COE3DQ5 - Project Report

Group 4 - Tuesday

Aaron Billones, Gurkaran Sondhi
billonea@mcmaster.ca, sondhg1@mcmaster.ca

November 30, 2020

1. Introduction

The objective of this project was to design a digital system that will implement the custom image compression method known as McMaster Image Compression revision 14 (.mic14) using hardware. Universal asynchronous receiver/transmitter (UART) will be the serial transfer method used to send the encoded (compressed) image data to the Altera DE2-115 board from a personal computer. The encoded data is stored in external static random access memory where it will be read from to perform the necessary computations to decode the data using hardware and circuitry. The design focus of this project will be performed through the stages of decoding. Once the data has been decoded, it will be read by a video graphics array (VGA) controller to display the decoded image on a monitor.

2. Design Structure

There were several modules and controllers used in the design structure of this project. The ‘experiment4’ module was used as the top level entity which came from Lab 5 experiment 4 of the course. Being the top level entity, ‘experiment4’ controlled other modules in which instances were created of ‘SRAM_controller’, ‘VGA_SRAM_interface’, and ‘UART_SRAM_interface’ that were also reused from Lab 5 experiment 4. ‘experiment4’ was responsible for initially transmitting the compressed data to the Altera DE2-115 board via UART, going through the decoding process with use of the custom circuitry, and finally outputting the decoded data through the VGA port to display the image on a monitor. The purpose of the ‘SRAM_controller’ module was to allow manipulation of the external SRAM located on the board. The use of external SRAM was required because it allowed for larger amounts of storage where embedded SRAM was not enough to hold all the data of the image. The ‘VGA_SRAM_interface’ module was used to allow data transfer from the external SRAM to the VGA port in order to display the image. The appropriate data first went from the external SRAM to the field programmable gate array (FPGA), then from the FPGA to the VGA port. Similarly, the ‘UART_SRAM_interface’ module was used to allow data transfer between UART serial transmission and the external SRAM. Using a UART receiver controller, it allowed the compressed image data to be received by the FPGA and after, transferred from the FPGA to the external SRAM. ‘Milestone1’ and ‘Milestone2’ were two custom modules that were instantiated in ‘experiment4’. Both were custom design circuitry that took part in the decoding process. ‘Milestone2’ was responsible for controlling the inverse discrete cosine transform (IDCT) where blocks of data were manipulated to reconstruct the downsampled YUV data. The purpose of the ‘Milestone1’ module was to take the downsampled YUV data from ‘Milestone2’, upsample it through interpolation, then perform colourspace conversion to obtain the corresponding RGB data. The instantiation of the embedded SRAMs (Dual Port Rams 1, 2 , 3) were initialized in ‘experiment4’ and used within Milestone2 to store the data for each of the YUV blocks. Overall, the design structure explained above allowed data transfer between interfaces and hardware components in order to implement the image compression .mic14.

3. Implementation Details

For Milestone1 and Milestone2, it was made sure that the modules were properly instantiated in the top level module ‘experiment4’. ‘experiment4’ contained the top level finite state machine (FSM) in which it would control the UART data transfer, milestone1, milestone2, and VGA data transfer. A ‘start’ and ‘finish’ bit was used to control the two milestones. Each milestone would remain in its idle state (in their corresponding FSM) until the start bit was set high in the top level FSM in which the milestones would enter the next state in their FSM. Once the FSM ran to completion in each milestone, then a ‘finish’ bit was set high and ‘start’ back to low so that the control would be brought back over to the top level FSM. Since ‘experiment4’ was the bridge between the SRAM controller and Milestone1, Milestone2, VGA, and UART controllers, there was a multiplexer (MUX) that controlled which module would drive the external SRAM’s address, enable, and write data. Depending on the state in the top level FSM, the external SRAM would be subject to the inputs of the corresponding module.

Once Milestone1 left its idle state, the appropriate stage in the decoding process would begin. In the FSM, there were 10 lead in states, 6 common case states, and 9 lead out states. Based on the formula needed to compute the upsampled U (and V) data, it was required to use 6 values of U (and V) each multiplied by a constant. The constants were inputs to a 4-1 MUX where a select counter would determine the coefficients to multiply by. The 6 values of U were stored in a shift register structure and in each common case iteration, a MUX would determine the next value to be inserted into the shift register (only 1 value of U/V needed per common case). The lead in states were used (for each row of pixels) to fill up the shift register in which the values were read from the external SRAM. Since the latency of reading from the SRAM is 2 clock cycles, the lead in states were also used for this reason. The limit of multipliers for this portion of the project was 4. To ensure that only 4 would be implemented, a MUX was used to determine the operands that would be multiplied together. Depending on the cycle, the operands would be different constants and U values from the shift register. The average utilization of the multipliers was calculated to be 90.4% (1).

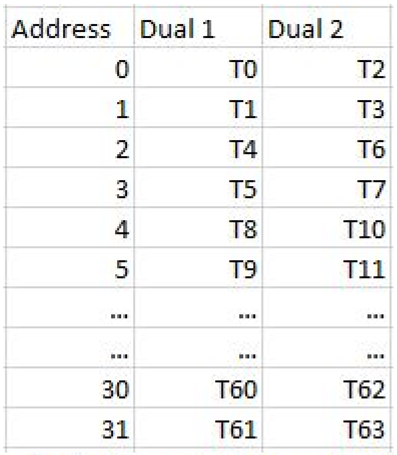
(1)
$$\frac{240(10 + (22)(160) + 12)}{240((160)(6)(4) + (19)(4))} = 90.4\%$$

The multipliers were organized in the following manner: multiplier1 was used for computing the U odd values (from the formula provided), multiplier2 was used for computing the V odd values, multiplier3 was used for computing RGB even values, and multiplier4 was used for computing RGB odd values. During each iteration of the common case there were 3 main objectives. In any given common case there would be reading YUV values and computing the Uodd/Ueven and

Vodd/Veven (upsampling) for the next set of values, computing the RGB values even and odd (colourspace conversion, from YUV to RGB) to prepare for writing for the current set, and writing the RGB values back to the external SRAM for the previous set. Since the external SRAM stored 2 values of U/V and only 1 value was needed for every common case iteration, the reading took place every other iteration. Accumulators (adders) were used to compute the values of U odd /V odd and RGB even and odd based on the formulas provided. The lead out states were used to ensure that reading from the external SRAM would stop (until the next row of pixels), and so that the final RGB values could be computed and written back to the external SRAM. The input data (downsampled YUV data) was represented as 8 bits unsigned. During computation, all arithmetic was performed in 32 bits signed. To obtain 32 bits signed from 8 bits unsigned, combinational logic was used to extend the most significant bit value or insert the appropriate number of zeros so that all the operands and products would be represented in the same way. Once all the computations were complete, RGB values (output data) needed to be represented as 8 bits unsigned. To do so, scaling and clipping were performed. In scaling, the appropriate number of bits were ignored (in this case the 16 LSB). With the remaining 16 bits, a priority encoder was used to clip the number of bits. If the MSB was a 1 (meaning negative value), then the clipped value would be 8'h00. If any of the 8 MSB were 1, then the clipped value would be 8'hFF. Otherwise, the clipped value would take the value of the 8 LSB. In any case, the output was 8 bits unsigned in which RGB data is represented as a value from 0-255. After all the values of RGB were written to the appropriate addresses in the SRAM, the finish bit was set to 1, start to 0, and the FSM in the Milestone1 module remained in its idle state. The control was then switched back over to the 'experiment4' module.

Milestone 2 also had a start bit controlling when it would leave its idle state and begin its stages of decoding. For fetching Fs, Dual port Ram 1 port A was used to write the data read from the Sram as 32 bits signed. For fetching values from the SRAM as blocks of 64, a counter of 6 bits was implemented. The counter was increased every clock cycle and concurrently, the values of the 3 LSB's were used as the column counter (ci) and the 3 MSB's were used as the row counter (ri). These counters were then used to drive the address within the SRAM using a formula 320 * (8 * rb +ri) + (8*cb+ci). The values rb and cb are the block counters, rb increases every time cb increments 40 times, and rb increments 30 times in total. There were 9 states used in this block of code in order to accomplish the reading and writing. States 1-4 are lead in states which allow the Ca, Ra, SRAM address to update. They also allow the SRAM data to be read as there is a 2 clock cycle latency. On state 5, the data is finally read and then a write is initiated to the Dual port Ram 1 port A. In state 5, the state is looped until the counter value reaches 64, and then the lead out states (State 6-8) are run in order to finish the writes to the Dual Port. The offset for this Dual port Ram is manipulated using a register with an initial value of 0, and then it is incremented after every write until 63 (first incremented initiated in State 5). The fetching segment runs for 70 clock cycles to account for updates of registers and for reading data from the SRAM.

For calculating the T matrix, Dual port Ram 1 port A was used to read the values stored from address 0-63 within its memory. Dual port Ram 2 port A and B and Dual port Ram 3 port A and B were used to write the 4 values obtained of T every 8 clock cycles. Initially, a buffer state is run in order to reset all counters and to turn off writing to unused ports. States 10 and 11 are used as Lead in States in order to read from the Dual Port Ram 1 port A and receive data from it due to the 1 clock cycle latency. Then, states 12-19 are looped to read 8 values within the Fs matrix and to calculate 4 values for the T matrix each iteration of the loop. The data being read from the Dual Port Ram 1 Port A in each clock cycle is passed to a 32 bit value called op1. The 4 C matrix values used in each cycle depend on what iteration the code is going through. The iterations will depend on a 4 bit counter called coeff_counter, its value is used in a multiplexer to obtain the correct 4 operands passed to op2, op3, op4 and op5. This counter will keep incrementing down the rows of the first four columns of the C matrix for 8 clock cycles till we get the first 4 values of the T matrix. Then, it will move on to the last four columns of the C matrix and compute another 8 values which then complete 1 row of the T matrix. So in total, its value is checked 16 times, and then it resets for calculating the next row of the T matrix. The multiplication of op1 with op2, op3, op4 and op5 are passed to the 4 accumulator registers. In order to read a row of the Fs matrix twice, the 3 LSB's and 3 MSB's of a 7 bit counter called Dual offset is used. This will then allow us to read a row twice (0-7, 0-7) which is crucial to calculate the T matrix. On state 19, the 4 accumulator values are scaled by 256 and then written to the 4 Dual ports (shown in figure 1). In the first iteration, T0, T1 are written in the same Dual port and T2 T3 are written in the same Dual port Ram.

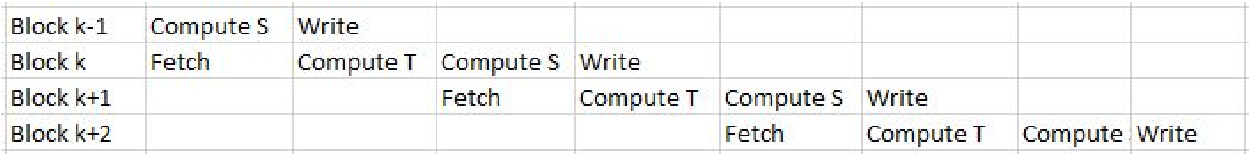


Address	Dual 1	Dual 2
0	T0	T2
1	T1	T3
2	T4	T6
3	T5	T7
4	T8	T10
5	T9	T11
...	...	...
...	...	...
30	T60	T62
31	T61	T63

Figure 1: T values stored in embedded Ram

For calculating the S matrix, Dual Port Ram 2 A, 2 B, 3 A and 3 B are being read and Dual Port Ram 1 B is being written to. States 30-31 are used to initiate reads to the 4 Dual Ports and receive data from them. States 32-39 are looped in order to read 32 values of the T matrix and select what values of the C matrix are to be used in order to calculate the final matrix S. A 7 bit counter called c_counter was used in a multiplexer in order to select values from the transposed C matrix. We used the 3 Lsb's and the 3 MSB's in order to read each row of the C matrix twice, as we have to multiply it by the left half (columns 0-3) and right half (columns 4 -7) of the T matrix in order to calculate 1 row of the S matrix. The counter will repeat a row of 8 values 2 times for 8 cycles (0-7, 0-7, 8-15, 8-15, etc), its value will be passed to a 32 bit value called op1. op2, op3, op4 and op5 will receive the values being read from the 4 Dual Port Rams each clock cycle. Then, op1 will be multiplied by each of op 2 to 5 and will then be passed to accumulators 1 to 4. After 8 clock cycles, the accumulators will then have 4 values of the S matrix. The values in accumulator 1 and 2 will first be scaled then clipped, then they will be written to the Dual Port Ram 1 B together and accumulator 3 and 4 together to another address within Ram 1 B. So on the first iteration, we will write S00 and S01 together and S02 and S03 together. The addresses that will be written to within Ram 1 B are 64 – 95. This block will run a total of 132 clock cycles.

States 51 – 55 are used for Writing S to the Sram. State 51 is used to initiate a read from the Dual port Ram 1 B starting at address 64 and receiving data due to the 1 clock cycle latency. State 52 is where a write is initiated to the Sram and the values stored in the Dual Port Ram are written. In order to write to the correct Sram address, a 6 bit counter was used. The 2 Lsb's represented the column counter (ci) and the 3 MSB's were used as the row counter (ri). These counters were then used in the formula Sram Address = 160*(8 * rb + ri) + (4*cb + ci), where cb is the column the current block is at and rb is the row the current block is at. The 4 blocks for Fetching Fs, Computing T and S and writing to the Sram were then combined in the following manner.



Block k-1	Compute S	Write						
Block k	Fetch	Compute T	Compute S	Write				
Block k+1			Fetch	Compute T	Compute S	Write		
Block k+2					Fetch	Compute T	Compute	Write

Figure 2: Combination of states to generate "Mega State"

$$\frac{2400*(256)}{64+2400*(131+136)+32}=95\%$$

*Average utilization of multipliers in Milestone 2

	Aaron Billones	Gurkaran Sondhi
Week 1	- familiarize ourselves with Lab 5 experiment 4 - read over project document and get a general understanding of what is expected	- familiarize ourselves with Lab 5 experiment 4 - read over project document and get a general understanding of what is expected
Week 2	- work on state table for Milestone 1 and organize our design decisions	- work on state table for Milestone 1 and organize our design decisions
Week 3	- begin coding Milestone 1	- begin coding Milestone 1
Week 4	- completed Milestone 1 - began planning and creating state table for Milestone 2	- completed Milestone 1 - began planning and creating state table for Milestone 2

Week 5	- begin coding Milestone 2	- begin coding Milestone 2

***Note**: Through frequent communication on MS Teams and FaceBook Messenger we would communicate our ideas/solutions/issues/decisions to each other to gain a common understanding of the material. We mostly coded on our own time with ideas and concepts being shared frequently. We consulted each other for critical design decisions and implementations in the project when necessary.

4. Conclusion

Implementation of .mic14 using hardware was a difficult task to complete. It was difficult to keep track of everything. There were a lot of specific cases and things to keep in mind while coding and creating the state tables. Generating a state table for Milestone 1 was difficult due to the constraints given which includes using only 4 multipliers. Due to this, the common case had to be 6 clock cycles long. However, computing values of U odd and V odd takes 7 clock cycles due to the accumulator being reset to 128. So, we had to include 2 extra registers called U odd and V odd which held the final accumulated value. This allowed us to initiate a reset to the accumulators and obtain U odd and Vodd in the same clock cycle without skipping a multiplication. Also, we had a register called read which would be set to either 1 for reading new U and V values, or 0 for not reading. This register was complemented every iteration of the common case. The register initially is supposed to be set to a 1 as we read values in the first iteration. However, we did not do so and thus the problem was fixed after debugging. For milestone 2, utilizing the Dual Port Rams in a way that was easy to obtain 4 values of T matrix every clock cycle for computing S was difficult. Also our milestone 2 code only until address 6524 of the testbench. After doing this project, the biggest takeaway is that it is crucial to stay organized. It can be very easy to get lost in your own work and keep track of decisions that have been made. We have gained a significant understanding of making state tables and coding and debugging within verilog.

Milestone 1 was completed on November 16, 2020 with a commit message "Milestone 1 complete".
Milestone 2 works for some portion of the testbench. It receives errors in writing the YUV data starting at address 6524.

The last commit to Milestone 1 and Milestone 2 are the final versions.

5. References

To aid in implementation of .mic14, notes and lectures from COE3DQ5, project documentation, and Labs 1-5 from COE3DQ5 were used.