# COE3DQ5 – Project Report
# Group 33
**Fahad Mahmood – 001414984 – mahmof4@mcmaster.ca**
**Wei Che Kao – 001328256 – kaow@mcmaster.ca**
**November 26, 2018**

## Introduction

The purpose of the project was to design and implement image compression specification in hardware. The project was broken down into 3 milestones that performed decoding, de-quantization, inverse signal transformation, interpolation and color space conversion. We were required to be proficient with Verilog, coding multiple different interfaces for communication between the PC, timing simulation for debugging waveform and hardware board as well as the monitor display. After finishing all that we used universal asynchronous receiver/transmitter (UART) from a PC to compressed data from a 320 by 240 pixel image and store it back to the SRAM.

## Design Structure

In the beginning of the project, we reused lab 5 experiment 4b modules as a guideline to start milestone 1. These reused modules consisted of modules that performed basic required tasks such as the clock, seven segment display and controller modules for push buttons. We also added the top-level module for testing the milestones which was connected to the UART, SRAM and VGA together. The data was sent from the computer to the board through UART and was written to SRAM before being displayed on screen via VGA. We also added a of couple header files which held VGA parameters for different resolutions and the second one handled the enumeration of the multiple required state machines in the different modules. The custom modules were milestone 1 and milestone 2 modules which consisted our codes, design concept and thought process. Finally, we used the test bench module to debug the custom modules and help us bring all the modules together to make the project working.

## Implementation Details:

To implement our design, we had to create 3 different modules called milestone 1, milestone 2 and milestone 3 and reuse the previously created modules called VGA, UART and SRAM. The in-depth explanation for top level, milestone 1 and milestone 2 modules is provided below.

### Top Level Module, UART SRAM Interface:

Majority of the changes were made to milestone 1 and milestone 2 modules but there were a handful of changes that needed to be made to the top-level module as well. Firstly, we integrated different modules for milestone 1 and milestone 2 in our top-level module. We created signals to start and finish milestone 1 and milestone 2 module which we called milestone1_start, milestone2_start, milestone1_done and milestone2_done. We made a new state in our FSM to go to our milestone 1 module or milestone 2 module. We altered our FSM in such a way that it directly went to milestone 1 module or milestone 2 module. To achieve that we used a counter that incremented to 10 before we moved to the next state. Another change that we made to our top-level module was appropriately assigning SRAM address, SRAM write data and SRAM write enable. To do that we added a condition saying if we were in milestone 1 or milestone 2 state

it would assign the beforementioned to the appropriate modules. Lastly, we changed the VGA base address to 146944.
When we put our code on board, we were not getting the proper image. There was some distortion. So, we needed to make a change to UART SRAM Interface state machine where instead of stripping we went straight into the first byte received.

**Milestone 1 Module**

We created a separate module for milestone 1 which took Clock, Reset, SRAM read data and start signal as input from top level module and outputted SRAM write data, SRAM address, SRAM write enable and done signal.
Firstly, for addresses, we created an always comb block that according to state we were in assigned address to SRAM address and we had separate counters for fetching UV and Y values from SRAM. Although there was no need to create a separate always comb block for this, this was purely done to make the code easier to read during debugging and making changes. Secondly, we created another always comb block for using multipliers as well as writing data to SRAM depending on the state we were in. This block worked together with always ff block in using buffers. We used this block for writing data to SRAM because we wanted to avoid any delay and write data to SRAM right away. Lastly, we created an always ff block where we had all our states, flags, buffers, counters and shift registers.

***LEAD IN STATES:***
In our implementation of state table, we had 6 lead in states. In first four states we asked for U, V, U and V addresses and as we got the values, we fed them into U and V shift registers respectively right away by firstly filling registers with U0 and V0 values. As the shift registers got filled with the values to U and V we moved to our Common case. The common case repeated for 78 cycles till we got U160, U161 and V160, V161 values, which we replaced with U159 and V159 for two cycles. The results from multipliers for U odd and V odd were right shifted by 8 to account for division by 256 and results for RGB were right shifted by 16 to account for division by 65536 and then clipped to 8 bits.

***COMMON CASE:***
We had constraint of 80% utilization of multipliers so to achieve that we either had to go with 6 states of 12 states in common case. We went with 12 states and only 1 of our multipliers was not doing anything in 4 states which gave us a total utilization of $\frac{3(8)+2(4)}{3(12)} = 89\%$.
First state: We asked for Y values from SRAM as well as used multipliers for doing odd calculation of U and buffer the result from multiplier for j odd and buffered j even.
Second state: We asked for U values from SRAM. Used multipliers for doing odd calculation of V and buffer the result from multiplier for j odd and buffered j even.
Third state: We asked for V values from SRAM. Meanwhile, we received Y values that we buffer to use later for RGB calculation. In this state we do matrix multiplication for R values, that we buffer. We also buffer multiplication 1 result because that was reused later to do the multiplication for G and B.
Fourth state: We check if we were in our first iteration of common case. If we were, we did not write RGB values to SRAM because they were not yet fully calculated at that moment. Meanwhile we calculated G and B values from multiplier and buffered them.
Fifth and Sixth state: These states were similar to third and fourth except, we did multiplication to find the next RGB values and buffered them.
Seventh - Twelfth state: These states were exactly like the above six states except we only asked for Y values in contrary to asking for YUV values.

***LEAD OUT STATES:***

We had three lead out states which did nothing but writing the last buffered RGB values from our common case. So, we wrote R318G318, B318R319, G319B319 in these three states. After writing these three we moved on to our final state which sent milestone1_done signal to our top-level module which enabled VGA from where we started putting code on the board.

***Debugging:***

Debugging milestone 1 for us mainly consisted of getting the state table right, which is why we spent huge chunk of time working on it. So, we did not run into big problems in milestone 1. Despite of that we encountered couple of problems debugging milestone 1. The approach we used to see if we were writing correct values was using Hex editor to see whether the values we were writing corresponded correctly to the addresses or not.

In the beginning when we created state table, we thought we could write RGB values in the lead in stage. However, we were told by TA that it is not recommended because it makes it hard to debug when we reach the common stage cycle since the data would not fetch properly and it was also not very efficient to do that. The solution we came up with is that just wrote RGB values in the common stage and loop it 80 times and write the last two RGB values in the lead out since there was a two-clock cycle delay when reading the address for YUV.

The other common errors we encountered were that we were not writing the correct values in the correct location when dubbing the common stage loop. The problem was that we fetched two values from the Y address in the common stage, but we were supposed to fetch 4 values from the Y address. Some other minor issues of incrementing counters were fairly simple to debug.

**Milestone 2 Module:**

For milestone 2 we were only able to finish one block. We only needed to code mega states and two counters to count to the next block, but we were not able to get to that point in time. We created a milestone 2 module same way as milestone 1 module, except we changed the names of signals. Also, the addressing, multiplication and writing was done the same way as milestone 1 in two separate always comb blocks and we had an always ff block for buffers, counters, flag and registers. C values were stored in dual port RAM 2 using mif file, and 2 16-bit values of C were stored in 1 address location.

***FETCH S':***

For fetch S', we had 5 states in total. 2 for lead in, 1 common and 2 lead out. In lead in states we asked for SRAM addresses which we got in our common states, which we then wrote to dual port RAM 0. This repeated for 32 clock cycles. We had to write last 2 values from SRAM addresses to dual port RAM 0, which we did in the 2 lead out states. One value of S' was stored in one address in dual port RAM 0. In order to make sure we get right values from SRAM for the first block, we made a column counter that looped from 0 to 7 and row counter which incremented from 0 to 7 when column counter reached 0 again.

***COMPUTE T:***

After fetching the S' values we moved to compute T. Compute T had 4 states, 1 for lead in, 2 were common states and 1 for lead out. In lead in state, we asked for S' value from dual port RAM 0 and 4 values of C using two ports in dual port RAM 2 using even and odd addresses. In common state, we kept asking for next values meanwhile using multipliers to do matrix multiplication and accumulating. The way we did matrix multiplication was that we multiplied 1 row of S' with 4 columns of C and then we multiplied the same row to the next 4 columns of C. We then repeated that for the next row of S'. As shown in the Figure 1. We stored one value of T in one address of dual port RAM 1 and before storing we right shifted the value by 8

bits to account for division by 256. Writing two 32-bit values at a time was challenging task however, we were able to do it using even and odd addresses using both ports. The accumulated values were written in two clock cycles, hence two values were buffered.

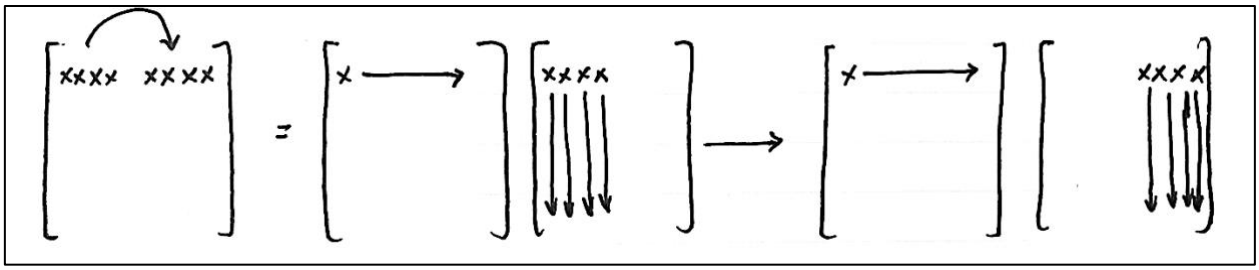

*Figure 1 - Approach to compute T*

***COMPUTE S:***

Compute S was more or less exactly the same as compute T. The difference being writing 4 values to one address in dual port RAM 0, so we eliminated the need to use two ports to write at separate locations. Hence our states were cut down to 3 instead of 4. Lead in, same as in compute T asked for one address of T and 4 addresses of C transpose which we got in our common state where we computed using four multipliers and wrote to dual port RAM 0 in state 3 where all the values were accumulated from multipliers. We did not need a lead out because all the values were written by the time we finished the loop. Before we wrote the values to the dual port RAM 0, we right shifted them by 16 bits to account for the division by 65536 and then clipped them to 8 bits as the values are written back to SRAM in that format. The approach to compute S in show in Figure 2.

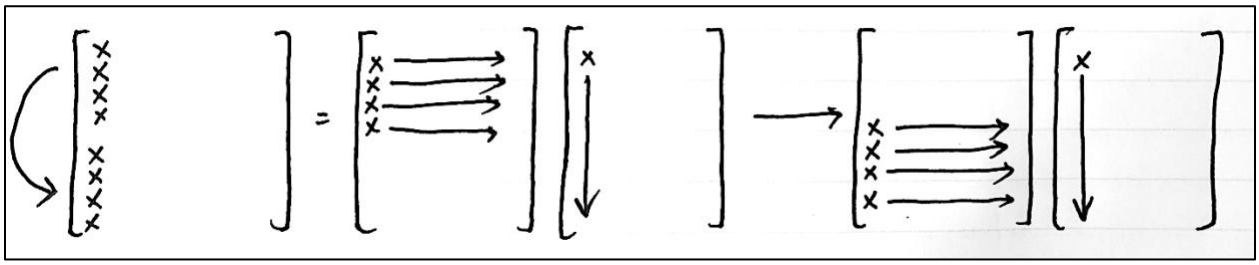

*Figure 2 - Approach to compute S*

***WRITE S:***

Writing the values of S back to SRAM was a very challenging task. As shown in Figure 3 we were storing values S0, S8, S16 and S24 in one address which meant we could not directly fetch values from dual port RAM and write them to SRAM. So, we had to think of some logic which would help up simplify this process. We ended up with having two address counters; even and odd that incremented 3 times each. Firstly, the even counter incremented, and we fetched values from locations 0 and 2. We ended up getting Y0Y1, Y320Y321, Y640Y641 and Y960Y961. We then used the odd flag to increment SRAM address accordingly

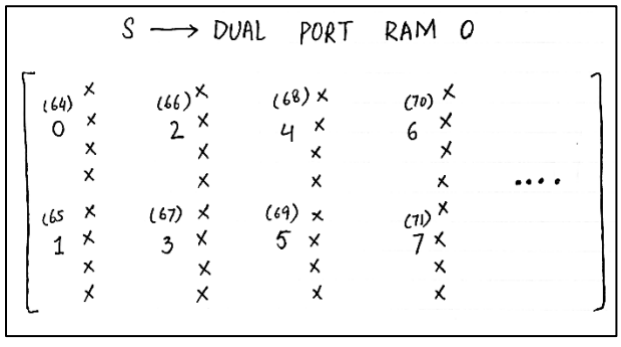


*Figure 3 - Approach to fetch S from dual port RAM 0*

and write to that address. Secondly, the odd counter incremented, and we fetched values from locations 1 and 3. These got written same way as before, but SRAM address started from 640 as we got Y1280Y1281, Y1640Y1641, Y1960Y1961 and Y2280Y2281. Since, we could only write 2 values at a time to SRAM, we ended up using 5 states. This process was 32 clock cycles. Next, we moved to last state where we sent milestone 2 done signal to top-level module.

***Debugging:***

The debugging approach we took for milestone 2 was fixing the addresses and counters. At first, we outputted all the addresses from dual port RAMs and SRAM in ModelSim to make sure we were getting them at the right time according to our state table, before inputting any values to the RAMs. This took couple of days to fix because we changed our approach numerous times. But once we got our approach right the debugging for milestone 2 was fairly straight forward. After inputting the values in RAMs, we debugged one state at a time. Firstly, we started debugging fetch S' where we did not face any difficulty. Then we move onto compute T. Once we fixed our approach, compute T was simple enough. The only problem we faced in compute T was that we were not storing signed bits properly which was a simple solution. One compute T started working, compute S was fairly straight forward too. The hard part was debugging write S because we had to use even and odd address counters and increment SRAM address in different ways using flags. An example of debugging approach we took was first computing T and S values on paper and then using hex editor to see if we were indeed getting the right output and then comparing that to the write S values.

## Timeline:

| | |
|---|---|
| Week 1 | Reading over the project document and understanding the motivation behind projects and requirements. |
| Week 2 | Since both us were busy during this week, we were not able to do much however, state table was started, and we got the feedback from TA. |
| Week 3 | This week mainly consisted of getting the state table perfect. We made multiple drafts of state table. By the end of the week we started coding. |
| Week 4 | We finished coding milestone 1 earlier this week and started debugging it. Coding was mostly done by Fahad Mahmood while Wei Che Kao helped in debugging and helping out with coding. By the end of this week, we finished milestone 1 and moved on to milestone 2. |
| Week 5 | By early this week we finalized our approach for milestone 2 and started coding right away. We had to rethink our approach numerous times because of the complications with matrices. Debugging started later this week. Due to time spent of changing our approaches we were only able to finish one block. |

## Conclusion:

Working on this project was a very fulfilling experience. We learned a lot while working on this project that includes technical skills like hardware programming and soft skills like working with group members. This project not only taught us how to work with memory (SRAM and DP RAM), registers, counters, multipliers and adders to achieve a task but it also taught us about iterative design process. Overall, it was a very valuable experience as computer engineers that will benefit us in future.

There were no problems between the group members, Fahad Mahmood and Wei Che Kao. Both of us tried our best to work around our schedule to be able to work on this project together. We did not collaborate with any other group as the instructor and TAs were very helpful throughout the whole project. However, to conceptualize the milestones and requirements we had discussions with our peers.

## References:

N. Nicolici, "COE3DQ5", *COE3DQ5*, 2018. [Online]. Available: http://www.ece.mcmaster.ca/~nicola/3dq5/2018/. [Accessed: 26- Nov- 2018].

## State Tables:

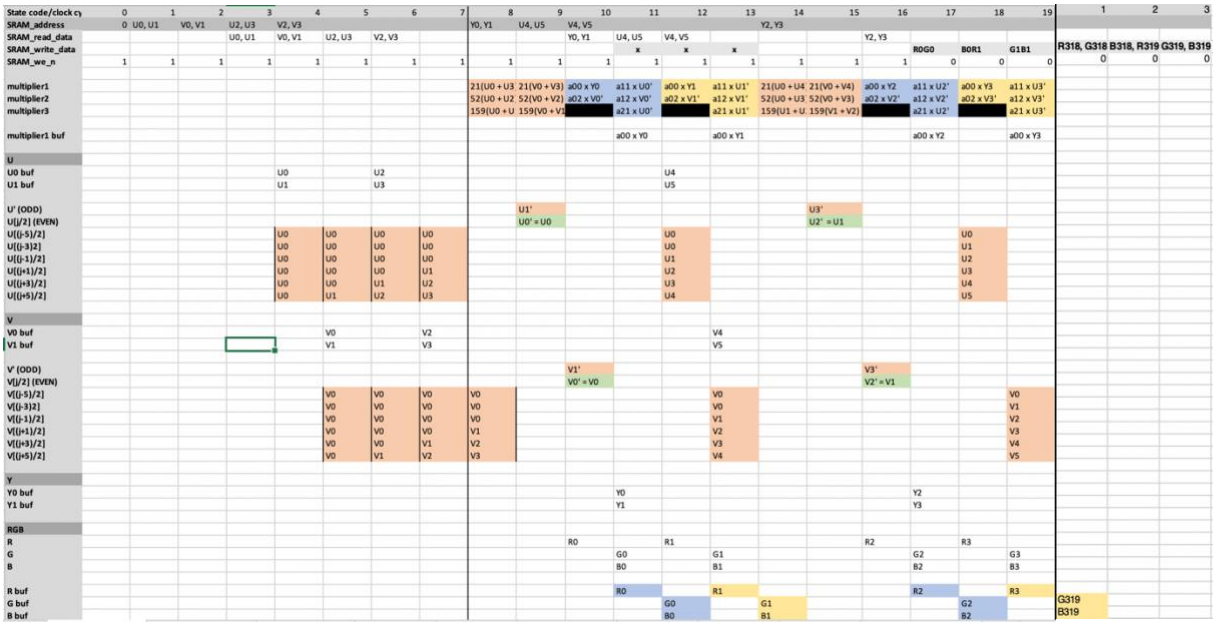

*Figure 4 - Lead In, Common Case and Lead Out for milestone 1*

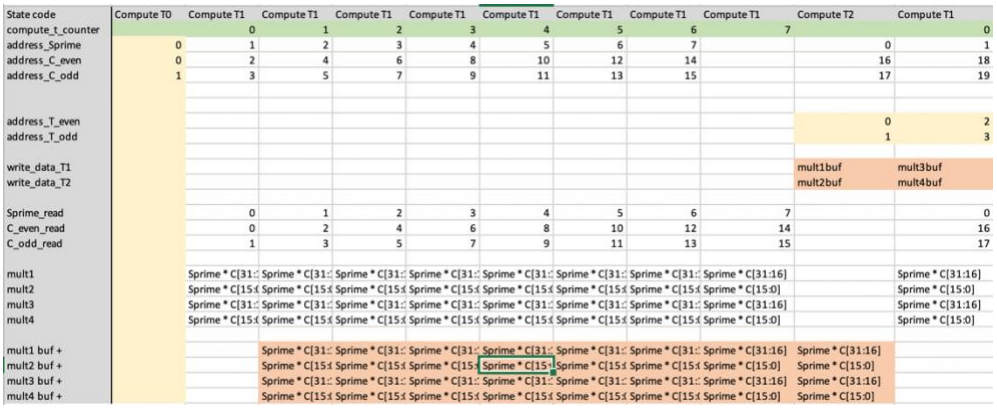

*Figure 5 - Compute T*

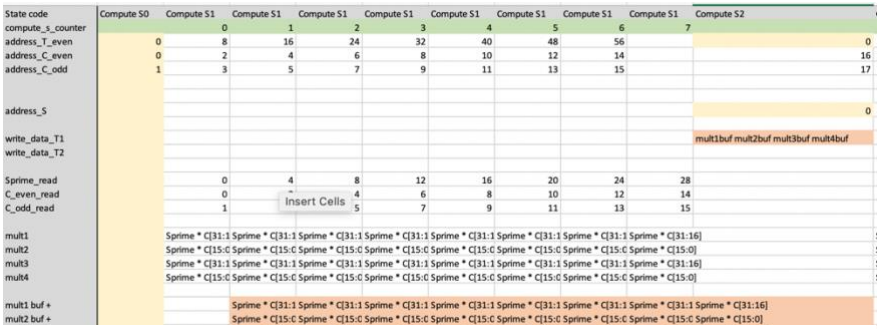

*Figure 6 - Compute S*