An Implementation of the LTE PCCC Encoder for the PicoArray

Doc ID: CTO-TN-0008

Date:	21-01-2007
Version:	v2.00
Status:	Preliminary
Type:	CTO Office Technical Note
Author:	Hao Wang

An Implementation of the LTE PCCC Encoder for the PicoArray

picoChip (Beijing) Technology Co. Ltd.
Room 108, Bldg 10,
ZGC Software Park, Haidian District,
Beijing, 100094
P.R. China
www.picochip.com

Table of Contents

Version History

Version	Date	Author(s)	Reason for Change
V1.00	12-Dec-2007	Hao Wang	1st draft
V1.10	12-Dec-2007	Hao Wang	refined figure 2.
V1.20	13-Dec-2007	Hao Wang	refined figure 3
V1.30	14-Dec-2007	Hao Wang	added a flow chart (Fig. 4)
V1.40	14-Dec-2007	Hao Wang	added result verification using matlab in section 3.
V 1.50	18-Dec-2007	Hao Wang	PN Fifo now has 3 output
V 1.60	20-Dec-2007	Hao Wang	Added title& doc ID on the front page
V 1.70	04-Jan-2008	Hao Wang	Modifications based on the review comments by CS
V 1.80	09-Jan-2008	Hao Wang	Modifications based on the review comments by SJ
V 1.90	15-Jan-2008	Sam Jenkins, Hao Wang	Editorial modifications.
V 2.00	21-Jan-2008	Hao Wang	Editorial modifications.

1 Introduction to picoArray

The picoArray is a multi-processor IC which integrates hundreds of processing elements into a single array. The individual elements have been optimized for signal processing and wireless algorithm computation and control. The result is a general purpose wireless communications processor, capable of executing all contemporary wireless standards, which combines the computational density of a dedicated ASIC with the programmability of a traditional high-end Digital Signal Processor (DSP). Details of picoArray can be found in [3]. For the time being there are two main picoArray DSP products: PC102 and PC20x (PC202, PC203 and PC205).

The PC102 contains four different types of array elements (AEs) which are detailed in Table 1, three of which are programmable and the fourth is a configurable hardware accelerator unit. Minor differences exist between the three programmable AE types (STAN2, MEM2 and CTRL2). These differences include the size of instruction/data memory, additional processing units and instructions supported (e.g. multiply-accumulate, multiply). Each AE can issue a long instruction word (LIW) of up to 64 bits into up to 3 execution units in a single cycle (at 160 MHz). Each AE communicates with other AEs within the array over a bus which is connected to by several ports.

In addition to the STAN2, MEM2 and CTRL2 AE types specified in Table 1, software for the PC102 can also be targeted at the ANY2 AE type implying that: (1) the function does not use any AE-specific instructions and (2) the code and data memory requirements can be met by all AE types.

Software, written in C or ASM, is targeted at an AE type depending on the processing units used and memory required.

Table 1: PC102 processor variants and memory distribution.

Type	Description	Number	Memory (bytes)
STAN2	Standard. A standard AE type includes multiply-accumulate peripheral as well as special instructions optimized for CDMA spread and de-spread. Memory is divided between 512 bytes code and 256 bytes data.	240	768
MEM2	Memory An AE having multiply unit and additional memory. Memory division between code and data is configurable.	64	8,704
FAU	Function Accelerator Unit A co-processor optimized for specific signal processing tasks (FEC, preamble detect, FHT, etc). Includes dedicated hardware for trellis operations.	14	n/a
CTRL2	Control An AE type with a multiply unit and larger amounts of data and instruction memory optimized for the implementation of base station control functionality. Memory division between code and data is configurable.	4	65,536
	Totals per PC102 device:	**322**	**1,003,520**

The picoBus is the name given to the switching fabric running vertically and horizontally between the processing elements in the array. AEs are assigned 32-bit slots on the picoBus at compile time thereby removing the need for arbitration and making performance completely deterministic. Each AE communicates over the picoBus via its ports. These are defined using picoVHDL. Each AE has a number of ports which can be configured to be read (incoming) or write (outgoing). Data sent between AEs is:

1. Written to a write port FIFO by the sending AE.
2. Sent over the picoBus on the next available slot.
3. Read from the read port FIFO by the receiving AE.

By default, communication between AEs is data blocking. On an attempt to read data from the picoBus, an AE will block until data becomes available in the read port FIFO. Similarly, when attempting to write data to the picoBus, the sending AE will block if its write port FIFO is full. A full write port FIFO infers that the receiving AE’s read port is not taking data (i.e., is full itself).

Bandwidth on the picoBus between communicating AEs is assigned via @-rates. A signal is assigned an @-rate which is a positive integer power of 2, e.g., @8, @16. The @-rate is defined in the port declarations in both the sending and receiving AEs. This @-rate is relative to the system clock (160 MHz for the PC102 and

PC20x) and indicates how often data may be sent. For example, @8 means that a 32-bit data value can be sent every 8 cycles (of the 160 MHz bus). The receiving AEs must therefore issue a read (against the associated port) once every 8 cycles in order to prevent the sending AE from blocking.

PC20x and PC102 are similar except that they have different number of AEs and different accelerators. In Table 2 we give a brief overview of PC202, PC203 and PC205.

Table 2: Brief overview of PC202, PC203 and PC205

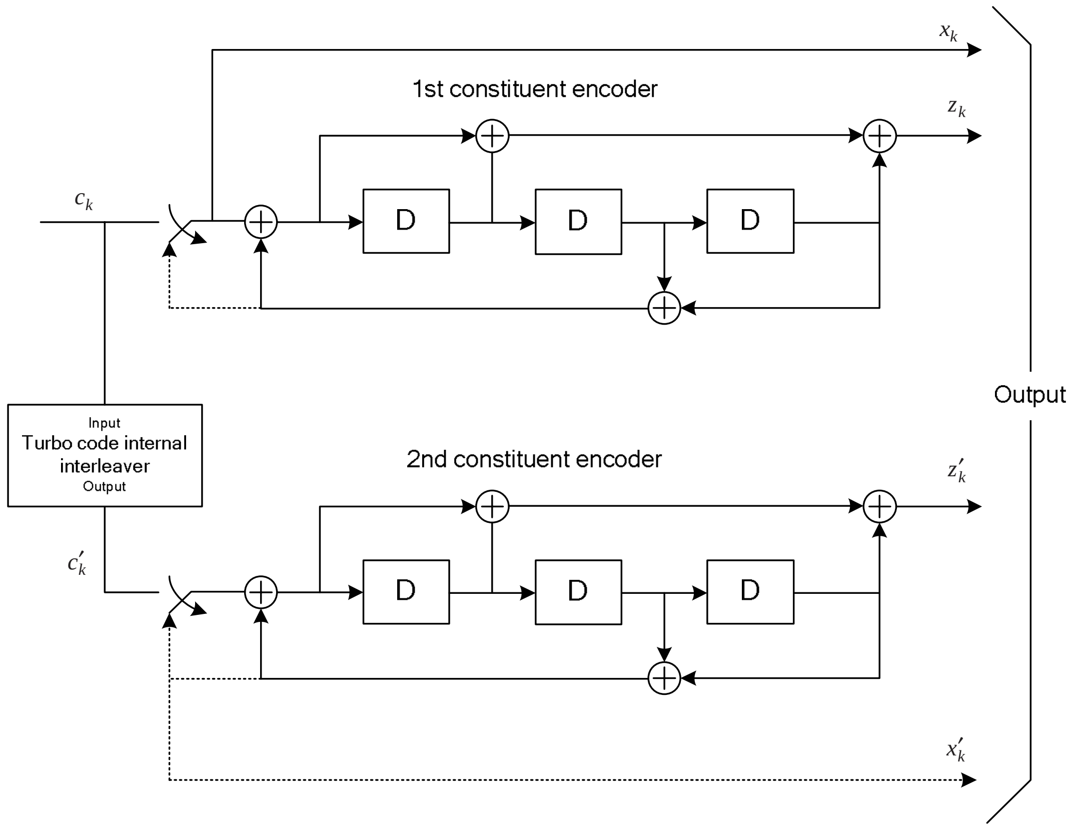
	AE Type	Number of AEs	Memory (bytes)	FAU
PC20x	STAN	196	768	FFT/IFFT Viterbi Turbo decoder, Reed-Solomon decoder, Cryptography accelerator PC202 & PC205 only: ARM9 host & peripherals
	MEM	50	8,704	
	CTRL	2	65,536	
	Total*	248	716,800	

* FAU AEs not included

2 Introduction to Turbo encoder

In this document we demonstrate a design and implementation of the turbo encoder supporting a throughput of 100 Mbps. The structure of rate 1/3 turbo encoder [1] is shown in Figure 1.

Figure 1. Structure of LTE PCCC Turbo Encoder (dotted lines apply for trellis termination only)



The bits input to the turbo encoder are denoted by $c_0, c_1, c_2, \ldots. c_{K-1}$, and the bits output from the first and second constituent encoders are denoted by $z_0, z_1, z_2, \ldots. z_{K-1}$ and $z'_0, z'_1, z'_2, \ldots. z'_{K-1}$ respectively. The bits output from the turbo internal interleaver are denoted by $c'_0, c'_1, c'_2, \ldots. c'_{K-1}$, and these bits are to be the input to the second constituent encoder. Here where K is the block size, taking values from 40 to 6144, according to [1].

The 3 outputs of the turbo encoder d^0 , d^1 , d^2 are:

$$d_k^0 = x_k, \quad d_k^1 = z_k, \quad d_k^2 = z'_k, \quad k = 0,1,\ldots K-1 \qquad (1)$$

The trellis termination bits shall be:

$$d_K^0 = x_K, \quad d_{K+1}^0 = z_{K+1}, \quad d_{K+2}^0 = x'_K, \quad d_{K+3}^0 = z'_{K+1},$$
$$d_K^1 = z_K, \quad d_{K+1}^1 = x_{K+2}, \quad d_{K+2}^1 = z'_K, \quad d_{K+3}^1 = x'_{K+2},$$
$$d_K^2 = x_{K+1}, \quad d_{K+1}^2 = z_{K+2}, \quad d_{K+2}^2 = x'_{K+1}, \quad d_{K+3}^2 = z'_{K+2}, \qquad (2)$$

3 Design of Turbo encoder on PicoArray

The general block diagram of the Turbo encoder is given in Figure 2. The entity "CtrlK" and "PN9Gen" are part of the test bench and those blocks in the dotted square form the structure of the turbo encoder. We give a description of the different entities and modules of in Table 3.

Table 3: Description of the entities in Turbo Encoder

Entity	Description	Type
CtrlK	Read block sizes (K's) from file Kseq.dat and feed the Turbo encoder with a stream of K's. This is an entity for test bench.	STAN
PN9Gen	Generate pseudo random source bits and feed them to the Turbo encoder. This is an entity for test bench	STAN
KFifo1	Provide 3 identical but independent K sequences to different modules of the turbo encoder. This is to prevent the program from blocking since the 1st encoder, 2nd encoder, interleaver and CTCMUX run at different speeds.	MEM
KFifo2	Same as KFifo1	MEM
ProcPN	Make the input bits block to be of size 32x. For example, if K=40, ProcPN will put out 64 bits with the last 24 bits filled with 0's.	STAN
PNFifo	Provide 3 identical but independent input bit sequences to different modules of the turbo encoder. The purpose of doing so is the same as KFifo1 / KFifo2	MEM
1st encoder	1st constituent encoder (Figure 1) which comprises of 6 STANs to support 100 Mbps throughput. Please refer to Figure 3 for details.	group
2nd encoder	2nd constituent encoder (Figure 1) which comprises of 6 STANs to support 100 Mbps throughput. Please refer to Figure 3 for details.	group
interleaver	Turbo internal interleaver. This is not discussed in this document. Please refer to [2] for details.	group
CTC MUX	Get 3 bit stream: x_k, z_k, z'_k and output 3 bit stream d_0, d_1 and d_2. The input-output relationship is given in equation (1) and (2) in section 2. Note that the input bit streams are of size K while the output having size (K+4) because there are 4 trellis termination bits appended.	MEM

Figure 2: Block Diagram of the Turbo Encoder

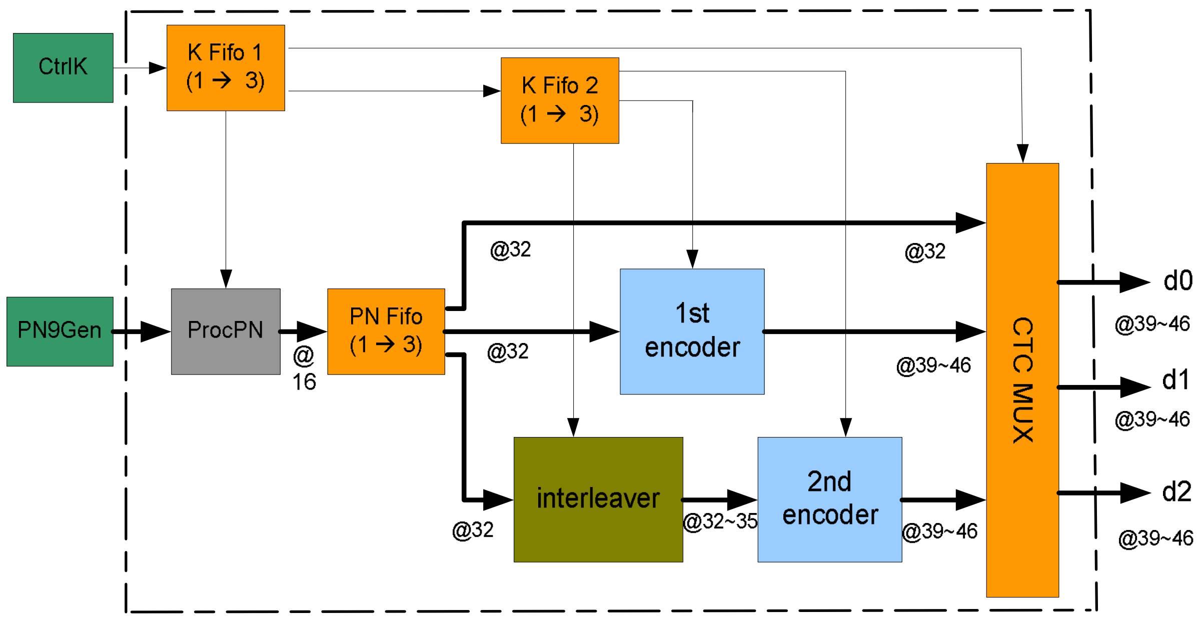
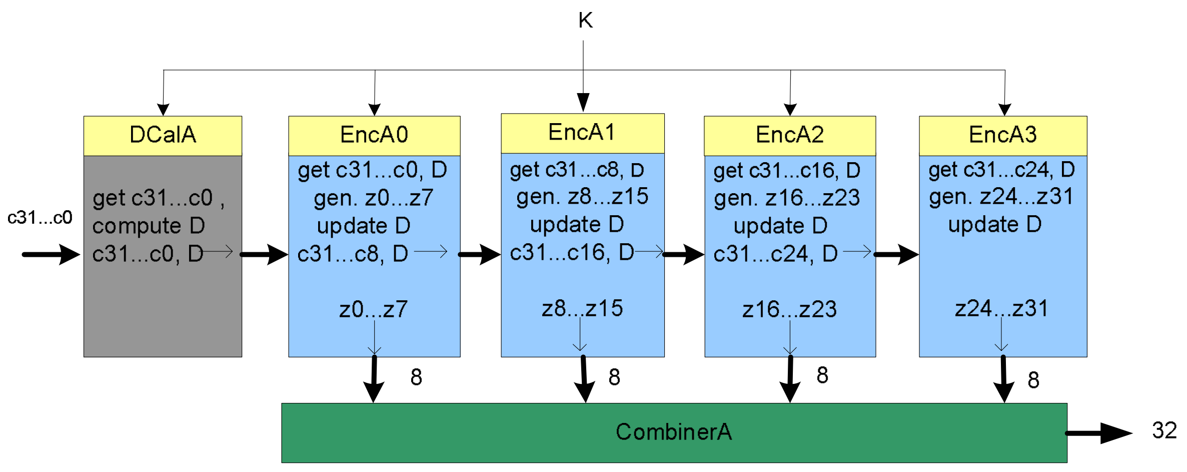


Figure 3 is the block diagram of the 1st constituent encoder. We denote by $c_0, c_1, c_2, \ldots c_{31}$ the 32 input bits and $z_0, z_1, z_2, \ldots z_{31}$ the output bits. We denote the state of the three linear feedback shift registers (LFSRs) by D, i.e., D = [D2 D1 D0]. The 2nd constituent encoder is identical to the 1st encoder in structure except that the input bit stream is the output of the interleaver and the output bits are denoted by $z_0', z_1', z_2', \ldots z_{31}'$.

Figure 3: Block Diagram of the 1st /2nd Encoder



The idea of the structure in Figure 3 is to split the computation of the 32 output bits to 4 individual AEs (EncA0, EncA1, EncA2 and EncA3), each dedicated to its own 8 bit portion. EncA0 generates the first 8 output bits, EncA1 generates the second 8 output bits, and so on. From Figure 1 we see that the input bits and the initial LFSR state are the information needed for generating the corresponding output bits. The last 3 individual encoders can get the D state from its previous encoders and EncA0 gets D from entity “DCalA”.

Assuming $c_0, c_1, c_2, \ldots. c_{31}$ are the 32 input bits, the computation of the 3 LFSRs by DCalA is as follows:

$$D_2 = c_{31} \wedge c_{29} \wedge c_{28} \wedge c_{27} \wedge c_{24} \wedge c_{22} \wedge c_{21} \wedge c_{20} \wedge c_{17} \wedge c_{15} \wedge c_{14} \wedge c_{13} \wedge c_{10} \wedge c_8 \wedge c_7 \wedge c_6 \wedge c_3 \wedge c_1 \wedge c_0 \wedge D_2 \wedge D_0$$

$$D_1 = c_{30} \wedge c_{28} \wedge c_{27} \wedge c_{26} \wedge c_{23} \wedge c_{21} \wedge c_{20} \wedge c_{19} \wedge c_{16} \wedge c_{14} \wedge c_{13} \wedge c_{12} \wedge c_9 \wedge c_7 \wedge c_6 \wedge c_5 \wedge c_2 \wedge c_0 \wedge D_2 \wedge D_1 \wedge D_0$$

$$D_1 = c_{29} \wedge c_{27} \wedge c_{26} \wedge c_{25} \wedge c_{22} \wedge c_{20} \wedge c_{19} \wedge c_{18} \wedge c_{15} \wedge c_{13} \wedge c_{12} \wedge c_{11} \wedge c_8 \wedge c_6 \wedge c_5 \wedge c_4 \wedge c_1 \wedge D_2 \wedge D_1$$

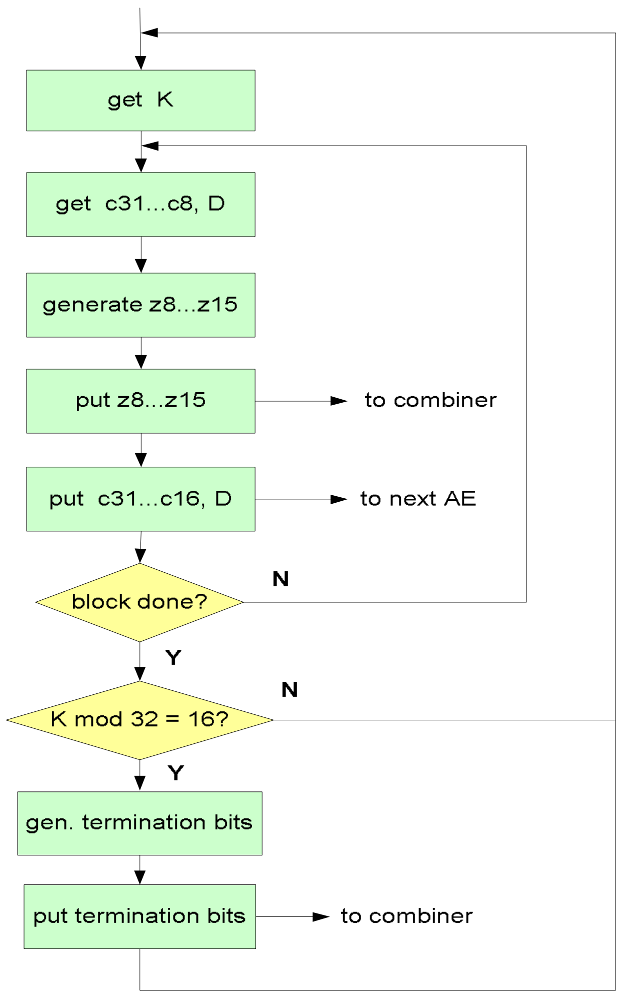
where D_0, D_1, D_2 on the right hand side are the current states and those on the left hand side are the new states after absorbing $c_0, c_1, c_2, \ldots. c_{31}$.

The flow chart of EncA1 (working on bit 8~15) is given in Figure 4 as an example. The other 3 individual encoders follow more or less the same procedure. The 4 individual encoders form a pipeline working in parallel on different 8 bit portions of a 32 bit trunk. In Table 4 we give an example of the working scenario for a 20 byte (160 bit) block. At iteration 5, the four individual encoders are computing byte 17, 14, 11, 8 simultaneously. Once done, the combiner will collect byte 5, 6, 7, 8 and output 32 bits.

Table 4: Timeline of the 4 Individual Encoders for a 160 bit (20 byte) block

iteration #	EncA0	EncA1	EncA2	EncA3	combiner
1	**1**	prev	prev	prev	prev
2	**5**	**2**	prev	prev	prev
3	**9**	**6**	**3**	prev	prev
4	**13**	**10**	**7**	**4**	1,2,3,4
5	**17**	**14**	**11**	**8**	5,6,7,8
6	next	**18**	**15**	**12**	9,10,11,12
7	next	next	**19**	**16**	13,14,15,16
8	next	next	next	**20**	17,18,19,20

Figure 4: Flow Chart of the 2nd Individual Encoder



4 Resources

Resources: 4 MEMs + 12 STANs + Interleaver resources for 100 Mbps

In Table 5 we give the number of AEs required vs. the throughput. If the throughput reduced to 50 Mbps, we can use 2 individual encoders instead of 4, as in Figure 3. Therefore we can save 4 STANS. If the throughput requirement further reduces to 25 Mbps, we can simply use 1 AE to implement the 1st or 2nd encoder. In other words, the 6 AEs in Figure 3 can be replaced by 1 AE only since the throughput requirement is not to onerous. Note that the resources for Turbo internal interleaver [2] are not included in Table 5. The total resources should add the resources demonstrated in [2].

Table 5: Resources Needed For Turbo Encoder (Interleaver excluded)

BW	25 Mbps	50 Mbps	75 Mbps	100 Mbps
# of MEMs	4	4	4	4
# of STANs	3	8	12	12

5 Implementation on PicoArray

Source Code: TurboEncoder.zip

Input file: Kseq.dat
Output file: d0_pico.dat, d1_pico.dat, d2_pico.dat

How to test:

(1) modify the block sizes in "Kseq.dat" to arbitrary valid K's and append a few "0"s to the end. The actual block size sequence will be the K's repetitively. For example, if the content in "Kseq.dat" is 64, 256, 128, 0, 0, the K sequence in simulation will be 64, 256, 128, 64, 256, 128, 64….

(2) make and run "./tstTurboEnc –s ./TCL/speed.tcl" and "./OutputFiles/matlab/verify.m"

(3) compare "./OutputFiles/dx_pico.dat" with "./OutputFiles/dx_mat.dat" for verification.

(4)The throughput data is in ./OutputFiles/Throughput.dat

Throughput: 103 Mbps ~ 131 Mbps except for K = ***40, 48, 56*** and ***72***. (Blocks with size 40, 48 and 56 use the same amount of processing time as K=64, and the resulting throughput is about 5/8, 6/8 and 7/8 of the throughput of K=64. Similarly for K=72 the throughput will be 72/96 of that of K=96. The throughput of these block sizes are around 70Mbps ~ 95 Mbps).

Latency: around 45 μs

6 References

1. 3GPP TS 36.212 V8.1.0, December 2007

2. An Implementation of LTE PCCC Internal Interleaver for picoArray (CTO-TN-0007).

3. picoTools Documentation, V7.2.4, August 16, 2007.

7 Glossary

picoArray™	picoChip Designs Limited proprietary array processing architecture
PCCC	Parallel Concatenated Convolutional Codes
LFSR	Linear Feedback Shift Register
FIFO	First-in-first-out