HY430 – Digital Systems Lab

Lab assignment 4

13/1/2020

Βαγενάς Αναστάσης

This report describes the implementation of the 4th laboratory work where it is divided into 2 parts and in each of them we show its implementation in code / logic, the verification of its operation and the results of the experiment on FPGA.

Prelude

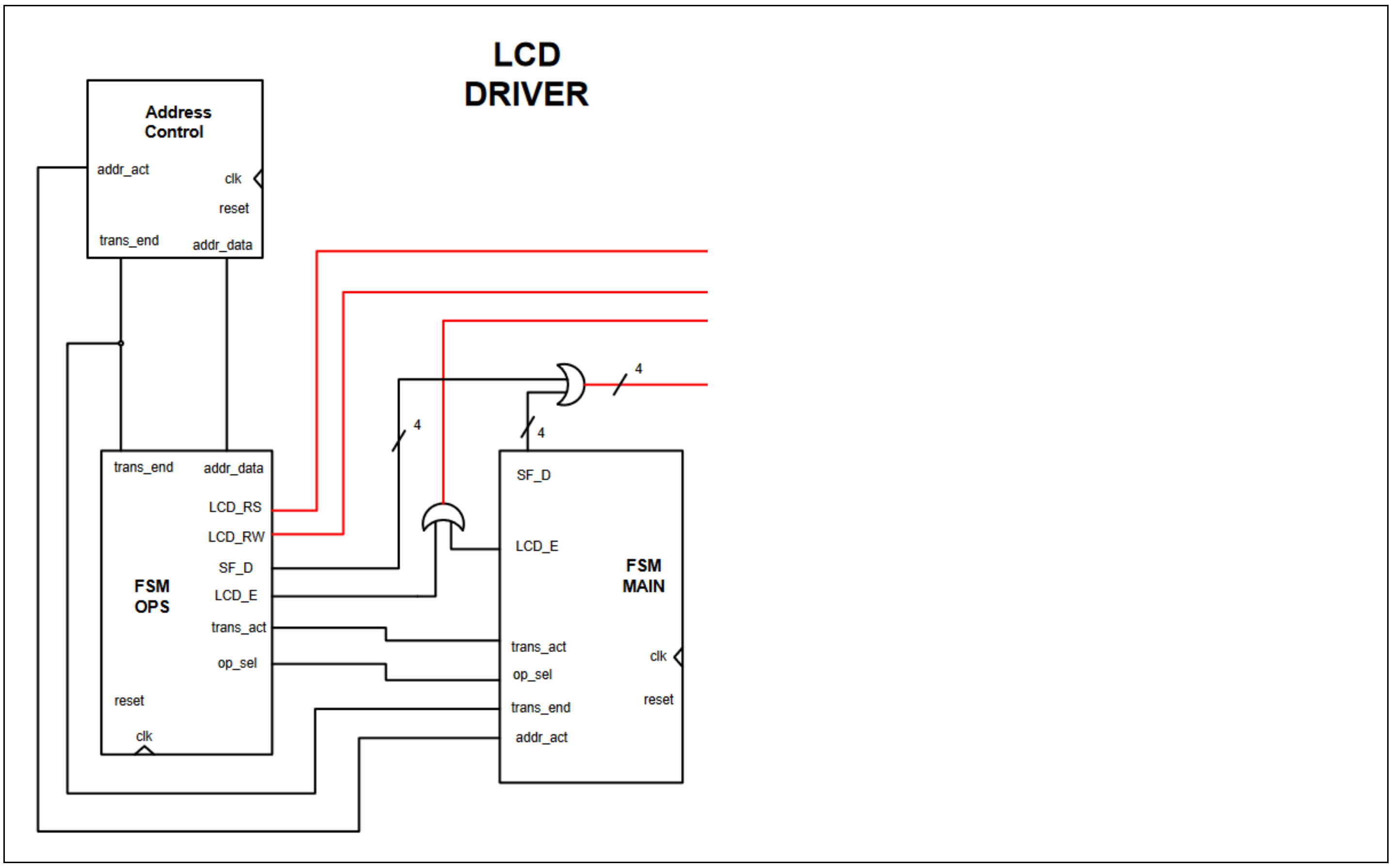
The aim of the laboratory exercise is the implementation of an LCD screen driver, where we will present a static message with a flashing cursor. The message will be stored in the BRAM of FPGA and this will be presented with appropriate implementation of finite state machines.

Par A/B

There is no differentiation in the parts of the work due to the great dependence of the NMS and between them, with the result that for proper documentation the operation of each in parallel must be explained..

Implementation

Below is the top level schematic of the driver (in red the output signals):



As shown for the implementation we use 3 modules each with a specific function:

- The main fsm unit, which is the central machine that manages all the control signals for the other units and operates in all phases (Initialization, Configuration, Data Sending) .
- The fsm ops unit, which is the one that sends the commands to the LCD screen and takes action immediately after the screen is configured. Receives the signal from trans_act and sends the trans end back to the main fsm.
- Finally is the address control unit, which is the unit that manages for which address we will put in the DDRAM of the LCD screen and from which address of BBRAM to read data that we will write on the screen. It accepts the addr_act signal for activation and after that the synchronization for state switching is done with the trans_end signal.

So for the best explanation of the implementation we will start from the simplest module, the address control unit.

Address Control

As mentioned above this unit accepts the control signals addr_act and trans_end (both clk and reset of course). The only output is addr_data which is an 8-bit signal that will contain an address that we assign to DDRAM or the character from BRAM that we want to display.

In addition the module counts the time the cursor will flash on the screen.

So for all this we use a state machine of 3 states, written in the form of two always blocks, one where we manage the alternation of states and one the output of each state.

(We follow this logic for state machines in every module and for this it will be considered a given below)

Below is the code for the state machine:

```
always @ (posedge clk or posedge reset) begin
  if(reset)
    state<=IDLE;
  else begin
     case(state)
        IDLE:if(addr_act) state<=SEND_ADDRESS;
        SEND_ADDRESS:if(trans_end) state<=WRITE_DATA;
        WRITE_DATA:if(trans_end) state<=SEND_ADDRESS;
     endcase
end
```

And the output of the machine:

```
  always @ (state or addr_ddram or char_out) begin
  case(state)
    IDLE:addr_data = 8'bx;
    SEND_ADDRESS:addr_data = addr_ddram;
    WRITE_DATA:addr_data = char_out;
    default:addr_data = 8'bx;
 endcase
```

We have the 3 situations that were mentioned above and in each situation we extract the appropriate data.

Then, we set 3 counters, one of which is a 1-bit register:

1. Both are the pointers for the BRAM and DDRAM of the LCD with the names addr_bram and addr_ddram respectively
2. A counter (refresh_count) that counts 1 second for the cursora to flash on the screen
3. The 1-bit register (cursor_flag) which is a toggle signal with alternation every 1 second and checks the status of the cursora, whether it will be on or off.

<u>Code for refresh_count and cursor_flag:</u>

```
always @ (posedge clk or posedge reset) begin
  if(reset)
    refresh_count<=0;
  else begin
    if(refresh_count == DELAY_1_S || state==IDLE) ///only when maxed out or in idle
       refresh_count<=0;
    else if(trans_end) //The counter changes at the end of every state
       refresh_count<=refresh_count+1;
  end
//////////////////////////////////////////////////
always @ (posedge clk or posedge reset) begin
  if(reset)
    cursor_flag<=0;
  else begin
   if(refresh_count == DELAY_1_S) ///when counter is maxed toggle the value
     cursor_flag<=~cursor_flag;
  end
```

As it seems their operation is quite simple, refresh_count counts until the time DELAY_1_S we have set and then returns to 0. He returns to 0 and starts counting again.

Cursor_flag in turn changes value each time the counter reaches its maximum value.

<u>Addr_bram</u>

For BRAM we must first explain how the message is stored in the figure below:

Starting Address(Dec)	Data (Each cell is 8-bits and contains the character to be displayed)							
0	A	B	C	D	E	F	G	H
8	I	J	K	L	M	N	O	P
16	a	b	c	d	e	f	g	h
24	i	j	k	l	m	n	o	(*blank_char*)
32	(*cursor*)							

So when we want to present the normal format of the message we will count with the pointer addr_bram up to 31 (the blank character), instead we pass it and count up to 32 (the cursor). This is achieved with the signal cursor_flag.

Code for addr_bram:

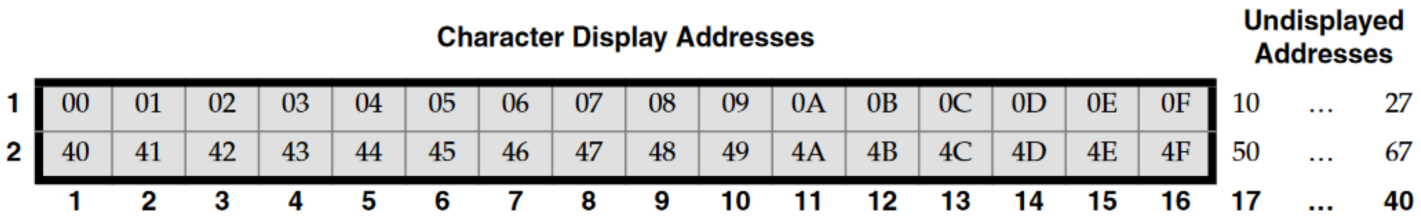
```
always @ (posedge clk or posedge reset) begin
  if(reset)
    addr_bram<=0;
  else begin
   if(state == WRITE_DATA && trans_end && cursor_flag && addr_bram == MAX_BRAM_ADDRESS-1)
     addr_bram<=addr_bram+2;
   else if(state==IDLE || (state == WRITE_DATA && addr_bram >=MAX_BRAM_ADDRESS && trans_end))
      addr_bram<=0;
   else if(state == WRITE_DATA && trans_end)
      addr_bram<=addr_bram+1;
end
```

As shown in the code, the changes in the counter are synchronized with the trans_end signal that changes our situations. Thus, while we are at the end of state WRITE_DATA where we write to DDRAM of the display, we are free to change the address of the BRAM.

Then while the signal cursor_flag is 0 we count up to MAX_BRAM_ADDRESS (31). But if the signal is 1 then when we reach the value 30 of addr_bram, the next value we will give it is 32, ie the position where we have stored the cursor.

<u>Addr_ddram</u>

Respectively in DDRAM to explain the logic we quote the contents of DDRAM from the Xillinx manual:



	Character Display Addresses																Undisplayed Addresses		
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	…	27
2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	…	67
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**…**	**40**

In order to write correctly on the screen in the 1st phase we count to 0F and then in the 2nd phase, we set the address to 40 and count normally again to 4F where we reset the addr_ddram.

The code for addr_ddram counter:

```
always @ (posedge clk or posedge reset) begin
  if(reset)
    addr_ddram<=0;
  else begin
    if(state==IDLE || (addr_ddram == MAX_DDRAM_ADDRESS && trans_end && state == SEND_ADDRESS))
       addr_ddram<=0;
    else if(state == SEND_ADDRESS && trans_end && addr_bram == END_LINE_DDRAM_ADDRESS)
       addr_ddram<=7'h40;
     else if(state == SEND_ADDRESS && trans_end)
       addr_ddram<=addr_ddram+1;
end
```

As shown in the code, changes are made only when we are in the state SEND_ADDRESS and specifically at the end of it (trans_end).

It will be explained below how the trans_end signal is exported, so far let it be considered that it synchronizes the FSMs between them.

If these are valid, if we are at the end of the 1st line the next value will be 0x40. If we are at the end of the 2nd line or in IDLE state then we reset the pointer. Finally, if none of these are valid then we just count up.

BRAM

In addition to the above, in the module we have instatiated a module containing BRAM (Xillinx primitive was used based on the assignment specifications). In the bram_module we give as input the clk and the addr_bram (we always keep the enable of BRAM in 1) and we have as output the char_out (which we saw in the description of the state machine above) .

FSM Ops

This module sends commands to the LCD controller and sets the appropriate values to all the output signals of the LCD driver. Its main part is the FSM which executes the transmission of the command to the controller based on the following figure:

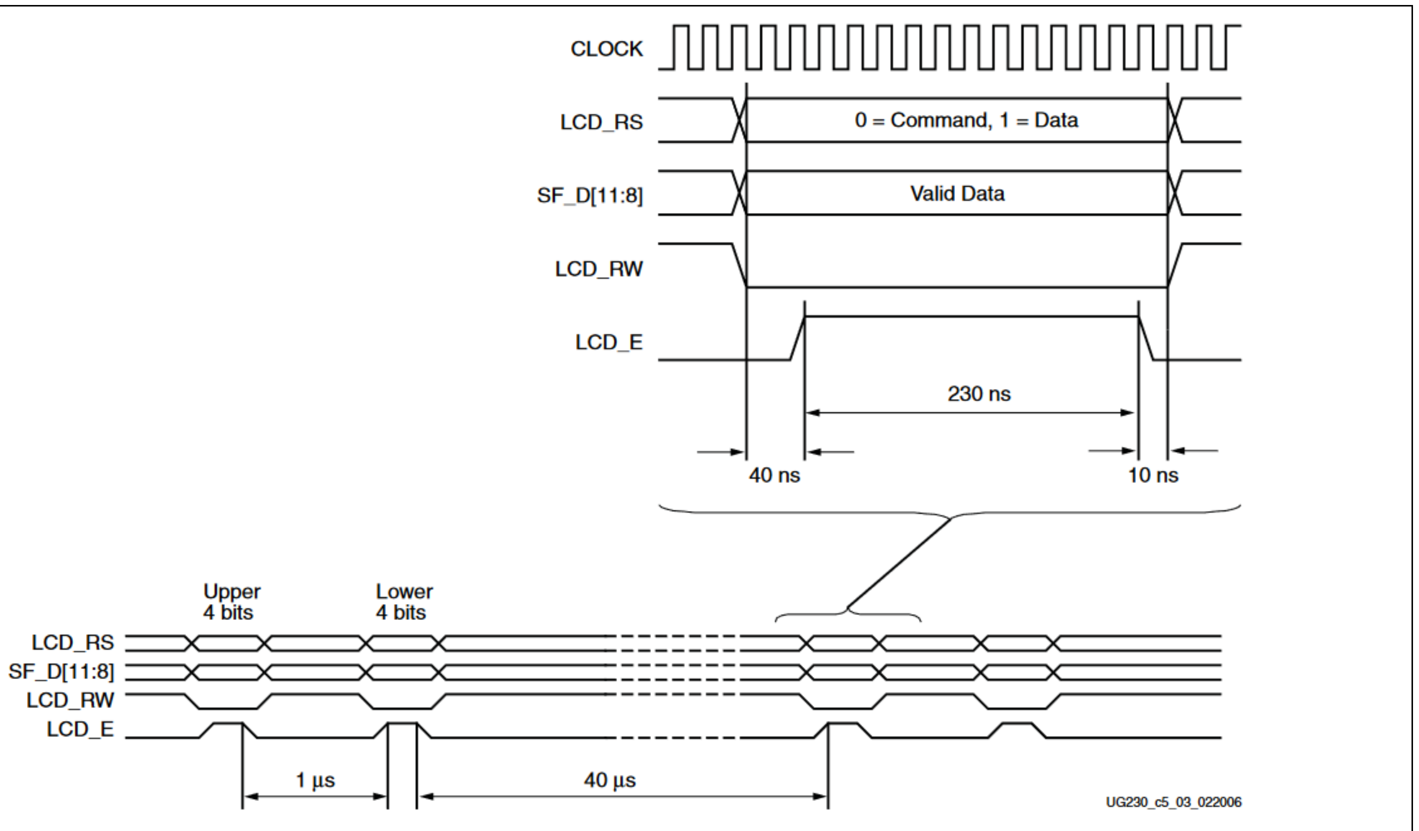


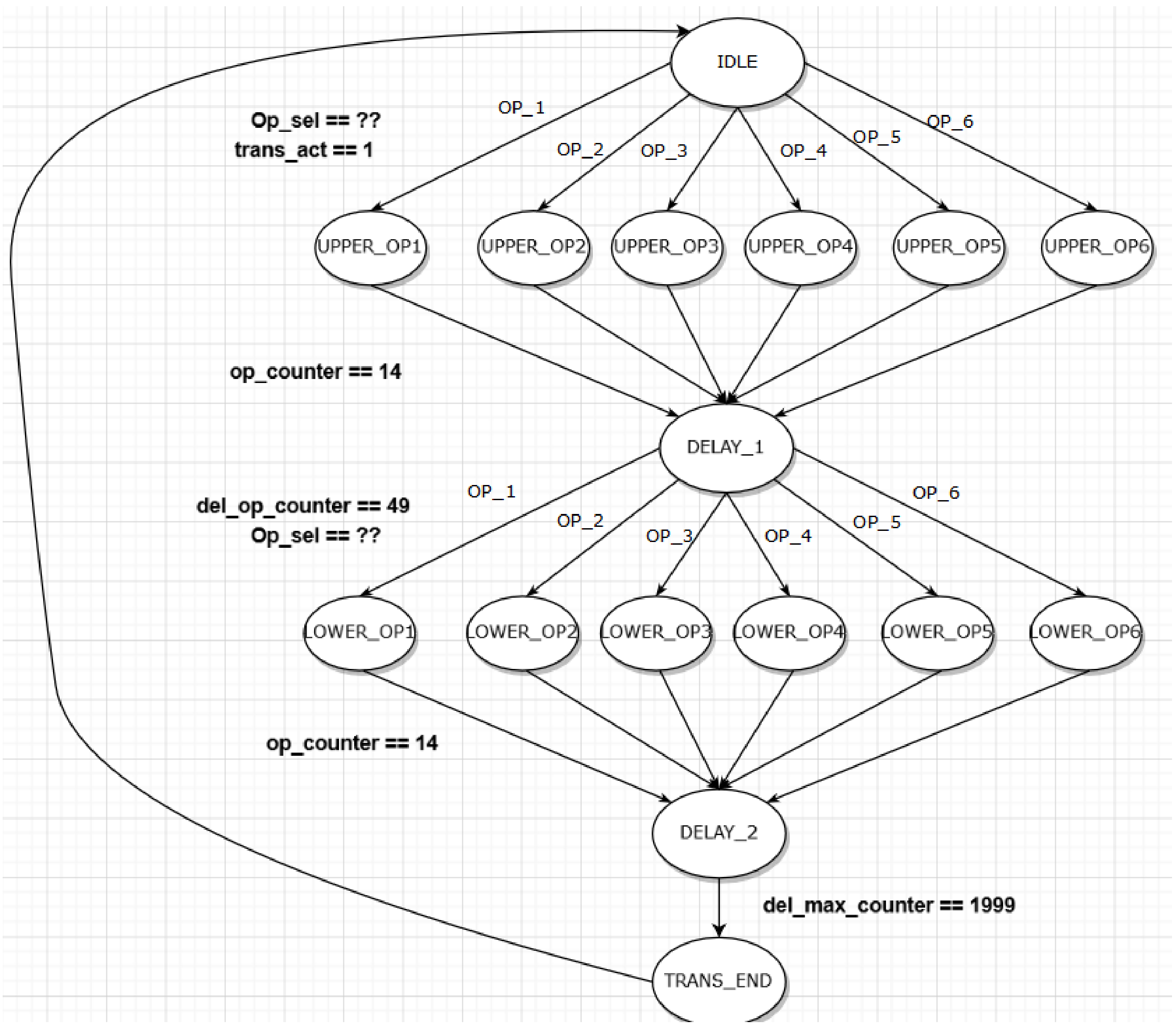
Figure 5-6: **Character LCD Interface Timing**

Since we have a clock with a period of 20ns , the only difference is in its time LCD_E where LCD_E is high for **240ns** and we will set a hold time of **20ns** rather than 10ns.

In addition the module itself will receive the trans_act and op_sel signals from the FSM MAIN module which are activation signals and a signal for which command it will send respectively. In turn, the FSM OP when it has completed the transmission of the command will send the trans_end signal to FSM MAIN (and address control as we saw earlier) and returns to the inactive state.

The module in is only active in configuring and sending data to the LCD screen. In addition, it accepts while it is in the data transmission (ie we write on the screen) the address where it will write in DDRAM and the character that it must display in it from BRAM via addr_data .

A schematic of the FSM:



As shown by the IDLE state we go to a UPPER_OP state when trans_act is 1 and to which OP_ # based on op_sel. So in a UPPER_OP state we send the SF_D [7-4] with the appropriate configuration of LCD_E, LCD_RS (the LCD_RW is always at 0).

After that we have the status DELAY_1 which is the interval of 1μs that we wait until we send the rest of the command. When the time passes based on op_sel again we choose which LOWER_OP mode we will go to. In these situations we send SF_D [3: 0] and the appropriate configuration of LCD_E, LCD_RS.

Then, we go to the state DELAY_2 where after measuring the appropriate time, ie 40μs, we go to the state TRANS_END. We remain in this state only 20ns, a cycle, in which we raise the trans_end to high to inform that the sending of the whole command is completed.

Finally, we return to IDLE state where we wait for a request for a new command to be sent.

Code

Based on the schematic we will use 3 counters of different sizes :

1. The op_counter that will count for the duration of 300ns of LOWEP_OP / UPPER_OP states. Based on these we configure the signals that will control the LCD screen (LCD_E, LCD_RS, SF_D).
2. The del_op_counter that will measure the delay of 1μs between LOWER and UPPER. It is active only in DELAY_1 state.
3. The del_max_counter that will measure the delay of 40μs after UPPER and until TRANS_END. It is active only in DELAY_2 state.

In addition, we have the 5-bit state register which states what state we are in. It is larger than we need (we have 16 states) to be able to make some simplifications in the code to be shown.

Below is the code for the counters:

```
always @(posedge clk or posedge reset) begin
  if(reset==1)
    op_counter<=0;
  else if(op_counter==14 || state==DELAY_1 || state==DELAY_2 ||state==TRANS_END ||state==IDLE) //max value 15 cycles
    op_counter<=0;
  else
    op_counter<=op_counter+1;
end
///….///same code
  else if(del_op_counter==49 || state!=DELAY_1)//max value 50cycles
    del_op_counter<=0;
///…./// same code
  else if(del_max_counter==1999 || state!=DELAY_2)//max value 2000cycles
    del_max_counter<=0;
///…./// same code
```

All counters have the same format with the only difference being the size and conditions in the else if statement.

First, we see op_counter which only works in UPPER and LOWER states and counts 15 cycles or 300ns.

In the same way we have del_op_counter which measures 50 cycles or 1μs and works only in state DELAY_1.

Similarly we have the del_max_counter which measures 2000 cycles or 40μs and works only in state DELAY_2.

Below is the code for the transitions of the states (part of it):

```
always @(posedge clk or posedge reset) begin
  if(reset==1)
    state<=IDLE;
  else begin
            casex(state)
            5'b1xxxx://LOWER OPS STATES
               if(op_counter==14)
                 state<=DELAY_2;
             5'b01xxx://UPPER OPS STATES
               if(op_counter==14)
                  state<=DELAY_1;
              IDLE://Idle state gets the trigers and chooses which op to move on to
                  if(trans_act==1) begin
                   case(op_sel)
                           CLEAR_DISPLAY:state<=UPPER_CLEAR_DISPLAY;
                           ENTRY_MODE_SET:state<=UPPER_ENTRY_MODE_SET;
                           FUNCTION_SET:state<=UPPER_FUNCTION_SET;
                           SET_DDRAM_ADDRESS:state<=UPPER_SET_DDRAM_ADDRESS;
                           WRITE_DATA_TO_DDRAM:state<=UPPER_WRITE_DATA_TO_DDRAM;
                           DISPLAY_ON_OFF:state<=UPPER_DISPLAY_ON_OFF;
                 endcase
            end
///….///  more states
```

In the above part we can see the transitions we described. If we are in IDLE then when we have trans_act at high based on the op_sel we go to some UPPER_OP.

Then, when we are in a UPPER state then when op_counter is set to 14 then we go to DELAY_1 state, which is done in any UPPER command line and we are.

To achieve this we use casex and we have put appropriate coding in the situations so we can merge them all into one option of casex (specifically all in "01xxx").

We act the same for the LOWER states where we have all merged them in "1xxxx".

Hence the larger size of the state register, we used 1 extra bit to be able to simplify the transition logic and also reduce the volume of code.

Below is the code for the output of FSM (part of it):

```
//State Machine outputs for every state

always @(state or addr_data) begin

  case(state)

              IDLE: SF_D = 4'b0000;

              ..........

               UPPER_DISPLAY_ON_OFF:SF_D = 4'b0000;

               UPPER_SET_DDRAM_ADDRESS:SF_D = {1'b1,addr_data[6:4]};

              ..........

               TRANS_END: SF_D = 4'b0000;

///…./// more states
```

As shown in the output we set the appropriate value in SF_D based on what situation we are in. What value we set depends on the table given to us by the Xillinx manual, so we quote a few indicative values in the code.

Below is the code for the other output of the module :

```
//** Read/Write signal always high except when we are sending the upper or lower op command

assign LCD_RW = 1'b0;

//** Register Select signal ,only high during sending upper/lower data to DDRAM

assign LCD_RS = (op_counter <15 && ( state==LOWER_WRITE_DATA_TO_DDRAM ||
state==UPPER_WRITE_DATA_TO_DDRAM))?1'b1:1'b0;

//** LCD Enable signal,active for a specific time

assign LCD_E = (op_counter>1 && op_counter <14)?1'b1:1'b0;

//Handshake signal for the main FSM

//When high the Mom Fsm knows we are done

assign trans_end = (state==TRANS_END)?1'b1:1'b0;
```

In the last piece of code we see the assignment of the on-screen control signals and the trans_end signal.

LCD_RW for work needs always remains at 0 and LCD_E remains active only when it is in the state where op_counter has the specified values.

LCD_RS for its part is only active when we need to send the WRITE_DATA_TO_DDRAM command for this and it has these limitations.

Finally trans_end remains high only when we are in state TRANS_END.

FSM Main

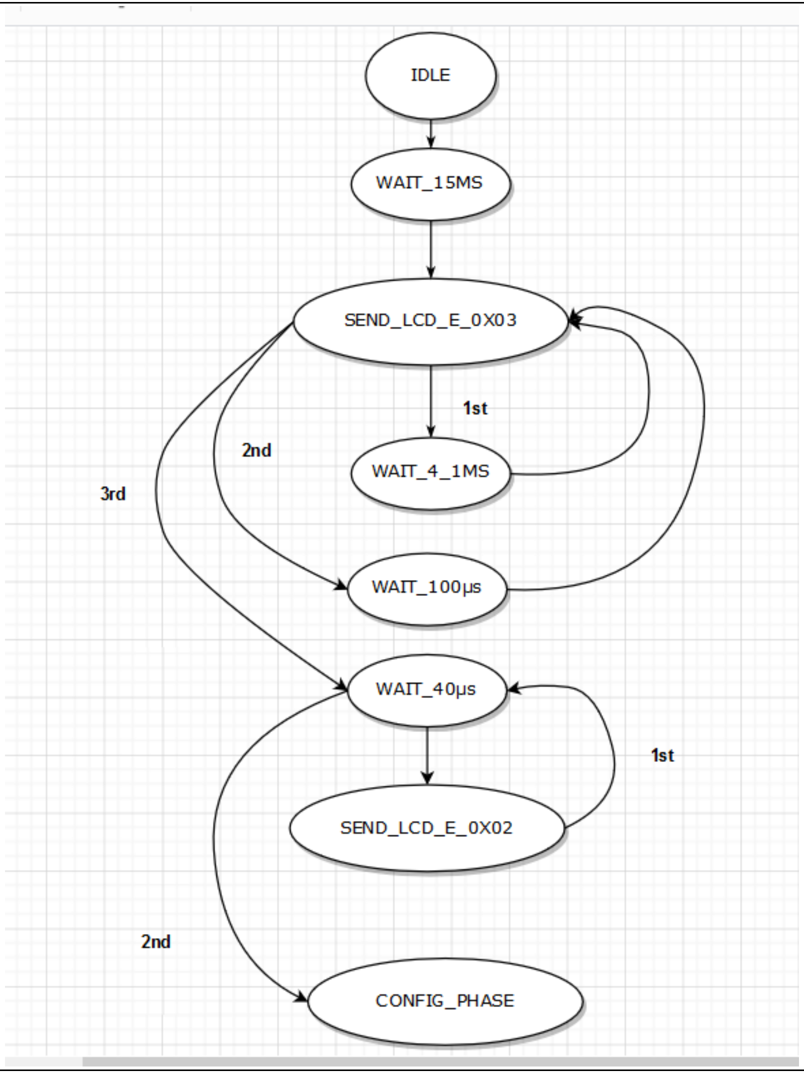
This module is the main controller of the LCD Driver operation and has the full load to initialize the screen and then send the appropriate commands (using FSM OPS) in the correct order (in Configuration and data transmission).

Initially, we 'divide' the FSM into 3 types of situations:

- The states it goes through when it is in the initialization state (init phase).
- The states it goes through when it is in the config phase.

- The situations it goes through when it is in data sending mode, ie displaying characters on the screen.

For an easier explanation, right below a figure with the initialization phase:



As shown, we follow the instructions for initialization and reuse situations. To do this we have 2 register flags, state_op_flag and state_wait_flag.

The first is to distinguish the next state after SEND_LCD_E_Ox03 and has a size of 2 bits (3 different cases). The second is to distinguish the next state after WAIT_40μs.

In addition, in the same spirit as before we have 3 counters that count times:

1. The b_delay_counter that measures times of the order of ms (WAIT_15MS, WAIT_4_1MS) of 20-bit size.
2. The s_delay_counter that measures times of the order of μs (WAIT_100μs, WAIT_40μs) of 13-bit size.
3. The 4-bit e_delay_counter that measures when LCD_E (SEND_LCD_E_0X03, SEND_LCD_E_0X02) will be active.

In the input / output signals:

The input in the module is clock, reset and trans_end (the signal for completing a command from FSM OPS)

As output we have SF_D and LCD_E which are the screen control signals (they are active only in the initialization phase when FSM OPS does not work)

Other outputs are trans_act (FSM OPS activation signal), addr_act (Address control activation signal) and op_sel (which command to send).

Finally, we have the 5-bit state register that stores the state we are in.

The code for the counters:

```
always @(posedge clk or posedge reset) begin
  if(reset)
    b_delay_counter<=0;
  else if(state!=WAIT_15_MS && state!=WAIT_4_1_MS && state!=WAIT_1_64_MS)
    b_delay_counter<=0;
  else
    b_delay_counter<=b_delay_counter+1;
end
///….///same code
else if(state!=WAIT_100_MIC_S && state!=WAIT_40_MIC_S)
    s_delay_counter<=0;
///…./// same code
  else if(state!=SEND_LCD_E_0X02 && state!=SEND_LCD_E_0X03)
    e_delay_counter<=0;
///…./// same code
```

As shown by else if, b_delay_counter only works when we have states with ms delays, s_delay_counter counts with ms delays and e_delay when we send 0x02 or 0x03.

Below is the code for the flags:

```
always @(posedge clk or posedge reset) begin
  if(reset)
    state_op_flag<=2'b00;
  else begin
    if(state==WAIT_4_1_MS)
      state_op_flag<=2'b01;
    else if(state==WAIT_100_MIC_S)
      state_op_flag<=2'b10;
    else if(state!=SEND_LCD_E_0X03)
      state_op_flag<=2'b00;
///….///  same code
    if(state==SEND_LCD_E_0X02)
      state_wait_flag<=1'b1;
    else if(state!=WAIT_40_MIC_S)
      state_wait_flag<=1'b0;
///….///  same code
```

State_op_flag and state_wait_flag take values depending on what state we are in and then they are used to go to the next states (from SEND_LCD_E_0X03 and WAIT_40_MIC_S respectively).

Then the next phase of the machine is Configuration, which basically just sends the appropriate op_sel and trans_act signal to FSM OPS to send all the commands sequentially (based on the assignment of the lab exercise). So, in this phase we just control which command will we send and assign the appropriate values to the LCD control signals to the FSM OPS.

Similar to the data sending phase, we do the same process we will just send the orders.

Below is the code for the transition of states to the init phase (part of it):

```
always @(posedge clk or posedge reset) begin
///….///
else begin
   case(state)
            ///***********INIT PHASE**************
           IDLE: state<=WAIT_15_MS;
           WAIT_15_MS:if(b_delay_counter == DELAY_15_MS) state<=SEND_LCD_E_0X03;
           SEND_LCD_E_0X03://we have multiple choices if its the 1st,2nd or 3rd time we visit this state
            if(state_op_flag == 2'b00 && e_delay_counter == DELAY_LCD_E)
                state<=WAIT_4_1_MS;
            else if(state_op_flag == 2'b01 && e_delay_counter == DELAY_LCD_E)
               state<=WAIT_100_MIC_S;
            else if(state_op_flag == 2'b10 && e_delay_counter == DELAY_LCD_E)
               state<=WAIT_40_MIC_S;
           WAIT_4_1_MS: if(b_delay_counter == DELAY_4_1_MS) state<=SEND_LCD_E_0X03;
           WAIT_100_MIC_S: if(s_delay_counter == DELAY_100_MIC_S)  state<=SEND_LCD_E_0X03;
///….///
```

As it appears when state_op_flag has the values we have set for it and e_delay_counter is at the maximum value then from the state SEND_LCD_E_0X03 we go to the correct next one. In addition, in states where we have delays, the trigger for the change is essentially the maximum value of the corresponding counter (eg for WAIT_4_1_MS is DELAY_4_1_MS).

Below is the code for the outputs in the init phase (part of it):

```
always @(state) begin
      casex(state)    ///***********INIT PHASE**************
        5'b000xx://bundled states(idle or delay states)
            begin
               SF_D=4'b0000;     LCD_E=1'b0;   trans_act=1'b0;   op_sel=3'bxxx;  addr_act=1'b0;
            end
         SEND_LCD_E_0X03:
            begin
               SF_D=4'b0011;    LCD_E=1'b1;    trans_act=1'b0;   op_sel=3'bxxx;  addr_act=1'b0;
           end
///….///
```

In the above part we have 000xx (which contains all the states that we are inactive, except WAIT_40_MIC_S) where we set the appropriate values in the signals SF_D, LCD_E, trans_act, op_sel and addr_act (as we have said, it is the signal that activates the Address Control).

In a similar way in the state SEND_LCD_E_0X03 we send the appropriate values to the signals. It is worth mentioning that we write on LCD_E and SF_D with both FSM MAIN and FSM OPS and for this reason the signals from the 2 modules are combined with OR and they work completely complementary so that we do not have an issue.

Below is the code for the transitions of the states to the other phases (part of it):

```
///…./// 
SEND_CLEAR_DISPLAY: if(trans_end) state<=WAIT_1_64_MS;
WAIT_1_64_MS: if(b_delay_counter == DELAY_1_64_MS) state<=SEND_SET_DDRAM_ADDRESS;
            ///**********DRAWING TO SCREEN*************
SEND_SET_DDRAM_ADDRESS: if(trans_end) state<=SEND_WRITE_DATA_TO_DDRAM;
SEND_WRITE_DATA_TO_DDRAM: if(trans_end) state<=SEND_SET_DDRAM_ADDRESS;
///…./// 
```

We can see that the trigger for the transition of states is the trans_end that FSM OPS sends us when it has sent the command to the controller (except for WAIT_1_64_MS where it is when we have measured the appropriate time).

Below is the code for the outputs in the other phases (part of it):

```
///…./// 
  SEND_CLEAR_DISPLAY:
     begin
       SF_D=4'b0000;  LCD_E=1'b0;  trans_act=1'b1;  op_sel=CLEAR_DISPLAY;  addr_act=1'b0;
     end
///…./// 
 ///**********DRAWING TO SCREEN*************
SEND_SET_DDRAM_ADDRESS:
    begin
       SF_D=4'b0000;  LCD_E=1'b0;  trans_act=1'b1;  op_sel=SET_DDRAM_ADDRESS;  addr_act=1'b1;
    end
///…./// 
```

We can see that in the output we just set SF_D and LCD_E to 0 (so that we do not have an issue with the output from FSM OPS), trans_act to 1, op_sel to the appropriate value.

Addr_act is the address control activation signal and therefore remains at 0 as long as we are in the initialization and configuration while it becomes 1 when we go to the SET_DDRAM_ADDRESS command (and we are in the data sending phase)

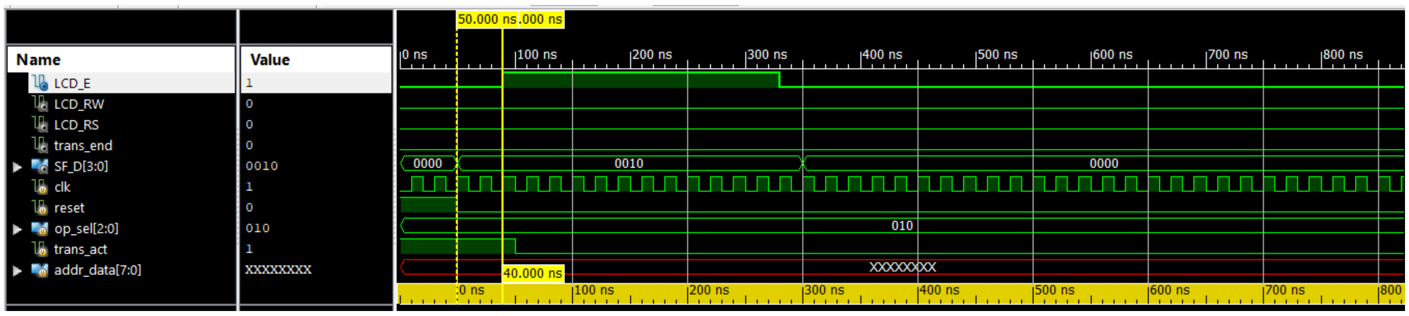
Verification

For the part of the verification of the correct operation of the circuit we use 2 testbenchs. One checks the correct operation of the FSM OPS separately and the other verifies the correct operation of the whole circuit

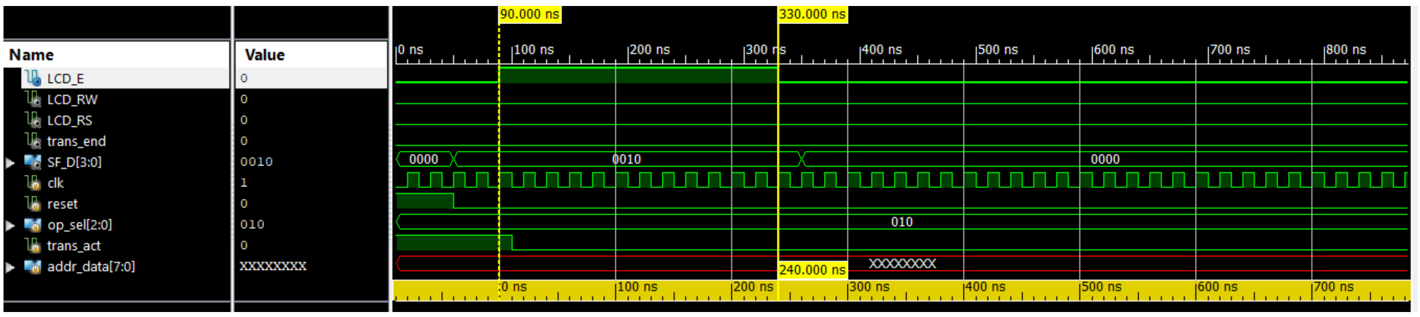
.

FSM OPS testbench

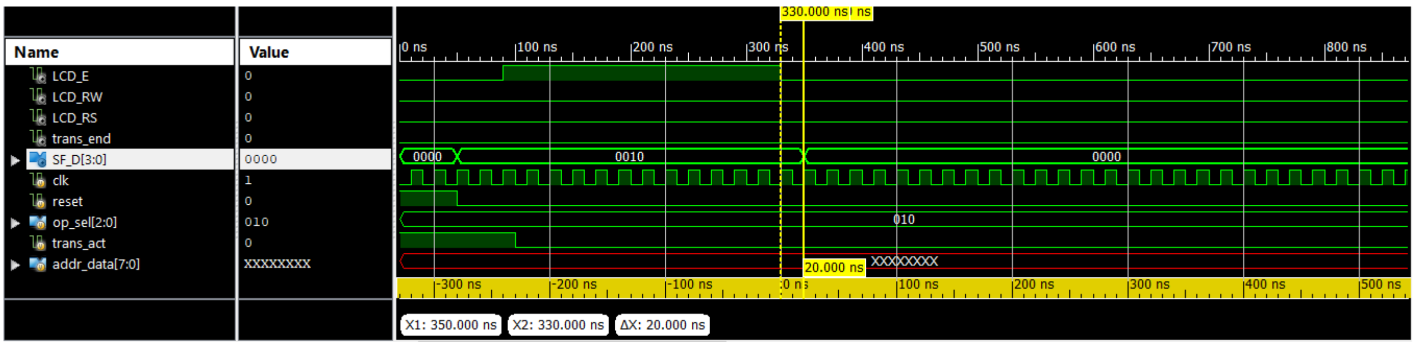
We check for each phase of sending the operation, if all the times are observed and if we have the correct data sending:



Name	Value
LCD_E	1
LCD_RW	0
LCD_RS	0
trans_end	0
SF_D[3:0]	0010
clk	1
reset	0
op_sel[2:0]	010
trans_act	1
addr_data[7:0]	XXXXXXXX

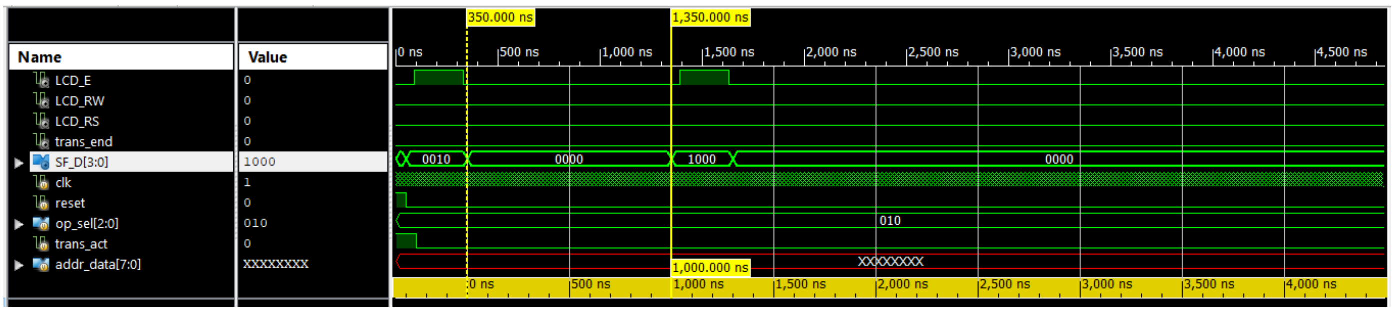


Name	Value
LCD_E	0
LCD_RW	0
LCD_RS	0
trans_end	0
SF_D[3:0]	0010
clk	1
reset	0
op_sel[2:0]	010
trans_act	0
addr_data[7:0]	XXXXXXXX

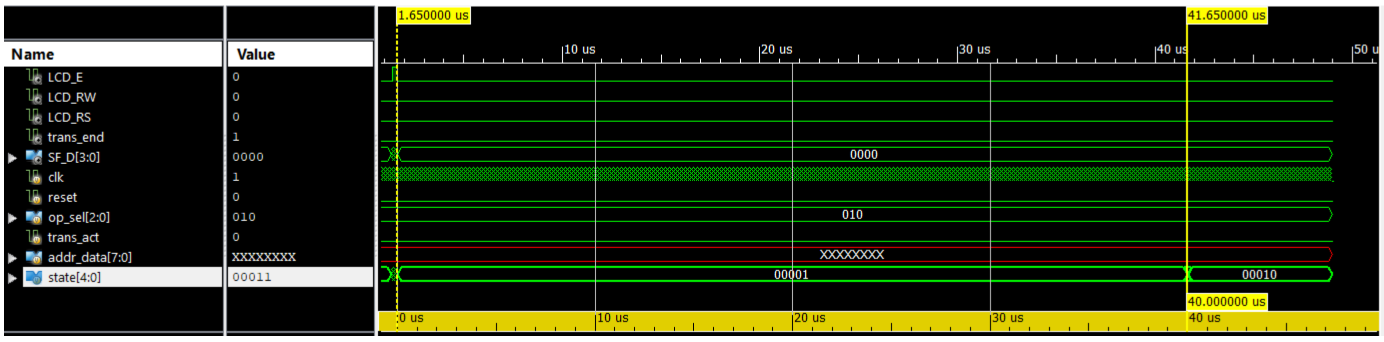


Name	Value
LCD_E	0
LCD_RW	0
LCD_RS	0
trans_end	0
SF_D[3:0]	0000
clk	1
reset	0
op_sel[2:0]	010
trans_act	0
addr_data[7:0]	XXXXXXXX

In the simulation images we have activated FSM OPS. To do this we set the trans_end to high and set the value 010 in op_sel (ie to send the command FUNCTION_SET). In the first phase we send the UPPER (0x2) middle of the channel SF_D. For LCD_E, we have setup times 40ns, active high 240ns and hold 20ns as we said in the implementation analysis.



Name	Value
LCD_E	0
LCD_RW	0
LCD_RS	0
trans_end	0
SF_D[3:0]	1000
clk	1
reset	0
op_sel[2:0]	010
trans_act	0
addr_data[7:0]	XXXXXXXX



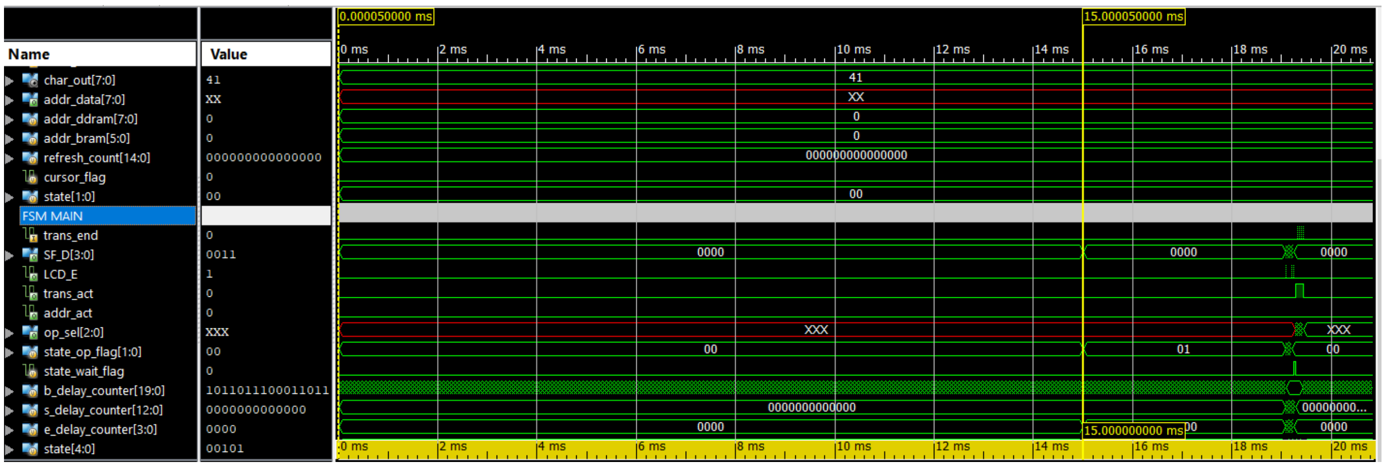
Name	Value
LCD_E	0
LCD_RW	0
LCD_RS	0
trans_end	1
SF_D[3:0]	0000
clk	1
reset	0
op_sel[2:0]	010
trans_act	0
addr_data[7:0]	XXXXXXXX
state[4:0]	00011

In the following pictures we have the delay of 1µs the mission of 0x8 and then the delay of 40µs. Since the times are correct, based on the simulation, we can conclude that the operation of the module is correct.

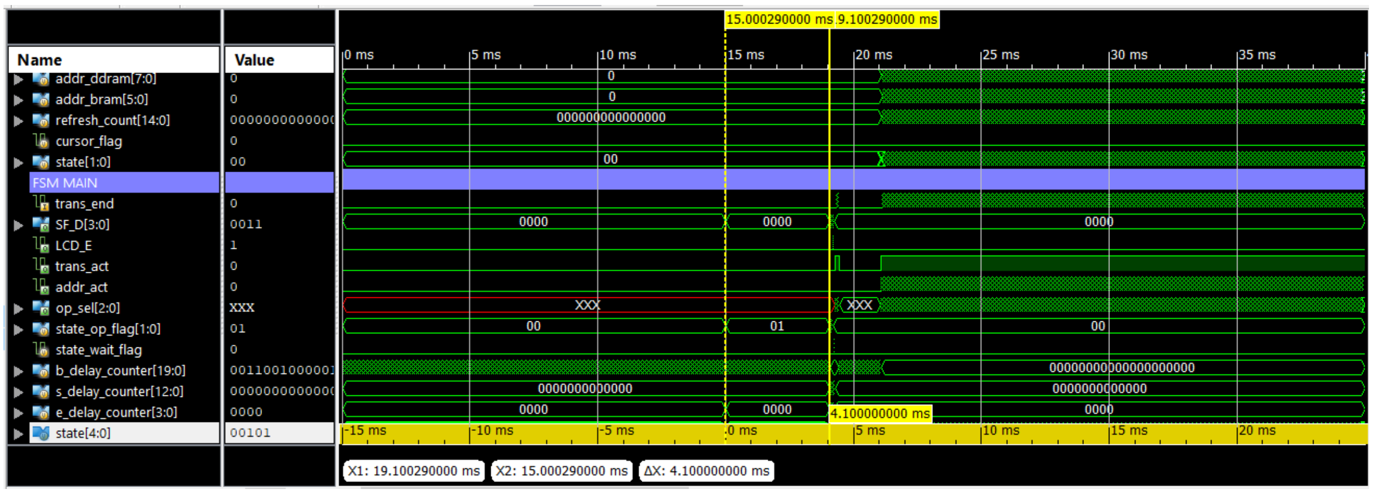
<u>FSM MAIN testbench</u>

In this simulation we control the operation of the entire driver.

First we show the times for the delays in the states of FSM MAIN:

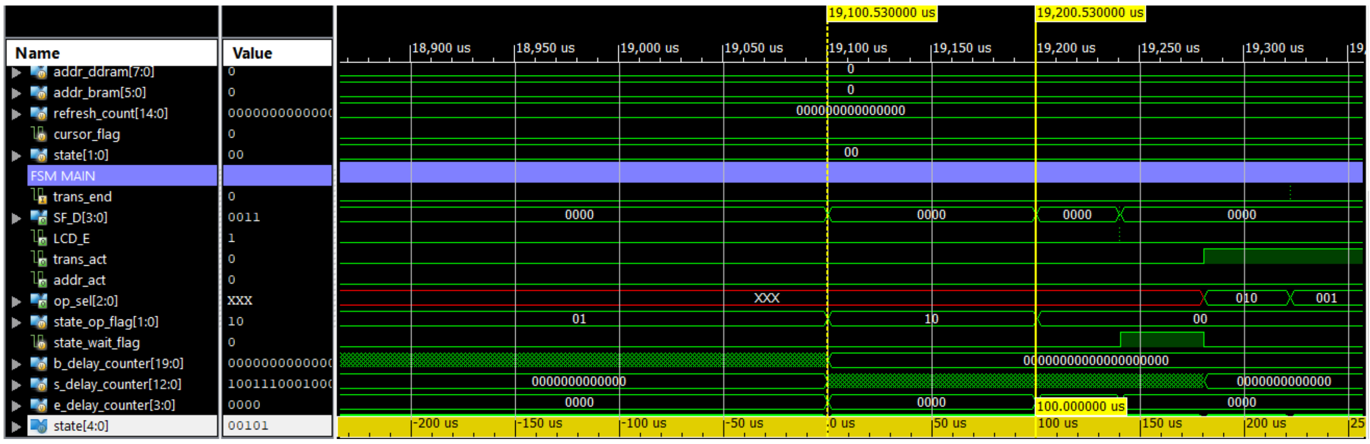


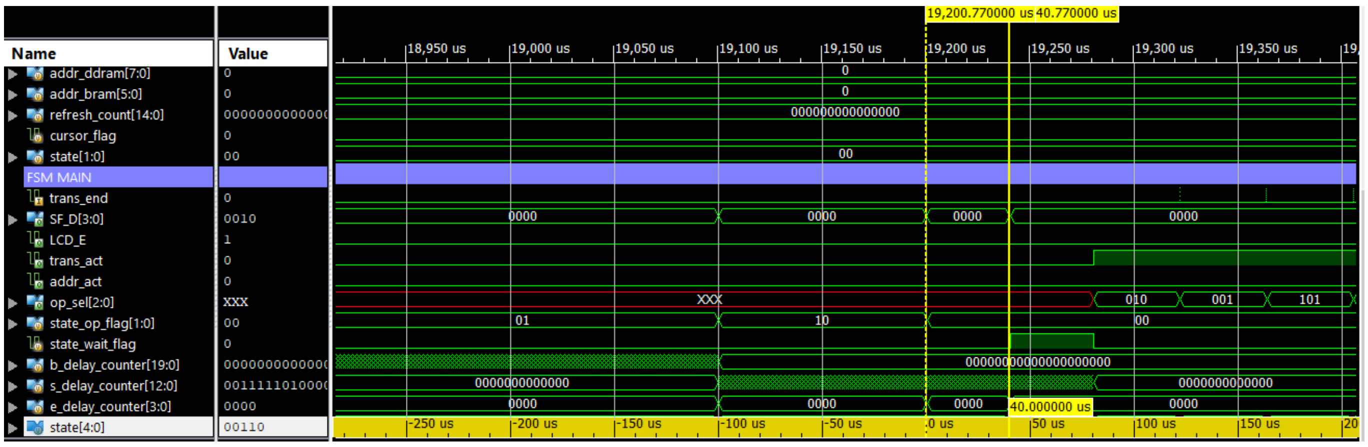
Name	Value
char_out[7:0]	41
addr_data[7:0]	XX
addr_ddram[7:0]	0
addr_bram[5:0]	0
refresh_count[14:0]	000000000000000
cursor_flag	0
state[1:0]	00
FSM MAIN	
trans_end	0
SF_D[3:0]	0011
LCD_E	1
trans_act	0
addr_act	0
op_sel[2:0]	XXX
state_op_flag[1:0]	00
state_wait_flag	0
b_delay_counter[19:0]	1011011100011011
s_delay_counter[12:0]	0000000000000
e_delay_counter[3:0]	0000
state[4:0]	00101

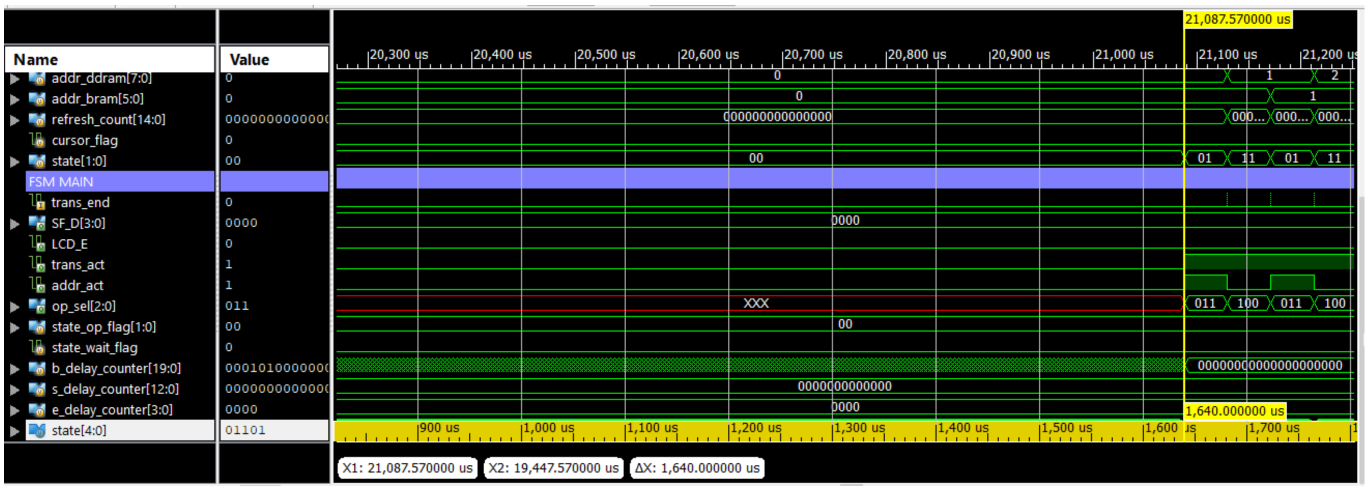


Name	Value
addr_ddram[7:0]	0
addr_bram[5:0]	0
refresh_count[14:0]	000000000000
cursor_flag	0
state[1:0]	00
FSM MAIN	
trans_end	0
SF_D[3:0]	0011
LCD_E	1
trans_act	0
addr_act	0
op_sel[2:0]	XXX
state_op_flag[1:0]	01
state_wait_flag	0
b_delay_counter[19:0]	0011001000001
s_delay_counter[12:0]	000000000000
e_delay_counter[3:0]	0000
state[4:0]	00101

X1: 19.100290000 ms X2: 15.000290000 ms ΔX: 4.100000000 ms

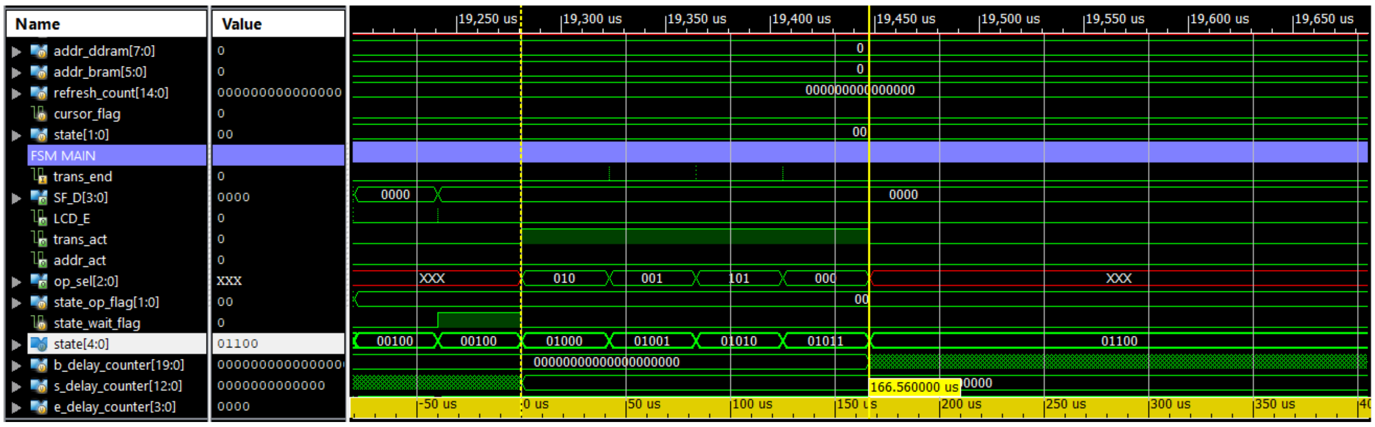






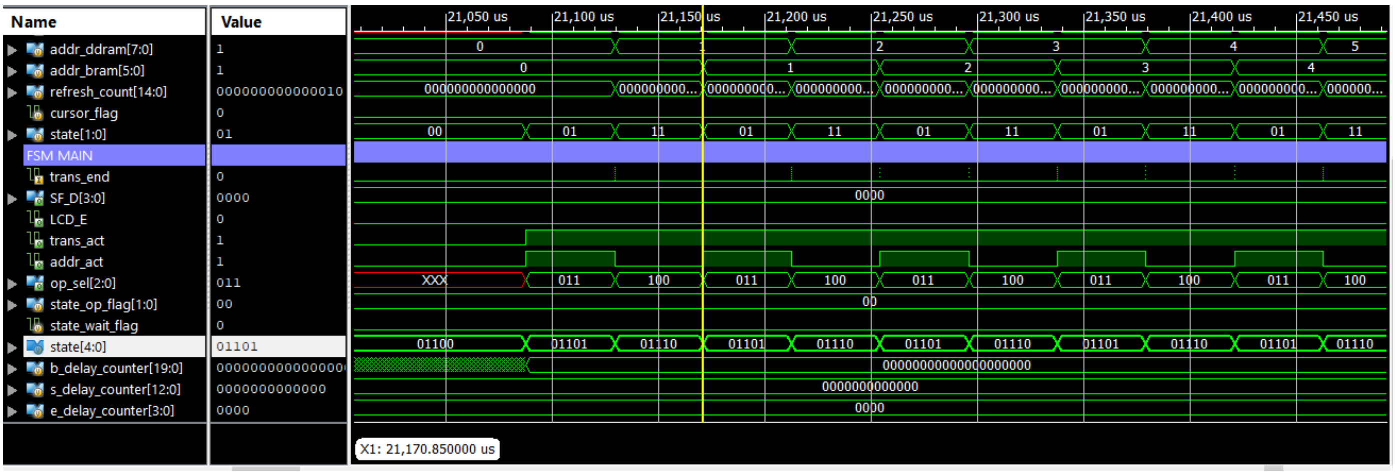
Apparently all times during initialization and configuration are correct (times for delays).

Then we check if all the commands are sent in order for the configuration phase



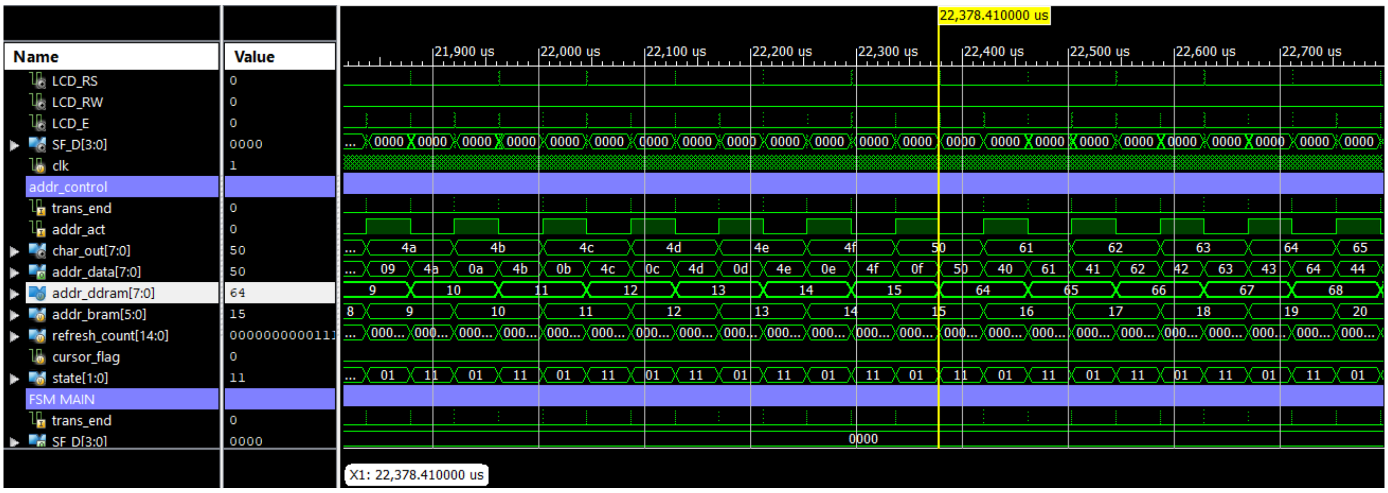
As it appears after the last delay (40μs) in the initialization phase, we move on to the configuration phase where we send the 4 different commands to the controller. Based on the state we are in we can see that this process is actually done correctly.

Finally, the data sending phase remains where we are just in an endless repetition (we send SET_DDRAM_ADDRESS and then WRITE_DATA_TO_DDRAM again and again).

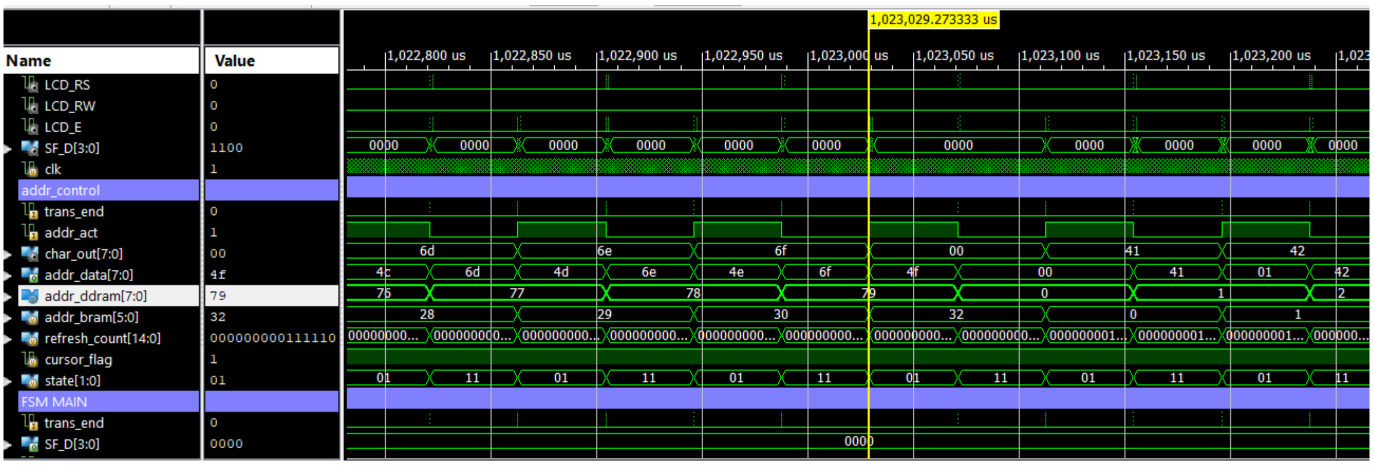


As shown in the simulation image the repeated states are 01101 and 01110 (the commands mentioned earlier).

The last thing we check through the testbench is the function of the Address Control module .



In the above snapshot we have the function of address control. After FSM MIN sends addr_act to high then it comes out of IDLE mode and sends data. As we see in the addr_data channel we send alternately the address we write and then the data we want to show. With addr_pointers we keep the address to which we must write and read respectively. A special case for addr_ddram is shown in the cursor, where when it becomes 15 we have to go to 64 (hex 0x40) to write on the second line of the screen.



In the last simulation image we have the special case for addr_bram where it normally counted to 31 and reset (message without cursor on the LCD), while cursor_flag was at low. Now that it is at high when we reach 30 we skip to 31 and go to 32 (ie we display the symbol we have defined as a cursor on the screen).

Experiment/Final implementation

The experiment required only two tests on the board for the LCD driver to work successfully.

In the first test the screen appeared to be initialized correctly (the squares on the screen disappeared) but no message was displayed. was at 1 so we do not have the correct command to apply in each case.

After this was corrected, in the second test we had a successful operation of the LCD driver, with every requirement of the assignment being met.

Conclusion

In the laboratory exercise we wanted to implement a LCD driver in circuit form in Verilog.

Successful operation of the circuit required only two tests on the board (in the same lab class).

Although it was the last laboratory exercise, difficulties did not arise for the following reasons:

1. We had strict restrictions on how each function should be implemented (timings, order and value of control signals).
2. The FSM (at least for me), if coded correctly, are easy to extend their function and control.
3. We had experience from previous laboratory exercises and a more extensive test was performed on the simulated behavior of the circuit (where in fact most errors are made).

In conclusion, with the help of the laboratory assistants, our theoretical knowledge and the practical control on the board, the 4th laboratory exercise was successful.