ECE 212 – Digital Circuits II

Lab 3 – Scrolling LED Matrix Display

This is a two-week lab

Revised February 14, 2019

1. Introduction

In this lab you will expand on the LED matrix controller that you designed in Lab 2 to create a scrolling text display. This controller will display a stored text message and have the ability to dynamically update a portion of that message based on user input. Your design will use block RAMs (BRAMs) in the FPGA as a frame buffer to store the desired content of the display and include support for reading from different locations in the BRAMs to create the scrolling effect. It will also modify what is displayed based on user input from a pushbutton.

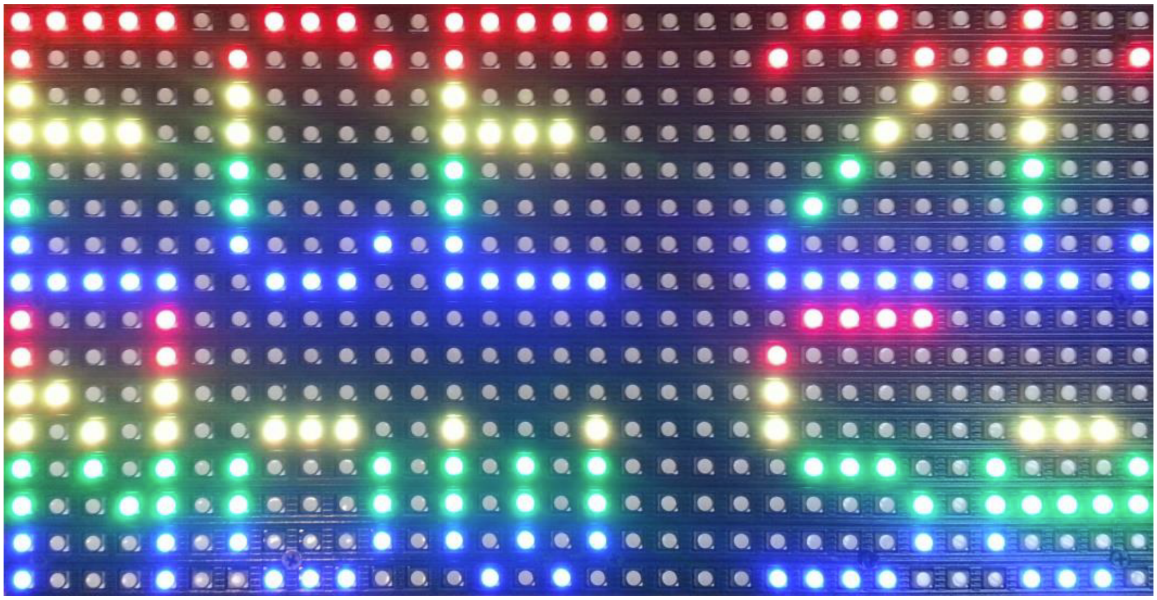

Figure 1 – Scrolling Display

2. The Design Task

The goal of this lab is to create a new pixel generator module that interfaces with the sequencer module designed in Lab 2 to create a scrolling display that shows two lines of alphanumeric messages, as shown in Figure 1. This display will function as a "now serving" sign that displays a message and a "next customer" number. The scrolling display should provide the following capabilities:

- Messages will be displayed using alphanumeric characters that are 5 pixels wide and 8 pixels high, separated by a one-pixel spacing.
- The top 8 rows display a fixed alphanumeric message, such as "ECE 212" or something similar.

- The bottom 8 rows display the string “Now Serving” followed by a number (0-9) indicating the next person to be served. This “Now Serving” text is displayed in a fixed 4-color rainbow.
- Both lines of text should scroll across the screen smoothly. The text should scroll from right to left; it should begin with a blank display into which the message text shifts in from the right side of the matrix and shifts out of the left side until the display is blank again. This process should repeat after a short pause.
- The value of the displayed number can be incremented by pressing a button connected to the `NEXTPB` input. The number should increment exactly once for each button press.
- The color of the displayed number may be changed to help indicate the expected wait time. The color is set using three switches connected to the `NEXTCOLOR` input that set the red, green, and blue components of the displayed number.

Figure 2 shows the arrangement of the sequencer circuit and the pixel generator, which has been modified by adding a clock input, a pushbutton input `NEXTPB`, and a three-bit `NEXTCOLOR` input connected to three switches on the Nexys4 board.

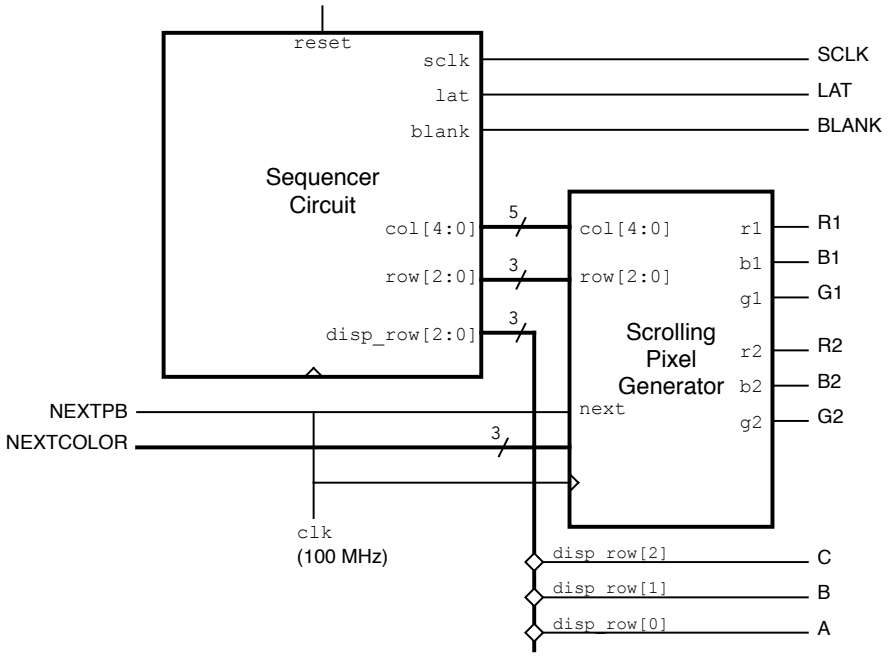

Figure 2 – LED Matrix Controller Organization

3. Background

3.1 Block RAMs

As we discussed in ECE 211, the core function of FPGAs is provided by an array of configurable logic blocks (CLBs) that can be connected by programmable interconnection network. Both CLBs and the interconnect network are programmed by writing bits into a configuration file (known as a bitstream file in Xilinx FPGAs). Each CLB contains lookup tables and flip-flops, and the flip-flops can be combined to

implemented small memories that can be used as data buffers. However, these are not very efficient.

To address the need of many designers for efficient memories, FPGA manufactures have embedded dedicated memories into the fabric of their FPGAs. Xilinx refers to these as block RAM (BRAMs). Conceptually a BRAM is a two-port memory two ports, as shown in Figure 3. A single BRAM can be used either as a single 36-kbit memory or two 18-kbit memories.

BRAMs can be configured in a number of different ways, including

- Number of ports – as shown in Figure 3, a BRAM can be operated with two completely independent ports which can read and write data independently of each other using separate address, data, clock, and control signals. Xilinx refers to this as a *True Dual Port (TDP)* configuration. They can also be configured in a *Simple Dual Port (SDP)* configuration, which uses one port strictly for reading and another port strictly for writing, as shown in Figure 4.
- Word width and memory depth – a variety of width and depth combinations are supported are supported. For example, in SDP mode BRAMS can be configured as:
    - 36-Kbit: 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36, or 512 x 72
    - 18-Kbit: 16K x 1, 8K x 2 , 4K x 4, 2K x 9, 1K x 18 or 512 x 36
- Error correction – a small number of bits in each BRAM can be used to implement error-correcting memory.
- FIFOs – BRAMs can be configured to operate as first-in-first-out (FIFO) buffers. We will discuss these in class later in the semester.

FPGAs generally contain multiple BRAMs distributed through the FPGA fabric. For example, the Xilinx XC7A100T FPGA on the Nexys4 board contains 135 BRAMs.

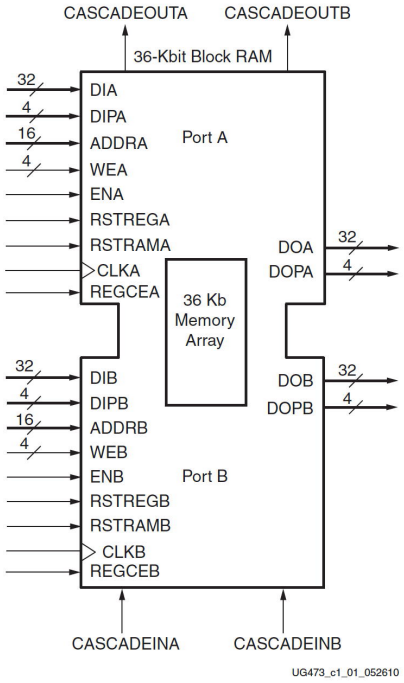

Figure 3 – Block RAM

When designing circuits for FPGAs in SystemVerilog BRAMs can be incorporated in two ways:

- SystemVerilog code can describe "RAM-like" behavior and the synthesis tool can *infer* a BRAM configuration that will implement that behavior. This approach has the advantage that it may be "portable" between different FPGA devices with different characteristics. On the other hand, it can be difficult to use some features of a BRAM effectively.
- SystemVerilog code can *instantiate* a parameterized module that represents configuration in a way that can be used by the synthesis tool. Different parameter values can be set to select configuration and memory features. In Vivado, BRAMs can be specified and configured using modules named `BRAM_SINGLE_MACRO`, `BRAM_SDP_MACRO`, `BRAM_TDP_MACRO`. These can be instantiated using different parameters to configure different features of the BRAM. One particularly useful set of parameters allows a RAM to be initialized with values when the FPGA is configured; we will use this feature extensively in this lab.

3.2 Using BRAMs as Frame Buffers

In Lab 2 we built a pixel generator circuit that used the row and column addresses provided by the sequence generator to "turn on" LEDs at particular locations. A more general approach to implementing a pixel generator is to incorporate a memory that stores color values for each pixel in a display and is addressed by the row and column values. This memory is often called a *frame buffer*. Frame buffers are usually constructed with two ports – one which reads pixel data for the hardware interface, and one which is used to write new contents and is often connected to a computer system.

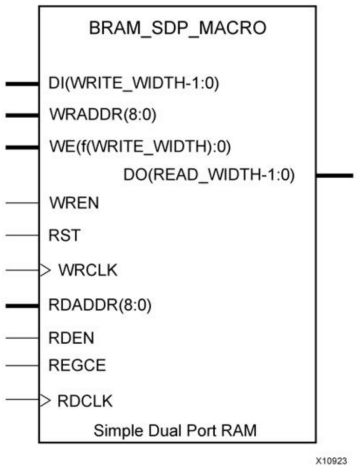

Figure 4 – Simple Dual-Port BRAM Configuration

In this lab we will use BRAMs to implement two frame buffers for the upper and lower half-panels of the LED matrix. Recall that each half-panel is an array of 8 x 32 pixels, each of which has a red, green, and blue component. Therefore, each frame buffer would contain 256 3-bit words. However, since the the closest word width supported by BRAMs is 4 bits, we will use a 256x4 memory instead. The address for a specific pixel in the BRAM is provided by concatenating the column and row address of the pixel, as shown in Figure 5.

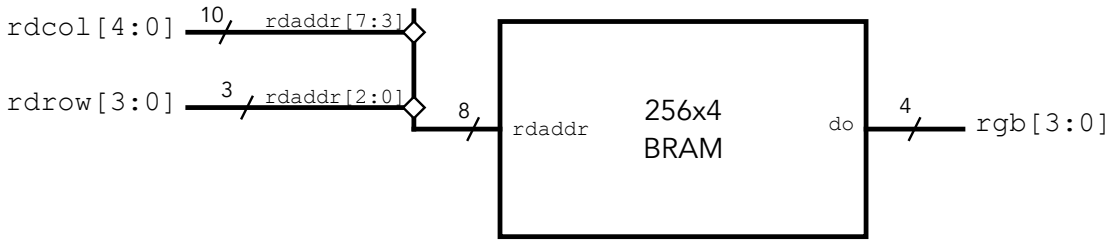

Figure 5 – Frame Buffer for the LED Matrix

We can display characters in this frame buffer by writing 4-bit RGB values in locations corresponding to different row and column locations. For example, Figure 6 shows an example of how a red "1" character would be stored in the BRAM if it was placed starting in column 0 of the frame buffer. When reading the memory to display the digit, a single 4-bit chunk would be read at a time. For example, reading address `0b010_000` (highlighted in yellow) would output `0x4` while reading address `0b100_010` (highlighted in purple) would output `0x0`. If a "1" digit was to be written to this location in memory, `0x00000000` would be written to address `0b000`, `0x40000040` would be written to address `0b001`, etc. Longer messages can be constructed by placing data in additional columns to the right.

	Column				
Row	000	001	010	011	100
000			0x4		
001		0x4	0x4		
010			0x4		
011			0x4		
100			0x4		
101			0x4		
110			0x4		
111		0x4	0x4	0x4	

Figure 6 – Row and Column Example (All unspecified values are 0x0)

3.3 Templates for Block RAM Instantiation

Vivado provides a set of templates of module instantiations for common BRAM configurations. These templates are just blocks of sample code (with comments containing instructions) that can be copied and pasted into your own modules and customized by changing parameter values..

You can find macros to instantiate common BRAM configurations by going to the `Templates` tab in Vivado (also `Language Templates` item or by going to `Tools->Language Templates` in the menu) and navigating to `Verilog->Device Macro Instantiation->Artix-7->RAM`. The Simple Dual Port

RAM (BRAM_SDP_MACRO) is a good fit for this lab as it provides a separate read and write port, each of which can be different widths. The provided template instantiates a parameterized module with two main pieces: the module parameters and the module inputs and outputs.

BRAM Parameter Overview

- BRAM_SIZE, DEVICE, DO_REG, INIT_FILE, SIM_COLLISION_CHECK, SRVAL, INIT, and WRITE_MODE can all be left with their default values.
- WRITE_WIDTH – bit width entry for write port.
- READ_WIDTH – bit width of entry for read port.
- INIT_xx lines – the template contains a sequence of these parameters INIT_00, INIT_01, etc. that allow you to specify the initial contents as a sequence of 256-bit constants. Since these parameters have a default value of zero you can remove these lines for regions of the RAM that you want initialized to zero. We will use these parameters to initialize BRAMS with the text of our messages as described later in this document.
- INITP_xx lines – these parameters specify initial contents of bits used to store parity for error correcting codes. Since we are not using this feature we can remove these from the template.

BRAM I/O Overview

- RST – reset signal
- Read Port
  - DO – data output; updated to value at address RDADDR on clock edge
  - RDADDR – address of entry to read
  - RDCLK – clock used for read port
  - RDEN – enables read port (should likely always be 1)
  - REGCE – enables output register (should be set to 0 as not using output register)
- Write Port
  - DI – data input; written on clock edge if WE and WREN are both 1
  - WRADDR – address of entry to write
  - WRCLK – clock used for write port (should be same as clock for RDCLK)
  - WREN – enables write port (can likely always be 1 if use WE to control when to write)
  - WE – this input contains one bit for each byte in the DI input, allowing bytes to be selectively written into the specified address (in our case, we will always want the entire word to be written, so to write into memory all bits should be set to one).

3.4 Scrolling

If we use the BRAM as a frame buffer, then the result will be a fixed display starting with the leftmost part of the display in column zero. However, the LED matrix is only 32 pixels wide, which limits the display to only about 6 characters. Our goal is to use a scrolling display to allow users to view longer messages.

We can implement scrolling by using a BRAM to store the image of a longer message that is addressed by column and row but that stores many more columns than can be displayed on the LED Matrix. For example, we can use a 36-kbit BRAM to implement 1024 columns and 8 rows of 4-bit pixel values. We can add an *offset* to the column part of the address from the sequence generator to vary what is displayed on the LED matrix.

For example, at the start the offset will be zero and the LED matrix will display the leftmost 32 columns stored in the BRAM. A short time later, the offset is incremented which has the effect of shifting what is displayed on the LED matrix one column to the left. If we continue to increment the offset, we will see the entire message stored in the BRAM scrolling from right to left.

The rate at which scrolling occurs depends on how quickly the offset is incremented. You should choose a speed that allows the message to be easily read without having to wait too long for the whole message to be displayed. A likely good range is to shift the display by one column every 100-200 ms.

Figure 7 shows a configuration using a BRAM to store and display a message that is up to 1024 columns long. The read port accesses a total of 8,192 pixels, where each pixel consists of four bits (three to store the red, green, and blue pixels, and one unused). Each pixel value is accessed using a 13-bit address that is formed by the concatenation of a 10-bit column address and a 3-bit row address. It is up to you to design the hardware to generate the column address by adding the column address from the sequence generator to the column offset.

The write port of the BRAM in Figure 7 is configured to write 32-bit values; this is equivalent to writing an entire column and will be used as described in the next section. The `WE` input of the BRAM is 4 bits wide and enables the writing of each byte in the data input separately; since we want to write all four bytes when we write a column these inputs are tied together to a control signal `we_all`.

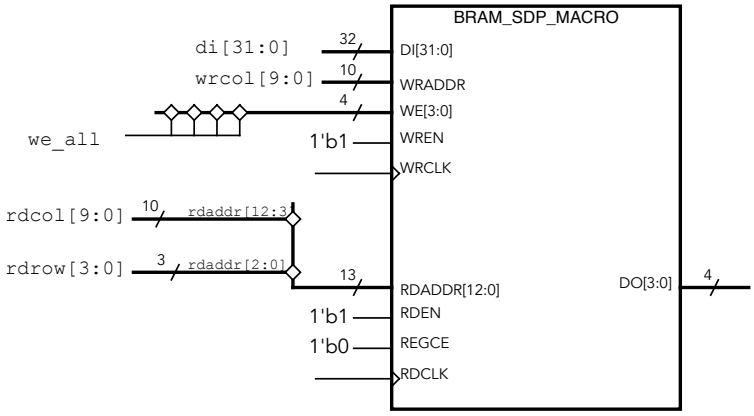

Figure 7 – Block RAM Configuration for Scrolling Display

3.5 Generating and Displaying the "Next Customer" Number

Each of the characters used in the scrolling display is 5×8 pixels in size and each character is separated by a single column. The memory patterns for the basic alphanumeric messages are provided in the appendix and can be configured at startup.

However, another part of the design task is to display a "next customer" number between 0-9 at the end of the "Now Serving" message on the bottom half of the LED Matrix. Figure 8 shows the pixel representation of these digits.

Each time the "NEXTPB" button is pressed, the "next customer" number must changed by inserting it into the BRAM and overwriting the previous "next customer" digit. The color of the digit should be controlled by the NEXTCOLOR input at the time the button is pressed.

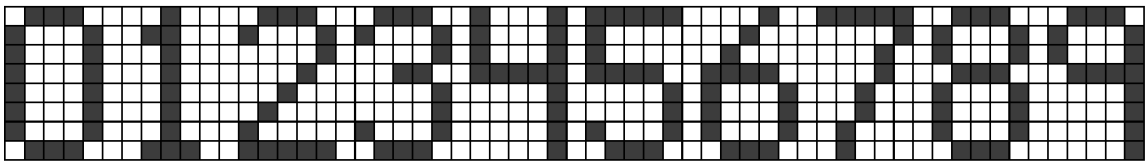

Figure 8 – 5x8 Pixel Encoding of the Numbers 0-9

We can update the digit display by writing it directly into the BRAM for the lower half of the LED Matrix at the appropriate column location.

To make updating a part of the memory easier, the BRAM should store the pixel values in column-major order. This means the 8 values for a each column should be stored in consecutive addresses in the memory. We can then configure the write port of the BRAM to write 32 bits at a time and use it to write columns, as shown in Figure 7. This allows us to update the number displayed in the BRAM using just 5 32-bit write operations (one for each of the 5 columns in the 5x8 character.

In creating this part of the design you will need to add hardware that keeps track of the current "next customer" value, decodes that number into a 5 x 8 character, and writes that character into the appropriate columns in the BRAM with the appropriate color.

4. Prelab

This is a complex design that will require a substantial amount of thought and planning if you are to complete it during the lab period. For this reason, the prelab assignment for this lab is to create a **design plan** for your LED Matrix Controller. A design plan is a document that describes the planned organization and operation of your intended circuit. It should be clear enough that you could give it to an engineer familiar with digital design and they would be able to implement your design without further information. Required components of this design plan are:

1. **Block diagrams** of your controller circuit and important submodules (particularly the sequencer).
2. **State transition diagrams** for any and all finite state machines in your design. Make sure to show all of the inputs and output of the FSM.
3. **Module descriptions** – a listing of each module in your design by name, followed by at least a sentence or two describing its functionality.

Diagrams may be either created with a drawing tool or *neatly* hand-drawn but must be integrated with the rest of the design plan. The end result should be a document in PDF format that is uploaded to Moodle. One design plan is required for each lab group.

5. In the Lab

This section describes the procedure you will follow in the lab. Please read over this procedure before you come to lab; feel free to ask questions at any time about anything you don't understand.

1. Log in to the PC at your lab station and visit the `ece_212_labs` directory of your Git repository. Make sure that your local repository is up to date with respect to GitLab.
2. Create a subdirectory within the `ece_212_labs` directory named `Lab03`. Create subdirectories `RTL`, `Simulation`, and `Constraints` to contain your source code and copy the source code for your sequencer design in Lab 2 into the appropriate subdirectories of `Lab03`.
3. Implement your new pixel generator design in SystemVerilog. You may find it helpful to do this in stages – start by instantiating and debugging the BRAM; then add the and debug the scrolling capability, and finally add and debug the "Next Customer" feature. During this process you may want to simulate parts of the design before attempting to debug them in hardware.
4. Debug your circuit and verify that the display on the LED Matrix meets the requirements for the design. **Demonstrate your working design to the instructor**.
5. Create screen captures of your simulation on two different time scales to show (1) the timing for a single row; and (2) the timing for display of all 8 rows. **Include these screen captures in your report**.
6. Include in your report a finalize version or your **block diagram and SystemVerilog listings for all modules used in your design**, including modules that you used in previous labs or downloaded from the Moodle page. Format all SystemVerilog listings using a fixed-width font and single-line spacing. You need not include a listing of your constraints file.
7. Commit your corrected SystemVerilog code to your Git repository and push your repository back to the GitLab server.

6. Report

You must submit an electronic copy of your lab report in PDF format on Moodle. The name of your PDF file should be of the form

"`Lab03_Report_LastName1_LastName2.pdf`"

Be sure to label each section and organize them in the order given. Messy or disorganized labs will lose points.

Each lab report should have a "scribe" (i.e., the person who writes most of the report). The scribe should alternate every lab. Please clearly indicate in the title block of your report the names of the lab group and the name of the scribe.

Your report should include the following sections:

1. **Introduction** – write a brief paragraph describing what you did in the lab. This should be a summary written in your own words; it is not acceptable to copy text for this from the lab handout. For this lab, describe the general strategy that your program takes to successfully met the lab requirements.

2. **Results** – Include the results specified in the "In the Lab" section including the boldfaced label (if any). Results in this lab include answers to questions, simulation timing diagrams captured from Vivado, and code listings of SystemVerilog modules. All code listings should be labeled to identify them and formatted single-space in a fixed-width font such as `Courier New`.
3. **Conclusion** – Briefly describe any difficulties that you encountered and how you resolved them. Also, if you have any suggestions for how to improve this lab, please include them here.
4. **Time Spent on Lab** – Please indicate how many hours you spent on this lab (including time during the lab period). This will not affect your grade, but will be helpful for calibrating the workload for future offerings of ECE 211.
5. **Suggestions for Improvement** – if you have any feedback about how this lab might be improved, please include it here.

Each section should begin with a section heading that includes the section number and title in a boldface font as in the report for Lab 1. Be sure to label each section and organize them in the order given. Messy or disorganized labs will lose points.

Appendix

The following are BRAM initialization pieces to display "ECE 212" and "Now Serving" in a red, yellow, green, and blue pattern. The content includes 32 blank columns at the start to assist in scrolling in the message.

NOTE: Each INIT row specifies a total of 256 bits with the least significant bit on the right. They are written using the spacer character "_" to break into 32-bit chunks that represent one column each. . They are shown broken down into 32 bit chunks with each chunk representing a single column. The least significant 32-bits are on the far right. The far right "`11226644`" in `INIT_04` for ECE 212 is the far left column of the E. The "`10006004`" is the 2nd column of the E, etc.

"ECE 212" Message

```
.INIT_00(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_04(256'h10000004_01226640_00000000_10000004_10006004_10006004_10006004_11226644),
.INIT_05(256'h10006004_10006004_10006004_11226644_00000000_01000040_10000004_10000004),
.INIT_06(256'h10020004_10200004_11000040_00000000_00000000_00000000_00000000_10000004),
.INIT_07(256'h11000040_00000000_10000000_11226644_10000040_00000000_10000640_10006004),
.INIT_08(256'h00000000_00000000_00000000_00000000_10000640_10006004_10020004_10200004),
```

"Now Serving" Message

```
.INIT_00(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_04(256'h10006000_01220000_00000000_11226644_00200000_00026000_00000600_11226644),
.INIT_05(256'h10000000_01220000_10000000_01226000_00000000_01220000_10006000_10006000),
.INIT_06(256'h10020004_10020004_10006640_00000000_00000000_00000000_00000000_01226000),
.INIT_07(256'h00220000_10206000_10206000_10206000_01220000_00000000_01200004_10020004),
.INIT_08(256'h00226000_00000000_00020000_00006000_00006000_00020000_11226000_00000000),
.INIT_09(256'h11226000_00000000_11226040_00000000_00226000_01000000_10000000_01000000),
.INIT_0A(256'h10020040_10020040_00006600_00000000_11226000_00000600_00000600_00006000),
.INIT_0B(256'h0000000000000000000000000000000000000000000000000000_01226600_10020040),
```

References

Xilinx Corporation, “7 Series FPGAs Memory Resources User Guide (UG473)”, 2016. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

Xilinx Corporation, “Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for HDL Designs (UG 953)”, 2017. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug953-vivado-7series-libraries.pdf