3GPP Polar Code Bit Allocation Table Generation

XAPP1337 (v1.0) November 14, 2018

Summary

Polar code has been adopted by 3rd Generation Partnership Project (3GPP) for broadcast and control information-forward error correction in 5G wireless systems. This application note describes an equivalent algorithm to generate the bit allocation table, which can then be implemented in C programming language and compiled into RTL by the Vivado® High-Level Synthesis (HLS) tools. The RTL can be integrated with Xilinx® Polar Encoder/Decoder IP to support all 3GPP codes that meet stringent throughput and latency requirements. For more information, refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) and *Polar Encoder/Decoder LogiCORE IP Product Guide* (PG280) (registration required).

Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

The adoption of polar codes by 3GPP provides a channel for broadcast and control information coding in 5G wireless systems. The Rel-15 standard, published in June 2018, defined the details of the Polar code, with other communication protocols using Polar code on the horizon.

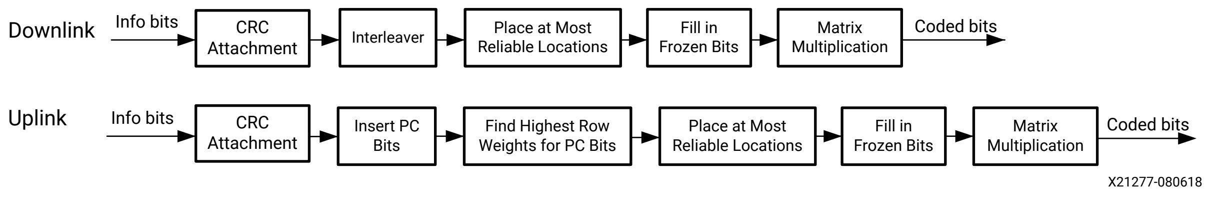
Three types of channels can use Polar codes of different configurations: BCH, DCI, UCI. In most cases, the interleaver is only enabled in the downlink while the parity check (PC) bits only exist in the uplink. To avoid unnecessary logic, the implementations of downlink and uplink are often separated while sharing the same architecture.

Table 1: **Polar Code Configurations for Channel Types**

Channel	CRC Length	Interleave	Npc	Npc_wm	Nmax	E
BCH	24	Enabled	0	0	512	864
DCI	24	Enabled	0	0	512	Variable up to 8192
UCI	{6, 11}	Disabled	{0, 3}	{0, 1}	1024	Variable up to 8192

As defined in 3GPP Standard and shown in the following figure, each block of Polar coded bits consists of information, CRC, parity check, and frozen bits that are interleaved, punctured, and sorted in multiple operations that should be performed serially in multiple iterations, and result in prolonged latency and reduced throughput.

Figure 1: **Polar Code Encoding Procedures**



This application note explains the algorithm for bit allocation table calculation, and proposes an equivalent algorithm that computes the bit allocation table in two iterations. The first iteration requires random access to the bits while the second outputs the calculation results in natural order. The algorithm can be implemented in C programming language and tested against the MATLAB reference model, which then can be synthesized into RTL by Vivado HLS and integrated with Xilinx Polar Encoder/Decoder IP and run on programmable logic. The implementation results show that high throughput and short latency can be achieved with a small amount of logic resource on UltraScale+™ devices.

Features

The method for generating 3GPP Polar code bit allocation table has the following features:

- An algorithm equivalent to that specified in the standard is developed to determine the types of bits in two iterations and output the results in natural order. This natural order output is critical to minimize processing latency.
- An implementation of the algorithm in C programming language is provided. The C code is easy to maintain and modify for future changes to the standards.
- The C code is validated against the MATLAB reference model for all 3GPP codes, and then the same C code is synthesized into RTL by the Vivado HLS tools. The correctness of the code is assured by C simulations instead of the time-consuming RTL simulations.
- The RTL generated by the Vivado HLS tools has a small footprint and runs at a high clock frequency to achieve short latency and high throughput on Xilinx UltraScale+ products.

Equivalent Algorithm

The algorithm to determine the type of each bit in one Polar code word is described in Section 5.3.1.2 of the 3GPP standard. The following table lists the symbols defined in the 3GPP standard which are reused in this application note.

Table 2: **Symbols Defined in the 3GPP Standard**

Symbol	Definition
A	Number of information bits
L	Number of CRC bits
E	Number of bits after rate-matching
N	Number of coded bits
N_{pc}	Number of parity check (PC) bits
n_{PC}^{wm}	Number of parity check bits with minimum weights.
$Q_0^{N_{max}-1} = \{Q_0^{N_{max}}, Q_1^{N_{max}}, \ldots, Q_{N_{max}-1}^{N_{max}}\}$	The Polar sequence $Q_0^{N_{max}-1}$ is in ascending order of reliability $W(Q_0^{N_{max}}) < W(Q_1^{N_{max}}) < \ldots < W(Q_{N_{max}-1}^{N_{max}})$, where $W(Q_0^{N_{max}})$ denotes the reliability of bit index .

Moreover, the symbols in the following table are defined in this application note to facilitate the description of the equivalent algorithm.

Table 3: **Symbols Used in the Equivalent Algorithm**

Symbol	Definition
$\mathbf{Q}_N(k)$	The Polar sequence $\mathbf{Q}_N(k)$ is in descending order of reliability $W(\mathbf{Q}_N(0)) > W(\mathbf{Q}_N(1)) > \ldots > W(\mathbf{Q}_N(N-1))$, where $W(\mathbf{Q}_N(k))$ denotes the reliability of bit index $\mathbf{Q}_N(k)$.This look-up table can be pre-calculated from $\mathbf{Q}_0^{N_{max}-1}$ and stored in a ROM.
$\mathbf{G}_N(k)$	Log2 of the weight of the *k* th row in the coding matrix of *N*-bit Polar code word
$\mathbf{C}_A^L$	The set of CRC bit locations given the number of information bits (*A*) and the number of CRC bits (*L*).
$\mathbf{F}_E^{A+L}$	The set of bit locations that should be excluded from the set of information and parity bits. These bits will be punctured or shortened in rate matching and should be forced to frozen bits
M(k)	The bit allocation table calculated at the first iteration and further processed at the second iteration to determine CRC bit locations.
i	A counter used in the first iteration for the number of bits that are not going to be punctured or shortened after rate matching.
m	A counter used in the second iteration for the number of bits that can be either information or CRC bits.

Using the computations defined in the previous table, the equivalent algorithm can be described as.

```
%---------------------- FIRST ITERATION STARTS HERE-------------------

Compute N from E and (A+L). Compute $\mathbf{F}_E^{A+L}$ * i = 0; $g_{min}$=0;

For k = 0 to N – 1 loop

    b= $\mathbf{G}_N(k)$

    If $b \in \mathbf{F}_E^{A+L}$ or $i \geq (A + L + N_{PC})$, M (b) = Frozen;

    elseif $i \geq (A + L + N_{PC}^{wm})$, M (b) = Parity;

    else

        M (b) = Info temporarily. It can be info, CRC or PC_wm.

    End if.

    If $b \notin \mathbf{F}_E^{A+L}$ ,

        g= $\mathbf{G}_N(k)$;

        If $g < g_{min}$ and $i < (A + L)$,

            $g = g_{min}$ ; $w = b$;

        End if.

        i = i+1;

    End if.

End loop

    If $N_{PC}^{wm} > 0$ 0, M (w)=Parity; End if.
```

```
%--------------------- SECOND ITERATION STARTS HERE------------------
m = 0;
For k = 0 to N – 1 loop
    If M(k) is Frozen, then it must be a Frozen bit;
    elseif M is Parity, then it must be a Parity bit;
    else
        If $m \in \mathbf{C}_A^L$ then it must be a CRC bit;
        else it must be an information bit;
        Endif
        m ++;
    End if
End loop
```

The above algorithm takes advantage of the pre-stored look-up tables to shorten the processing time. More specifically,

- The number of loops in the first iteration is $N \in \{32,\ 64,\ 128,\ 256,\ 512,\ 1024\}$ instead of N_max = 1024, which reduces the latency of short code words significantly. This requires decomposing the original look-up table $\mathbf{Q}_0^{N_{max}-1}$ into smaller tables, i.e., Q_32, Q_64, ... Q_1024, and the size of memory to store all the decomposed tables is 2048 x 10 bits = 20480 bits.
- For each $N \in \{32,\ 64,\ 128,\ 256,\ 512,\ 1024\}$, there will be an encoding matrix and corresponding array of weights G_N. It is observed that the weights are all power of 2, and it suffices to use 4 bits to store one weight, and it takes 2048 x 4 bits = 8192 bits to store all the weight look-up tables.
- The set $\mathbf{F}_E^{A+L}$ contains the bit locations that are punctured in rate matching. Instead of calculating the set explicitly, it is possible to use a number of comparisons to determine whether a bit location belongs to the set or not. No pre-calculated storage is needed for this set.
- The set $\mathbf{C}_A^L$ contains the interleaved locations of *L* CRC bits where *L* is fixed to 24 in the 3GPP standard when the interleaver is enabled. For each value of *A* from 1 to 140, a set of interleaved locations needs to be calculated from Table 5.3.1.1-1 in the TS38.212 standard. It is observed that the last 17 locations of the interleaver are the same as the original bit sequence, and therefore only 7 locations of the CRC bits need to be stored in the look-up table. The total amount of storage is given by 140 x 7 x 8 bits = 7840 bits.

According to the 3GPP standard, the interleaver is only enabled for downlink, while the parity check bits only exist in uplink. To avoid the unnecessary logic, the downlink and uplink designs are separated into two independent designs sharing the same architecture. They are to be integrated with the Polar encoder and decoder, respectively. For more information, refer to the *Polar Encoder/Decoder LogiCORE IP Product Guide* (PG280) (registration required).

Implementation Details

The algorithm described in Equivalent Algorithm is implemented in C programming language. The same C code can be validated in MATLAB and then synthesized into RTL by Vivado HLS tools to run on programmable logic. Additional implementation details are explained as follows:

- The number of coded bits, *N*, is derived from the input parameters *E* and *K=A+L*, which requires the calculation of $\lceil\log_2 E\rceil$ and $\lceil\log_2 K\rceil$. The ceiling operation is converted to floor by using the equation $\lceil\log_2 x\rceil = \lfloor\log_2(2x-1)\rfloor$, and the C function ceillog2(x) simply locates the first non-zero bit of (2x-1).
- A function J*(n)* is defined in the 3GPP standard to map a bit location n in the encoded bit sequence to the interleaved location J*(n)*. The interleaver writes and reads the data in a rectangular table, column-by-column, and a column-wise permutation leads to changes in the bit locations that are always integer multiples of the number of rows. Because the number of columns is fixed to 32, and the number of encoded bits, N, is a power of 2, the number of rows also must be a power of 2. It means that the offsets between the original and interleaved bit locations can be implemented by a bit shift of the column offset. This is implemented in the C function J*(n)* of the reference design.
- In the first iteration, it needs to be determined whether or not a given bit location will be punctured, i.e., $b \in \mathbf{F}_E^{A+L}$. The set $\mathbf{F}_E^{A+L}$ is not computed explicitly but described by a few thresholds for comparison. Before the iteration, the thresholds are pre-calculated, and then in the loop the following code determines whether the given bit is frozen or not.

```
...

    // set Frozen Bit flag
    flg_t isF =(i==K+Npc)? 1:
                 (setID==0)? 0:
              (setID==1)? (Jbitloc=E):((bitloc<Z)||(Jbitloc<(N-E)));
```

  The variable isF indicates whether the bit location should be forced to a frozen bit or not. i is a counter for the number of bits not in the set $\mathbf{F}_E^{A+L}$, K=A+L is the number of information and CRC bits, Npc is the number of parity check bits. Because the bits are accessed in the descending order of reliability in the 1st iteration, a bit location must be frozen bit after (K +Npc) information, CRC, and parity check bits have been identified. The variable setID is to determine which branch of the IF conditions in Section 5.4.1.1 of TS38.212 corresponds to current parameters. The variable Jbitloc saves the interleaved location of the encoded bit.
- In the second iteration, the main complexity lies in the judgement of whether or not a given bit location is CRC, i.e., $m \in \mathbf{C}_A^L$. This can be implemented using the following C code:

```
...

    // flag whether is an interleaved crc location
    flg_t isCa = (m==BA_TABLE_DL[addr_ilv]);

    // flag whether is a crc location at the end
    flg_t isCb = (m>(A+6));

    // increment addr_ilv
    addr_ilv+= (isCa==1)?1:0;
```

The variable isCa indicates whether the bit location m matches the pre-calculated CRC position addressed by addr_ilv; isCb indicates whether the bit location m is one of the last 17 bits of CRC. If either isCa is true or isCb is true, the bit location m must be a CRC bit. If isCa is true, the CRC position pointer `addr_ilv` must be incremented to point to the next CRC position.

The C code is validated against the MATLAB model, and then the same code is synthesized into RTL by Vivado HLS, which is further tested by C/RTL co-simulations. The RTL is also compiled and implemented with Vivado 2018.2 for Xilinx UltraScale+ MPSoC. The results are summarized in the following table.

Table 4: **Implementation Results on xczu9eg-ffvc900-1l-i**

Description	DL	UL
Fmax	395 MHz	440 MHz
Throughput	> 718K codes per second	> 400K codes per second
Latency	<1096 cycles	< 2151 cycles
LUTs	545	621
FFs	490	494
BRAM18Ks	3	4
DSPs	0	0

Interface Details

Both uplink and downlink designs take one set of input parameters and output the computation results into memory. A list of input and output signals are shown in the following table.

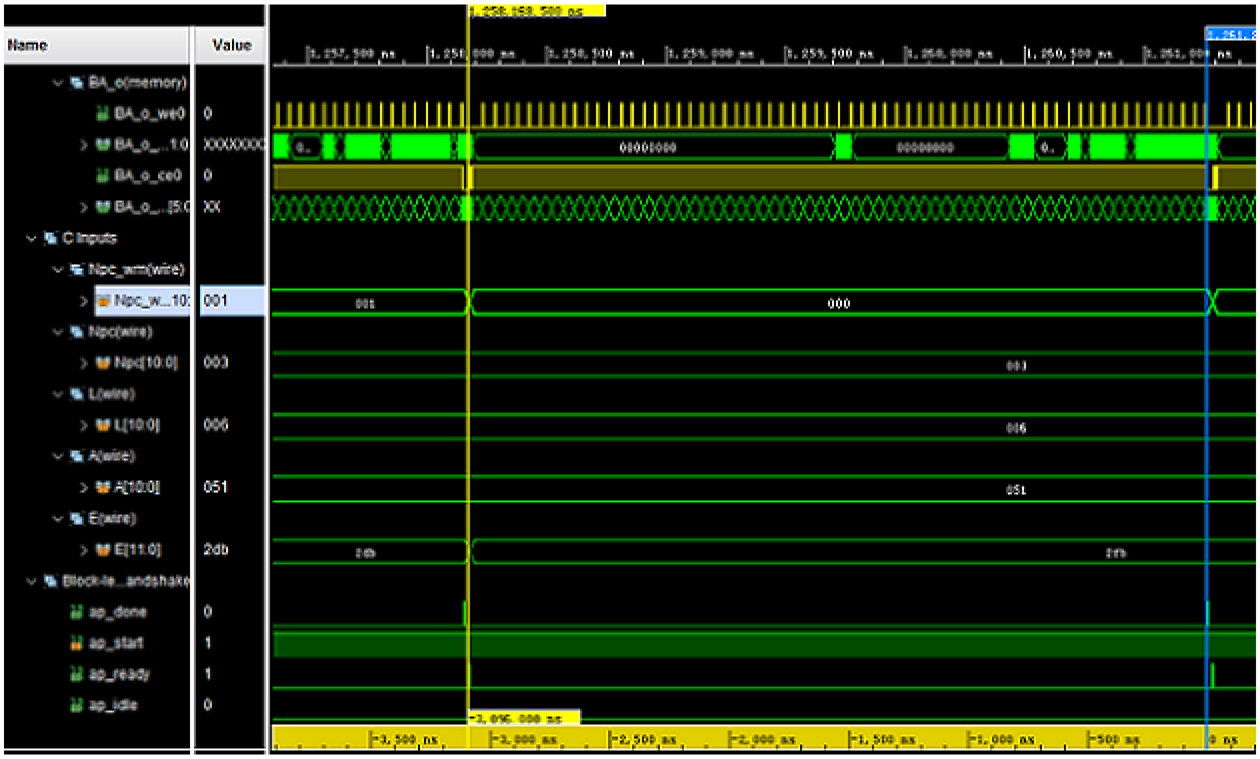
Table 5: **Interface Parameters**

I/O	DL	UL	Description
Input	E	E	Number of bits after rate matching
Input	A	A	Number of information bits before CRC attachment
Input	N/A	L	Number of CRC bits
Input	N/A	Npc	Number of parity check bits
Input	N/A	Npc_wm	Number of PC_wm bits
Output	Nidx	Nidx	Log2 of the number of coded bits
Output	BA[0:31]	BA[0:63]	Bit allocation table

The handshaking signals `ap_start`, `ap_idle`, `ap_done`, and `ap_ready` indicate the state of the module. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on the interface signals.

The following figure shows an exemplary uplink interface waveform generated by the reference design.

Figure 2: **Uplink Interface Waveform Example**



Reference Design

Download the reference design files for this application note from the Xilinx website.

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 6: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Matt Ruan
Target devices	UltraScale+ FPGA, Zynq UltraScale+ MPSoC, Zynq UltraScale+ RFSoC
Source code provided?	Yes
Source code format	MATLAB m file, C code, and TCL script

Table 6: **Reference Design Matrix** *(cont'd)*

Parameter	Description
Design uses code or IP from existing reference design, application note, 3rd party or Vivado software? If yes, list.	No
Simulation	
Functional simulation performed	Yes
Timing simulation performed?	Yes
Test bench provided for functional and timing simulation?	Yes
Test bench format	C code
Simulator software and version	XSIM in Vivado Design Suite 2018.2
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/versions used	Vivado Synthesis
Implementation software tool(s) and version	Vivado Design Suite 2018.2
Static timing analysis performed?	Yes
Hardware Verification	
Hardware verified?	No

Conclusion

3GPP Polar code bit allocation table can be computed by the programmable logic in real time. The simplified equivalent algorithm can be easily described by C programming language and validated against the MATLAB reference model. The same C code is synthesized into RTL by Vivado HLS tools, and hundreds of bit allocation tables can be computed within one millisecond on Xilinx UltraScale+ programmable devices.

References

These documents provide supplemental material useful with this guide:

1. 3GPP TS 38.212, “3GPP Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (Release 15)”, v15.2.0, June 2018.
2. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
3. *Polar Encoder/Decoder LogiCORE IP Product Guide* (PG280) (registration required)
4. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/14/2018 Version 1.0	
Initial Xilinx release.	N/A

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright