COE3DQ5 – Project Report
Group 05
Muaz Akhtar akhtam17@mcmaster.ca
Abdelmoniem Hassan hassaa73@mcmaster.ca
November 28, 2022

Introduction

This project was about designing and implementing hardware to construct an image decompressor with the bigger picture being more so about working with larger digital systems and being able to integrate smaller parts together to make a functional product. These smaller parts include modules such as UART transmitter/receiver, FSM, and reading and writing to SRAM's and DRAM's. The project was broken into 3 different milestones, each of which was further broken down into multiple smaller modules that dealt with the different aspects of a image decompressor including modules such as mathematical computations and storing and reading results from different RAMs available which will be discussed in further detail.

Design Structure

At a high level, our design consists of 6 modules working together which include the UART transmitter/receiver, milestone 1, milestone 2, the top-level FSM, dual port RAMs, and the VGA controller. The top-level FSM's purpose is to essentially facilitate the flow of the required process for decompressing an image and controls where the SRAM_address, SRAM_write_data, and RAM_we_n get their assignments from. For example, if the top-level FSM is currently in the milestone 1 module (meaning it is performing upsampling and color space conversion), then the SRAM address, write data, and write enable will be controlled by the M1 SRAM address, write data and write enable. This ensures that the correct addressing is applied depending on which milestone is being performed. The milestone 2 module is where the IDCT transformation is applied to the YUV blocks in which 4 smaller tasks are done in sequence to perform the transformation which includes fetching an 8x8 block (S prime), computing the T matrix (S prime * C), computing the S matrix (C transpose * T) and writing the resulting matrix to the SRAM. The dual port RAM modules are used here o temporarily store the results of these smaller tasks (for example, storing the T matrix result in the DRAM to buffer it). The result of this milestone 2 module is to fill the SRAM with the down-sampled YUV data which will be passed into milestone 1. The milestone 1 module is where the up-sampling and color space conversion is performed to obtain all pixel data of the image. In this module, we already have the up-sampled data for the even pixels, however, perform interpolation for the odd-numbered pixels. The final aspect is performing the color space conversion which consists of matrix multiplication with the up-sampled YUV values to convert them to the corresponding RGB values for each pixel of the image and storing the results in the SRAM. Once all the RGB is stored in the SRAM, the VGA module will read the RGB data in the SRAM and display the pixels on the display.

Implementation Details

The following section will describe the different milestone modules in depth and how exactly they were designed and tested.

Milestone 1

The milestone 1 module consists of an FSM which includes 11 lead-ins, 7 common states, and 7 lead-out states. The idea with our design implementation is to perform the lead-in, common and lead-out cases for each horizontal line in the image. We repeat this process for all 240 lines which is all kept track of using a register called 'linelinecounter' that increases by 1 each time in our last lead out case. Once this register reaches the value 239, the FSM instead of going back to our lead in states from the final lead out state will go to the final lead out state of milestone 1 where the final GB value is written to the SRAM before the image decompression is done. Our lead in case, we initialize our first set of reads where we perform 2 U and V reads and 2 Y reads. The reason for performing 2 reads for U and V is due to the required pixels needed in the up sampling of the odd U and V values (keeping in mind the U and V values are in even pairs). For example for the up sampling of U1, the equation requires pixel data for pixels 0, 2, 4 and 6. This data is buffered into different registers that keep track of each component in the up-sampling equation (for example Ujp5, Vjm3). These registers essentially act as shift registers because for the following U up sampling, the values are shifted to the left in the equation (U[j+3/2] takes the previous value of U[j+5/2]) and so once up sampling is complete, these values are shifted to the left and the Up5 register (contains the U[j+5/2] value) is updated with a new read of U or V respectively. The purpose for this lead in case is to account for the edge cases where the values U[j-5/2], U[j-3/2] and U[j-1/2] are negative and so they must use the pixel data at the edge of the image for up sampling. Also keeping in mind that for one U and V read, we require two reads of Yas the U and V reads are in even pairs. For this reason in our design, we buffer the U and V values and have a flag called 'readcycle' that alternates each common cycle to be able to use the first even value in one common cycle and the other in the second. This helped us resolve issues with reading U and V too fast compared to Y. The first two sets of RGB values are computed in the lead in state. The common case essentially consists of computing two sets of RGB values, one even set and one odd set. The way our multipliers are set up for up sampling is by adding terms with a common coefficient and then multiplying with that coefficient. For example, the U[j-3/2] and U[j+3/2] are added together and multiplied by 52. The reason for this approach is because this allows us to be able to perform color space calculations for the even RGB set in parallel as those pixels do not require up sampling. The registers 'UprimeEven', 'UprimeOdd', 'VprimeEven' and 'VprimeOdd' are buffer registers where each up-sampling component is added to so that eventually, we accumulate all the components and obtain the U' and V' components needed in the color space conversion calculation and this is split between the even and odd values. Similarly, there are 6 RGB buffers called Rbuf, Gbuf, Bbuf, R, G and B. The Rbuf, Gbuf and Bbuf registers are reserved for the even RGB set while R, G and B are for the odd RBG set. These registers act as buffers as well where each component of the color space conversion math is stored in so that eventually, they accumulate all required components to obtain the final RGB values for the respective pixel in question. Finally, we begin to write these RGB values to the SRAM in pairs as soon as they are computed. A MUX is used for clipping

before passing in the data to the M1_SRAM_write_data register as some of the values could either be negative (clipped to 0) or larger than 255 (clipped down to 255). Since all arithmetic was done on 32 bits signed, the MUX checks bit number 31 to see if it is 1 which would indicate it is negative and writes 0 for that R, G or B value if so. If this is not the case, another MUX is used to check if the value is larger than 255 by using the bitwise OR operator on bits 30 to 24. The reason for this is that if any of these bits is a 1, that automatically indicates that the value is greater than 255 and the value is clipped down to 255. If this is not the case either, than the value from the RGB buffers is used and passed in to be written. In terms of the FSM, there is a register called 'j' which keeps track of which set of which set of pixels we are currently on. In the last common case, we check to see if j is equal to 311 before moving to the lead out case and the reason is because this is where we start to approach the end of the line and so interpolation will have to be done by using the edge of the image pixel data. In each line there are 320 RGB sets required and so the final 4 sets (keeping in mind one even and one odd set is calculated in a cycle) are performed in the lead out case. The lead out case consists of performing the final reads to calculate the final U', V' and RGB values and perform the final writes to the SRAM for the line of the image in question as explained above.

In our design, in our 7 common states, since we used 3 multipliers, that comes out to 21 total uses of the multiplier across the entire common case. Out of the 21, our design used 17 which results in an average utilization of about 81%. Throughout the process of implementing milestone 1, our main method of debugging pertained to using model sim to view the contents of our registers in different states as well as mapping out the flow of our design on paper and performing the math calculations on paper as well to compare with model sim values. We knew that there were 4 main aspects where our design could be failing which includes reading of values, up sampling calculations, color space conversion calculations and the writing of values. We compared our math on paper to the values seen in model sim which helped us uncover all the bug related to the math portion of this milestone. For the reading and writing of values, most of our bugs were from addressing issues which we were able to solve by going through the first few iterations and seeing if the values we expected were matching the values that were in the SRAM.

Milestone 2

The approach we took for milestone 2 was a divide and conquer approach. We decided to break the problem into 4 smaller tasks and then implemented a top-level FSM for milestone 2 that facilitates when each task is performed. The 4 tasks are fetching a S' 8x8 block, computing the T matrix, computing the S matrix and writing the S matrix to the SRAM. Note as well that since the C and C transpose matrices are constant, we decided to hard code the C matrix into the bottom portion of the first DRAM and hard code the C transpose matrix as a MUX in our design to avoid complex indexing of C to get C transpose values.

Fetching the 8x8 S' Matrix

In our implementation, we decided to fetch two rows of the matrix at a time and write the values in column pairs. For example, we write S0 and S8 together in the DRAM, S1 and S9 and so on. This means we read one value from the first row and then one from the second until we have

read 16 values where then we move on to the next two rows. The reason for this comes into play when we compute T. In our lead in cases, we initialize the first 3 reads from the SRAM and then move the common case. In the common cases, since we get one value at a time, in one common case we buffer the value in a register called 'Spimebuffer' and in the next when we get the second value, we then write the buffered value and the new value that just came in to the register called 'F_write_data_0_a' (which indicates DRAM 0 port A while Fetch S' is happening) that will write the data to the DRAM. Two registers called 'columnIndex' and 'rowIndex' keep track of what row and column are currently being processed within the current block. A MUX is used to check if the values of these registers are both 7 (indicating the end of the 8x8 matrix) and if so, we move to our lead out case otherwise we start our common case again. The lead out case essentially consists of buffering and writing the last pair of the matrix. In our last lead out case, we increment a register called 'ColumnBlockIndex' which keeps track of which 8x8 matrix from the matrix is currently being processed. A MUX is used to check if the value of this register is 39 (because there are 40 8x8 matrixes per row) and if this is the case, we reset the columnBlockIndex back to 0 and increment the register 'rowBlockIndex' which keeps track of which row of 8x8 matrix's we are on.

Compute T

In our implementation of this subtask, since we wrote S' as column pairs, we iterate through all the columns of the C matrix for the two rows of S prime and repeat this for the other 3 row groups of the S' matrix. The reason the S' values were written as column pairs is because then they can be multiplied by the same C matrix coefficient at once. For example S0 and S8 both have to be multiplied by C0. Because the RAM is dual port, we use 1 port to read the S' values and the other to read the C matrix values. The result from each of the multiplications are added to two registers called 'Tbuf' (for the first row in question) and 'Tbuf2' (for the second row in question). These registers continue to accumulate the sums of the matrix multiplication and after 8 multiplications will have the first two T values for the respective block of S'. After going through all the columns in C, this will result in an entire two rows of the T matrix calculated. This process repeats for all the S' groups (2 rows per group again) and each S' group goes through every column of C resulting in the full T matrix at the end. Each T value is written to the DRAM individually as 32 bits

Compute S

In our implementation of this subtask, we need to compute C transpose multiplied by T and since the T values were written individually, we iterate through all the columns of the T matrix for the two rows of the C transpose matrix and repeat this for the other 3 row groups of the C transpose matrix. In doing so, this allows us to multiply the C transpose values with the common T value between them. For example, C Transpose 0 and C Transpose 8 both get multiplied by T0 and so this can both be done in a clock cycle resulting in the calculation of one value for each of the first two rows in the S matrix and so on. Port A on the second DRAM is used to read the T values using the 'S_address_1_a' register and C transpose values come from the implemented MUX using the 'c_transpose_address' register. The result from each of the multiplications are added to

two registers called ‘S_buf’ (for the first row in question) and ‘S_buf2 (for the second row in question). These registers continue to accumulate the sums of the matrix multiplication and after 8 multiplications will have the first two S values for the respective block of C Transpose. After going through all the columns in T, this will result in an entire two rows of the S matrix being calculated. This process repeats for all the C Transpose groups (2 rows per group again) and each C Transpose group goes through every column of T resulting in the full S matrix at the end. Each S value is written to the DRAM as column pairs (for example S0 and S8) as a total of 32 bits together. The reason they were written like this was a design choice made by us where we directly write the calculated values together as soon as they are calculated.

Write S

In our implementation of this subtask, since we wrote the S matrix values as column pairs, we are essentially reading a value from the DRAM and buffering it using the ‘writseSbuffer’ register and then when the next read value is available, the two row pairs will then be written to the SRAM and the second-row pairs will be buffered and written in the next cycle. We use port B on the second DRAM to read these S pair values using the ‘W_address_1_b’ register and then slice them correspondingly and write them to the SRAM using the ‘W_write_SRAM_data’ register. This is essentially what occurs in our common state of this subtask. The lead in states initiates the first two reads and buffers the values and the lead out states stop reading and write the final two values to the SRAM.

Top Level FSM Of Milestone 2

The top-level Finite State Machine dictates which of sub-tasks are currently running through a series of enable and done flags. Each of these sub-tasks will remain in the idle state until its respective enable flag is toggled. The subtask will then reset its counters and move into the lead in states and carry out its functions. Once the sub-task enters its final state it will toggle the done to be true and return to its idle state. This done flag is then read within the top-level FSM and allows us to proceed with the next task. Running multiple sub tasks in parallel is as simple as toggling both their enables to be true and waiting for each of their respective done flags to be toggled. This allowed us to divide milestone two into smaller easier to design processes that can run independently and in parallel rather than implementing all of them into one large and convoluted state machine. Each sub-task had its own register for accessing the SRAM and the DRAM memories and using the top-level's state we allocated the actual port's value to that from the task that was supposed to used it, this helps ensure that our reads and writes are correct even when multiple tasks are running in parallel. This FSM will run until all the down-sampled Y, U, and V components are write memory. The M2 State Machine has two lead-in states, two common mega states, and two lead-out states. The two lead-inlead in states fetch S’ and compute T respectively. The first mega-state computes S and fetches the next S’. The second mega state will write S and compute the next T. The lead-out states compute S and write S for the final block. The Lead-ins, Common mega states, and lead-out states run for Y, U, and V. The transitions from common states to lead-out states is determined by the index of the current 8x8 block being evaluated.

Timing Analysis

After viewing the timing analyzer in Quartus, it was noted that the critical path was about 16.37 ns based on the worst-case timing paths in the timing analyzer. This path originates from the bit 17 in operand1 to bit 31 in operand4, both from our milestone 1 implementation. After considering our design, this critical path likely originates in the portion where we compute our odd U' values which would be the 'UPrimeOdd'/ 'VPrimeOdd' registers in our design. In our design where we compute and set our odd U'/V' values in our common case in milestone 1, we pass one of our multipliers into this register. The multiplier uses the 'operand4' register to perform the multiplication in question.

Project Progress

Milestone 1 was successfully completed, and successfully passes all test benches, and displayed correctly when tested on the board. Can be tested using this commit. (if for some reason this does not work try the latest commit but transition into M1 directly from the UART states.)

The architecture and implementation of Milestone 2 is complete. However, we are seeing mismatches while running the simulation. The following image is generated from the testbench after running Milestone 2. Clearly the Y, U, V values are close to the expected values and the general content of the image is recreated after the IDCT, however there is clear distortion in the image. We suspect that there is a bug in our calculations, However we were unable to identify and fix this bug before the deadline.

Project Progress Timeline

Week	Muaz	Abdel
1	Begin to read project description and formulate state table for M1	Begin to read project description and formulate state table for M1
2	Further implementation of state table for M1 and beginning implementation of M1	Further implementation of state table for M1 and beginning implementation of M1
3	Start to debug M1 and adjusting state table and code as need be	Finish implementation of M1 and start to debug M1

4	Begin to create top level design and state tables for M2	Begin to create top level design and state tables for M2 and start implementation of M2
5	Debugging M2 and adjusting state tables and code as need be	Finish implementation of M2 and debugging of M2

Conclusion

Above all this project was a very intense yet interesting project to work on. Our group felt that as the days went on, we were slowly making more and more progress and learning a lot along the way making it easier to understand and start work for other milestones. Our group really enjoyed being able to take a handful of smaller modules and being able to make them work together to be able to create a larger digital system. There were no issues between group members as we were able to work together successfully for at each point in this project. The only other collaboration our group had was with other groups to discuss and clarify guideline meanings as well as how to set testbenches up.