Chapter 3

Optimization techniques based on HLS directives

Chapter 2 discussed multiple HLS optimization techniques available for the area, speed, and power optimization of designs. In this chapter, some of the applications that were created using MATLAB HDL coder and Vivado HLS are discussed. Some of the available directives from the tool vendors were used to optimize the designs and results compared against those available in the literature. Some of the commonly used directives are described in Appendix. The comparison is done against previously published results obtained by other researchers. This chapter discusses five such applications: QPSK modulator, DSP filter, MIPS processor core, AES encryption algorithm and you-only-look-once (YOLO) v2 deep learning algorithm. Section 3.1 – Section 3.3 discuss designs and optimizations based on MATLAB HDL coder directives (Band pass DSP filter, QPSK modulator, MIPS processor core). Section 3.4 and 3.5 discuss design and optimization using Vivado HLS (AES encryption algorithm and YOLO v2 algorithm). The tools are chosen for these design applications depending on the availability of library blocks and ease of coding. The metrics used to compare the design implementations are – maximum frequency of operation (function of largest critical path), area utilization (function of LUTs, Flip-flops and registers used for the implementation) and power utilization (sum of dynamic and static power dissipation). Dynamic power is a function of power consumed during design operation and depends on factors such as functional resources (logic blocks, PLLs, registers, etc.) and signal toggle rates. Static power is the power dissipated by the device even when there is no design running on the FPGA. It is a function of leakage current, sub threshold leakage, junction leakage etc.

3.1 Bandpass digital signal processing filter design (HDL coder)

Herein, design, implementation and optimization of a band-pass DSP filter is discussed using MATLAB HDL coder.

3.1.1 Introduction to DSP filters

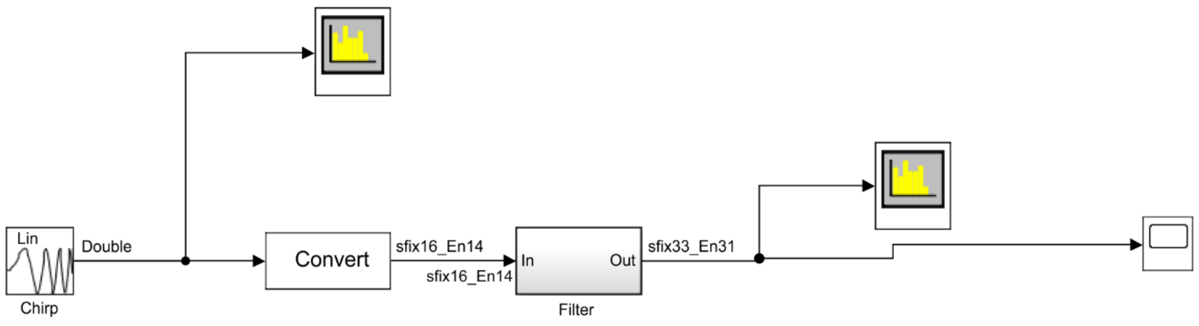
DSP filters are an integral part of multiple signal processing circuits used in communications, image processing, etc.; thus, achieving their optimal FPGA implementations are crucial in VLSI designs. The DSP filters help in mathematical manipulation of information signals such as images, audio etc. These filters are gaining importance in multiple application areas over conventional analog filters because digital signal processing is more immune to noise. Moreover, digital signals can be easily reproduced in large quantities with lesser storage cost for data.

3.1.2 Design method and FPGA implementation results

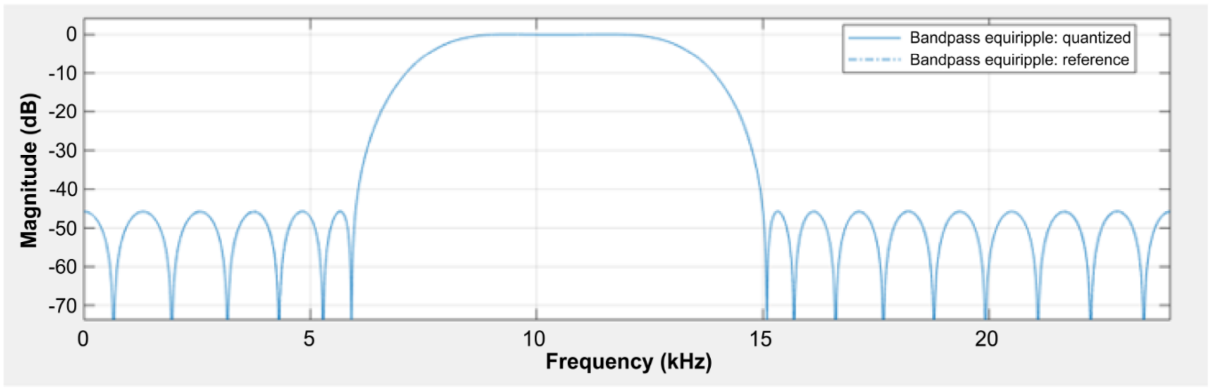
Using MATLAB and Simulink, a bandpass FIR filter was designed and simulated [3.1]. The following characteristics were chosen for the study:
Sampling Frequency = 48 kHz; Lower stop band frequency (Fstop1) = 6 kHz; Lower pass frequency (Fpass1) = 9 kHz; Higher pass band frequency (Fpass2) = 12 kHz; Higher stop band frequency (Fstop2) = 15 kHz.

Sampling frequency of 48 kHz is chosen to be sufficiently higher than Nyquist frequency of 30 kHz (2 × 15 kHz) to avoid aliasing. This sampling frequency of the filter gets mapped to the fastest design clock frequency achieved on the target FPGA device.

The filter was designed using the filter designer app available with MATLAB and exported the same to Simulink. The filter coefficients were quantized for HDL implementation using fixed-point designer tool [3.2]. The design and simulation environment for the fixed-point HLS model is shown in Figure 3.1. Time taken for a single linear chirp is 33.66 micro seconds. MATLAB HDL coder and multiple HLS directives were used as explained in Chapter 2, to optimize the implementation [3.3]. Implementation results are shown in Tables 3.1 and 3.2. Table 3.1 shows the implementation results for the bandpass filter RTL code targeted for Kintex 7(XCK770T-FBG676) FPGA generated using the MATLAB and Simulink model. Table 3.2 shows a similar table for hand-coded RTL for a filter with the same functional specifications. The functional simulation results as well as FPGA-in-the-loop simulation results suggest that functionality of the design remains the same when using HLS. But since a higher platform frequency is seen (without seeing any timing violations), the total wall clock time for filtering operation is reduced as compared to hand-coded RTL implementation.



(A)



(B)

Figure 3.1. (A) Fixed-point bandpass filter design (B) Simulation in MATLAB and Simulink.

Table 3.1 Kintex 7 FPGA implementation results for BP filter (HLS optimized)

Resource	Utilization	Available	Utilization (%)
LUTs	2565	41000	6.256
Flip-flops	1280	82000	1.560
DSP	140	240	58.333
IOs	106	300	35.333
BUFG	1	32	3.125

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

Table 3.2 Kintex 7 FPGA implementation results for BP filter (hand-coded RTL)

Resource	Utilization	Available	Utilization (%)
LUTs	3268	41000	7.970
Flip-flops	1167	82000	1.423
DSP	151	240	62.916
IOs	106	300	35.333
BUFG	1	32	3.125

Critical Path: 29.68 ns, Operation Freq. = 33 MHz, Total On-chip Power = 0.467 W

As is evident from Tables 3.1 and 3.2, owing to the usage of distributed pipelining and multiplier sharing HLS directives (as explained in Chapter 2), better frequency of operation was achieved for the design created using HLS [3.3]. Even though a better frequency of operation was achieved using directives, a notable observation is that due to distributed pipelining, the flip-flop usage in the design created using HLS is slightly higher than the design created using hand-coded RTL. Further, since the directives of multiplier sharing was used, number of multipliers used was reduced and hence the implementation used lesser power on the target FPGA. It may be noted that one has the option to optimize the hand-coded implementation as well but the goal here is to demonstrate superior or comparable implementation and simulation results with shorter design cycle time using HLS.

3.2 QPSK modulator design and implementation (HDL coder)

Herein, design, implementation and optimization of a QPSK modulator is discussed using MATLAB HDL coder.

3.2.1 QPSK modulator introduction

QPSK modulation is a class of Phase Shift Keying. In QPSK, by choosing one of four possible carrier phase shifts: 0, 90, 180, or 270 degrees, two bits are modulated at once. Using the same bandwidth, **QPSK** provides the signal to carry twice as much information than ordinary PSK [3.4]. The QPSK modulator employs a bit-splitter, two multipliers with a local oscillator, a 2-bit serial to parallel converter, and a summer circuit.

The message signal's even bits (i.e., 2nd bit, 4th bit, 6th bit, etc.) and odd bits (i.e., 1st bit, 3rd bit, 5th bit, etc.) are separated, at the modulator's input, by the bits splitter, and they are multiplied with the same carrier to produce odd BPSK (called as $\mathbf{PSK_I}$) and even BPSK (called as $\mathbf{PSK_Q}$). The $\mathbf{PSK_Q}$ signal is anyway phase shifted by 90° before the modulation.

3.2.2 Design methodology and implementation results

Figure 3.2 shows the block diagram depicting the QPSK modulation scheme. Figure 3.3 shows the design and verification framework developed in MATLAB. Figure 3.3 additionally presents the HDL code generated by applying MATLAB HDL coder.

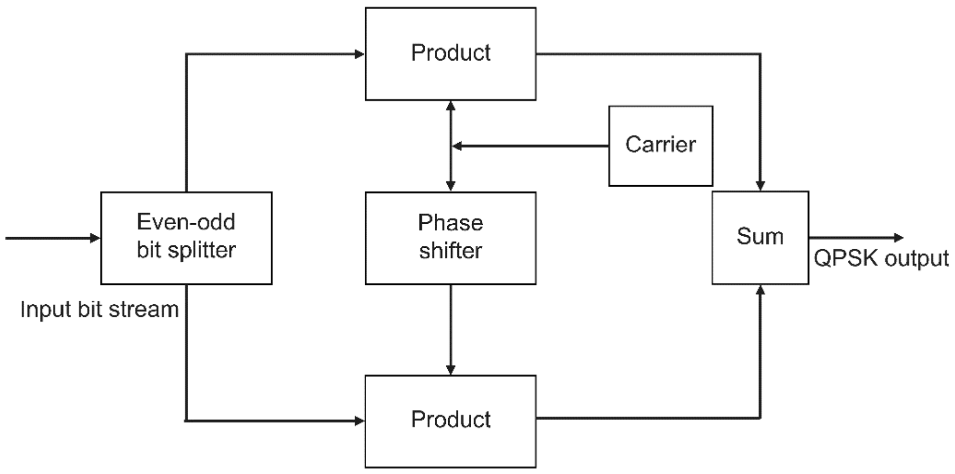
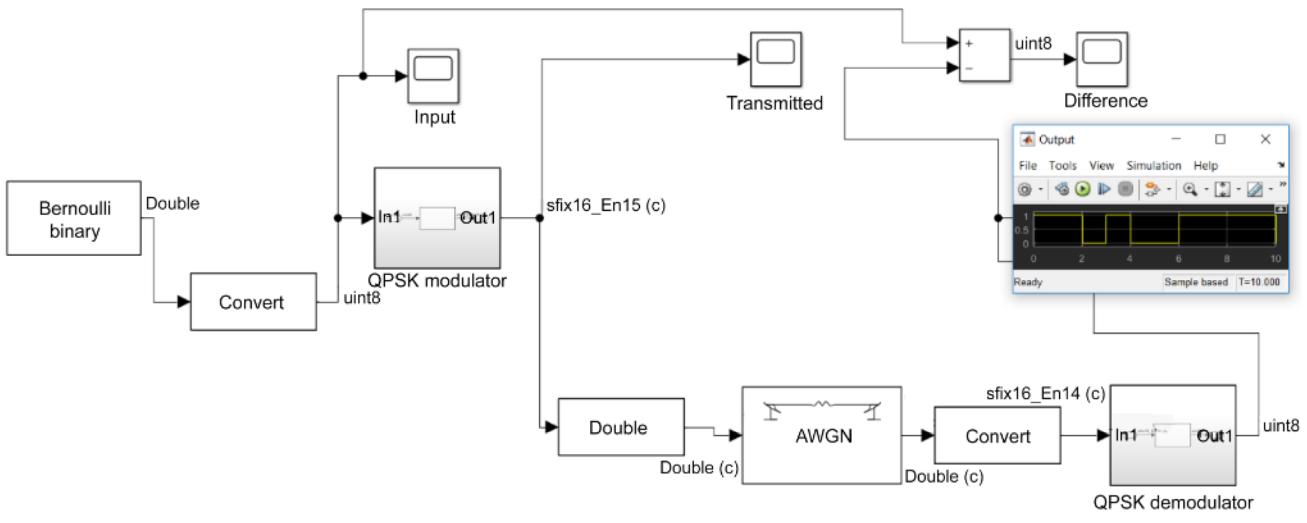


Figure 3.2. QPSK modulator block diagram.



(A)

```
process(clk, reset)
begin
if(reset = '1') then
dout_real <= x"0000";
dout_imag <= x"0000";
valid_out1 <= '0';
sig_valid_out <= '0';
sig_valid_out1 <= '0';
elsif(clk = '1' and clk'event) then
valid_out1 <= '1';
sig_valid_out <= valid_out1;
sig_valid_out1 <= sig_valid_out;
if valid_in = '1' then
case data_in is
when "00" => dout_imag <= x"2d41";
dout_real <= x"2d41";
valid_out1 <= '1'; -- Based on the input data assign
when "01" => dout_imag <= x"D2BF"; -- values to the output ports
dout_real <= x"2d41";
valid_out1 <= '1';
when "10" => dout_imag <= x"2d41";
dout_real <= x"D2BF";
valid_out1 <= '1';
when "11" => dout_imag <= x"D2BF";
dout_real <= x"D2BF";
```

(B)

Figure 3.3. (A) QPSK modulator design and verification using MATLAB and Simulink. (B) HDL code generated using HDL coder.

Table 3.3 shows the synthesis reports for Kintex 7(XCK770T-FBG676) FPGA for HDL code generated from optimized model using distributed pipelining directive [3.5]. Table 3.4 shows the synthesis reports for the same application using hand-coded RTL. Further, functional verification of the generated RTL design was performed using xSim simulation. A comparison was also done against the reference implementation in MATLAB using FPGA-in-the-loop feature of MATLAB HDL Verifier.

Table 3.3. Kintex 7 FPGA implementation results for QPSK modulator (HLS optimized)

Resource	Utilization	Available	Utilization (%)
LUTs	212	41000	0.517
IOs	34	300	11.333
Flip-flops	765	82000	0.933
BUFG	1	32	3.125

Critical Path: 2.97 ns, Operation Freq. = 330 MHz, Total On-chip Power = 0.263 W

Table 3.4. Kintex 7 FPGA implementation results for QPSK modulator (hand-coded RTL)

Resource	Utilization	Available	Utilization (%)
LUTs	328	41000	0.800
IOs	34	300	11.333
Flip-flops	740	82000	0.902
BUFG	1	32	3.125

Critical Path: 3.62 ns, Operation Freq. = 276 MHz, Total On-chip Power = 0.221 W

As can be seen from Tables 3.3 and 3.4, the results obtained with HLS optimization are better than those obtained for hand-coded RTL implementation in terms of resource utilization as well as the operating frequency. Another point to note is that flip-flop utilization is slightly increased in HLS implementation owing to the usage of distributed pipelining. This is because it introduces flip-flops on combinational paths. A marginal increase in power dissipation is also attributed to the increase in the number of flip-flops. It may be noted that one has the option to optimize the hand-coded implementation as well but the goal here is to demonstrate superior or comparable implementation and simulation results with shorter design cycle time using HLS.

So, to conclude, for this application of QPSK, the HLS directive of “distributed pipelining” helps in the overall reduction of resource utilization (Flip-flops and LUTs together) as well as improvement in frequency of operation at the cost of a slight increase in power dissipation of the design.

3.3 MIPS processor core (HDL coder)

Herein, design, implementation and optimization for a MIPS processor core is discussed using MATLAB HDL coder.

3.3.1 Introduction to MIPS cores

MIPS processors are based on the reduced instruction set computer architecture. This was developed by MIPS technologies and Imagination Technologies and has evolved from 32-bit to 64-bit version over the last few years. These processors have been in use for years and remain in wide use today as well in varied applications such as automation, information processing, and communication. MIPS processors are often used in applications involving consumer audio devices, such as audio players, set-top boxes, DVD recorders and players, and digital displays, which are typically implemented with a multifunction system on-chip.

3.3.2 Previous works for MIPS processor implementation

Numerous research efforts have focused on MIPS architecture in the past. In 2019, Indira et al. implemented a 32-bit MIPS processor and targeted the same on a Xilinx Virtex 7 FPGA [3.6]. They also discussed possible pipeline hazards and the associated remedies. In 2017, Rashidah et al. proposed a simulator for the RISC-16 instruction set that was based on visual basic programming and five pipeline stages [3.7]. In 2016, Husainali et al. proposed a three-stage, 32-bit pipelined processor that they designed in Verilog and implemented on a Xilinx Virtex 7 FPGA using Xilinx ISE software [3.8]. In 2018, Mangalwedhe et al. proposed a low-power RISC processor that they designed in Verilog [3.9]. They used clock gating to decrease the dynamic power consumption. The design was then implemented on a Spartan 6 FPGA. In 2014, Rakesh et al. proposed a novel architecture for a 17-bit address RISC processor [3.10]. They implemented their Harvard architecture-based design on a Xilinx FPGA. In the present work, we use HLS to design a MIPS core and implement it on a Xilinx Virtex 7 FPGA target. Furthermore, we compare this implementation with previous implementations. In the proposed design, we use multiple HLS directives to decrease the area and boost the speed of implementation. The proposed FPGA’s results are clearly better than those of previous implementations proposed in literature.

3.3.3 Proposed design and optimization using directives

The processor model for the MIPS core was created using Simulink with MATLAB function blocks. Figures 3.4 shows the top level implementation. MIPS block includes the data path and controller as shown in Figures 3.5 and 3.6 respectively. An instruction parser operates within the data path and consisted of the opcode, source register, destination register, immediate operand, and jump address. The parser is directly linked to a sign-extend block and a jump calculator block. A 32-bit register file is also available which consists of one writing port and two reading ports. The ALU result consists of three inputs: ALU control, Scr A, and Scr B. This block performs four major operations on the input: addition, subtraction, AND and OR between the Scr operands A and B. The register file provides Scr A, and the Scr B output data are obtained from the sign-extended immediate value. The three-bit ALU control specifies the operation to perform on operands while the ALU generates a 32-bit result and a zero flag: this is to indicate if ALU result is equal to zero. The ALU Scr multiplexer is used to handle the R-type instructions, which write the ALU result in the register file. Therefore, we add this multiplexer to select between Read Data and ALU result and call the output as “Result.” This multiplexer was controlled by the signal “Mem to Reg,” which is zero for R-type instructions to choose result from the ALU Result, and unity for load word instruction (lw) to choose Read Data. For verification of the design implementation, a pre-compiled machine code was used as boot software without having the need to use a C compiler.

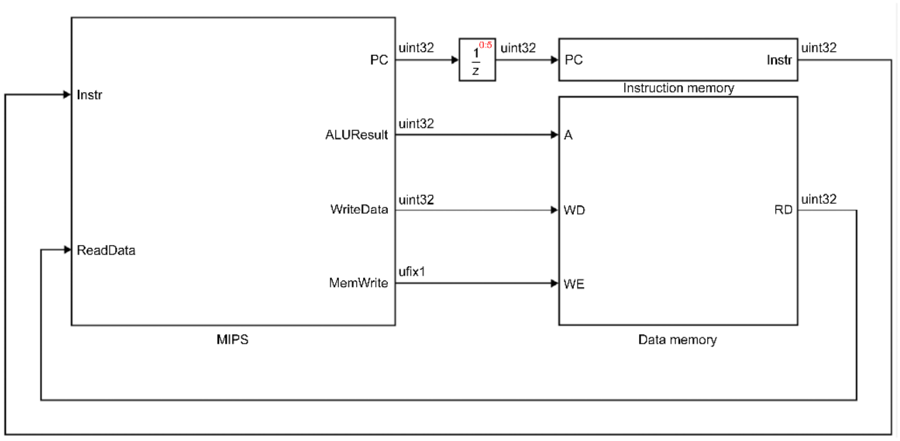


Figure 3.4. Top-Level Implementation of MIPS processor system with memory.

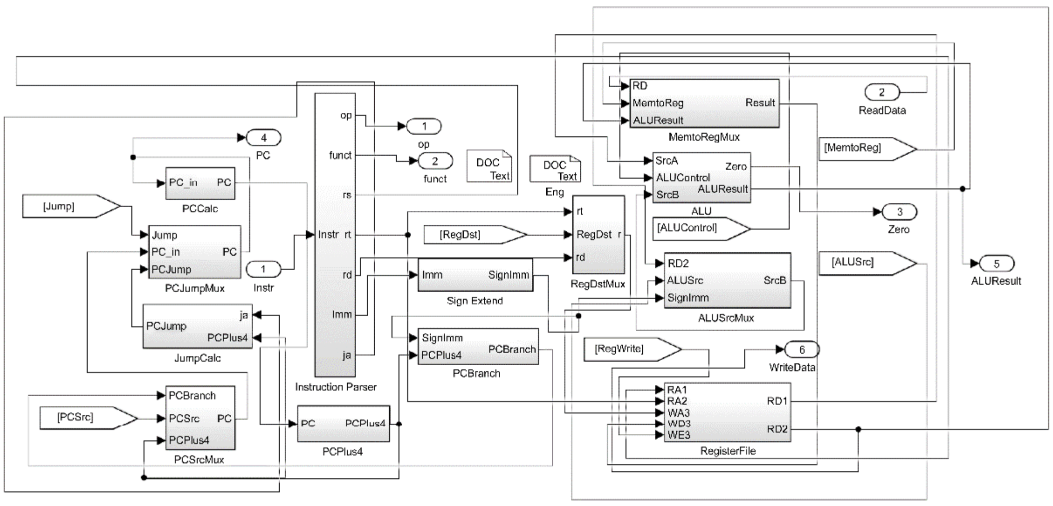


Figure 3.5. MIPS data path model.

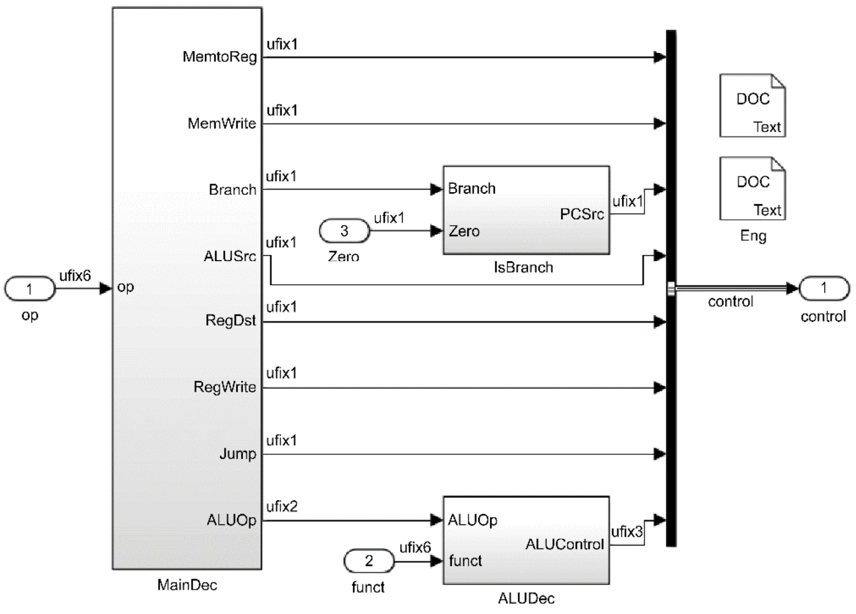


Figure 3.6. MIPS processor controller.

After the base implementation was created, MATLAB HDL coder was used to convert the MIPS core model to synthesizable Verilog code and subsequently run it through FPGA synthesis using Xilinx Vivado. As a second step, the following HLS directives were applied to optimize the results of the MATLAB HDL coder:

i. Hardware Pipeline. The HLS directive enables the concurrent execution of operations for memory read and write. This is done by decreasing the initiation interval for a loop or a function (memory read and write in this case) implemented in hardware. While using this directive, a tradeoff in area and speed must be considered. Since the proposed implementation uses loops for reads and writes to memory, a pipeline directive with an initiation interval of two was used for all hardware loops [3.11].

ii. Loop unroll allows the loop iterations to run in parallel by generating various copies of the same loop body in the generated RTL [3.11]. The directive supports either partial or full unrolling of loops with minor trade-off on resource consumption. Unroll factor of two is used in the implementation. Further, loop unroll applied on memory read and write loops helped in reducing the pessimism for input and output paths by pipelining the IO pads. Further, it also helps to replicate the single loop into two separate read and write paths from memory, leading to better performance with slight area penalty.

3.3.4 Simulation, FPGA implementation results and comparison

After generating the RTL code, we performed an RTL simulation for the design using a nonsynthesizable Verilog test bench using xSim software. Additionally, FPGA-in-the- loop simulation was also performed. The Verilog memory model in the testbench was preloaded with pre-compiled memory contents generated by the compiler (boot firmware). The simulation results were identical to the simulation results obtained using a high-level simulation in Simulink. Table 3.5 shows the FPGA implementation results for Virtex 7(XC7V585T-3) FPGA and Table 3.6 shows a comparison with other works in the literature.

Table 3.5. Virtex 7 FPGA resource utilization for proposed MIPS implementation

Resource	Utilization	Available	Utilization (%)
Slice Reg	43	91050	0.05
Slice LUTs	178	582720	0.03
Flip-flops	41	728400	0.005
Bonded IOBs	47	850	5.53
BUFG	1	32	3.125

Critical Path: 2.47 ns, Operation Freq. = 404.1 MHz, Total On-chip Power = 0.021 W

Table 3.6. Comparison of FPGA implementation results of MIPS for Virtex 7

Resource	Proposed design	Indira et al. [3.6]	Rakesh et al. [3.10]
Slice registers	43	81	56
Slice LUTs	178	321	203
Flip-flops	41	81	43
Bonded IOBs	47	71	51
BUFG	1	2	1
Power (W)	0.021	0.023	1.318
Maximum Frequency (MHz)	404.100	420.028	100.000

As can be seen from Tables 3.5 and 3.6, even though the proposed implementation operates at almost the same operating frequency as that of Indira et al. [3.6], the resource usage is 40%-50% less. It also important to note that even with almost same flip-flop count, the implementation is about 4x faster than the one proposed by Rakesh et al. [3.10]. This is because of reduced timing calculation through IOs and replication of separate input and output paths owing to unrolling of memory write and read loops. Further, both Indira et al. and Rakesh et al. have used hand coded RTL along with pipelining optimize their respective implementations. To conclude, the results of FPGA synthesis clearly indicate that the proposed implementation is superior to previous implementations, despite having the same design specifications.

3.4 Encryption and cryptography algorithms: AES (Vivado HLS)

Herein, design, implementation and optimization of a common cryptographic algorithm, AES using Vivado HLS is discussed.

3.4.1 Introduction to AES algorithm

Nowadays, the consumer industry extensively utilizes communication technology to connect devices without wires, i.e., wireless communication [3.12]. Further mobile commerce applications and wireless information services are needed because number of mobile users have grown significantly worldwide [3.13]. In the last two decades, cellular networks' transformation from 2G to 3G and 4G to 5G was considerable [3.14]. Compared with preceding technologies, 5G has much better reliability and higher bandwidth support [3.15]. Still, higher connectivity has various shortcomings, like more possibilities of data theft. For example, consider the web's online transactions where multiple devices are connected. These transactions are more susceptible to financial scams than cash payments at merchants: in the second case, just two parties are included.

Cryptography is one method for ensuring that messages are encrypted and the messages' reception is performed only by the intended receiver. One of such algorithms for the key encoding process is the AES [3.16]. The AES algorithm is regularly used for encryption in several applications, which include IEEE standards like 802.11i, 802.15.4 and ZigBee [3.17, 3.18, 3.19].

In the context of cryptographic algorithms, a chip block is a method to protect the symbolic importance of the message to be transmitted. A block cipher is a computable and deterministic function that produces n-bit ciphertext employing k-bit keys and n-bit plaintext blocks. Because it is deterministic, the same output ciphertext would be produced every time the input text and keys being used are the same. AES receives a 128-bit input for each block and a key size: 128, 192, 256 bits. Consequently, it produces a ciphertext after a finite number of encryption rounds (Nr), which is the function's key size. A key size of 128 bits with 10 rounds of block size is the most common. The ciphertext is the output scrambled version of the input plaintext. AES encryption rounds (iterations) are conducted in a finite field, specifically a Galois Field (GF) of 28 [3.20]. In a GF, mathematical operations, such as addition and subtraction, can be executed smoothly as data is presented in vectors. AES is an iterative algorithm acting on a square matrix of symmetric size, denominated as a state. The state arranges the message's finite number of bytes into columns. Each column consists of a fixed number of bytes. Once a message is set up, five functions are called on the message, and they act on the state: byte substitution, shift rows, mix columns, add round key, and key scheduling described later in the section.

The AES algorithm allows multiple modes of operation. Such modes differ in the way the input text is arranged into blocks and how they are transformed. There are four major modes: counter (CTR), cipher block chaining (CBC), electronic codebook (ECB), and cipher feedback. ECB is the simplest of the AES modes. In the ECB mode, the encryption and decryption occur independently of the other blocks. The implementation is more straightforward in ECB mode, but one disadvantage is the patterns' replication which is avoided in CTR mode using a counter value and initialization vector (IV). In CBC mode, there is no correlation between input and output, making it slightly more complicated, but at the same time more secure. A pseudo-random IV application accomplishes this as an input message to plaintext and the input variable's derivation as output from the previous block, hence the name chaining. Below is the detailed description of functions in the encryption process.

3.4.1.1 Byte substitution function

In the AES algorithm, the byte substitution function involves substituting the inputs with new bytes using a pre-defined matrix (called substitution box or S-box) [3.21]. Figure 3.7 (B) shows the AES S-box. For byte substitution to happen, first hexadecimal character is used as row, second character is treated as column and intersection point becomes new byte. As an example, if input is 0x11, it becomes 0x82 (Entry in Row1, Column 1), Input of 0x12 becomes 0xC9 (Row 1, Column 2).

Another way to explain the same behavior is using an affine transform in GF (2^8). It is described in Figure 3.7 (A) and 3.7 (C). Taking the same example of input 0x11 (00010001). It corresponds to polynomial $x^4 + 1$. Multiplicative inverse of this is g(x) $x^7 + x^5 + x^4 + x^2$ such that f(x) × g(x) is 1. Hence g(x) = 0xB4 = ‘b10110100. When we substitute g(x) as 00101101(LSB to MSB) in Figure 3.7 (A), we get output 01000001(LSB to MSB) which is 0x82. This is the same result we get from S-box in Figure 3.7 (B). Figure 3.7 (C) shows the pseudo-code for the same algorithm.

$$\begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} x0 \\ x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

(A)

$$\begin{bmatrix} 63 & 7C & 77 & .. & .. & D7 & AB & 76 \\ CA & 82 & C9 & .. & .. & A4 & 72 & C0 \\ B7 & FD & 93 & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & .. \\ 70 & 3E & B5 & .. & .. & C1 & 1D & 9E \\ E1 & F8 & 98 & .. & .. & 55 & 28 & DF \\ 8C & A1 & 89 & .. & .. & 54 & BB & 16 \end{bmatrix}$$

(B)

```
unsigned char sbox (unsigned char in) {
unsigned char count, s, x;
s = x = multiplicative  inverse (in);
for (count = 0; count<4; count ++)
{
        s = (s<<1) | (s>>7);
        x ^= s;
}
X ^= 0x63;
return x;
}
```

(C)

Figure 3.7. (A) Affine transform for byte-substitution.
(B) S-box representing each element
(C) Psuedo-code for byte substitution

3.4.1.2 Shift rows function

A linear operation shifts each state matrix’ row by a finite number. The I row is unchanged; the II row is circularly left shift by one byte, the III row is changed circularly with a two-byte left shift, and the IV row is circularly shifted to the left with 3 bytes. This method provides diffusion. The shift rows function, operating on the cipher, is displayed in Figure 3.8.

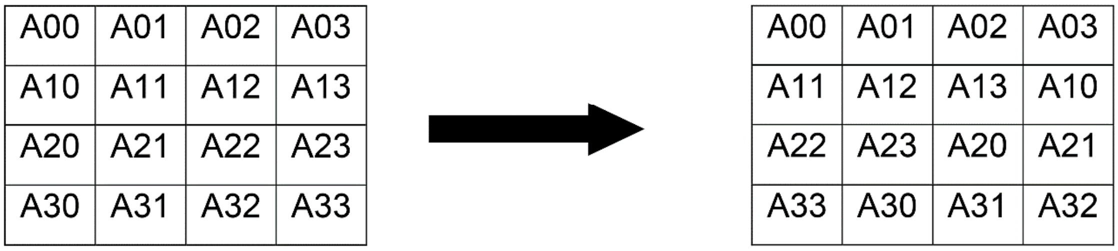


Figure 3.8. Shift rows function on block cipher.

3.4.1.3 Mix column function

The Mix columns function provides, in a similar fashion to the shift rows function in AES, diffusion to the data by mixing the inputs. This operation is executed by splitting the matrix through columns instead of rows. Matrix multiplication is computed according to the GF 28. Figure 3.9 shows how there are an independent multiplication of each column.

$$\begin{array}{|c|c|c|c|}\hline A00 & A01 & A02 & A03 \\ \hline A10 & A11 & A12 & A13 \\ \hline A20 & A21 & A22 & A23 \\ \hline A30 & A31 & A32 & A33 \\ \hline \end{array} \mathbf{X} \begin{array}{|c|c|c|c|}\hline 2 & 3 & 1 & 1 \\ \hline 1 & 2 & 3 & 1 \\ \hline 1 & 1 & 2 & 3 \\ \hline 3 & 1 & 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline B00 & B01 & B02 & B03 \\ \hline B10 & B11 & B12 & B13 \\ \hline B20 & B21 & B22 & B23 \\ \hline B30 & B31 & B32 & B33 \\ \hline \end{array}$$

Figure 3.9. Mix column function on block cipher.

3.4.1.4 Add round key

During this stage, the state matrix is bit-by-bit XOR (or addition in GF) by the 16-byte round key (128 bits). This feature is invoked 11 times (10 rounds and one additional before the first round). Consequently, 11 × 16 = 176 bytes of the key are required. The 16-byte key is proceeded to expand to 176 bytes in this stage.

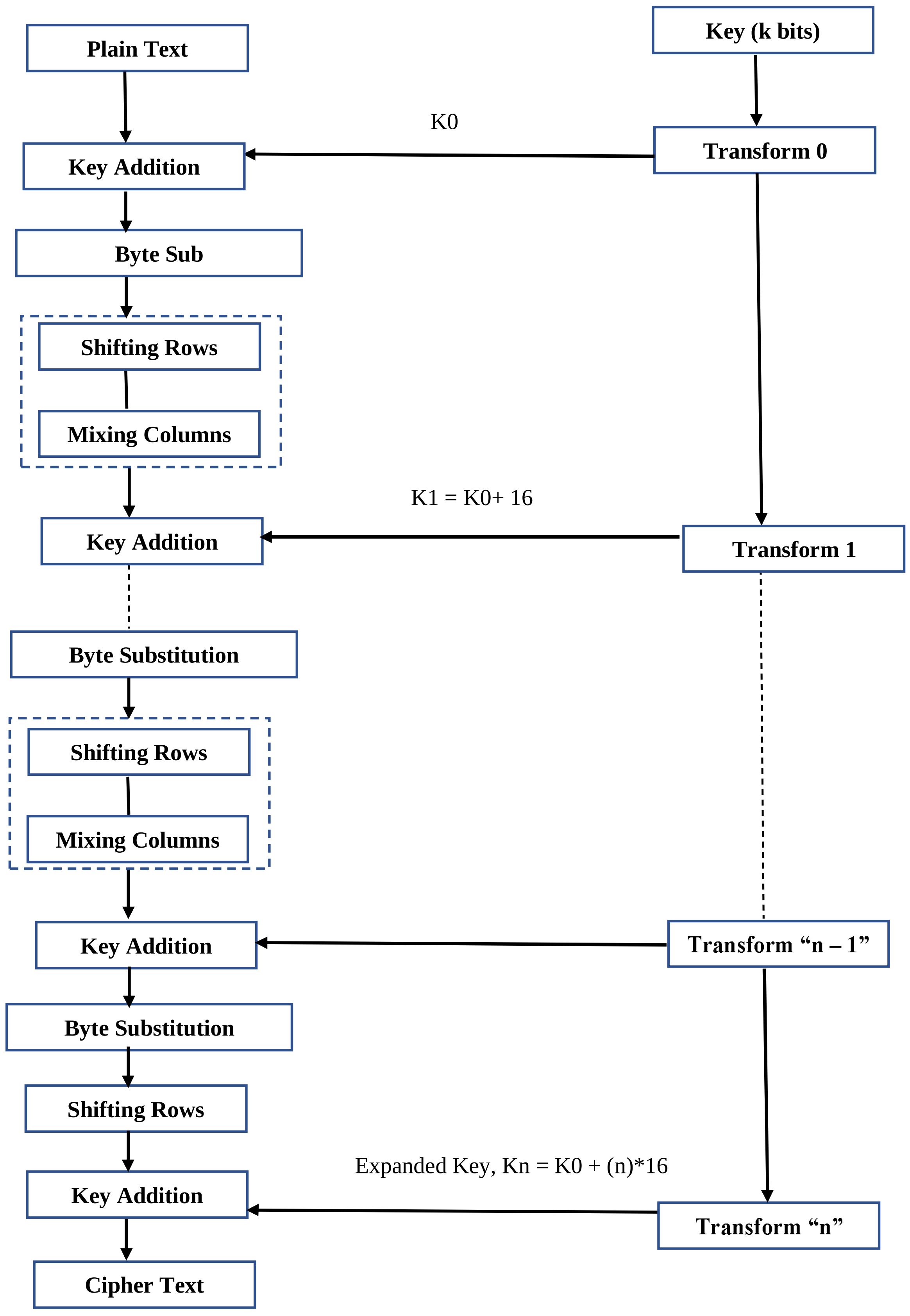


Figure 3.10. Functional stages of AES.

3.4.1.5 Key scheduling

This round determines the keys utilized in the algorithm from the starting input key (16 bytes). A different key is generated from the previous fundamental exploitation for each key addition: an XOR of some bits with the last key value. Thus, while working on words N to M, the value used in the XOR is the previous word of the previous round key, i.e., N-1 to M-1. Figure 3.10 presents the AES algorithm’s included functions and steps.

3.4.2 Previous works for AES algorithm implementation

In the recent past, various optimal FPGA hardware implementations of the AES algorithms have been introduced. In 2015, Soltani and Sharifian proposed an ultra-high throughput implementation of a fully pipelined AES algorithm. Their implementation is based on the counter (CTR) mode of the operation and targeted for the Virtex-6 Xilinx FPGA [3.22]. In 2018, Zhang et al. introduced another AES architecture offering increased throughput because of an inner and outer pipelined architecture [3.23]. Their implementation showed a throughput of above 60 Gbps on Xilinx Virtex-6 FPGA. In 2018, Smekal et al. presented a comparison of two distinct encryption algorithms. They targeted design implementations on the FPGAs Virtex-7 and Ultrascale+ and obtained a maximum throughput of about 50 Gbps on both targets [3.24]. In 2019, a custom AES encryption algorithm was implemented on the Spartan 3 E FPGA kit. This was done by Noorbasha et al. Their implementation produced better data security at the expense of slightly decreased throughput [3.25]. In 2019, Chen et al. presented an encryption algorithm for pipelining in big data applications that reached above 30 Gbps [3.26].

3.4.3 Proposed implementation and optimization using directives

For our design implementation, we chose the CTR mode of operation. A 16-byte initialization vector was created with an unchanged nonce with an incrementing counter value with each block encryption completion. Because of CTR mode’s parallel nature, it is commonly used in wireless network application designs with streaming data and ciphers. Figure 3.11 portrays the AES algorithm’s CTR mode of operation.

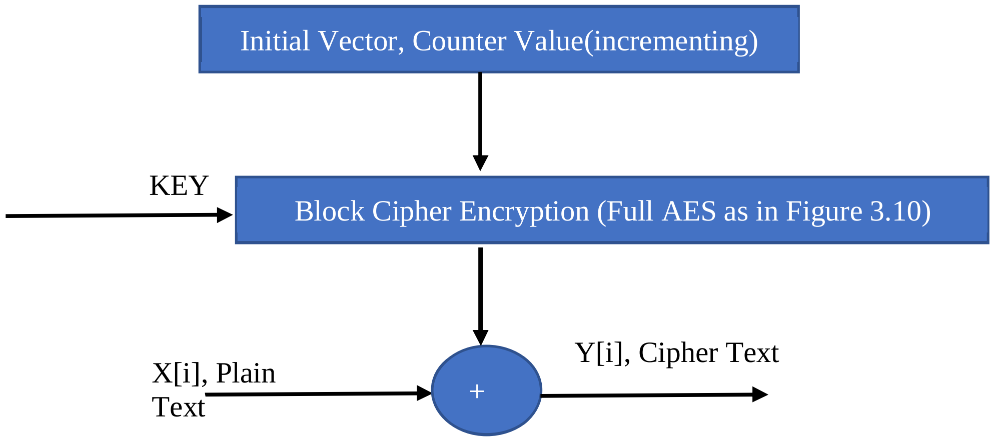

Initial Vector, Counter Value(incrementing)
KEY
Block Cipher Encryption (Full AES as in Figure 3.10)
X[i], Plain Text
+
Y[i], Cipher Text

Figure 3.11. CTR mode of operation for AES Algorithm.

The C++ model for the AES algorithm in CTR mode was created using 10 rounds and a block length of 128 bits. Simulation of the algorithm was performed with a C++ testbench using Vivado HLS. The input signal was chosen to be an input text string or a stream of characters, i.e., plaintext. The implementation ran on the Vivado HLS high level simulator, and the output was a ciphertext corresponding to plaintext at any given instant in time. Figure 3.12 shows the algorithm's design using floating type data types for design nodes.

```
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)) {
                MixColumns(state);
        }
        AddRoundKey(state, expandedKey + (16 * (i + 1)));
}


AddRoundKey(state,newState);

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

Figure 3.12. Algorithm model for the advanced encryption standard.

Once the baseline model was created, Vivado HLS was used to generate synthesizable Verilog RTL. It was then implemented on the target FPGA. Moreover, to optimize the operation results (throughput), HLS directives offered by tool were used [3.11]. The pipeline, unroll, and inline-off directives that we used are explained below.

i. The **pipeline** permits the operations' concurrent execution by decreasing the initiation interval for a loop or a function. Therefore, while using this directive, a tradeoff exists between timing and area. This AES implementation is optimized by applying a pipeline with an initiation interval of two for all design loops.

ii. **Loop unroll** enables the loop iterations to run in parallel by generating multiple single-loop body copies in RTL design. This pragma assists in improving the throughput by making the loops either fully or partially unrolled. For our AES application, the partially unroll directive was used to enhance the performance with minimal resource usage. The unroll factor applied on the code is two.

iii. After a function is inlined, it can no longer be interpreted as a separate entity in the hierarchy. Inline functions are merged into the calling function. These functions, consequently, cannot be shared, increasing the area utilized by the target FPGA. The **inline-off** directive was used in our application to restrict the functions from inline on their own. This helps alleviate the effect of an area increase produced by the loop unrolling and pipelining. Table 3.7 summarizes the different HLS directives applied on distinct functions or loops in the design.

Table 3.7. Functions in AES implementation and applied HLS directives

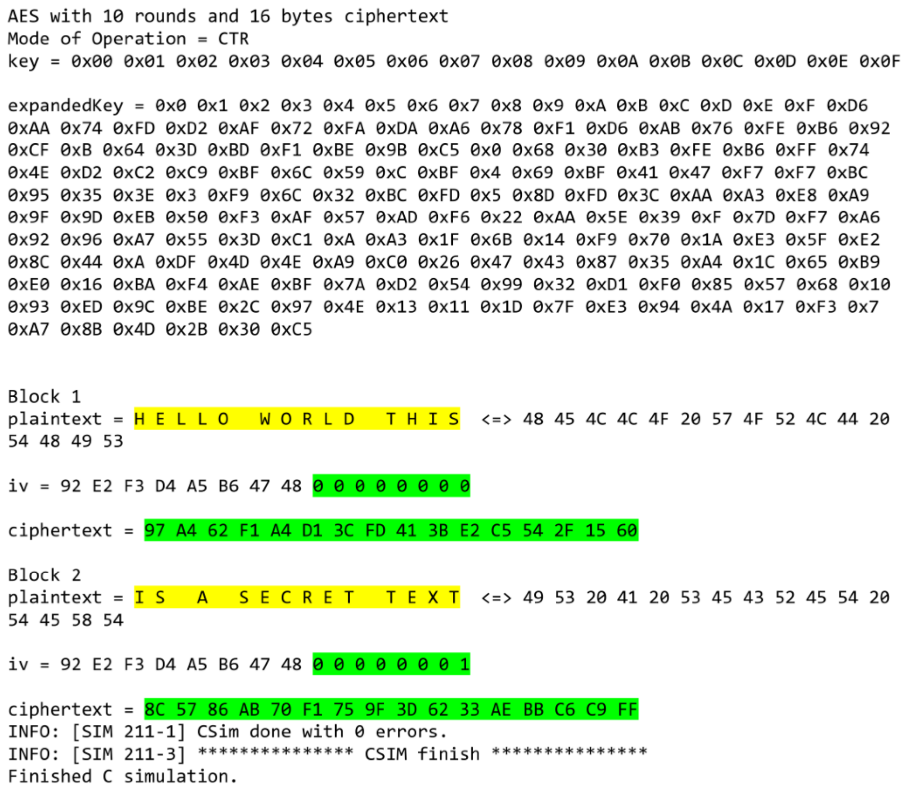
Function	Pipeline	Unroll	Inline (Off)
Sub Bytes	YES	YES	YES
Shift Rows	YES	YES	YES
Mix Column	YES	YES	YES
Add Round Key	YES	YES	YES
AES Encrypt (Top)	YES	NO	NO
L Rounds (Inner Loop)	YES	NO	YES

This proposed design and optimization method is suitable for other HLS tools and other applications apart from AES having similar bottlenecks. Moreover, as the produced RTL code is optimal, the method is also applicable to all FPGA targets. Depending on the types of directives available in the tool and the bottlenecks identified for the application, one can optimize the synthesis results. In the next section, the effect of these HLS directives on optimization for FPGA synthesis is presented for AES.

3.4.4 Simulation, FPGA implementation results, and comparison

Simulation results

Since AES is a published standard, this study confirmed that the functional verification results achieved by using a C++ testbench matches with the standard. We confirmed the same by comparing the output from MATLAB against the output (ciphertext) from the design's HLS model for the same input text. Figure 3.13 displays the encryption results for the plaintext "HELLO WORLD THIS IS A SECRET TEXT." The algorithm requires the input two times (16 bytes per text), with a completed counter to two (after the increment) as there are 32 characters in the input file.



```
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 3.13. HLS model simulation results of AES.

As depicted in Figure 3.13, and considering the implementation with the applied directives, the base implementation's simulation results were identical. This demonstrates functional equivalence between HLS and design implementation on the target FPGA. Moreover, the RTL functional simulation results obtained from xSim (an HDL simulator integrated with Xilinx Vivado) were also equal to the same plaintext's theoretical AES output. This was confirmed by the HLS simulation using a C++ testbench. Additionally, FPGA-in-the-loop verification confirmed the same results.

FPGA synthesis

From a guided Vivado HLS framework, the Verilog RTL code was created. The same code was implemented on the Kintex 7 FPGA from Xilinx (XC7K70T-FBG676 -3). Vivado HLS software version 2019.1 was used. The generated RTL source code was also implemented on Virtex 6 device using iSE 14.7 software from Xilinx. This was done to facilitate a direct comparison with some of the available literature results for same target boards. Even though our results are presented for Virtex 6 and Kintex 7 FPGA, the directives help RTL level optimization. This indicates that the optimizations are technology-independent and consequently appropriate for other different FPGA targets.

Tables 3.8 and 3.9 show the corresponding FPGA implementation reports from both the baseline design as well as one created using HLS directives, respectively.

Table 3.8. Base implementation results of AES on Kintex 7

Resource	Utilization	Available	Utilization (%)
LUTs	572	41000	1.395
LUT-RAMs	8	13400	0.060
Flip-flops	446	82000	0.543
BRAM	4	135	2.962
IOs	264	300	88.000
BUFG	1	32	3.125

Throughput = 28 Gbps, Operation Freq. = 218.8 MHz, Total On-chip Power = 0.114 W

Energy dissipation, $E = P \times T$, where P is power and T is time required for one round of AES for 128 bit input. Since throughput is 28 Gbps, we see $T = (1/28) \times 128$ ns.

Hence, energy dissipation, $E = (1/28) \times 128 \times 0.114 = 0.52$ nJ (Nano Joules)

Table 3.9. HLS directives optimized implementation of AES on Kintex 7

Resource	Utilization	Available	Utilization (%)
LUTs	749	41000	1.827
LUT-RAMs	9	13400	0.070
Flip-flops	866	82000	1.056
BRAM	4	135	2.962
IO	264	300	88.000
BUFG	1	32	3.125

Throughput = 54.2 Gbps, Operation Freq. = 425 MHz, Total On-chip Power = 0.124 W

Energy dissipation, E = Px T, where P is power and T is time required for one round of AES for 128 bit input. Since throughput is 54.2 Gbps, we see T = (1/54.2) × 128 ns.

Hence, energy dissipation, E = (1/54.2) × 128 × 0.114 = 0.293 nJ (Nano Joules)

As is evident from Tables 3.8 and 3.9, there is an improvement in design's performance with usage of pipeline and loop unroll directives. This is due to the fact that critical paths in the design are reduced due to retiming and logic replication. However, due to additional retiming flip-flops and loop unrolling (which leads to logic replication), an increased resource usage is also observed. Nevertheless, the throughput increase of almost 2x is attributed to faster synthesis frequency and the pipelining of several loops and functions, as described in Section 3.4.3.

Results comparison with literature

Our implementation was compared with other proposed implementations available in the literature [3.22, 3.23, 3.26]. Table 3.10 shows comparison of results for Virtex-6 (V6) and Kintex-7 (K7) FPGA devices. Soltani et.al [3.22] used memory and non-memory based approaches along with loop unrolling and pipelining in their implementation. Zhang et.al [3.23] in their implementation combined multiple steps of round units, used a dual port ROM structure and used 3 stage pipelining to optimize the design. Chen et.al [3.26] used deep pipelining and full expansion technology in their implementation for optimization.

Table 3.10. Comparison of AES synthesis results with the literature

FPGA metrics	Proposed design	Soltani et.al [3.22]	Zhang et.al [3.23]	Chen et.al [3.26]
Max Freq (V6) (MHz)	542.102	508.102	471.000	Not available
Throughput (V6) (Gbps)	276.50	260.14	60.30	Not available
Max Freq (K7) (MHz)	425.0	Not available	Not available	244.4 MHz
Throughput (K7) (Gbps)	54.2	Not available	Not available	31.2

As is clear from Table 3.10 and the simulation results discussed above, the proposed implementation has better throughput than other implementations available in the literature with identical functionality. The throughput for the proposed design is roughly 8% higher than that of Soltani et al., 17% higher than the design proposed by Zhang et al., and 70% higher than Chen et al. [3.22, 3.23, 3.26]. Moreover, the improvement obtained is technology-independent, i.e., it applies to all FPGA targets. We have tested and obtained the results for two widely used Xilinx FPGAs - Virtex 6 and Kintex 7. From the previous analysis, it is concluded that our design implementation has a higher throughput than the ones presented by other researchers, with a small tradeoff in resource usage. Further, we could not provide comparison of energy utilization with literature as same was not reported by other researchers. We also validated that usage of these performance directives is benign for design functionality.

3.5 YOLO v2 deep learning algorithm (Vivado HLS)

Herein, design, implementation and optimization for a YOLO v2 algorithm which is based on convolutional neural network using Vivado HLS is discussed.

3.5.1 Introduction to YOLO v2 algorithm

Artificial intelligence and machine learning techniques have gained increasing popularity in recent years and almost all fields of engineering and technology, including computer vision. The models based on convolutional neural networks (CNNs) intended for object detection have been constantly evolving. The object detection task is challenging in terms of accuracy and speed. Accordingly, several perspective algorithms for object detection have been developed based on deep learning, including single-shot multibox detection (SSD), region-based CNN (R-CNN) and YOLO [3.27, 3.28, 3.29]. The YOLO CNN model that was slightly larger than previous implementations but almost three times faster was introduced by Redmon and his research team in 2016 [3.30]. A speed of 45 frames per second was achieved with certain accuracy, enabling video detection [3.31]. An improved version of YOLO, namely YOLO v2, could maintain high recognition accuracy at a high speed and provided a large improvement in real-time image processing [3.32]. Comparing the YOLO algorithm with the R-CNN algorithm, which requires multiple CNN operations, it can be noted that the accuracy of the latter is better, but the former is faster [3.33]. Therefore, YOLO provides one of the best tradeoffs between accuracy and speed in object detection. YOLO is based on a single neural network used to predict the bounding boxes of objects and the class confidence in a single evaluation. A GPU is commonly employed to implement deep learning techniques. However, it becomes ineffective for optimization tasks, such as selecting the data width and managing data access in peripheral memory. Therefore, extensive research is conducted to design deep learning accelerators, which can be deployed on FPGAs to tackle this challenge. Additionally, FPGAs provide parallel architectures, thus enabling execution on high data rates. FPGAs have a flexible design and short development cycles, and therefore, they have been extensively applied to high-efficiency deep learning tasks.

3.5.2 Previous works for YOLO v2 implementation

Several implementations of YOLO rely on floating-point representation; however, they are associated with large computational costs [3.33, 3.34, 3.35]. FPGAs support only fixed-point implementations, and therefore, they can be applied to address this problem.

The research introduced by Nguyen et al. reported that a floating-point representation was unnecessarily redundant [3.36]. Several studies clearly demonstrated that CNNs could be quantized to a lower number of bits and trained without the significant loss of accuracy [3.37, 3.38, 3.39]. Quantization facilitated designing a low-power and fast CNN accelerator based on an FPGA. Such FPGA could store the complete quantized CNN model in its on-chip block RAM comprising tens to hundreds of MB. Therefore, FPGA is merged with low-bit CNN quantization to create a low-power accelerator for deep neural networks, providing a throughput of the order of tera operations per second. Various FPGA implementations were realized using CNNs based on Vivado HLS [3.33, 3.35, 3.40, 3.41, 3.42]. Zhang et al. deployed a single processing element based on a theoretical roofline model applied to design an accelerator for the implementation of each CNN layer [3.33]. However, to realize a small network of five levels, the accelerator consumed a significant area on the target FPGA chip. All of this while operating at a low throughput of 61 giga operations per second.

Alwani et al. [3.35] and Xiao et al. [3.42] proposed a fused-convolutional layer in CNNs for downgrading the number of off-chip accesses by optimizing intermediate data between adjacent layers in a group. However, a significantly greater number of block RAMs were required in these designs.

A CNN accelerator developed by Sun et al. [3.40] was applied to optimize the data path, applying a loop unrolling directive and providing improved performance for each layer. The authors employed Vivado HLS in the separate design of each layer of the proposed CNN accelerator. However, a double buffer was required to store entire intermediate feature maps produced by each layer. Therefore, this design did not scale appropriately when the CNN became deeper due to large buffers needed. Li et al. proposed a design that was associated with a similar limitation, in spite of proving good performance in the Alex Net network [3.43]. Shen et al. also employed Vivado HLS for their implementation, and their design achieved a similar optimization as the approach proposed by Zhang et al. [3.41, 3.33]. However, in their implementation, the available resources were partitioned in a way that allowed scaling down the size of multiple convolutional layer processors (CLP) into a smaller size, rather than using a single large CLP.
Ma et al. presented a proposal in which a RTL compiler was used to generate, for a given network, an RTL code for each layer [3.44]. This aimed to solve the problem of the excessively high memory bandwidth requirement. Every convolutional layer read the inputs and wrote outputs to a dynamic RAM (DRAM) in this configuration. Each layer was launched consecutively, meaning that the following layer began only when the current layer finished its execution. This processing scheme and repeated access to the external DRAM drastically reduced the processing speed. Wnograd et al. proposed a custom minimal filtering algorithm that differed from conventional convolution [3.45]. This algorithm was further utilized by Aydonat et al. and Lu et al. to improve the speed of convolutional computations [3.46, 3.47]. However, these designs still required a large number of DSPs and LUTs although applying the Winograd algorithm allowed reducing the number of required multipliers. They also reported that the performance of their design decreased as the network deepened due to the need to transfer data back and forth between the accelerator and external memory. This limitation was mitigated by Umuroglu et al. and Liang et al. by reducing the number of expensive external memory accesses [3.48, 3.49]. They realized a low-cost pop-count computation that replaced the multiplier-accumulator (MAC) operation and utilized an OR gate to implement the comparator in the max-

pooling layer.

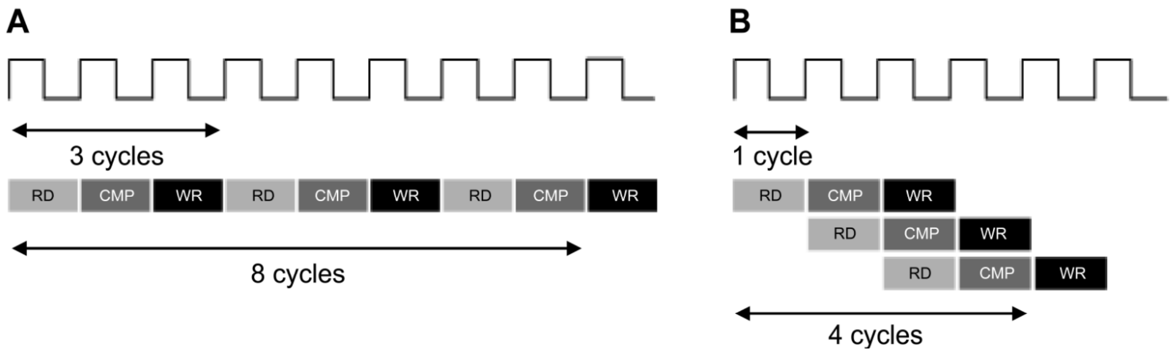
Other several FPGA implementations were proposed to realize the YOLO algorithm too. One such implementation, presented by Preußer, comprised twelve hidden layers to enable programmable logic in Zynq Ultra scale+ FPGA [3.50]. This was an extended version of the design proposed by Umuroglu et al. [3.48]. Finally, a lightweight YOLO v2 combined with a binary network with support vector machine regression was proposed by Arcos-Garcia et al. [3.51].

3.5.3 Proposed HLS implementation and optimization using HLS directives

In this work, we aim to develop a lower area, high throughput implementation of the YOLO v2 algorithm. The model used in the YOLO v2 network was implemented using the Vivado HLS software. Then, it was simulated to test functionality using high-level simulation in HLS (C++ testbench simulation). The synthesizable RTL code was generated using the same platform, and RTL simulation was performed using a nonsynthesizable hand-coded Verilog testbench. The synthesis and simulation results are discussed in more detail in the next section. The “vanilla” or base implementation was optimized using the below-mentioned HLS directives.

i. Pipelining and loop optimization directives

The pipeline directive enables the concurrent execution of operations in a function or a loop [3.11]. Such a directive allows reducing the size of an initiation interval. A pipelined function or loop can process new inputs after every N clock cycles. Here N denotes an initiation interval of a loop or a function, and its value is set to one for the pipeline pragma, which can be changed manually. Fig. 3.14 (a) illustrates the pipelining process of a loop that realizes the operation of a loop to work concurrently. In this figure, three different operations are executed during three clock cycles for a single loop. Therefore, it requires eight clock cycles to complete the execution of three loops. In Fig 3.14 (b), Initiation interval is set to three, and therefore, it requires five clock cycles to finish three loop operations and to obtain the final output.



(A) without pipelining (B) with loop pipelining

Figure 3.14. Effect of the pipelining directive on loops.

ii. Loop Unrolling

Loop unrolling is performed to launch multiple independent operations that can be run in parallel, rather than establishing an individual group of operations being executed serially [3.11]. The unroll pragma converts loops by generating multiple copies of a loop operation in the RTL design. This process permits running all or some loops in parallel. If a loop is rolled (default behavior), then the synthesis tool generates the logic for one iteration of the loop, and the RTL design implements this logic for each iteration of a loop in sequence. The unroll pragma can be used to unroll loops to increase the design's data access throughput. This pragma supports a full or partial unroll. Full unrolling implies creating a copy of the loop body in the RTL for each loop iteration. Therefore, a complete loop can be run in parallel. If the partial unrolling of a loop is performed with a factor of N, then N copies of the loop body are created, thereby reducing the number of loop iterations accordingly.

iii. Latency

Latency is defined as the duration expressed in terms of clock cycles required to produce an output for a given input. The latency pragma can be defined as function latency and loop latency [3.11]. The former is the total number of clock cycles required to compute all output values in a function and return the result. Loop latency is the total number of clock cycles required to execute all iterations in a loop. In Xilinx Vivado HLS, the latency pragma can be used to specify the maximum or minimum latency, or both of them, for the completion of functions, loops, and regions. When the latency pragma is used, the tool attempts to minimize latency in the design by completing a task within a specified number of clock cycles.

Latency is defined as the duration expressed in terms of clock cycles required to produce an output for a given input. The latency pragma can be defined as function latency and loop latency [3.11]. The former is the total number of clock cycles required to compute all output values in a function and return the result. Loop latency is the total number of clock cycles

required to execute all iterations in a loop. In Xilinx Vivado HLS, the latency pragma can be used to specify the maximum or minimum latency, or both of them, for the completion of functions, loops, and regions. When the latency pragma is used, the tool attempts to minimize latency in the design by completing a task within a specified number of clock cycles.

In the implemented code, different functions, such as in_to_buff, wt_to_buff, convolute, and max_pool were used. Accordingly, many loops were used in these functions. While running HLS simulation and using different directives, the reduction in the amount of computation and utilized memory resources was observed.

The synthesis results were analyzed post the usage of HLS directives. As a consequence, in the results, it was observed that timing slack was equal to 0, and the use of resources, including block RAMs (BRAMs), look-up tables (LUTs) and DSPs, was less compared with those required to implement the baseline implementation (without using directives).

We examined the computation process of loops in the program, and for purposes of performance optimization, HLS directives were applied: loop pipelining, loop merging, loop unrolling, interfacing, loop flattening, latency, and expression balancing.

Due to manual transformations applied during the code restructuring step, the opportunities for loop merging and loop flattening were limited. When used, the obtained results demonstrated negative slack, and the resource usage reported was much greater than the available ones. This problem was solved by performing loop pipelining. In some instances, an inner loop's execution required multiple reads and/or writes from/to distinct addresses in the same BRAM. Therefore, in such loops, the initiation interval was prolonged to reflect the latency of multiple BRAM reads and writes.

In some instances, if an inner loop had a small loop bound, and the loop content performed the execution or search operations (rather than memory writes), complete unrolling and pipelining was applied. For example, both the loop unrolling and the pipelining allow multiple loop iterations to be executed in parallel, leading to the increased resource usage (for example, in terms of registers or functional units). However, various optimization ways have improved the performance and the use of resources in several ways. In the proposed YOLO v2 implementation, the best results were observed after applying the pipelining, unrolling, and latency directives. Specifically, loop pipelining and loop unroll were applied to the inner loop of the top function, namely, YOLO v2.

3.5.4 Simulation, FPGA Implementation Results, and Comparison

Simulation results

Using the Vivado HLS model, the obtained functional simulation results were identical to the YOLO algorithm's theoretical results. The test objects were correctly classified as fork, knife, spoon, bowl, etc. in accordance with the test and training images. In addition to the HLS simulation, the RTL (Generated from Vivado HLS) simulation was performed using the xSim simulator and the results were identical. The RTL implementation achieved a throughput of 40 frames per second. Additionally, to have a comparison with golden implementation, an FPGA-in-the-loop simulation was also performed using MATLAB and results were found to be identical.

FPGA synthesis

After the successful functional verification of the proposed design (HLS model), we generated a synthesizable RTL and implemented it on a target FPGA device, namely, Xilinx Zynq xc7z020clg484-1. Various directives were applied to the code aiming to optimize it, including pipeline, loop unrolling, function in-lining, etc. The FPGA synthesis results obtained after the application of multiple directives, one at a time, are summarized in Table 3.11. The table indicates that the combined use of directives (pipeline, unroll, and latency) provides the best implementation results (the last row in Table 3.11).

Table 3.11. Implementation results on Zynq for YOLO v2 using HLS directives.

Directive used	Frequency (MHz)	BRAM	DSP	Flip-flops	LUTs	Slack (ns)
Without Directives	176	182	151	27606	420241	−5.4 (Not met)
Pipeline	220	182	164	27647	40180	0
Pipeline + Loop Unroll	220	182	38	15943	19297	0
Latency (min = 20, max = 200)	220	182	30	14655	18601	0

LUT: Look-up table; RAM: random access memory; BRAM: block RAM, IO: input/output.

Total On-chip Power = 0.461 W

Energy dissipation, E = P × T, where P is power dissipation and T is time required to process one input frame. Since frame rate is 40fps, therefore T = 1/40 = 0.025 s.
Hence, energy dissipation, E = 0.461 × 0.025 = 11.5 mJ (Milli Joules) per cipher.

Results of comparison with the literature

Our implementation was compared with other proposed implementations available in the literature [3.52, 3.53]. Table 3.12 shows the comparison with other implementations available in the literature targeted for Xilinx Zynq Ultrascale+ and Virtex 7 FPGAs. Nakahara et.al [3.52] have used binarized CNN with pipeline based architecture for their design. Full parallelization of all CNN layers in the design was also implemented by Nguyen et.al [3.53] in their design.

Table 3.12. Comparison of the YOLO v2 Implementation results

Metric	Proposed design	Proposed design	Nakahara et.al [3.52]	Nguyen et.al [3.53] Tiny	Nguyen et.al [3.53] Sim
FPGA	Zynq	Virtex-7	Zynq	Virtex-7	Virtex-7
Frequency (MHz)	220	176	300	200	200
BRAM	182	234	1706	1026	1144
DSPS	30	68	377	168	272
Flip-flops	14655	9200	135000	86000	155000
LUTs	18061	9120	370000	60000	115000

As can be seen from Table 3.12, the resource usage for the proposed implementation decreased by an order of magnitude when compared with other implementations proposed in the literature. Functionally, we achieved a mean average precision of 0.48 at intersection of union of 0.5. The generated RTL implementation was able to achieve a frame rate of 40 frames per second and took a total energy of 11.5 milli Joules per frame. The same is not reported by other researchers for a direct comparison. It is clear from the above results that our implementation is superior without any effect or change in functionality.

3.6 Concluding remarks

In this chapter, we studied that different HLS tools offer multiple directives which one can use to optimize a particular application for area, speed, or power on a target FPGA. Continuing from the prior works in the same domain, we have explored multiple tools and applications and understood the effect of HLS directives on those applications. Five different designs which find place in multiple applications were used and optimized using HLS directives. As an example, loop unroll directive (which unrolls a loop to create multiple copies in hardware) can help achieve better performance at cost of slight increase in area. But this will be effective if the application is architected with help of loops in high level language. Similarly, resource sharing HLS directive will be effective if a common resource (like a multiplier) is shared between two or more data paths in the design. Three unique application designs were created using MATLAB HDL coder and two unique designs were created using Vivado HLS as part of this work and synthesis results were optimized after using applicable directives. All these designs were thoroughly verified using xSim RTL simulation along with FPGA-in-the- loop simulation feature of MATLAB HDL Verifier. The verification results proved that HLS optimization directives did not have any effect on the design functionality. Finally, the optimized designs were implemented on target FPGAs. A comparison of synthesis results for all these five optimized application designs with other implementations published in literature was also presented. It was noted that directives helped to achieve better synthesis characteristics of area, speed and power depending on the usage and application. However, the outcome of this exercise is that none of the directives are generic but application and tool specific.