



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicaciones

UNIVERSIDAD POLITÉCNICA DE VALENCIA

MEMORIA LABORATORIO PROYECTO 2

DISEÑO CONTROLADOR DE VIDEO

Autores:

DOMÍNGUEZ MARTÍNEZ, Carlos

JIMÉNEZ BOU, Óscar

Telecommunications Engineering,

April 2, 2023

Índice

1. Señales de sincronismo

1.1.Contador parametrizable

1.2.Sincronización con la pantalla

2. Barras de colores en pantalla

3. Imagen en pantalla

3.1.Instanciación y direccionamiento

3.2.Uso de la Memoria ROM

4. Caracteres en pantalla

4.1.Diseño del programa principal

4.2.Direccionamiento y tamaño en pantalla

4.3.Selector de color

5. Texto en pantalla

5.1.Instanciación y direccionamiento

5.2.Uso del registro

5.3.Uso de las Memorias ROM

6. Comprobación de los códigos

6.1.Explicación general de un *Testbench*

6.2.Casos particulares

1. Señales de sincronismo

1.1. Contador parametrizable

El primer paso para desarrollar el diseño del sistema de visualización de la pantalla es desarrollar un bloque que genere las señales de sincronismo de la pantalla. Es crucial para poder representar imágenes en la pantalla. Debido a que, para representar imágenes en la FPGA, no se utilizan todos los píxeles de la pantalla. El diseño propuesto para este módulo está representado a continuación en la *figura 1*.

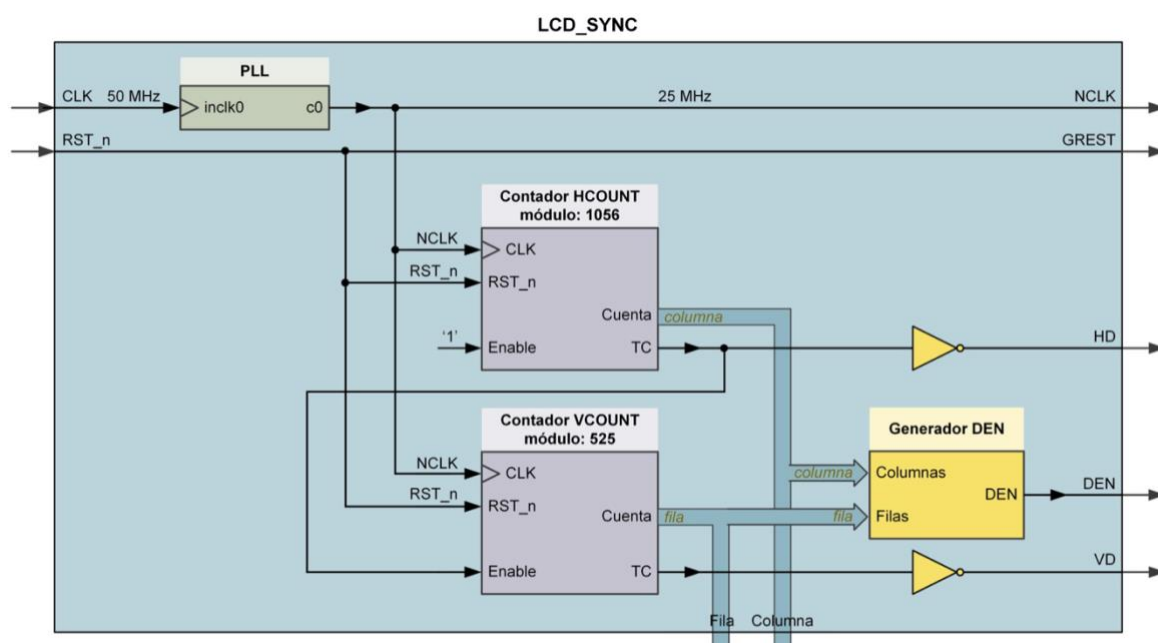


Figura 1. Diagrama LCD_SYNC

El módulo COUNT que se muestra en la *figura 1-2*, se encarga de contar hasta un número específico que elegimos como parámetro, y genera una señal de salida *TC* cuando se alcanza este número.

El módulo cuenta con tres entradas: *CLK*, *RST_n* y *EN*, y dos salidas: cuenta y *TC*.

La salida cuenta es un registro que almacena el valor actual de la cuenta. Por otro lado, el parámetro modulo define el número máximo al que se desea contar. En este caso, el parámetro modulo es igual a 1056, lo que significa que el contador cuenta hasta 1055.

La lógica principal del módulo se encuentra en el *always*. Este bloque se activa en cada flanco de subida de la señal *CLK* y en cada flanco de bajada de la señal *RST_n*. Si la señal *RST_n* está en nivel bajo, se reinicia la cuenta ($\text{cuenta} \leq 0$). Si la señal *EN* está activa, se incrementa la cuenta en uno ($\text{cuenta} \leq \text{cuenta} + 1$). Si la cuenta alcanza el valor máximo (modulo - 1), se reinicia a cero.

La señal TC se calcula mediante una asignación condicional. Si la cuenta alcanza el valor máximo (modulo - 1) y la señal EN está activa, la señal TC se activa. En caso contrario, TC se mantiene en cero.

```

1  module COUNT(CLK, RST_n, EN, cuenta, TC);
2
3  parameter modulo = 1056;
4  parameter n = $clog2(modulo-1);
5
6  input CLK, RST_n, EN;
7  output TC;
8  output reg [n-1:0] cuenta;
9
10 always @(posedge CLK, negedge RST_n) begin
11     if(~RST_n)
12         cuenta <= 0;
13     else if(EN)
14         if(cuenta == modulo - 1)
15             cuenta <= 0;
16         else
17             cuenta <= cuenta + 1;
18     end
19 assign TC = ((cuenta == modulo - 1) && (EN==1)) ? 1:0;
20 endmodule
21
22
23
24

```

Figura 1.1 Módulo COUNT

El último paso, tal y como hemos visto en la figura 1. Vamos a instanciar dos contadores con distinto módulo, el módulo PPL para cambiar la frecuencia del reloj y por último crearemos una secuencia DEN activa cuando este dentro del cuadrante de la pantalla (que hemos especificado).

```

1  module LCD_SYNC (CLK, RST_n, NCLK, GREST, HD, VD, DEN, columna, fila);
2
3  input CLK, RST_n;
4  output HD, VD, GREST, NCLK;
5  output reg DEN;
6  output [10:0] columna;
7  output [9:0] fila;
8
9  wire tc1, tc2;
10
11 pll_ltm pll_ltm_inst( .inclk0(CLK), .c0(NCLK)); //Modulo PPL
12
13 COUNT HCOUNT(NCLK, RST_n, 1, columna, tc1);
14 defparam HCOUNT.modulo = 1056;
15
16 COUNT VCOUNT(NCLK, RST_n, tc1, fila, tc2);
17 defparam VCOUNT.modulo = 525;
18
19 always @(columna or fila) begin
20     if ((columna > 513 && columna < 1016) && (fila > 34 && fila < 217))
21         DEN <= 1;
22     else
23         DEN <= 0;
24 end
25
26 assign HD = ~tc1;
27 assign VD = ~tc2;
28 assign GREST = RST_n;
29 //assign NCLK = CLK;
30 //
31 endmodule
32
33
34

```

Figura 1.2 Módulo LCD_SYNC

Para la creación del módulo, le hemos definido con las entradas CLK, y RST_n, que es la señal de reinicio. Las salidas son HD, VD y GREST, NCLK que es la señal de reloj de salida y DEN que es la señal de habilitación de datos.

Además, hay dos salidas más, *columna* y *fila*, que son las coordenadas de la posición actual de la pantalla LCD. El módulo también utiliza dos cables, *tc1* y *tc2*, para ayudar con la sincronización de la pantalla.

En el código, hay dos módulos COUNT que se utilizan para contar los ciclos de reloj y generar las señales de sincronización para la pantalla LCD. El primer módulo COUNT se llama HCOUNT y se utiliza para contar las columnas. El segundo módulo COUNT se llama VCOUNT y se utiliza para contar las filas.

Los contadores están configurados con los valores de modulo adecuados para la resolución de pantalla deseada. En este caso, el valor de modulo para HCOUNT es 1056 y el valor de modulo para VCOUNT es 525.

Dentro del bloque "always", se utiliza una comparación de coordenadas para determinar cuándo habilitar la señal DEN. Si las coordenadas están dentro de un rango específico, DEN se establece en 1. De lo contrario, se establece en 0.

Finalmente, se utilizan varias asignaciones para generar las señales de sincronización para la pantalla LCD. HD y VD se generan a partir de las señales *tc1* y *tc2*, respectivamente, mientras que GREST simplemente refleja la señal de reinicio. La señal NCLK se genera a partir de la señal de reloj de entrada, CLK.

El test bench que se presenta es para verificar el correcto funcionamiento del módulo LCD_SYNC. En este caso, se definen una serie de señales de entrada (CLK, RST_n) y de salida (HD, VD, GREST, NCLK, DEN, *columna* y *fila*) que se conectan al módulo que se quiere verificar (*t1* en este caso).

La simulación se realiza mediante un bucle *always* que se encarga de generar los flancos del reloj (CLK) y mediante una sentencia *initial* que establece los valores de entrada del módulo (CLK y RST_n) y espera un tiempo determinado antes de liberar el *reset*.

En este caso, se define una condición en el bucle *always* que verifica si la señal de fila ha alcanzado un valor de 456, momento en el que se detiene la simulación mediante la instrucción \$stop. De esta forma, se evita que la simulación se ejecute indefinidamente.

En resumen, el test bench se utiliza para verificar el correcto funcionamiento del módulo LCD_SYNC mediante la simulación de las señales de entrada y salida y la generación de flancos de reloj. Además, se define una condición de finalización de la simulación para evitar su ejecución indefinida.

2. Barras de colores en pantalla

Después de diseñar módulo de generación de las líneas de control LCD_SYNC. Ahora vamos a verificar su funcionamiento. Para ello, se debe asignar un valor a las señales correspondientes de los colores (RGB) y mostramos barras de colores en la pantalla. El esquema de bloques para realizar esto se muestra en la Figura inferior. Cabe destacar que las barras de colores tienen todas la misma anchura y el valor de las señales RGB depende del valor de la columna generado por el bloque LCD_SYNC.

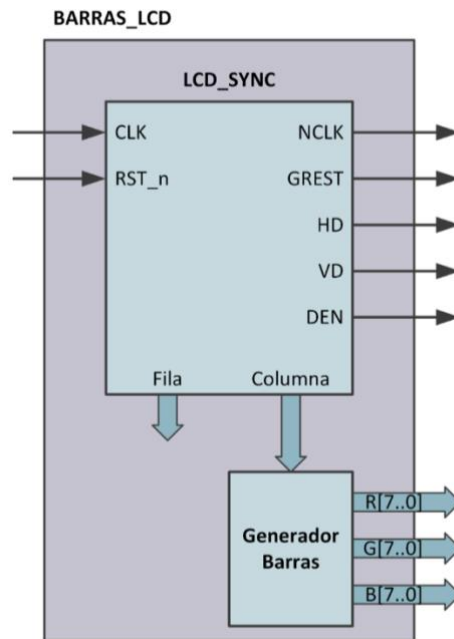


Figura 2. Diagrama BARRAS_LCD

En este apartado, vamos a crear un módulo llamado *BARRAS_LCD* que controla la salida de color a una pantalla LCD.



Figura 2.1 Barras en Pantalla

En este código vamos a instanciar el módulo creado en el apartado anterior. Con sus entradas y salidas correspondientes. Pero en este caso, diseñaremos un bloque *always* con una estructura de selección múltiple *if-else* para asignar los valores de los canales de color "R", "G" y "B" según el valor de *col*. Cada bloque *if-else* asigna valores de color diferentes en función del rango de valores de *col*. La pantalla estará dividida en 8 colores, los cuales estarán divididos en 100 pixeles/color. El código corresponde con la figura 2.3.

```

1  module BARRAS_LCD(CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
2
3  input  CLK, RST_n;
4  output NCLK,GREST, HD, VD,DEN;
5  output reg [7:0] R,G,B;
6
7  wire [10:0]col;
8
9  LCD_SYNC lcd1(CLK,RST_n, NCLK, GREST,HD,VD,DEN,col,);
10
11  /*
12  800/8colores = 100 pixeles
13
14  area visualizacion = 800x480
15  |
16  colocacion pantalla =>
17  desde [216,35].....[1015,35]
18  hasta [216,514].....[1015,514]-----PANTALLA-----
19  */
20
21  always@(col)begin
22  |
23  |
24  | if( col < 315) begin //blanco
25  |   R <= 8'd255;
26  |   G <= 8'd255;
27  |   B <= 8'd255;
28  |   end
29  |
30  | else if( col < 415)begin //amarillo
31  |   R <= 8'd255;
32  |   G <= 8'd255;
33  |   B <= 8'd0;
34  |   end
35  |
36  | else if( col < 515)begin //azul claro
37  |   R <= 8'd0;
38  |   G <= 8'd255;
39  |   B <= 8'd255;
40  |   end
41  |
42  |
43  | else if( col < 615)begin //verde
44  |   R <= 8'd0;
45  |   G <= 8'd255;
46  |   B <= 8'd0;
47  |   end
48  |
49  | else if( col < 715)begin //rosa
50  |   R <= 8'd255;
51  |   G <= 8'd0;
52  |   B <= 8'd255;
53  |   end
54  |
55  |
56  | else if( col < 815)begin //rojo
57  |   R <= 8'd255;
58  |   G <= 8'd0;
59  |   B <= 8'd0;
60  |   end
61  |
62  |
63  | else if( col < 915)begin //azul
64  |   R <= 8'd0;
65  |   G <= 8'd0;
66  |   B <= 8'd255;
67  |   end
68  |
69  |
70  | else begin //negro
71  |   R <= 8'd0;
72  |   G <= 8'd0;
73  |   B <= 8'd0;
74  |   end
75  |
76  | end
77
78
79
80
81  endmodule
82

```

Figura 2.3 Módulo BARRAS_LCD

3. Imagen en pantalla

El siguiente programa a desarrollar permite al usuario poder visualizar una imagen previamente almacenada, en una pantalla LCD de resolución 400x800p a color.

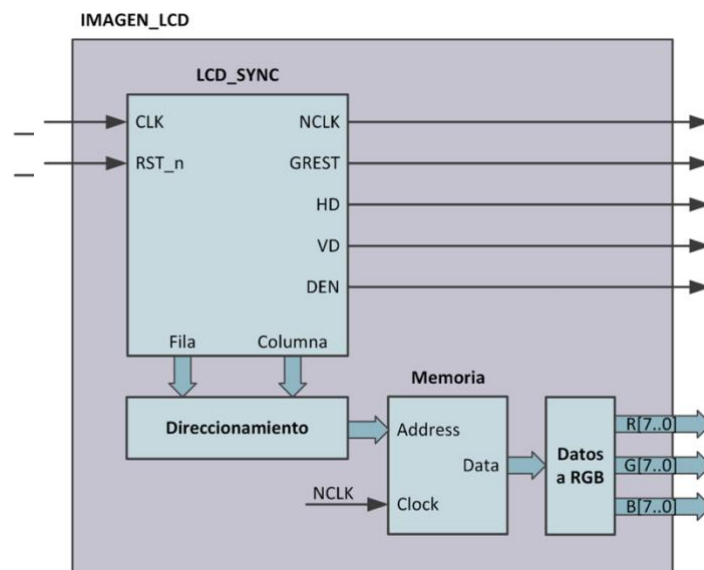


Figura 3. Diagrama IMAGEN_LCD

3.1. Instanciación y Direccionamiento

Para comenzar a diseñar este módulo hemos de tener en cuenta la tasa de refresco de la pantalla y el recorrido por la matriz de píxeles. Es por ello que reutilizamos las señales del programa instanciado “LCD_SYNC”: reloj y reinicio, así como señales de control para la pantalla (*NCLK*, *GREST*, *HD*, *VD* y *DEN*). Pero principalmente nos interesan las señales de filas y columnas (*fil* y *col*) para poder direccionarlas correctamente a la memoria ROM.

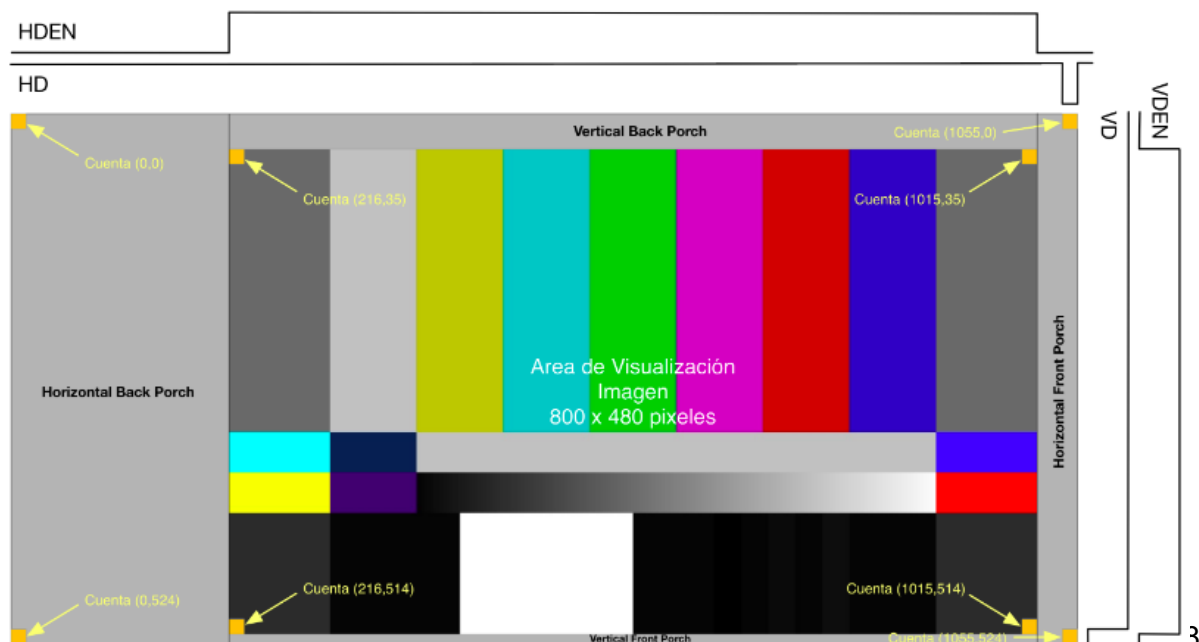


Figura 3.1 Características de la pantalla LCD

Según las especificaciones indicadas por la guía de la pantalla (Figura 3.1), sabemos que el área de visualización de esta empieza en la fila 35 y columna 216. En cuanto al modo de la distribución vectorial de la información, hemos seguido un direccionamiento X-Y de tal manera que cada fila está en un múltiplo de 512. Es por ello que en el código expuesto en la figura 3.2 encontramos el direccionamiento con dichas condiciones dentro del *always* y la asignación del *address*.

```

1  module IMAGEN_LCD(CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
2
3  input  CLK, RST_n;
4  output NCLK,GREST, HD, VD,DEN;
5  output [7:0] R,G,B;
6
7  wire [9:0] fil;
8  wire [10:0] col;
9
10 wire [20:0] address;
11 wire [15:0] data; //Data
12
13 reg [9:0] Y;
14 reg [10:0] X;
15
16 LCD_SYNC lcd2(CLK,RST_n, NCLK, GREST,HD,VD,DEN,col,fil);
17
18 always @(fil, col) // Direccionamiento
19 begin
20     Y = (fil - 35);
21     X = (col - 216);
22 end
23
24 assign address = Y[9:1]*512 + X[10:1];
25
26 ROM_image ROM_image_inst ( .address(address) , .clock ( NCLK ) , .q ( data ) ); //ROM
27
28 assign R = {data[15:11],3'd000};
29 assign G = {data[10:5],2'd00};
30 assign B = {data[4:0],3'd000};
31
32 //♥
33
34 endmodule
35
36
37
38

```

Figura 3.2 Módulo IMAGEN_LCD

3.2. Uso de la Memoria ROM

La imagen se almacena en la memoria ROM como una secuencia de valores hexadecimales, que representan los valores de color de cada píxel de la imagen. Es por ello que el programa usa la dirección *address* (calculada a partir de la fila y columna mencionadas) para leer la imagen de la ROM y luego actualiza los valores de RGB correspondientes a ese píxel en particular.

Esto se consigue habiendo creado previamente una memoria ROM con la herramienta *IP-config*, la instanciamos y le pasamos las variables del direccionamiento ya descritas, asignando una ultima variable *data* que es la que finalmente segmentamos por los tamaños que se nos indica para los valores de RGB.

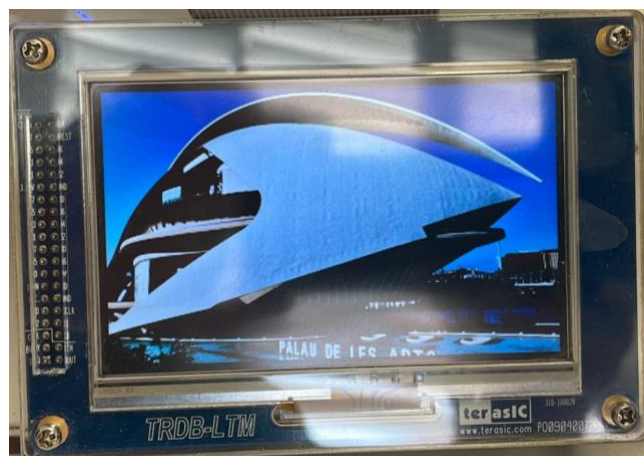


Figura 3.3 Imagen en Pantalla

4. Caracteres en pantalla

En este programa, se desarrollará un módulo permita al usuario visualizar una letra o símbolo elegido previamente en codificación ASCII la misma pantalla LCD. Pudiéndose además elegir el color de la letra y del fondo.

4.1. Diseño del programa principal

En este apartado, vamos explicar el funcionamiento del programa como tal y porqué está dividido en estos módulos representados en la figura 4.

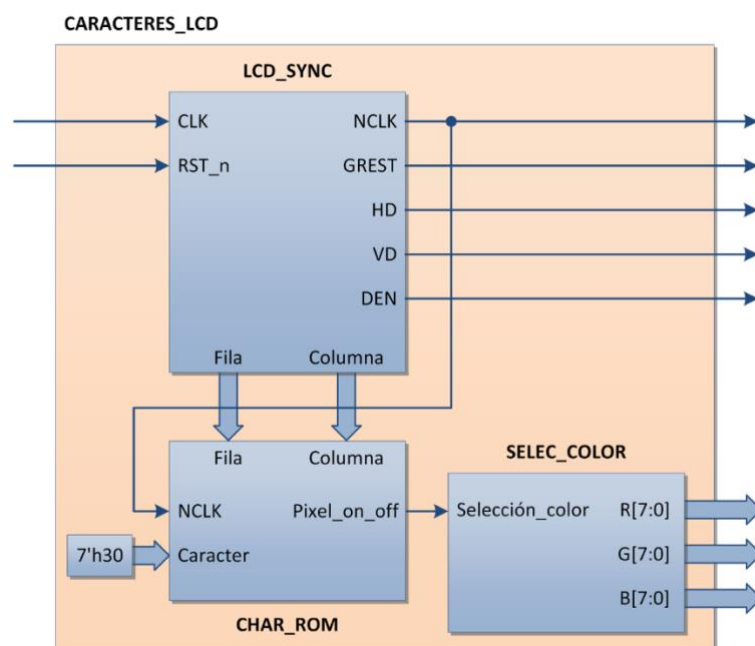


Figura 4. Diagrama CARACTERES_LCD

Como siempre, al estar usando la misma pantalla LCD de los anteriores programas, es imprescindible la instanciación de los módulos de sincronismo como son LCD_SYNC y su respectivo direccionamiento.

Por otro lado, en este caso al estar trabajando con la codificación ASCII necesitamos un “traductor” entre nuestros inputs de caracteres y lo que se va a representar en pantalla, de ahí la funcionalidad del CHAR_ROM.

Y por último hemos de darle la posibilidad al usuario de elegir los colores de su fuente y fondo, mediante el módulo SELEC_COLOR.

4.2. Direccionamiento y tamaño en pantalla

Como ya hemos mencionado, el direccionamiento es muy similar al del programa anterior debido a que estamos usando los mismos componentes y la misma vectorización. Pero con una particularidad, en este caso la señal *address* está definida de manera distinta, ya que en este caso al no tratarse de una imagen con toda la información predeterminada acerca del tamaño y colores de esta, tenemos la posibilidad de elegirla.

El tamaño que tomará el carácter en pantalla puede aumentarse dependiendo cuan lejos del vector 0 comience nuestra concatenación de filas y columnas. Como mostramos en la figura 4.1, en nuestro caso hemos elegido los vectores 5:3 para que la pantalla quede repleta de los caracteres, pero podría escogerse un par de vectores mayor o menor si se desea.

En cuanto al uso de la memoria ROM en este caso la usamos como decodificador entre el carácter que elegimos (*car*) y como se va a representar en la pantalla. Teniendo que tener en cuenta la salida *q*, ya que es la encargada de la selección entre el color del fondo y el carácter en el siguiente módulo.

```
1 module CARACTERES_LCD (CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
2
3 input CLK, RST_n;
4 output NCLK,GREST, HD, VD,DEN;
5 output [7:0] R,G,B;
6
7 wire [9:0] fil;
8 wire [10:0] col;
9 wire [6:0] car; //Caracter
10 wire [12:0] address; // Direccion CHAR_ROM
11 wire q;
12
13 reg [9:0] Y;
14 reg [10:0] X;
15
16 LCD_SYNC lcd2(CLK,RST_n, NCLK, GREST,HD,VD,DEN,col,fil);
17
18
19
20 assign car = 7'h30; //Elige el caracter en ASCII
21
22 //CHAR_ROM
23
24 always @(fil, col) // Direccionamiento
25 begin
26
27     Y = (fil - 35);
28     X = (col - 216);
29
30 end
31
32 assign address = {car,Y[5:3], X[5:3]} ;
33
34
35 ROM_char ROM_char_inst ( .address ( address ), .clock ( NCLK ), .q ( q ) );
36
37 SELEC_COLOR s1(q,R,G,B); //Programa secundario
38
39
40 endmodule
41
```

Figura 4.1 Módulo CARACTERES_LCD

4.3.Selector de Color

El módulo SELEC_COLOR (figura 4.2) recibe una señal de selección sel, que en nuestro código final (figura 4.1) viene determinado por la variable q . Este valor es el que determina la condición del if/else donde se eligen los dos colores que se emiten con tres señales RGB:

- Cuando sel está en bajo (0), el módulo asigna el valor máximo (255) a cada una de las señales de color, lo que resulta en un color blanco.
- Cuando sel está en alto (1), el módulo asigna el valor mínimo (0) a cada una de las señales de color, lo que resulta en un color negro.

**Nota: Estos colores son a libre elección, nosotros hemos usado el blanco y el negro por practicidad visual y de escritura en el código.*

```
1  module SELEC_COLOR (sel,R,G,B);
2
3  input sel;
4  output reg [7:0] R,G,B;
5
6  always @(*)
7  begin
8      if(~sel)
9      begin
10         R = 8'd255;
11         G = 8'd255;
12         B = 8'd255;
13     end
14     else
15     begin
16         R = 8'd0;
17         G = 8'd0;
18         B = 8'd0;
19     end
20 end
21 endmodule
22
```

Figura 4.2 Modulo SELEC_COLOR



Figura 4.3 Caracteres en pantalla

5. Texto en pantalla

Este último módulo permitirá visualizar por pantalla un conjunto de texto codificado en formato hexadecimal, de manera que estos estén centrados en la pantalla y separados por un párrafo, de nuevo pudiéndose elegir los colores del texto y el fondo.

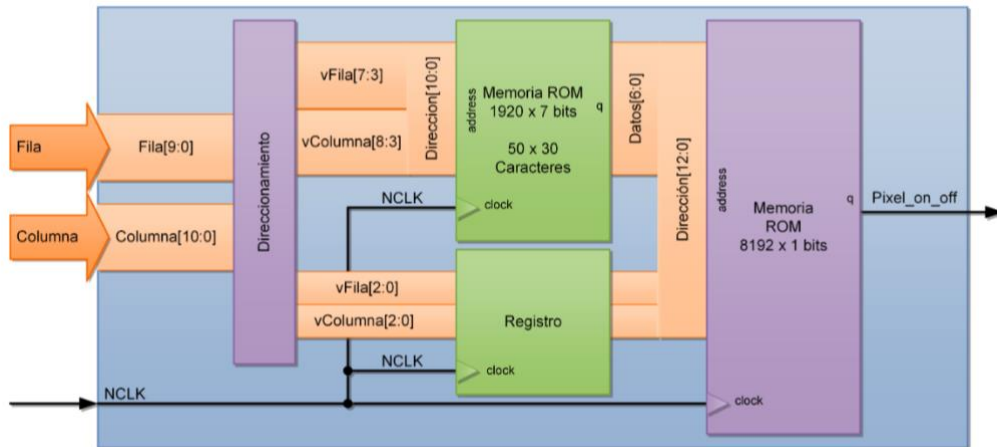


Figura 5. Diagrama FRASE_LCD

5.1.Instanciación y direccionamiento

En este módulo en concreto requiere de dos direccionamientos distintos para que la representación por pantalla sea correcta. Esto es debido a que por un lado tendremos la información a decodificar y por otro lado el decodificador. Es por ello que en la figura 5.1 podemos observar como hemos creado dos direcciones (*dir1*, y *dir2*) además del *address* final.

```

1  module FRASE_LCD (CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
2
3  input CLK, RST_n;
4  output NCLK,GREST, HD, VD,DEN;
5  output [7:0] R,G,B;
6
7
8  wire [9:0] fil;
9  wire [10:0] col;
10 wire [6:0] car;
11 wire [10:0] dir1;
12 reg [5:0] dir2;
13 wire [12:0] address;
14 wire q;
15
16
17 reg [9:0] Y;
18 reg [10:0] X;
19
20
21 LCD_SYNC lcd3(CLK,RST_n, NCLK, GREST,HD,VD,DEN,col,fil);
22
23
24 always @(fil, col) // Direccionamiento
25 begin
26
27     Y = (fil - 35);
28     X = (col - 216);
29
30 end
31
32 assign dir1 = {Y[8:4], X[9:4]} ;
33
34 TEXTO t1 ( .address ( dir1 ), .clock ( NCLK ), .q ( car ) ); //Memoria ROM 1920x7
35
36
37 //Registro
38 always @(posedge NCLK) begin
39
40     dir2 <= {Y[3:1],X[3:1]};
41
42 end
43
44 assign address = {car,dir2};
45
46
47 ROM_char ROM_char_inst1 ( .address ( address ), .clock ( NCLK ), .q ( q ) ); //memoria ROM 8192x1
48
49 SELEC_COLOR s2(q,R,G,B);
50
51
52
53 endmodule
54

```

Figura 5.1 Módulo FRASE_LCD

De esta forma podemos enviar una parte de los vectores a la memoria y otra al registro, y si deseamos aumentar el tamaño de visualización de los caracteres en la pantalla, hemos de asegurarnos que desplazamos ambos conjuntos de vectores en la misma cuantía, o por el contrario la imagen no se verá correctamente.

5.2. Uso del registro

El registro se usa en este programa para poder completar la información necesaria en el decodificador final, ya que parte de los vectores filas y columnas ha de pasar por la memoria que contiene la información de texto. Es por ello que para que se mantenga la información sincronizada hemos de asegurarnos de que funciona con flancos de *NCLK* síncronos (*posedge*), al igual que nuestras memorias ROM.

5.3. Uso de las Memorias ROM

El módulo tiene dos bloques de memoria ROM: una memoria ROM de 1920x7 y otra memoria ROM de 8192x1. La primera se utiliza para almacenar el texto de las frases codificadas en ASCII (hexadecimal) y la segunda para realizar la decodificación de los caracteres correspondientes a cada letra de las frases.

Por último, el módulo utiliza el selector de color mostrado anteriormente en la figura 4.2 para determinar el color del fondo y del texto de las frases instanciado de tal manera que la variable de selección (*sel*) contenga la información correcta del texto y su fondo.

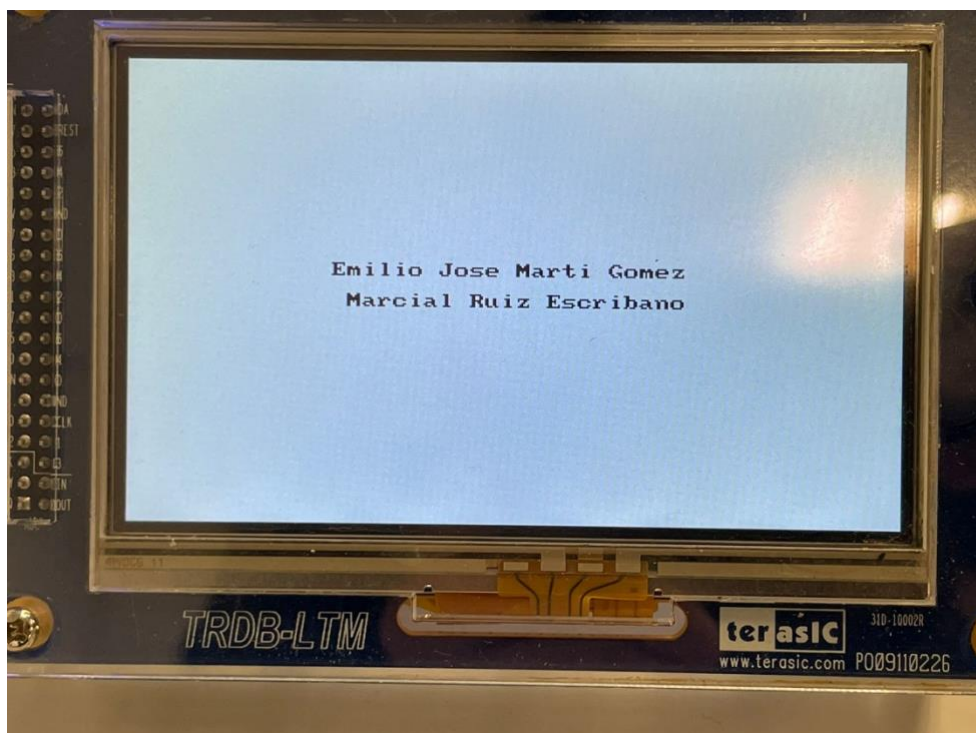


Figura 5.2 Texto en Pantalla

6. Comprobación de los códigos

Durante la creación de un programa completo, normalmente formado e integrado con múltiples módulos. Es frecuente separarlos en módulos distintos, con el fin de verificar el comportamiento y funcionamiento de ese módulo en concreto. Así poder verificar y aislar los errores. Para ello en este apartado, vamos a crear un test bench. Que no es más que un entorno de simulación que permite probar su diseño sin necesidad de implementarlo físicamente en el hardware.

6.1. Explicación general de un *Test Bench*

En primer lugar, vamos a detenernos en ver como se crea realmente un *test bench* en general, para luego ir particularizandolo según los distintos programas a comprobar.

Antes de comenzar con el módulo como tal definimos la escala de tiempo (*timescale 1ns/100ps*) para la simulación. Una vez hecho esto tenemos que definir las entradas y salidas con *wire* o *reg* respectivamente usadas en nuestro código a comprobar. Después de generar las señales para probar el módulo, lo instanciamos.

En todos nuestros programas necesitamos un clock (CLK), por lo que en todos los test bench lo tenemos que definir de la misma manera: mediante un bloque *always*, para simular el cambio según el periodo le hayamos definido.

Más tarde inicializamos las variables de entrada y las modificaremos tras cierto tiempo arbitrario para visualizar su funcionamiento en las salidas ante las combinaciones que sean de nuestro interés. y añadimos un *\$stop* para parar la simulación.

6.2. Casos particulares

A continuación, vamos a ver el módulo *test bench* realizado para el contador en la figura 6. Es un claro ejemplo del *test bench* general comentado anteriormente. Pero hemos definido el *wire* cuenta con 11 bits. Esto es debido a el contador debe llegar hasta el 1056.

```
1  `timescale 1ns/100ps
2
3  module tb_COUNT();
4
5
6  localparam T = 20;
7
8  reg CLK, RST_n, EN;
9  wire TC;
10 wire [11-1:0] cuenta;
11
12     COUNT c1(CLK, RST_n, EN, cuenta, TC);
13
14     always
15     begin
16         #(T/2) CLK = ~CLK;
17     end
18
19     initial
20     begin
21         CLK = 0;
22         RST_n = 0;
23         EN = 1;
24         #(T*10)
25         RST_n = 1;
26         #(T*2000)
27         $stop;
28     end
29
30 endmodule
31
```

Figura 6. Código tb_COUNT

En la figura 6.1 podemos comprobar como el contador llega a su máximo, activa la señal TC y se reinicia .

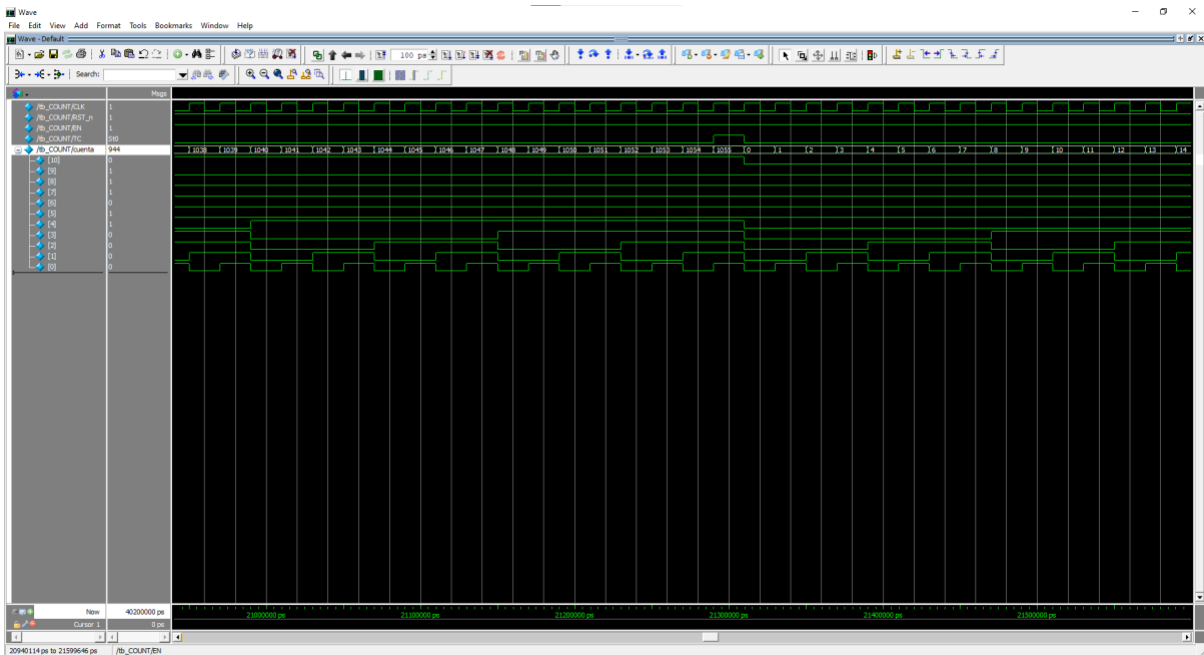


Figura 6.1 Modelsim tb_COUNT

Para el test bench del módulo *LCD_SYNC* (figura 6.2) hemos definido un caso de cuando el programa ya haya recorrido todas las filas de la pantalla se pare. De tal forma, que cuando la componente fila sea igual a 456, la simulación termine.

```

1  `timescale 1ns/100ps
2
3  module tb_LCD_SYNC();
4
5
6      localparam T = 20;
7
8      reg CLK, RST_n;
9      wire HD,VD,GREST,NCLK;
10     wire DEN;
11     wire [10:0] columna;
12     wire [9:0] fila;
13
14     LCD_SYNC t1(CLK,RST_n, NCLK, GREST,HD,VD,DEN,columna,fila);
15
16     always
17     begin
18         #(T/2) CLK = ~CLK;
19     end
20
21     initial
22     begin
23         CLK = 0;
24         RST_n = 0;
25         #(T*2)
26         RST_n = 1;
27     end
28
29     always@(*)begin
30         if(fila==456)
31             $stop;
32         end
33     endmodule
34
35
36
37
38
39
40
41
42

```

Figura 6.2 Código tb_LCD_SYNC

En la simulación de Modelsim (figura 6.3) podemos comprobar su correcto funcionamiento, de manera similar al anterior. Ya que las señales de sincronismo de pantalla se activan una vez las filas y columnas llegan a su valor máximo.

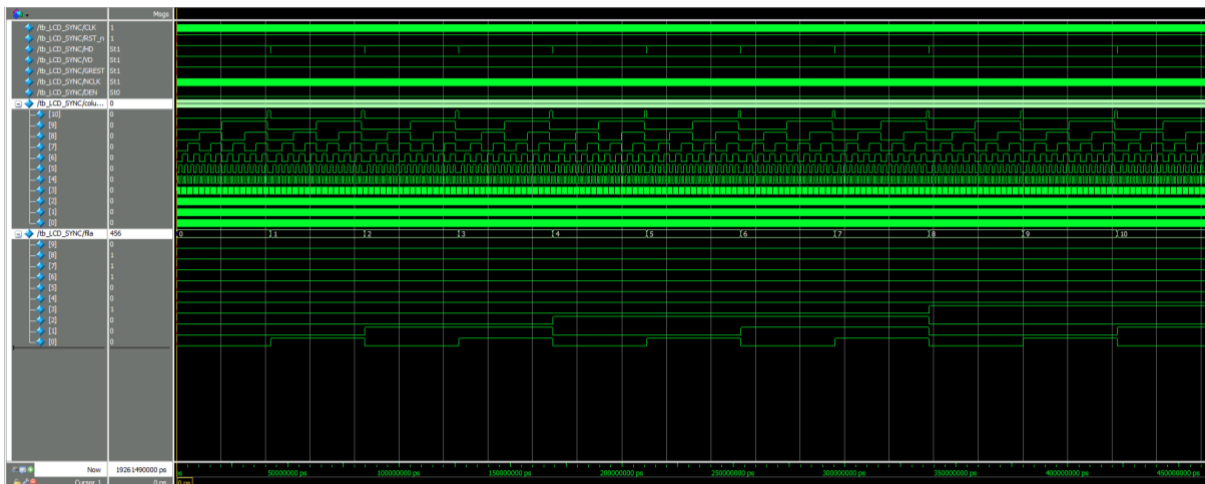


Figura 6.3 Modelsim tb_LCD_SYNC


En el siguiente código (figura 6.4), hemos realizado el test bench de BARRAS_LCD como en los anteriores apartados hemos seguido los mismos pasos y, tal y como nos indicaba las instrucciones de la práctica, hemos introducido unas líneas de código que nos crean un archivo.txt con la codificación de colores en 24 bits.

```

1  `timescale 1ns/100ps
2
3  module tb_BARRAS_LCD();
4
5      localparam T = 20;
6
7      reg CLK, RST_n;
8      wire [7:0] R,G,B;
9      wire HD,VD,GREST,NCLK;
10     wire DEN;
11
12     BARRAS_LCD b1(CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
13
14     always
15     begin
16         #(T/2) CLK = ~CLK;
17     end
18
19     integer fd;
20     event cierraFichero;
21
22     initial begin
23         fd = $fopen ("tb_BARRAS.txt", "w");
24         @(cierraFichero);
25         disable guardaFichero;
26         $display("Cierro Fichero");
27         $fclose(fd);
28     end
29
30     initial begin
31         CLK = 0;
32         RST_n = 0;
33         reset();
34         @(posedge VD)
35         -> cierraFichero;
36         #10;
37         $stop;
38     end
39
40     initial forever begin: guardaFichero
41         @(posedge NCLK)
42         $fwrite(fd,"%t ps: %b %b %b %b %b %b\n", $time,HD,VD,DEN,R,G,B);
43     end
44
45     task reset;
46     begin
47         @(negedge CLK);
48         RST_n = 0;
49         repeat(2) @(negedge CLK);
50         RST_n = 1;
51     end
52 endtask
53
54 endmodule










```

Figura 6.4 Código tb_BARRAS_LCD



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA


poli[format]



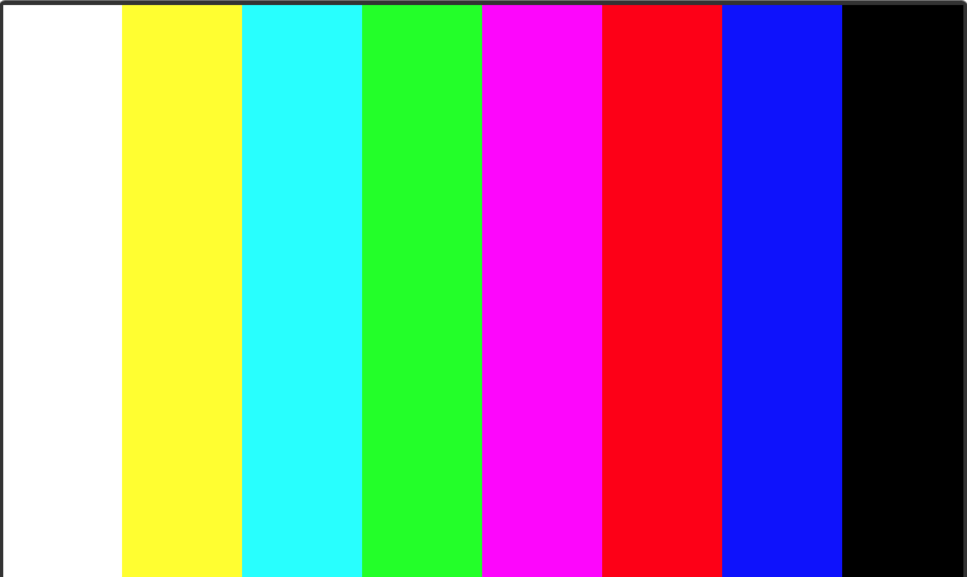
Submit Stop

Working...
554401 / 554401

Click on frames to review:

0:


[Download current frame](#)



Además de esto, también obtuvimos una correcta simulacion con modelSim (figura 6.5) donde los colores van cambiando según la posicion de la fila y columna.

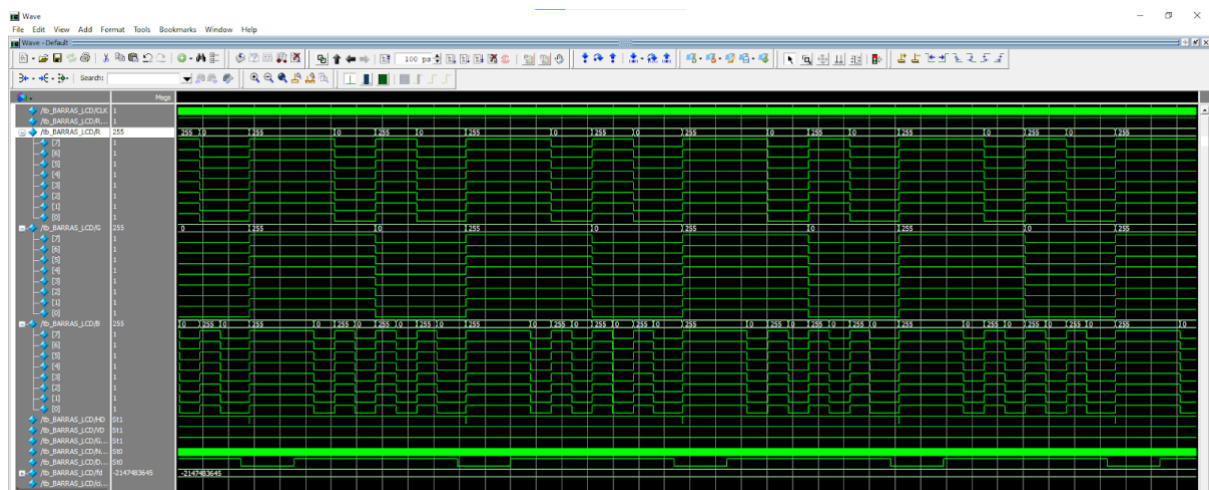


Figura 6.4 Emulación tb_BARRAS_LCD

Los últimos tres apartados (figuras 6.5-7), los hemos definido aparte debido a que en estos casos los test bench son idénticos salvo por la instanciación.

```

1 timescale 1ns/100ps
2
3 module tb_IMAGEN_LCD();
4 localparam T = 20;
5
6 reg CLK, RST_n;
7 wire NCLK, GREST, HD, VD, DEN;
8 wire [7:0] R, G, B;
9
10 IMAGEN_LCD i1(CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
11
12 always
13 begin
14 #T/2 CLK = ~CLK;
15 end
16
17 integer fd;
18 event cierraFichero;
19
20 initial begin
21 fd = $fopen("tb_IMAGE.txt", "w");
22 @(cierraFichero);
23 disable guardaFichero;
24 $display("Cierro Fichero");
25 $fclose(fd);
26 end
27
28 initial begin
29 CLK = 0;
30 RST_n = 0;
31 reset();
32 @(posedge VD)
33 -> cierraFichero;
34 #10;
35 $stop;
36 end
37
38 initial forever begin: guardaFichero
39 @(posedge NCLK)
40 $fwrite(fd,"%t ps: %b %b %b %b %b %b\n", $time,HD,VD,DEN,R,G,B);
41 end
42
43 task reset;
44 begin
45 @(negedge CLK);
46 RST_n = 0;
47 repeat(2) @(negedge CLK);
48 RST_n = 1;
49 end
50 endtask
51
52 endmodule
53

```

Figura 6.5 Código tb_IMAGEN_LCD

```

1 timescale 1ns/100ps
2
3 module tb_CARACTERES_LCD();
4 localparam T = 20;
5
6 reg CLK, RST_n;
7 wire NCLK, GREST, HD, VD, DEN;
8 wire [7:0] R, G, B;
9
10 CARACTERES_LCD c1(CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
11
12 always
13 begin
14 #T/2 CLK = ~CLK;
15 end
16
17 integer fd;
18 event cierraFichero;
19
20 initial begin
21 fd = $fopen("tb_CARACTERES.txt", "w");
22 @(cierraFichero);
23 disable guardaFichero;
24 $display("Cierro Fichero");
25 $fclose(fd);
26 end
27
28 initial begin
29 CLK = 0;
30 RST_n = 0;
31 reset();
32 @(posedge VD)
33 -> cierraFichero;
34 #10;
35 $stop;
36 end
37
38 initial forever begin: guardaFichero
39 @(posedge NCLK)
40 $fwrite(fd,"%t ps: %b %b %b %b %b %b\n", $time,HD,VD,DEN,R,G,B);
41 end
42
43 task reset;
44 begin
45 @(negedge CLK);
46 RST_n = 0;
47 repeat(2) @(negedge CLK);
48 RST_n = 1;
49 end
50 endtask
51
52 endmodule
53

```

Figura 6.6 Código tb_CARACTERES_LCD

```

1 timescale 1ns/100ps
2
3 module tb_FRASE_LCD();
4 localparam T = 20;
5
6 reg CLK, RST_n;
7 wire NCLK, GREST, HD, VD, DEN;
8 wire [7:0] R, G, B;
9
10 FRASE_LCD f1(CLK,RST_n,NCLK,GREST,HD,VD,DEN,R,G,B);
11
12 always
13 begin
14 #T/2 CLK = ~CLK;
15 end
16
17 integer fd;
18 event cierraFichero;
19
20 initial begin
21 fd = $fopen("tb_FRASE.txt", "w");
22 @(cierraFichero);
23 disable guardaFichero;
24 $display("Cierro Fichero");
25 $fclose(fd);
26 end
27
28 initial begin
29 CLK = 0;
30 RST_n = 0;
31 reset();
32 @(posedge VD)
33 -> cierraFichero;
34 #10;
35 $stop;
36 end
37
38 initial forever begin: guardaFichero
39 @(posedge NCLK)
40 $fwrite(fd,"%t ps: %b %b %b %b %b %b\n", $time,HD,VD,DEN,R,G,B);
41 end
42
43 task reset;
44 begin
45 @(negedge CLK);
46 RST_n = 0;
47 repeat(2) @(negedge CLK);
48 RST_n = 1;
49 end
50 endtask
51
52 endmodule
53

```

Figura 6.7 Código tb_FRASE_LCD

Por último la simulación se ejecuta correctamente pero no permite la visualización de información relevante de los colores pixeles, ya que esta se encuentra en variables internas *wire* y *reg* que no se pueden visualizar de manera directa en el Modelsim (figura 6.8).

Es por ello que no funciona tampoco el simulador de polifomat con los archivos.text que nos genera el código. Este inconveniente podría ser solucionando, modificando el código y separándolo en módulos distintos para poder visualizar correctamente la información contenida en las variables internas..

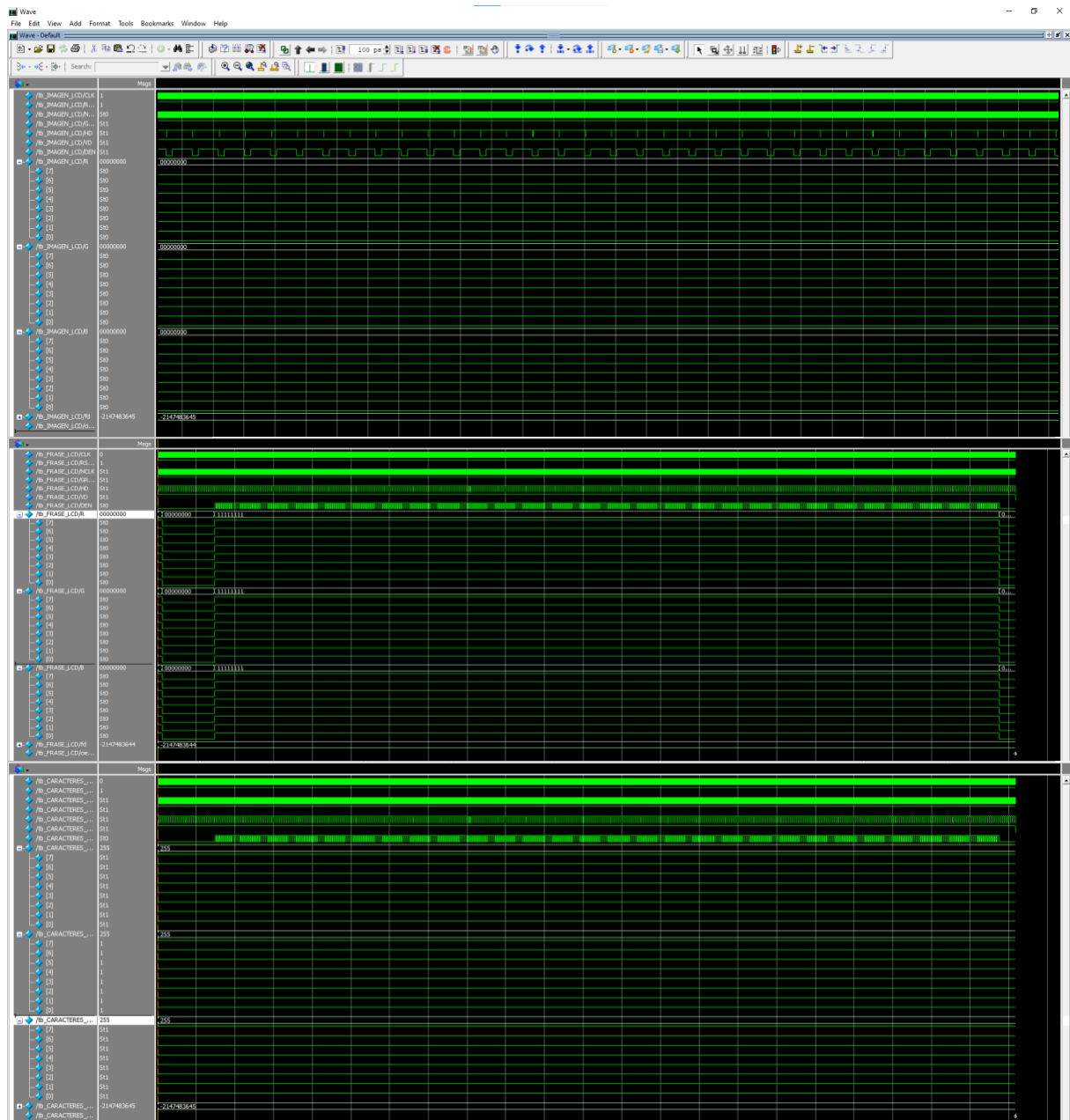


Figura 6.8 Modelsim sin información relevante.