Solution for Wavelet Image Compression E2LP Laboratory Exercise

There are many possible ways of approaching this assignment. This document describes only one of these solutions, which may not be optimal, but demonstrates the required functionality.

We approach this assignment by splicing into small components that can be addressed and designed separately:

1. ROM component containing the original greyscale image. This component should be implemented using a block RAM, otherwise it will consume a lot, if not all of the FPGA device. For simplicity, the ROM could be preloaded from a file on disk containing the original input image.
2. Wavelet image compression component. This is a composite consisting of several key components that will be discussed further in the following sections.
3. RAM component that will be used to retain the solution of the exercise.

This top level design is depicted in the following image:

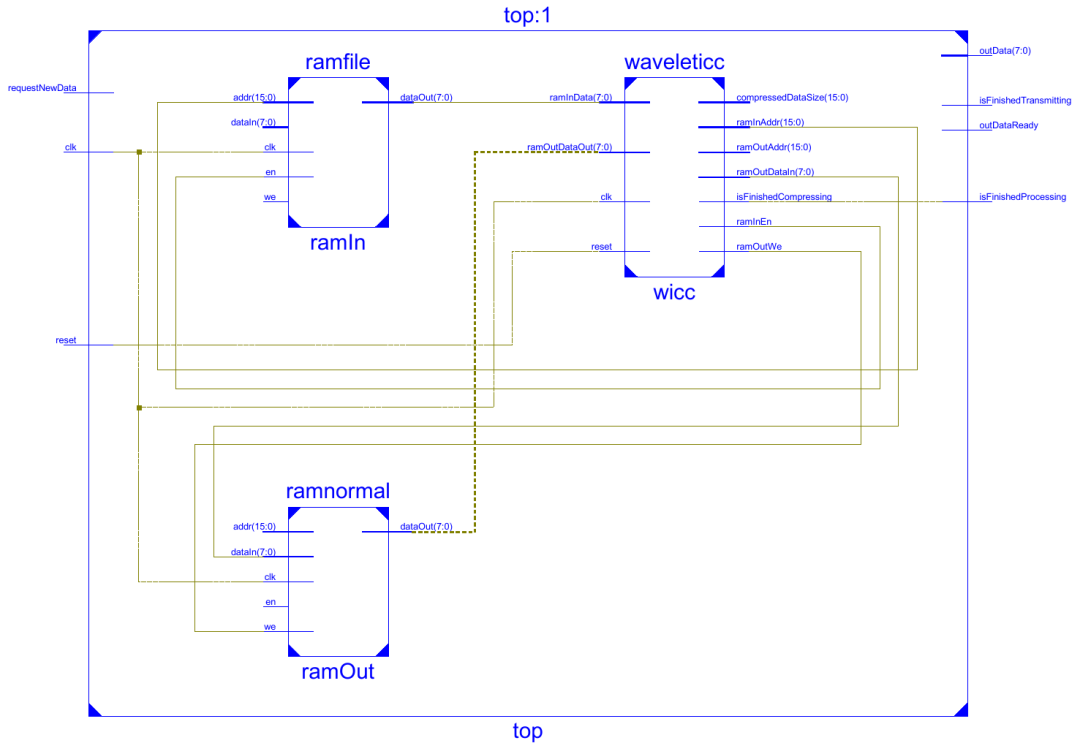

Figure 1. Inital top-level design.

In Figure 1, input ROM is denoted with "ramIn", output RAM with "ramOut" and the Wavelet Image Compression Component with "wicc".

To allow that the RAM contents be accessed after the image compression process is concluded, we expand this top level design with an additlonal "transmitter" component which can be used to stream the contents of the RAM to the output of the device:

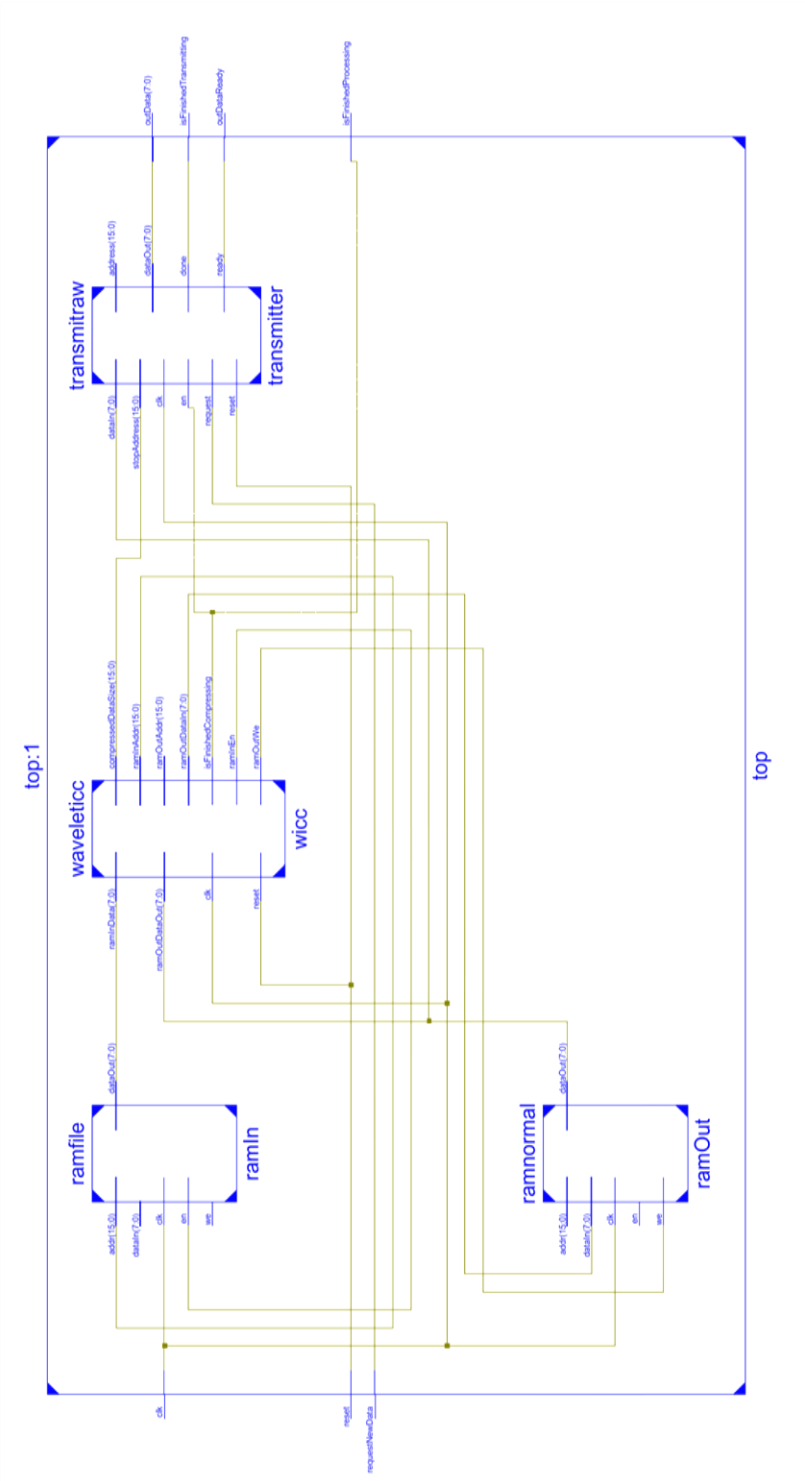

Figure 2. Updated design of the top-level schematic.

The design of the described components is given in the following text.

1 ROM component

Utilizing block RAM for memory is a relatively simple task. This is due to existance of Language Templates within the Xilinx IDE (Edit -> Language Templates).

There are two difficulties that a student needs to address to successfuly design a pre-initialized ROM component:

1. Modify the block RAM template to enable loading of data from a file during initialization.
2. Preprocess an image (outside of Xilinx IDE) in order to convert it to the appropriate data format (dependant on the implemented VHDL loading function in the previous step).

The ROM component is implemented in `ramfile.vhd` file located in the bundle of this solution. The data format expected by the implemented loading function is defined as:

- The file contains data for a single grayscale image of dimensions 256x256 pixels, where each pixel is encoded as an unsigned byte (bit vector of size 8).
- File contains a list of pixels, each pixel in one line. The pixels are writen as bit-vectors in ASCII format (8 simbols in each line, containing either a '0' or a '1' as their value).
- Pixels are stored from the top left to bottom right, line by line from left to right.

For the purpose of supplying the appropriate data, a Python script for conversion of images in this format is provided in the bundle. This script addresses the second of the two difficulties.

2 RAM component for retaining the solution

This component can be implemented straightforward according to the examples in Language Templates.

One should keep in mind that this component needs to be used both by the wavelet image compression component, and that users from the outside will need to access the RAM to access the results of the compression.

This component is implemented in `ramnormal.vhd` file.

3 Wavelet Image Compression Component (ICC)

The central part of the wavelet image compression is the wavelet transform. However, implementing the Haar low-pass and high-pass filters are by far the simplest steps in designing the ICC. The expressions for the low-pass and the high-pass filters are given in (1) and (2):

$$H_0(z) = \frac{1}{\sqrt{2}}(1 + z^{-1}) \qquad (1)$$

$$H_1(z) = \frac{1}{\sqrt{2}}(1 - z^{-1}) \tag{2}$$

In other words, the low-pass filter can be implemented as an average between the current value of a signal, and the previous value (signal delayed by 1 clock):

$$y_L = \frac{x[n] + x[n-1]}{2} \tag{3}$$

Intuitively, the high-pass filter is calculated by determining the distance between the average and one of the signal values (or, in other words, the difference between the current and the previous signal value, divided by 2):

$$y_H = \frac{x[n] - x[n-1]}{2} \tag{4}$$

These expressions can be implemented directly in VHDL as combinatorial components in a behavioral design of the filters (as implemented in `haarlp.vhd` and `haarhp.vhd`).

These filters will be used both for filtering the image in rows, and in columns. What has not yet been discussed is how the row and column delays will be implemented.

Also, note that wavelet image decomposition requires decimation, as depicted in the introduction section of this laboratory exercise's description.

There are several major difficulties when proceeding with the design of the IIC:

1. Implementing the delay for rows and columns of the image.
2. Accessing the data from the input ROM.
3. Decimating the obtained values after the decomposition.
4. Compressing the image
5. Storing the data in the output RAM.

3.1 Row and column delays – FIFO Memory

The simplest way of thinking of delays in digial design is using shift registers. Shift registers are usually used in 1-bit version, where data is passed through one bit at a time, and thus a single shift register contains only n-bits (e.g. a byte of data). In our case, a single register needs to be n-bit wide, and we need to stack several of them in a row.

To simplify things, we implement the delay operation as a small memory device, where the number of n-bit registers can be predetermined through a generic parameter. This is especially useful, because the same component can be used both for delaying data in a single row (column delay), and between two rows (row delay).

To explain, here is a simple example. Delaying the data in a single row is straight-forward: if we instantiate a FIFO memory device of size 2, we can store the current and the previous byte of data (data is shifted in on clock rising edge, one byte at a time). Delaying the data between rows can be done analogously – instead of having a FIFO memory of size 2, we could instantiate an array of size (IMAGE_WIDTH + 1). This way, the first and the last element of our FIFO memory will present the pixel in the current row, and the corresponding pixel in the previous row.

Example design for this component can be found in `fifomem.vhd` file.

3.2 Accessing the data from the input ROM – Control Unit

The data from the ROM needs to be accessed sequentially, in a controlled manner. As in many applications, this is done by desigining a Control Unit, a custom one in this case. The CU implements a state machine which sends the control signals to other components of the ICC to control the data flow in the device. Its operation can be described in the following manner:

1. Initial reset state, where everything is brought down to the default values, and all registers cleared.
2. Fetch the next pixel of the original image. Stream it through the row delay component (FIFO_ROW, size of array is 2). Wait for the new data to be written to the FIFO memory.
3. The data from the FIFO_ROW is passed to Haar LP and Haar HP filters in parallel.
4. Write the data to FIFO_COLUMN_L and FIFO_COLUMN_H components (both of them implement (IMAGE_WIDTH + 1) long array of registers).
5. Pass the outputs of the FIFO_COLUMN_L through a Haar LP and a Haar HP filter in parallel. Do the same for the outputs of FIFO_COLUMN_H component.
6. Pass the results of the decomposition to the Memory Controller which accumulates the results and stores them to output RAM.
7. Wait until the Memory Controller has set the ready signal to a high value, meaning that it processed its inputs.
8. If the end of the row is reached, send a reset signal to FIFO_ROW component. This is done in order to clear its contents, otherwise the last pixel of the previous line and the first pixel of the new line would get processed together.
9. If the end of the original image is reached, set the finished processing signal toa high value, and jump into the finished state.
10. Otherwise, fetch the next pixel (step 2).

The Control Unit is implemented in `controlunit.vhd` file.

3.3 Decimating the values after decomposition

Although the decimation components are given explicitly in the depictions of wavelet image decomposition (as can be found in the introduction section of the description of this laboratory exercise), in this implementation we apply the decimation implicitly. The decimation is performed by counting the processed rows and columns of the image. Since every other column and every other row of the decomposition need to be ignored, the control signals (row_ready and column_ready) are

passed to the Memory Controller, which either ignores the input data, or processes them. Only in case both of these signals are in the high state will the memory controller process the decomposition data.

3.4 Storing the data in output RAM

The data cannot be directly stored to the output RAM, because the compression yet needs to be performed. For this purpose, a memory controller device is created which takes the decomposition values and controls the address and data buses of the output RAM to store the correct values.

3.5 Compressing the image

The compression of the image is performed by thresholding or quantizing the values of the detail components of the image (y_{LH}, y_{HL} and y_{HH}). In this implementation, the thresholding approach was used. We choose a certain value as a threshold, and round all decomposition values that are lower than the threshold to zero.

Compression is conducted by counting the number of consecutive zero values in the detail components, and representing them by two 8-bit values:

1. Number of consecutive zero-values, and
2. The decomposition following the zero values.

This type of compression is performed on all three detail components of the image, while the approximation component (y_{LL}) is left uncompressed, and stored as-is.

The counting of the zero values (or in other words, values lower than a given threshold) is performed by a component called Memory Controller. This component implements another state machine whose task can be described as follows:

1. Initialize 4 stack pointers for storing the intermediate data: one (SP_LL) for the approximation component of the decomposition at address 0, and three for LH, HL and HH components of the decomposition, at addresses (1 * IMAGE_WIDTH*IMAGE_HEIGHT / 4), (2 * IMAGE_WIDTH*IMAGE_HEIGHT / 4) and (3 * IMAGE_WIDTH*IMAGE_HEIGHT / 4), respectively. Each stack is of size (IMAGE_WIDTH*IMAGE_HEIGHT / 4), which is the size of each of the decomposition components of the image.
2. If row_ready and column_ready are both high (decimation), jump to step 3.
3. Store the current y_{LL} at memory address pointed to by SP_LL. Increase SP_LL.
4. If y_{LH} is less than threshold, increase the count of zero values. Do the same for y_{HL} and y_{HH}. If the zero count is of size 255 (more than 255 consecutive zero-values exist), jump to step 5. Otherwise, jump to step 6.
5. If yLH is greater than or equal to the threshold value, store two bytes of data in the output RAM at addresses SP_LH and (SP_LH + 1): zero-count and the value of the non-zero pixel. Increase SP_LH. Do the same for HL and HH components of the image.
6. Send the ready signal back to the Control Unit.

7. If the Control Unit signalizes that it has finished processing all the data, store the remaining zero count to the RAM. Jump to step 9.
8. Jump to step 2.
9. If all data has been counted and written to RAM, move the data in stack space for components HL and HH to directly follow after the current value of SP_LH. This needs to be performed to achieve the actual memory compression and simplify the data streaming. When finished, Memory Controller will set the done signal to high.

The Memory Controller is implemented in `memcontroler.vhd` file.

4 Data transfer component - Transmitter

To access the data in the output RAM once the compression process is finished, a Transmitter component has been designed. When new data byte is requested from the memory, the request_new_data signal is driven high from the outside. The transmitter sets the current address on the address bus, and signalizes that the data is ready, by setting the out_data_ready signal to high. The outside reciever should set the request_new_data signal to low, process the data, and then drive it high again. The current address is implemented as a counter, which increases after each data byte has been sent.

Once all contents of the memory have been sent, the Transmitter sets the finished_transmitting signal to high, to signalize the reciever not to expect any further data.

The Transmitter component does not enable fetching of data until Memory Controller sets the done signal to high.

The Transmitter component is very useful to obtain the contents of the RAM after simulating the operation of the device. The testbench can fetch one byte at a time, and store it to an output file. This file can then later be analyzed and evaluated.

The Transmitter component in implemented in `transmitraw.vhd` file.

5 Tying it all together

The complete design of the device can be found in `top.vhd` file.

The block schematic of the entire design can be found in the Appendix.

6 Evaluating the results of the Wavelet Image Compression

To evaluate the operation of the designed device, a testbench has been written, and can be found in `tb_top.vhd` file. Other testbenches are also available in the same folder.

The tb_top testbench is a very simple simulation which performs only two things:

1. Updates the clock until the device signalizes that it has finished processing the input image (the done signal is driven high).
2. After done is set to high, request for RAM memory data by setting the request_new_data signal until finished_transmitting signal changes to high.

The contents of the RAM are written to a file on disk named `compressed.out`. To view the contents of this file, a Python script `plot_results.py` has been written. The script automatically loads the predefined file, decompresses the components of the image, and plots each component separately. It also returns the compression ratio of the image.

Appendix

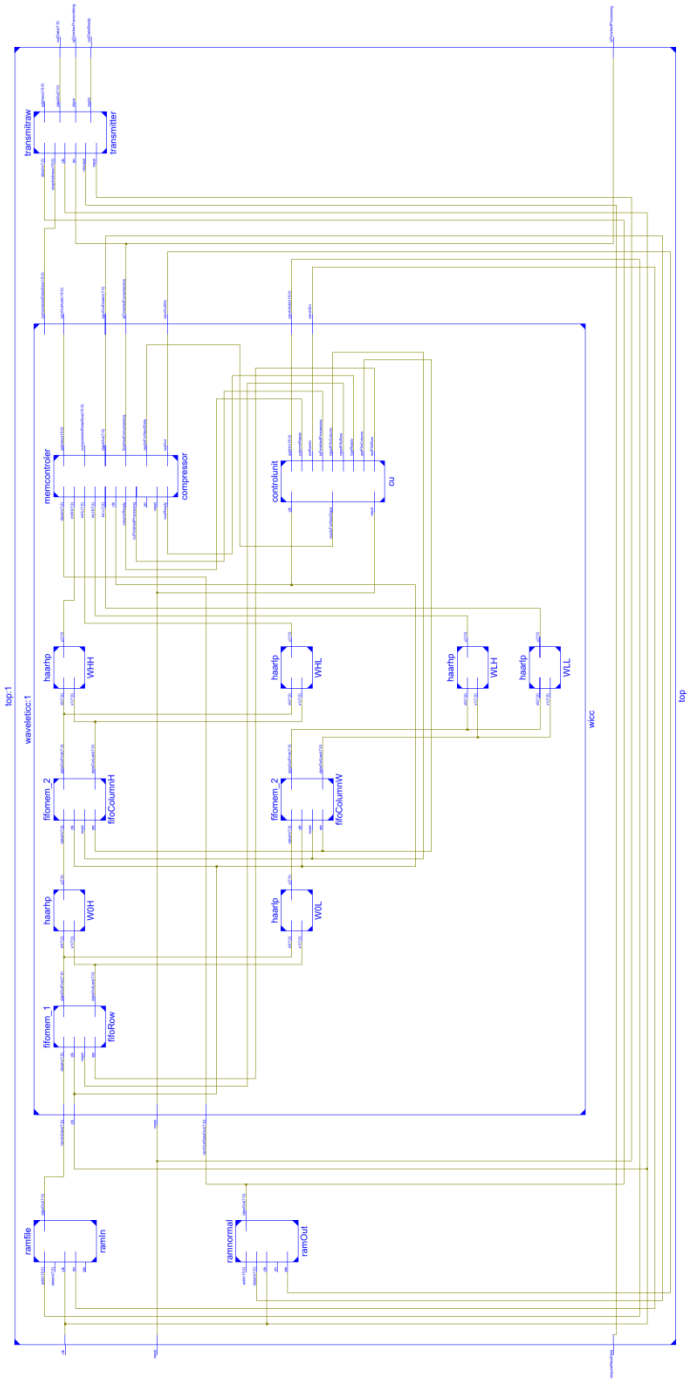

Figure 3. Schematic of the entire design.

Higher resolution schematics can be found in accompanying PDF files.