CSEE 4840 Project Design: FPGA JPEG Compression Accelerator

Xinyi Chang(xc2323), Yuxiang Chen(yc3096), Song Wang(sw2996), Nan Zhao(nz2250)
Spring 2016
Electrical Engineering, Columbia University

I. Introduction

When technology rapidly evolves, computing architectures must be very flexible and easy to upgrade. FPGAs are considered as an very attractive solution for image processing implementation, not only because of it integrates millions of gate and a large number of internal memory banks, but also it's fast-to-market, low cost, and high performance. In this project, we will build specialized hardware to accelerate the JPEG image compression process taking the advantages of FPGA characteristic.

As shown in figure 1.1, there are four main steps in JPEG compression: divide an image into 8-pixel by 8-pixel blocks, DCT on image blocks, quantization, and inverse DCT. We will build a dedicated DCT, quantization and inverse DCT processor on the FPGA board, in hopes of speeding up the compression process when compared to running solely on the ARM CPU core.

To implement this accelerator, image pre-processing will be done in software side. It includes reading image data from file, interpret image data and divide image into 8-pixels by 8-pixels blocks, and sending pixels to the hardware. Then FPGA will handle the heavy image processing and send back the processed data using Avalon bus. Figure 2.1 shows the high-level block diagram of the entire image compression. In the end, we expect a compressed image date file stored in Linux end.

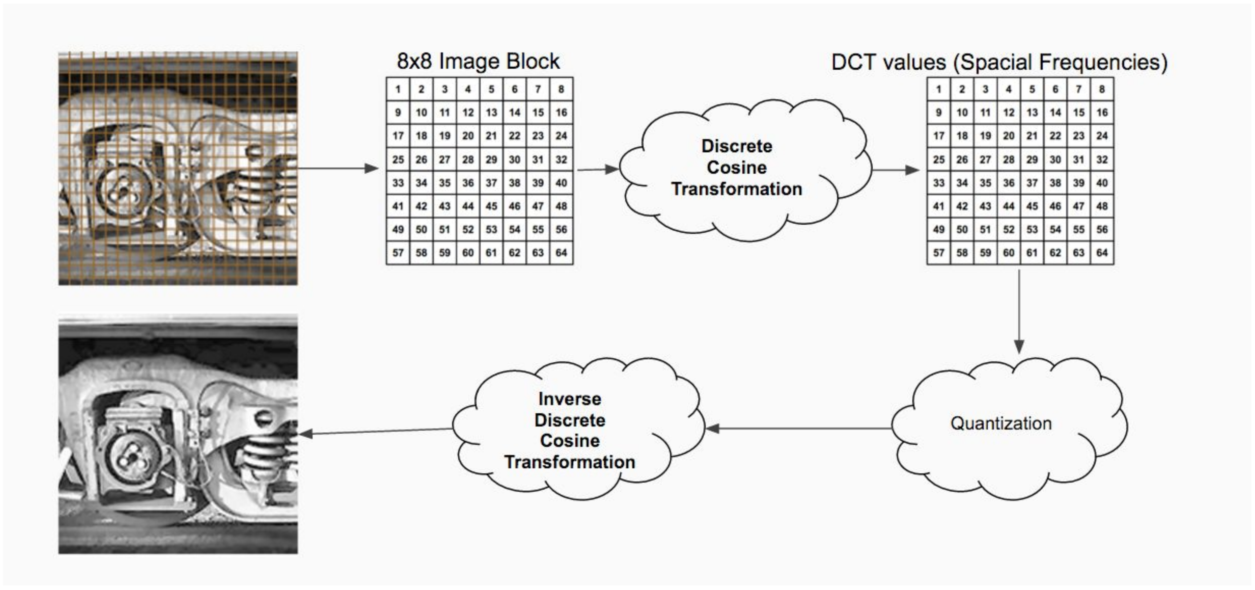
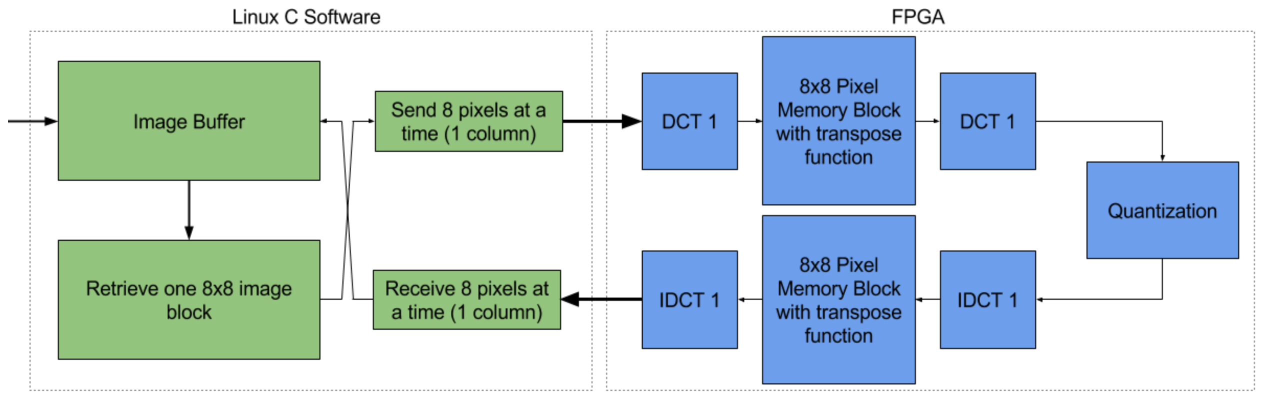

8x8 Image Block
DCT values (Spacial Frequencies)
Discrete Cosine Transformation
Quantization
Inverse Discrete Cosine Transformation

Figure 1.1 JPEG Image Compression Flow



Linux C Software
FPGA
Image Buffer
Send 8 pixels at a time (1 column)
Retrieve one 8x8 image block
Receive 8 pixels at a time (1 column)
DCT 1
8x8 Pixel Memory Block with transpose function
DCT 1
Quantization
IDCT 1
8x8 Pixel Memory Block with transpose function
IDCT 1

Figure 1.2 Block Diagram

II. Implementation

Software

Stage I. Picture Conversion

Due to the fact that c program does not have libraries to handle pictures, we need to first use Matlab to convert given pictures into matrices of ARGB value and then analysis the data with c.
The pseudo code below is our intended code algorithm for implementing such picture conversion functions.

Input: Image file Output:Matrix of the Image file in ARGB values

Pseudo Code:(Matlab)

```
Load_image(*.jpg)
Fetch parameters...Height, Width, Resolution
Fetch pixel values...A,R,G,B
Store above values into a .txt file.
Close_image.
```

Stage II. Matrix Processing

In our case, we are not processing the picture entirely. Our idea is to first convert each pixel into a grayscale level data and then divide the whole picture into multiple 8*8 blocks. The blocks will end up be sent to FPGA module.

Input:Matrix File Output: 8*8 Blocks and index info

Pseudo Code:(C)

```
Read_file(image.txt)
```

```
Fetch_parameters(image.txt)
Calculate # of blocks {return ceiling(width/8) * ceiling(height/8)}
Estalbish_index_info{
  Line_1_bl_1  …… line_1_bl_n
                 :
                  :
                  :
                  :
  Line_m_bl_1 ……… line_m_bl_m
}
For( i=0;i<index#;i++){
        q=fetch_pixel_data(i)
        q=update_grayscale(q)
        send_to_FPGA(q,i)
}
```

Stage III. Data Collection

After FPGA finishes the transformation of data blocks, we need to collect them from communication bus and combine them into a single .txt file. Pseudo code is for the algorithm we will use of data combination.

Input:8*8 block data, index info   Output: Single .txt file

Pseudo Code:(C)

```
Index = 1;
Open_file(filename.txt)

while(buffer){
Data = Receieve_data(buffer)
Index_ = Data.index;
Matrix = Data.matrix;
If ( index!=index_ ){ error;}
Else {
        write_to_file(filename.txt, Matrix)
}
}
```

Hardware

1. **DCT**

   The implementation of DCT is on FPGA and is using systemVerilog. The N-point DCT is defined as Figure 2.1. Our DCT optimization focuses on reducing the number of required arithmetic operations, especially the number of multiplications. By using Loeffler Algorithm, the number of multiplications in DCT can reach the theoretical low limit. Loeffler Algorithm

proposed to compute DCT outputs on four stages as shown in Figure 2.2, and each stage contains some arithmetic operations.
For the further optimization, we use CSD representation for those coefficients in multiplications. CSD is a signed representation which contains the fewest number of nonzero bits. With constant number multiplications in DCT, the number of additions and subtractions will be minimized. For any multiplication of 2^N we can use shifter instead of multiplier.

$$y(k) = w(k)\sum_{n=1}^{N} x(n)\cos\left(\frac{\pi}{2N}(2n-1)(k-1)\right), \quad k = 1, 2, \dots, N, \qquad w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1, \\ \sqrt{\frac{2}{N}}, & 2 \le k \le N, \end{cases}$$

Figure 2.1 N-point DCT formula

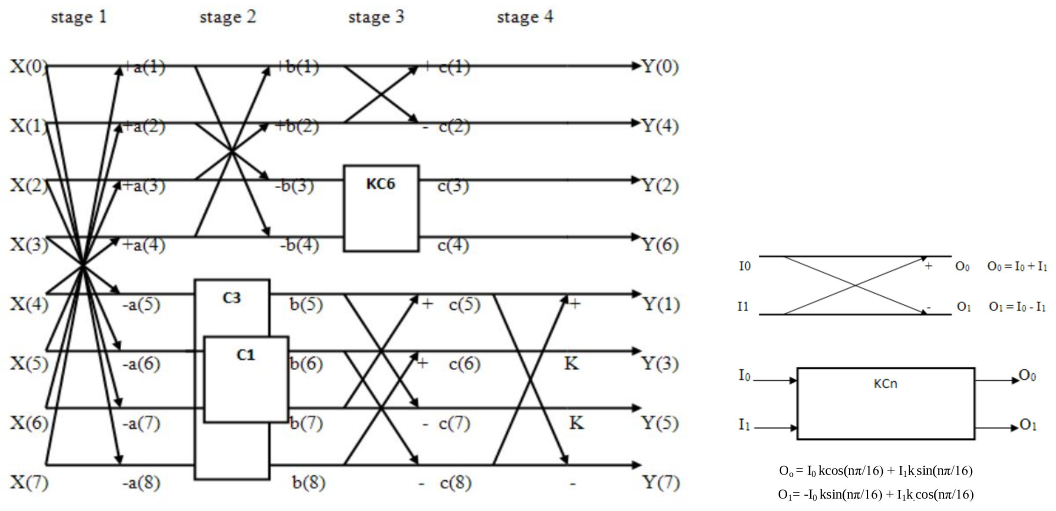


Figure 2.2 Loeffler architecture of 8-point DCT algorithm

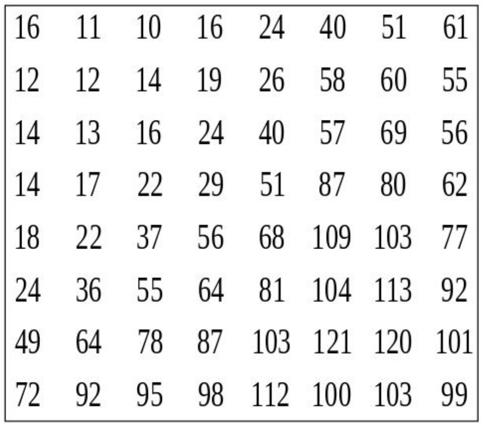
Real value	Decimal	Natural binary	Partial products	CSD	Partial products
$cos\frac{3\pi}{16}$	106	01101010	4	+0-0+0+0	4
$sin\frac{3\pi}{16}$	71	01000111	4	0+00+00-	3
$cos\frac{\pi}{16}$	126	01111110	6	+00000-0	2
$sin\frac{\pi}{16}$	25	00011001	3	00+0-00+	3
$cos\frac{6\pi}{16}$	49	00110001	3	0+0-000+	3
$sin\frac{6\pi}{16}$	118	01110110	5	+000-0-0	3
$\sqrt{(2)}$	181	10110101	5	+0-0-0+0+	5
Total Partial products		30		23	

Figure 2.3 8-Point DCT Fixed Coefficient Representation

2. **Memory Block Transpose**
   We plan to use register files to store the 8*8 blocks row by row after the first 1-DCT operation, and read the datas from register file column by column to achieve the transposing purpose before datas are popped into 1-DCT module again.
3. **Quantization**

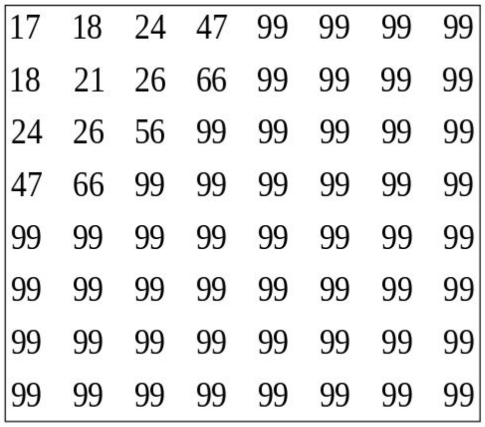
Quantization is the part of the process that actually allows for compression. The quantization matrix can be altered to create an acceptable balance between image quality and compression ratio. Once quantization has occurred, the data are encoded to a bit-stream in which form they are stored or
transported.
The quantization operation is an integer division of the 2D DCT coefficients by pre-defined values. These pre-defined values are stored in tables called quantization tables. In JPEG baseline mode there are two quantization tables. One for luminance components and the other for chrominance components. There are the recommended table as shown below:

- Luminance



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	36	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

- Chrominance, subsampled 2:1



17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

4. **Inverse DCT**
The definition of IDCT shows in Figure 2.4. Due to the limit of time, we implement the IDCT only based on the IDCT formula。

$$x(n) = \sum_{k=1}^{N} w(k)\, y(k) \cos\left(\tfrac{\pi(2n-1)(k-1)}{2N}\right), \quad n = 1, 2, \ldots N \qquad w(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 1 \\ \sqrt{\frac{2}{N}}, & 2 \le k \le N \end{cases}$$

Figure 2.4 N-point DCT formula

III. Milestone

Milestone 1:

1. JPEG Compression in Matlab Implementation
2. DCT design in SystemVerilog
3. Image file generated from Matlab.

Milestone 2:

1. Image data dividing in C
2. Memory blocks with transpose function
3. Inverse DCT design in SystemVerilog

Milestone 3:

1. Quantization function in SystemVerilog
2. Avalon Bus setup.
3. Final assembling.

IV. 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.