FPGA Based Digital System Design

7067CEM

Coursework Submission

Module Leader – **Amjad S. Khan**

Prepared by:
Samuel Iyiolade Osafehinti
SID - 14283716
osafehints@uni.coventry.ac.uk

Walid Bouzidi
SID - 14054035
bouzidiw@uni.coventry.ac.uk

Paras Mahajan
SID - 14324134
mahajanp3@uni.coventry.ac.uk

March 30th, 2024

TABLE OF CONTENTS

1. HARDWARE REQUIREMENTS

1.1 PROJECT DESIGN SPECIFICATIONS AND REQUIREMENTS

The project requirement is the design of an FPGA based driver for an LED Dot-matrix Display.

1.2 PROVIDED HARDWARE

The following are the hardware components provided to us to carry out the task of fulfilling the design requirements according to specifications.

DIGILENT NEXYS A7 FPGA DEVELOPMENT BOARD

The **FPGA** utilized is the **XILINX ARTIX-7** (XC7A100T-1CSG324C) made available on the Digilent NEXYS A7 development board. The development board is shown in **Fig:1** below.

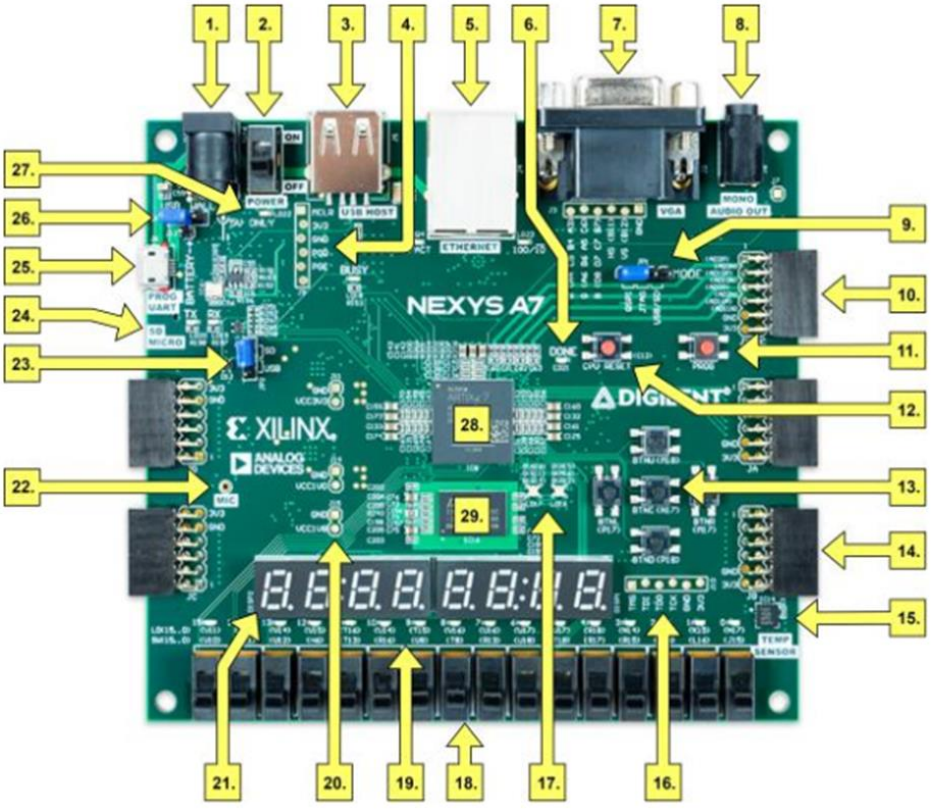


Fig1: Digilent NEXYS A7 FPGA Board [1]

8x8 LED DOT-MATRIX COLUMN-ANODE DISPLAY

The dot-matrix display provided is the 8x8 column-anode, 3mm RED LED dot-matrix display with part number **1088BS.** [2]

A diagram of the internal connection structure of the LED matrix and a picture of the physical display is shown below.

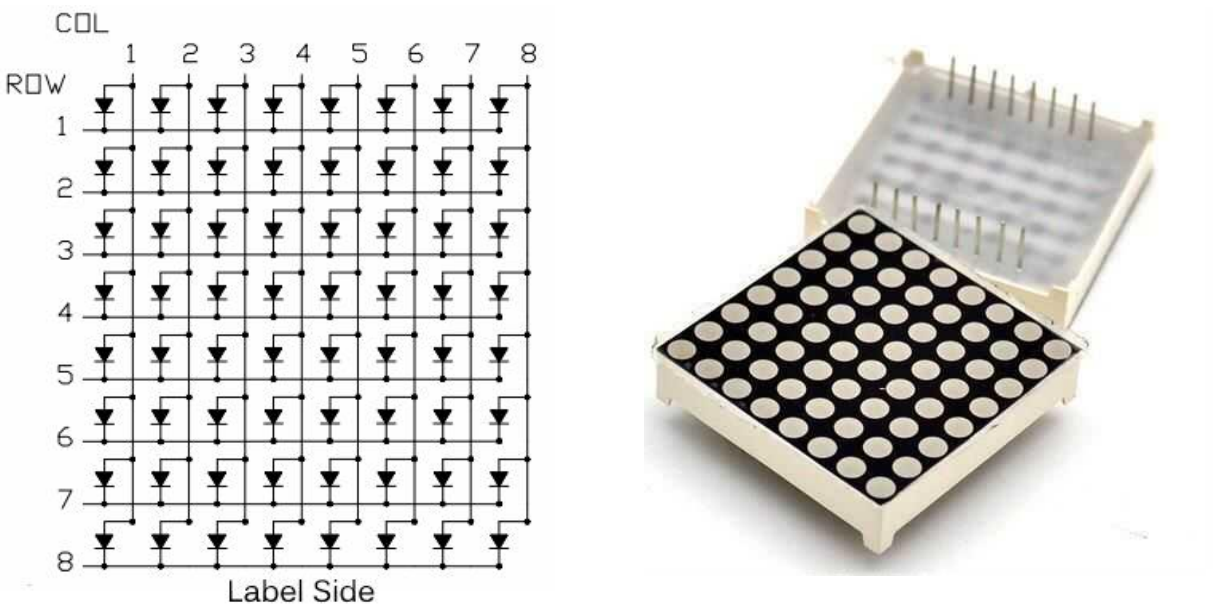


Fig2: Dot-matrix Display Module and Internal Connection

1.3 DESIGN APPROACH AND METHODOLOGY

The design approach follows these two processes:

- Hardware Implementation
- VHDL logic implementation

Each of these processes is discussed below.

1.3.1. HARDWARE IMPLEMENTATION

To properly implement a design based on any hardware, a detailed understanding of the hardware is necessary. The primary hardware component for this design is the LED Dot-Matrix display, therefore, the display physical layout and electrical characteristics must be properly understood to proceed.

LED DOTMATIX-DISPLAY

The dot-matrix display comprises an array of LEDs arranged in a matrix configuration of COLUMN and ROW intersections. This configuration is usually specified in their specification sheets as (COLUMN x ROW). For instance, a Dot-Matrix display with a five LED COLUMNs and a seven LED ROWs is specified as a (5x7) LED dot-matrix display in its specification sheet.
LEDs on the same row and column are linked together, therefore, each LED represents a unique connection of a and a COLUMN ROW.
Also, since each LED on the matrix has two terminals, anode, and cathode, there are two possible configurations of the LED matrix formation:

- Column-Anode Dot-matrix Display
- Column-Cathode Dot-matrix Display

Our provided dot-matrix display is the column-anode type as shown previously in **Fig:2.**

DOT-MATRIX DISPLAY SCANNING

Because the LEDs on a dot matrix display are arranged in an array form (each LED is a connection of a COLUMN and a ROW), the LEDs MUST be lit in a special way to display any character. This special lighting method (as concerned LED Dot-matrix Displays) is called **Display Scanning**. This involves lighting a specific LED or a specific group of LEDs on the matrix for a fraction of a second continually. Then, because of the **"persistence of vision"** phenomenon in the human eye, previously lit LEDs are still seen. The complete intended character is fully observed by the human viewer during the scan process.

For a typical LED dot matrix, one of the following three scanning methods can be used. The eventual scanning choice is a function of the length of the display, (if more are cascaded), intended brightness and flickering elimination.

These scanning methods are:

- Dot Scanning
- Horizontal Scanning
- Vertical Scanning

Each scanning method is illustrated below using the column-anode configuration.

Dot Scanning

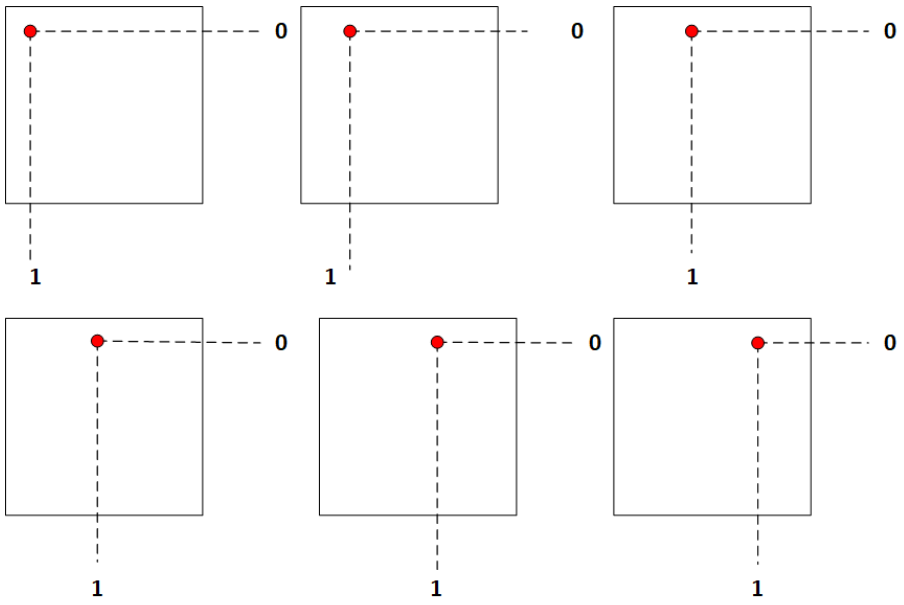


Fig.3: Dot Scanning

Horizontal Scanning

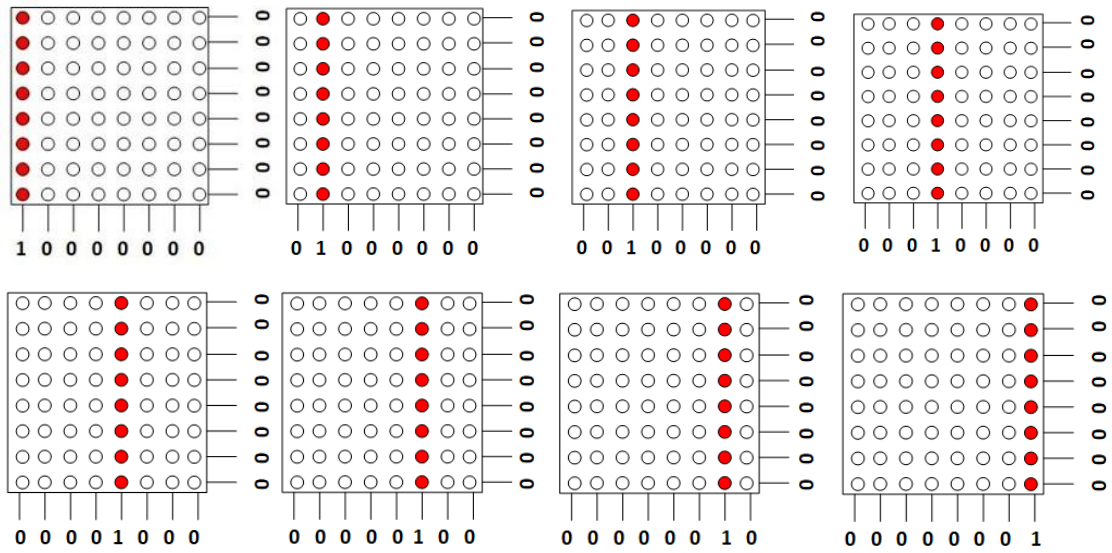


Fig.4: Horizontal Scanning

Vertical Scanning

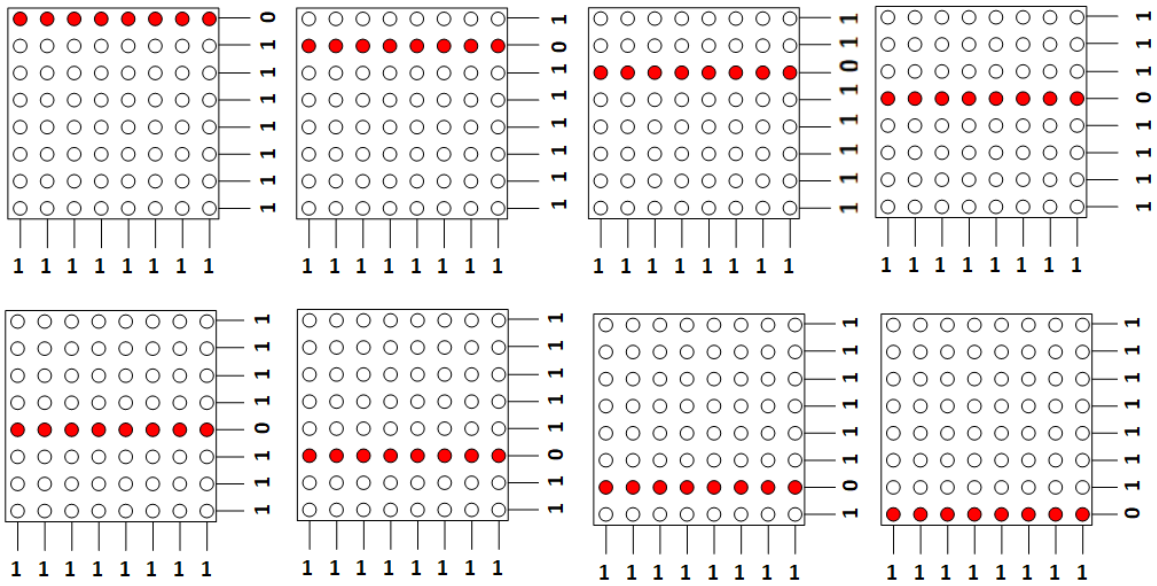


Fig.5: Vertical Scanning

The choice of scanning method used is a function of the intended brightness and screen refresh speed and flicker elimination.

In most applications employing the use of the dot-matrix display, vertical scanning is used. Why?

In most use cases, multiple LED display-matrices are cascaded horizontally for longer character applications while the height vertically remains constant. Taking a cascading of four LED dot-matrices as an example would result in an effective matrix size of 32x8 (32 columns by 8 rows). Vertically, 32 columns need to be scanned but vertically, only 8 rows need to be scanned.

As an effective illustration, consider a display scan period of 16ms (a refresh rate of 62.5Hz) and an even distribution of this time among each scanned element. The table below shows the time allotted to each element of each display scanning method implemented on a **32x8 LED dot-matrix display**. The higher the time, the brighter the LED and vice versa.

From the table, it can easily be deduced that the vertical scan would produce better outcomes on the dot-matrix display.

Timing analysis of scanning methods

Scanning method	**Scan steps**	**On-time in 16 ms time frame**	**Entity**
Vertical (row by row)	8	16 ms / 8 rows = 2 ms	All leds on a row
Horizontal (col by col)	32	16 ms / 32 cols = 0.5 ms	All leds on a columns
Dot by dot	256	16 ms / 256 dots = 0.0625 ms	Each led on the matrix

HARDWARE CONNECTIONS

From the display scanning illustration in Fig: 5, opposite logic must be applied to the ROWS and the COLUMNS to light any LED on the matrix. Our LED matrix is the column-anode type; therefore, HIGH logic must be applied to the COLUMN and LOW logic must be applied to the ROWS to light any LED on a scanned ROW.

In this design case, the FPGA is to supply this logic. A typical connection to the FPGA demo-board is shown below:

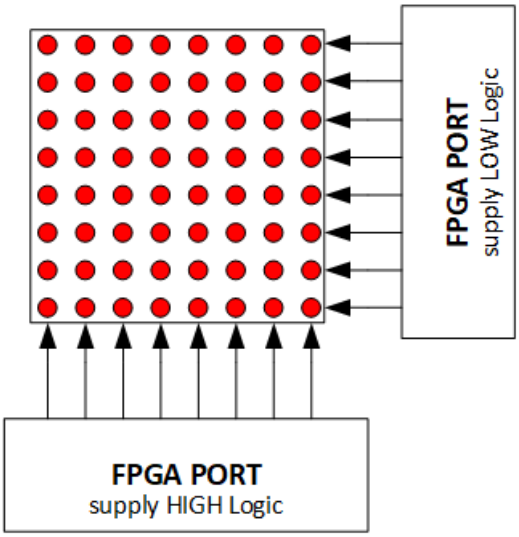


Fig.6: Dot-matrix connection to FPGA port

BUT…. there is an exception. For adequate brightness, the dot-matrix will be powered from a +5V supply, but the maximum programmable voltage on the output of the FPGA is +3.3V. Therefore, adequate buffer drivers, usually Level Translators, MUST be placed between the FPGA output ports and the pins of the LED dot-matrix display. HIGH-logic output level translator is needed for the column connection while a LOW-logic output level translator is needed for the ROW connections.

In this design, the LSI digital chip **74HC245** [3] is used as the HIGH-logic translator, while the **ULN2803** [4] Darlington array chip is used as the LOW-logic translator.

With the inclusion of these **ICs**, the connection is modified as shown below:

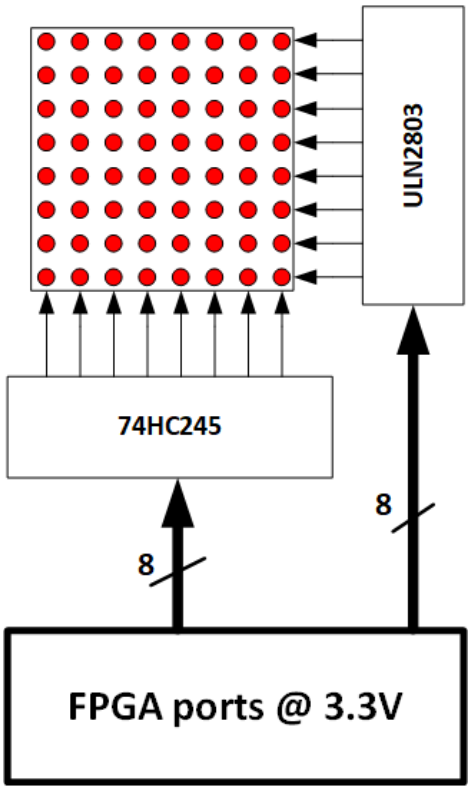


Fig.7: Dot-matrix buffered connection to FPGA port

From the diagram above, the total number of ports needed from the FPGA to facilitate this connection is 8. Eventually, we decided to go further by cascading 4 led dot-matrix display units to accommodate a full moving text while also being able to display single alphanumeric characters.

A cascade of 4 dot-matrix displays would require 32 port pins from the FPGA for the COLUMN control and an additional 8 port pins for the ROW control as shown in **Fig: 8** below.

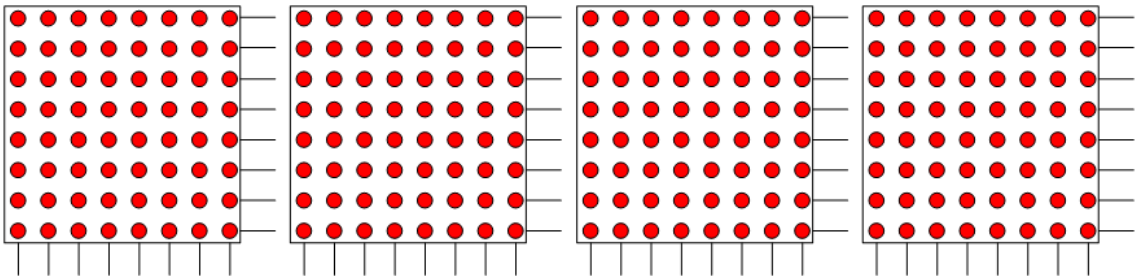


Fig.8: Cascade of 4 Dot-matrix modules

But the maximum digital port pins available from the Digilent FPGA demo-board is 32. Therefore, an IO expansion MUST be performed on the FPGA port pins designated to control the COLUMNS of the dot-matrix display.

IO EXPANSION

The IO expansion was performed on the FPGA port pins using the **74HC595** LSI serial to parallel shift register chip. The **74HC595** is an 8-bit serial to parallel shift register. Multiple of these **ICs** can be daisy-chained to form a wider-bit serial to parallel shift register. In this case, 4 of these **ICs** cascade to form a 32-bit wide serial to parallel shift register. Below in **FIG: 9** is a functional diagram of the **74HC595** 8-bit serial to parallel shift register.

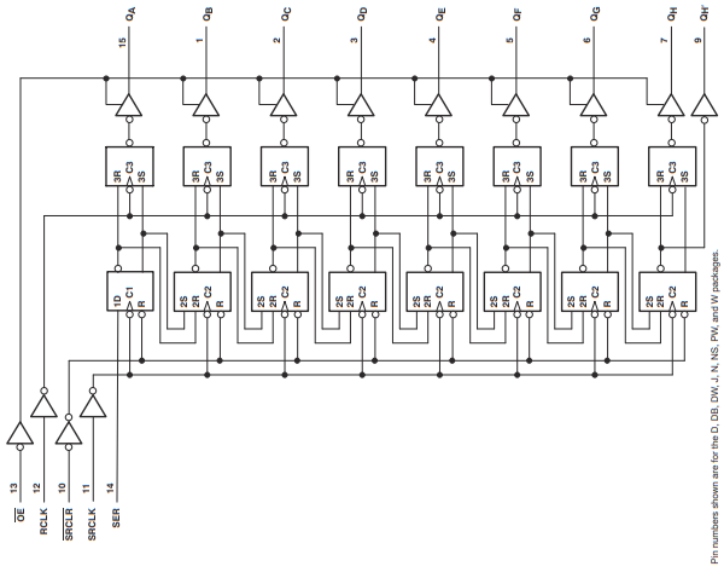


Fig.9: Functional Block Diagram of the 74HC595

COMPLETE CIRCUIT IMPLEMENTATION

The final circuit implementation using the **74HC595** 8-bit serial to parallel shift register to expand the FPGA port pins is shown below in **FIG:10.**

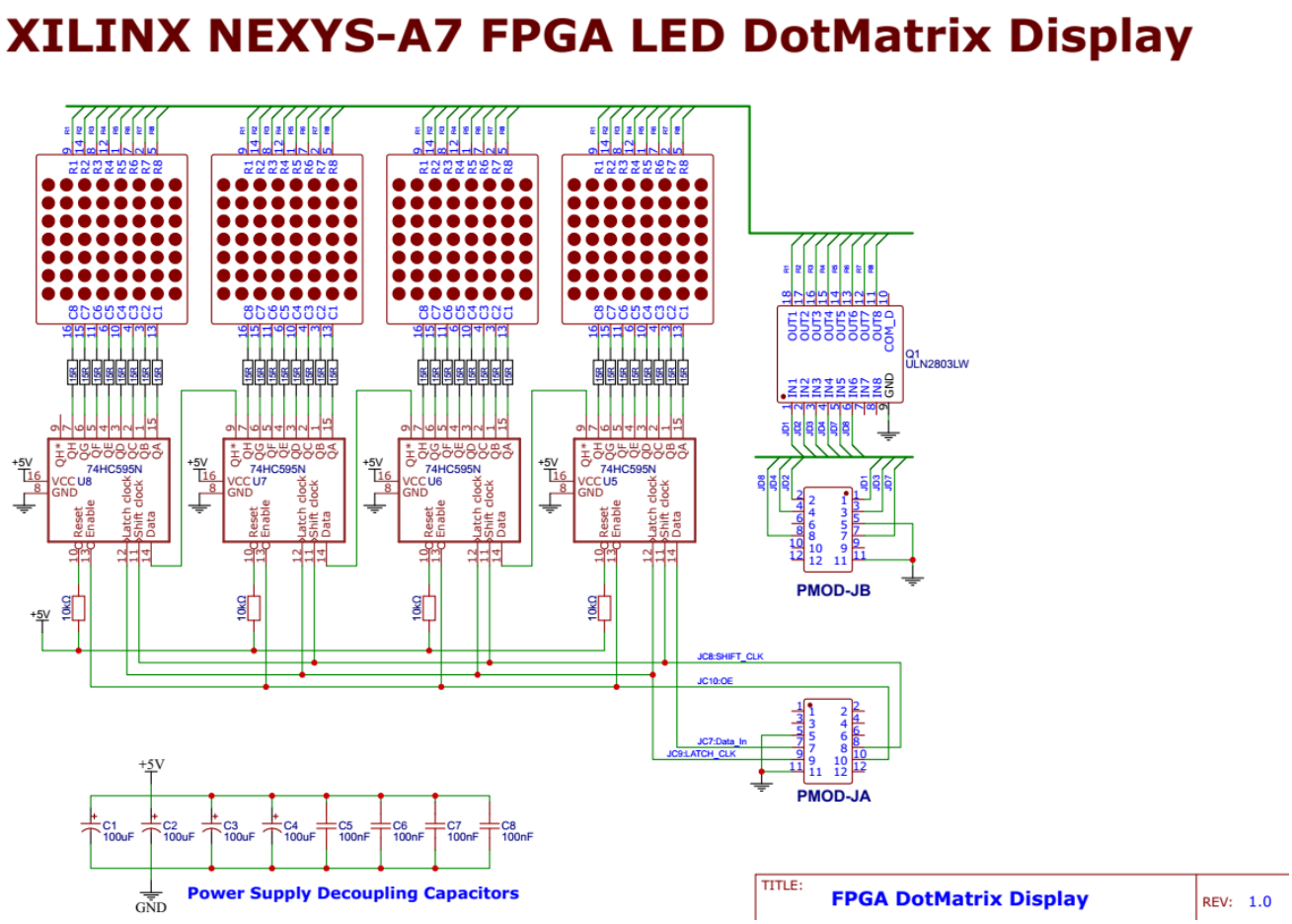


Fig.10: Complete Circuit Design

1.3.2. LOGIC IMPLEMENTATION USING VHDL

A detailed explanation of the logic implementation of the project design specifications using VHDL is given below. Our approach to the logic implementation using VHDL is clarified by itemizing each VHDL digital construct employed in the logic design.

Apart from the default project requirement, we went further to implement a moving text display using 4 LED dot-matrix modules, therefore, we implemented two architectures. To manage which architecture is running at any time, the two implemented architectures is instantiated from another VHDL file called the Top Entity. In the **TOP ENTITY**, selection between the two architectures is implemented through a multiplexer triggered by an external switch.

Below in **FIG:11**, is the block diagram of the **TOP ENTITY** design and an **RTL view** of the same implementation.

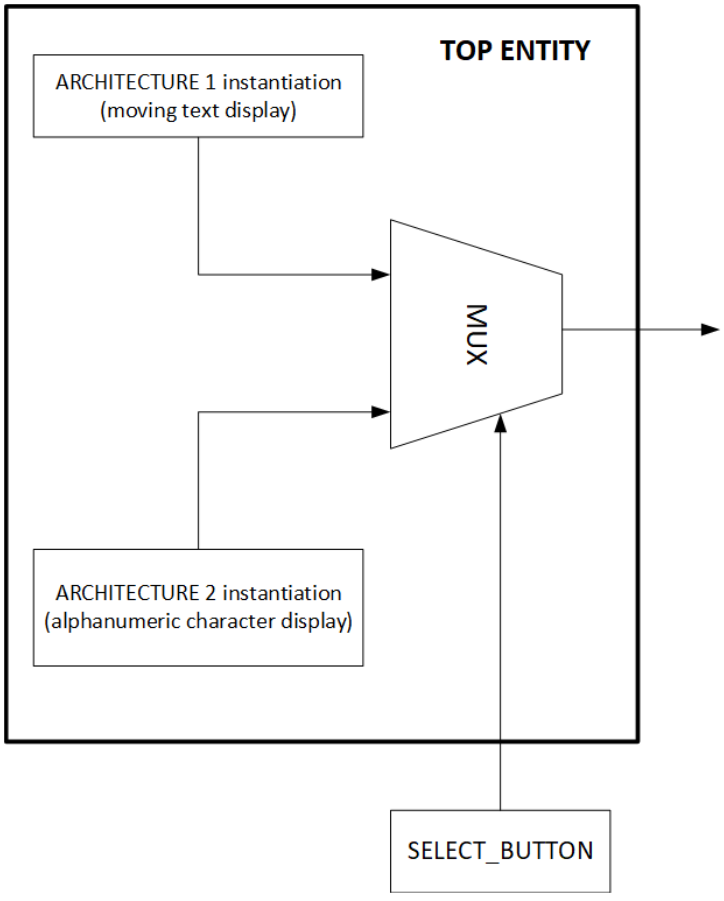


Fig.11a: TOP Entity Block Diagram

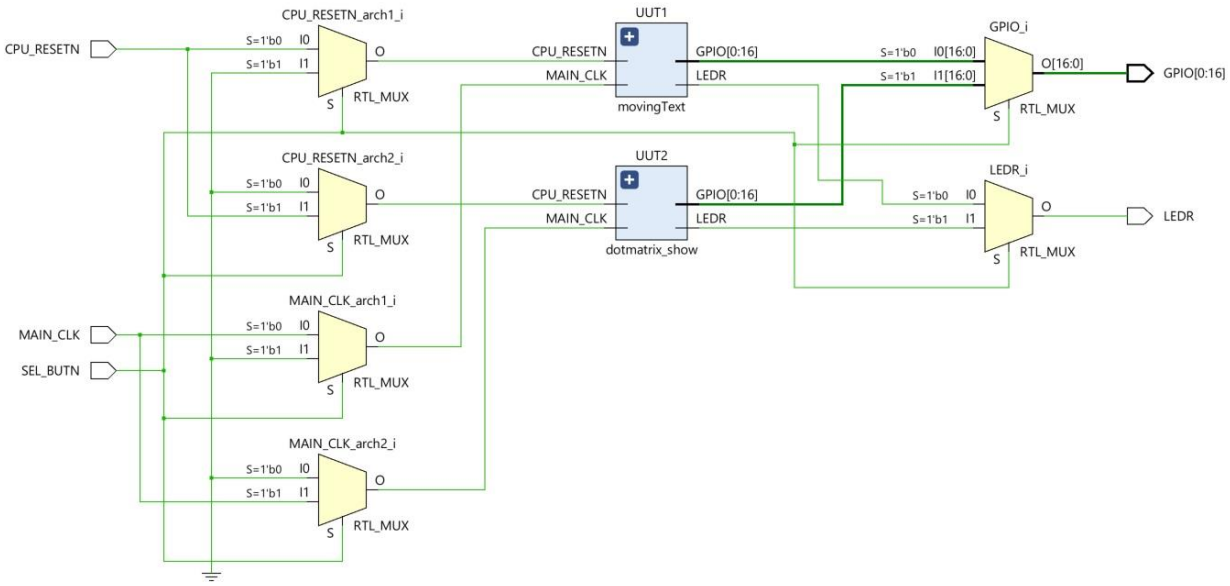


Fig.11b: RTL view of TOP Entity

TOP ENTITY VHDL DIGITAL CONSTRUCT BLOCKS

Each VHDL block implementation in this TOP entity module as shown in FIG.11a is described below.

- **MUX (multiplexer)**

The MUX block as shown in the block diagram is a simple 2-to-1 multiplexer. This is easily implemented in VHDL with a one-liner **when-else** statement and a process controlled by the signal assigned by the **when-else** statement. This is shown in the VHDL code snippet below in FIG:12

```
75
76  arch_variable <= ARCH2 when SEL_BUTN = '1' else ARCH1;
77
78  process(MAIN_CLK)
79  begin
80      case arch_variable is
81          when ARCH1 =>
82              MAIN_CLK_arch1 <= MAIN_CLK;
83              CPU_RESETN_arch1 <= CPU_RESETN;
84              LEDR <= LEDR_arch1;
85              GPIO <= GPIO_arch1;
86
87              MAIN_CLK_arch2 <= '0';
88              CPU_RESETN_arch2 <= '0';
89
90          when ARCH2 =>
91              MAIN_CLK_arch2 <= MAIN_CLK;
92              CPU_RESETN_arch2 <= CPU_RESETN;
93              LEDR <= LEDR_arch2;
94              GPIO <= GPIO_arch2;
95
96              MAIN_CLK_arch1 <= '0';
97              CPU_RESETN_arch1 <= '0';
98      end case;
99  end process;
100
```

Fig.12: VHDL Implementation of MUX Block

- **ARCHITECTURE-1 (Moving Text Display)**

The ARCITECTURE-1 block is the complete **VHDL** implementation of the moving text display. The decomposition of this block to its constituent blocks is shown below in **FIG:13**.

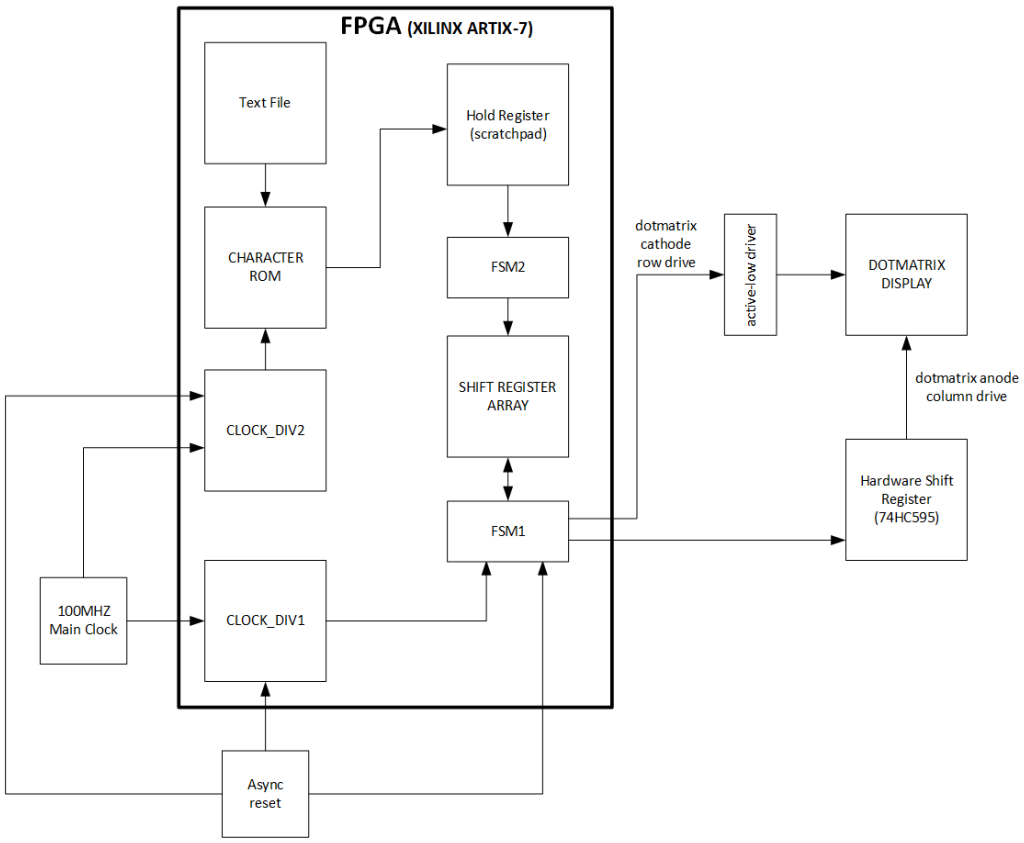


Fig.13: Block Structure of Architecture-1

Each composing block is described next.

TextFile

The text displayed on the dot-matrix module is hard coded into the VHDL file. This was implemented in the VHDL design by a simple array of characters (string).

The hard coded text is: ***"Welcome to 7067CEM: FPGA-Based Digital System Design "***

The code snippet where this is implemented in the VHDL file is shown below.

```
25
26  constant message_info: string(1 to 63)
27              := "Welcome to 7067CEM: FPGA-Based Digital System Design.          ";
28
```

Fig.14: VHDL Hard Coding of Text Object

Character ROM

The character ROM (read only memory) holds the font pattern for each character to be displayed on the LED dot-matrix. This ROM is implemented as an array or arrays in the VHDL file. But first, how are these font patterns obtained?

A special dot-matrix font character generator software is used. The software used is the **LED Dot-Matrix** Font Creator software from **Xantorohara.[5]** The software allows the user to generate binary data from either existing system fonts or newly created fonts. In our case, most system fonts will not fit into an 8x8 matrix space, hence, designing our own fonts from scratch was necessary.

The GLCD Font Creator software easily facilitates this. Below in **FIG:15** is shown how binary data is obtained from font pattern on an LED-matrix display taking the alphanumeric character "A" as an example.

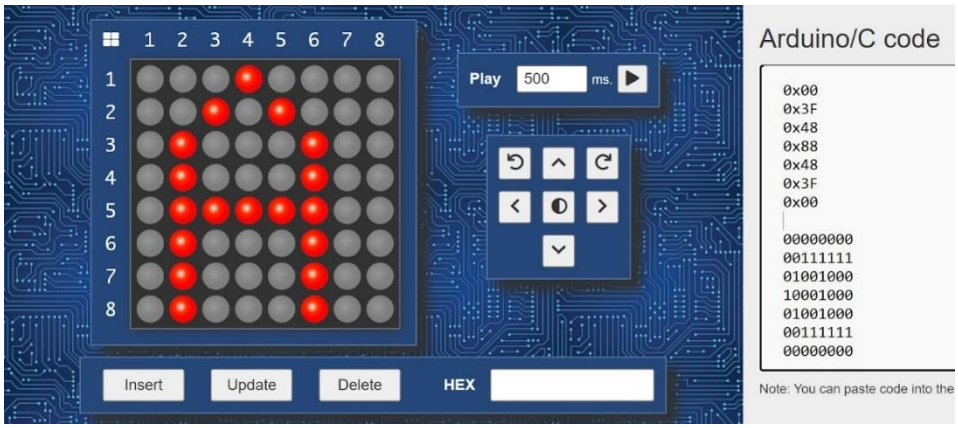


Fig.15: Binary Coding of Font Pattern

The binary data representing this font is obtained column-wise from left to right and from top (MSB) to bottom (LSB). The dark grey dots represent LEDs that are OFF (0) while the red dots represent LEDs that are ON (1). Therefore, for the first column, the binary data from top (MSB) to bottom (LSB) is "00000000". For the second column, the binary data from top (MSB) to bottom (LSB) is "00111111". Following the same encoding style, the binary data for the remaining columns are: "01001000", "10001000", "01001000", "00111111", "00000000".

The complete binary data for character "A" in hexadecimal according to this font design is:

A => (0x00, 0x3F, 0x48, 0x88, 0x48, 0x3F, 0x00)

As can be observed, the font has a matrix dimension of **7x8** (7 COLUMNS x 8 ROWS).

This process was applied to all the printable characters of the **ascii** table to obtain the binary data of all the said characters. These were hard coded into the VHDL file in a ROM implemented as an array of arrays. There are 95 printable ascii characters from the **space character** (“ ”) to **Tilde** (“~”). Therefore, the ROM is implemented as an array of 95 elements with each element also an array of 7 byte-wide elements. The code snippet where this is implemented in the VHDL file is shown below.

```
77
78  type sevenby8 is array(0 to 6) of std_logic_vector(7 downto 0);
79  type newFont is array(0 to 94) of sevenby8;
80
81  constant LedFont: newFont := (   --dotmatrix fonts used
82                                  (x"00", x"00", x"00", x"00", x"00", x"00", x"00"),  -- Code for char
83                                  (x"00", x"fd", x"00", x"00", x"00", x"00", x"00"),  -- Code for char !
84                                  (x"00", x"00", x"e0", x"00", x"e0", x"00", x"00"),  -- Code for char "
85                                  (x"00", x"24", x"7E", x"24", x"7E", x"24", x"00"),  -- Code for char #
86                                  (x"00", x"24", x"52", x"FF", x"4A", x"24", x"00"),  -- Code for char $
87                                  (x"00", x"62", x"04", x"08", x"10", x"23", x"00"),  -- Code for char %
88                                  (x"00", x"36", x"49", x"4D", x"32", x"05", x"00"),  -- Code for char &
89                                  (x"00", x"00", x"00", x"00", x"90", x"E0", x"00"),  -- Code for char '
90                                  (x"00", x"00", x"00", x"00", x"7e", x"81", x"00"),  -- Code for char (
91                                  (x"00", x"81", x"7e", x"00", x"00", x"00", x"00"),  -- Code for char )
92                                  (x"00", x"24", x"18", x"7E", x"18", x"24", x"00"),  -- Code for char *
93                                  (x"00", x"10", x"10", x"7E", x"10", x"10", x"00"),  -- Code for char +
94                                  (x"00", x"00", x"05", x"06", x"00", x"00", x"00"),  -- Code for char ,
95                                  (x"00", x"10", x"10", x"10", x"10", x"10", x"00"),  -- Code for char -
96                                  (x"00", x"00", x"03", x"03", x"00", x"00", x"00"),  -- Code for char .
97                                  (x"00", x"03", x"04", x"18", x"20", x"C0", x"00"),  -- Code for char /
98                                  (x"00", x"7E", x"85", x"99", x"A1", x"7E", x"00"),  -- Code for char 0
99                                  (x"00", x"00", x"41", x"FF", x"01", x"00", x"00"),  -- Code for char 1
100                                 (x"00", x"43", x"85", x"89", x"91", x"61", x"00"),  -- Code for char 2
101                                 (x"00", x"42", x"81", x"91", x"91", x"6E", x"00"),  -- Code for char 3
102                                 (x"00", x"0E", x"32", x"42", x"FF", x"02", x"00"),  -- Code for char 4
103                                 (x"00", x"F2", x"91", x"91", x"91", x"8E", x"00"),  -- Code for char 5
104                                 (x"00", x"7E", x"91", x"91", x"91", x"0E", x"00"),  -- Code for char 6
105                                 (x"00", x"40", x"87", x"88", x"90", x"e0", x"00"),  -- Code for char 7
106                                 (x"00", x"6E", x"91", x"91", x"91", x"6E", x"00"),  -- Code for char 8
107                                 (x"00", x"70", x"89", x"89", x"89", x"7E", x"00"),  -- Code for char 9
```

Fig.16: VHDL Implementation of the Character ROM Block

CLOCK DIV1 and CLOCK DIV2

The CLOCK_DV1 and CLOCK_DV2 blocks are used to generate slower clock signals from the main clock source 100MHZ. These blocks are implemented as counters dividing the main clock source 100MHz to produce the needed frequencies. The code snippet where this is implemented in the VHDL file is shown below..

```
368
369 SCAN_CLK_PROC: process(MAIN_CLK, cpu_rst) --generate the 74HC595 shift in frequency
370     constant count_range: integer range 0 to FREQ := FREQ/25e6;
371     variable counter: integer range 0 to count_range := 0;
372
373     begin
374         if cpu_rst = reset_logic then
375             SCAN_CLK <= '0';
376             counter := 0;
377         elsif rising_edge(MAIN_CLK) then
378             if counter < count_range-1 then
379                 SCAN_CLK <= '0';
380                 counter := counter + 1;
381             else
382                 SCAN_CLK <= '1';
383                 counter := 0;
384             end if;
385         end if;
386     end process;
```

Fig.17a: VHDL Implementation of CLOCK_DIV1 Block

```
224
225 CHAR_CHANGE_CLK_PROC: process(MAIN_CLK, cpu_rst)
226     constant count_range: integer := 5e6;
227     variable drtcount: integer range 0 to FREQ := 0;
228     begin
229         if cpu_rst = reset_logic then
230             drtcount := 0;
231             CHAR_CLK <= '0';
232         elsif rising_edge(MAIN_CLK) then
233             if drtcount < count_range-1 then
234                 drtcount := drtcount + 1;
235             else
236                 drtcount := 0;
237                 CHAR_CLK <= not CHAR_CLK;
238             end if;
239         end if;
240     end process;
```

Fig.17b: VHDL Implementation of CLOCK_DIV2 Block

HOLD REGISTER (scratchpad)

The next block in this architecture is the HOLD register (scratchpad). This is a temporary storage area for the pre-fetched font binary data from the character ROM for the character to be displayed on the LED dot-matrix at a particular time. This hold-register is filled with the font data in the same way the character will look like on the physical display. Then each column of this register (starting from left to right) is selected and shifted to the SHIFT REGISTER ARRAY block. The code snippet where this is implemented in the VHDL file is shown below in **FIG:18**.

```
219
220     --copy pattern to scratchpad
221     LEVEL1: for i in 0 to scratchPad'high generate
222     MAPPING:    scratchPad(i) <= LedFont(char_font_select)(i);
223                 end generate LEVEL1;
```

Fig.18: VHDL Implementation of HOLD Register (scratchpad) Block

SHIFT REGISTER ARRAY

The shift register array is a size replica of the LED dot-matrix display area. The dot-matrix display area is 32x8 (32 COLUMNS x 8 ROWS) in size, therefore, the shift register array is also set up as a 32x8 register. Also, since this a moving text is to be displayed on the dot-matrix, this array is implemented as a shift register. This approach makes displaying and scanning anything on the dot-matrix display very easy. The code snippets where this array is implemented in the VHDL file is shown below in **FIG:19**.

```
64
65  type screenAreaFormat is array(0 to 7) of std_logic_vector(DOTMATRIX_WIDTH-1 downto 0); --structure of the screen area
66  signal screenArea: screenAreaFormat := (others => (others =>'0'));
67
```

```
screenArea(0) <= screenArea(0)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(7);
screenArea(1) <= screenArea(1)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(6);
screenArea(2) <= screenArea(2)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(5);
screenArea(3) <= screenArea(3)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(4);
screenArea(4) <= screenArea(4)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(3);
screenArea(5) <= screenArea(5)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(2);
screenArea(6) <= screenArea(6)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(1);
screenArea(7) <= screenArea(7)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(0);
```

Fig.19: VHDL Implementation of the Shift Register Array Block

FSM1 and FSM2

These blocks are **FINITE STATE MACHINES** used in sequencing (driving) the entire implementation. **FSM1** is the finite state machine that drives the process of scanning the internally implemented shift register array to the external dot-matrix display serially through the IO expansion circuit comprising the cascaded 74HC595 ICs.

On the other hand, **FSM2** state machine drives the process of left shifting the entire display area and appending a new column of data from the scratchpad area. This produces the moving text effect observed on the external dot-matrix display. Below in **FIG:20** are the STATE TRANSITION DIAGRAMS of these Finite State Machines.

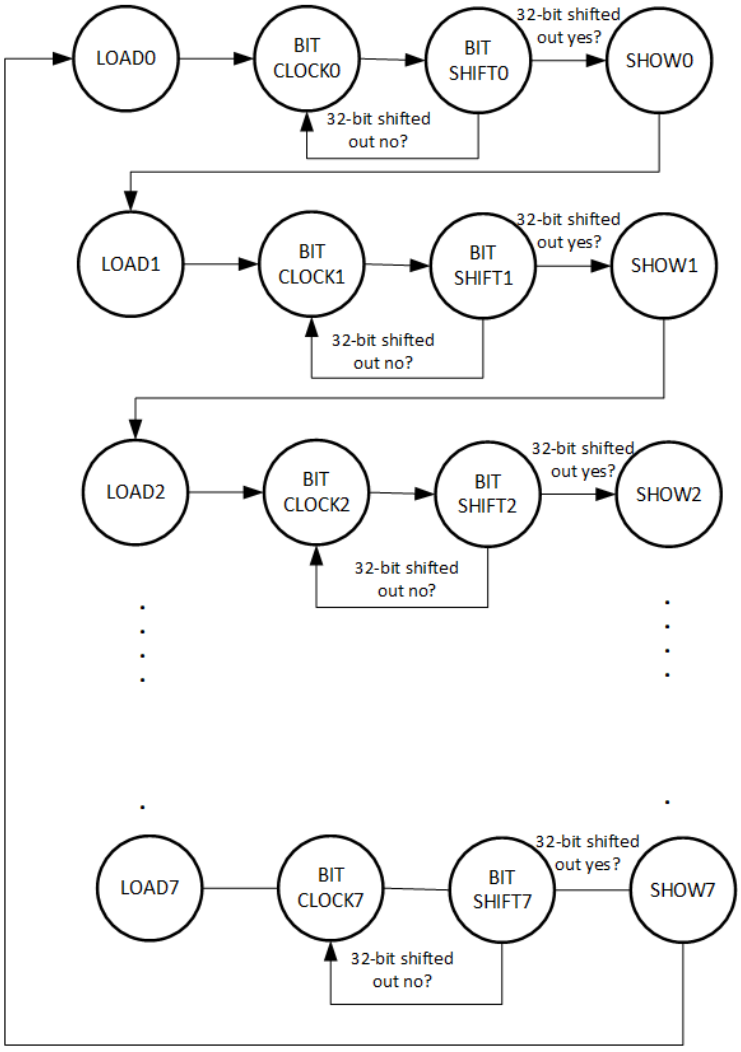


Fig.20a: State Transition Diagram of FSM1

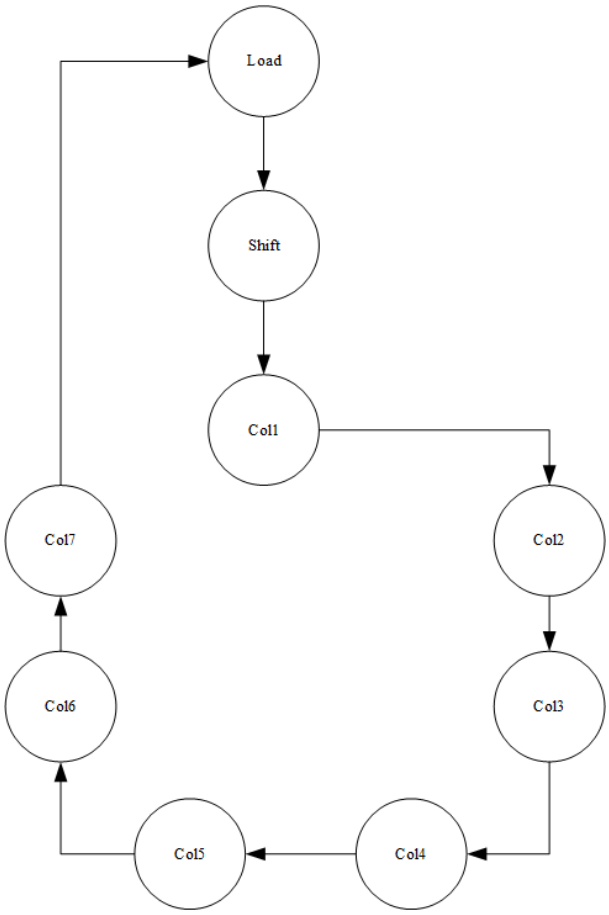


Fig.20b: State Transition Diagram of FSM2

The code snippets where these Finite State Machines are implemented in the VHDL file is shown below in **FIG:21**.

```
411         dummy <= "001";
412         case scanStateVariable is
413
414             when LOAD0 =>
415                 serial_clk <= '0';
416                 parallel_load <= '0';
417                 screenAreaLatch <= screenArea(0)(screenArea(0)'left-1 downto 0) & '0'; --copy the fi
418                 serial_data_in <= screenArea(0)(screenArea(0)'left); -- make the MSB immediately ava
419                 scanStateVariable <= BIT_CLOCK0;
420             when BIT_CLOCK0 =>
421                 if clock_count = 2 then
422                     scanStateVariable <= BIT_SHIFT0;
423                     clock_count := 0;
424                 else
425                     serial_clk <= not serial_clk;
426                     clock_count := clock_count + 1;
427                 end if;
428             when BIT_SHIFT0 =>
429                 if serial_data_count < DOTMATRIX_WIDTH-1 then
430                     serial_data_in <= screenAreaLatch(screenAreaLatch'left);
431                     screenAreaLatch <= screenAreaLatch(screenAreaLatch'left-1 downto 0) & '0';
432                     serial_data_count := serial_data_count + 1;
433                     scanStateVariable <= BIT_CLOCK0;
434                 else
435                     serial_data_count := 0;
436                     output_enable <= '1';
437                     row_driver <= (others => '0'); --turn off the row drivers
438                     scanStateVariable <= SHOW0;
439                 end if;
440             when SHOW0 =>
441                 parallel_load <= '1';
442                 output_enable <= '0';
443                 row_driver <= (row_driver'left => '1', others => '0');
444                 if scanCount < scanCountRange then
445                     scanCount := scanCount + 1;
446                 else
447                     scanCount := 0;
448                     scanStateVariable <= LOAD1;
449                 end if;
```

Fig.21a: VHDL Implementation of FSM1

```
258         case shiftStateVariable is
259
260             when LOAD =>
261                 if stringIndex < message_info'high then
262                     ascii_char <= message_info(stringIndex); --get each character in the string
263                     shiftStateVariable <= SHIFT; --go to next state
264                     stringIndex := stringIndex + 1;
265                 else
266                     stringIndex := 1;   --restart the string
267                 end if;
268
269             when SHIFT =>
270                 if shiftSpeedCounter < shiftSpeedCountRange-1 then
271                     shiftSpeedCounter := shiftSpeedCounter + 1;
272                 else
273                     shiftSpeedCounter := 0;
274
275                     case shiftedColumnState is
276                         when COL1 =>
277                                 screenArea(0) <= screenArea(0)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(7);
278                                 screenArea(1) <= screenArea(1)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(6);
279                                 screenArea(2) <= screenArea(2)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(5);
280                                 screenArea(3) <= screenArea(3)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(4);
281                                 screenArea(4) <= screenArea(4)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(3);
282                                 screenArea(5) <= screenArea(5)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(2);
283                                 screenArea(6) <= screenArea(6)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(1);
284                                 screenArea(7) <= screenArea(7)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(0)(0);
285
286                                 shiftedColumnState <= COL2;
287
288                         when COL2 =>
289                                 screenArea(0) <= screenArea(0)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(7);
290                                 screenArea(1) <= screenArea(1)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(6);
291                                 screenArea(2) <= screenArea(2)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(5);
292                                 screenArea(3) <= screenArea(3)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(4);
293                                 screenArea(4) <= screenArea(4)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(3);
294                                 screenArea(5) <= screenArea(5)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(2);
295                                 screenArea(6) <= screenArea(6)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(1);
296                                 screenArea(7) <= screenArea(7)(DOTMATRIX_WIDTH-2 downto 0) & scratchPad(1)(0);
297
298                                 shiftedColumnState <= COL3;
299
```

Fig.21b: VHDL Implementation of FSM2

- **ARCHITECTURE-2 (Alphanumeric Character Display)**

The ARCITECTURE-2 block is the complete **VHDL** implementation of the alphanumeric character display. The decomposition of this block to its constituent blocks is shown below in **FIG:22**.

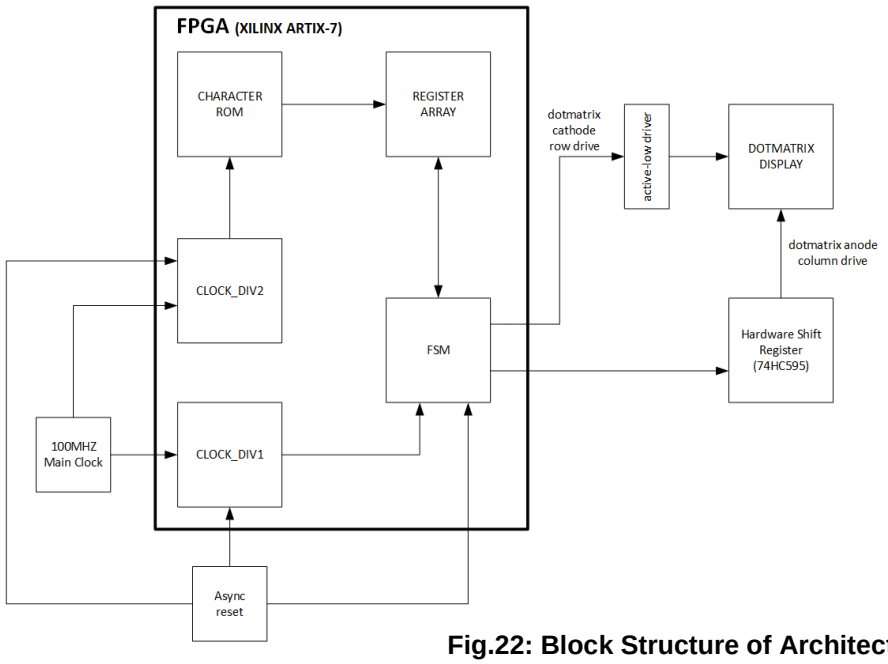


Fig.22: Block Structure of Architecture-2

The composing blocks are same ones previously described in architecture-1 with the exception of the **RERGISTER ARRAY** block.

REGISTER ARRAY

The register array is a size replica of the LED dot-matrix display area implemented as a STORAGE REGISTER ONLY and not a SHIFT REGISTER. This array is also 32x8 (32 COLUMNS x 8 ROWS) in size. The binary font data of the alphanumeric character to be displayed on the dot-matrix display is copied directly from the character ROM to the lower end of this array. CLOCK_DIV2 determines the frequency by which binary font data of new characters is copied to this register array. The finite state machine **FSM**, like finite state machine **FSM1** of the previous architecture, drives the process of scanning this register array to the external dot-matrix display serially through the IO expansion circuit comprising the cascaded 74HC595 ICs.

The code snippets showing the implementation of this Register Array in the VHDL file is shown below in **FIG:23**.

```
177
178     LEVEL1: for i in 8 downto 2 generate
179     MAPPING:    screenArea(0)(i) <= LedFont(char_font_select)(8-i)(7);
180                 screenArea(1)(i) <= LedFont(char_font_select)(8-i)(6);
181                 screenArea(2)(i) <= LedFont(char_font_select)(8-i)(5);
182                 screenArea(3)(i) <= LedFont(char_font_select)(8-i)(4);
183                 screenArea(4)(i) <= LedFont(char_font_select)(8-i)(3);
184                 screenArea(5)(i) <= LedFont(char_font_select)(8-i)(2);
185                 screenArea(6)(i) <= LedFont(char_font_select)(8-i)(1);
186                 screenArea(7)(i) <= LedFont(char_font_select)(8-i)(0);
187             end generate LEVEL1;
188
```

Fig.23: VHDL Implementation of the Register Array Block

2. TEST BENCHES

FPGA circuit design using VHDL follows a unique approach. For us, these are:

- Breaking down the circuit into smaller blocks to be implemented with VHDL.
- Verifying each block for proper functionality using test benches.
- When all blocks are fully verified, the blocks are connected to form the complete design unit. The complete design unit is then finally verified for expected functionality also using a testbench.

The test bench result of each block described in the previous chapter is shown the figures below.

Fig.24: Main Clock

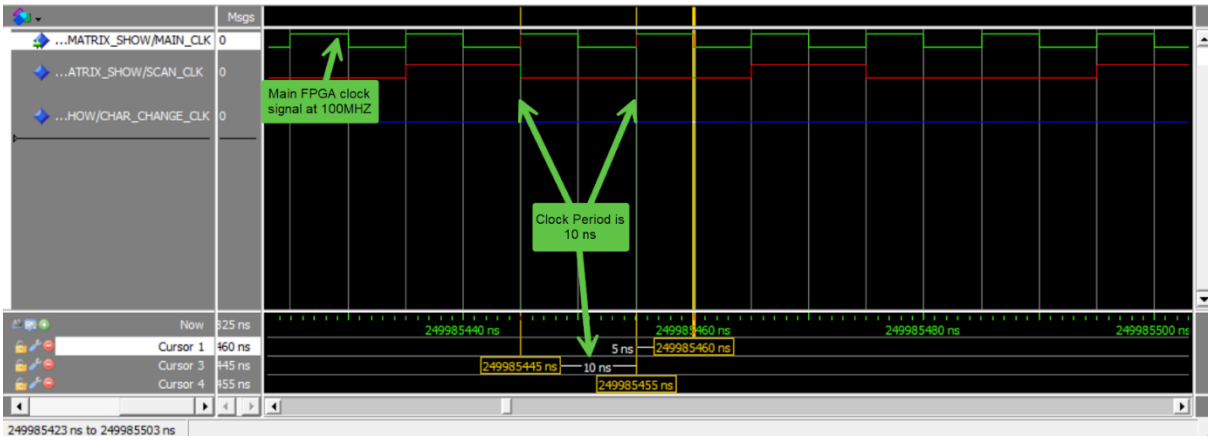


Fig.25: CLOCK_DIV1

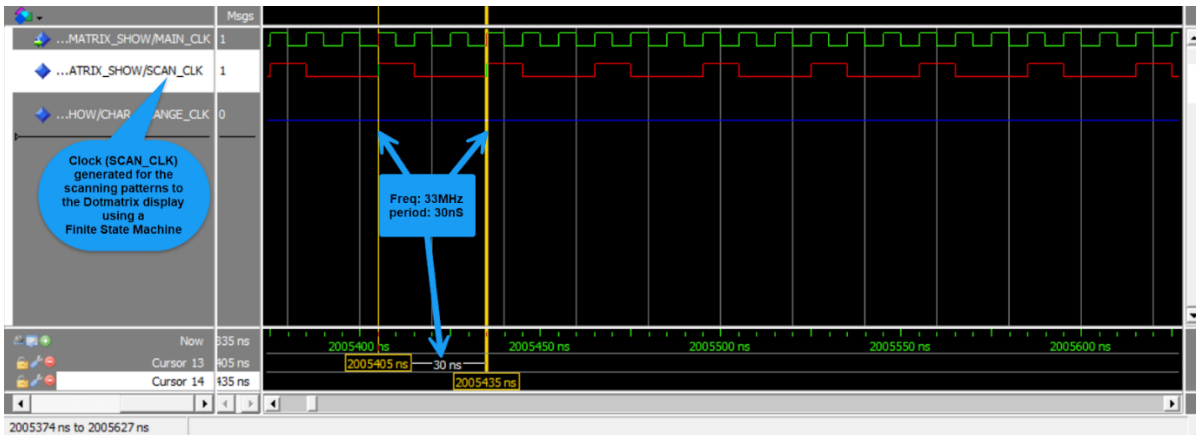


Fig.26: CLOCK_DIV2

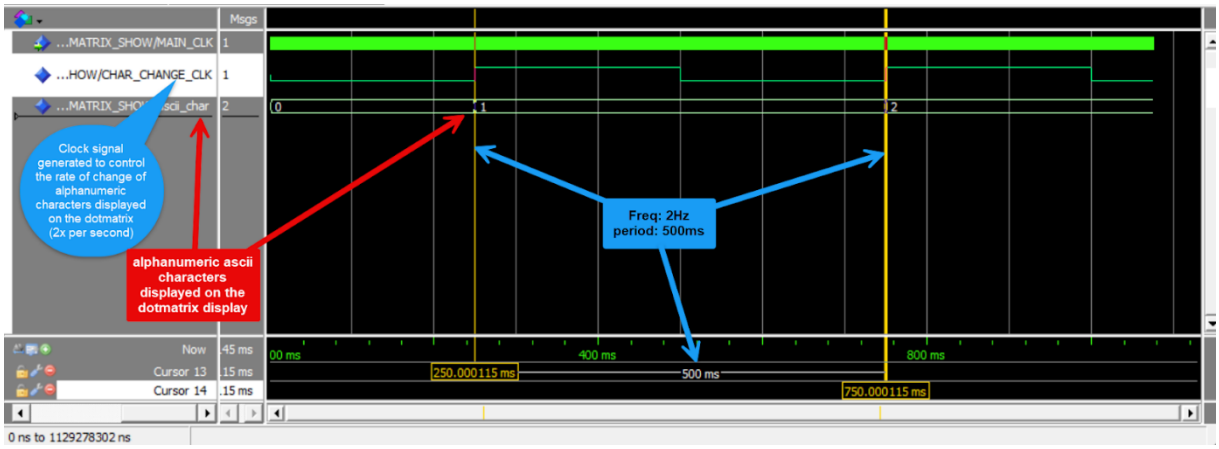


Fig.27: FSM/FSM1

***Note1:** This is the state variable used in the FSM implementation to scan the dot-matrix Display. "SHOW0....SHOW7" are the states with the longest time duration of approx. 2 ms. These states represent when each respective row from top to bottom (0 to 7) of the matrix is lit for a fraction of a second in the vertical scanning implementation. The FSM control clock (SCAN_CLK) was adjusted from the test bench to precisely produce a 2ms period between each "SHOW" for a well-lit display.

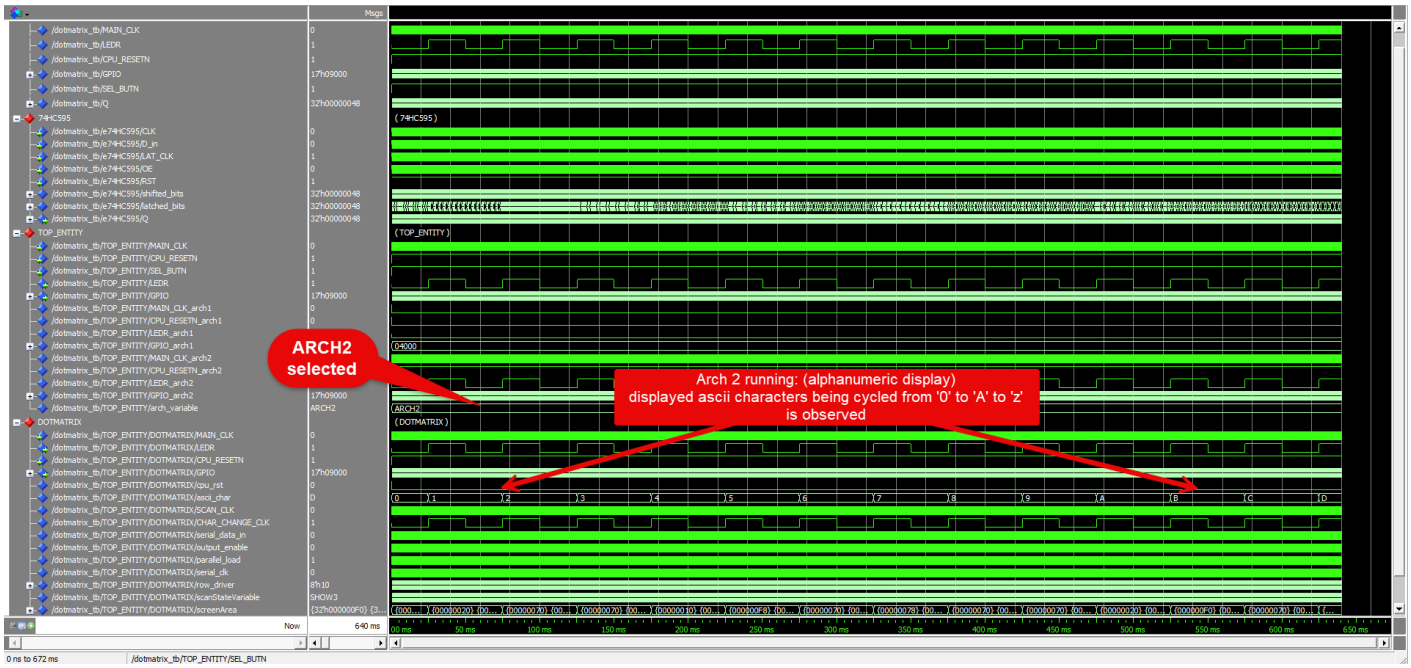
Fig.28: 74HC595

***Note2:** The 32-bit 74HC595 IO expansion was modelled in VHDL to verify the final circuit behaviour in actual operating mode.

Fig.29: COMLETE CIRCUIT IMPLEMENTATION: ARCH-1

ARCH1 selected and running. Part of the hard coded text ***("Welco")*** can be observed.

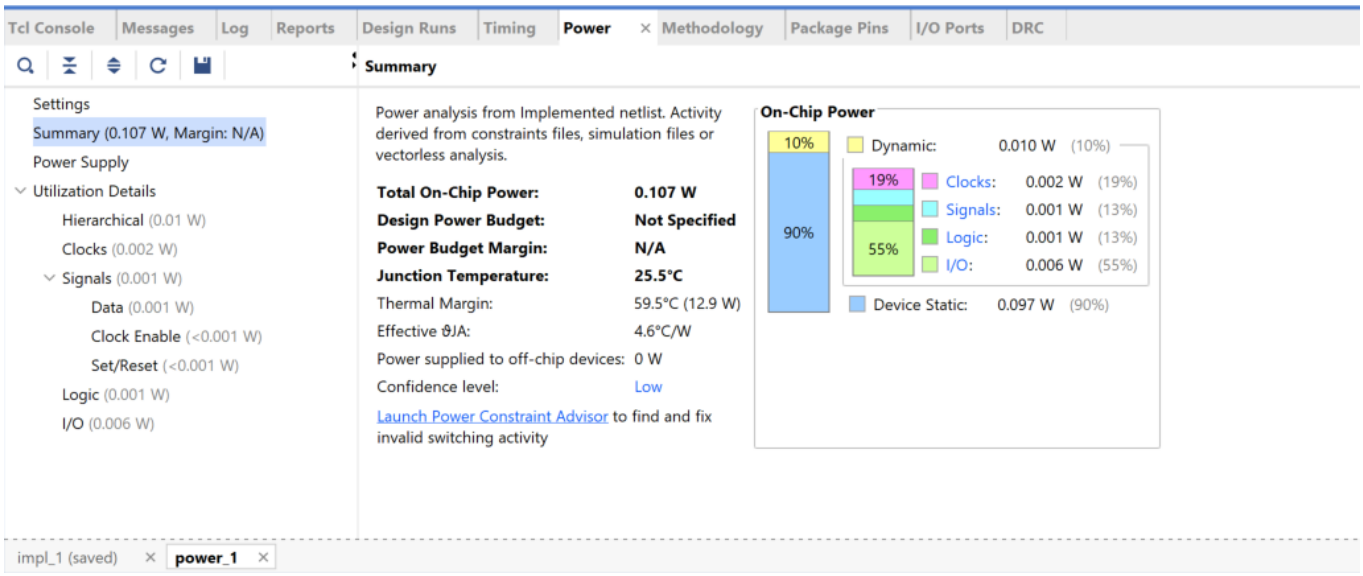
Fig.30: COMLETE CIRCUIT IMPLEMENTATION: ARCH-2

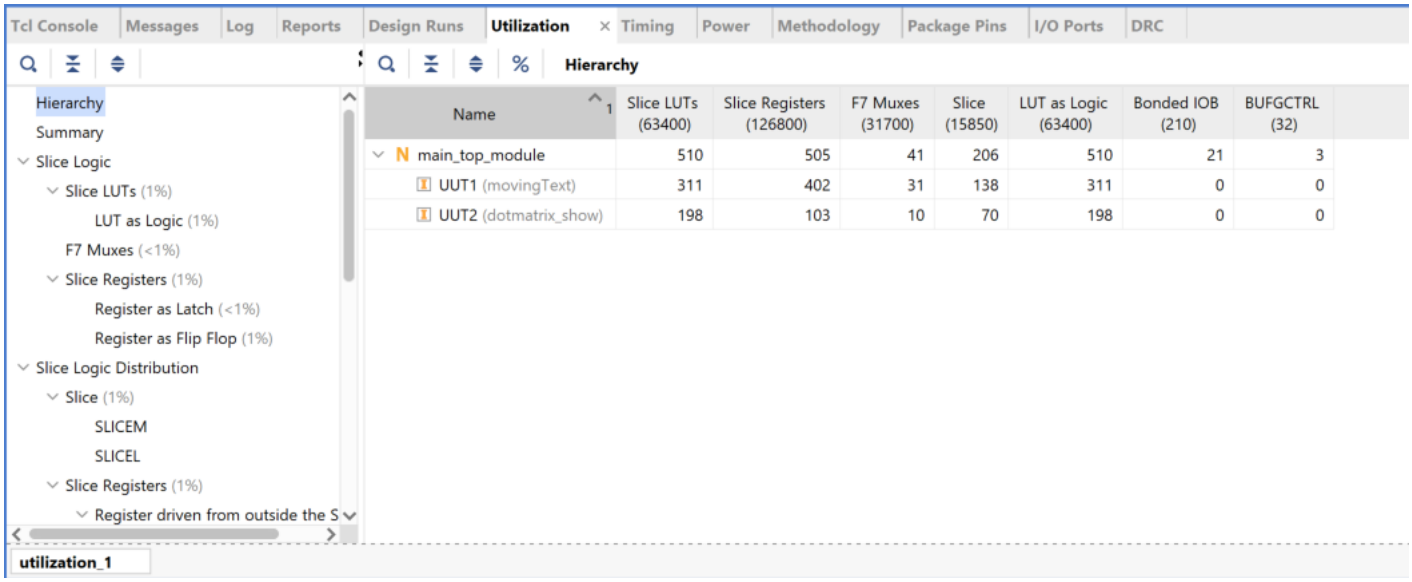


ARCH2 selected and running. Ascii characters cycled from ‘0’ to ‘A’ to ‘z’ can be observed.

3. POWER ANALYSIS AND OPTIMIZATION

The complete VHDL implementation was ran through the Power Analyzer tool in **VIVADO**. Below are shown the results. The Dynamic power is very low (10%). Although the static power is reported as 90%, it is because most of the time the device is in static mode. The static power consumption is 0.097W while the total chip power consumption is 0.107W.





The total LUTs used in the design is 63,400.

4. REFERENCES

1. **Digilent. (2019). Nexys A7™ FPGA Board Reference Manual.** *https://digilent.com/reference/_media/reference/programmable-logic/nexys-a7/nexys-a7_rm.pdf*
2. **Kingbright. (2012). 30mm (0.8 INCH) SINGLE COLOR DOT MATRIX DISPLAY**
3. **Texas Instruments. (2004). SH74HC595N 8-bit shift register datasheet.** *https://www.sparkfun.com/datasheets/IC/SN74HC595.pdf*
4. **ULN2803 High-voltage, High-current Darlington Arrays.** *https://www.sparkfun.com/datasheets/IC/uln2803a.pdf*
5. **Xantorohara. LED Matrix Editor** *https://xantorohara.github.io/led-matrix-editor/#0000000000000000|66667e7e66663c18*

***Please Note**: links to all design files have been provided in the appendix.*

5. APPENDIX

Below is the link to all design files pertaining to this project design: The link opens a folder in **one-drive** containing the following: ***project report, schematics, VHDL Source Files and the Demonstration Video***.

[7067CEM FPGA Digital Systems Design]