Ryerson University
Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Systems-on-Chip Design
Course Number:	COE 838
Semester/Year (e.g.F2016)	W2022

Instructor:	Dr. Gul Khan

Assignment/Lab Number:	2b
Assignment/Lab Title:	JPEG Encoder/Decoder Unit

Submission Date:	Thursdsay, February 10th, 2022
Due Date:	Sunday, February 13th, 2022

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Pano	Toni	500822828	01	T. P.

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Contents

Introduction

This lab implements a JPEG encoder/decoder for a system on a chip by using SystemC. Its purpose is to compress an image and decompress an image using the JPEG format. This will make the decompressed image lose some information, but ideally it should retain most of the important features of the original image.

The hardware/software codesign method used specifies that the Discrete Cosine Transform and Inverse Discrete Cosine Transform are the program components that will have the greatest impact on improving speed if implemented as dedicated hardware modules. Therefore those units are implemented as SystemC modules.

1. Compression Procedure

I. Read 64 consecutive bytes from input file for image.
II. Store bytes in an 8 by 8 array.
III. Shift numbers from unsigned range (0 to 2^8-1) to signed range (-2^7 to 2^7-1).
IV. Perform the Discrete Cosine Transform on the array.
V. Quantize array values.
VI. Order array values in a “zigzag” sequence.
VII. Write 64 consecutive bytes to output file for compressed data.
VIII. Repeat steps I to VII until no more data can be read from input image file.

2. Decompression Procedure

I. Read 64 consecutive bytes from input file for compressed data.
II. Store bytes in an 8 by 8 array.
III. Undo “zigzag” order of bytes.
IV. Undo the quantization of array values.
V. Perform the Inverse Discrete Cosine Transform on the array.
VI. Shift numbers from signed range (-2^7 to 2^7-1) to unsigned range (0 to 2^8-1).
VII. Write 64 consecutive bytes to output file for image.
VIII. Repeat steps I to VII until no more data can be read from input compressed data file.

3. Results

Figure 1. Left: Original image file. Right: Image file after being compressed and decompressed.

Conclusion

The final image of the file after compression and decompression preserves the outline of major features in the original image, but does not preserve small details. The brightness of each color in the original image also appears to be reversed in the final image, and pink (null) spots appear too. Either the compression and decompression algorithms have a systematic error, or my implementation of the algorithms are incorrect. The final image appears to be split into 1 row by 64 column rectangles. This implies that the image was written to memory sequentially. This is due to the data being extracted from the original image sequentially. In order to reduce artifacts, more data could be read at once before performing compression stages I to VII. Alternatively, the data could be completely read, then divided into square segments instead of thin rectangles.

Appendix A: Modified Code

Makefile

```
#
CC=/usr/bin/g++
ARCH := $(shell arch)
SYSTEMC_HOME=/usr/local/SystemC-2.3.0

# 64bit or 32bit libaries to link to
LINUXLIB := $(shell if [ ${ARCH} = "i686" ]; \
                    then \
                         echo lib-linux; \
                    else \
                         echo lib-linux64; \
                    fi)

INCLUDES = -I$(SYSTEMC_HOME)/include -I.

LIBRARIES = -L. -L$(SYSTEMC_HOME)/$(LINUXLIB) -lsystemc -lm

RPATH = -Wl,-rpath=$(SYSTEMC_HOME)/$(LINUXLIB)

PROGRAM = sc_jpeg.x
SRCS    = functions.h functions.cpp fdct.h fdct.cpp idct.h idct.cpp sc_main.cpp
OBJS    = functions.o fdct.o  idct.o  sc_main.o

all : $(PROGRAM)

$(OBJS) : $(SRCS)
	$(CC) $(INCLUDES) -c $(SRCS)

$(PROGRAM) : $(OBJS)
	$(CC) $(INCLUDES) $(LIBRARIES) $(RPATH) -o $(PROGRAM) $(OBJS)

clean:
	@rm -f $(OBJS) $(PROGRAM)
```

sc_main.cpp

```
#include "systemc.h"
#include "functions.h"
#include "fdct.h"
//#include inverse module
#include "idct.h"

#define NS *1e-9 // use this constant to make sure that the clock signal is in nanoseconds

int sc_main(int argc, char *argv[]) {
	char choice;
	sc_signal<FILE *> sc_input;   // input file pointer signal
	sc_signal<FILE *> sc_output;  // output file pointer signal
	sc_signal<double> dct_data[8][8]; // signal to the dc transformed data
	sc_signal<double> cosine_tbl[8][8]; // signal for the cosine table values

	sc_signal<bool> clk1, clk2;           // clock signal for FDCT (also need a 2nd clock for
IDCT)

	FILE *input, *output; // input and output file pointers
	double cosine[8][8]; // the cosine table
	double data[8][8]; // the data read from the signals, which will be zigzagged

	//Decompression signals and variables
	int i, j;
	sc_signal<double> trans64[8][8];


	if (argc == 4) {
		if (!(input = fopen(argv[1], "rb"))) // error occurred while trying to open file
			printf("\nSystemC JPEG LAB:\ncannot open file '%s'\n", argv[1]), exit(1);

		if (!(output = fopen(argv[2], "wb"))) // rror occurred while trying to create file
			printf("\nSystemC JPEG LAB:\ncannot create file '%s'\n", argv[2]),
exit(1);

		// copy the input and output file pointer onto the respective ports
		sc_input.write(input);
		sc_output.write(output);

		choice = *argv[3];
	} else
		fprintf(stderr, "\nSystemC JPEG LAB: insufficient command line arguments\n"
			"usage:  ./sc_jpeg.x [input file] [output file] [(C)ompress or
(D)ecompress]\n")
				, exit(1);

	 // write the header, read from the input file
	write_read_header(input, output);

	 // make the cosine table
	make_cosine_tbl(cosine);

	// call the forward discrete transform module
	fdct fdct("fdct");
	// and bind the ports -FDCT
	fdct.clk(clk1);
	fdct.sc_input(sc_input);
	for(i = 0; i < 8; i++){
		for(j = 0; j < 8; j++){
			fdct.fcosine[i][j](cosine_tbl[i][j]);
			fdct.out64[i][j](trans64[i][j]);
		}
	}

	// copy the cosine table and the quantization table onto the corresponding signals to
send to DCT module
	for(i = 0; i < 8; i++){
		for(j = 0; j < 8; j++){
			cosine_tbl[i][j].write(cosine[i][j]);
```

```
                }
        }

        //binds ports - idct
        idct idct("idct");
        idct.clk(clk2);
        idct.sc_output(sc_output);
        for(i = 0; i < 8; i++){
                for(j = 0; j < 8; j++){
                        idct.fcosine[i][j](cosine_tbl[i][j]);
                        idct.in64[i][j](trans64[i][j]);
                }
        }

        // because compression and decompression are two different processes, we must use
        // two different clocks, to make sure that when we want to compress, we only compress
        // and dont decompress by mistake
        sc_start(SC_ZERO_TIME);       // initialize the clock
        if ((choice == 'c') || (choice == 'C')) { // for compression
                while (!(feof(input))) { // cycle the clock for as long as there is something to
be read from the input file

                        // create the FDCT clock signal
                        clk1.write(1);         // convert the clock to high
                        sc_start(10, SC_NS);   // cycle the high for 10 nanoseconds
                        clk1.write(0);         // start the clock as low
                        sc_start(10, SC_NS);   // cycle the low for 10 nanoseconds

                        //fdct.read_data();
                        //fdct.calculate_dct();

                        // read back signals from module
                        for(i = 0; i < 8; i++){
                                for(j = 0; j < 8; j++){
                                        data[i][j] = trans64[i][j].read();
                                }
                        }

                        // zigzag and quantize the outputted data - will write out to file (see
functions.h)
                        zigzag_quant(data, sc_output);


                }
        } else if ((choice == 'd') || (choice == 'D')) { // for decompression
                while (!(feof(input))) {
                        //unzigzag and inverse quatize input file and result will be placed in
data
                        unzigzag_iquant(data, input);

                        //write unzigzag data to ports
                        for(i = 0; i < 8; i++){
                                for(j = 0; j < 8; j++){
                                        trans64[i][j].write(data[i][j]);
                                }
                        }

                        clk2.write(1);         // convert the clock to high
                        sc_start(10, SC_NS);   // cycle the high for 10 nanoseconds
                        clk2.write(0);         // start the clock as low
                        sc_start(10, SC_NS);   // cycle the low for 10 nanoseconds

                         //read idct data from ports & write to output file
                }

        }
        fclose(sc_input.read());
        fclose(sc_output.read());

        return 0;
}
```

idct.h

```
#include "systemc.h"
#include <math.h>

#ifndef IDCT_H
#define IDCT_H

struct idct : sc_module{
	sc_in<double> in64[8][8]; //the dc transformed 8x8 block
	sc_in<double> fcosine[8][8]; //cosine table input
	sc_out<FILE *> sc_output; //output file pointer port
	sc_in<bool> clk; //clock signal

	char output_data[8][8]; //the data to write to the output file

	void write_data( void ); //write the transformed 8x8 block
	void calculate_idct(void ); //perform inverse discrete cosine transform

	//define idct as constructor
	SC_CTOR( idct ){
		SC_METHOD( calculate_idct );
		dont_initialize();
		sensitive << clk.pos();

		SC_METHOD( write_data );
		dont_initialize();
		sensitive << clk.neg();
	}
};

#endif
```

idct.cpp

```
#include "idct.h"

//inverse discrete cosine transform
void idct :: calculate_idct(void){
	unsigned char  u, v, x, y;
	double  temp;

	// do forward discrete cosine transform
	for (x = 0; x < 8; x++)
		for (y = 0; y < 8; y++) {
			temp = 0.0;
			for (u = 0; u < 8; u++)
				for (v = 0; v < 8; v++)
					temp += in64[u][v].read() * fcosine[x][u].read() *
fcosine[y][v].read();

					if ((u == 0) && (v == 0))
						temp /= 8.0;
					else if (((u == 0) && (v != 0)) || ((u != 0) && (v == 0)))
						temp /= (4.0*sqrt(2.0));
					else
						temp /= 4.0;

			output_data[x][y] = temp;
		}

	printf(".");
}

//read 8x8 block and shift signed integers
void idct :: write_data(void){
	// shift the signed integers to the unsigned integer range of [0, 2^8 - 1]
	// of range [2^(8-1), 2^(8-1) - 1]
	for (unsigned char uv = 0; uv < 64; uv++)
		output_data[uv/8][uv%8] += (char) (pow(2,8-1));
	// read the 8x8 block as a continuous 64 values and store them in
	// input_data as an 8x8 block
	fwrite(output_data, 1, 64, sc_output.read());
}
```

Appendix B: Unmodified Code

fdct.h

```
#include "systemc.h"
#include <math.h>

#ifndef FDCT_H
#define FDCT_H

struct fdct : sc_module {
	sc_out<double> out64[8][8]; // the dc transformed 8x8 block
	sc_in<double> fcosine[8][8]; // cosine table input
	sc_in<FILE *> sc_input; // input file pointer port
	sc_in<bool> clk; // clock signal

	char input_data[8][8]; // the data read from the input file

	void read_data( void ); // read the 8x8 block
	void calculate_dct( void ); // perform dc transform

	// define fdct as a constructor
	SC_CTOR( fdct ) {
		// make read_data method sensitive to the positive clock edge, and
		// the calculate_dct method sensitive to the negative clock edge
		// this way, the entire read and performing dct takes only one clock cycle
		// as apposed to two
		SC_METHOD( read_data ); // define read_data as a method
		dont_initialize();
		sensitive << clk.pos();

		SC_METHOD( calculate_dct ); // define calculate_dct as a method
		dont_initialize();
		sensitive << clk.neg();
	}
};

#endif
```

fdct.cpp

```
#include "fdct.h"

void fdct :: calculate_dct( void ) {
	unsigned char  u, v, x, y;
	double  temp;

	// do forward discrete cosine transform
	for (u = 0; u < 8; u++)
		for (v = 0; v < 8; v++) {
			temp = 0.0;
			for (x = 0; x < 8; x++)
				for (y = 0; y < 8; y++)
					temp += input_data[x][y] * fcosine[x][u].read() *
fcosine[y][v].read();
			if ((u == 0) && (v == 0))
				temp /= 8.0;
			else if (((u == 0) && (v != 0)) || ((u != 0) && (v == 0)))
				temp /= (4.0*sqrt(2.0));
			else
				temp /= 4.0;
			out64[u][v].write(temp);
		}

	printf(".");
}

void fdct :: read_data( void ) {
	// read the 8x8 block as a continuous 64 values and store them in
	// input_data as an 8x8 block
	fread(input_data, 1, 64, sc_input.read());

	// shift the unsigned integers from range [0, 2^8 - 1] to signed integers
	// of range [2^(8-1), 2^(8-1) - 1]
	for (unsigned char uv = 0; uv < 64; uv++)
		input_data[uv/8][uv%8] -= (char) (pow(2,8-1));
}
```

function.h

```
#include <stdio.h>
#include <math.h>

// read the header of the bitmap and write it to the output file
void write_read_header(FILE *in, FILE *out);

// make the cosine table
void make_cosine_tbl(double cosine[8][8]);

// zigzag the quantized input data
void zigzag_quant(double data[8][8], FILE *output);

// unzigzag the zigzagged input data
void unzigzag_iquant(double data[8][8], FILE *input);
```

function.cpp

```
#include "functions.h"

#define rnd(x) (((x)>=0)?((signed char)((signed char)((x)+1.5)-1)):((signed char)((signed
char)((x)-1.5)+1)))
#define rnd2(x) (((x)>=0)?((short int)((short int)((x)+1.5)-1)):((short int)((short int)((x)-
1.5)+1)))
#define PI 3.14159265358979323846264338327950 // the value of PI

// the end of block marker, something which is highly unlikely to be found in a dct block
signed char MARKER = 127;

// quantization table
unsigned char quant[8][8] =
               {{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}};

// zigzag table
unsigned char zigzag_tbl[64] = {
               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};


void write_read_header(FILE *in, FILE *out) {
       unsigned char temp[60]; // temporary array of 60 characters, which is enough
                                                     // for the bitmap header, which is 54 bytes
       printf("\nInput Header read and written to the output file");
       fread(temp, 1, 54, in);       // read 54 bytes from the input file and store them in temp
       fwrite(temp, 1, 54, out);     // write the 54 bytes to the output file
       printf("......Done\n");
       printf("Image is a %d bit Image. Press Enter to Continue\n>", temp[28]);
       getchar();
}


void make_cosine_tbl(double cosine[8][8]) {
       printf("Creating the cosine table for use in FDCT and IDCT");
       // calculate the cosine table as defined in the formula
       for (unsigned char i = 0; i < 8; i++)
              for (unsigned char j = 0; j < 8; j++)
                     cosine[i][j] = cos((((2*i)+1)*j*PI)/16);
       printf("......Done\n");
}


void zigzag_quant(double data[8][8], FILE *output) {
       signed char to_write[8][8], final_write[8][8]; // this is the rounded values, to be
written to the file
       char last_non_zero_value = 0; // this is the index to the last non-zero value in a block

       //quantization calculation with rounding (rnd) to nearest integer
       for (unsigned char i = 0; i < 8; i++) {
              for(unsigned char j = 0; j < 8; j++){
                     to_write[i][j] = rnd(data[i][j] / quant[i][j]);
              }
       }

       // zigzag the data array and copy it back to final_write array
```

```
        //find out the index to the last non-zero value in the 8x8 block
        for (unsigned char i = 0; i < 64; i++) {
                final_write[zigzag_tbl[i]/8][zigzag_tbl[i]%8] = to_write[i/8][i%8];
                        if (final_write[i/8][i%8] != 0)
                                last_non_zero_value = i;
        }

        // write all the values in the block up to and including the last non-zero value
        for (unsigned char i = 0; i <= last_non_zero_value; i++)
                fwrite(&final_write[i/8][i%8], sizeof(signed char), 1, output);

        // write the end of block marker
        fwrite(&MARKER, sizeof(signed char), 1, output);
}


void unzigzag_iquant(double data[8][8], FILE *input) {
        signed char to_read[8][8]; // this is the data just read from the input file
        signed char temp_value = 0; // this is just the temporary value being read from the file

        // set all the values in the array to zeroes
        for (unsigned char i = 0; i < 64; i++)
                to_read[i/8][i%8] = 0;

        // read from file byte by byte, compare with the marker, and if it is not the marker
        // and it is not the end of file, read the next byte and store it in the array
        // keep doing this until the end of block is reached
        fread(&temp_value, sizeof(signed char), 1, input);

        for (unsigned char i = 0; (!(feof(input))) && temp_value != MARKER;
                        i++, fread(&temp_value, sizeof(signed char), 1, input))
                                to_read[i/8][i%8] = temp_value;

        // unzigzag the temporary array and copy it back to data
        for (unsigned char i = 0; i < 64; i++)
                data[i/8][i%8] = (double) to_read[zigzag_tbl[i]/8][zigzag_tbl[i]%8] *
quant[i/8][i%8];
}
```