Diseño de Sistemas Digitales
Práctica 2: Diseño de un Multiplicador Secuencial

Realizar el diseño, verificación e implementación de un multiplicador secuencial paramétrico de *N* bit por *N* bits.

Especificaciones:

1. La interfaz del multiplicador debe ser como se muestra en la Fig. 1

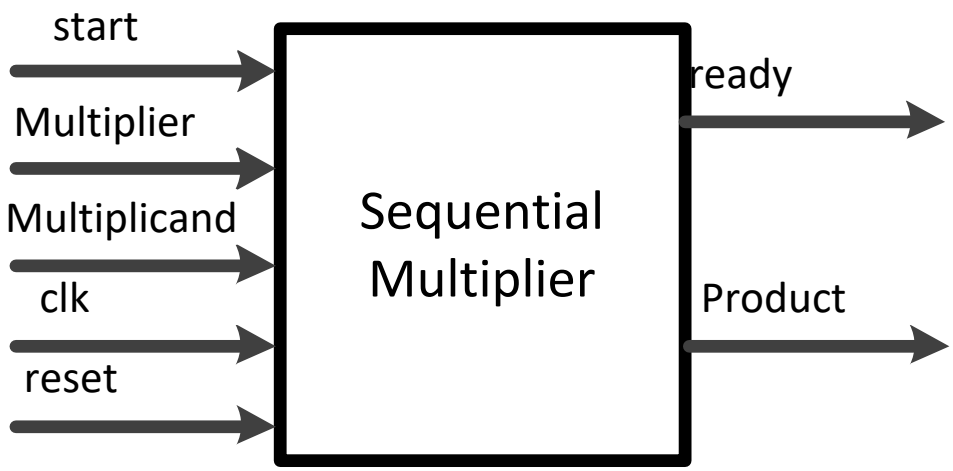


Fig. 1 Caja negra del bloque multiplicador secuencial.

2. El Sequential_Multiplier calcula la multiplicación entre dos números enteros en sistema numérico complemento a 2 de *N*-bits (*Multiplier* X *Multiplicand*).
3. El algoritmo de multiplicación debe ser secuencial. Una señal de reloj debe marcar el ritmo del algoritmo de multiplicación: al menos se deben requerir *N* ciclos de reloj para obtener el resultado esperado (*Product*).
4. El tamaño de la salida *Product* debe ser de 2N bits.
5. El procesamiento debe iniciar cuando la señal de *start* = 1.
6. Cuando el resultado *Product* esté completo el puerto de salida *ready* debe ser activado. La siguiente Fig. 2 muestra un ejemplo de simulación de una multiplicación.

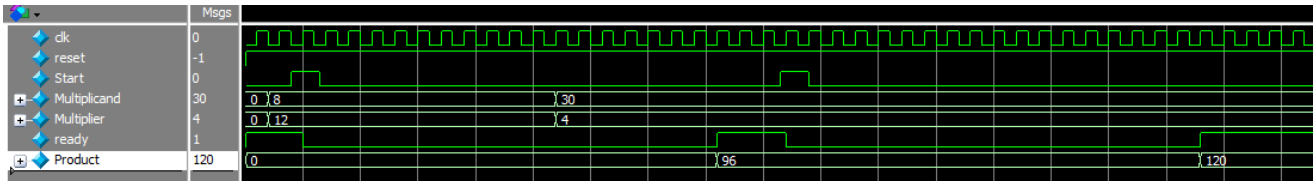


Fig. 2 Formas de onda de la simulación del multiplicador.

7. El diseño de mayor jerarquía del multiplicador secuencial debe estar modelado con Verilog a nivel estructural: debe instanciar los sub-modulos, los cuales deben ser bloques de uso frecuente claramente identificables: registros, contadores, registros de corrimiento, multiplexores, sumadores, decodificadores máquinas de estado, etc.
8. La verificación del multiplicador se realiza en el simulador considerando N = 8.
9. Para la implementación en la tarjeta DE10-Standard limitar el tamaño de las entradas a 5 bits e incluir un decodificador de binario a BCD para mostrar los resultados en el display en decimal signado.
10. En la implementación, no se deben mostrar en display los productos intermedios o productos parciales. El resultado solo debe actualizarse después de presionar *start* y se haya generado la señal de *ready*.

Entregables

a) El proyecto comprimido del diseño top generado con la herramienta Quartus, subido en el link correspondiente.
b) Reporte subido en Canvas en la fecha de entrega definida. Usa este mismo documento para entregar tu reporte. El reporte debe contener los siguientes puntos.

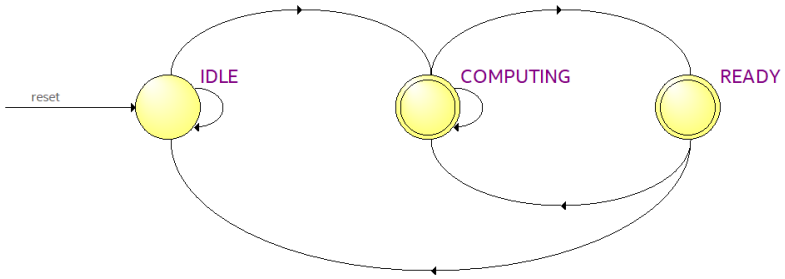
1. Explicación del algoritmo de multiplicación secuencial que usaste y planteamiento de la solución al diseño [10%].

   El método o algoritmo que se esta implementando para el desarrollo del multiplicador secuencial es el método “shift and add method” Si cualquier bit en el multiplicador (b) es 0, entonces el multiplicando (a) se suma con cero. Se utiliza un sumador que tiene la misma longitud que los operandos. La salida del sumador y el multiplicador se aumenta en un banco de registro. Después de cada adición, el contenido del banco de registros se desplaza a la derecha.

2. Diagrama de bloques (arquitectura) del diseño de mayor jerarquía y diagrama de estados de la FSM para controlar los módulos del multiplicador sequencial [15%].

Arquitectura:

Diaframa de Maquina de estados:



3. Modelos Verilog comentados, de cada uno de los bloques del sistema y del diseño de mayor jerarquía [25%].

Modulo TOP, Sequential_Multiplier

```
module Sequential_Multiplier #(parameter LENGTH = 5)(
    //Entradas necesarias para el funcionamiento del multiplicador secuencial
     input start, clock, reset,
     input [LENGTH-1:0]multiplier,
     input [LENGTH-1:0]multiplicand,
     //Salidas para los displays de 7 segmentos.
     output seg_a_disp0, seg_b_disp0, seg_c_disp0, seg_d_disp0, seg_e_disp0, seg_f_disp0, seg_g_disp0,
     output seg_a_disp1, seg_b_disp1, seg_c_disp1, seg_d_disp1, seg_e_disp1, seg_f_disp1, seg_g_disp1,
     output seg_a_disp2, seg_b_disp2, seg_c_disp2, seg_d_disp2, seg_e_disp2, seg_f_disp2, seg_g_disp2,
     //Banderas de estados
     output Computing, Ready, Negative
     );
    //Variables para el registro de las entradas
     wire [LENGTH-1:0]multiplierRegister;
     wire [LENGTH-1:0]multiplicandRegister;

     //Resultados
     wire [LENGTH*2-1:0]productRegisterIn;
     wire [LENGTH*2-1:0]productRegisterOut;

     // Salidas del BCD
     wire [3:0] unos;
     wire [3:0] dieces;
     wire [3:0] cientos;

 //Registro de la entrada multiplier
register #(.LENGTH(LENGTH)) registerMultiplier_TOP(
     .clock(clock),
     .reset(~reset),
     .enable(~start),
     .D(multiplier),
     .Q(multiplierRegister)
     );

 //Registro de la entrada multiplicand
register #(.LENGTH(LENGTH)) registerMultiplicand_TOP(
     .clock(clock),
     .reset(~reset),
     .enable(~start),
     .D(multiplicand),
     .Q(multiplicandRegister)
     );
```

```
44    //FSM Donde se encuentra toda la logica de los estados
45    FSM_SecuentialMultiplier #(.LENGTH(LENGTH)) FSM(
46        .clock(clock),
47        .reset(~reset),
48        .start(~start),
49        .multiplicand(multiplicandRegister),
50        .multiplier(multiplierRegister),
51        .result(productRegisterIn),
52        .Computing(Computing),
53        .Ready(Ready),
54        .Negative(Negative)
55        );
56
57    //Registro del resultado
58    register #(.LENGTH(LENGTH*2)) registerResult_TOP(
59         .clock(clock),
60         .reset(~reset),
61         .enable(Ready),
62         .D(productRegisterIn),  //this should be provied by secuential multiplier (ready result) module
63         .Q(productRegisterOut)
64         );
65
66    //Se envia el resultado al modulo de binario a BCD
67    BinaryToBCD BinaryToBCD_TOP(
68         .clk(clock),
69         .binario(productRegisterOut[7:0]),
70         .unos(unos),
71        .dieces(dieces),
72        .cientos(cientos)
73         );
74
75    //Se envian los 4 bits que le pertenecen a cada numero de la salida del BCD a cada uno de los displays de 7 segmentos
76    BinaryToHexadecimal BinaryToHexadecimal_display0_TOP(
77        .w(unos[3]),
78        .x(unos[2]),
79        .y(unos[1]),
80        .z(unos[0]),
81         .seg_a(seg_a_disp0),
82         .seg_b(seg_b_disp0),
83         .seg_c(seg_c_disp0),
84         .seg_d(seg_d_disp0),
85         .seg_e(seg_e_disp0),
86         .seg_f(seg_f_disp0),
87         .seg_g(seg_g_disp0)
88         );
89
90    //Se envian los 4 bits que le pertenecen a cada numero de la salida del BCD a cada uno de los displays de 7 segmentos
91    BinaryToHexadecimal BinaryToHexadecimal_display1_TOP(
92        .w(dieces[3]),
93        .x(dieces[2]),
94        .y(dieces[1]),
95        .z(dieces[0]),
96         .seg_a(seg_a_disp1),
97         .seg_b(seg_b_disp1),
98         .seg_c(seg_c_disp1),
99         .seg_d(seg_d_disp1),
100        .seg_e(seg_e_disp1),
101        .seg_f(seg_f_disp1),
102        .seg_g(seg_g_disp1)
103        );
104
105   //Se envian los 4 bits que le pertenecen a cada numero de la salida del BCD a cada uno de los displays de 7 segmentos
106   BinaryToHexadecimal BinaryToHexadecimal_display2_TOP(
107       .w(cientos[3]),
108       .x(cientos[2]),
109       .y(cientos[1]),
110       .z(cientos[0]),
111        .seg_a(seg_a_disp2),
112        .seg_b(seg_b_disp2),
113        .seg_c(seg_c_disp2),
114        .seg_d(seg_d_disp2),
115        .seg_e(seg_e_disp2),
116        .seg_f(seg_f_disp2),
117        .seg_g(seg_g_disp2)
118        );
119
120   endmodule
```

Modulo register:

```
1  module register
2      #(parameter LENGTH = 5)
3      (
4      //Entradas
5          input clock,reset,enable,
6          input [LENGTH-1:0]D,
7      //Salidas
8          output reg [LENGTH-1:0]Q
9      );
10
11 //Se activa en el flanco positivo del clock o del reset
12 always @(posedge clock, posedge reset)
13     begin
14     //En reset, resetear el valor de Q (puede ser las entradas B, A por ejemplo)
15         if (reset)
16             Q <= {LENGTH{1'b0}};
17     //Si se activo el always y fue por el clock
18         else
19         //Si es enable, se pasa el valor al registro para poder ser utilizado en la ALU
20             if (enable)
21                 Q <= D;
22        //Si no se apreto el boton de enable, el registro sigue teniendo el mismo valor de antes.
23             else
24                 Q <= Q;
25     end
26 endmodule
```

Modulo FSM_SecuentialMultiplier:

```
1  module FSM_SecuentialMultiplier #(parameter LENGTH = 5)(
2  //Entradas necesarias para el funcionamiento de la FSM
3      input clock, reset, start,
4      input [LENGTH-1:0]multiplicand,multiplier,
5  //Salidas de resultados y banderas de estado.
6      output reg [LENGTH*2-1:0] result,
7      output reg Computing, Ready, Negative
8  );
9
10 //Los diferentes estados de la FSM
11 localparam IDLE = 2'b00;
12 localparam COMPUTING = 2'b01;
13 localparam READY = 2'b10;
14
15 //Temporales utilizados para realizar la logica
16 reg [1:0]FSM_state;
17 reg readyResult_temp;
18 reg [LENGTH*2-1:0]Result_temp;
19 reg [LENGTH-1:0]multiplicand_temp,multiplier_temp;
20 reg [LENGTH*LENGTH*2-1:0] partials;
21 reg Negative_temp;
22 integer i;
23
24 //Estados de la FSM, que es lo que hace que cambiemos de estado
25 always @(posedge clock, posedge reset)
26 begin
27     if (reset)
28       begin
29         FSM_state <= IDLE;
30       end
31     else
32         case (FSM_state)
33             IDLE:
34                     if (start)
35                         FSM_state <= COMPUTING;
36             COMPUTING:
37                     if (readyResult_temp)
38                         FSM_state <= READY;
39             READY:
40                     if (start)
41                         FSM_state <= COMPUTING;
42                     else
43                         FSM_state <= IDLE;
44             default: FSM_state <= IDLE;
45         endcase
46 end
47
```

```
48    //Logica que se realiza en cada estado
49    always @(FSM_state)
50    begin
51        case (FSM_state)
52            IDLE: //No deberia mostrar mas que 0s
53                begin
54                  result <= {LENGTH*2-1{1'b0}};
55                  Computing <= 1'b0;
56                  Ready <= 1'b0;
57                end
58
59            COMPUTING: //Calculando el resultado
60                begin
61                  result <= {LENGTH*2-1{1'b0}};
62                    Computing <= 1'b1;
63                    Ready <= 1'b0;
64                end
65
66            READY: //Muestra el resultado y muestra que esta en ready
67                begin
68                    result <= Result_temp;
69                    Computing <= 1'b0;
70                    Ready <= 1'b1;
71                end
72
73            default: //Estado default
74                begin
75                  result <= {LENGTH*2-1{1'b0}};
76                  Computing <= 1'b0;
77                  Ready <= 1'b0;
78                end
79        endcase
80    end
```

```
81
82    //Logica que calcula el resultado de la multiplicacion, este resultado se utiliza en la parte de ready, en computing debera estarlo calculando y debera hacer calculo por calculo
83    //De acuerdo a los ciclos de reloj
84    always@(posedge clock)
85    begin
86        if (FSM_state == COMPUTING)
87            begin
88              multiplicand_temp <= multiplicand[LENGTH-1] ? ~multiplicand + 1'b1 : multiplicand;
89              multiplier_temp <= multiplier[LENGTH-1] ? ~multiplier + 1'b1 : multiplier;
90              partials[LENGTH*2-1:0] <= multiplicand_temp[0] ? multiplier_temp : {LENGTH{1'b0}};
91              if (i < LENGTH)
92                  begin
93                      partials[LENGTH*2*(i+1)-1-:LENGTH*2] <= (multiplicand_temp[i] ? (({LENGTH*2{1'b0}} + multiplier_temp ) << i) : 0) + partials[LENGTH*2*i-1-:LENGTH*2];
94                      i = i+1;
95                      readyResult_temp = 1'b0;
96                  end
97              else
98                  begin
99                      Result_temp <= partials[LENGTH*LENGTH*2-1:LENGTH*(LENGTH-1)*2];
100                     Negative <= multiplicand[LENGTH-1] ^ multiplier[LENGTH-1] ? 0 : 1;
101                     i = 1;
102                     readyResult_temp = 1'b1;
103                 end
104         end
105   end
106
107   endmodule
```

Modulo BinaryToBCD:

```
module BinaryToBCD (
//Entradas
input clk,
input [7:0] binario,
//Salidas
output reg [3:0] unos = 0,
output reg [3:0] dieces = 0,
output reg [3:0] cientos = 0);

//Utilizado como contador para la secuencia
reg [3:0] i = 0;
//Registro utilizado para realizar el corrimiento y guardar el valor
reg [19:0] registro_corrimiento;
//Guardadermos los 4 bits que se tomaran como los 1s en BCD
reg [3:0] temp_unos = 0;
//Guardadermos los 4 bits que se tomaran como los 10es en BCD
reg [3:0] temp_dieces = 0;
//Guardadermos los 4 bits que se tomaran como los 100es en BCD
reg [3:0] temp_cientos = 0;
//Registro que nos servira para checar si la entrada valor ha cambiado
reg [7:0] temp_binario = 0;

always@ (posedge clk)
begin
  // Si el contador se reinicio y el valor de la entrada binario ha cambiado
  if (i == 0 & (temp_binario != binario))
      begin
      //Inicializar el registro de corrimiento en 0
      registro_corrimiento = 20'd0;
      //Asignamos el valor de la variable binario a temp_binario, para que podamos saber despues si hemos tenido una nueva entrada
      temp_binario = binario;
      //Asignamos los 8 bits de la variable binario a los 8 bits menos significativos del registro_corrimiento
      registro_corrimiento[7:0] = binario;
      //Asignamos valores de los unos, dices y cientos a partir de la variable registro_corrimiento
      temp_unos = registro_corrimiento[11:8];
      temp_dieces = registro_corrimiento[15:12];
      temp_cientos = registro_corrimiento[19:16];
      //Aumentamos contador
      i=i+1;
      end
  if (i < 9 & i>0)
      begin
      //Checamos si las variables unos, dices y cientos son mayores o iguales a 5
      if (temp_unos>=5)
        temp_unos = temp_unos+3;
      if (temp_dieces>=5)
        temp_dieces = temp_dieces+3;
      if (temp_cientos>=5)
        temp_cientos = temp_cientos+3;
      //Metemos estos nuevos valores en la variable registro_corrimiento
      registro_corrimiento[19:8] = {temp_cientos,temp_dieces,temp_unos};
      //Despues hacemos un shift left
      registro_corrimiento = registro_corrimiento<<1;
      //Y actualizamos los nuevos valores de los unos, dieces y cienes
      temp_unos = registro_corrimiento[11:8];
      temp_dieces = registro_corrimiento[15:12];
      temp_cientos = registro_corrimiento[19:16];
      //Actualizamos el contador
      i=i+1;
      end
  if (i == 9)
      begin
      //El contador llego al limite y lo reiniciamos
      i=0;
      //Por ultimo asignamos las salidas del BCD
      unos = temp_unos;
      dieces = temp_dieces;
      cientos = temp_cientos;
      end
end
endmodule
```

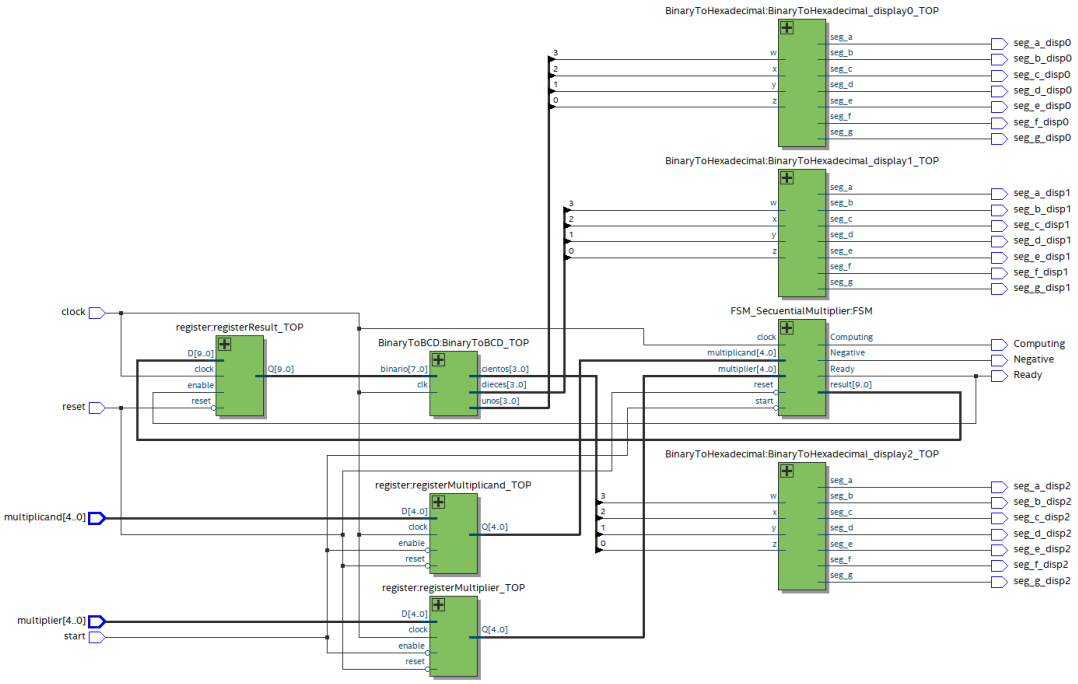
Modulo BinaryToHexadecimal:

```
module BinaryToHexadecimal (
//4 bits de entrada
input w,x,y,z,
//Salidas a los segmentos
output seg_a,seg_b,seg_c,seg_d,seg_e,seg_f,seg_g);

//Logica combinacional para la salida a cada segmento
assign seg_a = ~w&~x&~y&z | ~w&x&~y&~z | w&x&~y&z |w&~x&y&z;
assign seg_b = ~w&x&~y&z | x&y&~z | w&x&~z | w&y&z;
assign seg_c = ~w&~x&y&~z | w&x&~z | w&x&y;
assign seg_d = ~x&~y&z | ~w&x&~y&~z | x&y&z | w&~x&y&~z;
assign seg_e = ~x&~y&z | ~w&z | ~w&x&~y;
assign seg_f = ~w&~x&z | ~w&~x&y | ~w&y&z | w&x&~y&z;
assign seg_g = ~w&~x&~y | ~w&x&y&z | w&x&~y&~z;

endmodule
```

4. Diagrama esquemático que genera la herramienta Quartus Prime (Tools > Netlist Viewers > RTL Viewer) [5%].



5. Reporte de recursos utilizados por el Sequential_Multiplier para N = 8 bits, y su comparación con una versión de un multiplicador modelado directamente con el operador * (*assign Product = Multiplicand * Multiplier*) implementado en LUTs, no en bloques DSP de multiplicación. Comentar los resultados de los recursos utilizados [5%].

Recursos utilizados para el "Sequential_Multiplier" donde N=8:

Flow Status	Successful - Mon Oct 17 11:32:34 2022
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Lite Edition
Revision Name	Sequential_Multiplier
Top-level Entity Name	Sequential_Multiplier
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	167
Total pins	37
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Recursos utilizados para un multiplicador modelado directamente de N=8

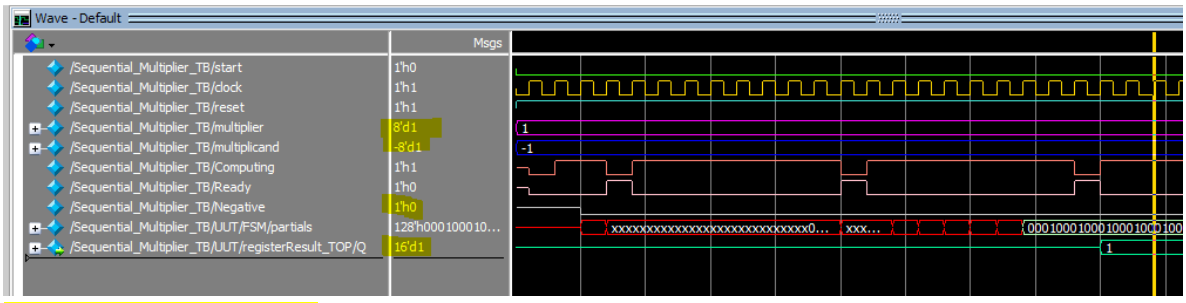
Flow Status	Successful - Mon Oct 17 18:04:09 2022
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Lite Edition
Revision Name	multiplier
Top-level Entity Name	multiplier
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	0
Total pins	8
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	1
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Como previamente se comento en clase, se cuenta con un numero limitado de DSPs para usa sola multiplicación donde N=8 vemos como solo se usa un DSP y en cambio en el "Secuential_Multiplier" vemos como ningún DSP esta siendo utilizado

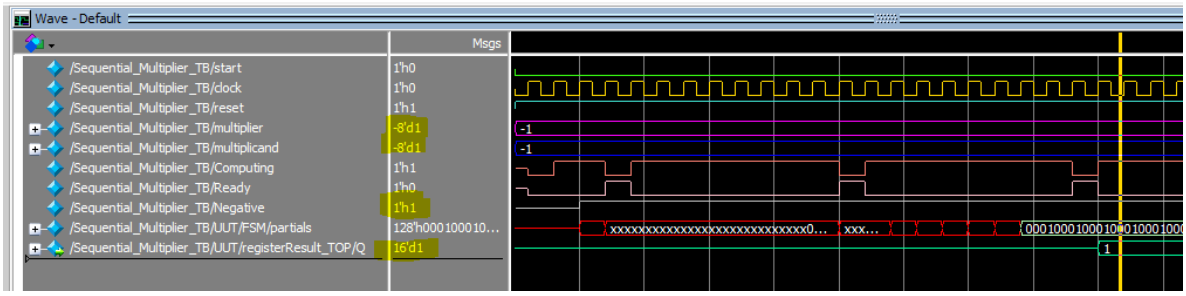
6. El *test-bench* y formas de onda de la simulación del *Sequential_Multiplier*, solamente el *Sequential_Multiplier*, sin display ni decodificador a decimal. Donde se muestren los siguientes casos: 1 x -1, -1 x -1, 15 x -16, 15 x 15, -16 x -16, -16 x 0. Incluye en esta simulación el procesamiento de señales intermedias de interés, por ejemplos, señales de control de la FSM, valores en los registros de corrimiento y productos intermedios en el acumulador [25%].

   Tomar en cuenta que la bandera de Negativo está en lógica negativa para poder encender el signo de menos en el display de 7 segmentos, además de los botones de start y reset que también cuentan con lógica negativa.

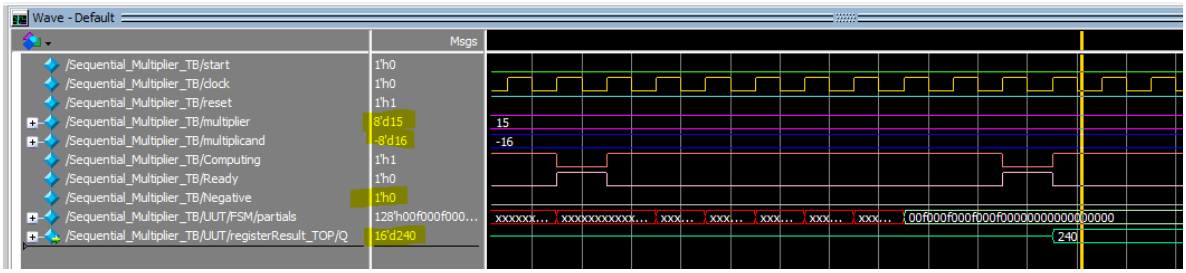
   Resultados para 1x-1:



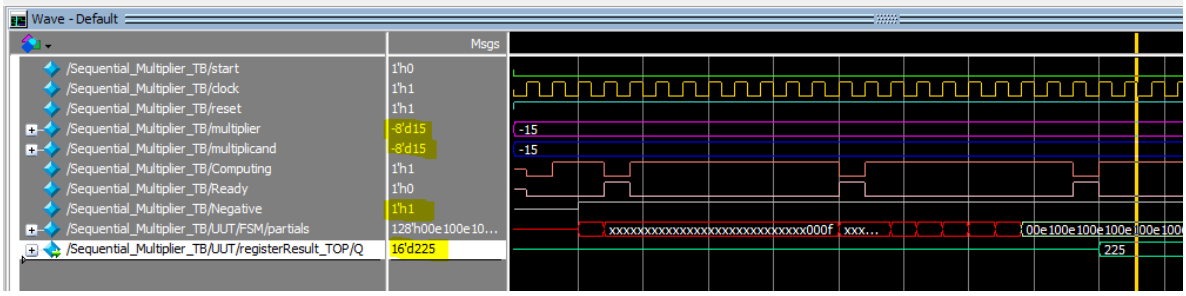
   Resultados para -1x-1:



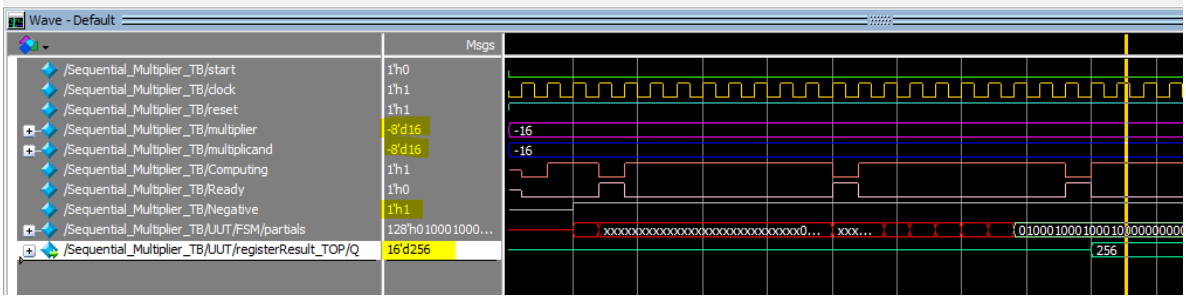
Resultados para 15x-16:



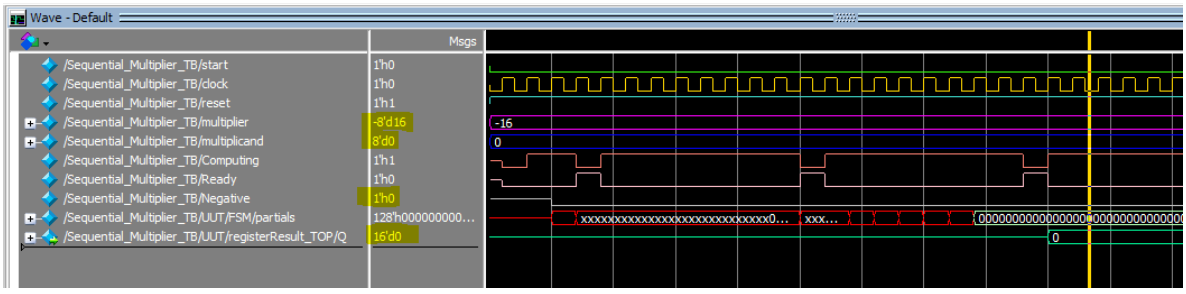
Resultados para 15x15:



Resultados para -16x-16:



Resultados para -16x0:



7. Conclusiones y reflexiones sobre los aprendizajes [10%].

Al finalizar la practica unos de los principales a puntos a resaltar es el continuo aprendizaje en el lenguaje de modelado de HW Verilog y el desarrollo de los testbench

Otro punto muy importante a resaltar el concepto de FSM Maquinas de estados finitas, en esta práctica se introdujo este concepto al desarrollar una máquina de estados que estaría controlando el funcionamiento de nuestro multiplicador secuencial

Y finalmente comenzamos a aplicar optimización de código, el ejemplo esta en evitar usar el multiplicador que es una de las operaciones que mas demanda recursos e implementar en su lugar adders y corrimientos, el hecho si este es la mejor implementación que se puede lograr queda sujeta a la intención o finalidad de la aplicación que se busca desarrollar.

8. Referencias consultadas en formato estándar utilizando Zotero [5%].

[1] Stack Exchange. (2013, Agosto 28). Why are inferred latches bad?. [En linea]. Disponible en: https://electronics.stackexchange.com/questions/38645/why-are-inferred-latches-bad

[2] StackOverflow. (2014, Marzo 17). What is inferred latch and how it is created when it is missing else statement in if condition. Can anybody explain briefly?. [En linea]. Disponible en: https://stackoverflow.com/questions/22459413/what-is-inferred-latch-and-how-it-is-created-when-it-is-missing-else-statement-i#:~:text=A%20latch%20is%20inferred%20within,sensitivity%20list%20and%20feedback%20loops.

[3] StackOverflow. (2016, Abril 19). conditionally calling a module using case statement. [En linea]. Disponible en: https://stackoverflow.com/questions/36661085/conditionally-calling-a-module-using-case-statement

[4] ITESO Canvas. Finite State Machines. [En linea]. Disponible en: https://iteso.instructure.com/courses/25896/files/4377462?module_item_id=1042875

Presentación mostrando la implementación en la tarjeta DE10-Standard y un enlace a un video en algún *drive* donde se muestre el funcionamiento de la implementación para los casos indicados en el punto 6. Sin presentación no se toma en cuenta el reporte.

Sequential Multiplier