ENGS 31: JPEG Encoding Accelerator

Adam McQuilkin, '22

June 3rd, 2022

Contents

1 Introduction and Problem Statement

1.1 Introduction

The purpose of this final project for ENGS 031: Digital Electronics was to apply the skills we had learned throughout the course of the term to a real-world problem, with the intention of exposing us to the hardware development flow.

1.2 Problem Statement

For my final project, I chose to implement the JPEG image encoding algorithm in hardware. I was interested in working on a problem for which hardware would reasonably be applicable, and JPEG encoding is a problem for which hardware acceleration is very useful.

The problem statement for my project is as follows:

> The JPEG encoding algorithm is widely used but relatively time-complex to implement on a traditional processor. How could we accelerate this JPEG encoding process using an FPGA fabric?

1.3 JPEG Encoding Algorithm

For background, JPEG encoding is the process of compressing raw (RGB) image data in such a way that perceived image quality is maintained. JPEG encoding is a lossy compression algorithm, meaning image quality is lost after compression.

The JPEG encoding algorithm consists of the following steps:

Algorithm 1 JPEG Image Encoding

procedure EncodeImage(Img)
    Convert Img from RGB colorspace to YCbCr[1] colorspace
    Pad image with white pixels such that both the image width and image height are divisible by 8
    (Optional) Downsample individual channels based on human visual perception models[2]
    for each 8x8 block within Img **do**
        Shift pixel values from a positive range to a range centered around zero
        Run the M-D DCT-II[3] algorithm on the block
        Divide the resulting matrix by a predefined quantization matrix[4]
        Encode the resulting values using entropy encoding, run-length encoding, and huffman encoding
    end for
    Return processed image bitstream
end procedure

[1] https://en.wikipedia.org/wiki/YCbCr
[2] https://en.wikipedia.org/wiki/JPEG#Downsampling
[3] https://en.wikipedia.org/wiki/Discrete_cosine_transform#M-D_DCT-II
[4] https://en.wikipedia.org/wiki/JPEG#Quantization

For this project, I will focus on implementing the DCT step of the EncodeImage procedure above. The DCT is the most mathematically intense portion of the DCT algorithm and will require interfacing with prebuilt IP cores to build. Other elements of the algorithm could also effectively be parallelized, although since this is an educational project I will not build such modules as this would not include many learning opportunities not already presented by the DCT.

1.4 Discrete Cosine Transform

The Discrete Cosine Transform, or the DCT, is a transform that takes an image of size $N \times N$ and represents it as N^2 distinct cosine waves of varying frequency. The proof for this is outside the scope of this project, but is linked below[5].

The M-D DCT-II transform is shown below, where:

- u is the horizontal spatial frequency[6]for $u \in \{0..7\}$
- v is the horizontal spatial frequency for $v \in \{0..7\}$
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0 \\ 1 & \text{otherwise} \end{cases}$
- $g_{x,y}$ is the pixel value at image coordinates (x, y)
- $G_{u,v}$ is the DCT coefficient at image coordinates (u, v)

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right] \tag{1}$$

The DCT is extremely valuable within image compression algorithms because of its strong energy compaction property[7]. This is very useful since it allows for a significant reduction in the resulting size of a compressed image since many higher-frequency cosine components can be discarded with minimal loss in image quality. This discarding of data is accomplished within the quantization step after the image has been run through the DCT.

The following C++ code implements the DCT on an 8x8 image:

[5] https://en.wikipedia.org/wiki/Discrete_cosine_transform#Informal_overview
[6] https://en.wikipedia.org/wiki/Spatial_frequency
[7] http://ntur.lib.ntu.edu.tw/bitstream/246246/2007041910031915/1/00628094.pdf

```
#define IMAGE_WIDTH 8
#define IMAGE_HEIGHT 8

float dctOutput[IMAGE_WIDTH][IMAGE_HEIGHT];
int8_t data[IMAGE_WIDTH][IMAGE_HEIGHT] = {{93, 90, 83, 68, 61, 61, 46, 21},
                          {102, 92, 95, 77, 65, 60, 49, 32},
                          {69, 55, 47, 57, 65, 70, 72, 65},
                          {55, 55, 40, 42, 23, 1, 11, 38},
                          {55, 57, 47, 53, 35, 59, -2, 26},
                          {64, 41, 42, 55, 60, 57, 25, -8},
                          {77, 87, 58, -2, -5, 14, -10, -35},
                          {38, 14, 33, 33, -21, -23, -43, -34}};

float ci, cj, dct1, sum = 0;

/* ---- run DCT on each pixel in image ---- */
for (auto row = 0; row < IMAGE_WIDTH; ++row)
{
    for (auto col = 0; col < IMAGE_HEIGHT; ++col)
    {
        ci = (row == 0)
                ? 1.0 / std::sqrt(IMAGE_WIDTH)
                : std::sqrt(2) / sqrt(IMAGE_WIDTH);

        cj = (col == 0)
                ? 1.0 / std::sqrt(IMAGE_HEIGHT)
                : std::sqrt(2) / std::sqrt(IMAGE_HEIGHT);

        sum = 0;

        /* ---- iterate through each pixel in the image within DCT ---- */
        for (auto r = 0; r < IMAGE_WIDTH; ++r)
        {
            for (auto c = 0; c < IMAGE_HEIGHT; ++c)
            {
                dct1 = data[r][c] *
                        std::cos((2 * r + 1) * row * M_PI / (2 * IMAGE_WIDTH)) *
                        std::cos((2 * c + 1) * col * M_PI / (2 * IMAGE_HEIGHT));

                sum = sum + dct1;
            }
        }

        dctOutput[row][col] = ci * cj * sum;
    }
}
```

Figure 1: C++ code implementing M-D DCT-II on an 8x8 single-channel image

2 Design Solution

2.1 Project Specifications

As discussed in the section above, the deliverable for this project is a Xilinx IP block that implements the M-D DCT-II functionality as described in section 1.4. The following figure is a visualization of the specified IP block:

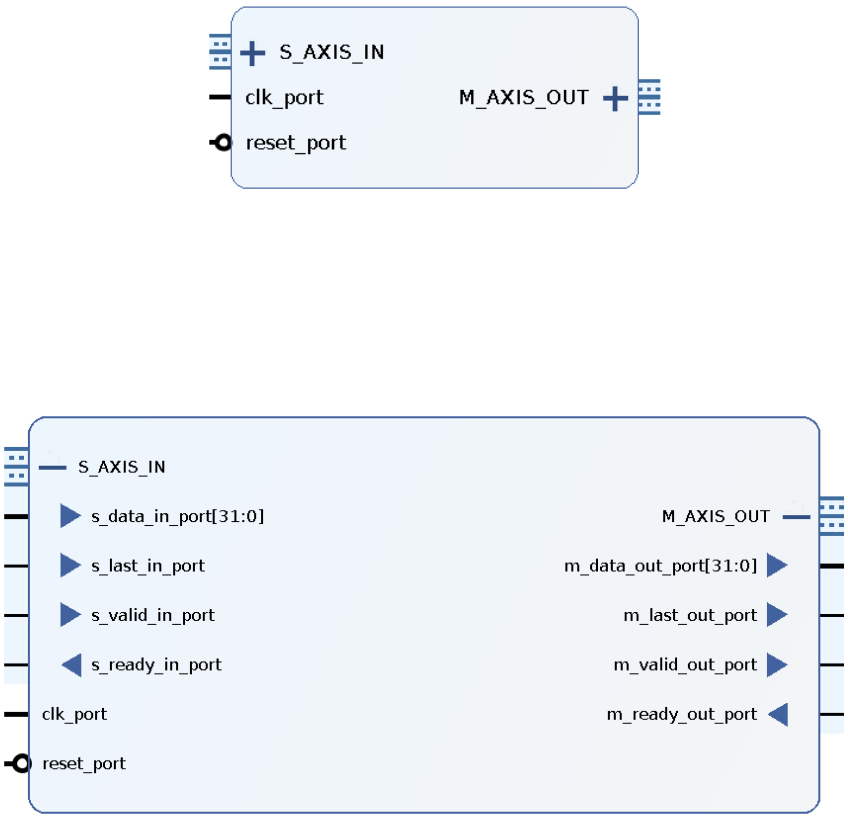


Figure 2: Diagram showing the ports contained within the DCT IP Block. Note that this block contains the non-mandatory AXI *last* line.

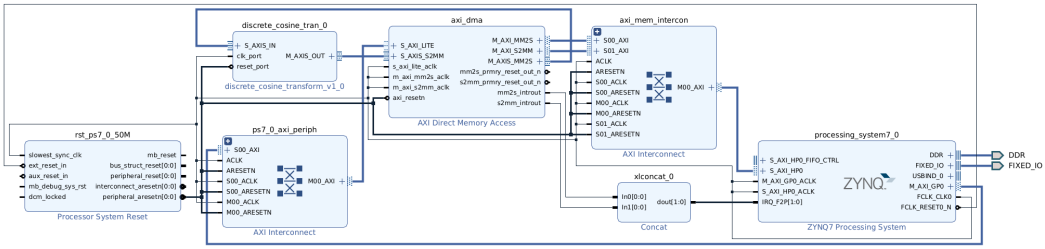


Figure 3: Diagram showing a typical use case of the DCT IP block (*discrete_cosine_tran_0* module).

As shown in figure 2.1, this Xilinx IP block will have two 32-bit AXI stream[8] ports in addition to clock and reset ports. The input port will accept 64 single-precision[9] floating-point numbers, where each 8 packets represents a row in the image. The output port will output 64 32-bit single-precision floating-point numbers in the same ordering as the input port. This is demonstrated in the table below:

(0,0)	(0,1)	⋮	(0,7)
(1,0)	(1,1)	⋮	(1,7)
...	...	⋱	...
(7,0)	(7,1)	⋮	(7,7)

Figure 4: Zero-indexed coordinates within a generic 8×8 single-channel image.

(0,0)	(0,1)	⋯	(0,7)	(1,0)	(1,1)	⋯	(7,0)	(7,1)	⋯	(7,7)

Figure 5: Resulting stream representation of the image represented in figure 4.

The clock port is used both to clock the IP block and to clock the output AXI stream port. The reset port is used to completely reset the block.

2.2 Operating Instructions

Currently this project has not been validated in hardware, and as such does not have any hardware operating instructions. To view this project in simulation, you will need to perform the following steps:

[8] https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol
[9] https://en.wikipedia.org/wiki/Single-precision_floating-point_format

1. Import all HDL files into the project
2. Import all testbench and waveform files into the project
3. Configure all Xilinx IP blocks
4. Set the top-level testbench as the top module within Vivado
5. Run a behavioral simulation for `4ms`

2.3 Theory of Operation

At its core, this DCT block is a series of interconnected FSMs that run incremental arithmetic operations on data that has been transmitted in over the AXI Stream input line. This flow is controlled by the top-level state machine, which handles the AXI data management and then delegates work to other IP cores. A state diagram for this FSM is shown below as well as in section 6.7.

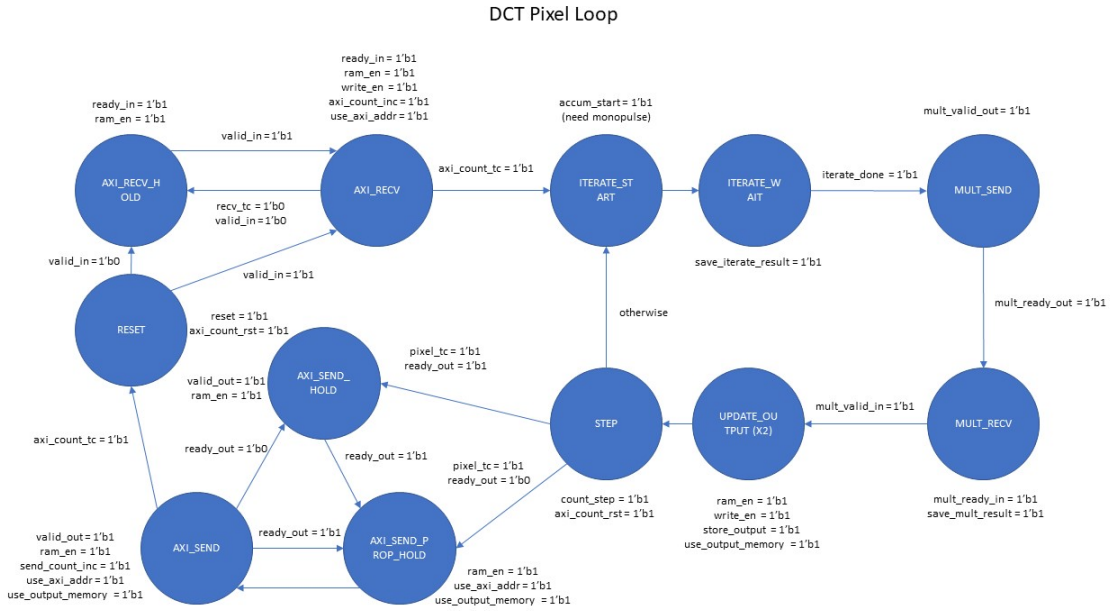


Figure 6: FSM state diagram for the top-level module of the DCT IP block.

Note that there are two FSM states for storing the calculated data since the BRAM block being used has a two clock-cycle delay for storing and receiving data.

2.3.1 Top-level Module

This top-level FSM delegates the work of calculating the DCT to the accumulate block, which for each pixel (u, v) in the image will iterate over each pixel (x, y) in the image as discussed in section 1.4. The accumulator will delegate the cosine calculations and the multiplication operations to the

multiplication module, and will accumulate the result of all multiplication operations into a single sum. This will then be passed back to the top-level module for storage.

The top-level module is tasked with completing the following calculation found in section 1.4 in addition to receiving and transmitting AXI data, where $R_{accum}(u, v)$ is the result of the accumulator module operating on (u, v):

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v) \cdot R_{accum}(u, v) \tag{2}$$

This means the top-level module needs to, for each pixel (u, v) in the image, calculate $G_{u,v}$ as shown above. This requires iteration through each pixel in the image, which is done with a dedicated pixel stepper module. When this module is enabled, it counts up a pixel each clock cycle. The top-level module's *STEP* state enables this stepper for one cycle each iteration.

The top-level module will trigger the accumulator module with the *ITERATE_START* state, after which it will wait for the accumulator to complete its processing. It will latch one of its internal operand registers to save the value coming out of the accumulator, which it will then send through an AXI multiplication block multiplying with the appropriate values of $\alpha(u)\alpha(v)$. There are four possible values for the product $\alpha(u)\alpha(v)$, meaning these can be trivially precomputed and save a multiplication operation. This precomputed value is then multiplied with the result of the accumulator operation and saved into the output memory block. This will repeat 64 times for an 8×8 image.

A simplified diagram of the datapath that implements this is shown below:

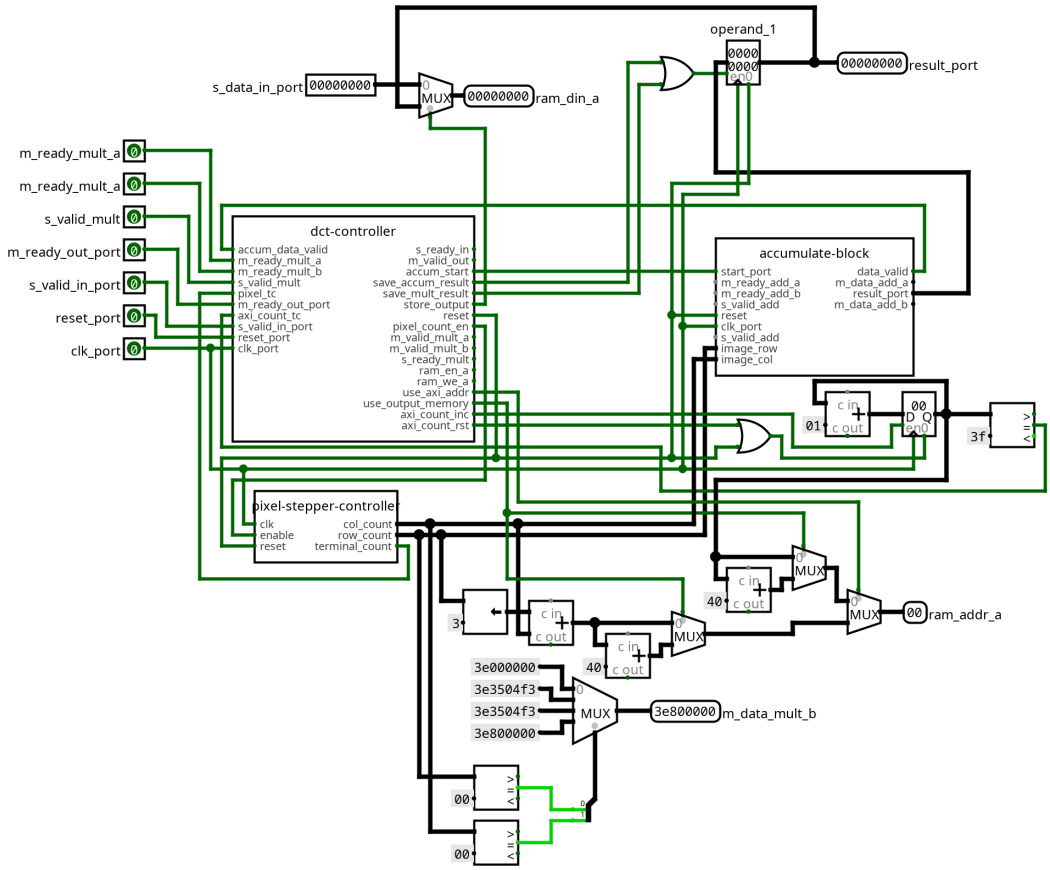


Figure 7: DCT Top-level Shell

2.3.2 Accumulator Module

The accumulation module is tasked with completing the following calculation found in section 1.4, where $R_{mult(u,v,x,y)}$ is the result of the multiplication module finding the DCT of pixel (u, v) in the image and calculating the value for the inner summation with (x, y):

$$R_{accum}(u, v) = \sum_{x=0}^{7} \sum_{y=0}^{7} R_{mult}(u, v, x, y) \tag{3}$$

The accumulator module is triggered by the top-level module, and will itself trigger the multiplication module. The accumulator module also uses a pixel stepper module, since the DCT calculation requires an $O\left(m^2 n^2\right)$ algorithm. The control FSM for the accumulator module is shown below:

The datapath that implements this functionality is shown below:

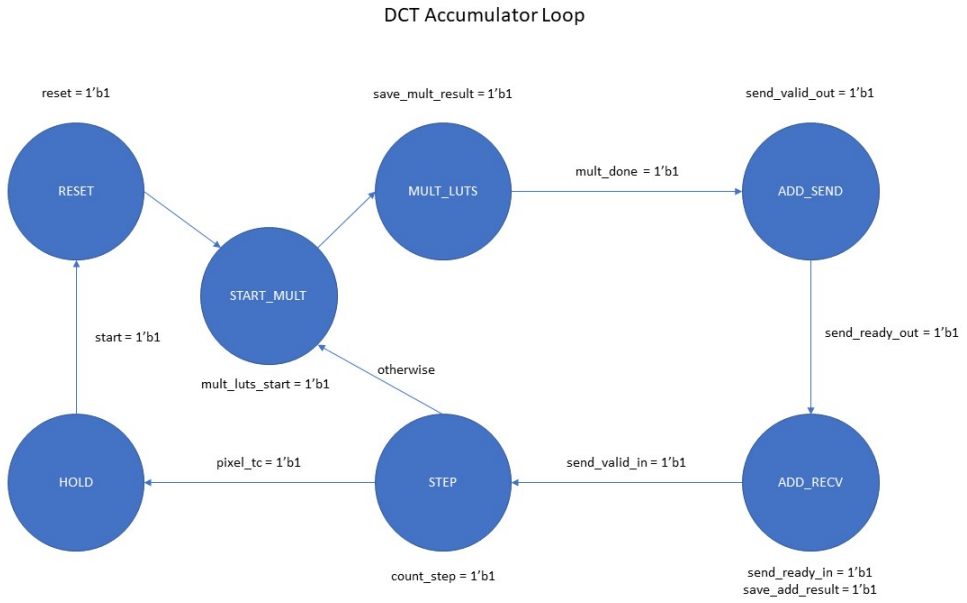


Figure 8: FSM state diagram for the DCT accumulator which runs the DCT for each pixel in the image.

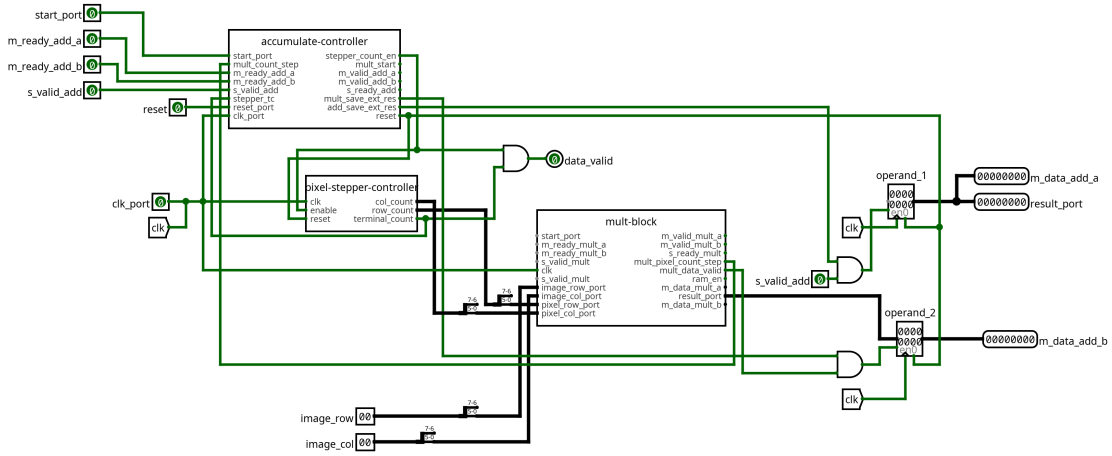


Figure 9: DCT Accumulator

2.3.3 Multiplication Module

The multiplication module is tasked with completing the following calculation found in section 1.4:

$$R_{mult}(u,v,x,y) = g_{x,y} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right] \tag{4}$$

The largest challenge with implementing the DCT in hardware is the required cosine calculations. Cosine calculations are computationally expensive, as they typically require one clock cycle for a (binary) decimal point of accuracy[10]. Since this module is looking for single-precision floating-point accuracy, this would require 32 clock cycles of computation for each cosine operation. This, compounded with the complexity of the required multiplication and division, makes this operation very inefficient if implemented completely as shown in figure 1.

With this in mind, it is vital to optimize this calculation. The important observation to make is that there are only 64 possible values for the equation $\cos\left[\frac{(2a+1)b\pi}{16}\right]$, since a and b can each individually take on 8 unique values. This means it is possible to load a ROM block with each possible value this expression can take on and index into the block using the values of a and b. This ROM can then be used to determine the required values of both cosine calculations, which saves two cosine operations, six multiplication operations, two division operations, and two addition operations. A MATLAB script that implements this ROM initialization is discussed in section 6.8.1.

The control FSM for the multiplication module is shown below:

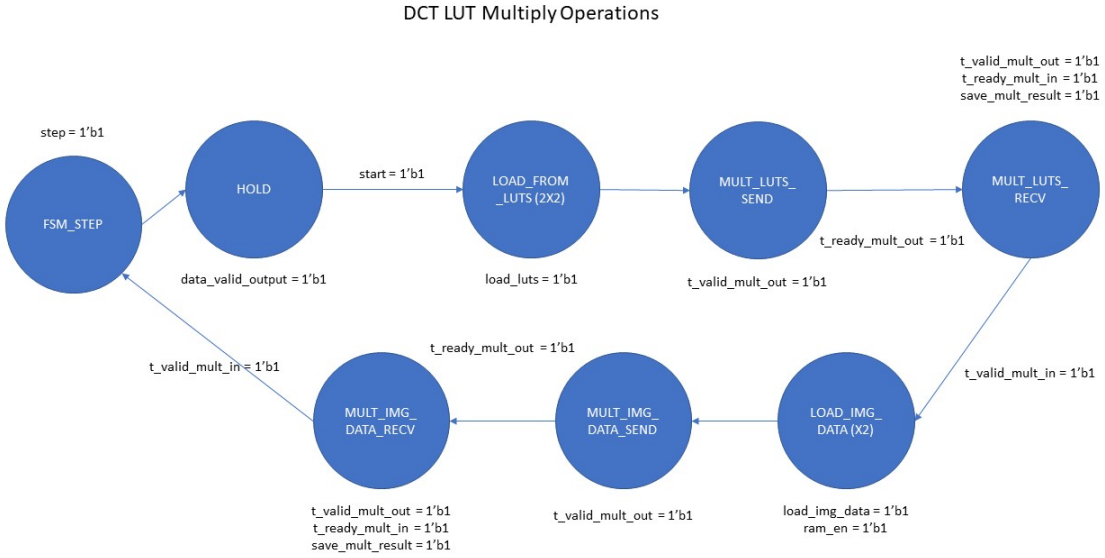


Figure 10: FSM state diagram for the multiplication module of the DCT design.

The datapath that implements this functionality is shown below:

[10] https://en.wikipedia.org/wiki/CORDIC

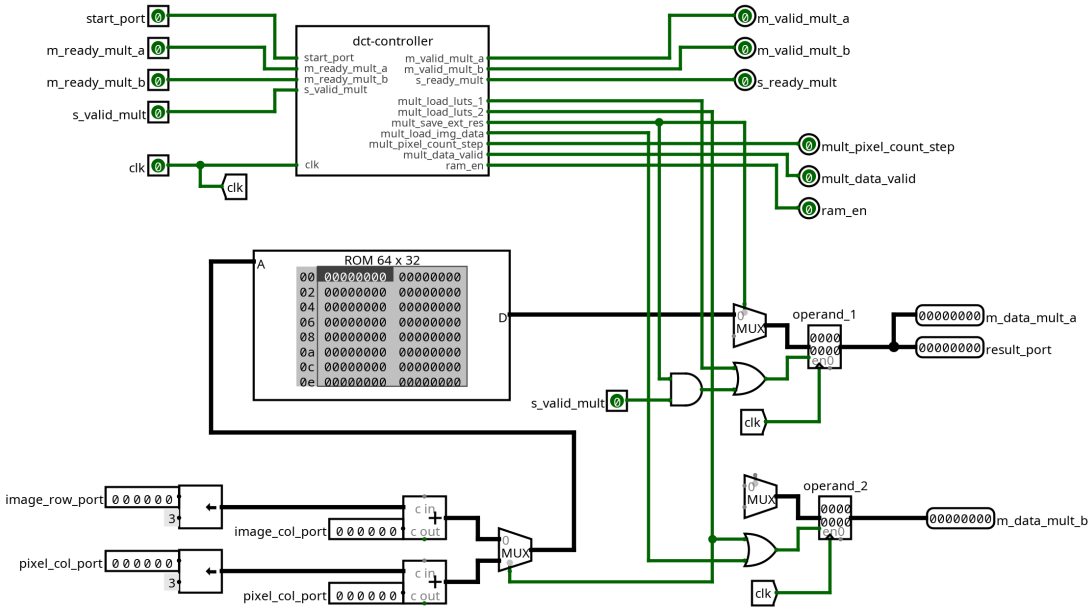


Figure 11: DCT Multiplication Operation

2.4 Language Choice

A notable feature of this project is that it is written in SystemVerilog[11]. This was a major choice early in the project to swtich away from VHDL, since the majority of the HDL experience surrounding the course pertains to VHDL. This choice meant that I could not receive materal development support from the course TAs nor the course faculty. This being said, SystemVerilog has the following two major advantages over VHDL for this project that made this choice worthwhile:

1. SystemVerilog has a C-style syntax, meaning HDL development could proceed much more quickly than with VHDL's less intuitive syntax. Since this project was a more ambitious undertaking than is expected from the class, this is a major advantage on a short timeline.

2. SystemVerilog is designed with highly robust RTL verification features, which are used extensively within Xilinx's IP cores. An example of this is an AXI stream validation IP (AXIS VIP)[12], which was used during the (ultimately unsuccessful) development of the microprocessor acceleration design.

As for improvements over standard Verilog, there are numerous improvements[13] made to the original Verilog standard in addition to the verification improvements mentioned above.

[11] https://en.wikipedia.org/wiki/SystemVerilog
[12] https://docs.xilinx.com/v/u/en-US/pg277-axi4stream-vip
[13] https://en.wikipedia.org/wiki/SystemVerilog#General_improvements_to_classical_Verilog

2.5 Construction and Debugging

At the beginning of this project, the problem of implementing the DCT was broken down into modular chunks that could be built using a bottom-up approach. As shown in figure 1, the DCT requires the following hardware modules:

1. A top-level system wrapper to interface outside of the block
2. A pixel iterator that counts through each pixel in the image
3. A multiplier module that implements the cosine calculations and multiplication with the current image pixel value
4. An accumulator that calls the multiplier module to calculate the value of $G_{u,v}$ for each pixel in the image (see section 1.4)
5. Floating-point mathematical operation cores (e.g. addition, multiplication)

This breakdown of modules was chosen to improve the testability of the system as well as to reduce the amount of repeated code within the project. For example, since this is an $O(m^2n^2)$ algorithm[14], the HDL code will need to iterate through the image pixel-by-pixel two times. As such, the iteration logic for this was broken out into its own module.

During development, modules would be developed and completely tested before continuing in the development flow. This ensured that any bugs would be contained within the module currently under development and that changes to previously developed modules would be minimized. This significantly sped up the development flow, since testing previous modules requires switching the top-level development file and switching the simulation context.

[14] https://en.wikipedia.org/wiki/Time_complexity

3 Design Evaluation

This project was largely experimental in nature, and as such was not guaranteed to succeed. Since the goal of this project was to implement a Xilinx IP block, the simplest way to validate this project in hardware was to incorporate the IP block into a block design within another project then build that project into hardware.

The complexity with this approach comes from the fact that an AXI Stream needs to be generated either from hardware or from software. The approach this project attempted to take was to use a Xilinx Soc development board and connect the DCT IP block, built in the Programmable Logic, to the Processing System[15] using the aforementioned AXI Stream protocol. This requires the usage of the Xilinx DMA IP core[16], which requires a moderately complex software stack on the microprocessor to implement correctly.

This software DMA implementation was the major limiting factor in the success of this project. According to all run simulations the project was successful in running the DCT on an 8x8 single-channel image, which then could have been used by the microprocessor to accelerate the JPEG encoding process. Due to time limitations on the project, this DMA driver development could not be completed. As such, the remainder of this section discusses the efficacy of the final design within the Vivado behavioral simulations.

3.1 AXI Data Input

The AXI Stream protocol in its most basic form requires three lines: a data line (32-bit wide in this design), a ready line, and a valid line. For an AXI input port, the transmitter will assert the valid line when it is ready to send data, and the receiver will assert the ready line when it is ready to receive data. When both lines are asserted, the transmitter transmits one data packet each clock cycle. If either line is deasserted, both lines wait to continue the transmission.

As shown in section 6.6, this design implements a functional AXI Stream receiver which takes in data and stores it within a block RAM (BRAM) block.

3.2 AXI Data Output

As with an AXI Stream receiver, an AXI Stream transmitter requires a data, ready, and valid line. As discussed above, the transmitter asserts the valid line when it is ready to transmit data and the receiver asserts the ready line when it is ready to receive data. Data is then transmitted when both lines are asserted.

As shown in section 6.6, this design implements a valid AXI Stream transmitter. The one notable element of the transmission waveforms is the transmitter repeatedly asserting and deasserting the valid line. This is to account for the fact that the BRAM containing the processed image data has a read delay of two clock cycles[17], and waiting a cycle to read from the BRAM was the most reliable method of transmitting the data out for a project MVP. This behavior is controlled by the FSM and can be easily modified as such.

[15] Xilinx Zynq-7000 SoC Documentation
[16] https://www.xilinx.com/products/intellectual-property/axi_dma.html#overview
[17] https://docs.xilinx.com/v/u/en-US/blk_mem_gen_ds512

3.3 Internal Data Processing

As discussed in section 6.6, this block successfully implements the required calculations after receiving data on the AXI Stream input line.

4 Conclusions and Next Steps

As mentioned above, due to time constraints and issues with required software packages this project was not able to be validated in hardware on a physical FPGA fabric. This being said, the project is fully functional according to behavioral simulation. This was not guaranteed that even the simulation would function since this project both relies heavily on external Xilinx IP cores and relies on correctly implementing the AXI stream protocol, two topics not discussed in class or required in standard projects. Additionally, this project was implemented in a langauge not taught in the standard curriculum, further adding to the experimental nature of this project.

This being said, I would still consider the project a nearly complete success. This project presented numerous learning opportunities both from technical and design perspectives, and I feel significantly more confident in my hardware design abilities. It is regrettable that the microprocessor drivers couldn't be fully implemented, but since the development of such drivers was outside of the scope of the class, I don't consider it a fundamental failure of the project.

If I were to redo this project in the future, I would look to design the system in such a way that it could be hardware validated during earlier steps of development. Simulations were helpful, but there were times in which design issues couldn't be discovered in simulation and only failed during implementation. For example, the FPGA fabric cannot support AXI lines of 512-bit width and this is something I didn't realize until significantly into the project.

Additionally, I would conduct more research into the existing ecosystem before committing to a design approach. I didn't realize at the beginning of this project that Xilinx offers tools for creating custom AXI hardware which would have greatly simplified the data transfer into and out of the IP block.This would have made the design more robust and compliant with more features of the AXI Stream protocol than with a completely custom solution.

For next steps of this project, I hope to revisit the design and continue development of the microprocessor DMA drivers. This will require more research into the PS/PL AXI ports and the DMA control IP cores, but should be feasible with enough time and patience. Additionally, it is possible to highly parallelize this design, which was not attempted in the MVP of this system. I also hope to reach out to Professor Stephen Taylor[18] regarding these problems.

I hope that this project can serve as an inspiration for future students looking for the confidence to attempt non-standard projects.

5 Acknowledgements

I would like to extend my gratitude to Benjamin Dobbins for his significant support, specifically helping me overcome development environment challenges and for his support on Xilinx IP integration best practices. I would also like to thank Professor Geoffrey Luke for his support with this non-standard project.

[18] https://engineering.dartmouth.edu/community/faculty/stephen-taylor

6 Appendices

6.1 System-level Block Diagrams

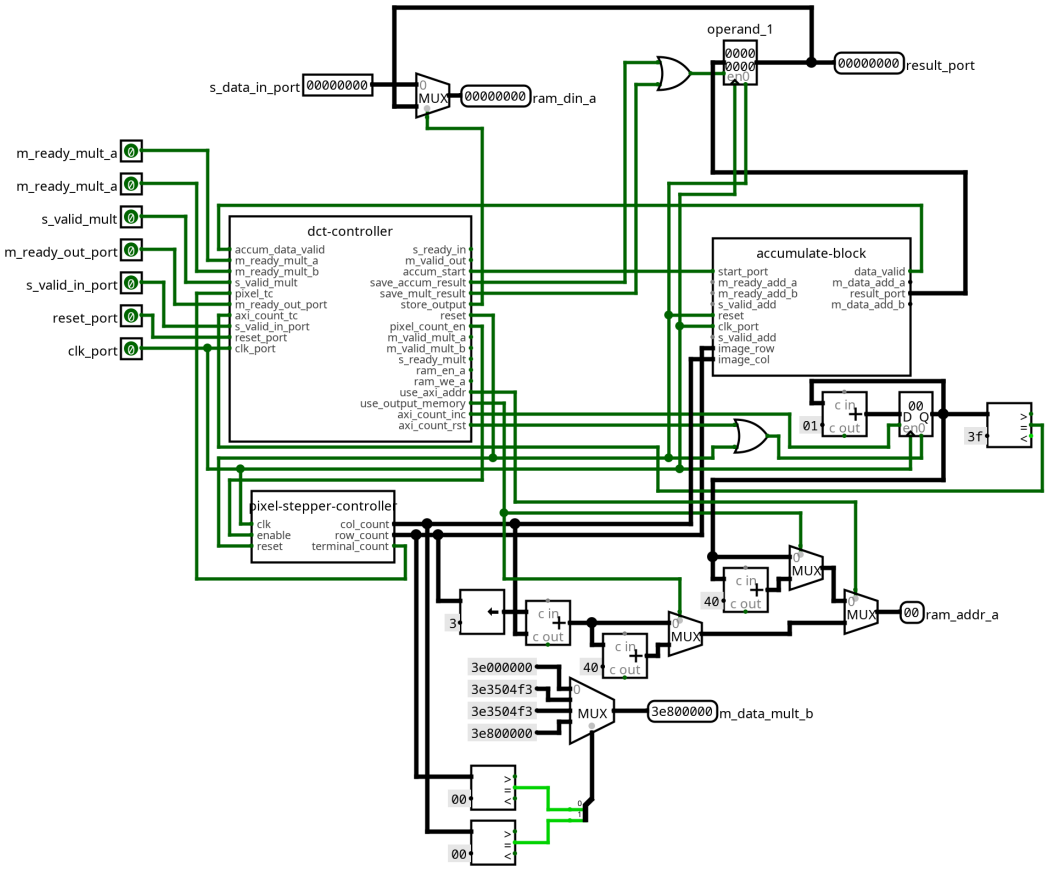


Figure 12: DCT Top-level Shell

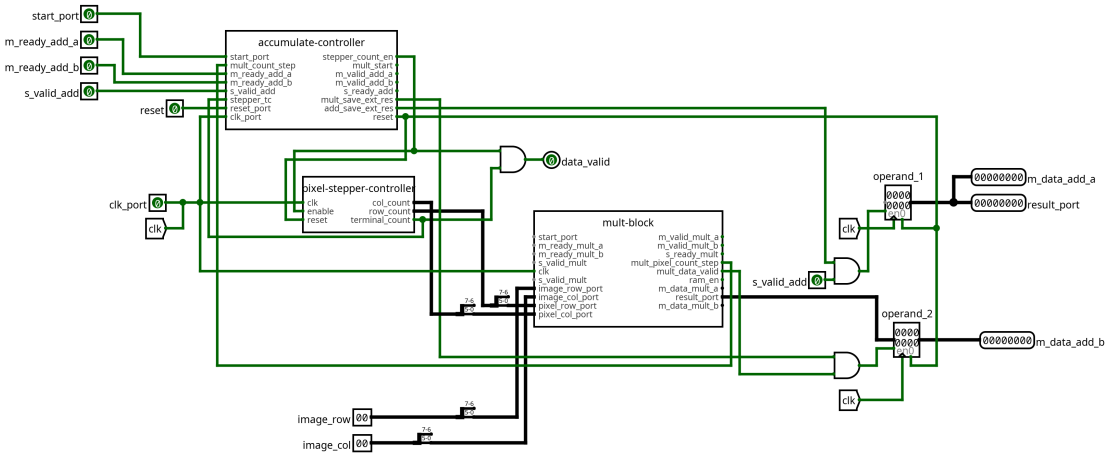


Figure 13: DCT Accumulator

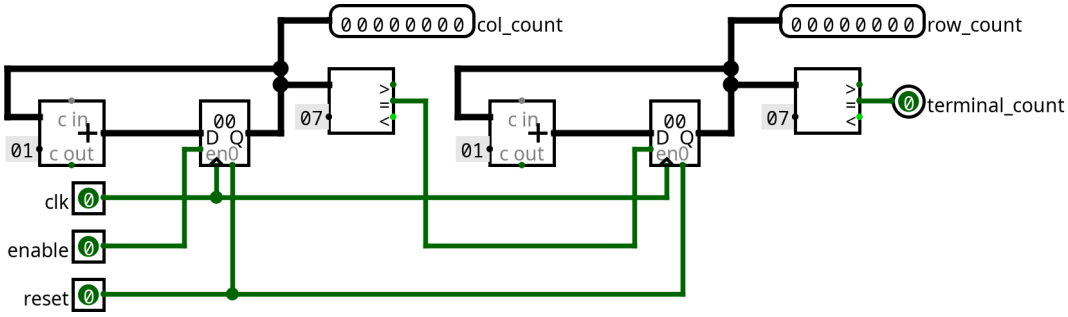


Figure 14: Pixel Stepper

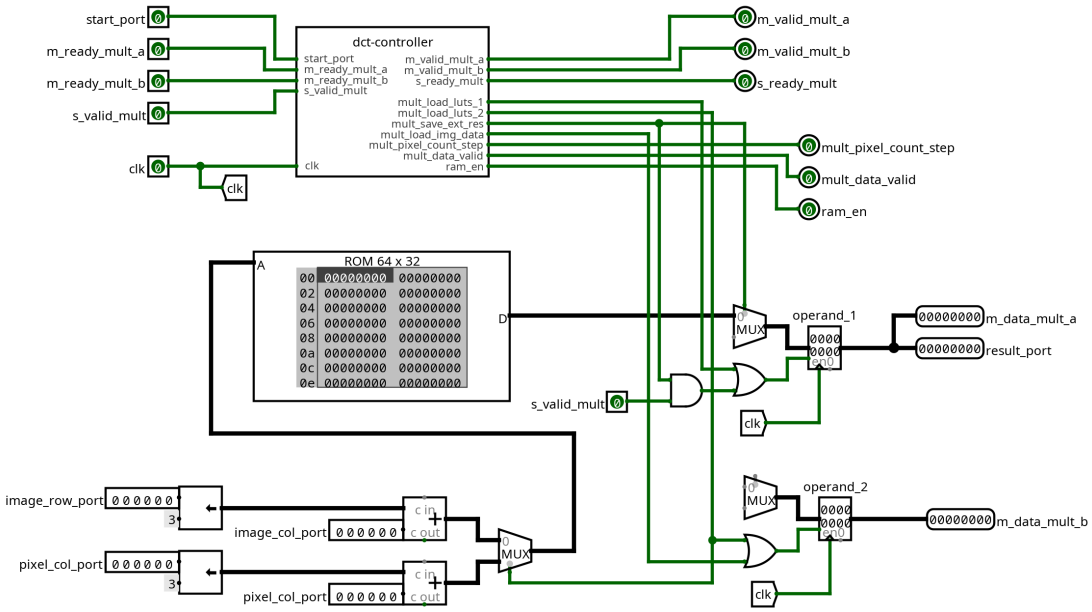


Figure 15: DCT Multiplication Operation

6.2 System HDL Code

6.2.1 discrete_cosine_transform.sv

```
1  module discrete_cosine_transform(
2      input clk_port,
3      input reset_port,
4      input s_valid_in_port, // input AXI lines
5      output s_ready_in_port,
6      input [31:0] s_data_in_port,
7      output m_valid_out_port, // output AXI lines
8      input m_ready_out_port,
9      output [31:0] m_data_out_port
10 );
11
12 // block AXI lines
13 reg s_ready_in = 1'b0;
14 reg m_valid_out = 1'b0;
15
16 reg [7:0] axi_count = 8'b0;
17 reg axi_count_inc = 1'b0;
18 reg axi_count_rst = 1'b0;
19 reg axi_count_tc;
20
21 reg use_axi_addr = 1'b0;
22 reg use_output_memory = 1'b0;
23
24 // fsm signal lines
25 reg save_accum_result = 1'b0;
26 reg save_mult_result = 1'b0;
27 reg store_output = 1'b0;
28 reg reset = 1'b0;
29
30 // image stepper lines
31 reg pixel_count_en = 1'b0;
32 wire [7:0] pixel_row_count;
33 wire [7:0] pixel_col_count;
34 wire pixel_tc;
35
36 // image accumulator lines
37 reg accum_start = 1'b0;
38 wire [31:0] accum_data;
39 wire accum_data_valid;
40
41 // floating point multiplication interface (OUT)
42 wire m_ready_mult_a;
43 reg m_valid_mult_a = 1'b0;
44 wire [31:0] m_data_mult_a;
45
46 wire m_ready_mult_b;
47 reg m_valid_mult_b = 1'b0;
48 wire [31:0] m_data_mult_b;
49
50 // floating point multiplication interface (IN)
51 reg s_ready_mult = 1'b0;
52 wire s_valid_mult;
53 wire [31:0] s_data_mult;
54
55 // ram lines (port a, for loading ram and saving results)
56 reg [6:0] ram_addr_a;
```

```
57  reg [31:0] ram_din_a = 32'b0;
58  wire [31:0] ram_dout_a;
59  reg ram_en_a = 1'b0;
60  reg ram_we_a = 1'b0;
61
62  // ram lines (port b, for reading image data in multiplier)
63  wire [6:0] ram_addr_b;
64  wire [31:0] ram_din_b;
65  wire [31:0] ram_dout_b;
66  wire ram_en_b;
67  wire ram_we_b;
68
69  typedef enum {
70      DCT_ITER_START,
71      DCT_ITER_WAIT,
72      DCT_MULT_SEND,
73      DCT_MULT_RECV,
74      DCT_UPDATE_OUTPUT,
75      DCT_STEP,
76      DCT_AXI_SEND_HOLD,
77      DCT_AXI_SEND_PROP_HOLD,
78      DCT_AXI_SEND,
79      DCT_RESET,
80      DCT_AXI_RECV_HOLD,
81      DCT_AXI_RECV
82  } dct_fsm_states;
83  dct_fsm_states dct_fsm_curr_state, dct_fsm_next_state = DCT_RESET;
84
85  wire [7:0] curr_pixel_addr;
86  wire [7:0] curr_output_addr;
87
88  reg [31:0] operand_1 = 32'b0;
89  reg [31:0] operand_2 = 32'b0;
90
91  reg [31:0] fr_fc_const = 32'h3e000000;
92  reg [31:0] fr_nfc_const = 31'h3e3504f3;
93  reg [31:0] nfr_fc_const = 32'h3e3504f3;
94  reg [31:0] nfr_nfc_const = 32'h3e800000;
95
96  // ------------------- output assignments ------------------- //
97
98  assign s_ready_in_port = s_ready_in;
99  assign m_valid_out_port = m_valid_out;
100
101 assign curr_pixel_addr = (pixel_row_count << 3) + pixel_col_count;
102
103 assign m_data_mult_a = operand_1;
104 assign m_data_mult_b = operand_2;
105
106 assign m_data_out_port = ram_dout_a; // only valid when axi flag high
107
108 // ------------------- component instantiation ------------------- //
109
110 pixel_stepper image_stepper(
111     .clk_port(clk_port),
112     .reset_port(reset),
113     .en_port(pixel_count_en),
114     .row_count_port(pixel_row_count),
115     .col_count_port(pixel_col_count),
116     .tc_port(pixel_tc));
```

```
117
118 dct_accumulate_operation accumulator(
119     .clk_port(clk_port),
120     .start_port(accum_start),
121     .reset_port(reset),
122     .ram_addr_port(ram_addr_b),
123     .ram_din_port(ram_din_b),
124     .ram_dout_port(ram_dout_b),
125     .ram_en_port(ram_en_b),
126     .ram_we_port(ram_we_b),
127     .image_row_port(pixel_row_count),
128     .image_col_port(pixel_col_count),
129     .result_port(accum_data),
130     .data_valid_port(accum_data_valid)
131     );
132
133 floating_point_mult mult_axi(
134     .aclk(clk_port),
135     .s_axis_a_tdata(m_data_mult_a),
136     .s_axis_a_tready(m_ready_mult_a),
137     .s_axis_a_tvalid(m_valid_mult_a),
138     .s_axis_b_tdata(m_data_mult_b),
139     .s_axis_b_tready(m_ready_mult_b),
140     .s_axis_b_tvalid(m_valid_mult_a),
141     .m_axis_result_tdata(s_data_mult),
142     .m_axis_result_tready(s_ready_mult),
143     .m_axis_result_tvalid(s_valid_mult));
144
145 image_data_ram image_block_ram(
146     .clka(clk_port),
147     .addra(ram_addr_a),
148     .dina(ram_din_a),
149     .douta(ram_dout_a),
150     .ena(ram_en_a),
151     .wea(ram_we_a),
152     .clkb(clk_port),
153     .addrb(ram_addr_b),
154     .dinb(ram_din_b),
155     .doutb(ram_dout_b),
156     .enb(ram_en_b),
157     .web(ram_we_b)
158     );
159
160 // ------------------ fsm control blocks ------------------ //
161
162 always_ff @(posedge clk_port) begin
163     dct_fsm_curr_state <= dct_fsm_next_state;
164 end
165
166 always_comb begin
167     dct_fsm_next_state <= dct_fsm_curr_state;
168
169     case (dct_fsm_curr_state)
170         DCT_ITER_START: begin
171             dct_fsm_next_state <= DCT_ITER_WAIT;
172         end
173
174         DCT_ITER_WAIT: begin
175             if (accum_data_valid == 1'b1) begin
176                 dct_fsm_next_state <= DCT_MULT_SEND;
```

```
177             end
178         end
179
180         DCT_MULT_SEND: begin
181             if (m_ready_mult_a == 1'b1 && m_ready_mult_b == 1'b1) begin
182                 dct_fsm_next_state <= DCT_MULT_RECV;
183             end
184         end
185
186         DCT_MULT_RECV: begin
187             if (s_valid_mult == 1'b1) begin
188                 dct_fsm_next_state <= DCT_UPDATE_OUTPUT;
189             end
190         end
191
192         DCT_UPDATE_OUTPUT: begin
193             dct_fsm_next_state <= DCT_STEP;
194         end
195
196         DCT_STEP: begin
197             if (pixel_tc == 1'b1 && m_ready_out_port == 1'b1) begin
198                 dct_fsm_next_state <= DCT_AXI_SEND_PROP_HOLD;
199             end else if (pixel_tc == 1'b1 && m_ready_out_port == 1'b0) begin
200                 dct_fsm_next_state <= DCT_AXI_SEND_HOLD;
201             end else begin
202                 dct_fsm_next_state <= DCT_ITER_START;
203             end
204         end
205
206         DCT_AXI_SEND_HOLD: begin
207             if (m_ready_out_port == 1'b1) begin
208                 dct_fsm_next_state <= DCT_AXI_SEND_PROP_HOLD;
209             end
210         end
211
212         DCT_AXI_SEND_PROP_HOLD: begin
213             dct_fsm_next_state <= DCT_AXI_SEND;
214         end
215
216         DCT_AXI_SEND: begin
217             if (axi_count_tc == 1'b1) begin
218                 dct_fsm_next_state <= DCT_RESET;
219             end else begin
220                 if (m_ready_out_port == 1'b0) begin
221                     dct_fsm_next_state <= DCT_AXI_SEND_HOLD;
222                 end else begin
223                     dct_fsm_next_state <= DCT_AXI_SEND_PROP_HOLD;
224                 end
225             end
226         end
227
228         DCT_RESET: begin
229             if (s_valid_in_port == 1'b1) begin
230                 dct_fsm_next_state <= DCT_AXI_RECV;
231             end else begin
232                 dct_fsm_next_state <= DCT_AXI_RECV_HOLD;
233             end
234         end
235
236         DCT_AXI_RECV_HOLD: begin
```

```
237                 if (s_valid_in_port == 1'b1) begin
238                     dct_fsm_next_state <= DCT_AXI_RECV;
239                 end
240             end
241 
242             DCT_AXI_RECV: begin
243                 if (axi_count_tc == 1'b1) begin
244                     dct_fsm_next_state <= DCT_ITER_START;
245                 end else if (s_valid_in_port == 1'b0) begin
246                     dct_fsm_next_state <= DCT_AXI_RECV_HOLD;
247                 end
248             end
249 
250             default: begin
251                 dct_fsm_next_state <= DCT_RESET;
252             end
253         endcase
254 
255         // global reset
256         if (reset_port == 1'b1) begin
257             dct_fsm_next_state <= DCT_RESET;
258         end
259     end
260 
261     always_comb begin
262         s_ready_in <= 1'b0;
263         m_valid_out <= 1'b0;
264         accum_start <= 1'b0;
265         save_accum_result <= 1'b0;
266         save_mult_result <= 1'b0;
267         store_output <= 1'b0;
268         reset <= 1'b0;
269         pixel_count_en <= 1'b0;
270         m_valid_mult_a <= 1'b0;
271         m_valid_mult_b <= 1'b0;
272         s_ready_mult <= 1'b0;
273         ram_en_a <= 1'b0;
274         ram_we_a <= 1'b0;
275 
276         use_axi_addr <= 1'b0;
277         use_output_memory <= 1'b0;
278 
279         axi_count_inc <= 1'b0;
280         axi_count_rst <= 1'b0;
281 
282         case (dct_fsm_curr_state)
283             DCT_ITER_START: begin
284                 accum_start <= 1'b1;
285             end
286 
287             DCT_ITER_WAIT: begin
288                 save_accum_result <= 1'b1;
289             end
290 
291             DCT_MULT_SEND: begin
292                 m_valid_mult_a <= 1'b1;
293                 m_valid_mult_b <= 1'b1;
294             end
295 
296             DCT_MULT_RECV: begin
```

```
297                 s_ready_mult <= 1'b1;
298                 save_mult_result <= 1'b1;
299             end
300
301             DCT_UPDATE_OUTPUT: begin
302                 ram_en_a <= 1'b1;
303                 ram_we_a <= 1'b1;
304                 store_output <= 1'b1;
305                 use_output_memory <= 1'b1;
306             end
307
308             DCT_STEP: begin
309                 pixel_count_en <= 1'b1;
310                 axi_count_rst <= 1'b1;
311             end
312
313             DCT_AXI_SEND_HOLD: begin
314                 m_valid_out <= 1'b1;
315                 ram_en_a <= 1'b1;
316             end
317
318             DCT_AXI_SEND_PROP_HOLD: begin
319                 ram_en_a <= 1'b1;
320                 use_axi_addr <= 1'b1;
321                 use_output_memory <= 1'b1;
322             end
323
324             DCT_AXI_SEND: begin
325                 m_valid_out <= 1'b1;
326                 ram_en_a <= 1'b1;
327                 axi_count_inc <= 1'b1;
328                 use_axi_addr <= 1'b1;
329                 use_output_memory <= 1'b1;
330             end
331
332             DCT_RESET: begin
333                 reset <= 1'b1;
334                 axi_count_rst <= 1'b1;
335             end
336
337             DCT_AXI_RECV_HOLD: begin
338                 s_ready_in <= 1'b1;
339                 ram_en_a <= 1'b1;
340             end
341
342             DCT_AXI_RECV: begin
343                 s_ready_in <= 1'b1;
344                 ram_en_a <= 1'b1;
345                 ram_we_a <= 1'b1;
346                 axi_count_inc <= 1'b1;
347                 use_axi_addr <= 1'b1;
348             end
349
350             default: begin
351                 // handled in initialization
352             end
353         endcase
354     end
355
356     // ---------- datapath blocks ----------
```

```
357
358 always_comb begin
359     if (pixel_row_count == 8'b0 && pixel_col_count == 8'b0) begin
360         // first row, first col
361         operand_2 <= fr_fc_const;
362     end else if (pixel_row_count == 8'b0 && pixel_col_count != 8'b0) begin
363         // first row, not first col
364         operand_2 <= fr_nfc_const;
365     end else if (pixel_row_count != 8'b0 && pixel_col_count == 8'b0) begin
366         // not first row, first col
367         operand_2 <= nfr_fc_const;
368     end else begin
369         // not first row, not first col
370         operand_2 <= nfr_nfc_const;
371     end
372 end
373
374 always_comb begin
375     ram_addr_a <= curr_pixel_addr; // first half of memory block
376
377     if (use_output_memory == 1'b1) begin
378         ram_addr_a <= curr_pixel_addr + 8'd64; // second half of memory block
379     end
380
381     if (use_axi_addr == 1'b1) begin
382         ram_addr_a <= axi_count; // first half of memory block
383
384         if (use_output_memory == 1'b1) begin
385             ram_addr_a <= axi_count + 8'd64; // second half of memory block
386         end
387     end
388 end
389
390 always_latch begin
391     if (save_accum_result == 1'b1) begin // && accum_data_valid == 1'b1
392         operand_1 <= accum_data;
393     end
394
395     if (save_mult_result == 1'b1 && s_valid_mult == 1'b1) begin
396         operand_1 <= accum_data;
397     end
398 end
399
400 always_comb begin
401     ram_din_a <= s_data_in_port;
402
403     if (store_output == 1'b1) begin
404         ram_din_a <= operand_1;
405     end
406 end
407
408 always_comb begin
409     axi_count_tc <= 1'b0;
410
411     if (axi_count == 8'd63) begin
412         axi_count_tc <= 1'b1;
413     end
414 end
415
416 always_ff @(posedge clk_port) begin
```

```
417         if (axi_count_inc == 1'b1) begin
418             axi_count <= axi_count + 1;
419         end
420 
421         if (axi_count_rst == 1'b1) begin
422             axi_count <= 8'b0;
423         end
424     end
425     endmodule
```

6.2.2 dct_accumulate_operation.sv

```
1   module dct_accumulate_operation(
2       input clk_port,
3       input start_port,
4       input reset_port,
5       input [7:0] image_row_port,
6       input [7:0] image_col_port,
7       output [6:0] ram_addr_port,
8       output [31:0] ram_din_port,
9       input [31:0] ram_dout_port,
10      output ram_en_port,
11      output ram_we_port,
12      output reg [31:0] result_port,
13      output reg data_valid_port
14  );
15
16  typedef enum {
17      ADD_HOLD,
18      ADD_RESET,
19      ADD_START_MULT,
20      ADD_WAIT_MULT,
21      ADD_SEND,
22      ADD_RECV,
23      ADD_STEP
24  } accumulator_fsm_states;
25  accumulator_fsm_states add_fsm_curr_state, add_fsm_next_state = ADD_HOLD;
26
27  // image stepper lines
28  reg stepper_count_en = 1'b0;
29  wire [7:0] stepper_row_count;
30  wire [7:0] stepper_col_count;
31  wire stepper_tc;
32
33  // LUT multiplier lines
34  reg mult_start = 1'b0;
35  wire [31:0] mult_result;
36  wire mult_data_valid;
37  wire mult_count_step;
38
39  // floating point addition interface (OUT)
40  wire m_ready_add_a;
41  reg m_valid_add_a = 1'b0;
42  reg [31:0] m_data_add_a;
43
44  wire m_ready_add_b;
45  reg m_valid_add_b = 1'b0;
46  reg [31:0] m_data_add_b;
47
```

```
48  // floating point addition interface (IN)
49  reg s_ready_add = 1'b0;
50  wire s_valid_add;
51  wire [31:0] s_data_add;
52
53  reg mult_save_ext_res = 1'b0;
54  reg add_save_ext_res = 1'b0;
55  reg data_valid = 1'b0;
56  reg reset = 1'b0;
57
58  // output registers
59  reg [31:0] operand_1 = 32'b0; // holds previous state
60  reg [31:0] operand_2 = 32'b0; // updated for each operation
61
62  // ------------------ output assignments ------------------ //
63
64  always_comb begin
65      result_port = operand_1;
66      m_data_add_a = operand_1;
67      m_data_add_b = operand_2;
68      data_valid_port = data_valid;
69  end
70
71  // ------------------ component instantiation ------------------ //
72
73  pixel_stepper image_stepper(
74      .clk_port(clk_port),
75      .reset_port(reset),
76      .en_port(stepper_count_en),
77      .row_count_port(stepper_row_count),
78      .col_count_port(stepper_col_count),
79      .tc_port(stepper_tc));
80
81  dct_multiply_operation mult_op(
82      .clk_port(clk_port),
83      .start_port(mult_start),
84      .ram_addr_port(ram_addr_port),
85      .ram_din_port(ram_din_port),
86      .ram_dout_port(ram_dout_port),
87      .ram_en_port(ram_en_port),
88      .ram_we_port(ram_we_port),
89      .image_row_port(image_row_port),
90      .image_col_port(image_col_port),
91      .pixel_row_port(stepper_row_count),
92      .pixel_col_port(stepper_col_count),
93      .result_port(mult_result),
94      .mult_data_valid_port(mult_data_valid),
95      .pixel_count_step_port(mult_count_step));
96
97  floating_point_add float_adder_axi(
98      .aclk(clk_port),
99      .s_axis_a_tdata(m_data_add_a),
100     .s_axis_a_tready(m_ready_add_a),
101     .s_axis_a_tvalid(m_valid_add_a),
102     .s_axis_b_tdata(m_data_add_b),
103     .s_axis_b_tready(m_ready_add_b),
104     .s_axis_b_tvalid(m_valid_add_a),
105     .m_axis_result_tdata(s_data_add),
106     .m_axis_result_tready(s_ready_add),
107     .m_axis_result_tvalid(s_valid_add));
```

```
108
109 // ---------- fsm control blocks ---------- //
110
111 always_ff @(posedge clk_port) begin
112     add_fsm_curr_state <= add_fsm_next_state;
113 end
114
115 always_comb begin // next state update logic
116     add_fsm_next_state <= add_fsm_curr_state;
117
118     case (add_fsm_curr_state)
119         ADD_HOLD: begin
120             if (start_port == 1'b1) begin
121                 add_fsm_next_state <= ADD_RESET;
122             end
123         end
124
125         ADD_RESET: begin
126             add_fsm_next_state <= ADD_START_MULT;
127         end
128
129         ADD_START_MULT: begin
130             add_fsm_next_state <= ADD_WAIT_MULT;
131         end
132
133         ADD_WAIT_MULT: begin
134             if (mult_count_step == 1'b1) begin
135                 add_fsm_next_state <= ADD_SEND;
136             end
137         end
138
139         ADD_SEND: begin
140             if (m_ready_add_a == 1'b1 && m_ready_add_b == 1'b1) begin
141                 add_fsm_next_state <= ADD_RECV;
142             end
143         end
144
145         ADD_RECV: begin
146             if (s_valid_add == 1'b1) begin
147                 add_fsm_next_state <= ADD_STEP;
148             end
149         end
150
151         ADD_STEP: begin
152             if (stepper_tc == 1'b1) begin
153                 add_fsm_next_state <= ADD_HOLD;
154             end else begin
155                 add_fsm_next_state <= ADD_START_MULT;
156             end
157         end
158
159         default: begin
160             add_fsm_next_state <= ADD_HOLD;
161         end
162     endcase
163
164     // global reset
165     if (reset_port == 1'b1) begin
166         add_fsm_next_state <= ADD_RESET;
167     end
```

```
168 end
169
170 always_comb begin // fsm control line update logic
171     stepper_count_en <= 1'b0;
172     mult_start <= 1'b0;
173     m_valid_add_a <= 1'b0;
174     m_valid_add_b <= 1'b0;
175     s_ready_add <= 1'b0;
176     mult_save_ext_res <= 1'b0;
177     add_save_ext_res <= 1'b0;
178     reset <= 1'b0;
179
180     case (add_fsm_curr_state)
181         ADD_HOLD: begin
182             // no outputs
183         end
184
185         ADD_RESET: begin
186             reset <= 1'b1;
187         end
188
189         ADD_START_MULT: begin
190             mult_start <= 1'b1;
191         end
192
193         ADD_WAIT_MULT: begin
194             mult_save_ext_res <= 1'b1;
195         end
196
197         ADD_SEND: begin
198             m_valid_add_a <= 1'b1;
199             m_valid_add_b <= 1'b1;
200         end
201
202         ADD_RECV: begin
203             s_ready_add <= 1'b1;
204             add_save_ext_res <= 1'b1;
205         end
206
207         ADD_STEP: begin
208             stepper_count_en <= 1'b1;
209         end
210
211         default: begin
212             // handled in initialization
213         end
214     endcase
215 end
216
217 // ------------------ datapath blocks ------------------ //
218
219 always_latch begin
220     if (mult_save_ext_res == 1'b1 && mult_data_valid == 1'b1) begin
221         operand_2 <= mult_result;
222     end
223
224     if (add_save_ext_res == 1'b1 && s_valid_add == 1'b1) begin
225         operand_1 <= s_data_add;
226     end
227
```

```
228     if (reset == 1'b1) begin
229         operand_1 <= 32'b0;
230         operand_2 <= 32'b0;
231     end
232 end
233 
234 always_comb begin
235     data_valid <= 1'b0;
236 
237     if (stepper_count_en == 1'b1 && stepper_tc == 1'b1) begin
238         data_valid <= 1'b1;
239     end
240 end
241 endmodule
```

6.2.3 pixel_stepper.sv

```
1  module pixel_stepper(
2      input clk_port,
3      input en_port,
4      input reset_port,
5      output [7:0] row_count_port,
6      output [7:0] col_count_port,
7      output tc_port
8  );
9  
10 reg [7:0] row_count = 3'b0;
11 reg [7:0] col_count = 3'b0;
12 reg tc = 1'b0;
13 
14 assign row_count_port = row_count;
15 assign col_count_port = col_count;
16 assign tc_port = tc;
17 
18 always_ff @(posedge clk_port) begin
19     if (reset_port == 1'b1) begin
20         row_count <= 3'b0;
21         col_count <= 3'b0;
22     end else if (en_port == 1'b1) begin
23         col_count <= col_count + 1'b1;
24 
25         if (col_count == 3'd7) begin
26             col_count <= 3'b0;
27             row_count <= row_count + 1'b1;
28 
29             if (row_count == 3'd7) begin
30                 row_count <= 3'b0;
31             end
32         end
33     end
34 end
35 
36 always_comb begin
37     tc <= 1'b0;
38 
39     if (col_count == 3'd7 && row_count == 3'd7) begin
40         tc <= 1'b1;
41     end
42 end
```

```
43 endmodule
```

6.2.4 dct_multiply_operation.sv

```
1  module dct_multiply_operation(
2      input clk_port,
3      input start_port,
4      output [6:0] ram_addr_port,
5      output [31:0] ram_din_port,
6      input [31:0] ram_dout_port,
7      output ram_en_port,
8      output ram_we_port,
9      input [7:0] image_row_port,
10     input [7:0] image_col_port,
11     input [7:0] pixel_row_port,
12     input [7:0] pixel_col_port,
13     output [31:0] result_port,
14     output mult_data_valid_port,
15     output pixel_count_step_port
16 );
17
18 // LUT interface
19 reg [5:0] lut_addr = 6'b0;
20 reg lut_en = 1'b1; // LUT always enabled
21 wire [31:0] lut_dout;
22
23 // floating point multiplication interface (OUT)
24 wire m_ready_mult_a;
25 reg m_valid_mult_a = 1'b0;
26 wire [31:0] m_data_mult_a;
27
28 wire m_ready_mult_b;
29 reg m_valid_mult_b = 1'b0;
30 wire [31:0] m_data_mult_b;
31
32 // floating point multiplication interface (IN)
33 reg s_ready_mult = 1'b0;
34 wire s_valid_mult;
35 wire [31:0] s_data_mult;
36
37 // fsm control lines
38 reg mult_load_luts_1 = 1'b0;
39 reg mult_load_luts_2 = 1'b0;
40 reg mult_save_ext_res = 1'b0;
41 reg mult_load_img_data = 1'b0;
42 reg mult_data_valid = 1'b0;
43 reg mult_pixel_count_step = 1'b0;
44
45 // ram control lines
46 reg [31:0] ram_din = 32'b0; // never writing to RAM in this block
47 reg ram_en = 1'b0;
48 reg ram_we = 1'b0; // we don't write to RAM in this block
49
50 // data registers
51 reg [31:0] operand_1 = 32'b0; // holds previous state
52 reg [31:0] operand_2 = 32'b0; // updated for each operation
53
54 typedef enum {
55     MULT_HOLD,
```

```
56          MULT_LOAD_FROM_LUTS_1a,
57          MULT_LOAD_FROM_LUTS_1b,
58          MULT_LOAD_FROM_LUTS_2a,
59          MULT_LOAD_FROM_LUTS_2b,
60          MULT_LUTS_SEND,
61          MULT_LUTS_RECV,
62          MULT_LOAD_IMG_DATA_1a,
63          MULT_LOAD_IMG_DATA_1b,
64          MULT_IMG_DATA_SEND,
65          MULT_IMG_DATA_RECV,
66          MULT_COUNT_STEP
67      } multiplication_fsm_states;
68      multiplication_fsm_states mult_fsm_curr_state, mult_fsm_next_state = MULT_HOLD;
69
70      // ------------------- output assignments ------------------- //
71
72      assign result_port = operand_1;
73      assign mult_data_valid_port = mult_data_valid;
74      assign pixel_count_step_port = mult_pixel_count_step;
75
76      assign ram_addr_port = (pixel_row_port << 3) + pixel_col_port;
77      assign ram_din_port = ram_din;
78      assign ram_en_port = ram_en;
79      assign ram_we_port = ram_we;
80
81      assign m_data_mult_a = operand_1;
82      assign m_data_mult_b = operand_2;
83
84      // ------------------- component instantiation ------------------- //
85
86      dct_cos_lookup lut(
87          .addra(lut_addr),
88          .clka(clk_port),
89          .douta(lut_dout),
90          .ena(lut_en));
91
92      floating_point_mult mult_axi(
93          .aclk(clk_port),
94          .s_axis_a_tdata(m_data_mult_a),
95          .s_axis_a_tready(m_ready_mult_a),
96          .s_axis_a_tvalid(m_valid_mult_a),
97          .s_axis_b_tdata(m_data_mult_b),
98          .s_axis_b_tready(m_ready_mult_b),
99          .s_axis_b_tvalid(m_valid_mult_a),
100         .m_axis_result_tdata(s_data_mult),
101         .m_axis_result_tready(s_ready_mult),
102         .m_axis_result_tvalid(s_valid_mult));
103
104     // ------------------- fsm control blocks ------------------- //
105
106     always_ff @(posedge clk_port) begin
107         mult_fsm_curr_state <= mult_fsm_next_state;
108     end
109
110     always_comb begin // update multiplication fsm next state
111         mult_fsm_next_state <= mult_fsm_curr_state;
112
113         case (mult_fsm_curr_state)
114             MULT_HOLD: begin
115                 if (start_port == 1'b1) begin
```

```
116                 mult_fsm_next_state <= MULT_LOAD_FROM_LUTS_1a;
117             end
118         end
119
120         MULT_LOAD_FROM_LUTS_1a: begin
121             mult_fsm_next_state <= MULT_LOAD_FROM_LUTS_1b;
122         end
123
124         MULT_LOAD_FROM_LUTS_1b: begin
125             mult_fsm_next_state <= MULT_LOAD_FROM_LUTS_2a;
126         end
127
128         MULT_LOAD_FROM_LUTS_2a: begin
129             mult_fsm_next_state <= MULT_LOAD_FROM_LUTS_2b;
130         end
131
132         MULT_LOAD_FROM_LUTS_2b: begin
133             mult_fsm_next_state <= MULT_LUTS_SEND;
134         end
135
136         MULT_LUTS_SEND: begin
137             // both AXI input ports ready
138             if (m_ready_mult_a == 1'b1 && m_ready_mult_b == 1'b1) begin
139                 mult_fsm_next_state <= MULT_LUTS_RECV;
140             end
141         end
142
143         MULT_LUTS_RECV: begin
144             if (s_valid_mult == 1'b1) begin // AXI mult output data valid
145                 mult_fsm_next_state <= MULT_LOAD_IMG_DATA_1a;
146             end
147         end
148
149         MULT_LOAD_IMG_DATA_1a: begin
150             mult_fsm_next_state <= MULT_LOAD_IMG_DATA_1b;
151         end
152
153         MULT_LOAD_IMG_DATA_1b: begin
154             mult_fsm_next_state <= MULT_IMG_DATA_SEND;
155         end
156
157         MULT_IMG_DATA_SEND: begin
158             // both AXI input ports ready
159             if (m_ready_mult_a == 1'b1 && m_ready_mult_b == 1'b1) begin
160                 mult_fsm_next_state <= MULT_IMG_DATA_RECV;
161             end
162         end
163
164         MULT_IMG_DATA_RECV: begin
165             if (s_valid_mult == 1'b1) begin
166                 mult_fsm_next_state <= MULT_COUNT_STEP;
167             end
168         end
169
170         MULT_COUNT_STEP: begin
171             mult_fsm_next_state <= MULT_HOLD;
172         end
173
174         default: begin
175             mult_fsm_next_state <= MULT_HOLD;
```

```
176             end
177         endcase
178     end
179
180     always_comb begin // update multiplication fsm outputs @(mult_fsm_curr_state)
181         // AXI interface lines
182         m_valid_mult_a <= 1'b0;
183         m_valid_mult_b <= 1'b0;
184         s_ready_mult <= 1'b0;
185
186         // control lines
187         mult_load_luts_1 <= 1'b0;
188         mult_load_luts_2 <= 1'b0;
189         mult_save_ext_res <= 1'b0;
190         mult_load_img_data <= 1'b0;
191         mult_pixel_count_step <= 1'b0;
192         mult_data_valid <= 1'b0;
193         ram_en <= 1'b0;
194
195         case (mult_fsm_curr_state)
196             MULT_HOLD: begin
197                 // no outputs
198             end
199
200             MULT_LOAD_FROM_LUTS_1a: begin
201                 mult_load_luts_1 <= 1'b1;
202             end
203
204             MULT_LOAD_FROM_LUTS_1b: begin
205                 mult_load_luts_1 <= 1'b1;
206             end
207
208             MULT_LOAD_FROM_LUTS_2a: begin
209                 mult_load_luts_2 <= 1'b1;
210             end
211
212             MULT_LOAD_FROM_LUTS_2b: begin
213                 mult_load_luts_2 <= 1'b1;
214             end
215
216             MULT_LUTS_SEND: begin
217                 m_valid_mult_a <= 1'b1;
218                 m_valid_mult_b <= 1'b1;
219             end
220
221             MULT_LUTS_RECV: begin
222                 s_ready_mult <= 1'b1;
223                 mult_save_ext_res <= 1'b1;
224             end
225
226             MULT_LOAD_IMG_DATA_1a: begin
227                 mult_load_img_data <= 1'b1;
228                 ram_en <= 1'b1;
229             end
230
231             MULT_LOAD_IMG_DATA_1b: begin
232                 mult_load_img_data <= 1'b1;
233                 ram_en <= 1'b1;
234             end
235
```

```
236             MULT_IMG_DATA_SEND: begin
237                 m_valid_mult_a <= 1'b1;
238                 m_valid_mult_b <= 1'b1;
239             end
240
241             MULT_IMG_DATA_RECV: begin
242                 s_ready_mult <= 1'b1;
243                 mult_save_ext_res <= 1'b1;
244             end
245
246             MULT_COUNT_STEP: begin
247                 mult_pixel_count_step <= 1'b1;
248                 mult_data_valid <= 1'b1;
249             end
250
251             default: begin
252                 // handled by initialization
253             end
254         endcase
255     end
256
257     // ------------------- datapath blocks ------------------- //
258
259     always_latch begin
260         if (mult_load_luts_1 == 1'b1) begin
261             // multiply image_row_port by 8
262             lut_addr <= (image_row_port << 3) + pixel_row_port;
263             operand_1 <= lut_dout; // load
264         end
265
266         if (mult_load_luts_2 == 1'b1) begin
267             // multiply image_col_port by 8
268             lut_addr <= (image_col_port << 3) + pixel_col_port;
269             operand_2 <= lut_dout; // load
270         end
271
272         // save AXI multiplier result
273         if (mult_save_ext_res == 1'b1 && s_valid_mult == 1'b1) begin
274             operand_1 <= s_data_mult;
275         end
276
277         if (mult_load_img_data == 1'b1) begin // overwrite LUT operand
278             operand_2 <= ram_dout_port;
279         end
280     end
281     endmodule
```

6.2.5 discrete_cosine_transform_tb.sv

```
1   module discrete_cosine_transform_tb();
2
3   reg clk_port = 1'b1;
4
5   reg t_valid_in = 1'b0;
6   wire t_ready_in;
7   reg [63:0] t_data_in = 64'b0;
8
9   wire t_valid_cos_bram;
10  reg t_ready_cos_bram = 1'b0;
```

```
11  reg [15:0] t_data_cos_bram = 16'b0;
12
13  wire t_valid_out;
14  reg t_ready_out = 1'b0;
15  wire [63:0] t_data_out;
16
17  discrete_cosine_transform uut (
18      .clk_port(clk_port),
19      .t_valid_in(t_valid_in),
20      .t_ready_in(t_ready_in),
21      .t_data_in(t_data_in),
22      .t_valid_cos_bram(t_valid_cos_bram),
23      .t_ready_cos_bram(t_ready_cos_bram),
24      .t_data_cos_bram(t_data_cos_bram),
25      .t_valid_out(t_valid_out),
26      .t_ready_out(t_ready_out),
27      .t_data_out(t_data_out)
28  );
29
30  initial begin
31      // no explicit signal changes are required
32  end
33
34  always begin
35      #10 clk_port = ~clk_port;
36  end
37  endmodule
```

6.2.6 dct_accumulate_operation_tb.sv

```
1   module dct_accumulate_operation_tb();
2   reg clk_port = 1'b1;
3   reg start_port = 1'b1;
4   reg [7:0] image_row_port = 8'h0; // this selects a row in the image
5   reg [7:0] image_col_port = 8'h0; // this selects a column in the image
6   wire [31:0] result_port;
7   wire data_valid_port;
8
9   wire [6:0] ram_addr_port;
10  wire [31:0] ram_din_port;
11  reg [31:0] ram_dout_port = 32'hc3000000; // -128
12  wire ram_en_port;
13  wire ram_we_port;
14
15  dct_accumulate_operation uut (
16      .clk_port(clk_port),
17      .start_port(start_port),
18      .ram_addr_port(ram_addr_port),
19      .ram_din_port(ram_din_port),
20      .ram_dout_port(ram_dout_port),
21      .ram_en_port(ram_en_port),
22      .ram_we_port(ram_we_port),
23      .image_row_port(image_row_port),
24      .image_col_port(image_col_port),
25      .result_port(result_port),
26      .data_valid_port(data_valid_port));
27
28  initial begin
29      // no explicit signal changes are required
```

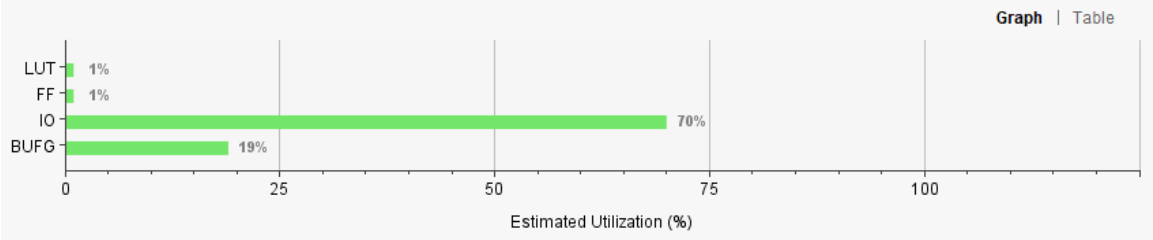
```
30 end
31
32 always begin
33     #10 clk_port = ~clk_port;
34 end
35 endmodule
```

6.2.7 pixel_stepper_tb.sv

```
1  module pixel_stepper_tb();
2
3  reg clk_port = 1'b1;
4  reg count_en = 1'b0;
5
6  wire [2:0] row_count;
7  wire [2:0] col_count;
8  wire tc;
9
10 pixel_stepper uut (
11     .clk_port(clk_port),
12     .count_en(count_en),
13     .row_count(row_count),
14     .col_count(col_count),
15     .tc(tc)
16 );
17
18 initial begin
19 end
20
21 // intermittently disable count_en line
22 always begin
23     #50 count_en <= 1'b1;
24     #10 count_en <= 1'b0;
25 end
26
27 always begin
28     #10 clk_port = ~clk_port;
29 end
30 endmodule
```

6.2.8 dct_multiply_operation_tb.sv

6.3 Resource Utilization



Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	157	17600	0.89
FF	63	35200	0.18
IO	70	100	70.00
BUFG	6	32	18.75

6.4 Residual Warning Analysis

When running synthesis on the DCT IP block, there were three categories of warnings that were encountered:

1. Design has unconnected port (6 warnings)
2. Inferring latch for variable (6 warnings)
3. No constraints selected for write (1 warning)

The six **design has unconnected port** errors are not harmful to the design as they simply warn that extra data bus lines within the design are not being used. This was an intentional design choice made within the image iterator block, since limiting the image row and column counts to six bits could potentially be a limiting factor in the future if we felt the need to expand the width of the image blocks being processed.

Normally **inferring latch** warnings are high severity, but in this case this is the intended behavior of the HDL. All inferred latch warnings within the design are contained in FSM datapaths, and are due to intentional latching behavior. An example of this behavior is shown in figure 16.

This top-level datapath, which loads the operands required to run the top-level block, latches the values of `operand_1`, `operand_2`, and `lut_addr` to save on two clock cycles for each `if` block. To remove this latching behavior would require adding a new FSM state that was switched to when the value of the AXI multiplier (in this case) was flagged as valid, which would take one cycle to register and one cycle to save the result.

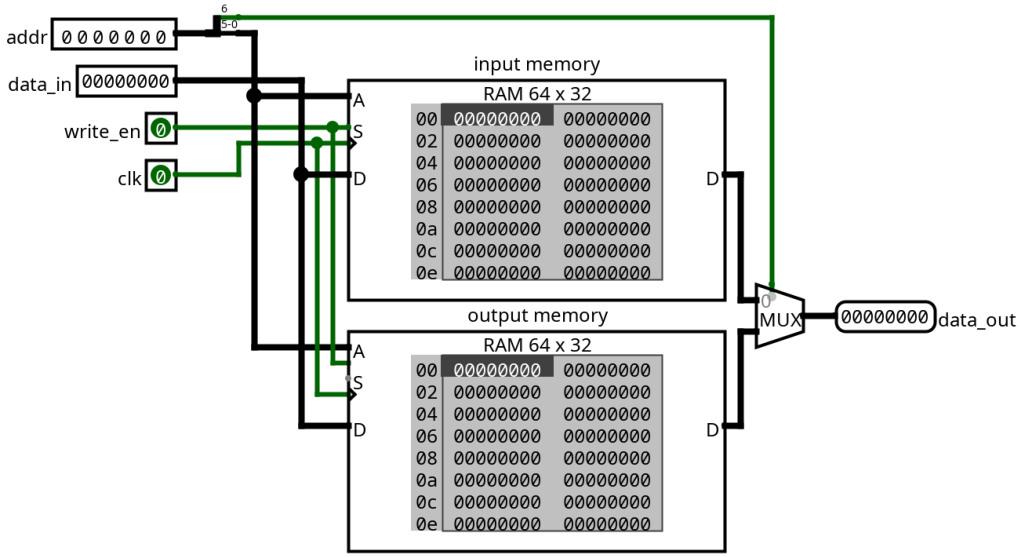
The final **no constraints selected** error is not an issue in this design since this design is intended for use as an IP block and is not intended to be written to a chip directly.

6.5 Memory Mapping

This design uses two block memory blocks: one BRAM block and one BROM block. The BROM block is a 64×32-bit ROM storing single-precision floating-point numbers. The BRAM is a tru dual-port 128×32-bit block split into two separate partitions, shown below as *input memory* and *output memory*. Port A of the BRAM is used by the top-level shell, and port B of the BRAM is used by the multiplication operation module.

```
1  always_latch begin
2      // multiply image_row_port by 8
3      if (mult_load_luts_1 == 1'b1) begin
4          lut_addr <= (image_row_port << 3) + pixel_row_port;
5          operand_1 <= lut_dout; // load
6      end
7
8      // multiply image_col_port by 8
9      if (mult_load_luts_2 == 1'b1) begin
10         lut_addr <= (image_col_port << 3) + pixel_col_port;
11         operand_2 <= lut_dout; // load
12     end
13
14     // save AXI multiplier result
15     if (mult_save_ext_res == 1'b1 && s_valid_mult == 1'b1) begin
16         operand_1 <= s_data_mult;
17     end
18
19     // overwrite LUT operand
20     if (mult_load_img_data == 1'b1) begin
21         operand_2 <= ram_dout_port;
22     end
23 end
```

Figure 16: SystemVerilog showing explicit, intentional latch behavior within the top-level datapath



The *input memory* partition is used by the top-level shell to store memory from the incoming AXI Stream. This partition is only ever written to by the top-level shell, and only ever read from by the multiplication operation module. The data from the *input memory* partition is then transmitted upwards in the design until the resulting output from all required operations is then written into

the *output memory* partition by the top-level shell. This approach is taken because it cannot be guaranteed that there will not be any race conditions within a single-partition memory setup. The *output memory* partition is then read from by the output AXI Stream driver.

6.6 Annotated Simulation Waveforms

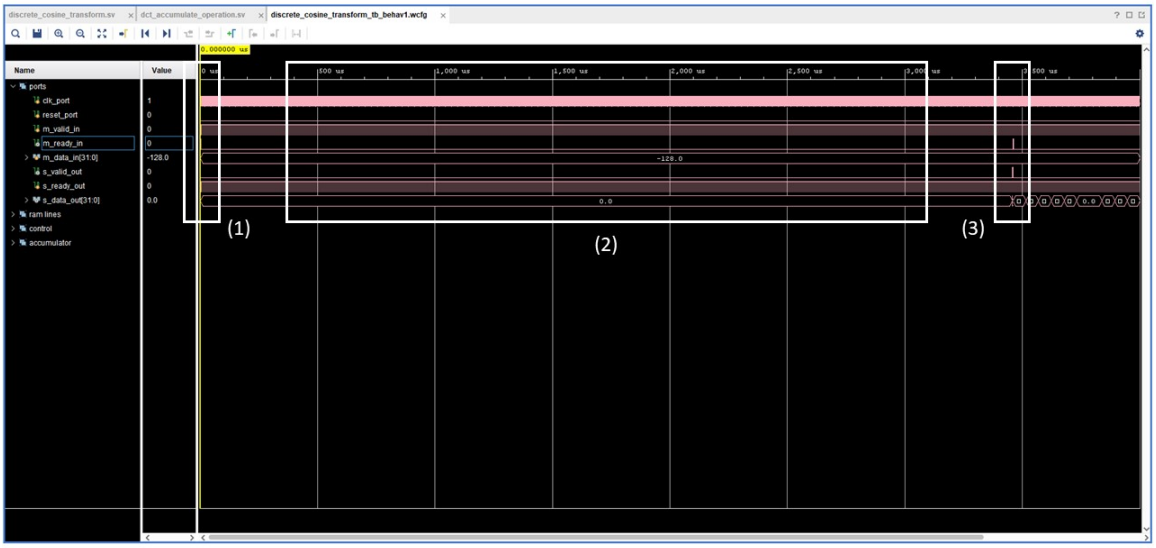


Figure 17: A simulated waveform diagram showing the processing of a single 8×8 image. Data is received at **1** via the input AXI Stream port, the data is is during **2**, and data is transferred out at **3** via the output AXI Stream port.

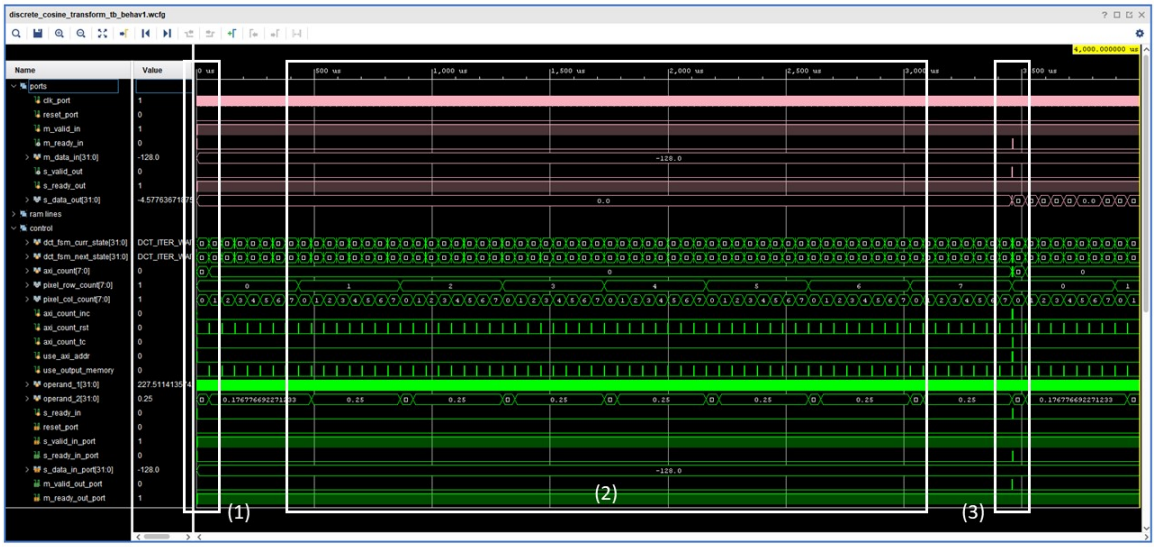


Figure 18: A simulated waveform diagram showing a more detailed view into the processing of a single 8×8 image. Data is received at **1**, processed during **2**, and transferred out at **3** as in figure 6.6. Shown lower in the waveform are the *pixel_row_count* and *pixel_col_count* lines. These lines show the pixel for which the DCT is currently being calculated. As discussed in section 2.1, pixels are indexed in the form $(\text{pixel\_row\_count} \times 8) + \text{pixel\_col\_count}$.

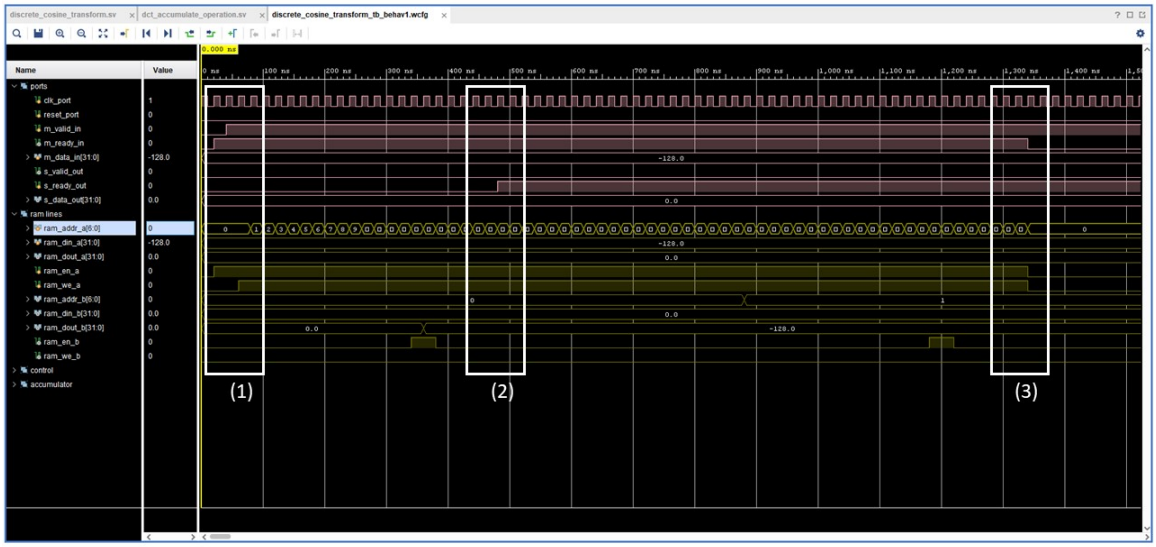


Figure 19: A simulated waveform diagram showing a detailed view of the AXI Stream data reception. As shown in **1**, the simulated receiver asserts the *m_ready_in* line, and once the block asserts the *m_valid_line*, a single-precision floating point number is transmitted once per clock cycle while both lines are asserted. Note that the *ram_en_a* and *ram_we_a* lines are asserted during the data transfer. At **2** the receiver indicates it is ready to receive data from the block once it has finished processing the image. At **3**, the block deasserts the AXI ready line, meaning it is no longer ready to receive data. The block then begins processing the received data.

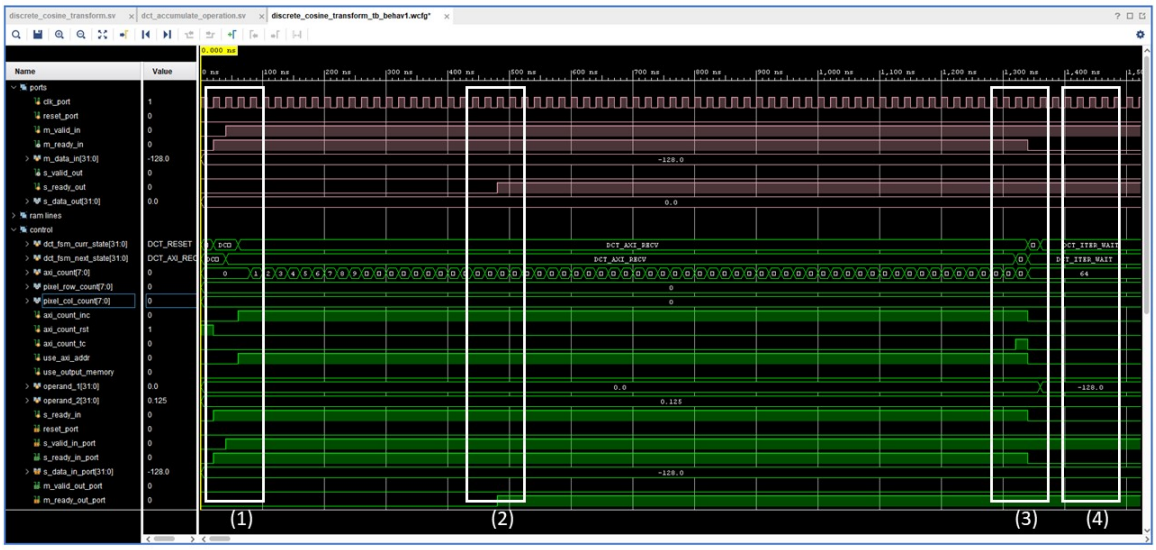


Figure 20: A simulated waveform diagram showing additional signals during the same reception process as in figure 6.6. During the initialization of the transfer at **1**, the block shifts into the *DCT_AXI_RECV* state during which it will assert the *m_ready_in* line and will load data into the BRAM block. The block asserts the *use_axi_addr* line, which will use the count of the number of transmissions to index the data into the BRAM block. The asserted *axi_count_inc* line will increment this count each time a valid float is received.

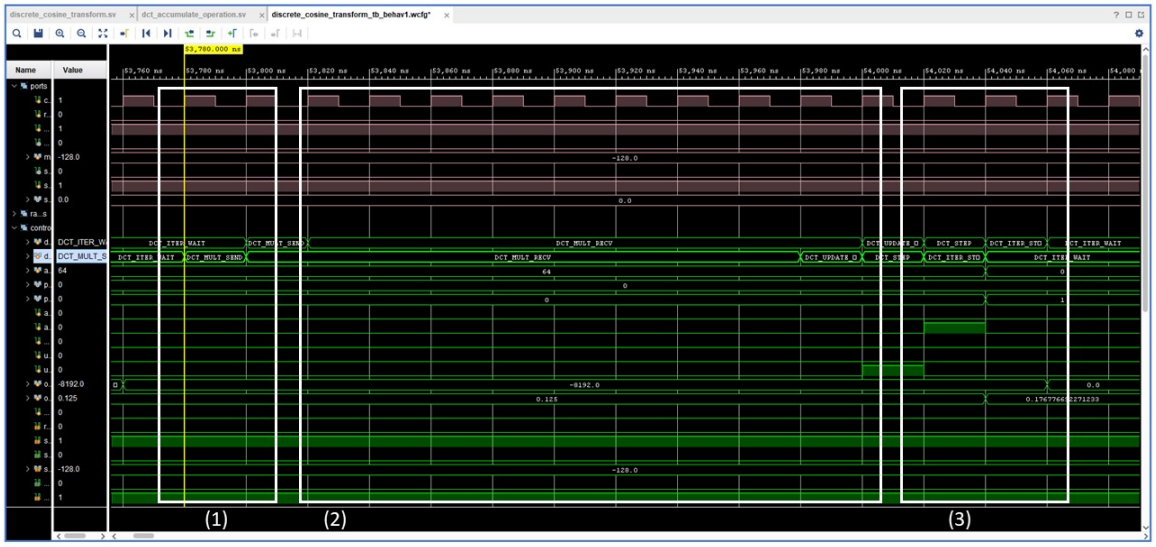


Figure 21: A simulated waveform diagram showing the processing of a single pixel of the received image. At **1** the FSM enables the multiplier module, then waits to receive data from the module in **2**. Once the data is received, the FSM stores the data then repeats the process in **3**.

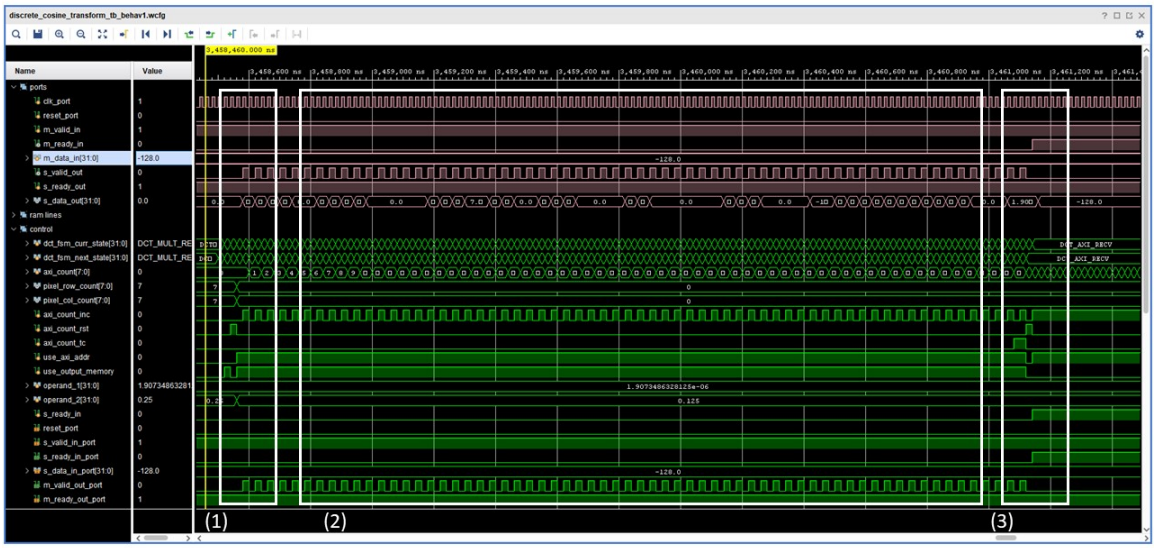


Figure 22: A simulated waveform diagram showing the complete transmission process of the processed data from the block over the AXI Stream output port. Note that the simulated receiver has asserted the *s_ready_out* line, meaning it is ready to receive data. The block asserts the *s_valid_out* line each time it is ready to transmit data in **2**, and does this until the *axi_count* line indicates it has reached its terminal count (*axi_count_tc* is asserted). The block then deasserts the *s_valid_out* line and resets its internal transmission count state (*axi_count_rst*), ending the transmission in **3**.

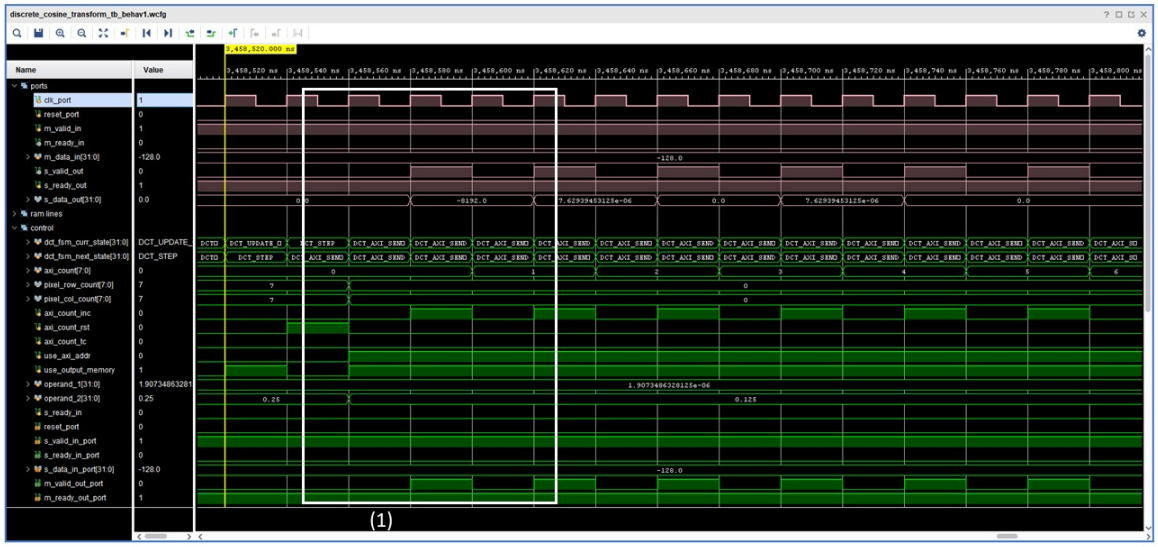


Figure 23: A simulated waveform diagram showing that the data is valid coming out of the DCT block. The expected results from running the DCT on a completely black image (input: -128 for all pixels) should yield -8192 for $G_{0,0}$ and 0 for all other u, v. This diagram shows the expected value -8192 being transmitted at **1**, where all other values transmitted round to 0 (not perfectly 0 due to inaccuracies in the division blocks).

6.7 Control State Machines

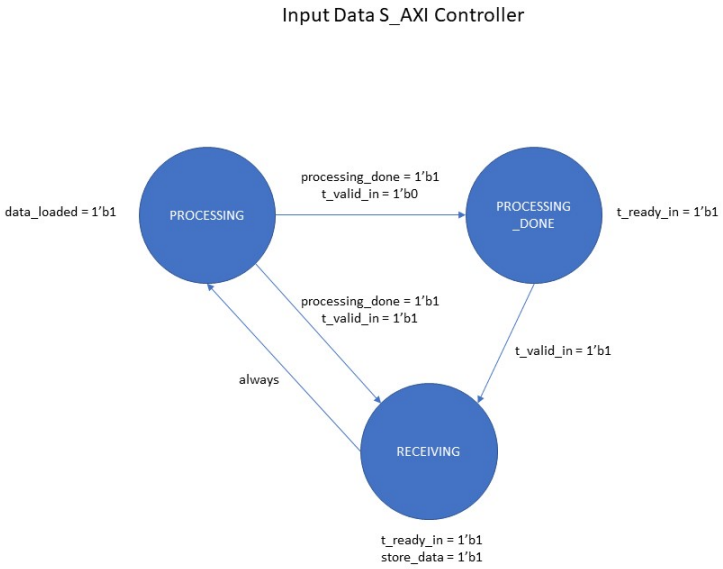


Figure 24: FSM state diagram for a generic AXI Stream receiver.

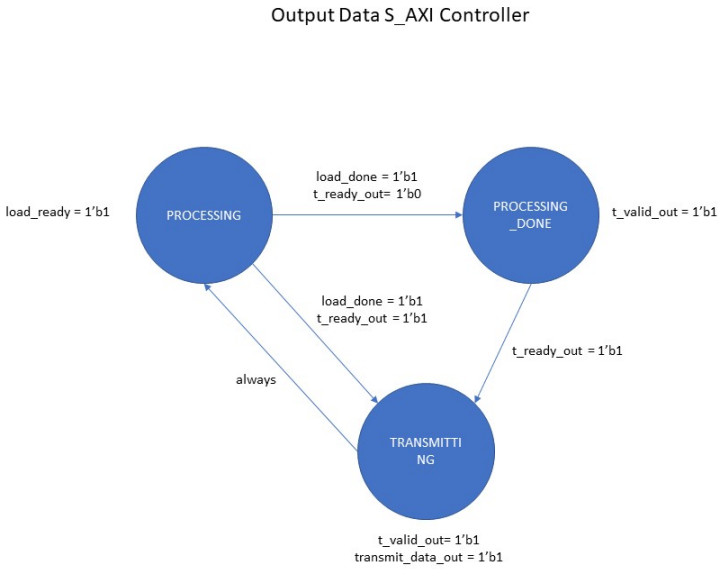


Figure 25: FSM state diagram for a generic AXI Stream transmitter.

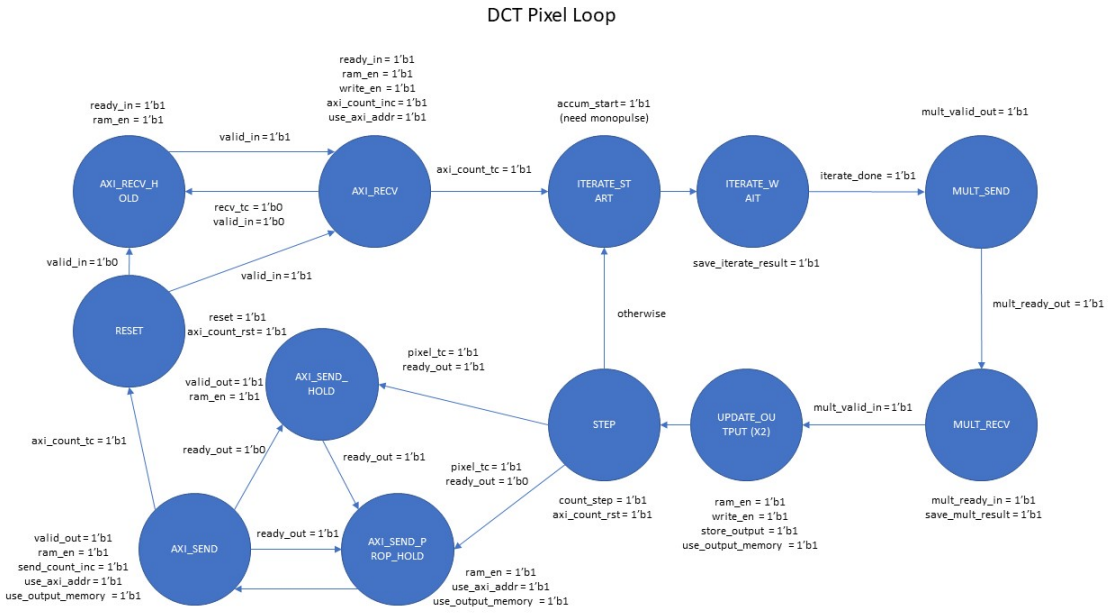


Figure 26: FSM state diagram for the top-level module of the DCT IP block.

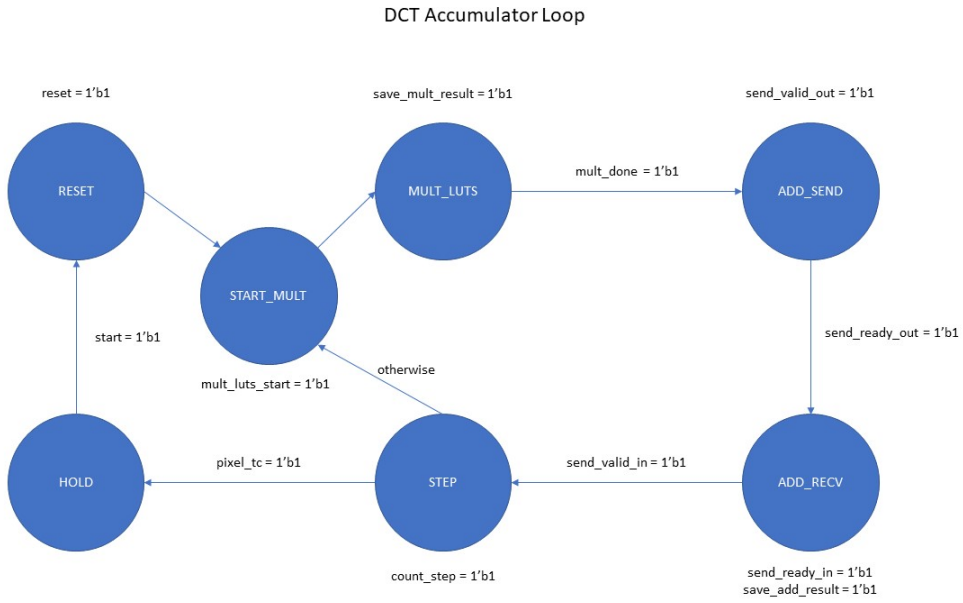


Figure 27: FSM state diagram for the DCT accumulator which runs the DCT for each pixel in the image.

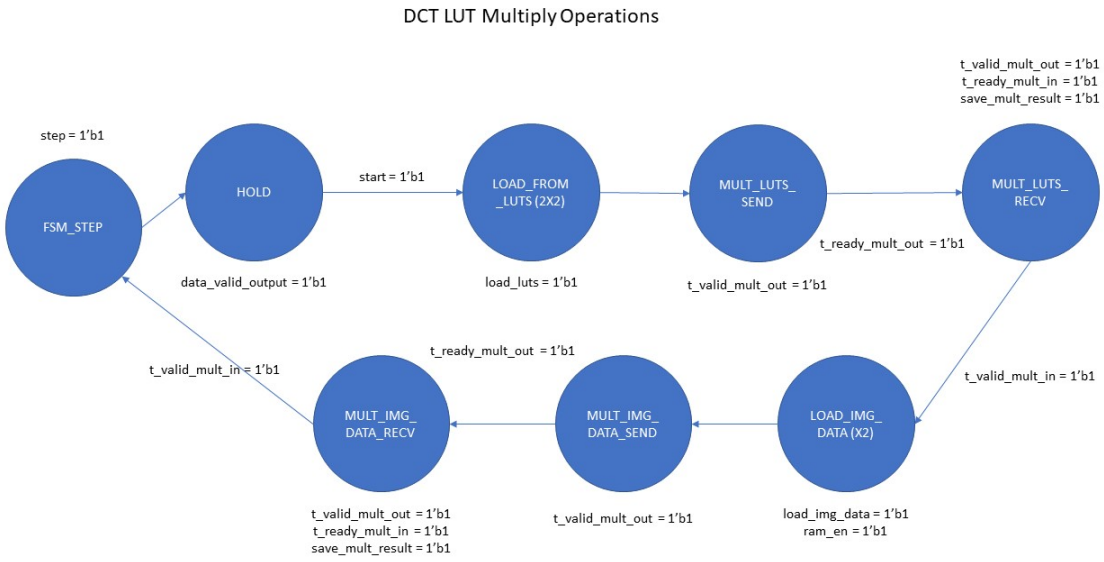


Figure 28: FSM state diagram for the multiplication module of the DCT design.

6.8 Additional Scripts

This section contains scripts that were utilized during the development process to validate the working design. Some scripts have had non-vital functionality removed to simplify the attached code.

6.8.1 DCT LUT COE Generator (MATLAB)

```
%% Constants Declaration

IMAGE_DIMENSION = 8;
OUTFILE_LOCATION = "ENTER_OUTPUT_FILE_LOCATION_HERE";

I = [
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
    -128,-128,-128,-128,-128,-128,-128,-128;
];

%% Calculate LUT Values

LUT = zeros(IMAGE_DIMENSION,IMAGE_DIMENSION);

for x = 1:IMAGE_DIMENSION
    for y = 1:IMAGE_DIMENSION
        LUT(x,y) = cos((2*(y-1)+1)*(x-1)*pi/(2*IMAGE_DIMENSION));
    end
end

%% Output LUT as Floating-Point COE file

out_file = fopen(OUTFILE_LOCATION, "w");

fprintf(out_file, "MEMORY_INITIALIZATION_RADIX=16;\n");
fprintf(out_file, "MEMORY_INITIALIZATION_VECTOR=\n");

for outer_it = 1:IMAGE_DIMENSION
    for inner_it = 1:IMAGE_DIMENSION
        fixed_point = num2hex(single(LUT(outer_it,inner_it)));
        fprintf(out_file, "%s,\n", fixed_point);
    end
end

fclose(out_file);
```

Figure 29: A MATLAB script to generate the contents of the DCT cosine lookup (see section 2.2). When provided a location to export a file, this script will calculate the required values of the LUT used in the DCT top-level block and export them into a coefficients (.coe) file. This file can then be used to initialize a block ROM (BROM) block within a Vivado project.

6.8.2 Image Resize Script (MATLAB)

```
%% Constants Declaration

IMAGE_LOCATION = "ENTER_IMAGE_DISK_LOCATION_HERE";
IMAGE_SIZE = 8;

OUTFILE_LOCATION = "ENTER_OUTPUT_FILE_LOCATION_HERE";

%% Load and Crop Image

image_data = imread(IMAGE_LOCATION);
image_data_downscaled = imresize(image_data, [IMAGE_SIZE NaN]);

target_size = [IMAGE_SIZE IMAGE_SIZE];
crop_rectangle = centerCropWindow2d(size(image_data_downscaled), target_size);
image_data_cropped = imcrop(image_data_downscaled, crop_rectangle);

%% Display Processed Image

[R,G,B] = imsplit(image_data_cropped);

%% Parse and Load into COE File

CR_B = cellstr(dec2bin(R));
out_file = fopen(OUTFILE_LOCATION, "w");

fprintf(out_file, "MEMORY_INITIALIZATION_RADIX=2;\n");
fprintf(out_file, "MEMORY_INITIALIZATION_VECTOR=\n");

fprintf(out_file, "%s,\n", CR_B{:});
fclose(out_file);
```

Figure 30: A MATLAB script to load an image for disk and split it into RGB channels and storing an arbitrary channel (in this case the red channel) in a coefficients (.coe) file. This script was used during testing to produce testing outputs for the top-level DCT block. An image known to produce good results with this script can be found here, although any image will work. Note that the typical JPEG encoding process requires the conversion of the image from RGB to YCbCr. This conversion is not necessary for testing, since all we need to validate the DCT functionality is a 64-wide block of memory containing single-precision floating-point numbers.