Sparsity-Aware Medium-Density Parity-Check Decoder for McEliece Cryptosystems

Xinmiao Zhang and Zhenshan Xie

***Abstract*—McEliece cryptosystem based on medium-density parity-check (MDPC) codes is one of the finalists for the post-quantum cryptography standard. Although decoder design for low-density parity-check (LDPC) codes used for digital communications is well-investigated, the design of MDPC decoders faces many new challenges due to the different structure in the parity-check matrix. Even though the parity-check matrices of MDPC codes have relatively higher density, they are still very sparse. Previous decoder designs did not explore such sparsity and derive the columns of the parity check matrix one after the other by cyclic shifting. This paper proposes a low-complexity MDPC decoder design by exploiting the sparsity of the parity-check matrix. The processing corresponding to zero segments of each parity check matrix column is skipped to substantially reduce the latency. Moreover, the columns are processed in a novel non-consecutive order to significantly reduce the number of memory writes for deriving all the columns and accordingly the power consumption. For an example MDPC code considered for the standard, the proposed design can reduce both the decoding latency and the number of memory writes by 70% with 35% area saving.**

***Index Terms*—Decoder, McEliece cryptosystem, Medium-density parity-check codes, Post-quantum cryptography**

I. Introduction

The fast development of quantum processors brings the imminent need for new cryptography schemes that are secure against quantum computing attacks. Recently, the National Institute of Standards and Technology (NIST) has initiated the standardization of post-quantum cryptography. The McEliece cryptosystem based on medium-density parity-check (MDPC) codes [1] is one of the finalists. In particular, quasi-cyclic (QC-)MDPC codes are utilized since the key size can be substantially reduced. The parity-check matrix of a QC-MDPC code, $\mathbf{H}$, consists of a few randomly-constructed very large circulant matrices of relatively higher weight instead of a large number of smaller zero and cyclic permutation matrices (CPMs) meticulously designed to avoid short cycles [2] as in LDPC codes for communications. Due to the different structure, previous parallel processing schemes for QC-LDPC decoders that process one or multiple CPMs in each clock cycle are not applicable to QC-MDPC decoders.

To reduce the decoder complexity, the simple bit-flipping (BF) decoding algorithm and its variations have been mainly considered for MDPC codes. Different methodologies have been adopted to decide whether to flip a bit in [3] and error-correction bounds for BF decoding have been derived in [4]. Various implementations of BF MDPC decoders are also available in the literature. Memories contribute to the majority of the BF decoder complexity. To reduce the memory requirement, only the first column of each circulant in the QC-MDPC parity check matrix is stored. The next column can be derived by cyclical shifting. The design in [5] processes 32 bits in a column in each clock cycle, and utilizes an optimized method to store the cyclically shifted column back to BRAMs in FPGA devices. In [6], the syndromes get updated after every bit flipping instead of waiting until the end of each decoding iteration. The decision of whether to flip a bit is typically made based on the syndrome weight. An approximate method is proposed in [7] to reduce the Hamming weight computation complexity. However, existing work did not exploit the sparsity of the $\mathbf{H}$ matrices of the QC-MDPC codes considered for the standard. Even though they have higher density than those of QC-LDPC codes, they are still very sparse and have at most 1.79% nonzero entries.

This paper proposes a low-complexity decoder design for QC-MDPC codes by exploiting the sparsity of their parity check matrices. For moderate parallelism L, each column of the $\mathbf{H}$ matrix has many segments of L zero bits. The corresponding processing can be skipped and accordingly many clock cycles can be saved. Besides, instead of processing the columns in counting order as in prior designs, our proposed design processes column $j+L$ after column j and wraps around to the first un-processed column after each round of processing. This out-of-order scheme still processes each column and does not lead to any performance loss. On the other hand, only one segment of L bits needs to be shifted to derive column $j+L$ from column j in the circulant matrix. As a result, the memory writes can be substantially reduced. The proposed design is synthesized using CMOS process. For an example MDPC code achieving 80-bit security, the proposed architecture with $L=32$ reduces both the latency and the number of memory writes by 70% with 35% area saving.

II. MDPC Code-Based McEliece Cryptosystem and Bit-Flipping Decoding

The $\mathbf{H}$ matrix of an MDPC code used in the McEliece cryptosystem is in the format of $[\mathbf{H}_0|\mathbf{H}_1|\cdots|\mathbf{H}_{n_0-1}]$. Each $\mathbf{H}_i$ is a circulant matrix of dimension $r\times r$ with column weight w. The first columns of $\mathbf{H}_i$ are randomly generated and they form the private key. An invertible $\mathbf{H}_{n_0-1}$ needs to be selected. Then a generator matrix is computed as $\mathbf{G}=[\mathbf{I}|\mathbf{B}^T]$, where $\mathbf{B}=[\mathbf{H}^{-1}_{n_0-1}\mathbf{H}_0|\mathbf{H}^{-1}_{n_0-1}\mathbf{H}_1|\cdots|\mathbf{H}^{-1}_{n_0-1}\mathbf{H}_{n_0-2}]$, and the first columns of the circulants in $\mathbf{B}$ form the public key. For encryption, the plain text is multiplied with $\mathbf{G}$ to calculate a codeword $\mathbf{c}$, which is added with a random vector with at

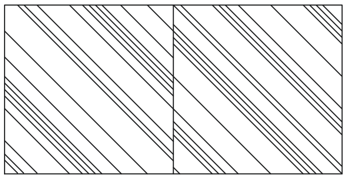


Fig. 1. $\mathbf{H}$ matrix structure for a toy QC-MDPC code with n_0=2.

TABLE I
MDPC CODE PARAMETERS USED FOR THE MCELIECE CRYPTOSYSTEM [7]

security level (bits)	n_0	n	r	w	t
80	2	9602	4801	45	84
	3	10779	3593	51	53
	4	12316	3079	55	42
128	2	19714	9857	71	134
	3	22299	7433	81	85
	4	27212	6803	85	68
256	2	65542	32771	137	264
	3	67593	22531	155	167
	4	81932	20483	161	137

most t nonzero bits to derive $\mathbf{x}$. Let $\mathbf{x} = [x_0, x_1, \cdots, x_{n-1}]$. The decryption is to carry out MDPC decoding on $\mathbf{x}$ to recover the plain text. The parameters of the MDPC codes considered for the McEliece cryptosystem are listed in Table I. For a toy code with $n_0 = 2$, the structure of the $\mathbf{H}$ matrix is illustrated in Fig. 1. The diagonals indicate the nonzero entries. In each circulant, the next column is the current column cyclically shifted by one bit.

The BF decoding algorithm has been considered in most implementations of the McEliece system due to its low complexity [5]–[7]. Denote the set of indices of the nonzero entries in column j of $\mathbf{H}$ by $\mathcal{N}_j$. A basic BF MDPC decoding algorithm is described in Algorithm 1.

Algorithm 1: Bit-Flipping (BF) Decoding Algorithm

```
input: $\mathbf{H}$, $\mathbf{x}$, $th$
Compute $\mathbf{s} = \mathbf{H}\mathbf{x}^T$; Stop and return $\mathbf{x}$ if $\mathbf{s} = \mathbf{0}$
For $i = 0$ to $I_{max}$
  for each column $j$
    flip $x_j$ if $\sum_{i \in \mathcal{N}_j} s_i > th$
  Stop and return $\mathbf{x}$ if $\mathbf{H}\mathbf{x}^T = \mathbf{0}$
```

A vector $\mathbf{x}$ is a codeword iff the syndrome vector, $\mathbf{s}$, is zero. x_j is flipped if the count of the nonzero syndromes participating in the corresponding column of $\mathbf{H}$ is larger than a threshold th. This count is referred to as the nonzero syndrome count for column j. The decoding terminates when a codeword is found. Decoding failure is declared if the maximum iteration, I_{max}, is reached without finding a codeword. The BF algorithm has many variations. th can be dynamically adjusted over the iterations and the bits can be flipped in a probabilistic way [3]. Also the syndromes can be updated after every bit flip [6] instead of being calculated at the end of each iteration.

Although there are many decoder designs for QC-LDPC codes, most of them are processing blocks of CPMs in the $\mathbf{H}$ matrix in each clock cycle and cannot be extended to MDPC codes. Instead, QC-MDPC decoders typically process the $\mathbf{H}$ matrix column by column [5]–[7]. To reduce the memory, the first column of each circulant is stored and the other columns are derived by cyclical shifting.

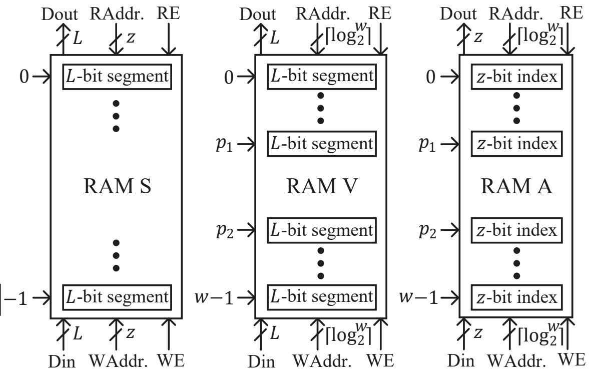


Fig. 2. Memories for storing syndromes and $\mathbf{H}$ matrix information in the proposed design.

III. SPARSITY-AWARE QC-MDPC DECODER

This section proposes a low-latency and low-complexity decoder design for QC-MDPC codes by utilizing the sparsity of their $\mathbf{H}$ matrices. Besides, a novel out-of-order computation scheduling scheme is developed to substantially reduce the number of memory writes needed to derive the cyclically-shifted columns of the $\mathbf{H}$ matrix.

A. *Sparsity-Aware $\mathbf{H}$ Matrix Storage*

Although the $\mathbf{H}$ matrix for an MDPC code is denser compared to that of an LDPC code, it is still very sparse. At most 1.79% of the entries are nonzero for the codes listed in Table I. Also the higher level the security, the sparser the $\mathbf{H}$ matrices. For low-cost implementations, a segment of $L < r$ bits of a column of $\mathbf{H}$ is processed in each clock cycle. For moderate or smaller L, many segments of L bits in each column of $\mathbf{H}$ are zero. For example, for a randomly-generated QC-MDPC code with $r = 4801$ and $w = 45$, if each column of $\mathbf{H}$ is divided into 32-bit segments for 32-parallel processing, at least 106 of the 151 32-bit segments are zero in each column. The zero-segment processing can be skipped and their storage can be eliminated. Accordingly, the number of clock cycles needed and the memory requirement are greatly reduced.

Our design also stores only one column of each circulant of $\mathbf{H}$ at any time. The other columns are derived by cyclical shifting. The information about $\mathbf{h}_j$, a column of $\mathbf{H}$, is recorded using RAM V and RAM A as shown in Fig. 2 in our design. $\mathbf{h}_j$ is divided into L-bit segments from the first bit, and the nonzero L-bit segments are stored consecutively in RAM V. The indices of the nonzero segments are stored in RAM A with the same depth. In the case that $d = r \mod L$ is small, the last d bits of $\mathbf{h}_j$ can be efficiently stored in registers. The depths of both RAM V and RAM A are w. These memories are dual-port. One read and one write at different addresses can be carried out in the same clock cycle. 'RE' and 'WE' in Fig. 2 are the read and write enable pins, respectively. For an (n, r, w)=$(9602, 4801, 45)$ MDPC decoder with $L = 32$, the memory requirement for recording the information about $\mathbf{h}_j$ is reduced from 4801 bits in [6] to $w \times (L + \lceil \log_2 w \rceil) = 45 \times (32 + \lceil \log_2 45 \rceil) = 1710$ bits using our design.

Different from $\mathbf{h}_j$, every syndrome needs to be stored. $\mathbf{s}$ is divided into segments of L bits and stored in RAM S. Similarly, the last d syndromes can be more efficiently stored in registers if d is small. In this case, the dimension of RAM S is $\lfloor r/L \rfloor \times L$ as shown in Fig. 2.

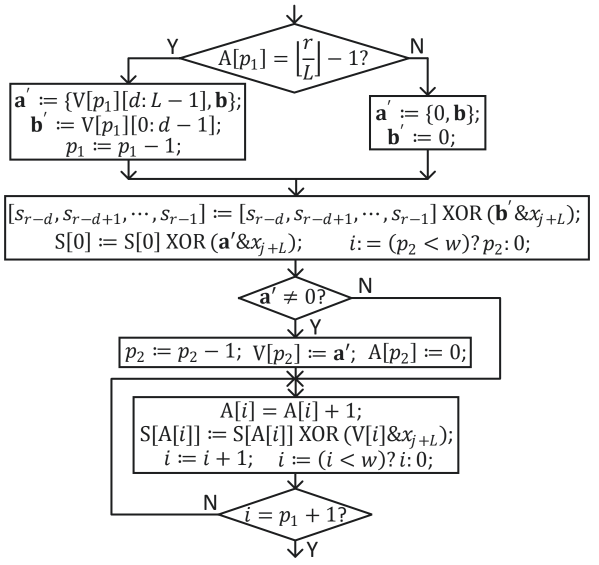
B. Out-of-Order Column Processing-Shifting by L Columns

In all previous designs, such as [5] and [6], $\mathbf{h}_{j+1}$ is processed after $\mathbf{h}_j$ and it is generated by cyclically shifting $\mathbf{h}_j$ by one bit. In this case, each L-bit segment of $\mathbf{h}_j$ stored in the memory is overwritten with a new segment of $\mathbf{h}_{j+1}$. The repeated memory writes cause high power consumption [8].

To substantially reduce memory writes, this paper proposes to process column $\mathbf{h}_{j+L}$ after $\mathbf{h}_j$. In this case, the i-th segment of $\mathbf{h}_j$, except the last one, is the $(i+1)$-th segment of $\mathbf{h}_{j+L}$. Hence, no write to RAM V is needed for these segments, and only the contents of RAM A, whose width and power consumption are much lower than that of RAM V, need to be incremented by one. Utilizing this idea, the columns are processed in the order of $\mathbf{h}_0, \mathbf{h}_L, \mathbf{h}_{2L}, \cdots$ in our design. After $\mathbf{h}_{r-d}$ is processed, the decoding continues with $\mathbf{h}_1, \mathbf{h}_{L+1}, \mathbf{h}_{2L+1}, \cdots$. Such a wrap-around process is repeated until all columns of $\mathbf{H}$ are processed. This subsection addresses the processing of $\mathbf{h}_{j+L}$ after $\mathbf{h}_j$. The processing of $\mathbf{h}_1$ after $\mathbf{h}_{r-d}$ and other wrap around will be discussed in the next subsection.

From Table I, r is always a prime number and hence $d \neq 0$. Denote the last d bits of $\mathbf{h}_j$ by $\mathbf{b} = [h_{j,r-d}, h_{j,r-d+1}, \cdots, h_{j,r-1}]$ and the L bits right before the d bits by $\mathbf{a} = [h_{j,r-d-L}, h_{j,r-d-L+1}, \cdots, h_{j,r-d-1}]$. For $\mathbf{h}_{j+L}$, the last d bits should be $\mathbf{b}' = [h_{j,r-d-L}, \cdots, h_{j,r-L-1}]$ and its first L-bit segment is $\mathbf{a}' = [h_{j,r-L}, \cdots, h_{j,r-1}]$. These notations are illustrated in Part (1) of Fig. 3. In the case that $\mathbf{a} = \mathbf{0}$ and $\mathbf{a}' \neq \mathbf{0}$, $\mathbf{h}_{j+L}$ has one more nonzero segment compared to $\mathbf{h}_j$. If $\mathbf{a} \neq \mathbf{0}$ and $\mathbf{a}' = \mathbf{0}$, then $\mathbf{h}_{j+L}$ has one less nonzero segment. The number of nonzero segments and their locations change over the columns of $\mathbf{H}$. To keep track of the storage of nonzero segments in RAM V and A, two pointers, p_1 and p_2, are utilized in our design as shown in Fig. 2. p_1 is always pointing at the last nonzero L-bit segment. Initially, p_2 is w and the first nonzero segment is stored at address 0. Whenever $\mathbf{a}' \neq 0$ for $\mathbf{h}_{j+L}$, p_2 is decreased by one and $\mathbf{a}'$ is written into the address pointed by p_2. In other words, p_2 points to the first nonzero segment of $\mathbf{h}_{j+L}$ if at least one of the $\mathbf{a}'$ vectors from the previous column shifting is nonzero.

$\mathbf{h}_{j+L}$ can be processed after $\mathbf{h}_j$ following the data flow chart in Part (2) of Fig. 3. In this chart, ':=' denotes value assignment. V$[i]$ stands for the content of RAM V at address i and V$[i][j{:}k]$ denotes the j-th through k-th bits of V$[i]$. Similar notations are used for the other memories and vectors. The nonzero segments of $\mathbf{h}_{j+L}$ are processed in increasing order. The index of the last nonzero segment of $\mathbf{h}_j$ is A$[p_1]$. If it equals $\lfloor r/L \rfloor - 1$, it means $\mathbf{a} = $ V$[p_1]$ and p_1 for $\mathbf{h}_{j+L}$ should be decreased by one. Otherwise, $\mathbf{a} = 0$. Then $\mathbf{a}'$ and $\mathbf{b}'$ are derived accordingly. They are used to update the corresponding syndromes and $\mathbf{b}'$ is written back into registers. If $\mathbf{a}' \neq 0$, then it is the first nonzero segment of $\mathbf{h}_{j+L}$. It is stored into RAM V at address $p_2 - 1$ and its index to be stored at RAM A is 0. The index i in Fig. 3 is initially the address of the first nonzero segment of $\mathbf{h}_j$. Each nonzero segment



Y
$A[p_1] = \lfloor \frac{r}{L} \rfloor - 1?$
N
$\mathbf{a}' := \{V[p_1][d{:}L-1], \mathbf{b}\};$
$\mathbf{b}' := V[p_1][0{:}d-1];$
$p_1 := p_1 - 1;$
$\mathbf{a}' := \{0, \mathbf{b}\};$
$\mathbf{b}' := 0;$
$[s_{r-d}, s_{r-d+1}, \cdots, s_{r-1}] := [s_{r-d}, s_{r-d+1}, \cdots, s_{r-1}]$ XOR $(\mathbf{b}' \& x_{j+L});$
$S[0] := S[0]$ XOR $(\mathbf{a}' \& x_{j+L});$ $i := (p_2 < w)?\, p_2 : 0;$
$\mathbf{a}' \neq 0?$
N
Y
$p_2 := p_2 - 1;$ $V[p_2] := \mathbf{a}';$ $A[p_2] := 0;$
$A[i] = A[i] + 1;$
$S[A[i]] := S[A[i]]$ XOR $(V[i] \& x_{j+L});$
$i := i + 1;$ $i := (i < w)?\, i : 0;$
N
$i = p_1 + 1?$
Y

Fig. 3. Part (1): Illustration of shifting for generating $\mathbf{h}_{j+L}$ from $\mathbf{h}_j$; Part (2): Flow chart for processing $\mathbf{h}_{j+L}$ after $\mathbf{h}_j$.

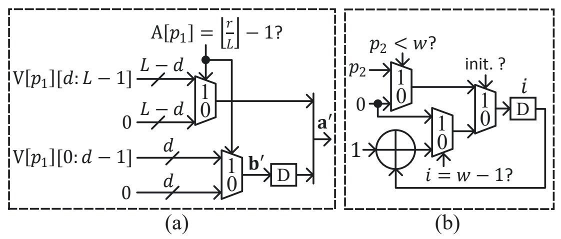


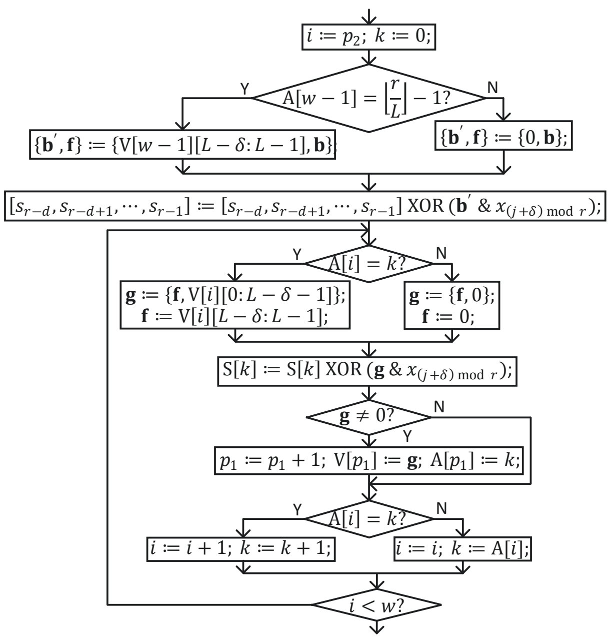
Fig. 4. Architectures of control signal generation for implementing $\mathbf{H}$ column shifting by L bits.

of $\mathbf{h}_j$, except the last one if its index is $\lfloor r/L \rfloor$-1, becomes a nonzero segment of $\mathbf{h}_{j+L}$ with the corresponding index increased by one. This is achieved by increasing A$[i]$ by one. The corresponding syndromes are also updated simultaneously as the index updating. This is repeated for every such nonzero segment by increasing i to $w-1$ and then going through $i = 0$ to p_1.

The $\mathbf{a}'$, $\mathbf{b}'$, and index i of the proposed decoding process according to the data flow in Fig. 3 can be generated by the architectures in Fig. 4. The control signals for the multiplexers in this figure are derived by simple comparators. The p_1 and p_2 updating is also implementable by simple logic.

C. Out-of-Order Column Processing-Shifting by $1 < \delta < L$ Columns

In the first round of processing $\mathbf{H}$, columns $\mathbf{h}_0, \mathbf{h}_L, \cdots, \mathbf{h}_{r-d}$ are handled. Then in each of the later rounds, the decoder moves back to the first un-processed column and continues by jumping over L columns each time until the index of the column becomes larger than $r-1$ if it is further increased by L. This process is repeated until every column has been processed. For example, in the case of $r = 4801$ and $L = 32$, $\mathbf{h}_0, \mathbf{h}_{32}, \cdots, \mathbf{h}_{4800}$ are processed in the first round followed by $\mathbf{h}_1, \mathbf{h}_{33}, \cdots, \mathbf{h}_{4769}$ in the second round and $\mathbf{h}_2, \mathbf{h}_{34}, \cdots, \mathbf{h}_{4770}, \cdots$ in later rounds. Let the last column processed in round l be $\mathbf{h}_{u_l}$. The first column handled in the $(l+1)$-th round is $\mathbf{h}_l$. To get $\mathbf{h}_l$, $\mathbf{h}_{u_l}$ needs to be cyclically shifted by $(l+r) - u_l$ bits. This



$i := p_2;\ k := 0;$
$\mathrm{A}[w-1] = \lfloor \frac{r}{L} \rfloor - 1?$
Y
N
$\{\mathbf{b}', \mathbf{f}\} := \{\mathrm{V}[w-1][L-\delta : L-1], \mathbf{b}\}$
$\{\mathbf{b}', \mathbf{f}\} := \{0, \mathbf{b}\};$
$[s_{r-d}, s_{r-d+1}, \cdots, s_{r-1}] := [s_{r-d}, s_{r-d+1}, \cdots, s_{r-1}]$ XOR $(\mathbf{b}'\ \&\ x_{(j+\delta) \bmod r});$
$\mathrm{A}[i] = k?$
Y
N
$\mathbf{g} := \{\mathbf{f}, \mathrm{V}[i][0:L-\delta-1]\};$
$\mathbf{f} := \mathrm{V}[i][L-\delta:L-1];$
$\mathbf{g} := \{\mathbf{f}, 0\};$
$\mathbf{f} := 0;$
$\mathrm{S}[k] := \mathrm{S}[k]$ XOR $(\mathbf{g}\ \&\ x_{(j+\delta) \bmod r});$
$\mathbf{g} \neq 0?$
N
Y
$p_1 := p_1 + 1;\ \mathrm{V}[p_1] := \mathbf{g};\ \mathrm{A}[p_1] := k;$
$\mathrm{A}[i] = k?$
Y
N
$i := i + 1;\ k := k + 1;$
$i := i;\ k := \mathrm{A}[i];$
$i < w?$

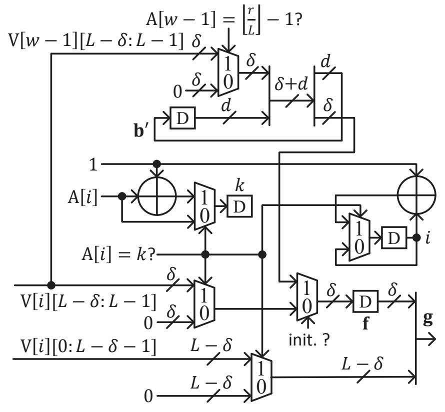
Fig. 5. Flow chart for shifting by δ bits ($1 < \delta < L$).

subsection presents the details for shifting $\mathbf{h}_j$ by $1 < \delta < L$ bits that is needed when the decoder moves back to the first un-processed column.

Different from shifting by L bits, every nonzero segment of $\mathbf{h}_j$ needs to be shifted by δ bits to derive those of $\mathbf{h}_{j+\delta}$. In the processing of the first column in round l, the last d bits of this column are derived first. Since $p_1 = -1$ after processing the last column in round $l-1$, only the nonzero segments stored at addresses p_2 to $w-1$ need to be processed one by one to derive the first column for round l. The nonzero segments of the first column of round l and their indices are written to RAM V and A, respectively, at addresses starting from 0.

The proposed data flow for shifting $\mathbf{h}_j$ by δ bits ($1 < \delta < L$) is shown in Fig. 5. Similarly, $\mathbf{b}$ and $\mathbf{b}'$ are used to denote the last d bits of $\mathbf{h}_j$ and $\mathbf{h}_{(j+\delta) \bmod r}$, respectively. $\mathbf{f}$ is a vector utilized to track the δ bits shifted out of the register or a data segment. $\mathbf{g}$ concatenates $\mathbf{f}$ and the bits in the next segment and it forms an L-bit segment for $\mathbf{h}_{(j+\delta) \bmod r}$. i is initialized to p_2 and points to the nonzero segments of $\mathbf{h}_j$ stored in RAM V and A. k is used to track the indices of the segments of $\mathbf{h}_{(j+\delta) \bmod r}$ that have been processed. In the beginning, $\mathbf{b}'$ is derived in a similar way as in Fig. 3. Also the δ bits shifted out from the register are held in $\mathbf{f}$ and the last d syndromes are updated. Since $\delta < L$, some bits in segment k of $\mathbf{h}_j$ are also bits in segment k of $\mathbf{h}_{(j+\delta) \bmod r}$. If $\mathrm{A}[i] = k$, the other bits of segment k comes from $\mathrm{V}[i]$. Otherwise, the other bits are '0's. If the segment $\mathbf{g}$ is zero, nothing needs to be recorded. If $\mathbf{g} \neq 0$, then $\mathbf{g}$ and index k are stored in RAM V and A, respectively, and p_1 is increased by one. Similar to that in Fig. 3, p_1 is the pointer for the last nonzero segment stored in the memories. If $\mathrm{A}[i] \neq k$, the next possible nonzero segment of $\mathbf{h}_{(j+\delta) \bmod r}$ has index $\mathrm{A}[i]$. Instead of increasing k by one each time, it can be directly set to $\mathrm{A}[i]$ to save clock cycles. The processing of $\mathbf{h}_{(j+\delta) \bmod r}$ is completed when i reaches $w-1$.

The control signal generation architecture for shifting an $\mathbf{H}$ column by δ ($1 < \delta < L$) bits is shown in Fig. 6. Similarly, the multiplexer select signals are derived by simple comparators. The updating of p_1 and RAM contents is implemented by simple logic and hence is not included in Fig. 6.



$\mathrm{A}[w-1] = \lfloor \frac{r}{L} \rfloor - 1?$
$\mathrm{V}[w-1][L-\delta:L-1]$
δ
0
$\mathbf{b}'$
D
d
$\delta+d$
1
$\mathrm{A}[i]$
k
D
i
$\mathrm{A}[i] = k?$
$\mathrm{V}[i][L-\delta:L-1]$
init. ?
$\mathbf{f}$
$\mathbf{g}$
$\mathrm{V}[i][0:L-\delta-1]$
$L-\delta$

Fig. 6. Architecture of control signal generation for implementing $\mathbf{H}$ column shifting by δ ($1 < \delta < L$) bits.

TABLE II
Configurations of memories for $(n, r, w) = (9602, 4801, 45)$ MDPC decoder with $L = 32$

proposed design	RAM S	RAM V	RAM A
depth×width	150×32	45×32 (2 copies)	45×8 (2 copies)
prior design [6]	syndrome RAM	$\mathbf{H}$ column RAM	-
depth×width	151×32	151×32 (2 copies)	-

TABLE III
Synthesis results using TSMC $65nm$ process with $T = 1ns$ for $(n, r, w) = (9602, 4801, 45)$ MDPC decodesr with $L = 32$

	Logic Area (μm^2)	Memory (Bits)	Total Area (# of NAND2)
prior design [6]	1406	14496	22720
proposed design	2831	8400	14566

IV. Complexity Comparisons

In this section, the proposed design is compared with prior design that does not utilize the sparsity of $\mathbf{H}$ and processes the columns in counting order for an example $(n, r, w) = (9602, 4801, 45)$ MDPC decoder with $L = 32$.

In MDPC decoding, the syndromes are computed as illustrated in Fig. 3 and 5. The nonzero syndrome count for column j, $\sum_{i \in \mathcal{N}_j} s_i$, can be computed by ANDing each L-bit nonzero segment of $\mathbf{h}_j$ with the corresponding syndrome segment and then adding up the number of nonzero bits in the AND result. To reduce the complexity of finding the number of nonzero bits in an L-bit segment, the L bits can be divided into multiple groups. The count of nonzero bits for each group can be found by using a lookup table (LUT), and the counts for different groups are added up using a tree structure. On the other hand, the data path and area of the adder tree also need to be considered. From synthesis results using TSMC $65nm$ process with $1ns$ timing constraint, dividing each 32-bit segment into 8 4-bit groups leads to the smallest area.

For the example code, the depths and widths of the RAMs required in the proposed design are listed in Table II. Two copies of RAM V and RAM A are needed since there are $n_0 = 2$ circulant matrices in $\mathbf{H}$. Comparatively, the previous design [6] needs a RAM with size 151×32 to store a column of $\mathbf{H}$ since all segments are recorded. The synthesis results of the logic components of both the proposed design and that from [6] using TSMC 65nm process under timing constraint $T = 1ns$ are listed in Table III. Each NAND2 gate occupies around 1.44 μm^2. Assume that each bit of memory takes the area of 1.5 NAND2 gates [9]. The equivalent total gate counts in terms of NAND2 can be computed as shown in Table III. Although the proposed design has more complicated logic than that in [6], it requires much smaller memories. Overall, our design achieves $(22720 - 14566)/22720 \approx 35\%$ area saving. The same LUTs and adder tree can be utilized to compute the nonzero syndrome count in the proposed decoder and that in [6]. Their data path is much longer than that of the architectures in Fig. 4 and 6. Hence, the proposed decoder has the same critical path as the design in [6].

For the example code, $\mathbf{h}_j$ has at most $w = 45$ nonzero segments and it takes at most 45 clock cycles to process $\mathbf{h}_{j+L}$ after $\mathbf{h}_j$. To process $\mathbf{h}_{(j+\delta) \mod r}$ with $1 < \delta < L$, one more clock cycle is needed if the shifting of a nonzero segment by δ bits generates a new nonzero segment. Hence, it takes at most $45 \times 2 = 90$ clock cycles. However, only $32/4801 \approx 0.67\%$ of the columns need 90 instead of 45 clock cycles to process. On the other hand, 151 clock cycles are required to process each column in the design from [6]. Therefore, the proposed design achieves around $(151 - 45)/151 \approx 70\%$ reduction on the latency. The number of memory writes is also reduced by 70% in our design. Additionally, most of the writes in our design are to RAM A, which is narrower than RAM V.

When L is larger, the portion of nonzero segments in $\mathbf{H}$ is larger, and hence the latency reduction achievable by the proposed design is less significant. On the other hand, the width of RAM A is reduced for larger L. Since most of the writes are into RAM A in our design, the power consumption of memory writes is further reduced. For MDPC codes used to achieve higher security level, the $\mathbf{H}$ matrices are even sparser. Accordingly, the proposed design would achieve even more significant reductions on latency and memory writes.

Our proposed design also helps to simplify software implementations of MDPC decoders. Since the nonzero segments form a small portion of each column of $\mathbf{H}$ and only the nonzero segments are processed in the decoding, our design would also lead to significant reductions in latency and memory requirement in software implementations.

V. Conclusions

This paper proposes a low-complexity MDPC decoder design for the McEliece cryptosystem by exploiting the sparsity of the parity check matrix. Only the nonzero segments of $\mathbf{H}$ are stored and processed to reduce the latency and memory requirements. Besides, a novel out-of-order processing scheme is developed to derive the columns of $\mathbf{H}$ with substantially reduced memory writes. Detailed data flows with simple logic are also developed to implement the proposed scheme. Analysis and synthesis results show that the proposed design achieves substantial latency and memory write reductions with significantly smaller area. Future research will address the implementation of other MDPC decoding schemes.

References

[1] D. J. Bernstein, et al. "Classic McEliece: conservative code-based cryptography," available at https://classic.mceliece.org/nist.html.

[2] S. Lin and D. J. Costello, *Error Control Coding*, Pearson, 2004.

[3] H. Bartz and G. Liva, "On decoding schemes for the MDPC-McEliece cryptosystem," *Proceedings of International ITG Conference on Systems, Communications, and Coding*, Mar. 2019, pp. 1-6.

[4] P. Santini, M. Battaglioni, M. Baldi, and F. Chiaraluce, "Analysis of the error correction capability of LDPC and MDPC codes under parallel bit-flipping decoding and application to cryptography," *IEEE Transactions on Communications*, vol. 68, no. 8, pp. 4648-4660, Aug. 2020.

[5] I. V. Maurich and T. Güneysu, "Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices," *IEEE Design, Automation & Test in Europe Conference & Exhibition*, 2014, pp. 1-6.

[6] I. V. Maurich, T. Oder, and T. Güneysu, "Implementing QC-MDPC McEliece encryption," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 3, pp. 1-27, Apr. 2015.

[7] J. Hu and R. Cheung, "Area-time efficient computation of Niederreiter encryption on QC-MDPC codes for embedded hardware," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1313-1325, Aug. 2017.

[8] M. Sharifkhani, "Design and analysis of low-power SRAMs," Ph.D dissertation, Electrical and Computer Engineering, University of Waterloo, Canada, 2006. [Online]. Available: https://www.collectionscanada.gc.ca/obj/s4/f2/dsk3/OWTU/TC-OWTU-1005.pdf

[9] X. Chen and C. Wang, "High-throughput efficient non-binary LDPC decoder based on the simplified min-sum algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 11, pp. 2784-2794, Nov. 2012.