1 Introduction

In this lab we would need to design JPEG decoder. The JPEG encoding and decoding processes are used in image compression and aids greatly in image storage, media transfer etc. There are many steps that form the crux of JPEG compression. Figure 1 details the steps involved in the JPEG encode process.

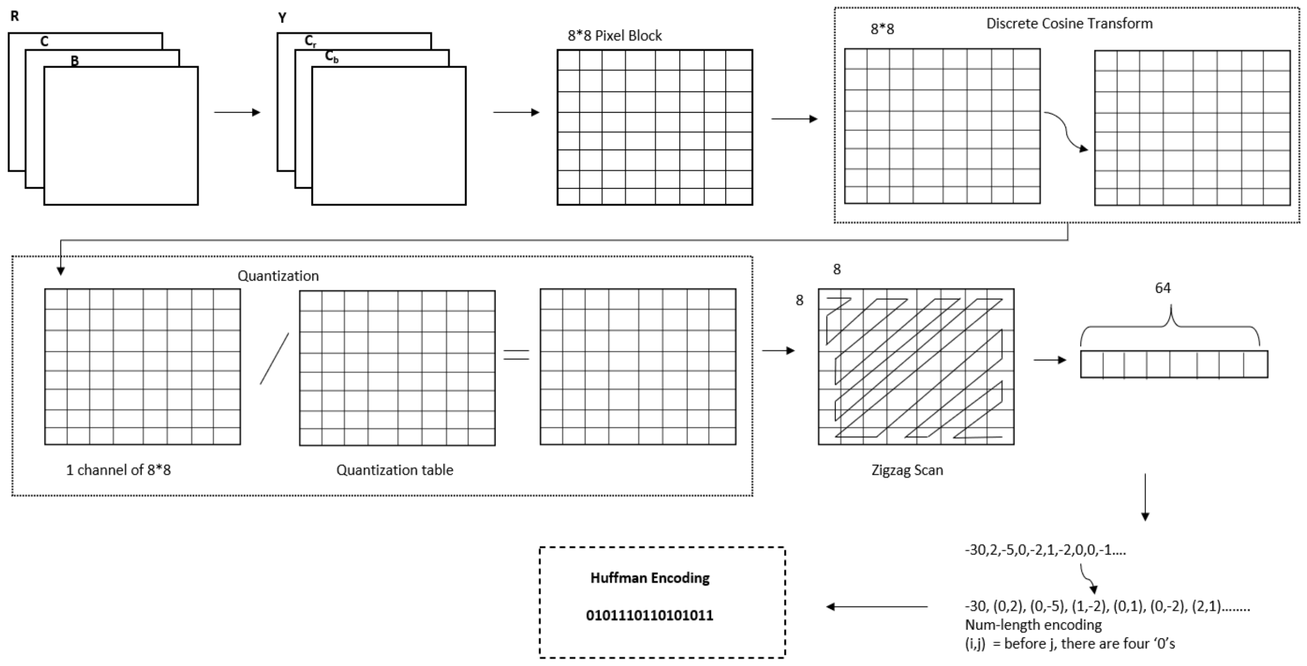

Figure 1: JPEG Encoding Process

For a more detailed understanding of the process and steps please refer to the following link: https://koushtav.me//jpeg/tutorial/2017/11/25/lets-write-a-simple-jpeg-library-part-1/#value-category-and-bitstream-table

JPEG decoding is the reverse process of encoding. In order to design a JPEG decoder, students would need to design Inverse Zigzag Transform and Huffman Decoding. All the other parts are done by the given software.

2 Lab Preparation

In this section, we need to implement the inverse zigzag modules in Vivado. Before you proceed, please download "**Lab5_student_code.zip**" from Piazza and **extract it**.
After extraction, you will get a folder named as "**Lab5_student_code/**".

a) Copy your project folder from lab 4 and rename it as "**lab5_vivado**". From the source panel, remove unnecessary source files from lab4.

b) Open the project by double-click on "**lab5_vivado/base/base.xpr**".

c) Add "**top_decoder.v**", "**izigzag.v**", "**huffmandecode.v** to the project from "**Lab5_student_code/**" and implement your design according to Section 1.

d) Please use 32-bit signed fixed-point number, with 16 bits for the fraction.

3 Lab Design on Inverse Zigzag Transform

For the original 8*8 block, each number in Figure 2 below represents the position in the zigzag block (the position is counted from the left to the right of each row, and from the upmost row to the bottom row). For example, the data at row 2 column 3 in the original block is put at the 8th position in the zigzag block (which is at row 1 column 8).

$$\begin{bmatrix} 1 & 2 & 6 & 7 & 15 & 16 & 28 & 29 \\ 3 & 5 & 8 & 14 & 17 & 27 & 30 & 43 \\ 4 & 9 & 13 & 18 & 26 & 31 & 42 & 44 \\ 10 & 12 & 19 & 25 & 32 & 41 & 45 & 54 \\ 11 & 20 & 24 & 33 & 40 & 46 & 53 & 55 \\ 21 & 23 & 34 & 39 & 47 & 52 & 56 & 61 \\ 22 & 35 & 38 & 48 & 51 & 57 & 60 & 62 \\ 36 & 37 & 49 & 50 & 58 & 59 & 63 & 64 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{bmatrix}$$

Figure 2: Before and After ZigZag Transformation

Figure 2 denotes the zigzag transform, and you are required to do INVERSE zigzag transform. In "**izigzag.v**", there are 64 data elements at the input port *zigzag*, with each data element having 32 bits. The first data is put at the lowest 32 bits of *zigzag*, and the last data is put at the highest 32 bits. You are required to write your Verilog code satisfying the following:

a) When *rst* is 0, *finish* is set to 0.

b) When *rst* is 1, the module starts calculating.

c) The rearranged data is finally put at the output port *outdata*. The first data is put at the lowest 32 bits of *outdata*, and the last data is assigned to the highest 32 bits.

d) When the *outdata* is ready, *finish* is set to 1, and both *outdata* and *finish* are supposed to hold their values.

4 Lab Design on Huffman Code Decoding

Huffman code is a renowned encoding algorithm and employed in coding. Usually, Huffman tree is used for encoding and decoding. However, the tree structure is not directly stored in JPEG file and instead, the Huffman table is stored to represent the Huffman coding information.

The Huffman table only contains how many codes there are, for each code length. For example, assume there are 6 Huffman codes: 00, 01, 100, 101, 110, 111, then, the Huffman table only records: 2 codes of length 2, 4 codes of length 3. However, given such a Huffman table, there is only 1 valid Huffman encoding according to it, and thus, only 1 Huffman tree that satisfies the Huffman table.

Besides Huffman table, JPEG file also records the symbols that each code represents. In the above example, since there are 6 codes, there are also 6 symbols. If they are "1", "12", "205", "36", "21"and "75", then, if the code is 1101001, the result should be "21", "205". (The rest of the code '1' is ignored because it can't be decoded as Huffman code)

In "**huffmandecode.v**", there is one 16-bit data to be decoded at the input port *code*. The Huffman decoding should decode from the highest bits of *code* and output the **first** decoded data to output *data*. There are 16 data elements at the input port *hufftable*, with each data consisting of 8 bits. The *nth* 8-bit data represents how many codes of length *n* are there. For example, if *hufftable[31:24]=5*, it means there are 5 codes of length 4. There are 256 data elements at the input port *huffsymbol*, with each data consisting of 8 bits. The *nth* 8-bit data represents the decoded data of the *nth* Huffman code.

You are required to write Verilog code such that:

a) When *rst* is 0, *finish* is set to 0.

b) When *rst* is 1, the module starts calculating.

c) The decode data is finally placed at the output port *data*. The length that decoded this data is put at output port *length*.

d) When the *data* and *length* are ready, *finish* is set to 1, and *data*, *length* and *finish* are required to hold their values.

The clock cycles needed to finish these 2 tasks are automatically counted and will be reported in the Linux terminal. There is no testbench requirement for this lab, but you are highly encouraged to write testbenches to test your design.

5 Implementation on the FPGA

In this section, we will implement the design on the FPGA.

a) Right click "**top_decoder.v**" in the "source" panel and click "set as top" (If this file is already in bold font, it is already the top module).

b) Now, "**top_decoder.v**" needs a dual port RAM. We add the RAM for "**top_decoder.v**".

c) On the left panel, click *IP catalog*, on the top right corner, search *"*ram*"*. Double click "block memory generator*"*.

d) If there is a window popped up, asking if you want to add IP to block design or customize IP, choose customize IP.

e) We need a two ports ram for "**top_decoder.v**". For memory type, select "**True Dual Port Ram**".

f) In both port A and port B options, change write width to **8**, and write depth to 65536.

g) Operation mode "**Read First**", enable port type **"Always Enabled".** Click *"ok"*.

h) In the pop-up window, select "Global" in the synthesis option, and click *"*generate*"*.

i) Now, click generate bitstream. After the bitstream is generated, click *file->export->export hardware*. Check include bitstream, click *"*ok*"*.

j) Report hardware utilization, power consumption, and critical path delay as in the previous lab.

k) Please launch SDK and generate the boot image (BOOT.bin) as in the previous lab with one exception:

Use the bitstream file base/base.sdk/top_decoder_hw_platform_0/top_decoder.bit.

l) Copy the updated **BOOT.bin, lab5_JPEG_test, testpic1.jpg and testpic2.jpg** into your SD card, boot the FPGA and run the test with the following command:

./lab5_JPEG_test [JPEG image file] [output file (bmp)] [other configurations]

m) The configurations are "-zigzag", and "-huffman". When you input one configuration, the task of it will be mapped to FPGA to be processed. So, if you only complete Huffman decoding, and you want to test the image testpic1.jpg, you could run the test as

```
./lab5_JPEG_test testpic1.jpg pic1.bmp -huffman
```

n) Finally, you should add these 2 configurations, and get the output bmp picture almost the same as the input picture (A little bit color degradation is allowed).

o) The results of the FPGA clock cycles are shown in the terminal, screenshot the output in the terminal.

p) Take a screen shot of the terminal when the result shows.

q) Unmount the SD card, exit the serial communication and turn off your FPGA.

Some commonly used commands:

```
mount /dev/mmcblk0p1 /mnt/
cd /mnt/
insmod transfpga.ko
mknod /dev/transfpga c 245 0
./ lab5_JPEG_test testpic1.jpg pic1.bmp -zigzag -huffman
cd /
umount /mnt/
```

6 Questions

a) What is the purpose for using zigzag transformation in JPEG encoding?

b) What is the purpose for using Huffman encoding in JPEG encoding?

You can find the answers from the lectures and the online tutorial given in section 1.

7 Pre-lab Submission

1. Please only submit 1 PDF file, containing your answers to the questions in the previous section.

2. Please name the PDF file "Lab#_PreLab_Section#_LastName_FirstName.pdf".
3. Please submit the PDF file on Canvas before February 28 (Monday) 11:59 pm.

8 Post-lab Submission

1. Please only submit 1 PDF file, containing the following items:
    a. Screenshots of the terminal after you run:
        i. **./lab5_JPEG_test testpic1.jpg zig_pic1.bmp -zigzag**
        ii. **./lab5_JPEG_test testpic2.jpg zig_pic2.bmp -zigzag**
        iii. **./lab5_JPEG_test testpic1.jpg zig_huff_pic1.bmp -zigzag -huffman**.
        iv. **./lab5_JPEG_test testpic2.jpg zig_huff_pic2.bmp -zigzag -huffman**.
    b. Screenshots of the four bmp pictures.
    c. A few words explaining the results.
    d. Screenshots of your code in this design.
2. Please name the PDF file as "Lab#_PostLab_Section#_LastName_FirstName.pdf".
3. Please submit the PDF file on Canvas before March 11 (Friday) 11:59 pm.