Chapter 4
Application specific bit width for intermediate data nodes: method and results

Chapter 3 discussed some of the HLS optimization directives available in existing tools. We also discussed the results achieved as a result of the usage of these directives on applications that we created using MATLAB HDL coder and Vivado HLS. The implementation results achieved were compared against the those achieved from hand-coded RTL-based design flows or which are already published in the literature. In this chapter, we propose a novel optimization method for HLS-based designs. We have named this novel methodology as "Application-Specific Bit Width for Intermediate Data Nodes," i.e., ASBWIDN. We discuss this methodology in Section 4.1. The methodology is applied to multiple designs from different application areas like signal processing (bandpass filter, software defined radio), image processing (Sobel edge filter, Harris corner detector). The obtained results are discussed in subsequent sections (4.2, 4.3, etc.). A comparison of the implementation results for these applications against hand-coded RTL implementation results previously published in the literature is also presented.

4.1 Introduction to ASBWIDN method

Chapter 3 discussed the design and implementation methods for multiple applications using high-level synthesis platforms like MATLAB HDL coder and Vivado HLS. With the increasing complexity of digital designs, there is a need for verification methods to catch corner case RTL functional errors in digital circuits or SoCs. Prior methods of verification consume long test runtime for the completion of various tests, making the verification methodology time-consuming and expensive. Moreover, running a design on FPGA during the RTL verification phase may result in inefficient usage of the target hardware if the area utilized by the design or speed of operation is sub-optimal. This is not cost-effective as some of these platforms are much more expensive as compared to the cost of licensing simulation tools [4.1, 4.2, 4.3].

Hence, there is a need for a methodology such that an HLS designer may deploy to improve the area utilization, speed, and power dissipation of the actual implementation to be targeted.

There are primary ports that function as inputs or outputs for any design implementation in HLS or any hand-written RTL, depending on the functionality. Additionally, there exist internal signals employed to connect internal modules or store intermediate results. While coding in any HDL such as, but not limited to, Verilog or VHDL, the designers specify the width (size) of all input and output signals. This is based on information the signal(s) is expected to carry. Hence, the designer is pessimistic about the variables' choice as the widths are not known at the start of the project.

This problem gets more complex when there are intermediate nodes involved specifically for designs where there is a high degree of data processing involved. In light of the above-described points, there is a need for a methodology that helps to find out the optimum bit widths of input, output, and intermediate nodes. The objective aims for better area utilization, power reduction, and speed improvement for a given application.

Herein, a novel HLS design method is proposed which:

1) Calculates optimum bit widths of various inputs, outputs, and intermediate signal nodes of the design to be used in the HLS design process.
2) Allows redesigning if simulation results do not adhere to mandatory threshold values. Such values are important because they establish when the design is effectively considered as verified, or when synthesis results are beyond the area and timing budgets.
3) Is fully scalable and applicable for all HLS tools and platforms.
4) Provides improvement in all three parameters: area, speed, and power (which are otherwise orthogonal) without any impact on the functionality of the design.

4.2 Detailed description of the methodology

In high-level programming languages like C/C++, dealing with real numbers, numbers with a fractional part like 6.5, 3.2, etc., is a common practice. However, we can represent floating-point numbers in fixed-point notation [4.4]. We represent 53 (decimal) as 110101 in binary. The key to representing fractional numbers such as 26.5 is the notion of the binary point. A binary point is similar to the decimal point in the decimal system. It serves as a divider between the integer and fractional part of the number. In a decimal system, the decimal point indicates the numeral position that the coefficient should multiply by $10^0 = 1$. For example, in 26.5 (decimal arithmetic), 2 has a weight of 10, 6 has a weight of 1 and 5 has a weight of 1/10, i.e., $26.5 = 2 \times 10 + 6 \times 1 + 5 \times (1/10)$. The same idea of the decimal point can be implemented to the binary representation, thus making a binary point.

As an example, 11010.1 = 16 + 8 + 0 + 2 + 0 + 0.5 = 26.5. So, to represent 26.75, we need one extra bit on the fraction side, i.e. 11010.11 = 26.75. So, a node that has a minimum value of −26.75 and a maximum value of 26.75 would need 8 bits (5 bits for the integer part, 2 bits for the fraction part and 1 bit for the sign).

Similarly, a number varying between −1 and 1 may be represented using 1 bit for the integer and multiple bits for the fraction. Moreover, a number varying from 1 to 4 may be represented using only 3 bits with no bits used for the sign. Another example being if a signal takes the value between 0 and 23.5 (0.5 is the resolution on the decimal side), it needs 1 bit for the fraction part and 5 bits for the integer part, so, in total 6 bits.

23.5d = 10111.1.

Figure 4.1 illustrates a flow chart of the method deployed for performing synthesis as part of the verification of digital design by calculation of maxima and minima bit width(s). In this operation, an HLS model is designed with the highest permissible bit width for all the inputs/outputs (I/Os) and all the intermediate nodes. System-level simulation tests are performed on the digital design by applying multiple stimuli from the verification environment. In the same operation, the values of I/O and intermediate nodes are captured for all the tests run on digital design. During the verification process, every signal's bit width is calculated based on each I/O and the intermediate nodes' range of values. The simulation process is repeated with multiple tests that are part of the regression suite. Moreover, the database of signal values, with subsequent runs, is also appended. The optimal signal width is calculated based on the range of values for all signals and the design is resynthesized based on the same bit width for signals as calculated. The pseudo-code for the method is depicted in the Appendix. Further, the method is implemented as a stand-alone C based program which gets called at the start of HLS tool as soon as the model or design is run.

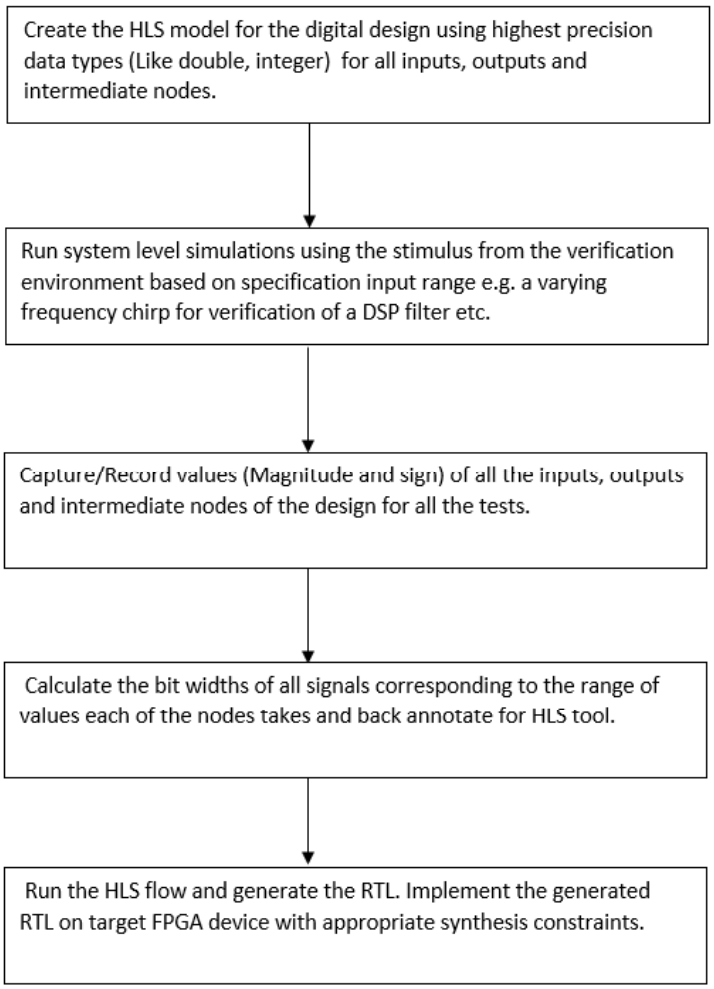

Figure 4.1. Algorithm for application-specific bit width for intermediate data nodes.

The proposed methodology for high-level synthesis and verification of the digital designs is compatible with all HLS tools and may be applied to digital designs targeting any application. The method is quickly scalable for any inputs, outputs, and internal signals that the design could contain. As the modification is done in the RTL design itself using our methodology, it is independent of the target FPGA platform used. Hence, it is applicable for all synthesis tools and hardware platforms.

The proposed method can be performed without manual intervention. Moreover, it may be automated with the deployment of multiple applications to target the area, speed, or power optimization, depending on the application of the design. It also enables the redesign if simulation results do not adhere to threshold values mandatory for the design to be considered verified, or when synthesis results are beyond the area and timing budgets.

We applied our proposed methodology to five different application designs. The results achieved using the same are described in the subsequent Sections 4.3–4.7.

4.3 Bandpass DSP filter design, implementation, and optimization

In almost every digital signal processing system, signal filtering is an important aspect. The process of filtering, allows specific frequency components of a signal to pass through while attenuating the rest of the frequency components. Attenuated component is typically noise in a particular range of frequencies depending on whether it is a high pass, low pass, or bandpass filter.

In this work, a bandpass DSP filter was designed using HLS and the implementation was optimized using the method proposed in Sections 4.1 and 4.2. The filter specifications are as mentioned below:

Filter type: Bandpass FIR Equiripple filter [4.5]

Order of filter: 40

Sampling Frequency, Fs = 48000 Hz (48 kHz)

Lower stop band frequency, Fstop1 = 6000 Hz (6 kHz)

Lower pass band frequency, Fpass1 = 9000 Hz (9 kHz)

Upper pass band frequency, Fpass2 = 12000 Hz (12 kHz)

Upper stop band frequency, Fstop2 = 15000 Hz (15 kHz).

An HLS model of the filter was created using fixed-point data types using MATLAB and Simulink and a system-level simulation was performed using a chirp signal as input. Figure 4.2 shows the system-level model, and Figure 4.3 shows the magnitude response of the filter for both the pessimistic bit width as well as optimized bit width designs. Since the quantization error is minimal, the two magnitude response plots almost overlap with each other.

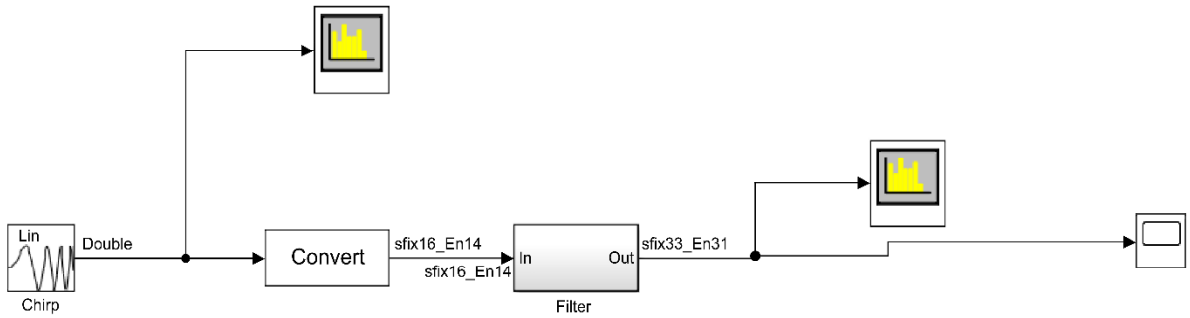

Figure 4.2. Fixed-point bandpass filter design fed with chirp signal input.

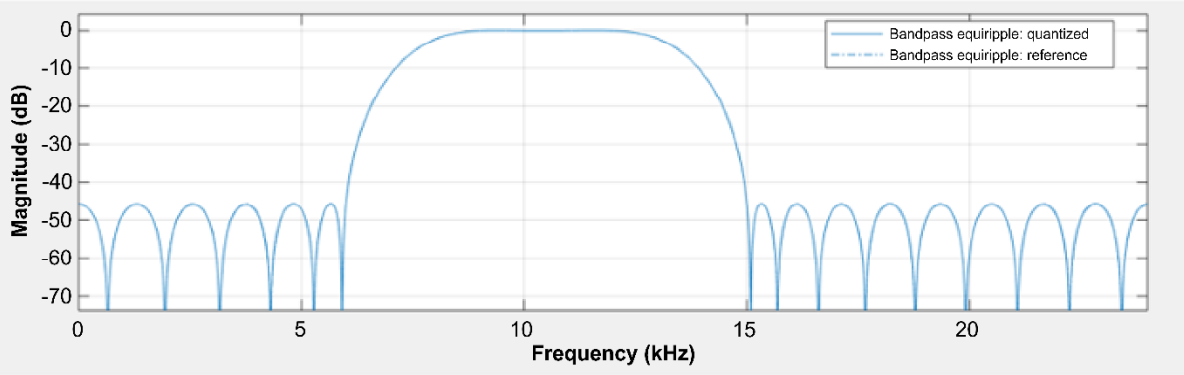

Figure 4.3. Magnitude response of bandpass FIR filter.

Table 4.1. Kintex 7 implementation results for bandpass FIR filter (Default bit width)

Resource	Utilization	Available	Utilization (%)
LUTs	2565	41000	6.26
Flip-flops	1280	82000	1.56
DSP	140	240	58.33
IO	106	300	35.33
BUFG	1	32	3.13

Critical Path: 23.711 ns, Operation Freq. = 40 MHz, Total on-chip Power = 0.379 W

Table 4.2. Kintex 7 implementation results for bandpass FIR filter (using bit width optimization)

Resource	Utilization	Available	Utilization (%)
LUTs	630	41000	1.54
Flip-flops	640	82000	0.78
DSP	39	240	16.25
IOs	53	300	17.67
BUFG	1	32	3.13

Critical Path: 19.065 ns, Operation Freq. = 52 MHz, Total on-chip Power = 0.18 W

MATLAB HDL coder was used to generate synthesizable Verilog RTL for the filter and the same was implemented on Xilinx Kintex 7(XC7K70T-FBG676) FPGA.

Table 4.1 shows the implementation results for the same application (default bit width

which is pessimistic). We also optimized the application using application-specific bit widths for intermediate data nodes. Table 4.2 shows the implementation results post the optimization. As tables 4.1 and 4.2 show, there is almost a 50 percent reduction in FPGA resources and about 20 percent improvement in frequency of operation using bit width optimization. Additionally, the power dissipation is also reduced by about 50 percent using the optimization method.

4.4 Sobel edge filter design, implementation, and optimization

Sobel edge filtering, also known as Sobel-Feldman filtering is a signal filtering algorithm commonly applied to image processing and computer vision applications. Essentially, this is a discrete differentiating operator used to approximate the gradient of image intensity in both dimensions. The output is an image enhancing the edges (region where gradient is large) in the input image.

4.4.1 Introduction to Sobel edge detector application design

In almost all modern image processing and computer vision systems, the extraction of a region of interest is fundamental requirement. For example, it is crucial in applications such as advanced driver assistance systems (ADAS) for pedestrian detection, traffic signal detection, blind-spot detection, and lane departure warning systems. It is also commonly used in video surveillance applications such as object tracking, scene reconstruction, and weather condition monitoring. An “edge” or a line is one of such features in images. Multiple algorithms exist for detecting edges in images. Some of the commonly applied algorithms include the Roberts cross, Prewitt edge, and Sobel edge detection [4.6]. Because of their simple algorithmic approach based on approximate gradient calculation, Sobel edge filters are extensively used for data extraction and image segmentation in various application designs. Such filters are typically required to run at real-time speeds and hence are good candidates for FPGA prototyping. Most commonly, they run alongside other algorithms such as non-maxima suppression, matching using the sum of absolute differences, matrix computation, and triangulation because typically the output of the filter is used to make a decision or take an action like triggering the brake for a car, controlling the exposure for a camera shutter etc. For that reason, Sobel edge algorithm implementation is expected to be area and speed efficient for target FPGA devices.

4.4.2 Previous works for Sobel edge filter implementation

An implementation of the Sobel edge filter on Spartan 3, Spartan 6, and Virtex 5 FPGA, was presented by Chaple et al. in 2014 [4.7]. However, it was not a real-time implementation. In 2016, Israni et al. introduced a similar implementation in Verilog for license plate detection using the same algorithm in MATLAB [4.8]. In the same year, Amara et al. presented an HD video streaming architecture based on Sobel edge detection [4.9]. In 2018, a low-area implementation of the Sobel operator for full HD real-time video streaming was proposed by Eetha et al. [4.10]. Their design was implemented on a Xilinx Zynq (ZC 702) board. Such a board contains an ARM CORTEX A9 core and programmable FPGA on a single fabric. In 2015 Ismail et al. proposed an implementation of the Sobel edge algorithm on Virtex 6 FPGA using VHDL [4.11]. In 2013, Sanjay et al. proposed an implementation for Sobel edge filter which occupied almost 40 percent lesser resources while maintaining real time frame rates. They used pipelining to further optimize the performance for their proposed implementation [4.12].

4.4.3 Proposed implementation for Sobel edge filter using ASBWIDN

As part of our contributions, we propose an implementation of the Sobel edge detection algorithm on Xilinx Kintex 7 device. Our implementation is better in terms of speed and area utilization on FPGA when compared to other implementations. This design was constructed using a novel HLS design method that constrains intermediate signal widths according to the application (input stimulus). For hand-coded RTL hardware implementations of image processing filters such as the Sobel, there are primary ports that function as inputs or outputs depending on the functionality. There are also internal registers and wires used for connecting internal modules or for storing intermediate results. While coding in any HDL, such as Verilog or VHDL, designers define the width (size) of all input and output signals based on data the signal(s) is/are required to carry. A designer is habitually pessimistic toward selecting variables as the widths are unknown at the beginning of the project. In the Sobel edge filter case, there are cascaded stages of multipliers and adders that act on incoming image pixels and stored gradient matrices in both the horizontal and vertical directions. In such cases, manual selection of data types for the primary ports and intermediate nodes leads to pessimistic synthesis and slower operation frequency, and poor area utilization on the target FPGAs. For the Sobel edge filter produced using HLS for this study, a MATLAB model was created with the highest permissible bit width for all input and output nodes, along with intermediate nodes of the design. For the

stimulus, a video stream was chosen, inputting back-to-back image frames with a resolution of 1920 (width) × 1080 (height) × 3 (colors) with 8 bits representing each color. The frames were transformed to pixels using the Vision HDL toolbox in MATLAB. The generated pixels were fed into the design on a per clock cycle basis. The output pixels were transformed back to frames and written to an image file that can be visualized using any image viewer software. Figure 4.4 shows the method for executing the multiplication and addition on incoming image pixels using the Sobel operator. Figure 4.5 shows the pseudo-code for the method. In addition to proposed design, a parallel Simulink library block was added for the Sobel edge detector for visualizing the results in an overlaid manner. The Simulink standard library block received the same input images as the stimulus to compare the results against the proposed hardware implementation.

Input image:

A11	A12	A13	..	
A21	A22	A23	..	
A31	A32	A33	..	
..	..	..	..	

Kernel:

1	0	−1
2	0	−2
1	0	−1

Output image:

B11	B12	B13	..	
B21	B22	B23	..	
B31	B32	B33	..	
..	..	..	..	

$$B22 = A13 - A11 + 2 \times A23 - 2 \times A21 + A33 - A31$$

Figure 4.4. Sobel operator applied on input image pixels.

```
Gx = fi( 0 , 1 , 32, 0, fm);
        Gy = fi ( 0 , 1 , 32, 0, fm);
                for yi = 1:3
                        for xi = 1:3
                        Gx (:) = Gx + k(yi,xi) * Kx(yi,xi);
                Gy (:) = Gy + k(yi,xi) * Ky(yi,xi);
                        end
                end
        G = fi ( abs(Gx) + abs(Gy) , 1 , 32 , 0 , 0 , fm);
        Gd = fi ( floor(fi_div_by_shift(G,2)), 1 , 32, 0 , fm);
        Gdm = fi ( min (Gd, fi(255, 1, 9, 0, fm)), 1 , 32 , 0 , fm);
```

Figure 4.5. Multiplication and additions using the Sobel operator.

The same method as described in Section 4.2 for ASBWIDN was used for this design as well. The simulation was run over the entire set of input images. The absolute minima and maxima value for each of the inputs, outputs and intermediate nodes in the design was recorded. This minima and maxima database was appended with data from subsequent simulation runs until all the varied images were processed. Afterward, the widths for all the primary input, output, and internal signals were recalculated based on the range of values it covered. The HLS tool was then back-annotated with the signal widths to generate RTL with updated "optimum" constraints for all the data nodes. For example, if there is a 32-bit multiplier inferred in the design for 2-pixel multiplication (each pixel being 8 bit), this could be substituted by an 8-bit multiplier, which leads to lesser resources and shorter critical path. This points to a better compile frequency. The method was applied in this work for MATLAB HDL coder-based design for the Sobel edge filter. The method can be readily scaled for any number of inputs, outputs, and internal signals that a design could contain. In this study, calculating the minima and maxima for each node was completely automated and did not require any manual intervention. The final generated RTL, with constrained bit widths, was implemented on the target FPGA.

4.4.4 Simulation, FPGA implementation results, and comparison

The Verilog design produced using MATLAB HDL coder was implemented on Virtex 6 and Kintex 7 FPGA (Xilinx) [4.11]. It was checked for functionality using the "FPGA-in-the-loop" feature of MATLAB and HDL Verifier [4.12]. Figure 4.6 shows the input and output images for the FPGA implemented design. The results are equal to the golden MATLAB model results for the same input image. Moreover, the generated Verilog RTL code was also simulated utilizing a nonsynthesizable testbench using Vivado xSim software. Functional simulation results from the Vivado xSim simulator also matched the high-level simulation results with regards to the output pixel values. Figure 4.7 presents the RTL simulation results. Using the FPGA-in-the-loop feature of MATLAB HDL Verifier, it was also deduced that quantization error introduced (Due to the

fact that we used reduced bit widths instead of double precision data types) was less than 1 percent.

Figure 4.6. 1920 × 1080 image before and after Sobel edge filtering.

Figure 4.7. Functional RTL simulation results for HDL implementation of Sobel edge filter.

The proposed Sobel edge detector design was built using optimum bit width data types, as described in Section 4.4.3, and simulated on xSim. The Verilog design was synthesized using Vivado 2019.1 targeting Xilinx Kintex 7(XC7K70T-FBG676). The implementation reports from the design are presented in Table 4.3.

Table 4.3. Proposed Sobel edge detector implementation for Kintex7

Resource	Utilization	Available	Utilization (%)
LUTs	335	41000	0.82
LUT-RAMs	33	13400	0.25
Flip-flops	309	82000	0.38
DSP	1	135	0.74
IOs	64	300	21.33
BUFG	1	32	3.125

Critical Path: 4.041 ns, Operation Freq. = 250 MHz, Total On-chip Power = 6.573 W

The critical path is a function of maximum combinational logic delay in the design calculated by the synthesis tool. We also compared the synthesis results for the proposed implementation with the golden HDL coder implementation (library block) of the same design for the same target device [4.13]. The comparison of the results is compiled in Table 4.4. Additionally, our implementation was also compared with available published results from other authors. Table 4.5 shows a comparison of the speed of operation and percentage gain in area saving over other available implementations [4.14, 4.15].

Table 4.4. Comparison results with benchmark implementation for Sobel edge (Kintex 7)

Resource	Proposed design	Benchmark design (Default bit widths)
LUTs	335	707
LUT-RAMs	33	74
Flip-flops	309	502
DSP	1	2
IOs	64	182
BUFG	1	1
Total Power (W)	6.573	7.891
Fmax (MHz) (Maximum frequency of operation)	250	182

Table 4.5. Comparison results with literature for Sobel edge

Parameter	Proposed design	Ismail et al. [4.14]	Sanjay Singh et al. [4.15]
Operation Speed (MHz)	250	160	108
Percentage area reduction over benchmark (%)	50	Not available	40

Area calculation in table 4.5 assumes Gate count = (LUT count × 4)+ Flip-flops Tables 4.4 and 4.5 imply the suggested implementation utilizes 50% less area, is approximately 30% faster, and consumes 20% less power than the reference designs. The results can be further enhanced if we apply extra tool-specific optimization features such as pipelining and resource sharing. Such optimization options have already been outlined, in detail, in Chapter 3. Superimposing the functional simulation results of the implementation on top of golden benchmark results prove that optimization method has no impact on design functionality.

4.5 Harris corner detector design, implementation, and optimization

Harris corner detector is a commonly used algorithm in image processing and computer vision applications to extract corners and infer features in an image. It was first proposed by Chris Harris and Mike Stephens in 1988. Compared to previous architectures, this algorithm takes the differential of the corner score with reference to direction. Since it was invented, it has been improved by multiple researchers and implemented for different application designs.

4.5.1 Introduction to Harris corner detector application design

A fundamental problem while processing images for applications like filtering is corner detection. Section 4.4 discussed Sobel edge detection algorithm. A corner is another important characteristic or feature in an image. It is defined as a point where two edges meet.

Several algorithms exist to detect corners in images. The Moravec algorithm, the Susan algorithm, and the Harris corner detector are well-known examples of some commonly applied corner extraction methods [4.16, 4.17, 4.18].

The Harris corner detector is one of the most accurate corner detection algorithms. Its operation is remarkably straightforward. However, the algorithm is computationally intensive. Hence, it is an ideal candidate for optimized implementation on hardware like FPGAs.

4.5.2 Previous works for Harris corner detector implementation

Several authors have proposed novel studies on area and speed-efficient implementations of the Harris corner detector on FPGAs. Liu et al. recently introduced a method that can process RGB565 video in 640 × 480 resolution at a rate of 154 frames per second [4.19] . They implemented the design using a Xilinx ZedBoard. Xu et al. presented a slightly different algorithm by attaching a prefilter and using a simplified matrix rather than the original Gaussian kernel matrix [4.20]. In this way, the design complexity was reduced and headed to efficient hardware resource usage with Spartan 3 FPGA for robotics applications. By conducting the experiments using input images with a resolution of 256 × 256 at a processing time of 2.3 ms, Chao et al. tried to simplify the maximum suppression procedure with the Harris corner detector [4.21]. Their design, which was particularly developed for use with ZedBoard, reached a data rate of 144 frames per second in simulations. Lee et al. introduced a modified Harris corner detector for breast cancer detection from MRI and x-ray images [4.22]. They used an automated

adaptive radius suppression technique, decreasing corner clustering, consequently avoiding losing useful corners due to over-suppression.

4.5.3 Proposed implementation for Harris corner detection algorithm using ASBWIDN

A cascaded model of the Harris corner detector was generated in MATLAB. Input video frames had resolution 240 (width) × 320 (height) × 3 (colors) with eight bits representing each color, as presented in Figure 4.8. Because HDL implementation uses pixels instead of frames, the input frames were transformed to pixels and each pixel was sent as an input to the design per clock cycle. A Simulink library block was also concurrently employed for the Harris corner detector. The same input image was fed to the library block, and the results were consequently compared against the hardware implementation.

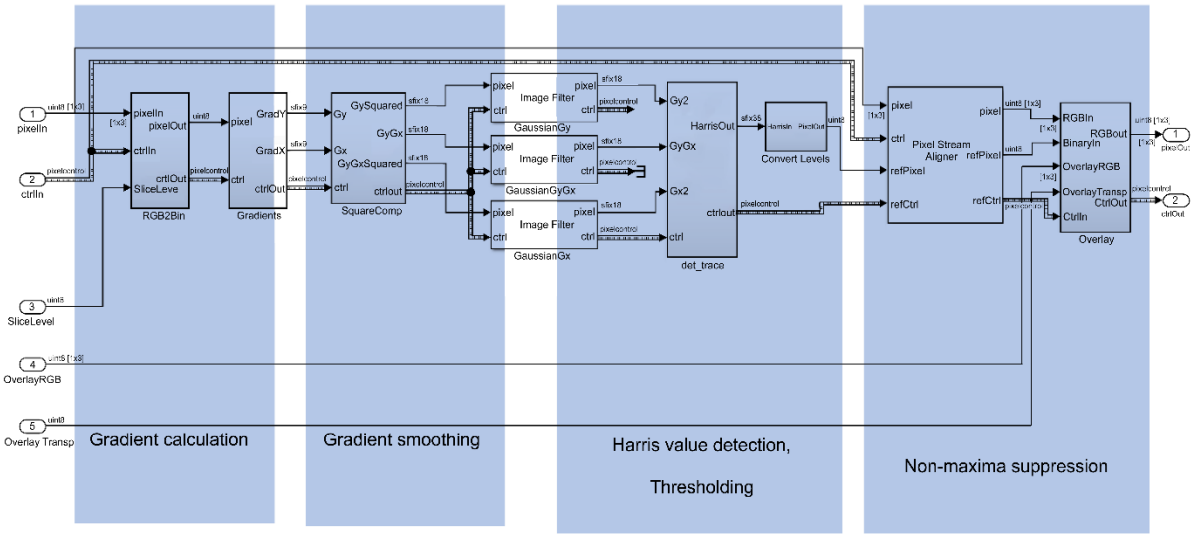

Figure 4.8. Cascaded stages of Harris corner detector.

Input video stream was chosen as set of RGB-888 images. During the simulation time of 100 seconds (video stream's duration), the absolute minima and maxima for the inputs, outputs, and intermediate nodes in the design were registered. These included all design nets, intermediate variables for storage, pixels, multiplier constants etc. This minima and maxima database was appended on subsequent simulation runs until all varied video signal input frames were processed. Afterward, all the signals' width was recalculated based on the range of the values for each input, output, and intermediate nodes. The HLS tool was then constrained to produce the

RTL with the updated widths for all the data nodes, primary inputs and outputs. The optimized RTL was then implemented on the FPGA. As the output from FPGA is in pixels, it was converted back to a frame image using the MATLAB function. The output image was a corner-marked image overlaid onto the input image. The entire model with HDL implementation, behavioral implementation, and the common image source is shown in Figure 4.9. As illustrated in Figure 4.9, delay elements were added to the behavioral and input image paths to balance the actual cycle's delay by accurate hardware implementation running on the FPGA. Figure 4.10 shows the input image, the output image from the MATLAB model, and the HDL implementation output on the FPGA. The input image source, the MATLAB behavioral model, and the FPGA HDL model ran independently and processed different frames from the input video stream.

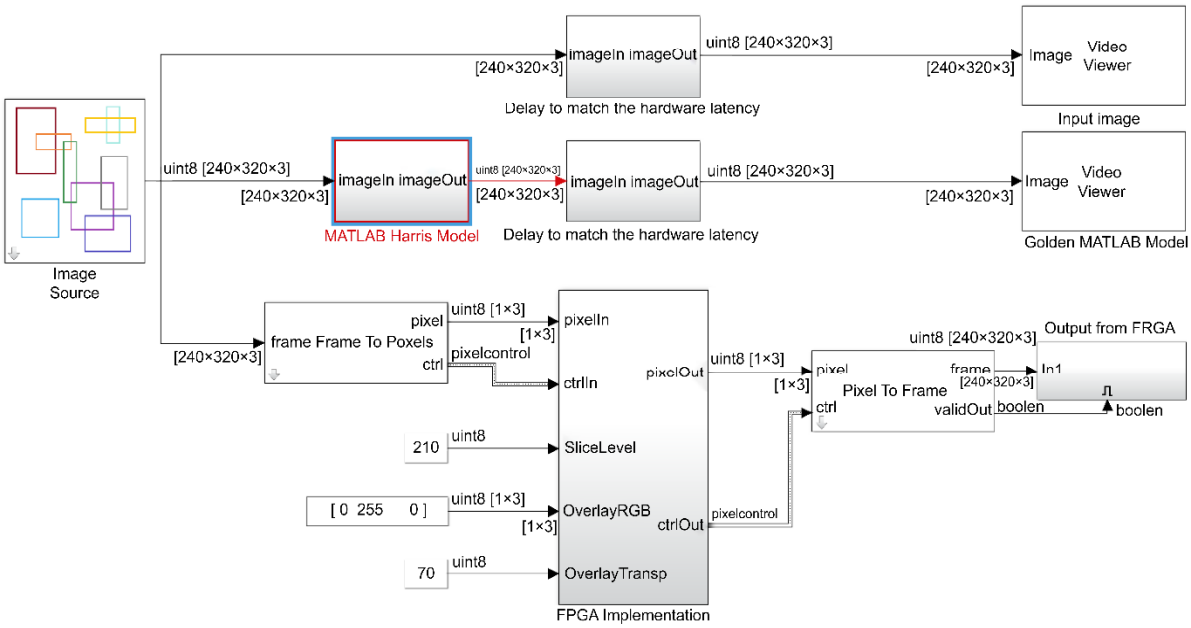

Figure 4.9. HLS model for the Harris corner detection algorithm.

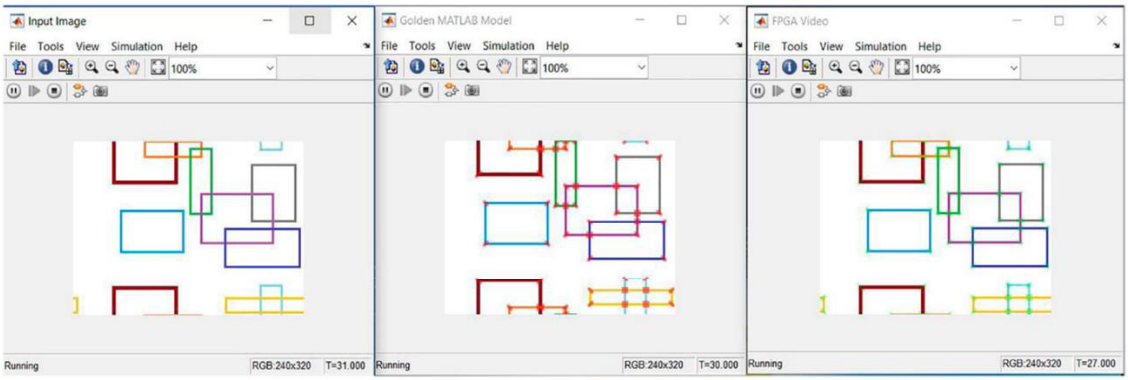

Figure 4.10. Input and output images from MATLAB and FPGA models.

The optimum datatype and data widths chosen for the model in RTL were back-annotated to the HLS model. Figure 4.11 shows the back-annotated datatypes for an intermediate stage (gradient calculation) in the model. In the highlighted example, two 18-bit signed fixed-point numbers generated a 36-bit result. In the pessimistic data width model, all widths were 32-bit, which is obviously suboptimal. This method is efficiently scalable for inputs, outputs, and internal signals that the design could include. For this study's case, each node's minima and maxima calculation method was completely automated and did not need any manual intervention. Moreover, all phases of the algorithm pictured in Figure 4.8 are completely pipelined. This leads to a better design throughput at the cost of a slightly increased number of flip-flops.

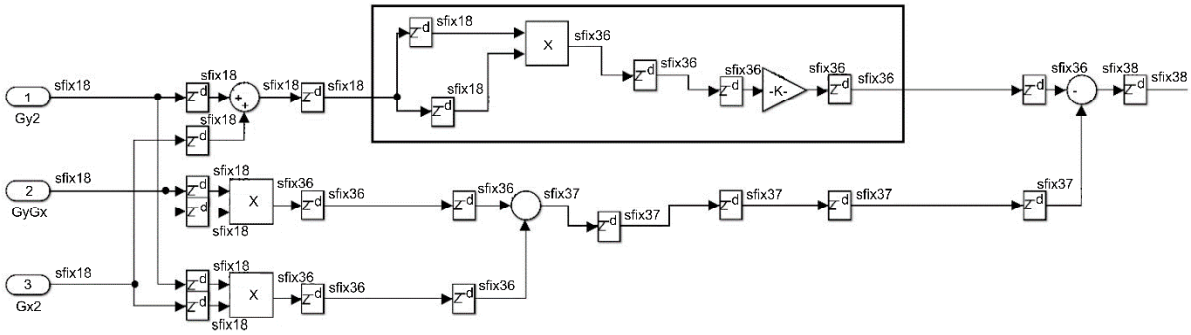

Figure 4.11. HLS model of Harris corner detector using application-specific bit widths.

4.5.4 Simulation, FPGA implementation results, and comparison

As explained in Section 4.5.3, the generated Verilog RTL code was simulated with a nonsynthesizable testbench using Vivado xSim software. Figure 4.12 presents the functional simulation results from the Vivado xSim simulator. As illustrated in the figure, the reference pixel values are identical to the proposed method's pixel output. They were also identical to high-level simulation results observed with MATLAB on the optimum bit width model. The quantization error which is introduced because of bit quantization from the double precision model was also measured. This was performed by applying the "FPGA-in-the-loop" feature of the HDL Verifier from MathWorks [4.12]. The measured RMS value of the quantization error was less than 1%. The same Verilog design was synthesized using Vivado 2019.1 software targeting the Xilinx ZedBoard(xc7z020clg484). Table 4.6 presents the synthesis and implementation results for the proposed design as well as the reference design (using pessimistic bit widths). The total power reported by the Vivado synthesis tool was 0.315 W, covering 0.205 W dynamic power and 0.110 W static power. As can be seen from Table 4.6, the resource

utilization is almost reduced by 50% as compared to reference design when the bit width optimization technique was applied.

Figure 4.12. FPGA simulation results of Harris corner detection algorithm.

Table 4.6. FPGA implementation reports for Harris corner detector (Zynq)

Resource	Proposed design	Benchmark design (Default bit widths)	Available Resources
LUTs	5917	10947	53200
LUT-RAMs	506	961	17400
Flip-flops	10981	17570	106400
BRAM	37	44	140
DSP	21	36	220
IOs	102	216	200
BUFG	1	1	32

Optimal design: Critical Path: 25.84 ns, Operation Freq. = 38.7 MHz, Total On-chip Power = 0.315 W

Benchmark design: Critical Path:41.6 ns, Operation Freq. = 24 MHz, Total On-chip Power = 0.513 W

Table 4.7. Comparison of results with the literature for Harris corner detector (Zynq)

Resource	Proposed design	Komorkiewicz et al. [4.23]	Liu et al. [4.19]	Chao et al. [4.21]
Gate Count	22815	51859	37447	28626
LUTs	5917	20660	17555	9485
LUT-RAMs	506	NA	5443	4131
Flip-flops	10981	10539	2337	9656
BRAM	37	77	75	64
DSP	21	59	55	110
IOs	102	NA	48	NA
BUFG	1	NA	7	NA
Frame rate (fps)	168	120	154	98

The proposed algorithm's synthesis results were also compared with other implementations published in the literature [4.19, 4.21, 4.23].

The results of this comparison are compiled in Table 4.7. The implementation introduced in this study is better in terms of frame rate and the area used for the same target FPGA than the other implementations. Concerning area, even with the assumption that a LUT uses two ASIC gates, proposed implementation utilized almost 50% fewer resources than Komorkiewicz et al., 30% fewer resources than Liu et al. and 20% fewer resources than Chao et al. [4.23, 4.19, 4.21]. Moreover, the proposed implementation has better performance regarding the frame rate than other implementations due to reduced design complexity. This is because critical paths are shorter and hence a better operational frequency is observed on the FPGA. Other researchers have not shared their data on power dissipation for designs.

4.6 Digital down converter for software defined radio applications: design, implementation, and optimization

This section of the thesis discusses the results achieved by our novel HLS design methodology, ASBWIDN, applied to a digital down converter design for software defined radio (SDR) applications**.**

4.6.1 Introduction to digital down converter design

In the processing of digital signals, digital down converters (DDCs) transform digitized, band-limited signals to lower frequency signals at a lower sampling rate to simplify subsequent filtering phases. DDCs with input signal bandwidths above 1 MHz are known as wideband DDCs, whereas those with less than 1 MHz bandwidth are known as narrowband DDCs. Wideband DDCs are commonly used in applications such as 4G and satellite communications. On the other hand, narrowband DDCs have applications in commercial broadcasting. DDCs are extensively used in modern communication systems, such as Software Defined Radio [4.24, 4.25]. Due to high data rate requirements for modern communication systems, high-speed and area-efficient implementation of these systems on target FPGAs is the need of the hour.

The DDC we chose for application down-converts an input signal of 200 MHz to an output signal of 2 MHz. This implementation was done on FPGA hardware (Xilinx Kintex-7) and checked against the design specifications using the FPGA-in-the-loop feature of HDL Verifier and MATLAB [4.12].

4.6.2 Previous works for Digital down converter implementation

Numerous authors have proposed novel works on efficient DDC architectures for targeting low-area and high-speed FPGA implementations. Most have been implemented in HDL languages such as Verilog or VHDL. Debarshi Datta recently introduced an implementation which down converts 70 MHz to 130 kHz using three filter stage filtering [4.26]. Their design used a coordinate rotation digital computer (CORDIC) processor as an oscillator, accompanied by a multistage decimation filter implemented on a Kintex-7 FPGA board. In 2017, Liu et al. introduced a reconfigurable DDC for down-converting input bandwidths from 3.6 GHz to an output range of 1 KS/s–225 MS/s [4.27]. They achieved optimum hardware resources with a

variable sample rate on a Xilinx Kintex-7 device. G Maiti et al. introduced a variable fractional rate DDC for satellite communication. Their implementation, which was experimented on Kintex-7, enables a user to dynamically change the carrier frequency and output sample rate at runtime [4.28]. Mingshao et al. introduced an optimization technique for DDCs on FPGAs for pulse radar receiver applications. They intended to optimize by using the intrinsic relationship for FIR filters by implementing it utilizing a distributed algorithm. They also introduced an IP core for executing a 32K point FFT required for RADAR applications [4.29] . Obradovic et al. discussed implementing DDC on a Xilinx Kintex-7 device for wideband direction-finding applications. They presented increasing the instantaneous bandwidth using four DDCs per channel of the direction finder. They applied Xilinx IP cores to integrate DDC chain elements, such as Numerically Controller Oscillator (NCO), Cascaded Integrator Comb, and Labview FPGA module, for data acquisition and synchronization between various channels [4.30].

The latest work in this field by Debarshi et al. applied a CORDIC-based implementation for a local oscillator, which uses a smaller LUT count, thereby saving area resources [4.26].

4.6.3 Proposed implementation

A block diagram of the designed DDC is shown in Figure 4.13. It comprises of a bandpass filter that passes the intermediate frequency signal. The analog input signal is then digitized at a particular sampling rate. A numerically controlled oscillator, which is tuned to the input signal, produces a local signal which is multiplied by the analog to digital converter's output. In our design implementation, the NCO produces a multichannel, complex sinusoidal signal, with an independent phase and frequency in each channel. The signal amplitude is the unity generated using a LUTs with 16385 data points and a spurious-free dynamic range of 108 decibel (dB). Lastly, an internal dither of 15 bits was added to randomize the quantization error.

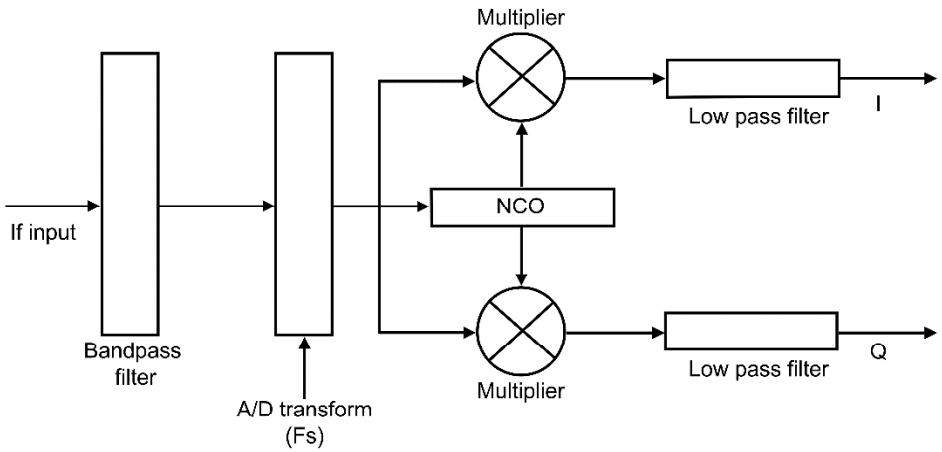

Figure 4.13. Block diagram of a digital down converter.

Multiplication by the NCO output utilizes the fact that the product of two sinusoidal waves (i.e., the input sinusoid and the NCO output) generates two signals, as shown using the following simple example:

$$\sin(f_1 t)\sin(f_2 t) = \frac{1}{2}\{\cos[(f_1 - f_2)t] - \cos[(f_1 + f_2)t]\}$$. (Equation 4.1)

Hence, the product of two sinusoidal signals comprises two components, one at the sum of the two frequencies and one at the difference of the two frequencies. The sum frequency component, i.e., the high-frequency component, is attenuated by a low pass filter in the DDC. Consequently, only the difference frequency component is important in the design's next phase. In Eq. (4.1), $\cos[(f_1 - f_2)t]$ is the low frequency component and $\cos[(f_1 + f_2)t]$ is the high-frequency component.

The same method outlined in Section 4.2 was applied for designing the DDC. As part of the DDC design method, which was produced using HLS herein, a MATLAB model was constructed for the DDC using the highest permitted bit width for all input, output, and intermediate nodes in the design, i.e., double-precision. The input signal was a chirp centered at 200 MHz. The absolute minima and maxima values for each of the inputs, outputs and intermediate nodes in the design were recorded during the simulation over the entire input frequency range. The output signal was a chirp centered at 0 with a 30 dB bandwidth of 2 MHz, as displayed in Figures 4.14 and 4.15 (time and frequency domain response of the DDC, respectively). The design model, which applies double-precision or floating-point data types (as previously explained) for all nodes of the design, is portrayed in Figure 4.16. The minima and maxima database was annexed on subsequent simulation runs until all the varied chirp signal inputs were processed. Afterward, width for all the signals was computed based on the range of values for each input, output, and intermediate nodes. The HLS tool was then constrained to produce the RTL with the updated signal width constraints for all the intermediate signals, inputs, and outputs.

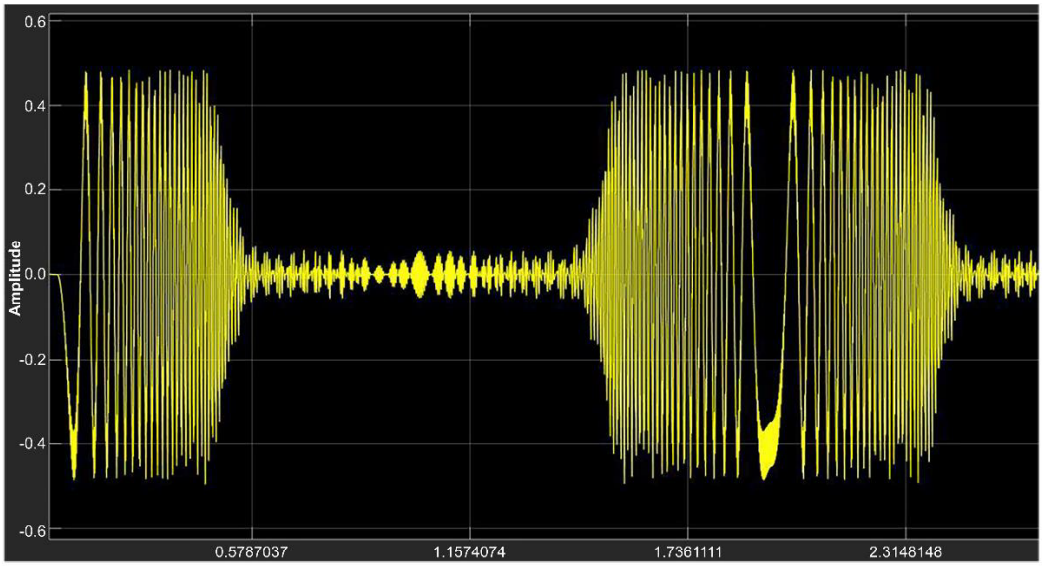

Figure 4.14. Time domain response for the digital down converter.

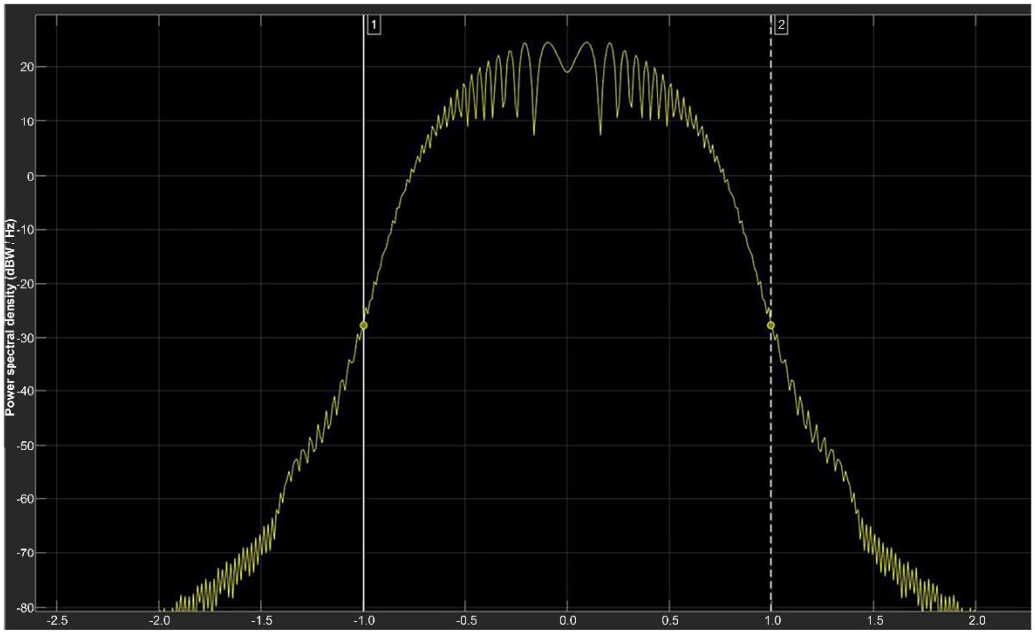

Figure 4.15. Frequency domain response for digital down converter.

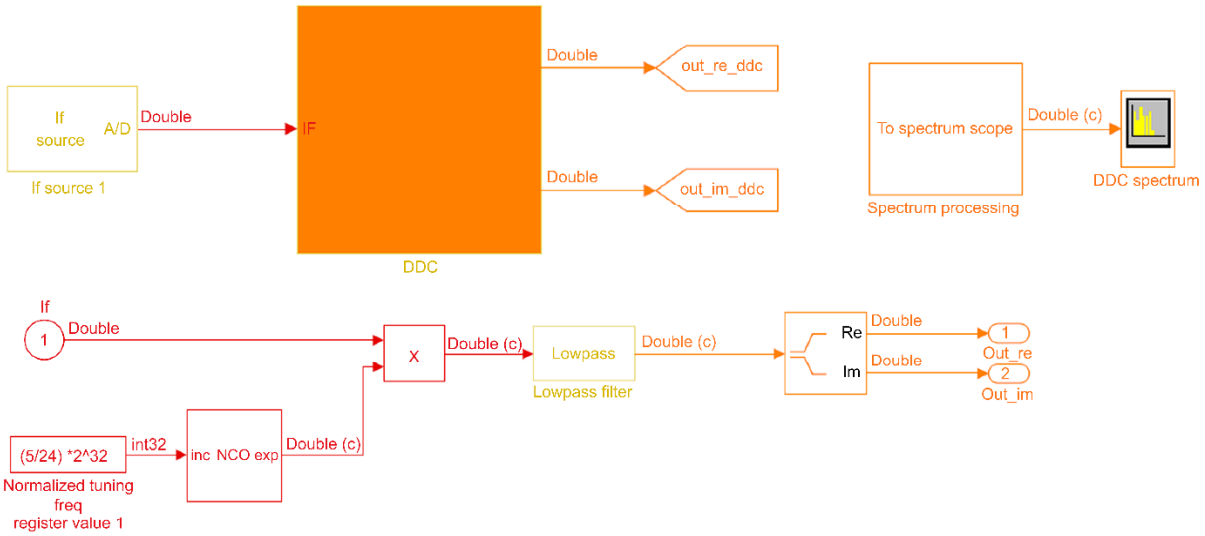

Figure 4.16. Digital down converter using floating-point data types.

For example, an input signal sin(x) or cos(x) must extend between −1 and 1; therefore, more bits are needed to express the fractional part than the integer part. Therefore, double-precision could be substituted with a signed representation, with one bit for the integer part, one bit for the sign part, and four or five bits for the fraction depending on the needed accuracy. The "optimum" bit width model using fixed-point data types is depicted in Figure 4.17.

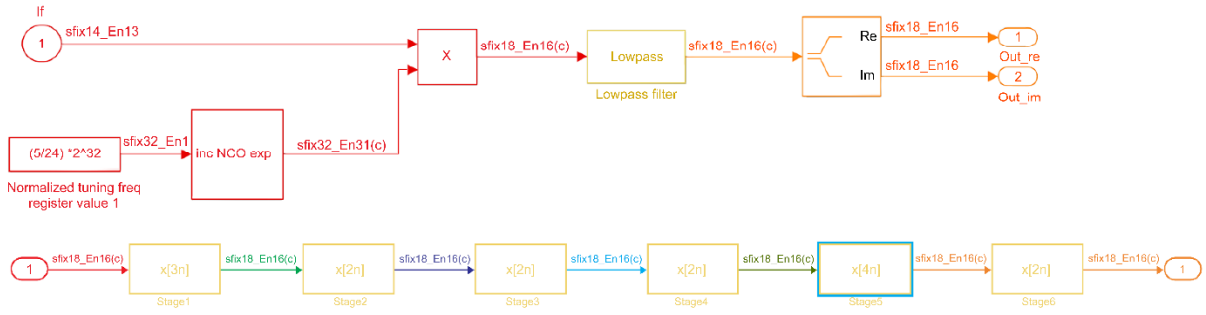

Figure 4.17. Fixed-point (optimal bit width) implementation of digital down converter.

4.6.4 Results and comparison with the literature

For the functional simulation of the DDC (design under test), an input signal was chosen having a unity magnitude, bi-directional sinusoidal chirp centered at 200 MHz with a bandwidth of 20 MHz.

The code produced from the HLS model with stimulus-aware bit widths was implemented on the Xilinx XC7K70T-FBG676 target device with a speed grade of 3 using Vivado 2019.1 software. Verilog non-synthesizable testbench was also produced from the same model, and

simulation was run on Vivado xSim. The functional simulation results acquired from the Vivado xSim simulator were found to be equal to the high-level simulation results acquired using MATLAB on an optimum bit width model.

Figure 4.18 shows the RTL simulation results. Additional to the output signal's spectrum, the quantization error was also measured. Such error was included due to the selection of reduced "optimal" signal widths. Next, it was compared against the golden double-precision model running in MATLAB. This was achieved using FPGA-in-the-loop feature of HDL Verifier from MathWorks [4.12]. The measured Root Mean Square value of the quantization error was less than 1%. The superimposed output results of the DDC running on MATLAB and the same design running on the FPGA, are presented in Figure 4.19. As displayed in the figure, the two output plots (yellow and blue) almost overlap, showing a minimal quantization error.

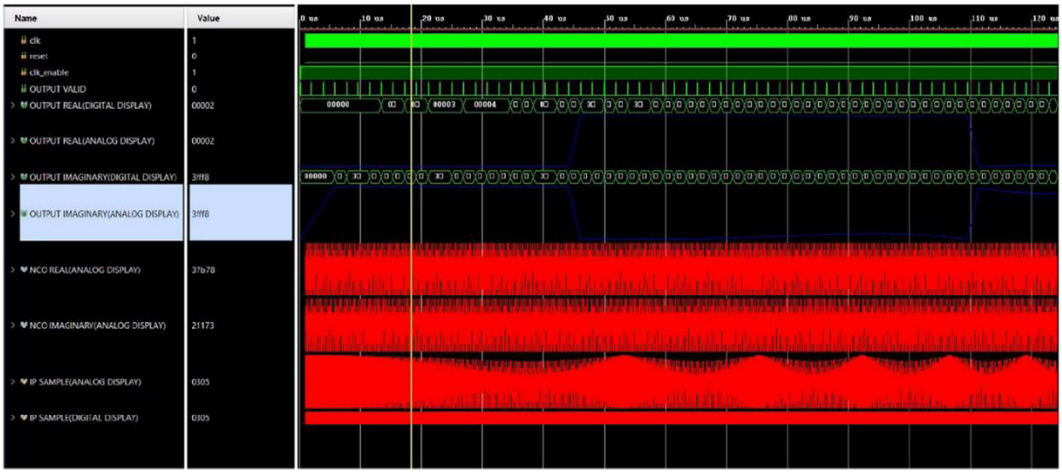

Figure 4.18. RTL Simulation results for digital down converter.

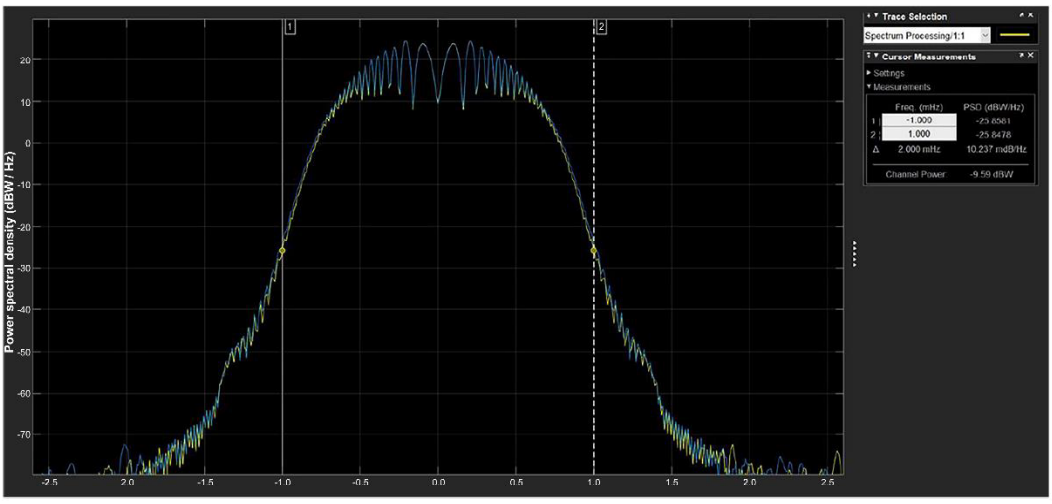

Figure 4.19. Frequency domain response for DDC running on FPGA superimposed on MATLAB model output.

Table 4.8 shows the implementation results for the same design targeting Xilinx Kintex 7 FPGA device. We also compared results against a reference design that was generated without using optimization methodology but using the HLS method. The reference design corresponds to double-precision (pessimistic bit width is chosen for the design as presented in Figure 4.16). As Table 4.8 shows, resource utilization, power dissipation as well as runtime frequency is superior for the optimized bit width implementation as compared to the reference implementation (with pessimistic bit widths)

Table 4.8. Kintex 7 implementation results for the digital down converter

Resource	NCO	Filter (six-stage CDC)	Total utilization (Optimized)	Total utilization (Benchmark)
LUTs	261	1567	1828	38122
Flip-flops	70	2183	2253	4373
DSP	2	147	149	240
IO	12	56	68	106
BUFG	0	1	1	1

Optimized implementation: Critical Path: 2 ns, Operation Freq. = 498 MHz, Total On-chip Power = 0.325 W

Reference implementation: Critical Path: 2.19 ns, Operation Freq. = 456 MHz, Total On-chip Power = 0.601 W

Table 4.9. Comparison of results with literature for digital down converter (Kintex 7)

Resource	Proposed design	Full Precision	Debarshi et al. [4.26]	Liu et al. [4.27]	Vuk et al. [4.30]
Gate Count	5909	80617	12968	28090	209610
LUTs	1828	38122	5543	7269	37066
Flip-flops	2253	4373	1882	13552	135478
DSP	149	240	230	83	1034
Fmax (MHz)	498	456	495	455	NA
Power (W)	0.325	0.601	1.022	1.466	NA

The total power reported by the Vivado synthesis tool is 0.325 W, containing a dynamic component of 0.243 W, and static component of 0.082 W.

The synthesis results for the proposed implementation (design corresponding to Figure 4.17) were compared against other implementations available in the literature [4.26, 4.27, 4.30]. The results of the comparison are summarized in Table 4.9. In all four cases, the same target hardware and synthesis tool are used.

As is evident from Table 4.9, Figure 4.18 and Figure 4.19, the proposed implementation has almost the same throughput and functionality as those described by other researchers. The presented implementation outperforms in terms of area and power usage. Regarding area, even if we consider a LUT applying two ASIC gates, the proposed implementation uses almost 55% fewer resources (5909 = 45% of 12968) and 68% less power than the one proposed by Debarshi et al. (0.325 = 32% of 1.022). Nevertheless, this implementation requires a slightly higher flip-flop count than the one proposed by Debarshi et al. The higher flip-flop count is attributed to the use of a ROM Table rather than CORDIC for storing the sine samples for NCO, where each sample is 16 bit wide, unlike those used by Debarshi et al., where a CORDIC implementation translated in an improvement in terms of flip-flop count [4.26]. To summarize, given the use of smaller bit widths, the proposed implementation is faster and demands fewer resources as well as less power. Furthermore, Table 4.9 clearly shows that the proposed implementation results are superior than those reported by Liu et al. and Vuk et al. [4.27, 4.30].

4.7 Advanced encryption standard for automotive applications – design, implementation, and optimization

In Section 3.4, we presented the significance of optimal implementations of encryption algorithms like the advanced encryption standard (AES) on FPGAs for telecom applications. We also presented the optimization results achieved for the same using multiple HLS directives. In this section, we use the same design for automotive applications to prove the efficacy of the novel design methodology for HLS tools as described in Section 4.2. For this application, Vivado HLS is used as the design platform.

4.7.1 Introduction to AES

Following the introduction of electronics in the automotive industry in the 1970s, there has been a constant increment in the complexity of microcontrollers used in vehicles. For instance, with the introduction of vehicular ad-hoc networks at the beginning of the 21st century, automobiles became equipped with various advanced wireless communication technologies, such as vehicle-to-vehicle, vehicle-to-infrastructure, and vehicle-to-cloud communications [4.31].

Such communication standards support modern automotive microcontrollers in applications like bluetooth connectivity, telematics, audio-visual controls, object detection, and pedestrian detection. This enables the industry paradigm shift towards "connected" smart cars. Because of such car connectivity, numerous microcontrollers in modern vehicles are also being applied to simplify financial transactions at gas stations, parking lots, toll booths, etc. However, high connectivity has several drawbacks, including increased chances of data theft and personal information hacking. In automotive applications, illegal access to private data could also lead to safety infringements, thus putting lives and materials at risk [4.32]. Designers of automotive microcontrollers require to predict security attacks and restrict unauthorized access to electronic systems and data. In this vein, the automotive industry has developed interoperable standards for vehicular data security. One way of securing data is cryptography, i.e., encrypting messages before transmission so that only the designated receiver can decrypt important messages.

The AES, also known as Rijndael, is one such electronic data encryption standard. It was established, in 2001, by the United States department of commerce's National Institute of Standards Technology [4.33]. Consider the encryption of an inter-vehicle communication message or the authentication of a user application to be placed in shared memory space, then AES is a very commonly used algorithm in autonomous vehicles. Nevertheless, software based implementations of this algorithm fall short of the required speed and performance. For example, on a 1.2 GHz Intel processor, a throughput of 400 Mbit/s (or 50 MByte/s) is possible. The fastest known implementation on a 64-bit Athlon CPU provides a theoretical throughput of more than 1.6 Gbit/s [4.34]. Hardware-based implementations are faster, more energy efficient, and considerably more secure.

4.7.2 Previous works for AES algorithm implementation

Various authors have presented optimal hardware AES algorithms' implementations on FPGAs. In 2015, Soltani and Sharifian introduced an ultra-high throughput and fully pipelined AES

implementation in the counter (CTR) mode of operation [4.35]. Using multiple optimization techniques, such as loop unrolling and pipelining, they implemented the design on a Virtex-6 (XC6VLX240T-FF784-3) Xilinx FPGA. In 2018, Zhang et al. introduced a fully unrolled three-stage inner and outer double-pipelined architecture to enhance data throughput [4.36]. Their implementation was targeted on the Virtex-6 platform, and it achieved a throughput of 60.29 Gbps. In the same year, Smekal et al. presented a comparative study of two algorithms: AES and TwoFish [4.37]. They implemented the AES algorithm on both Virtex7 and Ultrascale+ FPGAs from Xilinx. They achieved a throughput of 49.38 and 49.66 Gbps, respectively. In 2019, Noorbasha et al. introduced a slightly modified version of the AES algorithm using a triple key. They implemented this version on Spartan 3E FPGA kit [4.38]. Their version offered increased security, even though lesser throughput was obtained. In 2019, Chen et al. presented the AES implementation using deep pipelining, obtaining a data rate of 31.3 Gbps for big data applications [4.39].

4.7.3 Proposed implementation using ASBWIDN

As part of the design method for the AES algorithm implemented in this work, a Vivado HLS model was built for the AES algorithm. This model had the highest permissible bit width for all input, output, and intermediate nodes, i.e., double-precision arithmetic. The input signal was random streaming data (corresponding to the text to be encrypted), i.e., plain text. The absolute minima and maxima for each of the inputs, outputs and intermediate nodes of the design were recorded during the simulation over the entire input plaintext range. The output signal at any given point was a ciphertext corresponding to the plain text. Figure 4.20 shows the pseudo-code, which uses double-precision or floating-point data types (as explained before) for all nodes of the design. As seen in the algorithm, each of these blocks had a length of 16 bytes (128 bits), and 10 rounds were executed. The minima and maxima database was appended following subsequent simulation runs until all the input message bytes were encrypted according to the design.

```
memcpy(state,iv,16);

AddRoundKey(state, expandedKey + (16 * 0));

L_rounds: for (unsigned short i = 0; i < 10; i++) {
        SubBytes(state);
        ShiftRows(state);
        if (i != (9)) {
                MisColumns(state);
        }
        AddRoundKey(state, expandedKey + (16 * (i + 1)));
}


AddRoundKey(state,newState);

memcpy(ciphertext,state,16);
```

Figure 4.20. Psuedo-code for AES algorithm using floating-point data types.

Subsequently, all the signal widths were recalculated based on the range of values for each input, output and intermediate nodes. The HLS tool was then constrained to produce the RTL with the updated signal widths for all nodes. For example, assume none of the input values exceed 12 bits for a multiplier in the byte substitution layer. Therefore, a 12-bit implementation (optimized) could have a lesser delay than a 32-bit multiplier, which is inferred if the datatype is an integer by default (pessimistic) keeping the functionality intact. Hence, the system can accomplish higher throughput and data rate on the target FPGA.

4.7.4 Results and comparison with literature

The functional simulation results achieved from the Vivado HLS model matched with the AES algorithm's theoretical results with the same set of stimuli obtained using MATLAB simulation.

Figure 4.21 displays the AES encryption results for the plain text "HELLO WORLD THIS IS A SECRET TEXT." The figure also shows the counter value incrementing, which is a typical characteristic of the CTR mode. In this case, because there are 32 characters in the string, including the spaces, the algorithm runs twice (16 bytes = 128 bits), processing 16 characters in each run. Hence, the final counter value is two. The simulation results acquired from the optimum bit width model and double-precision model (double-precision data type was used for all the intermediate nodes) were identical. This proves that when the proposed method is used for RTL generation and downstream synthesis, the functionality remains intact. The HDL code produced from the optimized HLS model with stimulus-aware bit widths was implemented on

the Xilinx Kintex 7 (XC7K70T-FBG676 target device with a speed grade of three) using Vivado 2019.1 software. The code then was checked for functionality using the xSim simulator and a hand-written nonsynthesizable Verilog testbench. Because some of the available implementations in the literature were on Xilinx Virtex 6 target boards, the same HDL design was also implemented on Virtex 6 device (XC6VLX240T) using iSE 14.7 software for a direct and fair comparison.

```
AES with 10 rounds and 16 bytes ciphertext
Mode of Operation = CTR
key = 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F

expandedKey = 0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF 0xD6
0xAA 0x74 0xFD 0xD2 0xAF 0x72 0xFA 0xDA 0xA6 0x78 0xF1 0xD6 0xAB 0x76 0xFE 0xB6 0x92
0xCF 0xB 0x64 0x3D 0xBD 0xF1 0xBE 0x9B 0xC5 0x0 0x68 0x30 0xB3 0xFE 0xB6 0xFF 0x74
0x4E 0xD2 0xC2 0xC9 0xBF 0x6C 0x59 0xC 0xBF 0x4 0x69 0xBF 0x41 0x47 0xF7 0xF7 0xBC
0x95 0x35 0x3E 0x3 0xF9 0x6C 0x32 0xBC 0xFD 0x5 0x8D 0xFD 0x3C 0xAA 0xA3 0xE8 0xA9
0x9F 0x9D 0xEB 0x50 0xF3 0xAF 0x57 0xAD 0xF6 0x22 0xAA 0x5E 0x39 0xF 0x7D 0xF7 0xA6
0x92 0x96 0xA7 0x55 0x3D 0xC1 0xA 0xA3 0x1F 0x6B 0x14 0xF9 0x70 0x1A 0xE3 0x5F 0xE2
0x8C 0x44 0xA 0xDF 0x4D 0x4E 0xA9 0xC0 0x26 0x47 0x43 0x87 0x35 0xA4 0x1C 0x65 0xB9
0xE0 0x16 0xBA 0xF4 0xAE 0xBF 0x7A 0xD2 0x54 0x99 0x32 0xD1 0xF0 0x85 0x57 0x68 0x10
0x93 0xED 0x9C 0xBE 0x2C 0x97 0x4E 0x13 0x11 0x1D 0x7F 0xE3 0x94 0x4A 0x17 0xF3 0x7
0xA7 0x8B 0x4D 0x2B 0x30 0xC5


Block 1
plaintext = H E L L O   W O R L D   T H I S  <=> 48 45 4C 4C 4F 20 57 4F 52 4C 44 20
54 48 49 53

iv = 92 E2 F3 D4 A5 B6 47 48 0 0 0 0 0 0 0 0

ciphertext = 97 A4 62 F1 A4 D1 3C FD 41 3B E2 C5 54 2F 15 60

Block 2
plaintext = I S   A   S E C R E T   T E X T  <=> 49 53 20 41 20 53 45 43 52 45 54 20
54 45 58 54

iv = 92 E2 F3 D4 A5 B6 47 48 0 0 0 0 0 0 0 1

ciphertext = 8C 57 86 AB 70 F1 75 9F 3D 62 33 AE BB C6 C9 FF
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Figure 4.21. HLS simulation results for the AES encryption algorithm.

Table 4.10 shows the synthesis results for the proposed design along with the synthesis results for the design with pessimistic (non-optimized) bit widths. It is clear from the table that the proposed implementation is superior than the reference design. Table 4.11 shows a comparison against other implementations available in the literature.

Table 4.10. Proposed AES implementation on Kintex 7 FPGA

Resource	Proposed design	Benchmark design (Default bit widths)	Available Resources
LUTs	577	632	41000
LUT-RAMs	8	12	13400
Flip-flops	449	514	82000
BRAM	4	6	135
IO	265	284	300
BUFG	1	1	32

Optimized: Throughput = 38.05 Gbps, Operation Freq. = 298 MHz, Power = 0.114 W

Benchmark: Throughput = 32.4 Gbps, Operation Freq = 253 MHz, Power = 0.234 W

Further, calculated energy dissipation, E = P × T, where P is power dissipation and T is time required for one cycle of AES for 128bits. Since throughput is 38.05 Gbps, T = (1/38.05) × 128 ns. Hence, energy dissipation, E = (1/38.05) × 128 × 0.114 = 0.383 nJ (Nano Joules) for the optimized implementation.

Table 4.11. AES algorithm implementation result comparison with the literature

FPGA results	Proposed design	Soltani et al. [4.35]	Zhang et al. [4.36]	Chen et al. [4.39]
Virtex 6 (MHz)	562.108	508.104	470.998	N/A
Virtex6 Throughout (Gbps)	287.8	260.15	60.29	N/A
Kintex 7 (MHz)	297.3	N/A	N/A	244.5
Kintex7 Throughput (Gbps)	38.05	N/A	N/A	31.30

As is clearly evident from Table 4.11, the proposed implementation has achieved a better throughput when compared to others available in the literature. The accomplished throughput is approximately 10% better than that of Soltani et al., 20% better than that of Zhang et al. and 20% better than that of Chen et al. [4.35, 4.36, 4.39]. The energy dissipation is not reported by other researchers and hence a direct comparison was not done. Another point to note is that the proposed design is proven to be superior via optimized RTL generation and is independent of the FPGA technology target.

4.8 Concluding remarks

In this chapter, we proposed a novel High level synthesis design optimization methodology which is tool, application and FPGA platform independent. The automated method removes bit width pessimism for inputs, outputs, and intermediate nodes in the design and helps in generating optimal, technology independent RTL code. It helps to optimize the final FPGA implementation by constraining the downstream synthesis tool. As the optimization happens at RTL level (higher abstraction), it is applicable to all FPGA targets. We have also used the optimization method on two tools - MATLAB HDL coder and Vivado HLS. The novel HLS methodology was used to optimize 5 different unique designs from different real-life application areas. We also presented comparison of the optimized results with other benchmark implementations done by other researchers for the same applications. To prove the efficacy of the optimization method, comparison of the results for same applications without using the optimization methodology (pessimistic bit widths) is also presented. For all the design applications, RTL simulation using xSim as well as FPGA- in-the-loop simulation confirmed that optimizations achieved were benign for design functionality.

As the spectrum of applications is large and proposed results are superior than presented in literature for each of the application designs, we can claim with a high confidence that the methodology would give superior results with other application designs as well. In the next chapter, the results obtained by applying this method on another application design used in autonomous driving systems is presented.