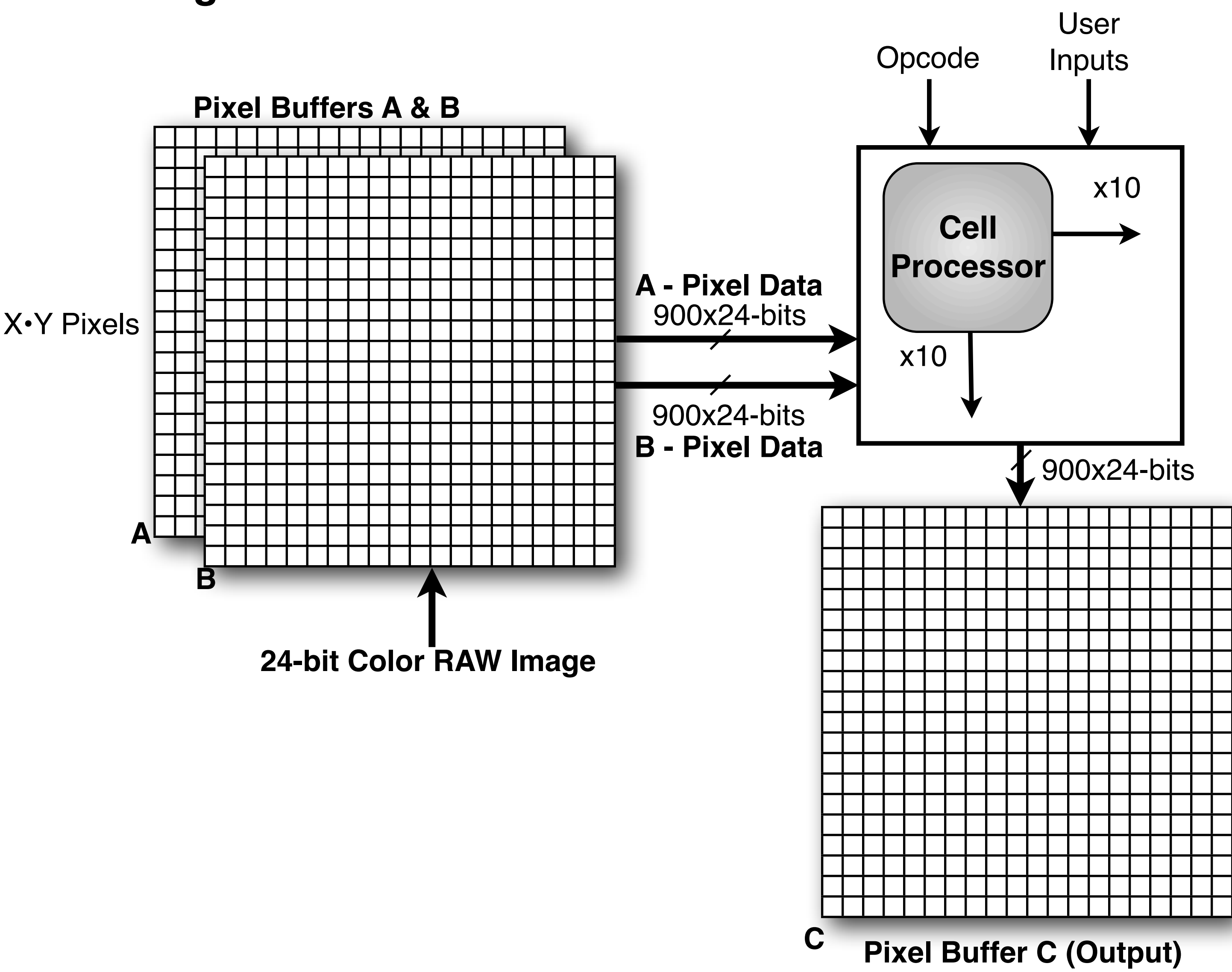
SIMD Image Processor

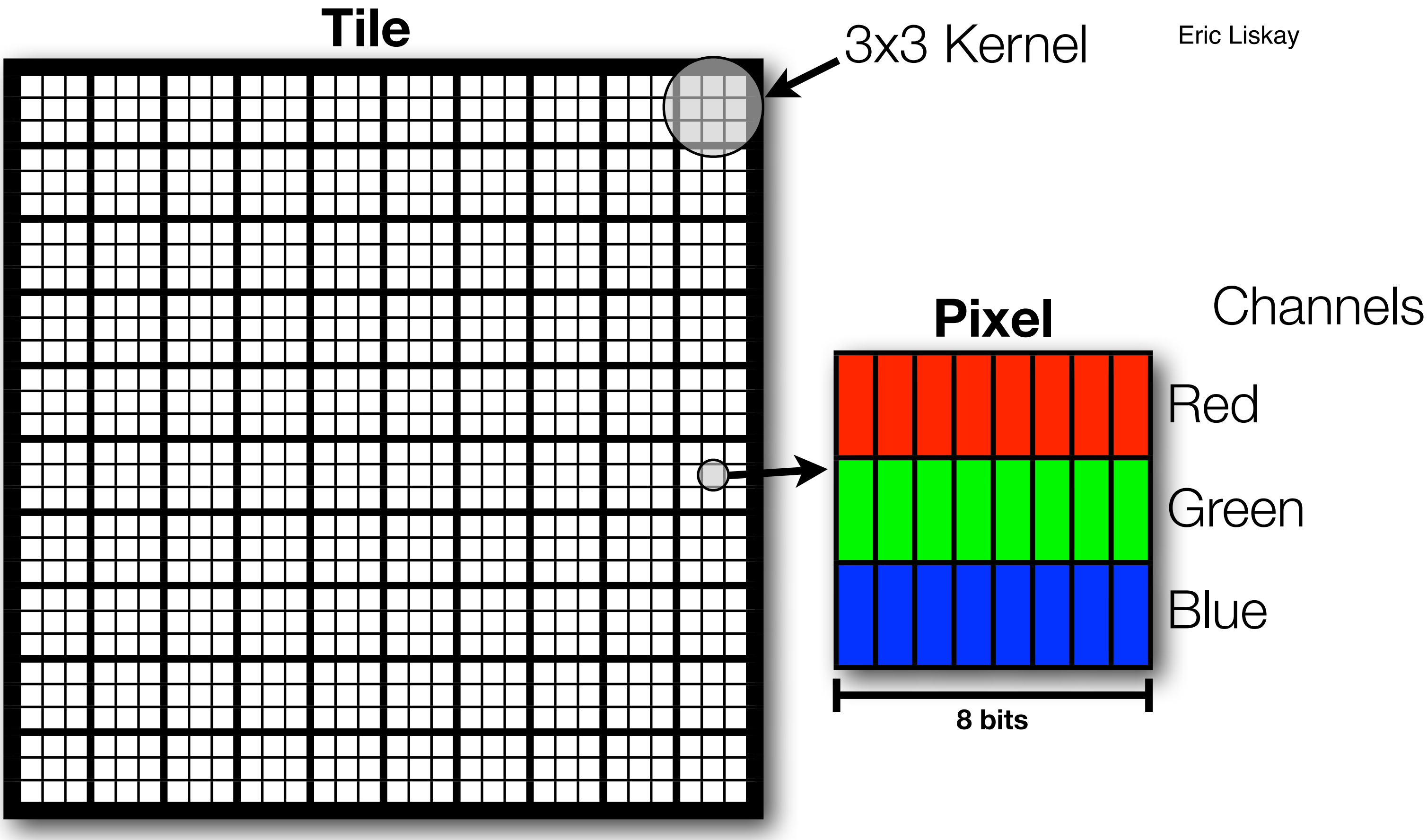
Eric Liskay
Andrew Northy
Neraj Kumar
ECE 590
Spring 2012

Objective

The objective of this project is to create an image processor using an SIMD(Single Instruction Multiple Data) architecture. Each SIMD processor will be able to perform operations on a small kernel of pixels. Every processor will perform the same operation its data. Once the main architecture of the design is complete, we plan to add operations as time allows.

Design





Cell Processor Controller

The top level module of the image processor can also be called the Cell Processor Controller. It is where the pixel buffers reside and where data is sent to the cell processors and received from them.

The imageProcessor entity has 8 ports. The first is a *clock*. The second is a *go* signal which triggers it to start processing whenever this signal is changed. The third is an input of an integer *number* which can be used by a testbench to label the output file when processing multiple images at a time. The fourth input is an *opcode* which is used to determine the operation to be done on the image and is passed onto the cell processor. The fifth and sixth inputs are *userInputA* and *userInputB*. These allow for the user to pass arguments/parameters along with the opcodes to affect the image. Their usage depends on the opcode. The last input is *userKernel*. This allows the user to pass a custom 3x3 kernel to the image processor to be used in convolution with the appropriate opcode. Finally, the image processor has an output signal called *complete* which indicates to the testbench when the image has been completed.

The pixel buffers are where images are stored when loaded into the image processor. We define some of the data structures in a package at the top of the ImagProcessor.vhd file. As shown in the figure above, each channel has 8 bits. Each pixel consists of three channels: a red channel, a green channel, and a blue channel. The image processor supports images in 24-bit RGB color. The exact file type that is able to be read in are Photoshop RAW files. Pixel buffers consist of a two-dimensional array of pixels. The size of the pixel buffer depends on the image size, which is set by the user in the IMAGE_WIDTH and IMAGE_HEIGHT parameters. The parameters

IMAGE_A_NAME, IMAGE_B_NAME, and IMAGE_C_NAME identify the input images and the desired name for the output image. These parameters can be passed in from outside sources such as the testbench.

Inside the architecture of the image processor, the component Cell is defined, passing the clock, opcode, user inputs, user kernel and three pixel arrays. There are two input arrays, one for each image, and one output. These pixel arrays are the size of the kernel which is currently defined as being a 3x3 pixel array. The Cell processors are created and connected by a two-dimensional generate block. Each processor receives the same opcode and user inputs. The number of cell processors generated is determined by the TILE_WIDTH. A tile is shown on the figure on the previous page. Tiling was implemented because the number of processors required if the whole image was to be worked on at once would become prohibitively larger as image size increased. This method allows one “tile” at a time to be processed, moving sequentially through the image. The width of a tile is how many kernels wide that it contains. This constant can be modified. Currently, we have it set to 10. This creates a total of 100 cell processors. Each processor works on one kernel of 9 pixels. This means that 900 pixels are processed in parallel per clock.

The image processor controller contains four processes. The first process is triggered by the go signal. This process reads in the images. Currently for simulation, it reads in these files from the drive using file_open. If either of the file names cannot be found it will fail on an assertion and display an error. Since you may only want to operate on a single image, it is okay to set both the images to the same name. Next, the two images are read into the two respective pixel buffers, one channel(one byte) at a time. Since the image processor will usually operate on color images, there are three channels. This parameter could however be set to 1 for 8-bit grayscale images. After the entire image is read in, the file handles are closed. A ready signal is set to high.

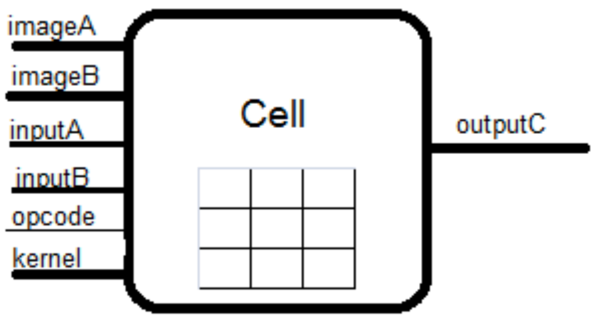
Upon the raising of the ready signal, image processing begins and the next two processes are activated. The first of these processes triggers on the rising edge of the clock. In this process kernels are sent to all the cell processors for processing. The second of these processes places the output of the cells in the correct position in the Pixel Buffer C. It also contains code to control the iteration of the image processor through the tiles in the image. As it is currently coded, there is a limitation on the image processor where input dimensions must be set to a multiple of the tile width in pixels(currently 30). An input image of any size can be used, however the IMAGE_WIDTH and IMAGE_HEIGHT parameters must be a multiple of 30. Smaller values than the input image will just cause it to be cropped. Once the last tile is done processing, a done signal is set to high.

Upon the rising of the done signal, the final process is triggered. This processes creates the output file and writes the data of Pixel Buffer C into the file. The file is named based on the IMAGE_C_NAME concatenated with a number and the file extension “.raw”. Once this is complete, the *complete* output is toggled to let the testbench know processing has been completed.

Cell Processor

Our design includes the image processor sending pixel arrays to 100 cell processors to perform the work in parallel.

Defined types: Pixel
Pixel is 8 bits and has 3 channels (red, green, blue).
Pixel_array is a 3x3 array of pixels, and this size is sent to each Cell processor.



Inputs:

imageA takes a 3x3 pixel array from the ImageProcessor. This is the main operand used in every opcode.
imageB takes a 3x3 pixel array from the ImageProcessor. This second image is used in operations that are performed on two images (ADD, SUB, MULT, AND, OR, NOR, and XOR)
userInputA is an 8 bit immediate value used in operations that utilize an immediate value
userInputB used in the Darken Highlights and Brighten Shadows operations. userInputA is used as the threshold value to compare against, and userInputB is the amount to increase/decrease the pixel values by.
opcode defines what operation the cell processor is to perform
kernel allows a user-specified kernel to be used in conjunction with convolution mode.

Operations

The cell processor has the ability to perform a wide variety of functions. Many of these are well known, and their hardware implementation is trivial. These types of functions include the following:

- ADD – Addition
- SUB – Subtraction
- INV – Inversion (bit flip)
- AND – Logical AND
- OR – Logical OR
- NOR – Logical NOR
- XOR – Logical XOR

The cell processor can perform these operations between two images, an image and an immediate value, or perform the operation on just a single color channel rather than all three.

There are other operations that warrant further explanation on the intended hardware implementation.

MULT – Multiply

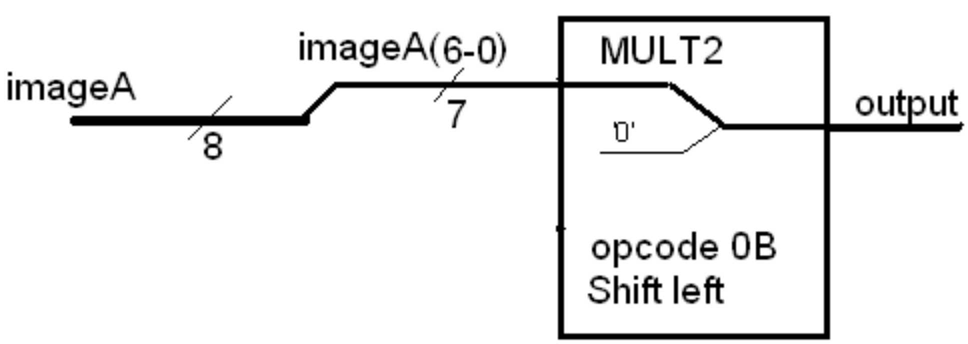
Our multiplication function has been simplified for space constriction, and results in a look-up table. Rather than multiplying two 8 bit numbers, it takes the most-significant 4 bits of the two operands, resulting in an 8 bit output rather than 16 bit output.

Due to the operation being on 4 bit inputs, a look-up table to calculate the operation makes the most sense in terms of space and speed.

imageB(7-4) \ imageA(7-4)	0000	0001	0010	…	1111
0000	00000000	00000000	00000000		00000000
0001	00000000	00000001	00000010		00001111
0010	00000000	00000010	00000100		00011110
…					
1111	00000000	00001111	00011110		11100001

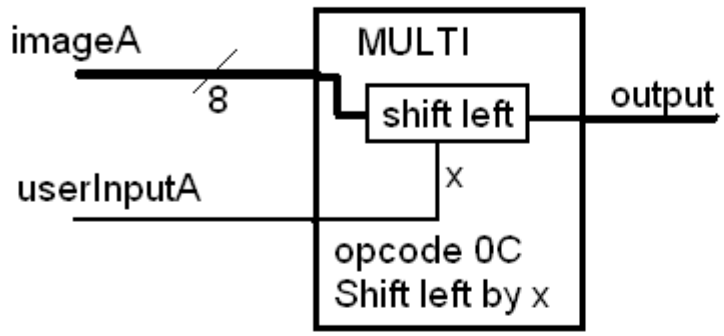
MULT2 – Multiply by 2

This operation takes imageA and shifts the pixel bits left by one, affectively multiplying it by two.



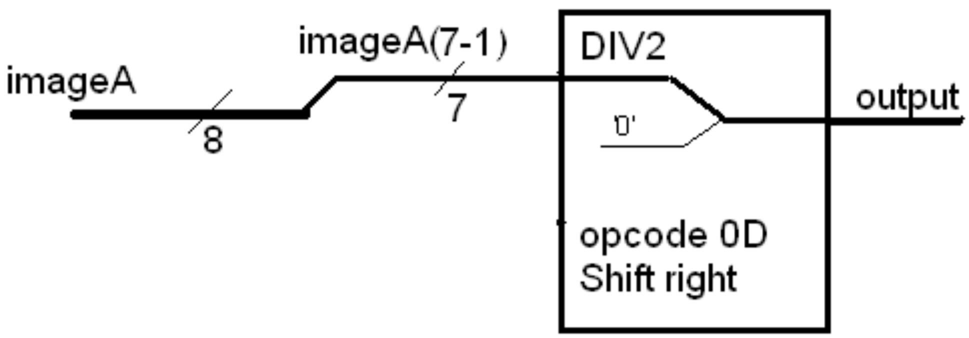
MULTI – Multiply immediate

Multiply immediate takes imageA, and shifts the pixel values left by userInputA



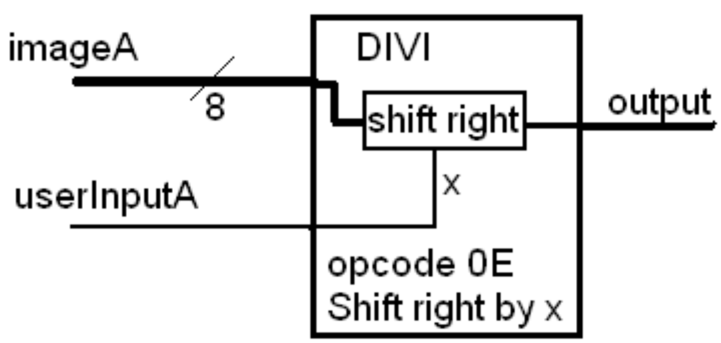
DIV2 – Divide by two

Much like MULT2, the DIV2 operation takes imageA, and shifts it right by one, effectively dividing it by two.



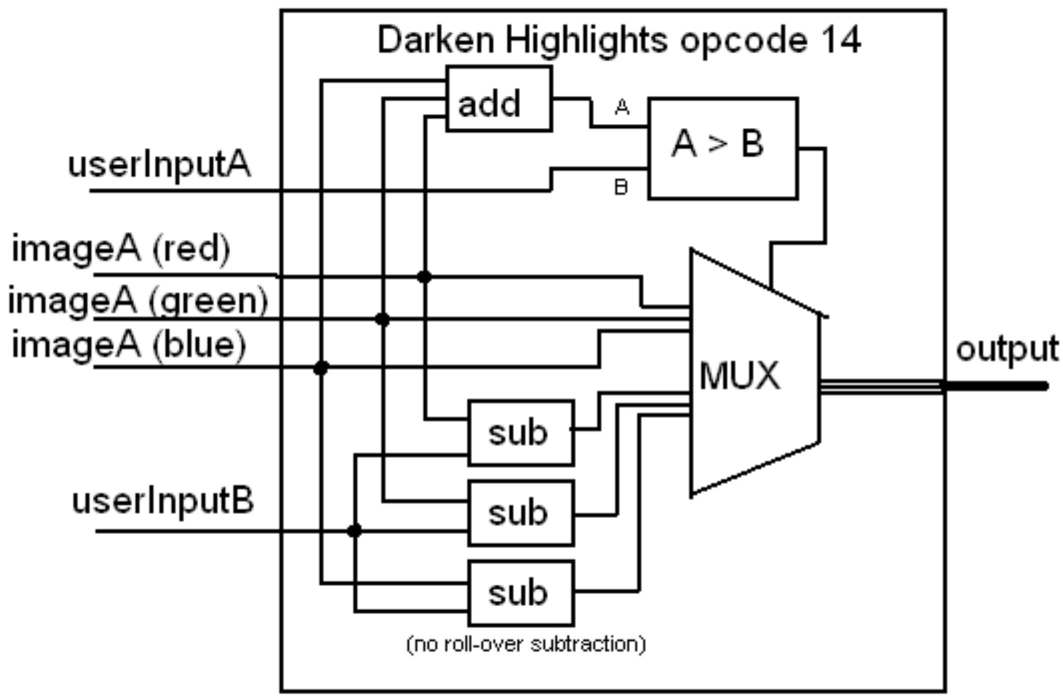
DIVI – Divide Immediate

Divide immediate takes imageA and shifts the pixel values right by userInputA.



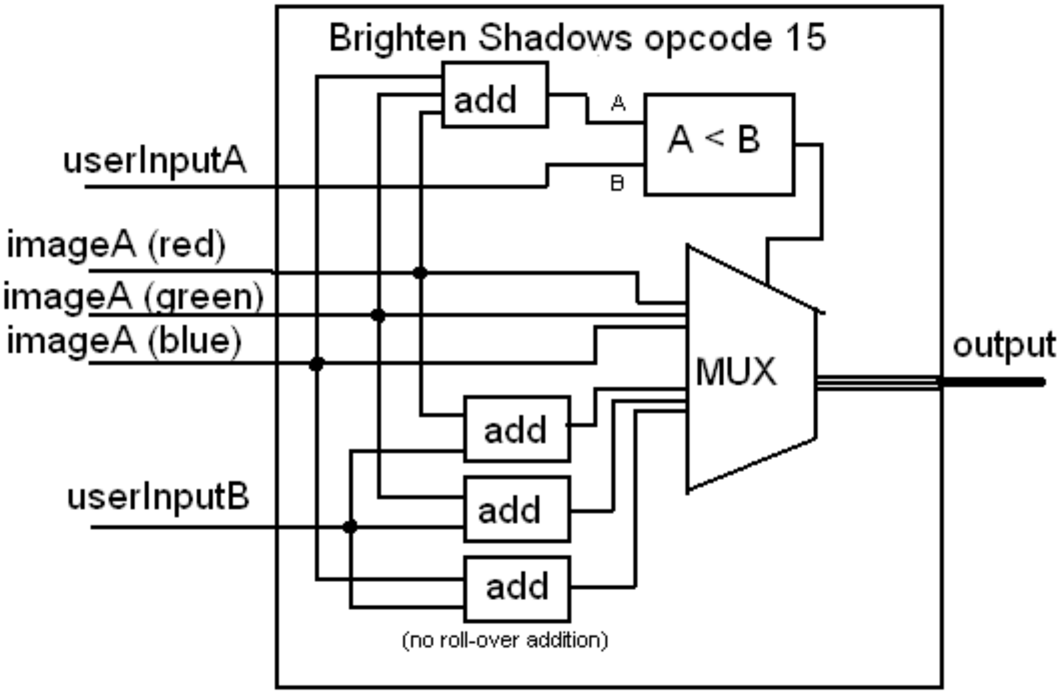
Darken Highlights

For each pixel, the three color channel values are added together. This is compared with userInputA (threshold).
If the sum is *not greater* than userInputA, the pixel is returned unchanged
If the sum is *greater* than userInputA for this pixel, we will *subtract* the value of each color channel by userInputB.



Brighten Shadows

For each pixel, the three color channel values are added together. This is compared with userInputA (threshold).
If the sum is *not less* than userInputA, the pixel is returned unchanged
If the sum is *less* than userInputA for this pixel, we will *add* the value of each color channel by userInputB.



Pixel flip

This operation connects the output of the incoming 3x3 pixel array in a flipped order.

original

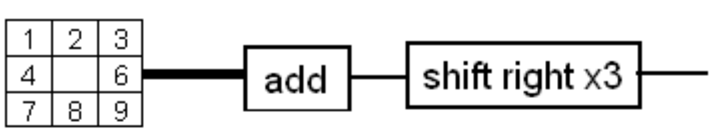
1	2	3
4	5	6
7	8	9

flipped

9	8	7
6	5	4
3	2	1

Average

This function will sum the outer 8 values of the 3x3 pixel array, and then shift the results right three times (effectively dividing by 8). This calculated value is what the 3x3 pixel output is set to.



Sobel Algorithm

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * A$$

$$G = \sqrt{G_x^2 + G_y^2}$$

The above figure shows how Sobel edge detection is done mathematically. The Sobel operator uses two 3x3 kernels which are convolved with a 3x3 array of pixels from the image, denoted as *A*. G_x and G_y are the derivatives of the convolution. G_x *is* for horizontal changes and G_y is for vertical changes. The * denotes the 2-dimensional convolution operation.

The output *G*, the gradient magnitude, is obtained by squaring G_x and G_y, adding them, and then taking the square root of the resulting sum. The approximate magnitude can also be computed by just adding the absolute values of G_x and G_y. This may be less accurate, but is faster to compute.

Convolution Mode Operations

A “regular mode” operation reads in a 3x3 pixel kernel and writes out a 3x3 pixel kernel with all pixels being operated on individually. A “convolution mode” operation reads in a 3x3 pixel kernel and uses data from those 9 pixels to write out one pixel, the center one. In the first Sobel operation, I used one line of combinational logic to compute derivatives G_x and G_y. For all the remaining convolution operations, I use for-loops to multiply and accumulate the derivatives. They are multiplied by a constant 3x3 kernel based on the operation such as Sobel or Prewitt. For opcode 0x27, a user-defined kernel is used.

Opcodes

Mneumonic	Opcode	Description
add	0x00	add pixel channel values in array A to array B
addi	0x01	add userInputA to values in all channels from array A
addir	0x02	add userInputA to value in red channel from array A
addig	0x03	add userInputA to value in green channel from array A
addib	0x04	add userInputA to value in blue channel from array A
sub	0x05	subtract pixel channel values in array B from array A
subi	0x06	subtract userInputA from values in all channels from array A
subir	0x07	subtract userInputA from value in red channel from array A
subig	0x08	subtract userInputA from value in green channel from array A
subib	0x09	subtract userInputA from value in blue channel from array A
mult	0x0A	multiply pixel channel values in array A by array B
mult2	0x0B	multiply pixel channel values in array A by 2
multi	0x0C	multiply pixel channel values in array A by shifting left by userInputA
div2	0x0D	divide pixel channel values in array A by 2
divi	0x0E	divide pixel channel values in array A by shifting right by userInputA
inv	0x0F	Invert pixel channel values in array A
and	0x10	AND pixel channel values in Array A with pixels in array B
or	0x11	OR pixel channel values in Array A with pixels in array B
nor	0x12	NOR pixel channel values in Array A with pixels in array B
xor	0x13	XOR pixel channel values in Array A with pixels in array B
drkn	0x14	Darken Highlights: If sum of pixel channel values in array A are above userInputA, subtract userInputB
brtn	0x15	Brighten Shadows: If sum of pixel channel values in array A are below userInputA, add userInputB
grayr	0x16	Grayscale based on Red Channel
grayg	0x17	Grayscale based on Green Channel
grayb	0x18	Grayscale based on Blue Channel
pixflip	0x19	Pixels mirrored over middle pixel
sobel	0x1A	Sobel
sobel2	0x1B	Sobel Method 2
prewitt	0x1C	Prewitt
robn	0x1D	Robinson North
robnw	0x1E	Robinson Northwest
robne	0x1F	Robinson Northeast
robe	0x20	Robinson East
robw	0x21	Robinson West
robs	0x22	Robinson South
robse	0x23	Robinson Southeast
robsw	0x24	Robinson Southwest
avggry	0x25	3x3 pixel average in grayscale
avgclr	0x26	3x3 pixel average in color
ukernel	0x27	User programmable filter
min	0x28	3x3 pixel minimum
max	0x29	3x3 pixel maximum

Color Codes for Manipulating pictures:

Basic Color Codes Table:

Basic colors:

Color	HTML/CSS Name	Hex Code #RRGGBB	Decimal Code (R,G,B)
	Black	#000000	(0,0,0)
	White	#FFFFFF	(255,255,255)
	Red	#FF0000	(255,0,0)
	Lime	#00FF00	(0,255,0)
	Blue	#0000FF	(0,0,255)
	Yellow	#FFFF00	(255,255,0)
	Cyan / Aqua	#00FFFF	(0,255,255)
	Magenta / Fuchsia	#FF00FF	(255,0,255)
	Silver	#C0C0C0	(192,192,192)
	Gray	#808080	(128,128,128)
	Maroon	#800000	(128,0,0)
	Olive	#808000	(128,128,0)
	Green	#008000	(0,128,0)
	Purple	#800080	(128,0,128)
	Teal	#008080	(0,128,128)
	Navy	#000080	(0,0,128)

Using the above color table, we can give any kind of user defined color codes to manipulate the pictures. As shown in the picture above, an array of RGB values is declared as 24 bits each channel consists of 8 bits. Different combinations of bits produce different colors according to which, we can select different combinations and mix and match the colors in the picture. Some of the basic color codes are represented as shown in the table above. But there are many different combinations which are possible

ImageProcessor.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

package pkg is
   subtype channel is std_logic_vector(7 downto 0);  -- 8-bits per channel
   type pixel is array (2 downto 0) of channel;
   type pixel_array is array (integer range <>,integer range <>) of pixel;
   type kernel is array(2 downto 0,2 downto 0) of integer;
end pkg;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use std.textio.all;
use ieee.STD_LOGIC_TEXTIO.all;
use ieee.std_logic_arith.CONV_STD_LOGIC_VECTOR;
use ieee.numeric_std.unsigned;
use ieee.numeric_std.to_integer;
use work.pkg.all;


entity imageProcessor is
   generic (
      IMAGE_A_NAME   : string  := "toucan.raw"; -- Filename of Image A
      IMAGE_B_NAME   : string  := "toucan.raw"; -- Filename of Image B
      IMAGE_C_NAME   : string  := "output"; -- Filename of Image C
      IMAGE_WIDTH    : integer := 300; -- Width of image(s) in pixels
      IMAGE_HEIGHT   : integer := 180; -- Height of image(s) in pixels
      IMAGE_CHANNELS : integer := 3 -- Channels: 3=RGBColor, 1=Grayscale
   );
   port (
      clock       : in  std_logic := '0';
      go          : in  std_logic := '0';
      number      : in  integer := 0;
      opcode      : in  std_logic_vector(7 downto 0) := x"FF";
      userInputA  : in  std_logic_vector(7 downto 0) := b"00000000";
      userInputB  : in  std_logic_vector(7 downto 0) := b"00000000";
      userKernel  : in  kernel;
      complete    : out std_logic := '0'
   );
end imageProcessor;

architecture main of imageProcessor is

   component Cell is
      generic (
         IMAGE_CHANNELS : integer := IMAGE_CHANNELS -- Channels: 3=RGBColor, 1=Grayscale
```

```
    );
    port (
        clock      : in  std_logic;
        opcode     : in  std_logic_vector(7 downto 0);
        userInputA : in  std_logic_vector(7 downto 0);
        userInputB : in  std_logic_vector(7 downto 0);
        userKernel : in  kernel;
        inputA     : in  pixel_array(2 downto 0,2 downto 0);
        inputB     : in  pixel_array(2 downto 0,2 downto 0);
        outputC    : out pixel_array(2 downto 0,2 downto 0)
    );
end component;

-- Width of tiles in pixels
constant TILE_WIDTH : integer := 10;

type character_file is file of character;
file imageA, imageB, imageC : character_file;

-- Type definitions for Pixel Buffers
type pixelColumn is array ((IMAGE_HEIGHT-1) downto 0) of pixel;
type pixelBuffer is array ((IMAGE_WIDTH-1) downto 0) of pixelColumn;
signal pixelBufferA, pixelBufferB, PixelBufferC : pixelBuffer;

--Connections to Cell Processors
type cell_bus is array ((TILE_WIDTH-1) downto 0,(TILE_WIDTH-1) downto 0)
    of pixel_array(2 downto 0,2 downto 0);
signal imageA2Cell, imageB2Cell, cell2ImageC : cell_bus;

signal ready, done, isComplete : std_logic := '0';
signal X, Y : integer := 0;

begin

    -- Generate Cell Processors
    CellGenY:
        for l in 0 to (TILE_WIDTH-1) generate
            CellGenX:
            for w in 0 to (TILE_WIDTH-1) generate
                CellXY : Cell port map (
                    clock => clock,
                    opcode => opcode,
                    userInputA => userInputA,
                    userInputB => userInputB,
                    userKernel => userKernel,
                    inputA => imageA2Cell(w,l),
                    inputB => imageB2Cell(w,l),
                    outputC => cell2ImageC(w,l));
            end generate CellGenX;
```

```
   end generate CellGenY;



readImages: process(go)

   variable char : character;
   variable fstatus : FILE_OPEN_STATUS;

begin

   report "Working on image #" & integer'image(number);

   ready <= '0';

   -- Open files and check to make sure they opened correctly
   file_open(fstatus, imageA, IMAGE_A_NAME, read_mode);
   assert (fstatus = open_ok) report "imageA not found" severity FAILURE;
   file_open(fstatus, imageB, IMAGE_B_NAME, read_mode);
   assert (fstatus = open_ok) report "imageB not found" severity FAILURE;

   -- Read pixel data from files into Pixel Buffers
   for h in 0 to (IMAGE_HEIGHT-1) loop
      for w in 0 to (IMAGE_WIDTH-1) loop
         for c in 0 to (IMAGE_CHANNELS-1)  loop
            read(imageA, char);
            pixelBufferA(w)(h)(c) <= CONV_STD_LOGIC_VECTOR(character'pos(char), 8);
            read(imageB, char);
            pixelBufferB(w)(h)(c) <= CONV_STD_LOGIC_VECTOR(character'pos(char), 8);
         end loop;
      end loop;
   end loop;

   -- close files
   file_close(imageA);
   file_close(imageB);

   ready <= '1';



   --wait;
end process readImages;


SendToCells: process(clock)
   variable my_line : line;
begin
   if (rising_edge(clock) and (ready='1')) then
```

```
        --Print for Debug
--        for h in 0 to (IMAGE_HEIGHT-1) loop
--           for w in 0 to (IMAGE_WIDTH-1) loop
--              if ((pixelBufferA(w)(h)(0) = b"00000000") and
--                  (pixelBufferA(w)(h)(1) = b"00000000") and
--                  (pixelBufferA(w)(h)(2) = b"00000000"))then
--                 write(my_line, string'("*"));
--              elsif ((pixelBufferA(w)(h)(0) = b"11111111") and
--                     (pixelBufferA(w)(h)(1) = b"11111111") and
--                   (pixelBufferA(w)(h)(2) = b"11111111"))then
--                 write(my_line, string'("."));
--              elsif (pixelBufferA(w)(h)(0) = b"11111111")then
--                 write(my_line, string'("R"));
--              elsif (pixelBufferA(w)(h)(1) = b"11111111") then
--                 write(my_line, string'("G"));
--              elsif (pixelBufferA(w)(h)(2) = b"11111111") then
--                 write(my_line, string'("B"));
--              else
--                 write(my_line, pixelBufferA(w)(h)(2)); --string'("?"));
--              end if;
--           end loop;
--           writeline(output,my_line);
--        end loop;
        if(opcode >= x"1A") then
           --Output tile to all Cells
           for j in 0 to (TILE_WIDTH-1) loop
              for i in 0 to (TILE_WIDTH-1) loop
                 for b in 0 to 2 loop
                    for a in 0 to 2 loop
                       if(((x+i+a) < IMAGE_WIDTH) and ((y+j+b) < IMAGE_HEIGHT)) then
                          imageA2Cell(i,j)(a,b) <= PixelBufferA(x+i+a)(y+j+b);
                          imageB2Cell(i,j)(a,b) <= PixelBufferB(x+i+a)(y+j+b);
                       end if;
                    end loop;
                 end loop;
              end loop;
           end loop;
        else
           --Output tile to all Cells
           for j in 0 to (TILE_WIDTH-1) loop
              for i in 0 to (TILE_WIDTH-1) loop
                 for b in 0 to 2 loop
                    for a in 0 to 2 loop
                       imageA2Cell(i,j)(a,b) <= PixelBufferA(x+i*3+a)(y+j*3+b);
                        imageB2Cell(i,j)(a,b) <= PixelBufferB(x+i*3+a)(y+j*3+b);
                    end loop;
                 end loop;
              end loop;
```

```
        end loop;
      end if;
    end if;
end process SendToCells;




ReadFromCells: process(clock)
begin
    if (falling_edge(clock) and (ready = '1')) then

      if(opcode >= x"1A") then
          for j in 0 to (TILE_WIDTH-1) loop
            for i in 0 to (TILE_WIDTH-1) loop
              if(((x+i+1) < IMAGE_WIDTH) and ((y+j+1) < IMAGE_HEIGHT)) then
                PixelBufferC(x+i+1)(y+j+1) <= cell2ImageC(i,j)(1,1);
              end if;
            end loop;
        end loop;

        --Iterate through tiles on the x and y dimention
        if (x < (IMAGE_WIDTH-TILE_WIDTH)) then
            done <= '0';
            x <= x + TILE_WIDTH;
        elsif ((x >= (IMAGE_WIDTH-TILE_WIDTH)) and
               (y < (IMAGE_HEIGHT-TILE_WIDTH))) then
            x <= 0;
            y <= y + TILE_WIDTH;
        elsif (y >= (IMAGE_HEIGHT-TILE_WIDTH)) then
            done <= '1';
            x <= 0;
            y <= 0;
        end if;
      else
      -- Read output from all cells into Pixel Buffer C
        for j in 0 to (TILE_WIDTH-1) loop
          for i in 0 to (TILE_WIDTH-1) loop
            for b in 0 to 2 loop
              for a in 0 to 2 loop
                PixelBufferC(x+i*3+a)(y+j*3+b) <= cell2ImageC(i,j)(a,b);
              end loop;
            end loop;
          end loop;
        end loop;
        --Iterate through tiles on the x and y dimention
        if (x < (IMAGE_WIDTH-3*TILE_WIDTH)) then
            done <= '0';
            x <= x + 3*TILE_WIDTH;
```

```
          elsif ((x >= (IMAGE_WIDTH-3*TILE_WIDTH)) and
                 (y < (IMAGE_HEIGHT-3*TILE_WIDTH))) then
              x <= 0;
              y <= y + 3*TILE_WIDTH;
          elsif (y >= (IMAGE_HEIGHT-3*TILE_WIDTH)) then
              done <= '1';
              x <= 0;
              y <= 0;
          end if;
        end if;
      end if;
   end process ReadFromCells;



   writeOutputImage: process(done)

      variable char : character;
      variable fstatus : FILE_OPEN_STATUS;
      variable buf : line;

   begin

      if (done = '1') then

         -- Open output file
         file_open(fstatus, imageC, (IMAGE_C_NAME & integer'image(number) & ".raw"), write_mode);
         assert (fstatus = open_ok) report "imageC could not be created" severity FAILURE;

         -- Read pixel data from files into Pixel Buffers
         for h in 0 to (IMAGE_HEIGHT-1) loop
            for w in 0 to (IMAGE_WIDTH-1) loop
               for c in 0 to (IMAGE_CHANNELS-1) loop
                  write(imageC, character'val(to_integer(unsigned(pixelBufferC(w)(h)(c)))));
               end loop;
            end loop;
         end loop;


         -- close files
         file_close(imageC);

         complete <=  not isComplete;
         isComplete <= not isComplete;
         --assert (1=2) report "Finished processing!" severity FAILURE;

      end if;
   end process writeOutputImage;
end main;
```

Cell.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.pkg.all;


entity Cell is
   generic (
      IMAGE_CHANNELS : integer := 3 -- Channels: 3=RGBColor, 1=Grayscale
   );
   port (
      clock      : in  std_logic;
      opcode     : in  std_logic_vector(7 downto 0);
      userInputA : in  std_logic_vector(7 downto 0);
      userInputB : in  std_logic_vector(7 downto 0);
      userKernel : in  kernel;
      inputA     : in  pixel_array(2 downto 0,2 downto 0);
      inputB     : in  pixel_array(2 downto 0,2 downto 0);
      outputC    : out pixel_array(2 downto 0,2 downto 0)
   );
end Cell;

architecture CellArch of Cell is

   constant Sobelx    : kernel := ((-1,0,1),(-1,0,2),(-1,0,1));
   constant Sobely    : kernel := ((-1,-2,-1),(0,0,0),(1,2,1));
   constant Prewittx : kernel := ((-1,-1,-1),(0,0,0),(1,1,1));
   constant Prewitty : kernel := ((-1,0,1),(-1,0,1),(-1,0,1));

   constant RobinsonN  : kernel := ((1,2,1),(0,0,0),(-1,-2,-1));
   constant RobinsonNE : kernel := ((0,1,2),(-1,0,1),(-2,-1,0));
   constant RobinsonE  : kernel := ((-1,0,1),(-2,0,2),(-1,0,1));
   constant RobinsonSE : kernel := ((-2,-1,0),(-1,0,1),(0,1,2));
   constant RobinsonS  : kernel := ((-1,-2,-1),(0,0,0),(1,2,1));
   constant RobinsonSW : kernel := ((0,-1,-2),(-1,0,-1),(2,1,0));
   constant RobinsonW  : kernel := ((1,0,-1),(2,0,-2),(1,1,-1));
   constant RobinsonNW : kernel := ((2,1,0),(1,0,-1),(0,-1,-2));

   constant Average : kernel := ((1,1,1),(1,1,1),(1,1,1));


   function  sqrt  ( d : UNSIGNED ) return UNSIGNED is
      variable a : unsigned(31 downto 0):=d;  --original input.
      variable q : unsigned(15 downto 0):=(others => '0');  --result.
      variable left,right,r : unsigned(17 downto 0):=(others => '0');
      variable i : integer:=0;
   begin
      for i in 0 to 15 loop
         right(0):='1';
         right(1):=r(17);
         right(17 downto 2):=q;
         left(1 downto 0):=a(31 downto 30);
```

```
            left(17 downto 2):=r(15 downto 0);
            a(31 downto 2):=a(29 downto 0);  --shifting by 2 bit.
            if ( r(17) = '1') then
               r := left + right;
            else
               r := left - right;
            end if;
            q(15 downto 1) := q(14 downto 0);
            q(0) := not r(17);
         end loop;
         return q;
      end sqrt;

   begin

      process(opcode,userInputA,inputA,inputB) is
         variable Gx, Gy : signed(15 downto 0) := (others=> '0');
         variable Gu : unsigned(15 downto 0);
         variable Gsqd : unsigned(31 downto 0);
         variable tempvar : unsigned(10 downto 0);
         variable tempsignal_red : std_logic_vector(10 downto 0);
         variable tempsignal_green : std_logic_vector(10 downto 0);
         variable tempsignal_blue : std_logic_vector(10 downto 0);
      begin
         case opcode is
            when x"00" =>   -- ADD
               for y in 0 to 2 loop
                  for x in 0 to 2 loop
                     for c in 0 to (IMAGE_CHANNELS-1) loop
                        outputC(x,y)(c)   <= inputA(x,y)(c) + inputB(x,y)(c);
                     end loop;
                  end loop;
               end loop;

            when x"01" =>   -- ADDI
               for y in 0 to 2 loop
                  for x in 0 to 2 loop
                     for c in 0 to (IMAGE_CHANNELS-1) loop
                        outputC(x,y)(c)   <= inputA(x,y)(c) + userInputA;
                     end loop;
                  end loop;
               end loop;

            when x"02" =>   -- ADDIR
               for y in 0 to 2 loop
                  for x in 0 to 2 loop
                     outputC(x,y)(0)   <= inputA(x,y)(0) + userInputA;
                     outputC(x,y)(1)   <= inputA(x,y)(1);
                     outputC(x,y)(2)   <= inputA(x,y)(2);
                  end loop;
               end loop;

            when x"03" =>   -- ADDIG
               for y in 0 to 2 loop
                  for x in 0 to 2 loop
                     outputC(x,y)(0)   <= inputA(x,y)(0);
```

```
            outputC(x,y)(1)   <= inputA(x,y)(1) + userInputA;
            outputC(x,y)(2)   <= inputA(x,y)(2);
         end loop;
      end loop;

   when x"04" =>    -- ADDIB
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            outputC(x,y)(0)   <= inputA(x,y)(0);
            outputC(x,y)(1)   <= inputA(x,y)(1);
            outputC(x,y)(2)   <= inputA(x,y)(2) + userInputA;
         end loop;
      end loop;

   when x"05" =>    -- SUB
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            for c in 0 to (IMAGE_CHANNELS-1) loop
               outputC(x,y)(c)   <= inputA(x,y)(c) - inputB(x,y)(c);
            end loop;
         end loop;
      end loop;

   when x"06" =>    -- SUBI
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            for c in 0 to (IMAGE_CHANNELS-1) loop
               outputC(x,y)(c)   <= inputA(x,y)(c) - userInputA;
            end loop;
         end loop;
      end loop;

   when x"07" =>    -- SUBIR
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            outputC(x,y)(0)   <= inputA(x,y)(0) - userInputA;
            outputC(x,y)(1)   <= inputA(x,y)(1);
            outputC(x,y)(2)   <= inputA(x,y)(2);
         end loop;
      end loop;

   when x"08" =>    -- SUBIG
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            outputC(x,y)(0)   <= inputA(x,y)(0);
            outputC(x,y)(1)   <= inputA(x,y)(1) - userInputA;
            outputC(x,y)(2)   <= inputA(x,y)(2);
         end loop;
      end loop;

   when x"09" =>    -- SUBIB
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            outputC(x,y)(0)   <= inputA(x,y)(0);
            outputC(x,y)(1)   <= inputA(x,y)(1);
            outputC(x,y)(2)   <= inputA(x,y)(2) - userInputA;
```

```
      end loop;
   end loop;

when x"0A" =>  -- MULT
   for y in 0 to 2 loop
      for x in 0 to 2 loop
         for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(x,y)(c)   <= (inputA(x,y)(c)(7 downto 4) *
               inputB(x,y)(c)(7 downto 4));
         end loop;
      end loop;
   end loop;

when x"0B" =>   -- MULT 2
   for y in 0 to 2 loop
      for x in 0 to 2 loop
         for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(x,y)(c) <= inputA(x,y)(c)(6 downto 0) & '0';
         end loop;
      end loop;
   end loop;

when x"0C" =>   -- MULTI
   for y in 0 to 2 loop
      for x in 0 to 2 loop
         for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(x,y)(c) <= shl(inputA(x,y)(c), userInputA);
         end loop;
      end loop;
   end loop;

when x"0D" =>    -- DIV 2
   for y in 0 to 2 loop
      for x in 0 to 2 loop
         for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(x,y)(c) <= '0' & inputA(x,y)(c)(7 downto 1);
         end loop;
      end loop;
   end loop;

when x"0E" =>    -- DIVI
   for y in 0 to 2 loop
      for x in 0 to 2 loop
         for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(x,y)(c) <= shr(inputA(x,y)(c), userInputA);
         end loop;
      end loop;
   end loop;

when x"0F" =>    -- INV
   for y in 0 to 2 loop
      for x in 0 to 2 loop
         for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(x,y)(c) <= inputA(x,y)(c) xor "11111111";
         end loop;
      end loop;
```

```
    end loop;

when x"10" =>     -- AND
    for y in 0 to 2 loop
        for x in 0 to 2 loop
            for c in 0 to (IMAGE_CHANNELS-1) loop
                outputC(x,y)(c) <= inputA(x,y)(c) and inputB(x,y)(c);
            end loop;
        end loop;
    end loop;

when x"11" =>     -- OR
    for y in 0 to 2 loop
        for x in 0 to 2 loop
            for c in 0 to (IMAGE_CHANNELS-1) loop
                outputC(x,y)(c) <= inputA(x,y)(c) or inputB(x,y)(c);
            end loop;
        end loop;
    end loop;

when x"12" =>     -- NOR
    for y in 0 to 2 loop
        for x in 0 to 2 loop
            for c in 0 to (IMAGE_CHANNELS-1) loop
                outputC(x,y)(c) <= inputA(x,y)(c) nor inputB(x,y)(c);
            end loop;
        end loop;
    end loop;

when x"13" =>     -- XOR
    for y in 0 to 2 loop
        for x in 0 to 2 loop
            for c in 0 to (IMAGE_CHANNELS-1) loop
                outputC(x,y)(c) <= inputA(x,y)(c) xor inputB(x,y)(c);
            end loop;
        end loop;
    end loop;

when x"14" =>     -- Darken Highlights
    for y in 0 to 2 loop
        for x in 0 to 2 loop
            tempvar := unsigned(("000" & inputA(x,y)(0)) + ("000" &
                  inputA(x,y)(1)) + ("000" & inputA(x,y)(2)));
            tempvar := "00" & tempvar(10 downto 2);
            if (tempvar > unsigned(userInputA)) then
                for c in 0 to (IMAGE_CHANNELS-1) loop
                    if((inputA(x,y)(c) - userInputB) <= (inputA(x,y)(c))) then
                        outputC(x,y)(c) <= inputA(x,y)(c) - userInputB;
                    else
                        outputC(x,y)(c) <= (others => '0');
                    end if;
                end loop;
            else
                outputC(x,y)(0) <= inputA(x,y)(0);
                outputC(x,y)(1) <= inputA(x,y)(1);
                outputC(x,y)(2) <= inputA(x,y)(2);
```

```
          end if;
       end loop;
    end loop;

 when x"15" =>    -- Brighten Shadows
    for y in 0 to 2 loop
       for x in 0 to 2 loop
          tempvar := unsigned(("000" & inputA(x,y)(0)) + ("000" &
                inputA(x,y)(1)) + ("000" & inputA(x,y)(2)));
          tempvar := "00" & tempvar(10 downto 2);
          if (tempvar < unsigned(userInputA)) then
              for c in 0 to (IMAGE_CHANNELS-1) loop
                 if((inputA(x,y)(c) + userInputB) >= (inputA(x,y)(c))) then
                    outputC(x,y)(c) <= inputA(x,y)(c) + userInputB;
                 else
                    outputC(x,y)(c) <= (others => '1');
                 end if;
              end loop;
          else
              outputC(x,y)(0) <= inputA(x,y)(0);
              outputC(x,y)(1) <= inputA(x,y)(1);
              outputC(x,y)(2) <= inputA(x,y)(2);
          end if;
       end loop;
    end loop;

 when x"16" =>    -- Grayscale (Red channel)
    for y in 0 to 2 loop
       for x in 0 to 2 loop
          for c in 0 to (IMAGE_CHANNELS-1) loop
             outputC(x,y)(c) <= inputA(x,y)(0);
          end loop;
       end loop;
    end loop;

 when x"17" =>    -- Grayscale (Green channel)
    for y in 0 to 2 loop
       for x in 0 to 2 loop
          for c in 0 to (IMAGE_CHANNELS-1) loop
             outputC(x,y)(c) <= inputA(x,y)(1);
          end loop;
       end loop;
    end loop;

 when x"18" =>    -- Grayscale (Blue channel)
    for y in 0 to 2 loop
       for x in 0 to 2 loop
          for c in 0 to (IMAGE_CHANNELS-1) loop
             outputC(x,y)(c) <= inputA(x,y)(2);
          end loop;
       end loop;
    end loop;

when x"19" =>                    --pixel flip

       for c in 0 to 2 loop
```

```
            outputC(2,2)(c) <= inputA(0,0)(c);
            outputC(2,1)(c) <= inputA(0,1)(c);
            outputC(2,0)(c) <= inputA(0,2)(c);
            outputC(1,2)(c) <= inputA(1,0)(c);
            outputC(1,1)(c) <= inputA(1,1)(c);
            outputC(1,0)(c) <= inputA(1,2)(c);
            outputC(0,2)(c) <= inputA(2,0)(c);
            outputC(0,1)(c) <= inputA(2,1)(c);
            outputC(0,0)(c) <= inputA(2,2)(c);

         end loop;  -- c

   when x"1A" =>     -- Sobel Edge
      Gx := (others=>'0');
      Gy := (others=>'0');

      --sobel mask for gradient in horiz. direction
      Gx :=conv_signed(conv_integer((inputA(0,2)(0)-inputA(0,0)(0))
            +(shl((inputA(1,2)(0)-inputA(0,1)(0)), B"0000000001"))
            +(inputA(2,2)(0)-inputA(0,2)(0))),16);

      --sobel mask for gradient in vertical direction
      Gy :=conv_signed(conv_integer((inputA(0,0)(0)-inputA(0,2)(0))
            +(shl((inputA(1,0)(0)-inputA(1,2)(0)), B"0000000001"))
            +(inputA(0,2)(0)-inputA(2,2)(0))),16);

      Gsqd := conv_unsigned((conv_integer(Gx)**2)+ (conv_integer(Gy)**2),32);
      Gu := sqrt(Gsqd);

      if((Gu > conv_UNSIGNED(255, 16)) or (Gu < conv_UNSIGNED(70, 16))) then
         Gu :=  (others => '1');
      end if;

      for c in 0 to (IMAGE_CHANNELS-1) loop
            outputC(1,1)(c) <= conv_std_logic_vector(Gu,8) xor "11111111";
      end loop;

   when x"1B" =>     -- Sobel Chrome
      Gx := (others=>'0');
      Gy := (others=>'0');
      for y in 0 to 2 loop
         for x in 0 to 2 loop
            Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
                  (0))) * Sobelx(x,y)),16);
            Gy := Gy + conv_signed((conv_integer(unsigned(inputA(x,y)
                  (0))) * Sobely(x,y)),16);
         end loop;
      end loop;

      Gsqd := conv_unsigned((conv_integer(Gx)**2)+ (conv_integer(Gy)**2),32);
      Gu := sqrt(Gsqd);

      if(Gu > conv_UNSIGNED(255, 16)) then
         Gu :=  (others => '1');
      end if;
```

```
    for c in 0 to (IMAGE_CHANNELS-1) loop
          outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
    end loop;

when x"1C" =>     -- Prewitt
    Gx := (others=>'0');
    Gy := (others=>'0');
    for y in 0 to 2 loop
       for x in 0 to 2 loop
          Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
                (0))) * Prewittx(x,y)),16);
          Gy := Gy + conv_signed((conv_integer(unsigned(inputA(x,y)
                (0))) * Prewitty(x,y)),16);
       end loop;
    end loop;

    Gsqd := conv_unsigned((conv_integer(Gx)**2)+ (conv_integer(Gy)**2),32);
    Gu := sqrt(Gsqd);

    if(Gu > conv_UNSIGNED(255, 16)) then
       Gu :=  (others => '1');
    end if;

    for c in 0 to (IMAGE_CHANNELS-1) loop
          outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
    end loop;

when x"1D" =>     -- RobinsonN
    Gx := (others=>'0');

    for y in 0 to 2 loop
       for x in 0 to 2 loop
          Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
                (0))) * RobinsonN(x,y)),16);
       end loop;
    end loop;

    Gu := conv_unsigned(abs(Gx),16);

    if(Gu > conv_UNSIGNED(255, 16)) then
       Gu :=  (others => '1');
    end if;

    for c in 0 to (IMAGE_CHANNELS-1) loop
          outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
    end loop;

when x"1E" =>     -- RobinsonNW
    Gx := (others=>'0');

    for y in 0 to 2 loop
       for x in 0 to 2 loop
          Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
                (0))) * RobinsonNW(x,y)),16);
       end loop;
```

```
    end loop;

    Gu := conv_unsigned(abs(Gx),16);

    if(Gu > conv_UNSIGNED(255, 16)) then
       Gu :=  (others => '1');
    end if;

    for c in 0 to (IMAGE_CHANNELS-1) loop
          outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
    end loop;

 when x"1F" =>     -- RobinsonNE
    Gx := (others=>'0');

    for y in 0 to 2 loop
       for x in 0 to 2 loop
          Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
                (0))) * RobinsonNE(x,y)),16);
       end loop;
    end loop;

    Gu := conv_unsigned(abs(Gx),16);

    if(Gu > conv_UNSIGNED(255, 16)) then
       Gu :=  (others => '1');
    end if;

    for c in 0 to (IMAGE_CHANNELS-1) loop
          outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
    end loop;

 when x"20" =>     -- RobinsonE
    Gx := (others=>'0');

    for y in 0 to 2 loop
       for x in 0 to 2 loop
          Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
                (0))) * RobinsonE(x,y)),16);
       end loop;
    end loop;

    Gu := conv_unsigned(abs(Gx),16);

    if(Gu > conv_UNSIGNED(255, 16)) then
       Gu :=  (others => '1');
    end if;

    for c in 0 to (IMAGE_CHANNELS-1) loop
          outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
    end loop;

 when x"21" =>     -- RobinsonW
    Gx := (others=>'0');

    for y in 0 to 2 loop
```

```
      for x in 0 to 2 loop
         Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
               (0))) * RobinsonW(x,y)),16);
      end loop;
   end loop;

   Gu := conv_unsigned(abs(Gx),16);

   if(Gu > conv_UNSIGNED(255, 16)) then
      Gu :=  (others => '1');
   end if;

   for c in 0 to (IMAGE_CHANNELS-1) loop
         outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
   end loop;

when x"22" =>    -- RobinsonS
   Gx := (others=>'0');

   for y in 0 to 2 loop
      for x in 0 to 2 loop
         Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
               (0))) * RobinsonS(x,y)),16);
      end loop;
   end loop;

   Gu := conv_unsigned(abs(Gx),16);

   if(Gu > conv_UNSIGNED(255, 16)) then
      Gu :=  (others => '1');
   end if;

   for c in 0 to (IMAGE_CHANNELS-1) loop
         outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
   end loop;

when x"23" =>    -- RobinsonSE
   Gx := (others=>'0');

   for y in 0 to 2 loop
      for x in 0 to 2 loop
         Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
               (0))) * RobinsonSE(x,y)),16);
      end loop;
   end loop;

   Gu := conv_unsigned(abs(Gx),16);

   if(Gu > conv_UNSIGNED(255, 16)) then
      Gu :=  (others => '1');
   end if;

   for c in 0 to (IMAGE_CHANNELS-1) loop
         outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
   end loop;
```

```
when x"24" =>    -- RobinsonSW
   Gx := (others=>'0');

   for y in 0 to 2 loop
      for x in 0 to 2 loop
         Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
               (0))) * RobinsonSW(x,y)),16);
      end loop;
   end loop;

   Gu := conv_unsigned(abs(Gx),16);

   if(Gu > conv_UNSIGNED(255, 16)) then
      Gu :=  (others => '1');
   end if;

   for c in 0 to (IMAGE_CHANNELS-1) loop
         outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
   end loop;

when x"25" =>    -- average Grayscale
   Gx := (others=>'0');

   for y in 0 to 2 loop
      for x in 0 to 2 loop
         Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
               (0))) * Average(x,y)),16);
      end loop;
   end loop;

   Gu := "0000" & conv_unsigned(abs(Gx),16)(15 downto 4);

   if(Gu > conv_UNSIGNED(255, 16)) then
      Gu :=  (others => '1');
   end if;

   for c in 0 to (IMAGE_CHANNELS-1) loop
         outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
   end loop;


when x"26" =>                  --average color
    tempsignal_red := ("000"&inputA(0,0)(0)) + ("000"&inputA(0,1)(0))
               + ("000"&inputA(0,2)(0)) + ("000"&inputA(1,0)(0)) +
               ("000"&inputA(1,2)(0)) + ("000"&inputA(2,0)(0)) +
               ("000"&inputA(2,1)(0)) + ("000"&inputA(2,2)(0));

    tempsignal_green := ("000"&inputA(0,0)(1)) + ("000"&inputA(0,1)(1))
               + ("000"&inputA(0,2)(1)) + ("000"&inputA(1,0)(1)) +
               ("000"&inputA(1,2)(1)) + ("000"&inputA(2,0)(1)) +
               ("000"&inputA(2,1)(1)) + ("000"&inputA(2,2)(1));

    tempsignal_blue := ("000"&inputA(0,0)(2)) + ("000"&inputA(0,1)(2))
               + ("000"&inputA(0,2)(2)) + ("000"&inputA(1,0)(2)) +
               ("000"&inputA(1,2)(2)) + ("000"&inputA(2,0)(2)) +
               ("000"&inputA(2,1)(2)) + ("000"&inputA(2,2)(2));
```

```
     outputC(1,1)(0) <= tempsignal_red(10 downto 3);
     outputC(1,1)(1) <= tempsignal_green(10 downto 3);
     outputC(1,1)(2) <= tempsignal_blue(10 downto 3);

when x"27" =>    -- UserFilter
   Gx := (others=>'0');

   for y in 0 to 2 loop
      for x in 0 to 2 loop
         Gx := Gx + conv_signed((conv_integer(unsigned(inputA(x,y)
               (0))) * userKernel(x,y)),16);
      end loop;
   end loop;

   Gu := conv_unsigned(abs(Gx),16);

   if(Gu > conv_UNSIGNED(255, 16)) then
      Gu :=  (others => '1');
   end if;

   for c in 0 to (IMAGE_CHANNELS-1) loop
         outputC(1,1)(c) <= conv_std_logic_vector(Gu,8);
   end loop;

when x"28" =>                   --min per channel

  tempsignal_red(7 downto 0)   := inputA(0,0)(0);
  tempsignal_green(7 downto 0) := inputA(0,0)(1);
  tempsignal_blue(7 downto 0)  := inputA(0,0)(2);

  for x in 0 to 2 loop
    for y in 0 to 2 loop
      if tempsignal_red(7 downto 0) > inputA(x,y)(0) then
        tempsignal_red(7 downto 0) := inputA(x,y)(0);
      end if;
    end loop;  -- y
  end loop;  -- x

  for x in 0 to 2 loop
    for y in 0 to 2 loop
      if tempsignal_green(7 downto 0) > inputA(x,y)(1) then
        tempsignal_green(7 downto 0) := inputA(x,y)(1);
      end if;
    end loop;  -- y
  end loop;  -- x

  for x in 0 to 2 loop
    for y in 0 to 2 loop
      if tempsignal_blue(7 downto 0) > inputA(x,y)(2) then
        tempsignal_blue(7 downto 0) := inputA(x,y)(2);
      end if;
    end loop;  -- y
  end loop;  -- x

  outputC(1,1)(0) <= tempsignal_red(7 downto 0);
```

```
          outputC(1,1)(1) <= tempsignal_green(7 downto 0);
          outputC(1,1)(2) <= tempsignal_blue(7 downto 0);

        when x"29" =>                    --max per channel

          tempsignal_red(7 downto 0)   := inputA(0,0)(0);
          tempsignal_green(7 downto 0) := inputA(0,0)(1);
          tempsignal_blue(7 downto 0)  := inputA(0,0)(2);

          for x in 0 to 2 loop
            for y in 0 to 2 loop
              if tempsignal_red(7 downto 0) < inputA(x,y)(0) then
                tempsignal_red(7 downto 0) := inputA(x,y)(0);
              end if;
            end loop;  -- y
          end loop;  -- x

          for x in 0 to 2 loop
            for y in 0 to 2 loop
              if tempsignal_green(7 downto 0) < inputA(x,y)(1) then
                tempsignal_green(7 downto 0) := inputA(x,y)(1);
              end if;
            end loop;  -- y
          end loop;  -- x

          for x in 0 to 2 loop
            for y in 0 to 2 loop
              if tempsignal_blue(7 downto 0) < inputA(x,y)(2) then
                tempsignal_blue(7 downto 0) := inputA(x,y)(2);
              end if;
            end loop;  -- y
          end loop;  -- x

          outputC(1,1)(0) <= tempsignal_red(7 downto 0);
          outputC(1,1)(1) <= tempsignal_green(7 downto 0);
          outputC(1,1)(2) <= tempsignal_blue(7 downto 0);


                    -- INSERT MORE INSTRUCTIONS HERE!

        when x"FF" =>   -- NOP
           null;
        when others =>
           report "NOT A VALID OPCODE!" severity FAILURE;
      end case;
   end process;
end CellArch;
```

Simulation and Testing

Simulation was done using Mentor Graphics Questa 6.3g. Since I, Eric Liskay, came up with the idea for an SIMD image processor, I had to first prove that reading images into an array using a HDL was possible. Initially, I used SystemVerilog since I was more familiar with the language. The description of the code in SystemVerilog turned out to be much more concise and easy to read than the VHDL in the previous pages. On the next page is the SV code that I created to read in an image and print out the input to the screen in ASCII. I used this to determine how the RAW file was formatted and how to correctly read it into a two dimensional array.

I then created an equivalent VHDL program to prove that this could be done in the VHDL as well. The equivalent VHDL code, performing the exact same operation and display can be seen on the following 2(!) pages.

The majority of testing and debugging that I did involved looking at the output image. If it was not displaying correctly, it was usually pretty obvious where the problem was. Once the image processor controller was working correctly, there was no need to look at the waveform output. I created a testbench which instantiates the ImageProcessor, sets inputs, and performs processing on all the opcodes in sequence. The testbench can be seen on later pages.

For simulation, I found that Questa(32-bit) ran out of memory if the image size was much larger than 600x390 pixels. With a 64-bit simulator, I would have been able to process larger images.

With a clock period of 20 ns, a 600x390 image took 5.2 μs to process in regular mode. The same image took 46.8 μs to process in convolution mode. This includes the time it takes to read the image from the drive, process the operation, and write the output image to the drive. In real time, a regular mode image took on average 0.76s to process. The same image in convolution mode took on average 2.7s to process in real time.

I attempted to synthesize the image processor in Altera Quartus, expecting to get errors on the file open commands. However, Quartus gets to the fitting stage of synthesis and runs into that program’s 32-bit memory limit. In order to run in 64-bit mode, I would have to use the purchased “subscription edition” version of Quartus.

ImageTest.sv (used in initial testing and proof-of-concept)

```
module  imagetest();
  integer file, r;
  int i,j;
  typedef struct packed {
    byte unsigned R;
    byte unsigned G;
    byte unsigned B;
  } pixel;
  pixel array[150][150]; // [height][width]

  initial begin
    file = $fopen("RGBint150.raw", "rb");
    r =$fread(array,file);
    for(i = 0; i < $size(array,1); i++) begin
      for (j = 0; j < $size(array,2); j++) begin
        if (array[i][j] == 0)
          $write("*");
        else if (array[i][j] == '1)
          $write(".");
        else if (array[i][j].R == '1)
          $write("R");
        else if (array[i][j].G == '1)
          $write("G");
        else if (array[i][j].B == '1)
          $write("B");
        else
          $write("?");
      end
      $write("\n");
    end
  end
endmodule
```

ImageTest.vhd (used in initial testing and proof-of-concept)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_textio.all;
use ieee.std_logic_arith.all;
use std.textio.all;


entity imageTest is
end imageTest;

architecture main of imageTest is

function to_string(sv: Std_Logic_Vector) return string is
    use Std.TextIO.all;
    variable bv: bit_vector(sv'range) := to_bitvector(sv);
    variable lp: line;
  begin
    write(lp, bv);
    return lp.all;
end;

begin
   process is
      variable i,j,c : integer;

      subtype channel is std_logic_vector(7 downto 0);
      type pixel is array (2 downto 0) of channel;
      type pixelArray1d is array (149 downto 0) of pixel;
      type pixelArray2d is array (149 downto 0) of pixelArray1d;
      variable pixelarray : pixelArray2d;

      type character_file is file of character;
      file myfile: character_file;
      variable character_variable : character;
      variable my_line : line;

      variable fstatus: FILE_OPEN_STATUS;

      begin
         file_open(fstatus, myfile, "RGBint150.raw", read_mode);
         assert (fstatus = open_ok);

         for i in 0 to 149 loop
          for j in 0 to 149 loop
            for c in 0 to 2  loop
```

```
          read(myfile, character_variable);
          pixelarray(i)(j)(c) := CONV_STD_LOGIC_VECTOR(character'pos(character_variable), 8);
        end loop;
        if ((pixelarray(i)(j)(0) = b"00000000") and
           (pixelarray(i)(j)(1) = b"00000000") and
           (pixelarray(i)(j)(2) = b"00000000"))then
            write(my_line, string'("*"));
          elsif ((pixelarray(i)(j)(0) = b"11111111") and
                (pixelarray(i)(j)(1) = b"11111111") and
                (pixelarray(i)(j)(2) = b"11111111"))then
            write(my_line, string'("."));
          elsif (pixelarray(i)(j)(0) = b"11111111")then
            write(my_line, string'("R"));
          elsif (pixelarray(i)(j)(1) = b"11111111") then
            write(my_line, string'("G"));
          elsif (pixelarray(i)(j)(2) = b"11111111") then
            write(my_line, string'("B"));
          else
            write(my_line, string'("?"));
          end if;
      end loop;
      writeline(output,my_line);
     end loop;

     wait;
  end process;
end main;
```

Testbench.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use std.textio.all;
use ieee.STD_LOGIC_TEXTIO.all;
use ieee.std_logic_arith.CONV_STD_LOGIC_VECTOR;
use ieee.numeric_std.all;
use work.pkg.all;


entity testbench is
end testbench;

architecture test of testbench is

   signal number     : integer := 0;
   signal clock      : std_logic := '0';
   signal go         : std_logic := '0';
   signal opcode     : std_logic_vector(7 downto 0) := x"FF";
   signal userInputA : std_logic_vector(7 downto 0) := b"00000000";
   signal userInputB : std_logic_vector(7 downto 0) := b"00000000";
   signal complete   : std_logic := '0';
   signal userKernel : kernel := ((0,0,0),(0,0,0),(0,0,0));

   component ImageProcessor is
       generic (
         IMAGE_A_NAME    : string  := "toucan600x390.raw"; -- Filename of Image A
        IMAGE_B_NAME    : string  := "toucan600x390.raw"; -- Filename of Image B
         IMAGE_C_NAME    : string  := "output"; -- Filename of Image C
         IMAGE_WIDTH     : integer := 600; -- Width of image(s) in pixels
         IMAGE_HEIGHT    : integer := 390; -- Height of image(s) in pixels
         IMAGE_CHANNELS : integer := 3 -- Channels: 3=RGBColor, 1=Grayscale
      );
      port (
         clock       : in  std_logic := '0';
         go          : in  std_logic := '0';
         number      : in  integer := 0;
         opcode      : in  std_logic_vector(7 downto 0);
         userInputA  : in  std_logic_vector(7 downto 0);
         userInputB  : in  std_logic_vector(7 downto 0);
         userKernel  : in  kernel;
         complete    : out std_logic := '0'
      );
   end component;
```

```
begin

   I1 : ImageProcessor port map (clock, go, number, opcode, userInputA,
                                 userInputB, userKernel, complete);

   clock <= not clock after 10 ns;

   opcode <= std_logic_vector(to_unsigned(number,8));
   userInputA <= B"00010000";
   userInputB <= B"01000000";
   userKernel <= ((0,-1,0),(-1,4,-1),(0,-1,0));

   process(complete) is
   begin
      assert (number<50) report "Finished processing!" severity FAILURE;
      if(not((number=0) and (complete = '0'))) then
         go <= not go;
         number <= number + 1;
      end if;
   end process;
end test;
```

Comparison between VHDL and System Verilog:

1. In System Verilog Package is used more extensively, this can be imported by any module in their $Compilation Unit and can be referenced. Functions and tasks can be defined in package and can be called by any module by referencing it. It helps to avoid unnecessary repetition.

   While in VHDL use of package is limited.

2. System Verilog introduces **unique Case**, which states that only one condition could be true at a time. It makes very easy for fault verification.

```
function automatic pixel operate(byte userinput,byte opcode, pixel X,
pixel Y);
begin
pixel Z;
unique case(opcode)

8'h00:begin

            Z.R  = (X.R + Y.R);
            Z.G   = (X.G + Y.G);
             Z.B   = (X.B + Y.B);
        end

8'h01:begin

            Z.R   = (X.R);
            Z.G   = (X.G);
            Z.B   = (X.B);
        end
default: $info("Invalid Opcode");

endcase
```

   There is no such tool in VHDL.

3. System Verilog makes very easy to read any file with a single line Code.

```
     file1 = $fopen("toucan.raw", "rb");
             r =$fread(ImageA,file1);
                     $fcloe(file1);
            file1 = $fopen("toucan.raw", "rb");
            r =$fread(ImageB,file1);
                    $fclose(file1);
```

But in VHDL we have to define the width and height of file and have to read it sequentially and have to go through long iterations in loops.

```
 file_open(fstatus, imageA, IMAGE_A_NAME, read_mode);
assert (fstatus = open_ok) report "imageA not found" severity FAILURE;
file_open(fstatus, imageB, IMAGE_B_NAME, read_mode);
assert (fstatus = open_ok) report "imageB not found" severity FAILURE;


for h in 0 to (IMAGE_HEIGHT-1) loop
   for w in 0 to (IMAGE_WIDTH-1) loop
      for c in 0 to (IMAGE_CHANNELS-1)  loop
         read(imageA, char);
         pixelBufferA(w)(h)(c) <= CONV_STD_LOGIC_VECTOR(character'pos(char), 8);
         read(imageB, char);
         pixelBufferB(w)(h)(c) <= CONV_STD_LOGIC_VECTOR(character'pos(char), 8);
        end loop;
        end loop;
       end loop;

    file_close(imageA);
            file_close(imageB);
```

4. System Verilog also introduces concept of Randomization which produces random test stimulus for the verification of design. It gives maximum verification with least number of test inputs and sometime very helpful to identify those errors which are not detected easily by normal testing.

   While VHDL is not so accurate for verification of design and have to verify with regular testing inputs.

System Verilog Code for Simple Reading and Writing the output File:

```
include "pack.sv"

 module Imageprocessor(input logic clock,byte number, opcode,byte
                      userinput,bit control,output logic complete);

     int r,w;                                        //Image Reading
     int i,j,file,file1;
     bit ready,done,iscomplete;
     always_comb

       begin

        a1:assert($isunknown(control==0))
        else $error("Grant not asserted");
      end



          pixel ImageA[330][500],ImageB[330][500],ImageC[330][500];

          initial begin
             file1 = $fopen("toucan.raw", "rb");

             $info("Working on Image %d",number);
             ready = 0;

             r =$fread(ImageA,file1);
             $fclose(file1);

           end


                initial begin

             file1 = $fopen("toucan.raw", "rb");

            r =$fread(ImageB,file1);
             $fclose(file1);


         ready=1;
       end
```

```
always_comb
    begin
        if(ready==1)
            begin
        if (control==0)
                begin
                 for(i= 0; i <330; i++) begin                    //
operation
                         for (j = 0; j <500; j++) begin

                          ImageC[i][j] =
operate(userinput,opcode,ImageA[i][j],ImageB[i][j]);

                                end
           end
         a2:assert(ImageC[i][j].R <=255)
                 else $error("Maximum red");

                 a3:assert(ImageC[i][j].G <=255)
                 else $error("Maximum Green");

                 a4:assert(ImageC[i][j].R <=255)
                 else $error("Maximum Blue");
               end
          else
               begin

        for(i= 0; i <330; i++) begin                        //operation
                 for (j = 0; j <500; j++) begin

                 ImageC[i][j] = ImageA[i][j];
                      end
                end

        for(i= 62; i<215; i++) begin                        //operation
                for (j = 185; j <300; j++) begin

                ImageC[i-62][j] = ImageB[i][j];

                                 end
                  end
               end
      done=1;

          if(done==1)
```

```
      begin
        $write(" doone = %b ",done);
        file= $fopen("neeraj.raw","wb");

      for(i = 0; i < 330; i++) begin
      for (j = 0; j <500; j++) begin

      $fwrite(file,"%s",ImageC[i][j].R);
      $fwrite(file,"%s",ImageC[i][j].G);
      $fwrite(file,"%s",ImageC[i][j].B);
                          end
                       end
     $fclose(file);
       end

end

complete=(done*ready);

 a5:assert(complete==1)
         else $error("Maximum Blue");

end
endmodule
```

Future Improvements

1. Our Current SIMD Processor is able to Process a tile of pixels only one time but if we can improve it process a tile second time as well before writing the output file then we can develop many more applications. Like Image Flipping like horizontal flipping, vertical flipping, 90,180,270 degree clockwise or counterclockwise rotation, jumbling of tiles etc.

Original Image

Horizontally Flipped Image

2. Data bus between pixel buffers to Processor is 900*24 bits, this could create a problem. So, we will try to reduce the data load on this bus while maintaining the parallelism and speed.

3. Major changes to the code and design would be needed to make the design synthesizable. Images would have to be loaded into the memory on the board somehow since right now, VHDL commands are just reading the image file from the computer’s drive.
    a. One approach to emulating the design would be using Mentor Graphics’ Veloce. With Veloce, the non-synthesizable portions such as the testbench and file I/O would be located in the HVL(Hardware Verification Language) space. The image processor controller and the cell entities would be located in the HDL(Hardware Description Language) space.
4. In order to reduce memory usage during simulation and process larger images, only parts of the image could be loaded into the pixel buffer at a time. This would increase file I/O and probably processing time, but would decrease memory usage.
5. A parameterized kernel size could be added without many changes to the code to enable larger kernels.
6. Other convolution instructions could be added such as noise reduction. With larger kernels, convolution procedures such as a 9x9 average filter could be added.
7. For the operations “Darken Highlights” and “Brighten Shadows”, a gradient threshold could be added to prevent the abrupt transition between processed and un-processed pixels.

Results

Please see the presentation slides for examples of the output images for each opcode.

References

All Photographs © Eric Liskay

Lecture 10. Edge detection. Sobel and similar filters. Convolution
http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL_99/2012/2012-lecture010_edge-detection-SobelG.ppt

Wikipedia - Sobel Operator
http://en.wikipedia.org/wiki/Sobel_operator

A VHDL Function for finding SQUARE ROOT
http://vhdlguru.blogspot.com/2010/03/vhdl-function-for-finding-square-root.html

Tools

- Adobe Photoshop CS6
- Mentor Graphics Questa© Advanced Simulator 6.3g
- Altera Quartus II Web Edition v12.0