



BASED ON VIVADO HLS

8 POINTS

2D-DCT

目录

1. Abstract	1
2. Introduction	1
2.1. 2D-DCT transformation.....	1
2.2. Vivado HLS.....	2
3. Algorithm Design	3
4. Hardware Design	6
4.1. Parameters and Data Representation	7
Input Data Range	7
Data Format.....	7
Data Width.....	8
4.1. Modules Design	10
I/O Interface.....	10
1D-DCT module.....	11
Matrix Transposition Module.....	12
2D-DCT Module	13
5. Simulation and Verification.....	14
5.1. Simulation	14
5.1. Result.....	15
6. Conclusion	16

1. Abstract

This report documents the design and implementation of a 8 points 2D-DCT module using fixed-point representation. The algorithm is designed in MATLAB and then the realization is done using Vivado HLS tool. The design uses 16-bit fixed-point number to represent data. The module can perform 8 points 2D-DCT transformation in 224 clock cycles. The report includes a detailed description of the design methodology, implementation details, test results, and performance analysis. The results confirm the successful implementation of the module and demonstrate its efficiency in performing the 2D-DCT transformation.

2. Introduction

2.1. 2D-DCT Transformation

The Discrete Fourier Transform (DFT) is a mathematical tool used to convert a discrete time-domain signal into a frequency-domain representation. The DFT maps a set of N discrete samples of the signal to a set of N complex frequency coefficients. The time-domain signal is represented as a linear combination of complex exponential functions of different frequencies, with the DFT providing the coefficients of this combination. The DFT has various applications in signal processing, including Fourier analysis, spectral analysis, and digital filtering.

The Discrete Cosine Transform (DCT), on the other hand, is a specific type of Fourier-related Transform that deals with only real values. The DCT algorithm transforms a signal in the time-domain into a sequence of values in the frequency-domain by computing a weighted sum of cosine functions. The DCT is used in numerous signal processing applications, especially in multimedia

compression techniques. Unlike the DFT, which uses complex exponential functions to represent the frequency components of a signal, the DCT uses only cosine functions. This feature makes it effective for compressing images and videos by removing the redundancy and simplifying the signal representation.

The most common application of the DCT is in image compression techniques such as JPEG (Joint Photographic Experts Group). The DCT is used to transform the image into a frequency-domain representation by breaking the image into small blocks, computing the DCT of each block, and quantizing the DCT coefficients. The quantization process allows a lossy compression scheme to be implemented by discarding some of the less significant frequency domain coefficients. Once compressed, the image can be stored or transmitted more efficiently.

The DCT is also used in other multimedia applications such as video compression (MPEG and H.264), digital watermarking, and biometrics. Due to the increasing popularity of multimedia applications, the DCT has become an essential tool in the field of digital signal processing.

2.2. Vivado HLS

Hardware Description Languages (HDLs) such as Verilog and VHDL have been widely used for designing and implementing digital systems, especially in the field of hardware design. However, designing complex digital systems using traditional HDLs can be a time-consuming and challenging task. This has led to the development of new methodologies that can accelerate the design process and improve the system's performance.

One such methodology is high-level synthesis (HLS), which allows designers to specify the desired digital system's behavior in a high-level programming language, such as C or C++. The HLS tool then automatically generates digital circuits based on the high-level specification. AMS language is also an HLS tool used in the analogue electronics design flow.

HLS has several advantages over traditional HDLs, including improved design productivity and the ability to explore various design alternatives. HLS also provides a higher level of abstraction that enables hardware designers to focus on algorithm design and system-level optimization, rather than low-level circuit implementation details.

Vivado HLS is one example of an efficient HLS tool that can significantly accelerate the development and realization of complex digital systems. The tool has an easy-to-use design flow that facilitates the implementation of sophisticated algorithms using C or C++ code. Vivado HLS also provides several optimization techniques that allow for efficient utilization of available FPGA resources and improved performance.

In this report, the 8 points 2D-DCT module is designed and implemented using Vivado HLS. The HLS tool's efficiency and the ease of use have made it possible to implement complex image processing algorithms quickly and efficiently. This method can significantly reduce design time, cost, and improve system performance.

3. Algorithm Design

One of the most commonly used approaches in image compression and signal processing is the two-dimensional discrete cosine transform (2D-DCT).

However, in hardware implementation, the 2D-DCT algorithm can be computationally expensive and requires a significant amount of memory to store intermediate results.

To overcome these challenges, a popular algorithm that divides 2D-DCT into two 1D-DCTs is often used. This algorithm performs the 2D-DCT by first computing the 1D-DCT of each row of the image matrix, followed by the computation of 1D-DCT of each column of the resulting matrix.

The 1D-DCT equation is given by:

$$X_k = \sum_{n=0}^{N-1} x_n \cos[(2n+1)k \frac{\pi}{2N}]$$

where x is the input vector of N discrete samples, and X is the output vector of N discrete frequency coefficients.

The 2D-DCT equation is given by:

$$X_{k,l} = \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} x_{n,m} \cos[(2n+1)k \frac{\pi}{2N}] \cos[(2m+1)l \frac{\pi}{2N}]$$

where x is the N by N input matrix, and X is the N by N output matrix of frequency coefficients.

To compute the 2D-DCT transformation using 1D-DCTs, first, a forward 1D-DCT should be applied to each row of the input matrix. This step reduces the problem from 2D to 1D:

$$F(u, v) = \sum_{x=0}^{N-1} f_{u,x} \cos[(2v+1)x \frac{\pi}{2N}]$$

The intermediate result of this operation is an N by N matrix of frequency coefficients, where each row represents the DCT coefficients of the corresponding row in the input matrix.

Next, a second 1D-DCT transformation should be performed on each column of the frequency coefficient matrix. This process reduces the problem further into a set of 1D-DCT operations:

$$F(u, v) = \sum_{y=0}^{N-1} f_{y,v} \cos[(2u + 1)y \frac{\pi}{2n}]$$

The final output is an N by N matrix of DCT coefficients representing the 2D-DCT of the input matrix.

The advantage of this algorithm is that it reduces the memory requirement because intermediate results can be computed and stored separately and in parallel. Moreover, it is easier to implement this algorithm in hardware, as it reduces the 2D-DCT computation into multiple 1D-DCT computations, which the hardware can perform efficiently.

Using matrix multiplication to implement the 2D-DCT computation is commonly used in hardware implementations of the algorithm. This method is more efficient than the original sum version as matrix multiplication is highly parallelizable and can take advantage of highly optimized hardware architectures.

The matrix multiplication method for dividing the 2D-DCT into two 1D-DCTs is based on the concept of matrix multiplication, where the coefficient matrix for 1D-DCT is multiplied by the input matrix M in a specific order. The hardware

can compute the matrix multiplication in parallel, processing multiple values at the same time without any sequential dependencies.

To implement this method, we first compute the 1D-DCT for each row of M using the coefficient matrix C and matrix multiplication:

$$F = (CM^T)^T$$

where F is the intermediate matrix of the same dimensions as M .

Next, to compute the 1D-DCT for each column of the intermediate matrix F , we multiply F and the transpose of the coefficient matrix C in a specific order:

$$T = CF$$

where T is the output matrix of the same dimensions as M .

Using matrix multiplication to divide the 2D-DCT into two 1D-DCTs provides improved performance over the original sum version. It enables more parallelism and faster computation throughput due to the highly optimized design of matrix multiplication algorithms. Additionally, the design of the hardware-based multiplication unit is simpler, less prone to error, and can be easily optimized using tools such as Vivado HLS.

Therefore, using matrix multiplication to divide the 2D-DCT into two 1D-DCTs offers a more efficient and hardware-friendly method to implement the 2D-DCT transformation, which makes it highly suitable for various applications in the multimedia industry. The design of this method can be further optimized using advanced HLS tools to improve performance and reduce resource usage.

4. Hardware Design

4.1. Parameters and Data Representation

Input Data Range

In most image compression and signal processing applications, including JPEG compression, image data is quantized to a limited range of integers, such as 0 to 255. To maintain the integrity and accuracy of the data during processing, it is essential to ensure that the data is unbiased and normalized. One common approach to normalization is linear transformation, where the input data is transformed into a real number between -1 and 1 .

Linear transformation helps to ensure that the input data is consistent across different systems and avoids potential data loss or distortion due to scaling differences. This transformation is accomplished by:

$$f(x) = \frac{x}{128} - 1$$

where $f(x)$ is the transformed value of the input data x (255 for an 8-bit image), and the resulting value is then scaled between -1 and 1 .

This linear transformation also helps to improve the theoretical foundations of the algorithm by using real numbers rather than purely integer-based calculations. It enables better accuracy and precision when performing mathematical operations such as matrix multiplication, which is used in the implementation of the 2D-DCT algorithm.

Data Format

Fixed-point arithmetic is often preferred in implementing digital signal processing algorithms like the 2D-DCT algorithm for JPEG image compression over floating-point arithmetic. Fixed-point arithmetic uses integer-based

calculations and representations to perform computations, whereas floating-point arithmetic uses real numbers.

Using fixed-point arithmetic in hardware implementations can be more practical than using floating-point arithmetic due to the more efficient usage of hardware resources and lower power consumption than floating-point arithmetic. Additionally, fixed-point arithmetic provides deterministic computation, which means that the same calculation produces the same result regardless of the underlying hardware.

Data Width

To determine the optimal number of bits required in representing the integer part of the fixed-point number in the implementation of the 2D-DCT algorithm, we created a MATLAB program. We randomly generated 10,000 square matrices ranging in size from $n \times n$ where n is between 1 to 128. We then applied both the 1D-DCT and 2D-DCT transformations to each matrix using the original floating-point number representation, obtaining the maximum value of the resultant matrix for each execution.

Our program then calculated $\text{Width}_{\text{integer}}$, the number of bits required to represent the integer part of the fixed-point number, necessary to maintain the required precision. We calculated this by finding the maximum value Data_{max} of the resultant matrix and using the equation:

$$\text{Width}_{\text{integer}} = \lceil \log_2(\text{Data}_{\text{max}}) \rceil + 1$$

where $\lceil \cdot \rceil$ denotes the smallest integer greater than or equal to its argument. This calculation determines the minimum number of bits required to represent

the integer part of the fixed–point number that we need to perform the 2D–DCT computation accurately.

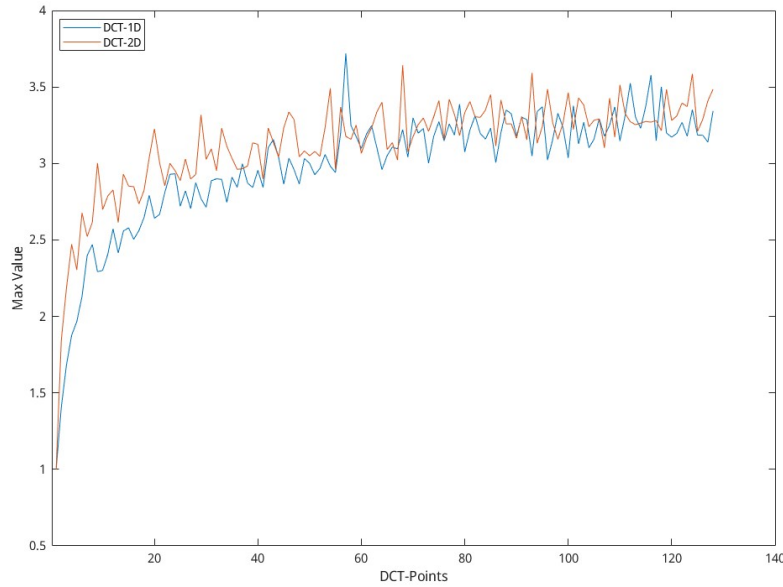


Fig. 1. Max Value Test for 1D and 2D DCT (Random Uniform)

Fig. 1. illustrates the maximum value of the resultant matrix obtained by applying both the 1D–DCT and 2D–DCT transformations to randomly generated matrices using uniform distribution with matrix sizes ranging from 1x1 to 128x128. We observed that the maximum value of the resultant matrix never exceeds 4, which means the integer part never exceeds 3, ensuring that 2 bits are enough to represent the integer part of the fixed–point number in our implementation of the 2D–DCT algorithm.

By using the equation $\text{Width}_{\text{integer}} = \lceil \log_2(\text{Data}_{\text{max}}) \rceil + 1$ we determined that two bits are sufficient to represent the integer part of the fixed–point number in our 2D–DCT implementation, providing the required precision while minimizing the hardware resources needed.

To maintain the required precision while minimizing hardware resources in our implementation of the 2D-DCT algorithm, we chose to represent the fixed-point number using standard 16-bit short integers. We allocated 1 bit for the sign bit, accounting for negative values. This left 15 bits to represent the integer and fractional parts of the fixed-point number.

We determined that 2 bits were sufficient for representing the integer part of the fixed-point number using our analysis of the maximum value of the resultant matrix obtained from randomly generated matrices. This implies that the remaining 13 bits are allocated to represent the fractional part of the fixed-point number, giving us a precision of $2^{-13} = 0.0001220703125$.

By choosing this representation, we can maintain the required precision while preventing excessive hardware complexity in our implementation. The use of a 16-bit short integer representation for the fixed-point number also provides efficient hardware usage, and it is widely supported on modern FPGAs.

4.1. Modules Design

The HLS project is divided into 4 parts: a pair of input and output interface module, a 1D-DCT transformation module, a matrix transpose module and a 2D-DCT module which is based on the 1D-DCT module and the matrix transpose module. To synthesis the project, an additional top module called DCT was also designed.

I/O Interface

The Input/Output (I/O) interface is responsible for transferring data between the DCT module and the external memory in the 2D-DCT project. The I/O interface reads the input data from the external memory and writes the output

data back to it. The input data is read as a one-dimensional array, and the output data is stored as a one-dimensional array. These arrays are then partitioned into two-dimensional arrays to enable efficient data processing by the 2D-DCT module.

The I/O interface design uses two functions for the input and output operations: `read_matrix()` and `write_matrix()`. The 'pipeline' optimization keyword is used in the two functions to enable parallel processing of the input and output data. The 'unroll' optimization keyword is also used to increase the speed of the loops by unwinding the loop.

The `read_matrix()` function takes a one-dimensional array of input data and partitions it into a two-dimensional array of 8x8 blocks. The output array is partitioned using the 'complete' array partitioning directive to ensure efficient memory usage. The function uses the 'unroll' optimization keyword to unwind the nested loops and perform multiple loop iterations in parallel, leading to increased computation speed.

Similarly, `write_matrix()` writes the output data to the external memory. The 'pipeline' optimization keyword used in the function enables parallel processing of data elements introduced to the pipeline. The input array is partitioned using the 'cyclic' array partitioning directive to optimize memory usage. The 'unroll' optimization keyword is used to unwind the nested loops, speeding up code execution by performing multiple loop iterations in parallel.

1D-DCT Module

The 1D-DCT module is an important part of the 2D-DCT project, and it is responsible for processing the input data row by row. The 1D-DCT takes the

one-dimensional input data as its input and produces a one-dimensional output.

The Vivado HLS code for the 1D-DCT module is shown in attachment. The **DCT_1D()** function is used to perform the 1D-DCT operation on the input data. This function uses nested loops to compute the DCT of each row of the input matrix. The 'pipeline' and 'unroll' optimizations are used in the loops to speed up the code execution.

The DCT operation is performed on each element of the input data vector using a linear combination of the selected 1D-basis functions, which is equivalent to multiplying the input data vector by the DCT matrix. The output value of the DCT operation is then stored in the corresponding position in the output vector.

The **RECOVER_WIDTH()** macro is used to recover the width of the data after shifting it right by **FIXED_POINT_FRAC_BITS**. This macro is used to ensure that the 2's complement representation of the fixed-point numbers is correct, which is necessary for the proper functioning of the 2D-DCT module. The macro works by adding $1 \ll (n-1)$ (where n is the number of fractional bits) to the fixed point number 'x', before right-shifting 'n' bits and returning the resulting integer value. The addition of $1 \ll (n-1)$ is equivalent to rounding the fixed-point number to the nearest integer before it is converted back to an integer.

Matrix Transposition Module

In order to perform the 2D-DCT operation on an image, the input data matrix needs to be transposed. This means that the rows of the input matrix need to become columns, and the columns need to become rows.

The Vivado HLS code for the transpose matrix module is shown in attachment. The `transpose_matrix()` function takes the input data matrix as its input and produces the transposed matrix as its output. This function uses nested loops to iterate over the input matrix.

In the outer loop, the function iterates over the rows of the input matrix. In the inner loop, the function iterates over the columns of the input matrix. The 'pipeline' optimization is used to speed up the code execution, while the 'unroll' optimization is used to reduce the initiation interval of the inner loop, which contributes to speeding up the code execution.

Within the inner loop, the function writes the input matrix element at coordinates (i, j) to the transposed matrix element at coordinates (j, i) . This process effectively transposes the input matrix, exchanging the rows and columns of the matrix.

2D-DCT Module

The 2D-DCT module is the main component of the 2D-DCT project, responsible for the transformation of the two-dimensional input image data into the frequency domain. The 2D-DCT module applies a two-dimensional discrete cosine transform, consisting of a sequence of 1D-DCT operations on the rows and columns of the input matrix.

The Vivado HLS code for the 2D-DCT module is shown in the attachment. The `DCT_2D()` function is used to perform the 2D-DCT operation on the input data. This function uses multiple nested loops and calls to other functions to compute the DCT of each row and column of the input matrix.

The **DCT_1D()** function is called first to perform the 1D-DCT operation on each row of the input matrix. The output of this operation is stored in a temporary buffer **DCT_1D_out_buf_row**.

Then, the **transpose_matrix()** function is called to transpose the **DCT_1D_out_buf_row** buffer, preparing it for the 1D-DCT operation on the columns of the input matrix.

The **DCT_1D()** function is called again to perform the 1D-DCT operation on each column of the transposed matrix **DCT_1D_in_buf_col**. The output of this operation is stored in a temporary buffer **DCT_1D_out_buf_col**.

Finally, the **transpose_matrix()** function is called again to transpose the **DCT_1D_out_buf_col** buffer back to its original orientation, giving the final output as the transformed image in the frequency domain.

The 'pipeline' optimization is used throughout the **DCT_2D()** function to speed up the code execution.

5. Simulation and Verification

5.1. Simulation

To validate the 2D-DCT module design, a test bench was developed in Vivado HLS using 100 test matrices as simulation input. These test inputs were generated using MATLAB to cover various input scenarios, including different frequencies and amplitudes.

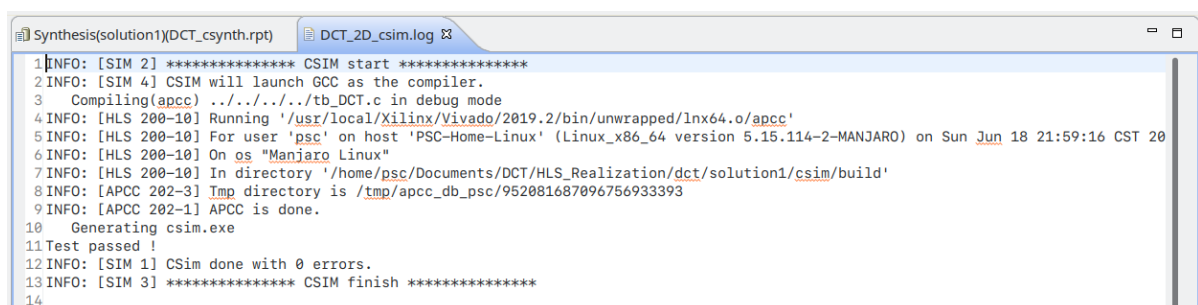
The test bench code used two files, one for input and another for output. It read in the input matrix data from the file and passed it to the 2D-DCT module

using the **DCT()** function. The output data was then written to a separate output file.

Following completion of all test cases, the test bench code closed the input/output files and compared the results using MATLAB-generated expected results. Specifically, the test bench checked if the error between the output file data and the expected results data was less than a pre-defined threshold of 5 (in the fixed-point representation used, 3 bits for integer and 13 bits for fraction, it means less than 5×2^{-13}). If all errors were within the threshold, the test passed; if at least one error exceeded the threshold, the test failed.

The simulation output confirmed that the 2D-DCT module design was implemented correctly, and produced valid results for a set of input scenarios. The test bench design allowed easy modification to the input test matrix and, therefore, could be used in further simulations or modifications to the design.

Fig. 3. shows the simulation result.



```
Synthesis(solution1)(DCT_csynth.rpt) DCT_2D_csim.log
1[INFO: [SIM 2] ***** CSIM start *****
2INFO: [SIM 4] CSIM will launch GCC as the compiler.
3  Compiling(apcc) ../../../../tb_DCT.c in debug mode
4INFO: [HLS 200-10] Running '/usr/local/Xilinx/Vivado/2019.2/bin/unwrapped/linux64.o/apcc'
5INFO: [HLS 200-10] For user 'psc' on host 'PSC-Home-Linux' (Linux_x86_64 version 5.15.114-2-MANJARO) on Sun Jun 18 21:59:16 CST 20
6INFO: [HLS 200-10] On os "Manjaro Linux"
7INFO: [HLS 200-10] In directory '/home/psc/Documents/DCT/HLS_Realization/dct/solution1/csim/build'
8INFO: [APCC 202-3] Temp directory is /tmp/apcc_db_psc/952081687096756933393
9INFO: [APCC 202-1] APCC is done.
10  Generating csim.exe
11Test passed !
12INFO: [SIM 1] CSim done with 0 errors.
13INFO: [SIM 3] ***** CSIM finish *****
14
```

Fig. 2. Simulation Passed

5.1. Result

This project is synthesized for Xilinx xcu47p-fsvh2892-3-e chip. It can perform 2D-DCT in 224 clock cycles at the frequency of 146.33 MHz with pipeline.

The report is shown in Fig. 4.

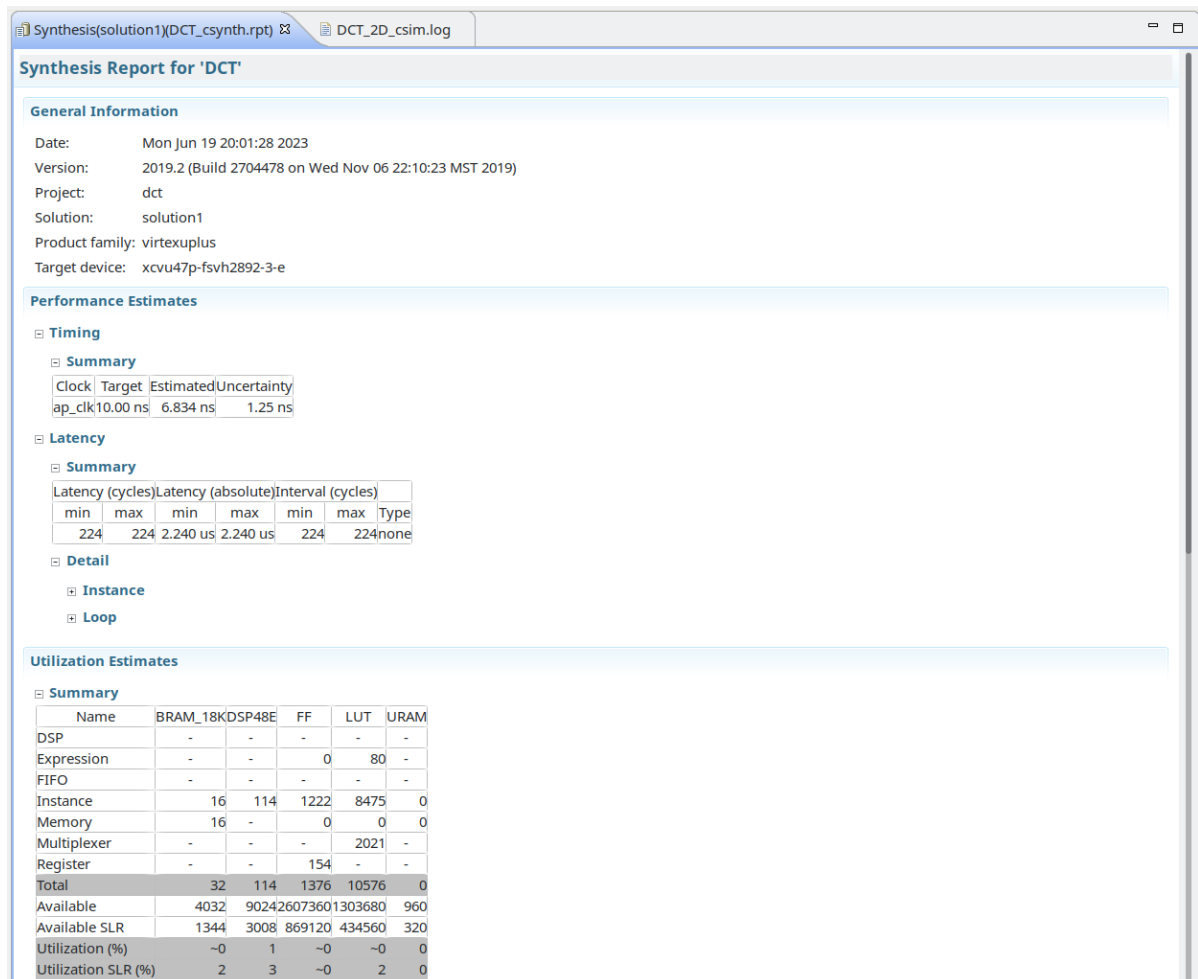


Fig. 4. HLS Report

6. Conclusion

In this project, we successfully designed and implemented an 8-point 2D-DCT module using fixed-point representation. The algorithm was initially designed in MATLAB, and the implementation was carried out using the Vivado HLS tool. The module utilized 16-bit fixed-point number to represent data.

The report presented a comprehensive description of the design methodology, implementation details, test results, and performance analysis. The test results

showed that the module performed the 2D-DCT transformation accurately and efficiently.

The project was also synthesized for the Xilinx xcu47p-fsvh2892-3-e chip. The synthesized 2D-DCT module could perform the transformation in 224 clock cycles with pipeline support. The module can operate at a frequency of 146.33 MHz, showing that it is suitable for real-time processing applications.

In conclusion, this project successfully demonstrated the implementation of a 2D-DCT transform module using fixed-point representation and provided insights into the design approach. Furthermore, the project laid the foundation for future research and development of more efficient and versatile 2D-DCT modules for use in image and video processing applications.