Xpriori Technical Paper

Optimized Coding Methods for Icon Generation and Manipulation in DPP™

by Chris Brandin

Release 1.1

NOTE: In October 2003, Xpriori, LLC acquired NeoCore Holdings, LLC including all technology and patents. Any references to Neo, NeoCore or NeoCore Holdings, LLC technology or patents as such are now the property of Xpriori, LLC.

Xpriori, LLC
2864 S. Circle Dr.
Ste. 401
Colorado Springs, CO 80906
(719) 425-9840
www.xpriori.com

Version 1.1

Xpriori technology is protected by the following patents:
US Patent #5,742,611 (21 Apr 98)
US Patent #5,942,002 (8 Aug 99)
US Patent #6,157,617 (5 Dec 00)
US Patent #6,167,400 (26 Dec 00)
US Patent #6,324,636 (27 Nov 01)
US Patent #6,493,813 (10 Dec 02)
US Patent #6,792,428 (14 Sept 04)

Other U.S. and international patents pending.

The information in this white paper has been provided by Xpriori, LLC. To the best knowledge of Xpriori, it contains information concerning the current state of information processing technology. Xpriori, LLC disclaims any and all liabilities for and makes no warranties, expressed or implied, with respect to products described in this paper, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose. No specific reliance should be made on the material provided herein without thorough investigation of the technology and its proposed application to specific circumstances. Product and technology information is subject to change without notice.

Introduction

In the NeoCore Technical Paper "Finite Fields and Properties of the NeoCore Icon Generator, Associative Processing Unit, and Associative Memory Controller used in Digital Pattern Processing" (H. Direen Jr. and Keith Phillips), the properties and characteristics of the coding (IG/APU) methods employed in DPP are formally presented in a mathematical context. This paper continues the topic by addressing some of the architectural issues related to implementing coding methods in software and hardware.

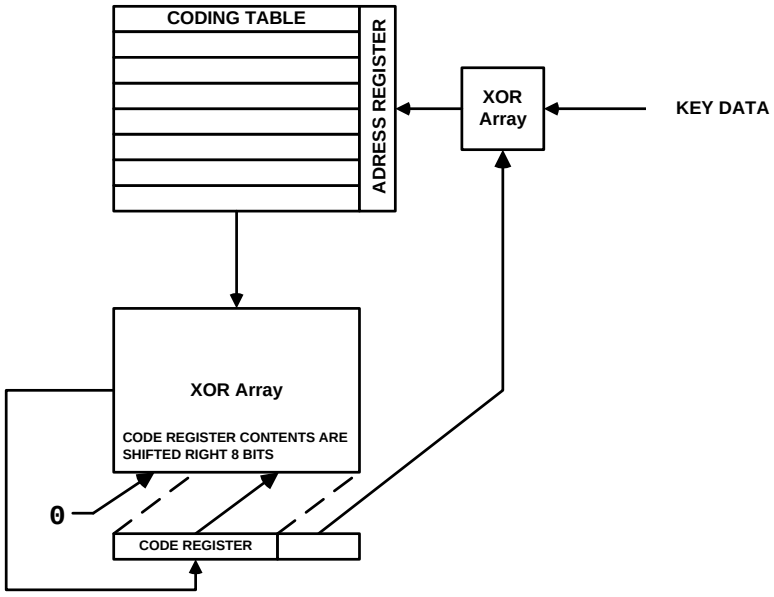
There are a number of well-known methods for generating codes in both bit-serial and word-serial fashions. There are several methods for generating CRC values, for example, including engines that process one bit at a time and "table-driven" engines that process one byte at a time. It is assumed that the reader is familiar with these methods. This paper addresses primarily table-driven (byte at a time) coding mechanisms, although the basic principles can be applied to any sized data quanta. Although the disclosed coding methods can be used to generate CRCs, the codes used in DPP are crafted to achieve different characteristics and goals than generally required of CRC coding.

The primary reason table driven coding mechanisms are employed is to optimize for speed. For example, generating CRC values with tables is generally eight times faster than generating them one bit at a time. Most table-driven coding mechanisms are byte-serial processors. In order to accomplish this, tables are generally 256 entries deep – a very manageable size. Larger tables are usually impractical (to process two bytes at a time, for example, would require 65,536 entries), and smaller ones are usually not necessary.

Digital Pattern Processing requires the accommodation of several functions not typical to most coding applications – mostly to support "Icon Algebra". Some of the methods discussed here are applicable to common coding tasks (like CRC generation), but others are unique to DPP.

Table-driven Code Generators

Probably the best-known coding application is CRC generation. An engine to generate CRC values might be implemented as follows:



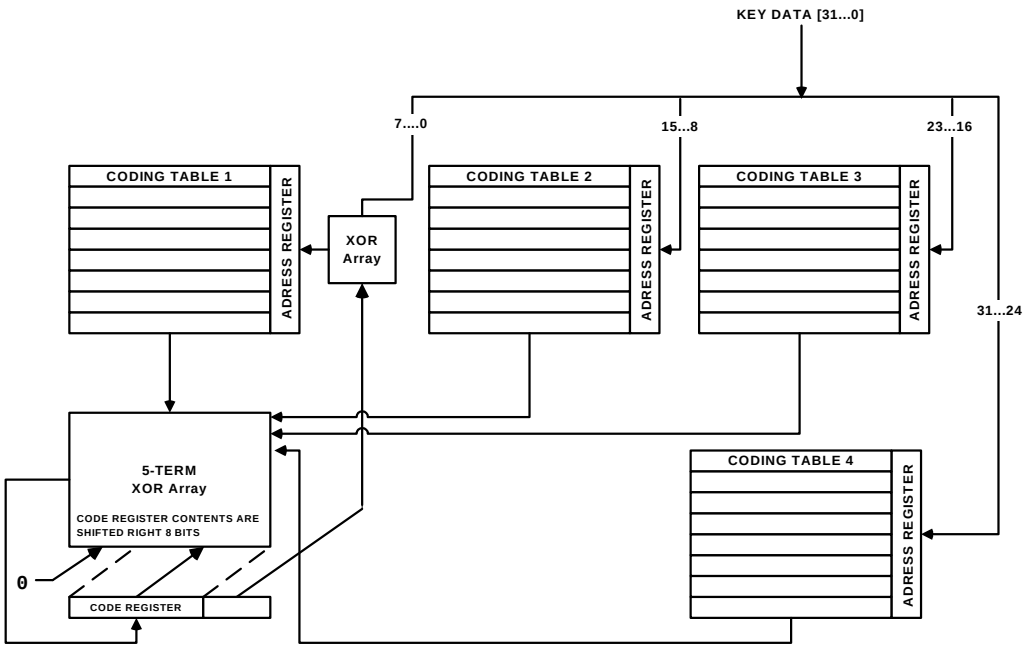
The code generating process is as follows:

1. XOR the data byte to be encoded with the least-significant byte of the contents of the Code Register. This will be used as an address for the Coding Table.

2. Shift the contents of the Code Register right eight bits (shifting zeros into the most-significant bits).

3. XOR the value contained in the Coding Table, pointed to by the address supplied in step 1, with the contents of the Code Register and replace the Code Register contents with the result.

The contents of the coding table are pre-calculated, so there are 256 entries representing every possibility for an 8-bit value. Thus, this engine processes one byte of data at a time. Similarly, a larger table (65,536 entries) could be used to process a 16-bit word of data at a time. In order to be able to process 32-bit data, a table with over 4 billion entries would be required – clearly, this would be impractical. It is possible, however, to build hardware that can process 32-bits at a time by using four 256-entry tables. A diagram of such a device follows:



The trick to making this work is to pre-calculate the table's contents according to the position of each byte of the input data word. If we take an input word of 11223344h, for example, the entry for each table corresponding to entry 11h would be the pre-calculated CRC values for the following data items:

Table 1 11000000h
Table 2 220000h
Table 3 3300h
Table 4 44h

When the pre-calculated CRC values are XOR'ed together, the resulting value is identical to what would be returned had we used a 4-billion entry table. With this implementation, the total table size is 4x256, or 1024 entries.

Pre-calculated Codes and Icon Algebra

So far we have discussed simple code generation. DPP makes extensive use of "Icon Algebra", which requires the ability to un-generate and manipulate coding results. To illustrate this, we will use a simple example – sliding window signature matching. Let us suppose that we want to locate any instance of the following four-letter words in a block of data:

love
cash
cars
home
kids
dogs

cats
food

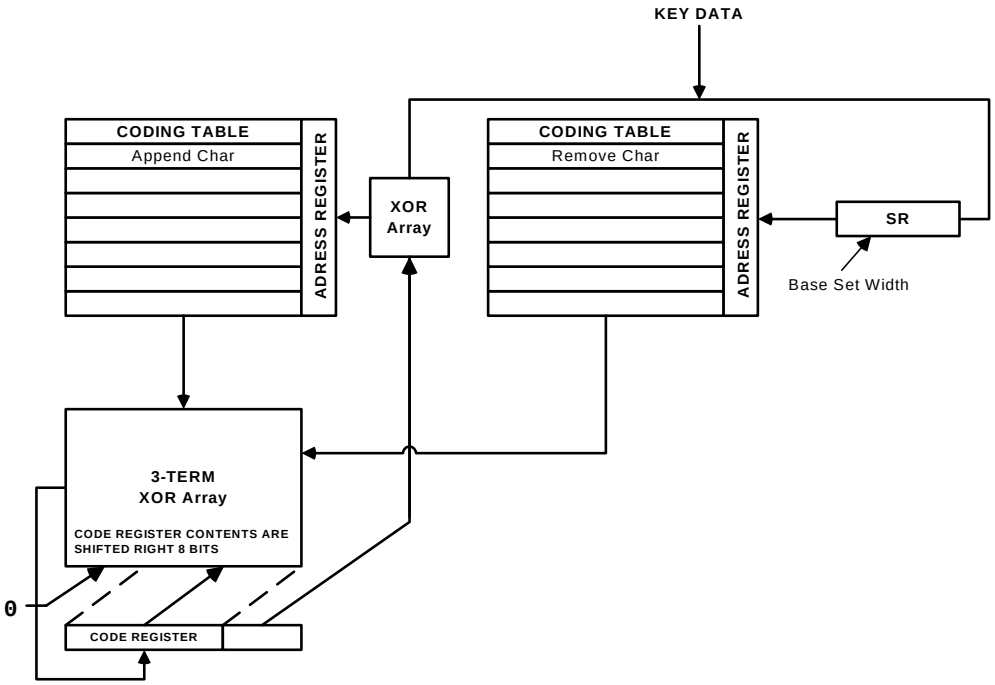
First we create an associative memory Store containing these words. As we slide through the data block, we must create Icons in order to match the data buffer contents against the signature Store. The naïve approach would be as follows:

1. Start at the first byte.
2. Iconize four bytes.
3. Attempt a look-up against the signature Store.
4. Move forward one byte.
5. Continue processing at step 2.

This approach is inefficient because we must Iconize four bytes at every byte location of the data buffer. If the signatures were longer (100 bytes each, for example), performance would degrade, resulting in even less efficiency. Using Icon Algebra, a better approach can be implemented:

1. Start at the first byte.
2. Iconize four bytes.
3. Attempt a look-up against the signature Store.
4. Un-process the first byte (APU function “cut data”).
5. Process the next end-byte (APU function “append data”).
6. Continue processing at step 3.

This is more efficient because, as we slide across the buffer, only two bytes are processed: the one removed from the beginning and the one appended to the end. Thus, we are processing two bytes – rather than four – at each location of the data buffer (a savings of 50%). If the signatures were 100 bytes each, the savings would be 98%. In fact, the processing required as we shift across the data buffer remains flat, no matter how long the signatures are. In order to accomplish this, two coding tables are used. One contains pre-calculated coding values for each of 256 possible values a byte of data can represent. The other contains pre-calculated values that correspond to each of the 256 possible values with as many nulls appended as the signature width (4, in this case). If the coding mechanism is implemented as hardware, steps 4 and 5 above can be collapsed into one operation, resulting in re-doubled efficiency:



By employing multiple coding tables and taps in the shift-register, complex or fragmented signatures can be accommodated. At the relatively low cost of additional tables, search engines can be built that do not degrade in performance as signature complexity increases.

All Icon Algebra functions can be optimized in this manner – append, prepend, remove, cut, and change data. The result is that all Icon Algebra tasks take the same amount of time – one operation. Furthermore, combined operations (such as the append and cut operations in the example above) can be collapsed into one step.

Removing Information From the End of an Icon

So far we have discussed Icon generation and manipulation in the forward direction. One additional circumstance remains to be accommodated – the removal of information from the end of an Icon (reverse transform). This can be accomplished with pre-iconized information (remove an Icon from an Icon), or with key-data (remove data from an Icon). We will discuss the latter case here.

By adding an additional table to the coding engine, we can remove information from the end of an Icon, producing an Icon that represents less information. Careful examination of the basic coding mechanism will reveal that the most-significant byte of an Icon can be used to indicate the most recently used entry from the Coding Table. This is because the most-significant byte portion of the contents of the coding table is a perfect set of entries consisting of one instance of each value ranging from 0 to 255 (in pseudo random order). The added table allows the correct entry to be located by providing sorted (by most-significant byte) pointers into the Coding Table. When the original Icon was produced, the address used to select the Coding Table entry during the coding process was generated by XOR'ing the least-significant byte of the current contents of the Code Register with a byte of data to be encoded. Because we know the data byte we want

to remove, and the most recent Coding Table entry used, we can calculate the previous least-significant byte of the Code Register (the one that was shifted out) by XOR'ing the data byte value (being removed) with the address (normalized to an 8-bit value) of the most recently used Coding Table entry. The steps to remove data from the end of an Icon are as follows:

1. Use the most-significant byte of the Code Register to address the new "Reverse Coding Table". The address of the most recently used Coding Table entry is returned.

2. XOR the most recently used Coding Table entry value with the contents of the Code Register.

3. Shift the contents of the Code Register left eight bits (shifting zeros into the least significant bits).

4. XOR the address returned in step 1 with the data byte to be removed, and OR this value with the contents of the Code Register. This will become the least-significant byte of the Code Register.

Conclusion

By employing multiple coding tables, where tables are loaded with pre-calculated values that take data location into account, processes that normally require multiple operations of varied complexity can be reduced to a single step.

Appendix

Related Papers

Brandin, Chris, "A Definition of Digital pattern Processing™", NeoCore, 2000

Brandin, Chris, "DPP™ Memory Management", NeoCore, 2000

Brandin, Chris, "Three-Dimensional Content Scanning Using DPP™", NeoCore, 2000

Brandin, Chris, "Management of Duplicate Data Elements in DPP™ Virtual Associative Memories", NeoCore, 2000

Brandin, Chris, "Behavioral Set and Field Descriptor Implementations in DPP™", NeoCore, 2000

Direen, Harry and Phillips, Keith, "Finite Fields and Properties of the NeoCore Icon Generator, Associative Processing Unit, and Associative Memory Controller used in Digital Pattern Processing™", NeoCore, 2000