Parallel Polar Encoding in 5G Communication

Yang Guo[1 2], Shaolin Xie[1], Zijun Liu[1], Lei Yang[1 2], Donglin Wang[1]
[1] Institute of Automation, Chinese Academy of Sciences (CASIA), Beijing, China
[2] University of Chinese Academy of Sciences (UCAS), Beijing, China
E-mail:{guoyang2014, shaolin.xie, zijun.liu, yanglei2013, donglin.wang}@ia.ac.cn

***Abstract*—Because of its theoretical capacity-achieving property, polar code has become the coding scheme of the control channel in the 5G communication standard. Although its encoding complexity is low, the data dependency in polar code makes it difficult to parallelize. This paper proposes a parallel polar encoding method for 5G communication and evaluates its performance with extended digital signal processor (DSP) instructions. Compared with the existing field-programmable gate array (FPGA) implementation, the performance improved by 300× with negligible area and power overhead. The extended instructions are based on our in-house DSP architecture, but the parallel scheme is applicable to other single instruction multiple data (SIMD) architectures.**

Key words—Polar code, 5G communication system, DSP, SIMD

I. Introduction

Polar code [1] is the first provable capacity-achieving channel code and has become one of the most attractive codewords in the coding theory community. At the 3GPP RAN1# 87 conference in Las Vegas in November 2016, polar code became the coding scheme for control channels in the 5G eMBB (enhanced mobile broadband) scenario [2]. Although polar encoding has lower computational complexity than other channel codes (such as turbo code and low-density parity-check [LDPC]), its intrinsic data dependency makes it difficult to parallelize, which results in high latency and affects the overall throughput.

Polar encoding has been studied in several other works, but most of them targeted filed-programmable gate arrays (FPGAs). Sarkis et al. [3] reduced the complexity of systematic encoding via matrix transformation. Other groups [4-6] partially parallelized basic polar encoding based on the fast Fourier transform (FFT) folding transformation. Zhang et al. [7] designed a hardware pipeline structure of the basic polar encoder.

Our work optimizes the polar encoding for a programmable digital signal processor (DSP), taking advantage of the wide single instruction multiple data (SIMD) data planes to parallelize the encoding while maintaining programmability. The optimization is based on our in-house mathematic processing unit (MaPU) architecture [8], but the parallel scheme is applicable to other SIMD architectures. In addition, our work focuses only on 5G standard polar code that uses *basic polar* instead of *systematic polar* [9-10], and the code length is limited to 1024 bits [11-12]. Few works meet the 5G standard [13]. Table I shows a comparison with other related works.

TABLE I. Related Work

Related Work	Coding Scheme	Implement	Optimal Code Length
Sarkis, et al [3]	systematic polar	FPGA	16384
Yoo, et al [4]	basic polar	ASIC	8192
Raj, et al [5]	basic polar	FPGA	Not specified
Arpure, et al [6]	basic polar	FPGA	Not specified
Zhang, et al [7]	basic polar	FPGA	Not specified
Polaran[13]	basic polar	FPGA	128,1024
Our Work	basic polar	DSP	128,1024

Compared to other studies, our work optimizes polar encoding for DSP that operates with general register in byte granularity. The contributions of our work are as follows.

- We proposed the parallel polar encoding scheme with 300× performance improvement for 5G communication standard.
- We proposed the DSP instructions that can efficiently support the proposed encoding algorithm.
- We evaluated the overall performance at circuit level implementation.

The paper is organized as follows. The basic operation in polar encoding is introduced first, and then we present our parallel encoding scheme for DSP. Next, we describe the DSP instructions for polar encoding, and finally discuss the evaluation method and results.

II. Preliminaries

Polar code is a linear block code. A code block is identified by a parameter vector (N, K, A, μ_{A^c}), as shown in Fig.1. N is the code length. K is the number of message bits in a code block. K/N is the bit rate. The set A and A^C represents the position of the message bits and frozen bits in the code block respectively. The set μ_A is message bit vector. The set μ_{A^c} is the frozen bit vector, which is usually fixed to zero. Its generator matrix G_N is the n-order Kronecker product [14] of $F = \begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$, denoted by $G_N = F^{\otimes n}$, Where $n = log_2 N$. The equation may be written as (1).

$$x = \mu_A G_N(A) \oplus \mu_{A^c} G_N(A^C) \tag{1}$$

$\oplus$ denotes mod-2 sum. $G_N(A)$ represents the sub-matrix that is indexed by set A. The complexity of encoding based on matrix multiplication is $O(N^2)$.

The Strategic Priority Research Program of Chinese Academy of Sciences (under Grant XDA-06010402) and Youth Innovation Promotion Association of Chinese Academy of Sciences (Y5S7061G51)

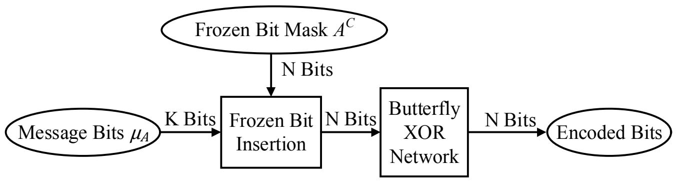

Fig.1 **Basic polar encoding**. K bits message is inserted with frozen bits according to the N bits mask. The N bits result is then fed into the butterfly XOR network.

To reduce the complexity of the algorithm to $O(Nlog_2N)$, it is generally processed via a butterfly XOR network [1]. The encoding process includes two main steps: *Frozen Bit Insertion* and *Butterfly XOR Network*, as shown in Fig.1

A. *Frozen Bit Insertion*

The operands of frozen bit insertion are an N-bit mask string and a K-bit message string. The bit "0" in mask indicates the position of a frozen bit, and the bit "1" indicates the position of a message bit. Take a code block (N=32, K=24) as an example:

Message: 11100110_01101101_00010010

Mask: 11111011_11110110_11111000_11111010

Result: **11100011_00110010_10100**000_**01001**0**0**0

The bold bits in the result come from the message string, which corresponds to the bit "1" in mask. The other bits (which are all zeros) correspond to the bit "0" in mask. This operation is described with *Algorithm 1*.

```
Algorithm 1: Frozen Bits Insertion
Input: mask string P (N bits), message string μ (K bits).
Output: result string W (N bits).

1 Initialize W[0:K-1] ← μ and W[K:N-1] ← 0
2  for i = 0 : N-1
3    if P[i] = 0 then
4      W[i+1:N-1] ← W[i:N-2]
5      W[i] ← 0
6    end if
7  end for
```

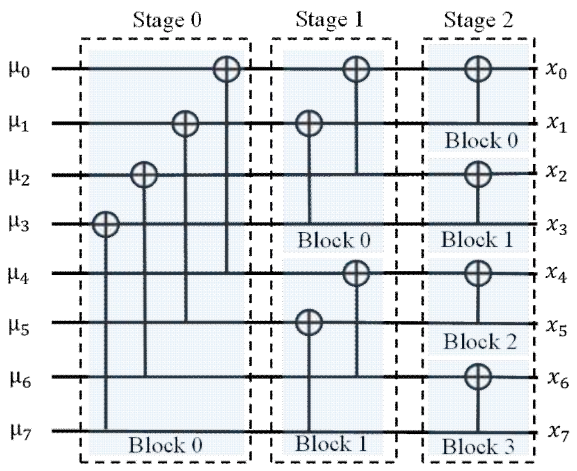

Fig.2 XOR butterfly network example for N = 8.

B. *Butterfly XOR Network*

After frozen bits insertion, the message is processed by a butterfly XOR network. Fig.2 shows that it takes multiple stages. For bit length N, it includes $\log_2 N$ stages. In the ith ($0 \leq$ i $\leq$ $\log_2 N$-1) stage, the N bits are divided into 2^{i+1} blocks. In each block, the top half will XOR with the bottom half to generate the top half output, whereas the bottom half output is just the copy of the bottom half input. Although the network is intrinsically parallel, its bit-level addressing pattern is difficult to implement with DSP. The solutions are discussed in the following sections.

III. FROZEN BIT INSERTION PARALLELIZATION

In *Algorithm 1*, the position of one message bit depends on that of the previous bit, which makes it difficult to parallelize. When N increases, the latency of the serial increases in a linear manner. The directly mapped circuit will show very poor performance and is not adaptable to different code lengths.

The principle of our idea is two-fold. *First, we divide the long operation into several parallel group operations*. The size of the group can be flexible. It can be byte, word or other size. An array of specially designed units can process the group operations in parallel so we can easily incorporate this scheme into DSP and take advantage of the massive parallel data planes in SIMD. The logic level of each group operation is low; thus, it can be done in 1 clock cycle and can be fully pipelined.

Second, we pre-compute the parameters, which is an intrinsic serial operation, and then parallel apply these parameters in real time encoding, converting the serial frozen bit insertion to fully parallel. This idea originates from two important observations: First, the data dependency is only related to the mask; and second, the 5G standard [11] defined a fixed number of masks for different lengths and rates. Combining these two factors, we found that frozen bit insertion can be parallelized in real time processing.

To break down long bit stream operations into parallel group operations, we found in *Algorithm 1* that the number of bits "1" in one group of mask is one group size at most, which means that the message bits in one group of the result will be derived from only two adjacent groups at most. We call these two adjacent groups the Low group and the High group.

For each result group, we can calculate the indices of the corresponding adjacent groups. For the result group vector, the indices of the low groups and the high groups are stored in an *L* and *H* vector respectively. The position of the first bit "1" in each mask group corresponding bit in a message group can also be calculated and stored in an *M* vector. The *L*, *H*, and *M* vectors are only related to the mask and are independent of the message string; thus, for a fixed mask, these vectors can be calculated in advance and stored in memory for later use.

The 3GPP TSG-RAN WG1#88 [11] conference proposed fixed masks for different lengths and rates. In our proposed scheme, the *L*, *H*, and *M* vectors of these masks are pre-computed and stored in the data memory. The DSP will load these vectors to facilitate frozen bits insertion in parallel. Fig.3 shows bits grouping and how to utilize parameters to get result string. And Fig.4 shows the processing each group including three steps.

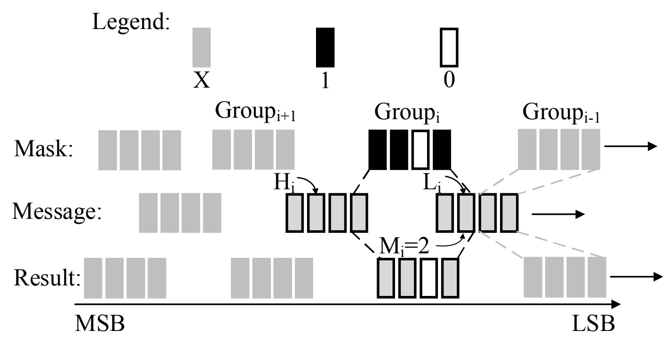

Fig.3 Bit Grouping and Parameters: The bits in Mask, Message, and Result are divided into groups that have the same size (4 in this example). L_i and H_i are the Message (Low and High) group indices for Result group i. In this example, Mask group i have 3 ones and 1 zero, which means the Result group i needs 3 bits from Message. The location of these bits in the Message stream is determined by how many bits has been consumed by the previous Result groups (group i-1 to group 0). In this example, Result group i need 2 bits from the Low group and 1 bits from High group. The value of M_i is the first bits location in Low group, which is equal to total bits consumed modulo by group size.

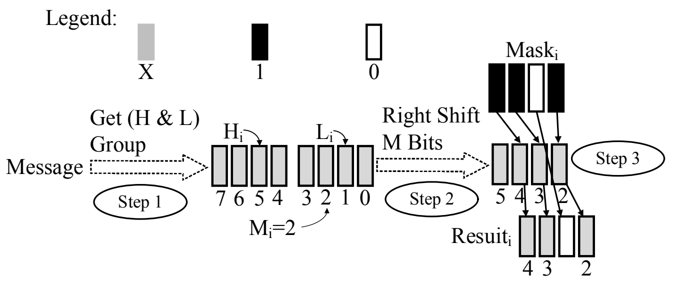

Fig.4 Processing in Each Group: The step 1 is that the corresponding message groups are extracted to 2-group-size intermediate result by indices (L and H vectors). The step 2 is that intermediate result is right shifted according to M vectors. The step 3 is that each group data expand according to corresponding mask group.

IV. Hardware Implementation And Instructions

To evaluate the parallelization schemes above, we extended our in-house MaPU [8] instruction set and implemented it with register transfer level (RTL) and pushed the design through a standard chip flow. MaPU is a novel DSP architecture that uses 512-bit SIMD vector processing units, as shown in Fig.5. It includes several function units (FUs) that operate in parallel. The microcode pipeline uses coarse grain reconfigurable architecture (CGRA), in which multiple FUs are connected by a configurable compact crossbar. The inputs and outputs of FUs can be chained together to handle complex signal processing kernels like FFT without accessing the local memory. Intermediate data are stored in temporary registers referenced with "**T**" and a suffix (for example, *T0*, *Tm*, *Tn*, and *Ts*). More details about the MaPU architecture were given by Wang et al. [8].

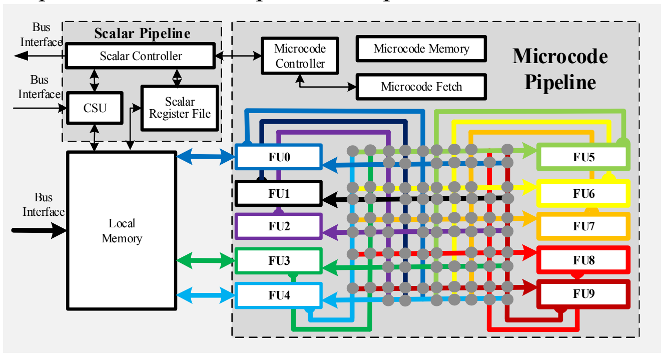

Fig.5 **MaPU Architecture**: Scalar pipeline supports 32-bit SIMD, whereas microcode pipeline supports 512-bit SIMD. The operations and interconnection of FUs are controlled by the microcode.

A. Frozen Bit Insertion Implementation

Taking into account the architecture of MaPU and the degree of parallelism of the algorithm, the size of a group in Frozen Bit Insertion is designed to be one byte.

The process of frozen bit insertion: **Longitudinal Concentration**, **Concatenation and Right Shifting**, and **Insertion in Byte** as shown in Fig.7.

Longitudinal Concentration is corresponded to the step 1 in Fig.4. The Shuffle unit (shown in Fig.6) [8] uses vectors *L* and *H* as the indices to select the adjacent bytes in a message string. The message bits of each byte in the result string are longitudinally concentrated into two *T* registers. High bytes are concentrated into *T1*, and low bytes are concentrated into *T0* as shown in Fig.7 (a).

Concatenation and Right Shifting is corresponded to the step 2 in Fig.4. Two *T* registers are longitudinally concatenated in byte granularity, where the byte from *T1* is placed in the high position and the byte from *T0* is placed in the low position. These two bytes are then right-shifted, and the result only takes the low byte after shifting. The *Concatenation and Right Shifting* of one byte is shown in Fig.7 (b). *M* contains the amount of shifting.

In this way, the serial zero insertion operation in a long string is converted into several parallel byte operations. The step 3 in Fig.4 is corresponded to **Insertion in Byte**, as shown in Fig.7 (c). Its inputs are data stored in the T2 register and a mask stored in another T register. If the corresponding bit of the mask is 0, the result bit is 0; otherwise, the result bit is from *T2*.

B. Butterfly XOR Network Implementation

To implement the XOR butterfly network in an SIMD manner, we optimize the original network shown in Fig.2. The ith stage now includes 2^{i+1} blocks, and a *bit interleave* substage is introduced.

In the bit interleave substage, the blocks are interleaved in an even-odd manner, in which the even blocks are extracted into the top half operand and the odd blocks are extracted into the bottom half operand. In the XOR substage, the blocks in the top half will XOR with the bottom half to generate the top half input of the next stage, whereas the bottom half input is just a copy of the bottom half blocks.

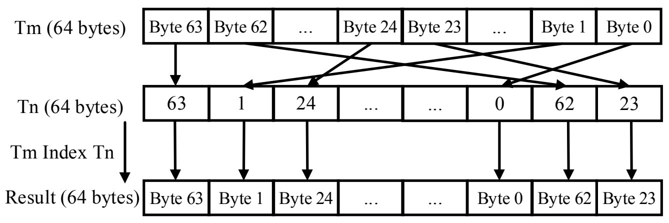

Fig.6 The **original shuffle unit** operates in byte granularity, in which Result[i] = *Tm*[*Tn*[i]]. For example, *Tn*[0]= 23, so the value of Result[0] = *Tm*[23]=Byte23.

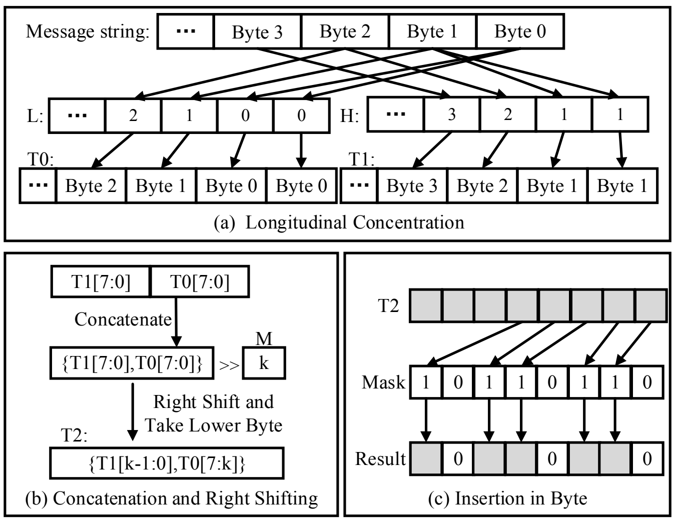

Fig.7 Parallel scheme of frozen bits insertion: (a) Data in message string are extracted into T0 and T1 in byte granularity according to vectors L and H, respectively. (b) 512 bits T0 and T1 are divided into 64 bytes of segment. Each byte performs the same operation. The corresponding bytes of T0 and T1 are concatenated into half-words of intermediate result. The intermediate results are then shifted right, and the number of the shifted bits is determined by the corresponding byte data in vector M. The lowest 8 bits of the half word after right shifting are taken as the final result. (c) 512 bits T2 and mask string are divided into 64 bytes of segment. To form the resulting byte, each byte of T2 is inserted with zero according to the corresponding mask string byte.

Fig.8 shows an example in which N = 1024. At stage 0, the N-bit code block is divided into two blocks with a size of N/2. The even block (block 0) is placed in the *Tm* register, and the odd block (block 1) is placed in the *Tn* register. The XOR result and *Tn* become the N-bit inputs of stage 1, in which the N bits are divided into 4 blocks with a size of N/4. At stage 1, the even blocks (block 0 and block 2) are placed in the *Tm* register, and the odd blocks (block 1 and block 3) are placed in the *Tn* register.

C. *Optimized Instruction Design*

To optimize the parallel polar encoding on MaPU, we first implemented the algorithm without any extended instructions and found that the performance is limited by serial frozen bit insertion. To boost the MaPU performance in processing polar encoding, several new instructions are proposed here, as shown in Table II.

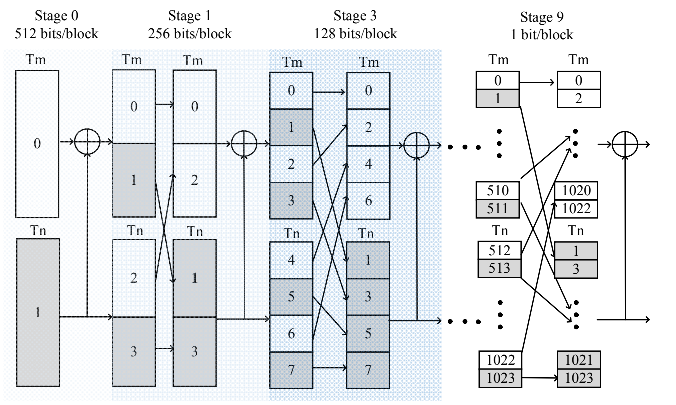

Fig.8 Scheme of butterfly XOR network: In each stage, the even blocks are extracted into the top half operand and the odd blocks are extracted into the bottom half operand. The blocks in the top half will then XOR with the bottom half to generate the top half input of the next stage, whereas the bottom half input is just a copy of the bottom half blocks.

TABLE II. EXTENDED INSTRUCTIONS IN MAPU

Instruction Memo	Operations	Inputs
CRS	Concatenate and Right Shift	Tm, Tn, Tk
BitExpd	Bit Expansion	Tm, Tn
IND	Cross-Border Indexing	Tm, Tn, Tk
StepExt	Interval Step Extraction	Tm,Tn

- Concatenate and Right Shift Instruction (CRS)

This instruction is designed for Concatenate and Right Shift operation. The specific behavior is shown in Fig.7 (b). Each byte of Tn and Tm is concatenated and right-shifted. The degree of the right shift is specified by the lower three bits of each byte in Tk. The lower byte is the result.

- Bit Expansion Instruction (BitExpd)

This instruction is designed for the *Insertion in Byte* operation, whose specific behavior is shown in Fig.7 (c). In *BitExpd*, the data in *Tm* are inserted with zeroes according to the mask in *Tn*. The operation is similar to *Algorithm 1*, in which N is 8. The hardware structure is shown in Fig.9.

- Cross-Border Indexing (IND)

The indices of the original shuffle unit described in Fig.6 only support 512-bit input. However, the maximum length of the polar code in the 5G standard is 1024, and parallel implementation of Fig.7 (a) and Fig.8 shows 1024 bits of interleaving. Taking these factors into account, we extended the original shuffle unit. This extended instruction is the same with Fig.6, except that the inputs are two *T* registers that are concatenated to form a 128-byte vector instead of a single 64-byte *T* register.

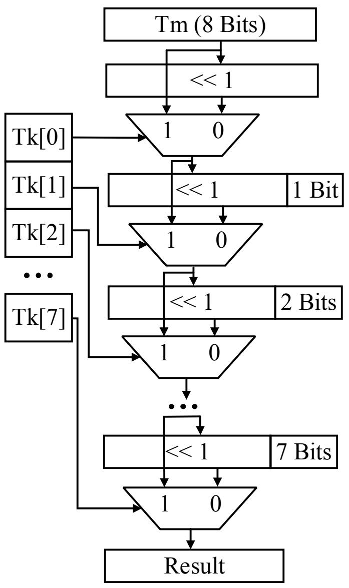

Fig.9 Hardware structure of bit expansion instruction. *Algorithm 1* (when N is 8) is its behavior description. There is an 8-level operation in this instruction implementation. Each level of operation is controlled by 1 bit of Tk orderly. If the corresponding bit of Tk is “0”, the substring left shift. If the corresponding bit of Tk is “1”, the string remains unchanged.

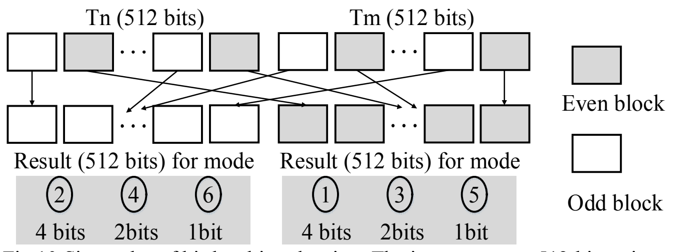

Fig.10 Six modes of bit-level interleaving: The inputs are two 512-bit registers Tm and Tn. For mode {1, 2}, {3, 4}, {5, 6}, the 1024-bit input are divided into 4, 2, 1 bits blocks respectively. Shadowed blocks are even blocks, and the others are odd blocks. For mode {1, 3, 5}, the results are even blocks from the input; for mode {2, 4, 6} the results are odd blocks from the input.

- Interval Step Extraction (StepExt)

This StepExt instruction is much the same as the aforementioned IND instruction, except that StepExt is designed for bit interleaving operation. As depicted in Fig.6, the bits in a block at the earlier stages are more than 8, and we can use IND instruction for interleaving. At later stages when the bits in a block are less than 8, we use StepExt instruction for interleaving. StepExt supports 3 bit-level granularities (1, 2, 4 bits) and two block types (even and odd). Combined with the granularity and block types, StepExt supports six interleaving modes, as shown in Fig.10.

The FUs used in polar encoding are shown in Fig.11. They include integer ALU (IALU), integer and float ALU (IFALU), integer MAC (IMAC), three bus interface units (BIUs), and three shuffle units. To accommodate parallel XOR operations in the polar encoding, IMAC is augmented with XOR instruction. The input data and the parameters (L, H and M vectors) are stored in the local memory. BIU is responsible for accessing the memory. The dashed box on the left shows the pipeline for frozen bit insertion, and that on the right shows the pipeline for butterfly XOR.

V. Result and Discussion

In this section, the performance of the new instructions is compared with that of the original instructions and with other works. The hardware overhead is also analyzed.

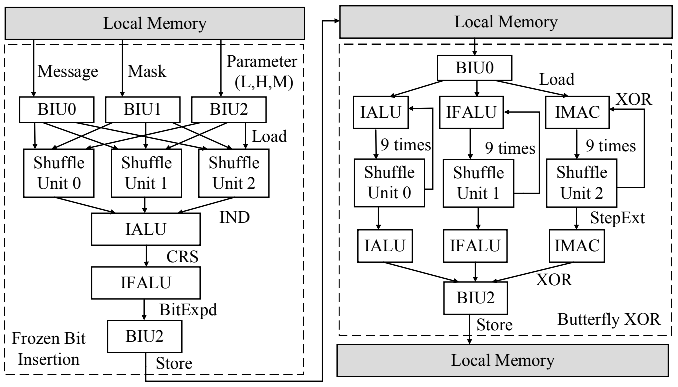

Fig. 11 **FU cascading for parallel polar encoding**: All FUs run in parallel with 512-bit SIMD support. To increase throughput, all FUs are pipelined and run at **1.4 GHz**.

Because the longest code length in the 5G communication standard is 1024 [11-12], only 128 and 1024 bits are discussed and compared.

A. Performance Comparison

To evaluate the performance with the extended instructions, we augmented our original MaPU tool chain, coded them in RTL, implemented the polar encoding algorithm in assembly code, and then ran the simulation. The DSP with extended instruction was pushed through standard chip implementation flow in which the resulting circuit can run at **1.4 GHz** with TSMC (Taiwan Semiconductor Manufacturing Company) 16-nm nodes.

The simulation results are compared with the FPGA implementation of [13] and the original MaPU [8] without extended instructions. The existing polar encoding implementation [13] **meets the 5G protocol and agreement** [9-12]. It has been implemented on Kintex-7 (XC7K325T-2FFG900C) Xilinx FPGAs at 308 MHz. We attempted to compare the results with other works, but either it uses systematic polar encoding [3], or it optimized only for long bit streams in which the performance of a short bit stream is not directly available [3-4]. Other studies [5-7] focused only on hardware resources without providing any data on performance.

As shown in Table III, the performance of our work is improved by almost 300× over that in the literature [13] and by 10× over that with the original MaPU architecture without extended instructions.

The performance gain benefits from three aspects; the first is the parallel encoding algorithm with a wide SIMD. With pre-computed parameters as described in Section III, we actually converted the intrinsic serial encoding to fully parallel. The MaPU architecture supports 512-bit SIMD, so it can take full advantage of these parallel operations. With a wide SIMD, the input data (≤1024 bits) can be completely placed in registers, and multiple FUs can run in parallel to increase the overall throughput. For example, Fig.11 shows that three shuffle units can work in parallel to output 3×512 bits result in one clock cycle.

Second, as we break down the long bit stream operations into parallel byte operations, the circuit can run at a high frequency with pipelines. While supporting other complexed instructions, the augmented FUs still can run at 1.4 GHz without a customized circuit.

TABLE III. Throughput of Different Schemes

Code Length (bits)	**Scheme**	**Performance (Gbps)**	**Speedup**
128	Polaran [13]	0.44	**1**×
	MaPU **Without** Extended Instructions	13.57	**31**×
	MaPU **With** Extended Instructions	134.88	**307**×
1024	Polaran [13]	0.36	**1**×
	MaPU **Without** Extended Instructions	11.92	**33**×
	MaPU **With** Extended Instructions	105.33	**293**×

TABLE IV. THE RELATIVE AREA AND POWER OVERHEAD OF EACH FU

	Area (um^2)				**Power (mW)**			
	IALU	***IFALU***	***Shuffle unit***	***Total***	***IALU***	***IFALU***	***Shuffle unit***	***Total***
Orignial FUs	61,882	72,098	63,686	325,039	20.602	30.583	28.953	138.044
FUs with new instructions	65,420	76,252	66,294	340,556	21.512	31.956	30.032	143.564
Absolute overhead	3,538	4,154	2,608	15,517	0.910	1.373	1.079	5.52
Relative overhead	5.72%	5.76%	4.09%	4.78%	4.42%	4.49%	3.72%	3.99%

Third, the extended instructions also contribute to the performance improvement. As we can see in Table III, with the extended instructions, the throughput increased by 10×.

We believe that our parallel polar encoding is not limited to MaPU architecture or DSP architecture. If the algorithm is implemented in ASICs, it can run at an even higher frequency with less power consumption.

B. *Overhead Analysis*

Table IV shows the area and power overhead of the three FUs. The result is evaluated after synthesizing with Design Compiler of Synopsys, with a 16-nm logic library. Because the augmented FUs heavily reuse existing logic resources, we can see that the overhead with the extended instruction is negligible, only 4.78% for area and 3.99% for power.

VI. CONCLUSIONS

In this paper, we present parallel polar encoding with remarkable performance for the 5G communication standard. We also propose DSP instructions that can be efficiently implemented in the proposed encoding algorithm. The performance and the hardware implementation are evaluated at a detailed circuit level, which showed up to 300× performance improvement compared with the existing FPGA implementation with negligible overhead.

Although we only evaluated the parallel polar encoding algorithm with MaPU architecture, the encoding scheme and extended instructions are applicable to other SIMD architectures.

MaPU architecture [8] is flexible and extensible with fully open-sourced tool chains (https://github.com/mapu/toolchains). In future studies, more instructions can be extended to optimize other algorithms like polar decoding and LDPC encoding/decoding.

ACKNOWLEDGMENT

This work is supported by the Strategic Priority Research Program of Chinese Academy of Sciences (under Grant XDA-06010402) and Youth Innovation Promotion Association of Chinese Academy of Sciences (Y5S7061G51). We also thank Dr. Hongyu Meng, Yinping Jiang, and Jing Feng for useful discussions.

REFERENCES

[1] Arikan, Erdal. "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels." IEEE Transactions on Information Theory 55.7(2008):3051-3073.

[2] 3GPP TSG RAN WG1 #87 Meeting MMC. Final Report of 3GPP TSG RAN WG1 #87 v1.0.0 [DB/OL]. [2017-6-30]. http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_87/Report/

[3] Sarkis, Gabi, et al. "Flexible and Low-Complexity Encoding and Decoding of Systematic Polar Codes." IEEE Transactions on Communications 64.7(2015):2732-27

[4] Yoo, Hoyoung, and I. C. Park. "Partially Parallel Encoder Architecture for Long Polar Codes." IEEE Transactions on Circuits & Systems II Express Briefs 62.3(2015):306-310.

[5] Raj, U. Mahendra Narasimha, and E. V. Narayana. "An advanced architecture with low complexity of partially parallel polar encoder." International Conference on Communication and Electronics Systems IEEE, 2017:1-5.

[6] Arpure, Alok, and S. Gugulothu. "FPGA implementation of polar code based encoder architecture." International Conference on Communication and Signal Processing IEEE, 2016:0691-0695.

[7] Zhang, Chuan, et al. "Pipelined implementations of polar encoder and feed-back part for SC polar decoder." IEEE International Symposium on Circuits and Systems IEEE, 2015:3032-3035.

[8] Wang, Donglin, et al. "MaPU: A novel mathematical computing architecture." IEEE International Symposium on High Performance Computer Architecture IEEE, 2016:457-468.

[9] 3GPP TSG RAN WG1 #89 Meeting MMC. Final Report of 3GPP TSG RAN WG1 #88bis v1.0.0 [DB/OL]. [2017-6-30]. http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_88b/Report/

[10] Huawei, HiSilicon. Polar Coding Design for Control Channel [DB/OL]. [2017-6-30]. http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_88b/Docs/

[11] 3GPP TSG RAN WG1 #88 Meeting MMC. Final Report of 3GPP TSG RAN WG1 #88 v1.0.0 [DB/OL]. [2017-6-30]. http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_88/Report/

[12] Samsung. Maximum Polar Code Size [DB/OL]. [2017-6-30]. http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_88/Docs/

[13] Polaran. Polar Encoder [EB/OL], [2017-6-21]. http://polaran.com/documents/PB-PE-NE-1.0.pdf

[14] Du, Juan, X. Fan, and S. Feng. "Kronecker product of special matrices." Journal of Sichuan Normal University 32.1(2009):56-59.