Low-Cost and Programmable CRC Implementation based on FPGA (Extended Version)

Huan Liu [1,1,1], Zhiliang Qiu [2], Weitao Pan [2], Jun Li [2], Ling Zheng [2], and Ya Gao [2]

[1]Xidian University
[2]Affiliation not available

November 8, 2023

Abstract

Cyclic redundancy check (CRC) is a well-known error detection code that is widely used in Ethernet, PCIe, and other transmission protocols. The existing FPGA-based implementation solutions are faced with the problem of excessive resource utilization in high-performance scenarios. The padding zeros problem and the introduction of programmability further exacerbate this problem. In this brief, the stride-by-5 algorithm is proposed to achieve the optimal utilization of FPGA resources. The pipelining go back algorithm is proposed to solve the padding zeros problem. The method of reprogramming by HWICAP is proposed to realize programmability with a small and constant resource utilization. The experimental results show that the resource utilization of proposed non-segmented architecture is 84.1% and 37.6% lower than those of two state-of-the-art FPGA-based CRC implementations, and the proposed segmented architecture has a lower resource utilization by 83.9% and 8.9% compared wtih the two state-of-the-art architectures; meanwhile, the throughput and programmability are guaranteed. We made the source code available on GitHub.

Low-Cost and Programmable CRC Implementation based on FPGA (Extended Version)

Huan Liu, Zhiliang Qiu, Weitao Pan, Jun Li, Ling Zheng and Ya Gao

***Abstract*—Cyclic redundancy check (CRC) is a well-known error detection code that is widely used in Ethernet, PCIe, and other transmission protocols. The existing FPGA-based implementation solutions are faced with the problem of excessive resource utilization in high-performance scenarios. The padding zeros problem and the introduction of programmability further exacerbate this problem. In this brief, the stride-by-5 algorithm is proposed to achieve the optimal utilization of FPGA resources. The pipelining go back algorithm is proposed to solve the padding zeros problem. The method of reprogramming by HWICAP is proposed to realize programmability with a small and constant resource utilization. The experimental results show that the resource utilization of proposed non-segmented architecture is 80.7%-87.5% and 25.1%-46.2% lower than those of two state-of-the-art FPGA-based CRC implementations, and the proposed segmented architecture has a lower resource utilization by 81.7%-85.9% and 2.9%-20.8% compared wtih the two state-of-the-art architectures; meanwhile, the throughput and programmability are guaranteed. We made the source code available on GitHub[1].**

***Index Terms*—Cyclic redundancy check, FPGA, low cost, programmable, HWICAP.**

I. Introduction

As the throughput of networks is on a constant rise, increasingly more packet processing tasks are being offloaded to the FPGA-based SmartNIC[2], including the generation and verification of cyclic redundancy check (CRC). The 400G and the coming multi-terabit Ethernet demand faster CRC caculations, and the implementation of high-performance CRC calculations based on FPGAs needs to meet three requirements: 1) *Reduce parallelization cost.* The end of Dennard scaling[3] results in a bottleneck for improving the frequency of integrated circuits, and higher throughput means a wider bus inside chips. The slicing-by-4 and slicing-by-8 algorithms are proposed for parallel processing in[4], which is suitable for CPU but not optimal for FPGA[5]. 2) *Solve the padding zeros problem.* The parallelization means that the final word of a transaction is composed of valid bytes along with padding zeros. The number of padding zeros is uncertain, and CRC calculations using the complete final word would cause an erroneous result, which is called the padding zeros problem. Multiple tables can be used to process the final word, and every table corresponds to a possible length of valid bytes[6]. The scheme introduces an $O\left(n^2\right)$ resource utilization when the bus width is n bits. [7][8] is one of the state-of-the-art schemes for solving this problem. The tables for the final word are organized in the manner of a pipeline, and each pipeline step corresponds to one layer of a binary search tree. An $O(n)$ resource utilization is introduced. 3) *Keep programmability.* A programmable implementation of the CRC algorithm can achieve better reusability; thus, a wide range of applications can be supported without circuit modification. The demand can be found in iSCSI[9] and P4[10]. A specific circuit architecture is used to guarantee programmability[11][12], but it is not suitable for FPGA. [5] is one of the state-of-the-art schemes that is suitable for FPGA, but it requires a complex configuration circuit that leads to a large resource utilization increase with the bus width.

All three of the aforementioned requirements lead to a considerable resource utilization. Although slicing[4][5], aggressive strides, simultaneous processing of multiple streams[7] and many other principles behind CRC acceleration are well known, they can't achieve low cost, high performance and programmability at the same time. A multi-core, multi-socket system with Intel's CRC instruction[13] can achieve high throughput, but they suffer from high latency and high power consumption in packet processing applications. In this brief, two algorithms and a method corresponding to the three requirements are proposed to decrease the resource utilization with guaranteed throughput and programmability. First, the stride-by-5 algorithm is proposed, which can reduce the resource utilization by 79.69%-79.98% compared with the slicing-by-4 and slicing-by-8 algorithms. Second, the pipelining go back algorithm is proposed to solve the padding zeros problem, which will introduce an $O\left(\log_2 n\right)$ resource utilization. Finally, a hardware internal configuration access port (HWICAP) is used to realize dynamic programmability, and it leads to a small and constant resource utilization regardless of the bus width.

The remainder of this brief is organized as follows. Section II provides preliminaries to our proposals. Section III discusses the system architecture and the three proposals. Section IV shows the synthesis results and the board-level implementation results. Section V concludes this brief.

This work is supported in part by the National Key Laboratory Foundation (HTKJ2019KL504012), National Natural Science Foundation (61502204).

H. Liu, Z. Qiu, and W. Pan are with the State Key Laboratory of Integrated Service Networks, Xidian University, Xi'an 710071, China (e-mail: huanliu@stu.xidian.edu.cn).

J. Li is with the National Key Laboratory of Science and Technology on Space Microwave, China Academy of Space Technology (Xi'an), Xi'an 710100, China.

Ling Zheng is with the School of Communication and Information Engineering, Xi'an University of Posts and Telecommunications, Xi'an 710121, China.

Ya Gao is with the School of Internet of Things Technology, Wuxi Institute of Technology, Wuxi 214121, China.

Corresponding author: Weitao Pan (wtpan@mail.xidian.edu.cn)

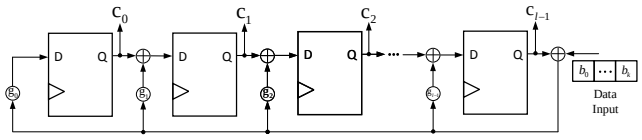


Fig. 1: LFSR for CRC computing.

II. PRELIMINARIES

A. FPGA LUT Architecture

The basic logic resource of modern Xilinx FPGAs is look-up tables (LUTs), which can be considered a RAM with five inputs and two outputs[14]. A truth table can be stored in a LUT, and two logical equations with the same five inputs can be realized using it. This is an important property that will be used in the stride-by-5 algorithm. LUT is the most consumed resource in FPGA-based CRC implementations, and the number of consumed LUTs is used as the indicator of the resource utilization.

B. Serial and Parallel CRC Algorithms

Serial CRC Algorithm

The CRC algorithm is a long division performed with modulo-2 arithmetic. The dividend is a polynomial $B(x)$ whose coefficient is the input data. The divisor is a given polynomial $G(x)$, and the coefficient of the remainder $R(x)$ is the wanted CRC value. Addition and subtraction can be realized by the xor operation in GF(2), and "+" means xor in the remainder of this brief. The aforementioned division can be realized by the linear feedback shift register (LFSR), as shown in Fig. 1.

The coefficient of $G(x)$ is $[g_l, g_{l-1}, \ldots, g_0]$. The coefficient of $B(x)$ is $[b_0, b_1, \ldots, b_k]$, with b_0 being the most significant bit. The initial value of the LFSR is $C^{(0)} = \left[\mathbf{c}_{l-1}^{(0)}, \mathbf{c}_{l-2}^{(0)}, \cdots, \mathbf{c}_0^{(0)}\right]^T$. The value of the LFSR is $C^{(m)}$ when bit b_{m-1} enters the LFSR, and it will become $C^{(m+1)}$ after bit b_m enters the LFSR. We can obtain the relationship between $C^{(m)}$ and $C^{(m+1)}$ from Fig. 1, which is

$$C^{(m+1)} = TC^{(m)} + Sb_m \tag{1}$$

where T is a matrix of size $l \times l$. S is a column vector of size l, and

$$T = \begin{bmatrix} g_{l-1} & 1 & 0 & \cdots & 0 \\ g_{l-2} & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ g_1 & 0 & 0 & \cdots & 1 \\ g_0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{2}$$

$$S = [\mathrm{g}_{l-1}, \mathrm{g}_{l-2}, \ldots, \mathrm{g}_0]^T \tag{3}$$

Parallel CRC Algorithm

The parallel CRC algorithm can process multiple data input bits simultaneously[15], and its theoretical equation can be derived from (1). The number of bits processed in parallel is n, which is also the width of the inner bus in the remainder of this brief. The parallel input data are $B_n = [b_0, b_1, \ldots, b_{n-1}]^T$. The value of the LFSR is $C^{(k)}$ before B_n enters. The relationship between $C^{(n+k)}$ and $C^{(k)}$ is

$$\begin{aligned} C^{(n+k)} &= TC^{(n+k-1)} + Sb_{n-1} \\ &= T^nC^{(k)} + T^{n-1}Sb_0 + T^{n-2}Sb_1 + \cdots + Sb_{n-1} \\ &= T^nC^{(k)} + W_{ln}B_n \end{aligned} \tag{4}$$

where W_{ln} is a matrix of size $l \times n$ and

$$W_{ln} = \left[T^{n-1}S, T^{n-2}S, \ldots, TS, S\right] \tag{5}$$

T^n and W_{ln} can be calculated by equations (2), (3) and (5) after $G(x)$ is given, and parallel processing by n bits can be achieved by equation (4).

C. Programmability and HWICAP

T^n and W_{ln} are generally stored inside LUTs for the FPGA-based implementation of CRC algorithms, and a programmable implementation requires the ability to modify the content of the LUTs at runtime. Previous research using logic resources (LUTs and registers) to realize configuration logic would lead to several thousands of LUTs consumed when $n \geq 1024$[5].

HWICAP is an Xilinx IP core that can afford users with access to ICAP primitives using the AXI4-Lite protocol[14]. It can modify the content of the LUTs dynamically. The resource utilization of HWICAP is as low as 186 LUTs, and it will not increase with increasing inner bus width. For the Intel/Altera FPGAs, similar function can be achieved by using PR-IP[16][17].

III. PROPOSED WORK

A. Non-Segmented System Architecture

The proposed non-segmented system architecture is shown in Fig. 2. Non-segmented system architecture means that there should be one frame in a single word, and segmented system architecture can process multiple frames at the same time[18]. Region 1 and Region 2 correspond to the computation of $W_{ln}B_n$ in (4). Region 1 consumes most of the LUTs, and the number of consumed LUTs linearly depends on the size of W_{ln}. The stride-by-5 algorithm, which is discussed in Section B, is proposed to reduce the LUT consumption of Region 1. Region 2 is implemented by the means of an xor tree instead of a one-stage xor function to achieve higher performance. Region 3 completes the computation of (4). It consumes few LUTs for the small size of $\boldsymbol{T}^n$. The padding zeros problem is solved by Region 4, and the pipelining go back algorithm, which results in an $O(\log_2 n)$ resource utilization, is proposed and discussed in Section C. Region 5 is a HWICAP controller that can modify the content of the LUTs dynamically. The operation procedure is discussed in Section D. A segmented system architecture is proposed in Section E. The packet processing flow of the two system architectures are illustrated in Section F.

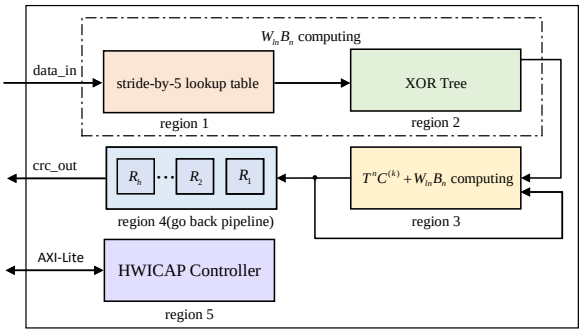


Fig. 2: Proposed non-segmented system architecture.

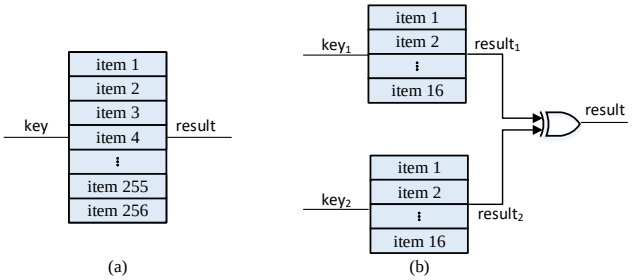


Fig. 3: Function implementation with (a) stride-by-8 and (b) stride-by-4.

B. Stride-by-5 Algorithm

In this section, the model of the resource utilization is established, the stride-by-5 is proven to be the best stride for various bus widths, and the stride-by-5 algorithm is described in Algorithm 1.

Stride, as its name implies, means the number of bits processed by a single logical table[19]. The logical table can be realized using FPGA LUTs, and it can load the truth table of a function. For example, an 8-input function is defined as

$$y = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \tag{6}$$

which can be transformed equivalently as

$$\begin{cases} y_1 = x_1 + x_2 + x_3 + x_4 \\ y_2 = x_5 + x_6 + x_7 + x_8 \\ y = y_1 + y_2 \end{cases} \tag{7}$$

Equations (6) and (7), whose strides are 8 and 4, can be implemented as shown in Fig. 3(a) and Fig. 3(b), respectively. A smaller stride means that a smaller logical table can be realized by a single LUT or cascaded LUTs. Can stride-by-1 be considered the best stride for FPGA implementation? We will establish the resource utilization model and find the answer.

l equations with the same n inputs are required to realize the computation of $W_{ln}B_n$ in (4). n is also the bus width, and

$$n = ms + r \tag{8}$$

in which s is stride. m equals $\lfloor n/s \rfloor$. r means remainder, which equals $n \bmod s$.

The function $A(x)$ is defined as

$$A(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0 \end{cases} \tag{9}$$

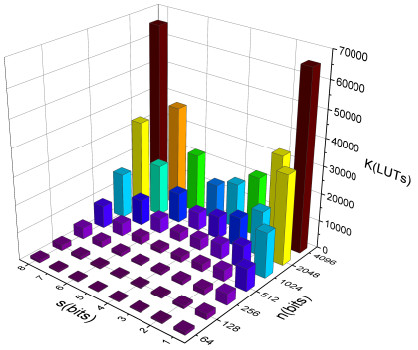


Fig. 4: The relationship between $K_{R_1}(n, s, 32)$, n and s.

and the resource utilization function $K(m, s, r, l)$ is defined as

$$K(m, s, r, l) = \begin{cases} (l/2) \cdot (m + A(r)) & s \le 5 \\ (l/2) \cdot (m \cdot 2^{s-5} + A(r)) & s > 5, r \le 5 \\ (l/2) \cdot (m \cdot 2^{s-5} + A(r) \cdot 2^{r-5}) & s > 5, r > 5 \end{cases} \tag{10}$$

There are three equations corresponding to different s and r. A single LUT is required to realize a logical table when s is smaller than five, and cascaded LUTs are needed to realize a logical table when s is larger than five. This is because a single LUT has five inputs. The computation of the remainder r is the same as that of the stride s. l is divided by 2 for the two outputs of a single LUT. (10) can be simplified as

$$K(n, s, l) = \begin{cases} (l/2) \cdot (\lfloor n/s \rfloor + A(n \bmod s) & s \le 5 \\ (l/2) \cdot (\lfloor n/s \rfloor \cdot 2^{s-5} + A(n \bmod s)) & s > 5, n \bmod s \le 5 \\ (l/2) \cdot (\lfloor n/s \rfloor \cdot 2^{s-5} + A(n \bmod s) \cdot 2^{(n \bmod s)-5}) & s > 5, n \bmod s > 5 \end{cases} \tag{11}$$

We should determine the values of l and s before exploring the relationship between K and s and find the best s. l is related to the generator polynomial, and it is set to 32 here because CRC32 is the most widely used polynomial. The value of n is set to $[64, 128, 256 \cdots 4096]$ to explore the influence of the bus width. The value of s is set to $[1, 2, 3 \cdots 8]$. A stride larger than 8 will lead to an excessively large resource utilization. The resource utilization of Region 1 is $K_{R_1}(n, s, 32)$, and the relationship between $K_{R_1}(n, s, 32)$, n and s is illustrated in Fig. 4. As shown, stride-by-5 is optimal for any bus width. Stride-by-5 reduces the resource utilization by 79.69%-79.98% compared with stride-by-8, which is used in the slicing-by-4 and slicing-by-8 algorithms[4].

The stride-by-5 algorithm is optimal for the 5-input LUTs in FPGA. Because the cascaded LUTs are needed if the stride is larger than 5, a single LUT cannot be fully used if the stride is smaller than 5. For the FPGAs with non-5-input LUTs (prior to Xilinx Virtex-5 or Altera Stratix-II), the stride defined by the number of LUT inputs should be used, and the LUT sharing mechanism should be exploited. The stride-

by-5 algorithm is described in Algorithm 1; it processes the computation in Region 1 here, but the algorithm can also be used in Regions 3 and Region 4.

Algorithm 1 Stride-by-5 algorithm.

Input: The bus width n. The stride s. The input vector $B[n]$. The computing matrix $W[l][n]$.
Output: The meta matrix $MD[l][m+1]$, which is the input of Region 2.

```
Initialize s to 5;
Initialize m to ⌊n/s⌋;
Initialize MD[l][m+1] to the null matrix;
for i = 0 to l − 1 do
    for j = 0 to m − 1 do
        for k = 0 to s − 1 do
            MD[i][j] = MD[i][j] ⊕ (B[j × s + k] · W[i][j × s + k]);
        end for
    end for
end for
for i = 0 to l − 1 do
    for j = s · m to n − 1 do
        MD[i][m] = MD[i][m] ⊕ (B[j] · W[i][j]);
    end for
end for
```

C. Pipelining Go Back Algorithm

In this section, the pipelining go back algorithm is proposed with an $O(\log_2 n)$ resource utilization, and the derivation and description of the algorithm are presented.

The padding zeros problem is discussed in Section I. p is used to represent the number of valid bits in the final word. q is used to represent the number of padding zeros. The data vector of the final word is $B_{p+q} = [\mathrm{b}_0, \cdots, \mathrm{b}_{p-1}, 0, \cdots, 0]^T$. Substitute B_{p+q} into (4), and then

$$\begin{aligned} C^{(p+q+k)} &= T^{p+q}C^{(k)} + W_{(p+q)n}B_{p+q} \\ &= T^q\left(T^pC^{(k)} + T^{p-1}Sb_0 + \cdots + Sb_{p-1}\right) \\ &= T^qC^{(p+k)} \end{aligned} \tag{12}$$

The relationship between $C^{(p+k)}$ and $C^{(p+q+k)}$ is

$$C^{(p+k)} = T^{-q}C^{(p+q+k)} \tag{13}$$

There will be an $O(1)$ resource utilization to realize the computation of T^{-q} because the size of T^{-q} is $l \times l$ and has no relation with n. However, q varies and $0 \le q < n$, and if we use the n table corresponding to every possible q, there will be an $O(n)$ resource utilization. We introduce a pipeline to reduce the resource utilization to $O(\log_2 n)$.

Inspired by the binary representation, q is represented as

$$q = 8 \cdot \left(x_{h-1} \cdot 2^{h-1} + x_{h-2} \cdot 2^{h-2} + \cdots + x_1 \cdot 2 + x_0\right) \tag{14}$$

q and n are multiples of 8 because the data transfer in bytes. The value of x can be 0 or 1, and h is the number of pipeline stages, which can be represented as

$$h = \log_2(n/8) \tag{15}$$

(14) and (15) can be used to convert (13) to

$$\begin{aligned} C^{(p+k)} &= \left(\left(T^{-8\cdot 2^{n-1}}\right)^{x_{n-1}} \cdots \left(T^{-8}\right)^{x_0}\right) C^{(p+q+k)} \\ &= \left(R_1^{x_{n-1}} \cdot R_2^{x_{n-1}} \cdots R_h^{x_0}\right) C^{(p+q+k)} \end{aligned} \tag{16}$$

where $[R_1, R_2, \ldots, R_h]$ is the h matrices for the h-stage pipeline, and the size of each matrix is $l \times l$. $[R_1, R_2, \ldots, R_h]$ can be used to convert $C^{(p+q+k)}$ to $C^{(p+k)}$. Stride-by-5 algorithm can be used to convert $[R_1, R_2, \ldots, R_h]$ to the content of the LUTs. Using the resource utilization function in (11), the resource utilization of the pipeline is $K_{R_4} = h \cdot K(l, s, l)$, where R_4 means the Region 4 in Fig. 2. K_{R_4} can be represented as

$$K_{R_4} = \begin{cases} \log_2(n/8) \cdot (l/2) \cdot (\lfloor l/s \rfloor + A(l \bmod s)) & s \le 5 \\ \log_2(n/8) \cdot (l/2) \cdot (\lfloor l/s \rfloor \cdot 2^{s-5} + A(l \bmod s)) & s > 5, n \bmod s \le 5 \\ \log_2(n/8) \cdot (l/2) \cdot (\lfloor l/s \rfloor \cdot 2^{s-5} + A(l \bmod s) \cdot 2^{(l \bmod s)-5}) & s > 5, n \bmod s > 5 \end{cases} \tag{17}$$

As shown in (17), we can achieve an $O(\log_2 n)$ resource utilization using the pipelining go back algorithm. The algorithm is described in Algorithm 2.

Algorithm 2 Pipelining go back algorithm.

Input: The temporary CRC value $C^{(p+q+k)}$, the bus width n, the number of padding zeros q, the computing matrix T.
Output: The wanted CRC value $C^{(p+k)}$.

```
Initialize h to log2(n/8), q to q/8;
Initialize matrix R to null matrix, matrix C^(p+k) to C^(p+q+k);
for i = h − 1 to 0 do
    if q ≥ 2^i then
        R = T^(−8·2^i);
        C^(p+k) = R C^(p+k);
        q = q − 2^i;
    else
        continue ;
    end if
end for
```

D. Reprogramming by HWICAP

Region 5 in Fig. 2 represents an HWICAP IP core, which can dynamically modify the content of the LUTs. It consumes 186 LUTs for any bus width. In contrast, configuration logic realized by logic resources leads to several thousands of LUTs being consumed when $n \ge 1024$ [5], and the resource utilization increases with increasing bus width. The operation procedure of reprogramming using the HWICAP IP core is described as follows:

1) Complete the initial design, generate the bitstream using Vivado, and download the bitstream into the FPGA;
2) Extract the locations of the LUTs used;
3) When reprogramming is needed, compute the new content of the LUTs using (4) and (16);
4) Map the content of the LUTs to the initial value of the LUTs;
5) Write the initial value to the LUTs using the AXI Lite interface of the HWICAP IP core.

The method of reprogramming by HWICAP is useful in engineering. Our contributions are as follows:

1) We verify the feasibility of reprogramming the FPGA implementation of the CRC algorithm using the HWICAP IP core. It leads to a small and constant resource utilization regardless of the bus width;

2) The proposed method can change the CRC polynomial directly, without re-coding and re-synthesizing;
3) The code of the above procedure can be accessed in [1], as a part of the entire project. To our best knowledge, this is the first open source code covering the whole procedure described above.

E. Segmented System Architecture

Non-segmented system architecture can't process multiple frames in one word (clock), which decreases the throughput of short or misaligned frames. It is called bus efficiency problem. The segmented system architecture is proposed to solve the problem. The bus format is just like that in [7], and the block in [7] is another name for the segment in [18]. For an example, a 4096-bit bus can process 8 complete frames at the same time; hence, the bus can be divided into 8 regions [7]. The number of regions only depends on the bus width. Different segment widths are feasible, and if 64-bit segment width is chosen, one region can be divided into 8 segments (blocks). The proposed segmented system architecture is shown in Fig. 5. Compared with the proposed non-segmented system architecture, proposed segmented system architecture has slightly more complex Region 1 and Region 2, and multiple duplicates of Region 3 and Region 4. The number of duplicates is just the maximum number of the frames processed in a single word.

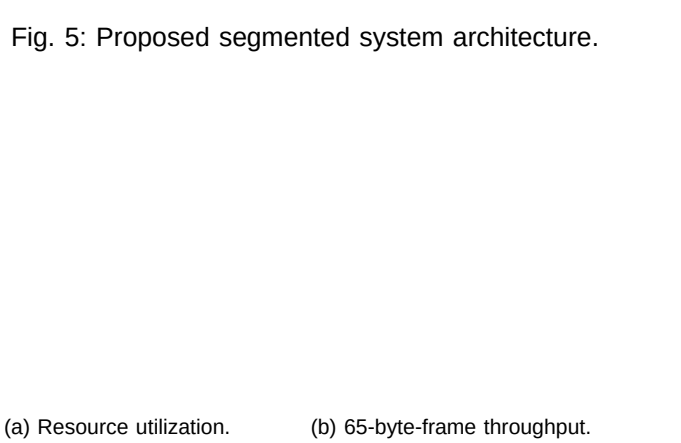
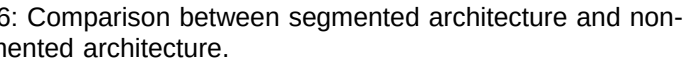


Fig. 5: Proposed segmented system architecture.

The comparison between the proposed segmented system architecture and the proposed non-segmented system architecture can be found in Fig. 6. The red cuboid represent the non-segmented system architecture. The blue cuboid represent the increment between the proposed segmented system architecture and the proposed non-segmented system architecture. The yellow slice (Bus width = 1024, Segment width = 512) represent the decrement between the two architectures. Fig. 6a shows that the increment in resource utilization mainly depends on the bus width instead of segment width. This is because the increment in resource utilization mainly depends the number of duplicates of Region 3 and Region 4, which only depends on the bus width. Fig. 6b shows that the increment in 65-byte-frame throughput is obvious for most cases. The only decrese in throughput can be found when the bus width is 1024 bits and the segment width is 512 bits, where the two architectures have the same bus efficiency for 65-byte-frame throughput and the non-segmented architecture has a slightly higher frequency. 64 bits is chosen as the segment width in the rest of the brief. The dataset of the Fig. 6 can be found at [1].



(a) Resource utilization. (b) 65-byte-frame throughput.

Fig. 6: Comparison between segmented architecture and non-segmented architecture.

F. Packet Processing Flow of Two System Architecture

In this section, the packet processing flow of the two system architectures are illustrated. The bus width of the two architectures is 4096 bits, and the segment width of the segmented system architecture is 64 bits (64 segments, 8 regions).

Non-Segmented System Architecture

The packet processing flow of non-segmented system architecture is described as follows:

1) Region 1: Data input is composed of 512-bit frame and 3584-bit padding zeros; Region 1 compute the $MD[l][m+1]$ in Algorithm 1, where $l = 32$ and $m = 819$;
2) Region 2: Data input is a $32 \quad 820$ bit vector; Region 2 compute the $W_{ln}B_n$ in (4);
3) Region 3: Data input is the 32-bit $W_{ln}B_n$; Region 3 compute the $T^nC^{(k)} + W_{ln}B_n$ in (4);
4) Region 4: Data input is the 32-bit $T^nC^{(k)} + W_{ln}B_n$; Region 4 eliminates the impact of padding zeros ((16) is used here), and the correct CRC value is achieved.

Segmented System Architecture

The packet processing flow of segmented system architecture is described as follows:

1) Region 1: Data input is composed of 8 512-bit frames, which can be divided into 64 segments; Every segment has its own sub-region to compute the $MD[l][m+1]$ in Algorithm 1; Every $MD[l][m+1]$ is a $32 \quad 13$ bit vector, and a small xor function is use to convert the $MD[l][m+1]$ to $W_{ln}B_n$.
2) Region 2: Data input is 64 $W_{ln}B_n$ of 64 segments; Region 2 merge the $W_{ln}B_n$ of the same frame; 8 $W_{ln}B_n$ of the 8 frames are achieved.
3) Region 3 and 4: Data input is the 8 $W_{ln}B_n$ of 8 frames; Every $W_{ln}B_n$ has its own Region 3 and 4, and the Region 3 and 4 here do the same work as that of the non-segmented architectures. Finally, 8 CRC values for 8 frames can be achieved.