EE4292 IC Design Laboratory, Fall 2021

Final Project

JPEG

Team member: 107061105 羅宇辰、107061234 張博閔、107061252 吳書磊

I. Functionality

In our project, we implement **JPEG encoder** and most of **JPEG decoder** in Verilog RTL, and the left of the decoder parts (**upsample** and **YUV to RGB**) are implement by software.

1. JPEG Encode

i. RGB to YUV:

Transform the color space from **RGB** space to **YUV(YCbCr)** space.

ii. Chroma Subsampling and zero padding:

We provide two subsample mode, 4:1:1 and 4:2:0. Since chrominance is not sensitive to our human eyes, we can decrease its storage by sampling. In addition, if width or height of subsampled image is not the multiplies of 8, it should apply zero padding.

iii. 2D – DCT:

Divide image into several blocks, each contain 8x8 pixels. For each block, we apply **Discrete Cosine Transform** to convert each value of pixel to its frequency domain.

iv. Quantization:

For each block of **DCT** output, we divide the value by a given matrix (luminance and quantization table), eliminating the high spatial frequency value.

v. Entropy Encoding:

For each block, we apply **DPCM(Differential Pulse Code Modulation)** for encoding of DC value and apply **RLC(Run-Length-Coding)** for encoding of AC value. Then we use **4 Huffuman LUT** (combination of DC/AC and luminance/chrominance) to apply **Huffuman encode**.

2. JPEG Decode

i. Decode, De-quantize, 2-D IDCT

The inverse operation of the encode part. Use **4 Huffuman LUT** to find corresponding **DPCM** and **RLC** encoding then reconstruct each 8x8 pixels. Then multiply the value of each pixel by given table, and apply **2D-Inverse Discrete Cosine Transform** to obtain value of each pixel of Y, U and V.

II. Specification

Expected performance and post-sim result with are listed in **Table 1**.

	Expected	**Post-sim**
Timing	3 ns (333MHz)	3.2 ns (312.5MHz)
Area	300000μm^2	130941μm^2
Power	0.2 mW	12.2 mW
Compression ratio	-	> 10:1

Throughput(encode)	10Mpixel/sec	150Mpixels/sec
Throughput(decode)*	-	75Mpixels/sec
P&R Core utilization	-	0.8

Table 1. Performance Index

*Since decode part is partially done by software and the hardware architecture have not optimize due to limit time, throughput of this part is only for reference.

III. Implementation

Block diagram:

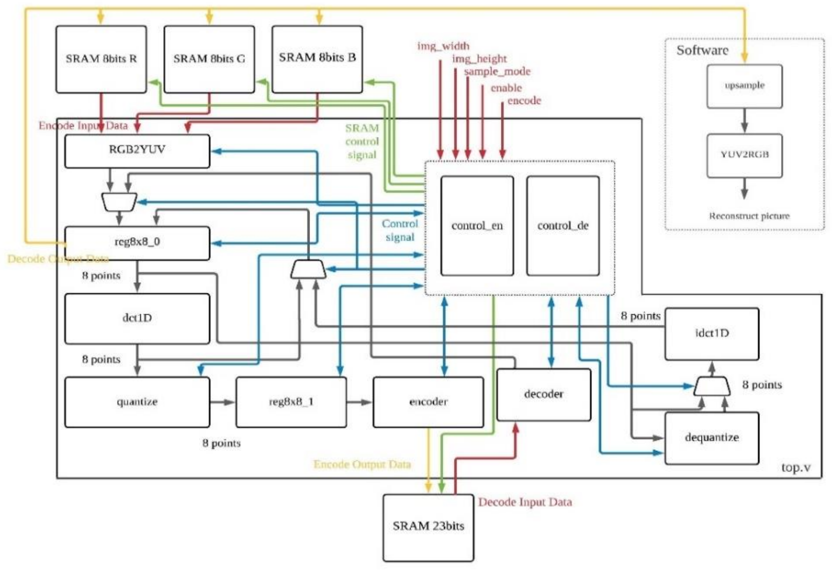

Figure 1. block diagram

In our design, we use 8x8 register file *reg8x8* for temporary storage. Two *reg8x8* for the encode part in order to increase the hardware usage and parallelism, one is shared with the decode part. And our design is serial for encode of Y, U, and V. This can reduce the hardware resource as we need 3x computing module to support parallel processing for Y, U, and V.

Next, we will describe the design flow in detailed as follow. In addition, we handle the following design flow for each 8x8 pixels instead of whole image.

JPEG Encode:

1. **RGB to YUV**

   In this step, we process a single pixel in a cycle due to the SRAM read bandwidth. We convert all the fraction coefficient in this project into 8-bit fixed point format with 7 bits floating point length. After some multiplications and additions, we will apply rounding to each pixel.

2. **Chroma Subsampling and zero padding**

   We implement chroma subsampling by reading the correct SRAM address corresponding to the sample mode. In this case, extra read cycles can be saved.

3. **2D – DCT**

   For each 8x8 pixels, we doesn't apply 2D – DCT directly. In detail, we apply 1D – DCT for each row and then apply 1D – DCT for each column to accomplish 2D – DCT function and store the output value into *reg8x8*, and rounding is applied after an 1D – DCT. **Figure 2.** and **Figure 3.** show hardware architecture[1] and the sketch of this step. By applying this hardware architecture, we can share the multiplier to compute an 8-point 1D-DCT by only 16 multipliers and 26 adders. Since the first 1D-DCT of whole 8x8 matrix need to be finished than the second 1D-IDCT can be execute, we can't pipeline the procedure of two 1D-DCT, so we only need one 1D-DCT module to implement 2D-DCT.

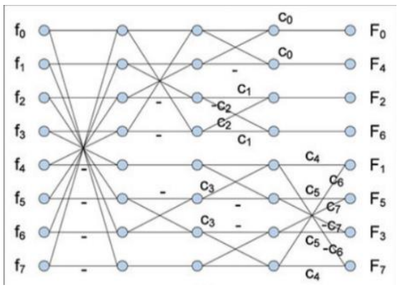

Figure 2. 1D – DCT architecture

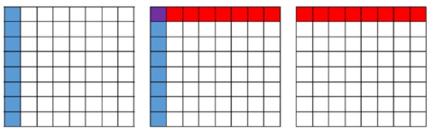

Figure 3. sketch of 2D - DCT

4. **Quantization**

The way we implement divide computation is to multiply the 2D-DCT output by the inverse value of quantization table.

5. **Entropy Encoding (Encoder)**

To perform RLC, traverse through the *reg8x8* in JPEG zigzag order cycle by cycle. Reaching non-zero value, 15 consecutive zero, or the end of the block, Huffuman encode is implemented by LUT and use enable signal to indicate the valid output. Also, DC value of an 8x8 block is stored to perform DPCM.

JPEG Decode:

1. **Decoder**

Since each Huffuman code may have different bit length, we sort the Huffuman encode table by the bit length and inverse it to be the Huffuman decode table. To decode a Huffuman code, the hardware lookup for the result from different bit length Huffuman decode LUTs, there exist the only one output that is valid, and the LUT output can be used for filling up the *reg8x8*.

2. **Dequantize and 2D-IDCT**

Similar as quantize and 2D-DCT part. And we also use the Chen algorithm to implement 1D-IDCT. Since the architecture is a little different with 1D-DCT, we didn't share their hardware resource.

IV. Verification

In order to verify our RTL design, we use **Python** to write two software code, *final_encode.ipynb* and *final_decode.ipynb*, which generate the test pattern and golden data. We also write the **Testbench** to verify our RTL design, feeding input image into a SRAM and input the data from SRAM to RTL, then store the encode data to another SRAM and verify if the encode method is correct between the output code and golden. Then read the encode data from SRAM in order to decode them, after decoding then store decode data into the other SRAM and verify if the decode method is correct. Also, we have verified our intermediate work (RGB2YUV, DCT, quantize, dequantize, IDCT) while finish each module. The following show the way we verify the whole chip.

1. **Encode:** When SRAM write enable is low, check the output code correctness of our encode method.
2. **Decode:** Check the value of reconstructed YUV matrix in SRAM is correct.

After Reconstructed Y, U, V matrix, we use **Python** to do "**upsample**" and "**YUV to RGB**" function to reconstruct the image.

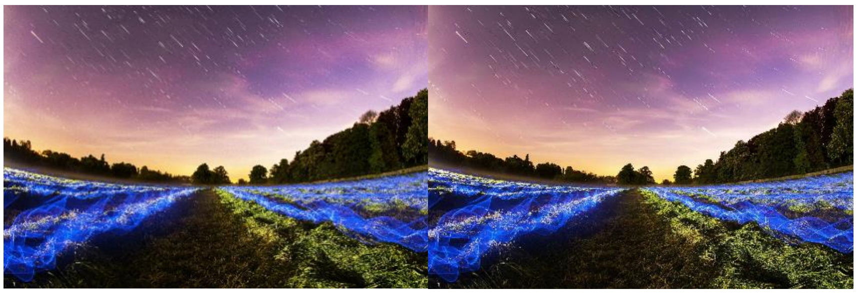

Figure 5. Image Comparison (Left: raw image, right: reconstructed image.)

V. Discussion

1. Discussion of Memory Usage

In our implementation, we implement a **3-bank SRAM group** to store the RGB channels of image. The reason why we interleave banks is that we can access 3 banks parallelly by sending 3 SRAM address to 3 banks respectively. As a consequence, we can process a pixel in **one cycle** in RGB2YUV block.

Moreover, in order to minimize the frequency of accessing SRAM, we use 8x8 register buffer to store our data during encode/decode process. Before computing one function block, we need to take the data from buffer, and put the data back into buffer after finish computing, therefore reducing SRAM access.

2. Optimization of Encoder

We found that in encode process, there are not related in time among all modules, it means that it will cause a huge inefficiency when we implement the encoder. Therefore, we can apply **pipeline processing** during encode process: we divided RGB2YUV block and one 1D-DCT block into stage 1; the other 1D-DCT block, quantization block and entropy encoding block as stage 2. When one 8x8 pixels in stage 1 is complete, then it will be moved to stage 2, then stage 1 can keep encoding next 8x8 pixels, thus improve the hardware usage significantly.

Consequently, since stage 1 and stage 2 are both only need ½ **cycles counts of original encoder,** we can increase the throughput to **2 times** of throughput of encoder without parallel processing with the cost of **one more 8x8 register**.

3. Discussion of Hardware Resource and Precision

Since we round the output value of every step (RGB2YUV, 1D-DCT, etc.) and we only use 8-bit to represent the coefficients, the multiplier in our design may not be such large, this reduce the area significantly. Although frequently rounding may lead to degradation, we still can reach the PSNR of about **30dB** in our design, so we think the quality is acceptable. We also can raise the precision by modify the parameter of our parameterized module.

4. **Pipelining**

In order to improve the operating frequency of the chip, we **pipelining all sub-modules** (4 stages 1D-DCT, 2stages RGB2YUV, etc.) and cut the critical path to about **a multiplier plus an adder**. Finally, the critical path is 3.2ns.

VI. Conclusion and Future Work

In this final project, we implement JPEG encoder and most of JPEG decoder which supporting two different chroma subsample mode. With the parallel processing and optimized memory usage, we can improve the total cycle count of the encoding process, and the area of our RTL design.

After P&R and post-simulation, our ASIC design can reach the **throughput** (450MB/sec) that higher than the *Intel IPP-7.0 + Core i7 920* (430MB/sec)[2] in the encode part. Since we don't have time to optimize the decode part, the throughput of the decode part is the half of the encode part. Additionally, the **area** (130941μm^2) of our chip is relatively small due to the serial dataflow and low bit multiplier in the design, but we still can reach the PSNR of about **30dB**.

In the future work, we can add the header and encode buffer in the encode part and finish the remaining part of decode, optimize the architecture and dataflow of decode part to halve the throughput, and the extra 8x8 register file can be shared with encode part. Moreover, the architecture of 1D-DCT and 1D-IDCT in our current design is different and they account for the largest portion in the total area (12.3% and 20.2%), we may try to explore an architecture that can share their hardware resource. The last, since our current design can't handle the bit error in the bit-stream while decoding, which may cause the error in Huffuman decode lookup. We can directly detect that there are bit error in the bit-stream if we can't obtain a reasonable result in Huffuman decode lookup, and we can stop and report an error. However, the bit error also can be decode to the unexpected answer. Our tentative idea is that we can add checksum while encode, and most of the bit error can be detected.

VII. Contribution

107061105 羅宇辰: RTL design (encode part & optimization of encode part & integration), APR, project report

107061234 張博閎: RTL design (encode part & decode part & integration), testbench, APR, project report

107061252 吳書磊: Software model, test pattern, testbench, APR, slides and presentations, project report

VIII. Reference

[1] W. Chen, CH Smith, and S. Fralic, "A fast computational algorithm for the discrete cosine transform," IEEE Trans. Commun., vol. COM-25, pp. 1004-1009, Sept 1977.

[2] https://www.fastcompression.com/pub/2012/S0273-Fast-JPEG-Coding-on-the-GPU.pdf, page6