COE3DQ5 – Project Report

Group 12
Arjun Snider, Cameron Yap
snidea4@mcmaster.ca, yapc2@mcmaster.ca

November 28, 2022

To test our work please run milestone 1 project file in its folder, and milestone 2 project file in its folder, the milestone 1 folder contains a working version of milestone 1

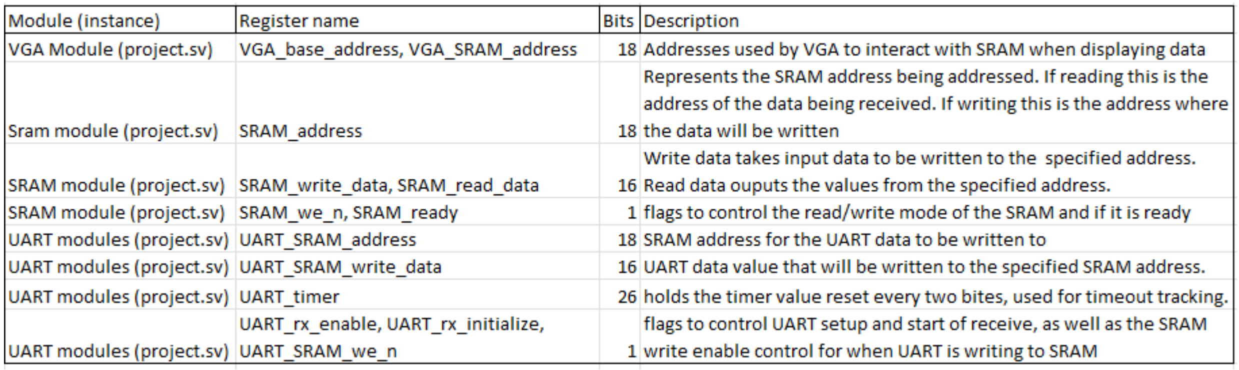
1 - Introduction

The COE 3DQ5 final project focuses on the decoding of images with the .mic16 compression format. We create the fully encoded image through the use of C programs provided to us for the project.Milestone 3 focuses on decoding the encoded bitstream from the fully encoded file, as well as dequantization, resulting in a large block of pre-IDCT data. Milestone 2 takes the pre IDCT data and performs Inverse Discrete Cosine Transformation on it. This transformation is performed on 8x8 blocks of data and results in the creation of Y, U, and V data. Milestone 1 takes the Y, U, V data and through the upsampling equation, interpolates the values for the odd U and V rows in the data. It then performs colourspace conversion on all the data, resulting in sets of RGB data for each of the pixels in the 320x240 pixel image. Through the course of our project we were able to successfully implement milestone 1, as well as create a full implementation of Milestone 2 that contains some flaw resulting in pixel mismatches. Due to time constraints and the unresolved errors in Milestone 2, we did not implement milestone 3.

2 - Design Structure

Our design can be split into several different modules: 1. UART module. 2. VGA Module. 3. SRAM Module. 4. Project Module containing milestones 1 and 2. 5. Dual Port Ram (DP-RAM) Modules. These modules are generally controlled in our top level FSM, with transition points between them. The exceptions are the SRAM module which controls the SRAM for the entire system and interacts with all the modules through its drivers in the project module, along with the Dual Port Ram module which is used by milestone two to store intermediate calculation data. Modules 1, 2, 3 and 5 were all provided to us for the project, with Module 1(UART) needing minor modification to stop it from removing transmission headers, and the DP-RAM module modified to take a parameter which defines its mif file initialization values. These modules were reused from previous labs and are the base units upon which the project is built. The UART module receives the incoming test data and uses the SRAM module to write the test data to SRAM. The DP-RAM module is used throughout Milestone 2 to store the blocks of data between calculations, and is controlled by the project module. The VGA module takes the final, decoded RGB data from SRAM and displays it on screen. The table below outlines registers used for Modules other than Milestones 1 and 2.

Module 4, the project module is the core file of the project which integrates all the modules together. It includes the top level FSM which controls modules 1 and 2, as well as the Milestones. It also contains an always_comb block that determines which values will drive the SRAM module based on the state of the top level FSM. While they are not defined in separate files in their own module blocks, Milestones 1 and 2 can be thought of in a similar manner to the other modules. They have their own registers which they use to complete their operations, they take input data and provide processed output data usually through the SRAM. They are controlled through the top level FSM just like Modules 1 and 2, and they have their own resources, such as multipliers and DP-RAM. We did not implement the milestones as separate modules because we were unfamiliar with creating module blocks, and had progressed quite far into the development of both milestones before we recognized they could be their own modules. As a result, we were hesitant to alter the organization of our design to make each milestone its own module, for fear of disrupting the functional parts of our design which had been extensively tested at that point.



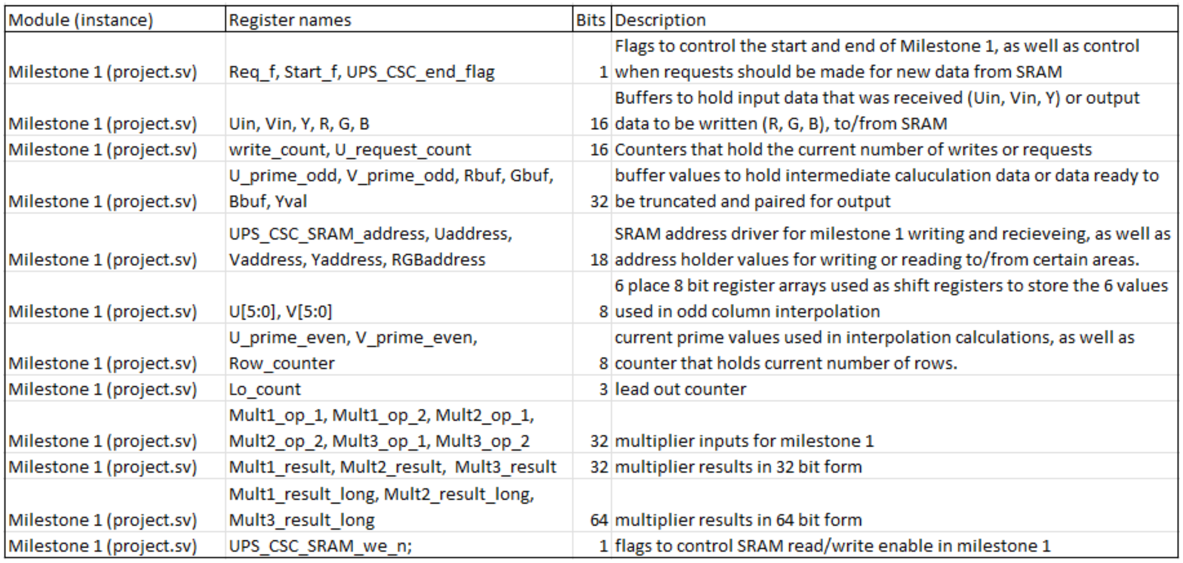
Module (instance)	Register name	Bits	Description
VGA Module (project.sv)	VGA_base_address, VGA_SRAM_address	18	Addresses used by VGA to interact with SRAM when displaying data
Sram module (project.sv)	SRAM_address	18	Represents the SRAM address being addressed. If reading this is the address of the data being received. If writing this is the address where the data will be written
SRAM module (project.sv)	SRAM_write_data, SRAM_read_data	16	Write data takes input data to be written to the specified address. Read data ouputs the values from the specified address.
SRAM module (project.sv)	SRAM_we_n, SRAM_ready	1	flags to control the read/write mode of the SRAM and if it is ready
UART modules (project.sv)	UART_SRAM_address	18	SRAM address for the UART data to be written to
UART modules (project.sv)	UART_SRAM_write_data	16	UART data value that will be written to the specified SRAM address.
UART modules (project.sv)	UART_timer	26	holds the timer value reset every two bites, used for timeout tracking.
UART modules (project.sv)	UART_rx_enable, UART_rx_initialize, UART_SRAM_we_n	1	flags to control UART setup and start of receive, as well as the SRAM write enable control for when UART is writing to SRAM

3 - Implementation Details

Milestone 1 - Hardware Structure

The hardware structure for the implementation of Milestone 1 uses three multipliers, multiple adders, comparators, counters, flags, shift registers, and registers to store and buffer data. The three multipliers are used to perform multiplication calculations with U, Uprime, V, Vprime, and Y data. The adders are wired to the address registers, counter registers, some of our data buffer registers. The adders are used to increment or decrement the address registers and counter registers, and they are used to add some of the results from the multipliers into data buffer registers in some states. The comparators in our implementation, denoted by several if conditions in code or hardware multiplexers, are used for handling clipping when we are setting our RGB registers that get written to SRAM with our RGB buffer registers and they check edge cases for when the RGB buffer is greater than 255 or less than or equal to 0. These comparators take the inputs 255 and our R, G, or B buffer value or 0 and our R, G, or B buffer value and output 255 when the buffer value is greater than 255, 0 when the buffer value is less than 0, or just the buffer value if it is between 255 and 0. The comparator outputs the clipping handled R, G, or B value and is wired with UPS_CSC_SRAM_write_data which writes the data to the SRAM. We use shift registers to shift out the most significant index in the U or V interpolation array, shift in the newly received value of U or V in the least significant index, and shift all of the U or V values to one index higher in the array. We have several flags in our implementation that act as select values for a multiplexer to perform certain actions based on whether the select bit is 0 or 1.

Milestone 1, as well as the other modules, depend on an always comb block at the end of the project Module. This block assigns the SRAM driver holders (such as UPS_CSC_SRAM_address) to their respective SRAM module values, based on which state the top level FSM is in. We implemented this block to resolve the multiple-driver errors we were encountering so that UART, VGA, Milestone 1 and Milestone 2 can all drive SRAM when their modules are active.



Module (instance)	Register names	Bits	Description
Milestone 1 (project.sv)	Req_f, Start_f, UPS_CSC_end_flag	1	Flags to control the start and end of Milestone 1, as well as control when requests should be made for new data from SRAM
Milestone 1 (project.sv)	Uin, Vin, Y, R, G, B	16	Buffers to hold input data that was received (Uin, Vin, Y) or output data to be written (R, G, B), to/from SRAM
Milestone 1 (project.sv)	write_count, U_request_count	16	Counters that hold the current number of writes or requests
Milestone 1 (project.sv)	U_prime_odd, V_prime_odd, Rbuf, Gbuf, Bbuf, Yval	32	buffer values to hold intermediate caluculation data or data ready to be truncated and paired for output
Milestone 1 (project.sv)	UPS_CSC_SRAM_address, Uaddress, Vaddress, Yaddress, RGBaddress	18	SRAM address driver for milestone 1 writing and recieveing, as well as address holder values for writing or reading to/from certain areas.
Milestone 1 (project.sv)	U[5:0], V[5:0]	8	6 place 8 bit register arrays used as shift registers to store the 6 values used in odd column interpolation
Milestone 1 (project.sv)	U_prime_even, V_prime_even, Row_counter	8	current prime values used in interpolation calculations, as well as counter that holds current number of rows.
Milestone 1 (project.sv)	Lo_count	3	lead out counter
Milestone 1 (project.sv)	Mult1_op_1, Mult1_op_2, Mult2_op_1, Mult2_op_2, Mult3_op_1, Mult3_op_2	32	multiplier inputs for milestone 1
Milestone 1 (project.sv)	Mult1_result, Mult2_result, Mult3_result	32	multiplier results in 32 bit form
Milestone 1 (project.sv)	Mult1_result_long, Mult2_result_long, Mult3_result_long	64	multiplier results in 64 bit form
Milestone 1 (project.sv)	UPS_CSC_SRAM_we_n;	1	flags to control SRAM read/write enable in milestone 1

Milestone 1 - Finite State Machines

The Milestone 1 module FSM is entered when the top_state FSM reaches the state S_UPS_CSC_TOP. In the S_UPS_CSC_TOP state, if the UPS_CSC_end_flag is equal to 1, the top_state transitions to the next state. The UPS_CSC_end_flag is initially set to 0 in the Milestone 1 FSM. When all the RGB data has been written to SRAM, the UPS_CSC_end_flag is set to 1 and top_state exits S_UPS_CSC_TOP and enters the next state. The Milestone 1 FSM has a group of 8 lead in states that request U, V, and Y data from the SRAM and fill the U and V interpolation arrays with the requested data. We chose to have 8 lead in states because it let us request all the data needed as well as fill the U and V interpolation arrays with enough values to set up our common case. The Milestone 1 FSM has 8 common case states with a start flag to indicate whether the program should write to SRAM with RGB data or not. This start flag starts with the value 1 and is checked in the first 4 states. This flag is changed to 0 after the 4th state of the first common case since we do not have any data to write to SRAM after our lead in. The first 4 states use the U and V arrays for multiplication operations that output Uprime odd and Vprime odd values. These first 4 states also handle some clipping and writing of RGB data to the SRAM. For the next 2 states, the common case then uses the Y value, and Uprime and Vprime even values to calculate R, G, and B even values which are stored in buffers and arithmetically shifted by 16 to account for the division by 65536. Since 76284 is always multiplied by Y[15:8] in the matrix multiplication, we chose to use one multiplier to calculate this value and stored the result in a register called Yval. For the last 2 states, the common case uses the Y value, and Uprime and Vprime odd values to calculate R, G, and B odd values which are stored in buffers and arithmetically shifted by 16. Yval changes here to the multiplication of the Y[7:0] bits and 76284 for the calculation of the R, G, and B odd values. The reason our common case is split up into calculating Uprime, Vprime, R, G, and B values is to maximize the utilization of our multipliers as per the requirements for Milestone 1. A request flag is used to check if we need to request data for U and V since one request of U and V data can generate 4 RGB values. This was a design decision to add this request flag in order to shorten our common case to 8 states instead of 16. The Milestone 1 FSM also has 8 lead out states that are reached when we reach our last common case. These states are very similar to our common case except we do not request any more data from SRAM. We then have 3 final lead out states that perform the last writes of R, G, and B data to SRAM.

Milestone 1 - Latency of Decoding Process and Time Spent

Based on the compilation report, Milestone 1 implementation takes 1019 registers, with 682 in the project module, 66 in pushbutton control (unused but part of base project provided), 56 in SRAM module, 74 in UART Module and 141 in VGA Module.

Using the worst case timing analyzer slow 1200mV 85C model, the critical path is from Mult1_op_1[10] to U_prime_odd[17], in 5.491ns. This is likely part of an interpolation calculation, generating an odd value bit. This false within a safe range given our 50Mhz clock has a period of 20ns, so the delay is less than a clock.

The total time taken to run a full simulation of M2 including an output to VGA was 33536135 ns in modelsim testing, which represents approximately 1,676,807 clocks.

Milestone 2 - Hardware Structure

The hardware structure for the implementation for Milestone 2 uses 2 multipliers, comparators, adders, counters, flags, and registers to hold buffer data and addresses. The 2 multipliers are used in each part of our FSM with maximum utilization as per our design. We decided to store C arbitrarily in our DP-RAM0 in addresses 64-95 as horizontal pairs (storing values 0 and 1 in the matrix together). We came to this conclusion when deciding how to store C as we were trying to store C individually as 16 bit values in 64 places in DP-RAM0 or in vertical pairs. We decided on storing C in horizontal pairs since we could fully utilize the multipliers in both calculating T and S as when we go to calculate T, we can calculate 2 rows of T at the same time using our horizontal C pairs and our vertical Sprime pairs. We also decided to store Sprime as vertical pairs (with value 0 and 8 being stored together in a 8x8 matrix) to again fully utilize our multipliers when calculating T. In doing this, we only use 50% of DP-RAM0. We stored T in singular 16 bit values from addresses 0-63 in order to make calculations easier. We tried an alternative where we stored T in horizontal pairs and tried to multiply T transpose with C in singular pairs but in the end it did not give us full utilization of our multipliers. We decided to store S in horizontal pairs as well since we are writing S into SRAM as horizontal pairs anyways, we thought that storing them in DP-RAM1 when calculating S was the best option. We decided to store Sprime and C together in DP-RAM0 and S and T together in DP-RAM1 since we need to request certain values from one DP-RAM and write to another and we wanted to make sure there were enough ports for us to do that. After much deliberation, we decided on these configurations for Sprime, C, T, and S due to meeting utilization requirements for our multipliers and convenience.

Module (instance)	Register names	Bits	Description
Milestone 2 (project.sv)	IDCT_end_flag, write_s_f, fetch_sp_f	1	Flags to control the end of Milestone 2, skipping the write S on first loop and not fetching at the end of block
Milestone 2 (project.sv)	Sp_data_buf, C_data_buf;	16	Buffers to hold input data that needs to be paired up before being used or to isolate parts of the pair for different clocks
Milestone 2 (project.sv)	IDCT_SRAM_write_data;	16	driver for data to be written to SRAM in milestone 2
Milestone 2 (project.sv)	R0_address_0, R0_address_1, R1_address_0, R1_address_1, C_address, Sp_address, T_address, S_address	7	Address drivers for each of the 4 ports of DP-RAM and address holders for different sections of milestone 2
Milestone 2 (project.sv)	R0_data_in_0, R0_data_in_1, R1_data_in_0, R1_data_in_1, R0_data_out_0, R0_data_out_1, R1_data_out_0, R1_data_out_1	32	data inputs and outputs for each of the 4 ports of DP-RAM
Milestone 2 (project.sv)	IDCT_SRAM_address, Sp_read_address, S_write_address, read_address_offset, write_address_offset	18	SRAM address driver for milestone 2 writing and recieveing, address holder values for writing S or reading S', and offsets for reading Y, U or V values from S'
Milestone 2 (project.sv)	TS_buf_0, TS_buf_1, Calc_buf_0, Calc_buf_1, Calc_buf_2, Calc_buf_3	32	buffers to hold intermediate calculation data before it is truncated and writen to DP-RAM
Milestone 2 (project.sv)	UV_transition;	2	controls how the read and write addresses are caculated based on if we are reading from/ writing U, V or Y data blocks
Milestone 2 (project.sv)	IDCT_Mult1_op_1, IDCT_Mult1_op_2, IDCT_Mult2_op_1, IDCT_Mult2_op_2	32	multiplier inputs for milestone 2
Milestone 2 (project.sv)	IDCT_Mult1_result, IDCT_Mult2_result	32	multiplier results in 32 bit form
Milestone 2 (project.sv)	IDCT_Mult_result_long, IDCT_Mult2_result_long	64	multiplier results in 64 bit form
Milestone 2 (project.sv)	IDCT_SRAM_we_n, R0_wren_0, R0_wren_1, R1_wren_0, R1_wren_1	1	flags to control SRAM read/write enable in milestone 2, as well as DP RAM read/write enables

Milestone 2 - Finite State Machines

The Milestone 2 module FSM is interacted with by the top_state FSM when top_state enters S_IDCT_TOP in which it exits that state under a similar condition to the Milestone 1 FSM. A flag is used to indicate that Milestone 2 is done and that the top_state FSM will enter the next state. The Milestone 2 FSM is split up into 5 different parts, fetching Sprime, calculating T and writing S, calculating S and fetching Sprime, only calculating S, and only writing S. We chose to split up the Milestone 2 FSM like this because we wanted lead in parts for fetching Sprime and calculating T, a common case part that would loop until all the pre-IDCT block data is read with calculating T while writing S and fetching Sprime while calculating S, and lead out parts that would calculate S and write S for the last block of data. All parts of the Milestone 2 FSM are designed to fulfill maximum utilization of our 2 multipliers.

The first part for fetching Sprime is the first of our two lead in parts for the Milestone 2 FSM. This group of states consists of 3 lead in, 2 common case, and 3 lead out states. The lead in states begin requesting data from the pre-IDCT address while incrementing and decrementing the Sp_counter through each request which controls which SRAM_address is being requested data from. We decided to increase Sp_counter by 8 on our first request and decrease it by 7 on the next request to get vertical pairs of Y, U, or V data in which we could then multiply by the horizontal pairs in the C data in DP-RAM0 to make full use of our utilization when calculating T. The first common case state then receives data from previous requests, stores it in the Sp_data_buf buffer and requests new data. The first common case state is where we check if a full 8x8 block of data has been requested using the counter Sp_counter in which we switch to the lead out states if a full block has been requested. The second common case state receives data, writes that data to DP-RAM0 in addresses 0 to 31 in a vertical pair of data with the buffer data being in bits 32-16 and the newly received data in bits 15-0. The lead out states receive data, buffers it, and writes it into DP-RAM0.

The lead out to fetching Sprime data transitions to the lead in states for calculating T. Since we are calculating T and writing S in the same states, the write_s_f flag is used to check whether it’s the first time calculating T in which we should not be writing S. The write_s_f is set to 1 initially and after the first calculating T part, it is set to 0. The 5 lead in states start writing S constantly and request C and Sprime data from DP-RAM0. Since C is stored in horizontal pairs and Sprime is stored in vertical pairs, we request data iterating the Sp_address by 1 and C_address by 4. When writing S data, we ensure to check for possible clipping and write data accordingly in both the lead in states and common case for calculating T and writing S. There are two common case states that loop for calculating T and writing S multiply the received C and Sprime data while also buffering the Sprime data for use in the next state. This first looping common case state also checks whether we have calculated T for all the rows and columns, reached end of a column of C values, or we have reached the end of a row of Sprime values and adjusts the addresses accordingly. The second looping common case state multiplies T with received C values and uses the buffered Sprime data to calculate the bottom values of T. Once we have reached the end of a column for C, we transition into 3 lead out common case states that write the T values into DP-RAM1 while calculating more T values using the same method as the looping common case states. These then transition back to the first looping common case state. When writing S to SRAM in this part, S_counter iterates whenever we write an S value to SRAM and when that counter is not less than 32, we stop writing S. This condition exists because writing S will end before calculating T is done. The final lead out states of this part consist of just calculating T values and writing them to DP-RAM1 while also checking whether we have calculated the T values for the last pre-IDCT block of data in which we will transition to our lead out parts.

The next part calculates S and fetches Sprime data at the same time. The fetching of Sprime data in this part is the same as the first part of the Milestone 2 FSM except a Sp_counter keeps track of how many requests we made and if we have requested all of our Sprime data, we stop requesting data from SRAM. There are 6 lead in states that first request data from the SRAM for Sprime to fill in DP-RAM0 and then we request C data from DP-RAM0 and T data from DP-RAM1 and multiply the pairs of C data and T data together to add together to get S data. Since C data is stored in horizontal pairs, and we use C transpose to calculate S, C transpose is stored in vertical pairs in which we request them similarly to how we requested them when calculating T. There are two counters that keep track of whether we need to transition into writing S to DP-RAM1 using S_mult_counter or whether we are finished iterating through a whole matrix of T values using S_state_counter. There are two common case states that loop and calculate S values while requesting C and T values always. The first common case state checks whether we have finished calculating a full matrix of S or whether we have calculated a full two rows of S. The second common case state stores the first vertical pair of S values when they are calculated in Calc_buf_2 and Calc_buf_3 since we want to write S to DP-RAM1 in horizontal pairs instead of vertical pairs. The next 3 common case states buffers the next set of vertical S pairs in TS_buf_0 and TS_buf_1 singularly and combines the buffered vertical pair of S values and the buffered pairs in horizontal pairs and writes them to DP-RAM1. The lead out states for this part just use the buffered pair of calculated S values and newly received S values and also combine them into horizontal pairs to write to DP-RAM1 without requesting new C or T values.

The next part of just calculating S happens only when we are calculating S for the last pre-IDCT block in which we only calculate S like in the previous part. This part uses the same logic as the previous part except excludes the fetching of Sprime data.

The last part of the lead out parts for the Milestone 2 FSM is just writing S. This part uses 4 states, 1 lead in, 2 common cases, and one lead out state. These states just constantly write S to SRAM and in the lead out state we iterate UV_transition by 1 which tells the program that we’re either reading from the Y, U, or V block of pre-IDCT data and we’re writing to Y, U, or V in SRAM. This part transitions back to the idle state.

Milestone 2 - Latency of Decoding Process and Register Usage

Based on the compilation report, Milestone 2 implementation takes 1587 registers, with 1250 in the project module, 66 in pushbutton control (unused but part of base project provided), 56 in SRAM module, 74 in UART Module and 141 in VGA Module. Given that milestone 2 includes milestone 1, we can estimate that the additional registers required for milestone 2 are around 568 registers.

Using the worst case timing analyzer slow 1200mV 85C model, the critical path is from Mult1_op_1[9] to U_prime_odd[22], in 5.491ns. This is likely part of an interpolation calculation, generating an odd value bit. This false within a safe range given our 50Mhz clock has a period of 20ns, so the delay is less than a clock. As a result of our implementation, the longest critical path always runs through milestone 1 operations, and in the timing analyzer they make up the top 100 longest paths.

The total time taken to run a full simulation of M2 including an output to VGA was 50304135ns in modelsim, which represents approximately 2,515,207 clocks.

Verification Strategy and Debugging

In both milestones we used a very similar strategy when debugging and verifying. We mostly used the different test bench files to test whether we had any mismatches with pixels or not and if we did, we would use the waveforms in Modelsim to see what the problem was. The problems sometimes were very hard to understand as some values were very slightly off and others were working just fine. We would use the waveforms, go to our project.sv file and see where that was being calculated in the code and see if we needed to handle any edge cases or clipping. For milestone 1 we had some clipping issues which we discovered in the waveforms when our pixels were mismatchgin.To resolve it we had to implement $signed arithmetic shifts in order to maintain the sign in our calculated value and clip the written value when the value we were trying to write to SRAM was negative or larger than 255. For milestone 2 we first ran the program in modelsim and found a couple of problems with our T values. We then checked the waveforms and saw that we were calculating T wrong since we didn't use the $signed function. We then changed this in our code. Most of the time, we could find small mistakes in the code once we checked where the waveforms had our mistakes, and were able to solve issues such as addressing edge cases

Time Usage and Development Efforts

Week	Progress	Arjun Contribution	Cameron Contribution
Week 1	Read through project documentation	Read through documentation	Watched lectures on project
Week 2	Write state tables for milestone 1	Revised state table from initial write	Wrote initial state table for CC
Week 3	Implement milestone 1 and debug	Debugged in lab with Cameron on clipping	Debugged in lab with Arjun on clipping
Week 4	Work on milestone 2 state tables	Worked on conceptualizing how to store Sprime, T, S, and C in DP-RAM	Worked on designing state tables for calculating, fetching and writing
Week 5	Implement milestone 2	Implemented clipping logic and flags for writing S	Implemented Calculating T, S and fetching Sprime states as well as lead out

4 - Conclusion

We had a very good learning experience through this project. We were able to experience working on large format development projects in a new hardware description language which had a steep and challenging learning curve. Unfortunately we were unable to complete the last two milestones but we made significant progress and learned a lot along the way.

5 - References

Class notes Nov. 16, 18 and 23, 3dq5-2022-project-description and project report guidelines document.