COMPENG3DQ5 - PROJECT REPORT

GROUP 66

Haoran Qi-qih12

Hongwei Niu-niuh4

I. Introduction

This project implements the image decompression process in hardware via lossless decoding, dequantization, signal transform, interpolation and color space conversion.

The general procedure for each milestone is as follows: In milestone 3, we decoded the compressed images from the bitstream and quantized image into SRAM. In milestone 2, we recover the down-sampled image which stores in IDCT in the form of YUV by matrix multiplication using 4 multipliers. In milestone 1, we used 3 multipliers to compute up-sampling and color space conversion, then store the final image in the form of RGB.

II. Design Structure

General speaking, we have six modules in this project: color space conversion module, inverse signal transform module, lossless decoding module, provided UART_interface, VGA_SRAM and dual-port RAM modules. Begin with the top-level file, the FSM controls the flow between module to module by SRAM_address, SRAM_we_n and SRAM_write_data. In detail, the image is first sent to SRAM by UART protocol, to accomplish the UART transmission, we reused UART_interface module. Since we transmitted data using UART protocol, so all data is stored in the form of bitstream first, to decode the bitstream codes, we used lossless decoding module and converted the bitstream data to YUV value and stores them to SRAM location relatively. After that, to quantize the signal and implement inverse signal transformation, we built our inverse signal transform module and reused three dual-port RAM modules to manipulate data transfer and storage. Finally, to convert the YUV segments to RGB segments, we created our color space conversion and upsampling module and reused VGA_SRAM_interface to display the final decoded image.

III. Implementation Details

Milestone 1

a) Hardware usage and implementation

In this milestone, to implement colorspace conversion and upsampling, we used FSM as a submodule for YUV to RGB conversion processing. Specifically, we used lead-in state, which contains 10 substates, to read the first couple of YUV and store them into y, u, v register and do the RGB conversion calculation as reading data from SRAM has three cycles delay and the limitation of three multipliers usage, then we use adder to add partial result of u' and v' from 5 U and 5V data, after doing u to u', v to v' conversion, we stored them into u' and

v' buffer to do the color space conversion later on. Then after calculating the first two sets RGB values (R0G0, B0R1, G1B1), we used the common state combined with six substates to do the iteration of RGB conversion calculation. And in the common state, we used an additional y buffer to store the calculated y value prevent from overwritten. Finally, after iterations in common state, we designed the lead-out state with 33 substates as in the last few cycle of common state, we have ran out of our YUV value in the same row and do not need to read from SRAM anymore, therefore, our lead-out state is almost the same algorithm as the common state except not reading UV segments from RGB.

AS for mathematical operations, we used two main algorithms to implement U to U' and V to V' conversion, scaling and clipping. The first algorithm shown in figure 1 is for scaling and clipping, which is used for ensuring that rounding and overflows that have lead to values out of the 8 bit unsigned range are addressed by saturation. The second algorithm is shifting register logic for uv to u'v' conversion as we need five adjacent u and v values to calculate u prime and v prime, and one shifted up from un to un+1.

```
assign R_write[1] = (R[1][31]) ? 8'd0 : (|R[1][30:24]) ? 8'd255 : R[1][23:16];
assign R_write[0] = (R[0][31]) ? 8'd0 : (|R[0][30:24]) ? 8'd255 : R[0][23:16];
assign G_write[1] = (G[1][31]) ? 8'd0 : (|G[1][30:24]) ? 8'd255 : G[1][23:16];
assign G_write[0] = (G[0][31]) ? 8'd0 : (|G[0][30:24]) ? 8'd255 : G[0][23:16];
assign B_write[1] = (B[1][31]) ? 8'd0 : (|B[1][30:24]) ? 8'd255 : B[1][23:16];
assign B_write[0] = (B[0][31]) ? 8'd0 : (|B[0][30:24]) ? 8'd255 : B[0][23:16];
```

Figure 1. scaling and clipping

```
u_plus_3 <= u_plus_5;
u_plus_1 <= u_plus_3;
u_minus_1 <= u_plus_1;
u_minus_3 <= u_minus_1;
u_minus_5 <= u_minus_3;
u_plus_5 <= SRAM_read_data[15:8];
```

```
v_minus_5 <= v_minus_3;
v_minus_3 <= v_minus_1;
v_minus_1 <= v_plus_1;
v_plus_1 <= v_plus_3;
v_plus_3 <= v_plus_5;
v_plus_5 <= SRAM_read_data[15:8];
```

Figure 2 & 3. UV to U'V' conversion

b) Time efficiency

As indicated in the specifications, we need to meet the limitation of least utilization of multipliers 85%, the multipliers we used are shown as figure 3, which is 22/24, that is, 91% utilization.

cu0	cu3	cv1	a00y2		a02v'3
cu0	cu4	cv2	a02v'2		a11u'3
cu1	cv0	cv3	a11u'2	a21u'2	a12v'3
cu2	cv0	cv4	a12v'2	a00y3	a21u'3

c) Verification Strategy

To fix the bugs and verify the correctness of our design, we first used Modlesim to check if the YUV data read from SRAM is stored in the right register and cycle as three cc delays for SRAM reading data and then checked the calculation of u prime and v prime, after that, we checked the result of RGB and if the RGB segments are stored in the right location. The most challenging part is to handle the edge case of each row, as there is no value for index -1 and -2 or index over the maximum row index, therefore, when calculating u prime and v prime, we

need to set u[-1] = u[-2] = u[0] and u[160] = u[161] = u[159], for example, then we need to check how many repeated value need to be set for each case to obtain the correct u prime and v prime value.

Milestone 2

a) Hardware usage and implementation

In this milestone, the FSM is used to produce the process of computing the YUV Post-IDCT value from the YUV Pre-IDCT value. It is accomplished by using 4 types of computations (FS', CT, CS and WS). The matrix C and C transpose are used in calculation and it is pre-stored into RAM1 location 0 to 64Firstly, what FS' does is that fetch one by one of the 8x8 blocks from the SRAM memory and store them into the dual-port RAM2. Also, only one YUV value is fetched once, so the main approach we used to achieve this is that we have a cycle of {buffer,write,buffer,write,……}. Therefore, two values are stored in one location of the dual-port RAM2. Based on the CT stage, T is calculated by the equation of T = S' x C. In this situation, S' is read from dual-port RAM2, The computed T is written into dual-port RAM1 one value per location but with a 8 bits shift (divided by 256). Based on the state table given below, we use one S' value multiply four C values at one time and 8 T values are obtained after 16 clock cycles.

	LI_CT_0	LI_CT_1	LI_CT_2	LI_CT_cc0	LI_CT_cc1	LI_CT_cc2	LI_CT_cc3
SRAM_address							
SRAM_read							
read_address_a[0]			C0C1	C4C5	C8C9	C12C13	C16C17
read_address_b[0]			C2C3	C6C7	C10C11	C14C15	C18C19
read_address_a[1]							
read_address_b[1]							
read_address_a[2]							
read_address_b[2]							
address_a[0]		0					
address_b[0]		1					
address_a[1]		0	2 4	6	8	10	12
address_b[1]		1	3 5	7	9	11	13
address_a[2]		0					
Mult1_op			S'0	ct_y_buff	S'2	ct_y_buff	S'4
Mult2_op			S'0	ct_y_buff	S'2	ct_y_buff	S'4
Mult3_op			S'0	ct_y_buff	S'2	ct_y_buff	S'4
Mult4_op			S'0	ct_y_buff	S'2	ct_y_buff	S'4
C_0			C0	C4	C8	C12	C16
C_1			C1	C5	C9	C13	C17
C_2			C2	C6	C10	C14	C18
C_3			C3	C7	C11	C15	C19
ct_y_buff			S'1		S'3		S'5
T0				Mult1_result+T0	Mult1_result+T4	Mult1_result+T0	Mult1_result+T4
T1				Mult2_result+T1	Mult2_result+T5	Mult2_result+T1	Mult2_result+T5
T2				Mult3_result+T2	Mult3_result+T6	Mult3_result+T2	Mult3_result+T6
T3				Mult4_result+T3	Mul4_result+T7	Mult4_result+T3	Mul4_result+T7

In the CS implementation, the pre-computed T values and transpose of matrix C are used to calculate the result by using equation $S = (int)\frac{1}{65531}(C^T \times T)$.

The approach we used is that we use two rows of C values multiply one column of T in or der to get two values in different rows of a S block in 4 clock cycles. Since the limited number of dual-port RAM is 3, we use the bottom half memory of dual-port RAM2 to write in and store the S values from location 64.The detailed calculation is performed below.

8	16	24	1
12	20	28	5
C0C1	C16C17	C32C33	C48C49
C8C9	C24C25	C40C41	C56C57
T0	T16	T32	T48
T8	T24	T40	T56

In the final sequence WS, the implement is simply reading from pre-computed S values in dual-port RAM2 memory and directly write it back to the SRAM YUV memory location correspondingly. In order to meet the multiplier utilization constrain of 85% at least, two mega states are designed. The first mega state consists of CS and FS' and the second mega state consist of stage WS and CT. The main idea is utilizing the property of parallel reading and writing of dual-port RAMs.

b) Time efficiency

In order to meet the requirement constrain of a multiplier utilization of 85%, we used the mega states 1 and 2 to make iterations described above. The first mega state has 128 clock cycles of multiplication and the second mega state has 128 clock cycles of multiplier usage as well. And the leadIn and leadOut states contain 64 and 32 clock cycles respectively. Therefore the total utilization is :

$$\frac{2400\times(128+128)}{64+2400\times(128+128)+32} = 99.98\%$$

c) Verification Strategy

During the debug process, the biggest problem is the design of the R/W address generators for all YUV segments. And then since the read_data time delays are different between dual-port RAMs and SRAM, we encountered a lot of cross mismatches when reading and writing values as well as preparing for one single common case. That causes consequences errors when debugging using simulated waveforms. In addition, since the mega states we designed are not very short so we may have a lot of buffers, we got plenty of overwritten errors also. Finally, since the whole process is very long, each time running the simulation are very time-cost.

IV. Project Timeline Schedule and Team Member Contribution

Week	Project Schedule	Member Contribution
1	1. Finish reading the project documentations and watched project video. 2. Built the lead-in state and checked the given top-level file to get fully	Haoran Qi: Read the documentations of the project. Hongwei Niu: Check the top-level file and reusable module.

	understanding of the top-level usage.	Together: Build the lead-in state table for M1.
2	1. Check the output and input of M1 and build the frame of M1. 2. Code the lead-in state based on the state table. 3. Write the common state and lead-out state of M1.	Haoran Qi: Code lead-in state and write the lead-out states. Hongwei Niu: Code the output and input variable for M1 and build the general frame for M1. Together: Write the common states and fix the bugs for leading in states.
3	1. Finish coding common state and lead-out state for M1. 2. Use ModelSim to eliminate bugs in M1 module.	Together: Code the common state and lead-out state and fix the bugs for M1 and verify the correctness.
4	1. Start writing stable for M2 and get the big picture for M2. 2. Check the top-level file and FSM which controls M2 input and output. 3. Write the input and output variable in M2. 4. Starting coding the lead-in state and mega state 1(FS' and CS) for M2.	Haoran Qi: Code the input and output variables for M2. Hongwei Niu: Check the top-level file and reusable module. Together: Write the state table and check the top-level FSM, code the lead-in state and mega state.
5	1. Finish writing state table for M2. 2. Finish coding milestone 2. 3. Debug M2 and check correctness of M2. 4. Start coding milestone 3.	Haoran Qi: Write the rest of the stable for M2. Hongwei Niu: Code the rest part of M2. Together: Debug M2 and write project report, write the frame of milestone 3.

V. Conclusion

In conclusion, we gained a lot of fun and learnt a lot about image decoding process in this project, it is a very precious experience in our studies. We have a deeper understanding of how to debug the hardware and how important the cycle delay for SRAM and RAM, more importantly, we have learnt that how critical to divide the big chunk to the small subtask and use state table to simplify the logic.