

# Pipelined and Partitionable Forward Error Correction and Cyclic Redundancy Check Circuitry Implementation for PCI Express® 6.0

Debendra Das Sharma, *Senior Member, IEEE*  
Intel Corporation  
debendra.das.sharma@intel.com

Swadesh Choudhary  
Intel Corporation  
swadesh.choudhary@intel.com

**Abstract**— PCI Express® (PCIe®) specification has been doubling the data rate every generation in a backward compatible manner every three years. PCIe 6.0 specification will adopt PAM-4 signaling at 64.0 GT/s for maintaining the same channel reach, cost, and power profile as prior generations. A Forward Error Correction (FEC) mechanism will offset the high BER of PAM-4. A strong Cyclic Redundancy Check (CRC) and a link level replay mechanism will deliver a low-latency, high bandwidth efficiency, and highly reliable solution expected of a Load-Store interconnect. We propose a non-pipelined implementation of the FEC and CRC that is part of the PCIe 6.0 base specification. We also propose a partitionable and pipelined implementation for FEC and CRC for lowering gate count and latency. We have tested the correctness of our register transfer logic (RTL) implementation in a field programmable gate array (FPGA) implementation in addition to simulation. Synthesis results from the Synopsys DC compiler demonstrates that for a x16 PCIe Link partitionable to up to x4s, with 4 independent controllers using independently partitionable logic, we achieve a gate count of about 100,000 for the transmit and receive side with a FEC + CRC delay of less than 1 nano second (nsec) in each direction.

**Index Terms**— PCI-Express, Reliability, Availability, FIT, Replay, Forward Error Correction, Cyclic Redundancy Check

## I. INTRODUCTION

THE Peripheral Component Interconnect Express (PCIe) architecture is the ubiquitous I/O across the wide spectrum of computing platforms. This is due to its ability to seamlessly deliver cost-effective, HVM (high volume manufacturing)-friendly, power-efficient, high bandwidth, and low latency solution with full backward compatibility through seven generations, doubling the data rate every generation in 2-3 years [1, 2, 5].

The sixth generation of PCIe (PCIe 6.0), adopted PAM-4 signaling to deliver 64.0 GT/s. With PAM-4 signaling, the bit error rate (BER) several orders of magnitude [1,2,4,8,9,10] over past generations as the eye height and width reduces significantly due to the presence of three eyes in each UI. However, as a load-store interconnect, PCIe poses stringent latency and bandwidth requirements. Additionally, PCIe PHY is used for alternate protocols such compute express Link® (CXL) [1,2,3,11] as well as proprietary CPU to CPU symmetric cache coherency links carrying coherency and memory traffic, where latency is critical to maintaining performance[13]. Thus,

the Forward Error Correction (FEC) latency hit exceeding 100ns to mitigate the high Bit Error Rate (BER) of PAM-4 of existing networking protocols [1,2,8,9,10,13,14] is unacceptable for the PCIe specification.

A Flow Control Unit (Flit)-based approach with light weight FEC, strong CRC and replay mechanism has been adopted in PCIe 6.0 specification to meet the stringent latency and bandwidth efficiency requirements [1,2,4,13]. Our proposed non-pipelined Verilog implementation of FEC and CRC has been adopted by the PCIe 6.0 specification [4] as the reference design. In this paper, we present pipelined microarchitectures for FEC and CRC for area and latency optimization to implement the Flit based mechanism proposed in [13]. With our pipelined approach, a x16 PCIe Link can be partitioned into smaller independent partitions if the PCIe link is partitioned into independent narrower-width Links. Our results demonstrate significant gate count savings with our proposed pipelined approach while achieving the low-latency targets.

## II. OVERVIEW OF PCIe 6.0 FLIT MODE

PCIe 6.0 defines a 256-Byte Flit as the unit of transfer. The Flit layout for an 8-Lane (x8) PCIe 6.0 Link is shown in Figure 1 [1,2,4]. The same Flit layout will be interleaved across Lanes for all PCIe Link widths: x1, x2, x4, x8, and x16 [1, 2, 4].

PCIe 6.0 defines First Bit Error Rate (FBER) as the probability of a bit error occurring at the Receiver pin, which can cause subsequent correlated errors, within the Lane due to error propagation in the decision feedback equalizer (DFE) circuit or across Lanes due to common noise events such as power supply noise [1, 2, 4]. FEC is used to alleviate errors.

The FEC is a single byte correct code. Three sets of FECs are interleaved across consecutive bytes in each Lane (Figure 1) [1,2,4]. Each color represents an FEC group, comprising 86B, 85B, and 85B and referred to as FEC groups 0, 1, and 2 respectively. Due to interleaving, one occurrence of an error burst in a Lane can be corrected if the burst length is  $\leq 16$  bits with no correlation errors across Lanes within each Flit. The choice of a single symbol correct FEC is to minimize the encode and decode time to be less than 2 nsec after Flit accumulation for latency critical applications. The H-matrix for each FEC group [1,2,4], using a primitive polynomial over GF(2<sup>8</sup>), is

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & \dots & 1 & 0 & 1 \\ \alpha^{84} & \alpha^{83} & \dots & \alpha & 1 & 0 \end{bmatrix}$$

x8 Lanes	0	1	2	3	4	5	6	7
256 UI								
TLP Bytes	0	1	2	3	4	5	6	7
(0-299)	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63
	64	65	66	67	68	69	70	71
	72	73	74	75	76	77	78	79
	80	81	82	83	84	85	86	87
	88	89	90	91	92	93	94	95
	96	97	98	99	100	101	102	103
	104	105	106	107	108	109	110	111
	112	113	114	115	116	117	118	119
	120	121	122	123	124	125	126	127
	128	129	130	131	132	133	134	135
	136	137	138	139	140	141	142	143
	144	145	146	147	148	149	150	151
	152	153	154	155	156	157	158	159
	160	161	162	163	164	165	166	167
	168	169	170	171	172	173	174	175
	176	177	178	179	180	181	182	183
	184	185	186	187	188	189	190	191
	192	193	194	195	196	197	198	199
	200	201	202	203	204	205	206	207
	208	209	210	211	212	213	214	215
	216	217	218	219	220	221	222	223
	224	225	226	227	228	229	230	231
	232	233	234	235	dlp0	dlp1	dlp2	dlp3
	dlp4	dlp5	crc0	crc1	crc2	crc3	crc4	crc5
	crc6	crc7	ecc0	ecc0	ecc0	ecc1	ecc1	ecc1

Figure 1: 256B Flit in a x8 Link: 236B TLP, 6B DLP, 8B CRC, 6B FEC: the DLPs occupy Bytes 236 through 241, the CRC occupies Bytes 242 through 249, and ECC Bytes are in Bytes 250 through 255. FEC group 0 (blue color) has 86B whereas FEC groups 1 and 2 (orange and green color respectively) have 85B each

The Flit CRC is guaranteed to detect up to 8 Symbols in error (post-FEC), where each Symbol is a byte. The CRC generator polynomial is given below.  $\alpha$  is the root of a different (than FEC) primitive polynomial over  $GF(2^8)$  [1, 2, 4].

$$g(x) = (x + \alpha)(x + \alpha^2) \dots (x + \alpha^8) \quad (2)$$

On the transmit side, the Flit CRC is computed after which the FEC bytes are generated. The Transaction Layer Packet (TLP) part of a Flit is stored in a replay buffer which is removed when the transmitter receives an 'Ack' in the DLP for that Flit or is replayed if it gets a 'Nak'. On receipt, the FEC decode is applied first followed by the CRC check. On a failure, a replay is requested. A representative micro-architecture of these components is shown in Figure 2.

### III. FEC IMPLEMENTATION

As described in Section II, PCIe 6.0 uses 84, 83 and 83 bytes of message symbols in the three FEC groups. A byte value of 00h is post-fixed (MSB) to the messages of FEC Groups 1 and 2 to make three sets of identical 84 byte messages for a 84 to 86 byte encoder that can be reused. The check (C) and parity (P) computations are given by the following formulae:

$$P = \sum_{i=0}^{83} B_i \quad (3)$$

$$C = \sum_{i=0}^{83} B_i \cdot \alpha^{(84-i)} = \sum_{i=0}^{83} \sum_{j=0}^7 8\{B_{ij}\} \cdot \alpha^{(84-i+j)} \quad (4)$$

where  $B_i$  is the  $i$ th Byte of each FEC group,  $B_{ij}$  is the  $j$ th bit

of byte  $B_i$  ( $0 \leq j \leq 7$ ), operating over  $GF(2^8)$  [12].

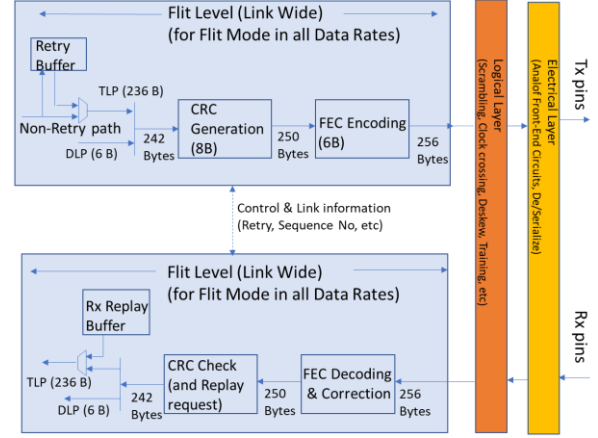


Figure 2: Micro-architecture of PCIe 6.0 with FEC, CRC, and Replay

#### A. Non-pipelined Implementation of FEC

The non-pipelined RTL implementation of FEC encoder is based on the two equations above [4]. The FEC decoder is shown in Figure 3. For each FEC group, the 86 Bytes are derived from the 256B Flit, with the bytes in increasing order, as follows: FEC Group 0 (blue):  $\{B_0, B_3, B_6, \dots, B_{255}\}$ , FEC Group 1 (orange):  $\{B_1, B_4, \dots, B_{247}, 8'h00, B_{250}, B_{253}\}$  and FEC Group 2 (green):  $\{B_2, B_5, \dots, B_{248}, 8'h00, B_{251}, B_{254}\}$ . Thus, for FEC Group 2: B0 is B2 of the Flit, B1, ..., C is B251 of the Flit, and P is B254 of the Flit.

The decoder comprises two phases: the syndrome generation part (for the 8 check bits and 8 parity bits, using the same logic as the FEC encoder, syndromes are generated by XORing the received bytes with the computed bytes) and the correction / detection part, if syndrome is non-zero. The latter comprises a look-up table for each of the two 8-bit syndromes. The following action is taken based on the value of the two syndromes:  $\{Synd\_Check, Synd\_Parity\}$ :

- $\{00h, 00h\}$ : No error
- $\{00h, non-zero\}$ : Error in  $B_{85}$ : Corrected  $B_{85} = B_{85} \wedge Synd\_Parity$
- $\{non-zero, 00h\}$ : Error in  $B_{84}$ : Corrected  $B_{84} = B_{84} \wedge Synd\_Check$
- $\{non-zero, non-zero\}$ : Follow the log table for  $Synd\_Check$ ,  $Synd\_Parity$  and then follow the modulo subtraction +1 logic to identify the failing column number. If the column number is  $<0$  or  $>85$ , it is an uncorrectable error. Once the column number is known, we can correct the Symbol by XORing with the  $Synd\_Parity$

In the sample RTL code we provided for the specification [4], and as represented by Figure 3, the reverse look-up that is performed to get the alpha powers, and subsequent derivation of the error location turned out to have more gates and logic levels than the following approach. They are equivalent in functionality, demonstrated with formal equivalence check.

We use the intermediate result of the encoder that is present in the decoder. For a single symbol error in symbol  $k$  of magnitude  $e_k$ , the parity and check syndromes are:

$$SyndParity = e_k \text{ and } SyndCheck = \alpha^{(84-k)} e_k$$

Thus, for each of the received data byte, if we XOR the byte with SyndParity (with a starting value of 0), we obtain the error magnitude  $e_k$ . We also multiply  $e_k$  with the corresponding alpha powers of each byte position (0 through 83) using the encoder logic. The only match with the received SyndCheck will be the byte location in error. Due to XORing, we basically get rid of the information bits and are only left with the error term. This approach results in a lower gate count and lower levels of logic than a flat reverse look-up table.

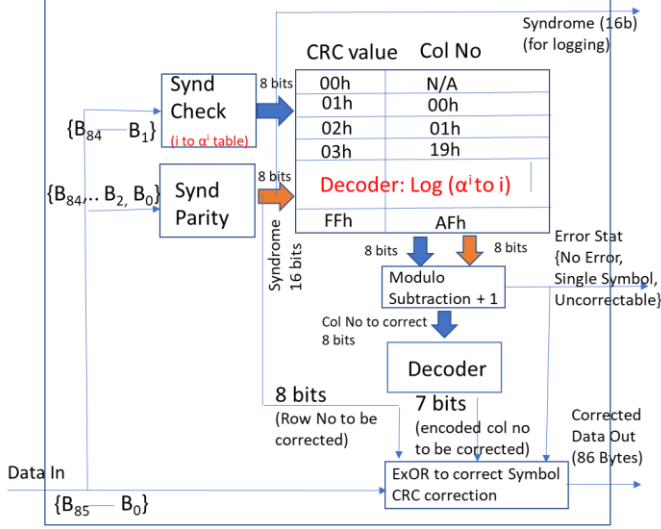


Figure 3: The FEC Decoder comprises of two stages: the syndrome generation part (16 bits) and the correction part if needed

### B. Implementation of link sub-division

A set of PCIe Lanes can be sub-divided into multiple independent Links in the downstream direction (Figure 4). This is commercially prevalent in CPUs and switches. In those cases, it is highly desirable to partition the logic into independent partitions (i.e., a x8 or a x4 gets half or quarter the logic of a x16). This approach results in a lower gate count than the widespread approach of replicating the same logic for the maximum number of the Links possible. In our approach, the x4 is the basic building block used for constructing a x16, two x8s, four x4s, or a x8 plus two x4s. We propose to extend the same partitioning concept to FEC and CRC. In order to support this feature, we have implemented a pipelined version of x4 that can be used as a basic building block for x8 and x16.

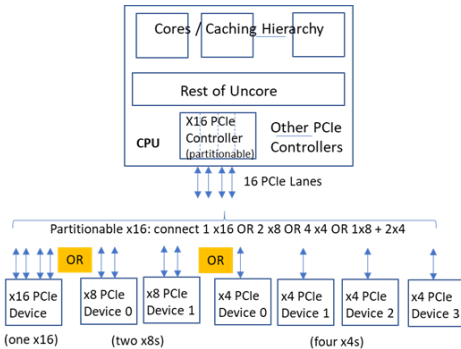


Figure 4: A x16 Link from a CPU can connect to either one x16 device or multiple smaller widths independent PCIe devices

### C. Pipelined FEC Implementation

Pipelining the parity computation is trivial by computing the intermediate parity of the received bytes and XORing with the stored result (of all the previous bytes if any) and storing them for subsequent cycle, if any.

We use the following property of GF arithmetic to pipeline the check computation of check bits efficiently:

$$\alpha^{(x+y)} \bmod p(x) = (\alpha^x (\alpha^y \bmod p(x))) \bmod p(x) \quad (5)$$

where  $p(x)$  is the primitive polynomial over  $GF(2^8)$  whose root  $\alpha$  is used in the H-matrix of equation 1.

#### Proof Outline for property (5)

$$\text{Let } \alpha^y = a(x) * p(x) + r(x) \quad (6)$$

Where  $a(x)$  is chosen such that degree of  $r(x) < \text{degree of } p(x)$  and:

$$\alpha^y \bmod p(x) = r(x) \quad (7)$$

$$\text{Let } \alpha^x = b(x) * p(x) + s(x) \quad (8)$$

$$\alpha^{(x+y)} \bmod p(x) = (\alpha^x \alpha^y) \bmod p(x) \quad (9)$$

$$\alpha^{(x+y)} \bmod p(x) = (a(x) * p(x) + r(x)) (b(x) * p(x) + s(x)) \bmod p(x) \quad (10)$$

$$\alpha^{(x+y)} \bmod p(x) = (r(x) * s(x)) \bmod p(x) \quad (11)$$

Similarly,

$$(\alpha^x (\alpha^y \bmod p(x))) \bmod p(x) = ((b(x) * p(x) + s(x)) * r(x)) \bmod p(x) \quad (12)$$

$$(\alpha^x (\alpha^y \bmod p(x))) \bmod p(x) = (s(x) * r(x)) \bmod p(x) \quad (13)$$

$$(\alpha^x (\alpha^y \bmod p(x))) \bmod p(x) = (r(x) * s(x)) \bmod p(x) \quad (14)$$

The property in (5) implies that we can accumulate the alpha powers over bytes and stick to only manipulating 8-bit results and accumulating them with newer bytes that show up in subsequent cycles i.e.,

$$B_i * \alpha^{x+y} + B_j * \alpha^{z+y} = (B_i * \alpha^x + B_j * \alpha^z) * \alpha^y \quad (15)$$

### D. Pipelined FEC Implementation with partitioning

For a x16 controller running at 1GHz (128B data path), we can either receive 42 or 43 bytes of data in the first cycle, and the remaining 42 or 41 bytes of data in the next cycle per FEC group, as demonstrated in Table 1. For a x8 controller running at 1GHz (64B data path), data comes over four clock cycles at 20 to 22 bytes per cycle per FEC group. For a x4 controller running at 1GHz (32B data path), data comes over eight clock cycles at 9 to 11 bytes per cycle per FEC group.

Table 1: Data Bytes to each FEC Group Encoder in pipelined implementation, including the forced 00h in Groups 1 and 2 at 1 GHz

Width	Cycle	FEC Group		
		0	1	2
X16	0	B [42:0]	B [42:0]	B [41:0]
	1	B[83:43]	00h,B[82:43]	00h,B[82:42]
X8	0	B[21:0]	B[20:0]	B[20:0]
	1	B[42:22]	B[42:21]	B[41:21]
	2	B[63:43]	B[63:43]	B[63:42]
	3	B[83:64]	00h,B[82:64]	00h,B[82:64]
X4	0	B[10:0]	B[10:0]	B[9:0]
	1	B[21:11]	B[20:11]	B[20:10]
	2	B[31:22]	B[31:21]	B[31:21]
	3	B[42:32]	B[42:32]	B[41:32]
	4	B[53:43]	B[52:43]	B[52:42]
	5	B[63:54]	B[63:53]	B[63:53]
	6	B[74:64]	B[74:64]	B[73:64]

	7	B[83:75]	00h,B[82:75]	00h,B[82:74]
--	---	----------	--------------	--------------

For the decoder, the parity part will be pipelined since we do not incur any additional latency as we calculate partial results as the Flit data is received. Once the synd\_parity ( $e_k$ ) is available; for each byte position, synd\_parity is bitwise AND'ed with the corresponding alpha power and subsequently bitwise XOR'ed with the received check byte. If the result of this XOR is 8'h00, the corresponding byte is in error, and the original received byte in that position is XOR'ed with the synd\_parity to generate the corrected byte.

### 1) x4 encoder

Figure 5 shows the basic building block (template) used to implement the check computation of the encoder. Each FEC group has an instance of this logic for a x4 port.

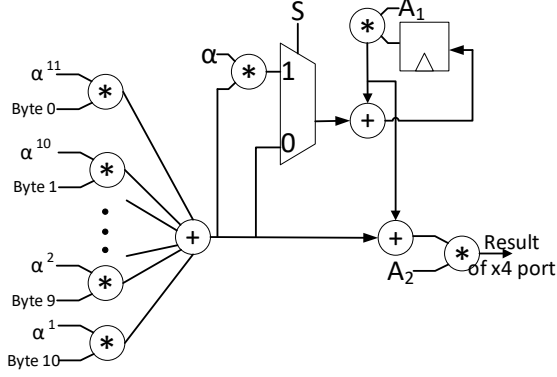


Figure 5: Single x4 port logic for pipelined FEC encoder

The check result is accumulated every cycle and is valid on the 8<sup>th</sup> cycle. The mux select S is always set to 0 for x4 (the extra multiplier and mux is there for x8 width). Parameter  $A_2$  is  $\alpha^{253}$  for FEC Group 0 and FEC Group 1, and  $\alpha^{254}$  for FEC Group 2. The 8-bit intermediate result in the flip-flops in Figure 5 is reset to 00h on clock cycle 0. Parameter  $A_1$  is a function of the FEC group and cycle number, as shown in Table 2.

Table 2: Values of parameter  $A_1$  for different FEC groups and cycles

Cycle No.	FEC Group 0	FEC Group 1	FEC Group 2
0	n/a	n/a	n/a
1	$\alpha^{11}$	$\alpha^{11}$	$\alpha^{10}$
2	$\alpha^{11}$	$\alpha^{10}$	$\alpha^{11}$
3	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{11}$
4	$\alpha^{11}$	$\alpha^{11}$	$\alpha^{10}$
5	$\alpha^{11}$	$\alpha^{10}$	$\alpha^{11}$
6	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{11}$
7	$\alpha^{11}$	$\alpha^{11}$	$\alpha^{10}$

### 2) x8 encoder using x4 templates

A x8 port's encoder can be constructed using two instances of the x4 template of Figure 5, as shown in Figure 6. The check result of the x8 port is accumulated and is valid on Cycle 3.

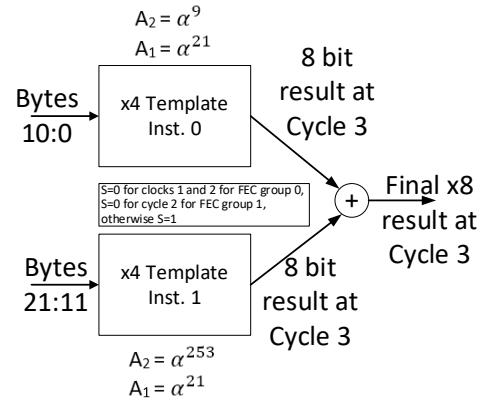


Figure 6: constructing x8 encoder using x4 template

### 3) x16 encoder from x4 templates

A x16 encoder (single port) can be constructed using four instances of the x4 template as shown in Figure 7. The check result of the x16 port is accumulated and is valid on cycle 1.. Table 3 shows the value of parameter  $A_2$  for the different instances and modes.

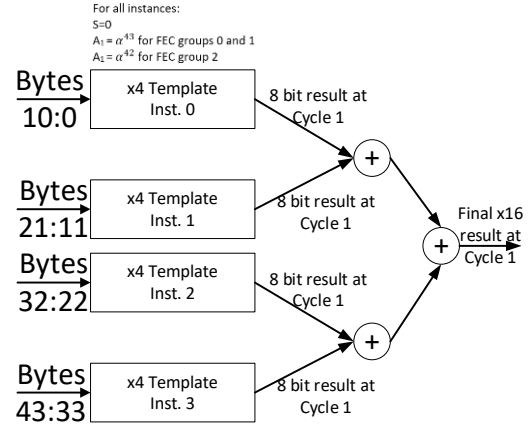


Figure 7 Constructing x16 encoder using x4 template

Table 3: Values of parameter  $A_2$  for different instances/ FEC Groups

Instance No.	FEC Group 0/1	FEC Group 2
0	$\alpha^{30}$	$\alpha^{31}$
1	$\alpha^{19}$	$\alpha^{20}$
2	$\alpha^8$	$\alpha^9$
3	$\alpha^{252}$	$\alpha^{253}$

## IV. CRC IMPLEMENTATION

PCIe 6.0 CRC uses the Reed Solomon (RS) code [12] using the generator polynomial [1, 2, 4]:

$$g(x) = (x + \alpha)(x + \alpha^2) \dots (x + \alpha^8) = x^8 + \alpha^{172}x^7 + \alpha^{116}x^6 + \alpha^{186}x^5 + \alpha^{172}x^4 + \alpha^{195}x^3 + \alpha^{134}x^2 + \alpha^{199}x + \alpha^{36} \quad (16)$$

Let  $M(x)$  be the polynomial representation of the 242 Byte message, including the first 236 Bytes of TLP followed by 8 Bytes of Data Link Layer Payload (DLP), as shown in Figure 1, represented as  $m_i$  below. Thus, Byte 0 of TLP which goes out first is represented as  $m_0$ , Byte  $i$  as  $m_i$ , and dlp5 is  $m_{241}$ .

$$M(x) = m_0x^{241} + m_1x^{240} + \dots + m_{240}x + m_{241} \quad (17)$$

The CRC symbols for the (250,242) code is given by the co-efficients of the polynomial:

$$c(x) = M(x)x^8 \bmod g(x) \quad (18)$$

The rest of this sub-section describes our proposed non-pipelined implementation followed by the pipelined implementation.

The generator matrix for the CRC code can be expressed as:

$$G = \begin{bmatrix} 1 & \dots & 0 & b_{07} & \dots & b_{00} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & b_{(241)7} & \dots & b_{(241)0} \end{bmatrix} \quad (19)$$

Or,  $G = [I | R]$  (20)

where,  $I$  is a 242x242 identity matrix, and the elements of  $i^{\text{th}}$  row ( $b_{i7}, b_{i6}, \dots, b_{i1}, b_{i0}$ ), corresponding to  $m_i$ , of  $R$  are given by the co-efficients of the polynomial  $r_i(x)$  ( $0 \leq i \leq 241$ ):

$$\sum_{j=0}^7 b_{ij}x^j = x^{249-i} \bmod g(x) \quad (21)$$

Thus, row 241 is  $x^8 \bmod g(x)$ , which is  $\{\alpha^{172}, \alpha^{116}, \alpha^{186}, \alpha^{172}, \alpha^{195}, \alpha^{134}, \alpha^{199}, \alpha^{36}\}$  from equation 16. Similarly, row 240 will be derived as:

$$x^9 \bmod g(x) = (x \cdot (x^8 \bmod g(x))) \bmod g(x) = (\alpha^{172} * \alpha^{172} + \alpha^{116}x^7 + (\alpha^{172} * \alpha^{116} + \alpha^{186})x^6 + \dots + (\alpha^{172} * \alpha^{199} + \alpha^{36})x + (\alpha^{172} * \alpha^{36})) \quad (22)$$

Other rows can be derived using the approach above as follows ( $0 \leq i \leq 240$ ) (note that  $b_{i,j}$  is same as  $b_{ij}$ ):

$$b_{i,j} = \begin{cases} b_{(i+1),(j-1)} + b_{(i+1),7} * b_{241,j} & \text{if } (1 \leq j < 7) \\ b_{(i+1),7} * b_{241,j} & \text{if } (j = 0) \end{cases} \quad (23)$$

The CRC symbols can be generated as follows:

$$[c_7 \ c_6 \ \dots \ c_1 \ c_0] = [m_0 \ m_1 \ \dots \ m_{240} \ m_{241}] * R \quad (24)$$

which can be rewritten as:

$$c_j = \sum_{i=0}^{241} m_i * b_{(i+1)j} \quad (0 \leq j \leq 7) \quad (25)$$

For non-pipelined version of CRC we implemented equation (25), which appears in PCIe 6.0 specification [4].

#### A. Our Approach for Pipelining CRC computation

For our pipelined implementation, we exploit some mathematical properties of the polynomial division to significantly reduce the gate count and latency. As data bytes stream in over multiple clock cycles, we compute intermediate results using a subset of the CRC computation, followed by a translation and accumulation operation (which also map to a subset of the CRC computations) on the intermediate results.

We illustrate the idea with an example followed by the general mathematical construct. In this example, we use the second message symbol (byte 2 used for TLP of the 256B Flit) to illustrate how an early arriving byte is transformed in subsequent cycles by leveraging the same logic used for bytes that will arrive later in an effort to minimize gate count.

**Example:** We refer to the second message symbol as  $m_2$ . We will refer to contribution of  $m_2$  to the CRC bytes as  $c_{m_2}(x)$ , which can be computed using (24) by assigning  $m_i = 0$ , for all ( $i \neq 2$ ), which gives (26) when represented in polynomial form.

$$c_{m_2}(x) = \sum_{j=0}^7 m_2 * b_{2j}x^j \quad (26)$$

An alternate way to compute  $c_{m_2}(x)$  is by setting  $i=2$  in (21).

Since  $247 = 239 + 8$ , we get the following:

$$m_2 x^{239+8} \bmod g(x) = \sum_{j=0}^7 m_2 * b_{2j}x^j \quad (27)$$

Where  $g(x)$  is the generator polynomial from (16).

Similarly, this same can be written with  $m_3$ , such that

$$m_3 x^{238+8} = q_{238}(x)g(x) + (\sum_{j=0}^7 m_3 * b_{3j}x^j) \quad (28)$$

For some non-zero polynomial  $q_{238}(x)$ . Since (28) is true for any value of  $m_3$ , then it is also true if we replace it with  $m_2$ . Multiply by  $x$  on both sides and apply mod  $g(x)$ .

$$m_2(x^{239+8}) \bmod g(x) = (\sum_{j=0}^7 m_2 * b_{3j}x^{j+1}) \bmod g(x) \quad (29)$$

But from (21) with  $i=241$ ,

$$x^8 \bmod g(x) = \sum_{j=0}^7 b_{241j}x^j. \quad (30)$$

Thus,

$$\sum_{j=0}^7 m_2 * b_{2j}x^j = \sum_{s=0}^7 m_2 * b_{37} * b_{241s}x^s + \sum_{j=0}^6 m_2 * b_{3j}x^{j+1} \quad (31)$$

$$\sum_{j=0}^7 m_2 * b_{2j}x^j = (\sum_{j=1}^7 ((m_2 * b_{37}) * b_{241j} + (m_2 * b_{3(j-1)}))x^j) + m_2 * b_{37} * b_{2410} \quad (32)$$

Based on the above example, we can take the message symbol  $m_2$ , operate it with row 3 of the generator matrix to get  $m_2 * b_{3j}$ . After that, we can take the most significant byte of that ( $m_2 * b_{37}$ ) and operate that on row 241 to obtain an intermediate result. This intermediate result is added to single byte left shifted value of the remaining bytes i.e.  $\{m_2 * b_{36}, \dots, m_2 * b_{30}, 00h\}$  to get the final result which is equivalent to operating message symbol  $m_2$  on row 2.

We can extend the example for  $m_2$  for any message symbol  $m_k$  as follows: We can apply recursion to (23) and express the computation over row  $i$  as a combination of computations using subset of rows  $> i$ . An algebraic approach to reaching the same conclusion is outlined below.

Let ( $b_{i7} \ b_{i6} \ \dots \ b_{i1} \ b_{i0}$ ) be the co-efficients of the polynomial  $r_i(x)$ . From (21), we have:

$$\sum_{j=0}^7 m_k * b_{ij}x^j = m_k * x^{249-i} \bmod g(x) \quad (33)$$

$$\sum_{j=0}^7 m_k * b_{ij}x^j = (m_k * x^{249-(i+a)} * x^a) \bmod g(x) \quad (34)$$

Since  $x^{249-(i+a)} = q_{i+a}(x)g(x) + r_{i+a}(x)$

$$\sum_{j=0}^7 m_k * b_{ij}x^j = (m_k * (q_{i+a}(x)g(x) + r_{i+a}(x)) * x^a) \bmod g(x) \quad (35)$$

Since  $(X g(x)) \bmod g(x) = 0$  for any finite  $X$ ,

$$\sum_{j=0}^7 m_k * b_{ij}x^j = (m_k * r_{i+a}(x) * x^a) \bmod g(x) \quad (36)$$

Substituting  $r_{i+a}(x)$  in polynomial form:

$$\sum_{j=0}^7 m_k * b_{ij}x^j = (\sum_{j=0}^7 (m_k * b_{(i+a)j})x^{j+a}) \bmod g(x) \quad (37)$$

Since from (21),  $x^{j+a} \bmod g(x) = \sum_{s=0}^7 b_{(249-j-a)s}x^s$

$$\sum_{j=0}^7 m_k * b_{ij}x^j = \sum_{j=0}^7 ((m_k * b_{(i+a)j}) * \sum_{s=0}^7 b_{(249-j-a)s}x^s) \quad (38)$$

#### B. CRC Pipelining implementation with partitioning

As with FEC, for our link subdivision capability, we use the pipelined x4 CRC as the basic building block (template) that is used to construct the x8 and x16 Link CRC. First, we will discuss the pipeline for each width independently followed by the common x4 template that can be used to construct the partitionable design.

For the CRC, we implemented a 2GHz design since our initial analysis showed that the number of stages in CRC can easily meet the 2 GHz timing. Running the CRC at a higher frequency has two benefits: lower gate count and lower latency as the results are ready to feed to the FEC logic. The decoder is identical to the pipelined encoder for the part where CRC is generated from incoming message bytes. Decoding has the



additional step of comparing the computed CRC bytes with the received CRC bytes. If there is a mismatch, CRC detected an error. This compare logic is minimal. Hence the CRC decoder is almost identical to the CRC encoder, unlike the difference between FEC encoder vs decoder.

#### 1) x4 single port CRC

For a x4 port, data arrives over 16 cycles, with 16 bytes per cycle except the last cycle with 2 bytes, as shown in Table 4 and demonstrated in Figure 8. The principle behind the pipelining is intuitively the same as what was derived in equation (5). Instead of multiplexing the entire G matrix with the corresponding message byte that comes in, every set of 16 Bytes is multiplied with the last 16 rows (226 through 241) in the G-matrix which is effectively multiplying the message with  $x^8$  through  $x^{23}$ , depending on the byte position, and then multiplying the partial CRC in the following cycles (represented by  $C_{n-1}$ ) to adjust for the corresponding power of  $x$  for the byte. The only special case here is in cycle 15 since we receive two bytes (instead of 16 bytes). Hence the two bytes of incoming message are multiplied by  $x^8$  and  $x^9$  which involves multiplying the message with rows 240 and 241 of the G-matrix. The partial CRC in this cycle also needs to be multiplied by  $x^2$  since only 2 new byte offset needs to be adjusted. Hence only the partial CRC bytes 7 and 6 are multiplied with the last two rows of the H-matrix and the remaining bytes are left shifted by 2 bytes, representing the multiplication by  $x^2$ .

Table 4: Pipelined CRC computation for a x4 Link

Cycle n	Input Data	$C_n(x)$
0	m [0:15]	$\sum_{i=0}^{15} m_i x^{23-i} \text{modg}(x)$ [= $\sum_{i=0}^{15} m_i * b_{(i+226)j}$ ], ( $0 \leq j \leq 7$ )]
1	m [16:31]	$\sum_{i=0}^{31} m_i x^{39-i} \text{modg}(x)$ [= $\sum_{i=16}^{31} m_i * b_{(i-16+226)j}$ ] + $\{\sum_{i=0}^7 C_{[n-1,7-i]} * b_{(i+226)j}\}$ ( $0 \leq j \leq 7$ )]
2 .. 14	m [16n:16n+15]	$\sum_{i=0}^{16n+15} m_i x^{16n+23-i} \text{modg}(x)$ [= $\sum_{i=16n}^{16n+15} m_i * b_{(i-16n+226)j}$ ] + $\{\sum_{i=0}^7 C_{[n-1,7-i]} * b_{(i+226)j}\}$ ( $0 \leq j \leq 7$ )]
15	m [240:241]	$\sum_{i=0}^{241} m_i x^{249-i} \text{modg}(x)$ [= $\sum_{i=240}^{241} m_i * b_{ij}$ ] + $\{\sum_{i=6}^7 C_{[n-1,i]} * b_{(240-i+7)j}\}$ ( $0 \leq j \leq 7$ ) + $\{C_{[n-1,j]}, 00h, 00h\}$ ( $0 \leq j \leq 5$ )]

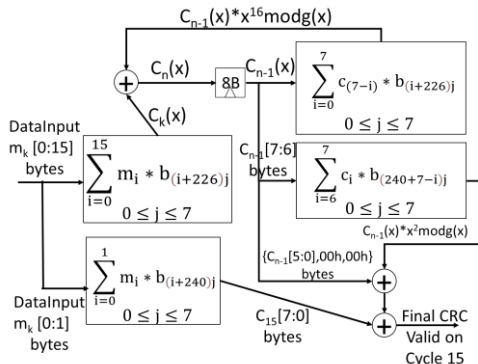


Figure 8: Pipelined CRC encoder implementation of a single x4

#### 2) x8 single port CRC

For a x8 single port, data arrives over 8 cycles, with 32 bytes per cycle, except the last cycle with 18 bytes. The pipelining is similar to x4 width and the same x4 logic is mostly reused. This is critical to building the common building block template with the least additional gate count while having a higher priority in reducing the levels of logic. Figure 9 shows a block diagram of a pipelined x8 logic for CRC computation. The final result is available on the 8<sup>th</sup> cycle (clock cycle 7).

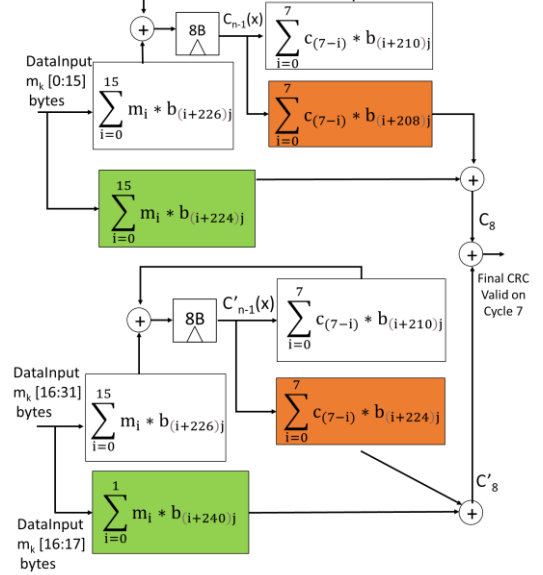


Figure 9: Pipelined CRC encoder implementation of a single x8

Table 5: Pipelined CRC computation for a x8 Link

Cycle (n)	Input Data	$C_n(x)$ (two independent sets: C and C' for odd and even 16B halves)
0	m [0:15]	$\sum_{i=0}^{15} m_i x^{23-i} \text{modg}(x)$ [= $\sum_{i=0}^{15} m_i * b_{(i+226)j}$ ], ( $0 \leq j \leq 7$ )]
	m [16:31]	$\sum_{i=16}^{31} m_i x^{39-i} \text{modg}(x)$ [= $\sum_{i=16}^{31} m_i * b_{(i-16+226)j}$ ], ( $0 \leq j \leq 7$ )]
1..6	m [32n:32n+15]	$\sum_{i=32n}^{32n+15} m_i x^{32n+23-i} \text{modg}(x) + C_{n-1}(x) \cdot x^{32} \text{modg}(x)$ [= $\sum_{i=32n}^{32n+15} m_i * b_{(i-32n+226)j}$ ] + $\{\sum_{i=0}^7 C_{[n-1,7-i]} * b_{(i+210)j}\}$ , ( $0 \leq j \leq 7$ )]
	m [32n+16:32n+31]	$\sum_{i=32n+16}^{32n+31} m_i x^{32n+39-i} \text{modg}(x) + C'_{n-1}(x) \cdot x^{32} \text{modg}(x)$ [= $\sum_{i=32n+16}^{32n+31} m_i * b_{(i-32n-16+226)j}$ ] + $\{\sum_{i=0}^7 C'_{[n-1,7-i]} * b_{(i+210)j}\}$ , ( $0 \leq j \leq 7$ )]
7	m [224:239]	$C_8 = \sum_{i=224}^{239} m_i x^{249-i} \text{modg}(x) + C_{n-1}(x) \cdot x^{34} \text{modg}(x)$ [= $\sum_{i=224}^{239} m_i * b_{ij}$ ] + $\sum_{i=0}^7 C_{[n-1,7-i]} * b_{(i+208)j}$ ( $0 \leq j \leq 7$ )]
	m [240:241]	$C'_8 = \sum_{i=240}^{241} m_i x^{249-i} \text{modg}(x) + C'_{n-1}(x) \cdot x^{18} \text{modg}(x)$ [= $\sum_{i=240}^{241} m_i * b_{ij}$ ] + $\sum_{i=0}^7 C'_{[n-1,7-i]} * b_{(i+224)j}$ ( $0 \leq j \leq 7$ )]
		<b>CRC = <math>C_8 + C'_8</math></b>

The even 16 Bytes (e.g., bytes 0..15, 32..47, etc) are multiplied the same way as the odd 16 bytes in the first stage in cycles 0-6, even though the even 16 bytes should have been multiplied by an additional  $x^{16}$ . That is done to leverage the same logic in that stage as a  $x^4$ . We multiply the stored partial CRC ( $C_{n-1}$ ) by  $x^{32}$  as opposed to  $x^{16}$  for  $x^4$ . In the last stage, we multiply according to the required powers at the incoming bytes (green boxes) and the pipelined CRC is also multiplied by the appropriate power of  $x$  (orange boxes), as shown above. The rationale is it takes less logic to manipulate 8 bytes of CRC vs having different logic for 16 bytes. In the case of even 16 bytes instead of creating a separate logic to multiply  $x^{34}$ , we could have connected the input to the partial CRC ( $C_{n-1}$ ), which is a multiplication of  $x^{32}$ , with the  $x^2$  multiplier logic in the last stage of the  $x^4$  CRC. That would have saved us gates but at the expense of increased the levels of logic. For cases such as these, we prioritize reduced number of levels over reduced gate count since performance is critical for the Load-store applications.

### 3) $x16$ single port CRC

For a  $x16$  port, we construct the pipelined implementation using four  $x4$  building blocks to handle 64 bytes of data every clock cycle. Thus, the message bytes are received over four cycles, 64 bytes in the first three cycles and 50 bytes on the fourth cycle. Figure 10 shows the building blocks, there will be 4 instances of the “Per 16 byte port” logic, which handle the first 3 cycles and 1 instance of the “Common” logic which handles the 4<sup>th</sup> cycle. In the “Per 16 byte port” logic, differences between the different instances are denoted by “Port X : \*”.

The final CRC is obtained through an XOR of the  $x16$  component of CRC from each instance of the “Per 16 byte port” logic as well as the “common” logic on clock cycle 3 (this XOR is not shown in the figure for compactness).

Table 6: Pipelined CRC computation for a  $x16$  Link

Cycle (n)	Input Data	$C_p^n(x)$ (4 independent sets): p = Port No. 0, 1, 2, 3
0	[16p:16p+15]	$\sum_{i=0}^{15} m_{16p+i} x^{23-i} \text{modg}(x)$ $[=\{\sum_{i=0}^{15} m_{16p+i} * b_{(i+226)}\}, (0 \leq j \leq 7)]$
1,2	64n+16p: 64n+16p+15	$\sum_{i=64n+16p}^{64n+16p+15} m_i x^{64n+16p+23-i} \text{modg}(x) + C_{n-1}^p(x) * x^{64} \text{modg}(x)$ $[=\{\sum_{i=0}^{15} m_{64n+16p+i} * b_{(i+226)}\} + \sum_{i=0}^7 C_{[n-1,7-i]}^p * b_{(i+178)}] (0 \leq j \leq 7)]$
3	192:241	$\sum_{i=192}^{241} m_i x^{249-i} \text{modg}(x)$ $C_3 = \sum_{i=192}^{241} m_i * b_{ij} (0 \leq j \leq 7)$
		<b>CRC</b> = $C_3 + \sum_{i=0}^7 [C_2^0(x) * b_{(144+i,j)} + C_2^1(x) * b_{(160+i,j)} + C_2^2(x) * b_{(176+i,j)} + [C_2^3(x) * b_{(192+i,j)}] (0 \leq j \leq 7)$

### 4) Implementation of $x16$ CRC with link sub-division for up to 4 Ports

As described above, we use the  $x4$  building blocks to create a pipelined CRC encoder for a  $x16$  port that supports link sub-division combinations of  $x8$  and  $x4$ .

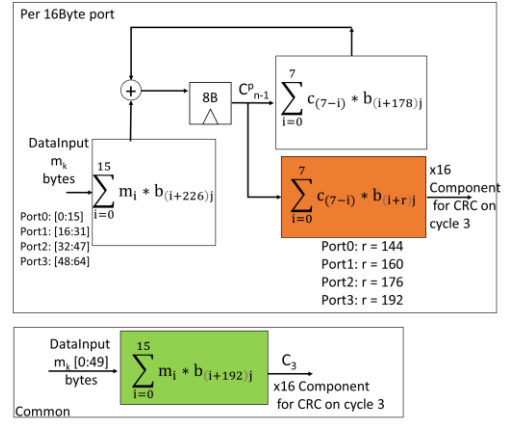


Figure 10: Building blocks of a pipelined  $x16$  CRC implementation

Similar to the  $x16$  single port example above, the building blocks are implemented as a combination of “Per Port” logic that operates on 16 bytes of input data (shown in Figure 11), and sets of “Common” logic (green boxes from the  $x8$ , and  $x16$  sections above) – 2 sets for supporting two  $x8$  ports, and 1 set for supporting  $x16$ , with appropriate muxes and accumulation of intermediate results. Thus, Port 0 would enable the  $x4$ ,  $x8$ , and  $x16$  paths, Ports 1 and 3 would only instantiate the  $x4$  path, and Port 2 would instantiate the  $x4$  and  $x8$  paths, through the DC compiler directives on the same RTL code.

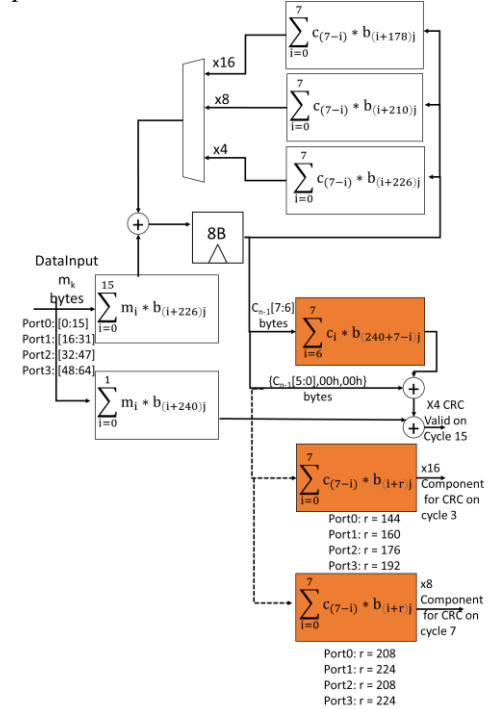


Figure 11: Building block logic per port for link subdivision

## V. RESULTS

### A. Experimental Set-up for validating our implementation

We have implemented the FEC and CRC schemes in C (for the non-pipelined version) as well as in System Verilog (non-pipelined as well as pipelined versions). For the System Verilog implementation, the logic was tested using Chronologic’s Verilog compiler and simulator (VCS). We connect the transmit and receive side back-to-back with an error injection logic in

between, mimicking the errors that the FEC can correct and the CRC can detect. While testing the FEC functionality, we inject single symbol errors in one, two, and three FEC groups exhaustively and check the corrected data against the transmitted data along with a match of the expected syndrome. Thus, for each FEC group, each of the 85 (or 86) symbols will be injected errors of magnitudes 0 through 255 (i.e., every one of the 256 permutations in a Symbol). For testing the CRC, we have a lot of permutations that precludes exhaustive testing. So we deployed two mechanisms: exhaustive testing for up to 4 symbols in errors and pseudo-random testing for more than 5 symbols in error. For errors injected in 8 Symbols or less, we check that the 'Flit Retry' at the output of CRC check is set to 1b if an uncorrectable error was injected.

We also performed formal equivalence checking between the pipelined and non-pipelined versions of FEC RTL to prove their equivalence. We unrolled the pipelined implementation by cascading the logic in different clock cycles in series.

In order to check equivalence for CRC, we compared CRC output for the basis vector space (1936x1936 identity matrix where each row is a message input) – since we have 242 bytes \* 8bits, any message is going to be a scaled linear combination of the rows of our basis matrix. In our attempts, the formal equivalence checking tool was not able to handle the large state space of CRC code to formally prove equivalence.

#### B. Gate Count and Timing results from RTL Synthesis

We use the post synthesis standard cell count as a proxy for area and power scaling when implementing this logic for link sub-divided ports (separate modules for FEC and CRC were compared). Standard cell count results provided here are by running Synopsys Design Compiler (DC) synthesis with Intel's standard library cells. The design was constrained to a clock frequency of 1GHz for FEC and 2GHz for CRC logic. Table 7 gives the number of logic levels observed on the worst-case timing path post synthesis. We chose 2GHz to reduce the gate count for CRC since the number of levels of logic permits us to do so. However, for FEC, a 2GHz logic would have been challenging with having to pipeline logic that had lots of interdependencies. Standard cell count results for both are summarized in

Table 8. As expected, the levels of logic are about the same between pipelined versions vs non-pipelined version since the XOR tree levels are about the same between flat vs a 4-way partitioned implementation. The difference is significant in the gate count.

These results prove that PCIe 6.0 FEC and CRC can be implemented without any noticeable impact to latency or area. For an entire PCIe stack the latency impact is 1ns in either direction and the gate count impact is less than 0.1% even within the physical layer.

Table 7: Number of logic levels post synthesis (allowing for complex gates available in the library)

	FEC (vs flat)	CRC (vs flat)
Encoder	12 (vs 9)	9 (vs 10)
Decoder	25 (vs 25)	9 (vs 10)

Table 8: Area improvements using the pipelined implementation

	Flat logic (single port)	Flat logic (4 ports) $\sim(4 \times 1 \text{ port})$	Pipeline d logic (4 ports)	Area improvement
FEC encoder	6,185	24,740	9,619	>2.6x
FEC decoder	25,969	103,876	88,755	>1.17x
CRC Encoder/decoder	17,887	71,548	28,082	>2.5x

## VI. CONCLUSIONS

PCI Express is already moving to the 7<sup>th</sup> generation at 128 GT/s using the same Flit, FEC and CRC mechanism. Development of a low-latency FEC and CRC algorithm along with a proven implementation with low-latency and low gate count for low power even with partitioned links is critical for this technology to thrive for 64.0 GT/s and beyond, as PCIe continues its backwards compatible evolution to seventh generation and beyond. With the approaches delineated in this paper to drive low-latency, low-power, and low-cost implementations, we can easily deploy PCIe PHY for PCIe as well as other coherency applications like CXL for 64.0 GT/s and beyond with high-volume manufacturing scaling to hundreds of Lanes in a platform.

## References

- [1] D. Das Sharma, "PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 signaling: a low latency, high bandwidth, high reliability and cost-effective interconnect", IEEE Micro, Jan-Feb 2021
- [2] D. Das Sharma, "PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 signaling: a low latency, high bandwidth, high reliability and cost-effective interconnect", Hot Interconnects, 2020
- [3] PCI-SIG, "PCI Express® Base Specification Revision 5.0, Version 1.0", May 28, 2019
- [4] PCI-SIG, "PCI Express® Base Specification Revision 6.0, Version 1.0", Jan, 2022
- [5] D. Das Sharma, "The Evolution of the PCI-Express® Architecture: Going Strong 15 years and Five Generations Later", Embedded Systems Engineering, Aug 2017
- [6] <http://www.bb-elec.com/Learning-Center/All-White-Papers/Fiber-MTBF-MTTR-MTTF-FIT-Explanation-of-Terms-MTBF-MTTR-MTTF-FIT-10262012-pdf.pdf>
- [7] "Fault-Tolerant Computing: Theory and Techniques , Vol I and II", edited by Pradhan, D. K., ISBN-13: 978-0133082302
- [8] Intel Corp, "PAM4 Signaling Fundamentals", Dec 3, 2019
- [9] C. Liu, "100+ Gb/s Ethernet Forward Error Correction (FEC) Analysis", Signal Integrity Journal, July 9, 2019
- [10] X. Wang, T. Yang, and W. Wang, "FEC Options for Extended Reach of 50/200/400GbE", IEEE 802.3 NG-ECDC
- [11] CXL Consortium, "Compute Express Link 1.1 Specification", www.computeexpresslink.org
- [12] S. Lin and D. J. Costello, "Error Control Coding: Fundamentals and Applications", Prentice-Hall Inc, Englewood Cliffs, NJ, 1983, pp. 141-1
- [13] D. Das Sharma, "A Low-Latency and Low-Power Approach for Coherency and Memory Protocols on PCI Express 6.0 PHY at 64.0 GT/s with PAM-4 Signaling", IEEE Micro, Mar/ Apr 2022
- [14] Ethernet Technology Consortium, "Low Latency Reed Solomon Forward Error Correction", November 2018.