COE 3DQ5 - Project Report

Group 44, November 29, 2021

Zain Talat - talatz@mcmaster.ca | Hassan Mahdi - mahdih2@mcmaster.ca | Parthav Patel - patep62@mcmaster.ca

I. Introduction

The objective of this collaborative project is to decode a 320 x 240 pixel image through a hardware implementation. Provided compressed data stored in the SRAM is decoded through a series of stages, from which the original image will be restored. Processing begins with lossless decoding and dequantization to revert the given data back to the frequency domain. This is followed by inverse signal transformation, producing a set of Y/U/V values. The Y data represents the brightness levels of each pixel, while U and V provide colour. However, since the human eye is more sensitive to brightness, double the Y values will be produced as opposed to the U/V values. This is done during compression to maximize efficiency while maintaining image quality. Finally, the Y/U/V values are processed through a series of up sampling (through interpolation) and colour space conversion to restore the original raw RGB data which is then sent to the VGA controller to be displayed as a .ppm file.

II. Design Structure

In this project both Milestone 1 and 2 were implemented in the top-level FSM in order to design an image hardware decompressor. In this top level FSM three data busses were used in order to flow data within the module; SRAM_address, SRAM_we_n and SRAM_write_data. Other modules used throughout this project were UART_SRAM_Interface and VGA_SRAM_Interface.

III. Implementation Details

1.1 - Milestone 1: Up Sampling and Colour Space Conversion.

The hardware constraints applied to this segment of design are as follows. We are only able to use two multipliers, which must be utilized in at least 85% of all clock cycles in this milestone. As a result, the processes of up sampling and colorspace conversion must be done simultaneously in order to meet the utilization constraint. Where U’ and U represent the up sampled and down sampled values respectively, for pixel j on the image. The down sampled U values can represent a 160 x 240 pixel image, whereas the up sampled U’ values represent a 320 x 240 pixel image. For example, in processing pixel 1, the first partial product yields 21*U[-2]. This value obviously does not exist, and thus must be handled differently as opposed to the interpolation of pixels away from the image edges. As a result

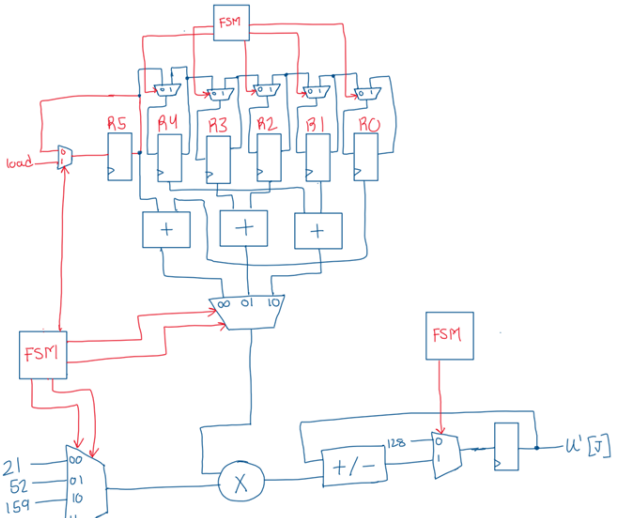
of these design requirements, Milestone 1 is split up into a series of lead in cases, lead out cases, and a common case. The designed FSM will rotate through these stages (performing up sampling and color space conversion) until all rows of pixel data are produced.

1.2 - The Common Case States:

During execution, the hardware will be spending the vast majority of it's time and computations in the common case states. As such, meeting the 85% utilization constraint is imperative, and will allow a large amount of leniency with the design of our lead in and lead out cases. Every iteration of the common case will produce a pair of pixels. The interpolation is done differently for even and odd pixels, therefore it makes sense to process one pair at a time consisting of an even and an odd pixel. So, when considering the number of multiplications that needs to be done in each iteration of the common case, 6 are required for the interpolation of the odd U/V values (when exploiting the symmetry in the equation), and then 5 are required for each even/odd pixel during colorspace conversion. This totals to 16 multiplications per common case, which will require minimum 8 clock cycles since we are given 2 multipliers to use. We can use them at most, 2*k times per common case, where k is the number of clock cycles. If we have 9 clock cycles, we have a utilization of 16/18 which is 88.89%. However, with 10 clock cycles we have a utilization of 16/20 which is only 80%. As such, our common case consists of 9 clock cycles instead of 8 to allow for some leniency.

Beginning with perhaps the most influential design decision made here, is the way we approach the shifting of U/V registers. The design consists of 6 U and 6 V registers which will store a series of U/V values depending on which iteration of the common case we are on. 6 registers are necessary because each odd interpolated U'/V' value is a function of 6 U/V values. Exploiting symmetry, it is always going to be the opposite registers whose values will be added together. Therefore, R5 and R0 are passed through an adder, just as R4 and R1, and R3 and R2. The sums are then sent into a 4 input MUX from which the sum selected will be the input of the a multiplier. The other input for the multiplier is provided by a single 4 input MUX that will pass through the appropriate coefficient. The partial product is then stored into an accumulator (MAC unit). Once all the partial products are computed and accumulated, the resulting value is sent for colourspace conversion, and the accumulator is reset to a value of 128. It is only after all the partial products are accumulated, will the U/V registers shift and load the next value into R5. The implemented hardware circuit is outlined below for reference.

Note that the exact same hardware implementation is used to compute V'. Since the implementation uses only one multiplier, the U' and V' values are computed simultaneously. This takes 3 clock cycles, performing 6 multiplications. Once these clock cycles finish,



we can move onto the colourspace conversion. The even pixel is processed first because the U'/V' registers require an extra clock cycle to be loaded. So instead of waiting an extra clock cycle, we can just begin with the even pixel and use both multipliers for it. All Y/U/V registers are fed through MUXs which select the inputs for the multipliers. The other inputs to the multipliers are provided by 8 input MUXs which select the corresponding coefficient. We chose to then use accumulator registers representing red, green, and blue, to store the partial products eliminating the need to manage a buffer register. These computations require 5 clock cycles. An extra clock cycle is used at the end to load the last products into their respective registers, and then we loop back around to begin the next pixel pair. It is important to note however, that the final write for each pixel pair (green odd and blue odd) cannot be done in the last clock cycle and must instead spill over to the first clock cycle of the next iteration.

Reading U/V values from the SRAM becomes slightly complicated in the common case where we are loading only one U/V value every iteration. Since each location in the memory stores a pair of U/V values, it makes sense to only read every other iteration of the common case. However, this requires a buffer register to store the odd U/V value that does not get loaded into the registers in that iteration. Instead, we chose to read each U/V value twice and simply index the most significant byte or least significant byte for the even or odd value respectively. Essentially, an additional mux is connected to the U/V shift registers which pass through either the MSB or LSB of the read data register, depending on a 1 bit flag. This bit is inverted after every iteration of the common case. Furthermore, only on iterations where the LSB (odd U/V value) is loaded, will the data counters used for addressing the SRAM, be incremented. Y values are read in every iteration of the common case as required for each pixel pair.

1.3 - The Lead in/out Cases

As mentioned earlier, the purpose of these lead in/out cases is to cover the unique computations for edge pixels. However, the lead in cases will also load the first 6 U/V values for every new row. As a result, no additional reads are necessary during the lead out cases. The lead out states will transition directly into the lead in states where 3 reads are done, filling all 6 registers. The multipliers are not being used during these clock cycles, but we have a quite a bit of leniency with the utilization in these states. With reference to the interpolation formula, there are two pixel pairs that require unique computations are the beginning of each row, and three pixel pairs at the end. The lead in/out states compute just these pixels.

In terms of the critical path, we can observe from the Timing analyzer that the critical path starts from the node Mult_op_2A[17:0] and ends at the node AccumulatorV[30]. The reason why this path has the longest delay is due to the fact the largest value in Mult_op_2A is 132251 which is 18 bits ([17:0]), this register goes through a series of calculations that are

needed to calculate data for V'[odd] before it reaches AccumulatorV[30], thus it will be the longest path.

1.4 - Verification Strategies

A very large amount of time was spent debugging and verifying proper functionality of milestone 1. The lead in states were coded in Verilog first, and was verified before moving onto the common case and lead in states. We wanted to confirm that the basic arithmetic and reads/writes were being performed correctly. Thus, when programming the rest of the states, the same lines of code were used to prevent some basic errors from occurring. With the testbench allowing for 10 mismatches at most, we went through fixing most of the errors quickly. When debugging, the waves proved to be extremely useful. We would compare what we saw on the waves against our state table and most of the time, the mistakes would present themselves shortly after. However, after coding all three cases together we started running into much more interesting mismatches. Firstly, we saw that in some spots we were writing large values while they should have been zero. We knew these probably had to be a problem with our arithmetic. Upon revision of certain lectures, we saw that we were not clipping properly, and those large numbers were actually supposed to be negative. Obviously, we cannot be writing negative numbers to the SRAM so these had to be clipped to zero and the issue was solved. Most notably however, an odd series of mismatches occurring at seemingly random pixels in the lead out states. Some mismatches would be thousands of pixels apart, however all were in the lead out cases. This was particularly confusing as we wondered why so few pixels were causing issues. After a conversation with Dr. Nicolici, we realized that the pixels near the edge of a display are often times very similar. This could mean that even though there was something wrong with the way we coded the lead out cases, most of the produced pixels would seem to be correct. This perfectly explained the issue we were facing and now that we knew there was a problem with the lead out cases, we got to work debugging. Due to the nature of the errors, we knew that there was probably something wrong in the way our U/V registers were set up, or how we used them in our computations. So, our first step was to assess the contents of the U/V registers during the lead out cases where the mismatches were occurring. Upon comparing the contents against what we saw in the SRAM file (using a hex viewer), we realized that the registers had shifted up one additional time. Once we figured this out, fixing the issue was just a matter of changing the indexing of the U/V registers when doing the computations for U'/V'.

Week 1	Watching Lecture videos, learning about the project as a whole, and beginning to understand Milestone 1. We all did learning separately for the most part.
Week 2	Formulated the state tables for Milestone 1. Worked together on this.

Week 3	Began coding Milestone 1 in verilog according to the created state table. Worked separately coding the states, coming together to debug.
Week 4	We spent most of this week debugging many issues with Milestone, going to office hours frequently to ask questions. We began to brainstorm ideas for M2 towards the end of this week.
Week 5	We spent roughly the first half of the week conceptualizing/finalizing ideas for Milestone 2. Then spent the rest of the week coding/debugging through Milestone 2. We debugged the majority together.

Milestone 2: IDCT:

DPRAM Organization:

For this milestone we were given permission to use 3 Dual-Port RAMs that are 32 bits wide by 128 bits long (addresses) (4096 bits) . We chose to use the first half of DPRAM 1, addresses 0-63 to load in a single S' value at a time per each address of the DPRAM. Since the DPRAM has 32 bits per address and S' is only 16 bits we had to use signed extension replicating the 15th of bit SRAM_read_data 15 times and contacanting that with SRAM_read_data and the passing this to the write data register port to load in one S' value at a time. Furthermore the bottom half of the ram was used to pack three 8-bit S values in one single address which can hold up to 32 bits. Which are then read in MSA. DPRAM 2 was used to hold T even values and DPRAM 3 was used to hold Todd values. We repeat this process 7 more times for the remaining Matrix A rows.

Matrix Multiplication Method:

For the Matrix Multiplication of Matrix A x Matrix B = Matrix C, we iterate through one row of A for all 8 columns of B to produce 8 output values which consist of the first row of C; which takes 8 clock cycles per output value. Since we are allowed to utilize three multipliers in our design we could compute 3 output values per 8 cc cutting down CC latency.

SRAM Address Generation:

As for the LEADin case of FS' we have created a Finite State Machine which consists of 4 states. The first three leading states M2_FETCH_0, M2_FETCH_1 and M2_FETCH_3 in which each of these states we set the M2_SRAM_we_n to be able to read from the SRAM. Furthermore, we set M2_SRAM_address by passing preIDCT_offsetY, write_count,i to two adders which passes the result to the M2_SRAM_address increasing the address based on an offset of where preIDCT_offsetY counter is which points to the correct address. Write_count is a register which holds the value for which row we should be reading from the [8x8] block.This register should be passed to an adder to be incremented by 320 addresses everytime we compute the first 8 results which represent the first row in the output matrix (Refer to the Matrix Multiplication Method above). For example, the reading for row 0 to row 7 of the S' Matrix from the SRAM should go like this 0,320,640….. 2240. The 'i'

register is passed to an adder with 8 and the result should write back to the ‘i’ register. This process happens every time the Row Block is passed to an adder and incremented by 1 and the results are written back to the Row Block. The Row Block Counter register is passed to anadder to be incremented every time the last state in this FSM M2_state <= M2_FETCH_3 reads the remaining 61 values from the SRAM by doing the same operation on M2_SRAM_address as mentioned above. Once Row Block reaches 39 the preIDCT_offsetY register is loaded with the previous value of preIDCT_offsetY with an adder that increments 2240.

Compute T:

As mentioned earlier, the computed T values are written to the top halves of the 2nd and 3rd DRAMs. As for how we store the matrix C, we chose to use 3 64 to 1 MUXs from which we could index 3 values to use for multiplications. The compute T multiplications are done through a series of 6 states. 3 of which correspond to the downwards traversal of the C matrix, doing 3 columns in the first and second, and 2 in the last. Furthermore, 3 states correspond to writing the values in the accumulators after each of the 3 compute states. The FSM spends a total of 9 clock cycles in each of the 3 compute states to traverse down the C matrix, and then spends 1 clock cycle in each of the write states.

Compute S/Write S:

To compute S, we first need to read values T values that we stored in DRAMs 2 and 3. This is done through using a 1 bit flag that flips every clock cycle to alternate from reading DRAM1 and DRAM2. The C matrix is then traversed in a way such that C is transposed. In order to then write the computed S values (packed into 3/location) that were stored in the bottom half of DRAM 1, we use a buffer and a pattern of write cycles to appropriately pack 2 Y values per location.

Verification:

One of the most things we had to verify was when we were verifying if all 3 DPRAMs had the correct values in the correct addresses after A compute/Fetch/Write task. This is where we ultimately spent the most of our time debugging. We also implemented a system similar to Lab 4 in which everytime we run the testbench 3 DPRAM mem files output in the sim subfolder which helped us a lot when verifying.

Conclusion:

Although we could not get all the way through Milestone 2, this project overall has been an incredible learning experience. We spent a very large portion of the past few weeks debugging and analyzing errors. As we progressed we got better and better at being able to seek out and pin point errors in code relatively quickly. We also became very familiar with the project management process as a whole, beginning with brainstorming and getting our ideas down on paper, then moving into state tables and visualizing hardware circuits.

Since our Milestone 2 is incomplete, the proper Github commit message and date for the completion of Milestone 1 is as follows: “Completed Milestone1” Nov 24 202