Mini Project - VGA Interface and LFSR

Maryam Hemmati

Department of Electrical, Computer and Software Engineering
University of Auckland

email: m.hemmati@auckland.ac.nz

COMPSYS 305-Digital Systems Design

6 April 2022

Content

1. VGA Interface Review
2. Graphics Display on VGA Screen
3. Text Display on VGA Screen
4. Linear Feedback Shift Register (LFSR)

VGA Interface - Review

Image on VGA screen is displayed by turning the pixels ON and OFF.

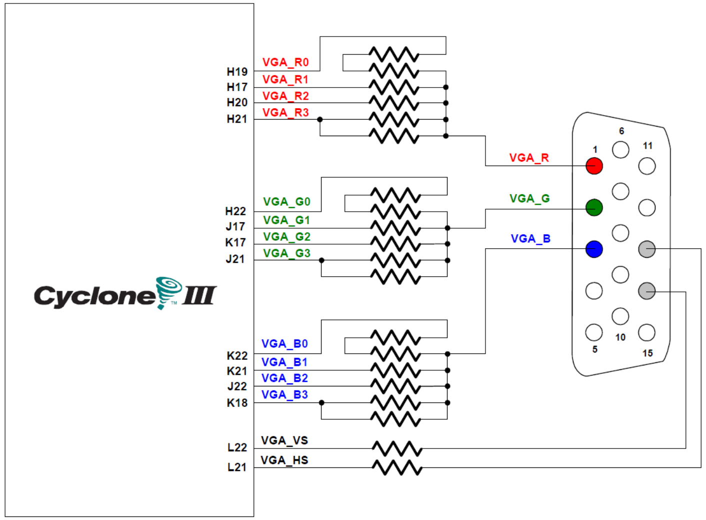
- Video signal must redraw the entire screen 60 times per sec (**60Hz**) to avoid flickers.
  - Human eyes detect flickers at refresh rate less than 30Hz.
- We will use the common VGA display standard at **25MHz** pixel rate with **640x480** resolution.
  - Each pixel takes 40ns at 25MHz pixel rate.

VGA Interface - Review

VGA video standard contains 5 active signals:

- **Horizontal** and **vertical synchronisation** signals.
- Three analog signals for **red**, **green** and **blue** (**RGB**) colours formation.
  - By changing the analog voltage levels of the RGB signals, different colours can be produced.
  - Depending on the number of bits supported by the development board, different amount of colours can be represented.

VGA Interface - Review



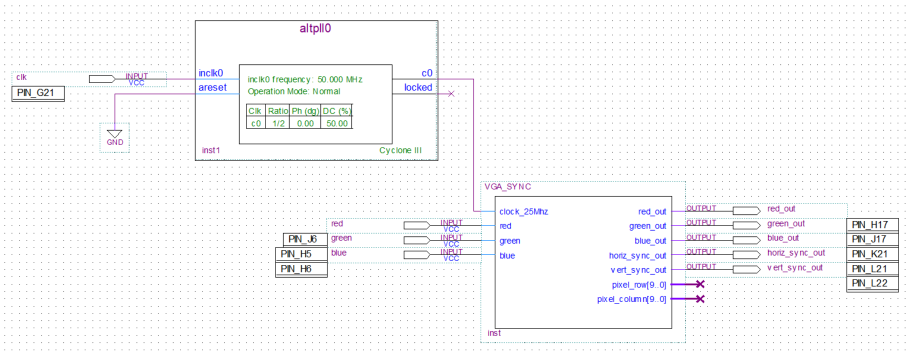
VGA Interface - VGA_Sync Component - Review

We need a component to drive the control signals to the display and provide pixel values at the right rate.

- In order to generate the VGA signal at 25 MHz, the clock signal provided by DE0 (50MHz) needs to be halved.
- 25MHz clock signal can be used by counters to generate the horizontal and vertical sync signals.
- The counters also represent row and column address of a pixel, which can be used by other components to retrieve pixel information.

VGA Interface - Example

Try this simple example and see how you can change the background colour on your VGA screen by using three switches on the DE0 board:



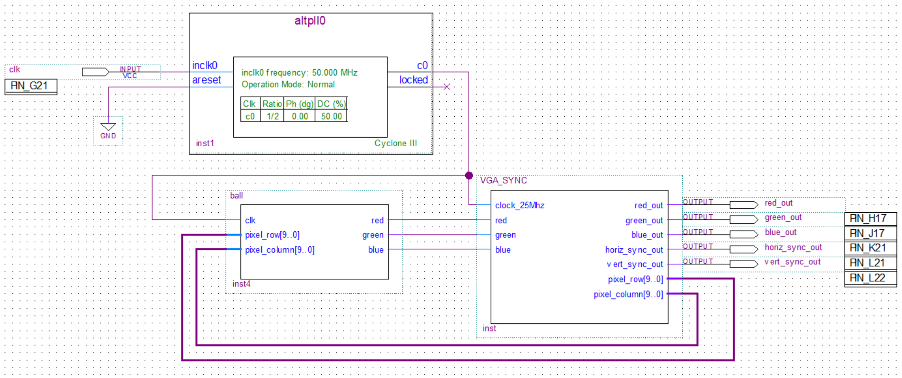
Graphics Display - Ball Example

- (x,y) position of the square are set to some constant values.
- Background colour and ball colour are defined as white and red respectively.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;


ENTITY ball IS
    PORT
        (SIGNAL clk                         : IN std_logic;
        SIGNAL pixel_row, pixel_column      : IN std_logic_vector(9 DOWNTO 0);
        SIGNAL red, green, blue             : OUT std_logic);
END ball;

architecture behavior of ball is

SIGNAL ball_on                  : std_logic;
SIGNAL size                     : std_logic_vector(9 DOWNTO 0);
SIGNAL ball_y_pos, ball_x_pos   : std_logic_vector(9 DOWNTO 0);

BEGIN

size <= CONV_STD_LOGIC_VECTOR(8,10);
-- ball_x_pos and ball_y_pos show the (x,y) for the centre of ball
ball_x_pos <= CONV_STD_LOGIC_VECTOR(590,10);
ball_y_pos <= CONV_STD_LOGIC_VECTOR(350,10);


ball_on <= '1' when ( ('0' & ball_x_pos <= pixel_column + size) and ('0' & pixel_column <= ball_x_pos + size)
                    and ('0' & ball_y_pos <= pixel_row + size) and ('0' & pixel_row <= ball_y_pos + size) )  else
            '0';


-- Colours for pixel data on video signal
-- Keeping background white and square in red
Red <=  '1';
-- Turn off Green and Blue when displaying square
Green <= not ball_on;
Blue <=  not ball_on;

END behavior;
```

Graphics Display - Ball Example

Try this example to see the red square on white background. You may change the colour and position of the square in ball component.



Graphics Display - Bouncy Ball Example

The motion feature is added to our simple object to make it bounce off the edges.

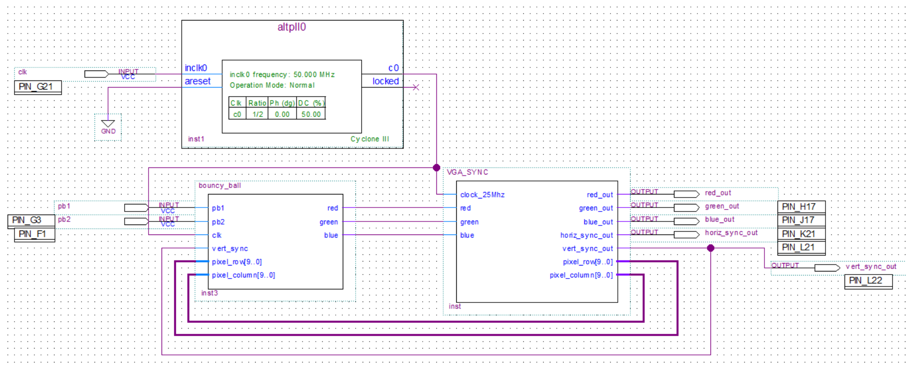
- The new position of the ball should be updated once for each frame.
  - One update per each vertical sync.
- Ball position is calculated by adding its current Y position and its vertical motion.
- Screen boundaries are checked; ball speed is changed once it reaches the boundaries at row 0 and 479.
- Two pushbuttons are used to change the background and ball colour.

Graphics Display - Bouncy Ball Example

```vhdl
ENTITY bouncy_ball IS
    PORT
        (SIGNAL pb1, pb2, clk, vert_sync    : IN std_logic;
        SIGNAL pixel_row, pixel_column      : IN std_logic_vector(9 DOWNTO 0);
        SIGNAL red, green, blue             : OUT std_logic);
END bouncy_ball;

architecture behavior of bouncy_ball is

SIGNAL ball_on                  : std_logic;
SIGNAL size                     : std_logic_vector(9 DOWNTO 0);
SIGNAL ball_y_pos               : std_logic_vector(9 DOWNTO 0);
SiGNAL ball_x_pos               : std_logic_vector(10 DOWNTO 0);
SIGNAL ball_y_motion            : std_logic_vector(9 DOWNTO 0);

BEGIN

size <= CONV_STD_LOGIC_VECTOR(8,10);
-- ball_x_pos and ball_y_pos show the (x,y) for the centre of ball
ball_x_pos <= CONV_STD_LOGIC_VECTOR(590,11);

ball_on <= '1' when ( ('0' & ball_x_pos <= '0' & pixel_column + size) and ('0' & pixel_column <= '0' & ball_x_pos + size)
                and ('0' & ball_y_pos <= pixel_row + size) and ('0' & pixel_row <= ball_y_pos + size) )  else
            '0';


-- Colours for pixel data on video signal
-- Changing the background and ball colour by pushbuttons
Red <=  pb1;
Green <= (not pb2) and (not ball_on);
Blue <=  not ball_on;


Move_Ball: process (vert_sync)
begin
    -- Move ball once every vertical sync
    if (rising_edge(vert_sync)) then
        -- Bounce off top or bottom of the screen
        if ( ('0' & ball_y_pos >= CONV_STD_LOGIC_VECTOR(479,10) - size) ) then
            ball_y_motion <= - CONV_STD_LOGIC_VECTOR(2,10);
        elsif (ball_y_pos <= size) then
            ball_y_motion <= CONV_STD_LOGIC_VECTOR(2,10);
        end if;
        -- Compute next ball Y position
        ball_y_pos <= ball_y_pos + ball_y_motion;
    end if;
end process Move_Ball;

END behavior;
```


Graphics Display - Bouncy Ball Example

Try this example and see how you can change the colour of background and bouncy ball by using **pushutton 1** and **pushbutton 2** on DE0 board:



Graphics Display - Bouncy Ball Example

- When **no button** is pressed:
  - Pixels showing the ball has R='1', G='0', B='0': **Red ball**
  - Background pixels has R='1', G='0', B='1': **Magenta background**
- When only **pushbutton 1** is pressed:
  - Pixels showing the ball has R='0', G='0', B='0': **Black ball**
  - Background pixels has R='0', G='0', B='1': **Blue background**
- When only **pushbutton 2** is pressed:
  - Pixels showing the ball has R='1', G='0', B='0': **Red ball**
  - Background pixels has R='1', G='1', B='1': White background
- When both **pushbutton 1 and 2** are pressed:
  - Pixels showing the ball has R='0', G='0', B='0': **Black ball**
  - Background pixels has R='0', G='1', B='1': **Cyan background**

Text Display

If we want to put a text on the screen, we need to know the pattern of characters.

- Based on the character pattern, pixel row, and column information, we decide on RGB values to be sent to the VGA_Sync component.
- The following lines of code can put **H** on the screen:

```
if (((8<row<18) and (col = 8)) or ((8<row<18) and (col = 13))
  or ((row=13) and (8<col<13))) then
    red <= '1';
else
    red <= '0';
end if;
```

We can store the display pattern of characters in a memory and access the memory for writing text on the screen.

Text Display

A group of characters are stored in a memory block in the FPGA.

- This memory is instantiated in the ***char_rom.vhd***
- The memory should be initialized with the information of character patterns.
  - A *.mif file is used to initialize the memory.
  - TCGROM.mif is the memory initialization file that contains the patterns of 64 characters.
  - Each character in a .mif file is described through 8 lines of memory address and is translated to a block of 8x8 pixels.

Address	Font Data
000001000 :	00011000 ;
000001001 :	00111100 ;
000001010 :	01100110 ;
000001011 :	01111110 ;
000001100 :	01100110 ;
000001101 :	01100110 ;
000001110 :	01100110 ;
000001111 :	00000000 ;

8 x 8 Font Pixel Data

0	0	0	1	1	0	0	0
0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	0	0	0	0	0	0	0

Text Display

char_rom.vhd gets an instance of ***altsyncram*** component which is a memory IP core.

```
ENTITY char_rom IS
    PORT
    (
        character_address   :   IN STD_LOGIC_VECTOR (5 DOWNTO 0);
        font_row, font_col  :   IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        clock               :   IN STD_LOGIC ;
        rom_mux_output      :   OUT STD_LOGIC
    );
END char_rom;


ARCHITECTURE SYN OF char_rom IS

    SIGNAL rom_data     : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL rom_address  : STD_LOGIC_VECTOR (8 DOWNTO 0);

    COMPONENT altsyncram
    GENERIC (
        address_aclr_a          : STRING;
        clock_enable_input_a    : STRING;
        clock_enable_output_a   : STRING;
        init_file               : STRING;
        intended_device_family  : STRING;
        lpm_hint                : STRING;
        lpm_type                : STRING;
        numwords_a              : NATURAL;
        operation_mode          : STRING;
        outdata_aclr_a          : STRING;
        outdata_reg_a           : STRING;
        widthad_a               : NATURAL;
        width_a                 : NATURAL;
        width_byteena_a         : NATURAL
    );
    PORT (
        clock0      : IN STD_LOGIC ;
        address_a   : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
        q_a         : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
    END COMPONENT;
```

Text Display

We only need to provide ***rom_address*** and extract one bit of ***rom_data*** as an output for each pixel.

```
BEGIN

    altsyncram_component : altsyncram
    GENERIC MAP (
        address_aclr_a => "NONE",
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        init_file => "tcgrom.mif",
        intended_device_family => "Cyclone III",
        lpm_hint => "ENABLE_RUNTIME_MOD=NO",
        lpm_type => "altsyncram",
        numwords_a => 512,
        operation_mode => "ROM",
        outdata_aclr_a => "NONE",
        outdata_reg_a => "UNREGISTERED",
        widthad_a => 9,
        width_a => 8,
        width_byteena_a => 1
    )
    PORT MAP (
        clock0 => clock,
        address_a => rom_address,
        q_a => rom_data
    );

    rom_address <= character_address & font_row;
    rom_mux_output <= rom_data (CONV_INTEGER(NOT font_col(2 DOWNTO 0)));

END SYN;
```


Text Display

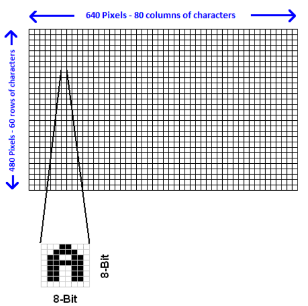
The following table shows the contents of the CharROM which is initialized through TCGROM.mif file.

- Memory depth is **512**.
- Memory width is **8**. The content of memory for each address is an 8-bit value.
- Notice that the address is in **Oct** format.

CHAR	ADDRESS	CHAR	ADDRESS	CHAR	ADDRESS	CHAR	ADDRESS
@	*00*	P	*20*	Space	*40*	0	*60*
A	*01*	Q	*21*	!	*41*	1	*61*
B	*02*	R	*22*	"	*42*	2	*62*
C	*03*	S	*23*	#	*43*	3	*63*
D	*04*	T	*24*	$	*44*	4	*64*
E	*05*	U	*25*	%	*45*	5	*65*
F	*06*	V	*26*	&	*46*	6	*66*
G	*07*	W	*27*	'	*47*	7	*67*
H	*10*	X	*30*	(	*50*	8	*70*
I	*11*	Y	*31*	)	*51*	9	*71*
J	*12*	Z	*32*	*	*52*	A	*72*
K	*13*	[	*33*	+	*53*	B	*73*
L	*14*	Dn Arrow	*34*	,	*54*	C	*74*
M	*15*	]	*35*	-	*55*	D	*75*
N	*16*	Up Arrow	*36*	.	*56*	E	*76*
O	*17*	Lft Arrow	*37*	/	*57*	F	*77*

Text Display

- The way we use part of pixel-row and pixel-column value as the address to the CharROM defines the size of the text.
  - If we use 3 lower bits of the pixel-row address, we will get the text in its original size of 8x8.
- To make characters larger, each dot in the font should map to several pixels.
  - To double the size, each dot should map to a 2x2 pixel block.
  - pixel-row[3 downto 1] and pixel-column[3 downto 1] are used as the font row and font column.

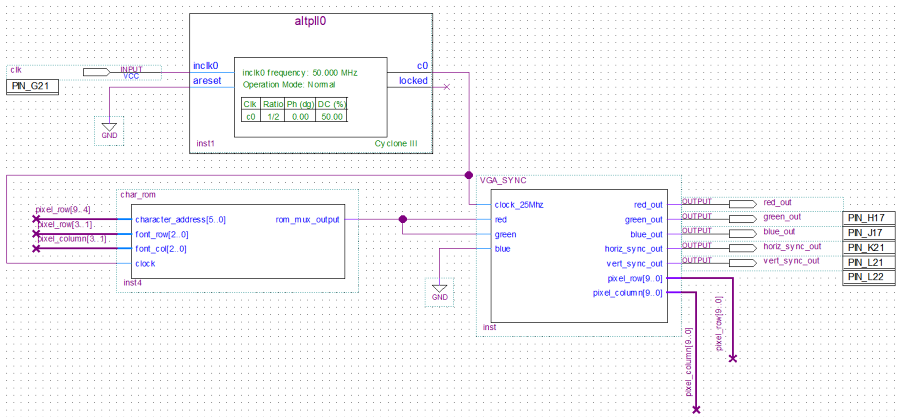


Pixel Row (10-bit)

	Character Address -	Font Row
0	000000	000
0	000000	001
0	000000	010
0	000000	011
0	000000	100
0	000000	101
0	000000	110
0	000000	111
0	000001	000
0	000001	001
0	000001	010
0	000001	011
0	000001	100
0	000001	101
0	000001	110
0	000001	111
0	000010	000
0	000010	001
0	000010	010
0	000010	011
0	000010	100
0	000010	101
0	000010	110
0	000010	111

Text Display Example

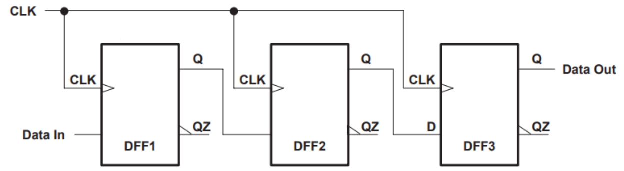
Try this example and see how you can fill the screen with rows of different characters:



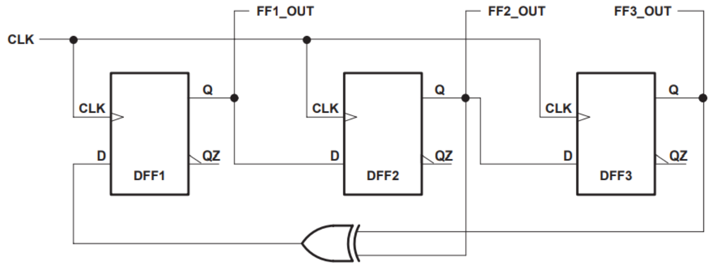
Linear Feedback Shift Register (LFSR)

- A linear feedback shift register **(LFSR)** is a shift register whose input bit is the output of a linear function of two or more bits of its previous states.
- The linear feedback can be formed by performing exclusive-OR on the outputs of two or more of the flip flops together.
  - Alternatively XNOR can be used for the feedback.
- LFSRs can be used in variety of applications such as
  - Pseudo-random number generators
  - Test pattern generation
  - Cyclic Redundancy Check (CRC)
  - Cryptography

Linear Feedback Shift Register (LFSR)



A 3-Bit Shift Register



Linear Feedback Shift Register

Linear Feedback Shift Register (LFSR)

- The points within the register chain, where the feedback comes from are called **taps**.
  - Taps are the bits that influence the output.
  - Two LFSRs with the same seed but different taps generate different sequences.
- The initial value of the LFSR is called the **seed**.
  - It should be a **non-zero** value, otherwise LFSR would be stuck at the seed value.
- An LFSR is of maximal length if it sequence through every possible value.
  - A maximal length **n-bit** LFSR can sequence through $2^n - 1$ values.
  - The state "0000. . . " (all zeros) is not included in the sequence.

Linear Feedback Shift Register (LFSR)

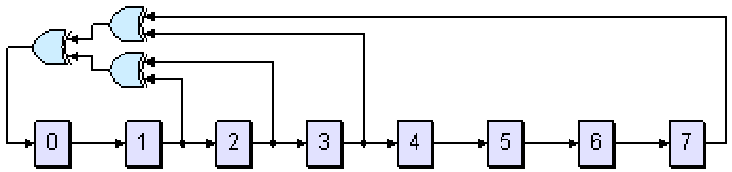
- The choice of taps determines how many values there are in a given sequence before the sequence is repeated.
- Some tap choices for maximal length sequence is provided:

Number of bits	Length of loop	Taps
2	3	0,1
3	7	0,2
4	15	0,3
5	31	1,4
6	63	0,5
7	127	0,6
8	255	1,2,3,7
9	511	3,8
10	1023	2,9
11	2047	1,10

Linear Feedback Shift Register (LFSR)

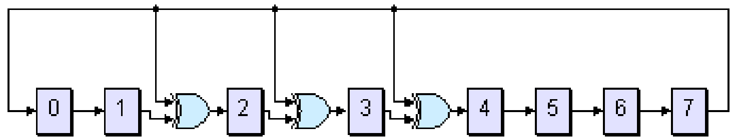
There are two types of LFSRs, depending on how feedback is formed:

- In **Fibonacci LFSR**, the XOR gates (taps), are placed on the feedback path.
- Increasing the levels of logic in the combinational feedback path can **negatively impact** the maximum clocking frequency of the function.



Linear Feedback Shift Register (LFSR)

- In **Galois LFSR**, the XOR gates (taps), are placed between the registers.
- Galois type is more recommended in this project.



Summary

- We looked at VGA interface and discussed how to show graphics and text on the VGA screen through several examples.
- We introduced LFSR to be used as a pseudo-random number generator.
  - LFSR can be used in your mini project to generate random values for the gaps in the pipes.

Acknowledgment

- Some figures/notes are taken from or inspired by the
  - CS305 Lecture notes by Muhammad Nadeem, 2019