26/10/2023

CSSE4010 Project

Hardware Implementations of 2D DCT

David Gaul
UNIVERSITY OF QUEENSLAND

Introduction

The purpose of this project was to explore the hardware implementation of a 2D Discrete Cosines Transform (DCT). DCTs have been employed as a major form of image compression due to their ability to vastly reduce size complexity of data whilst retaining image quality (Uma Sadhvi Potluri, 2014). Within this project, several methods of DCT implementation will be tested. A 'perfect' software implementation will be tested and used as a benchmark model with which to compare the hardware implementations to.

To begin with, a 1D DCT will be developed using only 14 adder blocks. This is based off the sparse matrix DCT transformations proposed by Potluri et al (Uma Sadhvi Potluri, 2014). Scaling of this 1D DCT architecture to accommodate for 2D image compression will be performed using the method described by Wijenayake et al (Kulasekera, et al., 2015). Finally, the full adder system will be replaced using approximate adders in line with the works of Prabakaran et al (Bharath Srinivas Prabakaran, 2018). These will be compared against a baseline software implementation of DCT filters.

This project is being undertaken with the intention of providing an optimised model in terms of runtime and FPGA resource consumption. For this reason, multiple optimisation steps including pipelining, bit length selection and approximate adder selection will be undertaken. In doing so, the final model will be able to minimise critical path delay as well as resource consumption. In reducing the latter, this will facilitate the capacity to accommodate for parallelised DCT filtering. This is, however, outside the scope of this project and will only be mentioned in passing.

Software Implementation of the 2D DCT.

As previously mentioned, a software implementation of the DCT will serve as the benchmark model for analysis in this report. This particular implementation involves performs DCT compression over all 8*8 grids within an image using the transformational matrix T_m. T_m is defined traditionally defined as the cosine transform:

$$\mathbb{C}_{m,n} = k_n \cos\left[\frac{(2m+1)n\pi}{2N}\right] \text{where } k_n = \begin{cases} \sqrt{1/N} & \text{if } n = 0 \\ \sqrt{2/N} & \text{if otherwise} \end{cases}$$

Which, when translated to an 8*8 matrix becomes:

0.3536	0.3536	0.3536	0.3536	0.3536	0.3536	0.3536	0.3536
0.4904	0.4157	0.2778	0.0975	-0.0975	-0.2778	-0.4157	-0.4904
0.4619	0.1913	-0.1913	-0.4619	-0.4619	-0.1913	0.1913	0.4619
0.4157	-0.0975	-0.4904	-0.2778	0.2778	0.4904	0.0975	-0.4157
0.3536	-0.3536	-0.3536	0.3536	0.3536	-0.3536	-0.3536	0.3536
0.2778	-0.4904	0.0975	0.4157	-0.4157	-0.0975	0.4904	-0.2778
0.1913	-0.4619	0.4619	-0.1913	-0.1913	0.4619	-0.4619	0.1913
0.0975	-0.2778	0.4157	-0.4904	0.4904	-0.4157	0.2778	-0.0975

The Matlab implementation for this is given as:

```
%2-D DCT operation
T_m = dctmtx(8);
dct = @(block_struct) T_m * block_struct.data * T_m';    % DCT function
I_dct = blockproc(I_org,[8 8],dct); %DCT Operation for each segment
```

I_org represents the picture being used within this study, Lena.jpg, which is controversial enough to ban this report from all major scientific journals.

Not that this report will be good enough quality for that, anyway.

Following successful implementation of the DCT, a mask is used to retain only the most valuable coefficients from the 2D transform. The masks used within this study retain the top 10, 20 and 64 coefficients respectively.

```
%Thersholding
I_th = blockproc(I_dct,[8 8],@(block_struct) mask .* block_struct.data);  % Thersholding for each segment
```

To revert the compressed image to its previous form, an inverse DCT can be applied. This takes advantage of the orthogonality of matrix T_m to remove its effects from the original image.

```
%Inverse DCT
invdct = @(block_struct) T_m' * block_struct.data * T_m;  %Inverse DCT function
I_recon = blockproc(I_th,[8 8],invdct); %Inverse DCT Operation for each segment
```

Figure 1 shows a comparison between the reconstructed images using DCT co-efficients r=10, r=20 and r=64 to the original image.

Figure 1: Comparison between original image (left) compared to filter co-efficients r=10 (Second), r=20 (third) and r=64 (right)

Asides from the traditional DCT method employed above, Potluri et al (Uma Sadhvi Potluri, 2014) allude to an Approximate Discrete Cosine Transform (ADCT) which uses a simplified transformational

matrix T_m. This matrix is intended to save on complexity and runtime at the expense of an acceptable loss in accuracy. The T_m matrix employed through this method is given as:

$$\mathbf{C}^* = \mathbf{D}^* \cdot \mathbf{T}^*$$

$$= \mathbf{D}^* \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

where $\mathbf{D}^* = \mathrm{diag}([1/\sqrt{8}, 1/\sqrt{2}, 1/2, 1/\sqrt{2}, 1/\sqrt{8}, 1/\sqrt{2}, 1/2, 1/\sqrt{2})$.

Employing the ADCT on the original image returns similar results to those found in the traditional DCT. These can be observed in Figure 2.

Figure 2: Comparison of reconstructed ADCT images for r=10, r=20 and r=64

Along-side visual performance of the image compression capabilities, it is necessary to provide numerical measurements. The performance metrics used within this report were the reconstructed image's Mean Squared Error (MSE) and Peak Signal to Noise Ratio (PSNR).

$$MSE = \frac{1}{m\,n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2.$$

$$\begin{aligned} PSNR &= 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \\ &= 20 \cdot \log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE). \end{aligned}$$

Applying these equations to the ADCT and DCT transforms for r=10 and r=20 returns the error metrics found in Table 1.

Table 1: Performance comparison of software DCT and ADCT

Co-efficients (r)	Transform	MSE	PSNR
10	ADCT	263.150	23.928
	DCT	149.169	26.393
20	ADCT	188.447	25.379

	DCT	90.891	28.545

The performances found in Table 1 indicate that there is a substantial loss in accuracy present within the ADCT. However, visual inspection of the images shows that this is an acceptable level of error, as the pictures still retain their original shape. The rest of this report will focus on recreating the ADCT using hardware implementations for use with FPGA design boards.

Hardware Implementation

The hardware implementation of the ADCT matrix proposed by Potluri et al (Uma Sadhvi Potluri, 2014) can be found within section 6, figure 7a. This recreates the sparse matrix using only 14 adder and subtractor blocks. The block diagram for this image can be found in Figure 3.

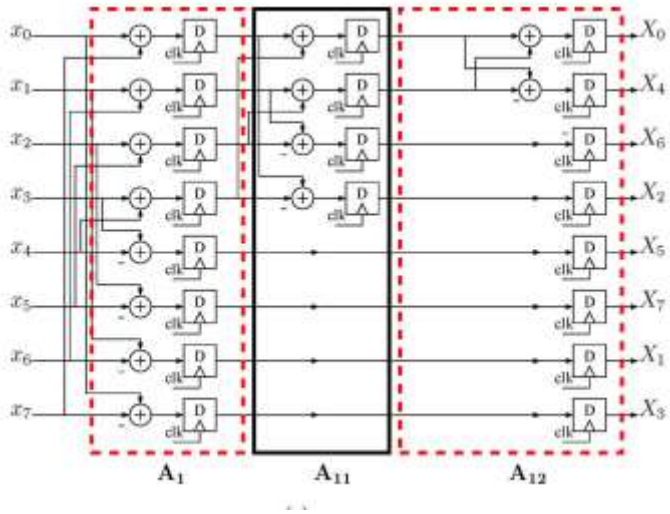

Figure 3: 1D ADCT hardware implementation proposed by Potluri et al (Uma Sadhvi Potluri, 2014).

Within the above figure, the delay blocks represent a register within which the intermediary calculations are stored. Within Model Composer, these can be replaced with a delay block, or by increasing the latency of the adder blocks by 1. Of additional note is the inversing function applied to output X6. The full model composer implementation of this system can be found in Figure 4.

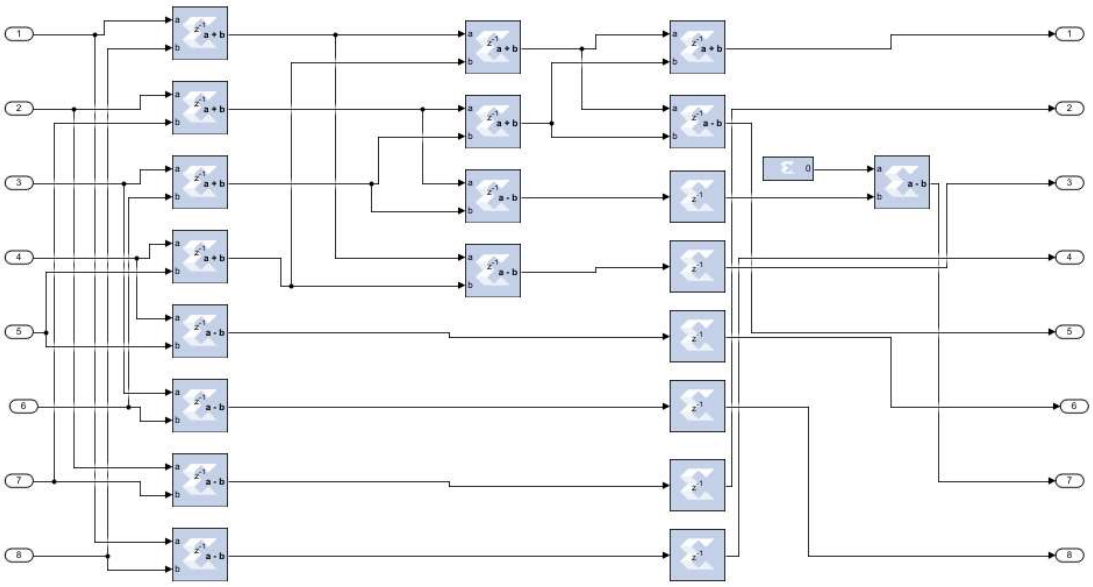

Figure 4: Model Composer implementation of the 1D ADCT

From here, the design can be scaled up to a 2D version. This was achieved by employing the architecture proposed by Wijenayake et al (Kulasekera, et al., 2015), which can be found in Figure 5

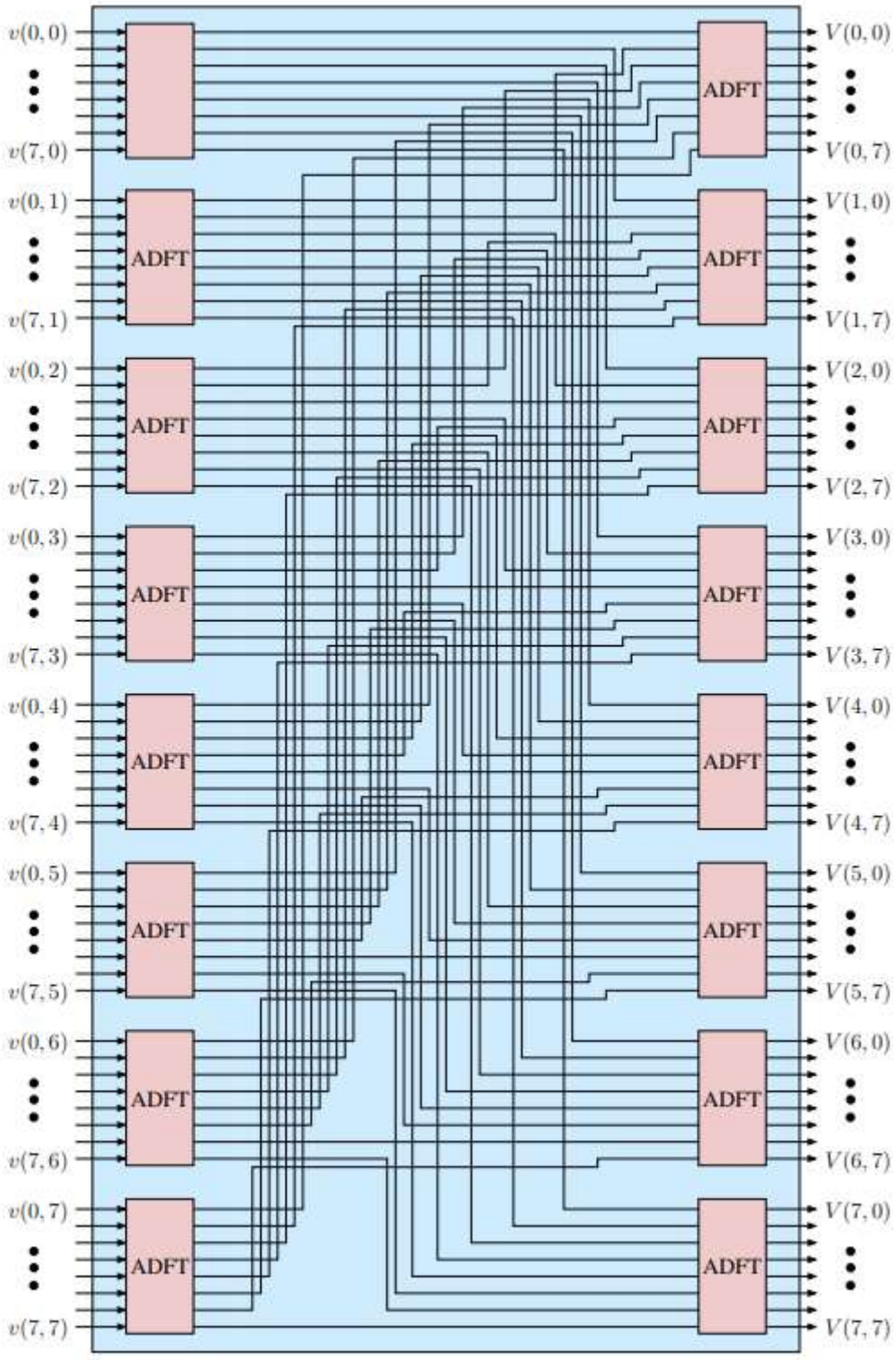

Figure 5: 2D ADFT architecture proposed by Wijenayake et al (Kulasekera, et al., 2015).

The Model Composer version of this can be found in Figure 6

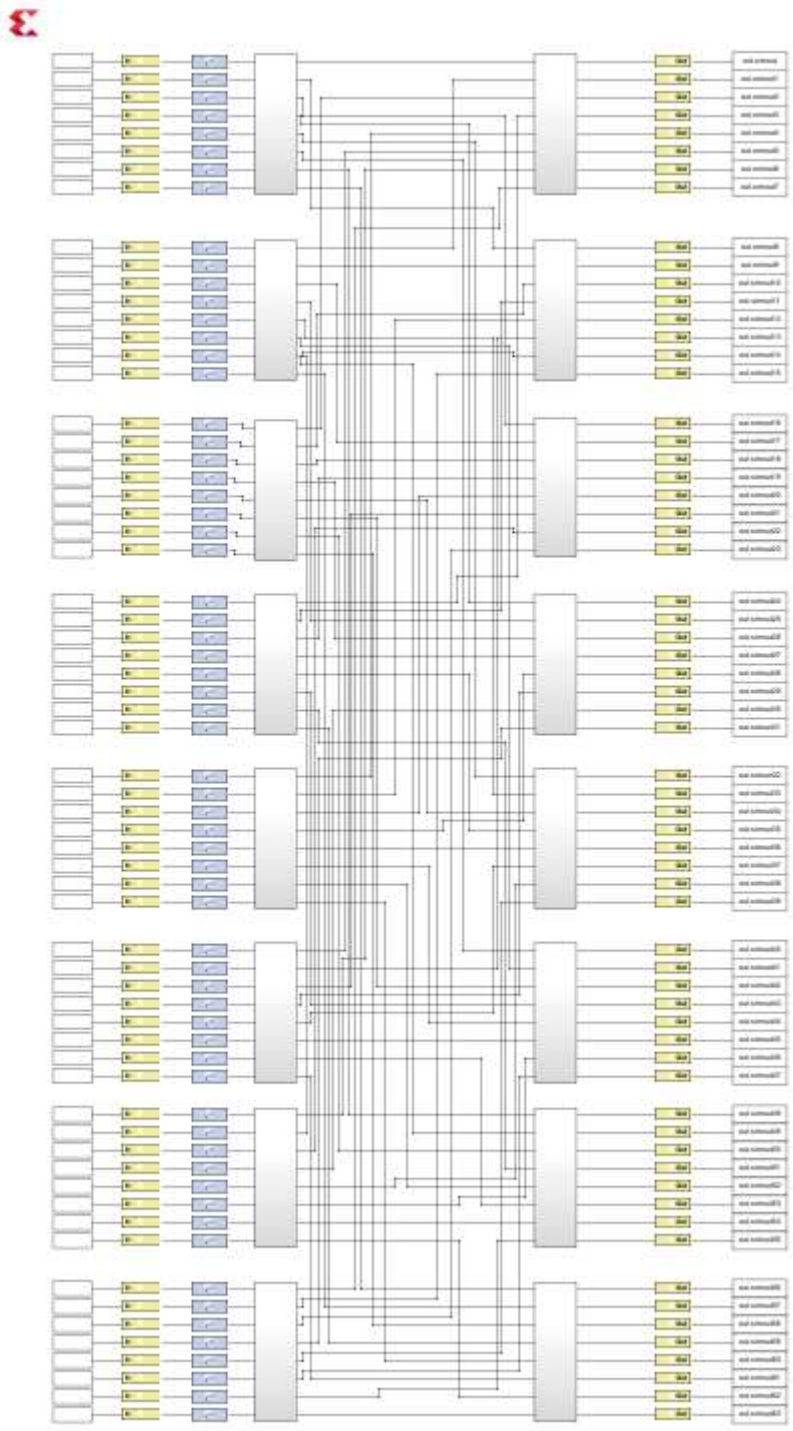

Figure 6: Model Composer implementation of the 2D ADCT architecture

Although the design was unoptimized in terms of throughput, it could still be used to provide accurate image reconstruction. The performance metrics of this design could be found in Table 2.

Table 2: Performance metrics for the untuned hardware implementation

r	MSE	PSNR
10	263.150	23.928
20	188.446	25.378

These results are directly comparable to those achieved by the software implementations of the ADCT. However, the lack of pipelining within this design meant that it performed far slower than desirable. This was due to the lack of registers within which to store intermediary values from inputs 5-8. However, this did not affect the critical path delay of the entire design, which can be found in figure …….

	Slack (ns)	Delay (ns)	Logic Delay (ns)	uting Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Cloc	Path Constraints
1	5.492	4.465	1.708	2.757	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
2	5.532	4.423	1.601	2.822	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
3	5.534	4.421	1.63	2.791	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
4	5.542	4.457	1.708	2.749	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
5	5.647	4.31	1.653	2.657	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
6	5.673	4.285	1.659	2.626	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
7	5.737	4.221	1.256	2.965	4	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
8	5.767	4.19	1.709	2.481	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
9	5.781	4.175	1.637	2.538	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
10	5.788	4.17	1.646	2.524	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
11	5.792	4.163	1.598	2.565	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
12	5.809	4.191	1.601	2.59	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
13	5.819	4.138	1.601	2.537	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
14	5.841	4.16	1.55	2.61	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
15	5.843	4.112	1.598	2.514	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
16	5.883	4.074	1.712	2.362	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
17	5.888	4.111	1.564	2.547	4	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
18	5.89	4.111	1.625	2.486	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
19	5.912	4.085	1.609	2.476	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
20	5.913	4.042	1.55	2.492	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
21	5.914	4.044	1.635	2.409	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
22	5.919	4.08	1.564	2.516	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
23	5.933	4.066	1.694	2.372	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
24	5.933	4.065	1.664	2.401	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
25	5.934	4.023	1.76	2.263	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
26	5.935	4.021	1.677	2.344	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
27	5.946	4.055	1.638	2.417	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
28	5.972	4.029	1.58	2.449	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
29	5.979	3.976	1.761	2.215	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]

Figure 7: Timing analysis of the unoptimised hardware ADCT

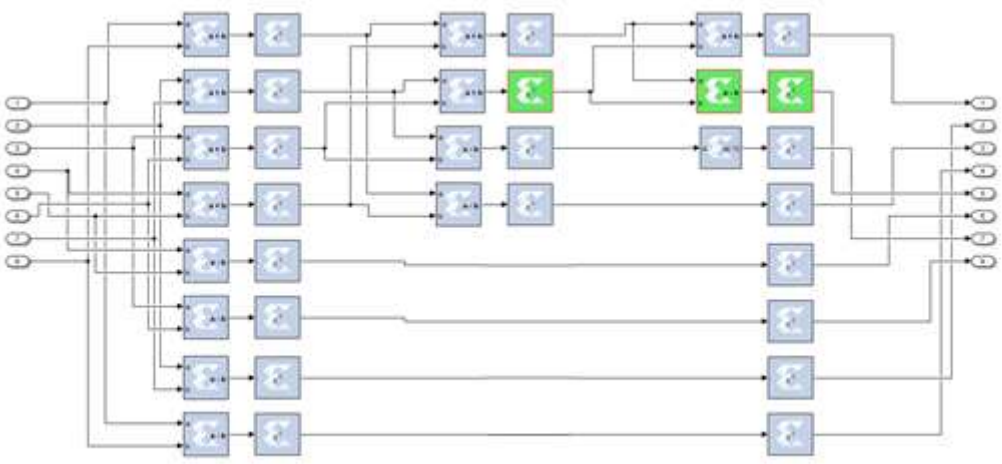

Figure 8: Critical path of the unoptimised hardware ADCT

Although the critical path delay was found to be 4.465ns, the rate of throughput for this architecture is significantly less than that. Due to the aforementioned lack of registers over inputs 5-8, each DCT

is only able to generate outputs every 2 clock cycles. This attributes to a throughput of around 100MHz. To fix this, pipelining was applied to the 1D ADCT transform within each of the subsystems of the 2D architecture. As mentioned, by providing register within which to store intermediary values for inputs 5-8, the throughput of the system could be doubled. Model Composer automatically integrates register allocation via the use of delay blocks. The optimised model can be found within Figure 9.

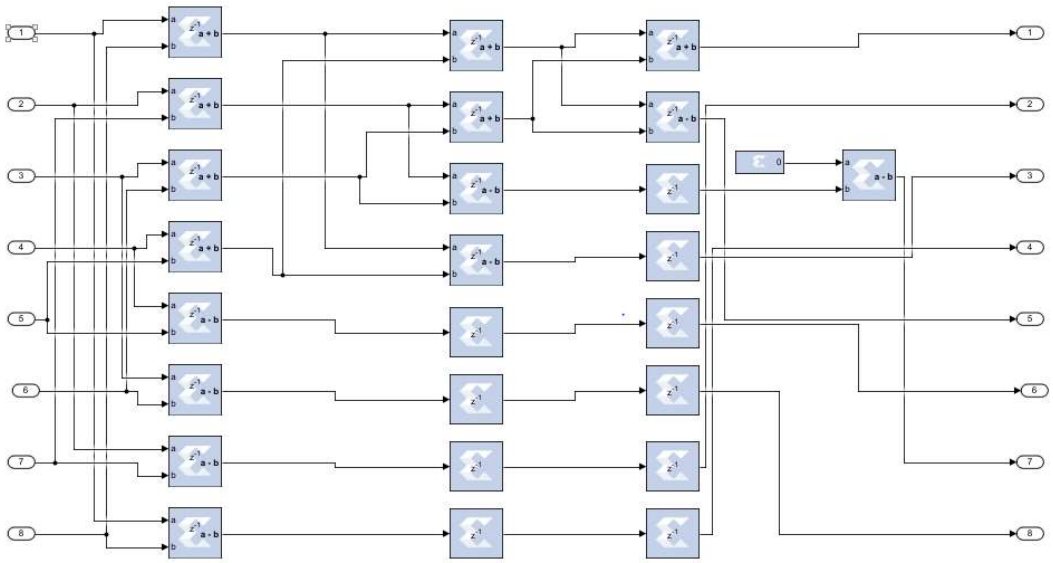

Figure 9: Pipelined version of the 1D ADCT

Although this served to slightly increase the area complexity of the design, it halved the processing time for any input image. This is a desirable and strongly recommended decision. Figure 10 shows the updated timing analysis whereas Figure 11 shows the new critical path.

	Slack (ns)	Delay (ns)	Logic Delay (ns)	uting Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Cloc	Path Constraints
1	5.492	4.465	1.708	2.757	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
2	5.532	4.423	1.601	2.822	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
3	5.534	4.421	1.63	2.791	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
4	5.542	4.457	1.708	2.749	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
5	5.647	4.31	1.653	2.657	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
6	5.673	4.285	1.659	2.626	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
7	5.737	4.221	1.256	2.965	4	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
8	5.767	4.19	1.709	2.481	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
9	5.781	4.175	1.637	2.538	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
10	5.788	4.17	1.646	2.524	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
11	5.792	4.163	1.598	2.565	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
12	5.809	4.191	1.601	2.59	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
13	5.819	4.138	1.601	2.537	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
14	5.841	4.16	1.55	2.61	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
15	5.843	4.112	1.598	2.514	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
16	5.883	4.074	1.712	2.362	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
17	5.888	4.111	1.564	2.547	4	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
18	5.89	4.111	1.625	2.486	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
19	5.912	4.085	1.609	2.476	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
20	5.913	4.042	1.55	2.492	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
21	5.914	4.044	1.635	2.409	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
22	5.919	4.08	1.564	2.516	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
23	5.933	4.066	1.694	2.372	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
24	5.933	4.065	1.664	2.401	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
25	5.934	4.023	1.76	2.263	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
26	5.935	4.021	1.677	2.344	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
27	5.946	4.055	1.638	2.417	6	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
28	5.972	4.029	1.58	2.449	5	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]
29	5.979	3.976	1.761	2.215	7	tuned_filt…	tuned_filt…	clk	clk	create_clock -name clk -period 10 [get_ports clk]

Figure 10: Timing analysis for the optimised 1D ADCT

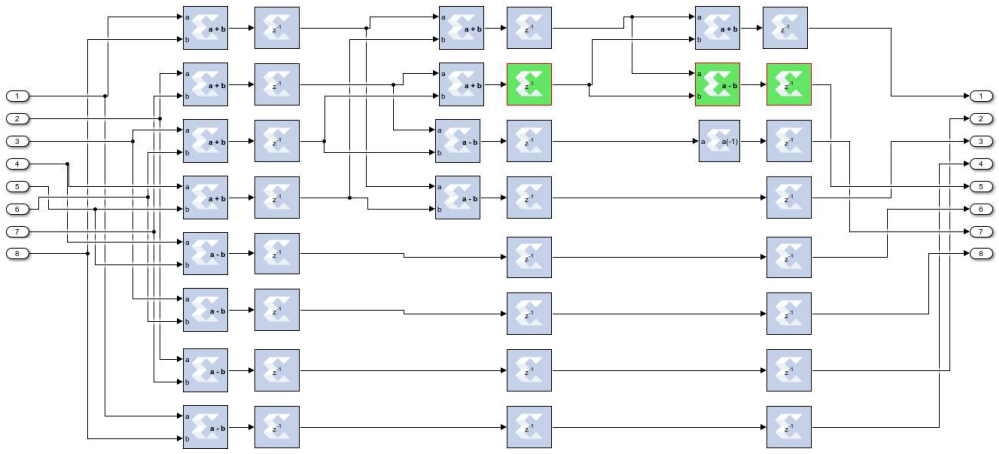

Figure 11: Critical path for the optimised 1D ADCT

The above figures indicate that there's been no change in critical path or critical path delay. However, due to the introduction of pipelining, the throughput of the entire system has doubled. The critical path of 4.465ns, therefore, corresponds with a throughput of around 200MHz.

Following successful pipelining, the model could be further refined via bit length selection. Due to the nature of the images comprising of integer values between 0 and 255, it was known that all adding and subtracting functions undertaken through the ADCT would return integer values. For this reason, decimal point precision was not necessary for model accuracy. Furthermore, the maximum

valued output could be calculated from the case where all 64 input values are pure white (255). The highest valued output under these circumstances could be calculated as:

$$X_0 = 8 \times 8 \times 255 \approx 8 \times 8 \times 256$$

$$X_0 \approx 2^3 \times 2^3 \times 2^8$$

$$X_0 \approx 2^{14}$$

Where the exact maximum value is found to be

$$X_0 = 16320$$

Which can be expressed using 14 bits. Including the signed bit for 2's compliment, this would indicate that a total of 15 bits would be needed for absolute accuracy. This was verified by testing the returned MSE using bit lengths 1-16, which can be found in Figure 12.

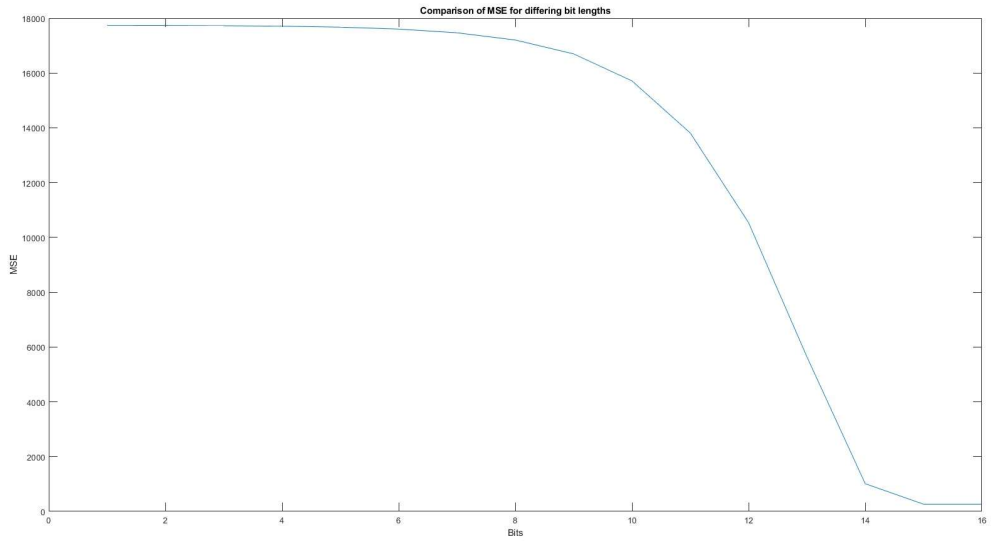

Figure 12: Comparison of MSE against bit length

This indicates that 15 bits was needed to obtain optimum accuracy within the design, exactly as predicted. Post-pipelining and bit length optimisation, the area and runtime complexity of the system have been fully refined, with no loss in model compression performance. This is reflected in Table 3, which shows a comparison of performance metrics between different designs using cosine co-efficients r=10.

Table 3: Comparison of different DCT architectures

Metric	Implementation		
	Software DCT	Software ADCT	Hardware ADCT
MSE	149.168	263.150	263.150
PSNR	26.393	23.928	23.928

Figure 13: Comparison of reconstructed images of hardware ADCT for r=10 and r=20

These reconstructed images come out so cleanly that you can see her eyeliner. It's almost as perfect as the eyeliner from one of the tutors in this course.

The performance metrics are identical to the software implementation of the ADCT, indicating that it is performing exactly as desired. Finally, Figure 14 shows the resource consumption of the final design.

Resource	Utilization	Available	Utilization %
LUT	2176	63400	3.43
FF	7168	126800	5.65
IO	2049	210	975.71

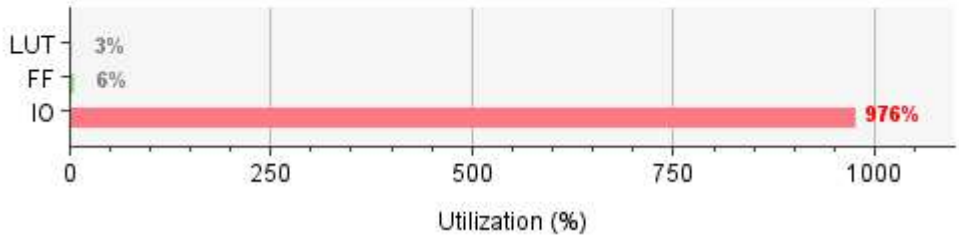

Figure 14: Resource utilisation of hardware ADCT

As can be seen, 3% of the total look-up tables and 6% of the slice registers have been used by the synthesized design. However, 2049 bonded IOB have been used within the synthesized design: 10x the 210 ports available on the Nexys 4 board. In order to parallelise the design, the bonded IOB consumption would have to be reduced. This can be conducted in the following ways:

- Changing bus to serial inputs
- Time multiplexing the input and output buses
- Altering the hardware implementation to rely less on input-output systems

Ignoring the constraints on the bonded IOB, a relatively small consumption has been attained on the total area of look-up tables and slice registers. A fully parallelised design could potentially run up to 16 2D DCT implementations synchronously. This would be achieved by increasing the number of 8*8 matrices being acted upon within the implementation. By creating a Model Composer file containing 16 2D DCT architectures with unique mappings, image compression could potentially be sped up to 16x its current processing speed.

Optimisation of this form is, however, beyond the scope of this project. Furthermore, optimisation of the bonded IOB would have to be conducted before this was possible.

Integrating Approximate Adders

To further reduce the area complexity of this design, it was possible to replace Model Composer's full adder blocks with approximate adder systems in lines with the works of Prabakaran et al (Bharath Srinivas Prabakaran, 2018). A VHDL implementation of these adders could be placed in Model Composer via the use of a 'black box' module. However, the provided VHDL adders were incapable of performing subtraction. To achieve this, a 2's complement inverter was required. This was developed through VHDL, and would simply find the 2's complement inverse of any given input. The coded logic for this was:

```
process(sys_in)
begin
    sys_out <= not sys_in + "0000000000000001";
end process;
end Behavioral;
```

The total 1D DCT architecture for this implementation can be found in Figure 15.

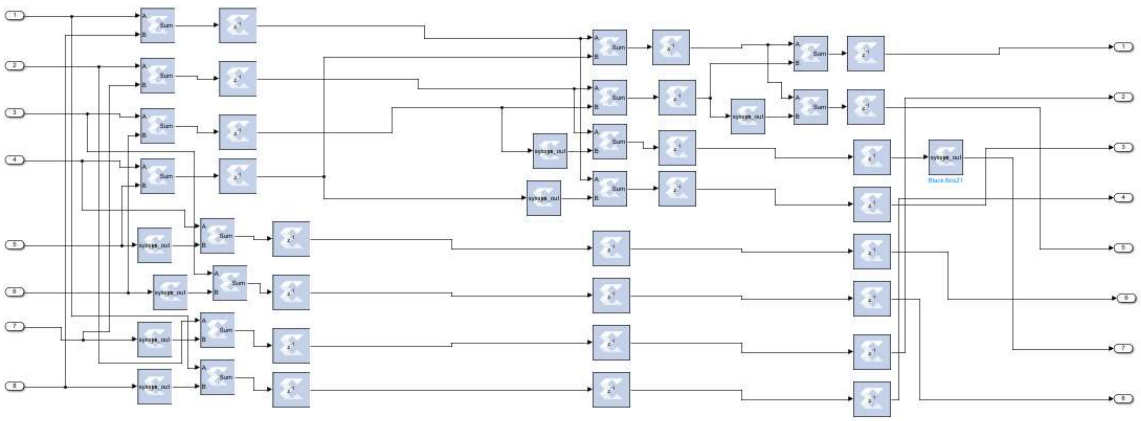

Figure 15: Hardware architecture for the approximate adder system

These black box modules would automatically implement n full adders and k approximate adders to fully satisfy the input word length requirements. Full adders operated over the MSBs to ensure the highest possible accuracy. Through thorough testing the accuracy of the 1D DCT could be gauged alongside the increase of k approximate adders. It was found that increasing the number of approximate adders was inversely proportional with MSE and PSNR performance. This decline in performance can be found in Table 4 using filter co-efficients r=10.

Table 4: Performance metrics compared with k approximate adders

Metric	Approximate Adders (k)								
	0	2	4	6	8	10	12	14	16
MSE	263	263	265	306	1281	6072	9735	11638	11638
PSNR	23.928	23.926	23.891	23.267	17.052	10.297	8.247	7.471	7.471

From the above table and figure, it can be determined that a total of 6 approximate adder values can be attained before the system falls below the design requirement of PSNR > 20dB. By using k=6, an updated table of performance metrics became:

Table 5: Finalised table of performance comparisons

Metric	Implementation			
	Software DCT	Software ADCT	Hardware ADCT	Approximate Hardware
MSE	149.168	263.150	263.150	306
PSNR	26.393	23.928	23.928	23.267

Interestingly, the integration of approximate adders increased the critical path delay of the system significantly. This is due to the architecture of the multibit adders, which required that the logic move through multiple layers of VHDL adder architecture, not including the inverters required for subtraction. Figure 17 shows the timing report of the system for 10ns clock periods, which failed by 1.171ns. It also shows a marked increase of 16 logic levels, compared to the maximum 7 from the full adder system. Figure 16 shows the critical path within this architecture.

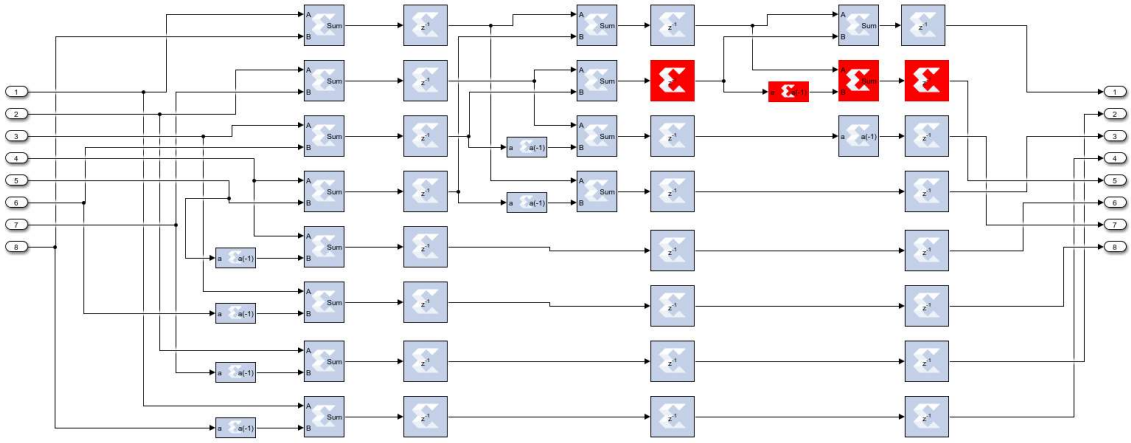

Figure 16: Critical path for the approximate adder system

	Slack (ns)	Delay (ns)	Logic Delay (ns)	uting Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Cloc	Path Constraints
1	-1.171	11.128	2.319	8.809	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
2	-1.109	11.11	2.275	8.835	15	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
3	-1.05	11.007	2.413	8.594	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
4	-1.049	11.048	2.467	8.581	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
5	-1.014	10.971	2.413	8.558	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
6	-0.967	10.966	2.487	8.479	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
7	-0.936	10.893	2.487	8.406	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
8	-0.932	10.931	2.671	8.26	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
9	-0.923	10.88	2.413	8.467	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
10	-0.921	10.878	2.413	8.465	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
11	-0.921	10.879	2.373	8.506	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
12	-0.912	10.911	2.262	8.649	15	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
13	-0.89	10.847	2.433	8.414	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
14	-0.868	10.825	2.671	8.154	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
15	-0.868	10.826	2.717	8.109	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
16	-0.865	10.823	2.413	8.41	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
17	-0.862	10.553	2.322	8.231	15	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
18	-0.859	10.816	2.487	8.329	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
19	-0.858	10.815	2.427	8.388	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
20	-0.843	10.8	2.717	8.083	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
21	-0.843	10.8	2.413	8.387	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
22	-0.841	10.8	2.486	8.314	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
23	-0.84	10.797	2.413	8.384	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
24	-0.834	10.791	2.383	8.408	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
25	-0.826	10.825	2.39	8.435	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
26	-0.825	10.782	2.596	8.186	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
27	-0.824	10.823	2.487	8.336	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
28	-0.822	10.779	2.467	8.312	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]
29	-0.804	10.761	2.537	8.224	16	approx_add...	approx_add...	clk	clk	create_clock -name clk -period 10 [get_ports clk]

Figure 17: Timing report for the approximate adder system

By reducing the clock speed to 12ns, it was possible to pass timing analysis tests. This comes with a corresponding 83.33MHz throughput frequency, which is significantly slower than the 200MHz from the original system.

Lastly, the total memory utilisation could be found in Figure 18. Although the area consumption for lookup tables and slice registers had decreased to 1973 and 6882 respectively, the bonded IOB remained the same at 2049 registers. This is due to the lack of change in the input-output for the Model Composer architecture.

Resource	Utilization	Available	Utilization %
LUT	1973	63400	3.11
FF	6882	126800	5.42
IO	2049	210	975.71

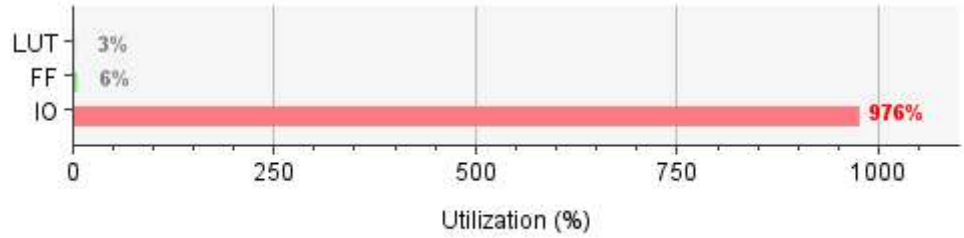

Figure 18: Memory utilisation for the approximate adder system

Comparison and Conclusion

This research paper has been successful in its goal of developing a hardware implementation for 2D DCT image compression. Through successful pipelining and word length selection, it was possible to increase the throughput of the original hardware design to 200MHz, with accuracy readings on par with the software implementation. This was twice the throughput of the unoptimized pipeline, which could only generate results every second clock cycle.

In substituting the full adders with an approximate adder system, it became possible to generate comparable outputs using 6 approximations. This also decreased the lookup table and slice register utilisation to 1973 and 6882 respectively. However, due to the increased logic level complexity, the throughput was reduced to 83.33MHz, making it the slowest and least accurate model trialled. Furthermore, the area saved through approximations wasn't large enough to make any meaningful difference to total resource consumption. It is recommended that this approach is not taken compared to the full adder system.

Finally, the potential for parallelisation exists due to the low memory utilisation for all hardware implementations. It would be possible to achieve this by developing a Model Composer model that could run up to 16 8*8 2D DCTs simultaneously. However, the bonded IOB consumption would have to be reduced to manageable input levels before this was possible. This is a recommendation for future research, as it would be possible to increase throughput by 1600%.

References

Bharath Srinivas Prabakaran, S. R. (2018). DeMAS: An Efficient Design Methodology for Building Approximate Adders for FPGA-Based Systems. *IEEE*.

Kulasekera, S., Madanayake, A., Wijenayake, C., Bayer, F. M., Suarez, D., & Cintra, R. J. (2015). Multi-beam 8×8 RF aperture digital beamformers using multiplierless 2-D FFT approximations. *IEEE*.

Uma Sadhvi Potluri, A. M. (2014). Improved 8-Point Approximate DCT for Image and Video Compression Requiring only 14 Additions. *IEEE Transactions on Circuits and Systems*, 1727-1740.