic my_clk,
5                 output logic my_output);
6
7     logic [31:0]myreg;
8
9     always @(posedge my_clk)
10         myreg +=1;
11
12     assign my_output = myreg[28];
13
14 endmodule
15
```

FSM_Semaforo:

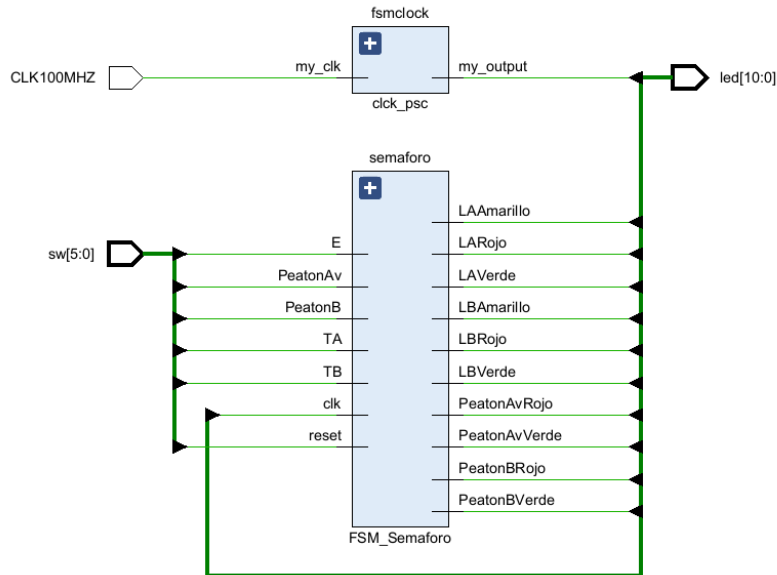
En este se describe el funcionamiento de la maquina de estados finitos, donde se representan todas las entradas y cada una de las salidas de los semáforos, también podemos ver las etapas de la maquina de estados como el next state logic, state register y el output logic.

```
1  `timescale 1ns / 1ps
2
3  module FSM_Semaforo(
4      input  logic clk,
5      input  logic reset,
6      input  logic TA, TB, E, PeatonB, PeatonAv,
7      output logic LARojo, LAAmarillo, LAVerde, LBRojo, LBamarillo, LBVerde, PeatonAvRojo, PeatonAvVerde, PeatonBROjo, PeatonBVerde
8  );
9
10     typedef enum logic [2:0]{S0,S1,S2,S3} statetype;
11     statetype state, nextstate;
12
13     logic LARojol, LAamarillol, LAVerdel, LBRojol, LBamarillol, LBVerdel, PeatonAvRojol, PeatonAvVerdel, PeatonBROjol, PeatonBVerdel;
14
15     always_ff @(posedge clk, posedge reset)
16     if (reset) state <= S0;
17     else state <= nextstate;
18
19     always_comb
20     case (state)
21         S0: if((TA)&&(!PeatonAv)) nextstate = S0;
22             else if((!TA)&&(!PeatonAv)) nextstate = S1;
23             else if(PeatonAv) nextstate = S1;
24         S1: if(!E) nextstate = S2;
25             else if(E) nextstate = S0;
26         S2: if((TB)&&(!E)&&(!PeatonB)) nextstate = S2;
27             else if((E)) nextstate = S0;
28             else if((!TB)&&(!E)&&(!PeatonB)) nextstate = S3;
29             else if((!E)&&(PeatonB)) nextstate = S3;
30         S3: nextstate = S0;
31         default: nextstate = S0;
32     endcase
33
34     always_comb
35     case (state)
36         S0: begin
37             {LARojol, LAamarillol, LAVerdel, LBRojol, LBamarillol, LBVerdel, PeatonAvRojol, PeatonAvVerdel, PeatonBROjol, PeatonBVerdel} = 10'b0011001001;
38             end
39         S1: begin
40             {LARojol, LAamarillol, LAVerdel, LBRojol, LBamarillol, LBVerdel, PeatonAvRojol, PeatonAvVerdel, PeatonBROjol, PeatonBVerdel} = 10'b0101001010;
41             end
42         S2: begin
43             {LARojol, LAamarillol, LAVerdel, LBRojol, LBamarillol, LBVerdel, PeatonAvRojol, PeatonAvVerdel, PeatonBROjol, PeatonBVerdel} = 10'b1000010110;
44             end
45         S3: begin
46             {LARojol, LAamarillol, LAVerdel, LBRojol, LBamarillol, LBVerdel, PeatonAvRojol, PeatonAvVerdel, PeatonBROjol, PeatonBVerdel} = 10'b1000101010;
47             end
48         default: ;
49     endcase
50
51     assign LARojo = LARojol;
52     assign LAAmarillo = LAamarillol;
53     assign LAVerde = LAVerdel;
54     assign LBRojo = LBRojol;
55     assign LBamarillo = LBamarillol;
56     assign LBVerde = LBVerdel;
57     assign PeatonAvRojo = PeatonAvRojol;
58     assign PeatonAvVerde = PeatonAvVerdel;
59     assign PeatonBROjo = PeatonBROjol;
60     assign PeatonBVerde = PeatonBVerdel;
61
62 endmodule
63
```

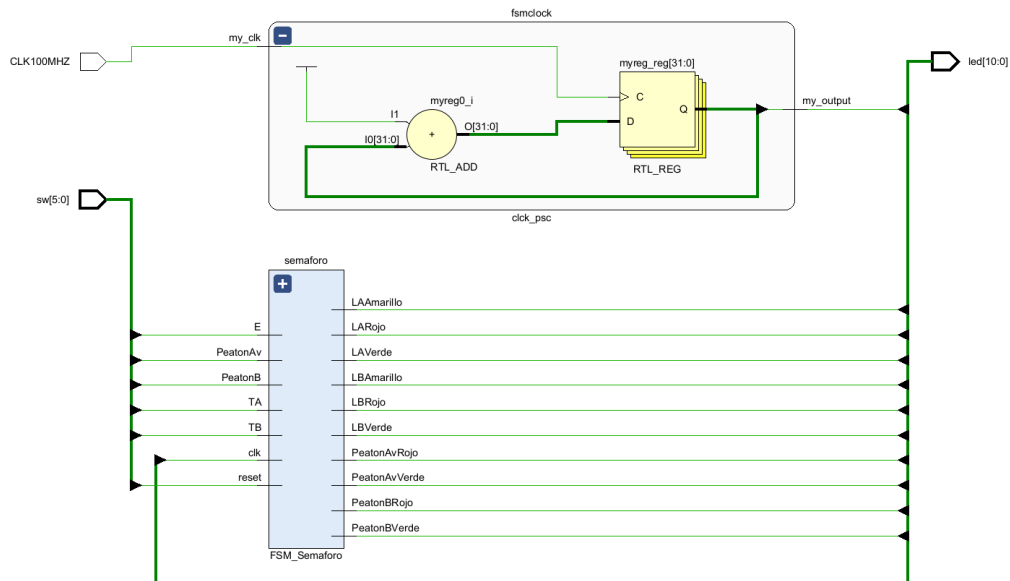
1. RTL Analysis

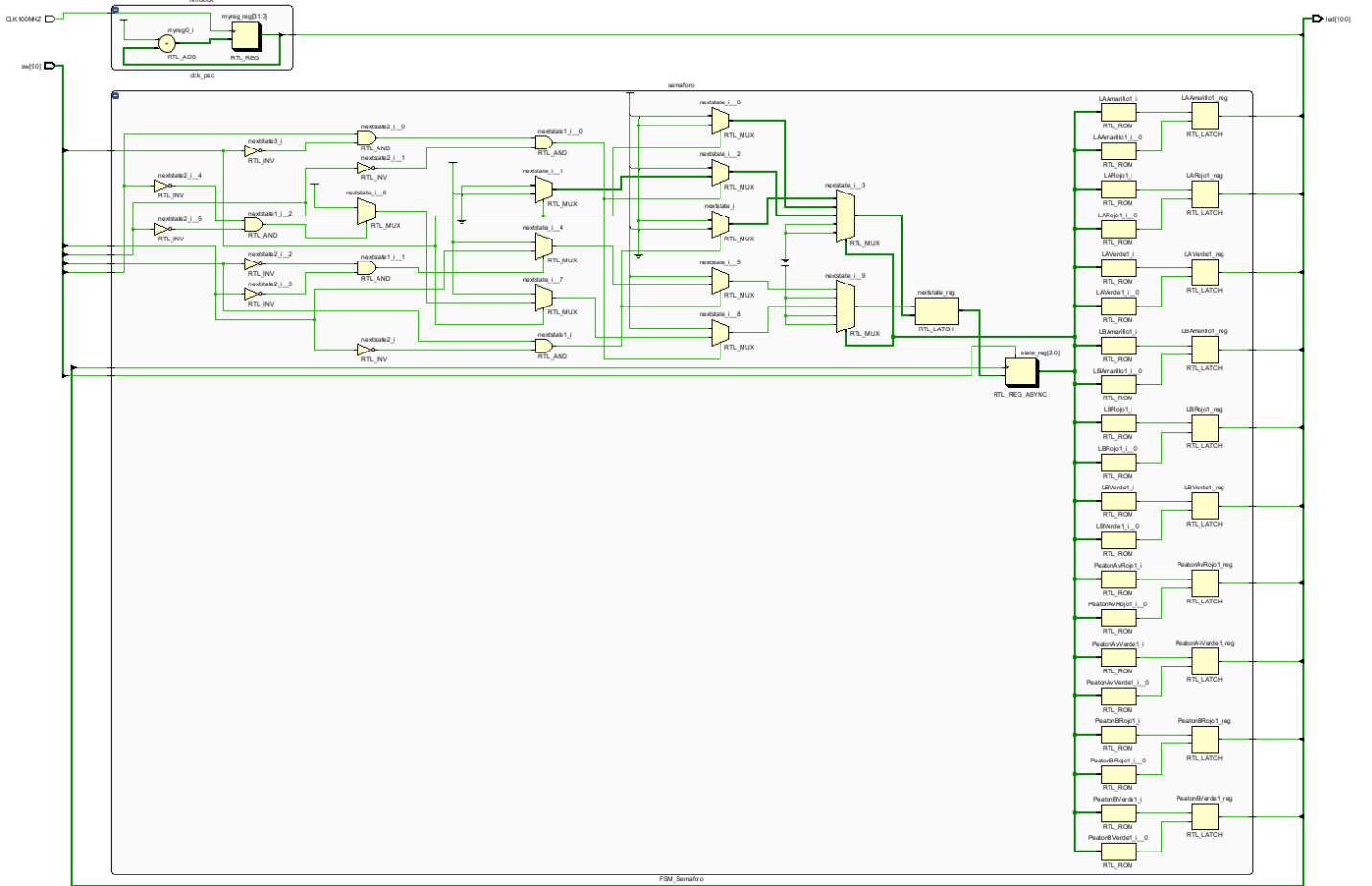
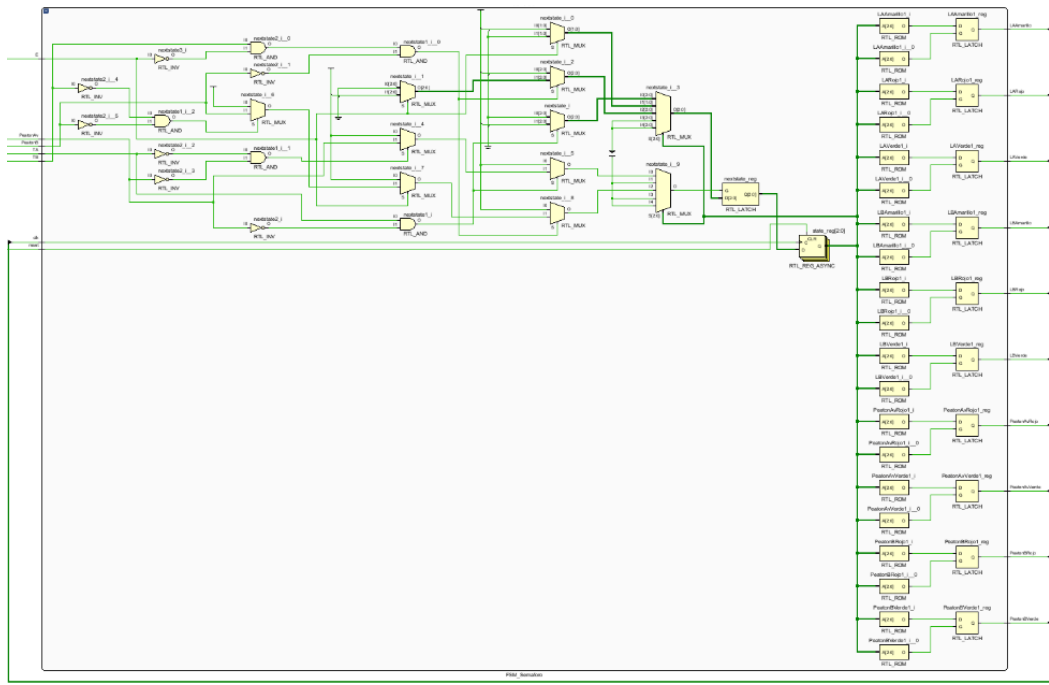
En esta etapa se verifica la sintaxis del código, código en el cual describimos el sistema utilizando HDL, se verifica también los constraints y que todo cumpla correctamente, también se hace un esquemático de el sistema descrito en código.

Por ejemplo, en esta ocasión ya que se crean dos módulos, en el esquemático podemos ver los módulos representados como bloques.



Al seleccionar cada módulo podemos ver la circuitería interna:

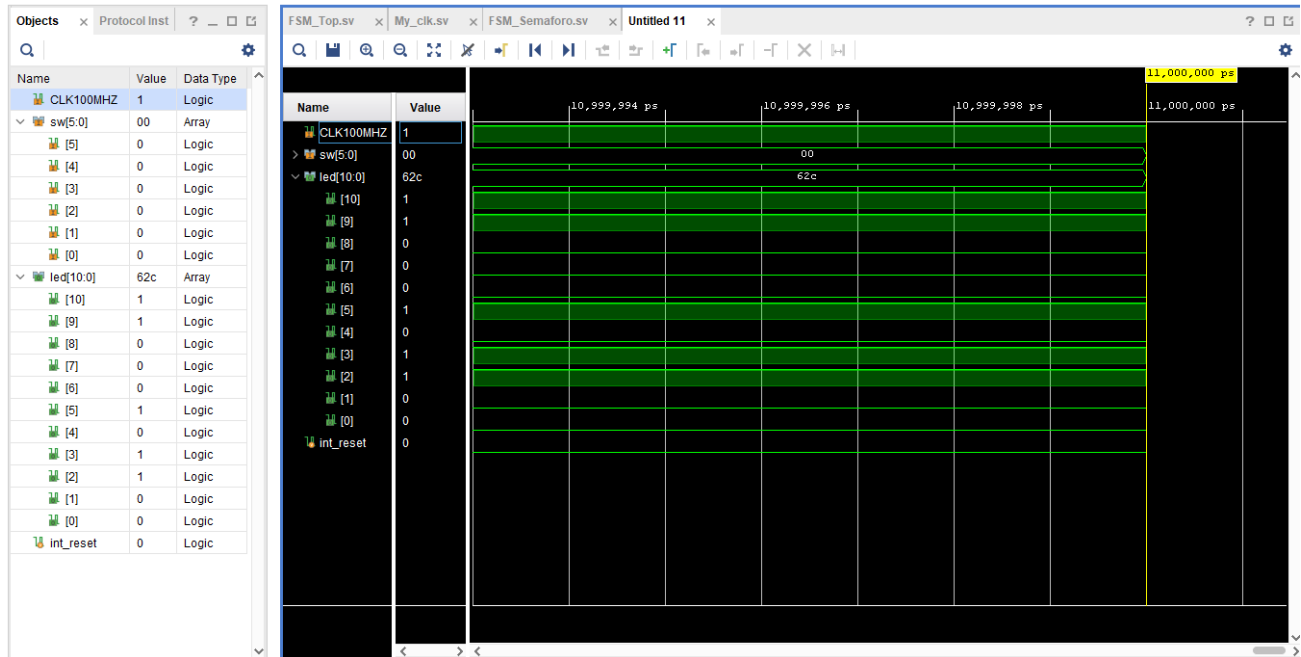




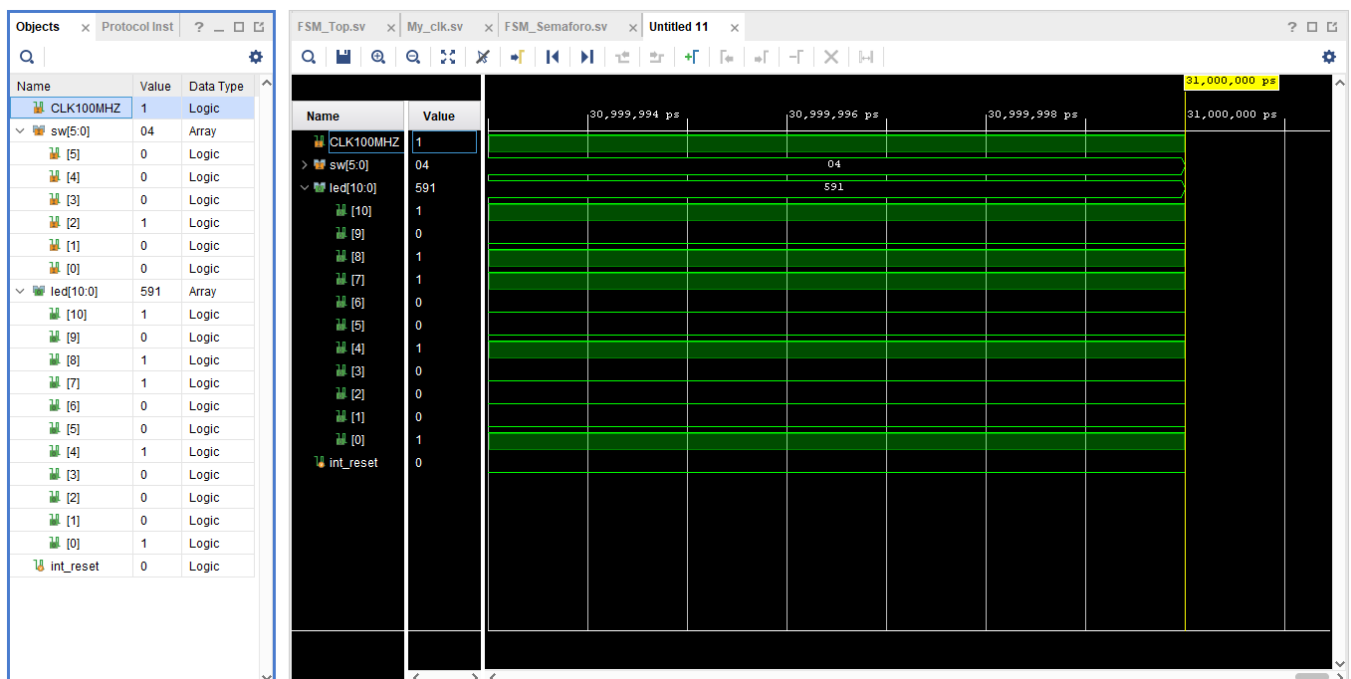
2. Simulación

En esta parte se verifica el desempeño del código, si las funcionalidades y especificaciones satisfacen con las especificaciones del sistema, esta disponible en el pre y post de varias etapas.

Por ejemplo, podemos ir forzando constantes y corriendo la simulación para ver qué es lo que sucede en las salidas, en este forzamos los switches y corremos la simulación nos proporciona las salidas exactas para el estado S0, y del lado derecho podemos ver los cambios de los estados conforme al tiempo.

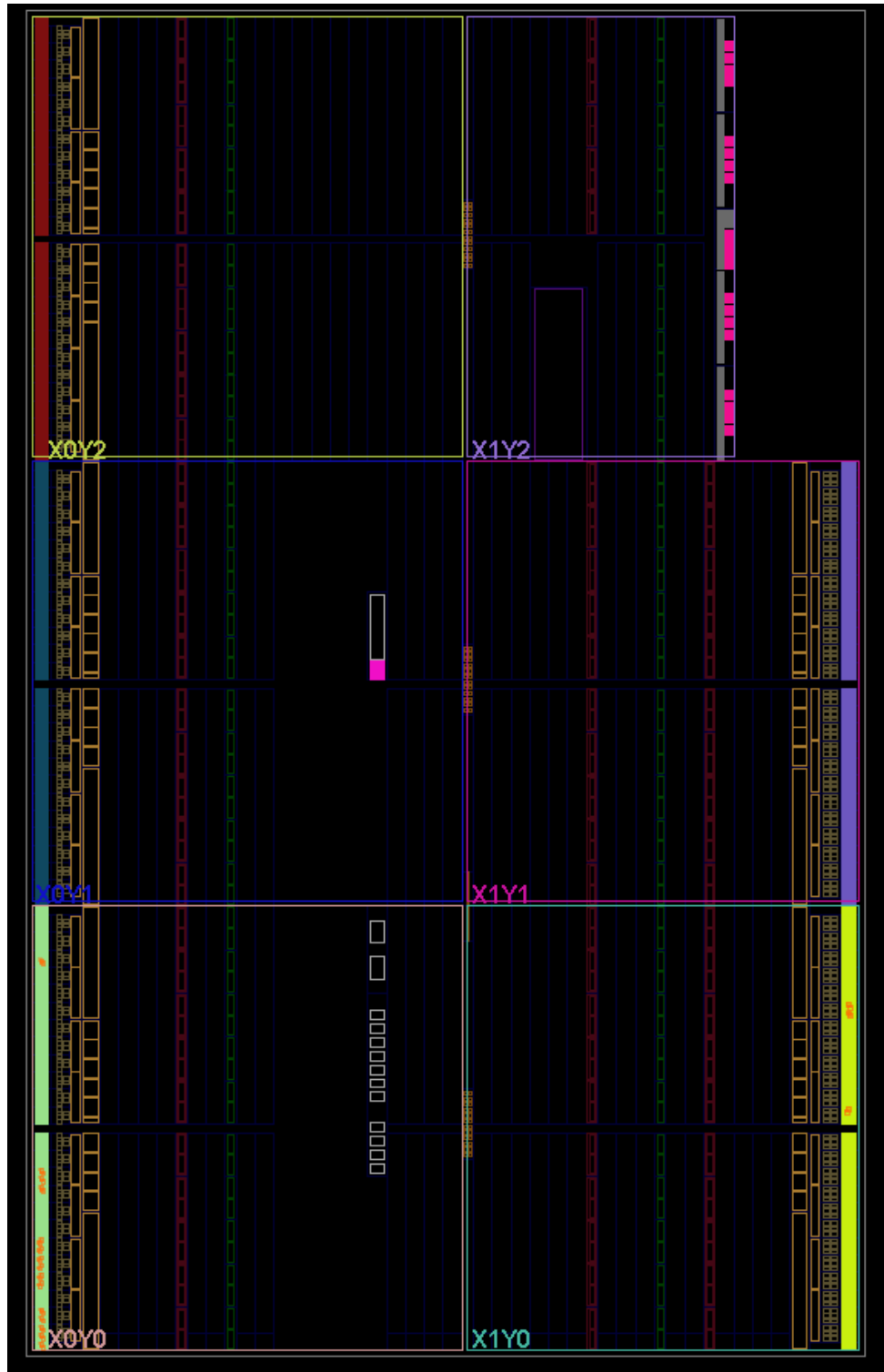


Aquí tenemos una segunda simulación donde ponemos 1 en el Switch 2 lo que proporciona un cambio en las salidas, y podemos verlo del lado derecho.

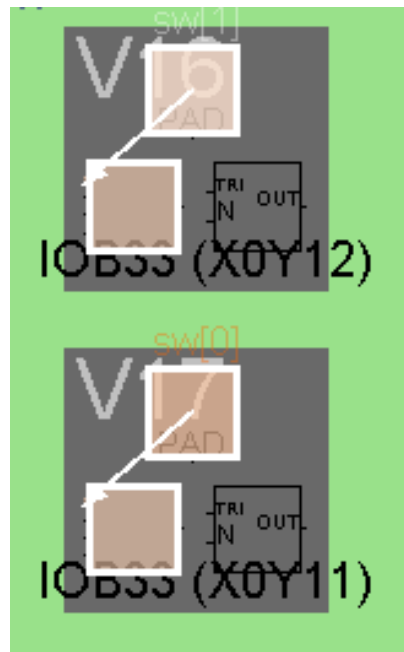
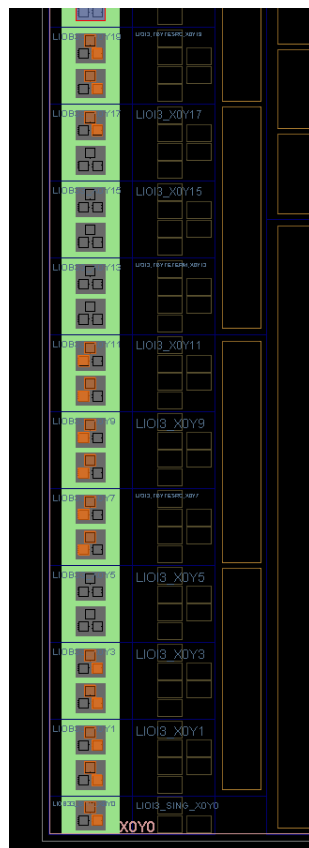


3. Synthesis

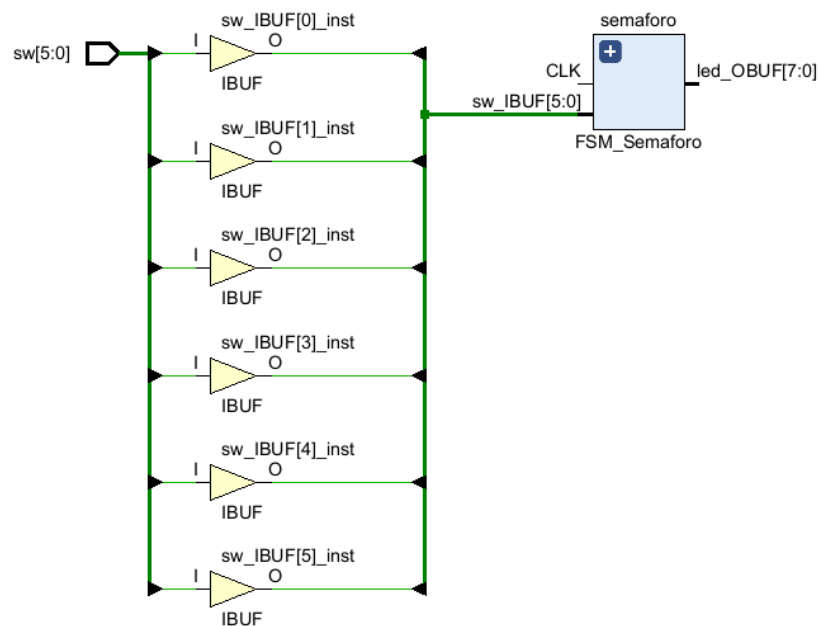
Al realizar la synthesis podemos ver si nuestro dispositivo es capaz de poder realizar la función de nuestro código, con los componentes que este posee. A continuación, podemos ver la basys3, y las partes resaltadas son los componentes que se utilizaran para correr este código.

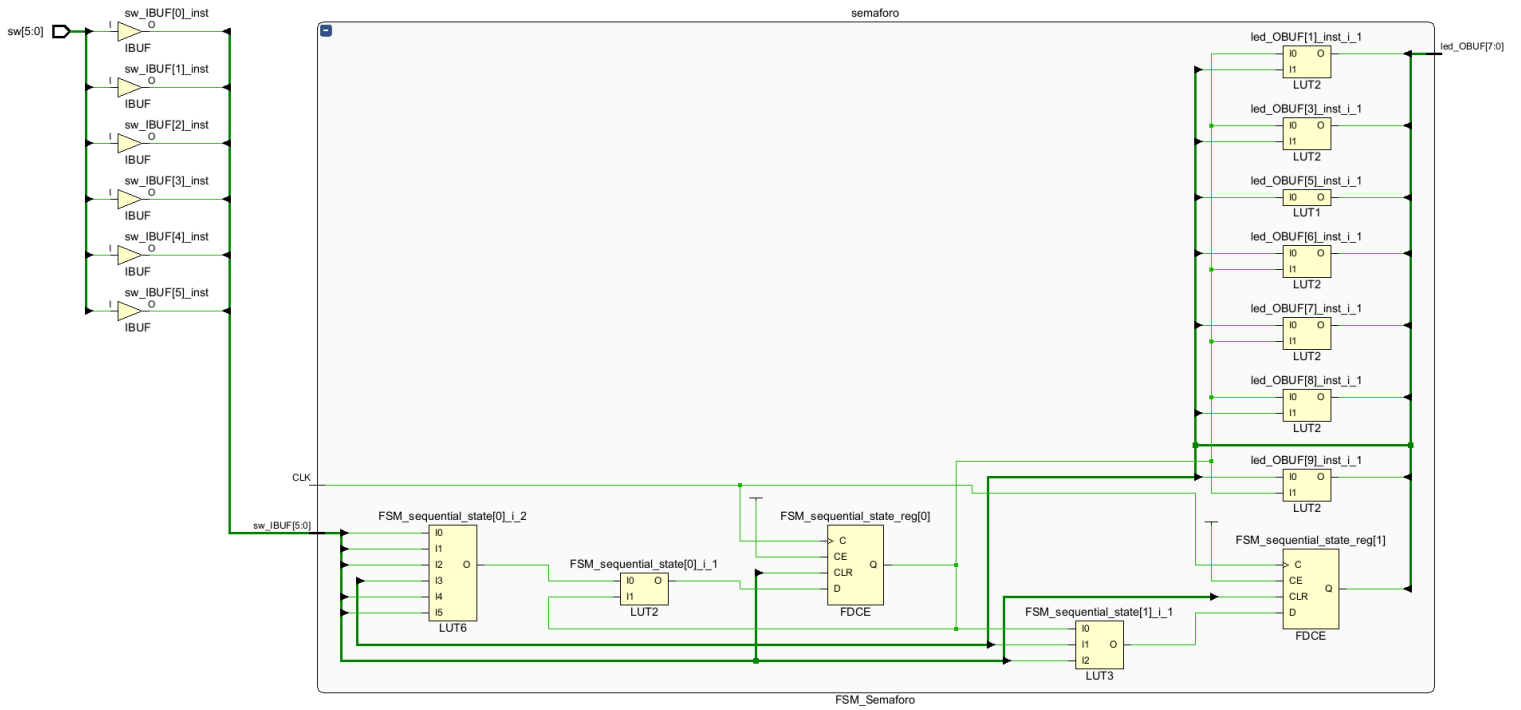


Si hacemos zoom a estos componentes marcados podemos ver conectividad y de que se trata.



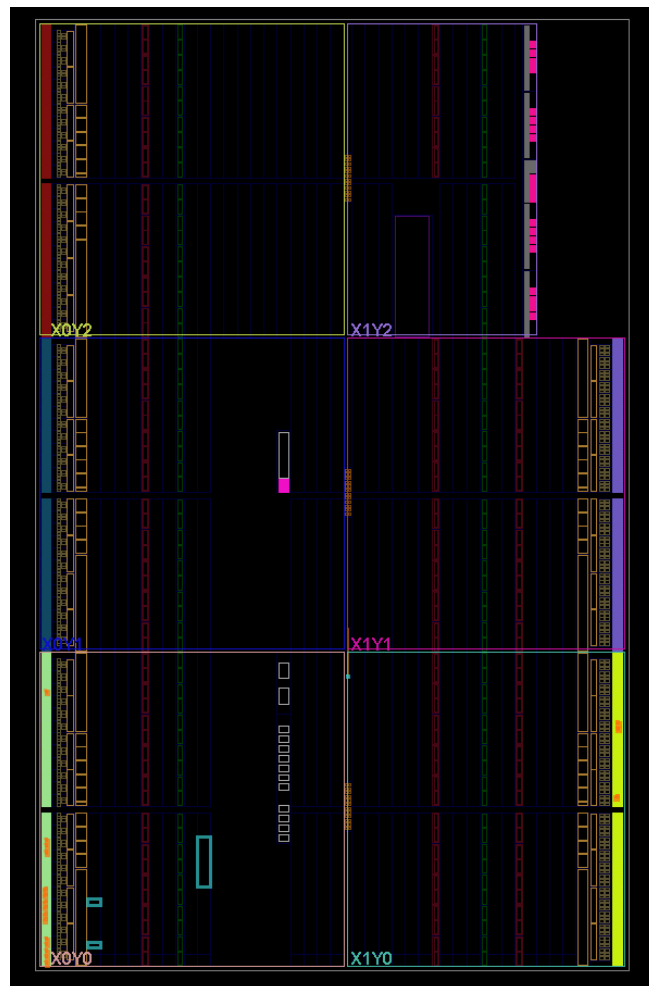
También lo que podemos observar es el esquemático con los componentes que se usaran de nuestra placa basys3, podemos ver que LUTs se vana utilizar y que otros componentes son los que se usaran para realizar los semáforos.





4. Implementación

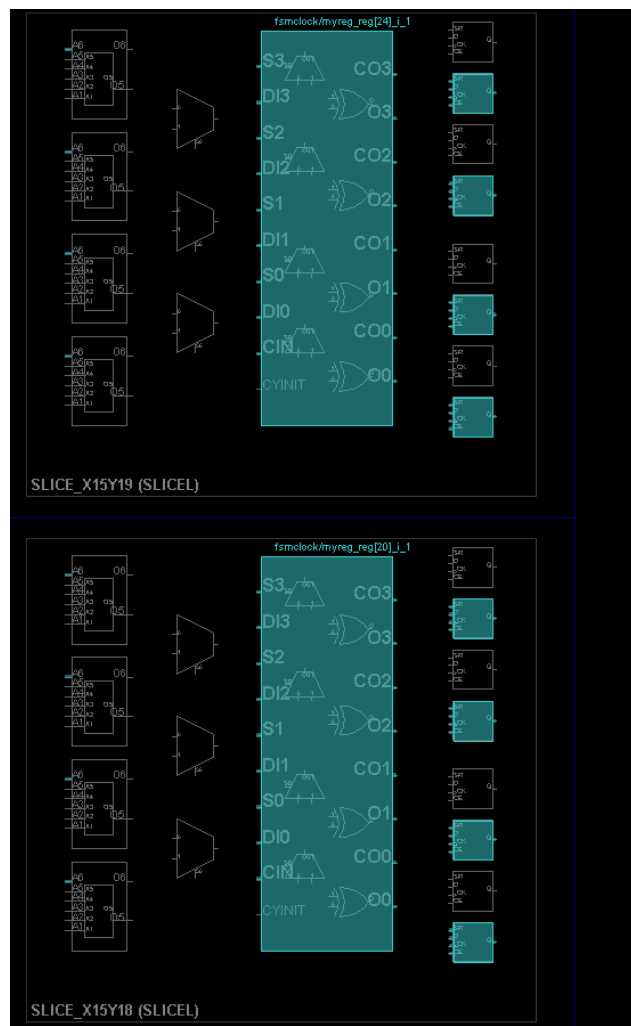
En este apartado ya podemos ver las partes que se van a utilizar para la implementación, así como las conexiones que se realizan, por ejemplo, podemos ver que se genera una imagen como en synthesis pero ahora con ciertas slice resaltadas.



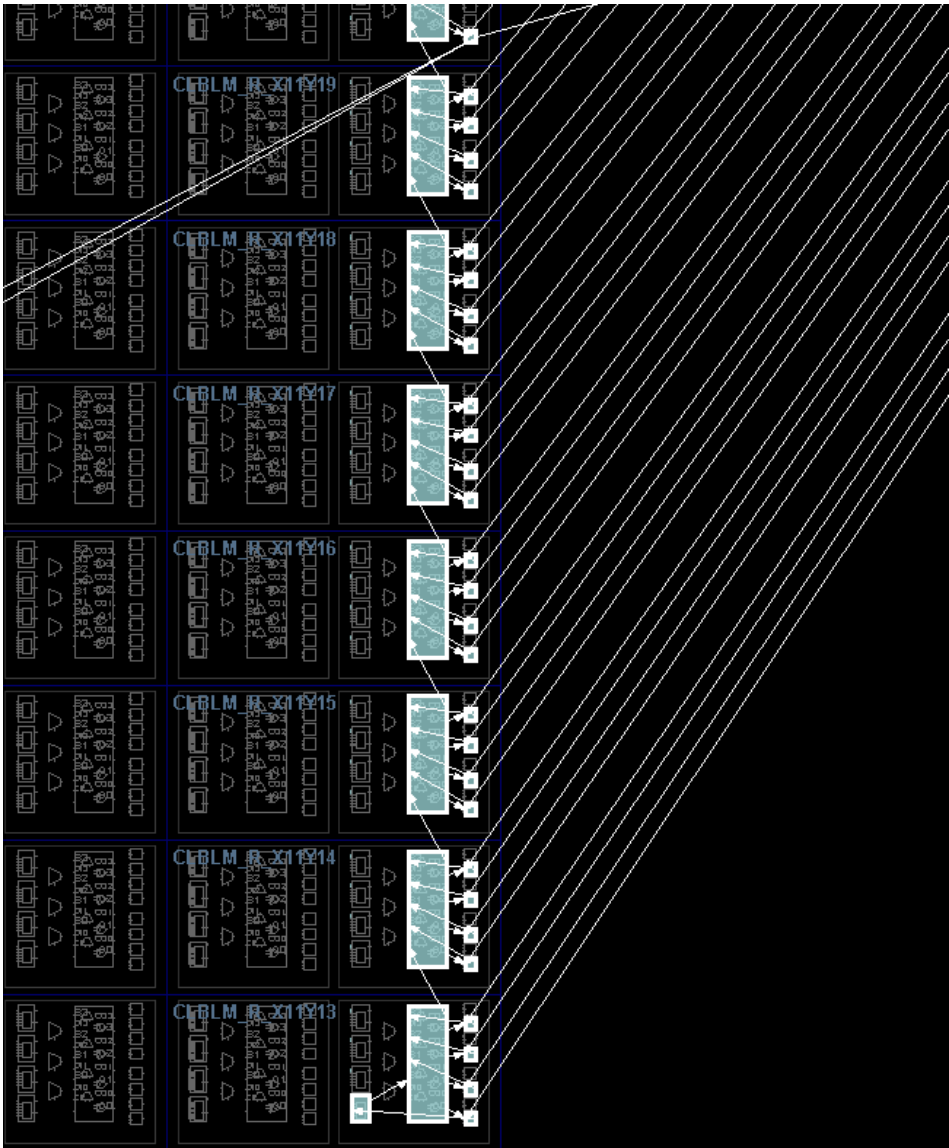
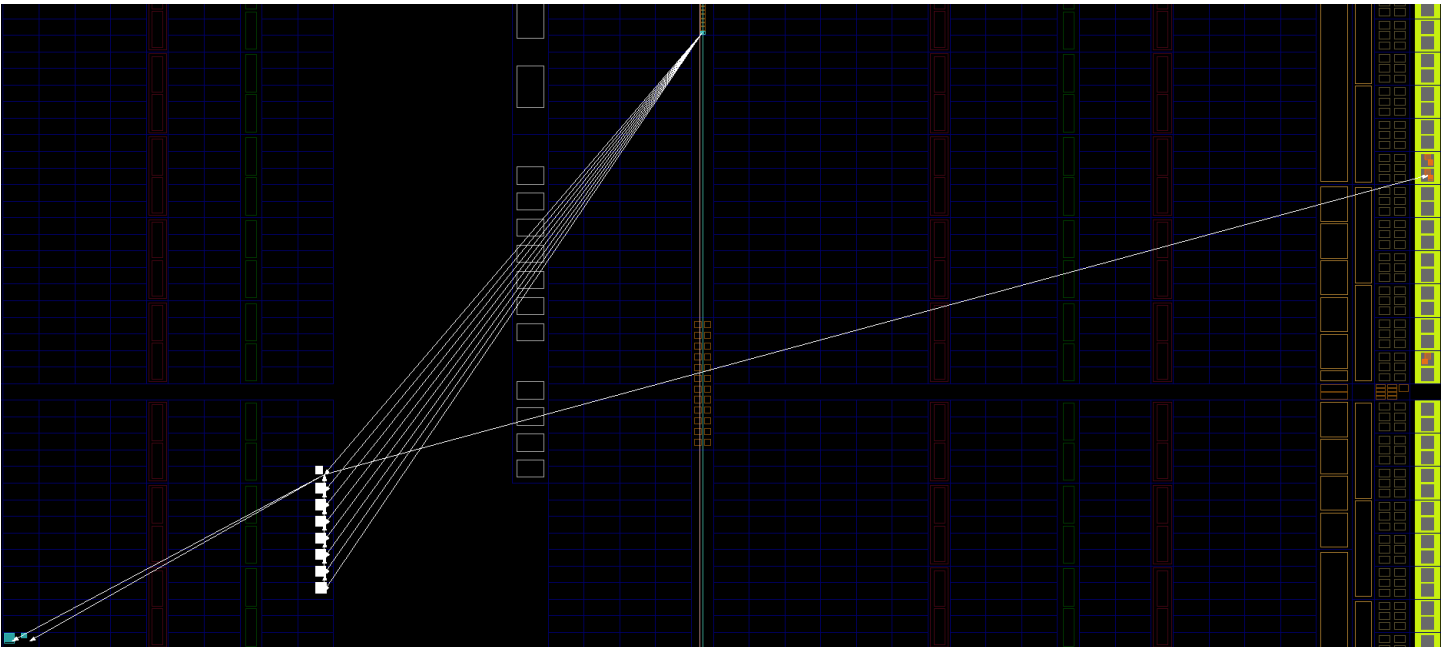
Si vamos haciendo zoom por partes podemos ver las slice que están seleccionadas,




Si nos acercamos aún más podemos ver los componentes de los slice que se están utilizando.



Por último, podemos seleccionar los componentes y ver sus conexiones.



5. Program and Debug

- ▼ PROGRAM AND DEBUG
 -  [Generate Bitstream](#)
 - ▼ Open Hardware Manager
 - Open Target
 - Program Device
 - Add Configuration Memory Device

Por último, en este apartado lo que hacemos es generar un bitstream que configura las celdas lógicas y los switches programables, genera acorde al netlist final, luego de generarlo y conectar la basys3 podemos cargar y descargar hacia el FPGA utilizando diversos métodos.