COE3DQ5 – Project Report

Samer Rafidi - rafidis - 40033524, Tamer Rafidi – rafidit - 400333527

Group 85

November 25th, 2024

1. Introduction

The goal of this project is to develop a hardware-based image decompressor by implementing critical operations such as lossless decoding, dequantization, inverse signal transformation, interpolation, and color space conversion. The process is structured in a reverse-engineering style, starting with the higher-level steps like interpolation and color space conversion and gradually working backward through the Inverse Discrete Cosine Transform (IDCT), dequantization, and finally lossless decoding.

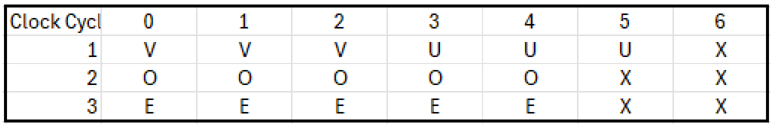
This methodology offers valuable insight into both the theoretical and practical aspects of image compression and digital design. The hardware implementation is developed using Verilog, with tools like Quartus to perform simulations, evaluate circuit timing, optimize the design, and program the FPGA. Once deployed, the hardware demonstrates its ability to decode and reconstruct compressed images effectively.

By completing this project, we gain hands-on experience in translating complex algorithms into hardware solutions, bridging the gap between abstract digital design concepts and real-world applications. This experience underscores the significance of hardware design in enabling efficient, high-performance systems.

2. Implementation Details

2.1 Upsampling and Colour Space Conversion (Milestone 1)

Colour space conversions use 5 multiplications for odd pixels, 5 multiplications for even pixels. While interpolation uses 3 multiplications for V pixels and 3 multiplications for U pixels. That is a total of 16 multiplications in 21 clock cycles. Usage of our multipliers to achieve our utilization of 75% below.



Clock Cycl	0	1	2	3	4	5	6
1	V	V	V	U	U	U	X
2	O	O	O	O	O	X	X
3	E	E	E	E	E	X	X

Register name	Bits	Description
R_odd, R_even, G_odd, G_even, B_odd, B_even,	32	Stores calculated RGB odd and even values that will be stored in the SRAM after clipping and scaling.
r_even, r_odd, g_even, g_odd, b_even, b_odd	8	Clipped and scaled values of RGB odd and even values that will be written back to the SRAM.
Y_even, Y_odd, Y_even_buf, Y_odd_buf, U_even, V_even, Up_even, Up_odd, Vp_even, Vp_odd	16	Y_even and Y_odd are used to compute RGB even and odd values respectively. Buf versions are used to store SRAM read data before passing it to Y_even and odd respectively. U and V registers hold values used to calculate RGB odd and even values.
U_buf, U_buf_2, U_buf_3, V_buf_1, V_buf_2, V_buf_3,	16	Hold U and V values in buffers to simulate U' and V' calculations. 6 U and 6 V values are stored at once.
U_buf_4, V_buf_4	8	Holds U and V values as they come in from the SRAM.
VU_mul1, VU_mul2, VU_mul3	32	Holds the result for the three U or V multiplications before its computation is passed on to Vp_odd or Up_odd register.
Mul_1, Mul_2, Mul_3	32	Holds one part of the multiplier calculation. Used for V and U odd calculations as well as RGB calculations.
RGB_odd1, RGB_odd2, RGB_odd3, RGB_even1, RGB_even2, RGB_even3	32	Holds multiplied partial products before passed on to registers used to store RGB values in the SRAM

Y_BASE_ADDRESS_COUNTER, U_BASE_ADDRESS_COUNTER, V_BASE_ADDRESS_COUNTER, RGB_BASE_ADDRESS_COUNTER	18	Base addresses set to given memory locations. Used to read data from the SRAM.
counter	18	Counter used to keep track of when to go into lead out or end milestone1. Also used to decide whether we should do a V or U pixel read from the SRAM.

Our implementation has 7 clock cycles per common case (7 x 3 multipliers = 21). Since we must do 16 multiplications, we have a utilization of approximately 76% (16 multiplications / 21 clock cycles).

21 lead-in clock cycles with 14 of those using multipliers, 7 common case clock cycles with all using multipliers, 7 lead out clock cycles with all using multipliers, common case will run 157 times and lead out will run 3 times.

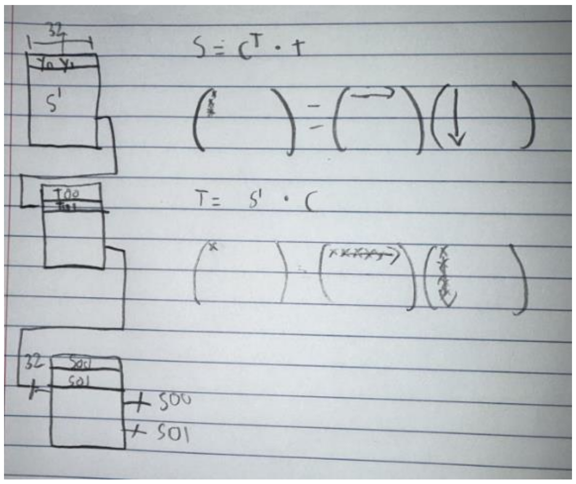
$$Total\ row\ clock\ cycles = 21 + (7 * 157) + (7 * 3) = 1141$$

$$Total\ Clock\ Cycles\ per\ row\ with\ multipliers = 7\ lead\ in + 1099\ common\ case + 21\ lead\ out = 1127$$

$$Total\ clock\ cycles = 1141 * 240\ rows = 273{,}600\ to\ complete\ M1$$

$$Multipliers\ utilzation = \frac{Total\ Clocl\ Cycles\ per\ row\ with\ multipliers}{Total\ Clock\ Cycles\ per\ Row} = \left(\frac{1127}{1141}\right) * \left(\frac{16}{21}\right) = 75.256\%$$

2.2 Inverse Discrete Cosine Transform (Milestone 2)



Our first DPRAM is used to fetch and store our S' pixels from the SRAM. Our second DPRAM stores the product of our C matrix coefficients which are stored in a MUX and the S' pixels fetched from the first DPRAM. Our third DPRAM stores the products of the T values, and the C transpose values from the MUX. Then, we write the S values into the SRAM. For calculating T values, we calculate one data point every 3 clock cycles. For calculating S values, we calculate 3 data points every 9 clock cycles.

Register name	Bits	Description
Counter, CT_counter, read_counter, CS_counter, column, WS_counter, WS_offset, column_counter, never_reset, tracker	18	Counters/trackers used to facilitate T and S calculations and checks to go to lead out.
FS_stopper, FSCT_stopper, CT_stopper, CTCS_stopper, CS_stopper, CSWS_stopper, WS_stopper, WSCS_stopper, FS_stop, CT_stop, CS_stop, WS_stop	18	These stoppers keep incrementing and allows our different FSMs to run simultaneously.
address_x (0-5)	7	Holds the address each port is looking in DPram.
write_en_x (0-5)	1	Assets the write enable for the DPrams.
Row_block, col_block, ws_row_block, ws_col_block	7	Counters used to calculate the row_address and column_address which calculates SRAM_address
col_index, row_index, ws_row_index	4	Counters used to calculate row and column address to calculate SRAM_address

ws_col_index	3	Counter used to calculate column address to calculate SRAM_address
YUV_fetch	1	Flag to decide if we are fetching from Y pixels or U/V pixels (switches equation to calculate SRAM_address)
Row_address, col_address, WS_row_address, WS_col_address,	18	Row and col address are used to dictate which SRAM_addresses we fetch or write into
C0, C1, C2	32	Holds values of the C coefficients.
c_0, c_1, c2_, coef_offset	7	Used to locate correct C coefficients in the MUX.
FS_start, CT_start, CS_start, WS_start	1	Used to start its respective state machine.
T_sum, S_sum1, S_sum2, S_sum3	32	Holds value of T and S matrix calculation after partial products are summed. These values are written into their respective DPram.
s_sum1, s_sum2, s_sum3,	8	Clipped and scaled version of the S values. These are written into the SRAM.

Latency Calculation

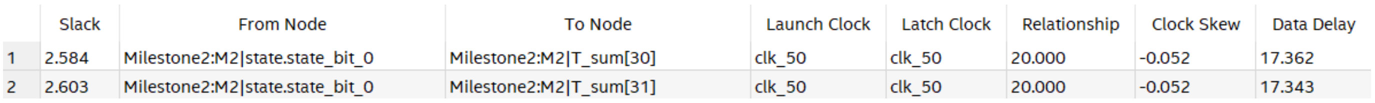
$$Total\ clock\ cycles\ for\ Y\ segment = Clock\ cycle\ per\ row * total\ column * total\ rows$$
$$= 524\ clock\ cycle\ per\ row * 40 * 30 = 628{,}800$$

$$Total\ clock\ cycles\ for\ U\ and\ V\ segment = Clock\ cycle\ per\ row * total\ colum * total\ rows = 524 * 20 * 30 * 2$$
$$= 628{,}800$$

$$Total\ clock\ cycle\ to\ complete\ M2 = 628{,}800 + 628{,}800 = 1{,}257{,}600$$

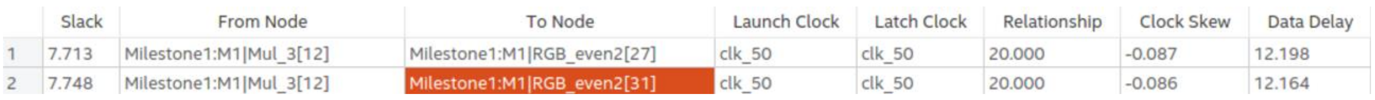
2.4 Resource Usage and Critical Path

When counting the total number of logic elements. I counted 2101 for milestone 2 and 1267 for milestone 1. This is a total of 3368. However, Quartus estimates a total of 3698 logic elements for the whole project. The number provided by Quartus is not higher than expected because Quartus automatically removes inactive logic elements bits that are not being used for in our milestones. The starting point for this project was 616 logic elements so we see a 2x increase just from milestone 1 and another ~3.4x for milestone 2.



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	2.584	Milestone2:M2|state.state_bit_0	Milestone2:M2|T_sum[30]	clk_50	clk_50	20.000	-0.052	17.362
2	2.603	Milestone2:M2|state.state_bit_0	Milestone2:M2|T_sum[31]	clk_50	clk_50	20.000	-0.052	17.343

Our critical path from milestone 2 is at our T_sum which is updated constantly. The T_sum register is an accumulator which is constantly being updated to keep track of all the results. However, our milestone 2 is not 100% complete.



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	7.713	Milestone1:M1|Mul_3[12]	Milestone1:M1|RGB_even2[27]	clk_50	clk_50	20.000	-0.087	12.198
2	7.748	Milestone1:M1|Mul_3[12]	Milestone1:M1|RGB_even2[31]	clk_50	clk_50	20.000	-0.086	12.164

Our critical path is from our Mul_3 to our RGB_even2. Our RGB_even2 is where our multiplier result is stored. This makes sense because our RGB_even2 multiplier is updating 5 times in every 7 common cases. It is constantly being updated with multipliers which are very cost heavy. Mul_3 is where our U and V prime values were being stored. It makes sense that this register gave us a bigger delay than our Mul_2 because that register was storing a coefficient.

Apart from trimming down registers to only the bits we need, we've pinpointed three main areas where we could improve our design in future updates:

Simplifying the Lead-In States (Milestone 1)
Early on, while we were still getting the hang of the project specs, we set up the lead-in sequence using 21 states. The last 7 states (14 to 21) basically repeated the same logic for our common cases. After finishing Milestone 1 and

reviewing our state table, we realized we could've combined these steps into just 14 states. This would make the design much simpler and more efficient without losing any functionality.

Improving the C Coefficient MUX
We knew we could cut down the number of coefficients in each MUX to only the ones the multiplier needed. This was something we planned to do after debugging the Milestone 2 code since we'd already accounted for offsets in our design. Unfortunately, we ran out of time to fully debug and make these changes. With more time, this would've been a straightforward optimization to clean up the design.

Fixing the Calc S State
Right now, the Calc S state stores calculated values immediately in registers or DPRAM, which doesn't leave any room for scaling or clipping beforehand. This means we're storing a 32-bit value in a DPRAM address, but the top 24 bits are just zeros. While it worked for writing S values back into SRAM, it's not efficient. With a better design, we could store only 8 values in the DPRAM before writing them, which would save space and make things more streamlined.

3. Weekly Activity and Progress

Week	Project Progress	Individual Contributions
Week 1 (Transition week)	We began to read project document and understand the concepts to work on M1	
Week 2	Worked on M1 state table and almost finalized it. Talked to TAs about our implementation	Worked in unison at this stage, no separate individual contributions.
Week 3	Coded our M1 implementation and debugged it.	Worked in unison at this stage, no separate individual contributions.
Week 4	Finished debugging M1 (Wednesday) and began working on M2 state tables and understanding the concepts	Worked in unison at this stage, no separate individual contributions
Week 5 (Deadline week)	Coded and debugged M2 (Finished many cycles but get random mismatches at col block 26, row block 2).	Samer worked on SRAM_addressing for Fs and Ws while Tamer worked on DPRam addressing and the usage of DPrams. Debugging for all stages were done together but expertise was split.

4. Conclusion

In conclusion, this project has been a great learning experience for our team, teaching us how to work together effectively, solve problems, and step out of our comfort zones. We got hands-on experience with state tables and worked with hardware components like dual-port RAMs, shift registers, multiplexers, and multiple always_ff blocks. Debugging timing issues was challenging but helped us sharpen our problem-solving skills. Along the way, we learned how to design with buffers, counters, arithmetic shifting, and multipliers, which made us more confident in handling complex systems. By fixing designs and debugging, we built up our ability to adapt and work through problems, setting us up well for future projects.

Milestone 1 completion: Nov 14th, 2024. Commit message "Fixed flashing to board".

Milestone 2 completes writing to the SRAM for 26 column blocks and 2 row blocks (106 blocks) but gets a mismatch at SRAM_address 2664 when writing to Y segment. All separate tasks (Fs, Ct, Cs, Ws all implemented properly and debugged). Unsure why the mismatch occurs after many 106 blocks.

5. References

[1] Nicolici, N., Thong, J., Kinsman, A. 2024. "COE3DQ5 Project description 2024: Hardware Implementation of an Image Decompressor" In: 3DQ5. Pages 1-31.