Int. J. Computer Applications in Technology, Vol. 64, No. 3, 2021

Design, Optimization and Implementation of a DCT/IDCT Based Image Processing System on FPGA

Shensheng Tang*, Monali Sinare, Yi Zheng

St. Cloud State University, St. Cloud, MN 56301, USA

stang@stcloudstate.edu; mksinare@go.stcloudstate.edu; zheng@stcloudstate.edu

*Corresponding author

Abstract: In this paper, a discrete cosine transform (DCT) and its inverse transform IDCT are designed and optimized for FPGA using the Xilinx VIVADO High-Level Synthesis (HLS) tool. The DCT and IDCT algorithms along with a filter logic written by C/C++ are simulated for functional verification and optimized through HLS and packaged as custom IPs. The IPs are incorporated into a VIVADO project to form an image processing system for hardware validation. The VIVADO design along with a Xilinx SDK application written by C language is implemented on a Zynq FPGA development board, ZedBoard. A C# GUI is developed to transfer image data to/from the FPGA and display the original and processed images. Experimental results are presented with discussion. The FPGA development method including the DCT/IDCT IP design, optimization and implementation via HLS as well as the VIVADO project integration can be extended to a wider range of FPGA applications.

Keywords: DCT; IDCT; FPGA; VIVADO HLS; IP; ZedBoard; GUI; C/C++; Verilog; C#; Optimization; C/RTL Co-simulation; Hardware Validation.

Reference to this paper should be made as follows: Tang, S., Sinare, M., and Zheng, Y. (20xx) ‘Design, Optimization and Implementation of a DCT/IDCT Based Image Processing System on FPGA’, Int. J. Computer Applications in Technology, Vol. xx, No. xx, pp.xxx–xxx.

Biographical notes: Shensheng Tang is an Associate Professor in the Department of Electrical and Computer Engineering in St. Cloud State University, USA. He received his Ph.D. from University of Toledo, USA. He has eight years of product design and development experience, as hardware engineer, system engineer, and manager respectively, in the electronics and wireless industry. His current research interests include embedded systems, networking (wireless, wired), Internet of things (IoT), and modelling and performance evaluation. He has served or is serving as editor or guest editor for international journals and technical program committee (TPC) member of international conferences. He has produced about 100 peer-reviewed publications in the above areas. He is a senior member of IEEE.

Monali Sinare is a student at the Department of Electrical and Computer Engineering, St. Cloud State University, Minnesota, pursuing a Master of Science in Electrical Engineering. She has received a Master of Science degree in Electronics Science from Savitribai Phule Pune University, Pune, Maharashtra, India, in 2007. She has nine years of experience in FPGA based control system design and development. She has worked on the design and development of Verilog and VHDL modules, testing, and integration of various submodules for FPGA based systems designed for control and monitoring of medical instruments. Her research interests include digital signal processing, image processing, high-level synthesis, hardware-software co-design, and embedded systems.

Dr. Yi Zheng, graduated from Iowa State University, joined faculty of Electrical and Computer Engineering of St. Cloud State University in 1987, served as department chair from 1997 to 2004, full professor since 1993. His research interest is wave propagation including optical wave in motion, ultrasound and EM waves in tissue, and embedded systems. He worked at Very Large Array and Ames Laboratory in 80s. From 1991 to 1992, He was with IBM to apply artificial neural networks for large data processing. From 1993 to 2009, he was with Ultrasound Research Lab of Mayo Clinic to develop neural networks for medical image analysis, and ultrasound vibrometry for measuring tissue elasticity and viscosity. He was with IMI vision for embedded system design and sensor design. From 2006 to 2009, he was with Force 10 Network for high-speed circuit research. He worked with Lift-Touch Studio, Motorola, LionPrecision, Medtronic, Born-Fuke, and Emerson, etc.

1 Introduction

Discrete cosine transform (DCT) [1] is used to represent spatial domain data into their frequency domain representation so that the image information exists in a quantitative form that can be processed. In image processing, DCT can retrieve image features with the help of frequency domain data. The inverse discrete cosine transform (IDCT) can be used to transfer the frequency domain data back to spatial domain data. The computations like DCT require high computational power and resources. Parallel processing, in case of a design involving complex computations such as DCT, can improve the performance of the design with respect to speed of the computation. Field-programmable gate arrays (FPGAs) [2] are programmable logic devices that inherently contain parallel processing and pipelining features and allow flexible reconfigurable computing. FPGA has applications in many areas such as telecommunication systems [3], encoder and decoder systems [4] and electronic circuit implementation [5]. Moreover, high-level synthesis (HLS) tools [6] provided by FPGA vendors such as Xilinx VIVADO HLS [7] allow users to program in high level language such as C or C++ and convert the design into RTL implementation for FPGA programming.

DCT is a widely used transformation technique in many application fields such as signal processing [8]-[11], data compression [12]-[14], spectrum analysis [15][16], and multimedia telecommunications [17][18]. In [8], an approach to the implementation of a DCT for application toward single speech channel encoding was proposed with a detailed computer simulation. In [9], a unified DCT/IDCT algorithm based on the subband decompositions of a signal was proposed and implemented by an FPGA. In [10], a low complexity 2D-DCT architecture was proposed to transform spatial pixels to spectral pixels while taking into account the constraints of the considered compression standard. In [11], a fast discrete cosine transform algorithm was proposed that utilizes the energy compactness and matrix sparseness properties in frequency domain to achieve higher computation performance. In [12], an orthogonal approximation algorithm for the 8-point DCT was introduced for image compression with comparable computational complexity. In [13], a JPEG encoder was designed and implemented using Verilog HDL that involves a complex sub-block DCT. In [14], a computational method based on the Loeffler 1D-DCT algorithm was proposed for 2D 8×8 DCT based on an algebraic integer architecture that maintains error-free computations.

In [15], a DCT spectral analysis technique was employed for producing power spectra from two-dimensional atmospheric fields and extracting information at specific spatial scales. In [16], a new pseudo-spectral method was derived for image interpolation using the DCT symmetric extension in the forward transform. In [17], a DCT based compression and decompression technique was proposed to reduce the volume of multimedia data over wireless channels for multimedia communications. In [18], a DCT based compressed wideband spectrum sensing method was proposed to alleviate the sampling requirements in cognitive radio networks by utilizing the compressive sampling principle and exploiting the unique sparsity structure.

Other notable areas of DCT applications are biometrics [19], geospatial remote sensing [20], and security [21]. As a result, the two-dimensional version of the 8-point DCT was adopted in several imaging standards such as JPEG [22], MPEG-2 [23], H.263 [24] and H.264/AVC [25]. Although fast algorithms can largely reduce the computational complexity of DCT [26][27], floating-point operations are still needed for the accuracy. There is a trade-off between complexity and accuracy. In practice, approximate transforms are used to reduce the floating-point operations in a fast algorithm.

In this paper, DCT and IDCT computation algorithms for an 8-bit grayscale image are designed and optimized using Xilinx VIVADO HLS tool [7] as well as implemented for the Xilinx Zynq-7020 SoC (System on Chip) FPGA device [28]. The DCT and IDCT algorithms along with a filter logic written by C/C++ code are simulated for functional verification in the HLS and packaged as custom IPs (Intellectual Property) for hardware validation in a VIVADO project (via VIVADO Design Suite [29]) as well as tested on a Zynq FPGA development board, ZedBoard [30]. A C# GUI (Graphical User Interface) is designed through Visual Studio IDE [31] for transferring image data to/from the FPGA over a UART (Universal Asynchronous Receiver Transmitter) serial port and displaying the original and processed images. The major contribution of the paper is detailed as follows:

- Design and simulate the DCT and IDCT algorithms with C++ for processing an 8-bit grayscale image data using the HLS.
- Design and simulate a basic filter logic with C++ for attenuating high or low frequencies in the image data using HLS.
- Optimize the DCT, IDCT and Filter logic design by applying various HLS directives, and perform synthesis and C/RTL Co-simulation for functional verification as well as package them as IPs.
- Design and develop a VIVADO project through VIVADO Design Suite to test the developed DCT, IDCT and Filter IP cores.
- Develop a Xilinx SDK (Software Development Kit) [32] application program by C language, which works with the hardware design created with VIVADO Design Suite.
- Design and develop a C# GUI to test the VIVADO project on the Zynq 7020 SoC device and visualize the original and processed images.

The remainder of the paper is organized as follows: Section 2 describes the design, optimization and implementation of the DCT, Filter, and IDCT IPs; Section 3 details the design and implementation of a VIVADO project on FPGA that incorporates the three HLS IPs; Section 4 describes the design and implementation of a C# GUI that communicates with the FPGA design via serial port; Section 5 presents the hardware/software system operation and experimental results; Finally, Section 6 concludes the paper.

2 Design, Optimization and Implementation of DCT, Filter and IDCT Modules

2.1 Background Knowledge

DCT and IDCT: The DCT represents a real valued spatial domain signal in the form of summation of sinusoidal functions of different frequencies. The DCT is widely used in image processing. Compared with discrete Fourier transform (DFT), DCT only uses cosine coefficients. There are four types of DCT, DCT-I, DCT-II, DCT-III and DCT-IV. As the image is represented as two-dimensional (2d) data, the DCT-II that is 2d DCT is mostly used.

The following equation shows the DCT-I computation. The N-point DCT-I (1d DCT) [33] of a discrete signal, $x(n)$ is given as

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

for $k = 0, 1, ..., N-1$,

where $c(k)$, called scale factor, is given as

$$c(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 0, \\ \sqrt{\frac{2}{N}}, & otherwise. \end{cases}$$

The IDCT can be used to transform the frequency domain data back to spatial domain. The 1d IDCT is given as

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

for $k = 0, 1, ..., N-1$.

The 2d DCT is separable. It can be calculated using 1d DCT by taking the 1d DCT of the rows and then taking 1d DCT of the columns of the data generated by the first 1d DCT. In this work, an image is divided into 8×8 matrices of pixel data. An 8-point DCT computation logic is implemented in the HLS. The 1d DCT of rows is calculated using Equation (1). The columns of the resulting matrix are used to calculate the 1d DCT again to get the 2d DCT of 8×8 image data matrix. Similarly, IDCT computation logic is implemented in the HLS using Equation (2). It is observed from Equations (1) and (2) that for constant DCT size, the cosine coefficients can be separately calculated. In this work, the cosine coefficients for DCT and IDCT are separately calculated and stored in text files and will be loaded to program by the HLS.

Frequency Domain Image Representation: In the spatial domain, a pixel is represented by its location and value. In the frequency domain, the pixel location corresponds to the frequency of a pixel occurring within the image and the pixel value corresponds to the amplitude. If the image data in the spatial domain are converted to the frequency domain using DCT, the lower frequencies are in upper left corner of the data matrix and the high frequencies are in the lower right corner. Depending upon the amplitude of pixel in the frequency domain, it can be derived which frequency contains more information in the image.

To calculate the DCT, an image pixel matrix is divided into *N*×*N* blocks. In case of 8-bit grayscale image, *N* can be taken as 8, as the grayscale values vary from 0 to 255 and can be expressed using 8 bits. Hence, DCT calculation of 8×8 pixel data using Equation (1) gives 8×8 frequency domain data.

Filter Logic Description: For image data in frequency domain, larger values of high frequency components indicate that the pixel data are changing faster over a short distance scale, e.g., edges or stripes in an image. On the other hand, larger values of low frequency components indicate that the image contains more smooth regions. The frequency domain analysis of image data provides important information about image which can be utilized to implement different applications such as smoothing, sharpening, removing noise, and edge detection. In this work, since the focus is on the design and implementation of DCT and IDCT logic on FPGA, we treat the frequency domain processing as simple as possible and introduce a basic filter logic in the FPGA implementation, which simply attenuates either low frequency components or high frequency components for the validation of image processing effect. However, due to the modular design method, more complicated filtering algorithms or data processing methods can easily be applied to replace the basic filter logic.

Bitmap File Format: The bitmap file can store the image data in different formats, e.g., grayscale or color. The bitmap does not include any compression, which makes it feasible to be used for testing in image processing applications. The bitmap file structure typically consists of fixed-sized headers and variable-sized pixel data appearing in a predetermined sequence.

In most image processing applications, only pixel data of the image need to be modified. Hence, the headers are separated from the actual image data array. The pixel data are then modified as per the application requirement. For displaying the processed image, the modified pixel data are attached to the previously stored header and then the modified image is displayed.

In this work, 8-bit grayscale bitmap files are used for processing. The pixel data are separated using C# bitmap handling functions. The image data are sent to the FPGA for processing. The processed image data are sent back to the GUI to replace the original image data.

2.2 DCT Module Design and Simulation in HLS

Figure 1 shows the 2d DCT computation algorithm implemented in HLS. The DCT algorithm written by C++ is designed to read an 8×8 image data, calculate the DCT and write the 8×8 DCT output. The 2d DCT computation is separated into two 1d DCT computations. First, the 1d DCT of rows is calculated and then the 1d DCT of columns are calculated. Figure 1 also shows a code snippet of 1d DCT calculation. The input is stored in a 2d local array (local[][]) and used to compute 1d DCT of rows. As per Equation (1), the multiplied and accumulated result for every *k* needs to be multiplied by a coefficient *c(k)*. Here *c(k)* is stored as a constant and used for final multiplication. Then we perform transpose operation on the result (a 2d array) and store it in another local array (local_1[][]). Finally we calculate the 1d DCT of columns and transpose the result back to complete the 2d DCT computation.

The coefficients are stored in floating point format and the computation is also performed using floating point format for accuracy. The input image data are read as unsigned integer. However, the input data will be first stored in the DDR of the ZedBoard, which is 32-bit aligned. The image data are to be passed to the DCT IP from the Processing System (PS) of the Zynq 7020 SoC device through DMA (Direct Memory Access). To keep the data transfer from the DDR to DCT IP less complicated, the 8-bit data are read as 32-bit unsigned integer. Since the DCT computation is in floating point, the resulting output will also be in floating point format. This way when the DCT output is passed to IDCT IP to convert the frequency domain data back to spatial domain, the resulting image data will not be distorted due to DCT/IDCT factor.

The C Simulation in HLS is run to verify the functionality of the logic developed for DCT calculation. A testbench written by C++ is used to provide image data input and print the corresponding DCT output. The input image data are taken from an image data extracted by MATLAB [34]. The output generated by the DCT algorithm is printed and compared with the output generated by the MATLAB simulation. Figure 2 shows the comparison based on the first 8×8 matrix of the image data. Since the calculation is based on floating point, the result is considerably accurate except for slight changes after the decimal point.

2.3 DCT Module Optimization and Package in HLS

Next we run synthesis on the DCT module in the HLS. The latency of the DCT logic is 157.18 μs, which is because of the running of nested loops. Various HLS directives [7] are then applied to the DCT design for optimization. The design includes six sections. The first is for reading input; second for 1d DCT of rows; third for transposing the columns; fourth for 1d DCT of the transposed columns; fifth for transposing the columns back and sixth for writing the output. Each section includes two for-loops. To unroll and parallelize the loops, the HLS PIPELINE directive is used for the inner for-loops. Figure 3 shows the HLS PIPELINE directive applied to the for-loops.

To further utilize the FPGA's parallel processing feature, we perform the ARRAY_PARTITION directive to the input 2d arrays (local[][] and local_1[][]), as shown in Figure 4. Figure 4 also shows that the input and output data interfaces are set as AXI-stream (which is indicated by axis in the HLS) [35].

After applying these directives, we run synthesis again and the latency of the DCT logic has been reduced from 157.18 μs to 5.10 μs. Figure 5 shows the performance comparison of the design before applying any directives (solution1) and after applying the directives (solution 2). Figure 6 shows the loop latency in the design. All the loops are pipelined.

After the synthesis, the C/RTL Co-simulation is run to verify the functionality in RTL (register-transfer level) simulation. The Co-simulation report generated in HLS shows the duration taken for the DCT computation from reading input to writing output as 5.10 μs, which can also be found in the RTL simulation waveform generated in a pop-out VIVADO [29] window (Figure 7). This shows that the DCT design is successful and can now be packaged as a custom IP in HLS. The DCT IP can be added to a VIVADO repository and used by a VIVADO project for system integration.

2.4 Filter Logic Development in HLS

As mentioned earlier, we keep the basic filter logic design as simple as possible in the design. The 8×8 DCT output array is taken as the input of a filter logic. A selector input is taken from outside to decide whether to attenuate lower frequencies or higher frequencies. The selector input is 0 for higher frequencies attenuation and 1 for lower frequencies attenuation. The filtered output is written to an output stream. We run the synthesis to verify the functionality of the filter logic. A testbench is created to provide the test data and store the output in a *.dat* file, which is compared with the output generated by MATLAB simulation. If their difference is below the predefined threshold, the testbench will display "Test passed"; otherwise it will generate an error message. Figure 8 shows the C simulation result.

Similar to the DCT logic, we perform synthesis and optimization to the filter logic and compare the performance of the design without optimization (solution1) and with optimization (solution 2) in Figure 9. It is observed that the directives improve the performance from 4.35 µs to 1.67 µs. Because our filter logic is too simple, the performance improvement is not much.

We then run the C/RTL Co-simulation to verify the functionality in RTL simulation. The Co-simulation report shows the duration taken for the filter logic from reading input to writing output as 1.67 µs, which can also be found in the RTL simulation waveform generated in a pop-out VIVADO window in Figure 10. This shows that the filter logic design is successful and can be packaged as a custom IP for future use.

2.5 IDCT Module Development in HLS

The IDCT module development process is similar to the DCT module. The IDCT algorithm is designed according to Equation (2) with a similar algorithm to Figure 1. The algorithm is designed to read an 8x8 frequency domain image data, perform the IDCT computation and write the 8×8 IDCT output. The 2d IDCT computation is also separated in two 1d IDCT computations.

An HLS C Simulation is run to verify the functionality of the IDCT logic. A testbench is written to give the frequency domain image data input and write the output in a *.dat* file. The output generated by the HLS IDCT logic is compared with the IDCT output from a MATLAB simulation. Figure 11 shows the IDCT logic C simulation result.

We run synthesis and obtain the latency of 121.98 µs. We then perform optimization by applying various HLS directives and run synthesis again. To maximize the parallel processing, the HLS PIPELINE and HLS ARRAY_PARTITION directives are applied to the IDCT logic. The input and output data interfaces are set as AXI-stream. Figure 12 shows the performance comparison of the IDCT design without optimization (solution1) and with optimization (solution 2). It is observed that the optimization improves the performance from 121.98 µs to 4.88 µs. Figure 13 shows the loop latency in the design with all the loops pipelined.

We then perform the C/RTL Co-simulation in HLS to verify the functionality in RTL (register-transfer level) simulation. The Co-simulation report verified that the latency for the IDCT computation is 4.88 µs, which can also be found in the RTL simulation waveform generated in a pop-out VIVADO window in Figure 14. Finally we package the IDCT design as a custom IP in HLS.

3 Design and Implementation of a VIVADO Project

3.1 Design Description

In order to validate the DCT, Filter and IDCT IPs on FPGA, a VIVADO project is developed to incorporate these HLS IPs and run on the ZedBoard, an FPGA development board designed for Xilinx Zynq 7020 SoC devices. Figure 15 shows the block diagram of the VIVADO project including DCT, Filter and IDCT IP cores. The VIVADO design involves two parts, programmable logic (PL) design and processing system (PS) design. The PS transfers the image data to and from the FPGA through AXI DMA IP core. The HLS IPs are controlled by the PS through AXI GPIO IP core.

The PS passes an 8×8 pixel image data to the DCT IP through AXI DMA. The frequency domain output of the DCT IP is connected to the Filter IP. The Filter IP gets a filter selection input that comes from the PS through the AXI GPIO. The Filter IP filters the DCT output data depending upon the selector input. The filtered output of Filter IP is fed into the IDCT IP. The later converts the image data from frequency domain to spatial domain. The 64-pixel image data output of the IDCT IP are transferred to the PS through the AXI DMA. Both the output interface of the IDCT IP and the input interface of DMA IP are of type AXI-stream. However, the AXI-stream interface requires two additional signals, Tlast and Tkeep signals. Hence, a custom IP, i.e., Tlast gen IP, is developed to generate the Tlast and Tkeep signals with the help of the Tvalid signal of the IDCT IP output interface. The three HLS IPs need a start signal (e.g., logic high) in order to start operation. The start signal is sent by the PS through the AXI GPIO.

A Xilinx SDK application is designed to communicate with the GUI over a UART serial port. The initial image data are transferred from the GUI to the FPGA. The IDCT output is read by the PS from the PL and passed to the GUI over the serial port. Hence, the PS mostly handles the data transfer and flow of operation of the PL design.

3.2 Project Implementation in VIVADO

Figure 16 shows the VIVADO block design project, which is composed of the PS, AXI DMA, AXI GPIO, DCT, IDCT, Filter and Tlast_gen IPs. The development of the DCT, IDCT and Filter IPs have been described in the previous section. In the following, we briefly introduce the rest IPs such as AXI GPIO, AXI DMA and Tlast_gen IP. Note that a few IPs are automatically generated during the project creation such as PS Reset, AXI Interconnect, AXI SmartConnect, which are more tightly integrated into the VIVADO design environment for automatic configuration with minimal user intervention.

- AXI GPIO IP

The PS controls the FPGA design flow through AXI GPIO. The processor needs to issue start signal to the DCT and IDCT IPs and filter selection signal to the Filter IP.

Hence, one AXI GPIO channel with two-bit width will be used. Bit 0 is used for the filter selection and bit 1 is used to enable the DCT, Filter and IDCT IPs. As can be seen in Figure 16, the two-bit control signals are differentiated in the PL design using the Xlslice IP from the VIVADO library. The Xlslice IP core slices the bits as per requirement.

- AXI DMA IP

The Xilinx LogiCORE IP AXI DMA in the VIVADO library provides direct memory access between the DDR memory of PS and the AXI-stream peripherals of PL [36] through an AXI_lite interface. In this project, the DMA IP core is used to transfer the image data to the DCT IP and receive the processed image data from the IDCT IP. The DMA reads data directly from the DDR and writes the received data to the DDR. The PS must initiate the data transfer by setting the source address, destination address in the DDR and transfer length [37]. The DMA reads the data from source address and writes the data to destination address. Hence, both the DMA transmit and receive channels are enabled.

The DMA IP has an AXI_lite interface to communicate with the PS. It has AXI stream interfaces to connect on the PL side. The M_AXIS_MM2S is a memory map to slave data output channel, which is connected to the image data input port of the DCT IP. The S_AXIS_S2MM is a slave to memory map AXI-stream data input channel, which is used to receive the processed image data from the IDCT IP. The M_AXI_S2MM is a slave to memory map interface and M_AXI_MM2S is a memory map to slave interface which connect to the high-performance port of the PS for data transfer through AXI smart connect IP core.

- Tlast_gen IP

As mentioned above, the DMA AXI-stream input interface needs two additional signals, Tlast and Tkeep signals, which are not directly generated from the AXI-stream output interface of the IDCT IP. A custom IP is needed to generate the Tlast and Tkeep signals. The Tlast signal indicates the last word in the stream data transfer, which is generated by taking the Tvalid signal from the IDCT IP as input with shifting by a clock pulse. The Tkeep signal indicates the valid data bytes. In this design, the data width is 32 bits (4 bytes), thus the Tkeep signal width is 4 bits.

3.3 SDK Application Design

The Xilinx SDK is an Integrated Development Environment (IDE) that works with VIVADO Design Suite. The PS in the ZedBoard controls the flow of the VIVADO design and performs serial communication with the GUI. Hence, an SDK application project needs to be created to work with the VIVADO project. The flowchart of the SDK application code is shown in Figure 17. The SDK application written by C language mainly performs following functions:

- Initialize UART in interrupt mode
- Initialize AXI DMA and AXI GPIO
- Communicate with GUI over serial port to for commands, responses and image data
- Transfer image data to and from PL through DMA

UART Configuration: The PS UART in the Xilinx SDK has the default baud rate of 115200, data width 8bits, stop bit of 1 and no parity. The UART interrupt is enabled so that an interrupt is generated when data are received on serial port. In Zynq FPGA SoC devices, there is a Generic Interrupt Controller (GIC) that controls the interrupt request from the peripherals [37]. The GIC is configured for monitoring the UART interrupt. The UART can generate an interrupt on multiple events. It includes an interrupt mask register that can enable or disable particular interrupt for the design [28]. A mask value is generated and loaded in the interrupt mask register that enables interrupt for receive_FIFO_full event, receive_FIFO_overflow event and Timeout error event. The Timeout error occurs when the receiver has remained idle for more than the time set in the timeout register.

AXI DMA Configuration: The AXI DMA transfer functions in the SDK application are used to transfer image data to the PL and receive processed image data from the PL. The transfer completion is checked by polling whether the DMA transmit and receive channels are busy. Once the reception is completed, the HLS IPs are disabled, and the processed image data are printed to the serial port. Figure 18 shows the code snippet showing DMA transferring and polling method.

Communication Protocol Design: A communication protocol structure is developed to handle the data transfer between the GUI and the PS. There are three types of data transfer modes: filter setting, start transfer image, and image data matrix. The GUI issues commands and data and the PS issues responses.

The communication protocol has the following command structure from the GUI to the PS.

Command Format

Header byte 1	Header byte 0	Length of data	Command type	Data (if any)

Here Header byte 1 = 255 and Header byte 0 = 254, both are fixed for all commands. Length of data represents the length of data bytes in the packet. As there are only three types of commands, three possible lengths and corresponding command types are defined as shown in Table 1. The data field represents the data sent with a related command. This filed only exists in case of filter setting or image data commands.

Table 1 Data length and command type fields in the protocol structure

Command mode	***Data Length***	***Command type***
Filter setting	1	123
Start transfer image	0	222
Image data matrix	64	111

When the PS receives a command from the GUI, it will send a command response to the GUI. The response fields are separated by a comma character (CC) and the command response ends with a new line character (\n). The command response structure from the PS to the GUI has the following format.

Command Response Format

Header byte	CC	Command response type	CC	Data (if any)	\n

Here Header byte = 255, which is fixed for all commands. As there are three types of commands, the command response types will also be three, which are defined in Table 2. The data field is available only in the case of Image data matrix response, as in this case the processed image data are sent from the PS to the GUI. The image data bytes will be separated by comma character (CC).

Table 2 Command response type field in the protocol structure

Command mode	***Command type***
The filter setting response	123
Start transfer image response	222
Image data matrix response	111

To handle the aforementioned communication protocol, the UART is configured in interrupt mode. The interrupt handler is set to read the received data and raise a flag once the data are read. The main function of the SDK application keeps checking for the data_receive event. When the data_receive flag is high, it indicates that there are data in the receive buffer. Then, the header bytes are checked and then the data length byte is checked. If the length byte is 64, it indicates that the received command is the image data; if the length byte is 1, it indicates that the received command is the filter setting; if the length byte is 0, it indicates that the received command is the start transfer image command. For each option, the processing detail is described in the flowchart in Figure 17.

4 Design and Implementation of a GUI

A GUI is designed using windows form application in .net framework [31]. Figure 19 shows the GUI panel developed in this project, which includes the following functionalities:

- Communicating with the ZedBoard over the serial port
- Setting the frequency
- Loading image to process (from computer)
- Transferring image data to the FPGA
- Receiving image data from the FPGA
- Display both the original and processed images

The GUI running on a computer communicates with the ZedBoard through a UART serial port. The GUI panel includes *Serial Port Settings*, *Select filter types*, *Load image* button, *Transfer image* button, a *Message box* showing the messages during the flow of operation, and two picture boxes for displaying the original and processed images. The picture boxes are set to display images up to 512×512 pixels. When the user clicks on the *Serial Port Settings* menu, a separate form pops out as shown in Figure 20. The user can select the communication port, baud rate, and other options. The communication port number may differ for different devices. Click on the Apply Settings button to select the port settings.

The *Load Image* button allows users to select a bitmap file from the computer. This project is designed for processing 8-bit grayscale bmp images up to 512x512 pixels. The dimension of the image is considered to be square. If an image file is selected, it will be stored in a bitmap variable. The C# provides a Bitmap file structure to store the images. If the image is square in dimensions, it will be displayed in the original picture box; otherwise, an error message will be displayed.

The image data will also be stored in another bitmap variable for further processing. A LockBits() function is used to lock the image data into memory. This function returns a Bitmap data structure, which includes a pointer to the first line of the image data. The pointer is read and used to copy the image data into a local array. Figure 21 provides the code snippet showing the image data copy process. A Marshal.Copy service is used to copy the image data array from the pointer to an unsigned 8-bit integer array. After copying the image data, the image is unlocked. A flag is set to indicate that an image is loaded in the memory. Here most of the variables are declared as public and static so that the variables can be modified and used in different function calls.

Two radio buttons of the filter type settings are provided on the GUI. One is for low pass filter (by default) and the other for high pass filter. If the *Transfer image* button is clicked, an Image_Data_Loaded flag will be checked. If the flag is not set, If the flag is set, the transfer image command will be transmitted and the Processed Image picture box will be cleared (if any); otherwise, a message will be displayed to remind loading image.

For every command, a response is expected from the FPGA. If there are data received on the serial port, the data will be read until the new line character and stored as a string. As shown in the command response format, the command response data bytes are separated by comma characters. The data length of the separated string array is checked. If the data length is two, the command response type will be checked. If the response type is Filter_setting, a message of filter type will be displayed in the message box. If it is Start_transfer_image, the index for Image_data_read will be initialized to zero and a Transmit_data() function will be executed, which transfers the image data matrix of size 8×8 to the serial port.

If the data length is 67 (header byte, command response byte, 64-byte image data, and the comma character at the end of the data), the command response type will be checked. If the type is Image_data_matrix, the received image data will be type-casted to byte and stored into a public byte array. The data are then added to the image array by calling the function Add_data(). Next, the index used to read and transmit the next matrix is incremented and the next matrix is transmitted using the Transmit_data() function call. If the indices have reached to the maximum limit, the processed image will be displayed using displayImage() function call. The flowchart of the C# program for the GUI is shown in Figure 22.

5 System Operation and Results

By putting it all together, we are now ready to run the whole project. We first run the synthesis and implementation of the VIVADO design and generate the bitstream file. We then connect the ZedBoard power cable, UART cable and USB-JTAG cable as shown in the bottom-right of Figure 23. Make sure the jumpers are connected in the JTAG boot mode. Power on the ZedBoard and the power led should glow as green. Next, we export hardware and launch an SDK project from the VIVADO project and load the developed SDK application code. In the opened SDK window, perform the *Program FPGA* operation, the *Done* LED on the ZedBoard will glow as blue. Finally, right click on the SDK application and run "Launch on hardware". The FPGA is ready to take commands from the GUI over the serial port.

The following are the steps to operate the C# GUI and test the VIVADO design.

- At the start of the GUI, a message dialog pops out reminding you to select a serial port, set the serial port using the *Serial Port Settings* Menu.
- Click on *Load Image* button, select an image from a file dialog.
- Select a filter type as per requirement.
- Click Transfer Image and wait till the message box showing "Finished Processing".

The project is designed to process 8-bit grayscale bmp format images. The provided test files are of 256×256 and 512×512 pixels. Note that the image data can be continuously transmitted from the FPGA to the GUI and vice versa, as shown from the two yellow LEDs in a running snapshot in Figure 23. Following are some of the test results.

1) Image Size: 512×512; Filter Type: Lowpass

Figure 24 shows a kid image of size 512×512 processed by the lowpass filter. In the processed image, the edges such as the collar edge, pole outline and water wave outline can be observed to have blurred due to attenuating high frequencies. The sharp changes (edges) in the image corresponds to the high frequencies. When the high frequencies are attenuated, some edges will be blurred. For further image processing, one needs to analyze where most of the energy of the original image lies and then design an appropriate filter. For the kid image, we debugged internal data in program execution and observed that the maximum energy lies in the very small range of low frequency, hence most data of large energy passed the filter and the blurring was minimum. The energy distribution differs for different images. For the power adapter image in Figure 25, the energy distribution tends to diverge towards the high frequency, so the blurring tends to be more obvious, e.g., the adapter information surface. To vary the blurring in the image, an appropriate analysis of the image energy should be done.

2) Image Size: 512×512; Filter Type: Highpass

Figure 26 shows the kid image of size 512×512 processed by the highpass filter. The edges in the image show the high frequency components. As can be observed, attenuating lower frequencies keeps only most of the edges in the processed image and filters out the smooth regions. Again, to vary the high frequency contents in the image, an appropriate analysis of the image energy should be done and specific filters can be designed accordingly. A similar result can be observed for the power adapter image of size 512×512 in Figure 27.

For the images with size of 256×256 and lowpass and high pass filters, similar results can be obtained in Figures 28, 29, 30 and 31.

6 Conclusions

This paper developed a DCT/IDCT based image processing system on the FPGA for processing 8-bit grayscale images. The DCT and IDCT were designed, optimized and implemented in the VIVADO HLS for converting spatial domain image data to frequency domain and frequency domain image data to spatial domain respectively. To understand the frequency domain data processing, a basic filter logic was deigned to attenuate either high frequencies or low frequencies in the frequency domain of image data. The latency performance has been significantly improved for the DCT from 157.18 µs to 5.10 µs, for the IDCT from 121.98 µs to 4.88 µs, and for the filter from 4.35 µs to 1.67 µs. A VIVADO design was developed to validate the DCT, IDCT and Filter HLS IPs. The VIVADO project was targeted for the ZedBoard. The Zynq FPGA processor was configured to pass the spatial domain image data to and from DCT and IDCT IPs. A GUI was developed using C# for controlling the flow of operation and visualizing the images. Finally, the integrated system was tested with various 8-bit grayscale images and experimental results were presented with discussion. For the future work, we consider to include the system implementation with more complicated image processing techniques instead of the basic filter logic in the integrated system.

This paper provided hands-on experience on FPGA based HLS IP design, optimization and implementation as well as the modular-style VIVADO project development method, which can be extended to a wider range of FPGA applications.

Acknowledgement

The authors would like to acknowledge the support of the project development from St Cloud State University, MN, USA through the Early Career Grant (No. 211129) and the support from the Department of Electrical and Computer Engineering through providing its equipment and software.

References

[1] N. Ahmed, T. Natarajan, and K.R. Rao, "Discrete Cosine Transform", IEEE Transactions on Computers, Vol. C-23, pp. 90–93, Jan. 1974.

[2] S. Brown and J. Rose, "FPGA and CPLD architectures: A tutorial", IEEE Design and Test of Computers, 13(2):42–57, 1996.

[3] V. Venkataramanan, S. Lakshmi, and V. A. Kanetkar, "Design and implementation of LTE physical layer on FPGA", International Journal of Computer Applications in Technology, Vol. 61, No. 1-2, pp. 127-134, 2019.

[4] W. M. El-Medany, "Reconfigurable CRC IP core design on Xilinx Spartan 3AN FPGA", International Journal of Computer Applications in Technology, Vol. 55, No. 4, pp. 257-265, 2017.

[5] S. Vaidyanathan, E. Tlelo-Cuautle, A. Sambas, L. G. Dolvis; O. Guillén-Fernández, "FPGA design and circuit implementation of a new four-dimensional multistable hyperchaotic system with coexisting attractors", International Journal of Computer Applications in Technology, Vol. 64, No. 3, pp. 223-234, 2020.

[6] R. Nane, V.M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y.T. Chen, H. Hsiao, and S. Brown, "A Survey and Evaluation of FPGA High-Level Synthesis Tools", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 35 (10): 1591–1604, 2016.

[7] Xilinx, Inc, "Vivado Design Suite User Guide: High-Level Synthesis", UG902, v2020.1, June 2020.

[8] G. Bertocci, B.W. Schoenherr, and D.G. Messerschmitt, "An Approach to the Implementation of a Discrete Cosine Transform," IEEE Transactions on Communications, Vol. Com-30, No. 4, pp. 635-641, April 1982.

[9] L.-T. Ko, J.-E. Chen, H.-C. Hsin, Y.-S. Shieh, and T.-Y. Sung, "A unified algorithm for subband-based discrete cosine transform," Mathematical Problems in Engineering, vol. 2012, Article ID 912194, 2012.

[10] M. Jridi, Y. Ouerhani, and A. Alfalou, "Low complexity DCT engine for image and video compression", Real-Time Image and Video Processing, vol. 8656, pp. 1–9, 2013.

[11] S.E. Tsai, S.M. Yang, "A Fast DCT Algorithm for Watermarking in Digital Signal Processor", Mathematical Problems in Engineering, vol. 2017, Article ID 7401845, 2017. https://doi.org/10.1155/2017/7401845

[12] R.J. Cintra and F.M. Bayer, "A DCT Approximation for Image Compression," IEEE Signal Processing Letters, vol. 18, no. 10, pp. 579-582, Oct. 2011.

[13] S. Sanjeevannanavar and A. N. Nagamani, "Efficient design and FPGA implementation of JPEG encoder using verilog HDL", International Conference on Nanoscience, Engineering and Technology (ICONSET 2011), Chennai, India, pp. 584-588, 2011. DOI: 10.1109/ICONSET.2011.6168038.

[14] D. Coelho, S. Nimmalapalli, V. Dimitrov, A. Madanayake, Renato Cintra, and A. Tisserand, "Computation of 2D 8x8 DCT Based on the Loeffler Factorization Using Algebraic Integer Encoding", IEEE Transactions on Computers, Vol. 67, NO. 12, pp. 1692-1702, 2018.

[15] B. Denis, J. Côté, and R. Laprise, "Spectral Decomposition of Two-Dimensional Atmospheric Fields on Limited-Area Domains Using the Discrete Cosine Transform (DCT)", Monthly Weather Review, Vol. 130, No. 7, pp. 1812–1829, July 2002.

[16] I. Ito, "A New Pseudo-Spectral Method Using the Discrete Cosine Transform", Journal of Imaging, Vol. 6, No. 4, Article 15, 2020. https://doi.org/10.3390/jimaging6040015

[17] P. Telagarapu, B. Biswal and V. S. Guntuku, "Design and analysis of multimedia communication system", 2011 Third International Conference on Advanced Computing, Chennai, India, pp. 193-197, 2011. DOI: 10.1109/ICoAC.2011.6165174

[18] Y. Wang and G. Zhang, "Compressed Wideband Spectrum Sensing Based on DiscreteCosine Transform", The Scientific World Journal, Vol. 2014, Article ID 464895, 2014. http://dx.doi.org/10.1155/2014/464895

[19] Z.M. Hafed and M.D. Levine, "Face Recognition Using the Discrete Cosine Transform", International Journal of Computer Vision, 43(3), pp. 167–188, 2001.

[20] E. Magli and D. Taubman, "Image compression practices and standards for geospatial information systems," in IEEE International Geoscience and Remote Sensing Symposium, Vol. 1, pp. 654–656, Jul. 2003.

[21] A. Solichin and E. W. Ramadhan, "Enhancing data security using DES-based cryptography and DCT-based steganography", 2017 3rd International Conference on Science in Information Technology (ICSITech), Bandung, Indonesia, pp. 618-621, 2017. DOI: 10.1109/ICSITech.2017.8257187

[22] W. B. Pennebaker and J. L. Mitchell, JPEG Still Image Data Compression Standard. NewYork, NY: Van Nostrand Reinhold, 1992.

[23] International Organization for Standardization, “Information technology - Generic coding of moving pictures and associated audio information - Part 2: Video”, ISO/IEC 13818-2:2013. https://www.iso.org/standard/61152.html

[24] K. Rijkse, "H.263: video coding for low-bit-rate communication," IEEE Communications Magazine, Vol. 34, No. 12, pp. 42-45, Dec. 1996, doi: 10.1109/35.556485.

[25] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, “Overview of the H.264/AVC videocoding standard”, IEEE Trans. Circuits Syst. Video Technol., Vol. 13, No. 7, pp. 560–576, July 2003.

[26] C. Loeffler, A. Ligtenberg, and G. Moschytz, “Practical fast 1D DCT algorithms with 11multiplications”, International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, pp. 988–991, May 1989.

[27] E. Feig and S. Winograd, “Fast algorithms for the discrete cosine transform”, IEEE Trans.Signal Process., Vol. 40, No. 9, pp. 2174–2193, Sep. 1992.

[28] Xilinx, Inc, “Zynq-7000 SoC: Technical Reference Manual”, UG585, v1.12.2, July 2018. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[29] Xilinx, Inc, “Vivado Design Suite User Guide: Using the Vivado IDE”, UG893, v2020.1, June 2020. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug893-vivado-ide.pdf

[30] Digilent, Inc, "ZedBoard Hardware User’s Guide", Ver 2.2, January 2014. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[31] Microsoft Corp., Visual Studio 2019. Available: https://visualstudio.microsoft.com/

[32] Xilinx, Inc, “Getting Started with Xilinx SDK”, v2016.2. Available: https://www.xilinx.com/html_docs/xilinx2016_2/SDK_Doc/index.html

[33] B.H. Shakibaei Asli, J. Flusser, Y. Zhao, J.A. Erkoyuncu, K.B. Krishnan, Y. Farrokhi, and R. Roy, "Ultrasound Image Filtering and Reconstruction Using DCT/IDCT Filter Structure", IEEE Access, Vol. 8, pp. 141342-141357, 2020, doi: 10.1109/ACCESS.2020.3011970.

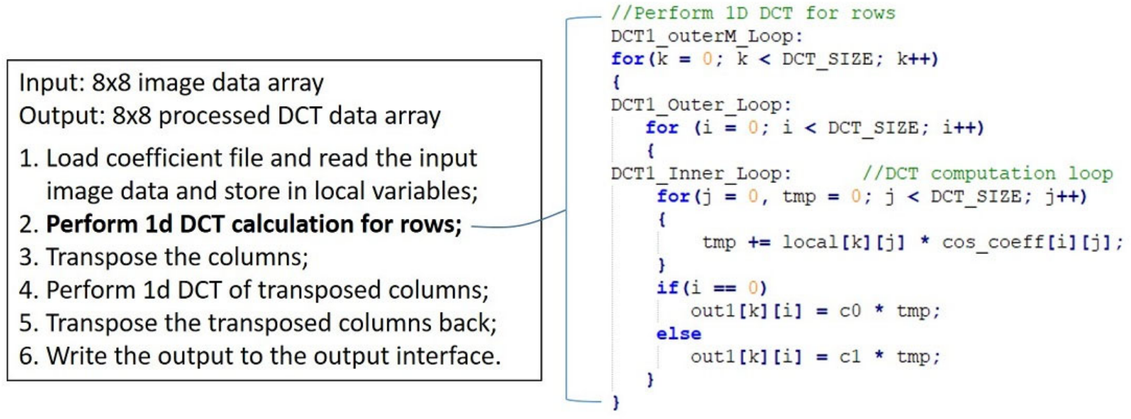
[34] B. Hahn and D. Valentine, Essential MATLAB for Engineers and Scientists, 7th Edition, Academic Press; April 2019.

[35] Xilinx, Inc, “Vivado Design Suite: Vivado AXI Reference”, UG1037, v4.0, July 2017. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[36] Xilinx, Inc, “AXI DMA v7.1: LogiCORE IP Product Guide”, June 2019. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[37] M.A. Enderwitz, R.A. Elliot, C.H. Louise, and R.W. Stewart, The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC, First Edition, Strathclyde Academic Media, 2014.

Appendix


Input: 8x8 image data array
Output: 8x8 processed DCT data array
1. Load coefficient file and read the input image data and store in local variables;
2. **Perform 1d DCT calculation for rows;**
3. Transpose the columns;
4. Perform 1d DCT of transposed columns;
5. Transpose the transposed columns back;
6. Write the output to the output interface.

```
//Perform 1D DCT for rows
DCT1_outerM_Loop:
for(k = 0; k < DCT_SIZE; k++)
{
DCT1_Outer_Loop:
   for (i = 0; i < DCT_SIZE; i++)
   {
DCT1_Inner_Loop:      //DCT computation loop
    for(j = 0, tmp = 0; j < DCT_SIZE; j++)
    {
       tmp += local[k][j] * cos_coeff[i][j];
    }
    if(i == 0)
      out1[k][i] = c0 * tmp;
    else
      out1[k][i] = c1 * tmp;
   }
}
```

Fig 1. Algorithm of the 2d DCT computation and part implementation code

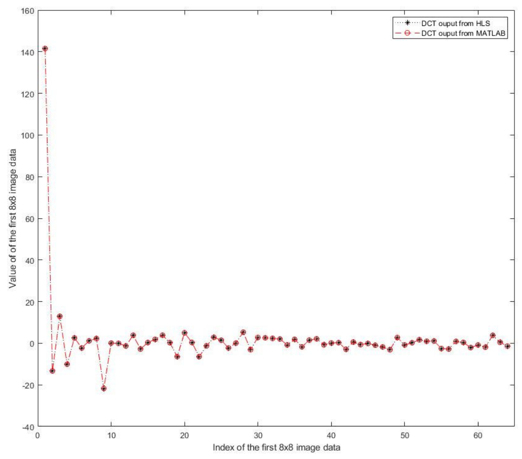
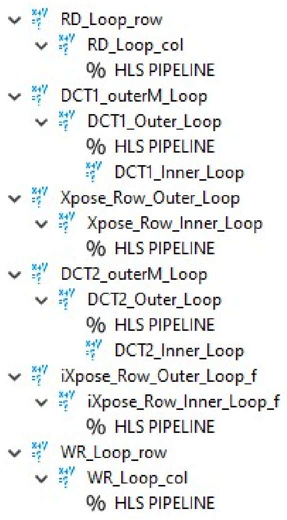
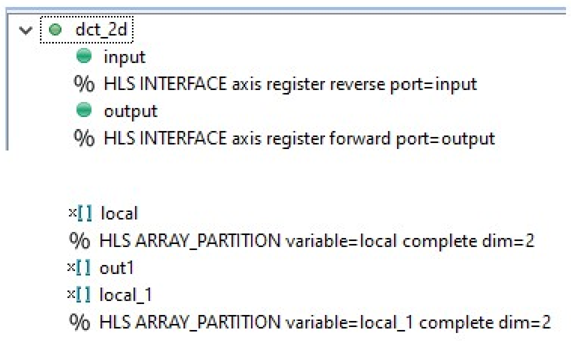


Fig 2. Comparison of DCT results from HLS C simulation and MATLAB calculation for the first 8×8 matrix of the image data



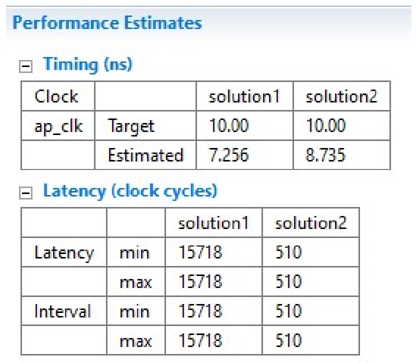
```
RD_Loop_row
  RD_Loop_col
    % HLS PIPELINE
DCT1_outerM_Loop
  DCT1_Outer_Loop
    % HLS PIPELINE
    DCT1_Inner_Loop
Xpose_Row_Outer_Loop
  Xpose_Row_Inner_Loop
    % HLS PIPELINE
DCT2_outerM_Loop
  DCT2_Outer_Loop
    % HLS PIPELINE
    DCT2_Inner_Loop
iXpose_Row_Outer_Loop_f
  iXpose_Row_Inner_Loop_f
    % HLS PIPELINE
WR_Loop_row
  WR_Loop_col
    % HLS PIPELINE
```

Fig 3. HLS PIPELINE directive applied to DCT loops



```
dct_2d
  input
  % HLS INTERFACE axis register reverse port=input
  output
  % HLS INTERFACE axis register forward port=output

 local
 % HLS ARRAY_PARTITION variable=local complete dim=2
 out1
 local_1
 % HLS ARRAY_PARTITION variable=local_1 complete dim=2
```

Fig 4. ARRAY_PARTITION and INTERFACE directives applied to DCT loops



Performance Estimates

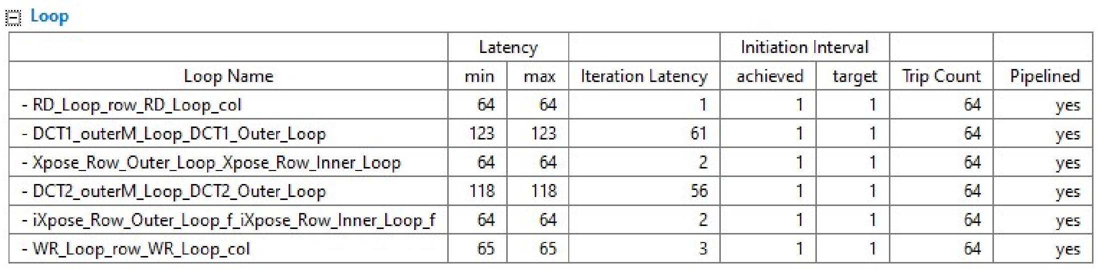
Timing (ns)

Clock		solution1	solution2
ap_clk	Target	10.00	10.00
	Estimated	7.256	8.735

Latency (clock cycles)

		solution1	solution2
Latency	min	15718	510
	max	15718	510
Interval	min	15718	510
	max	15718	510

Fig 5. DCT performance comparison with and without HLS optimization



Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- RD_Loop_row_RD_Loop_col	64	64	1	1	1	64	yes
- DCT1_outerM_Loop_DCT1_Outer_Loop	123	123	61	1	1	64	yes
- Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop	64	64	2	1	1	64	yes
- DCT2_outerM_Loop_DCT2_Outer_Loop	118	118	56	1	1	64	yes
- iXpose_Row_Outer_Loop_f_iXpose_Row_Inner_Loop_f	64	64	2	1	1	64	yes
- WR_Loop_row_WR_Loop_col	65	65	3	1	1	64	yes

Fig 6. Loop latency of the DCT logic after applying the directives

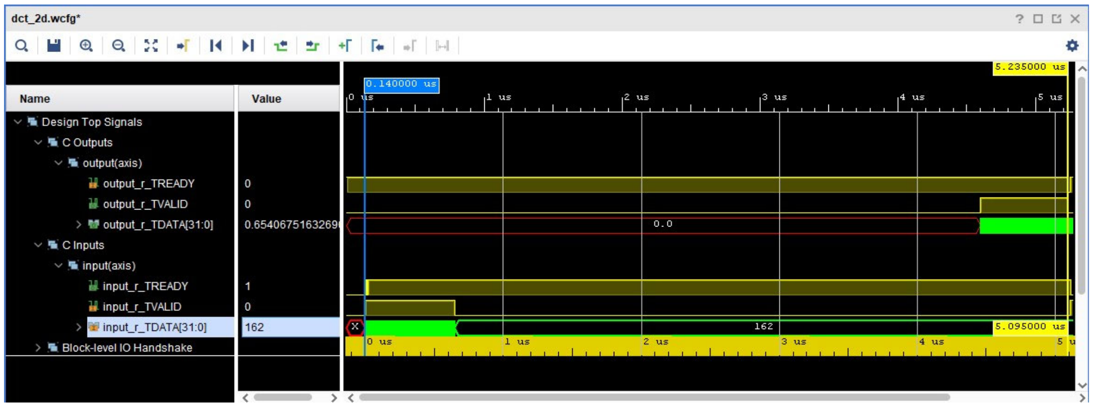
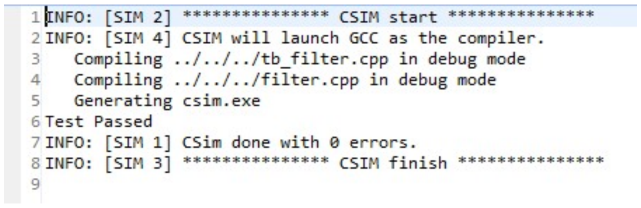
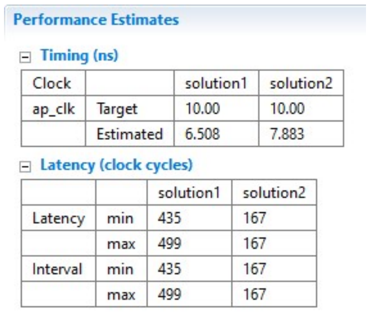


Fig 7. The DCT logic RTL simulation waveforms



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../tb_filter.cpp in debug mode
   Compiling ../../../filter.cpp in debug mode
   Generating csim.exe
Test Passed
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Fig 8. C simulation for the filter logic



Performance Estimates

Timing (ns)

Clock		solution1	solution2
ap_clk	Target	10.00	10.00
	Estimated	6.508	7.883

Latency (clock cycles)

		solution1	solution2
Latency	min	435	167
	max	499	167
Interval	min	435	167
	max	499	167

Fig 9. Filter performance comparison with and without optimization

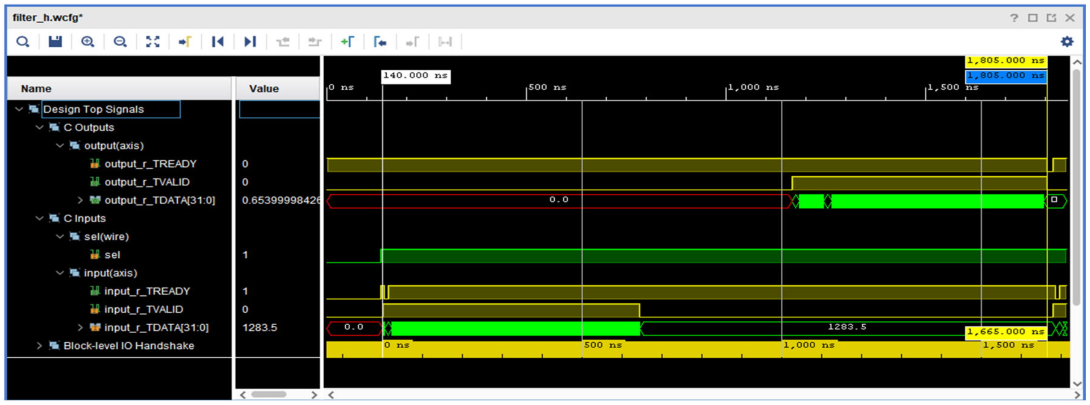
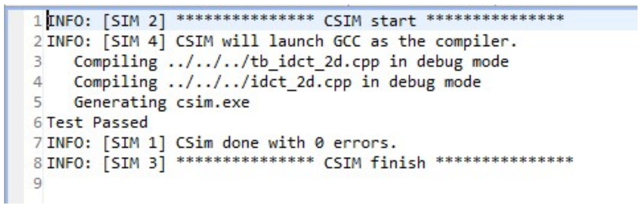
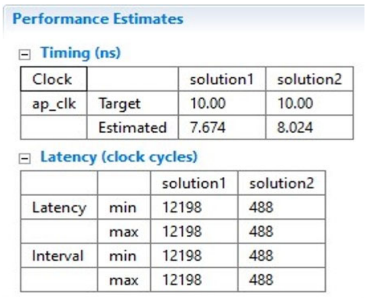


Fig 10. The filter logic RTL simulation waveforms



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../tb_idct_2d.cpp in debug mode
   Compiling ../../../idct_2d.cpp in debug mode
   Generating csim.exe
Test Passed
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Fig 11. C simulation for the IDCT logic



Performance Estimates

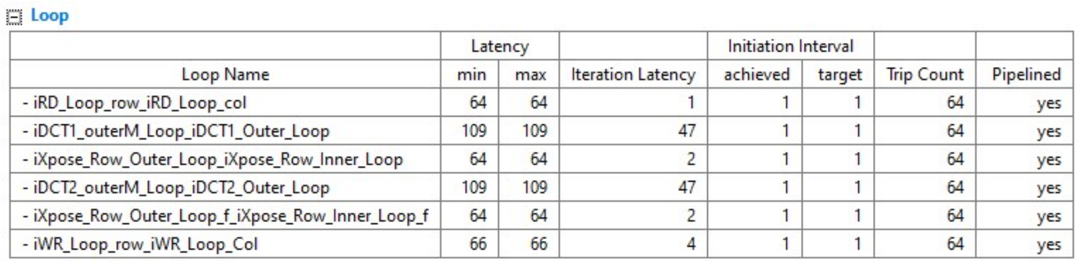
Timing (ns)

Clock		solution1	solution2
ap_clk	Target	10.00	10.00
	Estimated	7.674	8.024

Latency (clock cycles)

		solution1	solution2
Latency	min	12198	488
	max	12198	488
Interval	min	12198	488
	max	12198	488

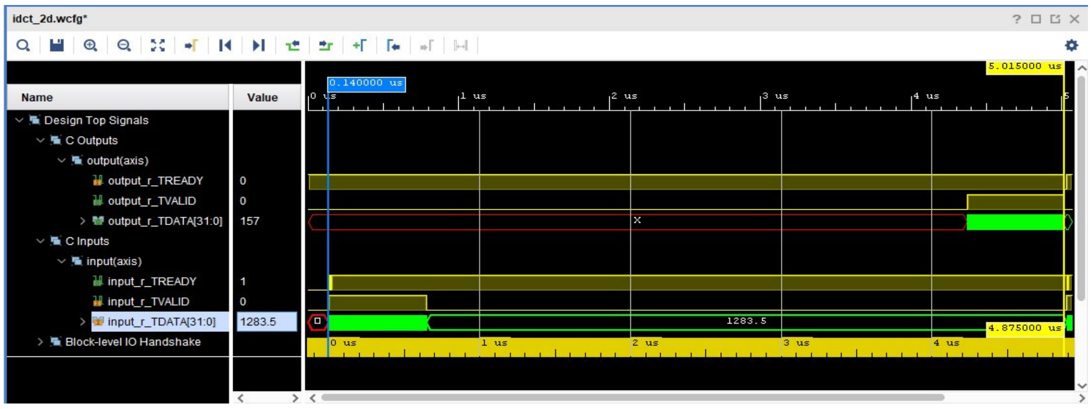
Fig 12. IDCT performance comparison with and without optimization



Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- iRD_Loop_row_iRD_Loop_col	64	64	1	1	1	64	yes
- iDCT1_outerM_Loop_iDCT1_Outer_Loop	109	109	47	1	1	64	yes
- iXpose_Row_Outer_Loop_iXpose_Row_Inner_Loop	64	64	2	1	1	64	yes
- iDCT2_outerM_Loop_iDCT2_Outer_Loop	109	109	47	1	1	64	yes
- iXpose_Row_Outer_Loop_f_iXpose_Row_Inner_Loop_f	64	64	2	1	1	64	yes
- iWR_Loop_row_iWR_Loop_Col	66	66	4	1	1	64	yes

Fig 13. Loop latency of the IDCT logic after applying the directives



idct_2d.wcfg*

Name	Value
Design Top Signals	
C Outputs	
output(axis)	
output_r_TREADY	0
output_r_TVALID	0
output_r_TDATA[31:0]	157
C Inputs	
input(axis)	
input_r_TREADY	1
input_r_TVALID	0
input_r_TDATA[31:0]	1283.5
Block-level IO Handshake	

Fig 14. The IDCT logic RTL simulation waveforms

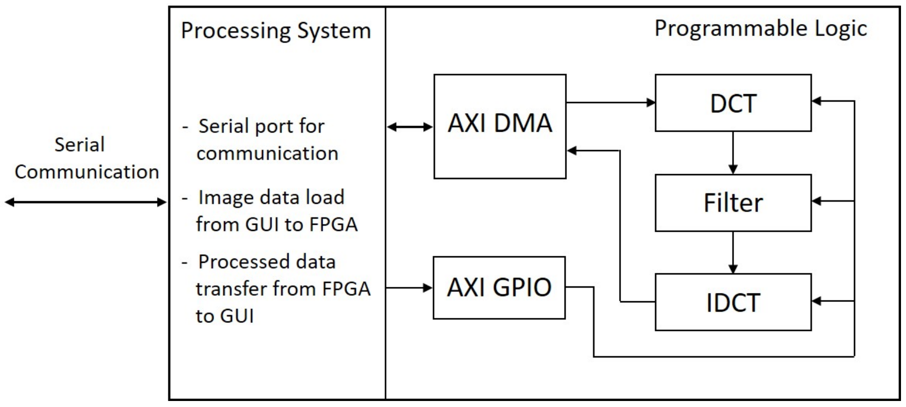

Serial Communication
Processing System
- Serial port for communication
- Image data load from GUI to FPGA
- Processed data transfer from FPGA to GUI
Programmable Logic
AXI DMA
AXI GPIO
DCT
Filter
IDCT

Fig 15. The VIVADO project block diagram

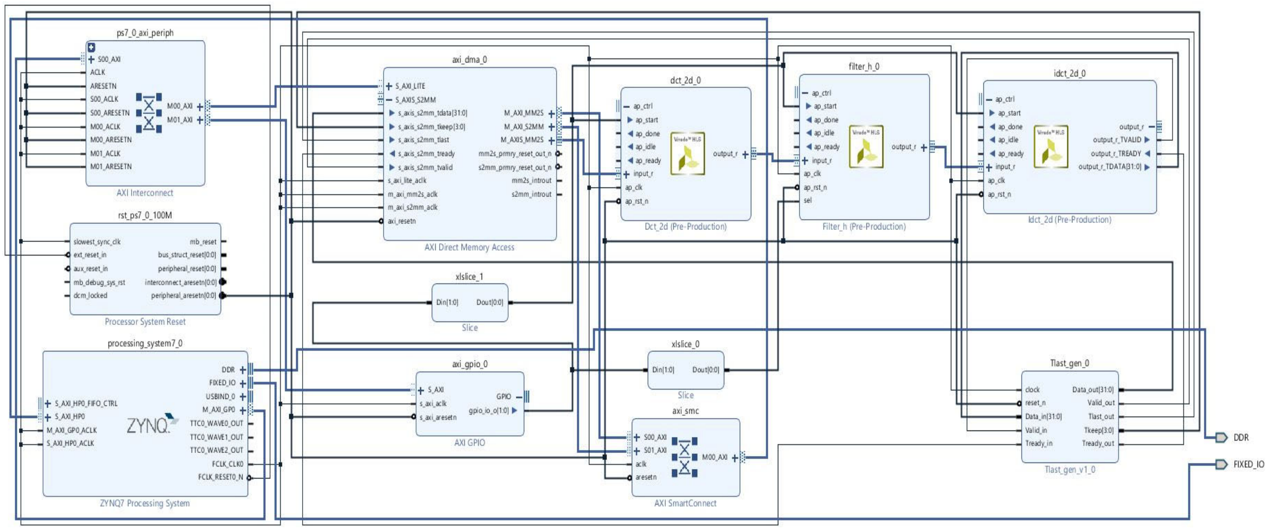


Fig 16. The VIVADO project system construction

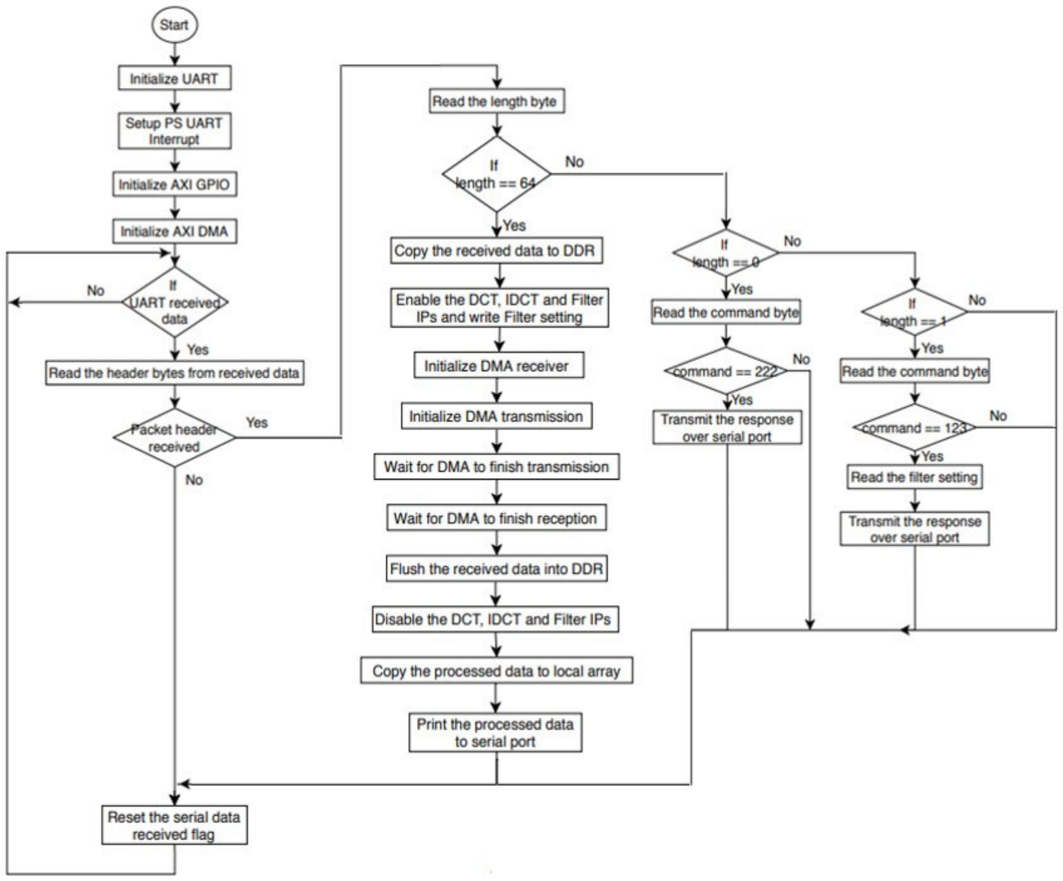
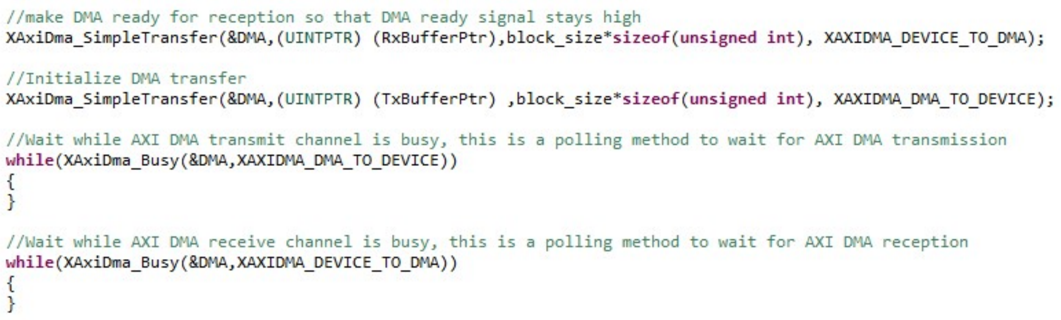
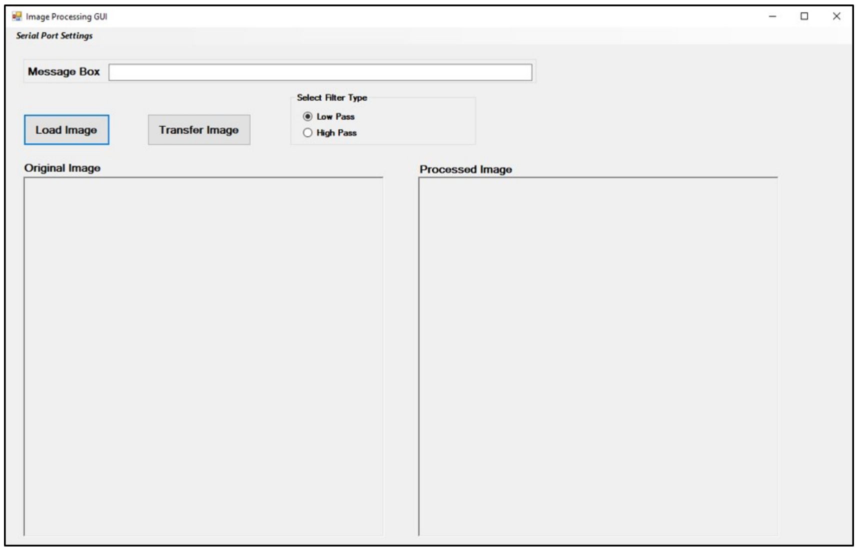

Start
Initialize UART
Setup PS UART Interrupt
Initialize AXI GPIO
Initialize AXI DMA
If UART received data
No
Yes
Read the header bytes from received data
Packet header received
Yes
No
Reset the serial data received flag
Read the length byte
If length == 64
No
Yes
Copy the received data to DDR
Enable the DCT, IDCT and Filter IPs and write Filter setting
Initialize DMA receiver
Initialize DMA transmission
Wait for DMA to finish transmission
Wait for DMA to finish reception
Flush the received data into DDR
Disable the DCT, IDCT and Filter IPs
Copy the processed data to local array
Print the processed data to serial port
If length == 0
No
Yes
Read the command byte
command == 222
No
Yes
Transmit the response over serial port
If length == 1
No
Yes
Read the command byte
command == 123
No
Yes
Read the filter setting
Transmit the response over serial port

Fig 17. Flowchart of the SDK application code



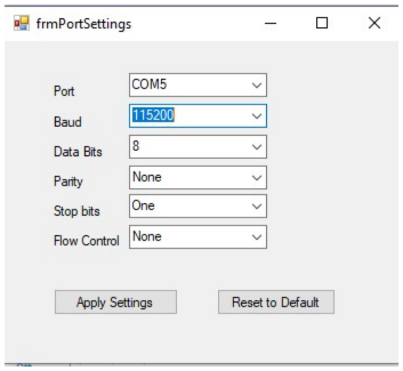
```
//make DMA ready for reception so that DMA ready signal stays high
XAxiDma_SimpleTransfer(&DMA,(UINTPTR) (RxBufferPtr),block_size*sizeof(unsigned int), XAXIDMA_DEVICE_TO_DMA);

//Initialize DMA transfer
XAxiDma_SimpleTransfer(&DMA,(UINTPTR) (TxBufferPtr) ,block_size*sizeof(unsigned int), XAXIDMA_DMA_TO_DEVICE);

//Wait while AXI DMA transmit channel is busy, this is a polling method to wait for AXI DMA transmission
while(XAxiDma_Busy(&DMA,XAXIDMA_DMA_TO_DEVICE))
{
}

//Wait while AXI DMA receive channel is busy, this is a polling method to wait for AXI DMA reception
while(XAxiDma_Busy(&DMA,XAXIDMA_DEVICE_TO_DMA))
{
}
```

Fig 18. AXI DMA transfer code snippet



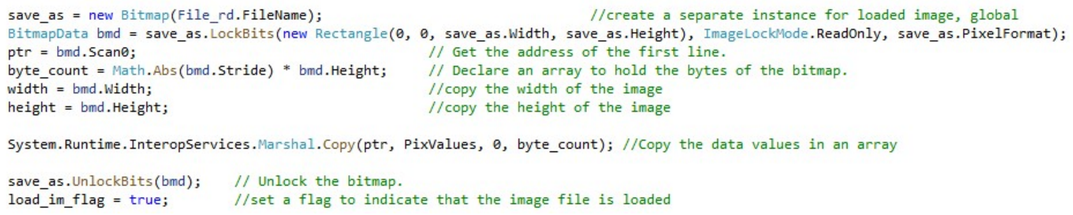
Select Filter Type: [✓ Low Pass] High Pass

Fig 19. The GUI panel for communicating with the ZedBoard



Port	COM5
Baud	115200
Data Bits	8
Parity	None
Stop bits	One
Flow Control	None

Fig 20. The serial port settings menu



```
save_as = new Bitmap(File_rd.FileName);                              //create a separate instance for loaded image, global
BitmapData bmd = save_as.LockBits(new Rectangle(0, 0, save_as.Width, save_as.Height), ImageLockMode.ReadOnly, save_as.PixelFormat);
ptr = bmd.Scan0;                                     // Get the address of the first line.
byte_count = Math.Abs(bmd.Stride) * bmd.Height;      // Declare an array to hold the bytes of the bitmap.
width = bmd.Width;                                   //copy the width of the image
height = bmd.Height;                                 //copy the height of the image

System.Runtime.InteropServices.Marshal.Copy(ptr, PixValues, 0, byte_count); //Copy the data values in an array

save_as.UnlockBits(bmd);    // Unlock the bitmap.
load_im_flag = true;        //set a flag to indicate that the image file is loaded
```

Fig 21. Image data read and stored in array

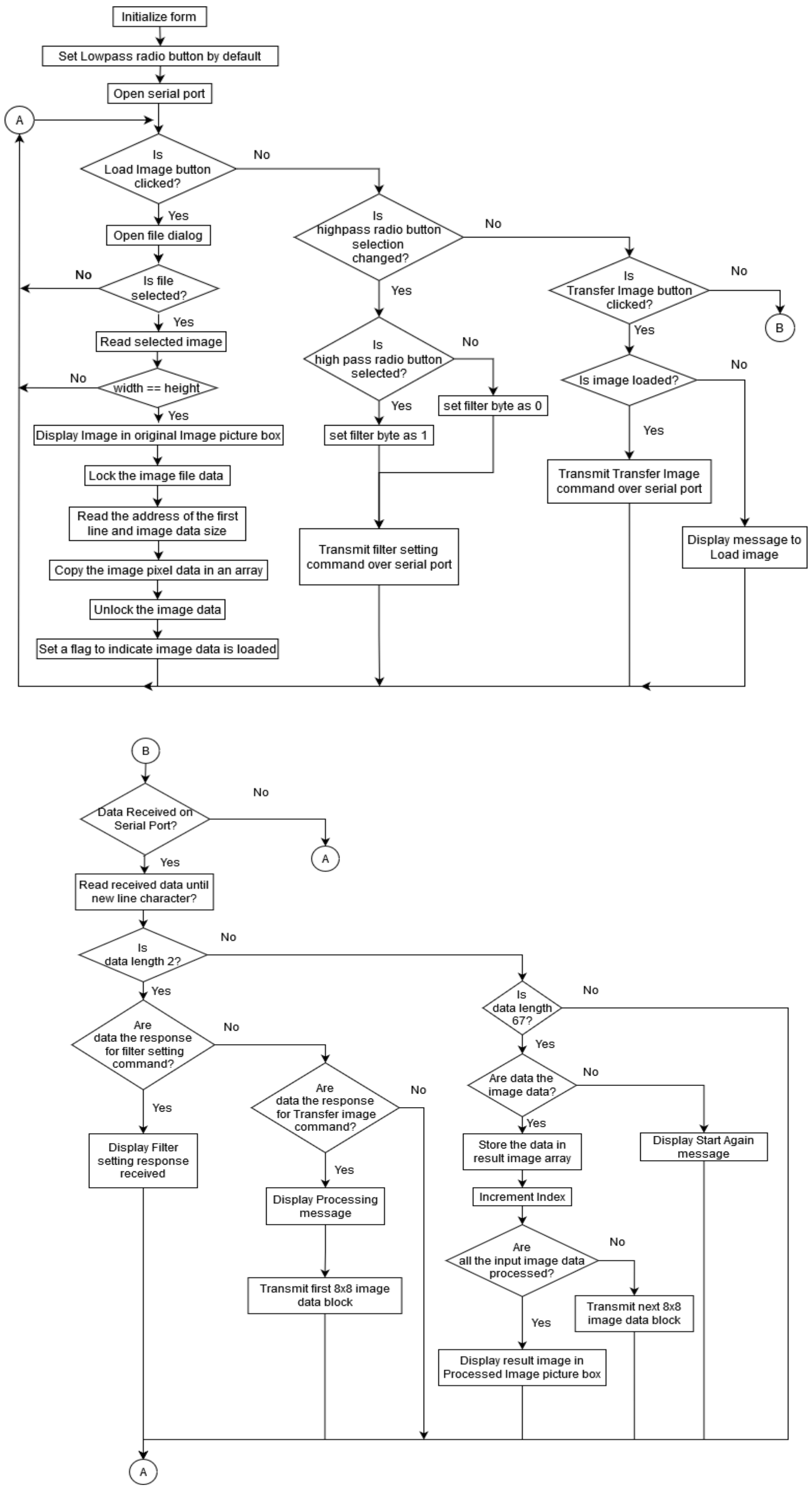

Initialize form
Set Lowpass radio button by default
Open serial port
A
Is Load Image button clicked?
No
Yes
Open file dialog
Is file selected?
No
Yes
Read selected image
width == height
No
Yes
Display Image in original Image picture box
Lock the image file data
Read the address of the first line and image data size
Copy the image pixel data in an array
Unlock the image data
Set a flag to indicate image data is loaded
Is highpass radio button selection changed?
No
Yes
Is high pass radio button selected?
No
Yes
set filter byte as 1
set filter byte as 0
Transmit filter setting command over serial port
Is Transfer Image button clicked?
No
B
Yes
Is image loaded?
No
Yes
Transmit Transfer Image command over serial port
Display message to Load image
B
Data Received on Serial Port?
No
A
Yes
Read received data until new line character?
Is data length 2?
No
Yes
Are data the response for filter setting command?
No
Yes
Display Filter setting response received
Are data the response for Transfer image command?
No
Yes
Display Processing message
Transmit first 8x8 image data block
Is data length 67?
No
Yes
Are data the image data?
No
Yes
Store the data in result image array
Display Start Again message
Increment Index
Are all the input image data processed?
No
Yes
Transmit next 8x8 image data block
Display result image in Processed Image picture box
A

Fig 22. Flowchart of the C# program for the GUI

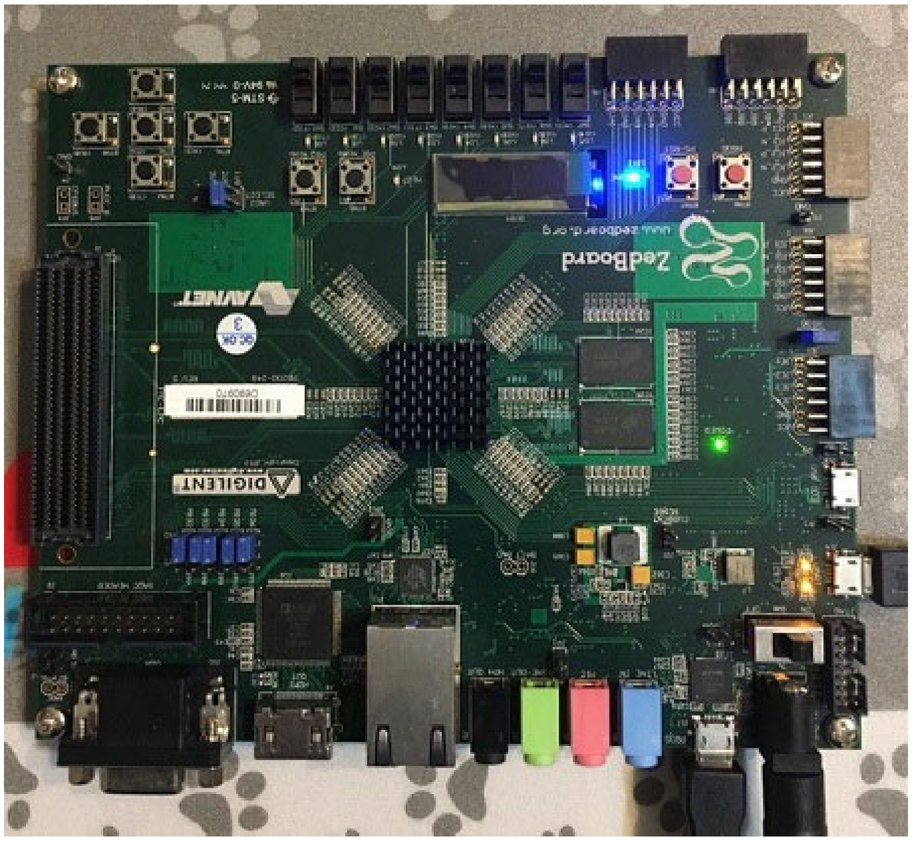
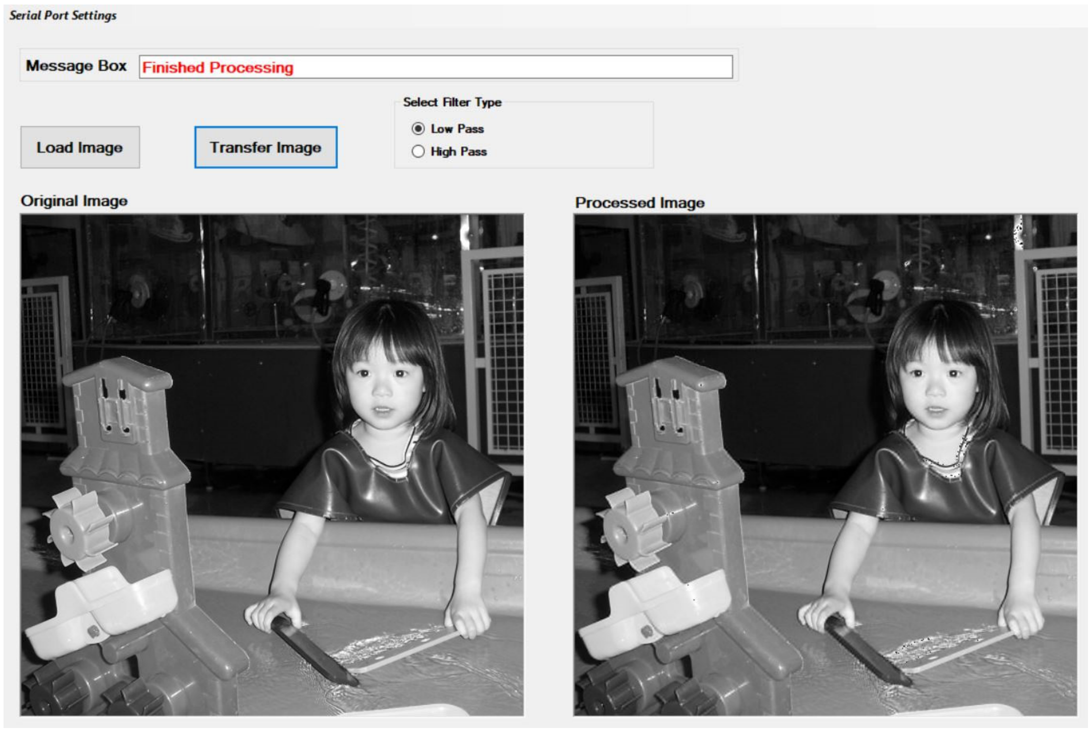


Figure 23. The ZedBoard in running

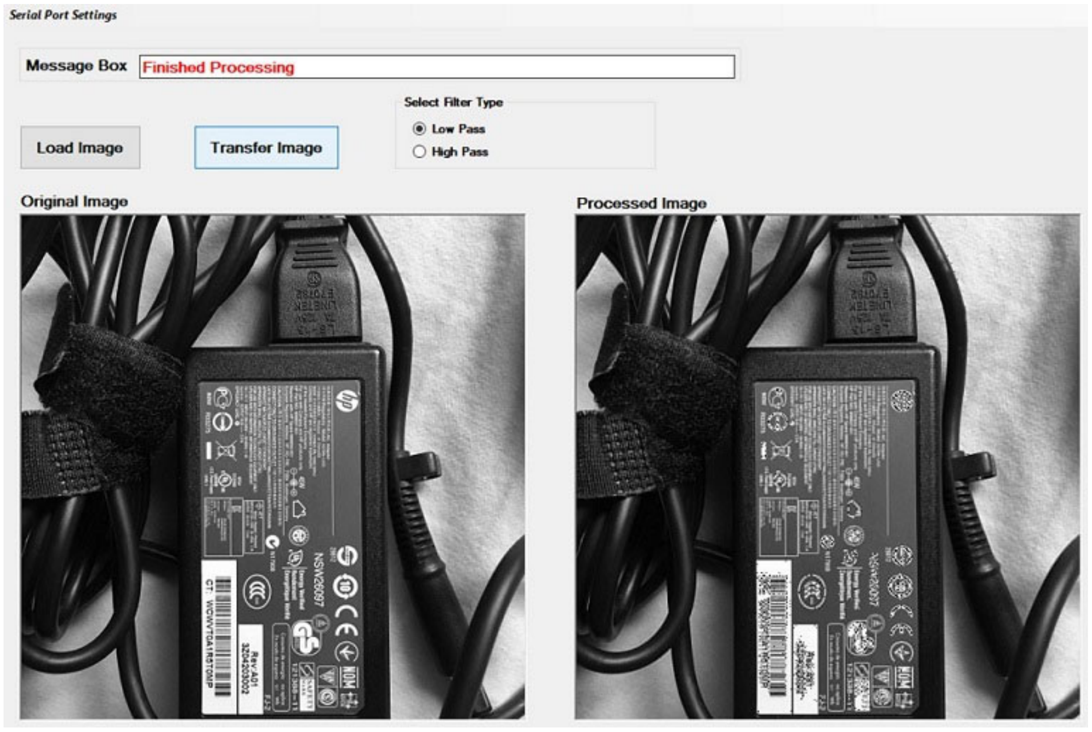


Serial Port Settings

Message Box: Finished Processing

Select Filter Type: [✓ Low Pass] High Pass

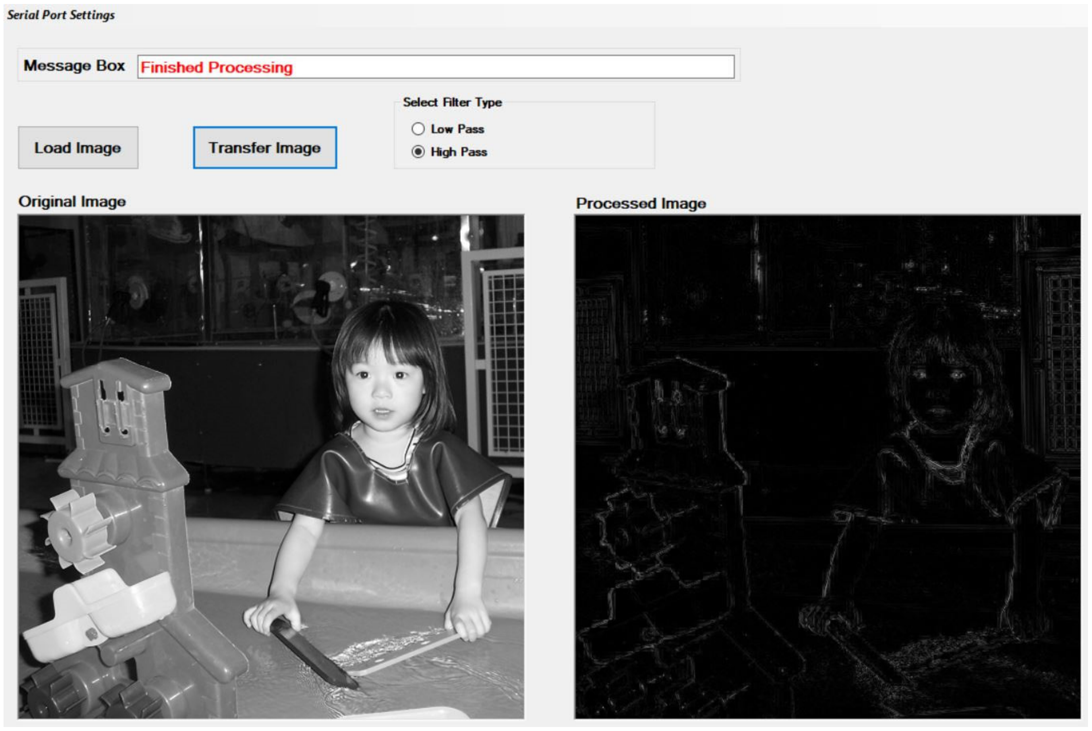
Fig 24. Kid image of 512×512 with lowpass filter


Serial Port Settings

Message Box: Finished Processing

Select Filter Type: [✓ Low Pass] High Pass

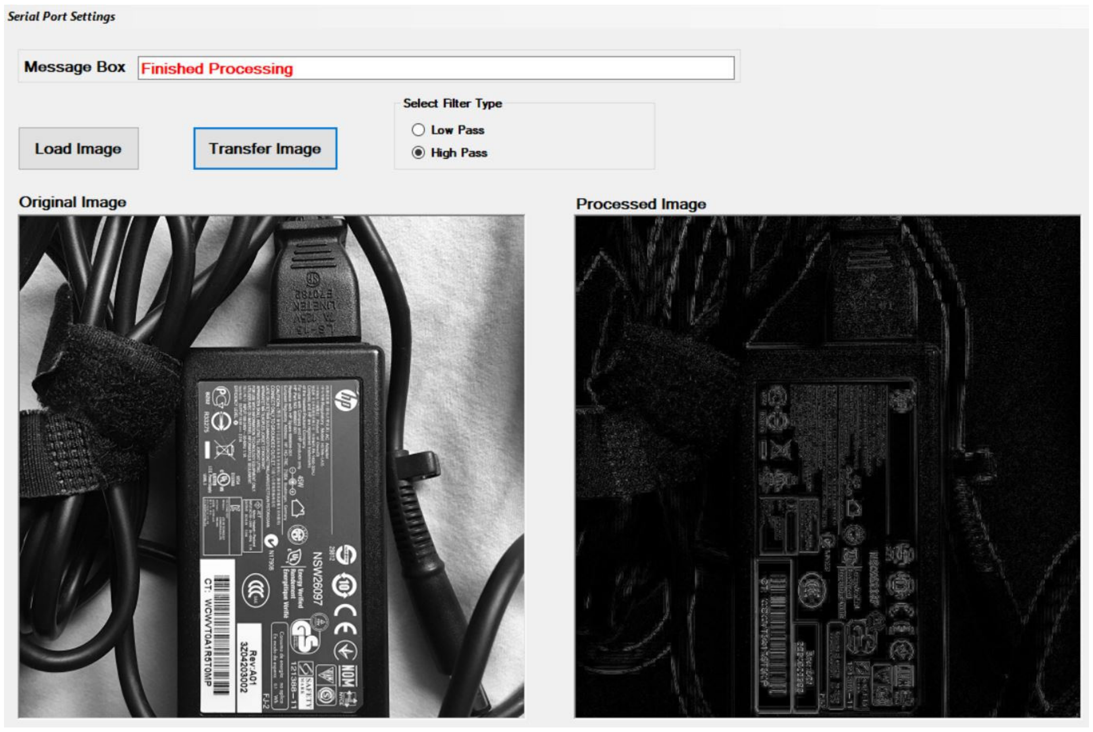
Fig 25. Power adapter image of 512×512 with lowpass filter


Serial Port Settings

Message Box: Finished Processing

Select Filter Type: Low Pass [✓ High Pass]

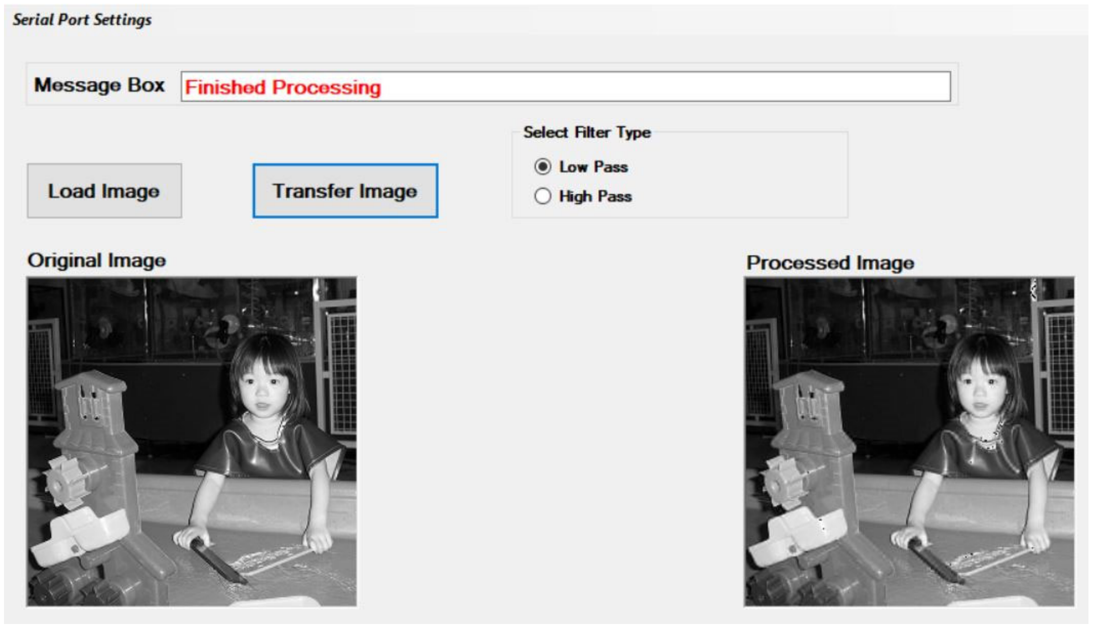
Fig 26. Kid image of 512×512 with highpass filter



Message Box: Finished Processing

Select Filter Type: Low Pass [✓ High Pass]

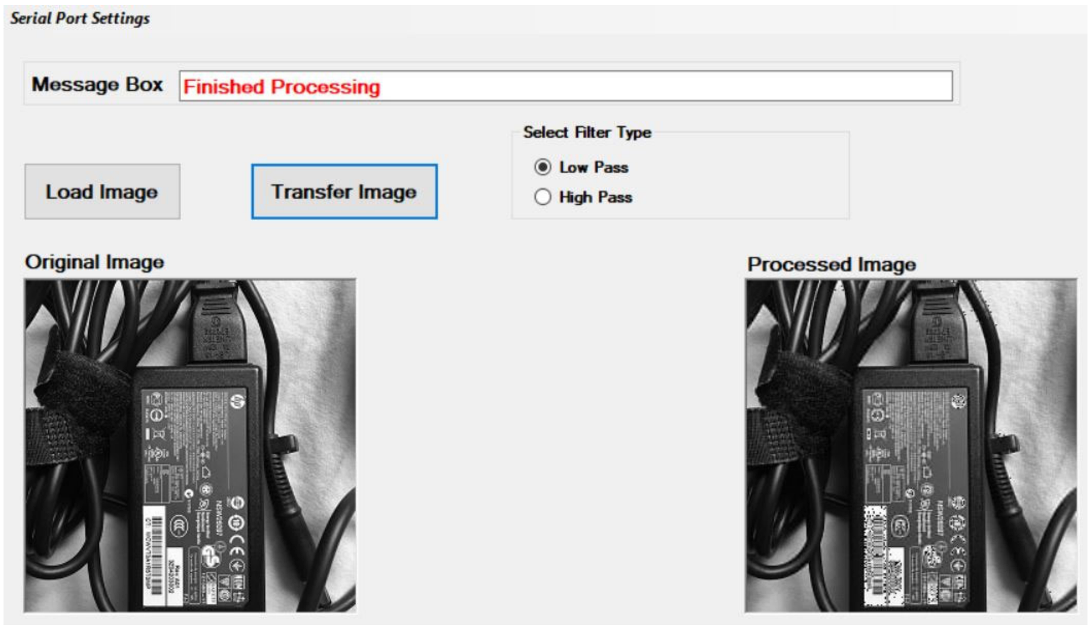
Fig 27. Power adapter image of 512×512 with highpass filter



Message Box: Finished Processing

Select Filter Type: [✓ Low Pass] High Pass

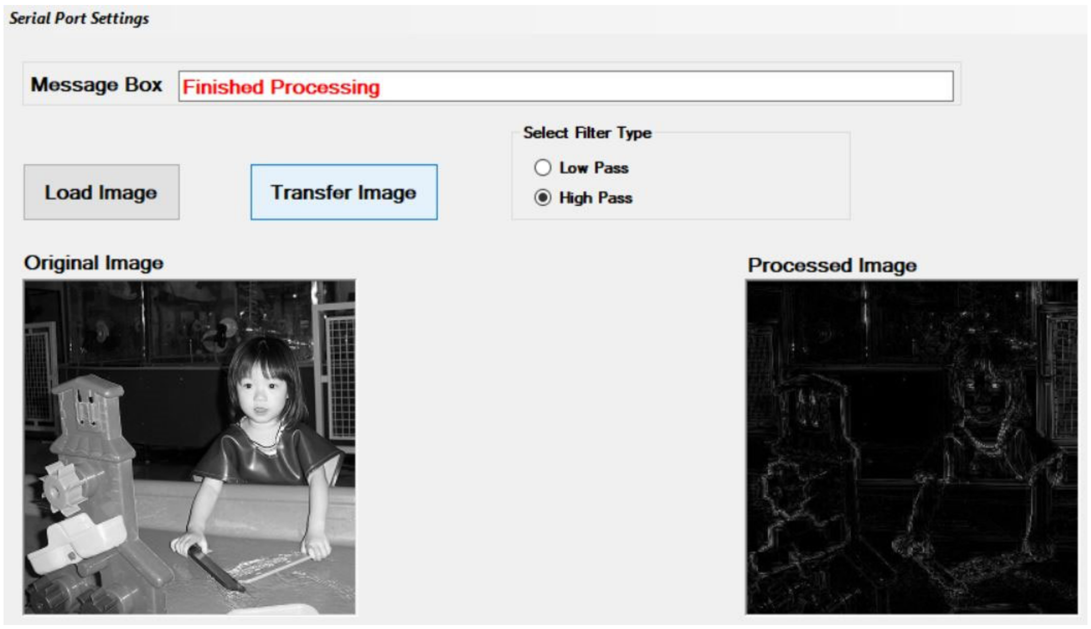
Fig 28. Kid image of 256×256 with lowpass filter



Message Box: Finished Processing

Select Filter Type: [✓ Low Pass] High Pass

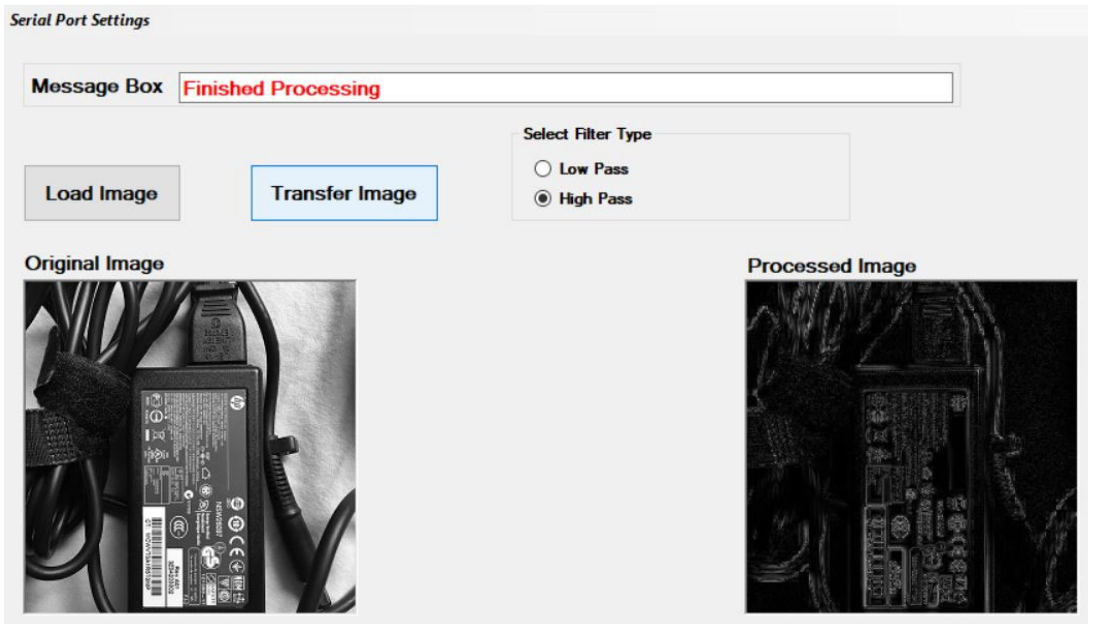
Fig 29. Power adapter image of 256×256 with lowpass filter



Message Box: Finished Processing

Select Filter Type: Low Pass [✓ High Pass]

Fig 30. Kid image of 256×256 with highpass filter



Message Box: Finished Processing

Select Filter Type: Low Pass [✓ High Pass]

Fig 31. Power adapter image of 256×256 with highpass filter