COE3DQ5 - PROJECT REPORT

GROUP 14
Nicole Wu Mengjia Li
wuz78@mcmaster.ca lim14@mcmaster.ca
November 25, 2019

I. Introduction

This project implements the image decompression process in hardware via lossless decoding, dequantization, signal transform, interpolation and colour space conversion. The mechanism satisfies the constraint for limited space and efficiency requirements of the multiplier.

The following section consists of the design structure of the project, explanations of the design in terms of hardware structure, time efficiency and approaches. Milestone 3 decodes the compressed image from a bitstream file to a quantized image into SRAM without integration. Milestone 2 perform matrix multiplication to IDCT to recover the image into a downsampled form. In milestone 1, compute the upsampling and colouring conversion using 3 multipliers and store the final RGB value into SRAM.

II. Design Structure

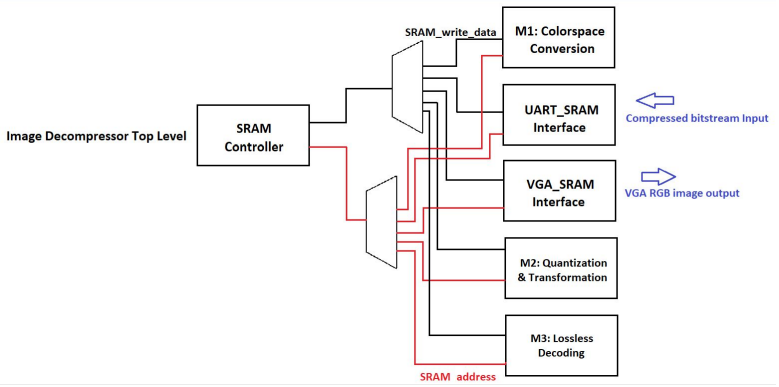

Figure 1: top-level block diagram design

Within this project, five modules are utilized to accomplish the goal of implementing an image decompressor. An FSM design in the top-level file controls the three main data buses: SRAM_address, SRAM_we_n and SRAM_write_data, and the overall data flow from modules to modules since the external SRAM storage space is shared by various modules for different purposes. In the beginning, compressed image data is sent through the UART protocol, where the module UART_SRAM_interface from lab 5 is reused to provide service of dumping the receiving data into SRAM. Since all binary information is stored as a bitstream file, a lossless decoding module is created in milestone 3 to take care of reading header, obtaining values and put them in the correct place within SRAM. Furthermore, a quantization module (milestone 2) is also newly created to convert the original image data in the time domain to YUV format in the frequency domain. During these two milestones, DPRAM module dual_port_RAM is reused as an intermediate storage place for matrix computation and achieving parallel R/W to SRAM as well. To export and display the image, a YUV-to-RGB colourspace conversion module for milestone 1 is added into the whole design. It takes 4:2:2 chroma and luma sampling data, compute them through the downsampling equation to gather 4:4:4 RGB pixels that are wanted. Finally, the VGA_SRAM_interface unit from lab 5 is reused for the sake of extracting computed RGB pixels from SRAM and transmit them to display using the VGA analog interface.

III. Implementation Details

3.1. Milestone 1

3.1.1. Hardware Structure

This module consists of a general FSM, which controls the reading from external SRAM, compute for U' values and performs YUV-to-RGB conversion using only three multipliers then writes back pixel data registers to SRAM in R0G0, B0R1, and G1B1 format.

Starting from the lead in the case, there are in total 21 states for this case to solve all the discrepancies compared to common cases. Other than lead in the case, the common case consists of 12 states that are in charge of calculations for two odd U' and V', converting four RGB pixels, along with the interaction between modules and SRAM. Moreover, in the last 5 common states, a comparison logic checking Y counter is used to help the FSM deciding whether it should go back to CC_0 or the following 26 lead-out states since it will finish reading YUV addresses before the computation is done. During the lead-out case, FSM finishes the conversion of the last three RGB pixels, then write them back to SRAM.

Within milestone 1, two main algorithms are the interpolation of odd U and V samples and colourspace conversion, respectively. In the interpolation process, 10 shift registers (i.e. *u_5* to *u_1*) are filled with chroma samples that are required in the beginning. One multiplier is used in combinational logic that multiplies zero-padding operand 1 (shift register) and sign-padding operand 2 (coefficient). After each odd value is computed, a shift logic is performed as figure 2 to prepare for the next sample:

```
//shift v'5 register
v_5[1] <= v_3[1];
v_3[1] <= v_1[1];
v_1[1] <= v_1[0];
v_1[0] <= v_3[0];
v_3[0] <= v_5[0];
v_5[0] <= SRAM_read_data[15:8]; //U6/U8...
```

Figure 2: shift logic

```
assign op1_extended = {23'd0, OP1};
assign op2_extended = {{23{OP2[8]}},OP2};
assign op3_math = (coeff == 32'sd76284) ? OP3 - 32'd16 : OP3 - 32'd128;
assign op4_math = (coeff == 32'sd76284) ? OP4 - 32'd16 : OP4 - 32'd128;

assign MULTI = op1_extended * op2_extended;
assign MULTI_RGB1 = coeff * op3_math;
assign MULTI_RGB0 = coeff * op4_math;

assign R_OUT[1] = (RED[1][31]) ? 8'd0 : (|RED[1][30:24]) ? 8'd255 : RED[1][23:16];
assign G_OUT[1] = (GREEN[1][31]) ? 8'd0 : (|GREEN[1][30:24]) ? 8'd255 : GREEN[1][23:16];
assign B_OUT[1] = (BLUE[1][31]) ? 8'd0 : (|BLUE[1][30:24]) ? 8'd255 : BLUE[1][23:16];
```

Figure 3: multiplication and clipping

Another important mathematical operation is to transform image samples from YUV to RGB. Two multipliers outside *always_ff* work at the same time to multiply 32 bits signed chroma/luma samples with signed coefficients. During all three multiplications, accumulating registers such as *u_odd_com, RED* are assigned with updated adders (i.e. *MULTI*) within the sequential logic. In the end, simple bits clipping in figure 3 divide the signed result by 65536 and converts it to desired 8 bits unsigned outcome.

3.1.2. Timing and Efficiency

To calculate efficiency, which is the total clock cycles using multipliers divided by the total clock cycles used in a common case is calculated as $\frac{2\times6+5\times4}{12\times3} = 88.9\%$. It takes 12 clock cycles to compute 4 RGB values. Two multipliers take 5 clock cycles each to calculate one RGB value. Another multiplier does the multiplication of chroma samples at the same time.

3.1.3. Approaches

Our design approach started by splitting the processing of interpolation and colourspace conversion, by which means the latter one only starts after interpolation is done. Not much time later we discovered this was a very inefficient way of implementing milestone 1 since the number of SRAM R/W operations was doubled. Therefore, we switched to implementing two processes together, which used 5 clock cycles computing 2 RGB pixels and 1 chroma sample. When got into the debugging stage, we first tried to check if SRAM_address is incrementing correctly and whether the read data is desired. During this time we found that with the current input, there was no way to get a reasonable product after multiplication when looking at the waveform in Modelsim. Thus, we used software HxD to verify the address, which further helped us to find out that we assigned a wrong offset when reading the U address section. Once after we proved that SRAM address/data and math is correct, the next step is to check registers such as *RED, u_even_com*, and *R_register_com* had correct bits updated at the correct time so no wrong values are used as multiplication operands. Finally, we verified that *y_counter_loop* incremented to the desired amount so the FSM can exit at the end.

3.2. Milestone 2

3.2.1. Hardware Structure

In the milestone 2, FSM computes the Pre-IDCT to Post-IDCT value (YUV) through 4 stages: Fs, Ct, Cs, and Ws. In the first stage Fs, the Pre-IDCT value S' of a block is fetched

from SRAM into a dual-port memory every other cycle to achieve that DPRAM has 2 values per address. In stage Ct and Cs, the FSM controls address buses and decide where to write value S', T and C. In Ct, matrix multiplication of $T = (S' \times C) / 256$ is performed while in Cs $S = C^T \times T$ is computed. To rationally use the parallel R/W function in DPRAM, during Ct T is stored one value per address (32 bits) in DP-RAM1 for further calculation. S has stored two values (every 8 bits) per address into DP-RAM0 whereas S is read and written into SRAM in the next state Ws. For efficiency requirement, states Cs (n^{th} block) and Fs ($n+1^{th}$ block) is executed at the same time, similar interpretation is done to Ct ($n+1^{th}$ block) and Ws (n^{th} block) as well. In this case, the lead-in is the Fs & Ct for the first Y block while lead-out computes the last V block's Cs & Ws.

As for the dual-port RAM planning, DP-RAM0 stores 2 16 bits S' together from address 0 to 63 and 2 8 bits S in Y0Y1 format from 64 to 127, both of them follow the same order in SRAM (i.e. S'0S'1 at address 0, S'2S'3 at address 1/Y0Y1 at address 64, Y2Y3 at address 65). Since 32 bits are necessary to compute T under the constraint of 92% utilization, every T is stored as matrix row by row in DP-RAM1 so that two Ts can be read parallelly to compute S in 4 cycles. For example, T(0,0) at address 0 and T(0,1) at address 1 until hits next row (T(1,0) at address 8).

For computation, because two multipliers are allowed in 1 clock cycle, 8 multiplication $\times 4$ clock cycles are required for obtaining 1 value. C is implemented using a mux selected by *c_index_0* and *c_index_1*. In Ct, the FSM read the first row of S' 8 times and multiply each number with 7 rows from the C matrix. The detail is explained in Figure 4.

ROW 0	Y0	Y2	Y4	Y6	Y0	Y2	Y4	Y6	Y0	Y2	Y4	Y6	...	...
	Y1	Y3	Y5	Y7	Y1	Y3	Y5	Y7	Y1	Y3	Y5	Y7	...	...
		Y0*C0	Y2*C16	Y4*C32	Y6*C48	Y0*C1	Y2*C17	Y4*C33	Y6*C49	Y0*C2	Y2*C18	Y4*C34	Y6*C50	...
		Y1*C8	Y3*C24	Y5*C40	Y7*C56	Y1*C9	Y3*C25	Y5*C41	Y7*C57	Y1*C10	Y3*C26	Y5*C42	Y7*C58	...
ROW 1	Y8	Y10	Y12	Y14	Y8	Y10	Y12	Y14	Y8	Y10	Y12	Y14	...	...
	Y9	Y11	Y13	Y15	Y9	Y11	Y13	Y15	Y9	Y11	Y13	Y15	...	...
		Y8*C0	Y10*C16	Y12*C32	Y14*C48	Y8*C1	Y10*C17	Y12*C33	Y14*C49	Y8*C2	Y10*C18	Y12*C34	Y14*C50	...
		Y9*C8	Y11*C24	Y13*C40	Y15*C56	Y9*C9	Y11*C25	Y13*C41	Y15*C57	Y9*C10	Y11*C26	Y13*C42	Y15*C58	...

Figure 4: Ct matrix multiplication

Understands that the order in matrix multiplication matters, C^T coefficients become the new row multiplies 8 columns, in this case, they are Ts. Detail mathematical interpretation is done in Figure 5.

	COL 0				COL 1				COL 2				
	T(0,0)	T(2,0)	T(4,0)	T(6,0)	T(0,1)	T(2,1)	T(4,1)	T(6,1)	T(0,2)	T(2,2)	T(4,2)	T(6,2)	...
	T(1,0)	T(3,0)	T(5,0)	T(7,0)	T(1,1)	T(3,1)	T(5,1)	T(7,1)	T(1,2)	T(3,2)	T(5,2)	T(7,2)	...
ROW 0	C0	C16	C32	C48	C0	C16	C32	C48	C0	C16	C32	C48	...
	C8	C24	C40	C56	C8	C24	C40	C56	C8	C24	C40	C56	...
	T(0,0)	T(2,0)	T(4,0)	T(6,0)	T(0,1)	T(2,1)	T(4,1)	T(6,1)	T(0,2)	T(2,2)	T(4,2)	T(6,2)	...
	T(1,0)	T(3,0)	T(5,0)	T(7,0)	T(1,1)	T(3,1)	T(5,1)	T(7,1)	T(1,2)	T(3,2)	T(5,2)	T(7,2)	...
ROW 0	C1	C17	C33	C49	C1	C17	C33	C49	C1	C17	C33	C49	...
	C9	C25	C41	C57	C9	C25	C41	C57	C9	C25	C41	C57	...

Figure 5: Cs matrix multiplication

3.2.2. Timing and Efficiency

To keep the multiplication synchronized with the reading from or writing into SRAM, it fetch S' into DPRAM and write S into SRAM every other common-case. since Fs takes 64 times but only writes 32 times, Ct and Cs take 64 CCs, Ws takes 32 times for an entire block. 2 multipliers are used together to compute Ct and Cs. Therefore, when two blocks are processed together in common-case, the utilization is 100%. Since there are 2400 block to compute, the lead in Fs and lead out Ws is ignored. Considering the lead in and lead out of state in common case, The utilization can be estimated as follows:

$$\text{Cs \& Fs: utilization} = \frac{8\times8\times4}{8\times8\times4+5} = 98\%$$

$$\text{Ct \& Ws: utilization} = \frac{8\times8\times4}{8\times8\times4+8} = 96.9\%$$

$$\text{Fs and Ws} = 0\% \text{ for } 64 + 64 \text{ clock cycle}$$

$$\text{Total utilization} = \frac{8\times8\times4\times2}{8\times8\times4\times2+5+8} = 97.5\%$$

3.2.3. Approaches

The first step is to define the memory structure and matrix storage order in DPRAM. S and S' are stored in one SRAM because 2 Ts has to be read at the same time, and T has to be in 32 bits. In the beginning, we put S' and S one value per location. As project going, we realized that the number of addresses is not enough during computing Ct and Ws. Therefore, we

switched to store two samples in the same address to support matrix calculation while achieving the efficiency requirement.

During the debugging stage, we first check the SRAM address and DPRAM address and guarantee them to be correct. To track the address of input and output of DPRAM during the multiplication, the 4 bits *row_index* and 4 bits *col_index* are used to track the element position in an 8x8 matrix. The address of DPRAM is determined by *block_row_index* and *block_col_index* along with column/row index themselves. After that, we verified the multiplication equation by comparing the value read from SRAM using HxD with the operation factors assigned, and compares the result in waveform with the theoretical result.

When debugging this milestone, we also made a mistake when trying to calculate the result S = $C^T \times T$. In the beginning, we did not realize that the expression order matters in matrix multiplication after consulting TA and we coded this in the exact opposite way, which is $T \times C^T$, furthermore lead to big trouble with very unreasonable results.

3.3. Milestone 3

3.3.1. Hardware Structure

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

Figure 6: zig-zag reading order

For this milestone, this section is discussed based on three parts: zig-zag writing, header detection, and shifting. For zig-zag writing, the diagonal index expresses the line of data in the diagonal direction as shown in Figure 6. To find the diagonal index, a combinational logic is used to compare the *we_counter*, which is the number of elements that have written, with the first *we_counter* of the row (i.e. 3 for row 2, 6 for row 3). After that, by subtracting the current *we_counter* with the first *we_counter*, the position of the data in the diagonal line can be determined. The pattern is found to be the difference between two adjacent values is +7 in the odd diagonal index or -7 in even diagonal index. To find the location of each matrix element, the sum of the first address ± 7 × (*we_counter* - *first we_coutner*) is determined. Finally, the location of the data is converted into the *row_index* and *col_index* for writing into the memory. Two assignments define the *row_index* as the location divide by 8 and the *col_index* equals the location % 8.

In the decoding process, various cases are determined by 2 bits header. According to the scan pattern, the next bits indicate whether to have a loop to write zero or value, or goes to another state. When a value written into memory is determined a shifting logic is used, then back to decode header process. In order to continuously read from the input bitstream, the 32 bits shift_reg plays a significant role in this milestone. Before starting decoding, this shift register is filled with two 16 bits inputs from embedded memory. Meanwhile, a 16 bits temporary storing register memory_data is filled with a portion of input bitstream whenever appending is needed. For each 2 bits header case, 5 different shift values are considered: 4, 5, 7, 8, 12 using a register *bit_shift_unit*. an *always_comb* selection logic is written to choose the correct shifting algorithm. For instance, if *bit_shift_unit* is 5, shift register is assigned with *SHIFT = {shift_reg[26:0],memory_data[15:11]}* and memory_data is shifted by corresponding bits to the right. Within FSM, the shift condition is examined in state *S_shift_DETECT* every time a header identification is finished and a tracking shift counter is incremented by the shifting value. In *S_WAIT*, an exception is handled to check and solve the overshift issue.

3.3.2. Approaches

In the first approach, we assign a shift register for every bit to do the shifting. Since 32 shift registers consume large space in coding and is easy to make mistakes, we changed to the method described above in hardware structure.

In the debugging process, we first verify the diagonal index, writing addresses, and block index incrementation. When we load the code on the hardware, the FSM got stuck in milestone 3 unless we reset the board before sending the picture after verifying the write enable, top states and address. We found that the *SRAM_we_n* is not initialized in the reset condition in M3, which results in a "don't care" value at the beginning until It is assigned as 1 in the IDLE state. Therefore, we need to reset the FSM, it takes the old value and the FSM can go to the next state.

3.4. Project Timeline

Week	Project Progress	Member Contribution
1	**1**. Read through the project description document, get a general big picture of image decompressor. **2**. Brainstorming the starting point.	**Lily**: Read the document, understanding the concept and project flow. **Nicole**: Understand the data flow between milestones, the input and output.
2	**1**. Started writing, modifying and completing the state table for milestone 1. **2**. Started writing the Verilog code.	**Lily:** Implemented a preliminary state table, discuss with Nicole and chose the final state table. Also wrote the common case code within milestone 1 module. **Nicole**: Implemented a preliminary state table, discuss with Lily to select the final state table. Wrote the lead-in and lead-out case within milestone 1 module.
3	**1.** Filling in details for milestone 1 module Verilog code. Adding FSM control logic in top level file, get simulation running. **2**. Debugging milestone module, eliminate errors using Modelsim. **3**. Tested the verified code on board, and confirmed it is successfully working on FPGA.	**Lily**: Take a second look at the lead-in/lead-out portion. Debugged milestone 1 module and verified on board. **Nicole:** Take a second look at common case portion. Modified the testbench and do. files for simulation purposes. Helped speed up the debug process.
4	**1**. Started brainstorming, writing and finishing the state table for milestone 2. **2**. Halfway through implementing Verilog code for milestone 2.	**Lily**: Worked with another member to finalize the state table. Wrote the lead out Cs and Ws case code. **Nicole**: Wrote the state table and give the equation for computing matrix. Wrote the common Cs_Fs and Ct_Ws cases code. **Together:** wrote the lead-in Fs and Ct case code.
5	**1**. Finalizing the milestone 2 module code, fixing syntax and algorithm mistakes. **2**. Set up the simulation for milestone 2 by modifying the corresponding testbench and top-level files.	**Lily**: Debug M2 module, found and fixed some mistakes within the code. Wrote some code for milestone 3. **Nicole**: Debug M2 module, found and fixed some mistakes

	3. Debugged and fixed errors using Modelsim and verified milestone 2 modules on board. **4**. Wrote the code and debugged milestone 3 without integration.	within the code. Wrote the code for milestone 3. **Together**: Debug milestone 2 simulation waveform, fixing errors and verify this implementation on FPGA. Debug milestone 3 simulation waveform.

IV. Conclusion

In conclusion, this project has given us a remarkable learning experience, it was fun but also painful. After finishing it in four weeks, we had a better understanding of how to implement a scaled project in the future. Some keywords play an enormous role: teamwork, communication and time management. To us, the difficulty of the project is intermediate but quite time consuming, which requires the abilities above to finish this project. We have learned how to split up tasks into smaller subtasks and complete them one at a time, also we understood what is called to efficiently communicate with team members and work together to solve each question. More importantly, we need to take responsibility for work assigned to individuals and use time wisely.