COE3DQ5 – Project Report

Group 46 - Justin Covach, Daniel Wei
covachj@mcmaster.ca, weid10@mcmaster.ca
Nov 28th, 2022

Introduction

The purpose of this project is to create an image decompressor using hardware implementation. There are 3 main components to this project. Milestone 3 implements lossless decoding and dequantization to the bitstream from a .mic16 compressed file. Milestone 2 then uses the data generated and performs an inverse-discrete cosine transform with the challenge of using only 2 multipliers with at least 92.5% utilization, producing YUV pixel values. Milestone 1 then upscales the U and V values and performs colour space conversion to make RGB pixel values. Only 3 multipliers are used in this stage at a 77.5% utilization rate. These three milestones make up the 3 steps in image decompression. We will discuss the design structure and implementation we used in the following sections.

Design Structure

The image decompressor is partitioned into three modules. The first module implements Milestone 3, which reads the .mic16 bitstream stored in the SRAM and generate pre-IDCT YUV blocks. The second module implements Milestone 2, which reads the generated blocks and writes to the SRAM the transformed YUV values. Because Milestone 3 and Milestone 2 writes and read the same 8x8 blocks in order, the two modules are pipelined so that instead of writing each block to SRAM, the blocks are written to a dual-port memory and can accessed by the next module immediately. This will reduce the total number of clock cycles needed to process the bitstream as the SRAM will be accessed more efficiently, stopping concurrent read/writes. To facilitate debugging and test benching both modules have a flag to either read/write to the SRAM or the dual-port memory.

The third module implements Milestone 1, which reads from the SRAM the YUV values and performs colour space conversion on each set or interpolates U and V when necessary. The resulting RGB set is written to the SRAM by pixel number where it is then read by the VGA controller. The third module activates only after the first two modules completed their processing tasks and writes the YUV values to SRAM. It would be extremely difficult to pipeline and directly use the outputted YUV values to the RGB. To produce an RGB set requires a corresponding YUV set, however Milestone 2 processes data in 8x8 blocks of Y values, U values, then V values, while Milestone 1 reads data sequentially.

Implementation Details

Module 1 (Milestone 3)

FSM

When a new block is requested, SRAM data is read and stored into a 32-bit bitstream, enough to contain 2 SRAM values. In the state S_LD_DQ_READ_HEADER, the first 2-4 bits of the bitstream are compared to get the instructions. The number of data (bit_num) and the size of the data (bit_size) is obtained from the instruction and is stored, and the 0-12 bits to write is stored in write_bit_acc. The bitstream is then shifted to get rid of the used bits and updates the total number of bits in the bitstream.

The next state, S_LD_DQ_WRITE, is when the SRAM or the dual-port is written to. Depending on the number of data to write, this state can repeat from 0 to 16 times or to the end of block. The state subtracts one from bit_num until it reaches zero, where it would either go to the read header or the finished block state. A ZZ_counter keeps track of the current number of elements in the block, and is mapped to ZZ_address_offset to get the correct address. The write data is parsed from the write_bit_acc, adjusted for signed values, and dequantized through bit shifting.

For every state, there is functionality to add bits to the bitstream. It follows the subsequent rules:

1. If there is no space in the bitstream, store the next value into the buffer if possible
2. If there is space in the bitstream and a buffer value, insert the buffer value to the bitstream and write the next value into the buffer if possible
3. If there is space and no buffer value, insert directly the next value to the bitstream and leave buffer empty

Figure 1. Rules to insert values in the bitstream

To check if the SRAM_read_data is a valid read data, a shift register is used to keep track of the number of clock cycles when the last read address was sent.

Decoding Latency

Overall, the worst case for the number of clock cycles per block used in this module will not exceed about 2*64 (common states) + 3 (block states) + lead in/lead out states, meeting the timing requirements for M2. This is because it takes one cc to get the instructions, and it takes one cc to write. Worst case, only one value is read for each instruction. For each block, an additional 3 states are used as lead in and lead out.

The strict timing requirements of Milestone 3 was achieved using an exceptional amount of combinational circuitry. For example, zig-zag traversal was mapped to an offset using 6 6-LUTs. The number of bits to shift in dequantization was also mapped, likely to 3 4-LUTs. To parse write_bit_acc, it uses several MUX for the offset of the value, and the size of the value. It is then adjusted for signed using several more MUX, which is then shifted using the dequantize value. This happens all during one clock cycle, making the circuit extremely fast, but the worst-case for timing becomes more of an issue.

Debugging and Verification

For this single module, verification was perhaps more difficult than implementing the states. To verify, the values must be written to SRAM. However, when writing to the SRAM the read values would also change to the written value. Originally the module used the SRAM read data to "store" the next bitstream data; now the data needs to be stored in a buffer. This resulted in a host of issues.

When an erroneous value is written to a memory location, it could be because of an error several clock cycles ago. Figure out which bits were wrong, then figure out which value was wrongly written to the bitstream. With this technique several bugs were caught:

1. Skipping over data during changes in blocks: Sometimes, data would be skipped over before inserted to the bitstream in between blocks. This was resolved by allowing the bitstream_buf to be written to in more states.
2. Overwriting bitstream_buf: when a new value is read from the SRAM before the value in the bitstream_buf was used, it would be overwritten and lost. This was resolved by setting a condition to check if bitstream_buf is empty before writing (see above rule 1)

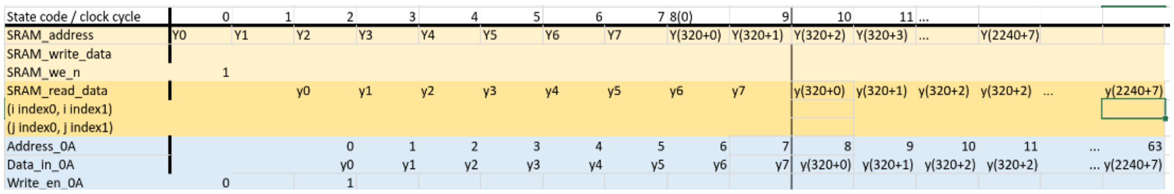
Register name	**Bits**	**Description**
q	1	Flag sets which quantization matrix to use
Bitstream	32	Bitstream
Bitstream_counter	6	Counts number of usable bits in bitstream
Bit_num	4	Determined by the instruction, sets the number of data to write
Bit_size	4	Determined by the instruction, sets the size of the data to write
Write_bit_acc	12	Collection of all data from one instruction
Added_value	1	Flag when data is added to the bitstream, increases the next SRAM address
Write_to_buf	4	Shift register that indicates when the SRAM_read_data is good to read
zero	1	Flag to fill rest of the block with zero
Bitstream_buf	16	Buffer to store next SRAM data
ZZ_counter	6	Counter for the number of elements in the block
Block_counter	12	Block counter, resets for each YUV type
Total_block	12	Counter total blocks
Block_row	17	For tb SRAM addressing for the block row
Block_col	9	For tb SRAM addressing for the block column
Read_address_counter	18	For tb SRAM addressing

The critical path is between ZZ_counter and SRAM_address. This is likely due to the sheer amount of MUX's the data passes through in a single clock cycle. The path likely goes from: ZZ_counter-> ZZ_address_offset->di->shift->write_data-> SRAM_write_data->SRAM_address

Module 2 (Milestone 2)

The FSM starts by loading in all values of our first 8x8 block by iterating through the SRAM and storing the values in the upper half of dual port 0

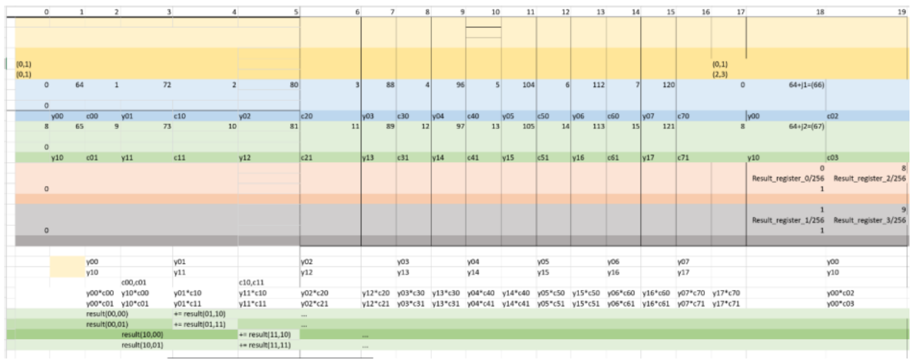
<u>Leadin State Table</u>



State code / clock cycle	0	1	2	3	4	5	6	7	8(0)	9	10	11	...		
SRAM_address	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y(320+0)	Y(320+1)	Y(320+2)	Y(320+3)	...	Y(2240+7)	
SRAM_write_data															
SRAM_we_n	1														
SRAM_read_data			y0	y1	y2	y3	y4	y5	y6	y7	y(320+0)	y(320+1)	y(320+2)	y(320+2) ...	y(2240+7)
(i index0, i index1)															
(j index0, j index1)															
Address_0A			0	1	2	3	4	5	6	7	8	9	10	11	... 63
Data_in_0A			y0	y1	y2	y3	y4	y5	y6	y7	y(320+0)	y(320+1)	y(320+2)	y(320+2)	... y(2240+7)
Write_en_0A	0		1												

Once we have received the first block of 8x8 values, we start the first repeating phase of our FSM. The S'C matrix multiplication. In this state the FSM loops through for 16 iterations multiplying 2 rows of each matrix at once.These are calculated, one value at a time from each row and stored in respective Sum_registers0...4. At the end of one row and column set we have 4 values for our T matrix. These values are stored in the upper half of dual port ram1 which is initially empty. The matrix values are stored with respect to their column and row values. The C values are stored into the ram through the .mif file that we created for it.

<u>State table for matrix multiplication for S'C</u>



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	(0,1)																	(0,1)		
	(0,1)																	(2,3)		
	0	64	1	72	2	80	3	88	4	96	5	104	6	112	7	120		0	64+j1=(66)	
	0																			
		y00	c00	y01	c10	y02	c20	y03	c30	y04	c40	y05	c50	y06	c60	y07	c70		y00	c02
	8	65	9	73	10	81	11	89	12	97	13	105	14	113	15	121		8	64+j2=(67)	
	0																			
		y10	c01	y11	c11	y12	c21	y13	c31	y14	c41	y15	c51	y16	c61	y17	c71		y10	c03
																			0	8
																			Result_register_0/256	Result_register_2/256
	0																		1	
																			1	9
																			Result_register_1/256	Result_register_3/256
	0																		1	
			y00		y01		y02		y03		y04		y05		y06		y07			y00
			y10		y11		y12		y13		y14		y15		y16		y17			y10
				c00,c01		c10,c11														
			y00*c00	y10*c00	y01*c10	y11*c10	y02*c20	y12*c20	y03*c30	y13*c30	y04*c40	y14*c40	y05*c50	y15*c50	y06*c60	y16*c60	y07*c70	y17*c70		y00*c02
			y00*c01	y10*c01	y01*c11	y11*c11	y02*c21	y12*c21	y03*c31	y13*c31	y04*c41	y14*c41	y05*c51	y15*c51	y06*c61	y16*c61	y07*c71	y17*c71		y00*c03
			result(00,00)		+= result(01,10)		...													
			result(00,01)		+= result(01,11)		...													
				result(10,00)		+= result(11,10)		...												
				result(10,01)		+= result(11,11)		...												

During this repeating section we are storing the calculated S values back into the SRAM. This process only occurs once our First_pass_complete variable is toggled to high.

This section follows the exact same flow as the last repeating section except it accesses different values and right shifts them by 16 to store them into the lower half of dual port ram 1. The other difference that occurs during this section is the writing of the next 8x8 SRAM values into the upper half of dual port ram 0. In between sections of reading C transpose values we read the next block of 8x8 values for our next calculation and store them into the 0...63 addresses of our first dual port ram.

For this Section we have 4 stages, two stages which share the amount of clock cycles spent. Our first stage, the lead in uses 8 clock cycles per row of matrix, this is repeated 8 times. There is also 2 cycles of delay from the SRAM_read_data being received so this section uses 66 clock cycles. Our second and third stage use the same amount of clock cycles. For once cycle of data we use 16 cycles for the multiplication and 1 for the storage. This repeats 16 times. Therefore, for each of these sections we use 272 clock cycles. For one block these are each used once. We repeat this for 40x30 blocks of Y, 20x30 blocks of U and V. For our last section we are leading out. This section repeats for 32 cycles and has a delay of 1 for the retrieval of data.

Debugging

For the debugging of this milestone, we had many issues that arose. In all stages of this debugging, we would look at the expected values from simulation, and compare the values that we had in Modelsim. Many time the issue was with indexing. The other issue we resolved was the improper indexing of the C transpose and T matrix. This issue was resolved by checking through the system flow when results were invalid.

Registers

Register name	**Bits**		**Description**
I_index	6		Holds the current matrix row
J_index	6		Holds the current matrix column
I_index_buf	6		Holds the last matrix row for storage
J_index_buf	6		Holds the last matrix column for storage
Datablock_I_index	6		Holds the current matrix block row index
Datablock_J_index	6		Holds the current matrix block column index
Matrix_counter	10		Holds the current k value for matrix multiplication
Sprime_write_counter	6		Holds the current index for storing S’
S_write_counter	6		Holds the current index for storing S
First_pass_complete	1		Toggles on when first pass has completed
Y_complete	1		Toggles on when Y section has been completed
SRAM_write_counter_I	4		Holds the current row for SRAM storage
SRAM_write_counter_J	4		Holds the current column for SRAM storage
C_buf_0	16		Holds the C values for Matrix multiplication
C_buf_1	16		Holds the C value for Matrix multiplication
C_buf_sel	1		Toggles which data values are selected for Matrix multiplication
Sprime_T_buf0	16		Holds the S’ and T values for Matrix multiplication
Sprime_T_buf1	16		Holds the S’ and T values for Matrix multiplication
Sum_register0…4	32		Holds the sums of values for matrix multiplication

Module 3 (Milestone 1)

FSM

The FSM waits for the US_SCS_enable flag to be enabled. When it is, the FSM goes into the lead in states, reading the first 3 U and V values as they will be needed to upsample the odd pixel. Because there is a constraint on the number of multipliers, their operands need to be able to change between many different registers. MUX’s are used to select between US and CSC modes, each with their own corresponding values. When in US mode, U_V_RB_G_select switches between using U_buf or V_buf for its calculations. When in CSC mode, it switches between the two sets of integers to calculate R and B or G. The outputs of the multiplication are added, truncated, and rounded before being written to the SRAM.

In the common case, only 17 multiplications and 6 read/writes on average to calculate two pixels. Adding one clock cycle to make timing easier, 7 clock cycles are used for every two pixels. This means that the entire 320*240 picture can be processed in around 268800 + 9 (lead in) cycles. The multiplier usage is at 6/7 or around 85.7%.

Debugging

During simulation, one bug found was that the pixels around the 160th were resulting writing the wrong values. The issue was that the last pixels of the row should not be able to access values of the next row, i.e. pixel 159 cannot use 160, 161, 162, and the code did not take this into account. The reason for the error was found only after printing out the write count when the errors occurred and seeing that there was a pattern.

Registers

Register name	**Bits**	**Description**
Write_address_counter	17	Counter for the RGB write address
Read_address_counter	16	Counter for the SRAM read for YUV values
Col_counter	8	Detects when its near the last elements in a row
CSC_select	1	Selects if the multipliers should do up sampling calculations or colour space conversion calculation
U_V_RB_G_select	1	If doing up sampling calculations, selects U or V, if doing CSC calculations, selects RG or G resulting multiplications
U_buf [5]	8	Buffer to store U values for US and CSC calculations
V_buf [4]	8	Buffer to store V values for US and CSC calculations
U21 [5]	16	Stores the U[x]*21 values used in US calculations and shifts values in and out
V21 [5]	16	Stores the V[x]*21 values used in US calculations and shifts values in and out
U52 [3]	16	Stores the U[x]*52 values used in US calculations and shifts values in and out
V52 [3]	16	Stores the V[x]*52 values used in US calculations and shifts values in and out
U159	16	Stores the U[x]*159 values used in US calculations and shifts values in and out
V159	16	Stores the V[x]*159 values used in US calculations and shifts values in and out
Y_prime	8	Values used in CSC calculation
U_prime	8	Values used in CSC calculation
V_prime	8	Values used in CSC calculation
US_CSC_complete		Flag to mark Milestone completion

As reported in the timing analyser, the critical path is between col_counter and U_prime. The likely path is from col_counter->Multi0_op1-> Multi0_out->U_acc->U->U_prime.

Contributions

Week	Project Progress	Contributions
Week 1 (Oct. 24)	Began reading project requirements	
Week 2 (Oct. 31)	Conceptualized and began discussions on the state table for milestone 1	
Week 3 (Nov. 7)	Completed state table for milestone 1 and implemented code for milestone 1	Justin: Completed repeating cycle for M1 and began concept for M2
Week 4 (Nov. 14)	Verified milestone 1 with tb_v0 and tb_v1, began state table for milestone 2 and 3	Daniel: Completed milestone 1, and verified using tb_v0 and tb_v1. Began work on the state table for M3 Justin: Completed state table for M2 and began implementation for M2
Week 5 (Nov. 21)	Completed state table for milestone 2 and 3, implemented code for milestone 2 and 3, verified milestone 2 and 3	Daniel: Completed state table and code for milestone 3, and verified using tb_v0 Justin: Worked on M2, incomplete, Matrix multiplication working, indexing and storage unverified

Conclusion

This project was a very good learning experience for us. The process of designing and implementing was very challenging and offered many opportunities to learn and experiment. The testing and verification was also a good learning experience as it made us more familiar with the technology and the process of debugging and iterative implementation. Overall we were able to learn a lot about the design process and implementation of computer systems and hardware through this project. We can learn to manage ourselves better in future experiences to come out with more success.

Milestone 1 was completed on Nov 23rd with the message: "Milestone 1 is completed and verified"
Milestone 2 is partially complete in the most recent commit
Milestone 3 is complete in the most recent commit

References

Thong, J., Kinsman, A., and Nicolici, N. “COE3DQ5 Project Description 2022 Hardware Implementation of an Image Decompressor”