COE3DQ5 – Project Report
Group 43
November 20, 2016

lysv@mcmaster.ca

kamadae@mcmaster.ca

Introduction

The goal of our final project was to create a hardware implementation of an Image decompressor to successfully decompress images formatted in the Mcmaster image compression specification. Compressed images will be sent from the computer to the Altera DE2 board where it will be stored on an external static random access memory(SRAM) and then will be decompressed in stages and displayed on the monitor. The first stage of decompression is lossless decoding and dequantization in which the encoded bitstream is decoded in blocks and the Y(brightness), U(blue chroma), and V(red chroma) planes are be stored in the memory as values in the frequency domain. The next step is to perform a signal transform on the YUV blocks to convert them back to the spatial domain and then store back In the SRAM. The last stage in the process called upsampling and colorspace conversion in which the downsampled U and V blocks are interpolated back to their original size which will enable the YUV elements to be converted into RGB format

Design Structure

The creation of this image decompressor can be made much easier by partitioning our device into modules. This is important as it gives us the ability to reuse modules that have been supplied to us in the past. Modules that we have reused for this project include:

-PB_Controller
-VGA_SRAM_interface
-VGA_controller
-UART_SRAM_interface
-UART_Receive_Controller
-SRAM_controller
-convert_hex_to_seven_segment
-tb_project_v2 and tb_project
-tb_SRAM_emulator

The most important modules that were re-used for the design are the ones that have the labels “VGA”, “SRAM” and “UART”. The UART is what enables us to communicate data between the computer and the DE2 board so without it the entire project would not be possible. The “SRAM” modules are also vital as it it where all of our picture data will be stored and written too. The final key module that was added were the VGA modules which enable us to display our converted images to the monitor and verify that our design functions as intended. Other important modules include the “tb” modules which are used for simulation purposes. Without these it would be incredibly difficult to troubleshoot our design. The other included modules “PB_controller” and “convert hex to seven segment” are not key to the design and were included mainly to provide the user with some form of feedback.

All of these modules must be connected by a single module in order to have the design work together. Furthermore, custom modules must be added to the design as well in order to perform operations on the data. Finally, dual port RAMs must be created to store data in milestone 2.

-project
-milestone1
-milestone2
-dp_ram0
-dp_ram1

The project module is our top level module and it what connects all other modules together, as it has control over which components are active and which that aren't. The "d_ramX" modules are created using the Quartus MegaWizard and are needed to store the intermediate data of the spatial domain signal transform. The milestone 2 module is responsible for performing a part of the image decompression as it produces spatial values from values in the frequency domain that can be utilized in milestone 1. The milestone 1 module handles the last stage of decompression, as it interpolates the downsampled data and computes the RGB pixel values. After this, the values are written back into the SRAM to be used by the VGA_SRAM interface.

Implementation Details

Milestone 1

For this milestone the first decision was how the multipliers would be utilized to meet the usage constraint. From this point we would build our common case followed by the lead-in and lead-out cases. The organization that we settled on uses 3 multipliers at a time to compute U and V. Each of these computations requires six multiplications so 2 clock cycles are needed for U and V each. The conversion between YUV and RGB requires 5 multiplications so to implement this is the fewest amount of clock cycles possible and the highest multiplier utilization three multipliers are used in the first clock cycle and 2 multipliers are used in the second. One multiplier is unused in each computation but this format will still allow us to meet our utilization constraint. In both the interpolation calculations and the RGB conversion a multiply and accumulate (MAC) unit is implemented as a resource efficient solution for determining the result of our computations.

U odd		V odd		RGB even		RGB odd	
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X		X	

Fig. this table is a representation of our multiplier utilization. "X" represents a multiplier in use

After determining the organization of the the multipliers we can move forward and design our states to fit. Our lead in state consists of 16 states and is much longer than the common case due to the fact that multiple U and V values must be read from the SRAM and stored into buffers before interpolation can occur. After interpolation begins, more UV addresses are read and loaded into buffers as preparation for the common case. Once interpolation is complete RGB conversion begins and the first pixels are written into SRAM. A delay state was added between states 10 and 11 in order to properly coordinate the address reads from the SRAM. At end of the lead-in case another delay state was included so that the the last 2 bytes of the second pixel could be written before the common case begins.The lead-in case is followed by the common case which is where the bulk of the computations for milestone 1 occur. The common case consists of 9 states, the first four of which are used solely for interpolation, and the last five being used for RGB conversion as well as writing to the SRAM. when new UV data is read from the SRAM it is stored into a shift register which stores all the UV values needed for interpolation as well as RGB conversion. One important aspect to remember that for each U or V data read, there is an interpolated value as well which means that there are twice as many values to work with. Because of this, U and V values are only read every other clock cycle. Furthermore because each UV read data has 2 values you cannot compute interpolated values using the same process every time. The first time, the most significant value could be the first byte of one buffer, and the next time it could be the the second byte of another buffer. Both this issue and the one before it can both be solved simultaneously by using a multiplexer that enables different actions based on what state you are in. Noting that extra U and V addresses were read, it is important to keep track of when the last value is loaded into the shift registers as it will read the end before the Y data and the end of the common case. When this point has been reached, The UV address stops incrementing and the shift register the will load the final value into the shift register from this point forward. Once 318 of the pixels have been computed and written into the SRAM, the lead out case begins. The lead out case is very similar to the the common case and follows a very similar process but does not require the shift register or any address reading as the data has already been loaded.

In this milestone we wrote all of the states out before verification and debugging began. Using ModelSim, many mistakes became apparent at first and a number of small changes were made. When the bugs became more difficult to find, more timing waves were added to the simulation, which helped to narrow down where the problem was occurring. Once we have narrowed down a few places where the problem is, we scan through the timing diagrams looking for anomalies until the error is found. One example of this is when we noticed that the near the end of the common case, UV address was continued to increment after it was supposed to stop. This issue was tracked back to the shift registers that load the read data as well as the register that acts as a counter for the UV addresses. Once this issue was resolved and ran simulation, the image was noticeably better, and much closer to correct. This debugging process was used again after clipping had been implemented but we continued to observe a large number of pixels with very high color values. To resolve this we added the multiplier results to the waveform and cross-referenced our written values to hand calculations. After scanning for values that had a value, “FF” we found one that did not match the multiplier results. Because of the way we had

implemented clipping there was a bug that prevented any negative values from being clipped and the values would be interpreted as a very large unsigned value. Once the root issue was discovered. We redesigned our method of clipping and eliminated the bug.

Milestone 2

For milestone 2, the first design decision to be made is again in regards to the usage of the multipliers available. There are a number of ways to implement matrix multiplication using two multipliers, the design chosen multiplies two rows from the first matrix with a column from the second matrix. Many of the designs do not have a utilization or latency advantage over one another so the reason for choosing to use the multipliers in this format as it was a less complicated design and therefore less prone to error. After deciding the multiplier design the next step is to format where each matrix will be stored in the two dual port RAMs. Where each matrix is positioned is important because three values will need to be read from the two RAMs for the matrix multiplication, but can only perform two reads per RAM. this means that any matrices that are multiplied must be placed into a different RAM. The design chosen separates the frequency domain(S') matrix from the cosine matrix(C), and then loads the product of these two separate from the cosine matrix. Implementing this format will allow us to smoothly implement matrix multiplication without conflicts.

Similar to the first milestone, this module is separated into an initial case, a case that repeats and performs most of the operations, and a closing case. The initial case is used to fetch the frequency domain(S') values from the SRAM and store them in dual port RAM 0.

The common case starts by computing the the matrix multiplication of S' and C whose product(T) is then stored into a dual port RAM. The matrix multiplication is implemented using the method described above with the partial products stored in a buffer using a MAC unit. The next step is to compute the matrix multiplication of C transpose and T to complete the signal transform with the product(S) written into the RAM. This multiplication operates using a similar method to determining the T matrix but the columns of C are multiplied instead of the rows because it is a transpose matrix. The final part of the common case is to write the converted S values into the SRAM so they are ready to be used for milestone 1. This process will repeat until all values have been converted into the spatial domain.

It is important to note that due to multiplier utilization constraints that the common case must start its repetition before the end of the case that is currently running, causing some overlap to

occur. This overlap creates a “megastate” which can either fetch or write values while also computing the T and S matrices. The lead out case will start when the last T matrix has been computed and will be used to compute the last S matrix and write the last block of S values back into the SRAM.

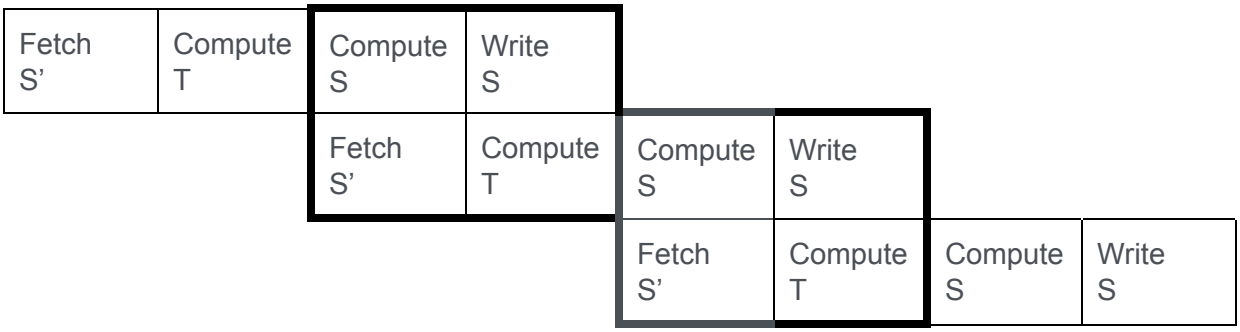

Fig. the states used in milestone 2. The blocks with the thick border represent the common case Still need to discuss latency and verification(debugging) so also cover what doesn't work and why you think the reason is.

Milestone Two is generally implemented, some aspects of the design still faces bugs and issues. The fetch S’ state, where SRAM is in a read mode and a dual port is writing, is functioning properly as required. This can be verified by running a simple simulation and cross reference read values from the initialized SRAM data. The SRAM address is read in a sequence that represents an 8x8 matrix and begins two clock cycles prior to reading the dual port ram0 address. Once the SRAM data is read, it is written to the address location of the dual port in a sequential manner. Since the hex editor displays data per byte and our SRAM stores two bytes per location, the y-offsets are doubled. The first pre-idct value in the hex file will stored in an absolute address offset of 153600. After the fetch S’ state, the compute T state begins. Unfortunately, once two values of matrix T is completed there is an overlap of reading and writing to the dual port ram. The probably arises when the address_a[1] is shifted by 320 address locations and address_a[0] equals 0 and address_b[0] equals 8. TS_accum will hold the partial sum of the previous calculation and instead of the new one. Besides, compute_t appears to contain no other issues. Compute_s is similar to compute_t in terms of the approach to the problem and debugging the code. Write s is essentially the same as fetch S’ with minor changes. We did not manage to run the entire milestone 2 implementation because the test bench throws an error-catch regarding correct information being inputted into the SRAM.

The measure of latency can be approximated by interpreting the amount of clock cycles required per module then multiplying it by the period of the clock cycle. Milestone 2 uses 64 clock cycles when fetching S’ because there are 64 elements in an 8x8 matrix. Computing matrix multiplications would take 512 cycles but since our design utilizes 2 multipliers it takes 256. Then it takes 32 clock cycles to write S. The total clock cycles for milestone 1 is the sum of the clock cycles of the lead in, common case and lead out.The consumption of time for milestone 1 is shorter than milestone 2 because there are twice as many clock cycles.

Milestone 2	Milestone 1
$64 + 256 + 256 + 32 = 608$ *cycles/block* $320/8 = 40 : 240/8 = 30$ $40 * 30 = 1200$ *blocks* $1200 * 608 = 729600$ *cycles* $20ns * 729600 = 14.59$ *ms*	$320 * 240 = 76800$ *pixels* $76800/2 = 38400$ *common case loops* $16 + 38400 * 9 + 10 = 345626$ *clock cycles* $345626 * 20ns = 6.91$ *ms*

Week	**Progress**
1	Reading project document and design of state tables(Eric/Simon)
2	begin coding Milestone 1 lead-in and common case (Eric/Simon)
3	Finish first draft of Milestone 1, simulations(Eric/Simon)
4	
5	Debugging milestone 1(Eric) revising Milestone 1 code(Simon), Milestone 2(Eric/Simon), Report(Eric/Simon)

Conclusion

Even though we were unable to complete all of the milestones this project has been a valuable learning experience as we have had to learn a great deal about the designing a complete project in verilog. There was so much more to learn in comparison to the labs because we were required to design entire components to a program without a template setup for us already. This required some time getting used to as we needed to pay attention to every detail of program instead of just a few specific operations. Using the previous labs as reference we took time to better understand them so that we would know what works and what doesn't. Furthermore, through simulation and many hours debugging, We were able to learn what works, what doesn't and the reasoning behind all of it.