COE3DQ5 – Project Report

Group 26

Ahmed Allam, Mohamed Negm

allama2@mcmaster.ca, negmm@mcmaster.ca

Nov 28, 2021

Introduction

The final project of the digital systems design course applies the concepts learned throughout the term to implement an image decompressor hardware provided an image compression specification. The project uses a McMaster Image Compression revision 15 (.mic15) software system to perform data compression on a 320 x 240 pixel image. This embedded system relies on the Universal Asynchronous receiver/transmitter (UART) serial communication method to store the compressed data on an external SRAM located on the Altera DE2-115 board. The Field Programmable Gate Array (FPGA) receives the data and carries it through multiple decoding stages in order to reproduce the original image with very limited reduction in the overall image quality. The decompressed data is then restored to the external SRAM to be picked up by the Video Graphics Array (VGA) controller and displayed on a personal monitor.

Design Structure

Milestone 2 module

Following the lossless decoding and dequantization processes done in the first stage of image decompression (i.e milestone 3), milestone 2 module receives a .sram_d2 file at its input and applies inverse signal transform on the data to output a .sram_d1 file. To further elaborate, the data picked up by this module is presented as 8x8 sized matrices which are a representation of the dequantized and downsampled YUV segments commonly referred to by S'. The Inverse Discrete Cosine Transform (IDCT) is performed in order to retrieve the downsampled image. Abiding with the design constraint of the number of multipliers allowed, two matrix multiplications are performed on each S' block in order to compute the S matrix. The resulting values are finally appended to the new .sram_d1 file which behaves as the link between milestone 1 and milestone 2.

Milestone 1 module

The completion of milestones 2 and 3 reproduces the downsampled sizes of the YUV segments to be converted back into RGB values which is the core purpose of this module. Receiving the .sram_d1 file from the milestone 2 module which now acts as the input to milestone 1, the data is first taken through an upsampling process to retrieve the original sizes of the U and V segments whereas the Y segment was not subject to any downsampling initially. The downsampled data is used as the even columns of the two segments in the upsampled version while the odd columns undergo horizontal interpolation by producing an approximated value based on the adjacent samples. Having restored the original YUV planes, the Color Space Conversion (CSC) stage provides the final transformation to the RGB basis. Performing a 3x1 scalar matrix multiplication of the YUV segments yields the RGB values which marks the final step of image decompression. The scaled results are stored into the designated RGB segment of the SRAM in the form of .sram_d0 file. The other output file is a Portable Pixel Map (.ppm) file displaying the decompressed image with close to non-existent compromise to the overall image quality.

Project module

This module is retrieved from Lab 5 experiment 4 and plays the role of the supervisor controlling the individual modules and units while ensuring successful transition between the milestones. It contains the building blocks such as the VGA SRAM interface unit, the UART SRAM interface

unit, and the SRAM controller unit all working together for the sake of transmitting and displaying information of the data being computed in the milestone 1 and milestone 2 modules. This module also contains an instantiation of the two milestones each with their respective start and stop bits to aid in the transition process. The Top level Finite State Machine (FSM) directs how the data is fetched from the SRAM and read by the VGA controller. A design decision was implemented to create our custom separate modules instead of programming them under the project module. The reason behind this decision was to enhance the overall code optimization and project organization. Having separate modules for image decompression stages allowed for independent testing and verification which was rather convenient considering the current size of each module. In addition, the decision to include both the CSC and upsampling processes under the same module was based on their relativity. Satisfying the constraint of 85% multiplier utilization became very possible when both multipliers were working interchangeably for the two stages. The aforementioned design decisions were the main factor in coming up with our current design structure.

Implementation Details

Milestone 1:

The main approach taken for the upsampling and CSC stages which highlight the overall purpose of this milestone was the state planning and computation of the common cases before switching our focus to the lead-in and lead-out cases.

A project constraint for this milestone induces the limited use of only two multipliers where they must satisfy an overall utilization larger than 85%. A design decision was executed to compute a common case in 9 clock cycles which resulted in an overall 90% utilization of the multipliers. This decision allowed the lead-in case to go over the provided limit since it only contributes approximately 0.001% to the total utilization percentage. Another reason behind the use of 9 clock cycles instead of 8 which would suggest an approximate utilization of 100% was the necessity of computing the last GB values and writing them to the SRAM. Moving on, the hardware structure of the different variables used throughout the milestone whether they are treated as registers or signals was another decision the group had to make in order to satisfy the constraint mentioned above. For instance, the YUV and RGB related variables were treated as registers while the adders and multipliers were used as asynchronous signals such that they are computed the moment their operands are assigned a value (more details are explained in table2).

To further simplify the milestone, it is broken down into three main states which are lead-in followed by common cases and finally lead-out. Since the U/V segments are downsampled and their values are available at 2 bytes/read, it was necessary to have two versions of the common cases where the first one (known as CC1) is only responsible for reading from the Y segment while the second version (known as CC2) does the reading from all three segments and stores one of the U/V bytes into a buffer to be used for the next common case. To achieve a 90% utilization of both multipliers, each multiplier was used to compute the three products needed to provide the interpolated values of U/V in a way such that one multiplier computes the U products while another computes the V products. During the interpolation process, U/V accumulators (declared as U_ACC/V_ACC) were used to store the products by adding them to the current results. Furthermore, satisfying the constraint was only possible by starting the next stage once the multipliers computed all the U/V interpolation products and during the time the accumulators were storing and updating the values of the U and V registers

(declared as U_Reg and V_Reg respectively). The approach used in this stage was the use of the two multipliers to collectively compute the even pixels followed by the odd pixels.

The lead-in state (L_IN) followed a similar convention to the common case along with the addition of two states in the beginning to initialize the reads from the SRAM and the accumulators. An extra 2 states were also added during the computation process to account for retrieving the values for the shift registers (compared to having them preassigned as in the common case) which resulted in a total of 13 states required to compute the lead-in. On the other hand, no special states were assigned separately for the lead-out cases, instead, a few conditional statements were added to CC1 to check if the end of the line is approached to stop the shifting process for the top register and store its current value. The result of this decision is a massive reduction in the overall number of states used, due to the fact that the lead-out condition is executed **three** reads before the end of the line which would suggest the addition of 27 extra states otherwise.

The connection between the milestones and the top level FSM is related by the start and finish bits (declared as M1_start, M1_finish). Actions taken based on the status of the start and stop bits all take place in the hierarchical FSM. At the end of milestone 2, M1_start is turned high signalling the start of this milestone. A row counter was used to mark the end of the milestone to check if the number of rows processed has reached 240 to which the M1_finish flag is raised.

The worst case time for this milestone was recorded to be of approximately 6.9 ps. The suggested critical path for such delay implies the use of a multiplier as they tend to hold the max amount of delay compared to other hardware structures. This path is followed by going through an adder which is used in the case of the accumulators or the RGB registers. The data delay for multiple paths was noticed to be relatively close which is self-explanatory since the critical path mentioned above is applied in many cases for both the upsampling and CSC stages.

To conclude, two main alternatives that we tried exploring and would have implemented if given the time relate to the code optimization and organization. Firstly, we would implement three main cases in the milestone corresponding to L_IN, CC1, and CC2 with each having their own sub-cases in a separate case statement. Another change to be applied in the future would be the computation of one common case and adding a condition to check if we need to read from the SRAM since this is the only difference between the two common cases currently implemented.

Milestone 2:

This milestone consisted of 3 main FSMs, the milestone top level FSM (M2_state), and two others for matrix multiplications. For M2_state, there are 6 states that serve different functions, those being, M2_IDLE, M2_LIN_Fetch, M2_LIN_CT, M2_Mega_CS, M2_Mega_CT, M2_LOUT_WS. In M2_IDLE, the milestone 2 module remains idle until the M2_start input is high, at which the top state moves to M2_LIN_Fetch. Once in M2_LIN_Fetch, the first block of Pre-IDCT data is fetched from the SRAM using a reading address generator. This address generator uses the following structure to generate the SRAM addresses. Since our C matrix is of size 8x8, the 320x240 image can be split into discrete 8x8 blocks. Then, the complete image will have a total of 800 blocks. The number of blocks in a given row and the number of rows of blocks are kept track of using a column block counter (CBC) and a row block counter(RBC). Also, to monitor the address within each block, another counter, sample counter (CSC), was used. The sample counter, when enabled, increments by 1 every clock cycle until 64 samples have been read, at which point, it rolls over back to zero. These

counters are implemented using the same logic as a modulo counter with differences in the MUX select conditions. Also, the counters relate to each other similar to a multi-digit BCD counter with the CSC being the least significant digit. Finally, using the mentioned counters and some bit manipulation, the SRAM addresses for the Pre-IDCT section are generated for reading and writing. Note that the MUX select conditions are different for Y and U/V data, in addition to reading and writing. For example, since Y data is not compressed, the CSC, CBC and RBC count up to values 64, 40, and 30, respectively. On the other hand, for U/V data, the image data is horizontally compressed, thus the CBC counts only up to 20. For simplicity, the address reading and writing generators were kept separate. Moving on, once the data is fetched from the SRAM, it's packed with 2 samples per RAM0 location starting at address 0.

To calculate and store the T = S'xC data in M2_LIN_CT and M2_Mega_CT, the same inner CT FSM was used. This FSM consists of four states, CT_PREP, CT_0, CT_1, and CT_2. The CT_PREP state is only entered once in the transition between M2_state_CT and M2_state_CS. This state prepares the first data set for multiplication in the next states by passing the first data addresses into RAM0. For this matrix multiplication, we've decided to perform the multiplications according to the equation T = S'C where the first row of S' is multiplied by all of the C matrix rows before moving to the next row. As such, the 8 data samples are read from RAM0 using both ports in 2 cc. As for the C and C-Transpose matrices, they were implemented using 3 multiplexers. Since this matrix contains many repeated values, Quartus is able to reduce the number of LUTs required for the hardware implementation. Furthermore, in each of the CT_0, CT_1, and CT_2 states, the multipliers are used to perform the matrix multiplication and results are accumulated in the register T. In CT_2, only ⅔ of the multipliers are used for a total of 8 multiplications. Finally, the T data is stored in both RAM1 and RAM2 without packing. In addition to the T computation, this state is used to read the S data from the second half of RAM0 and store 4 bytes in the SRAM every 3 clock cycles.

The second matrix multiplication S = C-TransposexT is calculated in the M2_Mega_CS state. Similar to the CT state, this state contains an inner FSM consisting of CS_PREP, CS_0, CS_1, CS_2. However, in this state the multiplication was done according to the equation S = C-TxT where the first row of C-T is multiplied by all of the T matrix rows before moving to the next row. Also, to determine the correct C-Transpose data, the C MUXs were used again, but in a separate order. Aside from these differences, the CS multiplication was nearly identical to that of Mega_CT. Finally, in addition to the S computation, this state is used to read the S' data from the SRAM and store it in RAM0 using the same address generator as the M2_LIN_Fetch state.

For 1 block of 64 data, the number of clock cycles required for reading, computation and writing are 549 clock cycles. These can split into 67cc for reading, 386cc for all computation, and 96 for writing. This structure brings the total simulation time for milestone 2 to be around 18533us.

Overall Design:

The total number of registers used for this project is 1142 registers with 506 registers for milestone 1, 265 for milestone 2, and the remainder used by the other modules. For milestone 2, the number of registers used could have been reduced significantly by reusing some in multiple states. However, due to the project time constraints, this optimization was not implemented. The design critical path delay was determined to be 18.99us from Node Milestone2:Milestone2_Unit|dual_port_RAM0:dual_port_RAM_inst0|altsyncram:altsyncram_component|altsyncram_llk2:auto_generated|ram_block1a0~porta_we_reg
To Node: Milestone2:Milestone2_Unit|dual_port_RAM0:dual_port_RAM_inst0|altsyncram:altsyncram_component|altsyncram_llk2:auto_generated|ram_block1a0~portb_datain_reg0. The

critical path is guaranteed to involve a multiplier, a RAM read/write, and an accumulator. For example, in the CS phase, we read from RAMs one and two, and compute the values in the multiplier, which passes to an accumulator register. But, since the nodes involve RAM0, this path is in either Mega_CS or Mega_CT since both states involve all of the mentioned components. Moving on, The design verification was mainly done using the Modelsim waves and the SRAM data. For both milestones, the number of maximum mismatches was set to 10. Once mismatches occurred, the waves were observed near to the mismatched value. For Milestone 2, motorcycle.sram_d1 and motorcycle.sram_d1 were used extensively to validate the calculated data. Also, to verify specific system functionalities, some control assumptions were set. For example, to verify Milestone 1 multiplication, it was necessary to assume that the data in shift registers is correct. This method allowed us to determine the implementation errors quickly and efficiently.

<table>
<tr><th>Work Breakdown</th><th>Project Progress</th><th>Ahmed Allam’s Contribution</th><th>Mohamed Negm’s Contribution</th></tr>
<tr><td>Week 0</td><td>Transition state, includes the understanding of the project requirements and the completion of all lab material</td><td colspan="2">Both individuals read the project description document and attended all lectures pertaining to lead-in instructions</td></tr>
<tr><td rowspan="2">Week 1</td><td rowspan="2">Conceptual understanding of milestone 1 and developing the state table</td><td colspan="2">Collectively worked on the common case in the state table and identified important variables to observe.</td></tr>
<tr><td>-Extended the state table to include the lead-out states of milestone 1.
-Assigned variables of their type as registers or signals.</td><td>-Extended the state table to include the lead-in states.
-Revised the complete version of the state table.</td></tr>
<tr><td>Week 2</td><td>Milestone 1 code implementation</td><td>-Implemented the lead-in states in the sequential logic block.
-Implemented the second type of common cases in the sequential logic block.
-Programmed the combinational logic block.</td><td>-Created a separate module related to milestone 1 and connected it to the top level FSM.
-Implemented the first type of common cases.
-Implemented conditions to function as the lead-out.
-Designed the transitions between the three cases.</td></tr>
<tr><td>Week 3</td><td>Milestone 1 debugging</td><td>-Verified and debugged tb_0 and wrote useful messages in the testbench to enhance the debugging performance.
-Attended office hours to clarify any functional misunderstandings from the programmable logic.</td><td>-Participated in tb_0 and completed tb_1 debugging.
-Added desired signals to the waves.do
-Set up a work schedule for the coming weeks.</td></tr>
</table>

Week 4	Milestone 2 conceptual understanding and code implementation part 1	-Created two state tables related to the C_S and C_T states. -Implemented the lead-in C_T state.	-Created a separate milestone 2 module and connected it to the top level FSM. -Created a separate module to test the writing and reading address generators. -Implemented the lead-in $F_{S'}$ state.
Week 5	Milestone 2 code implementation part 2, design verification, project report writing	-Implemented the C_S state and combined it with the $F_{S'}$ to create the mega-state. -Implemented lead-out C_S state Verified hardware design through modelsim debugging of state transitions and address generators. -Was responsible for the design structure and milestone 1 related parts of the project report	-Implemented the C_T state and combined it with the W_S state to create the mega-state. -Implemented the lead-out W_S state. -Identified and debugged multiplier operands and ensured correct storage of computed matrices. -Was responsible for milestone 2 related discussions and conclusion in the report.

Conclusion

This project gave us the opportunity to learn more about digital systems design whether that be through the understanding of combinational logic vs. sequential logic and how both of them contribute to the timing analysis or through investigating the four main FPGA elements and how they all connect to build such complex circuits such as the one in this case. Not only was the learning experience limited to the conceptual level, in addition, applying the knowledge learned through the 5 labs allowed us to further understand the concepts of logic elements, registers, etc. Interacting with the TAs or instructor was mostly done for verification purposes and sometimes to clarify a concept. For instance, we requested the TAs' assistance to verify the state table for milestone 1 and requested the instructor's assistance to verify the design plan for the dual port RAM storage. One example where the two group members were not in agreement happened early on during the project when both members were disagreeing on how the writing to the SRAM is done. The first approach was done by referring back to the notes, however, the issue was not completely resolved. The next step was done by asking a TA to explain the process and based on that we went ahead and implemented it in our code. Lastly, we asked the instructor to validate this part of the code and that is how we resolved the issue.

References

N. Nicolici, *Login - McMaster University*, 2021. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/414166/viewContent/3421823/View. [Accessed: 29-Nov-2021].