9

Implementation of Medical Image Processing Algorithms on FPGA Using Xilinx System Generator

Tijana I. Šušteršič[1,2] *and Nenad D. Filipovic*[1,2]

[1] *Faculty of Engineering, University of Kragujevac (FINK), Kragujevac, Serbia*
[2] *Bioengineering Research and Development Center (BioIRC), Kragujevac, Serbia*

9.1 Brief Introduction to FPGA

A field programmable gate array (FPGA) is a general-purpose integrated circuit that can be "programmed" by the designer, which means that it is not preprogrammed by device manufacturer. In comparison to the application-specific integrated circuit (ASIC), an FPGA can be programmed and reprogrammed, adapted to a specific problem, but later updated, even after it has been deployed into a system [1].

After the program design, it is converted into a bitstream configuration program and sent to static on-chip random-access memory. This bitstream represents the product of compilation tools that translate the high-level abstractions created by a designer into low-level and executable form. A FPGA offers a two-dimensional array of configurable resources that can execute a set of arithmetically and logically related features. These include DSP blocks, dual-port memories, multipliers, lookup tables (LUTs), buffers, registers, multiplexers, and digital clock managers. Xilinx FPGAs also feature specialized I/O structures that satisfy different voltage and bandwidth requirements [1].

For the purposes such as digital signal processing (DSP), it is of utmost importance to enable short delivery time with the possibility to modify the device function in the lab or the working site where the device is installed to suit the purpose [2]. Mostly due to the ever-decreasing cost and re-configurability, FPGAs have also found its place in DSP [3]. A lot of engineers give advantage to FPGAs over traditional digital signal processors (DSPs) because of the massive parallel processing capabilities and shorter time to market. Since FPGAs can be configured in hardware, they offer a complete hardware customization while implementing various

Computational Modeling and Simulation Examples in Bioengineering, First Edition. Edited by Nenad D. Filipovic.
Published 2022 by John Wiley & Sons, Inc.

DSP applications [3, 4]. DSP performance comes from the ability of FPGA to create incredibly parallel data processing architectures. Unlike the DSP processor or a microprocessor, where the performance is linked to the processor clock rate, FPGA performance is related to the amount of parallel that can be applied to the algorithms that constitute a signal processing system [1]. Therefore, it is suitable for implementation of many signal processing algorithms in the area of artificial neural networks [5], classification in medical diagnosis [6, 7] etc. The system designer is able to use parallel DSP with a combination of increasingly high clock rates (current system frequencies between 100 and 200 MHz are common today), along with the highly distributed memory architecture. For example, hundreds of terabytes per second would be the raw memory bandwidth of a massive FPGA with a clock rate of 150 MHz [1].

FPGAs have many advantages in applications that involve acquisition of digital data but also large-scale data processing, especially in real time [2]. Therefore, FPGAs are widely used to design applications that require high-speed parallel data processing, such as image processing [3]. Some of the examples of image processing purposes involve filtering [8], image compression [9], wireless communication [10], medical imaging [11], face recognition [12–14], target recognition, robotics application, video surveillance, etc., where high-speed parallel data processing is the main request. This is primarily due to the fact that FPGAs provide the parallelism that allows image processing in real time. The key downside of FPGAs is that it takes time to create a finished output and time to redesign the system if there are requests for that [3].

Another advantage of using the FPGA includes significantly lower nonrecurring engineering costs compared with the custom IC costs (FPGAs are off-shelf commercial equipment), shorter time on the market, and a more configurable FPGA, which makes it possible to modify the design in the final application, even after deployment [1]. As FPGAs continue to increase their chip density according to Moore's law, their potential applications have also expanded [3]. Additionally, FPGA boards are equipped with many I/O ports, AD/DA converters as well as a soft/hard core microprocessor, which allows these boards to behave as a system on chip (SoC). Furthermore, software/hardware codesigns using FPGAs are becoming very popular as a solution to certain complex problems, especially those that involve intensive computations [3].

For all the previously discussed reasons, Xilinx System Generator (XSG) brought the idea of compiling an FPGA program starting from a high-level Simulink model, rather than programming in a low-level language [1]. It is important to bear in mind, when working with System Generator, that an FPGA offers freedom in implementation of different functions for signal processing. There is a possibility, for example, to define data path widths throughout the designed system or to use a variety of individual data processors (e.g. multiply-accumulate engines), all

depending on the requirements and needs. On the other hand, System Generator provides the level of abstraction that allows the user to primarily focus on the algorithm and not the programming language.

9.1.1 Xilinx System Generator

XSG [1, 15] is a Xilinx tool that allows coupling with Mathworks Simulink models to be adapted for FPGA-design. For a variety of hardware operations, XSG provides a series of Simulink blocks for implementation. These blocks can be used to simulate the hardware system's functionality with Simulink. The design of most DSP applications includes a floating data point format [1]. Although this is relatively easy to incorporate models in many computer systems using applications such as Simulink, the difficulty of the application of floating-point arithmetic makes it more complicated in the hardware environment. For portable DSP systems, these problems increase where the restricting constraints of the system design. The XSG uses a fixed-point format to represent all numerical values. System Generator offers multiple blocks to convert the data supplied by the program (in our case Simulink) and the hardware (System Generator blocks) in the simulation environment.

The Simulink environment for higher level [16, 17] is expanded by Xilinx 's unique Blockset [3, 18]. For the use of a XSG, no previous experience with Xilinx FPGAs or RTL design methods is needed. The Matlab-Simulink high-level graphical interface makes it very easier to work with in comparison to the other software for hardware description [19]. The Xilinx-specific Simulink DSP Block contains more than 90 building blocks. These blocks include not just common components, such as adders, multipliers, and registers, but also complex building blocks including FFTs, filters, and memories. DSP blocks are also included. These blocks are used to generate optimized results for a chosen target system by Xilinx LogiCORE™ IP.

Downstream implementation of the Simulink environment for the generation of a *.bit* FPGA programming file are automated – synthesizing, placing, and route. In order to implement JTAG hardware co-simulation, I/O planning and Clock planning can also be performed. After this, a netlist is generated and a draft for the model and programming file in VHDL/Verilog is created. The module is checked for behavioral syntax check, synthesized, and then it can be implemented on FPGA. The XSG itself has also the possibility to generate User constraints file (UCF), Test bench, and Test vectors for testing architecture [3]. In order to optimize FPGA resources, System Generator enables some features such as System Resource Estimation, Hardware Co-Simulation [20], and accelerated simulation through hardware in the loop co-simulation, which can be used to increase the simulation performance [17].

As previously stated, the designs are made using Xilinx-specific Blockset using DSP-friendly Simulink modeling environment [18]. The XSG library includes over 90 DSP building blocks that allow faster prototyping and design from a high-level programming stand point [1, 3, 21]. There are also blocks such as the M code and Black Box, which allow the user to program in MATLAB M code or C code, and Verilog to simplify integration with already existing projects or customized block behavior [3, 21]. Figure 9.1 shows the design flow of hardware implementation of XSG model.

The prerequisite of using System Generator is to have compatible versions of Matlab and System Generator, which will be explained later, as well as that the System Generator token available along with Xilinx has to be configured in Matlab to be able to use the Blocksets [22].

System Generator can be useful for many purposes, out of which the main one could be:

- to test the algorithm to explore its possibilities without hardware translation;
- to use a System Generator design as part of a bigger design;
- to implement a complete design in FPGA hardware.

This part of the chapter summarizes all three possibilities [23].

Algorithm Exploration

System Generator is particularly suitable for exploring algorithms, designs prototyping, and model analyses. This is achieved with the aim to get a sense of design concepts, and to predict the cost and efficiency of the hardware implementation. The work is preparatory and the designs do not have to be translated to hardware. In this environment, key parts of the architecture are installed without caring about fine detail or information. MATLAB M-code and Simulink blocks are

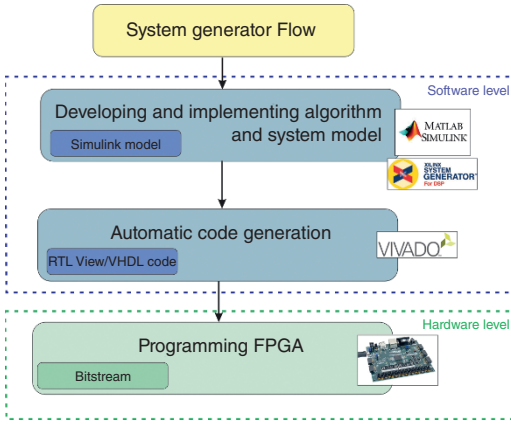

Figure 9.1 Design flow of Xilinx System Generator use in implementing an algorithm on hardware.

stimulants for simulation and outcomes analysis. A rough understanding of the cost of hardware production is provided by the resource estimate [1, 23].

Hardware generation tests may indicate hardware speeds that are possible. System Generator encourages step-by-step changes to be done, meaning that some of the architecture can be ready for hardware execution, while others remain abstract and high-level. System Generator hardware co-simulation facilities are particularly beneficial for optimizing portions of a design [1, 23].

Implementing Part of a Larger Design

System Generator is also used to build a design that is only a component of a bigger design. System Generator, for example, is an excellent environment for introducing data paths and testing, but not ideal for complex and strict timing requirements. It might be useful in this case to incorporate design parts via System Generator, to deploy other components externally and then to merge the components in a greater design.

A common solution to this flow is to construct a full HDL wrapper that represents an entire design and to use the System Generator part as a component. The parts of the architecture of the non-System Generator may also be components in the wrapper or can be instantiated directly in the wrapper [1, 23].

Implementing a Complete Design

Most of the time, all the necessary components for a design are available inside System Generator. To build this design, pressing the Generate button and the System Generator will instruct to convert the design into HDL and write the files necessary to process the HDL using downstream tools. The necessary files are:

- HDL that implements the design itself.
- A clock wrapper that encloses the design.
- An HDL testbench that encloses the clock wrapper. The testbench allows comparison of the outcomes of the Simulink simulations with those of a logic simulator.
- Files and scripts of projects that allow different synthesis tools like XST and Symplify.
- Files that enable System Generator HDL to be used as a project in the Project Navigator [1, 23].

9.1.2 Image Processing on FPGAs Using XSG

A variety of methods are used in different strategies for image processing. Some of them such as acquisition, enhancement, restoration, segmentation, and analysis are usually part of every image processing algorithm [9]. The images, once they are inside the computer system, are represented by the matrices and theoretically,

all operations applied on matrices, can be applied on images too – addition, subtraction, multiplication, and division [3]. General processing techniques include removing the noise and performing enhancement [11], transformations to extract features for pattern recognition, compressing for storage and retrieval, or transmission via a computer network or a communication system [3].

Mohammed et al. studied the processing of digital image, especially point processing algorithms [3]. In the co-simulation of both software and hardware, Hanisha et al. examined the image denoising algorithm by using XSG [24]. Using Matlab-Simulink coupled with XSG tools, Kumar et al. examined various image processing operations to transform the image into suitable form for further processing on FPGAs [25]. A standard image processing pipeline was used: file reading, image preprocessing, main image processing techniques, post-processing, and visual output representation [25].

Back to the processing of images in real time, it is impractical and also time consuming to write thousands of lines in low-level language. Therefore, software environment for hardware description [1, 2, 21] is created to be supported by a tool XSG [2] coupled with a Simulink graphical interface in Matlab. While software implementations of various image processing techniques are appropriate for general use, they must be incorporated in hardware in order to satisfy real-time applications [3]. Moreover, processing speed of software versions is not sufficient when it comes to meeting the real-time needs, because a significant amount of information has to be processed in a limited period of time (meaning that throughput is higher) [26]. In comparison to their corresponding software, FPGA-based application-specific hardware design offers higher processing speeds, with the benefits of parallel architecture [26].

Some research on implementation of different image processing algorithms on FPGA using the XSG tool have already been carried out [27]. Christe et al. proposed FPGA architecture for MRI image analysis and tumor characterization. Experimental results for different filtering algorithms are presented in this work using the XSG [16]. It was shown that with a chip XC3S500E-FG320, only 50% of the total resources were used. Que et al. performed the reconstruction of the image from computer tomography and its FPGA-based hardware implementation [28]. This algorithm is implemented on Virtex-2 processor using both Simulink and XSG block for both hardware/software.

Sudeep and Majumdar investigated FPGA-based image edge detection architecture [29]. Sobel, Prewitt, and Robert algorithms are implemented in this architecture using XSG on a Spartan-3A FPGA device. Harinarayan et al. researched feature extraction of digital aerial images on FPGA-based architecture [30]. The Virtex-4 processor is used to implement XSG blocks. Edge detection methods such as Sobel and Prewitt are used to perform feature extraction.

Shanshan and Xiaohong introduced the FPGA image edge detection architecture using the XSG tool [31]. This architecture was mainly used to identify the edge features of the images of the vehicle. This system is implemented on Spartan-3E Board and the summary of resource utilization is also reported. Basu et al. outlined the design and deployment of a real-time DSP program using coupled Matlab-Simulink and XSG [32]. The configuration is implemented on the Spartan-3A Board. The implementation of the FPGA image processing system with the help of XSG is also investigated by Said et al. [33]. This design implements Sobel algorithm for edge detection. The design was implemented on the Spartan 3A and Virtex-5 FPGA devices and their utilization summary is compared. FPGA-based implementation of conventional edge detectors such as Sobel, Prewitt, Robert, and Laplacian of Gaussian (LoG) are already investigated for implementation using XSG [29, 33]. Since XSG-based edge detection based on Simpler Gabor Wavelet hardware implementation was not investigated up to that point, Sujatha et al. implemented it on FPGA and compared with other edge detection methods [27].

Although some of the mentioned algorithms were already implemented on FPGA using XSG, very rarely have the papers investigated the application on medical images. Among those, Šušteršič et al. have studied the use of XSG in the medical image processing [34] and this chapter represents the extension of that work. Therefore, it is very interesting to see how these algorithms are behaving in the case of implementation on different medical images. All this leads to the motivation of this chapter to implement several image processing algorithms as part of a complex medical and biomedical applications using Matlab/Simulink coupled with XSG tool. In this chapter, Vivado System Generator 2019.1 was used, and Matlab version was 2018b; however, the requirements and steps necessary to build any model, independent of the versions are also described in Table 9.2.

9.2 Building a Simple Model Using XSG

System Generator allows device-specific hardware designs to be built directly in a flexible, high-level system modeling environment. Signals in System Generator can be in fixed-point format (signed or unsigned) and transformed into appropriate signal types. Blocks react to their surroundings, automatically adjusting the results they produce to the chosen hardware [1].

Data flow models, traditional hardware design languages (i.e. VHDL, Verilog), and MATLAB programming language functions can be used alongside, simulated, and synthesized together to a working hardware. The results of the System Generator simulation are bit and cycle-accurate, which ensures that the effects shown

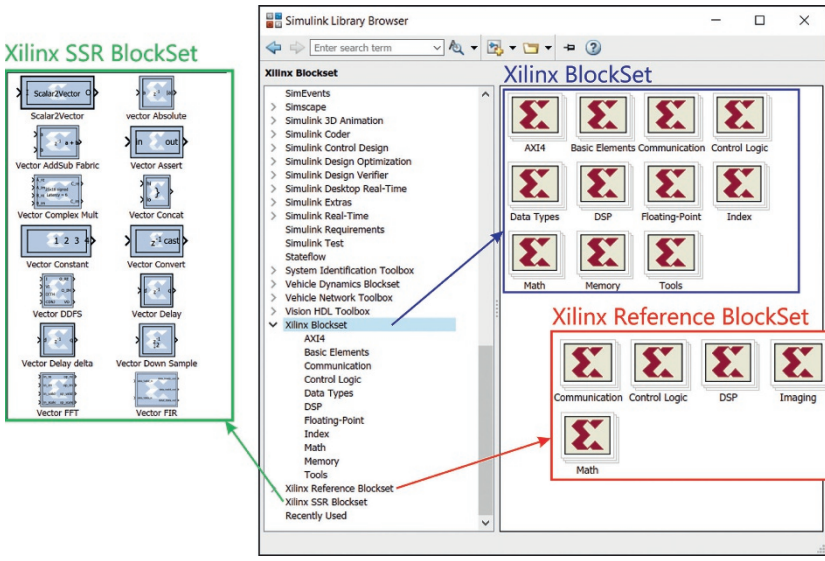

Figure 9.2 Xilinx System Generator library. *Source:* The MathWorks, Inc.

in the simulation precisely match the results found on the hardware. System Generator simulations are much faster to produce rather than conventional HDL simulators, and results are simpler to interpret [1].

The Simulink Xilinx Library is a library of blocks that can be connected to the Simulink block editor to construct fully functional systems (Figure 9.2). System Generator blocksets are in that sense used as any other Simulink blocksets. The blocks include abstractions of logical, mathematical, memory, and DSP functions that can be used to construct complex signal processing systems. There are also blocks that provide interfaces to other tools (i.e. FDATool, ModelSim) as well as code generation from System Generator [1].

The Xilinx Blockset represents a family of libraries that include basic System Generator blocks. Some of the blocks are low-level and provide the access to device-specific hardware. Other blocks are more high-level and their purpose is to implement advanced designs (i.e. signal processing). Those blocks that have the wide application (i.e. I/O blocks) belong to several libraries. Every block is part of the Index library given in Table 9.1.

Prerequisites

In order to be able to build any model using XSG, Matlab and Vivado versions need to be compatible. A full list of supported versions is given in Table 9.2 [22].

Table 9.1 Xilinx Blockset library description [1].

Library Name	Description
Index	Every block in the Xilinx Blockset
Basic elements	Elements Standard building blocks for digital logic
Communication	Forward error correction and modulator blocks, commonly used in digital communications systems
Control Logic	Blocks for control circuitry and state machines
Data Types	Blocks that convert data types (includes gateways)
DSP	Digital signal processing (DSP) blocks
Math	Blocks that implement mathematical functions
Memory	Blocks that implement and access memories
Shared Memory	Blocks that implement and access Xilinx shared memories
Tools	"Utility" blocks, e.g. code generation (System Generator block), resource estimation, HDL co-simulation, etc.

Table 9.2 Supported Matlab and XSG versions.

System Generator version	Supported Matlab release
2020.1	R2020a, R2019b, R2019a
2019.2	R2019b, R2019a, R2018b, R2018a
2019.1	R2019a, R2018b, R2018a
2018.3	R2018a, R2017b, R2017a
2018.2	R2018a, R2017b, R2017a
2018.1	R2017b, R2017a
2017.4	R2017b, R2017a, R2016b, R2016a
2017.3	R2017a, R2016b, R2016a
2017.2	R2017a, R2016b, R2016a
2017.1	R2017a, R2016b, R2016a
2016.4	R2016b, R2016a, R2015b, R2015a
2016.3	R2016b, R2016a, R2015b, R2015a
2016.2	R2015b, R2015a, R2014b, R2014a
2016.1	R2015b, R2015a, R2014b, R2014a

In order to configure MATLAB to the Vivado® Design Suite, the following steps should be performed:

Windows system

- Select Start > All Programs > Xilinx Design Tools > Vivado 2019.x > System Generator > System Generator 2019.x MATLAB Configurator
- Click the check box of the version of MATLAB compatible for configuration and then click OK

Linux systems

- Launching System Generator under Linux is handled using a shell script called *sysgen* located in the *<Vivado install dir>/bin* directory. Before launching this script, MATLAB executable must be found in Linux system's *$PATH* environment variable for Linux system. When *sysgen* script is executed, it will launch the first MATLAB executable found in *$PATH* and attach System Generator to that session of MATLAB. Also, the *sysgen* shell script supports all the options that MATLAB supports and all options can be passed as command line arguments to the *sysgen* script [23].

System Generator can be started using Start > All Programs >Xilinx Design Tools > Vivado 2019.x > System Generator. In the Simulink library browser, there is a list of all different Toolboxes installed within MATLAB, among which the XSG components will be shown and divided into three subcategories:

- Xilinx Blockset
- Xilinx Reference Blockset
- Xilinx SSR Blockset

The category Xilinx Blockset contains all the basic blocks, which can be used in many different applications, out of which some will be shown in this chapter. As in general Simulink, a new model can be created via File> New> Model.

Basic XSG blocks, used in almost any application are explained below, alongside with their icons given in Figure 9.3.

- **Xilinx System Generator Block**: One of the most important blocks used in any System Generator model is the System Generator Block (Figure 9.3a). This block is found in the Simulink category: Xilinx Blockset>Basic Elements>System Generator. The System Generator block in the new model window should be placed as shown in Figure 9.20.
- **Input/Output Gateways**: These blocks belong to System Generator blocks that are used to convert data inside Simulink in the floating-point format to fixed-point format that is used inside the hardware system and modeled by System Generator (Figure 9.3b). In the end of processing, output gateway is used again

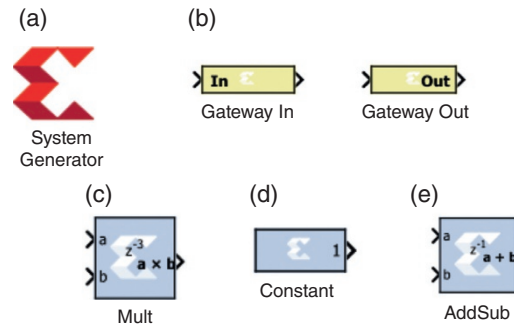

Figure 9.3 Schematic icons for some of the widely used XSG elements. (a) Xilinx System Generator Block, (b) Input/Output Gateways, (c) Multipliers, (d) Constants, (e) Adder/Subtractor

to convert the system output back to floating point. In that sense, there are two types of gateways inside System Generator:

- **Input Gateways**: are used to convert floating point data into fixed-point format. Input gateway properties define in detail the fixed-point format with its three important properties:
  - Output type: the values that can be assigned to this property are *boolean* (single bit data representation), *two's complement* data representation, and *unsigned* data representation.
  - Number of bits: Number of bits that are used to represent the data. The resolution of the system is directly defined by the number of bits.
  - Binary point: represents the position of the binary point in fixed-point format.
- **Output Gateways**: are used to convert the fixed-point format back into the floating-point format. Output gateways are able to automatically detect the fixed-point format from the system output and therefore do not require any other modification.

- **Multipliers**: Most of the models require some multiplication. XSG Blockset provides several blocks for arithmetic operations, among which multiplication is one of them (Figure 9.3c). Multiplier block can be placed using Xilinx Blockset>Math>Multblock. Double click on the Multblock will enable the change of its properties. Two basic options for output precision are to be found in its basic properties: *full* and *user defined*. In the *full precision* option, the multiplier block uses the input fixed-point format to determine the format of the output. In the *user-defined* precision mode, the designer specifies a different format. This means that the designer needs to specify a rounding method for excessive data values. Additionally, in the Implementation Menu, the user has the possibility to optimize speed/area using the dedicated Multipliers embedded in an FPGA or use the LUTs instead.

- **Constants**: Constants are necessary part of every design. Constants are usually implemented using hardwired configurations. They can be found in Xilinx Blockset>Basic Elements>Constant (Figure 9.3d). Double click on the constant block will enable the change of constant properties. If fixed-point (Signed (2's comp)) is used, the user should pay attention to have enough bits to represent the number (i.e. number of bits 16 would allow to represent fraction numbers between the +7 and −8).
- **Adder/Subtractor**: The adders/subtractors are almost a necessity in every model. These blocks can be found in Xilinx Blockset>Math>AddSub (Figure 9.3e). The AddSub block performs addition or subtraction operations using two operands. The fixed-point format of the output is determined based on the format of the inputs.

More detailed explanation on building a simple model and Hardware/Software Co-Simulation is given in [1, 23]. In this chapter, we will focus on implementation of medical image processing algorithms using XSG.

9.3 Medical Image Processing Using XSG

9.3.1 Image Pre- and Post-Processing

In software co-simulation, all other Xilinx blocks are integrated between two blocks – Gateway In and Gateway Out, which serve as input and output for the H/W design, respectively [35]. As Xilinx blocks operate in a fixed-point format, there is a necessity to translate real signals, written with a floating point, to a fixed point, and vice versa. For that purpose, Gateway In and Gateway Out blocks act as translators for conversion of signals into different formats [2, 21, 35].

For every image processing design, blocks like Image Source and Image Viewer will be used to read the source file and later to view the output. Preprocessing and post-processing image units are the same independently of the image processing purpose and are constructed using traditional Simulink block sets [3, 24, 34]. Therefore, preprocessing subsystem should consist of a variety of mentioned blocks – *File image, Transpose, Transform 2D to 1D, Frame translation, and Unbuffer* (Figure 9.4).

Since most MRI/CT images are grayscale images and therefore represented as M-by-N arrays, where the number of rows and the number of columns are M and N, respectively, *Image from file* block is used to read the image of interest as matrix [25]. In case an image is color image, its representation is M-by-N-by-P array, where M and N are rows and columns, respectively, on each color plane. Block *Transpose* performs the mathematic transpose operation of an M-by-N

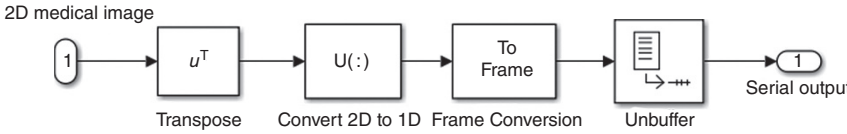

Figure 9.4 Image preprocessing module.

matrix to obtain the output N-by-M matrix. Block *Convert 2D to 1D* serves to transform a 2D image matrix to a 1D sequence as the name itself reads. *Frame conversion* gives two options: setting the sampling mode to frame based or sample based and helping prepare the data to unbuffer. *Unbuffer* block performs unbuffering of M-by-N input into a 1-by-N output, where unbuffering is performed row wise [34].

Image post-processing subsystem is again made up of several blocks — *Data type conversion, Buffer, Convert 1D to 2D, Transpose, and Video viewer* (Figure 9.5).

These blocks help the image to be reconstructed based on the obtained 1D array. *Data type conversion* serves to convert the signal to multiple formats, where unsigned integer is used. *Buffer* Block implements frame-based processing by redistributing the data in each column input to generate an output of a different frame size. The buffering of a signal with a greater frame size would result in an output that would be slower than the input frame rate [25]. *Block Convert 1D to 2D* is used to transform the 1D array back to the 2D image matrix to be able to view the image on the screen (using *Video Viewer* block) [34].

The next section of the chapter performs implementation of several algorithms for image processing on medical images, which can serve as a guidance to more complex analyses of images applied in biomedical fields.

9.3.2 Algorithms for Image Preprocessing

9.3.2.1 Algorithm for Negative Image

Transforming an image into negative has many applications in medical imaging, where inverted image is used for further processing. Inverted image can be obtained in several ways, among which an XOR function block/simple Inverter block/Add sub block are just some ways. The algorithm for obtaining grayscale image negative using XOR is given in Figure 9.6.

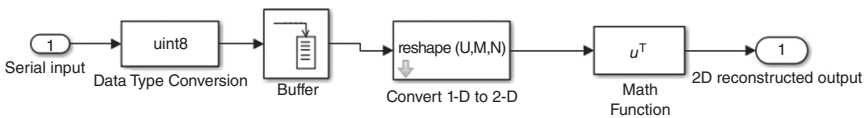

Figure 9.5 Image post-processing module.

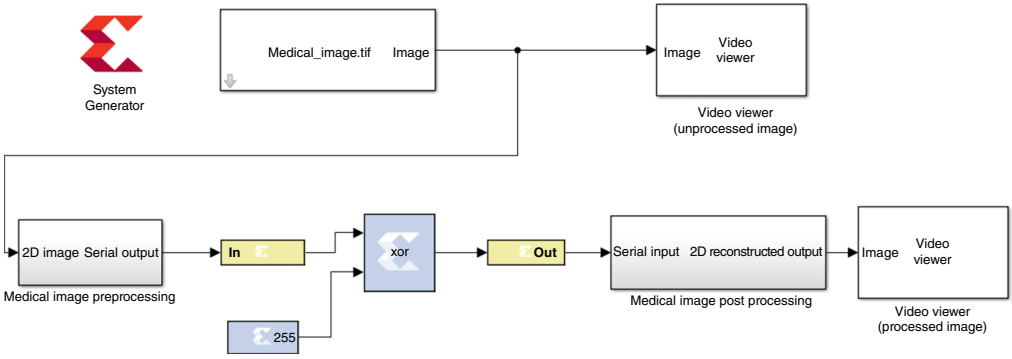

Figure 9.6 Simulink – System Generator DSP model for creating negative image.

Other ways to obtain negative image is using Addsub block by subtracting the constant 255 from each pixel value to generate finally the negation of an input image. The same result would be obtained when an Inverter block is used.

9.3.2.2 Algorithm for Image Contrast Stretching

Contrast of an image is the distribution of light and dark pixels in an image. To enhance the contrast on an image, histogram should be stretched on a range between 0 and 255, meaning to fill the full dynamic range of the image. However, if there is at least one pixel with value 0 or 255, contrast stretching does not make sense. The new pixel value is calculated using the equation [36]:

$$g(x,y) = \frac{f(x,y) - f_{\min}}{f_{\max} - f_{\min}} \bullet \left(2^{\text{bpp}} - 1\right) \tag{9.1}$$

This formula requires finding the minimum $f_{\min}$ and maximum $f_{\max}$ pixel intensity multiplied by levels of gray. In our case, the image is 8 bit, so levels of gray are $2^8 = 256$. In the case of the used image with sagittal view of the spine disc, $f_{\min} = 44$ and $f_{\max} = 255$, obtaining the Subsystem for image enhancement as in Figure 9.7. In the case of the used image with lungs infected with COVID-19, $f_{\min} = 20$ and $f_{\max} = 209$; therefore, the constant and multiplier values are 44 and 1.349, respectively.

When visualized in the histogram, that is equivalent to expanding or compressing the histogram around the midpoint value [21, 25].

9.3.2.3 Image Edge Detection

The edge of the image is defined as a point where there is a sharp difference in the image, in the sense of where pixel positions have a sudden shift in brightness, i.e. a discontinuity in gray level values. For this purpose, edge detection in image processing means recognizing and localizing regions in which such brightness transitions occur [21, 35, 37]. Commonly, edge detection techniques are applied using software, but with the advent of Very Large Scale Integration (VLSI) technologies, the hardware implementation of edge detectors has played an important role in real-time applications [35].

Gradient-based methods are often used for the identification of edges. They search for the edge by finding the maximum and the minimum in the first derivative of the image. The key edge detection operator is the gradient operation of the matrix field, which defines the variance between different pixels. The edge detection operator is then determined by constructing a matrix centered around the pixel that is chosen as the center of the matrix area. If the value of this matrix region is above the threshold, the middle pixel is treated as the edge [38]. In the sense of mathematical operations, edge detection in image processing means

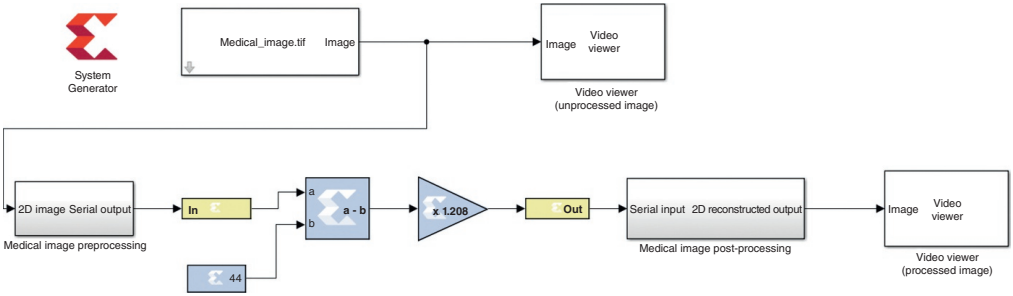

Figure 9.7 Simulink – System Generator DSP model for image contrast stretching.

a masking operation with an acceptable filter mask, meaning the input image is convolved with some of aforementioned filter masks [38].

The use of edge detection algorithms reduces the amount of details that need to be processed and clears information that are less relevant without modifying any of the image properties [26]. As a consequence, several researchers have researched edge detection algorithms for a number of purposes. Fuad et al. published a comparative survey of the edge detection techniques and implemented Canny Edge Detection Algorithm using the XSG on the Nexys3 Board [35]. Kabir et al. have researched another algorithm for the identification of edges – Sobel edge detection in hardware [26]. Although various edge detection methods are available as software implementations such as Roberts, Prewitt, and Sobel method, software versions are suitable for general use. According to the motivation of this chapter, we further study several gradient-based edge detection methods and implement them on medical images in hardware using the XSG.

For a given image I, in the process of calculating horizontal and vertical gradient of each pixel *I(x, y)*, Eqs. (9.2) and (9.3) are applied [33, 34]:

$$G_x(x,y) = \text{horizontal\_mask} * I(x,y) \tag{9.2}$$

$$G_y(x,y) = \text{vertical\_mask} * I(x,y) \tag{9.3}$$

In each pixel (*x,y*), the horizontal and vertical gradients are combined as in Eq. (9.4), in order to obtain the gradient value:

$$G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \tag{9.4}$$

Approximation of the magnitude is calculated using the formula:

$$G(x,y) = |G_x(x,y)| + |G_y(x,y)| \tag{9.5}$$

where $G_x$and G_y are the gradients in the x- and y-directions, respectively. The orientation of the gradient is calculated using the formula:

$$\theta(x,y) = \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right) \tag{9.6}$$

In many real-world implementations, the Canny approach is dominant due to its strong localization efficiency and the ability to remove major edges very well. However, the Canny algorithm consists of comprehensive preprocessing (i.e. smoothing) and post-processing (i.e. Non-Maximum Suppression [NMS]). It is also more computationally complex than other gradient-based edge detection algorithms such as Roberts, Prewitt, and Sobel [35, 39, 40]. As a result, we will concentrate on several gradient-based edge detection methods and compare their efficiency, resource utilization, performance time, etc.

Robert method

The basis for Robert method is the use of the Robert operator [38]. It is a type of differential operator that approximates the gradient of an image using a discrete differentiation that calculates the sum of the squares of the differences between the diagonally adjacent pixels of two 2×2 kernels (masks) given by Eq. (9.7) [2, 34].

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{9.7}$$

Masks are separately applied to the input image in order to generate different measurements of the gradient component in the x and y directions (G_x and G_y). Local gradients use convoluted mask coefficients for each pixel. If we define G_x and G_y as horizontal and vertical gradient filter masks, the gradient calculation formulas in both directions are [38, 41] as follows:

$$| G_x | = | p1 - p4 | \text{ and } | G_y | = | p2 - p3 | \tag{9.8}$$

They are then combined together to find the absolute magnitude of the gradient at each pixel using Eq. (9.5). The model for edge detection using the XSG with implemented Robert method, separately shown for horizontal and vertical calculations is given in Figures 9.8 and 9.9, respectively. The masks are than added and forwarded to the thresholding subsystem (explained later in this chapter).

Prewitt method

The basis for Prewitt method is that the operator uses the central difference. This method is considered to be better in certain aspects than Roberts. Image is convolved with the kernel in horizontal and vertical direction with the kernels given by Eq. (9.9) [2, 34].

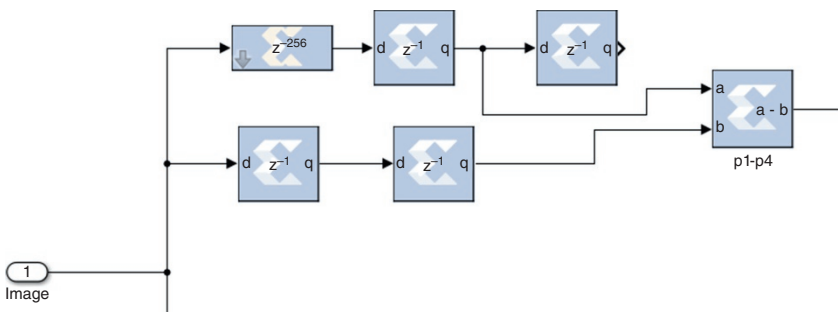

Figure 9.8 Robert method – System Generator DSP sub-model for horizontal mask in edge detection.

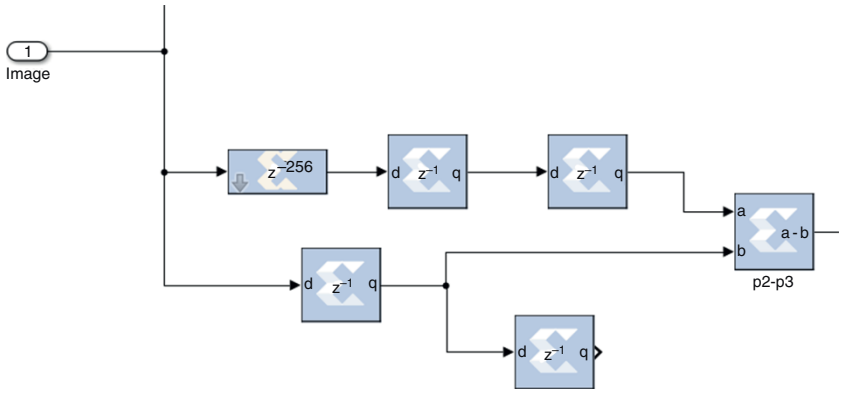

Figure 9.9 Robert method - System Generator DSP sub-model for vertical mask in edge detection.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{9.9}$$

Prewitt edge detection uses moving masks over the image to calculate the gradient of the image. If we define G_x and G_y as horizontal and vertical gradient filter masks, equations for gradient measurement are as Eq. (9.10) [38, 41]:

$$\begin{aligned} |G_x| &= |(p3 + p6 + p9) - (p1 + p4 + p7)| \\ |G_y| &= |(p1 + p2 + p3) - (p7 + p8 + p9)| \end{aligned} \tag{9.10}$$

Prewitt 's method is less susceptible to noise due to its kernel [2]. The methodology for the identification of edges using the XSG with implemented Prewitt method is given in Figures 9.10 and 9.11.

Sobel method

The Sobel operator is similar to the Prewitt operator, and has a basis in central difference, but larger weights are added to the central pixels and the mask is therefore as Eq. (9.11) [2, 34]:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{9.11}$$

These kernels are generated to adapt as much as possible to edges that are vertical and horizontal relatively to the pixel grid, where one kernel is used for each direction x and y. These kernels can be used separately on the input image to allow

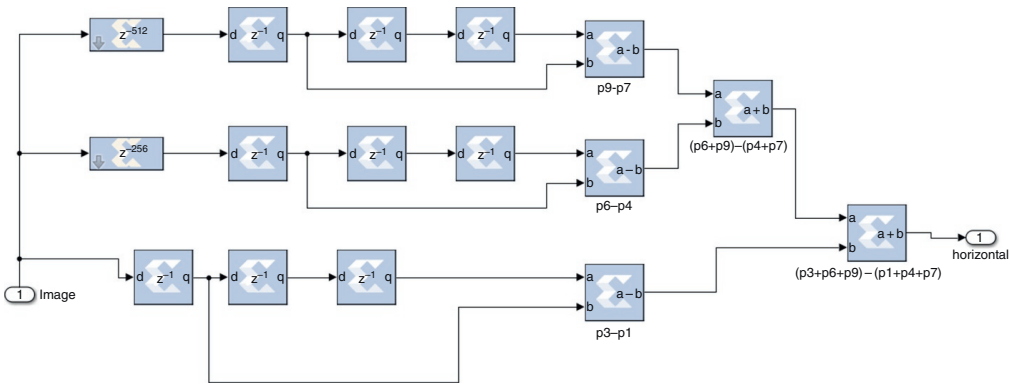

Figure 9.10 Prewitt method – System Generator DSP sub-model for horizontal mask in edge detection.

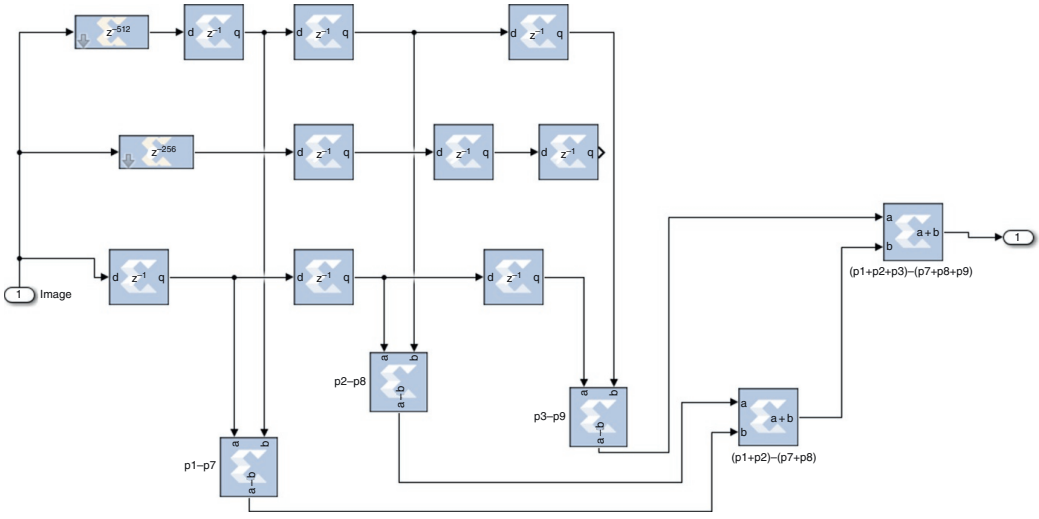

Figure 9.11 Prewitt method – System Generator DSP sub-model for vertical mask in edge detection.

for independent calculations of the gradient components in each direction [38]. Similarly as other methods, moving mask in Sobel edge detection goes over the image and calculates the gradient of the image. If G_x and G_y are defined as masks for horizontal and vertical gradient kernels, gradient calculation equations are given as [38, 41]:

$$\begin{aligned} |\, G_x \,| &= |\, (p3 + 2{*}p6 + p9) - (\, p1 + 2{*}p4 + p7\,) \,| \\ |\, G_y \,| &= |\, (p1 + 2{*}p2 + p3) - (p7 + 2{*}p8 + p9) \,| \end{aligned} \tag{9.12}$$

The Sobel method improves noise reduction and is one of the simpler operators with very good results [33]. The methodology for the detection of edges using the XSG with implemented Sobel algorithm is given in Figures 9.12 and 9.13.

Thresholding method

Once the edge detection algorithm is finished, it is important to evaluate the strength of the edges, since some of the detected edges may not be the actual edges. We will achieve this by using thresholding. Thresholding is performed in such a way that each pixel with a value greater than the threshold value is marked as a strong edge and the edge pixels with the values smaller than the threshold are discarded, which means that the following equation is applied [25, 34, 35]:

$$F(x,y) = \begin{cases} 255, & \text{if } f(x,y) > T \\ 0, & \text{otherwise} \end{cases} \tag{9.13}$$

where T is the threshold assigned. In the context of application, Mux is used to compare and replace the pixels with the new values. The explained methodology using XSG is given in Figure 9.14.

Canny method

Canny method is one of the more accurate edge detection methods among edge detection techniques based on intensity gradients [42]. The Canny Edge Detection method is divided into several sub-steps:

- Gaussian smoothing: noise is eliminated using Gaussian filter over the input image (using a Gaussian discrete kernel).
- Calculate Intensity Gradients: identifies the image areas with the strongest intensity gradients (using a kernel of Sobel, Prewitt, or Robert).
- Non-Maximum Suppression: NMS is applied to thin out the edges. We are attempting to delete unnecessary pixels that may not be part of the edges.
- Thresholding with Hysteresis: Hysteresis or double thresholding involves accepting the pixels as edges if the intensity gradient magnitude overcomes the upper threshold. The pixels are rejected as edges, if the intensity gradient value is below the lower threshold [42].

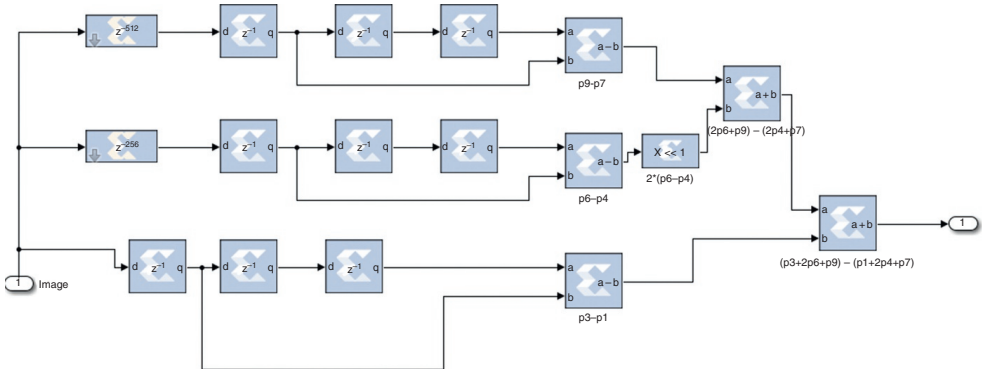

Figure 9.12 Sobel method – System Generator DSP sub-model for horizontal mask in edge detection.

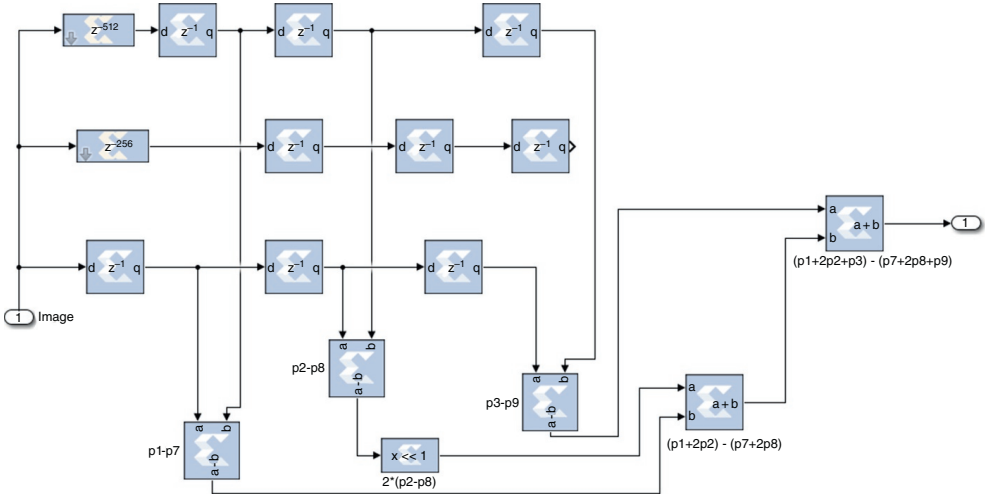

Figure 9.13 Sobel method – System Generator DSP sub-model for vertical mask in edge detection.

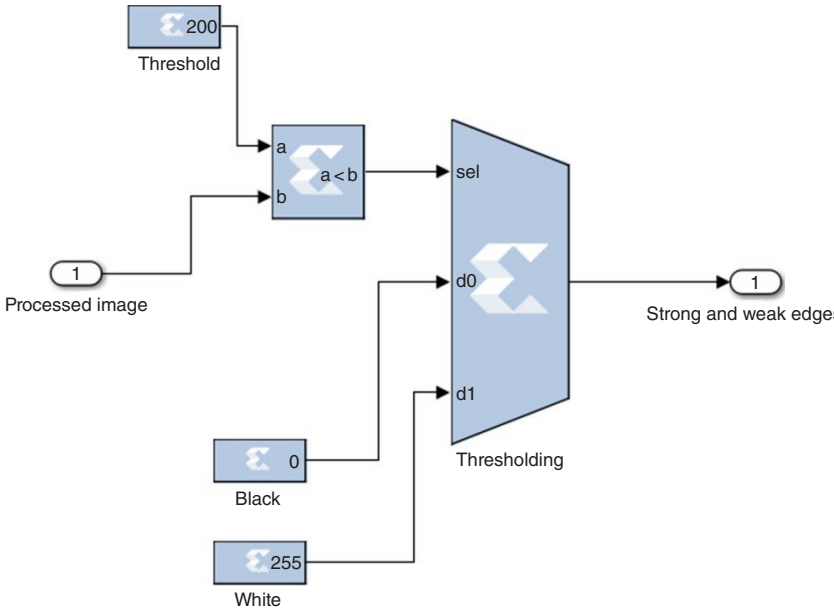

Figure 9.14 System Generator DSP sub-model for thresholding method.

The methodology for the detection of edges using the XSG with implemented Canny algorithm is given in Figure 9.15. Different subsystems are then detailed in Figures 9.16–9.19.

Filter is applied on each of the 5×5 regions in the image and convolved with that region (with the pixel of interest as the center cell called anchor). The result of the convolution is the new intensity value for all the pixels. The block in XSG that performs this operation has a total of nine applications, out of which we choose Gaussian filtering. In that sense, it acts as a low-pass filter and removes unwanted high-frequency elements – noise [42].

After removing the noise using the Gaussian smoothing, the next step is to calculate the intensity gradients. Any of the discussed kernels – Robert, Prewitt, or Sobel – can be used, but usually the adopted kernel is Sobel. We would not give the detailed description of Sobel horizontal and vertical filters, as they were already explained in this chapter. In order to calculate the resultant magnitude, square root operation is necessary. However, the cost to realize the square root operation in hardware is high, therefore low-cost hardware implementation equation based on square approximation is adopted to compute the magnitude [25]. This formula is given in Eq. (9.5).

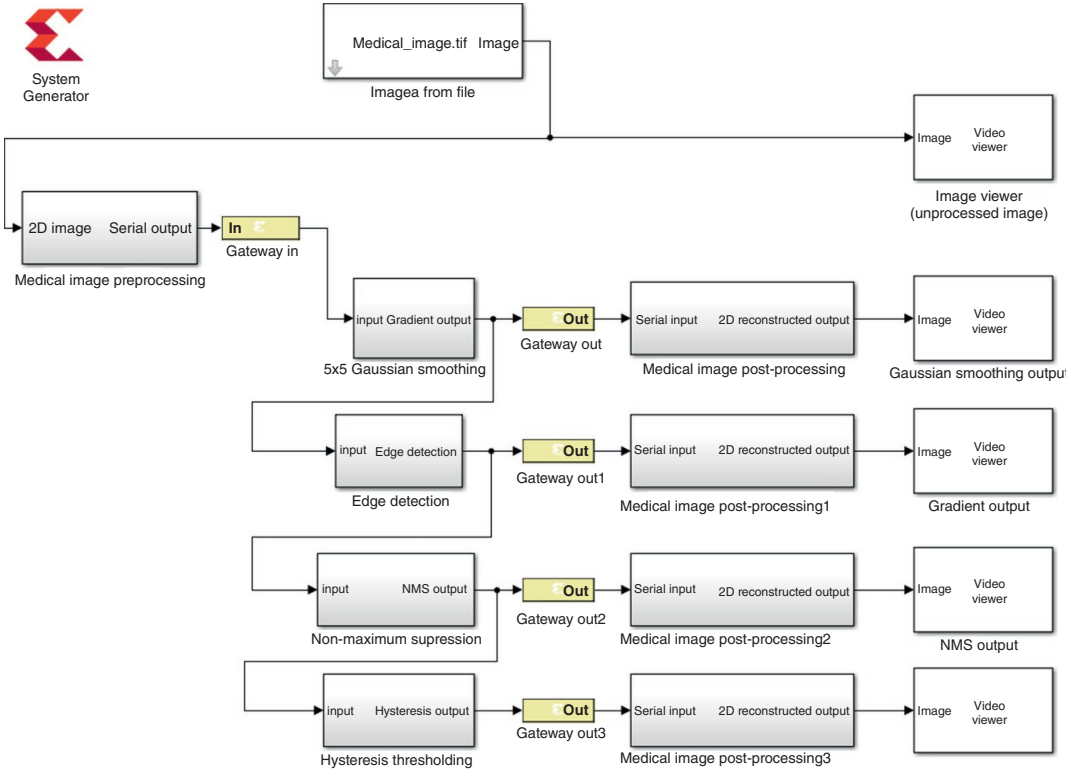

Figure 9.15 System Generator DSP model for Canny Edge detection.

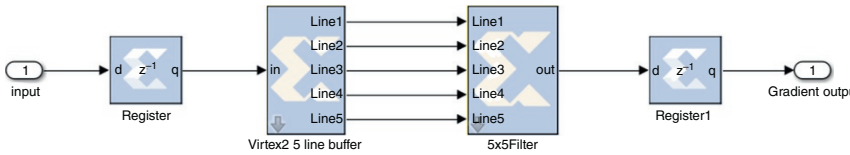

Figure 9.16 Canny method – System Generator DSP sub-model for Gaussian smoothing.

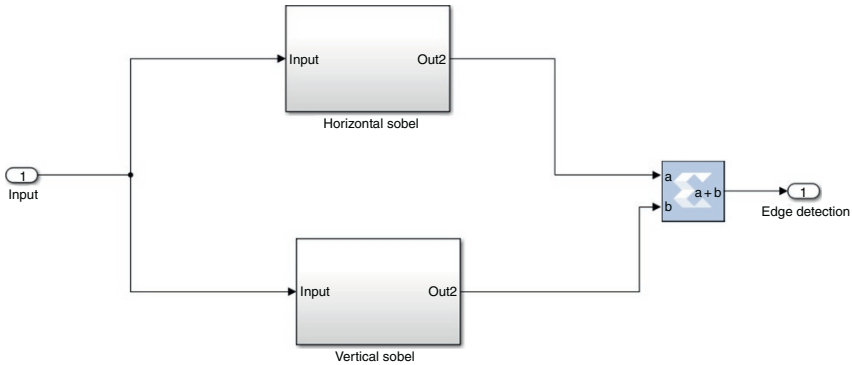

Figure 9.17 Canny method – System Generator DSP sub-model for edge detection (includes Sobel vertical and horizontal kernels).

Once the gradient magnitude has been calculated, NMS is performed. This means that the whole image is scanned in order to remove the pixels that might not be the part of the edges. This is done in such a way that pixels that are local maxima are found in the direction of the gradient (gradient direction is perpendicular to edges). An example of this is to look at the three pixels that are next to each other: pixels *a, b*, and *c*. Let pixel b have larger intensity than both *a* and *c*, where pixels *a* and *c* are in the gradient direction of *b*. As a result, pixel *b* is marked as an edge. Otherwise, if pixel *b* was not a local maximum, it would be set to 0, meaning that it would not be considered an edge pixel.

This block (designed as shown in Figure 9.18) provides edge thinning. It compares all neighboring pixel values to the central pixel and suppresses it to zero value if it is not maximum. Otherwise, it is forwarded to the next block [42].

NMS does not give the perfect result, as some edges might still actually be noise. In order to solve this, canny method applies thresholding to remove the weakest edges and keep the strongest. All of this is achieved using high and low threshold. These thresholds values are calculated based on gradient magnitude histogram of the image. If the gradient magnitude is greater than the high threshold value, the pixel is marked as a strong edge. If the gradient magnitude is lower than the low threshold value, the pixel is marked as a weak edge. The high threshold value is

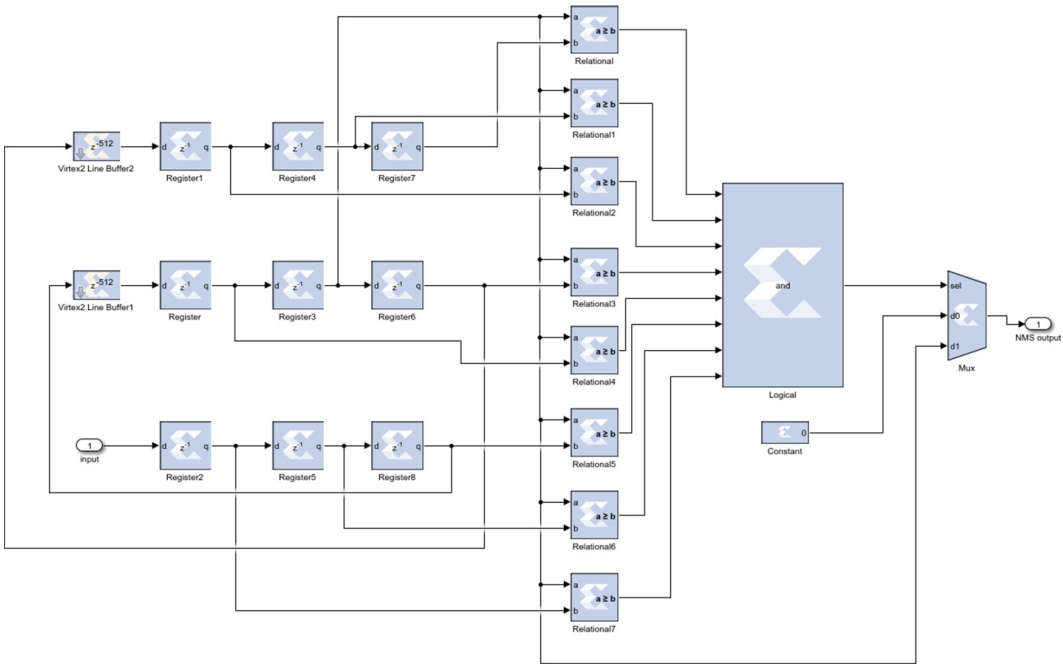

Figure 9.18 Canny method – System Generator DSP sub-model for Non-Maximum Suppression.

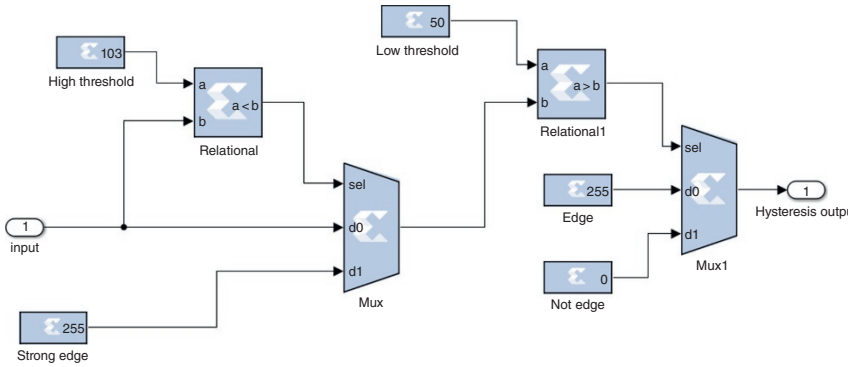

Figure 9.19 Canny method – System Generator DSP sub-model for Hysteresis thresholding.

chosen based on P1 percentage of the total value of pixel intensities. The lower threshold value is P2 percentage of the total value of pixel intensities. Although the percentage values are usually calculated using the cumulative distribution function (CDF), empirically the values of P1 are P2 are chosen as 20 and 40% of the pixel intensities, respectively [42]. Figure 9.19 gives the design of the described subsystem.

9.3.3 Hardware Co-Simulation

The model has been software co-simulated to check that it satisfies the specifications of the selected hardware platform. After software simulation, the System Generator token has to be configured to allow the model to be compiled into hardware [3]. System Generator token dialog box has to be configured with the necessary settings for compilation, synthesis, and clocking. Firstly, compilation Target needs to be chosen by setting up the preferred FPGA platform. Secondly, synthesis tool helps in synthesizing the design (settings are hardware language, creating test bench and design that is synthesized and implemented) and clocking tab (settings are period of the system clock (in nanoseconds) and pin location for hardware clock) [2, 3, 24].

Once the block is configured with the adequate settings, a bit stream (.bit) file is generated. After the bit stream file is generated, hardware co-simulation target is selected and a block will be generated. For our purposes, Artix 7 AC701 Evaluation Platform 1.0 (with the chip Artix7 xc7a200t-1sbg484) is chosen to test the algorithms. This block can be now added to the existing design and both software and hardware the blocks now coexist. The complete design with the hardware and software co-simulation subsystems for the edge detection using Sobel method

is presented in Figure 9.20. All the other designs can be hardware co-simulated using the same process.

9.4 Results and Discussion

The model can generally accept and process any medical image. All input images in this chapter were grayscale, size 256×256, 8-bit representation. The model can also work with other dimensions of the images, although the parts of the design dependent on the size of the image have to be adapted. Also, it should be emphasized that the simulation time has to be greater than product of dimensions. Therefore, for all the simulations in this chapter, time was set to 70 000, which is greater than 65 536.

For comparative study in implementation of the algorithm for negative, two images are used – *spine_axial.tiff* and *lungs.tiff*. The results of creating negative image are given in Figure 9.21, one for the example of axial view of the spine discus (Figure 9.21a) and second for the lungs infected with COVID-19 (Figure 9.21b). As it can be seen, negative image represents total inversion, where light areas became dark and vice versa. Reversing the intensity levels of an image produces the equivalent of a photographic negative. The implementation of this type of processing is suited when white or gray detail enhancement in dark regions of an image is needed, especially in those cases where black areas are dominant. Also, image negatives may contribute to localizing regions of interest better and therefore in setting up an adequate diagnosis.

If the image was color image, the same procedure for creating negative image would be applied separately on R, G, and B color components. In this case, negative color image is color-reversed, where red areas become cyan, greens become magenta, and blues become yellow.

For comparative study in implementation of the algorithm for negative, two images are used – *spine_sagittal.tiff* and *lungs.tiff*. The results of creating negative image are given in Figure 9.22, one for the example of sagittal view of the spine discus (Figure 9.22a) and second for the lungs infected with COVID-19 (Figure 9.22b).

It should be noted that in the case where there is no enough contrast in an image (meaning there were both pixels with values 0 and 255), the implementation of Eq. (9.1) would not make sense. Therefore, contrast stretching makes only sense if pixel intensities fall under narrower histogram, with no values close to 0 and 255.

If the image was color image, the same procedure for enhancing contrast in an image would be applied separately on R, G, and B color components. This algorithm is suitable for use in more complex processing.

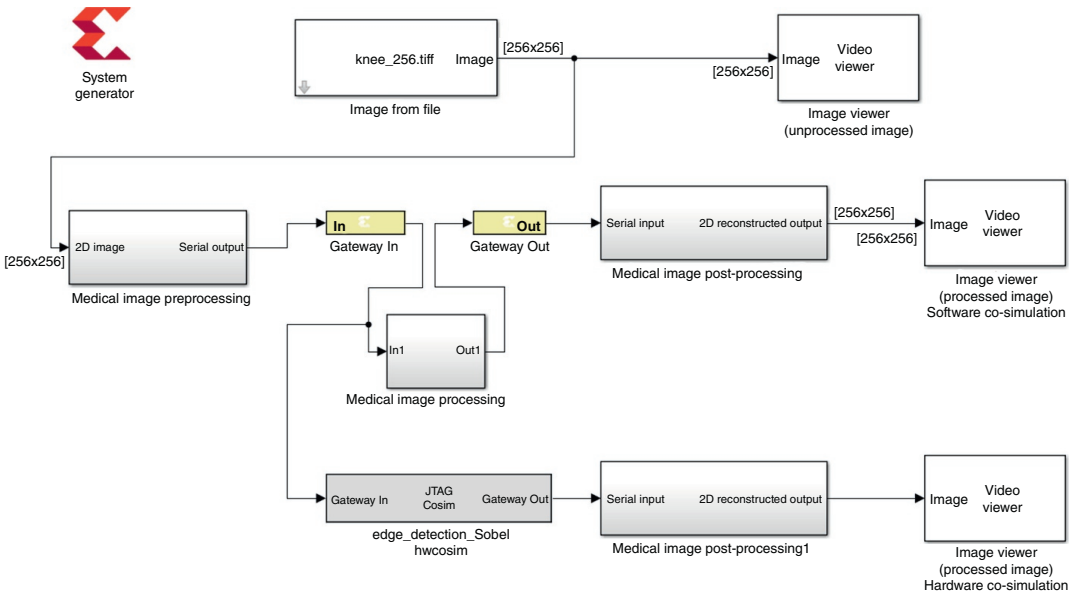

Figure 9.20 Combined software and hardware co-simulation model for the edge detection (implementation of Sobel method).

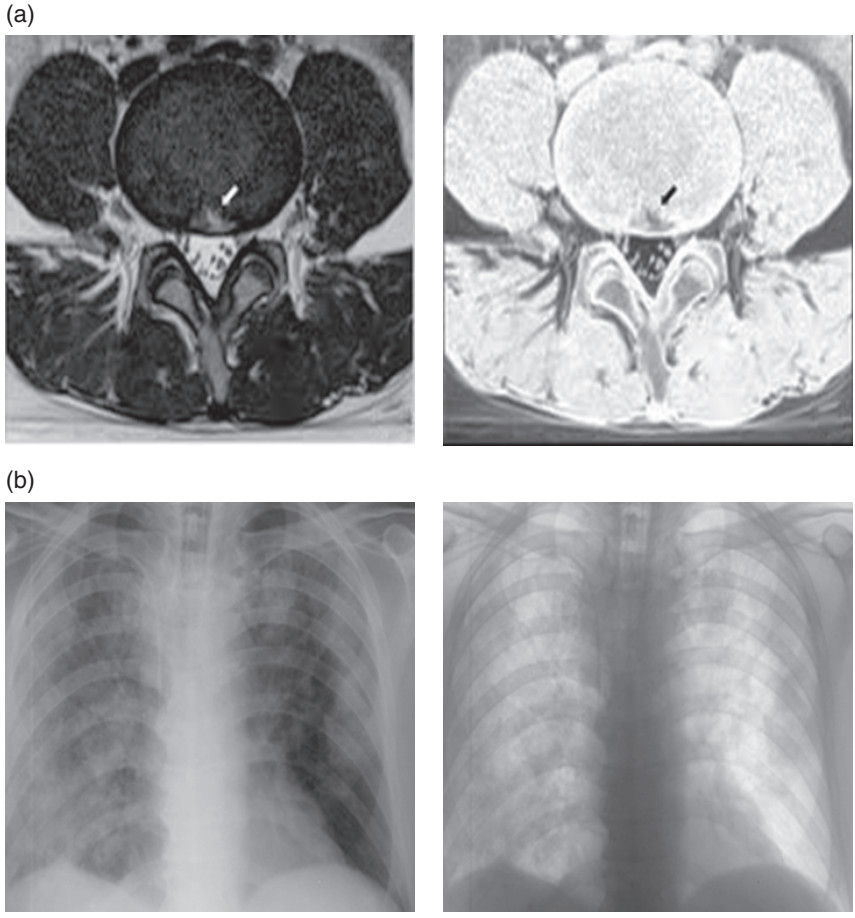

Figure 9.21 Original and output image after application of algorithm for image negative: (a) axial view of spine discus and (b) lungs infected with COVID-19.

For comparative study in implementation of the algorithm for negative, image *knee.tiff* is used. Gradient-based edge detection methods Robert (Figure 9.23b), Prewitt (Figure 9.23b), Sobel (Figure 9.23d), and Canny (Figure 9.23e) were applied and their detection was compared. Visual analysis of results is shown in Figure 9.23, and it can be seen that Sobel and Canny methods achieve more accurate detection. In contrast, the calculation using Robert or Prewitt kernel is simpler and less resource demanding in comparison to the other kernels, since only adder–subtractor logic is used. It should be also emphasized that Canny outputs thinner edges, due to its hysteresis thresholding. The threshold set in all simulations can be

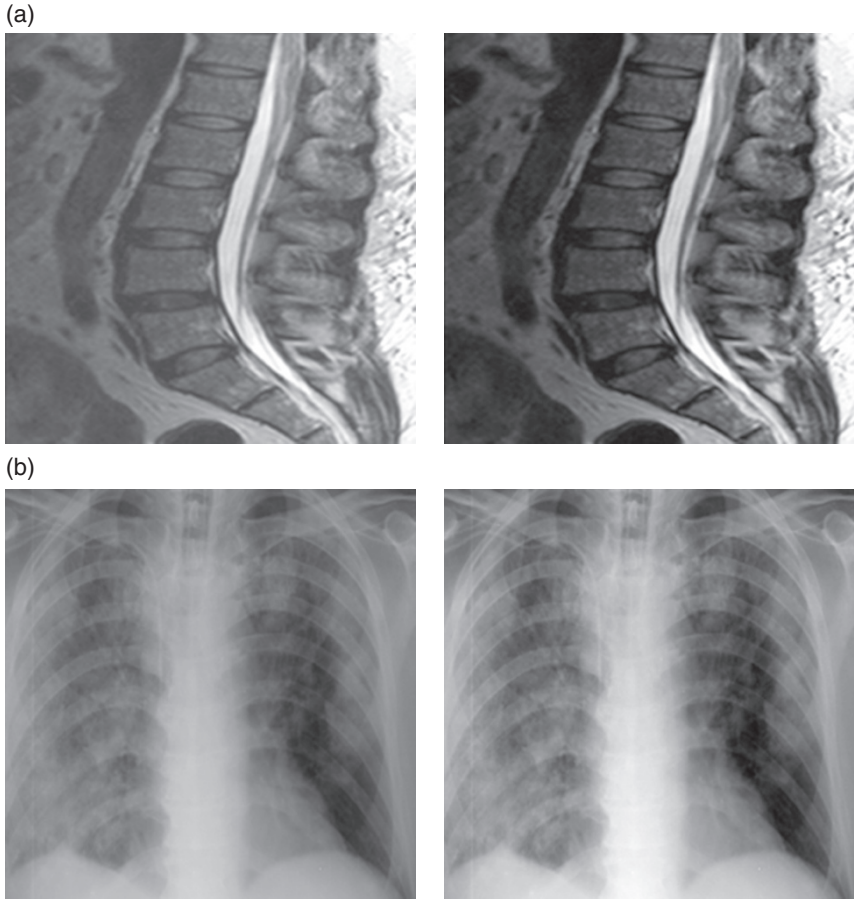

Figure 9.22 Original and output image after application of algorithm for contrast stretching: (a) sagittal view of spine discus and (b) lungs infected with COVID-19.

changed and varied in order to increase signal-to-noise-ratio and achieve best results (Figure 9.23a).

It should be emphasized that Canny method, due to several additional steps in edge detection Gaussian smoothing (Figure 9.24a), calculation of Intensity Gradients (Figure 9.24b), NMS (Figure 9.24c), and Thresholding with Hysteresis (Figure 9.24d), could be plotted after each of the subprocesses (Figure 9.24).

It is noticeable that the presence of noise threshold dependent and some false edges may be detected. With the threshold that was varied, Sobel tends to detect more false edges, and the performance comparable with that of Prewitt method.

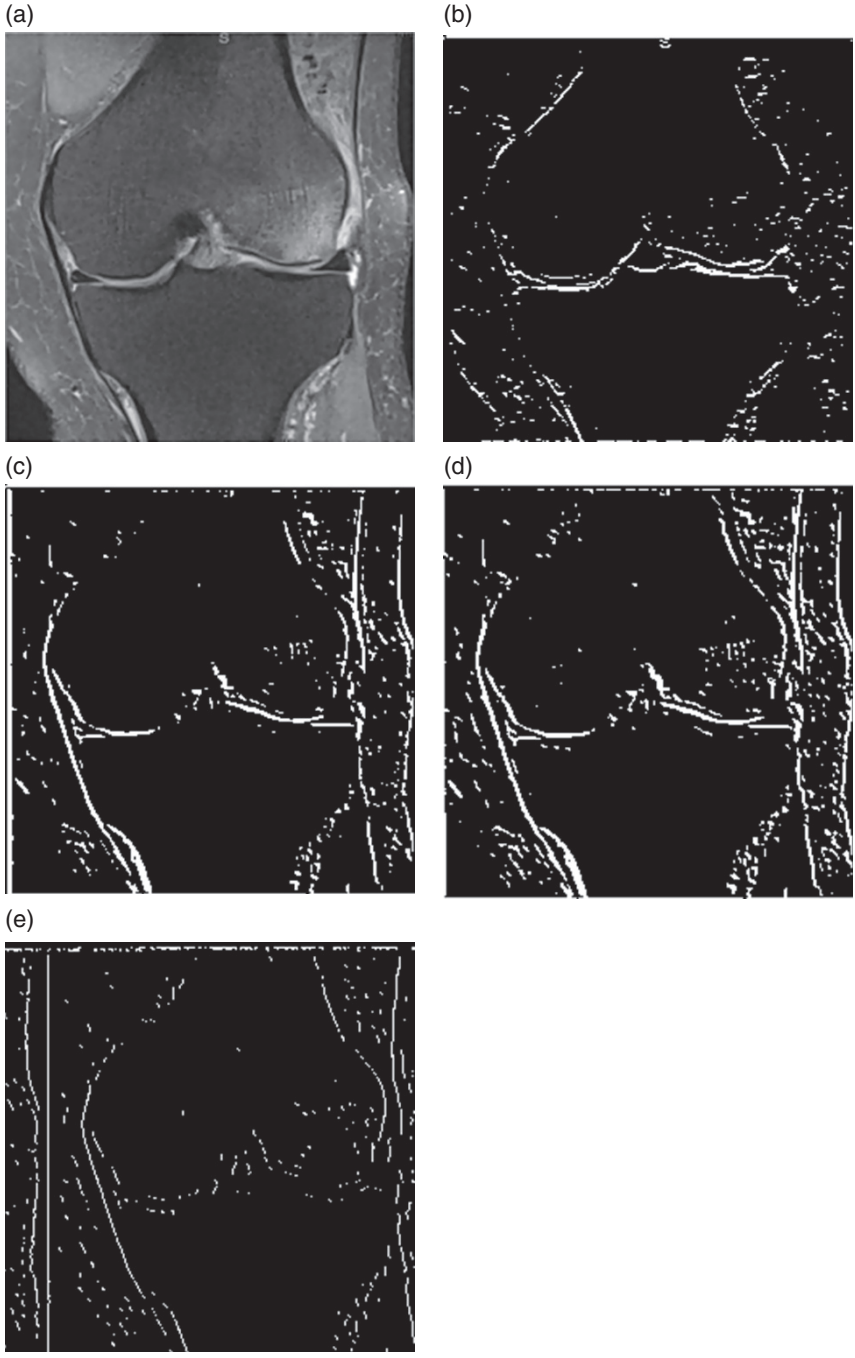

Figure 9.23 Original image (a) compared with output images after application of algorithm for edge detection using Robert (b), Prewitt (c), Sobel (d), and Canny (e) methods.

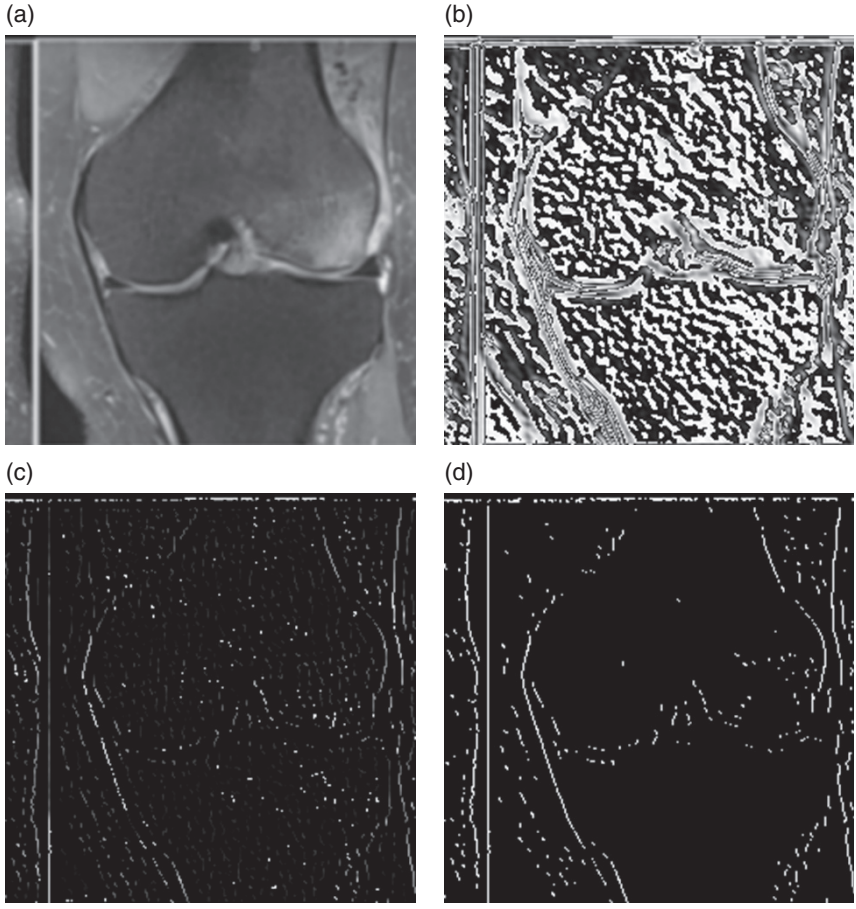

Figure 9.24 Canny edge detection sub-steps: Gaussian smoothing (a), calculation of Intensity Gradients (b), Non-Maximum Suppression (c), and Thresholding with Hysteresis (d).

The same algorithms can be tested with different medical images, which would lead to the conclusion that Robert method is more efficient in images with lower contrast, in which cases the presence of noise is reduced. In the sense of number of edges, Sobel outputs maximum edges whereas Robert provides minimum edges. This is due to the fact that Sobel operator emphasizes on central pixels resulting in thicker edges in comparison to the Robert method. In contrast, Canny output achieves thinner edges.

All the designs are also synthesized and implemented using Xilinx Vivado 2019.1. Table 9.3 shows a comparative analysis of the investigated edge detection

Table 9.3 Post synthesis resource utilization summary.

Edge detection method	LUTs	DSPs	BRAMs	Registers
Robert method	75	0	1	112
Prewitt method	223	0	1.5	266
Sobel method	223	0	1.5	266
Canny method	686	5	7	1024
available	13460	740	365	269200

methods, with respect to resource utilization. The resource utilization speaks a lot about the complexity of the algorithm, meaning that from this analysis it can be also seen that Robert is the simplest and Canny is the most complex algorithm, out of those investigated. Classical gradient-based operators typically have low latency and high throughput and are used in systems where image processing has to be performed in real time [41]. They are the option when applications where precision is not a big concern (i.e. security, tracking, and barcode reader). However, additional techniques need to be added when processing biomedical images, in order to improve precision.

Table 9.4 provides a comparison of the timing paths for the investigated gradient-based edge detection techniques. All the algorithms have passed the Timing path analysis without any violations. Number of paths reported in Timing analysis for Robert method is 8, both for Prewitt and Sobel 19, and Canny 50.

Thus, in applications where medium-range contrast images are of primary concern, Sobel and Prewitt tend to be more appropriate. Robert is the best choice for faster performance with less device resources. As an optimal compromise, the attention should be paid to the desired output – if we do not care about resource utilization in terms that complexity is not a problem, Canny algorithm will satisfy the needs. However, if the resource and timing consumption is the priority, Robert

Table 9.4 Post synthesis timing paths.

Edge detection method	Max_Slack (ns)	Min_Slack (ns)	Max_Delay (ns)	Min_Delay (ns)	Number of paths
Robert	9.118	7.122	2.771	0.635	8
Prewitt	9.118	7.122	2.771	0.635	19
Sobel	9.118	7.122	2.771	0.635	19
Canny	8.699	7.122	2.771	1.332	50

would be the chosen algorithm. In between lies Sobel and Prewitt with balance complexity and accuracy.

9.5 Conclusions

Due to short time-to-market constraints, the need for fast prototyping using the tools such as XSG or Matlab Simulink is now very significant. The XSG tool makes it possible to construct complex designs and implement algorithms using friendly environment with processing units represented by blocks. This tool not only supports the software simulation of different algorithms, but it is also very useful in FPGA hardware synthesization to maximally exploit the hardware parallelism, robustness, and speed. Simulink/XSG characteristics that give them advantages over other solutions are fast time to market for image processing algorithms (shortening the production period from algorithm design to implementation on hardware); easy-to-use interface (easily arranged and separated into subsystems); flexible modeling and simulation (relatively easy debugging process); easier software to hardware transfers (the generator produces all necessary files for the Xilinx FPGAs, with additional possibility to monitor and control the utilization of resources).

In this chapter, image contrast stretching, edge detection, and thresholds are implemented using coupled Simulink-XSG tool, in order to be prepared for FPGA. Algorithms are compared, both visually and quantitatively (resulting accuracy, device utilization, power, minimum period, and maximum frequency). The results suggest that Robert method is, due to its simplicity, most efficient when it comes to device utilization; however, other algorithms, such as Prewitt, Sobel, and Canny are more accurate. The result given in this chapter proves that the proposed hardware implementation of different algorithms gives optimal results for (bio)medical images. Thus, this proposed architecture is very well suited for real-time medical image processing applications.

The main advantages of implementing different algorithms on a FPGA is full use of large memory, embedded multipliers, and the programming the application-specific hardware that offers greater processing speed. The methodology used in this chapter can be expanded further to be used in more complex real-time medical image processing systems.

Acknowledgments

This research is funded by Serbian Ministry of Education, Science, and Technological Development [451-03-68/2020-14/200107 (Faculty of Engineering, University of Kragujevac)]. This research was also supported by the CEI project “Use of

Regressive Artificial Intelligence (AI) and Machine Learning (ML) Methods in Modelling of COVID-19 spread – COVIDAi."

References

1 Xilinx, (2009). *System Generator for DSP User Guide*, USA: Xilinx.

2 Reddy, G.B. and Anusudha, K., (2016). Implementation of image edge detection on FPGA using XSG. Nagercoil, India.

3 Mohammed, A., Rachid, E., and Laamari, H. (2014). High level FPGA modeling for image processing algorithms using Xilinx System Generator. *International Journal of Computer Science and Telecommunications* **5** (6): 1–8.

4 Ownby, M. and Mahmoud, W.H. (2003). *A design methodology for implementing DSP with Xilinx/sup/spl reg//System Generator for Matlab/sup/spl reg*. Morgantown, WV, USA, USA.

5 Vukicevic, A.M., Stojadinovic, M., Radovic, M. et al. (2016). Automated development of artificial neural networks for clinical purposes: Application for predicting the outcome of choledocholithiasis surgery. *Computers in Biology and Medicine* **75**: 80–89.

6 Šušteršič, T., Milovanović, V., Ranković, V., and Filipović, N. (2020). A comparison of classifiers in biomedical signal processing as a decision support system in disc hernia diagnosis. *Computers in Biology and Medicine* **125** (103978).

7 Šušteršič, T., Ranković, V., Peulić, M., and Peulić, A. (2019). An early disc herniation identification system for advancement in the standard medical screening procedure based on bayes theorem. *IEEE Journal of Biomedical and Health Informatics* **24** (1): 151–159.

8 Hasan, S., Yakovlev, A. and Boussakta, S. (2010). *Performance efficient FPGA implementation of parallel 2-D MRI image filtering algorithms using Xilinx System Generator*. Newcastle upon Tyne, UK.

9 Othman, M.F.B., Abdullah, N. and Rusli, N.A.B.A. (2010). *An overview of MRI brain classification using FPGA implementation*. Penang, Malaysia.

10 Tana, H., Sazish, A.N., Ahmad, A. et al., (2010). *Efficient FPGA implementation of a wireless communication system using bluetooth connectivity*. Paris, France.

11 Moses, C.J., Selvathi, D. and Rani, S.S. (2010). *FPGA implementation of an efficient partial volume interpolation for medical image registration*. Ramanathapuram, India.

12 Šušteršič, T. and Peulić, A. (2019). Implementation of face recognition algorithm on field programmable gate array (FPGA). *Journal of Circuits, Systems and Computers* **28** (8): 1950129.

13 Šušteršič, T., Vulović, A., Filipović, N., and Peulić, A. (2016). *FPGA implementation of face recognition algorithm*. In: *Pervasive Computing Paradigms for Mental Health* (eds. N. Oliver, S. Serino, A. Matic, et al.), 93–99. Cham: Springer.

14 Šušteršič, T., Vulovic, A., Trifunovic, N. et al. (2019). Face recognition using maxeler dataflow. In: *Exploring the DataFlow Supercomputing Paradigm* (eds. V. Milutinovic and M. Kotlar), 171–196. Cham: Springer.

15 Elamaran, V., Aswini, A., Niraimathi, V., and Kokilavani, D. (2012). FPGA implementation of audio enhancement using adaptive LMS filters. *Journal of Artificial Intelligence* **5** (4): 221.

16 Christe, S.A., Vignesh, M., and Kandaswamy, A. (2011). An efficient FPGA implementation of MRI image filtering and tumor characterization using Xilinx system generator. *International Journal of VLSI Design and Communication Systems* **2**: 95–109.

17 Karthigaikumar, P., Kirubavathy, K.J., and Baskaran, K. (2011). FPGA based audio watermarking—covert communication. *Microelectronics Journal* **42** (5): 778–784.

18 Raut, N.P. and Gokhale, A.V. (2013). FPGA implementation for image processing algorithms using Xilinx System Generator. *IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)* **2** (4): 26–36.

19 Moreo, A.T., Lorente, P.N., Valles, F.S. et al. (2005). Experiences on developing computer vision hardware algorithms using Xilinx System Generator. *Microprocessors and Microsystems* **29** (8-9): 411–419.

20 Saidani, T., Dia, D., Elhamzi, W. et al. (2009). Hardware co-simulation for video processing using xilinx system generator. London, U.K.

21 Gupta, A., Vaishnav, H., and Garg, H. (2015). Image processing using Xilinx System Generator (XSG) in FPGA. *International Journal of Research and Scientific Innovation* **2** (9): 119–125.

22 Xilinx, (2019). *Compatibilty versions of system generator for DSP with versions of Vivado design tools*. https://www.xilinx.com/support/answers/55830.html (accessed 25 January 2019).

23 Design, M.B.D. (2012). Vivado Design Suite Reference Guide

24 Hanisha, V. and Kumari, G.V. (2016). Hardware implementation of image denoise filters using system generator. *International Journal of VLSI System Design and Communication System* **4** (10): 940–946.

25 Kumar, K.A. and Kumar, M.V. (2014). Implementation of image processing lab using Xilinx System Generator. *Advances in Image and Video Processing* **2** (5): 27–35.

26 Kabir, S. and Alam, A.A. (2014). Hardware design and simulation of sobel edge detection algorithm. *International Journal of Image, Graphics and Signal Processing* **6** (5): 10.

27 Sujatha, C. and Selvathi, D. (2014). Hardware implementation of image edge detection using Xilinx System Generator. *Asian journal of scientific research* **7** (2): 188.

28 Que, Z., Zhu, Y., Wang, X. et al. (2010). Implementing medical CT algorithms on stand-alone FPGA based systems using an efficient workflow with SysGen and simulink. *Proceeding of the IEEE International Conference on Computer and*

Information Technology (CIT 2010), Bradford, UK: (29 June to 1 July 2010), pp. 2391–2396.

29 Sudeep, K.C. and Majumdar, J. (2011). A novel architecture for real time implementation of edge detectors on FPGA. *International Journal of Computer Science Issues (IJCSI)* **8** (1): 193.

30 Harinarayan, R.R., Pannerselvam, M.M. and Tripathi, D. K. (2011). Feature extraction of digital aerial images by FPGA based implementation of edge detection algorithms. *Proceedings of the International Conference on Emerging Trends in Electrical and Computer Technology,* Tamil Nadu, March 23-24

31 Shanshan, Z. and Xiaohong, W. (2010). Vehicle image edge detection algorithm hardware implementation on FPGA. *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*.

32 Basu, A., Das, T.S., and Sarkar, S.K. (2012). SysGen architecture for visual information hiding framework. *International Journal of Emerging Technology and Advanced Engineering* **2**: 32–40.

33 Said, Y., Saidani, T., Smach, F. and Atri, M., (2012). *Real time hardware co-simulation of edge detection for video processing system*. Yasmine Hammamet, Tunisia.

34 Šušteršič, T., Milovanović, V, Ranković, V. et al., (2019). Medical image processing using Xilinx system generator. *International Conference on Computational Bioengineering*. Springer, Cham, pp. 104–116.

35 Fuad, K.A.A. and Rizvi, S.M. (2015). Hardware software co-simulation of Canny edge detection algorithm. *International Journal of Computer Applications* **122** (19): 7–12.

36 Tutorials_point, (2019). *Histogram stretching*. https://www.tutorialspoint.com/dip/histogram_stretching.htm (accessed 25 April 2019).

37 Gonzalez, R.C. and Woods, R.E. (2008). *Digital Image Processing*, 3e. New Jersey: Prentice Hall.

38 Yasri, I., bin Hamid, N.H. and Yap, V.V., (2009). *An FPGA implementation of gradient based edge detection algorithm design*. Kota Kinabalu, Malaysia.

39 Xu, Q., Chakrabarti, C. and Karam, L.J., (2011). *A distributed Canny edge detector and its implementation on FPGA*. Sedona, AZ, USA.

40 Xu, Q., Varadarajan, S., Chakrabarti, C., and Karam, L.J. (2014). A distributed canny edge detector: algorithm and FPGA implementation. *IEEE Transactions on Image Processing* **23** (7): 2944–2960.

41 Mahalle, A.G. and Shah, A.M. (2018). Hardware co-simulation of classical edge detection algorithms using Xilinx System Generator. *International Research Journal of Engineering and Technology (IRJET)* **5** (1): 912–916.

42 Kumar, M. and Singh, S. (2020). Hardware and software implementation of canny edge detection algorithm. *Journal of Information and Computational Science* **10** (4): 415–426.