Hardware Implementation of Connected Component Labelling

Project Report

Embedded Systems
CSEE W4840

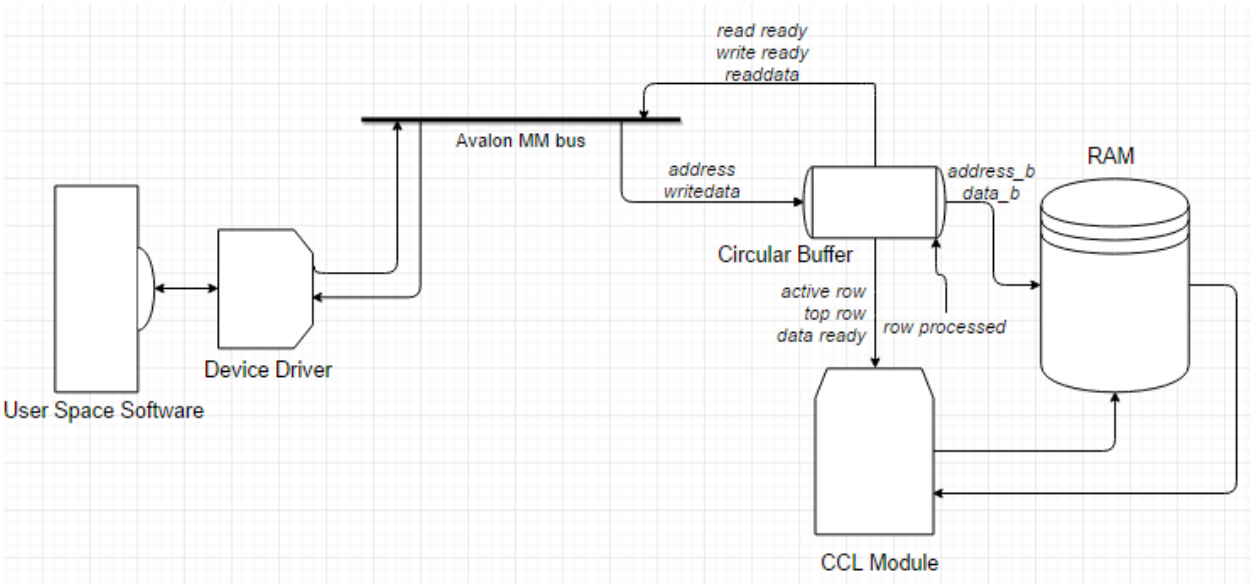
Avinash Nair, Manushree Gangwar, Jerry Barona
asn2129, mg3631, jab2397

Contents

Preface

This document describes the components and procedures that constitute of the project. It mainly consists of two parts involving the two interacting interfaces (Linux and FPGA). The entities and algorithms within each interfaces are also described. For deeper understanding of the procedures, flowcharts have been added.

System Architecture



System Components

Linux

User Space Program

The main user space program is responsible for the overall flow of the program and ensuring that each of the subcomponents have the required inputs to execute properly in a timely manner. The main tasks performed by the user space program are described below:

Establishing connectivity with the CCL device through the device driver

We have written a device driver for the CCL device (the FPGA) which allows user space programs to connect and communicate with the device. The device driver for our CCL device implements multiple ioctl commands to send data, commands and status notifications between the CCL device and the user space program. The user space program is responsible for establishing a connection to the device through the driver when the program initializes.

Reading Input Images

The program assumes that the input images that need to be processed are stored on the hard disk in a PNG file format and we need to read the pixel data from the file in order to process them. The connected component labeling algorithm also assumes that the input images have a width (horizontal resolution) of 640 pixels and will not work if this criterion is not met.

We make use of an open source library called FreeImage to process the image file headers and extract the pixel data from the file. We convert the input images to an 8 bit greyscale pixel format if they are in a different format using functions from the FreeImage library.

Transferring pixel data to and from the FPGA

Since the FPGA has limited memory capacity, the user space program needs to divide the image into smaller pieces, transfer the smaller blocks to the FPGA for processing and merge the processed blocks back together to get the final result image. The connected component algorithm does raster scans of an input image to determine the labels for each pixel and needs to process information from the top, top-left and left neighbors of each pixel. In order to facilitate this processing we implement a circular buffer on the FPGA that is large enough to hold 3 full rows of image data and transfer an entire row of pixels at a time.

The user space program divides the input image into its constituent rows and transfers the rows to the FPGA one at a time. We make use of the FreeImage library to extract pixels corresponding to each row of the image.

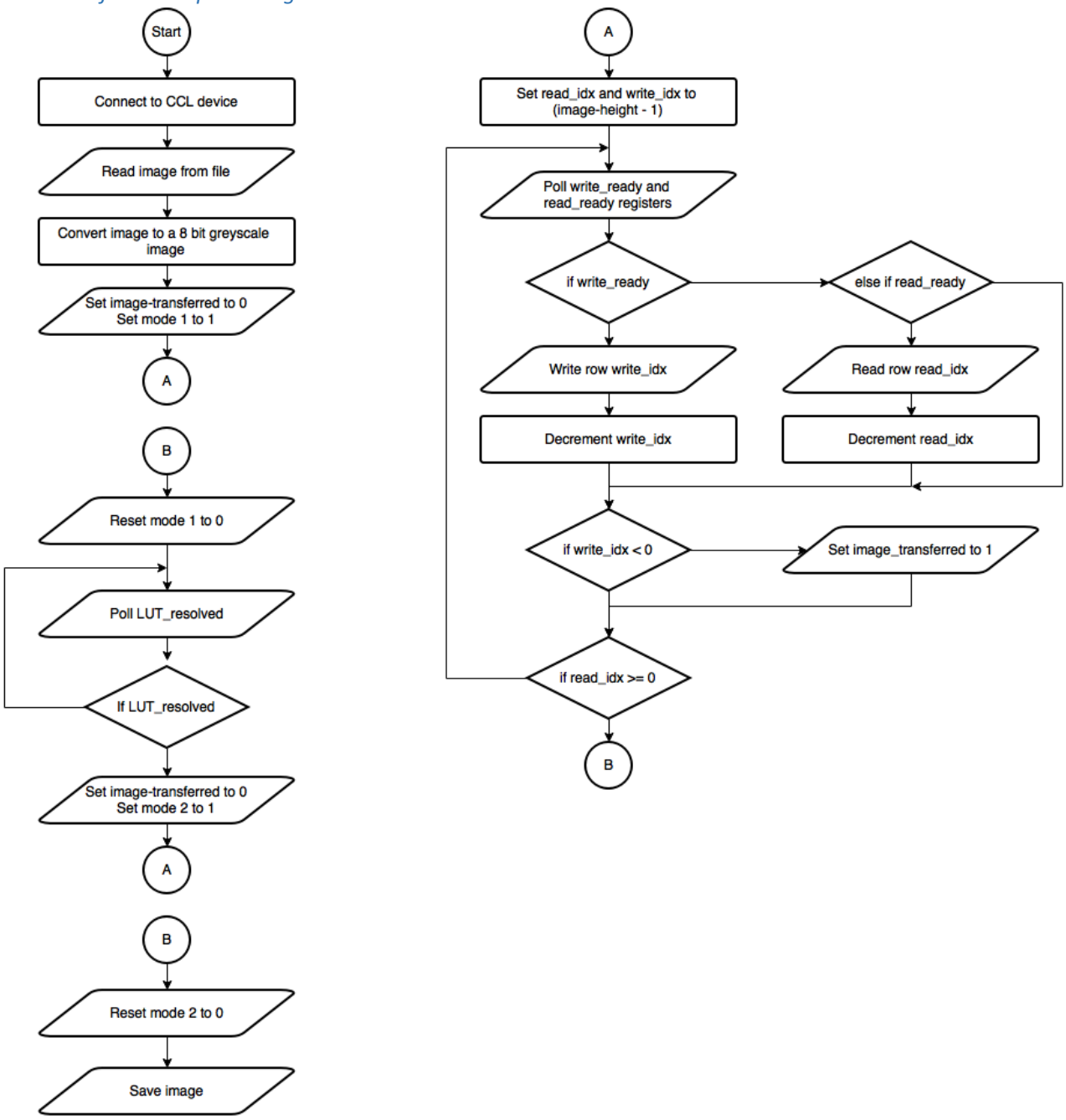
Program Flow

The main function of the user space program is to define the program flow for computing the labels for the connected components in the image and ensure that all the sub-components are provided with the proper input and are called in the proper sequence to compute the labeled image. The key steps of execution in the user space program are as follows:

1. Establish connection with the CCL device through the device driver
2. Read the input image's pixel data from the image file
3. Convert the image to an 8 bit greyscale image
4. Write 0 to the address corresponding to the image--transferred register to clear the previous value of the register
5. Write 1 to the address corresponding to the mode1 register to notify the CCL device that it has to perform the first stage of processing on the pixel data that we are about to send.
6. Start the image transfer. Since the circular buffer on the device can only store 3 rows of pixel information at a time, we need to keep polling for the write-ready register on the device to see if the circular buffer has empty space that we can write into. The program also keeps polling for the read-ready register to see if the device finished processing a row of image data and is ready to be read. Once a row is read from the device that will clear that row and we can subsequently write more data into the circular buffer.
7. The image transfer continues till all the rows of the image are read back after processing. Once we finish writing all the rows we set the image-transferred register to notify the CCL device that the image data corresponding to the entire image has been written and once it finishes processing that data it can move to the next state of processing.
8. The next state of processing does not require more input from the user space but it needs to be completed before the last stage of the algorithm can be initiated. So we poll the LUT-resolved register from the user space to check if the CCL device has completed the process.
9. Once the LUT-resolved bit is set to high we reset image transferred back to 0 and set mode 2 to 1. This initiates the last stage of processing for the connected component labelling.

10. The image for the final stage is then transferred and the processed image is read back in the same way described above. The processed image is the result of the connected component labelling operation and is saved as the result image using the FreeImage library's API.

Flowchart for User Space Program



Device Driver

A device driver is a program that allows us to control a particular device. We build a device driver for our CCL device and load it on to the Linux kernel so that our user space program knows how to interact with the device. Our device driver for the CCL device uses only ioctl commands to send both commands and data between the user space program and the device. The CCL device is treated as a char device with resource registers that a user space program can directly write into or read from.

We define our Connected Component Labeling (CCL) device to have 2048 registers of size one byte each which is addressed by a 11 bit address bus. Our user space program initiates all communication between the user application and the CCL device; the CCL device does not have the ability to interrupt the user space program. Any status change on the CCL device is identified by the user space application through polling. We use the address values ranging from 0 to 640 while sending pixel data corresponding to a particular row in an image to the device driver. In this case the column index of a pixel will correspond to the address to which the pixel's value is sent. The CCL device internally adjusts the address by adding offsets to ensure that the image data is saved in the correct locations on the circular buffer. We use some of the remaining registers for sending commands and communicating status changes as the program executes.

In order to enable us to send and receive data in this fashion we implement four ioctl commands in our device driver. The four ioctl commands are as follows:

CCL_WRITE_REG:

This command is used to write a user specified 32bit unsigned integer value to a user specified address. We use this command to send instructions or status notifications to CCL device. For example, we use this command when we want to instruct the CCL device to start running the first stage of the algorithm to compute the connected component labels.

CCL_READ_REG:

This command is used to read a 32bit unsigned integer value that is represented a user specified address. This command is used to poll for status changes in the CCL device. For example, we use this command to poll for the write-ready register on the CCL device. The register denotes whether or not the circular buffer in the CCL device has empty space into which the user space program can write a row of pixel data.

CCL_WRITE_DATA:

This command writes an entire row of pixel data from the user space program to the CCL device. This command is used to write the image data that needs to be processed by the CCL device.

CCL_READ_DATA:

This command reads an entire row of pixel data from the CCL device to the user space program. This command is used to read the image data back to the user space after it has been processed by the CCL device.

FPGA

Avalon MM bus

In order for the device driver to successfully interact with the FPGA modules, an Avalon memory mapped interface has to be implemented. The Avalon provides a set of predefined interfaces that are mapped to the user-created interfaces. By means of this set of Avalon interfaces, the device driver and the FPGA modules can communicate over the Avalon bus in a timely and reliable manner.

Implementation of the Avalon MM bus and interfaces for the CCL project

Altera Quartus's Qsys tool provides the framework over which the interconnection between

Upon configuring the Avalon on Qsys, hardware and software implications have to be carefully considered. In a nutshell, these main considerations are:

- Communication between kernel software and hardware modules has to be enabled in both ways

- Since for our case, the kernel software will be providing the input image pixel information, and after the final pass, the processed labels will be read out back to the software for the image displaying, the kernel software on one end of the Avalon bus will be designated as Master, thus leaving the FPGA modules end as Slave

- Upon performing read or write operations, the Avalon Bus will introduce control assertion signals that have to be properly managed by the FPGA modules.

The designated Avalon MM interfaces for our design are: (M->S or S->M designates what end initiates the transaction)

- Clock.- carries the clock signal

- Reset.- a mandatory input signal (M->S)

- Chipselect.- a mandatory input signal that is always asserted on write and read transactions

- Writedata.- Carries the data to be written into the FPGA modules. We set it up to be 32-bit wide to allow 4-byte long transactions (M->S)

- Readdata.-Carries the data to be read out from the FPGA to the software. It has the same width as writedata. (S->M)

- Write, read.- assertion signals sent along with writedata and readdata respectively (M->S)

Since our modules will not be interacting with other peripherals in the FPGA, our design does not make use of conduit signals.

RAM

In order to store the data coming from the kernel as well as from other hardware modules in the FPGA like the circular buffer and the CCL device, a RAM with the following characteristics was implemented. The following considerations were taken into account for our project RAM design:

- 2096 byte-size memory address locations
- Single clock port, Dual write enable port
- Different data and address bus sizes: Port one data bus is 8 bit long while Port two data bus is 32 bit long
- Similarly, the address bus sizes of both ports are accordingly adjusted.

The purpose behind the utilization of a Dual port RAM is the transfer time optimization. Indeed, over the 32 bit wide data bus, 4 bytes can be transferred at once. In contrast, port one can transfer only 1 byte at a time. The width of port one has to necessarily be 8 bit long because the CCL algorithm required bit wise computation.

Circular Buffer

Primary interface module between the kernel software and the FPGA hardware. The circular buffer is a component that will temporarily

The reason for implementing a circular buffer in our project stems from two facts>

- Limited memory capacity in the FPGA: a full image cannot be stored at once in memory
- Limited bus size: Even if the FPGA memory would hold a full image, the rate at which an image would be transferred could not be larger than 4 bytes every clock cycle.

The main purpose of the circular buffer is to ensure that data that the CCL needs to be processed is updated as new data is written in and processed data is read out.

The design is as follows:

- The capacity of the circular buffer is 3 pixel rows each of them equal to the size in bytes of a full image row (in our, case 640 bytes). Each byte-long pixel will be stored in a RAM single address location
- With a 3-row capacity, and a CCL device requiring 2 pixel rows to conduct the pixel labeling operation, the Circular buffer must be able to ensure that new row pixel information is available and ready for the CCL device to process.
- When a row has been processed by the CCL device, the circular buffer will “scroll’ making the remaining row available for the CCL to start processing
- Accordingly, when a row is processed, the circular buffer will notify the kernel via Avalon bus that that particular row is ready to be read out, so that that same row can be cleared out and become available for new pixel data.

FPGA implementation of the circular buffer

Based on the design explained in the previous section, the circular buffer was implemented in the following initial conditions (upon the start of the processing of a new image).

- The three rows of the circular buffer are referred to as row 0, 1 and 2 respectively. Each of the 640 pixels of each row will be written into or read out from a specific memory address in the RAM. Since the address coming from the Avalon bus run from 0 to 159 and carry 4 bytes at a time, the circular buffer will account for a suitable row memory address offset. This offset ensures that the circular buffer know where each pixel of each corresponding row is being stored to.

- At the beginning, the circular buffer is empty. Therefore, the “row_filled 0, 1 and 2” signals are set to low.

- Whenever the circular buffer has empty rows, the natural behavior of the circular buffer will be to notify the kernel software that it is ready to be written into by asserting the “write_ready” signal. At the beginning, since the circular buffer is empty, “write_ready” is set to 1.

- Similarly, at the beginning, the CCL device has not yet processed any row; therefore, the “row_processed 0, 1 nd 2” signals are set to low. Consequently, since there is no processed row at this time, there is nothing that the kernel software can read, therefore the circular buffer keeps the “read_ready” signal to low.

- The circular buffer will at all times inform the CCL device of which rows it can work with. Initially, the “active_row” is the second row of the circular buffer (row 1) and the “top_row” will be the first row (row 0). The CCL device will know when to start reading from the circular buffer when the data is ready. Therefore, at the beginning, the “data_ready” signal is set to low.

- When either the “write_ready” or “read_ready” signals are activated, the kernel software will initiate a transaction. The circular buffer must be able to produce the right address at the RAM address bus so that the software can reliably read or write into the right location. The row number that the kernel will have to write into or read from are kept by “write_row” or “read_row” signals respectively. At the beginning, the first row that the kernel will read or write is row 0

Upon starting the circuit operation:

- The kernel drive will write into the circular buffer rows one by one until filling it up. As soon as the kernel fills 2 out of the 3 rows, the “data_ready” signal will be activated, enabling the CCL to start processing. The “row_filled” signals will be set to high and the “write_row” signal will be incremented accordingly. When the kernel finishes writing into the third row, the circular buffer will be filled up, and “write_ready” will be set to low.

- Things will remain in this state until the CCL device processes row 0. Upon this happening, “row_processed 0” will be activated, “read_ready” will also be activated, “active_row” and “top_row” will be increased by one and “data_ready” will be temporarily set back to low. This will trigger the beginning of a read process on behalf of the kernel which will start reading the processed row out of the circular buffer. If a next row is filled, “data_ready” will be set back to 1 in the very next clock cycle.

- When the reading out transaction is completed, “read_ready”, “row_filled” and “row_processed” will be set back to low, and “read_row” will be incremented.

This behavior will continue regularly until the whole image is transferred, processed and read back. The circular will also handle additional signaling including

- The current operation mode (1 or 2 depending if the CCL is in the first or second pass)
- An “image transferred” signal informing the CCL device
- A “Look up table complete” signal informing the kernel software
- The current status of the “write ready” and “read ready” to keep the kernel updated.

These signals will be polled by the kernel software to circular buffer registers by using a number of unutilized address locations.

Connected component labeling on FPGA

Connected component labeling assigns a unique label to each set of connected pixels. The algorithm involves thresholding a grayscale image to obtain a binary image and two subsequent labeling steps. In the first labeling step, temporary labels are assigned to each pixel and equivalences are recorded in a look-up table. After the first pass for the image is over, equivalences are resolved and final labels are assigned in the second pass.

Details of Connected Component Labeling algorithm:

Chose a threshold Th to convert the input grayscale image into a binary image.

Compare the value of each pixel with Th.

If I [x,y] > Th then I [x,y] = 255, foreground pixel

Else I [x,y] = 0, background pixel

Define a 3-pixel neighborhood.

Top Left	Top
Current Left	Current

Scan the image row-wise from left to right.

In the first pass, in case C = foreground pixel / I [x,y] = 255:

If none of its neighbors are labelled, then assign a new label to C.

If top left pixel TL has a label L(TL), then assign the same label to C.

If top left is 0, and either left or top has a label, then assign that label to C.

If both top and left have a label, assign L(C) = L(T) and enter equivalence in the LUT as LUT [L(CL)] = L(T).

0	0
0	$L_{(New)}$

L(TL)	X
X	L(TL)

0	0
L(CL)	L(CL)

0	L(T)
L(CL)	L(C)

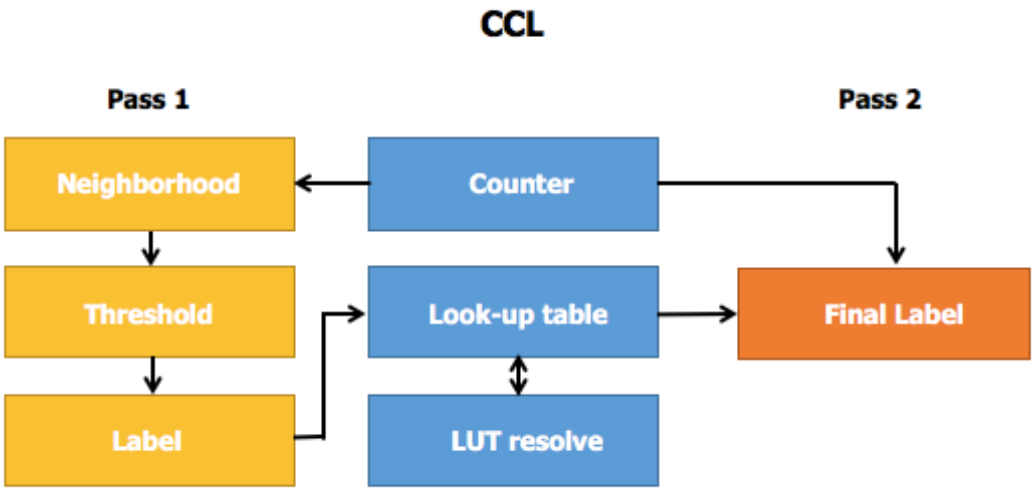
In the second pass, find the smallest label for each set of equivalences and replace each label in the image with the corresponding smallest label.

Look-up table label resolution:

LUT is resolved by traversing from top to bottom and resolving equivalences as we go.

Index	Label		Resolution:
0	0		0 -> 0 Next
1	1		1 -> 1 Next
2	1		2 -> 1 -> 1 Next
3	2	1	3 -> 2 -> 1 Change label to 1
4	2	1	4 -> 2 -> 1 Change label to 1
5	4	1	5 -> 4 -> 1 Change label to 1
6	6		6 -> 6

Algorithm implementation on FPGA:



Following modules are defined to implement the algorithm:

CCL (connected component labeling) module: This is the top module which links and controls the modules for pass 1, pass 2 and look-up table. It also communicates with the buffer memory to read and write pixel values.

Once two unprocessed rows of pixels are available in the buffer, the data is ready to be processed. It receives indices of top and active rows from the circular buffer as well as the mode in which it has to operate (pass 1 or pass 2).

For the first/second pass, it fetches the pixel value of the address specified by pass 1/pass 2 and writes the processed pixel value back to the memory.

This is the module that passes control variables to sub-modules in order to activate and deactivate them. It also sends variables to the memory module that reflect the progress of its sub-modules. At the end of pass 1 for the active row, it sets tells the buffer that current top row is no longer required for further processing. The buffer then scrolls the indices to next pair of rows.

Counter: This is a simple counter module which counts pixel index from 1 to 639 (last pixel of row). It increments whenever enable is high and resets when it reaches the end-of-row. Enable for this module is triggered by two variables, one controlled in mode 1 and other in mode 2.

Pass 1: This module performs the function of populating current and neighborhood pixels. Neighborhood consists of left, top and top left pixels. Pass 1 calculates and sends the address for current pixel to CCL and receives the pixel value after two clock cycles. Next, it does the same for top pixel. Left and top left pixels are assigned values of current and top pixels respectively from the previous neighborhood. It also thresholds the current pixel value using 128 as the limit. Next, it labels the current pixel and sends the value of equivalences to be registered in the LUT.

This pass requires 6 consecutive clock cycles for thresholding and labeling one pixel. At the end on sixth clock cycle, processed value of pixel is sent to RAM with write enable set. A brief overview is presented below:

Cycle 0: At posedge, address for current pixel is assigned to RAM address wire. Left and top left pixels are assigned values from previous neighborhood current and top pixels. Write enable for RAM and LUT are set to 0.

Cycle 1: At posedge, address for top pixel is assigned to RAM and the RAM receives the address assigned in the previous clock cycle.

Cycle 2: In this clock cycle, the RAM reads the value at address assigned in cycle 0, i.e., for current pixel.

Cycle 3: At posedge, current pixel register is assigned its value and the RAM reads the value at the address assigned in cycle 1, i.e., for top pixel. Variable nbd_ready, which is used to set enable counter to high, is set high.

Cycle 4: At posedge, top pixel register is assigned its value and current pixel is thresholded. In this clock cycle, variable nbd_ready is set to low and counter get enabled. RAM address wire gets the value of current pixel as the processed value needs to be written to this address later.

Cycle 5: In this cycle, current pixel is assigned a label depending on its neighbors. In case an equivalence is encountered, index and data for LUT is set appropriately as well as LUT is write enabled. For RAM,

data is set to current pixel value and write enable wire is set high. Counter is also incremented by 1 in this clock cycle.

Pass 1 repeats until we reach the end of one row. Upon reaching the end, CCL module receives mode1 signal as 0. This in turn disables pass 1.

Look-up table: This module is a memory of size 256 and holds 8 bit values. The address represents 8-bit label value. The data stored is the label equivalent to the address label. In pass 1, equivalences are entered into the LUT and in the next module, LUT_resolve, equivalences are resolved and written back to LUT. Eventually, in pass 2, for each pixel value, the corresponding equivalent label is read.

LUT_resolve: This module resolves the equivalences such that all labels with direct or transitive equivalences get the smallest label in the equivalent set. The module loops through the LUT and for each label it traverses through all the linked labels until it encounters the case when index and label are equal. That label is assigned to the root label. This module is operational when variable resolve_en is high and mode 1 is set to 0.

Pass 2: This module becomes active after LUT is resolved and mode 2 signal becomes active. For each pixel, this module takes 6 clock cycles. A brief overview is presented below:

Cycle 0: Address for current pixel to be processed is assigned to RAM wire.

Cycle 1 and 2: The RAM gets the address and reads the data at that address.

Cycle 3: Index for LUT is assigned as the value read from RAM and variable p2_ready is set for counter to be enabled.

Cycle 4: Variable p2_ready is set to 0 and enable for counter module goes high. LUT value at index is read into dout for LUT.

Cycle 5: The label value from LUT is assigned to RAM data wire and write is enable for RAM.

Contribution

The user space program and the device driver was written by Avinash Nair, the implementation of the Circular Buffer and the interface between the device driver and the Connected Component Labelling module was written by Jerry Barona. Manushree Gangwar wrote the code for implementing connected component labelling including the submodules used by the connected component labelling module.

All three project members were involved in debugging of all modules and getting the final output for the system. All the members also contributed to all the documentation including the proposal, design documents and final project report.

CODE

Filename: ccl_device.sv

```
	/*
 * Circular Buffer
 *
 * Embedded Systems Project
 * Columbia University
 */

module circ_buffer(
		input logic 			clk,
		input logic 			reset,
		input logic [31:0] 	data_b,
		input logic [8:0] 	address,
		input 				chipselect,
		input logic 			read,
		input logic 			write,
		output logic [31:0] 	readdata);

	logic [7:0] 		data_a;
	logic [10:0] 		address_a;
	logic [8:0] 		address_b;
	logic [7:0] 		q_a;
	logic [31:0] 		q_b;

	logic 			write_ready; //lets the device driver know when
to write into RAM
	logic 			read_ready; // lets the device driver know when
to read from RAM

	logic [1:0] 		active_row; // informs the algorithm which row to work with
	logic [1:0] 		top_row; // informs the algorithm which row to work with
	logic 			data_ready;
	logic [1:0] 		read_row;
	logic [1:0] 		write_row;

  logic 			wren_a;
  logic 			wren_b;

	//logic 			row_filled0, row_filled1, row_filled2;
	logic 			row_filled[2:0];
	logic 			row_processed0, row_processed1,
row_processed2;
	logic 			reading_row0, reading_row1, reading_row2;
	logic 			row_processed[2:0];
	logic 			read_toggle;

	logic 			mode1;
	logic 			mode2;
	logic 			image_transferred;
	logic [7:0] 		max;
	logic 			LUT_complete;
```

```
    logic [7:0]                         LUT_add, LUT_val;

parameter ROWADDRESSOFFSET      = 9'd160;

RAM_cc RAM_cc_instance(
                .clock(clk),
                .aclr(reset),
                .address_a(address_a),
                .address_b(address_b),
                .data_a(data_a),
                .data_b(data_b),
                .wren_a(wren_a),
                .wren_b(wren_b),
                .q_a(q_a),
                .q_b(q_b) );


    ccl ccl_instance(
                .clk(clk),
                .data_ready(data_ready),
                .ACTIVE_ROW(active_row),
                .TOP_ROW(top_row),
                .mode1(mode1),
                .mode2(mode2),
                .row_processed0(row_processed0),
                .row_processed1(row_processed1),
                .row_processed2(row_processed2),
                .reading_row0(reading_row0),
                .reading_row1(reading_row1),
                .reading_row2(reading_row2),
                .LUT_COMPLETE(LUT_complete),
                .image_transferred(image_transferred),
                .max(max),
                .q_a(q_a), //RAM
                .address_a(address_a), //RAM
                .data_a(data_a), //RAM
                .wren_a(wren_a),
                .LUT_add(LUT_add),
                .LUT_val(LUT_val)); //RAM



    // Initial assignments
    initial begin
       active_row = 2'd1; // initially, there's no filled row to process
       top_row = 2'd0; // initially, the third row will serve as the zero-padded top row
       //data_ready = 1'b0;
       write_row = 2'd0; // initially, begin by writing into the first row
       read_row = 2'd0; // initially, there's no row to read

       row_filled = '{1'b0, 1'b0, 1'b0};
       //row_processed = '{1'b0, 1'b0, 1'b0};

       reading_row0 = 1'b0;
       reading_row1 = 1'b0;
       reading_row2 = 1'b0;
```

```
    //write_ready = 1'b1;
    //read_ready = 1'b0;

    mode1 = 1'b0;
    mode2 = 1'b0;
    read_toggle = 1'b0;
    image_transferred = 1'b0;
    //LUT_complete = 1'b0;
  end

  //
  always_comb begin
    LUT_add = 0;
    readdata = q_b;
    wren_b = write && chipselect;
    address_b = 9'b111111111;

    row_processed[0] = row_processed0;
    row_processed[1] = row_processed1;
    row_processed[2] = row_processed2;

    read_ready = (row_processed[0] || row_processed[1] || row_processed[2]);

    write_ready = ~(row_filled[0] && row_filled[1] && row_filled[2]);

    data_ready = (row_filled[active_row] && row_filled[top_row] && (~row_processed[active_row]));

    if(address < ROWADDRESSOFFSET) begin
      if(read) begin
                    readdata = q_b;
                    address_b = read_row * ROWADDRESSOFFSET + address;
            end
            if(write) begin
                    address_b = write_row * ROWADDRESSOFFSET + address;
            end
    end else if (address == 9'd184) begin
            if(read) begin
                    readdata = {31'd0, write_ready};
            end
    end else if (address == 9'd185) begin
            if(read) begin
                    readdata = {31'd0, read_ready};
            end
    end else if (address == 9'd186) begin
            if(read) begin
                    readdata = {31'd0, LUT_complete};
            end
    end else if (address == 9'd187) begin
            if(read) begin
                    readdata = {30'd0, write_row};
            end
    end else if (address == 9'd188) begin
            if(read) begin
                    readdata = {30'd0, read_row};
            end
```

```
end else if (address == 9'd189) begin
        if(read) begin
                readdata = {31'd0, row_filled[0]};
        end
end else if (address == 9'd190) begin
        if(read) begin
                readdata = {31'd0, row_filled[1]};
        end
end else if (address == 9'd191) begin
        if(read) begin
                readdata = {31'd0, row_filled[2]};
        end
end else if (address == 9'd192) begin
        if(read) begin
                readdata = {31'd0, row_processed[0]};
        end
end else if (address == 9'd193) begin
        if(read) begin
                readdata = {31'd0, row_processed[1]};
        end
end else if (address == 9'd194) begin
        if(read) begin
                readdata = {31'd0, row_processed[2]};
        end
end else if (address == 9'd195) begin
        if(read) begin
                readdata = {31'd0, reading_row0};
        end
end else if (address == 9'd196) begin
        if(read) begin
                readdata = {31'd0, reading_row1};
        end
end else if (address == 9'd197) begin
        if(read) begin
                readdata = {31'd0, reading_row2};
        end
end else if (address == 9'd198) begin
        if(read) begin
                readdata = {31'd0, data_ready};
        end
end else if (address == 9'd199) begin
        if(read) begin
                readdata = {30'd0, active_row};
        end
end else if (address == 9'd200) begin
        if(read) begin
                readdata = {30'd0, top_row};
        end
end else if (address == 9'd201) begin
        if(read) begin
                readdata = {24'd0, max};
        end
end else if (address > 9'd255 && address <= 9'd511) begin
        if(read) begin
                LUT_add = address[7:0];
                readdata = {24'd0, LUT_val};
```

```
            end
    end
end

always_ff @(posedge clk) begin

    if (row_processed[top_row]) begin
            if (active_row == 2'd0) begin
                    active_row <= 2'd1;
                    top_row <= 2'd0;
            end else if(active_row == 2'd1) begin
                    active_row <= 2'd2;
                    top_row <= 2'd1;
            end else if (active_row == 2'd2) begin
                    active_row <= 2'd0;
                    top_row <= 2'd2;
            end
    end else begin
            active_row <= active_row;
            top_row <= top_row;
    end

    if(address == 9'd180) begin
            if(write) begin
                    mode1 <= data_b[0];
                    active_row <= 2'd1; // initially, there's no filled row to process
                    top_row <= 2'd0; // initially, the third row will serve as the zero-padded top row
                    //data_ready <= 1'b0;
                    write_row <= 2'd0; // initially, begin by writing into the first row
                    read_row <= 2'd0;
                    row_filled <= '{1'b0, 1'b0, 1'b0};
                    //row_processed <= '{1'b0, 1'b0, 1'b0};
                    read_toggle <= 1'b0;
                    image_transferred <= 1'b0;
                    reading_row0 <= 1'b0;
                    reading_row1 <= 1'b0;
                    reading_row2 <= 1'b0;
                    //LUT_complete <= 1'b0;
            end
    end else if (address == 9'd181) begin
            if(write) begin
                    mode2 <= data_b[0];
                    active_row <= 2'd1; // initially, there's no filled row to process
                    top_row <= 2'd0; // initially, the third row will serve as the zero-padded top row
                    //data_ready <= 1'b0;
                    write_row <= 2'd0; // initially, begin by writing into the first row
                    read_row <= 2'd0;
                    row_filled <= '{1'b0, 1'b0, 1'b0};
                    //row_processed <= '{1'b0, 1'b0, 1'b0};
                    read_toggle <= 1'b0;
                    image_transferred <= 1'b0;
                    reading_row0 <= 1'b0;
                    reading_row1 <= 1'b0;
                    reading_row2 <= 1'b0;
            end
    end else if (address == 9'd182) begin
```

```
                if(write) begin
                        image_transferred <= data_b[0];
                end
        end

        if(write) begin
                if(address == ROWADDRESSOFFSET - 9'b1) begin
                        row_filled[write_row] <= 1'b1;
                        if (write_row == 2'd0) begin
                                write_row <= 2'd1;
                        end else if(write_row == 2'd1) begin
                                write_row <= 2'd2;
                        end else if (write_row == 2'd2) begin
                                write_row <= 2'd0;
                        end
                end
        end

        if (read && ~read_toggle) begin
                if(address == 0) begin
                        row_filled[read_row] <= 1'b0;
                        if(read_row == 2'd0) begin
                                reading_row0 <= 1'b1;
                                reading_row1 <= 1'b0;
                                reading_row2 <= 1'b0;
                        end else if(read_row == 2'd1) begin
                                reading_row0 <= 1'b0;
                                reading_row1 <= 1'b1;
                                reading_row2 <= 1'b0;
                        end else if(read_row == 2'd2) begin
                                reading_row0 <= 1'b0;
                                reading_row1 <= 1'b0;
                                reading_row2 <= 1'b1;
                        end
                end else if(address == ROWADDRESSOFFSET - 9'b1) begin
                        read_toggle <= 1'b1;
                        reading_row0 <= 1'b0;
                        reading_row1 <= 1'b0;
                        reading_row2 <= 1'b0;
                        //row_processed[read_row] <= 1'b0;
                        if (read_row == 2'd0) begin
                                read_row <= 2'd1;
                        end else if(read_row == 2'd1) begin
                                read_row <= 2'd2;
                        end else if (read_row == 2'd2) begin
                                read_row <= 2'd0;
                        end
                end
        end

        if(~read) begin
                read_toggle <= 1'b0;
        end
      end
endmodule
```

Filename: ccl.sv

```
// CSEE 4840L Connected component labeling

//Top Module: CCL - Communicates with the memory buffer and connects the submodules.
module ccl (input logic clk,
                        input data_ready,
                        input logic [1:0] ACTIVE_ROW,
                        input logic [1:0] TOP_ROW,
                        input logic mode1, /**for pass 1 operation**/
                        input logic mode2, /**for pass 2 operation**/
                        input logic reading_row0, /**set when row 0 is being read by software**/
                        input logic reading_row1,
                        input logic reading_row2,
                        output logic row_processed0, /**set row 0 has finished pass 1 or pass 2**/
                        output logic row_processed1,
                        output logic row_processed2,
                        output logic LUT_COMPLETE, /**set when LUT equivalences have been
resolved**/
                        input image_transferred, /**set when all image rows have been written on
buffer**/
                        output logic [7:0] max,
                        input logic [7:0] q_a, /**value in RAM**/
                        output logic [10:0] address_a, /**address of RAM**/
                        output logic [7:0] data_a, /**data to be written on RAM**/
                        output logic wren_a, /**write enable for RAM**/
                        input logic [7:0] LUT_add,
                        output logic [7:0] LUT_val
                        );

                        logic [7:0] label_no; /**Starts from 1, total 255 possible labels**/

                        logic [10:0] OFFSET;
                        assign OFFSET = 11'd640; /**Row offset**/

                        logic c_en, c_reset, nbd_ready, resolve_en, p2ready;
                        logic [10:0] count; /**pixel counter**/

                        logic [7:0] curr, curr_l, top, top_l; /**pixel values for neighboring and current
pixel**/

                        logic [7:0]  index, din, dout, p1index, p2index, lindex, lutrout;
                        logic we, p1we, lwe;
                        logic [7:0] p1din, ldin;

                        logic [10:0] p1address_a, p2address_a;
                        logic [7:0] p1data_a, p2data_a;
                        logic p1wren_a, p2wren_a, wren_default;

                        initial begin
                                c_en = 1'b0; //for count enable
                                c_reset = 1'b1; //for count reset
                                resolve_en = 1'b0; //for LUT resolve
                                row_processed0 = 1'b0;
```

```
                row_processed1 = 1'b0;
                row_processed2 = 1'b0;
                max = 1'b0;
        end

        always_ff @(posedge clk) begin
                if (count == 11'd639 && (nbd_ready || p2ready)) begin
                        c_reset <=1'b1;
                        if(TOP_ROW == 2'd0) row_processed0 <= 1'b1; /**Row
processed set when counter reaches 639**/
                        else if(TOP_ROW == 2'd1) row_processed1 <= 1'b1;
                        else if(TOP_ROW == 2'd2) row_processed2 <= 1'b1;
                        else begin
                                row_processed0 <= row_processed0;
                                row_processed1 <= row_processed1;
                                row_processed2 <= row_processed2;
                        end
                end else if (data_ready) begin
                        c_reset <= 1'b0;
                end else begin
                        c_reset <= 1'b1;
                end

                if (reading_row0 == 1'b1) row_processed0 <= 1'b0; /**When software
starts reading row0, row processed is reset**/
                else if (reading_row1 == 1'b1) row_processed1 <= 1'b0;
                else if (reading_row2 == 1'b1) row_processed2 <= 1'b0;

                if (nbd_ready || p2ready) c_en <= 1'b1;
                else c_en <=1'b0;

                if (image_transferred && (~data_ready)) begin
                        max = label_no - 8'd1;
                        resolve_en = 1'b1; /**Set when LUT is ready to be resolved**/
                        if(TOP_ROW == 2'd0) row_processed0 <= 1'b1;
                        else if(TOP_ROW == 2'd1) row_processed1 <= 1'b1;
                        else if(TOP_ROW == 2'd2) row_processed2 <= 1'b1;
                end
                if (LUT_COMPLETE) begin
                        resolve_en = 1'b0;
                end
        end

        always_comb
        begin
                if (mode1 && data_ready) begin
                        index = p1index;
                        we = p1we;
                        din = p1din;
                end
                else if (resolve_en) begin
                        index = lindex;
                        we = lwe;
                        din = ldin;
                end
                else if (mode2 && data_ready) begin
```

```
                                        index = p2index;
                                        we = 1'b0;
                                        din = 8'd0;
                                end
                                else begin
                                        index = LUT_add; /**To print LUT**/
                                        we = 1'b0;
                                        din = 8'd0;
                                end

                                LUT_val = dout; /**To print LUT**/

                                if (mode1 && data_ready)begin
                                        address_a = p1address_a;
                                        data_a = p1data_a;
                                        wren_a = p1wren_a;
                                end
                                else if(mode2 && data_ready) begin
                                        address_a = p2address_a;
                                        data_a = p2data_a;
                                        wren_a = p2wren_a;
                                end else begin
                                        wren_a = 1'b0;
                                        address_a = 11'd2047;
                                        data_a = 8'd0;
                                end
                        end

                        counter c(.*);
                        pass1 p1(.*);
                        lookup l(.*);
                        lut_resolve r(.*);
                        pass2 p2(.*);
endmodule

// - MODULE - COUNTER: pixel index counter
module counter (input logic clk, input logic c_en, input logic c_reset, output logic [10:0] count);

        initial begin
           count = 11'd0;
        end

        always_ff @(posedge clk) begin
           if (c_reset) begin
                        count <= 11'd0;
           end
           else if(c_en) begin
                                count <= count + 11'd1;
           end
        end

endmodule

// MODULE - PASS1: Neighborhood + Threshold + Label 1
module pass1(input logic clk,
                        input logic [10:0] OFFSET,
```

```
            input logic [1:0] ACTIVE_ROW,
            input logic [1:0] TOP_ROW,
            input logic [10:0] count,
            input logic mode1,
            input logic c_reset,
            input logic [7:0] q_a,
            output logic [10:0] p1address_a, /**wire connecting to RAM address**/
            output logic [7:0] p1data_a,
            output logic p1wren_a,
            output logic [7:0] label_no,
            output logic [7:0]  p1index, p1din, /**LUT module inputs**/
    output logic p1we, /**LUT write enable**/
            output logic nbd_ready /**set when all four neighboring pixels are assigned values**/
            );

logic [2:0] i;
logic [7:0] curr, curr_l, top, top_l;

initial begin
        i = 3'd0;
        label_no = 8'd1;
        nbd_ready = 1'b0;
        p1wren_a = 1'b0;
        p1we = 1'b0;
        curr = 8'd0;
        top = 8'd0;
        curr_l = 8'd0;
        top_l = 8'd0;
end

always_ff @(posedge clk) begin
        if (mode1 && ~c_reset) begin /**Enables pass 1 operations**/
                if (i == 3'd0) begin
                        p1we = 1'b0;
                        p1wren_a = 1'b0;
                        p1address_a = ACTIVE_ROW*OFFSET + count;
                        curr_l = curr;
                        top_l = top;
                        i++;
                end

                else if (i == 3'd1) begin
                        p1address_a = TOP_ROW*OFFSET + count;
                        i++;
                end

                else if (i == 3'd2) begin
                        i++;
                end

                else if (i == 3'd3) begin
                        curr = q_a;
                        nbd_ready = 1'b1;
                        i++;
                end
```

```
                else if (i == 3'd4) begin
                        i++;
                        top = q_a;
                        /**Threshold = 128**/
                        if (curr[7] == 1'b1) curr = 8'd255; /**Foreground**/
                        else curr = 8'd0; /**Background**/
                        p1address_a = ACTIVE_ROW*OFFSET + count;
                        nbd_ready = 1'b0;
                end

                else begin
                        i = 3'b0;
                        if (curr == 8'd255) begin
                                if (top_l != 8'd0) curr = top_l;
                                else if (top != 8'd0 && top_l == 8'd0 && curr_l == 8'd0) curr = top;
                                else if (curr_l != 8'd0 && top_l == 8'd0 && top == 8'd0 ) curr =
curr_l;
                                else if (top_l == 8'd0 && top != 8'd0 && curr_l != 8'd0 ) begin
                                  /**For LUT module using approach 1**/
                                        curr = top;
                                  p1index = curr_l;
                                  p1din = top;
                                        p1we = 1'b1; /**Write in LUT at p1index to update label
**/

                                  /**For LUT module using approach 2**/
                                  /*if (top < curr_l) begin
                                                curr = top;
                                                p1index = curr_l;
                                                p1din = top;
                                                p1we = 1'b1; //write din in LUT at p1index
                                  /*end
                                  else if (curr_l < top) begin
                                                curr = curr_l;
                                                p1index = top;
                                                p1din = curr_l;
                                                p1we = 1'b1; //write din in LUT at p1index
                                  end
                                  else curr = top;*/
                                end
                                else begin
                                        curr = label_no;
                                        p1index = label_no;
                                        p1din = label_no;
                                        p1we = 1'b1;
                                        label_no++;
                                end
                                p1data_a = curr; /**Value to write in RAM for current pixel**/
                                p1wren_a = 1'b1; /**RAM write enabled**/
                        end
                        else begin
                                p1data_a = curr;
                                p1wren_a = 1'b1;
                        end
                end
        end
```

```
                else if (mode1) begin
                        i = 3'd0;
                        nbd_ready= 1'b0;
                        p1wren_a = 1'b0;
                        p1we = 1'b0;
                end

                else begin /**Reset condition**/
                        i = 3'd0;
                        label_no = 8'd1;
                        nbd_ready = 1'b0;
                        p1wren_a = 1'b0;
                        p1we = 1'b0;
                end
        end

endmodule

//MODULE - LOOK-UP: Stores equivalences encountered during pass 1
module lookup(input logic  clk,
              input logic [7:0]  index,
              input logic [7:0]  din,
              input logic        we,
              output logic [7:0] dout);

  logic [7:0] LUT[255:0]; /**LUT memory**/

      initial begin
        for(int i =0; i < 256; i++) begin
                LUT[i] = 8'b0;
        end
      end

  always_ff @(posedge clk) begin
    if (we) LUT[index] <= din;
    dout <= LUT[index];
  end
endmodule

//MODULE - LUT_RESOLVE: Resolving equvalences
module lut_resolve (input logic clk,
                input logic mode1,
                input logic [7:0] max,
                input resolve_en,
            input logic [7:0] dout,
                output logic [7:0]  lindex,
                output logic LUT_COMPLETE,
                output logic lwe,
                output logic [7:0] ldin);

        logic [7:0] loopindex, label_index;
        logic [4:0] c;

        initial begin
                lwe = 1'b0;
                ldin = 8'd0;
```

```
                LUT_COMPLETE = 1'b0;
                lindex = 8'd0;
                loopindex = 8'd1;
                c = 3'd0;
                label_index = 8'd0;
        end

        /**Iterate through the LUT after pass1 one to merge equivalent labels**/

        always_ff @(posedge clk) begin

          /**The module is run when resolve_en is set and the program has exited mode 1**/
                /**Iterates through the lookup table to find the final label**/
                if(resolve_en && loopindex <= max && (~mode1)) begin
                        if(c == 3'd0) begin
                                lwe = 1'b0;
                                lindex = loopindex;
                                c = 3'd1;
                        end else if(c == 3'd1) begin
                                c = 3'd2;
                        end else if(c == 3'd2) begin
                                if(dout == lindex) begin
                                        ldin = dout;
                                        lindex = loopindex;
                                        lwe = 1'b1;
                                        loopindex = loopindex + 1'b1;
                                        c = 3'd0;
                                end else begin
                                        lindex = dout;
                                        c = 3'd1;
                                end
                        end
                end
                /**Set LUT_COMPLETE once all equivalences have been resolved**/
                else if (resolve_en && (~mode1)) begin
                        LUT_COMPLETE = 1'b1;
                        lwe = 1'b0;
                        ldin = 8'd0;
                end
                else if (mode1) begin
                        LUT_COMPLETE = 1'b0;
                        lwe = 1'b0;
                        ldin = 8'd0;
                        loopindex = 8'd1;
                        lindex = 8'd0;
                        c = 3'd0;
                end

        end
endmodule


// MODULE - PASS2
module pass2 (input logic clk,
                        input logic mode2,
                        input logic [10:0]          OFFSET,
```

```
		input logic [1:0] ACTIVE_ROW,
	input logic [10:0] count,
	input logic c_reset,
	output logic [10:0] p2address_a,
	input logic [7:0] q_a,
	output logic [7:0] p2data_a,
	output logic p2wren_a,
	output logic [7:0]  p2index, /**LUT index**/
		input logic [7:0] dout, /**LUT output**/
		output logic p2ready /**Triggers c_en for count**/
);

		logic [2:0] i;

		initial begin
			i = 3'd0;
			p2wren_a = 1'b0;
			p2address_a = 11'd0;
			p2data_a = 8'd0;
			p2ready = 1'b0;
			p2index = 8'd0;
		end

		always_ff @(posedge clk)
		begin
			if (mode2 & ~c_reset) begin
				if (i == 3'd0) begin
					p2address_a <= ACTIVE_ROW*OFFSET + count;
					i <= 3'd1;
					p2wren_a <= 1'b0;
				end

				else if (i == 3'd1) begin
					i <= 3'd2;
				end

				else if (i == 3'd2) begin
					i <= 3'd3;
				end

				else if (i == 3'd3) begin
					i <= 3'd4;
					p2index <= q_a; /**Assigns the pixel value in RAM to
LUT index**/
					p2ready <= 1'b1;
				end

				else if (i == 3'd4) begin
					i <= 3'd5;
					p2ready <= 1'b0;
				end

				else if (i == 3'd5) begin
					if (dout != 8'd0) begin
						p2data_a <= dout;
					end
```

```
                                        else begin
                                                p2data_a <= p2index; /**Assigns the LUT index
to pixel value in RAM**/
                                        end
                                        p2wren_a <= 1'b1;
                                        i <= 3'd0;
                                end
                        end else begin
                                i = 3'd0;
                                p2wren_a = 1'b0;
                                p2address_a = 11'd0;
                                p2data_a = 8'd0;
                                p2ready = 1'b0;
                                p2index = 8'd0;
                        end
                end
endmodule
```

Filename: RAM_cc.v

```
// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: RAM_cc.v
// Megafunction Name(s):
//                  altsyncram
//
// Simulation Library Files(s):
//                  altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 13.1.1 Build 166 11/26/2013 SJ Full Version
// ************************************************************


//Copyright (C) 1991-2013 Altera Corporation
//Your use of Altera Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Altera Program License
//Subscription Agreement, Altera MegaCore Function License
//Agreement, or other applicable license agreement, including,
//without limitation, that your use is for the sole purpose of
//programming logic devices manufactured by Altera and sold by
//Altera or its authorized distributors.  Please refer to the
//applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module RAM_cc (
        aclr,
        address_a,
        address_b,
        clock,
        data_a,
        data_b,
        wren_a,
        wren_b,
        q_a,
        q_b);
```

```
	input		aclr;
	input	[10:0]	address_a;
	input	[8:0]	address_b;
	input		clock;
	input	[7:0]	data_a;
	input	[31:0]	data_b;
	input		wren_a;
	input		wren_b;
	output	[7:0]	q_a;
	output	[31:0]	q_b;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
	tri0		aclr;
	tri1		clock;
	tri0		wren_a;
	tri0		wren_b;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

	wire [7:0] sub_wire0;
	wire [31:0] sub_wire1;
	wire [7:0] q_a = sub_wire0[7:0];
	wire [31:0] q_b = sub_wire1[31:0];

	altsyncram	altsyncram_component (
				.clock0 (clock),
				.wren_a (wren_a),
				.address_b (address_b),
				.data_b (data_b),
				.wren_b (wren_b),
				.aclr0 (aclr),
				.address_a (address_a),
				.data_a (data_a),
				.q_a (sub_wire0),
				.q_b (sub_wire1),
				.aclr1 (1'b0),
				.addressstall_a (1'b0),
				.addressstall_b (1'b0),
				.byteena_a (1'b1),
				.byteena_b (1'b1),
				.clock1 (1'b1),
				.clocken0 (1'b1),
				.clocken1 (1'b1),
				.clocken2 (1'b1),
				.clocken3 (1'b1),
				.eccstatus (),
				.rden_a (1'b1),
				.rden_b (1'b1));
```

```
        defparam
            altsyncram_component.address_reg_b = "CLOCK0",
            altsyncram_component.clock_enable_input_a = "BYPASS",
            altsyncram_component.clock_enable_input_b = "BYPASS",
            altsyncram_component.clock_enable_output_a = "BYPASS",
            altsyncram_component.clock_enable_output_b = "BYPASS",
            altsyncram_component.indata_reg_b = "CLOCK0",
            altsyncram_component.intended_device_family = "Cyclone V",
            altsyncram_component.lpm_type = "altsyncram",
            altsyncram_component.numwords_a = 2048,
            altsyncram_component.numwords_b = 512,
            altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
            altsyncram_component.outdata_aclr_a = "CLEAR0",
            altsyncram_component.outdata_aclr_b = "CLEAR0",
            altsyncram_component.outdata_reg_a = "CLOCK0",
            altsyncram_component.outdata_reg_b = "CLOCK0",
            altsyncram_component.power_up_uninitialized = "FALSE",
            altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
            altsyncram_component.read_during_write_mode_port_a =
"NEW_DATA_NO_NBE_READ",
            altsyncram_component.read_during_write_mode_port_b =
"NEW_DATA_NO_NBE_READ",
            altsyncram_component.widthad_a = 11,
            altsyncram_component.widthad_b = 9,
            altsyncram_component.width_a = 8,
            altsyncram_component.width_b = 32,
            altsyncram_component.width_byteena_a = 1,
            altsyncram_component.width_byteena_b = 1,
            altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0";


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
// Retrieval info: PRIVATE: CLRdata NUMERIC "0"
// Retrieval info: PRIVATE: CLRq NUMERIC "1"
// Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
```

// Retrieval info: PRIVATE: CLRrren NUMERIC "0"
// Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
// Retrieval info: PRIVATE: CLRwren NUMERIC "0"
// Retrieval info: PRIVATE: Clock NUMERIC "0"
// Retrieval info: PRIVATE: Clock_A NUMERIC "0"
// Retrieval info: PRIVATE: Clock_B NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MEMSIZE NUMERIC "16384"
// Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING ""
// Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "3"
// Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "1"
// Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "1"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
// Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
// Retrieval info: PRIVATE: REGdata NUMERIC "1"
// Retrieval info: PRIVATE: REGq NUMERIC "1"
// Retrieval info: PRIVATE: REGrdaddress NUMERIC "0"
// Retrieval info: PRIVATE: REGrren NUMERIC "0"
// Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
// Retrieval info: PRIVATE: REGwren NUMERIC "1"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
// Retrieval info: PRIVATE: VarWidth NUMERIC "1"
// Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "32"
// Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "32"
// Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "1"
// Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
// Retrieval info: PRIVATE: enable NUMERIC "0"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
// Retrieval info: CONSTANT: INDATA_REG_B STRING "CLOCK0"

```
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "2048"
// Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "512"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "BIDIR_DUAL_PORT"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "CLEAR0"
// Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "CLEAR0"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "CLOCK0"
// Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK0"
// Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS STRING
"OLD_DATA"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING
"NEW_DATA_NO_NBE_READ"
// Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_B STRING
"NEW_DATA_NO_NBE_READ"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "11"
// Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "9"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_B NUMERIC "32"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_B NUMERIC "1"
// Retrieval info: CONSTANT: WRCONTROL_WRADDRESS_REG_B STRING "CLOCK0"
// Retrieval info: USED_PORT: aclr 0 0 0 0 INPUT GND "aclr"
// Retrieval info: USED_PORT: address_a 0 0 11 0 INPUT NODEFVAL "address_a[10..0]"
// Retrieval info: USED_PORT: address_b 0 0 9 0 INPUT NODEFVAL "address_b[8..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: data_a 0 0 8 0 INPUT NODEFVAL "data_a[7..0]"
// Retrieval info: USED_PORT: data_b 0 0 32 0 INPUT NODEFVAL "data_b[31..0]"
// Retrieval info: USED_PORT: q_a 0 0 8 0 OUTPUT NODEFVAL "q_a[7..0]"
// Retrieval info: USED_PORT: q_b 0 0 32 0 OUTPUT NODEFVAL "q_b[31..0]"
// Retrieval info: USED_PORT: wren_a 0 0 0 0 INPUT GND "wren_a"
// Retrieval info: USED_PORT: wren_b 0 0 0 0 INPUT GND "wren_b"
// Retrieval info: CONNECT: @aclr0 0 0 0 0 aclr 0 0 0 0
// Retrieval info: CONNECT: @address_a 0 0 11 0 address_a 0 0 11 0
// Retrieval info: CONNECT: @address_b 0 0 9 0 address_b 0 0 9 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @data_a 0 0 8 0 data_a 0 0 8 0
// Retrieval info: CONNECT: @data_b 0 0 32 0 data_b 0 0 32 0
// Retrieval info: CONNECT: @wren_a 0 0 0 0 wren_a 0 0 0 0
// Retrieval info: CONNECT: @wren_b 0 0 0 0 wren_b 0 0 0 0
// Retrieval info: CONNECT: q_a 0 0 8 0 @q_a 0 0 8 0
// Retrieval info: CONNECT: q_b 0 0 32 0 @q_b 0 0 32 0
// Retrieval info: GEN_FILE: TYPE_NORMAL RAM_cc.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL RAM_cc.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL RAM_cc.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL RAM_cc.bsf TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL RAM_cc_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL RAM_cc_bb.v TRUE
// Retrieval info: LIB_FILE: altera_mf
```

Filename: ccl.cpp

```
/* This code uses APIs from the Free Image project */

#include <iostream>
#include <stdio.h>
#include <FreeImage.h>

#include "ccl_dd.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

using namespace std;

/**
        FreeImage error handler
        @param fif Format / Plugin responsible for the error
        @param message Error message
*/
void FreeImageErrorHandler(FREE_IMAGE_FORMAT fif, const char *message) {
        printf("\n*** ");
        if(fif != FIF_UNKNOWN) {
           printf("%s Format\n", FreeImage_GetFormatFromFIF(fif));
        }
        printf("%s\n", message);
        printf(" ***\n");
}

// Device file descriptor
int ccl_fd;

/* Transmit a row of pixel data to the ccl device */
void write_img_row(BYTE *ip_bits, unsigned int width)
{
  img_row_arg_t ira;
  ira.pix_count = width;
  ira.pix_data = ip_bits;
  if(ioctl(ccl_fd, CCL_WRITE_DATA, &ira))  {
    perror("ioctl(CCL_WRITE_DATA) failed");
    return;
  }
}

/* Read a row of pixel data from the ccl device */
void read_img_row(BYTE *op_bits, unsigned int width)
{
  img_row_arg_t ira;
  ira.pix_count = width;
  ira.pix_data = op_bits;
  if(ioctl(ccl_fd, CCL_READ_DATA, &ira))  {
    perror("ioctl(CCL_READ_DATA) failed");
    return;
  }
```

```
}

/* Send commands and status notifications to the ccl device*/
void write_register(unsigned int address, unsigned int val)
{
  reg_arg_t ra;
  ra.address = address;
  ra.reg_data = val;
  if(ioctl(ccl_fd, CCL_WRITE_REG, &ra))  {
    perror("ioctl(CCL_WRITE_REG) failed");
    return;
  }
}

/* Poll status registers to control program flow */
int poll_register(unsigned int address)
{
  reg_arg_t ra;
  ra.address = address;
  if(ioctl(ccl_fd, CCL_POLL_REG, &ra))  {
    perror("ioctl(CCL_POLL_REG) failed");
    return -1;
  }
  return ra.reg_data;
}

int main()
{
  static const char filename[] = "/dev/ccl";

  // Connect to ccl device
  if ( (ccl_fd = open(filename, O_RDWR)) == -1)  {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }

  // Initialize the FreeImage library
  FreeImage_Initialise(TRUE);

  // Print FreeImage copyright message
  const char *copyright = FreeImage_GetCopyrightMessage();
  printf("%s\n", copyright);

  // Initialize our own FreeImage error handler
  FreeImage_SetOutputMessage(FreeImageErrorHandler);


  // Read image from file
  FIBITMAP *img = FreeImage_Load(FIF_PNG, "test4.png", BMP_DEFAULT);

  // Initialize variables for intermediate data and results
  FIBITMAP *img_8b = NULL;
  FIBITMAP *res1_img = NULL;
  FIBITMAP *res2_img = NULL;
  unsigned int width, height, bpp, wib, pitch;
```

```
if (img)  {

  // Collects and prints image information
  width  = FreeImage_GetWidth(img);
  height  = FreeImage_GetHeight(img);
  bpp  = FreeImage_GetBPP(img);
  wib  = FreeImage_GetLine(img);
  pitch  = FreeImage_GetPitch(img);
  printf("Loaded Image. Width: %d Height: %d BPP: %d\n", width, height, bpp);
  printf("Width-Bytes: %d Stride: %d\n", wib, pitch);

  // Converts image to 8bit greyscale images if the input is in a different format
  if (bpp != 8)  {
    img_8b = FreeImage_ConvertTo8Bits(img);
  }
  else  {
    img_8b = FreeImage_Clone(img);
  }
  if(!img_8b)  {
    printf("Invalid pointer for 8 bit img\n");
  }

  // Allocate memory for result images
  res1_img = FreeImage_Allocate(width, height, 8);
  res2_img = FreeImage_Allocate(width, height, 8);

  // Reset img_transferred to 0
  write_register(182, 0);

  // Set mode 1
  write_register(180, 1);

  // Send image to ccl device for first stage of processing
  int write_idx = height-1, read_idx = height-1;
  while(read_idx >= 0)  {
    // Check write ready
    if(poll_register(184) && write_idx >= 0)  {
      BYTE *ip_bits = FreeImage_GetScanLine(img_8b, write_idx);
      write_img_row(ip_bits, width);
      write_idx--;
    }
    // Check read ready
    else if(poll_register(185) && read_idx >= 0)  {
      BYTE *op_bits = FreeImage_GetScanLine(res1_img, read_idx);
      read_img_row(op_bits, width);
      read_idx--;
    }
    if (write_idx < 0)  {
      // Set img_transferred
      write_register(182, 1);
    }
  }

  // Reset mode 1
  write_register(180, 0);
```

```
// Print max number of computed lables
printf("Max Labels: %d\n", poll_register(201));

// Wait till LUT values are resolved
while(!poll_register(186)) {
  usleep(10000);
}

// Set img_transferred
write_register(182, 0);

// Set mode 2
write_register(181, 1);

// Send image for second stage of processing
write_idx = height-1;
read_idx = height-1;
while(read_idx >= 0)  {
  // Check write ready
  if(poll_register(184) && write_idx >= 0)  {
    BYTE *ip_bits = FreeImage_GetScanLine(res1_img, write_idx);
    write_img_row(ip_bits, width);
    write_idx--;
  }
  // Check read ready
  else if(poll_register(185) && read_idx >= 0)  {
    BYTE *op_bits = FreeImage_GetScanLine(res2_img, read_idx);
    read_img_row(op_bits, width);
    read_idx--;
  }
  if (write_idx < 0)  {
    // Set img_transferred
    write_register(182, 1);
  }
}

// Reset mode 2
write_register(181, 0);

// Save pass-1 result image
if (FreeImage_Save(FIF_PNG, res1_img, "result1.png", 0))  {
  printf("Saved Image\n");
}
else  {
  printf("Save Failed\n");
}

// Save pass-2 result image
if (FreeImage_Save(FIF_PNG, res2_img, "result2.png", 0))  {
  printf("Saved Image\n");
}
else  {
  printf("Save Failed\n");
}

// Set background pixels to 255 to improve visibility of labeled regions
```

```
    for (int i = height-1; i >= 0; i--)  {
      BYTE *bits = FreeImage_GetScanLine(res2_img, i);
      for (int j = 0; j < width; j++)  {
        if(bits[j] == 0)  {
          bits[j] = 255;
        }
      }
    }

    // Save final result image
    if (FreeImage_Save(FIF_PNG, res2_img, "result.png", 0))  {
      printf("Saved Image\n");
    }
    else  {
      printf("Save Failed\n");
    }

    // Free memory
    FreeImage_Unload(img);
    FreeImage_Unload(img_8b);
    FreeImage_Unload(res1_img);
    FreeImage_Unload(res2_img);
  }

  // release the FreeImage library
  FreeImage_DeInitialise();

  return 0;
}
```

Filename: ccl_dd.cc

```
/*
Adapted from boiler plate code provided by Prof.Stephen Edwards
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/delay.h>
#include "ccl_dd.h"

#define DRIVER_NAME "ccl"

/*
 * Information about the device
 */
struct ccl_dev {
	struct resource res; /* Resource: our registers */
	void __iomem *virtbase; /* Where registers can be accessed in memory */
	unsigned char *pix_data;
	unsigned int pix_count;
} dev;

/*
 * Transmit a row of pixel data to the ccl device
 */
static void write_row(unsigned char* pix_data, unsigned int pix_count)
{
	int i;
	unsigned int temp;

	/**Group and format pixel data in little endian format before sending it to the RAM**/
	for(i = 0; i < pix_count; i++)
	{
	   if(i%4 == 0) {
			temp = 0;
	   }
	   temp = (temp << 8) | *(pix_data +  4*((int)i/4) + (3-(i%4)));
	   if(i%4 == 3)  {
			iowrite32(temp, dev.virtbase + (i-3));
```

```
            }
        }
}

/*
 * Read a row of pixel data from the ccl device
 */
static void read_row(unsigned char* pix_data, unsigned int pix_count)
{
        int i;
        unsigned int temp = 0;

        /**Unpack pixel data in little endian format and update the byte array in the image**/
        for(i = 0; i < pix_count; i++)
        {
           if(i%4 == 0) {
                    temp = ioread32(dev.virtbase + i);
           }
           *(pix_data + 4*((int)i/4) + ((i%4))) = temp & 0xFF;
           temp = temp >> 8;
               if(i%4 == 3) {
                    temp = 0;
           }
        }
}

/*
 * Send commands and status notifications to the ccl device
 */
static void write_reg(unsigned int address, unsigned int val)
{
        iowrite32(val, dev.virtbase + 4*address);
}

/*
 * Poll statu registers from the ccl device to control program flow
 */
static void poll_reg(unsigned int address, unsigned int *val)
{
        *val = ioread32(dev.virtbase + 4*address);
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the pixel data.
 * Note extensive error checking of arguments
 */
static long ccl_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        img_row_arg_t img_row;
        reg_arg_t ra;
```

```
        switch (cmd) {
        case CCL_WRITE_DATA:
          if (copy_from_user(&img_row, (img_row_arg_t *) arg,
                             sizeof(img_row_arg_t)))
                return -EACCES;
          write_row(img_row.pix_data, img_row.pix_count);
          break;

        case CCL_READ_DATA:
          if (copy_from_user(&img_row, (img_row_arg_t *) arg,
                             sizeof(img_row_arg_t)))
                return -EACCES;

          read_row(img_row.pix_data, img_row.pix_count);

          if (copy_to_user((img_row_arg_t *) arg, &img_row,
                           sizeof(img_row_arg_t)))
                return -EACCES;
          break;

        case CCL_WRITE_REG:
          if (copy_from_user(&ra, (reg_arg_t *) arg,
                             sizeof(reg_arg_t)))
                return -EACCES;

          write_reg(ra.address, ra.reg_data);
          break;

        case CCL_POLL_REG:
          if (copy_from_user(&ra, (reg_arg_t *) arg,
                             sizeof(reg_arg_t)))
                return -EACCES;

          poll_reg(ra.address, &ra.reg_data);

          if (copy_to_user((reg_arg_t *) arg, &ra,
                           sizeof(reg_arg_t)))
                return -EACCES;
          break;

        default:
          return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations ccl_fops = {
        .owner              = THIS_MODULE,
```

```
        .unlocked_ioctl = ccl_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice ccl_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &ccl_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init ccl_probe(struct platform_device *pdev)
{
        int ret;

    printk(KERN_INFO "ccl_dd: In init");

        /* Register ourselves as a misc device: creates /dev/ccl */
        ret = misc_register(&ccl_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
           ret = -ENOENT;
           goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
           ret = -EBUSY;
           goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
           ret = -ENOMEM;
           goto out_release_mem_region;
        }

        ///* Display a welcome message */
        //write_digit(cx, cy);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
```

```
out_deregister:
        misc_deregister(&ccl_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int ccl_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&ccl_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id ccl_of_match[] = {
        { .compatible = "altr,ccl" },
        {},
};
MODULE_DEVICE_TABLE(of, ccl_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver ccl_driver = {
        .driver     = {
           .name  = DRIVER_NAME,
           .owner = THIS_MODULE,
           .of_match_table = of_match_ptr(ccl_of_match),
        },
        .remove  = __exit_p(ccl_remove),
};

/* Called when the module is loaded: set things up */
static int __init ccl_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&ccl_driver, ccl_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit ccl_exit(void)
{
        platform_driver_unregister(&ccl_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(ccl_init);
module_exit(ccl_exit);

MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("Avinash, Manushree, Jerry, Columbia University");
MODULE_DESCRIPTION("Connected Component Labeling");
```

Filename: ccl_dd.h

```
#ifndef _CCL_DD_H
#define _CCL_DD_H
/* Datatype to facilitate transfer of image pixel data to and fro between user space and kernel space */
typedef struct {
  unsigned char *pix_data;
  unsigned int pix_count;
} img_row_arg_t;


/* Datatype to facilitate transfer of commands and status notifications to and fro between user space and kernel space */
typedef struct {
  uint32_t reg_data;
  unsigned int address;
} reg_arg_t;

#define CCL_MAGIC 'q'

/* ioctls and their arguments */
#define CCL_WRITE_DATA _IOW(CCL_MAGIC, 1, img_row_arg_t *)
#define CCL_READ_DATA  _IOWR(CCL_MAGIC, 2, img_row_arg_t *)
#define CCL_WRITE_REG  _IOW(CCL_MAGIC, 3, reg_arg_t *)
#define CCL_POLL_REG  _IOWR(CCL_MAGIC, 4, reg_arg_t *)

#endif //_CCL_DD_H
```