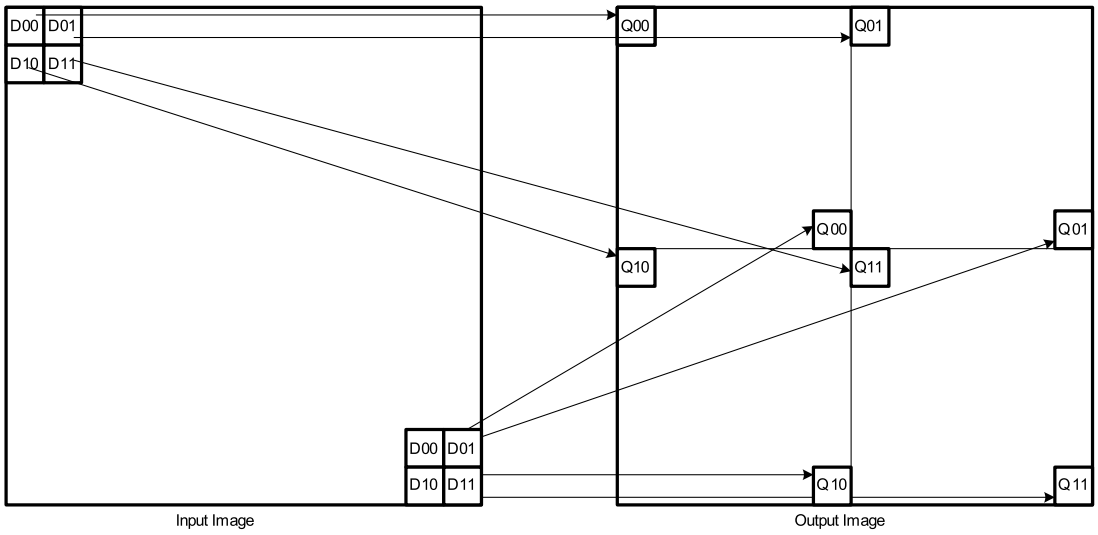
CSE P567 Homework #4 Winter 2010

Due: Tuesday, Feb 9, in Lab

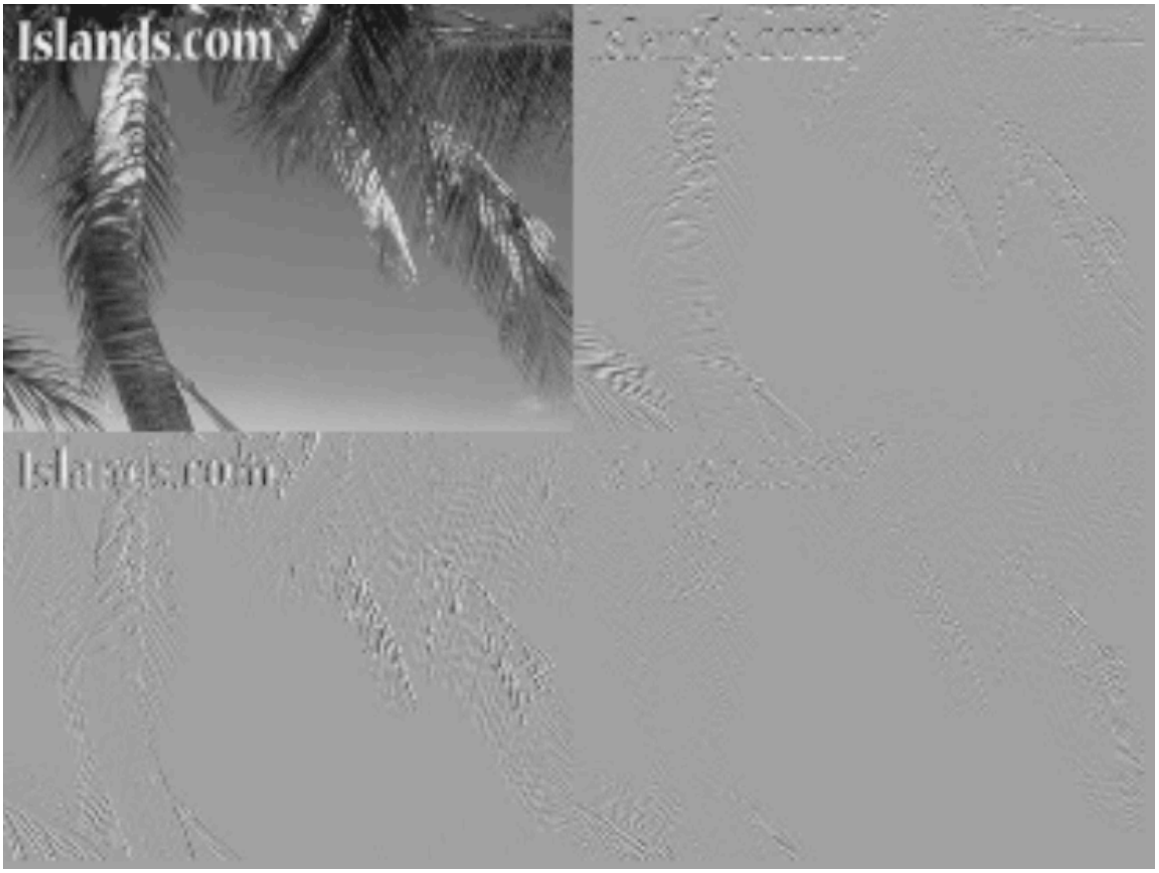
There is no reading for this homework assignment. You want to glance at Combinational Logic Synthesis for LUT-Based FPGAs (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.3571) in preparation for next week's lecture, which will be a tour through logic synthesis land presented by Ramine Roane.

2. The final design we will be implementing using the FPGA is a simple 2D wavelet transform that will transform the input image into 4 images each ¼ the size of the original. The basic idea is that most of the image information will be in one of these images (Q00), with only residual information in the other 3. These 3 subimages can then be compressed since the residuals are generally very small. The transform can also be applied recursively to the output frames to get even better compression.



The next page has two before and after images, where we have first applied gray scale conversion and then the wavelet transform. You can see how most of the information is in one quadrant and that the residuals are pretty small. (The output image was produced by a simulation of the solution hardware – more later.)


Islands.com


Islands.com

As shown in the above figure, we will label the pixels in the input 2x2 block D00, D01, D10, and D11 and the output block Q00, Q01, Q10, and Q11. The equation for each Q is:

Q00 = D00 + D01 + D10 + D11
Q01 = D00 – D01 + D10 – D11
Q10 = D00 + D01 – D10 – D11
Q11 = D00 – D01 – D10 + D11

Note that D00 = (Q00+Q01+Q10+Q11)/4. Similar equations can recover D01, D10 and D11 so this transform preserves the information in the original image.

There are essentially just two problems that you have to solve.

1. Collect each 2x2 block of input pixels D00 – D11, one block at a time. Since the pixels are streaming in, in row order (raster scan), this is a bit of a challenge. The key is that you have to save the even row of pixels (e.g. row 0) in a memory so that when you get to the odd row (e.g. row 1), you can read the pixels from the even row (e.g. row 0). We have given you the memory module below that you can use to save these rows of pixels. This is a dual-ported memory since you can read and write to the memory on the same clock cycle. The memory is straightforward: The memory is always reading even if you don't use the data – readData is the value at readAddr. If you assert the write signal, the writeData is written to the writeAddr at the end of this clock cycle. Reads and writes can occur at the same time.

```
module memory(
  input clk,
  input [8:0] readAddr,
  output [7:0] readData,
  input [8:0] writeAddr,
  input write,              // Assert to enable write to memory
  input [7:0] writeData);


  reg [7:0] memory [0:511];
  assign readData = memory [readAddr];
  always @(posedge clk) begin
    if (write) memory[writeAddr] <= writeData;
  end
endmodule;
```

2. Once you have the 2x2 block of input data, you need to compute the values for Q00 – Q11 and output them with the correct pixel address. Q00 goes to the sub-image in quadrant 00 (upper left), Q01 goes to quadrant 01 (upper right) and so on. You should recall the 1D wavelet example from Lab 2 and use a similar method to compute the Q's. Note that when you are reading an even row, you cannot produce any pixels, while when you are reading an odd row, you have to generate 4 pixels for every 2 pixels you get. No problem: the input pixels only arrive every 2 clock periods. You must use the input pixelValidIn signal to know when to process an input pixel, and you must generate an output pixelValidOut when you generate an output pixel (Q). And you have to generate a valid output address at the same time you assert pixelValid!

Pay Attention to the Following!

He are providing a template for the filter.v that you should use to perform the wavelet transform – see the Web page for this and the other files that you will need including the test fixture. This filter module has much of the solution filled in to save some work. However, there are several places where the comment /* FILL THIS IN */ appears. This is where you need to figure out what needs to be filled in. You will have to understand what the rest of the code is doing to do this, but it is relatively straightforward.

I strongly recommend that you proceed incrementally using the enable switch, which you can set to 0 or 1 in the test fixture when you are testing your design. You can use the switch as well when you are testing the design in hardware of course.

0. Test the simple pass-through (enable=0) just to get started.

1. Test the memory. The output of the memory is the variable evenGray. After you have decided how to connect the other memory I/Os, connect the module color outputs to evenGray (instead gray) to see that the memory is working.

2. Test the pixel addressing on the output (Lab 4/Part 3). Using the enable code again, modify the pixel addressing to put pixels in the 4 different quadrants.

3. Using the 4 states, which write the Q's to the 4 different quadrants, just write constant values in each of the quadrants. e.g. make one black, one white, etc.

4. Now you are ready to do the wavelet computation.

We have constructed a test fixture for you to test your design. This test fixture reads an entire image from an input file and writes out a file with the image produced by the simulation. This file can be turned into an image that you can look at. We are using ppm format, which is a nice text format with RGB values. When you plug your design into the camera pipeline, you should see the same behavior.

We will post the solution bit file so you can see how it works in hardware.

The Test Procedure

1. Convert a ppm image to the simulation input format:

```
ppm2rgb foo.ppm > imageIn.dat
```

2. Run the simulation - takes a few seconds to process the whole image. The simulation will print out that it is reading the imageIn.dat file and then that it is writing the imageOut.dat file. These files must be in the simulation directory.

3. Convert the simulation output file to an image:

```
rgb2ppm imageOut.dat > newfoo.ppm
```

Note that if there are any x's in the output file, the image will be garbaged.

There are programs that will convert jpg's to ppm's and vs. We will give you a few ppm images to use. The photo used in the example on the previous page is the beach.ppm image.

We will give you the source for the conversion programs. I believe they are vanilla C programs that should compile anywhere. (They are a bit crufty but they work.)

Some Notes:

1. The wavelet can operate on either gray-scale or color. Don't try color unless you have time on your hands.
2. ** The wavelet outputs need 10 bits for complete accuracy – you need to scale these down to 8 bits.
3. ** Also note that there are negative values produced. Make your life easy and use signed variables, e.g.
   ```
   reg signed [10:0] tempValue;
   ```
4. How will negative values look on the output? The residual values will be numbers (generally) between -8 and +8. You might want to display these so that they look "natural".
5. It turns out that you can change the exposure setting of the camera using switch 0 and button 1. Depending on the setting of the switch, pressing button 1 will cause the exposure to increase or decrease.

** These have already been provided in the template code.