COE3DQ5 Lab #5
Integrating SRAM, VGA and UART Interfaces

Objective

To gain experience with self-checking testbenches. To design a digital system synchronized to a video graphics array (VGA) interface and an external static random access memory (SRAM). To understand how the universal asynchronous receiver/transmitter (UART) works. To transmit image files from the personal computer (PC) to the DE2 board, where they are stored, processed and displayed.

Preparation

- Revise the first four labs, especially finite state machines (FSMs) and SRAM and VGA interfaces
- Read this document and get familiarized with the source code and the in-lab experiments

Experiment 1

The objective of this experiment is to familiarize you with self-checking testbenches. One of the most powerful features of hardware description languages is the use of “software”-like language constructs for design verification. These features do not have an equivalent in terms of digital hardware; rather they are used exclusively to control the simulation, supply stimuli to the unit under test and to observe its responses. In addition, internal signals can be probed at any particular event.

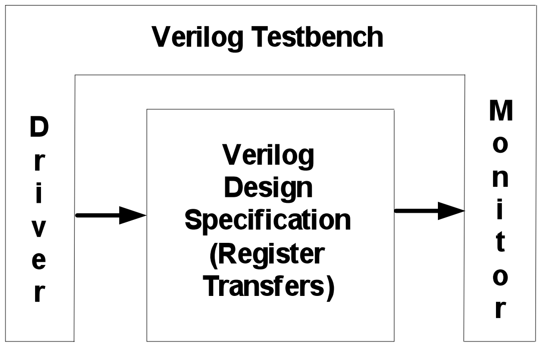

Figure 1 – Design verification using testbenches

In the lab you will explore the use of self-checking features by simulating the SRAM built-in self-test (BIST) that was introduced in the previous lab. Instead of using simulation waveforms to observe the discrepancies between the expected and actual responses, the top-level testbench (provided in the ***experiment1*** folder) checks for mismatches by monitoring a transition on the BIST mismatch signal. If this mismatch occurs, you can monitor the internal data lines using an always block sensitive to the positive edge of the clock signal. Note, in both the initial and always blocks you can use the *$write* task that has similar behavior to *printf* in C/C++. Custom tasks can also be defined (see for example master_reset) and included in always or initial blocks. The timing in the initial block can be controlled either by waiting for events (e.g., @) or by advancing the simulation time (using #). For this experiment, when programmed on the board, the BIST engine will start when push button 0 is pressed. Hence, in the testbench we will emulate this behavior by using push-button signals controlled in the initial block.

You have to perform the following tasks in the lab for this experiment:

- understand self-checking verification using SystemVerilog testbenches and ModelSim
- change the BIST FSM in order to generate 4 bursts (64k writes and 64k reads each) after push button 1 is pressed; if BIST fails monitor the failing data in the testbench in order to fix your FSM

Experiment 2

The aim of this experiment is to show how the external SRAM is interfaced to the VGA controller.

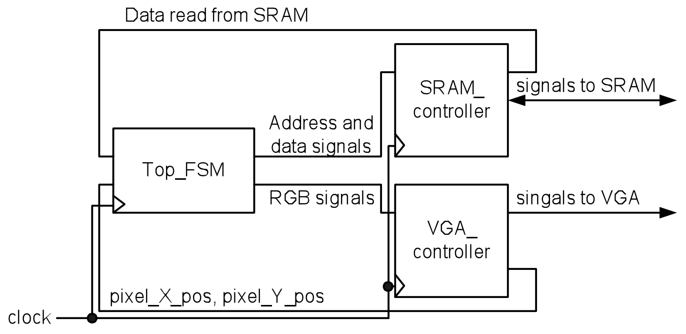

Figure 2 – Circuit for experiment2

After a picture of size 320x240 is generated on-chip and stored in the SRAM, it is read by an FSM that passes it to the VGA controller for displaying it in the center of the 640x480 VGA screen. The SRAM organization determines how the data is transferred between the two controllers. First, it is important to note that all the internal flip-flops are clocked at 50 MHz with the exception of a couple of control signals in the SRAM controller (as required by the SRAM protocol; this detail is not relevant to the system integrator). Note however although the VGA controller's flip-flops are clocked at 50 MHz, they are enabled every other clock cycle. Because the SRAM has only 2 bytes per location, we use one location to store the 8 most significant bits for two colors as follows: location 0 stores RG for pixel 0, location 1 stores BR for pixels 0 and 1 respectively and location 2 stores GB for pixel 1, …. Thus to provide 6 colors for 2 pixels, 3 memory reads are done every 4 clock cycles (SRAM is clocked at 50 MHz and the 3-byte color data is passed every second clock cycle to the VGA controller). The "Top FSM" from Figure 2 is used to direct how the data is fetched from the SRAM and transferred to the VGA. The inner behavior of this FSM can be explained using Table 1 and Figure 3 (see the next page for a detailed description).

State code	0	1	2	3	4	5	6	7	8	9	6	7	8	9
pixel_X_pos	157		158		159		160		161		162		163	
SRAM_address	A0	A1	A2		A3	A4	A5		A6	A7	A8		A9	A10
VGA_sram_data[2]				D0 (R0, G0)				D3 (R2, G2)				D6 (R4, G4)		
VGA_sram_data[1]					D1 (B0, R1)				D4 (B2, R3)				D7 (B4, R5)	
VGA_sram_data[0]						D2 (G1, B1)				D5 (G3, B3)				D8 (G5, B5)
VGA_red							R0		R1		R2		R3	
VGA_green							G0		G1		G2		G3	
VGA_blue							B0		B1		B2		B3	

Table 1 – State information for experiment2a

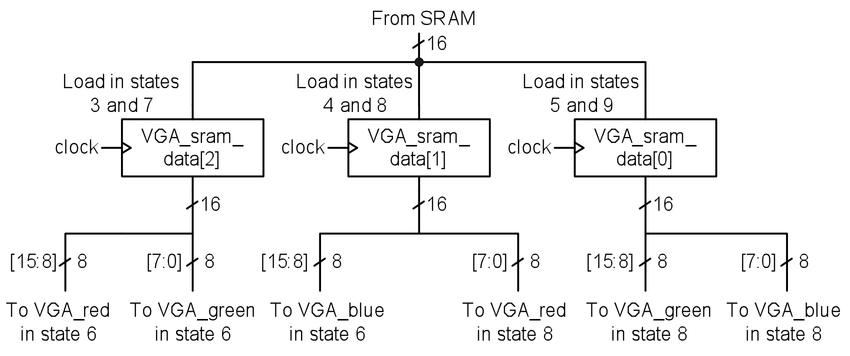

Figure 3 – Data transfer between the SRAM interface and the VGA interface for experiment2a

Because the picture is displayed from column 160, the SRAM must supply the appropriate RGB data at the right time. This is achieved by addressing memory location 0 (A0 in column 2, row 3 in the table) six clock cycles ahead. After A0 is generated, the data will “arrive” from the SRAM two clock cycles later and it will be buffered in the VGA_sram_data register in state 3. The first memory location will pass the red and green information for pixel 0 (R0 and G0). Note, the “State code” from the table is the one given by the state enumeration from the source code and it is used to show that once the data has started to be fetched, the same state sequence (states 6 to 9) is repeated until the end of the viewing area (on the currently displayed line) is reached. The same sequence is resumed when the following line is displayed. The three VGA_sram_data registers (16 bits each) are necessary to buffer the RGB data for the next two pixels before they are transferred to the VGA controller in states 6 and 8 respectively. This is necessary because there are only 2 bytes of data in one memory location, while each pixel requires 3 bytes (every second clock cycle because the clock is 50 MHz). Note the state transitions for filling in the SRAM are not covered in the table however its burst sequence can be extracted from the source code.

You have to perform the following tasks in the lab for this experiment:

- in the source code from the ***experiment2a*** folder, the data is organized in the memory as follows: RG, BR, GB, RG, …; modify the design in such way that the memory layout is changed to RR, GG, BB, RR, …; the first location has the red values for pixels 0 and 1 (from the viewing area); the second location has the green values for the first two pixels; …; several points are to be noted:
    - to ease your development the state table for reading the new memory layout is given in Table2; note, the data that comes from the SRAM is organized as (R0, R1), (G0, G1), (B0, B1), …
    - the reference design files are given in the ***experiment2b*** folder; the state transitions for filling in the RAM are already given and you need to complete the interface from the SRAM to the VGA;
    - note, in state 7, VGA_sram_data[2][7:0] must be buffered in “VGA_red_buf” in order to avoid being overwritten by the incoming data; this buffered value is to be passed to the VGA controller in the following clock cycle (this is illustrated in the circuit diagram shown in Figure 4)
    - your modified design must first pass the top-level testbench before being placed on the board

State code	0	1	2	3	4	5	6	7	8	9	6	7	8	9
pixel_X_pos	157		158		159		160		161		162		163	
SRAM_address	A0	A1	A2		A3	A4	A5		A6	A7	A8		A9	A10
VGA_sram_data[2]				D0 (R0, R1)				D3 (R2, R3)				D6 (R4, R5)		
VGA_sram_data[1]					D1 (G0, G1)				D4 (G2, G3)				D7 (G4, G5)	
VGA_sram_data[0]						D2 (B0, B1)				D5 (B2, B3)				D8 (B4, B5)
VGA_red_buf								R1				R3		
VGA_red							R0		R1		R2		R3	
VGA_green							G0		G1		G2		G3	
VGA_blue							B0		B1		B2		B3	

Table 2 – State information for experiment2b

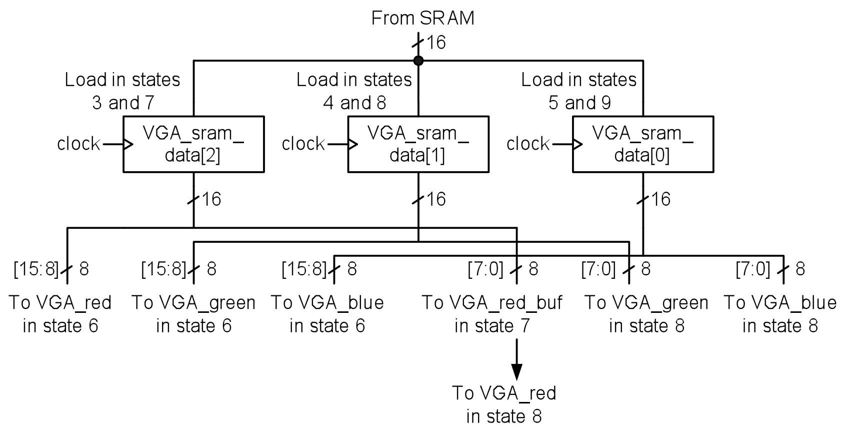

Figure 4 – Data transfer between the SRAM interface and the VGA interface for experiment2b

Experiment 3

This experiment introduces the UART interface which is commonly used for serial communication between digital devices. To communicate through an UART interface two separate paths are needed: one for transmitting and one for receiving data, as shown in Figure 5. In the following we will introduce the UART protocol and the state machines from the design implemented in the DE2 board (the ***experiment3*** folder). These state machines are used to receive/transmit 512 bytes of data from/to the PC.

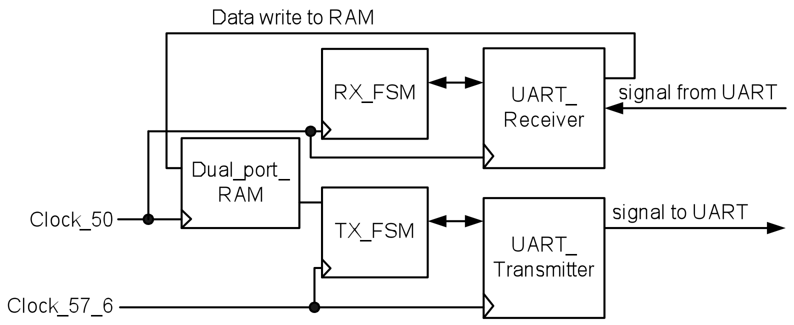

Figure 5 – Circuit using the UART interface for experiment 3

The UART interface is asynchronous because it does not have a dedicated clock line and it recovers the timing information from the data stream by monitoring the start and stop bits, as shown in Figure 6. In our implementation each packet of data embedded between a start and a stop bit has 8 bits (the number of bits can vary between 5 to 8, depending on the settings of the UART protocol). The data can be sent at different bit rates (or baud rates) varying from 300 bits per second (bps) to 115,200 bps. Additional info (such as type of parity, handshaking, longer duration for stop bits) can be used in the communication protocol between two UART devices. For the sake of simplicity, we have implemented our UART controller to accept data at 115,200 bps, 8 data bits, no parity, 1 stop bit and no handshaking.

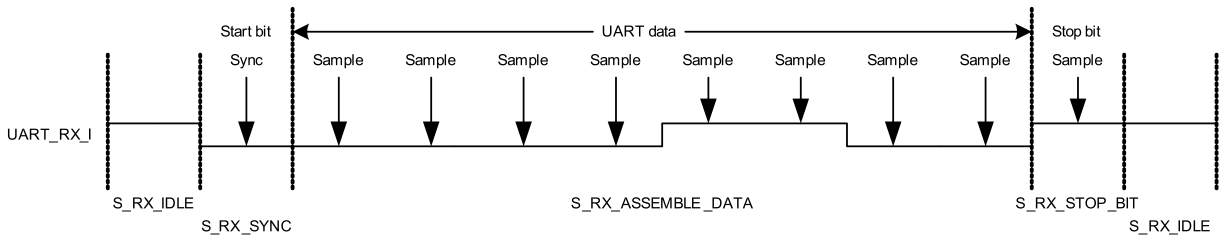

Figure 6 – The timing diagram of the UART protocol

The waveform (that is initiated by the transmitter) is shown in Figure 6. For this example, the transmitter sends the value 8'h30 (the first data bit on the left-hand side is the least significant bit of the byte that is assembled). After the receiver senses a negative edge on the UART_RX_I line, it will go into a sync state to confirm the presence of the start bit. Due to the asynchronous nature of the UART protocol, it is required that the receiver samples the data half-way during the transmission time of each bit. Since in our implementation we use a 50MHz reference clock, in order to receive data at 115,200 bps, we employ a counter that goes from 0 to (434/2 -2) before a sampling occurs (434 is approximately 50,000,000 divided by 115,200, which gives the number of 20ns pulses for the duration of 1 bit of received data). This ensures that successive samples will occur roughly in the "middle" of the receive time of each bit.

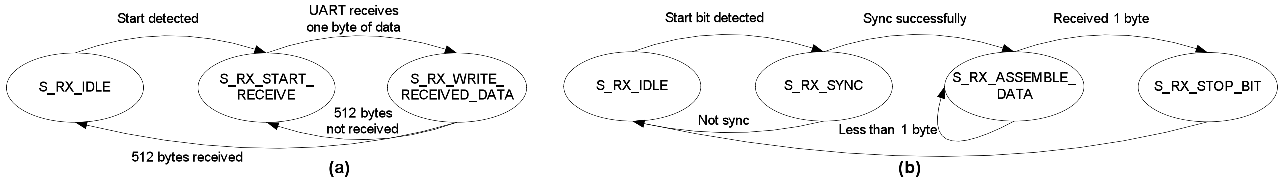

Figure 7 – State transition diagrams for the UART receiver

The state transition diagram of the UART receiver is shown in Figure 7(b). This state machine assembles data bit by bit and it passes the bytes to an FSM in the top-level design whose state transition diagram is shown in Figure 7(a). The FSM from Figure 7(a) (called RX_FSM in Figure 5) controls the UART receiver by deciding when the receiver should be enabled and when the data should be off-loaded for processing (in our case the only processing is storing the received bytes in a dual-port RAM). If the UART receives a new byte before the old one has been processed an overrun flag will be set. If the start/stop bit protocol is violated, a frame error flag will be raised.

You have to perform the following tasks in the lab for this experiment:

- understand how the receive part of the UART interface works; note, switch 0 (switch 1) needs to be toggled from 0 to 1 each time a new file is received (transmitted);
- develop the code for the transmit part of the UART interface and verify its correctness
    - the state transition diagrams of the TX_FSM (from Figure 5) and the UART transmitter are shown in Figures 8(a) and 8(b) respectively; the TX_FSM reads the data from the embedded memory and it will control the UART transmitter by providing 1 byte of data at a time; this data needs to be loaded in the data buffer when start becomes a 1 in the S_TX_IDLE state from Figure 8(b) (at the same time the Empty flag and the data count register need to be cleared)
    - the UART transmitter FSM will subsequently serialize the data buffer according to the UART protocol (illustrated in Figure 6); it is important to note that to achieve transmission at exactly 115,200 bps, it is required to use an additional clock in the design as explained below
    - a 27 MHz oscillator from the DE2 board will supply the reference clock to an internal phase-locked loop (PLL) that generates an internal 57.6 MHz clock; this clock will subsequently be divided to obtain 115.2 KHz, as it is required to transmit data at 115,200 bps
    - the FSM from Figure 8(a) is already implemented, as well as all the details regarding clock division (PLL, …); your only task is to complete the implementation of the FSM shown in Figure 8(b) and confirm that data is correctly sent back to the PC terminal

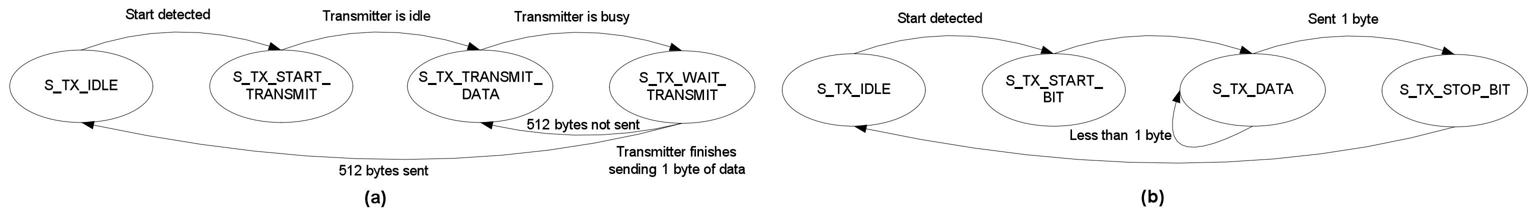

Figure 8 – State transition diagrams for the UART transmitter

Experiment 4

In this experiment you will put together the UART, SRAM and VGA interfaces in order to display images sent from the PC. Using the terminal software available on the PC, you will send PPM files. The PPM format has a small header that contains information such as picture size, which is followed by raw RGB data in the order RGB RGB RGB … There is 1 byte of data for each color for each pixel. An image of size 320x240 will thus require 320x240x3, or 230,400 bytes (or 225Kbytes) of data stored in the PPM file.

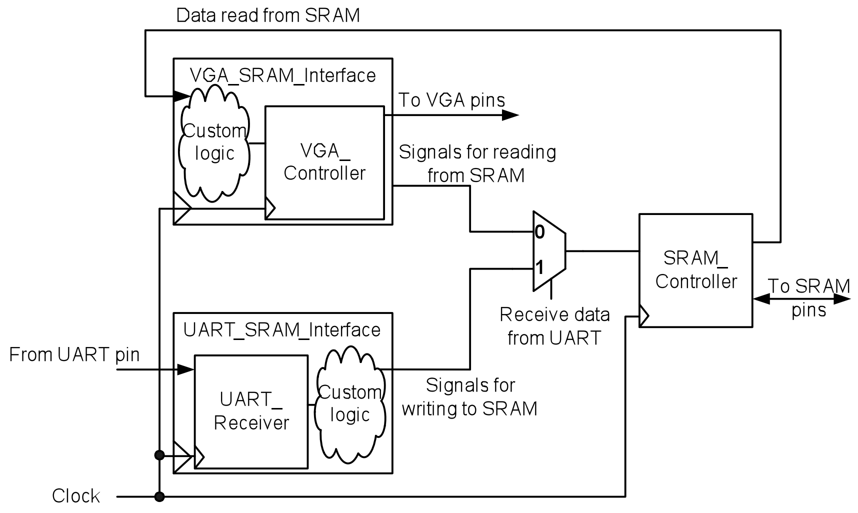

Figure 9 – Circuit for experiment 4

The top-level diagram of the circuit for this experiment is shown in Figure 9. It utilizes the different components (i.e., the SRAM Controller, VGA Controller, UART Receiver) that were introduced in the previous experiments. The UART_SRAM_Interface enables you to send the PPM file from the PC and store it in the external SRAM. The VGA_SRAM_Interface will read the stored image from the SRAM and display it on the screen (based on the same principles detailed in ***experiment 2***).

You have to perform the following tasks in the lab for this experiment:

- understand how the entire design works together, including its top-level testbench
- modify the design from ***experiment4b*** as follows; a low-pass filter is used to reduce the "noise" from the image by averaging the RGB values of consecutive pixels; this simple filter has a "smoothing" effect on the image; note, the quality of the filter is irrelevant in this experiment; rather the main focus is its real-time implementation between the SRAM and VGA interfaces; the following should be noted:
  - the data that comes from the SRAM (8 bits per color) is extended to 10 bits by using 00 on the two least significant bits; this value is buffered for each pixel; when the data for the next pixel arrives, the buffered values for RGB are averaged with the new pixel values, as explained next
  - the 10 bit values are extended to 11 bits by placing a 0 on the most significant bit (to avoid overflow when adding two large positive values); to perform division by 2 for (averaging), the result of the addition is truncated by removing the least significant bit (as shown in Figure 10);

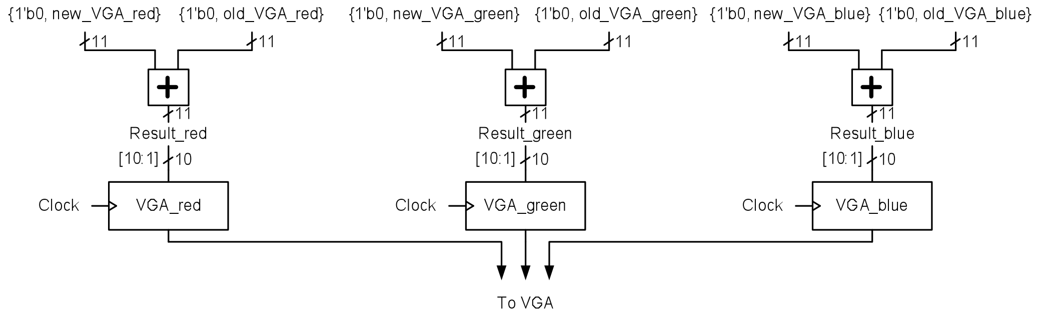

Figure 10 – Implementing a simple low-pass filter through averaging consecutive pixels

Write-up Template

COE3DQ5 – Lab #5 Report
Group Number
Student names and IDs
McMaster email addresses
Date

There are 2 take-home exercises that you have to complete as specified below. When the source files are requested, submit the Verilog and "QSF" files, as well as the ModelSim "do" files. Label the top-level modules as exercise1 and exercise2. If you will add/remove/change the signals in the port list from the port names used in the in-lab experiments, make sure that these changes are properly documented in the source code.

Exercise 1 (2 marks) – Modify the BIST engine from ***experiment1*** as follows. First, write pattern 1…1 (ALL1 pattern) in all the even memory locations and pattern 0…0 (ALL0 pattern) in all the odd memory locations. After all the memory locations have been written, read back each memory location to make sure that there are no defects in the memory. Then write the ALL1 pattern in all the odd locations and the ALL0 pattern in all the even locations. As before, this should be followed by reading back the values to check for any mismatches. When writing/reading the ALL1 pattern to/from the memory, the address lines should switch in the *increasing* order. When writing/reading the ALL0 pattern to/from the memory, the address lines should switch in the *decreasing* order. Use a *self-checking testbench* to verify if your BIST engine works correctly. Please note that no waveforms need to be used and your submitted testbench file must have "always" blocks that are monitoring if the BIST engine is implemented correctly.

Submit your sources and in your report write at most a half-a-page paragraph describing your reasoning.

Exercise 2 (2 marks) – Modify ***experiment2*** in order to account for the following memory map. The external SRAM memory should be organized in five memory segments, as shown in Figure 11 (see next page). The first memory segment stores all the R values (the "R segment"). The second memory segment stores all the even G values (the "even G" segment). The ~~third~~ fourth memory segment stores all the even B values (the "even B segment"). The ~~fourth~~ third memory segment stores all the odd G values (the "odd G" segment). The fifth memory segment stores all the odd B values (the "odd B segment").

The number of memory locations in the R segment is 38,400. The first location from the R segment stores the red color values for pixels 0 and 1. The second location from the R segment stores the red color values for pixels 2 and 3, and so on. The number of memory locations in each of the G and B segments is 19,200. The first location from the even G segment stores the green color values for pixels 0 and 2. The second location from the even G segment stores the green color values for pixels 4 and 6, and so on. The first location from the odd G segment stores the green color values for pixels 1 and 3. The second location from the odd G segment stores the green color values for pixels 5 and 7, and so on. The same memory organization as for the even and odd G segments is followed by the even and odd B segments. Note, the image that is written into the SRAM and displayed on the VGA monitor is the same as for the in-lab ***experiment2***. Your task is to modify the in-lab design in order to account for the change in the memory map as described above.

Submit your sources and in your report write at most a half-a-page paragraph describing your reasoning.

VERY IMPORTANT NOTE:

This lab has a weight of 4% of your final grade. The report has no value without the source files, where requested. Your report must be in "pdf" format and together with the requested source files it should be included in a directory called "coe3dq5_group_xx_takehome5" (where xx is your group number). Upload this directory onto the SVN server (in your group directory as described in the SVN guide posted on the course website) before noon one week plus one extra day after the day you have done the in-lab experiments for lab 5. Late submissions will be penalized.

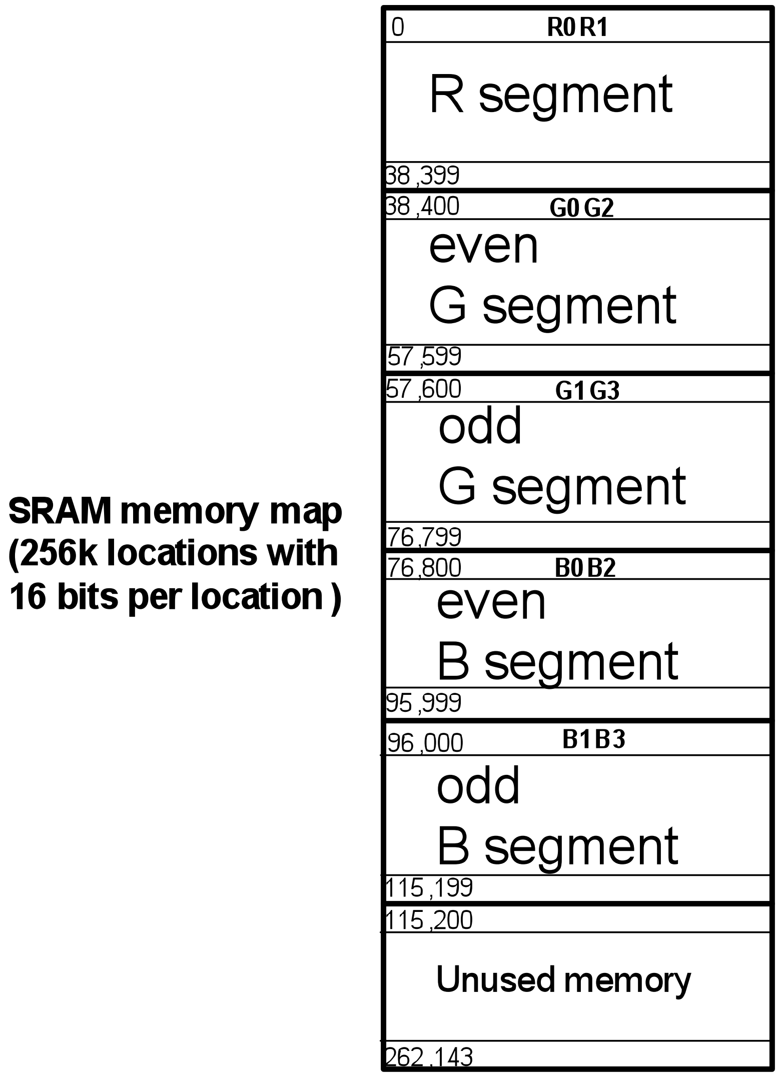

Figure 11 – The segmented memory map for take-home exercise 2.