Student Name: Hazem Taha Student ID: 400550393

Project 1 Report:

Hardware Implementation of an Image Compressor

Milestone 1: Color Conversion and Down Sampling

For milestone 1, I started by drafting a state table (shared in file Appendix_A.pdf) in order to see the transitions and decide the registers I needed. I basically have 3 leads in states and 6 common states. I assign 3 multipliers to calculate the values of Y, U and V. A pair of Y's, U's and V's is calculated every 6 clocks (iteration of the common case). When the first 2 values of Y are collected, I start to write them in the SRAM.

For calculating the U` and V` , we can do a small simplification that will require the use only use 3 multiplications and a set of additions and shifting. Instead of 7 multiplications and additions.

$$\begin{aligned} S'[i] &= 22S[2i-5] - 52S[i-3] + 159S[2i-1] + 256S[2i] + 159S[i+1] + -52S[i+3] \\ &\quad + 22S[i+5] + 256 \\ &= 22(S[2i-5] + S[i+5]) - 52(S[i-3] + S[i+3]) + 159(S[2i-1] + S[i+1]) \\ &\quad + (S[2i] \ll 8) + 256 \end{aligned}$$

The computed U's and V's are buffered in a way that allows reusing them multiple times to calculate the sequantial U` and V` values in an efficient manner. Finally, a new pair of U` and V` every 12 cycles (every other iteration of the common case).

Registers used in Milestone 1:

Module	Register Name	Bits	Description
CSCD_UNIT	RGB_pointer	$\log_2 \frac{No.\ Cols \times No.Rows \times 3}{2}$	Pointer used to index in the RGB Segment
CSCD_UNIT	Y_pointer	$\log_2 \frac{No.\ Cols \times No.Rows \times 3}{2} - 1$	Pointer used to index in the Y Segment (also used in U and V)
CSCD_UNIT	SRAM_read_data_Buffer	8	Buffers RGB read values to next clock when needed
CSCD_UNIT	U/V	8	Store new calculated U, V values
CSCD_UNIT	Y_Buffer	2x8	Buffers 2 Y values to write them at once to SRAM
CSCD_UNIT	U_E_Buffer / V_E_Buffer	3x8	Buffers the last 3 even U and V values to use later in computations
CSCD_UNIT	U_O_Buffer / V_O_Buffer	6x8	Buffers the last 6 odd U and V values to use later in computations
CSCD_UNIT	Ud_Buffer/ Vd_Buffer	2x8	Buffers that store pairs of calculated U' and V'
CSCD_UNIT	Y_Write_EN	1	Flag to allow the writing Y value to SRAM.
CSCD_UNIT	UV_Buffer_EN	1	Flag to allow the new pushing of values into U_E_Buffer and V_E_Buffer

CSCD_UNIT	Ud_Vd_Pair_Ready	1	Flag indicating pairs of U` and V` are ready, it alternates every full iteration of the common state.
CSCD_UNIT	begin_Ud_Vd_Write	1	Flag to allow the writing U' and V' values to SRAM.
CSCD_UNIT	Accumulator	4x26	4 MAC Units Accumulators. 3 used in calculating the Y, U, V and 1 for calculating U' and V'
CSCD_UNIT	pixel_X_pos	$\log_2 \text{No. Cols}$	Keeps track of the current X pixel location
CSCD_UNIT	pixel_Y_pos	$\log_2 \text{No. Rows}$	Keeps track of the current Y pixel location

The size of the accumulators was designed to accumulate the worst-case scenario of accumulated multiplications + 1 sign bit.

Milestone 2: Discrete Cosine Transform

The below equation (1) describes the calculation of the DCT Matrix $\boldsymbol{S'}$ for a given 8x8 block of values scanned from the Y, U and V channels produced by milestone 1.

$$\boldsymbol{S'} = \boldsymbol{CSC^T} \tag{1}$$

Where $\boldsymbol{C}$ is a constant precalculated matrix of values. To start, we need to fetch the $\boldsymbol{S}$ blocks. I have done this by combinatory calculating the fetching addresses from the SRAM for a given clock cycle according to the equation:

$$\begin{aligned}\boldsymbol{SRAM\ Fetch\ Address} \\ &= \{\boldsymbol{block\_column\_start\_counter}, \boldsymbol{block\_relative\_column\_counter}\} \\ &+ \boldsymbol{block\_row\_start\_address} + \boldsymbol{Channel\ Base\ Adderss}\end{aligned}$$

Notice that {} are for signal/wire concatenation. I have tried to find a way to use a counter for the row start just as in the columns instead of keeping track and accumulating the 20 bits $\boldsymbol{block\_row\_start\_address}$ but this will need to have an additional multiplier which to calculate the block row starting address form the counter so I had to stick to this way.

Then, equation (1) can be divided in to two steps:

1) $$\boldsymbol{T} = \boldsymbol{SC^T} = \begin{bmatrix} \boldsymbol{S_{0,0}} & \cdots & \boldsymbol{S_{0,7}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{S_{7,0}} & \cdots & \boldsymbol{S_{7,7}} \end{bmatrix} \times \begin{bmatrix} \boldsymbol{C_{0,0}} & \cdots & \boldsymbol{C_{0,7}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{C_{7,0}} & \cdots & \boldsymbol{C_{7,7}} \end{bmatrix}^{\boldsymbol{T}} = \begin{bmatrix} \boldsymbol{S_{0,0}} & \cdots & \boldsymbol{S_{0,7}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{S_{7,0}} & \cdots & \boldsymbol{S_{7,7}} \end{bmatrix} \times \begin{bmatrix} \boldsymbol{C_{0,0}} & \cdots & \boldsymbol{C_{7,0}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{C_{0,7}} & \cdots & \boldsymbol{C_{7,7}} \end{bmatrix}$$

Where every 8 partial products can be calculated in 4 multiplications (mega partial products) as follows:

$$\begin{aligned} \boldsymbol{T_{i,j}} &= \sum_{k=0}^{3} \boldsymbol{C_{j,k}}\left(\boldsymbol{S_{i,k}} + (-\mathbf{1})^{j}\boldsymbol{S_{i,7-k}}\right) \\ &= \boldsymbol{C_{j,0}}\left(\boldsymbol{S_{i,0}} + (-\mathbf{1})^{j}\boldsymbol{S_{i,7}}\right) + \boldsymbol{C_{j,1}}\left(\boldsymbol{S_{i,1}} + (-\mathbf{1})^{j}\boldsymbol{S_{i,6}}\right) + \boldsymbol{C_{j,2}}\left(\boldsymbol{S_{i,2}} + (-\mathbf{1})^{j}\boldsymbol{S_{i,5}}\right) \\ &\quad + \boldsymbol{C_{j,3}}\left(\boldsymbol{S_{i,3}} + (-\mathbf{1})^{j}\boldsymbol{S_{i,4}}\right) \end{aligned} \tag{2}$$

2) $$\boldsymbol{S'} = \boldsymbol{CT} = \begin{bmatrix} \boldsymbol{C_{0,0}} & \cdots & \boldsymbol{C_{0,7}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{C_{7,0}} & \cdots & \boldsymbol{C_{7,7}} \end{bmatrix} \times \begin{bmatrix} \boldsymbol{T_{0,0}} & \cdots & \boldsymbol{T_{0,7}} \\ \vdots & \ddots & \vdots \\ \boldsymbol{T_{7,0}} & \cdots & \boldsymbol{T_{7,7}} \end{bmatrix}$$

Where every 8 partial products can be calculated in 4 multiplications (mega partial products) as follows:

$$\begin{aligned} \boldsymbol{S'_{i,j}} &= \sum_{k=0}^{3} \boldsymbol{C_{i,k}}\left(\boldsymbol{T_{k,j}} + (-\mathbf{1})^{i}\boldsymbol{T_{7-k,j}}\right) \\ &= \boldsymbol{C_{i,0}}\left(\boldsymbol{T_{0,j}} + (-\mathbf{1})^{i}\boldsymbol{T_{7,j}}\right) + \boldsymbol{C_{i,1}}\left(\boldsymbol{T_{1,j}} + (-\mathbf{1})^{i}\boldsymbol{T_{6,j}}\right) + \boldsymbol{C_{i,2}}\left(\boldsymbol{T_{2,j}} + (-\mathbf{1})^{i}\boldsymbol{T_{5,j}}\right) \\ &\quad + \boldsymbol{C_{i,3}}\left(\boldsymbol{T_{3,j}} + (-\mathbf{1})^{i}\boldsymbol{T_{4,j}}\right) \end{aligned} \tag{3}$$

Before we start there are some notations I want to highlight:

1) **S_Embedded_RAM**: Embedded RAM used to store the fetched $\boldsymbol{S}$ block from SRAM.
2) **T_Embedded_RAM**: Embedded RAM used to store the calculated $\boldsymbol{T}$ block values.
3) **Sd_Embedded_RAM**: Embedded RAM used to store the calculated $\boldsymbol{S'}$ block values.

Starting with equation (2), I considered different ways to implement it efficiently in hardware using only 4 multipliers and with throughput of equivalent to 1 element of $\boldsymbol{S'}$ per clock. The most promising solutions were as follows:

1) Best method using a single port S_Embedded_RAM: Share 4 fetched elements from $\boldsymbol{S}$ to compute 2 (mega partial products) for two different $\boldsymbol{T}$ elements per clock. Will need two accumulators and 4 elements ready per clock.
2) Best method using a double port S_Embedded_RAM: Compute 4 (mega partial products) to get one full $\boldsymbol{T}$ element per clock. Will need 1 accumulator and 8 elements ready per clock.

I was leaning more towards method 1 to reduce the use of embedded memories but I got convinced by your explanation that maybe in this small capacity it will not make a difference so I implemented method 2.

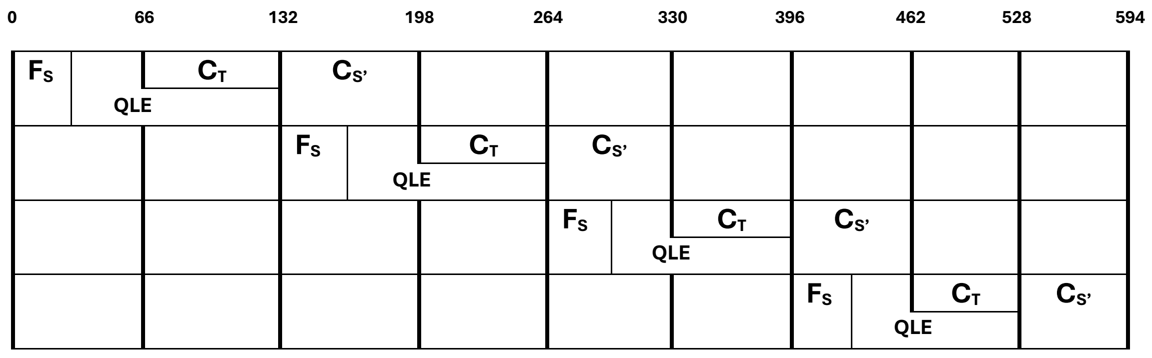
Another point to discuss is how many T_Embedded_RAMs to use. I decided to plan ahead and analyze my needs before I pass this stage to avoid revisiting it again. Which brings me to equation (3). To calculate 1 $\boldsymbol{S'}$ element per clock we need to have eight $\boldsymbol{T}$ elements (each of size 16), with a total size of 128 bits. The maximum we can get from our dual-port T_Embedded_RAM is 64 bits per clock. Which means I must make use of at least 2 dual-port T_Embedded_RAMs per clock to satisfy the needed bandwidth. The two T_Embedded_RAMs will be then used to store the $\boldsymbol{T}$'s in such order:

Student Name: Hazem Taha Student ID: 400550393

$$\overbrace{\begin{bmatrix} T_{0,0} & T_{1,0} \\ T_{4,0} & T_{5,0} \\ \vdots & \vdots \\ T_{0,7} & T_{1,7} \\ T_{4,7} & T_{5,7} \end{bmatrix}}^{T_{RAM\,1}} \overbrace{\begin{bmatrix} T_{2,0} & T_{3,0} \\ T_{6,0} & T_{7,0} \\ \vdots & \vdots \\ T_{2,7} & T_{3,7} \\ T_{6,7} & T_{7,7} \end{bmatrix}}^{T_{RAM2}}$$

It's logical to use only one Sd_Embedded_RAM to store the S', however, planning ahead, as I will explain in the next section for milestone 3, we need to have 2 distinct Sd_Embedded_RAMs to perform a ping-pong approach where one RAM is being written to by milestone 2 while the other is being scanned for milestone 3. Therefore, I switch between writing the S' block between the 2 embedded RAMs after every Compute S' phase **($C_{S'}$).**

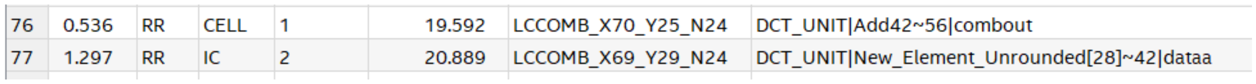
The below diagram shows how different stages (including QLE for milestone 3) are applied on different blocks in parallel in a pipelined approach. Notice that each window (enclosed between two bold columns) has a size of 66 clocks which is 2 clock cycles more than expected for the calculations of **C_T** and **$C_{S'}$**. This is due to the fact that 1 clock is necessary before the multiplication starts to fetch the element, and the other clock was artificially added to break down the longest path as will be explained shortly.



Timing Violation:

After implementing and verifying my solution for milestone 2, I found that the maximum frequency dropped heavily to 48MHZ. This was due to 2 reasons:

1) False paths both inside milestone 2 and between milestones 1 and 2. I solved this by declaring the false paths in the "timing_50_MHz.sdc" file.
2) A true setup time violation from the T_Embedded_RAMs to Sd_Embedded_RAMs, passing through a number of adders, multipliers, and shifters, which causes the frequency to drop to 48MHZ. I searched for a good spot to split the computation to maintain the best frequency I had earlier and found this part and decided to split between these two steps:



76	0.536	RR	CELL	1	19.592	LCCOMB_X70_Y25_N24	DCT_UNIT|Add42~56|combout
77	1.297	RR	IC	2	20.889	LCCOMB_X69_Y29_N24	DCT_UNIT|New_Element_Unrounded[28]~42|dataa

Cutting here and buffering part of the summation of the multiplied elements to the next clock solved the violation successfully. However, this added an extra clock cycle in the **C_T** and **$C_{S'}$** window making them 66 clocks each.

Registers used in Milestone 2:

Module	**Register Name**	**Bits**	**Description**
DCT (FS)	Buffer_SRAM	16	Buffers the 2 fetched S values from SRAM to write 4 at once in S_Embedded_RAM.
DCT (FS)	SRAM_Fetch_Segment	2	Flag that indicates which channel are we currently fetching to control how to move through rows and when to stop fetching.
DCT (FS)	block_relative_row_counter	3	Counter that is used to traverse over the rows relative to a block in SRAM.
DCT (FS)	block_relative_col_counter	2	Counter that is used to traverse over the columns relative to a block in SRAM.
DCT (FS)	block_col_start_counter	$\log_2 \frac{No.\ Cols}{8}$	Counter that is used to traverse over the blocks' starting columns in SRAM.
DCT (FS)	block_row_start_address	20	Keeps track and accumulate the current blocks' starting row address.
DCT (CT)	T_i_counter, T_j_counter	3	Counters to iterate over the rows and columns of the T block to write the elements.
DCT (CT)	T_Buffer	16	Buffers the calculated T value to store 2 T's at once in the T_Embedded_RAM.
DCT (CT)	Write_in_T_RAM1	1	Flag that alternates the writes between T_Embedded_RAM1 and T_Embedded_RAM2.
DCT (CSd)	Sd_i_counter, Sd_j_counter	3	Counters to iterate over the rows and columns of the S' block to write the elements.
DCT	MULTs_Buffer	64	Buffer that stores summed outputs from the multipliers (halfway through a long combinational circuit) to cut the longest path delay in half. Used in **C_T** and **$C_{S'}$**.

I tried to reuse the T_i_counter, T_j_counter from **C_T** in **$C_{S'}$** but the interleaving overhead was worse than having them separate.

Module	**Embedded RAM Name**	**Size**	**Number of ports**	**Description**
DCT (FS)	S_Embedded_RAM	128x32 bits	2	Stores the fetched S values from SRAM
DCT (CT)	T_Embedded_RAM1	128x32 bits	2	Alternate with 2 and stores half the computed T values
DCT (CT)	T_Embedded_RAM2	128x32 bits	2	Alternate with 1 and stores half the computed T values
DCT (CSd)	Sd_Embedded_RAM1	128x32 bits	2	Alternate with 2 and all computed S' values for a given window.
DCT (CSd)	Sd_Embedded_RAM2	128x32 bits	2	Alternate with 1 and all computed S' values for the next window.

Milestone 3: Quantization and Lossless encoding

After we are done with applying the DCT for a block and storing it one of the 2 embedded rams, we can start to encode this block while the next block of DCT is being written to the other embedded ram. I followed a direct Scan-Encode approach to not iterate multiple times over the same block. For the scanning, two possible methods could be used (algorithmic and brute force). I tested both and reported the findings below.

Each address in the block can be composed by concatenating 2 counters for i and j as shown:

$$\begin{bmatrix} \{3'd0, 3'd0\} & \cdots & \{3'd0, 3'd7\} \\ \vdots & \ddots & \vdots \\ \{3'd7, 3'd0\} & \cdots & \{3'd7, 3'd7\} \end{bmatrix}$$

If we look at the matrix this way, then the scanning pattern can described be as follows:

$$\{3'd0, 3'd0\} \rightarrow \{3'd0, 3'd1\} \rightarrow \{3'd1, 3'd0\} \rightarrow \{3'd2, 3'd0\} \rightarrow \{3'd1, 3'd1\} \rightarrow \{3'd0, 3'd2\} \rightarrow \{3'd0, 3'd3\} \rightarrow \{3'd1, 3'd2\} \rightarrow \{3'd2, 3'd1\} \rightarrow \{3'd3, 3'd0\} \rightarrow \cdots$$

Hence the scanning algorithm can be deduced:

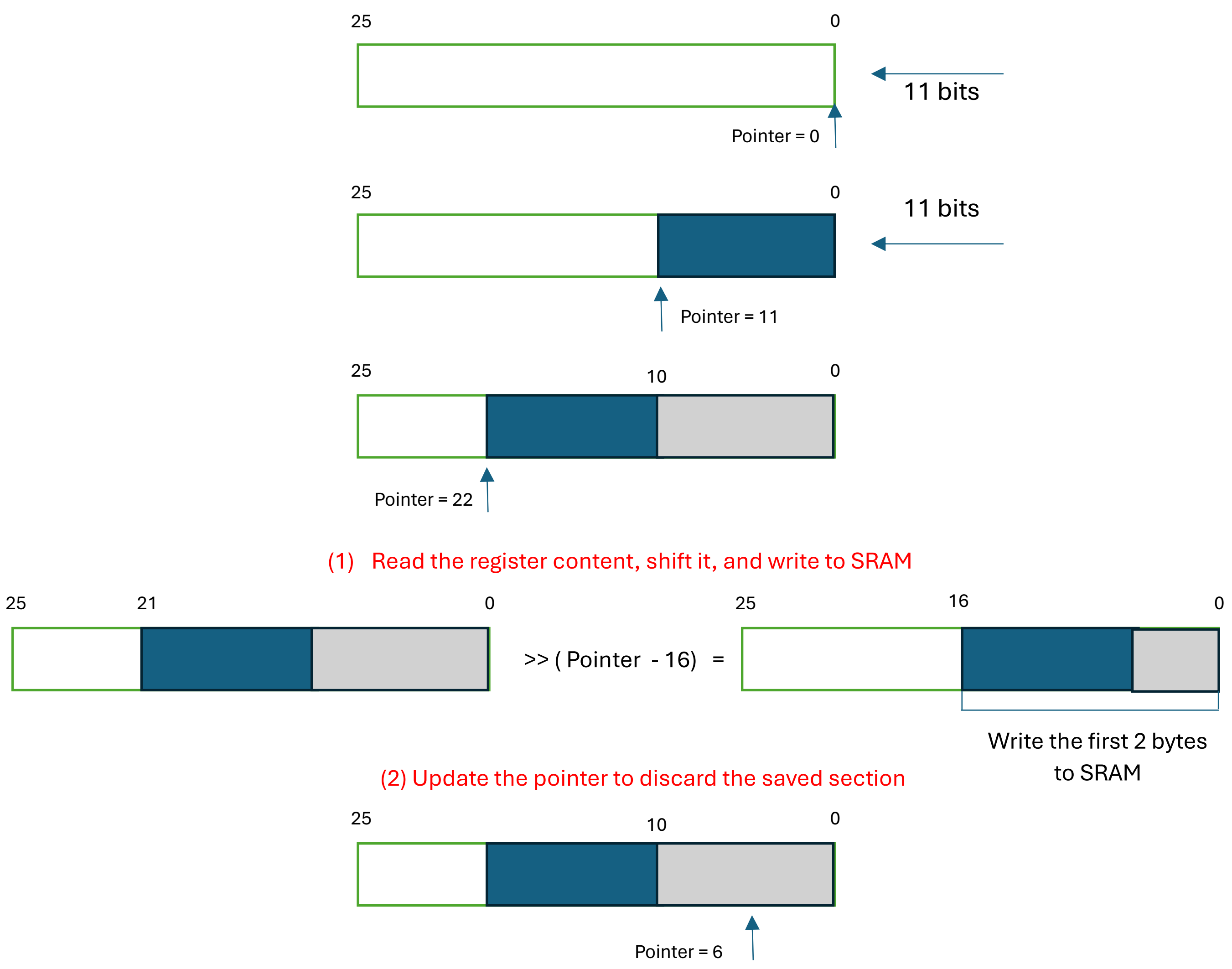
1. Initialize counters as 0 and a polarity bit as 0 "indicating j counter".
2. Increment the counter corresponding to the polarity bit.
3. Decrement the counter corresponding to the polarity bit while incrementing the other counter. Repeat this step until you reach 0.
4. Flip the polarity and repeat from step 2.

There are a couple of conditions that were also added to handle corner cases and termination. I implemented the above algorithm against the brute force (using a long switch case statement), and I found that this has lower utilization by 73 logic elements and 57 registers and doesn't affect the critical path. Therefore, I adopted this solution and moved to the next part.

For quantization, I used case statements to round and divide the scanned value by the corresponding coefficient in the chosen Q matrix based on the scanned element's diagonal index each diagonal elements share the same coefficient in the Q matrix.

For the lossless encoding part, I chose to have a shift-register of size 26 bits (chosen based on worst case scenario analysis). Each new encoded element gets pushed into this register and a counter "pointer" is incremented by the length of the pushed value. When the pointer is greater than or equal to 16, the value of the shift register is extracted, shifted by a variable amount, and written to the memory. The pointer is decreased by 16.

Of course, the actual implementation is more complicated than the diagram below as the register is dynamically taking new elements while outputting value at the same time, so the pointer moves by the difference of those length. Also, there are some corner cases that were handled.


25
0
11 bits
Pointer = 0
25
0
11 bits
Pointer = 11
25
10
0
Pointer = 22
(1) Read the register content, shift it, and write to SRAM
25
21
0
>> (Pointer - 16) =
25
16
0
Write the first 2 bytes to SRAM
(2) Update the pointer to discard the saved section
25
10
0
Pointer = 6

It's also worth noting that the shift register only shifts every bit by either 2, 5, or 11 positions as these are the only possible length of encoded values. This information was used to do a worst-case analysis for the optimal size of buffer.

Registers used in Milestone 3:

Module	**Register Name**	**Bits**	**Description**
DCT (QLE)	SRAM_address_QLE	20	Stores the current location to write the encoded values.
DCT (QLE)	SRAM_address_QLE_Buffer	20	Buffers the SRAM_address_QLE when I want to go write the channel offset bytes after the last block of each channel so that I can come back again where I left.
DCT (QLE)	QLE_j_counter, QLE_i_counter	3	Counters to scan over the S' block to encode the elements.
DCT (QLE)	polarity	1	Polarity bit used in the scanning algorithm as described earlier.
DCT (QLE)	diag_index	4	Carry the current scanned value's diagonal index so to be used in the quantization as explained earlier.

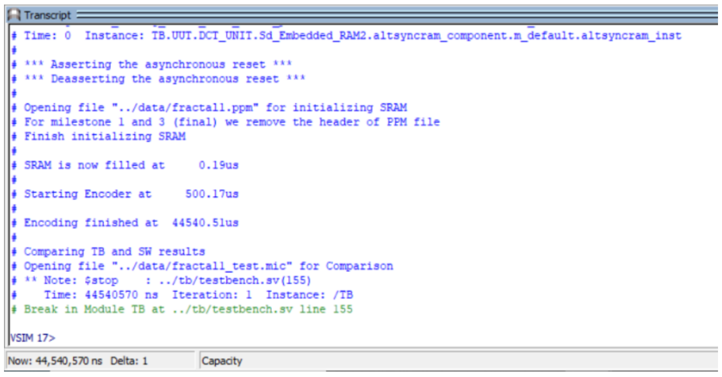
DCT (QLE)	zeros_counter	3	Count the number of zeros
DCT (QLE)	group_zeros_counter	3	Count the number of sets of 8 zeros
DCT (QLE)	Encoded_Buffer	26	The main shift register that hold and dispatch the encoded values to SRAM
DCT (QLE)	pointer	5	Accumulator for the number of bits in the buffer that were not yet saved in SRAM
DCT (QLE)	buffer_Quantized_clipped	9	Buffers the current Quantized element (nonzero) to go to release the accumulated zeros into the buffer and come back again and use it (Storing and retaining the previous address was more costly and complex).
DCT (QLE)	Two_Byte_Counter	19	Counter for the number of 2-bytes that have been stored so far in SRAM.
DCT (QLE)	done_releasing_zeros	1	Flag that I have finished releasing the accumulated zeros into the buffer so that I can retain the value from buffer_Quantized_clipped and use it.
DCT (QLE)	Hold_COM_0	1	Flag to stay in COM_0 for 1 more clock cycle to wait for the last fetched value from S' to get arrive and get processed.
DCT (QLE)	UV_Offset_index	1	Counter used to iterate and write the offset of (U or V) channels in two consecutive SRAM locations each.
DCT (QLE)	Header_Offset_index	3	Counter used to iterate and write the rest of the header in 6 consecutive SRAM locations.

Utilization and Timing Summary:

At last, these are the final solution utilization information:

Total logic elements	2,903 / 114,480 (3 %)
Total registers	997
Total pins	131 / 529 (25 %)
Total virtual pins	0
Total memory bits	16,128 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	8 / 532 (2 %)
Total PLLs	1 / 4 (25 %)

The design runs at a maximum frequency of 56.46 MHz and takes 44ms to decode an image of size 640x480 which is the maximum allowable size.



```
# Time: 0  Instance: TB.UUT.DCT_UNIT.Sd_Embedded_RAM2.altsyncram_component.m_default.altsyncram_inst
#
# *** Asserting the asynchronous reset ***
# *** Deasserting the asynchronous reset ***
#
# Opening file "../data/fractall.ppm" for initializing SRAM
# For milestone 1 and 3 (final) we remove the header of PPM file
# Finish initializing SRAM
#
# SRAM is now filled at     0.19us
#
# Starting Encoder at     500.17us
#
# Encoding finished at  44540.51us
#
# Comparing TB and SW results
# Opening file "../data/fractall_test.mic" for Comparison
# ** Note: $stop    : ../tb/testbench.sv(155)
#    Time: 44540570 ns  Iteration: 1  Instance: /TB
# Break in Module TB at ../tb/testbench.sv line 155

VSIM 17>
```

Now: 44,540,570 ns Delta: 1   Capacity