COE3DQ5 - Project Report

Group 55

William Siddeley - siddelew@mcmaster.ca
Mohamed Al-Asfar - alasfarm@mcmaster.ca

11/29/2021

## Introduction

The objectives of this project revolved around implementing the custom McMaster Image Compression specification in hardware. Ideally, compressed data for a 320x240 pixel image would be delivered to the Altera DE2-115 board via the UART interface. The image would then be recovered and stored in the SRAM, which would be displayed to the monitor via the VGA controller. Given the online conditions of the course, ModelSim is instead used to simulate the function of the board and provides a detailed breakdown of the output signals which contain key information for debugging and design analysis. When given a closer look into the specifications, the design of the project must comply with several outlined constraints that are specific to each of the project milestones such as exploiting the symmetry in the interpolation filter, the number of multipliers and their utilization, as well as the number of dual-port memories.

## Design Structure

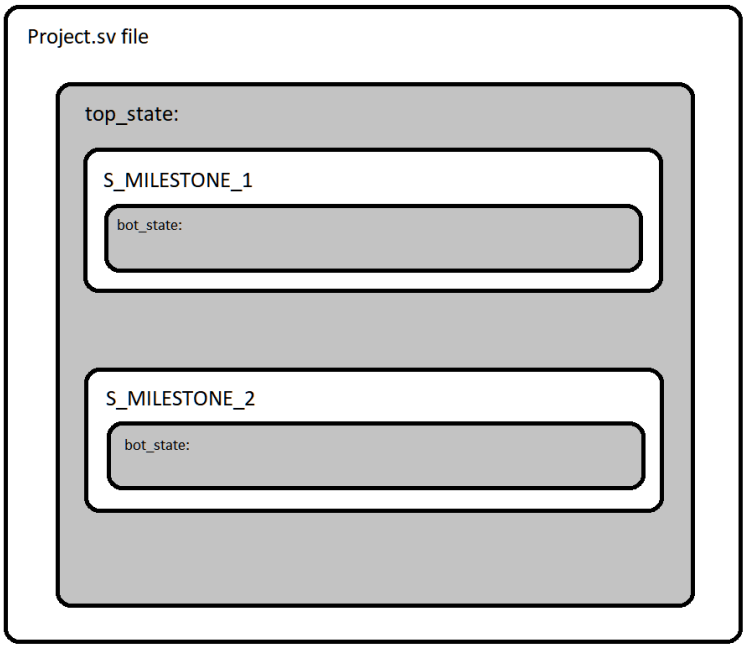
For the most part, our design was not partitioned into multiple modules and was instead completed in a single project file. The reason behind this design choice was to focus on implementing the milestones before focusing on module organization in order to ensure any issues in our project were a result of our code and not a formatting error. Once the milestones were complete we would split the project into three modules, one for each milestone, but since Milestone 3 was not attempted our project remained in a singular module at our deadline. The reason for this initial plan was to organize our project in order to increase the efficiency of any final edits and changes to the code. In hindsight, implementing this philosophy earlier in the design process would have proven to be extremely beneficial but given the time constraints we decided to settle with our final design structure. This design structure is composed of a design similar to that which was used in the labs, where a design would have multiple states, and transitioning through the states would allow for us to implement our logic on a per-clock-cycle basis.

Even though our design is not broken up into modules, there are benefits that we found when working with this design type. Variables that are created will be created for the entire project, so reusing registers is possible, as well as having registers hold data about the overall completion state of the milestones. However, we still had a few that were used throughout the project. Besides the default modules that came with the project, such as UART, SRAM, VGA interfaces; we also created our own modules to assist us with repetitive code that we utilized in Milestone 2. We created a module called SampleCounter.sv that, when an enable signal is driven to, iterates through a block of 8x8 samples and calculates the SRAM addresses to retrieve the matching data from the SRAM. This module had to be created as reusing this code would be very repetitive inside our project file and would decrease the efficiency of the debugging process.

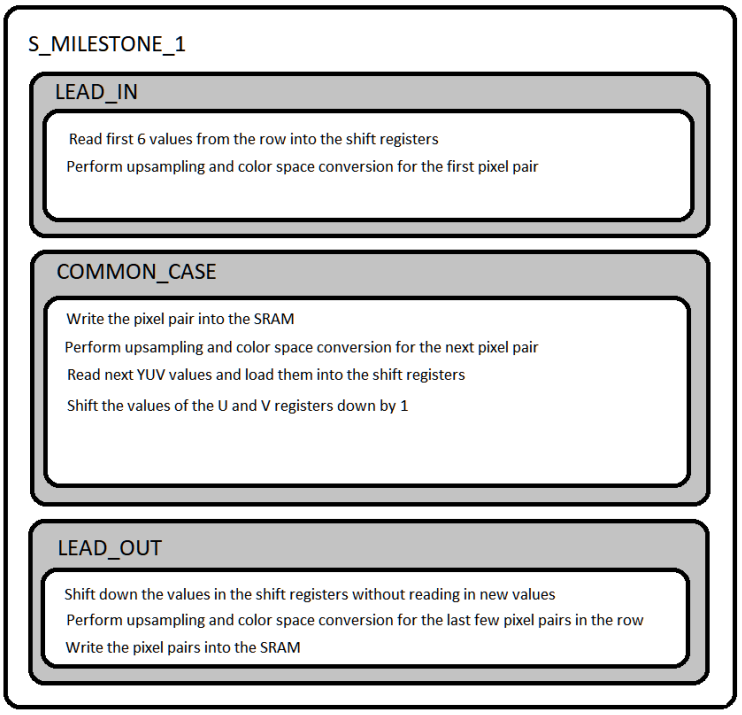
**Implementation Details**

The modules that we designed were our main project file module, and our SampleCounter module. For the SampleCounter module, it consists mainly of shift registers, adders and multiplexers. The module takes multiple inputs that we set outside the module, so that we can control when it counts, what the base address is, and the column/row index of the block we are calculating addresses for. When driven by a high enable signal, a six bit shift register we called the sample counter begins counting upwards (incrementing once per clock cycle) from 0 to 63. Then, we use two other 3 bit registers (rowAddress and colAddress) to hold information about the 8x8 block that we are calculating addresses for. We take the first 3 bits of the sample counter register as the "row" and the last 3 bits as the "column" of the block. Finally, we use bit shifting and adders to generate the address, and send it from the module as output logic.

In terms of the main project module, as stated above, is mostly composed of a hierarchical finite state machine (FSM) that we use to control what the module is currently doing. Our main project is broken up into the following hierarchical FSM:

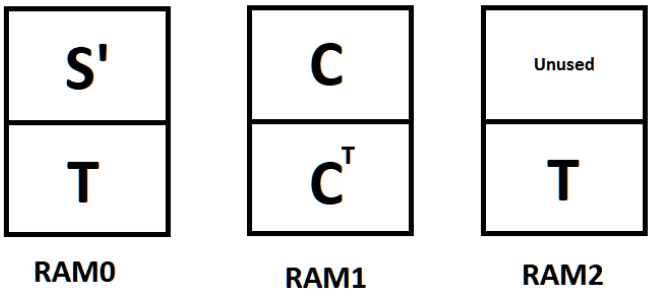



Our top state controls which milestone is currently being worked on, and the bottom state controls the logic within each milestone. This way, by changing the bottom state, we can move through the different states of our FSM and when the milestone is complete we can move out by simply changing the top state. When looking more in depth inside the Milestone 1 top state, we have grouped the bot states according to their function for doing upsampling and color space conversion.



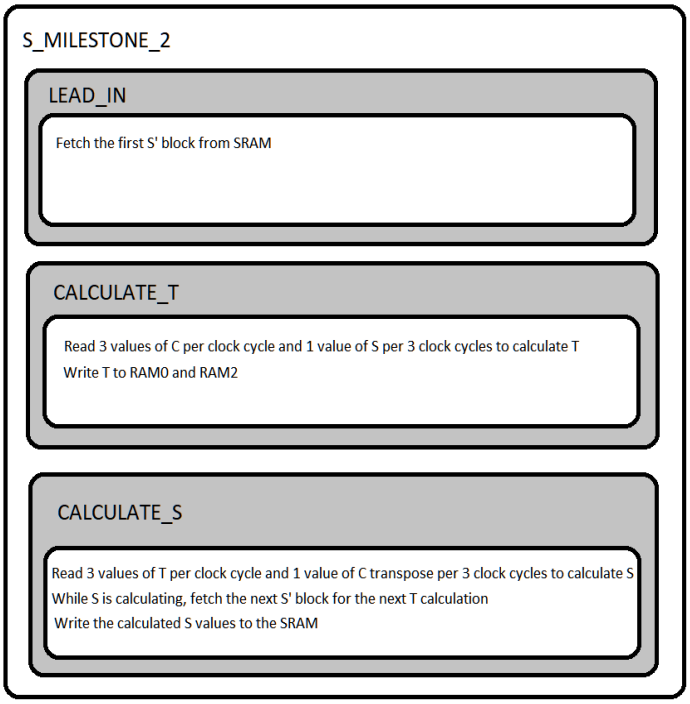

As seen above, our states within S_MILESTONE_1 are broken up into the following functions: lead in, common case, and lead out. The purpose of the lead in state is to read the first six U and V values for each pixel row into the U and V shift registers from the SRAM, read the first Y value, and calculate the first pixel pair. The most important shift registers here are the twelve 8 bit registers for U plus 5, 3, 1, U minus 1, 3 5 and V plus 5, 3, 1 and V minus 1, 3, 5. These registers hold the six values necessary for the upsampling calculation. In addition, the two 8 bit registers, Y and Y_buf hold the Y values that are read in from the SRAM and are necessary for the color space conversion calculation. Once the first pixel pair is calculated, the bot state moves into the common case, which is a group of nine states responsible for the majority of calculations. The common case is nine states since in every state of the common case, except for one, a multiplication operation is being performed on the multipliers for either upsampling, or color space conversion. Since the multipliers are in use for eight states out of nine, the utilization constraint of 85% is met. In addition, since we are doing two multiplications per common case cycle, for 16 operations in total, we also meet the interpolation filter symmetry constraint when multiplying our Y values with the constant coefficient matrix during color space conversion. Lastly, our common case state will move to the lead out state once the pixels have been calculated for most of the row. Once the common case state has calculated pixel 315 and 316 of the row, it moves to the lead out. Now, the operations that were present in the common case are repeated, except no new values are read, and the value in the U plus 5 register will continuously shift itself down. Once the last four pixels for the row have been calculated, the FSM moves into the lead in state, and the whole cycle repeats until the entire picture is finished.

For Milestone 2, when creating the memory layout, we decided to lay out our memory into the RAM as follows:




**Note: S', C and $C^T$ are packed, ie, there are two 16 bit data entries in one 32 bit location**

Since S', C and $C^T$ are packed, it allows us to read the required number of samples for all calculations at all times. In addition, since we store two copies of our T values across two RAMs, we can also fetch new S' values in the same state that we read T. This allows us to reduce one of the megastates that we would need to complete this milestone. When looking more in depth inside the Milestone 2 top state, we have grouped the bot states according to their function for leading in, calculating T and calculating/writing S:




Within S_MILESTONE_2, our lead in state uses the SampleCounter module mentioned above to calculate the addresses to read from the SRAM. Once 64 values have been packed into the first 32 locations of RAM0, the FSM enters the common case state CALCULATE_T. This group of 24 states will take the previously obtained S' samples and perform matrix multiplication with the C values from the first quarter of RAM1. Since C is packed, we only need to read two values per clock cycle to get the three we need, and the same for the S' values. After T values are calculated, they are stored in both RAM0 and RAM2, since we will need to read two T values per clock cycle in the next calculation. The purpose of the next group of 24 states, CALCULATE_S, is to use the previously calculated T

values and the $C^T$ values to calculate the S values. Since we have two RAMs storing T values, we can use the other port of RAM0 to write in the next S' block, as well as write the currently calculated S values to the SRAM as well. At the end of the S calculations, we repeat the process and loop back into CALCULATE_T until the entire pre-IDCT Y block is calculated. After that, we change up our address calculation formula, as the pre-IDCT U and V blocks are downsampled, and repeat until all Y, U and V data is written to the SRAM. For both CALCULATE_T and CALCULATE_S states, since we have 24 states and we are doing 24 multiplications within these 24 states to calculate one row of the resulting matrix, it takes 192 clock cycles. Even though we have a few transition states between large groups of states, we still meet the 85% multiplier utilization constraint.

When debugging Milestone 1 and Milestone 2, a variety of debugging strategies were used so that we could effectively find problems and resolve them. Utilizing the given testbenches, it would indicate to us in the console whenever there was an error or a mismatch between the provided and expected data. Using this, we could pinpoint where the error was in ModelSim and work backwards to find the root cause. For example, in Milestone 1, whenever we had a mismatch we would find the location in ModelSim where that value was calculated and then take the values in the registers and calculate them manually by hand; this way we could find math or sign errors that would crop up in our designs. In Milestone 2, we took this a step further and wrote a Python script to take an 8x8 hexadecimal matrix (from the SRAM.mem file) and convert it to signed decimals, which we would then put in MATLAB to verify if our matrix multiplication operations were correct.

In terms of decoding latency, in Milestone 1, we use nine common case clock cycles, and (a rough estimate) of 30 lead in/lead out states. This means, for a picture the size of 320x240, for Milestone 1 alone, we will have approximately 350,000 clock cycles at 20 ns per clock cycle, for a total of about 7ms. For Milestone 2, we use 64 clock cycles to pack S into RAM0, 192 clock cycles to calculate T and another 192 clock cycles to calculate S, for one block of 8x8. For the pre-IDCT data that was given (Y being 40x30 blocks and U/V being 20x30), we will have approximately 1,075,200 clock cycles at 20ns per clock cycle, for a total of 21.5ms. When looking at the critical path, we found that our design had no timing violations and our slack values were positive. The largest slack value we had was when we were driving read data from the SRAM into one of our multipliers. Our best explanation for this is that it takes time to look up and retrieve the data from the SRAM before it is moved to the multiplier.

| | | |
|---|---|---|
| Week 1 | Joined github classroom and transitioned from lab work to project preparation. | |
| Week 2 | Strictly read the project document and developed a theoretical foundation of the project's big picture. | Both group members read the project document together and brainstormed possibilities for approaching the project objectives. |

| | | |
|---|---|---|
| Week 3 | Designed a basic state table for M1 based on Week 2 conceptualization as well as lecture information. | Will created the excel document and filled in the labels given in the lab document while both group members designed the state table itself. |
| Week 4 | Implemented and debugged Milestone 1 until completion. | Mohamed:<br>- implemented the lead in and lead out states for Milestone 1<br>- noted down documentation during Milestone 1 to be used later on in the report writing process<br><br>Will:<br>- implemented the common case states for Milestone 1<br>- created labelled images to help with conceptualization of the requirements of Milestone 1<br><br>Both group members debugged the values that Milestone 1 calculated |
| Week 5 | Implemented and debugged Milestone 2 until completion. | Mohamed:<br>- implemented the lead in state for Milestone 2<br>- created the .mif files for each of the matrices<br><br>Will:<br>- implemented the calculate T and calculate S states<br>- created a Python script to assist with debugging matrix multiplication<br><br>Both group members debugged the values that Milestone 2 calculated as well as completed the report |

**Conclusion**

In this project, we gained valuable digital system design experience via implementation of the design objectives. Through the process of analysing the project document, attending lectures for key information, brainstorming initial theory, and implementing our ideas, we have learnt how to focus on project specifics and tackling individual tasks while remaining thoughtful of the bigger picture at all times. Another major facet of our learning experience was the debugging process. By constantly analyzing the output waves of our hardware and using several tools such as MATLAB and HxD in order to compare current and expected values, we learnt how to ‘think outside of the box’ and try tackling an issue from multiple angles rather than get stuck making minute changes repeatedly. [Milestone 1 completion: Nov 18, 2021 ‘Milestone 1 finished and commented’]