COE3DQ5 - Project Report
Group 60
Caleb Gannon and Benjamin Stephens
gannonc@mcmaster.ca and stephb5@mcmaster.ca
30 November 2020

Introduction

The project focuses on the challenges faced in system verilog hardware description language (HDL) design. It's high level focus is on the construction of a hardware image decompressor. The project consists of three milestones. Though the order of the milestones ascends from 1-3, *milestone 1* is actually the final step in the decompression process, and *milestone 3* is the first.

In *milestone 1*, our goal was to convert pixels from their YUV matrices to the corresponding RGB matrix using matrix multiplication. This process is called colour space conversion (CSC). The UV matrices are 2:1 downsampled (U_DY_D), and thus interpolation was required prior to CSC.

Milestone 2 involved the design of a hardware inverse discrete cosine transform (IDCT). Through a series of matrix multiplications, the data was converted from its frequency components, to the YUV space. Two-port DRAM modules are used to store the data while it is being processed. The pre-IDCT data is loaded into the module, where it is processed against a *C* matrix, holding the left-shifted IDCT coefficients. The coefficients are left shifted to account for fixed-point integer multiplication. The processed data is processed again by the transpose of *C*, where it is finally stored as an 8x8 block of YU_DY_D values.

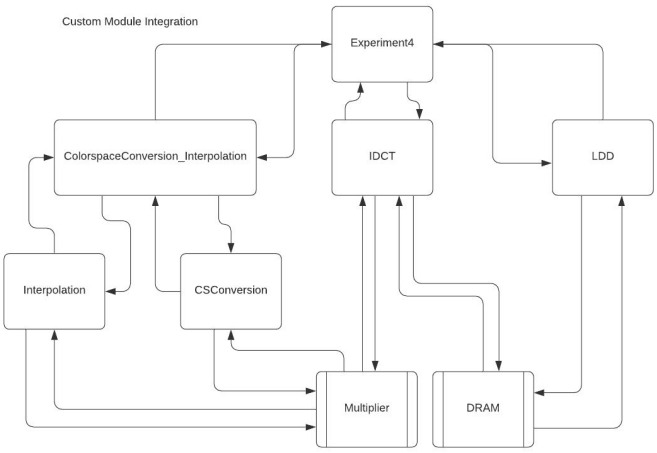
In *milestone 3*, a bitstream decoder was implemented. The bitstream is losslessly decoded and dequantized, based on two possible quantization matrices, and passed directly into *milestone 2* as the YU_DY_D frequency data.

Design Structure

The general partitioning structure that we followed for our design was to have separate modules dedicated to each milestone of our overall design. We did this because separate modules keep the thought process behind the individual modules in their own segments, hence playing a part in limiting the potential of the editing of one milestone affecting the performance of another.

Our top level module is "experiment4", which connects the UART, SRAM, VGA and our custom modules together. We use PB_controller, which lights up a green LED to indicate baseline functionality of our system. We use VGA_SRAM_interface to read the RGB data from the SRAM and render image frames. UART_SRAM_interface is used to load data to the SRAM from the environment (testbench). Finally, we use SRAM_controller to manage general communication between the modules and the SRAM. In terms of custom modules, six were created across the duration of the project. Three high level modules were created to represent each of the milestones (LDD, IDCT, ColorspaceConversion_Interpolation) The ColorspaceConversion_Interpolation module has two nested modules (CSConversion and Interpolation), to handle colorspace conversion and interpolation respectively. A general purpose *Multiplier* module is also instantiated.

In terms of different alternatives that we explored, one thing that we did notice towards the end of the project was that another feasible way of creating our custom modules was to put the logic for all 3 milestones into one single module. There are pros and cons to this - the pro is that it would allow us to use information freely between milestones without having to worry about passing data in and out of a top level structure, while the con is that errors in a single module may prove to be much harder to debug than those in a module relating to a single milestone (due to the nature of modularity).



Implementation Details

Milestone 1

Three modules were created (excluding multiplier). The first was a high level module, and the other two handled CSC and interpolation respectively.

The interpolation module was the first designed. We used a 6 element shift register to hold the values used to calculate the interpolated value. Op1 of the multiplication was the 0th value in the shift register. A multiplexor was used to select the second operand based on the interpolation coefficients. A counter was used to track the number of multiplications. On the 6th multiplication, the final result was outputted to the higher level module. A wire ran from the higher level module to grab the 3rd value in the shift register, which was used in the even-pixel RGB calculations.

Challenges faced and changes made:

The combinational logic was initially incorporated directly into the always_ff block. This was a result of inexperience in hardware design, and a misinterpretation of HDL principles. Upon later testing, the extreme difficulty in accounting for timing using this method of design was compensated for by properly constructing a paired always_comb block.

The CSC module was designed next. This module uses data from its input lines to perform the desired matrix multiplication of YUV data to produce RGB. With a greater understanding of the data flow between modules, this one is relatively simple. It consists of a counter to track which state of the matrix multiplication process is taking place, and combinational logic to output and clip the final RGB values to the high level module. The R values are calculated first, then G, then B. Registers are used to store the Y and U multiplication results that carry over between two counter states.

Challenges faced and changes made:

Similar to the interpolation module, initially all of the combinational operations were attempted within the always_ff. This attempt not only used considerably more registers, but also proved extremely difficult to time. It was observed that most of the operations performed in the always_ff could be moved to always_comb, considerably changing the design. This realization sparked the change in the interpolation module.

The high level module handled the flow of data between the four instantiated modules (Even + Odd CSC, U + V Interpolation). This module was also responsible for communication with the top level module, and thus the SRAM. In accordance with the initial design, the load case sees the shift registers of U and V interpolators filled with the initial row data, using the first byte as the first three register values to account for the image border. Then, the next major state is generateUV1, which runs only the interpolators

to compute the first odd UV values. Then the common case begins. This state runs the complex timing seen in our state table. The state uses a counter to track which stage of the common case the high level module is in. This counter is fed into the lower level modules to synchronize the design. It is not fed during the feed state, as the interpolators must be loaded with the initial values, and therefore a flag is used to differentiate these conditions. R values are stored in registers, the even G and B are on wires so they can be written immediately, and all odd GB values are in registers. Writing occurs in the final three states of the process. Y is read each iteration, and UV are read every other iteration, with a flag register to track whether they should be read in this case. Thus, the same flag that checks if they are read, determines whether the high or low byte of the UV registers is fed into the interpolators.

Challenges faced and changes made:

Timing initially was a disaster. Due to inexperience in both the syntax and design principles, data was not moving as intended throughout the design. Registers should've been wires, data should've been processed a state earlier, the always_comb was inefficiently used. Time in modelsim accounting for the errors compared against our state table revealed each problem, until the design was fixed. Additionally, a significant problem was handling the edge cases. Accounting for the flip-flop of loading values, and the x_value to stop was a complex timing challenge, using Modelsim as an aid to solve this.

Milestone 2:

This module (IDCT) was constructed in 5 parts: Load S, Compute T, Compute S, Store S, and Block tracking.

The Load S module was designed to grab a block of 8x8 elements from the SRAM, and load them into DRAM 1. It used registers to track the movement across a row of addresses, and every 8 would add 313 or 153 (Y / UV) to move to the start of the next row.

The Compute T module reads data from the S' DRAM 1, and cross multiplies it with the pre-loaded C coefficients stored in DRAM 0. We used the 4 x 1 8cc approach, where 4 columns of C are multiplied by one row of S'. A limitation of this design is the inability to write 4 values to DRAM at once. To account for this problem, data is buffered in two registers and written at the start of the next process.

The Compute S module uses a very similar structure to Compute T. The 4 x 1 8cc approach is used, however the left matrix is now C^T. to simplify the process, 4 C columns are multiplied by a row of T. Thus, the only changes necessary in the module design are the movements through the DRAMs go from linear (horizontally) in S to +8 in T (vertically). Additionally, the buffered values are now written across the next 4 states in a process, to account for the use of a single DRAM port.

The Store S module uses a similar process to Load S, with a counter to track the row of 4 writes, which then increments the address to the next row of the block. One additional difference is the buffering of values before a write, as there is only one port available in DRAM 1 to fetch values on account of Compute S.

Load S and Compute T were combined into one mega state, and Compute S Store S in another. This is done because it is ideal to perform another state (load S and store S) while computing to minimize clock cycles used, however compute T and S can not be performed at the same time on account of multiplier constraints (4x).

The block tracking module was integrated into the transfer conditional between mega states. Register counters are used to store the current position in the high level process, (i.e moving 8 each transition, as blocks are 8 values apart in Load S) This counter is fed into the Load S module, which then uses it as a starting point to grab an 8x8 block. There are separate counters to track the offset of loading from SRAM and writing to SRAM.

Challenges faced and changes made:

Edge cases made this milestone a challenging one, specifically in working out the correct internal index-wise arithmetic related to traversing each matrix during multiplication. Also, managing proper sign-extension of operands being fed to multipliers was a problem observed and corrected through modelsim. The largest bug involved the 7th multiplication in the Compute modules. Initially, the multiplication result was not added to the accumulator when the value was written to DRAM, so the 7th multiplication was ignored. What caused this to be a particularly challenging problem, is that it only affects cases of extreme high frequency. The image visually appeared quite similar to the software model. The bug was found only after performing an entire run of Compute S and Compute T by hand for an erroneous S value, and comparing the results with ModelSim.

Milestone 3

As our syntax and design skills were well improved by the time of writing milestone 3, errors regarding timing, and comb vs ff were virtually nonexistent. The initialization of the state was straightforward. Multiplexors check to ensure the first 4 bytes read *DEADBEEF*. The next bit is stored as *Q_type*, the quantization matrix. The next 15 bits are skipped by incrementing the SRAM_address by 2 (these would indicate the width). Decoding occurs in a state called RUN. It is processed using a nested set of multiplexores, which are fed the corresponding code bits offset in the buffer by *shift_counter*. The multiplexores update the shift_counter, and write the correct amount of bits to DRAM. Given the code encodes a write of zeroes, a *zeroes* counter is used. This counter is set to the number of zeroes to write - 1. A conditional occurs at the beginning of each RUN, and if *zeroes* > 0, a zero is written and no code is read. A DELAY1 state is used to compensate for the delay to access SRAM. Using a direct wire to the controller, data arrives 2 ccs after requested. A counter is used to track the index within the buffer. An always_comb is used to compute the current offset, or the addition that will be applied to the counter for the next state. If the counter + this offset exceeds 15, a request for new SRAM data is pulled for the next RUN. Always_comb constructs take the write_counter "*counter*" as input, and output the desired quantization shift, as well as the appropriate DRAM index (Zig Zag, pass it to the MAC).

Challenges and Changes:

The next challenges faced were concentrated not in the implementation of a decoding design, but in the logic of the design itself. Milestone 3 offered the largest range of possible approaches. We considered increasing the size of the buffer, to allow the flexibility to design a M3 module that would complete in 64ccs. If a large enough buffer were used, the delay in SRAM access would be properly compensated for. However, M3 is constrained by 132ccs from M2, thus a buffer of 32 bits is used. Some other issues that arose in testing M3 prior to integration, surrounded the design of the RUN, and DELAY states. Notably, the buffer index solution faced a small degree of typos. Also, the implementation of the zeroes counter did not account for the currently written zero. Otherwise, it was a clean initial implementation.

Integration

The integration finalized itself as a process where M3 is given direct write access to M2 - DRAM 1. As M3 takes 128ccs maximum, it fills DRAM1 during the Load_S_Compute_S state. Milestone 3 was a challenge to integrate into Milestone 2. The initial proposition observed that a megastate of M2 took 130ccs + 132ccs (262ccs), and thus two DELAY states were implemented in M3 would be used to grant instant access of SRAM data to the RUN state given the SRAM_address was buffered before sent to the SRAM_controller. However, if both M2 and M3 are attempting to use the SRAM simultaneously, problems occur. We attempted to solve this using complex conditionals to share the SRAM within a state, however we quickly realized that if we used a wire to pass the data directly to the SRAM_controller then

the number of states to perform M3 would be reduced to 128. Thus M3 was nested in the opposite megastate to Store S, and therefore they never compete for the SRAM.

Project-Specific Controls

WhatsTheAddy controls whether or not milestone 3 is integrated or if milestone 2 runs independently (1 indicates integration, 0 indicates independence).

Usage Constraints, Time Spent, and Requirements

M1: 4 32-bit Multipliers were used. They are instantiated in each U + V Interpolation module, as well as Even + Odd CSC.

14 loading states * 240 rows = 3360 No-Op # of Common cases = 160 * 240 = 38 400 Common states w/ Op = 22. Common states w/o Op = 2.	Total No-op = 2*38400 + 3360 = 80160 ccs Total Ops = 22 * 38400 = 844 800 ccs Total States = 924 960 ccs ***(total ccs for M1)*** Efficiency = 844 800 / 924 960=***91.3%***>85%

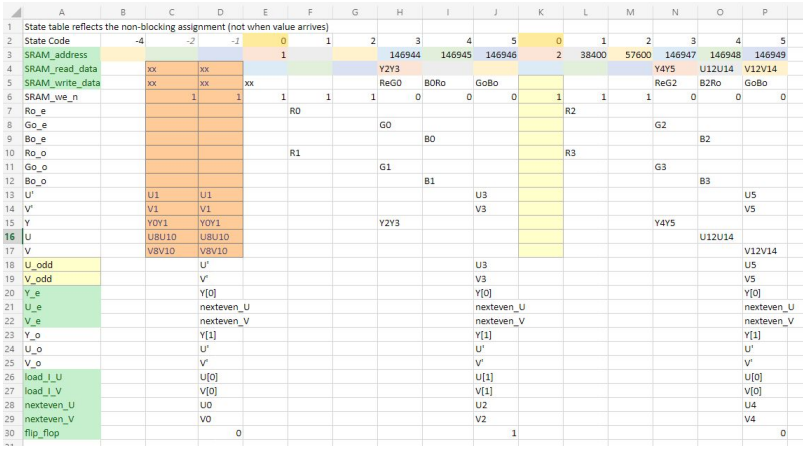
Total Time in M1 = 924960*20ns = 18.499ms

M2: MAX 3 Dual port rams: Three were used. 4kb capacity, 32 bits per location.
Initial load is 67 ccs. Much like the ability to destroy a planet, this is insignificant next to the power of the force, and does not impact the efficiency. Common case Compute S - 132ccs. Compute T - 130ccs. 128ccs with multiplication in both. Efficiency=128*2 / 262=***97.7%***>90%.
Total CCs = 262ccs * 2400blocks + 67ccs = ***628867ccs***
Total Time in M2 = 12.577ms

M3: MAX 1 additional DRAM. No additional DRAMs were used. M3 writes directly to M2 DRAM1.
Total time = M2 as they are integrated.
∴ Total time to display an image is 31.076ms, or 32.18Hz

State Tables

M1:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	State table reflects the non-blocking assignment (not when value arrives)															
2	State Code	-4	-2	-1	0	1	2	3	4	5	0	1	2	3	4	5
3	SRAM_address				1			146944	146945	146946	2	38400	57600	146947	146948	146949
4	SRAM_read_data		xx	xx				Y2Y3						Y4Y5	U12U14	V12V14
5	SRAM_write_data		xx	xx	xx			ReG0	B0Ro	GoBo				ReG2	B2Ro	GoBo
6	SRAM_we_n		1	1	1	1	1	0	0	0	1	1	1	0	0	0
7	Ro_e					R0						R2				
8	Go_e							G0						G2		
9	Bo_e								B0						B2	
10	Ro_o					R1						R3				
11	Go_o							G1						G3		
12	Bo_o								B1						B3	
13	U'		U1	U1						U3						U5
14	V'		V1	V1						V3						V5
15	Y		Y0Y1	Y0Y1				Y2Y3						Y4Y5		
16	U		U8U10	U8U10											U12U14	
17	V		V8V10	V8V10												V12V14
18	U_odd			U'						U3						U5
19	V_odd			V'						V3						V5
20	Y_e			Y[0]						Y[0]						Y[0]
21	U_e			nexteven_U						nexteven_U						nexteven_U
22	V_e			nexteven_V						nexteven_V						nexteven_V
23	Y_o			Y[1]						Y[1]						Y[1]
24	U_o			U'						U'						U'
25	V_o			V'						V'						V'
26	load_I_U			U[0]						U[1]						U[0]
27	load_I_V			V[0]						V[1]						V[0]
28	nexteven_U			U0						U2						U4
29	nexteven_V			V0						V2						V4
30	flip_flop			0						1						0

Note: M2 and M3 have no formal Excel State tables, as their structure and design was developed through discussion, note taking, and informal conceptualization of ideas (Whiteboarding).

Project Timeline		
Group Member	Ben	Caleb
Week 1 (October 26-November 1)	Beginning to write milestone 1 + overview of project specs	Beginning to write milestone 1 + overview of project specs
Week 2 (November 2-November 8)	Milestone 1 state table conceptualized and created, verilog written	Milestone 1 state table conceptualized and created, verilog written
Week 3 (November 9-November 15)	Milestone 1 debugged	Milestone 1 debugged Additional milestone 1 testing
Week 4 (November 16-November 22)	Milestone 2 conceptualized + verilog written	Milestone 2 conceptualized + verilog written
Week 5 (November 23-November 30)	Milestone 2 debugging completed Milestone 3 conceptualized, verilog written and debugged Project report completed	Milestone 2 debugging completed Milestone 3 conceptualized, verilog written and debugged Project report completed
Overall Contribution	48%	52% (additional M1 testing)

Conclusion

Over the course of this project, we were able to take major steps in our learning process as a team. We experienced many challenges along the way, but in the end were able to put together a finished product.

One major challenge faced was working with our time, where a major push was required in the final two weeks.

Another final challenge that we faced was to do with debugging. Due to the detail-oriented nature of verilog hardware design and the strictness of the language, as a result of the nature of computers and how explicit one's instructions to the computer must be. Our mistakes from past milestones helped us improve on this throughout the course of the project.

Our biggest learning outcome from this project was that the project is highly resemblant of the real world. The so-called "valley of despair" in project flow is highly emphasized in the midst of debugging a module. Lessons learned involve finishing, committing to, and seeing a project through without any guarantee of success. This was amplified by the knowledge that each milestone needed to be completed before moving to the next. As true engineers, half solutions no longer get us where we need to go. To reiterate, this was an important lesson in resilience and whole-hearted work.

References

In terms of resources used, the project spec and class notes/live lectures were heavily used as reference for the completion of this project. Additionally, no references/outside resources were used apart from interactions with the instructor/teaching assistants for this project.