# JPEG Compression with FPGA Implementation

Yuxiang Chen - yc3096
Xinyi Chang - xc2323
Song Wang - sw2996
Nan Zhao - nz2250

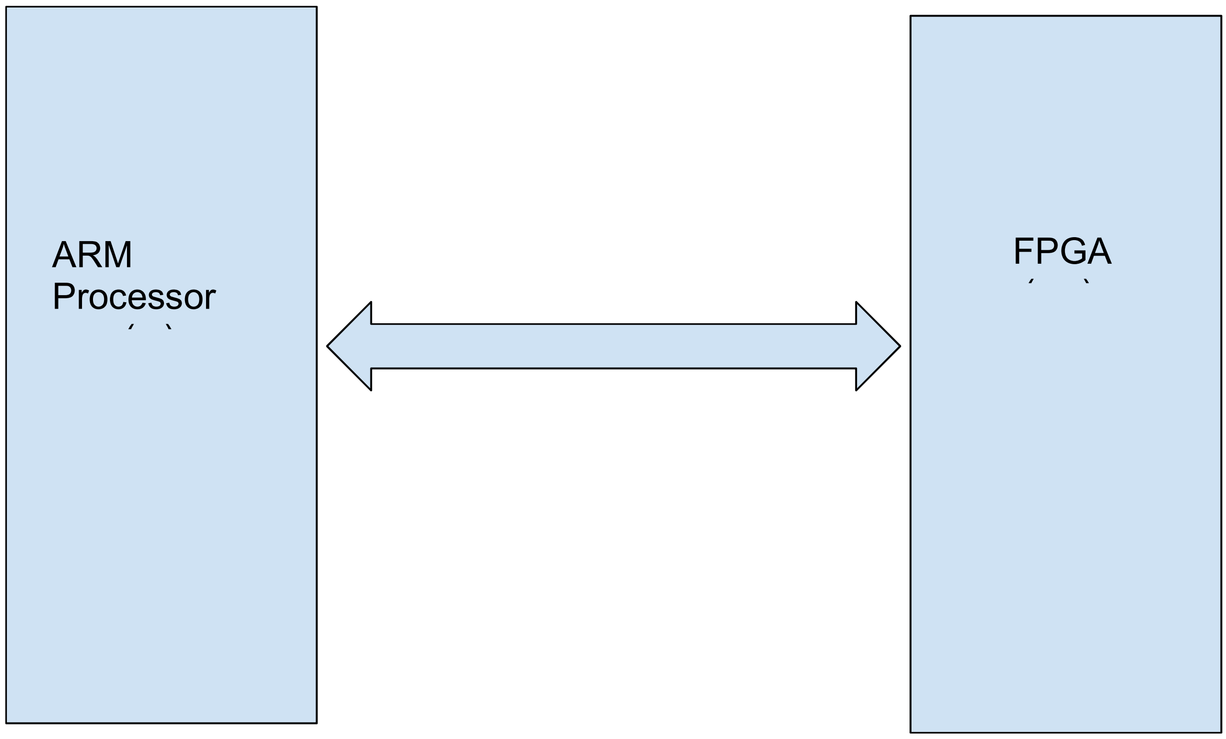
# Table of Contents

## 1. Overview

In recent years, the development and demand of multimedia product grows increasingly fast, contributing to insufficient bandwidth of network and storage of memory device. Therefore, the theory of data compression becomes more and more significant for reducing the data redundancy to save more hardware space and transmission bandwidth. In computer science and information theory, data compression or source coding is the process of encoding information using fewer bits or other information-bearing units than an unencoded representation. Compression is useful because it helps reduce the consumption of expensive resources such as hard disk space or transmission bandwidth. JPEG is the most commonly used method of lossy compression for digital photography. In this paper, we elaborate on the entire JPEG compression flow design and implementation in FPGA. We also provide the Matlab implementation which is used to justify the result of our hardware implementation.

## 2. Design and Implementation

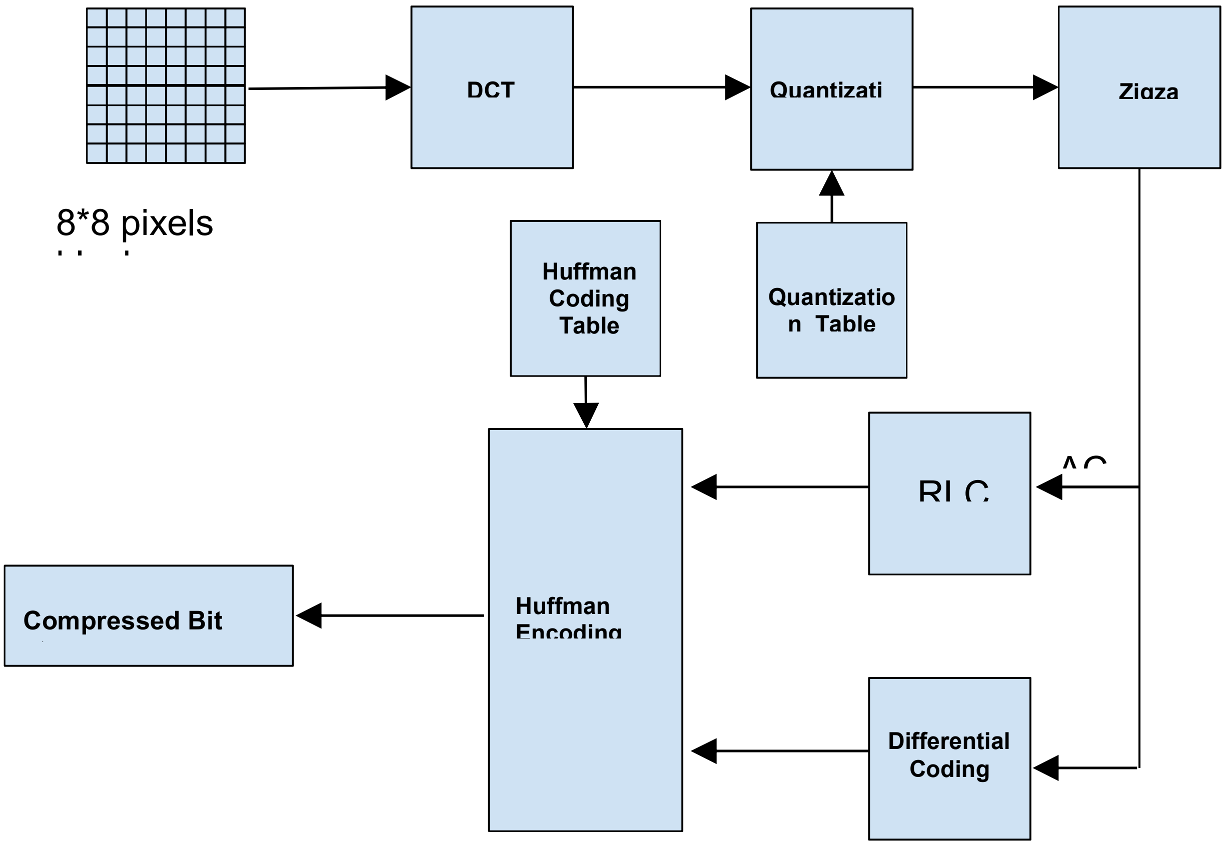
### 2.1. Architecture Overview



High frequency computation in FPGA communicates with ARM processor, data read and writes operates on the Avalon Bus port. The total design is based on socket board.

## 2.2. Hardware

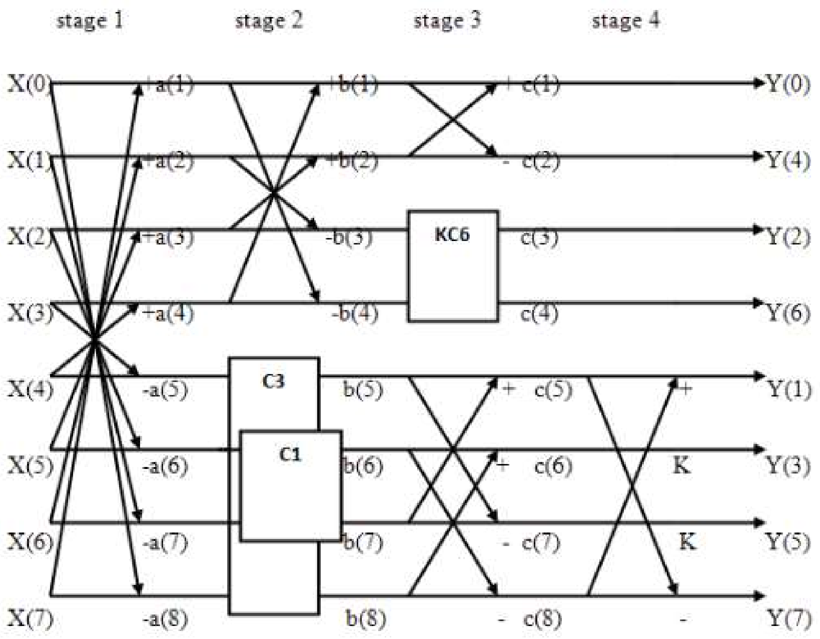
The hardware architecture is shown below, all blocks are coded in system verilog and compiled and synthesized with Quartus.



### 2.2.1. Algorithm Analysis and Implementation

#### 2.2.1.1 8-bit DCT Loeffler Architecture

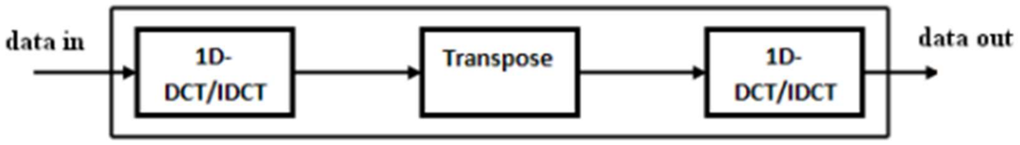
The Loeffler algorithm for the DCT is as shown below. It has four stages, each stage has to be executed in series due to the data dependency. As is seen in the figure, stage 1 requires 4 additions and 4 subtractions. In the second stage the algorithm is split into two parts, one of which for even coefficients and the other half for the odd coefficients. Again in the third stage the even coefficients are separated into even and odd parts. The scaling factor $k = \sqrt{2}$ .



To replace the multipliers with adders and shifters, we adopt CSE technique which enhances the usage of adders and shifters by identifying the common expressions. The CSD coefficients table is shown below.

| **constant** | **Fractional value** | **Binary value** | **Csd equivalent** |
|---|---|---|---|
| cos(6π/16 ) | 0.38268 | 00110001 | 0+0-000+ |
| sin(6π/16) | 0.92388 | 01110110 | +000-0-0 |
| cos(3π/16 ) | 0.83147 | 01101010 | +0-0+0+0 |
| sin(3π/16) | 0.55557 | 01000111 | 0+00+00- |
| cos(π/16) | 0.98079 | 01111110 | +00000-0 |
| sin(π/16) | 0.19509 | 00011001 | 00+0-00+ |

The 2D DCT is implemented by using two 1d DCT blocks and a transposition block.



While implementing this part, we use a 64*20 memory to store the output of 1d DCT and load the data out from the memory in a transposed order and feed each data group into 1d DCT block again to complete the 2d DCT computation.

Besides, we use the shifter to replace the divider used to compute coefficients, which keeps the consistency of no divider and multiplier within the DCT block, even though the output may be slightly different from the Matlab result.

Lo-effler is able to reduce the cost to only 11 multiplications and 29 additions.

### 2.2.1.2 Quantization and Zigzag

The reason we perform quantization is because that we want to discard some image information which is not critical for visual experience. By lowering the amount of information that has been stored in each pixel, we are able to further compress the image.

The goal of quantization is to reduce most of the less important high frequency DCT coefficients to zero, the more zeros we generate the better the image will compress. To implement quantization by dividing each coefficient with the corresponding value in normalization matrix in table 1.

Table 1: A typical normalization matrix

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 16 | 11 | 10 | 16 | 24 | 40 | 51 | 61 |
| 12 | 12 | 14 | 19 | 26 | 58 | 60 | 55 |
| 14 | 13 | 16 | 24 | 40 | 57 | 69 | 56 |
| 14 | 17 | 22 | 29 | 51 | 87 | 80 | 62 |
| 18 | 22 | 37 | 56 | 68 | 109 | 103 | 77 |
| 24 | 35 | 55 | 64 | 81 | 104 | 113 | 92 |
| 49 | 64 | 78 | 87 | 103 | 121 | 120 | 101 |
| 72 | 92 | 95 | 98 | 112 | 100 | 103 | 99 |

However, the output we get after we use this normalized table is not correct, the reason is probably that it is too complicated for tool to synthesize divider. So we use another modified normalization table for hardware simplification.

Table 3: A modified normalization matrix for hardware simplification

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 16 | 16 | 16 | 16 | 32 | 64 | 64 | 64 |
| 16 | 16 | 16 | 16 | 32 | 64 | 64 | 64 |
| 16 | 16 | 16 | 32 | 32 | 64 | 64 | 64 |
| 16 | 16 | 32 | 32 | 32 | 64 | 64 | 64 |
| 32 | 32 | 32 | 64 | 128 | 128 | 128 | 128 |
| 64 | 64 | 64 | 64 | 128 | 128 | 128 | 128 |
| 128 | 128 | 128 | 128 | 128 | 128 | 128 | 128 |
| 128 | 128 | 128 | 128 | 128 | 128 | 128 | 128 |

By using this table, only shifter is required, and the compression quality is almost the same.

After quantization, we perform a zigzag scan to get DC and AC values. DC value is treated as the average value of the original 64 image samples. AC values are the read consequently after reading DC value. The order we read the DCT values after quantization is shown in the zigzag order table.

Table 2: Zig-zag order sequence

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 5 | 6 | 14 | 15 | 27 | 28 |
| 2 | 4 | 7 | 13 | 16 | 26 | 29 | 42 |
| 3 | 8 | 12 | 17 | 25 | 30 | 41 | 43 |
| 9 | 11 | 18 | 24 | 31 | 40 | 44 | 53 |
| 10 | 19 | 23 | 32 | 39 | 45 | 52 | 54 |
| 20 | 22 | 33 | 38 | 46 | 51 | 55 | 60 |
| 21 | 34 | 37 | 47 | 50 | 56 | 59 | 61 |
| 35 | 36 | 48 | 49 | 57 | 58 | 62 | 63 |

Due to the data dependency, we can't reuse the buffer which is used to store the output after the first round DCT computation to store the output data after the second round DCT computation. To maintain the integrality of data and reduce the complexity of coding, we use another buffer to store the data of the second round DCT and then load the data in a zigzag order.

#### 2.2.1.3 Zero-Run-Length Coding

After running quantization and zigzag scan, we reorder the data structure of the 1-D vector coming from zigzag scan. Then we way reorder the data is to apply zero-run-length coding.
RLC is based on the following two observations:

1. Consecutively occurring zeros in the input data stream is very common.
2. Non-zero digits with same value occurring consecutively is a very rare care case.

Using this coding adds the redundancy in the encoded data for a series of consecutive non-zero numbers, when it was meant to compress the original data.

In this method values are presented by pair of number. The first number is the information about number of preceding zeros, the second is the non-zero value. There are two special codes: eob (0,0), which represents tailing zeros and zrl (15,0), which represents 16 subsequent zeroes (maximum allowed number of subsequent zeros). In addition, we use size and actual value to represent the corresponding non-zero data.
The data layout we get after rlc is shown below.

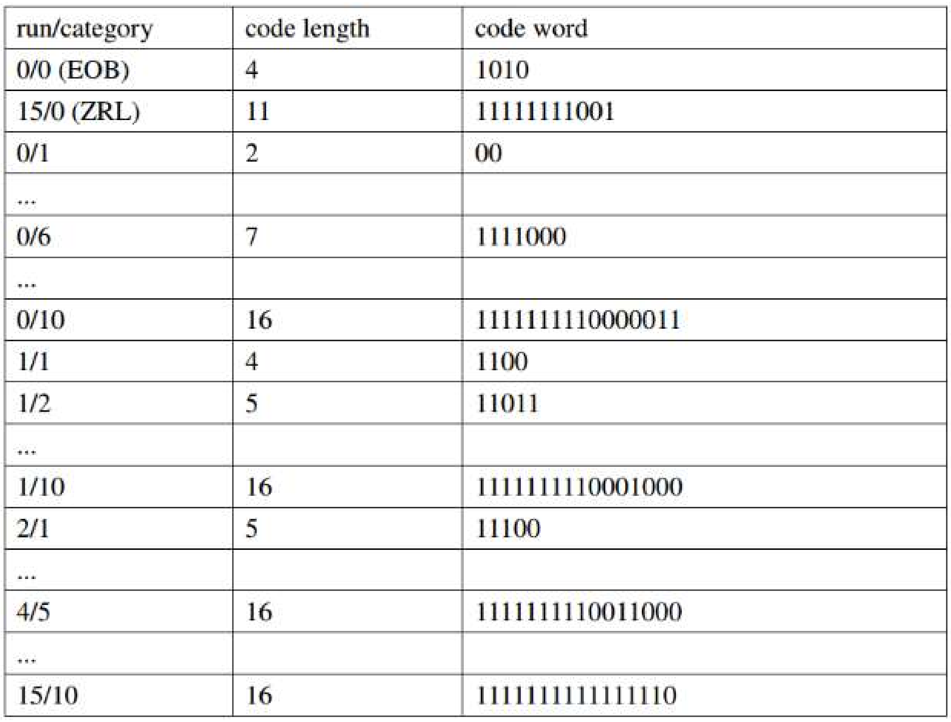
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Run | Category | Bit Value | ...... | Run | Category | Bit Value | EOB |

### 2.2.1.4 Huffman Encoding

After we get the data after RLC, the next step is Huffman coding.

Huffman coding is a method that takes symbols (e.g. bytes, DCT coefficients, etc.) and encodes them with variable length codes that are assigned according to statistical probabilities. A frequently-used symbol will be encoded with a code that takes up only a couple bits, while symbols that are rarely used are represented by symbols that take more bits to encode.

A JPEG file contains up to 4 Huffman tables that define the mapping between these variable-length codes (which take between 1 and 16 bits) and the code values (which is an 8-bit byte). Creating these tables generally involves counting how frequently each symbol (DCT code word) appears in an image, and allocating the bit strings accordingly. But, most JPEG encoders simply use the Huffman tables presented in the JPEG standard. Some encoders allow one to optimize these tables, which means that an optimal binary tree is created which allows a more efficient Huffman table to be generated. The Huffman table for AC coefficients is shown below.



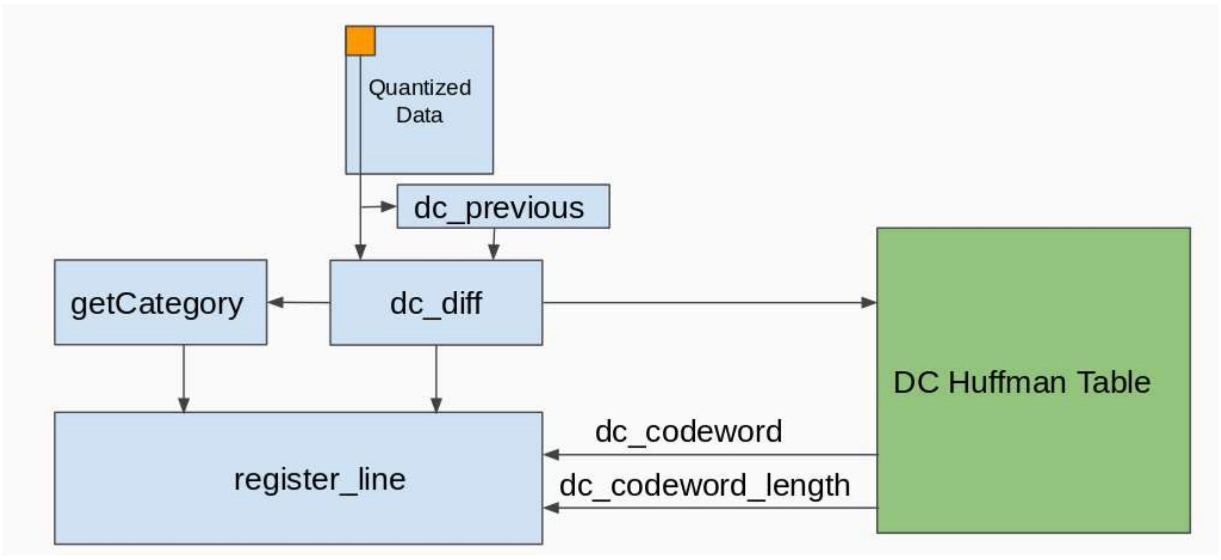
| run/category | code length | code word |
|---|---|---|
| 0/0 (EOB) | 4 | 1010 |
| 15/0 (ZRL) | 11 | 11111111001 |
| 0/1 | 2 | 00 |
| ... | | |
| 0/6 | 7 | 1111000 |
| ... | | |
| 0/10 | 16 | 1111111110000011 |
| 1/1 | 4 | 1100 |
| 1/2 | 5 | 11011 |
| ... | | |
| 1/10 | 16 | 1111111110001000 |
| 2/1 | 5 | 11100 |
| ... | | |
| 4/5 | 16 | 1111111110011000 |
| ... | | |
| 15/10 | 16 | 1111111111111110 |

To implement Huffman coding in Systemverilog, we use the following algorithm to compute dc coding.

- dc_diff = dc_current -dc_previous
- dc_diff_length = getCategory(dc_diff)

- dc_codeword = dc_lookup_table(dc_diff)
- register_line = register_line + (ac_codeword << category) + dc_diff

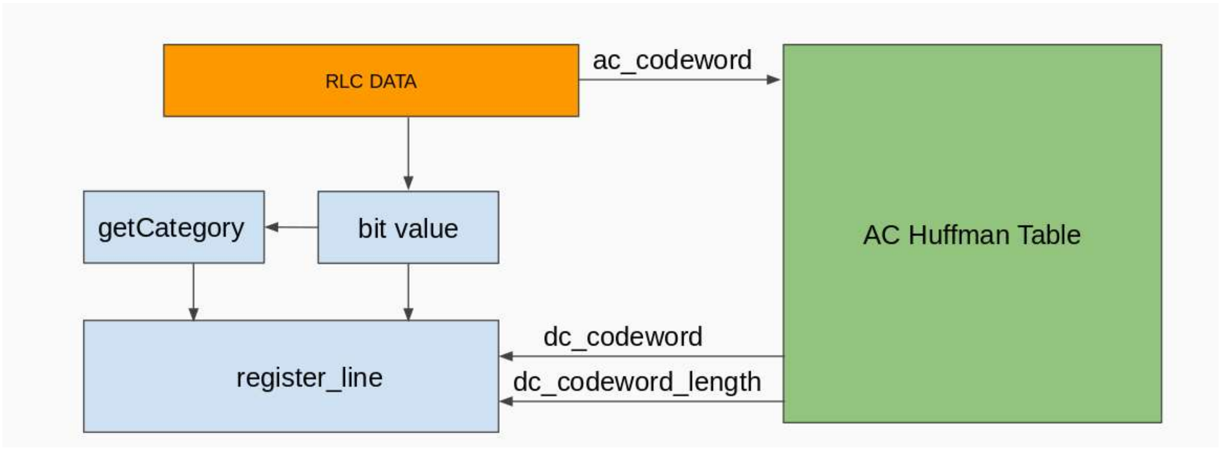
And the block view about how to implement this is shown below.
For the dc Huffman coding:



For the ac part, we use the algorithm shown below:

- ac_diff_length = getCategory(bit_value)
- ac_codeword = ac_lookup_table()
- register_line = register_line + (ac_codeword << category) + bit_value

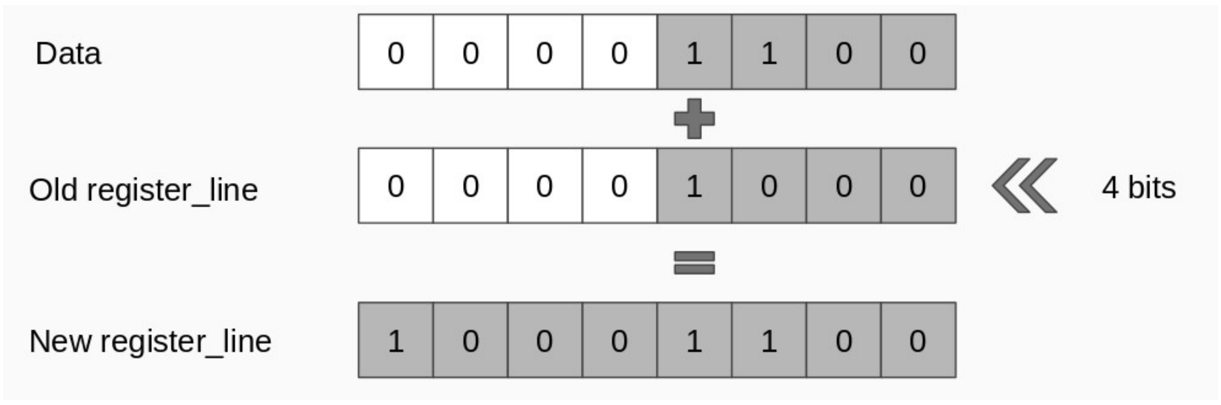
And the block view about how to implement this is shown below.



Besides, we need to adopt the following operation to integrate the codeword.

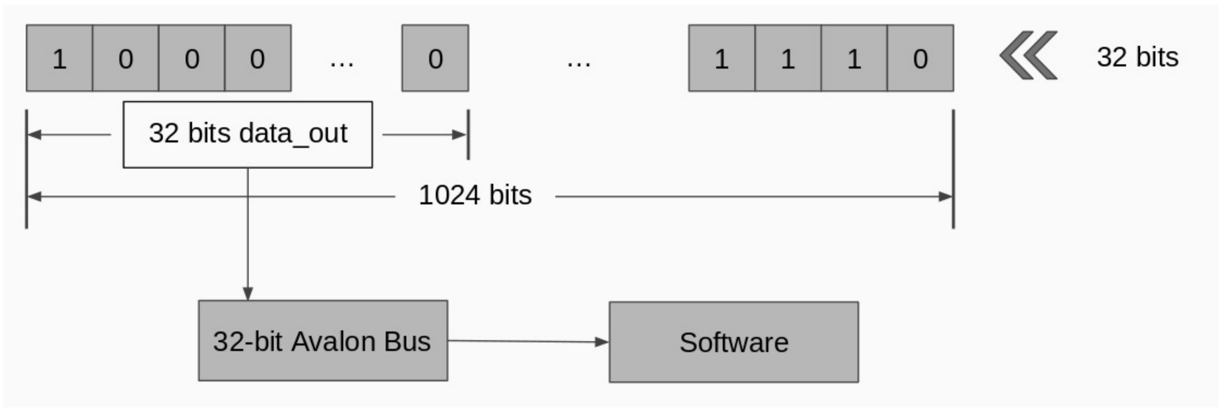
- Initialized a 1024-bit length register_line,
- While (there is data):

- register_line = (register_line << data_length) + data;
- total_line_size += data_length



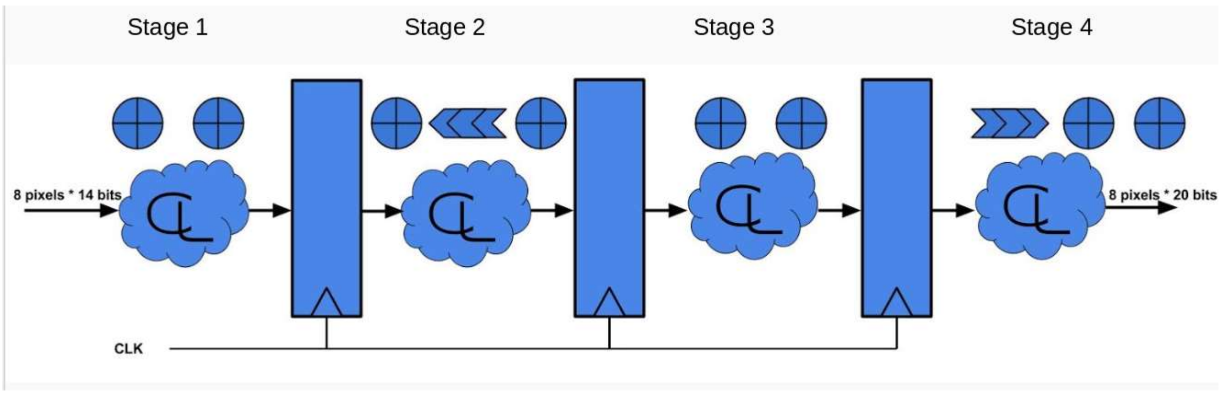
After we get all the compressed bits from Huffman coding, we need to integrate the separated code words into a single array in order to send it back to software domain. The algorithm is shown below:

- register_line << register_length,
- do:
    - data_back = register_line[1023:991]
    - Register_line << 32 bits
- while(data != 0 )
-



### 2.2.2. Pipeline Design

To achieve the largest throughput we adopt pipeline architecture. We divide the 1-d dct computation unit into four pipeline stages as seen in the figure below.



## 2.3. Software

We write c file to receive data from FPGA and print out the final output for comparing with that of Matlab.

## 2.4. Hardware and Software Interface

### 2.4.1. Device Driver

Our design uses a very similar driver comparing to Lab3. Since our system requires to receive 8*8 DCT data block from computer, it's necessary for us to create hardware-software interface to connect image input module (from computer) and hardware module (FPGA). We wrote a driver code to enable computer's ability to transport data while the FPGA is running. This is established by using sample code from Lab 3 and modify certain module/variable names/functions. Please see appendix for our device driver.

### 2.4.2. Avalon Bus

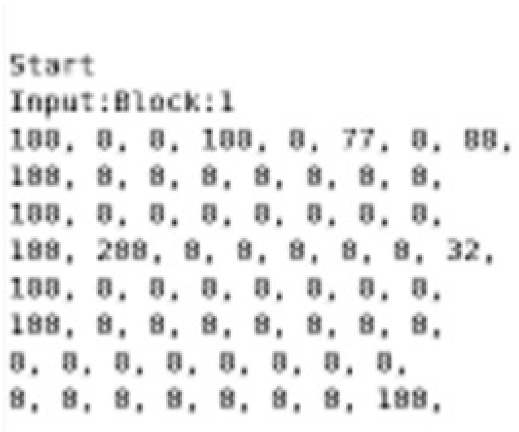
Avalon Bus is a communication terminal between peripherals and ARM core processor. This communication terminal is an asynchronous terminal which requires logically implemented I/O protocol to establish I/O communications. The peripherals for ARM processor is like a black box to developers. Once data goes into the FPGA, we will not be able to check whether the data is correct until FPGA outputs data. Enabling avalon bus simulator from Qsys makes it a lot easier for developers in terms of debugging FPGA circuit. Since Avalon Bus follows certain input and output protocols, we need to emulate this protocol as well.

The port bandwidth is 32 bits for read operation. We need to rearrange the data layout in hardware domain and divide the data array into 32-bit packet as the data used to be sent back to software domain.

# 3. Functional Verification

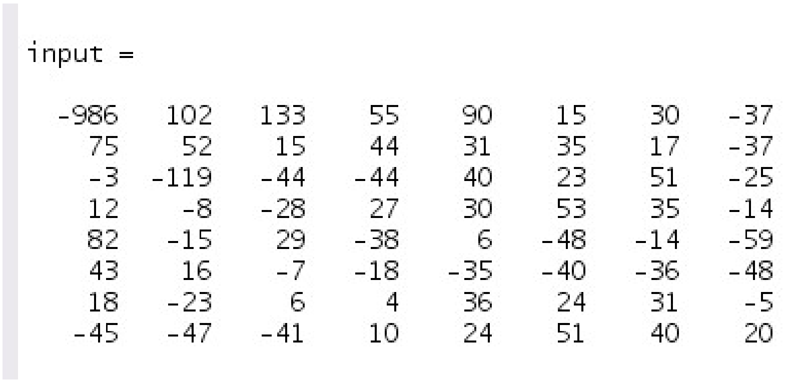
To test the functionality of our project, we test our FPGA module using a sample matrix. We run Matlab DCT simulation and FPGA DCT using the same sample matrix and the outcome shows consistency. Here are the results:

1. DCT Input



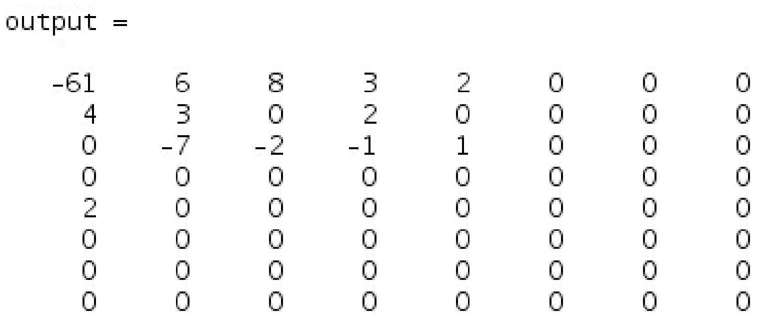
```
Start
Input:Block:1
100, 0, 0, 100, 0, 77, 0, 88,
100, 0, 0, 0, 0, 0, 0, 0,
100, 0, 0, 0, 0, 0, 0, 0,
100, 200, 0, 0, 0, 0, 0, 32,
100, 0, 0, 0, 0, 0, 0, 0,
100, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 100,
```

2. DCT Output



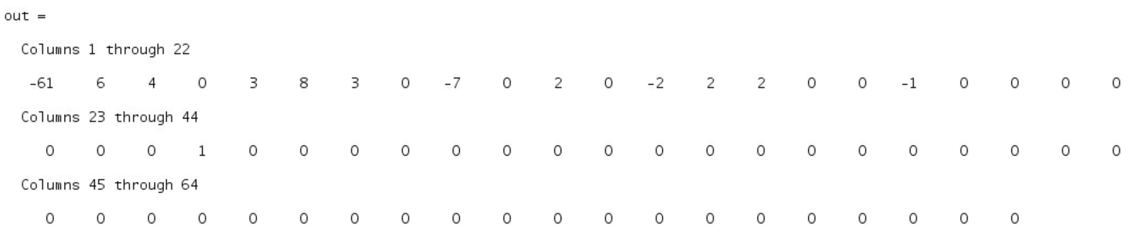
```
input =

  -986   102   133    55    90    15    30   -37
    75    52    15    44    31    35    17   -37
    -3  -119   -44   -44    40    23    51   -25
    12    -8   -28    27    30    53    35   -14
    82   -15    29   -38     6   -48   -14   -59
    43    16    -7   -18   -35   -40   -36   -48
    18   -23     6     4    36    24    31    -5
   -45   -47   -41    10    24    51    40    20
```

3. Quantization Output



```
output =

   -61     6     8     3     2     0     0     0
     4     3     0     2     0     0     0     0
     0    -7    -2    -1     1     0     0     0
     0     0     0     0     0     0     0     0
     2     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

4. Zigzag output

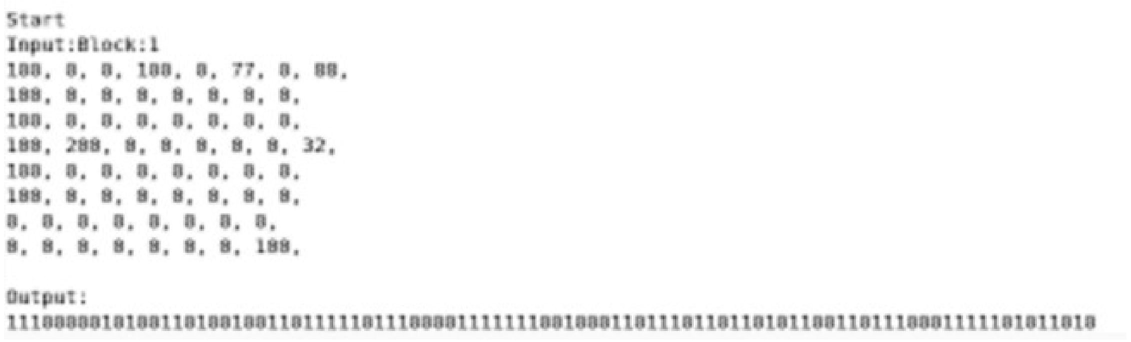


```
out =

  Columns 1 through 22

   -61     6     4     0     3     8     3     0    -7     0     2     0    -2     2     2     0     0    -1     0     0     0     0

  Columns 23 through 44

     0     0     0     1     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0

  Columns 45 through 64

     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
```

## 5. RLC Output

| | |
|---|---|
| DC: -61 -> (14) -> | 111000001 |
| AC: (0, 6)->(0,3,6)->(100,110)->38 | 100110 |
| (0, 4)->(0,3,4)->(100,100)->36 | 100100 |
| (1, 3)->(1,2,3)->(11011,11)->111 | 1101111 |
| (0, 8)->(0,4,8)->(1011,1000)->184 | 10111000 |
| (0, 3)->(0,2,3)->(01,11)->7 | 0111 |
| (1,-7)->(1,3,000)->(1111001,000)->968 | 11110010000 |
| (1,2)->(1,2,2)->(11011,10)->110 | 1101110 |
| (1,-2)->(1,2,2)->(11011,01)->109 | 1101101 |
| (0,2)->(0,2,2)->(01,10)->6 | 0110 |
| (0,2)->(0,2,2)->(01,10)->6 | 0110 |
| (2,-1)->(2,1,0)->(11100,0)->56 | 111000 |
| (7,1)->(7,1,1)->(11111010,1)->501 | 111110101 |
| (0,0)->(1010)->10 | 1010 |

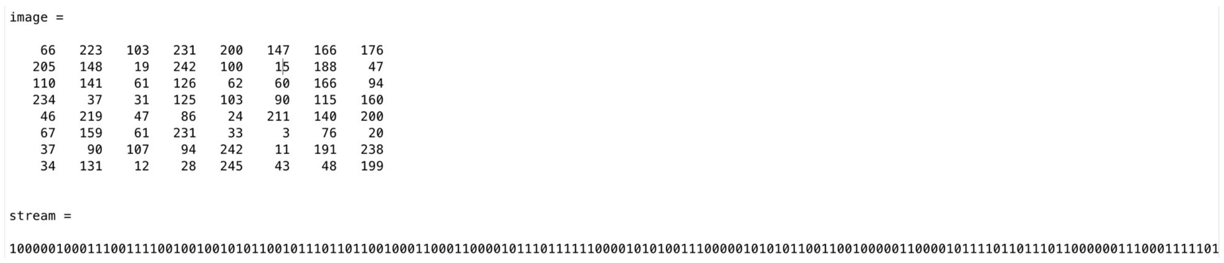
## 6. Bitstream Output



```
Start
Input:Block:1
100, 0, 0, 100, 0, 77, 0, 88,
100, 0, 0, 0, 0, 0, 0, 0,
100, 0, 0, 0, 0, 0, 0, 0,
100, 200, 0, 0, 0, 0, 0, 32,
100, 0, 0, 0, 0, 0, 0, 0,
100, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 100,

Output:
11100000101001101001001101111101110000111111100100011011101101101011001101110001111101011010
```

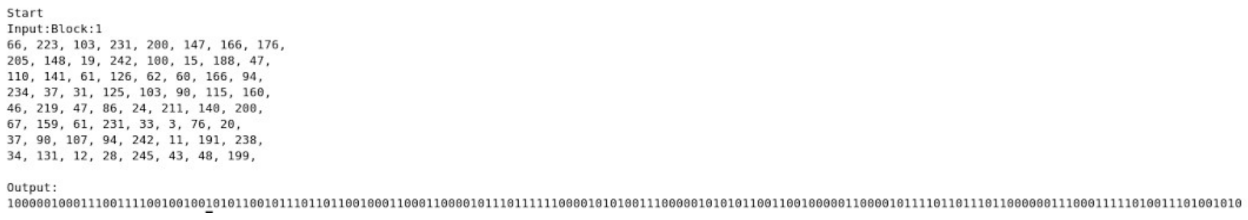
We build the Matlab model for the whole process of JPEG compression and compare the bitstream output from FPGA with the bitstream output from Matlab.

Below is another test we run on both platforms.



```
image =

    66   223   103   231   200   147   166   176
   205   148    19   242   100    15   188    47
   110   141    61   126    62    60   166    94
   234    37    31   125   103    90   115   160
    46   219    47    86    24   211   140   200
    67   159    61   231    33     3    76    20
    37    90   107    94   242    11   191   238
    34   131    12    28   245    43    48   199


stream =

100000100011100111100100100101011001011101101100100011000110000101110111111000010101001110000010101011001100100000110000101111011011101100000011100011111101
```

Output from FPGA



```
Start
Input:Block:1
66, 223, 103, 231, 200, 147, 166, 176,
205, 148, 19, 242, 100, 15, 188, 47,
110, 141, 61, 126, 62, 60, 166, 94,
234, 37, 31, 125, 103, 90, 115, 160,
46, 219, 47, 86, 24, 211, 140, 200,
67, 159, 61, 231, 33, 3, 76, 20,
37, 90, 107, 94, 242, 11, 191, 238,
34, 131, 12, 28, 245, 43, 48, 199,

Output:
1000001000111001111001001001010110010111011011001000110001100001011101111111000010101001110000001010101100110010000001100001011110110111011000000011100011111010011101001010
```

## 4. Issues and Challenges

### 4.1. Hardware Domain

1. DCT and subsequent functional unit are too complicated, the FSM needed to control the data path is hard design and requires all-sided consideration.
2. Due to large size of data, the utilization of multiplier, divider and even shifter is heavily restricted.

### 4.2. Software Domain

At the starting stage of the project, we planned to first write a C code simulation before we actually write or implement any algorithm related Verilog code. We tried to simulate the whole process starting from reading image, apply 2D DCT, quantization, hHffman encoding and etc. However, after spending a quite good amount of time in terms of looking for libraries, it turns out that the effort we spent has little effect in helping the project.

1. The simulation of whole transcode process can solely done in Matlab
2. The difficulty in translating Matlab language to Systemverilog language is equivalent to translating c language to Systemverilog language
3. Using C language to perform simulation tends to encounter a lot of memory access issue. We should spend more time in developing SystemVerilog instead of debugging the simulator when Matlab can do a great job as simulator

## 5. Conclusion and Prospective Optimization

The whole compression flow including dct2, quantization, zigzag, rlc, and Huffman encoding was implemented in FPGA, and we transmitted the data after compression back to ARM processor. The data flow in the FPGA is much faster than image compression processing in the ARM processor. Thus we get a lot of speedup by separating the compression process from the processor to the FPGA. The quality of the image we get in the end is pretty good as compared with the original one.

Of course, there are some points which can be further optimized. We didn't take full advantage of the pipeline. In order to maintain the integrality of pixel data for each image block and make each block independent from each other, there are some bubbles in the DCT pipeline. Digging into the sequential data flow further, we can probably improve the throughput greater.

Another place where we can do further optimization is the memory. We may have not use the memory provided by the FPGA due to the inefficient Systemverilog coding.

## 6. Contribution

Xinyi Chang - JPEG Matlab, SW in C, JPEG Algorithm Finding.
Yuxiang Chen - SW in C, DCT SystemVerilog, RLC/Huffman SystemVerilog
Song Wang - DCT Matlab, Quantization/Zigzag SystemVerilog, JPEG Algorithm Finding.
Nan Zhao - SW Image Reading, Huffman simulation, Document Preparation.

## 7. Reference

[1] M. Jridi, A. Alfalou, "A low-power, high-speed DCT architecture for image compression: Principle and implementation," 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC), 2010, pp. 304-309.

[2] Y. H. Chen , T. Y. Chang and C. Y. Li , "Highthroughput DA-based DCT with high accuracy error-compensated adder tree" ,  IEEE Trans. VeryLarge Scale Integr. (VLSI) Syst. , vol. 19 , no. 4 , pp.709 -714 , 2011

[3] V. Gupta, D. Mohapatra, A. Raghunathan and K. Roy , "Low-power digital signal processing using approximate adders" ,  IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. , vol. 32 , no. 1 , pp.124 -137 , 2013

[4] P. Kulkarni, P. Gupta, and M. D. Ercegovac, "Trading accuracy for power in a multiplier architecture," J. Low Power Electron., vol. 7, no. 4, pp.490–501, 2011.

[5] Y. V. Ivanov and C. J. Bleakley, "Real-time h.264 video encoding in software with fast mode decision and dynamic complexity control," ACM Trans. Multimedia Comput. Commun. Applicat., vol. 6, pp. 5:1–5:21, Feb. 2010.

[6] Avalon Interface Specifications, Altera

[7] Preethi, D., and A.M Vijaya Prakash. "A Low Power VLSI Architecture for Image Compression System Using DCT and IDCT." *International Journal of Engineering and*

*Advanced Technology (IJEAT)* 2249 – 8958 1.5 (2012): 0-5. Web. 5 May 2016. <http://www.ijeat.org/attachments/File/v1i5/E0544061512.pdf>.

[8] Wei, Weiyi. "An Introduction to Image Compression." (2009): 1-29. Web. 5 May 2016. <http://disp.ee.ntu.edu.tw/meeting/維毅/An Introduction to Image Compression/An introduction to Image Compression.pdf>.

# 8. Appendix

## 8.1 Software Code

### 8.1.1 VGA_LET.c

```
/*
 * Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *             drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_led.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_led.c
 */

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

#define DRIVER_NAME "vga_led"

/*
 * Information about our device
 */
struct vga_led_dev {
        struct resource res; /* Resource: our registers */
```

```
		void __iomem *virtbase; /* Where registers can be accessed in memory */
		u8 segments[VGA_LED_DIGITS];
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */

static void write_32(unsigned long number)
{
		iowrite32(number, dev.virtbase);
}



/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static unsigned long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{


		switch (cmd) {
  case DCT_WRITE32_DIGIT:
				write_32(arg);
				break;
		case DCT_READ_DIGIT:
				//return read_32();
				return ioread32(dev.virtbase);
				break;
		default:
				return -EINVAL;
		}

		return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_led_fops = {
		.owner			= THIS_MODULE,
		.unlocked_ioctl = vga_led_ioctl,
};

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

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

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

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

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

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

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&vga_led_misc_device);
	return ret;
}

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

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

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

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

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

```
module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

### 8.1.2 VGA_LED.c

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 8

typedef struct {
  unsigned char digit;    /* 0, 1, .. , VGA_LED_DIGITS - 1 */
  unsigned char segments; /* LSB is segment a, MSB is decimal point
*/
} vga_led_arg_t;



#define DCT_MAGIC 'q'

/* ioctls and their arguments */
#define DCT_WRITE32_DIGIT  _IOW(DCT_MAGIC, 3, long)
#define DCT_READ_DIGIT  _IOWR(DCT_MAGIC, 2, long)
#endif
```

### 8.1.3 JPEG.c

```
#include <stdio.h>
#include "vga_led.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

int vga_led_fd;

unsigned long read_data() {
        unsigned long number;
        number = ioctl(vga_led_fd, DCT_READ_DIGIT, 0);
        return number;
}

void write_data(unsigned long number)
{
        ioctl(vga_led_fd, DCT_WRITE32_DIGIT, number);
}
```

```
int main(){
               static const char filename[] = "/dev/vga_led";
               if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
                       fprintf(stderr, "could not open %s\n", filename);
                       return -1;
               }
               int i = 0;
               int numBlocks = 1;


               int num;
               int input[64];
               FILE *file             = fopen("rand.txt"     ,"r");
               FILE *file_out         = fopen("output_rand.txt","w");

               int input_i = 0;
               int blocks = 1;
               while(blocks <= numBlocks){
                                      input_i = 0;
                                      while (input_i < 64){
                                              fscanf(file, "%d", &num);
                                              input[input_i] = num;
                                              input_i += 1;
                                      }
                                      unsigned long message = 0x00000064;
                                      printf("\n\nStart\n");
                                  printf("Input:");
                                      printf("Block:%d", blocks);
                                      int index;
                                      for (i = 0; i < 16; i++){
                                                      index = i * 4;
                                                      message = input[index] +
256*input[index+1] + 65536*input[index+2] + 16777216*input[index+3];
                                                      if (i % 2 == 0){
                                                              printf("\n");
                                                      }
                                                      printf("%d, %d, %d, %d, ",
input[index], input[index+1], input[index+2], input[index+3]);
                                              write_data(message);
                                      }
                                  unsigned long num;
                                      int g,j;
                                      char zero[6] = "001010";
                                  printf("\n\nOutput:");
                                      for (g = 0; g < 64; g++){
                                                      if (g % 8 == 0){
                                                              printf("\n");
                                                      }
                                                      num = read_data();
                                                      if (num == 0){
                                                      break;
                                                      }
                                                  //fprintf(file_out, "%lu,", num);
                                                      //printf(" ,%lu, ", num);
                                                  int category = 0;
                                                  unsigned long temp[32];
                                                  unsigned long decimalNum = num;
                                                  long bitCodedValue[32] = {0};
                                                  while(decimalNum > 0){
                                                      temp[category] = decimalNum % 2;
                                                      category++;
                                                      decimalNum = decimalNum/2;
                                                  }
                                                  int start =  32 - category;
                                                  for(i = start; i < 32; i ++){
                                                      bitCodedValue[i]= temp[31-i];
                                                  }

                                                  int last = 32;
                                                  int add = 0;
                                                  for(i = 31; i > 3; i --){
                                                      if(bitCodedValue[i] == 0 &&
bitCodedValue[i-1] == 1 && bitCodedValue[i-2] == 0 && bitCodedValue[i-3] == 1) {
                                                          for (j = i; j < 32; j++) {
                                                             add = add + bitCodedValue[j];
                                                          }
                                                          if(add == 0) {
                                                              last = i+1;
                                                          }
                                                      }
                                                  }

                                                  for(i = 0; i <last; i++) {
                                                      printf("%ld", bitCodedValue[i]);
                                                             fprintf(file_out, "%ld",
bitCodedValue[i]);
                                                  }
                                                  //fprintf(file_out, "\n");
                                                     //printf("\n");
                                      }
                                      //
                                      blocks = blocks + 1;
               }
               printf("\nEnd");
               fclose(file);
               fclose(file_out);
               return 0;
}
```

## 8.2 Hardware Code

### 8.2.1 VGA_LED.sv

```
/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 * Stephen A. Edwards
 * Columbia University
 */

module VGA_LED(  input       logic              clk,
                             input     logic                              reset,
                             input     logic     [31:0]     writedata,
                             input     logic                              write,
                                input     logic                                       read,
                             input                                                      chipselect,
                             input     logic     [2:0]      address,
                                output    logic      [31:0]     readdata);

                                          logic      signed     [31:0]        output_data;
                                          logic                              [5:0]
        addr_write;
                                          logic                              [5:0]
        addr_read;
                                          logic                              [5:0]
        addr_dct_input;
                                          logic                              [4:0]                   stages;

                                          logic
        read_before;
```

```
                                        logic
     input_loaded;
                                        logic
     dct_finished;
                                        logic
     dct1_finished;

                                        // 64 Addresses Memory Blocks
                                        logic      signed     [13:0]      input_memory [63:0];
                                        logic      signed     [19:0]      dct_memory   [63:0];
                                        logic  signed    [19:0]   zz_memory    [63:0];
                                        // DCT INPUT OUTPUT
                                        logic      signed     [13:0]      x1, x2, x3, x4, x5, x6, x7, x8;
                                        logic      signed     [19:0]      y1, y2, y3, y4, y5, y6, y7, y8;
                                        logic      signed     [19:0]      d1, d2, d3, d4, d5, d6, d7, d8;

                                        logic                 [31:0]      data_out_wire;
                                        logic                             [31:0]     neg;



                                        // RLC
                                        logic signed [31:0] buckets [63:0];
                                        logic [5:0] count;
                                        logic [22:0] rlc [63:0];
                                        logic signed [19:0] data;
                                        logic [5:0]  rlc_index;
                                        logic [3:0]  zero_count;
                                        logic [3:0]  overhead;
                                        logic [5:0]  last_index;
                                        logic [19:0] data_new;
                                        logic [3:0]  category;
                                        logic [3:0]  category2;
                                        logic [3:0]  run;
                                        logic [14:0] rlc_data;
                                        logic data_before_zero;        // 1 if data before is zero, 0 if
nonzero
                                        // HUFFMAN
                                        logic [5:0]  count_ac;
                                        logic [15:0] ac_codeword;
                                        logic [4:0]  ac_codeword_length;
                                        logic [31:0] bucket_out;
                                        logic [19:0] data_neg;
                                        logic [4:0] bucket_length;
                                        //*****FINAL OUTPUT *****//
                                        logic [511:0] final_out;
                                        logic [9:0] final_length;
                                        assign run  = rlc[count_ac][22:19];
                                        assign category2 = rlc[count_ac][18:15];
                                        assign rlc_data = rlc[count_ac][14:0];
                                        assign bucket_out = (ac_codeword << category2) + rlc_data;
                                        assign bucket_length = category2 + ac_codeword_length;
                                        assign data = zz_memory[count];
                                        assign data_neg = ~((~($unsigned(data))) + 1);
                                        assign data_new = (data >= 0)?data:

     (data >= -1)?                  0:

     (data >= -3)?                 ({18'd0, data_neg[1:0]}):

     (data >= -7)?                 ({17'd0, data_neg[2:0]}):

     (data >= -15)?                ({16'd0, data_neg[3:0]}):

     (data >= -31)?                ({15'd0, data_neg[4:0]}):

     (data >= -63)?                ({14'd0, data_neg[5:0]}):

     (data >= -127)?      ({13'd0, data_neg[6:0]}):
     (data >= -255)?       ({12'd0, data_neg[7:0]}):

     (data >= -511)?       ({11'd0, data_neg[8:0]}):

     (data >= -1023)?     ({10'd0, data_neg[9:0]}):

     (data >= -2047)?     ({9'd0, data_neg[10:0]}): 0;

                                        getcategory gc0(data, category);
                                        ac_huffman_table ac1 (run, category2, ac_codeword,
ac_codeword_length);
                                        // DC
                                        logic  signed [19:0]    data_old;
                                        logic  signed [19:0]    dc_diff;
                                        logic  [8:0]     dc_codeword;
                                        logic  [4:0]     dc_codeword_length;
                                        assign dc_diff = zz_memory[0] - data_old;
                                        dc_huffman_table dc1 (category, dc_codeword,
dc_codeword_length);
                                        logic [19:0] diff_neg;
                                        logic [19:0] diff_new;
                                        logic [19:0] diff_new_reg;
                                        assign diff_neg = ~((~($unsigned(dc_diff))) + 1);
                                        assign diff_new = (dc_diff >= 0)?dc_diff:

     (dc_diff >= -1)?                    0:

     (dc_diff >= -3)?                   ({18'd0, diff_neg[1:0]}):

     (dc_diff >= -7)?                   ({17'd0, diff_neg[2:0]}):

     (dc_diff >= -15)?                  ({16'd0, diff_neg[3:0]}):

     (dc_diff >= -31)?                  ({15'd0, diff_neg[4:0]}):

     (dc_diff >= -63)?                  ({14'd0, diff_neg[5:0]}):

     (dc_diff >= -127)?   ({13'd0, diff_neg[6:0]}):

     (dc_diff >= -255)?   ({12'd0, diff_neg[7:0]}):

     (dc_diff >= -511)?   ({11'd0, diff_neg[8:0]}):

     (dc_diff >= -1023)?  ({10'd0, diff_neg[9:0]}):

     (dc_diff >= -2047)?  ({9'd0, diff_neg[10:0]}): 100;

                                        assign     readdata = data_out_wire;
                                        //assign    data_out_wire = {12'b000000000000,
zz_memory[addr_read]};
                                        //assign    data_out_wire = {9'b000000000, rlc[addr_read]};
                                        //assign    data_out_wire = buckets[addr_read];
                                        assign     data_out_wire = final_out[511:480];
                                        //assign data_out_wire = final_length;
                                        //assign    data_out_wire = zz_memory[addr_read];
                                        assign     x1 =
(dct1_finished)?d1[13:0]:input_memory[addr_dct_input];
                                        assign     x2 =
(dct1_finished)?d2[13:0]:input_memory[addr_dct_input+1];
                                        assign     x3 =
(dct1_finished)?d3[13:0]:input_memory[addr_dct_input+2];
                                        assign     x4 =
(dct1_finished)?d4[13:0]:input_memory[addr_dct_input+3];
                                        assign     x5 =
(dct1_finished)?d5[13:0]:input_memory[addr_dct_input+4];
                                        assign     x6 =
(dct1_finished)?d6[13:0]:input_memory[addr_dct_input+5];
```

```
                                        assign    x7 = 
(dct1_finished)?d7[13:0]:input_memory[addr_dct_input+6];
                                        assign    x8 = 
(dct1_finished)?d8[13:0]:input_memory[addr_dct_input+7];

                                        assign    d1 = dct_memory[addr_dct_input];
                                        assign    d2 = dct_memory[addr_dct_input+1];
                                        assign    d3 = dct_memory[addr_dct_input+2];
                                        assign    d4 = dct_memory[addr_dct_input+3];
                                        assign    d5 = dct_memory[addr_dct_input+4];
                                        assign    d6 = dct_memory[addr_dct_input+5];
                                        assign    d7 = dct_memory[addr_dct_input+6];
                                        assign    d8 = dct_memory[addr_dct_input+7];
                                        dct1 DCT_1(clk, x1, x2, x3, x4, x5, x6, x7, x8, y1, y2, y3, y4, 
y5, y6, y7, y8);


                        always_ff @(posedge clk) begin
                            if (reset) begin
                                                addr_write
    <= 6'b000000;
                                                addr_read
    <= 6'b111111;
                                                addr_dct_input      <= 6'd0;
                                                input_loaded                <= 1'b0;
                                                stages
    <= 5'b00000;
                                                dct_finished                <= 1'b0;
                                                dct1_finished               <= 1'b0;

                                                count          <=  0;
                            data_before_zero    <=  0;
                            rlc_index          <=  0;
                            zero_count         <=  0;
                            overhead           <=  0;
                            count_ac           <=  1;

                                                data_old                     <=  0;
                                                diff_new_reg       <=  0;
                                                final_out                    <=  0;
                                                final_length                 <=  0;
                                end else if (chipselect && write && !input_loaded) begin //loads inputs 
to buffer
                                                if (addr_write == 6'b111100) begin
                                                    input_loaded <= 1'b1;
                                                end
                                                input_memory[addr_write]      <= 
writedata[7:0] -   8'd128;
                                                input_memory[addr_write+1] <= 
writedata[15:8] -  8'd128;
                                                input_memory[addr_write+2] <= 
writedata[23:16] - 8'd128;
                                                input_memory[addr_write+3] <= 
writedata[31:24] - 8'd128;
                                                addr_write <= addr_write + 4;
                                        end else if(chipselect && read && !read_before) begin
                                                read_before <= 1'b1;
                                                addr_read <= addr_read + 1;
                                                if (addr_read != 6'b111111) begin
                                                    final_out <= (final_out << 32);
                                                end
                                        end else if (input_loaded) begin             // DCT
                                            if (stages == 5'd0) begin
                            // state 0
                                                    addr_dct_input <= 0;
                                                    stages <= 5'd1;
                                                    final_out
<=  0;
```

```
                                                            final_length
<=  0;
                                                end else if(stages == 5'd1) begin
                    // state 1
                                                            addr_dct_input <= 8;
                                                            stages <= 5'd2;
                                                end else if(stages == 5'd2) begin
                    // state 2
                                                            addr_dct_input <= 16;
                                                            stages <= 5'd3;
                                                end else if(stages == 5'd3) begin
                    // state 3
                                                            addr_dct_input <= 24;
                                                            stages <= 5'd4;
                                                end else if(stages == 5'd4) begin
                    // state 4
                                                            addr_dct_input      <= 32;
                                                            dct_memory[0]       <= y1;
                                                            dct_memory[8]       <= y2;
                                                            dct_memory[16]      <= y3;
                                                            dct_memory[24]      <= y4;
                                                            dct_memory[32]      <= y5;
                                                            dct_memory[40]      <= y6;
                                                            dct_memory[48]      <= y7;
                                                            dct_memory[56]      <= y8;
                                                            stages <= 5'd5;
                                                end else if(stages == 5'd5) begin
                    // state 5
                                                            addr_dct_input      <= 40;
                                                            dct_memory[1]       <= y1;
                                                            dct_memory[9]       <= y2;
                                                            dct_memory[17]      <= y3;
                                                            dct_memory[25]      <= y4;
                                                            dct_memory[33]      <= y5;
                                                            dct_memory[41]      <= y6;
                                                            dct_memory[49]      <= y7;
                                                            dct_memory[57]      <= y8;
                                                            stages <= 5'd6;
                                                end else if(stages == 5'd6) begin
                    // state 6
                                                            addr_dct_input      <= 48;
                                                            dct_memory[2]       <= y1;
                                                            dct_memory[10]      <= y2;
                                                            dct_memory[18]      <= y3;
                                                            dct_memory[26]      <= y4;
                                                            dct_memory[34]      <= y5;
                                                            dct_memory[42]      <= y6;
                                                            dct_memory[50]      <= y7;
                                                            dct_memory[58]      <= y8;
                                                            stages <= 5'd7;
                                                end else if(stages == 5'd7) begin
                    // state 7
                                                            addr_dct_input      <= 56;
                                                            dct_memory[3]       <= y1;
                                                            dct_memory[11]      <= y2;
                                                            dct_memory[19]      <= y3;
                                                            dct_memory[27]      <= y4;
                                                            dct_memory[35]      <= y5;
                                                            dct_memory[43]      <= y6;
                                                            dct_memory[51]      <= y7;
                                                            dct_memory[59]      <= y8;
                                                            stages <= 5'd8;
                                                end else if(stages == 5'd8) begin
                    // state 8
                                                            dct_memory[4]       <= y1;
                                                            dct_memory[12]      <= y2;
                                                            dct_memory[20]      <= y3;
                                                            dct_memory[28]      <= y4;
                                                            dct_memory[36]      <= y5;
                                                            dct_memory[44]      <= y6;
                                                            dct_memory[52]      <= y7;
                                                            dct_memory[60]      <= y8;
                                                            stages <= 5'd9;
                                                end else if(stages == 5'd9) begin
                    // state 9
                                                            dct_memory[5]       <= y1;
                                                            dct_memory[13]      <= y2;
                                                            dct_memory[21]      <= y3;
                                                            dct_memory[29]      <= y4;
                                                            dct_memory[37]      <= y5;
                                                            dct_memory[45]      <= y6;
                                                            dct_memory[53]      <= y7;
                                                            dct_memory[61]      <= y8;
                                                            stages <= 5'd10;
                                                end else if(stages == 5'd10) begin
                    // state 10
                                                            dct_memory[6]       <= y1;
                                                            dct_memory[14]      <= y2;
                                                            dct_memory[22]      <= y3;
                                                            dct_memory[30]      <= y4;
                                                            dct_memory[38]      <= y5;
                                                            dct_memory[46]      <= y6;
                                                            dct_memory[54]      <= y7;
                                                            dct_memory[62]      <= y8;
                                                            stages <= 5'd11;
                                                end else if(stages == 5'd11) begin
                    // state 11  dct1 end
                                                            dct_memory[7]       <= y1;
                                                            dct_memory[15]      <= y2;
                                                            dct_memory[23]      <= y3;
                                                            dct_memory[31]      <= y4;
                                                            dct_memory[39]      <= y5;
                                                            dct_memory[47]      <= y6;
                                                            dct_memory[55]      <= y7;
                                                            dct_memory[63]      <= y8;
                                                            dct1_finished       <= 1'b1;
                                                            stages
                        <= 5'd12;
                                                end else if(stages == 5'd12) begin
                    // state 12
                                                            addr_dct_input <= 0;
                                                            stages <= 5'd13;
                                                end else if(stages == 5'd13) begin
                    // state 13
                                                            addr_dct_input <= 8;
                                                            stages <= 5'd14;
                                                end else if(stages == 5'd14) begin
                    // state 14
                                                            addr_dct_input <= 16;
                                                            stages <= 5'd15;
                                                end else if(stages == 5'd15) begin
                    // state 15
                                                            addr_dct_input <= 24;
                                                            stages <= 5'd16;
                                                end else if(stages == 5'd16) begin
                    // state 16
                                                            addr_dct_input      <= 32;
                                                            zz_memory[0]        <= (y1 <
                    0) ? ((y1 >>> 4) + 1) : (y1 >>> 4);
                                                            zz_memory[2]        <= (y2 <
                    0) ? ((y2 >>> 4) + 1) : (y2 >>> 4);
                                                            zz_memory[3]        <= (y3 <
                    0) ? ((y3 >>> 4) + 1) : (y3 >>> 4);
                                                            zz_memory[9]        <= (y4 <
                    0) ? ((y4 >>> 4) + 1) : (y4 >>> 4);
                                                            zz_memory[10]       <= (y5 <
                    0) ? ((y5 >>> 5) + 1) : (y5 >>> 5);
                                                            zz_memory[20]       <= (y6 <
                    0) ? ((y6 >>> 6) + 1) : (y6 >>> 6);
```

```
0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);

0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                // state 17

0) ? ((y1 >>> 4) + 1) : (y1 >>> 4);

0) ? ((y2 >>> 4) + 1) : (y2 >>> 4);

0) ? ((y3 >>> 4) + 1) : (y3 >>> 4);

0) ? ((y4 >>> 4) + 1) : (y4 >>> 4);

0) ? ((y5 >>> 5) + 1) : (y5 >>> 5);

0) ? ((y6 >>> 6) + 1) : (y6 >>> 6);

0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);

0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                // state 18

0) ? ((y1 >>> 4) + 1) : (y1 >>> 4);

0) ? ((y2 >>> 4) + 1) : (y2 >>> 4);

0) ? ((y3 >>> 4) + 1) : (y3 >>> 4);

0) ? ((y4 >>> 5) + 1) : (y4 >>> 5);

0) ? ((y5 >>> 5) + 1) : (y5 >>> 5);

0) ? ((y6 >>> 6) + 1) : (y6 >>> 6);

0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);

0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                // state 19

0) ? ((y1 >>> 4) + 1) : (y1 >>> 4);

0) ? ((y2 >>> 4) + 1) : (y2 >>> 4);

0) ? ((y3 >>> 5) + 1) : (y3 >>> 5);

0) ? ((y4 >>> 5) + 1) : (y4 >>> 5);

0) ? ((y5 >>> 6) + 1) : (y5 >>> 6);

0) ? ((y6 >>> 6) + 1) : (y6 >>> 6);

0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);

0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                // state 20

0) ? ((y1 >>> 5) + 1) : (y1 >>> 5);
```

```
                zz_memory[21]       <= (y7 <

                zz_memory[35]       <= (y8 <

                stages <= 5'd17;
        end else if(stages == 5'd17) begin

                addr_dct_input      <= 40;
                zz_memory[1]        <= (y1 <

                zz_memory[4]        <= (y2 <

                zz_memory[8]        <= (y3 <

                zz_memory[11]       <= (y4 <

                zz_memory[19]       <= (y5 <

                zz_memory[22]       <= (y6 <

                zz_memory[34]       <= (y7 <

                zz_memory[36]       <= (y8 <

                stages <= 5'd18;
        end else if(stages == 5'd18) begin

                addr_dct_input      <= 48;
                zz_memory[5]        <= (y1 <

                zz_memory[7]        <= (y2 <

                zz_memory[12]       <= (y3 <

                zz_memory[18]       <= (y4 <

                zz_memory[23]       <= (y5 <

                zz_memory[33]       <= (y6 <

                zz_memory[37]       <= (y7 <

                zz_memory[48]       <= (y8 <

                stages <= 5'd19;
        end else if(stages == 5'd19) begin

                addr_dct_input <= 56;
                zz_memory[6]        <= (y1 <

                zz_memory[13]       <= (y2 <

                zz_memory[17]       <= (y3 <

                zz_memory[24]       <= (y4 <

                zz_memory[32]       <= (y5 <

                zz_memory[38]       <= (y6 <

                zz_memory[47]       <= (y7 <

                zz_memory[49]       <= (y8 <

                stages <= 5'd20;
        end else if(stages == 5'd20) begin

                zz_memory[14]       <= (y1 <
0) ? ((y2 >>> 5) + 1) : (y2 >>> 5);
0) ? ((y3 >>> 5) + 1) : (y3 >>> 5);
0) ? ((y4 >>> 5) + 1) : (y4 >>> 5);
0) ? ((y5 >>> 7) + 1) : (y5 >>> 7);
0) ? ((y6 >>> 7) + 1) : (y6 >>> 7);
0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);
0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                    // state 21
0) ? ((y1 >>> 6) + 1) : (y1 >>> 6);
0) ? ((y2 >>> 6) + 1) : (y2 >>> 6);
0) ? ((y3 >>> 6) + 1) : (y3 >>> 6);
0) ? ((y4 >>> 6) + 1) : (y4 >>> 6);
0) ? ((y5 >>> 7) + 1) : (y5 >>> 7);
0) ? ((y6 >>> 7) + 1) : (y6 >>> 7);
0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);
0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                    // state 22
0) ? ((y1 >>> 6) + 1) : (y1 >>> 6);
0) ? ((y2 >>> 6) + 1) : (y2 >>> 6);
0) ? ((y3 >>> 6) + 1) : (y3 >>> 6);
0) ? ((y4 >>> 6) + 1) : (y4 >>> 6);
0) ? ((y5 >>> 7) + 1) : (y5 >>> 7);
0) ? ((y6 >>> 7) + 1) : (y6 >>> 7);
0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);
0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);

                    // state 23
0) ? ((y1 >>> 6) + 1) : (y1 >>> 6);
0) ? ((y2 >>> 6) + 1) : (y2 >>> 6);
0) ? ((y3 >>> 6) + 1) : (y3 >>> 6);
0) ? ((y4 >>> 6) + 1) : (y4 >>> 6);
0) ? ((y5 >>> 7) + 1) : (y5 >>> 7);
0) ? ((y6 >>> 7) + 1) : (y6 >>> 7);
0) ? ((y7 >>> 7) + 1) : (y7 >>> 7);
```

```
            zz_memory[16]      <= (y2 <
            zz_memory[25]      <= (y3 <
            zz_memory[31]      <= (y4 <
            zz_memory[39]      <= (y5 <
            zz_memory[46]      <= (y6 <
            zz_memory[50]      <= (y7 <
            zz_memory[57]      <= (y8 <
            stages <= 5'd21;
end else if(stages == 5'd21) begin
            zz_memory[15]      <= (y1 <
            zz_memory[26]      <= (y2 <
            zz_memory[30]      <= (y3 <
            zz_memory[40]      <= (y4 <
            zz_memory[45]      <= (y5 <
            zz_memory[51]      <= (y6 <
            zz_memory[56]      <= (y7 <
            zz_memory[48]      <= (y8 <
            stages <= 5'd22;
end else if(stages == 5'd22) begin
            zz_memory[27]      <= (y1 <
            zz_memory[29]      <= (y2 <
            zz_memory[41]      <= (y3 <
            zz_memory[44]      <= (y4 <
            zz_memory[52]      <= (y5 <
            zz_memory[55]      <= (y6 <
            zz_memory[59]      <= (y7 <
            zz_memory[62]      <= (y8 <
            stages <= 5'd23;
end else if(stages == 5'd23) begin
            zz_memory[28]      <= (y1 <
            zz_memory[42]      <= (y2 <
            zz_memory[43]      <= (y3 <
            zz_memory[53]      <= (y4 <
            zz_memory[54]      <= (y5 <
            zz_memory[60]      <= (y6 <
            zz_memory[61]      <= (y7 <
                                        zz_memory[63]       <= (y8 <
0) ? ((y8 >>> 7) + 1) : (y8 >>> 7);
                                        stages <= 5'd24;
                                end else if(stages == 5'd24) begin
                // state 24 DC Calculation
                                                stages
        <=      5'd25;
                                                        rlc_index
        <=      0;
                                                        count
                <=      0;
                                end else if(stages == 5'd25) begin
                // state 25   ****RLC******
                                if (data != 0 && !data_before_zero) begin // if data is
nonzero, and data before is nonzero, put [0, data]
rlc[rlc_index]      <= {zero_count[3:0], category[3:0], data_new[14:0]};
                                   rlc_index          <= rlc_index + 1;
                                   zero_count         <= 0;
                                   data_before_zero   <= 0;
                                   overhead           <= 0;
                                end else if (data == 0 && !data_before_zero) begin  //
if data is zero, data before is nonzero, zero_count=1,
                                   rlc_index          <= rlc_index;
                                   zero_count         <= 1;
                                   data_before_zero   <= 1;
                                   overhead           <= overhead;
                                end else if (data != 0 && data_before_zero) begin
//if data is nonzero,data before is zero, then put [zero_count, data]
                                   rlc[rlc_index + overhead]    <= {zero_count[3:0],
category[3:0], data_new[14:0]};
                                   rlc_index          <= rlc_index + overhead + 1;
                                   zero_count         <= 0;
                                   data_before_zero   <= 0;
                                   overhead           <= 0;
                                end else if (data == 0 && data_before_zero) begin   //
if data is zero, data before is also zero, zero_count ++
                                   if (zero_count == 15 && count != 63) begin
                                      rlc[rlc_index + overhead]    <= {4'b1111, 4'd0,
15'd0};
                                      rlc_index          <= rlc_index;
                                      zero_count         <= 0;
                                      data_before_zero   <= 0;
                                      overhead           <= overhead + 1;
                                   end else begin
                                      if (count == 63) begin                     // end EOB if
last number is 0, put [0, 0]
                                         rlc[rlc_index]    <= {4'd0, 4'd0, 15'd0};
                                         rlc_index          <= rlc_index;
                                         zero_count         <= 0;
                                         data_before_zero   <= 1;
                                         overhead           <= overhead;
                                         last_index         <= rlc_index;
                                      end else begin
                                         rlc_index          <= rlc_index;
                                         zero_count         <= zero_count + 1;
                                         data_before_zero   <= 1;
                                         overhead           <= overhead;
                                      end
                                   end
                                end
                                if (count == 63) begin
                                   stages <= 5'd26;                              // RLC
FINISHED
                                   count_ac <= 1;
                                end
                                count <= count + 1;
                                end else if(stages == 5'd26) begin
                // state 26
```

```
                                        zz_memory[0]        <=
        dc_diff;
                                        data_old            <=
        zz_memory[0];
                                        count
        <=          0;
                                        diff_new_reg  <=   diff_new;
                                        stages
        <= 5'd27;
                                end else if(stages == 5'd27) begin
            // state 27
                                        buckets[0]   <= dc_codeword;
                                        final_out    <= (dc_codeword
<< category) + diff_new_reg;
                                        final_length <=
dc_codeword_length + category;
                                    //final_out    <= dc_codeword;
                                        //final_length <=
dc_codeword_length;
                                        stages <= 5'd28;
                                end else if(stages == 5'd28) begin
            // state 28
                                        if (count_ac == last_index ||
bucket_out == 10) begin
                                 stages <= 5'd29;
                                        end         else begin
                                count_ac    <= count_ac  + 1;
                              end
                              buckets[count_ac] <= bucket_out;
                                        final_out <= (final_out <<
bucket_length) + bucket_out;
                                        final_length <= final_length +
bucket_length;
                                end else if(stages == 5'd29) begin
            // state 29  shift final output to the MSB
                                    stages <= 5'd30;
                                       //final_out <= (final_out << 20);
                                        final_out <= (final_out << (512
- final_length));
                                end else if(stages == 5'd30) begin
            // state 30
                                        addr_write
        <= 6'b000000;
                                        addr_read
        <= 6'b111111;
                                        addr_dct_input      <= 6'd0;
                                        input_loaded
        <= 1'b0;
                                        stages
        <= 5'd0;
                                        dct_finished
        <= 1'b1;
                                        dct1_finished
        <= 1'b0;

                                    count             <=  0;
                  data_before_zero    <=  0;
                  rlc_index          <=  0;
                  zero_count         <=  0;
                  overhead           <=  0;
                  count_ac           <=  1;

                                        //final_out
<=  0;
                                        //final_length
<=  0;
                                end
                            end else begin
```

```
                                                                        read_before <= 1'b0;
                                                    end
                                             end

endmodule


module dct1 (clk, x1, x2, x3, x4, x5, x6, x7, x8, y1, y2, y3, y4, y5, y6, y7, y8);
       input clk;
       input signed [13:0] x1, x2, x3, x4, x5, x6, x7, x8;
       output signed [19:0] y1, y2, y3, y4, y5, y6, y7, y8;
       //output signed [26:0] temp0, temp1, temp2, temp3;
       wire signed [14:0] a1, a2, a3, a4, a5, a6, a7, a8;
       reg signed [15:0] a5_reg, a6_reg, a7_reg, a8_reg;



       reg signed [15:0] sb1_reg, sb2_reg, sb3_reg, sb4_reg, sb5_reg, sb6_reg, sb7_reg, sb8_reg, sb9_reg, sb10_reg,
sb11_reg, sb12_reg, sb13_reg;
       wire signed [15:0] sb1, sb2, sb3, sb4, sb5, sb6, sb7, sb8, sb9, sb10, sb11, sb12, sb13;

       // y1
       wire signed [16:0] y1_1_1;
       reg signed [24:0] y1_3_1_reg;
       wire signed [26:0] y1_5_1;
       reg signed [26:0] y1_5_1_reg;
       wire signed [26:0] y1_6_1, y1_6_2;
       wire signed [19:0] y1_7_1;

       // y2
       wire signed [16:0] y2_1_1, y2_1_2, y2_1_3, y2_1_4, y2_1_5;
       wire signed [23:0] y2_2_1, y2_2_2, y2_2_3, y2_2_4, y2_2_5, y2_2_6;
       wire signed [24:0] y2_3_1, y2_3_2, y2_3_3;
       reg signed [24:0] y2_3_1_reg, y2_3_2_reg, y2_3_3_reg;
       wire signed [25:0] y2_3_3_new;
       wire signed [25:0] y2_4_1;
       wire signed [26:0] y2_5_1;
       reg signed [26:0] y2_5_1_reg;
       wire signed [26:0] y2_6_1, y2_6_2;
       wire signed [19:0] y2_7_1;

       // y3
       wire signed [23:0] y3_2_1, y3_2_2, y3_2_3, y3_2_4, y3_2_5;
       wire signed [23:0] sb5_reg_new;
       wire signed [24:0] y3_3_1, y3_3_2, y3_3_3;
       reg signed [24:0] y3_3_1_reg, y3_3_2_reg, y3_3_3_reg;
       wire signed [25:0] y3_3_3_new;
       wire signed [25:0] y3_4_1;
       wire signed [26:0] y3_5_1;
       reg signed [26:0] y3_5_1_reg;
       wire signed [26:0] y3_6_1;
       wire signed [19:0] y3_7_1;
       // y4
       wire signed [16:0] y4_1_1, y4_1_2;
       wire signed [23:0] sb12_reg_new;
       wire signed [23:0] y4_2_1, y4_2_2, y4_2_3, y4_2_4, y4_2_5;
       wire signed [24:0] y4_3_1, y4_3_2, y4_3_3;
       reg signed  [24:0] y4_3_1_reg, y4_3_2_reg, y4_3_3_reg;
       wire signed [25:0] y4_3_3_new;
       wire signed [25:0] y4_4_1;
       wire signed [26:0] y4_5_1;
       reg signed [26:0] y4_5_1_reg;
       wire signed [26:0] y4_6_1;
       wire signed [19:0] y4_7_1;
       // y5
       wire signed [16:0] y5_1_1;
       reg signed  [24:0] y5_3_1_reg;
       wire signed [26:0] y5_5_1;
```

```
		reg signed [26:0] y5_5_1_reg;
		wire signed [26:0] y5_6_1, y5_6_2;
		wire signed [19:0] y5_7_1;

		// y6
		wire signed [16:0] y6_1_1, y6_1_2;
		wire signed [23:0] y6_2_1, y6_2_2, y6_2_3, y6_2_4, y6_2_5;
		wire signed [23:0] sb10_reg_new;
		wire signed [24:0] y6_3_1, y6_3_2, y6_3_3;
		reg signed  [24:0] y6_3_1_reg, y6_3_2_reg, y6_3_3_reg;
		wire signed [25:0] y6_3_3_new;
		wire signed [25:0] y6_4_1;
		wire signed [26:0] y6_5_1;
		reg signed [26:0] y6_5_1_reg;
		wire signed [26:0] y6_6_1;
		wire signed [19:0] y6_7_1;
		// y7
		wire signed [23:0] y7_2_1, y7_2_2, y7_2_3, y7_2_4, y7_2_5;
		wire signed [23:0] sb6_reg_new;
		wire signed [24:0] y7_3_1, y7_3_2, y7_3_3;
		reg signed [24:0] y7_3_1_reg, y7_3_2_reg, y7_3_3_reg;
		wire signed [25:0] y7_3_3_new;
		wire signed [25:0] y7_4_1;
		wire signed [26:0] y7_5_1;
		reg signed [26:0] y7_5_1_reg;
		wire signed [26:0] y7_6_1;
		wire signed [19:0] y7_7_1;
		// y8
		wire signed [16:0] y8_1_1,y8_1_2;
		wire signed [23:0] y8_2_1,y8_2_2,y8_2_3,y8_2_4,y8_2_5;
		wire signed [23:0] sb11_reg_new;
		wire signed [24:0] y8_3_1,y8_3_2,y8_3_3;
		reg  signed [24:0] y8_3_1_reg,y8_3_2_reg,y8_3_3_reg;
		wire signed [25:0] y8_3_3_new;
		wire signed [25:0] y8_4_1;
		wire signed [26:0] y8_5_1;
		reg signed [26:0] y8_5_1_reg;
		wire signed [26:0] y8_6_1;
		wire signed [19:0] y8_7_1;
//////////////////////////////////////////////////////////////////////////////
// stage 1
//////////////////////////////////////////////////////////////////////////////
		adder_14 add0(x1, x8, 0, a1);
		adder_14 add1(x2, x7, 0, a2);
		adder_14 add2(x3, x6, 0, a3);
		adder_14 add3(x4, x5, 0, a4);
		adder_14 add4(x4, x5, 1, a5);
		adder_14 add5(x3, x6, 1, a6);
		adder_14 add6(x2, x7, 1, a7);
		adder_14 add7(x1, x8, 1, a8);

		adder_15 add8(a5, a8, 0, sb1);
		adder_15 add9(a8, a5, 1, sb2);
		adder_15 add10(a6, a7, 1, sb3);
		adder_15 add11(a6, a7, 0, sb4);

		adder_15 add12(a2, a3, 1, sb5);
		adder_15 add13(a1, a4, 1, sb6);
		adder_15 add14(a1, a4, 0, sb7);
		adder_15 add15(a2, a3, 0, sb8);

		adder_15 add16(a6, a5, 1, sb9);
		adder_15 add17(a6, a8, 1, sb10);
		adder_15 add18(a7, a8, 0, sb11);
		adder_15 add19(a5, a7, 1, sb12);
		adder_15 add20(a8, a8, 0, sb13);
//////////////////////////////////////////////////////////////////////////////
// stage 2
```

```
/////////////////////////////////////////////////////////////////////////////
        // y1 adders:
        adder_16 addy1_1_1(sb7_reg, sb8_reg, 0, y1_1_1);
        assign y1_5_1 = y1_3_1_reg;

        assign y1_6_1 = y1_5_1_reg >>> 2;
        assign y1_6_2 = y1_5_1_reg >>> 3;
        adder_19 addy1_7_1(y1_6_1[18:0],y1_6_2[18:0],0,y1_7_1);

        // y2 adders:
        adder_16 addy2_1_1(sb1_reg, sb4_reg, 0, y2_1_1);
        adder_16 addy2_1_2(sb1_reg, sb3_reg, 0, y2_1_2);
        adder_16 addy2_1_3(sb1_reg, sb4_reg, 1, y2_1_3);
        adder_16 addy2_1_4(sb2_reg, sb3_reg, 0, y2_1_4);
        adder_16 addy2_1_5(sb13_reg, sb3_reg, 0, y2_1_5);
        assign y2_2_1 = y2_1_1 <<< 7;
        assign y2_2_2 = sb2_reg <<< 6;
        assign y2_2_3 = y2_1_2 <<< 5;
        assign y2_2_4 = y2_1_5 <<< 3;
        assign y2_2_5 = y2_1_3 <<< 1;
        assign y2_2_6 = y2_1_4;
        adder_24 addy2_3_1(y2_2_1, y2_2_2, 0, y2_3_1);
        adder_24 addy2_3_2(y2_2_3, y2_2_4, 1, y2_3_2);
        adder_24 addy2_3_3(y2_2_5, y2_2_6, 1, y2_3_3);
        assign y2_3_3_new = y2_3_3_reg;                 // 25 bits -> 26 bits
        adder_25 addy2_4_1(y2_3_1_reg, y2_3_2_reg, 1, y2_4_1);
        adder_26 addy2_5_1(y2_4_1, y2_3_3_new, 0, y2_5_1);

        assign y2_6_1 = y2_5_1_reg >>> 9;
        assign y2_6_2 = y2_5_1_reg >>> 10;
        adder_19 addy2_7_1(y2_6_1[18:0],y2_6_2[18:0],0,y2_7_1);

        // y3 adders:
        assign y3_2_1 = sb5_reg <<< 6;
        assign y3_2_2 = sb5_reg <<< 4;
        assign y3_2_3 = sb6_reg <<< 7;
        assign y3_2_4 = sb6_reg <<< 3;
        assign y3_2_5 = sb6_reg <<< 1;
        adder_24 addy3_3_1(y3_2_1, y3_2_2, 1, y3_3_1);
        adder_24 addy3_3_2(y3_2_3, y3_2_4, 1, y3_3_2);
        assign sb5_reg_new = sb5_reg;
        adder_24 addy3_3_3(sb5_reg_new, y3_2_5, 1, y3_3_3);
        assign y3_3_3_new = y3_3_3_reg;                 // 25 bits -> 26 bits
        adder_25 addy3_4_1(y3_3_1_reg, y3_3_2_reg, 0, y3_4_1);
        adder_26 addy3_5_1(y3_4_1, y3_3_3_new, 0, y3_5_1);
        assign y3_6_1 = y3_5_1_reg >>> 8;
        assign y3_7_1 = y3_6_1[19:0];

        // y4 adders:
        adder_16 addy4_1_1(sb11_reg, a5_reg, 1, y4_1_1);
        adder_16 addy4_1_2(a6_reg, a8_reg, 0, y4_1_2);
        assign y4_2_1 = sb10_reg <<< 7;
        assign y4_2_2 = a5_reg <<< 6;
        assign y4_2_3 = sb11_reg <<< 5;
        assign y4_2_4 = y4_1_1 <<< 3;
        assign y4_2_5 = y4_1_2 <<< 1;
        assign sb12_reg_new = sb12_reg;
        adder_24 addy4_3_1(y4_2_2, y4_2_1, 0, y4_3_1);
        adder_24 addy4_3_2(y4_2_4, y4_2_3, 1, y4_3_2);
        adder_24 addy4_3_3(y4_2_5, sb12_reg_new, 0, y4_3_3);
        adder_25 addy4_4_1(y4_3_2_reg, y4_3_1_reg, 1, y4_4_1);
        assign y4_3_3_new = y4_3_3_reg;
        adder_26 addy4_5_1(y4_4_1, y4_3_3_new, 0, y4_5_1);
        assign y4_6_1 = y4_5_1_reg >>> 8;
        assign y4_7_1 = y4_6_1[19:0];
        // y5 adders:

        adder_16 addy5_1_1(sb7_reg, sb8_reg, 1, y5_1_1);
        assign y5_5_1 = y5_3_1_reg;
        assign y5_6_1 = y5_5_1_reg >>> 2;
        assign y5_6_2 = y5_5_1_reg >>> 3;
        adder_19 addy5_7_1(y5_6_1[18:0],y5_6_2[18:0],0,y5_7_1);

        // y6 adders:
        adder_16 addy6_1_1(sb1_reg, a6_reg, 1, y6_1_1);
        adder_16 addy6_1_2(a5_reg, a7_reg, 0, y6_1_2);
        assign y6_2_1 = sb12_reg <<< 7;
        assign y6_2_2 = a8_reg <<< 6;
        assign y6_2_3 = sb9_reg <<< 5;
        assign y6_2_4 = y6_1_1 <<< 3;
        assign y6_2_5 = y6_1_2 <<< 1;
        assign sb10_reg_new = sb10_reg;
        adder_24 addy6_3_1(y6_2_1, y6_2_2, 0, y6_3_1);
        adder_24 addy6_3_2(y6_2_3, y6_2_4, 0, y6_3_2);
        adder_24 addy6_3_3(y6_2_5, sb10_reg_new, 0, y6_3_3);
        adder_25 addy6_4_1(y6_3_1_reg, y6_3_2_reg, 0, y6_4_1);
        assign y6_3_3_new = y6_3_3_reg;
        adder_26 addy6_5_1(y6_4_1, y6_3_3_new, 0, y6_5_1);
        assign y6_6_1 = y6_5_1_reg >>> 8;
        assign y6_7_1 = y6_6_1[19:0];
        // y7 adders:
        assign y7_2_1 = sb6_reg <<< 6;
        assign y7_2_2 = sb6_reg <<< 4;
        assign y7_2_3 = sb5_reg <<< 7;
        assign y7_2_4 = sb5_reg <<< 3;
        assign y7_2_5 = sb5_reg <<< 1;
        assign sb6_reg_new = sb6_reg;
        adder_24 addy7_3_1(y7_2_1, y7_2_2, 1, y7_3_1);
        adder_24 addy7_3_2(sb6_reg_new, y7_2_3, 1, y7_3_2);
        adder_24 addy7_3_3(y7_2_4, y7_2_5, 0, y7_3_3);
        assign y7_3_3_new = y7_3_3_reg;                 // 25 bits -> 26 bits
        adder_25 addy7_4_1(y7_3_1_reg, y7_3_2_reg, 0, y7_4_1);
        adder_26 addy7_5_1(y7_4_1, y7_3_3_new, 0, y7_5_1);
        assign y7_6_1 = y7_5_1_reg >>> 8;
        assign y7_7_1 = y7_6_1[19:0];

        // y8 adders:
        adder_16 addy8_1_1(sb10_reg,a7_reg, 1 , y8_1_1);
        adder_16 addy8_1_2(a6_reg, a5_reg, 0 , y8_1_2);
        assign y8_2_1 = sb9_reg <<< 7;
        assign y8_2_2 = a7_reg <<< 6;
        assign y8_2_3 = sb10_reg <<< 5;
        assign y8_2_4 = y8_1_1 <<< 3;
        assign y8_2_5 = y8_1_2 <<< 1;
        adder_24 addy8_3_1(y8_2_1,y8_2_2, 1 , y8_3_1);
        adder_24 addy8_3_2(y8_2_4,y8_2_3, 1 , y8_3_2);
        assign sb11_reg_new = sb11_reg;
        adder_24 addy8_3_3(y8_2_5,sb11_reg_new, 0 , y8_3_3);
        adder_25 addy8_4_1(y8_3_1_reg,y8_3_2_reg, 0 , y8_4_1);
        assign y8_3_3_new = y8_3_3_reg;
        adder_26 addy8_5_1(y8_3_3_new,y8_4_1, 0 , y8_5_1);
        assign y8_6_1 = y8_5_1_reg >>> 8;
        assign y8_7_1 = y8_6_1[19:0];
////////////////////////////////////////////////////////////////////////////////
// Pipline Registers
////////////////////////////////////////////////////////////////////////////////

        always @(posedge clk) begin
                // Pipline 1
                a5_reg <= a5;
                a6_reg <= a6;
                a7_reg <= a7;
                a8_reg <= a8;


                sb1_reg <= sb1;
                sb2_reg <= sb2;
                sb3_reg <= sb3;
                sb4_reg <= sb4;
                sb5_reg <= sb5;
                sb6_reg <= sb6;
                sb7_reg <= sb7;
                sb8_reg <= sb8;
                sb9_reg <= sb9;
                sb10_reg <= sb10;
                sb11_reg <= sb11;
                sb12_reg <= sb12;
                sb13_reg <= sb13;

                // Pipline 2
                // y1
                y1_3_1_reg  <= y1_1_1;          // 16 bits -> 24 bits
                // y2
                y2_3_1_reg <= y2_3_1;
                y2_3_2_reg <= y2_3_2;
                y2_3_3_reg <= y2_3_3;
                // y3
                y3_3_1_reg <= y3_3_1;
                y3_3_2_reg <= y3_3_2;
                y3_3_3_reg <= y3_3_3;
                // y4
                y4_3_1_reg <= y4_3_1;
                y4_3_2_reg <= y4_3_2;
                y4_3_3_reg <= y4_3_3;
                // y5
                y5_3_1_reg <= y5_1_1;
                // y6
                y6_3_1_reg <= y6_3_1;
                y6_3_2_reg <= y6_3_2;
                y6_3_3_reg <= y6_3_3;
                // y7
                y7_3_1_reg <= y7_3_1;
                y7_3_2_reg <= y7_3_2;
                y7_3_3_reg <= y7_3_3;
                // y8
                y8_3_1_reg <= y8_3_1;
                y8_3_2_reg <= y8_3_2;
                y8_3_3_reg <= y8_3_3;

                // Pipline 3
                y1_5_1_reg <= y1_5_1;
                y2_5_1_reg <= y2_5_1;
                y3_5_1_reg <= y3_5_1;
                y4_5_1_reg <= y4_5_1;
                y5_5_1_reg <= y5_5_1;
                y6_5_1_reg <= y6_5_1;
                y7_5_1_reg <= y7_5_1;
                y8_5_1_reg <= y8_5_1;
        end


        assign y1 = y1_7_1;
        assign y2 = y2_7_1;
        assign y3 = y3_7_1;
        assign y4 = y4_7_1;
        assign y5 = y5_7_1;
        assign y6 = y6_7_1;
        assign y7 = y7_7_1;
        assign y8 = y8_7_1;
        //assign temp0 =  0;
        //assign temp1 =  0;
        //assign temp2 =  0;
        //assign temp3 =  0;

endmodule
```

```
module adder_14(a, b, cin, sum);
		input signed [13:0]a, b;
		input cin;
		output signed [14:0]sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13;
		wire overFlow;
		wire signed [13:0] b_b;

   assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];

		//approxAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		//approxAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		//approxAdder a2 (a[2], b_b[2], c2, sum[2], c3);
		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);
		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], cout);
		xor x0(overFlow, cout, c13);
		assign sum[14] = (overFlow)? cout:sum[13];

endmodule


module adder_15(a, b, cin, sum);
		input signed [14:0] a, b;
		input cin;
		output signed [15:0] sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14;
		wire overFlow;
		wire signed [14:0] b_b;

   assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];
		assign b_b[14] = (cin)? !b[14]: b[14];

		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);

		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], c14);
		fullAdder f14 (a[14], b_b[14], c14, sum[14], cout);
		xor x0(overFlow, cout, c14);
		assign sum[15] = (overFlow)? cout:sum[14];
endmodule

module adder_16(a, b, cin, sum);
		input signed [15:0]a, b;
		input cin;
		output signed [16:0]sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16;
		wire overFlow;
		wire signed [15:0]b_b;

  assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];
		assign b_b[14] = (cin)? !b[14]: b[14];
		assign b_b[15] = (cin)? !b[15]: b[15];

		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);

		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], c14);
		fullAdder f14 (a[14], b_b[14], c14, sum[14], c15);
		fullAdder f15 (a[15], b_b[15], c15, sum[15], cout);
		xor x0(overFlow, cout, c15);
		assign sum[16] = (overFlow)? cout:sum[15];
endmodule

module adder_19(a, b, cin, sum);
		input signed [18:0]a, b;
		input cin;
		output signed [19:0]sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18;
		wire overFlow;
		wire signed [18:0]b_b;

   assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];
		assign b_b[14] = (cin)? !b[14]: b[14];
		assign b_b[15] = (cin)? !b[15]: b[15];
		assign b_b[16] = (cin)? !b[16]: b[16];
		assign b_b[17] = (cin)? !b[17]: b[17];
		assign b_b[18] = (cin)? !b[18]: b[18];

		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);

		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], c14);
		fullAdder f14 (a[14], b_b[14], c14, sum[14], c15);
		fullAdder f15 (a[15], b_b[15], c15, sum[15], c16);
		fullAdder f16 (a[16], b_b[16], c16, sum[16], c17);
		fullAdder f17 (a[17], b_b[17], c17, sum[17], c18);
		fullAdder f18 (a[18], b_b[18], c18, sum[18], cout);
		xor x0(overFlow, cout, c18);
		assign sum[19] = (overFlow)? cout:sum[18];
endmodule

module adder_24(a, b, cin, sum);
		input signed [23:0]a, b;
		input cin;
		output signed [24:0]sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, c23;
		wire overFlow;
		wire signed [23:0]b_b;
```

```
   assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];
		assign b_b[14] = (cin)? !b[14]: b[14];
		assign b_b[15] = (cin)? !b[15]: b[15];
		assign b_b[16] = (cin)? !b[16]: b[16];
		assign b_b[17] = (cin)? !b[17]: b[17];
		assign b_b[18] = (cin)? !b[18]: b[18];
		assign b_b[19] = (cin)? !b[19]: b[19];
		assign b_b[20] = (cin)? !b[20]: b[20];
		assign b_b[21] = (cin)? !b[21]: b[21];
		assign b_b[22] = (cin)? !b[22]: b[22];
		assign b_b[23] = (cin)? !b[23]: b[23];

		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);

		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], c14);
		fullAdder f14 (a[14], b_b[14], c14, sum[14], c15);
		fullAdder f15 (a[15], b_b[15], c15, sum[15], c16);
		fullAdder f16 (a[16], b_b[16], c16, sum[16], c17);
		fullAdder f17 (a[17], b_b[17], c17, sum[17], c18);
		fullAdder f18 (a[18], b_b[18], c18, sum[18], c19);
		fullAdder f19 (a[19], b_b[19], c19, sum[19], c20);
		fullAdder f20 (a[20], b_b[20], c20, sum[20], c21);
		fullAdder f21 (a[21], b_b[21], c21, sum[21], c22);
		fullAdder f22 (a[22], b_b[22], c22, sum[22], c23);
		fullAdder f23 (a[23], b_b[23], c23, sum[23], cout);
		xor x0(overFlow, cout, c23);
		assign sum[24] = (overFlow)? cout:sum[23];
endmodule

module adder_25(a, b, cin, sum);
		input signed [24:0]a, b;
		input cin;
		output signed [25:0]sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24;
		wire overFlow;
		wire signed [24:0]b_b;

   assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];
		assign b_b[14] = (cin)? !b[14]: b[14];
		assign b_b[15] = (cin)? !b[15]: b[15];
		assign b_b[16] = (cin)? !b[16]: b[16];
		assign b_b[17] = (cin)? !b[17]: b[17];
		assign b_b[18] = (cin)? !b[18]: b[18];
		assign b_b[19] = (cin)? !b[19]: b[19];
		assign b_b[20] = (cin)? !b[20]: b[20];
		assign b_b[21] = (cin)? !b[21]: b[21];
		assign b_b[22] = (cin)? !b[22]: b[22];
		assign b_b[23] = (cin)? !b[23]: b[23];
		assign b_b[24] = (cin)? !b[24]: b[24];

		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);

		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], c14);
		fullAdder f14 (a[14], b_b[14], c14, sum[14], c15);
		fullAdder f15 (a[15], b_b[15], c15, sum[15], c16);
		fullAdder f16 (a[16], b_b[16], c16, sum[16], c17);
		fullAdder f17 (a[17], b_b[17], c17, sum[17], c18);
		fullAdder f18 (a[18], b_b[18], c18, sum[18], c19);
		fullAdder f19 (a[19], b_b[19], c19, sum[19], c20);
		fullAdder f20 (a[20], b_b[20], c20, sum[20], c21);
		fullAdder f21 (a[21], b_b[21], c21, sum[21], c22);
		fullAdder f22 (a[22], b_b[22], c22, sum[22], c23);
		fullAdder f23 (a[23], b_b[23], c23, sum[23], c24);
		fullAdder f24 (a[24], b_b[24], c24, sum[24], cout);
		xor x0(overFlow, cout, c24);
		assign sum[25] = (overFlow)? cout:sum[24];
endmodule

module adder_26(a, b, cin, sum);
		input signed [25:0]a, b;
		input cin;
		output signed [26:0]sum;
		wire cout;
		wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24,
c25;
		wire overFlow;
		wire signed [25:0]b_b;

   assign b_b[0] = (cin)? !b[0]: b[0];
		assign b_b[1] = (cin)? !b[1]: b[1];
		assign b_b[2] = (cin)? !b[2]: b[2];
		assign b_b[3] = (cin)? !b[3]: b[3];
		assign b_b[4] = (cin)? !b[4]: b[4];
		assign b_b[5] = (cin)? !b[5]: b[5];
		assign b_b[6] = (cin)? !b[6]: b[6];
		assign b_b[7] = (cin)? !b[7]: b[7];
		assign b_b[8] = (cin)? !b[8]: b[8];
		assign b_b[9] = (cin)? !b[9]: b[9];
		assign b_b[10] = (cin)? !b[10]: b[10];
		assign b_b[11] = (cin)? !b[11]: b[11];
		assign b_b[12] = (cin)? !b[12]: b[12];
		assign b_b[13] = (cin)? !b[13]: b[13];
		assign b_b[14] = (cin)? !b[14]: b[14];
		assign b_b[15] = (cin)? !b[15]: b[15];
		assign b_b[16] = (cin)? !b[16]: b[16];
		assign b_b[17] = (cin)? !b[17]: b[17];
		assign b_b[18] = (cin)? !b[18]: b[18];
		assign b_b[19] = (cin)? !b[19]: b[19];
		assign b_b[20] = (cin)? !b[20]: b[20];
		assign b_b[21] = (cin)? !b[21]: b[21];
		assign b_b[22] = (cin)? !b[22]: b[22];
		assign b_b[23] = (cin)? !b[23]: b[23];
		assign b_b[24] = (cin)? !b[24]: b[24];
		assign b_b[25] = (cin)? !b[25]: b[25];

		fullAdder a0 (a[0], b_b[0], cin, sum[0], c1);
		fullAdder a1 (a[1], b_b[1], c1, sum[1], c2);
		fullAdder a2 (a[2], b_b[2], c2, sum[2], c3);

		fullAdder f3 (a[3], b_b[3], c3, sum[3], c4);
		fullAdder f4 (a[4], b_b[4], c4, sum[4], c5);
		fullAdder f5 (a[5], b_b[5], c5, sum[5], c6);
		fullAdder f6 (a[6], b_b[6], c6, sum[6], c7);
		fullAdder f7 (a[7], b_b[7], c7, sum[7], c8);
		fullAdder f8 (a[8], b_b[8], c8, sum[8], c9);
		fullAdder f9 (a[9], b_b[9], c9, sum[9], c10);
		fullAdder f10 (a[10], b_b[10], c10, sum[10], c11);
		fullAdder f11 (a[11], b_b[11], c11, sum[11], c12);
		fullAdder f12 (a[12], b_b[12], c12, sum[12], c13);
		fullAdder f13 (a[13], b_b[13], c13, sum[13], c14);
		fullAdder f14 (a[14], b_b[14], c14, sum[14], c15);
		fullAdder f15 (a[15], b_b[15], c15, sum[15], c16);
		fullAdder f16 (a[16], b_b[16], c16, sum[16], c17);
		fullAdder f17 (a[17], b_b[17], c17, sum[17], c18);
		fullAdder f18 (a[18], b_b[18], c18, sum[18], c19);
		fullAdder f19 (a[19], b_b[19], c19, sum[19], c20);
		fullAdder f20 (a[20], b_b[20], c20, sum[20], c21);
		fullAdder f21 (a[21], b_b[21], c21, sum[21], c22);
		fullAdder f22 (a[22], b_b[22], c22, sum[22], c23);
		fullAdder f23 (a[23], b_b[23], c23, sum[23], c24);
		fullAdder f24 (a[24], b_b[24], c24, sum[24], c25);
		fullAdder f25 (a[25], b_b[25], c24, sum[25], cout);
		xor x0(overFlow, cout, c25);
		assign sum[26] = (overFlow)? cout:sum[25];
endmodule



module fullAdder (a, b, cin, sum, cout);
		input a, b, cin;
		output sum, cout;
		wire w1,w2,w3;
		xor x1(w1,a,b);
		xor x2(sum,w1,cin);
		and a1(w2,a,b);
		and a2(w3,w1,cin);
		or o1(cout,w2,w3);
endmodule


module getcategory (data, category);
   input  logic signed [19:0] data;
   output logic [3:0] category;
   always @(data)
   begin
```

```
        if      (data <= 1    &&  data >= -1) begin category = 4'd1; end
        else if (data <= 3    &&  data >= -3) begin category = 4'd2; end
        else if (data <= 7    &&  data >= -7) begin category = 4'd3; end
        else if (data <=15    &&  data >= -15) begin category = 4'd4; end
        else if (data <=31    &&  data >= -31) begin category = 4'd5; end
        else if (data <=63    &&  data >= -63) begin category = 4'd6; end
        else if (data <=127   &&  data >= -127) begin category = 4'd7; end
        else if (data <=255   &&  data >= -255) begin category = 4'd8; end
        else if (data <=511   &&  data >= -511) begin category = 4'd9; end
        else if (data <=1023  &&  data >= -1023) begin category = 4'd10; end
        else if (data <=2047  &&  data >= -2047) begin category = 4'd11; end
        else if (data <=4095  &&  data >= -4095) begin category = 4'd12; end
        else if (data <=8191  &&  data >= -8191) begin category = 4'd13; end
        else if (data <=16383 &&  data >= -16383) begin category = 4'd14; end
        else if (data <=32767 &&  data >= -32767) begin category = 4'd15; end
        else begin category = 4'd0; end
    end
endmodule


module dc_huffman_table(category, codeword, code_length);
    input          [3:0] category;
    output          [8:0] codeword;
    output logic    [4:0] code_length;
    always @(category) begin
        case(category)
            4'd0:   begin codeword = 9'b00; code_length = 2;end
            4'd1:   begin codeword = 9'b010; code_length = 3;end
            4'd2:   begin codeword = 9'b011; code_length = 3;end
            4'd3:   begin codeword = 9'b100; code_length = 3;end
            4'd4:   begin codeword = 9'b101; code_length = 3;end
            4'd5:   begin codeword = 9'b110; code_length = 3;end
            4'd6:   begin codeword = 9'b1110; code_length = 4;end
            4'd7:   begin codeword = 9'b11110; code_length = 5;end
            4'd8:   begin codeword = 9'b111110; code_length = 6;end
            4'd9:   begin codeword = 9'b1111110; code_length = 7;end
            4'd10:  begin codeword = 9'b11111110; code_length = 8;end
            4'd11:  begin codeword = 9'b111111110; code_length = 9;end
                                                default : begin codeword = 9'd0; code_length = 0;end
        endcase
    end
endmodule
// Huffman AC Table
module ac_huffman_table(run, category, ac_codeword, ac_codeword_length);
                input [3:0]run;
                input [3:0]category;
                output logic [15:0] ac_codeword;
                output logic [4:0] ac_codeword_length;

                always @(run or category)
                begin
                                case({run, category})
                                                {4'd0, 4'd0} : begin ac_codeword = 16'b1010; ac_codeword_length = 5'd4; end
                                                {4'd0, 4'd1} : begin ac_codeword = 16'b00; ac_codeword_length = 5'd2; end
                                                {4'd0, 4'd2} : begin ac_codeword = 16'b01; ac_codeword_length = 5'd2; end
                                                {4'd0, 4'd3} : begin ac_codeword = 16'b100; ac_codeword_length = 5'd3; end
                                                {4'd0, 4'd4} : begin ac_codeword = 16'b1011; ac_codeword_length = 5'd4; end
                                                {4'd0, 4'd5} : begin ac_codeword = 16'b11010; ac_codeword_length = 5'd5; end
                                                {4'd0, 4'd6} : begin ac_codeword = 16'b1111000; ac_codeword_length = 5'd7; end
                                                {4'd0, 4'd7} : begin ac_codeword = 16'b11111000; ac_codeword_length = 5'd8; end
                                                {4'd0, 4'd8} : begin ac_codeword = 16'b1111110110; ac_codeword_length = 5'd10; end
                                                {4'd0, 4'd9} : begin ac_codeword = 16'b1111111110000010; ac_codeword_length = 5'd16;
end
                                               {4'd0, 4'd10} : begin ac_codeword = 16'b1111111110000011; ac_codeword_length = 5'd16;
end
                                                {4'd1, 4'd1} : begin ac_codeword = 16'b1100; ac_codeword_length = 5'd4; end
                                                {4'd1, 4'd2} : begin ac_codeword = 16'b11011; ac_codeword_length = 5'd5; end
                                                {4'd1, 4'd3} : begin ac_codeword = 16'b1111001; ac_codeword_length = 5'd7; end
                                                {4'd1, 4'd4} : begin ac_codeword = 16'b111110110; ac_codeword_length = 5'd9; end
                                                {4'd1, 4'd5} : begin ac_codeword = 16'b11111110110; ac_codeword_length = 5'd11; end
```

```
                        {4'd1, 4'd6} : begin ac_codeword = 16'b1111111110000100; ac_codeword_length = 5'd16;
end
                        {4'd1, 4'd7} : begin ac_codeword = 16'b1111111110000101; ac_codeword_length = 5'd16;
end
                        {4'd1, 4'd8} : begin ac_codeword = 16'b1111111110000110; ac_codeword_length = 5'd16;
end
                        {4'd1, 4'd9} : begin ac_codeword = 16'b1111111110000111; ac_codeword_length = 5'd16;
end
                       {4'd1, 4'd10} : begin ac_codeword = 16'b1111111110001000; ac_codeword_length = 5'd16;
end
                        {4'd2, 4'd1} : begin ac_codeword = 16'b11100; ac_codeword_length = 5'd5; end
                        {4'd2, 4'd2} : begin ac_codeword = 16'b11111001; ac_codeword_length = 5'd8; end
                        {4'd2, 4'd3} : begin ac_codeword = 16'b1111110111; ac_codeword_length = 5'd10; end
                        {4'd2, 4'd4} : begin ac_codeword = 16'b111111110100; ac_codeword_length = 5'd12; end
                        {4'd2, 4'd5} : begin ac_codeword = 16'b1111111110001001; ac_codeword_length = 5'd16;
end
                        {4'd2, 4'd6} : begin ac_codeword = 16'b1111111110001010; ac_codeword_length = 5'd16;
end
                        {4'd2, 4'd7} : begin ac_codeword = 16'b1111111110001011; ac_codeword_length = 5'd16;
end
                        {4'd2, 4'd8} : begin ac_codeword = 16'b1111111110001100; ac_codeword_length = 5'd16;
end
                        {4'd2, 4'd9} : begin ac_codeword = 16'b1111111110001101; ac_codeword_length = 5'd16;
end
                       {4'd2, 4'd10} : begin ac_codeword = 16'b1111111110001110; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd1} : begin ac_codeword = 16'b111010; ac_codeword_length = 5'd6; end
                        {4'd3, 4'd2} : begin ac_codeword = 16'b111110111; ac_codeword_length = 5'd9; end
                        {4'd3, 4'd3} : begin ac_codeword = 16'b111111110101; ac_codeword_length = 5'd12; end
                        {4'd3, 4'd4} : begin ac_codeword = 16'b1111111110001111; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd5} : begin ac_codeword = 16'b1111111110010000; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd6} : begin ac_codeword = 16'b1111111110010001; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd7} : begin ac_codeword = 16'b1111111110010010; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd8} : begin ac_codeword = 16'b1111111110010011; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd9} : begin ac_codeword = 16'b1111111110010100; ac_codeword_length = 5'd16;
end
                        {4'd3, 4'd10} : begin ac_codeword = 16'b1111111110010101;ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd1} : begin ac_codeword = 16'b111011;          ac_codeword_length = 5'd6; end
                        {4'd4, 4'd2} : begin ac_codeword = 16'b1111111000;       ac_codeword_length = 5'd10; end
                        {4'd4, 4'd3} : begin ac_codeword = 16'b1111111110010110; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd4} : begin ac_codeword = 16'b1111111110010111; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd5} : begin ac_codeword = 16'b1111111110011000; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd6} : begin ac_codeword = 16'b1111111110011001; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd7} : begin ac_codeword = 16'b1111111110011010; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd8} : begin ac_codeword = 16'b1111111110011011; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd9} : begin ac_codeword = 16'b1111111110011100; ac_codeword_length = 5'd16;
end
                        {4'd4, 4'd10}: begin ac_codeword = 16'b1111111110011101; ac_codeword_length = 5'd16;
end
                        {4'd5, 4'd1} : begin ac_codeword = 16'b1111010;          ac_codeword_length = 5'd7; end
                        {4'd5, 4'd2} : begin ac_codeword = 16'b11111110111;      ac_codeword_length = 5'd11;
end
                        {4'd5, 4'd3} : begin ac_codeword = 16'b1111111110011110; ac_codeword_length = 5'd16;
end
                        {4'd5, 4'd4} : begin ac_codeword = 16'b1111111110011111; ac_codeword_length = 5'd16;
end
                        {4'd5, 4'd5} : begin ac_codeword = 16'b1111111110100000; ac_codeword_length = 5'd16;
end
```

```
                              {4'd5, 4'd6} : begin ac_codeword = 16'b1111111110100001; ac_codeword_length = 5'd16;
end
                              {4'd5, 4'd7} : begin ac_codeword = 16'b1111111110100010; ac_codeword_length = 5'd16;
end
                              {4'd5, 4'd8} : begin ac_codeword = 16'b1111111110100011; ac_codeword_length = 5'd16;
end
                              {4'd5, 4'd9} : begin ac_codeword = 16'b1111111110100100; ac_codeword_length = 5'd16;
end
                              {4'd5, 4'd10} : begin ac_codeword = 16'b1111111110100101;ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd1} : begin ac_codeword = 16'b1111011;          ac_codeword_length = 5'd7; end
                              {4'd6, 4'd2} : begin ac_codeword = 16'b111111110110;     ac_codeword_length = 5'd12;
end
                              {4'd6, 4'd3} : begin ac_codeword = 16'b1111111110100110; ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd4} : begin ac_codeword = 16'b1111111110100111; ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd5} : begin ac_codeword = 16'b1111111110101000; ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd6} : begin ac_codeword = 16'b1111111110101001; ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd7} : begin ac_codeword = 16'b1111111110101010; ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd8} : begin ac_codeword = 16'b1111111110101011; ac_codeword_length = 5'd16;
end
                              {4'd6, 4'd9} : begin ac_codeword = 16'b1111111110101100; ac_codeword_length = 5'd16;
end
                             {4'd6, 4'd10} : begin ac_codeword = 16'b1111111110101101; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd1} : begin ac_codeword = 16'b11111010; ac_codeword_length = 5'd8; end
                              {4'd7, 4'd2} : begin ac_codeword = 16'b111111110111; ac_codeword_length = 5'd12; end
                              {4'd7, 4'd3} : begin ac_codeword = 16'b1111111110101110; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd4} : begin ac_codeword = 16'b1111111110101111; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd5} : begin ac_codeword = 16'b1111111110110000; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd6} : begin ac_codeword = 16'b1111111110110001; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd7} : begin ac_codeword = 16'b1111111110110010; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd8} : begin ac_codeword = 16'b1111111110110011; ac_codeword_length = 5'd16;
end
                              {4'd7, 4'd9} : begin ac_codeword = 16'b1111111110110100; ac_codeword_length = 5'd16;
end
                             {4'd7, 4'd10} : begin ac_codeword = 16'b1111111110110101; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd1} : begin ac_codeword = 16'b111111000; ac_codeword_length = 5'd9; end
                              {4'd8, 4'd2} : begin ac_codeword = 16'b111111111000000; ac_codeword_length = 5'd15;
end
                              {4'd8, 4'd3} : begin ac_codeword = 16'b1111111110110110; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd4} : begin ac_codeword = 16'b1111111110110111; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd5} : begin ac_codeword = 16'b1111111110111000; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd6} : begin ac_codeword = 16'b1111111110111001; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd7} : begin ac_codeword = 16'b1111111110111010; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd8} : begin ac_codeword = 16'b1111111110111011; ac_codeword_length = 5'd16;
end
                              {4'd8, 4'd9} : begin ac_codeword = 16'b1111111110111100; ac_codeword_length = 5'd16;
end
                             {4'd8, 4'd10} : begin ac_codeword = 16'b1111111110111101; ac_codeword_length = 5'd16;
end
                              {4'd9, 4'd1} : begin ac_codeword = 16'b111111001; ac_codeword_length = 5'd9; end
                              {4'd9, 4'd2} : begin ac_codeword = 16'b1111111110111110; ac_codeword_length = 5'd16;
end
```

```
                         {4'd9, 4'd3} : begin ac_codeword = 16'b1111111110111111; ac_codeword_length = 5'd16;
end
                         {4'd9, 4'd4} : begin ac_codeword = 16'b1111111111000000; ac_codeword_length = 5'd16;
end
                         {4'd9, 4'd5} : begin ac_codeword = 16'b1111111111000001; ac_codeword_length = 5'd16;
end
                         {4'd9, 4'd6} : begin ac_codeword = 16'b1111111111000010; ac_codeword_length = 5'd16;
end
                         {4'd9, 4'd7} : begin ac_codeword = 16'b1111111111000011; ac_codeword_length = 5'd16;
end
                         {4'd9, 4'd8} : begin ac_codeword = 16'b1111111111000100; ac_codeword_length = 5'd16;
end
                         {4'd9, 4'd9} : begin ac_codeword = 16'b1111111111000101; ac_codeword_length = 5'd16;
end
                        {4'd9, 4'd10} : begin ac_codeword = 16'b1111111111000110; ac_codeword_length = 5'd16;
end
                         {4'd10, 4'd1} : begin ac_codeword = 16'b111111010; ac_codeword_length = 5'd9; end
                        {4'd10, 4'd2} : begin ac_codeword = 16'b1111111111000111; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd3} : begin ac_codeword = 16'b1111111111001000; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd4} : begin ac_codeword = 16'b1111111111001001; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd5} : begin ac_codeword = 16'b1111111111001010; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd6} : begin ac_codeword = 16'b1111111111001011; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd7} : begin ac_codeword = 16'b1111111111001100; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd8} : begin ac_codeword = 16'b1111111111001101; ac_codeword_length = 5'd16;
end
                        {4'd10, 4'd9} : begin ac_codeword = 16'b1111111111001110; ac_codeword_length = 5'd16;
end
                         {4'd10, 4'd10} : begin ac_codeword = 16'b1111111111001111; ac_codeword_length =
5'd16; end
                         {4'd11, 4'd1} : begin ac_codeword = 16'b1111111001; ac_codeword_length = 5'd10; end
                        {4'd11, 4'd2} : begin ac_codeword = 16'b1111111111010000; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd3} : begin ac_codeword = 16'b1111111111010001; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd4} : begin ac_codeword = 16'b1111111111010010; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd5} : begin ac_codeword = 16'b1111111111010011; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd6} : begin ac_codeword = 16'b1111111111010100; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd7} : begin ac_codeword = 16'b1111111111010101; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd8} : begin ac_codeword = 16'b1111111111010110; ac_codeword_length = 5'd16;
end
                        {4'd11, 4'd9} : begin ac_codeword = 16'b1111111111010111; ac_codeword_length = 5'd16;
end
                         {4'd11, 4'd10} : begin ac_codeword = 16'b1111111111011000; ac_codeword_length =
5'd16; end
                         {4'd12, 4'd1} : begin ac_codeword = 16'b1111111010; ac_codeword_length = 5'd10; end
                        {4'd12, 4'd2} : begin ac_codeword = 16'b1111111111011001; ac_codeword_length = 5'd16;
end
                        {4'd12, 4'd3} : begin ac_codeword = 16'b1111111111011010; ac_codeword_length = 5'd16;
end
                        {4'd12, 4'd4} : begin ac_codeword = 16'b1111111111011011; ac_codeword_length = 5'd16;
end
                        {4'd12, 4'd5} : begin ac_codeword = 16'b1111111111011100; ac_codeword_length = 5'd16;
end
                        {4'd12, 4'd6} : begin ac_codeword = 16'b1111111111011101; ac_codeword_length = 5'd16;
end
                        {4'd12, 4'd7} : begin ac_codeword = 16'b1111111111011110; ac_codeword_length = 5'd16;
end
                        {4'd12, 4'd8} : begin ac_codeword = 16'b1111111111011111; ac_codeword_length = 5'd16;
end
```

```
                {4'd12, 4'd9} : begin ac_codeword = 16'b1111111111100000; ac_codeword_length = 5'd16; end
                {4'd12, 4'd10}: begin ac_codeword = 16'b1111111111100001; ac_codeword_length = 5'd16; end
                {4'd13, 4'd1} : begin ac_codeword = 16'b11111111000; ac_codeword_length = 5'd11; end
                {4'd13, 4'd2} : begin ac_codeword = 16'b1111111111100010; ac_codeword_length = 5'd16; end
                {4'd13, 4'd3} : begin ac_codeword = 16'b1111111111100011; ac_codeword_length = 5'd16; end
                {4'd13, 4'd4} : begin ac_codeword = 16'b1111111111100100; ac_codeword_length = 5'd16; end
                {4'd13, 4'd5} : begin ac_codeword = 16'b1111111111100101; ac_codeword_length = 5'd16; end
                {4'd13, 4'd6} : begin ac_codeword = 16'b1111111111100110; ac_codeword_length = 5'd16; end
                {4'd13, 4'd7} : begin ac_codeword = 16'b1111111111100111; ac_codeword_length = 5'd16; end
                {4'd13, 4'd8} : begin ac_codeword = 16'b1111111111101000; ac_codeword_length = 5'd16; end
                {4'd13, 4'd9} : begin ac_codeword = 16'b1111111111101001; ac_codeword_length = 5'd16; end
                {4'd13, 4'd10} : begin ac_codeword = 16'b1111111111101010; ac_codeword_length = 5'd16; end
                {4'd14, 4'd1} : begin ac_codeword = 16'b1111111111101011; ac_codeword_length = 5'd16; end
                {4'd14, 4'd2} : begin ac_codeword = 16'b1111111111101100; ac_codeword_length = 5'd16; end
                {4'd14, 4'd3} : begin ac_codeword = 16'b1111111111101101; ac_codeword_length = 5'd16; end
                {4'd14, 4'd4} : begin ac_codeword = 16'b1111111111101110; ac_codeword_length = 5'd16; end
                {4'd14, 4'd5} : begin ac_codeword = 16'b1111111111101111; ac_codeword_length = 5'd16; end
                {4'd14, 4'd6} : begin ac_codeword = 16'b1111111111110000; ac_codeword_length = 5'd16; end
                {4'd14, 4'd7} : begin ac_codeword = 16'b1111111111110001; ac_codeword_length = 5'd16; end
                {4'd14, 4'd8} : begin ac_codeword = 16'b1111111111110010; ac_codeword_length = 5'd16; end
                {4'd14, 4'd9} : begin ac_codeword = 16'b1111111111110011; ac_codeword_length = 5'd16; end
                {4'd14, 4'd10} : begin ac_codeword = 16'b1111111111110100; ac_codeword_length = 5'd16; end
                {4'd15, 4'd1} : begin ac_codeword = 16'b1111111111110101; ac_codeword_length = 5'd16; end
                {4'd15, 4'd2} : begin ac_codeword = 16'b1111111111110110; ac_codeword_length = 5'd16; end
                {4'd15, 4'd3} : begin ac_codeword = 16'b1111111111110111; ac_codeword_length = 5'd16; end
                {4'd15, 4'd4} : begin ac_codeword = 16'b1111111111111000; ac_codeword_length = 5'd16; end
                {4'd15, 4'd5} : begin ac_codeword = 16'b1111111111111001; ac_codeword_length = 5'd16; end
                {4'd15, 4'd6} : begin ac_codeword = 16'b1111111111111010; ac_codeword_length = 5'd16; end
                {4'd15, 4'd7} : begin ac_codeword = 16'b1111111111111011; ac_codeword_length = 5'd16; end
                {4'd15, 4'd8} : begin ac_codeword = 16'b1111111111111100; ac_codeword_length = 5'd16; end
                {4'd15, 4'd9} : begin ac_codeword = 16'b1111111111111101; ac_codeword_length = 5'd16; end
                {4'd15, 4'd10} : begin ac_codeword = 16'b1111111111111110; ac_codeword_length = 5'd16; end
                {4'd15, 4'd0} : begin ac_codeword = 16'b11111111001; ac_codeword_length = 5'd11; end
            endcase

        end
endmodule
```

## 8.3 Matlab Code

### 8.3.1 test_jpeg.m

```
clear all;
close all;
clc;



stream = [];
%image =imread('3.JPG');
%% Convert Image To YCBCR
%image = rgb2ycbcr(image);
image = randi([0 256],8, 8)



image = [100 0 0 100 0 77 0 88;
         100 0 0 0 0 0 0 0;
         100 0 0 0 0 0 0 0;
         100 200 0 0 0 0 0 32;
         100 0 0 0 0 0 0 0;
         100 0 0 0 0 0 0 0;
         0 0 0 0 0 0 0 100;
         0 0 0 0 0 0 0 100];

[row, col, dim] = size(image);
image = image - 128;
q=  [16 16 16 16 32 64 64 64;
      16 16 16 16 54 64 64 64;
      16 16 16 32 32 64 64 64;
      16 16 32 32 32 64 64 64;
      32 32 32 64 128 128 128 128;
      64 64 64 64 128 128 128 128;
      128 128 128 128 128 128 128 128;
      128 128 128 128 128 128 128 128];



%% Segmenting Image Blocks Of 8x8
%% DCT operation
k=0;
image_arr = zeros(row,col);
for i = 1:row
    for j = 1:col
    image_arr(i,j) = image(i,j);
    end
end
fileInput = fopen('input.txt','w');
for i=1:8:col
    for j=1:8:row
        input =image_arr(i:i+7,j:j+7);

        for x = 1:8
            for y = 1:8
                fprintf(fileInput,'%d ', (input(x,y)+128));

            end
            fprintf(fileInput,'\n');
        end


        for a=1:8
            input(:,a) = dct1(input(:,a));
        end
        for a=1:8
            input(a,:) = dct1(input(a,:));
        end
        output = input ./ q;
        k=k+1;
        for b=1:8
             for c=1:8
                if output(b,c) < 0
                   output(b,c) = floor(output(b,c)) + 1;
                else
                    output(b,c) = floor(output(b,c));
                end
             end
        end;
        out(k,:) = zigzag(output);
    end
end
fclose(fileInput);

%% Huffman Compression
dpcm(1,1)=out(1,1);
stream = cat(2,stream,huffman_dc(dpcm(1,1)),huffman_ac(out(1,2:64)));



for m=2:k
    dpcm(m,1)=out(m,1)-out(m-1,1);
    stream=cat(2,stream,huffman_dc(dpcm(m,1)),huffman_ac(out(m,2:64)));
    huffman_dc(dpcm(m,1))
end
stream

fileOutput = fopen('result_matlab.txt','w');
fprintf(fileOutput,'%s', stream);
fclose(fileOutput);

%% Byte Stuffing
%{
p=0;
G=size(stream,2);
for i=1:8:size(stream,2)
    bit_val(1,1:8)=stream(1,i:i+7);
    if strcmp(bit_val(1,1:8),'11111111')==1
        tempbitstream = stream(1,i+8:G+p);
        stream(1,i+8:i+15)='00000000';
        p=p+8;
        temp2_bitstream=stream(1,1:i+15);
        stream(1,1:G+p)=cat(2,temp2_bitstream,tempbitstream);
    end
end
%}




%% Convert String To Decimal
numbytes=floor(length(stream)/8);
diff_stream=length(stream)-numbytes*8;
if diff_stream==0
    matrix_code_decimal= zeros(numbytes+2,8);
else
    matrix_code_decimal= zeros(numbytes+3,8);
```

```
end
s=0;
for count2=1:8:numbytes*8
   s=s+1;
   matrix_code_decimal(s,1)=bin2dec(stream(1,count2:count2+7));
end

if diff_stream~=0
   s=s+1;
   matrix_code_decimal(s,1)=bin2dec(stream(1,numbytes*8+1:length(stream)));
end

matrix_code_decimal(s+1,1)=255;
matrix_code_decimal(s+2,1)=217;



%% Header JFIF
signal=[255         216     255     224     000     016     074     070     073     070     000
            001     002     000     000     096     000     096     000     000     ...
      255   219     000     067     000 016 016 016 016 016 016 016 016 016 016 032 016 016 016 032 ...
      064 054 032 032 032 064 128 064 032 032 032 064 064 064 064 064 032 064 064 128 ...
      128 128 128 064 128 064 064 064 064 064 128 128 128 128 128 128 128 128 064 128 ...
      128 128 128 128 128 128 128 128 128 255   219     000     067     001     016 016 016 016 016 016 ...
      016 016 016 016 032 016 016 016 032 064 054 032 032 032 064 128 064 032 032 032 ...
      064 064 064 064 064 032 064 064 128 128 128 128 064 128 064 064 064 064 064 128 ...
      128 128 128 128 128 128 128 064 128 128 128 128 128 128 128 128 128 128 255     192 ...
      000   011     008     000     000     000     000     001     001     034     000     255
            196     000     031     000     000     001     005     001     ...
      001   001     001     001     001     000     000     000     000     000     000     000
            000     001     002     003     004     005     006     007     ...
      008   009     010     011     255     196     000     181     016     000     002     001
            003     003     002     004     003     005 005 004     ...
      004   000     000     001     125     001     002     003     000     004     017     005
            018     033     049     065     006     019     081     097     ...
      007   034     113     020     050     129     145     161     008     035     066     177
            193     021     082     209     240     036     051     098     ...
      114   130     009     010     022     023     024     025     026     037     038     039
            040     041     042     052     053     054     055     056     ...
      057   058     067     068     069     070     071     072     073     074     083     084
            085     086     087     088     089     090     099     100     ...
      101   102     103     104     105     106     115     116     117     118     119     120
            121     122     131     132     133     134     135     136     ...
      137   138     146     147     148     149     150     151     152     153     154     162
            163     164     165     166     167     168     169     170     ...
      178   179     180     181     182     183     184     185     186     194     195     196
            197     198     199     200     201     202     210     211     ...
      212   213     214     215     216     217     218     225     226     227     228     229
            230     231     232     233     234     241     242     243     ...
      244   245     246     247     248     249     250     255     218     000     008     001
            001     000     000     063     000];

   % Start of Image (SOI) marker:FFD8=255,216
% JFIF marker:FFE0=255,224
%       Length=000,016
%       Identifier:4A46494600=074,070,073,070,000
%       Version=001,002
%       Units=000
%       Xdensity=000,096
%       Ydensity=000,096
%       Xthumbnail=000
%       Ythumbnail=000
%       (RGB)n, n=Xthumbnail*Ythumbnail required 3*n bytes=null


% Define Quantization table marker (luma):FFDB=255,219
%        Length:two bytes that indicate the number of bytes, including the two length bytes, that this header
contains=000,067
%        Precision=000 (baseline)
%        Quantization values=016,011,012,014,012,010,016,...,101,103,099 (zigzag)
% Define Quantization table marker (Chroma):FFDB=255,219
%        Length:two bytes that indicate the number of bytes, including the two length bytes, that this header
contains=000,067
%        Precision=000 (baseline)
%        Quantization values=017,018,018,024,021,024,047,...,099,099,099 (zigzag)

% Start of frame marker:FFC0=255,192
%        Length:two bytes that indicate the number of bytes, including the two length bytes, that this header
contains=000,011
%        Sample precision=008
%        X=000,000 (This will be defined later)
%        Y=000,000 (This will be defined later)
%        Number of components in the image=001
%            * 3 for color baseline
%            * 1 for grayscale baseline
%        Component ID=001
%        H and V sampling factors=034
%        Quantization table number=000

% Define Huffman table marker (DC):FFC4=255,196
%        Length:two bytes that indicate the number of bytes, including the two length bytes, that this header
contains=000,031
%        Index=000 (Huffman DC)
%        Bits=The next 16 bytes from an array of unsigned 1-byte integers whose elements give the number of Huffman
codes for each possible code length (1-16).
%        Huffman values=000,001,002,...,010,011

% Define Huffman table marker (AC):FFC4=255,196
%        Length:two bytes that indicate the number of bytes, including the two length bytes, that this header
contains=000,181
%        Index=016 (Huffman AC)
%        Bits=The next 16 bytes from an array of unsigned 1-byte integers whose elements give the number of Huffman
codes for each possible code length (1-16).
%        Huffman values=001,002,003,...,249,250

% Start of Scan marker:FFDA=255,218
%        Length:two bytes that indicate the number of bytes, including the two length bytes, that this header
contains=000,008
%        Number of components=001
%        Component ID=001
%        DC and AC table numbers=000
%        Ss=000
%        Se=063
%        Ah and Al=000

%% Define Size Image
Y = dec2hex(row,4);
X = dec2hex(col,4);
signal(1,164) = hex2dec(Y(1,1:2));
signal(1,165) = hex2dec(Y(1,3:4));
signal(1,166) = hex2dec(X(1,1:2));
signal(1,167) = hex2dec(X(1,3:4));

%% Concatenate Coding + Header
JP_STREAM(1,1:size(signal,2))=signal(1,1:size(signal,2));

for j=1:1:size(matrix_code_decimal,1)
    JP_STREAM(1,size(signal,2)+j)=matrix_code_decimal(j,1);
end

%% JPG Data Store
JP_STREAM=JP_STREAM';
fid = fopen(['compressed2.JPG'], 'wb');
if fid < 0
    error('Failed to open data file for write');
end
```

```
fwrite(fid,JP_STREAM,'uint8');
fclose(fid);
```

## 8.3.2 dct1.m

```
function manual_result = dct1(x)
        N = 8;
        manual_result = zeros(1,N);
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %   First Stage  10s
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        a=zeros(1,N);
        a(1)=x(1)+x(8);
        a(2)=x(2)+x(7);
        a(3)=x(3)+x(6);
        a(4)=x(4)+x(5);
        a(5)=x(4)-x(5);
        a(6)=x(3)-x(6);
        a(7)=x(2)-x(7);
        a(8)=x(1)-x(8);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %   Last Stage  5s
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        y=zeros(1,N);
    sb1=a(5)+a(8);
    sb2=a(8)-a(5);
        sb3=a(6)-a(7);
        sb4=a(6)+a(7);
        sb5=a(2)-a(3);
        sb6=a(1)-a(4);
        sb7=a(1)+a(4);
        sb8=a(2)+a(3);
        sb9=a(6)-a(5);
        sb10=a(6)-a(8);
        sb11=a(7)+a(8);
        sb12=a(5)-a(7);
        sb13=a(8)+a(8);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %   Reg
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % 30s

    y1_5_1=sb7+sb8;
        %y(2)=2^7*(sb1+sb4)+2^6*(sb2)-2^5*(sb1+sb3)+2^3*(sb1+sb2+sb3)+2^1*(sb1-sb4)-2^0*(sb2+sb3)


        y2_1_1=sb1+sb4;
    y2_1_2=sb1+sb3;
        y2_1_3=sb1-sb4;
        y2_1_4=sb2+sb3;
        y2_1_5=sb13+sb3;

        y2_2_1=2^7*y2_1_1;
        y2_2_2=2^6*(sb2);
        y2_2_3=2^5*y2_1_2;
        y2_2_4=2^3*y2_1_5;
        y2_2_5=2^1*y2_1_3;
        y2_2_6=2^0*y2_1_4;



        y2_3_1=y2_2_1+y2_2_2;
        y2_3_2=y2_2_3-y2_2_4;
        y2_3_3=y2_2_5-y2_2_6;
         %%%
     %%%
```

```
        y2_4_1=y2_3_1-y2_3_2;
        y2_5_1=y2_4_1+y2_3_3;

        %y(2)=y2_5_1;

        %y(3)=sb5*2^6-sb5*2^4+sb5+sb6*2^7-sb6*2^3-sb6*2^1

        y3_2_1=sb5*2^6;
y3_2_2=sb5*2^4;
        y3_2_3=sb6*2^7;
        y3_2_4=sb6*2^3;
        y3_2_5=sb6*2^1;

        y3_3_1=y3_2_1-y3_2_2;
        y3_3_2=y3_2_3-y3_2_4;
        y3_3_3=sb5-y3_2_5;

        y3_4_1=y3_3_1+y3_3_2;
        y3_5_1=y3_4_1+y3_3_3;

        %y(3)=y3_5_1;

        %y(4)=-sb10*2^7-a(5)*2^6-sb11*2^5+(sb11-a(5))*2^3+(a(6)+a(8))*2^1+sb12

        y4_1_1=sb11-a(5);
y4_1_2=a(6)+a(8);

        y4_2_1=sb10*2^7;
        y4_2_2=a(5)*2^6;
        y4_2_3=sb11*2^5;
        y4_2_4=y4_1_1*2^3;
        y4_2_5=y4_1_2*2^1;

        y4_3_1=y4_2_2+y4_2_1;
        y4_3_2=y4_2_4-y4_2_3;
        y4_3_3=y4_2_5+sb12;

        y4_4_1=y4_3_2-y4_3_1;
        y4_5_1=y4_4_1+y4_3_3;

        %y(4)=y4_5_1;



        % y5
        y5_5_1=(sb7-sb8);

        %y(6)= sb12*2^7+a(8)*2^6+sb9*2^5+(sb1-a(6))*2^3+(a(5)+a(7))*2^1+sb10

        y6_1_1=sb1-a(6);
y6_1_2=a(5)+a(7);

        y6_2_1=sb12*2^7;
        y6_2_2=a(8)*2^6;
        y6_2_3=sb9*2^5;
        y6_2_4=y6_1_1*2^3;
        y6_2_5= y6_1_2*2^1;
        y6_3_1=y6_2_1+y6_2_2;
        y6_3_2=y6_2_3+y6_2_4;
        y6_3_3=y6_2_5+sb10;



        y6_4_1=y6_3_1+y6_3_2;
        y6_5_1=y6_3_3+y6_4_1;
        %y(6)=y6_5_1;


        %y(7)=sb6*2^6-sb6*2^4+sb6-sb5*2^7+sb5*2^3+sb5*2

        y7_2_1=sb6*2^6;
        y7_2_2=sb6*2^4;
        y7_2_3=sb5*2^7;
        y7_2_4=sb5*2^3;
        y7_2_5=sb5*2^1;

        y7_3_1=y7_2_1-y7_2_2;
        y7_3_2=sb6-y7_2_3;
        y7_3_3=y7_2_4+y7_2_5;

        y7_4_1=y7_3_1+y7_3_2;
        y7_5_1=y7_4_1+y7_3_3;

        %y(7)=y7_5_1;

        %y(8)=sb9*2^7-a(7)*2^6-sb10*2^5+(sb10-a(7))*2^3+(a(6)+a(5))*2^1+sb11

        y8_1_1=sb10-a(7);
    y8_1_2=a(6)+a(5);

        y8_2_1=sb9*2^7;
        y8_2_2=a(7)*2^6;
        y8_2_3=sb10*2^5;
        y8_2_4=y8_1_1*2^3;
        y8_2_5=y8_1_2*2^1;

        y8_3_1=y8_2_1-y8_2_2;
        y8_3_2=y8_2_4-y8_2_3;
        y8_3_3=y8_2_5+sb11;

        y8_4_1=y8_3_1+y8_3_2;

        y8_5_1=y8_3_3+y8_4_1;

        %y(8)=y8_5_1;

        out_file = fopen('./output.txt','w');
        for i = 1:N
                fprintf(out_file,'%d\n',y(i));
        end
        y1_5_1;
        y1_6_1=floor(y1_5_1/4);        % 12:0
        y1_6_2=floor(y1_5_1/8);        % 12:0
        y1_7_1=y1_6_1+y1_6_2;   % 13 + 13
        y(1)=y1_7_1;                    % 14

        y2_6_1=floor(y2_5_1/512);      %
        y2_6_2=floor(y2_5_1/1024);     % 22 >> 10
        y2_7_1=y2_6_1+y2_6_2;   % 13 + 13
        y(2)=y2_7_1;                    % 14

        y(3)=floor(y3_5_1/256);
        y(4)=floor(y4_5_1/256);
        y5_6_1=floor(y5_5_1/4);
        y5_6_2=floor(y5_5_1/8);
        y5_7_1=y5_6_1+y5_6_2;
        y(5)=y5_7_1;
        y(6)=floor(y6_5_1/256);   % 27 >> 8 = 19 >> 5
```

```
		y(7)=floor(y7_5_1/256);
		y(8)=floor(y8_5_1/256);

		fclose(out_file);
		for i=1:N
				%manual_result(i)=y(i)*coeff(i);
				manual_result(i)=y(i);
		end
end

%{

function value = coeff(index)
		N = 8;
		w = [sqrt(1/N),sqrt(2/N)];
	switch (index)
				case {1,2,5}
						value = w(1);
				case {3,4,6,7,8}
						value = w(2);
		end
end
%}
```