Laboratory Exercise 10

Hardware Accelerator

The purpose of this exercise is to learn how to implement hardware acceleration. You will create an application to draw images and perform animation using a DE-series board. The application will draw lines from the center of the screen to its boundary. You will implement the line-drawing in software first, using the Bresenham's line-drawing algorithm. Then, you will accelerate your application (i.e. improve its performance) by implementing the line-drawing algorithm as a hardware circuit. Your accelerator will be controlled by the Nios II processor using an Avalon Slave Interface and will draw lines on the screen via an Avalon Master Interface.

Prerequisite: It is necessary to do Laboratory Exercise 7 to learn how to draw lines on a screen.

Background

Hardware accelerators are circuits designed to offload specific tasks from the processor. They perform functions that run faster in hardware than if executed entirely in software. An example of a system with a hardware accelerator is shown in Figure 1. It is similar to the DE-series Media Computer except that it includes a hardware accelerator designed to implement the line-drawing algorithm.

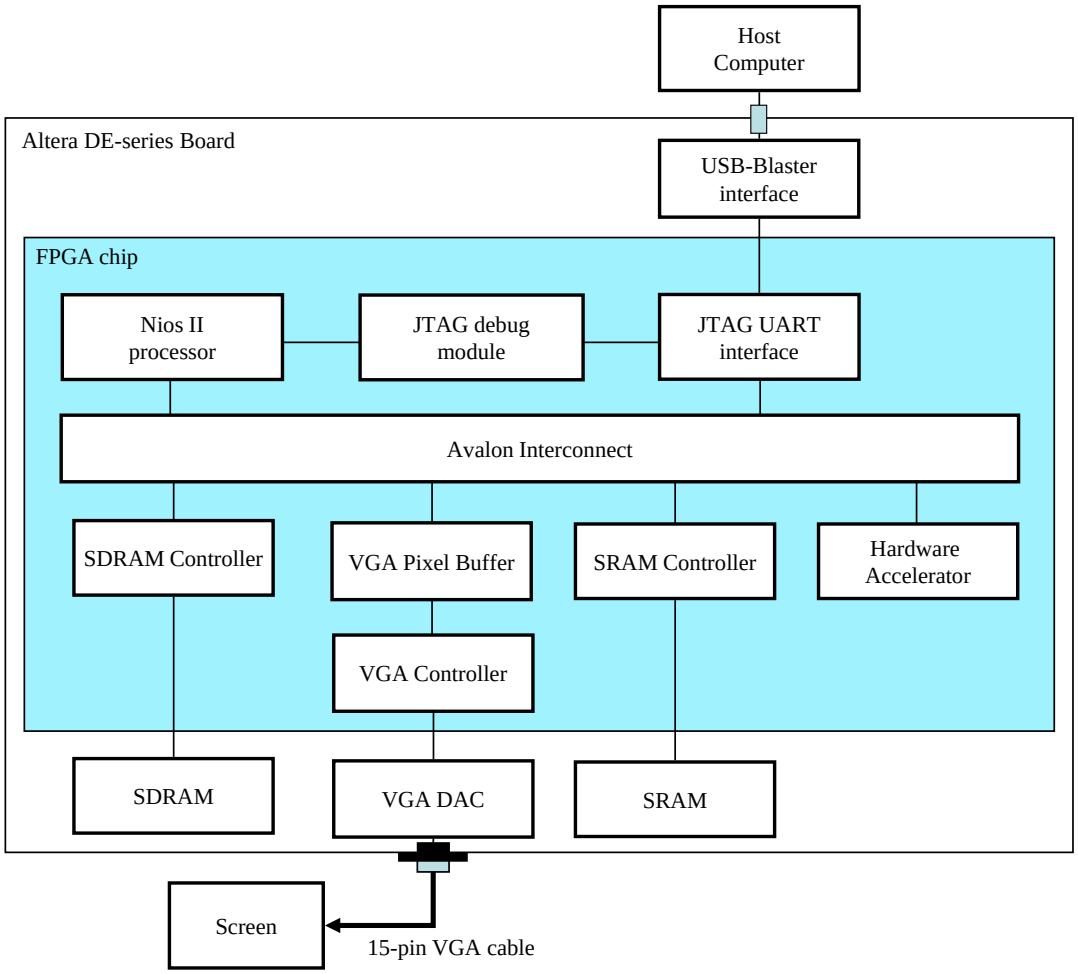

Figure 1: A system with a hardware accelerator.

When an application uses software to draw a line, the processor has to first calculate the location of each pixel on the line and then store it in the SRAM memory. On the other hand, when the hardware accelerator is used, the processor only needs to give the information about the line to the accelerator and ask it to draw. The processor can then wait for the accelerator to finish or work on other tasks.

The system in Figure 1 can be implemented using the Quartus II Qsys tool. You should read the tutorial *Making Qsys Components* to learn how to implement the hardware accelerator in such a system.

Part I

Write a C-language program that randomly draws lines from the center to the boundary of the screen using the Bresenham's algorithm learned in Laboratory Exercise 7. The program should first clear the screen by setting all pixels to black, and then try to fill the whole screen with the same color by drawing random lines from the center. It should include a **for** loop which will execute the line-drawing algorithm for a large number of iterations (e.g. 10,000 times). After the lines fill the screen, you should change the color and repeat the procedure.

Complete this part of the exercise as follows:

1. Write a C-language function to implement the line-drawing algorithm.
2. Write a program that performs the desired animation, using the line-drawing function.
3. Create a new project in the Altera Monitor Program using the DE-series Media Computer.
4. Change the $-O1$ to $-O0$ under $Additional\ compiler\ flags$ field. This reduces any optimizations during compilation.
5. Download the computer system onto the DE-series board.
6. Compile and run your program.

Part II

In this part of the exercise you will begin designing the hardware accelerator, called the Line-Drawing Algorithm (LDA) peripheral, which is shown in Figure 2. The accelerator includes three components: an LDA circuit, an Avalon Slave Interface, and an Avalon Master Interface. The LDA circuit implements the line-drawing algorithm in hardware. The slave interface is used by the processor to control the LDA peripheral, and the master interface is used by the LDA circuit to draw a pixel at the given location on the screen. Details about these three components will be given later.

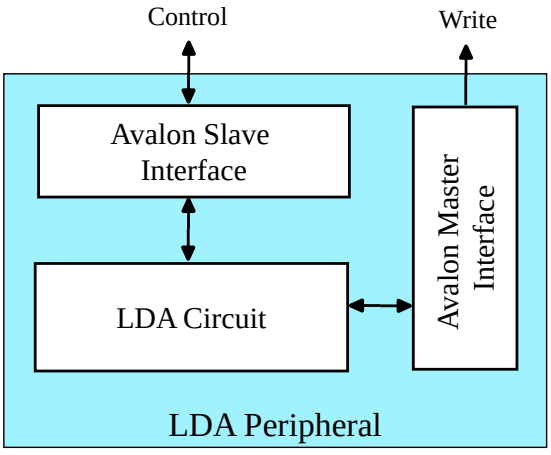

Figure 2: LDA peripheral diagram.

Design the LDA circuit which implements a line-drawing algorithm in hardware. The pseudo-code for the algorithm is shown in Figure 3

```
1  draw_line(x0, x1, y0, y1)
2
3      boolean is_steep = abs(y1 - y0) > abs(x1 - x0)
4      if is_steep then
5          swap(x0, y0)
6          swap(x1, y1)
7      if x0 > x1 then
8          swap(x0, x1)
9          swap(y0, y1)
10
11     int deltax = x1 - x0
12     int deltay = abs(y1 - y0)
13     int error = -(deltax / 2)
14     int y = y0
15     if y0 < y1 then y_step = 1 else y_step = -1
16
17     for x from x0 to x1
18         if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
19         error = error + deltay
20         if error >= 0 then
21             y = y + y_step
22             error = error - deltax
```

Figure 3: Pseudo-code for a line-drawing algorithm.

The circuit should have the following inputs and outputs, as shown in Figure 4:

- clk – clock signal.
- $resetn$ – active-low reset signal.
- $X0$ – 9-bit input signal, specifying the x coordinate of the starting point.
- $Y0$ – 8-bit input signal, specifying the y coordinate of the starting point.
- $X1$ – 9-bit input signal, specifying the x coordinate of the end point.
- $Y1$ – 8-bit input signal, specifying the y coordinate of the end point.
- $Color$ – 16-bit inout signal, specifying the color of the line to be drawn. Its value is unchanged in the LDA circuit.
- Go – 1-bit input signal, triggers the calculations of pixel locations of the line. This signal is raised high only when all the signals above have already been asserted.
- $Write\_Finish$ – 1-bit input signal to acknowledge that the pixel has been drawn, prompting the LDA circuit to calculate the next pixel location.

- $Pixel\_Address$ – 32-bit output signal, indicating the pixel location (i.e. SRAM address that stores the pixel).
- $Draw$ – 1-bit output signal, specifying that the $Pixel\_Address$ signal is valid and the pixel at that address should be drawn. After the $Draw$ signal is asserted, the LDA circuit should stall until a $Write\_Finish$ signal is received.
- $Done$ – 1-bit output signal, indicating that all pixels in the line have been drawn, and the LDA circuit is ready to draw another line.

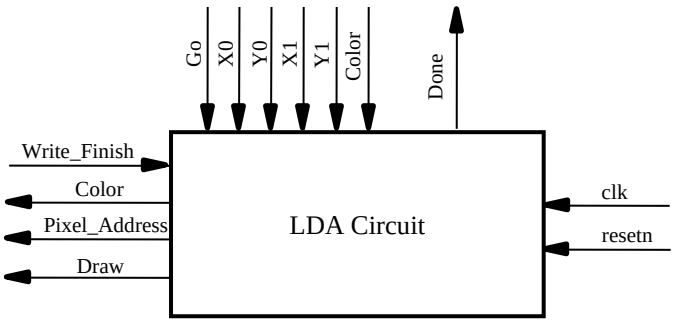

Figure 4: LDA circuit inputs and outputs.

Design and implement the LDA module by completing the following:

1. Create a new Quartus II project for this exercise.
2. Create the Verilog code for the LDA module, include it in your project, and compile the circuit.
3. Use functional simulation to verify that your circuit is correct. An example of the output produced by the functional simulation for a correctly designed circuit is given in Figure 5. The figure describes the sequence of drawing a line from (0,0) to (4,2).

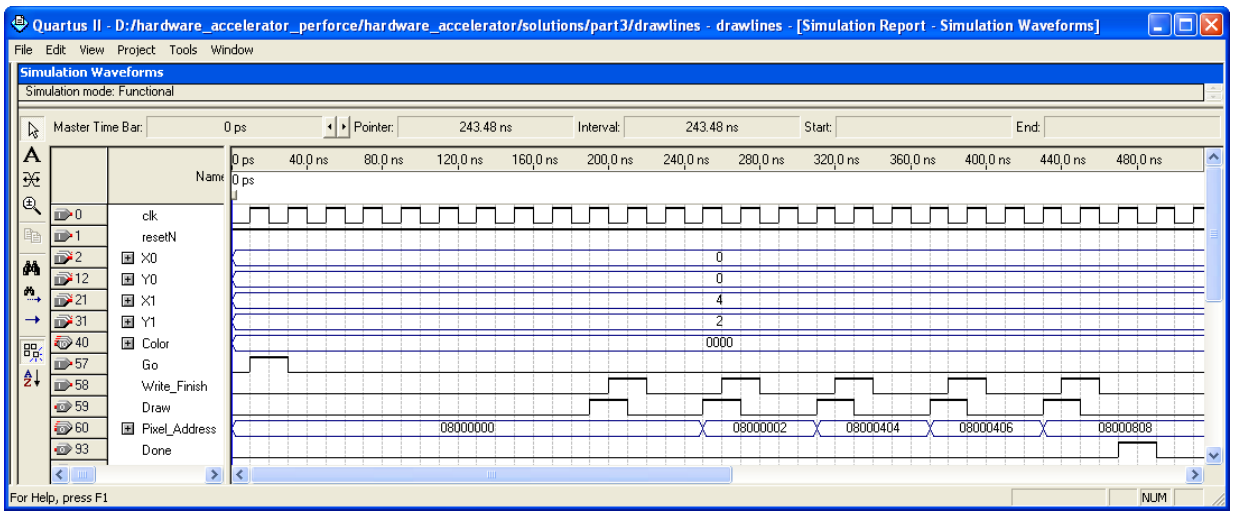

Figure 5: Simulation of the LDA circuit.

Part III

In this part, you will enhance your LDA circuit by adding an Avalon Slave Interface, so that the LDA peripheral can communicate with the processor. The interface has to provide the following inputs to the LDA circuit: Go signal, the starting point of the line (X_0, Y_0), the end point (X_1, Y_1), and $Color$ signal. After the LDA circuit finishes drawing a line, a $Done$ signal has to be received by the interface.

Avalon Memory-Mapped Slave Interface

The LDA peripheral should implement the Avalon Memory-Mapped Interface by including several registers, which the processor can access to instruct the peripheral to draw lines on the screen. The required registers are shown in Figure 6.

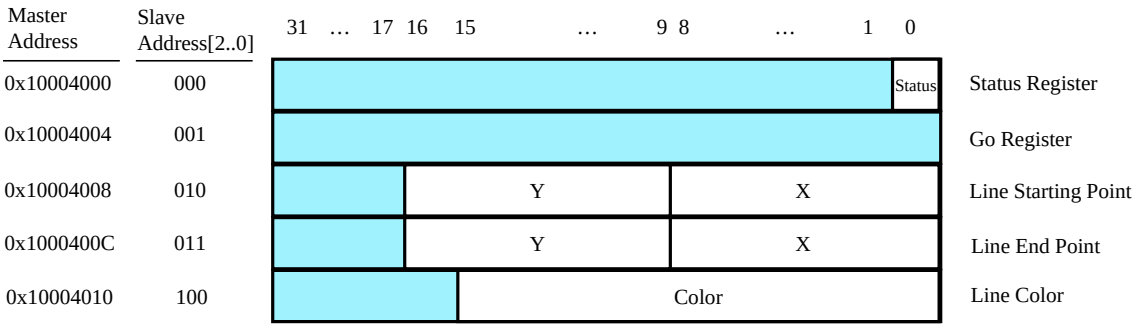

Figure 6: The LDA peripheral memory-mapped registers.

The *Line Starting Point* and the *Line End Point* registers store the *X* and *Y* coordinates for the starting point and the end point of the line, respectively. The *X* coordinate is placed in bits [8..0] and the *Y* coordinate in bits [16..9]. The color is stored in the *Line Color* register, in bits [15..0]. By writing into the *Go* register, the processor initiates the drawing algorithm implemented by the peripheral. Note that the processor must set the *Line Point* registers and the *Color* register before writing to the *Go* register.

After the processor initiates the execution of drawing a line, it should continuously poll bit b_0 of the *Status* register to check whether the LDA peripheral finished drawing the line. The bit b_0 will assume a value of 1 once the line drawing has finished. Only then can the processor instruct the peripheral to draw a new line. The peripheral will not allow (i.e. the command is ignored) the processor to initiate a new drawing operation until it finishes drawing the current line. After the processor has read the value of 1, it has to reset the status register to 0.

Pay attention to the *master address* and the *slave address* in Figure 6. The *master address* is a 32-bit address in Nios II's address space. So, if Nios II wants to write to the *Go* register, it will present the address 0x10004004 to the Avalon Interconnect. This address will then be translated by the Avalon Interconnect into the slave address, which is 001 in binary. The slave address width is only 3 bits because there are only 5 registers, hence 3 bits are enough to encode the offset of each register.

Avalon read/write transfers

A typical Avalon Memory-Mapped interface supports read and write transfers with a slave-controlled *waitrequest*. To begin a transfer, the master asserts $address$, $byteenable$, $read$ or $write$, and $writedata$ signals on the rising edge of the clock. When a slave receives these signals, it captures the data written to it or it outputs the requested data (read). If the slave is unable to respond within one cycle, it can assert $waitrequest$ before the next rising edge of the clock to hold the transfer. When the $waitrequest$ is asserted, the transfer is delayed and the address and control signals are held constant. The slave can stall the interconnect for as many cycles as required by keeping the $waitrequest$ signal high. The transfer will complete on the first rising edge of the clock after the slave deasserts $waitrequest$.

In this part of the exercise, your slave interface does not need to assert the $waitrequest$ as long as it can respond within one cycle. This means that the slave should either present valid data (read transfer) or capture the data (write transfer) before the next cycle. Figure 7 shows two examples of transfers.

1. The master asserts $address$ and $read$ on the rising edge of the clock. In the same cycle the slave decodes the signals from the master and presents valid $readdata$.
2. The master captures $readdata$ on the rising edge of the clock and deasserts the address and control signals. The transfer ends.
3. No control signals are asserted.
4. The master asserts $address$, $write$, and $writedata$ on the rising edge of the clock. The master signals are held constant and the slave decodes them.
5. The slave captures $writedata$ on the rising edge of the clock. The master deasserts the $address$, $writedata$ and control signals. The transfer ends.

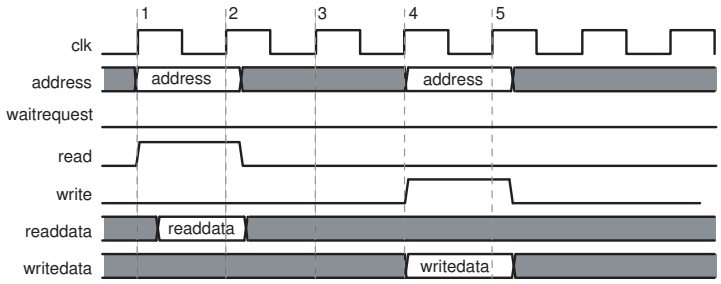

Figure 7: Avalon timing diagrams for read/write transfers (without $waitrequest$) - 1 cycle duration.

Note that the $chipselect$ signal is not shown in Figure 7. This signal is asserted by the Avalon interconnect whenever any transfer is active. Hence, the $chipselect$ signal is asserted along with a *read* or a *write* control signal.

To help you complete this part of the exercise, the system shown in Figure 1 is provided in the starter kit that comes with this exercise. Implement the Avalon slave interface using the Quarts II project provided:

1. Complete the Verilog code for the $LDA\_slave\_interface$ module and copy the LDA circuit you designed in Part II into the project. These files should be copied into the directory *nios_system/synthesis/submodules* as well.
2. Use $LEDR_{17-0}$ to display X_0 and X_1, HEX_{3-0} for $Color$, HEX_{5-4} for Y_1, and HEX_{7-6} for Y_0. Connect the Go signal to $LEDG_0$ and connect the $Done$ signal to $LEDG_1$. In order to do that, you should use the conduit signals provided in the top-level file $LDA\_peripheral$.
3. Compile the circuit.
4. Create a new project in Altera Monitor Program using <*Custom System*>. You should include the *.qsys* and *.sof* files generated by Quartus II in $System\ details$.
5. Download the computer system onto the DE-series board.
6. Test the functionality of your peripheral by executing a C-language program and observing the LEDs and 7-segment displays.

Part IV

In this part, you will complete your accelerator by adding the Avalon Master Interface to be used for transferring data between the LDA circuit and the SRAM Controller. The master interface is supposed to receive the $Pixel\_Address$ and $Color$ signals from the LDA circuit when the $Draw$ signal is asserted. It should then put the $Pixel\_Address$ on the $address$ lines and the $Color$ information on the $writedata$ lines in the Avalon interconnect. This data will get passed on to the SRAM. After the drawing is finished, the master interface will assert the $Write\_Finish$ signal to let the LDA circuit know that a new pixel can be drawn. Refer to Figure 2 for the block diagram of the LDA peripheral.

When designing the master interface, keep in mind that the $waitrequest$ signal may be asserted by the SRAM Controller. In such a case, the master interface for the LDA peripheral should maintain all of its control signals in subsequent clock cycles until the $waitrequest$ is deasserted. Notice that a $byteenable$ signal is associated with the master interface. This signal enables writing to specific bytes during transfers. Each bit in $byteenable$ corresponds to a byte in $writedata$. In our case of data transfer between the LDA peripheral and the SRAM, both bytes are written. Therefore, the $byteenable$ value should remain $(11)_2$ all the time.

Figure 8 shows examples of variable duration transfers. Below is the explanation for a write transfer that is 3 cycles in duration. The slave prolongs the transfer for 2 extra cycles by using the $waitrequest$ signal.

1. The master asserts $address$, $write$ and $writedata$ on the rising edge of the clock. In the same cycle, the slave decodes the signals from the master and asserts $waitrequest$, which stalls the transfer.
2. The master samples the asserted $waitrequest$ on the rising edge of the clock. Thus, the $address$, $write$ and $writedata$ signals remain constant.
3. The slave deasserts $waitrequest$ on the rising edge of the clock.
4. The slave captures $writedata$ on the rising edge of the clock. The master samples the deasserted $waitrequest$ and ends the transfer by deasserting all signals.

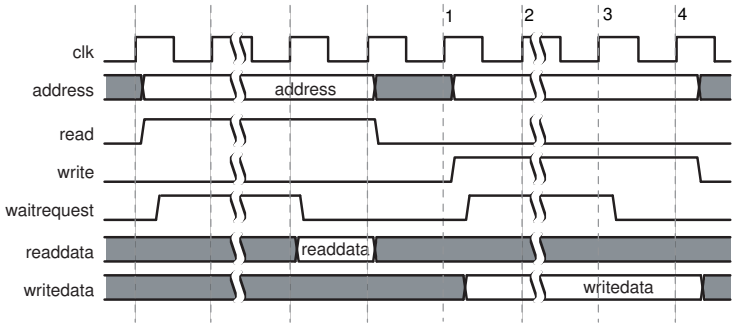

Figure 8: Avalon timing diagram for read/write transfers (with $waitrequest$) - variable duration.

Finish the LDA peripheral that has a LDA circuit, an Avalon slave interface, and an Avalon master interface. Complete the following:

1. Complete the Verilog code of $LDA\_master\_interface$ and compile the Quartus II project. Make sure the file is copied to the directory *nios_system/synthesis/submodules* before the compilation.
2. Modify the line-drawing function in part II so that the line is drawn using the LDA peripheral.
3. Download the computer system onto the DE-series board.
4. Compile and run your program.
5. Compare the speed of the animation obtained in Parts I and IV.

Preparation

The recommended preparation for this laboratory exercise includes:

1. C code for Part I
2. Verilog code for Parts II, III, and IV