COE3DQ5 – Project Report

Wednesday Group 38
Yazdan Rahman - Kevin Cheung
rahmay1 - cheunk20
November 30th 2020

Introduction

The objective for this project was to implement a custom image decompression specification in hardware. The custom digital circuit created in this project takes compressed data and converts it back into the original image, which is then displayed onto a monitor. To complete this we broke the project down into 3 separate steps/milestones.

The first step of this process would have been done in milestone 3, where we would have taken the compressed image and decoded the bitstream into blocks of data. This data would then be dequantized into a transformed image. In milestone 2 we then transformed the image from the spatial domain into the frequency domain to recover the downsampled image. Finally in milestone 1 we used colorspace conversion and interpolation to turn the downsampled image back into its RGB form to be displayed on a monitor.

Design Structure

Milestone 1:

In terms of milestone 1, we broke down the project requirements into 3 sets of states, Lead In, Common Case, and Lead Out states. These 3 sets of states were broken down further into primary modules which are classified as Reading from SRAM (Read), Computing U and V values (Compute U/V), Computing RGB values (Computer RGB), and Writing back to SRAM (Write).

In the Read Module, reading addresses are determined and the values of U, V, and Y are first obtained. It then sends the U and V values to the Compute U/V Module for calculations and sends the Y values to the Compute RGB Module. In the Computing U/V Module, the values of U_{odd}, U_{even}, V_{odd} and V_{even} are computed. These are then sent to the Compute RGB Module, where the values of RGB are calculated. In total, 6 values are calculated, 3 even and 3 odd values of R, G and B. These 6 values are then sent to the Write Module to be written into their respective SRAM memory blocks.

Milestone 2:

For milestone 2 we separated the process into four separate modules, Fetch S', Compute T, Compute S and Write S. The approach we decided to go with was to incorporate Fetch S' and Compute T together in one "Mega State," and Compute S and Write S in another "Mega State."

In the first "Mega state" Fetch S' retrieves the Y, U or V values from the SRAM, which then get stored in a DP (dual port) RAM. At the same time, these values are then multiplied with the C matrix to turn them into T values, which is in the second part of the "Mega State." Storing the T values in another DP RAM we then move onto "Compute S," where we calculate the final S values by multiplying them by a transposed C matrix. Finally, in Write S, these values are then written back into memory to be used in the next step of the decompression.

Implementation Details

Milestone 1:

To allow for simplicity and efficiency in the code, we split the process into 3 states. All 3 states are implemented in the always_ff block. The first state, the "Lead In" state, spanning 14 states/clock cycles, is used to start the first border approximation calculations. This allows for Reading, Writing, Computing U/V, and Computing RGB to be done in one iteration in the next "Common Case" state. This state is used to continuously read pairs of YUV values on the odd clock cycles (1,3,5), computing U/V and RGB values, and writing 2 sets of RGB values on even clock cycles (2.4.6). The common case iterates even through border cases (For example at 319/321), where approximations are triggered by flags and an increment variable which reset every 319 occurrences. The final state is the "Lead Out" state, where it is used to finish the last pixel write by itself. The Lead Out State consists of 4 clock cycles where the RGB addresses are incremented, the last RGB values are written into memory, and the Stop Bit is asserted to finish off the Milestone 1 implementation.

Each set of RGB values takes a total of 4 common cases to be computed. In the first common case the YUV values are read. The next common case the odd U and V values are upsampled with multiplication, using 2 of the 4 multipliers (32-bits). In the third common case the other 2 multipliers are being used to multiply the RGB with the previously upsampled odd U and V values, with the products being stored in 32-bit accumulator units. These RGB values are then written in the final common case, completing the set.

The Read and Write Modules are implemented using 4 registers which enumerate the Y, U, V, and RGB addresses separately by incrementing (adders are used) and setting them to the SRAM address register. There are 6 more registers which store the Y, U, and V values for both the even and odd index set obtained from the SRAM read data input wire. To read the

necessary YUV values for computation in the correct clock cycle we read Y every common case, and U and V every other common case. The values which are used for writing are computed in the always ff block but the writing itself takes place in always comb. The reason for this is due to the 0 cc latency of the always comb block.

In a single common case there are 12 total multiplier computations for the odd V and U values, and 10 total computations for the even and odd RGB values. This means that in every common case all the multipliers are being used in every clock cycle with the exception of the first clock cycle, where only 2 multipliers are being used, resulting in 91.6% utilization for every common case.

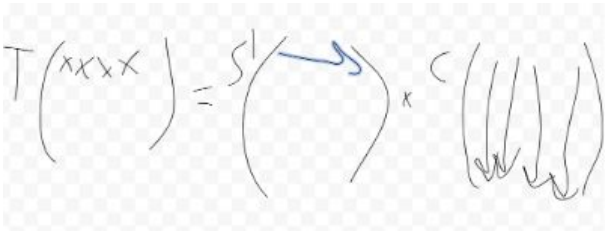
In total, 14 cc is spent in the Lead In State, 230394 cc is spent in the Common Case State, and 4 cc is spent in the Lead Out State for a total of 230412 cc. With each clock cycle being 0.02 us, that leads us to an expected run time of 4,608.24us. The simulation ends at 4608.815 us, which is in agreement with the theoretical value.

This design decision could also have had an alternative linear approach, where we compute each operation completely before computing the next. To do this all 4 multipliers would be used to compute U_{odd} first, then in the next clock cycle 2 would be used to complete the rest of U_{odd} and 2 would be used to compute V_{odd}, and so on until everything is computed.

Milestone 2:

To implement the “Mega States” we first start with Fetch S’. In this state we have sets of 8 repeating clock cycles to read and store the S’ (Y, U and V) values from the SRAM. As the calculations for this milestone require 8x8 blocks of data, we read 8 values at a time. These 8 values are written into the first DP RAM, RAM 0, where it will be used for calculations in the next step. The S’ values are only read every other 8 clock cycles, as we must allow time for the computations in the next state, Compute T.

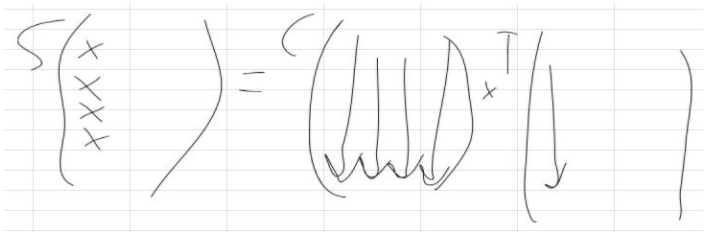
The next stage, Compute T, happens in the same clock cycles as Fetch S’. The computations are done in rows of T, shown in the figure to the right:



As each of the blocks are 8 values wide and we only have 4 multipliers, we would need 16 clock cycles to calculate a full row. In the first set of 8 clock cycles the values of S’ are directly read from the SRAM, and the values from the corresponding C matrix are read from the second DP RAM, RAM 1. These values are then immediately multiplied over the next 8 clock cycles to produce the first 4 values of T in a given row. The multiplications happen with each individual S’ value getting multiplied by all 4 C values and adding them to their respective accumulator. These values are then written into a third DP RAM, RAM 2. The values are written in pairs, 2 in the last clock cycle of the same set, and 2 T in the first clock cycle in the

next set. For the last 4 values in the row, the values of S' are read from DP RAM 0, which were stored in the Fetch S' step. These values are multiplied and written in the same way as the previous set, finishing the row of T values and continuing onto the next. This mega state happens for 130 cc (clock cycles), 128 cc for the calculations and 2cc for the final write and transition into the next "Mega State."

The next "Mega State" starts with Compute S. In this stage we compute the values for S to be written to memory. To achieve this we use a similar technique as seen in Compute T, with a slight variation seen to the right:



Instead of computing by the row, we must compute by the column as we chose to compute T and Fetch S' during the same clock cycle. As we multiply the T matrix by C transpose, calculating a value of S is computed from the sum of products of a row of T by a row of C. To achieve this, the T values from the previous step are read from RAM 1. The values of C are still stored DP RAM 1, which are read and multiplied by the corresponding T values. Following the same multiplication pattern as Compute T, 4 values are calculated over 8 clock cycles, and the full column of 8 values are fully computed over 16 clock cycles.

In the final Write S stage we take the values calculated in the previous computation and write them into the appropriate spot in the SRAM. To write these values to memory they must be clipped and paired. As we calculate the values by the column, the writing only happens every 16 clock cycles, as columns must be paired up. To achieve this, the first column is stored in DP RAM 0. Once the first 4 values of the next column are calculated, the first column values are paired with its corresponding second column value, and clipped to be sent to the SRAM over the next 4 clock cycles. This occurs again for the final 4 values of that column. Once the pair of columns are fully shipped, the next "first" column is calculated and stored, and the pattern repeats.

For 1 iteration of both common cases, 4 cc is spent in the Lead In State, 130 cc is spent in the Fetch S' and Compute T Mega Common Case State, and 130 cc is spent in the Compute S and Write S Mega Common Case State for a total of 264 cc. With each clock cycle being 0.02 us, that leads us to an expected run time of 5.2828 us. The simulation ends at 5.8328 us, which is in agreement with the theoretical value.

Another approach we considered was having Read S' and Compute S in one "Mega State," and Write S and Compute T in another. After asking a TA for assistance, he proposed the current design. We decided to go with this approach as the design was linear, and as such it was more intuitive to us.

Verification Process:

Our main approach to the Verification stage would be using the project_v0 testbench and going through the modelsim viewer. We first start with the moment the problem occurs, we then check what computations lead to the output, and finally what inputs lead to those computations. If the problem still persists we can use the sram_d0, and sram_d1 files to manually go over the calculations and check if the variables hold and calculate the same set of values.

We encountered an error Milestone 1 where the simulation gave an error at parts where RGB values would equal ff instead of 00. We first checked if the corresponding inputs were wrong but they were not. Then we checked if the computation right before the output was wrong and it was correct. We then started from the beginning calculating the values manually at each computational stage and cross checked with what we had. This led to the source of the problem, and was fixed.

Another problem in Milestone 1 was when the output was wrong after 3 common case iterations in, with this we checked if this was an input problem 3 cases before since we read 3 clock cycles with our implementation. As it turned out we found the bug in code which did not get the proper input values.

In Milestone 2 we found an error where computing 1 value of negative T was sent incorrectly into the DP ram 0 memory file, but every other value was computed correctly. At first the T block was checked to see if the other values were put in incorrectly and they were not. This led us to believe it was an individual T sign extending error, so we checked to each T assign value in the code. As we initially thought, the error was due to the first T value out of the 4 values not being sign extended.

Another problem in Milestone 2 was when we were running the testbench, and we found the values were not matching up. To troubleshoot this we computed the values of S manually, to see if the problem was before or after this step. Seeing how the values in modelsim matched with our calculations, we concluded the issue was not with the calculations of the S values. From there we checked our shifting, and finally landed on the writing of the values to the SRAM, which ultimately was our problem.

	Group Members	Work Done
Week 1	Kevin	Read project specifications and discussed preliminary designs for State Table
	Yazdan	

Week 2	Kevin	Started and finished Milestone 1 State Table	
	Yazdan		
Week 3	Kevin	Watched screenshare of Yazdan - assisted in debugging, stateflow reference, inputted ideas	Updated final report with Milestone 1 work
	Yazdan	Started to implement Milestone 1 into code - Coding was done on Yazdan's computer	
Week 4	Kevin	Watched screenshare of Yazdan - assisted in debugging, stateflow reference, inputted ideas	Started Milestone 2 State table
	Yazdan	Finished implementing Milestone 1 into code - Coding was done on Yazdan's computer	
Week 5	Kevin	Watched screenshare of Yazdan - assisted in debugging, stateflow reference, inputted ideas Updated final report with Milestone 2 work	Finished Milestone 2 State table (Monday) Finished final report
	Yazdan	Implemented Milestone 2 into code - Coding was done on Yazdan's computer	

Conclusion

With the project finished, we believe we have gained many experiences and skills. One of the main takeaways from this project is the importance of understanding the design and stateflows before coding. In milestone 1 we were too hasty in our desire to start with the code, without a proper understanding of what our code actually needed to do. This resulted in many errors, which led to many frustrations. However in milestone 2 we spent more time on the stateflow, which helped greatly in our progress.

We did not finish Milestone 2 in its entirety. We have completed the full process for the Y values (reading, computing, writing), however we did not get to the U and V values. The working code for Milestone 1 was pushed at 10:07pm on Nov 21st.