COE3DQ5 - PROJECT REPORT
Group 36
Sarah McIntosh, mcintosc
Kevin Mano, manok2
mcintosc@mcmaster.ca
manok2@mcmaster.ca
25th November 2017

1. Introduction

The goal of this project was to produce a hardware implementation of an image decompressor[1]. This decompressor was composed of three parts for modular implementation: lossless decoding and dequantization, inverse signal transform, and interpolation and colourspace conversion. All three parts were completed, and the lossless decoding and dequantization module integrated into the inverse signal transform module. To summarize the main three modules, lossless decoding and dequantization takes an input bitstream and decodes it into the values of an 8 by 8 matrix, in zigzag order. The matrix is multiplied element-wise by its quantization matrix, and is then passed into an embedded memory. The inverse signal transform module reads the matrices from the embedded memory, and multiplies them by the Inverse Discrete Cosine Transform matrix, then the transpose of the Inverse Discrete Cosine Transform matrix is multiplied by that result. The final matrices are written to the SRAM, in blocks of 64. Interpolation and colourspace conversion reads the values from the SRAM and converts them to RGB, which is the final decompressed image.

2. Design Structure

Each milestone has its own module, each of which instantiates custom modules. For Milestone 1, there is one milestone1 module which implements a state machine, and four 32-bit multipliers (mult1 through 4). The multiplier module was created to ensure that no more than four multipliers were used at any time, as the system will not create more multipliers than the number of modules instantiated. Milestone 1 does not communicate directly with any other milestone, but makes use of the values written to the SRAM during Milestone 2.

For Milestone 2, the main milestone2 module which communicates with the top module (and also implements the main state machine for Milestone 2) instantiates several other structures. It instantiates three dual port RAMs, which each have 128 locations and 32 bits per location. The first DP-RAM is initialized with the rows of the Inverse Discrete Cosine Transform matrix. Since each element of the matrix can be represented on 16 bits, the MIF file for this RAM contains two elements per location. This design was chosen so that by reading from both ports, four matrix elements can be extracted from the memory at a time (and used in multiplications). The other two RAMs are instantiated with values of zero at every location, and are filled as Milestone 2 runs with the values required by matrix multiplications. This milestone also instantiates four multiplier modules, such as were used for Milestone 1. Again, this is to prevent the board from using more than four multipliers.

For Milestone 3, the main milestone3 module performs lossless decoding, and sends the decoded information to a variable shifter module, which it instantiates. It was determined that the variable shifter would be separate from the main module so that it would be easier to debug and track changes. The variable shifter sends addresses, write data, and a write enable outside of milestone3, so effectively information goes from milestone3 to the variable shifter to milestone2. In addition, it should be noted that Milestones 2 and 3 communicate directly with each other due to their close integration (Milestone 2 sets a flag to indicate to Milestone 3 that it can continue passing values).

Outside of the Milestones, one 256 location DP-RAM with 16 bits per location was also instantiated. It is initialized to all zeros, and is used to transfer information between Milestones 2 and 3. This memory is required, rather than transferring values directly from Milestone 3 to 2, for two reasons: Milestone 3 does not generate values in the same order that Milestone 2 uses them, and Milestone 3 does not always generate values continuously (sometimes it must wait to read more of the input bitstream).

3. Implementation Details

3.1 Milestone 1

Milestone 1 is broadly composed of a lead in, a common case, and a lead out. The common case yields RGB values for four pixels. It was chosen to be this length because it matches the frequency with which U and V values must be read from the memory. In order to meet utilization and resource usage requirements (four multipliers with 85% utilization) one multiplier each was given to U' calculations, V' calculations, calculations for even RGB values, and calculations for odd RGB values. The calculations for U' and V' both take six clock cycles, and RGB calculations for a pixel take five clock cycles. Thus, U' and V' calculations are performed continually, and RGB calculations are performed with a one clock cycle break after completion. This makes the initial estimation of multiplier utilization about 91%, excluding lead in and lead out.

Muxes were employed to send values from different registers and constants towards the multipliers, depending on the current state. The values from the multipliers were stored in multiply accumulate units, and then once the units were full they were stored in registers until written to the SRAM or used in another multiplication. By buffering the multiply accumulate units, it was possible to continue to perform calculations before the values were used.

The main debugging method that was employed for Milestone 1 was to step through the common case states, read the values in the relevant Y, U and V registers, and perform the calculations by hand, before comparing it with the output. In this way, a bug was discovered that involved writing the wrong section of the write data register to the wrong pixel value. This was easily corrected, as it was simply a typo. Another bug was an infinite loop in the lead out caused by a state always returning to another state five states before (also a typo).

The main issue that was encountered during the debugging stage of Milestone 1 was being unable to test the project correctly. The simulation would break during the very first pixel, yet when computing the values that were being fetched (by probing the registers that read from the SRAM), it was determined that everything was working as anticipated. The testbench was being used incorrectly. As a result, from this point, all testing and debugging was done using the tb_project_v2.v file instead of the tb_project.v file. Another major bug that was encountered during the testing and debugging of Milestone 1 was that there were no values written to the final few pixel values. It was determined quickly from the waveforms that the number of lead out states that had initially been implemented were too few (the values for two more pixels needed to be calculated). To correct this, a few more lead out states were added.

Based on the testbench, the total time for the simulation of Milestone 1 is 4.66ms. To determine the exact multiplier utilization percentage, the following methodology was used. The utilization of the multipliers in each state is determined by counting how many multipliers were used in that state. It follows that the total utilization is therefore the average of the utilization of the multipliers in all the states. The lead in states write 2 pixel values to the SRAM, while the lead out states write 6 pixel values to the SRAM. This leaves 312 pixel values written to the SRAM during the common case. Since 4 pixel values are written during each common case, there are 78 common case cycles. The average utilization per common case state is 91.67%. The average utilization per lead in state is 60%, and the average utilization per lead out state is 70%. Taking the grand weighted average over all the states, we get a total multiplier utilization of 90% per row, hence the multiplier utilization for the whole image is 90%. A simple sanity check is to total the number of states that occurred in this Milestone and multiply it with the time taken per clock cycle. This gives (240*(78*12 + 20 + 15) clock cycles)*(20 ns/clock cycle) = 4.66 ms, which was the simulation time.

3.2 Milestone 2

Milestone 2 can be split into four main states: fetching, megastate A, megastate B, and writing. During fetching, the values of a matrix are read row by row from the memory and stored in the top 64 locations of a RAM instantiated in the Milestone. In writing, values are read out of the top of one of the instantiated RAMs and written to the SRAM in blocks of 64. Megastate B also does writing, but in addition performs a matrix calculation, multiplying the matrix in

the top of an embedded RAM by the Inverse Discrete Cosine Transform matrix (C) and putting each computed matrix element into the top of that same RAM. Megastate A does fetching, and also multiplies the transpose of C by the values stored in the top of a RAM and writes the computed matrix elements back into the bottom of the same RAM. The lead in and lead out to the Milestone are effectively: fetching and megastate B without writing, and megastate A without fetching and writing. Note that matrices and their calculated counterparts are always stored in the same RAM. Near the end of every megastate A, the RAM being used for storage of matrix elements switches.

In megastate B, the values from lossless decoding and dequantization (S') are multiplied by matrix C to yield matrix T. In order to only use four multipliers, the matrix multiplication is performed as follows. There are eight multiply accumulate units, four of which are being filled with values at any point in time. The values of S' are multiplied by the C rows one at a time, and row by row. The megastate takes S'_{00} and multiplies it by every value in the first row of C over two clock cycles, storing the results in the multiply accumulate units. Then S'_{01} is multiplied by the second row of C, and added to the results from the previously stored values. This continues until the end of the row of S' is reached, at which point the multiply accumulate units contain the first row of T. This is repeated another seven times to store the other rows. Megastate A works similarly, yielding the final information to be written to the SRAM by multiplying C^T by T. In this case, T is multiplied column by column with the rows of C.

Megastates A and B each have a common case. The first row's worth of calculations are performed for each as a lead in overlapping with the previous megastate, after which the calculations for the following rows are performed in a common case 16 clock cycles long. There is a very high level of overlap between the two megastates, such that all four multipliers are always being used so long as the system is in a megastate. The multiplier utilization for Milestone 2 was initially estimated at close to 100%.

The addresses that were used in Milestone 2 for fetching and writing to the SRAM were generated using an address generator module. This was done simply because it made the debugging of Milestone 2 easier, and because it reduced the number of if/else statements in the common cases (reducing the number of muxes used). The address generator module used flag bits that were tied to the end of the Y address space, since the addresses that were generated for writing were different when writing to the different Y/U/V memory blocks. The rest of the address generator was simply a set of modulo counters. Two of these kept track of the rows and columns within a matrix. Another two were needed for the rows and columns of the matrix blocks. Each of these modulo counters had their own offsets that were determined by calculations of where the next memory locations for writing or reading needed to be. Finally, the registers that were used for reading and writing SRAM values were intentionally separated. This allowed fetching and writing to increment normally when resumed, instead of having to restore the previous read/write address.

There were several bugs encountered in Milestone 2. One major issue was that the values being written to the Y, U, and V locations in the SRAM after performing the Inverse Discrete Cosine Transform (IDCT) were incorrect in the very first matrix. The main debugging method that was first used to determine if the values were correct were to open the sram_d2 file, write the entire matrix out in hexadecimal, convert the values to decimal, and perform the calculations in MATLAB to cross reference them with the actual outputs that were contained in the registers. This proved to be very useful as it allowed the expected and actual outputs to be determined relatively quickly once the matrix number and location in the input file could be determined. Upon inspecting the waveform, it was determined that the issue was due to the way the S and T values were being calculated in milestone 2. The first computations as per the state table that was created would obtain the first row of T. However, the second set of multiplications use columns and were extracting the values that were stored in the RAM as rows, resulting in the wrong values being calculated. This was fixed by simply changing the order in which the T values were multiplied by the C^T coefficients.

The other major bug that was encountered during Milestone 2 was that incorrect values were being written to SRAM location 5102. This was a more subtle bug since its position was proof that the addressing up till that point had been correct, and the calculations that were being performed were also correct. The usual debugging method of finding the matrix and checking the values that were being calculated using MATLAB did not work, since the reading pattern and calculations were correct. Upon inspecting the states during which the post-IDCT values were being written, it was discovered that the same SRAM address values from before were being written to. This revealed the

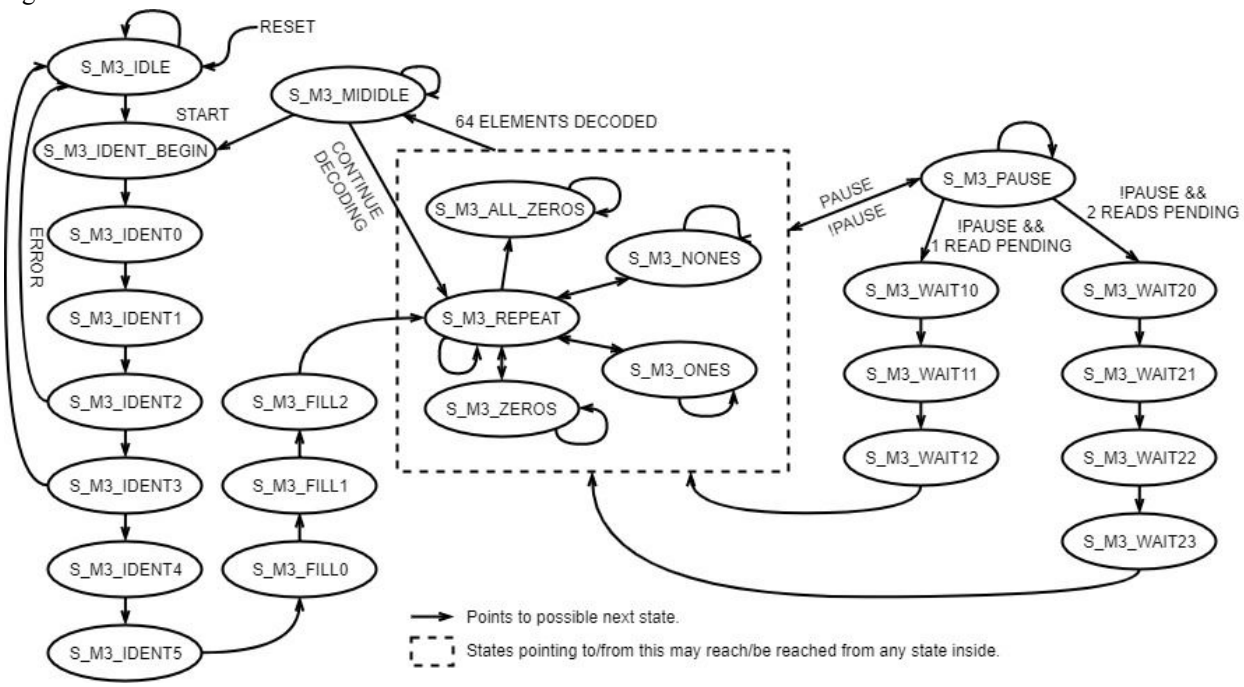
true problem: that there were not enough bits in one of the counters that was being used to determine the SRAM address to write the post-IDCT values to, and the counter had rolled over. By increasing the number of bits in that counter to 18 bits, the problem was solved.

Based on the testbench provided, the time for Milestone 2 to run to completion is 12.29 ms. To verify that this time was a result of an optimal (at least 90%) multiplier utilization, a different approach from Milestone 1 was adopted. The minimum and maximum possible times that the simulation can run for at 90% utilization is determined, and the values are then compared with the value obtained in simulation. This method is easier for Milestone 2, since Milestone 2 involves lots of overlaps between the common case states of the two megastates, hence determining that the final working version of Milestone 2 is within the timing bounds is sufficient. As discussed in class, the minimum number of clock cycles that could be used to calculate the T and S matrices are 128 clock cycles each. These 256 clock cycles have to be repeated 2400 times. Including the first fetching cycle, which can be done in as few as 64 clock cycles, and the final writing cycle, which can be done in as few as 32 clock cycles, we obtain a minimum time of (((256*2400)+ 64 + 32) clock cycles)*(20 ns/clock cycle) = 12.29 ms. Since this is the exact time that our simulation ran for, we can safely assume that the multiplier utilization is around 99.99%.

3.3 Milestone 3

The state machine for Milestone 3 is most easily explained using a diagram (Figure 1). There are four main sections used for reading the bitstream. When the Milestone starts running, it first spends several states reading the header of the file (the IDENT states). If the header does not start with the 32 bit hex code DEADBEEF, an error is triggered and the machine returns to IDLE. It then spends three states filling a buffer with 48 bits worth of data from the bitstream. This can be thought of as the lead in. After filling the buffer, there are five states for decoding. If multiple ones, negative ones, or zeros need to be written, the machine goes into separate states before returning to the main decoder ("REPEAT"). These states form the part where the Milestone is under normal operation. It can also be paused ("PAUSE") after which it may need to fetch one or two 16-bit sections of information from the SRAM. The final section of Milestone 3 operation is MIDIDLE: waiting to be told to fetch another matrix after 64 matrix values have been decoded.

Figure 1: Milestone 3 State Machine



A variable shifter module was designed to handle the dequantization process. The first version of the variable shifter used case statements for performing the zig-zag writing pattern for the matrix. This, although a working solution, could have been improved upon. The solution eventually implemented involved counters for the matrix rows and columns, both of which had incrementing limits. However, these limits only changed before passing the leading diagonal. After passing the leading diagonal, the scan pattern would end up in the 7th row and 7th column each time it moved along the diagonal. As such, upon passing the leading diagonal (which was handled using a flag), the row and column limits would be clamped at 7. One issue to this implementation was that the row and column limit increased at the same time the scan pattern reached the old row and column limit, potentially causing the traversal to get stuck on the row and column limits. Hence, another flag was used to determine, upon shifting to the right or downwards at the row and column limit, whether the element that was being scanned was the first in the diagonal line or not. If it was, it would go in the diagonal direction, and if it was not, it would increment the row and column limit, toggle off the first in line flag bit, and shift right or down as necessary. Despite the more complex logic, the final solution was implemented because it had fewer if/else statements, and hence, multiplexers.

The first major bug in Milestone 3 was that it starts filling the DP-RAM with data immediately after Milestone 2 finishes fetching all of the information in that same RAM, and later overlaps with Milestone 2's writing to the SRAM. Milestone 3 was trying to read from the SRAM while Milestone 2 was writing to it, and as such started receiving completely wrong values. This was discovered by comparing Milestone 3's read buffer to the .mic11 hex values. It was realized that Milestone 3 could not run either while Milestone 2 was fetching from the RAM or writing to the SRAM. The solution to this was to introduce another set of states for Milestone 3, which are triggered by a "pause" signal from Milestone 2. This forces Milestone 3 to stop reading and writing, and save the number of reads from the SRAM it is currently waiting to perform. When unpaused (Milestone 2 being done writing), Milestone 3 either performs the pending reads or continues with decoding.

Another major bug that was found in the integrated version of Milestones 2 and 3 was that the very last set of Milestone 2 values was written to the wrong address. This was because control of the address space for the SRAM was not handed over to Milestone 2 for the last write, causing it to write to the location Milestone 3 had last read from. This was observed simply by reading the waveform generated from the last write state, the SRAM address did not change even though it was supposed to. This was fixed by ensuring the address flag was set to allow control to switch back to Milestone 2.

One of the major decisions that was made during the testing and debugging of Milestone 3 was to immediately integrate Milestone 3 with the now debugged Milestone 2. Being aware of the risk that was being taken, this decision was made because of a few reasons. Firstly, Milestone 2 was already assessed to be working, therefore, any errors that were outside of the integration would be solely due to Milestone 3. Secondly, Milestone 3 wrote the values into a Dual Port RAM, that was not big enough to hold all the values. Therefore, there would be an unnecessary amount of extra states in Milestone 3 which would involve re-writing the values to the SRAM. It was determined that, in the interest of time, it would be more direct to have Milestone 2 read the matrices directly from Milestone 3, hence removing the process of implementing (and then removing) write states from Milestone 3. As a result of this decision, it was significantly more difficult to perform a timing analysis on Milestone 3.

Simulations of the integrated Milestones 2 and 3 were performed using the second version of the variable shifter. The timing analysis method for Milestone 3 was made more difficult as it was not possible to decouple the two Milestones once they had been integrated. As such, there was no timing simulation performed on Milestone 3 independently. Furthermore, as there were timing anomalies in the form of wait states (that could be of different lengths depending on how long Milestone 3 was in the wait states), it would not have sufficed to simply take a single measurement of the time Milestone 3 took to run a single fetch cycle, decoding and dequantization cycle. In this case, an average time for which lossless decoding and dequantization for four sample matrices, consisting of the first cycle of Milestone 3 and three random matrix blocks throughout the implementation of Milestone 3. The simulation time for the integrated Milestones 2 and 3 was 12.29ms. Out of this total time, the first cycle of Milestone 3 ran for 0.00160 ms (this was the only cycle for which Milestone 2 was not also running), 0.00132 ms of which was spent on decoding and dequantization (the rest being spent on lead in). For the other three random matrices, the lossless decoding and dequantization cycles took 0.00142 ms, 0.00132 ms, and 0.00132 ms. Therefore, the average time for a lossless decoding and dequantization cycle is (0.00132 + 0.00132 + 0.00142 + 0.00132)/4 = 0.001345

ms/matrix. This makes the total time for Milestone 3 (0.001345 ms/matrix)*(2400 matrices) + (0.0016 - 0.00132) ms = 3.228 ms.

Table 1: Work Schedule and Contributions per Group Member

Week Number	**Sarah McIntosh**	**Kevin Mano**
1 (23/10 - 29/10)	● Jointly started state table for milestone 1.	● Jointly started state table for milestone 1.
2 (30/10 - 5/11)	● Jointly worked on lead in and common case for state table for milestone 1.	● Jointly worked on lead in and common case for state table for milestone 1.
3 (6/11 - 12/11)	● Determined state table for milestone 1 lead in and jointly, the common case.	● Determined state table for milestone 1 lead out and jointly, the common case.
4 (13/11 - 19/11)	● Jointly finished debugging milestone 1. ● Determined state table for milestone 2, designed general milestone 2 structure.	● Jointly finished debugging milestone 1. ● Designed and debugged the address generator for Milestone 2.
5 (20/11-26/11)	● Finished debugging milestone 2. ● Designed and debugged lossless decoder for milestone 3. ● Jointly integrated milestones 2 and 3, and debugged integration.	● Designed and debugged the variable shifter module for milestone 3. ● Jointly integrated milestones 2 and 3. ● Timing analysis and utilization calculations for all milestones.

4. Conclusion

Over the course of the last few weeks, both group members participating in this project learned a great deal, both technically and in the practice of technical knowledge. This includes several different topics, including effective debugging techniques using waveforms, the design of complex finite state machines with multiple states and nonlinear control flow, and the basic ideas behind image compression and decompression. One of the group's largest takeaways from this project was that there are always multiple ways to solve a single problem, and that any design decisions must be made with justifications. Those justifications may include different non functional parameters, such as timing and resource utilization. This was a valuable experience due to the many challenges that were encountered along the way.

5. References

[1] Dr. N. Nicolici. (2017) COE3DQ5 Project Description 2017 [Online]. Available: http://www.ece.mcmaster.ca/~nicola/3dq5/2017/coe3dq5_project_description_2017.pdf

[2] Dr. N. Nicolici (2017) COE3DQ5 Project Report [Online]. Available: http://www.ece.mcmaster.ca/~nicola/3dq5/2017/coe3dq5_project_report_2017.pdf