COE3DQ5 - Project Report
Group Number: 27
Nasar Khan, Nathen Mathew
khann16@mcmaster.ca, mathen3@mcmaster.ca
Date: November 25th, 2019

Introduction

The objective of this project was to design and implement an image decompressor using hardware descriptive language. Compressed data of a 320x240 image is transmitted/received using a UART interface and stored in a SRAM where it is read by a VGA controller. The compressed data must go through a lossless decoding process after which the data is dequantized (milestone 3). Consequently, the data must then go through an inverse discrete cosine transform (milestone 2). Lastly, the data is interpolated and colourspace conversion is performed to receive the final decompressed image (milestone 1).

Design Structure

A top-level finite state machine was used to facilitate the design structure and control the interconnection of various modules. The following modules were used:
PB_Controller: This module detects whether the push button on the Altera board was pushed and provides the appropriate signal. In the top level FSM, this signal is then used to initiate the UART transmission.
SRAM_Controller: This module facilitates the use of the SRAM by driving and providing the write data, read data, write enable, and addressing signals. These signals are used by other modules such as the UART, VGA, milestone 1, and milestone 2 to manipulate and interact with the data stored in the SRAM.
UART_SRAM_interface: This module is responsible for monitoring the data that is received from the UART and stripping file headers if necessary. The module will then assemble and write data to the SRAM.
VGA_SRAM_interface: In order to generate the decompressed image on the monitor, this module generates the SRAM addressing to be used alongside the VGA controller.
M1_unit: This module handles the interpolation and colourspace conversion of compressed image data and drives the appropriate SRAM signals.
M2_unit: This module handles the inverse cosine transform of compressed image data and drives the appropriate SRAM signals.

Both modules that represent milestone 1 and 2 have start and stop signals which are used to communicate with other modules and control the top-level FSM. For example, in the top-level FSM after the UART has transmitted the image, the FSM moves to the milestone 2 state and the m2_start signal is set to HI. The FSM will only move to the milestone 1 state once milestone 2 has completed and this is signified once the module asserts the m2_done signal. Likewise the same process occurs for milestone 1.

Implementation Details

Milestone 1:

An initial idle state was used to synchronize the module with the top-level FSM. The idle state would only proceed to the lead-in case states if the start signal had been asserted to HI from the top-level and if the finish signal was LO. Multipliers initialized in combinational logic were used to handle products in calculations. The appropriate selects were assigned values one clock cycle before the product was needed in order to account for the clock cycle delay of the

select registers being updated. The RGB values that were calculated were also clipped to 8 bits unsigned.

Lead-In Case:

The lead-in case represents the situation where at the beginning of a row, during the interpolation phase, odd U'/V' values must be calculated using U/V values that are present only in that row. For example, for U'[1] the calculation 21*U[(j-5)/2] would result in a position that does not exist in the first row and therefore the first U value in the row (U[0]) must be used to replace all such cases in the calculation.

Since this scenario only occurs at the beginning of a row, it has a minimal impact on the 75% utilization constraint of the multipliers. As a result, the design decision was made to implement the scenario in 20 states. This provided the freedom to spread out calculations and ensure that the constraint of using a maximum of 3 multipliers per state was met with ease. This also allowed the design to be set up in a way that would allow an easy transition to the common case scenario. In terms of overall flow, U'[0] and U'[1] values were calculated followed by the respective V' prime values. The U/V values were stored in a shift register structure and shift out the last U/V value (first value in row) for a newer value once a prime value had been calculated. While this ongoing process occurred, RGB values were calculated over a total of 3 clock cycles per set. The design decision was made to calculate odd prime values over 3 clock cycles as well, such that one product was accumulated using an adder each clock cycle (manipulating symmetric nature of interpolation filter). As a result, this implementation allowed for use of multipliers to be spread out such that RGB values could be calculated as soon as the respective YUV values were available. The first few states were used to initialize the addressing for Y and U values to be read 3 clock cycles later (clock cycle delay of flip-flop and SRAM). In terms of addressing, the decision was made to use counters as well as offset registers to hop between YUV and RGB SRAM regions. Once the desired address was set, the respective counter was incremented using an adder to account for the next value to be read of that region.

Common Case

The common case consists of 7 cyclical states that are used to perform the bulk of interpolation and colourspace conversion in this milestone. In order to meet the multiplier utilization requirement of 75%, this meant that the multipliers needed to be used in at least 6 of the 7 states. To start, two sets of RGB values that were calculated on the previous iteration of the common case (or from lead-in case if first iteration of common case) are written into the SRAM. Since U and V values are read in pairs and their prime counterparts are calculated at half the rate, the design decision was made to read U and V pairs every other cycle of the common case. This was done by inverting a flag register at the end of every cycle. If the flag was HI, the U/V shift registers would shift the first U/V value read from the SRAM and the second value in the pairs would be stored in a buffer register. On the other hand, if the flag was LO, then data was not read from the SRAM and the buffered value from the previous cycle would be shifted to the shift registers. Odd Prime values are now calculated in one clock cycle per value, utilizing all three multipliers and manipulating the symmetric nature of the interpolation filter. The remainder of the common case is used to calculate the next two sets of RGB values to be written into the SRAM on the next cycle.

In order to detect when to transition to the lead-out states, a pixel counter was implemented. The pixel counter was incremented by 2 every time the U prime even register was set such that the pixel counter's value would always match the U prime even number. As a result in the beginning of the common case, if the pixel counter had reached a decimal value of 310, a flag was asserted to HI and this would allow the FSM to transition to the lead-out case states at the end of the current common-case cycle. A decimal value of 310 is used because the

313th prime value of each row uses U/V values that are located at the end of the row and this must be handled in the lead-out case. As a result, the 310th and 311th prime values in the row are handled as usual in the common case before transitioning to the lead-out case.

Lead-Out Case

The lead-out case can be seen as the opposite of the lead-in case. This is because at the end of the row, if the U/V values that are used in the interpolation filter are located in the next row, they must be replaced by the U/V value that is located at the end of the row instead. A procedure similar to the common case is followed in calculating odd prime and RGB values. However, instead of shifting a new U/V value into the shift registers, the value at the end of the row is shifted instead. At the end of the row (detected by pixel count) a burst of RGB writes is performed for the remainder of RGB sets for the row. Afterwards, the FSM transitions back to the lead-in states in order to begin the next row of values. However, if the SRAM address has reached the end of the RGB region, this means that the process is complete and the m1_done signal is asserted to HI while the FSM moves to the idle state.

Debugging and Simulation

The basic verification strategy was to first check if the SRAM reads and writes were occurring at the correct locations and if the correct values were being retrieved/written. If successful, then it was deduced that the source of the issue at hand was due to the calculations.

An example of a problem that occurred during this milestone was that the FSM/state table was designed where many registers were being assigned values and being used for calculations in the same state. This caused problems because the registers would only update with their newly assigned values after a clock cycle and therefore could not be used for calculations in the same state. This problem was identified in the simulation where there were many registers initialized with ‘don't care’ values. As a result, the state table and FSM needed to be altered in order to accommodate for the clock cycle delay of registers being updated.

Milestone 2:

Utilization of Dual Ports

Multiple versions of the dual port configuration were set before the final one was used. In the beginning, the idea was to store two values of T for every address in the first dual port RAM. After starting some simulation, it was realized that a minimum of 24 bits would be needed for T values and major changes were made to accommodate one T value for each address. Another thing was in the beginning configurations of the dual port, it was considered storing the C transpose values in a separate section of one of the dual port RAMs, this was not needed in the end as the C transpose values would be transversed in the same way the regular values would be transversed. Another design consideration was using MUXs to store the C values, it was decided to use the dual port instead to keep the delays between reading S prime values and T values the same as reading C values.

In the final configuration of the dual port RAMS, the first dual port ram stored S prime values in locations 0-63, and stored T values in locations 64-127, both S prime and T values were store 1 value per address. The second dual port RAM had C values in locations 0-31, and S values in 32-63, both of these values were stored two per location.

Fetch S’

Milestone 2 starts with a lead in fetch states that fetches the first block of S prime values. These lead in states start with three dummy states in the beginning where SRAM addresses begin to be set to read the pre IDCT data. Then after these three dummy states, the corresponding SRAM read data would begin to arrive and that would be stored in the first dual port RAM. This continues until 64 values are read into the first dual port RAM. After the 64th

address are set, which is 64 clock cycles from the first address being set, three more S prime values are read into the dual port in the next three clock cycles due to the 3 clock cycle delay from setting the SRAM address and receiving the data from that address.

The addressing for these values is set in another always_ff block, this always_ff block handles the SRAM addresses for this entire milestone. When a flag called fetch_en is enabled, in every clock cycle, the next value that is needed in the 8 by 8 block of S prime values will be set. Using a 6 bit sample counter which is to count each element in every S prime block it is possible to set row index, row block index, column index, and column block index, all which can be used to use the proper SRAM address. A comparator would check if the sample counter is 63, if so the column block value would go up, unless it is in the last block of the column, then the row block value would increase and the column block value would be set to 0. The first 3 bits of the sample index counter is for the row index of the block and the last 3 bits is for the column index. The row address is the row block address concatenated with the row index, and the column address is done in a similar fashion. The SRAM address for the Y region is the row address multiplied by 320, which is made by adding the row address shifted by 8 bits (multiplied by 256) added with the row address shifted by 6 bits (multiplied by 64), plus the column address. In the U and V regions, this was done by adding the row address multiplied by 160 (shifting by 5 bits and 7 bits instead) and the column address plus the appropriate offsets.

In the calculation of S states, fetch_en is enabled so that 64 values are read into the first dual port. This happens in every compute S mega state except for the last one.

Compute T

During the compute T mega state, S prime values are read from the first dual port and C values are read from the second dual port, and the result T values are written into the first dual port RAM. If it's not the first time through the compute T states, the final values of S would be read into the second dual port as well.

To compute the T matrix, elements from the rows of the S prime matrix are multiplied with columns from the C matrix. Two values of C are read for every value of S, after 8 clock cycles two T values are calculated. For example, it would be S00 * C00 and S00 * C01 in two seperate accumulators, and then S01 * C10 and S01 * C11 would be added in their respective accumulators. After 8 values are added to the two accumulators, it is shifted by 8 bits (to divide by 256). One of them is written directly to the DP RAM, and the other value is written one clock cycle later after being stored in a buffer. As the first value is being written into the dual port, the accumulators are reset and the first multiplications are added into the Accumulators. This repeats until the entire T matrix is calculated. To deal with delays, 4 dummy states were set in the beginning and 1 state was put in the end so it could write the last value in the buffer.

A multiplication counter is used to change the addressing of the dual port RAMs. There are a total of 512 multiplications in multiplying two 8 by 8 matrices. So a counter was used that went up by two every clock cycle, every 16 multiplications, the address of the S prime values would go back 7 to start again from the beginning of the row. Every 64 multiplications, the S prime address would move to the next row and the C address would start at the beginning again. After 512 multiplications, the T matrix is complete.

One setback was that the design originally didn't meet utilization constraints, as there was originally gaps between setting addresses. After making sure that every state sets an address that can be used in calculation, the utilization constraint was met.

Compute S

During the compute S mega state, C transpose values are read from the second dual port and T values are read from the first dual port, and the result S values are written into the

second dual port RAM. If it's not the last time through the compute S states, the S prime values would not be fetched into the first dual port RAM.

The computer S state multiplied the C transpose matrix and the calculated T matrix. This time there are two values per address in the dual port RAM for the C transpose values and 1 address per location for the T values. The C transpose matrix gets read in the same way the C matrix did. To compute the S matrix, rows of C transpose were multiplied by columns of T. First two values in a column would be calculated and be stored in a buffer, then the values in the same rows but the column over would be calculated. After these values are calculated, they are written in the DP RAM in one address, and then 4 addresses later to signify the values in the same column but a row down. In the end, the values would be in order in the dual port RAM.

Just like in the compute T states, there is also a multiplication counter in the compute S states. Every 16 multiplications, the addresses start at the beginning of the row of C transpose and at a new column of T values. Every 127 multiplications, the T values start at 0 again and the C transpose values go to the next row. After 512 multiplications, the S matrix is calculated and ready to be written into the SRAM from the dual port RAM.

Write S

The same always_ff block that handles the SRAM addresses for the fetch states also handles the SRAM addresses for the write states. If a flag called write_en is enabled, SRAM addresses are set for the next S values to be written into. In the Y region, each row is separated by 160 locations in the SRAM and in the U and V region, each row is separated by 80 locations. Values will be written until 32 S locations in the dual port RAM have been written into the SRAM. Then write_en will be disabled.

Debugging and Simulation

During debugging, most issues came with the edge cases, for example, switching rows, switching from Y to U to V sections, final writes. Row block and column block incrementation was imported to check during these edge cases, as the set the addresses in the correct location. Also from switching to Y to the U and V sections resulted in necessary modifications to the SRAM addressing during the fetching of S prime, where the row address would be multiplied by 160 instead of 320.

A major tool in finding errors was inputting values into a spreadsheet on excel and calculating what the values should be. This would help diagnose exactly where the error occurred and what caused it, and if there was no error in the calculations, it was often an addressing issue than needed to be fixed.

Milestone 3:

Due to time constraints, milestone 3 was not finished and no proper testbench simulation could be completed, but there is a module for traversing the matrix in a zigzag order and the incomplete module 3.

Timeline and work distribution:

Almost all work, conceptualization, and debugging was done together.

Week 1	Nathen (50%) and Nasar (50%) - Read the project and conceptualized
Week 2	Nathen (50%) and Nasar (50%) - Worked on state table for milestone 1
Week 3	Nathen (50%) and Nasar (50%) - Finished state table and began coding milestone 1

Week 4	Nathen (50%) and Nasar (50%) - Finished debugging milestone 1 and began conceptualizing milestone 2
Week 5	Nathen (50%) and Nasar (50%) - Finished debugging milestone 2 and began conceptualizing milestone 3

Conclusion

This project was to successfully decompress an image from the mic13 compression specification in hardware. Two of the three milestones were successfully completed, the colourspace conversion and upsampling (milestone 1) and the Inverse Discrete Cosine Transform (milestone 2). Due to time constraints, the decoding and dequantization aspect of the project was not completed (milestone 3). This project helped open up the world of hardware design and challenged the use of creativity and the material taught in class.

References

Hardware diagrams provided by the instructor in class and in lab were used to help conceptualize ideas and begin the design process.