Minhaj Shah
Jack Wawrychuk

Introduction

The objective of this project was to gain experience with digital systems through producing a hardware implementation of the custom McMaster Image Compression revision 14 (.mic14).

Design Structure

The code for each milestone was put into a separate module, *Milestone_1* and *Milestone_2* which were all controlled through a top state, *experiment4*. This allowed for easy switching of control of the SRAM to the VGA, UART, and milestones. A FSM was used to decide when each module would begin/end. Once the UART timed out and was finished, the state moved to milestone 2, which raised a start flag causing milestone 2 to start. Once the milestone finished, an end flag was raised in the module, which then caused the top state to move to milestone 1. The same start and end flag logic was used for milestone 1. Once milestone 1 finished, the top state returned to idle so the VGA could start. In order to control which module had control of the SRAM, a combinational case statement was used. Depending on the state of the aforementioned FSM, the SRAM address, write data, and write enable could be switched, e.g. if the top state was milestone 1, the SRAM address would be set to M1_address, but if the top state was milestone 2 the SRAM address would be set to M2_address. Some additional custom modules that were utilized are as follows: *multiplier_module* takes two 32 bit inputs and returns a 32 bit product of each operand. This is used in milestone 2 only. *Clipping_module* is also just used in milestone 2. This module takes a 32 bit input and returns the respective 8 bit value. *dual_port_RAM_A/B/C* modules are all used in milestone 2 to store, read and write data needed for computations.

Implementation

Milestone 1

Milestone 1 begins by first entering the S_IDLE state. Here, we initiate the RGB, Y, U, and V addresses to their corresponding locations in the SRAM memory map. Once a start flag is received from the top state FSM, the state transitions from idle to the first lead in state. The lead in states are used to read all the initial values for each row, and reset any counters at the beginning of each row. A counter called pixel count, which tracks how far along in each row we are, is reset to 0, and the values of both MAC units are set to a starting value of 128 according to equation 10 from the project spec.

$$U'[j] = \begin{cases} U\left[\frac{j}{2}\right] & j\ even \\ (int)\frac{1}{256}\left(21U\left[\frac{j-5}{2}\right] - 52U\left[\frac{j-3}{2}\right] + 159U\left[\frac{j-1}{2}\right] + 159U\left[\frac{j+1}{2}\right] - 52U\left[\frac{j+3}{2}\right] + 21U\left[\frac{j+5}{2}\right] + 128\right) & j\ odd \end{cases} \quad (10)$$

There is a three clock cycle latency when reading from the SRAM, meaning that if a read address is changed in state 1 of the common case, the data from that location would not be available until state state 4, when it would be stored in the SRAM_read_data register, which can then be read from. This latency is due to the inherent SRAM latency, combined with the fact that it is controlled through always_ff blocks, meaning there is an extra delay. The first two states are used to start the read for the initial U values, which are stored into registers in states four and five. The second and third states are used to start the reads for the initial V values, which are completed in states six and seven. Only two reads each are needed for the initial U/V values, as at the beginning of each row the values corresponding to U/V[j-5], U/V[j-3], and U/V[j-1] are all set to U/V[j] to avoid reading data meant

for previous rows. The read for the first set of Y data is initiated in lead in state 6, and put into registers in common case state 2.
As with the beginning of each row, at the end of each row data must be duplicated to avoid data meant for the next row. Only 159 bytes each row are read from the SRAM for each of U and V. This is because our image is 320 pixels wide, and there is one byte of U/V data needed for every two RGB pixels, meaning that the data needed is from [0:159] resulting in 160 bytes total. Thus, at the end of each row of size n the data becomes:

$$U[n-1] = U[n] = U[n+1] = U[n+2].$$

Since the data is replicated four times for the last set of calculations, we know that data 159 first appears four sets of pixels earlier, in the set of calculations for pixels 313 and 314. A pixel counter register was implemented so we know when we reach this point. Once this point was reached, all SRAM reads for U and V were stopped, and instead of shifting new or buffered data in U/V[j+5], U/V[j+5] was left the same. The rest of the shifting functionality remained, i.e. U/V[j+3] <= U/V[j+5], so that the last data byte is replicated.

After 7 Lead In states, the first Common Case begins.. The reads for the next set of U and V values are initiated in the third and fourth common case states respectively, with the U values being put into registers in state 6, and the V values put into registers in state 1 of the next common case. These reads are only performed every second common case, as there are two data bytes per location, but only one new value, U/V[j+5] is needed each time. Every time a read for these values is performed, the most significant byte is stored directly into the U/V[j+5] registers, and the lower byte is stored in a data buffer. In the next common case, instead of reading new data from memory into U/V[j+5,] the byte in the data buffer is used. In each common case the values in the U/V[j-5] to U/V[j+5] registers are shifted so that U/V[j-5] <= U/V[j-3] etc..., with the aforementioned SRAM data/buffer data being used to fill U/V[j+5]. The Y data is read in every common case in state 5, and stored into registers in the next common case, as two Y bytes are needed for each set of RGB pixels.

There are four multipliers available for use, and all four are used in each state except for the last one which uses two, in order to meet the utilization constraints. The values needed to compute U'/V' are computed in the first 3 states, so that the calculations for colour space conversion (CSC) can begin in state 4. The CSC calculations for the even pixel are completed first, using the upper byte of the Y data and U/V[j/2]. The values for the odd pixel are calculated in states five and six, with all values calculated according to equation 11 in the project spec.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = (int)\frac{1}{65536}\left(\begin{bmatrix} 76284 & 0 & 104595 \\ 76284 & -25624 & -53281 \\ 76284 & 132251 & 0 \end{bmatrix}\left(\begin{bmatrix} Y \\ U \\ V \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}\right)\right) \qquad (11)$$

Since there are five unique values to be calculated from equation 11 but only four multipliers, the CSC for each pixel needed to be done over two states, and three states in total. In the first CSC state, the CSC values for the red and green data were multiplied first, as they are the first two bytes that need to be stored in the SRAM eg {Reven,Geven}. In state five in the common case, the last value for the even blue data is multiplied, and data for the odd red and blue pixels are multiplied The values calculated from the multipliers in the previous state are added, so that the finalized data for the even red and green pixels can be sent to the SRAM in the next state. In this state blue data for both the even and odd pixels is calculated, so a buffer register is used. In the third CSC state, and last state of the common case, the green data for the odd pixel is finished being calculated, and the odd red and blue data are stored in registers, with the finished even blue data being stored in a buffer so it does not get overwritten. The even blue byte and odd red byte are stored in the SRAM in the first common case state, and the odd green and blue bytes are sent to the SRAM in the second common case state.

Since we are storing the calculated values in the SRAM in the next common case, we need some lead out states to accomplish the final writes each row. This was done by implementing two lead out states that performed writes exactly the same as in the common case. In the last lead out state, the RGB address being written to was checked. If the address was at the end of the SRAM, then the code would raise an end flag, return to the idle state and wait for the next start flag. If it was less than the end of the SRAM, then the code would return to the lead in states to get ready for the next row.

When writing, the same three clock cycle latency exists as when reading, due to always_ff blocks controlling the SRAM. If the write_enable is set to 0 and write data and address supplied, the data would not show in the SRAM until two clock cycles later. As stated above, the data for the previous set of pixels is sent to the SRAM in the last state of the current common case, and the first two states of the next common case. This means that in the first common case of each row we need to make sure that the writes in that state are not performed. This is achieved through a flag raised in the lead in state for each row and lowered in the last state of the first common case. Before the RGB data is written, checks need to be made to ensure that the data is within the bounds of colour values, [0,255], as some calculated values are small negatives or >255, using a series of muxes. To check if a value was negative, the MSB was checked and if it was 0, the value is positive, and if a 1, negative. To check if the value exceeded 255, all the bits between the MSB and the actual data being sent were or'd, and if any of those bits were a 1 the number would have to be bigger than 255, eg if our data is stored in bits [23:16], we would perform an or on bits [30:24]. If any value was negative, it was set to 0, and if any value exceeded 255 it was set to 255.

There were few major bugs and issues that needed to be addressed while in the process of coding. Once the state table was constructed there were no major logic changes from it, with most of the issues coming from typos, or from minor details that were not explicitly addressed in the state table. The largest examples of these details not explicitly addressed were adding checks for RGB data out of the bounds [0:255], checks for being at the end of the row, and the duplication of SRAM data at the end of each row ie $U[n-1] = U[n] = U[n+1] = U[n+2]$. As for the time spent in M1, the cycles per row are 7 for lead in, (320/2, number of pixel pairs to be calculated) * 6 cycles in the common case, and 2 lead out for 969 per row, * 240 rows = 232560 cycles. When multiplied by 0.02us per cycle (because 50Mhz clock), we get 4.6512ms. When inspecting modelsim, the simulation time for milestone 1 is 4.561265ms, very consistent with theory.

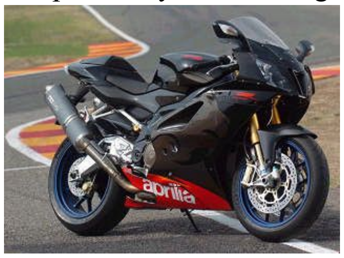
Milestone 2
The following is a breakdown of the different stages used to implement Milestone 2. Each stage has a specific purpose which contributes to the computation of this Milestone. Each stage then consists of a series of different states which will be explained in detail as well.

The first stage of Milestone 2 involves reading in values from SRAM, and storing them directly in RAM. This stage is only executed 2 times. Specifically it is just used for reading in the first block of Y and U. This is done for multiple reasons. Reading from SRAM comes along with latency which requires 3 clock cycles for data to become available after an operation is initiated. By storing these values in RAM we can reduce this to have data available after 2 clock cycles instead. This plays a big role for meeting utilization requirements since all the multipliers are needed to be working at least 90% of the execution time of the entire milestone. Another critical benefit that this stage brings, is the ability to write the incoming values from SRAM to multiple RAMS. Ports 0 of RAM_A and RAM_B are used for this. A very important detail about this stage that must be mentioned is the manner to which the values are stored in RAM. Due to the bit width of the S' values being 16 bits each, and the width of each location in RAM being 32 bits, 2 values of S' can be stored at one location. The reason this is done, is so that, during the matrix multiplication stage, a port of each RAM can be read in 2 S' values each. This allows us to have one operand of each of our 4 multipliers being assigned a value of S' during the computation stage which will later be discussed, which ensures the utilization requirements specified above are met. The way 2 S' values are stored at one location is done is by using a buffer called *S_prime_buffer*. When a value of S' arrives, it is buffered and not read again, until the next S' also arrives. Then both are concatenated, and stored together at one location in RAM. This stage consists of a repetition of two states with an additional 3 lead in states to initiate the first read from SRAM since 3 Clock Cycles are needed for data to arrive. The 2 repeating states, read in future values from SRAM and write the current ones to RAM simultaneously.

The next stage of Milestone 2 is for the first matrix multiplication computation of S'*C, which is denoted as T. In this stage a few key things happen. The first thing is that S' values are read in from RAM which are already stored and ready from the previous stage for the first block. These values are read in, and assigned as an operand for each multiplier. The other operand for each multiplier would be the respective values from the C matrix, which is the next key operation. RAM_C and both of it's ports are used solely for reading in values of C. A MIF file was created which holds all the C values in 32 locations, with 2 values at each location which is possible due to the bit width being able to fit again. This stage is a repetition of 2 states. Which means that in each of these states, 4 multipliers are being used, to calculate 4 individual partial products, making a total of 8 after the first iteration of these two states. Because of the way the S' values were stored in the previous stage, in these 2 states, we can read in a total of 8 unique S' values along with 8 unique C values, which ensures utilization constraints are met. After the first 4 partial products are calculated, they are stored in a buffer, which are not read until the next 4 partial products are determined in the second state. Then they are all summated together and written to RAM. Just like before with the S' values in the first stage, we also write duplicates of the S'*C product, denoted as T, to 2 different RAMS. The reason for this is so utilization constraints can be met in the final computation stage, which will later be discussed. This value is written every second state of each iteration of the general case in this stage. There is one more very important thing that happens in this stage which is a crucial aspect of the implementation of milestone 2. Since we had the first stage which read values from SRAM to RAM for the first block, and it was mentioned that this stage is never visited again until the next segment is ready to be computed, we need to somehow ensure the next block of S' values from SRAM are readily available at the beginning of this current stage. The solution to this is the last important action that happens in this stage. We read in future values from SRAM for the next block and write them to RAM. To prevent overwriting, we store them in the same RAMS as the current S' values, but at a different address sequence. The address sequences are alternated so we are always overwriting old values that are no longer needed. The first write happens in the previous stage from location 0 – 31 in RAM_A and RAM_B. So we read from here that simultaneously we have the future values in this stage being written to location 32 – 63 in the same RAMS. Then in the next iteration of this stage, we alternate this and read from 32 – 63, while writing to 0 – 31. This way, we don't overwrite any needed information and preserve plenty of free space in memory. Due to the 3 clock cycle latency of SRAM, we do these writes to RAM for future values of S', every other iteration of the 2 states that make up this General Case for this stage. There is also an additional 3 lead in stages for this stage. The first of these stages initiate an SRAM Read for the first next block value of S'. Since the latency makes the data available after 3 clock cycles, it will be available in the first state of the General case mentioned above. The next 2 lead-in stages initiate reads for S' values in RAM_A and RAM_B, and then C values in RAM_C. This stage executes for an entire 8x8 block, then moves on to the next stage.

The final stage for the block computations is for the last matrix multiplication of the T matrix computed previously and the C Transpose Matrix. We simply use all 6 ports from all RAMS to just read data. In the previous stage it was mentioned that we double wrote the values of T to two different RAMS. The reason for this is because the value of S'*C, is a 32 bit long value. This means that it must be stored in a single location in RAM. So we can not read in 2 values at once from one location like we do for C values and S' values. This is a critical difference because it poses a challenge to meet utilization constraints for our multipliers. Without double writing to two different RAMS, we would only be able to read in 2 T values in one clock cycle, which makes 2 of our multipliers useless in this case. This would result in half the utilization of our multipliers which would not meet the specified requirement. By having these T values written in 2 different RAMS, we can read in 4 of these values in one clock cycle, by utilizing all 2 ports on each RAM. Each of these values would serve as one operand on each of our 4 multipliers. The other operand on each would be the respective C Transpose value from RAM_C. Since the output of this Milestone must be clipped 8 bit values with 2 stored at each location in SRAM so Milestone 1 can run properly, we must utilize a buffer. We write to SRAM with the final S values, every other iteration of our general case. After one iteration of our general case is complete, we clip our value of S, and store it in a buffer. Then in the next iteration, we clip the next value to 8 bits, and concatenate it with the value from the buffer and store them both at one 16-bit location in SRAM. After a full 8x8 block of values are computed for S and written to SRAM, we go back to the previous T matrix computation stage. These stages repeat until the entire segment is computed for Y, and then we go back to the first SRAM Reading Stage for U. Then this process repeats for V. This then completes Milestone 2. The following is a breakdown of the clock cycles that Milestone 2 takes to complete all computations. To determine this value, calculating the amount of clock cycles for a single block will make things much simpler. Since 2400 block computations need to occur in total, we can simply multiply the result for one block by this value. For the first stage, for the SRAM Reads States, each RAM write happens every 2 clock cycles and 32 must happen, so this stage takes 64 clock cycles. There are also 3 Lead in States, so this whole stage takes 67 Clock Cycles. This stage happens twice, once before the Y segment, and once before the V segment, making a subtotal of 134 clock cycles. For the second stage, 64 values of the T matrix must be computed where each value takes 2 clock cycles, making a total of 128 clock cycles. Plus there are 3 Lead in states as well making a subtotal of 131 clock cycles for this stage. Similarly for the last stage for the final S matrix computation, 64 values are needed where 2 clock cycles are needed for each making 128 clock cycles as well, but with an additional 2 clock cycles for the Lead in States, making a subtotal of 130 clock cycles. Since the first stage for the SRAM Reads only occurs 2 times, this does not get multiplied by every block. Adding the totals from Stage 2 and Stage 3 along with the Reset State we get 262 Clock Cycles. Multiplying this by 2400 for all blocks gets 628800. Adding these to the subtotal of 134 from the SRAM Read in States from Stage 1, gets 628934. Finally adding the S_IDLE State and S_DELAY transition adds 2 more clock cycles getting a final grand total of 628936 Clock Cycles. This value is confirmed through Model Sim because Milestone 2 runs for approximately 12578.74 Microseconds. By dividing this by the time of 1 clock cycle which is 0.02 Microseconds, the total clock cycle number is 628936 Clock Cycles.

Milestone 2 was quite complicated and had quite a few challenges during implementation. The design of the state table was very valuable and helped development from start to end. The state table created is quite detailed and involves multiple iterations of clock cycles for each stage to ensure a thorough understanding. The verification strategy used made debugging the code very easy. An Excel Sheet was created which maps out matrices for each stage of the computation for one block. One block each for S', C, T, Ct, and S. Deriving this was quite timely, but worth it in the long run. By using a hex editor, the S' values were easily populated to make the first matrix. C and Ct transpose were given, and then by using Excel's matrix multiplication tools, the T matrix and S matrix were computed quickly. This allowed a mismatch to be located for the first block, and immediately see what the values should be, and then any discrepancies were immediately corrected. This would then just leave small bugs which occurred between transitions of blocks or segments. This Excel File is included as a separate Sheet in the Milestone 2 State Table Excel file which is in the *docs* folder in the project repo. With so much happening in the second stage of Milestone 2, with the future block S' value writes and also read in's and computations for the T values, it became very difficult to ensure correct addressing and alternating writes for data upon arrival. This method of verification made it much easier to locate the source of error. Below is a picture of *motorcycle_tb.ppm* which is produced after Milestone 2 and then Milestone 1 have finished running together. This is configured in *tb_project_v0* This confirms that the milestones are functional independently and also together.



motorcycle_tb.ppm output file

Minhaj Shah denoted as "MS" & Jack Wawrychuk denoted as "JW"

Timeline	**Summary of Tasks**
W1	- Organization of responsibilities adhering to the entirety of the project ***(MS & JW)*** - Open ended discussion of project ensuring mutual understanding of each aspect needed for completion ***(MS & JW)***
W2	- Rough proposal/methodologies for each Milestone as a game plan ***(MS & JW)*** - Initial development of State Table for Milestone 1 ***(MS & JW)***
W3	- Finalization of State Table for Milestone 1 ***(MS & JW)*** - Initial Coding for Milestone 1 ***(JW)*** - Initial development of State Table for Milestone 2 ***(MS & JW)***
W4	- Finalization of coding, debugging and verification for Milestone 1 ***(MS & JW)*** - Finalization of State Table for Milestone 2 ***(MS & JW)*** - Initial Coding for Milestone 2 ***(MS)***
W5	- Finalization of coding, debugging and verification for Milestone 2 ***(MS & JW)*** - Completion of integration and testing for Milestone 1 & Milestone 2 ***(MS & JW)*** - Discussion of approach for Milestone 3 ***(MS & JW)*** - Creation of Project Report ***(MS & JW)***

Conclusion

In conclusion, this project has covered a ranging variety of skills which ties the lecture content and labs all together to one big learning journey. The project had a creative aspect to it, which left things up to the creator to design the way one see's fit. This is a unique experience and is very valuable especially when it comes time to create our own designs for personal work or on the job.

References

- Nicola Nicolici 3DQ5 Course and Project Spec