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

Huan Liu, Zhihong 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 with 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 calculations, 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 $O(n^2)$ 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 $O(\log_2 n)$ 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.
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 can be realized by the linear feedback shift register (LFSR), as shown in Fig. 1. The coefficient of \( g(x) \), \( c(0) \), \( c(1) \), \( c(2) \), \( c(3) \), with \( b_0 \) being the most significant bit. The initial value of the LFSR is \( C^{(0)} = \begin{bmatrix} c(0) & c(1) & c(2) & c(3) \end{bmatrix}^T \). 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

\[
C^{(m+1)} = TC^{(m)} + Sb_m
\]  

where \( T \) is a matrix of size \( l \times l \), \( S \) is a column vector of size \( l \), and

\[
T = \begin{bmatrix}
g_{l-1} & 1 & 0 & \cdots & 0 
g_{l-2} & 0 & 1 & \cdots & 0 
\vdots & \vdots & \ddots & \ddots & \vdots 
g_1 & 0 & 0 & \cdots & 1 
g_0 & 0 & 0 & \cdots & 0
\end{bmatrix}
\]

\[
S = \begin{bmatrix} g_{l-1}, g_{l-2}, \ldots, g_0 \end{bmatrix}^T
\]  

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 \( W_{in}B_n \) in (4). Region 1 consumes most of the LUTs, and the number of consumed LUTs linearly depends on the size of \( W_{in} \). 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 \( T^n \). 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.
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

\[ y = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \]  

which can be transformed equivalently as

\[
\begin{align*}
  y_1 &= x_1 + x_2 + x_3 + x_4 \\
  y_2 &= x_5 + x_6 + x_7 + x_8 \\
  y &= y_1 + y_2
\end{align*}
\]  

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.

Let equations with the same \( n \) inputs are required to realize the computation of \( W_{16}B_n \) in (4). \( n \) is also the bus width, and

\[ n = ms + r \]  

in which \( s \) is stride, \( m \) equals \( \lfloor n/s \rfloor \), \( r \) means remainder, which equals \( n \mod s \).

The function \( A(x) \) is defined as

\[ A(x) = \begin{cases} 
  0 & x = 0 \\
  1 & x > 0 
\end{cases} \]  

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

\[
K(m, s, r, l) = \begin{cases} 
  (l/2) \cdot (m + A(r)) & s \leq 5 \\
  (l/2) \cdot (m \cdot 2^{s-5} + A(r)) & s > 5, r \leq 5 \\
  (l/2) \cdot (m \cdot 2^{s-5} + A(r) \cdot 2^{r-5}) & s > 5, r > 5 
\end{cases}
\]  

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

\[
K(n, s, l) = \begin{cases} 
  (l/2) \cdot (\lfloor n/s \rfloor + A(n \mod s)) & s \leq 5 \\
  (l/2) \cdot (\lfloor n/s \rfloor \cdot 2^{s-5} + A(n \mod s)) & s > 5, n \mod s \leq 5 \\
  (l/2) \cdot (\lfloor n/s \rfloor \cdot 2^{s-5} + A(n \mod s) \cdot 2^{n \mod s-5}) & s > 5, n \mod s > 5 
\end{cases}
\]  

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-
The value of no relation with we use the vector of the final word is used to represent the number of valid bits in the final word.

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 $B_{p+q} = [b_0, \ldots, b_{p-1}, 0, \ldots, 0]$. Substitute $B_{p+q}$ into (4), and then

$$C^{(p+q+k)} = T^{p+q}C^{(k)} + W_{p+q}B_{p+q}$$

$$= T^q \left( T^p C^{(k)} + T^{p-1} S_{b_0} + \cdots + S_{b_{p-1}} \right) + \sum_{j=0}^{q} S_{j}$$

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

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

Theorem 1: the resource utilization of $C^{(p+k)}$ is $O(1)$.

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 $n \geq 1024$ [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.

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) represents the decrement between the two architectures.

Fig. 5: Proposed segmented system architecture.

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

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

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 $32 \times 820$ bit vector; Region 2 compute the $W_{1n}B_n$ in (4);
3) Region 3: Data input is the 32-bit $W_{1n}B_n$; Region 3 compute the $T_nC(k) + W_{1n}B_n$ in (4);
4) Region 4: Data input is the 32-bit $T_nC(k) + W_{1n}B_n$; 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 $32 \times 13$ bit vector, and a small xor function is used to convert the $MD[l][m+1]$ to $W_{1n}B_n$.
2) Region 2: Data input is $64 \ W_{1n}B_n$ of 64 segments; Region 2 merge the $W_{1n}B_n$ of the same frame; $8 \ W_{1n}B_n$ of the 8 frames are achieved.
3) Region 3 and 4: Data input is the $8 \ W_{1n}B_n$ of 8 frames; Every $W_{1n}B_n$ has its own Region 3 and 4, and the Region 3 and 4 do the same work as that of the non-segmented architectures. Finally, 8 CRC values for 8 frames can be achieved.

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:
IV. EXPERIMENTAL RESULTS

A. Synthesis Result

There are three state-of-the-art studies[7][5][20]. Architecture in [5][20] can be reprogrammed, whereas architecture in [7] cannot be reprogrammed. Two proposed architectures are implemented with Virtex-7 XC7VX690T, and [7][5][20] are implemented with Virtex-7 XCVH870T, Virtex-6 XC6VLX550T and Stratix-V 5SGSED6N1F45I2, respectively. In this section, two proposed architecture are compared with their works in terms of resource utilization and maximum throughput. The proposed segmented architecture is compared with the architecture in [7] in terms of throughputs on all frame lengths. Architecture in [7] and proposed two architectures are based on CRC32, and architecture in [5] is based on CRC64; hence, the resource utilization in [5] halves for a fair comparison. Power consumption of two proposed architectures is reported at last. In the following we use the SA to refer to the segmented architecture.

The synthesis result is illustrated in Fig. 7. Fig. 7a shows that the resource utilization of the proposed non-SA is lower than that of architectures in [5] and [7] by 80.7%-87.5% and 25.1%-46.2%, respectively. The proposed SA has a lower resource utilization by 81.7%-85.9% and 2.9%-20.8%. The lower resource utilization results from the algorithms and method in Section 3, which can also guarantee high performance and programmability. If the programmability is not required, the resource utilization can be furtherly decreased. In contrast, the architecture in [5] is based on a slicing-by-N algorithm whose stride is 8, which will lead to a much higher resource utilization. Furthermore, the complex configuration logic of the architecture in [5] also leads to a considerable resource utilization. The architecture in [7] cannot be reprogrammed, which will reduce much resource utilization, even though its resource utilization is higher than that of two proposed architectures. Architecture in [7] is not concerned with the stride; hence, the LUT resource cannot be fully utilized, which leads to a higher resource utilization. Architecture in [20] has a lower resource utilization than that of the non-SA by 74.4%-81.3%. The reasons for the lower resource utilization of [20] are: 1). [20] only need to process half filled and fully filled packets. In word other, the padding zeros problem is partly addressed. By contrast, two proposed architectures and [7][5] can fully address the padding zeros problem; 2). The cost of Nios II IP core is not taken into account in [20]. By contrast, proposed two architectures take the cost of HWICAP into account. Moreover, it is difficult to scale the bus width of [20] up to 1024 bits.

Fig. 7b shows that the maximum throughput of proposed non-SA is higher than that of the architecture in [5][7][20] by 24.2%-37.9%, 37.4%-75.0% and 259.4%-284.5%, respectively. The maximum throughput of the proposed SA is higher than that of the architecture in [5][7] by 28.7%-30.2% and 32.2%-80.2%, respectively. The higher frequency leads to higher throughput, and the proposed two architectures can achieve higher frequency for the well-arranged pipelines in Region 1, Region 2 and Region 4.

The throughput on frame lengths from 64 bytes to 256 bytes can be found in Fig. 7c. The architecture in [5] and proposed non-SA can't process multiple frames at the same time; hence, only the architecture in [7] and proposed SA are compared. The two architectures use 4096-bit bus width and 64-bit segment width, therefore they have the same bus efficiency. The proposed SA has a 80.2% higher frequency and throughput than the architecture in [7]. The lowest throughput of 1933.9Gbps is achieved when the frame length is 65 byte. The complete dataset with frame lengths from 64 bytes to 1518 bytes can be accessed at [1].

Power consumption of two proposed architectures is illustrated in Fig. 7d. The power consumption is composed of static power consumption and dynamic power consumption. The static power consumption varies from 0.32 W to 0.48 W, and the dynamic power consumption increases linearly with the bus width. The power consumption of proposed SA has a faster growth than that of the proposed non-SA. This is because the resource consumption of the proposed SA increases faster than that of the proposed non-SA.

B. Comparison with More Works

[21] presents an implementation and analysis of encoder and checker of CRC8 which takes 16-bits of input data to create 24-bits code-word and can detect errors up to 8-bits. [22] introduce a hardware architecture for parallel CRC computation. It can be configured for a different polynomial at any time externally. [23] introduces a design and development of parallel CRC algorithm for the hardware implementation on FPGA to meet the specifications for FC. The algorithm can process 128-bit parallel data in a block by broken it into four 32-bit data and calculate their CRC, respectively, based on the LFSR. The aim of [24] is to design and implement the CRC5 and CRC8 systems that are used for USB token packet and ATM protocols, respectively. A parallel pipelining method is used to implement the proposed CRC architecture for both CRC encoder and decoder systems to achieve high throughput data with optimized hardware resources.

These works are suitable for embedded systems for their low resource consumption. However, their performance is also limited. Moreover, the padding zeros problem can't be solved. The comparison between the four works and the two proposed architectures can be founded in Table I.

C. Board-Level Implementation Result

We have verified our design on the VC709 FPGA development board. The system topology is shown in Fig. 8. The polynomial of Ethernet CRC32 is adopted. The bus width is 1024 bits. The clock frequency is 500 Mhz. The maximum throughput of the system is 512 Gbps. The final report generated by Vivado shows that the resource utilization of the CRC computing block in Fig. 8a is the same as the result in Fig. 7a. The reprogramming procedure will not be discussed in detail due to space limitations, and all the codes used here can be accessed at [1].

The correctness of the CRC computation and the system throughput should be verified. Sprient TestCenter N4U is used.
to generate 10 Gbps test frames with some error frames in them. The correctness can be verified if all the error frames can be checked out. The L2 frame rate is lower than 10 Gbps (L1 frame rate) for the 20 byte frame gap. The statistics of the bus occupancy ratio are gathered. The L2 frame rate and bus occupancy ratio are lower than 10 Gbps to generate 10 Gbps test frames with some error frames in them. The correctness can be verified if all the error frames can be checked out. The L2 frame rate is lower than 10 Gbps (L1 frame rate) for the 20 byte frame gap. The statistics of the bus occupancy ratio are gathered. The L2 frame rate and bus occupancy ratio can be used to compute the bus efficiency, and the throughput can be inferred using the bus efficiency.

The test result is shown in Table II. It can be found that the SA can achieve a higher throughput than the non-SA, especially for the 64-byte frames. This is because the SA can achieve a higher bus efficiency. If the short or misaligned frames have a large share in the network traffic, SA should be adopted for a guaranteed throughput. Otherwise, non-SA is a better choice for its lower resource utilization.

<table>
<thead>
<tr>
<th>Frame Length(Bytes)</th>
<th>Throughput (Gbps)</th>
<th>Power Consumption (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>12.2</td>
<td>240</td>
</tr>
<tr>
<td>1024</td>
<td>22.0</td>
<td>400</td>
</tr>
<tr>
<td>2048</td>
<td>31.8</td>
<td>560</td>
</tr>
<tr>
<td>4096</td>
<td>41.6</td>
<td>720</td>
</tr>
</tbody>
</table>

![Fig. 7: Synthesis result.](image)

### V. Conclusion and Future Work

Two algorithms and a method are proposed to realize low-cost, high performance, and programmable CRC computation. These algorithms and method can be used in segmented or non-segmented architectures. The synthesis results show that proposed architectures can achieve a lower resource utilization and higher throughput compared with two of the state-of-the-art architectures. The board-level implementation results show that the theoretical low resource utilization, high throughput, and programmability can be achieved. The source code can be accessed at [1]. Our future work will focus on making the hardware reconfiguration method (HWICAP) technology-independent.

### REFERENCES


