November 25, 2019

COE3DQ5 - Project Report

Group 015 - Maariz Almamun, Saaijith Vigneswaran, 400074597, 001318580
almamunm@mcmaster.ca, vignes4@mcmaster.ca

Introduction

The purpose of the project is to implement image decompression in hardware. Inorder to complete the project both group members should have an understanding of interpolation ,colorspace conversion , inverse signal transform, Dequantization and lossless Decoding. For the interpolation and colour space conversion portion of the project requirement was to take in YUV values , convert it into RGB values and output it on to the VGA controller. A constraint was put on to this portion to achieve 75% efficiency of multipliers. For the inverse transform portion of the project we are converting the pre-idct values to post idct value(YUV).

Design Structure

The design structure of this project consists of top project file (project.v modified from lab 5 4a) that acts as a top level state machine that connects all of the different design files. First the UART - SRAM interface (modified from lab 5 4a) takes the ppm file from the computer and inputs it into the SRAM. Then milestone 2(the_cosine_milstone.v) is enabled and begins to process read data taken in from the SRAM and outputs the m2_sram enable ,the m2_sram write data , and m2 disable and outputs it to the top level. The top state level was modified to so that if in the milestone 2 state use these for the SRAM . In addition, milestone two contains two instances of DPRAM that only it uses hence it was instantiated in the milestone 2 rather then the project file. Initially DPRAM 1 holds Ct from address 0-31 , C values from address 32 -63 and S values 64 -95. In addition the implementation of milestone 2 stores the S' to DPRAM 0 from memory address 0-31 (once they have been used to calculate T's they are overwritten) and T values 64-127 After milestone 2 has finished a flag is set to high and outputted to the top project at which point a flag to start milestone one is set in the top state and imputed to milestone 1 and it starts. Then milestone 1(the_fundamental_milstone.v) is enabled and begins to process read data taken in from the SRAM. and outputs the m1_sram enable ,the m1_sram write data , and m1 disable.The top level state machine was modified to so that if in the milestone 1 state to use these values for the SRAM . Similarly after this is completed a m1_disable flag is raised to turn of milestone on proceed to enabling the VGA -Interface.v (taken from lab 5 experiment 4 a). This outputs the image to the vga controller.

Implementation Details

Milestone 1 - The Fundamental Milestone

This milestone is initially broken down into three stages lead in ,common block and lead out. In addition each one of these stages is broken down into multiple states in a hierarchical FSM ; the lead in takes the first 10 states, the common block takes 12 repeating recursively until the lead out and the lead out consists of 32 states. Furthermore in each of the three stages there is either data being read from the SRAM , arithmetic manipulation with the read data , or writing back computed values to the SRAM. The overall state logic for the lead in and common block can be verified in the state table listed below. As for the lead out block it just repetition of the common block three times with the reading of u/v values left out in the first iteration , no reading of the y values in the second and no computations in the last. This milestone has approximately 88% utilization (multiplicators are used 16/18 times).

Reading Stage

In this milestone YUV values were read in from different parts of the SRAM. The first thing to consider is that every time a value is read in there is a two clock cycle latency until the input signal, read_data, obtain the value.

Consequently the output signal that points to the SRAM address has to be incremented two clock cycles ahead of time as well. Two constant signals are driven to act as offsets to insure that UV values are read correctly from the SRAM. The U values are stored in memory location 38 400 to 57 599 , as a result, a counter is added to the offset and set to SRAM address. This will insure that after the read is enabled, and the SRAM address signal will point to correct location in memory. The counter will increment up to 19200. The same counter is used for V but with an offset of 57 600 rather than 38 400.A different counter is needed for Y then uv because for every 2 y values u need 4 uv values. Once the values are read in correctly they are stored in sixteen bit registers. The Y values are shifted out sixteen bits at a time. This is also true for the u an v values . For the lead out stage MUX is implemented inorder to stop reading values in. The select lines of the MUX are driven by the pixel counter that checks if it greater than or equal to 318 , at this point the Y value stop being reading Y register. Similarly when pixel counter equals 308 for U and V for their respective register's.

Computing Stage

The next part of this milestone is arithmetic calculations with the stored YUV values. All arithmetic is done in combinational logic is done on 32 bits and signed values. The multipliers are implemented using two operators for the multiplication are also set using combinational logic that uses the states to determine what values to load in the operator signals. The operator signals determine what multiplication is being done ; U' V', even values for RGB and odd values for RGB. After the values of the even and odd terms have been calculated the arithmetic involving adding and subtracting are done in assign statements to output the RGB values that need to be sent to the new location in memory. Lastly because all the calculations were done on 32 bit signed numbers the RGB values are reduced to 8 bits.This was done using the circuit derived in class. The circuit checks if the msb is equal to 1 , if that is the case the RGB value is negative thus set to zero.In addition the circuit also checks if the value is greater than 255. If that is the case it sets the value is set 255 .

Writing stage

At this point the RGB values are ready to be written into SRAM. In order to find the correct memory location to store the RGB value a counter and an offset is used together to drive the SRAM address during these stages. The writing is declared and has no latency so values are able to be read in at the end of the clock cycle .

Design Decisions

While implementing milestone one the lead in was developed first rather than common block. This decision was taken in order to observe a pattern in the way u and values were being stored initially so the transition to the common block would happen smoothly. It was observed that the value stored in the initial register is U0u0 and then U0 U1 in the second register then there would be a significant amount of UO in the first iteration of the common block. This set up of the rest of the common blocks to work as is without any special cases for the first set of computations.

Milestone 2 - The Cosine Milestone

The trickiest aspect of this milestone next to computing all of the matrix multiplication is taking care of addressing for both SRAM/ DRAM reading/writing, both pin assignments and bit manipulation with regards to an 8x8 blocks.

For SRAM reading/writing, 3 cascaded counters were instantiated and incremented in a separate always_ff block seperate from our FSM. The implementation discussed in class is used, where 3 cascaded counters keep track of the pixel next read, current column, and row (labelled counter_pixel, counter_row, counter_column). counter_pixel increments by 1 upon receiving a signal called "increment_read" every clock cycle through a comparator. If

increment_read is true AND if counter_pixel == 63, counter_column is incremented up until 39 for Y and 19 for U/V. The same principle applies for counter_row, which increments when counter_column goes from 39 to 0.

The exact same approach occurs for the signals controlling SRAM writing. To determine if we are currently reading/writing Y's, U's, or V's, we utilize single bit signals which change from Y to U to V everytime row counter resets to 0. These signals are called iz_y and iz_u. These signals are set in the main FSM logic, however in hindsight a simpler implementation would have been changing them in the always_ff block used for the above counters.

The bits of these 3 counters (pixel, row, and column) are manipulated to give us our desired offsets for appropriately reading form the SRAM in 8x8 blocks. A long assign statement is used to declare this offset for reading/writing from the SRAM, one for reads and writes (ultimate_offset and ultimate_offset_write respectively). The same implementation in terms of bit concatenation between these signals to address SRAM is used as described in class. This includes multiplying row count by 360 for read/180 for write through the use of bit shifting the count and adding the resultant signals together. Iz_y and iz_u signals are used to determine which offset to add to ultimate_offset.

Signal concatenation of counters and their individual bits is used to set DRAM_address to increment in specific directions within the 8x8 block for the purpose of performing matrix multiplication. For matrix multiplication of 8x8 blocks, 256 clock cycles are required, and so an 8 bit signal is used as a counter for DRAM addressing.As an example when fetching S' from DRAM, we utilize the concatenation of specific bits of the 8 bit signal. (Ie. For Traversing locations 0,1,2,3 eight times, followed by 4,5,6,7 eight times, .. 28,29,30,31 x 8, bits [7:5],[1:0] are concatenated).
Our multipliers are declared in an assign statement to ensure only 2 instances are instantiated. The operands are instantiated in combinational logic, where DRAM_read_data information is directly fed into them depending on the state. When computing S, the T term signal is first right shifted by 8 before feeding into the operand as a signed value. All operands are instantiated as signed values. Each major state (3-4 in table below) utilizes a counter to count the number of clock cycles within the state. Fetch S' is 64 clock cycles and uses a 6 bit 'state' counter in states 3 and 4. States 4, 5, and 6 are 256 clock cycles and utilizes an 8 bit 'state counter'.

The nature of the computational states (3-6) is such that every single state utilizes both multipliers at 100% efficiency. The consequence of this is careful workarounds were made to ensure this can occur. This included requesting for DRAM reads 2 clock cycles before from the previous state so that values readily available at the beginning of each state. It also included having the common blocks output the final write of the final matrix multiplication product to DRAM in the first clock cycle of the following state.

Finally, the total multiplication efficiency of milestone 2 is 99.97%. It is calculated as: (256*2400 + 256*2399)/(1+3+64+256*2400*2). States 1-3 require 67 clock cycles and do not utilize the multipliers. Every clock cycle beyond this utilizes them, with the exception of the last iteration of common state 6, where it is only used as a lead out to write the final S block and computation is not used.

1	2	3	4	5(Common)	6(Common)
IDLE	Initialization	Fetch S'	Compute T	Compute S Fetch S'	Compute T Write S

<u>IDLE:</u>

IDLE is responsible for initializing all registers and signals to predetermined values (often 0, depends on function). An enable bit signal is received from top state which allows to change states to our initialization state. Increment_read also needs to be set to logic high here, as it has a 1 clock cycle latency until ultimate_offset increments (the end of the clock cycle signal is logic high, next clock cycle actual incrementation occurs).

Initialization States:

In the initialization states reads are declared from SRAM to fetch S' values. These states take care of the 2 cycle latency period of the SRAM_reads so SRAM_read_values can be loaded into DRAM immediately in the next state.

Fetch S'

This FSM state is exactly 64 clock cycles long. 5 bit counter state 'counter_fetch_state' increments for 64 clock cycles and rolls over to a value of zero when it moves onwards to the next state. This block receives SRAM_read_data every clock cycle and stores it into DRAM0 registers 0 to 63 (slot for S'). Since SRAM registers are 16 bits each and DRAM are 32 bits each, we store 2 S' values per location. To do this, in even 'states' of counter (bit 0 = 1'b0) we store read data into a buffer, and on odd 'states' we instantiate a write to DRAM with the two S' values concatenation. The last two clock cycles declares reads from DRAMs for S'/C values (to workaround the latency period) to prepare for computation with multipliers to immediately begin in the next state. These occur exactly on clock cycles 63 and 64.

This state utilizes SRAM for reading, and DRAM0's B port to write to memory locations 0 to 31. The last two clock cycles utilize DRAM0 and DRAM1's A port to read from their respective memory locations 0 to 31 (S' and C transposed values)

Compute_T

This state is exactly 256 clock cycles as is all the other states following. They all utilize an 8 bit counter called counter_common_state which increments upto 255 before rolling over to 0, signalling a state change condition.

When condition counter_common_state = 0, we enable writing for the T values. We receive DRAM_read_data from state 0 to 255. We use the bit manipulation of our counter counting reads as discussed previously to allow for reading S' and C values in patterns along the 8x8 block for matrix multiplication. Also to note the transposed C matrix is used for pulling in values as these pairs of signals are used simultaneously during matrix multiplication.

The MAC unit accumulates results throughout every 4 clock cycles to obtain the value for 1 T pre-bit shift. Thus an 8 bit counter signal implemented in the milestone is used for a comparison to check for its bits. In the case of this block 'counter_DRAM_read' is used, however the state counters were used in later states to make the logic simpler to follow along once it was implemented (Read from MAC and load new values in MAC for bits [1:0] == 2'b00). One note with the bit 00 condition is that it cannot occur in state 0 as the final computation finishes at the end of the last clock cycle. We add this as an additional condition, and instead the final write of T occurs in the first clock cycle of the next state.

In this state, up until the second last clock cycle, DRAM0/1 ports A read S' and C(transposed) values respectively. The last 2 clock cycles DRAM0/1 ports A begin reading T and C values respectively.

Common 1 - Compute S, Write S

For this state a similar implementation model is used as Compute T. It also fetches S, and so the same logical structures as the fetch S state are utilized here so they can occur concurrently.

Instead of computing one result every 4 clock cycles, instead it outputs 2 S values every 8 clock cycles. Through a few failures with implementation, we realized that the number of ports was a big constraint. It now uses DRAM0/1 ports A for reading T and C values, however we previously wanted to read in 2 T values with ports A and B. However, we must reserve DRAM0 port B to write S' values for fetch. Thus, the T value is read from the DRAM and is declared as an operand for both multipliers. This utilization required us to instead use the C matrix. Also, this constraint requires us to instead compute 2 T values simultaneously every 8 clock cycles. Nonetheless utilizing similar concepts mentioned previously (address bit manipulation, buffering), this is achieved.

Of note, this state sends a flag_complete on its second last iteration. This was initially supposed to be our last iteration of the state as we were creating lead out states, however we realized a much simpler implementation would be simply using our common states to finish off as the last two states would've done exactly the same function. Flag_complete was already declared, so we decided the easiest way to complete the milestone at this point just adding a second flag complete. While not ideal practice, given the late realization the implementation was sufficient.

This state utilizes DRAM0/1's A port to read T/C values. B port DRAM0/1 is used to write S'/S values respectively.

Common 2 - Compute T, Write S

The exact same as computing T with the exception of writes. Again, this works transversely to read S' values and so the ultimate_offset methodology is used here concurrently with T computation.

Problems

A big issue was found when uploading to quartus. There was a timing constraint error which was causing green lines through our image. This was solved by fixing minor warnings within quartus (ie. proper configuration of qsf, proper bit addressing) which lead to problem being solved.

Debugging

For both milestones, the debugging process was a matter of going through all the states and conceptualizing what we were expecting versus what was coming out. While a lengthy process this was not challenging at all. We did our hand calculations for a few blocks in both milestones until all errors in code were determined. We did not find any major errors during debugging of either milestone as we spent much more time than average conceptualizing and understanding how to implement our code with respect to hardware logic. All major conceptual flaws were realized before or during coding of the milestone, not during simulation. In hindsight advancing quicker may have been feasible between the milestones. This would have allowed more time for milestone 3 and a better chance to finish.

Wk #	Hrs	Weeks Progress	Contributions
1	10	Here we read all of milestone 1 and general project documentation together and determined how to begin approaching milestone 1 and the state table. We also looked at previous labs and how we may build off these concepts.	Maariz: Helped partner understand difficult concepts Saaijih: Gained better understanding of concepts through partner explanations Both: Brainstormed how to use previous labs

2	12	Developed lead in states, followed by common states. Recreated state table with minor changes (Due to errors in writing back to memory). Perfected state table and began implementing in code.	Maariz: created state table and programmed the code (helped set up top level stat machine) Saaijith: steps help create state table and kept quality of code and double checked code quality; set up top level state machine, quartus and simulations Equal time spent collaborating on work
3	30	Began working through M1 simulations and completed near the end of the week after minor debugging fixes.	Maariz: Lead debugging Saaijih: Set up test and simulation (helped debugging) Equal time spent collaborating on work
4	52	Began M2, worked through the logic and made plan as well as DRAM configuration with mif. Coding completed M2 Simulation done Saturday night -Simulation debugging completed sunday night after errors found -Saaijith helped with debugging and toplevel integration	Maariz: Developed logic for Milestone 2 , programmed milestone 2, and debugged Saaijith: set top level structure for milestone one and 2 work together and helped set up dp ram (helped debug) Equal time spent collaborating on work, most implementation work done by Maariz with verification by Saaijith
5	20	Milestone 2 finally working on quartus Attempt on M3 (Week 5 is considered 2 days in this report)	Maariz: Attempted to write milestone 3 code and wrote report Saaijith : Wrote write report

Conclusion

All in all, this project was an insightful journey with many trials and tribulations. The project taught us significantly about digital logic and manipulation of registers, counters, FSMs; real applicable skills that can be applied in industry indefinitely. Furthermore , during the course of this project we collaborated with a few other groups. An example of this was a discussion with Milan on how to skip striping the header for milestone 1 to work. We also had a discussion with Nathan on how to utilize using 1 T value and 2 C values to compute S given constraints.